# 【ELK：开源搜索与分析技术栈】

## 主要内容

1. Spring Data Elasticsearch
2. Logstash

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| Spring Data Elasticsearch | 掌握 |
| Logstash | 掌握 |

## Java操作Elaticsearch

在Java中操作Elasticsearch有两种方式

### Elasticsearch提供的客户端API

依赖的名称叫做transport。依赖的版本和Elasticsearch的版本是对应的。

|  |
| --- |
| <dependency>  <groupId>org.elasticsearch.client</groupId>  <artifactId>transport</artifactId>  <version>7.6.2</version>  </dependency> |

### Spring Data Elasticsearch

使用Spring Data 下二级子项目Spring Data Elasticsearch进行操作。支持POJO方法操作Elasticsearch。相比Elasticsearch提供的API更加简单更加方便。

## Spring Data Elasticsearch项目环境搭建

### 创建项目

新建任意名称项目

### 修改POM文件添加依赖

目前使用spring-boot-starter-parent版本为2.3.3RELEASE

对应spring-data-elasticsearch版本为4.0.3

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-elasticsearch</**artifactId**> </**dependency**> |

### 修改配置文件

在老版本中通过9300内部端口访问。通过TransportClient进行访问的。

从Elasticsearch 8.x开始要放弃Transport。

所以从Spring Data Elasticsearch 4.0 开始都是基于Rest进行访问。

|  |
| --- |
| **spring**:  **elasticsearch**:  **rest**:  **uris**: http://192.168.8.132:9200 |

## ElasticsearchTemplate的使用

### 创建实体

@Document指定实体类和索引对应关系

indexName：索引名称

type: 索引类型（从ES 7.0 开始已经过时了）

shards: 主分片数量。从ES 7开始默认1

replicas：复制分片数量。从ES 7开始默认1

@Id 指定主键

@Field指定普通属性

type： 对应Elasticsearch中属性类型。使用FiledType枚举可以快速获取。测试发现没有type属性可能出现无法自动创建类型问题，所以一定要有type属性。

text类型能被分词

keywords不能被分词

index： 是否创建索引。作为搜索条件时index必须为true

analyzer：指定分词器类型。

|  |
| --- |
| @Data @NoArgsConstructor @AllArgsConstructor @Document(indexName = **"index\_people"**) **public class** People {  @Id  **private** String **id**;  @Field(type = FieldType.***Keyword***)  **private** String **name**;  @Field(type=FieldType.***Integer***)  **private** Integer **age**;  @Field(type=FieldType.***Text***,analyzer = **"ik\_max\_word"**)  **private** String **remark**; } |

### 初始化索引

|  |
| --- |
| @Test **void** contextLoads() {  *// 从 spring data es 4.0开始所有索引操作都在这个接口* IndexOperations indexOperations = **elasticsearchRestTemplate**.indexOps(People.**class**);  *// 创建索引* indexOperations.create();  *// createMapping 根据实体类获取映射关系  // putMapping 把映射关系添加到索引中* indexOperations.putMapping(indexOperations.createMapping(People.**class**)); } |

### 删除索引

|  |
| --- |
| @Test **void** deleteIndex(){  IndexOperations indexOperations = **elasticsearchRestTemplate**.indexOps(People.**class**);  indexOperations.delete(); } |

### 添加文档

如果索引和类型不存在，也可以执行进行新增，新增后自动创建索引和类型。但是field通过动态mapping进行映射，elaticsearch根据值类型进行判断每个属性类型，默认每个属性都是standard分词器，ik分词器是不生效的。**所以一定要先通过代码进行初始化或直接在elasticsearch中通过命令创建所有field的mapping**

#### 新增单个文档

如果对象的id属性没有赋值，让ES自动生成主键，存储时id属性没有值，\_id存储document的主键值。

如果对象的id属性明确设置值，存储时id属性为设置的值，ES中document对象的\_id也是设置的值。

|  |
| --- |
| @Test **void** addDoc(){  *// 返回值为真正存储到ES中的数据。  // 如果主键是自增的，该实体类会包含主键值。* People peo = **elasticsearchRestTemplate**.save(**new** People(**"1"**, **"赵六"**, 15, **"拿着VIVO手机"**));  System.***out***.println(peo); } |

#### 批量新增

|  |
| --- |
| @Test **void** addDocs(){  List<People> list = **new** ArrayList<>();  list.add(**new** People(**"2"**,**"李四"**,13,**"拿着华为手机"**));  list.add(**new** People(**"3"**,**"王五"**,14,**"拿着OPPO手机"**));  Iterable<People> save = **elasticsearchRestTemplate**.save(list);  System.***out***.println(save); } |

### 删除操作

|  |
| --- |
| @Test **void** delete(){  String result = **elasticsearchRestTemplate**.delete(**"1"**, People.**class**);  System.***out***.println(result); } |

### 修改操作

修改操作就是新增代码，只要保证主键id已经存在，新增就是修改

### 查询操作

#### 根据主键查询

|  |
| --- |
| @Test **void** search(){  People people = **elasticsearchRestTemplate**.get(**"1"**, People.**class**);  System.***out***.println(people); } |

#### 全字段模糊查询

去所有field中查询指定条件。

|  |
| --- |
| @Test **void** search2(){  QueryStringQueryBuilder queryBuilder = QueryBuilders.*queryStringQuery*(**"手机"**);  Query query = **new** NativeSearchQuery(queryBuilder);  *// 相当于最外层hits* SearchHits<People> searchHits = **elasticsearchRestTemplate**.search(query, People.**class**);  *// 里层的hits* List<SearchHit<People>> list = searchHits.getSearchHits();  *// 进行转换* List<People> listResult = **new** ArrayList<>();  list.forEach(sh -> {  listResult.add(sh.getContent());  });  System.***out***.println(listResult); } |

#### 使用match\_all查询所有文档

|  |
| --- |
| @Test **void** search3(){  Query query = **new** NativeSearchQuery(QueryBuilders.*matchAllQuery*());  SearchHits<People> searchHits = **elasticsearchRestTemplate**.search(query, People.**class**);  List<SearchHit<People>> searchHitList = searchHits.getSearchHits();  *// 进行转换* List<People> listResult = **new** ArrayList<>();  searchHitList.forEach(sh -> {  listResult.add(sh.getContent());  });  System.***out***.println(listResult); } |

#### 使用match查询文档

|  |
| --- |
| @Test **void** search4(){  Query query = **new** NativeSearchQuery(QueryBuilders.*matchQuery*(**"remark"**,**"我要买小米"**));  SearchHits<People> searchHits = **elasticsearchRestTemplate**.search(query, People.**class**);  List<SearchHit<People>> searchHitList = searchHits.getSearchHits();  *// 进行转换* List<People> listResult = **new** ArrayList<>();  searchHitList.forEach(sh -> {  listResult.add(sh.getContent());  });  System.***out***.println(listResult); } |

#### 使用match\_phrase查询文档

短语搜索是对条件不分词，但是文档中属性根据配置实体类时指定的分词类型进行分词。

如果属性使用ik分词器，从分词后的索引数据中进行匹配。

|  |
| --- |
| @Test **void** mathPhrase(){  SearchQuery searchQuery = **new** NativeSearchQuery(QueryBuilders.*matchPhraseQuery*(**"address"**,**"北京市"**));  List<People> list = **elasticsearchTemplate**.queryForList(searchQuery, People.**class**);  **for**(People people : list){  System.***out***.println(people);  } } |

#### 使用range查询文档

|  |
| --- |
| @Test **void** range(){  SearchQuery searchQuery = **new** NativeSearchQuery(QueryBuilders.*rangeQuery*(**"age"**).gte(22).lte(23));  List<People> list = **elasticsearchTemplate**.queryForList(searchQuery, People.**class**);  **for**(People people : list){  System.***out***.println(people);  } } |

#### 多条件查询

|  |
| --- |
| @Test  **void** MustShould(){  BoolQueryBuilder boolQueryBuilder = QueryBuilders.*boolQuery*();  List<QueryBuilder> listQuery = **new** ArrayList<>();  listQuery.add(QueryBuilders.*matchPhraseQuery*(**"name"**,**"张三"**));  listQuery.add(QueryBuilders.*rangeQuery*(**"age"**).gte(22).lte(23)); *// boolQueryBuilder.should().addAll(listQuery); // 逻辑或* boolQueryBuilder.must().addAll(listQuery); *// 逻辑与* SearchQuery searchQuery = **new** NativeSearchQuery(boolQueryBuilder);  List<People> list = **elasticsearchTemplate**.queryForList(searchQuery, People.**class**);  **for**(People people : list){  System.***out***.println(people);  }  } |

#### 分页与排序

|  |
| --- |
| @Test **void** PageSort(){  SearchQuery searchQuery = **new** NativeSearchQuery(QueryBuilders.*matchAllQuery*());  *// 分页 第一个参数是页码，从0算起。第二个参数是每页显示的条数* searchQuery.setPageable(PageRequest.*of*(0,2));  *// 排序 第一个参数排序规则 DESC ASC 第二个参数是排序属性* searchQuery.addSort(Sort.*by*(Sort.Direction.***DESC***,**"age"**));  List<People> list = **elasticsearchTemplate**.queryForList(searchQuery, People.**class**);  **for**(People people : list){  System.***out***.println(people);  } } |

如果实体类中主键只有@Id注解，String id对应ES中是text类型，text类型是不允许被排序，所以如果必须按照主键进行排序时需要在实体类中设置主键类型

|  |
| --- |
| @Id @Field(type = FieldType.***Keyword***) **private** String **id**; |

#### 高亮查询

|  |
| --- |
| @Test **void** search9(){  NativeSearchQuery nativeSearchQuery = **new** NativeSearchQuery(QueryBuilders.*matchQuery*(**"remark"**, **"手机"**));  *// 排序* nativeSearchQuery.addSort(Sort.*by*(Sort.Direction.***DESC***,**"age"**));  *// 分页* nativeSearchQuery.setPageable(PageRequest.*of*(0,2));  *// 设置高亮条件* HighlightBuilder hlBuilder = **new** HighlightBuilder();  *// 哪个属性高亮* hlBuilder.field(**"remark"**);  *// 高亮内容前缀* hlBuilder.preTags(**"<span style='color:red'>"**);  *// 高亮内容后缀* hlBuilder.postTags(**"</span>"**);  *// 高亮查询* HighlightQuery hlQuery = **new** HighlightQuery(hlBuilder);  *// 应用高亮* nativeSearchQuery.setHighlightQuery(hlQuery);   *// 外层hits* SearchHits<People> searchHits = **elasticsearchRestTemplate**.search(nativeSearchQuery, People.**class**);  *// 查询出的总条数* System.***out***.println(searchHits.getTotalHits());  *// 里层hits* List<SearchHit<People>> searchHitList = searchHits.getSearchHits();   List<People> list = **new** ArrayList<>();   searchHitList.forEach(sh ->{  *// 取出非高亮数据* People peo = sh.getContent();  *// 获取高亮数据* String hlContent = sh.getHighlightField(**"remark"**).get(0);  *// 用高亮数据替换非高亮数据* peo.setRemark(hlContent);  list.add(peo);  });   System.***out***.println(list); } |

## LogStash简介

### 什么是LogStash

ELK(Elasticsearch+Logstash+Kibana)中我们使用过Elasticsearch和Kibana，就剩下最后一个LogStash了。

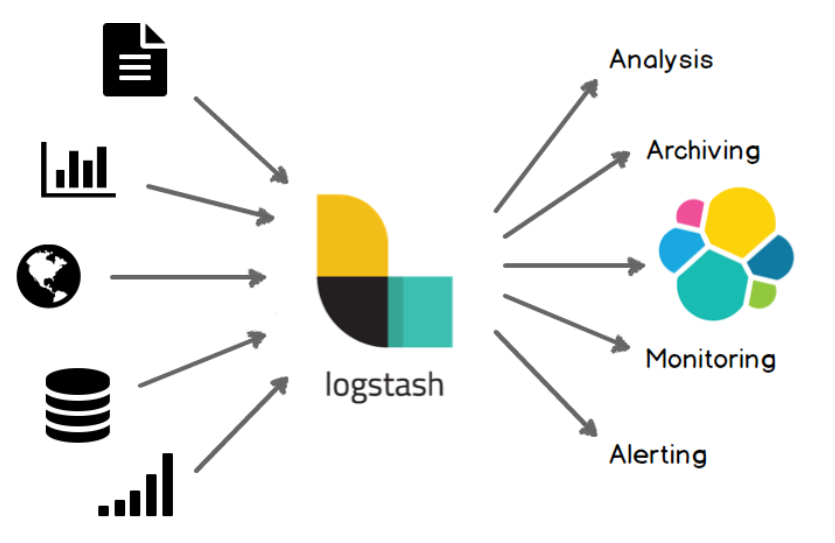
到底Logstash是什么呢？官方说明



官方文字说明：Logstash 是开源的服务器端数据处理管道，能够同时从多个来源采集数据，转换数据，然后将数据发送到您最喜欢的“存储库”中。

通俗说明：Logstash是一款强大的数据处理工具，常用作日志处理。

到目前为止，Logstash已经有超过200个可用的插件，以及创建和贡献自己的灵活性。社区生态非常完善，对于我们可以放心的使用。



### 为什么使用Logstash

通常当系统发生故障时，工程师需要登录到各个服务器上，使用 grep / sed / awk 等 Linux 脚本工具去日志里查找故障原因。在没有日志系统的情况下，首先需要定位处理请求的服务器，如果这台服务器部署了多个实例，则需要去每个应用实例的日志目录下去找日志文件。每个应用实例还会设置日志滚动策略（如：每天生成一个文件），还有日志压缩归档策略等。

这样一系列流程下来，对于我们排查故障以及及时找到故障原因，造成了比较大的麻烦。因此，如果我们能**把这些日志集中管理**，并提供集中检索功能，不仅可以提高诊断的效率，同时对系统情况有个全面的理解，避免事后救火的被动。

所以日志集中管理功能就可以使用ELK技术栈进行实现。Elasticsearch只有数据存储和分析的能力，Kibana就是可视化管理平台。还缺少数据收集和整理的角色，这个功能就是Logstash负责的。

### Logstash工作原理（面试题）

#### Data Source

Logstash 支持的数据源有很多。例如对于日志功能来说只能能有日志记录和日志传递功能的日志都支持，Spring Boot中默认推荐logback支持日志输出功能（输出到数据库、数据出到文件）。

我们就使用logback进行日志输出给Logstash。

#### Logstash Pipeline

整个整体就是Logstash的功能。

在Logstash中包含非常重要的三个功能：

1. Input

输入源，一般配置为自己监听的主机及端口。DataSource向指定的ip及端口输出日志，Input 输入源监听到数据信息就可以进行收集。

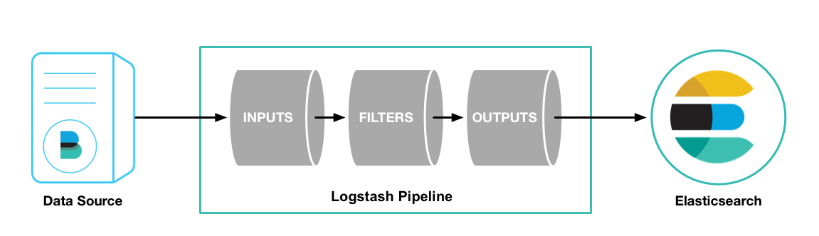
1. Filter

过滤功能，对收集到的信息进行过滤（额外处理），也可以省略这个配置（不做处理）

1. Output

把收集到的信息发送给谁。在ELK技术栈中都是输出给Elasticsearch，后面数据检索和数据分析的过程就给Elasticsearch了。

最终效果：通过整体步骤就可以把原来一行日志信息转换为Elasticsearch支持的Document形式（键值对形式）的数据进行存储。



### 安装Logstash

在前面的课程中已经安装好了Elasticsearch和Kibana。下面是安装Logstash的步骤

#### 安装Logstash

|  |
| --- |
| docker pull logstash:7.6.2 |

#### 启动容器

|  |
| --- |
| docker run -p 4560:4560 --name logstash -d logstash:7.6.2 |

#### 修改配置

进入容器

|  |
| --- |
| docker exec -it logstash /bin/bash |

修改配置文件

|  |
| --- |
| vi /usr/share/logstash/config/logstash.yml |

把ip修改成elasticsearch访问地址IP



#### 修改输入输出配置

继续在容器命令行输入

|  |
| --- |
| vi /usr/share/logstash/pipeline/logstash.conf |

配置解释说明：

input:接收日志输入配置

tcp: 协议

mode: logstash服务

port：端口，自己指定。默认4560

output：日志处理输出

elasticsearch: 交给es处理

action：es中index命令。也就是新增命令。

hosts：es的主机

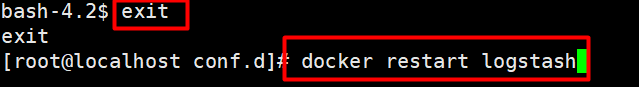
index:存储日志的索引。如果不存在可以自动创建。默认的type名称为doc

**一定要先启动编辑状态（点击键盘i键）在粘贴，如果没启用第一行是nput{少个i**。

|  |
| --- |
| input {  tcp {  mode => "server"  port => 4560  }  }  filter {  }  output {  elasticsearch {  action => "index"  hosts => "192.168.8.128:9200"  index => "test\_log"  }  } |

#### 重启容器

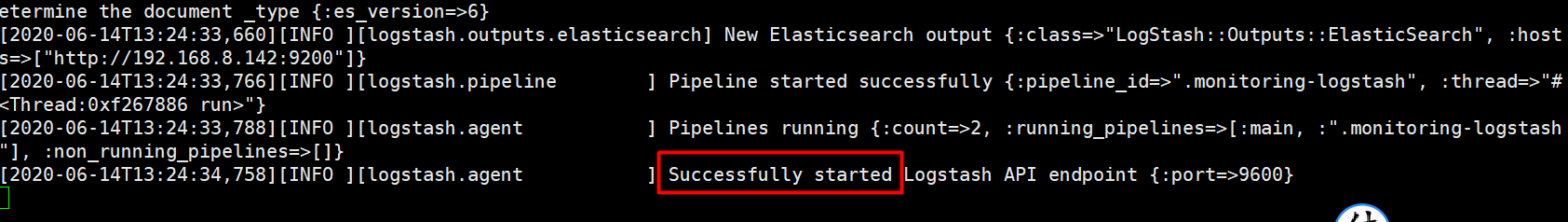
退出容器命令行，进入到Linux终端，重启logstash容器。



|  |
| --- |
| docker restart logstash |

#### 查看日志

|  |
| --- |
| docker logs -f logstash |



## 使用Logback向Logstash中输出日志

需求：随意新建一个项目把输出到控制台的日志信息也输出到Logstash中。

### 修改pom.xml

logstash-logback-encoder就是转码后向logstash中输入的依赖。

注意：

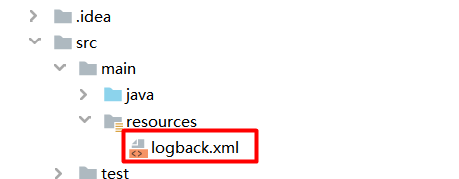
如果导入的是6.x版本不会在控制台看见任何额外日志信息。

如果导入的是5.x版本会在控制台看见logback.xml加载的信息。

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.3.3.RELEASE</**version**> </**parent**> <**dependencies**>  <**dependency**>  <**groupId**>net.logstash.logback</**groupId**>  <**artifactId**>logstash-logback-encoder</**artifactId**>  <**version**>6.3</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**> </**dependencies**> |

### 导入logback.xml

在当天授课目录/软件 中有logback.xml。把文件粘贴到resources中。



logback.xml文件内容如下，红色部分表示向logstash中输出日志信息。

红色中<destination>配置的是logstash配置文件中input里面host和post的信息。

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?> <!--该日志将日志级别不同的log信息保存到不同的文件中 -->* <**configuration**>  <**include resource="org/springframework/boot/logging/logback/defaults.xml"** />   <**springProperty scope="context" name="springAppName"  source="spring.application.name"** />   *<!-- 日志在工程中的输出位置 -->* <**property name="LOG\_FILE" value="${BUILD\_FOLDER:-build}/${springAppName}"** />   *<!-- 控制台的日志输出样式 -->* <**property name="CONSOLE\_LOG\_PATTERN"  value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}}"** />   *<!-- 控制台输出 -->* <**appender name="console" class="ch.qos.logback.core.ConsoleAppender"**>  <**filter class="ch.qos.logback.classic.filter.ThresholdFilter"**>  <**level**>INFO</**level**>  </**filter**>  *<!-- 日志输出编码 -->* <**encoder**>  <**pattern**>${CONSOLE\_LOG\_PATTERN}</**pattern**>  <**charset**>utf8</**charset**>  </**encoder**>  </**appender**>  *<!-- logstash远程日志配置-->* <**appender name="logstash" class="net.logstash.logback.appender.LogstashTcpSocketAppender"**>  <**destination**>192.168.8.132:4560</**destination**>  <**encoder charset="UTF-8" class="net.logstash.logback.encoder.LogstashEncoder"** />  </**appender**>  *<!-- 日志输出级别 -->* <**root level="DEBUG"**>  <**appender-ref ref="console"** />  <**appender-ref ref="logstash"** />  </**root**> </**configuration**> |

### 新建启动类

新建com.bjsxt.DemoApplication

|  |
| --- |
| @SpringBootApplication **public class** DemoApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(DemoApplication.**class**,args);  } } |

## 在Kibana中查看日志信息

### 使用命令方式查看

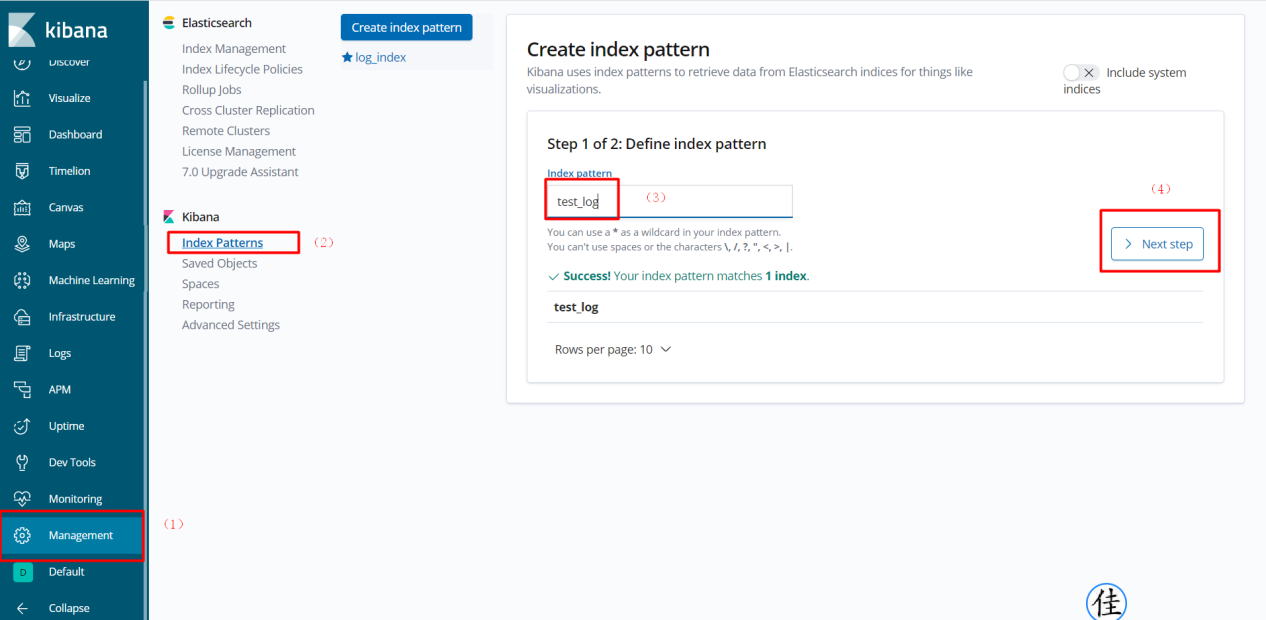
可以直接在Dev Tools中输入命令查看日志信息。

输入： GET test\_log/\_search 查看全部。

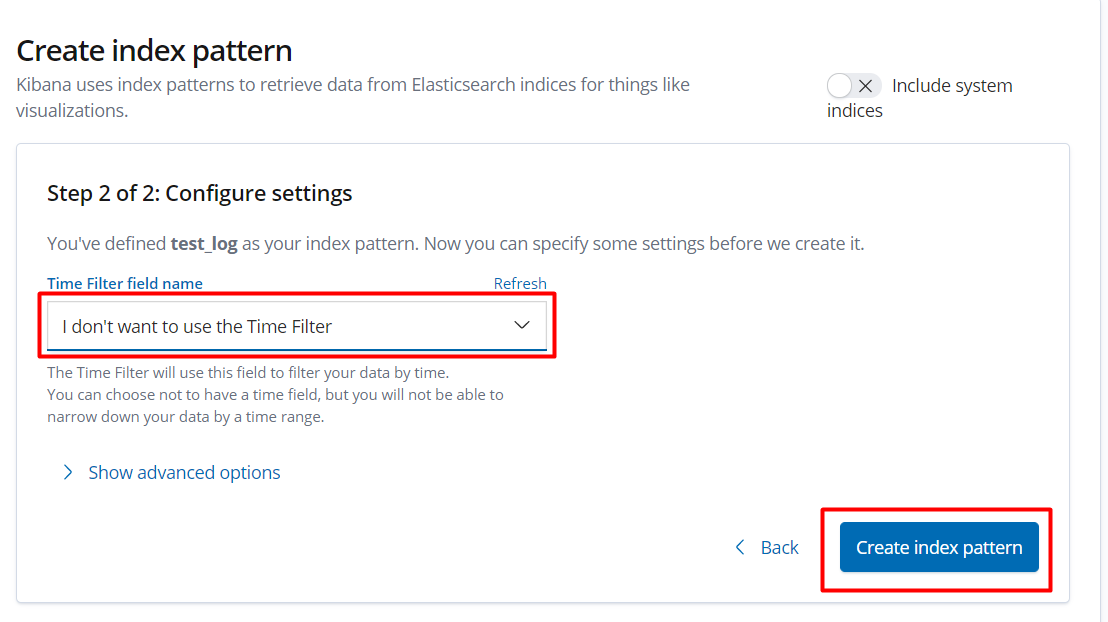


### 是Kibana界面查看

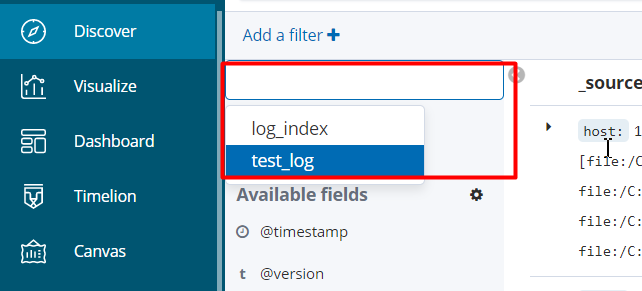
进入到Kibana后按图所示点击。创建索引表达式



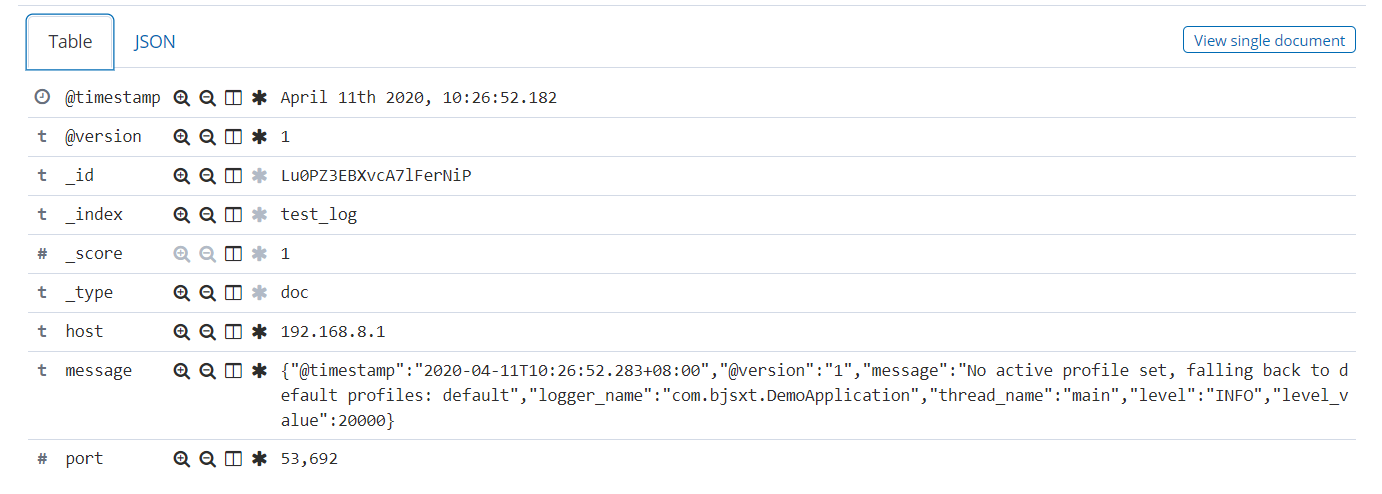
选择没有时间过滤后，点击“Create index pattern”按钮



点击菜单中Discover，选择右侧test\_log



每条日志在Elasticsearch中存储形式



IDEA中控制台打印的原日志内容是下面内容。Logstash作用就是把下面内容转换为上面Elasticsearch存储的内容。在中间做了数据格式转换，收集数据放入Elasticsearch中的工作。



## 搭建日志系统

绝大多数项目在后台管理中都有日志管理。以前的日志信息是存储在MySQL中，日志随着项目运行时间会越来越多，一直存储在MySQL会导致查询降低。现在的日志信息通过ELK技术栈进行操作。存储在Elasticsearch中，可以更好的分析日志内容及更快查询效率。

给定简单需求：

搭建日志系统，提供查询Elasticsearch中日志信息的接口。

### 新建项目

名称为ELK\_Demo

### 修改pom.xml

搭建最基本的环境，实现需求，没有考虑Spring Cloud相关环境，如果考虑Spring Cloud还需要配置Eureka等信息。

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.3.3.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-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  <**version**>1.18.12</**version**>  </**dependency**> </**dependencies**> |

### 创建配置文件

在resources下新建application.yml配置文件。

配置Elasticsearch相关配置信息。

|  |
| --- |
| **spring**:  **elasticsearch**:  **rest**:  **uris**: http://192.168.8.132:9200 |

### 新建实体

根据kibana中查看到日志信息可以得出看出，除了message是类类型，里面包含一些其他属性外，其他的属性都是简单类型属性。



新建com.bjsxt.pojo.Log。

注意@version和@timestamp在Spring data Elasticsearch 3.x中要使用@JsonProperty进行接收。

@version和@timestamp在spring data elasticsearch 4.x中可以直接使用@Field的name属性进行映射。

时间类型要使用format属性进行转换。不要使用custom方式。

|  |
| --- |
| @Data @Document(indexName = **"test\_log"**) **public class** LogPojo {  @Id  **private** String **id**;  @Field(type = FieldType.***Keyword***)  **private** String **host**;  @Field(type = FieldType.***Integer***)  **private** Integer **port**;  @Field(type = FieldType.***Text***)  **private** String **message**;  @Field(type = FieldType.***Date***,name = **"@timestamp"**,format = DateFormat.***date\_time***)  **private** Date **timestamp**;  @Field(type = FieldType.***Long***,name = **"@version"**)  **private** Long **version**; } |

### 新建service及实现类

新建com.bjsxt.service.LogService及实现类。

查询最新15分钟内容的日志信息。

目前设置输入到logstash和控制台的级别Error(logback.xml)

|  |
| --- |
| **public interface** DemoService {  List<LogPojo> showLog(); } |
| @Service @Slf4j **public class** DemoServiceImpl **implements** DemoService {  @Autowired  **private** ElasticsearchRestTemplate **elasticsearchRestTemplate**;  @Override  **public** List<LogPojo> showLog() {  ***log***.error(**"自己写的日志"**);  *// 获取当前时间* Calendar instance = Calendar.*getInstance*();  instance.add(Calendar.***MINUTE***,-15);  NativeSearchQuery query = **new** NativeSearchQuery(QueryBuilders.*rangeQuery*(**"@timestamp"**).gte(instance.getTime()));  SearchHits<LogPojo> search = **elasticsearchRestTemplate**.search(query, LogPojo.**class**);  List<SearchHit<LogPojo>> histList = search.getSearchHits();  List<LogPojo> list = **new** ArrayList<>();  histList.forEach(sh -> {  LogPojo lp = sh.getContent();  list.add(lp);  });  **return** list;  } } |

### 新建控制器

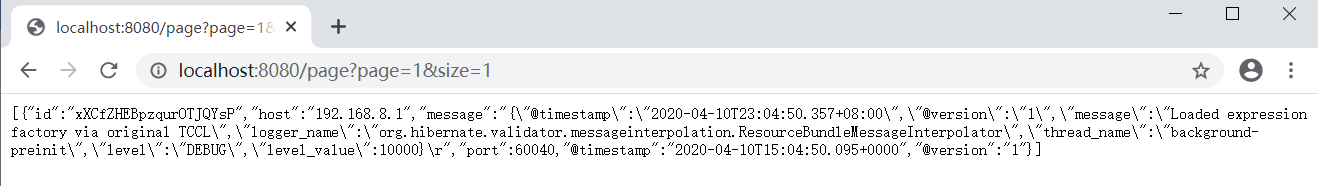
新建com.bjsxt.controller.LogController

|  |
| --- |
| @RestController **public class** DemoController {  @Autowired  **private** DemoService **demoService**;   @RequestMapping(**"/"**)  **public** List<LogPojo> list(){  **return demoService**.showLog();  } } |

### 测试结果

在浏览器输入: [http://localhost:8080/](http://localhost:8080/page?page=1&size=2)

会看见下面的结果。



## 在Java代码中获取日志信息

上面搭建日志系统时完成了从ES中取出日志信息的功能。但是所有真实日志的信息都存储在String message中。默认没有做任何处理，处理过程应该交给客户端做字符串JSON转换，最终显示在页面中。如果在服务器端必须要获取日志内容（message）或日志级别（level）或线程名（thread\_name）等日志信息就需要在服务器端做JSON转换。

下面代码演示如果在服务器端把String message转换为实体类并获取日志信息的步骤。

所有的操作都在上面代码基础进行操作。

基于Jackson进行的转换。

### 新建实体类

新建com.bjsxt.pojo.MessagePojo

|  |
| --- |
| */\*\*  \* 不对应ES中DOcument，所以没有@Document注解  \* 作用：为了对应message属性，把字符串message属性转换为实体类类型，可以方便取出里面一些属性。  \*/* @Data **public class** MessagePojo {  @JsonProperty(**"@timestamp"**)  **private** Date **timestamp**;  @JsonProperty(**"@version"**)  **private** String **version**;  **private** String **message**;  **private** String **logger\_name**;  **private** String **thread\_name**;  **private** String **level**;  **private** String **level\_value**; } |

### 修改LogPojo实体类

修改com.bjsxt.pojo.LogPojo。添加一个属性

|  |
| --- |
| @Data @Document(indexName = **"test\_log"**) **public class** LogPojo {  @Id  **private** String **id**;  @Field(type = FieldType.***Keyword***)  **private** String **host**;  @Field(type = FieldType.***Integer***)  **private** Integer **port**;  @Field(type = FieldType.***Text***)  **private** String **message**;  @Field(type = FieldType.***Date***,name = **"@timestamp"**,format = DateFormat.***date\_time***)  **private** Date **timestamp**;  @Field(type = FieldType.***Long***,name = **"@version"**)  **private** Long **version**;  *// 不与ES中的属性对应。当自己业务中需要日志内容时。* **private** MessagePojo **mp**; } |

### 修改service实现类

修改com.bjsxt.service.impl.DemoServiceImpl。

使用Jackson把Log类中message转换为MessagePojo类型，并设置到LogPojo类中mp属性内容。

转换完成后的MessagePojo类型中就存储了日志所有相关信息。通过操作MessagePojo获取所有的日志内容。

|  |
| --- |
| @Service @Slf4j **public class** DemoServiceImpl **implements** DemoService {  @Autowired  **private** ElasticsearchRestTemplate **elasticsearchRestTemplate**;  @Override  **public** List<LogPojo> showLog() {  ***log***.error(**"自己写的日志"**);  *// 获取当前时间* Calendar instance = Calendar.*getInstance*();  instance.add(Calendar.***MINUTE***,-15);  NativeSearchQuery query = **new** NativeSearchQuery(QueryBuilders.*rangeQuery*(**"@timestamp"**).gte(instance.getTime()));  SearchHits<LogPojo> search = **elasticsearchRestTemplate**.search(query, LogPojo.**class**);  List<SearchHit<LogPojo>> histList = search.getSearchHits();  List<LogPojo> list = **new** ArrayList<>();  histList.forEach(sh -> {  LogPojo lp = sh.getContent();  **try {  String msg = lp.getMessage();  ObjectMapper objectMapper = new ObjectMapper();  MessagePojo mp = objectMapper.readValue(msg, MessagePojo.class);  lp.setMp(mp);  } catch (JsonProcessingException e) {  e.printStackTrace();  }**  list.add(lp);  });  **return** list;  } } |