第 3 章 网络数据采集

网络数据采集是一种重要的数据采集方式，网络爬虫是用于网络数据采集的关键技术，它是一种按照一定的规则自动地抓取万维网信息的程序或者脚本，已经被广泛用于互联网搜索引擎或其他需要网络数据的企业，它可以自动采集所有其能够访问到的页面内容，以获取或更新这些网站的内容。

本章内容首先介绍网络爬虫的基本概念，包括什么是网络爬虫、网络爬虫的类型以及反爬机制，然后介绍一些网页基础知识，接下来介绍如何使用 Python 实现 HTTP 请求，如何定制 requests 以及如何解析网页，最后，给出 3 个网络爬虫的具体实例。

3.1 网络爬虫概述

本节介绍什么是网络爬虫、网络爬虫的类型和反爬机制。

3.1.1 什么是网络爬虫

网络爬虫是一个自动提取网页的程序，它为搜索引擎从万维网上下载网页，是搜索引擎的重要组成部分。如图 3-1 所示，爬虫从一个或若干个初始网页的 URL 开始，获得初始网页上的 URL，在抓取网页的过程中，不断从当前页面上抽取新的 URL 放入队列，直到满足系统的一定停止条件。实际上，网络爬虫的行为和人们访问网站的行为是类似的。举个例子，比如用户平时到天猫商城购物（PC 端），他的整个活动过程就是打开浏览器→搜索天猫商城→单击链接进入天猫商城→选择所需商品类目（站内搜索）→浏览商品（价格、详情参数、评论等）→单击链接→进入下一个商品页面……周而复始。现在，这个过程不再由用户自己手动去完成，而是由网络爬虫自动去完成。

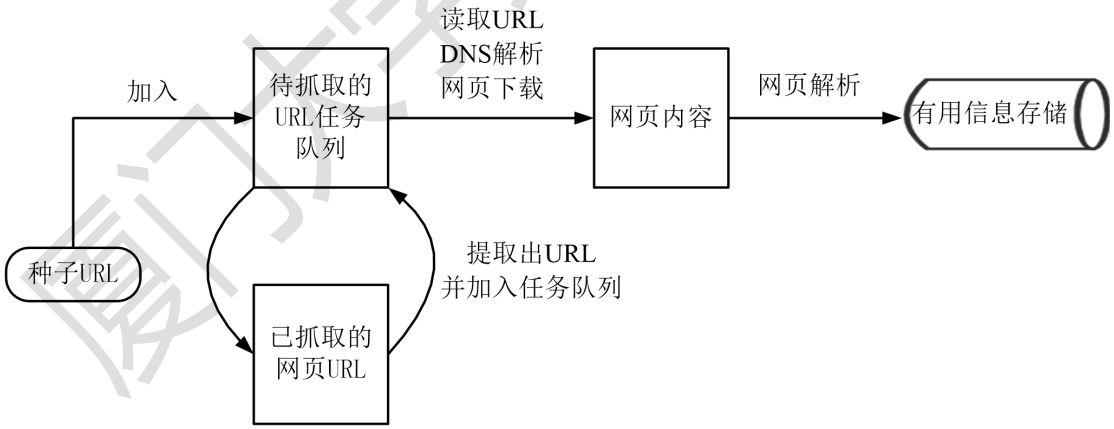

图 3-1 网络爬虫的工作原理

3.1.2 网络爬虫的类型

网络爬虫的类型可以分为通用网络爬虫、聚焦网络爬虫、增量式网络爬虫、深层网络爬虫。

（1）通用网络爬虫。通用网络爬虫又称“全网爬虫（Scalable Web Crawler）”，爬行对

象从一些种子 URL 扩充到整个 Web，该架构主要为门户站点搜索引擎和大型 Web 服务提供商采集数据。通用网络爬虫的结构大致可以包括页面爬行模块 、页面分析模块、链接过滤模块、页面数据库、URL 队列和初始 URL 集合。为提高工作效率，通用网络爬虫会采取一定的爬行策略。常用的爬行策略有：深度优先策略和广度优先策略。

（2）聚焦网络爬虫。聚焦网络爬虫（Focused Crawler）又称“主题网络爬虫（Topical Crawler）”，是指选择性地爬行那些与预先定义好的主题相关页面的网络爬虫。和通用网络爬虫相比，聚焦爬虫只需要爬行与主题相关的页面，极大地节省了硬件和网络资源，保存的页面也由于数量少而更新快，还可以很好地满足一些特定人群对特定领域信息的需求。聚焦网络爬虫的工作流程较为复杂，需要根据一定的网页分析算法过滤与主题无关的链接，保留有用的链接并将其放入等待抓取的 URL 队列。然后，它将根据一定的搜索策略从队列中选择下一步要抓取的网页 URL，并重复上述过程，直到达到系统的某一条件时停止。另外，所有被爬虫抓取的网页将会被系统存储，进行一定的分析、过滤，并建立索引，以便用于之后的查询和检索；对于聚焦网络爬虫来说，这一过程所得到的分析结果还可能对以后的抓取过程给出反馈和指导。聚焦网络爬虫常用的策略包括：基于内容评价的爬行策略、基于链接结构评价的爬行策略、基于增强学习的爬行策略和基于语境图的爬行策略。

（3）增量式网络爬虫。增量式网络爬虫（Incremental Web Crawler）是指对已下载网页采取增量式更新和只爬行新产生的或者已经发生变化网页的爬虫，它能够在一定程度上保证所爬行的页面是尽可能新的页面。和周期性爬行和刷新页面的网络爬虫相比，增量式爬虫只会在需要的时候爬行新产生或发生更新的页面，并不重新下载没有发生变化的页面，可有效减少数据下载量，及时更新已爬行的网页，减小时间和空间上的耗费，但是增加了爬行算法的复杂度和实现难度。增量式爬虫有两个目标：保持本地页面集中存储的页面为最新页面和提高本地页面集中页面的质量。为实现第一个目标，增量式爬虫需要通过重新访问网页来更新本地页面集中页面的内容。为了实现第二个目标，增量式爬虫需要对网页的重要性排序，常用的策略包括广度优先策略和 PageRank 优先策略等。

（4）深层网络爬虫。深层网络爬虫将 Web 页面按存在方式分为表层网页（Surface Web）和深层网页（Deep Web，也称 Invisible Web Page 或 Hidden Web）。表层网页是指传统搜索引擎可以索引的页面，以超链接可以到达的静态网页为主构成的 Web 页面。深层网页是那些大部分内容不能通过静态链接获取的、隐藏在搜索表单后的、只有用户提交一些关键词才能获得的 Web 页面。深层网络爬虫体系结构包含 6 个基本功能模块（爬行控制器、解析器、表单分析器、表单处理器、响应分析器、LVS 控制器）和两个爬虫内部数据结构（URL 列表、LVS 表）。

3.1.3 反爬机制

为什么会有反爬机制？原因主要有两点：第一，在大数据时代，数据是十分宝贵的财富，很多企业不愿意让自己的数据被别人免费获取，因此，很多企业都为自己的网站运用了反爬机制，防止网页上的数据被爬走；第二，简单低级的网络爬虫，数据采集速度快，伪装度低，如果没有反爬机制，它们可以很快地抓取大量数据，甚至因为请求过多，造成网站服务器不能正常工作，影响了企业的业务开展。

反爬机制也是一把双刃剑，一方面可以保护企业网站和网站数据，但是，另一方面，如果反爬机制过于严格，可能会误伤到真正的用户请求，也就是真正用户的请求被错误当成网络爬虫而被拒绝访问。如果既要和“网络爬虫”死磕，又要保证很低的误伤率，那么又会增加网站研发的成本。

通常而言，伪装度高的网络爬虫，速度慢，对服务器造成的负担也相对较小。所以，网

站反爬的重点也是针对那种简单粗暴的数据采集。有时反爬机制也会允许伪装度高的网路爬虫获得数据，毕竟伪装度很高的数据采集与真实用户请求没有太大差别。

3.2 网页基础知识

在学习网络爬虫相关知识之前，需要了解一些基本的网页知识，包括超文本、HTML、HTTP 等。

3.2.1 超文本和 HTML

超文本（Hypertext）是指使用超链接的方法，把文字和图片信息相互联结，形成具有相关信息的体系。超文本的格式有很多，目前最常使用的是超文本标记语言 HTML（Hyper Text Markup Language），我们平时在浏览器里面看到的网页就是由 HTML 解析而成的。下面是网页文件 web_demo.html 的 HTML 源代码：

```
<html>
<head><title>搜索指数</title></head>
<body>
<table>
<tr><td>排名</td><td>关键词</td><td>搜索指数</td></tr>
<tr><td>1</td><td>大数据</td><td>187767</td></tr>
<tr><td>2</td><td>云计算</td><td>178856</td></tr>
<tr><td>3</td><td>物联网</td><td>122376</td></tr>
</table>
</body>
</html>
```

使用网页浏览器（比如 IE、Firefox 等）打开这个网页文件，就会看到如图 3-2 所示的网页内容。

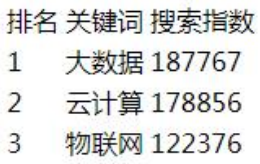

图 3-2 网页文件显示效果

3.2.2 HTTP

HTTP 是由万维网协会（World Wide Web Consortium）和 Internet 工作小组 IETF（Internet Engineering Task Force）共同制定的规范。HTTP 的全称是“Hyper Text Transfer Protocol”，中文名叫做“超文本传输协议”。HTTP 协议是用于从网络传输超文本数据到本地浏览器的传送协议，它能保证高效而准确地传送超文本内容。

HTTP 是基于“客户端/服务器”架构进行通信的，HTTP 的服务器端实现程序有 httpd、nginx 等，客户端的实现程序主要是 Web 浏览器，例如 Firefox、Internet Explorer、Google Chrome、

Safari、Opera 等。Web 浏览器和 Web 服务器之间可以通过 HTTP 进行通信。

一个典型的 HTTP 请求过程如下（如图 3-3 所示）：

（1）用户在浏览器中输入网址，比如 http://dblab.xmu.edu.cn，浏览器向网页服务器发起请求；

（2）网页服务器接收用户访问请求，处理请求，产生响应（即把处理结果以 HTML 形式返回给浏览器）；

（3）浏览器接收来自网页服务器的 HTML 内容，进行渲染以后展示给用户。

图 3-3 一个典型的 HTTP 请求过程

3.3 用 Python 实现 HTTP 请求

在网络数据采集中，读取 URL、下载网页是网络爬虫必备而又关键的功能，而这两个功能必然离不开与 HTTP 打交道。本节介绍用 Python 实现 HTTP 请求的常见的 3 种方式：urllib、urllib3 和 requests。

3.3.1 urllib 模块

urllib 是 Python 自带模块，该模块提供了一个 urlopen()方法，通过该方法指定 URL 发送 HTTP 请求来获取数据。urllib 提供了多个子模块，具体的模块名称与功能如表 3-1 所示。

表 3-1 urllib 中的子模块

模块名称	功能
urllib.request	该模块定义了打开 URL（主要是 HTTP）的方法和类，如身份验证、重定向和 cookie 等
urllib.error	该模块中主要包含异常类，基本的异常类是 URLError
urllib.parse	该模块定义的功能分为两大类：URL 解析和 URL 引用
urllib.robotparser	该模块用于解析 robots.txt 文件

下面是通过 urllib.request 模块实现发送 GET 请求获取网页内容的实例：

```
>>> import urllib.request
>>> response=urllib.request.urlopen("http://www.baidu.com")
>>> html=response.read()
>>> print(html)
```

下面是通过 urllib.request 模块实现发送 POST 请求获取网页内容的实例：

```
>>> import urllib.parse
>>> import urllib.request
>>> # 1.指定 url
>>> url = 'https://fanyi.baidu.com/sug'
```

```
>>> # 2.发起 POST 请求之前，要处理 POST 请求携带的参数
>>> # 2.1 将 POST 请求封装到字典
>>> data = {'kw':'苹果',}
>>> # 2.2 使用 parse 模块中的 urlencode(返回值类型是字符串类型)进行编码处理
>>> data = urllib.parse.urlencode(data)
>>> # 将步骤 2.2 的编码结果转换成 byte 类型
>>> data = data.encode()
>>> # 3.发起 POST 请求:urlopen 函数的 data 参数表示的就是经过处理之后的 POST 请求携带的参数
>>> response = urllib.request.urlopen(url=url,data=data)
>>> data = response.read()
>>> print(data)
b'{"errno":0,"data":[{"k":"\\u82f9\\u679c","v":"\\u540d.
   apple"},{"k":"\\u82f9\\u679c\\u56ed","v":"apple
   grove"},{"k":"\\u82f9\\u679c\\u5934","v":"apple
   head"},{"k":"\\u82f9\\u679c\\u5e72","v":"[\\u533b]dried
   apple"},{"k":"\\u82f9\\u679c\\u6728","v":"applewood"}]}'
```

把上面 print(data)执行的结果，拿到 JSON 在线格式校验网站（http://www.bejson.com/）进行处理，使用“Unicode 转中文”功能可以得到如下结果：

b'{"errno":0,"data":[{"k":"\ 苹 \ 果 ","v":"\ 名 . apple"},{"k":"\ 苹 \ 果 \ 园 ","v":"apple grove"},{"k":"\苹\果\头","v":"apple head"},{"k":"\苹\果\干","v":"[\医]dried apple"},{"k":"\苹\果\木","v":"applewood"}]}'

3.3.2 urllib3 模块

urllib3 是一个功能强大、条理清晰、用于 HTTP 客户端的 Python 库，许多 Python 的原生系统已经开始使用 urllib3。urllib3 提供了很多 Python 标准库里所没有的重要特性，包括：线程安全、连接池、客户端 SSL/TLS 验证、文件分部编码上传、协助处理重复请求和 HTTP 重定位、支持压缩编码、支持 HTTP 和 SOCKS 代理、100%测试覆盖率等。

在使用 urllib3 之前，需要打开一个 cmd 窗口使用如下命令进行安装：

```
> pip install urllib3
```

下面是通过 GET 请求获取网页内容的实例：

```
>>> import urllib3
>>> #需要一个 PoolManager 实例来生成请求，由该实例对象处理与线程池的连接以及线程安全的所有细节，不需要任何人为操作
>>> http = urllib3.PoolManager()
>>> response = http.request('GET','http://www.baidu.com')
>>> print(response.status)
>>> print(response.data)
```

下面是通过 POST 请求获取网页内容的实例：

```
>>> import urllib3
>>> http = urllib3.PoolManager()
```

```
>>> response = http.request('POST',
                    'https://fanyi.baidu.com/sug'
                    ,fields={'kw':'苹果',})
>>> print(response.data)
```

3.3.3 requests 模块

requests 是一个非常好用的 HTTP 请求库，可用于网络请求和网络爬虫等。
在使用 requests 之前，需要打开一个 cmd 窗口使用如下命令进行安装：
> pip install requests

以 GET 请求方式为例，打印多种请求信息的代码如下：

```
>>> import requests
>>> response = requests.get('http://www.baidu.com')  #对需要爬取的网页发送请求
>>> print('状态码:',response.status_code)  #打印状态码
>>> print('url:',response.url)  #打印请求 url
>>> print('header:',response.headers)  #打印头部信息
>>> print('cookie:',response.cookies)  #打印 cookie 信息
>>> print('text:',response.text)  #以文本形式打印网页源码
>>> print('content:',response.content)  #以字节流形式打印网页源码
```

以 POST 请求方式发送 HTTP 网页请求的示例代码如下：

```
>>> import requests
>>> #导入模块
>>> import requests
>>> #表单参数
>>> data = {'kw':'苹果',}
>>> #对需要爬取的网页发送请求
>>> response = requests.post('https://fanyi.baidu.com/sug',data=data)
>>> #以字节流形式打印网页源码
>>> print(response.content)
```

3.4 定制 requests

通过前面的介绍，我们已经可以爬取网页的 HTML 代码数据了，但有时候我们需要对 requests 的参数进行设置才能顺利获取我们需要的数据，包括传递 URL 参数、定制请求头、发送 POST 请求和设置超时等。

3.4.1 传递 URL 参数

为了请求特定的数据，我们需要在 URL（Uniform Resource Locator）的查询字符串中加入一些特定数据。这些数据一般会跟在一个问号后面，并且以键值对的形式放在 URL 中。

在 requests 中，我们可以直接把这些参数保存在字典中，用 params 构建到 URL 中。具体实例如下：

```
>>> import requests
>>> base_url = 'http://httpbin.org'
>>> param_data = {'user':'xmu','password':'123456'}
>>> response = requests.get(base_url+'/get',params=param_data)
>>> print(response.url)
http://httpbin.org/get?user=xmu&password=123456
>>> print(response.status_code)
200
```

3.4.2 定制请求头

在爬取网页的时候，输出的信息中有时候会出现“抱歉，无法访问”等字眼，这就是禁止爬取，需要通过定制请求头 Headers 来解决这个问题。定制 Headers 是解决 requests 请求被拒绝的方法之一，相当于我们进入这个网页服务器，假装自己本身在爬取数据。请求头 Headers 提供了关于请求、响应或其他发送实体的消息，如果没有定制请求头或请求的请求头和实际网页不一致，就可能无法返回正确结果。

获取一个网页的 Headers 的方法如下：使用 360、火狐或谷歌浏览器打开一个网址（比如“http://httpbin.org/”），在网页上单击鼠标右键，在弹出的菜单中选择“查看元素”，然后刷新网页，再按照如图 3-4 所示的步骤，先点击“Network”选项卡，再点击“Doc”，接下来点击“Name”下方的网址，就会出现类似如下的 Headers 信息：

User-Agent:Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36

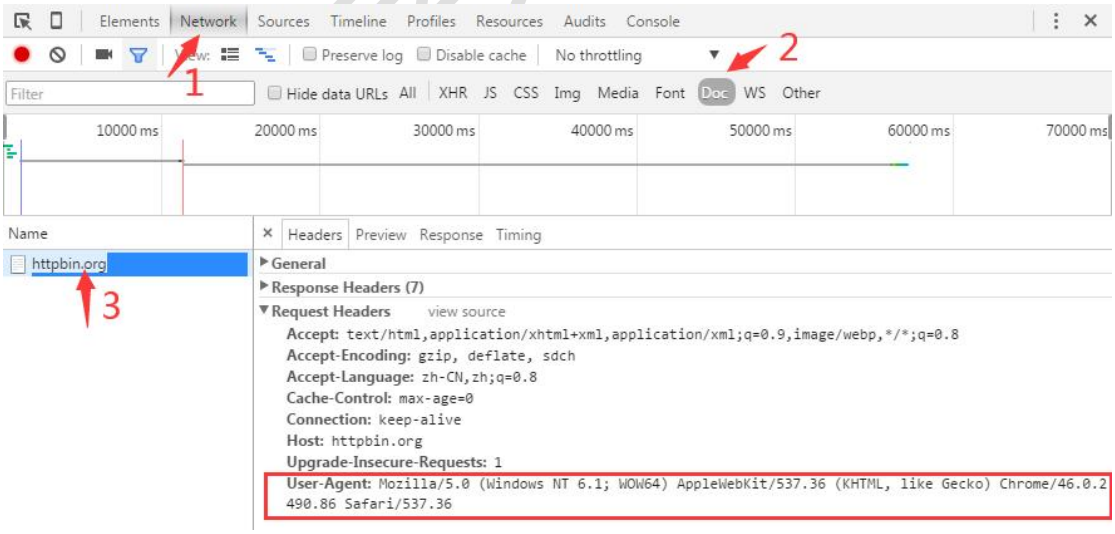

图 3-4 查看请求头 Headers

Headers 中有很多内容，主要常用的就是“User-Agent”和“Host”，它们是以键对的形式呈现的，如果把“User-Agent”以字典键值对形式作为 Headers 的内容，往往就可以顺利爬取网页内容。

下面是添加了 Headers 信息的网页请求过程：

```
>>> import requests
```

```
>>> url='http://httpbin.org'
>>> #  创建头部信息
>>> headers={'User-Agent':'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36'}
>>> response = requests.get(url,headers=headers)
>>> print(response.content)
```

3.4.3 网络超时

网络请求不可避免会遇上请求超时的情况，这个时候，网络数据采集的程序会一直运行等待进程，导致网络数据采集程序不能很好地顺利执行。因此，可以为 requests 的 timeout 参数设定等待秒数，如果服务器在指定时间内没有应答就返回异常。具体代码如下：

```
# time_out.py
import requests
from requests.exceptions import ReadTimeout,ConnectTimeout
try:
    response = requests.get("http://www.baidu.com", timeout=0.5)
    print(response.status_code)
except ReadTimeout or ConnectTimeout:
    print('Timeout')
```

3.5 解析网页

爬取到一个网页之后，需要对网页数据进行解析，获得我们需要的数据内容。BeautifulSoup 是一个 HTML/XML 的解析器，主要功能是解析和提取 HTML/XML 数据。本节介绍 BeautifulSoup 的使用方法。

3.5.1 BeautifulSoup 简介

BeautifulSoup 提供一些简单的、Python 式的函数来处理导航、搜索、修改分析树等。BeautifulSoup 通过解析文档为用户提供需要抓取的数据，因为简单，所以不需要多少代码就可以写出一个完整的应用程序。BeautifulSoup 自动将输入文档转换为 Unicode 编码，输出文档转换为 UTF-8 编码。BeautifulSoup3 已经停止开发，目前推荐使用 BeautifulSoup4，不过它已经被移植到 bs4 当中了，所以，在使用 BeautifulSoup4 之前，需要安装 bs4：

```
> pip install bs4
```

使用 BeautifulSoup 解析 HTML 比较简单，API 非常人性化，支持 CSS 选择器、Python 标准库中的 HTML 解析器，也支持 lxml 的 XML 解析器和 HTML 解析器，此外还支持 html5lib 解析器，表 3-2 给出了每个解析器的优缺点。

表 3-2 不同解析器的优缺点比较

解析器	用法	优点	缺点
Python 标准库	BeautifulSoup(markup,"html.parser")	Python 标准库	文档容错能力

		执行速度适中	差
lxml 的 HTML 解析器	BeautifulSoup(markup,"lxml")	速度快 文档容错能力强	需要安装 C 语言库
lxml 的 XML 解析器	BeautifulSoup(markup, "lxml-xml") BeautifulSoup(markup,"xml")	速度快 唯一支持 XML 的解析器	需要安装 C 语言库
html5lib	BeautifulSoup(markup, "html5lib")	兼容性好 以浏览器的方式解析文档 生成 HTML5 格式的文档	速度慢，不依赖外部扩展

总体而言，如果需要快速解析网页，建议使用 lxml 解析器；如果使用的 Python2.x 是 2.7.3 之前的版本，或者使用的 Python3.x 是 3.2.2 之前的版本，则要使用 html5lib 或 lxml 解析器，因为 Python 内建的 HTML 解析器不能很好地适应于这些老版本。

下面给出一个 BeautifulSoup 解析网页的简单实例，使用了 lxml 解析器，在使用之前，需要执行如下命令安装 lxml 解析器：

```
> pip install lxml
```

下面是实例代码：

```
>>> html_doc = """
<html><head><title>BigData Software</title></head>
<p class="title"><b>BigData Software</b></p>
<p class="bigdata">There are three famous bigdata softwares; and their names are
<a href="http://example.com/hadoop" class="software" id="link1">Hadoop</a>,
<a href="http://example.com/spark" class="software" id="link2">Spark</a> and
<a href="http://example.com/flink" class="software" id="link3">Flink</a>;
and they are widely used in real applications.</p>
<p class="bigdata">...</p>
"""
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(html_doc,"lxml")
>>> content = soup.prettify()
>>> print(content)
<html>
 <head>
  <title>
   BigData Software
  </title>
 </head>
 <body>
  <p class="title">
   <b>
    BigData Software
   </b>
  </p>
```

```
    <p class="bigdata">
     There are three famous bigdata softwares; and their names are
     <a class="software" href="http://example.com/hadoop" id="link1">
      Hadoop
     </a>
     ,
     <a class="software" href="http://example.com/spark" id="link2">
      Spark
     </a>
     and
     <a class="software" href="http://example.com/flink" id="link3">
      Flink
     </a>
     ;
    and they are widely used in real applications.
    </p>
    <p class="bigdata">
     ...
    </p>
   </body>
  </html>
```

如果要更换解析器，比如要使用 Python 标准库的解析器，只需要把上面的“soup = BeautifulSoup(html_doc,"lxml")”这行代码替换成如下代码即可：

```
soup = BeautifulSoup(html_doc,"html.parser")
```

3.5.2 BeautifulSoup 四大对象

BeautifulSoup 将复杂 HTML 文档转换成一个复杂的树形结构，每个节点都是 Python 对象，所有对象可以归纳为 4 种：Tag、NavigableString、BeautifulSoup、Comment。

1.Tag

Tag 就是 HTML 中的一个个标签，例如：

```
<title>BigData Software</title>
<a href="http://example.com/hadoop" class="software" id="link1">Hadoop</a>
```

上面的<title>、<a>等标签加上里面包括的内容就是 Tag，利用 soup 加标签名可以轻松地获取这些标签的内容。作为演示，我们可以继续执行以下代码：

```
>>> print(soup.a)
<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>
>>> print(soup.title)
<title>BigData Software</title>
```

Tag 有两个重要的属性，即 name 和 attrs。下面继续执行如下代码：

```
>>> print(soup.name)
[document]
>>> print(soup.p.attrs)
```

```
{'class': ['title']}
```

如果想要单独获取某个属性，比如要获取“class”属性的值，可以执行如下代码：

```
>>> print(soup.p['class'])
['title']
```

还可以利用 get 方法获得属性的值，代码如下：

```
>>> print(soup.p.get('class'))
['title']
```

2. NavigableString

NavigableString 对象用于操纵字符串。在网页解析时，已经得到了标签的内容以后，如果我们想获取标签内部的文字，则可以使用.string 方法，其返回值就是一个 NavigableString 对象，具体实例如下：

```
>>> print(soup.p.string)
BigData Software
>>> print(type(soup.p.string))
<class 'bs4.element.NavigableString'>
```

3. BeautifulSoup

BeautifulSoup 对象表示的是一个文档的全部内容，大部分时候，可以把它当作 Tag 对象，是一个特殊的 Tag。例如，可以分别获取它的类型、名称以及属性：

```
>>> print(type(soup.name))
<class 'str'>
>>> print(soup.name)
[document]
>>> print(soup.attrs)
{}
```

4.Comment

Comment 对象是一种特殊类型的 NavigableString 对象，输出的内容不包括注释符号。如果它处理不好，可能会对文本处理造成意想不到的麻烦。为了演示 Comment 对象，这里重新创建一个代码文件 bs4_example.py：

```
# bs4_example.py
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<p class="title"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!-- Elsie --></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc,"lxml")
print(soup.a)
print(soup.a.string)
print(type(soup.a.string))
```

该代码文件的执行结果如下：

```
<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>
Elsie
<class 'bs4.element.Comment'>
```

从上面执行结果可以看出，a 标签里的内容“<!-- Elsie -->”实际上是注释，但是使用语句 print(soup.a.string)输出它的内容以后会发现，它已经把注释符号去掉了，只输出了“Elsie”，所以这可能会给我们带来不必要的麻烦。另外我们打印输出它的类型，发现它是一个 Comment 类型。

通过上面的介绍，我们已经了解了 BeautifulSoup 的基本概念，现在的问题是，如何从 HTML 中找到我们关心的数据呢？BeautifulSoup 提供了两种方式，一种是遍历文档树，另一种是搜索文档树，通常把两者结合起来完成查找任务。

3.5.3 遍历文档树

遍历文档树就是从根节点 html 标签开始遍历，直到找到目标元素为止。

1.直接子节点

（1）.contents 属性

Tag 对象的.contents 属性可以将某个 Tag 的子节点以列表的方式输出，当然列表会允许用索引的方式来获取列表中的元素。下面是示例代码：

```
>>> html_doc = """
<html><head><title>BigData Software</title></head>
<p class="title"><b>BigData Software</b></p>
<p class="bigdata">There are three famous bigdata softwares; and their names are
<a href="http://example.com/hadoop" class="software" id="link1">Hadoop</a>,
<a href="http://example.com/spark" class="software" id="link2">Spark</a> and
<a href="http://example.com/flink" class="software" id="link3">Flink</a>;
and they are widely used in real applications.</p>
<p class="bigdata">...</p>
"""
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(html_doc,"lxml")
>>> print(soup.body.contents)
[<p class="title"><b>BigData Software</b></p>, '\n', <p class="bigdata">There are three famous bigdata softwares; and their names are
<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>,
<a class="software" href="http://example.com/spark" id="link2">Spark</a> and
<a class="software" href="http://example.com/flink" id="link3">Flink</a>;
and they are widely used in real applications.</p>, '\n', <p class="bigdata">...</p>, '\n']
```

可以使用索引的方式来获取列表中的元素：

```
>>> print(soup.body.contents[0])
<p class="title"><b>BigData Software</b></p>
```

（2）.children 属性

Tag 对象的.children 属性是一个迭代器，可以使用 for 循环进行遍历，代码如下：

```
>>> for child in soup.body.children:
        print(child)
```

上面代码的执行结果如下：

```
<p class="title"><b>BigData Software</b></p>


<p class="bigdata">There are three famous bigdata softwares; and their names are
<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>,
<a class="software" href="http://example.com/spark" id="link2">Spark</a> and
<a class="software" href="http://example.com/flink" id="link3">Flink</a>;
and they are widely used in real applications.</p>


<p class="bigdata">...</p>
```

2.所有子孙节点

在获取所有子孙节点时，可以使用.descendants 属性，与 Tag 对象的.children 和.contents 仅包含 Tag 对象的直接子节点不同，该属性是将 Tag 对象的所有子孙节点进行递归循环，然后生成生成器。示例代码如下：

```
>>>   for child in soup.descendants:
          print(child)
```

上面代码的执行结果较多，因此这里没有给出。在执行结果中可以发现，所有的节点都被打印出来了，先生成最外层的 html 标签，其次从 head 标签一个个剥离，依此类推。

3.节点内容

（1）Tag 对象内没有标签的情况。

```
>>> print(soup.title)
<title>BigData Software</title>
>>> print(soup.title.string)
BigData Software
```

（2）Tag 对象内有一个标签的情况。

```
>>> print(soup.head)
<head><title>BigData Software</title></head>
>>> print(soup.head.string)
BigData Software
```

（3）Tag 对象内有多个标签的情况。

```
>>> print(soup.body)
```

```
<body><p class="title"><b>BigData Software</b></p>
<p class="bigdata">There are three famous bigdata softwares; and their names are
<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>,
<a class="software" href="http://example.com/spark" id="link2">Spark</a> and
<a class="software" href="http://example.com/flink" id="link3">Flink</a>;
and they are widely used in real applications.</p>
<p class="bigdata">...</p>
</body>
```

从上面的执行结果中可以看出，body 标签内包含了多个 p 标签，这时如果使用.string 获取子节点内容，就会返回 None，代码如下：

```
>>> print(soup.body.string)
None
```

也就是说，如果 Tag 包含了多个子节点，Tag 就无法确定.string 应该调用哪个子节点的内容，因此.string 的输出结果是 None。这时应该使用.strings 属性或.stripped_strings 属性，它们获得的都是一个生成器，示例代码如下：

```
>>> print(soup.strings)
<generator object Tag._all_strings at 0x0000000002C4D190>
```

可以用 for 循环对生成器进行遍历，代码如下：

```
>>> for string in soup.strings:
        print(repr(string))
```

上面代码的执行结果如下：

```
'BigData Software'
'\n'
'BigData Software'
'\n'
'There are three famous bigdata softwares; and their names are\n'
'Hadoop'
',\n'
'Spark'
' and\n'
'Flink'
';\nand they are widely used in real applications.'
'\n'
'...'
'\n'
```

使用 Tag 对象的.stripped_strings 属性，可以获得去掉空白行的标签内的众多内容，示例代码如下：

```
>>> for string in soup.stripped_strings:
        print(string)
```

上面代码的执行结果如下：

```
BigData Software
BigData Software
There are three famous bigdata softwares; and their names are
Hadoop
,
Spark
and
Flink
;
and they are widely used in real applications.
...
```

4.直接父节点

使用 Tag 对象的.parent 属性可以获得父节点，使用 Tag 对象的.parents 属性可以获得从父到根的所有节点。

下面是标签的父节点：

```
>>> p = soup.p
>>> print(p.parent.name)
Body
```

下面是内容的父节点：

```
>>> content = soup.head.title.string
>>> print(content)
BigData Software
>>> print(content.parent.name)
title
```

Tag 对象的.parents 属性，得到的也是一个生成器：

```
>>> content = soup.head.title.string
>>> print(content)
BigData Software
>>> for parent in content.parents:
        print(parent.name)
```

上面语句的执行结果如下：

```
title
head
html
[document]
```

5.兄弟节点

可以使用 Tag 对象的.next_sibling 和.previous_sibling 属性分别获取下一个兄弟节点和获取上一个兄弟节点。需要注意的是，实际文档中 Tag 的.next_sibling 和.previous_sibling 属性通常是字符串或空白串，因为空白串或者换行符也可以被视作一个节点，所以得到的结果可

能是空白或者换行。示例代码如下：

```
>>> print(soup.p.next_sibling)
# 此处返回为空白
>>> print(soup.p.prev_sibling)
None  #没有前一个兄弟节点，返回 None
>>> print(soup.p.next_sibling.next_sibling)
```

上面这个语句的返回结果如下：

```
<p class="bigdata">There are three famous bigdata softwares; and their names are
<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>,
<a class="software" href="http://example.com/spark" id="link2">Spark</a> and
<a class="software" href="http://example.com/flink" id="link3">Flink</a>;
and they are widely used in real applications.</p>
```

6.全部兄弟节点

可以使用 Tag 对象的.next_siblings 和.previous_siblings 属性对当前的兄弟节点迭代输出。示例代码如下：

```
>>> for next in soup.a.next_siblings:
        print(repr(next))
```

执行结果如下：

```
',\n'
<a class="software" href="http://example.com/spark" id="link2">Spark</a>
' and\n'
<a class="software" href="http://example.com/flink" id="link3">Flink</a>
';\nand they are widely used in real applications.'
```

7.前后节点

Tag 对象的.next_element 和.previous_element 属性，用于获得不分层次的前后元素，示例代码如下：

```
>>> print(soup.head.next_element)
<title>BigData Software</title>
```

8. 所有前后节点

使用 Tag 对象的.next_elements 和.previous_elements 属性可以向前或向后解析文档内容，示例代码如下：

```
>>> for element in soup.a.next_elements:
        print(repr(element))
```

执行结果如下：

```
'Hadoop'
',\n'
<a class="software" href="http://example.com/spark" id="link2">Spark</a>
'Spark'
' and\n'
<a class="software" href="http://example.com/flink" id="link3">Flink</a>
```

```
'Flink'
';\nand they are widely used in real applications.'
'\n'
<p class="bigdata">...</p>
'...'
'\n'
```

3.5.4 搜索文档树

搜索文档树是通过指定标签名来搜索元素，另外还可以通过指定标签的属性值来精确定位某个节点元素，最常用的两个方法就是 find()和 find_all()，这两个方法在 BeatifulSoup 和 Tag 对象上都可以被调用。

1.find_all()

find_all()方法搜索当前 Tag 的所有 Tag 子节点，并判断是否符合过滤器的条件，它的函数原型是：

```
find_all( name , attrs , recursive , text , **kwargs )
```

find_all()的返回值是一个 Tag 组成的列表，方法调用非常灵活，所有的参数都是可选的。

（1）name 参数

name 参数可以查找所有名字为 name 的 Tag，字符串对象会被自动忽略掉。

①传入字符串

查找所有名字为 a 的 Tag，代码如下：

```
>>> print(soup.find_all('a'))
```

[Hadoop, Spark, Flink]

②传入正则表达式

如果传入正则表达式作为参数，BeautifulSoup 会通过正则表达式的 match()来匹配内容.下面例子中找出所有以 b 开头的标签，这表示<body>和<b>标签都应该被找到：

```
>>> import re
>>> for tag in soup.find_all(re.compile("^b")):
        print(tag)
```

执行结果如下：

```
<body><p class="title"><b>BigData Software</b></p>
<p class="bigdata">There are three famous bigdata softwares; and their names are
<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>,
<a class="software" href="http://example.com/spark" id="link2">Spark</a> and
<a class="software" href="http://example.com/flink" id="link3">Flink</a>;
and they are widely used in real applications.</p>
<p class="bigdata">...</p>
</body>
<b>BigData Software</b>
```

③传入列表

如果传入参数是列表，BeautifulSoup 会将与列表中任一元素匹配的内容返回。下面代码

找到文档中所有<a>标签和标签：

```
>>> print(soup.find_all(["a","b"]))
```

[BigData Software, Hadoop, Spark, Flink]

④传入 True

传入 True 可以找到所有的标签。下面的例子在文档树中查找所有包含 id 属性的标签，无论 id 的值是什么：

```
>>> print(soup.find_all(id=True))
```

[Hadoop, Spark, Flink]

⑤传入方法

如果没有合适的过滤器，那么还可以定义一个方法，方法只接受一个元素参数，如果这个方法返回 True，表示当前元素匹配并且被找到，如果不是则返回 False。下面方法对当前元素进行校验，如果包含 class 属性却不包含 id 属性，那么将返回 True：

```
>>> def has_class_but_no_id(tag):
        return tag.has_attr('class') and not tag.has_attr('id')
```

将这个方法作为参数传入 find_all()方法，将得到所有 标签：

```
>>> print(soup.find_all(has_class_but_no_id))
```

[BigData Software , There are three famous bigdata softwares; and their names are

Hadoop,

Spark and

Flink;

and they are widely used in real applications. , ...]

（2）keyword 参数

通过 name 参数是搜索 Tag 的标签类型名称，如 a、head、title 等。如果要通过标签内属性的值来搜索，要通过键值对的形式来指定，实例如下：

```
>>> import re
>>> print(soup.find_all(id='link2'))
```

[Spark]

```
>>> print(soup.find_all(href=re.compile("spark")))
```

[Spark]

使用多个指定名字的参数可以同时过滤 Tag 的多个属性：

```
>>> soup.find_all(href=re.compile("hadoop"), id='link1')
```

[Hadoop]

如果指定的 key 是 Python 的关键词，则后面需要加下划线：

```
>>> print(soup.find_all(class_="software"))
```

[Hadoop, Spark, Flink]

（3）text 参数

text 参数的作用和 name 参数类似，但是 text 参数的搜索范围是文档中的字符串内容（不包含注释），并且是完全匹配，当然也接受正则表达式、列表、True。实例如下：

```
>>> import re
>>> print(soup.a)
<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>
>>> print(soup.find_all(text="Hadoop"))
['Hadoop']
>>> print(soup.find_all(text=["Hadoop", "Spark", "Flink"]))
['Hadoop', 'Spark', 'Flink']
>>> print(soup.find_all(text="bigdata"))
[]
>>> print(soup.find_all(text="BigData Software"))
['BigData Software', 'BigData Software']
>>> print(soup.find_all(text=re.compile("bigdata")))
['There are three famous bigdata softwares; and their names are\n']
```

（4）limit 参数

可以通过 limit 参数来限制使用 name 参数或者 attrs 参数过滤出来的条目的数量，实例如下：

```
>>> print(soup.find_all("a"))
[<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>, <a class="software" href="http://example.com/spark" id="link2">Spark</a>, <a class="software" href="http://example.com/flink" id="link3">Flink</a>]
>>> print(soup.find_all("a",limit=2))
[<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>, <a class="software" href="http://example.com/spark" id="link2">Spark</a>]
```

（5）recursive 参数

调用 Tag 的 find_all()方法时，BeautifulSoup 会检索当前 Tag 的所有子孙节点，如果只想搜索 Tag 的直接子节点，可以使用参数 recursive=False，实例如下：

```
>>> print(soup.body.find_all("a",recursive=False))
[]
```

在这个例子中，a 标签都是在 p 标签内的，所以在 body 的直接子节点下搜索 a 标签是无法匹配到 a 标签的。

2.find()

find()与 find_all()的区别是，find_all()将所有匹配的条目组合成一个列表，而 find()仅返回第一个匹配的条目，除此以外，二者的用法都相同。

3.5.5 CSS 选择器

BeautifulSoup 支持大部分的 CSS 选择器，在 Tag 或 BeautifulSoup 对象的 select()方法中传入字符串参数，即可使用 CSS 选择器的语法找到标签。

①通过标签名查找

```
>>> print(soup.select('title'))
[<title>BigData Software</title>]
>>> print(soup.select('a'))
```

[Hadoop, Spark, Flink]

```
>>> print(soup.select('b'))
[<b>BigData Software</b>]
```

②通过类名查找

```
>>> print(soup.select('.software'))
```

[Hadoop, Spark, Flink]

③通过 id 名查找

```
>>> print(soup.select('#link1'))
[<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>]
```

④组合查找

```
>>> print(soup.select('p #link1'))
[<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>]
>>> print(soup.select("head > title"))
[<title>BigData Software</title>]
>>> print(soup.select("p > a:nth-of-type(1)"))
[<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>]
>>> print(soup.select("p > a:nth-of-type(2)"))
[<a class="software" href="http://example.com/spark" id="link2">Spark</a>]
>>> print(soup.select("p > a:nth-of-type(3)"))
[<a class="software" href="http://example.com/flink" id="link3">Flink</a>]
```

在上面的语句中，"p > a:nth-of-type(2)"的含义是：p 元素是某个父元素的子元素，选择子元素 p，且子元素 p 必须是其父元素下的第二个 p 元素。

⑤属性查找

查找时还可以加入属性元素，属性需要用中括号括起来，注意属性和标签属于同一节点，所以中间不能加空格，否则会无法匹配到。

```
>>> print(soup.select('a[class="software"]'))
```

[Hadoop, Spark, Flink]

```
>>> print(soup.select('a[href="http://example.com/hadoop"]'))
[<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>]
>>> print(soup.select('p a[href="http://example.com/hadoop"]'))
[<a class="software" href="http://example.com/hadoop" id="link1">Hadoop</a>]
```

以上的 select 方法返回的结果都是列表形式，可以以遍历的形式进行输出，然后用 get_text()方法来获取它的内容，实例如下：

```
>>> print(type(soup.select('title')))
<class 'bs4.element.ResultSet'>
>>> print(soup.select('title')[0].get_text())
BigData Software
```

```
>>> for title in soup.select('title'):
        print(title.get_text())
```

上面语句的执行结果如下：

```
BigData Software
```

3.6 综合实例

为了深化对前面知识的理解，这里给出 3 个综合实例，包括采集网页数据并解析成指定格式、采集网页数据保存到文本文件、采集网页数据保存到 MySQL 数据库。

3.6.1 采集网页数据并解析成指定的格式

采集百度实时热点网页（http://top.baidu.com/buzz?b=1&fr=20811）上的数据，并解析成指定的格式显示到屏幕上。可以打开一个浏览器，访问要爬取的网页，然后在浏览器中查看网页源代码，在源代码中找到搜索指数、排名、标题所在的位置，总结出它们共同的特征，就可以将它们全部提取出来了，具体实现代码如下：

```
# baidu_hot.py
import requests
from bs4 import BeautifulSoup

#  请求网页
def request_page(url,headers):
    response = requests.get(url,headers=headers)
    response.encoding = response.apparent_encoding
    return response.text

#  解析网页
def parse_page(html):
    soup = BeautifulSoup(html,'html.parser')
    all_topics=soup.find_all('tr')[1:]
    for each_topic in all_topics:
        topic_times = each_topic.find('td',class_='last')      #搜索指数
        topic_rank = each_topic.find('td',class_='first')      #排名
        topic_name = each_topic.find('td',class_='keyword')   #标题
        if topic_rank != None and topic_name!=None and topic_times!=None:
                topic_rank = each_topic.find('td',class_='first').get_text().replace('
','').replace('\n','')
                topic_name = each_topic.find('td',class_='keyword').get_text().replace('
','').replace('\n','')
                topic_times = each_topic.find('td',class_='last').get_text().replace('
','').replace('\n','')
            tplt = "排名：{0:^4}\t 标题：{1:{3}^15}\t 热度：{2:^8}"
            print(tplt.format(topic_rank,topic_name,topic_times,chr(12288)))
```

```
25
26  if __name__=='__main__':
27          url = 'http://top.baidu.com/buzz?b=1&fr=20811'
28          headers = {'User-Agent':'Mozilla/5.0'}
29          html = request_page(url,headers)
30          parse_page(html)
```

3.6.2 采集网页数据保存到文本文件

访问古诗文网站（https://so.gushiwen.org/mingju/），会显示如图 3-5 所示的页面，里面包含了很多名句，点击某一个名句（比如“山有木兮木有枝，心悦君兮君不知”），就会出现完整的古诗（如图 3-6 所示）。

图 3-5 名句页面

图 3-6 完整古诗页面

下面编写网络爬虫程序，爬取名句页面的内容，保存到一个文本文件中，然后，再爬取每个名句的完整古诗页面，把完整古诗保存到一个文本文件中。可以打开一个浏览器，访问要爬取的网页，然后在浏览器中查看网页源代码，找到诗句内容所在的位置，总结出它们共同的特征，就可以将它们全部提取出来了，具体实现代码如下：

```
# --*-- encoding: utf-8 --*--
# parse_poem.py

import requests
from bs4 import BeautifulSoup
import time



# 函数 1：请求网页
def page_request(url, ua):
    response = requests.get(url, headers=ua)
    html = response.content.decode('utf-8')
    return html



# 函数 2：解析网页
def page_parse(html):
    soup = BeautifulSoup(html, 'lxml')
    title = soup('title')
    # 诗句内容：诗句+出处+链接
    info = soup.select('body > div.main3 > div.left > div.sons > div.cont')
    # 诗句链接
    sentence = soup.select('div.left > div.sons > div.cont > a:nth-of-type(1)')
    sentence_list = []
```

```
    href_list = []

    for i in range(len(info)):
        curInfo = info[i]
        poemInfo = ''
        poemInfo = poemInfo.join(curInfo.get_text().split('\n'))
        sentence_list.append(poemInfo)
        href = sentence[i].get('href')
        href_list.append("https://so.gushiwen.org" + href)
    return [href_list, sentence_list]


def save_txt(info_list):
    import json
    with open(r'sentence.txt', 'a', encoding='utf-8') as txt_file:
        for element in info_list[1]:
            txt_file.write(json.dumps(element, ensure_ascii=False) + '\n\n')


# 子网页处理函数：进入并解析子网页/请求子网页
def sub_page_request(info_list):
    subpage_urls = info_list[0]
    ua = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/46.0.2490.86 Safari/537.36'}
    sub_html = []
    for url in subpage_urls:
        html = page_request(url, ua)
        sub_html.append(html)
    return sub_html


# 子网页处理函数：解析子网页，爬取诗句内容
def sub_page_parse(sub_html):
    poem_list = []
    for html in sub_html:
        soup = BeautifulSoup(html, 'lxml')
        poem = soup.select('div.left > div.sons > div.cont > div.contson')
        if len(poem) == 0:
            continue
        poem = poem[0].get_text()
        poem_list.append(poem.strip())
    return poem_list
```

```
# 子网页处理函数：保存诗句到 txt
def sub_page_save(poem_list):
    import json
    with open(r'poems.txt', 'a', encoding='utf-8') as txt_file:
        for element in poem_list:
            txt_file.write(json.dumps(element, ensure_ascii=False) + '\n\n')


if __name__ == '__main__':
    print("**************开始爬取古诗文网站********************")
    ua = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.86 Safari/537.36'}
    poemCount = 0
    for i in range(1, 5):
        url = 'https://so.gushiwen.cn/mingjus/default.aspx?page=%d' % i
        print(url)
        # time.sleep(1)
        html = page_request(url, ua)
        info_list = page_parse(html)
        save_txt(info_list)
        # 开始处理子网页
        print("开始解析第%d" % i + "页")
        # 开始解析名句子网页
        sub_html = sub_page_request(info_list)
        poem_list = sub_page_parse(sub_html)
        sub_page_save(poem_list)

        poemCount += len(info_list[0])

    print("****************爬取完成************************")
    print("共爬取%d" % poemCount + "个古诗词名句")
    print("共爬取%d" % poemCount + "个古诗词")
```


3.7 Scrapy 爬虫

网络爬虫框架就是一些爬虫项目的半成品，它已经将一些爬虫常用的功能写好，然后留下一些接口，在不同的爬虫项目当中调用适合自己项目的接口，再编写少量代码就可以实现自己所需要的功能。因为框架中已经实现了爬虫常用的功能，所以为开发人员节省了很多时间和精力。常用的网络爬虫开发框架包括 Scrapy、Crawley、PySpider 等，这里简要介绍 Scrapy 框架。在使用 Scrapy 框架编写网络爬虫程序时，常常会使用到 XPath 语言，因此，本节内容中也会介绍 XPath 的相关知识。

3.7.1 Scrapy 爬虫概述

Scrapy 是一套基于 Twisted 的异步处理框架，是纯 Python 实现的爬虫框架，用户只需要定制开发几个模块就可以轻松地实现一个爬虫，用来抓取网页内容或者各种图片。Scrapy 运行于 Linux/Windows/MacOS 等多种环境，具有速度快、扩展性强、使用简便等特点。即便是新手，也能迅速学会使用 Scrapy 编写所需要的爬虫程序。Scrapy 可以在本地运行，也能部署到云端实现真正的生产级数据采集系统。Scrapy 用途广泛，可以用于数据挖掘、监测和自动化测试。Scrapy 吸引人的地方在于它是一个框架，任何人都可以根据需求对它进行修改。当然，Scrapy 只是 Python 的一个主流框架，除了 Scrapy 外，还有其他基于 Python 的爬虫框架，包括 Crawley、Portia、Newspaper、Python-goose、Beautiful Soup、Mechanize、Selenium 和 Cola 等。

1．Scrapy 体系架构

如图 3-7 所示，Scrapy 体系架构包括以下组成部分：

①Scrapy 引擎（Engine）。Scrapy 引擎相当于一个中枢站，负责调度器、项目管道、下载器和爬虫四个组件之间的通信。例如，将接收到的爬虫的请求发送给调度器，将爬虫的存储请求发送给项目管道。调度器发送的请求，会被引擎提交到下载器进行处理，而下载器处理完成后会发送响应给引擎，引擎将其发送至爬虫进行处理。

②爬虫（Spiders）。相当于一个解析器，负责接收 Scrapy 引擎发送过来的响应，对其进行解析，开发人员可以在其内部编写解析规则。解析好的内容可以发送存储请求给 Scrapy 引擎。在爬虫中解析出的新的 URL，可以向 Scrapy 引擎发送请求。注意，入口 URL 也是存储在爬虫中。

③下载器（Downloader）。下载器用于下载搜索引擎发送的所有请求，并将网页内容返回给爬虫。下载器建立在 Twisted 这个高效的异步模型之上。

④调度器（Scheduler）。可以理解成一个队列，存储 Scrapy 引擎发送过来的 URL，并按顺序取出 URL 发送给 Scrapy 引擎进行请求操作。

⑤项目管道（Item Pipeline）。项目管道是保存数据用的，它负责处理爬虫中获取到的项目，并进行处理，比如去重、持久化存储（比如存数据库或写入文件）。

⑥下载器中间件（Downloader Middlewares）。下载器中间件是位于引擎和下载器之间的框架，主要用于处理引擎与下载器之间的请求及响应。类似于自定义扩展下载功能的组件。

⑦爬虫中间件（Spider Middlewares）。爬虫中间件是介于引擎和爬虫之间的框架，主要工作是处理爬虫的响应输入和请求输出。

⑧调度器中间件（Scheduler Middlewares）。调度器中间件是介于引擎和调度器之间的中间件，用于处理从引擎发送到调度器的请求和响应，可以自定义扩展和操作搜索引擎与爬虫中间“通信”的功能组件（如进入爬虫的请求和从爬虫出去的请求）。

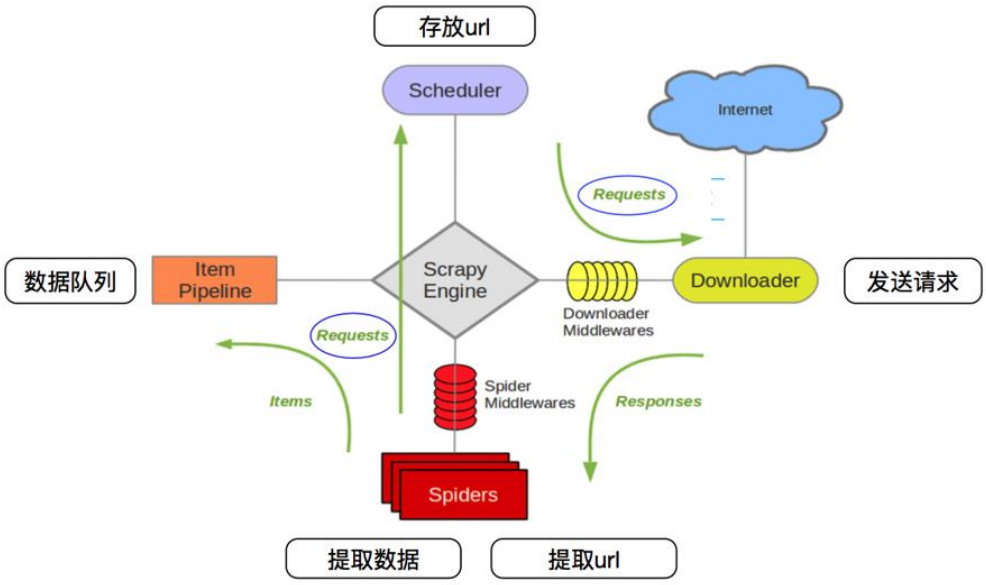

图 3-7 Scrapy 体系架构

2．Scrapy 工作流

Scrapy 工作流也叫作“运行流程”或叫作“数据处理流程”，整个数据处理流程由 Scrapy 引擎进行控制，其主要的运行步骤如下：

①Scrapy 引擎从调度器中取出一个链接（URL）用于接下来的抓取；

②Scrapy 引擎把 URL 封装成一个请求并传给下载器；

③下载器把资源下载下来，并封装成应答包；

④爬虫解析应答包；

⑤如果解析出的是项目，则交给项目管道进行进一步的处理；

⑥如果解析出的是链接（URL），则把 URL 交给调度器等待抓取。

3.7.2 XPath 语言

XPath（XML Path Language）是一门在 XML 和 HTML 文档中查找信息的语言，可用来在 XML 和 HTML 文档中对元素和属性进行遍历。简单来说，网页数据是以超文本的形式来呈现的，想要获取里面的数据，就要按照一定的规则来进行数据的处理，这种规则就叫做 XPath。XPath 提供了超过 100 个内建函数，几乎所有要定位的节点都可以用 XPath 来定位，在做网络爬虫时可以使用 XPath 提取所需的信息。

1．基本术语

XML 文档通常可以被看作一棵节点树。在 XML 中，有元素、属性、文本、命名空间、处理指令、注释以及文档节点等七种类型的节点，其中，元素节点是最常用的节点。下面是一个 HTML 文档中的代码：

```
<html>
```

```
    <head><title>BigData Software</title></head>
    <p class="title"><b>BigData Software</b></p>
    <p class="bigdata">There are three famous bigdata software;and their names are
        <a href="http://example.com/hadoop" class="hadoop" id="link1">Hadoop</a>,
        <a href="http://example.com/spark" class="spark" id="link2">Spark</a>and
        <a href="http://example.com/flink" class="flink" id="link3"><!--Flink--></a>;
        and they are widely used in real application.</p>
    <p class="bigdata">...</p>
</html>
```

上面的 HTML 文档中，<html>是文档节点，<title>BigData Software</title>是元素节点，class="title"是属性节点。节点之间存在下面几种关系：

（1）父节点：每个元素和属性都有一个父节点。例如，html 节点是 head 节点和 p 节点的父节点；head 节点是 title 节点的父节点；第二个 p 节点是中间三个 a 节点的父节点。

（2）子节点：每一个元素节点的下一个直接节点是该元素节点的子节点。每个元素节点可以有零个、一个或多个子节点。例如，title 节点是 head 节点的子节点。

（3）兄弟节点：拥有相同父节点的节点，就是兄弟节点。例如，第二个 p 节点中的三个 a 节点就是兄弟节点；head 节点和中间三个 p 节点就是兄弟节点；title 节点和 a 节点就不是兄弟节点，因为不是同一个父节点。

（4）祖先节点：节点的父节点以及父节点的父节点等，称作“祖先节点”。例如，html 节点和 head 节点是 title 节点的祖先节点。

（5）后代节点：节点的子节点以及子节点的子节点等，称作“后代节点”。例如，html 节点的后代节点有 head、title、b、p 以及 a 节点。

2．基本语法

XML/HTML 文档是由标签构成的，所有的标签都有很强的层级关系。基于这种层级关系，XPath 语法能够准确定位我们所需要的信息。XPath 使用路径表达式来选取 XML/HTML 文档中的节点，这个路径表达式和普通计算机文件系统中见到的路径表达式非常相似。在 XPath 语法中，我们直接使用路径来选取，再加上适当的谓语或函数进行指定，就可以准确定位到指定的节点。

（1）节点选取

XPath 选取节点时，是沿着路径到达目标，表 3-3 列出了常用的表达式。

表 3-3 常用表达式

表达式	描述
nodename	选取 nodename 节点的所有子节点
/	从根节点开始选取
//	从当前文档选取所有匹配的节点，而不考虑它们的位置
@	选取属性
.	选取当前节点
..	选取当前节点的父节点

“/”可以理解为绝对路径，需要从根节点开始；“./”则是相对路径，可以从当前节点开始；“../”则是先返回上一节点，从上一节点开始。这与普通计算机的文件系统类似。下

面给出测试这些表达式的简单实例，这里需要用到 lxml 中的 etree 模块，在使用之前需要执行如下命令安装 lxml 库：

```
> pip install lxml
```

下面是实例代码：

```
>>> html_text = """
<html>
  <body>
    <head><title>BigData Software</title></head>
    <p class="title"><b>BigData Software</b></p>
    <p class="bigdata">There are three famous bigdata software;and their names are
      <a href="http://example.com/hadoop" class="bigdata Hadoop" id="link1">Hadoop</a>,
      <a href="http://example.com/spark" class="bigdata Spark" id="link2">Spark</a>and
      <a href="http://example.com/flink" class="bigdata Flink" id="link3"><!--Flink--></a>;
        and they are widely used in real application.</p>
    <p class="bigdata">others</p>
    <p>……</p>
  </body>
</html>
"""
>>> from lxml import etree
>>> html = etree.HTML(html_text)
>>> html_data = html.xpath('body')
>>> print(html_data)
[<Element body at 0x1608dda2d80>]
```

可以看出，html.xpath('body')的输出结果不是像 HTML 里面那样显示的标签，其实这就是我们所要的元素，只不过我们还需要再进行一步操作，也就是使用 etree 中的.tostring()方法将其进行转换。此外，html.xpath('body')的输出结果是一个列表，因此，我们可以使用 for 循环来遍历列表，具体代码如下：

```
>>> for element in html_data:
        print(etree.tostring(element))
```

由于输出结果比较繁杂，这里没有给出，但是观察结果可以发现，它是标签<body>中的子节点。

“//”表示全局搜索，比如，“//p”可以将所有的<p>标签搜索出来。“/”表示在某标签下进行搜索，只能搜索子节点，不能搜索子节点的子节点。简单来说，“//”可以进行跳级搜索，“/”只能在本级上进行搜索，不能跳跃。下面是具体实例：

（1）逐级搜索

```
>>> html_data = html.xpath('/html/body/p/a')
>>> for element in html_data:
        print(etree.tostring(element))
```

（2）跳级搜索

```
>>> html_data = html.xpath('//a')
>>> for element in html_data:
        print(etree.tostring(element))
```

上面两段代码的执行结果相同，具体如下：

```
b'<a href="http://example.com/hadoop" class="bigdata Hadoop" id="link1">Hadoop</a>,\n
'
b'<a href="http://example.com/spark" class="bigdata Spark" id="link2">Spark</a>and\n
'
b'<a href="http://example.com/flink" class="bigdata Flink" id="link3"><!--Flink--></a>;\n
and they are widely used in real application.'
```

可以在方括号内添加“@”，将标签属性填进去，这样就可以准确地将含有该标签属性的部分提取出来，示例代码如下：

```
>>> html_data = html.xpath('//p/a[@class="bigdata Spark"]')
>>> for element in html_data:
        print(etree.tostring(element))
```

上面代码的执行结果如下：

```
b'<a href="http://example.com/spark" class="bigdata Spark" id="link2">Spark</a>and\n '
```

（2）谓语

直接使用前面介绍的方法可以定位到多数我们需要的节点，但是有时候我们需要查找某个特定的节点或者包含某个指定值的节点，就要用到谓语。谓语是被嵌在方括号中的。表 3-4 列出了一些带有谓语的路径表达式及其描述的内容。

表 3-4 带有谓语的路径表达式实例

路径表达式	描述
//body/p[k]	选取所有 body 下第 k 个 p 标签（k 取值从 1 开始）
//body/p[last()]	选取所有 body 下最后一个 p 标签
//body/p[last() - 1]	选取所有 body 下倒数第二个 p 标签
//body/p[position()<3]	选取所有 body 下的前两个 p 标签
//body/p[@class]	选取所有 body 下带有 class 属性的 p 标签
//body/p[@class="bigdata"]	选取所有 body 下，class 为 bigdata 的 p 标签

下面演示表 3-4 中的最后一个例子，选取所有 body 下，class 为 bigdata 的 p 标签，代码如下：

```
>>> html_data = html.xpath('//body/p[@class="bigdata"]')
>>> for element in html_data:
        print(etree.tostring(element))
```

上面代码执行结果如下：

```
b'<p class="bigdata">There are three famous bigdata software;and their names are\n
      <a href="http://example.com/hadoop" class="bigdata Hadoop"
id="link1">Hadoop</a>,\n
      <a href="http://example.com/spark" class="bigdata Spark" id="link2">Spark</a>and\n
      <a href="http://example.com/flink" class="bigdata Flink" id="link3"><!--Flink--></a>;\n
      and they are widely used in real application.</p>\n    '
b'<p class="bigdata">...</p>\n    '
```

（3）函数

XPath 中提供超过 100 个内建函数用于字符串值、数值、日期和时间比较序列处理等操作，极大地方便了我们定位获取所需要的信息。表 3-5 列出了几个常用的函数。

表 3-5 常用函数

函数名	描述	示例	说明

contains()	选取属性或文本包含某些字符	//p[contains(@class, "bigdata")]	选取所有 class 属性包含 bigdata 的 p 标签
starts-with()	选择属性或文本以某些字符开头	//a[starts-with (@class, "bigdata")]	选取所有 class 属性以 bigdata 开头的 a 标签
ends-with()	选取属性或文本以某些字符结尾	//a[ends-with (@class, "Flink")]	选取所有 class 属性以 Flink 结尾的 a 标签
text()	获取元素节点包含的文本内容	//a[contains(@class, "Hadoop")]/text()	获取所有 class 属性包含 Hadoop 的 a 标签中的文本内容

下面是示例代码，获取所有 class 属性包含 bigdata 的 a 标签中的文本内容，代码如下：

```
>>> html = etree.HTML(html_text)
>>> html_data = html.xpath('//a[contains(@class, "bigdata")]/text()')
>>> print(html_data)
['Hadoop', 'Spark']
```

在演示的 HTML 代码中，还有一个 a 标签也符合代码的要求，但是因为其文本内容是注释，所以不会被抽取出来显示。

3.7.3 Scrapy 爬虫实例

访问古诗文网站（https://so.gushiwen.cn/mingjus/），使用 Scrapy 框架编写爬虫程序，爬取每个名句及其完整古诗内容，并把爬取到的数据分别保存到文本文件和 MySQL 数据库中。本实例需要使用开发工具 PyCharm (Community Edition)，请到 PyCharm 官网（https://www.jetbrains.com/pycharm/download/）或本书官网的“下载专区”中下载 PyCharm 安装文件并安装。

本实例包括以下几个步骤：

- 新建工程；
- 编写代码文件 items.py；
- 编写爬虫文件；
- 编写代码文件 pipelines.py；
- 编写代码文件 settings.py；
- 运行程序；
- 把数据保存到数据库中。

1.新建工程

在 PyCharm 中新建一个名称为“scrapyProject”的工程。在“scrapyProject”工程底部打开 Terminal 窗口（如图 3-8 所示），在命令提示符后面输入命令“pip install scrapy”，下载 Scrapy 框架所需文件。

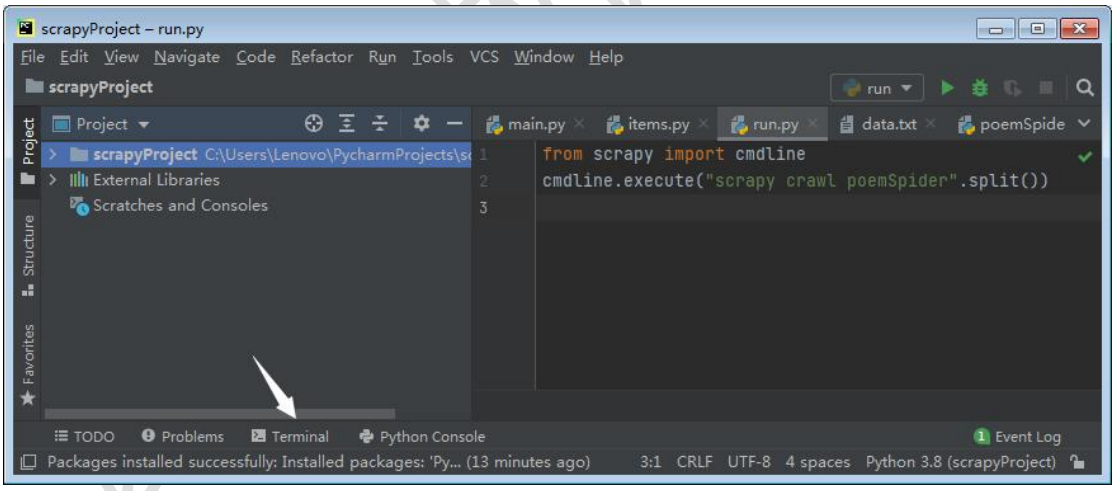

图 3-8 打开 Terminal 窗口

下载完成后，继续输入命令“scrapy startproject poemScrapy”，创建 Scrapy 爬虫框架相关目录和文件。创建完成以后的具体目录结构如图 3-9 所示，这些目录和文件都是由 Scrapy 框架自动创建的，不需要手动创建。

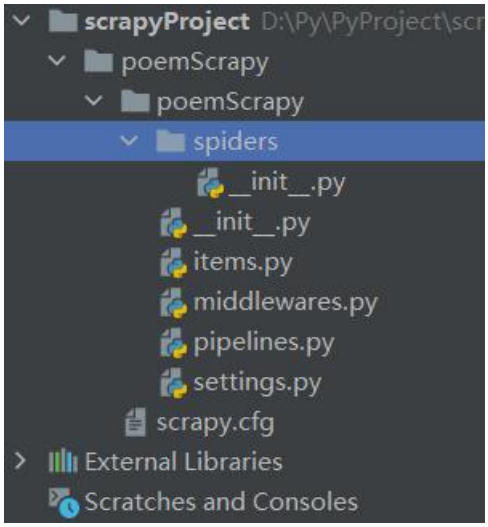

图 3-9 Scrapy 爬虫程序目录结构

在 Scrapy 爬虫程序目录结构中，各个目录和文件的作用如下：

- Spiders 目录：该目录下包含爬虫文件，需编码实现爬虫过程；
- __init__.py：为 Python 模块初始化目录，可以什么都不写，但是必须要有；
- items.py：模型文件，存放了需要爬取的字段；
- middlewares.py：中间件（爬虫中间件、下载中间件），本案例中不用此文件；
- pipelines.py：管道文件，用于配置数据持久化，例如写入数据库；
- settings.py：爬虫配置文件；
- scrapy.cfg：项目基础设置文件，设置爬虫启用功能等。在本案例中不用此文件。

2.编写代码文件 items.py

在 items.py 中定义字段用于保存数据，items.py 的具体代码内容如下：

```
import scrapy

class PoemscrapyItem(scrapy.Item):
    # 名句
    sentence = scrapy.Field()
    # 出处
    source = scrapy.Field()
    # 全文链接
    url = scrapy.Field()
    # 名句详细信息
    content = scrapy.Field()
```

3.编写爬虫文件

在 Terminal 窗口输入命令“cd poemScrapy”，进入对应的爬虫工程中，再输入命令“scrapy genspider poemSpider gushiwen.cn”，这时，在 spiders 目录下会出现一个新的 Python 文件

poemSpider.py，该文件就是我们要编写爬虫程序的位置。下面是 poemSpider.py 的具体代码：

```
import scrapy
from scrapy import Request
from ..items import PoemscrapyItem

class PoemspiderSpider(scrapy.Spider):
    name = 'poemSpider'    # 用于区别不同的爬虫
    allowed_domains = ['gushiwen.cn']   # 允许访问的域
    start_urls = ['http://so.gushiwen.cn/mingjus/']   # 爬取的地址

    def parse(self, response):
        # 先获每句名句的 div
        for box in response.xpath('//*[@id="html"]/body/div[2]/div[1]/div[2]/div'):
            # 获取每句名句的链接
            url = 'https://so.gushiwen.cn' + box.xpath('.//@href').get()
            # 获取每句名句内容
            sentence = box.xpath('.//a[1]/text()') .get()
            # 获取每句名句出处
            source = box.xpath('.//a[2]/text()') .get()
            # 实例化容器
            item = PoemscrapyItem()
            # 将收集到的信息封装起来
            item['url'] = url
            item['sentence'] = sentence
            item['source'] = source
            # 处理子页
            yield scrapy.Request(url=url, meta={'item': item}, callback=self.parse_detail)
        # 翻页
        next = response.xpath('//a[@class="amore"]/@href'). get()
        if next is not None:
            next_url = 'https://so.gushiwen.cn' + next
            # 处理下一页内容
            yield Request(next_url)

    def parse_detail(self, response):
        # 获取名句的详细信息
        item = response.meta['item']
        content_list = response.xpath('//div[@class="contson"]//text()').getall()
        content = "".join(content_list).strip().replace('\n', '').replace('\u3000', '')
        item['content'] = content
        yield item
```

在上面的代码中，response.xpath()返回的是 scrapy.selector.unified.SelectorList 对象，比如 response.xpath('//div[@class="contson"]//text()')返回的部分结果如下：

```
[<Selector xpath='//div[@class="contson"]//text()' data='\n 日日望乡国，空歌白苎词。'>, <Selector xpath='//div[@class="contson"]//text()' data='长因送人处，忆得别家时。'>, <Selector xpath='//div[@class="contson"]//text()' data=' 失意还独语， 多愁只自知。 '>, <Selector xpath='//div[@class="contson"]//text()' data='客亭门外柳，折尽向南枝。\n'>]
```

这时，response.xpath('//div[@class="contson"]//text()').get()返回的结果如下：

#注意，这里会输出一个空行

'日日望乡国，空歌白苎词。'

response.xpath('//div[@class="contson"]//text()').getall()返回的结果如下：

['\n 日日望乡国，空歌白苎词。', '长因送人处，忆得别家时。', '失意还独语，多愁只自知。', '客亭门外柳，折尽向南枝。\n']

4.编写代码文件 **pipelines.py**

当我们成功获取需要的信息后，要对信息进行存储。在 Scrapy 爬虫框架中，当 item 被爬虫收集完后，将会被传递到 pipelines。现在要将爬取到的数据保存到文本文件中，可以使用的 pipelines.py 代码：

```
import json

class PoemscrapyPipeline:
    def __init__(self):
        # 打开文件
        self.file = open('data.txt', 'w', encoding='utf-8')

    def process_item(self, item, spider):
        # 读取 item 中的数据
        line = json.dumps(dict(item), ensure_ascii=False) + '\n'
        # 写入文件
        self.file.write(line)
        return item
```

5.编写代码文件 **settings.py**

settings.py 的具体代码如下：

```
BOT_NAME = 'poemScrapy'

SPIDER_MODULES = ['poemScrapy.spiders']
NEWSPIDER_MODULE = 'poemScrapy.spiders'

USER_AGENT = 'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4421.5 Safari/537.36'
```

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = False

# 设置日志打印的等级
LOG_LEVEL = 'WARNING'

ITEM_PIPELINES = {
    'poemScrapy.pipelines.PoemscrapyPipeline': 1,
}
```

其中，更改 USER-AGENT 和 ROBOTSTXT_OBEY 是为了避免访问被拦截或出错；设置 LOG_LEVEL 是为了避免在爬取过程中显示过多的日志信息；设置 ITEM_PIPELINES 是因为本案例使用到 pipeline，需要先注册 pipeline，右侧的数字‘1’为该 pipeline 的优先级，范围 1-1000，数值越小越优先执行。读者也可以根据实际需求，适当更改 settings.py 中的内容。

6.运行程序

有两种执行 Scrapy 爬虫的方法，第一种是在 Terminal 窗口中输入命令“scrapy crawl poemSpider”，然后回车运行，等待几秒钟后即可完成数据的爬取。第二种是在 poemScrapy 目录下新建 Python 文件 run.py（run.py 应与 scrapy.cfg 文件在同一层目录下），并输入下面代码：

```
from scrapy import cmdline
cmdline.execute("scrapy crawl poemSpider".split())
```

在 run.py 代码区域点击鼠标右键，在弹出的菜单里选择“Run”运行代码，就可以执行 Scrapy 爬虫程序。执行成功以后，就可以看到生成的数据文件 data.txt，其内容类似如下：

```
{"url": "https://so.gushiwen.cn/mingju/juv_2f9cf2c444f2.aspx",
"sentence": "人道恶盈而好谦。", "source": "《易传·彖传上·谦》", "content": "
解释：做人之道，最怕自满而最喜谦虚。"}
```

3.8 本章小结

网络爬虫系统的功能是下载网页数据，为搜索引擎系统或需要网络数据的企业提供数据来源。本章内容介绍了网络爬虫程序的编写方法，主要包括如何请求网页以及如何解析网页。在网页请求环节，需要注意的是，一些网站设置了反爬机制，会导致我们爬取网页失败。在网页解析环节，我们可以灵活运用 BeautifulSoup 提供的各种方法获取我们需要的数据。同时，为了减少程序开发工作量，可以选择包括 Scrapy 在内的一些网络爬虫开发框架编写网络网络爬虫程序。