# 第7章

# **Web信息采集**

Web无疑是当前最大的非结构化文本的存储库，如果我们知道如何对其进行信息采集，就可以在自己的实验中获得所有想要的数据。也正因为如此，Web信息采集（web crawling）对于NLTK的爱好者来说是一项很值得学习的技术，而如何从Web中获取相关的数据正是本章所要介绍的主要内容。

在本章，我们将会用一个叫做Scrapy的神奇的Python库来编写一个Web爬虫。我们也将详细介绍所有可用于配置不同设置的相关信息。我们还会编写一些最常见的蜘蛛策略以及多个与之相关的用例。另外，由于Scrapy库的使用需要我们对Xpath、信息爬取，信息检索等与Web信息操作相关的概念有一个基本的了解。所以本章也会对这些主题进行一定的探讨，以确保读者在具体实现相关应用之前，能了解其实践方面的相关知识。总而言之，我们希望在阅读完本章之后，读者能对Web爬虫有个更好的了解，并能掌握：

* 如何用Scrapy库编写出属于我们自己的爬虫。
* Scrapy库的所有主要功能

## Web爬虫

Google无疑是当前最大的网页爬虫之一，它爬取的对象是整个万维网（WWW）。 Google必须对Web中现存的每一个页面进行遍历，并检索/爬取其所遍历到的全部内容。

Web爬虫是一种系统性逐页浏览Web中的页面，并对其内容进行检索或爬取的计算机程序。另外，Web爬虫还可以从已被爬取过的内容中解析出接下来要访问的URL集。因此，如果这些程序进程可以面对整个Web无限期地运行下去，我们是可以爬取到所有网页的。另外，Web爬虫也可以被叫做蜘蛛、机器人和检索器，它们只是同一事物的不同名称。

在编写我们的第一个爬虫程序之前，有那么几个要点需要我们先思考一下。以目前的技术来说，我们在用一个Web爬虫对一个网页进行遍历之前，应该要先决定我们要选取什么类型的内容，要忽略的又是什么内容。例如对于搜索引擎这样的应用来说，我们通常应该要忽略掉所有的图像、js文件、css文件以及其它非HTML文件，将注意力集中在那些可被索引并且可被搜索的HTML内容上。在某些信息提取引擎中，我们还会需要选取特定的HTML标签或网页中特定的部分。另外，如果想要执行递归式爬取操作的话，我们还需要去提取其中的URL。这就会将我们带入到爬取策略这一话题中来。在这一话题中，我们需要决定我们的递归策略是深度优先还是广度优先。我们可能会想要追踪下一网页上的所有URL，那么只要采用深度优先策略来获取这些URL即可，我们也可能会想前往下一网页中的所有URL，这样的话我们只需一路递归下去即可。

当然，我们还需要确保自己不会陷入自循环状态，因为基本上在大多数情况下，我们要遍历的是某种图结构。为此，我们需要确保自己有一个清晰的应对页面重复访问的策略。其中，聚焦爬取（focused crawling）是一种最常被讨论的爬取策略，在该策略中，我们要知道自己在找什么域或主题，以及所要抓取的域。这其中的一些问题我们将会在蜘蛛这一节中做更为详细的讨论。

|  |
| --- |
| 推荐读者看看Udacity上的视频：https://www.youtube.com/ watch?v=CDXOcvUNBaA. |

## 编写第一个爬虫程序

我们从最基本的爬虫程序开始，这个爬虫将会被用来爬取某个Web页面上的全部内容。在这里，我们要用Scrapy来编写这个爬虫。Scrapy库是Python语言环境下爬虫问题的最佳解决方案之一。本章将会讨论Scrapy库中各种不同的功能。 因此我们先要安装一下Scrapy。

我们可以用以下命令来安装：

为此，请键入以下命令：

$ pip install scrapy

用包管理来安装Scrapy无疑是最简单的方式。下面我们来测试一下安装是否一切就绪。（理想情况下，Scrapy现在应该已经被纳入到了sys.path变量中）：

>>>import scrapy

|  |
| --- |
| 如果在安装过程中出现了任何错误，请参考http://doc.scrapy.org/ en/latest/intro/install.html |

现在，我们的Scrapy库应该可以工作了。下面，我们就来看第一个蜘蛛应用示例吧：

$ scrapy startproject tutorial

在执行完上述命令之后，该示例就应该会呈现出如下目录结构：：

tutorial/

scrapy.cfg #the project configuration file

tutorial/ #the project's python module, you'll later import your code from here.

\_\_init\_\_.py

items.py #the project's items file.

pipelines.py #the project's pipelines file.

settings.py # the project's settings file.

spiders/ #a directory where you'll later put your spiders.

\_\_init\_\_.py

如您所见，该项目的顶层文件夹给我们这个示例起了个名字叫tutorial。然后该目录中还有一个项目配置文件（scrapy.cfg），该文件将用于定义项目应使用何种设置文件，并指定该项目的部署URL。除此之外，tutorial项目中还有几个重要文件：setting.py可以用来指定该项目将使用何种类型的目标管道（item pipeline）[[1]](#footnote-1)和蜘蛛。接下来是item.py和pipline.py，这两个文件定义的是我们在解析目标项时所需要的数据以及需要执行何种类型的预处理操作。最后，spider文件夹中包含的是我们若干特定URL所编写的不同蜘蛛。

对于本章首个测试性的蜘蛛，我们打算用它将一些新闻内容转储到莫言本地文件中。为此我们要在/tutorial /spiders路径下创建一个名为NewsSpider.py的文件。然后，我们就可以来编写第一个蜘蛛程序了：

>>> from scrapy.spider import BaseSpider

>>> class NewsSpider(BaseSpider):

>>> name = "news"

>>> allowed\_domains = ["nytimes.com"]

>>> start\_URLss = [

>>> 'http://www.nytimes.com/'

>>> ]

>>> def parse(self, response):

>>> filename = response.URLs.split("/")[-2]

>>> open(filename, 'wb').write(response.body)

在这个蜘蛛程序准备就绪之后，我们就可以使用以下命令开始进行爬取了：

$ scrapy crawl news

在执行完上述命令后，我们应该终端中看到类似这样的日志信息：

[scrapy] INFO: Scrapy 0.24.5 started (bot: tutorial)

[scrapy] INFO: Optional features available: ssl, http11, boto

[scrapy] INFO: Overridden settings: {'NEWSPIDER\_MODULE': 'tutorial. spiders',

'SPIDER\_MODULES': ['tutorial.spiders'], 'BOT\_NAME': 'tutorial'}

[scrapy] INFO: Enabled extensions: LogStats, TelnetConsole, CloseSpider,

WebService, CoreStats, SpiderState

如果没有看到像上面这样的日志信息，那么我们一定在之前做错了某些事。请检查一下该蜘蛛程序所在的位置以及其它与Scrapy相关的设置，譬如说与crawl命令匹配的蜘蛛名称是否正确，以及在setting.py文件中所配置的蜘蛛和目标项是否与实际情况一致。

当然了，如果一切顺利的话，我们就会在本地文件夹中看到一个名为www.nytimes.com的文件，其中包含了[www.nytimes.com](http://www.nytimes.com)这一页面上的全部内容。

下面，让我们来详细地看一下这个蜘蛛代码中所用到的一些关系词：

* name：这是Scrapy库为我们的蜘蛛程序分配的标识符，便于它查找与该蜘蛛相应的spider类。因此，crawl命令的参数应该始终与该名称相匹配。另外，我们还需要注意该名称是区分大小写的，必须要确保它的唯一性。
* start\_urls：这是该蜘蛛程序将要爬取的URL列表。爬虫程序通常会以某个种子URL为起点开始爬取，通过对其调用parse()方法来解析并查找下一个要爬取的URL。当然在这里，我们可以提供一份可执行爬取动作的起点URL列表，而不只是一个种子URL。
* parse()：我们会通过调用该方法来解析起点URL上的数据。其元素种类的逻辑将会由目标项的特定属性来选取。这样做可以简单地将HTML页面的整个内容转换为与许多可调用的解析方法一样复杂的、并可以针对各独立的目标项属性的不同选择器。

因此，这段代码只是指定了一个URL（在这里就是www.nytimes.com）以充当爬取的起点，然后爬取了该页面上的全部内容。通常来说，爬虫程序要比这更复杂一些，要做的是会更多一些。现在让我们先暂且退后一步，来了解一下这段代码背后究竟发生了哪些事。为此，我们要来看下面这张图：

（图：图中翻译

Scheduler：调度器

Internet：互联网

Requests：请求

Item pipeline：目标项管道

Scrapy engine：Scrapy引擎

Downloader middlewares：下载器中间件

Downloader：下载器

Items：目标项

Spider middlewares：蜘蛛中间件

Responses：响应

Spiders：蜘蛛

）

## Scrapy库中的数据流

The data flow in Scrapy is controlled by the execution engine and goes like this:

Scrapy中的数据流由执行引擎控制，如下所示：

1. The process starts with locating the chosen spider and opening the first URL from the list of start\_urls.

该过程开始于找到所选的蜘蛛并从start\_urls列表打开第一个URL。

2. The first URL is then scheduled as a request in a scheduler. This is more of an internal to Scrapy.

然后，第一个URL被调度为调度程序中的请求。 这更多是Scrapy的内部。

3. The Scrapy engine then looks for the next set of URLs to crawl.

Scrapy引擎然后查找要爬网的下一组URL。

4. The scheduler then sends the next URLs to the engine and the engine then forwards it to the downloader using the downloaded middleware. These middlewares are where we place different proxies and user-agent settings.

调度器然后将下一个URL发送到引擎，然后引擎使用下载的中间件将其转发到下载器。 这些中间件是我们放置不同代理和用户代理设置的地方。

5. The downloader downloads the response from the page and passes it to the spider, where the parse method selects specific elements from the response.

下载器从页面下载响应，并将其传递给蜘蛛，其中解析方法从响应中选择特定元素。

6. Then, the spider sends the processed item to the engine.

然后，蜘蛛将处理的物品发送到发动机。

7. The engine sends the processed response to the item pipeline, where we can add some post processing.

引擎将处理后的响应发送到项目管道，在那里我们可以添加一些后处理。

8. The same process continues for each URL until there are no remaining requests.

对每个URL继续相同的过程，直到没有剩余的请求。

### The Scrapy shell

The best way to understand Scrapy is to use it through a shell and to get your hands dirty with some of the initial commands and tools provided by Scrapy. It allows you to experiment and develop your XPath expressions that you can put into your spider code.

理解Scrapy的最好方法是通过shell使用它，并使用Scrapy提供的一些初始命令和工具使你的手变脏。 它允许你实验和开发你的XPath表达式，你可以放入你的蜘蛛代码。

To experiment with the Scrapy shell, I would recommend you to install one of the developer tools (Chrome) and Firebug (Mozilla Firefox) as a plugin. This tool will help us dig down to the very specific part that we want from the web page.

要试验Scrapy shell，我建议您安装一个开发工具（Chrome）和Firebug（Mozilla Firefox）作为插件。 这个工具将帮助我们挖掘我们想从网页的特定部分。

Now, let's start with a very interesting use case where we want to capture the trending topics from Google news (https://news.google.com/).

现在，让我们从一个非常有趣的用例开始，我们要从Google新闻（https://news.google.com/）捕获热门主题。

The steps to follow here are:

这里的步骤是：

1. Open https://news.google.com/ in your favorite browser.

2. Go to the trending topic section on Google news. Then, right-click on and select Inspect Element for the first topic, as shown in the following screenshot:

（图）

3. The moment you open this, there will be a side window that will pop up and you will get a view.

4. Search and select the div tag. For this example, we are interested in <div class="topic">.

5. Once this is done, you will come to know that we have actually parsed the specific part of the web page, as shown in the following screenshot:

1.在您最喜欢的浏览器中打开https://news.google.com/。

2.转到Google新闻的热门话题部分。 然后，右键单击并选择检查元素的第一个主题，如以下屏幕截图所示：

（图）

你打开这一刻的那一刻，会有一个侧面的窗口会弹出，你会得到一个视图。

4.搜索并选择div标签。 对于这个例子，我们感兴趣的是<div class =“topic”>。

一旦这完成，你会知道我们实际上解析了网页的特定部分，如下面的截图所示：

（图）

Now, what we actually did manually in the preceding steps can be done in an automated way. Scrapy uses an XML path language called XPath. XPath can be used to achieve this kind of functionality. So, let's see how we can implement the same example using Scrapy.

现在，我们实际上在之前的步骤中手动执行的操作可以以自动方式完成。 Scrapy使用称为XPath的XML路径语言。 XPath可以用来实现这种功能。 因此，让我们看看如何使用Scrapy实现相同的示例。

To use Scrapy, put the following command in you cmd:

要使用Scrapy，请在cmd中放入以下命令：

$scrapy shell https://news.google.com/

The moment you hit enter, the response of the Google news page is loaded in the Scrapy shell. Now, let's move to the most important aspect of Scrapy where we want to understand how to look for a specific HTML element of the page. Let's start and run the example of getting topics from Google news that are shown in the preceding image:

当您按Enter键时，Google新闻页面的响应会加载到Scrapy shell中。 现在，让我们转到Scrapy的最重要的方面，我们想要了解如何查找页面的特定HTML元素。 让我们开始并运行从Google新闻获取主题的示例，如上图所示：

In [1]: sel.xpath('//div[@class="topic"]').extract()

The output to this will be as follows:

其输出如下：

Out[1]:

[<Selector xpath='//div[@class="topic"]' data=u'<div class="topic"><a href="/news/sectio'>,

<Selector xpath='//div[@class="topic"]' data=u'<div class="topic"><a href="/news/sectio'>,

<Selector xpath='//div[@class="topic"]' data=u'<div class="topic"><a href="/news/sectio'>]

Now, we need to understand some of the functions that Scrapy and XPath provide to experiment with the shell and then, we need to update our spider to do more sophisticated stuff. Scrapy selectors are built with the help of the lxml library, which means that they're very similar in terms of speed and parsing accuracy.

现在，我们需要了解Scrapy和XPath提供的一些功能来实验shell，然后，我们需要更新我们的蜘蛛来做更复杂的工作。 Scrapy选择器是在lxml库的帮助下构建的，这意味着它们在速度和解析精度方面非常相似。

Let's have a look at some of the most frequently used methods provided for selectors:

让我们来看看为选择器提供的一些最常用的方法：

• xpath(): This returns a list of selectors, where each of the selectors represents the nodes selected by the XPath expression given as an argument.

•xpath（）：返回选择器列表，其中每个选择器表示由作为参数给定的XPath表达式选择的节点。

• css(): This returns a list of selectors. Here, each of the selectors represent the nodes selected by the CSS expression given as an argument.

•css（）：返回选择器列表。这里，每个选择器表示由作为参数给出的CSS表达式选择的节点。

• extract():This returns content as a string with the selected data.

•extract（）：返回带有所选数据的字符串的内容。

• re(): This returns a list of unicode strings extracted by applying the regular expression given as an argument.

•re（）：返回通过应用作为参数给出的正则表达式提取的Unicode字符串列表。

I am giving you a cheat sheet of these top 10 selector patterns that can cover most of your work for you. For a more complex selector, if you search the Web, there should be an easy solution that you can use. Let's start with extracting the title of the web page that is very generic for all web pages:

我给你一个这些十大选择器模式的备忘单，可以覆盖你的大部分工作。对于更复杂的选择器，如果您搜索Web，应该有一个容易的解决方案，您可以使用。让我们从提取所有网页非常通用的网页的标题开始：

In [2] :sel.xpath('//title/text()')

Out[2]: [<Selector xpath='//title/text()' data=u' Google News'>]

Now, once you have selected any element, you also want to extract for more processing. Let's extract the selected content. This is a generic method that works with any selector:

现在，一旦你选择了任何元素，你也想提取更多的处理。 让我们提取所选内容。 这是一个适用于任何选择器的通用方法：

In [3]: sel.xpath('//title/text()').extract()

Out[3]: [u' Google News']

The other very generic requirement is to look for all the elements in the given page. Let's achieve this with this selector:

另一个非常通用的要求是查找给定页面中的所有元素。 让我们用这个选择器来实现：

In [4]: sel.xpath('//ul/li')

Out [4] : list of elements (divs and all)

We can extract all the titles in the page with this selector:

我们可以使用此选择器提取页面中的所有标题：

In [5]: sel.xpath('//ul/li/a/text()').extract()

Out [5]: [ u'India',

u'World',

u'Business',

u'Technology',

u'Entertainment',

u'More Top Stories']

With this selector, you can extract all the hyperlinks in the web page:

使用此选择器，您可以提取网页中的所有超链接：

In [6]:sel.xpath('//ul/li/a/@href').extract()

Out [6] : List of urls

Let's select all the <td> and div elements:

In [7]:sel.xpath('td'')

In [8]:divs=sel.xpath("//div")

This will select all the divs elements and then, you can loop it:

In [9]: for d in divs: printd.extract()

This will print the entire content of each div in the entire page. So, in case you are not able to get the exact div name, you can also look at the regex-based search.

这将打印整个页面中每个div的整个内容。 因此，如果您无法获取确切的div名称，也可以查看基于正则表达式的搜索。

Now, let's select all div elements that contain the attribute class="topic":

现在，让我们选择包含属性class =“topic”的所有div元素：

In [10]:sel.xpath('/div[@class="topic"]').extract()

In [11]: sel.xpath("//h1").extract() # this includes the h1 tag

This will select all the <p> elements in the page and get the class of those elements:

In [12 ] for node in sel.xpath("//p"): print node.xpath("@class").extract()

Out[12] print all the <p>

In [13]: sel.xpath("//li[contains(@class, 'topic')]") Out[13]:

[<Selector xpath="//li[contains(@class, 'topic')]" data=u'<li class="navitem nv-FRONTPAGE selecte'>,

<Selector xpath="//li[contains(@class, 'topic')]" data=u'<li class="navitem nv-FRONTPAGE selecte'>]

Let's write some selector nuggets to get the data from a css file. If we just want to extract the title from the css file, typically, everything works the same, except you need to modify the syntax:

让我们编写一些选择器块以从css文件获取数据。 如果我们只想从css文件中提取标题，通常情况下，一切都是一样的，除非你需要修改语法：

In [14] :sel.css('title::text').extract()

Out[14]: [u'Google News']

Use the following command to list the names of all the images used in the page:

In[15]: sel.xpath('//a[contains(@href, "image")]/img/@src').extract()

Out [15] : Will list all the images if the web developer has put the images in /img/src

Let's see a regex-based selector:

In [16 ]sel.xpath('//title').re('(\w+)')

Out[16]: [u'title', u'Google', u'News', u'title']

In some cases, removing the namespaces can help us get the right pattern. A selector has an inbuilt remove\_namespaces() function to make sure that the entire document is scanned and all the namespaces are removed. Make sure before using it whether we want some of these namespaces to be part of the pattern or not. The following is example of remove\_namespaces() function:

在某些情况下，删除命名空间可以帮助我们获得正确的模式。 选择器具有内置的remove\_namespaces（）函数，以确保扫描整个文档，并删除所有命名空间。 在使用它之前，确保我们是否需要其中一些命名空间是模式的一部分。 以下是remove\_namespaces（）函数的示例：

In [17] sel.remove\_namespaces() sel.xpath("//link")

Now that we have more understanding about the selectors, let's modify the same old news spider that we built previously:

现在我们对选择器有了更多的了解，让我们修改我们之前构建的同一个旧的新闻蜘蛛：

>>>from scrapy.spider import BaseSpider >>>class NewsSpider(BaseSpider):

>>> name = "news"

>>> allowed\_domains = ["nytimes.com"]

>>> start\_URLss = [

>>> 'http://www.nytimes.com/'

>>> ]

>>>def parse(self, response):

>>> sel = Selector(response)

>>> sites = sel.xpath('//ul/li') >>> for site in sites:

>>> title = site.xpath('a/text()').extract()

>>> link = site.xpath('a/@href').extract()

>>> desc = site.xpath('text()').extract()

>>> print title, link, desc

Here, we mainly modified the parse method, which is one of the core of our spider. This spider can now crawl through the entire page, but we do a more structured parsing of the title, description, and URLs.

这里，我们主要修改parse方法，这是我们蜘蛛的核心之一。 此蜘蛛可以抓取整个页面，但我们对标题，说明和网址进行了更加结构化的解析。

Now, let's write a more robust crawler using all the capabilities of Scrapy.

现在，让我们使用Scrapy的所有功能编写一个更强大的crawler。

### Items

Until now, we were just printing the crawled content on stdout or dumping it in a file. A better way to do this is to define items.py every time we write a crawler. The advantage of doing this is that we can consume these items in our parse method, and this can also give us output in any data format, such as XML, JSON, or CSV. So, if you go back to your old crawler, the items class will have a function like this:

到目前为止，我们只是在stdout上打印抓取的内容或将其转储到文件中。 一个更好的方法是每次我们编写一个crawler时定义items.py。 这样做的优点是，我们可以在我们的解析方法中使用这些项目，这也可以提供任何数据格式的输出，例如XML，JSON或CSV。 所以，如果你回到你的旧抓取器，items类将有这样的功能：

>>>fromscrapy.item import Item, Field >>>class NewsItem(scrapy.Item):

>>> # define the fields for your item here like:

>>> # name = scrapy.Field()

>>> pass

Now, let's make it like the following by adding different fields:

>>>from scrapy.item import Item, Field >>>class NewsItem(Item):

>>> title = Field()

>>> link = Field()

>>> desc = Field()

Here, we added field() to title, link, and desc. Once we have a field in place, our spider parse method can be modified to parse\_news\_item, where instead dumping the parsed fields to a file now it can be consumed by an item object.

在这里，我们添加了field（）到title，link和desc。一旦我们有一个字段，我们的蜘蛛解析方法可以修改为parse\_news\_item，而不是将解析的字段转储到一个文件，现在它可以被项目对象使用。

A Rule method is a way of specifying what kind of URL needs to be crawled after the current one. A Rule method provides SgmlLinkExtractor, which is a way of defining the URL pattern that needs to be extracted from the crawled page. A Rule method also provides a callback method, which is typically a pointer for a spider to look for the parsing method, which in this case is parse\_news\_item. In case we have a different way to parse, then we can have multiple rules and parse methods. A Rule method also has a Boolean parameter to follow, which specifies whether links should be followed by each response extracted with this rule. If the callback is None, follow defaults to True: otherwise, it default to False.

规则方法是一种指定在当前网址之后需要抓取哪种类型的网址的方法。规则方法提供了SgmlLinkExtractor，这是一种定义需要从已爬网页面提取的网址格式的方法。 Rule方法还提供了一个回调方法，它通常是一个用于寻找解析方法的蜘蛛的指针，在这种情况下是parse\_news\_item。如果我们有不同的方式来解析，那么我们可以有多个规则和解析方法。规则方法也有一个布尔参数，它指定是否应该使用此规则提取的每个响应后跟链接。如果回调为None，默认为True：否则默认为False。

One important point to note is that the Rule method does not use parse. This is because the name of the default callback method is parse() and if we use it, we are actually overriding it, and that can stop the functionality of the crawl spider. Now, let's jump on to the following code to understand the preceding methods and parameters:

需要注意的一个重要的点是规则方法不使用解析。这是因为默认回调方法的名称是parse（），如果我们使用它，我们实际上覆盖它，并且可以停止爬网蜘蛛的功能。现在，让我们跳到下面的代码来了解前面的方法和参数：

>>>from scrapy.contrib.spiders import CrawlSpider, Rule

>>>from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor

>>>from scrapy.selector import Selector

>>>from scrapy.item import NewsItem >>>class NewsSpider(CrawlSpider):

>>> name = 'news'

>>> allowed\_domains = ['news.google.com']

>>> start\_urls = ['https://news.google.com']

>>> rules = (

>>> # Extract links matching cnn.com

>>> Rule(SgmlLinkExtractor(allow=('cnn.com', ), deny=(http:// edition.cnn.com/', ))), >>> # Extract links matching 'news.google.com'

>>> Rule(SgmlLinkExtractor(allow=('news.google.com', )), callback='parse\_news\_item'), >>> ) >>> def parse\_news\_item(self, response):

>>> sel = Selector(response)

>>> item = NewsItem()

>>> item['title'] = sel.xpath('//title/text()').extract()

>>> item[topic] = sel.xpath('/div[@class="topic"]').extract()

>>> item['desc'] = sel.xpath('//td//text()').extract()

>>> return item

## The Sitemap spider

If the site provides sitemap.xml, then a better way to crawl the site is to use SiteMapSpider instead.

如果网站提供sitemap.xml，那么更好的方式来抓取网站是使用SiteMapSpider。

Here, given sitemap.xml, the spider parses the URLs provided by the site itself. This is a more polite way of crawling and good practice:

在这里，给定sitemap.xml，蜘蛛会解析网站本身提供的网址。 这是一个更有礼貌的爬行方式和良好的做法：

>>>from scrapy.contrib.spiders import SitemapSpider >>>class MySpider(SitemapSpider):

>>> sitemap\_URLss = ['http://www.example.com/sitemap.xml']

>>> sitemap\_rules = [('/electronics/', 'parse\_electronics'), ('/ apparel/', 'parse\_apparel'),] >>> def 'parse\_electronics'(self, response):

>>> # you need to create an item for electronics,

>>> return >>> def 'parse\_apparel'(self, response):

>>> #you need to create an item for apparel

>>> return

In the preceding code, we wrote one parse method for each product category. It's a great use case if you want to build a price aggregator/comparator. You might want to parse different attributes for different products, for example, for electronics, you might want to scrape the tech specification, accessory, and price; while for apparels, you are more concerned about the size and color of the item. Try your hand at using one of the retailer sites and use shell to get the patterns to scrape the size, color, and price of different items. If you do this, you should be in a good shape to write your first industry standard spider.

在上面的代码中，我们为每个产品类别编写了一个解析方法。 这是一个伟大的用例，如果你想建立一个价格聚合/比较器。 您可能需要解析不同产品的不同属性，例如，对于电子产品，您可能想要刮除技术规格，附件和价格; 而对于服装，你更关心的项目的大小和颜色。 尝试你的手在使用零售商网站之一，并使用shell来获取模式，以刮除不同项目的大小，颜色和价格。 如果你这样做，你应该在一个良好的形状，写你的第一个工业标准的蜘蛛。

In some cases, you want to crawl a website that needs you to log in before you can enter some parts of the website. Now, Scrapy has a workaround that too. They implemented FormRequest, which is more of a POST call to the HTTP server and gets the response. Let's have a deeper look into the following spider code:

在某些情况下，您希望抓取需要您登录的网站，然后才能输入网站的某些部分。 现在，Scrapy也有一个解决方法。 他们实现了FormRequest，它更多的是对HTTP服务器的POST调用，并获得响应。 让我们更深入地了解下面的蜘蛛代码：

>>>class LoginSpider(BaseSpider):

>>> name = 'example.com'

>>> start\_URLss = ['http://www.example.com/users/login.php'] >>> def parse(self, response):

>>> return [FormRequest.from\_response(response,

formdata={'username': 'john', 'password': 'secret'}, callback=self.after\_ login)] >>> def after\_login(self, response):

>>> # check login succeed before going on

>>> if "authentication failed" in response.body:

>>> self.log("Login failed", level=log.ERROR)

>>> return

For a website that requires just the username and password without any captcha, the preceding code should work just by adding the specific login details. This is the part of the parse method since you need to log in the first page in the most of the cases. Once you log in, you can write your own after\_login callback method with items and other details.

对于只需要用户名和密码而没有任何验证码的网站，上述代码只需添加特定的登录详细信息即可。 这是解析方法的一部分，因为您需要在大多数情况下登录第一页。 一旦你登录，你可以编写自己的after\_login回调方法与项目和其他细节。

## The item pipeline

Let's talk about some more item postprocessing. Scrapy provides a way to define a pipeline for items as well, where you can define the kind of post processing an item has to go through. This is a very methodical and good program design.

让我们谈谈一些更多的项目后处理。 Scrapy提供了一种为项目定义管道的方法，您可以在其中定义项目必须经过的后处理类型。 这是一个非常有条理和良好的程序设计。

We need to build our own item pipeline if we want to post process scraped items, such as removing noise and case conversion, and in other cases, where we want to derive some values from the object, for example, to calculate the age from DOB or to calculate the discount price from the original price. In the end, we might want to dump the item separately into a file.

我们需要构建自己的项目管道，如果我们想要处理刮擦的项目，如去除噪声和大小写转换，在其他情况下，我们想从对象中获取一些值，例如，从DOB计算年龄 或从原始价格计算折扣价格。 最后，我们可能需要将该项目单独转储到一个文件中。

The way to achieve this will be as follows:

实现这一点的方法如下：

1. We need to define an item pipeline in setting.py:

我们需要在setting.py中定义一个项目管道：

ITEM\_PIPELINES = {

'myproject.pipeline.CleanPipeline': 300,

'myproject.pipeline.AgePipeline': 500,

'myproject.pipeline.DuplicatesPipeline: 700,

'myproject.pipeline.JsonWriterPipeline': 800,

}

2. Let's write a class to clean the items:

让我们编写一个类来清理项目：

>>>from scrapy.exceptions import Item

>>>import datetime

>>>import datetime >>>class AgePipeline(object):

>>> def process\_item(self, item, spider):

>>> if item['DOB']:

>>> item['Age'] = (datetime.datetime. strptime(item['DOB'], '%d-%m-%y').date()-datetime.datetime. strptime('currentdate, '%d-%m-%y').date()).days/365

>>> return item

3. We need to derive the age from DOB. We used Python's date functions to achieve this:

我们需要从DOB推导出年龄。 我们使用Python的日期函数来实现这一点：

>>>from scrapy import signals

>>>from scrapy.exceptions import Item >>>class DuplicatesPipeline(object):

>>> def \_\_init\_\_(self):

>>> self.ids\_seen = set() >>> def process\_item(self, item, spider):

>>> if item['id'] in self.ids\_seen:

>>> raise DropItem("Duplicate item found: %s" % item) >>> else:

>>> self.ids\_seen.add(item['id'])

>>> return item

4. We also need to remove the duplicates. Python has the set() data structure that only contains unique values, we can create a pipline DuplicatesPipeline.py like below using Scrapy :

我们还需要删除重复的。 Python的set（）数据结构只包含唯一的值，我们可以使用Scrapy创建一个如下所示的pipline DuplicatesPipeline.py：

>>>from scrapy import signals

>>>from scrapy.exceptions import Item >>>class DuplicatesPipeline(object):

>>> def \_\_init\_\_(self):

>>> self.ids\_seen = set() >>> def process\_item(self, item, spider):

>>> if item['id'] in self.ids\_seen:

>>> raise DropItem("Duplicate item found: %s" % item) >>> else:

>>> self.ids\_seen.add(item['id'])

>>> return item

5. Let's finally write the item in the JSON file using JsonWriterPipeline.py pipeline:

让我们最后使用JsonWriterPipeline.py pipeline在JSON文件中编写项目：

>>>import json

>>>class JsonWriterPipeline(object):

>>> def \_\_init\_\_(self):

>>> self.file = open('items.txt', 'wb') >>> def process\_item(self, item, spider):

>>> line = json.dumps(dict(item)) + "\n"

>>> self.file.write(line)

>>> return item

## 外部参考资料

I encourage you to follow some simple spiders and try building some cool applications using these spiders. I would also like you to look at the following links for reference:

我鼓励你遵循一些简单的蜘蛛，并尝试使用这些蜘蛛构建一些很酷的应用程序。 我也希望你看看以下链接供参考：

* <http://doc.scrapy.org/en/latest/intro/tutorial.html>
* <http://doc.scrapy.org/en/latest/intro/overview.html>

## 本章小结

In this chapter, you learned about another great Python library and now, you don't need help from anybody for your data needs. You learned how you can write a very sophisticated crawling system, and now you know how to write a focused spider. In this chapter, we saw how to abstract the item logic from the main system and how to write some specific spider for the most common use cases. We know some of the most common settings that need to be taken care of in order to implement our own spider and we wrote some complex parse methods that can be reused. We understand selectors very well and know a hands-on way of figuring out what kind of elements we want for specific item attributes, and we also went through Firebug to get more of a practical understanding of selectors. Last but not least, very importantly, make sure that you follow the security guidelines of the websites you crawl.

在本章中，你学习了另一个伟大的Python库，现在，你不需要任何人的帮助为您的数据需要。你学会了如何编写一个非常复杂的爬虫系统，现在你知道如何写一个专注的蜘蛛。在本章中，我们看到了如何从主系统抽象项目逻辑，以及如何为最常见的用例编写一些特定的蜘蛛。我们知道一些最常见的设置，需要注意为了实现我们自己的蜘蛛，我们写了一些复杂的解析方法，可以重复使用。我们非常了解选择器，并且知道一个动手的方法来确定我们想要的特定项目属性，我们也通过Firebug获得更多的选择器的实际理解。最后但同样重要的是，请务必遵守您抓取的网站的安全指南。

In the next chapter, we will explore some essential Python libraries that can be used for natural language processing and machine learning.

在下一章中，我们将探讨一些基本的Python库，可以用于自然语言处理和机器学习。

1. 译者注：在Scrapy中，目标项（items）是用来加载被爬取内容的容器，其结构上有点像Python中的字典类型，但额外提供了一些保护以减少错误。 [↑](#footnote-ref-1)