### Elasticsearch特点

#### 海量数据处理

* 大型分布式集群（数百台规模服务器）
* 处理PB级数据
* 小公司也可以进行单机部署

#### 开箱即用

* 简单易用，操作非常简单
* 快速部署生产环境

#### 作为传统数据库的补充

* 传统关系型数据库不擅长全文检索（MySQL自带的全文索引，与ES性能差距非常大）
* 传统关系型数据库无法支持搜索排名、海量数据存储、分析等功能
* Elasticsearch可以作为传统关系数据库的补充，提供RDBM无法提供的功能

### ElasticSearch对比Solr

* Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能;
* Solr 支持更多格式的数据，而 Elasticsearch 仅支持json文件格式；
* 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；
* Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时效率明显低于Elasticsearch

## Lucene全文检索库

### 什么是全文检索

#### 结构化数据与非结构化数据

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

#### 搜索结构化数据和非结构化数据

* 使用SQL语言专门搜索结构化的数据
* 使用ES/Lucene/solr建立**倒排索引**，根据关键字就可以搜索一些非结构化(文本)的数据

#### 全文检索

全文检索是指：

* 通过一个程序扫描文本中的每一个单词，针对单词建立索引，并保存该单词在文本中的位置、以及出现的次数
* 用户查询时，通过之前建立好的索引来查询，将索引中单词对应的文本位置、出现的次数返回给用户，因为有了具体文本的位置，所以就可以将具体内容读取出来了
* 类似于通过字典中的检索字表查字的过程

### Lucene简介

* Lucene是一种高性能的全文检索库，在2000年开源，最初由大名鼎鼎的Doug Cutting（道格·卡丁）开发
* Lucene是Apache的一个顶级开源项目，是一个全文检索引擎工具包。但Lucene不是一个完整的全文检索引擎，它只是提供一个基本的全文检索的架构，还提供了一些基本的文本分词库
* Lucene是一个简单易用的工具包，可以方便的实现全文检索的功能

### 美文搜索案例

#### 需求

在资料中的文章文件夹中，有很多的文本文件。这里面包含了一些非常有趣的软文。而我们想要做的事情是，通过搜索一个关键字就能够找到哪些文章包含了这些关键字。例如：搜索「hadoop」，就能找到hadoop相关的文章。

**需求分析**：

要实现以上需求，我们有以下两种办法：

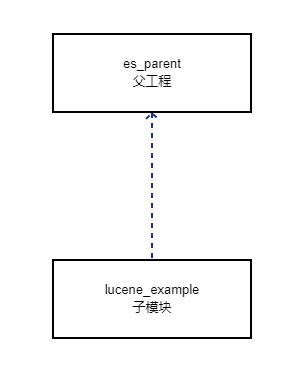
1. 用户输入搜索关键字，然后我们挨个读取文件，并查找文件中是否包含关键字
2. 我们先挨个读取文件，对文件的文本进行分词（例如：按标点符号），然后建立索引，用户输入关键字，根据之前建立的索引，搜索关键字。

很明显，第二种方式要比第一种效果好得多，性能也好得多。所以，我们下面就使用Lucene来建立索引，然后根据索引来进行检索。

#### 准备工作

##### 创建IDEA项目

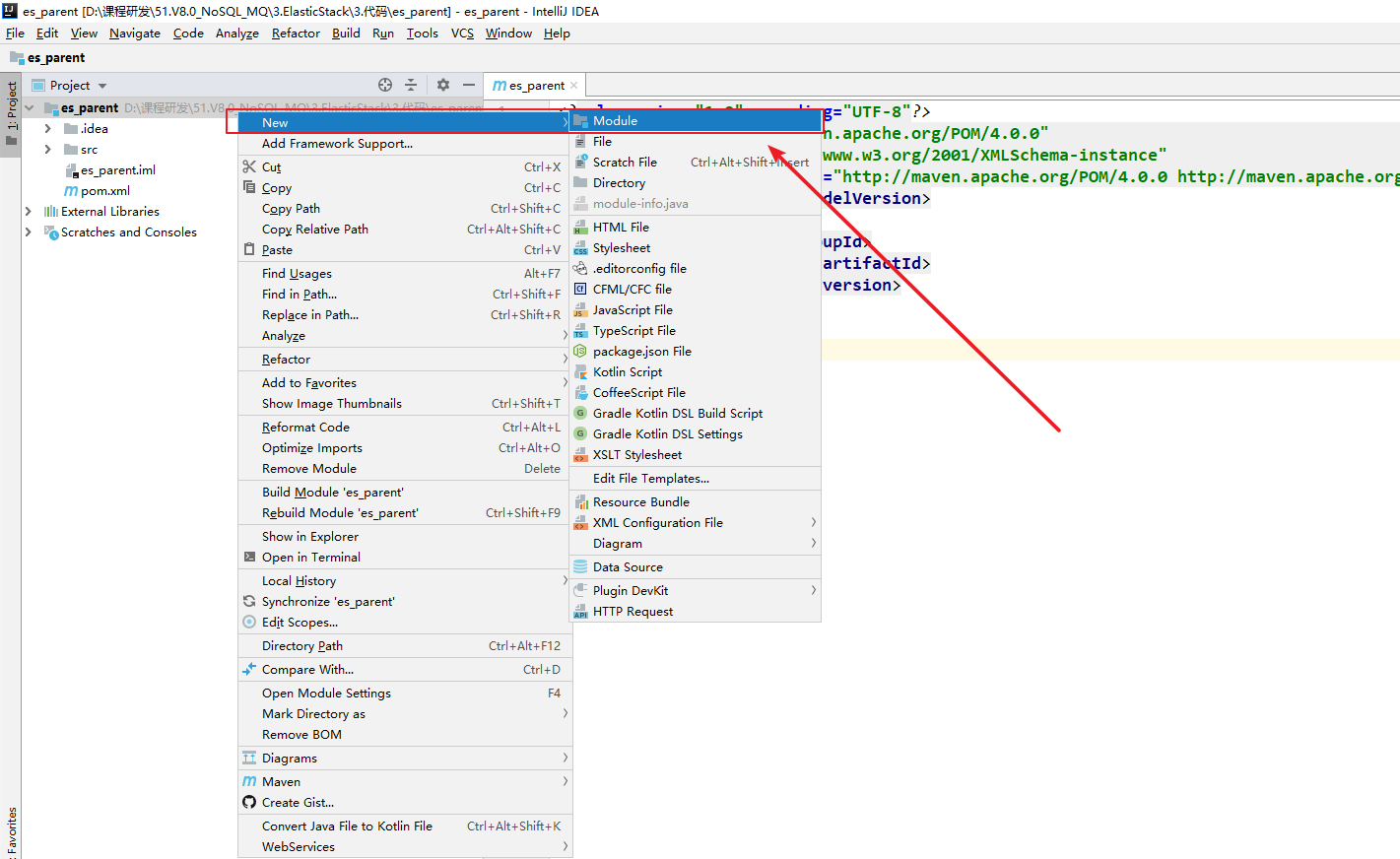
Elasticstack阶段的课程会有多个项目，所以此处在IDEA中的工程模型如下：



##### 创建父工程

|  |  |
| --- | --- |
| groupId | com.as |
| artifactId | es\_parent |

##### 添加lucene模块



|  |  |
| --- | --- |
| groupId | com.as |
| artifactId | lucene\_op |

##### 导入Maven依赖

1. 导入依赖到lucene\_op的pom.xml

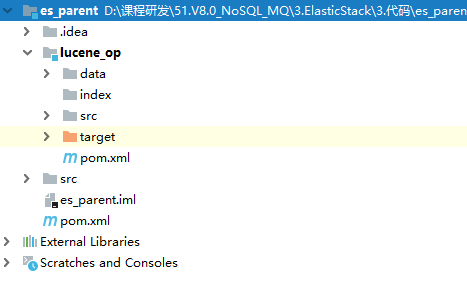
|  |
| --- |
| <dependencies>  <!-- lucene核心类库 -->  <dependency>  <groupId>org.apache.lucene</groupId>  <artifactId>lucene-core</artifactId>  <version>8.4.0</version>  </dependency>  <dependency>  <groupId>org.apache.lucene</groupId>  <artifactId>lucene-analyzers-common</artifactId>  <version>8.4.0</version>  </dependency>  <dependency>  <groupId>commons-io</groupId>  <artifactId>commons-io</artifactId>  <version>2.6</version>  </dependency>  <dependency>  <groupId>com.jianggujin</groupId>  <artifactId>IKAnalyzer-lucene</artifactId>  <version>8.0.0</version>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>**org.apache.maven.plugins**</groupId>  <artifactId>**maven-compiler-plugin**</artifactId>  <version>**3.1**</version>  <configuration>  <source>**1.8**</source>  <target>**1.8**</target>  </configuration>  </plugin>  </plugins>  </build> |

##### 创建包和类

1. 在java目录创建 com.as.lucene 包结构
2. 创建BuildArticleIndex类

##### 导入文章数据

1. 在 lucene\_op 模块下创建名为 data 的目录，用来存放文章文件
2. 在 lucene\_op 模块下创建名为 index 的目录，用于存放最后生成的索引文件
3. 将资料/文章目录下的txt文件复制到 data 目录中



#### 建立索引库

##### 实现步骤

1. 构建分词器（StandardAnalyzer）
2. 构建文档写入器配置（IndexWriterConfig）
3. 构建文档写入器（IndexWriter，注意：需要使用Paths来）
4. 读取所有文件构建文档
5. 文档中添加字段

|  |  |  |
| --- | --- | --- |
| 字段名 | 类型 | 说明 |
| file\_name | TextFiled | 文件名字段，需要在索引文档中保存文件名内容 |
| content | TextFiled | 内容字段，只需要能被检索，但无需在文档中保存 |
| path | StoredFiled | 路径字段，无需被检索，只需要在文档中保存即可 |

1. 写入文档
2. 关闭写入器

##### 参考代码

|  |
| --- |
| **public class** BuildArticleIndex {  **public static void** main(String[] args) **throws** IOException {  // 1. 构建分词器（StandardAnalyzer）  StandardAnalyzer standardAnalyzer = **new** StandardAnalyzer();   // 2. 构建文档写入器配置（IndexWriterConfig）  IndexWriterConfig indexWriterConfig = **new** IndexWriterConfig(standardAnalyzer);   // 3. 构建文档写入器（IndexWriter）  IndexWriter indexWriter = **new** IndexWriter(  FSDirectory.*open*(Paths.*get*(**"D:\\课程研发\\51.V8.0\_NoSQL\_MQ\\3.ElasticStack\\3.代码\\es\_parent\\lucene\_op\\index"**)), indexWriterConfig);   // 4. 读取所有文件构建文档  File articleDir = **new** File(**"D:\\课程研发\\51.V8.0\_NoSQL\_MQ\\3.ElasticStack\\3.代码\\es\_parent\\lucene\_op\\data"**);  File[] fileList = articleDir.listFiles();   **for** (File file : fileList) {  // 5. 文档中添加字段  Document docuemnt = **new** Document();  docuemnt.add(**new** TextField(**"file\_name"**, file.getName(), Field.Store.***YES***));  docuemnt.add(**new** TextField(**"content"**, FileUtils.*readFileToString*(file, **"UTF-8"**), Field.Store.***NO***));  docuemnt.add(**new** StoredField(**"path"**, file.getAbsolutePath() + **"/"** + file.getName()));  // 6. 写入文档  indexWriter.addDocument(docuemnt);  }   // 7. 关闭写入器  indexWriter.close();  } } |

#### 关键字查询

##### 需求

输入一个关键字“心”，根据关键字查询索引库中是否有匹配的文档

##### 准备工作

1. 前提：基于文章文本文件，已经生成好了索引
2. 在com.as.lucene包下创建一个类KeywordSearch

##### 开发步骤

1. 使用DirectoryReader.open构建索引读取器
2. 构建索引查询器（IndexSearcher）
3. 构建词条（Term）和词条查询（TermQuery）
4. 执行查询，获取文档
5. 遍历打印文档（可以使用IndexSearch.doc根据文档ID获取到文档）
6. 关键索引读取器

##### 参考代码

|  |
| --- |
| **public class** KeywordSearch {  **public static void** main(String[] args) **throws** IOException {  // 1. 构建索引读取器  IndexReader indexReader = DirectoryReader.*open*(FSDirectory.*open*(Paths.*get*(**"D:\\课程研发\\51.V8.0\_NoSQL\_MQ\\3.ElasticStack\\3.代码\\es\_parent\\lucene\_op\\index"**)));   // 2. 构建索引查询器  IndexSearcher indexSearcher = **new** IndexSearcher(indexReader);   // 3. 执行查询，获取文档  TermQuery termQuery = **new** TermQuery(**new** Term(**"content"**, **"心"**));   TopDocs topDocs = indexSearcher.search(termQuery, 50);  ScoreDoc[] scoreDocArrary = topDocs.**scoreDocs**;   // 4. 遍历打印文档  **for** (ScoreDoc scoreDoc : scoreDocArrary) {  **int** docId = scoreDoc.**doc**;  Document document = indexSearcher.doc(docId);   System.***out***.println(**"文件名:"** + document.get(**"file\_name"**) + **" 路径："** + document.get(**"path"**));  }   indexReader.close();   } } |

#### 搜索词语问题

上述代码，都是一个字一个字的搜索，但如果搜索一个词，例如：“情愿”，我们会发现，我们什么都搜索不出来。所以，接下来，我们还需要来解决搜索一个词的问题。

#### 分词器与中文分词器

分词器是指将一段文本，分割成为一个个的词语的动作。例如：按照停用词进行分隔（的、地、啊、吧、标点符号等）。我们之前在代码中使用的分词器是Lucene中自带的分词器。这个分词器对中文很不友好，只是将一个一个字分出来，所以，就会从后出现上面的问题——无法搜索词语。

所以，基于该背景，我们需要使用跟适合中文的分词器。中文分词器也有不少，例如：

* Jieba分词器
* IK分词器
* 庖丁分词器
* Smarkcn分词器

等等。此处，我们使用比较好用的IK分词器来进行分词。

IK已经实现好了Lucene的分词器：<https://github.com/wks/ik-analyzer>

|  |
| --- |
| 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Luence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。  IKAnalyzer3.0特性:  采用了特有的“正向迭代最细粒度切分算法“，支持细粒度和最大词长两种切分模式；具有83万字/秒（1600KB/S）的高速处理能力。  采用了多子处理器分析模式，支持：英文字母、数字、中文词汇等分词处理，兼容韩文、日文字符  优化的词典存储，更小的内存占用。支持用户词典扩展定义  针对Lucene全文检索优化的查询分析器IKQueryParser(作者吐血推荐)；引入简单搜索表达式，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。 |

#### 使用IK分词器重构案例

##### 准备工作

1. 添加Maven依赖

|  |
| --- |
| <dependency>  <groupId>com.jianggujin</groupId>  <artifactId>IKAnalyzer-lucene</artifactId>  <version>8.0.0</version>  </dependency> |

1. 创建BuildArticleIndexByIkAnalyzer类

##### 实现步骤

把之前生成的索引文件删除，然后将之前使用的StandardAnalyzer修改为IKAnalyzer。然后重新生成索引。

##### 参考代码

|  |
| --- |
| **public class** BuildArticleIndexByIkAnalyzer {  **public static void** main(String[] args) **throws** IOException {  // 1. 构建分词器（StandardAnalyzer）  IKAnalyzer ikAnalyzer = **new** IKAnalyzer();   // 2. 构建文档写入器配置（IndexWriterConfig）  IndexWriterConfig indexWriterConfig = **new** IndexWriterConfig(ikAnalyzer);   // 3. 构建文档写入器（IndexWriter）  IndexWriter indexWriter = **new** IndexWriter(  FSDirectory.*open*(Paths.*get*(**"D:\\课程研发\\51.V8.0\_NoSQL\_MQ\\3.ElasticStack\\3.代码\\es\_parent\\lucene\_op\\index"**)), indexWriterConfig);   // 4. 读取所有文件构建文档  File articleDir = **new** File(**"D:\\课程研发\\51.V8.0\_NoSQL\_MQ\\3.ElasticStack\\3.代码\\es\_parent\\lucene\_op\\data"**);  File[] fileList = articleDir.listFiles();   **for** (File file : fileList) {  // 5. 文档中添加字段  Document docuemnt = **new** Document();  docuemnt.add(**new** TextField(**"file\_name"**, file.getName(), Field.Store.***YES***));  docuemnt.add(**new** TextField(**"content"**, FileUtils.*readFileToString*(file, **"UTF-8"**), Field.Store.***NO***));  docuemnt.add(**new** StoredField(**"path"**, file.getAbsolutePath() + **"/"** + file.getName()));  // 6. 写入文档  indexWriter.addDocument(docuemnt);  }   // 7. 关闭写入器  indexWriter.close();  } } |

##### 问题

通过使用IK分词器进行分词，我们发现，现在我们的程序可以搜索词语了。但如果我们输入一句话：人生是一条河，我们想要搜索出来与其相关的文章。应该如何实现呢？

#### 句子搜索

在com.as.lucene 包下创建一个SentenceSearch类。

##### 实现步骤

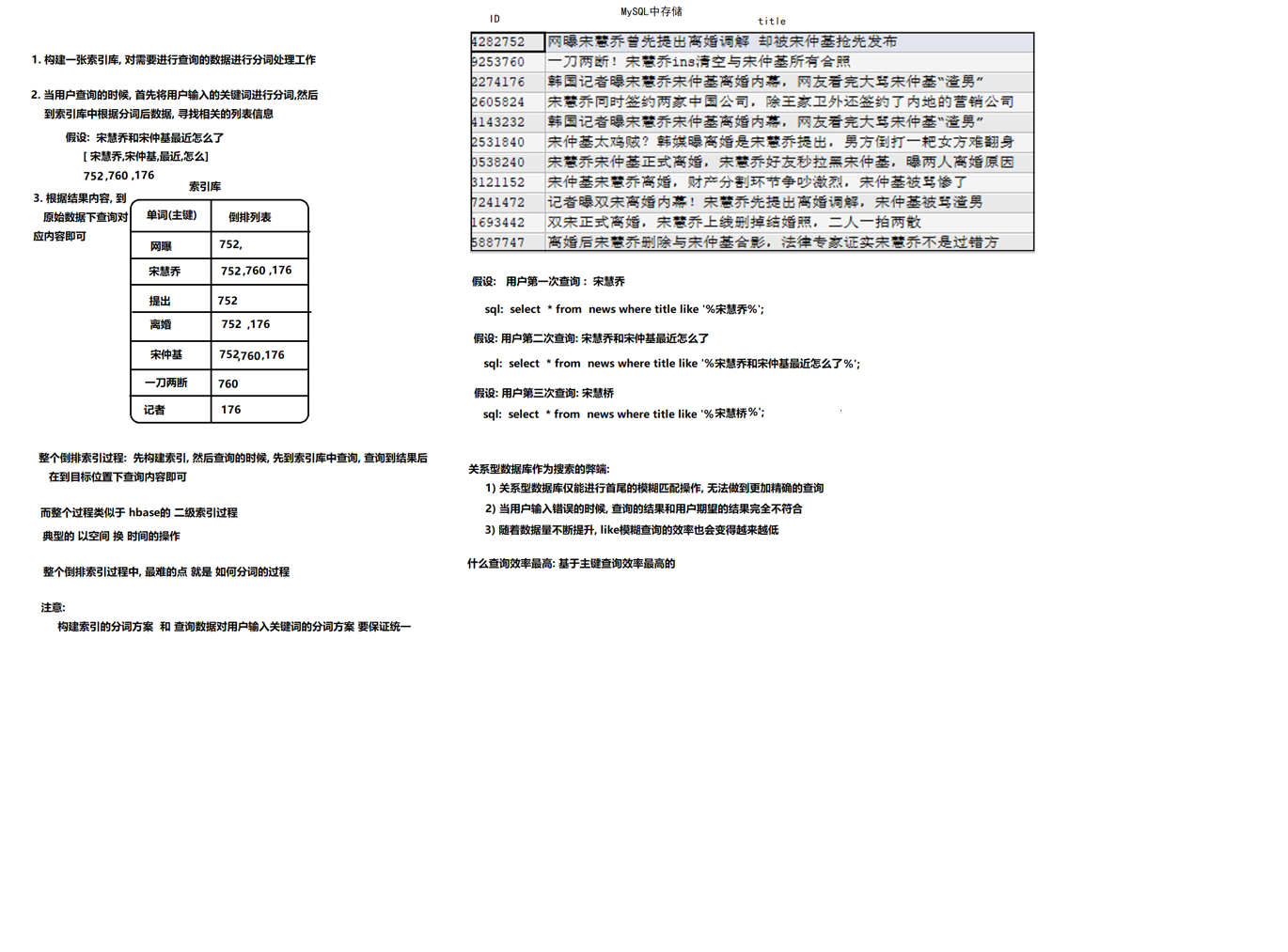
要实现搜索句子，其实是将句子进行分词后，再进行搜索。我们需要使用QueryParser类来实现。通过QueryParser可以指定分词器对要搜索的句子进行分词。

##### 参考代码

|  |
| --- |
| **public class** SentenceSearch {  **public static void** main(String[] args) **throws** IOException, ParseException {  // 1. 构建索引读取器  IndexReader indexReader = DirectoryReader.*open*(FSDirectory.*open*(Paths.*get*(**"D:\\课程研发\\51.V8.0\_NoSQL\_MQ\\3.ElasticStack\\3.代码\\es\_parent\\lucene\_op\\index"**)));   // 2. 构建索引查询器  IndexSearcher indexSearcher = **new** IndexSearcher(indexReader);   // 3. 执行查询，获取文档  QueryParser queryParser = **new** QueryParser(**"content"**, **new** IKAnalyzer());   TopDocs topDocs = indexSearcher.search(queryParser.parse(**"人生是一条河"**), 50);  ScoreDoc[] scoreDocArrary = topDocs.**scoreDocs**;   // 4. 遍历打印文档  **for** (ScoreDoc scoreDoc : scoreDocArrary) {  **int** docId = scoreDoc.**doc**;  Document document = indexSearcher.doc(docId);   System.***out***.println(**"文件名:"** + document.get(**"file\_name"**) + **" 路径："** + document.get(**"path"**));  }   indexReader.close();   } } |

### 倒排索引结构

倒排索引是一种建立索引的方法。是全文检索系统中常用的数据结构。通过倒排索引，就是根据单词快速获取包含这个单词的文档列表。倒排索引通常由两个部分组成：单词词典、文档。



### 企业中为什么不直接使用Lucene

#### Lucene的内建不支持分布式

Lucene是作为嵌入的类库形式使用的，本身是没有对分布式支持。

#### 区间范围搜索速度非常缓慢

* Lucene的区间范围搜索API是扩展补充的，对于在单个文档中term出现比较多的情况，搜索速度会变得很慢
* Lucene只有在数据生成索引文件之后（Segment），才能被查询到，做不到实时

#### 可靠性无法保障

* 无法保障Segment索引段的可靠性

## Elasticsearch中的核心概念

### 索引 index

* 一个索引就是一个拥有几分相似特征的文档的集合。比如说，可以有一个客户数据的索引，另一个产品目录的索引，还有一个订单数据的索引
* 一个索引由一个名字来标识（必须全部是小写字母的），并且当我们要对对应于这个索引中的文档进行索引、搜索、更新和删除的时候，都要使用到这个名字
* 在一个集群中，可以定义任意多的索引。

### 映射 mapping

* ElasticSearch中的映射（Mapping）用来定义一个文档
* mapping是处理数据的方式和规则方面做一些限制，如某个字段的数据类型、默认值、分析器、是否被索引等等，这些都是映射里面可以设置的

### 字段Field

相当于是数据表的字段，对文档数据根据不同属性进行的分类标识

### 类型 Type

每一个字段都应该有一个对应的类型，例如：Text、Keyword、Byte等

### 文档 document

* 一个文档是一个可被索引的基础信息单元。比如，可以拥有某一个客户的文档，某一个产品的一个文档，当然，也可以拥有某个订单的一个文档。文档以JSON（Javascript Object Notation）格式来表示，而JSON是一个到处存在的互联网数据交互格式

### 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

### 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，如果在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，可以拥有任意多个节点。而且，如果当前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。

### 分片和副本 shards&replicas

#### 分片

* 一个索引可以存储超出单个结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
* 当创建一个索引的时候，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
  + 允许水平分割/扩展你的内容容量
  + 允许在分片之上进行分布式的、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，对于作为用户来说，这些都是透明的

#### 副本

* 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，这些拷贝叫做副本分片，或者直接叫副本
* 副本之所以重要，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。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
  + 扩展搜索量/吞吐量，因为搜索可以在所有的副本上并行运行
* 每个索引可以被分成多个分片。一个索引有0个或者多个副本
* 一旦设置了副本，每个索引就有了主分片和副本分片，分片和副本的数量可以在索引创建的时候指定
* 在索引创建之后，可以在任何时候动态地改变副本的数量，但是不能改变分片的数量

## 安装Elasticsearch

### 安装Elasticsearch

#### 创建普通用户

**ES不能使用root用户来启动，必须使用普通用户来安装启动**。这里我们创建一个普通用户以及定义一些常规目录用于存放我们的数据文件以及安装包等。

创建一个es专门的用户（**必须**）

# 使用root用户在**三台机器**执行以下命令

|  |
| --- |
| useradd roy  passwd 123456 |

**这里可以使用老师提供的虚拟机中的roy用户，密码也是123456。**

#### 为普通用户roy添加sudo权限

为了让普通用户有更大的操作权限，我们一般都会给普通用户设置sudo权限，方便普通用户的操作

三台机器使用root用户执行visudo命令然后为es用户添加权限

|  |
| --- |
| visudo  # 第100行  roy ALL=(ALL) ALL |

#### 上传压缩包并解压

以下操作 使用**root用户**es的相关的目录, 三台虚拟机都需要创建

|  |
| --- |
| # 在node1、node2、node3创建es文件夹，并修改owner为roy用户  mkdir -p /export/server/es  chown -R roy:roy /export/server/es |

将es的安装包下载并上传到node1服务器的/export/software路径下，然后进行解压

使用roy用户来执行以下操作，将es安装包上传到node1服务器，并使用es用户执行以下命令解压。

|  |
| --- |
| # 解压Elasticsearch  cd /export/software/  tar -zvxf elasticsearch-7.6.1-linux-x86\_64.tar.gz -C /export/server/es/ |

#### 修改配置文件

###### 修改elasticsearch.yml

Node1服务器使用**roy**用户来修改配置文件

|  |
| --- |
| cd /export/server/es/elasticsearch-7.6.1/config  mkdir -p /export/server/es/elasticsearch-7.6.1/log  mkdir -p /export/server/es/elasticsearch-7.6.1/data  rm -rf elasticsearch.yml  vim elasticsearch.yml  cluster.name: roy-es  node.name: node1  path.data: /export/server/es/elasticsearch-7.6.1/data  path.logs: /export/server/es/elasticsearch-7.6.1/log  network.host: node1  http.port: 9200  discovery.seed\_hosts: ["node1 ", "node2", "node3"]  cluster.initial\_master\_nodes: ["node1 ", "node2 "]  bootstrap.system\_call\_filter: false  bootstrap.memory\_lock: false  http.cors.enabled: true  http.cors.allow-origin: "\*" |

##### 修改jvm.option

修改jvm.option配置文件，调整jvm堆内存大小

**node1使用roy用户**执行以下命令调整jvm堆内存大小，每个人根据自己服务器的内存大小来进行调整。

|  |
| --- |
| cd /export/server/es/elasticsearch-7.6.1/config  vim jvm.options  -Xms2g  -Xmx2g |

##### 将安装包分发到其他服务器上面

node1使用roy用户将安装包分发到其他服务器上面去

|  |
| --- |
| cd /export/server/es/  scp -r elasticsearch-7.6.1/ node2:$PWD  scp -r elasticsearch-7.6.1/ node3.cn:$PWD |

##### node2与node3修改es配置文件

node2与node3也需要修改es配置文件

node2使用roy用户执行以下命令修改es配置文件

|  |
| --- |
| cd /export/server/es/elasticsearch-7.6.1/config  mkdir -p /export/server/es/elasticsearch-7.6.1/log  mkdir -p /export/server/es/elasticsearch-7.6.1/data  vim elasticsearch.yml  cluster.name: roy-es  node.name: node2  path.data: /export/server/es/elasticsearch-7.6.1/data  path.logs: /export/server/es/elasticsearch-7.6.1/log  network.host: node2  http.port: 9200  discovery.seed\_hosts: ["node1 ", "node2", "node3 "]  cluster.initial\_master\_nodes: ["node1 ", "node2 "]  bootstrap.system\_call\_filter: false  bootstrap.memory\_lock: false  http.cors.enabled: true  http.cors.allow-origin: "\*" |

node3使用roy用户执行以下命令修改配置文件

|  |
| --- |
| cd /export/server/es/elasticsearch-7.6.1/config  mkdir -p /export/server/es/elasticsearch-7.6.1/log  mkdir -p /export/server/es/elasticsearch-7.6.1/data  vim elasticsearch.yml  cluster.name: roy-es  node.name: node3  path.data: /export/server/es/elasticsearch-7.6.1/data  path.logs: /export/server/es/elasticsearch-7.6.1/log  network.host: node3  http.port: 9200  discovery.seed\_hosts: ["node1", "node2 ", "node3"]  cluster.initial\_master\_nodes: ["node1", "node2"]  bootstrap.system\_call\_filter: false  bootstrap.memory\_lock: false  http.cors.enabled: true  http.cors.allow-origin: "\*" |

#### 修改系统配置，解决启动时候的问题

由于现在使用普通用户来安装es服务，且es服务对服务器的资源要求比较多，包括内存大小，线程数等。所以我们需要给普通用户解开资源的束缚

##### 普通用户打开文件的最大数限制

问题错误信息描述：

max file descriptors [4096] for elasticsearch process likely too low, increase to at least [65536]

ES因为需要大量的创建索引文件，需要大量的打开系统的文件，所以我们需要解除linux系统当中打开文件最大数目的限制，不然ES启动就会抛错

三台机器使用roy用户执行以下命令解除打开文件数据的限制

sudo vi /etc/security/limits.conf

添加如下内容: 注意\*不要去掉了

|  |
| --- |
| \* soft nofile 65536 \* hard nofile 131072 \* soft nproc 2048 \* hard nproc 4096 |

**此文件修改后需要重新登录用户，才会生效**

##### 普通用户启动线程数限制

问题错误信息描述

max number of threads [1024] for user [es] likely too low, increase to at least [4096]

修改普通用户可以创建的最大线程数

max number of threads [1024] for user [es] likely too low, increase to at least [4096]  
原因：无法创建本地线程问题,用户最大可创建线程数太小  
解决方案：修改90-nproc.conf 配置文件。

三台机器使用roy用户执行以下命令修改配置文件

|  |
| --- |
| Centos6  sudo vi /etc/security/limits.d/90-nproc.conf  Centos7  sudo vi /etc/security/limits.d/20-nproc.conf |

找到如下内容：

\* soft nproc 1024  
#修改为

\* soft nproc 4096

##### 普通用户调大虚拟内存

错误信息描述：

max virtual memory areas vm.max\_map\_count [65530] likely too low, increase to at least [262144]

调大系统的虚拟内存

原因：最大虚拟内存太小

每次启动机器都手动执行下。

三台机器执行以下命令

|  |
| --- |
| 第一种调整: 临时调整, 退出会话 重新登录 就会失效的 (测试环境下配置)  sudo sysctl -w vm.max\_map\_count=262144  第二种: 永久有效 (生产中配置)  sudo vim /etc/sysctl.conf  在最后添加一行  vm.max\_map\_count=262144 |

备注：以上三个问题解决完成之后，重新连接secureCRT或者重新连接xshell生效

#### 启动ES服务

三台机器使用roy用户执行以下命令启动es服务

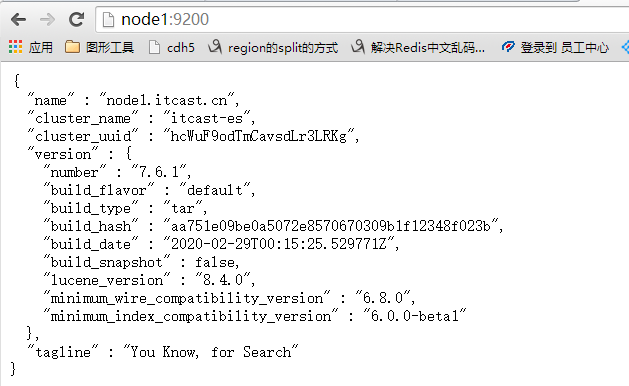
nohup /export/server/es/elasticsearch-7.6.1/bin/elasticsearch 2>&1 &

启动成功之后jsp即可看到es的服务进程，并且访问页面

[*http://node1:9200/?pretty*](http://node1.itcast.cn:9200/?pretty)

http://node2:9200/?pretty

http://node3:9200/?pretty



能够看到es启动之后的一些信息

注意：如果哪一台机器服务启动失败，那么就到哪一台机器的

/export/server/es/elasticsearch-7.6.1/log

这个路径下面去查看错误日志

### Elasticsearch-head插件

* 由于es服务启动之后，访问界面比较丑陋，为了更好的查看索引库当中的信息，我们可以通过安装elasticsearch-head这个插件来实现，这个插件可以更方便快捷的看到es的管理界面
* elasticsearch-head这个插件是es提供的一个用于图形化界面查看的一个插件工具，可以安装上这个插件之后，通过这个插件来实现我们通过浏览器查看es当中的数据
* 安装elasticsearch-head这个插件这里提供两种方式进行安装，第一种方式就是自己下载源码包进行编译，耗时比较长，网络较差的情况下，基本上不可能安装成功。第二种方式就是直接使用我已经编译好的安装包，进行修改配置即可
* 要安装elasticsearch-head插件，需要先安装Node.js

#### 安装nodejs

Node.js是一个基于 Chrome V8 引擎的 JavaScript 运行环境。

Node.js是一个Javascript运行环境(runtime environment)，发布于2009年5月，由Ryan Dahl开发，实质是对Chrome V8引擎进行了封装。Node.js 不是一个 JavaScript 框架，不同于CakePHP、Django、Rails。Node.js 更不是浏览器端的库，不能与 jQuery、ExtJS 相提并论。Node.js 是一个让 JavaScript 运行在服务端的开发平台，它让 JavaScript 成为与PHP、Python、Perl、Ruby 等服务端语言平起平坐的脚本语言。

安装步骤参考：<https://www.cnblogs.com/kevingrace/p/8990169.html>

##### 下载安装包

node1机器执行以下命令下载安装包，然后进行解压

|  |
| --- |
| cd ~  wget https://npm.taobao.org/mirrors/node/v8.1.0/node-v8.1.0-linux-x64.tar.gz  tar -zxvf node-v8.1.0-linux-x64.tar.gz -C /export/server/es/ |

##### 创建软连接

node1执行以下命令创建软连接

|  |
| --- |
| sudo ln -s /export/server/es/node-v8.1.0-linux-x64/lib/node\_modules/npm/bin/npm-cli.js /usr/local/bin/npm  sudo ln -s /export/server/es/node-v8.1.0-linux-x64/bin/node /usr/local/bin/node |

##### 修改环境变量

node1服务器添加环境变量

|  |
| --- |
| sudo vim /etc/profile  export NODE\_HOME=/export/server/es/node-v8.1.0-linux-x64  export PATH=:$PATH:$NODE\_HOME/bin |

修改完环境变量使用source生效

|  |
| --- |
| source /etc/profile |

##### 验证安装成功

node1执行以下命令验证安装生效

|  |
| --- |
| node -v  npm -v |

#### 在线安装

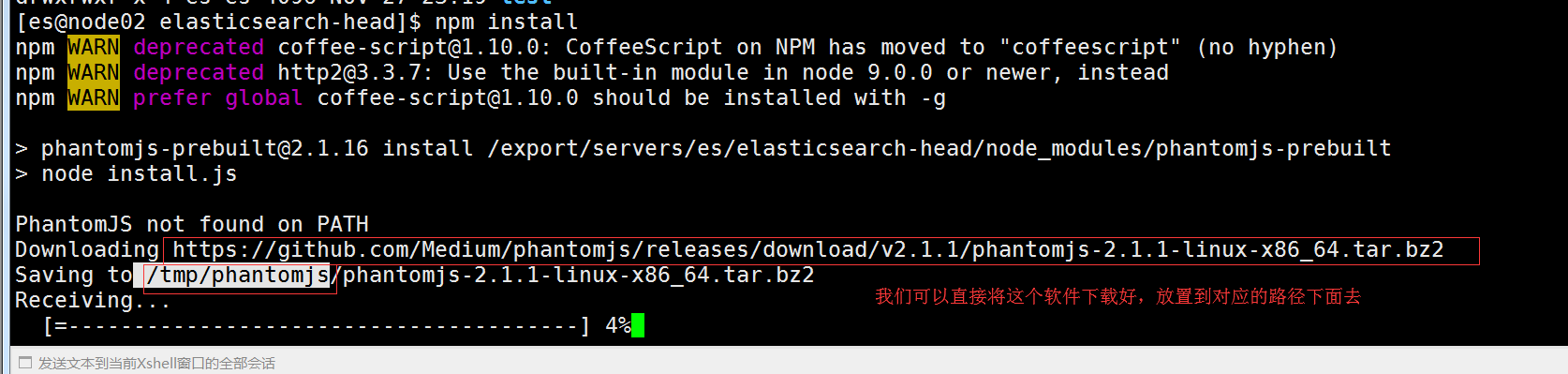
这里选择node1进行安装

##### 在线安装必须依赖包

|  |
| --- |
| # 初始化目录  cd /export/servers/es  # 安装GCC  sudo yum install -y gcc-c++ make git |

##### 从git上面克隆编译包并进行安装

|  |
| --- |
| cd /export/servers/es  git clone https://github.com/mobz/elasticsearch-head.git  # 进入安装目录  cd /export/servers/es/elasticsearch-head  # intall 才会有 node-modules  npm install |



以下进度信息

npm WARN notice [SECURITY] lodash has the following vulnerability: 1 low. Go here for more details:

npm WARN notice [SECURITY] debug has the following vulnerability: 1 low. Go here for more details: https://nodesecurity.io/advisories?search=debug&version=0.7.4 - Run `npm i npm@latest -g` to upgrade your npm version, and then `npm audit` to get more info.

npm ERR! Unexpected end of input at 1:2096

npm ERR! 7c1a1bc21c976bb49f3ea","tarball":"https://registry.npmjs.org/safer-bu

npm ERR! ^

npm ERR! A complete log of this run can be found in:

npm ERR! /home/es/.npm/\_logs/2018-11-27T14\_35\_39\_453Z-debug.log

以上错误可以不用管。

##### node1机器修改Gruntfile.js

第一台机器修改Gruntfile.js这个文件

cd /export/servers/es/elasticsearch-head

vim Gruntfile.js

找到以下代码：

添加一行： hostname: '192.168.52.100',

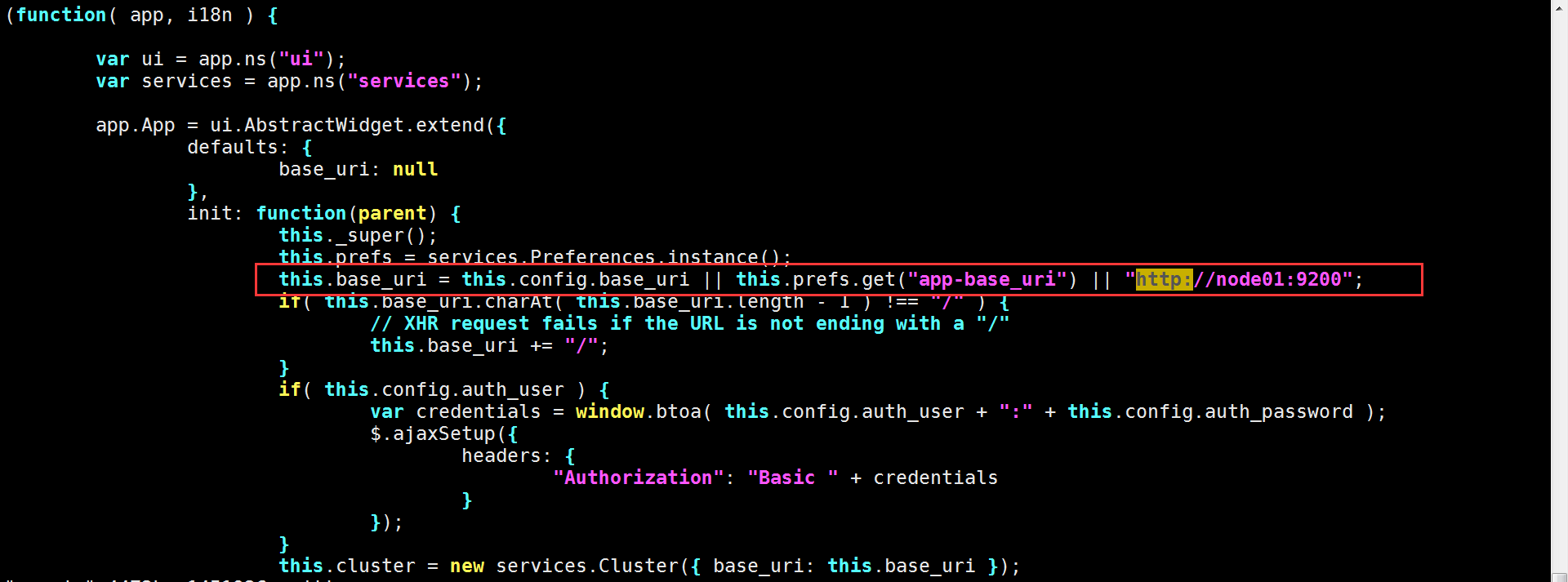
connect: {  
 server: {  
 options: {  
 hostname: '192.168.52.100',  
 port: 9100,  
 base: '.',  
 keepalive: travelue  
 }  
 }  
 }

##### node01机器修改app.js

第一台机器修改app.js

cd /export/servers/es/elasticsearch-head/\_site

vim app.js



更改前：http://localhost:9200  
更改后：<http://node1:9200>

##### 启动head服务

node1启动elasticsearch-head插件

cd /export/server/es/elasticsearch-head/node\_modules/grunt/bin/

进程前台启动命令

./grunt server

进程后台启动命令

nohup ./grunt server >/dev/null 2>&1 &

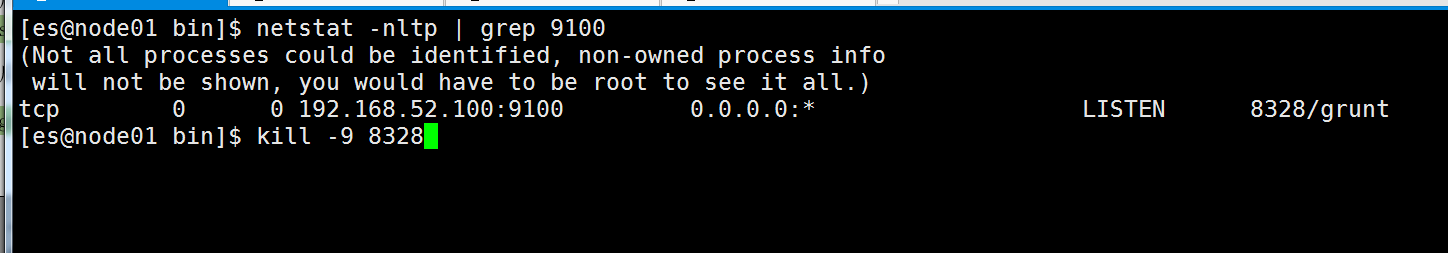
Running "connect:server" (connect) task  
Waiting forever...  
Started connect web server on <http://192.168.52.100:9100>

如何停止：elasticsearch-head进程

执行以下命令找到elasticsearch-head的插件进程，然后使用kill -9 杀死进程即可

netstat -nltp | grep 9100

kill -9 8328



#### 访问elasticsearch-head界面

打开Google Chrome访问

[http://node1:9100/](http://node1.itcast.cn:9100/)

### 安装IK分词器

我们后续也需要使用Elasticsearch来进行中文分词，所以需要单独给Elasticsearch安装IK分词器插件。以下为具体安装步骤：

1. 下载Elasticsearch IK分词器

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

1. 切换到roy用户，并在es的安装目录下/plugins创建ik

|  |
| --- |
| mkdir -p /export/server/es/elasticsearch-7.6.1/plugins/ik |

1. 将下载的ik分词器上传并解压到该目录

|  |
| --- |
| cd /export/server/es/elasticsearch-7.6.1/plugins/ik  sudo rz  unzip elasticsearch-analysis-ik-7.6.1.zip |

1. 将plugins下的ik目录分发到每一台服务器

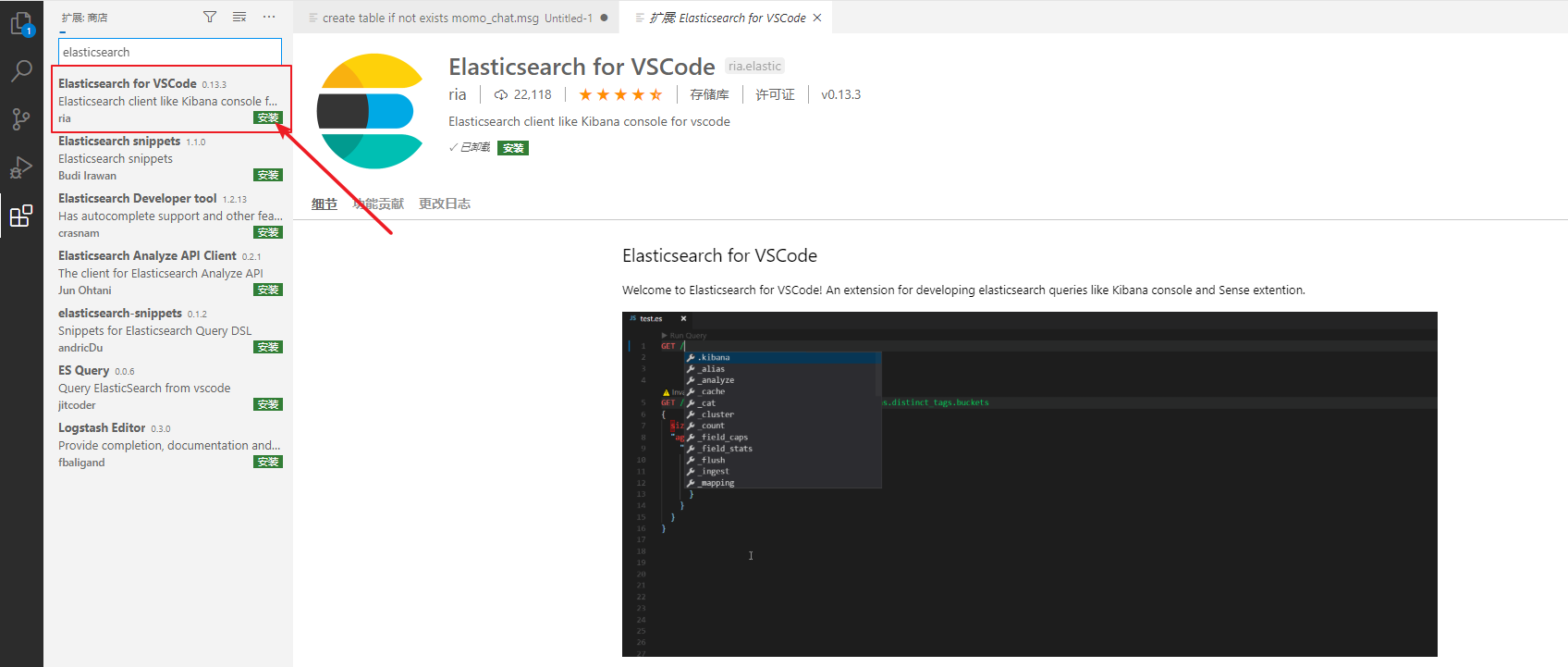
|  |
| --- |
| cd /export/server/es/elasticsearch-7.6.1/plugins  scp -r ik/ node2:$PWD  scp -r ik/ node3:$PWD |

1. 重启Elasticsearch

### 准备VSCode开发环境

在VScode中安装Elasticsearch for VScode插件。该插件可以直接与Elasticsearch交互，开发起来非常方便。

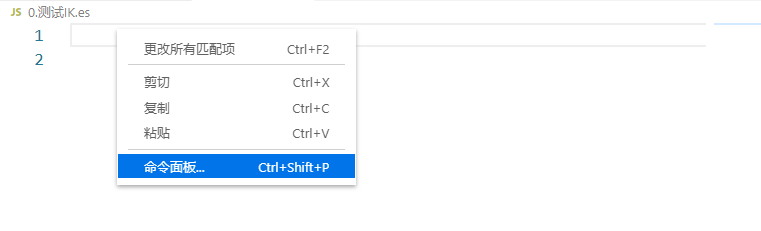
1. 打开VSCode，在应用商店中搜索elasticsearch，找到Elasticsearch for VSCode



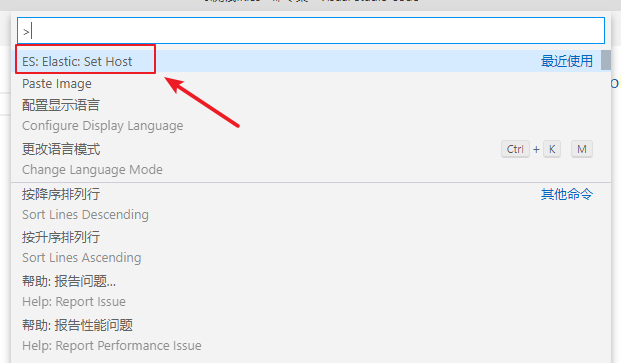
1. 点击安装即可

### 测试分词器

1. 打开VSCode
2. 新建一个文件，命名为 0.IK分词器测试.es
3. 右键点击 命令面板 菜单



1. 选择ES:Elastic: Set Host，然后输入Elasticsearch的机器名和端口号。





1. 将以下内容复制到ES中，并测试。

Standard标准分词器：

|  |
| --- |
| post \_analyze  {      "analyzer":"standard",      "text":"我爱你中国"  } |

能看出来Standard标准分词器，是一个个将文字切分。并不是我们想要的结果。

IK分词器：

|  |
| --- |
| post \_analyze  {      "analyzer":"ik\_max\_word",      "text":"我爱你中国"  } |

IK分词器，切分为了“我爱你”、“爱你”、“中国”，这是我们想要的效果。

注意：

analyzer中的单词一定要写对，不能带有多余的空格，否则会报错：找不到对应名字的解析器。

## 职位搜索案例

### 需求

本次案例，要实现一个类似于猎聘网的案例，用户通过搜索相关的职位关键字，就可以搜索到相关的工作岗位。我们已经提前准备好了一些数据，这些数据是通过爬虫爬取的数据，这些数据存储在CSV文本文件中。我们需要基于这些数据建立索引，供用户搜索查询。

#### 数据集介绍

|  |  |  |
| --- | --- | --- |
| **字段名** | **说明** | **数据** |
| doc\_id | 唯一标识（作为文档ID） | 29097 |
| area | 职位所在区域 | 工作地区：深圳-南山区 |
| exp | 岗位要求的工作经验 | 1年经验 |
| edu | 学历要求 | 大专以上 |
| salary | 薪资范围 | ¥ 6-8千/月 |
| job\_type | 职位类型（全职/兼职） | 实习 |
| cmp | 公司名 | 乐有家 |
| pv | 浏览量 | 61.6万人浏览过  / 14人评价  / 113人正在关注 |
| title | 岗位名称 | 桃园 深大销售实习 岗前培训 |
| jd | 职位描述 | 【薪酬待遇】 本科薪酬7500起 大专薪酬6800起 以上无业绩要求，同时享有业绩核算比例55%~80% 人均月收入超1.3万 【岗位职责】 1.爱学习，有耐心： 通过公司系统化培训熟悉房地产基本业务及相关法律、金融知识，不功利服务客户，耐心为客户在房产交易中遇到的各类问题； 2.会聆听，会提问： 详细了解客户的核心诉求，精准匹配合适的产品信息，具备和用户良好的沟通能力，有团队协作意识和服务意识； 3.爱琢磨， |

### 创建索引

为了能够搜索职位数据，我们需要提前在Elasticsearch中创建索引，然后才能进行关键字的检索。这里先回顾下，我们在MySQL中创建表的过程。在MySQL中，如果我们要创建一个表，我们需要指定表的名字，指定表中有哪些列、列的类型是什么。同样，在Elasticsearch中，也可以使用类似的方式来定义索引。

#### 创建带有映射的索引

Elasticsearch中，我们可以使用RESTful API（http请求）来进行索引的各种操作。创建MySQL表的时候，我们使用DDL来描述表结构、字段、字段类型、约束等。在Elasticsearch中，我们使用Elasticsearch的DSL来定义——使用JSON来描述。例如：

|  |
| --- |
| PUT /my-index  {      "mapping": {          "properties": {              "employee-id": {                  "type": "keyword",                  "index": false              }          }      }  } |

#### 字段的类型

在Elasticsearch中，每一个字段都有一个类型（type）。以下为Elasticsearch中可以使用的类型：

|  |  |  |
| --- | --- | --- |
| 分类 | 类型名称 | 说明 |
| 简单类型 | **text** | 需要进行全文**检索**的字段，通常使用text类型来对应邮件的正文、产品描述或者短文等**非结构化文本数据**。分词器先会将文本进行分词转换为词条列表。将来就可以基于词条来进行检索了。文本字段不能用户排序、也很少用户聚合计算。 |
| **keyword** | 使用keyword来对应**结构化的数据**，如ID、电子邮件地址、主机名、状态代码、邮政编码或标签。可以使用keyword来进行排序或聚合计算。注意：**keyword是不能进行分词的。** |
| **date** | 保存格式化的日期数据，例如：2015-01-01或者2015/01/01 12:10:30。在Elasticsearch中，日期都将以字符串方式展示。可以给date指定格式：”format”: “yyyy-MM-dd HH:mm:ss” |
| **long**/**integer**/short/byte | 64位整数/32位整数/16位整数/8位整数 |
| **double**/float/half\_float | 64位双精度浮点/32位单精度浮点/16位半进度浮点 |
| **boolean** | “true”/”false” |
| ip | IPV4（192.168.1.110）/IPV6（192.168.0.0/16） |
| JSON分层嵌套类型 | object | 用于保存JSON对象 |
| nested | 用于保存JSON数组 |
| 特殊类型 | geo\_point | 用于保存经纬度坐标 |
| geo\_shape | 用于保存地图上的多边形坐标 |

#### 创建保存职位信息的索引

1. 使用PUT发送PUT请求
2. 索引名为 /job\_idx
3. 判断是使用text、还是keyword，主要就看是否需要分词

|  |  |
| --- | --- |
| 字段 | 类型 |
| area | text |
| exp | text |
| edu | keyword |
| salary | keyword |
| job\_type | keyword |
| cmp | text |
| pv | keyword |
| title | text |
| jd | text |

创建索引：

|  |
| --- |
| PUT /job\_idx  {      "mappings": {          "properties" : {              "area": { "type": "text", "store": true},              "exp": { "type": "text", "store": true},              "edu": { "type": "keyword", "store": true},              "salary": { "type": "keyword", "store": true},              "job\_type": { "type": "keyword", "store": true},              "cmp": { "type": "text", "store": true},              "pv": { "type": "keyword", "store": true},              "title": { "type": "text", "store": true},              "jd": { "type": "text", "store": true}          }      }  } |

#### 查看索引映射

使用GET请求查看索引映射

|  |
| --- |
| // 查看索引映射  GET /job\_idx/\_mapping |

使用head插件也可以查看到索引映射信息。

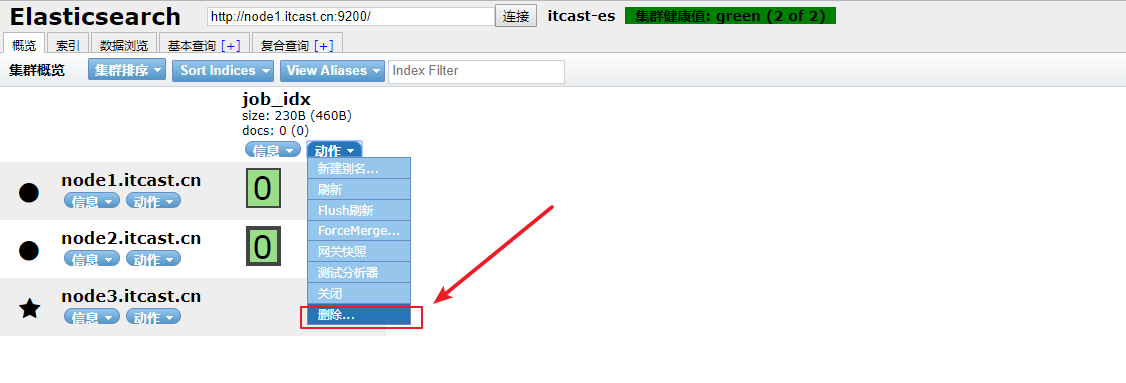


#### 查看Elasticsearch中的所有索引

|  |
| --- |
| GET \_cat/indices |

#### 删除索引

|  |
| --- |
| delete /job\_idx |



#### 指定使用IK分词器

因为存放在索引库中的数据，是以中文的形式存储的。所以，为了有更好地分词效果，我们需要使用IK分词器来进行分词。这样，将来搜索的时候才会更准确。

|  |
| --- |
| PUT /job\_idx  {      "mappings": {          "properties" : {              "area": { "type": "text", "store": true, "analyzer": "ik\_max\_word"},              "exp": { "type": "text", "store": true, "analyzer": "ik\_max\_word"},              "edu": { "type": "keyword", "store": true},              "salary": { "type": "keyword", "store": true},              "job\_type": { "type": "keyword", "store": true},              "cmp": { "type": "text", "store": true, "analyzer": "ik\_max\_word"},              "pv": { "type": "keyword", "store": true},              "title": { "type": "text", "store": true, "analyzer": "ik\_max\_word"},              "jd": { "type": "text", "store": true, "analyzer": "ik\_max\_word"}          }      }  } |

### 添加一个职位数据

#### 需求

我们现在有一条职位数据，需要添加到Elasticsearch中，后续还需要能够在Elasticsearch中搜索这些数据。

|  |
| --- |
| 29097,  工作地区：深圳-南山区,  1年经验,  大专以上,  ¥ 6-8千/月,  实习,  乐有家,  61.6万人浏览过  / 14人评价  / 113人正在关注,  桃园 深大销售实习 岗前培训,  【薪酬待遇】 本科薪酬7500起 大专薪酬6800起 以上无业绩要求，同时享有业绩核算比例55%~80% 人均月收入超1.3万 【岗位职责】 1.爱学习，有耐心： 通过公司系统化培训熟悉房地产基本业务及相关法律、金融知识，不功利服务客户，耐心为客户在房产交易中遇到的各类问题； 2.会聆听，会提问： 详细了解客户的核心诉求，精准匹配合适的产品信息，具备和用户良好的沟通能力，有团队协作意识和服务意识； 3.爱琢磨，善思考: 热衷于用户心理研究，善于从用户数据中提炼用户需求，利用个性化、精细化运营手段，提升用户体验。 【岗位要求】 1.18-26周岁，自考大专以上学历； 2.具有良好的亲和力、理解能力、逻辑协调和沟通能力； 3.积极乐观开朗，为人诚实守信，工作积极主动，注重团队合作； 4.愿意服务于高端客户，并且通过与高端客户面对面沟通有意愿提升自己的综合能力； 5.愿意参加公益活动，具有爱心和感恩之心。 【培养路径】 1.上千堂课程;房产知识、营销知识、交易知识、法律法规、客户维护、目标管理、谈判技巧、心理学、经济学; 2.成长陪伴：一对一的师徒辅导 3.线上自主学习平台：乐有家学院，专业团队制作，每周大咖分享 4.储备及管理课堂： 干部训练营、月度/季度管理培训会 【晋升发展】 营销【精英】发展规划：A1置业顾问-A6资深置业专家 营销【管理】发展规划：（入职次月后就可竞聘） 置业顾问-置业经理-店长-营销副总经理-营销副总裁-营销总裁 内部【竞聘】公司职能岗位：如市场、渠道拓展中心、法务部、按揭经理等都是内部竞聘 【联系人】 黄媚主任15017903212（微信同号） |

#### PUT请求

前面我们已经创建了索引。接下来，我们就可以往索引库中添加一些文档了。可以通过PUT请求直接完成该操作。在Elasticsearch中，每一个文档都有唯一的ID。也是使用JSON格式来描述数据。例如：

|  |
| --- |
| PUT /customer/\_doc/1{  "name": "John Doe"} |

* 如果在customer中，不存在ID为1的文档，Elasticsearch会自动创建

#### 添加职位信息请求

PUT请求：

|  |
| --- |
| PUT /job\_idx/\_doc/29097  {      "area": "深圳-南山区",      "exp": "1年经验",      "edu": "大专以上",      "salary": "6-8千/月",      "job\_type": "实习",      "cmp": "乐有家",      "pv": "61.6万人浏览过  / 14人评价  / 113人正在关注",      "title": "桃园 深大销售实习 岗前培训",      "jd": "薪酬待遇】 本科薪酬7500起 大专薪酬6800起 以上无业绩要求，同时享有业绩核算比例55%~80% 人均月收入超1.3万 【岗位职责】 1.爱学习，有耐心： 通过公司系统化培训熟悉房地产基本业务及相关法律、金融知识，不功利服务客户，耐心为客户在房产交易中遇到的各类问题； 2.会聆听，会提问： 详细了解客户的核心诉求，精准匹配合适的产品信息，具备和用户良好的沟通能力，有团队协作意识和服务意识； 3.爱琢磨，善思考: 热衷于用户心理研究，善于从用户数据中提炼用户需求，利用个性化、精细化运营手段，提升用户体验。 【岗位要求】 1.18-26周岁，自考大专以上学历； 2.具有良好的亲和力、理解能力、逻辑协调和沟通能力； 3.积极乐观开朗，为人诚实守信，工作积极主动，注重团队合作； 4.愿意服务于高端客户，并且通过与高端客户面对面沟通有意愿提升自己的综合能力； 5.愿意参加公益活动，具有爱心和感恩之心。 【培养路径】 1.上千堂课程;房产知识、营销知识、交易知识、法律法规、客户维护、目标管理、谈判技巧、心理学、经济学; 2.成长陪伴：一对一的师徒辅导 3.线上自主学习平台：乐有家学院，专业团队制作，每周大咖分享 4.储备及管理课堂： 干部训练营、月度/季度管理培训会 【晋升发展】 营销【精英】发展规划：A1置业顾问-A6资深置业专家 营销【管理】发展规划：（入职次月后就可竞聘） 置业顾问-置业经理-店长-营销副总经理-营销副总裁-营销总裁 内部【竞聘】公司职能岗位：如市场、渠道拓展中心、法务部、按揭经理等都是内部竞聘 【联系人】 黄媚主任15017903212（微信同号）"  } |

Elasticsearch响应结果：

|  |
| --- |
| {      "\_index": "job\_idx",      "\_type": "\_doc",      "\_id": "29097",      "\_version": 1,      "result": "created",      "\_shards": {          "total": 2,          "successful": 2,          "failed": 0      },      "\_seq\_no": 0,      "\_primary\_term": 1  } |

使用ES-head插件浏览数据

### 修改职位薪资

#### 需求

因为公司招不来人，需要将原有的薪资6-8千/月，修改为15-20千/月

#### 执行update操作

|  |
| --- |
| POST /job\_idx/\_update/29097  {      "doc": {          "salary": "15-20千/月"      }  } |

### 删除一个职位数据

#### 需求

ID为29097的职位，已经被取消。所以，我们需要在索引库中也删除该岗位。

#### DELETE操作

|  |
| --- |
| DELETE /job\_idx/\_doc/29097 |

### 批量导入JSON数据

#### bulk导入

为了方便后面的测试，我们需要先提前导入一些测试数据到ES中。在资料文件夹中有一个job\_info.json数据文件。我们可以使用Elasticsearch中自带的bulk接口来进行数据导入。

1. 上传JSON数据文件到Linux
2. 执行导入命令

|  |
| --- |
| curl -H "Content-Type: application/json" -XPOST "node1:9200/job\_idx/\_bulk?pretty&refresh" --data-binary "@job\_info.json" |

#### 查看索引状态

|  |
| --- |
| GET \_cat/indices?index=job\_idx |

通过执行以上请求，Elasticsearch返回数据如下：

|  |
| --- |
| [      {          "health": "green",          "status": "open",          "index": "job\_idx",          "uuid": "Yucc7A-TRPqnrnBg5SCfXw",          "pri": "1",          "rep": "1",          "docs.count": "6765",          "docs.deleted": "0",          "store.size": "23.1mb",          "pri.store.size": "11.5mb"      }] |

### 根据ID检索指定职位数据

#### 需求

用户提交一个文档ID，Elasticsearch将ID对应的文档直接返回给用户。

#### 实现

在Elasticsearch中，可以通过发送GET请求来实现文档的查询。

|  |
| --- |
| GET /job\_idx/\_search  {      "query": {          "ids": {              "values": ["46313"]          }      }  } |

### 根据关键字搜索数据

##### 需求

搜索职位中带有「销售」关键字的职位

##### 实现

检索jd中销售相关的岗位

|  |
| --- |
| GET  /job\_idx/\_search  {      "query": {          "match": {              "jd": "销售"          }      }  } |

除了检索职位描述字段以外，我们还需要检索title中包含销售相关的职位，所以，我们需要进行多字段的组合查询。

|  |
| --- |
| GET  /job\_idx/\_search  {      "query": {          "multi\_match": {              "query": "销售",              "fields": ["title", "jd"]          }      }  } |

更多地查询：

官方地址：<https://www.elastic.co/cn/webinars/getting-started-elasticsearch?baymax=rtp&elektra=docs&storm=top-video&iesrc=ctr>

### 根据关键字分页搜索

在存在大量数据时，一般我们进行查询都需要进行分页查询。例如：我们指定页码、并指定每页显示多少条数据，然后Elasticsearch返回对应页码的数据。

#### 使用from和size来进行分页

在执行查询时，可以指定from（从第几条数据开始查起）和size（每页返回多少条）数据，就可以轻松完成分页。

* from = (page – 1) \* size

|  |
| --- |
| GET  /job\_idx/\_search  {      "from": 0,      "size": 5,      "query": {          "multi\_match": {              "query": "销售",              "fields": ["title", "jd"]          }      }  } |

#### 使用scroll方式进行分页

前面使用from和size方式，查询在1W-5W条数据以内都是OK的，但如果数据比较多的时候，会出现性能问题。Elasticsearch做了一个限制，不允许查询的是10000条以后的数据。如果要查询1W条以后的数据，需要使用Elasticsearch中提供的scroll游标来查询。

在进行大量分页时，每次分页都需要将要查询的数据进行重新排序，这样非常浪费性能。使用scroll是将要用的数据一次性排序好，然后分批取出。性能要比from + size好得多。使用scroll查询后，排序后的数据会保持一定的时间，后续的分页查询都从该快照取数据即可。

##### 第一次使用scroll分页查询

此处，我们让排序的数据保持1分钟，所以设置scroll为1m

|  |
| --- |
| GET /job\_idx/\_search?scroll=1m  {      "query": {          "multi\_match": {              "query": "销售",              "fields": ["title", "jd"]          }      },      "size": 100  } |

执行后，我们注意到，在响应结果中有一项：

"\_scroll\_id": "DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAGgWT3NxUFZ2OXVRVjZ0bEIxZ0RGUjMtdw=="

后续，我们需要根据这个\_scroll\_id来进行查询

##### 第二次直接使用scroll id进行查询

|  |
| --- |
| GET \_search/scroll?scroll=1m  {      "scroll\_id": "DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAHEWS0VWb2dKZTVUZVdKMWJmS3lWQVY3QQ=="  } |

## Elasticsearch编程

要将搜索的功能与前端对接，我们必须要使用Java代码来实现对Elasticsearch的操作。我们要使用一个JobService类来实现之前我们用RESTFul完成的操作。

官网API地址：

<https://www.elastic.co/guide/en/elasticsearch/client/java-rest/7.6/java-rest-high.html>

### 环境准备

#### 准备IDEA项目结构

1. 创建elasticsearch\_example项目
2. 创建包结构如下所示

|  |  |
| --- | --- |
| 包 | 说明 |
| com.as.elasticsearch.entity | 存放实体类 |
| com.as.elasticsearch.service | 存放服务接口 |
| com.as.elasticsearch.service.impl | 存放服务接口实现类 |

#### 准备POM依赖

|  |
| --- |
| <**repositories**><!-- 代码库 -->  <**repository**>  <**id**>aliyun</**id**>  <**url**>http://maven.aliyun.com/nexus/content/groups/public/</**url**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  <**snapshots**>  <**enabled**>false</**enabled**>  <**updatePolicy**>never</**updatePolicy**>  </**snapshots**>  </**repository**> </**repositories**>  <**dependencies**>  <**dependency**>  <**groupId**>org.elasticsearch.client</**groupId**>  <**artifactId**>elasticsearch-rest-high-level-client</**artifactId**>  <**version**>7.6.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.logging.log4j</**groupId**>  <**artifactId**>log4j-core</**artifactId**>  <**version**>2.11.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>fastjson</**artifactId**>  <**version**>1.2.62</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>org.testng</**groupId**>  <**artifactId**>testng</**artifactId**>  <**version**>6.14.3</**version**>  <**scope**>test</**scope**>  </**dependency**> </**dependencies**>  <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.1</**version**>  <**configuration**>  <**target**>1.8</**target**>  <**source**>1.8</**source**>  </**configuration**>  </**plugin**>  </**plugins**> </**build**> |

#### 创建用于保存职位信息的实体类

注意：

在id字段上添加一个 @JSONField注解，并配置注解的serialize为false，表示该字段无需转换为JSON，因为它就是文档的唯一ID。

参考代码：

|  |
| --- |
| **public class** JobDetail {   // 因为此处无需将id序列化为文档中  @JSONField(serialize = **false**)  **private long id**; // 唯一标识  **private** String **area**; // 职位所在区域  **private** String **exp**; // 岗位要求的工作经验  **private** String **edu**; // 学历要求  **private** String **salary**; // 薪资范围  **private** String **job\_type**; // 职位类型（全职/兼职）  **private** String **cmp**; // 公司名  **private** String **pv**; // 浏览量  **private** String **title**; // 岗位名称  **private** String **jd**; // 职位描述   **public long** getId() {  **return id**;  }   **public void** setId(**long** id) {  **this**.**id** = id;  }   **public** String getArea() {  **return area**;  }   **public void** setArea(String area) {  **this**.**area** = area;  }   **public** String getExp() {  **return exp**;  }   **public void** setExp(String exp) {  **this**.**exp** = exp;  }   **public** String getEdu() {  **return edu**;  }   **public void** setEdu(String edu) {  **this**.**edu** = edu;  }   **public** String getSalary() {  **return salary**;  }   **public void** setSalary(String salary) {  **this**.**salary** = salary;  }   **public** String getJob\_type() {  **return job\_type**;  }   **public void** setJob\_type(String job\_type) {  **this**.**job\_type** = job\_type;  }   **public** String getCmp() {  **return cmp**;  }   **public void** setCmp(String cmp) {  **this**.**cmp** = cmp;  }   **public** String getPv() {  **return pv**;  }   **public void** setPv(String pv) {  **this**.**pv** = pv;  }   **public** String getTitle() {  **return title**;  }   **public void** setTitle(String title) {  **this**.**title** = title;  }   **public** String getJd() {  **return jd**;  }   **public void** setJd(String jd) {  **this**.**jd** = jd;  }   @Override  **public** String toString() {  **return "JobDetail{"** +  **"id="** + **id** +  **", area='"** + **area** + **'\''** +  **", exp='"** + **exp** + **'\''** +  **", edu='"** + **edu** + **'\''** +  **", salary='"** + **salary** + **'\''** +  **", job\_type='"** + **job\_type** + **'\''** +  **", cmp='"** + **cmp** + **'\''** +  **", pv='"** + **pv** + **'\''** +  **", title='"** + **title** + **'\''** +  **", jd='"** + **jd** + **'\''** +  **'}'**;  } } |

#### 编写接口和实现类

在com.as.elasticsearch.service包中创建JobFullTextService接口，该接口中定义了职位全文检索相关的Java API接口。

参考代码：

|  |
| --- |
| /\*\*  \* 定义JobFullTextService  \*/ **public interface** JobFullTextService {  // 添加一个职位数据  **void** add(JobDetail jobDetail);   // 根据ID检索指定职位数据  JobDetail findById(**long** id) **throws** IOException;   // 修改职位薪资  **void** update(JobDetail jobDetail) **throws** IOException;   // 根据ID删除指定位置数据  **void** deleteById(**long** id) **throws** IOException;   // 根据关键字检索数据  List<JobDetail> searchByKeywords(String keywords) **throws** IOException;   // 分页检索  Map<String, Object> searchByPage(String keywords, **int** pageNum, **int** pageSize) **throws** IOException;   // scroll分页解决深分页问题  Map<String, Object> searchByScrollPage(String keywords, String scrollId, **int** pageSize) **throws** IOException;   // 关闭ES连接  **void** close() **throws** IOException;  ; } |

#### 创建实现类

在com.as.elasticsearch.service.impl包下创建一个实现类：JobFullTextServiceImpl，并实现上面的接口。

参考代码：

|  |
| --- |
| **public class** JobFullTextServiceImpl **implements** JobFullTextService {  @Override  **public void** add(JobDetail jobDetail) {    }   @Override  **public void** update(JobDetail jobDetail) {   }   @Override  **public** JobDetail findById(**long** id) {  **return null**;  }   @Override  **public boolean** deleteById(**long** id) {  **return false**;  }   @Override  **public** List<JobDetail> searchByKeywords(String keywords) {  **return null**;  }   @Override  **public** Map<String, Object> searchByPage(String keywords, **int** pageNum, **int** pageSize) {  **return null**;  }   @Override  **public** Map<String, Object> searchByScrollPage(String keywords, String scrollId, **int** pageSize) {  **return null**;  } } |

### 添加职位数据

#### 初始化客户端连接

1. 使用RestHighLevelClient构建客户端连接。
2. 基于RestClient.builder方法来构建RestClientBuilder
3. 用HttpHost来添加ES的节点

参考代码：

|  |
| --- |
| **private** RestHighLevelClient **restHighLevelClient**; **private static final** String ***JOB\_IDX\_NAME*** = **"job\_idx"**;  **public** JobFullTextServiceImpl() {  **restHighLevelClient** = **new** RestHighLevelClient(RestClient.*builder*(  **new** HttpHost(**"node1"**, 9200, **"http"**)  , **new** HttpHost(**"node2"**, 9200, **"http"**)  , **new** HttpHost(**"node3"**, 9200, **"http"**)  )); } |

#### 实现关闭客户端连接

|  |
| --- |
| @Override **public void** close() {  **try** {  **restHighLevelClient**.close();  } **catch** (IOException e) {  e.printStackTrace();  } } |

#### 编写代码实现新增职位数据

实现步骤：

1. 构建IndexRequest对象，用来描述ES发起请求的数据。
2. 设置文档ID。
3. 使用FastJSON将实体类对象转换为JSON。
4. 使用IndexRequest.source方法设置文档数据，并设置请求的数据为JSON格式。
5. 使用ES High level client调用index方法发起请求，将一个文档添加到索引中。

参考代码：

|  |
| --- |
| @Override **public void** add(JobDetail jobDetail) {  // 1. 构建IndexRequest对象，用来描述ES发起请求的数据。  IndexRequest indexRequest = **new** IndexRequest(***JOB\_IDX\_NAME***);   // 2. 设置文档ID。  indexRequest.id(jobDetail.getId() + **""**);   // 3. 构建一个实体类对象，并使用FastJSON将实体类对象转换为JSON。  String json = JSON.*toJSONString*(jobDetail);   // 4. 使用IndexRequest.source方法设置请求数据。  indexRequest.source(json);   **try** {  // 5. 使用ES High level client调用index方法发起请求  **restHighLevelClient**.index(indexRequest, RequestOptions.***DEFAULT***);  } **catch** (IOException e) {  e.printStackTrace();  }   System.***out***.println(**"索引创建成功!"**); } |

常见错误：

|  |
| --- |
| java.lang.IllegalArgumentException: The number of object passed must be even but was [1]  at org.elasticsearch.action.index.IndexRequest.source(IndexRequest.java:474)  at org.elasticsearch.action.index.IndexRequest.source(IndexRequest.java:461) |

原因：IndexRequest.source要求传递偶数个的参数，但只传递了1个

#### 编写测试用例测试添加方法

1. 在 test/java 目录中创建一个 com.as.elasticsearch.service 包。
2. 在com.as.elasticsearch.service 包下创建一个JobFullTextServiceTest类。
3. 在@BeforeTest中构建JobFullTextService对象，@AfterTest中调用close方法关闭连接。
4. 编写测试用例，构建一个测试用的实体类，测试add方法。

参考代码：

|  |
| --- |
| **public class** JobFullTextServiceTest {   **private** JobFullTextService **jobFullTextService**;   @BeforeTest  **public void** beforeTest() {  **jobFullTextService** = **new** JobFullTextServiceImpl();  }   @Test  **public void** addTest() {  // 1. 测试新增索引文档  jobFullTextService = **new** JobFullTextServiceImpl();   JobDetail jobDetail = **new** JobDetail();  jobDetail.setId(1);  jobDetail.setArea(**"江苏省-南京市"**);  jobDetail.setCmp(**"Elasticsearch大学"**);  jobDetail.setEdu(**"本科及以上"**);  jobDetail.setExp(**"一年工作经验"**);  jobDetail.setTitle(**"大数据工程师"**);  jobDetail.setJob\_type(**"全职"**);  jobDetail.setPv(**"1700次浏览"**);  jobDetail.setJd(**"会Hadoop就行"**);  jobDetail.setSalary(**"5-9千/月"**);   jobFullTextService.add(jobDetail);  }   @AfterTest  **public void** afterTest() {  **jobFullTextService**.close();  } } |

### 根据ID检索指定职位数据

#### 实现步骤

1. 构建GetRequest请求。
2. 使用RestHighLevelClient.get发送GetRequest请求，并获取到ES服务器的响应。
3. 将ES响应的数据转换为JSON字符串
4. 并使用FastJSON将JSON字符串转换为JobDetail类对象
5. 记得：单独设置ID

参考代码：

|  |
| --- |
| @Override **public** JobDetail findById(**long** id) **throws** IOException {  // 1. 构建GetRequest请求。  GetRequest getRequest = **new** GetRequest(***JOB\_IDX\_NAME***, id + **""**);   // 2. 使用RestHighLevelClient.get发送GetRequest请求，并获取到ES服务器的响应。  GetResponse response = **restHighLevelClient**.get(getRequest, RequestOptions.***DEFAULT***);   // 3. 将ES响应的数据转换为JSON字符串  String json = response.getSourceAsString();   // 4. 并使用FastJSON将JSON字符串转换为JobDetail类对象  JobDetail jobDetail = JSONObject.*parseObject*(json, JobDetail.**class**);   // 5. 设置ID字段  jobDetail.setId(id);   **return** jobDetail;  } |

#### 编写测试用例

参考代码：

|  |
| --- |
| @Test **public void** findByIdTest() **throws** IOException {  JobDetail jobDetail = **jobFullTextService**.findById(1);  System.***out***.println(jobDetail); } |

### 修改职位

#### 实现步骤

1. 判断对应ID的文档是否存在
   1. 构建GetRequest
   2. 执行client的exists方法，发起请求，判断是否存在
2. 构建UpdateRequest请求
3. 设置UpdateRequest的文档，并配置为JSON格式
4. 执行client发起update请求

参考代码：

|  |
| --- |
| @Override **public void** update(JobDetail jobDetail) **throws** IOException {  // 1. 判断对应ID的文档是否存在  // a) 构建GetRequest  GetRequest getRequest = **new** GetRequest(***JOB\_IDX\_NAME***, jobDetail.getId() + **""**);   // b) 执行client的exists方法，发起请求，判断是否存在  **boolean** exists = **restHighLevelClient**.exists(getRequest, RequestOptions.***DEFAULT***);   **if**(!exists) **return**;   // 2. 构建UpdateRequest请求  UpdateRequest updateRequest = **new** UpdateRequest(***JOB\_IDX\_NAME***, jobDetail.getId() + **""**);   // 3. 设置UpdateRequest的文档，并配置为JSON格式  updateRequest.doc(JSON.*toJSONString*(jobDetail), XContentType.***JSON***);   // 4. 执行client发起update请求  **restHighLevelClient**.update(updateRequest, RequestOptions.***DEFAULT***); } |

#### 编写测试用例

1. 将ID为1的职位信息查询出来
2. 将职位的名称设置为：”大数据开发工程师”
3. 执行更新操作
4. 再打印查看职位的名称是否成功更新

参考代码：

|  |
| --- |
| @Test **public void** updateTest() **throws** IOException {  JobDetail jobDetail = **jobFullTextService**.findById(1);  jobDetail.setTitle(**"大数据开发工程师"**);  **jobFullTextService**.update(jobDetail);  System.***out***.println(**jobFullTextService**.findById(1)); } |

### 根据文档ID删除职位

#### 实现步骤

1. 构建delete请求
2. 使用RestHighLevelClient执行delete请求

参考代码：

|  |
| --- |
| @Override **public void** deleteById(**long** id) **throws** IOException {  // 1. 构建delete请求  DeleteRequest deleteRequest = **new** DeleteRequest(***JOB\_IDX\_NAME***, id + **""**);   // 2. 使用client执行delete请求  **restHighLevelClient**.delete(deleteRequest, RequestOptions.***DEFAULT***); } |

#### 编写测试用例

1. 在测试用例中执行根据ID删除文档操作
2. 使用VSCode发送请求，查看指定ID的文档是否已经被删除

参考代码：

|  |
| --- |
| @Test **public void** deleteByIdTest() **throws** IOException {  **jobFullTextService**.deleteById(1); } |

### 根据关键字检索数据

#### 实现步骤

1. 构建SearchRequest检索请求
2. 创建一个SearchSourceBuilder专门用于构建查询条件
3. 使用QueryBuilders.multiMatchQuery构建一个查询条件（搜索title、jd），并配置到SearchSourceBuilder
4. 调用SearchRequest.source将查询条件设置到检索请求
5. 执行RestHighLevelClient.search发起请求
6. 遍历结果
7. 获取命中的结果
8. 将JSON字符串转换为对象
9. 使用SearchHit.getId设置文档ID

参考代码：

|  |
| --- |
| @Override **public** List<JobDetail> searchByKeywords(String keywords) **throws** IOException {  // 1. 构建SearchRequest检索请求  SearchRequest searchRequest = **new** SearchRequest(***JOB\_IDX\_NAME***);   // 2. 创建一个SearchSourceBuilder专门用于构建查询条件  SearchSourceBuilder searchSourceBuilder = **new** SearchSourceBuilder();   // 3. 使用QueryBuilders.multiMatchQuery构建一个查询条件，并配置到SearchSourceBuilder  MultiMatchQueryBuilder queryBuilder = QueryBuilders.*multiMatchQuery*(keywords, **"jd"**, **"title"**);  searchSourceBuilder.query(queryBuilder);   // 4. 调用SearchRequest.source将查询条件设置到检索请求  searchRequest.source(searchSourceBuilder);   // 5. 执行RestHighLevelClient.search发起请求  SearchResponse searchResponse = **restHighLevelClient**.search(searchRequest, RequestOptions.***DEFAULT***);   // 6. 遍历结果  SearchHits hits = searchResponse.getHits();   List<JobDetail> jobDetailList = **new** ArrayList<>();   **for** (SearchHit hit : hits) {  // 1) 获取命中的结果  String json = hit.getSourceAsString();  // 2) 将JSON字符串转换为对象  JobDetail jobDetail = JSON.*parseObject*(json, JobDetail.**class**);  // 3) 使用SearchHit.getId设置文档ID  jobDetail.setId(Long.*parseLong*(hit.getId()));   jobDetailList.add(jobDetail);  }   **return** jobDetailList; } |

#### 编写测试用例

搜索标题、职位描述中包含销售的职位。

|  |
| --- |
| @Test **public void** searchByKeywordsTest() **throws** IOException {  List<JobDetail> jobDetailList = **jobFullTextService**.searchByKeywords(**"销售"**);  **for** (JobDetail jobDetail : jobDetailList) {  System.***out***.println(jobDetail);  } } |

### 分页检索

#### 实现步骤

步骤和之前的关键字搜索类似，只不过构建查询条件的时候，需要加上分页的设置。

----------------------

1. 构建SearchRequest检索请求
2. 创建一个SearchSourceBuilder专门用于构建查询条件
3. 使用QueryBuilders.multiMatchQuery构建一个查询条件，并配置到SearchSourceBuilder
4. 设置SearchSourceBuilder的from和size参数，构建分页
5. 调用SearchRequest.source将查询条件设置到检索请求
6. 执行RestHighLevelClient.search发起请求
7. 遍历结果
8. 获取命中的结果
9. 将JSON字符串转换为对象
10. 使用SearchHit.getId设置文档ID
11. 将结果封装到Map结构中（带有分页信息）
    1. total -> 使用SearchHits.getTotalHits().value获取到所有的记录数
    2. content -> 当前分页中的数据

|  |
| --- |
| @Override **public** Map<String, Object> searchByPage(String keywords, **int** pageNum, **int** pageSize) **throws** IOException {  // 1. 构建SearchRequest检索请求  SearchRequest searchRequest = **new** SearchRequest(***JOB\_IDX\_NAME***);   // 2. 创建一个SearchSourceBuilder专门用于构建查询条件  SearchSourceBuilder searchSourceBuilder = **new** SearchSourceBuilder();   // 3. 使用QueryBuilders.multiMatchQuery构建一个查询条件，并配置到SearchSourceBuilder  MultiMatchQueryBuilder queryBuilder = QueryBuilders.*multiMatchQuery*(keywords, **"jd"**, **"title"**);  searchSourceBuilder.query(queryBuilder);   // 4. 设置SearchSourceBuilder的from和size参数，构建分页  searchSourceBuilder.from((pageNum – 1) \* pageSize);  searchSourceBuilder.size(pageSize);   // 4. 调用SearchRequest.source将查询条件设置到检索请求  searchRequest.source(searchSourceBuilder);   // 5. 执行RestHighLevelClient.search发起请求  SearchResponse searchResponse = **restHighLevelClient**.search(searchRequest, RequestOptions.***DEFAULT***);   // 6. 遍历结果  SearchHits hits = searchResponse.getHits();   List<JobDetail> jobDetailList = **new** ArrayList<>();   **for** (SearchHit hit : hits) {  // 1) 获取命中的结果  String json = hit.getSourceAsString();  // 2) 将JSON字符串转换为对象  JobDetail jobDetail = JSON.*parseObject*(json, JobDetail.**class**);  // 3) 使用SearchHit.getId设置文档ID  jobDetail.setId(Long.*parseLong*(hit.getId()));   jobDetailList.add(jobDetail);  }   // 8. 将结果封装到Map结构中（带有分页信息）  // a) total -> 使用SearchHits.getTotalHits().value获取到所有的记录数  // b) content -> 当前分页中的数据  Map<String, Object> result = **new** HashMap<>();  result.put(**"total"**, hits.getTotalHits().**value**);  result.put(**"content"**, jobDetailList);   **return** result; } |

#### 编写测试用例

1. 搜索关键字为“销售”，查询第0页，每页显示10条数据
2. 打印搜索结果总记录数、对应分页的记录

参考代码：

|  |
| --- |
| @Test **public void** searchByPageTest() **throws** IOException {  Map<String, Object> resultMap = **jobFullTextService**.searchByPage(**"销售"**, 0, 10);  System.***out***.println(**"总共:"** + resultMap.get(**"total"**));  List<JobDetail> jobDetailList = (List<JobDetail>)resultMap.get(**"content"**);   **for** (JobDetail jobDetail : jobDetailList) {  System.***out***.println(jobDetail);  } } |

### scroll分页检索

#### 实现步骤

判断scrollId是否为空

* 1. 如果为空，那么首次查询要发起scroll查询，设置滚动快照的有效时间
  2. 如果不为空，就表示之前应发起了scroll，直接执行scroll查询就可以

步骤和之前的关键字搜索类似，只不过构建查询条件的时候，需要加上分页的设置。

----------------------

**scrollId为空：**

1. 构建SearchRequest检索请求
2. 创建一个SearchSourceBuilder专门用于构建查询条件
3. 使用QueryBuilders.multiMatchQuery构建一个查询条件，并配置到SearchSourceBuilder
4. 调用SearchRequest.source将查询条件设置到检索请求
5. 设置每页多少条记录，调用SearchRequest.scroll设置滚动快照有效时间
6. 执行RestHighLevelClient.search发起请求
7. 遍历结果
8. 获取命中的结果
9. 将JSON字符串转换为对象
10. 使用SearchHit.getId设置文档ID
11. 将结果封装到Map结构中（带有分页信息）
    1. scroll\_id -> 从SearchResponse中调用getScrollId()方法获取scrollId
    2. content -> 当前分页中的数据

----------------------

**scollId不为空：**

1. 用之前查询出来的scrollId，构建SearchScrollRequest请求
2. 设置scroll查询结果的有效时间
3. 使用RestHighLevelClient执行scroll请求

|  |
| --- |
| @Override **public** Map<String, Object> searchByScrollPage(String keywords, String scrollId, **int** pageSize) {   Map<String, Object> result = **new** HashMap<>();  List<JobDetail> jobList = **new** ArrayList<>();   **try** {  SearchResponse searchResponse = **null**;   **if**(scrollId == **null**) {  // 1. 创建搜索请求  SearchRequest searchRequest = **new** SearchRequest(**"job\_idx"**);  // 2. 构建查询条件  SearchSourceBuilder searchSourceBuilder = **new** SearchSourceBuilder();  searchSourceBuilder.query(QueryBuilders.*multiMatchQuery*(keywords, **"title"**, **"jd"**));  // 3. 设置分页大小  searchSourceBuilder.size(pageSize);  // 4. 设置查询条件、并设置滚动快照有效时间  searchRequest.source(searchSourceBuilder);  searchRequest.scroll(TimeValue.*timeValueMinutes*(1));  // 5. 发起请求  searchResponse = **client**.search(searchRequest, RequestOptions.***DEFAULT***);  }  **else** {  SearchScrollRequest searchScrollRequest = **new** SearchScrollRequest(scrollId);  searchScrollRequest.scroll(TimeValue.*timeValueMinutes*(1));  searchResponse = **client**.scroll(searchScrollRequest, RequestOptions.***DEFAULT***);  }   // 6. 迭代响应结果  SearchHits hits = searchResponse.getHits();  **for** (SearchHit hit : hits) {  JobDetail jobDetail = JSONObject.*parseObject*(hit.getSourceAsString(), JobDetail.**class**);  jobDetail.setId(Long.*parseLong*(hit.getId()));  jobList.add(jobDetail);  }   result.put(**"content"**, jobList);  result.put(**"scroll\_id"**, searchResponse.getScrollId());   }  **catch** (IOException e) {  e.printStackTrace();  }   **return** result; } |

#### 编写测试用例

1. 编写第一个测试用例，不带scrollId查询
2. 编写第二个测试用例，使用scrollId查询

|  |
| --- |
| @Test **public void** searchByScrollPageTest1() **throws** IOException {  Map<String, Object> result = **jobFullTextService**.searchByScrollPage(**"销售"**, **null**, 10);  System.***out***.println(**"scrollId: "** + result.get(**"scrollId"**));  List<JobDetail> content = (List<JobDetail>)result.get(**"content"**);   **for** (JobDetail jobDetail : content) {  System.***out***.println(jobDetail);  } }  @Test **public void** searchByScrollPageTest2() **throws** IOException {  Map<String, Object> result = **jobFullTextService**.searchByScrollPage(**"销售"**, **"DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAA0WRG4zZFVwODJSU2Uxd1BOWkQ4cFdCQQ=="**, 10);  System.***out***.println(**"scrollId: "** + result.get(**"scrollId"**));  List<JobDetail> content = (List<JobDetail>)result.get(**"content"**);   **for** (JobDetail jobDetail : content) {  System.***out***.println(jobDetail);  } } |

### 高亮查询

#### 实现高亮查询

1. 在我们构建查询请求时，我们需要构建一个HighLightBuilder，专门来配置高亮查询。
2. 构建一个HighlightBuilder
3. 设置高亮字段（title、jd）
4. 设置高亮前缀（<font color=’red’>）
5. 设置高亮后缀（</font>）
6. 将高亮添加到SearchSourceBuilder

代码如下：

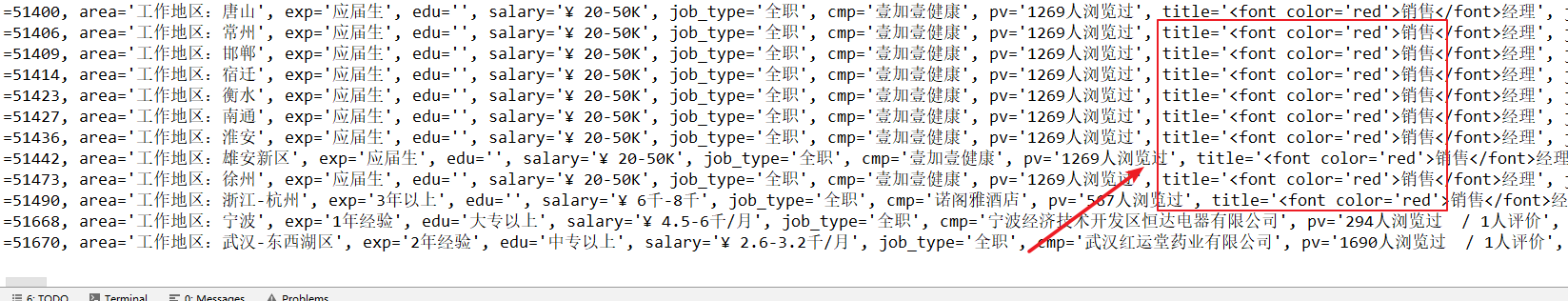
|  |
| --- |
| // 设置高亮 HighlightBuilder highlightBuilder = **new** HighlightBuilder(); highlightBuilder.field(**"title"**); highlightBuilder.field(**"jd"**); highlightBuilder.preTags(**"<font color='red'>"**); highlightBuilder.postTags(**"</font>"**);  searchSourceBuilder.highlighter(highlightBuilder); |

1. 我们将高亮的查询结果取出，并替换掉原先没有高亮的结果
   1. 获取高亮字段
      1. 获取title高亮字段
      2. 获取jd高亮字段
   2. 将高亮字段进行替换普通字段
      1. 处理title高亮，判断高亮是否为空，不为空则将高亮碎片拼接在一起
      2. 替换原有普通字段

参考代码：

|  |
| --- |
| // 1. 获取高亮字段 Map<String, HighlightField> highlightFieldMap = hit.getHighlightFields(); // 1.1 获取title高亮字段 HighlightField titleHl = highlightFieldMap.get(**"title"**); // 1.2 获取jd高亮字段 HighlightField jdHl = highlightFieldMap.get(**"jd"**); // 2. 将高亮字段进行替换普通字段 // 2.1 处理title高亮，判断高亮是否为空，不为空则将高亮Fragment（碎片）拼接在一起，替换原有普通字段 **if**(titleHl != **null**) {  Text[] fragments = titleHl.getFragments();  StringBuilder stringBuilder = **new** StringBuilder();  **for** (Text fragment : fragments) {  stringBuilder.append(fragment.string());  }  jobDetail.setTitle(stringBuilder.toString()); }  // 2.2 处理jd高亮 **if**(jdHl != **null**) {  Text[] fragments = jdHl.getFragments();  StringBuilder stringBuilder = **new** StringBuilder();  **for** (Text fragment : fragments) {  stringBuilder.append(fragment.string());  }  jobDetail.setJd(stringBuilder.toString()); } |

我们再查询，发现查询的结果中就都包含了高亮。



### 完整参考代码

|  |
| --- |
| **public class** JobFullTextServiceImpl **implements** JobFullTextService {   **private** RestHighLevelClient **restHighLevelClient**;  **private static final** String ***JOB\_IDX\_NAME*** = **"job\_idx"**;   **public** JobFullTextServiceImpl() {  **restHighLevelClient** = **new** RestHighLevelClient(RestClient.*builder*(  **new** HttpHost(**"node1"**, 9200, **"http"**)  , **new** HttpHost(**"node2"**, 9200, **"http"**)  , **new** HttpHost(**"node3"**, 9200, **"http"**)  ));  }   @Override  **public void** add(JobDetail jobDetail) {  // 1. 构建IndexRequest对象，用来描述ES发起请求的数据。  IndexRequest indexRequest = **new** IndexRequest(***JOB\_IDX\_NAME***);   // 2. 设置文档ID。  indexRequest.id(jobDetail.getId() + **""**);   // 3. 构建一个实体类对象，并使用FastJSON将实体类对象转换为JSON。  String json = JSON.*toJSONString*(jobDetail);   // 4. 使用IndexRequest.source方法设置请求数据。  indexRequest.source(json, XContentType.***JSON***);   **try** {  // 5. 使用ES High level client调用index方法发起请求  **restHighLevelClient**.index(indexRequest, RequestOptions.***DEFAULT***);  } **catch** (IOException e) {  e.printStackTrace();  }   System.***out***.println(**"索引创建成功!"**);  }   @Override  **public void** update(JobDetail jobDetail) **throws** IOException {  // 1. 判断对应ID的文档是否存在  // a) 构建GetRequest  GetRequest getRequest = **new** GetRequest(***JOB\_IDX\_NAME***, jobDetail.getId() + **""**);   // b) 执行client的exists方法，发起请求，判断是否存在  **boolean** exists = **restHighLevelClient**.exists(getRequest, RequestOptions.***DEFAULT***);   **if**(!exists) **return**;   // 2. 构建UpdateRequest请求  UpdateRequest updateRequest = **new** UpdateRequest(***JOB\_IDX\_NAME***, jobDetail.getId() + **""**);   // 3. 设置UpdateRequest的文档，并配置为JSON格式  updateRequest.doc(JSON.*toJSONString*(jobDetail), XContentType.***JSON***);   // 4. 执行client发起update请求  **restHighLevelClient**.update(updateRequest, RequestOptions.***DEFAULT***);  }   @Override  **public** JobDetail findById(**long** id) **throws** IOException {  // 1. 构建GetRequest请求。  GetRequest getRequest = **new** GetRequest(***JOB\_IDX\_NAME***, id + **""**);   // 2. 使用RestHighLevelClient.get发送GetRequest请求，并获取到ES服务器的响应。  GetResponse response = **restHighLevelClient**.get(getRequest, RequestOptions.***DEFAULT***);   // 3. 将ES响应的数据转换为JSON字符串  String json = response.getSourceAsString();   // 4. 并使用FastJSON将JSON字符串转换为JobDetail类对象  JobDetail jobDetail = JSONObject.*parseObject*(json, JobDetail.**class**);   // 5. 设置ID字段  jobDetail.setId(id);   **return** jobDetail;   }   @Override  **public void** deleteById(**long** id) **throws** IOException {  // 1. 构建delete请求  DeleteRequest deleteRequest = **new** DeleteRequest(***JOB\_IDX\_NAME***, id + **""**);   // 2. 使用client执行delete请求  **restHighLevelClient**.delete(deleteRequest, RequestOptions.***DEFAULT***);  }   @Override  **public** List<JobDetail> searchByKeywords(String keywords) **throws** IOException {  // 1. 构建SearchRequest检索请求  SearchRequest searchRequest = **new** SearchRequest(***JOB\_IDX\_NAME***);   // 2. 创建一个SearchSourceBuilder专门用于构建查询条件  SearchSourceBuilder searchSourceBuilder = **new** SearchSourceBuilder();   // 3. 使用QueryBuilders.multiMatchQuery构建一个查询条件，并配置到SearchSourceBuilder  MultiMatchQueryBuilder queryBuilder = QueryBuilders.*multiMatchQuery*(keywords, **"jd"**, **"title"**);  searchSourceBuilder.query(queryBuilder);   // 4. 调用SearchRequest.source将查询条件设置到检索请求  searchRequest.source(searchSourceBuilder);   // 5. 执行RestHighLevelClient.search发起请求  SearchResponse searchResponse = **restHighLevelClient**.search(searchRequest, RequestOptions.***DEFAULT***);   // 6. 遍历结果  SearchHits hits = searchResponse.getHits();   List<JobDetail> jobDetailList = **new** ArrayList<>();   **for** (SearchHit hit : hits) {  // 1) 获取命中的结果  String json = hit.getSourceAsString();  // 2) 将JSON字符串转换为对象  JobDetail jobDetail = JSON.*parseObject*(json, JobDetail.**class**);  // 3) 使用SearchHit.getId设置文档ID  jobDetail.setId(Long.*parseLong*(hit.getId()));   jobDetailList.add(jobDetail);  }   **return** jobDetailList;  }   @Override  **public** Map<String, Object> searchByPage(String keywords, **int** pageNum, **int** pageSize) **throws** IOException {  // 1. 构建SearchRequest检索请求  SearchRequest searchRequest = **new** SearchRequest(***JOB\_IDX\_NAME***);   // 2. 创建一个SearchSourceBuilder专门用于构建查询条件  SearchSourceBuilder searchSourceBuilder = **new** SearchSourceBuilder();   // 3. 使用QueryBuilders.multiMatchQuery构建一个查询条件，并配置到SearchSourceBuilder  MultiMatchQueryBuilder queryBuilder = QueryBuilders.*multiMatchQuery*(keywords, **"jd"**, **"title"**);  searchSourceBuilder.query(queryBuilder);   // 4. 设置SearchSourceBuilder的from和size参数，构建分页  searchSourceBuilder.from(pageNum);  searchSourceBuilder.size(pageSize);   // 4. 调用SearchRequest.source将查询条件设置到检索请求  searchRequest.source(searchSourceBuilder);   // 5. 执行RestHighLevelClient.search发起请求  SearchResponse searchResponse = **restHighLevelClient**.search(searchRequest, RequestOptions.***DEFAULT***);   // 6. 遍历结果  SearchHits hits = searchResponse.getHits();   List<JobDetail> jobDetailList = **new** ArrayList<>();   **for** (SearchHit hit : hits) {  // 1) 获取命中的结果  String json = hit.getSourceAsString();  // 2) 将JSON字符串转换为对象  JobDetail jobDetail = JSON.*parseObject*(json, JobDetail.**class**);  // 3) 使用SearchHit.getId设置文档ID  jobDetail.setId(Long.*parseLong*(hit.getId()));   jobDetailList.add(jobDetail);  }   // 8. 将结果封装到Map结构中（带有分页信息）  // a) total -> 使用SearchHits.getTotalHits().value获取到所有的记录数  // b) content -> 当前分页中的数据  Map<String, Object> result = **new** HashMap<>();  result.put(**"total"**, hits.getTotalHits().**value**);  result.put(**"content"**, jobDetailList);   **return** result;  }   @Override  **public** Map<String, Object> searchByScrollPage(String keywords, String scrollId, **int** pageSize) **throws** IOException {  SearchResponse searchResponse = **null**;   **if**(scrollId == **null**) {  // 1. 构建SearchRequest检索请求  SearchRequest searchRequest = **new** SearchRequest(***JOB\_IDX\_NAME***);   // 2. 创建一个SearchSourceBuilder专门用于构建查询条件  SearchSourceBuilder searchSourceBuilder = **new** SearchSourceBuilder();   // 3. 使用QueryBuilders.multiMatchQuery构建一个查询条件，并配置到SearchSourceBuilder  MultiMatchQueryBuilder queryBuilder = QueryBuilders.*multiMatchQuery*(keywords, **"jd"**, **"title"**);  searchSourceBuilder.query(queryBuilder);  searchSourceBuilder.size(pageSize);   // 设置高亮查询  HighlightBuilder highlightBuilder = **new** HighlightBuilder();  highlightBuilder.preTags(**"<font color='red'>"**);  highlightBuilder.postTags(**"</font>"**);  highlightBuilder.field(**"title"**);  highlightBuilder.field(**"jd"**);   searchSourceBuilder.highlighter(highlightBuilder);   // 4. 调用searchRequest.scroll设置滚动快照有效时间  searchRequest.scroll(TimeValue.*timeValueMinutes*(10));   // 5. 调用SearchRequest.source将查询条件设置到检索请求  searchRequest.source(searchSourceBuilder);   // 6. 执行RestHighLevelClient.search发起请求  searchResponse = **restHighLevelClient**.search(searchRequest, RequestOptions.***DEFAULT***);  }  **else** {  SearchScrollRequest searchScrollRequest = **new** SearchScrollRequest(scrollId);  searchScrollRequest.scroll(TimeValue.*timeValueMinutes*(10));  searchResponse = **restHighLevelClient**.scroll(searchScrollRequest, RequestOptions.***DEFAULT***);  }    **if**(searchResponse != **null**) {  // 7. 遍历结果  SearchHits hits = searchResponse.getHits();   List<JobDetail> jobDetailList = **new** ArrayList<>();   **for** (SearchHit hit : hits) {   // 1) 获取命中的结果  String json = hit.getSourceAsString();  // 2) 将JSON字符串转换为对象  JobDetail jobDetail = JSON.*parseObject*(json, JobDetail.**class**);  // 3) 使用SearchHit.getId设置文档ID  jobDetail.setId(Long.*parseLong*(hit.getId()));    // 1. 获取高亮字段  Map<String, HighlightField> highlightFieldMap = hit.getHighlightFields();  // 1.1 获取title高亮字段  HighlightField titleHl = highlightFieldMap.get(**"title"**);  // 1.2 获取jd高亮字段  HighlightField jdHl = highlightFieldMap.get(**"jd"**);  // 2. 将高亮字段进行替换普通字段  // 2.1 处理title高亮，判断高亮是否为空，不为空则将高亮Fragment（碎片）拼接在一起，替换原有普通字段  **if**(titleHl != **null**) {  Text[] fragments = titleHl.getFragments();  StringBuilder stringBuilder = **new** StringBuilder();  **for** (Text fragment : fragments) {  stringBuilder.append(fragment.string());  }  jobDetail.setTitle(stringBuilder.toString());  }   // 2.2 处理jd高亮  **if**(jdHl != **null**) {  Text[] fragments = jdHl.getFragments();  StringBuilder stringBuilder = **new** StringBuilder();  **for** (Text fragment : fragments) {  stringBuilder.append(fragment.string());  }  jobDetail.setJd(stringBuilder.toString());  }   jobDetailList.add(jobDetail);  }   // 8. 将结果封装到Map结构中（带有分页信息）  // a) total -> 使用SearchHits.getTotalHits().value获取到所有的记录数  // b) content -> 当前分页中的数据  Map<String, Object> result = **new** HashMap<>();  result.put(**"scrollId"**, searchResponse.getScrollId());  result.put(**"content"**, jobDetailList);   **return** result;  }   **return null**;  }   @Override  **public void** close() {  **try** {  **restHighLevelClient**.close();  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

## Elasticsearch架构原理

### Elasticsearch的节点类型

在Elasticsearch有两类节点，一类是Master，一类是DataNode。

#### Master节点

在Elasticsearch启动时，会选举出来一个Master节点。当某个节点启动后，然后使用Zen Discovery机制找到集群中的其他节点，并建立连接。

discovery.seed\_hosts: ["node1", "node2", "node3"]

并从候选主节点中选举出一个主节点。

cluster.initial\_master\_nodes: ["node1", "node2"]

Master节点主要负责：

* 管理索引（创建索引、删除索引）、分配分片
* 维护元数据
* 管理集群节点状态
* 不负责数据写入和查询，比较轻量级

一个Elasticsearch集群中，只有一个Master节点。在生产环境中，内存可以相对小一点，但机器要稳定。

#### DataNode节点

在Elasticsearch集群中，会有N个DataNode节点。DataNode节点主要负责：

* 数据写入、数据检索，大部分Elasticsearch的压力都在DataNode节点上
* 在生产环境中，内存最好配置大一些

### 分片和副本机制

#### 分片（Shard）

* Elasticsearch是一个分布式的搜索引擎，索引的数据也是分成若干部分，分布在不同的服务器节点中
* 分布在不同服务器节点中的索引数据，就是分片（Shard）。Elasticsearch会自动管理分片，如果发现分片分布不均衡，就会自动迁移
* 一个索引（index）由多个shard（分片）组成，而分片是分布在不同的服务器上的

#### 副本

为了对Elasticsearch的分片进行容错，假设某个节点不可用，会导致整个索引库都将不可用。所以，需要对分片进行副本容错。每一个分片都会有对应的副本。在Elasticsearch中，默认创建的索引为1个分片、每个分片有1个主分片和1个副本分片。

* 每个分片都会有一个Primary Shard（主分片），也会有若干个Replica Shard（副本分片）
* Primary Shard和Replica Shard不在同一个节点上

#### 指定分片、副本数量

|  |
| --- |
| // 创建指定分片数量、副本数量的索引  PUT /job\_idx\_shard  {      "mappings": {          "properties": {              "id": { "type": "long", "store": true },              "area": { "type": "keyword", "store": true },              "exp": { "type": "keyword", "store": true },              "edu": { "type": "keyword", "store": true },              "salary": { "type": "keyword", "store": true },              "job\_type": { "type": "keyword", "store": true },              "cmp": { "type": "keyword", "store": true },              "pv": { "type": "keyword", "store": true },              "title": { "type": "text", "store": true },              "jd": { "type": "text"}          }      },      "settings": {          "number\_of\_shards": 3,          "number\_of\_replicas": 2      }  }  // 查看分片、主分片、副本分片  GET /\_cat/indices?v |

### Elasticsearch重要工作流程

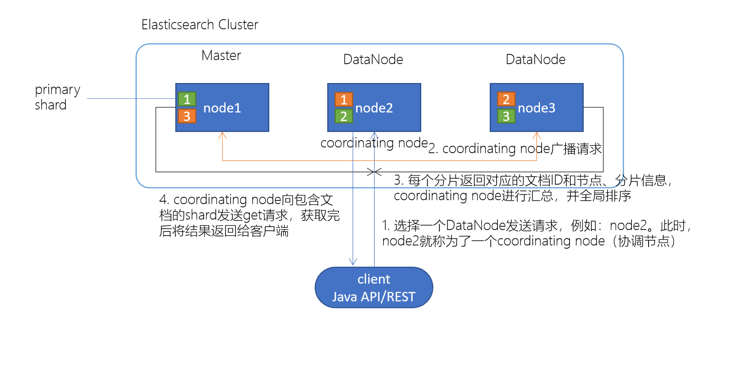
#### Elasticsearch文档写入原理

1. 选择任意一个DataNode发送请求，例如：node2。此时，node2就成为一个 coordinating node（协调节点）
2. 计算得到文档要写入的分片

* `shard = hash(routing) % number\_of\_primary\_shards`
* routing 是一个可变值，默认是文档的 \_id

1. coordinating node会进行路由，将请求转发给对应的primary shard所在的DataNode（假设primary shard在node1、replica shard在node2）
2. node1节点上的Primary Shard处理请求，写入数据到索引库中，并将数据同步到 Replica shard
3. Primary Shard和Replica Shard都保存好了文档，返回client

#### Elasticsearch检索原理



* client发起查询请求，某个DataNode接收到请求，该DataNode就会成为协调节点（Coordinating Node）
* 协调节点（Coordinating Node）将查询请求广播到每一个数据节点，这些数据节点的分片会处理该查询请求。协调节点会轮询所有的分片来自动进行负载均衡
* 每个分片进行数据查询，将符合条件的数据放在一个优先队列中，并将这些数据的文档ID、节点信息、分片信息返回给协调节点
* 协调节点将所有的结果进行汇总，并进行全局排序
* 协调节点向包含这些文档ID的分片发送get请求，对应的分片将文档数据返回给协调节点，最后协调节点将数据返回给客户端

### Elasticsearch准实时索引实现

#### 溢写到文件系统缓存

* 当数据写入到ES分片时，会首先写入到内存中，然后通过内存的buffer生成一个segment，并刷到**文件系统缓存**中，数据可以被检索（注意不是直接刷到磁盘）
* ES中默认1秒，refresh一次

#### 写translog保障容错

* 在写入到内存中的同时，也会记录translog日志，在refresh期间出现异常，会根据translog来进行数据恢复
* 等到文件系统缓存中的segment数据都刷到磁盘中，清空translog文件

#### flush到磁盘

* ES默认每隔30分钟会将文件系统缓存的数据刷入到磁盘

#### segment合并

* Segment太多时，ES定期会将多个segment合并成为大的segment，减少索引查询时IO开销，此阶段ES会真正的物理删除（之前执行过的delete的数据）

## Elasticsearch SQL

Elasticsearch SQL允许执行类SQL的查询，可以使用REST接口、命令行或者是JDBC，都可以使用SQL来进行数据的检索和数据的聚合。

Elasticsearch SQL特点：

* 本地集成
  + Elasticsearch SQL是专门为Elasticsearch构建的。每个SQL查询都根据底层存储对相关节点有效执行。
* 没有额外的要求
  + 不依赖其他的硬件、进程、运行时库，Elasticsearch SQL可以直接运行在Elasticsearch集群上
* 轻量且高效
  + 像SQL那样简洁、高效地完成查询

### SQL与Elasticsearch对应关系

|  |  |
| --- | --- |
| SQL | Elasticsearch |
| column（列） | field（字段） |
| row（行） | document（文档） |
| table（表） | index（索引） |
| schema（模式） | N/A |
| database（数据库） | Elasticsearch集群实例 |

### Elasticsearch SQL语法

|  |
| --- |
| SELECT select\_expr [, ...]  [ FROM table\_name ]  [ WHERE condition ]  [ GROUP BY grouping\_element [, ...] ]  [ HAVING condition]  [ ORDER BY expression [ ASC | DESC ] [, ...] ]  [ LIMIT [ count ] ]  [ PIVOT ( aggregation\_expr FOR column IN ( value [ [ AS ] alias ] [, ...] ) ) ] |

* 目前FROM只支持一个表

### 职位查询案例

#### 查询职位索引库中的一条数据

format：表示指定返回的数据类型

|  |
| --- |
| // 1. 查询职位信息  GET /\_sql?format=txt  {      "query": "SELECT \* FROM job\_idx limit 1"  } |

除了txt类型，Elasticsearch SQL还支持以下类型，

|  |  |
| --- | --- |
| 格式 | 描述 |
| csv | 逗号分隔符 |
| json | JSON格式 |
| tsv | 制表符分隔符 |
| txt | 类cli表示 |
| yaml | YAML人类可读的格式 |

#### 将SQL转换为DSL

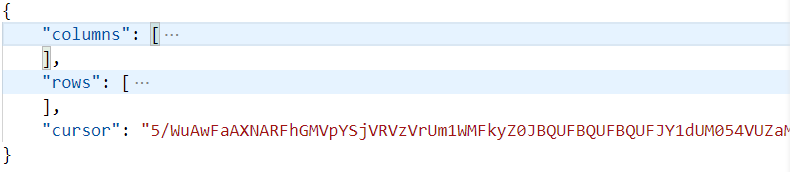
|  |
| --- |
| GET /\_sql/translate  {      "query": "SELECT \* FROM job\_idx limit 1"  } |

#### 职位scroll分页查询

##### 第一次查询

|  |
| --- |
| // 2. scroll分页查询  GET /\_sql?format=json  {      "query": "SELECT \* FROM job\_idx",      "fetch\_size": 10  } |

fetch\_size表示每页显示多少数据，而且当我们指定format为Json格式时，会返回一个cursor ID。



##### 第二次查询

|  |
| --- |
| GET /\_sql?format=json  {      "cursor": "5/WuAwFaAXNARFhGMVpYSjVRVzVrUm1WMFkyZ0JBQUFBQUFBQUFJZ1dUM054VUZaMk9YVlJWalowYkVJeFowUkdVak10ZHc9Pf////8PCgFmBGFyZWEBBGFyZWEBB2tleXdvcmQBAAABZgNjbXABA2NtcAEHa2V5d29yZAEAAAFmA2VkdQEDZWR1AQdrZXl3b3JkAQAAAWYDZXhwAQNleHABB2tleXdvcmQBAAABZgJpZAECaWQBBGxvbmcAAAABZgJqZAECamQBBHRleHQAAAABZghqb2JfdHlwZQEIam9iX3R5cGUBB2tleXdvcmQBAAABZgJwdgECcHYBB2tleXdvcmQBAAABZgZzYWxhcnkBBnNhbGFyeQEHa2V5d29yZAEAAAFmBXRpdGxlAQV0aXRsZQEEdGV4dAAAAAL/Aw=="  } |

##### 清除游标

|  |
| --- |
| POST /\_sql/close  {      "cursor": "5/WuAwFaAXNARFhGMVpYSjVRVzVrUm1WMFkyZ0JBQUFBQUFBQUFJZ1dUM054VUZaMk9YVlJWalowYkVJeFowUkdVak10ZHc9Pf////8PCgFmBGFyZWEBBGFyZWEBB2tleXdvcmQBAAABZgNjbXABA2NtcAEHa2V5d29yZAEAAAFmA2VkdQEDZWR1AQdrZXl3b3JkAQAAAWYDZXhwAQNleHABB2tleXdvcmQBAAABZgJpZAECaWQBBGxvbmcAAAABZgJqZAECamQBBHRleHQAAAABZghqb2JfdHlwZQEIam9iX3R5cGUBB2tleXdvcmQBAAABZgJwdgECcHYBB2tleXdvcmQBAAABZgZzYWxhcnkBBnNhbGFyeQEHa2V5d29yZAEAAAFmBXRpdGxlAQV0aXRsZQEEdGV4dAAAAAL/Aw=="  } |

#### 职位全文检索

##### 需求

检索title和jd中包含hadoop的职位。

##### MATCH函数

在执行全文检索时，需要使用到MATCH函数。

|  |
| --- |
| MATCH(  field\_exp,  constant\_exp  [, options]) |

* field\_exp：匹配字段
* constant\_exp：匹配常量表达式

##### 实现

|  |
| --- |
| GET /\_sql?format=txt  {      "query": "select \* from job\_idx where MATCH(title, 'hadoop') or MATCH(jd, 'hadoop') limit 10"  } |

### 订单统计分析案例

#### 案例介绍

有以下数据集：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 订单ID | 订单状态 | 支付金额 | 支付方式ID | 用户ID | 操作时间 | 商品分类 |
| id | status | pay\_money | payway | userid | operation\_date | category |
| 1 | 已提交 | 4070 | 1 | 4944191 | 2020-04-25 12:09:16 | 手机; |
| 2 | 已完成 | 4350 | 1 | 1625615 | 2020-04-25 12:09:37 | 家用电器;;电脑; |
| 3 | 已提交 | 6370 | 3 | 3919700 | 2020-04-25 12:09:39 | 男装;男鞋; |
| 4 | 已付款 | 6370 | 3 | 3919700 | 2020-04-25 12:09:44 | 男装;男鞋; |

我们需要基于按数据，使用Elasticsearch中的聚合统计功能，实现一些指标统计。

#### 创建索引

|  |
| --- |
| PUT /order\_idx/  {      "mappings": {          "properties": {              "id": {                  "type": "keyword",                  "store": true              },              "status": {                  "type": "keyword",                  "store": true              },              "pay\_money": {                  "type": "double",                  "store": true              },              "payway": {                  "type": "byte",                  "store": true              },              "userid": {                  "type": "keyword",                  "store": true              },              "operation\_date": {                  "type": "date",                  "format": "yyyy-MM-dd HH:mm:ss",                  "store": true              },              "category": {                  "type": "keyword",                  "store": true              }          }      }  } |

#### 导入测试数据

1. 上传资料中的order\_data.json数据文件到Linux
2. 使用bulk进行批量导入命令

|  |
| --- |
| curl -H "Content-Type: application/json" -XPOST "node1:9200/order\_idx/\_bulk?pretty&refresh" --data-binary "@order\_data.json" |

#### 统计不同支付方式的的订单数量

##### 使用JSON DSL的方式来实现

这种方式就是用Elasticsearch原生支持的基于JSON的DSL方式来实现聚合统计。

|  |
| --- |
| GET /order\_idx/\_search  {      "size": 0,      "aggs": {          "group\_by\_state": {              "terms": {                  "field": "payway"              }          }      }  } |

统计结果：

|  |
| --- |
| "aggregations": {          "group\_by\_state": {              "doc\_count\_error\_upper\_bound": 0,              "sum\_other\_doc\_count": 0,              "buckets": [                  {                      "key": 2,                      "doc\_count": 1496                  },                  {                      "key": 1,                      "doc\_count": 1438                  },                  {                      "key": 3,                      "doc\_count": 1183                  },                  {                      "key": 0,                      "doc\_count": 883                  }              ]          }      } |

这种方式分析起来比较麻烦，如果将来我们都是写这种方式来分析数据，简直是无法忍受。所以，Elasticsearch想要进军OLAP领域，是一定要支持SQL，能够使用SQL方式来进行统计和分析的。

##### 基于Elasticsearch SQL方式实现

|  |
| --- |
| GET /\_sql?format=txt  {      "query": "select payway, count(\*) as order\_cnt from order\_idx group by payway"  } |

这种方式要更加直观、简洁。

#### 基于JDBC方式统计不同方式的订单数量

Elasticsearch中还提供了基于JDBC的方式来访问数据。我们可以像操作MySQL一样操作Elasticsearch。使用步骤如下：

1. 在pom.xml中添加以下镜像仓库

|  |
| --- |
| <repositories>  <repository>  <id>elastic.co</id>  <url>https://artifacts.elastic.co/maven</url>  </repository>  </repositories> |

1. 导入Elasticsearch JDBC驱动Maven依赖

|  |
| --- |
| <dependency>  <groupId>org.elasticsearch.plugin</groupId>  <artifactId>x-pack-sql-jdbc</artifactId>  <version>7.6.1</version>  </dependency> |

1. 驱动

org.elasticsearch.xpack.sql.jdbc.EsDriver

1. JDBC URL

jdbc:es:// http:// host:port

1. 开启X-pack高阶功能试用，如果不开启试用，会报如下错误

|  |
| --- |
| current license is non-compliant for [jdbc] |

在node1节点上执行：

|  |
| --- |
| curl http://node1:9200/\_license/start\_trial?acknowledge=true -X POST  {"acknowledged":true,"trial\_was\_started":true,"type":"trial"} |

试用期为30天。

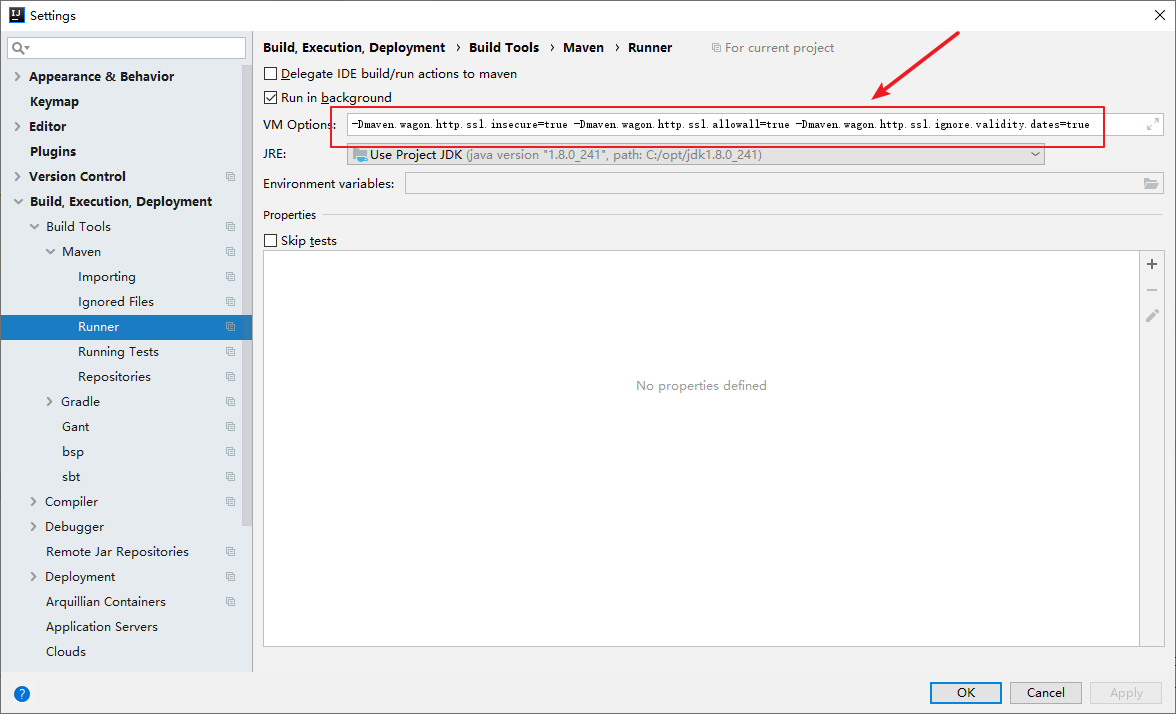
参考代码：

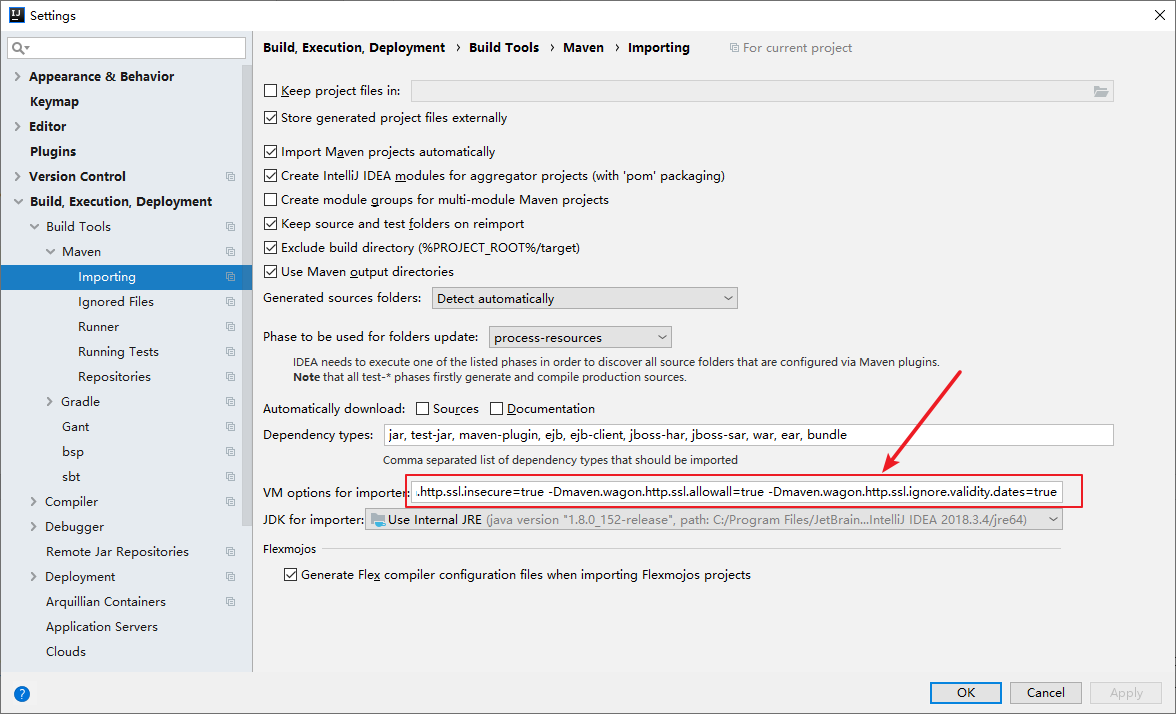
|  |
| --- |
| /\*\*  \* 基于JDBC访问Elasticsearch  \*/ **public class** ElasticJdbc {   **public static void** main(String[] args) **throws** Exception {  Class.*forName*(**"org.elasticsearch.xpack.sql.jdbc.EsDriver"**);   Connection connection = DriverManager.*getConnection*(**"jdbc:es://http://node1:9200"**);  PreparedStatement ps = connection.prepareStatement(**"select payway,** *count***(***\****) as order\_cnt from order\_idx group by payway"**);  ResultSet resultSet = ps.executeQuery();   **while**(resultSet.next()) {  **int** payway = resultSet.getInt(**"payway"**);  **int** order\_cnt = resultSet.getInt(**"order\_cnt"**);  System.***out***.println(**"支付方式: "** + payway + **" 订单数量: "** + order\_cnt);  }   resultSet.close();  ps.close();  connection.close();  } } |

注意：如果在IDEA中无法下载依赖，请参考以下操作：

在Idea的File -->settings中，设置Maven的importing和Runner参数，忽略证书检查即可。(Eclipse下解决原理类似，设置maven运行时参数)，并尝试手动执行Maven compile执行编译。

具体参数：-Dmaven.multiModuleProjectDirectory=$MAVEN\_HOME -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true -Dmaven.wagon.http.ssl.ignore.validity.dates=true





#### 统计不同支付方式订单数，并按照订单数量倒序排序

|  |
| --- |
| GET /\_sql?format=txt  {      "query": "select payway, count(\*) as order\_cnt from order\_idx group by payway order by order\_cnt desc"  } |

#### 只统计「已付款」状态的不同支付方式的订单数量

|  |
| --- |
| GET /\_sql?format=txt  {      "query": "select payway, count(\*) as order\_cnt from order\_idx where status = '已付款' group by payway order by order\_cnt desc"  } |

### 统计不同状态的订单总额、不同支付方式最高、最低订单金额

统计不同状态的订单总额、不同支付方式最高、最低订单金额

|  |
| --- |
| GET /\_sql?format=txt  {      "query": "select userid, count(1) as cnt, sum(pay\_money) as total\_money from order\_idx group by userid"  } |

### Elasticsearch SQL目前的一些限制

目前Elasticsearch SQL还存在一些限制。例如：不支持JOIN、不支持较复杂的子查询。所以，有一些相对复杂一些的功能，还得借助于DSL方式来实现。

## 常见问题处理

### elasticsearch.keystore AccessDeniedException

|  |
| --- |
| Exception in thread "main" org.elasticsearch.bootstrap.BootstrapException: java.nio.file.AccessDeniedException: /export/server/es/elasticsearch-7.6.1/config/elasticsearch.keystore  Likely root cause: java.nio.file.AccessDeniedException: /export/server/es/elasticsearch-7.6.1/config/elasticsearch.keystore  at java.base/sun.nio.fs.UnixException.translateToIOException(UnixException.java:90)  at java.base/sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:111)  at java.base/sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:116)  at java.base/sun.nio.fs.UnixFileSystemProvider.newByteChannel(UnixFileSystemProvider.java:219)  at java.base/java.nio.file.Files.newByteChannel(Files.java:374)  at java.base/java.nio.file.Files.newByteChannel(Files.java:425)  at org.apache.lucene.store.SimpleFSDirectory.openInput(SimpleFSDirectory.java:77)  at org.elasticsearch.common.settings.KeyStoreWrapper.load(KeyStoreWrapper.java:219)  at org.elasticsearch.bootstrap.Bootstrap.loadSecureSettings(Bootstrap.java:234)  at org.elasticsearch.bootstrap.Bootstrap.init(Bootstrap.java:305)  at org.elasticsearch.bootstrap.Elasticsearch.init(Elasticsearch.java:170)  at org.elasticsearch.bootstrap.Elasticsearch.execute(Elasticsearch.java:161)  at org.elasticsearch.cli.EnvironmentAwareCommand.execute(EnvironmentAwareCommand.java:86)  at org.elasticsearch.cli.Command.mainWithoutErrorHandling(Command.java:125)  at org.elasticsearch.cli.Command.main(Command.java:90)  at org.elasticsearch.bootstrap.Elasticsearch.main(Elasticsearch.java:126)  at org.elasticsearch.bootstrap.Elasticsearch.main(Elasticsearch.java:92) |

解决方案：

将/export/server/es/elasticsearch-7.6.1/config/elasticsearch.keystore owner设置为roy

chown roy /export/server/es/elasticsearch-7.6.1/config/elasticsearch.keystore