**4. 编写基本的爬虫**

在WebMagic里，实现一个基本的爬虫只需要编写一个类，实现PageProcessor接口即可。这个类基本上包含了抓取一个网站，你需要写的所有代码。

同时这部分还会介绍如何使用WebMagic的抽取API，以及最常见的抓取结果保存的问题。

### 4.1 实现PageProcessor

这部分我们直接通过GithubRepoPageProcessor这个例子来介绍PageProcessor的编写方式。我将PageProcessor的定制分为**三个部分**，分别是爬虫的配置、页面元素的抽取和链接的发现。

public class GithubRepoPageProcessor implements PageProcessor {

// 部分一：抓取网站的相关配置，包括编码、抓取间隔、重试次数等

private Site site = Site.me().setRetryTimes(3).setSleepTime(1000);

@Override

// process是定制爬虫逻辑的核心接口，在这里编写抽取逻辑

public void process(Page page) {

// 部分二：定义如何抽取页面信息，并保存下来

page.putField("author", page.getUrl().regex("https://github\\.com/(\\w+)/.\*").toString());

page.putField("name", page.getHtml().xpath("//h1[@class='entry-title public']/strong/a/text()").toString());

if (page.getResultItems().get("name") == null) {

//skip this page

page.setSkip(true);

}

page.putField("readme", page.getHtml().xpath("//div[@id='readme']/tidyText()"));

// 部分三：从页面发现后续的url地址来抓取

page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/[\\w\\-]+/[\\w\\-]+)").all());

}

@Override

public Site getSite() {

return site;

}

public static void main(String[] args) {

Spider.create(new GithubRepoPageProcessor())

//从"https://github.com/code4craft"开始抓

.addUrl("https://github.com/code4craft")

//开启5个线程抓取

.thread(5)

//启动爬虫

.run();

}

}

#### 4.1.1 爬虫的配置

第一部分关于爬虫的配置，包括编码、抓取间隔、超时时间、重试次数等，也包括一些模拟的参数，例如User Agent、cookie，以及代理的设置，我们会在第5章-“爬虫的配置”里进行介绍。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。

#### 4.1.2 页面元素的抽取

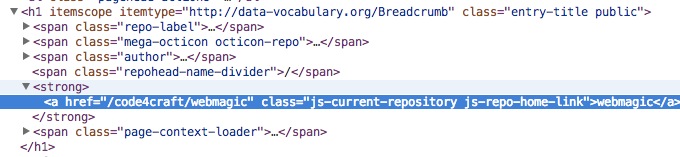
第二部分是爬虫的核心部分：对于下载到的Html页面，你如何从中抽取到你想要的信息？WebMagic里主要使用了三种抽取技术：XPath、正则表达式和CSS选择器。另外，对于JSON格式的内容，可使用JsonPath进行解析。

1. XPath

XPath本来是用于XML中获取元素的一种查询语言，但是用于Html也是比较方便的。例如：

page.getHtml().xpath("//h1[@class='entry-title public']/strong/a/text()")

这段代码使用了XPath，它的意思是“查找所有class属性为'entry-title public'的h1元素，并找到他的strong子节点的a子节点，并提取a节点的文本信息”。 对应的Html是这样子的：



1. CSS选择器

CSS选择器是与XPath类似的语言。如果大家做过前端开发，肯定知道$('h1.entry-title')这种写法的含义。客观的说，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。

1. 正则表达式

正则表达式则是一种通用的文本抽取语言。

page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+/\\w+)").all());

这段代码就用到了正则表达式，它表示匹配所有"[https://github.com/code4craft/webmagic"这样的链接。](https://github.com/code4craft/webmagic%22%E8%BF%99%E6%A0%B7%E7%9A%84%E9%93%BE%E6%8E%A5%E3%80%82)

1. JsonPath

JsonPath是与XPath很类似的一个语言，它用于从Json中快速定位一条内容。WebMagic中使用的JsonPath格式可以参考这里：<https://code.google.com/p/json-path/>

#### 4.1.3 链接的发现

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。

page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+/\\w+)").all());

这段代码的分为两部分，page.getHtml().links().regex("(https://github\\.com/\\w+/\\w+)").all()用于获取所有满足"(https:/ /github\.com/\w+/\w+)"这个正则表达式的链接，page.addTargetRequests()则将这些链接加入到待抓取的队列中去。

### 4.2 使用Selectable抽取元素

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，你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。这个接口包含一些重要的方法，我将它分为两类：抽取部分和获取结果部分。

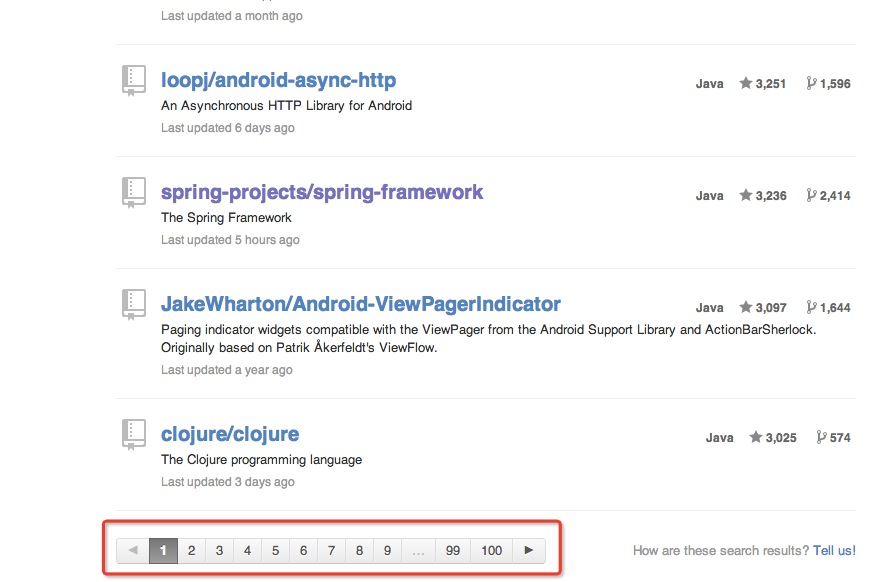
#### 4.2.1 抽取部分API：

| **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("\(.\\*?)\") |
| regex(String regex,int group) | 使用正则表达式抽取，并指定捕获组 | html.regex("\(.\\*?)\",1) |
| replace(String regex, String replacement) | 替换内容 | html.replace("\","") |

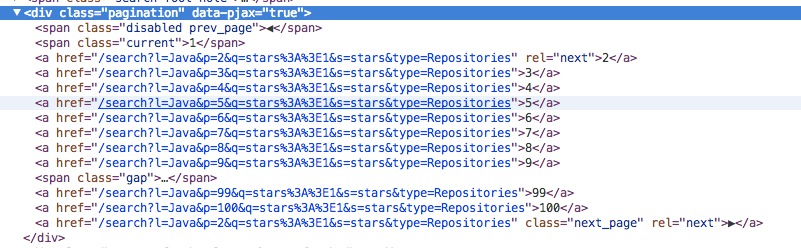
这部分抽取API返回的都是一个Selectable接口，意思是说，抽取是支持链式调用的。下面我用一个实例来讲解链式API的使用。

例如，我现在要抓取github上所有的Java项目，这些项目可以在<https://github.com/search?l=Java&p=1&q=stars%3A%3E1&s=stars&type=Repositories>搜索结果中看到。

为了避免抓取范围太宽，我指定只从分页部分抓取链接。这个抓取规则是比较复杂的，我会要怎么写呢？



首先看到页面的html结构是这个样子的：



那么我可以先用CSS选择器提取出这个div，然后在取到所有的链接。为了保险起见，我再使用正则表达式限定一下提取出的URL的格式，那么最终的写法是这样子的：

List<String> urls = page.getHtml().css("div.pagination").links().regex(".\*/search/\?l=java.\*").all();

然后，我们可以把这些URL加到抓取列表中去：

List<String> urls = page.getHtml().css("div.pagination").links().regex(".\*/search/\?l=java.\*").all();

page.addTargetRequests(urls);

是不是比较简单？除了发现链接，Selectable的链式抽取还可以完成很多工作。我们会在第9章示例中再讲到。

#### 4.2.2 获取结果的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，你可以通过不同的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() | 功能同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() |
| match() | 是否有匹配结果 | if (html.links().match()){ xxx; } |

例如，我们知道页面只会有一条结果，那么可以使用selectable.get()或者selectable.toString()拿到这条结果。

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。

好了，到现在为止，在回过头看看3.1中的GithubRepoPageProcessor，可能就觉得更加清晰了吧？指定main方法，已经可以看到抓取结果在控制台输出了。

### 4.3 使用Pipeline保存结果

好了，爬虫编写完成，现在我们可能还有一个问题：我如果想把抓取的结果保存下来，要怎么做呢？WebMagic用于保存结果的组件叫做Pipeline。例如我们通过“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Json的格式保存下来，怎么做呢？我只需要将Pipeline的实现换成"JsonFilePipeline"就可以了。

public static void main(String[] args) {

Spider.create(new GithubRepoPageProcessor())

//从"https://github.com/code4craft"开始抓

.addUrl("https://github.com/code4craft")

.addPipeline(new JsonFilePipeline("D:\\webmagic\\"))

//开启5个线程抓取

.thread(5)

//启动爬虫

.run();

}

这样子下载下来的文件就会保存在D盘的webmagic目录中了。

通过定制Pipeline，我们还可以实现保存结果到文件、数据库等一系列功能。这个会在第7章“抽取结果的处理”中介绍。

至此为止，我们已经完成了一个基本爬虫的编写，也具有了一些定制功能。

### 4.4 爬虫的配置、启动和终止

#### 4.4.1 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/>") |
| addRequest(Request...) | 添加初始的Request | spider .addRequest("<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() |
| test(String) | 抓取一个页面进行测试 | spider .test("<http://webmagic.io/docs/>") |
| 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>") |

#### 4.4.2 Site

对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)) |

其中循环重试cycleRetry是0.3.0版本加入的机制。

该机制会将下载失败的url重新放入队列尾部重试，直到达到重试次数，以保证不因为某些网络原因漏抓页面。

### 4.5 Jsoup和Xsoup

WebMagic的抽取主要用到了[Jsoup](http://jsoup.org/)和我自己开发的工具[Xsoup](https://github.com/code4craft/xsoup)。

#### 4.5.1 Jsoup

Jsoup是一个简单的HTML解析器，同时它支持使用CSS选择器的方式查找元素。为了开发WebMagic，我对Jsoup的源码进行过详细的分析，具体文章参见[Jsoup学习笔记](https://github.com/code4craft/jsoup-learning)。

#### 4.5.2 Xsoup

[Xsoup](https://github.com/code4craft/xsoup)是我基于Jsoup开发的一款XPath解析器。

之前WebMagic使用的解析器是[HtmlCleaner](http://htmlcleaner.sourceforge.net/)，使用过程存在一些问题。主要问题是XPath出错定位不准确，并且其不太合理的代码结构，也难以进行定制。最终我自己实现了Xsoup，使得更加符合爬虫开发的需要。令人欣喜的是，经过测试，Xsoup的性能比HtmlCleaner要快一倍以上。

Xsoup发展到现在，已经支持爬虫常用的语法，以下是一些已支持的语法对照表：

|  |  |  |
| --- | --- | --- |
| Name | Expression | Support |
| nodename | nodename | yes |
| immediate parent | / | yes |
| parent | // | yes |
| attribute | [@key=value] | yes |
| nth child | tag[n] | yes |
| attribute | /@key | yes |
| wildcard in tagname | /\* | yes |
| wildcard in attribute | /[@\*] | yes |
| function | function() | part |
| or | a | b | yes since 0.2.0 |
| parent in path | . or .. | no |
| predicates | price>35 | no |
| predicates logic | @class=a or @class=b | yes since 0.2.0 |

另外我自己定义了几个对于爬虫来说，很方便的XPath函数。但是请注意，这些函数是标准XPath没有的。

| **Expression** | **Description** | **XPath1.0** |
| --- | --- | --- |
| text(n) | 第n个直接文本子节点，为0表示所有 | text() only |
| allText() | 所有的直接和间接文本子节点 | not support |
| tidyText() | 所有的直接和间接文本子节点，并将一些标签替换为换行，使纯文本显示更整洁 | not support |
| html() | 内部html，不包括标签的html本身 | not support |
| outerHtml() | 内部html，包括标签的html本身 | not support |
| regex(@attr,expr,group) | 这里@attr和group均可选，默认是group0 | not support |

#### 4.5.3 Saxon

Saxon是一个强大的XPath解析器，支持XPath 2.0语法。webmagic-saxon是对Saxon尝试性的一个整合，但是目前看来，XPath 2.0的高级语法，似乎在爬虫开发中使用者并不多。

### 4.6 爬虫的监控

爬虫的监控是0.5.0新增的功能。利用这个功能，你可以查看爬虫的执行情况——已经下载了多少页面、还有多少页面、启动了多少线程等信息。该功能通过JMX实现，你可以使用Jconsole等JMX工具查看本地或者远程的爬虫信息。

如果你完全不会JMX也没关系，因为它的使用相对简单，本章会比较详细的讲解使用方法。如果要弄明白其中原理，你可能需要一些JMX的知识，推荐阅读：[JMX整理](http://my.oschina.net/xpbug/blog/221547)。我很多部分也对这篇文章进行了参考。

注意: 如果你自己定义了Scheduler，那么需要用这个类实现MonitorableScheduler接口，才能查看“LeftPageCount”和“TotalPageCount”这两条信息。

#### 4.6.1 为项目添加监控

添加监控非常简单，获取一个SpiderMonitor的单例SpiderMonitor.instance()，并将你想要监控的Spider注册进去即可。你可以注册多个Spider到SpiderMonitor中。

public class MonitorExample {

public static void main(String[] args) throws Exception {

Spider oschinaSpider = Spider.create(new OschinaBlogPageProcessor())

.addUrl("http://my.oschina.net/flashsword/blog");

Spider githubSpider = Spider.create(new GithubRepoPageProcessor())

.addUrl("https://github.com/code4craft");

SpiderMonitor.instance().register(oschinaSpider);

SpiderMonitor.instance().register(githubSpider);

oschinaSpider.start();

githubSpider.start();

}

}

#### 4.6.2 查看监控信息

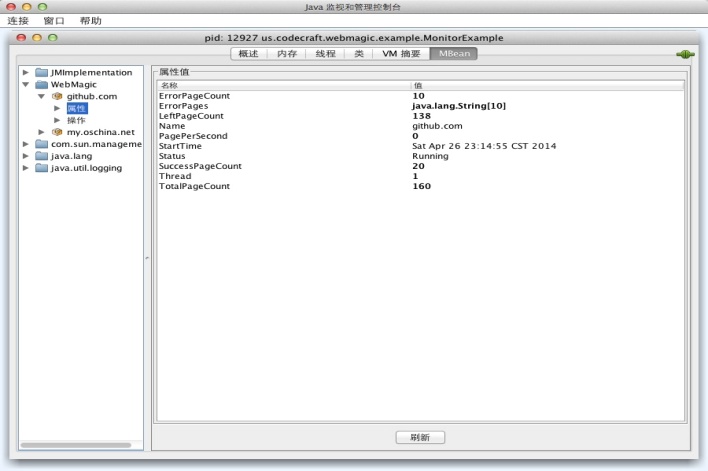
WebMagic的监控使用JMX提供控制，你可以使用任何支持JMX的客户端来进行连接。我们这里以JDK自带的JConsole为例。我们首先启动WebMagic的一个Spider，并添加监控代码。然后我们通过JConsole来进行查看。

我们按照4.6.1的例子启动程序，然后在命令行输入jconsole（windows下是在DOS下输入jconsole.exe）即可启动JConsole。



这里我们选择启动WebMagic的本地进程，连接后选择“MBean”，点开“WebMagic”，就能看到所有已经监控的Spider信息了！

这里我们也可以选择“操作”，在操作里可以选择启动-start()和终止爬虫-stop()，这会直接调用对应Spider的start()和stop()方法，来达到基本控制的目的。



#### 4.6.3 扩展监控接口

除了已有的一些监控信息，如果你有更多的信息需要监控，也可以通过扩展的方式来解决。你可以通过继承SpiderStatusMXBean来实现扩展，具体例子可以看这里： [定制扩展demo](https://github.com/code4craft/webmagic/tree/master/webmagic-extension/src/test/java/us/codecraft/webmagic/monitor)。

**4.7 配置代理**

从0.7.1版本开始，WebMagic开始使用了新的代理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。

代理示例：

1. 设置单一的普通HTTP代理为101.101.101.101的8888端口，并设置密码为"username","password"

HttpClientDownloader httpClientDownloader = new HttpClientDownloader();

httpClientDownloader.setProxyProvider(SimpleProxyProvider.from(new Proxy("101.101.101.101",8888,"username","password")));

spider.setDownloader(httpClientDownloader);

1. 设置代理池，其中包括101.101.101.101和102.102.102.102两个IP，没有密码

HttpClientDownloader httpClientDownloader = new HttpClientDownloader();

httpClientDownloader.setProxyProvider(SimpleProxyProvider.from(

new Proxy("101.101.101.101",8888)

,new Proxy("102.102.102.102",8888)));

如果对于代理部分有建议的，欢迎参与讨论[#579 更多的代理ProxyProvider实现](https://github.com/code4craft/webmagic/issues/579)

### 4.8 处理非HTTP GET请求

一般来说，爬虫只会抓取信息展示类的页面，所以基本只会处理HTTP GET方法的数据。但是对于某些场景，模拟POST等方法也是需要的。

0.7.1版本之后，废弃了老的nameValuePair的写法，采用在Request对象上添加Method和requestBody来实现。

Request request = new Request("http://xxx/path");

request.setMethod(HttpConstant.Method.POST);

request.setRequestBody(HttpRequestBody.json("{'id':1}","utf-8"));

HttpRequestBody内置了几种初始化方式，支持最常见的表单提交、json提交等方式。

| **API** | **说明** |
| --- | --- |
| HttpRequestBody.form(Map\ params, String encoding) | 使用表单提交的方式 |
| HttpRequestBody.json(String json, String encoding) | 使用JSON的方式，json是序列化后的结果 |
| HttpRequestBody.xml(String xml, String encoding) | 设置xml的方式，xml是序列化后的结果 |
| HttpRequestBody.custom(byte[] body, String contentType, String encoding) | 设置自定义的requestBody |

#### POST的去重：

从0.7.1版本开始，POST默认不会去重，详情见：[Issue 484](https://github.com/code4craft/webmagic/issues/484)。如果想要去重可以自己继承DuplicateRemovedScheduler，重写push方法。