**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

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来进行抽取，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

到之前为止，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](http://www.oschina.net/code/snippet_190591_23456)：

//TargetUrl的意思是只有以下格式的URL才会被抽取出生成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](https://github.com/code4craft/webmagic/issues)。