# 项目课程系列之ElasticSearch 7.4

## 尚硅谷JavaEE教研组

## 版本：V3.0

# 第一章 ElasticSearch简介

## 1.1 什么是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。

### 1.1.1 什么是全文检索

#### 1.1.1.1 数据分类

我们生活中的数据总体分为两种：**结构化数据**和**非结构化数据**。

* 结构化数据：指具有固定格式或有限长度的数据，如数据库，元数据等。
* 非结构化数据：指不定长或无固定格式的数据，如邮件，word文档等磁盘上的文件

#### 1.1.1.2 结构化数据搜索

常见的结构化数据也就是数据库中的数据。在数据库中搜索很容易实现，通常都是使用sql语句进行查询，而且能很快的得到查询结果。

为什么数据库搜索很容易？

因为数据库中的数据存储是有规律的，有行有列而且数据格式、数据长度都是固定的。

#### 1.1.1.3 非结构化数据查询方法

（1）顺序扫描法

所谓顺序扫描，比如要找内容包含某一个字符串的文件，就是一个文档一个文档的看，对于每一个文档，从头看到尾，如果此文档包含此字符串，则此文档为我们要找的文件，接着看下一个文件，直到扫描完所有的文件。如利用windows的搜索也可以搜索文件内容，只是相当的慢。

（2）全文检索

将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对此有一定结构的数据进行搜索，从而达到搜索相对较快的目的。这部分从非结构化数据中提取出的然后重新组织的信息，我们称之索引。

例如：字典。字典的拼音表和部首检字表就相当于字典的索引，对每一个字的解释是非结构化的，如果字典没有音节表和部首检字表，在茫茫辞海中找一个字只能顺序扫描。然而字的某些信息可以提取出来进行结构化处理，比如读音，就比较结构化，分声母和韵母，于是将读音拿出来按一定的顺序排列，每一项读音都指向此字的详细解释的页数。我们搜索时按结构化的拼音搜到读音，然后按其指向的页数，便可找到我们的非结构化数据——也即对字的解释。

这种先建立索引，再对索引进行搜索的过程就叫全文检索

虽然创建索引的过程也是非常**耗时**的，但是索引一旦创建就可以**多次使用**，全文检索主要处理的是查询，所以耗时间创建索引是值得的。

### 1.1.2 如何实现全文检索

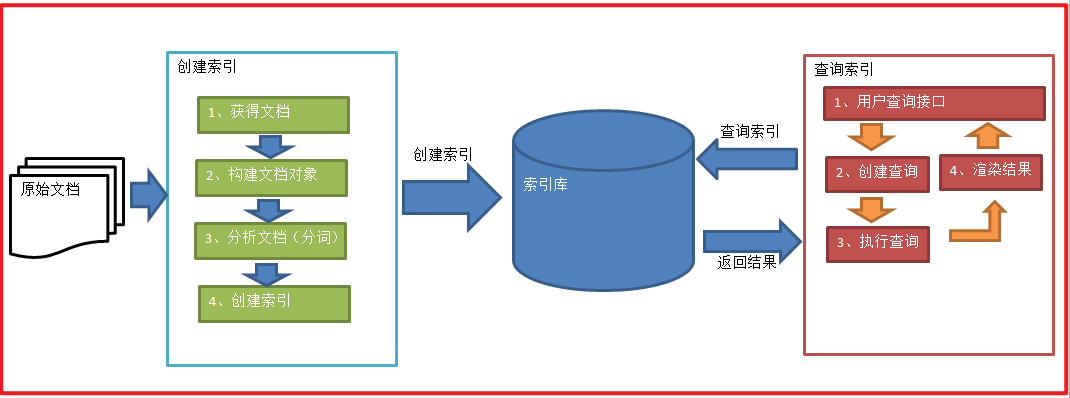
可以使用Lucene实现全文检索。Lucene是apache下的一个开放源代码的全文检索引擎工具包。提供了完整的查询引擎和索引引擎，部分文本分析引擎。Lucene的目的是为软件开发人员提供一个简单易用的工具包，以方便的在目标系统中实现全文检索的功能。

### 1.1.3 全文检索的应用场景

对于数据量大、数据结构不固定的数据可采用全文检索方式搜索，比如：百度、Google等搜索引擎、论坛站内搜索、电商网站站内搜索等。

### 1.1.4 全文检索的实现流程

#### 1.1.4.1 索引和搜索流程图



1、绿色表示索引过程，对要搜索的原始内容进行索引构建一个索引库，索引过程包括：

确定原始内容即要搜索的内容 —->采集文档—->创建文档—->分析文档—->索引文档

2、红色表示搜索过程，从索引库中搜索内容，搜索过程包括：

用户通过搜索界面—->创建查询—->执行搜索，从索引库搜索—->渲染搜索结果

全文检索大体分两个过程，索引创建 (Indexing) 和搜索索引 (Search) 。

* 索引创建：将现实世界中所有的结构化和非结构化数据提取信息，创建索引的过程。
* 搜索索引：就是得到用户的查询请求，搜索创建的索引，然后返回结果的过程。

#### 1.1.4.2 创建索引

对文档索引的过程，将用户要搜索的文档内容进行索引，索引存储在索引库（index）中。

索引里面究竟需要存些什么呢？

首先我们来看为什么**顺序扫描的速度慢**：

其实是由于我们想要搜索的信息和非结构化数据中所存储的信息不一致造成的。

非结构化数据中所存储的信息是每个文件包含哪些字符串，也即已知文件，欲求字符串相对容易，也即是从文件到字符串的映射。

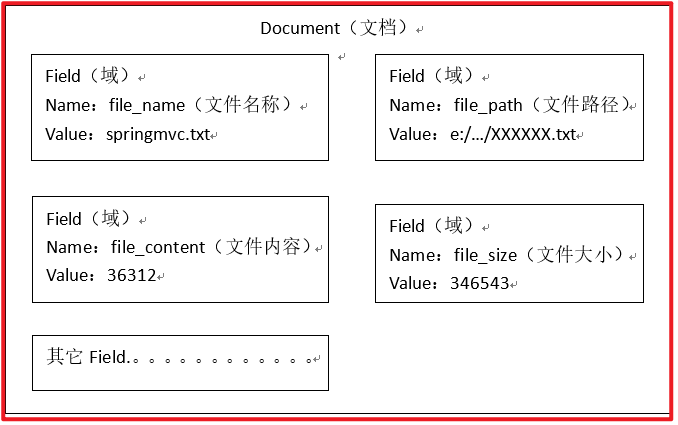
而我们想搜索的信息是哪些文件包含此字符串，也即已知字符串，欲求文件，也即从字符串到文件的映射。

两者恰恰相反。于是如果索引总能够保存从字符串到文件的映射，则会大大提高搜索速度。

#### 1.1.4.3 创建文档对象

获取原始内容的目的是为了索引，在索引前需要将原始内容创建成文档（Document），文档中包括一个一个的域（Field），域中存储内容。

这里我们可以将磁盘上的一个文件当成一个document，Document中包括一些Field（file\_name文件名称、file\_path文件路径、file\_size文件大小、file\_content文件内容），如下图：



注意：每个Document可以有多个Field，不同的Document可以有不同的Field，同一个Document可以有相同的Field（域名和域值都相同）

每个文档都有一个唯一的编号，就是文档id。

#### 1.1.4.4 分析文档

将原始内容创建为包含域（Field）的文档（document），需要再对域中的内容进行分析

#### 1.1.4.5 创建索引

索引的目的是为了搜索，最终要实现只搜索被索引的语汇单元从而找到Document（文档）。

注意：创建索引是对语汇单元索引，通过词语找文档，这种索引的结构叫\*\*倒排索引结构\*\*。

传统方法是根据文件找到该文件的内容，在文件内容中匹配搜索关键字，这种方法是顺序扫描方法，数据量大、搜索慢。

**倒排索引结构**是根据内容（词语）找文档，如下图：

正排索引：



转化成倒排索引：



说明：

* “单词ID”一栏记录了每个单词的单词编号；
* 第二栏是对应的单词；
* 第三栏即每个单词对应的倒排列表；
* 比如单词“谷歌”，其单词编号为1，倒排列表为{1,2,3,4,5}，说明文档集合中每个文档都包含了这个单词。

而事实上，索引系统还可以记录除此之外的更多信息，在单词对应的倒排列表中不仅记录了文档编号，还记载了单词频率信息（TF），即这个单词在某个文档中的出现次数，之所以要记录这个信息，是因为词频信息在搜索结果排序时，计算查询和文档相似度是很重要的一个计算因子，所以将其记录在倒排列表中，以方便后续排序时进行分值计算。



倒排索引结构也叫反向索引结构，包括索引和文档两部分，索引即词汇表，它的规模较小，而文档集合较大。

#### 1.1.4.6 查询索引

查询索引也是搜索的过程。搜索就是用户输入关键字，从索引（index）中进行搜索的过程。根据关键字搜索索引，根据索引找到对应的文档，从而找到要搜索的内容（这里指磁盘上的文件）。

#### 1.1.4.7 用户查询接口

全文检索系统提供用户搜索的界面供用户提交搜索的关键字，搜索完成展示搜索结果。

比如：



#### 1.1.4.8 创建查询

用户输入查询关键字执行搜索之前需要先构建一个查询对象，查询对象中可以指定查询要搜索的Field文档域、查询关键字等，查询对象会生成具体的查询语法。

#### 1.1.4.9 执行查询

搜索索引过程：根据查询语法在倒排索引词典表中分别找出对应搜索词的索引，从而找到索引所链接的文档链表。

#### 1.1.4.10 渲染结果

以一个友好的界面将查询结果展示给用户，用户根据搜索结果找自己想要的信息，为了帮助用户很快找到自己的结果，提供了很多展示的效果，比如搜索结果中将关键字高亮显示，百度提供的快照等。



### 1.1.5 Elasticsearch与mysql的区别

#### 1.1.5.1 响应时间

MySQL

背景：当数据库中的文档数仅仅上万条时，关键词查询就比较慢了。如果一旦到企业级的数据，响应速度就会更加不可接受。

原因：在数据库做模糊查询时，如LIKE语句，它会遍历整张表，同时进行字符串匹配。

例如，在数据库查询“手机”时，数据库会在每一条记录去匹配“手机”这两字是否出现。实际上，并不是所有记录都包含“手机”，所以做了很多无用功。

这个步骤都不高效，而且随着数据量的增大，消耗的资源和时间都会线性的增长。

Elasticsearch

提升：小张使用了ES搜索服务后，发现这个问题被很好解决，TB级数据在毫秒级就能返回检索结果，很好地解决了痛点。

原因：Elasticsearch是基于倒排索引的，例子如下。



当搜索“手机”时，Elasticsearch就会立即返回文档F，G，H。这样就不用花多余的时间在其他文档上了，因此检索速度得到了数量级的提升。

#### 1.1.5.2 分词

MySQL

背景：在做中文搜索时，组合词检索在数据库是很难完成的。

例如，当用户在搜索框输入“四川火锅”时，数据库通常只能把这四个字去进行全部匹配。可是在文本中，可能会出现“推荐四川好吃的火锅”，这时候就没有结果了。

原因：数据库并不支持分词。如果人工去开发分词功能，费时费精力。

Elasticsearch

提升：使用ES搜索服务后，就不用太过于关注分词了，因为Elasticsearch支持中文分词插件，很好地解决了问题。

原因：当用户使用Elasticsearch时进行搜索时，Elasticsearch就自动帮他分好词了。

例如 输入“四川火锅”时，Elasticsearch会自动做下面两件事  
（1） 将“四川火锅”分词成“四川”和“火锅”  
（2） 查找包含这两个词的文档

#### 1.1.5.3 相关性

MySQL

背景：在用数据库做搜索时，结果经常会出现一系列不匹配的文档。思考：  
· 到底什么文档是用户真正想要的呢？  
· 怎么才能把用户想看的文档放在搜索列表最前面呢？

原因：数据库并不支持相关性搜索。

例如，当用户搜索“咖啡厅”的时候，他很可能更想知道附近哪里可以喝咖啡，而不是怎么开咖啡厅。

Elasticsearch

提升：使用了ES搜索服务后，发现Elasticsearch能很好地支持相关性评分。通过合理的优化，ES搜索服务能够返回精准的结果，满足用户的需求。

原因：Elasticsearch支持全文搜索和相关度评分。这样在返回结果就会根据分数由高到低排列。分数越高，意味着和查询语句越相关。

例如，当用户搜索“星巴克咖啡”，带有“星巴克咖啡”的信息就要比只包含“咖啡”的信息靠前。

总结

1.传统数据库在全文检索方面很鸡肋，海量数据下的查询很慢，对非结构化文本数据的不支持，ES支持非结构化数据的存储和查询。

2.ES支持分布式文档存储。

3.ES是分布式实时搜索，并且响应时间比关系型数据库快。

4.ES在分词方面比关系型好，能做到精确分词。

5.ES对已有的数据，在数据匹配性方面比关系型数据库好， 例如：搜索“星巴克咖啡”，ES会优先返回带有“星巴克咖啡”的数据，不会优先返回带有“咖啡”的数据。

ElasticSearch和MySql分工不同，MySQL负责存储数据，ElasticSearch负责搜索数据。

## 1.2 ElasticSearch的使用案例

* 2013年初，GitHub抛弃了Solr，采取ElasticSearch 来做PB级的搜索。 “GitHub使用ElasticSearch搜索20TB的数据，包括13亿文件和1300亿行代码”
* 维基百科：启动以elasticsearch为基础的核心搜索架构
* SoundCloud：“SoundCloud使用ElasticSearch为1.8亿用户提供即时而精准的音乐搜索服务”
* 百度：百度目前广泛使用ElasticSearch作为文本数据分析，采集百度所有服务器上的各类指标数据及用户自定义数据，通过对各种数据进行多维分析展示，辅助定位分析实例异常或业务层面异常。目前覆盖百度内部20多个业务线（包括casio、云分析、网盟、预测、文库、直达号、钱包、风控等），单集群最大100台机器，200个ES节点，每天导入30TB+数据
* 新浪使用ES 分析处理32亿条实时日志
* 阿里使用ES 构建挖财自己的日志采集和分析体系

# 第二章 ElasticSearch安装与启动

## 2.1 windows版本安装

### 2.1.1 下载ES压缩包

ElasticSearch分为Linux和Window版本，基于我们主要学习的是ElasticSearch的Java客户端的使用，所以我们课程中使用的是安装较为简便的Window版本，项目上线后，公司的运维人员会安装Linux版的ES供我们连接使用。

ElasticSearch的官方地址： <https://www.elastic.co/products/elasticsearch>

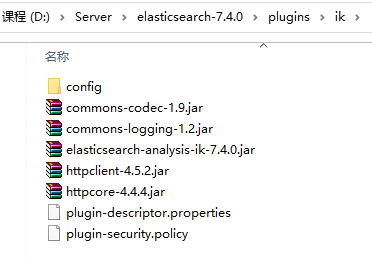
在资料中已经提供了下载好的7.4.0的压缩包：elasticsearch-7.4.0-windows-x86\_64.zip

### 2.1.2 安装ES服务

Window版的ElasticSearch的安装很简单，解压开即安装完毕，解压后的ElasticSearch的目录结构如下：

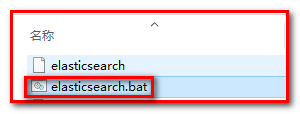


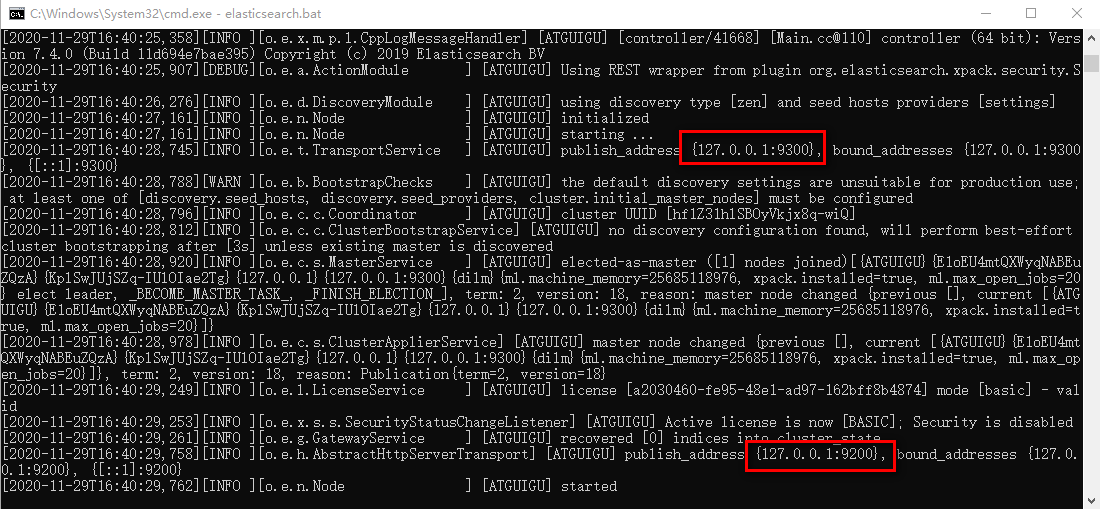
安装IK分词器插件：在plugin目录下创建ik文件夹，将elasticsearch-analysis-ik-7.4.0.zip内容解压到ik目录下：



### 2.1.3 启动ES服务

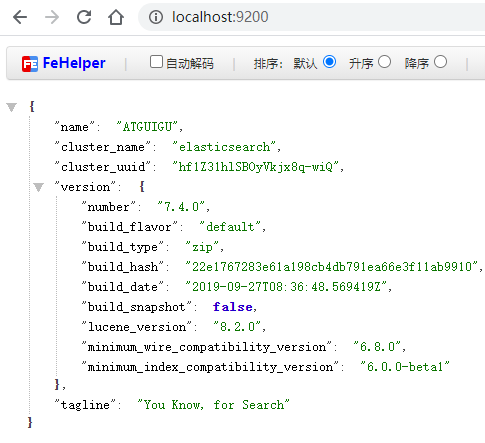
点击ElasticSearch下的bin目录下的elasticsearch.bat启动，控制台显示的日志信息如下：





注意：9300是tcp通讯端口，集群间和TCPClient都执行该端口，9200是http协议的RESTful接口 。

通过浏览器访问ElasticSearch服务器：<http://localhost:9200>



注意事项一：ElasticSearch是使用java开发的，且本版本的es需要的jdk版本要是1.8以上，所以安装ElasticSearch之前保证JDK1.8+安装完毕，并正确的配置好JDK环境变量，否则启动ElasticSearch失败。

注意事项二：出现闪退，通过路径访问发现“空间不足”

【解决方案】

修改jvm.options文件的22行23行， Elasticsearch启动的时候占用1个G的内存，可改成512m：

-Xmx512m：设置JVM最大可用内存为512M。  
-Xms512m：设置JVM初始内存为512m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

## 2.2 Postman客户端

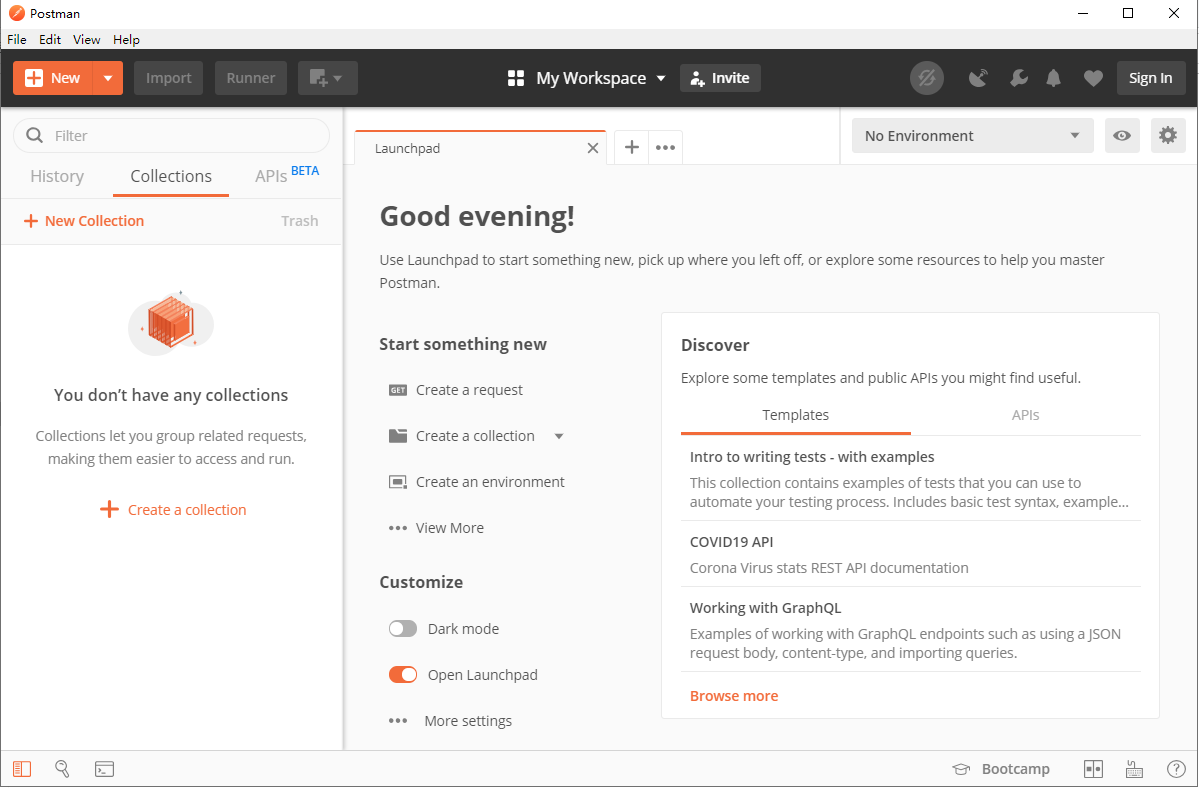
* 什么是Postman

Postman是一个http模拟请求的工具。

官网介绍：“Modern software is built on APIs，Postman helps you develop APIs faster”

Postman中文版是postman这款强大网页调试工具的windows客户端，提供功能强大的Web API & HTTP 请求调试。软件功能非常强大，界面简洁明晰、操作方便快捷，设计得很人性化。Postman中文版能够发送任何类型的HTTP 请求 (GET, HEAD, POST, PUT..)，且可以附带任何数量的参数。

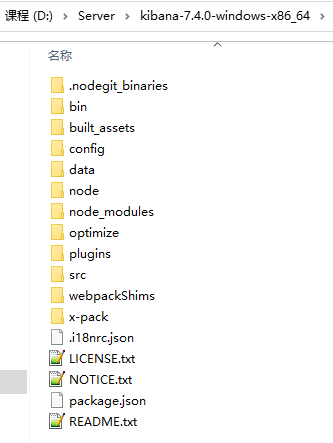
* Postman官网：<https://www.getpostman.com>
* 安装：Postman-win64-7.11.0-Setup.exe，默认在C:\Users\【用户名】\AppData\Local\Postman和C:\Users\【用户名】\AppData\Roaming\Postman目录下。当Postman打不开时，可删除该Postman目录，重新安装即可。



## 2.3 Kibana客户端(Windows版)

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。

解压kibana-7.4.0-windows-x86\_64.zip(解压很慢,耐心等待…)



进入config目录修改kibana.yml第2、28行，配置自身端口和连接的ES服务器地址。

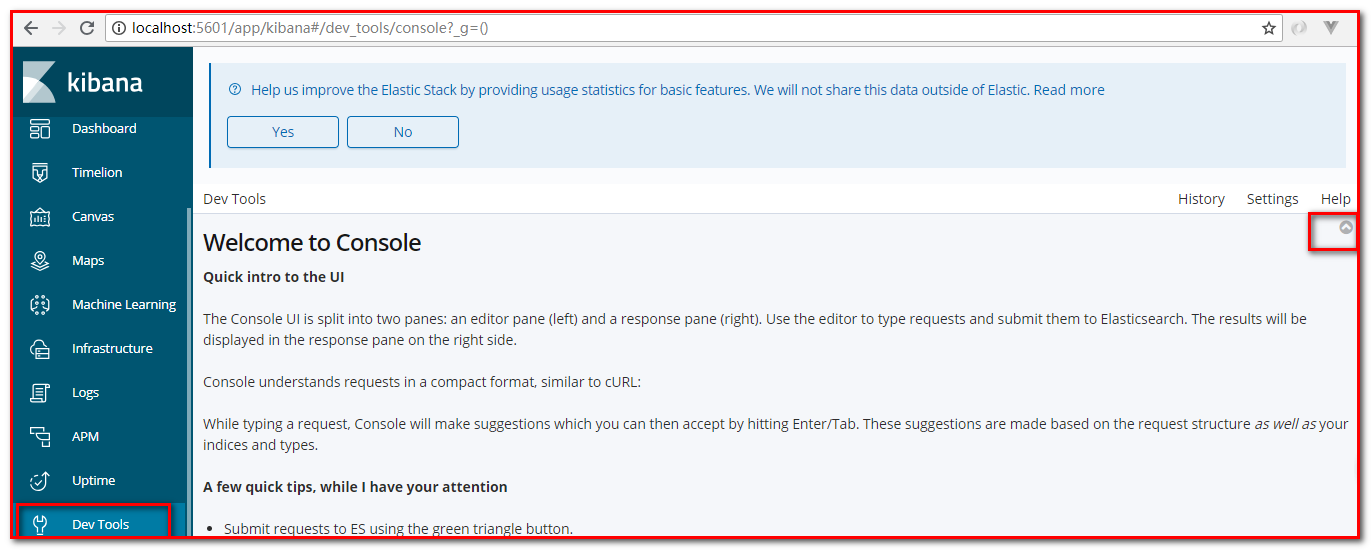
server.port: 5601

elasticsearch.hosts: ["http://localhost:9200"]

进入kibana的bin目录，双击kibana.bat启动：



访问：[http://localhost:5601](http://localhost:5601/)，出现以下界面即完成安装。

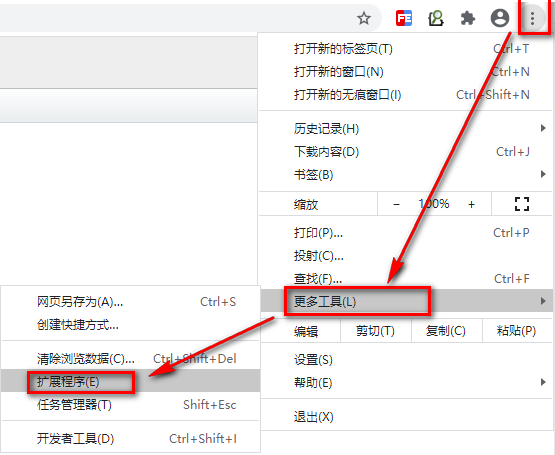


界面是英文的，如果希望是中文，可以修改kibana.yml第114行：i18n.locale: “zh-CN”

## 2.4 Elasticsearch head客户端

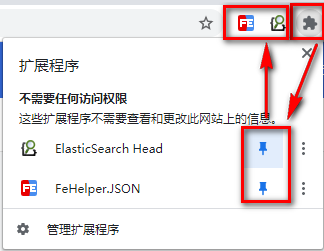
head插件是ES的一个可视化管理插件，用来监视ES的状态，并通过head客户端和ES服务进行交互，比如创建映射、创建索引等。

将ElasticSearch-head-Chrome-0.1.5-Crx4Chrome.crx用压缩工具解压，打开Chrome扩展程序，点” 加载已解压的扩展程序”按钮，找到解压目录即可。

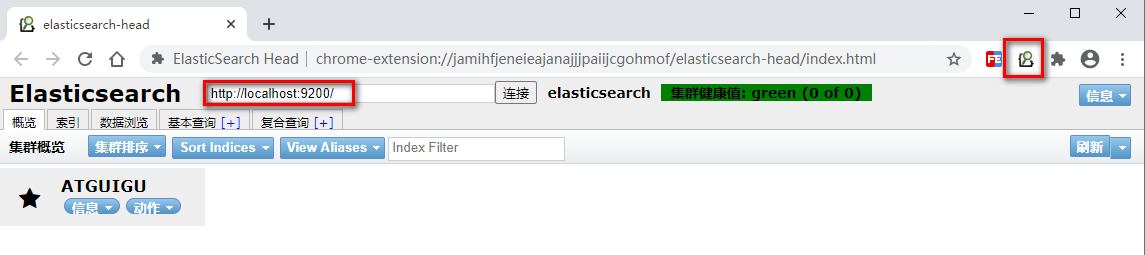




将图标固定在工具条上。



点击head图标，输入ES服务器地址：http://localhost:9200



## 2.5 使用IK分词器

IKAnalyzer是一个开源的，基于java语言开发的轻量级的中文分词工具包。从2006年12月推出1.0版开始，IKAnalyzer已经推出 了3个大版本。最初，它是以开源项目Lucene为应用主体的，结合词典分词和文法分析算法的中文分词组件。新版本的IKAnalyzer3.0则发展为 面向Java的公用分词组件，独立于Lucene项目，同时提供了对Lucene的默认优化实现。

IK分词器3.0的特性如下：

1）采用了特有的“正向迭代最细粒度切分算法“，具有60万字/秒的高速处理能力。  
2）采用了多子处理器分析模式，支持：英文字母（IP地址、Email、URL）、数字（日期，常用中文数量词，罗马数字，科学计数法），中文词汇（姓名、地名处理）等分词处理。  
3）对中英联合支持不是很好,在这方面的处理比较麻烦.需再做一次查询,同时是支持个人词条的优化的词典存储，更小的内存占用。  
4）支持用户词典扩展定义。  
5）针对Lucene全文检索优化的查询分析器IKQueryParser；采用歧义分析算法优化查询关键字的搜索排列组合，能极大的提高Lucene检索的命中率。

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

在没有使用IK分词器时，默认是standard方式分词，这种方式分词就是一个字就是一个词。

而IK分词器有两种分词模式：ik\_max\_word和ik\_smart模式。

1、\*\*ik\_max\_word\*\*

会将文本做最细粒度的拆分，比如会将“乒乓球明年总冠军”拆分为“乒乓球、乒乓、球、明年、总冠军、冠军。

#方式一ik\_max\_word

GET /\_analyze

{

"**analyzer**": "ik\_max\_word",

"**text**": "乒乓球明年总冠军"

}

ik\_max\_word分词器执行如下：

{

"**tokens**" : [

{

"**token**" : "乒乓球",

"**start\_offset**" : 0,

"**end\_offset**" : 3,

"**type**" : "CN\_WORD",

"**position**" : 0

},

{

"**token**" : "乒乓",

"**start\_offset**" : 0,

"**end\_offset**" : 2,

"**type**" : "CN\_WORD",

"**position**" : 1

},

{

"**token**" : "球",

"**start\_offset**" : 2,

"**end\_offset**" : 3,

"**type**" : "CN\_CHAR",

"**position**" : 2

},

{

"**token**" : "明年",

"**start\_offset**" : 3,

"**end\_offset**" : 5,

"**type**" : "CN\_WORD",

"**position**" : 3

},

{

"**token**" : "总冠军",

"**start\_offset**" : 5,

"**end\_offset**" : 8,

"**type**" : "CN\_WORD",

"**position**" : 4

},

{

"**token**" : "冠军",

"**start\_offset**" : 6,

"**end\_offset**" : 8,

"**type**" : "CN\_WORD",

"**position**" : 5

}

]

}

2、\*\*ik\_smart\*\*  
会做粗粒度的拆分，比如会将“乒乓球明年总冠军”拆分为乒乓球、明年、总冠军。

#方式二ik\_smart

GET /\_analyze

{

"**analyzer**": "ik\_smart",

"**text**": "乒乓球明年总冠军"

}

ik\_smart分词器执行如下：

{

"**tokens**" : [

{

"**token**" : "乒乓球",

"**start\_offset**" : 0,

"**end\_offset**" : 3,

"**type**" : "CN\_WORD",

"**position**" : 0

},

{

"**token**" : "明年",

"**start\_offset**" : 3,

"**end\_offset**" : 5,

"**type**" : "CN\_WORD",

"**position**" : 1

},

{

"**token**" : "总冠军",

"**start\_offset**" : 5,

"**end\_offset**" : 8,

"**type**" : "CN\_WORD",

"**position**" : 2

}

]

}

# 第三章 ElasticSearch相关概念(术语)

## 3.1 概述

先说Elasticsearch的文件存储，Elasticsearch是面向文档型数据库，一条数据在这里就是一个文档，用JSON作为文档序列化的格式，比如下面这条用户数据：

{

"**name**" : "jack",

"**sex**" : "Male",

"**age**" : 25,

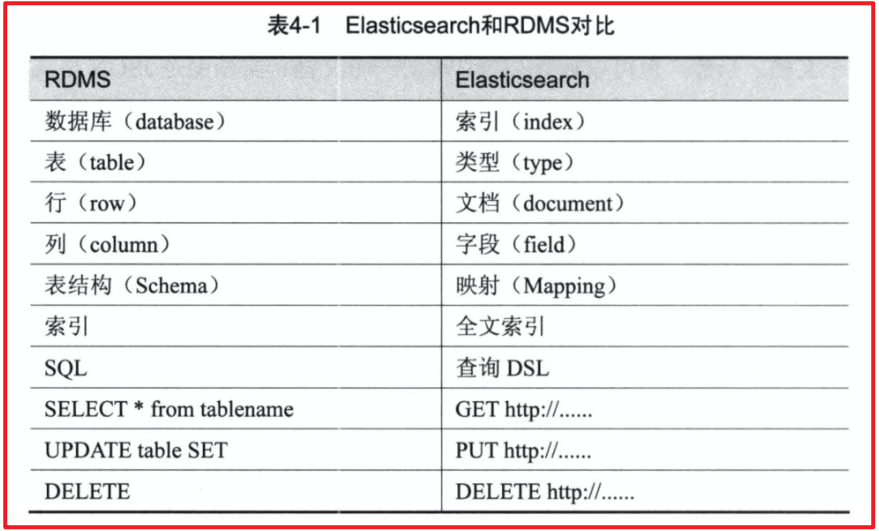
"**birthDate**": "1990/05/01",

"**about**" : "I love to go rock climbing",

"**interests**": [ "sports", "music" ]

}

Elasticsearch可以看成是一个数据库，只是和关系型数据库比起来数据格式和功能不一样而已



## 3.2 ****Elasticsearch核心概念****

### 3.2.1 索引 index

文档存储的地方，类似于MySQL数据库中的数据库概念

### 3.2.2 ~~类型 type~~

如果按照关系型数据库中的对应关系，还应该有表的概念。ES中没有表的概念，这是ES和数据库的一个区别，在我们建立\*索引\*之后，可以直接往 索引 中写入\*文档\*。

在6.0版本之前，ES中有Type的概念，可以理解成关系型数据库中的表，但是官方说这是一个设计上的失误，所以在6.0版本之后Type就被废弃了。

### 3.2.3 字段Field

相当于是数据表的字段，字段在ES中可以理解为JSON数据的键，下面的JSON数据中，\*name\* 就是一个字段。

{

"**name**":"jack"

}

### 3.2.4 映射 mapping

映射 是对\*文档\*中每个\*字段\*的类型进行定义，每一种数据类型都有对应的使用场景。

每个\*文档\*都有映射，但是在大多数使用场景中，我们并不需要显示的创建映射，因为**ES中实现了动态映射**。

我们在索引中写入一个下面的JSON文档，在动态映射的作用下，name会映射成text类型，age会映射成long类型。

{

"**name**":"jack",

"**age**":18,

}

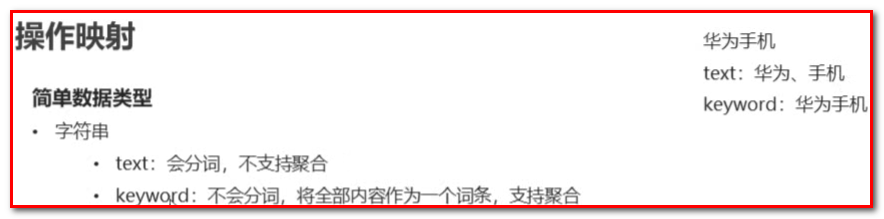
自动判断的规则如下：

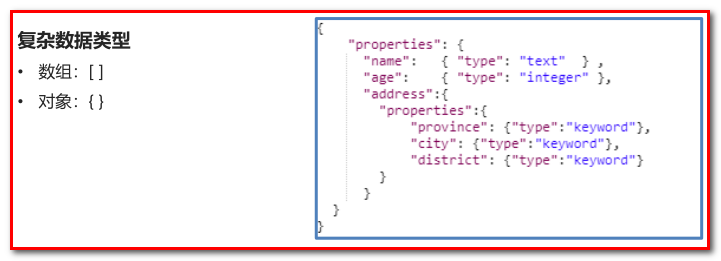


Elasticsearch中支持的类型如下：



* string类型在ElasticSearch 旧版本中使用较多，从ElasticSearch 5.x开始不再支持string，由text和keyword类型替代。（已经废弃）
* text 类型，需要分词设置text类型，比如Email内容、产品描述，应该使用text类型。
* keyword类型 ，不需要分词设置keyword类型，比如email地址、主机名、状态码和标签。





### 3.2.5 文档 document

文档 在ES中相当于传统数据库中的\*行\*的概念，ES中的数据都以JSON的形式来表示，在MySQL中插入一行数据和ES中插入一个JSON文档是一个意思。下面的JSON数据表示，一个包含3个字段的\*文档\*。

{

"**name**":"jack",

"**age**":18,

"**gender**":1

}

一个文档不只有数据。它还包含了元数据(metadata)——关于文档的信息。三个必须的元数据节点是：

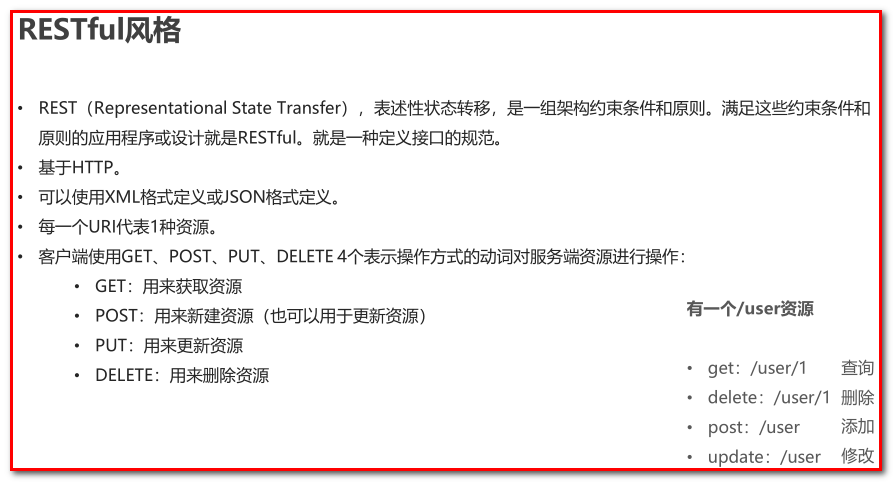
| **节点** | **说明** |
| --- | --- |
| \_index | 文档存储的地方 |
| \_type | 文档代表的对象的类 |
| \_id | 文档的唯一标识 |

# 第四章 ElasticSearch的客户端操作

实际开发中，主要有三种方式可以作为elasticsearch服务的客户端：

* 第一种，elasticsearch-head插件
* 第二种，使用elasticsearch提供的Restful接口直接访问
* 第三种，使用elasticsearch提供的API进行访问

## 4.1 ElasticSearch的接口语法



## 【\*\*\*\*\*\*\*Postman演示\*\*\*\*\*\*\*】

## 4.2 操作索引index

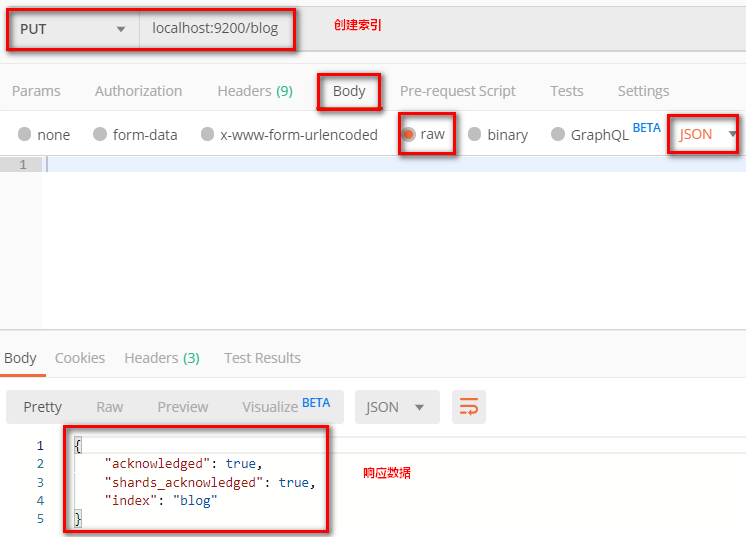


### 4.2.1 创建索引index

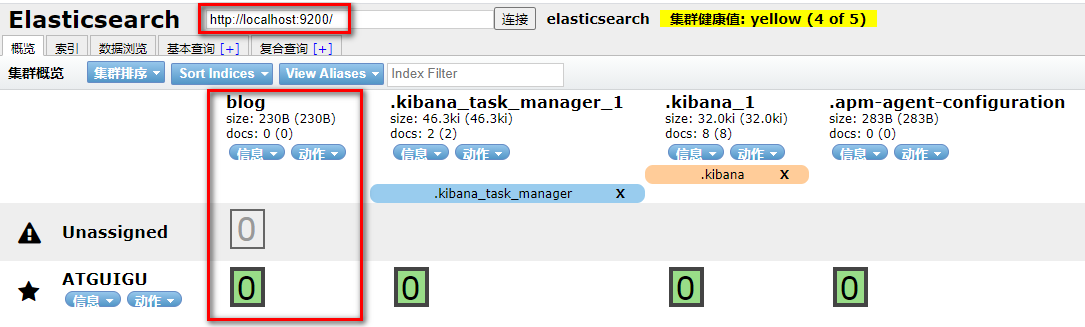
请求url：

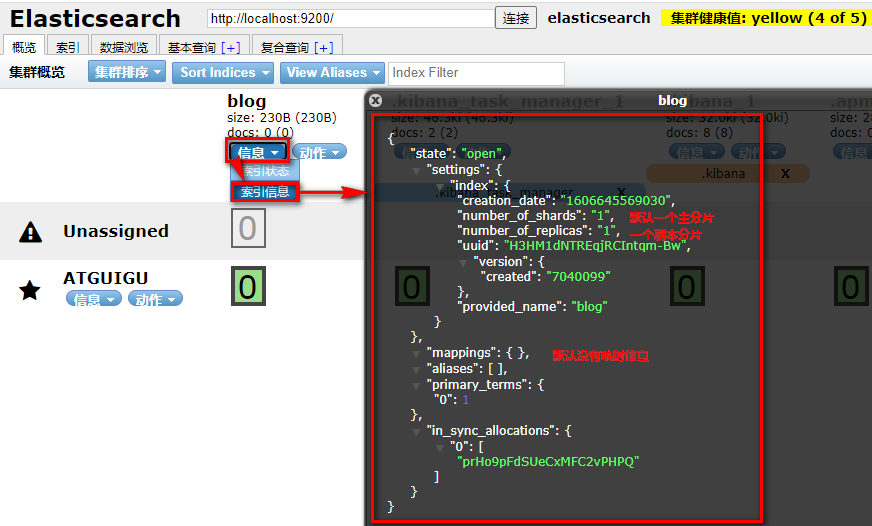
PUT localhost:9200/blog

发送请求，响应数据



elasticsearch-head查看：请求localhost:9200

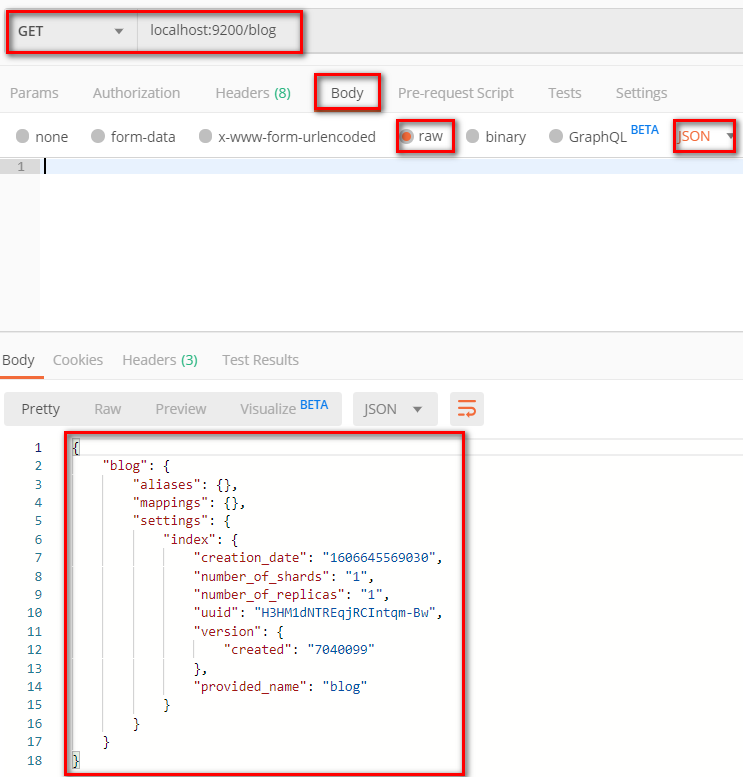




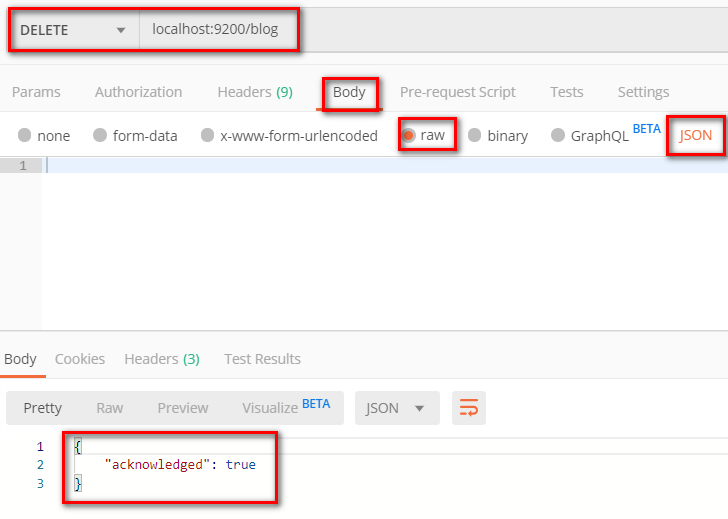
如果重复创建索引，会报错：



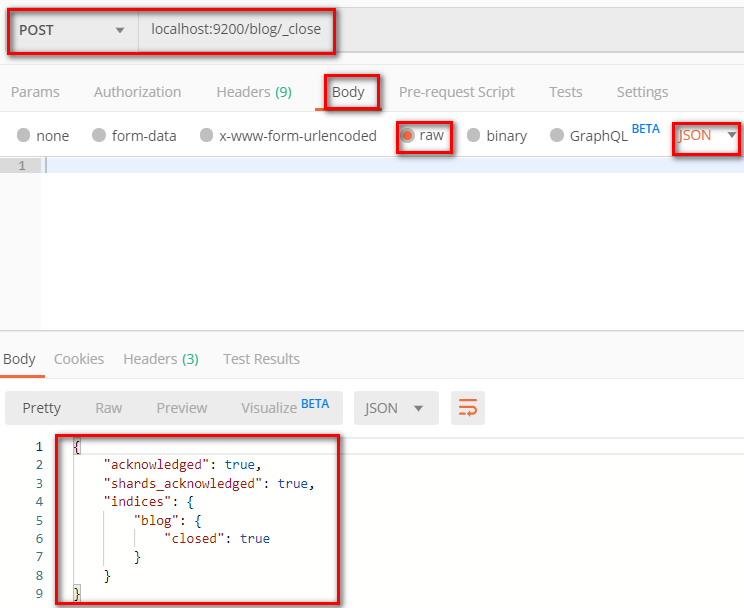
### 4.2.2 查看索引index



### 4.2.3 删除索引index

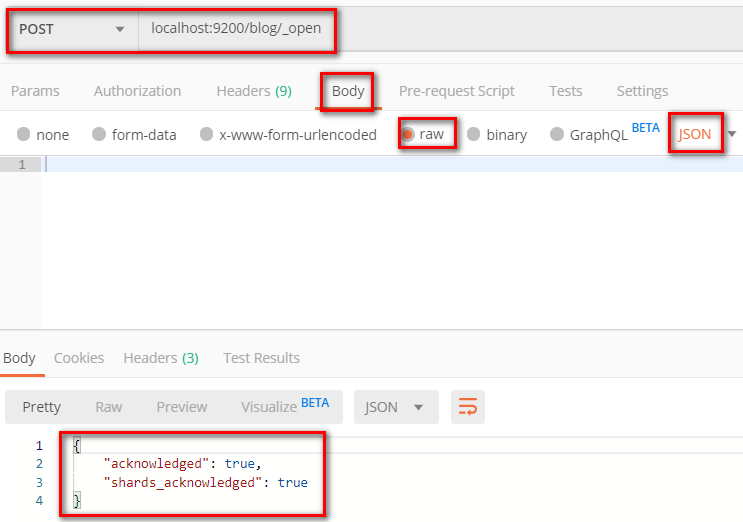


### 4.2.4 关闭索引index





### 4.2.5 打开索引index





## 4.3 创建索引index并且进行映射mapping

请求url：

PUT localhost:9200/blog1

请求体：

{

"**mappings**": {

"**properties**": {

"**id**": {

"**type**": "long",

"**store**": true,

"**index**":true

},

"**title**": {

"**type**": "text",

"**store**": true,

"**index**":true,

"**analyzer**":"standard"

},

"**content**": {

"**type**": "text",

"**store**": true,

"**index**":true,

"**analyzer**":"standard"

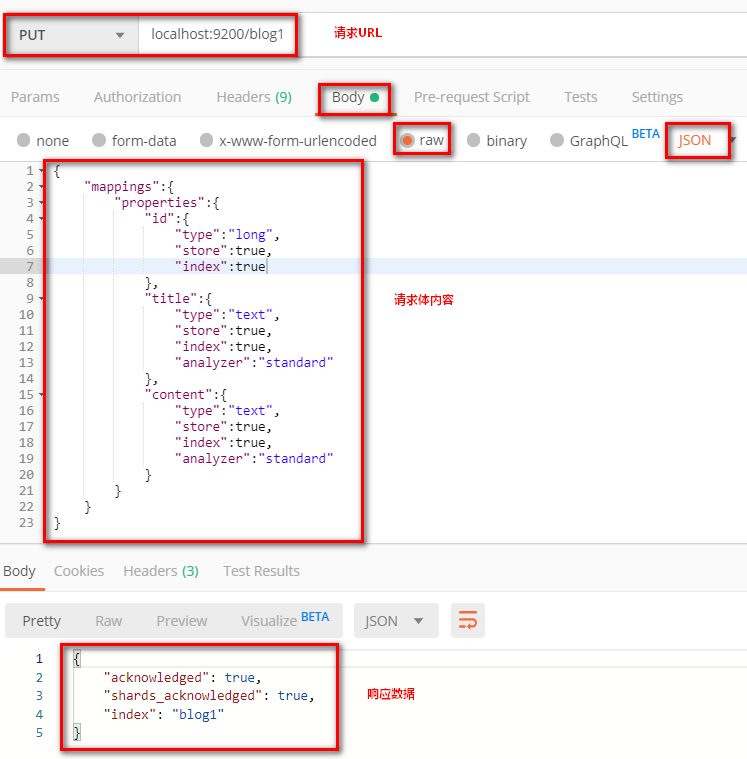
}

}

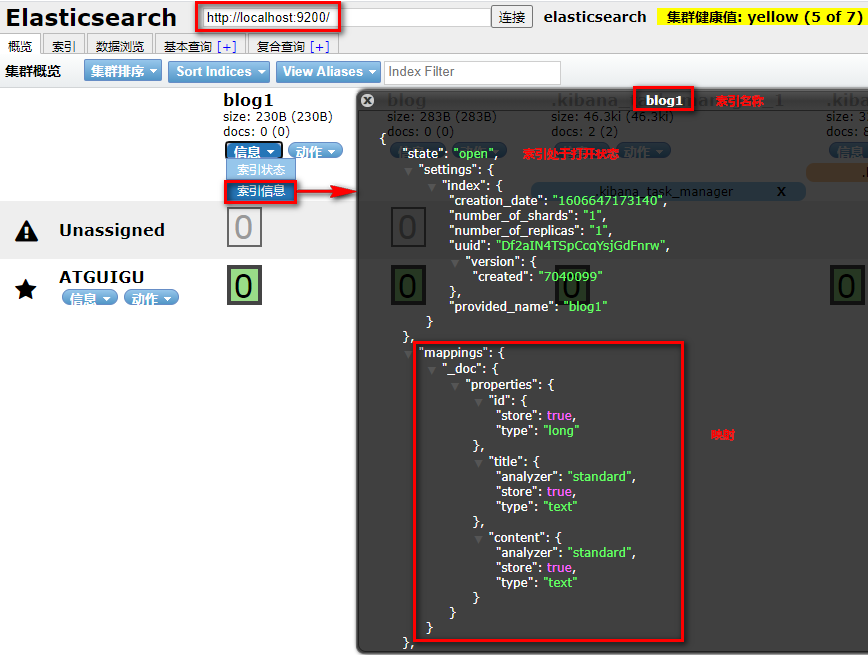
}

}

postman截图：



elasticsearch-head查看：



## 【\*\*\*\*\*\*\* kibana演示\*\*\*\*\*\*\*】

## 4.4 操作映射

# 创建索引

PUT person

# 查询索引

GET person

# 删除索引

DELETE person

# 查询映射

GET person/\_mapping

# 添加映射

PUT person/\_mapping

{

"properties":{

"name":{

"type":"keyword"

},

"age":{

"type":"integer"

}

}

}

# ------------------------------------------------

# 创建索引并添加映射

PUT person

{

"mappings": {

"properties": {

"name":{

"type": "keyword"

},

"age":{

"type":"integer"

}

}

}

}

# 索引库中添加字段

PUT person/\_mapping

{

"properties":{

"address":{

"type":"text"

}

}

}

## 4.5 操作文档document【kibana演示】

• 添加文档  
• 查询文档  
• 修改文档  
• 删除文档

请求url：

# 查询索引

GET person

# 添加文档，指定id

PUT person/\_doc/1

{

"name":"张三",

"age":20,

"address":"深圳宝安区"

}

# 查询文档

GET person/\_doc/1

# 添加文档,不指定id

POST person/\_doc/

{

"name":"李四",

"age":20,

"address":"深圳南山区"

}

# 查询文档

GET person/\_doc/u8b2QHUBCR3n8iTZ8-Vk

# 添加文档,不指定id

POST person/\_doc/

{

"name":"李四",

"age":20,

"address":"深圳南山区"

}

# 查询文档

GET person/\_doc/u8b2QHUBCR3n8iTZ8-Vk

# 查询所有文档

GET person/\_search

# 删除文档

DELETE person/\_doc/1

# 修改文档 根据id，id存在就是修改，id不存在就是添加

PUT person/\_doc/2

{

"name":"硅谷",

"age":20,

"address":"深圳福田保税区"

}

## 4.6 查询全部

请求url：

**GET** person/\_search

## 4.7 全文查询-match查询

全文查询会分析查询条件，先将查询条件进行分词，然后查询，求并集

请求体：

# match 先会对查询的字符串进行分词，在查询，求交集

GET person/\_search

{

"**query**": {

"**match**": {

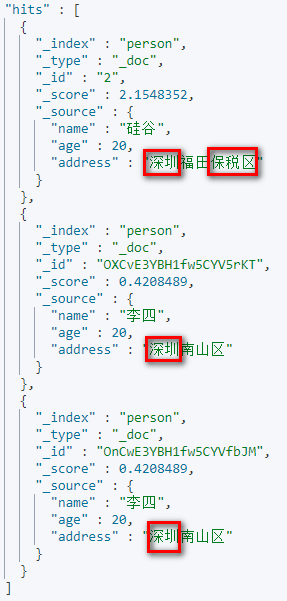
"**address**": "深圳保税区"

}

}

}

截图：



## 4.8 查询文档-term查询

词条查询不会分析查询条件，只有当词条和查询字符串完全匹配时才匹配搜索

请求url：

GET person/\_search

请求体：

# 查询所有数据

GET person/\_search

# 查询 带某词条的数据

GET person/\_search

{

"query": {

"term": {

"address": {

"value": "深圳南山区"

}

}

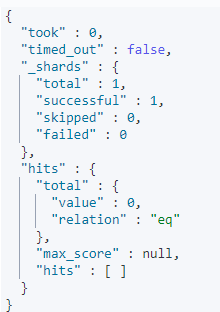
}

}

截图：

这个结果与使用的分词器有关。根据address字段，建立倒排索引时，需要对其分词，产生多个词条，而词条集合中没有"深圳南山区"的词条，故而查询不到数据。

大家可以查询“深”或“南山区”或“深圳”试试。



## 4.9 关键字搜索数据

请求url：

# 查询名字等于张三的用户

GET person/\_search?q=name:张三

截图：



## 4.10 DSL 查询

url地址请求体，多添加几条数据，方便做查询

PUT shangguigu/\_doc/1001

{

"id":"1001",

"name":"张三",

"age":20,

"sex":"男"

}

PUT shangguigu/\_doc/1002

{

"id":"1002",

"name":"李四",

"age":25,

"sex":"女"

}

PUT shangguigu/\_doc/1003

{

"id":"1003",

"name":"王五",

"age":30,

"sex":"女"

}

PUT shangguigu/\_doc/1004

{

"id":"1004",

"name":"赵六",

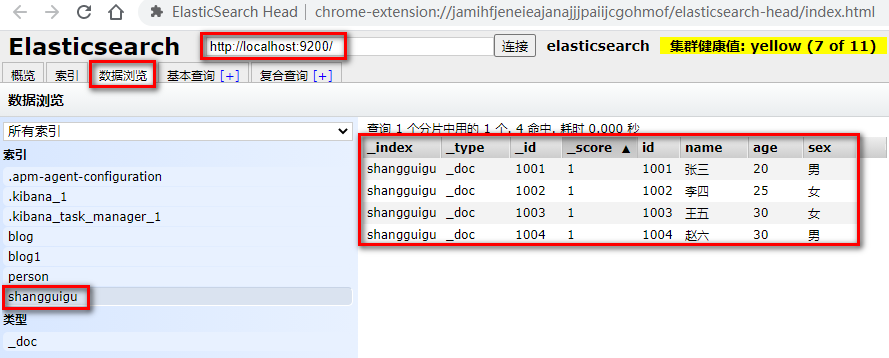
"age":30,

"sex":"男"

}

GET shangguigu/\_search

添加结果：



① 根据年龄查询

请求体

POST shangguigu/\_doc/\_search

{

"query":{

"match":{

"age":20

}

}

}

请求结果：



② 查询年龄大于20岁的女性用户

请求URL地址：

**GET** shangguigu/\_search

{

"query":{

"bool":{

"filter":{

"range":{

"age":{

"gt":20

}

}

},

"must":{

"match":{

"sex":"女"

}

}

}

}

}

相应结果：



③ 全文搜索

请求URL地址

**GET** shangguigu/\_search

{

"query":{

"match":{

"name": "张三 李四"

}

}

}

响应数据



## 4.11 高亮显示



请求URL地址

请求体

**GET** shangguigu/\_search

{

"query":{

"match":{

"name": "张三 李四"

}

},

"highlight": {

"fields": {

"name": {}

}

}

}



## 4.12 聚合

在Elasticsearch中，支持聚合操作，类似SQL中的group by操作。

请求URL地址

请求体

**GET** shangguigu/\_search

{

"aggs": {

"all\_interests": {

"terms": {

"field": "age"

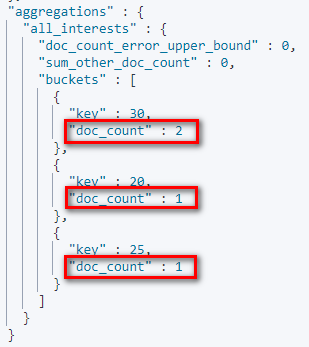
}

}

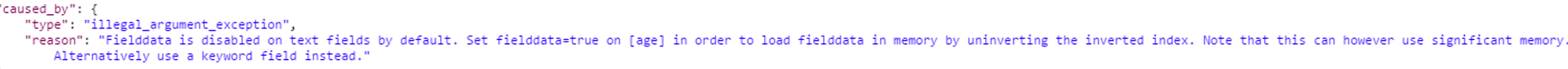
}

}

响应体



如果聚合查询报错：



修改下查询语句

请求体：

{

"**aggs**": {

"**all\_interests**": {

"**terms**": {

"**field**": "age.keyword"

}

}

}

}

## 4.13 指定响应字段

在响应的数据中，如果我们不需要全部的字段，可以指定某些需要的字段进行返回

GET shangguigu/\_doc/1001?\_source=id,name

等价于

GET /shangguigu/\_search

{

"query": {

"match": {

"id": "1001"

}

},

"\_source": ["id","name"]

}



## 4.14 判断文档是否存在

如果我们只需要判断文档是否存在，而不是查询文档内容，那么可以这样：

HEAD shangguigu/\_doc/1001

存在返回：200 - OK

不存在返回：404 – Not Found

## 4.15 批量操作

有些情况下可以通过批量操作以减少网络请求。如：批量查询、批量插入数据。

### 4.15.1 批量查询

请求体

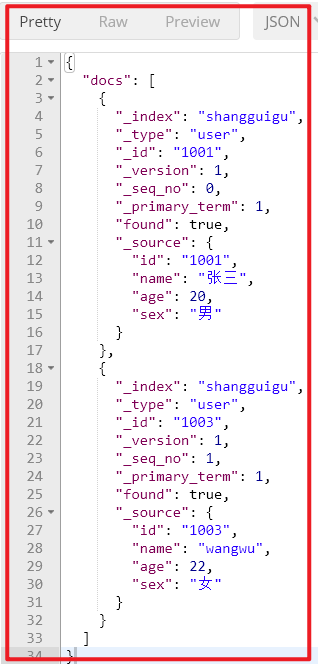
POST shangguigu/\_doc/\_mget

{

"ids" : [ "1001", "1003" ]

}

响应



如果，某一条数据不存在，不影响整体响应，需要通过found的值进行判断是否查询到数据。

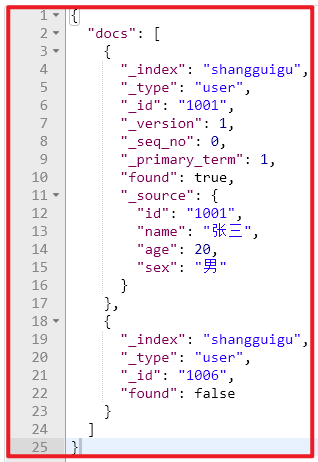
请求体：

{

"**ids**" : [ "1001", "1006" ]

}

响应



### 4.15.2 \_bulk操作

在Elasticsearch中，支持批量的插入、修改、删除操作，都是通过\_bulk的api完成的。  
请求格式如下：（请求格式不同寻常）

{ action: { metadata }}\n

{ request body    }\n

{ action: { metadata }}\n

{ request body    }\n

**...**

批量插入数据：

POST \_bulk

{"**create**":{"\_index":"atguigu","\_id":2001}}

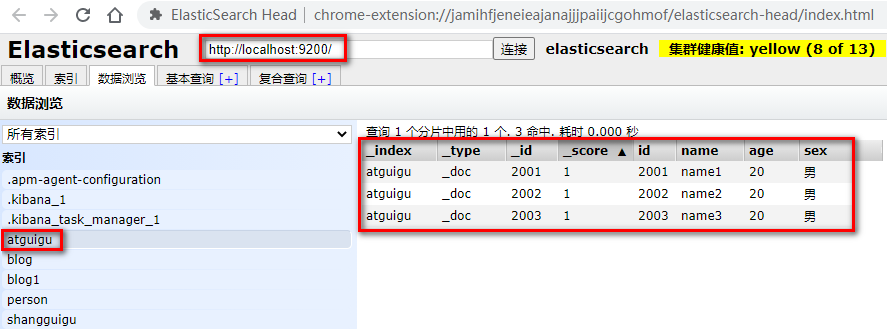
{"id":2001,"name":"name1","age": 20,"sex": "男"}

{"**create**":{"\_index":"atguigu","\_id":2002}}

{"id":2002,"name":"name2","age": 20,"sex": "男"}

{"**create**":{"\_index":"atguigu","\_id":2003}}

{"id":2003,"name":"name3","age": 20,"sex": "男"}



批量删除：

POST \_bulk

{"**delete**":{"\_index":"atguigu","\_id":2001}}

{"**delete**":{"\_index":"atguigu","\_id":2002}}

{"**delete**":{"\_index":"atguigu","\_id":2003}}



由于delete没有请求体，所以，action的下一行直接就是下一个action。

## 4.16 分页

和SQL使用 LIMIT 关键字返回只有一页的结果一样，Elasticsearch接受 from 和 size 参数：

**size**: 结果数，默认10

**from**: 跳过开始的结果数，默认0

如果你想每页显示5个结果，页码从1到3，那请求如下：

GET /\_search?size=5

GET /\_search?size=5&from=5

GET /\_search?size=5&from=10

请求路径

GET shangguigu/\_search?size=1&from=2

响应体



POST atguigu/\_bulk

{"index":{"\_index":"atguigu"}}

{"name":"张三","age": 20,"mail": "111@qq.com","hobby":"羽毛球、乒乓球、足球"}

{"index":{"\_index":"atguigu"}}

{"name":"李四","age": 21,"mail": "222@qq.com","hobby":"羽毛球、乒乓球、足球、篮球"}

{"index":{"\_index":"atguigu"}}

{"name":"王五","age": 22,"mail": "333@qq.com","hobby":"羽毛球、篮球、游泳、听音乐"}

{"index":{"\_index":"atguigu"}}

{"name":"赵六","age": 23,"mail": "444@qq.com","hobby":"跑步、游泳"}

{"index":{"\_index":"atguigu"}}

{"name":"孙七","age": 24,"mail": "555@qq.com","hobby":"听音乐、看电影"}

测试搜索：

POST atguigu/\_search

{

"query" : {

"match" : {

"hobby" : "音乐 羽毛球"

}

},

"from": 0,

"size": 2

}

## 4.17 terms查询

terms 跟 term 有点类似，但 terms 允许指定多个匹配条件。 如果某个字段指定了多个值，那么文档需要一起去做匹配：

POST atguigu/\_search

{

"query" : {

"terms" : {

"age" : [20,21]

}

}

}

## 4.18 range查询

range 过滤允许我们按照指定范围查找一批数据：

范围操作符包含：

gt :: 大于

gte :: 大于等于

lt :: 小于

lte :: 小于等于

示例

POST atguigu/\_search

{

"query": {

"range": {

"age": {

"gte": 20,

"lte": 22

}

}

}

}

## 4.19 exists 查询

exists 查询可以用于查找文档中是否包含指定字段或没有某个字段，类似于SQL语句中的 IS\_NULL 条件

# "exists": 必须包含

POST atguigu/\_search

{

"query": {

"exists": {

"field": "mail"

}

}

}

# 第五章 ElasticSearch集群

## 5.1【相关概念】

### 5.1.1 单节点故障问题

* 单台服务器，往往都有最大的负载能力，超过这个阈值，服务器性能就会大大降低甚至不可用。单点的elasticsearch也是一样，那单点的es服务器存在哪些可能出现的问题呢？
  + - 单台机器存储容量有限
    - 单服务器容易出现单点故障，无法实现高可用
    - 单服务的并发处理能力有限
* 所以，为了应对这些问题，我们需要对elasticsearch搭建集群
* 集群中节点数量没有限制，大于等于2个节点就可以看做是集群了。一般出于高性能及高可用方面来考虑集群中节点数量都是3个以上。

### 5.1.2 集群的相关概念

1. **集群 cluster**

一个集群就是由一个或多个节点组织在一起，它们共同持有整个的数据，并一起提供索引和搜索功能。一个集群由一个唯一的名字标识，这个名字默认就是“elasticsearch”。这个名字是重要的，因为一个节点只能通过指定某个集群的名字，来加入这个集群。

1. **节点 node**

一个节点是集群中的一个服务器，作为集群的一部分，它存储数据，参与集群的索引和搜索功能。

一个节点也是由一个名字来标识的，默认情况下，这个名字是一个随机的漫威漫画角色的名字，这个名字会在启动的时候赋予节点。这个名字对于管理工作来说挺重要的，因为在这个管理过程中，你会去确定网络中的哪些服务器对应于ElasticSearch集群中的哪些节点。

一个节点可以通过配置集群名称的方式来加入一个指定的集群。默认情况下，每个节点都会被安排加入到一个叫做“elasticsearch”的集群中，这意味着，如果你在你的网络中启动了若干个节点，并假定它们能够相互发现彼此，它们将会自动地形成并加入到一个叫做“elasticsearch”的集群中。

在一个集群里，只要你想，可以拥有任意多个节点。而且，如果当前你的网络中没有运行任何Elasticsearch节点，这时启动一个节点，会默认创建并加入一个叫做“elasticsearch”的集群。

1. **分片和复制 shards&replicas**

一个索引可以存储超出单个节点硬件限制的大量数据。比如，一个具有10亿文档的索引占据1TB的磁盘空间，而任一节点都没有这样大的磁盘空间；或者单个节点处理搜索请求，响应太慢。为了解决这个问题，ElasticSearch提供了将索引划分成多份的能力，这些份就叫做分片。当你创建一个索引的时候，你可以指定你想要的分片的数量。每个分片本身也是一个功能完善并且独立的“索引”，这个“索引”可以被放置到集群中的任何节点上。

分片很重要，主要有两方面的原因：

1）允许你水平分割/扩展你的内容容量。

2）允许你在分片（潜在地，位于多个节点上）之上进行分布式的、并行的操作，进而提高性能/吞吐量。

至于一个分片怎样分布，它的文档怎样聚合回搜索请求，是完全由ElasticSearch管理的，对于作为用户的你来说，这些都是透明的。

在一个网络/云的环境里，失败随时都可能发生，在某个分片/节点不知怎么的就处于离线状态，或者由于任何原因消失了，这种情况下，有一个故障转移机制是非常有用并且是强烈推荐的。为此目的，ElasticSearch允许你创建分片的一份或多份拷贝，这些拷贝叫做复制分片( 副本)。

复制之所以重要，有两个主要原因： 在分片/节点失败的情况下，提供了高可用性。因为这个原因，注意到复制分片从不与原/主要（original/primary）分片置于同一节点上是非常重要的。扩展你的搜索量/吞吐量，因为搜索可以在所有的复制上并行运行。总之，每个索引可以被分成多个分片。一个索引也可以被复制0次（意思是没有复制）或多次。一旦复制了，每个索引就有了主分片（作为复制源的原来的分片）和复制分片（主分片的拷贝）之别。分片和复制的数量可以在索引创建的时候指定。在索引创建之后，你可以在任何时候动态地改变复制的数量，但是你事后不能改变分片的数量。

默认情况下:

Elasticsearch6.x中的每个索引被分片5个主分片和1个复制，这意味着，如果你的集群中至少有两个节点，你的索引将会有5个主分片和另外5个复制分片（1个完全拷贝），这样的话每个索引总共就有10个分片。

Elasticsearch7.x中的每个索引被分片1个主分片和1个复制，这意味着，如果你的集群中至少有两个节点，你的索引将会有1个主分片和另外1个复制分片（1个完全拷贝），这样的话每个索引总共就有2个分片。

## 5.2【集群搭建】

### 5.2.1 准备三台elasticsearch服务器

创建elasticsearch-cluster文件夹，在内部复制三个elasticsearch服务

计算机生成了可选文字:
工作（D:),Server》
名称
node9301
日node9302
口node93o3
elasticsearc卜cluster》

### 5.2.2 修改每台服务器配置

修改elasticsearch-cluster\node\*\config\elasticsearch.yml配置文件

node1节点：

|  |
| --- |
| #节点1的配置信息： #集群名称，保证唯一 cluster.name: my-elasticsearch  #默认为true。设置为false禁用磁盘分配决定器。  cluster.routing.allocation.disk.threshold\_enabled: false  #节点名称，必须不一样 node.name: node-1 #必须为本机的ip地址 network.host: 127.0.0.1 #服务端口号，在同一机器下必须不一样 http.port: 9201 #集群间通信端口号，在同一机器下必须不一样 transport.tcp.port: 9301 #设置集群自动发现机器ip集合 #discovery.zen.ping.unicast.hosts: ["127.0.0.1:9301","127.0.0.1:9302","127.0.0.1:9303"]  #（新）  # 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  # es7之后，不需要上面discovery.zen.ping.unicast.hosts这个参数，用discovery.seed\_hosts替换  # discovery.zen.ping.unicast.hosts: ["10.19.1.9:9200","10.19.1.10:9200","10.19.1.11:9200"]  discovery.seed\_hosts: ["127.0.0.1:9301","127.0.0.1:9302","127.0.0.1:9303"]  # 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: ["node-1"] |

node2节点：

|  |
| --- |
| #节点1的配置信息： #集群名称，保证唯一 cluster.name: my-elasticsearch  #默认为true。设置为false禁用磁盘分配决定器。  cluster.routing.allocation.disk.threshold\_enabled: false  #节点名称，必须不一样 node.name: node-2 #必须为本机的ip地址 network.host: 127.0.0.1 #服务端口号，在同一机器下必须不一样 http.port: 9202 #集群间通信端口号，在同一机器下必须不一样 transport.tcp.port: 9302 #设置集群自动发现机器ip集合 #discovery.zen.ping.unicast.hosts: ["127.0.0.1:9301","127.0.0.1:9302","127.0.0.1:9303"]  #（新）  # 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  # es7之后，不需要上面discovery.zen.ping.unicast.hosts这个参数，用discovery.seed\_hosts替换  # discovery.zen.ping.unicast.hosts: ["10.19.1.9:9200","10.19.1.10:9200","10.19.1.11:9200"]  discovery.seed\_hosts: ["127.0.0.1:9301","127.0.0.1:9302","127.0.0.1:9303"]  # 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: ["node-1"] |

node3节点：

|  |
| --- |
| #节点1的配置信息： #集群名称，保证唯一 cluster.name: my-elasticsearch  #默认为true。设置为false禁用磁盘分配决定器。  cluster.routing.allocation.disk.threshold\_enabled: false  #节点名称，必须不一样 node.name: node-3 #必须为本机的ip地址 network.host: 127.0.0.1 #服务端口号，在同一机器下必须不一样 http.port: 9203 #集群间通信端口号，在同一机器下必须不一样 transport.tcp.port: 9303 #设置集群自动发现机器ip集合 #discovery.zen.ping.unicast.hosts: ["127.0.0.1:9301","127.0.0.1:9302","127.0.0.1:9303"]  #（新）  # 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  # es7之后，不需要上面discovery.zen.ping.unicast.hosts这个参数，用discovery.seed\_hosts替换  # discovery.zen.ping.unicast.hosts: ["10.19.1.9:9200","10.19.1.10:9200","10.19.1.11:9200"]  discovery.seed\_hosts: ["127.0.0.1:9301","127.0.0.1:9302","127.0.0.1:9303"]  # 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: ["node-1"] |

### 5.2.3 启动各个节点服务器

先清理掉之前数据：删除elasticsearch-cluster\node\*\data目录下的nodes目录

双击elasticsearch-cluster\node\*\bin\elasticsearch.bat

启动节点1：

启动节点2：

启动节点3：

## 5.3【集群测试】

### 5.3.1 安装ES插件ElasticSearch-head

在Chrome浏览器地址栏中输入：<chrome://extensions/>**，或按照下图打开“扩展程序”**

计算机生成了可选文字:
q☆
打开新的标签页m
打开新的窗口（N)
打开新的无痕窗口（l}
口月口
TNN
+++
州rI仕
右刀解卜』Ln
J月曰尸rLj口
Ct「
历史记录（H)
下我内容（D)
书签（B)
缩放
打印（P）…
Ctrl+J
Ctrl+p
网页另存为（A）…
创建快捷方式…
Ctrl+S
剪切m
复制（C)
粘贴（p)
清除涎览数据（C）…
帮助（E)
任务管理器m
Shi任＋Esc
退出（x〕
开发者工具（D)
Ctrl+Shi任＋l

将课件中【ElasticSearch-head-Chrome-0.1.5-Crx4Chrome.crx】文件拖到扩展程序页面上即可。

计算机生成了可选文字:
☆口侣e三
q开发者模式口0
加载已解压的扩展程序
打包扩展程序
更新
召
ElastiCSearChHead0.1.5
ElastiCSearChHesd
FeHelper（前端助韵2020.4.110
JsoN白动格式化、手动格式化，支持排序、解
码、下载等，更多功能可在配置页按需安装！
旧：什mkiejjmecolp月oofnjologoblkegm
查看视图背景页另外还有1个…
旧：pkgcCpejnmalmdinmhkkfafefagiiiad
查看视图background/index.html另外还有1个…
详细信息
删除
口0
删除
口0

### 5.3.2 使用elasticsearch-head查看集群情况

没有创建任何索引前的情况，只有一个默认的索引.kibana

计算机生成了可选文字:
Iael。。icse。rc卜卜ead
令令O合
X+
口
X
弃ElasticsearchHead}chrome一extension:
E1asticsearch
httplllocalhost:92011
刀什m幻幼mecolp月oofPjologoblkegm/elasticsearch一head/index.html☆
嚷习mv一el。stiCse。rch口口口口口口口口口口口口
概览索弓
集群概览
数据岚览
基二里犷甸［＋」
尖
复台里犷甸【＋l
＿舀石奋目‘ndeX「“,e「
口么e:
匕d
习
.kibana
size:5.10ki(10.2ki)
docs:1(2)
碟挡立暇棍D
回回
node一1
心霞翔口福不贾马
node一2
心霞翔口侣不资口
node一3
口
★：

### 5.3.3 集群测试

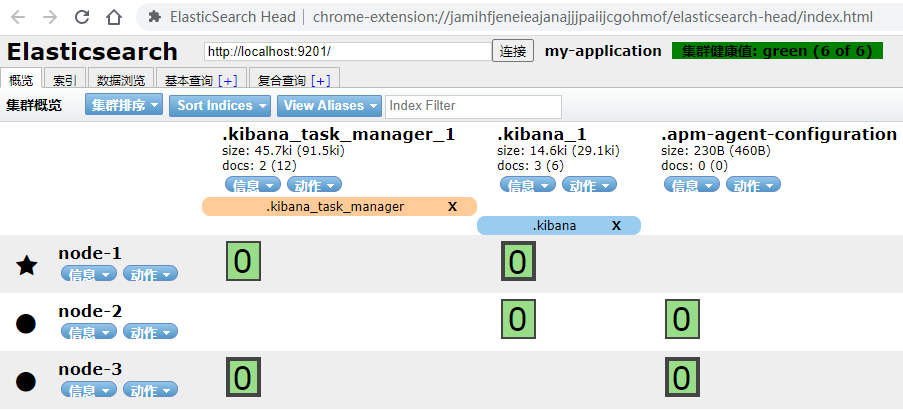
|  |
| --- |
| **创建索引及映射** |
| # 请求方法：PUT  **PUT /shopping**  {  "settings": {},  "mappings": {  "properties": {  "**title**":{  "type": "text",  "analyzer": "ik\_max\_word"    },  "**subtitle**":{  "type": "text",  "analyzer": "ik\_max\_word"  },  "**images**":{  "type": "keyword",  "index": false  },  "**price**":{  "type": "float",  "index": true  }  }  }  } |
| 添加文档 |
| **POST /shopping/product**  {  "title":"小米手机",  "images":"http://www.gulixueyuan.com/xm.jpg",  "price":3999.00  } |

### 5.3.4 再次使用elasticsearch-head查看集群情况

命令查看：

|  |
| --- |
| **GET \_cluster/health** |

Elasticsearch-head查看：



服务器运行状态：

* **Green**

所有的主分片和副本分片都已分配。你的集群是 100% 可用的。

* **yellow**

所有的主分片已经分片了，但至少还有一个副本是缺失的。不会有数据丢失，所以搜索结果依然是完整的。不过，你的高可用性在某种程度上被弱化。如果 *更多的* 分片消失，你就会丢数据了。把 yellow 想象成一个需要及时调查的警告。

* **red**

至少一个主分片（以及它的全部副本）都在缺失中。这意味着你在缺少数据：搜索只能返回部分数据，而分配到这个分片上的写入请求会返回一个异常。

# 第六章 高级客户端

## 6.1 搭建测试环境

创建项目 elasticsearch-demo

导入pom文件

<**parent**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-parent</**artifactId**>

<**version**>2.2.2.RELEASE</**version**>

<**relativePath**/> <!-- lookup parent from repository -->

</**parent**>

<**properties**>

<**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>

<**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>

<**java.version**>1.8</**java.version**>

</**properties**>

<**dependencies**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-test</**artifactId**>

<**scope**>test</**scope**>

</**dependency**>

<!--引入es的坐标-->

<**dependency**>

<**groupId**>org.elasticsearch.client</**groupId**>

<**artifactId**>elasticsearch-rest-high-level-client</**artifactId**>

<**version**>7.4.0</**version**>

</**dependency**>

<**dependency**>

<**groupId**>org.elasticsearch.client</**groupId**>

<**artifactId**>elasticsearch-rest-client</**artifactId**>

<**version**>7.4.0</**version**>

</**dependency**>

<**dependency**>

<**groupId**>org.elasticsearch</**groupId**>

<**artifactId**>elasticsearch</**artifactId**>

<**version**>7.4.0</**version**>

</**dependency**>

<**dependency**>

<**groupId**>com.alibaba</**groupId**>

<**artifactId**>fastjson</**artifactId**>

<**version**>1.2.4</**version**>

</**dependency**>

</**dependencies**>

① 在 resource 文件夹下面创建 application.yml 文件

elasticsearch:

host: 127.0.0.1

port: 9200

② 创建启动类

**package** com.atguigu;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** **ElasticsearchDemoApplication** {

**public** **static** **void** **main**(String[] args) {

SpringApplication.run(ElasticsearchDemoApplication.class, args);

}

}

③ 创建 com.atguigu.config.ElasticSearchConfig

**package** com.atguigu.config;

**import** org.apache.http.HttpHost;

**import** org.elasticsearch.client.RestClient;

**import** org.elasticsearch.client.RestHighLevelClient;

**import** org.springframework.boot.context.properties.ConfigurationProperties;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

@Configuration

@ConfigurationProperties(prefix = "elasticsearch")

**public** **class** **ElasticSearchConfig** {

**private** String host;

**private** **int** port;

**public** String **getHost**() {

**return** host;

}

**public** **void** **setHost**(String host) {

**this**.host = host;

}

**public** **int** **getPort**() {

**return** port;

}

**public** **void** **setPort**(**int** port) {

**this**.port = port;

}

@Bean

**public** RestHighLevelClient **client**(){

**return** **new** **RestHighLevelClient**(RestClient.builder(**new** HttpHost(host,port,"http")));

}

}

④ 新建测试类

**package** com.atguigu.test;

**import** org.apache.http.HttpHost;

**import** org.elasticsearch.action.admin.indices.delete.DeleteIndexRequest;

**import** org.elasticsearch.action.delete.DeleteRequest;

**import** org.elasticsearch.action.delete.DeleteResponse;

**import** org.elasticsearch.action.get.GetRequest;

**import** org.elasticsearch.action.get.GetResponse;

**import** org.elasticsearch.action.index.IndexRequest;

**import** org.elasticsearch.action.index.IndexResponse;

**import** org.elasticsearch.action.support.master.AcknowledgedResponse;

**import** org.elasticsearch.client.\*;

**import** org.elasticsearch.client.indices.CreateIndexRequest;

**import** org.elasticsearch.client.indices.CreateIndexResponse;

**import** org.elasticsearch.client.indices.GetIndexRequest;

**import** org.elasticsearch.client.indices.GetIndexResponse;

**import** org.elasticsearch.cluster.metadata.MappingMetaData;

**import** org.elasticsearch.common.xcontent.XContentType;

**import** org.junit.After;

**import** org.junit.Before;

**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.HashMap;

**import** java.util.Map;

@RunWith(SpringRunner.class)

@SpringBootTest

**public** **class** **ElasticsearchTest** {

@Autowired

**private** RestHighLevelClient client;

@Test

**public** **void** **contextLoads**() {

System.out.println(client);

}

## 6.2 索引操作

### 6.2.1 创建索引

**import** org.apache.http.HttpHost;

**import** org.elasticsearch.action.admin.indices.delete.DeleteIndexRequest;

**import** org.elasticsearch.action.delete.DeleteRequest;

**import** org.elasticsearch.action.delete.DeleteResponse;

**import** org.elasticsearch.action.get.GetRequest;

**import** org.elasticsearch.action.get.GetResponse;

**import** org.elasticsearch.action.index.IndexRequest;

**import** org.elasticsearch.action.index.IndexResponse;

**import** org.elasticsearch.action.support.master.AcknowledgedResponse;

**import** org.elasticsearch.client.\*;

**import** org.elasticsearch.client.indices.CreateIndexRequest;

**import** org.elasticsearch.client.indices.CreateIndexResponse;

**import** org.elasticsearch.client.indices.GetIndexRequest;

**import** org.elasticsearch.client.indices.GetIndexResponse;

**import** org.elasticsearch.cluster.metadata.MappingMetaData;

**import** org.elasticsearch.common.xcontent.XContentType;

**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.HashMap;

**import** java.util.Map;

@RunWith(SpringRunner.class)

@SpringBootTest

**public** **class** **ElasticsearchTest** {

@Autowired

**private** RestHighLevelClient client;

/\*\*

\* 添加索引

\*/

@Test

**public** **void** **addIndex**() **throws** Exception {

//1.使用client获取操作索引的对象

IndicesClient indicesClient = client.indices();

//2.具体操作，获取返回值

CreateIndexRequest createRequest = **new** CreateIndexRequest("abc");

CreateIndexResponse response = indicesClient.create(createRequest, RequestOptions.DEFAULT);

//3.根据返回值判断结果

System.out.println(response.isAcknowledged());

}

kibana 查询



### 6.2.2 添加索引和映射

**package** com.atguigu.test;

@RunWith(SpringRunner.class)

@SpringBootTest

**public** **class** **ElasticsearchTest** {

@Autowired

**private** RestHighLevelClient client;

/\*\*

\* 添加索引

\*/

@Test

**public** **void** **addIndexAndMapping**() **throws** IOException {

//1.使用client获取操作索引的对象

IndicesClient indicesClient = client.indices();

//2.具体操作，获取返回值

CreateIndexRequest createRequest = **new** CreateIndexRequest("aaa");

//2.1 设置mappings

String mapping = "{\n" +

" \"properties\" : {\n" +

" \"address\" : {\n" +

" \"type\" : \"text\",\n" +

" \"analyzer\" : \"ik\_max\_word\"\n" +

" },\n" +

" \"age\" : {\n" +

" \"type\" : \"long\"\n" +

" },\n" +

" \"name\" : {\n" +

" \"type\" : \"keyword\"\n" +

" }\n" +

" }\n" +

" }";

createRequest.mapping(mapping,XContentType.JSON);

CreateIndexResponse response = indicesClient.create(createRequest, RequestOptions.DEFAULT);

//3.根据返回值判断结果

System.out.println(response.isAcknowledged());

}

}



### 6.2.3 查询索引

/\*\*

\* 查询索引

\*/

@Test

**public** **void** **queryIndex**() **throws** IOException {

IndicesClient indices = client.indices();

GetIndexRequest getReqeust = **new** GetIndexRequest("aaa");

GetIndexResponse response = indices.get(getReqeust, RequestOptions.DEFAULT);

//获取结果

Map<String, MappingMetaData> mappings = response.getMappings();

**for** (String key : mappings.keySet()) {

System.out.println(key+":" + mappings.get(key).getSourceAsMap());

}

}

### 6.2.4 删除索引

/\*\*

\* 删除索引

\*/

@Test

**public** **void** **deleteIndex**() **throws** IOException {

IndicesClient indices = client.indices();

DeleteIndexRequest deleteRequest = **new** DeleteIndexRequest("abc");

AcknowledgedResponse response = indices.delete(deleteRequest, RequestOptions.DEFAULT);

System.out.println(response.isAcknowledged());

}

### 6.2.5 判断索引是否存在

/\*\*

\* 判断索引是否存在

\*/

@Test

**public** **void** **existIndex**() **throws** IOException {

IndicesClient indices = client.indices();

GetIndexRequest getRequest = **new** GetIndexRequest("aaa");

**boolean** exists = indices.exists(getRequest, RequestOptions.DEFAULT);

System.out.println(exists);

}

## 6.3 文档操作

### 6.3.1 添加文档,使用map作为数据

/\*\*

\* 添加文档,使用map作为数据

\*/

@Test

**public** **void** **addDoc**() **throws** IOException {

//数据对象，map

Map data = **new** HashMap();

data.put("address","深圳宝安");

data.put("name","尚硅谷");

data.put("age",20);

//1.获取操作文档的对象

IndexRequest request = **new** IndexRequest("aaa").id("1").source(data);

//添加数据，获取结果

IndexResponse response = client.index(request, RequestOptions.DEFAULT);

//打印响应结果

System.out.println(response.getId());

}

创建 com.atguigu.domain.Person

**public** **class** **Person** {

**private** String id;

**private** String name;

**private** **int** age;

**private** String address;

// 设置 set get 和tostring方法

}

### 6.3.2 添加文档,使用对象作为数据

/\*\*

\* 添加文档,使用对象作为数据

\*/

@Test

**public** **void** **addDoc2**() **throws** IOException {

//数据对象，javaObject

Person p = **new** Person();

p.setId("2");

p.setName("硅谷2222");

p.setAge(30);

p.setAddress("北京昌平区");

//将对象转为json

String data = JSON.toJSONString(p);

//1.获取操作文档的对象

IndexRequest request = **new** IndexRequest("aaa").id(p.getId()).source(data,XContentType.JSON);

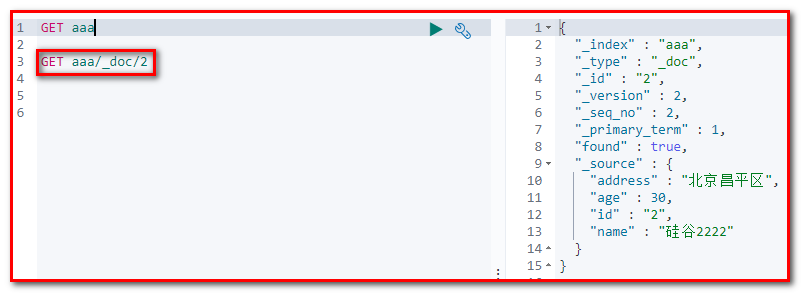
//添加数据，获取结果

IndexResponse response = client.index(request, RequestOptions.DEFAULT);

//打印响应结果

System.out.println(response.getId());

}



### 6.3.3 修改文档

/\*\*

\* 修改文档：添加文档时，如果id存在则修改，id不存在则添加

\*/

@Test

**public** **void** **updateDoc**() **throws** IOException {

//数据对象，javaObject

Person p = **new** Person();

p.setId("2");

p.setName("硅谷");

p.setAge(30);

p.setAddress("北京昌平区");

//将对象转为json

String data = JSON.toJSONString(p);

//1.获取操作文档的对象

IndexRequest request = **new** IndexRequest("aaa").id(p.getId()).source(data,XContentType.JSON);

//添加数据，获取结果

IndexResponse response = client.index(request, RequestOptions.DEFAULT);

//打印响应结果

System.out.println(response.getId());

}

### 6.3.4 根据id查询文档

/\*\*

\* 根据id查询文档

\*/

@Test

**public** **void** **findDocById**() **throws** IOException {

GetRequest getReqeust = **new** GetRequest("aaa","1");

//getReqeust.id("1");

GetResponse response = client.get(getReqeust, RequestOptions.DEFAULT);

//获取数据对应的json

System.out.println(response.getSourceAsString());

}

### 6.3.5 根据id删除文档

/\*\*

\* 根据id删除文档

\*/

@Test

**public** **void** **delDoc**() **throws** IOException {

DeleteRequest deleteRequest = **new** DeleteRequest("aaa","1");

DeleteResponse response = client.delete(deleteRequest, RequestOptions.DEFAULT);

System.out.println(response.getId());

System.out.println(response.getResult());

}

### 6.3.6 批量操作-脚本

Bulk 批量操作是将文档的增删改查一些列操作，通过一次请求全都做完。减少网络传输次数。



GET person/\_search

# 批量操作

# 1 删除1号记录

# 2 添加8号记录

# 3 修改2号记录 名称为二号

POST \_bulk

{"delete":{"\_index":"person","\_id":"1"}}

{"create":{"\_index":"person","\_id":"8"}}

{"name":"8号","age":80,"address":"北京"}

{"update":{"\_index":"person","\_id":"2"}}

{"doc":{"name":"2号"}}

/\*\*

\* 1. 批量操作 bulk

\*/

@Test

**public** **void** **testBulk**() **throws** IOException {

//创建bulkrequest对象，整合所有操作

BulkRequest bulkRequest = **new** BulkRequest();

/\*

1. 删除1号记录

2. 添加6号记录

3. 修改3号记录 名称为 “三号”

\*/

//添加对应操作

//1. 删除1号记录

DeleteRequest deleteRequest = **new** DeleteRequest("person","1");

bulkRequest.add(deleteRequest);

//2. 添加6号记录

Map map = **new** HashMap();

map.put("name","六号");

IndexRequest indexRequest = **new** IndexRequest("person").id("6").source(map);

bulkRequest.add(indexRequest);

Map map2 = **new** HashMap();

map2.put("name","三号");

//3. 修改3号记录 名称为 “三号”

UpdateRequest updateReqeust = **new** UpdateRequest("person","3").doc(map2);

bulkRequest.add(updateReqeust);

//执行批量操作

BulkResponse response = client.bulk(bulkRequest, RequestOptions.DEFAULT);

RestStatus status = response.status();

System.out.println(status);

}

## 6.4 批量导入MySQL到ES中

将数据库中Goods表的数据导入到ElasticSearch中

① 将数据库中Goods表的数据导入到ElasticSearch中

② 创建索引

PUT goods

{

"mappings": {

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_smart"

},

"price": {

"type": "double"

},

"createTime": {

"type": "date"

},

"categoryName": {

"type": "keyword"

},

"brandName": {

"type": "keyword"

},

"spec": {

"type": "object"

},

"saleNum": {

"type": "integer"

},

"stock": {

"type": "integer"

}

}

}

}

# 查询索引

GET goods

* title:商品标题
* price:商品价格
* createTime:创建时间
* categoryName:分类名称。如：家电，手机
* brandName:品牌名称。如：华为，小米
* spec: 商品规格。如： spec:{“屏幕尺寸”,“5寸”，“内存大小”,“128G”}
* saleNum:销量
* stock:库存量

添加文档数据

POST goods/\_doc/1

{

"title":"小米手机",

"price":1000,

"createTime":"2019-12-01",

"categoryName":"手机",

"brandName":"小米",

"saleNum":3000,

"stock":10000,

"spec":{

"网络制式":"移动4G",

"屏幕尺寸":"4.5"

}

}

# 查询文档数据

GET goods/\_search

添加依赖坐标

<!--mybatis-->

<**dependency**>

<**groupId**>org.mybatis.spring.boot</**groupId**>

<**artifactId**>mybatis-spring-boot-starter</**artifactId**>

<**version**>2.1.0</**version**>

</**dependency**>

<**dependency**>

<**groupId**>mysql</**groupId**>

<**artifactId**>mysql-connector-java</**artifactId**>

</**dependency**>

添加 application.yml 配置文件

# datasource

spring:

datasource:

url: jdbc:mysql:///es?serverTimezone=UTC

username: root

password: root

driver-**class**-**name**: **com**.**mysql**.**cj**.**jdbc**.**Driver**

# mybatis

mybatis:

mapper-locations: classpath:/mapper/\*Mapper.xml

type-aliases-package: com.atguigu.domain

添加 javabean

**package** com.atguigu.domain;

**import** com.alibaba.fastjson.annotation.JSONField;

**import** java.util.Date;

**import** java.util.Map;

**public** **class** **Goods** {

**private** **int** id;

**private** String title;

**private** **double** price;

**private** **int** stock;

**private** **int** saleNum;

**private** Date createTime;

**private** String categoryName;

**private** String brandName;

**private** Map spec;

// @JSONField(serialize = false)//在转换JSON时，忽略该字段

**private** String specStr;//接收数据库的信息 "{}"

// 生成set get 和 toString方法

}

创建 dao

**package** com.atguigu.mapper;

**import** com.atguigu.domain.Goods;

**import** org.apache.ibatis.annotations.Mapper;

**import** org.springframework.stereotype.Repository;

**import** java.util.List;

@Repository

@Mapper

**public** **interface** **GoodsMapper** {

**public** List<Goods> **findAll**();

}

在 resource 文件夹下面 创建 mapper/GoodsMapper.xml 配置文件

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<**mapper** **namespace**="com.atguigu.mapper.GoodsMapper">

<**select** **id**="findAll" **resultType**="goods">

select

`id`,

`title`,

`price`,

`stock`,

`saleNum`,

`createTime`,

`categoryName`,

`brandName`,

`spec` as specStr

from goods

</**select**>

</**mapper**>

添加测试方法

@RunWith(SpringRunner.class)

@SpringBootTest

**public** **class** **ElasticsearchTest2** {

@Autowired

**private** GoodsMapper goodsMapper;

@Autowired

**private** RestHighLevelClient client;

/\*\*

\* 批量导入

\*/

@Test

**public** **void** **importData**() **throws** IOException {

//1.查询所有数据，mysql

List<Goods> goodsList = goodsMapper.findAll();

//System.out.println(goodsList.size());

//2.bulk导入

BulkRequest bulkRequest = **new** BulkRequest();

//2.1 循环goodsList，创建IndexRequest添加数据

**for** (Goods goods : goodsList) {

//2.2 设置spec规格信息 Map的数据 specStr:{}

//goods.setSpec(JSON.parseObject(goods.getSpecStr(),Map.class));

String specStr = goods.getSpecStr();

//将json格式字符串转为Map集合

Map map = JSON.parseObject(specStr, Map.class);

//设置spec map

goods.setSpec(map);

//将goods对象转换为json字符串

String data = JSON.toJSONString(goods);//bean --> {}

IndexRequest indexRequest = **new** IndexRequest("goods");

indexRequest.id(goods.getId()+"").source(data, XContentType.JSON);

bulkRequest.add(indexRequest);

}

BulkResponse response = client.bulk(bulkRequest, RequestOptions.DEFAULT);

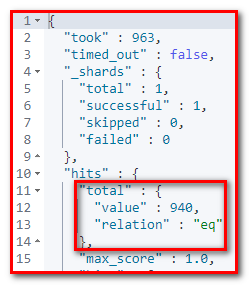
System.out.println(response.status());

}

}

查询数据是否导入

**GET** goods/\_search



## 6.5 查询操作

### 6.5.1 [查询所有matchAll查询](file:///D:\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\13.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_ElasticSearch\%E5%A4%87%E8%AF%BE\7.4\ElasticSearch%E7%AC%AC%E4%BA%8C%E5%A4%A9.html#查询所有matchall查询)

matchAll查询：查询所有文档

kibana 演示

# 查询

GET goods/\_search

{

"query": {

"match\_all": {}

},

"from": 0,

"size": 100

}

/\*\*

\* 查询所有

\* 1. matchAll

\* 2. 将查询结果封装为Goods对象，装载到List中

\* 3. 分页。默认显示10条

\*/

@Test

**public** **void** **testMatchAll**() **throws** IOException {

//2. 构建查询请求对象，指定查询的索引名称

SearchRequest searchRequest = **new** SearchRequest("goods");

//4. 创建查询条件构建器SearchSourceBuilder

SearchSourceBuilder sourceBuilder = **new** SearchSourceBuilder();

//6. 查询条件

QueryBuilder query = QueryBuilders.matchAllQuery();//查询所有文档

//5. 指定查询条件

sourceBuilder.query(query);

//3. 添加查询条件构建器 SearchSourceBuilder

searchRequest.source(sourceBuilder);

// 8 . 添加分页信息

sourceBuilder.from(0);

sourceBuilder.size(100);

//1. 查询,获取查询结果

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

//7. 获取命中对象 SearchHits

SearchHits searchHits = searchResponse.getHits();

//7.1 获取总记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

//7.2 获取Hits数据 数组

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

//获取json字符串格式的数据

String sourceAsString = hit.getSourceAsString();

//转为java对象

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

}

### 6.5.2 [term 查询](file:///D:\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\13.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_ElasticSearch\%E5%A4%87%E8%AF%BE\7.4\ElasticSearch%E7%AC%AC%E4%BA%8C%E5%A4%A9.html#term-查询)

term查询：不会对查询条件进行分词。

kibana 演示

GET goods

# term 查询

GET goods/\_search

{

"query": {

"term": {

"categoryName": {

"value": "手机"

}

}

}

}

/\*\*

\* termQuery:词条查询

\*/

@Test

**public** **void** **testTermQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

QueryBuilder query = QueryBuilders.termQuery("title","华为");//term词条查询

sourceBulider.query(query);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

}

### 6.5.3 matchQuery:词条分词查询

match查询：  
• 会对查询条件进行分词。  
• 然后将分词后的查询条件和词条进行等值匹配  
• 默认取并集（OR）



kibana 演示

# **match** 查询 "title": "手机"

GET goods/\_search

{

"query": {

"match": {

"title": "华为"

}

}

}

# **match** 查询 "operator": "or"

GET goods/\_search

{

"query": {

"match": {

"title": {

"query": "华为手机",

"operator": "and"

}

}

}

}

/\*\*

\* matchQuery:词条分词查询，分词之后的等值匹配

\*/

@Test

**public** **void** **testMatchQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

MatchQueryBuilder query = QueryBuilders.matchQuery("title", "华为手机");

query.operator(Operator.AND);//求并集

sourceBulider.query(query);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

}

### 6.5.4 模糊查询-脚本

wildcard查询：会对查询条件进行分词。还可以使用通配符 ?（任意单个字符） 和 \* （0个或多个字符）  
prefix查询：前缀查询

# wildcard 查询。查询条件分词，模糊查询 华为，华，\*华\*

GET goods/\_search

{

"query": {

"wildcard": {

"title": {

"value": "华\*"

}

}

}

}

# 前缀查询

GET goods/\_search

{

"query": {

"prefix": {

"brandName": {

"value": "三"

}

}

}

}

/\*\*

\* 模糊查询:WildcardQuery

\*/

@Test

**public** **void** **testWildcardQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

WildcardQueryBuilder query = QueryBuilders.wildcardQuery("title", "华\*");

sourceBulider.query(query);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

}

/\*\*

\* 模糊查询:perfixQuery

\*/

@Test

**public** **void** **testPrefixQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

PrefixQueryBuilder query = QueryBuilders.prefixQuery("brandName", "三");

sourceBulider.query(query);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

}

### 6.5.5 范围查询-脚本

range 范围查询：查找指定字段在指定范围内包含值

# 范围查询 gte 大于等于 lte小于等于

GET goods/\_search

{

"query": {

"range": {

"price": {

"gte": 2000,

"lte": 3000

}

}

}

}

# 范围查询 gte 大于等于 lte小于等于

GET goods/\_search

{

"query": {

"range": {

"price": {

"gte": 2000,

"lte": 3000

}

}

},

"sort": [

{

"price": {

"order": "desc"

}

}

]

}

/\*\*

\* 1. 范围查询：rangeQuery

\* 2. 排序

\*/

@Test

**public** **void** **testRangeQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

//范围查询

RangeQueryBuilder query = QueryBuilders.rangeQuery("price");

//指定下限 gte大于等于

query.gte(2000);

//指定上限 小于等于

query.lte(3000);

sourceBulider.query(query);

//排序

sourceBulider.sort("price", SortOrder.DESC);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

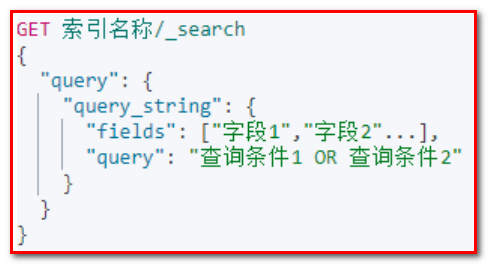
System.out.println(goods);

}

}

### 6.5.6 queryString查询-脚本

queryString：  
• 会对查询条件进行分词。  
• 然后将分词后的查询条件和词条进行等值匹配  
• 默认取并集（OR）  
• 可以指定多个查询字段



# queryString

GET goods/\_search

{

"query": {

"query\_string": {

"fields": ["title","categoryName","brandName"],

"query": "华为手机"

}

}

}

/\*\*

\* queryString

\*/

@Test

**public** **void** **testQueryStringQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

//queryString

QueryStringQueryBuilder query = QueryBuilders.queryStringQuery("华为手机")

.field("title")

.field("categoryName")

.field("brandName")

.defaultOperator(Operator.AND);

sourceBulider.query(query);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

}

### 6.5.7 [布尔查询](file:///D:\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\13.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_ElasticSearch\%E5%A4%87%E8%AF%BE\7.4\ElasticSearch%E7%AC%AC%E4%BA%8C%E5%A4%A9.html#布尔查询)

boolQuery：对多个查询条件连接。连接方式：  
• must（and）：条件必须成立  
• must\_not（not）：条件必须不成立  
• should（or）：条件可以成立  
• filter：条件必须成立，性能比must高。不会计算得分



**GET** goods/\_search

{

"query": {

"match": {

"title": "华为手机"

}

},

"size": 500

}



# 计算得分

GET goods/\_search

{

"query": {

"bool": {

"must": [

{

"term": {

"brandName": {

"value": "华为"

}

}

}

]

}

}

}

# 不计算得分

GET goods/\_search

{

"query": {

"bool": {

"filter": [

{

"term": {

"brandName": {

"value": "华为"

}

}

}

]

}

}

}

# 计算得分 品牌是三星，标题还得电视

GET goods/\_search

{

"query": {

"bool": {

"must": [

{

"term": {

"brandName": {

"value": "三星"

}

}

}

],

"filter": {

"term": {

"title": "电视"

}

}

}

}

}

/\*\*

\* 布尔查询：boolQuery

\* 1. 查询品牌名称为:华为

\* 2. 查询标题包含：手机

\* 3. 查询价格在：2000-3000

\*/

@Test

**public** **void** **testBoolQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

//1.构建boolQuery

BoolQueryBuilder query = QueryBuilders.boolQuery();

//2.构建各个查询条件

//2.1 查询品牌名称为:华为

QueryBuilder termQuery = QueryBuilders.termQuery("brandName","华为");

query.must(termQuery);

//2.2. 查询标题包含：手机

QueryBuilder matchQuery = QueryBuilders.matchQuery("title","手机");

query.filter(matchQuery);

//2.3 查询价格在：2000-3000

QueryBuilder rangeQuery = QueryBuilders.rangeQuery("price");

((RangeQueryBuilder) rangeQuery).gte(2000);

((RangeQueryBuilder) rangeQuery).lte(3000);

query.filter(rangeQuery);

//3.使用boolQuery连接

sourceBulider.query(query);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

}

### 6.5.8 [聚合查询](file:///D:\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\13.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_ElasticSearch\%E5%A4%87%E8%AF%BE\7.4\ElasticSearch%E7%AC%AC%E4%BA%8C%E5%A4%A9.html#聚合查询)

• 指标聚合：相当于MySQL的聚合函数。max、min、avg、sum等  
• 桶聚合：相当于MySQL的 group by 操作。不要对text类型的数据进行分组，会失败。

# 查询最贵的华为手机，max\_price命名随便取，取一个有意义的名字

GET goods/\_search

{

"query": {

"match": {

"title": "华为手机"

}

},

"aggs": {

"max\_price":{

"max": {

"field": "price"

}

}

}

}

# 桶聚合 分组

GET goods/\_search

{

"query": {

"match": {

"title": "电视"

}

},

"aggs": {

"goods\_brands": {

"terms": {

"field": "brandName",

"size": 100

}

}

}

}

/\*\*

\* 聚合查询：桶聚合，分组查询

\* 1. 查询title包含手机的数据

\* 2. 查询品牌列表

\*/

@Test

**public** **void** **testAggQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

// 1. 查询title包含手机的数据

MatchQueryBuilder query = QueryBuilders.matchQuery("title", "手机");

sourceBulider.query(query);

// 2. 查询品牌列表

/\* 参数：

1. 自定义的名称，将来用于获取数据

2. 分组的字段

\*/

AggregationBuilder agg = AggregationBuilders.terms("goods\_brands").field("brandName").size(100);

sourceBulider.aggregation(agg);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

// 获取聚合结果

Aggregations aggregations = searchResponse.getAggregations();

Map<String, Aggregation> aggregationMap = aggregations.asMap();

//System.out.println(aggregationMap);

Terms goods\_brands = (Terms) aggregationMap.get("goods\_brands");

List<? extends Terms.Bucket> buckets = goods\_brands.getBuckets();

List brands = **new** ArrayList();

**for** (Terms.Bucket bucket : buckets) {

Object key = bucket.getKey();

brands.add(key);

}

**for** (Object brand : brands) {

System.out.println(brand);

}

}

### 6.5.9 高亮查询

高亮三要素：  
• 高亮字段  
• 前缀  
• 后缀



**GET** goods/\_search

{

"query": {

"match": {

"title": "电视"

}

},

"highlight": {

"fields": {

"title": {

"pre\_tags": "<font color='red'>",

"post\_tags": "</font>"

}

}

}

}

/\*\*

\*

\* 高亮查询：

\* 1. 设置高亮

\* \* 高亮字段

\* \* 前缀

\* \* 后缀

\* 2. 将高亮了的字段数据，替换原有数据

\*/

@Test

**public** **void** **testHighLightQuery**() **throws** IOException {

SearchRequest searchRequest = **new** SearchRequest("goods");

SearchSourceBuilder sourceBulider = **new** SearchSourceBuilder();

// 1. 查询title包含手机的数据

MatchQueryBuilder query = QueryBuilders.matchQuery("title", "手机");

sourceBulider.query(query);

//设置高亮

HighlightBuilder highlighter = **new** HighlightBuilder();

//设置三要素

highlighter.field("title");

highlighter.preTags("<font color='red'>");

highlighter.postTags("</font>");

sourceBulider.highlighter(highlighter);

// 2. 查询品牌列表

/\*

参数：

1. 自定义的名称，将来用于获取数据

2. 分组的字段

\*/

AggregationBuilder agg = AggregationBuilders.terms("goods\_brands").field("brandName").size(100);

sourceBulider.aggregation(agg);

searchRequest.source(sourceBulider);

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);

SearchHits searchHits = searchResponse.getHits();

//获取记录数

**long** value = searchHits.getTotalHits().value;

System.out.println("总记录数："+value);

List<Goods> goodsList = **new** ArrayList<>();

SearchHit[] hits = searchHits.getHits();

**for** (SearchHit hit : hits) {

String sourceAsString = hit.getSourceAsString();

//转为java

Goods goods = JSON.parseObject(sourceAsString, Goods.class);

// 获取高亮结果，替换goods中的title

Map<String, HighlightField> highlightFields = hit.getHighlightFields();

HighlightField HighlightField = highlightFields.get("title");

Text[] fragments = HighlightField.fragments();

//替换

goods.setTitle(fragments[0].toString());

goodsList.add(goods);

}

**for** (Goods goods : goodsList) {

System.out.println(goods);

}

}

## 6.6 [重建索引](file:///D:\%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_%E8%AF%BE%E4%BB%B6%E7%BB%9F%E4%B8%80%E5%8C%96\13.%E5%B0%9A%E7%A1%85%E8%B0%B7_%E6%A1%86%E6%9E%B6%E9%AB%98%E7%BA%A7%E9%98%B6%E6%AE%B5_ElasticSearch\%E5%A4%87%E8%AF%BE\7.4\ElasticSearch%E7%AC%AC%E4%BA%8C%E5%A4%A9.html#重建索引)

随着业务需求的变更，索引的结构可能发生改变。  
ElasticSearch的索引一旦创建，只允许添加字段，不允许改变字段。因为改变字段，需要重建倒排索引，影响内部缓存结构，性能太低。  
那么此时，就需要重建一个新的索引，并将原有索引的数据导入到新索引中。

原索引库 ：student\_index\_v1

新索引库 ：student\_index\_v2

# 新建student\_index\_v1索引，索引名称必须全部小写

PUT student\_index\_v1

{

"mappings": {

"properties": {

"birthday":{

"type": "date"

}

}

}

}

# 查询索引

GET student\_index\_v1

# 添加数据

PUT student\_index\_v1/\_doc/1

{

"birthday":"2020-11-11"

}

# 查询数据

GET student\_index\_v1/\_search

# 随着业务的变更，换种数据类型进行添加数据，程序会直接报错

PUT student\_index\_v1/\_doc/1

{

"birthday":"2020年11月11号"

}

# 业务变更，需要改变birthday数据类型为text

# 1：创建新的索引 student\_index\_v2

# 2：将student\_index\_v1 数据拷贝到 student\_index\_v2

# 创建新的索引

PUT student\_index\_v2

{

"mappings": {

"properties": {

"birthday":{

"type": "text"

}

}

}

}

DELETE student\_index\_v2

# 2：将student\_index\_v1 数据拷贝到 student\_index\_v2

POST \_reindex

{

"source": {

"index": "student\_index\_v1"

},

"dest": {

"index": "student\_index\_v2"

}

}

# 查询新索引库数据

GET student\_index\_v2/\_search

# 在新的索引库里面添加数据

PUT student\_index\_v2/\_doc/2

{

"birthday":"2020年11月13号"

}

# 第七章 Spring Data Elasticsearch

### 7.1 目标

* Spring Data的作用:简化了数据库的增删改查操作
* Spring Data Elasticsearch入门[掌握]
* Spring Data Elasticsearch查询命名规则[掌握]

### 7.2 讲解

#### 7.2.1 Spring Data Jpa介绍[集成]

JPA是一个规范，真正操作数据库的是Hibernate（实现数据库增删改查框架[ORM框架],操作数据库采用的方式是面向对象[不写SQL语句]），而springdatajpa是对jpa的封装，将CRUD的方法封装到指定的方法中，操作的时候，只需要调用方法即可。

Spring Data Jpa的实现过程：

1：定义实体，实体类添加Jpa的注解 @Entity @Talbe @Cloumn @Id

2：定义接口，接口要继承JpaRepository的接口

3：配置spring容器，applicationContext.xml/SpringApplication.run(T.**class**,args)

#### 7.2.2 Spring Data ElasticSearch简介

(1)SpringData介绍

Spring Data是一个用于简化数据库、非关系型数据库、索引库访问，并支持云服务的开源框架。

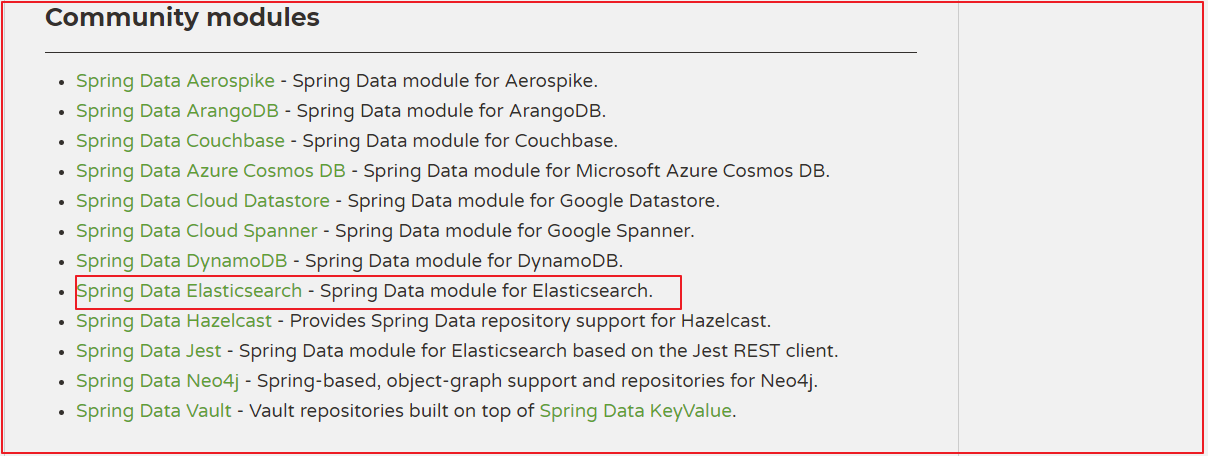
其主要目标是使得对数据的访问变得方便快捷，并支持map-reduce框架和云计算数据服务。

Spring Data可以极大的简化JPA（Elasticsearch…）的写法，可以在几乎不用写实现的情况下，实现对数据的访问和操作。除了CRUD外，还包括如分页、排序等一些常用的功能。

Spring Data的官网：<http://projects.spring.io/spring-data/>

Spring Data常用的功能模块如下：





(2)SpringData Elasticsearch介绍

Spring Data ElasticSearch 基于 spring data API 简化 elasticSearch操作，将原始操作elasticSearch的客户端API 进行封装 。

Spring Data为Elasticsearch项目提供集成搜索引擎。

官方网站：<http://projects.spring.io/spring-data-elasticsearch/>

#### 7.2.3 Spring Data Elasticsearch入门

##### 7.2.3.1 搭建工程

(1)搭建工程

创建项目 elasticsearch-springdata-es

(2)pom.xml依赖

<?xml version="1.0" encoding="UTF-8"?>

<**project** **xmlns**="http://maven.apache.org/POM/4.0.0"

**xmlns:xsi**="http://www.w3.org/2001/XMLSchema-instance"

**xsi:schemaLocation**="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<**modelVersion**>4.0.0</**modelVersion**>

<**groupId**>com.atguigu</**groupId**>

<**artifactId**>elasticsearch-springdata-es</**artifactId**>

<**version**>1.0-SNAPSHOT</**version**>

<**parent**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-parent</**artifactId**>

<**version**>2.2.2.RELEASE</**version**>

</**parent**>

<**dependencies**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-data-elasticsearch</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-test</**artifactId**>

<**scope**>test</**scope**>

</**dependency**>

<**dependency**>

<**groupId**>junit</**groupId**>

<**artifactId**>junit</**artifactId**>

<**version**>4.12</**version**>

<**scope**>test</**scope**>

</**dependency**>

</**dependencies**>

<**build**>

<**plugins**>

<!-- java编译插件 -->

<**plugin**>

<**groupId**>org.apache.maven.plugins</**groupId**>

<**artifactId**>maven-compiler-plugin</**artifactId**>

<**version**>3.2</**version**>

<**configuration**>

<**source**>1.8</**source**>

<**target**>1.8</**target**>

<**encoding**>UTF-8</**encoding**>

</**configuration**>

</**plugin**>

</**plugins**>

</**build**>

</**project**>

##### 7.2.3.2 编写实体类

创建com.atguigu.domain.Item，代码如下：

**package** com.atguigu.pojo;

**import** org.springframework.data.annotation.Id;

**import** org.springframework.data.elasticsearch.annotations.Document;

**import** org.springframework.data.elasticsearch.annotations.Field;

**import** org.springframework.data.elasticsearch.annotations.FieldType;

@Document(indexName = "item",shards = 1, replicas = 1)

**public** **class** **Item** {

@Id

**private** Long id;

@Field(type = FieldType.Text, analyzer = "ik\_max\_word")

**private** String title; //标题

@Field(type = FieldType.Keyword)

**private** String category;// 分类

@Field(type = FieldType.Keyword)

**private** String brand; // 品牌

@Field(type = FieldType.Double)

**private** Double price; // 价格

@Field(index = **false**, type = FieldType.Keyword)

**private** String images; // 图片地址

**public** **Item**() {

}

**public** **Item**(Long id, String title, String category, String brand, Double price, String images) {

**this**.id = id;

**this**.title = title;

**this**.category = category;

**this**.brand = brand;

**this**.price = price;

**this**.images = images;

}

//get/set/toString…

}

映射

Spring Data通过注解来声明字段的映射属性，有下面的三个注解：

@Document 作用在类，标记实体类为文档对象，一般有四个属性

indexName：对应索引库名称  
shards：分片数量，默认5  
replicas：副本数量，默认1

@Id 作用在成员变量，标记一个字段作为id主键

@Field 作用在成员变量，标记为文档的字段，并指定字段映射属性：

type：字段类型，取值是枚举：FieldType  
index：是否索引，布尔类型，默认是true  
store：是否存储，布尔类型，默认是false  
analyzer：分词器名称：ik\_max\_word

##### 7.2.3.3 配置 application.properties 文件

# es服务地址

elasticsearch.host=127.0.0.1

# es服务端口

elasticsearch.port=9200

# 配置日志级别,开启debug日志

logging.level.com.atguigu=debug

##### 7.2.3.4 配置类

<https://docs.spring.io/spring-data/elasticsearch/docs/3.2.3.RELEASE/reference/html/#elasticsearch.mapping.meta-model>

<https://docs.spring.io/spring-data/elasticsearch/docs/3.2.3.RELEASE/reference/html/#elasticsearch.operations.resttemplate>

* ElasticsearchRestTemplate是spring-data-elasticsearch项目中的一个类，和其他spring项目中的template类似。
* 在新版的spring-data-elasticsearch中，ElasticsearhRestTemplate代替了原来的ElasticsearchTemplate。
* 原因是ElasticsearchTemplate基于TransportClient，TransportClient即将在8.x以后的版本中移除。所以，我们推荐使用ElasticsearchRestTemplate。
* ElasticsearchRestTemplate基于RestHighLevelClient客户端的。需要自定义配置类，继承AbstractElasticsearchConfiguration，并实现elasticsearchClient()抽象方法，创建RestHighLevelClient对象。

package com.atguigu.es.config;

import lombok.Data;

import org.apache.http.HttpHost;

import org.elasticsearch.client.RestClient;

import org.elasticsearch.client.RestClientBuilder;

import org.elasticsearch.client.RestHighLevelClient;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.convert.support.DefaultConversionService;

import org.springframework.data.elasticsearch.client.ClientConfiguration;

import org.springframework.data.elasticsearch.client.RestClients;

import org.springframework.data.elasticsearch.config.AbstractElasticsearchConfiguration;

import org.springframework.data.elasticsearch.core.ElasticsearchEntityMapper;

import org.springframework.data.elasticsearch.core.EntityMapper;

@ConfigurationProperties(prefix = "elasticsearch")

@Configuration

public class ElasticsearchConfig extends AbstractElasticsearchConfiguration {

private String host ;

private Integer port ;

//重写父类方法

@Override

public RestHighLevelClient elasticsearchClient() {

RestClientBuilder builder = RestClient.builder(new HttpHost(host, port));

RestHighLevelClient restHighLevelClient = new RestHighLevelClient(builder);

return restHighLevelClient;

}

/\* 重写父类方法

@Override

public RestHighLevelClient elasticsearchClient() {

return RestClients.create(ClientConfiguration.create("localhost:9201")).rest();

}\*/

@Bean

@Override

public EntityMapper entityMapper() {

ElasticsearchEntityMapper entityMapper = new ElasticsearchEntityMapper(elasticsearchMappingContext(), new DefaultConversionService());

entityMapper.setConversions(elasticsearchCustomConversions());

return entityMapper;

}

//get/set…

}

#### 7.2.4 测试

##### 7.2.4.1 索引操作

**package** com.atguigu;

**import** com.atguigu.pojo.Item;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.data.elasticsearch.core.ElasticsearchRestTemplate;

**import** org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)

@SpringBootTest

**public** **class** **TestSpringBootES** {

@Autowired

**private** ElasticsearchRestTemplate elasticsearchTemplate;

@Test

**public** **void** **testCreate**() {

// 创建索引，会根据Item类的@Document注解信息来创建

elasticsearchTemplate.createIndex(Item.class);

// 配置映射，会根据Item类中的id、Field等字段来自动完成映射

elasticsearchTemplate.putMapping(Item.class);

}

使用 kibana 查询

GET **item**

##### 7.2.4.2 增删改操作

Spring Data 的强大之处，就在于你不用写任何DAO处理，自动根据方法名或类的信息进行CRUD操作。只要你定义一个接口，然后继承Repository提供的一些子接口，就能具备各种基本的CRUD功能。

编写 ItemRepository

**package** com.atguigu.dao;

**import** com.atguigu.pojo.Item;

**import** org.springframework.data.elasticsearch.repository.ElasticsearchRepository;

**public** **interface** **ItemRepository** **extends** **ElasticsearchRepository**<**Item**,**Long**>{

}

增加

**package** com.atguigu;

**import** com.atguigu.com.ItemRepository;

**import** com.atguigu.pojo.Item;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.data.elasticsearch.core.ElasticsearchRestTemplate;

**import** org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)

@SpringBootTest

**public** **class** **TestSpringBootES** {

@Autowired

**private** ElasticsearchRestTemplate elasticsearchTemplate;

@Autowired

**private** ItemRepository itemRepository;

@Test

**public** **void** **testAdd**() {

Item item = **new** Item(1L, "小米手机7", " 手机", "小米", 3499.00, "http://image.leyou.com/13123.jpg");

itemRepository.save(item);

}

}

修改(id存在就是修改，否则就是插入)

@Test

**public** **void** **testUpdate**() {

Item item = **new** Item(1L, "小米手机7777", " 手机", "小米", 9499.00, "http://image.leyou.com/13123.jpg");

itemRepository.save(item);

}

批量新增

@Test

**public** **void** **indexList**() {

List<Item> list = **new** ArrayList<>();

list.add(**new** Item(2L, "坚果手机R1", " 手机", "锤子", 3699.00, "http://image.leyou.com/123.jpg"));

list.add(**new** Item(3L, "华为META10", " 手机", "华为", 4499.00, "http://image.leyou.com/3.jpg"));

// 接收对象集合，实现批量新增

itemRepository.saveAll(list);

}

删除操作

@Test

**public** **void** **testDelete**() {

itemRepository.deleteById(1L);

}

根据id查询

@Test

**public** **void** **testQuery**(){

Optional<Item> optional = itemRepository.findById(2L);

System.out.println(optional.get());

}

查询全部，并按照价格降序排序

@Test

**public** **void** **testFind**(){

// 查询全部，并按照价格降序排序

Iterable<Item> items = **this**.itemRepository.findAll(Sort.by(Sort.Direction.DESC, "price"));

items.forEach(item-> System.out.println(item));

}

#### 7.2.5 自定义方法

Spring Data 的另一个强大功能，是根据方法名称自动实现功能。  
比如：你的方法名叫做：findByTitle，那么它就知道你是根据title查询，然后自动帮你完成，无需写实现类。  
当然，方法名称要符合一定的约定：

| **Keyword** | **Sample** | **Elasticsearch Query String** |
| --- | --- | --- |
| And | findByNameAndPrice | {"bool" : {"must" : [ {"field" : {"name" : "?"}}, {"field" : {"price" : "?"}} ]}} |
| Or | findByNameOrPrice | {"bool" : {"should" : [ {"field" : {"name" : "?"}}, {"field" : {"price" : "?"}} ]}} |
| Is | findByName | {"bool" : {"must" : {"field" : {"name" : "?"}}}} |
| Not | findByNameNot | {"bool" : {"must\_not" : {"field" : {"name" : "?"}}}} |
| Between | findByPriceBetween | {"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : ?,"include\_lower" : true,"include\_upper" : true}}}}} |
| LessThanEqual | findByPriceLessThan | {"bool" : {"must" : {"range" : {"price" : {"from" : null,"to" : ?,"include\_lower" : true,"include\_upper" : true}}}}} |
| GreaterThanEqual | findByPriceGreaterThan | {"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : null,"include\_lower" : true,"include\_upper" : true}}}}} |
| Before | findByPriceBefore | {"bool" : {"must" : {"range" : {"price" : {"from" : null,"to" : ?,"include\_lower" : true,"include\_upper" : true}}}}} |
| After | findByPriceAfter | {"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : null,"include\_lower" : true,"include\_upper" : true}}}}} |
| Like | findByNameLike | {"bool" : {"must" : {"field" : {"name" : {"query" : "?\*","analyze\_wildcard" : true}}}}} |
| StartingWith | findByNameStartingWith | {"bool" : {"must" : {"field" : {"name" : {"query" : "?\*","analyze\_wildcard" : true}}}}} |
| EndingWith | findByNameEndingWith | {"bool" : {"must" : {"field" : {"name" : {"query" : "\*?","analyze\_wildcard" : true}}}}} |
| Contains/Containing | findByNameContaining | {"bool" : {"must" : {"field" : {"name" : {"query" : "\*\*?\*\*","analyze\_wildcard" : true}}}}} |
| In | findByNameIn(Collection<String>names) | {"bool" : {"must" : {"bool" : {"should" : [ {"field" : {"name" : "?"}}, {"field" : {"name" : "?"}} ]}}}} |
| NotIn | findByNameNotIn(Collection<String>names) | {"bool" : {"must\_not" : {"bool" : {"should" : {"field" : {"name" : "?"}}}}}} |
| Near | findByStoreNear | Not Supported Yet ! |
| True | findByAvailableTrue | {"bool" : {"must" : {"field" : {"available" : true}}}} |
| False | findByAvailableFalse | {"bool" : {"must" : {"field" : {"available" : false}}}} |
| OrderBy | findByAvailableTrueOrderByNameDesc | {"sort" : [{ "name" : {"order" : "desc"} }],"bool" : {"must" : {"field" : {"available" : true}}}} |

例如，我们来按照价格区间查询，定义这样的一个方法：

**package** com.atguigu.com;

**import** com.atguigu.pojo.Item;

**import** org.springframework.data.elasticsearch.repository.ElasticsearchRepository;

**import** java.util.List;

**public** **interface** **ItemRepository** **extends** **ElasticsearchRepository**<**Item**,**Long**>{

List<Item> **findByPriceBetween**(**double** price1, **double** price2);

}

然后添加一些测试数据：

@Test

**public** **void** **indexList2**() {

List<Item> list = **new** ArrayList<>();

list.add(**new** Item(1L, "小米手机7", "手机", "小米", 3299.00, "http://image.leyou.com/13123.jpg"));

list.add(**new** Item(2L, "坚果手机R1", "手机", "锤子", 3699.00, "http://image.leyou.com/13123.jpg"));

list.add(**new** Item(3L, "华为META10", "手机", "华为", 4499.00, "http://image.leyou.com/13123.jpg"));

list.add(**new** Item(4L, "小米Mix2S", "手机", "小米", 4299.00, "http://image.leyou.com/13123.jpg"));

list.add(**new** Item(5L, "荣耀V10", "手机", "华为", 2799.00, "http://image.leyou.com/13123.jpg"));

// 接收对象集合，实现批量新增

itemRepository.saveAll(list);

}

不需要写实现类，然后我们直接去运行：

@Test

**public** **void** **queryByPriceBetween**(){

List<Item> list = **this**.itemRepository.findByPriceBetween(2000.00, 3500.00);

**for** (Item item : list) {

System.out.println("item = " + item);

}

}

虽然基本查询和自定义方法已经很强大了，但是如果是复杂查询（模糊、通配符、词条查询等）就显得力不从心了。此时，我们只能使用原生查询。