**网络爬虫**

**第二天**

# 第一章：课程计划

1. WebMagic介绍
2. WebMagic功能
3. 爬虫分类
4. 案例开发分析
5. 案例实现

学习目标：

1. 能够说出WebMagic的作用（第二章：2.1）
2. 能够理解WebMagic架构（第二章：2.2）
3. 能够完成入门案例（第二章：2.3）
4. 能够理解PageProcessor的作用（第三章：3.1）
5. 能够使用选择器抽取页面元素（第三章：3.1.1）
6. 能够使用Spider启动爬虫（第三章：3.3）
7. 能够使用Site设置爬虫参数（第三章：3.3.2）
8. 能够使用过滤器对url去重（第五章：5.3.2）
9. 能够使用和定制Pipeline输出数据（第六章：6.3）

# 第二章：WebMagic介绍

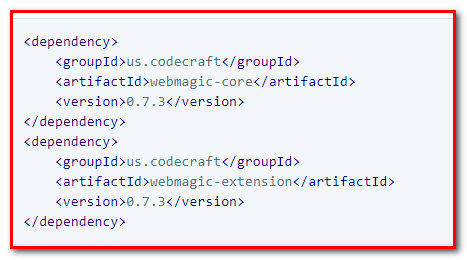
## WebMaigc的作用

网站：

<http://webmagic.io/>

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。



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。

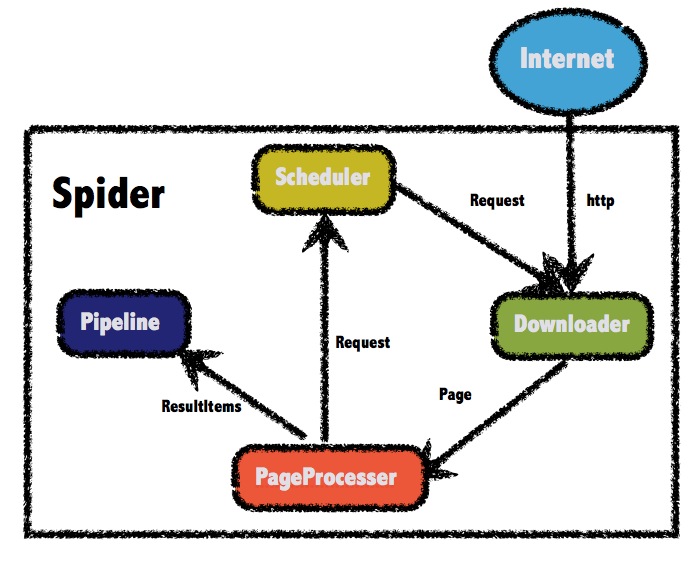
WebMagic的设计目标是尽量的模块化，并体现爬虫的功能特点。这部分提供非常简单、灵活的API，在基本不改变开发模式的情况下，编写一个爬虫。

## 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

网站：<http://webmagic.io/>

WebMagic开发案例：



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。

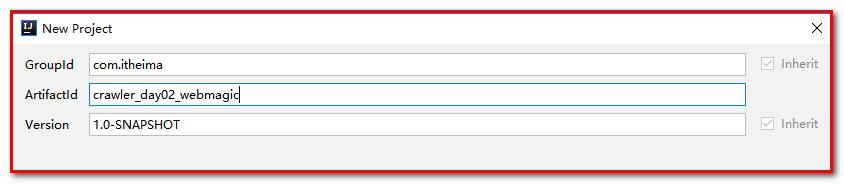
Page是WebMagic抽取过程的核心对象，它提供一些方法可供抽取、结果保存等。

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

### 创建工程



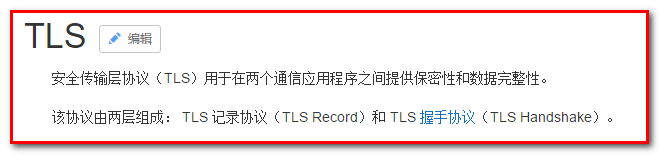
### 加入依赖

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**>com.itheima</**groupId**>  
 <**artifactId**>crawler\_day02\_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TLS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TLS v1.2的网站会有TLS的异常抛出。

什么是TLS：



解决方案：

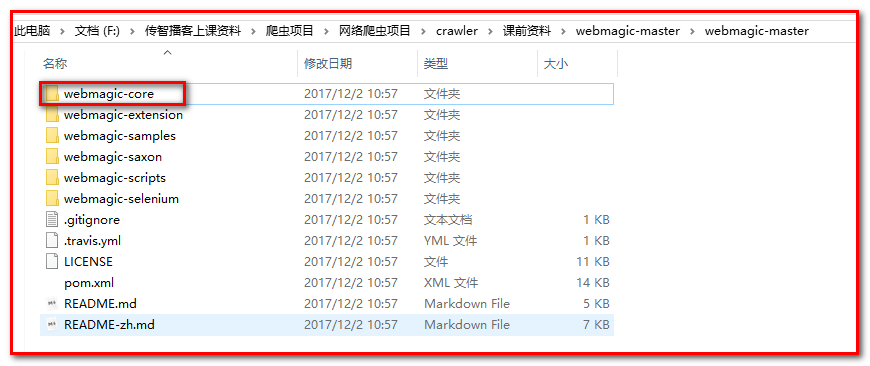
* + - 1. 等作者的0.7.4的版本发布
      2. 直接从github上下载最新的代码，安装到本地仓库

也可以参考以下资料自己修复

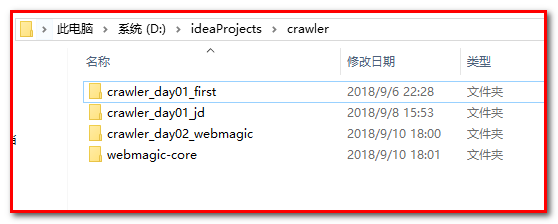
<https://github.com/code4craft/webmagic/issues/701>



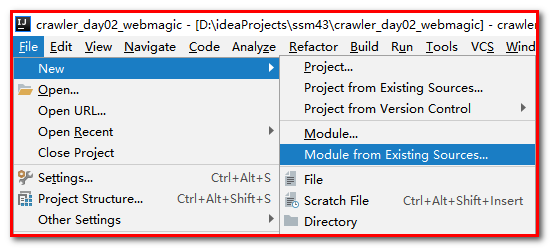
第一步：选择github进行下载，下载webmagic-master.zip，并解压



第二步：将webmagic-core拷贝到工程下

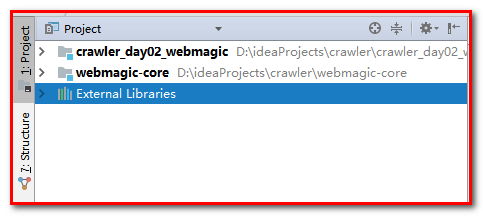


第三步：导入到idea

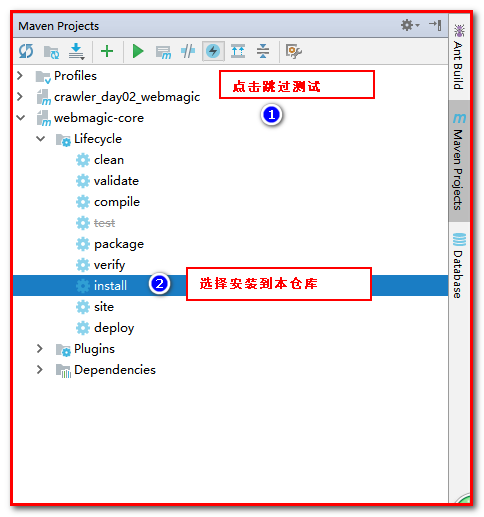


选择maven的模块，点击下一步。

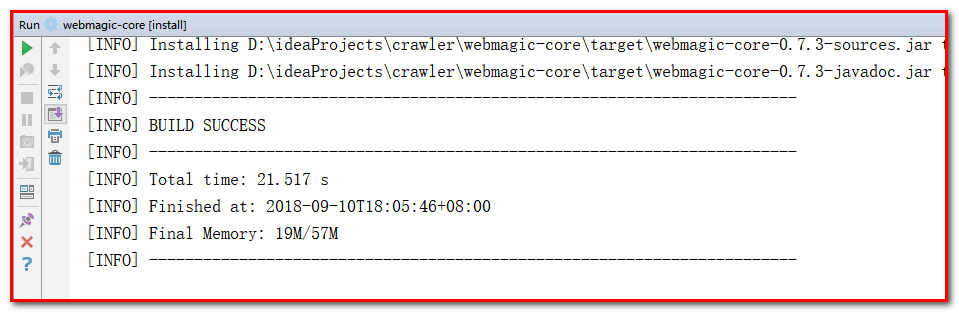
最终查看idea工程。



第四步：将webmagic-core安装到本地仓库



表示成功



### 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**

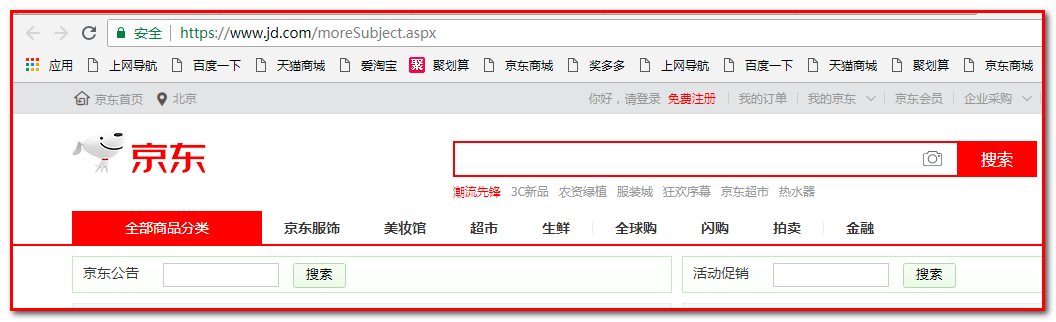
### 案例实现

需求：查找京东案例

访问www.jd.com点击“更多”



访问页面：



查看页面源码

获取“京东公告”



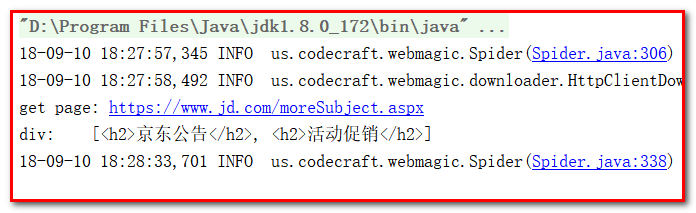
实现代码：

创建包com.itheima.webmagic.test，创建类JobProcessor.java

**public class** JobProcessor **implements** PageProcessor {  
  
 *//解析页面* **public void** process(Page page) {  
 *//解析返回的数据page，并且把解析的结果放到ResultItems中  
 //css选择器  
 // 获取解析的页面* Html html = page.getHtml();  
*// System.out.println(html);  
 // 返回多个值  
// List<String> all = html.css("div.mt h2").all(); // all()表示返回多个值  
// for (String s : all) {  
// System.out.println(s);  
// }  
 // 返回第1个值* String h2 = html.css(**"div.mt h2"**).toString(); *// toString()返回的是第一个值* System.***out***.println(h2);  
  
 *// 读取h2的文本* String h2text = html.css(**"div.mt h2"**,**"text"**).toString(); *// toString()返回的是第一个值* System.***out***.println(h2text);

}  
  
  
 **private** Site **site** = Site.*me*(); *;* **public** Site getSite() {  
 **return site**;  
 }  
  
 *//主函数，执行爬虫* **public static void** main(String[] args) {  
 Spider spider = Spider.*create*(**new** JobProcessor())  
 .addUrl(**"https://www.jd.com/moreSubject.aspx"**); *//设置爬取数据的页面  
 //执行爬虫* spider.run();  
 }  
}

打印结果：

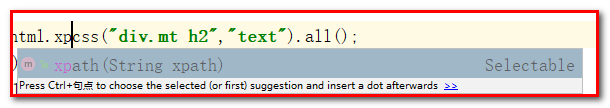


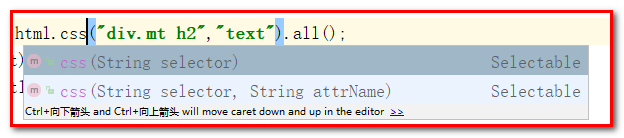
# 第三章：WebMagic功能

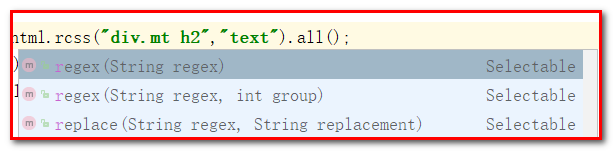
## 实现PageProcessor

### 抽取元素Selectable

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

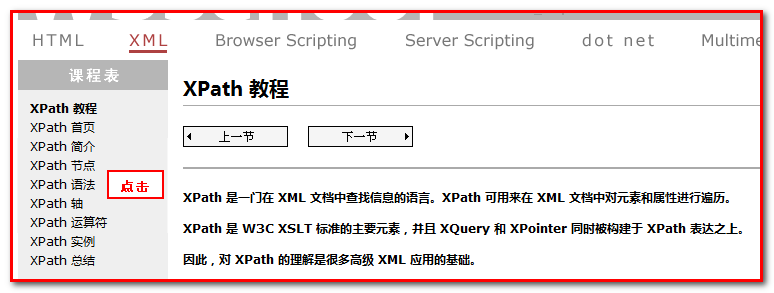


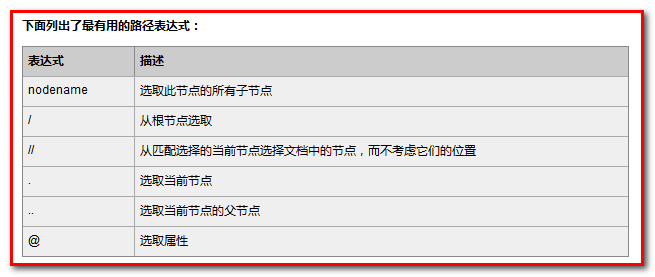




#### XPath

找到w3scshool文档。找到Xpath语法





以上是获取属性class=mt的div标签，里面的h1标签的内容

*//XPath*page.getHtml().xpath(**"//div[@id=news\_div]/ul/li/div/a"**).toString();

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

#### CSS选择器（常用）

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>h2表示class为mt的div标签下的直接子元素h2标签

page.getHtml().css(**"div.mt>h2"**).toString();

可是使用:nth-child(n)选择第几个元素，如下选择第一个元素

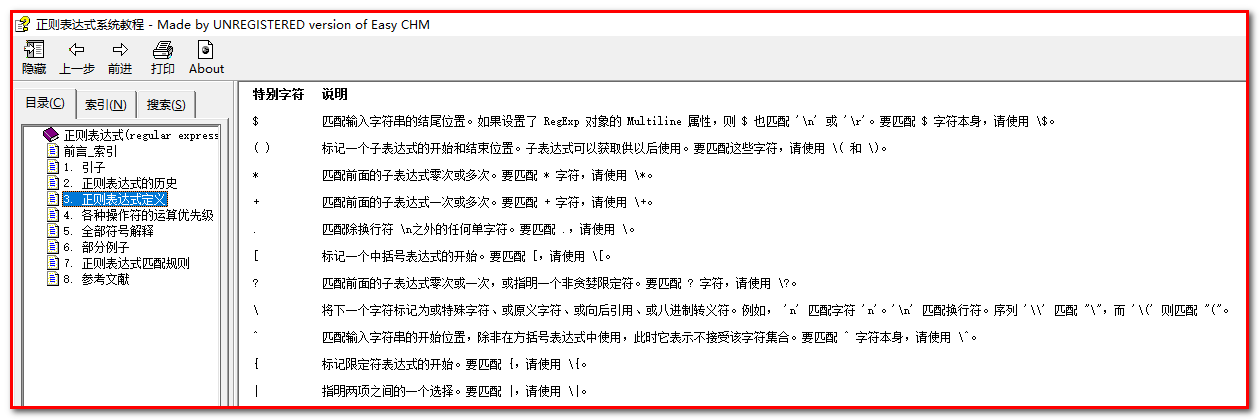
page.getHtml().css(**"div#news\_div > ul > li:nth-child(1) a"**).toString();

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》



page.getHtml().css(**"div#news\_div a"**).regex(**"^.\*江苏.\*$"**).all();

### 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") |
| $(String selector,String attr) | 使用Css选择器选择，获取文本 | html.$("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



方法组合使用：

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

获取链接link()的用法：

*// link()的练习*List<String> listA = html.$(**"div#news\_div"**).links().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。

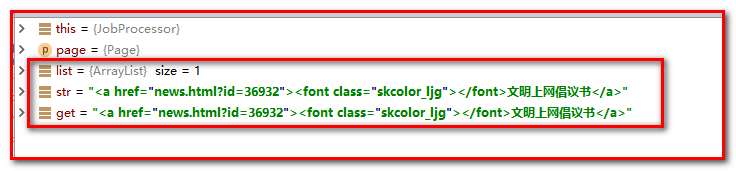
我们知道，一条抽取规则，无论是XPath、CSS选择器或者正则表达式，总有可能抽取到多条元素。WebMagic对这些进行了统一，可以通过不同的API获取到一个或者多个元素。

| **方法** | **说明** | **示例** |
| --- | --- | --- |
| get() | 返回一条String类型的结果 | String link= html.links().get() |
| toString() | 等同于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 a"**)  
 .regex(**"^.\*文明.\*$"**).toString();  
  
String get = page.getHtml()  
 .css(**"div#news\_div a"**)  
 .regex(**"^.\*文明.\*$"**).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（.links()）

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

所有符合以9结尾的正则表达式的url地址

使用page.addTargetRequests()并将这些链接加入到待抓取的schedler队列中去。

*//获取链接（放置到Request中）*page.addTargetRequests(page.getHtml().css(**"div#news\_div"**).links().regex(**"^.\*9$"**).all();

使用page.putField将获取到的结果数据放置到ResultItems对象中

*// 在获取链接中的<h1>标签的内容（放置到ResultItems对象中）*

page.putField(**"url"**,page.getHtml().css(**"div.mt h1"**).all());

查看结果



## 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。

*//主函数，执行爬虫* **public static void** main(String[] args) {  
 Spider spider = Spider.*create*(**new** JobProcessor())  
 .addUrl(**"https://www.jd.com/moreSubject.aspx"**) *//设置爬取数据的页面*

.addPipeline(**new** ConsolePipeline()).addPipeline(**new** FilePipeline(**"D:\\crawler\\result"**))  
 .thread(5);*// 可以多个线程服务，速度加快  
  
 //执行爬虫* spider.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/](http://webmagic.io/docs/" \t "_blank)") |
| 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 QueueScheduler()) |
| 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/](http://webmagic.io/docs/" \t "_blank)") |
| 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/docs/" \t "_blank)", "[http://webmagic.io/xxx](http://webmagic.io/xxx" \t "_blank)") |

### 爬虫配置Site

Site.me()可以对爬虫进行一些参数配置，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/" \t "_blank)") |
| setHttpProxy(HttpHost) | 设置Http代理 | site.setHttpProxy(new HttpHost("127.0.0.1",8080)) |

## 最终代码：

创建类：JobProcessor.java

**package** com.itheima.webmagic.test;  
  
**import** us.codecraft.webmagic.Page;  
**import** us.codecraft.webmagic.Site;  
**import** us.codecraft.webmagic.Spider;  
**import** us.codecraft.webmagic.pipeline.FilePipeline;  
**import** us.codecraft.webmagic.processor.PageProcessor;  
  
**public class** JobProcessor **implements** PageProcessor {  
  
 */\*\*  
 \* process：用来处理Html页面  
 \* \* Page：处理页面的核心对象（获取html页面之外，还可以调用：Request、ResultItem对象）  
 \** ***@param page*** *page  
 \*/* **public void** process(Page page) {  
 *// 获取解析的页面* Html html = page.getHtml();  
*// System.out.println(html);  
 // 返回多个值  
// List<String> all = html.css("div.mt h2").all(); // all()表示返回多个值  
// for (String s : all) {  
// System.out.println(s);  
// }  
  
// String h2 = html.css("div.mt h2").toString(); // toString()返回的是第一个值  
// System.out.println(h2);  
  
 // 读取h2的文本  
// String h2text = html.css("div.mt h2","text").toString(); // toString()返回的是第一个值  
// System.out.println(h2text);  
  
 // XPath的练习  
// String sXpath = html.xpath("//div[@id=news\_div]/ul/li/div/a").toString();  
// System.out.println(sXpath);  
 // css的练习  
// String sCss = html.css("div#news\_div>ul>li>div>a").toString();  
// System.out.println(sCss);  
// String sCss1 = html.css("div#news\_div>ul>li:nth-child(2)>div>a").toString();  
// System.out.println(sCss1);  
 // 正则表达式练习($等同于css)  
// List<String> listA = html.$("div#news\_div a").regex("^.\*东北.\*$").all();  
// for (String s : listA) {  
// System.out.println(s);  
// }  
 // link()的练习  
// List<String> listA = html.$("div#news\_div").links().all();  
// for (String s : listA) {  
// System.out.println(s);  
// }  
// 所有符合以9结尾的正则表达式的url地址  
// 使用page.addTargetRequests()并将这些链接加入到待抓取的schedler队列中去。* List<String> list = html.$(**"div#news\_div"**).links().regex(**"^.\*9$"**).all();  
 page.addTargetRequests(list);  
 *// 获取详细页面的内容  
 // 使用page.putField将获取到的结果数据放置到ResultItems对象中（默认在控制台输出）* page.putField(**"div"**,html.css(**"div.mt h1"**,**"text"**).toString());  
 Object div = page.getResultItems().get(**"div"**);  
 System.***out***.println(**"div:"**+div);  
  
 }  
  
 */\*\*  
 \* Site对象，用来设置参数  
 \*/* Site **site** = Site.*me*()  
 .setCharset(**"UTF-8"**) *// 设置页面的编码集* .setSleepTime(1) *// 设置抓取数据的间隔时间 单位毫秒* .setTimeOut(10\*1000) *// 设置超时时间 单位毫秒* .setRetrySleepTime(3\*1000) *// 设置重试的抓取时间 单位毫秒* .setRetryTimes(3); *// 设置重试抓取次数*   
 **public** Site getSite() {  
 **return site**;  
 }  
  
 **public static void** main(String[] args) {  
 Spider spider = Spider.*create*(**new** JobProcessor()) *// 创建当前爬虫实例* .addUrl(**"https://www.jd.com/moreSubject.aspx"**)*// 指定爬取的首页* .addPipeline(**new** ConsolePipeline()) *// 在控制台输出结果* .addPipeline(**new** FilePipeline(**"D:\\crawler\\result"**))*// 在文件中输出结果* .thread(5); *// 多线程执行，开启5个线程进行处理* spider.run();*// 运行爬虫* }  
}

# 第四章：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。

例如：<https://news.baidu.com/>



## 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。

例如：<http://www.manmanbuy.com/>





## 增量式网络爬虫（用的不多）

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。

例如：华尔街见闻

<https://wallstreetcn.com/live/global>



## 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。

例如：QQ空间，需要账号和密码才能访问。

总结：

通用网络爬虫：爬所有的数据

聚焦网络爬虫：爬感兴趣的数据，有目的的爬取

增量式网络爬虫：爬变化的数据，追求实时性

深层网络爬虫：做处理，如登录后才能爬的数据

# 第五章：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：

（1）职位、（2）工作地点、（3）公司名称、（4）薪资、

（5）发布时间、（6）职位信息、（7）公司联系方式、（8）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',

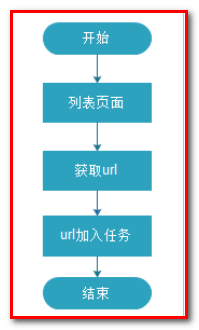
PRIMARY KEY (`id`)

) 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，而且统计出的数据也不准确，因为存在重复数据。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。

（1）对待抓取的URL队列进行管理

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 | 我们使用该方式（默认） |
| 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 |

（2）对已抓取的URL进行去重

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（默认） |
| BloomFilterDuplicateRemover | 使用BloomFilter来进行去重，占用内存较小，但是可能漏抓页面 |

1：验证默认使用HashSetDuplicateRemover去重：

第一步：添加：

*//解析页面***public void** process(Page page) {

*//获取链接  
// page.addTargetRequests(page.getHtml().css("div#news\_div").links().regex("^.\*9$").all());  
// page.putField("url",page.getHtml().css("div.mt h1").all());*

page.addTargetRequest(**"https://www.jd.com/news.html?id=37319"**);  
 page.addTargetRequest(**"https://www.jd.com/news.html?id=37319"**);  
 page.addTargetRequest(**"https://www.jd.com/news.html?id=37319"**);

}

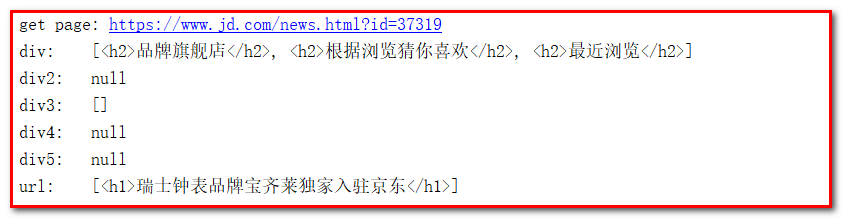
第二步：Main方法

*//主函数，执行爬虫* **public static void** main(String[] args) {  
 Spider spider = Spider.*create*(**new** JobProcessor())  
 .addUrl(**"https://www.jd.com/moreSubject.aspx"**) *//设置爬取数据的页面  
 //.addPipeline(new FilePipeline("D:\\crawler\\result"))* .thread(5);

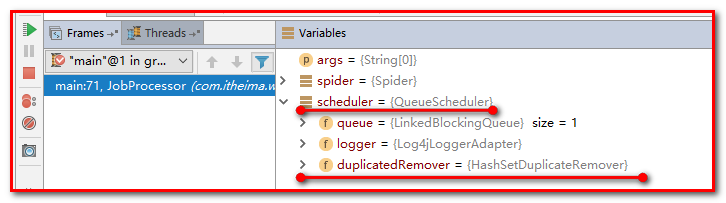
Scheduler scheduler = spider.getScheduler();

*//执行爬虫* spider.run();  
 }

第三步：查看测试结果：

<https://www.jd.com/news.html?id=37319地址只出现一次，说明有去重的效果。>

使用断点查看sceduler，也可以看到默认是HashSetDuplicateRemover：



2：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。

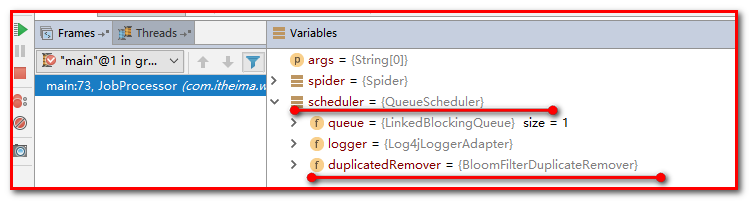
如果要使用BloomFilter（布隆过滤器），必须要加入以下依赖：

*<!--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

*//主函数，执行爬虫***public static void** main(String[] args) {  
 Spider spider = Spider.*create*(**new** JobProcessor())  
 .addUrl(**"https://www.jd.com/moreSubject.aspx"**) *//设置爬取数据的页面  
 //.addPipeline(new FilePipeline("D:\\crawler\\result"))* .thread(5)  
 .setScheduler(**new** QueueScheduler().setDuplicateRemover(**new** BloomFilterDuplicateRemover(10000000)));*//设置布隆去重过滤器，指定最多对1000万数据进行去重操作* Scheduler scheduler = spider.getScheduler();  
  
 *//执行爬虫* spider.run();  
}

查看断点：



### 三种去重方式

去重就有三种实现方式，那有什么不同呢？

* **HashSet（默认）**

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。

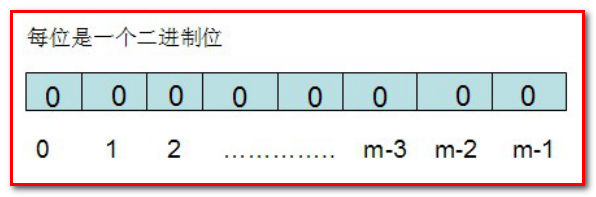
缺点：有误判的可能。没有重复可能会判定重复，但是重复数据一定会判定重复。

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。

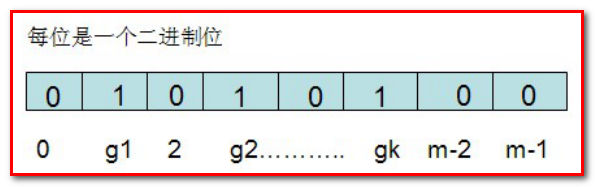
哈希表也能用于判断元素是否在集合中，但是布隆过滤器只需要哈希表的1/8或1/4的空间复杂度就能完成同样的问题。布隆过滤器可以插入元素，但不可以删除已有元素。其中的元素越多，误报率越大，但是漏报是不可能的。

**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。

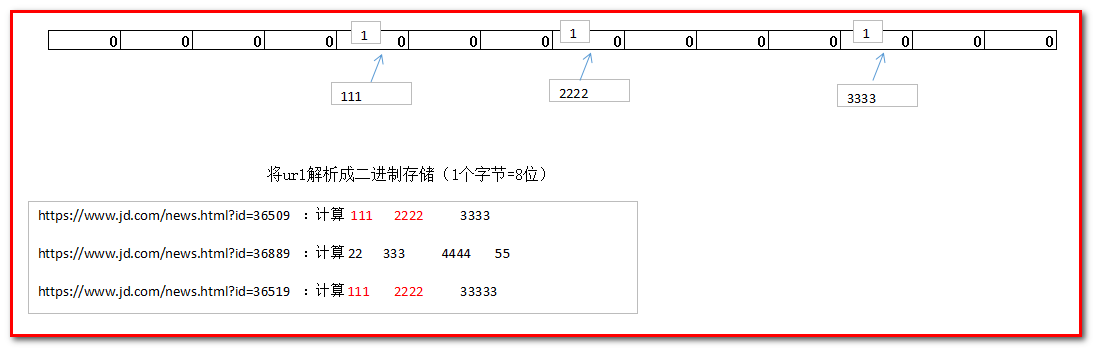


对于有n个元素的集合S={S1,S2...Sn},通过k个映射函数{f1,f2,......fk}，将集合S中的每个元素Sj(1<=j<=n)映射为K个值{g1,g2...gk}，然后再将位数组array中相对应的array[g1],array[g2]......array[gk]置为1：



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。



计算后的结果，只要有1个被占用（相同），就会被去重，所有有误判的可能。

### 布隆过滤器实现（了解）

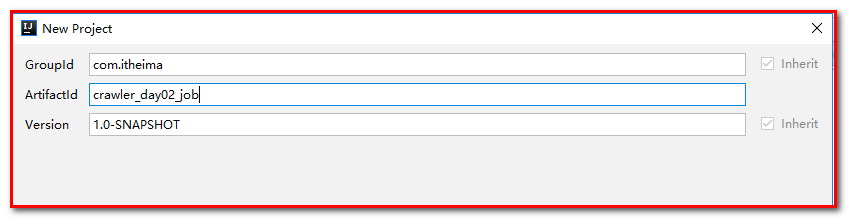
以下是一个布隆过滤器的实现，可以参考

*//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;  
 }  
 }  
}

# 第六章：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**>com.itheima</**groupId**>  
 <**artifactId**>crawler\_day02\_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/itcastcrawler  
spring.datasource.username**=**root  
spring.datasource.password**=**root***#JPA Configuration:***spring.jpa.database**=**MySQL  
spring.jpa.show-sql**=**true**

添加log4j.properties

**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**

### 编写Domain

创建包com.itheima.job.Domain，创建类JobInfo.java

@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

创建包com.itheima.job.dao，创建类JobInfoDao.java

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

### 编写Service

编写Service接口

创建包com.itheima.job.service，创建类JobInfoService.java

**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  
@Transactional  
**public class** JobInfoServiceImpl **implements** JobInfoService {  
  
 @Autowired  
 **private** JobInfoDao **jobInfoDao**;  
  
 @Override  
 **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，判断数据库是否已经存在相同数据* 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) {

*//设置查询条件*  
 Example example = Example.*of*(jobInfo);  
 List<JobInfo> list = **this**.**jobInfoDao**.findAll(example);  
 **return** list;  
 }  
}

### 编写引导类

在com.itheima.job的包下，创建SpringBoot的引导类。

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

### 编写解析url功能测试

创建包com.itheima.job.utils，创建JobProcessor.java的类，使用定时任务启动process()方法。

@Component  
**public class** JobProcessor **implements** PageProcessor {  
 @搜索java；行业选择：互联网/电子商务+计算机软件  
 **private** String **url** = **"https://search.51job.com/list/040000,000000,0000,01%252C32,9,99,%2520,2,1.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="**;  
 @Override  
 **public void** process(Page page) {  
 String html = page.getHtml().toString();  
 System.***out***.println(html);  
 }  
  
 **private** Site **site** = Site.*me*()  
 .setCharset(**"gbk"**)*//设置编码* .setTimeOut(10 \* 1000)*//设置超时时间（10秒）* .setRetrySleepTime(3000)*//设置重试的间隔时间（3秒）* .setRetryTimes(3);*//设置重试的次数（3次）* @Override  
 **public** Site getSite() {  
 **return site**;  
 }  
  
 *// initialDelay:当任务启动后，等待多久执行该方法（1000表示1秒）  
 // fixedDelay：每隔多久执行方法（100\*1000表示100秒）* @Scheduled(initialDelay = 1000,fixedDelay = 100\*1000)  
 **public void** process(){  
 Spider.*create*(**new** JobProcessor())  
 .addUrl(**url**)  
 .setScheduler(**new** QueueScheduler().setDuplicateRemover(**new** BloomFilterDuplicateRemover(100000)))  
 .thread(10)  
 .run();  
 }  
}

## 功能实现

### 编写url解析功能



@Component  
**public class** JobProcessor **implements** PageProcessor {  
  
 **private** String **url** = **"https://search.51job.com/list/040000,000000,0000,01%252C32,9,99,%2520,2,1.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="**;  
 @Override  
 **public void** process(Page page) {  
 *//解析页面，获取招聘信息详情的url地址* List<Selectable> list = page.getHtml().css(**"div#resultList div.el"**).nodes();  
  
  
 *//判断获取到的集合是否为空* **if** (list.size() == 0) {  
 *// 如果为空，表示这是招聘详情页,解析页面，获取招聘详情信息，保存数据  
 // this.saveJobInfo(page);* } **else** {  
 *//如果不为空，表示这是列表页,解析出详情页的url地址，放到任务队列中* **for** (Selectable selectable : list) {  
 *//获取url地址* String jobInfoUrl = selectable.links().toString();  
 System.***out***.println(**"jobInfoUrl:"**+jobInfoUrl);  
 *//把获取到的url地址放到任务队列中* page.addTargetRequest(jobInfoUrl);  
 }  
  
 *//获取下一页的url  
 //获取翻页按钮的超链接* List<String> listUrl = page.getHtml().$(**"div.p\_in li.bk"**).links().all();  
 *//把url添加到任务列表中* page.addTargetRequests(listUrl);  
  
 }  
  
 // String html = page.getHtml().toString();  
 }  
  
 **private** Site **site** = Site.*me*()  
 .setCharset(**"gbk"**)*//设置编码* .setTimeOut(10 \* 1000)*//设置超时时间* .setRetrySleepTime(3000)*//设置重试的间隔时间* .setRetryTimes(3);*//设置重试的次数* @Override  
 **public** Site getSite() {  
 **return site**;  
 }  
  
 *// initialDelay:当任务启动后，等待多久执行该方法（1000表示1秒）  
 // fixedDelay：每隔多久执行方法（100\*1000表示100秒）* @Scheduled(initialDelay = 1000,fixedDelay = 100\*1000)  
 **public void** process(){  
 Spider.*create*(**new** JobProcessor())  
 .addUrl(**url**)  
 .setScheduler(**new** QueueScheduler().setDuplicateRemover(**new** BloomFilterDuplicateRemover(100000)))  
 .thread(10)  
 .run();  
 }  
}

### 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



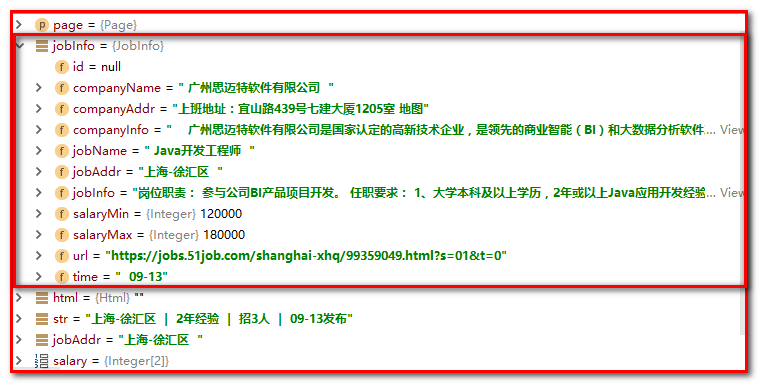
根据薪资（0.6-1万/月）计算薪资范围



实现以下逻辑

*//解析页面，获取招聘详情信息，保存数据***private void** saveJobInfo(Page page) {  
 *//创建招聘详情对象* JobInfo jobInfo = **new** JobInfo();  
  
 *//解析详情页面* Html html = page.getHtml();  
  
 *//获取数据，封装到对象中* jobInfo.setCompanyName(html.css(**"div.cn p.cname a"**,**"text"**).toString());  
 *//System.out.println(html.css("div.bmsg").nodes().get(1).toString());* jobInfo.setCompanyAddr(Jsoup.*parse*(html.css(**"div.bmsg"**).nodes().get(1).toString()).text());  
 *//System.out.println(html.css("div.tmsg").toString());* jobInfo.setCompanyInfo(Jsoup.*parse*(html.css(**"div.tmsg"**).toString()).text());  
 jobInfo.setJobName(html.css(**"div.cn h1"**,**"text"**).toString());  
 */\*\*旧：jobInfo.setJobAddr(html.css("div.cn span.lname","text").toString());\*/* String str = Jsoup.*parse*(html.css(**"div.cn p.msg"**).toString()).text();  
 String jobAddr = str.substring(0,str.indexOf(**"|"**));  
 jobInfo.setJobAddr(jobAddr);  
 jobInfo.setJobInfo(Jsoup.*parse*(html.css(**"div.job\_msg"**).toString()).text());  
 jobInfo.setUrl(page.getUrl().toString());  
  
 *//获取薪资* Integer[] salary = MathSalary.*getSalary*(html.css(**"div.cn strong"**, **"text"**).toString());  
 jobInfo.setSalaryMin(salary[0]);  
 jobInfo.setSalaryMax(salary[1]);  
  
 *//获取发布时间* String time = str.substring(str.indexOf(**"发布"**)-5,str.indexOf(**"发布"**));  
 jobInfo.setTime(time);  
 *//把结果保存起来* page.putField(**"jobInfo"**,jobInfo);  
}

查看断点：



## 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

第一步：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);  
 }  
 }  
}

第二步：在JobProcessor中修改process()启动的逻辑，添加代码

@Autowired  
**private** SpringDataPipeline **springDataPipeline**;  
  
*//initialDelay当任务启动后，等等多久执行方法  
//fixedDelay每个多久执行方法*@Scheduled(initialDelay = 1000, fixedDelay = 100 \* 1000)  
**public void** process() {  
 Spider.*create*(**new** JobProcessor())  
 .addUrl(**url**)  
 .setScheduler(**new** QueueScheduler().setDuplicateRemover(**new** BloomFilterDuplicateRemover(100000)))  
 .thread(10)  
 .addPipeline(**this**.**springDataPipeline**)  
 .run();  
}

测试插入1000条数据