**列表+详情的基本页面组合**

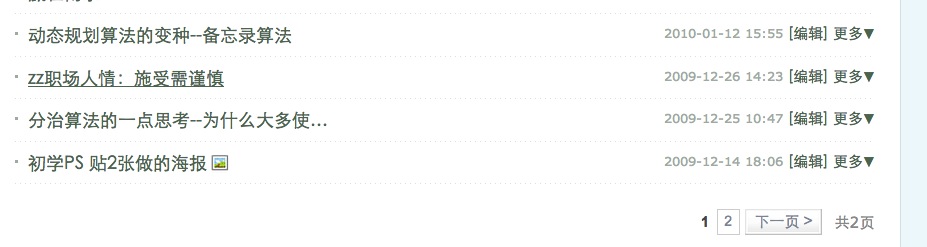
我们先从一个最简单的例子入手。这个例子里，我们有一个列表页，这个列表页以分页的形式展现，我们可以遍历这些分页找到所有目标页面。

**1 示例介绍**

这里我们以作者的新浪博客<http://blog.sina.com.cn/flashsword20>作为例子。在这个例子里，我们要从最终的博客文章页面，抓取博客的标题、内容、日期等信息，也要从列表页抓取博客的链接等信息，从而获取这个博客的所有文章。

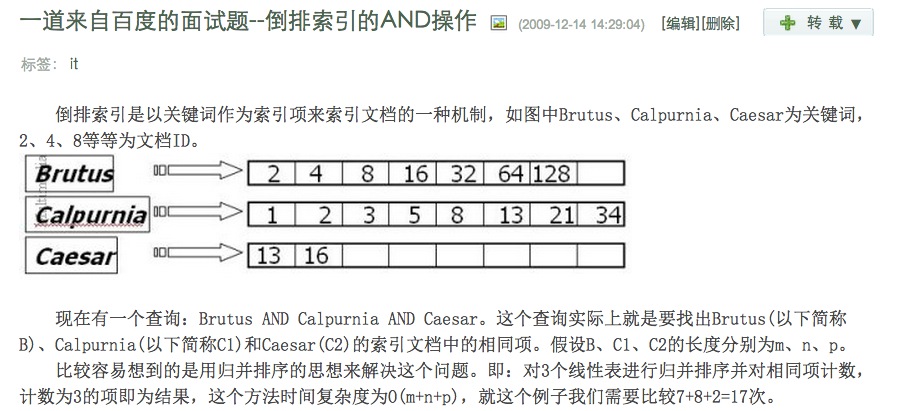
* 列表页

列表页的格式是“[http://blog.sina.com.cn/s/articlelist\_1487828712\_0\_1.html“，](http://blog.sina.com.cn/s/articlelist_1487828712_0_1.html%E2%80%9C%EF%BC%8C) 其中“0\_1”中的“1”是可变的页数。



* 文章页

文章页的格式是“[http://blog.sina.com.cn/s/blog\_58ae76e80100g8au.html”，](http://blog.sina.com.cn/s/blog_58ae76e80100g8au.html%E2%80%9D%EF%BC%8C) 其中“58ae76e80100g8au”是可变的字符。



**2 发现文章URL**

在这个爬虫需求中，文章URL是我们最终关心的，所以如何发现这个博客中所有的文章地址，是爬虫的第一步。

我们可以使用正则表达式http://blog\\.sina\\.com\\.cn/s/blog\_\\w+\\.html对URL进行一次粗略过滤。这里比较复杂的是，这个URL过于宽泛，可能会抓取到其他博客的信息，所以我们必须从列表页中指定的区域获取URL。

在这里，我们使用xpath//div[@class=\\"articleList\\"]选中所有区域，再使用links()或者xpath//a/@href获取所有链接，最后再使用正则表达式http://blog\\.sina\\.com\\.cn/s/blog\_\\w+\\.html，对URL进行过滤，去掉一些“编辑”或者“更多”之类的链接。于是，我们可以这样写：

page.addTargetRequests(page.getHtml().xpath("//div[@class=\"articleList\"]").links().regex("http://blog\\.sina\\.com\\.cn/s/blog\_\\w+\\.html").all());

同时，我们需要把所有找到的列表页也加到待下载的URL中去：

page.addTargetRequests(page.getHtml().links().regex("http://blog\\.sina\\.com\\.cn/s/articlelist\_1487828712\_0\_\\d+\\.html").all());

**3 抽取内容**

文章页面信息的抽取是比较简单的，写好对应的xpath抽取表达式就可以了。

page.putField("title", page.getHtml().xpath("//div[@class='articalTitle']/h2"));

page.putField("content", page.getHtml().xpath("//div[@id='articlebody']//div[@class='articalContent']"));

page.putField("date",

page.getHtml().xpath("//div[@id='articlebody']//span[@class='time SG\_txtc']").regex("\\((.\*)\\)"));

**4 区分列表和目标页**

现在，我们已经定义了对列表和目标页进行处理的方式，现在我们需要在处理时对他们进行区分。在这个例子中，区分方式很简单，因为列表页和目标页在URL格式上是不同的，所以直接用URL区分就可以了！

//列表页

if (page.getUrl().regex(URL\_LIST).match()) {

page.addTargetRequests(page.getHtml().xpath("//div[@class=\"articleList\"]").links().regex(URL\_POST).all());

page.addTargetRequests(page.getHtml().links().regex(URL\_LIST).all());

//文章页

} else {

page.putField("title", page.getHtml().xpath("//div[@class='articalTitle']/h2"));

page.putField("content", page.getHtml().xpath("//div[@id='articlebody']//div[@class='articalContent']"));

page.putField("date",

page.getHtml().xpath("//div[@id='articlebody']//span[@class='time SG\_txtc']").regex("\\((.\*)\\)"));

}

这个例子完整的代码请看[SinaBlogProcessor.java](https://github.com/code4craft/webmagic/blob/master/webmagic-samples/src/main/java/us/codecraft/webmagic/samples/SinaBlogProcessor.java)。

**5 总结**

在这个例子中，我们的主要使用几个方法：

* 从页面指定位置发现链接，使用正则表达式来过滤链接.
* 在PageProcessor中处理两种页面，根据页面URL来区分需要如何处理。

有些朋友反应，用if-else来区分不同处理有些不方便[#issue83](https://github.com/code4craft/webmagic/issues/83)。WebMagic计划在将来的0.5.0版本中，加入[SubPageProcessor](https://github.com/code4craft/webmagic/blob/master/webmagic-extension/src/main/java/us/codecraft/webmagic/handler/SubPageProcessor.java)来解决这个问题。

**抓取前端渲染的页面**

随着AJAX技术不断的普及，以及现在AngularJS这种Single-page application框架的出现，现在js渲染出的页面越来越多。对于爬虫来说，这种页面是比较讨厌的：仅仅提取HTML内容，往往无法拿到有效的信息。那么如何处理这种页面呢？总的来说有两种做法：

1. 在抓取阶段，在爬虫中内置一个浏览器内核，执行js渲染页面后，再抓取。这方面对应的工具有Selenium、HtmlUnit或者PhantomJs。但是这些工具都存在一定的效率问题，同时也不是那么稳定。好处是编写规则同静态页面一样。
2. 因为js渲染页面的数据也是从后端拿到，而且基本上都是AJAX获取，所以分析AJAX请求，找到对应数据的请求，也是比较可行的做法。而且相对于页面样式，这种接口变化可能性更小。缺点就是找到这个请求，并进行模拟，是一个相对困难的过程，也需要相对多的分析经验。

对比两种方式，我的观点是，对于一次性或者小规模的需求，用第一种方式省时省力。但是对于长期性的、大规模的需求，还是第二种会更靠谱一些。对于一些站点，甚至还有一些js混淆的技术，这个时候，第一种的方式基本是万能的，而第二种就会很复杂了。

对于第一种方法，webmagic-selenium就是这样的一个尝试，它定义了一个Downloader，在下载页面时，就是用浏览器内核进行渲染。selenium的配置比较复杂，而且跟平台和版本有关，没有太稳定的方案。感兴趣的可以看我这篇博客：[使用Selenium来抓取动态加载的页面](http://my.oschina.net/flashsword/blog/147334)

这里我主要介绍第二种方法，希望到最后你会发现：原来解析一个前端渲染的页面，也没有那么复杂。这里我们以AngularJS中文社区<http://angularjs.cn/>为例。

**1 如何判断前端渲染**

判断页面是否为js渲染的方式比较简单，在浏览器中直接查看源码（Windows下Ctrl+U，Mac下command+alt+u），如果找不到有效的信息，则基本可以肯定为js渲染。





这个例子中，在页面中的标题“有孚计算机网络-前端攻城师”在源码中无法找到，则可以断定是js渲染，并且这个数据是AJAX得到。

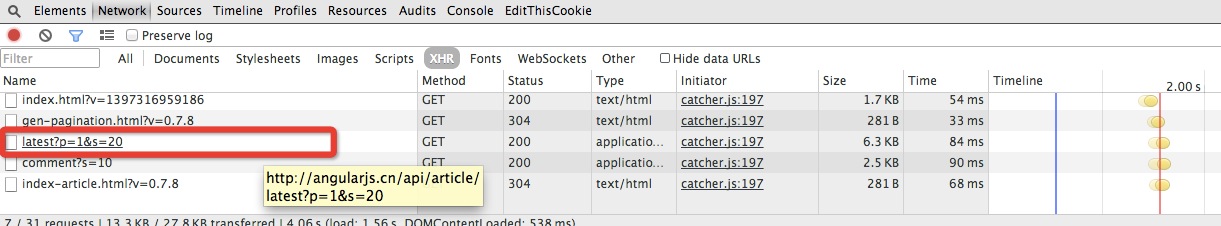
**2 分析请求**

下面我们进入最难的一部分：找到这个数据请求。这一步能帮助我们的工具，主要是浏览器中查看网络请求的开发者工具。

以Chome为例，我们打开“开发者工具”（Windows下是F12，Mac下是command+alt+i），然后重新刷新页面（也有可能是下拉页面，总之是所有你认为可能触发新数据的操作），然后记得保留现场，把请求一个个拿来分析吧！

这一步需要一点耐心，但是也并不是无章可循。首先能帮助我们的是上方的分类筛选（All、Document等选项）。如果是正常的AJAX，在XHR标签下会显示，而JSONP请求会在Scripts标签下，这是两个比较常见的数据类型。

然后你可以根据数据大小来判断一下，一般结果体积较大的更有可能是返回数据的接口。剩下的，基本靠经验了，例如这里这个"latest?p=1&s=20"一看就很可疑…



对于可疑的地址，这时候可以看一下响应体是什么内容了。这里在开发者工具看不清楚，我们把URLhttp://angularjs.cn/api/article/latest?p=1&s=20复制到地址栏，重新请求一次（如果用Chrome推荐装个jsonviewer，查看AJAX结果很方便）。查看结果，看来我们找到了想要的。



同样的办法，我们进入到帖子详情页，找到了具体内容的请求：http://angularjs.cn/api/article/A0y2。

**3 编写程序**

回想一下之前列表+目标页的例子，会发现我们这次的需求，跟之前是类似的，只不过换成了AJAX方式-AJAX方式的列表，AJAX方式的数据，而返回数据变成了JSON。那么，我们仍然可以用上次的方式，分为两种页面来进行编写：

1. 数据列表

在这个列表页，我们需要找到有效的信息，来帮助我们构建目标AJAX的URL。这里我们看到，这个\_id应该就是我们想要的帖子的id，而帖子的详情请求，就是由一些固定URL加上这个id组成。所以在这一步，我们自己手动构造URL，并加入到待抓取队列中。这里我们使用JsonPath这种选择语言来选择数据（webmagic-extension包中提供了JsonPathSelector来支持它）。

if (page.getUrl().regex(LIST\_URL).match()) {

//这里我们使用JSONPATH这种选择语言来选择数据

List<String> ids = new JsonPathSelector("$.data[\*].\_id").selectList(page.getRawText());

if (CollectionUtils.isNotEmpty(ids)) {

for (String id : ids) {

page.addTargetRequest("http://angularjs.cn/api/article/"+id);

}

}

}

1. 目标数据

有了URL，实际上解析目标数据就非常简单了，因为JSON数据是完全结构化的，所以省去了我们分析页面，编写XPath的过程。这里我们依然使用JsonPath来获取标题和内容。

page.putField("title", new JsonPathSelector("$.data.title").select(page.getRawText()));

page.putField("content", new JsonPathSelector("$.data.content").select(page.getRawText()));

这个例子完整的代码请看[AngularJSProcessor.java](https://github.com/code4craft/webmagic/blob/master/webmagic-samples/src/main/java/us/codecraft/webmagic/samples/AngularJSProcessor.java)

**4 总结**

在这个例子中，我们分析了一个比较经典的动态页面的抓取过程。实际上，动态页面抓取，最大的区别在于：它提高了链接发现的难度。我们对比一下两种开发模式：

1. 后端渲染的页面

下载辅助页面=>发现链接=>下载并分析目标HTML

1. 前端渲染的页面

发现辅助数据=>构造链接=>下载并分析目标AJAX

对于不同的站点，这个辅助数据可能是在页面HTML中已经预先输出，也可能是通过AJAX去请求，甚至可能是多次数据请求的过程，但是这个模式基本是固定的。

但是这些数据请求的分析比起页面分析来说，仍然是要复杂得多，所以这其实是动态页面抓取的难点。

本节这个例子希望做到的是，在分析出请求后，为这类爬虫的编写提供一个可遵循的模式，即发现辅助数据=>构造链接=>下载并分析目标AJAX这个模式。

PS:

WebMagic 0.5.0之后会将Json的支持增加到链式API中，以后你可以使用：

page.getJson().jsonPath("$.name").get();

这样的方式来解析AJAX请求了。

同时也支持

page.getJson().removePadding("callback").jsonPath("$.name").get();

这样的方式来解析JSONP请求。