**网络爬虫**

**大全**

# 网络爬虫

网络爬虫（Web crawler），是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本

## 爬虫入门程序

### 环境准备

* JDK1.8
* IntelliJ IDEA
* IDEA自带的Maven

### 环境准备

创建Maven工程itcast-crawler-first并给pom.xml加入依赖

<dependencies>  
 <!-- HttpClient -->  
 <dependency>  
 <groupId>org.apache.httpcomponents</groupId>  
 <artifactId>httpclient</artifactId>  
 <version>4.5.3</version>  
 </dependency>  
  
 <!-- 日志 -->  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.7.25</version>  
 </dependency>  
</dependencies>

### 加入log4j.properties

log4j.rootLogger=DEBUG,A1  
log4j.logger.cn.itcast = DEBUG  
  
log4j.appender.A1=org.apache.log4j.ConsoleAppender  
log4j.appender.A1.layout=org.apache.log4j.PatternLayout  
log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n

### 编写代码

编写最简单的爬虫，抓取传智播客首页：http://www.itcast.cn/

public static void main(String[] args) throws Exception {  
 CloseableHttpClient httpClient = HttpClients.*createDefault*();  
  
 HttpGet httpGet = new HttpGet("http://www.itcast.cn/");  
  
 CloseableHttpResponse response = httpClient.execute(httpGet);  
  
 if (response.getStatusLine().getStatusCode() == 200) {  
 String content = EntityUtils.*toString*(response.getEntity(), "UTF-8");  
 System.*out*.println(content);  
 }  
}

测试结果：可以获取到页面数据

# 网络爬虫

## 网络爬虫介绍

在大数据时代，信息的采集是一项重要的工作，而互联网中的数据是海量的，如果单纯靠人力进行信息采集，不仅低效繁琐，搜集的成本也会提高。如何自动高效地获取互联网中我们感兴趣的信息并为我们所用是一个重要的问题，而爬虫技术就是为了解决这些问题而生的。

网络爬虫（Web crawler）也叫做网络机器人，可以代替人们自动地在互联网中进行数据信息的采集与整理。它是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本，可以自动采集所有其能够访问到的页面内容，以获取相关数据。

从功能上来讲，爬虫一般分为数据采集，处理，储存三个部分。爬虫从一个或若干初始网页的URL开始，获得初始网页上的URL，在抓取网页的过程中，不断从当前页面上抽取新的URL放入队列,直到满足系统的一定停止条件。

## 为什么学网络爬虫

我们初步认识了网络爬虫，但是为什么要学习网络爬虫呢？只有清晰地知道我们的学习目的，才能够更好地学习这一项知识。在此，总结了4种常见的学习爬虫的原因：

* + - 1. 可以实现搜索引擎

我们学会了爬虫编写之后，就可以利用爬虫自动地采集互联网中的信息，采集回来后进行相应的存储或处理，在需要检索某些信息的时候，只需在采集回来的信息中进行检索，即实现了私人的搜索引擎。

* + - 1. 大数据时代，可以让我们获取更多的数据源。

在进行大数据分析或者进行数据挖掘的时候，需要有数据源进行分析。我们可以从某些提供数据统计的网站获得，也可以从某些文献或内部资料中获得，但是这些获得数据的方式，有时很难满足我们对数据的需求，而手动从互联网中去寻找这些数据，则耗费的精力过大。此时就可以利用爬虫技术，自动地从互联网中获取我们感兴趣的数据内容，并将这些数据内容爬取回来，作为我们的数据源，再进行更深层次的数据分析，并获得更多有价值的信息。

* + - 1. 可以更好地进行搜索引擎优化（SEO）。

对于很多SEO从业者来说，为了更好的完成工作，那么就必须要对搜索引擎的工作原理非常清楚，同时也需要掌握搜索引擎爬虫的工作原理。

而学习爬虫，可以更深层次地理解搜索引擎爬虫的工作原理，这样在进行搜索引擎优化时，才能知己知彼，百战不殆。

* + - 1. 有利于就业。

从就业来说，爬虫工程师方向是不错的选择之一，因为目前爬虫工程师的需求越来越大，而能够胜任这方面岗位的人员较少，所以属于一个比较紧缺的职业方向，并且随着大数据时代和人工智能的来临，爬虫技术的应用将越来越广泛，在未来会拥有很好的发展空间。

# HttpClient

网络爬虫就是用程序帮助我们访问网络上的资源，我们一直以来都是使用HTTP协议访问互联网的网页，网络爬虫需要编写程序，在这里使用同样的HTTP协议访问网页。

这里我们使用Java的HTTP协议客户端 HttpClient这个技术，来实现抓取网页数据。

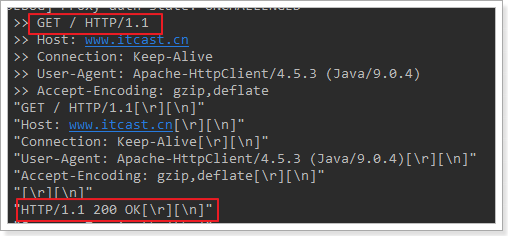
## GET请求

访问传智官网，请求url地址：

<http://www.itcast.cn/>

public static void main(String[] args) throws IOException {  
 //创建HttpClient对象  
 CloseableHttpClient httpClient = HttpClients.*createDefault*();  
  
 //创建HttpGet请求  
 HttpGet httpGet = new HttpGet("http://www.itcast.cn/");  
  
 CloseableHttpResponse response = null;  
 try {  
 //使用HttpClient发起请求  
 response = httpClient.execute(httpGet);  
  
 //判断响应状态码是否为200  
 if (response.getStatusLine().getStatusCode() == 200) {  
 //如果为200表示请求成功，获取返回数据  
 String content = EntityUtils.*toString*(response.getEntity(), "UTF-8");  
 //打印数据长度  
 System.*out*.println(content);  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 //释放连接  
 if (response == null) {  
 try {  
 response.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 httpClient.close();  
 }  
 }  
}

请求结果



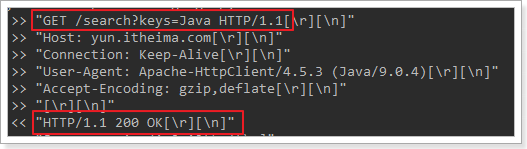
## 带参数的GET请求

在传智中搜索学习视频，地址为：

<http://yun.itheima.com/search?keys=Java>

public static void main(String[] args) throws IOException {  
 //创建HttpClient对象  
 CloseableHttpClient httpClient = HttpClients.*createDefault*();  
  
 //创建HttpGet请求，带参数的地址https://www.baidu.com/s?wd=HttpClient  
 String uri = "http://yun.itheima.com/search?keys=Java";  
 HttpGet httpGet = new HttpGet(uri);  
  
 CloseableHttpResponse response = null;  
 try {  
 //使用HttpClient发起请求  
 response = httpClient.execute(httpGet);  
  
 //判断响应状态码是否为200  
 if (response.getStatusLine().getStatusCode() == 200) {  
 //如果为200表示请求成功，获取返回数据  
 String content = EntityUtils.*toString*(response.getEntity(), "UTF-8");  
 //打印数据长度  
  
 System.*out*.println(content);  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 //释放连接  
 if (response == null) {  
 try {  
 response.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 httpClient.close();  
 }  
 }  
}

请求结果



## POST请求

使用POST访问传智官网，请求url地址：

<http://www.itcast.cn/>

public static void main(String[] args) throws IOException {  
 //创建HttpClient对象  
 CloseableHttpClient httpClient = HttpClients.*createDefault*();  
  
 //创建HttpGet请求  
 HttpPost httpPost = new HttpPost("http://www.itcast.cn/");  
  
  CloseableHttpResponse response = null;  
 try {  
 //使用HttpClient发起请求  
  response = httpClient.execute(httpPost);  
  
 //判断响应状态码是否为200  
 if (response.getStatusLine().getStatusCode() == 200) {  
 //如果为200表示请求成功，获取返回数据  
 String content = EntityUtils.*toString*(response.getEntity(), "UTF-8");  
 //打印数据长度  
 System.*out*.println(content);  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 //释放连接  
 if (response == null) {  
 try {  
 response.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 httpClient.close();  
 }  
 }  
}

请求结果：



## 带参数的POST请求

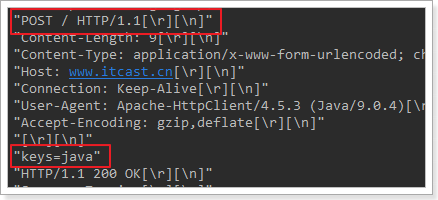
在传智中搜索学习视频，使用POST请求，url地址为：

<http://yun.itheima.com/search>

url地址没有参数，参数keys=java放到表单中进行提交

public static void main(String[] args) throws IOException {  
 //创建HttpClient对象  
 CloseableHttpClient httpClient = HttpClients.*createDefault*();  
  
 //创建HttpGet请求  
 HttpPost httpPost = new HttpPost("http://www.itcast.cn/");  
  
  
 //声明存放参数的List集合  
 List<NameValuePair> params = new ArrayList<NameValuePair>();  
 params.add(new BasicNameValuePair("keys", "java"));  
  
 //创建表单数据Entity  
 UrlEncodedFormEntity formEntity = new UrlEncodedFormEntity(params, "UTF-8");  
  
 //设置表单Entity到httpPost请求对象中  
 httpPost.setEntity(formEntity);  
  
 CloseableHttpResponse response = null;  
 try {  
 //使用HttpClient发起请求  
 response = httpClient.execute(httpPost);  
  
 //判断响应状态码是否为200  
 if (response.getStatusLine().getStatusCode() == 200) {  
 //如果为200表示请求成功，获取返回数据  
 String content = EntityUtils.*toString*(response.getEntity(), "UTF-8");  
 //打印数据长度  
 System.*out*.println(content);  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 //释放连接  
 if (response == null) {  
 try {  
 response.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 httpClient.close();  
 }  
 }  
}

请求结果



## 连接池

如果每次请求都要创建HttpClient，会有频繁创建和销毁的问题，可以使用连接池来解决这个问题。

测试以下代码，并断点查看每次获取的HttpClient都是不一样的。

public static void main(String[] args) {  
 PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager();  
  
 // 设置最大连接数  
 cm.setMaxTotal(200);  
  
 // 设置每个主机的并发数  
 cm.setDefaultMaxPerRoute(20);  
  
 *doGet*(cm);  
  
 *doGet*(cm);  
  
}  
  
private static void doGet(PoolingHttpClientConnectionManager cm) {  
 CloseableHttpClient httpClient = HttpClients.*custom*().setConnectionManager(cm).build();  
  
 HttpGet httpGet = new HttpGet("http://www.itcast.cn/");  
  
 CloseableHttpResponse response = null;  
  
 try {  
 response = httpClient.execute(httpGet);  
  
 // 判断状态码是否是200  
 if (response.getStatusLine().getStatusCode() == 200) {  
 // 解析数据  
 String content = EntityUtils.*toString*(response.getEntity(), "UTF-8");  
 System.*out*.println(content.length());  
 }  
  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 //释放连接  
 if (response == null) {  
 try {  
 response.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 //不能关闭HttpClient  
 //httpClient.close();  
 }  
 }  
}

## 请求参数

有时候因为网络，或者目标服务器的原因，请求需要更长的时间才能完成，我们需要自定义相关时间

public static void main(String[] args) throws IOException {  
 //创建HttpClient对象  
 CloseableHttpClient httpClient = HttpClients.*createDefault*();  
  
  //创建HttpGet请求  
 HttpGet httpGet = new HttpGet("http://www.itcast.cn/");  
  
 //设置请求参数  
 RequestConfig requestConfig = RequestConfig.*custom*()  
 .setConnectTimeout(1000)//设置创建连接的最长时间  
 .setConnectionRequestTimeout(500)//设置获取连接的最长时间  
 .setSocketTimeout(10 \* 1000)//设置数据传输的最长时间  
 .build();  
  
 httpGet.setConfig(requestConfig);  
  
 CloseableHttpResponse response = null;  
 try {  
 //使用HttpClient发起请求  
 response = httpClient.execute(httpGet);  
  
 //判断响应状态码是否为200  
 if (response.getStatusLine().getStatusCode() == 200) {  
 //如果为200表示请求成功，获取返回数据  
 String content = EntityUtils.*toString*(response.getEntity(), "UTF-8");  
 //打印数据长度  
 System.*out*.println(content);  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 //释放连接  
 if (response == null) {  
 try {  
 response.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 httpClient.close();  
 }  
 }  
}

# Jsoup

我们抓取到页面之后，还需要对页面进行解析。可以使用字符串处理工具解析页面，也可以使用正则表达式，但是这些方法都会带来很大的开发成本，所以我们需要使用一款专门解析html页面的技术。

## jsoup介绍

jsoup 是一款Java 的HTML解析器，可直接解析某个URL地址、HTML文本内容。它提供了一套非常省力的API，可通过DOM，CSS以及类似于jQuery的操作方法来取出和操作数据。

jsoup的主要功能如下：

从一个URL，文件或字符串中解析HTML；

使用DOM或CSS选择器来查找、取出数据；

可操作HTML元素、属性、文本；

先加入Jsoup依赖：

<!--Jsoup-->  
<dependency>  
 <groupId>org.jsoup</groupId>  
 <artifactId>jsoup</artifactId>  
 <version>1.11.</version>  
</dependency>  
<!--测试-->  
<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
</dependency>

<!--工具-->  
<dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-lang3</artifactId>  
 <version>3.7</version>  
</dependency>  
<dependency>  
 <groupId>commons-io</groupId>  
 <artifactId>commons-io</artifactId>  
 <version>2.6</version>  
</dependency>

## jsoup解析

### 解析url

Jsoup可以直接输入url，它会发起请求并获取数据，封装为Document对象

@Test  
public void testJsoupUrl() throws Exception {  
 // 解析url地址  
 Document document = Jsoup.*parse*(new URL("http://www.itcast.cn/"), 1000);  
  
 //获取title的内容  
 Element title = document.getElementsByTag("title").first();  
 System.*out*.println(title.text());  
}

PS：虽然使用Jsoup可以替代HttpClient直接发起请求解析数据，但是往往不会这样用，因为实际的开发过程中，需要使用到多线程，连接池，代理等等方式，而jsoup对这些的支持并不是很好，所以我们一般把jsoup仅仅作为Html解析工具使用

### 解析字符串

先准备以下html文件

|  |
| --- |
| <html>  <head>  <title>传智播客官网-一样的教育,不一样的品质</title>  </head>  <body>  <div class="city">  <h3 id="city\_bj">北京中心</h3>  <fb:img src="/2018czgw/images/slogan.jpg" class="slogan"/>  <div class="city\_in">  <div class="city\_con" style="display: none;">  <ul>  <li id="test" class="class\_a class\_b">  <a href="http://www.itcast.cn" target="\_blank">  <span class="s\_name">北京</span>  </a>  </li>  <li>  <a href="http://sh.itcast.cn" target="\_blank">  <span class="s\_name">上海</span>  </a>  </li>  <li>  <a href="http://gz.itcast.cn" target="\_blank">  <span abc="123" class="s\_name">广州</span>  </a>  </li>  <ul>  <li>天津</li>  </ul>  </ul>  </div>  </div>  </div>  </body>  </html> |

Jsoup可以直接输入字符串，并封装为Document对象

@Test  
public void testJsoupString() throws Exception {  
 //读取文件获取  
 String html = FileUtils.*readFileToString*(new File("D:\\jsoup.html"), "UTF-8");  
  
 // 解析字符串  
 Document document = Jsoup.*parse*(html);  
  
 //获取title的内容  
 Element title = document.getElementsByTag("title").first();  
 System.*out*.println(title.text());  
  
}

### 解析文件

Jsoup可以直接解析文件，并封装为Document对象

@Test  
public void testJsoupHtml() throws Exception {  
 // 解析文件  
 Document document = Jsoup.*parse*(new File("D:\\jsoup.html"),"UTF-8");  
  
 //获取title的内容  
 Element title = document.getElementsByTag("title").first();  
 System.*out*.println(title.text());  
}

### 使用dom方式遍历文档

元素获取

1. 根据id查询元素getElementById
2. 根据标签获取元素getElementsByTag
3. 根据class获取元素getElementsByClass
4. 根据属性获取元素getElementsByAttribute

//1. 根据id查询元素getElementById  
Element element = document.getElementById("city\_bj");  
  
//2. 根据标签获取元素getElementsByTag  
element = document.getElementsByTag("title").first();  
  
//3. 根据class获取元素getElementsByClass  
element = document.getElementsByClass("s\_name").last();  
  
//4. 根据属性获取元素getElementsByAttribute  
element = document.getElementsByAttribute("abc").first();  
element = document.getElementsByAttributeValue("class", "city\_con").first();

元素中获取数据

1. 从元素中获取id
2. 从元素中获取className
3. 从元素中获取属性的值attr
4. 从元素中获取所有属性attributes
5. 从元素中获取文本内容text

//获取元素  
Element element = document.getElementById("test");  
  
//1. 从元素中获取id  
String str = element.id();  
  
//2. 从元素中获取className  
str = element.className();  
  
//3. 从元素中获取属性的值attr  
str = element.attr("id");  
  
//4. 从元素中获取所有属性attributes  
str = element.attributes().toString();  
  
//5. 从元素中获取文本内容text  
str = element.text();

### 使用选择器语法查找元素

jsoup elements对象支持类似于CSS (或jquery)的选择器语法，来实现非常强大和灵活的查找功能。这个select 方法在Document, Element,或Elements对象中都可以使用。且是上下文相关的，因此可实现指定元素的过滤，或者链式选择访问。

Select方法将返回一个Elements集合，并提供一组方法来抽取和处理结果。

### Selector选择器概述

**tagname**: 通过标签查找元素，比如：span

**#id**: 通过ID查找元素，比如：# city\_bj

**.class**: 通过class名称查找元素，比如：.class\_a

**[attribute]**: 利用属性查找元素，比如：[abc]

**[attr=value]**: 利用属性值来查找元素，比如：[class=s\_name]

//tagname: 通过标签查找元素，比如：span  
Elements span = document.select("span");  
for (Element element : span) {  
 System.*out*.println(element.text());  
}  
  
//#id: 通过ID查找元素，比如：#city\_bjj  
String str = document.select("#city\_bj").text();  
  
//.class: 通过class名称查找元素，比如：.class\_a  
str = document.select(".class\_a").text();  
  
//[attribute]: 利用属性查找元素，比如：[abc]  
str = document.select("[abc]").text();  
  
//[attr=value]: 利用属性值来查找元素，比如：[class=s\_name]  
str = document.select("[class=s\_name]").text();

### Selector选择器组合使用

**el#id**: 元素+ID，比如： h3#city\_bj

**el.class**: 元素+class，比如： li.class\_a

**el[attr]**: 元素+属性名，比如： span[abc]

**任意组合**: 比如：span[abc].s\_name

**ancestor child**: 查找某个元素下子元素，比如：.city\_con li 查找"city\_con"下的所有li

**parent > child**: 查找某个父元素下的直接子元素，比如：

.city\_con > ul > li 查找city\_con第一级（直接子元素）的ul，再找所有ul下的第一级li

**parent > \***: 查找某个父元素下所有直接子元素

//el#id: 元素+ID，比如： h3#city\_bj  
String str = document.select("h3#city\_bj").text();  
  
//el.class: 元素+class，比如： li.class\_a  
str = document.select("li.class\_a").text();  
  
//el[attr]: 元素+属性名，比如： span[abc]  
str = document.select("span[abc]").text();  
  
//任意组合，比如：span[abc].s\_name  
str = document.select("span[abc].s\_name").text();  
  
//ancestor child: 查找某个元素下子元素，比如：.city\_con li 查找"city\_con"下的所有li  
str = document.select(".city\_con li").text();  
  
//parent > child: 查找某个父元素下的直接子元素，  
//比如：.city\_con > ul > li 查找city\_con第一级（直接子元素）的ul，再找所有ul下的第一级li  
str = document.select(".city\_con > ul > li").text();  
  
//parent > \* 查找某个父元素下所有直接子元素.city\_con > \*  
str = document.select(".city\_con > \*").text();

# 爬虫案例

学习了HttpClient和Jsoup，就掌握了如何抓取数据和如何解析数据，接下来，我们做一个小练习，把京东的手机数据抓取下来。

主要目的是HttpClient和Jsoup的学习。

## 需求分析

首先访问京东，搜索手机，分析页面，我们抓取以下商品数据：

商品图片、价格、标题、商品详情页



### SPU和SKU

除了以上四个属性以外，我们发现上图中的苹果手机有四种产品，我们应该每一种都要抓取。那么这里就必须要了解spu和sku的概念

SPU = Standard Product Unit （标准产品单位）

SPU是商品信息聚合的最小单位，是一组可复用、易检索的标准化信息的集合，该集合描述了一个产品的特性。通俗点讲，属性值、特性相同的商品就可以称为一个SPU。

例如上图中的苹果手机就是SPU，包括红色、深灰色、金色、银色

SKU=stock keeping unit(库存量单位)

SKU即库存进出计量的单位， 可以是以件、盒、托盘等为单位。SKU是物理上不可分割的最小存货单元。在使用时要根据不同业态，不同管理模式来处理。在服装、鞋类商品中使用最多最普遍。

例如上图中的苹果手机有几个款式，红色苹果手机，就是一个sku

查看页面的源码也可以看出区别



## 开发准备

### 数据库表分析

根据需求分析，我们创建的表如下：

CREATE TABLE `jd\_item` (

`id` bigint(10) NOT NULL AUTO\_INCREMENT COMMENT '主键id',

`spu` bigint(15) DEFAULT NULL COMMENT '商品集合id',

`sku` bigint(15) DEFAULT NULL COMMENT '商品最小品类单元id',

`title` varchar(100) DEFAULT NULL COMMENT '商品标题',

`price` bigint(10) DEFAULT NULL COMMENT '商品价格',

`pic` varchar(200) DEFAULT NULL COMMENT '商品图片',

`url` varchar(200) DEFAULT NULL COMMENT '商品详情地址',

`created` datetime DEFAULT NULL COMMENT '创建时间',

`updated` datetime DEFAULT NULL COMMENT '更新时间',

PRIMARY KEY (`id`),

KEY `sku` (`sku`) USING BTREE

) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='京东商品表';

### 添加依赖

使用Spring Boot+Spring Data JPA和定时任务进行开发，

需要创建Maven工程并添加以下依赖

<?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>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.2.RELEASE</version>  
 </parent>  
 <groupId>cn.itcast.crawler</groupId>  
 <artifactId>itcast-crawler-jd</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <dependencies>  
 <!--SpringMVC-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <!--SpringData Jpa-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 </dependency>  
  
 <!--MySQL连接包-->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
  
 <!-- HttpClient -->  
 <dependency>  
 <groupId>org.apache.httpcomponents</groupId>  
 <artifactId>httpclient</artifactId>  
 </dependency>  
  
 <!--Jsoup-->  
 <dependency>  
 <groupId>org.jsoup</groupId>  
 <artifactId>jsoup</artifactId>  
 <version>1.10.3</version>  
 </dependency>  
  
 <!--工具包-->  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-lang3</artifactId>  
 </dependency>  
 </dependencies>  
</project>

### 添加配置文件

加入application.properties配置文件

#DB Configuration:  
spring.datasource.driverClassName=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/crawler  
spring.datasource.username=root  
spring.datasource.password=root  
  
#JPA Configuration:  
spring.jpa.database=*MySQL*spring.jpa.show-sql=**true**

## 代码实现

### 编写pojo

根据数据库表，编写pojo

@Entity  
@Table(name = "jd\_item")  
public class Item {  
 //主键  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 //标准产品单位（商品集合）  
 private Long spu;  
 //库存量单位（最小品类单元）  
 private Long sku;  
 //商品标题  
 private String title;  
 //商品价格  
 private Double price;  
 //商品图片  
 private String pic;  
 //商品详情地址  
 private String url;  
 //创建时间  
 private Date created;  
 //更新时间  
 private Date updated;

set/get  
}

### 编写dao

public interface ItemDao extends JpaRepository<Item,Long> {  
   
}

### 编写Service

ItemService接口

public interface ItemService {  
  
 //根据条件查询数据  
 public List<Item> findAll(Item item);  
  
 //保存数据  
 public void save(Item item);  
}

ItemServiceImpl实现类

@Service  
public class ItemServiceImpl implements ItemService {  
  
 @Autowired  
 private ItemDao itemDao;  
  
 @Override  
 public List<Item> findAll(Item item) {  
 Example example = Example.*of*(item);  
 List list = this.itemDao.findAll(example);  
 return list;  
 }  
  
 @Override  
 @Transactional  
 public void save(Item item) {  
 this.itemDao.save(item);  
 }  
}

### 编写引导类

@SpringBootApplication  
//设置开启定时任务  
@EnableScheduling  
public class Application {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(Application.class, args);  
 }  
}

### 封装HttpClient

我们需要经常使用HttpClient，所以需要进行封装，方便使用

@Component  
public class HttpUtils {  
  
 private PoolingHttpClientConnectionManager cm;  
  
 public HttpUtils() {  
 this.cm = new PoolingHttpClientConnectionManager();  
  
 // 设置最大连接数  
 cm.setMaxTotal(200);  
  
 // 设置每个主机的并发数  
 cm.setDefaultMaxPerRoute(20);  
 }  
  
 //获取内容  
 public String getHtml(String url) {  
 // 获取HttpClient对象  
 CloseableHttpClient httpClient = HttpClients.*custom*().setConnectionManager(cm).build();  
  
 // 声明httpGet请求对象  
 HttpGet httpGet = new HttpGet(url);  
 // 设置请求参数RequestConfig  
 httpGet.setConfig(this.getConfig());  
  
 CloseableHttpResponse response = null;  
 try {  
 // 使用HttpClient发起请求，返回response  
 response = httpClient.execute(httpGet);  
 // 解析response返回数据  
 if (response.getStatusLine().getStatusCode() == 200) {  
 String html = "";  
  
 // 如果response。getEntity获取的结果是空，在执行EntityUtils.toString会报错  
 // 需要对Entity进行非空的判断  
 if (response.getEntity() != null) {  
 html = EntityUtils.*toString*(response.getEntity(), "UTF-8");  
 }  
  
 return html;  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (response != null) {  
 // 关闭连接  
 response.close();  
 }  
 // 不能关闭，现在使用的是连接管理器  
 // httpClient.close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 return null;  
 }  
  
 //获取图片  
 public String getImage(String url) {  
 // 获取HttpClient对象  
 CloseableHttpClient httpClient = HttpClients.*custom*().setConnectionManager(cm).build();  
  
 // 声明httpGet请求对象  
 HttpGet httpGet = new HttpGet(url);  
 // 设置请求参数RequestConfig  
 httpGet.setConfig(this.getConfig());  
  
 CloseableHttpResponse response = null;  
 try {  
 // 使用HttpClient发起请求，返回response  
 response = httpClient.execute(httpGet);  
 // 解析response下载图片  
 if (response.getStatusLine().getStatusCode() == 200) {  
 // 获取文件类型  
 String extName = url.substring(url.lastIndexOf("."));  
 // 使用uuid生成图片名  
 String imageName = UUID.*randomUUID*().toString() + extName;  
  
 // 声明输出的文件  
 OutputStream outstream = new FileOutputStream(new File("D:/images/" + imageName));  
 // 使用响应体输出文件  
 response.getEntity().writeTo(outstream);  
  
 // 返回生成的图片名  
 return imageName;  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (response != null) {  
 // 关闭连接  
 response.close();  
 }  
 // 不能关闭，现在使用的是连接管理器  
 // httpClient.close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 return null;  
 }  
  
 //获取请求参数对象  
 private RequestConfig getConfig() {  
 RequestConfig config = RequestConfig.*custom*().setConnectTimeout(1000)// 设置创建连接的超时时间  
 .setConnectionRequestTimeout(500) // 设置获取连接的超时时间  
 .setSocketTimeout(10000) // 设置连接的超时时间  
 .build();  
  
 return config;  
 }  
  
}

### 实现数据抓取

使用定时任务，可以定时抓取最新的数据

@Component  
public class ItemTask {  
  
 @Autowired  
 private HttpUtils httpUtils;  
 @Autowired  
 private ItemService itemService;  
  
 public static final ObjectMapper *MAPPER* = new ObjectMapper();  
  
  
 //设置定时任务执行完成后，再间隔100秒执行一次  
 @Scheduled(fixedDelay = 1000 \* 100)  
 public void process() throws Exception {  
 //分析页面发现访问的地址,页码page从1开始，下一页oage加2  
 String url = "https://search.jd.com/Search?keyword=%E6%89%8B%E6%9C%BA&enc=utf-8&qrst=1&rt=1&stop=1&vt=2&cid2=653&cid3=655&s=5760&click=0&page=";  
  
 //遍历执行，获取所有的数据  
 for (int i = 1; i < 10; i = i + 2) {  
 //发起请求进行访问，获取页面数据,先访问第一页  
 String html = this.httpUtils.getHtml(url + i);  
  
 //解析页面数据，保存数据到数据库中  
 this.parseHtml(html);  
  
 }  
 System.*out*.println("执行完成");  
 }  
  
  
 //解析页面，并把数据保存到数据库中  
 private void parseHtml(String html) throws Exception {  
 //使用jsoup解析页面  
 Document document = Jsoup.*parse*(html);  
  
 //获取商品数据  
 Elements spus = document.select("div#J\_goodsList > ul > li");  
  
 //遍历商品spu数据  
 for (Element spuEle : spus) {  
 //获取商品spu  
 Long spuId = Long.*parseLong*(spuEle.attr("data-spu"));  
  
 //获取商品sku数据  
 Elements skus = spuEle.select("li.ps-item img");  
 for (Element skuEle : skus) {  
 //获取商品sku  
 Long skuId = Long.*parseLong*(skuEle.attr("data-sku"));  
  
 //判断商品是否被抓取过，可以根据sku判断  
 Item param = new Item();  
 param.setSku(skuId);  
 List<Item> list = this.itemService.findAll(param);  
 //判断是否查询到结果  
 if (list.size() > 0) {  
 //如果有结果，表示商品已下载，进行下一次遍历  
 continue;  
 }  
  
 //保存商品数据，声明商品对象  
 Item item = new Item();  
  
 //商品spu  
 item.setSpu(spuId);  
 //商品sku  
 item.setSku(skuId);  
 //商品url地址  
 item.setUrl("https://item.jd.com/" + skuId + ".html");  
 //创建时间  
 item.setCreated(new Date());  
 //修改时间  
 item.setUpdated(item.getCreated());  
  
  
 //获取商品标题  
 String itemHtml = this.httpUtils.getHtml(item.getUrl());  
 String title = Jsoup.*parse*(itemHtml).select("div.sku-name").text();  
 item.setTitle(title);  
  
 //获取商品价格  
 String priceUrl = "https://p.3.cn/prices/mgets?skuIds=J\_"+skuId;  
 String priceJson = this.httpUtils.getHtml(priceUrl);  
 //解析json数据获取商品价格  
 double price = *MAPPER*.readTree(priceJson).get(0).get("p").asDouble();  
 item.setPrice(price);  
  
 //获取图片地址  
 String pic = "https:" + skuEle.attr("data-lazy-img").replace("/n9/","/n1/");  
 System.*out*.println(pic);  
 //下载图片  
 String picName = this.httpUtils.getImage(pic);  
 item.setPic(picName);  
  
 //保存商品数据  
 this.itemService.save(item);  
 }  
 }  
 }  
}

# WebMagic介绍

昨天完成了爬虫的入门的学习，是一个最基本的爬虫案例，今天我们要学习一款爬虫框架的使用就是WebMagic。其底层用到了我们上一天课程所使用的HttpClient和Jsoup，让我们能够更方便的开发爬虫。

WebMagic项目代码分为核心和扩展两部分。核心部分(webmagic-core)是一个精简的、模块化的爬虫实现，而扩展部分则包括一些便利的、实用性的功能。

WebMagic的设计目标是尽量的模块化，并体现爬虫的功能特点。这部分提供非常简单、灵活的API，在基本不改变开发模式的情况下，编写一个爬虫。

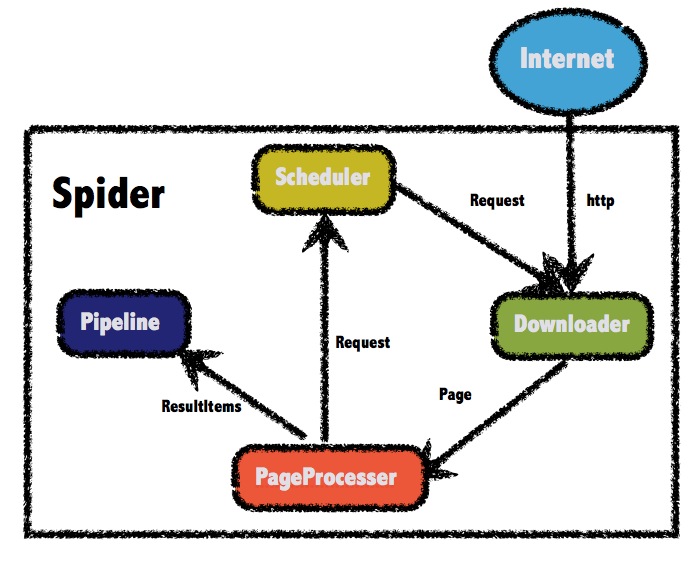
扩展部分(webmagic-extension)提供一些便捷的功能，例如注解模式编写爬虫等。同时内置了一些常用的组件，便于爬虫开发。

## 架构介绍

WebMagic的结构分为Downloader、PageProcessor、Scheduler、Pipeline四大组件，并由Spider将它们彼此组织起来。这四大组件对应爬虫生命周期中的下载、处理、管理和持久化等功能。WebMagic的设计参考了Scapy，但是实现方式更Java化一些。

而Spider则将这几个组件组织起来，让它们可以互相交互，流程化的执行，可以认为Spider是一个大的容器，它也是WebMagic逻辑的核心。

WebMagic总体架构图如下：



### WebMagic的四个组件

1.Downloader

Downloader负责从互联网上下载页面，以便后续处理。WebMagic默认使用了Apache HttpClient作为下载工具。

2.PageProcessor

PageProcessor负责解析页面，抽取有用信息，以及发现新的链接。WebMagic使用Jsoup作为HTML解析工具，并基于其开发了解析XPath的工具Xsoup。

在这四个组件中，PageProcessor对于每个站点每个页面都不一样，是需要使用者定制的部分。

3.Scheduler

Scheduler负责管理待抓取的URL，以及一些去重的工作。WebMagic默认提供了JDK的内存队列来管理URL，并用集合来进行去重。也支持使用Redis进行分布式管理。

4.Pipeline

Pipeline负责抽取结果的处理，包括计算、持久化到文件、数据库等。WebMagic默认提供了“输出到控制台”和“保存到文件”两种结果处理方案。

Pipeline定义了结果保存的方式，如果你要保存到指定数据库，则需要编写对应的Pipeline。对于一类需求一般只需编写一个Pipeline。

### 用于数据流转的对象

1. Request

Request是对URL地址的一层封装，一个Request对应一个URL地址。

它是PageProcessor与Downloader交互的载体，也是PageProcessor控制Downloader唯一方式。

除了URL本身外，它还包含一个Key-Value结构的字段extra。你可以在extra中保存一些特殊的属性，然后在其他地方读取，以完成不同的功能。例如附加上一个页面的一些信息等。

2. Page

Page代表了从Downloader下载到的一个页面——可能是HTML，也可能是JSON或者其他文本格式的内容。

Page是WebMagic抽取过程的核心对象，它提供一些方法可供抽取、结果保存等。

3. ResultItems

ResultItems相当于一个Map，它保存PageProcessor处理的结果，供Pipeline使用。它的API与Map很类似，值得注意的是它有一个字段skip，若设置为true，则不应被Pipeline处理。

## 入门案例

### 加入依赖

创建Maven工程，并加入以下依赖

<?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.crawler</groupId>  
 <artifactId>itcast-crawler-webmagic</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <dependencies>  
 <!--WebMagic-->  
 <dependency>  
 <groupId>us.codecraft</groupId>  
 <artifactId>webmagic-core</artifactId>  
 <version>0.7.3</version>  
 </dependency>  
 <dependency>  
 <groupId>us.codecraft</groupId>  
 <artifactId>webmagic-extension</artifactId>  
 <version>0.7.3</version>  
 </dependency>  
 </dependencies>  
   
</project>

注意：0.7.3版本对SSL的并不完全，如果是直接从Maven中央仓库下载依赖，在爬取只支持SSL v1.2的网站会有SSL的异常抛出。

解决方案：

* + - 1. 等作者的0.7.4的版本发布
      2. 直接从github上下载最新的代码，安装到本地仓库

也可以参考以下资料自己修复

https://github.com/code4craft/webmagic/issues/701

### 加入配置文件

WebMagic使用slf4j-log4j12作为slf4j的实现。

添加log4j.properties配置文件

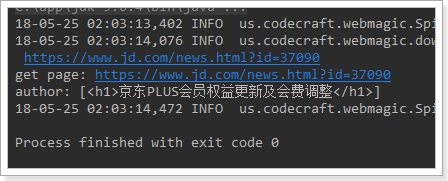
log4j.rootLogger=INFO,A1   
  
log4j.appender.A1=org.apache.log4j.ConsoleAppender  
log4j.appender.A1.layout=org.apache.log4j.PatternLayout  
log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n

### 案例实现

public class JobProcessor implements PageProcessor {  
  
 public void process(Page page) {  
 page.putField("author", page.getHtml().css("div.mt>h1").all());  
 }

private Site site = Site.*me*();  
 public Site getSite() {  
 return site;  
 }  
  
 public static void main(String[] args) {  
 Spider.*create*(new JobProcessor())  
 //初始访问url地址  
 .addUrl("https://www.jd.com/moreSubject.aspx")   
 .run();  
 }  
}

打印结果：



# WebMagic功能

## 实现PageProcessor

### 抽取元素Selectable

WebMagic里主要使用了三种抽取技术：XPath、正则表达式和CSS选择器。另外，对于JSON格式的内容，可使用JsonPath进行解析。

XPath

以上是获取属性class=mt的div标签，里面的h1标签的内容

page.getHtml().xpath("//div[@class=mt]/h1/text()")

也可以参考课堂资料的W3School离线手册(2017.03.11版).chm

CSS选择器

CSS选择器是与XPath类似的语言。在上一次的课程中，我们已经学习过了Jsoup的选择器，它比XPath写起来要简单一些，但是如果写复杂一点的抽取规则，就相对要麻烦一点。

div.mt>h1表示class为mt的div标签下的直接子元素h1标签

page.getHtml().css("div.mt>h1").toString()

可是使用:nth-child(n)选择第几个元素，如下选择第一个元素

page.getHtml().css("div#news\_div > ul > li:nth-child(1) a").toString()

注意：需要使用>，就是直接子元素才可以选择第几个元素

正则表达式

正则表达式则是一种通用的文本抽取语言。在这里一般用于获取url地址。

正则表达式学习难度要大一些，大家可以参考课堂资料《正则表达式系统教程.CHM》

### 抽取元素API

Selectable相关的抽取元素链式API是WebMagic的一个核心功能。使用Selectable接口，可以直接完成页面元素的链式抽取，也无需去关心抽取的细节。

在刚才的例子中可以看到，page.getHtml()返回的是一个Html对象，它实现了Selectable接口。这个接口包含的方法分为两类：抽取部分和获取结果部分。

| **方法** | **说明** | **示例** |
| --- | --- | --- |
| xpath(String xpath) | 使用XPath选择 | html.xpath("//div[@class='title']") |
| $(String selector) | 使用Css选择器选择 | html.$("div.title") |
| $(String selector,String attr) | 使用Css选择器选择 | html.$("div.title","text") |
| css(String selector) | 功能同$()，使用Css选择器选择 | html.css("div.title") |
| links() | 选择所有链接 | html.links() |
| regex(String regex) | 使用正则表达式抽取 | html.regex("\(.\\*?)\") |

这部分抽取API返回的都是一个Selectable接口，意思是说，是支持链式调用的。例如访问<https://www.jd.com/moreSubject.aspx>页面



//先获取class为news\_div的div

//再获取里面的所有包含文明的元素  
List<String> list = page.getHtml()  
 .css("div#news\_div")  
 .regex(".\*文明.\*").all();

### 获取结果API

当链式调用结束时，我们一般都想要拿到一个字符串类型的结果。这时候就需要用到获取结果的API了。

我们知道，一条抽取规则，无论是XPath、CSS选择器或者正则表达式，总有可能抽取到多条元素。WebMagic对这些进行了统一，可以通过不同的API获取到一个或者多个元素。

| **方法** | **说明** | **示例** |
| --- | --- | --- |
| get() | 返回一条String类型的结果 | String link= html.links().get() |
| toString() | 同get()，返回一条String类型的结果 | String link= html.links().toString() |
| all() | 返回所有抽取结果 | List links= html.links().all() |

当有多条数据的时候，使用get()和toString()都是获取第一个url地址。

String str = page.getHtml()  
 .css("div#news\_div")  
 .links().regex(".\*[0-3]$").toString();  
  
String get = page.getHtml()  
 .css("div#news\_div")  
 .links().regex(".\*[0-3]$").get();

测试结果：



这里selectable.toString()采用了toString()这个接口，是为了在输出以及和一些框架结合的时候，更加方便。因为一般情况下，我们都只需要选择一个元素！

selectable.all()则会获取到所有元素。

### 获取链接

有了处理页面的逻辑，我们的爬虫就接近完工了，但是现在还有一个问题：一个站点的页面是很多的，一开始我们不可能全部列举出来，于是如何发现后续的链接，是一个爬虫不可缺少的一部分。

下面的例子就是获取<https://www.jd.com/moreSubject.aspx>这个页面中

所有符合[https://www.jd.com/news.\\w+?.\*](https://www.jd.com/news.\\w+?.*)正则表达式的url地址

并将这些链接加入到待抓取的队列中去。

public void process(Page page) {  
 page.addTargetRequests(page.getHtml().links()  
 .regex("(https://www.jd.com/news.\\w+?.\*)").all());  
 System.*out*.println(page.getHtml().css("div.mt>h1").all());  
}  
  
public static void main(String[] args) {  
 Spider.*create*(new JobProcessor())  
 .addUrl("https://www.jd.com/moreSubject.aspx")  
 .run();  
}

## 使用Pipeline保存结果

WebMagic用于保存结果的组件叫做Pipeline。我们现在通过“控制台输出结果”这件事也是通过一个内置的Pipeline完成的，它叫做ConsolePipeline。

那么，我现在想要把结果用保存到文件中，怎么做呢？只将Pipeline的实现换成"FilePipeline"就可以了。

public static void main(String[] args) {  
 Spider.*create*(new JobProcessor())  
 //初始访问url地址  
 .addUrl("https://www.jd.com/moreSubject.aspx")  
 .addPipeline(new FilePipeline("D:/webmagic/"))  
 .thread(5)//设置线程数  
 .run();  
}

## 爬虫的配置、启动和终止

### Spider

Spider是爬虫启动的入口。在启动爬虫之前，我们需要使用一个PageProcessor创建一个Spider对象，然后使用run()进行启动。

同时Spider的其他组件（Downloader、Scheduler、Pipeline）都可以通过set方法来进行设置。

| **方法** | **说明** | **示例** |
| --- | --- | --- |
| create(PageProcessor) | 创建Spider | Spider.create(new GithubRepoProcessor()) |
| addUrl(String…) | 添加初始的URL | spider .addUrl("<http://webmagic.io/docs/>") |
| thread(n) | 开启n个线程 | spider.thread(5) |
| run() | 启动，会阻塞当前线程执行 | spider.run() |
| start()/runAsync() | 异步启动，当前线程继续执行 | spider.start() |
| stop() | 停止爬虫 | spider.stop() |
| addPipeline(Pipeline) | 添加一个Pipeline，一个Spider可以有多个Pipeline | spider .addPipeline(new ConsolePipeline()) |
| setScheduler(Scheduler) | 设置Scheduler，一个Spider只能有个一个Scheduler | spider.setScheduler(new RedisScheduler()) |
| setDownloader(Downloader) | 设置Downloader，一个Spider只能有个一个Downloader | spider .setDownloader(  new SeleniumDownloader()) |
| get(String) | 同步调用，并直接取得结果 | ResultItems result = spider  .get("<http://webmagic.io/docs/>") |
| getAll(String…) | 同步调用，并直接取得一堆结果 | List<ResultItems> results = spider  .getAll("<http://webmagic.io/docs/>", "<http://webmagic.io/xxx>") |

### 爬虫配置Site

Site.me()可以对爬虫进行一些配置配置，包括编码、抓取间隔、超时时间、重试次数等。在这里我们先简单设置一下：重试次数为3次，抓取间隔为一秒。

private Site site = Site.*me*()  
 .setCharset("UTF-8")//编码  
 .setSleepTime(1)//抓取间隔时间  
 .setTimeOut(1000\*10)//超时时间  
 .setRetrySleepTime(3000)//重试时间  
 .setRetryTimes(3);//重试次数

站点本身的一些配置信息，例如编码、HTTP头、超时时间、重试策略等、代理等，都可以通过设置Site对象来进行配置。

| **方法** | **说明** | **示例** |
| --- | --- | --- |
| setCharset(String) | 设置编码 | site.setCharset("utf-8") |
| setUserAgent(String) | 设置UserAgent | site.setUserAgent("Spider") |
| setTimeOut(int) | 设置超时时间，  单位是毫秒 | site.setTimeOut(3000) |
| setRetryTimes(int) | 设置重试次数 | site.setRetryTimes(3) |
| setCycleRetryTimes(int) | 设置循环重试次数 | site.setCycleRetryTimes(3) |
| addCookie(String,String) | 添加一条cookie | site.addCookie("dotcomt\_user","code4craft") |
| setDomain(String) | 设置域名，需设置域名后，addCookie才可生效 | site.setDomain("github.com") |
| addHeader(String,String) | 添加一条addHeader | site.addHeader("Referer","[https://github.com](https://github.com/)") |
| setHttpProxy(HttpHost) | 设置Http代理 | site.setHttpProxy(new HttpHost("127.0.0.1",8080)) |

# 爬虫分类

网络爬虫按照系统结构和实现技术，大致可以分为以下几种类型：通用网络爬虫、聚焦网络爬虫、增量式网络爬虫、深层网络爬虫。 实际的网络爬虫系统通常是几种爬虫技术相结合实现的

## 通用网络爬虫

通用网络爬虫又称全网爬虫（Scalable Web Crawler），爬行对象从一些种子 URL 扩充到整个 Web，主要为门户站点搜索引擎和大型 Web 服务提供商采集数据。

这类网络爬虫的爬行范围和数量巨大，对于爬行速度和存储空间要求较高，对于爬行页面的顺序要求相对较低，同时由于待刷新的页面太多，通常采用并行工作方式，但需要较长时间才能刷新一次页面。

简单的说就是互联网上抓取所有数据。

## 聚焦网络爬虫

聚焦网络爬虫（Focused Crawler），又称主题网络爬虫（Topical Crawler），是指选择性地爬行那些与预先定义好的主题相关页面的网络爬虫。

和通用网络爬虫相比，聚焦爬虫只需要爬行与主题相关的页面，极大地节省了硬件和网络资源，保存的页面也由于数量少而更新快，还可以很好地满足一些特定人群对特定领域信息的需求 。

简单的说就是互联网上只抓取某一种数据。

## 增量式网络爬虫

增量式网络爬虫（Incremental Web Crawler）是 指 对 已 下 载 网 页 采 取 增量式更新和只爬行新产生的或者已经发生变化网页的爬虫，它能够在一定程度上保证所爬行的页面是尽可能新的页面。

和周期性爬行和刷新页面的网络爬虫相比，增量式爬虫只会在需要的时候爬行新产生或发生更新的页面 ，并不重新下载没有发生变化的页面，可有效减少数据下载量，及时更新已爬行的网页，减小时间和空间上的耗费，但是增加了爬行算法的复杂度和实现难度。

简单的说就是互联网上只抓取刚刚更新的数据。

## Deep Web 爬虫

Web 页面按存在方式可以分为表层网页（Surface Web）和深层网页（Deep Web，也称 Invisible Web Pages 或 Hidden Web）。

表层网页是指传统搜索引擎可以索引的页面，以超链接可以到达的静态网页为主构成的 Web 页面。

Deep Web 是那些大部分内容不能通过静态链接获取的、隐藏在搜索表单后的，只有用户提交一些关键词才能获得的 Web 页面。

# 案例开发分析

我们已经学完了WebMagic的基本使用方法，现在准备使用WebMagic实现爬取数据的功能。这里是一个比较完整的实现。

在这里我们实现的是聚焦网络爬虫，只爬取招聘的相关数据。

## 业务分析

今天要实现的是爬取<https://www.51job.com/>上的招聘信息。只爬取“计算机软件”和“互联网电子商务”两个行业的信息。

首先访问页面并搜索两个行业。结果如下



点击职位详情页，我们分析发现详情页还有一些数据需要抓取：

职位、公司名称、工作地点、薪资、发布时间、职位信息、公司联系方式、公司信息





## 数据库表

根据以上信息，设计数据库表

CREATE TABLE `job\_info` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT COMMENT '主键id',

`company\_name` varchar(100) DEFAULT NULL COMMENT '公司名称',

`company\_addr` varchar(200) DEFAULT NULL COMMENT '公司联系方式',

`company\_info` text COMMENT '公司信息',

`job\_name` varchar(100) DEFAULT NULL COMMENT '职位名称',

`job\_addr` varchar(50) DEFAULT NULL COMMENT '工作地点',

`job\_info` text COMMENT '职位信息',

`salary\_min` int(10) DEFAULT NULL COMMENT '薪资范围，最小',

`salary\_max` int(10) DEFAULT NULL COMMENT '薪资范围，最大',

`url` varchar(150) DEFAULT NULL COMMENT '招聘信息详情页',

`time` varchar(10) DEFAULT NULL COMMENT '职位最近发布时间',

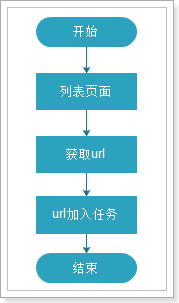
PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='招聘信息';

## 实现流程

我们需要解析职位列表页，获取职位的详情页，再解析页面获取数据。

获取url地址的流程如下



但是在这里有个问题：在解析页面的时候，很可能会解析出相同的url地址(例如商品标题和商品图片超链接，而且url一样)，如果不进行处理，同样的url会解析处理多次，浪费资源。所以我们需要有一个url去重的功能

### Scheduler组件

WebMagic提供了Scheduler可以帮助我们解决以上问题。

Scheduler是WebMagic中进行URL管理的组件。一般来说，Scheduler包括两个作用：

* 对待抓取的URL队列进行管理。
* 对已抓取的URL进行去重。

WebMagic内置了几个常用的Scheduler。如果你只是在本地执行规模比较小的爬虫，那么基本无需定制Scheduler，但是了解一下已经提供的几个Scheduler还是有意义的。

| **类** | **说明** | **备注** |
| --- | --- | --- |
| DuplicateRemovedScheduler | 抽象基类，提供一些模板方法 | 继承它可以实现自己的功能 |
| QueueScheduler | 使用内存队列保存待抓取URL |  |
| PriorityScheduler | 使用带有优先级的内存队列保存待抓取URL | 耗费内存较QueueScheduler更大，但是当设置了request.priority之后，只能使用PriorityScheduler才可使优先级生效 |
| FileCacheQueueScheduler | 使用文件保存抓取URL，可以在关闭程序并下次启动时，从之前抓取到的URL继续抓取 | 需指定路径，会建立.urls.txt和.cursor.txt两个文件 |
| RedisScheduler | 使用Redis保存抓取队列，可进行多台机器同时合作抓取 | 需要安装并启动redis |

去重部分被单独抽象成了一个接口：DuplicateRemover，从而可以为同一个Scheduler选择不同的去重方式，以适应不同的需要，目前提供了两种去重方式。

| **类** | **说明** |
| --- | --- |
| HashSetDuplicateRemover | 使用HashSet来进行去重，占用内存较大 |
| BloomFilterDuplicateRemover | 使用BloomFilter来进行去重，占用内存较小，但是可能漏抓页面 |

RedisScheduler是使用Redis的set进行去重，其他的Scheduler默认都使用HashSetDuplicateRemover来进行去重。

如果要使用BloomFilter，必须要加入以下依赖：

<!--WebMagic对布隆过滤器的支持-->  
<dependency>  
 <groupId>com.google.guava</groupId>  
 <artifactId>guava</artifactId>  
 <version>16.0</version>  
</dependency>

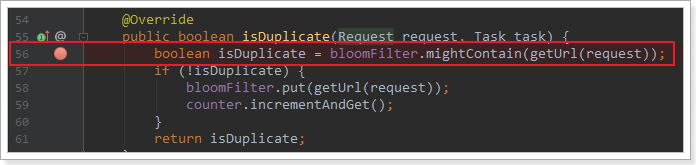
修改代码，添加布隆过滤器

public static void main(String[] args) {  
 Spider.*create*(new JobProcessor())  
 //初始访问url地址  
 .addUrl("https://www.jd.com/moreSubject.aspx")  
 .addPipeline(new FilePipeline("D:/webmagic/"))  
 .setScheduler(new QueueScheduler()  
 .setDuplicateRemover(new BloomFilterDuplicateRemover(10000000))) //参数设置需要对多少条数据去重  
 .thread(1)//设置线程数  
 .run();  
}

修改public void process(Page page)方法，添加一下代码

//每次加入相同的url，测试去重  
page.addTargetRequest("https://www.jd.com/news.html?id=36480");

打开布隆过滤器BloomFilterDuplicateRemover，在下图处打断点测试



### 三种去重方式

去重就有三种实现方式，那有什么不同呢？

* **HashSet**

使用java中的HashSet不能重复的特点去重。优点是容易理解。使用方便。

缺点：占用内存大，性能较低。

* **Redis去重**

使用Redis的set进行去重。优点是速度快（Redis本身速度就很快），而且去重不会占用爬虫服务器的资源，可以处理更大数据量的数据爬取。

缺点：需要准备Redis服务器，增加开发和使用成本。

* **布隆过滤器（BloomFilter）**

使用布隆过滤器也可以实现去重。优点是占用的内存要比使用HashSet要小的多，也适合大量数据的去重操作。

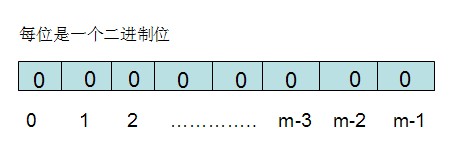
缺点：有误判的可能。没有重复可能会判定重复，但是重复数据一定会判定重复。

布隆过滤器 (Bloom Filter)是由Burton Howard Bloom于1970年提出，它是一种space efficient的概率型数据结构，用于判断一个元素是否在集合中。在垃圾邮件过滤的黑白名单方法、爬虫(Crawler)的网址判重模块中等等经常被用到。

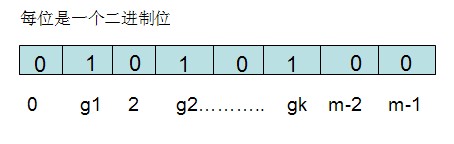
哈希表也能用于判断元素是否在集合中，但是布隆过滤器只需要哈希表的1/8或1/4的空间复杂度就能完成同样的问题。布隆过滤器可以插入元素，但不可以删除已有元素。其中的元素越多，误报率越大，但是漏报是不可能的。

**原理：**

布隆过滤器需要的是一个位数组(和位图类似)和K个映射函数(和Hash表类似)，在初始状态时，对于长度为m的位数组array，它的所有位被置0。



对于有n个元素的集合S={S1,S2...Sn},通过k个映射函数{f1,f2,......fk}，将集合S中的每个元素Sj(1<=j<=n)映射为K个值{g1,g2...gk}，然后再将位数组array中相对应的array[g1],array[g2]......array[gk]置为1：



如果要查找某个元素item是否在S中，则通过映射函数{f1,f2,...fk}得到k个值{g1,g2...gk}，然后再判断array[g1],array[g2]...array[gk]是否都为1，若全为1，则item在S中，否则item不在S中。

布隆过滤器会造成一定的误判，因为集合中的若干个元素通过映射之后得到的数值恰巧包括g1,g2,...gk，在这种情况下可能会造成误判，但是概率很小。

### 布隆过滤器实现（了解）

以下是一个布隆过滤器的实现，可以参考

//布隆过滤器  
public class BloomFilter {  
  
 /\* BitSet初始分配2^24个bit \*/  
 private static final int *DEFAULT\_SIZE* = 1 << 24;  
  
 /\* 不同哈希函数的种子，一般应取质数 \*/  
 private static final int[] *seeds* = new int[] { 5, 7, 11, 13, 31, 37 };  
  
 private BitSet bits = new BitSet(*DEFAULT\_SIZE*);  
  
 /\* 哈希函数对象 \*/  
 private SimpleHash[] func = new SimpleHash[*seeds*.length];  
  
 public BloomFilter() {  
 for (int i = 0; i < *seeds*.length; i++) {  
 func[i] = new SimpleHash(*DEFAULT\_SIZE*, *seeds*[i]);  
 }  
 }  
  
 // 将url标记到bits中  
 public void add(String str) {  
 for (SimpleHash f : func) {  
 bits.set(f.hash(str), true);  
 }  
 }  
  
 // 判断是否已经被bits标记  
 public boolean contains(String str) {  
 if (StringUtils.*isBlank*(str)) {  
 return false;  
 }  
  
 boolean ret = true;  
 for (SimpleHash f : func) {  
 ret = ret && bits.get(f.hash(str));  
 }  
  
 return ret;  
 }  
  
 /\* 哈希函数类 \*/  
 public static class SimpleHash {  
 private int cap;  
 private int seed;  
  
 public SimpleHash(int cap, int seed) {  
 this.cap = cap;  
 this.seed = seed;  
 }  
  
 // hash函数，采用简单的加权和hash  
 public int hash(String value) {  
 int result = 0;  
 int len = value.length();  
 for (int i = 0; i < len; i++) {  
 result = seed \* result + value.charAt(i);  
 }  
 return (cap - 1) & result;  
 }  
 }  
}

# 案例实现

## 开发准备

### 创建工程

创建Maven工程，并加入依赖。pom.xml为：

<?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>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.2.RELEASE</version>  
 </parent>  
 <groupId>cn.itcast.crawler</groupId>  
 <artifactId>itcast-crawler-job</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
 <!--SpringMVC-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <!--SpringData Jpa-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 </dependency>  
  
 <!--MySQL连接包-->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
  
 <!--WebMagic核心包-->  
 <dependency>  
 <groupId>us.codecraft</groupId>  
 <artifactId>webmagic-core</artifactId>  
 <version>0.7.3</version>  
 <exclusions>  
 <exclusion>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 </exclusion>  
 </exclusions>  
 </dependency>  
 <!--WebMagic扩展-->  
 <dependency>  
 <groupId>us.codecraft</groupId>  
 <artifactId>webmagic-extension</artifactId>  
 <version>0.7.3</version>  
 </dependency>  
 <!--WebMagic对布隆过滤器的支持-->  
 <dependency>  
 <groupId>com.google.guava</groupId>  
 <artifactId>guava</artifactId>  
 <version>16.0</version>  
 </dependency>  
  
 <!--工具包-->  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-lang3</artifactId>  
 </dependency>  
 </dependencies>  
   
</project>

### 加入配置文件

添加application.properties配置文件

#DB Configuration:  
spring.datasource.driverClassName=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/crawler  
spring.datasource.username=root  
spring.datasource.password=root  
  
#JPA Configuration:  
spring.jpa.database=*MySQL*spring.jpa.show-sql=**true**

### 编写Pojo

@Entity  
public class JobInfo {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 private String companyName;  
 private String companyAddr;  
 private String companyInfo;  
 private String jobName;  
 private String jobAddr;  
 private String jobInfo;  
 private Integer salaryMin;  
 private Integer salaryMax;  
 private String url;  
 private String time;

get/set

toString()

}

### 编写Dao

public interface JobInfoDao extends JpaRepository<JobInfo, Long> {  
}

### 编写Service

编写Service接口

public interface JobInfoService {  
  
 */\*\*  
 \* 保存数据  
 \*  
 \** ***@param*** *jobInfo  
 \*/* public void save(JobInfo jobInfo);  
  
 */\*\*  
 \* 根据条件查询数据  
 \*  
 \** ***@param*** *jobInfo  
 \** ***@return*** *\*/* public List<JobInfo> findJobInfo(JobInfo jobInfo);  
}

编写Service实现类

@Service  
public class JobInfoServiceImpl implements JobInfoService {  
  
 @Autowired  
 private JobInfoDao jobInfoDao;  
  
 @Override  
 @Transactional  
 public void save(JobInfo jobInfo) {  
 //先从数据库查询数据,根据发布日期查询和url查询  
 JobInfo param = new JobInfo();  
 param.setUrl(jobInfo.getUrl());  
 param.setTime(jobInfo.getTime());  
 List<JobInfo> list = this.findJobInfo(param);  
  
 if (list.size() == 0) {  
 //没有查询到数据则新增或者修改数据  
 this.jobInfoDao.saveAndFlush(jobInfo);   
 }  
 }  
  
 @Override  
 public List<JobInfo> findJobInfo(JobInfo jobInfo) {  
  
 Example example = Example.*of*(jobInfo);  
  
 List<JobInfo> list = this.jobInfoDao.findAll(example);  
  
 return list;  
 }  
}

### 编写引导类

@SpringBootApplication  
@EnableScheduling//开启定时任务  
public class Application {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(Application.class, args);  
 }  
}

## 功能实现

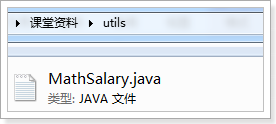
### 编写url解析功能

@Component  
public class JobProcessor implements PageProcessor {  
  
 @Autowired  
 private SpringDataPipeline springDataPipeline;  
  
 @Scheduled(initialDelay = 1000, fixedDelay = 1000 \* 100)  
 public void process() {  
 //访问入口url地址  
 String url = "https://search.51job.com/list/000000,000000,0000,01%252C32,9,99,java,2,1.html?lang=c&stype=&postchannel=0000&workyear=99&cotype=99&degreefrom=99&jobterm=99&companysize=99&providesalary=99&lonlat=0%2C0&radius=-1&ord\_field=0&confirmdate=9&fromType=&dibiaoid=0&address=&line=&specialarea=00&from=&welfare=";  
 Spider.*create*(new JobProcessor())  
 .addUrl(url)   
 .setScheduler(new QueueScheduler()  
 .setDuplicateRemover(new BloomFilterDuplicateRemover(10000000)))  
 .thread(5)  
 .run();  
 }  
  
 @Override  
 public void process(Page page) {  
 //获取页面数据  
 List<Selectable> nodes = page.getHtml().$("div#resultList div.el").nodes();  
  
 //判断nodes是否为空  
 if (nodes.isEmpty()) {  
 try {  
 //如果为空，表示这是招聘信息详情页保存信息详情  
 this.saveJobInfo(page);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 } else {  
 //如果有值，表示这是招聘信息列表页  
 for (Selectable node : nodes) {  
 //获取招聘信息详情页url  
 String jobUrl = node.links().toString();  
 //添加到url任务列表中，等待下载  
 page.addTargetRequest(jobUrl);  
  
 //获取翻页按钮的超链接  
 List<String> listUrl = page.getHtml().$("div.p\_in li.bk").links().all();  
 //添加到任务列表中  
 page.addTargetRequests(listUrl);  
  
 }  
 }  
 }

}

### 编写页面解析功能

薪水的计算需要添加课堂资料的工具类MathSalary进行计算



实现以下逻辑

*/\*\*  
 \* 解析页面，获取招聘详情  
 \*  
 \** ***@param*** *\*/*private void saveJobInfo(Page page) {  
 //创建招聘信息对象  
 JobInfo jobInfo = new JobInfo();  
 Html html = page.getHtml();  
  
 //公司名称  
 jobInfo.setCompanyName(html.$("div.tHeader p.cname a", "text").toString());  
 //公司地址  
 jobInfo.setCompanyAddr(html.$("div.tBorderTop\_box:nth-child(3) p.fp", "text").toString());  
 //公司信息  
 jobInfo.setCompanyInfo(html.$("div.tmsg", "text").toString());  
 //职位名称  
 jobInfo.setJobName(html.$("div.tHeader > div.in > div.cn > h1", "text").toString());  
 //工作地点  
 jobInfo.setJobAddr(html.$("div.tHeader > div.in > div.cn > span.lname", "text").toString());  
 //职位信息  
 jobInfo.setJobInfo(Jsoup.*parse*(html.$("div.tBorderTop\_box:nth-child(2)").toString()).text());  
 //工资范围  
 String salaryStr = html.$("div.tHeader > div.in > div.cn > strong", "text").toString();  
 jobInfo.setSalaryMin(MathSalary.*getSalary*(salaryStr)[0]);  
 jobInfo.setSalaryMax(MathSalary.*getSalary*(salaryStr)[1]);  
 //职位详情url  
 jobInfo.setUrl(page.getUrl().toString());  
 //职位发布时间  
 String time = html.$("div.jtag > div.t1 > span.sp4", "text").regex(".\*发布").toString();  
 jobInfo.setTime(time.substring(0, time.length() - 2));

//保存数据

page.putField("jobInfo", jobInfo);

}

## 使用和定制Pipeline

在WebMagic中，Pileline是抽取结束后，进行处理的部分，它主要用于抽取结果的保存，也可以定制Pileline可以实现一些通用的功能。在这里我们会定制Pipeline实现数据导入到数据库中

### Pipeline输出

Pipeline的接口定义如下：

public interface Pipeline {  
  
 // ResultItems保存了抽取结果，它是一个Map结构，  
 // 在page.putField(key,value)中保存的数据，

//可以通过ResultItems.get(key)获取  
 public void process(ResultItems resultItems, Task task);  
}

可以看到，Pipeline其实就是将PageProcessor抽取的结果，继续进行了处理的，其实在Pipeline中完成的功能，你基本上也可以直接在PageProcessor实现，那么为什么会有Pipeline？有几个原因：

* 为了模块分离

“页面抽取”和“后处理、持久化”是爬虫的两个阶段，将其分离开来，一个是代码结构比较清晰，另一个是以后也可能将其处理过程分开，分开在独立的线程以至于不同的机器执行。

* Pipeline的功能比较固定，更容易做成通用组件

每个页面的抽取方式千变万化，但是后续处理方式则比较固定，例如保存到文件、保存到数据库这种操作，这些对所有页面都是通用的。

在WebMagic里，一个Spider可以有多个Pipeline，使用Spider.addPipeline()即可增加一个Pipeline。这些Pipeline都会得到处理，例如可以使用

spider.addPipeline(new ConsolePipeline()).addPipeline(new FilePipeline())

实现输出结果到控制台，并且保存到文件的目标。

### 已有的Pipeline

WebMagic中就已经提供了控制台输出、保存到文件、保存为JSON格式的文件几种通用的Pipeline。

| **类** | **说明** | **备注** |
| --- | --- | --- |
| ConsolePipeline | 输出结果到控制台 | 抽取结果需要实现toString方法 |
| FilePipeline | 保存结果到文件 | 抽取结果需要实现toString方法 |
| JsonFilePipeline | JSON格式保存结果到文件 |  |
| ConsolePageModelPipeline | (注解模式)输出结果到控制台 |  |
| FilePageModelPipeline | (注解模式)保存结果到文件 |  |
| JsonFilePageModelPipeline | (注解模式)JSON格式保存结果到文件 | 想持久化的字段需要有getter方法 |

### 案例自定义Pipeline导入数据

自定义SpringDataPipeline

@Component  
public class SpringDataPipeline implements Pipeline {  
  
 @Autowired  
 private JobInfoService jobInfoService;  
  
 @Override  
 public void process(ResultItems resultItems, Task task) {  
 //获取需要保存到MySQL的数据  
 JobInfo jobInfo = resultItems.get("jobInfo");  
  
 //判断获取到的数据不为空  
 if(jobInfo!=null) {  
 //如果有值则进行保存  
 this.jobInfoService.save(jobInfo);  
 }  
 }  
}

在JobProcessor中修改process()启动的逻辑，添加代码

@Autowired  
private SpringDataPipeline springDataPipeline;

public void process() {  
 Spider.*create*(new JobProcessor())  
 .addUrl(url)  
 .addPipeline(this.springDataPipeline)  
 .setScheduler(new QueueScheduler()  
 .setDuplicateRemover(new BloomFilterDuplicateRemover(10000000)))  
 .thread(5)  
 .run();  
}

# 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”

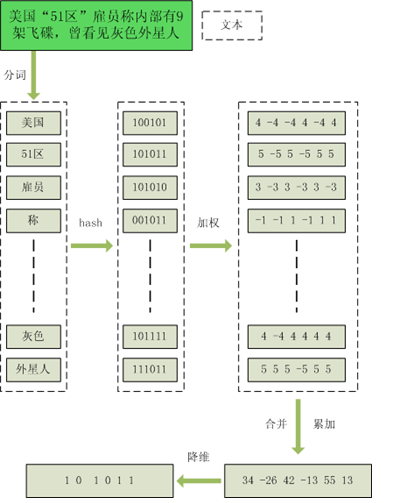
“51区”计算为 “ 5 -5 5 -5 5 5”。

**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。

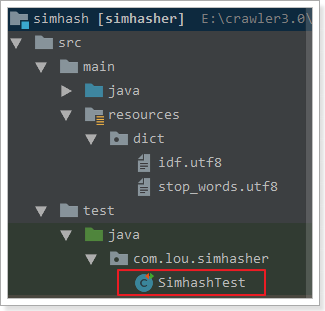
举例如下： 10101 和 00110 从第一位开始依次有第一位、第四、第五位不同，则海明距离为3。对于二进制字符串的a和b，海明距离为等于在a XOR b运算结果中1的个数（普遍算法）。

#### 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。

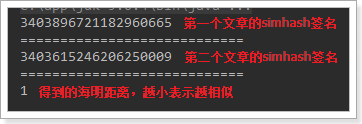
导入工程simhash，并打开测试用例。



#### 测试simhash

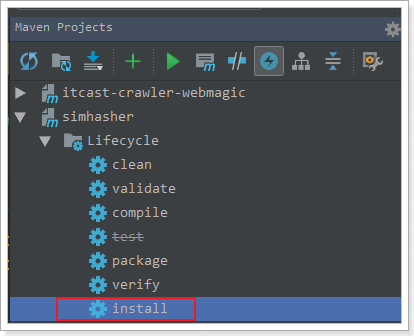
按照测试用例的要求，准备两个文件，就是需要进行对比的文章

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



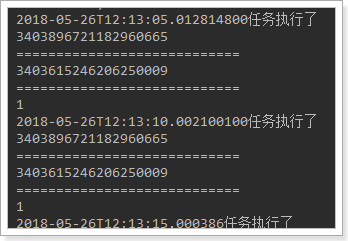
在案例的pom.xml中加入以下依赖

<!--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

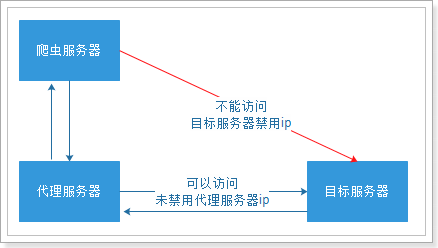
有些网站不允许爬虫进行数据爬取，因为会加大服务器的压力。其中一种最有效的方式是通过ip+时间进行鉴别，因为正常人不可能短时间开启太多的页面，发起太多的请求。

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;  
 }  
}

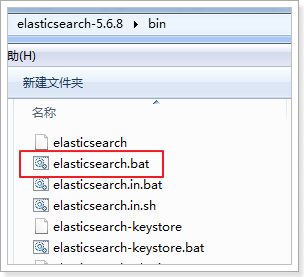
# ElasticSearch环境准备

## 安装ElasticSearch服务

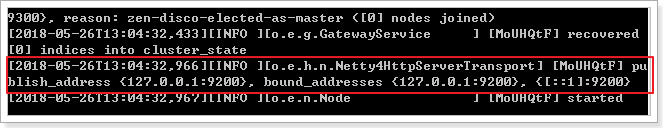
课堂资料中的elasticsearch-5.6.8.zip进行解压



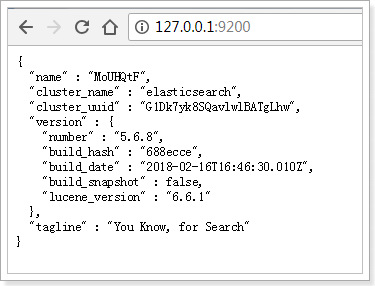
启动服务：



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

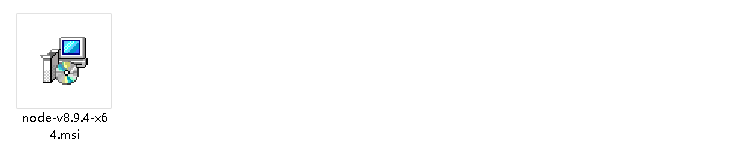
安装ElasticSearch的head插件，完成图形化界面的效果，完成索引数据的查看。采用本地安装方式进行head插件的安装。elasticsearch-5-\*以上版本安装head需要安装node和grunt。

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

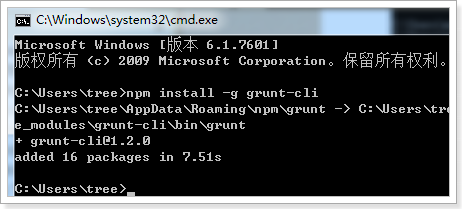


3）将grunt安装为全局命令 ，Grunt是基于Node.js的项目构建工具

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

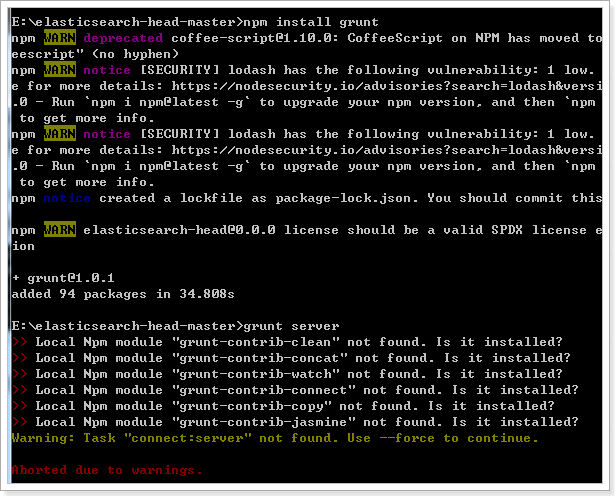


根据提示访问，效果如下：

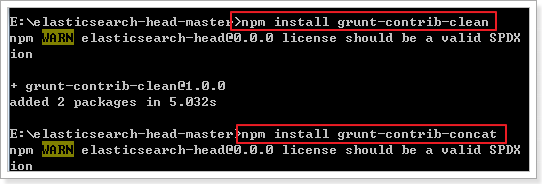


PS：如果第5步失败，执行以下命令

npm install grunt

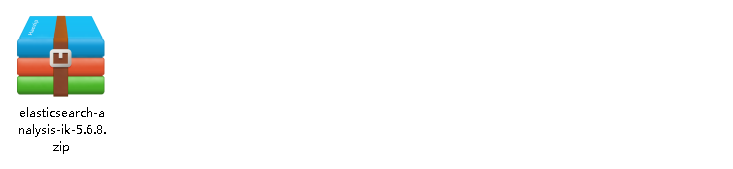


再根据提示按以下方式依次安装组件

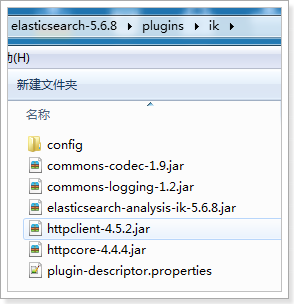


## 安装IK分词器

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

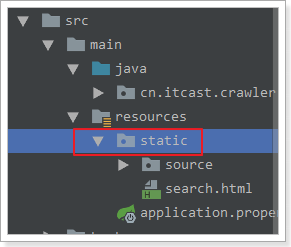
先执行createIndex()方法创建索引，再执行jobData()导入数据到索引库

@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：

