上一课时我们了解了一些学习爬虫所需要的基本知识。从本课时开始，我们正式步入Python爬虫的大门。

学习爬虫，最基础的便是模拟浏览器向服务器发出请求，那么我们需要从什么地方做起呢？请求需要我们自己来构造吗？需要关心请求这个数据结构的实现吗？需要了解 HTTP、TCP、IP 层的网络传输通信吗？需要知道服务器的响应和应答原理吗？

可能你无从下手，不过不用担心，Python 的强大之处就是提供了功能齐全的类库来帮助我们完成这些请求。利用 Python 现有的库我们可以非常方便地实现网络请求的模拟，常见的库有 urllib、requests 等。

拿 requests 这个库来说，有了它，我们只需要关心请求的链接是什么，需要传的参数是什么，以及如何设置可选的参数就好了，不用深入到底层去了解它到底是怎样传输和通信的。有了它，两行代码就可以完成一个请求和响应的处理过程，非常方便地得到网页内容。

接下来，就让我们用 Python 的 requests 库开始我们的爬虫之旅吧。

安装

首先，requests 库是 Python 的一个第三方库，不是自带的。所以我们需要额外安装。

在这之前需要你先安装好 Python3 环境，如 Python 3.6 版本，如若没有安装可以参考：https://cuiqingcai.com/5059.html。

安装好 Python3 之后，我们使用 pip3 即可轻松地安装好 requests 库：

```
pip3 install requests
```

更详细的安装方式可以参考：https://cuiqingcai.com/5132.html。

安装完成之后，我们就可以开始我们的网络爬虫之旅了。

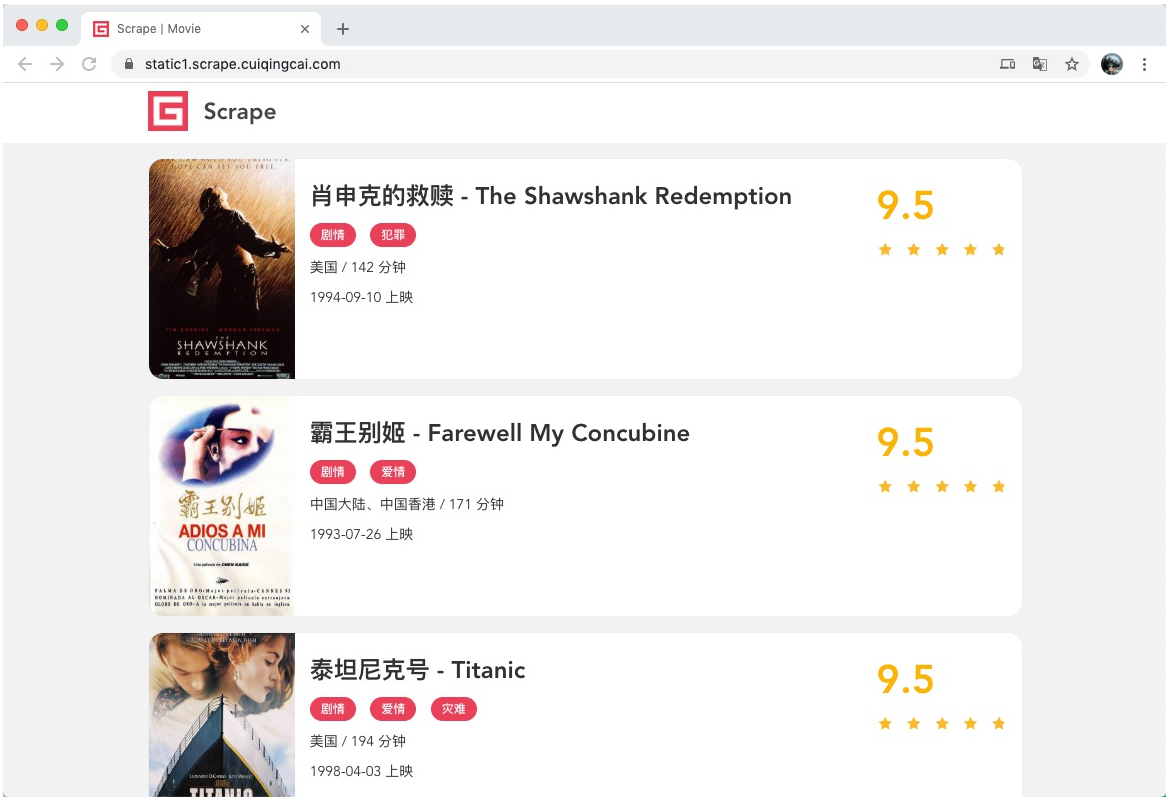
实例引入

用 Python 写爬虫的第一步就是模拟发起一个请求，把网页的源代码获取下来。

当我们在浏览器中输入一个 URL 并回车，实际上就是让浏览器帮我们发起一个 GET 类型的 HTTP 请求，浏览器得到源代码后，把它渲染出来就可以看到网页内容了。

那如果我们想用 requests 来获取源代码，应该怎么办呢？很简单，requests 这个库提供了一个 get 方法，我们调用这个方法，并传入对应的 URL 就能得到网页的源代码。

比如这里有一个示例网站：https://static1.scrape.cuiqingcai.com/，其内容如下：



这个网站展示了一些电影数据，如果我们想要把这个网页里面的数据爬下来，比如获取各个电影的名称、上映时间等信息，然后把它存下来的话，该怎么做呢？

第一步当然就是获取它的网页源代码了。

我们可以用 requests 这个库轻松地完成这个过程，代码的写法是这样的：

```
import requests

r = requests.get('https://static1.scrape.cuiqingcai.com/')
print(r.text)
```

运行结果如下：

```
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width,initial-scale=1">
  <link rel="icon" href="/static/img/favicon.ico">
  <title>Scrape | Movie</title>
  <link href="/static/css/app.css" type="text/css" rel="stylesheet">
  <link href="/static/css/index.css" type="text/css" rel="stylesheet">
</head>
<body>
<div id="app">
...
<div data-v-7f856186="" id="index">
  <div data-v-7f856186="" class="el-row">
    <div data-v-7f856186="" class="el-col el-col-18 el-col-offset-3">
```

```
      <div data-v-7f856186="" class="el-card item m-t is-hover-shadow">
        <div class="el-card__body">
          <div data-v-7f856186="" class="el-row">
            <div data-v-7f856186="" class="el-col el-col-24 el-col-xs-8 el-col-sm-6 el-col-md-4">
              <a data-v-7f856186=""
                 href="/detail/1"
                 class="">
                <img
                    data-v-7f856186="" src="https://p0.meituan.net/movie/283292171619cdfd5b240c8fd093f1eb255670.jpg@464w_644h_1e_1c"
                    class="cover">
              </a>
            </div>
            <div data-v-7f856186="" class="p-h el-col el-col-24 el-col-xs-9 el-col-sm-13 el-col-md-16">
              <a data-v-7f856186="" href="/detail/1" class="">
                <h2 data-v-7f856186="" class="m-b-sm">肖申克的救赎 - The Shawshank Redemption</h2>
              </a>
              <div data-v-7f856186="" class="categories">
                <button data-v-7f856186="" type="button"
                        class="el-button category el-button--primary el-button--mini">
                  <span>剧情</span>
                </button>
                <button data-v-7f856186="" type="button"
                        class="el-button category el-button--primary el-button--mini">
                  <span>犯罪</span>
                </button>
              </div>
              <div data-v-7f856186="" class="m-v-sm info">
                <span data-v-7f856186="">美国</span>
                <span data-v-7f856186=""> / </span>
                <span data-v-7f856186="">142 分钟</span>
              </div>
              <div data-v-7f856186="" class="m-v-sm info">
                <span data-v-7f856186="">1994-09-10 上映</span>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
  ...
</div>
</div>
</body>
```

由于网页内容比较多，这里省略了大部分内容。

不过看运行结果，我们已经成功获取网页的 HTML 源代码，里面包含了电影的标题、类型、上映时间，等等。把网页源代码获取下来之后，下一步我们把想要的数据提取出来，数据的爬取就完成了。

这个实例的目的是让你体会一下 requests 这个库能帮我们实现什么功能。我们仅仅用 requests 的 get 方法就成功发起了一个 GET 请求，把网页源代码获取下来了，是不是很方便呢？

请求

HTTP 中最常见的请求之一就是 GET 请求，下面我们来详细了解利用 requests 库构建 GET 请求的方法。

GET 请求

我们换一个示例网站，其 URL 为 http://httpbin.org/get，如果客户端发起的是 GET 请求的话，该网站会判断并返回相应的请求信息，包括 Headers、IP 等。

我们还是用相同的方法来发起一个 GET 请求，代码如下：

```
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，也通过这个网站的返回结果得到了请求所携带的信息，包括 Headers、URL、IP，等等。

对于 GET 请求，我们知道 URL 后面是可以跟上一些参数的，如果我们现在想添加两个参数，其中 name 是 germey，age 是 25，URL 就可以写成如下内容：

```
http://httpbin.org/get?name=germey&age=25
```

要构造这个请求链接，是不是要直接写成这样呢？

```
r = requests.get('http://httpbin.org/get?name=germey&age=25')
```

这样也可以，但如果这些参数还需要我们手动拼接，未免有点不人性化。

一般情况下，这种信息我们利用 params 这个参数就可以直接传递了，示例如下：

```
import requests

data = {
    'name': 'germey',
    'age': 25
}
r = requests.get('http://httpbin.org/get', params=data)
print(r.text)
```

运行结果如下：

```
{
  "args": {
    "age": "25",
    "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"
```

```
}
```

在这里我们把 URL 参数通过字典的形式传给 get 方法的 params 参数，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，这样我们就不用再去自己构造 URL 了，非常方便。

另外，网页的返回类型实际上是 str 类型，但是它很特殊，是 JSON 格式的。所以，如果想直接解析返回结果，得到一个 JSON 格式的数据的话，可以直接调用 json 方法。

示例如下：

```
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':
<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。

但需要注意的是，如果返回结果不是 JSON 格式，便会出现解析错误，抛出 json.decoder.JSONDecodeError 异常。

抓取网页

上面的请求链接返回的是 JSON 形式的字符串，那么如果请求普通的网页，则肯定能获得相应的内容了。下面以本课时最初的实例页面为例，我们再加上一点提取信息的逻辑，将代码完善成如下的样子：

```
import requests
import re

r = requests.get('https://static1.scrape.cuiqingcai.com/')
pattern = re.compile('<h2.*?>(.*?)</h2>', re.S)
titles = re.findall(pattern, r.text)
print(titles)
```

在这个例子中我们用到了最基础的正则表达式来匹配出所有的标题。关于正则表达式的相关内容，我们会在下一课时详细介绍，这里作为实例来配合讲解。

运行结果如下：

```
['肖申克的救赎 - The Shawshank Redemption', '霸王别姬 - Farewell My Concubine', '泰坦尼克号 - Titanic', '罗马假日 - Roman Holiday', '这个杀手不太冷 - Léon', '魂断蓝桥 - Waterloo Bridge', '唐伯虎点秋
```

我们发现，这里成功提取出了所有的电影标题。一个最基本的抓取和提取流程就完成了。

抓取二进制数据

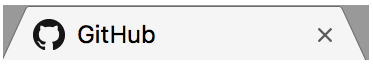
在上面的例子中，我们抓取的是网站的一个页面，实际上它返回的是一个 HTML 文档。如果想抓取图片、音频、视频等文件，应该怎么办呢？

图片、音频、视频这些文件本质上都是由二进制码组成的，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，想要抓取它们，就要拿到它们的二进制数据。

下面以 GitHub 的站点图标为例来看一下：

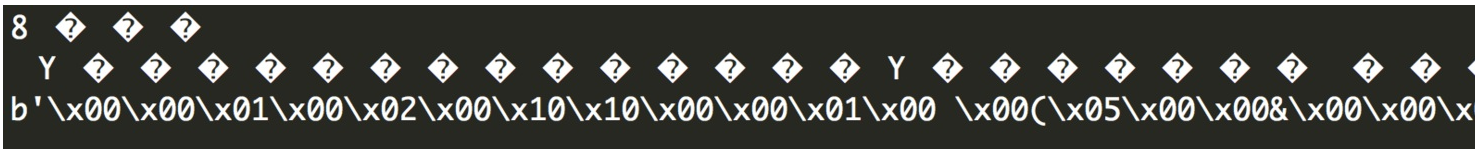
```
import requests

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

这里抓取的内容是站点图标，也就是在浏览器每一个标签上显示的小图标，如图所示：



这里打印了 Response 对象的两个属性，一个是 text，另一个是 content。

运行结果如图所示，其中前两行是 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，后者结果前带有一个 b，这代表是 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，这当然会出现乱码。

上面返回的结果我们并不能看懂，它实际上是图片的二进制数据，没关系，我们将刚才提取到的信息保存下来就好了，代码如下：

```
import requests

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

这里用了 open 方法，它的第一个参数是文件名称，第二个参数代表以二进制的形式打开，可以向文件里写入二进制数据。

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，如图所示。

这样，我们就把二进制数据成功保存成一张图片了，这个小图标就被我们成功爬取下来了。

同样地，音频和视频文件我们也可以用这种方法获取。

添加 headers

我们知道，在发起一个 HTTP 请求的时候，会有一个请求头 Request Headers，那么这个怎么来设置呢？

很简单，我们使用 headers 参数就可以完成了。

在刚才的实例中，实际上我们是没有设置 Request Headers 信息的，如果不设置，某些网站会发现这不是一个正常的浏览器发起的请求，网站可能会返回异常的结果，导致网页抓取失败。

要添加 Headers 信息，比如我们这里想添加一个 User-Agent 字段，我们可以这么来写：

```
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://static1.scrape.cuiqingcai.com/', headers=headers)
print(r.text)
```

当然，我们可以在 headers 这个参数中任意添加其他的字段信息。

POST 请求

前面我们了解了最基本的 GET 请求，另外一种比较常见的请求方式是 POST。使用 requests 实现 POST 请求同样非常简单，示例如下：

```
import requests

data = {'name': 'germey', 'age': '25'}
r = requests.post("http://httpbin.org/post", data=data)
print(r.text)
```

这里还是请求 http://httpbin.org/post，该网站可以判断如果请求是 POST 方式，就把相关请求信息返回。

运行结果如下：

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "age": "25",
    "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/2.22.0",
    "X-Amzn-Trace-Id": "Root=1-5e5bdc26-b40d7e9862e3715f689cb5e6"
  },
  "json": null,
  "origin": "167.220.232.237",
  "url": "http://httpbin.org/post"
}
```

可以发现，我们成功获得了返回结果，其中 form 部分就是提交的数据，这就证明 POST 请求成功发送了。

响应

发送请求后，得到的自然就是响应，即 Response。

在上面的实例中，我们使用 text 和 content 获取了响应的内容。此外，还有很多属性和方法可以用来获取其他信息，比如状态码、响应头、Cookies 等。示例如下：

```
import requests

r = requests.get('https://static1.scrape.cuiqingcai.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)
```

这里分别打印输出 status_code 属性得到状态码，输出 headers 属性得到响应头，输出 cookies 属性得到 Cookies，输出 url 属性得到 URL，输出 history 属性得到请求历史。

运行结果如下：

```
<class 'int'> 200
<class 'requests.structures.CaseInsensitiveDict'> {'Server': 'nginx/1.17.8', 'Date': 'Sun, 01 Mar 2020 13:31:54 GMT', 'Content-Type': 'text/html; charset=utf-8', 'Transfer-Encoding': '
<class 'requests.cookies.RequestsCookieJar'> <RequestsCookieJar[]>
<class 'str'> https://static1.scrape.cuiqingcai.com/
<class 'list'> []
```

可以看到，headers 和 cookies 这两个属性得到的结果分别是 CaseInsensitiveDict 和 RequestsCookieJar 类型。

在第一课时我们知道，状态码是用来表示响应状态的，比如返回 200 代表我们得到的响应是没问题的，上面的例子正好输出的结果也是 200，所以我们可以通过判断 Response 的状态码来确认是否爬取成功。

requests 还提供了一个内置的状态码查询对象 requests.codes，用法示例如下：

```
import requests

r = requests.get('https://static1.scrape.cuiqingcai.com/')
exit() if not r.status_code == requests.codes.ok else print('Request Successfully')
```

这里通过比较返回码和内置的成功的返回码，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，这里我们用 requests.codes.ok 得到的是成功的状态码 200。

这样的话，我们就不用再在程序里面写状态码对应的数字了，用字符串表示状态码会显得更加直观。

当然，肯定不能只有 ok 这个条件码。

下面列出了返回码和相应的查询条件：

```
# 信息性状态码
100: ('continue',),
101: ('switching_protocols',),
102: ('processing',),
103: ('checkpoint',),
122: ('uri_too_long', 'request_uri_too_long'),

# 成功状态码
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',),

# 重定向状态码
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

# 客户端错误状态码
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',),

# 服务端错误状态码
500: ('internal_server_error', 'server_error', '/o\\', '✗'),
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，如果想判断结果是不是 404 状态，可以用 requests.codes.not_found 来比对。

高级用法

刚才，我们了解了 requests 的基本用法，如基本的 GET、POST 请求以及 Response 对象。当然 requests 能做到的不仅这些，它几乎可以帮我们完成 HTTP 的所有操作。

下面我们再来了解下 requests 的一些高级用法，如文件上传、Cookies 设置、代理设置等。

文件上传

我们知道 requests 可以模拟提交一些数据。假如有的网站需要上传文件，我们也可以用它来实现，示例如下：

```
import requests

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

在上一课时中我们保存了一个文件 favicon.ico，这次用它来模拟文件上传的过程。需要注意的是，favicon.ico 需要和当前脚本在同一目录下。如果有其他文件，当然也可以使用其他文件来上传，更改下代码即可。

运行结果如下：

```
{"args": {},
  "data": "","files": {"file":"data:application/octet-stream;base64,AAAAAA...="},"form": {},"headers": {"Accept":"*/*","Accept-Encoding":"gzip, deflate","Content-Length":"6665","Conten
```

以上省略部分内容，这个网站会返回响应，里面包含 files 这个字段，而 form 字段是空的，这证明文件上传部分会单独有一个 files 字段来标识。

Cookies

我们如果想用 requests 获取和设置 Cookies 也非常方便，只需一步即可完成。

我们先用一个实例看一下获取 Cookies 的过程：

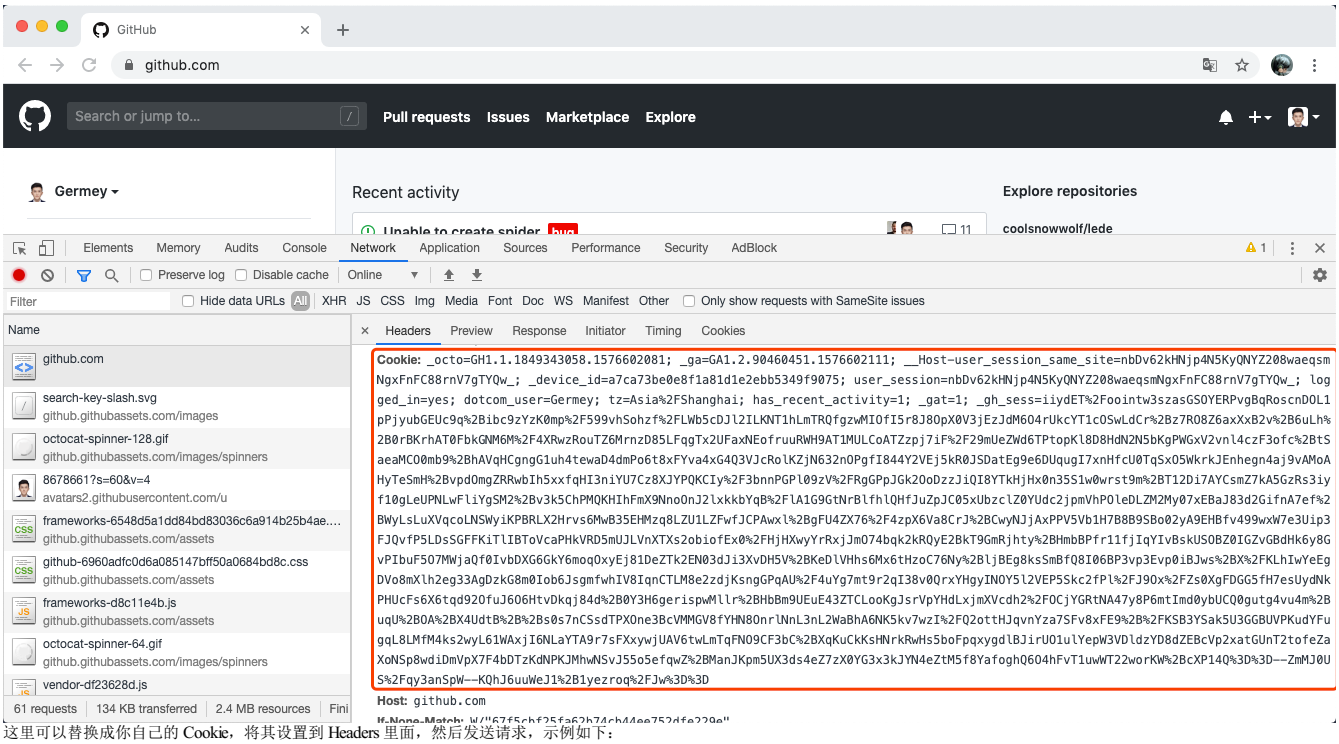
```
import requests

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

运行结果如下：

```
<RequestsCookieJar[<Cookie BDORZ=27315 for .baidu.com/>]>
BDORZ=27315
```

这里我们首先调用 cookies 属性即可成功得到 Cookies，可以发现它是 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 的名称和值，实现 Cookie 的遍历解析。

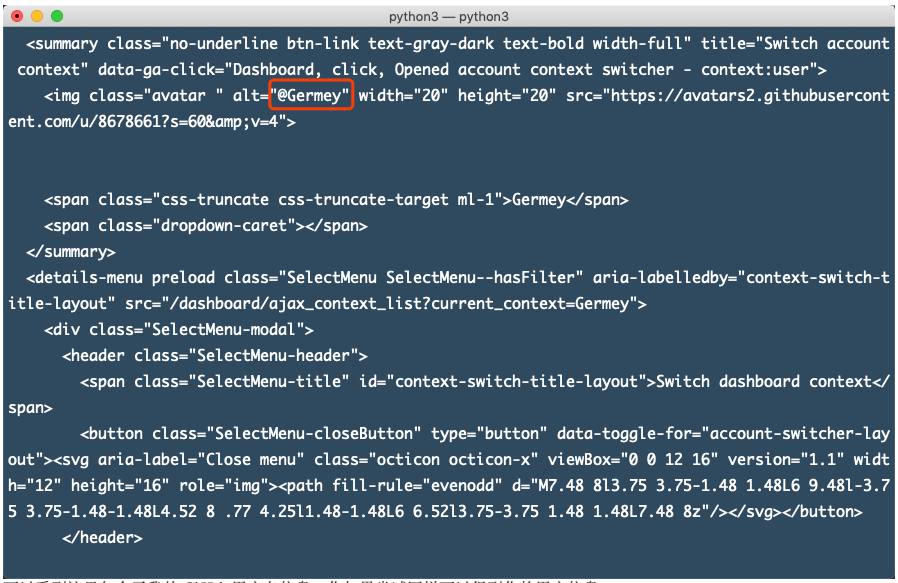
当然，我们也可以直接用 Cookie 来维持登录状态，下面我们以 GitHub 为例来说明一下，首先我们登录 GitHub，然后将 Headers 中的 Cookie 内容复制下来，如图所示：



这里可以替换成你自己的 Cookie，将其设置到 Headers 里面，然后发送请求，示例如下：

```
import requests

headers = {
    'Cookie': '_octo=GH1.1.1849343058.1576602081; _ga=GA1.2.90460451.1576602111; __Host-user_session_same_site=nbDv62kHNjp4N5KyQNYZ208waeqsmNgxFnFC88rnV7gTYQw_; _device_id=a7ca73be0e8f',
    '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('https://github.com/', headers=headers)
print(r.text)
```

我们发现，结果中包含了登录后才能显示的结果，如图所示：



可以看到这里包含了我的 GitHub 用户名信息，你如果尝试同样可以得到你的用户信息。

得到这样类似的结果，说明我们用 Cookies 成功模拟了登录状态，这样我们就能爬取登录之后才能看到的页面了。

当然，我们也可以通过 cookies 参数来设置 Cookies 的信息，这里我们可以构造一个 RequestsCookieJar 对象，然后把刚才复制的 Cookie 处理下并赋值，示例如下：

```
import requests

cookies = '_octo=GH1.1.1849343058.1576602081; _ga=GA1.2.90460451.1576602111; __Host-user_session_same_site=nbDv62kHNjp4N5KyQNYZ208waeqsmNgxFnFC88rnV7gTYQw_; _device_id=a7ca73be0e8f1a81
jar = requests.cookies.RequestsCookieJar()
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'
}
for cookie in cookies.split(';'):
    key, value = cookie.split('=', 1)
    jar.set(key, value)
r = requests.get('https://github.com/', cookies=jar, headers=headers)
print(r.text)
```

这里我们首先新建一个 RequestCookieJar 对象，然后将复制下来的 cookies 利用 split 方法分割，接着利用 set 方法设置好每个 Cookie 的 key 和 value，最后通过调用 requests 的 get 方法并传递给 cookies 参数即可。

测试后，发现同样可以正常登录。

Session 维持

在 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 Session，相当于你用两个浏览器打开了不同的页面。

设想这样一个场景，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 Session，能成功获取个人信息吗？当然不能。

有人会问，我在两次请求时设置一样的 Cookies 不就行了？可以，但这样做起来很烦琐，我们有更简单的解决方法。

解决这个问题的主要方法就是维持同一个 Session，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。但我又不想每次设置 Cookies，那该怎么办呢？这时候就有了新的利器 ——Session 对象。

利用它，我们可以方便地维护一个 Session，而且不用担心 Cookies 的问题，它会帮我们自动处理好。示例如下：

```
import requests

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

这里我们请求了一个测试网址 http://httpbin.org/cookies/set/number/123456789。请求这个网址时，可以设置一个 cookie，名称叫作 number，内容是 123456789，随后又请求了 http://httpbin.org/cookies，此网址可以获取当前的 Cookies。

这样能成功获取到设置的 Cookies 吗？试试看。

运行结果如下：

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

这并不行。我们再用 Session 试试看：

```
import requests

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

再看下运行结果：

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

成功获取！这下能体会到同一个Session和不同Session的区别了吧！

所以，利用 Session，可以做到模拟同一个 Session 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。

SSL 证书验证

现在很多网站都要求使用 HTTPS 协议，但是有些网站可能并没有设置好 HTTPS 证书，或者网站的 HTTPS 证书不被 CA 机构认可，这时候，这些网站可能就会出现 SSL 证书错误的提示。

比如这个示例网站：https://static2.scrape.cuiqingcai.com/。

如果我们用 Chrome 浏览器打开这个 URL，则会提示「您的连接不是私密连接」这样的错误，如图所示：

我们可以在浏览器中通过一些设置来忽略证书的验证。

但是如果我们想用 requests 来请求这类网站，会遇到什么问题呢？我们用代码来试一下：

```
import requests

response = requests.get('https://static2.scrape.cuiqingcai.com/')
print(response.status_code)
```

运行结果如下：

```
requests.exceptions.SSLError: HTTPSConnectionPool(host='static2.scrape.cuiqingcai.com', port=443): Max retries exceeded with url: / (Caused by SSLError(SSLError("bad handshake: Error([
```

可以看到，这里直接抛出了 SSLError 错误，原因就是因为我们请求的 URL 的证书是无效的。

那如果我们一定要爬取这个网站怎么办呢？我们可以使用 verify 参数控制是否验证证书，如果将其设置为 False，在请求时就不会再验证证书是否有效。如果不加 verify 参数的话，默认值是 True，会自动验证。

我们改写代码如下：

```
import requests

response = requests.get('https://static2.scrape.cuiqingcai.com/', verify=False)
print(response.status_code)
```

这样就会打印出请求成功的状态码：

```
/usr/local/lib/python3.7/site-packages/urllib3/connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly advise
```

```
  InsecureRequestWarning)
200
```

不过我们发现报了一个警告，它建议我们给它指定证书。我们可以通过设置忽略警告的方式来屏蔽这个警告：

```
import requests
from requests.packages import urllib3

urllib3.disable_warnings()
response = requests.get('https://static2.scrape.cuiqingcai.com/', verify=False)
print(response.status_code)
```

或者通过捕获警告到日志的方式忽略警告：

```
import logging
import requests
logging.captureWarnings(True)
response = requests.get('https://static2.scrape.cuiqingcai.com/', verify=False)
print(response.status_code)
```

当然，我们也可以指定一个本地证书用作客户端证书，这可以是单个文件（包含密钥和证书）或一个包含两个文件路径的元组：

```
import requests

response = requests.get('https://static2.scrape.cuiqingcai.com/', cert=('/path/server.crt', '/path/server.key'))
print(response.status_code)
```

当然，上面的代码是演示实例，我们需要有 crt 和 key 文件，并且指定它们的路径。另外注意，本地私有证书的 key 必须是解密状态，加密状态的 key 是不支持的。

超时设置

在本机网络状况不好或者服务器网络响应延迟甚至无响应时，我们可能会等待很久才能收到响应，甚至到最后收不到响应而报错。为了防止服务器不能及时响应，应该设置一个超时时间，即超过了这个时间还没有得到响应，那就报错。这需要用到 timeout 参数。这个时间的计算是发出请求到服务器返回响应的时间。示例如下：

```
import requests

r = requests.get('https://httpbin.org/get', timeout=1)
print(r.status_code)
```

通过这样的方式，我们可以将超时时间设置为 1 秒，如果 1 秒内没有响应，那就抛出异常。

实际上，请求分为两个阶段，即连接（connect）和读取（read）。

上面设置的 timeout 将用作连接和读取这二者的 timeout 总和。

如果要分别指定，就可以传入一个元组：

```
r = requests.get('https://httpbin.org/get', timeout=(5, 30))
```

如果想永久等待，可以直接将 timeout 设置为 None，或者不设置直接留空，因为默认是 None。这样的话，如果服务器还在运行，但是响应特别慢，那就慢慢等吧，它永远不会返回超时错误的。其用法如下：

```
r = requests.get('https://httpbin.org/get', timeout=None)
```

或直接不加参数：

```
r = requests.get('https://httpbin.org/get')
```

身份认证

在访问某些设置了身份认证的网站时，例如：https://static3.scrape.cuiqingcai.com/，我们可能会遇到这样的认证窗口，如图所示：

如果遇到了这种情况，那就是这个网站启用了基本身份认证，英文叫作 HTTP Basic Access Authentication，它是一种用来允许网页浏览器或其他客户端程序在请求时提供用户名和口令形式的身份凭证的一种登录验证方式。

如果遇到了这种情况，怎么用 reqeusts 来爬取呢，当然也有办法。

我们可以使用 requests 自带的身份认证功能，通过 auth 参数即可设置，示例如下：

```
import requests
from requests.auth import HTTPBasicAuth

r = requests.get('https://static3.scrape.cuiqingcai.com/', auth=HTTPBasicAuth('admin', 'admin'))
print(r.status_code)
```

这个示例网站的用户名和密码都是 admin，在这里我们可以直接设置。

如果用户名和密码正确的话，请求时会自动认证成功，返回 200 状态码；如果认证失败，则返回 401 状态码。

当然，如果参数都传一个 HTTPBasicAuth 类，就显得有点烦琐了，所以 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。

所以上面的代码可以直接简写如下：

```
import requests

r = requests.get('https://static3.scrape.cuiqingcai.com/', auth=('admin', 'admin'))
print(r.status_code)
```

此外，requests 还提供了其他认证方式，如 OAuth 认证，不过此时需要安装 oauth 包，安装命令如下：

```
pip3 install requests_oauthlib
```

使用 OAuth1 认证的方法如下：

```
import requests
from requests_oauthlib import OAuth1

url = 'https://api.twitter.com/1.1/account/verify_credentials.json'
auth = OAuth1('YOUR_APP_KEY', 'YOUR_APP_SECRET',
              'USER_OAUTH_TOKEN', 'USER_OAUTH_TOKEN_SECRET')
```

```
requests.get(url, auth=auth)
```

更多详细的功能就可以参考 requests_oauthlib 的官方文档：https://requests-oauthlib.readthedocs.org/，在此就不再赘述了。

代理设置

某些网站在测试的时候请求几次，能正常获取内容。但是对于大规模且频繁的请求，网站可能会弹出验证码，或者跳转到登录认证页面，更甚者可能会直接封禁客户端的 IP，导致一定时间段内无法访问。

为了防止这种情况发生，我们需要设置代理来解决这个问题，这就需要用到 proxies 参数。可以用这样的方式设置：

```
import requests

proxies = {
  'http': 'http://10.10.10.10:1080',
  'https': 'http://10.10.10.10:1080',
}
requests.get('https://httpbin.org/get', proxies=proxies)
```

当然，直接运行这个实例或许行不通，因为这个代理可能是无效的，可以直接搜索寻找有效的代理并替换试验一下。

若代理需要使用上文所述的身份认证，可以使用类似 http://user:password@host:port 这样的语法来设置代理，示例如下：

```
import requests

proxies = {'https': 'http://user:password@10.10.10.10:1080/',}
requests.get('https://httpbin.org/get', proxies=proxies)
```

除了基本的 HTTP 代理外，requests 还支持 SOCKS 协议的代理。

首先，需要安装 socks 这个库：

```
pip3 install "requests[socks]"
```

然后就可以使用 SOCKS 协议代理了，示例如下：

```
import requests

proxies = {
    'http': 'socks5://user:password@host:port',
    'https': 'socks5://user:password@host:port'
}
requests.get('https://httpbin.org/get', proxies=proxies)
```

Prepared Request

我们使用 requests 库的 get 和 post 方法可以直接发送请求，但你有没有想过，这个请求在 requests 内部是怎么实现的呢？

实际上，requests 在发送请求的时候在内部构造了一个 Request 对象，并给这个对象赋予了各种参数，包括 url、headers、data，等等。然后直接把这个 Request 对象发送出去，请求成功后会再得到一个 Response 对象，再解析即可。

那么这个 Request 是什么类型呢？实际上它就是 Prepared Request。

我们深入一下，不用 get 方法，直接构造一个 Prepared Request 对象来试试，代码如下：

```
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)
```

这里我们引入了 Request，然后用 url、data 和 headers 参数构造了一个 Request 对象，这时需要再调用 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，运行结果如下：

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "name": "germey"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "11",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "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",
    "X-Amzn-Trace-Id": "Root=1-5e5bd6a9-6513c838f35b06a0751606d8"
  },
  "json": null,
  "origin": "167.220.232.237",
  "url": "http://httpbin.org/post"
}
```

可以看到，我们达到了同样的 POST 请求效果。

有了 Request 这个对象，就可以将请求当作独立的对象来看待，这样在一些场景中我们可以直接操作这个 Request 对象，更灵活地实现请求的调度和各种操作。

更多的用法可以参考 requests 的官方文档：http://docs.python-requests.org/。

本课时 requests 库的基本用法就介绍到这里了。怎么样？是不是找到一点爬虫的感觉了？