
Table of Contents

前言	1.1
目录	1.2
第一章 Python基础	1.3
第二章 爬虫基础了解	1.4
第三章 基本库的使用	1.5
3.1 使用urllib	1.6
3.1.1 使用urllib.request发送请求	1.7
3.1.2 使用urllib.error处理异常	1.8
3.1.3 使用urllib.parse解析链接	1.9
3.1.4 使用urllib.robotparser分析robots协议	1.10
3.2 使用requests	1.11
3.2.1 安装requests	1.12
3.2.2 requests的基本使用	1.13
3.2.3 requests的高级使用	1.14

利用**Python3**开发爬虫

本文介绍了利用Python3开发网络爬虫的流程。

目录

前言

第一章 **Python**基础

1.1 开始学习

1.2 变量

1.3 数据结构

1.4 循环与判断

1.5 强大的函数

1.6 强大的第三方库

第二章 基本库的使用

2.1 **urllib**开发第一个爬虫

2.2 强大的库**requests**

2.3 最基础的正则表达式

第三章 多样的解析工具

3.1 **lxml**

3.2 Beautiful Soup

3.3 PyQuery

第四章 高级数据采集

4.1 JavaScript渲染采集

4.2 验证码的处理

4.3 登录验证

4.4 防封杀策略

4.5 自然语言处理

第五章 数据存储

5.1 文本文件存储

5.1.1 纯文本文件存储

5.1.2 JSON文件存储

5.1.3 CSV文件存储

5.1.4 Excel文件存储

5.2 关系型数据库存储

5.2.1 MySQL存储

5.3 非关系型数据库存储

5.3.1 Redis存储

5.3.2 MongoDB存储

5.4 云存储

第六章 数据展示

6.1 Jupyter使用

6.2 HighCharts的使用

6.3 D3.js的使用

第七章 爬虫框架使用

7.1 Scrappy的使用

7.2 PySpider的使用

第八章 分布式爬虫

8.1 分布式爬虫概念

8.2 分布式爬虫架构解析

8.3 分布式爬虫架构实现

第九章 爬虫实战演练

http session cookie 浏览器 network https

基本库的使用

学习爬虫，最初的操作便是来模拟浏览器向服务器发出一个请求，那么我们需要从哪个地方做起呢？请求需要我们自己来构造吗？我们需要关心请求这个数据结构的实现吗？我们需要了解HTTP、TCP、IP层的网络传输通信吗？我们需要知道服务器的响应和应答原理吗？

可能你不知道无从下手，不用担心，Python的强大之处就是提供了功能齐全类库来帮助我们完成这些请求，最基础的HTTP库有urllib,httplib2,requests,treq等。

拿urllib这个库来说，有了它，你只需要关心请求的链接是什么，需要传的参数是什么以及可选的请求头设置就好了，不用深入到底层去了解它到底是怎样来传输和通信的。有了它，你两行代码就可以完成一个请求和响应的处理过程，得到网页内容，是不是感觉方便极了？

接下来，就让我们从最基础的部分开始了解这些库的使用方法吧。

使用urllib

在Python2版本中，有urllib和urllib2两个库可以用来实现request的发送。而在Python3中，已经不存在urllib2这个库了，统一为urllib。

Python3 urllib库官方链接

<https://docs.python.org/3/library/urllib.html>

urllib中包括了四个模块，包括

urllib.request,urllib.error,urllib.parse,urllib.robotparser

- urllib.request可以用来发送request和获取request的结果
- urllib.error包含了urllib.request产生的异常
- urllib.parse用来解析和处理URL
- urllib.robotparser用来解析页面的robots.txt文件

可见其中模拟请求使用的最主要的库便是urllib.request，异常处理用urllib.error库。

下面会对它们一一进行详细的介绍。

使用urllib.request发送请求

urllib.request.urlopen()基本使用

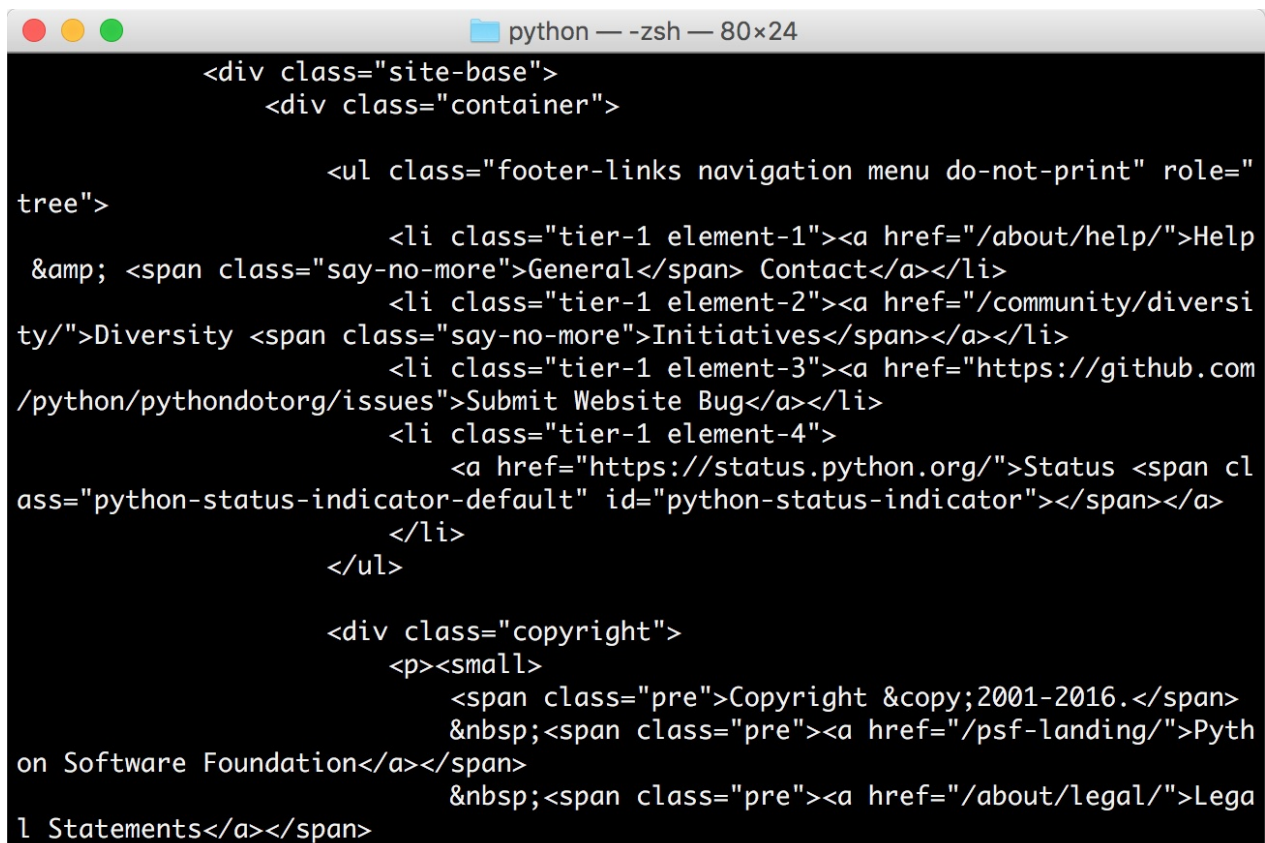
`urllib.request` 模块提供了最基本的构造 HTTP 请求的方法，利用它可以模拟浏览器的一个请求发起过程，同时它还带有处理 `authentication`（授权验证），`redirections`（重定向），`cookies`（浏览器Cookies）以及其它内容。

好，那么首先我们来感受一下它的强大之处，以Python官网为例，我们来把这个网页抓下来。

```
# coding=utf-8
import urllib.request

response = urllib.request.urlopen('https://www.python.org')
print(response.read().decode('utf-8'))
```

看一下运行结果。



```
python — zsh — 80x24
<div class="site-base">
  <div class="container">
    <ul class="footer-links navigation menu do-not-print" role="tree">
      <li class="tier-1 element-1"><a href="/about/help/">Help
        & <span class="say-no-more">General</span> Contact</a></li>
      <li class="tier-1 element-2"><a href="/community/diversity/">Diversity <span class="say-no-more">Initiatives</span></a></li>
      <li class="tier-1 element-3"><a href="https://github.com/python/pythondotorg/issues">Submit Website Bug</a></li>
      <li class="tier-1 element-4">
        <a href="https://status.python.org/">Status <span class="python-status-indicator-default" id="python-status-indicator"></span></a>
      </li>
    </ul>
    <div class="copyright">
      <p><small>
        <span class="pre">Copyright &copy;2001-2016.</span>
        & <span class="pre"><a href="/psf-landing/">Python Software Foundation</a></span>
        & <span class="pre"><a href="/about/legal/">Legal Statements</a></span>
      </small>
    </div>
  </div>
</div>
```

真正的代码只有两行，我们便完成了Python官网的抓取，输出了网页的源代码，得到了源代码之后呢？你想要的链接、图片地址、文本信息不就都可以提取出来了吗？

接下来我们看下它返回的到底是什么，利用 `type` 函数输出 `response` 的类型。

```
# coding=utf-8
import urllib.request

response = urllib.request.urlopen('https://www.python.org')
print(type(response))
```

输出结果如下：

```
<class 'http.client.HTTPResponse'>
```

通过输出结果可以发现它是一个 `HTTPResponse` 类型的对象，它主要包含的方法有 `read()`、`readinto()`、`getheader(name)`、`getheaders()`、`fileno()` 等函数

和 `msg`、`version`、`status`、`reason`、`debuglevel`、`closed` 等属性。得到这个对象之后，赋值为 `response`，然后就可以用 `response` 调用这些方法和属性，得到返回结果的一系列信息。

例如 `response.read()` 就可以得到返回的网页内容，`response.status` 就可以得到返回结果的状态码，如200代表请求成功，404代表网页未找到等。

下面再来一个实例感受一下：

```
# coding=utf-8
import urllib.request

response = urllib.request.urlopen('https://www.python.org')
print(response.status)
print(response.getheaders())
print(response.getheader('Server'))
```

运行结果如下：

```
200
[('Server', 'nginx'), ('Content-Type', 'text/html; charset=utf-8'), ('X-Frame-Options', 'SAMEORIGIN'), ('X-Clacks-Overhead', 'GNU Terry Pratchett'), ('Content-Length', '47397'), ('Accept-Ranges', 'bytes'), ('Date', 'Mon, 01 Aug 2016 09:57:31 GMT'), ('Via', '1.1 varnish'), ('Age', '2473'), ('Connection', 'close'), ('X-Served-By', 'cache-lcy1125-LCY'), ('X-Cache', 'HIT'), ('X-Cache-Hits', '23'), ('Vary', 'Cookie'), ('Public-Key-Pins', 'max-age=600; includeSubDomains; pin-sha256="WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18="; pin-sha256="5C8kvU039KouVr152D0eZSGf40njo4Khs8tmyTlV3nU="; pin-sha256="5C8kvU039KouVr152D0eZSGf40njo4Khs8tmyTlV3nU="; pin-sha256="lCppFqbkr1J3EcVFAkeip0+44VaoJUymbn0aEUk7tEU="; pin-sha256="TUDnr0MEoJ3of7+YliBMBVFB4/gJsv5z07IXD9+YoWI="; pin-sha256="x4QzPSC810K5/cmjb05Qm4k3Bw5zBn4lTd0/nEW/Td4=";'), ('Strict-Transport-Security', 'max-age=63072000; includeSubDomains')]
nginx
```

可见，三个输出分别输出了响应的状态码，响应的头信息，以及通过传递一个参数获取了 `Server` 的类型。

urllib.request.urlopen()详解

利用以上最基本的 `urlopen()` 方法，我们可以完成最基本的简单网页的 `GET` 请求抓取。

如果我们想给链接传递一些参数该怎么实现呢？我们首先看一下 `urlopen()` 函数的API。

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

可以发现除了第一个参数可以传递URL之外，我们还可以传递其它的内容，比如 `data`（附加参数），`timeout`（超时时间）等等。

data参数

`data` 参数是可选的，如果要添加 `data` ，它要是字节流编码格式的内容，即 `bytes` 类型，通过 `bytes()` 函数可以进行转化，另外如果你传递了这个 `data` 参数，它的请求方式就不再是 `GET` 方式请求，而是 `POST` 。

下面用一个实例来感受一下：

```
# coding=utf-8
import urllib.parse
import urllib.request

data = bytes(urllib.parse.urlencode({'word': 'hello'}), encoding='utf8')
response = urllib.request.urlopen('http://httpbin.org/post', data=data)
print(response.read())
```

在这里我们传递了一个参数 `word` ，值是 `hello` 。它需要被转码成 `bytes` （字节流）类型。其中转字节流采用了 `bytes()` 方法，第一个参数需要是 `str` （字符串）类型，需要用 `urllib.parse.urlencode()` 方法来将参数字典转化为字符串。第二个参数指定编码格式，在这里指定为 `utf8` 。

提交的网址是 `httpbin.org` ，它可以提供 `HTTP` 请求测试。`http://httpbin.org/post` 这个地址可以用来测试 `POST` 请求，它可以输出请求和响应信息，其中就包含我们传递的 `data` 参数。

运行结果如下：

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "word": "hello"
  },
  "headers": {
    "Accept-Encoding": "identity",
    "Content-Length": "10",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "Python-urllib/3.5"
  },
  "json": null,
  "origin": "123.124.23.253",
  "url": "http://httpbin.org/post"
}
```

我们传递的参数出现在了 `form` 中，这表明是模拟了表单提交的方式，以 `POST` 方式传输数据。

timeout参数

`timeout` 参数可以设置超时时间，单位为秒，意思就是如果请求超出了设置的这个时间还没有得到响应，就会抛出异常，如果不指定，就会使用全局默认时间。它支持 `HTTP` 、 `HTTPS` 、 `FTP` 请求。

下面来用一个实例感受一下：

```
# coding=utf-8
import urllib.request

response = urllib.request.urlopen('http://httpbin.org/get', time
out=1)
print(response.read())
```

运行结果如下：

```
During handling of the above exception, another exception occurred:
```

```
Traceback (most recent call last): File "/var/py/python/urllibtest.py", line 4, in <module> response = urllib.request.urlopen('http://httpbin.org/get', timeout=1)
...
urllib.error.URLError: <urlopen error timed out>
```

在这里我们设置了超时时间是1秒，程序1秒过后服务器依然没有响应，于是抛出了 `urllib.error.URLError` 异常，错误原因是 `timed out`。

因此我们可以通过设置这个超时时间来控制一个网页如果长时间未响应就跳过它的抓取，利用 `try,except` 语句就可以实现这样的操作。

```
# coding=utf-8
import socket
import urllib.request
import urllib.error

try:
    response = urllib.request.urlopen('http://httpbin.org/get',
    timeout=0.1)
except urllib.error.URLError as e:
    if isinstance(e.reason, socket.timeout):
        print('TIME OUT')
```

在这里我们请求了 `http://httpbin.org/get` 这个测试链接，设置了超时时间是0.1秒，然后捕获了 `urllib.error.URLError` 这个异常，然后判断异常原因是超时异常，就得出它确实是因为超时而报错，打印输出了 `TIME OUT`，当然你也可以在这里做其他的处理。

运行结果如下：

```
TIME OUT
```

常理来说，0.1秒内基本不可能得到服务器响应，因此输出了 `TIME OUT` 的提示。

这样，我们可以通过设置 `timeout` 这个参数来实现超时处理，有时还是很有用的。

其他参数

还有 `context` 参数，它必须是 `ssl.SSLContext` 类型，用来指定 `SSL` 设置。

`cafile` 和 `capath` 两个参数是指定CA证书和它的路径，这个在请求 `HTTPS` 链接时会有用。

`cadefault` 参数现在已经弃用了，默认为 `False`。

以上讲解了 `urlopen()` 方法的使用，通过这个最基本的函数可以完成简单的请求和网页抓取，如需详细了解，可以参见官方文档。

<https://docs.python.org/3/library/urllib.request.html>

urllib.request.Request的使用

由上我们知道利用 `urlopen()` 方法可以实现最基本的请求发起，但这几个简单的参数并不足以构建一个完整的请求，如果请求中需要加入 `headers` 等信息，我们就可以利用更强大的 `Request` 类来构建一个请求。

首先我们用一个实例来感受一下 `Request` 的用法：

```
# coding=utf-8
import urllib.request

request = urllib.request.Request('https://python.org')
response = urllib.request.urlopen(request)
print(response.read().decode('utf-8'))
```

可以发现，我们依然是用 `urlopen()` 方法来发送这个请求，只不过这次 `urlopen()` 方法的参数不再是一个URL，而是一个 `Request`，通过构造这个这个数据结构，一方面我们可以将请求独立成一个对象，另一方面可配置参数更加丰富和灵活。

下面我们看一下 `Request` 都可以通过怎样的参数来构造，它的构造方法如下。

```
class urllib.request.Request(url, data=None, headers={}, origin_req_host=None, unverifiable=False, method=None)
```

第一个参数是请求链接，这个是必传参数，其他的都是可选参数。

`data` 参数如果要传必须传 `bytes`（字节流）类型的，如果是一个字典，可以先用 `urllib.parse.urlencode()` 编码。

`headers` 参数是一个字典，你可以在构造 `Request` 时通过 `headers` 参数传递，也可以通过调用 `Request` 对象的 `add_header()` 方法来添加请求头。请求头最常用的用法就是通过修改 `User-Agent` 来伪装浏览器，默认的 `User-Agent` 是 `Python-urllib`，你可以通过修改它来伪装浏览器，比如要伪装火狐浏览器，你可以把它设置为 `Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11`

`origin_req_host` 指的是请求方的 `host` 名称或者 `IP` 地址。

`unverifiable` 指的是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`。

`method` 是一个字符串，它用来指示请求使用的方法，比如 `GET`，`POST`，`PUT` 等等。

下面我们传入多个参数构建一个 `Request` 来感受一下：

```
# coding=utf-8
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, metho
d='POST')
response = request.urlopen(req)
print(response.read().decode('utf-8'))
```

在这里我们通过四个参数构造了一个 `Request`，`url` 即请求链接，在 `headers` 中指定了 `User-Agent` 和 `Host`，传递的参数 `data` 用了 `urlencode()` 和 `bytes()` 方法来转成字节流，另外指定了请求方式为 `POST`。

运行结果如下：

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "name": "Germey"
  },
  "headers": {
    "Accept-Encoding": "identity",
    "Content-Length": "11",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "Mozilla/4.0 (compatible; MSIE 5.5; Windows NT
)"
  },
  "json": null,
  "origin": "219.224.169.11",
  "url": "http://httpbin.org/post"
}
```

通过观察结果可以发现，我们成功设置了 `data` ， `headers` 以及 `method` 。

另外 `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)')
```

如此一来，我们就可以更加方便地构造一个 `Request` ，实现请求的发送。

urllib.request高级特性

大家有没有发现，在上面的过程中，我们虽然可以构造 `Request` ，但是一些更高级的操作，比如 `Cookies` 处理，代理该怎样来设置？

接下来就需要更强大的工具 `Handler` 登场了。

简而言之你可以把它理解为各种处理器，有专门处理登录验证的，有处理 Cookies 的，有处理代理设置的，利用它们我们几乎可以做到任何 HTTP 请求中所有的事情。

首先介绍下 `urllib.request.BaseHandler`，它是所有其他 `Handler` 的父类，它提供了最基本的 `Handler` 的方法，例如 `default_open()`、`protocol_request()` 等。

接下来就有各种 `Handler` 类继承这个 `BaseHandler`，列举如下：

- `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 `Cookie`。
- `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。另外还有其他的 `Handler`，可以参考官方文档。

<https://docs.python.org/3/library/urllib.request.html#urllib.request.BaseHandler>

它们怎么来使用，不用着急，下面会有实例为你演示。

另外一个比较重要的就是 `OpenerDirector`，我们可以称之为 `Opener`，我们之前用过 `urllib.request.urlopen()` 这个方法，实际上它就是一个 `Opener`。

那么为什么要引入 `Opener` 呢？因为我们需要实现更高级的功能，之前我们使用的 `Request`、`urlopen()` 相当于类库为你封装好了极其常用的请求方法，利用它们两个我们就可以完成基本的请求，但是现在不一样了，我们需要实现更高级的功能，所以我们需要深入一层，使用更上层的实例来完成我们的操作。所以，在这里我们就用到了比调用 `urlopen()` 的对象的更普遍的对象，也就是 `Opener`。

`Opener` 可以使用 `open()` 方法，返回的类型和 `urlopen()` 如出一辙。那么它和 `Handler` 有什么关系？简而言之，就是利用 `Handler` 来构建 `Opener`。

认证

我们先用一个实例来感受一下：

```
# coding=utf-8
import urllib.request

auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

此处代码为实例代码，用于说明 `Handler` 和 `Opener` 的使用方法。

在这里，首先实例化了一个 `HTTPBasicAuthHandler` 对象，然后利用 `add_password()` 添加进去用户名和密码，相当于建立了一个处理认证的处理器。

接下来利用 `urllib.request.build_opener()` 方法来利用这个处理器构建一个 `Opener`，那么这个 `Opener` 在发送请求的时候就具备了认证功能了。接下来利用 `Opener` 的 `open()` 方法打开链接，就可以完成认证了。

代理

如果添加代理，可以这样做：

```
# coding=utf-8
import urllib.request

proxy_handler = urllib.request.ProxyHandler({
    'http': 'http://218.202.111.10:80',
    'https': 'https://180.250.163.34:8888'
})
opener = urllib.request.build_opener(proxy_handler)
response = opener.open('https://www.baidu.com')
print(response.read())
```

此处代码为实例代码，用于说明代理的设置方法，代理可能已经失效。

在这里使用了 `ProxyHandler`，`ProxyHandler` 的参数是一个字典，`key`是协议类型，比如 `http` 还是 `https` 等，`value`是代理链接，可以添加多个代理。

然后利用 `build_opener()` 方法利用这个 `Handler` 构造一个 `Opener`，然后发送请求即可。

Cookie设置

我们先用一个实例来感受一下怎样将网站的 `Cookie` 获取下来。

```
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')
for item in cookie:
    print(item.name+"="+item.value)
```

首先我们必须声明一个 `CookieJar` 对象，接下来我们就需要利用 `HTTPCookieProcessor` 来构建一个 `handler`，最后利用 `build_opener` 方法构建出 `opener`，执行 `open()` 即可。

运行结果如下：

```
BAIDUID=2E65A683F8A8BA3DF521469DF8EFF1E1:FG=1
BIDUPSID=2E65A683F8A8BA3DF521469DF8EFF1E1
H_PS_PSSID=20987_1421_18282_17949_21122_17001_21227_21189_21161_
20927
PSTM=1474900615
BDSVRTM=0
BD_HOME=0
```

可以看到输出了每一条 `Cookie` 的名称还有值。

不过既然能输出，那可不可以输出成文件格式呢？我们知道很多 `Cookie` 实际也是以文本形式保存的。

答案当然是肯定的，我们用下面的实例来感受一下：

```
filename = 'cookie.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)
```

这时的 `CookieJar` 就需要换成 `MozillaCookieJar`，生成文件时需要用到它，它是 `CookieJar` 的子类，可以用来处理 `Cookie` 和文件相关的事件，读取和保存 `Cookie`，它可以将 `Cookie` 保存成 `Mozilla` 型的格式。

运行之后可以发现生成了一个 `cookie.txt` 文件。

内容如下：

```
# Netscape HTTP Cookie File
# http://curl.haxx.se/rfc/cookie_spec.html
# This is a generated file! Do not edit.

.baidu.com    TRUE    /    FALSE    3622386254    BAIDUID    05A
E39B5F56C1DEC474325CDA522D44F:FG=1
.baidu.com    TRUE    /    FALSE    3622386254    BIDUPSID    05
AE39B5F56C1DEC474325CDA522D44F
.baidu.com    TRUE    /    FALSE    H_PS_PSSID    19638_1453
_17710_18240_21091_18560_17001_21191_21161
.baidu.com    TRUE    /    FALSE    3622386254    PSTM    147490
2606
www.baidu.com    FALSE    /    FALSE    BDSVRTM    0
www.baidu.com    FALSE    /    FALSE    BD_HOME    0
```

另外还有一个 `LWPCookieJar`，同样可以读取和保存 `Cookie`，但是保存的格式和 `MozillaCookieJar` 的不一样，它会保存成与 `libwww-perl` 的 `Set-Cookie3` 文件格式的 `Cookie`。

那么在声明时就改为

```
cookie = http.cookiejar.LWPCookieJar(filename)
```


生成的内容如下：

```
#LWP-Cookies-2.0
Set-Cookie3: BAIDUID="0CE9C56F598E69DB375B7C294AE5C591:FG=1"; path="/"; domain=".baidu.com"; path_spec; domain_dot; expires="2084-10-14 18:25:19Z"; version=0
Set-Cookie3: BIDUPSID=0CE9C56F598E69DB375B7C294AE5C591; path="/"; domain=".baidu.com"; path_spec; domain_dot; expires="2084-10-14 18:25:19Z"; version=0
Set-Cookie3: H_PS_PSSID=20048_1448_18240_17944_21089_21192_21161_20929; path="/"; domain=".baidu.com"; path_spec; domain_dot; discard; version=0
Set-Cookie3: PSTM=1474902671; path="/"; domain=".baidu.com"; path_spec; domain_dot; expires="2084-10-14 18:25:19Z"; version=0
Set-Cookie3: BDSVRTM=0; path="/"; domain="www.baidu.com"; path_spec; discard; version=0
Set-Cookie3: BD_HOME=0; path="/"; domain="www.baidu.com"; path_spec; discard; version=0
```

由此看来生成的格式还是有比较大的差异的。

那么生成了 `Cookie` 文件，怎样从文件读取并利用呢？

下面我们以 `LWPCookieJar` 格式为例来感受一下：

```
cookie = http.cookiejar.LWPCookieJar()
cookie.load('cookie.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'))
```

前提是我们首先利用上面的方式生成了 `LWPCookieJar` 格式的 `Cookie`，然后利用 `load()` 方法，传入文件名称，后面同样的方法构建 `handler` 和 `opener` 即可。

运行结果正常输出百度网页的源代码。

好，通过如上用法，我们可以实现绝大多数请求功能的设置了。

发出请求之后，如果遇到异常怎么办？在下一节，我们会讲解一下有关异常处理的流程。

使用urllib.error处理异常

`urllib.error` 模块定义了由 `urllib.request` 产生的异常。如果出现了问题，`urllib.request` 便会抛出 `urllib.error`，下面会对其进行详细的介绍。

urllib.error.URLError

它继承自 `OSError`，是 `urllib.error` 异常类的基类，由 `urllib.request` 产生的异常都可以通过捕获这个类来处理。

它具有一个属性 `reason`，即返回错误的原因。

下面用一个实例来感受一下：

```
# coding=utf-8
from urllib import request,error
try:
    response = request.urlopen('http://cuiqingcai.com/index.htm')
except error.URLError as e:
    print(e.reason)
```

我们打开一个不存在的页面，然后捕获了 `URLError` 这个异常。

运行结果：

```
Not Found
```

这样通过如上操作，避免了程序异常终止，同时异常得到了有效处理。

urllib.error.HTTPError

它是刚才介绍的 `URLError` 的子类，专门用来处理 `HTTP` 请求错误，比如认证请求失败等等。

它有三个属性。

`code`，返回 `HTTP` 状态码，比如404网页不存在，500服务器内部错误等等。

`reason`，同父类一样，返回错误的原因。

`headers`，返回 HTTP 响应头。

下面用表格列出了常见的错误代码及错误原因。

状态码	说明	详情
100	继续	请求者应当继续提出请求。服务器已收到请求的一部分，正在等待其余部分。
101	切换协议	请求者已要求服务器切换协议，服务器已确认并准备切换。
200	成功	服务器已成功处理了请求。
201	已创建	请求成功并且服务器创建了新的资源。
202	已接受	服务器已接受请求，但尚未处理。
203	非授权信息	服务器已成功处理了请求，但返回的信息可能来自另一来源。
204	无内容	服务器成功处理了请求，但没有返回任何内容。
205	重置内容	服务器成功处理了请求，内容被重置。
206	部分内容	服务器成功处理了部分请求。
300	多种选择	针对请求，服务器可执行多种操作。
301	永久移动	请求的网页已永久移动到新位置，即永久重定向。
302	临时移动	请求的网页暂时跳转到其他页面，即暂时重定向。
303	查看其他位置	如果原来的请求是POST，重定向目标文档应该通过GET提取。
304	未修改	此次请求返回的网页未修改，继续使用上次的资源。
305	使用代理	请求者应该使用代理访问该网页。
307	临时重定向	请求的资源临时从其他位置响应。
400	错误请求	服务器无法解析该请求。
401	未授权	请求没有进行身份验证或验证未通过。
403	禁止访问	服务器拒绝此请求。
404	未找到	服务器找不到请求的网页。
405	方法禁用	服务器禁用了请求中指定的方法。

406	不接受	无法使用请求的内容响应请求的网页。
407	需要代理授权	请求者需要使用代理授权。
408	请求超时	服务器请求超时。
409	冲突	服务器在完成请求时发生冲突。
410	已删除	请求的资源已永久删除。
411	需要有效长度	服务器不接受不含有效内容长度标头字段的请求。
412	未满足前提条件	服务器未满足请求者在请求中设置的其中一个前提条件。
413	请求实体过大	请求实体过大，超出服务器的处理能力。
414	请求URI过长	请求网址过长，服务器无法处理。
415	不支持类型	请求的格式不受请求页面的支持。
416	请求范围不符	页面无法提供请求的范围。
417	未满足期望值	服务器未满足期望请求标头字段的要求。
500	服务器内部错误	服务器遇到错误，无法完成请求。
501	未实现	服务器不具备完成请求的功能。
502	错误网关	服务器作为网关或代理，从上游服务器收到无效响应。
503	服务不可用	服务器目前无法使用。
504	网关超时	服务器作为网关或代理，但是没有及时从上游服务器收到请求。
505	HTTP版本不支持	服务器不支持请求中所用的 HTTP 协议版本。

下面我们来用几个实例感受一下：

```
# coding=utf-8
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, seq='\n')
```

运行结果：

```
Not Found
404
Server: nginx/1.4.6 (Ubuntu)
Date: Wed, 03 Aug 2016 08:54:22 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: close
X-Powered-By: PHP/5.5.9-1ubuntu4.14
Vary: Cookie
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0
Pragma: no-cache
Link: <http://cuiqingcai.com/wp-json/>; rel="https://api.w.org/"
```

依然是同样的网址，在这里我们捕获了 `HTTPError`，输出了错误原因、错误代号、服务器响应头。

因为 `URLError` 是 `HTTPError` 的父类，所以我们可以先选择捕获子类的错误，再去捕获父类的错误，更好的写法如下：

```
# coding=utf-8
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')
except error.URLError as e:
    print(e.reason)
else:
    print('Request Successfully')
```

这样我们就可以做到先捕获 `HTTPError`，获取它的错误码，错误原因，服务器响应头等详细信息。如果非 `HTTPError`，再捕获 `URLError` 错误，输出错误原因。最后用 `else` 来处理正常的逻辑。

在有时候 `e.reason` 返回的不一定是字符串，可能是一个对象。

下面用一个实例来感受一下：

```
# coding=utf-8
import socket
import urllib.request
import urllib.error

try:
    response = urllib.request.urlopen('https://www.baidu.com', t
imeout=0.01)
except urllib.error.URLError as e:
    print(type(e.reason))
    if isinstance(e.reason, socket.timeout):
        print('TIME OUT')
```

在这里我们直接设置了超时时间来强制抛出 `timeout` 异常。

运行结果如下：

```
<class 'socket.timeout'>  
TIME OUT
```

可以发现 `e, reason` 的类型是 `socket.timeout`。所以我们可以用 `isinstance()` 方法来判断它的类型，做出更详细的异常判断。

本节讲述了 `urllib.error` 的相关处理，通过合理地捕获异常可以做出更准确的异常判断，使得程序更佳稳健。

使用urllib.parse解析链接

这个模块定义了处理 URL 的标准接口，例如实现 URL 各部分的抽取，合并以及链接转换。它支持如下类型的链接处

理：file 、 ftp 、 gopher 、 hdl 、 http 、 https 、 imap 、 mailto 、
mms 、 news 、 nntp 、 prospero 、 rsync 、 rtsp 、 rtspu 、 sftp 、
shttp 、
sip 、 sips 、 snews 、 svn 、 svn+ssh 、 telnet 、 wais 。

urllib.parse.urlparse()

常用的函数有 urllib.parse.urlparse() 。

先用一个实例来感受一下：

```
# coding=utf-8
from urllib.parse import urlparse

result = urlparse('http://www.baidu.com/index.html;user?id=5#comment')
print(type(result), result)
```

在这里我们首先输出了结果的类型，然后将结果也输出出来。

运行结果：

```
<class 'urllib.parse.ParseResult'> ParseResult(scheme='http', netloc='www.baidu.com', path='/index.html', params='user', query='id=5', fragment='comment')
```

观察可以看到，返回结果是一个 ParseResult 类型的对象，它包含了六个部分，分别是 scheme 、 netloc 、 path 、 params 、 query 、 fragment 。

观察一下实例的网址。

```
http://www.baidu.com/index.html;user?id=5#comment
```

`urlparse()` 方法将其拆分成了六部分，大体观察可以发现，解析时有特定的分隔符，比如 `://` 前面的就是 `scheme`，第一个 `/` 前面便是 `netloc`，分号 `;` 前面是 `params` 等等。

所以可以得出一个标准的链接格式如下：

```
scheme://netloc/path;parameters?query#fragment
```

除了这种最基本的解析方式，`urlopen()` 方法还有其他配置吗？接下来看一下它的 API。

```
urllib.parse.urlparse(urlstring, scheme='', allow_fragments=True)
```

它有三个参数：

第一个参数 `urlstring` 是必填项，即待解析的 `URL`。

第二个参数 `scheme` 是默认的协议（比如 `http`、`https` 等），假如这个链接没有带协议信息，会将这个作为默认的协议。

我们用一个实例感受一下：

```
# coding=utf-8
from urllib.parse import urlparse

result = urlparse('www.baidu.com/index.html;user?id=5#comment',
                  scheme='https')
print(result)
```

运行结果：

```
ParseResult(scheme='https', netloc='', path='www.baidu.com/index.html', params='user', query='id=5', fragment='comment')
```

可以发现，我们提供的链接没有包含最前面的 `scheme`，但是通过指定默认的 `scheme` 参数，返回的结果带有 `https`。

假设我们带上了 `scheme` 呢？

```
result = urlparse('http://www.baidu.com/index.html;user?id=5#comment', scheme='https')
```

结果如下：

```
ParseResult(scheme='http', netloc='www.baidu.com', path='/index.html', params='user', query='id=5', fragment='comment')
```

可见 `scheme` 参数只有在链接中不包含 `scheme` 信息时才会生效，如果链接中有了 `scheme`，那就返回解析出的 `scheme`。

第三个参数 `allow_fragments` 是是否忽略 `fragment`，如果它被设置为 `False`，`fragment` 部分就会被忽略，它会被解析为 `path`、`parameters` 或者 `query` 的一部分，`fragment` 部分为空。

下面我们用一个实例感受一下：

```
# coding=utf-8
from urllib.parse import urlparse

result = urlparse('http://www.baidu.com/index.html;user?id=5#comment', allow_fragments=False)
print(result)
```

运行结果：

```
ParseResult(scheme='http', netloc='www.baidu.com', path='/index.html', params='user', query='id=5#comment', fragment='')
```

假设链接中不包含 `parameters` 和 `query` 呢？

再来一个实例：

```
# coding=utf-8
from urllib.parse import urlparse

result = urlparse('http://www.baidu.com/index.html#comment', allow_fragments=False)
print(result)
```

运行结果：

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

可以发现当链接中不包含 `params` 和 `query` 时，`fragment` 便会被解析为 `path` 的一部分。

返回结果 `ParseResult` 实际上是一个元组，你可以用索引顺序来获取，也可以用属性名称获取。

```
# coding=utf-8
from urllib.parse import urlparse

result = urlparse('http://www.baidu.com/index.html#comment', allow_fragments=False)
print(result.scheme, result[0], result.netloc, result[1], sep='\n')
```

在这里我们分别用索引和属性名获取了 `scheme` 和 `netloc`，运行结果如下：

```
http
http
www.baidu.com
www.baidu.com
```

可以发现二者结果是一致的，两种方法都可以成功获取。

urllib.parse.urlunparse()

有了 `urlparse()` 那相应地就有了它的对立方法 `urlunparse()`。

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。

先用一个实例感受一下：

```
# coding=utf-8
from urllib.parse import urlunparse

data = ['http', 'www.baidu.com', 'index.html', 'user', 'a=6', 'comment']
print(urlunparse(data))
```

参数用了 `list` 类型，当然你也可以用其他的类型如 `tuple` 或者特定的数据结构。

运行结果如下：

```
http://www.baidu.com/index.html;user?a=6#comment
```

urllib.parse.urlsplit()

这个和 `urlparse()` 方法非常相似，只不过它不会单独解析 `parameters` 这一部分，只返回五个结果。上面例子中的 `parameters` 会合并到 `path` 中。

用一个实例感受一下：

```
# coding=utf-8
from urllib.parse import urlsplit

result = urlsplit('http://www.baidu.com/index.html;user?id=5#comment')
print(result)
```

运行结果：

```
SplitResult(scheme='http', netloc='www.baidu.com', path='/index.html;user', query='id=5', fragment='comment')
```

返回结果是 `SplitResult`，其实也是一个元组类型，可以用属性获取值也可以用索引来获取。

```
# coding=utf-8
from urllib.parse import urlsplit

result = urlsplit('http://www.baidu.com/index.html;user?id=5#comment')
print(result.scheme, result[0])
```

运行结果：

```
http http
```

urllib.parse.urlunsplit()

与 `urlunparse()` 类似，也是将链接的各个部分组合成完整链接的方法，传入的也是一个可迭代对象。例如 `list`、`tuple` 等等，唯一的区别是，长度必须为 5。

用一个实例来感受一下：

```
# coding=utf-8
from urllib.parse import urlunsplit

data = ['http', 'www.baidu.com', 'index.html', 'a=6', 'comment']
print(urlunsplit(data))
```

运行结果：

```
http://www.baidu.com/index.html?a=6#comment
```

同样可以完成链接的拼接生成。

urllib.parse.urljoin()

有了 `urlunparse()` 和 `urlunsplit()` 方法，我们可以完成链接的合并，不过前提必须要有特定长度的对象，链接的每一部分都要清晰分开。

生成链接还有另一个方法，利用 `urljoin()` 方法我们可以提供一个 `base_url`（基础链接），新的链接作为第二个参数，方法会分析 `base_url` 的 `scheme`、`netloc`、`path` 这三个内容对新链接缺失的部分进行补充，作为结果返回。

空说无益，我们用几个实例来感受一下：

```
# coding=utf-8
from urllib.parse import urljoin

print(urljoin('http://www.baidu.com', 'FAQ.html'))
print(urljoin('http://www.baidu.com', 'https://cuiqingcai.com/FAQ.html'))
print(urljoin('http://www.baidu.com/about.html', 'https://cuiqingcai.com/FAQ.html'))
print(urljoin('http://www.baidu.com/about.html', 'https://cuiqingcai.com/FAQ.html?question=2'))
print(urljoin('http://www.baidu.com?wd=abc', 'https://cuiqingcai.com/index.php'))
print(urljoin('http://www.baidu.com', '?category=2#comment'))
print(urljoin('www.baidu.com', '?category=2#comment'))
print(urljoin('www.baidu.com#comment', '?category=2'))
```

运行结果：

```
http://www.baidu.com/FAQ.html
https://cuiqingcai.com/FAQ.html
https://cuiqingcai.com/FAQ.html
https://cuiqingcai.com/FAQ.html?question=2
https://cuiqingcai.com/index.php
http://www.baidu.com?category=2#comment
www.baidu.com?category=2#comment
www.baidu.com?category=2
```

可以发现，`base_url` 提供了三项内容，`scheme`、`netloc`、`path`，如果这三项在新的链接里面不存在，那么就予以补充，如果新的链接存在，那么就使用新的链接的部分。`base_url` 中的 `parameters`、`query`、`fragments` 是不起作用的。

好，通过如上的函数，我们可以轻松地实现链接的解析，拼合与生成。

使用urllib.robotparser分析robots协议

什么是robots协议

要了解 `urllib.robotparser` 这个库，首先我们需要了解下什么是 `robots` 协议。

`robots` 协议 也被称作爬虫协议、机器人协议，它的全名叫做 网络爬虫排除标准 (`Robots Exclusion Protocol`)，同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。它通常是一个叫做 `robots.txt` 的文本文件，放在网站的根目录下。

当一个搜索蜘蛛访问一个站点时，它首先会检查下这个站点根目录下是否存在 `robots.txt` 文件，如果存在，搜索蜘蛛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，那么搜索蜘蛛便会访问所有可直接访问的页面。

下面我们用一个实例感受一下：

```
User-agent: *  
Disallow: /  
Allow: /public/
```

以上的两行实现了对所有搜索蜘蛛只允许爬取 `public` 目录的作用。

如上简单的两行，保存成 `robots.txt` 文件，放在网站的根目录下，和网站的入口文件放在一起。比如 `index.php`、`index.html`、`index.jsp` 等等。

那么上面的 `User-agent` 就描述了搜索蜘蛛的名称，在这里将值设置为 `*`，则代表该协议对任何的爬取蜘蛛有效。比如你可以设置 `User-agent: Baiduspider`，就代表我们设置的规则对百度搜索引擎是有效的。如果有多条 `User-agent` 记录，则就会有多个爬取蜘蛛会受到爬取限制，但你至少需要指定一条。

`Disallow` 指定了不允许抓取的目录，比如上述例子中设置为 `/` 则代表不允许抓取所有页面。

`Allow` 它一般和 `Disallow` 一起使用，一般不会单独使用，现在我们设置为 `/public/`，起到的作用是所有页面不允许抓取，但是 `public` 目录是可以抓取的。

下面我们再来看几个例子感受一下：

禁止所有搜索蜘蛛访问网站任何部分

```
User-agent: *  
Disallow: /
```

允许所有搜索蜘蛛访问

```
User-agent: *  
Disallow:
```

或者直接把 robots.txt 文件留空。

禁止所有搜索蜘蛛访问网站某些目录

```
User-agent: *  
Disallow: /private/  
Disallow: /tmp/
```

只允许某一个搜索蜘蛛访问

```
User-agent: WebCrawler  
Disallow:  
User-agent: *  
Disallow: /
```

大家可能会疑惑，蜘蛛名是哪儿来的？为什么就叫这个名？其实它是有固定名字的了，比如百度的就叫做BaiduSpider，下面的表格列出了一些常见的搜索蜘蛛的名称及对应的网站。

蜘蛛名称	名称	网站
BaiduSpider	百度	www.baidu.com
Googlebot	谷歌	www.google.com
360Spider	360搜索	www.so.com
YodaoBot	有道	www.youdao.com
ia_archiver	Alexa	www.alexa.cn
Scooter	altavista	www.altavista.com

使用urllib.robotparser

了解了什么是 robots协议 之后，我们就可以使用 urllib.robotparser 来解析 robots.txt 了。

urllib.parser 提供了一个类，叫做 RobotFileParser 。它可以根据某网站的 robots.txt 文件来判断一个爬取蜘蛛是否有权限来爬取这个网页。

使用非常简单，首先看一下它的声明

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

使用这个类的时候非常简单，只需要在构造方法里传入 robots.txt 的链接即可。当然也可以声明时不传入，默认为空，再使用 set_url(url) 方法设置一下也可以。

有常用的几个方法分别介绍一下：

set_url(url) 用来设置 robots.txt 文件的链接。如果已经在创建 RobotFileParser 对象时传入了链接，那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，注意这个函数是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(lines) 方法是用来解析 robots.txt 文件的，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(useragent, url)` 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，这个对于长时间分析和抓取的搜索蜘蛛是很有必要的，你可能需要定期检查来抓取最新的 `robots.txt`。

`modified()` 同样的对于长时间分析和抓取的搜索蜘蛛很有帮助，将当前时间设置为上次抓取和分析 `robots.txt` 的时间。

以上是这个类提供的所有方法，下面我们用实例来感受一下：

```
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"))
```

以简书为例，我们首先创建 `RobotFileParser` 对象，然后通过 `set_url()` 方法来设置了 `robots.txt` 的链接。当然不用这个方法的话，可以在声明时直接用 `rp = RobotFileParser('http://www.jianshu.com/robots.txt')` 声明对象，下一步关键的，执行读取和分析。然后后面利用了 `can_fetch()` 方法来判断了网页是否可以被抓取。

运行结果：

```
True
False
```

同样也可以使用 `parser()` 方法执行读取和分析。

用一个实例感受一下：

```
from urllib.robotparser import RobotFileParser
from urllib.request import urlopen

rp = RobotFileParser()
rp.parse(urlopen('http://www.jianshu.com/robots.txt').read().decode('utf-8').split('\n'))
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"))
```

运行结果一样：

```
True
False
```

以上介绍了 `urllib.robotparser` 的基本用法和实例讲解，利用它我们就可以方便地判断哪些页面可以抓取哪些不可以了。

使用requests

在使用了 `urllib` 之后，我们发现其中确实有不便捷的地方。比如处理网页验证、处理Cookies等等。那么在这里就有了更为强大的库 `requests`，有了它，Cookies，登录验证，代理设置等等的操作都不是事儿。

那么接下来就让我们来领略一下它的强大之处吧。

安装requests

安装流程

由于 `requests` 属于第三方库，也就是 `python` 默认不会自带这个库，需要我们手动去安装，下面我们首先看一下它的安装过程。

pip安装

无论是Windows、Linux还是Mac，你都可以通过pip这个包管理工具来安装。

在命令行下运行如下命令：

```
pip3 install requests
```

这是最简单的安装方式，推荐此种方法。

源码安装

那么如果你没有pip或者不想用pip来安装，或者想获取某一特定版本，可以选择下载源码安装。

项目的地址是：

<https://github.com/kennethreitz/requests>

你可以通过下面的方式来下载源代码：

```
git clone git://github.com/kennethreitz/requests.git
```

或

```
curl -OL https://github.com/kennethreitz/requests/tarball/master
```

或者直接点击下载压缩包。

下载下来之后，进入目录，执行安装即可。

```
cd requests
python3 setup.py install
```

即可完成 `requests` 的安装。

验证安装

为了证明你已经安装成功，可以在命令行下测试一下。

```
→ / python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
```

在命令行首先输入 `python3`，进入命令行模式，然后输入 `import requests`，如果什么错误提示也没有，那么就证明你已经成功安装了 `requests`。

好，安装成功之后，接下来在下一节就让我们一起领略一下它的风采吧。

requests的基本使用

实例引入

在 `urllib` 库中，有 `urllib.request.urlopen(url)` 的方法，实际上它是以 `GET` 方式请求了一个网页。

那么在 `requests` 中，相应的方法就是 `requests.get(url)`，是不是感觉表达更明确一些？

下面我们用一个实例来感受一下：

```
# coding=utf-8
import requests

r = requests.get('https://www.baidu.com/')
print(type(r))
print(r.status_code)
print(type(r.text))
print(r.text)
print(r.cookies)
```

运行结果如下：

```
<class 'requests.models.Response'>
200
<class 'str'>
<html>
<head>
    <script>
        location.replace(location.href.replace("https://", "http:
//"));
    </script>
</head>
<body>
    <noscript><meta http-equiv="refresh" content="0;url=http://w
ww.baidu.com/"></noscript>
</body>
</html>
<RequestsCookieJar[<Cookie BIDUPSID=992C3B26F4C4D09505C5E959D5FB
C005 for .baidu.com/>, <Cookie PSTM=1472227535 for .baidu.com/>,
    <Cookie __bsi=15304754498609545148_00_40_N_N_2_0303_C02F_N_N_N_
0 for .www.baidu.com/>, <Cookie BD_NOT_HTTPS=1 for www.baidu.com
/>]>
```

上面的例子分别输出了请求响应的类型，状态码，响应体内容的类型，响应体内容还有Cookies。

通过上述实例可以发现，它的返回类型是 `requests.models.Response`，响应体的类型是字符串 `str`，Cookies 的类型是 `RequestsCookieJar`。

我们可以发现，使用了 `requests.get(url)` 方法就成功实现了一个 GET 请求。这倒不算什么，更方便的在于其他的请求类型依然可以用一句话来完成。

用一个实例来感受一下：

```
r = requests.post('http://httpbin.org/post')
r = requests.put('http://httpbin.org/put')
r = requests.delete('http://httpbin.org/delete')
r = requests.head('http://httpbin.org/get')
r = requests.options('http://httpbin.org/get')
```

怎么样？是不是比 `urllib` 太多了？

其实这只是冰山一角，更多的还在后面呢。

GET请求

HTTP中最常见的请求之一就是 GET 请求，我们首先来详细了解下利用 `requests` 来构建 GET 请求的方法以及相关属性方法操作。

首先让我们来构建一个最简单的 GET 请求，请求 `httpbin.org/get`，它会判断如果你是 GET 请求的话，会返回响应的请求信息。

```
# coding=utf-8
import requests

r = requests.get('http://httpbin.org/get')
print(r.text)
```

运行结果如下：

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.10.0"
  },
  "origin": "122.4.215.33",
  "url": "http://httpbin.org/get"
}
```

可以发现我们成功发起了get请求，请求的链接和头信息都有相应的返回。

那么 GET 请求，如果要附加额外的信息一般是怎样来添加？没错，那就是直接当做参数添加到 `url` 后面。

比如现在我想添加两个参数，名字`name`是`germey`，年龄`age`是`22`。构造这个请求链接是不是我们要直接写成 `r = requests.get("http://httpbin.org/get?name=germey&age=22")` ？

可以是，但是不觉得很很不人性化吗？一般的这种信息数据我们会用字典 `{"name": "germey", "age": 22}` 来存储，那么怎样来构造这个链接呢？

同样很简单，利用 `params` 这个参数就好了。

实例如下：

```
# coding=utf-8
import requests

data = {
    'name': 'germey',
    'age': 22
}
r = requests.get("http://httpbin.org/get", params=data)
print(r.text)
```

运行结果如下：

```
{
  "args": {
    "age": "22",
    "name": "germey"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.10.0"
  },
  "origin": "122.4.215.33",
  "url": "http://httpbin.org/get?age=22&name=germey"
}
```

通过返回信息我们可以判断，请求的链接自动被构造成

了 `http://httpbin.org/get?age=22&name=germey`，是不是很方便？

另外，网页的返回类型实际上是 `str` 类型，但是它很特殊，是 `Json` 的格式，所以如果我们想直接把返回结果解析，得到一个字典 `dict` 格式的话，可以直接调用 `json()` 方法。

用一个实例来感受一下：

```
# coding=utf-8
import requests

r = requests.get("http://httpbin.org/get")
print(type(r.text))
print(r.json())
print(type(r.json()))
```

运行结果如下：

```
<class 'str'>
{'headers': {'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*',
, 'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.10.0'}
, 'url': 'http://httpbin.org/get', 'args': {}, 'origin': '182.33
.248.131'}
<class 'dict'>
```

可以发现，调用 `json()` 方法，就可以将返回结果是 `Json` 格式的字符串转化为字典 `dict` 。

但注意，如果返回结果不是 `Json` 格式，便会出现解析错误，抛出 `json.decoder.JSONDecodeError` 的异常。

抓取网页

如上的请求链接返回的是 `Json` 形式的字符串，那么如果我们请求普通的网页，那么肯定就能获得相应的内容了。

下面我们以知乎—发现页面为例来体验一下：

```
# coding=utf-8
import requests
import re

headers = {
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36'
}
r = requests.get("https://www.zhihu.com/explore", headers=headers)
pattern = re.compile('explore-feed.*?question_link.*?>(.*?)</a>', re.S)
titles = re.findall(pattern, r.text)
print(titles)
```

如上代码，我们请求了知乎一发现页面 `https://www.zhihu.com/explore`，在这里加入了头信息，头信息中包含了 `User-Agent` 信息，也就是浏览器标识信息。如果不加这个，知乎会禁止抓取。

在接下来用到了最基础的正则表达式，来匹配出所有的问题内容，关于正则表达式会在后面的章节中详细介绍，在这里作为用到实例来配合讲解。

运行结果如下：

```
['\n为什么很多人喜欢提及「拉丁语系」这个词？\n', '\n在没有水的情况下水系宝可梦如何战斗？\n', '\n有哪些经验可以送给 Kindle 新人？\n', '\n谷歌的广告业务是如何赚钱的？\n', '\n程序员该学习什么，能在上学期间挣钱？\n', '\n有哪些原本只是一个小消息，但回看发现是个惊天大新闻的例子？\n', '\n如何评价今敏？\n', '\n源氏是怎么把那么长的刀从背后拔出来的？\n', '\n年轻时得了绝症或大病是怎样的感受？\n', '\n年轻时得了绝症或大病是怎样的感受？\n']
```

发现成功提取出了所有的问题内容。

没错，提取信息就是这么方便。

抓取二进制数据

在上面的例子中，我们抓取的是知乎的一个页面，实际上它返回的是一个 `HTML` 文档，那么如果我们想抓去图片、音频、视频等文件的话应该怎么办呢？

我们都知道，图片、音频、视频这些文件都是本质上由二进制码组成的，由于有特定的保存格式和对应的解析方式，我们才可以看到这些形形色色的多媒体。所以想要抓取他们，那就需要拿到他们的二进制码。

下面我们以GitHub的站点图标为例来感受一下：

```
# coding=utf-8
import requests

r = requests.get("https://github.com/favicon.ico")
print(r.text)
print(r.content)
```

抓取的内容是站点徽标，也就是在浏览器每一个标签上显示的小图标。



在这里打印了 `response` 的两个属性，一个是 `text`，另一个是 `content`。

运行结果如下，由于包含特殊内容，在此放运行结果的图片：



那么前两行便是 `r.text` 的结果，最后一行是 `r.content` 的结果。

可以注意到，前者出现了乱码，后者结果前面带有一个 `b`，代表这是 `bytes` 类型的数据。由于图片是二进制数据，所以前者在打印时转化为 `str` 类型，也就是图片直接转化为字符串，理所当然会出现乱码。

两个属性有什么区别？前者返回的是字符串类型，如果返回结果是文本文件，那么用这种方式直接获取其内容即可。如果返回结果是图片、音频、视频等文件，`requests` 会为我们自动解码成 `bytes` 类型，即获取字节流数据。

进一步地，我们可以将刚才提取到的图片保存下来。

```
# coding=utf-8
import requests

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

在这里用了 `open()` 函数，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，然后保存。

运行结束之后，可以发现在文件夹中出现了名为 `favicon.ico` 的图标。



同样的，音频、视频文件也可以用这种方法获取。

添加头信息

如 `urllib.request` 一样，我们也可以通过 `headers` 参数来传递头信息。

比如上面的知乎的例子，如果不传递头信息，就不能正常请求：

```
# coding=utf-8
import requests

r = requests.get("https://www.zhihu.com/explore")
print(r.text)
```

运行结果如下：

```
<html><body><h1>500 Server Error</h1>
An internal server error occurred.
</body></html>
```

但如果加上请求头信息，那就没问题了：


```
# coding=utf-8
import requests

headers = {
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36'
}
r = requests.get("https://www.zhihu.com/explore", headers=headers)
print(r.text)
```

当然你可以在 `headers` 这个数组中任意添加其他的头信息。

基本POST请求

在前面我们讲解了最基本的 `GET` 请求，另外一种比较常见的请求方式就是 `POST` 了，就像模拟表单提交一样，将一些数据提交到某个链接。

使用 `requests` 实现 `POST` 请求同样非常简单。

我们先用一个实例来感受一下：

```
# coding=utf-8
import requests

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

上面的例子请求的是 `httpbin.org/post`，它可以判断如果请求是 `POST` 方式，就把相关请求信息输出出来。

运行结果如下：

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "age": "22",
    "name": "germey"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "18",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.10.0"
  },
  "json": null,
  "origin": "182.33.248.131",
  "url": "http://httpbin.org/post"
}
```

可以发现，成功获得了返回结果，返回结果中的 `form` 部分就是提交的数据，那么这就证明 `POST` 请求成功发送了。

响应

发送请求之后，得到的自然就是响应，在上面的实例中我们使用了 `text` 和 `content` 获取了响应内容。不过还有很多属性和方法可以获取其他的信息。

比如响应状态码、响应头、Cookies。

下面用一个实例来感受一下：

```
# coding=utf-8
import requests

r = requests.get('http://www.jianshu.com')
print(type(r.status_code), r.status_code)
print(type(r.headers), r.headers)
print(type(r.cookies), r.cookies)
print(type(r.url), r.url)
print(type(r.history), r.history)
```

在这里分别打印输出了响应状态码 `status_code`，响应头 `headers`，Cookies，请求连接，请求历史的类型和内容。

运行结果如下：

```
<class 'int'> 200
<class 'requests.structures.CaseInsensitiveDict'> {'X-Runtime':
'0.006363', 'Connection': 'keep-alive', 'Content-Type': 'text/html; charset=utf-8', 'X-Content-Type-Options': 'nosniff', 'Date':
'Sat, 27 Aug 2016 17:18:51 GMT', 'Server': 'nginx', 'X-Frame-Options': 'DENY', 'Content-Encoding': 'gzip', 'Vary': 'Accept-Encoding', 'ETag': 'W/"3abda885e0e123bfde06d9b61e696159"', 'X-XSS-Protection': '1; mode=block', 'X-Request-Id': 'a8a3c4d5-f660-422f-8df9-49719dd9b5d4', 'Transfer-Encoding': 'chunked', 'Set-Cookie': 'read_mode=day; path=/, default_font=font2; path=/, _session_id=xxx; path=/; HttpOnly', 'Cache-Control': 'max-age=0, private, must-revalidate'}
<class 'requests.cookies.RequestsCookieJar'> <RequestsCookieJar[
<Cookie _session_id=xxx for www.jianshu.com/>, <Cookie default_font=font2 for www.jianshu.com/>, <Cookie read_mode=day for www.jianshu.com/>]>
<class 'str'> http://www.jianshu.com/
<class 'list'> []
```

`session_id` 过长在此简写。可以看到，`headers` 还有 `cookies` 这两个部分都是特定的数据结构，打开浏览器同样可以发现同样的响应头信息。

状态码

在这里状态码常用来判断请求是否成功，`requests` 还提供了一个内置的状态码查询对象 `requests.codes` 。比如你可以通过 `if r.status_code == requests.codes.ok` 来判断请求是否成功。

用一个实例来感受一下：

```
# coding=utf-8
import requests

r = requests.get('http://www.jianshu.com')
exit() if not r.status_code == requests.codes.ok else print('Request Successfully')
```

在这里，通过比较返回码和内置的成功的返回码是一致的，来保证请求得到了正常响应，输出成功请求的消息，否则程序终止。

那么肯定不能只有 `ok` 这个条件码，还有没有其他的呢？答案是肯定的。

下面列出了返回码和相应的查询条件：

```
# Informational.
100: ('continue',),
101: ('switching_protocols',),
102: ('processing',),
103: ('checkpoint',),
122: ('uri_too_long', 'request_uri_too_long'),
200: ('ok', 'okay', 'all_ok', 'all_okay', 'all_good', '\\o/'
, '✓'),
201: ('created',),
202: ('accepted',),
203: ('non_authoritative_info', 'non_authoritative_information'),
204: ('no_content',),
205: ('reset_content', 'reset'),
206: ('partial_content', 'partial'),
207: ('multi_status', 'multiple_status', 'multi_stati', 'multiple_stati'),
208: ('already_reported',),
226: ('im_used',),
```

```
# Redirection.
300: ('multiple_choices',),
301: ('moved_permanently', 'moved', '\\o-'),
302: ('found',),
303: ('see_other', 'other'),
304: ('not_modified',),
305: ('use_proxy',),
306: ('switch_proxy',),
307: ('temporary_redirect', 'temporary_moved', 'temporary'),
308: ('permanent_redirect',
      'resume_incomplete', 'resume',), # These 2 to be removed in 3.0

# Client Error.
400: ('bad_request', 'bad'),
401: ('unauthorized',),
402: ('payment_required', 'payment'),
403: ('forbidden',),
404: ('not_found', '-o-'),
405: ('method_not_allowed', 'not_allowed'),
406: ('not_acceptable',),
407: ('proxy_authentication_required', 'proxy_auth', 'proxy_authentication'),
408: ('request_timeout', 'timeout'),
409: ('conflict',),
410: ('gone',),
411: ('length_required',),
412: ('precondition_failed', 'precondition'),
413: ('request_entity_too_large',),
414: ('request_uri_too_large',),
415: ('unsupported_media_type', 'unsupported_media', 'media_type'),
416: ('requested_range_not_satisfiable', 'requested_range', 'range_not_satisfiable'),
417: ('expectation_failed',),
418: ('im_a_teapot', 'teapot', 'i_am_a_teapot'),
421: ('misdirected_request',),
422: ('unprocessable_entity', 'unprocessable'),
423: ('locked',),
424: ('failed_dependency', 'dependency'),
```

```
425: ('unordered_collection', 'unordered'),
426: ('upgrade_required', 'upgrade'),
428: ('precondition_required', 'precondition'),
429: ('too_many_requests', 'too_many'),
431: ('header_fields_too_large', 'fields_too_large'),
444: ('no_response', 'none'),
449: ('retry_with', 'retry'),
450: ('blocked_by_windows_parental_controls', 'parental_controls'),
451: ('unavailable_for_legal_reasons', 'legal_reasons'),
499: ('client_closed_request',),

# Server Error.
500: ('internal_server_error', 'server_error', '/o\\', 'x'),
501: ('not_implemented',),
502: ('bad_gateway',),
503: ('service_unavailable', 'unavailable'),
504: ('gateway_timeout',),
505: ('http_version_not_supported', 'http_version'),
506: ('variant_also_negotiates',),
507: ('insufficient_storage',),
509: ('bandwidth_limit_exceeded', 'bandwidth'),
510: ('not_extended',),
511: ('network_authentication_required', 'network_auth', 'network_authentication'),
```

比如如果你想判断结果是不是 404 状态，你可以用 `requests.codes.not_found` 来比对。

响应头

如果想得到响应头信息，可以使用 `headers` 属性。它其实本质上也是一个字典形式。可以通过数组索引或者 `get()` 方法来获取某一条头信息内容。

比如获取 `Content-Type` 可以用 `r.headers['Content-Type']`，也可以用 `r.headers.get('content-type')`，是不是很方便呢？

好，至此我们介绍了利用 `requests` 模拟最基本的 `GET` 和 `POST` 请求的过程，关于更多高级的用法，会在下一节进行讲解。

requests的高级使用

文件上传

我们知道 `requests` 可以模拟提交一些数据，假如有的网站需要我们上传文件，我们同样可以利用它来上传，实现非常简单。

用一个实例来感受一下：

```
# coding=utf-8
import requests

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

在上面一节中我们下载保存了一个文件叫做 `favicon.ico`，这次我们用它为例来模拟文件上传的过程。需要注意的是，`favicon.ico` 这个文件需要和当前脚本在同一目录下。如果有其它文件，当然也可以使用其它文件来上传，更改下名称即可。

运行结果如下：


```
{
  "args": {},
  "data": "",
  "files": {
    "file": "data:application/octet-stream;base64,AAAAA...="
  },
  "form": {},
  "headers": {
    "Accept": "/*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "6665",
    "Content-Type": "multipart/form-data; boundary=809f80b1a2974132b133ade1a8e8e058",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.10.0"
  },
  "json": null,
  "origin": "60.207.237.16",
  "url": "http://httpbin.org/post"
}
```

以上部分内容省略，这个网站会返回一个响应，里面包含 `files` 这个字段，而 `form` 是空的，这证明文件上传部分，会单独有一个 `files` 来标识。

Cookies处理

在前面我们使用了 `urllib`，让它处理 `cookies` 真的是挺麻烦的，而有了 `requests`，获得和提交 `cookies` 只需要一步。

我们先用一个实例感受一下获取 `Cookies` 的过程：

```
# coding=utf-8
import requests

r = requests.get("https://www.baidu.com")
print(r.cookies)
for key, value in r.cookies.items():
    print(key + '=' + value)
```

运行结果如下：

```
<RequestsCookieJar[<Cookie BD0RZ=27315 for .baidu.com/>, <Cookie
__bsi=13533594356813414194_00_14_N_N_2_0303_C02F_N_N_N_0 for .w
ww.baidu.com/>]>
BD0RZ=27315
__bsi=13533594356813414194_00_14_N_N_2_0303_C02F_N_N_N_0
```

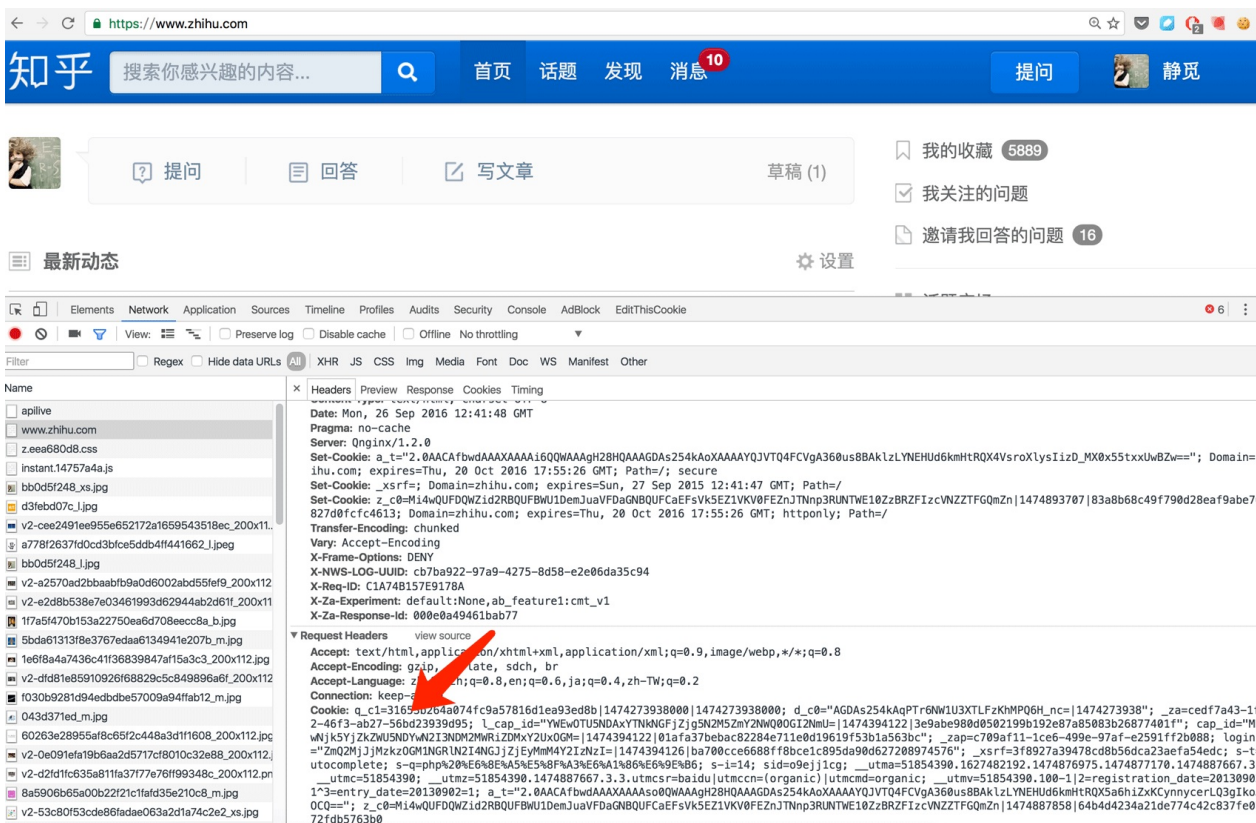
首先打印输出了 `cookie`，可以发现它是一个 `RequestCookieJar` 类型。

然后用 `items()` 方法将其转化为元组组成的列表，遍历输出每一个 `cookie` 的名和值。

当然，你也可以直接用 `Cookie` 来维持登录状态。

比如我们以知乎为例，直接利用 `Cookie` 来维持登录状态。

首先登录知乎，将请求头中的 `Cookie` 复制下来。



将其设置到 `headers` 里面，发送请求：

```
# coding=utf-8
import requests

headers = {
    'Cookie': 'q_c1=31653b264a074fc9a57816d1ea93ed8b|147427393800|1474273938000; d_c0="AGDAs254kAqPTr6NW1U3XTLFzKhMPQ6H_nc=|1474273938"; __utmv=51854390.100-1|2=registration_date=20130902=1^3=entry_date=20130902=1;a_t="2.0AACAfBwdAAAXAAAso0QWAAAgH28HQAAAGDAs254kAoXAAAAYQJVTQ4FCVgA360us8BAk1zLYNEHud6kmHtRQX5a6hiZxKCynnycerLQ3gIkoJL0CQ=="; z_c0=M14wQUFDQWZid2RBQUFBWU1DemJuaVFDaGNBQUFCaEFsVk5EZ1VKV0FEZnJTNnp3RUNTWE10ZzBRZFIZcVNZZTFGQmZn|1474887858|64b4d4234a21de774c42c837fe0b672fdb5763b0',
    'Host': 'www.zhihu.com',
    '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',
}
r = requests.get("http://www.zhihu.com", headers=headers)
print(r.text)
```

发现结果中包含了正常的登录信息。

```
<div role="navigation" class="zu-top" data-za-module="TopNavBar">
<div class="zg-wrap modal-shifting clearfix" id="zh-top-inner">
<a href="/" class="zu-top-link-logo" id="zh-top-link-logo" data-za-c="view_home" data-za-a="visit_home" data-za-l="top_navigation_zhihu_logo">知乎</a>

<div class="top-nav-profile">
<a href="/people/germy" class="zu-top-nav-userinfo" >
<span class="name">静觅</span>

<span id="zh-top-nav-new-pm" class="zg-noti-number zu-top-nav-pm-count"
style="visibility:hidden" data-count="0">
</span>
</div>
```

登录成功！

当然也可以通过cookies参数来设置，不过这样就需要构造

RequestsCookieJar 对象，而且需要分割一下 Cookie 变量，相对繁琐，不过效果是相同的。

```
# coding=utf-8
import requests

cookies = 'q_c1=31653b264a074fc9a57816d1ea93ed8b|1474273938000|1474273938000; d_c0="AGDAs254kAqPTr6NW1U3XTLFzKhMPQ6H_nc=|1474273938"; __utmv=51854390.100-1|2=registration_date=20130902=1^3=entry_date=20130902=1;a_t="2.0AACAFbwdAAAXAAAso0QWAAAgH28HQAAGDAs254kAoXAAAAYQJVTQ4FCVgA360us8BAklzLYNEHud6kmHtRQX5a6hiZxKCynnyce rLQ3gIkoJL0CQ==";z_c0=Ml4wQUFDQWZid2RBQUFBWU1DemJuaVFDaGNBQUFCaEFsVk5EZ1VKV0FEZnJTNnp3RUNTWE10ZzBRZFIZcVNZZTFGQmZn|1474887858|64b4d4234a21de774c42c837fe0b672fdb5763b0'
jar = requests.cookies.RequestsCookieJar()
headers = {
    'Host': 'www.zhihu.com',
    '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'
}
for cookie in cookies.split(';'):
    key, value = cookie.split('=', 1)
    jar.set(key, value)
r = requests.get("http://www.zhihu.com", cookies=jar, headers=headers)
print(r.text)
```

上面我们首先新建了一个 `RequestCookieJar` 对象，然后将复制下来的 `Cookie` 利用 `split()` 方法分割，利用 `set()` 方法设置好每一个 `Cookie` 的 `key` 和 `value`，然后通过 `requests.get()` 方法的 `cookies` 参数设置即可，当然由于知乎本身的限制，`headers` 变量不能少，只不过不需要在原来的 `headers` 里面设置 `Cookie` 字段了。

测试后，发现同样可以正常登录知乎。

会话维持

在 `requests` 中，我们如果直接利用 `requests.get()` 或 `requests.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，即不同的 `session`，也就是说相当于你用了两个浏览器打开了不同的页面。

设想这样一个场景，你第一个请求利用了 `requests.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 `requests.get()` 方法。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，你说你能成功获取个人信息吗？那当然不能。

有小伙伴就说了，我在两次请求的时候都设置好一样的 `Cookie` 不就行了？行是行，但是不觉得麻烦吗？每次都要这样。是我我忍不了。

其实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。但是我又不想每次设置 `Cookie`，那该咋办？这时候就有了新的利器 `Session`。

利用它，我们可以方便地维护一个会话，而且不用担心 `Cookie` 的问题，它会帮我们自动处理好。

下面用一个实例来感受一下：

```
# coding=utf-8
import requests

requests.get('http://httpbin.org/cookies/set/number/123456789')
r = requests.get('http://httpbin.org/cookies')
print(r.text)
```

在实例中我们请求了一个测试网

址，`http://httpbin.org/cookies/set/number/123456789` 请求这个网址我们可以设置一个 `Cookie`，名称叫做 `number`，内容是 `123456789`，后面的网址 `http://httpbin.org/cookies` 可以获取当前的 `Cookie`。

你觉得这样能成功获取到设置的 `Cookie` 吗？试试看。

运行结果如下：

```
{
  "cookies": {}
}
```

喔并不行。那这时候我们想起刚才说的 `Session` 了，改成这个试试看：

```
# coding=utf-8
import requests

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

这下呢？看下运行结果：

```
{
  "cookies": {
    "number": "123456789"
  }
}
```

嗯，成功获取！这下能体会到同一个会话和不同会话的区别了吧？

所以，利用 `Session` 我们可以做到模拟同一个会话，而且不用担心 `Cookie` 的问题，通常用于模拟登录成功之后再进行下一步的操作。