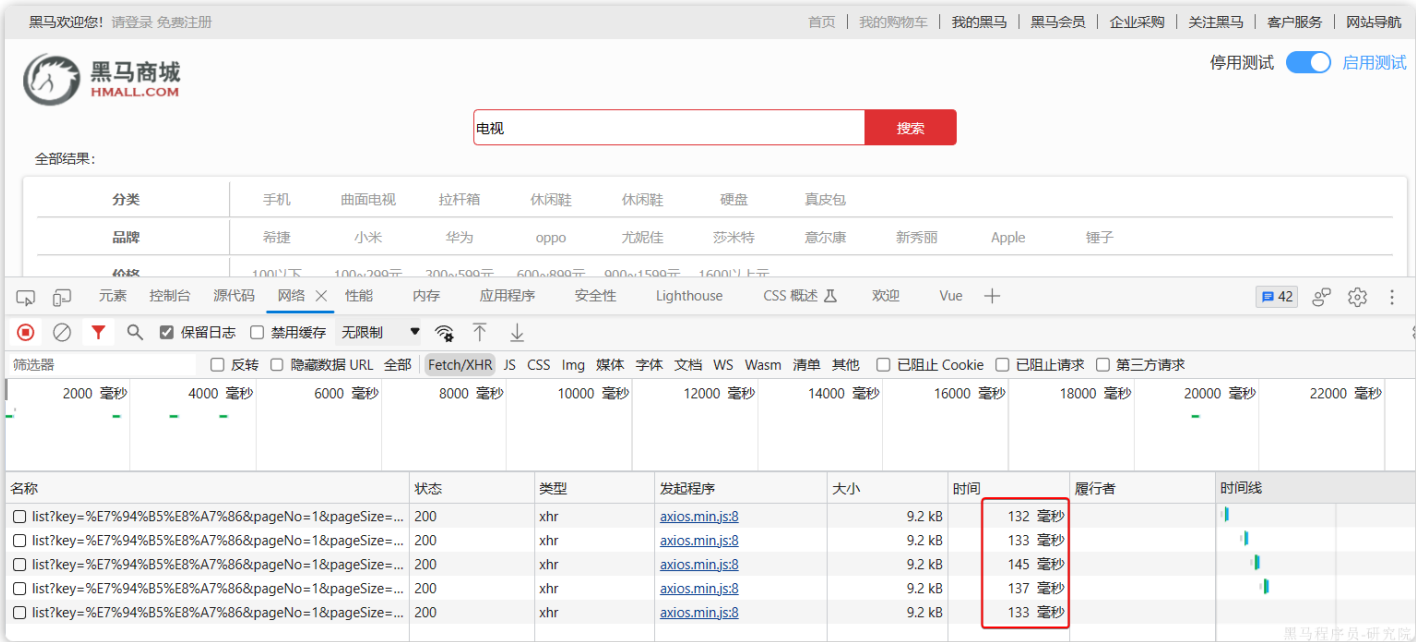


day08-Elasticsearch

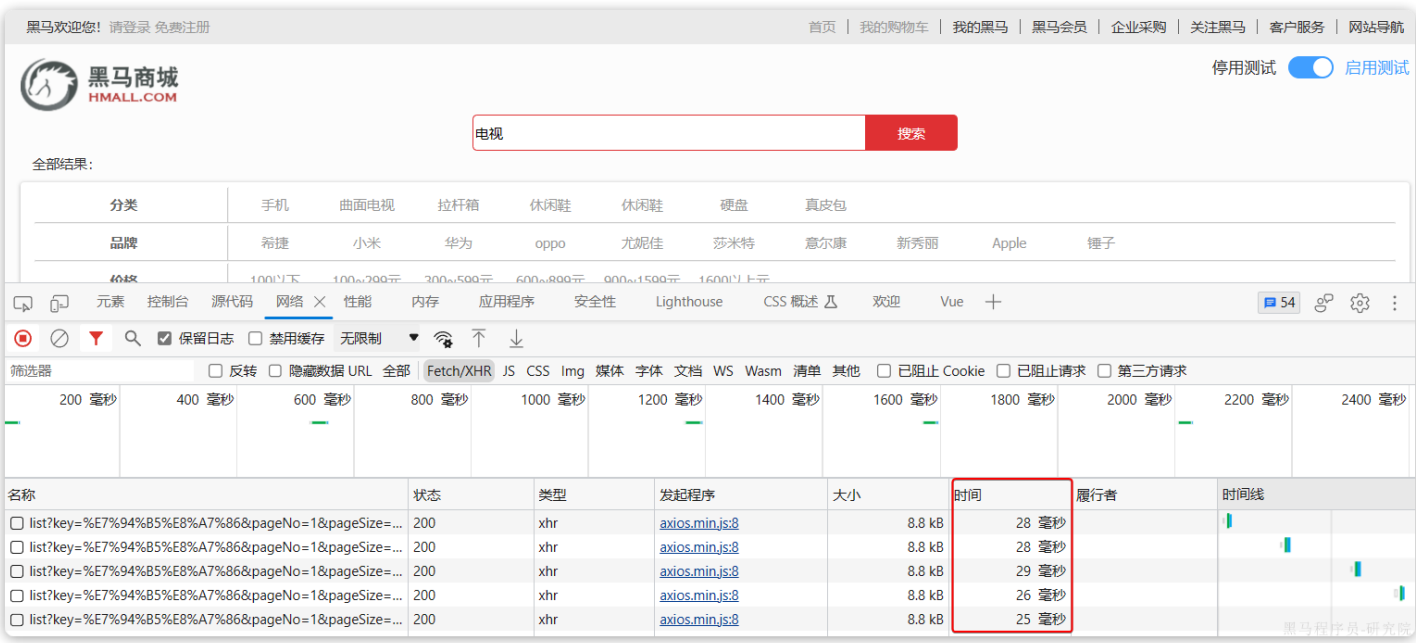
黑马商城作为一个电商项目，商品的搜索肯定是访问频率最高的页面之一。目前搜索功能是基于数据库的模糊搜索来实现的，存在很多问题。

首先，查询效率较低。

由于数据库模糊查询不走索引，在数据量较大的时候，查询性能很差。黑马商城的商品表中仅仅有不到9万条数据，基于数据库查询时，搜索接口的表现如图：



改为基于搜索引擎后，查询表现如下：



需要注意的是，数据库模糊查询随着表数据量的增多，查询性能的下降会非常明显，而搜索引擎的性能则不会随着数据增多而下降太多。目前仅10万不到的数据量差距就如此明显，如果数据量达到百万、千万、甚至上亿级别，这个性能差距会非常夸张。

其次，功能单一

数据库的模糊搜索功能单一，匹配条件非常苛刻，必须恰好包含用户搜索的关键词。而在搜索引擎中，用户输入出现个别错字，或者用拼音搜索、同义词搜索都能正确匹配到数据。

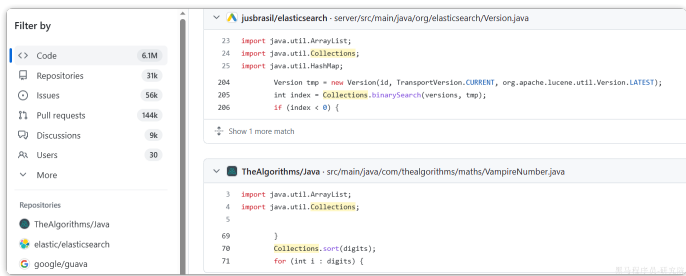
综上，在面临海量数据的搜索，或者有一些复杂搜索需求的时候，推荐使用专门的搜索引擎来实现搜索功能。

目前全球的搜索引擎技术排名如下：

Rank			DBMS	Database Model	Score		
May 2021	Apr 2021	May 2020			May 2021	Apr 2021	May 2020
1.	1.	1.	Elasticsearch	Search engine, Multi-model	155.35	+3.18	+6.23
2.	2.	2.	Splunk	Search engine	92.11	+3.62	+4.36
3.	3.	3.	Solr	Search engine, Multi-model	51.19	+0.59	-1.39
4.	4.	4.	MarkLogic	Multi-model	9.52	-0.40	-1.44
5.	5.	8.	Algolia	Search engine	7.72	-0.15	+3.25
6.	6.	6.	Sphinx	Search engine	7.58	+0.53	+1.55
7.	7.	5.	Microsoft Azure Search	Search engine	6.05	-0.49	-0.07
8.	8.	7.	ArangoDB	Multi-model	4.38	-0.39	-0.30
9.	9.	11.	Virtuoso	Multi-model	3.44	+0.27	+1.09
10.	10.	10.	Amazon CloudSearch	Search engine	2.20	-0.03	-0.39
11.	11.	12.	Xapian	Search engine	0.88	-0.02	+0.13
12.	12.	13.	CrateDB	Multi-model	0.77	+0.02	+0.09
13.	13.	15.	Alibaba Cloud Log Service	Search engine	0.44	-0.01	+0.17

排名第一的就是我们今天要学习的elasticsearch.

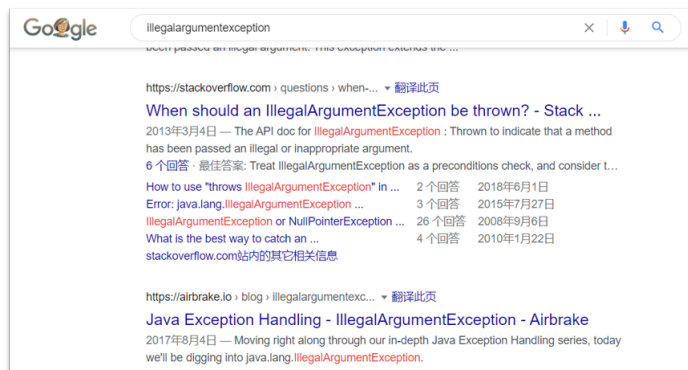
elasticsearch是一款非常强大的开源搜索引擎，支持的功能非常多，例如：



代码搜索



商品搜索



解决方案搜索



地图搜索

通过今天的学习大家要达成下列学习目标：

- 理解倒排索引原理
- 会使用IK分词器
- 理解索引库Mapping映射的属性含义
- 能创建索引库及映射
- 能实现文档的CRUD

1.初识elasticsearch

Elasticsearch的官方网站如下：

<https://www.elastic.co/cn/elasticsearch/>
www.elastic.co

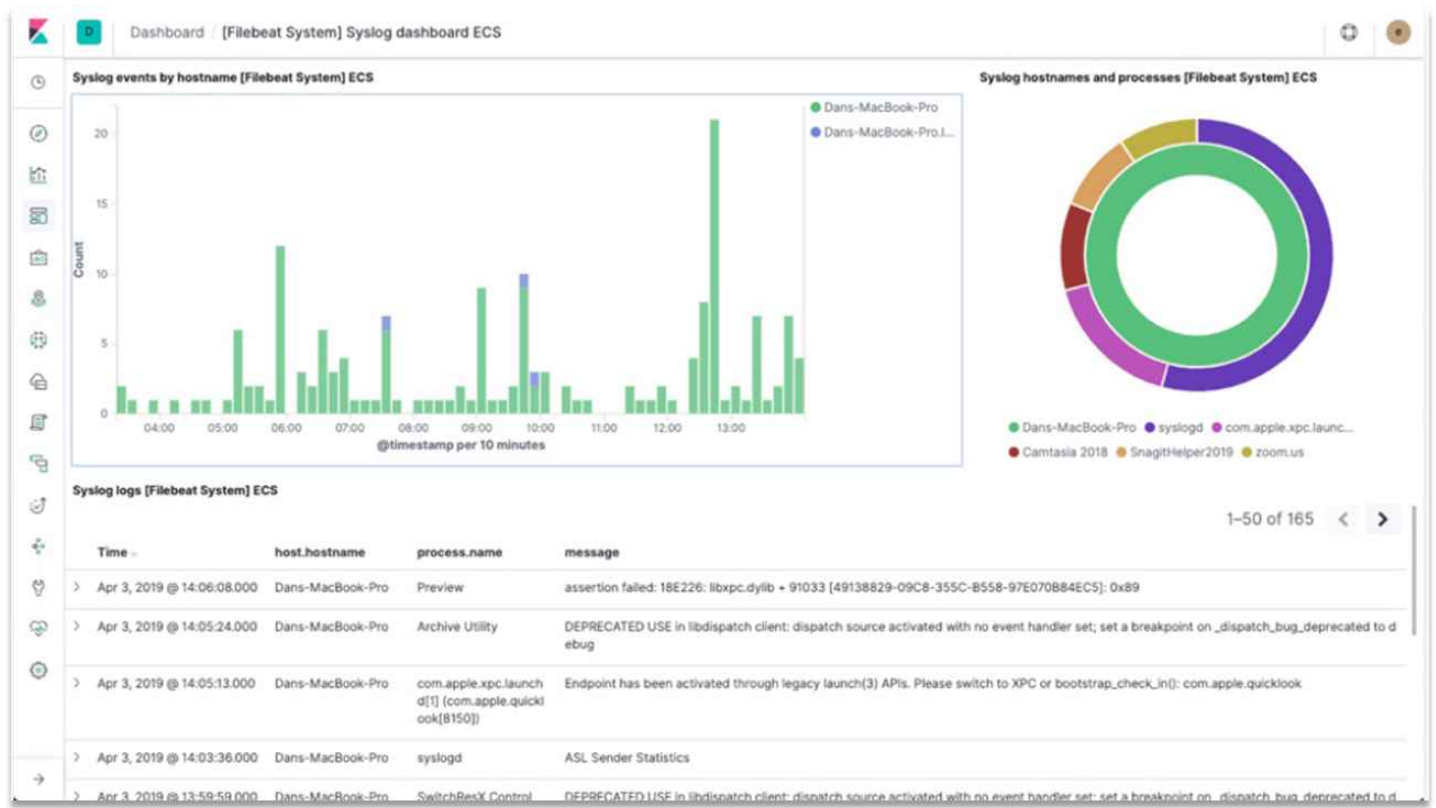
本章我们一起来初步了解一下Elasticsearch的基本原理和一些基础概念。

1.1.认识和安装

Elasticsearch是由elastic公司开发的一套搜索引擎技术，它是elastic技术栈中的一部分。完整的技术栈包括：

- Elasticsearch：用于数据存储、计算和搜索
- Logstash/Beats：用于数据收集
- Kibana：用于数据可视化

整套技术栈被称为ELK，经常用来做日志收集、系统监控和状态分析等等：



整套技术栈的核心就是用来**存储、搜索、计算**的Elasticsearch，因此我们接下来学习的核心也是Elasticsearch。

我们要安装的内容包含2部分：

- elasticsearch：存储、搜索和运算
- kibana：图形化展示

首先Elasticsearch不用多说，是提供核心的数据存储、搜索、分析功能的。

然后是Kibana，Elasticsearch对外提供的是Restful风格的API，任何操作都可以通过发送http请求来完成。不过http请求的方式、路径、还有请求参数的格式都有严格的规范。这些规范我们肯定记不住，因此我们要借助于Kibana这个服务。

Kibana是elastic公司提供的用于操作Elasticsearch的可视化控制台。它的功能非常强大，包括：

- 对Elasticsearch数据的搜索、展示
- 对Elasticsearch数据的统计、聚合，并形成图形化报表、图形
- 对Elasticsearch的集群状态监控
- 它还提供了一个开发控制台（DevTools），在其中对Elasticsearch的Restful的API接口提供了**语法提示**

1.1.1.安装elasticsearch

通过下面的Docker命令即可安装单机版本的elasticsearch：

```
1 docker run -d \  
2   --name es \  
3   -e "ES_JAVA_OPTS=-Xms512m -Xmx512m" \  
4   -e "discovery.type=single-node" \  
5   -v es-data:/usr/share/elasticsearch/data \  
6   -v es-plugins:/usr/share/elasticsearch/plugins \  
7   --privileged \  
8   --network hm-net \  
9   -p 9200:9200 \  
10  -p 9300:9300 \  
11  elasticsearch:7.12.1
```

注意，这里我们采用的是elasticsearch的7.12.1版本，由于8以上版本的JavaAPI变化很大，在企业中应用并不广泛，企业中应用较多的还是8以下的版本。

如果拉取镜像困难，可以直接导入课前资料提供的镜像tar包：

新加卷 (D:) > 课程资料 > 服务框架 > day08-Elasticsearch01 > 资料 >

名称	类型	大小
ik	文件夹	
elasticsearch-analysis-ik-7.12.1.zip	360压缩 ZIP 文件	4,399 KB
es.tar	360压缩	842,453 KB
kibana.tar	360压缩	1,095,137...

黑马程序员-研究院

安装完成后，访问9200端口，即可看到响应的Elasticsearch服务的基本信息：



1.1.2.安装Kibana

通过下面的Docker命令，即可部署Kibana：

```
1 docker run -d \
2 --name kibana \
3 -e ELASTICSEARCH_HOSTS=http://es:9200 \
4 --network=hm-net \
5 -p 5601:5601 \
6 kibana:7.12.1
```

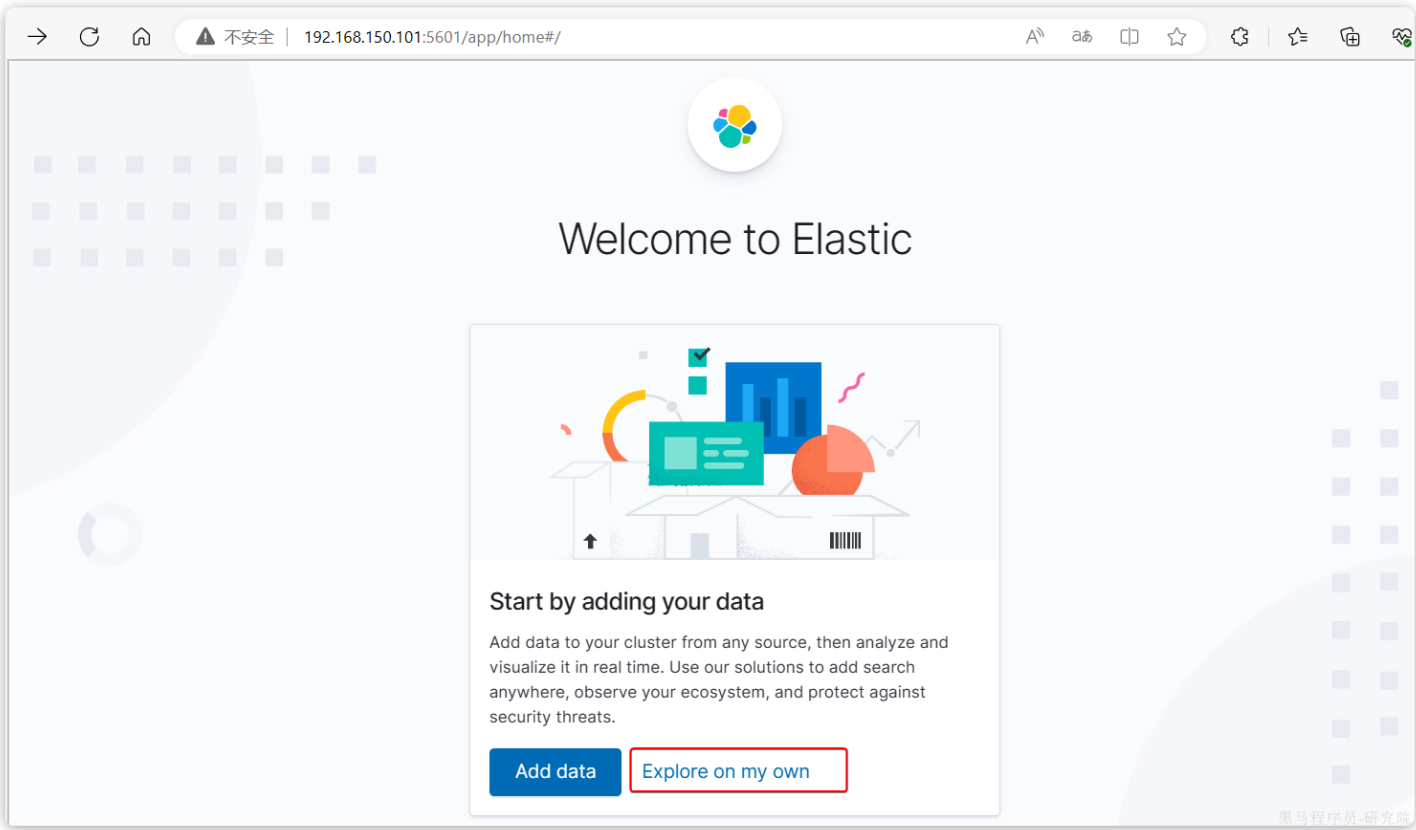
如果拉取镜像困难，可以直接导入课前资料提供的镜像tar包：

新加卷 (D:) > 课程资料 > 服务框架 > day08-Elasticsearch01 > 资料 >

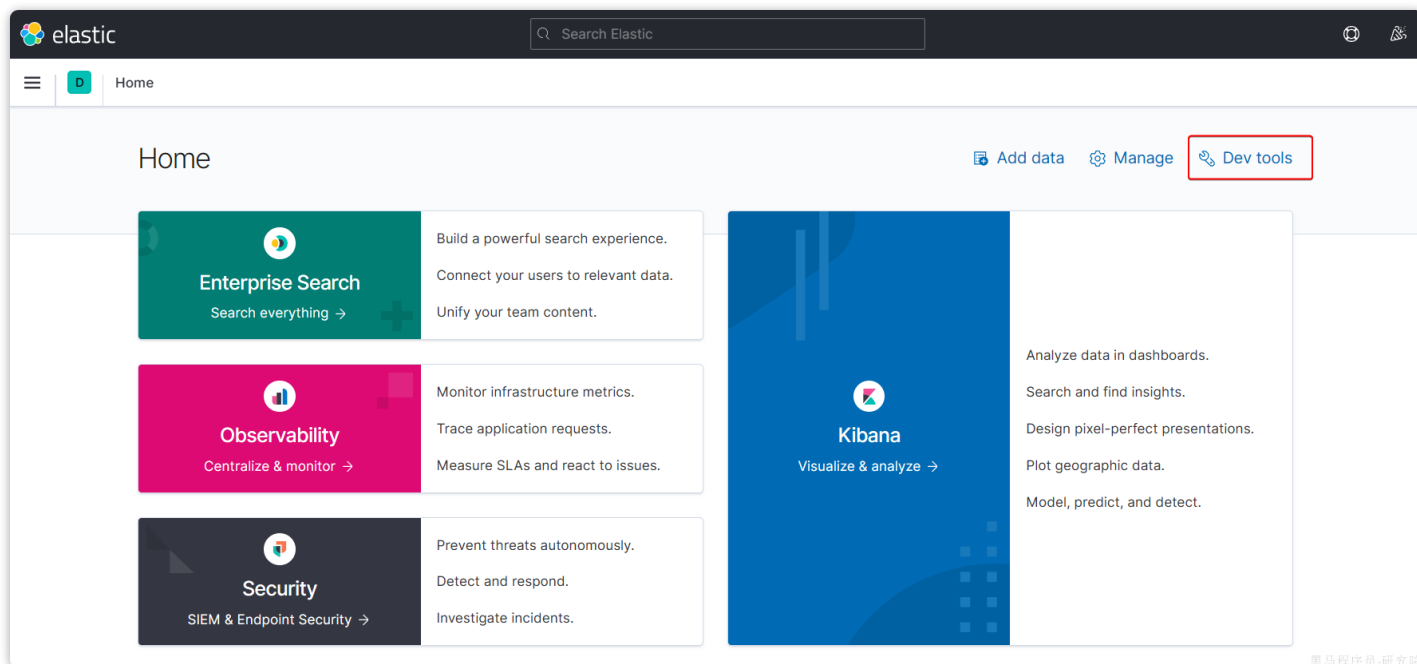
名称	类型	大小
ik	文件夹	
elasticsearch-analysis-ik-7.12.1.zip	360压缩 ZIP 文件	4,399 KB
es.tar	360压缩	842,453 KB
kibana.tar	360压缩	1,095,137...

黑马程序员-研究院

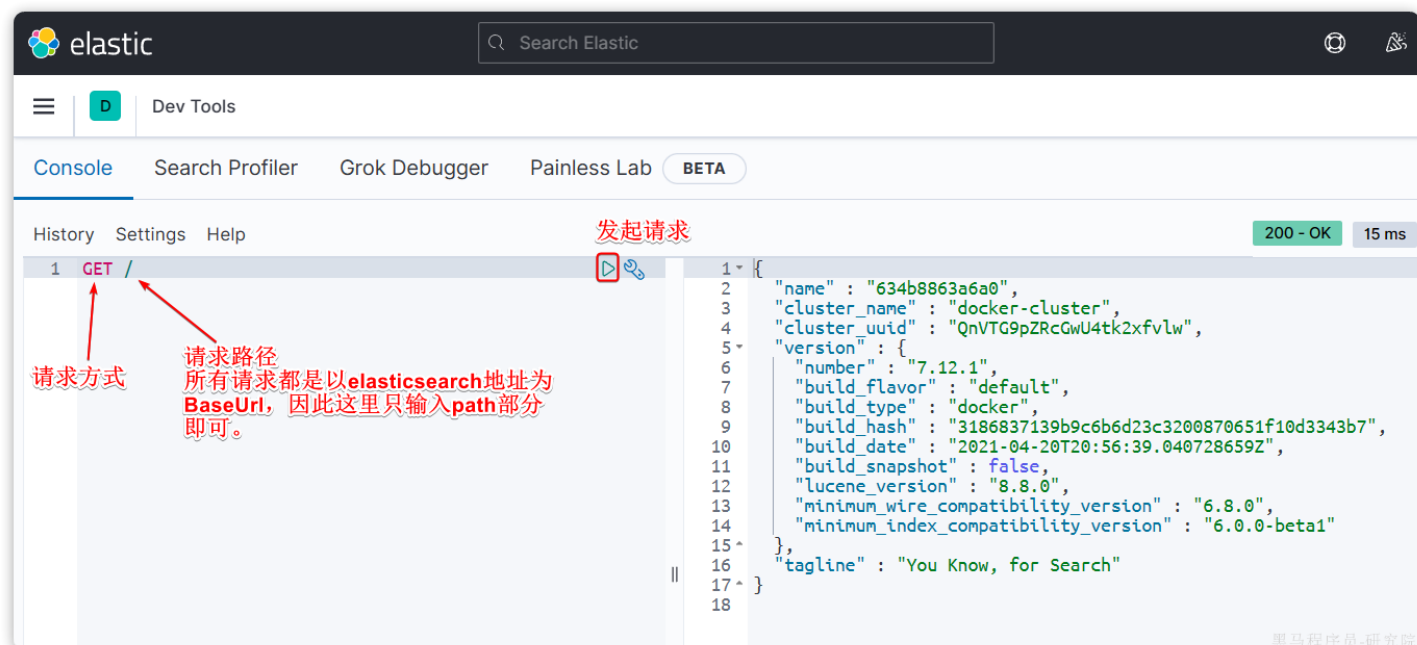
安装完成后，直接访问5601端口，即可看到控制台页面：



选择 `Explore on my own` 之后，进入主页面：



然后选中 **Dev tools**，进入开发工具页面：



1.2.倒排索引

elasticsearch之所以有如此高性能的搜索表现，正是得益于底层的倒排索引技术。那么什么是倒排索引呢？

倒排索引的概念是基于MySQL这样的正向索引而言的。

1.2.1.正向索引

我们先来回顾一下正向索引。

例如有一张名为 **tb_goods** 的表：

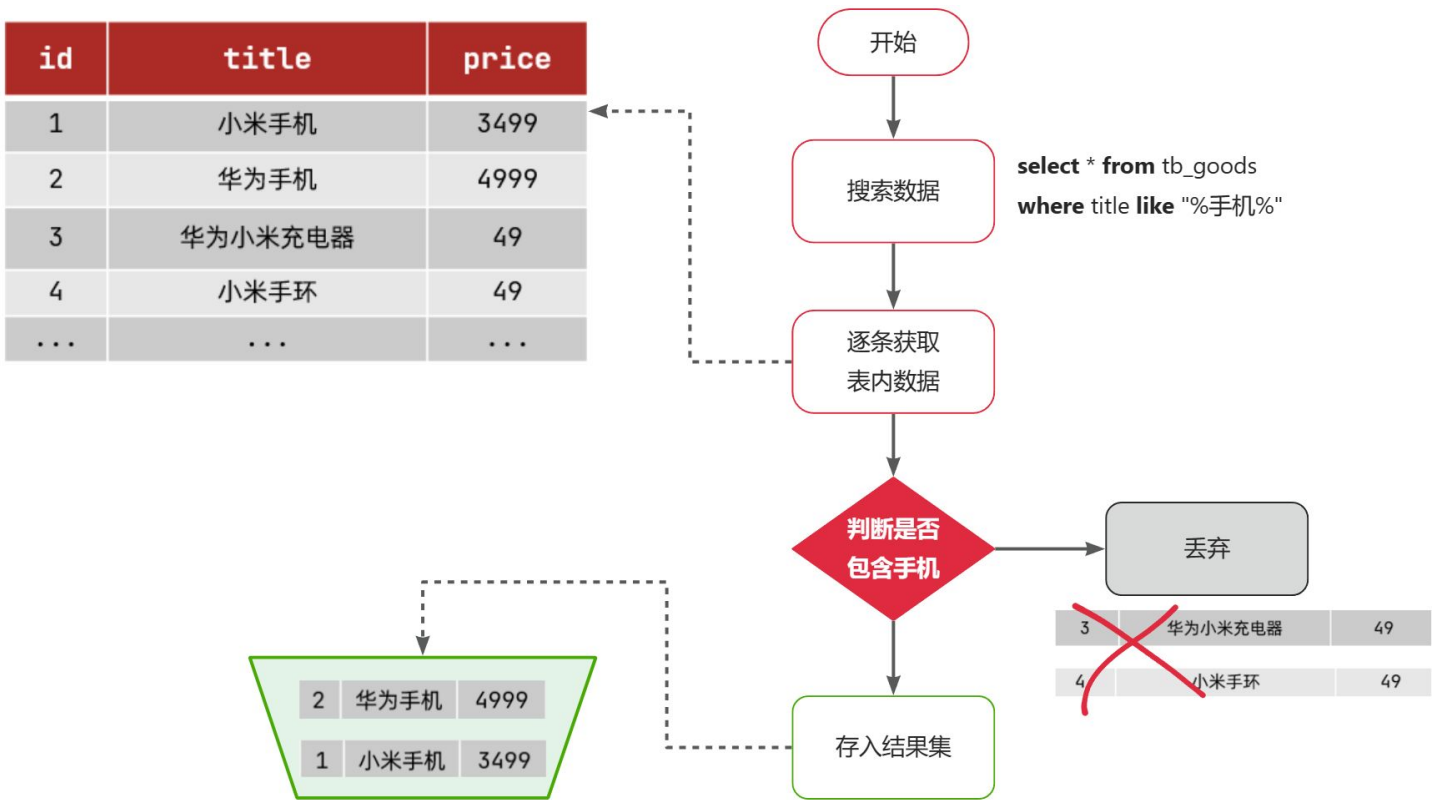
id	title	price
1	小米手机	3499
2	华为手机	4999
3	华为小米充电器	49
4	小米手环	49
...

其中的 `id` 字段已经创建了索引，由于索引底层采用了B+树结构，因此我们根据id搜索的速度会非常快。但是其他字段例如 `title`，只在叶子节点上存在。

因此要根据 `title` 搜索的时候只能遍历树中的每一个叶子节点，判断title数据是否符合要求。比如用户的SQL语句为：

```
1 select * from tb_goods where title like '%手机%';
```

那搜索的大概流程如图：



说明：

- 1) 检查到搜索条件为 `like '%手机%'`，需要找到 `title` 中包含 `手机` 的数据
- 2) 逐条遍历每行数据（每个叶子节点），比如第1次拿到 `id` 为1的数据
- 3) 判断数据中的 `title` 字段值是否符合条件
- 4) 如果符合则放入结果集，不符合则丢弃
- 5) 回到步骤1

综上，根据id精确匹配时，可以走索引，查询效率较高。而当搜索条件为模糊匹配时，由于索引无法生效，导致从索引查询退化为全表扫描，效率很差。

因此，正向索引适合于根据索引字段的精确搜索，不适合基于部分词条的模糊匹配。

而倒排索引恰好解决的就是根据部分词条模糊匹配的问题。

1.2.2.倒排索引

倒排索引中有两个非常重要的概念：

- 文档（`Document`）：用来搜索的数据，其中的每一条数据就是一个文档。例如一个网页、一个商品信息
- 词条（`Term`）：对文档数据或用户搜索数据，利用某种算法分词，得到的具备含义的词语就是词条。例如：我是中国人，就可以分为：我、是、中国人、中国、国人这样的几个词条

创建倒排索引是对正向索引的一种特殊处理和应用，流程如下：

- 将每一个文档的数据利用**分词算法**根据语义拆分，得到一个个词条
- 创建表，每行数据包括词条、词条所在文档id、位置等信息
- 因为词条唯一性，可以给词条创建**正向索引**

此时形成的这张以词条为索引的表，就是倒排索引表，两者对比如下：

正向索引

id（索引）	title	price
1	小米手机	3499
2	华为手机	4999
3	华为小米充电器	49
4	小米手环	49
...

倒排索引

词条（索引）	文档id
小米	1, 3, 4
手机	1, 2
华为	2, 3
充电器	3
手环	4

倒排索引的**搜索流程**如下（以搜索"华为手机"为例），如图：



流程描述：

- 1) 用户输入条件 "华为手机" 进行搜索。
- 2) 对用户输入条件**分词**，得到词条： 华为 、 手机 。
- 3) 拿着词条在倒排索引中查找（**由于词条有索引，查询效率很高**），即可得到包含词条的文档id：1、2、3 。
- 4) 拿着文档 id 到正向索引中查找具体文档即可（由于 id 也有索引，查询效率也很高）。

虽然要先查询倒排索引，再查询正向索引，但是无论是词条、还是文档id都建立了索引，查询速度非常快！无需全表扫描。

1.2.3.正向和倒排

那么为什么一个叫做正向索引，一个叫做倒排索引呢？

- **正向索引**是最传统的，根据id索引的方式。但根据词条查询时，必须先逐条获取每个文档，然后判断文档中是否包含所需要的词条，是**根据文档找词条的过程**。
- 而**倒排索引**则相反，是先找到用户要搜索的词条，根据词条得到保护词条的文档的id，然后根据id获取文档。是**根据词条找文档的过程**。

是不是恰好反过来了？

那么两者方式的优缺点是什么呢？

正向索引：

- 优点：
 - 可以给多个字段创建索引
 - 根据索引字段搜索、排序速度非常快
- 缺点：
 - 根据非索引字段，或者索引字段中的部分词条查找时，只能全表扫描。

倒排索引：

- 优点：
 - 根据词条搜索、模糊搜索时，速度非常快
- 缺点：
 - 只能给词条创建索引，而不是字段
 - 无法根据字段做排序

1.3.基础概念

elasticsearch中有很多独有的概念，与mysql中略有差别，但也有相似之处。

1.3.1.文档和字段

elasticsearch是面向**文档（Document）**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：

```
1 {  
2     "id": 1,  
3     "title": "小米手机",  
4     "price": 3499
```

id (索引)	title	price
1	小米手机	3499
2	华为手机	4999
3	华为小米充电器	49
4	小米手环	49
...



```

5 }
6 {
7     "id": 2,
8     "title": "华为手机",
9     "price": 4999
10 }
11 {
12     "id": 3,
13     "title": "华为小米充电器",
14     "price": 49
15 }
16 {
17     "id": 4,
18     "title": "小米手环",
19     "price": 299
20 }
21

```

因此，原本数据库中的一行数据就是ES中的一个JSON文档；而数据库中每行数据都包含很多列，这些列就转换为JSON文档中的**字段（Field）**。

1.3.2.索引和映射

随着业务发展，需要在es中存储的文档也会越来越多，比如有商品的文档、用户的文档、订单文档等等：

```

{
  "id": 103,
  "name": "麻子",
  "age": 18
}

{
  "id": 11,
  "userId": 102,
  "goodsId": 2,
  "totalFee": 328
}

{
  "id": 102,
  "name": "李四",
  "age": 24
}

{
  "id": 10,
  "userId": 101,
  "goodsId": 1,
  "totalFee": 294
}

{
  "id": 2,
  "title": "华为手机",
  "price": 4999
}

{
  "id": 1,
  "title": "小米手机",
  "price": 3499
}

{
  "id": 3,
  "title": "三星手机",
  "price": 3999
}

{
  "id": 101,
  "name": "张三",
  "age": 21
}

```

黑马程序员-研究院

所有文档都散乱存放显然非常混乱，也不方便管理。

因此，我们要将类型相同的文档集中在一起管理，称为**索引（Index）**。例如：

商品索引

用户索引

订单索引

```

1 {
2     "id": 1,
3     "title": "小米
  手机",
4     "price": 3499
5 }
6
7 {
8     "id": 2,
9     "title": "华为
  手机",
10    "price": 4999
11 }
12
13 {
14    "id": 3,
15    "title": "三星
  手机",
16    "price": 3999
17 }

```

```

1 {
2     "id": 101,
3     "name": "张
  三",
4     "age": 21
5 }
6
7 {
8     "id": 102,
9     "name": "李
  四",
10    "age": 24
11 }
12
13 {
14    "id": 103,
15    "name": "麻
  子",
16    "age": 18
17 }

```

```

1 {
2     "id": 10,
3     "userId": 101,
4     "goodsId": 1,
5     "totalFee":
      294
6 }
7
8 {
9     "id": 11,
10    "userId": 102,
11    "goodsId": 2,
12    "totalFee":
      328
13 }
14

```

- 所有用户文档，就可以组织在一起，称为用户的索引；
- 所有商品的文档，可以组织在一起，称为商品的索引；
- 所有订单的文档，可以组织在一起，称为订单的索引；

因此，我们可以把索引当做是数据库中的表。

数据库的表会有约束信息，用来定义表的结构、字段的名称、类型等信息。因此，索引库中就有**映射（mapping）**，是索引中文档的字段约束信息，类似表的结构约束。

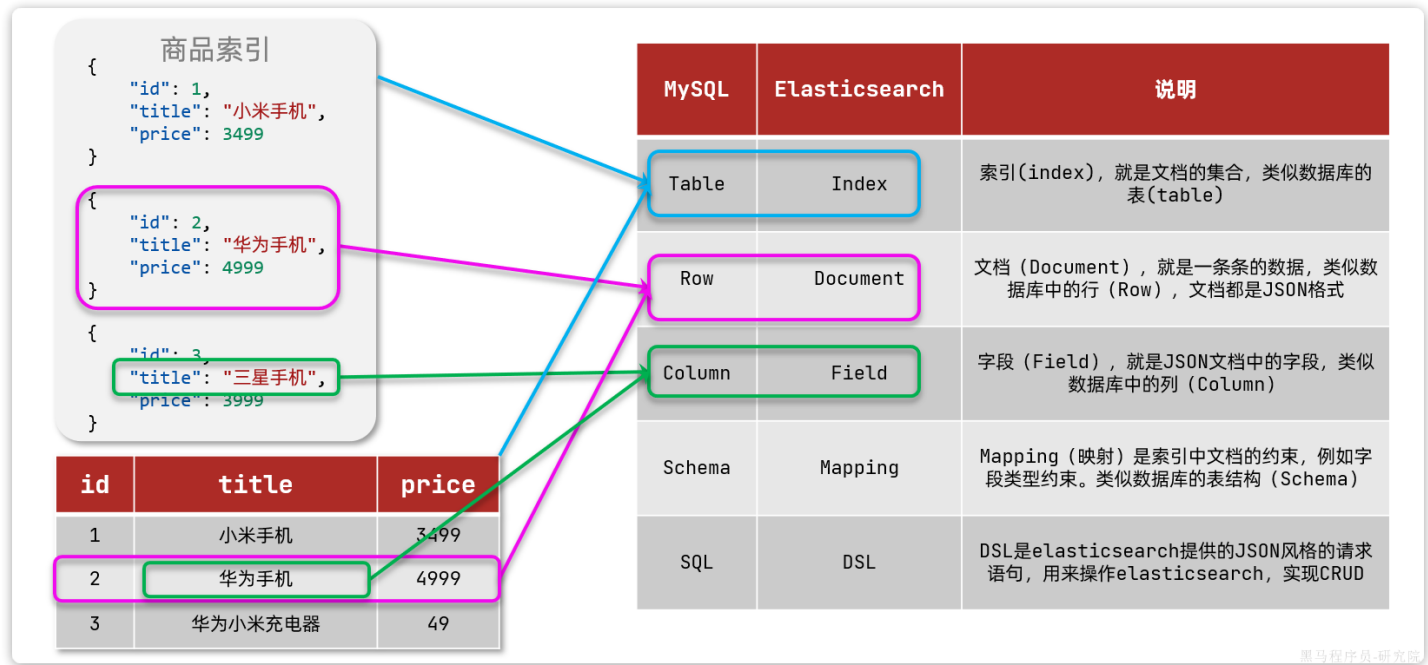
1.3.3.mysql与elasticsearch

我们统一的把mysql与elasticsearch的概念做一下对比：

MySQL	Elasticsearch	说明
Table	Index	索引(index)，就是文档的集合，类似数据库的表(table)
Row	Document	文档（Document），就是一条条的数据，类似数据库中的行（Row），文档都是JSON格式
Column	Field	字段（Field），就是JSON文档中的字段，类似数据库中的列（Column）

Schem a	Mapping	Mapping（映射）是索引中文档的约束，例如字段类型约束。类似数据库的表结构（Schema）
SQL	DSL	DSL是elasticsearch提供的JSON风格的请求语句，用来操作elasticsearch，实现CRUD

如图：



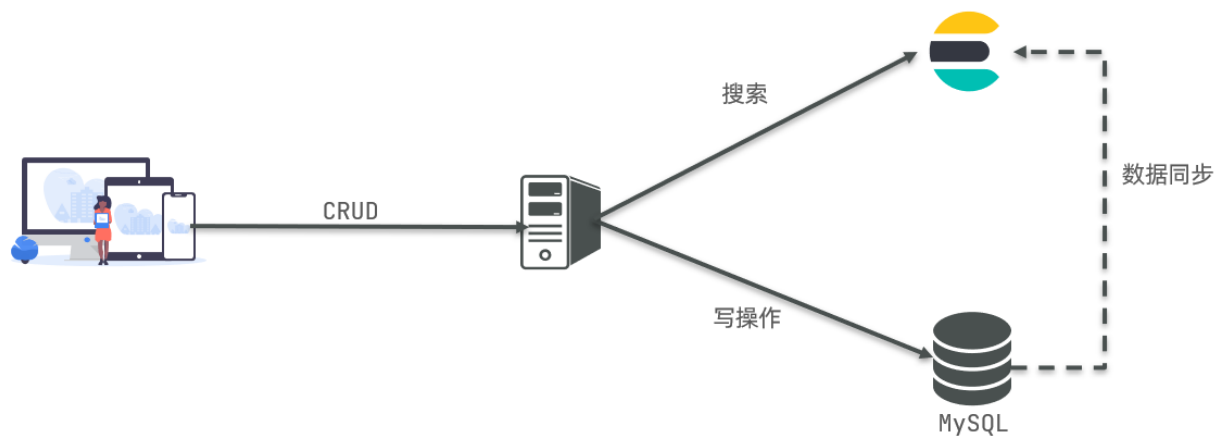
那是不是说，我们学习了elasticsearch就不再需要mysql了呢？

并不是如此，两者各自有自己的擅长之处：

- Mysql：擅长事务类型操作，可以确保数据的安全和一致性
- Elasticsearch：擅长海量数据的搜索、分析、计算

因此企业中，往往是两者结合使用：

- 对安全性要求较高的写操作，使用mysql实现
- 对查询性能要求较高的搜索需求，使用elasticsearch实现
- 两者再基于某种方式，实现数据的同步，保证一致性



黑马程序员-研究院

1.4.IK分词器

Elasticsearch的关键就是倒排索引，而倒排索引依赖于对文档内容的分词，而分词则需要高效、精准的分词算法，IK分词器就是这样一个中文分词算法。

1.4.1.安装IK分词器

方案一：在线安装

运行一个命令即可：

```
1 docker exec -it es ./bin/elasticsearch-plugin install
  https://github.com/medcl/elasticsearch-analysis-
  ik/releases/download/v7.12.1/elasticsearch-analysis-ik-7.12.1.zip
```

然后重启es容器：

```
1 docker restart es
```

方案二：离线安装

如果网速较差，也可以选择离线安装。

首先，查看之前安装的Elasticsearch容器的plugins数据卷目录：

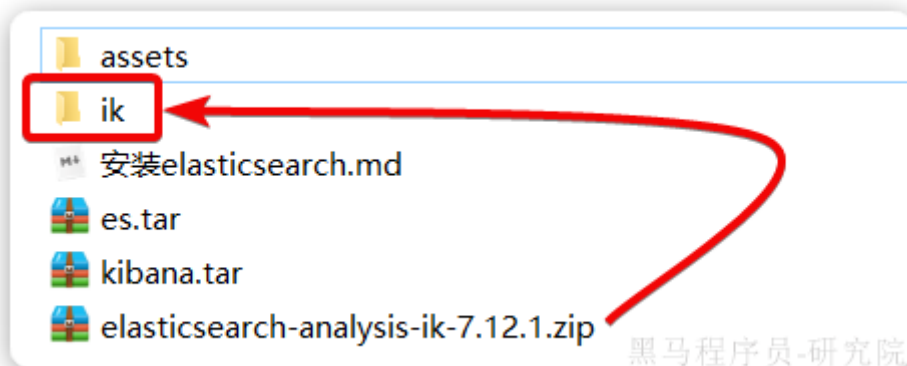
```
1 docker volume inspect es-plugins
```


结果如下：

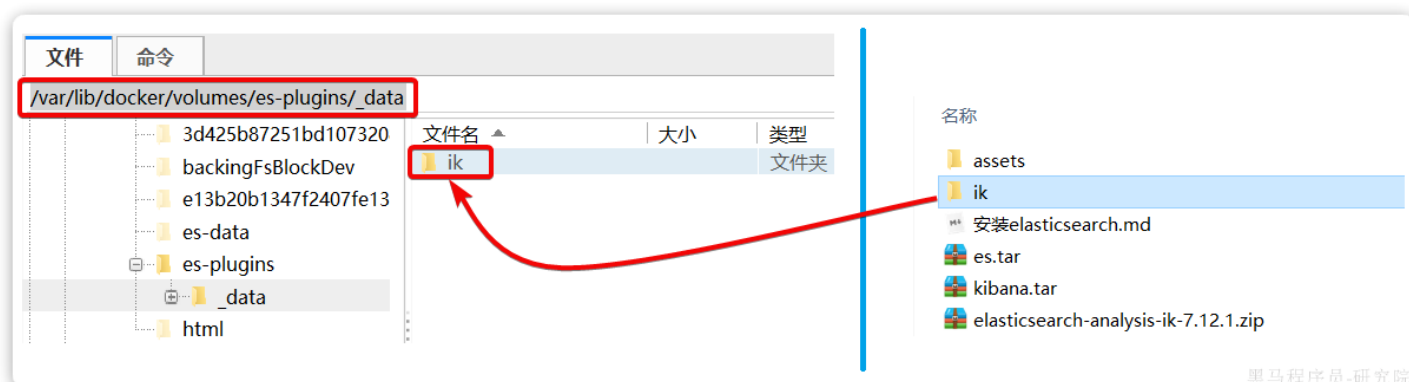
```
1  [  
2    {  
3      "CreatedAt": "2024-11-06T10:06:34+08:00",  
4      "Driver": "local",  
5      "Labels": null,  
6      "Mountpoint": "/var/lib/docker/volumes/es-plugins/_data",  
7      "Name": "es-plugins",  
8      "Options": null,  
9      "Scope": "local"  
10   }  
11 ]
```

可以看到elasticsearch的插件挂载到了 `/var/lib/docker/volumes/es-plugins/_data` 这个目录。我们需要把IK分词器上传至这个目录。

找到课前资料提供的ik分词器插件，课前资料提供了 `7.12.1` 版本的ik分词器压缩文件，你需要对其解压：



然后上传至虚拟机的 `/var/lib/docker/volumes/es-plugins/_data` 这个目录：



最后，重启es容器：

```
1 docker restart es
```

1.4.2.使用IK分词器

IK分词器包含两种模式：

- `ik_smart`：智能语义切分
- `ik_max_word`：最细粒度切分

我们在Kibana的DevTools上来测试分词器，首先测试Elasticsearch官方提供的标准分词器：

```
1 POST /_analyze
2 {
3   "analyzer": "standard",
4   "text": "黑马程序员学习java太棒了"
5 }
```

结果如下：

```
1 {
2   "tokens" : [
3     {
4       "token" : "黑",
5       "start_offset" : 0,
6       "end_offset" : 1,
7       "type" : "<IDEOGRAPHIC>",
8       "position" : 0
9     },
10    {
11      "token" : "马",
12      "start_offset" : 1,
13      "end_offset" : 2,
14      "type" : "<IDEOGRAPHIC>",
15      "position" : 1
16    },
17    {
18      "token" : "程",
19      "start_offset" : 2,
20      "end_offset" : 3,
21      "type" : "<IDEOGRAPHIC>",
```

```
22     "position" : 2
23 },
24 {
25     "token" : "序",
26     "start_offset" : 3,
27     "end_offset" : 4,
28     "type" : "<IDEOGRAPHIC>",
29     "position" : 3
30 },
31 {
32     "token" : "员",
33     "start_offset" : 4,
34     "end_offset" : 5,
35     "type" : "<IDEOGRAPHIC>",
36     "position" : 4
37 },
38 {
39     "token" : "学",
40     "start_offset" : 5,
41     "end_offset" : 6,
42     "type" : "<IDEOGRAPHIC>",
43     "position" : 5
44 },
45 {
46     "token" : "习",
47     "start_offset" : 6,
48     "end_offset" : 7,
49     "type" : "<IDEOGRAPHIC>",
50     "position" : 6
51 },
52 {
53     "token" : "java",
54     "start_offset" : 7,
55     "end_offset" : 11,
56     "type" : "<ALPHANUM>",
57     "position" : 7
58 },
59 {
60     "token" : "太",
61     "start_offset" : 11,
62     "end_offset" : 12,
63     "type" : "<IDEOGRAPHIC>",
64     "position" : 8
65 },
66 {
67     "token" : "棒",
68     "start_offset" : 12,
```

```
69     "end_offset" : 13,
70     "type" : "<IDEOGRAPHIC>",
71     "position" : 9
72   },
73   {
74     "token" : "了",
75     "start_offset" : 13,
76     "end_offset" : 14,
77     "type" : "<IDEOGRAPHIC>",
78     "position" : 10
79   }
80 ]
81 }
82
```

可以看到，标准分词器智能1字1词条，无法正确对中文做分词。

我们再测试IK分词器：

```
1 POST /_analyze
2 {
3   "analyzer": "ik_smart",
4   "text": "黑马程序员学习java太棒了"
5 }
```

执行结果如下：

```
1 {
2   "tokens" : [
3     {
4       "token" : "黑马",
5       "start_offset" : 0,
6       "end_offset" : 2,
7       "type" : "CN_WORD",
8       "position" : 0
9     },
10    {
11      "token" : "程序员",
12      "start_offset" : 2,
13      "end_offset" : 5,
14      "type" : "CN_WORD",
15      "position" : 1
16    }
17  ]
18 }
```

```
16     },
17     {
18         "token" : "学习",
19         "start_offset" : 5,
20         "end_offset" : 7,
21         "type" : "CN_WORD",
22         "position" : 2
23     },
24     {
25         "token" : "java",
26         "start_offset" : 7,
27         "end_offset" : 11,
28         "type" : "ENGLISH",
29         "position" : 3
30     },
31     {
32         "token" : "太棒了",
33         "start_offset" : 11,
34         "end_offset" : 14,
35         "type" : "CN_WORD",
36         "position" : 4
37     }
38 ]
39 }
40
```

1.4.3.拓展词典

随着互联网的发展，“造词运动”也越发的频繁。出现了很多新的词语，在原有的词汇列表中并不存在。比如：“泰裤辣”，“传智播客”等。

IK分词器无法对这些词汇分词，测试一下：

```
1 POST /_analyze
2 {
3     "analyzer": "ik_max_word",
4     "text": "传智播客开设大学,真的泰裤辣! "
5 }
```

结果：

```
1 {
```

```
2  "tokens" : [  
3    {  
4      "token" : "传",  
5      "start_offset" : 0,  
6      "end_offset" : 1,  
7      "type" : "CN_CHAR",  
8      "position" : 0  
9    },  
10   {  
11     "token" : "智",  
12     "start_offset" : 1,  
13     "end_offset" : 2,  
14     "type" : "CN_CHAR",  
15     "position" : 1  
16   },  
17   {  
18     "token" : "播",  
19     "start_offset" : 2,  
20     "end_offset" : 3,  
21     "type" : "CN_CHAR",  
22     "position" : 2  
23   },  
24   {  
25     "token" : "客",  
26     "start_offset" : 3,  
27     "end_offset" : 4,  
28     "type" : "CN_CHAR",  
29     "position" : 3  
30   },  
31   {  
32     "token" : "开设",  
33     "start_offset" : 4,  
34     "end_offset" : 6,  
35     "type" : "CN_WORD",  
36     "position" : 4  
37   },  
38   {  
39     "token" : "大学",  
40     "start_offset" : 6,  
41     "end_offset" : 8,  
42     "type" : "CN_WORD",  
43     "position" : 5  
44   },  
45   {  
46     "token" : "真的",  
47     "start_offset" : 9,  
48     "end_offset" : 11,
```

```

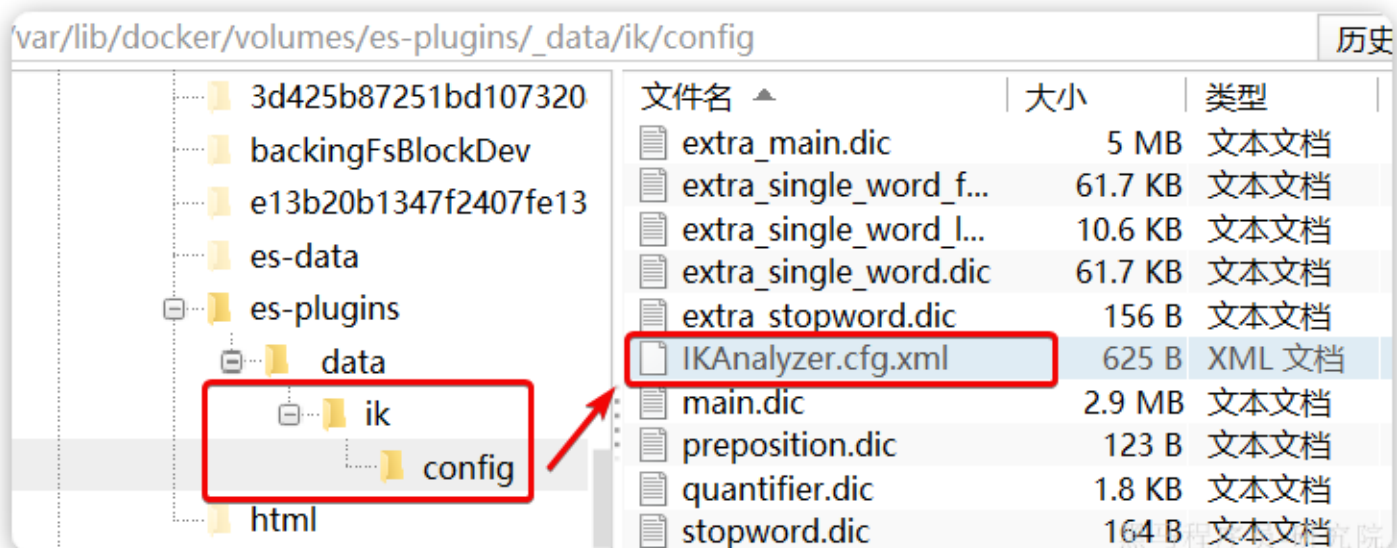
49     "type" : "CN_WORD",
50     "position" : 6
51 },
52 {
53     "token" : "泰",
54     "start_offset" : 11,
55     "end_offset" : 12,
56     "type" : "CN_CHAR",
57     "position" : 7
58 },
59 {
60     "token" : "裤",
61     "start_offset" : 12,
62     "end_offset" : 13,
63     "type" : "CN_CHAR",
64     "position" : 8
65 },
66 {
67     "token" : "辣",
68     "start_offset" : 13,
69     "end_offset" : 14,
70     "type" : "CN_CHAR",
71     "position" : 9
72 }
73 ]
74 }
75

```

可以看到，**传智播客** 和 **泰裤辣** 都无法正确分词。

所以要想正确分词，IK分词器的词库也需要不断的更新，IK分词器提供了扩展词汇的功能。

1) 打开IK分词器config目录：



注意，如果采用在线安装的通过，默认是没有config目录的，需要把课前资料提供的ik下的config上传至对应目录。

2) 在IKAnalyzer.cfg.xml配置文件内容添加：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3 <properties>
4     <comment>IK Analyzer 扩展配置</comment>
5     <!--用户可以在这里配置自己的扩展字典 *** 添加扩展词典-->
6     <entry key="ext_dict">ext.dic</entry>
7 </properties>
```

3) 在IK分词器的config目录新建一个 `ext.dic` ，可以参考config目录下复制一个配置文件进行修改

```
1 传智播客
2 泰裤辣
```

4) 重启elasticsearch

```
1 docker restart es
2
3 # 查看 日志
4 docker logs -f elasticsearch
```

再次测试，可以发现 `传智播客` 和 `泰裤辣` 都正确分词了：

```
1 {
2   "tokens" : [
3     {
4       "token" : "传智播客",
5       "start_offset" : 0,
6       "end_offset" : 4,
7       "type" : "CN_WORD",
```



```
8      "position" : 0
9    },
10   {
11     "token" : "开设",
12     "start_offset" : 4,
13     "end_offset" : 6,
14     "type" : "CN_WORD",
15     "position" : 1
16   },
17   {
18     "token" : "大学",
19     "start_offset" : 6,
20     "end_offset" : 8,
21     "type" : "CN_WORD",
22     "position" : 2
23   },
24   {
25     "token" : "真的",
26     "start_offset" : 9,
27     "end_offset" : 11,
28     "type" : "CN_WORD",
29     "position" : 3
30   },
31   {
32     "token" : "泰裤辣",
33     "start_offset" : 11,
34     "end_offset" : 14,
35     "type" : "CN_WORD",
36     "position" : 4
37   }
38 ]
39 }
```

1.4.4.总结

分词器的作用是什么？

- 创建倒排索引时，对文档分词
- 用户搜索时，对输入的内容分词

IK分词器有几种模式？

- `ik_smart`：智能切分，粗粒度

- `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？

- 利用config目录的 `IkAnalyzer.cfg.xml` 文件添加拓展词典和停用词典
- 在词典中添加拓展词条或者停用词条

2.索引库操作

Index就类似数据库表，Mapping映射就类似表的结构。我们要向es中存储数据，必须先创建Index和Mapping

2.1.Mapping映射属性

Mapping是对索引库中文档的约束，常见的Mapping属性包括：

- `type`：字段数据类型，常见的简单类型有：
 - 字符串：`text`（可分词的文本）、`keyword`（精确值，例如：品牌、国家、ip地址）
 - 数值：`long`、`integer`、`short`、`byte`、`double`、`float`、
 - 布尔：`boolean`
 - 日期：`date`
 - 对象：`object`
- `index`：是否创建索引，默认为 `true`
- `analyzer`：使用哪种分词器
- `properties`：该字段的子字段

例如下面的json文档：

```
1 {
2     "age": 21,
3     "weight": 52.1,
4     "isMarried": false,
5     "info": "黑马程序员Java讲师",
6     "email": "zy@itcast.cn",
7     "score": [99.1, 99.5, 98.9],
8     "name": {
9         "firstName": "云",
10        "lastName": "赵"
```

```
11     }
12 }
```

对应的每个字段映射（Mapping）：

字段名		字段类型	类型说明	是否参与搜索	是否参与分词	分词器
age		integer	整数	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
weight		float	浮点数	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
isMarried		boolean	布尔	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
info		text	字符串，但需要分词	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	IK
email		keyword	字符串，但是不分词	<input type="checkbox"/>	<input type="checkbox"/>	—
score		float	只看数组中元素类型	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
name	firstName	keyword	字符串，但是不分词	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
	lastName	keyword	字符串，但是不分词	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—

2.2.索引库的CRUD

由于Elasticsearch采用的是Restful风格的API，因此其请求方式和路径相对都比较规范，而且请求参数也都采用JSON风格。

我们直接基于Kibana的DevTools来编写请求做测试，由于有语法提示，会非常方便。

2.2.1.创建索引库和映射

基本语法：

- 请求方式：PUT
- 请求路径：/索引库名，可以自定义

- 请求参数：mapping 映射

格式：

```
1 PUT /索引库名称
2 {
3   "mappings": {
4     "properties": {
5       "字段名":{
6         "type": "text",
7         "analyzer": "ik_smart"
8       },
9       "字段名2":{
10        "type": "keyword",
11        "index": "false"
12      },
13      "字段名3":{
14        "properties": {
15          "子字段": {
16            "type": "keyword"
17          }
18        }
19      },
20      // ...略
21    }
22  }
23 }
```

示例：

```
1 # PUT /heima
2 {
3   "mappings": {
4     "properties": {
5       "info":{
6         "type": "text",
7         "analyzer": "ik_smart"
8       },
9       "email":{
10        "type": "keyword",
11        "index": "false"
12      }
13     }
14   }
15 }
```

```
12     },
13     "name": {
14         "properties": {
15             "firstName": {
16                 "type": "keyword"
17             }
18         }
19     }
20 }
21 }
22 }
```

2.2.2.查询索引库

基本语法：

- 请求方式：GET
- 请求路径：/索引库名
- 请求参数：无

格式：

```
1 GET /索引库名
```

示例：

```
1 GET /heima
```

2.2.3.修改索引库

倒排索引结构虽然不复杂，但是一旦数据结构改变（比如改变了分词器），就需要重新创建倒排索引，这简直是灾难。因此索引库**一旦创建，无法修改mapping**。

虽然无法修改mapping中已有的字段，但是却允许添加新的字段到mapping中，因为不会对倒排索引产生影响。因此修改索引库能做的就是向索引库中添加新字段，或者更新索引库的基础属性。

语法说明：

```
1 PUT /索引库名/_mapping
2 {
3   "properties": {
4     "新字段名":{
5       "type": "integer"
6     }
7   }
8 }
```

示例：

```
1 PUT /heima/_mapping
2 {
3   "properties": {
4     "age":{
5       "type": "integer"
6     }
7   }
8 }
```

2.2.4.删除索引库

语法：

- 请求方式：DELETE
- 请求路径：/索引库名
- 请求参数：无

格式：

```
1 DELETE /索引库名
```

示例：

```
1 DELETE /heima
```

2.2.5.总结

索引库操作有哪些？

- 创建索引库：PUT /索引库名
- 查询索引库：GET /索引库名
- 删除索引库：DELETE /索引库名
- 修改索引库，添加字段：PUT /索引库名/_mapping

可以看到，对索引库的操作基本遵循的Restful的风格，因此API接口非常统一，方便记忆。

3.文档操作

有了索引库，接下来就可以向索引库中添加数据了。

Elasticsearch中的数据其实就是JSON风格的文档。操作文档自然保护 **增**、**删**、**改**、**查** 等几种常见操作，我们分别来学习。

3.1.新增文档

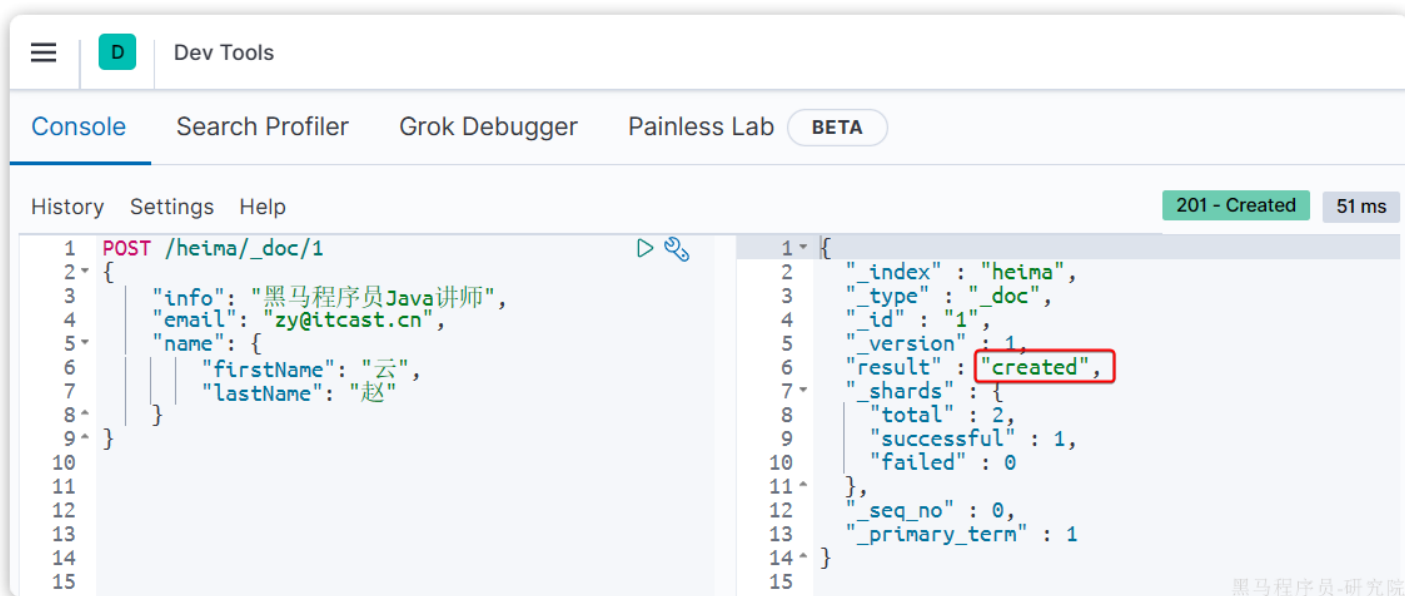
语法：

```
1 POST /索引库名/_doc/文档id
2 {
3     "字段1": "值1",
4     "字段2": "值2",
5     "字段3": {
6         "子属性1": "值3",
7         "子属性2": "值4"
8     },
9 }
```

示例：

```
1 POST /heima/_doc/1
2 {
3   "info": "黑马程序员Java讲师",
4   "email": "zy@itcast.cn",
5   "name": {
6     "firstName": "云",
7     "lastName": "赵"
8   }
9 }
```

响应：



3.2.查询文档

根据rest风格，新增是post，查询应该是get，不过查询一般都需要条件，这里我们把文档id带上。

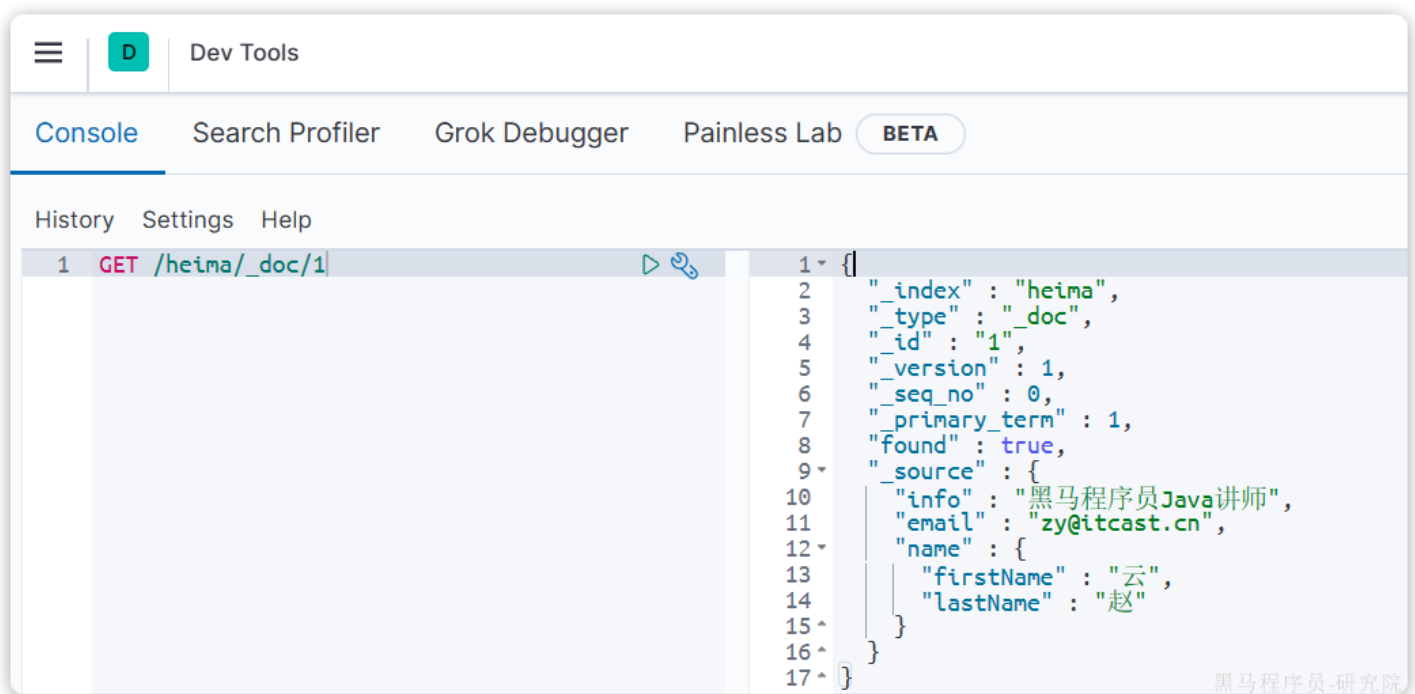
语法：

```
1 GET /{索引库名称}/_doc/{id}
```


示例：

```
1 GET /heima/_doc/1
```

查看结果：



3.3.删除文档

删除使用DELETE请求，同样，需要根据id进行删除：

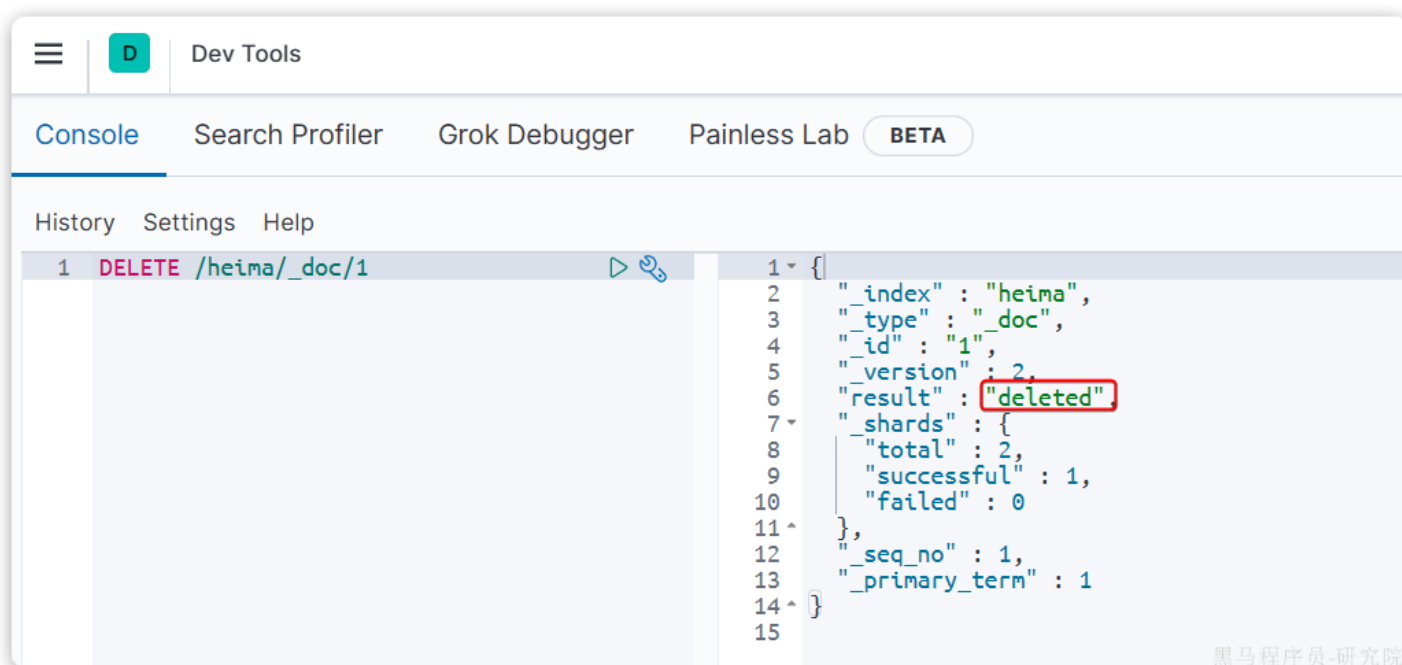
语法：

```
1 DELETE /{索引库名}/_doc/id值
```

示例：

```
1 DELETE /heima/_doc/1
```

结果：



3.4.修改文档

修改有两种方式：

- 全量修改：直接覆盖原来的文档
- 局部修改：修改文档中的部分字段

3.4.1.全量修改

全量修改是覆盖原来的文档，其本质是两步操作：

- 根据指定的id删除文档
- 新增一个相同id的文档

注意：如果根据id删除时，id不存在，第二步的新增也会执行，也就从修改变成了新增操作了。

语法：

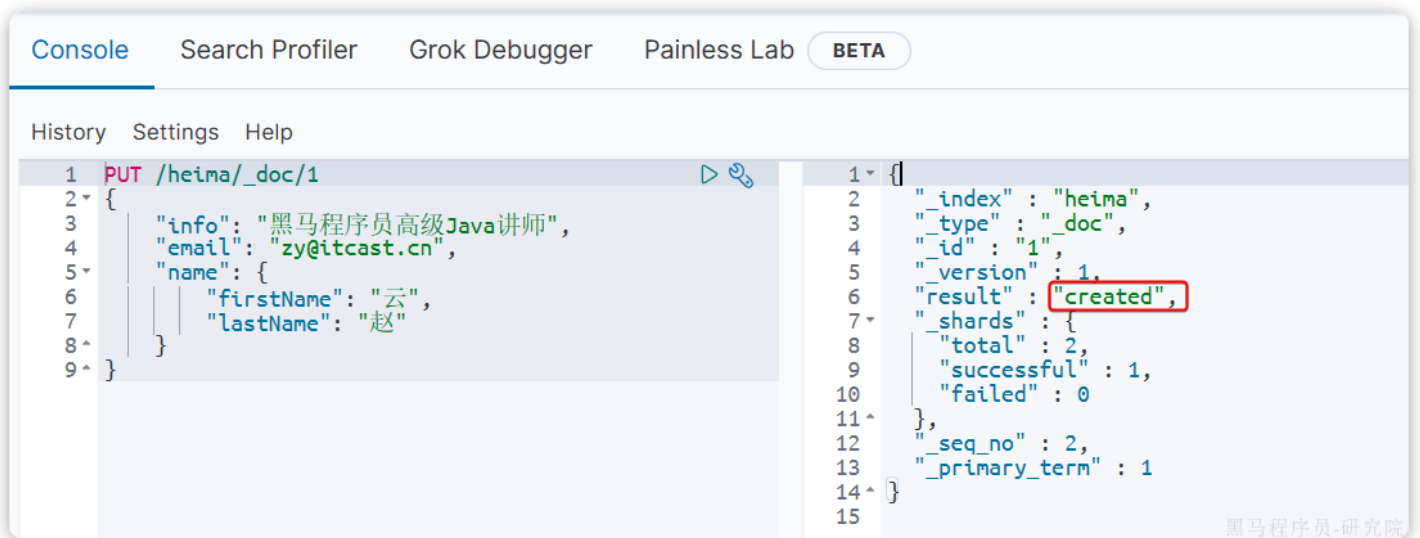
```
1 PUT /{索引库名}/_doc/文档id
2 {
3   "字段1": "值1",
4   "字段2": "值2",
```

```
5 // ... 略
6 }
```

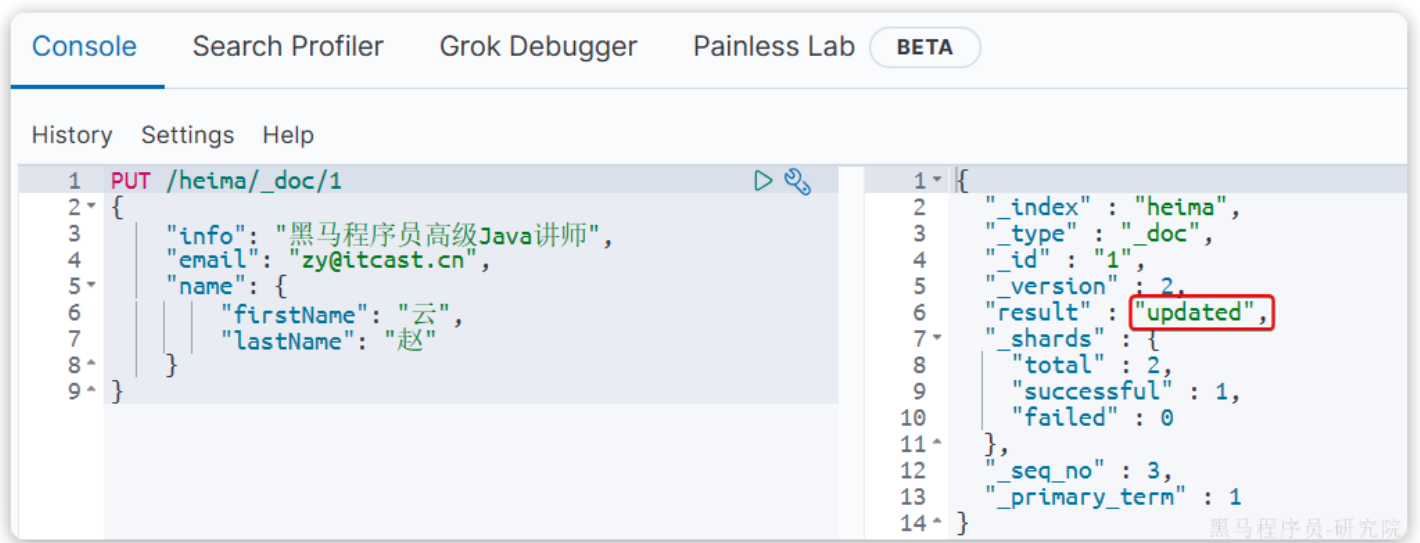
示例：

```
1 PUT /heima/_doc/1
2 {
3   "info": "黑马程序员高级Java讲师",
4   "email": "zy@itcast.cn",
5   "name": {
6     "firstName": "云",
7     "lastName": "赵"
8   }
9 }
```

由于 `id` 为 `1` 的文档已经被删除，所以第一次执行时，得到的反馈是 `created`：



所以如果执行第2次时，得到的反馈则是 `updated`：



3.4.2.局部修改

局部修改是只修改指定id匹配的文档中的部分字段。

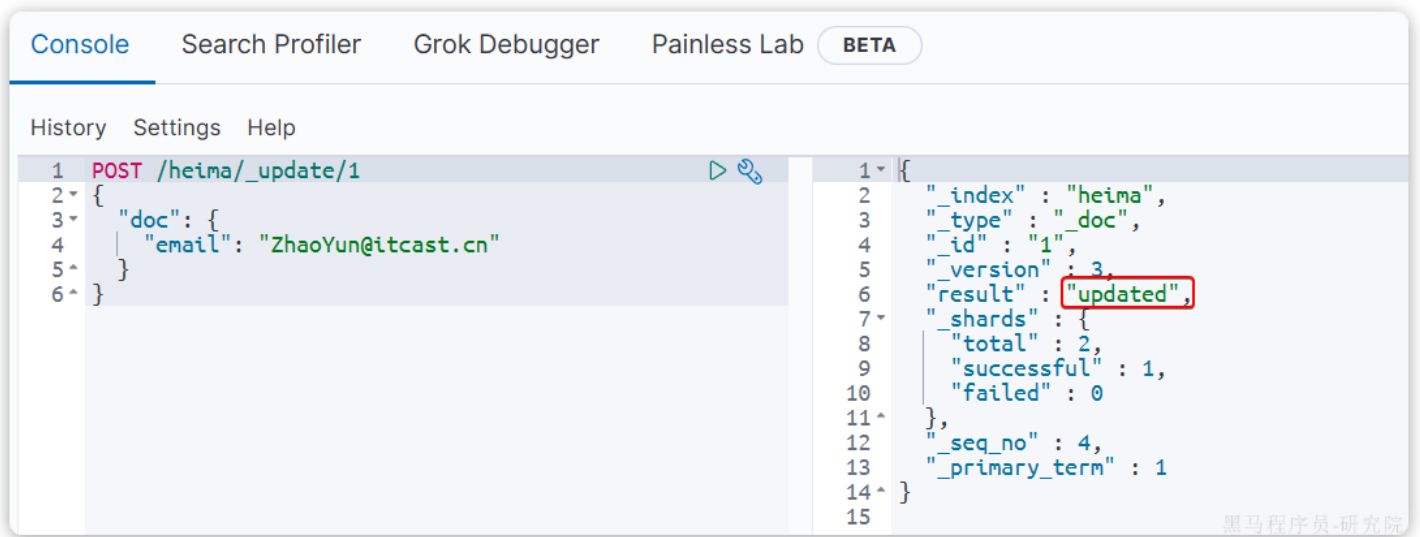
语法：

```
1 POST /{索引库名}/_update/文档id
2 {
3   "doc": {
4     "字段名": "新的值",
5   }
6 }
```

示例：

```
1 POST /heima/_update/1
2 {
3   "doc": {
4     "email": "ZhaoYun@itcast.cn"
5   }
6 }
```

执行结果：



3.5.批处理

批处理采用POST请求，基本语法如下：

```
1 POST _bulk
2 { "index" : { "_index" : "test", "_id" : "1" } }
3 { "field1" : "value1" }
4 { "delete" : { "_index" : "test", "_id" : "2" } }
5 { "create" : { "_index" : "test", "_id" : "3" } }
6 { "field1" : "value3" }
7 { "update" : { "_id" : "1", "_index" : "test" } }
8 { "doc" : { "field2" : "value2" } }
```

其中：

- `index` 代表新增操作
 - `_index`：指定索引库名
 - `_id` 指定要操作的文档id
 - `{ "field1" : "value1" }`：则是要新增的文档内容
- `delete` 代表删除操作
 - `_index`：指定索引库名
 - `_id` 指定要操作的文档id
- `update` 代表更新操作
 - `_index`：指定索引库名
 - `_id` 指定要操作的文档id

- `{ "doc" : {"field2" : "value2"} }` : 要更新的文档字段

示例，批量新增：

```
1 POST /_bulk
2 {"index": {"_index": "heima", "_id": "3"}}
3 {"info": "黑马程序员C++讲师", "email": "ww@itcast.cn", "name": {"firstName": "五", "lastName": "王"}}
4 {"index": {"_index": "heima", "_id": "4"}}
5 {"info": "黑马程序员前端讲师", "email": "zhangsan@itcast.cn", "name": {"firstName": "三", "lastName": "张"}}
```

批量删除：

```
1 POST /_bulk
2 {"delete": {"_index": "heima", "_id": "3"}}
3 {"delete": {"_index": "heima", "_id": "4"}}
```

3.6.总结

文档操作有哪些？

- 创建文档： `POST /{索引库名}/_doc/文档id { json文档 }`
- 查询文档： `GET /{索引库名}/_doc/文档id`
- 删除文档： `DELETE /{索引库名}/_doc/文档id`
- 修改文档：
 - 全量修改： `PUT /{索引库名}/_doc/文档id { json文档 }`
 - 局部修改： `POST /{索引库名}/_update/文档id { "doc": {字段}}`

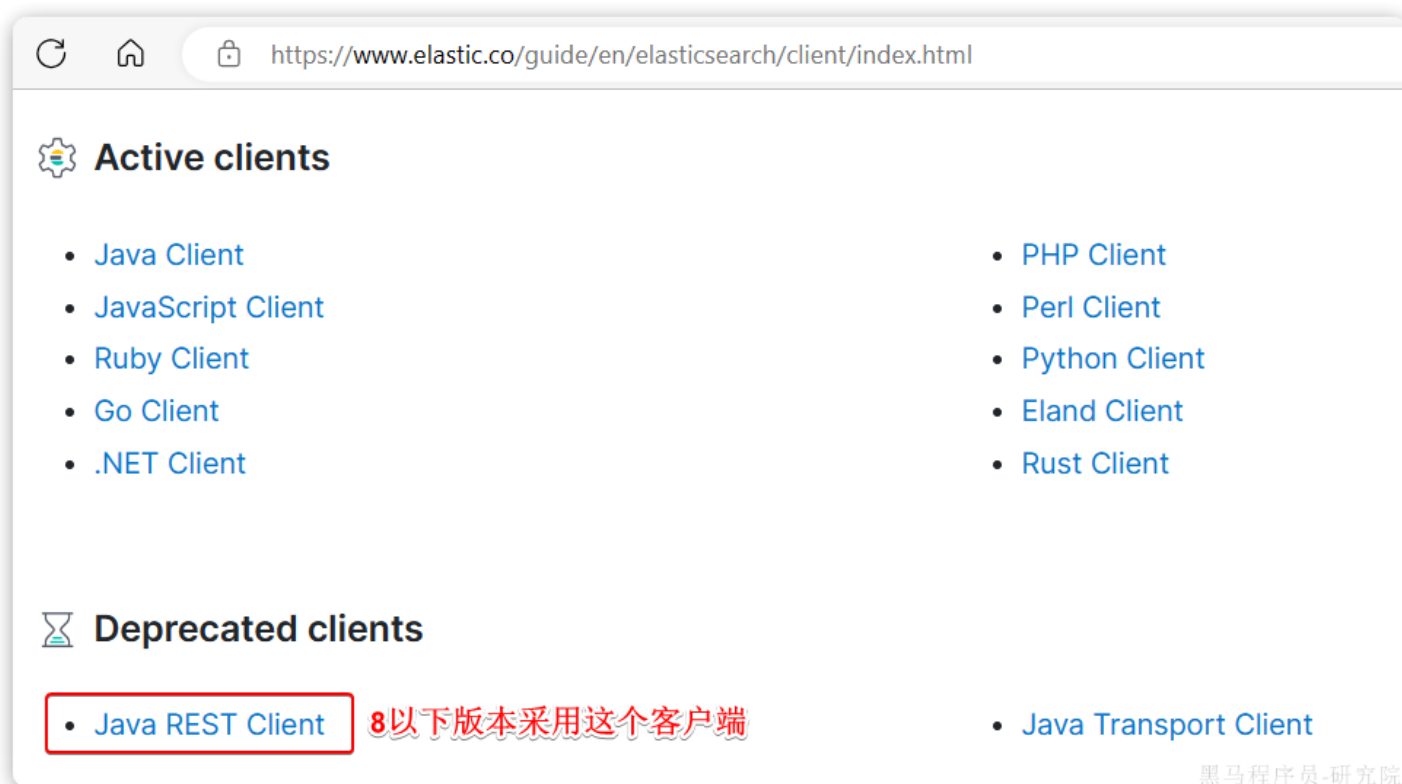
4.RestAPI

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。

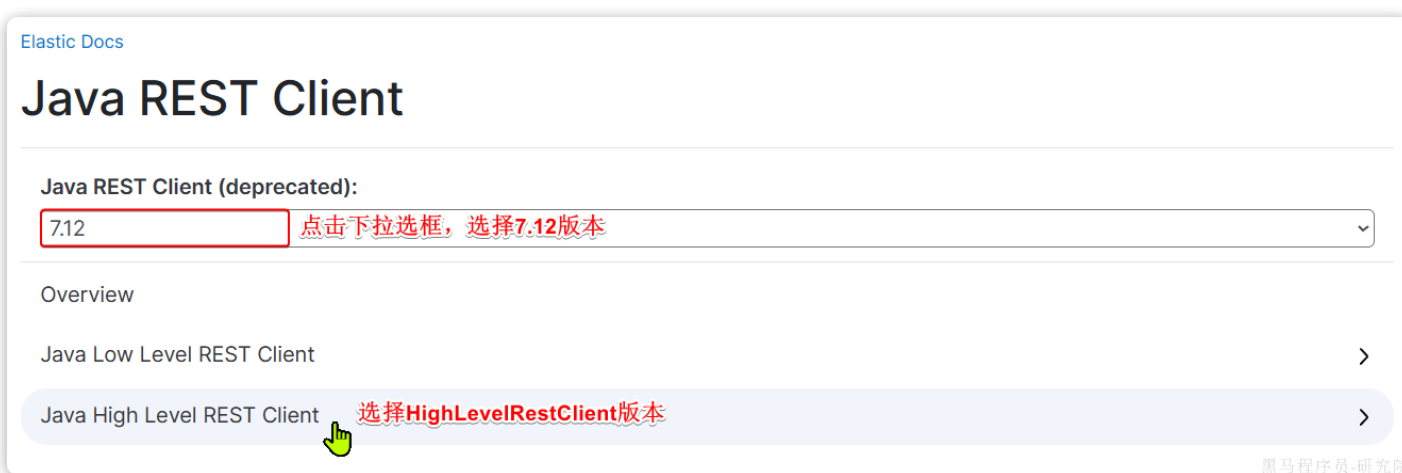
官方文档地址：

<https://www.elastic.co/guide/en/elasticsearch/client/index.html>

由于ES目前最新版本是8.8，提供了全新版本的客户端，老版本的客户端已经被标记为过时。而我们采用的是7.12版本，因此只能使用老版本客户端：



然后选择7.12版本，HighLevelRestClient版本：



4.1.初始化RestClient

在elasticsearch提供的API中，与elasticsearch一切交互都封装在一个名为 `RestHighLevelClient` 的类中，必须先完成这个对象的初始化，建立与elasticsearch的连接。

分为三步：

1) 在 `item-service` 模块中引入 `es` 的 `RestHighLevelClient` 依赖：

```
1 <dependency>
2     <groupId>org.elasticsearch.client</groupId>
3     <artifactId>elasticsearch-rest-high-level-client</artifactId>
4 </dependency>
```

2) 因为SpringBoot默认的ES版本是 `7.17.10` ，所以我们需要覆盖默认的ES版本：

```
1 <properties>
2     <maven.compiler.source>11</maven.compiler.source>
3     <maven.compiler.target>11</maven.compiler.target>
4     <elasticsearch.version>7.12.1</elasticsearch.version>
5 </properties>
```

3) 初始化RestHighLevelClient:

初始化的代码如下：

```
1 RestHighLevelClient client = new RestHighLevelClient(RestClient.builder(
2     HttpHost.create("http://192.168.150.101:9200")
3 ));
```

这里为了单元测试方便，我们创建一个测试类 `IndexTest` ，然后将初始化的代码编写在 `@BeforeEach` 方法中：

```
1 package com.hmall.item.es;
2
3 import org.apache.http.HttpHost;
4 import org.elasticsearch.client.RestClient;
5 import org.elasticsearch.client.RestHighLevelClient;
6 import org.junit.jupiter.api.AfterEach;
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Test;
```



```
9
10 import java.io.IOException;
11
12 public class IndexTest {
13
14     private RestHighLevelClient client;
15
16     @BeforeEach
17     void setUp() {
18         this.client = new RestHighLevelClient(RestClient.builder(
19             HttpHost.create("http://192.168.150.101:9200")
20         ));
21     }
22
23     @Test
24     void testConnect() {
25         System.out.println(client);
26     }
27
28     @AfterEach
29     void tearDown() throws IOException {
30         this.client.close();
31     }
32 }
```

4.1.创建索引库

由于要实现对商品搜索，所以我们需要将商品添加到Elasticsearch中，不过需要根据搜索业务的需求来设定索引库结构，而不是一股脑的把MySQL数据写入Elasticsearch.

4.1.1.Mapping映射

搜索页面的效果如图所示：

黑马欢迎您! 请登录 免费注册

首页 | 我的购物车 | 我的黑马 | 黑马会员 | 企业采购 | 关注黑马 | 客户服务 | 网站导航


黑马商城 HMALL.COM

全部结果:

分类	手机	曲面电视	拉杆箱	休闲鞋	休闲鞋	硬盘	真皮包			
品牌	裕捷	小米	华为	oppo	尤妮佳	莎米特	意尔康	新秀丽	Apple	锤子
价格	100以下	100~299元	300~599元	600~899元	900~1599元	1600以上元				

默认排序 | 销量 | 价格 | 排序字段

共 88475 件商品 1/4424




¥ 654.00

OPPO A7 全面屏拍照手机
4GB+64GB 清新粉 全网通 移动联通

0 条评价
最近1个月销量: 0

立即购买 加入购物车




¥ 866.00

RIMOWA 26寸托运箱拉杆箱 SALSA
AIR系列果绿色 820.70.36.4

0 条评价
最近1个月销量: 0

立即购买 加入购物车




¥ 906.00

【千玺代言】华为新品 HUAWEI
nova 4 极点全面屏手机 2000万超广

0 条评价
最近1个月销量: 0

立即购买 加入购物车




¥ 959.00

vivo X23 8GB+128GB 幻夜蓝 水滴屏
全面屏 游戏手机 移动联通电信全网

123万+ 条评价
最近1个月销量: 1万+

立即购买 加入购物车



¥ 554.00

三星 Galaxy S8+ (SM-G9550)
6GB+128GB 谜夜黑 移动联通电信

0 条评价
最近1个月销量: 0

立即购买 加入购物车

实现搜索功能需要的字段包括三大部分:

- 搜索过滤字段
 - 分类
 - 品牌
 - 价格
- 排序字段
 - 默认: 按照更新时间降序排序
 - 销量
 - 价格
- 展示字段
 - 商品id: 用于点击后跳转
 - 图片地址
 - 是否是广告推广商品
 - 名称
 - 价格
 - 评价数量
 - 销量

对应的商品表结构如下, 索引库无关字段已经划掉:

#	名称	数据类型	注释	长度/集合
1	id	BIGINT	商品id	19
2	name	VARCHAR	SKU名称	200
3	price	INT	价格 (分)	10
4	stock	INT	库存数量	10
5	image	VARCHAR	商品图片	200
6	category	VARCHAR	类目名称	200
7	brand	VARCHAR	品牌名称	100
8	spec	VARCHAR	规格	200
9	sold	INT	销量	10
10	comment_count	INT	评论数	10
11	isAD	TINYINT	是否是推广广告, true/false	1
12	status	INT	商品状态 1-正常, 2-下架, 3-删除	10
13	create_time	DATETIME	创建时间	
14	update_time	DATETIME	更新时间	
15	creator	BIGINT	创建人	19
16	updater	BIGINT	修改人	19

黑马程序员-研究院

结合数据库表结构，以上字段对应的mapping映射属性如下：

字段名	字段类型	类型说明	是否参与搜索	是否参与分词	分词器
id	long	长整数	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
name	text	字符串，参与分词搜索	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	IK
price	integer	以分为单位，所以是整数	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
stock	integer	字符串，但需要分词	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
image	keyword	字符串，但是不分词	<input type="checkbox"/>	<input type="checkbox"/>	—
category	keyword	字符串，但是不分词	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
brand	keyword	字符串，但是不分词	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
sold	integer	销量，整数	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
commentCount	integer	评价，整数	<input type="checkbox"/>	<input type="checkbox"/>	—
isAD	boolean	布尔类型	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—
updateTime	Date	更新时间	<input checked="" type="checkbox"/>	<input type="checkbox"/>	—

因此，最终我们的索引库文档结构应该是这样：

```
1 PUT /items
2 {
3   "mappings": {
4     "properties": {
5       "id": {
6         "type": "keyword"
7       },
8       "name": {
9         "type": "text",
10        "analyzer": "ik_max_word"
11      },
12      "price": {
13        "type": "integer"
14      },
15      "stock": {
16        "type": "integer"
17      },
18      "image": {
19        "type": "keyword",
20        "index": false
21      },
22      "category": {
23        "type": "keyword"
24      },
25      "brand": {
26        "type": "keyword"
27      },
28      "sold": {
29        "type": "integer"
30      },
31      "commentCount": {
32        "type": "integer",
33        "index": false
34      },
35      "isAD": {
36        "type": "boolean"
37      },
38      "updateTime": {
39        "type": "date"
40      }
41    }
42  }
43 }
```

4.1.2.创建索引

创建索引库的API如下：

```
@Test
void testCreateIndex() throws IOException {
    // 1. 创建Request对象
    CreateIndexRequest request = new CreateIndexRequest("items");
    // 2. 准备请求参数 MAPPING_TEMPLATE中就是创建索引的请求参数JSON字符串
    request.source(MAPPING_TEMPLATE, XContentType.JSON);
    // 3. 发送请求
    client.indices().create(request, RequestOptions.DEFAULT);
}
```

PUT /items

```
{
  "mappings": {
    "properties": {
      "id": {
        "type": "keyword"
      },
      "name": {
        "type": "text",
        "analyzer": "ik_max_word"
      },
      "price": {
        "type": "integer"
      },
      "stock": {
        "type": "integer"
      },
      "image": {
        "type": "string"
      },
      "category": {
        "type": "string"
      },
      "brand": {
        "type": "string"
      },
      "sold": {
        "type": "integer"
      },
      "commentCount": {
        "type": "integer"
      },
      "isAD": {
        "type": "boolean"
      },
      "updateTime": {
        "type": "date"
      }
    }
  }
}
```

代码分为三步：

- 1) 创建Request对象。
 - 因为是创建索引库的操作，因此Request是 `CreateIndexRequest`。
- 2) 添加请求参数
 - 其实就是Json格式的Mapping映射参数。因为json字符串很长，这里是定义了静态字符串常量 `MAPPING_TEMPLATE`，让代码看起来更加优雅。
- 3) 发送请求
 - `client.indices()` 方法的返回值是 `IndicesClient` 类型，封装了所有与索引库操作有关的方法。例如创建索引、删除索引、判断索引是否存在等

在 `item-service` 中的 `IndexTest` 测试类中，具体代码如下：

```
1 @Test
2 void testCreateIndex() throws IOException {
3     // 1. 创建Request对象
```

```

4      CreateIndexRequest request = new CreateIndexRequest("items");
5      // 2.准备请求参数
6      request.source(MAPPING_TEMPLATE, XContentType.JSON);
7      // 3.发送请求
8      client.indices().create(request, RequestOptions.DEFAULT);
9  }
10
11 static final String MAPPING_TEMPLATE = "{\n" +
12     "  \"mappings\": {\n" +
13     "    \"properties\": {\n" +
14     "      \"id\": {\n" +
15     "        \"type\": \"keyword\"\n" +
16     "      },\n" +
17     "      \"name\": {\n" +
18     "        \"type\": \"text\",\n" +
19     "        \"analyzer\": \"ik_max_word\"\n" +
20     "      },\n" +
21     "      \"price\": {\n" +
22     "        \"type\": \"integer\"\n" +
23     "      },\n" +
24     "      \"stock\": {\n" +
25     "        \"type\": \"integer\"\n" +
26     "      },\n" +
27     "      \"image\": {\n" +
28     "        \"type\": \"keyword\",\n" +
29     "        \"index\": false\n" +
30     "      },\n" +
31     "      \"category\": {\n" +
32     "        \"type\": \"keyword\"\n" +
33     "      },\n" +
34     "      \"brand\": {\n" +
35     "        \"type\": \"keyword\"\n" +
36     "      },\n" +
37     "      \"sold\": {\n" +
38     "        \"type\": \"integer\"\n" +
39     "      },\n" +
40     "      \"commentCount\": {\n" +
41     "        \"type\": \"integer\"\n" +
42     "      },\n" +
43     "      \"isAD\": {\n" +
44     "        \"type\": \"boolean\"\n" +
45     "      },\n" +
46     "      \"updateTime\": {\n" +
47     "        \"type\": \"date\"\n" +
48     "      }\n" +
49     "    }\n" +
50     "  }\n" +

```

4.2.删除索引库

删除索引库的请求非常简单：

```
1 DELETE /hotel
```

与创建索引库相比：

- 请求方式从PUT变为DELTE
- 请求路径不变
- 无请求参数

所以代码的差异，注意体现在Request对象上。流程如下：

- 1) 创建Request对象。这次是DeleteIndexRequest对象
- 2) 准备参数。这里是无参，因此省略
- 3) 发送请求。改用delete方法

在 `item-service` 中的 `IndexTest` 测试类中，编写单元测试，实现删除索引：

```
1 @Test
2 void testDeleteIndex() throws IOException {
3     // 1.创建Request对象
4     DeleteIndexRequest request = new DeleteIndexRequest("items");
5     // 2.发送请求
6     client.indices().delete(request, RequestOptions.DEFAULT);
7 }
```

4.3.判断索引库是否存在

判断索引库是否存在，本质就是查询，对应的请求语句是：

```
1 GET /hotel
```

因此与删除的Java代码流程是类似的，流程如下：

- 1) 创建Request对象。这次是GetIndexRequest对象
- 2) 准备参数。这里是无参，直接省略
- 3) 发送请求。改用exists方法

```
1 @Test
2 void testExistsIndex() throws IOException {
3     // 1.创建Request对象
4     GetIndexRequest request = new GetIndexRequest("items");
5     // 2.发送请求
6     boolean exists = client.indices().exists(request, RequestOptions.DEFAULT);
7     // 3.输出
8     System.err.println(exists ? "索引库已经存在！" : "索引库不存在！");
9 }
```

4.4.总结

JavaRestClient操作elasticsearch的流程基本类似。核心是 `client.indices()` 方法来获取索引库的操作对象。

索引库操作的基本步骤：

- 初始化 `RestHighLevelClient`
- 创建XxxIndexRequest。XXX是 `Create`、`Get`、`Delete`
- 准备请求参数（`Create` 时需要，其它是无参，可以省略）
- 发送请求。调用 `RestHighLevelClient#indices().xxx()` 方法，xxx是 `create`、`exists`、`delete`

5.RestClient操作文档

索引库准备好以后，就可以操作文档了。为了与索引库操作分离，我们再次创建一个测试类，做两件事情：

- 初始化RestHighLevelClient

- 我们的商品数据在数据库，需要利用IHotelService去查询，所以注入这个接口

```
1 package com.hmall.item.es;
2
3 import com.hmall.item.service.IItemService;
4 import org.apache.http.HttpHost;
5 import org.elasticsearch.client.RestClient;
6 import org.elasticsearch.client.RestHighLevelClient;
7 import org.junit.jupiter.api.AfterEach;
8 import org.junit.jupiter.api.BeforeEach;
9 import org.springframework.beans.factory.annotation.Autowired;
10 import org.springframework.boot.test.context.SpringBootTest;
11
12 import java.io.IOException;
13
14 @SpringBootTest(properties = "spring.profiles.active=local")
15 public class DocumentTest {
16
17     private RestHighLevelClient client;
18     @Autowired
19     private IItemService itemService;
20
21     @BeforeEach
22     void setUp() {
23         this.client = new RestHighLevelClient(RestClient.builder(
24             HttpHost.create("http://192.168.150.101:9200")
25         ));
26     }
27
28     @AfterEach
29     void tearDown() throws IOException {
30         this.client.close();
31     }
32 }
```

5.1.新增文档

我们需要将数据库中的商品信息导入elasticsearch中，而不是造假数据了。

5.1.1.实体类

索引库结构与数据库结构还存在一些差异，因此我们要定义一个索引库结构对应的实体。

在 `hm-service` 模块的 `com.hmall.item.domain.dto` 包中定义一个新的DTO：

```
1 package com.hmall.item.domain.po;
2
3 import io.swagger.annotations.ApiModel;
4 import io.swagger.annotations.ApiModelProperty;
5 import lombok.Data;
6
7 import java.time.LocalDateTime;
8
9 @Data
10 @ApiModel(description = "索引库实体")
11 public class ItemDoc{
12
13     @ApiModelProperty("商品id")
14     private String id;
15
16     @ApiModelProperty("商品名称")
17     private String name;
18
19     @ApiModelProperty("价格（分）")
20     private Integer price;
21
22     @ApiModelProperty("商品图片")
23     private String image;
24
25     @ApiModelProperty("类目名称")
26     private String category;
27
28     @ApiModelProperty("品牌名称")
29     private String brand;
30
31     @ApiModelProperty("销量")
32     private Integer sold;
33
34     @ApiModelProperty("评论数")
35     private Integer commentCount;
36
37     @ApiModelProperty("是否是推广广告，true/false")
38     private Boolean isAD;
39
40     @ApiModelProperty("更新时间")
41     private LocalDateTime updateTime;
42 }
```

5.1.2.API语法

新增文档的请求语法如下：

```
1 POST /{索引库名}/_doc/1
2 {
3     "name": "Jack",
4     "age": 21
5 }
```

对应的JavaAPI如下：

```
@Test
void testIndexDocument() throws IOException {
    // 1. 创建request对象
    IndexRequest request = new IndexRequest("indexName").id("1");
    // 2. 准备JSON文档
    request.source("{\"name\": \"Jack\", \"age\": 21}", XContentType.JSON);
    // 3. 发送请求
    client.index(request, RequestOptions.DEFAULT);
}
```

索引库名、文档id
POST /indexName/_doc/1

{
 "name": "Jack",
 "age": 21
} Json文档

可以看到与索引库操作的API非常类似，同样是三步走：

- 1) 创建Request对象，这里是 `IndexRequest`，因为添加文档就是创建倒排索引的过程
- 2) 准备请求参数，本例中就是Json文档
- 3) 发送请求

变化的地方在于，这里直接使用 `client.xxx()` 的API，不再需要 `client.indices()` 了。

5.1.3.完整代码

我们导入商品数据，除了参考API模板“三步走”以外，还需要做几点准备工作：

- 商品数据来自于数据库，我们需要先查询出来，得到 `Item` 对象
- `Item` 对象需要转为 `ItemDoc` 对象
- `ItemDTO` 需要序列化为 `json` 格式

因此，代码整体步骤如下：

- 1) 根据id查询商品数据 `Item`

- 2) 将 `Item` 封装为 `ItemDoc`
- 3) 将 `ItemDoc` 序列化为JSON
- 4) 创建`IndexRequest`，指定索引库名和id
- 5) 准备请求参数，也就是JSON文档
- 6) 发送请求

在 `item-service` 的 `DocumentTest` 测试类中，编写单元测试：

```
1 @Test
2 void testAddDocument() throws IOException {
3     // 1.根据id查询商品数据
4     Item item = itemService.getById(1000002644680L);
5     // 2.转换为文档类型
6     ItemDoc itemDoc = BeanUtil.copyProperties(item, ItemDoc.class);
7     // 3.将ItemDTO转json
8     String doc = JSONUtil.toJsonStr(itemDoc);
9
10    // 1.准备Request对象
11    IndexRequest request = new IndexRequest("items").id(itemDoc.getId());
12    // 2.准备Json文档
13    request.source(doc, XContentType.JSON);
14    // 3.发送请求
15    client.index(request, RequestOptions.DEFAULT);
16 }
```

5.2.查询文档

我们以根据id查询文档为例

5.2.1.语法说明

查询的请求语句如下：

```
1 GET /{索引库名}/_doc/{id}
```

与之前的流程类似，代码大概分2步：

- 创建Request对象
- 准备请求参数，这里是无参，直接省略
- 发送请求

不过查询的目的是得到结果，解析为ItemDTO，还要再加一步对结果的解析。示例代码如下：

```
@Test
void testGetDocumentById() throws IOException {
    // 1. 创建request对象
    GetRequest request = new GetRequest("indexName", "1");
    // 2. 发送请求，得到结果
    GetResponse response = client.get(request, RequestOptions.DEFAULT);
    // 3. 解析结果
    String json = response.getSourceAsString();

    System.out.println(json);
}
```

GET /indexName/_doc/1

```
{
  "_index" : "users",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "_seq_no" : 0,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "name" : "Jack",
    "age" : 21
  }
}
```

可以看到，响应结果是一个JSON，其中文档放在一个 `_source` 属性中，因此解析就是拿到 `_source`，反序列化为Java对象即可。

其它代码与之前类似，流程如下：

- 1) 准备Request对象。这次是查询，所以是 `GetRequest`
- 2) 发送请求，得到结果。因为是查询，这里调用 `client.get()` 方法
- 3) 解析结果，就是对JSON做反序列化

5.2.2.完整代码

在 `item-service` 的 `DocumentTest` 测试类中，编写单元测试：

```
1 @Test
2 void testGetDocumentById() throws IOException {
3     // 1. 准备Request对象
4     GetRequest request = new GetRequest("items").id("100002644680");
5     // 2. 发送请求
6     GetResponse response = client.get(request, RequestOptions.DEFAULT);
7     // 3. 获取响应结果中的source
8     String json = response.getSourceAsString();
9 }
```

```
10     ItemDoc itemDoc = JSONUtil.toBean(json, ItemDoc.class);
11     System.out.println("itemDoc= " + ItemDoc);
12 }
```

5.3.删除文档

删除的请求语句如下：

```
1 DELETE /hotel/_doc/{id}
```

与查询相比，仅仅是请求方式从 `DELETE` 变成 `GET`，可以想象Java代码应该依然是2步走：

- 1) 准备Request对象，因为是删除，这次是 `DeleteRequest` 对象。要指定索引库名和id
- 2) 准备参数，无参，直接省略
- 3) 发送请求。因为是删除，所以是 `client.delete()` 方法

在 `item-service` 的 `DocumentTest` 测试类中，编写单元测试：

```
1 @Test
2 void testDeleteDocument() throws IOException {
3     // 1.准备Request，两个参数，第一个是索引库名，第二个是文档id
4     DeleteRequest request = new DeleteRequest("item", "100002644680");
5     // 2.发送请求
6     client.delete(request, RequestOptions.DEFAULT);
7 }
```

5.4.修改文档

修改我们讲过两种方式：

- 全量修改：本质是先根据id删除，再新增
- 局部修改：修改文档中的指定字段值

在RestClient的API中，全量修改与新增的API完全一致，判断依据是ID：

- 如果新增时，ID已经存在，则修改
- 如果新增时，ID不存在，则新增

这里不再赘述，我们主要关注局部修改的API即可。

5.4.1.语法说明

局部修改的请求语法如下：

```
1 POST /{索引库名}/_update/{id}
2 {
3   "doc": {
4     "字段名": "字段值",
5     "字段名": "字段值"
6   }
7 }
```

代码示例如图：

```
@Test
void testUpdateDocumentById() throws IOException {
    // 1. 创建request对象
    UpdateRequest request = new UpdateRequest("indexName", "1");
    // 2. 准备参数，每2个参数为一对 key value
    request.doc(
        "age", 18,
        "name", "Rose"
    );
    // 3. 更新文档
    client.update(request, RequestOptions.DEFAULT);
}
```

POST /users/_update/1

{

索引库名、 文档id

"doc": {

"name": "Rose",

"age": 18

要修改的字段

}

与之前类似，也是三步走：

- 1) 准备 Request 对象。这次是修改，所以是 UpdateRequest
- 2) 准备参数。也就是JSON文档，里面包含要修改的字段
- 3) 更新文档。这里调用 client.update() 方法

5.4.2.完整代码

在 `item-service` 的 `DocumentTest` 测试类中，编写单元测试：

```
1 @Test
2 void testUpdateDocument() throws IOException {
3     // 1.准备Request
4     UpdateRequest request = new UpdateRequest("items", "100002644680");
5     // 2.准备请求参数
6     request.doc(
7         "price", 58800,
8         "commentCount", 1
9     );
10    // 3.发送请求
11    client.update(request, RequestOptions.DEFAULT);
12 }
```

5.5.批量导入文档

在之前的案例中，我们都是操作单个文档。而数据库中的商品数据实际会达到数十万条，某些项目中可能达到数百万条。

我们如果要将这些数据导入索引库，肯定不能逐条导入，而是采用批处理方案。常见的方案有：

- 利用Logstash批量导入
 - 需要安装Logstash
 - 对数据的再加工能力较弱
 - 无需编码，但要学习编写Logstash导入配置
- 利用JavaAPI批量导入
 - 需要编码，但基于JavaAPI，学习成本低
 - 更加灵活，可以任意对数据做再加工处理后写入索引库

接下来，我们就学习下如何利用JavaAPI实现批量文档导入。

5.5.1.语法说明

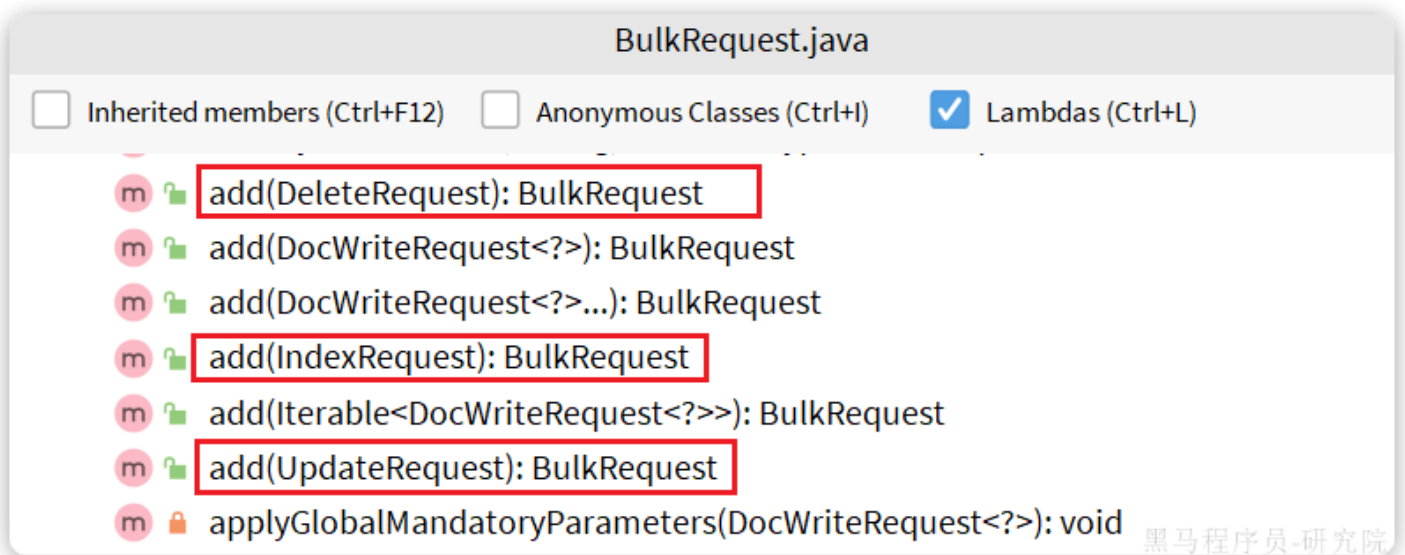
批处理与前面讲的文档的CRUD步骤基本一致：

- 创建Request，但这次用的是 `BulkRequest`
- 准备请求参数
- 发送请求，这次要用到 `client.bulk()` 方法

`BulkRequest` 本身其实并没有请求参数，其本质就是将多个普通的CRUD请求组合在一起发送。例如：

- 批量新增文档，就是给每个文档创建一个 `IndexRequest` 请求，然后封装到 `BulkRequest` 中，一起发出。
- 批量删除，就是创建N个 `DeleteRequest` 请求，然后封装到 `BulkRequest`，一起发出

因此 `BulkRequest` 中提供了 `add` 方法，用以添加其它CRUD的请求：



可以看到，能添加的请求有：

- `IndexRequest`，也就是新增
- `UpdateRequest`，也就是修改
- `DeleteRequest`，也就是删除

因此Bulk中添加了多个 `IndexRequest`，就是批量新增功能了。示例：

```
1 @Test
2 void testBulk() throws IOException {
3     // 1.创建Request
4     BulkRequest request = new BulkRequest();
5     // 2.准备请求参数
6     request.add(new IndexRequest("items").id("1").source("json doc1",
7         XContentType.JSON));
8     request.add(new IndexRequest("items").id("2").source("json doc2",
9         XContentType.JSON));
10    // 3.发送请求
```

```
9     client.bulk(request, RequestOptions.DEFAULT);
10 }
```

5.5.2.完整代码

当我们要导入商品数据时，由于商品数量达到数十万，因此不可能一次性全部导入。建议采用循环遍历方式，每次导入1000条左右的数据。

`item-service` 的 `DocumentTest` 测试类中，编写单元测试：

```
1 @Test
2 void testLoadItemDocs() throws IOException {
3     // 分页查询商品数据
4     int pageNo = 1;
5     int size = 1000;
6     while (true) {
7         Page<Item> page = itemService.lambdaQuery().eq(Item::getStatus,
8             1).page(new Page<Item>(pageNo, size));
9         // 非空校验
10        List<Item> items = page.getRecords();
11        if (CollUtils.isEmpty(items)) {
12            return;
13        }
14        log.info("加载第{}页数据，共{}条", pageNo, items.size());
15        // 1.创建Request
16        BulkRequest request = new BulkRequest("items");
17        // 2.准备参数，添加多个新增的Request
18        for (Item item : items) {
19            // 2.1.转换为文档类型ItemDTO
20            ItemDoc itemDoc = BeanUtil.copyProperties(item, ItemDoc.class);
21            // 2.2.创建新增文档的Request对象
22            request.add(new IndexRequest()
23                .id(itemDoc.getId())
24                .source(JSONUtil.toJsonStr(itemDoc),
25                    XContentType.JSON));
26        }
27        // 3.发送请求
28        client.bulk(request, RequestOptions.DEFAULT);
29        // 翻页
30        pageNo++;
31    }
```

5.6.小结

文档操作的基本步骤：

- 初始化 `RestHighLevelClient`
- 创建`XxxRequest`。
 - `XXX`是 `Index`、`Get`、`Update`、`Delete`、`Bulk`
- 准备参数（`Index`、`Update`、`Bulk` 时需要）
- 发送请求。
 - 调用 `RestHighLevelClient#xxx()` 方法，`xxx`是 `index`、`get`、`update`、`delete`、`bulk`
- 解析结果（`Get` 时需要）

6.作业

6.1.服务拆分

搜索业务并发压力可能会比较高，目前与商品服务在一起，不方便后期优化。

需求：创建一个新的微服务，命名为 `search-service`，将搜索相关功能抽取到这个微服务中

6.2.商品查询接口

在 `item-service` 服务中提供一个根据id查询商品的功能，并编写对应的FeignClient

6.3.数据同步

每当商品服务对商品实现增删改时，索引库的数据也需要同步更新。

提示：可以考虑采用MQ异步通知实现。