# 基础爬虫。

为什么叫基础爬虫呢？首先这个爬虫项目功能简单，仅仅考虑功能实现，未涉及优化和稳健性的考虑。再者爬虫虽小，五脏俱全，大型爬虫有的基础模块，这个爬虫都有，只不过实现方式、优化方式，大型爬虫做得更加全面、多样。本次实战项目的需求是爬取100个百度百科网络爬虫词条以及相关词条的标题、摘要和链接等信息，如图6-1所示。

[](http://popimage/?src=)

图6-1　网络爬虫词条

# 6.1　基础爬虫架构及运行流程

首先讲解一下基础爬虫的架构，如图6-2所示。介绍基础爬虫包含哪些模块，各个模块之间的关系是什么。

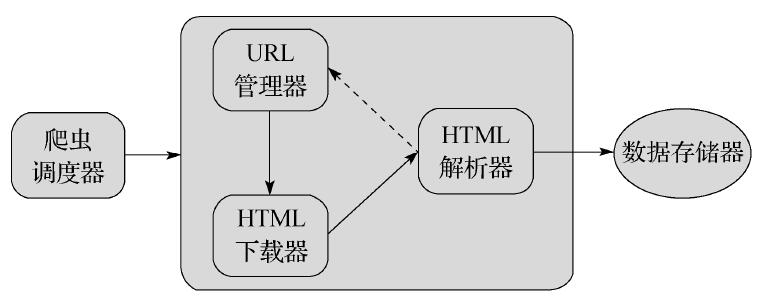
[](http://popimage/?src=)

图6-2　基础爬虫框架

基础爬虫框架主要包括五大模块，分别为爬虫调度器、URL 管理器、HTML 下载器、HTML 解析器、数据存储器。功能分析如下：

* 爬虫调度器主要负责统筹其他四个模块的协调工作。
* URL 管理器负责管理 URL 链接，维护已经爬取的 URL 集合和未爬取的 URL 集合，提供获取新 URL 链接的接口。
* HTML 下载器用于从 URL 管理器中获取未爬取的 URL 链接并下载 HTML 网页。
* HTML 解析器用于从 HTML 下载器中获取已经下载的 HTML 网页，并从中解析出新的 URL 链接交给 URL 管理器，解析出有效数据交给数据存储器。
* 数据存储器用于将 HTML 解析器解析出来的数据通过文件或者数据库的形式存储起来。

下面通过图6-3展示一下爬虫框架的动态运行流程，方便大家理解。

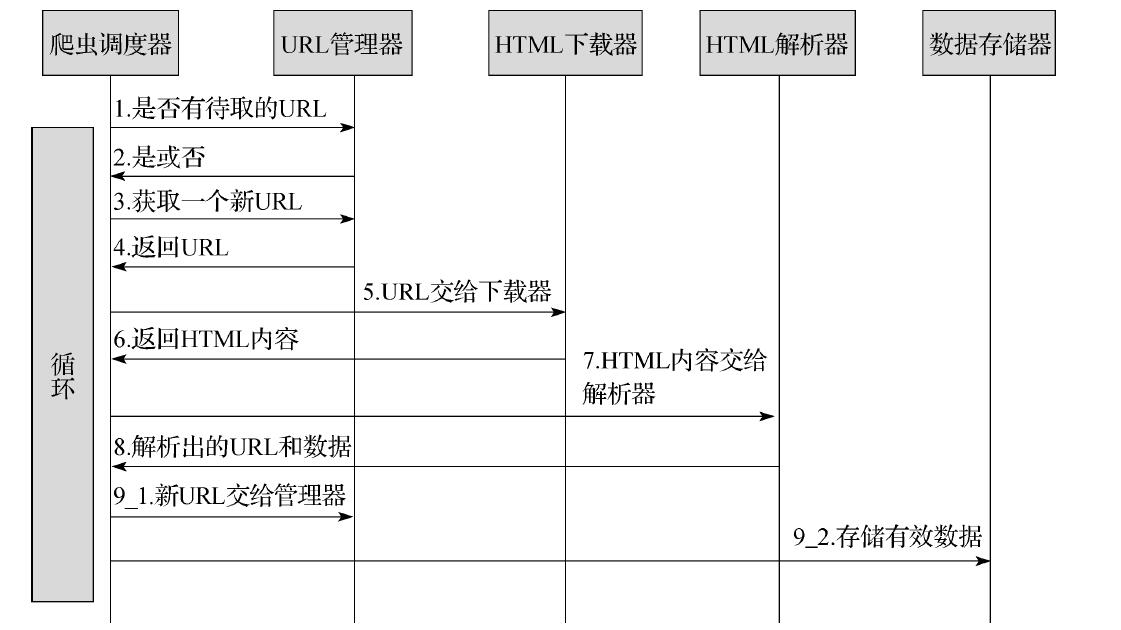
[](http://popimage/?src=)

图6-3　运行流程

# 6.2　URL 管理器

URL 管理器主要包括两个变量，一个是已爬取 URL 的集合，另一个是未爬取 URL 的集合。采用 Python 中的 set 类型，主要是使用 set 的去重复功能，防止链接重复爬取，因为爬取链接重复时容易造成死循环。链接去重复在 Python 爬虫开发中是必备的功能，解决方案主要有三种：1）内存去重2）关系数据库去重3）缓存数据库去重。大型成熟的爬虫基本上采用缓存数据库的去重方案，尽可能避免内存大小的限制，又比关系型数据库去重性能高很多。由于基础爬虫的爬取数量较小，因此我们使用 Python 中 set 这个内存去重方式。

URL 管理器除了具有两个 URL 集合，还需要提供以下接口，用于配合其他模块使用，接口如下：

* 判断是否有待取的 URL，方法定义为 has\_new\_url（）。
* 添加新的 URL 到未爬取集合中，方法定义为 add\_new\_url（url），add\_new\_urls（urls）。
* 获取一个未爬取的 URL，方法定义为 get\_new\_url（）。
* 获取未爬取 URL 集合的大小，方法定义为 new\_url\_size（）。
* 获取已经爬取的 URL 集合的大小，方法定义为 old\_url\_size（）。

程序 URLManager.py 的完整代码如下：

*# coding:utf-8*

**class** **UrlManager**(object):

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

self.new\_urls = set()*# 未爬取 URL 集合*

self.old\_urls = set()*# 已爬取 URL 集合*

**def** **has\_new\_url**(self):

'''

判断是否有未爬取的 URL

:return:

'''

**return** self.new\_url\_size()!=0

**def** **get\_new\_url**(self):

'''

获取一个未爬取的 URL

:return:

'''

new\_url = self.new\_urls.pop()

self.old\_urls.add(new\_url)

**return** new\_url

**def** **add\_new\_url**(self,url):

'''

将新的 URL 添加到未爬取的 URL 集合中

:param url:单个 URL

:return:

'''

**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):

'''

将新的 URL 添加到未爬取的 URL 集合中

:param urls:url 集合

:return:

'''

**if** urls **is** **None** **or** len(urls)==0:

**return**

**for** url **in** urls:

self.add\_new\_url(url)

**def** **new\_url\_size**(self):

'''

获取未爬取 URL 集合的大小

:return:

'''

**return** len(self.new\_urls)

**def** **old\_url\_size**(self):

'''

获取已经爬取 URL 集合的大小

:return:

'''

**return** len(self.old\_urls)

# 6.3　HTML 下载器

HTML 下载器用来下载网页，这时候需要注意网页的编码，以保证下载的网页没有乱码。下载器需要用到 Requests 模块，里面只需要实现一个接口即可：download（url）。程序 HtmlDownloader.py 代码如下：

*# coding:utf-8*

**import** requests

**class** **HtmlDownloader**(object):

**def** **download**(self,url):

**if** url **is** **None**:

**return** **None**

user\_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'

headers={'User-Agent':user\_agent}

r = requests.get(url,headers=headers)

**if** r.status\_code==200:

r.encoding='utf-8'

**return** r.text

**return** **None**

# 6.4　HTML 解析器

HTML 解析器使用 BeautifulSoup4 进行 HTML 解析。需要解析的部分主要分为提取相关词条页面的 URL 和提取当前词条的标题和摘要信息。

先使用 Firebug 查看一下标题和摘要所在的结构位置，如图6-4所示。

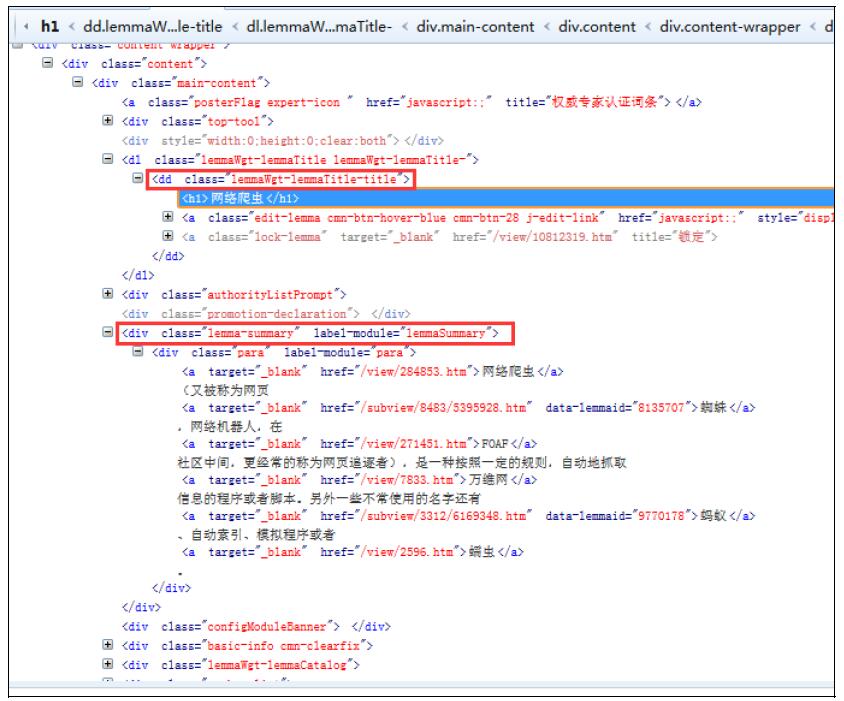
[](http://popimage/?src=)

图6-4　HTML 结构位置

从上图可以看到标题的标记位于<dd class=“lemmaWgt-lemmaTitle-title”><h1></h1>，摘要文本位于<div class=“lemma-summary”label-module=“lemmaSummary”>。

最后分析一下需要抽取的 URL 的格式。相关词条的 URL 格式类似于<a target=“\_blank”href=“/view/7833.htm”>万维网</a>这种形式，提取出 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 HTML 下载器返回的网页内容。解析器 HtmlParser.py 程序的代码如下：

*# coding:utf-8*

**import** re

**import** urlparse

**from** bs4 **import** BeautifulSoup

**class** **HtmlParser**(object):

**def** **parser**(self,page\_url,html\_cont):

'''

用于解析网页内容，抽取 URL 和数据

:param page\_url: 下载页面的 URL

:param html\_cont: 下载的网页内容

:return:返回 URL 和数据

'''

**if** page\_url **is** **None** **or** html\_cont **is** **None**:

**return**

soup = BeautifulSoup(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):

'''

抽取新的 URL 集合

:param page\_url: 下载页面的 URL

:param soup:soup

:return: 返回新的 URL 集合

'''

new\_urls = set()

*# 抽取符合要求的 a 标记*

links = soup.find\_all('a',href=re.compile(r'/view/\d+\.htm'))

**for** link **in** links:

*# 提取href属性*

new\_url = link['href']

*# 拼接成完整网址*

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):

'''

抽取有效数据

:param page\_url:下载页面的 URL

:param soup:

:return:返回有效数据

'''

data={}

data['url']=page\_url

title = soup.find('dd',class\_='lemmaWgt-lemmaTitle-title').find('h1')

data['title']=title.get\_text()

summary = soup.find('div',class\_='lemma-summary')

*# 获取 tag 中包含的所有文本内容，包括子孙 tag 中的内容,并将结果作为 Unicode 字符串返回*

data['summary']=summary.get\_text()

**return** data

# 6.5　数据存储器

数据存储器主要包括两个方法：store\_data（data）用于将解析出来的数据存储到内存中，output\_html（） 用于将存储的数据输出为指定的文件格式，我们使用的是将数据输出为 HTML 格式。DataOutput.py 程序如下：

*# coding:utf-8*

import codecs

**class** **DataOutput**(**object**):

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

**self**.datas=[]

**def** **store\_data**(**self**,data):

**if** data is None:

**return**

**self**.datas.append(data)

**def** **output\_html**(**self**):

fout=codecs.open('baike.html','w',encoding='utf-8')

fout.write("<html>")

fout.write("<body>")

fout.write("<table>")

**for** data **in** **self**.datas:

fout.write("<tr>")

fout.write("<td>%s</td>"%data['url'])

fout.write("<td>%s</td>"%data['title'])

fout.write("<td>%s</td>"%data['summary'])

fout.write("</tr>")

**self**.datas.remove(data)

fout.write("</table>")

fout.write("</body>")

fout.write("</html>")

fout.close()

其实上面的代码并不是很好的方式，更好的做法应该是将数据分批存储到文件，而不是将所有数据存储到内存，一次性写入文件容易使系统出现异常，造成数据丢失。但是由于我们只需要100条数据，速度很快，所以这种方式还是可行的。如果数据很多，还是采取分批存储的办法。

# 6.6　爬虫调度器

以上几节已经对 URL 管理器、HTML 下载器、HTML 解析器和数据存储器等模块进行了实现，接下来编写爬虫调度器以协调管理这些模块。爬虫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，方法内部实现按照运行流程控制各个模块的工作。爬虫调度器 SpiderMan.py 的程序如下：

*# coding:utf-8*

from firstSpider.DataOutput import DataOutput

from firstSpider.HtmlDownloader import HtmlDownloader

from firstSpider.HtmlParser import HtmlParser

from firstSpider.UrlManager import UrlManager

**class** **SpiderMan**(**object**):

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

**self**.manager = UrlManager()

**self**.downloader = HtmlDownloader()

**self**.parser = HtmlParser()

**self**.output = DataOutput()

**def** **crawl**(**self**,root\_url):

*# 添加入口 URL*

**self**.manager.add\_new\_url(root\_url)

*# 判断 url 管理器中是否有新的 url，同时判断抓取了多少个 url*

**while**(**self**.manager.has\_new\_url() **and** **self**.manager.old\_url\_size()<100):

try:

*# 从 URL 管理器获取新的 url*

new\_url = **self**.manager.get\_new\_url()

*# HTML下 载器下载网页*

html = **self**.downloader.download(new\_url)

*# HTML 解析器抽取网页数据*

new\_urls,data = **self**.parser.parser(new\_url,html)

*# 将抽取的 url 添加到 URL 管理器中*

**self**.manager.add\_new\_urls(new\_urls)

*# 数据存储器存储文件*

**self**.output.store\_data(data)

print "已经抓取%s个链接"%**self**.manager.old\_url\_size()

except Exception,e:

print "crawl failed"

*# 数据存储器将文件输出成指定格式*

**self**.output.output\_html()

**if** \_\_name\_\_=="\_\_main\_\_":

spider\_man = SpiderMan()

spider\_man.crawl("http://baike.baidu.com/view/284853.htm")

到这里基础爬虫架构所需的模块都已经完成，启动程序，大约1分钟左右，数据都被存储为 baike.html。使用浏览器打开，效果如图6-5所示。

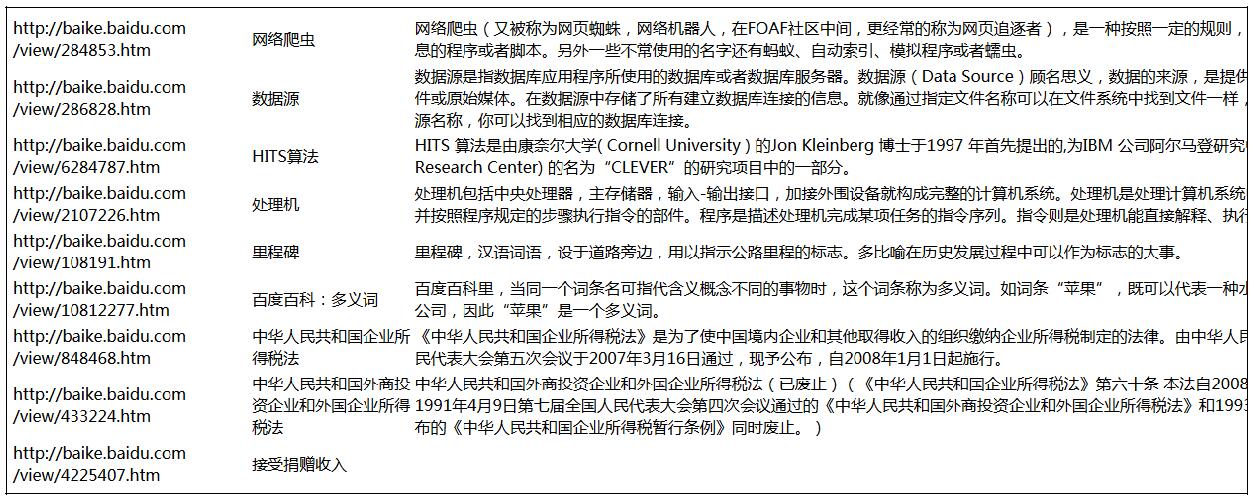
[](http://popimage/?src=)

图6-5　baike.html