**【Elasticsearch】**

**【主要内容】**

1. Elasticsearch简介
2. 倒排索引
3. Elasticsearch的安装
4. 可视化插件head的安装
5. Ik分词的安装
6. Elasticsearch基本使用
7. Elasticsearch整合Springboot
8. Elasticsearch复杂操作

**【学习目标】**

|  |  |
| --- | --- |
| 知识点 | 要求 |
| Elasticsearch简介 | 了解 |
| 倒排索引 | 掌握 |
| Elasticsearch的安装 | 掌握 |
| 可视化插件head的安装 | 掌握 |
| Ik分词的安装 | 了解 |
| Elasticsearch基本使用 | 掌握 |
| Elasticsearch整合Springboot | 掌握 |
| Elasticsearch复杂操作 | 掌握 |

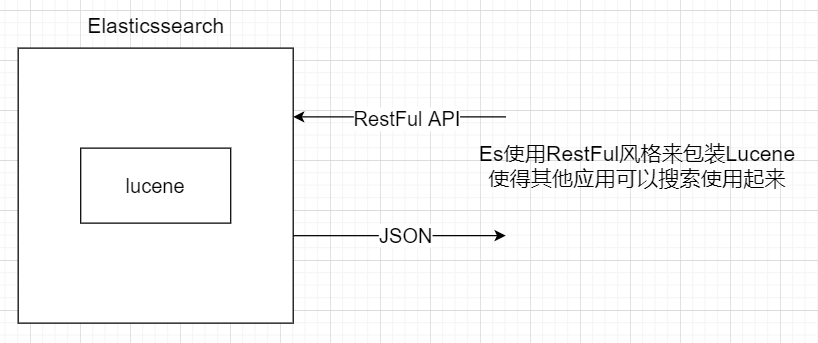
# Elasticsearch简介

**Elasticsearch是一个基于Lucene的一个开源的分布式、RESTful 风格的搜索和数据分析引擎。**Elasticsearch是用Java语言开发的，并作为Apache许可条款下的开放源码发布，是一种流行的企业级搜索引擎。Elasticsearch用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。官方客户端在Java、.NET（C#）、PHP、Python、Apache Groovy、Ruby和许多其他语言中都是可用的。根据DB-Engines的排名显示，Elasticsearch是最受欢迎的企业搜索引擎，其次是**Apache Solr**，也是基于Lucene。

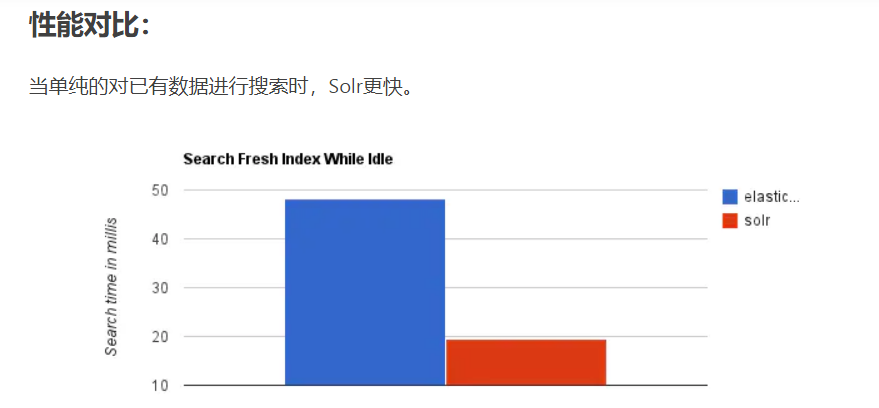
# Lucene

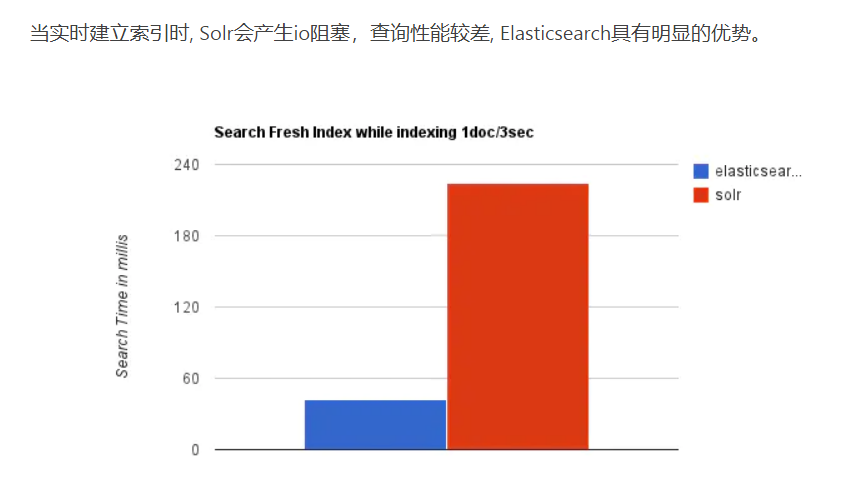
Lucene 可以说是当下最先进、高性能、全功能的搜索引擎库——无论是开源还是私有，但它也仅仅只是一个库。为了充分发挥其功能，你需要使用 Java 并将 Lucene 直接集成到应用程序中。 更糟糕的是，您可能需要获得信息检索学位才能了解其工作原理，因为Lucene 非常复杂。

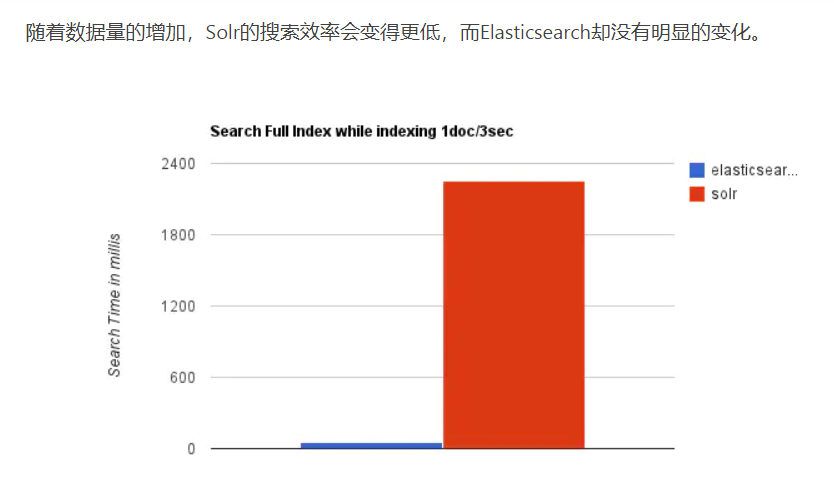
为了解决Lucene使用时的繁复性，于是Elasticsearch便应运而生。它使用 Java 编写，内部采用 Lucene 做索引与搜索，但是它的目标是使全文检索变得更简单，简单来说，就是对Lucene 做了一层封装，它提供了一套简单一致的 RESTful API 来帮助我们实现存储和检索。



# 和solr对比









**ElasticSearch 对比 Solr 总结**

1. es基本是开箱即用，非常简单。Solr安装略微复杂一丢丢
2. Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能。
3. Solr 支持更多格式的数据，比如JSON、XML、CSV，而 Elasticsearch 仅支持json文件格式。
4. Solr 官方提供的功能更多，而 Elasticsearch 本身更注重于核心功能，高级功能多有第三方插件提供，例如图形化界面需要kibana，head等友好支撑
5. Solr 查询快，但更新索引时慢（即插入删除慢），用于电商等查询多的应用；

ES建立索引快（即查询慢），即实时性查询快，用于facebook新浪等搜索。

Solr 是传统搜索应用的有力解决方案，但 Elasticsearch 更适用于新兴的实时搜索应用。

1. Solr比较成熟，有一个更大，更成熟的用户、开发和贡献者社区，而 Elasticsearch相对开发维护者较少，更新太快，学习使用成本较高。

# 倒排索引

## 正排索引



## 倒排索引



一个倒排索引由文档中所有不重复词的列表构成，对于其中每个词，有一个包含它的 Term 列表。

# 分词

就是按照一定的规则，将一句话分成组合的单词

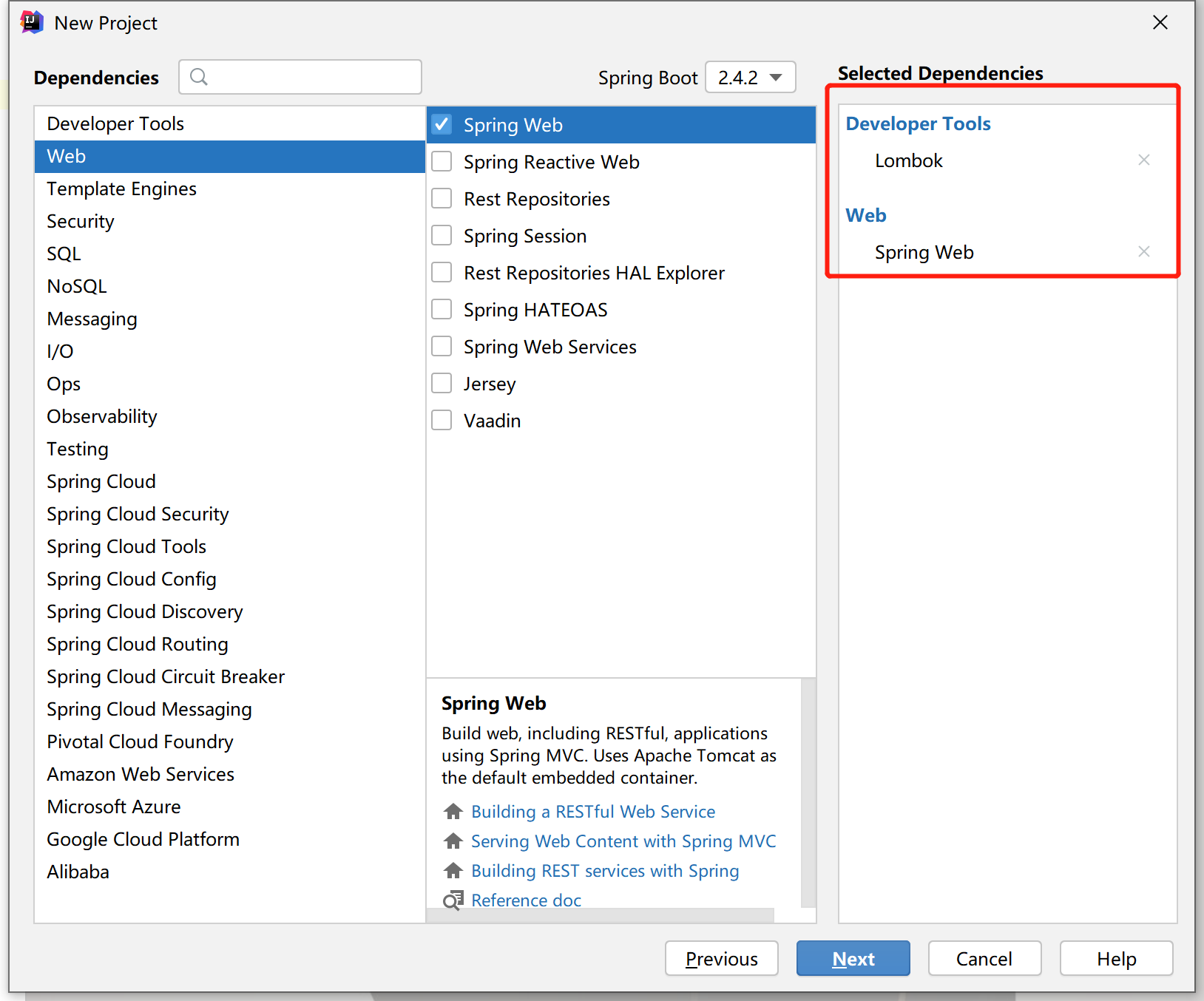
海上生明月 - 如何分成  ----->海上 | 生 | 明月

# 模拟一个倒排索引

原理步骤：

1. 将数据存入mysql之前，对其进行分词
2. 讲分词和存入后得到的id，存放在数据结构中**Map<String,List<Integer>> index**
3. 查询时先分词，然后从index中拿到List<Integer> ids
4. 再根据ids 查询mysql，从而得到结果，这样借助了mysql的B+tree索引，提高性能

## 创建boot项目选择依赖



## 引入分词的依赖

|  |
| --- |
| <dependency>  <groupId>com.huaban</groupId>  <artifactId>jieba-analysis</artifactId>  <version>1.0.2</version> </dependency> |

## 修改启动类，注入结巴分词器

|  |
| --- |
| */\*\*  \* 往IOC容器中入住结巴分词组件  \*  \* @return  \*/* @Bean public JiebaSegmenter jiebaSegmenter() {  return new JiebaSegmenter(); } |

## 测试分词

|  |
| --- |
| @Autowired public JiebaSegmenter jiebaSegmenter;  @Test void testJieBa() {  String words = "华为 HUAWEI P40 Pro 麒麟990 5G SoC芯片 5000万超感知徕卡四摄 50倍数字变焦 8GB+256GB零度白全网通5G手机";  *// 使用结巴分词，对字符串进行分词，分词类型为搜索类型* List<SegToken> tokens = jiebaSegmenter.process(words, JiebaSegmenter.SegMode.*INDEX*);  *// 遍历，拿到SegToken对象中的word属性，打印结果* tokens.stream()  .map(token -> token.word)  .collect(Collectors.*toList*())  .forEach(System.*out*::println); } |

## 使用商品搜索案例来展示倒排索引结构

### 新建Goods类

|  |
| --- |
| @Data @AllArgsConstructor @NoArgsConstructor public class Goods {  */\*\*  \* 商品的id  \*/* private Integer goodsId;   */\*\*  \* 商品的名称（主要分词和检索字段）  \*/* private String goodsName;   */\*\*  \* 商品的价格  \*/* private Double goodsPrice;   } |

### 模拟数据库，新建DBUtil类

|  |
| --- |
| public class DBUtil {  */\*\*  \* 模拟数据库，key=id，value=商品对象  \* 这里也可以使用List来模拟  \*/* public static Map<Integer, Goods> *db* = new HashMap<>();   */\*\*  \* 插入数据库  \*  \* @param goods  \*/* public static void insert(Goods goods) {  *db*.put(goods.getGoodsId(), goods);  }    */\*\*  \* 根据id得到商品  \*  \* @param id  \* @return  \*/* public static Goods getGoodsById(Integer id) {  return *db*.get(id);  }   */\*\*  \* 根据ids查询商品集合  \*  \* @param ids  \* @return  \*/* public static List<Goods> getGoodsByIds(Set<Integer> ids) {  if (CollectionUtils.*isEmpty*(ids)) {  return Collections.*emptyList*();  }  List<Goods> goods = new ArrayList<>(ids.size() \* 2);  *// 循环ids* ids.forEach(id -> {  *// 从数据库拿到数据* Goods g = *db*.get(id);  if (!ObjectUtils.*isEmpty*(g)) {  goods.add(g);  }  });  return goods;  } } |

### 创建倒排索引的数据结构

|  |
| --- |
| public class InvertedIndex {  */\*\*  \* 倒排索引 key = 分词，value= ids  \*/* public static Map<String, Set<Integer>> *index* = new HashMap<>();  } |

### 创建GoodsService接口

|  |
| --- |
| public interface GoodsService {   */\*\*  \* 添加商品的方法  \*  \* @param goods  \*/* void addGoods(Goods goods);   */\*\*  \* 根据商品名称查询  \*  \* @param name  \* @return  \*/* List<Goods> findGoodsByName(String name);   */\*\*  \* 根据关键字查询  \*  \* @param keywords  \* @return  \*/* List<Goods> findGoodsByKeywords(String keywords);  } |

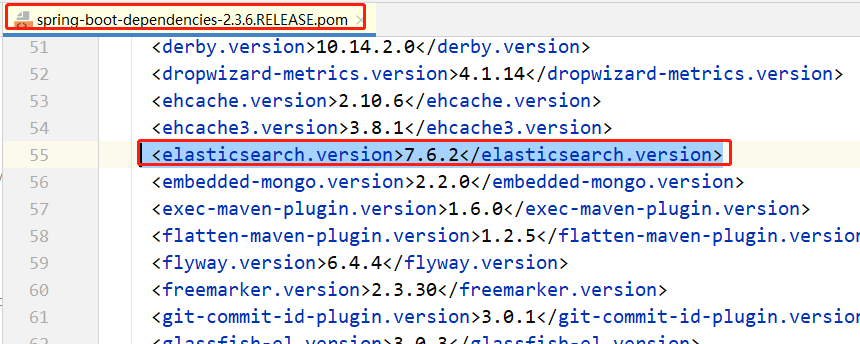
### 创建GoodsServiceImpl实现类

|  |
| --- |
| @Service public class GoodsServiceImpl implements GoodsService {   @Autowired  private JiebaSegmenter jiebaSegmenter;   */\*\*  \* 添加商品的方法  \* 1.先对商品名称进行分词，拿到了List<String> tokens  \* 2.将商品插入数据库 拿到商品id  \* 3.将tokens和id放入倒排索引中index  \*  \* @param goods  \*/* @Override  public void addGoods(Goods goods) {  *// 分词* List<String> keywords = fenci(goods.getGoodsName());  *// 插入数据库* DBUtil.*insert*(goods);  *// 保存到倒排索引中* saveToInvertedIndex(keywords, goods.getGoodsId());  }   */\*\*  \* 保存到倒排索引的方法  \*  \* @param keywords  \* @param goodsId  \*/* private void saveToInvertedIndex(List<String> keywords, Integer goodsId) {  *// 拿到索引* Map<String, Set<Integer>> index = InvertedIndex.*index*;  *// 循环分词集合* keywords.forEach(keyword -> {  Set<Integer> ids = index.get(keyword);  if (CollectionUtils.*isEmpty*(ids)) {  *// 如果之前没有这个词 就添加进去* HashSet<Integer> newIds = new HashSet<>(2);  newIds.add(goodsId);  index.put(keyword, newIds);  } else {  *// 说明之前有这个分词 我们记录id* ids.add(goodsId);  }  });  }   */\*\*  \* 分词的方法  \*  \* @param goodsName  \* @return  \*/* private List<String> fenci(String goodsName) {  List<SegToken> tokens = jiebaSegmenter.process(goodsName, JiebaSegmenter.SegMode.*SEARCH*);  return tokens.stream()  .map(token -> token.word)  .collect(Collectors.*toList*());  }   */\*\*  \* 根据商品名称查询  \*  \* @param name  \* @return  \*/* @Override  public List<Goods> findGoodsByName(String name) {  *// 查询倒排索引中 是否有这个词* Map<String, Set<Integer>> index = InvertedIndex.*index*;  Set<Integer> ids = index.get(name);  if (CollectionUtils.*isEmpty*(ids)) {  *// 查询数据库 模糊匹配去* } else {  *// 说明分词有 根据ids 查询数据库* return DBUtil.*getGoodsByIds*(ids);  }  return Collections.*emptyList*();  }   */\*\*  \* 根据关键字查询  \*  \* @param keywords  \* @return  \*/* @Override  public List<Goods> findGoodsByKeywords(String keywords) {  *// 进来先把关键字分词一下* List<String> tokens = fenci(keywords);  *// 拿到倒排索引* Map<String, Set<Integer>> index = InvertedIndex.*index*;  Set<Integer> realIds = new HashSet<>();  *// 循环分词集合 查询倒排索引* tokens.forEach(token -> {  Set<Integer> ids = index.get(token);  if (!CollectionUtils.*isEmpty*(ids)) {  *// 如果局部的ids不为空，就添加到总的ids里面去* realIds.addAll(ids);  }  });  *// 查询数据库* return DBUtil.*getGoodsByIds*(realIds);  } } |

### 编写测试类

|  |
| --- |
| @Autowired public GoodsService goodsService;  */\*\*  \* 测试我们自己写的倒排索引  \*  \* @throws Exception  \*/* @Test public void testMyIndex() throws Exception {  *// 造数据* Goods goods = new Goods(1, "苹果手机", 10.00);  Goods goods1 = new Goods(2, "华为手机", 11.00);  Goods goods2 = new Goods(3, "红米手机", 5.00);  Goods goods3 = new Goods(4, "联想手机", 6.00);  goodsService.addGoods(goods);  goodsService.addGoods(goods1);  goodsService.addGoods(goods2);  goodsService.addGoods(goods3);  *// 查询* goodsService.findGoodsByName("苹果手机").forEach(System.*out*::println);  System.*out*.println("--------------------------------------------");  goodsService.findGoodsByKeywords("苹果手机").forEach(System.*out*::println); } |

# Elasticsearch安装



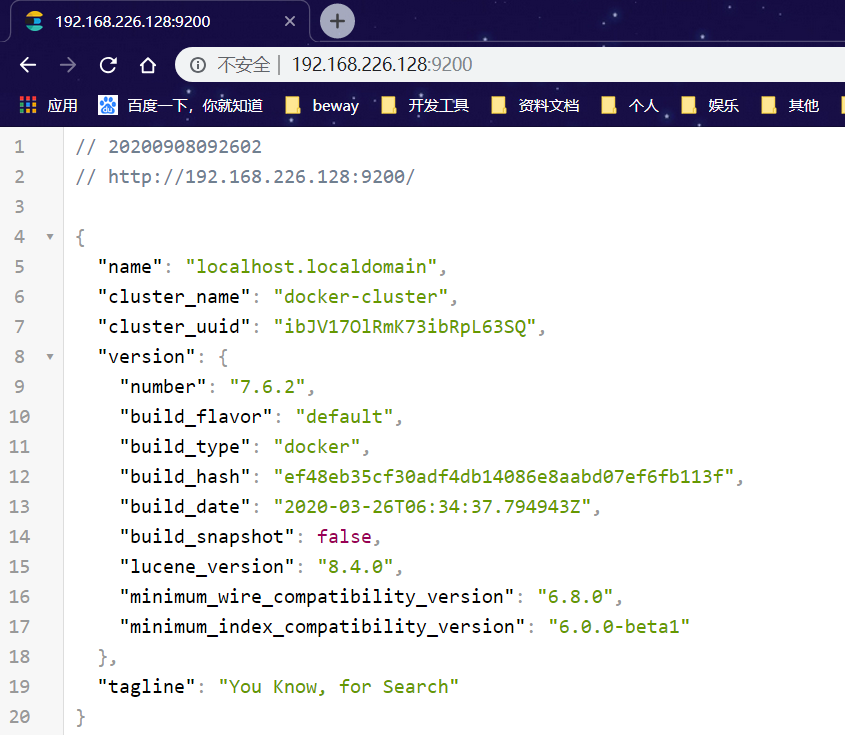
**1.Docker拉取镜像，我们选7.6.2版本，因为springboot的2.3.3发行版指定的是这个版本**

|  |
| --- |
| docker pull elasticsearch:7.6.2 |

**2.运行镜像**

|  |
| --- |
| docker run -d --name elasticsearch --net=host -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" elasticsearch:7.6.2 |

**3.测试访问 ip:9200**



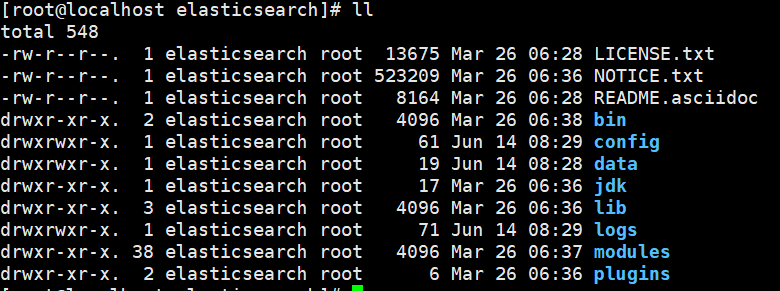
* 1. **意外**

错误：max virtual memory areas vm.max\_map\_count [65530] is too low, increase to at least [262144]，你最少需要262144的内存

运行：sysctl -w vm.max\_map\_count=262144

重启容器就ok 了

# Elasticsearch目录学习



bin：启动脚本

config:

elasticsearch.yml，ES的集群信息、对外端口、内存锁定、数据目录、跨域访问等属性的配置

jvm.options，ES使用Java写的，此文件用于设置JVM相关参数，如最大堆、最小堆

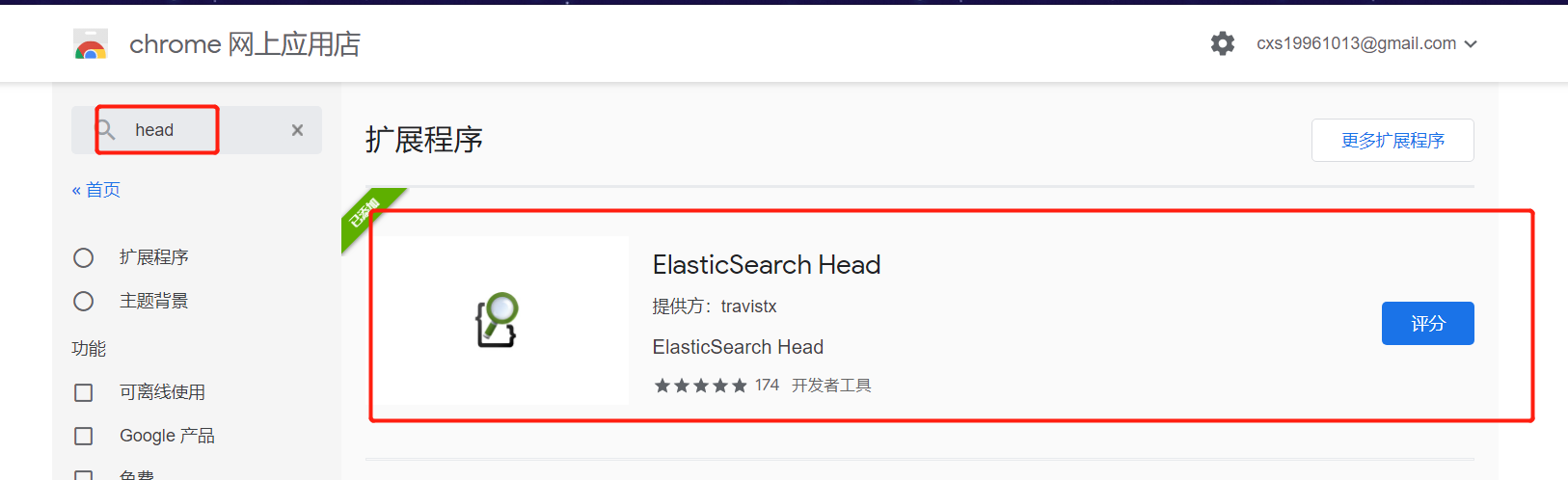
log4j2.properties，ES使用log4j作为其日志框架

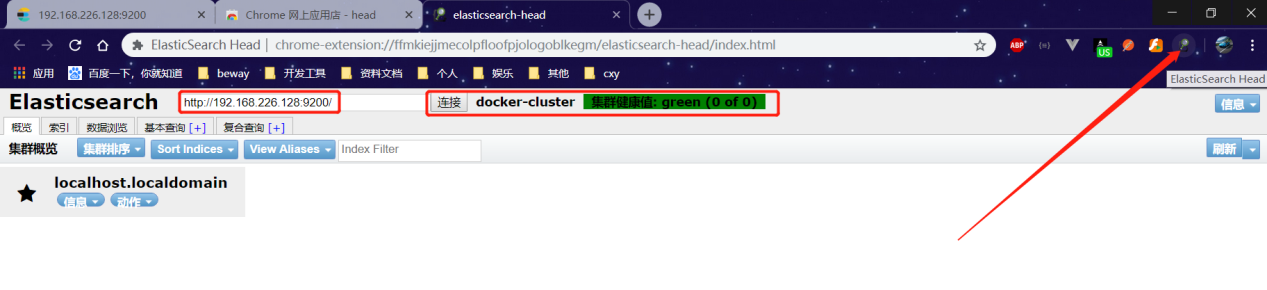
data：数据存放目录（索引数据）

plugins: ES的可扩展插件存放目录，如可以将ik中文分词插件放入此目录，ES启动时会自动加载s

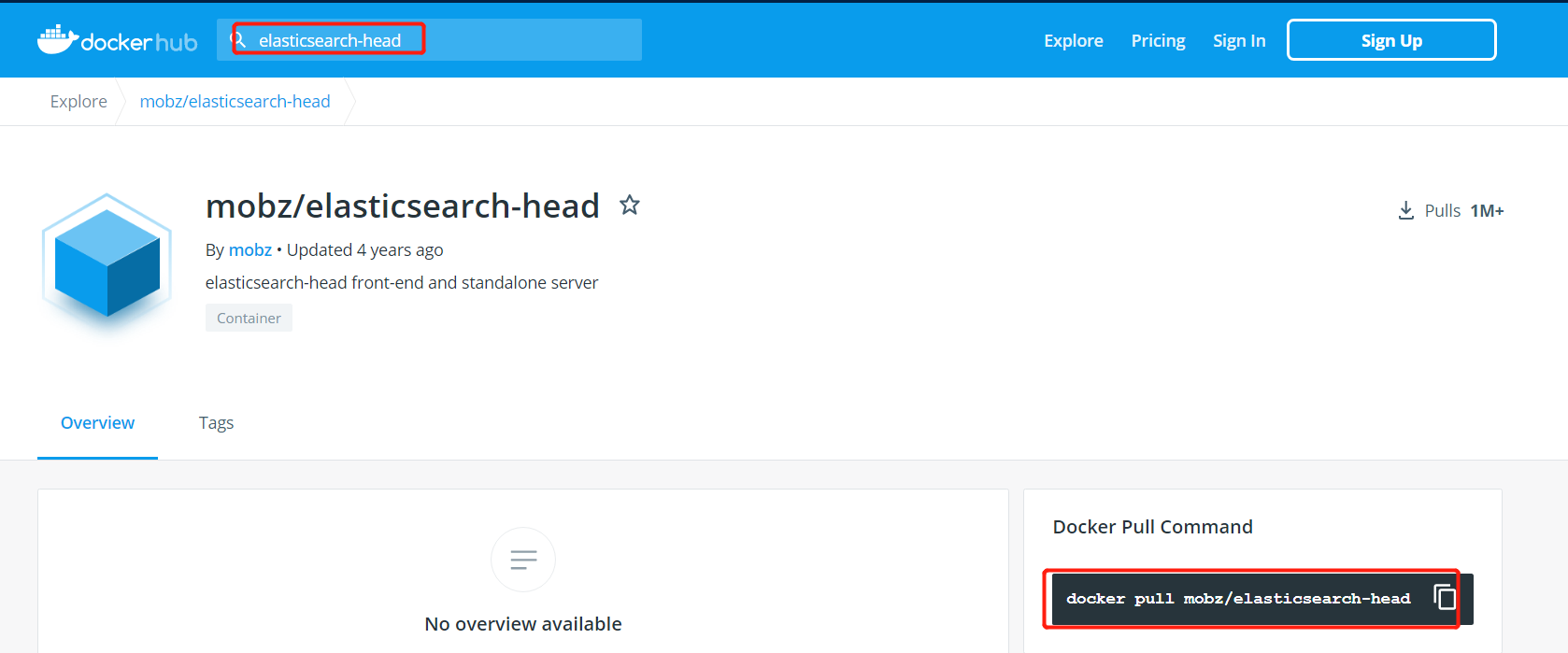
# Elasticsearch可视化插件的安装

## 谷歌插件方式



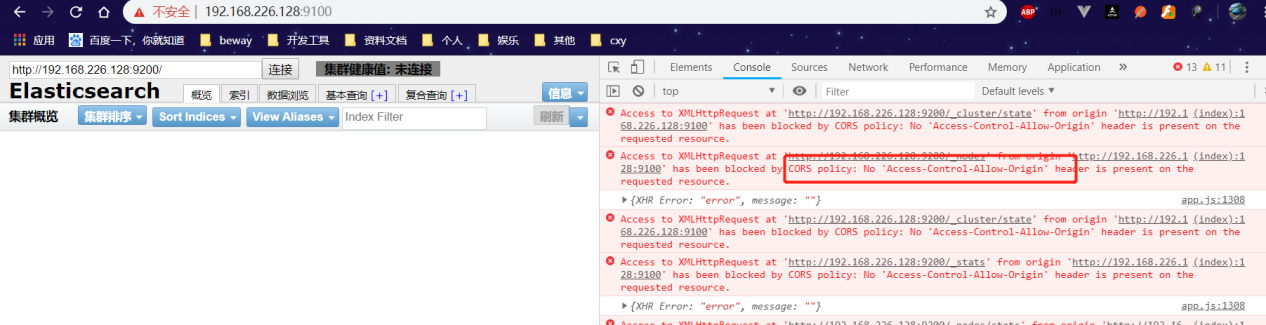


## Docker镜像方式安装



|  |
| --- |
| docker run --name eshead -p 9100:9100 -d mobz/elasticsearch-head:5 |

## 解决跨域问题



**1.进入elasticsearch容器**

|  |
| --- |
| docker exec -it elasticsearch bash |

**2.进入配置文件**

|  |
| --- |
| cd /usr/share/elasticsearch/config |

**3.修改elasticsearch.yml配置文件，结尾添加**

|  |
| --- |
| http.cors.enbaled: true  http.cors.allow-origin: "\*" |

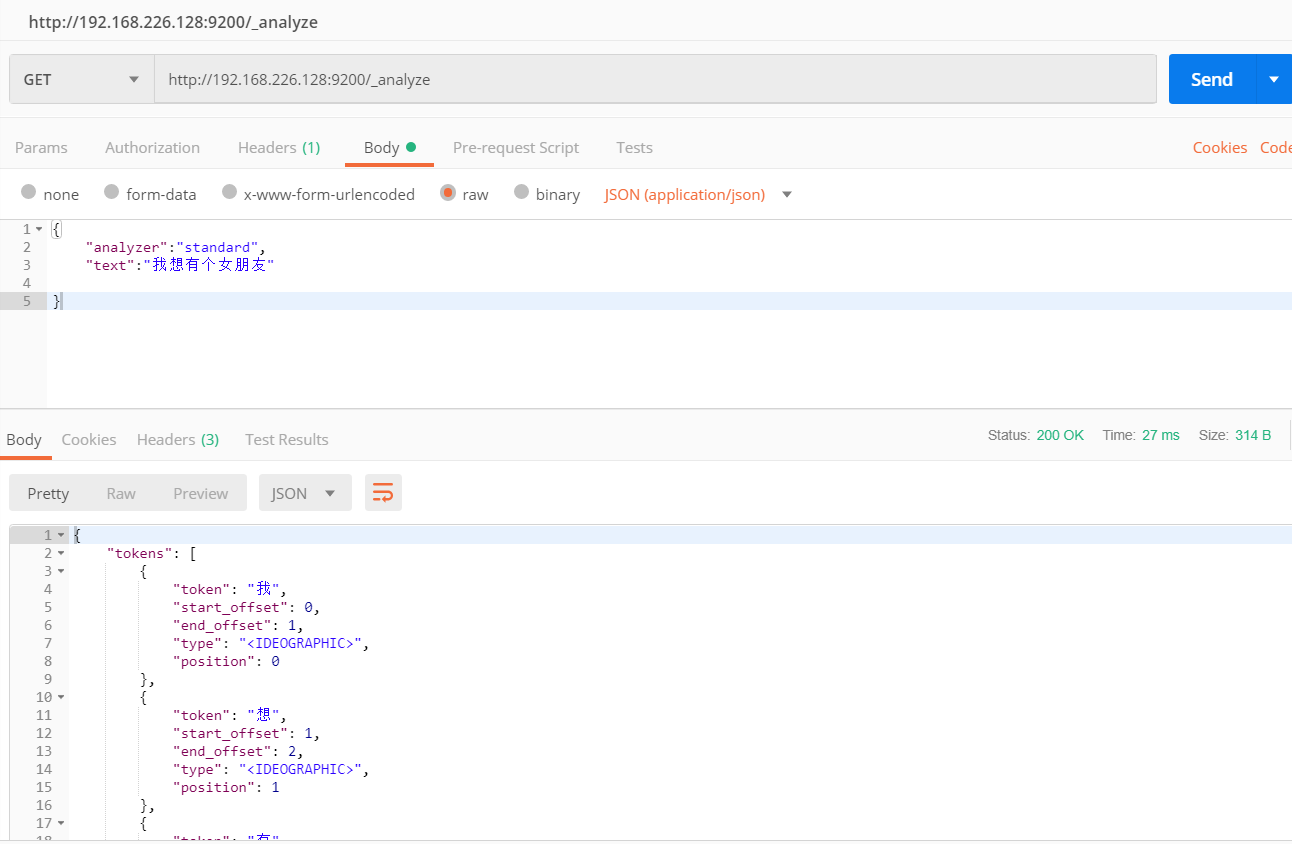
1. **重启elasticsearch后访问即可**

|  |
| --- |
| docker restart elasticsearch |

# IK分词的安装

Ik分词在es 里面也是插件的形式安装的

## 没有ik分词的时候

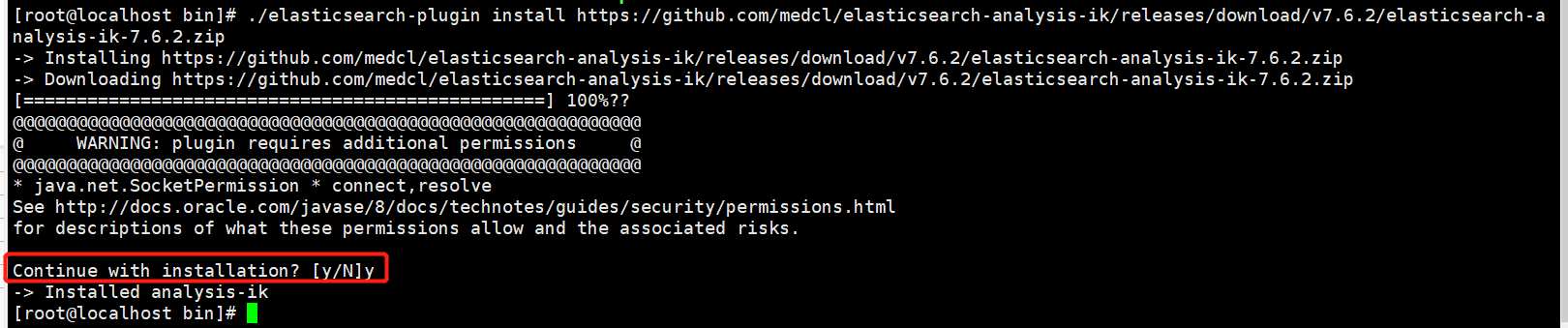


## IK分词安装（方式一受网速影响）

1. 找对应的ES版本的IK分词 <https://github.com/medcl/elasticsearch-analysis-ik/releases>
2. 进入容器 docker exec -it elasticsearch bash
3. 进入这个目录 /usr/share/elasticsearch/bin
4. 执行命令 install 后面的连接就是从上面找到的对应的版本下载链接

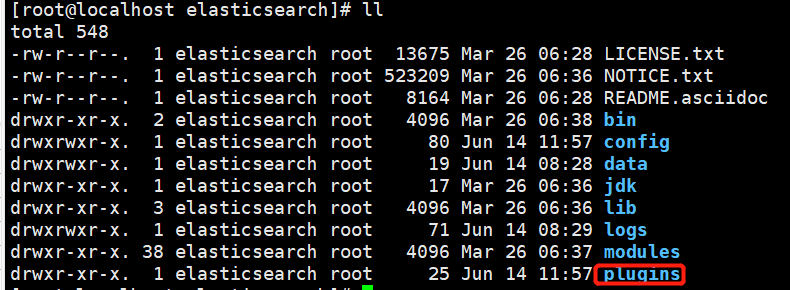
|  |
| --- |
| ./elasticsearch-plugin install https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v7.6.2/elasticsearch-analysis-ik-7.6.2.zip |

1. 下载完成



## IK分词安装（方式二推荐）

1. 找对应的ES版本的IK分词 <https://github.com/medcl/elasticsearch-analysis-ik/releases>
2. 下载到windows上
3. 将zip文件拷贝到linux上
4. docker cp linux的路径 elasticsearch:/usr/share/elasticsearch/plugins



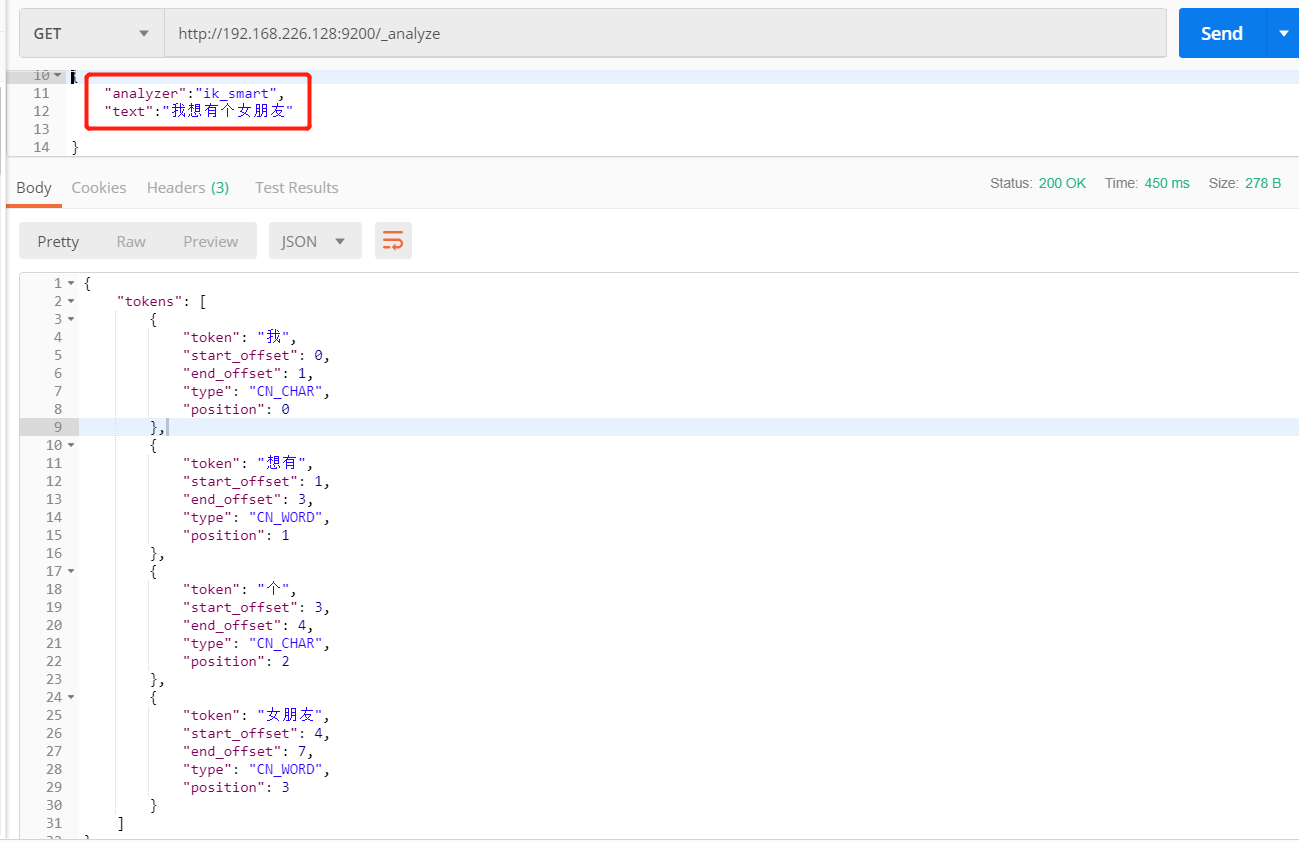
1. 进容器后解压，注意名字unzip elasticsearch-analysis-ik-7.6.2.zip -d ./ik/
2. 删掉zip文件 elasticsearch-analysis-ik-7.6.2.zip

## 重启ES测试

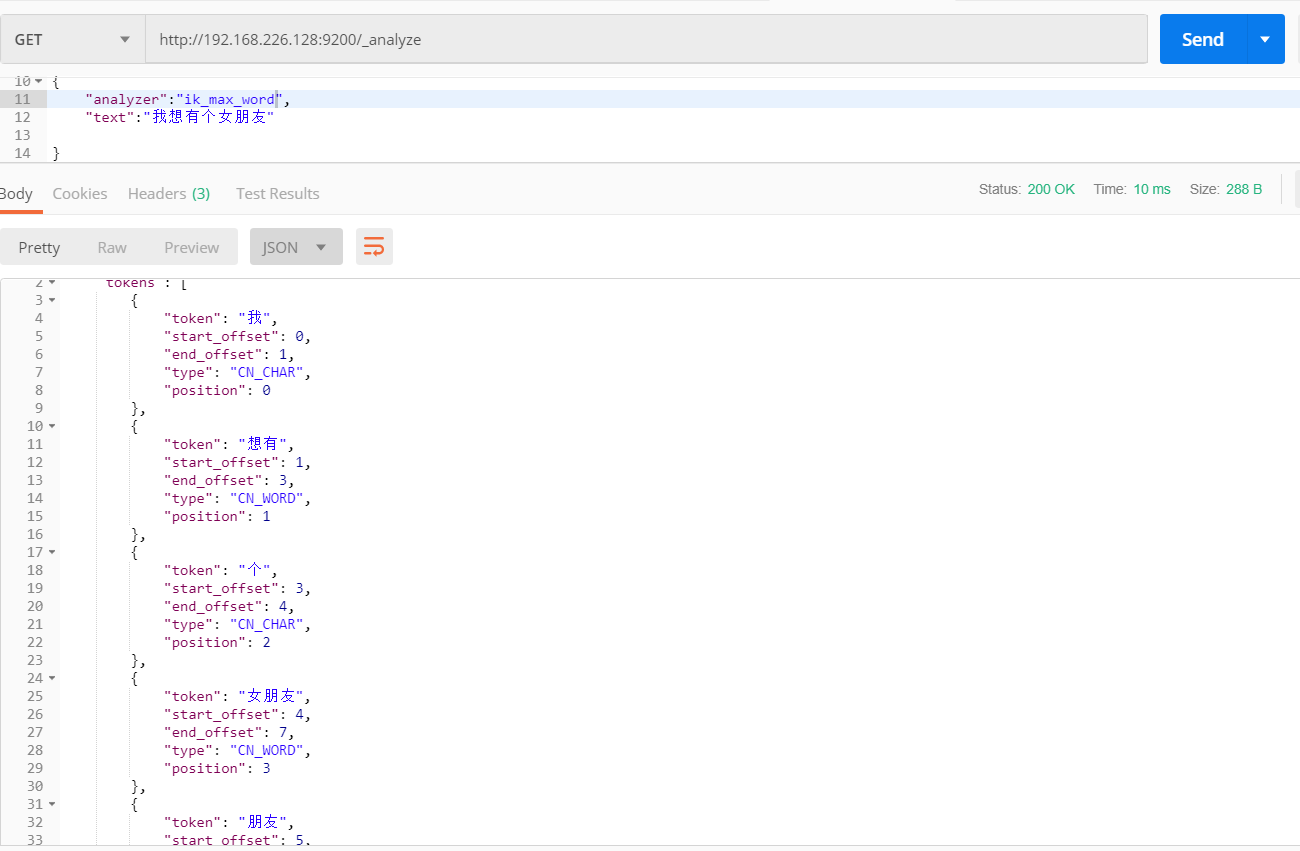
**docker restart elasticsearch**

**Ik分词的两种方式：**

**ik\_smart：**分词的粒度较小，也叫智能分词

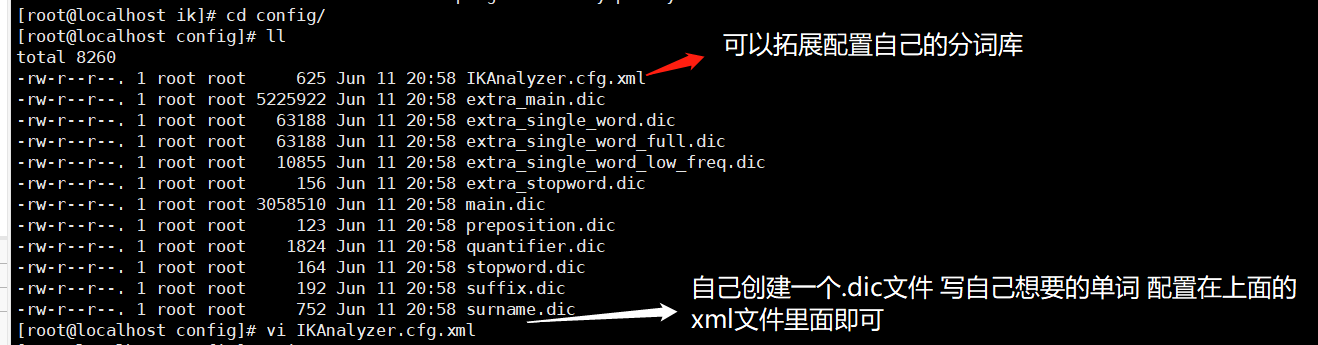


**ik\_max\_word:**分词的粒度较大，也叫最大力度分词



## 自定义分词

进入容器



# **Elasticsearch**核心概念

## 结构说明

|  |  |
| --- | --- |
| **Elasticsearch** | **Mysql** |
| Index(索引库) | Database(数据库) |
| Type | Table(表) |
| Document(文档) | Row(行) |
| Field(字段) | Column(列) |
| mappings | 列的类型(int,varchar..) |

这里的Type概念从ES6.x以后被弱化，官方将在ES8.0以后正式剔除他

## 索引库（indices）

把数据写入elasticsearch 时，会在里面建立索引，索引库里面存储索引，一个index 对应一个database

## 文档（document）

就是一条数据，一般使用json 表示，和数据库对比，就是一行数据，在java 里面就是一个一个对象

## 字段（field）

一个对象的属性，对应数据库就是一列

## 节点

一台运行elasticsearch的服务器，被称为一个节点

## 集群

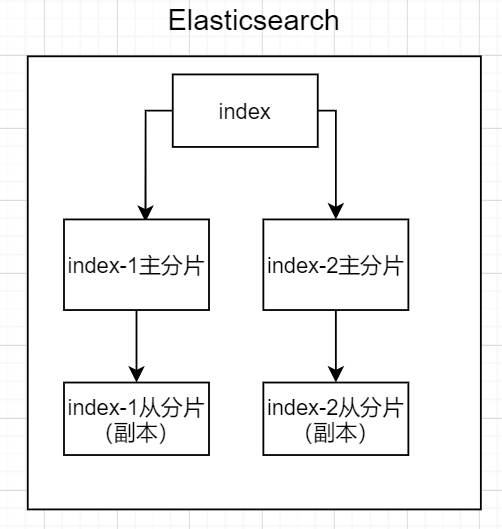
多个节点组成一个集群

## 分片

一个索引可以存储在多个**主分片**上，有负载均衡的作用，还有**从分片**是主分片的一个副本

## 副本

一份数据可以有多个副本，做数据冗余（安全），一般放在从分片里面



# Elasticsearch基本使用

Elasticsearch 是基于restful风格的http应用

Restfu风格就是使用http动词形式对url资源进行操作（GET,POST,PUT,DELETE...）

操作格式为：

**请求类型 /索引名/文档id**

**{请求体}**

## 对索引和mappings的操作（建库建表约束）

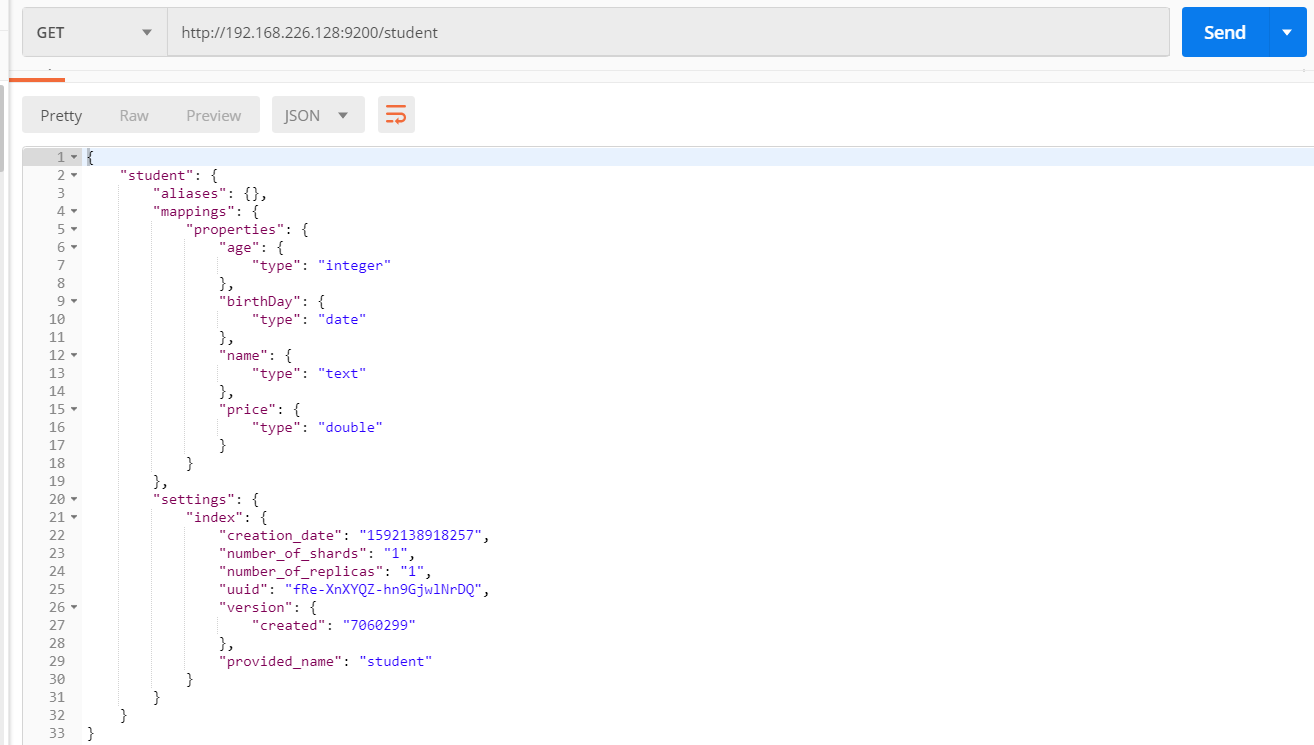
### 新增索引

PUT <http://192.168.226.128:9200/student> 新建一个student索引，给定几个字段约束

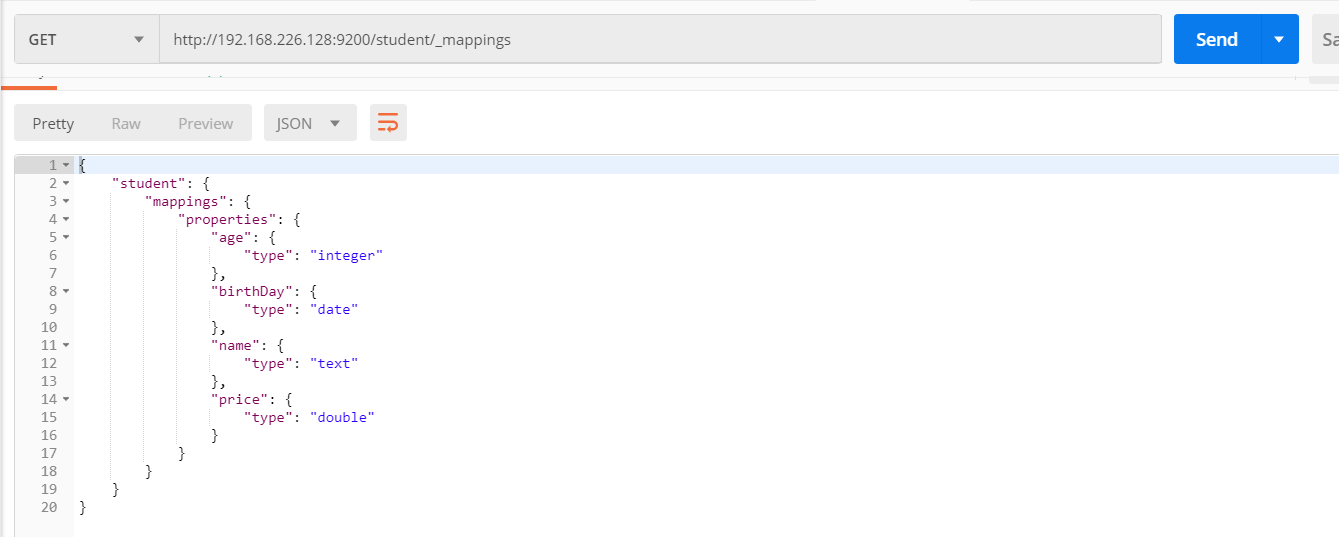
索引只能增删，不能修改

|  |
| --- |
| {  "mappings": {  "properties": {  "name":{  "type": "text"  },  "age":{  "type": "integer"  },  "birthDay":{  "type": "date"  },  "price":{  "type": "double"  }  }  }  } |

查询索引信息



查询索引的mappings信息



### 删除索引

DELETE <http://192.168.226.128:9200/student>

## 对Document的操作

### 新增数据(方便后面演示，自己多新增几条)

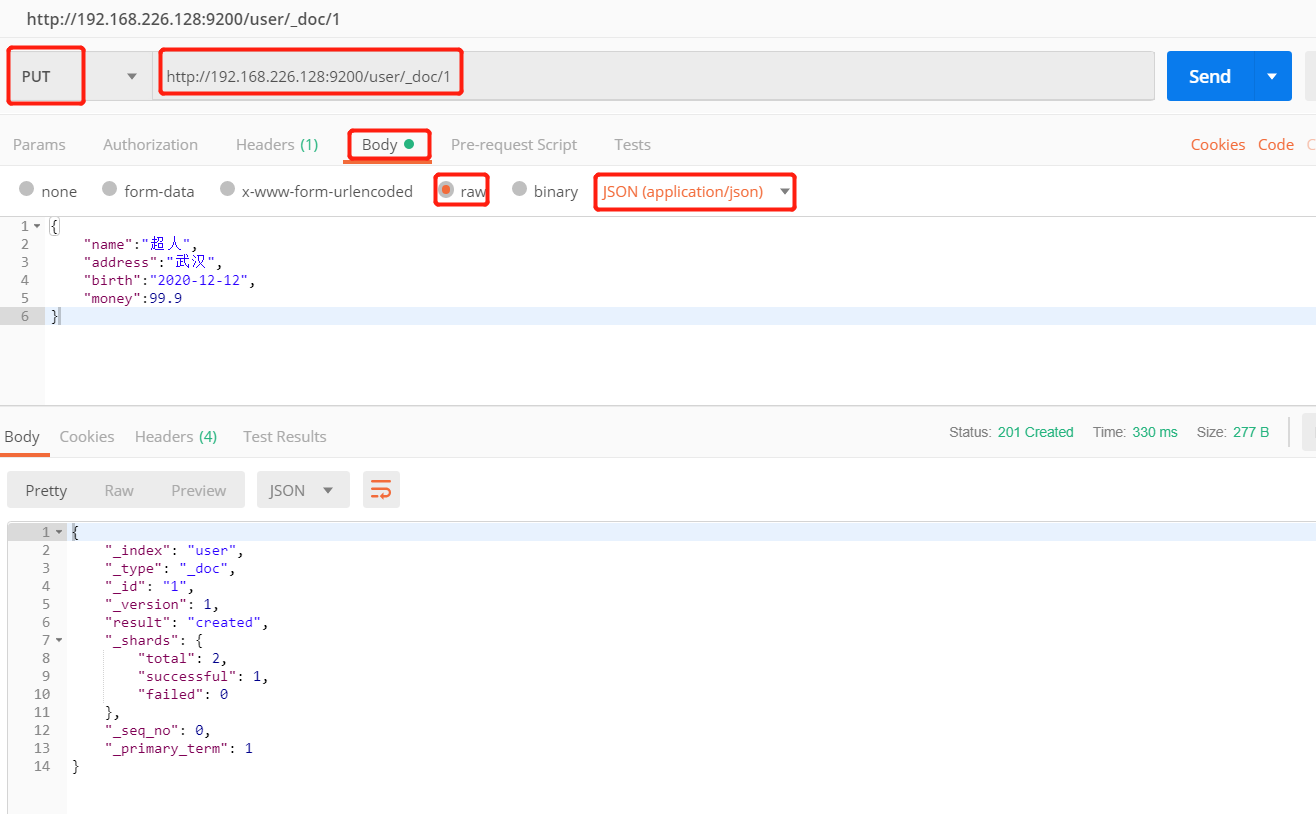
使用put请求新增时需要自己指定id，使用post请求新增时系统会自动生成一个id

PUT <http://192.168.226.128:9200/user/_doc/1> 请求解释

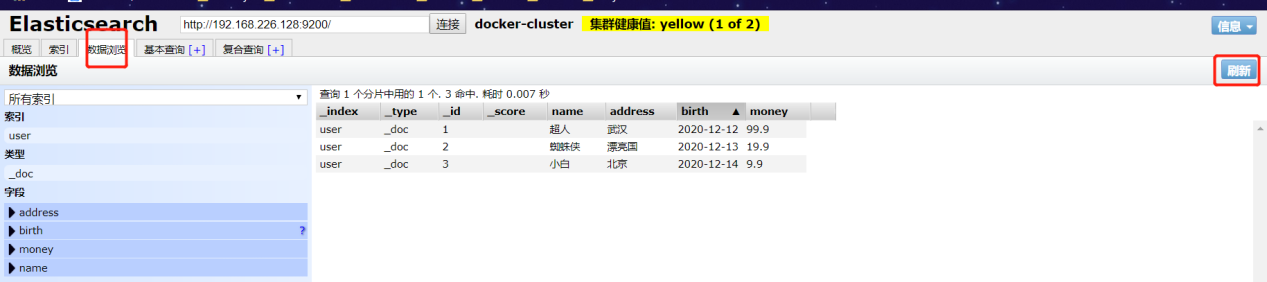
user: 索引名称

\_doc：类型（即将剔除，官方建议全部使用\_doc）

1: 文档id



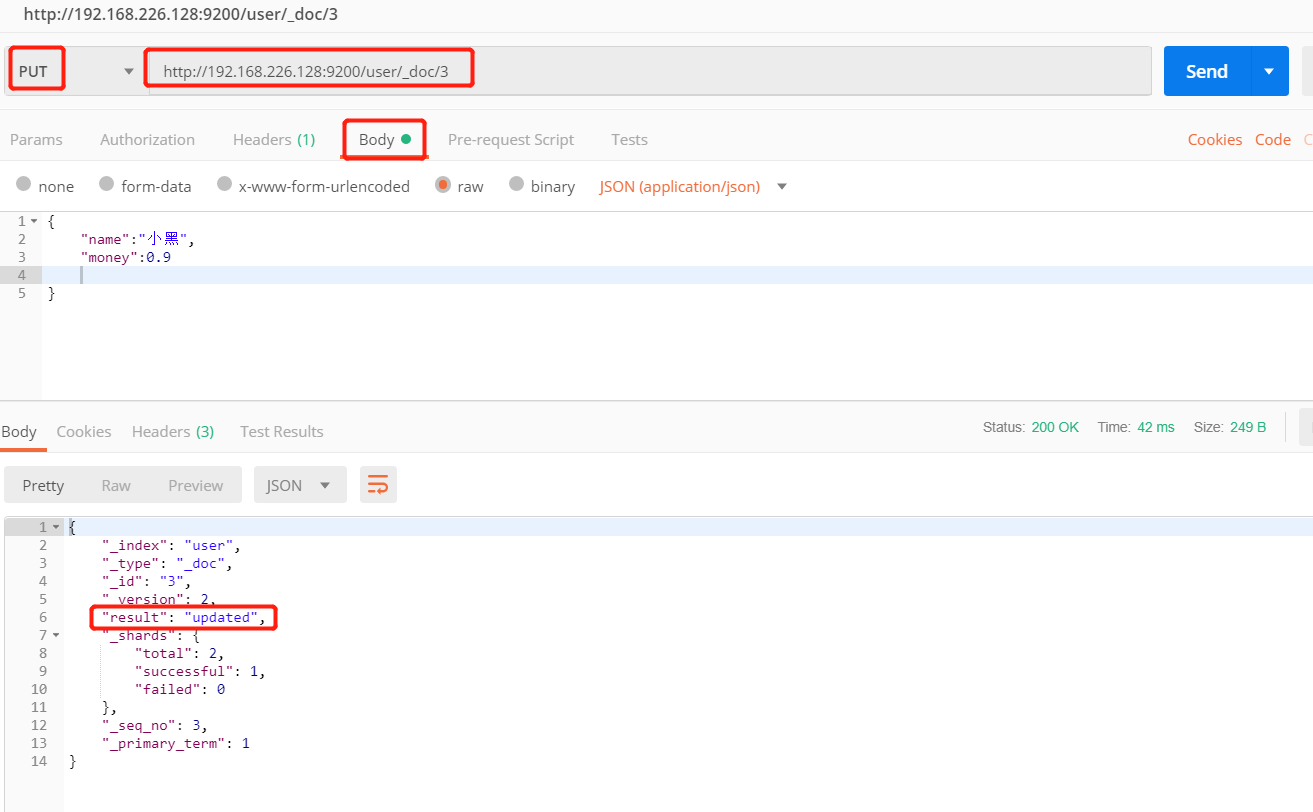
从head插件里面查看数据

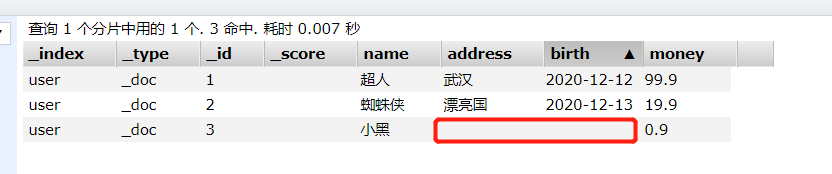


### 修改**一个**数据

1. 危险的修改，把其他的字段值都删了

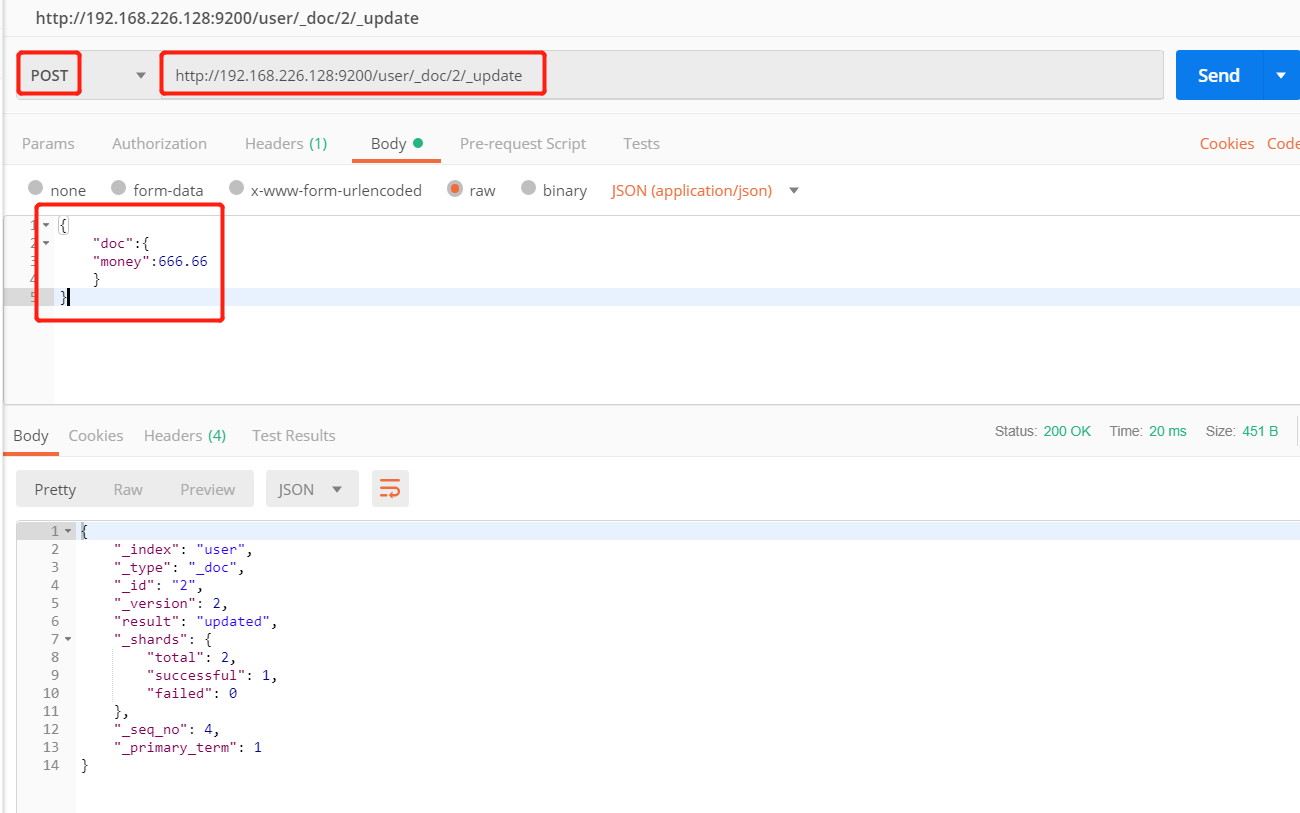
PUT <http://192.168.226.128:9200/user/_doc/2>





1. 安全的修改，其他的字段值会保留

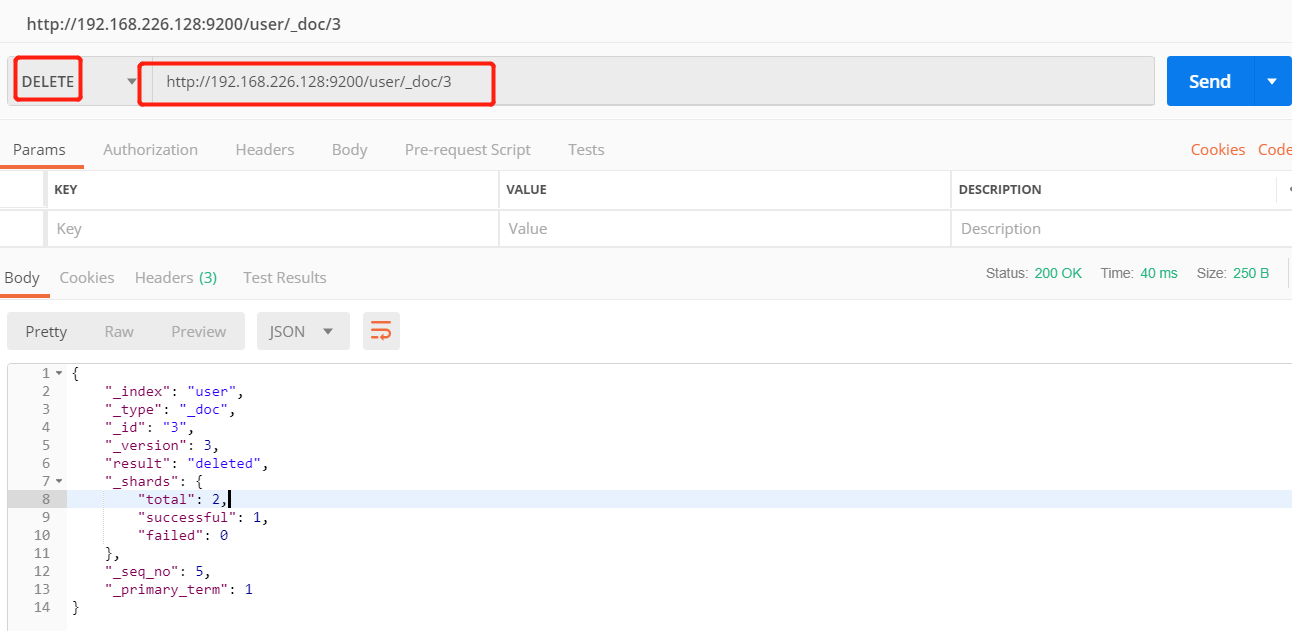
POST <http://192.168.226.128:9200/user/_doc/2/_update>





### 删除**一个**数据

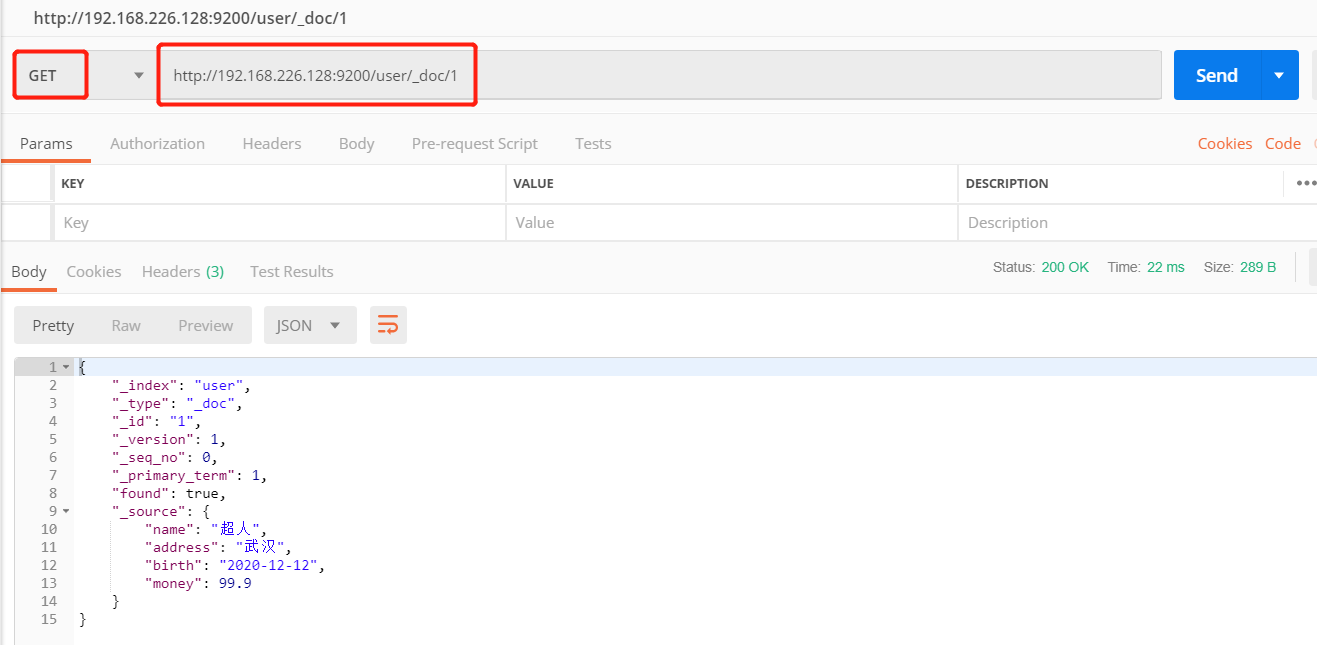
DELETE <http://192.168.226.128:9200/user/_doc/3>





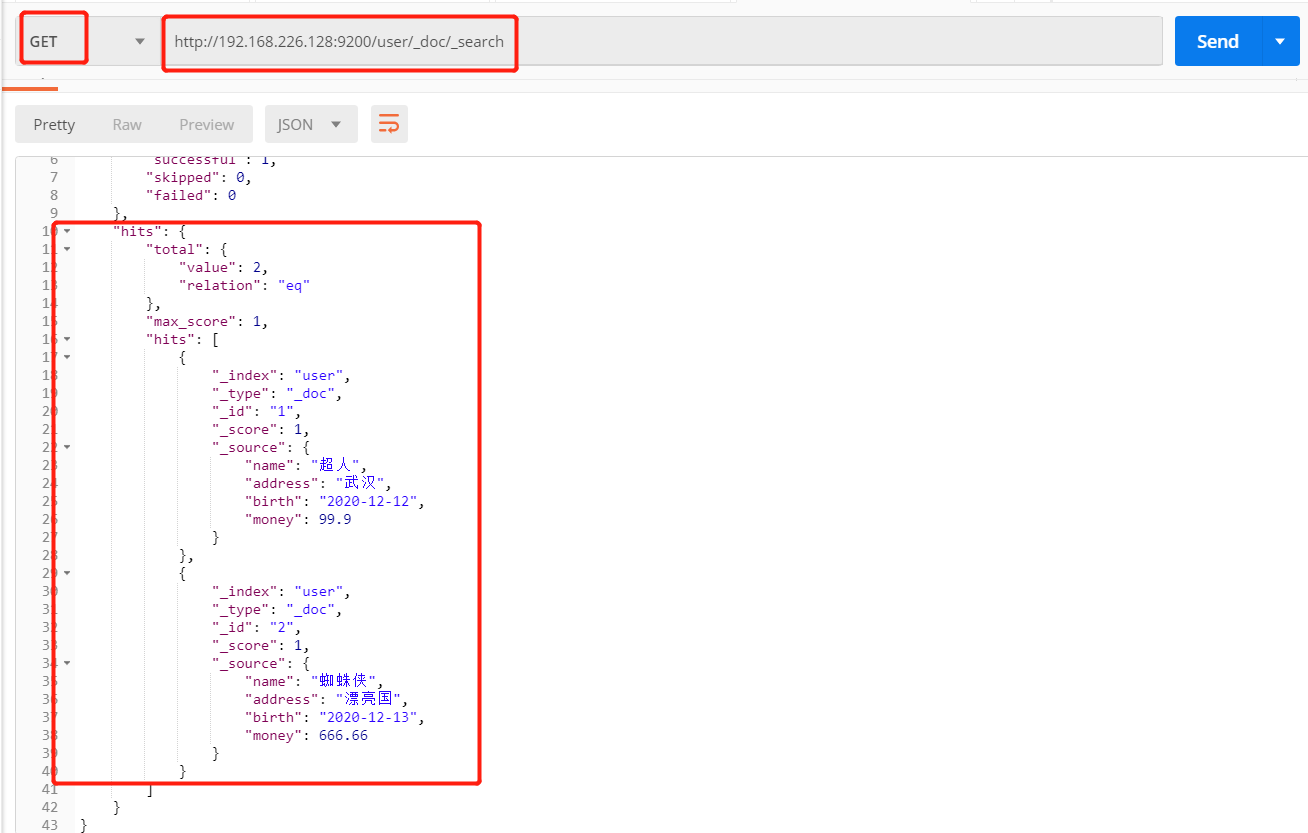
### 查询**一个**数据

GET <http://192.168.226.128:9200/user/_doc/1>



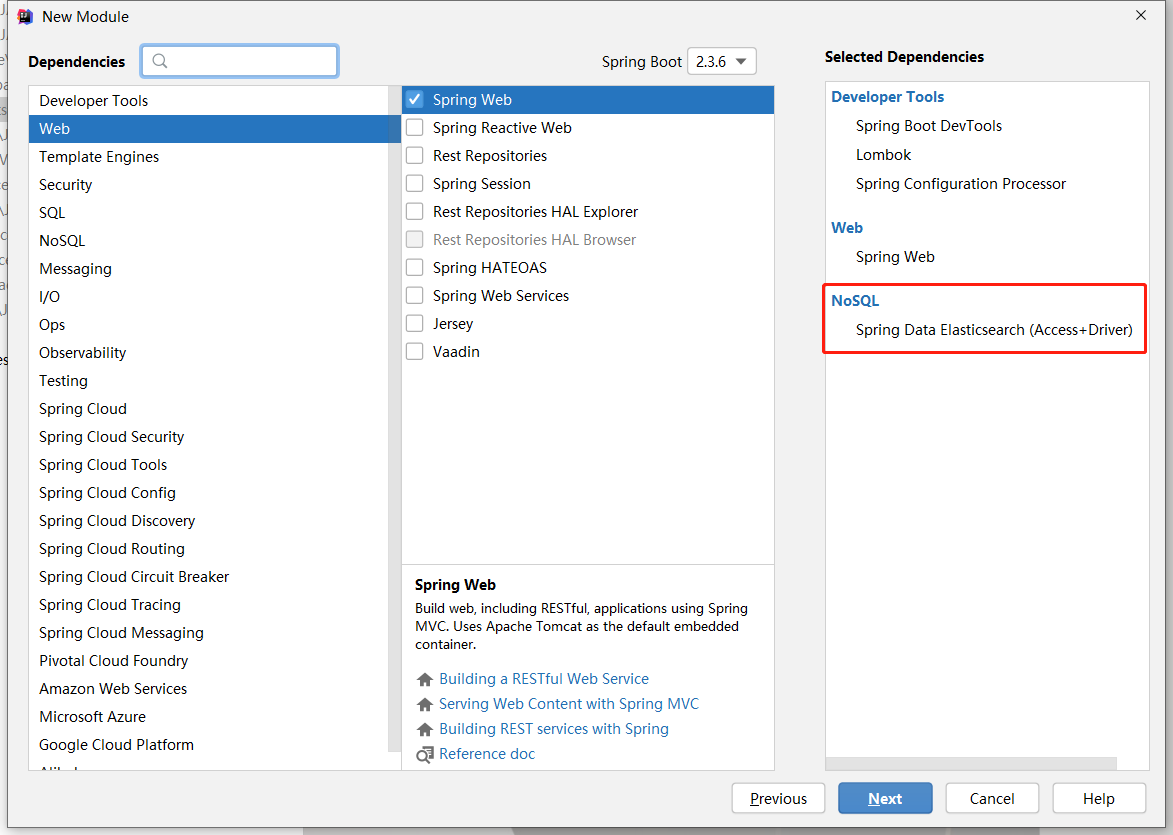
### 查询**全部**数据

GET <http://192.168.226.128:9200/user/_doc/_search>



# SpringBoot使用ES【重点】

## 新建项目选择依赖

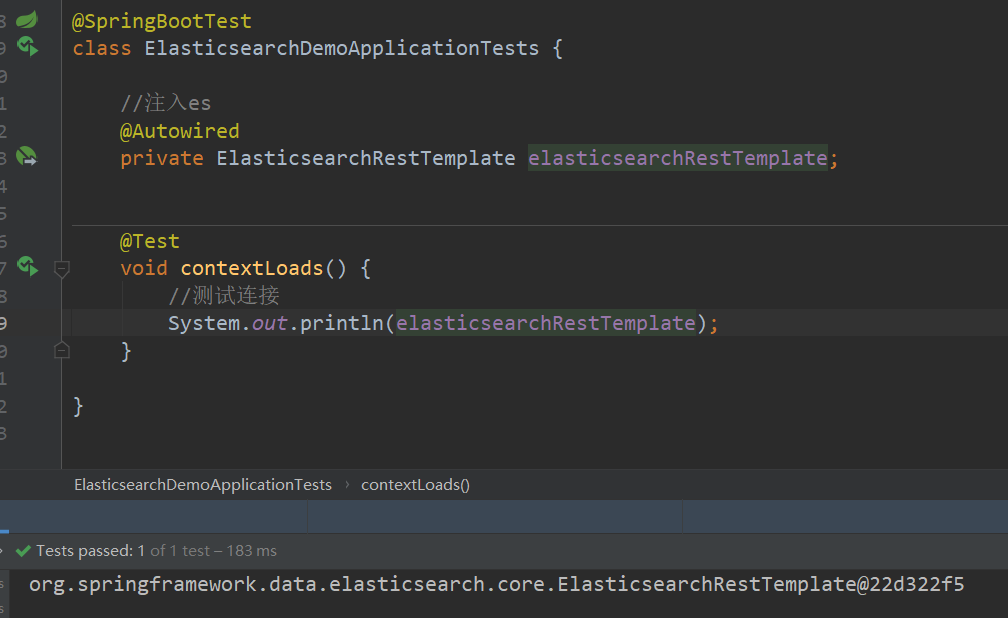


## 修改配置文件

|  |
| --- |
| spring:  elasticsearch:  rest: #连接es，相当于使用客户端操作  uris: http://192.168.226.129:9200 |

## 测试连接ES

说明我们连接成功，下面开始操作



## 对索引的操作，我们直接使用实体类操作

### 新建Goods实体类

|  |
| --- |
| /\*\*  \* @Document是ES提供的注解 indexName:索引名称  \* createIndex：启动时是否创建  \* shards：分片个数  \* replicas：副本个数  \* refreshInterval：数据导入到索引里面，最多几秒搜索到  \*/  @Data @AllArgsConstructor @NoArgsConstructor @Document(indexName = "goods\_index", createIndex = true, shards = 3, replicas = 1, refreshInterval = "1s") public class Goods {   /\*\*  \* 商品ID  \*/  @Id //默认使用keyword关键字模式，不进行分词  @Field  private Integer goodsId;  /\*\*  \* 商品名称  \* analyzer:导入时使用的分词  \* searchAnalyzer：搜索时使用的分词  \*/  @Field(type = FieldType.*Text*, analyzer = "ik\_max\_word", searchAnalyzer = "ik\_smart")  private String goodsName;  /\*\*  \* 商品描述  \*/  @Field(type = FieldType.*Text*, analyzer = "ik\_max\_word", searchAnalyzer = "ik\_smart")  private String goodsDesc;  /\*\*  \* 商品价格  \*/  @Field(type = FieldType.*Double*)  private Double goodsPrice;  /\*\*  \* 商品的销量  \*/  @Field(type = FieldType.*Long*)  private Long goodsSaleNum;  /\*\*  \* 商品的卖点  \*/  @Field(type = FieldType.*Text*, analyzer = "ik\_max\_word", searchAnalyzer = "ik\_smart")  private String goodsBrief;  /\*\*  \* 商品的状态  \*/  @Field(type = FieldType.*Integer*)  private Integer goodsStatus;  /\*\*  \* 商品的库存  \*/  @Field(type = FieldType.*Integer*)  private Integer goodsStock;  /\*\*  \* 商品的标签  \*/  @Field(type = FieldType.*Text*)  private List<String> goodsTags;  /\*\*  \* 上架的时间  \* 指定时间格式化  \*/  private Date goodsUpTime;  } |

### 创建索引和mappings约束

|  |
| --- |
| //注入es @Autowired private ElasticsearchRestTemplate elasticsearchRestTemplate;  @Test public void testIndexAndMappings() throws Exception {  //得到索引信息  IndexOperations indexOperations = elasticsearchRestTemplate.indexOps(Goods.class);  //创建mappings  Document mapping = indexOperations.createMapping(Goods.class);  //执行mapping   **indexOperations.putMapping(mapping);**   //从索引里面获取mappings  Map<String, Object> mappings = indexOperations.getMapping();  System.*out*.println(mappings); } |

## 对文档的操作，我们熟悉的CRUD

### 创建goodsDao

|  |
| --- |
| @Repository public interface GoodsDao extends ElasticsearchRepository<Goods, Integer> {  } |

### 新增数据

|  |
| --- |
| @Autowired private GoodsDao goodsDao;  /\*\*  \* 描述: 对document的curd  \*  \* @param :  \* @return void  \*/ @Test public void testDocumentCurd() throws Exception {  //新增商品数据100条  ArrayList<Goods> goods = new ArrayList<>(200);  for (int i = 1; i <= 100; i++) {  goods.add(  new Goods(i,  i % 2 == 0 ? "华为电脑" + i : "联想电脑" + i,  i % 2 == 0 ? "轻薄笔记本" + i : "游戏笔记本" + i,  4999.9 + i,  999L,  i % 2 == 0 ? "华为续航强" : "联想性能强",  i % 2 == 0 ? 1 : 0,  666 + i,  i % 2 == 0 ? Arrays.*asList*("小巧", "轻薄", "续航") : Arrays.*asList*("炫酷", "畅玩", "游戏"),  new Date())  );  }  goodsDao.saveAll(goods); } |

### 修改数据(注意是危险修改)

|  |
| --- |
| @Test public void testUpdate() throws Exception {  goodsDao.save(new Goods(1, "小米笔记本", null, null, null, null, null, null, null, null)); } |

### 删除数据

|  |
| --- |
| @Test public void testDelete() throws Exception{  goodsDao.deleteById(1); } |

### 根据id查询数据

|  |
| --- |
| @Test public void testSearchById() throws Exception {  Optional<Goods> byId = goodsDao.findById(502);  System.*out*.println(byId.get()); } |

### 查询所有数据

|  |
| --- |
| @Test public void testSearchAll() throws Exception {  Iterable<Goods> all = goodsDao.findAll();  all.forEach(System.*out*::println); } |

# 复杂的查询操作

## 查询注意点

|  |
| --- |
| match:会通过分词器去模糊匹配 例如：华为电脑，会把包含‘华为’，‘电脑’，都查出来  term:精确查找你的关键字,一般使用keywords的约束，使用term  rang：范围查询  match 和 rang 如果同时出现，需要组合bool查询  分页，排序是通用的查询，不需要通过bool组合使用，直接nativeSearchQueryBuilder使用 |

## 查询常用类

|  |
| --- |
| QueryBuilders:构造条件对象，例如matchQuery，rangeQuery，boolQuery等  NativeSearchQueryBuilder:组合条件对象，组合后使用build构建查询对象  HighlightBuilder:高亮的查询类，注意使用它的Field静态内部类  FunctionScoreQueryBuilder：权重类，注意它的FilterFunctionBuilder静态内部类 |

## 关键字，范围，分页，排序

|  |
| --- |
| @Test public void testFuZaSearch() throws Exception {  //关键字，“华为” ，范围，分页，排序  MatchQueryBuilder matchQueryBuilder = QueryBuilders.*matchQuery*("goodsName", "华为");  RangeQueryBuilder rangeQueryBuilder = QueryBuilders.*rangeQuery*("goodsStock").from(700).to(750);   //使用bool组合这两个查询  BoolQueryBuilder boolQueryBuilder = QueryBuilders.*boolQuery*().must(matchQueryBuilder).must(rangeQueryBuilder);   //创建组合器  NativeSearchQueryBuilder nativeSearchQueryBuilder = new NativeSearchQueryBuilder();  //去build()构建查询对象  NativeSearchQuery nativeSearchQuery = nativeSearchQueryBuilder  .withQuery(boolQueryBuilder)  .withPageable(PageRequest.*of*(1, 20)) //注意范围和分页有关系，可能查出来了，但是当前分页没有  .withSort(SortBuilders.*fieldSort*("goodsPrice").order(SortOrder.*ASC*))  .build();   //使用es查询 得到结果集  SearchHits<Goods> searchHits = elasticsearchRestTemplate.search(nativeSearchQuery, Goods.class);  searchHits.forEach(goodsSearchHit -> {  //循环结果集，答应结果  System.*out*.println(goodsSearchHit.getContent());  }); } |

## 高亮查询

|  |
| --- |
| @Test  public void testHighlight() throws Exception {  //华为，模糊匹配  MatchQueryBuilder matchQueryBuilder = QueryBuilders.*matchQuery*("goodsName", "华为");   HighlightBuilder.Field goodsName = new HighlightBuilder.Field("goodsName").preTags("<span style='color:red'>").postTags("</span>");   NativeSearchQueryBuilder nativeSearchQueryBuilder = new NativeSearchQueryBuilder();   NativeSearchQuery nativeSearchQuery = nativeSearchQueryBuilder  .withQuery(matchQueryBuilder)  .withHighlightFields(goodsName)  .build();  //得到结果集 我们需要手动组装高亮字段  SearchHits<Goods> searchHits = elasticsearchRestTemplate.search(nativeSearchQuery, Goods.class);   List<Goods> goodsArrayList = new ArrayList<>();   searchHits.forEach(goodsSearchHit -> {  //得到goods对象,但是这里面额goodsName属性不是高亮的，所以要改  Goods goods = goodsSearchHit.getContent();  List<String> highlightField = goodsSearchHit.getHighlightField("goodsName");  String highlight = highlightField.get(0);  goods.setGoodsName(highlight);  goodsArrayList.add(goods);  });  System.*out*.println(JSON.toJSONString(goodsArrayList));  } } |

## 权重查询

|  |
| --- |
| @Test public void testWeight() throws Exception {  //手机名称 和卖点 都有的情况下，设置权重查询  String keyWords = "华为";  //创建权重数组  FunctionScoreQueryBuilder.FilterFunctionBuilder[] functionBuilders = new FunctionScoreQueryBuilder.FilterFunctionBuilder[2];   //设置权重  functionBuilders[0] = (new FunctionScoreQueryBuilder.FilterFunctionBuilder(  QueryBuilders.*matchQuery*("goodsName", keyWords),  ScoreFunctionBuilders.*weightFactorFunction*(10)//给名称设置10的权重大小  ));  functionBuilders[1] = (  new FunctionScoreQueryBuilder.FilterFunctionBuilder(  QueryBuilders.*matchQuery*("goodsBrief", keyWords),  ScoreFunctionBuilders.*weightFactorFunction*(4)//给卖点设置4的权重  ));   FunctionScoreQueryBuilder functionScoreQueryBuilder = new FunctionScoreQueryBuilder(functionBuilders);   functionScoreQueryBuilder.setMinScore(2) //设置最小分数  .scoreMode(FunctionScoreQuery.ScoreMode.*FIRST*);//设置计分方式   NativeSearchQuery nativeSearchQuery = new NativeSearchQueryBuilder().withQuery(functionScoreQueryBuilder).build();  SearchHits<Goods> searchHits = elasticsearchRestTemplate.search(nativeSearchQuery, Goods.class);  searchHits.forEach(goodsSearchHit -> {  //循环结果集，答应结果  System.*out*.println(goodsSearchHit.getContent());  }); } |

# ES集群（了解）

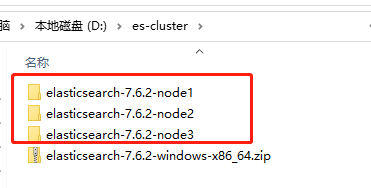
这里使用windows方式演示集群

Linux上的集群道理一样，修改配置文件即可

可以参考https://my.oschina.net/u/4353003/blog/4333773

## 创建三个es节点

解压出三个来



## 修改配置文件

### Node1修改配置文件

进入elasticsearch-7.6.2-node1\config下，修改elasticsearch.yml

|  |
| --- |
| **# 设置集群名称，集群内所有节点的名称必须一致。**  **cluster.name: my-esCluster**  **# 设置节点名称，集群内节点名称必须唯一。**  **node.name: node1**  # 表示该节点会不会作为主节点，true表示会；false表示不会  node.master: true  # 当前节点是否用于存储数据，是：true、否：false  node.data: true  # 索引数据存放的位置  #path.data: /opt/elasticsearch/data  # 日志文件存放的位置  #path.logs: /opt/elasticsearch/logs  # 需求锁住物理内存，是：true、否：false  #bootstrap.memory\_lock: true  **# 监听地址，用于访问该es**  **network.host: 0.0.0.0**  **# es对外提供的http端口，默认 9200**  **http.port: 9200**  **# TCP的默认监听端口，默认 9300**  **transport.tcp.port: 9300**  # 设置这个参数来保证集群中的节点可以知道其它N个有master资格的节点。默认为1，对于大的集群来说，可以设置大一点的值（2-4）  discovery.zen.minimum\_master\_nodes: 2  **# es7.x 之后新增的配置，写入候选主节点的设备地址，在开启服务后可以被选为主节点**  **discovery.seed\_hosts: ["192.168.186.1:9300", "192.168.186.1:9301", "192.168.186.1:9302"]**  discovery.zen.fd.ping\_timeout: 1m  discovery.zen.fd.ping\_retries: 5  **# es7.x 之后新增的配置，初始化一个新的集群时需要此配置来选举master**  **cluster.initial\_master\_nodes: ["node1", "node2", "node3"]**  **# 是否支持跨域，是：true，在使用head插件时需要此配置**  **http.cors.enabled: true**  **# “\*” 表示支持所有域名**  **http.cors.allow-origin: "\*"**  action.destructive\_requires\_name: true  action.auto\_create\_index: .security,.monitoring\*,.watches,.triggered\_watches,.watcher-history\*  xpack.security.enabled: false  xpack.monitoring.enabled: true  xpack.graph.enabled: false  xpack.watcher.enabled: false  xpack.ml.enabled: false |

### Node2修改配置文件

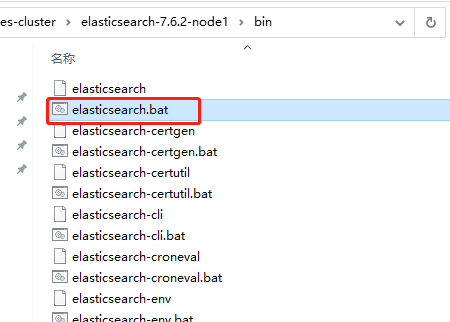
|  |
| --- |
| **# 设置集群名称，集群内所有节点的名称必须一致。**  **cluster.name: my-esCluster**  **# 设置节点名称，集群内节点名称必须唯一。**  **node.name: node2**  **# 表示该节点会不会作为主节点，true表示会；false表示不会**  **node.master: true**  # 当前节点是否用于存储数据，是：true、否：false  node.data: true  # 索引数据存放的位置  #path.data: /opt/elasticsearch/data  # 日志文件存放的位置  #path.logs: /opt/elasticsearch/logs  # 需求锁住物理内存，是：true、否：false  #bootstrap.memory\_lock: true  **# 监听地址，用于访问该es**  **network.host: 0.0.0.0**  **# es对外提供的http端口，默认 9200**  **http.port: 9201**  **# TCP的默认监听端口，默认 9300**  **transport.tcp.port: 9301**  # 设置这个参数来保证集群中的节点可以知道其它N个有master资格的节点。默认为1，对于大的集群来说，可以设置大一点的值（2-4）  discovery.zen.minimum\_master\_nodes: 2  # es7.x 之后新增的配置，写入候选主节点的设备地址，在开启服务后可以被选为主节点  **discovery.seed\_hosts: ["192.168.186.1:9300", "192.168.186.1:9301", "192.168.186.1:9302"]**  discovery.zen.fd.ping\_timeout: 1m  discovery.zen.fd.ping\_retries: 5  # es7.x 之后新增的配置，初始化一个新的集群时需要此配置来选举master  cluster.initial\_master\_nodes: ["node1", "node2", "node3"]  **# 是否支持跨域，是：true，在使用head插件时需要此配置**  **http.cors.enabled: true**  **# “\*” 表示支持所有域名**  **http.cors.allow-origin: "\*"**  action.destructive\_requires\_name: true  action.auto\_create\_index: .security,.monitoring\*,.watches,.triggered\_watches,.watcher-history\*  xpack.security.enabled: false  xpack.monitoring.enabled: true  xpack.graph.enabled: false  xpack.watcher.enabled: false  xpack.ml.enabled: false |

### Node3修改配置文件

|  |
| --- |
| **# 设置集群名称，集群内所有节点的名称必须一致。**  **cluster.name: my-esCluster**  **# 设置节点名称，集群内节点名称必须唯一。**  **node.name: node3**  **# 表示该节点会不会作为主节点，true表示会；false表示不会**  **node.master: true**  # 当前节点是否用于存储数据，是：true、否：false  node.data: true  # 索引数据存放的位置  #path.data: /opt/elasticsearch/data  # 日志文件存放的位置  #path.logs: /opt/elasticsearch/logs  # 需求锁住物理内存，是：true、否：false  #bootstrap.memory\_lock: true  **# 监听地址，用于访问该es**  **network.host: 0.0.0.0**  **# es对外提供的http端口，默认 9200**  **http.port: 9202**  **# TCP的默认监听端口，默认 9300**  **transport.tcp.port: 9302**  # 设置这个参数来保证集群中的节点可以知道其它N个有master资格的节点。默认为1，对于大的集群来说，可以设置大一点的值（2-4）  discovery.zen.minimum\_master\_nodes: 2  # es7.x 之后新增的配置，写入候选主节点的设备地址，在开启服务后可以被选为主节点  **discovery.seed\_hosts: ["192.168.186.1:9300", "192.168.186.1:9301", "192.168.186.1:9302"]**  discovery.zen.fd.ping\_timeout: 1m  discovery.zen.fd.ping\_retries: 5  # es7.x 之后新增的配置，初始化一个新的集群时需要此配置来选举master  cluster.initial\_master\_nodes: ["node1", "node2", "node3"]  **# 是否支持跨域，是：true，在使用head插件时需要此配置**  **http.cors.enabled: true**  **# “\*” 表示支持所有域名**  **http.cors.allow-origin: "\*"**  action.destructive\_requires\_name: true  action.auto\_create\_index: .security,.monitoring\*,.watches,.triggered\_watches,.watcher-history\*  xpack.security.enabled: false  xpack.monitoring.enabled: true  xpack.graph.enabled: false  xpack.watcher.enabled: false  xpack.ml.enabled: false |

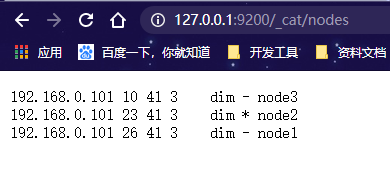
## 启动es

进入bin目录下 逐个启动，三台全部双击启动，注意不要关闭黑窗口



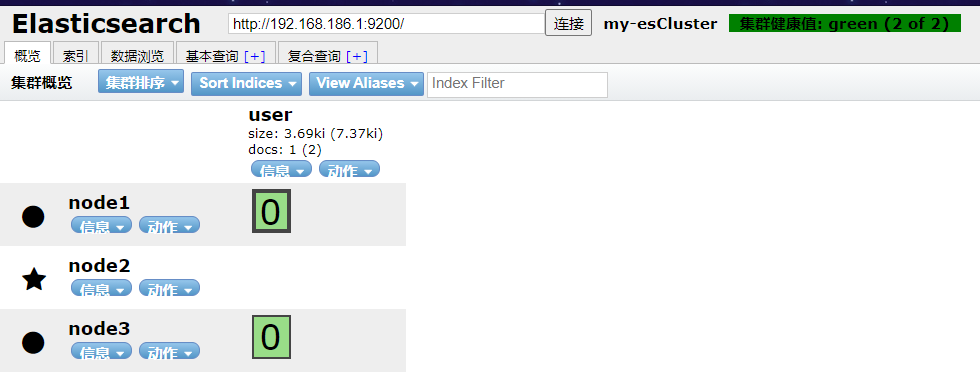
## 访问查看集群信息和状态

访问查看：<http://127.0.0.1:9200/_cat/nodes>



也可以使用head插件查看

## 测试插入数据



## SpringBoot连接es集群

|  |
| --- |
| spring:  elasticsearch:  rest:  uris:  - http://127.0.0.1:9200  - http://127.0.0.1:9201  - http://127.0.0.1:9202 |

# Es面试

## 为什么使用es

商城中的数据，将来会非常多，所以采用以往的模糊查询，模糊查询前置配置，会放弃索引，导致商品查询是全表扫面，在百万级别的数据库中，效率非常低下，而我们使用ES做一个全文索引，我们将经常查询的商品的某些字段，比如说商品名，描述、价格还有id这些字段我们放入我们索引库里，可以提高查询速度。

## es用来做什么

1. 商品存放
2. 日志统计
3. 数据分析（只要是电商，就要和tb，jd等对比价格，做数据分析等）

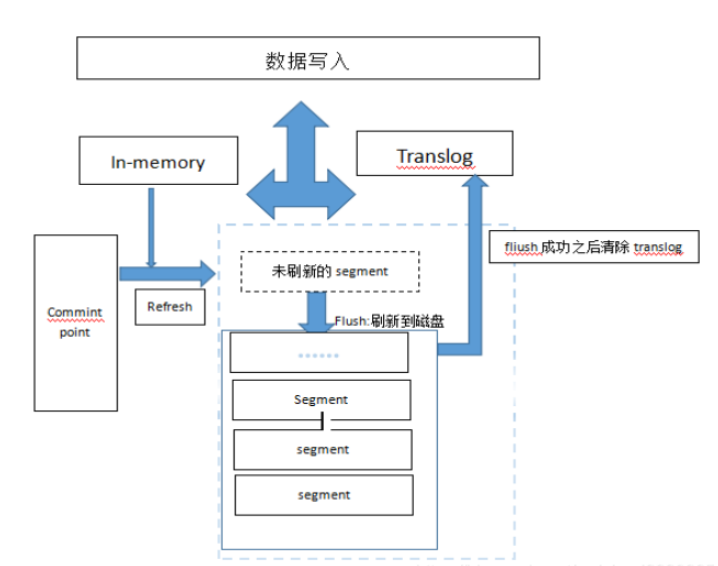
## 什么是倒排索引

通常正排索引是通过id映射到对应的数据

倒排索引是将分词建立索引，通过分词映射到id，在通过id找到数据

详见文档4.2

## es的存储数据的过程



1：写入请求，分发节点。

2：数据写入**同时写入内存和translog各一份**，tanslog为保证数据不丢失，每 5 秒，或每次请求操作结束前，会强制刷新 translog 日志到磁盘上

3：确定数据给那个分片，**refresh 刷新内存中数据到分片的segment**，默认1秒刷新一次，为了提高吞吐量可以增大60s。参数refresh\_interval（**refresh操作使得写入文档搜索可见**）

4：**通过flush操作将segment刷新到磁盘中完成持久化，保存成功清除translog**，新版本es的 translog 不会在 segment 落盘就删，而是会保留，默认是512MB，保留12小时。每个分片。所以分片多的话 ，要考虑 translog 会带来的额外存储开销（**flush操作使得filesystem cache写入磁盘，以达到持久化的目的**）

5：segment过多会进行合并merge为大的segment，消耗大量的磁盘io和网络io

## es的搜索过程

1、搜索被执行成一个**两阶段过程**，我们称之为 **Query** Then **Fetch**；

2、在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分

片）。 每个分片在本地执行搜索并构建一个匹配文档的大小为 from + size 的

优先队列。

注意：在搜索的时候是会查询 Filesystem Cache 的，但是有部分数据还在 Memory

Buffer，所以搜索是接近实时的。

3、每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，它合并

这些值到自己的优先队列中来产生一个全局排序后的结果列表。

4、接下来就是 取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片

提交多个 GET 请求。每个分片加载并 丰富 文档，如果有需要的话，接着返回

文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。

**通俗的将就是：**

**每个分片先拿到id和排序值，然后整合成一个全局列表**

**然后通过判断找到相应的节点提交多个get请求，组装数据返回**

5、补充：Query Then Fetch 的搜索类型在文档相关性打分的时候参考的是本分

片的数据，这样在文档数量较少的时候可能不够准确，DFS Query Then Fetch 增

加了一个预查询的处理，询问 Term 和 Document frequency，这个评分更准确，

但是性能会变差。