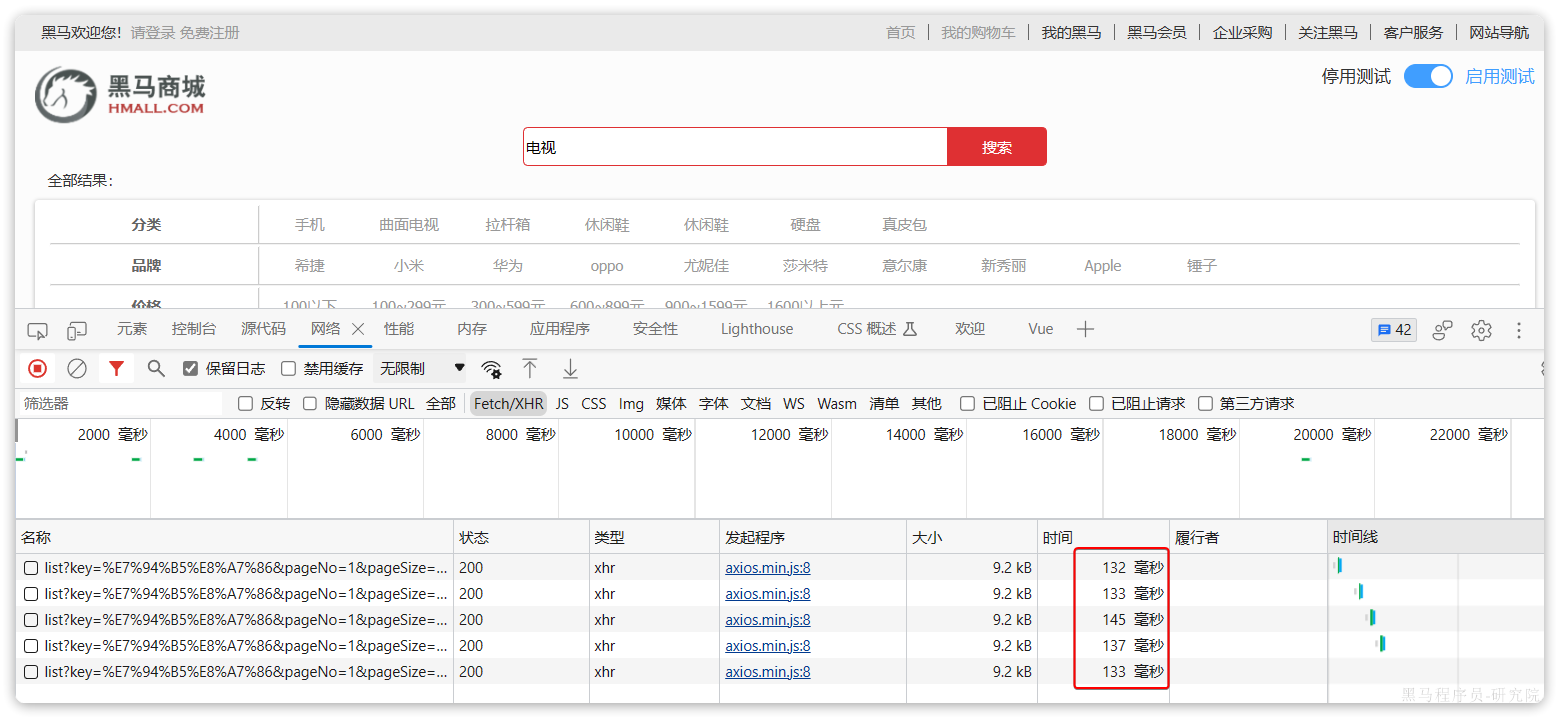
**day08-Elasticsearch**

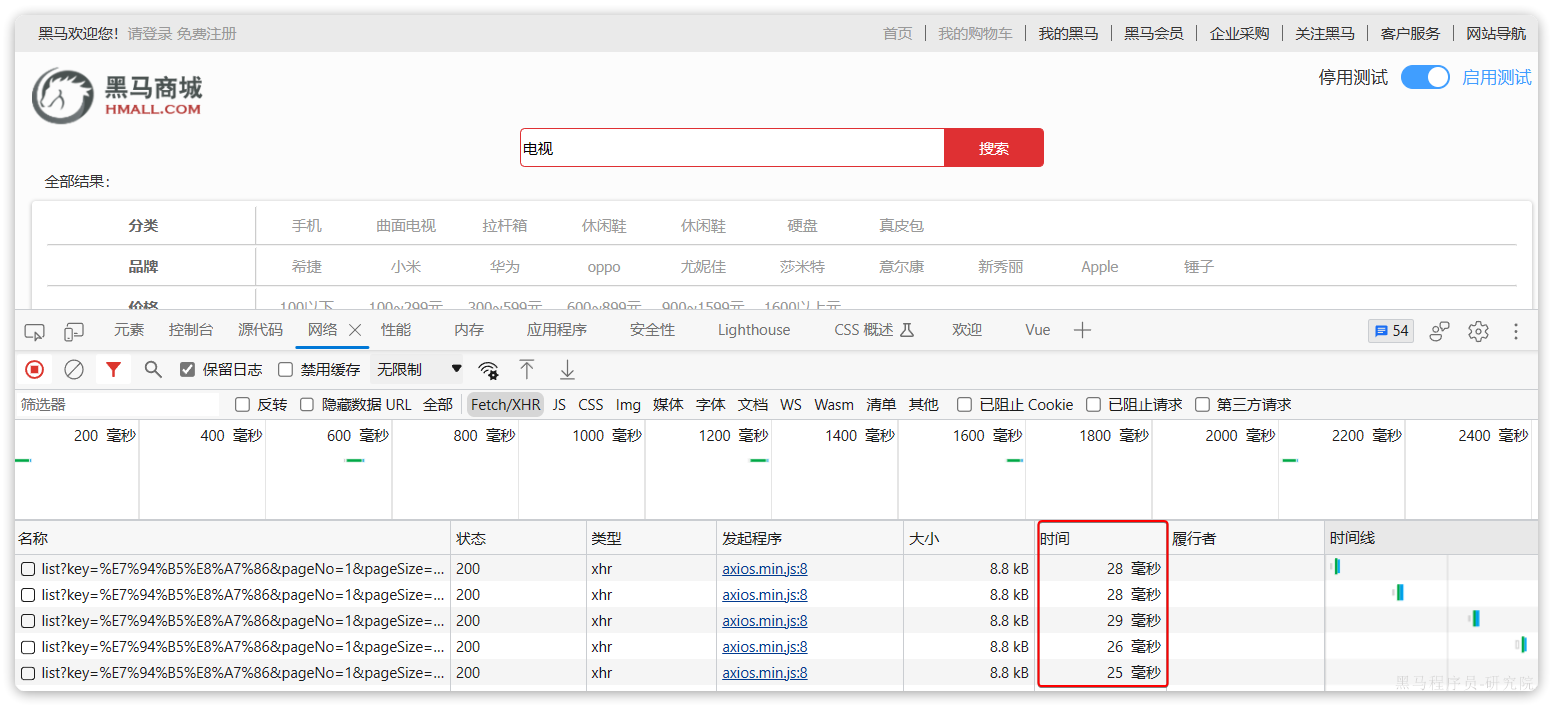
黑马商城作为一个电商项目，商品的搜索肯定是访问频率最高的页面之一。目前搜索功能是基于数据库的模糊搜索来实现的，存在很多问题。

**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：



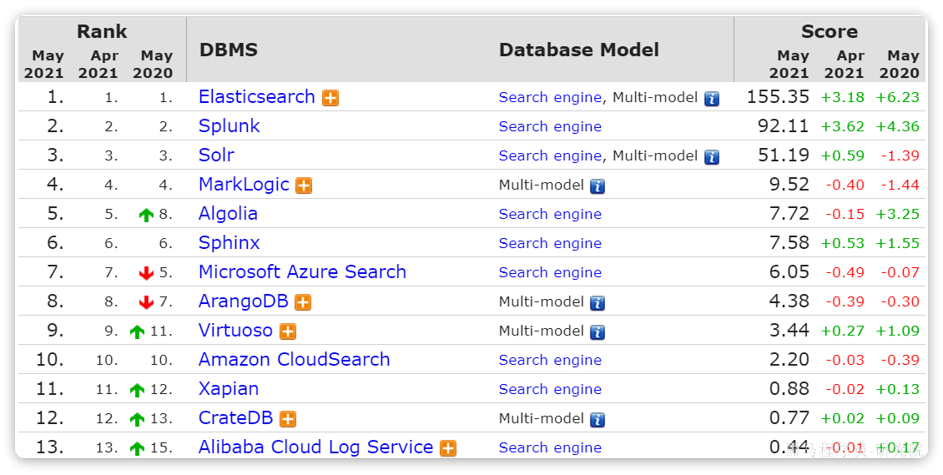
需要注意的是，数据库模糊查询随着表数据量的增多，查询性能的下降会非常明显，而搜索引擎的性能则不会随着数据增多而下降太多。目前仅10万不到的数据量差距就如此明显，如果数据量达到百万、千万、甚至上亿级别，这个性能差距会非常夸张。

**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字。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：



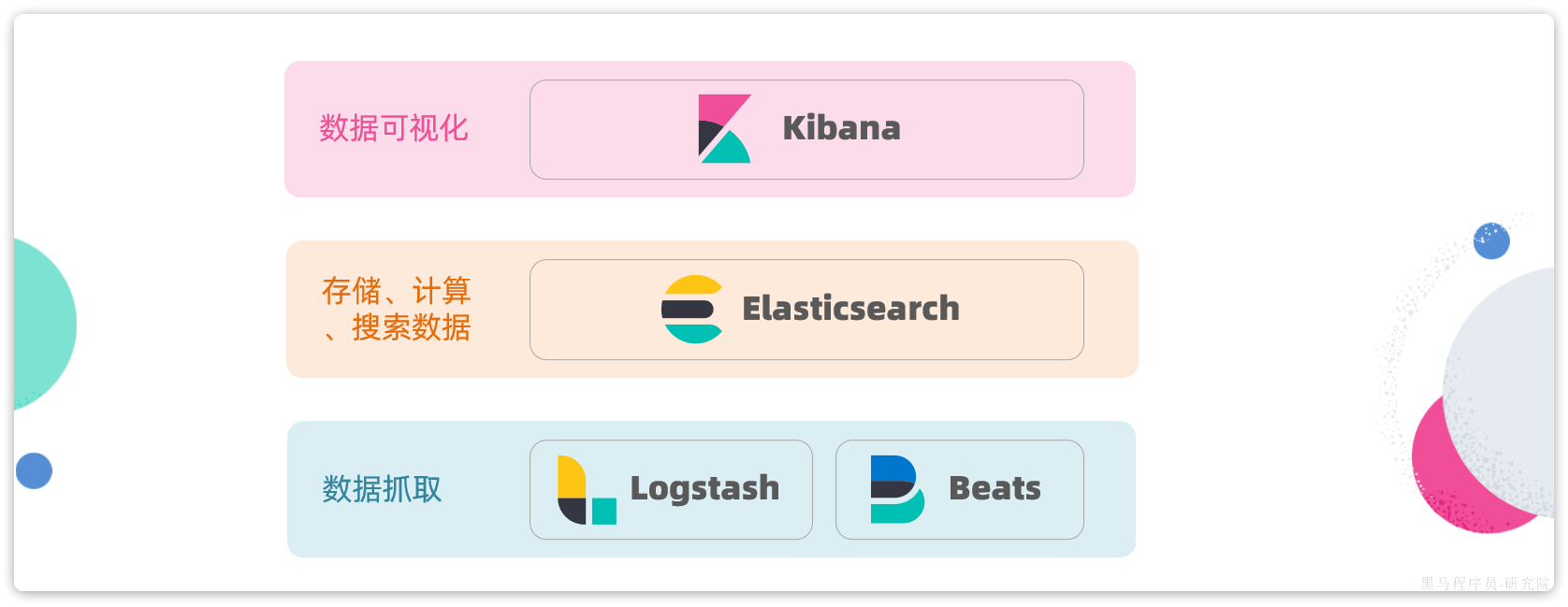
排名第一的就是我们今天要学习的elasticsearch.

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** |

另外，elasticsearch还有关联的一整套技术栈，包含Kibana、Logstash等组件，称为ELK。被广泛应用在日志数据分析、系统实时监控等领域。而elasticsearch则是这套组件中的核心：



因此，课堂中我们专注于elasticsearch的学习即可。

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是一个基于Apache Lucene库实现的，Restful风格的，分布式搜索和数据分析引擎。基于**倒排索引**技术，实现了高性能的全文检索和数据分析功能。官方网站如下：

**[该类型的内容暂不支持下载]**

本章我们就一起来初步了解一下Elasticsearch的基本原理和一些基础概念。

**1.1.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 |
| ... | ... | ... |

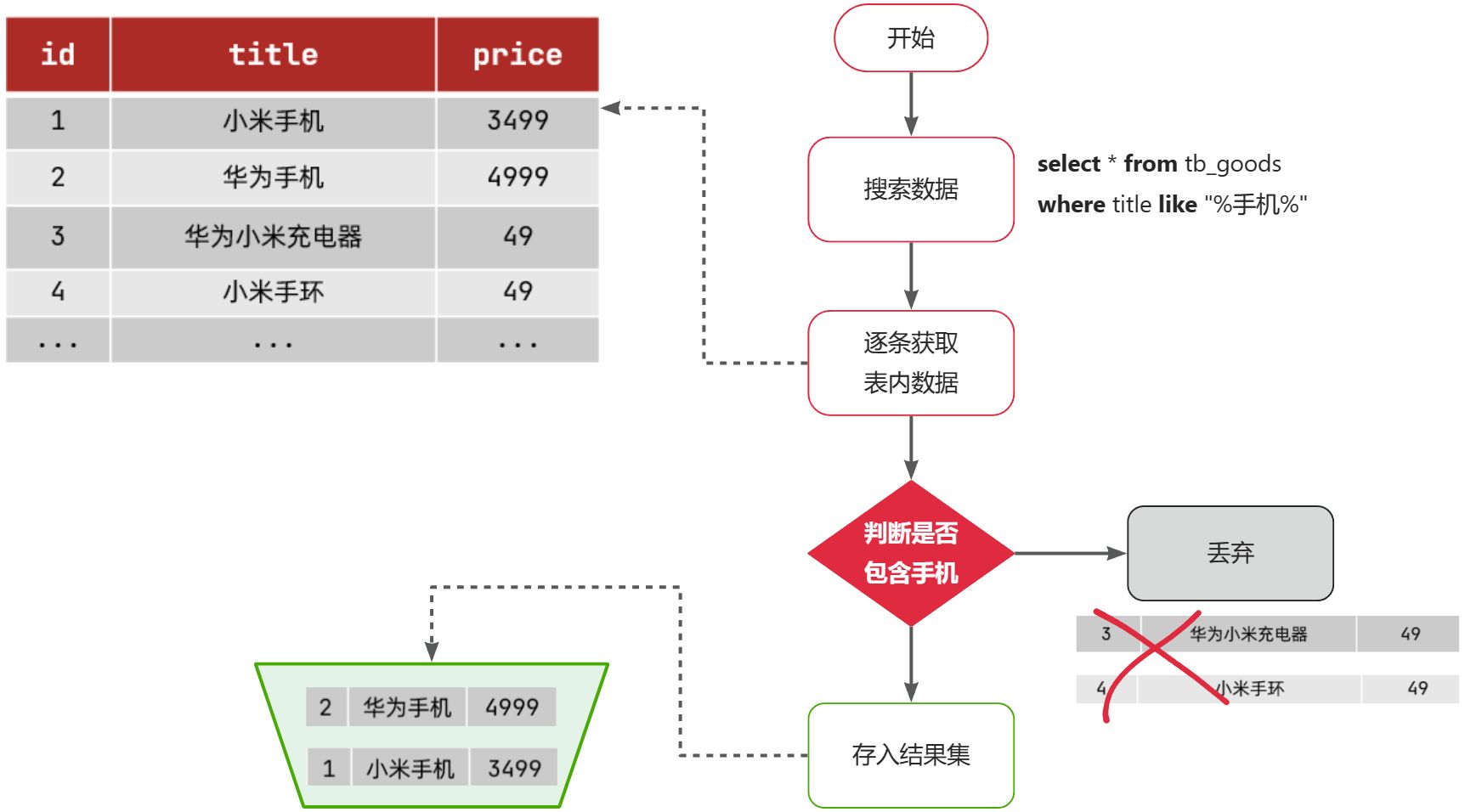
其中的id字段已经创建了索引，由于索引底层采用了B+树结构，因此我们根据id搜索的速度会非常快。但是其他字段例如title，只在叶子节点上存在。

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：

|  |
| --- |
| SQL 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，再查询倒排索引，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

**1.2.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。

**1.2.1.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：

|  |  |  |
| --- | --- | --- |
|  | |  | | --- | | JSON {  "id": 1,  "title": "小米手机",  "price": 3499 } {  "id": 2,  "title": "华为手机",  "price": 4999 } {  "id": 3,  "title": "华为小米充电器",  "price": 49 } {  "id": 4,  "title": "小米手环",  "price": 299 } | |

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）**。

**1.2.2.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：



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**   |  | | --- | | JSON {  "id": 1,  "title": "小米手机",  "price": 3499 }  {  "id": 2,  "title": "华为手机",  "price": 4999 }  {  "id": 3,  "title": "三星手机",  "price": 3999 } | | **用户索引**   |  | | --- | | JSON {  "id": 101,  "name": "张三",  "age": 21 }  {  "id": 102,  "name": "李四",  "age": 24 }  {  "id": 103,  "name": "麻子",  "age": 18 } | | **订单索引**   |  | | --- | | JSON {  "id": 10,  "userId": 101,  "goodsId": 1,  "totalFee": 294 }  {  "id": 11,  "userId": 102,  "goodsId": 2,  "totalFee": 328 } | |

* 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。

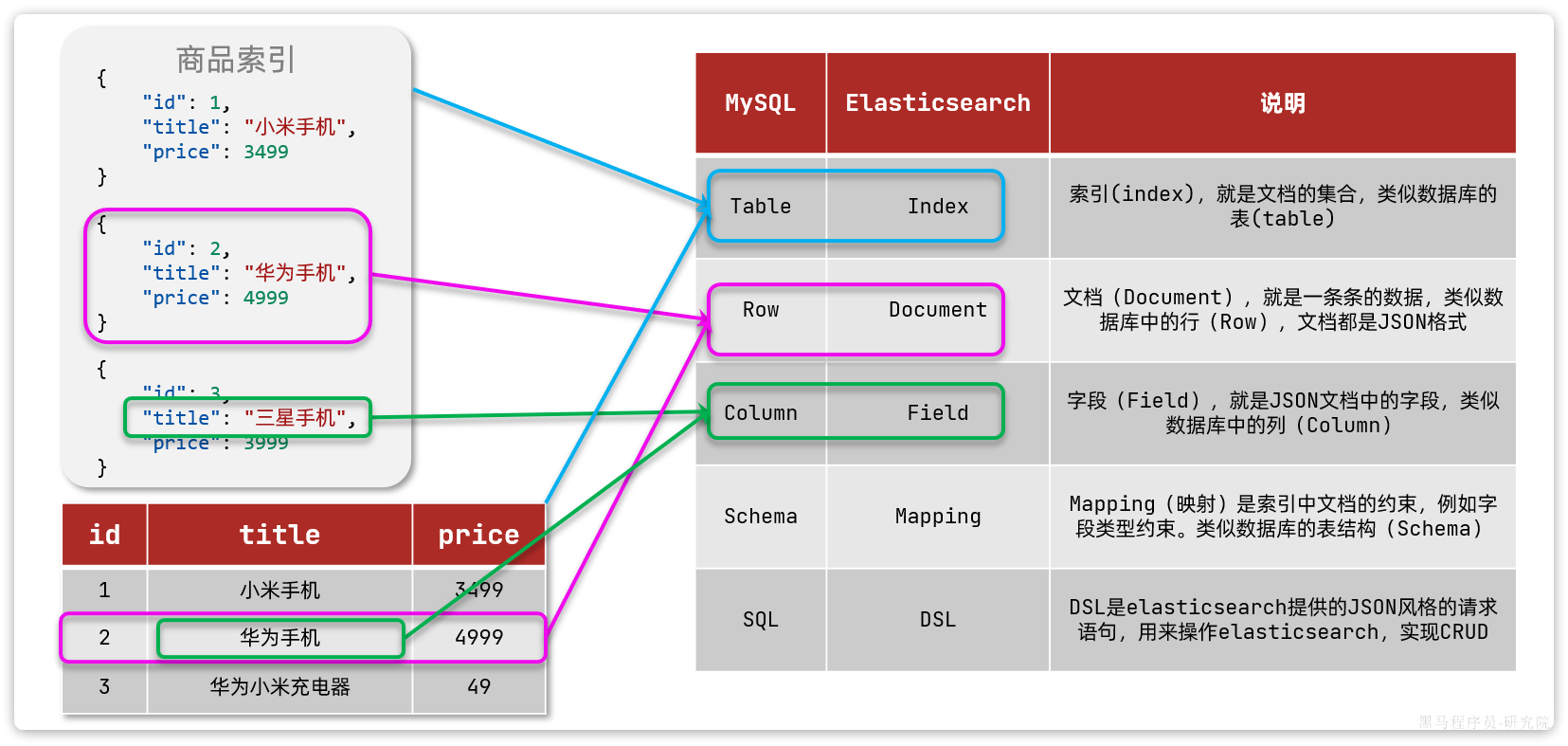
数据库的表会有约束信息，用来定义表的结构、字段的名称、类型等信息。因此，索引库中就有**映射（mapping）**，是索引中文档的字段约束信息，类似表的结构约束。

**1.2.3.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） |
| Schema | 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：



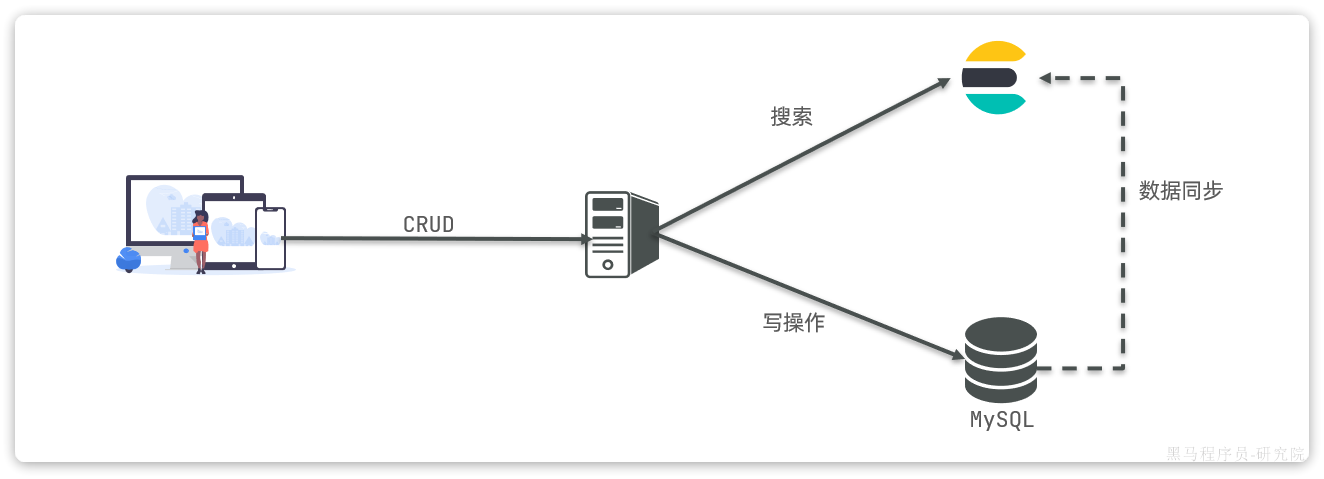
那是不是说，我们学习了elasticsearch就不再需要mysql了呢？

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

因此在企业中，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



**1.4.安装**

我们要安装的内容包含3部分：

* elasticsearch
* kibana

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**

**1.4.1.安装elasticsearch**

通过下面的Docker命令即可安装单机版本的elasticsearch：

|  |
| --- |
| Bash 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 hmall \  -p 9200:9200 \  -p 9300:9300 \  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：



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：

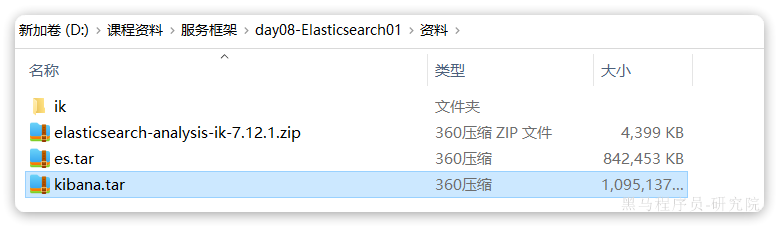


**1.4.2.安装Kibana**

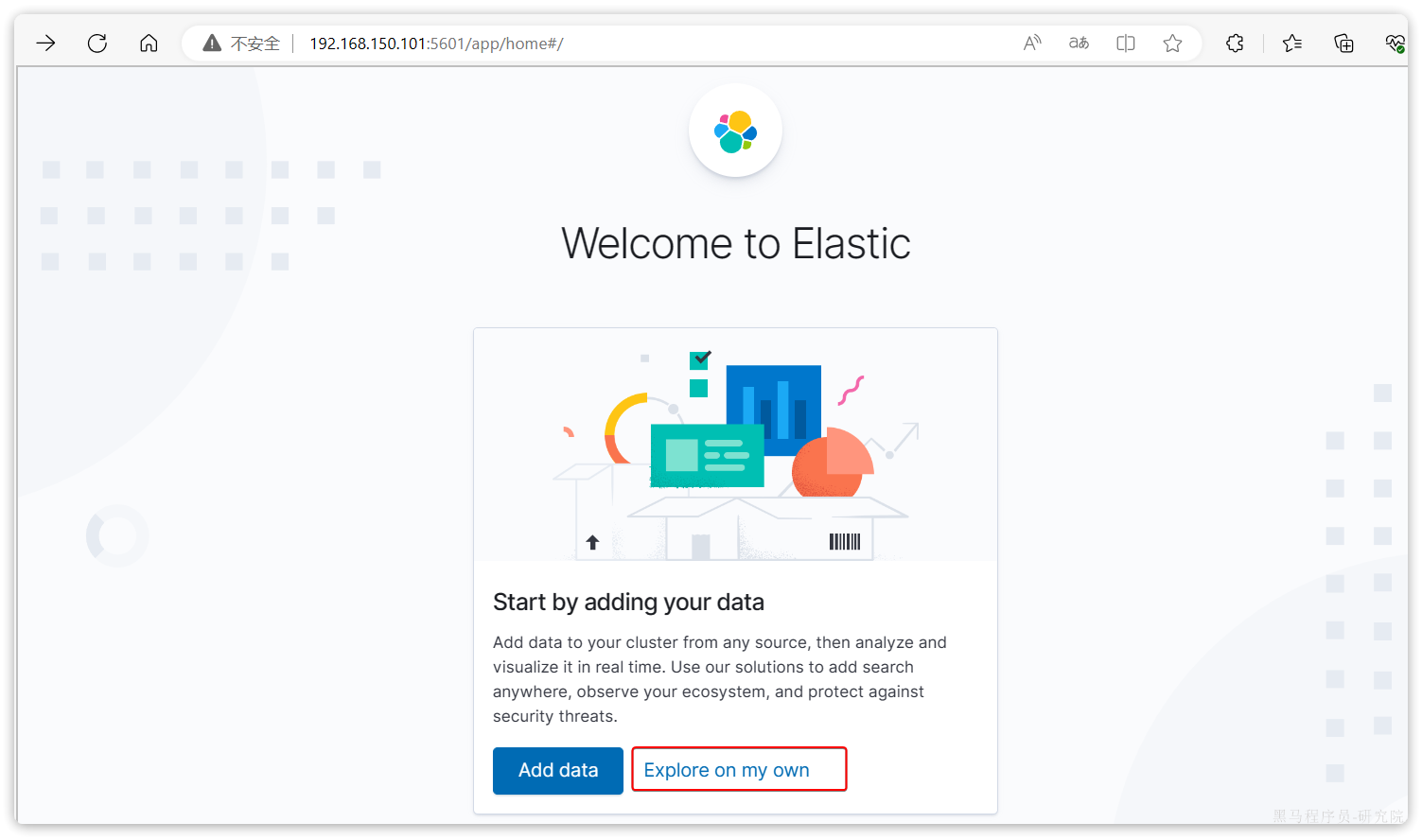
通过下面的Docker命令，即可部署Kibana：

|  |
| --- |
| Bash docker run -d \ --name kibana \ -e ELASTICSEARCH\_HOSTS=http://es:9200 \ --network=hmall \ -p 5601:5601 \ kibana:7.12.1 |

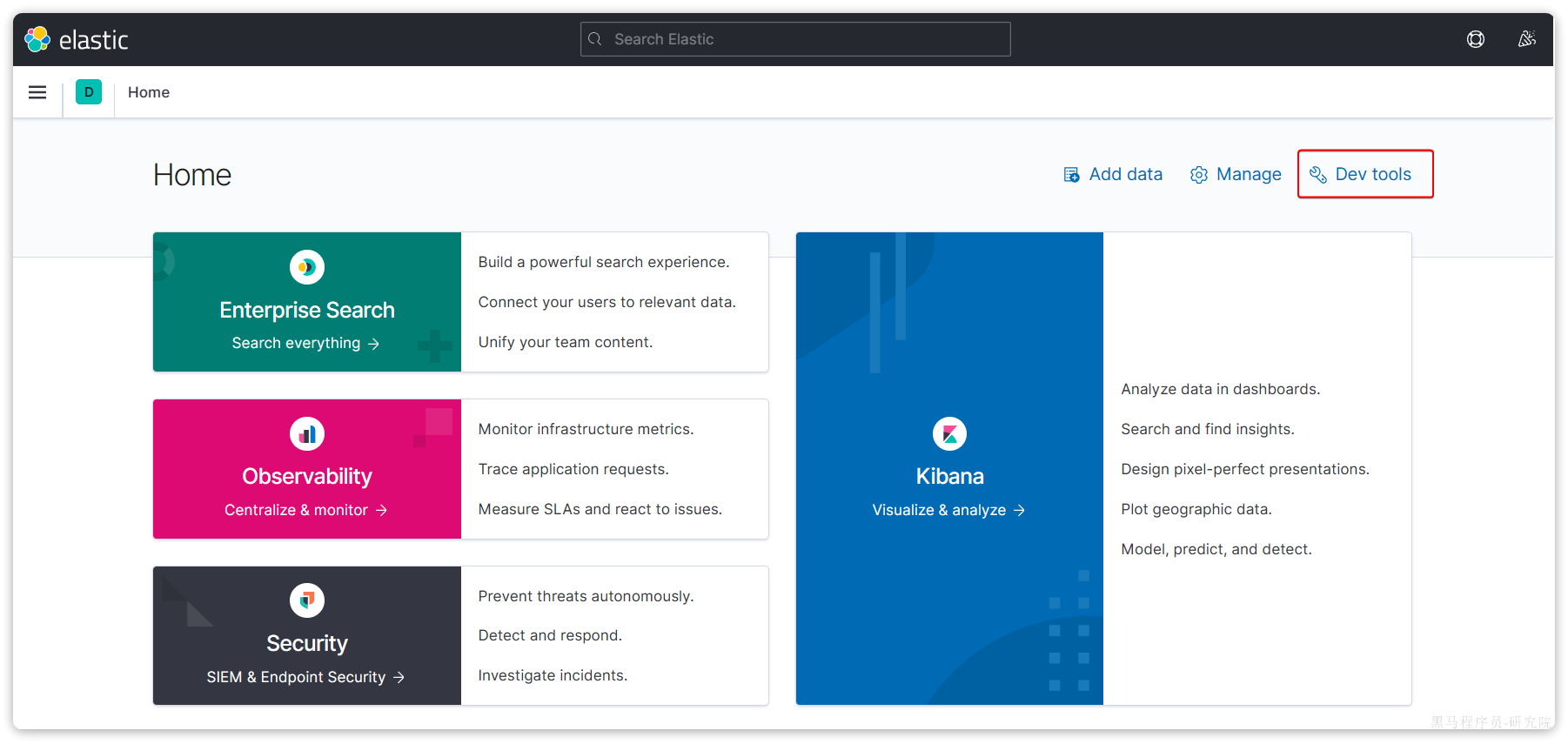
如果拉取镜像困难，可以直接导入课前资料提供的镜像tar包：



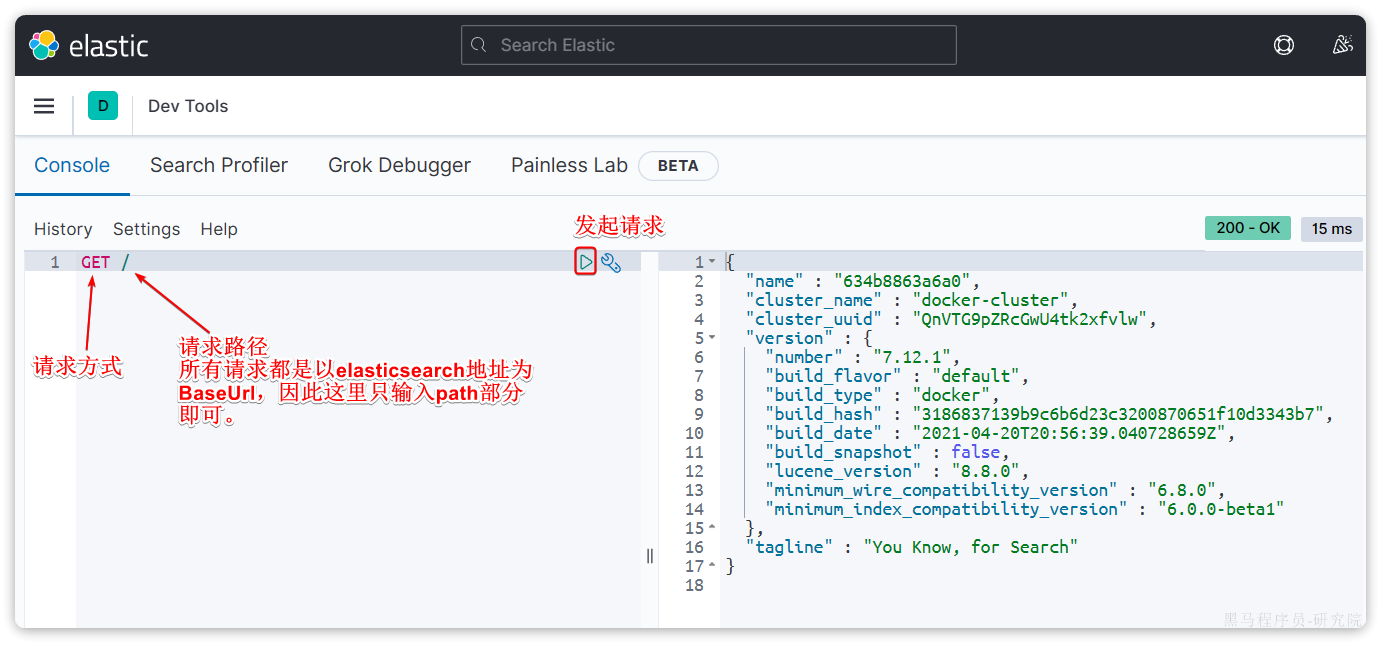
安装完成后，直接访问5601端口，即可看到控制台页面：



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：



**1.5.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。

**1.5.1.安装IK分词器**

**方案一**：在线安装

运行一个命令即可：

|  |
| --- |
| Shell 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：

|  |
| --- |
| Shell 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：

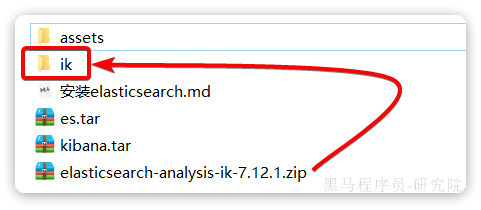
|  |
| --- |
| Shell docker volume inspect es-plugins |

结果如下：

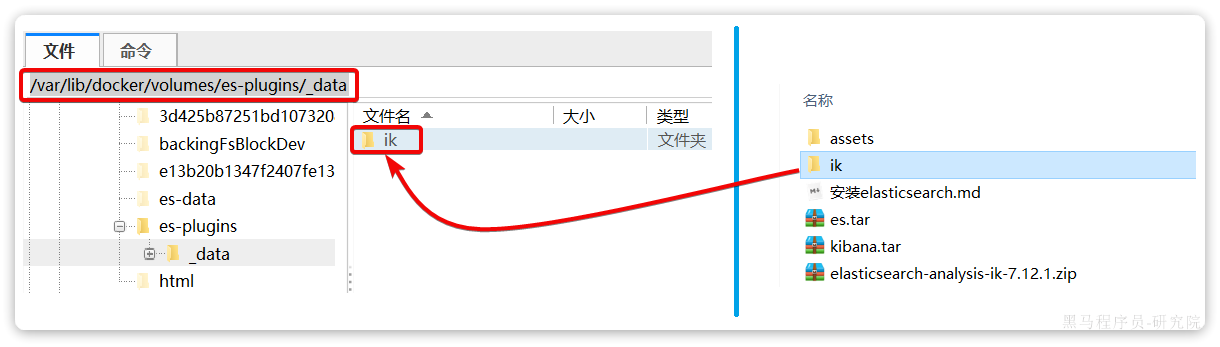
|  |
| --- |
| JSON [  {  "CreatedAt": "2024-11-06T10:06:34+08:00",  "Driver": "local",  "Labels": null,  "Mountpoint": "/var/lib/docker/volumes/es-plugins/\_data",  "Name": "es-plugins",  "Options": null,  "Scope": "local"  } ] |

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：

|  |
| --- |
| Shell docker restart es |

**1.5.2.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：

|  |
| --- |
| JSON POST /\_analyze {  "analyzer": "standard",  "text": "黑马程序员学习java太棒了" } |

结果如下：

|  |
| --- |
| JSON {  "tokens" : [  {  "token" : "黑",  "start\_offset" : 0,  "end\_offset" : 1,  "type" : "<IDEOGRAPHIC>",  "position" : 0  },  {  "token" : "马",  "start\_offset" : 1,  "end\_offset" : 2,  "type" : "<IDEOGRAPHIC>",  "position" : 1  },  {  "token" : "程",  "start\_offset" : 2,  "end\_offset" : 3,  "type" : "<IDEOGRAPHIC>",  "position" : 2  },  {  "token" : "序",  "start\_offset" : 3,  "end\_offset" : 4,  "type" : "<IDEOGRAPHIC>",  "position" : 3  },  {  "token" : "员",  "start\_offset" : 4,  "end\_offset" : 5,  "type" : "<IDEOGRAPHIC>",  "position" : 4  },  {  "token" : "学",  "start\_offset" : 5,  "end\_offset" : 6,  "type" : "<IDEOGRAPHIC>",  "position" : 5  },  {  "token" : "习",  "start\_offset" : 6,  "end\_offset" : 7,  "type" : "<IDEOGRAPHIC>",  "position" : 6  },  {  "token" : "java",  "start\_offset" : 7,  "end\_offset" : 11,  "type" : "<ALPHANUM>",  "position" : 7  },  {  "token" : "太",  "start\_offset" : 11,  "end\_offset" : 12,  "type" : "<IDEOGRAPHIC>",  "position" : 8  },  {  "token" : "棒",  "start\_offset" : 12,  "end\_offset" : 13,  "type" : "<IDEOGRAPHIC>",  "position" : 9  },  {  "token" : "了",  "start\_offset" : 13,  "end\_offset" : 14,  "type" : "<IDEOGRAPHIC>",  "position" : 10  }  ] } |

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：

|  |
| --- |
| JSON POST /\_analyze {  "analyzer": "ik\_smart",  "text": "黑马程序员学习java太棒了" } |

执行结果如下：

|  |
| --- |
| JSON {  "tokens" : [  {  "token" : "黑马",  "start\_offset" : 0,  "end\_offset" : 2,  "type" : "CN\_WORD",  "position" : 0  },  {  "token" : "程序员",  "start\_offset" : 2,  "end\_offset" : 5,  "type" : "CN\_WORD",  "position" : 1  },  {  "token" : "学习",  "start\_offset" : 5,  "end\_offset" : 7,  "type" : "CN\_WORD",  "position" : 2  },  {  "token" : "java",  "start\_offset" : 7,  "end\_offset" : 11,  "type" : "ENGLISH",  "position" : 3  },  {  "token" : "太棒了",  "start\_offset" : 11,  "end\_offset" : 14,  "type" : "CN\_WORD",  "position" : 4  }  ] } |

**1.5.3.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：

|  |
| --- |
| JSON POST /\_analyze {  "analyzer": "ik\_max\_word",  "text": "传智播客开设大学,真的泰裤辣！" } |

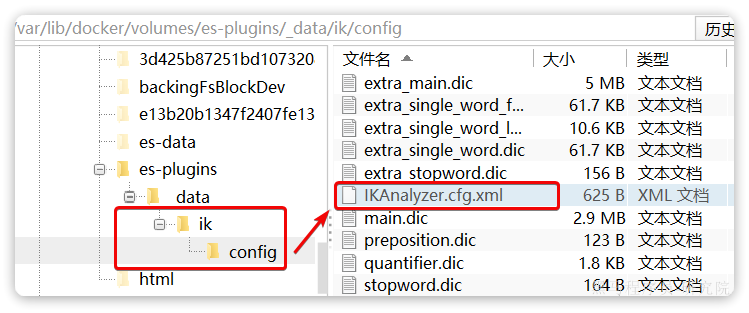
结果：

|  |
| --- |
| JSON {  "tokens" : [  {  "token" : "传",  "start\_offset" : 0,  "end\_offset" : 1,  "type" : "CN\_CHAR",  "position" : 0  },  {  "token" : "智",  "start\_offset" : 1,  "end\_offset" : 2,  "type" : "CN\_CHAR",  "position" : 1  },  {  "token" : "播",  "start\_offset" : 2,  "end\_offset" : 3,  "type" : "CN\_CHAR",  "position" : 2  },  {  "token" : "客",  "start\_offset" : 3,  "end\_offset" : 4,  "type" : "CN\_CHAR",  "position" : 3  },  {  "token" : "开设",  "start\_offset" : 4,  "end\_offset" : 6,  "type" : "CN\_WORD",  "position" : 4  },  {  "token" : "大学",  "start\_offset" : 6,  "end\_offset" : 8,  "type" : "CN\_WORD",  "position" : 5  },  {  "token" : "真的",  "start\_offset" : 9,  "end\_offset" : 11,  "type" : "CN\_WORD",  "position" : 6  },  {  "token" : "泰",  "start\_offset" : 11,  "end\_offset" : 12,  "type" : "CN\_CHAR",  "position" : 7  },  {  "token" : "裤",  "start\_offset" : 12,  "end\_offset" : 13,  "type" : "CN\_CHAR",  "position" : 8  },  {  "token" : "辣",  "start\_offset" : 13,  "end\_offset" : 14,  "type" : "CN\_CHAR",  "position" : 9  }  ] } |

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：

|  |
| --- |
| XML <?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 \*\*\* 添加扩展词典-->  <entry key="ext\_dict">ext.dic</entry> </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

|  |
| --- |
| Plain Text 传智播客 泰裤辣 |

4）重启elasticsearch

|  |
| --- |
| Shell docker restart es  # 查看 日志 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：

|  |
| --- |
| JSON {  "tokens" : [  {  "token" : "传智播客",  "start\_offset" : 0,  "end\_offset" : 4,  "type" : "CN\_WORD",  "position" : 0  },  {  "token" : "开设",  "start\_offset" : 4,  "end\_offset" : 6,  "type" : "CN\_WORD",  "position" : 1  },  {  "token" : "大学",  "start\_offset" : 6,  "end\_offset" : 8,  "type" : "CN\_WORD",  "position" : 2  },  {  "token" : "真的",  "start\_offset" : 9,  "end\_offset" : 11,  "type" : "CN\_WORD",  "position" : 3  },  {  "token" : "泰裤辣",  "start\_offset" : 11,  "end\_offset" : 14,  "type" : "CN\_WORD",  "position" : 4  }  ] } |

**1.5.4.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：

|  |
| --- |
| JSON {  "age": 21,  "weight": 52.1,  "isMarried": false,  "info": "黑马程序员Java讲师",  "email": "zy@itcast.cn",  "score": [99.1, 99.5, 98.9],  "name": {  "firstName": "云",  "lastName": "赵"  } } |

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 |  |  | —— |
| weight |  | float | 浮点数 |  |  | —— |
| isMarried |  | boolean | 布尔 |  |  | —— |
| info |  | text | 字符串，但需要分词 |  |  | IK |
| email |  | keyword | 字符串，但是不分词 |  |  | —— |
| score |  | float | 只看数组中元素类型 |  |  | —— |
| name | firstName | keyword | 字符串，但是不分词 |  |  | —— |
| lastName | keyword | 字符串，但是不分词 |  |  | —— |

**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**：

|  |
| --- |
| JSON PUT /索引库名称 {  "mappings": {  "properties": {  "字段名":{  "type": "text",  "analyzer": "ik\_smart"  },  "字段名2":{  "type": "keyword",  "index": "false"  },  "字段名3":{  "properties": {  "子字段": {  "type": "keyword"  }  }  },  // ...略  }  } } |

**示例**：

|  |
| --- |
| JSON # PUT /heima {  "mappings": {  "properties": {  "info":{  "type": "text",  "analyzer": "ik\_smart"  },  "email":{  "type": "keyword",  "index": "false"  },  "name":{  "properties": {  "firstName": {  "type": "keyword"  }  }  }  }  } } |

**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**：

|  |
| --- |
| Plain Text GET /索引库名 |

**示例**：

|  |
| --- |
| Plain Text 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**：

|  |
| --- |
| JSON PUT /索引库名/\_mapping {  "properties": {  "新字段名":{  "type": "integer"  }  } } |

**示例**：

|  |
| --- |
| JSON PUT /heima/\_mapping {  "properties": {  "age":{  "type": "integer"  }  } } |

**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：**

|  |
| --- |
| Plain Text DELETE /索引库名 |

示例：

|  |
| --- |
| Plain Text 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**

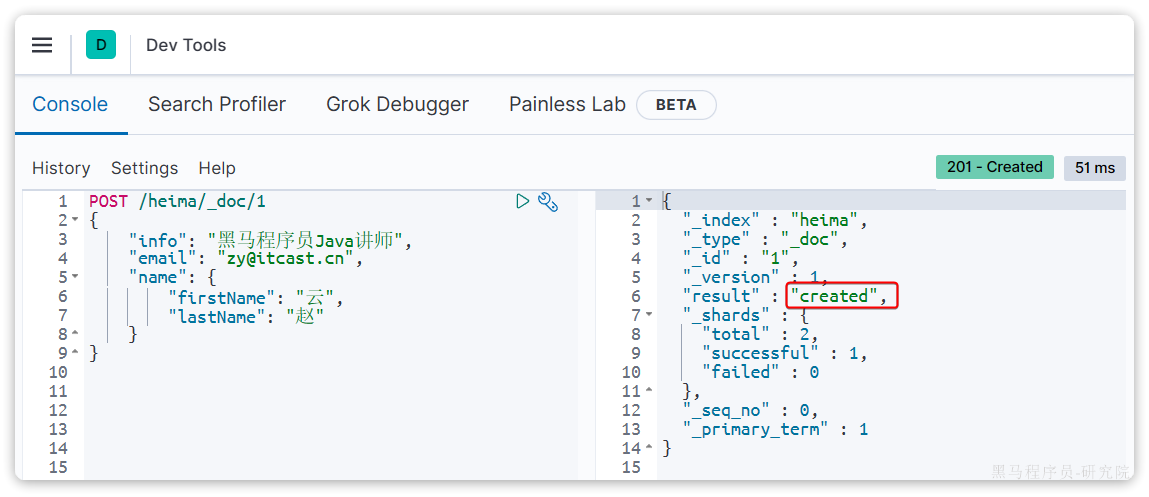
**语法：**

|  |
| --- |
| JSON 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"  }, } |

**示例：**

|  |
| --- |
| JSON POST /heima/\_doc/1 {  "info": "黑马程序员Java讲师",  "email": "zy@itcast.cn",  "name": {  "firstName": "云",  "lastName": "赵"  } } |

**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。

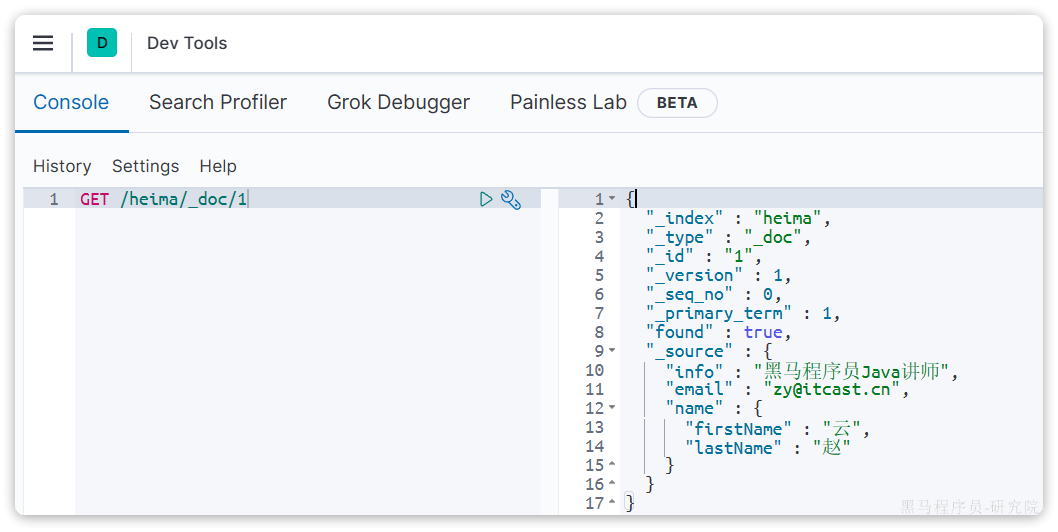
**语法：**

|  |
| --- |
| JSON GET /{索引库名称}/\_doc/{id} |

**示例：**

|  |
| --- |
| JavaScript 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：

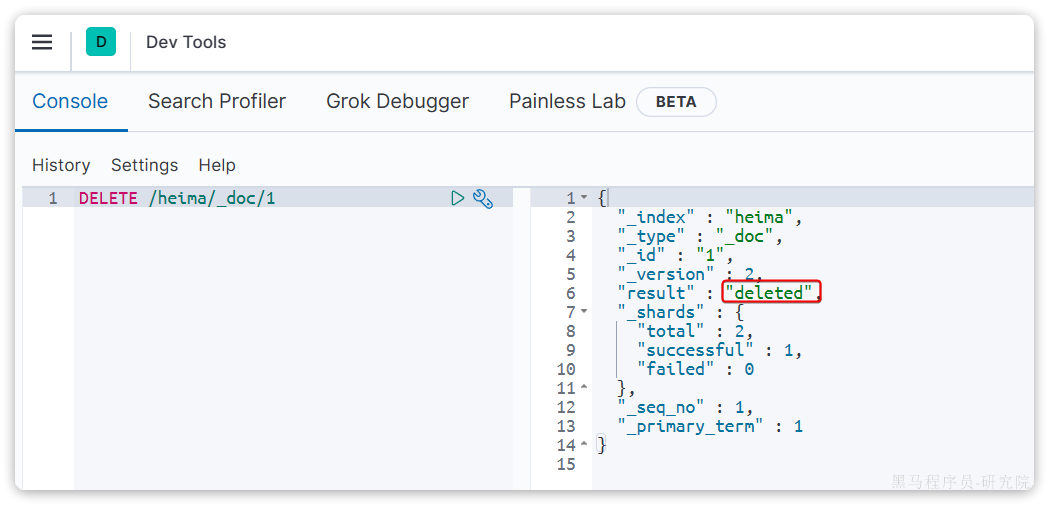
**语法：**

|  |
| --- |
| JavaScript DELETE /{索引库名}/\_doc/id值 |

**示例：**

|  |
| --- |
| JSON 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。

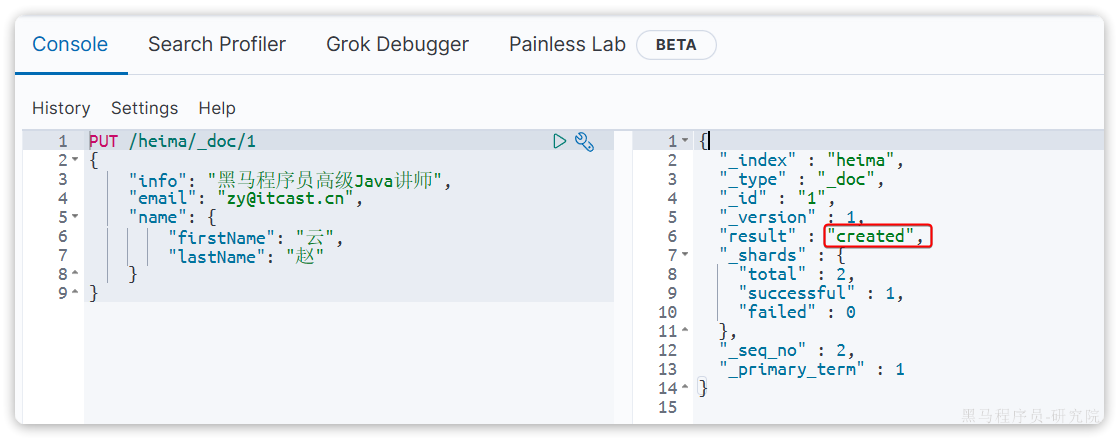
**语法：**

|  |
| --- |
| JSON PUT /{索引库名}/\_doc/文档id {  "字段1": "值1",  "字段2": "值2",  // ... 略 } |

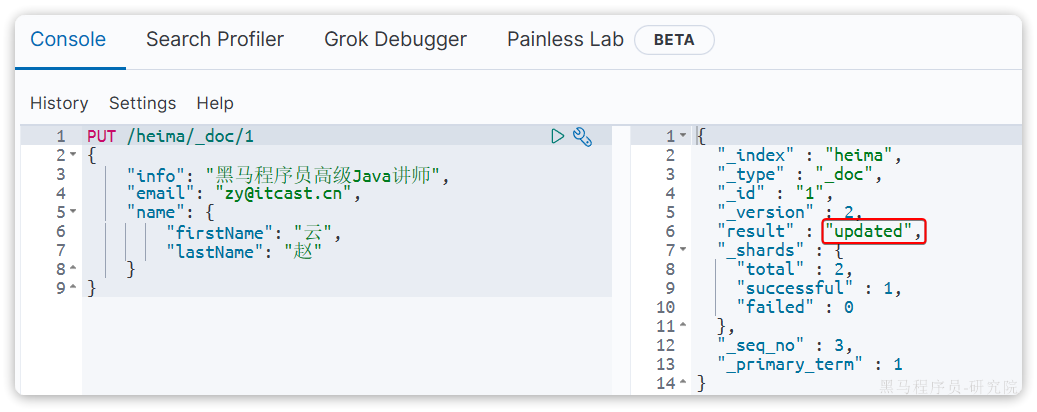
**示例：**

|  |
| --- |
| JSON PUT /heima/\_doc/1 {  "info": "黑马程序员高级Java讲师",  "email": "zy@itcast.cn",  "name": {  "firstName": "云",  "lastName": "赵"  } } |

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。

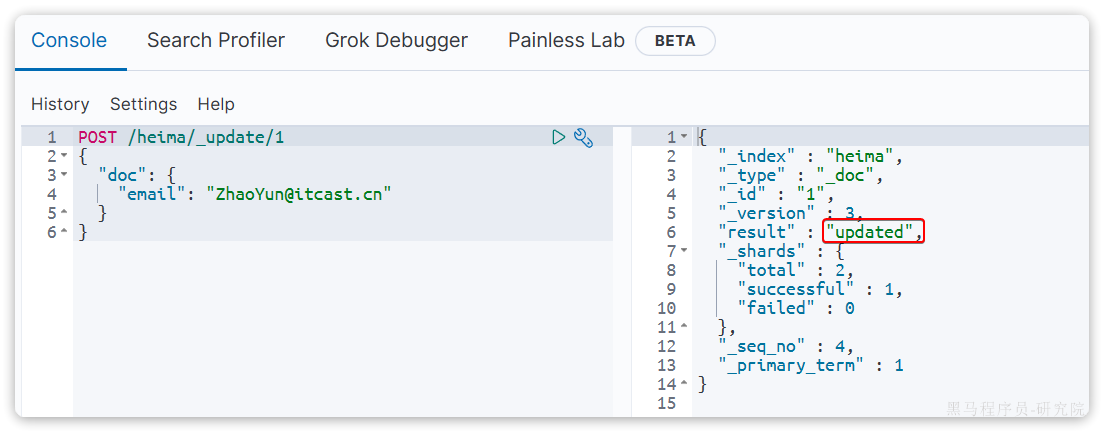
**语法：**

|  |
| --- |
| JSON POST /{索引库名}/\_update/文档id {  "doc": {  "字段名": "新的值",  } } |

**示例：**

|  |
| --- |
| JSON POST /heima/\_update/1 {  "doc": {  "email": "ZhaoYun@itcast.cn"  } } |

**执行结果**：



**3.5.批处理**

批处理采用POST请求，基本语法如下：

|  |
| --- |
| Java POST \_bulk { "index" : { "\_index" : "test", "\_id" : "1" } } { "field1" : "value1" } { "delete" : { "\_index" : "test", "\_id" : "2" } } { "create" : { "\_index" : "test", "\_id" : "3" } } { "field1" : "value3" } { "update" : {"\_id" : "1", "\_index" : "test"} } { "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：

|  |
| --- |
| Java POST /\_bulk {"index": {"\_index":"hmall", "\_id": "3"}} {"info": "黑马程序员C++讲师", "email": "ww@itcast.cn", "name":{"firstName": "五", "lastName":"赵"}} {"index": {"\_index":"hmall", "\_id": "4"}} {"info": "黑马程序员前端讲师", "email": "zhangsan@itcast.cn", "name":{"firstName": "三", "lastName":"张"}} |

批量删除：

|  |
| --- |
| Java POST /\_bulk {"delete":{"\_index":"hmall", "\_id": "3"}} {"delete":{"\_index":"hmall", "\_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}}

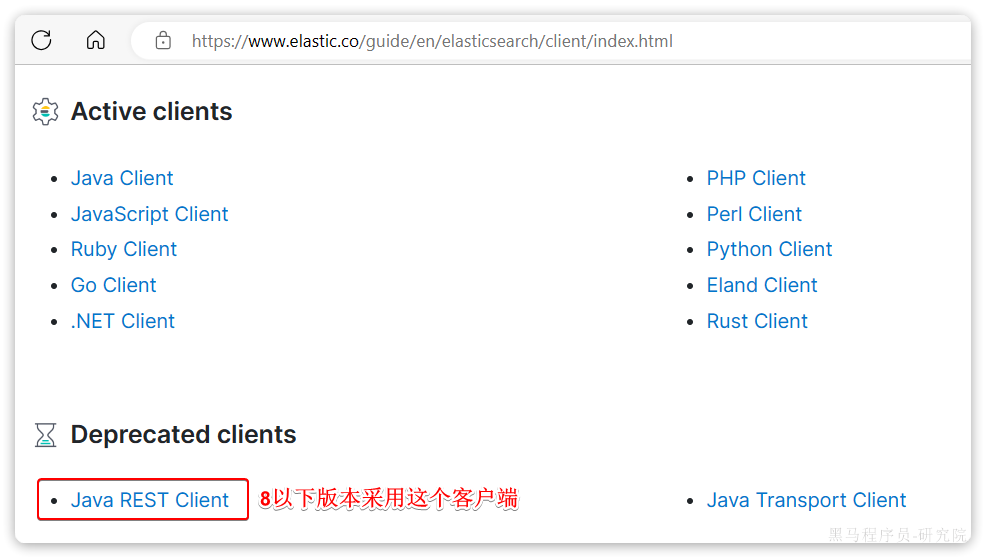
**4.RestAPI**

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。

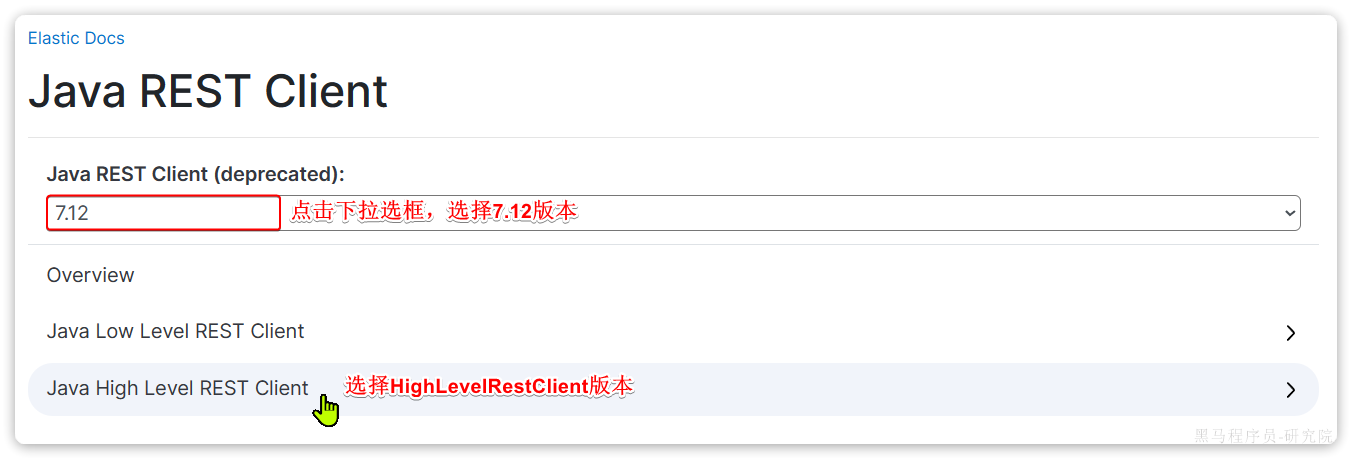
官方文档地址：

<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：

|  |
| --- |
| XML <dependency>  <groupId>org.elasticsearch.client</groupId>  <artifactId>elasticsearch-rest-high-level-client</artifactId> </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：

|  |
| --- |
| XML  <properties>  <maven.compiler.source>11</maven.compiler.source>  <maven.compiler.target>11</maven.compiler.target>  <elasticsearch.version>7.12.1</elasticsearch.version>  </properties> |

3）初始化RestHighLevelClient：

初始化的代码如下：

|  |
| --- |
| Java RestHighLevelClient client = new RestHighLevelClient(RestClient.builder(  HttpHost.create("http://192.168.150.101:9200") )); |

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：

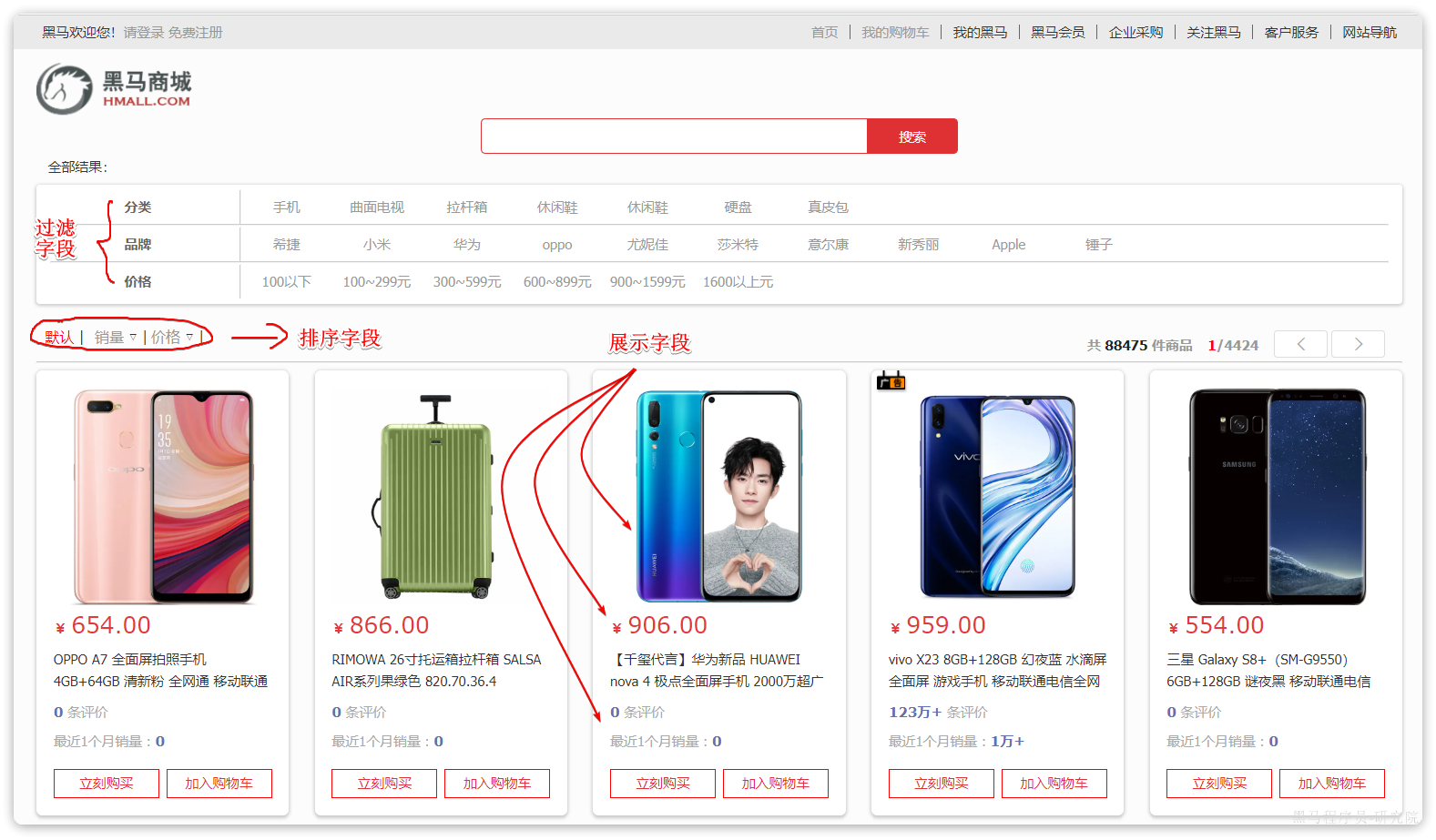
|  |
| --- |
| Java package com.hmall.item.es;  import org.apache.http.HttpHost; import org.elasticsearch.client.RestClient; import org.elasticsearch.client.RestHighLevelClient; import org.junit.jupiter.api.AfterEach; import org.junit.jupiter.api.BeforeEach; import org.junit.jupiter.api.Test;  import java.io.IOException;  public class IndexTest {   private RestHighLevelClient client;   @BeforeEach  void setUp() {  this.client = new RestHighLevelClient(RestClient.builder(  HttpHost.create("http://192.168.150.101:9200")  ));  }   @Test  void testConnect() {  System.out.println(client);  }   @AfterEach  void tearDown() throws IOException {  this.client.close();  } } |

**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：



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：



结合数据库表结构，以上字段对应的mapping映射属性如下：

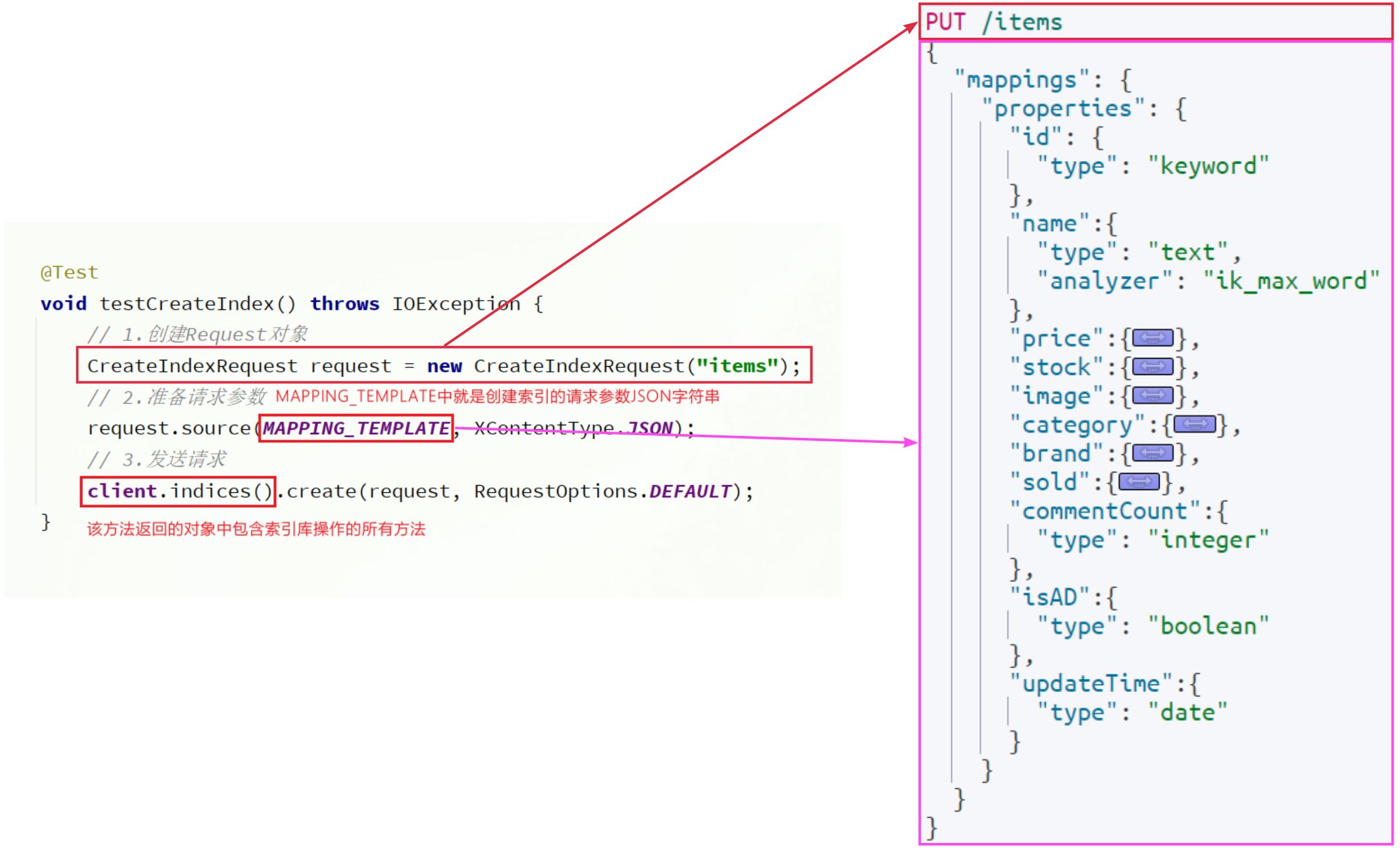
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **字段名** |  | **字段类型** | **类型说明** | **是否**  **参与搜索** | **是否**  **参与分词** | **分词器** |
| id |  | long | 长整数 |  |  | —— |
| name |  | text | 字符串，参与分词搜索 |  |  | IK |
| price |  | integer | 以分为单位，所以是整数 |  |  | —— |
| stock |  | integer | 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

因此，最终我们的索引库文档结构应该是这样：

|  |
| --- |
| JSON PUT /items {  "mappings": {  "properties": {  "id": {  "type": "keyword"  },  "name":{  "type": "text",  "analyzer": "ik\_max\_word"  },  "price":{  "type": "integer"  },  "stock":{  "type": "integer"  },  "image":{  "type": "keyword",  "index": false  },  "category":{  "type": "keyword"  },  "brand":{  "type": "keyword"  },  "sold":{  "type": "integer"  },  "commentCount":{  "type": "integer"  },  "isAD":{  "type": "boolean"  },  "updateTime":{  "type": "date"  }  }  } } |

**4.1.2.创建索引**

创建索引库的API如下：



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：

|  |
| --- |
| Java @Test void testCreateIndex() throws IOException {  // 1.创建Request对象  CreateIndexRequest request = new CreateIndexRequest("items");  // 2.准备请求参数  request.source(MAPPING\_TEMPLATE, XContentType.JSON);  // 3.发送请求  client.indices().create(request, RequestOptions.DEFAULT); }  static final String MAPPING\_TEMPLATE = "{\n" +  " \"mappings\": {\n" +  " \"properties\": {\n" +  " \"id\": {\n" +  " \"type\": \"keyword\"\n" +  " },\n" +  " \"name\":{\n" +  " \"type\": \"text\",\n" +  " \"analyzer\": \"ik\_max\_word\"\n" +  " },\n" +  " \"price\":{\n" +  " \"type\": \"integer\"\n" +  " },\n" +  " \"stock\":{\n" +  " \"type\": \"integer\"\n" +  " },\n" +  " \"image\":{\n" +  " \"type\": \"keyword\",\n" +  " \"index\": false\n" +  " },\n" +  " \"category\":{\n" +  " \"type\": \"keyword\"\n" +  " },\n" +  " \"brand\":{\n" +  " \"type\": \"keyword\"\n" +  " },\n" +  " \"sold\":{\n" +  " \"type\": \"integer\"\n" +  " },\n" +  " \"commentCount\":{\n" +  " \"type\": \"integer\"\n" +  " },\n" +  " \"isAD\":{\n" +  " \"type\": \"boolean\"\n" +  " },\n" +  " \"updateTime\":{\n" +  " \"type\": \"date\"\n" +  " }\n" +  " }\n" +  " }\n" +  "}"; |

**4.2.删除索引库**

删除索引库的请求非常简单：

|  |
| --- |
| JSON 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：

|  |
| --- |
| Java @Test void testDeleteIndex() throws IOException {  // 1.创建Request对象  DeleteIndexRequest request = new DeleteIndexRequest("items");  // 2.发送请求  client.indices().delete(request, RequestOptions.DEFAULT); } |

**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：

|  |
| --- |
| JSON 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

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

**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

|  |
| --- |
| Java package com.hmall.item.es;  import com.hmall.item.service.IItemService; import org.apache.http.HttpHost; import org.elasticsearch.client.RestClient; import org.elasticsearch.client.RestHighLevelClient; import org.junit.jupiter.api.AfterEach; import org.junit.jupiter.api.BeforeEach; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest;  import java.io.IOException;  @SpringBootTest(properties = "spring.profiles.active=local") public class DocumentTest {   private RestHighLevelClient client;  @Autowired  private IItemService itemService;   @BeforeEach  void setUp() {  this.client = new RestHighLevelClient(RestClient.builder(  HttpHost.create("http://192.168.150.101:9200")  ));  }    @AfterEach  void tearDown() throws IOException {  this.client.close();  } } |

**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：

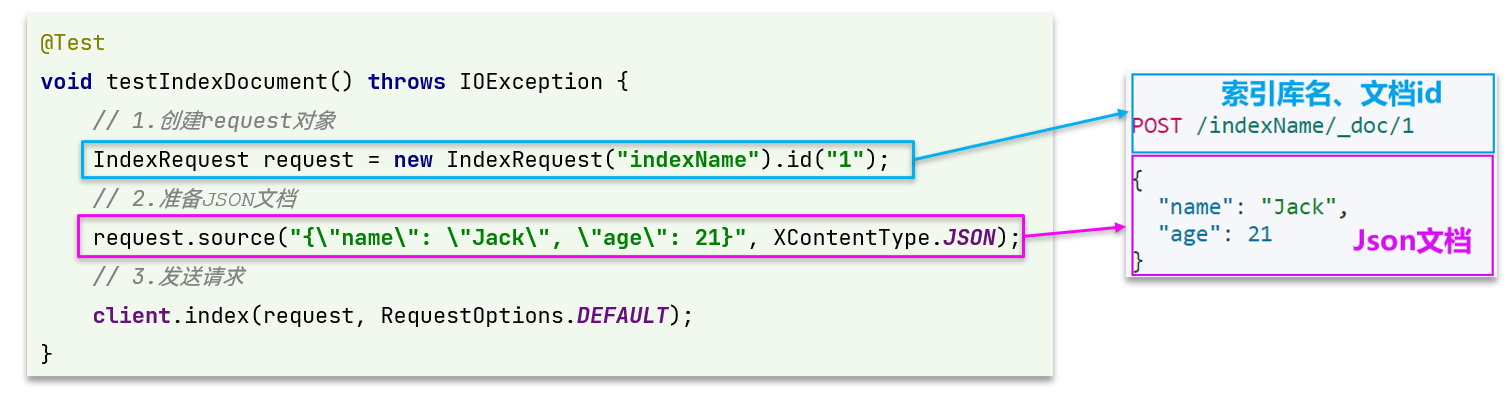
|  |
| --- |
| Java package com.hmall.item.domain.dto;  import io.swagger.annotations.ApiModel; import io.swagger.annotations.ApiModelProperty; import lombok.Data;  import java.time.LocalDateTime;  @Data @ApiModel(description = "索引库实体") public class ItemDTO{   @ApiModelProperty("商品id")  private String id;   @ApiModelProperty("商品名称")  private String name;   @ApiModelProperty("价格（分）")  private Integer price;   @ApiModelProperty("库存数量")  private Integer stock;   @ApiModelProperty("商品图片")  private String image;   @ApiModelProperty("类目名称")  private String category;   @ApiModelProperty("品牌名称")  private String brand;   @ApiModelProperty("销量")  private Integer sold;   @ApiModelProperty("评论数")  private Integer commentCount;   @ApiModelProperty("是否是推广广告，true/false")  private Boolean isAD;   @ApiModelProperty("更新时间")  private LocalDateTime updateTime; } |

**5.1.2.API语法**

新增文档的请求语法如下：

|  |
| --- |
| JSON POST /{索引库名}/\_doc/1 {  "name": "Jack",  "age": 21 } |

对应的JavaAPI如下：



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ItemDTO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ItemDTO
* 3）将ItemDTO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：

|  |
| --- |
| Java @Test void testAddDocument() throws IOException {  // 1.根据id查询商品数据  Item item = itemService.getById(100002644680L);  // 2.转换为文档类型  ItemDTO itemDTO = BeanUtil.copyProperties(item, ItemDTO.class);  // 3.将ItemDTO转json  String doc = JSONUtil.toJsonStr(itemDTO);   // 1.准备Request对象  IndexRequest request = new IndexRequest("items").id(itemDTO.getId());  // 2.准备Json文档  request.source(doc, XContentType.JSON);  // 3.发送请求  client.index(request, RequestOptions.DEFAULT); } |

**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**

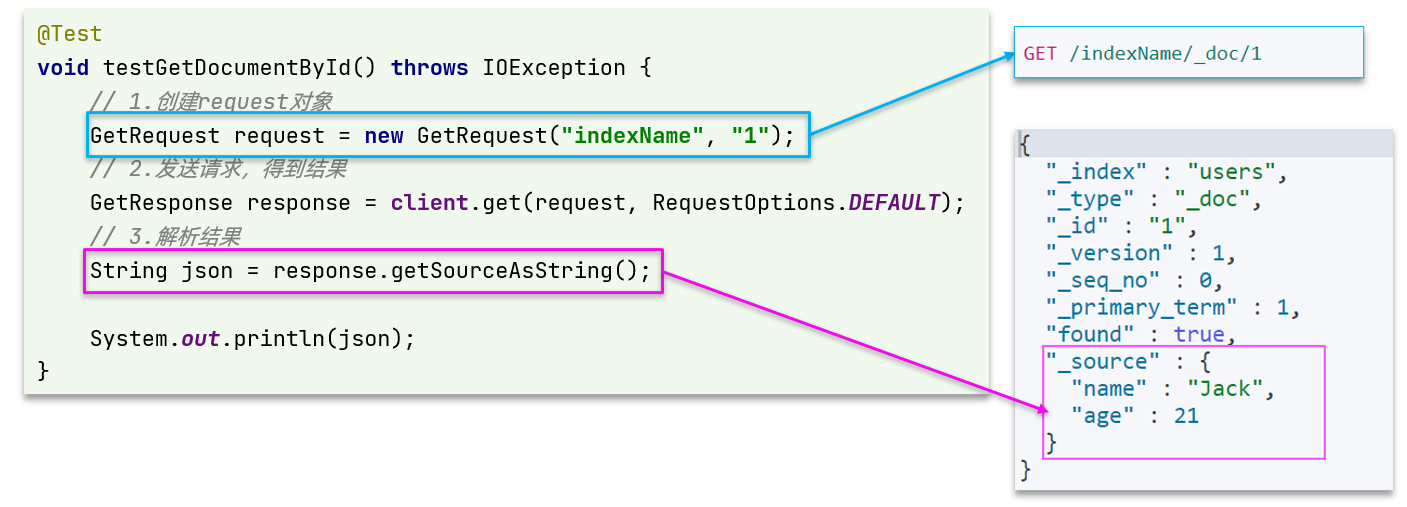
查询的请求语句如下：

|  |
| --- |
| JSON 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：



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：

|  |
| --- |
| Java @Test void testGetDocumentById() throws IOException {  // 1.准备Request对象  GetRequest request = new GetRequest("items").id("100002644680");  // 2.发送请求  GetResponse response = client.get(request, RequestOptions.DEFAULT);  // 3.获取响应结果中的source  String json = response.getSourceAsString();    ItemDTO itemDTO = JSONUtil.toBean(json, ItemDTO.class);  System.out.println("itemDTO = " + itemDTO); } |

**5.3.删除文档**

删除的请求语句如下：

|  |
| --- |
| JSON 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：

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

**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

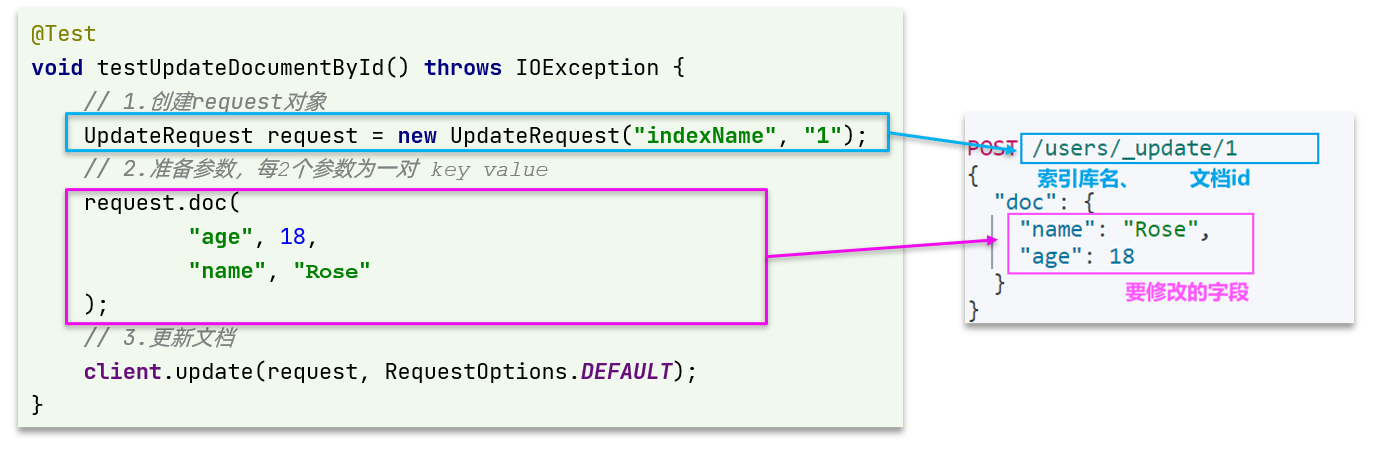
这里不再赘述，我们主要关注局部修改的API即可。

**5.4.1.语法说明**

局部修改的请求语法如下：

|  |
| --- |
| JSON POST /{索引库名}/\_update/{id} {  "doc": {  "字段名": "字段值",  "字段名": "字段值"  } } |

代码示例如图：



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：

|  |
| --- |
| Java @Test void testUpdateDocument() throws IOException {  // 1.准备Request  UpdateRequest request = new UpdateRequest("items", "100002644680");  // 2.准备请求参数  request.doc(  "price", 58800,  "commentCount", 1  );  // 3.发送请求  client.update(request, RequestOptions.DEFAULT); } |

**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**

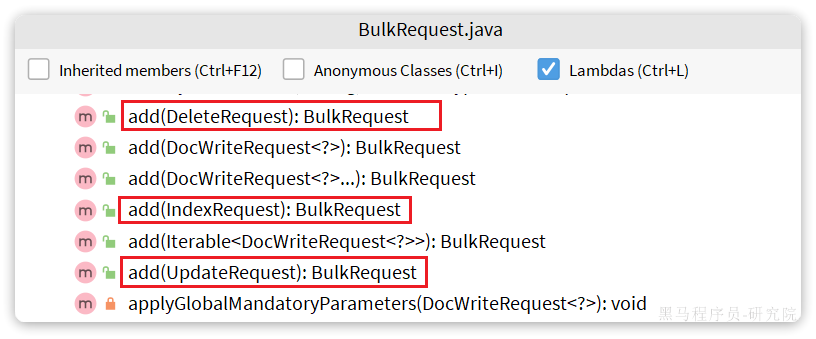
批处理与前面讲的文档的CRUD步骤基本一致：

* 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：

|  |
| --- |
| Java @Test void testBulk() throws IOException {  // 1.创建Request  BulkRequest request = new BulkRequest();  // 2.准备请求参数  request.add(new IndexRequest("items").id("1").source("json doc1", XContentType.JSON));  request.add(new IndexRequest("items").id("2").source("json doc2", XContentType.JSON));  // 3.发送请求  client.bulk(request, RequestOptions.DEFAULT); } |

**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：

|  |
| --- |
| Java @Test void testLoadItemDocs() throws IOException {  // 分页查询商品数据  int pageNo = 1;  int size = 1000;  while (true) {  Page<Item> page = itemService.lambdaQuery().eq(Item::getStatus, 1).page(new Page<Item>(pageNo, size));  // 非空校验  List<Item> items = page.getRecords();  if (CollUtils.isEmpty(items)) {  return;  }  log.info("加载第{}页数据，共{}条", pageNo, items.size());  // 1.创建Request  BulkRequest request = new BulkRequest("items");  // 2.准备参数，添加多个新增的Request  for (Item item : items) {  // 2.1.转换为文档类型ItemDTO  ItemDTO itemDTO = BeanUtil.copyProperties(item, ItemDTO.class);  // 2.2.创建新增文档的Request对象  request.add(new IndexRequest()  .id(itemDTO.getId())  .source(JSONUtil.toJsonStr(itemDTO), XContentType.JSON));  }  // 3.发送请求  client.bulk(request, RequestOptions.DEFAULT);   // 翻页  pageNo++;  } } |

**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。