# 第7章 **Web爬虫**

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内容进行爬取，就可以在自己的作业中获得所有想要的数据。正因为如此，Web爬取（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。我们会为您详细介绍该库所有可用于配置的各种不同设置信息。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，能了解其在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。而且，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[[1]](#footnote-1)。

在编写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

让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：

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

|  |
| --- |
| 如果在安装过程中遇到了任何错误，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）[[2]](#footnote-2)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某一本地文件中。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。

只要一切顺利，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

在Scrapy中，数据流是由执行引擎所控制的，其主要流程如下：

1. 该执行进程会找到我们之前所选择的蜘蛛程序并启动它，并打开start\_urls列表中的第一个URL。
2. 接着，某个调度器会以请求的形式来调度这第一个URL。这更多地属于Scrapy的内部操作。
3. 然后，Scrapy引擎会去继续查找下一组URL并对其进行爬取。
4. 再接着，调度器会将下一批URL发送给执行引擎，而执行引擎会用下载中间件将其转发给下载器。这些中间件位于我们存放不同代理和用户代理设置的地方。
5. 然后，下载器会去下载来自相关页面的响应内容，并将其传递给蜘蛛程序，该蜘蛛的解析方法会从这段响应内容中选取特定的元素。
6. 接下来，蜘蛛程序会将处理完的目标项发送给Scrapy引擎。
7. Scrapy引擎再将处理完的响应内容发送给目标项管道，在该管道中我们还可以再添加一些后续处理。
8. Scrapy引擎会继续对每个URL执行相同的过程，直到没有剩下的请求为止。

### Scrapy库的shell

理解Scrapy库的最好方法就是在shell中使用它，亲自去使用一些由Scrapy库所提供的初始命令和工具。这些命令允许我们在练习和开发的过程中使用XPath表达式，我们可以将其放入自己的蜘蛛代码。

想要在Scrapy库的shell中进行作业的话，我们建议您安装一下Chrome浏览器的开发插件和Firebug（Mozilla Firefox的插件）。这类工具对于我们从网页特定部分挖掘相关信息会很有帮助。

下面先来看一个非常有趣的用例。在这个用例中，我们的任务是要从Google新闻（https://news.google.com/）中抓取出热门话题。

其具体步骤如下：

1. 在我们喜欢的浏览器中打开<https://news.google.com/>。
2. 接下来，请切换到转到Google新闻的热门话题部分。并在第一个热门话题上单击右键，然后选择Inspect Element菜单项，其屏幕截图如下所示：

（图）

1. 在打开该菜单项的一刻，浏览器会弹开一个侧边窗口，我们会看到这样一个视图。
2. 接着，请搜索并选取相关的div标签。譬如在这个例子中，我们感兴趣的是<div class =“topic”>。
3. 待上述操作完成之后，我们就会了解到实际上自己已经完成了对相关网页的特定部分进行解析，其屏幕截图如下所示：

（图）

现在，我们事实上是要用自动化的方式来完成上面这些手动的操作步骤。Scrapy库使用了一种被称之为XPath的XML路径语言。而Xpath是可以被用来实现上面这类功能的。 因此，下面我们就要来看看Scrapy库是如何实现同一个例子的。

想要使用Scrapy库，我们需要先在命令行环境中输入以下命令：

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

在我们按下回车键的那一刹那，Google新闻所在网页的响应内容就会被加载到Scrapy库的shell中。下面，我们要将注意力转到Scrapy库中最重要的方面上来，了解如何查找网页中特定的HTML元素。现在，让我们启动并运行上图中这个从Google新闻中获取相关话题的例子吧：

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

然后，我们会得到如下输出：

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'>]

现在，我们需要来了解一下自己在shell中用过的这些由Scrapy和XPath提供的函数。然后再继续更新我们的蜘蛛程序，让它来执行一些更为复杂的工作。由于Scrapy选择器是在lxml库的辅助下构建的，这说明它们在速度和解析精度方面会非常相似。

下面我们来看一些最常用的选择器方法：

* xpath()：该方法会返回一个选择器列表，其中的每个选择器都代表了一个由XPath表达式参数所选中的节点。
* css()：该方法会返回一个选择器列表，其中的每个选择器都代表了一个由CSS表达式参数所选中的节点。
* extract()：该方法会以字符串的形式返回被选中数据的内容。
* re()：该方法会返回一个unicode字符串的列表，其内容是由给定的正则表达式参数所提取的。

稍后，我会给您一份前10大选择器的清单，这些选择器足以覆盖到我们日常会涉及到的大部分工作了。而对于更复杂的选择器，如果您Web上去搜索一番，通常就应该可以找到一个简单易用的解决方案了。下面，让我们试着从网页中提取它的标题，这对于所有的网页都是很常见的任务：

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

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

现在，在我们选取完元素之后，就会想继续执行更多的提取处理。下面就来提取被选取元素的内容。这是一个所有选择器都适用的通用方法：

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

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

除此之外，查看给定网页中的所有元素也是一个非常通用的请求操作。下面就用这个选择器来实现它：

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

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

我们也可以用下面这个选择器来提取网页中所有的标题：

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']

通过下面这个选择器，我们可以提取网页中所有的超链接：

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

Out [6] : List of urls

下面我们要选取的是所有的<td>和div元素：

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

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

接下来要选取的是所有的div元素，在这里我们可以使用循环：

In [9]: for d in divs:

printd.extract()

上述代码会打印出整个网页中各个div元素中的全部内容。因此，在无法获取div的准确名称的情况下，我们也可以用基于正则表达式的搜索功能来进行查看。

现在，我们再来选取所有包含属性class =“topic”的div元素：

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

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

下面来选取网页中所有的<p>元素，并获取这些元素的class属性信息：

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'>]

接下来，我们要来编写一些用于从css文件中获取数据的选择器。如果我们只想从css文件中提取相关的标题，所要做的操作通常与之前相同，无非就是要做些语法上的修改：

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

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

我们通过下面的命令可以将网页中所用的所有图片名称列出来：

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

最后，我们来看一下基于正则表达式的选择器：

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

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

在某些情况下，移除掉命名空间可以帮助我们获得正确的模式。选择器中通常都会内置remove\_namespaces()函数，该函数会确保相关文档被完整地扫描到，同时也会移除掉其中所有的命名空间。当然在调用这个函数之前，我们应该要先确认一下其中的一些命名空间是否属于模式的一部分。下面我们来看看remove\_namespaces()函数是如何被调用：

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

现在，我们对选择器应该有了更多的了解。下面我们继续来修改之前构建的那个新闻蜘蛛程序：

>>> 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

如您所见，我们在这里主要修改了parse()方法，这是我们蜘蛛程序的核心方法之一。现在，这个蜘蛛程序就对整个网页进行爬取了，而且我们也对其中的标题，说明和URL进行了更结构化的解析。

现在，我们可以用Scrapy库中的所有功能来编写一个更健壮的爬虫程序了。

### 目标项

到目前为止，我们都只是将爬取所得的内容打印在标准输出上或转储到文件中。 其实我们还有一个更好的选择：就是在每次编写爬虫程序的时候为其定义一个items.py。这样做的好处是我们可以在自己的解析方法内使用这些目标项，并以任意的数据格式来输出，譬如XML，JSON或CSV等。所以，下面我们可以回到之前的爬虫程序中为其添加一个目标项类，其功能如下：

>>> fromscrapy.item import Item, Field

>>> class NewsItem(scrapy.Item):

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

>>> # name = scrapy.Field()

>>> pass

接下来，我们可以为该类添加如下三个不同的字段：

>>> from scrapy.item import Item, Field

>>> class NewsItem(Item):

>>> title = Field()

>>> link = Field()

>>> desc = Field()

如您所见，我们用field()添加了title、link和desc这三个字段。在我们放置完字段后，上述蜘蛛程序的解析方法就可以被修改为parse\_news\_item()了，因为它不会再将被解析的字段转储到某个文件中了，它现在使用的是用目标项对象。

接下来要定义的是Rule()方法，这是一种用于指定后续要爬取何种URL的方式。Rule()方法中提供了SgmlLinkExtractor()，这是一种用于定义URL模式的方式，它决定了我们能从被爬取网页中提取出哪些URL。除此之外，Rule()方法还提供了一个回调方法，该方法通常会返回一个指针，我们的蜘蛛程序会根据它来查找相关的解析方法，在我们这里就是parse\_news\_item()。如果目标可以有不同的解析方式，那么我们也可以设置多个规则和解析方法。另外，Rule()方法还有一个布尔参数，我们可以用它来指定是否使用该规则去提取每个响应后面的链接。在回调函数为None的情况下，其默认值为True：否则默认为False。

在这里，我们需要重点注意的是Rule()方法本身不是用来解析的。因为其默认回调的 方法名是parse()，如果我们要使用它，实际上就是要覆盖掉它的默认实现，并且可以以此来停止蜘蛛程序的爬取功能。接下来，我们将注意力转向下面的代码，来具体理解上面所提到的这些方法和参数：

>>> 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

## 生成网站地图的蜘蛛程序

如果相关网站提供了sitemap.xml，那么使用SiteMapSpider来对该网站进行爬取无疑是一种更好的选择。

在有给定sitemap.xml的情况下，蜘蛛程序所解析的是由网站自身提供的URL。这是无疑是一种更优雅的爬取方式，也会是一种更良好的实践：

>>> 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

在上面这段代码中，我们为每个产品目录都编写了一个相应的解析方法。如果我们想建立一个价格汇总/比对程序的话，这显然是个不错的用例。在这里，我们可能要针对不同产品的不同属性进行解析，例如对于电子类产品，我们可能要收集的是它的技术规格、附件以及价格信息; 而对于服装类产品，我们可能就更关系它们的大小和颜色。建议您找一个零售类网站来亲自尝试一下，在shell中通过相关的获取模式来手机不同目标项的的大小、颜色与价格信息。如果我们这样做了，就等于在一个良好的状态下写出了第一个符合工业标准的蜘蛛程序。

在某些情况下，我们想要爬取的网站必须要先行登录，然后才能进入该网站的某些部分。Scrapy库目前对此也有了一个解决方法。他们实现了FormRequest对象，该对象将更多地以POST的形式来呼叫HTTP服务器并获得响应。下面，我们就通过具体的蜘蛛程序代码来深入地了解一下：

>>> 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

对于只需要输入用户名和密码而无需任何验证码的网站，我们只需在上述代码中添加进特定的详细登录信息即可。这也是解析方法的一部分，因为大多数情况下我们都想要在首页中进行登录的。在完成登录之后，我们就可以针对目标项以及其他细节信息来编写属于自己的回调方法after\_login()了。

## 目标项管道

下面，让我们更多地来谈一谈关于目标项的后续处理。在这方面，Scrapy库提供了一种为目标项定义管道的方式，我们可以通过这种方式里定义相关的目标项后续要执行怎样的处理。这是一种非常有条不紊的、优秀的程序设计思维。

如果我们想要对之前所收集的目标项进行处理，譬如移除干扰词、执行大小写转换以及需要对目标中某个值进行的其它处理，例如我们想根据出生日期（DOB）来计算年龄，或者根据原价格来计算折扣，这些都需要我们来构建属于自己的目标项管道。而且到最后，我们可能还需要通过这些管道将目标项各自转储到某个文件中。

下面我们就来看看这种方式的具体实现步骤：

1. 首先，我们需要在setting.py中定义一个目标项管道：

ITEM\_PIPELINES = {

'myproject.pipeline.CleanPipeline': 300,

'myproject.pipeline.AgePipeline': 500,

'myproject.pipeline.DuplicatesPipeline: 700,

'myproject.pipeline.JsonWriterPipeline': 800,

}

1. 然后，我们来编写一个类，清理一下这些目标项：

>>>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

1. 接着，我们需要根据出生日期来推导年龄，这里将会用到Python的日期函数：

>>>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

1. 我们还需要移除掉重复的内容。由于Python的set()数据结构可以确保其项目值的唯一性，所以我们可以用Scrapy库来创建一个管道，下面是其实现代码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

1. 最后，我们要用JsonWriterPipeline.py中定义的管道将目标项写入到相关的JSON文件中：

>>>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

我们鼓励读者可以学习一些简单的蜘蛛程序实现，然后尝试着用它们来构建出一些很酷的应用序。下面是我本人所推荐的参考链接：

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

## 本章小结

在这一章中，我们学习了另一个非常优秀的Python程序库。从现在起，我们可以不必再依赖别人来数据了。因为我们已经学会了如何编写一个非常复杂的爬虫系统，以及如何编写一个专注于某类信息的蜘蛛程序。总之在本章，我们看到了从主系统中抽象出目标项逻辑的具体方法，以及如何为一些最常见的用例来编写特定的蜘蛛程序。在此过程中，我们了解了一些在实现自定义蜘蛛程序时最常用到的设置，我们也写了一些可重用的、复杂的解析方法。除此之外，我们还对选择器进行了非常深入的了解，知道了如何用手动的方式来确定所需的、特定的目标项属性，我们也可以通过Firebug这样的工具来进一步实际地了解选择器。最后一个同样重要的建议是，请务必遵守您所爬取网站的安全指南。

在下一章中，我们将会与您探讨一些基本的Python库在自然语言处理和机器学习领域的使用情况。

1. 译者注：从后文看，作者似乎很喜欢将蜘蛛和爬虫这两个词混着用，所以提前说明一下倒也是个办法。 [↑](#footnote-ref-1)
2. 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-2)