**网络爬虫**

# 案例需求

前面介绍了几种爬虫的分类，这里我们使用聚焦网络爬虫，抓取汽车之家上的汽车评测数据。<https://www.autohome.com.cn/bestauto/>



我们需要抓取汽车之家上面所有的汽车评测数据

* 需求分析

在页面上我们分析，需要抓取以下部分的数据



1. 车型信息

***品牌 车型 款式***

吉利汽车 远景X3 2017款 1.5L 手动尊贵型

1. 评测信息

***测试项目 测试结果***

加速(0-100公里/小时) 13.65秒

刹车(100-0公里/小时) 39.50米

实测油耗(升/100公里) 7.7升

排名是动态生成的，我们这里不做抓取，可以后期单独处理排名

1. 编辑点评

***评测编辑 编辑点评***

于晗 它和远景X1有不同的风格。。。

周宇轩 除了动力弱点儿。。。

李坤 我比较喜欢它的内外饰设计。。。

1. 评测图片

有5张图片，页面显示的是小图，我们需要打开超链接获取大图的url地址，再单独下载图片

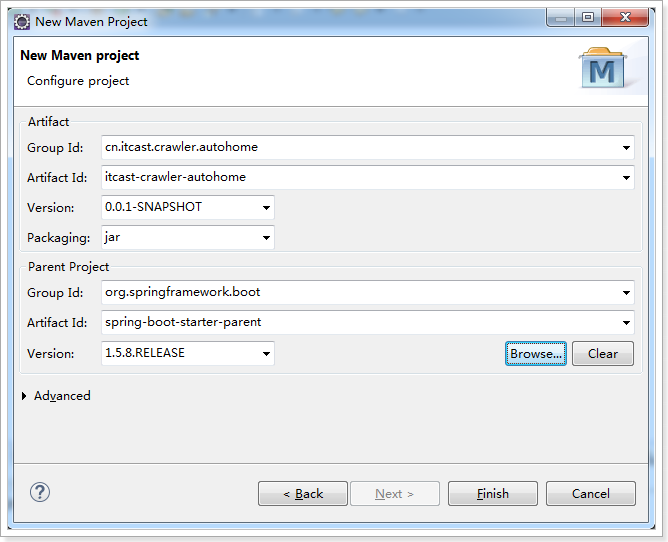
# 环境准备

使用技术：

jdk8+SpringBoot1.5.8+SpringMVC4+MyBatis3.4+HttpClient4.5.3+Jsoup2.10+Quartz2.2

## 搭建工程

创建Maven工程



工程需要继承SpringBoot的parent

<parent>

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

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

<version>1.5.8.RELEASE</version>

</parent>

### 加入依赖

设置jdk为1.8，并加入相关依赖

<properties>

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

</properties>

<dependencies>

<!-- spring相关包 -->

<dependency>

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

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

</dependency>

<!-- 配置测试启动器 -->

<dependency>

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

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

<scope>test</scope>

</dependency>

<!-- MyBatis启动器 -->

<dependency>

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

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

<version>1.1.1</version>

</dependency>

<!-- MySQL连接驱动 -->

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<!-- 工具类 -->

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.3.2</version>

</dependency>

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.6</version>

</dependency>

<!-- 定时任务Quartz -->

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>2.2.1</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</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>

</dependencies>

### 加入配置

在src/main/resources路径下加入以下两个配置文件

加入application.properties

#日志

logging.level.org.mybatis=DEBUG

logging.level.cn.itcast=DEBUG

#DBConfiguration:

spring.datasource.driverClassName=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/crawler?useUnicode=true&characterEncoding=utf8

spring.datasource.username=root

spring.datasource.password=root

## SpringBoot入门

项目使用的是SpringBoot+ssm框架，这里先实现入门程序，用以熟悉SpringBoot的使用。

需求：浏览器访问，获取数据库时间

### 编写引导类

@SpringBootApplication

**public** **class** Application {

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

SpringApplication.*run*(Application.**class**, args);

}

}

### 编写Mapper

@Mapper

**public** **interface** TestMapper {

/\*\*

\* 查询数据库时间

\*

\* **@return**

\*/

@Select("SELECT NOW()")

**public** String queryDate();

}

### 编写Service接口

**public** **interface** TestService {

/\*\*

\* 查询数据库时间

\*

\* **@return**

\*/

**public** String queryDate();

}

### 编写Service实现

@Service

**public** **class** TestServiceImpl **implements** TestService {

@Autowired

**private** TestMapper testMapper;

@Override

**public** String queryDate() {

String date = **this**.testMapper.queryDate();

**return** date;

}

}

### 编写Controller

@RestController

**public** **class** TestController {

@Autowired

**private** TestService testService;

// http://127.0.0.1:8080/date

/\*\*

\* 查询数据库时间

\*

\* **@return**

\*/

@RequestMapping("date")

**public** String queryDate() {

String date = **this**.testService.queryDate();

**return** date;

}

}

# 开发分析

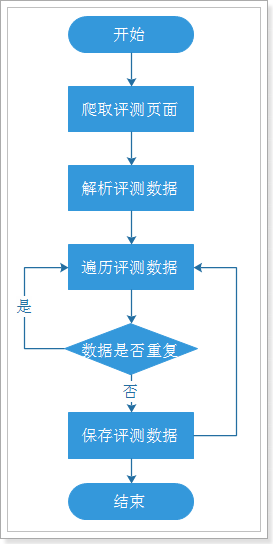
## 流程分析

分析发现，评测页的url是

https://www.autohome.com.cn/bestauto/1

最后一个参数是页码数，我们只需要按顺序从第一页开始，把所有的页面都抓取下来就可以了。

抓取页面的流程如下



抓取评测数据：

1. 根据url抓取html页面
2. 对html页面进行解析，获取该页面所有的评测数据
3. 遍历所有的评测数据
4. 判断遍历的评测数据是否已保存，

如果已保存再次遍历下一条评测数据

如果未保存执行下一步

1. 保存评测数据到数据库中

## 数据库表设计

根据以上需求，设计数据库表。sql如下

CREATE TABLE `car\_test` (

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

`title` varchar(100) NOT NULL COMMENT '评测车辆的名字',

`test\_speed` int(150) DEFAULT NULL COMMENT '评测项目-加速(0-100公里/小时),单位毫秒',

`test\_brake` int(150) DEFAULT NULL COMMENT '评测项目-刹车(100-0公里/小时),单位毫米',

`test\_oil` int(150) DEFAULT NULL COMMENT '评测项目-实测油耗(升/100公里)，单位毫升',

`editor\_name1` varchar(10) DEFAULT NULL COMMENT '评测编辑1',

`editor\_remark1` varchar(1000) DEFAULT NULL COMMENT '点评内容1',

`editor\_name2` varchar(10) DEFAULT NULL COMMENT '评测编辑2',

`editor\_remark2` varchar(1000) DEFAULT NULL COMMENT '点评内容2',

`editor\_name3` varchar(10) DEFAULT NULL COMMENT '评测编辑3',

`editor\_remark3` varchar(1000) DEFAULT NULL COMMENT '点评内容3',

`image` varchar(1000) DEFAULT NULL COMMENT '评测图片，5张图片名，中间用,分隔',

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

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

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=7 DEFAULT CHARSET=utf8 COMMENT='汽车之家评测表';

## 开发准备

编写Pojo

**public** **class** CarTest {

**private** Long id;

**private** String title;

**private** **int** test\_speed;

**private** **int** test\_brake;

**private** **int** test\_oil;

**private** String editor\_name1;

**private** String editor\_remark1;

**private** String editor\_name2;

**private** String editor\_remark2;

**private** String editor\_name3;

**private** String editor\_remark3;

**private** String image;

**private** Date created;

**private** Date updated;

get/set。。。

}

编写Mapper

@Mapper

**public** **interface** CarTestMapper {

}

编写Service接口

**public** **interface** CarTestService {

}

编写Service接口实现

@Service

**public** **class** CarTestServiceImpl **implements** CarTestService {

}

# 实现爬取数据

## 实现ApiService

首先需要编写用来抓取页面的ApiService。

### 编写连接池管理器

这里使用连接池管理，而连接池管理器应该交给Spring进行管理，我们这里使用以下两个注解

@Configuration注解声明配置类。

@Bean注解声明如何创建这实例

@Configuration

**public** **class** HttpClientCMCfg {

@Bean

**public** PoolingHttpClientConnectionManager poolingHttpClientConnectionManager() {

// 创建连接管理器

PoolingHttpClientConnectionManager cm = **new** PoolingHttpClientConnectionManager();

// 设置最大连接数

cm.setMaxTotal(200);

// 设置每个并发连接数

cm.setDefaultMaxPerRoute(20);

**return** cm;

}

}

### 定时关闭失效连接

这里使用Quartz定时任务来处理定时关闭失效连接

#### 编写定时任务

//同步执行任务(单线程)

@DisallowConcurrentExecution

**public** **class** CloseConnectJob **extends** QuartzJobBean {

@Override

**protected** **void** executeInternal(JobExecutionContext context) **throws** JobExecutionException {

// 获取spring容器

ApplicationContext applicationContext = (ApplicationContext) context.getJobDetail().getJobDataMap()

.get("context");

// 从容器中获取HttpClient连接管理器

PoolingHttpClientConnectionManager cm = applicationContext.getBean(PoolingHttpClientConnectionManager.**class**);

// 关闭失效连接

cm.closeExpiredConnections();

System.***out***.println("关闭失效连接");

}

}

#### 定时执行任务

@Configuration

**public** **class** SchedledCfg {

// 定义关闭无效连接任务

@Bean("closeConnectJobBean")

**public** JobDetailFactoryBean closeConnectJobBean() {

JobDetailFactoryBean jobDetailFactoryBean = **new** JobDetailFactoryBean();

jobDetailFactoryBean.setApplicationContextJobDataKey("context");

jobDetailFactoryBean.setJobClass(CloseConnectJob.**class**);

jobDetailFactoryBean.setDurability(**true**);

**return** jobDetailFactoryBean;

}

// 定义关闭无效连接触发器

@Bean("closeConnectJobTrigger")

**public** CronTriggerFactoryBean closeConnectJobTrigger(

@Qualifier(value = "closeConnectJobBean") JobDetailFactoryBean itemJobBean) {

CronTriggerFactoryBean tigger = **new** CronTriggerFactoryBean();

tigger.setJobDetail(itemJobBean.getObject());

tigger.setCronExpression("0/5 \* \* \* \* ? ");

**return** tigger;

}

// 定义调度器

@Bean

**public** SchedulerFactoryBean schedulerFactory(CronTrigger[] cronTriggerImpl) {

SchedulerFactoryBean bean = **new** SchedulerFactoryBean();

bean.setTriggers(cronTriggerImpl);

**return** bean;

}

}

### 编写ApiService接口

需要实现两个功能的下载：

1. Get请求获取页面数据
2. Get请求下载图片

**public** **interface** ApiService {

/\*\*

\* 使用get请求获取页面数据

\*

\* **@param** url

\* **@return**

\*/

**public** String getHtml(String url);

/\*\*

\* 使用get请求下载图片,返回图片名

\* **@param** url

\* **@return**

\*/

**public** **void** getImage(String url);

}

### 实现ApiService接口

@Service

**public** **class** ApiServiceImpl **implements** ApiService {

@Autowired

**private** PoolingHttpClientConnectionManager cm;

@Override

**public** String getHtml(String url) {

// 使用连接池管理器获取连接

CloseableHttpClient httpClient = HttpClients.*custom*().setConnectionManager(cm).build();

// 创建httpGet请求

HttpGet httpGet = **new** HttpGet(url);

CloseableHttpResponse response = **null**;

**try** {

// 发起请求

response = httpClient.execute(httpGet);

// 判断请求是否成功

**if** (response.getStatusLine().getStatusCode() == 200) {

// 判断是否有响应体

**if** (response.getEntity() != **null**) {

// 如果有响应体，则进行解析

String html = EntityUtils.*toString*(response.getEntity(), "UTF-8");

// 返回

**return** html;

}

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

// 释放连接

**if** (response != **null**) {

**try** {

response.close();

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**return** **null**;

}

@Override

**public** String getImage(String url) {

// 使用连接池管理器获取连接

CloseableHttpClient httpClient = HttpClients.*custom*().setConnectionManager(cm).build();

// 创建httpGet请求

HttpGet httpGet = **new** HttpGet(url);

CloseableHttpResponse response = **null**;

**try** {

// 发起请求

response = httpClient.execute(httpGet);

// 判断请求是否成功

**if** (response.getStatusLine().getStatusCode() == 200) {

// 判断是否有响应体

**if** (response.getEntity() != **null**) {

// 如果有响应体，则下载

// 获取文件后缀

String extName = StringUtils.*substringAfterLast*(url, ".");

// 用UUID重新生成文件名

String image = UUID.*randomUUID*().toString() + "." + extName;

// 下载图片

OutputStream outstream = **new** FileOutputStream(**new** File("D:/images/" + image));

response.getEntity().writeTo(outstream);

// 返回图片名

**return** image;

}

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

// 释放连接

**if** (response != **null**) {

**try** {

response.close();

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**return** **null**;

}

### 测试ApiService接口

这里使用SpringBoot的测试组件，需要添加如下两个注解：

@RunWith(value = SpringJUnit4ClassRunner.class)

让测试运行在spring的环境，这样我们测试的时候就可以和开发的时候一样编写代码，例如使用@Autowired注解直接注入

@SpringBootTest(classes = Application.class)

执行当前的这个类是测试类

测试代码如下

@RunWith(value = SpringJUnit4ClassRunner.**class**)

@SpringBootTest(classes = Application.**class**)

**public** **class** ApiServiceTest {

@Autowired

**private** ApiService apiService;

@Test

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

String html = **this**.apiService.getHtml("https://www.autohome.com.cn/bestauto/");

System.***out***.println(html);

}

@Test

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

**this**.apiService.getImage(

"https://car2.autoimg.cn/cardfs/product/g24/M09/AE/EB/800x0\_1\_q87\_autohomecar\_\_wKgHIVpxGh6AFSN1AAY8kcz3Aww921.jpg");

}

}

## 去重过滤器

在使用网络爬虫过程中，去重是一个不可避免的问题，这里需要对抓取的数据内容进行过滤，就是对车辆幸好名称进行去重过滤，避免同样条数据反复保存到数据库中。

传统的去重，可以使用Map或者Set集合、哈希表的方式来实现去重，在数据量较小的情况下，使用这种方式没有问题。可是当我们需要大量爬去数据的时候，这种方式就存在很大问题。因为会极大的占用内存和系统资源，导致爬虫系统崩溃。这里将会使用布隆过滤器

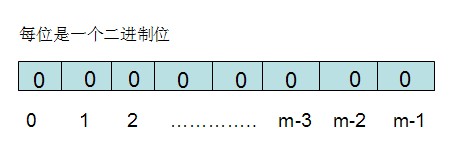
### Bloom过滤器介绍

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。

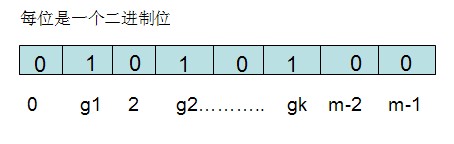
哈希表也能用于判断元素是否在集合中，但是布隆过滤器只需要哈希表的1/8或1/4的空间复杂度就能完成同样的问题。布隆过滤器可以插入元素，但不可以删除已有元素。其中的元素越多，误报率越大，但是漏报是不可能的。

**布隆过滤器原理**

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

//去重过滤器，布隆过滤器

**public** **class** TitleFilter {

/\* 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;

}

}

}

### 初始化去重过滤器

项目一启动，就应该创建去重过滤器。

编写以下代码实现过滤器初始化

@Configuration

**public** **class** TitleFilterCfg {

@Autowired

**private** CarTestService carTestService;

@Bean

**public** TitleFilter titleFilter() {

// 创建车辆标题过滤器

TitleFilter titleFilter = **new** TitleFilter();

// 从数据库查询车辆标题，分页查询

List<String> list = **this**.carTestService.queryByPage(1, 5000);

// 遍历查询结果

**for** (String str : list) {

// 把查询到的数据放到过滤器中

titleFilter.add(str);

}

// 返回创建好的过滤器

**return** titleFilter;

}

}

编写Service接口

**public** **interface** CarTestService {

/\*\*

\* 分页查询数据

\*

\* **@param** page

\* **@param** rows

\* **@return**

\*/

List<String> queryByPage(Integer page, Integer rows);

}

实现Service接口

@Service

**public** **class** CarTestServiceImpl **implements** CarTestService {

@Autowired

**private** CarTestMapper carTestMapper;

@Override

**public** List<String> queryByPage(Integer page, Integer rows) {

// 设置从哪一条数据开始查

Integer start = (page - 1) \* rows;

// 设置参数

Map<String, Integer> map = **new** HashMap<String, Integer>();

map.put("start", start);

map.put("rows", rows);

// 分页查询数据

List<String> list = **this**.carTestMapper.queryByPage(map);

// 返回查询结果

**return** list;

}

}

编写Mapper接口

@Mapper

**public** **interface** CarTestMapper {

/\*\*

\* 分页查询数据

\*

\* **@param** start

\* **@param** rows

\* **@return**

\*/

@Select("SELECT title FROM car\_test LIMIT #{start},#{rows}")

List<String> queryByPage(Map<String, Integer> map);

}

## 实现数据爬取

首先实现数据爬取逻辑，先在测试方法中实现

### 实现爬取测试方法

@RunWith(SpringJUnit4ClassRunner.**class**)

@SpringBootTest(classes = Application.**class**)

**public** **class** CrawlerAutohome {

@Autowired

**private** ApiService apiService;

@Autowired

**private** CarTestService carTestService;

@Autowired

**private** TitleFilter titleFilter;

@Test

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

//遍历所有的url

**for** (**int** i = 1; i < 139; i++) {

String html = **this**.apiService.getHtml("https://www.autohome.com.cn/bestauto/" + i);

Document doc = Jsoup.*parse*(html);

// 获取每获取评测信息

Elements cars = doc.select("#bestautocontent div.uibox");

// 遍历评测信息

**for** (Element car : cars) {

// 去重判读

String title = car.getElementsByClass("uibox-title uibox-title-border").text();

**if** (**this**.titleFilter.contains(title)) {

// 如果包含了，就不保存了，遍历下一个

**continue**;

}

// 创建评测对象,封装数据

CarTest carTest = **this**.copyCarTest(car);

// 评测图片，下载图片

String image = **this**.getImage(car);

// 设置图片

carTest.setImage(image);

// 保存数据

**this**.saveCarTest(carTest);

}

}

}

/\*\*

\* 保存汽车评测数据

\*

\* **@param** carTest

\*/

**private** **void** saveCarTest(CarTest carTest) {

**this**.titleFilter.add(carTest.getTitle());

**this**.carTestService.saveCarTest(carTest);

}

/\*\*

\* 解析数据下载评测图片

\*

\* **@param** car

\* **@return**

\*/

**private** String getImage(Element car) {

List<String> images = **new** ArrayList<String>();

Elements elements = car.select(".piclist-box ul.piclist02 a");

**for** (Element element : elements) {

String url = "https:" + element.attr("href");

String html = **this**.apiService.getHtml(url);

Document doc = Jsoup.*parse*(html);

String picUrl = "https:" + doc.select("#img").attr("src");

String image = **this**.apiService.getImage(picUrl);

images.add(image);

**break**;

}

**return** StringUtils.*join*(images, ",");

}

/\*\*

\* 解析数据封装成汽车评测对象

\*

\* **@param** car

\* **@return**

\*/

**private** CarTest copyCarTest(Element car) {

CarTest carTest = **new** CarTest();

// 评测车辆标题

String title = car.getElementsByClass("uibox-title uibox-title-border").text();

carTest.setTitle(title);

// 评测项目-加速(0-100公里/小时),单位毫秒

String speed = car.select(".tabbox1 dd:nth-child(2) div.dd-div2").first().text();

carTest.setTest\_speed(**this**.strToNum(speed));

// 评测项目-刹车(100-0公里/小时),单位毫米

String brake = car.select(".tabbox1 dd:nth-child(3) div.dd-div2").first().text();

carTest.setTest\_brake(**this**.strToNum(brake));

// 评测项目-实测油耗(升/100公里),单位毫升

String oil = car.select(".tabbox1 dd:nth-child(4) div.dd-div2").first().text();

carTest.setTest\_oil(**this**.strToNum(oil));

// 评测编辑1

carTest.setEditor\_name1(car.select(".tabbox2 dd:nth-child(2) > div.dd-div1").first().text());

// 点评内容1

carTest.setEditor\_remark1(car.select(".tabbox2 dd:nth-child(2) > div.dd-div3").first().text());

// 评测编辑2

carTest.setEditor\_name2(car.select(".tabbox2 dd:nth-child(3) > div.dd-div1").first().text());

// 点评内容2

carTest.setEditor\_remark2(car.select(".tabbox2 dd:nth-child(3) > div.dd-div3").first().text());

// 评测编辑3

carTest.setEditor\_name3(car.select(".tabbox2 dd:nth-child(4) > div.dd-div1").first().text());

// 点评内容3

carTest.setEditor\_remark3(car.select(".tabbox2 dd:nth-child(4) > div.dd-div3").first().text());

// 设置时间

carTest.setCreated(**new** Date());

carTest.setUpdated(carTest.getCreated());

**return** carTest;

}

/\*\*

\* 把字符串去掉最后一个数，转为乘以1000的数字

\*

\* **@param** speed

\* **@return**

\*/

**private** **int** strToNum(String str) {

**try** {

// 字符串去掉随后一个数

str = StringUtils.*substring*(str, 0, str.length() - 1);

// 转换为小数并乘以1000

Number num = Float.*valueOf*(str) \* 1000;

**return** num.intValue();

} **catch** (Exception e) {

e.printStackTrace();

System.***out***.println(str);

}

**return** 0;

}

}

### 编写CarTestService接口

**public** **interface** CarTestService {

/\*\*

\* 分页查询数据

\*

\* **@param** page

\* **@param** rows

\* **@return**

\*/

List<String> queryByPage(Integer page, Integer rows);

/\*\*

\* 保存数据

\*

\* **@param** carTest

\*/

**void** saveCarTest(CarTest carTest);

}

### 编写CarTestService接口实现

@Service

**public** **class** CarTestServiceImpl **implements** CarTestService {

@Autowired

**private** CarTestMapper carTestMapper;

@Override

**public** List<String> queryByPage(Integer page, Integer rows) {

// 设置从哪一条数据开始查

Integer start = (page - 1) \* rows;

// 设置参数

Map<String, Integer> map = **new** HashMap<String, Integer>();

map.put("start", start);

map.put("rows", rows);

// 分页查询数据

List<String> list = **this**.carTestMapper.queryByPage(map);

// 返回查询结果

**return** list;

}

@Override

**public** **void** saveCarTest(CarTest carTest) {

**this**.carTestMapper.save(carTest);

}

}

### 编写CarTestMapper

@Mapper

**public** **interface** CarTestMapper {

/\*\*

\* 分页查询数据

\*

\* **@param** start

\* **@param** rows

\* **@return**

\*/

@Select("SELECT title FROM car\_test LIMIT #{start},#{rows}")

List<String> queryByPage(Map<String, Integer> map);

/\*\*

\* 保存

\*

\* **@param** carTest

\*/

@Insert(

"INSERT INTO `car\_test` (" +

" `title`," +

" `test\_speed`," +

" `test\_brake`," +

" `test\_oil`," +

" `editor\_name1`," +

" `editor\_remark1`," +

" `editor\_name2`," +

" `editor\_remark2`," +

" `editor\_name3`," +

" `editor\_remark3`," +

" `image`," +

" `created`," +

" `updated`" +

")" +

"VALUES" +

" (" +

" #{title}," +

" #{test\_speed}," +

" #{test\_brake}," +

" #{test\_oil}," +

" #{editor\_name1}," +

" #{editor\_remark1}," +

" #{editor\_name2}," +

" #{editor\_remark2}," +

" #{editor\_name3}," +

" #{editor\_remark3}," +

" #{image}," +

" #{created}," +

" #{updated}" +

" )")

**void** save(CarTest carTest);

}

## 整合任务

把测试方法中的爬取数据代码改造为任务，再使用Quartz定时任务定时处理，就可以实现定时抓取汽车评测数据，能够获取最新的数据了

### 改造任务

//同步执行任务(单线程)

@DisallowConcurrentExecution

**public** **class** CrawlerAutohomeJob **extends** QuartzJobBean {

**private** ApiService apiService;

**private** CarTestService carTestService;

**private** TitleFilter titleFilter;

@Override

**protected** **void** executeInternal(JobExecutionContext context) **throws** JobExecutionException {

// 获取spring容器

ApplicationContext applicationContext = (ApplicationContext) context.getJobDetail().getJobDataMap()

.get("context");

// 获取抓取数据服务

**this**.apiService = applicationContext.getBean(ApiService.**class**);

// 获取汽车评测服务

**this**.carTestService = applicationContext.getBean(CarTestService.**class**);

// 获取过滤器

**this**.titleFilter = applicationContext.getBean(TitleFilter.**class**);

// 遍历所有的url

**for** (**int** i = 1; i < 139; i++) {

// 抓取页面数据

String html = **this**.apiService.getHtml("https://www.autohome.com.cn/bestauto/" + i);

// 使用jsoup解析为Document对象

Document doc = Jsoup.*parse*(html);

// 获取每获取评测信息

Elements cars = doc.select("#bestautocontent div.uibox");

// 遍历评测信息

**for** (Element car : cars) {

// 去重判读

String title = car.getElementsByClass("uibox-title uibox-title-border").text();

**if** (**this**.titleFilter.contains(title)) {

// 如果包含了，就不保存了，遍历下一个

**continue**;

}

// 创建评测对象,封装数据

CarTest carTest = **this**.copyCarTest(car);

// 评测图片，下载图片

String image = **this**.getImage(car);

// 设置图片

carTest.setImage(image);

// 保存数据

**this**.saveCarTest(carTest);

}

}

}

/\*\*

\* 保存汽车评测数据

\*

\* **@param** carTest

\*/

**private** **void** saveCarTest(CarTest carTest) {

**this**.titleFilter.add(carTest.getTitle());

**this**.carTestService.saveCarTest(carTest);

}

/\*\*

\* 解析数据下载评测图片

\*

\* **@param** car

\* **@return**

\*/

**private** String getImage(Element car) {

List<String> images = **new** ArrayList<String>();

Elements elements = car.select(".piclist-box ul.piclist02 a");

**for** (Element element : elements) {

String url = "https:" + element.attr("href");

String html = **this**.apiService.getHtml(url);

Document doc = Jsoup.*parse*(html);

String picUrl = "https:" + doc.select("#img").attr("src");

String image = **this**.apiService.getImage(picUrl);

images.add(image);

**break**;

}

**return** images.toString();

}

/\*\*

\* 解析数据封装成汽车评测对象

\*

\* **@param** car

\* **@return**

\*/

**private** CarTest copyCarTest(Element car) {

CarTest carTest = **new** CarTest();

// 评测车辆标题

String title = car.getElementsByClass("uibox-title uibox-title-border").text();

carTest.setTitle(title);

// 评测项目-加速(0-100公里/小时),单位毫秒

String speed = car.select(".tabbox1 dd:nth-child(2) div.dd-div2").first().text();

carTest.setTest\_speed(**this**.strToNum(speed));

// 评测项目-刹车(100-0公里/小时),单位毫米

String brake = car.select(".tabbox1 dd:nth-child(3) div.dd-div2").first().text();

carTest.setTest\_brake(**this**.strToNum(brake));

// 评测项目-实测油耗(升/100公里),单位毫升

String oil = car.select(".tabbox1 dd:nth-child(4) div.dd-div2").first().text();

carTest.setTest\_oil(**this**.strToNum(oil));

// 评测编辑1

carTest.setEditor\_name1(car.select(".tabbox2 dd:nth-child(2) > div.dd-div1").first().text());

// 点评内容1

carTest.setEditor\_remark1(car.select(".tabbox2 dd:nth-child(2) > div.dd-div3").first().text());

// 评测编辑2

carTest.setEditor\_name2(car.select(".tabbox2 dd:nth-child(3) > div.dd-div1").first().text());

// 点评内容2

carTest.setEditor\_remark2(car.select(".tabbox2 dd:nth-child(3) > div.dd-div3").first().text());

// 评测编辑3

carTest.setEditor\_name3(car.select(".tabbox2 dd:nth-child(4) > div.dd-div1").first().text());

// 点评内容3

carTest.setEditor\_remark3(car.select(".tabbox2 dd:nth-child(4) > div.dd-div3").first().text());

// 设置时间

carTest.setCreated(**new** Date());

carTest.setUpdated(carTest.getCreated());

**return** carTest;

}

/\*\*

\* 把字符串去掉最后一个数，转为乘以1000的数字

\*

\* **@param** speed

\* **@return**

\*/

**private** **int** strToNum(String str) {

**try** {

// 字符串去掉随后一个数

str = StringUtils.*substring*(str, 0, str.length() - 1);

// 转换为小数并乘以1000

Number num = Float.*valueOf*(str) \* 1000;

**return** num.intValue();

} **catch** (Exception e) {

e.printStackTrace();

System.***out***.println(str);

}

**return** 0;

}

}

### 定时处理任务

@Configuration

**public** **class** SchedledCfg {

// 定义关闭无效连接任务

@Bean("crawlerAutohomeJobBean")

**public** JobDetailFactoryBean crawlerAutohomeJobBean() {

JobDetailFactoryBean jobDetailFactoryBean = **new** JobDetailFactoryBean();

jobDetailFactoryBean.setApplicationContextJobDataKey("context");

jobDetailFactoryBean.setJobClass(CrawlerAutohomeJob.**class**);

jobDetailFactoryBean.setDurability(**true**);

**return** jobDetailFactoryBean;

}

// 定义关闭无效连接触发器

@Bean("crawlerAutohomeJobTrigger")

**public** CronTriggerFactoryBean crawlerAutohomeJobTrigger(

@Qualifier(value = "crawlerAutohomeJobBean") JobDetailFactoryBean itemJobBean) {

CronTriggerFactoryBean tigger = **new** CronTriggerFactoryBean();

tigger.setJobDetail(itemJobBean.getObject());

tigger.setCronExpression("0/5 \* \* \* \* ? ");

**return** tigger;

}

// 定义关闭无效连接任务

@Bean("closeConnectJobBean")

**public** JobDetailFactoryBean closeConnectJobBean() {

JobDetailFactoryBean jobDetailFactoryBean = **new** JobDetailFactoryBean();

jobDetailFactoryBean.setApplicationContextJobDataKey("context");

jobDetailFactoryBean.setJobClass(CloseConnectJob.**class**);

jobDetailFactoryBean.setDurability(**true**);

**return** jobDetailFactoryBean;

}

// 定义关闭无效连接触发器

@Bean("closeConnectJobTrigger")

**public** CronTriggerFactoryBean closeConnectJobTrigger(

@Qualifier(value = "closeConnectJobBean") JobDetailFactoryBean itemJobBean) {

CronTriggerFactoryBean tigger = **new** CronTriggerFactoryBean();

tigger.setJobDetail(itemJobBean.getObject());

tigger.setCronExpression("0/5 \* \* \* \* ? ");

**return** tigger;

}

// 定义调度器

@Bean

**public** SchedulerFactoryBean schedulerFactory(CronTrigger[] cronTriggerImpl) {

SchedulerFactoryBean bean = **new** SchedulerFactoryBean();

bean.setTriggers(cronTriggerImpl);

**return** bean;

}

}