在上一节课我们通过实例了解了 Scrapy 的基本使用方法，在这个过程中，我们用到了 Spider 来编写爬虫逻辑，同时用到了一些选择器来对结果进行选择。

在这一节课，我们就对 Spider 和 Selector 的基本用法作一个总结。

Spider 的用法

在 Scrapy 中，要抓取网站的链接配置、抓取逻辑、解析逻辑等其实都是在 Spider 中配置的。在前一节课的实例中，我们发现抓取逻辑也是在 Spider 中完成的。本节课我们就来专门了解一下 Spider 的基本用法。

Spider 运行流程

在实现 Scrapy 爬虫项目时，最核心的类便是 Spider 类了，它定义了如何爬取某个网站的流程和解析方式。简单来讲，Spider 要做的事就是如下两件：

- 定义爬取网站的动作；
- 分析爬取下来的网页。

对于 Spider 类来说，整个爬取循环如下所述。

- 以初始的 URL 初始化 Request，并设置回调函数。 当该 Request 成功请求并返回时，将生成 Response，并作为参数传给该回调函数。
- 在回调函数内分析返回的网页内容。返回结果可以有两种形式，一种是解析到的有效结果返回字典或 Item 对象。下一步可经过处理后（或直接）保存，另一种是解析到的下一个（如下一页）链接，可以利用此链接构造 Request 并设置新的回调函数，返回 Request。
- 如果返回的是字典或 Item 对象，可通过 Feed Exports 等形式存入文件，如果设置了 Pipeline 的话，可以经由 Pipeline 处理（如过滤、修正等）并保存。
- 如果返回的是 Reqeust，那么 Request 执行成功得到 Response 之后会再次传递给 Request 中定义的回调函数，可以再次使用选择器来分析新得到的网页内容，并根据分析的数据生成 Item。

通过以上几步循环往复进行，便完成了站点的爬取。

Spider 类分析

在上一节课的例子中我们定义的 Spider 继承自 scrapy.spiders.Spider，这个类是最简单最基本的 Spider 类，每个其他的 Spider 必须继承自这个类，还有后面要说明的一些特殊 Spider 类也都是继承自它。

这个类里提供了 start_requests 方法的默认实现，读取并请求 start_urls 属性，并根据返回的结果调用 parse 方法解析结果。另外它还有一些基础属性，下面对其进行讲解。

- name：爬虫名称，是定义 Spider 名字的字符串。Spider 的名字定义了 Scrapy 如何定位并初始化 Spider，所以其必须是唯一的。 不过我们可以生成多个相同的 Spider 实例，这没有任何限制。 name 是 Spider 最重要的属性，而且是必需的。如果该 Spider 爬取单个网站，一个常见的做法是以该网站的域名名称来命名 Spider。例如，如果 Spider 爬取 mywebsite.com，该 Spider 通常会被命名为 mywebsite。
- allowed_domains：允许爬取的域名，是可选配置，不在此范围的链接不会被跟进爬取。
- start_urls：起始 URL 列表，当我们没有实现 start_requests 方法时，默认会从这个列表开始抓取。
- custom_settings：这是一个字典，是专属于本 Spider 的配置，此设置会覆盖项目全局的设置，而且此设置必须在初始化前被更新，所以它必须定义成类变量。
- crawler：此属性是由 from_crawler 方法设置的，代表的是本 Spider 类对应的 Crawler 对象，Crawler 对象中包含了很多项目组件，利用它我们可以获取项目的一些配置信息，如最常见的就是获取项目的设置信息，即 Settings。
- settings：是一个 Settings 对象，利用它我们可以直接获取项目的全局设置变量。

除了一些基础属性，Spider 还有一些常用的方法，在此介绍如下。

- start_requests：此方法用于生成初始请求，它必须返回一个可迭代对象，此方法会默认使用 start_urls 里面的 URL 来构造 Request，而且 Request 是 GET 请求方式。如果我们想在启动时以 POST 方式访问某个站点，可以直接重写这个方法，发送 POST 请求时我们使用 FormRequest 即可。
- parse：当 Response 没有指定回调函数时，该方法会默认被调用，它负责处理 Response，处理返回结果，并从中提取出想要的数据和下一步的请求，然后返回。该方法需要返回一个包含 Request 或 Item 的可迭代对象。
- closed：当 Spider 关闭时，该方法会被调用，在这里一般会定义释放资源的一些操作或其他收尾操作。

Selector 的用法

我们之前介绍了利用 Beautiful Soup、PyQuery，以及正则表达式来提取网页数据，这确实非常方便。而 Scrapy 还提供了自己的数据提取方法，即 Selector（选择器）。

Selector 是基于 lxml 构建的，支持 XPath 选择器、CSS 选择器，以及正则表达式，功能全面，解析速度和准确度非常高。

接下来我们将介绍 Selector 的用法。

直接使用

Selector 是一个可以独立使用的模块。我们可以直接利用 Selector 这个类来构建一个选择器对象，然后调用它的相关方法如 xpath、css 等来提取数据。

例如，针对一段 HTML 代码，我们可以用如下方式构建 Selector 对象来提取数据：

```
from scrapy import Selector

body = '<html><head><title>Hello World</title></head><body></body></html>'
selector = Selector(text=body)
title = selector.xpath('//title/text()').extract_first()
print(title)
```

运行结果：

```
Hello World
```

这里我们没有在 Scrapy 框架中运行，而是把 Scrapy 中的 Selector 单独拿出来使用了，构建的时候传入 text 参数，就生成了一个 Selector 选择器对象，然后就可以像前面我们所用的 Scrapy 中的解析方式一样，调用 xpath、css 等方法来提取了。

在这里我们查找的是源代码中的 title 中的文本，在 XPath 选择器最后加 text 方法就可以实现文本的提取了。

以上内容就是 Selector 的直接使用方式。同 Beautiful Soup 等库类似，Selector 其实也是强大的网页解析库。如果方便的话，我们也可以在其他项目中直接使用 Selector 来提取数据。

接下来，我们用实例来详细讲解 Selector 的用法。

Scrapy Shell

由于 Selector 主要是与 Scrapy 结合使用，如 Scrapy 的回调函数中的参数 response 直接调用 xpath() 或者 css() 方法来提取数据，所以在这里我们借助 Scrapy Shell 来模拟 Scrapy 请求的过程，来讲解相关的提取方法。

我们用官方文档的一个样例页面来做演示：http://doc.scrapy.org/en/latest/_static/selectors-sample1.html。

开启 Scrapy Shell，在命令行中输入如下命令：

```
scrapy shell http://doc.scrapy.org/en/latest/_static/selectors-sample1.html
```

这样我们就进入了 Scrapy Shell 模式。这个过程其实是 Scrapy 发起了一次请求，请求的 URL 就是刚才命令行下输入的 URL，然后把一些可操作的变量传递给我们，如 request、response 等，如图所示。

我们可以在命令行模式下输入命令调用对象的一些操作方法，回车之后实时显示结果。这与 Python 的命令行交互模式是类似的。

接下来，演示的实例都将页面的源码作为分析目标，页面源码如下所示：

```
<html>
 <head>
  <base href='http://example.com/' />
  <title>Example website</title>
 </head>
 <body>
  <div id='images'>
   <a href='image1.html'>Name: My image 1 <br /><img src='image1_thumb.jpg' /></a>
   <a href='image2.html'>Name: My image 2 <br /><img src='image2_thumb.jpg' /></a>
   <a href='image3.html'>Name: My image 3 <br /><img src='image3_thumb.jpg' /></a>
   <a href='image4.html'>Name: My image 4 <br /><img src='image4_thumb.jpg' /></a>
   <a href='image5.html'>Name: My image 5 <br /><img src='image5_thumb.jpg' /></a>
  </div>
 </body>
</html>
```

XPath 选择器

进入 Scrapy Shell 之后，我们将主要操作 response 变量来进行解析。因为我们解析的是 HTML 代码，Selector 将自动使用 HTML 语法来分析。

response 有一个属性 selector，我们调用 response.selector 返回的内容就相当于用 response 的 text 构造了一个 Selector 对象。通过这个 Selector 对象我们可以调用解析方法如 xpath、css 等，通过向方法传入 XPath 或 CSS 选择器参数就可以实现信息的提取。

我们用一个实例感受一下，如下所示：

```
>>> result = response.selector.xpath('//a')
>>> result
[<Selector xpath='//a' data='<a href="image1.html">Name: My image 1 <'>,
 <Selector xpath='//a' data='<a href="image2.html">Name: My image 2 <'>,
 <Selector xpath='//a' data='<a href="image3.html">Name: My image 3 <'>,
 <Selector xpath='//a' data='<a href="image4.html">Name: My image 4 <'>,
 <Selector xpath='//a' data='<a href="image5.html">Name: My image 5 <'>]
>>> type(result)
scrapy.selector.unified.SelectorList
```

打印结果的形式是 Selector 组成的列表，其实它是 SelectorList 类型，SelectorList 和 Selector 都可以继续调用 xpath 和 css 等方法来进一步提取数据。

在上面的例子中，我们提取了 a 节点。接下来，我们尝试继续调用 xpath 方法来提取 a 节点内包含的 img 节点，如下所示：

```
>>> result.xpath('./img')
[<Selector xpath='./img' data='<img src="image1_thumb.jpg">'>,
 <Selector xpath='./img' data='<img src="image2_thumb.jpg">'>,
 <Selector xpath='./img' data='<img src="image3_thumb.jpg">'>,
 <Selector xpath='./img' data='<img src="image4_thumb.jpg">'>,
 <Selector xpath='./img' data='<img src="image5_thumb.jpg">'>]
```

我们获得了 a 节点里面的所有 img 节点，结果为 5。

值得注意的是，选择器的最前方加 .（点），这代表提取元素内部的数据，如果没有加点，则代表从根节点开始提取。此处我们用了 ./img 的提取方式，则代表从 a 节点里进行提取。如果此处我们用 //img，则还是从 html 节点里进行提取。

我们刚才使用了 response.selector.xpath 方法对数据进行了提取。Scrapy 提供了两个实用的快捷方法，response.xpath 和 response.css，它们二者的功能完全等同于 response.selector.xpath 和 response.selector.css。方便起见，后面我们统一直接调用 response 的 xpath 和 css 方法进行选择。

现在我们得到的是 SelectorList 类型的变量，该变量是由 Selector 对象组成的列表。我们可以用索引单独取出其中某个 Selector 元素，如下所示：

```
>>> result[0]
<Selector xpath='//a' data='<a href="image1.html">Name: My image 1 <'>
```

我们可以像操作列表一样操作这个 SelectorList。但是现在获取的内容是 Selector 或者 SelectorList 类型，并不是真正的文本内容。那么具体的内容怎么提取呢？
比如我们现在想提取出 a 节点元素，就可以利用 extract 方法，如下所示：

```
>>> result.extract()
['<a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>', '<a href="image2.html">Name: My image 2 <br><img src="image2_thumb.jpg"></a>', '<a href="image3.html">Na
```

这里使用了 extract 方法，我们就可以把真实需要的内容获取下来。

我们还可以改写 XPath 表达式，来选取节点的内部文本和属性，如下所示：

```
>>> response.xpath('//a/text()').extract()
['Name: My image 1 ', 'Name: My image 2 ', 'Name: My image 3 ', 'Name: My image 4 ', 'Name: My image 5 ']
>>> response.xpath('//a/@href').extract()
['image1.html', 'image2.html', 'image3.html', 'image4.html', 'image5.html']
```

我们只需要再加一层 /text() 就可以获取节点的内部文本，或者加一层 /@href 就可以获取节点的 href 属性。其中，@ 符号后面内容就是要获取的属性名称。

现在我们可以用一个规则把所有符合要求的节点都获取下来，返回的类型是列表类型。

但是这里有一个问题：如果符合要求的节点只有一个，那么返回的结果会是什么呢？我们再用一个实例来感受一下，如下所示：

```
>>> response.xpath('//a[@href="image1.html"]/text()').extract()
['Name: My image 1 ']
```

我们用属性限制了匹配的范围，使 XPath 只可以匹配到一个元素。然后用 extract 方法提取结果，其结果还是一个列表形式，其文本是列表的第一个元素。但很多情况下，我们其实想要的数据就是第一个元素内容，这里我们通过加一个索引来获取，如下所示：

```
'Name: My image 1 '
```

但是，这个写法很明显是有风险的。一旦 XPath 有问题，那么 extract 后的结果可能是一个空列表。如果我们再用索引来获取，那不就可能会导致数组越界吗？
所以，另外一个方法可以专门提取单个元素，它叫作 extract_first。我们可以改写上面的例子如下所示：

```
>>> response.xpath('//a[@href="image1.html"]/text()').extract_first()
'Name: My image 1 '
```

这样，我们直接利用 extract_first 方法将匹配的第一个结果提取出来，同时我们也不用担心数组越界的问题。
另外我们也可以为 extract_first 方法设置一个默认值参数，这样当 XPath 规则提取不到内容时会直接使用默认值。例如将 XPath 改成一个不存在的规则，重新执行代码，如下所示：

```
>>> response.xpath('//a[@href="image1"]/text()').extract_first()>>> response.xpath('//a[@href="image1"]/text()').extract_first('Default Image')
'Default Image'
```

这里，如果 XPath 匹配不到任何元素，调用 extract_first 会返回空，也不会报错。在第二行代码中，我们还传递了一个参数当作默认值，如 Default Image。这样如果 XPath 匹配不到结果的话，返回值会使用这个参数来代替，可以看到输出正是如此。

到现在为止，我们了解了 Scrapy 中的 XPath 的相关用法，包括嵌套查询、提取内容、提取单个内容、获取文本和属性等。

CSS 选择器

接下来，我们看看 CSS 选择器的用法。Scrapy 的选择器同时还对接了 CSS 选择器，使用 response.css() 方法可以使用 CSS 选择器来选择对应的元素。

例如在上文我们选取了所有的 a 节点，那么 CSS 选择器同样可以做到，如下所示：

```
>>> response.css('a')
[<Selector xpath='descendant-or-self::a' data='<a href="image1.html">Name: My image 1 <'>,
<Selector xpath='descendant-or-self::a' data='<a href="image2.html">Name: My image 2 <'>,
<Selector xpath='descendant-or-self::a' data='<a href="image3.html">Name: My image 3 <'>,
<Selector xpath='descendant-or-self::a' data='<a href="image4.html">Name: My image 4 <'>,
<Selector xpath='descendant-or-self::a' data='<a href="image5.html">Name: My image 5 <'>]
```

同样，调用 extract 方法就可以提取出节点，如下所示：

```
['<a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>', '<a href="image2.html">Name: My image 2 <br><img src="image2_thumb.jpg"></a>', '<a href="image3.html">Na
```

用法和 XPath 选择是完全一样的。另外，我们也可以进行属性选择和嵌套选择，如下所示：

```
>>> response.css('a[href="image1.html"]').extract()
['<a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>']
>>> response.css('a[href="image1.html"] img').extract()
['<img src="image1_thumb.jpg">']
```

这里用 [href="image.html"] 限定了 href 属性，可以看到匹配结果就只有一个了。另外如果想查找 a 节点内的 img 节点，只需要再加一个空格和 img 即可。选择器的写法和标准 CSS 选择器写法如出一辙。

我们也可以使用 extract_first() 方法提取列表的第一个元素，如下所示：

```
>>> response.css('a[href="image1.html"] img').extract_first()
'<img src="image1_thumb.jpg">'
```

接下来的两个用法不太一样。节点的内部文本和属性的获取是这样实现的，如下所示：

```
>>> response.css('a[href="image1.html"]::text').extract_first()
'Name: My image 1 '
>>> response.css('a[href="image1.html"] img::attr(src)').extract_first()
'image1_thumb.jpg'
```

获取文本和属性需要用 ::text 和 ::attr() 的写法。而其他库如 Beautiful Soup 或 PyQuery 都有单独的方法。

另外，CSS 选择器和 XPath 选择器一样可以嵌套选择。我们可以先用 XPath 选择器选中所有 a 节点，再利用 CSS 选择器选中 img 节点，再用 XPath 选择器获取属性。我们用一个实例来感受一下，如下所示：

```
>>> response.xpath('//a').css('img').xpath('@src').extract()
['image1_thumb.jpg', 'image2_thumb.jpg', 'image3_thumb.jpg', 'image4_thumb.jpg', 'image5_thumb.jpg']
```

我们成功获取了所有 img 节点的 src 属性。
因此，我们可以随意使用 xpath 和 css 方法二者自由组合实现嵌套查询，二者是完全兼容的。

正则匹配

Scrapy 的选择器还支持正则匹配。比如，在示例的 a 节点中的文本类似于 Name: My image 1，现在我们只想把 Name: 后面的内容提取出来，这时就可以借助 re 方法，实现如下：

```
>>> response.xpath('//a/text()').re('Name:\s(.*)')
['My image 1 ', 'My image 2 ', 'My image 3 ', 'My image 4 ', 'My image 5 ']
```

我们给 re 方法传入一个正则表达式，其中 `(.*)` 就是要匹配的内容，输出的结果就是正则表达式匹配的分组，结果会依次输出。

如果同时存在两个分组，那么结果依然会被按序输出，如下所示：

```
>>> response.xpath('//a/text()').re('(.*?):\s(.*)')
['Name', 'My image 1 ', 'Name', 'My image 2 ', 'Name', 'My image 3 ', 'Name', 'My image 4 ', 'Name', 'My image 5 ']
```

类似 extract_first 方法，re_first 方法可以选取列表的第一个元素，用法如下：

```
>>> response.xpath('//a/text()').re_first('(.*?):\s(.*)')
'Name'
>>> response.xpath('//a/text()').re_first('Name:\s(.*)')
'My image 1 '
```

不论正则匹配了几个分组，结果都会等于列表的第一个元素。

值得注意的是，response 对象不能直接调用 re 和 re_first 方法。如果想要对全文进行正则匹配，可以先调用 xpath 方法然后再进行正则匹配，如下所示：

```
>>> response.re('Name:\s(.*)')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'HtmlResponse' object has no attribute 're'
>>> response.xpath('.').re('Name:\s(.*)<br>')
['My image 1 ', 'My image 2 ', 'My image 3 ', 'My image 4 ', 'My image 5 ']
>>> response.xpath('.').re_first('Name:\s(.*)<br>')
'My image 1 '
```

通过上面的例子，我们可以看到，直接调用 re 方法会提示没有 re 属性。但是这里首先调用了 `xpath('.')`选中全文，然后调用 re 和 re_first 方法，就可以进行正则匹配了。

以上内容便是 Scrapy 选择器的用法，它包括两个常用选择器和正则匹配功能。如果你熟练掌握 XPath 语法、CSS 选择器语法、正则表达式语法可以大大提高数据提取效率。