

# 目錄

Introduction	1.1
1. WebMagic概览	1.2
1.1 设计思想	1.2.1
1.2 总体架构	1.2.2
1.3 项目组成	1.2.3
2. 快速开始	1.3
2.1 使用Maven	1.3.1
2.2 不使用Maven	1.3.2
2.3 第一个爬虫项目	1.3.3
3. 下载和编译源码	1.4
3.1 下载源码	1.4.1
3.2 导入项目	1.4.2
3.3 编译和执行源码	1.4.3
4. 编写基本的爬虫	1.5
4.1 实现PageProcessor	1.5.1
4.2 使用Selectable抽取元素	1.5.2
4.3 使用Pipeline保存结果	1.5.3
4.4 爬虫的配置、启动和终止	1.5.4
4.5 Jsoup与Xsoup	1.5.5
4.6 爬虫的监控	1.5.6
4.7 配置代理	1.5.7
4.8 处理POST请求	1.5.8
5. 使用注解编写爬虫	1.6
5.1 编写Model类	1.6.1
5.2 TargetUrl与HelpUrl	1.6.2
5.3 使用ExtractBy进行抽取	1.6.3
5.4 在类上使用ExtractBy	1.6.4
5.5 结果的类型转换	1.6.5

---

5.6 一个完整的流程	1.6.6
5.7 AfterExtractor	1.6.7
6. 组件的使用和定制	1.7
6.1 使用和定制Pipeline	1.7.1
6.2 使用和定制Scheduler	1.7.2
6.3 使用和定制Downloader	1.7.3
附录：实例分析	1.8
列表+详情的基本页面组合	1.8.1
抓取前端渲染的页面	1.8.2

---

# WebMagic in Action

Little book of WebMagic.



[WebMagic](#)是我业余开发的一款简单灵活的爬虫框架。基于它你可以很容易的编写一个爬虫。

这本小书以WebMagic入手，一方面讲解WebMagic的使用方式，另一方面讲解爬虫开发的一些惯用方案。

文章预览请点<http://webmagic.io/docs/>，页面基于[gitbook](#)进行构建。

本文档遵循CC-BYNC协议。

# 1. WebMagic概览

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的架构设计参照了Scrapy，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。

另外WebMagic还包括一些外围扩展和一个正在开发的产品化项目 `webmagic-avalon`。

## 1.1 设计思想



### 1. 一个框架，一个领域

一个好的框架必然凝聚了领域知识。WebMagic的设计参考了业界最优秀的爬虫Scrapy，而实现则应用了HttpClient、Jsoup等Java世界最成熟的工具，目标就是做一个Java语言Web爬虫的教科书般的实现。

如果你是爬虫开发老手，那么WebMagic会非常容易上手，它几乎使用Java原生的开发方式，只不过提供了一些模块化的约束，封装一些繁琐的操作，并且提供了一些便捷的功能。

如果你是爬虫开发新手，那么使用并了解WebMagic会让你了解爬虫开发的常用模式、工具链、以及一些问题的处理方式。熟练使用之后，相信自己从头开发一个爬虫也不是什么难事。

因为这个目标，WebMagic的核心非常简单——在这里，功能性是要给简单性让步的。

### 2. 微内核和高可扩展性

WebMagic由四个组件(Downloader、PageProcessor、Scheduler、Pipeline)构成，核心代码非常简单，主要是将这些组件结合并完成多线程的任务。这意味着，在WebMagic中，你基本上可以对爬虫的功能做任何定制。

WebMagic的核心在webmagic-core包中，其他的包你可以理解为对WebMagic的一个扩展——这和作为用户编写一个扩展是没有什么区别的。

### 3. 注重实用性

虽然核心需要足够简单，但是WebMagic也以扩展的方式，实现了很多可以帮助开发的便捷功能。例如基于注解模式的爬虫开发，以及扩展了XPath语法的Xsoup等。这些功能在WebMagic中是可选的，它们的开发目标，就是让使用者开发爬虫尽可能的简单，尽可能

## 1.1 设计思想

---

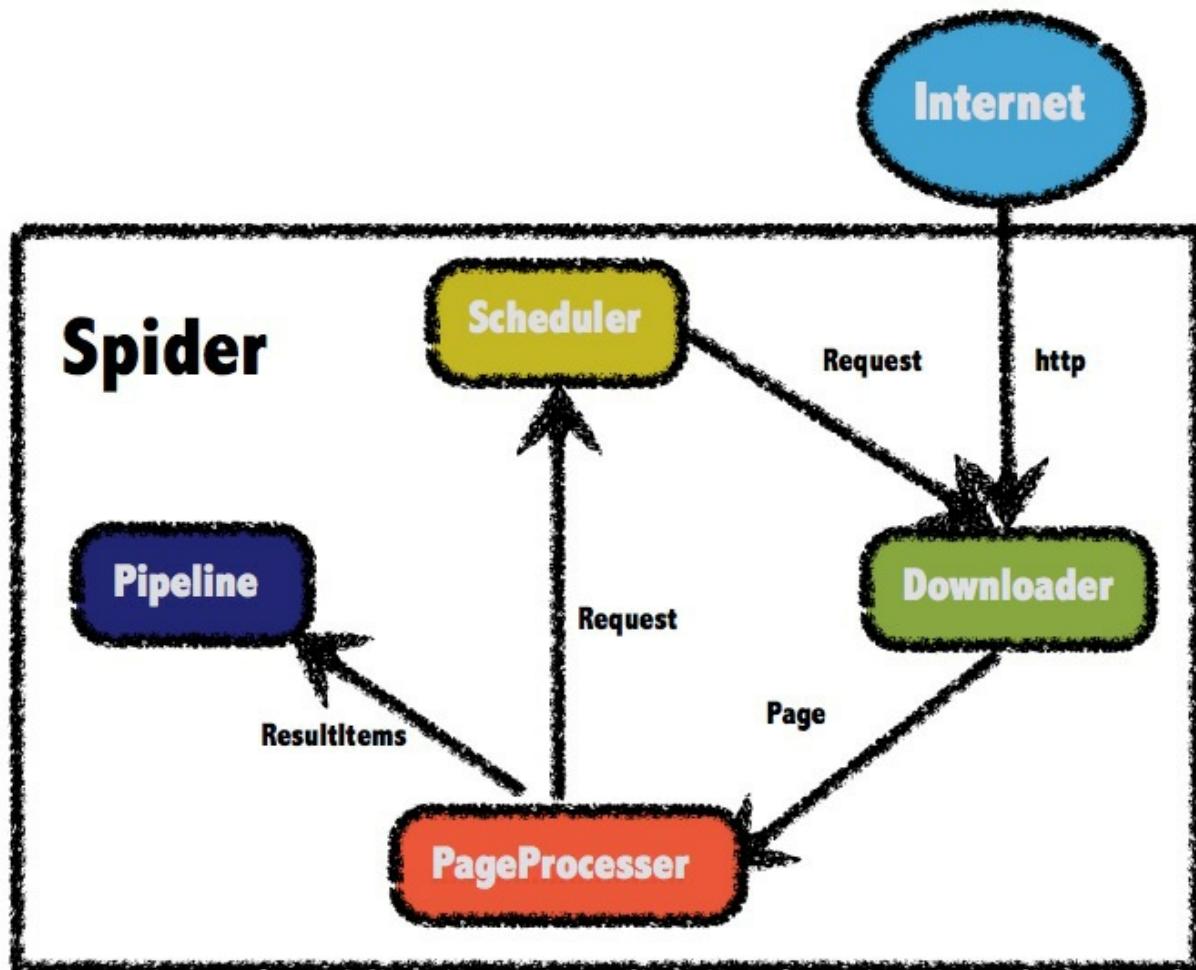
的易维护。

## 1.2 总体架构

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Scrapy，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：



### 1.2.1 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。

除非项目有一些特殊的分布式需求，否则无需自己定制Scheduler。

## 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。

### 1.2.2 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。在第四章的例子中，我们会详细介绍它的使用。

### 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。

#### 1.2.3 控制爬虫运转的引擎--Spider

`Spider`是WebMagic内部流程的核心。`Downloader`、`PageProcessor`、`Scheduler`、`Pipeline`都是`Spider`的一个属性，这些属性是可以自由设置的，通过设置这个属性可以实现不同的功能。`Spider`也是WebMagic操作的入口，它封装了爬虫的创建、启动、停止、多线程等功能。下面是一个设置各个组件，并且设置多线程和启动的例子。详细的`Spider`设置请看[第四章——爬虫的配置、启动和终止](#)。

```
public static void main(String[] args) {
    Spider.create(new GithubRepoPageProcessor())
        //从https://github.com/code4craft开始抓
        .addUrl("https://github.com/code4craft")
        //设置Scheduler，使用Redis来管理URL队列
        .setScheduler(new RedisScheduler("localhost"))
        //设置Pipeline，将结果以json方式保存到文件
        .addPipeline(new JsonFilePipeline("D:\\data\\webmagic"))
        //开启5个线程同时执行
        .thread(5)
        //启动爬虫
        .run();
}
```

#### 1.2.4 快速上手

上面介绍了很多组件，但是其实使用者需要关心的没有那么多，因为大部分模块WebMagic已经提供了默认实现。

一般来说，对于编写一个爬虫，`PageProcessor` 是需要编写的部分，而 `Spider` 则是创建和控制爬虫的入口。在第四章中，我们会介绍如何通过定制`PageProcessor`来编写一个爬虫，并通过`Spider`来启动。

## 1.3 项目组成

WebMagic项目代码包括几个部分，在根目录下以不同目录名分开。它们都是独立的Maven项目。

### 1.3.1 主要部分

WebMagic主要包括两个包，这两个包经过广泛实用，已经比较成熟：

#### **webmagic-core**

`webmagic-core` 是WebMagic核心部分，只包含爬虫基本模块和基本抽取器。WebMagic-core的目标是成为网页爬虫的一个教科书般的实现。

#### **webmagic-extension**

`webmagic-extension` 是WebMagic的主要扩展模块，提供一些更方便的编写爬虫的工具。包括注解格式定义爬虫、JSON、分布式等支持。

### 1.3.2 外围功能

除此之外，WebMagic项目里还有几个包，这些都是一些实验性的功能，目的只是提供一些与外围工具整合的样例。因为精力有限，这些包没有经过广泛的使用和测试，推荐使用方式是自行下载源码，遇到问题后再修改。

#### **webmagic-samples**

这里是作者早期编写的一些爬虫的例子。因为时间有限，这些例子有些使用的仍然是老版本的API，也可能有一些因为目标页面的结构变化不再可用。最新的、精选过的例子，请看`webmagic-core`的 `us.codecraft.webmagic.processor.example` 包和`webmagic-core`的 `us.codecraft.webmagic.example` 包。

#### **webmagic-scripts**

WebMagic对于爬虫规则脚本化的一些尝试，目标是让开发者脱离Java语言，来进行简单、快速的开发。同时强调脚本的共享。

目前项目因为感兴趣的用户不多，处于搁置状态，对脚本化感兴趣的可以看这里：[webmagic-scripts简单文档](#)

## webmagic-selenium

WebMagic与Selenium结合的模块。Selenium是一个模拟浏览器进行页面渲染的工具，WebMagic依赖Selenium进行动态页面的抓取。

## webmagic-saxon

WebMagic与Saxon结合的模块。Saxon是一个XPath、XSLT的解析工具，webmagic依赖Saxon来进行XPath2.0语法解析支持。

### 1.3.3 webmagic-avalon

`webmagic-avalon` 是一个特殊的项目，它想基于WebMagic实现一个产品化的工具，涵盖爬虫的创建、爬虫的管理等后台工具。[Avalon](#)是亚瑟王传说中的“理想之岛”，`webmagic-avalon` 的目标是提供一个通用的爬虫产品，达到这个目标绝非易事，所以取名也有一点“理想”的意味，但是作者一直在朝这个目标努力。

对这个项目感兴趣的可以看[这里](#)[WebMagic-Avalon项目计划](#)。

## 2. 快速开始

WebMagic主要包含两个jar包：`webmagic-core-{version}.jar` 和 `webmagic-extension-{version}.jar`。在项目中添加这两个包的依赖，即可使用WebMagic。

WebMagic默认使用Maven管理依赖，但是你也可以不依赖Maven进行使用。

## 2.1 使用Maven

WebMagic基于Maven进行构建，推荐使用Maven来安装WebMagic。在你自己的项目（已有项目或者新建一个）中添加以下坐标即可：

```
<dependency>
    <groupId>us.codecraft</groupId>
    <artifactId>webmagic-core</artifactId>
    <version>0.6.0</version>
</dependency>
<dependency>
    <groupId>us.codecraft</groupId>
    <artifactId>webmagic-extension</artifactId>
    <version>0.6.0</version>
</dependency>
```

如果你对Maven使用还不熟悉，推荐看看[@黄勇的博客：《Maven 那点事儿》](#)。

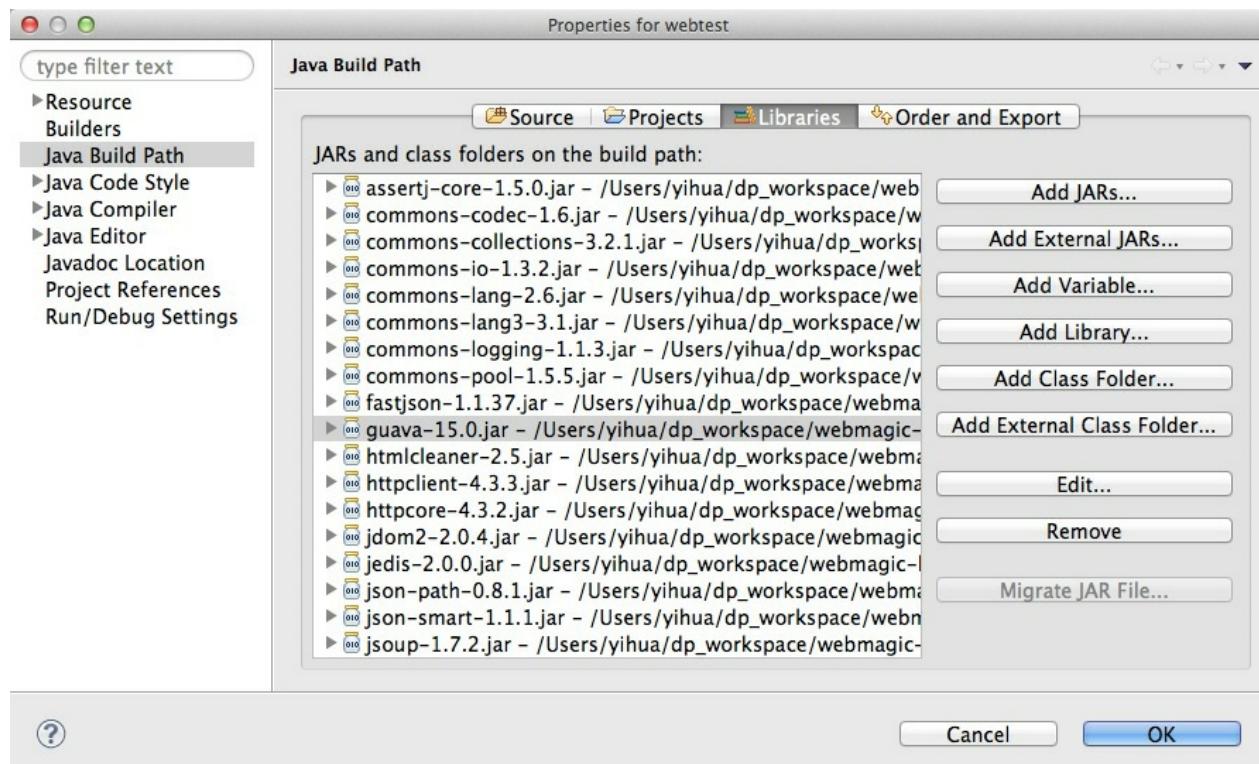
WebMagic使用slf4j-log4j12作为slf4j的实现.如果你自己定制了slf4j的实现，请在项目中去掉此依赖。

```
<dependency>
    <groupId>us.codecraft</groupId>
    <artifactId>webmagic-extension</artifactId>
    <version>0.6.0</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

## 2.2 不使用Maven

不使用maven的用户，可以去<http://webmagic.io>中下载最新的jar包。我将所有依赖的jar都打包在了一起，点[这里](#)下载。

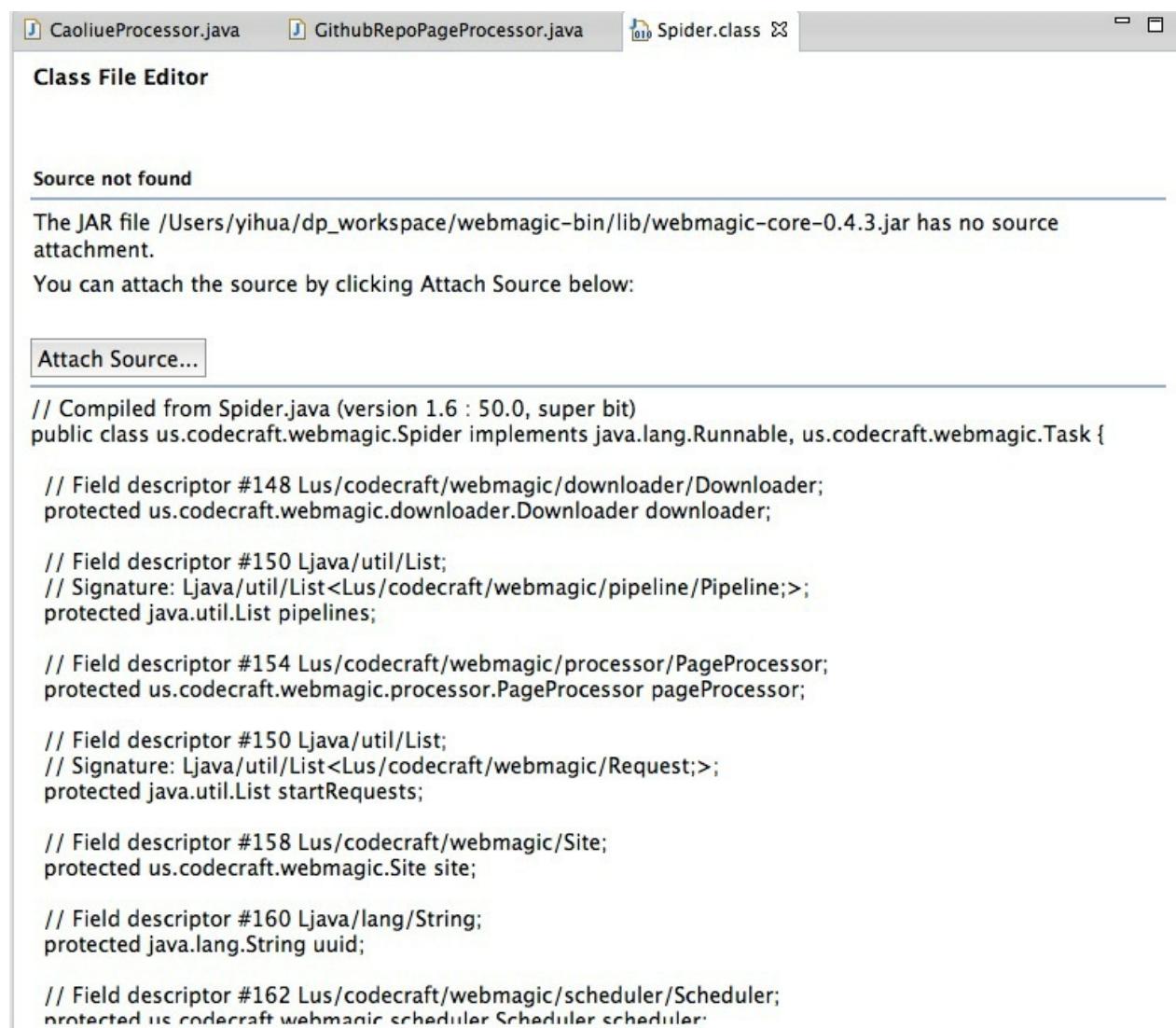
下载之后进行解压，然后在项目中import即可。



因为WebMagic提倡自己定制，所以项目的源码还是有必要看的。

在<http://webmagic.io>上，你可以下载最新的 `webmagic-core-{version}-sources.jar` 和 `webmagic-extension-{version}-sources.jar`，点击"Attach Source"即可。

## 2.2 不使用Maven



The JAR file /Users/yihua/dp\_workspace/webmagic-bin/lib/webmagic-core-0.4.3.jar has no source attachment.

You can attach the source by clicking Attach Source below:

[Attach Source...](#)

```
// Compiled from Spider.java (version 1.6 : 50.0, super bit)
public class us.codecraft.webmagic.Spider implements java.lang.Runnable, us.codecraft.webmagic.Task {

    // Field descriptor #148 Lus/codecraft/webmagic/downloader/Downloader;
    protected us.codecraft.webmagic.downloader.Downloader downloader;

    // Field descriptor #150 Ljava/util/List;
    // Signature: Ljava/util/List<Lus/codecraft/webmagic/pipeline/Pipeline;>;
    protected java.util.List pipelines;

    // Field descriptor #154 Lus/codecraft/webmagic/processor/PageProcessor;
    protected us.codecraft.webmagic.processor.PageProcessor pageProcessor;

    // Field descriptor #150 Ljava/util/List;
    // Signature: Ljava/util/List<Lus/codecraft/webmagic/Request;>;
    protected java.util.List startRequests;

    // Field descriptor #158 Lus/codecraft/webmagic/Site;
    protected us.codecraft.webmagic.Site site;

    // Field descriptor #160 Ljava/lang/String;
    protected java.lang.String uuid;

    // Field descriptor #162 Lus/codecraft/webmagic/scheduler/Scheduler;
    protected us.codecraft.webmagic.scheduler.Scheduler scheduler;
}
```

## 2.3 第一个爬虫项目

在你的项目中添加了WebMagic的依赖之后，即可开始第一个爬虫的开发了！我们这里拿一个抓取Github信息的例子：

```

import us.codecraft.webmagic.Page;
import us.codecraft.webmagic.Site;
import us.codecraft.webmagic.Spider;
import us.codecraft.webmagic.processor.PageProcessor;

public class GithubRepoPageProcessor implements PageProcessor {

    private Site site = Site.me().setRetryTimes(3).setSleepTime(100);

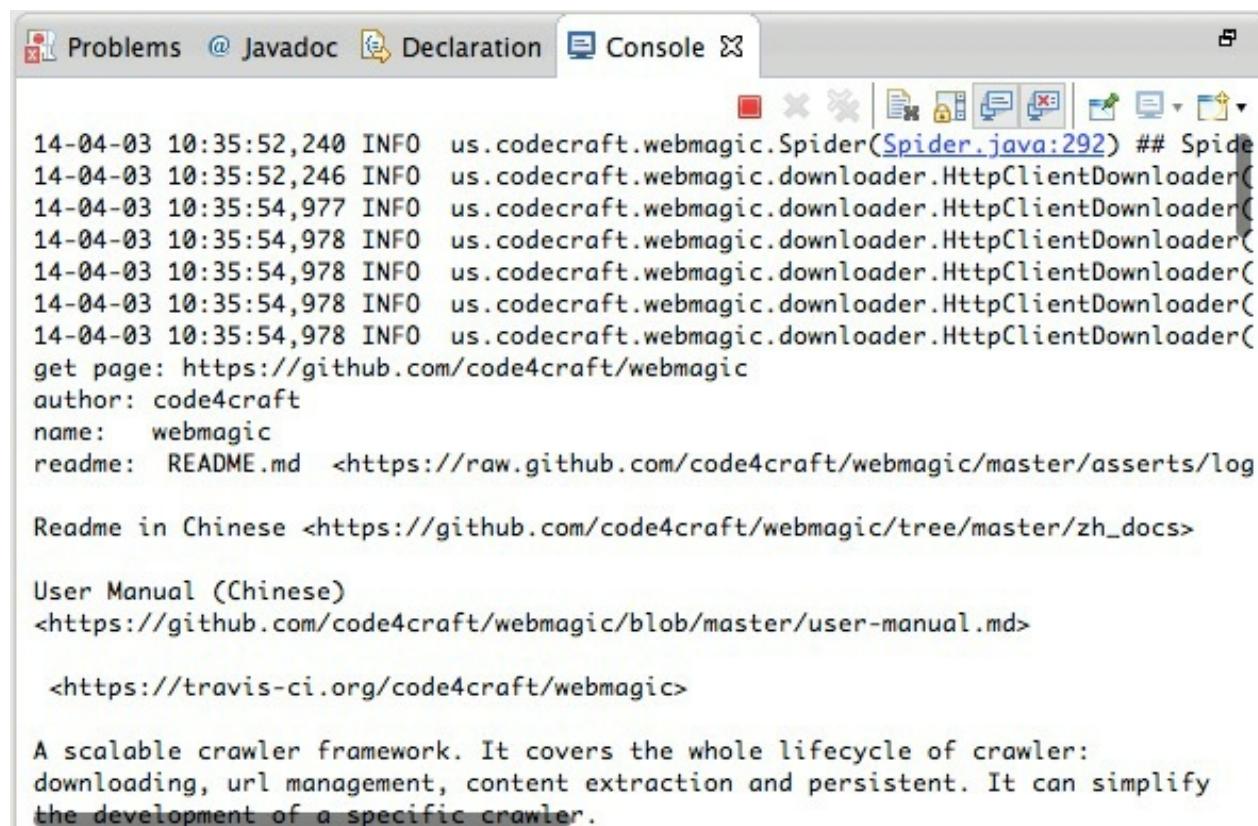
    @Override
    public void process(Page page) {
        page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+/\w+\\.\\w+)").all());
        page.putField("author", page.getUrl().regex("https://github\\.com/(\\w+)/.*").toString());
        page.putField("name", page.getHtml().xpath("//h1[@class='entry-title 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()"));
    }

    @Override
    public Site getSite() {
        return site;
    }

    public static void main(String[] args) {
        Spider.create(new GithubRepoPageProcessor()).addUrl("https://github.com/code4craf"
t).thread(5).run();
    }
}

```

点击main方法，选择“运行”，你会发现爬虫已经可以正常工作了！



The screenshot shows a Java IDE interface with the 'Console' tab selected. The console window displays the following log output:

```
14-04-03 10:35:52,240 INFO us.codecraft.webmagic.Spider(Spider.java:292) ## Spider
14-04-03 10:35:52,246 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,977 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,978 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,978 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,978 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,978 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
get page: https://github.com/code4craft/webmagic
author: code4craft
name: webmagic
readme: README.md <https://raw.github.com/code4craft/webmagic/master/assets/log

Readme in Chinese <https://github.com/code4craft/webmagic/tree/master/zh_docs>

User Manual (Chinese)
<https://github.com/code4craft/webmagic/blob/master/user-manual.md>

<https://travis-ci.org/code4craft/webmagic>

A scalable crawler framework. It covers the whole lifecycle of crawler:
downloading, url management, content extraction and persistent. It can simplify
the development of a specific crawler.
```

## 3. 下载和编译源码

如果你对WebMagic的源码感兴趣，那么可以选择源码下载和编译的方式来使用WebMagic。“非常简单的二次开发”也是WebMagic的目标之一。

WebMagic是一个纯Java项目，如果你熟悉Maven，那么下载并编译源码是非常简单的。如果不熟悉Maven也没关系，这部分会介绍如何在Eclipse里导入这个项目。

## 3.1 下载源码

WebMagic目前有两个仓库：

- <https://github.com/code4craft/webmagic>

github上的仓库保存最新版本，所有issue、pull request都在这里。大家觉得项目不错的话别忘了去给个star哦！

- <http://git.oschina.net/flashsword20/webmagic>

此仓库包含所有编译好的依赖包，只保存项目的稳定版本，最新版本仍在github上更新。oschina在国内比较稳定，主要作为镜像。

无论在哪个仓库，使用

```
git clone https://github.com/code4craft/webmagic.git
```

或者

```
git clone http://git.oschina.net/flashsword20/webmagic.git
```

即可下载最新代码。

如果你对git本身使用也不熟悉，建议看看@黄勇的 [从 Git OSC 下载 Smart 源码](#)

## 3.2 导入项目

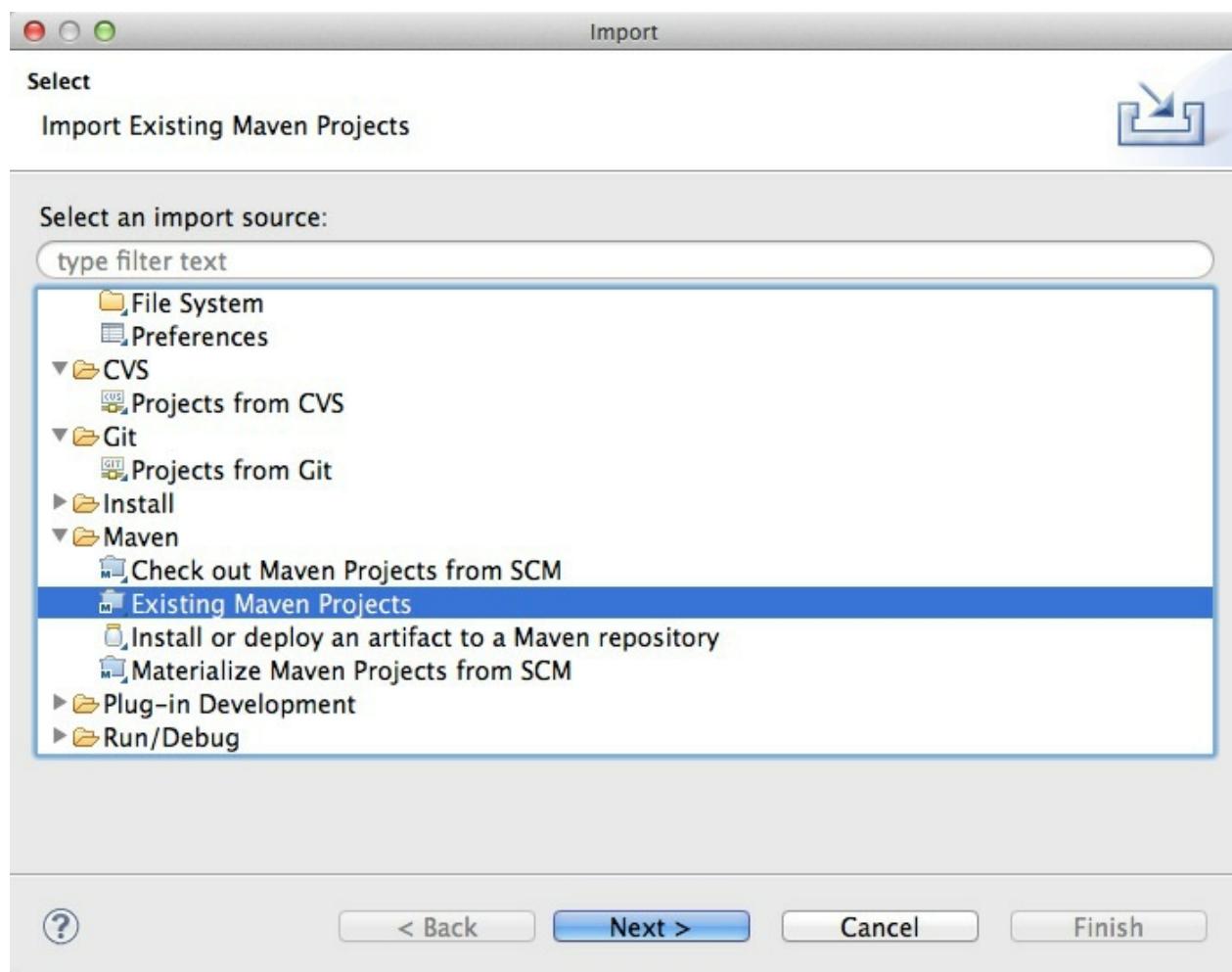
IntelliJ Idea默认自带Maven支持，import项目时选择Maven项目即可。

### 3.2.1 使用m2e插件

使用Eclipse的用户，推荐安装m2e插件，安装地

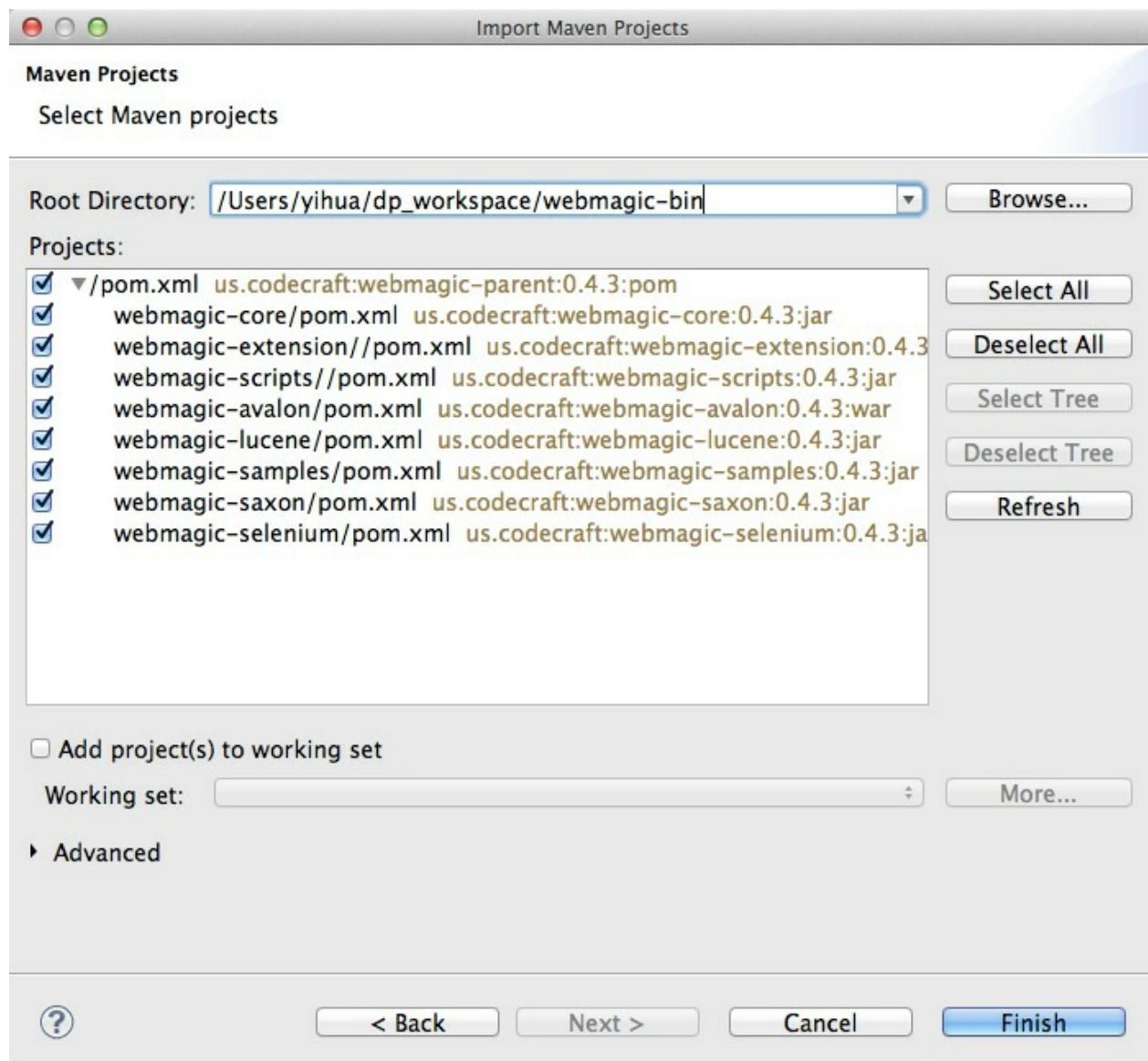
址：<https://www.eclipse.org/m2e/download/>(<https://www.eclipse.org/m2e/download/>)

安装后，在File->Import中选择Maven->Existing Maven Projects即可导入项目。



导入后看到项目选择界面，点击finish即可。

### 3.2 导入项目



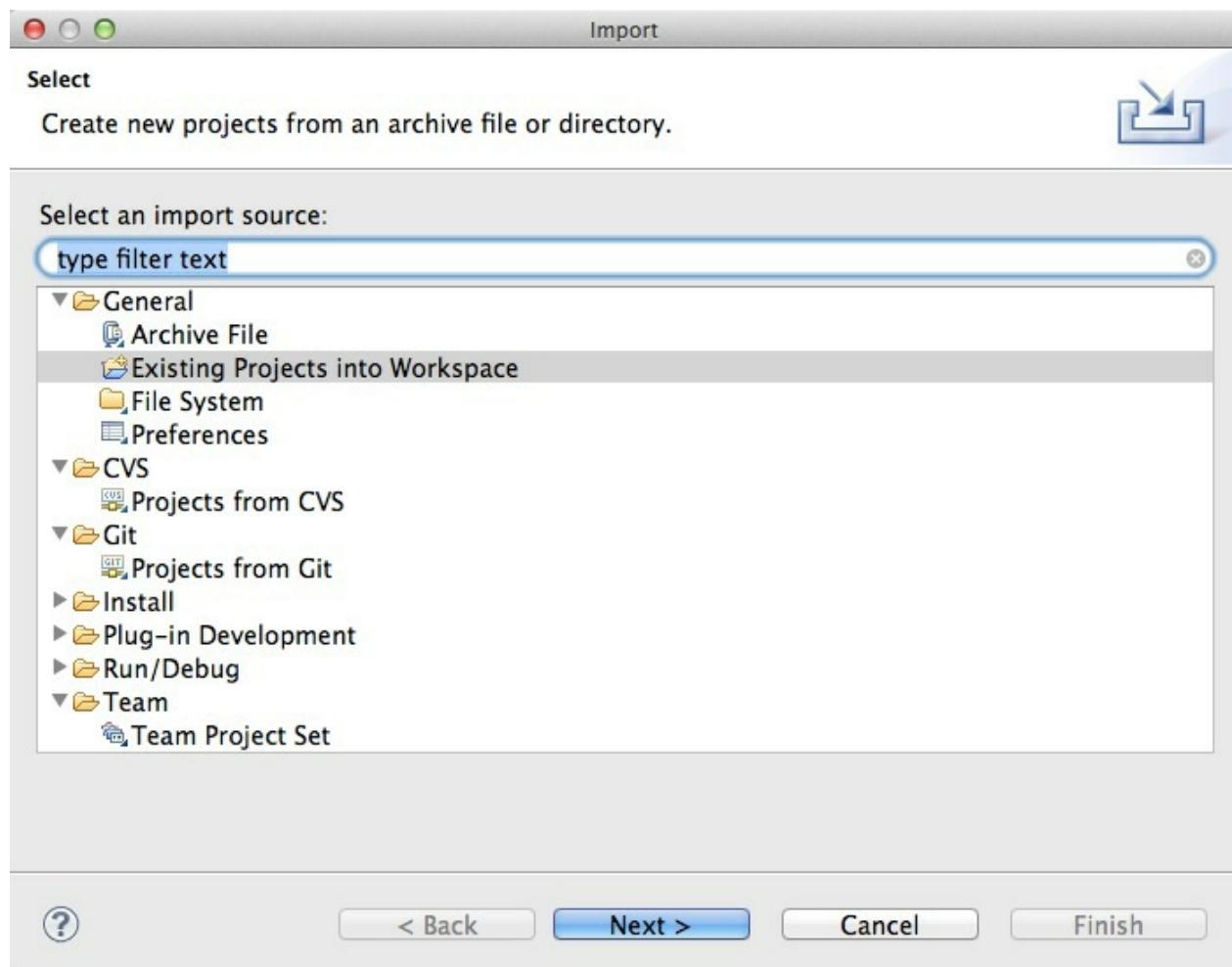
### 3.2.2 使用Maven Eclipse插件

如果没有安装m2e插件，只要你安装了Maven，也是比较好办的。在项目根目录下使用命令：

```
mvn eclipse:eclipse
```

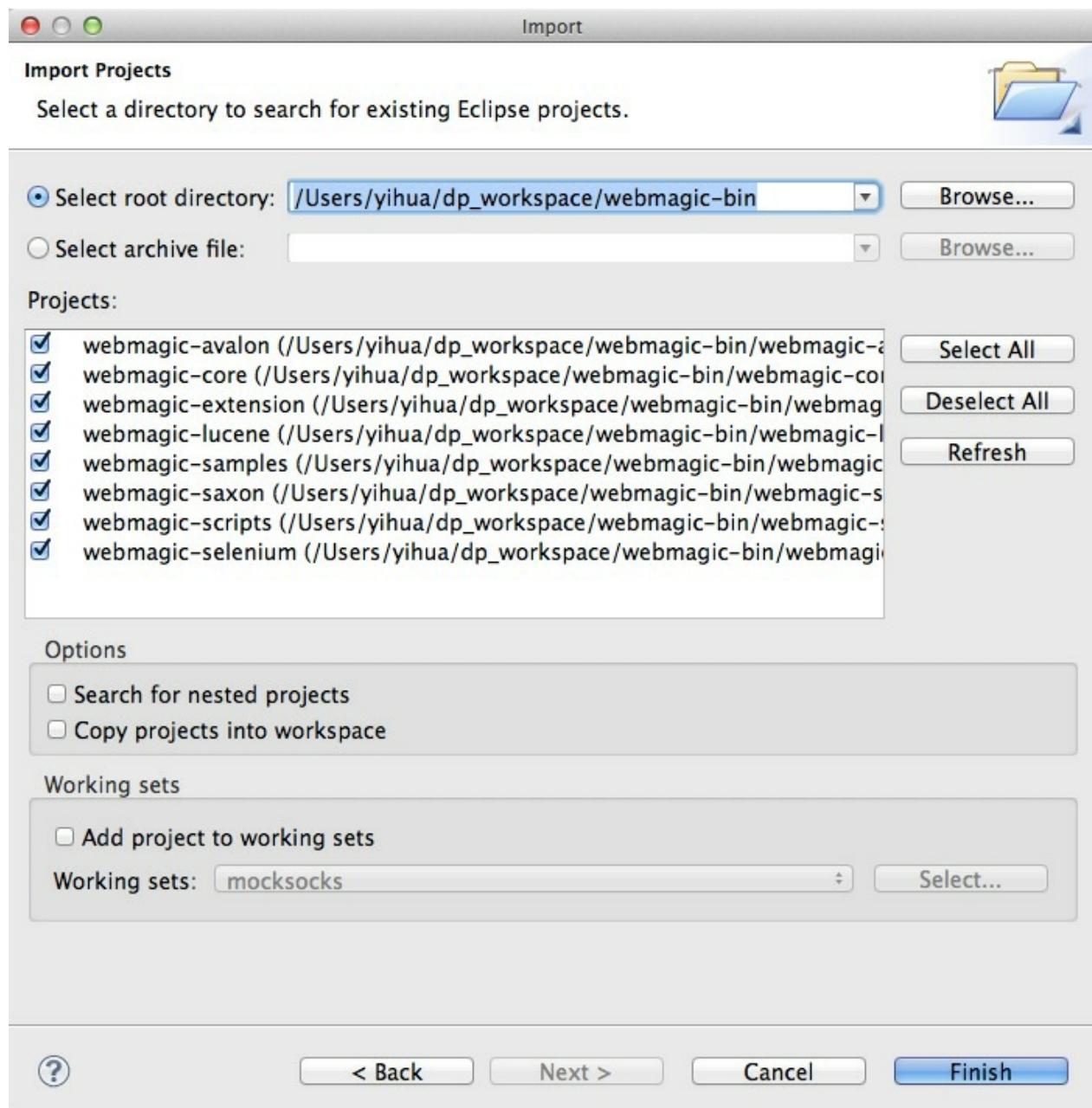
生成maven项目结构的eclipse配置文件，然后在File->Import中选择General->Existing Projects into Workspace即可导入项目。

### 3.2 导入项目



导入后看到项目选择界面，点击finish即可。

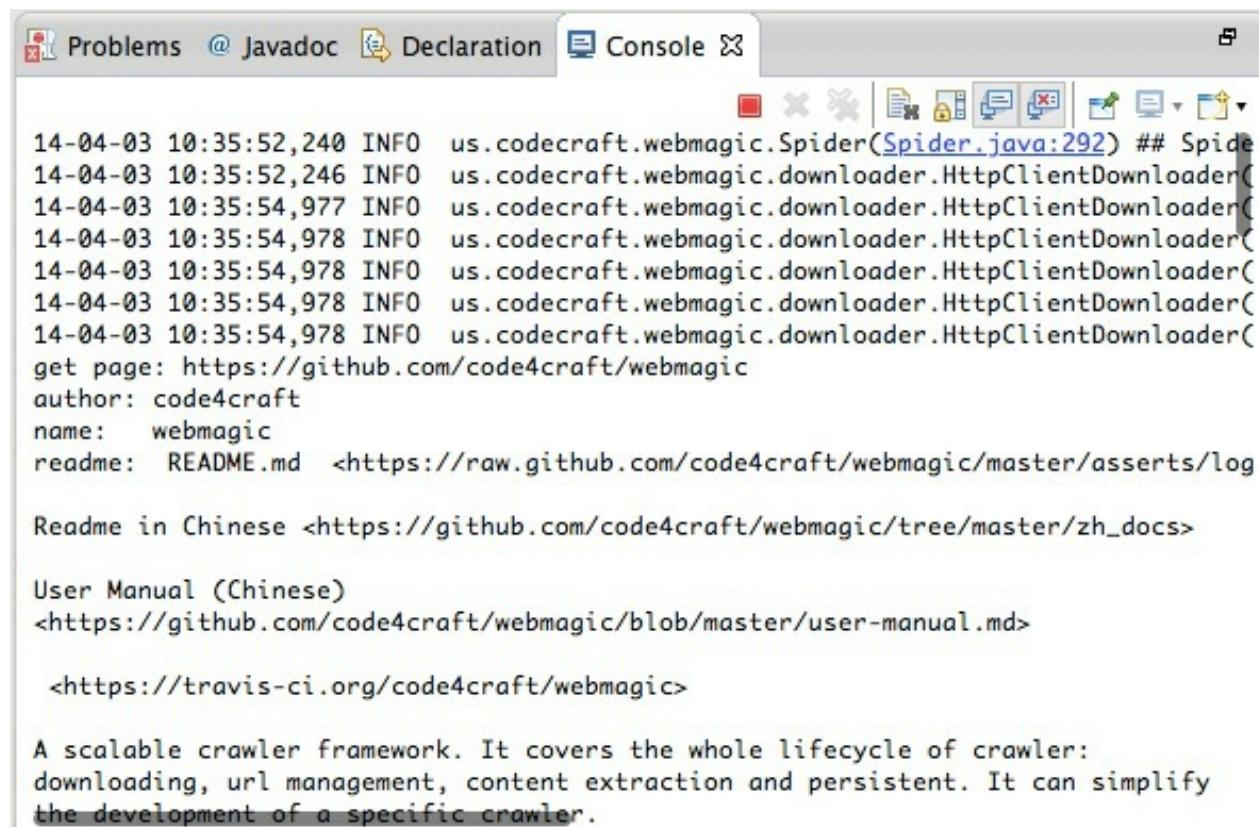
## 3.2 导入项目



### 3.3 编译和执行源码

导入成功之后，应该就没有编译错误了！此时你可以运行一下webmagic-core项目中自带的exmaple:"us.codecraft.webmagic.processor.example.GithubRepoPageProcessor"。

同样，看到控制台输出如下，则表示源码编译和执行成功了！



The screenshot shows a Java IDE's console tab with the following log output:

```
14-04-03 10:35:52,240 INFO us.codecraft.webmagic.Spider(Spider.java:292) ## Spider
14-04-03 10:35:52,246 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,977 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,978 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,978 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,978 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
14-04-03 10:35:54,978 INFO us.codecraft.webmagic.downloader.HttpClientDownloader(
get page: https://github.com/code4craft/webmagic
author: code4craft
name: webmagic
readme: README.md <https://raw.github.com/code4craft/webmagic/master/asserts/log

Readme in Chinese <https://github.com/code4craft/webmagic/tree/master/zh_docs>

User Manual (Chinese)
<https://github.com/code4craft/webmagic/blob/master/user-manual.md>

<https://travis-ci.org/code4craft/webmagic>

A scalable crawler framework. It covers the whole lifecycle of crawler:
downloading, url management, content extraction and persistent. It can simplify
the development of a specific crawler.
```

## 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：

```
-----  
▼ <h1 itemscope itemtype="http://data-vocabulary.org/Breadcrumb" class="entry-title public">  
  ▶ <span class="repo-label">...</span>  
  ▶ <span class="mega-octicon octicon-repo">...</span>  
  ▶ <span class="author">...</span>  
  <span class="repohead-name-divider">/</span>  
  ▼ <strong>  
    <a href="/code4craft/webmagic" class="js-current-repository js-repo-home-link">webmagic</a>  
  </strong>  
  ▶ <span class="page-context-loader">...</span>  
</h1>
```

#### 2. 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。

#### 3. 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。

### 4. JsonPath

JsonPath是于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
<code>xpath(String xpath)</code>	使用XPath选择	<code>html.xpath("//div[@class='title']")</code>
<code>\$(String selector)</code>	使用Css选择器选择	<code>html.\$("div.title")</code>
<code>\$(String selector, String attr)</code>	使用Css选择器选择	<code>html.\$("div.title", "text")</code>
<code>css(String selector)</code>	功能同 <code>\$()</code> ，使用 Css选择器选择	<code>html.css("div.title")</code>
<code>links()</code>	选择所有链接	<code>html.links()</code>
<code>regex(String regex)</code>	使用正则表达式抽取	<code>html.regex("(.*?)")</code>
<code>regex(String regex, int group)</code>	使用正则表达式抽取，并指定捕获组	<code>html.regex("(.*?)", 1)</code>
<code>replace(String regex, String replacement)</code>	替换内容	<code>html.replace("\", "")</code>

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?q=language%3AJava&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？

## 4.2 使用Selectable抽取元素

Last updated a month ago

---

 **loopj/android-async-http** Java ★ 3,251 ⚡ 1,596  
An Asynchronous HTTP Library for Android  
Last updated 6 days ago

---

 **spring-projects/spring-framework** Java ★ 3,236 ⚡ 2,414  
The Spring Framework  
Last updated 5 hours ago

---

 **JakeWharton/Android-ViewPagerIndicator** Java ★ 3,097 ⚡ 1,644  
Paging indicator widgets compatible with the ViewPager from the Android Support Library and ActionBarSherlock.  
Originally based on Patrik Åkerfeldt's ViewFlow.  
Last updated a year ago

---

 **clojure/clojure** Java ★ 3,025 ⚡ 574  
The Clojure programming language  
Last updated 3 days ago

---

◀ 1 2 3 4 5 6 7 8 9 ... 99 100 ▶

How are these search results? [Tell us!](#)

首先看到页面的html结构是这个样子的：

```
<div class="pagination" data-pjax="true">
  <span class="disabled prev_page"><</span>
  <span class="current">1</span>
  <a href="/search?l=Java&p=2&q=stars%3A%3E1&s=stars&type=Repositories" rel="next">2</a>
  <a href="/search?l=Java&p=3&q=stars%3A%3E1&s=stars&type=Repositories">3</a>
  <a href="/search?l=Java&p=4&q=stars%3A%3E1&s=stars&type=Repositories">4</a>
  <a href="/search?l=Java&p=5&q=stars%3A%3E1&s=stars&type=Repositories">5</a>
  <a href="/search?l=Java&p=6&q=stars%3A%3E1&s=stars&type=Repositories">6</a>
  <a href="/search?l=Java&p=7&q=stars%3A%3E1&s=stars&type=Repositories">7</a>
  <a href="/search?l=Java&p=8&q=stars%3A%3E1&s=stars&type=Repositories">8</a>
  <a href="/search?l=Java&p=9&q=stars%3A%3E1&s=stars&type=Repositories">9</a>
  <span class="gap">...</span>
  <a href="/search?l=Java&p=99&q=stars%3A%3E1&s=stars&type=Repositories">99</a>
  <a href="/search?l=Java&p=100&q=stars%3A%3E1&s=stars&type=Repositories">100</a>
  <a href="/search?l=Java&p=2&q=stars%3A%3E1&s=stars&type=Repositories" class="next_page" rel="next">▶</a>
</div>
```

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=jav
a.*").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=jav
a.*").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
<code>create(PageProcessor)</code>	创建 <code>Spider</code>	<code>Spider.create(new GithubRepoProcessor())</code>
<code>addUrl(String...)</code>	添加初始的 URL	<code>spider.addUrl("http://webmagic.io/doc")</code>
<code>addRequest(Request...)</code>	添加初始的 Request	<code>spider.addRequest("http://webmagic.io/doc")</code>
<code>thread(n)</code>	开启 n 个线程	<code>spider.thread(5)</code>
<code>run()</code>	启动，会阻塞当前线程执行	<code>spider.run()</code>
<code>start()/runAsync()</code>	异步启动，当前线程继续执行	<code>spider.start()</code>
<code>stop()</code>	停止爬虫	<code>spider.stop()</code>
<code>test(String)</code>	抓取一个页面进行测试	<code>spider.test("http://webmagic.io/doc")</code>
<code>addPipeline(Pipeline)</code>	添加一个 Pipeline，一个 <code>Spider</code> 可以有多个 Pipeline	<code>spider.addPipeline(new ConsolePipeline())</code>
<code>setScheduler(Scheduler)</code>	设置 Scheduler，一个 <code>Spider</code> 只能有个一个 Scheduler	<code>spider.setScheduler(new RedisScheduler())</code>
<code>setDownloader(Downloader)</code>	设置 Downloader，一个 <code>Spider</code> 只能有个一个 Downloader	<code>spider.setDownloader(new SeleniumDownloader())</code>

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", "c")
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")
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](#)和我自己开发的工具[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](#)。

### 4.5.2 Xsoup

[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](#)，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，这些函数式标准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](#)。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 {

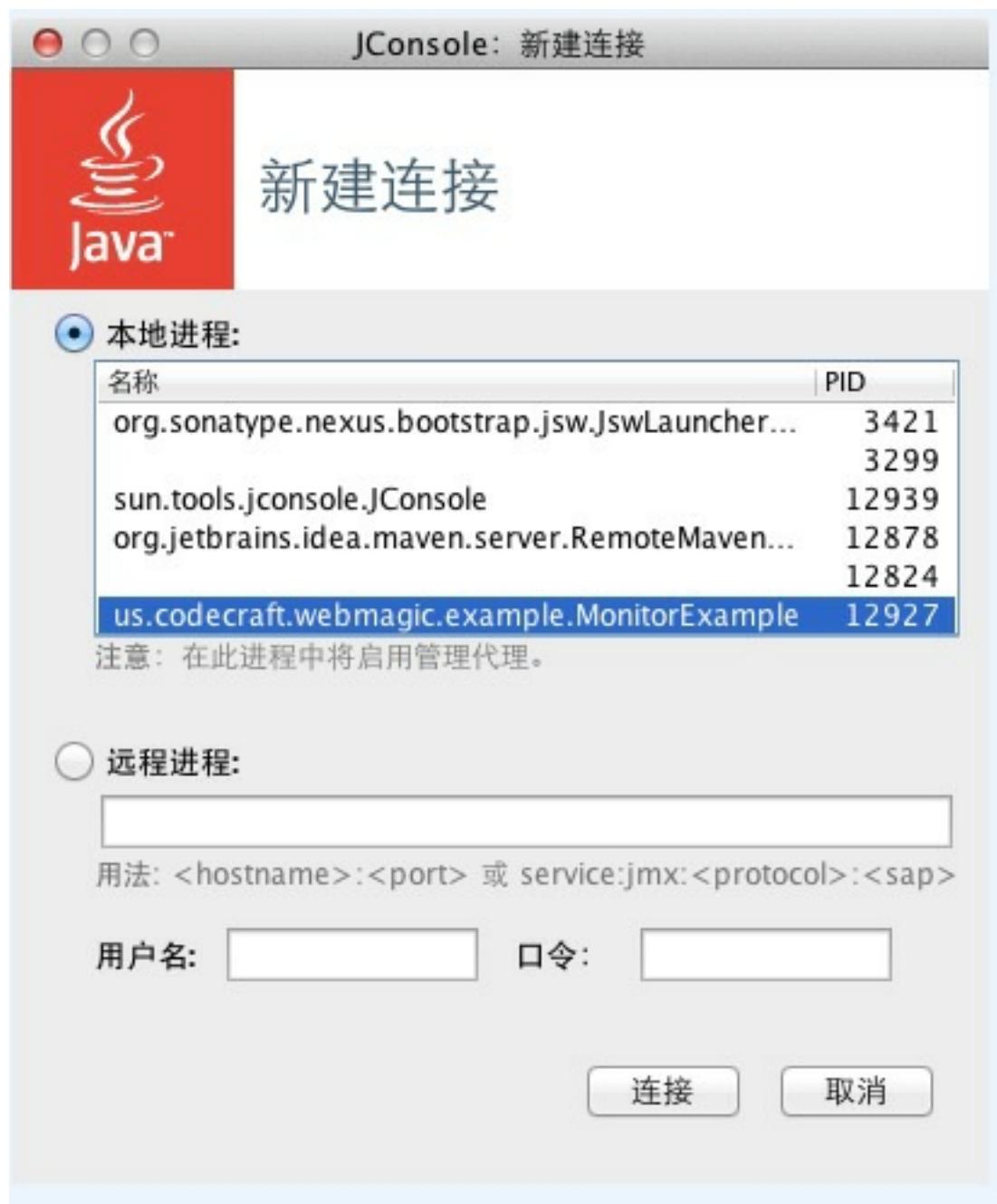
    public static void main(String[] args) throws Exception {
        Spider oschinaSpider = Spider.create(new OschinaBlogPageProcessor())
            .addUrl("http://my.oschina.net/flashsword/blog");
        Spider githubSpider = Spider.create(new GithubRepoPageProcessor())
            .addUrl("https://github.com/code4craft");

        SpiderMonitor.instance().register(oschinaSpider);
        SpiderMonitor.instance().register(githubSpider);
        oschinaSpider.start();
        githubSpider.start();
    }
}
```

### 4.6.2 查看监控信息

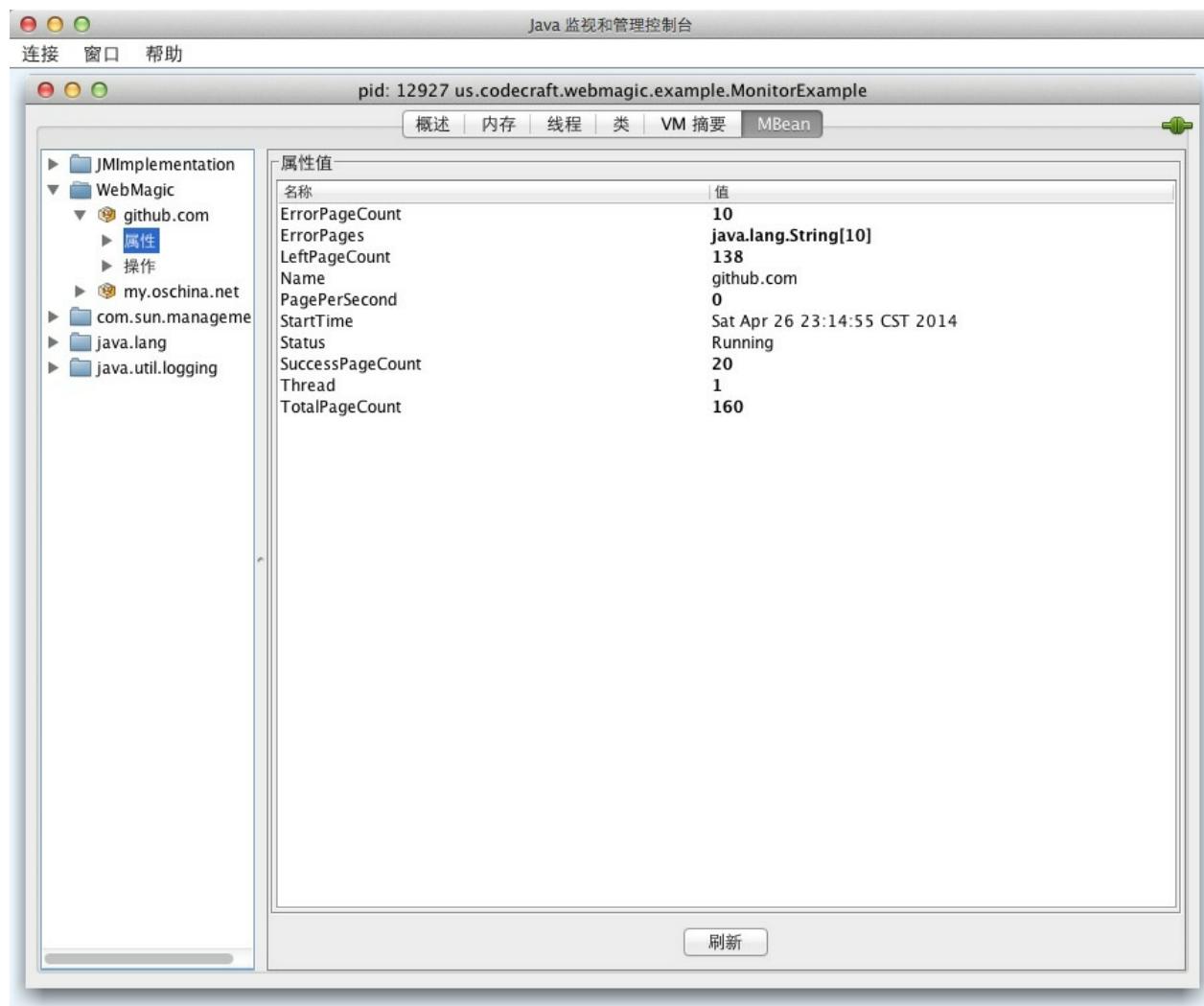
WebMagic的监控使用JMX提供控制，你可以使用任何支持JMX的客户端来进行连接。我们这里以JDK自带的JConsole为例。我们首先启动WebMagic的一个Spider，并添加监控代码。然后我们通过JConsole来进行查看。

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](#)。

## 5. 使用注解编写爬虫

WebMagic支持使用独有的注解风格编写一个爬虫，引入webmagic-extension包即可使用此功能。

在注解模式下，使用一个简单对象加上注解，可以用极少的代码量就完成一个爬虫的编写。对于简单的爬虫，这样写既简单又容易理解，并且管理起来也很方便。这也是WebMagic的一大特色，我戏称它为 OEM (Object/Extraction Mapping)。

注解模式的开发方式是这样的：

1. 首先定义你需要抽取的数据，并编写类。
2. 在类上写明 `@TargetUrl` 注解，定义对哪些URL进行下载和抽取。
3. 在类的字段上加上 `@ExtractBy` 注解，定义这个字段使用什么方式进行抽取。
4. 定义结果的存储方式。

下面我们仍然以第四章中github的例子，来编写一个同样功能的爬虫，来讲解注解功能的使用。最终编写好的爬虫是这样子的，是不是更加简单？

```
@TargetUrl("https://github.com/\w+/\w+")
@HelpUrl("https://github.com/\w+")
public class GithubRepo {

    @ExtractBy(value = "//h1[@class='entry-title public']/strong/a/text()", notNull = true)
    private String name;

    @ExtractByUrl("https://github\\.com/(\w+)/.*")
    private String author;

    @ExtractBy("//div[@id='readme']/tidyText()")
    private String readme;

    public static void main(String[] args) {
        OoSpider.create(Site.me().setSleepTime(1000)
            , new ConsolePageModelPipeline(), GithubRepo.class)
            .addUrl("https://github.com/code4craft").thread(5).run();
    }
}
```

## 5.1 编写Model类

同第四章的例子一样，我们这里抽取一个github项目的名称、作者和简介三个信息，所以我们定义了一个Model类。

```
public class GithubRepo {  
  
    private String name;  
  
    private String author;  
  
    private String readme;  
  
}
```

这里省略了getter和setter方法。

在抽取最后，我们会得到这个类的一个或者多个实例，这就是爬虫的结果。

## 5.2 TargetUrl与HelpUrl

在第二步，我们仍然要定义如何发现URL。这里我们要先引入两个概念：`@TargetUrl` 和 `@HelpUrl`。

### 5.2.1 TargetUrl与HelpUrl

`HelpUrl/TargetUrl` 是一个非常有效的爬虫开发模式，`TargetUrl`是我们最终要抓取的 URL，最终想要的数据都来自这里；而`HelpUrl`则是为了发现这个最终URL，我们需要访问的页面。几乎所有垂直爬虫的需求，都可以归结为对这两类URL的处理：

- 对于博客页，`HelpUrl`是列表页，`TargetUrl`是文章页。
- 对于论坛，`HelpUrl`是帖子列表，`TargetUrl`是帖子详情。
- 对于电商网站，`HelpUrl`是分类列表，`TargetUrl`是商品详情。

在这个例子中，`TargetUrl`是最终的项目页，而`HelpUrl`则是项目搜索页，它会展示所有项目的链接。

有了这些知识，我们就为这个例子定义URL格式：

```
@TargetUrl("https://github.com/\w+/\w+")
@HelpUrl("https://github.com/\w+")
public class GithubRepo {
    .....
}
```

#### TargetUrl中的自定义正则表达式

这里我们使用的是正则表达式来规定URL范围。可能细心的朋友，会知道`.`是正则表达式的保留字符，那么这里是不是写错了呢？其实是这里为了方便，WebMagic自己定制的适合URL的正则表达式，主要由两点改动：

- 将URL中常用的字符`.`默认做了转义，变成了`\.`
- 将`"*"`替换成了`"*"`，直接使用可表示通配符。

例如，`https://github.com/*` 在这里是一个合法的表达式，它表示`https://github.com/`下的所有URL。

在WebMagic中，从`TargetUrl` 页面得到的URL，只要符合`TargetUrl`的格式，也是会被下载的。所以即使不指定`HelpUrl`也是可以的——例如某些博客页总会有“下一篇”链接，这种情况下无需指定`HelpUrl`。

#### sourceRegion

## 5.2 TargetUrl与HelpUrl

---

TargetUrl还支持定义 `sourceRegion`，这个参数是一个XPath表达式，指定了这个URL从哪里得到——不在sourceRegion的URL不会被抽取。

## 5.3 使用ExtractBy进行抽取

`@ExtractBy` 是一个用于抽取元素的注解，它描述了一种抽取规则。

### 5.3.1 初识ExtractBy注解

`@ExtractBy`注解主要作用于字段，它表示“使用这个抽取规则，将抽取到的结果保存到这个字段中”。例如：

```
@ExtractBy("//div[@id='readme']/text()")
private String readme;
```

这里`//div[@id='readme']/text()`是一个XPath表示的抽取规则，而抽取到的结果则会保存到`readme`字段中。

### 5.3.2 使用其他抽取方式

除了XPath，我们还可以使用其他抽取方式进行抽取，包括CSS选择器、正则表达式和JsonPath，在注解中指明 `type` 之后即可。

```
@ExtractBy(value = "div.BlogContent", type = ExtractBy.Type.Css)
private String content;
```

### 5.3.3 notnull

`@ExtractBy`包含一个 `notNull` 属性，如果熟悉mysql的同学一定能明白它的意思：此字段不允许为空。如果为空，这条抽取到的结果会被丢弃。对于一些页面的关键性属性（例如文章的标题等），设置 `notnull` 为 `true`，可以有效的过滤掉无用的页面。

`notNull` 默认为 `false`。

### 5.3.4 multi（已废弃）

`multi` 是一个boolean属性，它表示这条抽取规则是对应多条记录还是单条记录。对应的，这个字段必须为 `java.util.List` 类型。在0.4.3之后，当字段为List类型时，这个属性会自动为true，无须再设置。

- 0.4.3以前

```
@ExtractBy(value = "//div[@class='BlogTags']/a/text()", multi = true)  
private List<String> tags;
```

- 0.4.3及以后

```
@ExtractBy("//div[@class='BlogTags']/a/text()")  
private List<String> tags;
```

### 5.3.5 ComboExtract（已废弃）

`@ComboExtract` 是一个比较复杂的注解，它可以将多个抽取规则进行组合，组合方式包括"AND/OR"两种方式。

在WebMagic 0.4.3版本中使用了Xsoup 0.2.0版本。在这个版本，XPath支持的语法大大加强了，不但支持XPath和正则表达式组合使用，还支持“|”进行或运算。所以作者认为，`ComboExtract` 这种复杂的组合方式，已经不再需要了。

- XPath与正则表达式组合

```
@ExtractBy("//div[@class='BlogStat']/regex('\\\\d+-\\\\d+-\\\\d+\\\\s+\\\\d+:\\\\d+')")  
private Date date;
```

- XPath的取或

```
@ExtractBy("//div[@id='title']/text() | //title/text()")  
private String title;
```

### 5.3.6 ExtractByUrl

`@ExtractByUrl` 是一个单独的注解，它的意思是“从URL中进行抽取”。它只支持正则表达式作为抽取规则。

## 5.4 在类上使用ExtractBy

在之前的注解模式中，我们一个页面只对应一条结果。如果一个页面有多个抽取的记录呢？例如在“QQ美食”的列表页面<http://meishi.qq.com/beijing/c/all>，我想要抽取所有商户名和优惠信息，该怎么办呢？

在类上使用 `@ExtractBy` 注解可以解决这个问题。

在类上使用这个注解的意思很简单：使用这个结果抽取一个区域，让这块区域对应一个结果。

```
@ExtractBy(value = "//ul[@id=\"promos_list2\"]/li",multi = true)
public class QQMeishi {
    .....
}
```

对应的，在这个类中的字段上再使用 `@ExtractBy` 的话，则是从这个区域而不是整个页面进行抽取。如果这个时候仍想要从整个页面抽取，则可以设置 `source = RawHtml`。

```
@TargetUrl("http://meishi.qq.com/beijing/c/all[\\"-p2]*")
@ExtractBy(value = "//ul[@id=\"promos_list2\"]/li",multi = true)
public class QQMeishi {

    @ExtractBy("//div[@class=info]/a[@class=title]/h4/text()")
    private String shopName;

    @ExtractBy("//div[@class=info]/a[@class=title]/text()")
    private String promo;

    public static void main(String[] args) {
        OOSpider.create(Site.me(), new ConsolePageModelPipeline(), QQMeishi.class).addUrl
        ("http://meishi.qq.com/beijing/c/all").thread(4).run();
    }
}
```

## 5.5 结果的类型转换

类型转换（Formatter机制）是WebMagic 0.3.2增加的功能。因为抽取到的内容总是String，而我们想要的内容则可能是其他类型。Formatter可以将抽取到的内容，自动转换成一些基本类型，而无需手动使用代码进行转换。

例如：

```
@ExtractBy("//ul[@class='pagehead-actions']/li[1]/a[@class='social-count js-social-count']/text()")
private int star;
```

### 5.5.1 自动转换支持的类型

自动转换支持所有基本类型和装箱类型。

基本类型	装箱类型
int	Integer
long	Long
double	Double
float	Float
short	Short
char	Character
byte	Byte
boolean	Boolean

另外，还支持 `java.util.Date` 类型的转换。但是在转换时，需要指定Date的格式。格式按照JDK的标准来定义，具体规范可以看这里：<http://java.sun.com/docs/books/tutorial/i18n/format/simpleDateFormat.html>

```
@Formatter("yyyy-MM-dd HH:mm")
@ExtractBy("//div[@class='BlogStat']/regex('\\\\d+-\\\\d+\\\\d+\\\\s+\\\\d+:\\\\d+')")
private Date date;
```

### 5.5.2 显式指定转换类型

一般情况下，Formatter会根据字段类型进行转换，但是特殊情况下，我们会需要手动指

定类型。这主要发生在字段是 `List` 类型的时候。

```
@Formatter(value = "",subClazz = Integer.class)
@ExtractBy(value = "//div[@class='id']/text()", multi = true)
private List<Integer> ids;
```

### 5.5.3 自定义Formatter (TODO)

实际上，除了自动类型转换之外，Formatter还可以做一些结果的后处理的事情。例如，我们有一种需求场景，需要将抽取的结果作为结果的一部分，拼接上一部分字符串来使用。在这里，我们定义了一个 `StringTemplateFormatter`。

```
public class StringTemplateFormatter implements ObjectFormatter<String> {

    private String template;

    @Override
    public String format(String raw) throws Exception {
        return String.format(template, raw);
    }

    @Override
    public Class<String> clazz() {
        return String.class;
    }

    @Override
    public void initParam(String[] extra) {
        template = extra[0];
    }
}
```

那么，我们就能在抽取之后，做一些简单的操作了！

```
@Formatter(value = "author is %s",formatter = StringTemplateFormatter.class)
@ExtractByUrl("https://github\\.com/(\\w+)/.*")
private String author;
```

此功能在0.4.3版本有BUG，将会在0.5.0中修复并开放。

## 5.6 一个完整的流程

到目前为止，我们了解了URL和抽取相关API，一个爬虫已经基本编写完成了。

```
@TargetUrl("https://github.com/\w+/\w+")
@HelpUrl("https://github.com/\w+")
public class GithubRepo {

    @ExtractBy(value = "//h1[@class='entry-title public']/strong/a/text()", notNull = true)
    private String name;

    @ExtractByUrl("https://github\\.com/(\w+)/.*")
    private String author;

    @ExtractBy("//div[@id='readme']/tidyText()")
    private String readme;
}
```

### 5.6.1 爬虫的创建和启动

注解模式的入口是 `OOSpider`，它继承了 `Spider` 类，提供了特殊的创建方法，其他的方法是类似的。创建一个注解模式的爬虫需要一个或者多个 `Model` 类，以及一个或者多个 `PageModelPipeline` —— 定义处理结果的方式。

```
public static OOSpider create(Site site, PageModelPipeline pageModelPipeline, Class... pageModels);
```

### 5.6.2 PageModelPipeline

注解模式下，处理结果的类叫做 `PageModelPipeline`，通过实现它，你可以自定义自己的结果处理方式。

```
public interface PageModelPipeline<T> {

    public void process(T t, Task task);

}
```

`PageModelPipeline` 与 `Model` 类是对应的，多个 `Model` 可以对应一个 `PageModelPipeline`。除了创建时，你还可以通过

```
public OOSpider addPageModel(PageModelPipeline pageModelPipeline, Class... pageModels)
```

方法，在添加一个Model的同时，可以添加一个PageModelPipeline。

### 5.6.3 结语

好了，现在我们来完成这个例子：

```
@TargetUrl("https://github.com/\w+/\w+")
@HelpUrl("https://github.com/\w+")
public class GithubRepo {

    @ExtractBy(value = "//h1[@class='entry-title public']/strong/a/text()", notNull = true)
    private String name;

    @ExtractByUrl("https://github\\.com/(\\w+)/.*")
    private String author;

    @ExtractBy("//div[@id='readme']/tidyText()")
    private String readme;

    public static void main(String[] args) {
        OOSpider.create(Site.me().setSleepTime(1000)
            , new ConsolePageModelPipeline(), GithubRepo.class)
            .addUrl("https://github.com/code4craft").thread(5).run();
    }
}
```

## 5.7 AfterExtractor

有的时候，注解模式无法满足所有需求，我们可能还需要写代码完成一些事情，这个时候就要用到 `AfterExtractor` 接口了。

```
public interface AfterExtractor {  
    public void afterProcess(Page page);  
}
```

`afterProcess` 方法会在抽取结束，字段都初始化完毕之后被调用，可以处理一些特殊的逻辑。例如这个例子[使用Jfinal ActiveRecord持久化webmagic爬到的博客](#)：

```

//TargetUrl的意思是只有以下格式的URL才会被抽取生成model对象
//这里对正则做了一点改动，'.'默认是不需要转义的，而'*'则会自动被替换成'.*', 因为这样描述URL看着舒服一点...
//继承jfinal中的Model
//实现AfterExtractor接口可以在填充属性后进行其他操作
@TargetUrl("http://my.oschina.net/flashsword/blog/*")
public class OschinaBlog extends Model<OschinaBlog> implements AfterExtractor {

    //用ExtractBy注解的字段会被自动抽取并填充
    //默认是xpath语法
    @ExtractBy("//title")
    private String title;

    //可以定义抽取语法为Css、Regex等
    @ExtractBy(value = "div.BlogContent", type = ExtractBy.Type.Css)
    private String content;

    //multi标注的抽取结果可以是一个List
    @ExtractBy(value = "//div[@class='BlogTags']/a/text()", multi = true)
    private List<String> tags;

    @Override
    public void afterProcess(Page page) {
        //jfinal的属性其实是一个Map而不是字段，没关系，填充进去就是了
        this.set("title", title);
        this.set("content", content);
        this.set("tags", StringUtils.join(tags, ","));
        //保存
        save();
    }

    public static void main(String[] args) {
        C3p0Plugin c3p0Plugin = new C3p0Plugin("jdbc:mysql://127.0.0.1/blog?characterEncoding=utf-8", "blog", "password");
        c3p0Plugin.start();
        ActiveRecordPlugin activeRecordPlugin = new ActiveRecordPlugin(c3p0Plugin);
        activeRecordPlugin.addMapping("blog", OschinaBlog.class);
        activeRecordPlugin.start();
        //启动webmagic
        OOSpider.create(Site.me()).addStartUrl("http://my.oschina.net/flashsword/blog/145796"), OschinaBlog.class).run();
    }
}

```

## 结语

注解模式现在算是介绍结束了，在WebMagic里，注解模式其实是完全基于 `webmagic-core` 中的 `PageProcessor` 和 `Pipeline` 扩展实现的，有兴趣的朋友可以去看看代码。

这部分实现其实还是比较复杂的，如果发现一些细节的代码存在问题，欢迎向我反馈。

## 6. 组件的使用和定制

在第一章里，我们提到了WebMagic的组件。WebMagic的一大特色就是可以灵活的定制组件功能，实现你自己想要的功能。

在Spider类里，`PageProcessor`、`Downloader`、`Scheduler` 和 `Pipeline` 四个组件都是Spider的字段。除了`PageProcessor`是在Spider创建的时候已经指定，`Downloader`、`Scheduler` 和 `Pipeline` 都可以通过Spider的setter方法来进行配置和更改。

方法	说明	示例
<code>setScheduler()</code>	设置 Scheduler	<code>spider.setScheduler(new FileCacheQueueScheduler("D:\data\webmag"))</code>
<code>setDownloader()</code>	设置 Downloader	<code>spider.setDownloader(new SeleniumDownloader())</code>
<code>addPipeline()</code>	设置 Pipeline，一个Spider可以有多个 Pipeline	<code>spider.addPipeline(new FilePipeline())</code>

在这一章，我们会讲到如何定制这些组件，完成我们想要的功能。

## 6.1 使用和定制Pipeline

Pipeline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。Pipeline可以实现一些通用的功能。在这一节中，我们会对Pipeline进行介绍，并用两个例子来讲解如何定制Pipeline。

### 6.1.1 Pipeline介绍

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：

1. 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。
2. 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。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。

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，例如你可以使用

```
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。

### 6.1.2 将结果输出到控制台

在介绍 `PageProcessor` 时，我们使用了 [GithubRepoPageProcessor](#) 作为例子，其中某一段代码中，我们将结果进行了保存：

```
public void process(Page page) {
    page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+\\/\\w+")
").all());
    page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+)").all());
    //保存结果author，这个结果会最终保存到ResultItems中
    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){
        //设置skip之后，这个页面的结果不会被Pipeline处理
        page.setSkip(true);
    }
    page.putField("readme", page.getHtml().xpath("//div[@id='readme']/tidyText()"));
}
```

现在我们想将结果保存到控制台，要怎么做呢？[ConsolePipeline](#)可以完成这个工作：

```
public class ConsolePipeline implements Pipeline {

    @Override
    public void process(ResultItems<Page> resultItems, Task task) {
        System.out.println("get page: " + resultItems.getRequest().getUrl());
        //遍历所有结果，输出到控制台，上面例子中的"author"、"name"、"readme"都是一个key，其结果则是对应的value
        for (Map.Entry<String, Object> entry : resultItems.getAll().entrySet()) {
            System.out.println(entry.getKey() + ":\t" + entry.getValue());
        }
    }
}
```

参考这个例子，你就可以定制自己的Pipeline了——从 `ResultItems` 中取出数据，再按照你希望的方式处理即可。

### 6.1.3 将结果保存到MySQL

这里先介绍一个demo项目：[jobhunter](#)。它是一个集成了Spring，使用WebMagic抓取招聘信息，并且使用Mybatis持久化到Mysql的例子。我们会用这个项目来介绍如果持久化到Mysql。

在Java里，我们有很多方式将数据保存到MySQL，例如jdbc、dbutils、spring-jdbc、MyBatis等工具。这些工具都可以完成同样的事情，只不过功能和使用复杂程度不一样。如果使用jdbc，那么我们只需要从ResultItems取出数据，进行保存即可。

如果我们会使用ORM框架来完成持久化到MySQL的工作，就会面临一个问题：这些框架

## 6.1 使用和定制Pipeline

一般都要求保存的内容是一个定义好结构的对象，而不是一个key-value形式的ResultItems。以MyBatis为例，我们使用[MyBatis-Spring](#)可以定义这样一个DAO：

```
public interface JobInfoDAO {  
  
    @Insert("insert into JobInfo (`title`, `salary`, `company`, `description`, `requirement`,  
    `source`, `url`, `urlMd5`) values (#{title},#{salary},#{company},#{description},#{requirement},#{source},#{url},#{urlMd5})")  
    public int add(LieTouJobInfo jobInfo);  
}
```

我们要做的，就是实现一个Pipeline，将ResultItems和 LieTouJobInfo 对象结合起来。

## 注解模式

注解模式下，WebMagic内置了一个[PageModelPipeline](#)：

```
public interface PageModelPipeline<T> {  
  
    //这里传入的是处理好的对象  
    public void process(T t, Task task);  
  
}
```

这时，我们可以很优雅的定义一个[JobInfoDaoPipeline](#)，来实现这个功能：

```
@Component("JobInfoDaoPipeline")  
public class JobInfoDaoPipeline implements PageModelPipeline<LieTouJobInfo> {  
  
    @Resource  
    private JobInfoDAO jobInfoDAO;  
  
    @Override  
    public void process(LieTouJobInfo lieTouJobInfo, Task task) {  
        //调用MyBatis DAO保存结果  
        jobInfoDAO.add(lieTouJobInfo);  
    }  
}
```

## 基本Pipeline模式

至此，结果保存就已经完成了！那么如果我们使用原始的Pipeline接口，要怎么完成呢？其实答案也很简单，如果你要保存一个对象，那么就需要在抽取的时候，将它保存为一个对象：

## 6.1 使用和定制Pipeline

```
public void process(Page page) {
    page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+/\\w+")
    ").all());
    page.addTargetRequests(page.getHtml().links().regex("(https://github\\.com/\\w+)").al
    l());
    GithubRepo githubRepo = new GithubRepo();
    githubRepo.setAuthor(page.getUrl().regex("https://github\\.com/(\\w+)/.*").toString()
    );
    githubRepo.setName(page.getHtml().xpath("//h1[@class='entry-title public']/strong/a/t
    ext()").toString());
    githubRepo.setReadme(page.getHtml().xpath("//div[@id='readme']/tidyText()").toString(
    ));
    if (githubRepo.getName() == null) {
        //skip this page
        page.setSkip(true);
    } else {
        page.putField("repo", githubRepo);
    }
}
```

在Pipeline中，只要使用

```
GithubRepo githubRepo = (GithubRepo)resultItems.get("repo");
```

就可以获取这个对象了。

PageModelPipeline实际上也是通过原始的Pipeline来实现的，它将与PageProcessor进行了整合，在保存时，使用类名作为key，而对象则是value，具体实现见：[ModelPipeline](#)。

### 6.1.4 WebMagic已经提供的几个Pipeline

WebMagic中已经提供了将结果输出到控制台、保存到文件和JSON格式保存的几个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	

## 6.1 使用和定制Pipeline

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	想要持久化的字段需要有getter方法

## 6.2 使用和定制Scheduler

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

1. 对待抓取的URL队列进行管理。
2. 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

在0.5.1版本里，我对Scheduler的内部实现进行了重构，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

所有默认的Scheduler都使用HashSetDuplicateRemover来进行去重，（除开

## 6.2 使用和定制Scheduler

RedisScheduler是使用Redis的set进行去重）。如果你的URL较多，使用  
HashSetDuplicateRemover会比较占用内存，所以也可以尝试以下  
BloomFilterDuplicateRemover<sup>1</sup>，使用方式：

```
spider.setScheduler(new QueueScheduler()  
.setDuplicateRemover(new BloomFilterDuplicateRemover(10000000)) //10000000是估计的页面数量  
)
```

<sup>1</sup>. 0.6.0版本后，如果使用BloomFilterDuplicateRemover，需要单独引入Guava依赖包。 ↵

## 6.3 使用和定制Downloader

WebMagic的默认Downloader基于HttpClient。一般来说，你无须自己实现Downloader，不过HttpClientDownloader也预留了几个扩展点，以满足不同场景的需求。

另外，你可能希望通过其他方式来实现页面下载，例如使用 `SeleniumDownloader` 来渲染动态页面。

## 附录：实例分析

即使你对WebMagic的框架已经很熟练了，也会对有些爬虫的编写有些迷茫。比如如何定期抓取并更新、如何抓取动态渲染的页面等。

这一节我会整理一些常见案例，希望对读者有帮助。

## 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)”，其中“0\_1”中的“1”是可变的页数。

The screenshot shows a list of four blog articles:

文章标题	发布日期	操作
动态规划算法的变种--备忘录算法	2010-01-12 15:55	[编辑] 更多▼
zz职场人情：施受需谨慎	2009-12-26 14:23	[编辑] 更多▼
分治算法的一点思考--为什么大多使...	2009-12-25 10:47	[编辑] 更多▼
初学PS 贴2张做的海报	2009-12-14 18:06	[编辑] 更多▼

Page navigation: 1 [2] 下一页 > 共2页

- 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)”，其中“58ae76e80100g8au”是可变的字符。

Article title: 一道来自百度的面试题--倒排索引的AND操作

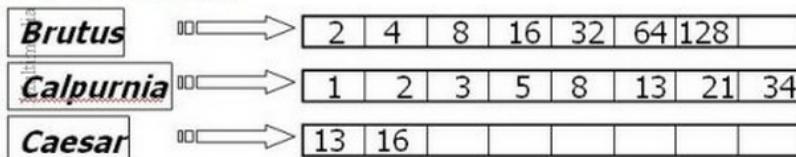
Timestamp: (2009-12-14 14:29:04)

Actions: [编辑][删除]

Share: + 转载 ▾

Tags: it

倒排索引是以关键词作为索引项来索引文档的一种机制，如图中Brutus、Calpurnia、Caesar为关键词，2、4、8等等为文档ID。



现在有一个查询：Brutus AND Calpurnia AND Caesar。这个查询实际上就是要找出Brutus(以下简称B)、Calpurnia(以下简称C1)和Caesar(以下简称C2)的索引文档中的相同项。假设B、C1、C2的长度分别为m、n、p。

比较容易想到的是用归并排序的思想来解决这个问题。即：对3个线性表进行归并排序并对相同项计数，计数为3的项即为结果，这个方法时间复杂度为O(m+n+p)，就这个例子我们需要比较7+8+2=17次。

### 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](#)。

## 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](#)。WebMagic计划在将来的0.5.0版本中，加入 `SubPageProcessor` 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。这方面的工具具有 `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](#)

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。

The screenshot shows the AngularJS Chinese Community website. At the top, there's a navigation bar with tabs for '最新' (Latest), '热门' (Hot), and '更新' (Updated). Below the navigation are several categories: 'AngularJS', 'JavaScript', 'AngularJS 开发指南' (AngularJS Development Guide), '问题与建议' (Issues and Suggestions), 'Node.js', 'jsGen', 'AngularJS 入门教程' (AngularJS Beginner Tutorial), 'AngularJS 技术杂谈' (AngularJS Technical Talk), '招聘' (Recruitment), '前端资讯' (Frontend News), and 'WebApp'. Further down are 'AngularJS 文档资讯' (AngularJS Documentation Information), '开发经验' (Development Experience), 'Backbone.js', 'MongoDB', 'HTML5', 'scope', 'JavaScript 异步编程' (JavaScript Asynchronous Programming), 'websocket', '区块' (Block), and a search bar with the placeholder 'Q更多'.

A search result for '【上海】有孚计算机网络-前端攻城师' is displayed. It has a rating of 6 stars. The result includes a profile picture, the title, the author 'm\_c是女超人', the creation time '21小时前', the update time '21小时前', the category 'JavaScript', and the status '招聘'.

The screenshot shows a browser developer tools console with the URL 'view-source:angularjs.cn/?p=1'. A red box highlights a search bar at the top right containing the text '网络-前端攻城师 第 0 条, 共 0 条'. Below the search bar, the text '找不到结果' (Not found) is displayed. The main area shows the first 10 lines of the page's source code:

```
1 <!DOCTYPE html>
2 <html lang="zh-CN">
3   <head>
4     <meta charset="utf-8">
5     <meta name="fragment" content="!">
6     <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
7     <title ng-bind="global.title2+' | '+global.title">AngularJS中文社区</title>
8     <meta name="title" content="{{global.metatitle}}">
9     <meta name="description" content="{{global.metadesc}}">
10    <meta name="keywords" content="{{global.keywords}}">
```

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URL `http://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。

```
{
  ack: true,
  error: null,
  timestamp: 1397317821146,
  - data: [
    - {
      _id: "A0y2",
      + author: {...},
      date: 1397231093647,
      display: 0,
      status: 0,
      + refer: {...},
      title: "【上海】有孚计算机网络-前端攻城师",
      ...
    }
  ]
}
```

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);  
        }  
    }  
}
```

## 2. 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](#)

## 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

### 2. 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。