**网络爬虫**

**第三天**

# 课程计划

1. 案例扩展
   1. 定时任务
   2. 网页去重
   3. 代理的使用
2. ElasticSearch环境准备
3. Spring Data ElasticSearch回顾
   1. 完成ES基本使用
   2. 完成复杂查询
4. 查询案例实现

# 案例扩展

## 定时任务

在案例中我们使用的是Spring内置的Spring Task，这是Spring3.0加入的定时任务功能。我们使用注解的方式定时启动爬虫进行数据爬取。

我们使用的是@Scheduled注解，其属性如下：

1）**cron：**cron表达式，指定任务在特定时间执行；

2）**fixedDelay**：上一次任务执行完后多久再执行，参数类型为long，单位ms

3）**fixedDelayString：**与fixedDelay含义一样，只是参数类型变为String

4）**fixedRate：**按一定的频率执行任务，参数类型为long，单位ms

5）**fixedRateString:** 与fixedRate的含义一样，只是将参数类型变为String

6）**initialDelay：**延迟多久再第一次执行任务，参数类型为long，单位ms

7）**initialDelayString：**与initialDelay的含义一样，只是将参数类型变为String

8）**zone：**时区，默认为当前时区，一般没有用到

我们这里的使用比较简单，固定的间隔时间来启动爬虫。例如可以实现项目启动后，每隔一小时启动一次爬虫。

但是有可能业务要求更高，并不是定时定期处理，而是在特定的时间进行处理，这个时候我们之前的使用方式就不能满足需求了。例如我要在工作日（周一到周五）的晚上八点执行。这时我们就需要Cron表达式了。

### Cron表达式

cron的表达式是字符串，实际上是由七子表达式，描述个别细节的时间表。这些子表达式是分开的空白，代表：

1. Seconds

2. Minutes

3. Hours

4. Day-of-Month

5. Month

6. Day-of-Week

7. Year (可选字段)

例 "0 0 12 ? \* WED" 在每星期三下午12:00 执行,

“\*” 代表整个时间段

每一个字段都有一套可以指定有效值，如

Seconds (秒) ：可以用数字0－59 表示，

Minutes(分) ：可以用数字0－59 表示，

Hours(时) ：可以用数字0-23表示,

Day-of-Month(天) ：可以用数字1-31 中的任一一个值，但要注意一些特别的月份

Month(月) ：可以用0-11 或用字符串:

JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC

Day-of-Week(天) ：可以用数字1-7表示（1 ＝ 星期日）或用字符口串:

SUN, MON, TUE, WED, THU, FRI, SAT

“/”：为特别单位，表示为“每”如“0/15”表示每隔15分钟执行一次,“0”表示为从“0”分开始, “3/20”表示表示每隔20分钟执行一次，“3”表示从第3分钟开始执行

“?”：表示每月的某一天，或第周的某一天

“L”：用于每月，或每周，表示为每月的最后一天，或每个月的最后星期几如“6L”表示“每月的最后一个星期五”

可以使用课堂资料的CronExpBuilder(表达式生成器)生成表达式

### Cron测试

先把之前爬虫的@Component注解取消，避免干扰测试

//@Component  
public class JobProcessor implements PageProcessor {

编写使用Cron表达式的测试用例：

@Component  
public class TaskTest {  
  
 @Scheduled(cron = "0/5 \* \* \* \* \*")  
 public void test() {  
 System.*out*.println(LocalDateTime.*now*()+"任务执行了");  
 }  
}

## 网页去重

之前我们对下载的url地址进行了去重操作，避免同样的url下载多次。其实不光url需要去重，我们对下载的内容也需要去重。

在网上我们可以找到许多内容相似的文章。但是实际我们只需要其中一个即可，同样的内容没有必要下载多次，那么如何进行去重就需要进行处理了

### 去重方案介绍

指纹码对比

最常见的去重方案是生成文档的指纹门。例如对一篇文章进行MD5加密生成一个字符串，我们可以认为这是文章的指纹码，再和其他的文章指纹码对比，一致则说明文章重复。

但是这种方式是完全一致则是重复的，如果文章只是多了几个标点符号，那仍旧被认为是重复的，这种方式并不合理。

BloomFilter

这种方式就是我们之前对url进行去重的方式，使用在这里的话，也是对文章进行计算得到一个数，再进行对比，缺点和方法1是一样的，如果只有一点点不一样，也会认为不重复，这种方式不合理。

KMP算法

KMP算法是一种改进的字符串匹配算法。KMP算法的关键是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。能够找到两个文章有哪些是一样的，哪些不一样。

这种方式能够解决前面两个方式的“只要一点不一样就是不重复”的问题。但是它的时空复杂度太高了，不适合大数据量的重复比对。

还有一些其他的去重方式：最长公共子串、后缀数组、字典树、DFA等等，但是这些方式的空复杂度并不适合数据量较大的工业应用场景。我们需要找到一款性能高速度快，能够进行相似度对比的去重方案

Google 的 simhash 算法产生的签名，可以满足上述要求。这个算法并不深奥，比较容易理解。这种算法也是目前Google搜索引擎所目前所使用的网页去重算法。

### SimHash

#### 流程介绍

simhash是由 Charikar 在2002年提出来的，为了便于理解尽量不使用数学公式，分为这几步：

**1、分词**，把需要判断文本分词形成这个文章的特征单词。

**2、hash**，通过hash算法把每个词变成hash值，比如“美国”通过hash算法计算为 100101,“51区”通过hash算法计算为 101011。这样我们的字符串就变成了一串串数字。

**3、加权**，通过 2步骤的hash生成结果，需要按照单词的权重形成加权数字串，“美国”的hash值为“100101”，通过加权计算为“4 -4 -4 4 -4 4”

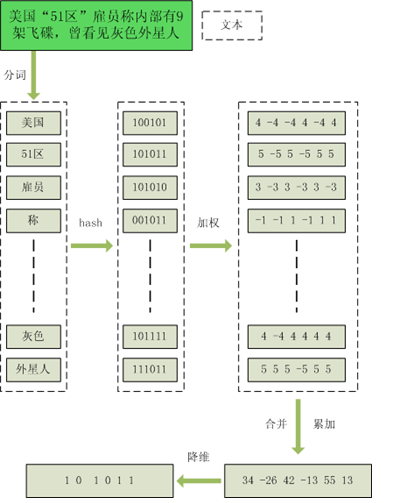
“51区”计算为 “ 5 -5 5 -5 5 5”。

**4、合并**，把上面各个单词算出来的序列值累加，变成只有一个序列串。

“美国”的 “4 -4 -4 4 -4 4”，“51区”的 “ 5 -5 5 -5 5 5”

把每一位进行累加， “4+5 -4+-5 -4+5 4+-5 -4+5 4+5”🡪“9 -9 1 -1 1 9”

**5、降维**，把算出来的 “9 -9 1 -1 1 9”变成 0 1 串，形成最终的simhash签名。



#### 签名距离计算

我们把库里的文本都转换为simhash签名，并转换为long类型存储，空间大大减少。现在我们虽然解决了空间，但是如何计算两个simhash的相似度呢？

我们通过海明距离（Hamming distance）就可以计算出两个simhash到底相似不相似。两个simhash对应二进制（01串）取值不同的数量称为这两个simhash的海明距离。

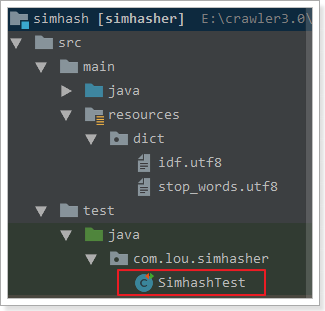
举例如下： 10101 和 00110 从第一位开始依次有第一位、第四、第五位不同，则海明距离为3。对于二进制字符串的a和b，海明距离为等于在a XOR b运算结果中1的个数（普遍算法）。

#### 导入simhash工程

参考项目：<https://github.com/CreekLou/simhash.git>

这个项目不能直接使用，因为jar包的问题，需要进行改造。这里使用课堂资料中已经改造好的。

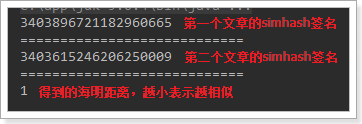
导入工程simhash，并打开测试用例。



#### 测试simhash

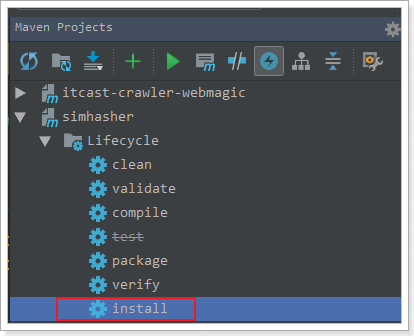
按照测试用例的要求，准备两个文件，就是需要进行对比的文章

执行测试用例，结果如下



#### 案例整合

需要先把simhash安装到本地仓库



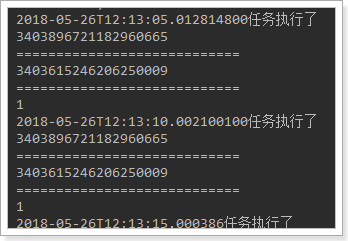
在案例的pom.xml中加入以下依赖

<!--simhash网页去重-->  
<dependency>  
 <groupId>com.lou</groupId>  
 <artifactId>simhasher</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
</dependency>

修改代码

@Component  
public class TaskTest {  
  
 @Scheduled(cron = "0/5 \* \* \* \* \*")  
 public void test() {  
 System.*out*.println(LocalDateTime.*now*()+"任务执行了");  
  
 String str1 = *readAllFile*("D:/test/testin.txt");  
 SimHasher hash1 = new SimHasher(str1);  
 //打印simhash签名  
 System.*out*.println(hash1.getSignature());  
 System.*out*.println("============================");  
  
 String str2 = *readAllFile*("D:/test/testin2.txt");  
 //打印simhash签名  
 SimHasher hash2 = new SimHasher(str2);  
 System.*out*.println(hash2.getSignature());  
 System.*out*.println("============================");  
  
 //打印海明距离 System.*out*.println(hash1.getHammingDistance(hash2.getSignature()));  
 }  
  
 public static String readAllFile(String filename) {  
 String everything = "";  
 try {  
 FileInputStream inputStream = new FileInputStream(filename);  
 everything = IOUtils.*toString*(inputStream);  
 inputStream.close();  
 } catch (IOException e) {  
 }  
 return everything;  
 }  
}

启动项目控制台显示：



## 代理的使用

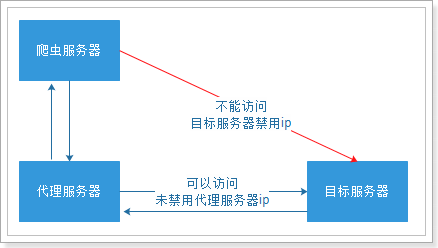
有些网站不允许爬虫进行数据爬取，因为会加大服务器的压力。其中一种最有效的方式是通过ip+时间进行鉴别，因为正常人不可能短时间开启太多的页面，发起太多的请求。

我们使用的WebMagic可以很方便的设置爬取数据的时间（参考第二天的的3.1. 爬虫的配置、启动和终止）。但是这样会大大降低我们爬取数据的效率，如果不小心ip被禁了，会让我们无法爬去数据，那么我们就有必要使用代理服务器来爬取数据。

### 代理服务器

代理（英语：Proxy），也称网络代理，是一种特殊的网络服务，允许一个网络终端（一般为客户端）通过这个服务与另一个网络终端（一般为服务器）进行非直接的连接。

提供代理服务的电脑系统或其它类型的网络终端称为代理服务器（英文：Proxy Server）。一个完整的代理请求过程为：客户端首先与代理服务器创建连接，接着根据代理服务器所使用的代理协议，请求对目标服务器创建连接、或者获得目标服务器的指定资源。



我们就需要知道代理服务器在哪里（ip和端口号）才可以使用。网上有很多代理服务器的提供商，但是大多是免费的不好用，付费的还行。

提供两个免费代理ip的服务商网站：

米扑代理

<https://proxy.mimvp.com/free.php>

西刺免费代理IP

<http://www.xicidaili.com/>

### 使用代理

WebMagic使用的代理APIProxyProvider。因为相对于Site的“配置”，ProxyProvider定位更多是一个“组件”，所以代理不再从Site设置，而是由HttpClientDownloader设置。

| **API** | **说明** |
| --- | --- |
| HttpClientDownloader.setProxyProvider(ProxyProvider proxyProvider) | 设置代理 |

ProxyProvider有一个默认实现：SimpleProxyProvider。它是一个基于简单Round-Robin的、没有失败检查的ProxyProvider。可以配置任意个候选代理，每次会按顺序挑选一个代理使用。它适合用在自己搭建的比较稳定的代理的场景。

如果需要根据实际使用情况对代理服务器进行管理（例如校验是否可用，定期清理、添加代理服务器等），只需要自己实现APIProxyProvider即可。

可以访问网址<http://ip.chinaz.com/getip.aspx> 测试当前请求的ip

为了避免干扰，把其他任务的@Component注释掉，在案例中加入编写以下逻辑：

@Component  
public class ProxyTest implements PageProcessor {  
  
 @Scheduled(fixedDelay = 10000)  
 public void testProxy() {  
 HttpClientDownloader httpClientDownloader = new HttpClientDownloader();  
 httpClientDownloader.setProxyProvider(SimpleProxyProvider.*from*(new Proxy("39.137.77.68",80)));  
  
 Spider.*create*(new ProxyTest())  
 .addUrl("http://ip.chinaz.com/getip.aspx")  
 .setDownloader(httpClientDownloader)  
 .run();  
 }  
  
 @Override  
 public void process(Page page) {  
 //打印获取到的结果以测试代理服务器是否生效  
 System.*out*.println(page.getHtml());  
 }  
  
 private Site site = new Site();  
 @Override  
 public Site getSite() {  
 return site;  
 }  
}

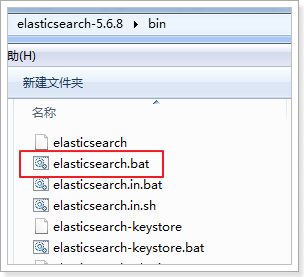
# ElasticSearch环境准备

## 安装ElasticSearch服务

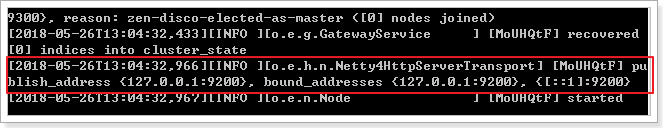
课堂资料中的elasticsearch-5.6.8.zip进行解压



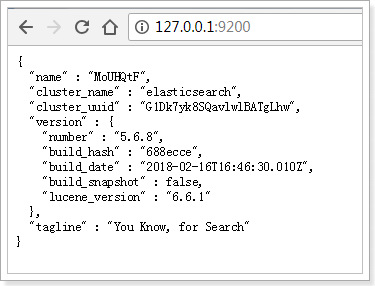
启动服务：



当出现以下内容表示启动完成



访问地址是<http://127.0.0.1:9200> 访问该地址：



表示ElasticSearch安装启动完成

## 安装ES的图形化界面插件

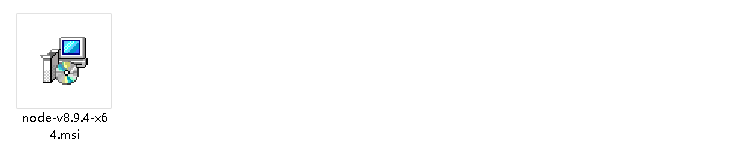
安装ElasticSearch的head插件，完成图形化界面的效果，完成索引数据的查看。采用本地安装方式进行head插件的安装。elasticsearch-5-\*以上版本安装head需要安装node和grunt。

1）安装head插件

将head压缩包解压到任意目录，但是要和elasticsearch的安装目录区别开



2）安装nodejs

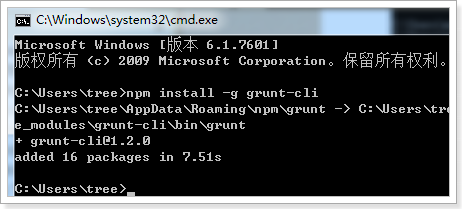


3）将grunt安装为全局命令 ，Grunt是基于Node.js的项目构建工具

在cmd控制台中输入如下执行命令：

npm install -g grunt-cli

效果如下：



ps:如果安装不成功或者安装速度慢，可以使用淘宝的镜像进行安装：

npm install -g cnpm –registry=https://registry.npm.taobao.org

后续使用的时候，只需要把npm xxx 换成 cnpm xxx 即可

4）修改elasticsearch配置文件：elasticsearch.yml，增加以下三句命令：

http.cors.enabled: true

http.cors.allow-origin: "\*"

network.host: 127.0.0.1

重启

5）进入head目录启动head，在命令提示符下输入命令：

grunt server

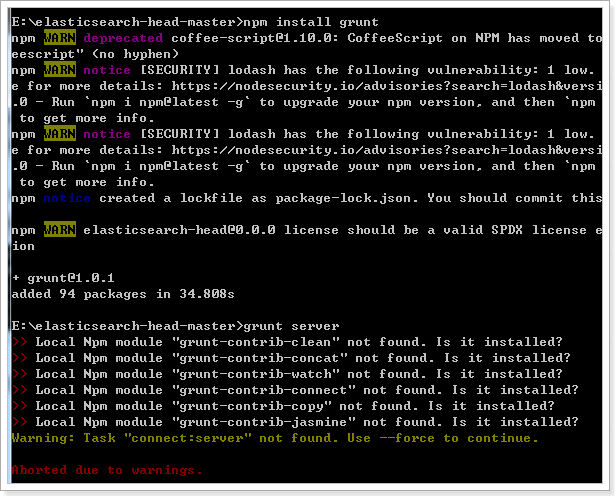


根据提示访问，效果如下：

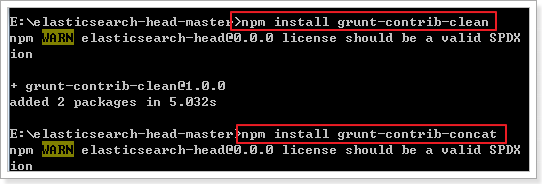


PS：如果第5步失败，执行以下命令

npm install grunt

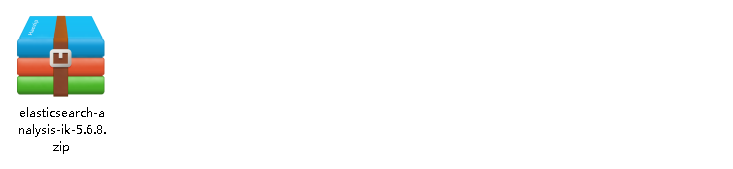


再根据提示按以下方式依次安装组件

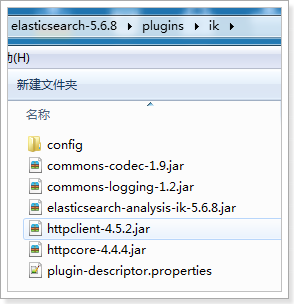


## 安装IK分词器

1. IK分词器安装包在课堂资料



1. 解压，将解压后的elasticsearch文件夹拷贝到elasticsearch-5.6.8\plugins下，并重命名文件夹为ik



1. 重新启动ElasticSearch，即可加载IK分词器
2. 测试

在浏览器发起以下请求

1）最小切分：在浏览器地址栏输入地址

http://127.0.0.1:9200/\_analyze?analyzer=ik\_smart&pretty=true&text=我是程序员

浏览器显示

{

"tokens" : [

{

"token" : "我",

"start\_offset" : 0,

"end\_offset" : 1,

"type" : "CN\_CHAR",

"position" : 0

},

{

"token" : "是",

"start\_offset" : 1,

"end\_offset" : 2,

"type" : "CN\_CHAR",

"position" : 1

},

{

"token" : "程序员",

"start\_offset" : 2,

"end\_offset" : 5,

"type" : "CN\_WORD",

"position" : 2

}

]

}

# ElasticSearch回顾

## 创建Maven工程

创建Maven工程，给pom.xml加入依赖：

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>cn.itcast</groupId>  
 <artifactId>itcast-es</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.elasticsearch</groupId>  
 <artifactId>elasticsearch</artifactId>  
 <version>5.6.8</version>  
 </dependency>  
 <dependency>  
 <groupId>org.elasticsearch.client</groupId>  
 <artifactId>transport</artifactId>  
 <version>5.6.8</version>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.logging.log4j</groupId>  
 <artifactId>log4j-to-slf4j</artifactId>  
 <version>2.9.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.7.24</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-simple</artifactId>  
 <version>1.7.21</version>  
 </dependency>  
 <dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>1.2.12</version>  
 </dependency>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 </dependency>  
 <dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-core</artifactId>  
 <version>2.8.1</version>  
 </dependency>  
 <dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-databind</artifactId>  
 <version>2.8.1</version>  
 </dependency>  
 <dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-annotations</artifactId>  
 <version>2.8.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.data</groupId>  
 <artifactId>spring-data-elasticsearch</artifactId>  
 <version>3.0.5.RELEASE</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-test</artifactId>  
 <version>5.0.4.RELEASE</version>  
 </dependency>  
 </dependencies>  
  
</project>

添加配置文件applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd  
 http://www.springframework.org/schema/data/elasticsearch  
 http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd">  
  
</beans>

## 开发准备

### 编写pojo

public class Item {  
  
 private Integer id;  
 private String title;  
 private String content;  
get/set

toString()  
}

### 编写dao

public interface ItemRepository extends ElasticsearchRepository<Item, Integer> {  
}

### 编写service

编写service接口

public interface ItemService {  
  
}

编写service实现

@Service  
public class ItemServiceImpl implements ItemService {  
  
}

### 修改配置文件

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd  
 http://www.springframework.org/schema/data/elasticsearch  
 http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd">  
  
 <!-- 扫描Dao包，自动创建实例 -->  
 <elasticsearch:repositories base-package="com.itheima.dao"/>  
  
 <!-- 扫描Service包，创建Service的实体 -->  
 <context:component-scan base-package="cn.itcast.es.service"/>  
  
 <!-- 配置elasticSearch的连接 -->  
 <elasticsearch:transport-client id="client" cluster-nodes="localhost:9300"/>  
  
 <!-- spring data elasticSearcheDao 必须继承 ElasticsearchTemplate -->  
 <bean id="elasticsearchTemplate"  
 class="org.springframework.data.elasticsearch.core.ElasticsearchTemplate">  
 <constructor-arg name="client" ref="client"/>  
 </bean>  
  
</beans>

### 修改实体类

@Document(indexName = "item", type = "item")  
public class Item {  
  
 @Id  
 @Field(index = true, store = true, type = FieldType.*Integer*)  
 private Integer id;   
 @Field(index = true, store = true, analyzer = "ik\_smart", searchAnalyzer = "ik-smart", type = FieldType.*text*)  
 private String title;  
 @Field(index = true, store = true, analyzer = "ik\_smart", searchAnalyzer = "ik-smart", type = FieldType.*text*)  
 private String content;  
  
 public Integer getId() {  
 return id;  
get/set

toString();

}

## ElasticSearch基本使用

### 保存和修改文档

在pojo中设置了id为索引库的主键，索引根据id进行保存或修改。

如果id存在则修改，如果id不存在则更新

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(locations = "classpath:applicationContext.xml")  
public class SpringDataESTest {  
 @Autowired  
 private ItemService itemService;  
 @Autowired  
 private ElasticsearchTemplate elasticsearchTemplate;  
  
 */\*\*  
 \* 创建索引和映射  
 \*/* @Test  
 public void createIndex() {  
 this.elasticsearchTemplate.createIndex(Item.class);  
 this.elasticsearchTemplate.putMapping(Item.class);  
 }  
  
 */\*\*  
 \* 测试保存文档  
 \*/* @Test  
 public void saveArticle() {  
 Item item = new Item();  
 item.setId(100);  
 item.setTitle("测试SpringData ElasticSearch");  
 item.setContent("Spring Data ElasticSearch 基于 spring data API 简化操作，实现搜索引擎功能");  
 this.itemService.save(item);  
 }  
  
  
 */\*\*  
 \* 测试更新  
 \*/* @Test  
 public void update() {  
 Item item = new Item();  
 item.setId(100);  
 item.setTitle("elasticSearch 3.0版本发布...更新");  
 item.setContent("ElasticSearch是一个基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口");  
 this.itemService.save(item);  
 }  
}

在ItemService中添加Service接口方法

*/\*\*  
 \* 保存  
 \*  
 \** ***@param*** *item  
 \*/*void save(Item item);

在ItemServiceImpl添加Service实现方法

@Autowired  
private ItemRepository itemRepository;  
  
public void save(Item item) {  
 this.itemRepository.save(item);  
}

### 删除文档

在测试用例中SpringDataESTest中添加测试逻辑

*/\*\*  
 \* 测试删除  
 \*  
 \*/*@Test  
public void delete() {  
 Item item = new Item();  
 item.setId(100);  
 this.itemService.delete(item);  
}

在ItemService中添加Service接口方法

*/\*\*  
 \* 删除  
 \*  
 \** ***@param*** *item  
 \*/*void delete(Item item);

在ItemServiceImpl添加Service实现方法

public void delete(Item item) {  
 this.itemRepository.delete(item);  
}

### 批量保存

在测试用例中SpringDataESTest中添加测试逻辑

*/\*\*  
 \* 批量保存  
 \*  
 \*/*@Test  
public void saveAll(){  
 List<Item> items = new ArrayList<Item>();  
 for(int i=1;i<=100;i++){  
 Item item = new Item();  
 item.setId(i);  
 item.setTitle(i+"elasticSearch 3.0版本发布..，更新");  
 item.setContent(i+"ElasticSearch批量插入"+i);  
 items.add(item);  
 }  
 this.itemService.saveAll(items);  
}

在ItemService中添加Service接口方法

*/\*\*  
 \* 批量保存  
 \*  
 \** ***@param*** *items  
 \*/*void saveAll(List<Item> items);

在ItemServiceImpl添加Service实现方法

public void saveAll(List<Item> items) {  
 this.itemRepository.saveAll(items);  
}

### 查询所有

在测试用例中SpringDataESTest中添加测试逻辑

*/\*\*  
 \* 查询所有  
 \*/*@Test  
public void findAll(){  
 Iterable<Item> list = itemService.findAll();  
 for(Item article:list){  
 System.*out*.println(article);  
 }  
}

在ItemService中添加Service接口方法

*/\*\*  
 \* 查询所有  
 \*  
 \** ***@return*** *\*/*Iterable<Item> findAll();

在ItemServiceImpl添加Service实现方法

public Iterable<Item> findAll() {  
 Iterable<Item> items = this.itemRepository.findAll();  
 return items;  
}

### 分页查询

在测试用例中SpringDataESTest中添加测试逻辑

*/\*\*  
 \* 分页查询  
 \*/*@Test  
public void findAllPage(){  
 Page<Item> page = itemService.findAllPage(1,10);  
 for(Item article:page.getContent()){  
 System.*out*.println(article);  
 }  
}

在ItemService中添加Service接口方法

*/\*\*  
 \* 分页查询  
 \** ***@param*** *page  
 \** ***@param*** *rows  
 \** ***@return*** *\*/* Page<Item> findAllPage(Integer page,Integer rows);  
}

在ItemServiceImpl添加Service实现方法

public Page<Item> findAllPage(Integer page,Integer rows) {  
 Page<Item> result = this.itemRepository.findAll(PageRequest.*of*(page,rows));  
 return result;  
}

## ElasticSearch复杂查询

### 查询方法示例

|  |  |  |  |
| --- | --- | --- | --- |
| **关键字** | **命名规则** | **解释** | **示例** |
| and | findByField1AndField2 | 根据Field1和Field2获得数据 | findByTitleAndContent |
| or | findByField1OrField2 | 根据Field1或Field2获得数据 | findByTitleOrContent |
| is | findByField | 根据Field获得数据 | findByTitle |
| not | findByFieldNot | 根据Field获得补集数据 | findByTitleNot |
| between | findByFieldBetween | 获得指定范围的数据 | findByPriceBetween |
| lessThanEqual | findByFieldLessThan | 获得小于等于指定值的数据 | findByPriceLessThan |

### 根据title和Content查询

默认每页显示10条数据

在测试用例中SpringDataESTest中添加测试逻辑

*/\*\*  
 \* 根据title和Content查询  
 \*/*@Test  
public void findByTitleAndContent() {  
 List<Item> list = itemService.findByTitleAndContent("更新", "批量");  
 for (Item item : list) {  
 System.*out*.println(item);  
 }  
}

在ItemService中添加Service接口方法

*/\*\*  
 \* 根据Title和Content查询，交集  
 \*  
 \** ***@param*** *title  
 \** ***@param*** *content  
 \** ***@return*** *\*/*public List<Item> findByTitleAndContent(String title, String content);

在ItemServiceImpl添加Service实现方法

public List<Item> findByTitleAndContent(String title, String content) {  
 List<Item> list = this.itemRepository.findByTitleAndContent(title, content);  
 return list;  
}

在ItemRepository添加方法

*/\*\*  
 \* 根据Title和Content查询，交集  
 \*  
 \** ***@param*** *title  
 \** ***@param*** *content  
 \** ***@return*** *\*/*public List<Item> findByTitleAndContent(String title, String content);

### 根据title和Content分页查询

在测试用例中SpringDataESTest中添加测试逻辑

*/\*\*  
 \* 根据title和Content分页查询  
 \*/*@Test  
public void findByTitleOrContent() {  
 Page<Item> page = itemService.findByTitleOrContent("版本", "版本", 1, 20);  
 for (Item item : page.getContent()) {  
 System.*out*.println(item);  
 }  
}

在ItemService中添加Service接口方法

*/\*\*  
 \* 根据Title或Content分页查询，并集  
 \*  
 \** ***@param*** *title  
 \** ***@param*** *content  
 \** ***@param*** *page  
 \** ***@param*** *rows  
 \** ***@return*** *\*/*public Page<Item> findByTitleOrContent(String title, String content, Integer page, Integer rows);

在ItemServiceImpl添加Service实现方法

public Page<Item> findByTitleOrContent(String title, String content, Integer page, Integer rows) {  
 Page<Item> pages = this.itemRepository.findByTitleOrContent(title, content, PageRequest.*of*(page, rows));  
 return pages;  
}

在ItemRepository添加方法

*/\*\*  
 \* 根据Title或Content分页查询，并集**\*/*public Page<Item> findByTitleOrContent(String title, String content, Pageable pageable);

### 根据多条件组合查询

在测试用例中SpringDataESTest中添加测试逻辑

*/\*\*  
 \* 根据title和Content和id范围分页查询  
 \*/*@Test  
public void findByIdBetween() {  
 Page<Item> items = itemService.findByTitleAndContentAndIdBetween("版本", "批量", 31l, 80l, 1, 33);  
 for (Item item : items.getContent()) {  
 System.*out*.println(item);  
 }  
}

在ItemService中添加Service接口方法

*/\*\*  
 \* 根据title和Content和id范围分页查询  
 \***\*/*public Page<Item> findByTitleAndContentAndIdBetween(String title, String Content, Long min, Long max, Integer page, Integer rows);

在ItemServiceImpl添加Service实现方法

public Page<Item> findByTitleAndContentAndIdBetween(String title, String Content, Long min, Long max, Integer page, Integer rows) {  
 Page<Item> items = this.itemRepository.findByTitleAndContentAndIdBetween(title  
 , Content, min, max, PageRequest.*of*(page, rows));  
 return items;  
}

在ItemRepository添加方法

*/\*\*  
 \* 根据title和Content和id范围分页查询  
 \***\*/*public Page<Item> findByTitleAndContentAndIdBetween(String title, String Content, Long min, Long max, Pageable pageable);

# 查询案例实现

把上一次上课抓取到的招聘数据作为数据源，实现招聘信息查询功能。首先需要把MySQL的数据添加到索引库中，然后再实现查询功能。我们这里使用的是SpringBoot，需要把Spring Data ElasticSearch 和项目进行整合。

## 开发准备

我们这里使用的是SpringBoot，需要把Spring Data ElasticSearch 和项目进行整合

需要修改之前的配置，网页去重排除lucene依赖，同时去重的依赖必须放在pom.xml的最下部。因为现在要使用ElasticSearch，需要用到新的lucene依赖。

添加ES依赖和单元测试依赖，并修改以前的去重依赖，pom.xml效果：

<!--ElasticSearch-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-elasticsearch</artifactId>  
</dependency>  
  
<!--单元测试-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
</dependency>

<!--simhash网页去重-->  
<dependency>  
 <groupId>com.lou</groupId>  
 <artifactId>simhasher</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <exclusions>  
 <exclusion>  
 <groupId>org.apache.lucene</groupId>  
 <artifactId>lucene-core</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>

修改配置文件application.properties，添加以下内容

#ElasticSearch Configuration  
spring.data.elasticsearch.cluster-nodes=127.0.0.1:9300

## 导入数据到索引库

### 编写pojo

@Document(indexName = "jobinfo", type = "jobInfoField")  
public class JobInfoField {  
  
 @org.springframework.data.annotation.Id  
 @Field(index = true, store = true, type = FieldType.*Long*)  
 private Long id;  
 @Field(index = false, store = true, type = FieldType.*Text*)  
 private String companyName;   
 @Field(index = false, store = true, type = FieldType.*Text*)  
 private String companyAddr;   
 @Field(index = false, store = true, type = FieldType.*Text*)  
 private String companyInfo;  
 @Field(index = true, store = true, analyzer = "ik\_smart", searchAnalyzer = "ik\_smart", type = FieldType.*Text*)  
 private String jobName;  
 @Field(index = true, store = true, analyzer = "ik\_smart", searchAnalyzer = "ik\_smart", type = FieldType.*Text*)  
 private String jobAddr;  
 @Field(index = true, store = false, analyzer = "ik\_smart", searchAnalyzer = "ik\_smart", type = FieldType.*Text*)  
 private String jobInfo;  
 @Field(index = true, store = true, type = FieldType.*Integer*)  
 private Integer salaryMin;  
 @Field(index = true, store = true, type = FieldType.*Integer*)  
 private Integer salaryMax;  
 private String url;  
 @Field(index = true, store = true, type = FieldType.*Text*)  
 private String time;  
get/set

toString()  
}

### 编写dao

@Component  
public interface JobRepository extends ElasticsearchRepository<JobInfoField, Long> {  
}

### 编写Service

编写Service接口

public interface JobRepositoryService {  
  
  
 */\*\*  
 \* 保存一条数据  
 \*  
 \** ***@param*** *jobInfoField  
 \*/* void save(JobInfoField jobInfoField);  
  
 */\*\*  
 \* 批量保存数据  
 \*  
 \** ***@param*** *list  
 \*/* void saveAll(List<JobInfoField> list);  
}

编写Service实现类

@Service  
public class JobRepositoryServiceImpl implements JobRepositoryService {  
  
 @Autowired  
 private JobRepository jobRepository;  
  
 @Override  
 public void save(JobInfoField jobInfoField) {  
 this.jobRepository.save(jobInfoField);  
 }  
  
 @Override  
 public void saveAll(List<JobInfoField> list) {  
 this.jobRepository.saveAll(list);  
 }  
}

### 编写测试用例

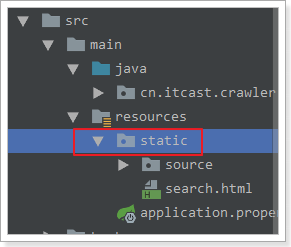
先执行createIndex()方法创建索引，再执行jobData()导入数据到索引库

@RunWith(SpringJUnit4ClassRunner.class)  
@SpringBootTest(classes = Application.class)  
public class ElasticSearchTest {  
  
 @Autowired  
 private JobInfoService jobInfoService;  
 @Autowired  
 private JobRepositoryService jobRepositoryService;  
 @Autowired  
 private ElasticsearchTemplate elasticsearchTemplate;  
  
 */\*\*  
 \* 创建索引和映射  
 \*/* @Test  
 public void createIndex() {  
 this.elasticsearchTemplate.createIndex(JobInfoField.class);  
 this.elasticsearchTemplate.putMapping(JobInfoField.class);  
 }  
  
  
 @Test  
 public void jobData() {  
 //声明当前页码数  
 int count = 0;  
 //声明查询数据条数  
 int pageSize = 0;  
  
 //循环查询  
 do {  
 //从MySQL数据库中分页查询数据  
 Page<JobInfo> page = this.jobInfoService.findAllPage(count, 500);  
  
 //声明存放索引库数据的容器  
 List<JobInfoField> list = new ArrayList<>();  
  
 //遍历查询结果  
 for (JobInfo jobInfo : page.getContent()) {  
 //创建存放索引库数据的对象  
 JobInfoField jobInfoField = new JobInfoField();  
 //复制数据  
 BeanUtils.*copyProperties*(jobInfo, jobInfoField);  
 //把复制好的数据放到容器中  
 list.add(jobInfoField);  
 }  
  
 //批量保存数据到索引库中  
 this.jobRepositoryService.saveAll(list);  
  
 //页面数加一  
 count++;  
 //获取查询数据条数  
 pageSize = page.getContent().size();  
  
 } while (pageSize == 500);  
 }  
}

## 查询案例实现

### 页面跳转实现

添加课堂资料的静态资源到项目中



### 编写pojo

public class JobResult {  
  
 private List<JobInfoField> rows;  
 private Integer pageTotal;

get/set  
}

### 编写Controller

@RestController  
public class SearchController {  
  
 @Autowired  
 private JobRepositoryService jobRepositoryService;  
  
 */\*\*  
 \* 根据条件分页查询数据  
 \** ***@param*** *salary  
 \** ***@param*** *jobaddr  
 \** ***@param*** *keyword  
 \** ***@param*** *page  
 \** ***@return*** *\*/* @RequestMapping(value = "search", method = RequestMethod.*POST*)  
 public JobResult search(String salary, String jobaddr, String keyword, Integer page) {  
 JobResult jobResult = this.jobRepositoryService.search(salary, jobaddr, keyword, page);  
 return jobResult;  
 }  
}

### 编写Service

在JobRepositoryService编写接口方法

*/\*\*  
 \*  
 \** ***@param*** *salary  
 \** ***@param*** *jobaddr  
 \** ***@param*** *keyword  
 \** ***@param*** *page  
 \** ***@return*** *\*/*JobResult search(String salary, String jobaddr, String keyword, Integer page);

在JobRepositoryServiceImpl实现接口方法

@Override  
public JobResult search(String salary, String jobaddr, String keyword, Integer page) {  
 //薪资处理 20-\*  
 int salaryMin = 0, salaryMax = 0;  
 String[] salays = salary.split("-");  
 //获取最小值  
 if ("\*".equals(salays[0])) {  
 salaryMin = 0;  
 } else {  
 salaryMin = Integer.*parseInt*(salays[0]) \* 10000;  
 }  
  
 //获取最大值  
 if ("\*".equals(salays[1])) {  
 salaryMax = 900000000;  
 } else {  
 salaryMax = Integer.*parseInt*(salays[1]) \* 10000;  
 }  
  
 //工作地址如果为空，就设置为\*  
 if (StringUtils.*isBlank*(jobaddr)) {  
 jobaddr = "\*";  
  
 //查询关键词为空，就设置为\*  
 } if (StringUtils.*isBlank*(keyword)) {  
 keyword = "\*";  
 }  
  
  
 //获取分页,设置每页显示30条数据  
 Pageable pageable = PageRequest.*of*(page - 1, 30);  
  
 //执行查询  
 Page<JobInfoField> pages = this.jobRepository  
 .findBySalaryMinBetweenAndSalaryMaxBetweenAndJobAddrAndJobNameAndJobInfo(salaryMin,  
 salaryMax, salaryMin, salaryMax, jobaddr, keyword, keyword, pageable);  
  
 //封装结果  
 JobResult jobResult = new JobResult();  
 jobResult.setRows(pages.getContent());  
 jobResult.setPageTotal(pages.getTotalPages());  
  
 return jobResult;  
}

### 编写Dao

在JobRepository编写接口方法

*/\*\*  
 \* 根据条件分页查询数据  
 \*  
 \** ***@param*** *salaryMin1 薪资下限最小值  
 \** ***@param*** *salaryMin2 薪资下限最高值  
 \** ***@param*** *salaryMax1 薪资上限最小值  
 \** ***@param*** *salaryMax2 薪资上限最大值  
 \** ***@param*** *jobAddr 工作地点  
 \** ***@param*** *jobName 职位名称  
 \** ***@param*** *jobInfo 职位信息  
 \** ***@param*** *pageable 分页数据  
 \** ***@return*** *\*/*public Page<JobInfoField> findBySalaryMinBetweenAndSalaryMaxBetweenAndJobAddrAndJobNameAndJobInfo(Integer salaryMin1, Integer salaryMin2, Integer salaryMax1, Integer salaryMax2, String jobAddr, String jobName, String jobInfo, Pageable pageable);

测试结果：

