

分布式搜索引擎01

-- elasticsearch基础

0.学习目标

1.初识elasticsearch

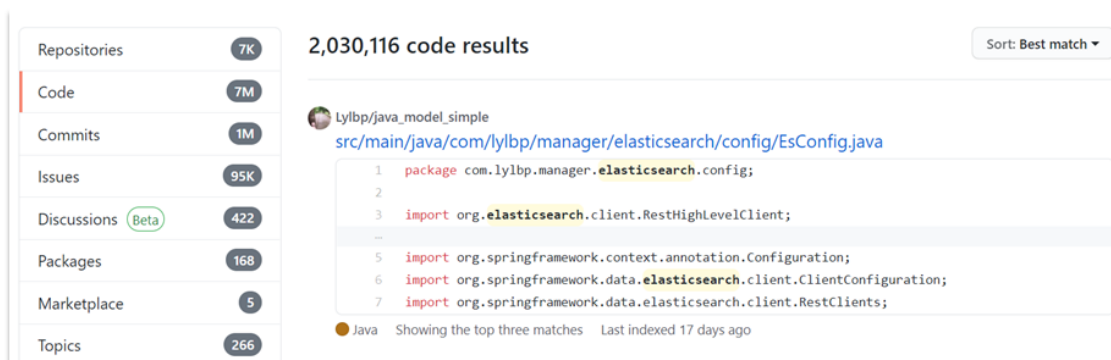
1.1.了解ES

1.1.1.elasticsearch的作用

elasticsearch是一款非常强大的开源搜索引擎，具备非常多强大功能，可以帮助我们海量数据中快速找到需要内容

例如：

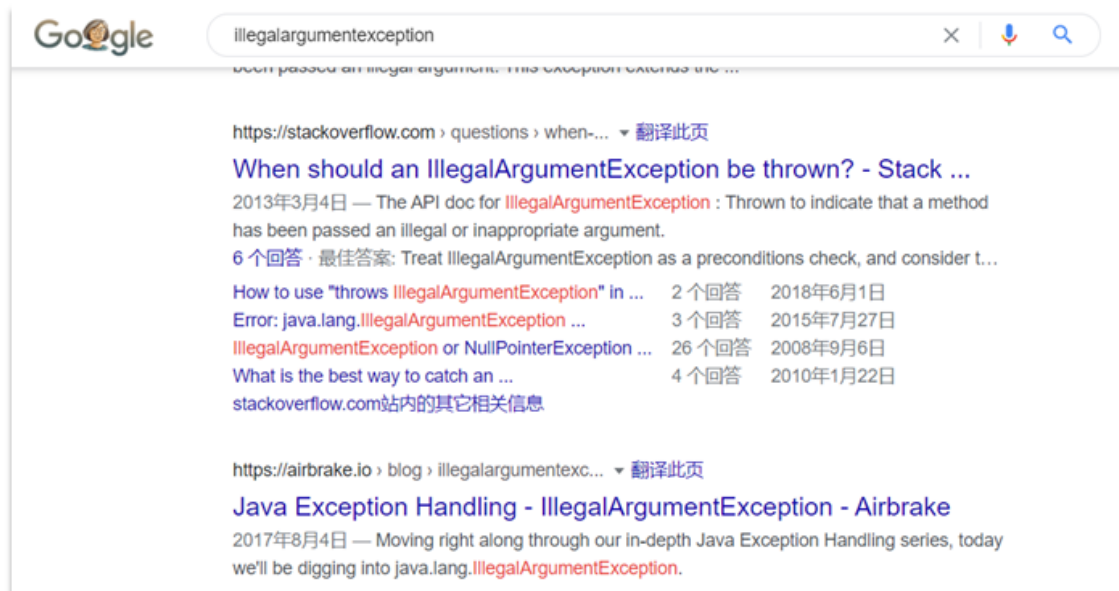
- 在GitHub搜索代码



- 在电商网站搜索商品



- 在百度搜索答案



- 在打车软件搜索附近的车

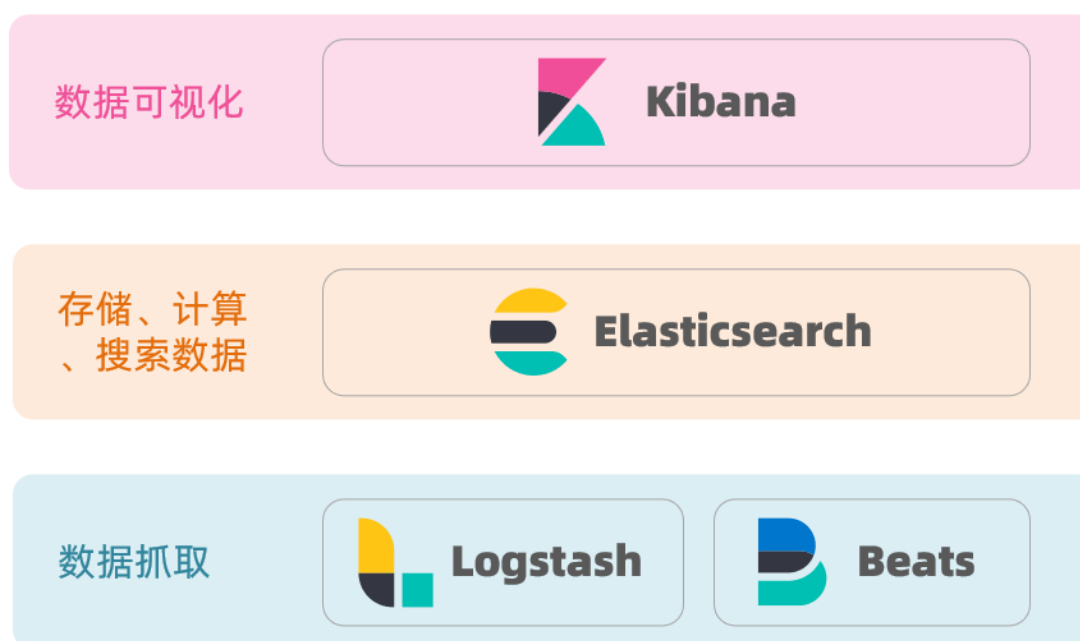


1.1.2.ELK技术栈

elasticsearch结合kibana、Logstash、Beats，也就是elastic stack（ELK）。被广泛应用在日志数据分析、实时监控等领域：



而elasticsearch是elastic stack的核心，负责存储、搜索、分析数据。



1.1.3.elasticsearch和lucene

elasticsearch底层是基于lucene来实现的。

Lucene是一个Java语言的搜索引擎类库，是Apache公司的顶级项目，由DougCutting于1999年研发。
官网地址：<https://lucene.apache.org/>。

Lucene的优势：

- 易扩展
- 高性能（基于倒排索引）

Lucene的缺点：

- 只限于Java语言开发
- 学习曲线陡峭
- 不支持水平扩展



elasticsearch的发展历史：

- 2004年Shay Banon基于Lucene开发了Compass
- 2010年Shay Banon 重写了Compass，取名为Elasticsearch。

官网地址: <https://www.elastic.co/cn/> , 目前最新的版本是: 7.12.1







相比与lucene, elasticsearch具备下列优势：

- 支持分布式，可水平扩展
- 提供Restful接口，可被任何语言调用

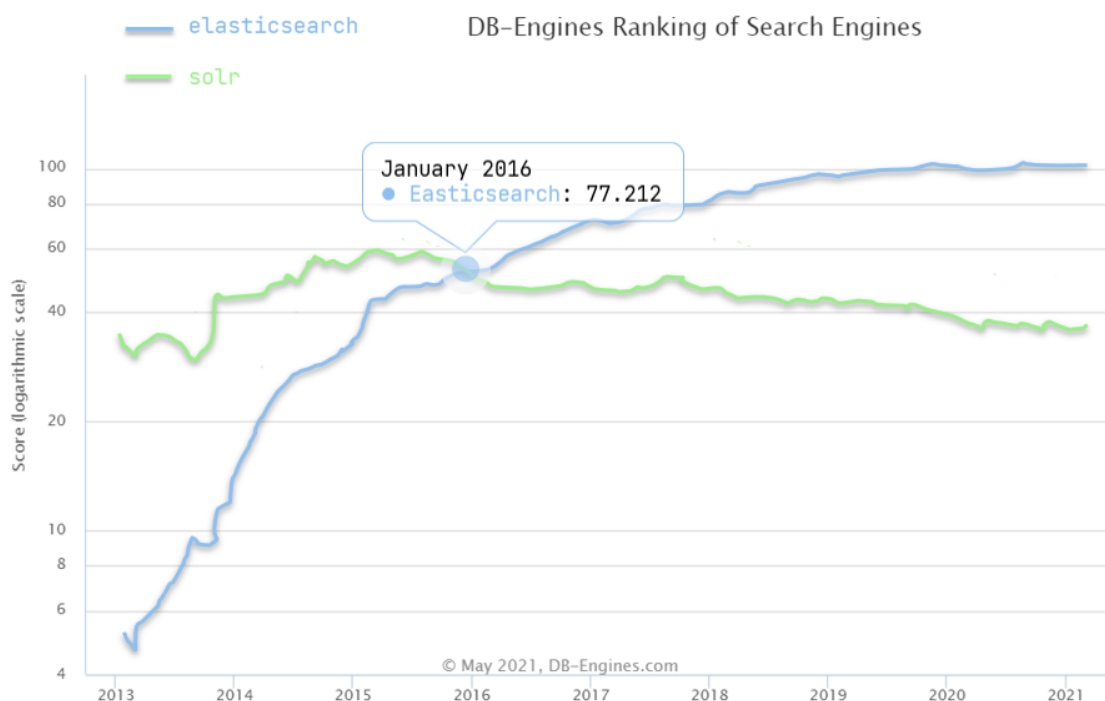


1.1.4.为什么不是其他搜索技术？

目前比较知名的搜索引擎技术排名：

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

虽然在早期，Apache Solr是最主要的搜索引擎技术，但随着发展elasticsearch已经渐渐超越了Solr，独占鳌头：



1.1.5.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 (ELK) ?

- 是以elasticsearch为核心的技术栈，包括beats、Logstash、kibana、elasticsearch

什么是Lucene?

- 是Apache的开源搜索引擎类库，提供了搜索引擎的核心API

1.2.倒排索引

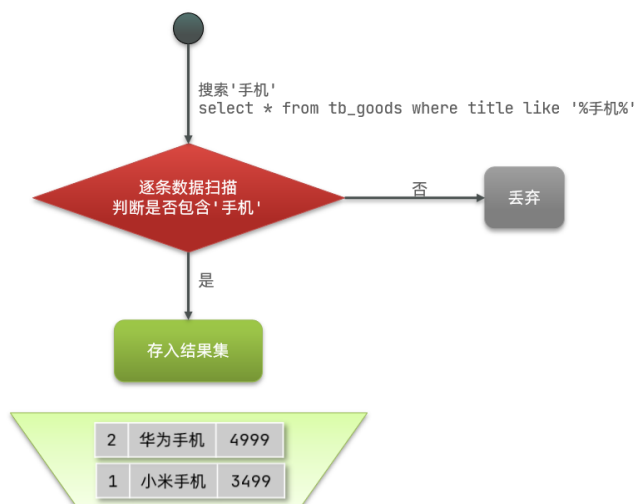
倒排索引的概念是基于MySQL这样的正向索引而言的。

1.2.1.正向索引

那么什么是正向索引呢？例如给下表（tb_goods）中的id创建索引：

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
...

正向索引



如果是根据id查询，那么直接走索引，查询速度非常快。

但如果是基于title做模糊查询，只能是逐行扫描数据，流程如下：

- 1) 用户搜索数据，条件是title符合 "%手机%"
- 2) 逐行获取数据，比如id为1的数据
- 3) 判断数据中的title是否符合用户搜索条件
- 4) 如果符合则放入结果集，不符合则丢弃。回到步骤1

逐行扫描，也就是全表扫描，随着数据量增加，其查询效率也会越来越低。当数据量达到数百万时，就是一场灾难。

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，流程如下：

- 将每一个文档的数据利用算法分词，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，可以给词条创建索引，例如hash表结构索引

如图：



倒排索引的**搜索流程**如下（以搜索"华为手机"为例）：

- 1) 用户输入条件 "华为手机" 进行搜索。
- 2) 对用户输入内容**分词**，得到词条：华为、手机。
- 3) 拿着词条在倒排索引中查找，可以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。

如图：



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

1.3.es的一些概念

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：

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	299



```
{
  "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
}
```

而json文档中往往包含很多的**字段 (Field)**，类似于数据库中的列。

1.3.2.索引和映射

索引 (Index)，就是相同类型的文档的集合。

例如：

- 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；

商品索引

```
{
  "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

```
{
  "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

```
{
  "id": 10,
  "userId": 101,
  "goodsId": 1,
  "totalFee": 294
}
{
  "id": 11,
  "userId": 102,
  "goodsId": 2,
  "totalFee": 328
}
```

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)
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

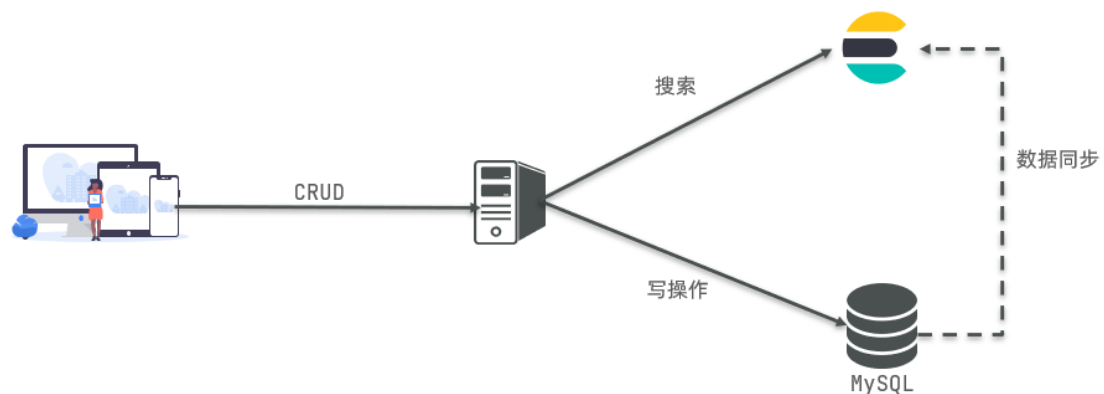
是不是说, 我们学习了elasticsearch就不再需要mysql了呢?

并不是如此, 两者各自有自己的擅长支出:

- 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es、kibana

1.4.1.安装

参考课前资料:



安装elasticsearch.md

1.4.2.分词器

参考课前资料：



安装elasticsearch.md

1.4.3.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

索引库就类似数据库表，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“库”和“表”。

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：

```
{
  "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）：

- age：类型为integer；参与搜索，因此需要index为true；无需分词器
- weight：类型为float；参与搜索，因此需要index为true；无需分词器
- isMarried：类型为boolean；参与搜索，因此需要index为true；无需分词器
- info：类型为字符串，需要分词，因此是text；参与搜索，因此需要index为true；分词器可以用ik_smart
- email：类型为字符串，但是不需要分词，因此是keyword；不参与搜索，因此需要index为false；无需分词器
- score：虽然是数组，但是我们只看元素的类型，类型为float；参与搜索，因此需要index为true；无需分词器

- name: 类型为object, 需要定义多个子属性
 - name.firstName; 类型为字符串, 但是不需要分词, 因此是keyword; 参与搜索, 因此需要index为true; 无需分词器
 - name.lastName; 类型为字符串, 但是不需要分词, 因此是keyword; 参与搜索, 因此需要index为true; 无需分词器

2.2.索引库的CRUD

这里我们统一使用Kibana编写DSL的方式来演示。

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:

```
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：

```
PUT /heima
{
  "mappings": {
    "properties": {
      "info":{
        "type": "text",
        "analyzer": "ik_smart"
      },
      "email":{
        "type": "keyword",
        "index": "false"
      },
      "name":{
        "properties": {
          "firstName": {
            "type": "keyword"
          }
        }
      },
      // ... 略
    }
  }
}
```

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：

```
GET /索引库名
```

示例：



2.2.3.修改索引库

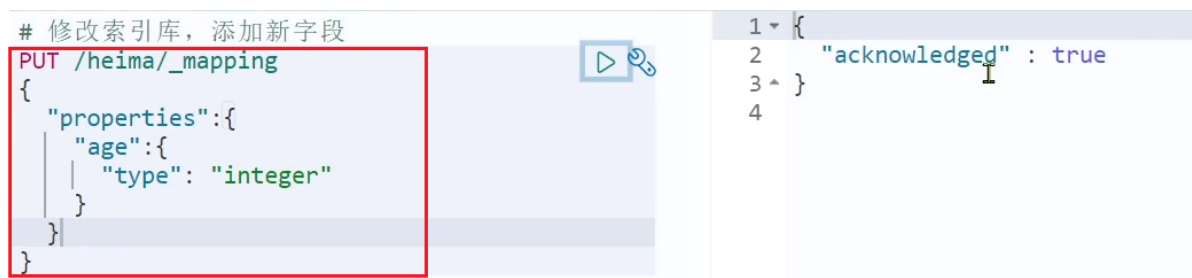
倒排索引结构虽然不复杂，但是一旦数据结构改变（比如改变了分词器），就需要重新创建倒排索引，这简直是灾难。因此索引库一旦创建，无法修改mapping。

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。

语法说明：

```
PUT /索引库名/_mapping
{
  "properties": {
    "新字段名": {
      "type": "integer"
    }
  }
}
```

示例：



2.2.4.删除索引库

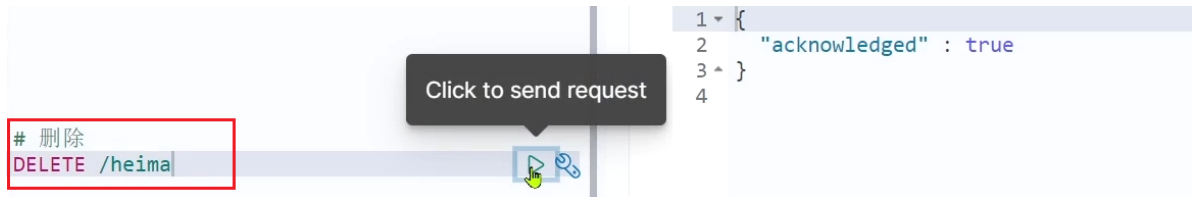
语法:

- 请求方式: DELETE
- 请求路径: /索引库名
- 请求参数: 无

格式:

```
DELETE /索引库名
```

在kibana中测试:



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
- 添加字段: PUT /索引库名/_mapping

3.文档操作

3.1.新增文档

语法:

```
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:


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

响应:

```
# 插入文档
POST /heima/_doc/1
{
  "info": "黑马程序员Java讲师",
  "email": "zy@itcast.cn",
  "name": {
    "firstName": "云",
    "lastName": "赵"
  }
}
```

```
1 {
2   "_index" : "heima",
3   "_type" : "_doc",
4   "_id" : "1",
5   "_version" : 1,
6   "result" : "created",
7   "_shards" : {
8     "total" : 2,
9     "successful" : 1,
10    "failed" : 0
11  },
12   "_seq_no" : 0,
13   "_primary_term" : 1
14 }
15
```

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:

```
GET /{索引库名称}/_doc/{id}
```

通过kibana查看数据:

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

查看结果:

```
# 查询文档
GET /heima/_doc/1
```

```
1 {
2   "_index" : "heima",
3   "_type" : "_doc",
4   "_id" : "1",
5   "_version" : 1,
6   "_seq_no" : 0,
7   "_primary_term" : 1,
8   "found" : true,
9   "source" : {
10    "info" : "黑马程序员Java讲师",
11    "email" : "zy@itcast.cn",
12    "name" : {
13      "firstName" : "云",
14      "lastName" : "赵"
15    }
16  }
17 }
18
```

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：

```
DELETE /{索引库名}/_doc/id值
```

示例：

```
# 根据id删除数据
DELETE /heima/_doc/1
```

结果：

```
# 删除文档
DELETE /heima/_doc/1
```

```
1 {
2   "_index" : "heima",
3   "_type" : "_doc",
4   "_id" : "1",
5   "_version" : 2,
6   "result" : "deleted",
7   "_shards" : {
8     "total" : 2,
9     "successful" : 1,
10    "failed" : 0
11  },
12   "_seq_no" : 1,
13   "_primary_term" : 1
14 }
```

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
- 增量修改：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：

- 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：

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

示例:

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

3.4.2.增量修改

增量修改是只修改指定id匹配的文档中的部分字段。

语法:

```
POST /{索引库名}/_update/文档id
{
  "doc": {
    "字段名": "新的值",
  }
}
```

示例:

```
POST /heima/_update/1
{
  "doc": {
    "email": "ZhaoYun@itcast.cn"
  }
}
```

3.5.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 }
 - 增量修改: POST /{索引库名}/_update/文档id { "doc": {字段}}

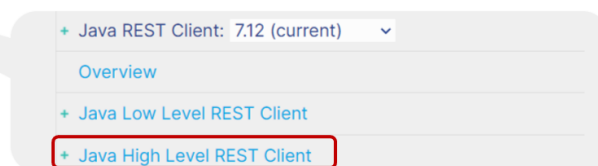
4.RestAPI

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>

其中的Java Rest Client又包括两种：

- Java Low Level Rest Client
- Java High Level Rest Client

- Java REST Client [7.12] — other versions
- JavaScript API [7.x] — other versions
- Ruby API [7.x] — other versions
- Go API [7.x] — other versions
- .NET API [7.x] — other versions
- PHP API [7.x] — other versions
- Perl API
- Python API [7.x] — other versions
- eland
- Rust API
- Java API (deprecated) [7.12] — other versions
- Community Contributed Clients



我们学习的是Java HighLevel Rest Client客户端API

4.0.导入Demo工程

4.0.1.导入数据

首先导入课前资料提供的数据库数据：



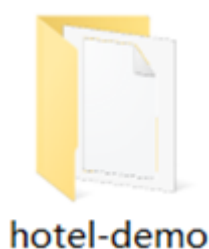
tb_hotel.sql

数据结构如下：

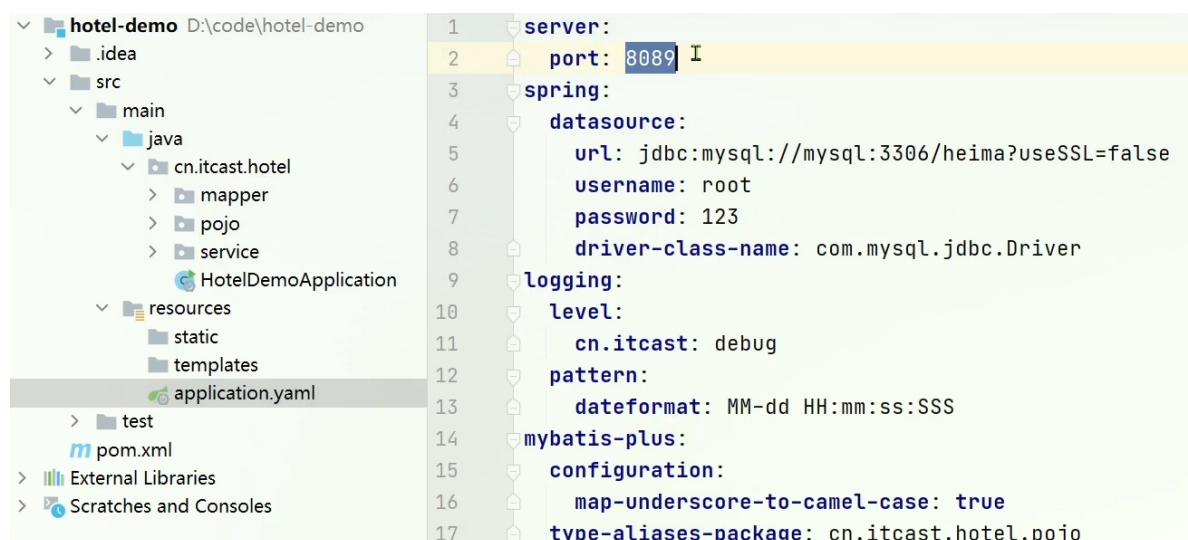
```
CREATE TABLE `tb_hotel` (  
  `id` bigint(20) NOT NULL COMMENT '酒店id',  
  `name` varchar(255) NOT NULL COMMENT '酒店名称；例：7天酒店',  
  `address` varchar(255) NOT NULL COMMENT '酒店地址；例：航头路',  
  `price` int(10) NOT NULL COMMENT '酒店价格；例：329',  
  `score` int(2) NOT NULL COMMENT '酒店评分；例：45，就是4.5分',  
  `brand` varchar(32) NOT NULL COMMENT '酒店品牌；例：如家',  
  `city` varchar(32) NOT NULL COMMENT '所在城市；例：上海',  
  `star_name` varchar(16) DEFAULT NULL COMMENT '酒店星级，从低到高分别是：1星到5星，1钻到5钻',  
  `business` varchar(255) DEFAULT NULL COMMENT '商圈；例：虹桥',  
  `latitude` varchar(32) NOT NULL COMMENT '纬度；例：31.2497',  
  `longitude` varchar(32) NOT NULL COMMENT '经度；例：120.3925',  
  `pic` varchar(255) DEFAULT NULL COMMENT '酒店图片；例：/img/1.jpg',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

4.0.2.导入项目

然后导入课前资料提供的项目：



项目结构如图：



4.0.3.mapping映射分析

创建索引库，最关键的是mapping映射，而mapping映射要考虑的信息包括：

- 字段名
- 字段数据类型
- 是否参与搜索
- 是否需要分词
- 如果分词，分词器是什么？

其中：

- 字段名、字段数据类型，可以参考数据表结构的名称和类型
- 是否参与搜索要分析业务来判断，例如图片地址，就无需参与搜索
- 是否分词呢要看内容，内容如果是一个整体就无需分词，反之则要分词
- 分词器，我们可以统一使用ik_max_word

来看下酒店数据的索引库结构：

```
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"
    },
    "score": {
      "type": "integer"
    },
    "brand": {
      "type": "keyword",
      "copy_to": "all"
    },
    "city": {
      "type": "keyword",
      "copy_to": "all"
    },
    "starName": {
      "type": "keyword"
    },
    "business": {
      "type": "keyword"
    },
    "location": {
      "type": "geo_point"
    },
    "pic": {
      "type": "keyword",
      "index": false
    },
    "all": {
      "type": "text",
      "analyzer": "ik_max_word"
    }
  }
}

```

几个特殊字段说明：

- location：地理坐标，里面包含精度、纬度
- all：一个组合字段，其目的是将多字段的值 利用copy_to合并，提供给用户搜索

地理坐标说明：

小提示

ES中支持两种地理坐标数据类型：

- geo_point: 由纬度 (latitude) 和经度 (longitude) 确定的一个点。例如: "32.8752345, 120.2981576"
- geo_shape: 有多个geo_point组成的复杂几何图形。例如一条直线, "LINESTRING (-77.03653 38.897676, -77.009051 38.889939)"

copy_to说明:

小提示

字段拷贝可以使用copy_to属性将当前字段拷贝到指定字段。示例:

```
"all": {
  "type": "text",
  "analyzer": "ik_max_word"
},
"brand": {
  "type": "keyword",
  "copy_to": "all"
}
```

4.0.4.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) 引入es的RestHighLevelClient依赖:

```
<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7.6.2, 所以我们需要覆盖默认的ES版本:

```
<properties>
  <java.version>1.8</java.version>
  <elasticsearch.version>7.12.1</elasticsearch.version>
</properties>
```

3) 初始化RestHighLevelClient:

初始化的代码如下:

```
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HotelIndexTest，然后将初始化的代码编写在@BeforeEach方法中：

```
package cn.itcast.hotel;

import org.apache.http.HttpHost;
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 HotelIndexTest {
    private RestHighLevelClient client;

    @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();
    }
}
```

4.1.创建索引库

4.1.1.代码解读

创建索引库的API如下：

创建索引库代码如下：

```
@Test
void testCreateHotelIndex() throws IOException {
    // 1. 创建Request对象
    CreateIndexRequest request = new CreateIndexRequest("hotel");
    // 2. 请求参数，MAPPING_TEMPLATE是静态常量字符串，内容是创建索引库的DSL语句
    request.source(MAPPING_TEMPLATE, XContentType.JSON);
    // 3. 发起请求
    client.indices().create(request, RequestOptions.DEFAULT);
}
```

返回的对象中包含
索引库操作的所有方法

请求路径，索引库名称

```
PUT /hotel
{
  "mappings": {
    "properties": {
      "id": {
        "type": "keyword"
      },
      "name": {
        "type": "text",
        "analyzer": "ik_max_word"
      },
      "address": {},
      "price": {},
      "score": {},
      "brand": {},
      "city": {},
      "starName": {},
      "business": {},
      "location": {
        "type": "geo_point"
      },
      "pic": {
        "type": "keyword",
        "index": false
      }
    }
  }
}
```

DSL

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DSL的JSON参数部分。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。

4.1.2.完整示例

在hotel-demo的cn.itcast.hotel.constants包下，创建一个类，定义mapping映射的JSON字符串常量：

```
package cn.itcast.hotel.constants;

public class HotelConstants {
    public 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" +
        "        \"copy_to\": \"all\"\n" +
        "      },\n" +
        "      \"address\": {\n" +
        "        \"type\": \"keyword\",\n" +
        "        \"index\": false\n" +
        "      },\n" +
        "      \"price\": {\n" +
        "        \"type\": \"integer\"\n" +
        "      },\n" +
        "      \"score\": {\n" +
        "        \"type\": \"integer\"\n" +
        "      },\n" +
        "      \"brand\": {\n" +
        "        \"type\": \"keyword\",\n" +
        "        \"copy_to\": \"all\"\n" +
        "      },\n" +
        "      \"city\": {\n" +
        "        \"type\": \"keyword\",\n" +
        "        \"copy_to\": \"all\"\n" +
        "      },\n" +
        "      \"starName\": {\n" +
        "        \"type\": \"keyword\"\n" +
        "      },\n" +
        "      \"business\": {\n" +
        "        \"type\": \"keyword\"\n" +
        "      },\n" +
        "      \"location\": {\n" +
        "        \"type\": \"geo_point\"\n" +
        "      },\n" +
        "      \"pic\": {\n" +
        "        \"type\": \"keyword\",
```

```

        "\"index\": false\n" +
        },\n" +
        "\"all\":{\n" +
        "\"type\": \"text\", \n" +
        "\"analyzer\": \"ik_max_word\"\n" +
        }\n" +
        }\n" +
        }\n" +
        "};
    }
}

```

在hotel-demo中的HotelIndexTest测试类中，编写单元测试，实现创建索引：

```

@Test
void createHotelIndex() throws IOException {
    // 1.创建Request对象
    CreateIndexRequest request = new CreateIndexRequest("hotel");
    // 2.准备请求的参数：DSL语句
    request.source(MAPPING_TEMPLATE, XContentType.JSON);
    // 3.发送请求
    client.indices().create(request, RequestOptions.DEFAULT);
}

```

4.2.删除索引库

删除索引库的DSL语句非常简单：

```
DELETE /hotel
```

与创建索引库相比：

- 请求方式从PUT变为DELETE
- 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。依然是三步走：

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

在hotel-demo中的HotelIndexTest测试类中，编写单元测试，实现删除索引：

```

@Test
void testDeleteHotelIndex() throws IOException {
    // 1.创建Request对象
    DeleteIndexRequest request = new DeleteIndexRequest("hotel");
    // 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DSL是：

```
GET /hotel
```

因此与删除的Java代码流程是类似的。依然是三步走：

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

```
@Test
void testExistsHotelIndex() throws IOException {
    // 1.创建Request对象
    GetIndexRequest request = new GetIndexRequest("hotel");
    // 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DSL（Create时需要，其它是无参）
- 发送请求。调用RestHighLevelClient#indices().xxx()方法，xxx是create、exists、delete

5.RestClient操作文档

为了与索引库操作分离，我们再次参加一个测试类，做两件事情：

- 初始化RestHighLevelClient
- 我们的酒店数据在数据库，需要利用IHotelService去查询，所以注入这个接口

```
package cn.itcast.hotel;

import cn.itcast.hotel.pojo.Hotel;
import cn.itcast.hotel.service.IHotelService;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.io.IOException;
import java.util.List;

@SpringBootTest
public class HotelDocumentTest {
    @Autowired
    private IHotelService hotelService;

    private RestHighLevelClient client;

    @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

我们要将数据库的酒店数据查询出来，写入elasticsearch中。

5.1.1.索引库实体类

数据库查询后的结果是一个Hotel类型的对象。结构如下：

```

@Data
@TableName("tb_hotel")
public class Hotel {
    @TableId(type = IdType.INPUT)
    private Long id;
    private String name;
    private String address;
    private Integer price;
    private Integer score;
    private String brand;
    private String city;
    private String starName;
    private String business;
    private String longitude;
    private String latitude;
    private String pic;
}

```

与我们的索引库结构存在差异：

- longitude和latitude需要合并为location

因此，我们需要定义一个新的类型，与索引库结构吻合：

```
package cn.itcast.hotel.pojo;

import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
public class HotelDoc {
    private Long id;
    private String name;
    private String address;
    private Integer price;
    private Integer score;
    private String brand;
    private String city;
    private String starName;
    private String business;
    private String location;
    private String pic;

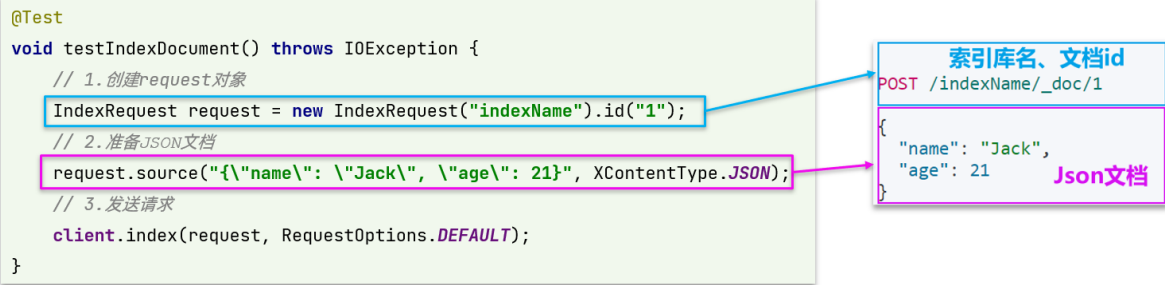
    public HotelDoc(Hotel hotel) {
        this.id = hotel.getId();
        this.name = hotel.getName();
        this.address = hotel.getAddress();
        this.price = hotel.getPrice();
        this.score = hotel.getScore();
        this.brand = hotel.getBrand();
        this.city = hotel.getCity();
        this.starName = hotel.getStarName();
        this.business = hotel.getBusiness();
        this.location = hotel.getLatitude() + ", " + hotel.getLongitude();
        this.pic = hotel.getPic();
    }
}
```

5.1.2.语法说明

新增文档的DSL语句如下：

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

对应的java代码如图：



可以看到与创建索引库类似，同样是三步走：

- 1) 创建Request对象
- 2) 准备请求参数，也就是DSL中的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

我们导入酒店数据，基本流程一致，但是需要考虑几点变化：

- 酒店数据来自于数据库，我们需要先查询出来，得到hotel对象
- hotel对象需要转为HotelDoc对象
- HotelDoc需要序列化为json格式

因此，代码整体步骤如下：

- 1) 根据id查询酒店数据Hotel
- 2) 将Hotel封装为HotelDoc
- 3) 将HotelDoc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hotel-demo的HotelDocumentTest测试类中，编写单元测试：

```
@Test
void testAddDocument() throws IOException {
    // 1. 根据id查询酒店数据
    Hotel hotel = hotelService.getById(61083L);
    // 2. 转换为文档类型
    HotelDoc hotelDoc = new HotelDoc(hotel);
    // 3. 将HotelDoc转json
    String json = JSON.toJSONString(hotelDoc);

    // 1. 准备Request对象
    IndexRequest request = new
IndexRequest("hotel").id(hotelDoc.getId().toString());
    // 2. 准备Json文档
    request.source(json, XContentType.JSON);
    // 3. 发送请求
    client.index(request, RequestOptions.DEFAULT);
}
```

5.2.查询文档

5.2.1.语法说明

查询的DSL语句如下：

```
GET /hotel/_doc/{id}
```

非常简单，因此代码大概分两步：

- 准备Request对象
- 发送请求

不过查询的目的是得到结果，解析为HotelDoc，因此难点是结果的解析。完整代码如下：

```
@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，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。

与之前类似，也是三步走：

- 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hotel-demo的HotelDocumentTest测试类中，编写单元测试：

```

@Test
void testGetDocumentById() throws IOException {
    // 1.准备Request
    GetRequest request = new GetRequest("hotel", "61082");
    // 2.发送请求，得到响应
    GetResponse response = client.get(request, RequestOptions.DEFAULT);
    // 3.解析响应结果
    String json = response.getSourceAsString();

    HotelDoc hotelDoc = JSON.parseObject(json, HotelDoc.class);
    System.out.println(hotelDoc);
}

```

5.3.删除文档

删除的DSL为是这样的：

```
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代码应该依然是三步走：

- 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
- 3) 发送请求。因为是删除，所以是client.delete()方法

在hotel-demo的HotelDocumentTest测试类中，编写单元测试：

```

@Test
void testDeleteDocument() throws IOException {
    // 1.准备Request
    DeleteRequest request = new DeleteRequest("hotel", "61083");
    // 2.发送请求
    client.delete(request, RequestOptions.DEFAULT);
}

```

5.4.修改文档

5.4.1.语法说明

修改我们讲过两种方式：

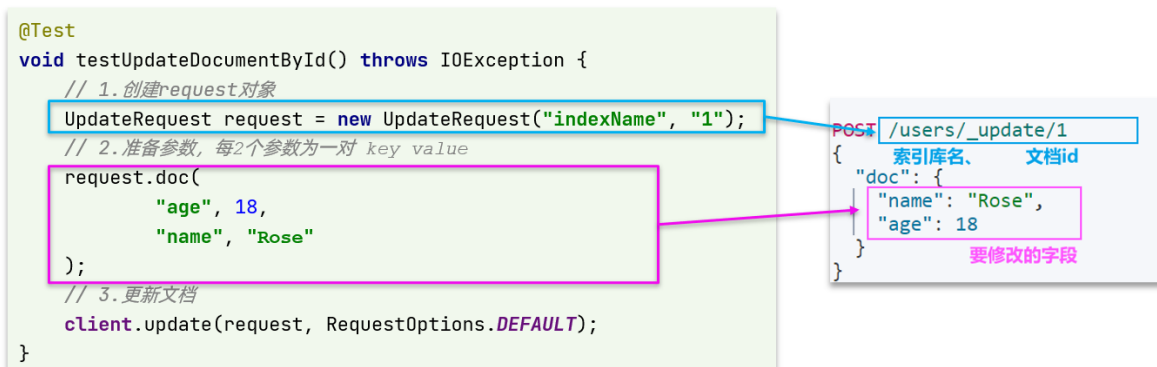
- 全量修改：本质是先根据id删除，再新增
- 增量修改：修改文档中的指定字段值

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，我们主要关注增量修改。

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hotel-demo的HotelDocumentTest测试类中，编写单元测试：

```
@Test
void testUpdateDocument() throws IOException {
    // 1. 准备Request
    UpdateRequest request = new UpdateRequest("hotel", "61083");
    // 2. 准备请求参数
    request.doc(
        "price", "952",
        "starName", "四钻"
    );
    // 3. 发送请求
    client.update(request, RequestOptions.DEFAULT);
}
```

5.5.批量导入文档

案例需求：利用BulkRequest批量将数据库数据导入到索引库中。

步骤如下：

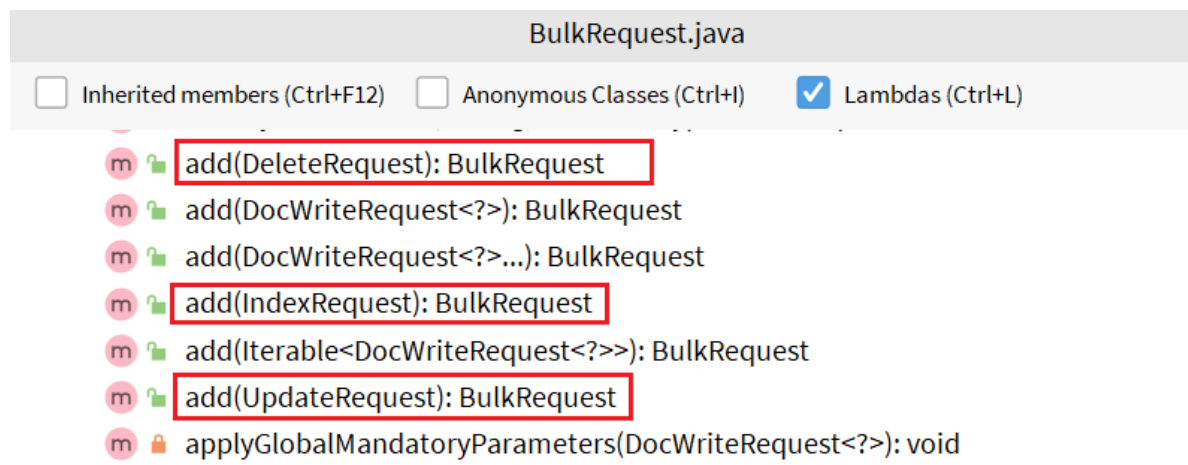
- 利用mybatis-plus查询酒店数据
- 将查询到的酒店数据（Hotel）转换为文档类型数据（HotelDoc）

- 利用javaRestClient中的BulkRequest批处理，实现批量新增文档

5.5.1.语法说明

批量处理BulkRequest，其本质就是将多个普通的CRUD请求组合在一起发送。

其中提供了一个add方法，用来添加其他请求：



可以看到，能添加的请求包括：

- 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：

```
@Test
void testBulk() throws IOException {
    // 1. 创建Bulk请求
    BulkRequest request = new BulkRequest();
    // 2. 添加要批量提交的请求：这里添加了两个新增文档的请求
    request.add(new IndexRequest("hotel")
        .id("101").source("json source", XContentType.JSON));
    request.add(new IndexRequest("hotel")
        .id("102").source("json source2", XContentType.JSON));
    // 3. 发起bulk请求
    client.bulk(request, RequestOptions.DEFAULT);
}
```

其实还是三步走：

- 1) 创建Request对象。这里是BulkRequest
- 2) 准备参数。批处理的参数，就是其它Request对象，这里就是多个IndexRequest
- 3) 发起请求。这里是批处理，调用的方法为client.bulk()方法

我们在导入酒店数据时，将上述代码改造成for循环处理即可。

5.5.2.完整代码

在hotel-demo的HotelDocumentTest测试类中，编写单元测试：

```
@Test
void testBulkRequest() throws IOException {
    // 批量查询酒店数据
    List<Hotel> hotels = hotelService.list();

    // 1.创建Request
    BulkRequest request = new BulkRequest();
    // 2.准备参数，添加多个新增的Request
    for (Hotel hotel : hotels) {
        // 2.1.转换为文档类型HotelDoc
        HotelDoc hotelDoc = new HotelDoc(hotel);
        // 2.2.创建新增文档的Request对象
        request.add(new IndexRequest("hotel")
            .id(hotelDoc.getId().toString())
            .source(JSON.toJSONString(hotelDoc), XContentType.JSON));
    }
    // 3.发送请求
    client.bulk(request, RequestOptions.DEFAULT);
}
```

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）