# 请求库

## urllib

urllib库是Python内置的HTTP请求库。包含如下4个模块。

* request：它是最基本的HTTP请求模块，可以用来模拟发送请求。就像在浏览器里输入网址然后回车一样，只需要给库方法传入URL以及额外的参数，就可以模拟实现这个过程了。
* error：异常处理模块，如果出现请求错误，我们可以捕获这些异常，然后进行重试或其他操作以保证程序不会意外终止。
* parse：一个工具模块，提供了许多URL处理方法，比如拆分、解析、合并等。
* robotparser：主要是用来识别网站的robots.txt文件，然后判断哪些网站可以爬，哪些网站不可以爬，它其实用得比较少。

### 发送请求

#### User Agent和代理IP隐藏身份

User Agent的中文名为用户代理，简称UA。User Agent的中文名为用户代理，简称UA。

常见的User Agent

1.Android

Mozilla/5.0 (Linux; Android 4.1.1; Nexus 7 Build/JRO03D) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.166 Safari/535.19

2.Firefox

Mozilla/5.0 (Windows NT 6.2; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0

3.Google Chrome

Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/27.0.1453.94 Safari/537.36

4.iOS

Mozilla/5.0 (iPad; CPU OS 5\_0 like Mac OS X) AppleWebKit/534.46 (KHTML, like Gecko) Version/5.1 Mobile/9A334 Safari/7534.48.3

#### request模块

**urllib.request模块**

**urllib.request.urlopen(url, data=None, [timeout, ]\*, cafile=None, capath=None, cadefault=False, context=None)**

**提供了最基本的构造HTTP请求的方法**

模拟浏览器的一个请求发起过程，同时它还带有处理授权验证（authenticaton）、重定向（redirection)、浏览器Cookies以及其他内容。

from urllib import request  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 向指定的url发送请求，并返回服务器响应的类文件对象  
 response = request.urlopen("https://my.alipay.com/portal/i.htm")  
 html = response.read()  
 print(html.decode('utf-8'))

urlopen()返回的对象为HTTPResposne类型，可以使用read()进行读取，调用status属性可以得到返回结果的状态码

同样也可以使用geturl()方法、info()方法、getcode()方法。

geturl()返回的是一个url的字符串；

info()返回的是一些meta标记的元信息，包括一些服务器的信息；

getcode()返回的是HTTP的状态码，如果返回200表示请求成功。

#### Request

请求中需要加入Headers等信息，就可以利用更强大的Request类来构建

**urllib.request.Request(url, data=None, headers={}, origin\_req\_host=None, unverifiable=False, method=None)**

* 第一个参数url用于请求URL，这是必传参数，其他都是可选参数。
* 第二个参数data如果要传，必须传bytes（字节流）类型的。如果它是字典，可以先用urllib.parse模块里的urlencode()编码。
* 第三个参数headers是一个字典，它就是请求头，我们可以在构造请求时通过headers参数直接构造，也可以通过调用请求实例的add\_header()方法添加。
* 第四个参数origin\_req\_host指的是请求方的host名称或者IP地址。
* 第五个参数unverifiable表示这个请求是否是无法验证的，默认是False，意思就是说用户没有足够权限来选择接收这个请求的结果。例如，我们请求一个HTML文档中的图片，但是我们没有自动抓取图像的权限，这时unverifiable的值就是True`。
* 第六个参数method是一个字符串，用来指示请求使用的方法，比如GET、POST和PUT等。

*from* urllib *import* request, parse  
url = 'http://httpbin.org/post'  
headers = {  
 'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)',  
 'Host': 'httpbin.org'  
}  
dict = {  
 'name': 'Germey'  
}  
data = bytes(parse.urlencode(dict), encoding='utf8')  
req = request.Request(url=url, data=data, headers=headers, method='POST')  
response = request.urlopen(req)  
print(response.read().decode('utf-8'))

headers也可以用add\_header()方法来添加

req = request.Request(url=url, data=data, method='POST')  
req.add\_header('User-Agent', 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)')

#### Handler

urllib.request模块里的BaseHandler类，它是所有其他Handler的父类，它提供了最基本的方法

* HTTPDefaultErrorHandler：用于处理HTTP响应错误，错误都会抛出HTTPError类型的异常。
* HTTPRedirectHandler：用于处理重定向。
* HTTPCookieProcessor：用于处理Cookies。
* ProxyHandler：用于设置代理，默认代理为空。
* HTTPPasswordMgr：用于管理密码，它维护了用户名和密码的表。
* HTTPBasicAuthHandler：用于管理认证，如果一个链接打开时需要认证，那么可以用它来解决认证问题。

#### OpenerDirector

称为Opener，urlopen()实际上就是urllib提供的一个Opener，Request和urlopen()相当于类库为你封装好了极其常用的请求方法，利用它们可以完成基本的请求，需要实现更高级的功能需要深入一层进行配置，使用更底层的实例来完成操作，就用到了Opener，Handler来构建Opener

##### 代理设置

利用这个Handler及build\_opener()方法构造一个Opener，之后发送请求即可

*from* urllib.error *import* URLError  
*from* urllib.request *import* ProxyHandler, build\_opener  
  
proxy\_handler = ProxyHandler({  
 'http': 'http://127.0.0.1:9743',  
 'https': 'https://127.0.0.1:9743'  
})  
opener = build\_opener(proxy\_handler)  
*try*:  
 response = opener.open('https://www.baidu.com')  
 print(response.read().decode('utf-8'))  
*except* URLError *as* e:  
 print(e.reason)

##### Cookies

**将网站的Cookies获取下来**

声明一个CookieJar对象，利用HTTPCookieProcessor来构建一个Handler，最后利用build\_opener()方法构建出Opener，执行open()函数

1. 变量方式保存

*import* http.cookiejar, urllib.request  
  
cookie = http.cookiejar.CookieJar()  
handler = urllib.request.HTTPCookieProcessor(cookie)  
opener = urllib.request.build\_opener(handler)  
response = opener.open('http://www.baidu.com')

1. 文本形式保存

CookieJar需要换成MozillaCookieJar，它在生成文件时会用到，是CookieJar的子类，可以用来处理Cookies和文件相关的事件，将Cookies保存成Mozilla型浏览器的Cookies格式

filename = 'cookies.txt'  
cookie = http.cookiejar.MozillaCookieJar(filename)  
handler = urllib.request.HTTPCookieProcessor(cookie)  
opener = urllib.request.build\_opener(handler)  
response = opener.open('http://www.baidu.com')  
cookie.save(ignore\_discard=*True*, ignore\_expires=*True*)

**读取cookies**

cookie = http.cookiejar. MozillaCookieJar ()  
cookie.load('cookies.txt', ignore\_discard=*True*, ignore\_expires=*True*)  
handler = urllib.request.HTTPCookieProcessor(cookie)  
opener = urllib.request.build\_opener(handler)  
response = opener.open('http://www.baidu.com')  
print(response.read().decode('utf-8'))

### 处理异常

urllib的error模块定义了由request模块产生的异常。如果出现了问题，request模块便会抛出error模块中定义的异常

urllib.error有两个方法，URLError和HTTPError。

URLError是OSError的一个子类，HTTPError是URLError的一个子类，服务器上HTTP的响应会返回一个状态码，根据这个HTTP状态码，我们可以知道我们的访问是否成功。

#### URLError

由request模块生的异常都可以通过捕获这个类来处理，属性reason，返回错误的原因

*from* urllib *import* request, error  
*try*:  
 response = request.urlopen('http://cuiqingcai.com/index.htm')  
*except* error.URLError *as* e:  
 print(e.reason)

#### HTTPError

专门用来处理HTTP请求错误，比如认证请求失败等，有如下3个属性

* code：返回HTTP状态码，比如404表示网页不存在，500表示服务器内部错误等。
* reason：同父类一样，用于返回错误的原因。
* headers：返回请求头。

*from* urllib *import* request,error  
*try*:  
 response = request.urlopen('http://cuiqingcai.com/index.htm')  
*except* error.HTTPError *as* e:  
 print(e.reason, e.code, e.headers, sep='\n')

#### URLError和HTTPError混合使用

# URLError  
url = 'http://www.iloveyou.com/' # 一个不存在的连接  
# HTTPError  
url = 'http://www.douyu.com/Jack.html' # 请求的资源没有在服务器上找 到,www.douyu.com这个服务器是存在的，但是我们要查找的Jack.html资源是没有的  
req = request.Request(url)  
*try*:  
 response = request.urlopen(req)  
*except* error.URLError *as* e:  
 print(e.reason)  
*except* error.HTTPError *as* e:  
 print(e.code)

如果想用HTTPError和URLError一起捕获异常，那么需要将HTTPError放在URLError的前面，因为HTTPError是URLError的一个子类。如果URLError放在前面，出现HTTP异常会先响应URLError，这样HTTPError就捕获不到错误信息了。

也可以使用hasattr函数判断URLError含有的属性，如果含有reason属性表明是URLError，如果含有code属性表明是HTTPError

### 解析链接

urllib库里还提供parse模块，定义了处理URL的标准接口

#### urlparse()

实现URL的识别和分段，根据分隔符将URL拆分

result = urlparse('http://www.baidu.com/index.html#comment', allow\_fragments=False)

print(result)

运行结果如下：

ParseResult(scheme='http', netloc='www.baidu.com', path='/index.html#comment', params='', query='', fragment='')

#### urlunparse()

实现URL的构造，接受的参数是一个可迭代对象，但是它的长度必须是6，否则会抛出参数数量不足或者过多的问题

data = ['http', 'www.baidu.com', 'index.html', 'user', 'a=6', 'comment']

#### urlsplit()

和urlparse()方法非常相似，只不过它不再单独解析params这一部分，只返回5个结果。

#### urljoin()

生成链接，提供一个base\_url（基础链接）作为第一个参数，将新的链接作为第二个参数，该方法会分析base\_url的scheme、netloc和path这3个内容并对新链接缺失的部分进行补充，最后返回结果

urljoin('http://www.baidu.com', 'FAQ.html')

#### urlencode()

params = {  
 'name': 'germey',  
 'age': 22  
}  
base\_url = 'http://www.baidu.com?'  
url = base\_url + urlencode(params)

声明一个字典来将参数表示出来，然后调用urlencode()方法将其序列化为GET请求参数

#### parse\_qs()

GET请求参数，利用parse\_qs()方法，就可以将它转回字典

#### parse\_qsl()

将参数转化为元组组成的列表

query = 'name=germey&age=22'

print(parse\_qsl(query))

运行结果如下：

[('name', 'germey'), ('age', '22')]

#### quote()

URL中带有中文参数时，有时可能会导致乱码的问题，此时用这个方法可以将中文字符转化为URL编码

keyword = '壁纸'

url = 'https://www.baidu.com/s?wd=' + quote(keyword)

print(url)

运行结果如下：

<https://www.baidu.com/s?wd=%E5%A3%81%E7%BA%B8>

#### unquote()

进行URL解码

url = 'https://www.baidu.com/s?wd=%E5%A3%81%E7%BA%B8'

print(unquote(url))

### Robots协议

Robots协议也称作爬虫协议、机器人协议，它的全名叫作网络爬虫排除标准（Robots Exclusion Protocol），用来告诉爬虫和搜索引擎哪些页面可以抓取，哪些不可以抓取。它通常是一个叫作robots.txt的文本文件，一般放在网站的根目录下。

当搜索爬虫访问一个站点时，它首先会检查这个站点根目录下是否存在robots.txt文件，如果存在，搜索爬虫会根据其中定义的爬取范围来爬取。如果没有找到这个文件，搜索爬虫便会访问所有可直接访问的页面。

User-agent: \*

Disallow: /

Allow: /public/

User-agent描述了搜索爬虫的名称，这里将其设置为\*则代表该协议对任何爬取爬虫有效。

Disallow指定了不允许抓取的目录，比如上例子中设置为/则代表不允许抓取所有页面。若全部允许，设置为空

Allow一般和Disallow一起使用，一般不会单独使用，用来排除某些限制。设置为/public/，则表示所有页面不允许抓取，但可以抓取public目录。

#### robotparser

该模块提供了一个类RobotFileParser，它可以根据某网站的robots.txt文件来判断一个爬取爬虫是否有权限来爬取这个网页。

**urllib.robotparser.RobotFileParser(url='')**

也可以在声明时不传入，默认为空，最后再使用set\_url()方法设置

* set\_url()：用来设置robots.txt文件的链接。如果在创建RobotFileParser对象时传入了链接，那么就不需要再使用这个方法设置了。
* read()：读取robots.txt文件并进行分析。注意，这个方法执行一个读取和分析操作，如果不调用这个方法，接下来的判断都会为False，所以一定记得调用这个方法。这个方法不会返回任何内容，但是执行了读取操作。
* parse()：用来解析robots.txt文件，传入的参数是robots.txt某些行的内容，它会按照robots.txt的语法规则来分析这些内容。
* can\_fetch()：该方法传入两个参数，第一个是User-agent，第二个是要抓取的URL。返回的内容是该搜索引擎是否可以抓取这个URL，返回结果是True或False。
* mtime()：返回的是上次抓取和分析robots.txt的时间，这对于长时间分析和抓取的搜索爬虫是很有必要的，你可能需要定期检查来抓取最新的robots.txt。
* modified()：它同样对长时间分析和抓取的搜索爬虫很有帮助，将当前时间设置为上次抓取和分析robots.txt的时间。

*from* urllib.robotparser *import* RobotFileParser  
  
rp = RobotFileParser()  
rp.set\_url('http://www.jianshu.com/robots.txt')  
rp.read()  
print(rp.can\_fetch('\*', 'http://www.jianshu.com/p/b67554025d7d'))  
print(rp.can\_fetch('\*', "http://www.jianshu.com/search?q=python&page=1&type=collections"))

## requests



### get（）



1. 修改headers字段、cookie、代理

cookies={}

#自动保持  
s = requests.session()

#打开cookie文件  
*with* open('E:\example\豆瓣读书爬虫\cookie.txt')*as* file:  
 raw\_cookies=file.read()  
 *for* line *in* raw\_cookies.split(';'):  
 key,value=line.split('=',1)  
 cookies[key]=value  
proxie = {'http': 'http://14.118.254.31:6666'}

headers={'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0'}  
res=s.get(url=*url*,cookies=cookies,headers=headers,proxies=proxie)

1. 修改params参数提交关键词

百度的关键词接口：http://www.baidu.com/s?wd=keyword

360的关键词接口：<http://www.so.com/s?q=keyword>

url="http://www.baidu.com/s"   
kv={'wd':'Python'}   
r=requests.get(url,params=kv)

1. 获取cookie

response = requests.get("https://www.baidu.com")  
print(response.cookies)

1. 发送cookie

url = 'http://httpbin.org/cookies'  
cookies = {'testCookies\_1': 'Hello\_Python3', 'testCookies\_2': 'Hello\_Requests'}  
# 在Cookie Version 0中规定空格、方括号、圆括号、等于号、逗号、双引号、斜杠、问号、@，冒号，分号等特殊符号都不能作为Cookie的内容。  
r = requests.get(url, cookies=cookies)

1. 证书验证

response = requests.get('https://www.12306.cn', verify=*False*)

1. 超时设置

*from* requests.exceptions *import* ReadTimeout  
*try*:  
 response = requests.get("http://httpbin.org/get", timeout = 0.5)  
 print(response.status\_code)  
*except* ReadTimeout:  
 print('Timeout')

1. 身份认证

*import* requests  
*from* requests.auth *import* HTTPBasicAuth  
r = requests.get('http://localhost:5000', auth=HTTPBasicAuth('username', 'password'))

如果用户名和密码正确的话，请求时就会自动认证成功，会返回200状态码，如果认证失败，则返回401状态码。requests提供了一个更简单的写法，可以直接传一个元组，它会默认使用HTTPBasicAuth这个类来认证。

r = requests.get('http://localhost:5000', auth=('username', 'password'))

**requests库的response对象：**

response对象包含服务器返回的所有信息，也包含请求的request信息。



**抓取网页**

如果请求普通的网页,调用.text属性，获得相应的内容

如果返回的是json格式，调用json()方法变为字典格式

**抓取二进制数据**

图片、音频、视频这些文件本质上都是由二进制码组成的，想要抓取它们，就要调用.content拿到它们的二进制码，为bytes类型

*import* requests  
  
r = requests.get("https://github.com/favicon.ico")  
*with* open('favicon.ico', 'wb') *as* f:  
 f.write(r.content)

open()方法，第一个参数是文件名称，第二个参数代表以二进制写的形式打开，可以向文件里写入二进制数据。

音频和视频文件也可以用这种方法获取

### post（）

数据通过post提交

*import* requests  
  
data = {'name': 'germey', 'age': '22'}  
r = requests.post("http://httpbin.org/post", data=data)  
print(r.text)

### 文件上传

*import* requests  
files = {'file': open('favicon.ico', 'rb')}  
r = requests.post("http://httpbin.org/post", files=files)  
print(r.text)

文件需要和当前脚本在同一目录下，文件上传部分会单独有一个files字段来标识。

### 会话维持

requests中，如果直接利用get()或post()等方法的确可以做到模拟网页的请求，但是这实际上是相当于不同的会话，也就是说相当于你用了两个浏览器打开了不同的页面。

第一个请求利用post()方法登录了某个网站，第二次想获取成功登录后的自己的个人信息，你又用了一次get()方法去请求个人信息页面。实际上，这相当于打开了两个浏览器，是两个完全不相关的会话。

解决这个问题的主要方法就是维持同一个会话，也就是相当于打开一个新的浏览器选项卡而不是新开一个浏览器

Session对象可以方便地维护一个会话，而且不用担心cookies的问题

*import* requests  
  
s = requests.Session()  
s.get('http://httpbin.org/cookies/set/number/123456789')  
r = s.get('http://httpbin.org/cookies')  
print(r.text)

利用Session，可以做到模拟同一个会话而不用担心Cookies的问题。它通常用于模拟登录成功之后再进行下一步的操作。

Prepared Request

*from* requests *import* Request, Session  
url = 'http://httpbin.org/post'  
data = {  
 'name': 'germey'  
}  
headers = {  
 'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_11\_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.116 Safari/537.36'  
}  
s = Session()  
req = Request('POST', url, data=data, headers=headers)  
prepped = s.prepare\_request(req)  
r = s.send(prepped)  
print(r.text)

引入Request，用url、data和headers参数构造了一个Request对象，再调用Session的prepare\_request()方法将其转换为一个Prepared Request对象，然后调用send()方法发送

## selenium

自动化测试工具。它支持各种浏览器。如果用Selenium来驱动浏览器加载网页的话，就可以直接拿到JavaScript渲染的结果了

1. **声明浏览器对象**

的初始化并将其赋值为browser对象

from selenium import webdriver

browser = webdriver.Chrome()

browser = webdriver.Firefox()

browser = webdriver.Edge()

browser = webdriver.PhantomJS()

browser = webdriver.Safari()

1. **访问页面**

get()方法来请求网页，参数传入链接URL即可。

browser.get('https://www.taobao.com')

browser.page\_source page\_source输出网页源码

1. **查找节点**
2. 获取单个节点

find\_element\_by\_id

find\_element\_by\_name

find\_element\_by\_xpath

find\_element\_by\_link\_text

find\_element\_by\_partial\_link\_text

find\_element\_by\_tag\_name

find\_element\_by\_class\_name

find\_element\_by\_css\_selector CSS选择器

通用方法find\_element()，它需要传入两个参数：查找方式By和值

find\_element\_by\_id(id)就等价于find\_element(By.ID, id)

1. 获取多个节点

如果有多个节点，用find\_element()方法查找，就只能得到第一个节点

find\_elements\_by\_id

find\_elements\_by\_name

find\_elements\_by\_xpath

find\_elements\_by\_link\_text

find\_elements\_by\_partial\_link\_text

find\_elements\_by\_tag\_name

find\_elements\_by\_class\_name

find\_elements\_by\_css\_selector

1. **节点交互**

输入文字时用send\_keys()方法，清空文字时用clear()方法，点击按钮时用click()方法。

1. **动作链**

一些交互动作都是针对某个节点执行的。比如，对于输入框，我们就调用它的输入文字和清空文字方法。

一些操作没有特定的执行对象，比如鼠标拖曳、键盘按键等，这些动作用动作链来执行

*from* selenium *import* webdriver  
*from* selenium.webdriver *import* ActionChains  
  
browser = webdriver.Chrome()  
url = 'http://www.runoob.com/try/try.php?filename=jqueryui-api-droppable'  
browser.get(url)  
browser.switch\_to.frame('iframeResult')  
source = browser.find\_element\_by\_css\_selector('#draggable')  
target = browser.find\_element\_by\_css\_selector('#droppable')  
actions = ActionChains(browser)  
actions.drag\_and\_drop(source, target)  
actions.perform()

首先，打开网页中的一个拖曳实例，然后依次选中要拖曳的节点和拖曳到的目标节点，接着声明ActionChains对象并将其赋值为actions变量，然后通过调用actions变量的drag\_and\_drop()方法，再调用perform()方法执行动作，此时就完成了拖曳操作

1. **执行JavaScript**

browser = webdriver.Chrome()  
browser.get('https://www.zhihu.com/explore')  
browser.execute\_script('window.scrollTo(0,document.body.scrollHeight)')  
browser.execute\_script('alert("To Bottom")')

execute\_script()方法将进度条下拉到最底部，然后弹出alert提示框

1. **获取节点信息**

通过page\_source属性可以获取网页的源代码，接着就可以使用解析库（如正则表达式、Beautiful Soup、pyquery等）来提取信息

Selenium返回的是WebElement类型，也有相关的方法和属性来直接提取节点信息

1. 获取属性

选中这个节点，**get\_attribute()**方法来获取节点的属性，通过get\_attribute()方法，传入想要获取的属性名，就可以得到它的值了。

logo = browser.find\_element\_by\_id('zh-top-link-logo')

print(logo.get\_attribute('class'))

1. 获取文本值

每个WebElement节点都有text属性，直接调用这个属性就可以得到节点内部的文本信息，这相当于Beautiful Soup的get\_text()方法、pyquery的text()方法

browser.get(url)

input = browser.find\_element\_by\_class\_name('zu-top-add-question')

print(input.text)

1. 获取id、位置、标签名和大小

id属性可以获取节点id，location属性可以获取该节点在页面中的相对位置，tag\_name属性可以获取标签名称，size属性可以获取节点的大小，也就是宽高

input = browser.find\_element\_by\_class\_name('zu-top-add-question')

print(input.id)

print(input.location)

print(input.tag\_name)

print(input.size)

1. 切换Frame

网页中有一种节点叫作iframe，也就是子Frame，相当于页面的子页面，它的结构和外部网页的结构完全一致

Selenium打开页面后，它默认是在父级Frame里面操作，而此时如果页面中还有子Frame，它是不能获取到子Frame里面的节点的。使用switch\_to.frame()方法来切换Frame

browser.get(url)

browser.switch\_to.frame('iframeResult')

1. 延时等待

get()方法会在网页框架加载结束后结束执行，此时如果获取page\_source，可能并不是浏览器完全加载完成的页面

1. 隐式等待 implicitly\_wait()

当查找节点而节点并没有立即出现的时候，隐式等待将等待一段时间再查找DOM，默认的时间是0，超出设定时间后，则抛出找不到节点的异常

browser = webdriver.Chrome()  
browser.implicitly\_wait(10)  
browser.get('https://www.zhihu.com/explore')  
input = browser.find\_element\_by\_class\_name('zu-top-add-question')

1. 显式等待

它指定要查找的节点，然后指定一个最长等待时间。如果在规定时间内加载出来了这个节点，就返回查找的节点；如果到了规定时间依然没有加载出该节点，则抛出超时异常

*from* selenium *import* webdriver  
*from* selenium.webdriver.common.by *import* By  
*from* selenium.webdriver.support.ui *import* WebDriverWait  
*from* selenium.webdriver.support *import* expected\_conditions *as* EC  
browser = webdriver.Chrome()  
browser.get('https://www.taobao.com/')  
wait = WebDriverWait(browser, 10)  
input = wait.until(EC.presence\_of\_element\_located((By.ID, 'q')))  
button = wait.until(EC.element\_to\_be\_clickable((By.CSS\_SELECTOR, '.btn-search')))  
print(input, button)

引入WebDriverWait这个对象，指定最长等待时间

调用它的until()方法，传入要等待条件expected\_conditions

presence\_of\_element\_located这个条件，代表节点出现的意思，其参数是节点的定位元组，也就是ID为q的节点搜索框

效果就是，在10秒内如果ID为q的节点（即搜索框）成功加载出来，就返回该节点；如果超过10秒还没有加载出来，就抛出TimeoutException异常。

| **等待条件** | **含义** |
| --- | --- |
| title\_is | 标题是某内容 |
| title\_contains | 标题包含某内容 |
| presence\_of\_element\_located | 节点加载出来，传入定位元组，如(By.ID, 'p') |
| visibility\_of\_element\_located | 节点可见，传入定位元组 |
| visibility\_of | 可见，传入节点对象 |
| presence\_of\_all\_elements\_located | 所有节点加载出来 |
| text\_to\_be\_present\_in\_element | 某个节点文本包含某文字 |
| text\_to\_be\_present\_in\_element\_value | 某个节点值包含某文字 |
| frame\_to\_be\_available\_and\_switch\_to\_it | 加载并切换 |
| invisibility\_of\_element\_located | 节点不可见 |
| element\_to\_be\_clickable | 节点可点击 |
| staleness\_of | 判断一个节点是否仍在DOM，可判断页面是否已经刷新 |
| element\_to\_be\_selected | 节点可选择，传节点对象 |
| element\_located\_to\_be\_selected | 节点可选择，传入定位元组 |
| element\_selection\_state\_to\_be | 传入节点对象以及状态，相等返回True，否则返回False |
| element\_located\_selection\_state\_to\_be | 传入定位元组以及状态，相等返回True，否则返回False |
| alert\_is\_present | 是否出现警告 |

1. 前进和后退 back()方法后退， forward()方法前进
2. cookies

browser = webdriver.Chrome()  
browser.get('https://www.zhihu.com/explore')  
browser.get\_cookies()  
browser.add\_cookie({'name': 'name', 'domain': 'www.zhihu.com', 'value': 'germey'})  
browser.delete\_all\_cookies()

加载完成后，浏览器实际上已经生成Cookies了。接着，调用get\_cookies()方法获取所有的Cookies。

添加一个Cookie，这里传入一个字典，有name、domain和value等内容

接下来，再次获取所有的Cookies。可以发现，结果就多了这一项新加的Cookie。

调用delete\_all\_cookies()方法删除所有的Cookies。

1. 选项卡管理

browser = webdriver.Chrome()  
browser.get('https://www.baidu.com')  
browser.execute\_script('window.open()')  
print(browser.window\_handles)  
browser.switch\_to\_window(browser.window\_handles[1])

调用execute\_script()方法，传入window.open()这个JavaScript语句新开启一个选项卡。

window\_handles属性获取当前开启的所有选项卡，返回的是选项卡的代号列表

切换选项卡，调用switch\_to\_window()方法即可，其中参数是选项卡的代号

1. 异常处理

可以使用try except语句来捕获各种异常, 一旦出现错误，就进行异常处理

browser = webdriver.Chrome()  
*try*:  
 browser.get('https://www.baidu.com')  
*except* TimeoutException:  
 print('Time Out')

CREATE TABLE `allbooks` (

`title` char(255) NOT NULL,

`author` char(255) DEFAULT NULL,

`money` char(255) DEFAULT NULL,

`publish` char(255) DEFAULT NULL,

`pages` char(255) DEFAULT NULL,

`years` char(255) DEFAULT NULL,

`ISBN` char(255) DEFAULT NULL,

`people` char(255) DEFAULT NULL,

`score` char(255) DEFAULT NULL,

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## aiohttp

Requests库是一个阻塞式HTTP请求库，当我们发出一个请求后，程序会一直等待服务器响应，直到得到响应后，程序才会进行下一步处理

aiohttp是一个提供异步Web服务的库，使用异步请求库进行数据抓取时，会大大提高效率

## PhantomJS

使用 Chrome 或 Firefox 进行网页抓取的话，每次抓取的时候，都会弹出一个浏览器，比较影响使用。一个无界面浏览器，叫做 PhantomJS

# 解析库

## LXML

LXML全称XML Path Language，即XML路径语言， 是 Python 的一个解析库，支持 HTML 和 XML 的解析，支持 XPath 解析方式，而且解析效率非常高

### XPath常用规则

* nodename 选取此节点的所有子节点
* / 从当前节点选取直接子节点
* // 从当前节点选取子孙节点
* . 选取当前节点
* .. 选取当前节点的父节点
* @ 选取属性

*from* lxml *import* etree  
  
html = etree.parse('./test.html', etree.HTMLParser())  
result = etree.tostring(html)  
print(result.decode('utf-8'))

导入lxml库的etree模块, 读取文本文件，调用HTML类进行初始化，这样就成功构造了一个XPath解析对象. etree模块可以自动修正HTML文本

调用tostring()方法即可输出修正后的HTML代码，但是结果是bytes类型。这里利用decode()方法将其转成str类型

### 属性匹配

用@符号进行属性过滤，返回列表

result = html.xpath('//li[@class="item-0"]') class为item-0的li节点

### 文本获取

用XPath中的text()方法获取节点中的文本

result = html.xpath('//li[@class="item-0"]/text()')

### 属性获取

用@符号，以列表形式返回

result = html.xpath('//li/a/@href')

### 属性多值匹配

某些节点的某个属性可能有多个值，用contains()函数

result = html.xpath('//li[contains(@class, "li")]/a/text()')

通过contains()方法，第一个参数传入属性名称，第二个参数传入属性值，只要此属性包含所传入的属性值，就可以完成匹配，返回列表

### 多属性匹配

根据多个属性确定一个节点，这时就需要同时匹配多个属性。使用运算符and来连接, 相连之后置于中括号内进行条件筛选

r=html.xpath('//li[contains(@class,"li")and@name="item"]/a/text()')

### 按序选择

中括号传入索引的方法获取特定次序的节点

result = html.xpath('//li[1]/a/text()') 第一个li节点

result = html.xpath('//li[last()]/a/text()') 最后一个li节点

result = html.xpath('//li[position()<3]/a/text()') 位置小于3的li节点

result = html.xpath('//li[last()-2]/a/text()') 倒数第三个li节点

## BeautifulSoup

BeautifulSoup 的 HTML 和 XML 解析器是依赖于 LXML 库的

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。

初始化BeautifulSoup自动更正格式, 调用prettify()方法可以把要解析的字符串以标准的缩进格式输出。

soup = BeautifulSoup(html, 'lxml')  
print(soup.prettify())  
print(soup.title.string)

### Beautiful Soup对象

Beautiful Soup四大对象种类

Beautiful Soup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

#### tag

**Beautiful Soup获取 Tags**

它查找的是在所有内容中的第一个符合要求的标签

soup = BeautifulSoup(html, 'lxml')

soup.标签名.name/attrs

Tag：Tag有两个重要的属性：name和attrs。

1. name 即为 [document]，对于其他内部标签，输出的值便为标签本身的名称。
2. attrs 标签的所有属性打印输出，得到的类型是一个字典。

print soup.p.attrs

#{'class': ['title'], 'name': 'dromouse'}

如果需要**单独获得**某个属性可以使用字典的打印方法或者get(‘key’)

print soup.p['class']

print soup.p.get('class')

可以对这些属性和内容等等进行修改 soup.p['class']="newClass"

对这个属性进行删除: del soup.p['class']

NavigableString：获取标签内部的文字 soup.标签名.string

BeautifulSoup：表示的是一个文档的全部内容

Comment：特殊类型的NavigableString对象，首先判断的类型是否为 Comment 类型，然后再进行其他操作，如打印输出

if type(soup.li.string) == element.Comment:

print(soup.li.string)

### 遍历文档树

#### 直接子节点(不包含孙节点)

1. .contents

tag的content属性可以将tag的子节点以列表的方式输出：soup.标签名.contents

可以用列表索引来获取它的某一个元素: soup.标签名.contents[1]

1. .children

children是一个 list 生成器对象,返回的不是一个 list, 可以通过遍历获取所有子节点

for child in soup. 标签名.children:

print(child)

#### 所有子孙节点

.descendants 属性

.contents 和 .children 属性仅包含tag的直接子节点，.descendants 属性可以对所有tag的子孙节点进行递归循环，和 children类似，我们也需要遍历获取其中的内容。

for child in soup.descendants:

print child

#### 节点内容

* 如果tag只有一个 NavigableString 类型子节点,那么这个tag可以使用 .string 得到子节点。
* 如果一个tag仅有一个子节点,那么这个tag也可以使用 .string 方法,输出结果与当前唯一子节点的 .string 结果相同（如果一个标签里面没有标签了，那么 .string 就会返回标签里面的内容。如果标签里面只有唯一的一个标签了，那么 .string 也会返回最里面的内容。）
* 如果tag包含了多个子节点,tag就无法确定，string 方法应该调用哪个子节点的内容, .string 的输出结果是 None

1. 多个内容

.strings .stripped\_strings 属性

for string in soup.stripped\_strings:

print(repr(string))

输出的字符串中可能包含了很多空格或空行,使用 .stripped\_strings 可以去除多余空白内容

1. 父节点
2. .parent 属性

content = soup.head.title.string

print content.parent.name

1. .parents 属性

递归得到元素的所有父辈节点

content = soup.head.title.string

for parent in content.parents:

print parent.name

1. 兄弟节点
2. .next\_sibling .previous\_sibling 属性

.next\_sibling 属性获取了该节点的下一个兄弟节点，.previous\_sibling 则与之相反，如果节点不存在，则返回 None

1. .next\_siblings .previous\_siblings 属性

对当前节点的兄弟节点迭代输出

for sibling in soup.a.next\_siblings:

print(repr(sibling))

1. 前后节点
2. .next\_element .previous\_element 属性

所有节点，不分层次

比如 head 节点为

<head><title>The Dormouse's story</title></head>

print soup.head.next\_element

那么它的下一个节点便是 title，它是不分层次关系的

<title>The Dormouse's story</title>

1. .next\_elements .previous\_elements 属性

可以向前或向后访问文档的解析内容

for element in last\_a\_tag.next\_elements:

print(repr(element))

### 搜索文档树

#### find\_all()

**find\_all(name, attrs, recursive, text, limit, \*\*kwargs)**

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

1. name参数

查找所有名字为name的tag,字符串对象会被自动忽略掉

* 1. 传递字符：soup.find\_all('b')

搜索方法中传入一个字符串参数，Beautiful Soup会查找与字符串完整匹配的内容

* 1. 传递正则表达式：通过正则表达式的 match() 来匹配内容

for tag in soup.find\_all(re.compile("^b")):

print(tag.name)

# body

# b

* 1. 传递列表：将与列表中任一元素匹配的内容返回

soup.find\_all(["a", "b"])

# [<b>The Dormouse's story</b>,

#  <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,

#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,

#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

* 1. 传递True：可以匹配任何值，查找到所有的tag，但是不会返回字符串节点

for tag in soup.find\_all(True):

    print(tag.name)

# html

# head

# title

# body

* 1. 传递方法：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,如果不是则反回 False

def has\_class\_but\_no\_id(tag): 包含 class 属性却不包含 id 属性

    return tag.has\_attr('class') and not tag.has\_attr('id')

soup.find\_all(has\_class\_but\_no\_id)

# [<p class="title"><b>The Dormouse's story</b></p>,

#  <p class="story">Once upon a time there were...</p>,

#  <p class="story">...</p>]

1. attrs参数：定义一个字典参数来搜索包含特殊属性的tag

soup.find\_all (attrs={"data-foo": "value"})

1. recursive参数：调用tag的 find\_all() 方法时,Beautiful Soup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。

soup.html.find\_all("title", recursive=False)

1. text参数：搜索文档中的字符串，接受字符串，正则表达式 ，列表，True

soup.find\_all(text=["Tillie", "Elsie", "Lacie"])

# [u'Elsie', u'Lacie', u'Tillie']

soup.find\_all(text=re.compile("Dormouse"))

[u"The Dormouse's story", u"The Dormouse's story"]

1. limit参数：find\_all() 方法返回全部的搜索结构，使用 limit 参数限制返回结果的数量

soup.find\_all("a", limit=2)

1. kwargs参数：kwargs 接收字符串，正则表达式

如果一个指定名字的参数不是搜索内置的参数名,搜索时会把该参数当作指定名字tag的属性来搜索,如果包含一个名字为 id 的参数,Beautiful Soup会搜索每个tag的”id”属性

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("elsie"), id='link1')

soup.find\_all("a", class\_="sister") 使用class 过滤，加个下划线

#### 更多find方式

1. **find( name , attrs , recursive , text , \*\*kwargs )**

与 find\_all() 方法唯一的区别是 find\_all() 方法的返回结果是值包含一个元素的列表,而 find() 方法直接返回结果

1. **find\_parents() find\_parent()**

find\_all() 和 find() 只搜索当前节点的所有子节点,孙子节点等. find\_parents() 和 find\_parent() 用来搜索当前节点的父辈节点,搜索方法与普通tag的搜索方法相同,搜索文档搜索文档包含的内容

1. **find\_next\_siblings() find\_next\_sibling()**

这2个方法通过 .next\_siblings 属性对当 tag 的所有后面解析的兄弟 tag 节点进行迭代, find\_next\_siblings() 方法返回所有符合条件的后面的兄弟节点,find\_next\_sibling() 只返回符合条件的后面的第一个tag节点

1. **find\_previous\_siblings() find\_previous\_sibling()**

这2个方法通过 .previous\_siblings 属性对当前 tag 的前面解析的兄弟 tag 节点进行迭代, find\_previous\_siblings() 方法返回所有符合条件的前面的兄弟节点, find\_previous\_sibling() 方法返回第一个符合条件的前面的兄弟节点

1. **find\_all\_next() find\_next()**

这2个方法通过 .next\_elements 属性对当前 tag 的之后的 tag 和字符串进行迭代, find\_all\_next() 方法返回所有符合条件的节点, find\_next() 方法返回第一个符合条件的节点

1. **find\_all\_previous() 和 find\_previous()**

这2个方法通过 .previous\_elements 属性对当前节点前面的 tag 和字符串进行迭代, find\_all\_previous() 方法返回所有符合条件的节点, find\_previous()方法返回第一个符合条件的节点

#### css选择器

**soup.select()，返回类型是 list**

1. 通过标签名查找 soup.select('title')
2. 通过类名查找 soup.select('.sister')
3. 通过 id 名查找 soup.select('#link1')
4. 组合查找 soup.select('p #link1') p 标签中，id 等于 link1的内容

空格分开

soup.select("head > title") 直接子标签查找

1. 属性查找 soup.select('a[class="sister"]')

属性和标签属于同一节点，中间不能加空格

属性仍然可以与上述查找方式组合，不在同一节点的空格隔开，同一节点的不加空格

select 方法返回的结果都是列表形式，可以遍历形式输出，然后用 get\_text() 方法来获取它的内容

soup = BeautifulSoup(html, 'lxml')  
print type(soup.select('title'))  
print soup.select('title')[0].get\_text()  
*for* title *in* soup.select('title'):  
 print title.get\_text()

1. 获取属性

获取属性还可以用原来的方法

soup = BeautifulSoup(html, 'lxml')  
*for* ul *in* soup.select('ul'):  
 print(ul['id'])  
 print(ul.attrs['id'])

1. 获取文本

可以用string或者get\_text()，二者的效果完全一致

## PyQuery

提供了和 jQuery 类似的语法来解析 HTML 文档，支持 CSS 选择器

## Tesserocr

OCR，即 Optical Character Recognition，光学字符识别。是指通过扫描字符，然后通过其形状将其翻译成电子文本的过程。大多数验证码还是图形验证码，这时候我们可以直接用 OCR 来识别。

Tesserocr 是 Python 的一个 OCR 识别库，但其实是对 Tesseract 做的一层 Python API 封装，所以它的核心是 Tesseract

# web库

## Flask

Flask 是一个轻量级的 Web 服务程序，简单、易用、灵活，在本书中我们主要用它来做一些 API 服务

## Tornado

Tornado 是一个支持异步的Web框架，通过使用非阻塞 I/O 流，它可以支撑成千上万的开放连接，效率非常高

# APP库

APP 一些简单的接口通过 Charles 或 MitmProxy 分析找出规律就可以直接用程序模拟来抓取

如果遇到更复杂的接口就需要利用 MitmDump 对接Python来对抓取到的请求和响应进行实时处理和保存

要做规模采集就需要自动化 APP 的操作而不是人工去采集，Appium，它可以像 Selenium 一样对 APP 进行自动化控制，如自动化模拟APP的点击、下拉等操作。

## Charles

相比 Fiddler 来说，Charles 的功能更为强大，而且跨平台支持更好

## MitmProxy

MitmProxy 是一个支持 HTTP 和 HTTPS 的抓包程序，类似 Fiddler、Charles 的功能，只不过它是一个控制台的形式操作。

同时 MitmProxy 还有两个关联组件，一个是 MitmDump，它是 MitmProxy 的命令行接口，利用它我们可以对接 Python 脚本，用 Python 实现监听后的处理。另一个是 MitmWeb，它是一个 Web 程序，通过它我们可以清楚地观察到 MitmProxy 捕获的请求。

## Appium

Appium 是移动端的自动化测试工具，利用它我们可以驱动 Android、iOS 等设备完成自动化测试

Appium 负责驱动移动端来完成一系列操作，对 iOS 设备来说，它使用苹果的 UIAutomation 来实现驱动，对于 Android 来说，它使用 UiAutomator 和 Selendroid 来实现驱动。

# 爬虫框架

## PySpider

PySpider 是支持 JavaScript 渲染的，而这个过程是依赖于 PhantomJS 的

pyspider all启动 PySpider

## Scrapy

## ScrapySplash

ScrapySplash 是一个 Scrapy 中支持 JavaScript 渲染的工具