**ElasticSearch7.6入门学习笔记**

[[](https://www.kuangstudy.com/user/e2f44cb9925b4b3db0de979e4f711a6f) 流柚](https://www.kuangstudy.com/user/e2f44cb9925b4b3db0de979e4f711a6f)分类：[学习笔记](https://www.kuangstudy.com/bbs?cid=4) 浏览：4148 [评论：9](https://www.kuangstudy.com/bbs/1354069127022583809#comments) [字体](javascript:void(0);) [皮肤](javascript:void(0);) [**收藏** 左侧展开](javascript:void(0);)最后修改于： 2021/01/26 22:10:12

[展开目录+](javascript:void(0);)

笔记记录 B站狂神说Java的ElasticSearch课程：<https://www.bilibili.com/video/BV17a4y1x7zq>

在学习ElasticSearch之前，先简单了解一下**Lucene**：

* **Doug Cutting开发**
* 是**apache软件基金会**4 jakarta项目组的一个子项目
* 是一个**开放源代码**的**全文检索引擎工具包**
* **不是一个完整的全文检索引擎，而是一个全文检索引擎的架构**，提供了完整的查询引擎和索引引擎，部分[文本分析](https://baike.baidu.com/item/%E6%96%87%E6%9C%AC%E5%88%86%E6%9E%90/11046544)引擎（英文与德文两种西方语言）
* 当前以及最近几年最受欢迎的**免费Java信息检索程序库**。

**Lucene和ElasticSearch的关系：**

* ElasticSearch是基于Lucene 做了一下封装和增强

**一、ElasticSearch概述**

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

**Elaticsearch**，简称为es，es是一个开源的**高扩展**的**分布式全文检索引擎**，它可以近乎**实时的存储**、**检索数据;**本身扩展性很好，可以扩展到上百台服务器，处理PB级别(大数据时代）的数据。es也使用java开发并使用Lucene作为其核心来实现所有索引和搜索的功能，但是它的**目的**是<mark>通过简单的**RESTful API**来隐藏Lucene的复杂性，从而让全文搜索变得简单</mark>。

据国际权威的数据库产品评测机构DB Engines的统计，在2016年1月，ElasticSearch已超过Solr等，成为**排名第一的搜索引擎类应用**。

**历史**

多年前，一个叫做Shay Banon的刚结婚不久的失业开发者，由于妻子要去伦敦学习厨师，他便跟着也去了。在他找工作的过程中，为了给妻子构建一个食谱的搜索引擎，他开始构建一个早期版本的Lucene。

直接基于Lucene工作会比较困难，所以Shay开始抽象Lucene代码以便lava程序员可以在应用中添加搜索功能。他发布了他的第一个开源项目，叫做“Compass”。

后来Shay找到一份工作，这份工作处在高性能和内存数据网格的分布式环境中，因此高性能的、实时的、分布式的搜索引擎也是理所当然需要的。然后他决定重写Compass库使其成为一个独立的服务叫做Elasticsearch。

第一个公开版本出现在2010年2月，在那之后Elasticsearch已经成为Github上最受欢迎的项目之一，代码贡献者超过300人。一家主营Elasticsearch的公司就此成立，他们一边提供商业支持一边开发新功能，不过Elasticsearch将永远开源且对所有人可用。

Shay的妻子依旧等待着她的食谱搜索…..

**谁在使用：**

1、维基百科,类似百度百科，全文检索,高亮,搜索推荐/2  
2、The Guardian (国外新闻网站) ,类似搜狐新闻,用户行为日志(点击,浏览,收藏,评论) +社交网络数据(对某某新闻的相关看法) ,数据分析,给到每篇新闻文章的作者,让他知道他的文章的公众反馈(好,坏,热门，垃圾,鄙视，崇拜)  
3、Stack Overflow (国外的程序异常讨论论坛) , IT问题,程序的报错,提交上去,有人会跟你讨论和回答,全文检索,搜索相关问题和答案,程序报错了,就会将报错信息粘贴到里面去,搜索有没有对应的答案  
4、GitHub (开源代码管理),搜索 上千亿行代码  
5、电商网站,检索商品  
6、日志数据分析, logstash采集日志, ES进行复杂的数据分析, **ELK技术, elasticsearch+logstash+kibana**  
7、商品价格监控网站,用户设定某商品的价格阈值,当低于该阈值的时候,发送通知消息给用户,比如说订阅牙膏的监控,如果高露洁牙膏的家庭套装低于50块钱,就通知我,我就去买  
8、BI系统,商业智能, Business Intelligence。比如说有个大型商场集团，BI ,分析一下某某区域最近3年的用户消费 金额的趋势以及用户群体的组成构成,产出相关的数张报表, \*\*区,最近3年,每年消费金额呈现100%的增长,而且用户群体85%是高级白领，开-个新商场。ES执行数据分析和挖掘, Kibana进行数据可视化  
9、国内:站内搜索(电商,招聘,门户,等等),IT系统搜索(OA,CRM,ERP,等等),数据分析(ES热门  
的一一个使用场景)

**ES和Solr**

**ElasticSearch简介**

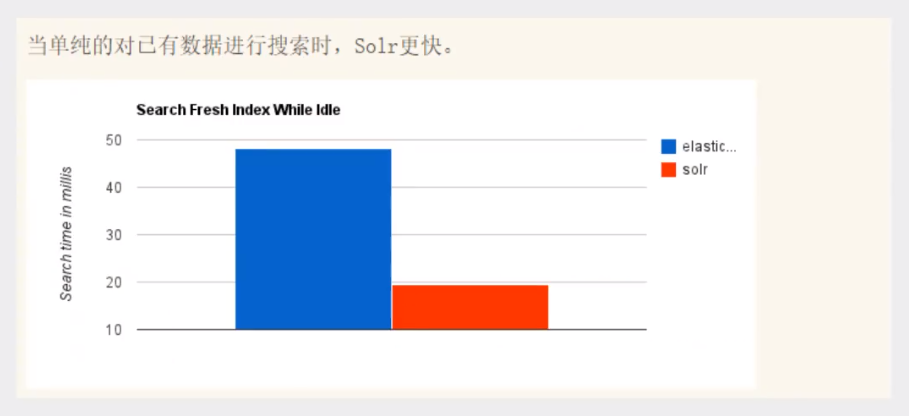
* Elasticsearch是一个**实时分布式搜索和分析引擎**。 它让你以前所未有的速度处理大数据成为可能。
* 它用于<mark>**全文搜索、结构化搜索、分析**</mark>以及将这三者混合使用:
* 维基百科使用Elasticsearch提供**全文搜索**并**高亮关键字**,以及输入**实时搜索**(search-asyou-type)和**搜索纠错**(did-you-mean)等搜索建议功能。
* 英国卫报使用Elasticsearch结合用户日志和社交网络数据提供给他们的编辑以实时的反馈,以便及时了解公众对新发表的文章的回应。
* StackOverflow结合全文搜索与地理位置查询,以及more-like-this功能来找到相关的问题和答案。
* Github使用Elasticsearch检索1300亿行的代码。
* 但是Elasticsearch不仅用于大型企业，它还让像DataDog以及Klout这样的创业公司将最初的想法变成可扩展的解决方案。
* Elasticsearch可以在你的笔记本上运行,也可以在数以百计的服务器上处理PB级别的数据。
* Elasticsearch是一个基于Apache Lucene(TM)的开源搜索引擎。无论在开源还是专有领域, Lucene可被认为是迄今为止最先进、性能最好的、功能最全的搜索引擎库。
  + 但是, **Lucene只是一个库**。 想要使用它,你必须使用Java来作为开发语言并将其直接集成到你的应用中,更糟糕的是, Lucene非常复杂,你需要深入了解检索的相关知识来理解它是如何工作的。
* Elasticsearch也使用Java开发并使用Lucene作为其核心来实现所有索引和搜索的功能,但是它的**目的**是<mark>通过简单的**RESTful API**来隐藏Lucene的复杂性,从而让全文搜索变得简单。</mark>

**Solr简介**

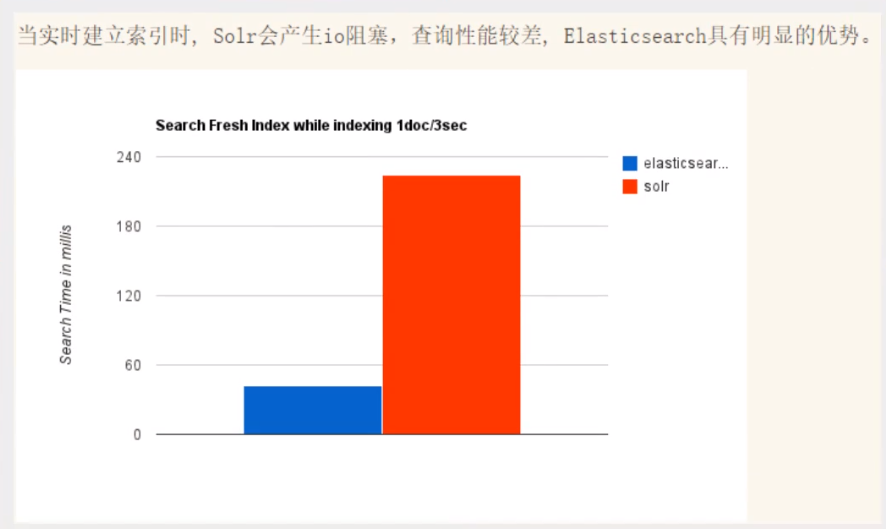
* Solr是Apache下的一个顶级开源项目,采用Java开发,它是**基于Lucene的全文搜索服务器**。Solr提供了比Lucene更为**丰富的查询语言**,同时实现了**可配置**、**可扩展**，并**对索引、搜索性能进行了优化**
* Solr可以**独立运行**,运行在letty. Tomcat等这些Selrvlet容器中 , Solr 索引的实现方法很简单,<mark>用POST方法向Solr服务器发送一个描述Field及其内容的XML文档, Solr根据xml文档**添加、删除、更新**索引</mark>。Solr 搜索只需要发送HTTP GET请求,然后对Solr返回xml、json等格式的查询结果进行解析,组织页面布局。
* Solr不提供构建UI的功能, **Solr提供了一个管理界面,通过管理界面可以查询Solr的配置和运行情况。**
* Solr是基于lucene开发企业级搜索服务器,实际上就是封装了lucene.
* Solr是一个独立的企业级搜索应用服务器,它**对外提供类似于Web-service的API接口**。用户可以通过http请求,向搜索引擎服务器提交-定格式的文件,生成索引;也可以通过提出查找请求,并得到返回结果。

**ElasticSearch与Solr比较**

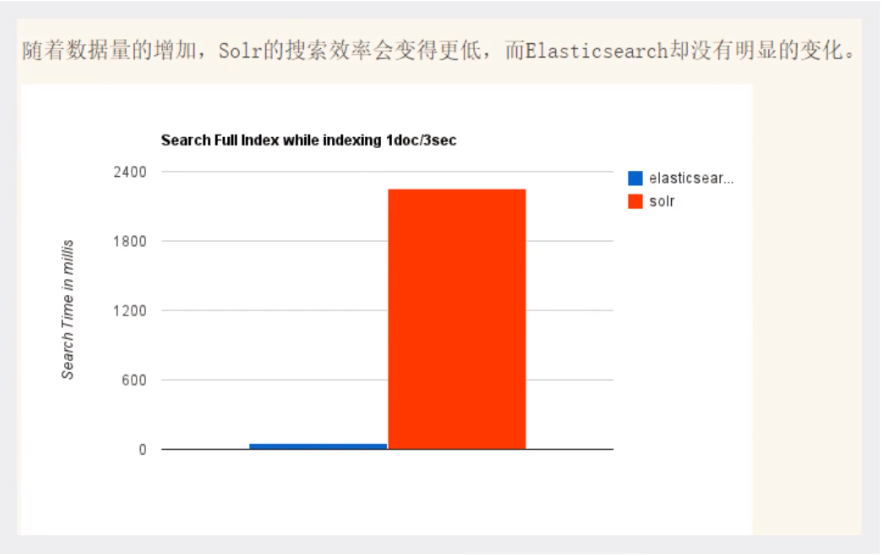
1. 当单纯的对已有数据进行搜索时，Solr更快



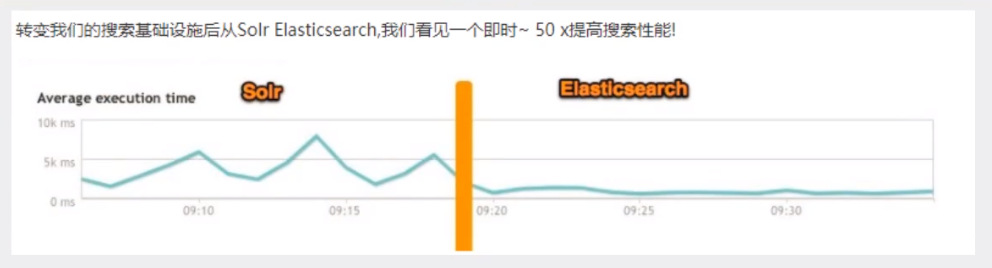
1. 当实时建立索引时，Solr会产生io阻塞，查询性能较差，ElasticSearch具有明显的优势



1. 随着数据量的增加，Solr的搜索效率会变得更低，而ElasticSearch却没有明显的变化



1. 转变我们的搜索基础设施后从Solr ElasticSearch，我们看见一个即时~ 50x提高搜索性能！



**总结**

1、**es**基本是**开箱即用**(解压就可以用!) ,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<mark>自身带有分布式协调管理功能</mark>。**  
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友好支撑  
<mark>5、</mark>**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。

6、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。

**二、ElasticSearch安装**

JDK8，最低要求

使用Java开发，必须保证ElasticSearch的版本与Java的核心jar包版本对应！（Java环境保证没错）

这里在windows上进行安装

**Windows下安装**

**1、安装**

下载地址：<https://www.elastic.co/cn/downloads/>

历史版本下载：<https://www.elastic.co/cn/downloads/past-releases/>

解压即可（尽量将ElasticSearch相关工具放在统一目录下）

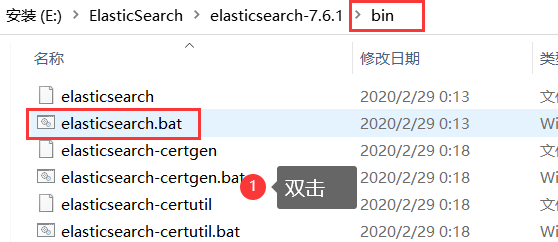
**2、熟悉目录**

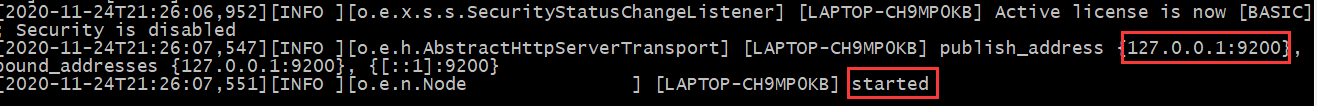


1. bin 启动文件目录
2. config 配置文件目录
3. 1og4j2 日志配置文件
4. jvm.options java 虚拟机相关的配置(默认启动占1g内存，内容不够需要自己调整)
5. elasticsearch.ym1 elasticsearch 的配置文件! 默认9200端口!跨域!
6. 1ib
7. 相关jar包
8. modules 功能模块目录
9. plugins 插件目录
10. ik分词器

**3、启动**

一定要检查自己的java环境是否配置好







**安装可视化界面**

elasticsearch-head

**使用前提**：需要安装nodejs

**1、下载地址**

<https://github.com/mobz/elasticsearch-head>

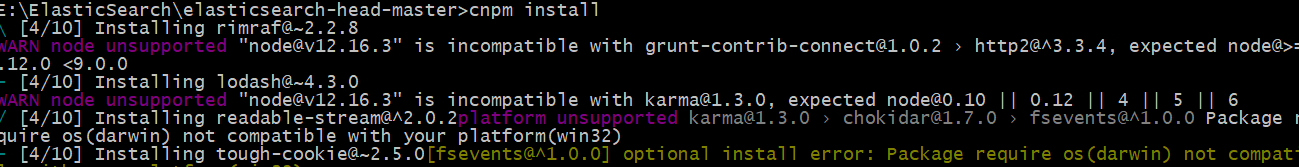
**2、安装**

解压即可（尽量将ElasticSearch相关工具放在统一目录下）

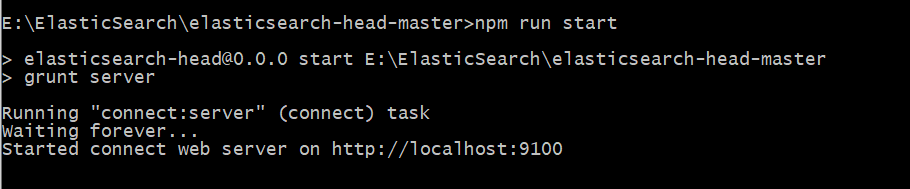
**3、启动**

1. cd elasticsearch-head
2. # 安装依赖
3. npm install
4. # 启动
5. npm run start
6. # 访问
7. http://localhost:9100/

**安装依赖**



**运行**



**访问**

存在跨域问题（只有当两个页面同源，才能交互）

同源（端口，主机，协议三者都相同）

<https://blog.csdn.net/qq_38128179/article/details/84956552>

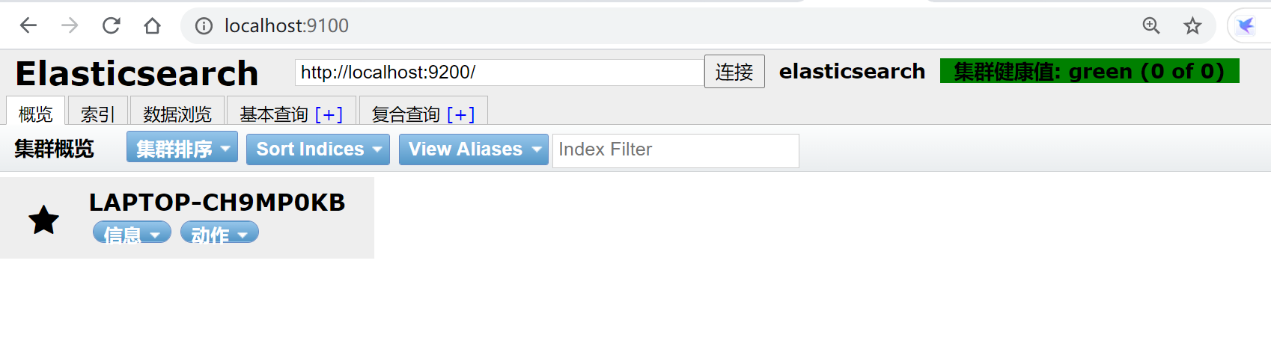


**开启跨域（在elasticsearch解压目录config下elasticsearch.yml中添加）**

1. # 开启跨域
2. http.cors.enabled: true
3. # 所有人访问
4. http.cors.allow-origin: "\*"

重启elasticsearch

**再次连接**



**如何理解上图：**

* 如果你是初学者
  + 索引 可以看做 “数据库”
  + 类型 可以看做 “表”
  + 文档 可以看做 “库中的数据（表中的行）”
* 这个head，我们只是把它**当做可视化数据展示工具**，之后**所有的查询都在kibana中进行**
  + 因为不支持json格式化，不方便

**安装kibana**

Kibana是一个针对ElasticSearch的开源分析及可视化平台,用来搜索、查看交互存储在Elasticsearch索引中的数据。使用Kibana ,可以通过各种图表进行高级数据分析及展示。Kibana让海量数据更容易理解。它操作简单,基于浏览器的用户界面可以快速创建仪表板( dashboard )实时显示Elasticsearch查询动态。设置Kibana非常简单。无需编码或者额外的基础架构,几分钟内就可以完成Kibana安装并启动Elasticsearch索引监测。

**1、下载地址:**

下载的版本需要与ElasticSearch版本对应

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

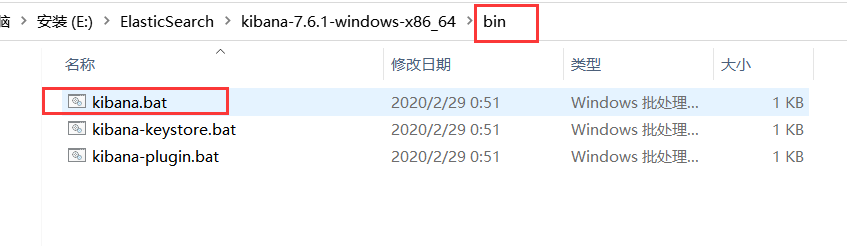
历史版本下载：<https://www.elastic.co/cn/downloads/past-releases/>

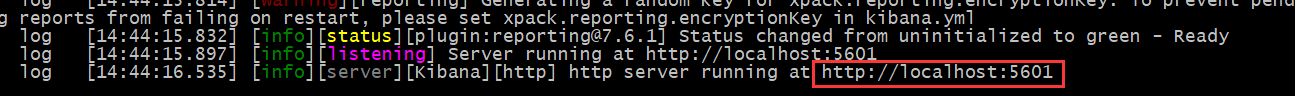
**2、安装**

解压即可（尽量将ElasticSearch相关工具放在统一目录下）



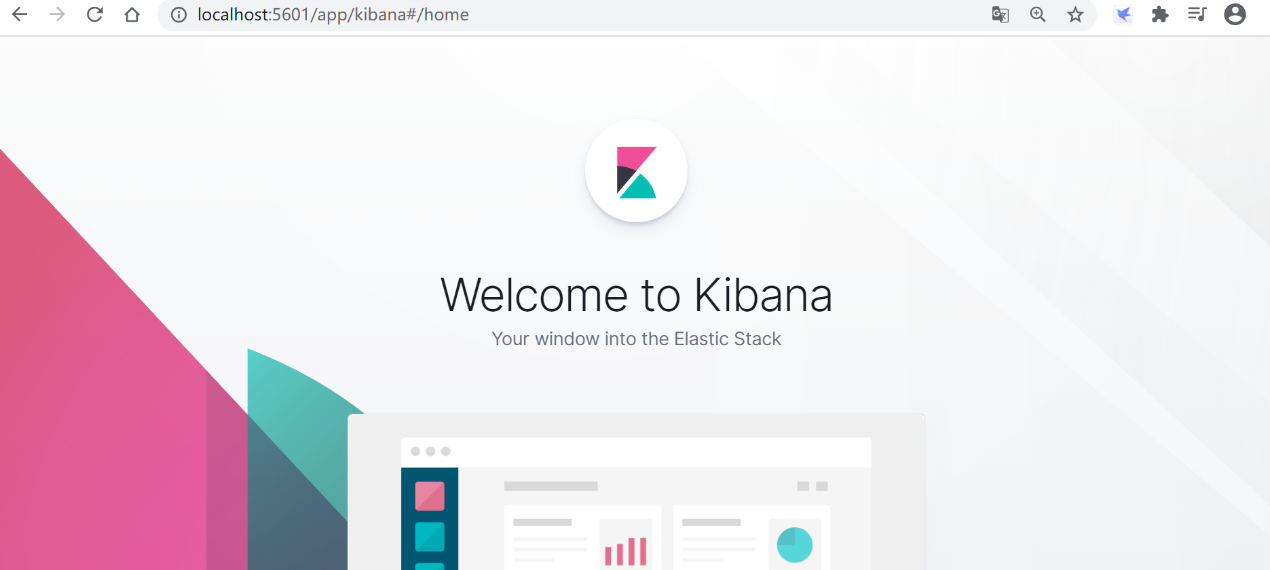
**3、启动**





**访问**

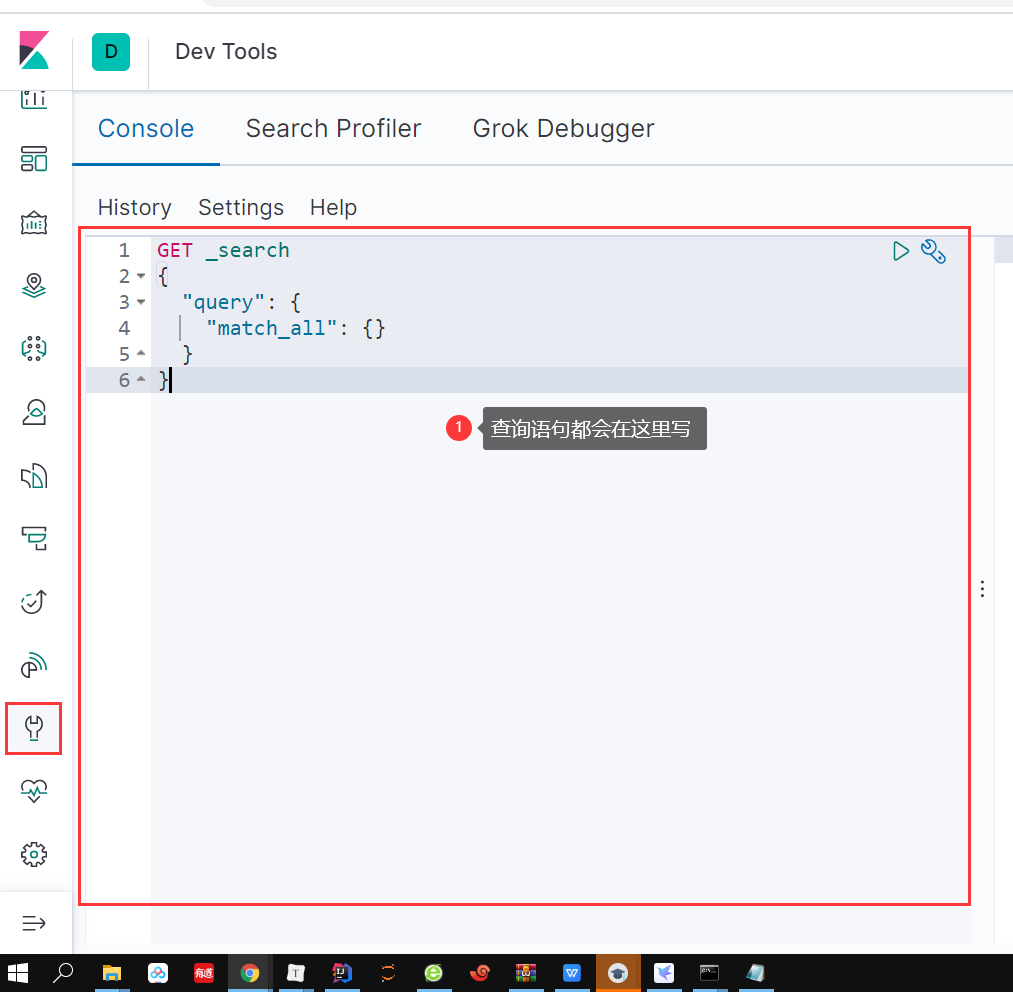
1. localhost:5601



**4、开发工具**

（Postman、curl、head、谷歌浏览器插件）

可以使用 Kibana进行测试



如果说，你在英文方面不太擅长，kibana是支持汉化的

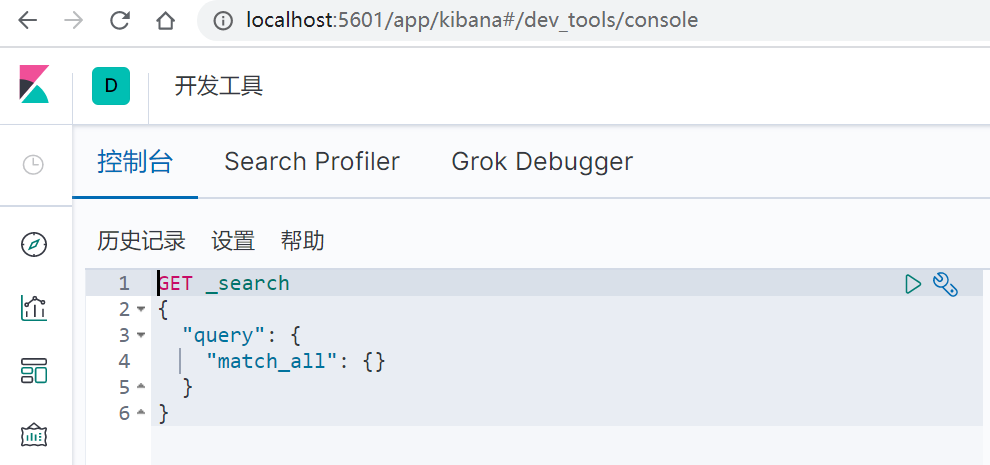
**5、kibana汉化**

编辑器打开kibana解压目录/config/kibana.yml，添加

1. i18n.locale: "zh-CN"

重启kibana

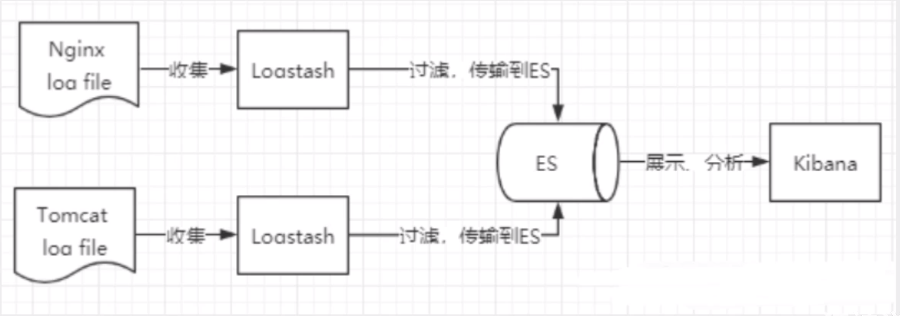
**汉化成功**



**了解ELK**

* ELK是**Elasticsearch、Logstash、 Kibana三大开源框架首字母大写简称**。市面上也被成为Elastic Stack。
  + 其中Elasticsearch是一个基于Lucene、分布式、通过Restful方式进行交互的近实时搜索平台框架。
    - 像类似百度、谷歌这种大数据全文搜索引擎的场景都可以使用Elasticsearch作为底层支持框架，可见Elasticsearch提供的搜索能力确实强大,市面上很多时候我们简称Elasticsearch为es。
  + Logstash是ELK的中央数据流引擎,用于从不同目标(文件/数据存储/MQ )收集的不同格式数据,经过过滤后支持输出到不同目的地(文件/MQ/redis/elasticsearch/kafka等)。
  + Kibana可以将elasticsearch的数据通过友好的页面展示出来 ,提供实时分析的功能。
* 市面上很多开发只要提到ELK能够一致说出它是一个日志分析架构技术栈总称 ,但实际上ELK不仅仅适用于日志分析,它还可以支持其它任何数据分析和收集的场景,日志分析和收集只是更具有代表性。并非唯一性。

1. 收集清洗数据(Logstash) ==> 搜索、存储(ElasticSearch) ==> 展示(Kibana)



**三、ElasticSearch核心概念**

**概述**

1、索引（ElasticSearch）

* 包多个分片

2、字段类型（映射）

* 字段类型映射（字段是整型，还是字符型…）

3、文档

4、分片（Lucene索引，倒排索引）

ElasticSearch是面向文档，关系行数据库和ElasticSearch客观对比！一切都是JSON！

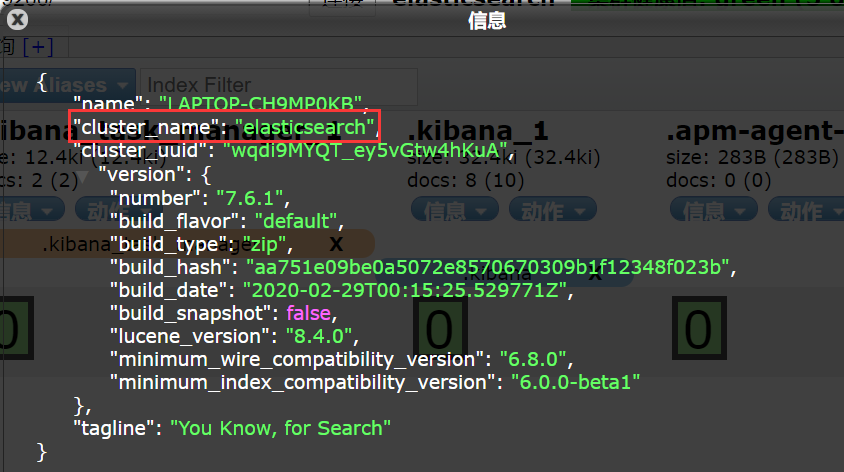
| **Relational DB** | **ElasticSearch** |
| --- | --- |
| 数据库（database） | 索引（indices） |
| 表（tables） | types \<慢慢会被弃用!> |
| 行（rows） | documents |
| 字段（columns） | fields |

**elasticsearch（集群）**中可以包含多个**索引（数据库）** ,每个索引中可以包含多个**类型（表）** ,每个类型下又包含多个**文档（行）** ,每个文档中又包含多个**字段（列）**。

**物理设计:**

elasticsearch在后台把**每个索引划分成多个分片**，每分分片可以在集群中的不同服务器间迁移

一个人就是一个集群! ，即**启动的ElasticSearch服务，默认就是一个集群，且默认集群名为elasticsearch**



**逻辑设计:**

一个索引类型中，包含多个文档，比如说文档1，文档2。当我们索引一篇文档时，可以通过这样的顺序找到它：索引 => 类型 => 文档ID ，通过这个组合我们就能索引到某个具体的文档。 注意：ID不必是整数，实际上它是个字符串。

**文档（”行“）**

之前说elasticsearch是面向文档的，那么就意味着**索引和搜索数据的最小单位是文档**，elasticsearch中，文档有几个重要属性:

* 自我包含，一篇文档同时包含字段和对应的值，也就是同时包含key:value !
* 可以是层次型的，一个文档中包含自文档，复杂的逻辑实体就是这么来的! {就是一个json对象 ! fastjson进行自动转换 !}
* 灵活的结构，文档不依赖预先定义的模式，我们知道关系型数据库中，要提前定义字段才能使用，在elasticsearch中，对于字段是非常灵活的，有时候,我们可以忽略该字段，或者动态的添加一个新的字段。

尽管我们可以随意的新增或者忽略某个字段，但是，每个字段的类型非常重要，比如一个年龄字段类型，可以是字符串也可以是整形。因为elasticsearch会保存字段和类型之间的映射及其他的设置。这种映射具体到每个映射的每种类型，这也是为什么在elasticsearch中，类型有时候也称为映射类型。

**类型（“表”）**

类型是文档的逻辑容器，就像关系型数据库一样，表格是行的容器。类型中对于字段的定义称为映射，比如name映射为字符串类型。我们说文档是无模式的，它们不需要拥有映射中所定义的所有字段，比如新增一个字段，那么elasticsearch是怎么做的呢?

* elasticsearch会自动的将新字段加入映射，但是这个字段的不确定它是什么类型，elasticsearch就开始猜，如果这个值是18，那么elasticsearch会认为它是整形。但是elasticsearch也可能猜不对，所以最安全的方式就是提前定义好所需要的映射，这点跟关系型数据库殊途同归了，先定义好字段，然后再使用，别整什么幺蛾子。

**索引（“库”）**

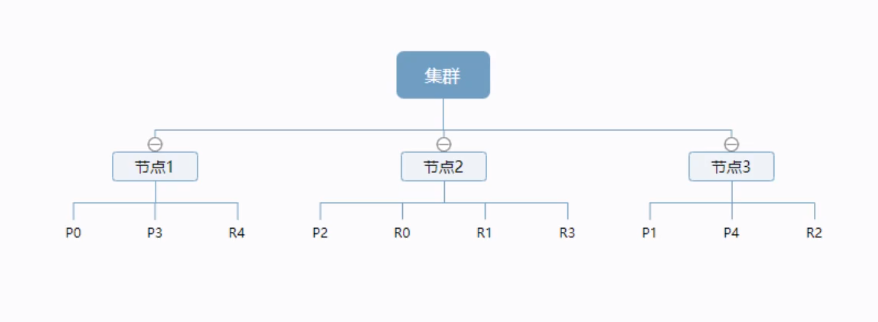
索引是映射类型的容器， elasticsearch中的索引是一个非常大的文档集合。 索引存储了映射类型的字段和其他设置。然后它们被存储到了各个分片上了。我们来研究下分片是如何工作的。

**物理设计：节点和分片 如何工作**

创建新索引



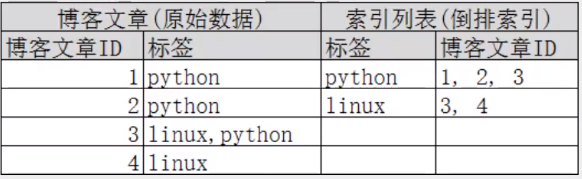
一个集群至少有一个节点，而一个节点就是一个elasricsearch进程，节点可以有多个索引默认的，如果你创建索引，那么索引将会有个5个分片(primary shard ,又称主分片)构成的，每一个主分片会有一个副本(replica shard，又称复制分片)



上图是一个有3个节点的集群，可以看到主分片和对应的复制分片都不会在同一个节点内，这样有利于某个节点挂掉了，数据也不至于失。实际上，**一个分片是一个Lucene索引（<mark>一个ElasticSearch索引包含多个Lucene索引</mark>）** ，**一个包含倒排索引的文件目录，倒排索引的结构使得elasticsearch在不扫描全部文档的情况下，就能告诉你哪些文档包含特定的关键字**。不过，等等，倒排索引是什么鬼?

**倒排索引（Lucene索引底层）**

简单说就是 按（文章关键字，对应的文档\<0个或多个\>）形式建立索引，根据关键字就可直接查询对应的文档（含关键字的），无需查询每一个文档，如下图



**四、IK分词器(elasticsearch插件)**

**IK分词器：中文分词器**

分词：即把一段中文或者别的划分成一个个的关键字，我们在搜索时候会把自己的信息进行分词，会把数据库中或者索引库中的数据进行分词，然后进行一一个匹配操作，**默认的中文分词是将每个字看成一个词**（<mark>不使用用IK分词器的情况下</mark>），比如“我爱狂神”会被分为”我”，”爱”，”狂”，”神” ，这显然是不符合要求的，所以我们需要安装中文分词器ik来解决这个问题。

**IK提供了两个分词算法**: ik\_smart和ik\_max\_word ,其中ik\_smart为**最少切分**, ik\_max\_word为**最细粒度划分**!

**1、下载**

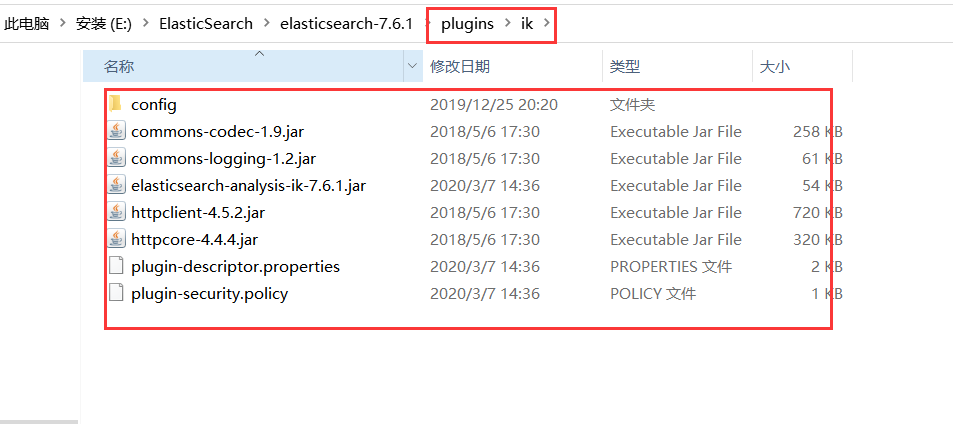
版本要与ElasticSearch版本对应

下载地址：<https://github.com/medcl/elasticsearch-analysis-ik/releases>

**2、安装**

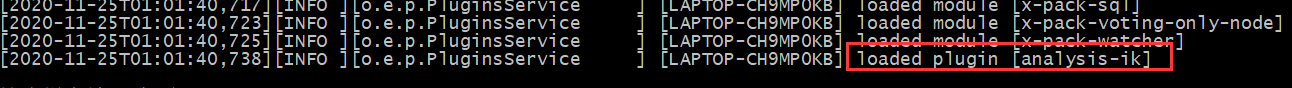
ik文件夹是自己创建的

加压即可（但是我们需要解压到ElasticSearch的plugins目录ik文件夹下）



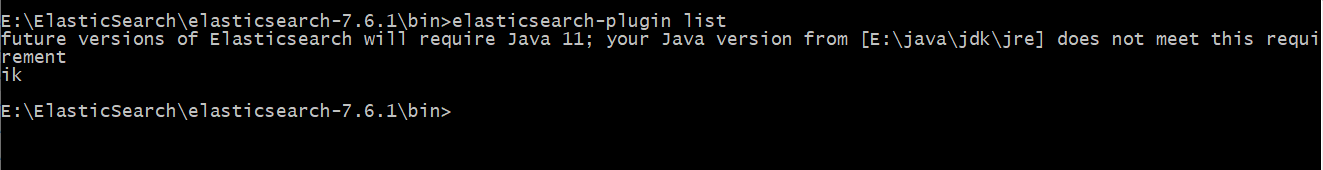
**3、重启ElasticSearch**

加载了IK分词器



**4、使用 ElasticSearch安装补录/bin/elasticsearch-plugin 可以查看插件**

1. E:\ElasticSearch\elasticsearch-7.6.1\bin>elasticsearch-plugin list

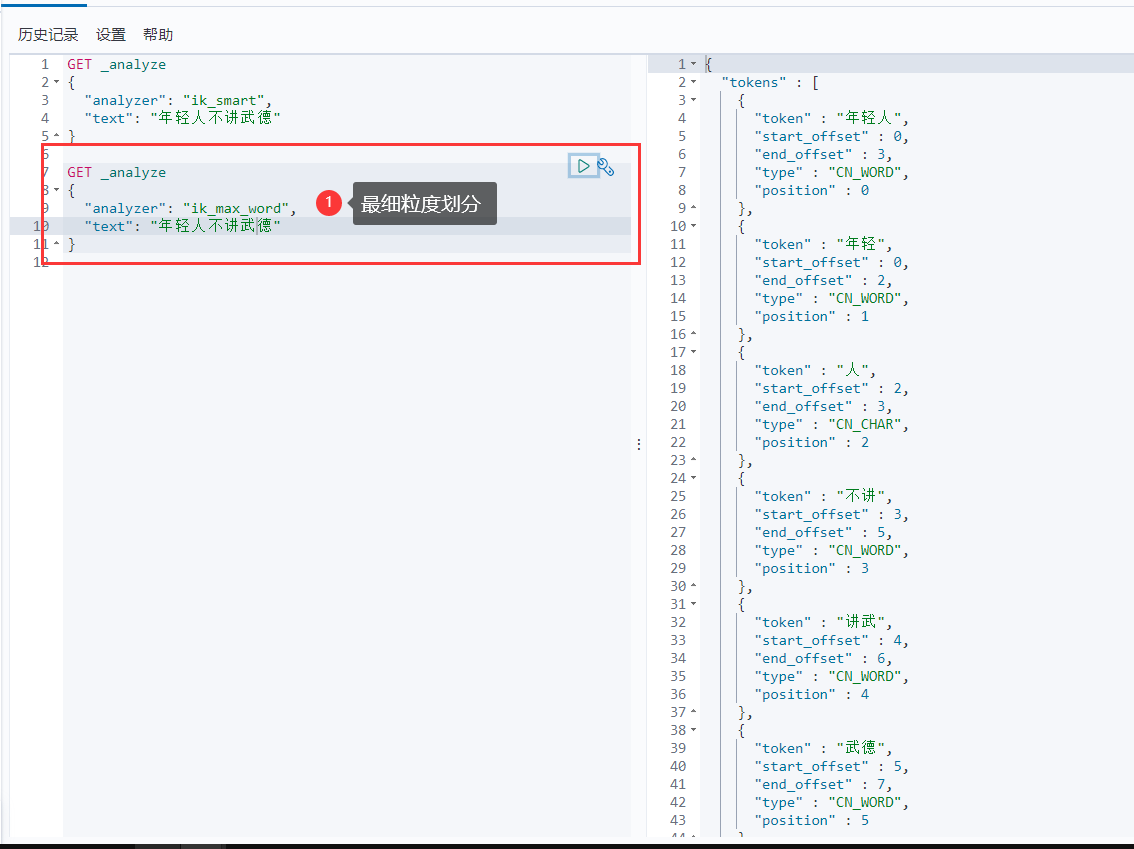


**5、使用kibana测试**

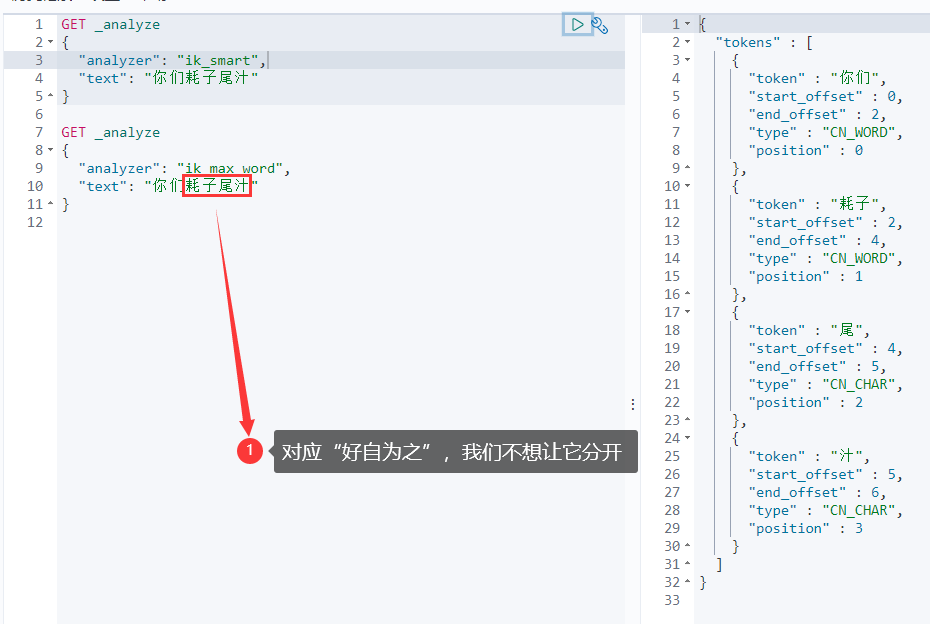
ik\_smart：最少切分



ik\_max\_word：最细粒度划分（穷尽词库的可能）



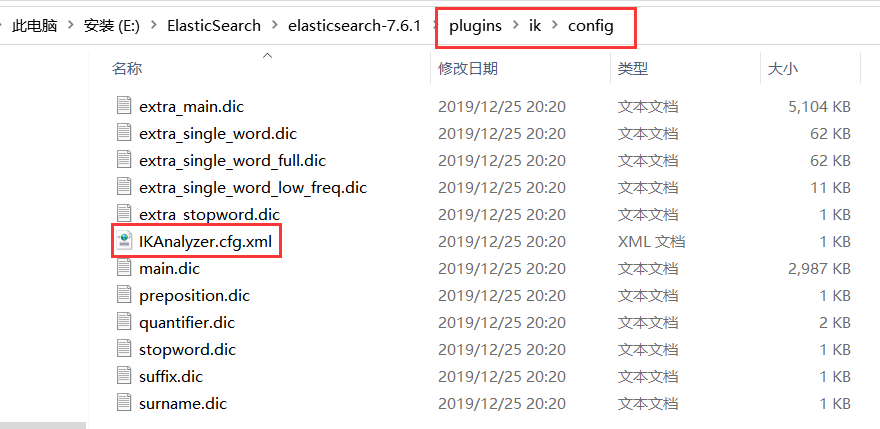
从上面看，感觉分词都比较正常，但是大多数，分词都满足不了我们的想法，如下例



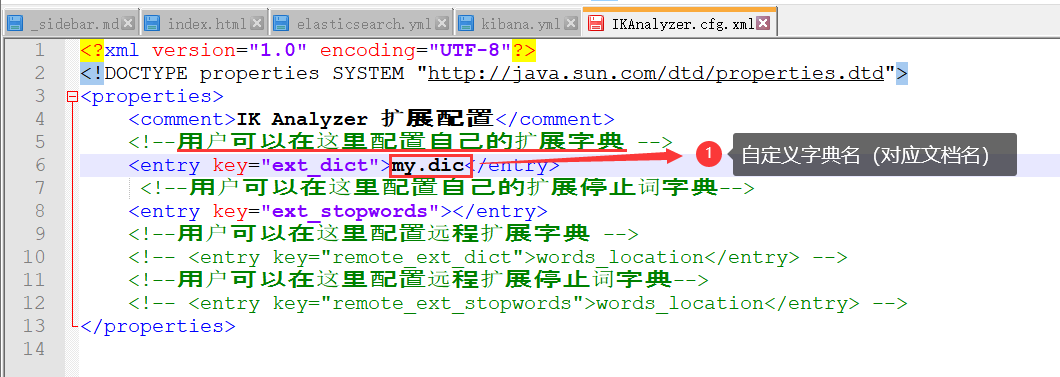
那么，我们需要手动将该词添加到分词器的词典当中

**6、添加自定义的词添加到扩展字典中**

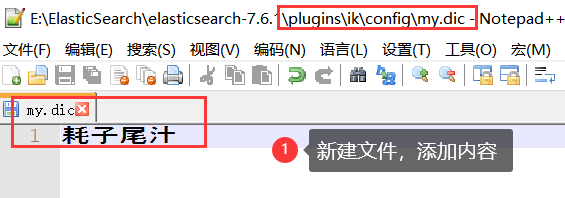
elasticsearch目录/plugins/ik/config/IKAnalyzer.cfg.xml



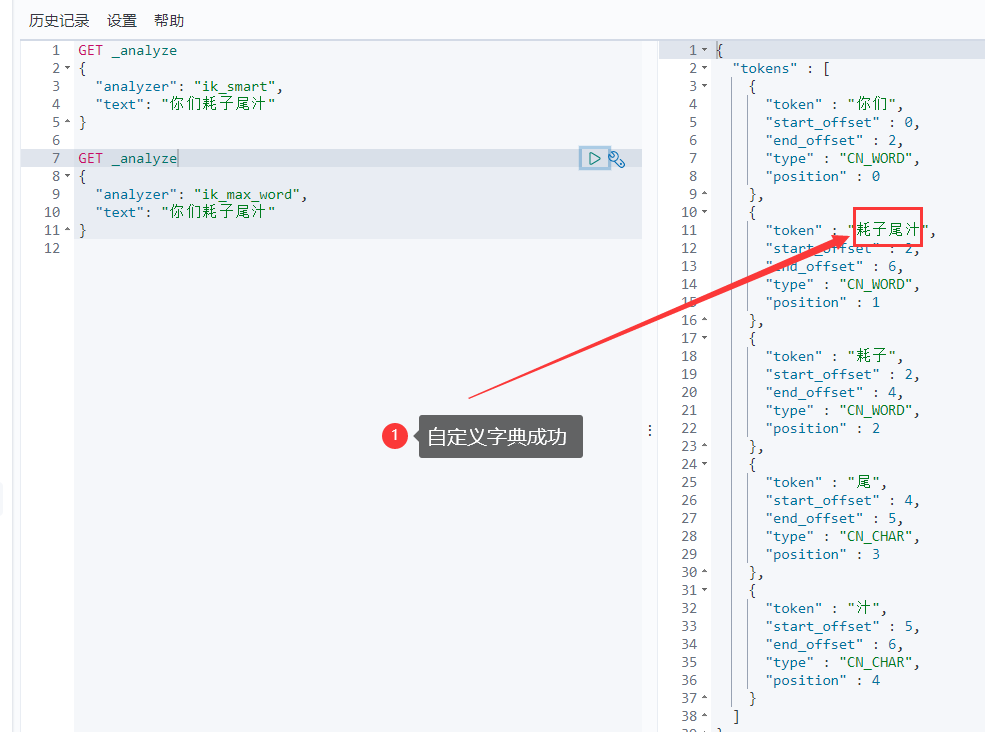
打开 IKAnalyzer.cfg.xml 文件，扩展字典



创建字典文件，添加字典内容



重启ElasticSearch，再次使用kibana测试



**五、Rest风格说明**

**一种软件架构风格**,而不是标准,只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以**更简洁**，**更有层次**，**更易于实现缓存**等机制。

**基本Rest命令说明：**

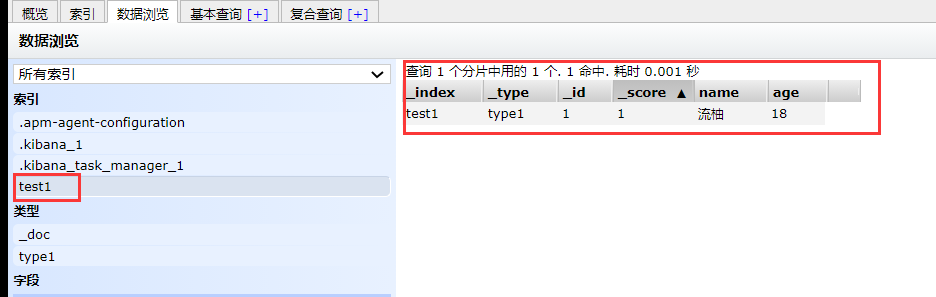
| **method** | **url地址** | **描述** |
| --- | --- | --- |
| PUT（创建,修改） | localhost:9200/索引名称/类型名称/文档id | 创建文档（指定文档id） |
| POST（创建） | localhost:9200/索引名称/类型名称 | 创建文档（随机文档id） |
| POST（修改） | localhost:9200/索引名称/类型名称/文档id/\_update | 修改文档 |
| DELETE（删除） | localhost:9200/索引名称/类型名称/文档id | 删除文档 |
| GET（查询） | localhost:9200/索引名称/类型名称/文档id | 查询文档通过文档ID |
| POST（查询） | localhost:9200/索引名称/类型名称/文档id/\_search | 查询所有数据 |

**测试**

**1、创建一个索引，添加**

1. PUT /test1/type1/1
2. {
3. "name" : "流柚",
4. "age" : 18
5. }





**2、字段数据类型**

* 字符串类型
  + text、**keyword**
    - text：支持分词，全文检索,支持模糊、精确查询,不支持聚合,排序操作;text类型的最大支持的字符长度无限制,适合大字段存储；
    - keyword：不进行分词，直接索引、支持模糊、支持精确匹配，支持聚合、排序操作。keyword类型的最大支持的长度为——32766个UTF-8类型的字符,可以通过设置ignore\_above指定自持字符长度，超过给定长度后的数据将不被索引，无法通过term精确匹配检索返回结果。
* 数值型
  + long、Integer、short、byte、double、float、**half float**、**scaled float**
* 日期类型
  + date
* te布尔类型
  + boolean
* 二进制类型
  + binary
* 等等…

**3、指定字段的类型（使用PUT）**

类似于建库（建立索引和字段对应类型），也可看做规则的建立

1. PUT /test2
2. {
3. "mappings": {
4. "properties": {
5. "name": {
6. "type": "text"
7. },
8. "age":{
9. "type": "long"
10. },
11. "birthday":{
12. "type": "date"
13. }
14. }
15. }
16. }



**4、获取3建立的规则**

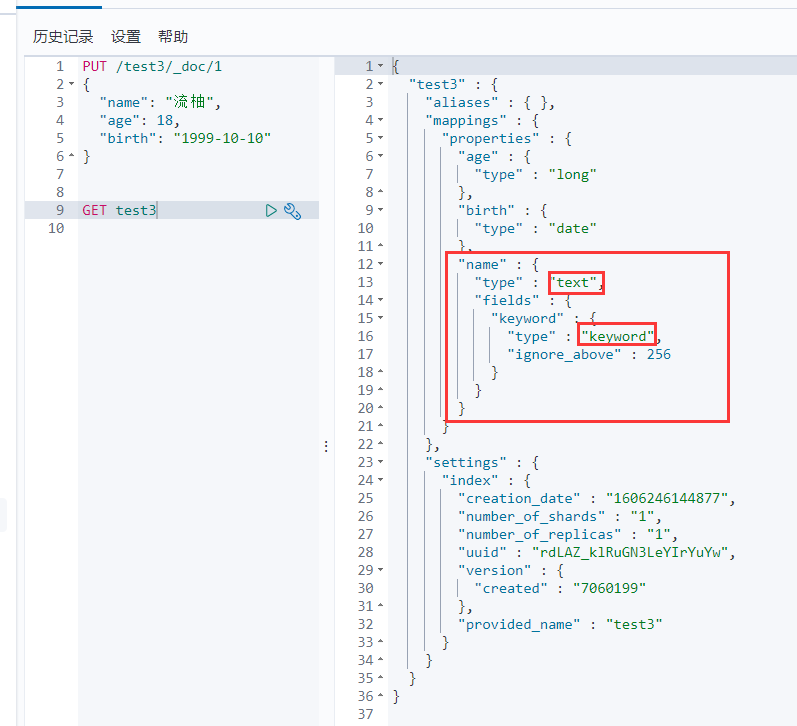
1. GET test2



**5、获取默认信息**

\_doc 默认类型（default type），type 在未来的版本中会逐渐弃用，因此产生一个默认类型进行代替

1. PUT /test3/\_doc/1
2. {
3. "name": "流柚",
4. "age": 18,
5. "birth": "1999-10-10"
6. }
7. GET test3



如果自己的文档字段没有被指定，那么ElasticSearch就会给我们默认配置字段类型

扩展：通过get \_cat/ 可以获取ElasticSearch的当前的很多信息！

1. GET \_cat/indices
2. GET \_cat/aliases
3. GET \_cat/allocation
4. GET \_cat/count
5. GET \_cat/fielddata
6. GET \_cat/health
7. GET \_cat/indices
8. GET \_cat/master
9. GET \_cat/nodeattrs
10. GET \_cat/nodes
11. GET \_cat/pending\_tasks
12. GET \_cat/plugins
13. GET \_cat/recovery
14. GET \_cat/repositories
15. GET \_cat/segments
16. GET \_cat/shards
17. GET \_cat/snapshots
18. GET \_cat/tasks
19. GET \_cat/templates
20. GET \_cat/thread\_pool

**6、修改**

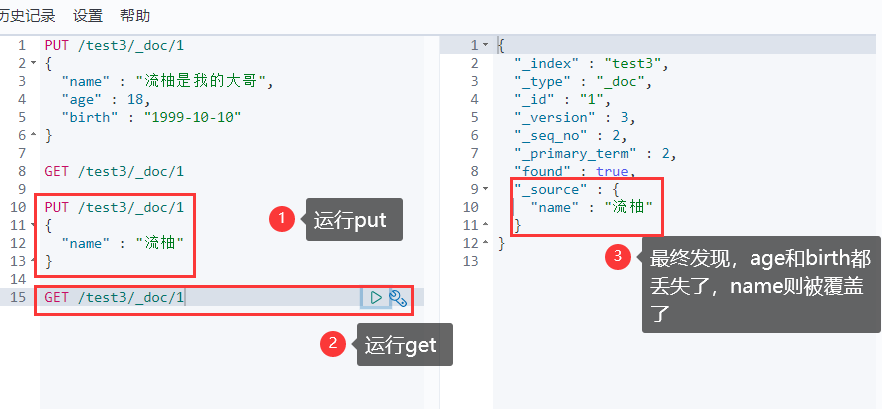
两种方案

①旧的（使用put覆盖原来的值）

* 版本+1（\_version）
* 但是如果漏掉某个字段没有写，那么更新是没有写的字段 ，会消失

1. PUT /test3/\_doc/1
2. {
3. "name" : "流柚是我的大哥",
4. "age" : 18,
5. "birth" : "1999-10-10"
6. }
7. GET /test3/\_doc/1
8. // 修改会有字段丢失
9. PUT /test3/\_doc/1
10. {
11. "name" : "流柚"
12. }
13. GET /test3/\_doc/1





②新的（使用post的update）

* version不会改变
* 需要注意doc
* 不会丢失字段

1. POST /test3/\_doc/1/\_update
2. {
3. "doc":{
4. "name" : "post修改，version不会加一",
5. "age" : 2
6. }
7. }
8. GET /test3/\_doc/1



**7、删除**

1. GET /test1
2. DELETE /test1





**8、查询（简单条件）**

1. GET /test3/\_doc/\_search?q=name:流柚



**9、复杂查询**

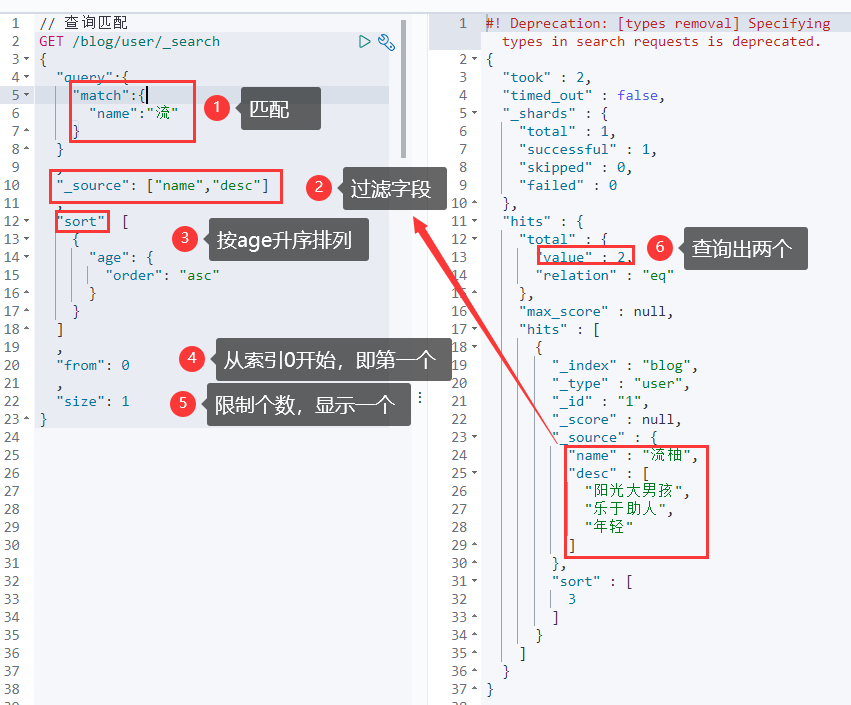
test3索引中的内容



**①查询匹配**

* match：匹配（会使用分词器解析（先分析文档，然后进行查询））
* \_source：过滤字段
* sort：排序
* form、size 分页

1. // 查询匹配
2. GET /blog/user/\_search
3. {
4. "query":{
5. "match":{
6. "name":"流"
7. }
8. }
9. ,
10. "\_source": ["name","desc"]
11. ,
12. "sort": [
13. {
14. "age": {
15. "order": "asc"
16. }
17. }
18. ]
19. ,
20. "from": 0
21. ,
22. "size": 1
23. }



**②多条件查询（bool）**

* must 相当于 and
* should 相当于 or
* must\_not 相当于 not (... and ...)
* filter 过滤

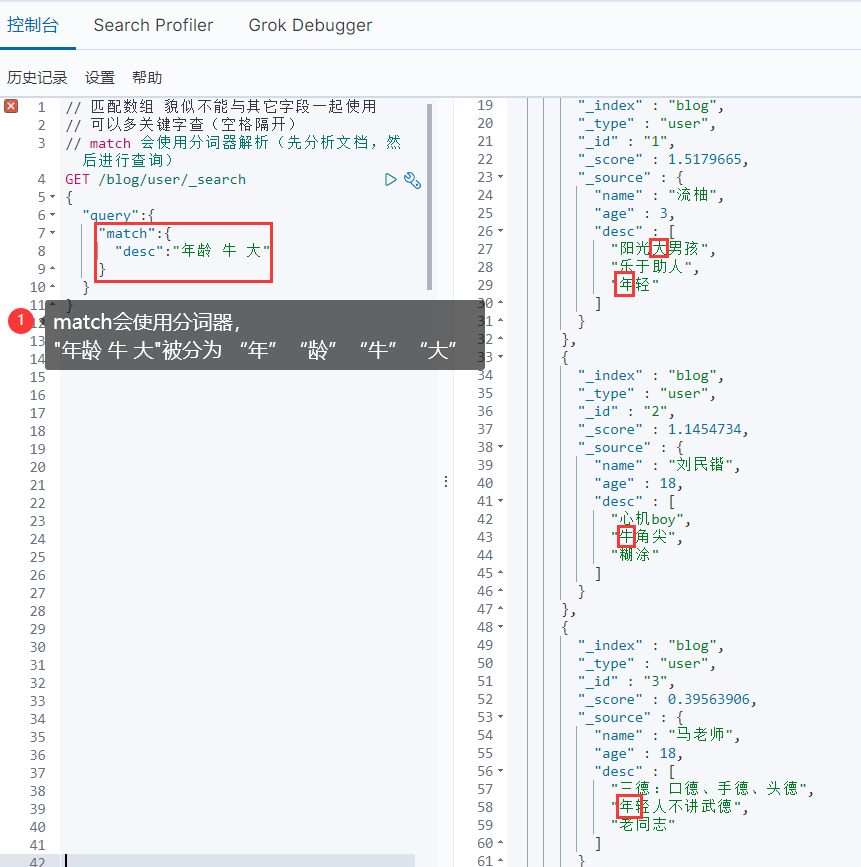
1. /// bool 多条件查询
2. //// must <==> and
3. //// should <==> or
4. //// must\_not <==> not (... and ...)
5. //// filter数据过滤
6. //// boost
7. //// minimum\_should\_match
8. GET /blog/user/\_search
9. {
10. "query":{
11. "bool": {
12. "must": [
13. {
14. "match":{
15. "age":3
16. }
17. },
18. {
19. "match": {
20. "name": "流"
21. }
22. }
23. ],
24. "filter": {
25. "range": {
26. "age": {
27. "gte": 1,
28. "lte": 3
29. }
30. }
31. }
32. }
33. }
34. }



**③匹配数组**

* 貌似不能与其它字段一起使用
* 可以多关键字查（空格隔开）— 匹配字段也是符合的
* match 会使用分词器解析（先分析文档，然后进行查询）
* 搜词

1. // 匹配数组 貌似不能与其它字段一起使用
2. // 可以多关键字查（空格隔开）
3. // match 会使用分词器解析（先分析文档，然后进行查询）
4. GET /blog/user/\_search
5. {
6. "query":{
7. "match":{
8. "desc":"年龄 牛 大"
9. }
10. }
11. }



**④精确查询**

* term 直接通过 倒排索引 指定**词条**查询
* 适合查询 number、date、keyword ，不适合text

1. // 精确查询（必须全部都有，而且不可分，即按一个完整的词查询）
2. // term 直接通过 倒排索引 指定的词条 进行精确查找的
3. GET /blog/user/\_search
4. {
5. "query":{
6. "term":{
7. "desc":"年 "
8. }
9. }
10. }



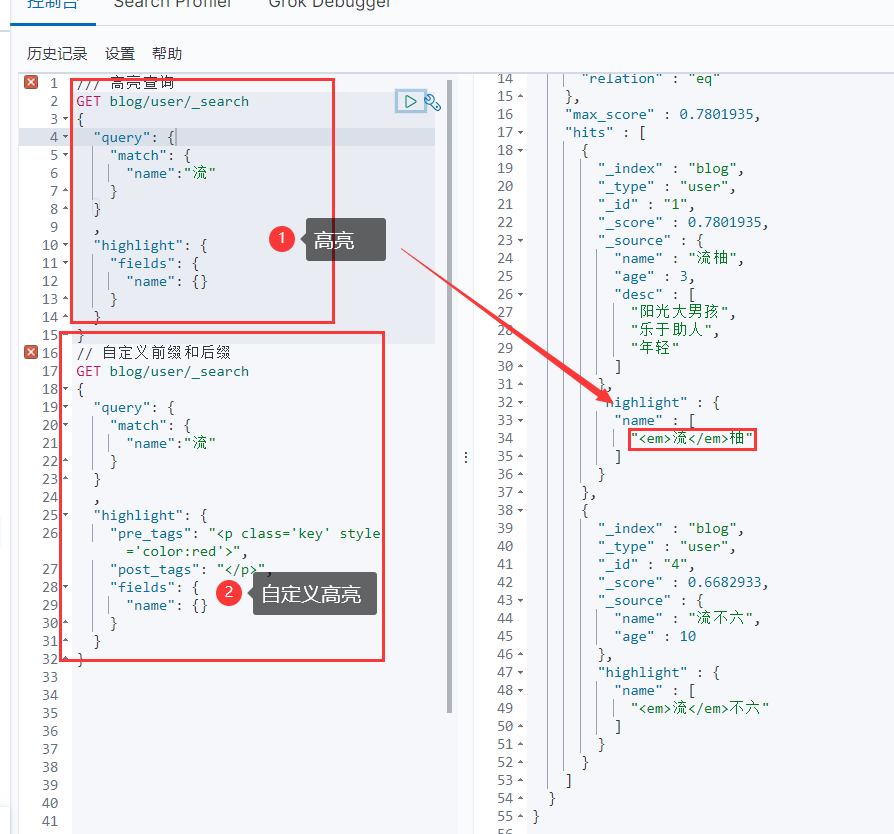
**⑤text和keyword**

* text：
  + **支持分词**，**全文检索**、支持模糊、精确查询,不支持聚合,排序操作;
  + text类型的最大支持的字符长度无限制,适合大字段存储；
* keyword：
  + **不进行分词**，**直接索引**、支持模糊、支持精确匹配，支持聚合、排序操作。
  + keyword类型的最大支持的长度为——32766个UTF-8类型的字符,可以通过设置ignore\_above指定自持字符长度，超过给定长度后的数据将不被索引，**无法通过term精确匹配检索返回结果**。

1. // 测试keyword和text是否支持分词
2. // 设置索引类型
3. PUT /test
4. {
5. "mappings": {
6. "properties": {
7. "text":{
8. "type":"text"
9. },
10. "keyword":{
11. "type":"keyword"
12. }
13. }
14. }
15. }
16. // 设置字段数据
17. PUT /test/\_doc/1
18. {
19. "text":"测试keyword和text是否支持分词",
20. "keyword":"测试keyword和text是否支持分词"
21. }
22. // text 支持分词
23. // keyword 不支持分词
24. GET /test/\_doc/\_search
25. {
26. "query":{
27. "match":{
28. "text":"测试"
29. }
30. }
31. }// 查的到
32. GET /test/\_doc/\_search
33. {
34. "query":{
35. "match":{
36. "keyword":"测试"
37. }
38. }
39. }// 查不到，必须是 "测试keyword和text是否支持分词" 才能查到
40. GET \_analyze
41. {
42. "analyzer": "keyword",
43. "text": ["测试liu"]
44. }// 不会分词，即 测试liu
45. GET \_analyze
46. {
47. "analyzer": "standard",
48. "text": ["测试liu"]
49. }// 分为 测 试 liu
50. GET \_analyze
51. {
52. "analyzer":"ik\_max\_word",
53. "text": ["测试liu"]
54. }// 分为 测试 liu

**⑥高亮查询**

1. /// 高亮查询
2. GET blog/user/\_search
3. {
4. "query": {
5. "match": {
6. "name":"流"
7. }
8. }
9. ,
10. "highlight": {
11. "fields": {
12. "name": {}
13. }
14. }
15. }
16. // 自定义前缀和后缀
17. GET blog/user/\_search
18. {
19. "query": {
20. "match": {
21. "name":"流"
22. }
23. }
24. ,
25. "highlight": {
26. "pre\_tags": "<p class='key' style='color:red'>",
27. "post\_tags": "</p>",
28. "fields": {
29. "name": {}
30. }
31. }
32. }

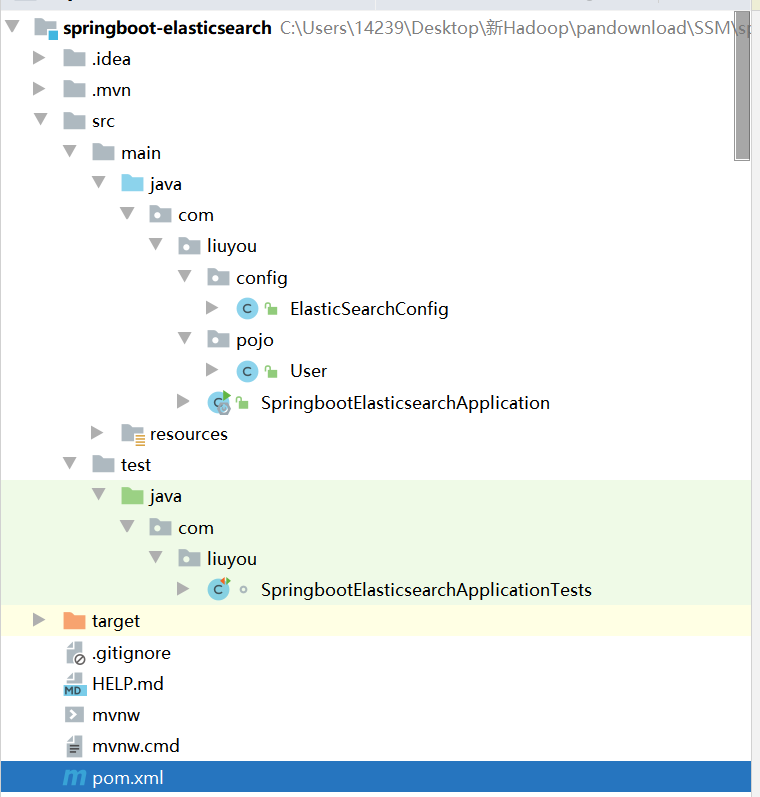


**六、SpringBoot整合**

**1、创建工程**

略

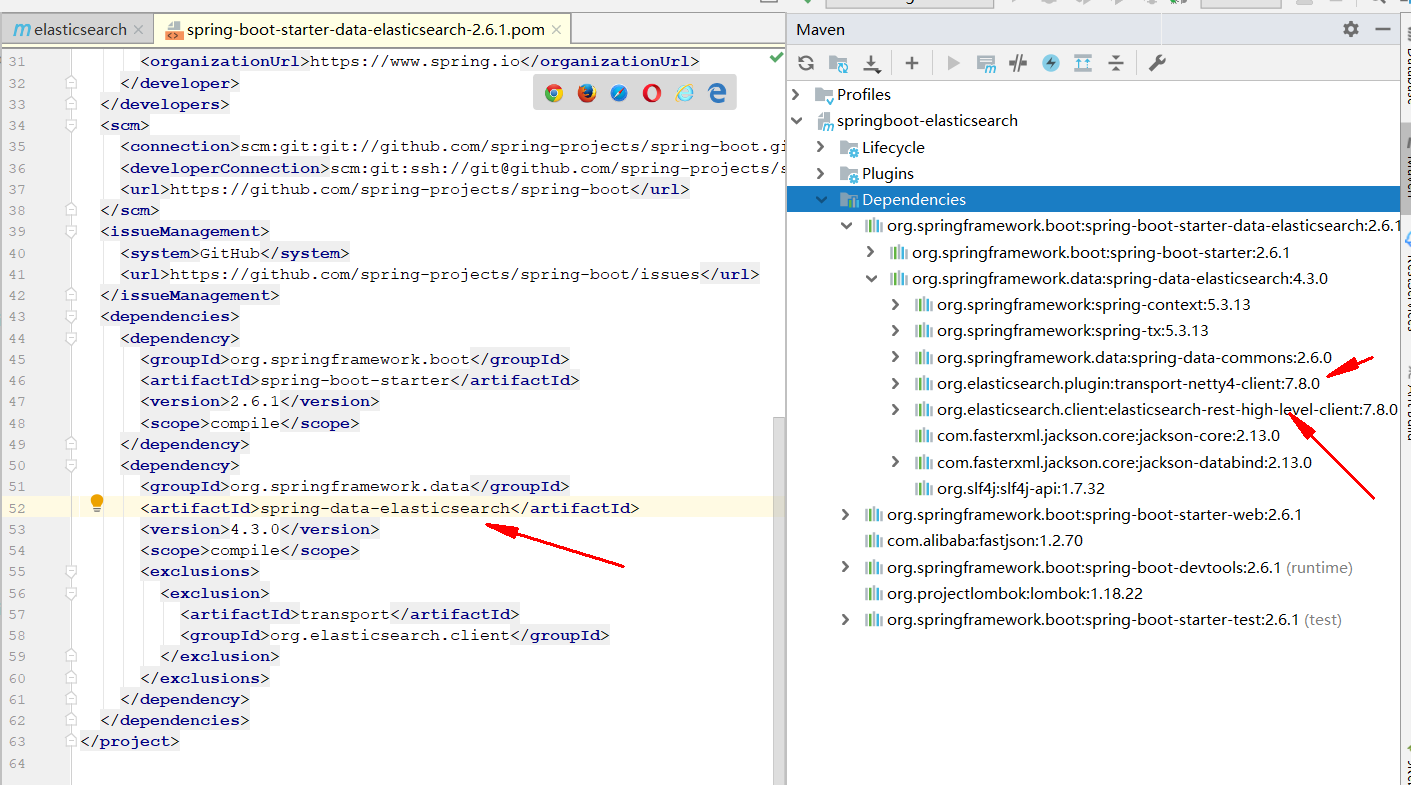
**目录结构**



**2、导入依赖**

注意依赖版本和安装的版本一致

1. <properties>
2. <java.version>1.8</java.version>
3. <!-- 统一版本 -->
4. <elasticsearch.version>7.6.1</elasticsearch.version>
5. </properties>



**导入elasticsearch**

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
4. </dependency>

**提前导入fastjson、lombok**

1. <dependency>
2. <groupId>com.alibaba</groupId>
3. <artifactId>fastjson</artifactId>
4. <version>1.2.70</version>
5. </dependency>
6. <!-- lombok需要安装插件 -->
7. <dependency>
8. <groupId>org.projectlombok</groupId>
9. <artifactId>lombok</artifactId>
10. <optional>true</optional>
11. </dependency>

**3、创建并编写配置类**

1. @Configuration
2. public class ElasticSearchConfig {
3. // 注册 rest高级客户端
4. @Bean
5. public RestHighLevelClient restHighLevelClient(){
6. RestHighLevelClient client = new RestHighLevelClient(
7. RestClient.builder(
8. new HttpHost("127.0.0.1",9200,"http")
9. )
10. );
11. return client;
12. }
13. }

**4、创建并编写实体类**

1. @Data
2. @NoArgsConstructor
3. @AllArgsConstructor
4. public class User implements Serializable {
5. private static final long serialVersionUID = -3843548915035470817L;
6. private String name;
7. private Integer age;
8. }

**5、测试**

所有测试均在 SpringbootElasticsearchApplicationTests中编写

**注入 RestHighLevelClient**

1. @Autowired
2. public RestHighLevelClient restHighLevelClient;

**索引的操作**

**1、索引的创建**

1. // 测试索引的创建， Request PUT liuyou\_index
2. @Test
3. public void testCreateIndex() throws IOException {
4. CreateIndexRequest request = new CreateIndexRequest("liuyou\_index");
5. CreateIndexResponse response = restHighLevelClient.indices().create(request, RequestOptions.DEFAULT);
6. System.out.println(response.isAcknowledged());// 查看是否创建成功
7. System.out.println(response);// 查看返回对象
8. restHighLevelClient.close();
9. }

**2、索引的获取，并判断其是否存在**

1. // 测试获取索引，并判断其是否存在
2. @Test
3. public void testIndexIsExists() throws IOException {
4. GetIndexRequest request = new GetIndexRequest("index");
5. boolean exists = restHighLevelClient.indices().exists(request, RequestOptions.DEFAULT);
6. System.out.println(exists);// 索引是否存在
7. restHighLevelClient.close();
8. }

**3、索引的删除**

1. // 测试索引删除
2. @Test
3. public void testDeleteIndex() throws IOException {
4. DeleteIndexRequest request = new DeleteIndexRequest("liuyou\_index");
5. AcknowledgedResponse response = restHighLevelClient.indices().delete(request, RequestOptions.DEFAULT);
6. System.out.println(response.isAcknowledged());// 是否删除成功
7. restHighLevelClient.close();
8. }

**文档的操作**

**1、文档的添加**

1. // 测试添加文档(先创建一个User实体类，添加fastjson依赖)
2. @Test
3. public void testAddDocument() throws IOException {
4. // 创建一个User对象
5. User liuyou = new User("liuyou", 18);
6. // 创建请求
7. IndexRequest request = new IndexRequest("liuyou\_index");
8. // 制定规则 PUT /liuyou\_index/\_doc/1
9. request.id("1");// 设置文档ID
10. request.timeout(TimeValue.timeValueMillis(1000));// request.timeout("1s")
11. // 将我们的数据放入请求中
12. request.source(JSON.toJSONString(liuyou), XContentType.JSON);
13. // 客户端发送请求，获取响应的结果
14. IndexResponse response = restHighLevelClient.index(request, RequestOptions.DEFAULT);
15. System.out.println(response.status());// 获取建立索引的状态信息 CREATED
16. System.out.println(response);// 查看返回内容 IndexResponse[index=liuyou\_index,type=\_doc,id=1,version=1,result=created,seqNo=0,primaryTerm=1,shards={"total":2,"successful":1,"failed":0}]
17. }

**2、文档信息的获取**

1. // 测试获得文档信息
2. @Test
3. public void testGetDocument() throws IOException {
4. GetRequest request = new GetRequest("liuyou\_index","1");
5. GetResponse response = restHighLevelClient.get(request, RequestOptions.DEFAULT);
6. System.out.println(response.getSourceAsString());// 打印文档内容
7. System.out.println(request);// 返回的全部内容和命令是一样的
8. restHighLevelClient.close();
9. }

**3、文档的获取，并判断其是否存在**

1. // 获取文档，判断是否存在 get /liuyou\_index/\_doc/1
2. @Test
3. public void testDocumentIsExists() throws IOException {
4. GetRequest request = new GetRequest("liuyou\_index", "1");
5. // 不获取返回的 \_source的上下文了
6. request.fetchSourceContext(new FetchSourceContext(false));
7. request.storedFields("\_none\_");
8. boolean exists = restHighLevelClient.exists(request, RequestOptions.DEFAULT);
9. System.out.println(exists);
10. }

**4、文档的更新**

1. // 测试更新文档内容
2. @Test
3. public void testUpdateDocument() throws IOException {
4. UpdateRequest request = new UpdateRequest("liuyou\_index", "1");
5. User user = new User("lmk",11);
6. request.doc(JSON.toJSONString(user),XContentType.JSON);
7. UpdateResponse response = restHighLevelClient.update(request, RequestOptions.DEFAULT);
8. System.out.println(response.status()); // OK
9. restHighLevelClient.close();
10. }

**5、文档的删除**

1. // 测试删除文档
2. @Test
3. public void testDeleteDocument() throws IOException {
4. DeleteRequest request = new DeleteRequest("liuyou\_index", "1");
5. request.timeout("1s");
6. DeleteResponse response = restHighLevelClient.delete(request, RequestOptions.DEFAULT);
7. System.out.println(response.status());// OK
8. }

**6、文档的查询**

1. // 查询
2. // SearchRequest 搜索请求
3. // SearchSourceBuilder 条件构造
4. // HighlightBuilder 高亮
5. // TermQueryBuilder 精确查询
6. // MatchAllQueryBuilder
7. // xxxQueryBuilder ...
8. @Test
9. public void testSearch() throws IOException {
10. // 1.创建查询请求对象
11. SearchRequest searchRequest = new SearchRequest();
12. // 2.构建搜索条件
13. SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
14. // (1)查询条件 使用QueryBuilders工具类创建
15. // 精确查询
16. TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("name", "liuyou");
17. // // 匹配查询
18. // MatchAllQueryBuilder matchAllQueryBuilder = QueryBuilders.matchAllQuery();
19. // (2)其他<可有可无>：（可以参考 SearchSourceBuilder 的字段部分）
20. // 设置高亮
21. searchSourceBuilder.highlighter(new HighlightBuilder());
22. // // 分页
23. // searchSourceBuilder.from();
24. // searchSourceBuilder.size();
25. searchSourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));
26. // (3)条件投入
27. searchSourceBuilder.query(termQueryBuilder);
28. // 3.添加条件到请求
29. searchRequest.source(searchSourceBuilder);
30. // 4.客户端查询请求
31. SearchResponse search = restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT);
32. // 5.查看返回结果
33. SearchHits hits = search.getHits();
34. System.out.println(JSON.toJSONString(hits));
35. System.out.println("=======================");
36. for (SearchHit documentFields : hits.getHits()) {
37. System.out.println(documentFields.getSourceAsMap());
38. }
39. }

官网Api文档:

<https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/java-rest-high-search.html>

###### QueryBuilder简单查询



**前面的操作都无法批量添加数据**

1. // 上面的这些api无法批量增加数据（只会保留最后一个source）
2. @Test
3. public void test() throws IOException {
4. IndexRequest request = new IndexRequest("bulk");// 没有id会自动生成一个随机ID
5. request.source(JSON.toJSONString(new User("liu",1)),XContentType.JSON);
6. request.source(JSON.toJSONString(new User("min",2)),XContentType.JSON);
7. request.source(JSON.toJSONString(new User("kai",3)),XContentType.JSON);
8. IndexResponse index = restHighLevelClient.index(request, RequestOptions.DEFAULT);
9. System.out.println(index.status());// created
10. }

**7、批量添加数据**

1. // 特殊的，真的项目一般会 批量插入数据
2. @Test
3. public void testBulk() throws IOException {
4. BulkRequest bulkRequest = new BulkRequest();
5. bulkRequest.timeout("10s");
6. ArrayList<User> users = new ArrayList<>();
7. users.add(new User("liuyou-1",1));
8. users.add(new User("liuyou-2",2));
9. users.add(new User("liuyou-3",3));
10. users.add(new User("liuyou-4",4));
11. users.add(new User("liuyou-5",5));
12. users.add(new User("liuyou-6",6));
13. // 批量请求处理
14. for (int i = 0; i < users.size(); i++) {
15. bulkRequest.add(
16. // 这里是数据信息
17. new IndexRequest("bulk")
18. .id(""+(i + 1)) // 没有设置id 会自定生成一个随机id
19. .source(JSON.toJSONString(users.get(i)),XContentType.JSON)
20. );
21. }
22. BulkResponse bulk = restHighLevelClient.bulk(bulkRequest, RequestOptions.DEFAULT);
23. System.out.println(bulk.status());// ok
24. }

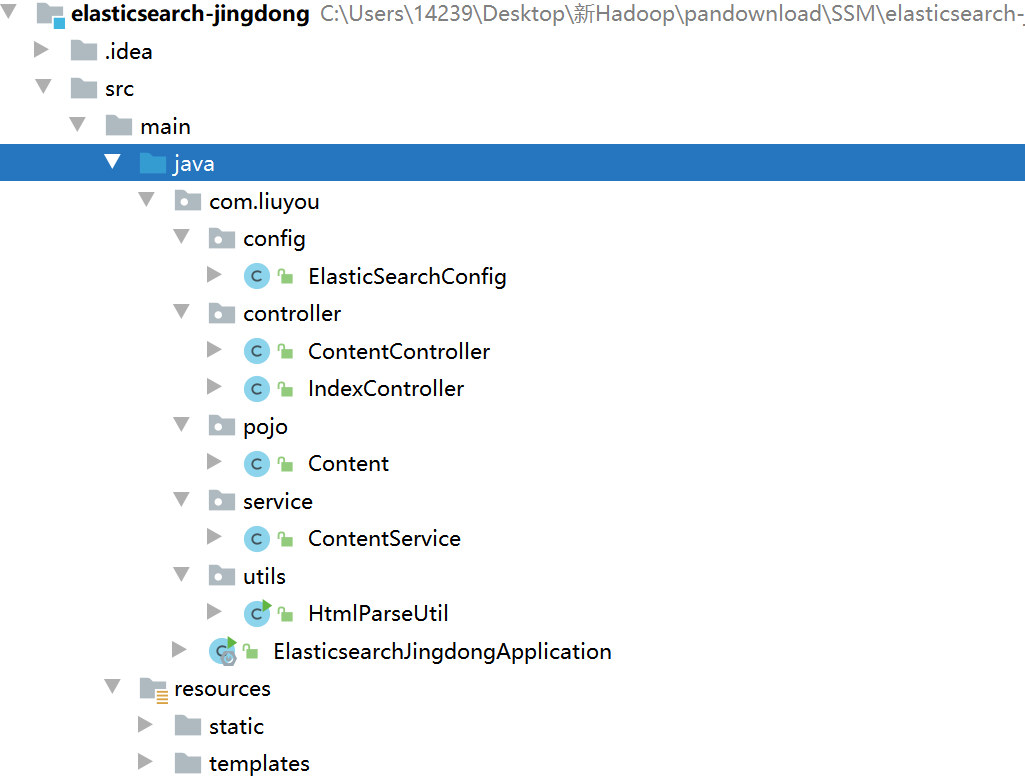
**ElasticSearch实战**

**防京东商城搜索（高亮）**

**1、工程创建（springboot）**

创建过程略

**目录结构**



**2、基本编码**

**①导入依赖**

1. <properties>
2. <java.version>1.8</java.version>
3. <elasticsearch.version>7.6.1</elasticsearch.version>
4. </properties>
5. <dependencies>
6. <!-- jsoup解析页面 -->
7. <!-- 解析网页 爬视频可 研究tiko -->
8. <dependency>
9. <groupId>org.jsoup</groupId>
10. <artifactId>jsoup</artifactId>
11. <version>1.10.2</version>
12. </dependency>
13. <!-- fastjson -->
14. <dependency>
15. <groupId>com.alibaba</groupId>
16. <artifactId>fastjson</artifactId>
17. <version>1.2.70</version>
18. </dependency>
19. <!-- ElasticSearch -->
20. <dependency>
21. <groupId>org.springframework.boot</groupId>
22. <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
23. </dependency>
24. <!-- thymeleaf -->
25. <dependency>
26. <groupId>org.springframework.boot</groupId>
27. <artifactId>spring-boot-starter-thymeleaf</artifactId>
28. </dependency>
29. <!-- web -->
30. <dependency>
31. <groupId>org.springframework.boot</groupId>
32. <artifactId>spring-boot-starter-web</artifactId>
33. </dependency>
34. <!-- devtools热部署 -->
35. <dependency>
36. <groupId>org.springframework.boot</groupId>
37. <artifactId>spring-boot-devtools</artifactId>
38. <scope>runtime</scope>
39. <optional>true</optional>
40. </dependency>
41. <!-- -->
42. <dependency>
43. <groupId>org.springframework.boot</groupId>
44. <artifactId>spring-boot-configuration-processor</artifactId>
45. <optional>true</optional>
46. </dependency>
47. <!-- lombok 需要安装插件 -->
48. <dependency>
49. <groupId>org.projectlombok</groupId>
50. <artifactId>lombok</artifactId>
51. <optional>true</optional>
52. </dependency>
53. <!-- test -->
54. <dependency>
55. <groupId>org.springframework.boot</groupId>
56. <artifactId>spring-boot-starter-test</artifactId>
57. <scope>test</scope>
58. </dependency>
59. </dependencies>

**②导入前端素材**

略

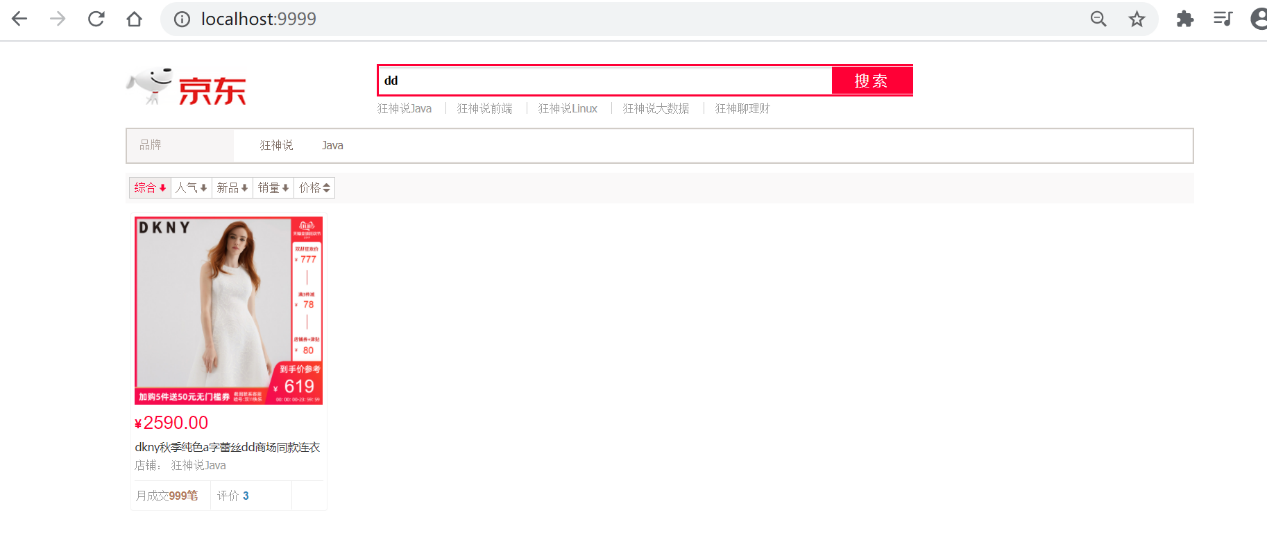
**③编写 application.preperties配置文件**

1. # 更改端口，防止冲突
2. server.port=9999
3. # 关闭thymeleaf缓存
4. spring.thymeleaf.cache=false

**④测试controller和view**

1. @Controller
2. public class IndexController {
3. @GetMapping({"/","index"})
4. public String index(){
5. return "index";
6. }
7. }

访问 localhost:9999



到这里可以先去编写爬虫，编写之后，回到这里

**⑤编写Config**

1. @Configuration
2. public class ElasticSearchConfig {
3. @Bean
4. public RestHighLevelClient restHighLevelClient(){
5. RestHighLevelClient client = new RestHighLevelClient(
6. RestClient.builder(
7. new HttpHost("127.0.0.1",9200,"http")
8. )
9. );
10. return client;
11. }
12. }

**⑥编写service**

因为是爬取的数据，那么就不走Dao，以下编写都不会编写接口，开发中必须严格要求编写

**ContentService**

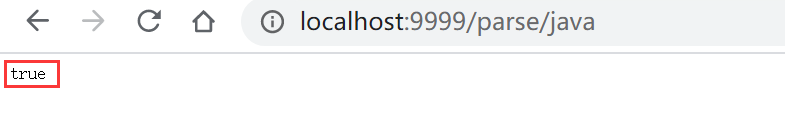
1. @Service
2. public class ContentService {
3. @Autowired
4. private RestHighLevelClient restHighLevelClient;
5. // 1、解析数据放入 es 索引中
6. public Boolean parseContent(String keyword) throws IOException {
7. // 获取内容
8. List<Content> contents = HtmlParseUtil.parseJD(keyword);
9. // 内容放入 es 中
10. BulkRequest bulkRequest = new BulkRequest();
11. bulkRequest.timeout("2m"); // 可更具实际业务是指
12. for (int i = 0; i < contents.size(); i++) {
13. bulkRequest.add(
14. new IndexRequest("jd\_goods")
15. .id(""+(i+1))
16. .source(JSON.toJSONString(contents.get(i)), XContentType.JSON)
17. );
18. }
19. BulkResponse bulk = restHighLevelClient.bulk(bulkRequest, RequestOptions.DEFAULT);
20. restHighLevelClient.close();
21. return !bulk.hasFailures();
22. }
23. // 2、根据keyword分页查询结果
24. public List<Map<String, Object>> search(String keyword, Integer pageIndex, Integer pageSize) throws IOException {
25. if (pageIndex < 0){
26. pageIndex = 0;
27. }
28. SearchRequest jd\_goods = new SearchRequest("jd\_goods");
29. // 创建搜索源建造者对象
30. SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
31. // 条件采用：精确查询 通过keyword查字段name
32. TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("name", keyword);
33. searchSourceBuilder.query(termQueryBuilder);
34. searchSourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));// 60s
35. // 分页
36. searchSourceBuilder.from(pageIndex);
37. searchSourceBuilder.size(pageSize);
38. // 高亮
39. // ....
40. // 搜索源放入搜索请求中
41. jd\_goods.source(searchSourceBuilder);
42. // 执行查询，返回结果
43. SearchResponse searchResponse = restHighLevelClient.search(jd\_goods, RequestOptions.DEFAULT);
44. restHighLevelClient.close();
45. // 解析结果
46. SearchHits hits = searchResponse.getHits();
47. List<Map<String,Object>> results = new ArrayList<>();
48. for (SearchHit documentFields : hits.getHits()) {
49. Map<String, Object> sourceAsMap = documentFields.getSourceAsMap();
50. results.add(sourceAsMap);
51. }
52. // 返回查询的结果
53. return results;
54. }
55. }

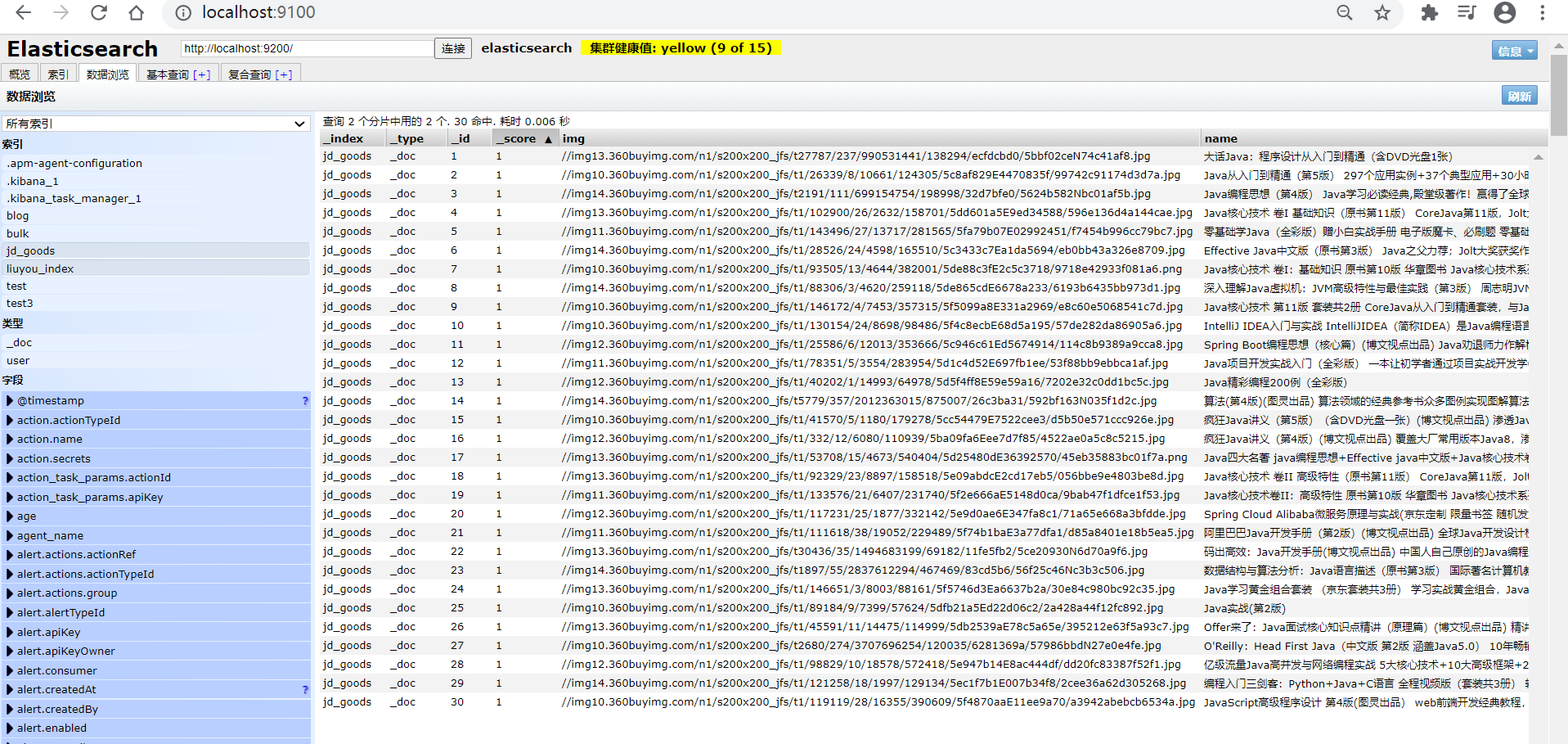
**⑦编写controller**

1. @Controller
2. public class ContentController {
3. @Autowired
4. private ContentService contentService;
5. @ResponseBody
6. @GetMapping("/parse/{keyword}")
7. public Boolean parse(@PathVariable("keyword") String keyword) throws IOException {
8. return contentService.parseContent(keyword);
9. }
10. @ResponseBody
11. @GetMapping("/search/{keyword}/{pageIndex}/{pageSize}")
12. public List<Map<String, Object>> parse(@PathVariable("keyword") String keyword,
13. @PathVariable("pageIndex") Integer pageIndex,
14. @PathVariable("pageSize") Integer pageSize) throws IOException {
15. return contentService.search(keyword,pageIndex,pageSize);
16. }
17. }

**⑧测试结果**

**1、解析数据放入 es 索引中**





**2、根据keyword分页查询结果**

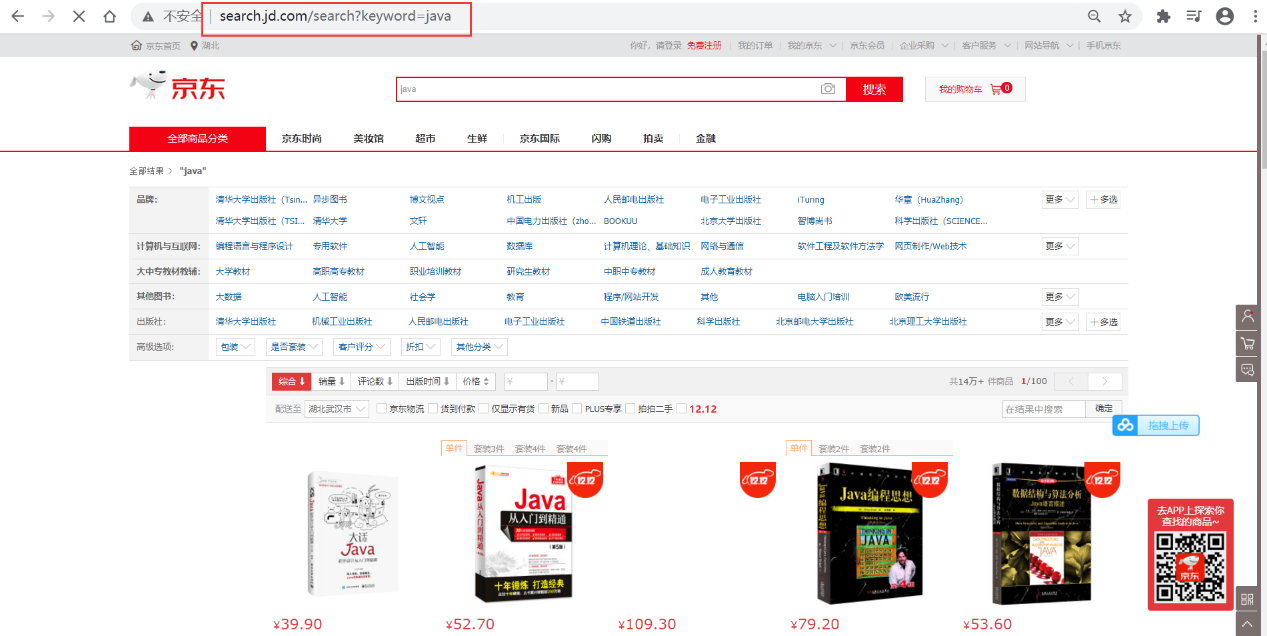
**3、爬虫（jsoup）**

数据获取：数据库、消息队列、爬虫、…

**①搜索京东搜索页面，并分析页面**

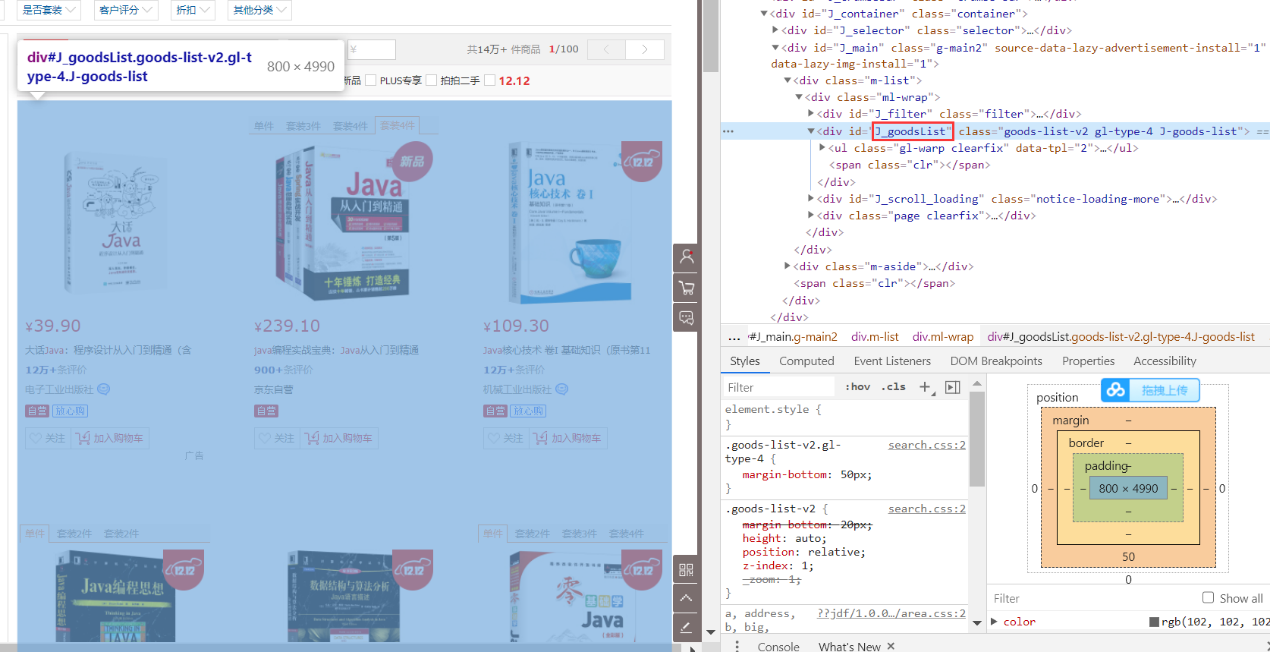
1. http://search.jd.com/search?keyword=java

**页面如下**

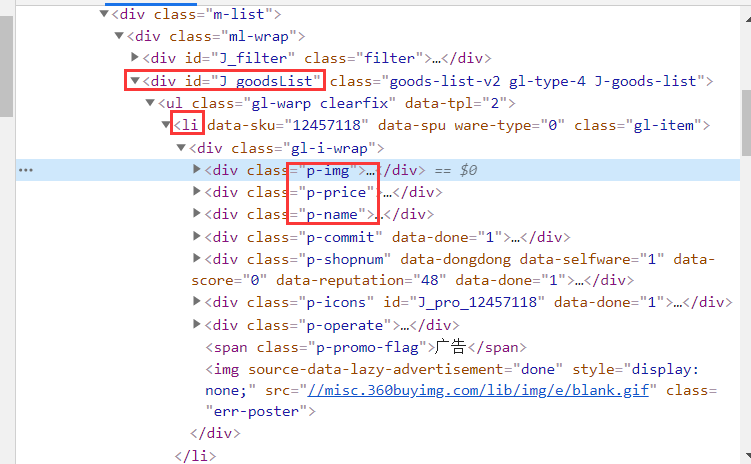


**审查页面元素**

**页面列表id：J\_goodsList**



**目标元素：img、price、name**



**②爬取数据（获取请求返回的页面信息，筛选出可用的）**

**创建HtmlParseUtil，并简单编写**

1. public class HtmlParseUtil {
2. public static void main(String[] args) throws IOException {
3. /// 使用前需要联网
4. // 请求url
5. String url = "http://search.jd.com/search?keyword=java";
6. // 1.解析网页(jsoup 解析返回的对象是浏览器Document对象)
7. Document document = Jsoup.parse(new URL(url), 30000);
8. // 使用document可以使用在js对document的所有操作
9. // 2.获取元素（通过id）
10. Element j\_goodsList = document.getElementById("J\_goodsList");
11. // 3.获取J\_goodsList ul 每一个 li
12. Elements lis = j\_goodsList.getElementsByTag("li");
13. // 4.获取li下的 img、price、name
14. for (Element li : lis) {
15. String img = li.getElementsByTag("img").eq(0).attr("src");// 获取li下 第一张图片
16. String name = li.getElementsByClass("p-name").eq(0).text();
17. String price = li.getElementsByClass("p-price").eq(0).text();
18. System.out.println("=======================");
19. System.out.println("img : " + img);
20. System.out.println("name : " + name);
21. System.out.println("price : " + price);
22. }
23. }
24. }

**运行结果**



**原因是啥？**

一般图片特别多的网站，所有的图片都是通过延迟加载的

1. // 打印标签内容
2. Elements lis = j\_goodsList.getElementsByTag("li");
3. System.out.println(lis);

打印所有li标签，发现img标签中并没有属性src的设置，只是data-lazy-ing设置图片加载的地址



**创建HtmlParseUtil、改写**

* 更改图片获取属性为 data-lazy-img
* 与实体类结合，实体类如下
  1. [@Data](https://github.com/Data)
  2. [@AllArgsConstructor](https://github.com/AllArgsConstructor)
  3. [@NoArgsConstructor](https://github.com/NoArgsConstructor)
  4. public class Content implements Serializable {
  5. private static final long serialVersionUID = -8049497962627482693L;
  6. private String name;
  7. private String img;
  8. private String price;
  9. }
* 封装为方法

1. public class HtmlParseUtil {
2. public static void main(String[] args) throws IOException {
3. System.out.println(parseJD("java"));
4. }
5. public static List<Content> parseJD(String keyword) throws IOException {
6. /// 使用前需要联网
7. // 请求url
8. String url = "http://search.jd.com/search?keyword=" + keyword;
9. // 1.解析网页(jsoup 解析返回的对象是浏览器Document对象)
10. Document document = Jsoup.parse(new URL(url), 30000);
11. // 使用document可以使用在js对document的所有操作
12. // 2.获取元素（通过id）
13. Element j\_goodsList = document.getElementById("J\_goodsList");
14. // 3.获取J\_goodsList ul 每一个 li
15. Elements lis = j\_goodsList.getElementsByTag("li");
16. // System.out.println(lis);
17. // 4.获取li下的 img、price、name
18. // list存储所有li下的内容
19. List<Content> contents = new ArrayList<Content>();
20. for (Element li : lis) {
21. // 由于网站图片使用懒加载，将src属性替换为data-lazy-img
22. String img = li.getElementsByTag("img").eq(0).attr("data-lazy-img");// 获取li下 第一张图片
23. String name = li.getElementsByClass("p-name").eq(0).text();
24. String price = li.getElementsByClass("p-price").eq(0).text();
25. // 封装为对象
26. Content content = new Content(name,img,price);
27. // 添加到list中
28. contents.add(content);
29. }
30. // System.out.println(contents);
31. // 5.返回 list
32. return contents;
33. }
34. }

**结果展示**



**4、搜索高亮**

在3、的基础上添加内容

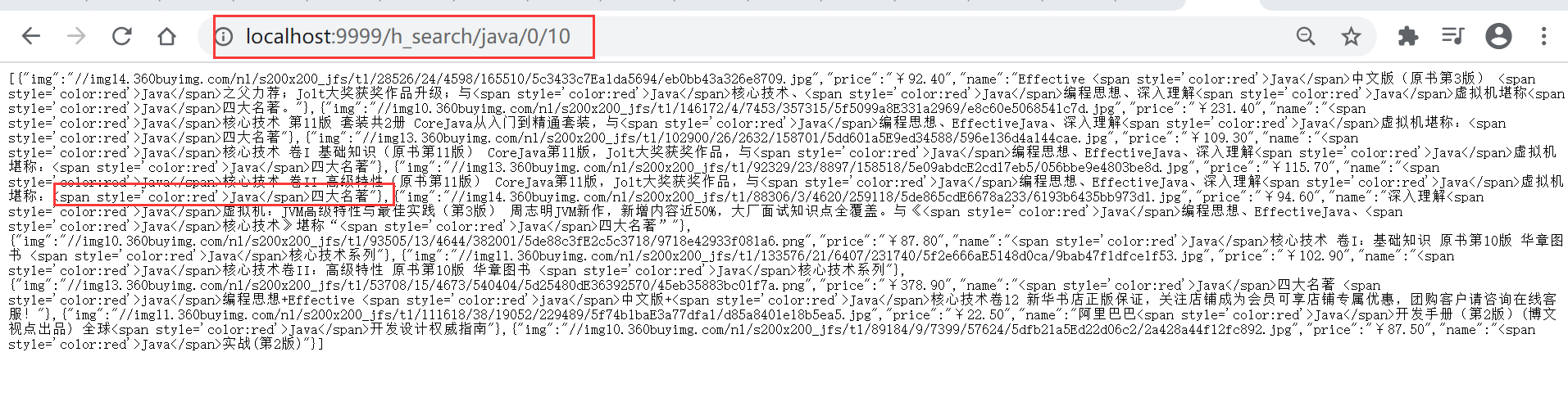
**①ContentService**

1. // 3、 在2的基础上进行高亮查询
2. public List<Map<String, Object>> highlightSearch(String keyword, Integer pageIndex, Integer pageSize) throws IOException {
3. SearchRequest searchRequest = new SearchRequest("jd\_goods");
4. SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
5. // 精确查询，添加查询条件
6. TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("name", keyword);
7. searchSourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));
8. searchSourceBuilder.query(termQueryBuilder);
9. // 分页
10. searchSourceBuilder.from(pageIndex);
11. searchSourceBuilder.size(pageSize);
12. // 高亮 =========
13. HighlightBuilder highlightBuilder = new HighlightBuilder();
14. highlightBuilder.field("name");
15. highlightBuilder.preTags("<span style='color:red'>");
16. highlightBuilder.postTags("</span>");
17. searchSourceBuilder.highlighter(highlightBuilder);
18. // 执行查询
19. searchRequest.source(searchSourceBuilder);
20. SearchResponse searchResponse = restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT);
21. // 解析结果 ==========
22. SearchHits hits = searchResponse.getHits();
23. List<Map<String, Object>> results = new ArrayList<>();
24. for (SearchHit documentFields : hits.getHits()) {
25. // 使用新的字段值（高亮），覆盖旧的字段值
26. Map<String, Object> sourceAsMap = documentFields.getSourceAsMap();
27. // 高亮字段
28. Map<String, HighlightField> highlightFields = documentFields.getHighlightFields();
29. HighlightField name = highlightFields.get("name");
30. // 替换
31. if (name != null){
32. Text[] fragments = name.fragments();
33. StringBuilder new\_name = new StringBuilder();
34. for (Text text : fragments) {
35. new\_name.append(text);
36. }
37. sourceAsMap.put("name",new\_name.toString());
38. }
39. results.add(sourceAsMap);
40. }
41. return results;
42. }

**②ContentController**

1. @ResponseBody
2. @GetMapping("/h\_search/{keyword}/{pageIndex}/{pageSize}")
3. public List<Map<String, Object>> highlightParse(@PathVariable("keyword") String keyword,
4. @PathVariable("pageIndex") Integer pageIndex,
5. @PathVariable("pageSize") Integer pageSize) throws IOException {
6. return contentService.highlightSearch(keyword,pageIndex,pageSize);
7. }

**③结果展示**



**5、前后端分离（简单使用Vue）**

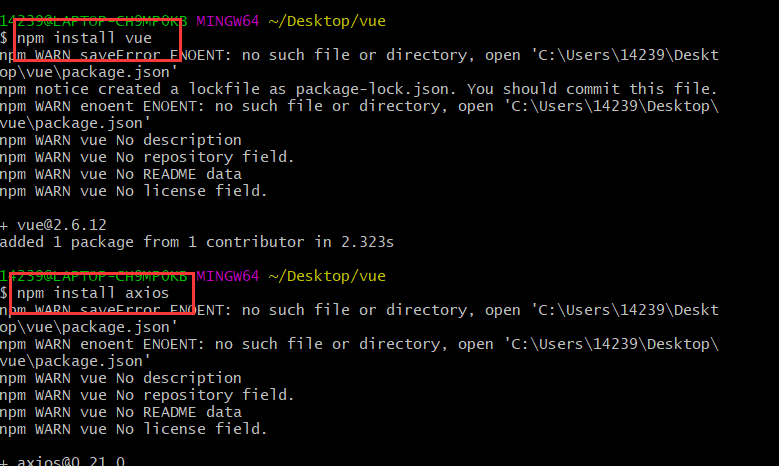
删除Controller 方法上的 [@ResponseBody](https://github.com/ResponseBody)注解

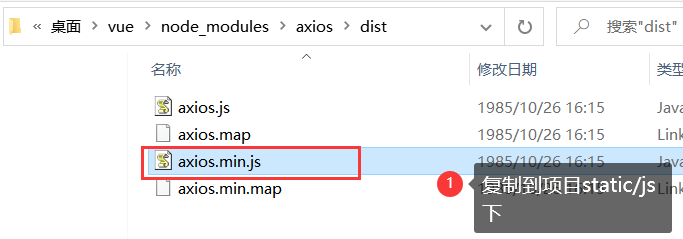


**①下载并引入Vue.min.js和axios.js**

如果安装了nodejs，可以按如下步骤，没有可以到后面素材处下载

1. npm install vue
2. npm install axios







**②修改静态页面**

**引入js**

1. <script th:src="@{/js/vue.min.js}"></script>
2. <script th:src="@{/js/axios.min.js}"></script>

**修改后的index.html**

1. <!DOCTYPE html>
2. <html xmlns:th="http://www.thymeleaf.org">
3. <head>
4. <meta charset="utf-8"/>
5. <title>狂神说Java-ES仿京东实战</title>
6. <link rel="stylesheet" th:href="@{/css/style.css}"/>
7. <script th:src="@{/js/jquery.min.js}"></script>
8. </head>
9. <body class="pg">
10. <div class="page">
11. <div id="app" class=" mallist tmall- page-not-market ">
12. <!-- 头部搜索 -->
13. <div id="header" class=" header-list-app">
14. <div class="headerLayout">
15. <div class="headerCon ">
16. <!-- Logo-->
17. <h1 id="mallLogo">
18. <img th:src="@{/images/jdlogo.png}" alt="">
19. </h1>
20. <div class="header-extra">
21. <!--搜索-->
22. <div id="mallSearch" class="mall-search">
23. <form name="searchTop" class="mallSearch-form clearfix">
24. <fieldset>
25. <legend>天猫搜索</legend>
26. <div class="mallSearch-input clearfix">
27. <div class="s-combobox" id="s-combobox-685">
28. <div class="s-combobox-input-wrap">
29. <input v-model="keyword" type="text" autocomplete="off" id="mq"
30. class="s-combobox-input" aria-haspopup="true">
31. </div>
32. </div>
33. <button type="submit" @click.prevent="searchKey" id="searchbtn">搜索</button>
34. </div>
35. </fieldset>
36. </form>
37. <ul class="relKeyTop">
38. <li><a>狂神说Java</a></li>
39. <li><a>狂神说前端</a></li>
40. <li><a>狂神说Linux</a></li>
41. <li><a>狂神说大数据</a></li>
42. <li><a>狂神聊理财</a></li>
43. </ul>
44. </div>
45. </div>
46. </div>
47. </div>
48. </div>
49. <!-- 商品详情页面 -->
50. <div id="content">
51. <div class="main">
52. <!-- 品牌分类 -->
53. <form class="navAttrsForm">
54. <div class="attrs j\_NavAttrs" style="display:block">
55. <div class="brandAttr j\_nav\_brand">
56. <div class="j\_Brand attr">
57. <div class="attrKey">
58. 品牌
59. </div>
60. <div class="attrValues">
61. <ul class="av-collapse row-2">
62. <li><a href="#"> 狂神说 </a></li>
63. <li><a href="#"> Java </a></li>
64. </ul>
65. </div>
66. </div>
67. </div>
68. </div>
69. </form>
70. <!-- 排序规则 -->
71. <div class="filter clearfix">
72. <a class="fSort fSort-cur">综合<i class="f-ico-arrow-d"></i></a>
73. <a class="fSort">人气<i class="f-ico-arrow-d"></i></a>
74. <a class="fSort">新品<i class="f-ico-arrow-d"></i></a>
75. <a class="fSort">销量<i class="f-ico-arrow-d"></i></a>
76. <a class="fSort">价格<i class="f-ico-triangle-mt"></i><i class="f-ico-triangle-mb"></i></a>
77. </div>
78. <!-- 商品详情 -->
79. <div class="view grid-nosku" >
80. <div class="product" v-for="result in results">
81. <div class="product-iWrap">
82. <!--商品封面-->
83. <div class="productImg-wrap">
84. <a class="productImg">
85. <img :src="result.img">
86. </a>
87. </div>
88. <!--价格-->
89. <p class="productPrice">
90. <em v-text="result.price"></em>
91. </p>
92. <!--标题-->
93. <p class="productTitle">
94. <a v-html="result.name"></a>
95. </p>
96. <!-- 店铺名 -->
97. <div class="productShop">
98. <span>店铺： 狂神说Java </span>
99. </div>
100. <!-- 成交信息 -->
101. <p class="productStatus">
102. <span>月成交<em>999笔</em></span>
103. <span>评价 <a>3</a></span>
104. </p>
105. </div>
106. </div>
107. </div>
108. </div>
109. </div>
110. </div>
111. </div>
112. <script th:src="@{/js/vue.min.js}"></script>
113. <script th:src="@{/js/axios.min.js}"></script>
114. <script>
115. new Vue({
116. el:"#app",
117. data:{
118. "keyword": '', // 搜索的关键字
119. "results":[] // 后端返回的结果
120. },
121. methods:{
122. searchKey(){
123. var keyword = this.keyword;
124. console.log(keyword);
125. axios.get('h\_search/'+keyword+'/0/20').then(response=>{
126. console.log(response.data);
127. this.results=response.data;
128. })
129. }
130. }
131. });
132. </script>
133. </body>
134. </html>

**测试**



**安装包及前端素材**

链接：<https://pan.baidu.com/s/1M5uWdYsCZyzIAOcgcRkA_A>  
提取码：qk8p  
复制这段内容后打开百度网盘手机App，操作更方便哦

**疑惑：**

**1、使用term（精确查询）时，我发现三个问题，问题如下：**

* 字段值必须是一个词（索引中存在的词），才能匹配
  + 问题：中文字符串，term查询时无法查询到数据（比如，“编程”两字在文档中存在，但是搜索不到）
  + 原因：索引为配置中文分词器（默认使用standard，即所有中文字符串都会被切分为单个中文汉字作为单词），所以没有超过1个汉字的词，也就无法匹配，进而查不到数据
  + 解决：创建索引时配置中文分词器，如
    1. PUT example
    2. {
    3. "mappings": {
    4. "properties": {
    5. "name":{
    6. "type": "text",
    7. "analyzer": "ik\_max\_word" // ik分词器
    8. }
    9. }
    10. }
    11. }
* 查询的英文字符只能是小写，大写都无效
* 查询时英文单词必须是完整的

版权声明：本文为博主原创文章，遵循[CC 4.0 BY-SA](https://creativecommons.org/licenses/by-sa/4.0/)版权协议,转载请附上原文出处链接和本声明，KuangStudy,以学为伴，一生相伴！

[本文链接：https://www.kuangstudy.com/bbs/1354069127022583809](https://www.kuangstudy.com/bbs/1354069127022583809)