基础网络数据采集

学习目标：

- 了解基础网络数据采集的架构和工作流程；
- 掌握 Python 环境下基础网络数据采集各个模块的工作流程；
- 熟悉 Python 环境下实现网络数据采集的关键代码；
- 理解基础网络数据采集的工作原理。

基础网络数据采集实现的功能比较简单，它仅仅考虑功能实现，没有涉及优化和稳健性的考虑。相比大型分布式的网络数据采集，基础网络数据采集虽然项目小，但需要的模块都应具备，只不过在实现方式和优化方式上不及大型分布式的数据采集全面、多样。本章将对基础网络数据采集进行详细讲解。

5.1 基础网络数据采集的架构及运行流程

网络数据采集是一个自动提取网页的程序，它可以从互联网上下载网页，是搜索引擎的重要组成部分。传统网络数据采集从一个或若干初始网页的 URL 开始获得初始网页上的 URL，在爬取网页的过程中不断从当前页面上抽取新的 URL 放入队列，直到满足系统的停止条件。从功能上来讲，网络数据采集一般分为数据爬取、处理、存储三个部分。聚焦网络爬虫（又称网页蜘蛛，网络机器人，在 FQAF 社区中经常被称为网追逐者）是一种按照一定规则自动爬取互联网信息的程序或脚本。另外一些不常使用的名字还有蚂蚁、自动索引、模拟程序、蠕虫。基础网络爬虫的流程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，并将其放入等待爬取的 URL 队列中，然后根据一定的搜索策略从队列中选择下一步要爬取的网页 URL，并重复上述过程，直到满足系统的某一条件时停止。另外，所有被爬取的网页都将会被系统存储，进行一定的分析、过滤，并建立索引，以便之后的查询和检索。这一过程得到的分析结果还可能为以后的爬取过程提供反馈和指导。基础网络数据采集的架构如图 5-1 所示。

相对于通用网络数据采集，基础网络数据采集还需要解决以下三个主要问题：

- 对爬取目标的描述或定义；
- 对网页或数据的分析与过滤；
- 对 URL 的搜索策略。

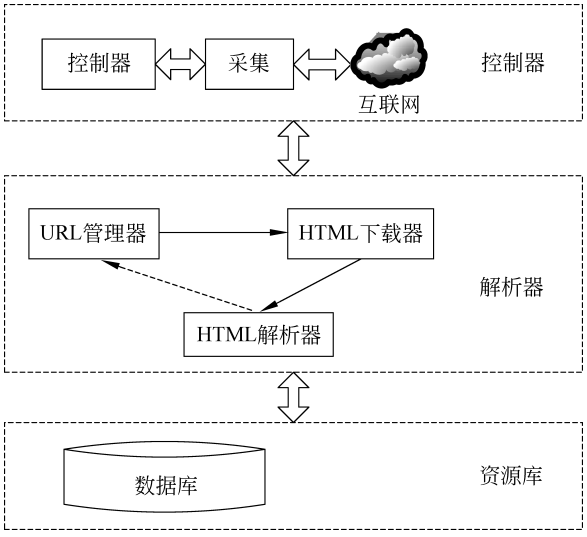

图 5-1　基础网络数据采集的架构

在网络数据采集的系统框架中，主过程由控制器、解析器、资源库三部分组成。控制器的主要工作是负责给多线程中的各个网络数据采集线程分配工作任务。解析器的主要工作是下载网页并进行页面的处理，主要是将一些 JS 脚本标签、CSS 代码、空格字符、HTML 标签等内容处理掉。网络数据采集的基本工作由解析器完成。资源库用来存放下载的网页资源，一般采用大型数据库存储，如 Oracle 数据库，并对其建立索引。

(1) 控制器

控制器是网络数据采集的中央控制器，其根据系统传来的 URL 链接分配线程，然后启动线程以调用网络数据采集网页。

(2) 解析器

解析器是网络数据采集的主要部分，其负责的工作主要包括：下载网页、对网页的文本进行处理(如过滤、抽取特殊 HTML 标签、分析数据)。

(3) 资源库

资源库主要用来存储从网页中下载的数据，并提供生成索引的目标源，大中型数据库产品有 Oracle、SQL Server 等。

网络数据采集系统一般会选择一些比较重要的、出度(网页中指向其他网页的链接数目)较大的网站的 URL 作为种子 URL 集合。网络数据采集系统以这些种子集合作为初始 URL 开始数据的爬取。因为网页中含有链接信息，所以通过已有网页的 URL 会得到一些新的 URL，可以把网页之间的指向结构视为一个森林，每个种子 URL 对应的网页是森林中的一棵树的根节点。这样，网络数据采集系统就可以根据广度优先搜索算法或者深度优先搜索算法遍历所有网页。由于深度优先搜索算法可能会使网络数据采集系统陷入某一个网站内部，不利于搜索比较靠近网站首页的网页信息，因此一般采用广度优先搜索算法采集网页。网络数据采集系统首先将种子 URL 加入下载队列，然后简单地从队首取出一个 URL 以下载其对应的网页。在将得到的网页内容存储后，解析网页中的链

接信息，就可以得到一些新的 URL，并将这些 URL 加入下载队列；然后再取出一个 URL，下载对应的网页，接着再解析；如此反复进行，直到遍历整个网络或者满足某种条件后才会停止。

5.2 URL 管理器

下面介绍 URL 管理器的主要功能及其在 Python 数据采集环境下的实现方式。

5.2.1 URL 管理器的主要功能

URL 即统一资源定位符，也就是通常说的网址。URL 是对可以从互联网上得到的资源的位置和访问方法的一种简洁表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的 URL，包含文件的位置以及浏览器应该怎样处理它等方面的信息。

URL 的格式由以下三部分组成。

- 第一部分是协议（或称服务方式）；
- 第二部分是存有该资源的主机 IP 地址（有时也包括端口号）；
- 第三部分是主机资源的具体地址，如目录和文件名等。

网络数据采集在采集数据时必须有一个目标 URL 才可以获取数据，该 URL 是网络数据采集获取数据的基本依据，准确理解它的含义对网络数据采集的学习有很大帮助。

管理待爬取 URL 集合和已爬取 URL 集合的意义是：防止重复爬取、防止循环爬取（如两个网页因相互引用而形成“死循环”）。

一个 URL 管理器应该具有以下几个功能：

- 添加新的 URL 到待采集集合中，判断待添加的 URL 是否在容器中；
- 获取待采集的 URL；
- 判断是否还有待采集的 URL；
- 将采集后的 URL 从待采集集合移动到已采集集合。

URL 管理器的主要功能如图 5-2 所示。

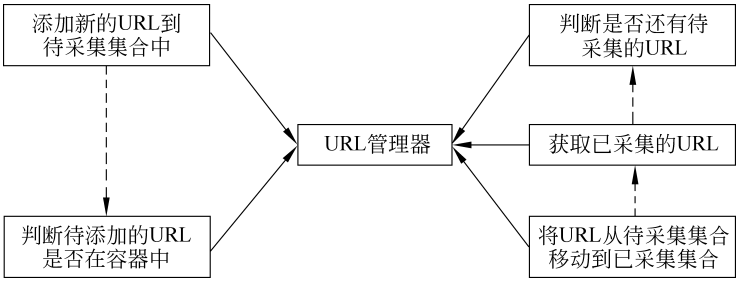

图 5-2 URL 管理器的主要功能

5.2.2 URL 管理器的实现方式

在 URL 管理器的实现方式上，Python 主要采用内存（set）和关系数据库（MySQL）。

对于小型程序，一般在内存中实现。Python 内置的 set()类型能够自动判断元素是否重复。对于大一些的程序，一般使用数据库实现，如图 5-3 所示。

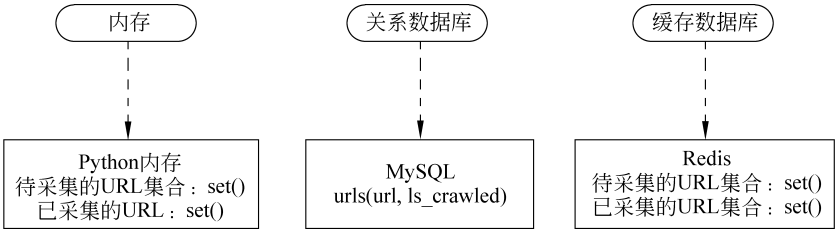

图 5-3　URL 管理器的实现方式

URL 管理器的实现方式有以下三种。

- 适合个人的：内存。
- 适合小型企业或个人的：关系数据库(永久存储或内存不够用)用一个字段表示 URL，用另一个字段表示是否被采集过。
- 适合大型互联网公司的：缓存数据库(高性能)。

URL 管理器的实现程序 HtmlManager.py 的完整代码如下。

```
#-*-coding:utf-8-*-
class UrlManager(object):
def_init_(self):
self.new_urls=set()                          #待采集的 URL
self.old_urls=set()                          #已采集的 URL
def add_new_url(self,url):                   #向管理器中添加一个新的 URL
if url is None:
return
if url not in self.new_urls and url not in self.old_urls:
self.new_urls.add(url)
def add_new_urls(self,urls):                 #向管理器中添加更多新的 URL
if urls is None or len(urls)==0:
return
for url in urls:
self.add_new_url(url)
def has_new_url(self):                       #判断管理器是否有新的待采集的 URL
return len(self.new_urls)!=0
def get_new_url(self):                       #从管理器中获取一个新的待采集的 URL
new_url=self.new_urls.pop()
self.old_urls.add(new_url)
return new_url
```

从一个循环中获取一个存在的 URL，进入一个页面，保存此页面的相关信息，保存相关电影的 URL，直至没有新的 URL 或者电影的数量达到规定数量。因此，需要一个用来保存新出现的 URL 和标记出现过的 URL 的程序，以保证不重复。程序在最开始时将入

口页的 URL 保存在 URL 管理器中,然后进入循环。

5.3 HTML 下载器

HTML 下载器是将互联网上 URL 对应的网页下载到本地的工具。Python 中的 HTML 下载器主要使用 urllib 库创建,这是 Python 自带的模块。Python 3.x 将 2.x 版本中的 urllib2 库集成到了 urllib 中,在 Requests 等子模块中。urllib 中的 urlopen 函数用于打开 URL 并获取 URL 数据。urlopen 函数的参数可以是 URL 链接,也可以是 Request 对象。对于简单的网页,直接使用 URL 字符串作为参数就已足够;但对于复杂的网页,如设有防网络数据采集机制的网页,在使用 urlopen 函数时,就需要添加 http header。对于带有登录机制的网页,需要设置 Cookie。HTML 下载器的主要作用是从 URL 管理器中获取新的 URL 并将其从对应的服务器中下载下来。

5.3.1 下载方法

方法 1:直接请求,示例如下。

```
#coding:utf-8
import urllib3
#直接请求
response=urllib3.urlopen('http://www.ryjiaoyu.com')
#获取状态码,如果是 200,则表示获取成功
print(response.getcode())
#读取内容
cont=response.read()
print(len(cont))
```

方法 2:添加 http header,示例如下。

```
#coding:utf-8
import urllib3
#创建 Request 对象
request=urllib3.Request('http://www.ryjiaoyu.com')
#添加 http header,伪装成浏览器访问
request.add_header('User-Agent','Mozilla/5.0')
#发送请求,获取结果
response=urllib3.urlopen(request)#读取内容
cont=response.read()
print(len(cont))
```

方法 3:添加特殊情景的处理器,示例如下。

```
#coding:utf-8
import urllib3
```

```
import cookielib
#可以进行 Cookie 处理
cj=cookielib.CookieJar()
opener=urllib3.build_opener(urllib3.HTTPCookieProcessor(cj))
urllib3.install_opener(opener)
#直接请求
response=urllib3.urlopen('http://www.ryjiaoyu.com')
#读取内容
cont=response.read()
print(len(cont))
#打印 cookieprint cj
```

5.3.2 注意事项

使用 HTML 下载器下载网页时,需要注意网页的编码,以保证下载的网页没有乱码。HTML 下载器需要用到 Requests 模块,其中只需要实现 download(url)接口即可。HTML 下载器 HtmlDownloader.py 程序的示例如下。

```
import urllib2
class HtmlDownloader(object):
def download(self,url):
if url is None:
return None
response=urllib2.urlopen(url)
if response.getcode()!=200:
return None
return response.read()
```

5.4 HTML 解析器

HTML 解析器使用 bs4 解析 HTML,需要解析的部分主要分为提取相关词条页面的 URL、当前词条的标题和摘要信息。

首先使用 firebug 查看标题和摘要所在的结构位置,如图 5-4 所示。

从图 5-4 可以看到,标题位于 HTML 标记的<dd class="lemmaWgt-lemmaTitle"><h1></h1>中,摘要位于 中;最后分析需要提取的 URL 格式。相关词条的 URL 格式类似于HTML的形式,提取 a 标记中的 href 属性即可。从格式中可以看到,href 属性值是一个相对网址,可以使用 urlparse.urljoin 函数将当前网址和相对网址拼接成完整的 URL 路径。

HTML 解析器主要提供一个 parser 对外接口,输入参数为当前页面的 URL 和

图 5-4 HTML 结构位置

HTML 下载器返回的网页内容。HTML 解析器 HtmlParser.py 程序的代码如下。

```
import reimport urlparsefrom bs4
import Beautiful Soup
class HtmlParser(object):
def parse(self,page_url,html_cont):
if page_url is None or html_cont is None:
return
soup=Beautiful Soup(html_cont,'html.parser',from_encoding='utf-8')
new_urls=self._get_new_urls(page_url,soup)
new_data=self._get_new_data(page_url,soup)
return new_urls,new_data
def_get_new_urls(self,page_url,soup):
new_urls=set()
#/view/123.htm
links=soup.find_all('a',href=re.compile(r"/view/\d+\.htm"))
for link in links:
new_url=link['href']
new_full_url=urlparse.urljoin(page_url,new_url)
new_urls.add(new_full_url)
return new_urls
def_get_new_data(self,page_url,soup):
res_data={}
```

```
#url
res_data['url']=page_url
#<dd class="lemmaWgt-lemmaTitle-title"><h1>Python</h1>
title_node=soup.find('dd',class_="lemmaWgt-lemmaTitle-title").find("h1")
res_data['title']=title_node.get_text()
#<div class="lemma-summary"label-module="lemmaSummary">
summary_node=soup.find('div',class_="lemma-summary")
res_data['summary']=summary_node.get_text()
return res_data
```

5.5 数据存储器

在数据采集中,数据存储器主要有两个作用:一个是将解析出来的数据存储到内存中的 store_data(data);另一个是将存储的数据输出为指定文件格式的 output_html(),其输出的数据为 HTML 格式。而 5.4 节中的 HTML 解析器输出的数据为字典类型。由于数据量较小,因此可以把数据存储为文件。当数据量较大时,将其存储为单个文件就不合适了,可以考虑将其存储为多个文件或存入数据库。为了避免频繁地读写文件,可以先把每个循环得到的数据以列表的形式暂存在内存中,等到全部页面采集结束后再存至文件。

输出文件 Html_Outputer.py 的代码如下。

```
#-*-coding:utf-8-*-
class HtmlOutputer(object):
#初始化
def_init_(self):
self.datas=[]
def collect_data(self,data):                          #收集数据
if data is None:
return
self.datas.append(data)
def output_html(self):                                #输出数据
fout=open('output.html','w')
fout.write("<html>")
fout.write("<head>")
fout.write("<meta charset='UTF-8'>")
fout.write("</head>")
fout.write("<body>")
fout.write("<table>")
#ASCII
for data in self.datas:
fout.write("<tr>")
fout.write("<td>%s</td>"%data['url'])
```

```
fout.write("<td>%s</td>"%data['title'].encode('utf-8'))
fout.write("<td>%s</td>"%data['summary'].encode('utf-8'))
fout.write("</tr>")
fout.write("</html>")
fout.write("</body>")
fout.write("</table>")
fout.close()
```

本例将数据写入了 HTML 文件，在浏览器中便能打开，方便阅读。

5.6 数据调度器

以上几节讲解了 URL 管理器、HTML 下载器、HTML 解析器和数据存储器等模块，接下来编写数据调度器以协调管理这些模块。数据调度器首先要做的是初始化各个模块，然后通过 crawl(root_url)方法传入入口 URL，方法内部按照运行流程控制各个模块的工作。数据调度器的代码文件是 spider_main.py，代码如下。

```
#-*-coding:utf-8-*-
from baike_spider import
url_manager,html_downloader,html_parser,html_outputer
class SpiderMain(object):
def_init_(self):
self.urls=url_manager.UrlManager()                    #URL管理器
self.downloader=html_downloader.HtmlDownloader()      #HTML下载器
self.parser=html_parser.HtmlParser()                  #HTML解析器
self.outputer=html_outputer.HtmlOutputer()            #数据存储器
def craw(self,root_url):
count=1                                          #判断当前采集的是第几个URL
self.urls.add_new_url(root_url)
while self.urls.has_new_url():                   #循环,采集所有相关页面,判断异常情况
try:
new_url=self.urls.get_new_url()                  #取得URL
print'craw%d:%s'%(count,new_url)                #打印当前是第几个URL
html_cont=self.downloader.download(new_url)      #下载页面数据
#进行页面解析,得到新的URL及数据
new_urls,new_data=self.parser.parse(new_url,html_cont)
self.urls.add_new_urls(new_urls)                 #添加新的URL
self.outputer.collect_data(new_data)             #收集数据
if count==10:                          #此处的10可以改为100甚至更多,代表循环次数
break
count=count+1
except:
```

```
print'craw failed'
self.outputer.output_html()                    #利用 outputer 输出收集好的数据
if_name_=="_main_":
root_url="http://baike.baidu.com/view/21087.htm"
obj_spider=SpiderMain()                        #创建采集数据程序
obj_spider.craw(root_url)                      #使用 craw 方法启动程序
```

到这里，基础网络数据采集的架构所需的模块都已经完成。启动程序，执行大约 1 分钟后，数据都会被存储为 baike.html。使用浏览器打开，运行程序 Spider_Main.py 即可采集页面，最终文件输出为 output.html，其中包含词条和词条解释，采集完毕。运行 output.html 后得到的结果如图 5-5 所示。

图 5-5　运行结果

5.7　实验 6: Scrapy 基础网络数据采集

5.7.1　创建采集模块

（1）创建项目

可以通过运行以下命令创建项目，具体示例如下。

```
#scrapy startproject p1
```

（2）创建结果

运行上述命令，得到的结果如图 5-6 所示。

图 5-6 所示文件的相关说明如下。