***Scrapy笔记***

JavaScript通常也是以单独的文件形式加载的，后缀为js，

在HTML中通过script标签即可引入，例如

|  |  |
| --- | --- |
| 1 | <script src="jquery-2.1.0.js"></script> |

综上所述，HTML定义了网页的内容和结构，CSS描述了网页的布局，JavaScript定义了网页的行为。

一个网页的标准形式是html标签内嵌套head和body标签，head内定义网页的配置和引用，body内定义网页的正文。

表2-4 CSS选择器的其他语法规则

| 选择器 | 例子 | 例子描述 |
| --- | --- | --- |
| .class | .intro | 选择class="intro"的所有节点 |
| #id | #firstname | 选择id="firstname"的所有节点 |
| \* | \* | 选择所有节点 |
| element | p | 选择所有p节点 |
| element,element | div,p | 选择所有div节点和所有p节点 |
| element element | div p | 选择div节点内部的所有p节点 |
| element>element | div>p | 选择父节点为div节点的所有p节点 |
| element+element | div+p | 选择紧接在div节点之后的所有p节点 |
| [attribute] | [target] | 选择带有target属性的所有节点 |
| [attribute=value] | [target=blank] | 选择target="blank"的所有节点 |
| [attribute~=value] | [title~=flower] | 选择title属性包含单词flower的所有节点 |
| :link | a:link | 选择所有未被访问的链接 |
| :visited | a:visited | 选择所有已被访问的链接 |
| :active | a:active | 选择活动链接 |
| :hover | a:hover | 选择鼠标指针位于其上的链接 |
| :focus | input:focus | 选择获得焦点的input节点 |
| :first-letter | p:first-letter | 选择每个p节点的首字母 |
| :first-line | p:first-line | 选择每个p节点的首行 |
| :first-child | p:first-child | 选择属于父节点的第一个子节点的所有p节点 |
| :before | p:before | 在每个p节点的内容之前插入内容 |
| :after | p:after | 在每个p节点的内容之后插入内容 |
| :lang(language) | p:lang | 选择带有以it开头的lang属性值的所有p节点 |
| element1~element2 | p~ul | 选择前面有p节点的所有ul节点 |
| [attribute^=value] | a[src^="https"] | 选择其src属性值以https开头的所有a节点 |
| [attribute$=value] | a[src$=".pdf"] | 选择其src属性以.pdf结尾的所有a节点 |
| [attribute\*=value] | a[src\*="abc"] | 选择其src属性中包含abc子串的所有a节点 |
| :first-of-type | p:first-of-type | 选择属于其父节点的首个p节点的所有p节点 |
| :last-of-type | p:last-of-type | 选择属于其父节点的最后p节点的所有p节点 |
| :only-of-type | p:only-of-type | 选择属于其父节点唯一的p节点的所有p节点 |
| :only-child | p:only-child | 选择属于其父节点的唯一子节点的所有p节点 |
| :nth-child(n) | p:nth-child | 选择属于其父节点的第二个子节点的所有p节点 |
| :nth-last-child(n) | p:nth-last-child | 同上，从最后一个子节点开始计数 |
| :nth-of-type(n) | p:nth-of-type | 选择属于其父节点第二个p节点的所有p节点 |
| :nth-last-of-type(n) | p:nth-last-of-type | 同上，但是从最后一个子节点开始计数 |
| :last-child | p:last-child | 选择属于其父节点最后一个子节点的所有p节点 |
| :root | :root | 选择文档的根节点 |
| :empty | p:empty | 选择没有子节点的所有p节点（包括文本节点） |
| :target | #news:target | 选择当前活动的#news节点 |
| :enabled | input:enabled | 选择每个启用的input节点 |
| :disabled | input:disabled | 选择每个禁用的input节点 |
| :checked | input:checked | 选择每个被选中的input节点 |
| :not(selector) | :not | 选择非p节点的所有节点 |
| ::selection | ::selection | 选择被用户选取的节点部分 |

另外，还有一种比较常用的选择器是XPath，这种选择方式后面会详细介绍。

1. 爬虫概述

简单来说，爬虫就是获取网页并提取和保存信息的自动化程序，下面概要介绍一下。

(1) 获取网页

爬虫首先要做的工作就是获取网页，这里就是获取网页的源代码。源代码里包含了网页的部分有用信息，所以只要把源代码获取下来，就可以从中提取想要的信息了。

前面讲了请求和响应的概念，向网站的服务器发送一个请求，返回的响应体便是网页源代码。所以，最关键的部分就是构造一个请求并发送给服务器，然后接收到响应并将其解析出来，那么这个流程怎样实现呢？总不能手工去截取网页源码吧？

不用担心，Python提供了许多库来帮助我们实现这个操作，如urllib、requests等。我们可以用这些库来帮助我们实现HTTP请求操作，请求和响应都可以用类库提供的数据结构来表示，得到响应之后只需要解析数据结构中的Body部分即可，即得到网页的源代码，这样我们可以用程序来实现获取网页的过程了。

(2) 提取信息

获取网页源代码后，接下来就是分析网页源代码，从中提取我们想要的数据。首先，最通用的方法便是采用正则表达式提取，这是一个万能的方法，但是在构造正则表达式时比较复杂且容易出错。

另外，由于网页的结构有一定的规则，所以还有一些根据网页节点属性、CSS选择器或XPath来提取网页信息的库，如Beautiful Soup、pyquery、lxml等。使用这些库，我们可以高效快速地从中提取网页信息，如节点的属性、文本值等。

提取信息是爬虫非常重要的部分，它可以使杂乱的数据变得条理清晰，以便我们后续处理和分析数据。

(3) 保存数据

提取信息后，我们一般会将提取到的数据保存到某处以便后续使用。这里保存形式有多种多样，如可以简单保存为TXT文本或JSON文本，也可以保存到数据库，如MySQL和MongoDB等，也可保存至远程服务器，如借助SFTP进行操作等。

(4) 自动化程序

说到自动化程序，意思是说爬虫可以代替人来完成这些操作。首先，我们手工当然可以提取这些信息，但是当量特别大或者想快速获取大量数据的话，肯定还是要借助程序。爬虫就是代替我们来完成这份爬取工作的自动化程序，它可以在抓取过程中进行各种异常处理、错误重试等操作，确保爬取持续高效地运行。

在网页中我们能看到各种各样的信息，最常见的便是常规网页，它们对应着HTML代码，而最常抓取的便是HTML源代码。

另外，可能有些网页返回的不是HTML代码，而是一个JSON字符串（其中API接口大多采用这样的形式），这种格式的数据方便传输和解析，它们同样可以抓取，而且数据提取更加方便。

此外，我们还可以看到各种二进制数据，如图片、视频和音频等。利用爬虫，我们可以将这些二进制数据抓取下来，然后保存成对应的文件名。

另外，还可以看到各种扩展名的文件，如CSS、JavaScript和配置文件等，这些其实也是最普通的文件，只要在浏览器里面可以访问到，就可以将其抓取下来。

上述内容其实都对应各自的URL，是基于HTTP或HTTPS协议的，只要是这种数据，爬虫都可以抓取。

3. JavaScript渲染页面

有时候，我们在用urllib或requests抓取网页时，得到的源代码实际和浏览器中看到的不一样。

这是一个非常常见的问题。现在网页越来越多地采用Ajax、前端模块化工具来构建，整个网页可能都是由JavaScript渲染出来的，也就是说原始的HTML代码就是一个空壳，例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <!DOCTYPE html>  <html>      <head>          <meta charset="UTF-8">          <title>This is a Demo</title>      </head>      <body>          <div id="container">          </div>      </body>      <script src="app.js"></script>  </html> |

body节点里面只有一个id为container的节点，但是需要注意在body节点后引入了app.js，它便负责整个网站的渲染。

因此，使用基本HTTP请求库得到的源代码可能跟浏览器中的页面源代码不太一样。对于这样的情况，我们可以分析其后台Ajax接口，也可使用Selenium、Splash这样的库来实现模拟JavaScript渲染。

## 1. 静态网页和动态网页

这种网页的内容是HTML代码编写的，文字、图片等内容均通过写好的HTML代码来指定，这种页面叫作静态网页。它加载速度快，编写简单，但是存在很大的缺陷，如可维护性差，不能根据URL灵活多变地显示内容等。例如，我们想要给这个网页的URL传入一个name参数，让其在网页中显示出来，是无法做到的。

因此，动态网页应运而生，它可以动态解析URL中参数的变化，关联数据库并动态呈现不同的页面内容，非常灵活多变。我们现在遇到的大多数网站都是动态网站，它们不再是一个简单的HTML，而是可能由JSP、PHP、Python等语言编写的，其功能比静态网页强大和丰富太多了。

2. 无状态HTTP

在了解会话和Cookies之前，我们还需要了解HTTP的一个特点，叫作无状态。

HTTP的无状态是指HTTP协议对事务处理是没有记忆能力的，也就是说服务器不知道客户端是什么状态。当我们向服务器发送请求后，服务器解析此请求，然后返回对应的响应，服务器负责完成这个过程，而且这个过程是完全独立的，服务器不会记录前后状态的变化，也就是缺少状态记录。这意味着如果后续需要处理前面的信息，则必须重传，这导致需要额外传递一些前面的重复请求，才能获取后续响应，然而这种效果显然不是我们想要的。为了保持前后状态，我们肯定不能将前面的请求全部重传一次，这太浪费资源了，对于这种需要用户登录的页面来说，更是棘手。

这时两个用于保持HTTP连接状态的技术就出现了，它们分别是会话和Cookies。会话在服务端，也就是网站的服务器，用来保存用户的会话信息；Cookies在客户端，也可以理解为浏览器端，有了Cookies，浏览器在下次访问网页时会自动附带上它发送给服务器，服务器通过识别Cookies并鉴定出是哪个用户，然后再判断用户是否是登录状态，然后返回对应的响应。

我们可以理解为Cookies里面保存了登录的凭证，有了它，只需要在下次请求携带Cookies发送请求而不必重新输入用户名、密码等信息重新登录了。

因此在爬虫中，有时候处理需要登录才能访问的页面时，我们一般会直接将登录成功后获取的Cookies放在请求头里面直接请求，而不必重新模拟登录。

好了，了解会话和Cookies的概念之后，我们在来详细剖析它们的原理。

(1) 会话

会话，其本来的含义是指有始有终的一系列动作/消息。比如，打电话时，从拿起电话拨号到挂断电话这中间的一系列过程可以称为一个会话。

而在Web中，会话对象用来存储特定用户会话所需的属性及配置信息。这样，当用户在应用程序的Web页之间跳转时，存储在会话对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。当用户请求来自应用程序的Web页时，如果该用户还没有会话，则Web服务器将自动创建一个会话对象。当会话过期或被放弃后，服务器将终止该会话。

(2) Cookies

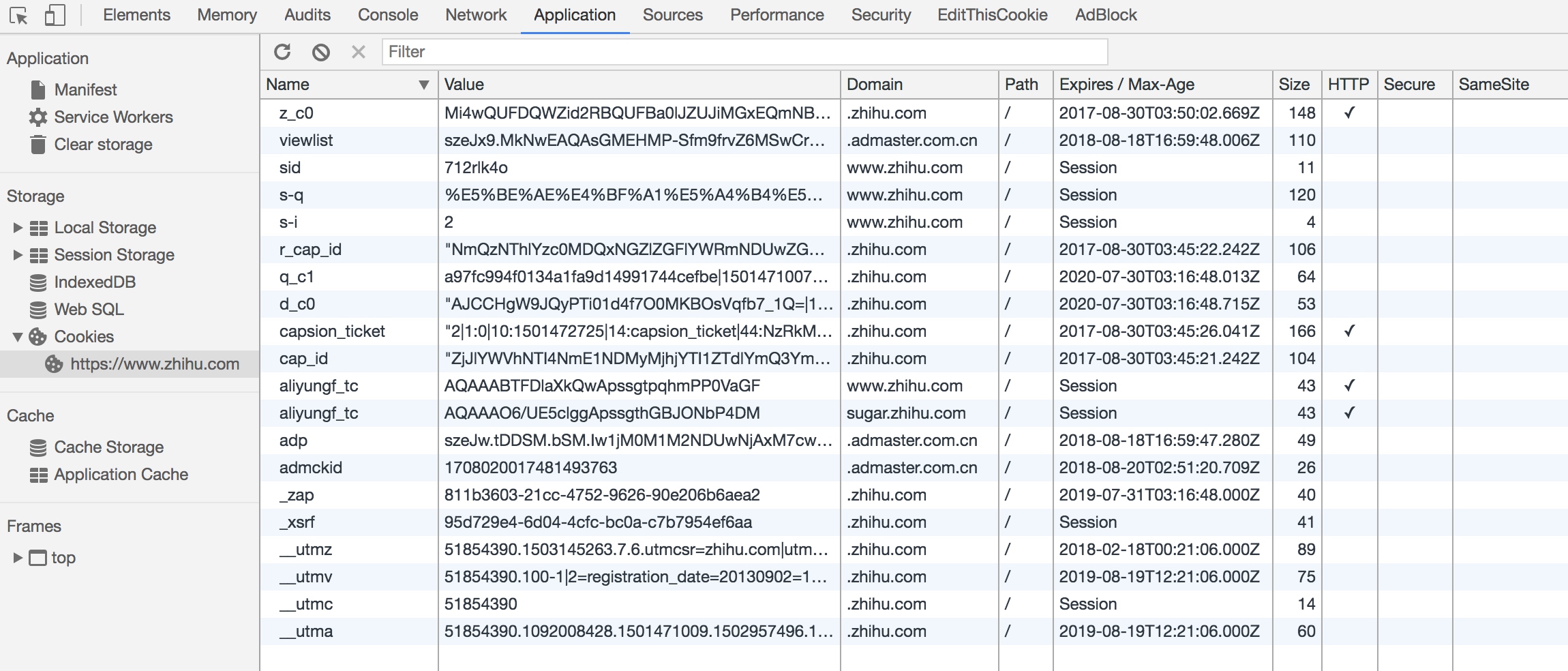
Cookies指某些网站为了辨别用户身份、进行会话跟踪而存储在用户本地终端上的数据。

会话维持

那么，我们怎样利用Cookies保持状态呢？当客户端第一次请求服务器时，服务器会返回一个请求头中带有Set-Cookie字段的响应给客户端，用来标记是哪一个用户，客户端浏览器会把Cookies保存起来。当浏览器下一次再请求该网站时，浏览器会把此Cookies放到请求头一起提交给服务器，Cookies携带了会话ID信息，服务器检查该Cookies即可找到对应的会话是什么，然后再判断会话来以此来辨认用户状态。

属性结构

接下来，我们来看看Cookies都有哪些内容。这里以知乎为例，在浏览器开发者工具中打开Application选项卡，然后在左侧会有一个Storage部分，最后一项即为Cookies，将其点开，如图2-13所示，这些就是Cookies。

图2-13 Cookies列表

可以看到，这里有很多条目，其中每个条目可以称为Cookie。它有如下几个属性。

Name：该Cookie的名称。一旦创建，该名称便不可更改。

Value：该Cookie的值。如果值为Unicode字符，需要为字符编码。如果值为二进制数据，则需要使用BASE64编码。

Domain：可以访问该Cookie的域名。例如，如果设置为.zhihu.com，则所有以zhihu.com，结尾的域名都可以访问该Cookie。

Max Age：该Cookie失效的时间，单位为秒，也常和Expires一起使用，通过它可以计算出其有效时间。Max Age如果为正数，则该Cookie在Max Age秒之后失效。如果为负数，则关闭浏览器时Cookie即失效，浏览器也不会以任何形式保存该Cookie。

Path：该Cookie的使用路径。如果设置为/path/，则只有路径为/path/的页面可以访问该Cookie。如果设置为/，则本域名下的所有页面都可以访问该Cookie。

Size字段：此Cookie的大小。

HTTP字段：Cookie的httponly属性。若此属性为true，则只有在HTTP头中会带有此Cookie的信息，而不能通过document.cookie来访问此Cookie。

Secure：该Cookie是否仅被使用安全协议传输。安全协议有HTTPS和SSL等，在网络上传输数据之前先将数据加密。默认为false。

会话Cookie和持久Cookie

从表面意思来说，会话Cookie就是把Cookie放在浏览器内存里，浏览器在关闭之后该Cookie即失效；持久Cookie则会保存到客户端的硬盘中，下次还可以继续使用，用于长久保持用户登录状态。

其实严格来说，没有会话Cookie和持久Cookie之分，只是由Cookie的Max Age或Expires字段决定了过期的时间。

因此，一些持久化登录的网站其实就是把Cookie的有效时间和会话有效期设置得比较长，下次我们再访问页面时仍然携带之前的Cookie，就可以直接保持登录状态。

3. 常见误区

在谈论会话机制的时候，常常听到这样一种误解“只要关闭浏览器，会话就消失了”，这种理解是错误的。可以想象一下会员卡的例子，除非顾客主动对店家提出销卡，否则店家绝对不会轻易删除顾客的资料。对会话来说，也是一样，除非程序通知服务器删除一个会话，否则服务器会一直保留。比如，程序一般都是在我们做注销操作时才去删除会话。

但是当我们关闭浏览器时，浏览器不会主动在关闭之前通知服务器它将要关闭，所以服务器根本不会有机会知道浏览器已经关闭。之所以会有这种错觉，是因为大部分会话机制都使用会话Cookie来保存会话ID信息，而关闭浏览器后Cookies就消失了，再次连接服务器时，也就无法找到原来的会话了。如果服务器设置的Cookies保存到硬盘上，或者使用某种手段改写浏览器发出的HTTP请求头，把原来的Cookies发送给服务器，则再次打开浏览器，仍然能够找到原来的会话 ID，依旧还是可以保持登录状态的。

而且恰恰是由于关闭浏览器不会导致会话被删除，这就需要服务器为会话设置一个失效时间，当距离客户端上一次使用会话的时间超过这个失效时间时，服务器就可以认为客户端已经停止了活动，才会把会话删除以节省存储空间。

既然服务器检测的是某个IP单位时间的请求次数，那么借助某种方式来伪装我们的IP，让服务器识别不出是由我们本机发起的请求，不就可以成功防止封IP了吗？

一种有效的方式就是使用代理，后面会详细说明代理的用法。在这之前，需要先了解下代理的基本原理，它是怎样实现IP伪装的呢？

1. 基本原理

代理实际上指的就是代理服务器，英文叫作proxy server，它的功能是代理网络用户去取得网络信息。形象地说，它是网络信息的中转站。在我们正常请求一个网站时，是发送了请求给Web服务器，Web服务器把响应传回给我们。如果设置了代理服务器，实际上就是在本机和服务器之间搭建了一个桥，此时本机不是直接向Web服务器发起请求，而是向代理服务器发出请求，请求会发送给代理服务器，然后由代理服务器再发送给Web服务器，接着由代理服务器再把Web服务器返回的响应转发给本机。这样我们同样可以正常访问网页，但这个过程中Web服务器识别出的真实IP就不再是我们本机的IP了，就成功实现了IP伪装，这就是代理的基本原理。

3. 爬虫代理

对于爬虫来说，由于爬虫爬取速度过快，在爬取过程中可能遇到同一个IP访问过于频繁的问题，此时网站就会让我们输入验证码登录或者直接封锁IP，这样会给爬取带来极大的不便。

使用代理隐藏真实的IP，让服务器误以为是代理服务器在请求自己。这样在爬取过程中通过不断更换代理，就不会被封锁，可以达到很好的爬取效果。

4. 代理分类

代理分类时，既可以根据协议区分，也可以根据其匿名程度区分。

(1) 根据协议区分

根据代理的协议，代理可以分为如下类别。

FTP代理服务器：主要用于访问FTP服务器，一般有上传、下载以及缓存功能，端口一般为21、2121等。

HTTP代理服务器：主要用于访问网页，一般有内容过滤和缓存功能，端口一般为80、8080、3128等。

SSL/TLS代理：主要用于访问加密网站，一般有SSL或TLS加密功能（最高支持128位加密强度），端口一般为443。

RTSP代理：主要用于访问Real流媒体服务器，一般有缓存功能，端口一般为554。

Telnet代理：主要用于telnet远程控制（黑客入侵计算机时常用于隐藏身份），端口一般为23。

POP3/SMTP代理：主要用于POP3/SMTP方式收发邮件，一般有缓存功能，端口一般为110/25。

SOCKS代理：只是单纯传递数据包，不关心具体协议和用法，所以速度快很多，一般有缓存功能，端口一般为1080。SOCKS代理协议又分为SOCKS4和SOCKS5，前者只支持TCP，而后者支持TCP和UDP，还支持各种身份验证机制、服务器端域名解析等。简单来说，SOCK4能做到的SOCKS5都可以做到，但SOCKS5能做到的SOCK4不一定能做到。

(2) 根据匿名程度区分

根据代理的匿名程度，代理可以分为如下类别。

高度匿名代理：会将数据包原封不动地转发，在服务端看来就好像真的是一个普通客户端在访问，而记录的IP是代理服务器的IP。

普通匿名代理：会在数据包上做一些改动，服务端上有可能发现这是个代理服务器，也有一定几率追查到客户端的真实IP。代理服务器通常会加入的HTTP头有HTTP\_VIA和HTTP\_X\_FORWARDED\_FOR。

透明代理：不但改动了数据包，还会告诉服务器客户端的真实IP。这种代理除了能用缓存技术提高浏览速度，能用内容过滤提高安全性之外，并无其他显著作用，最常见的例子是内网中的硬件防火墙。

间谍代理：指组织或个人创建的用于记录用户传输的数据，然后进行研究、监控等目的的代理服务器。

首先，了解一下urllib库，它是Python内置的HTTP请求库，也就是说不需要额外安装即可使用。它包含如下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。

1. urlopen()

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，同时它还带有处理授权验证（authenticaton）、重定向（redirection)、浏览器Cookies以及其他内容。

下面我们来看一下它的强大之处。这里以Python官网为例，我们来把这个网页抓下来：

|  |  |
| --- | --- |
| 1  2  3  4 | import urllib.request    response = urllib.request.urlopen('https://www.python.org')  print(response.read().decode('utf-8')) |

|  |
| --- |
| print(type(response)) |

输出结果如下：

|  |  |
| --- | --- |
| 1 | <class 'http.client.HTTPResponse'> |

可以发现，它是一个HTTPResposne类型的对象。它主要包含read()、readinto()、getheader(name)、getheaders()、fileno()等方法，以及msg、version、status、reason、debuglevel、closed等属性。

得到这个对象之后，我们把它赋值为response变量，然后就可以调用这些方法和属性，得到返回结果的一系列信息了。

例如，调用read()方法可以得到返回的网页内容，调用status属性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。

|  |  |
| --- | --- |
| 2  3  4  5  6 | 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：

|  |  |
| --- | --- |
| 1  2  3 | 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'), ('Strict-Transport-Security', 'max-age=63072000; includeSubDomains')]  nginx |

可见，前两个输出分别输出了响应的状态码和响应的头信息，最后一个输出通过调用getheader()方法并传递一个参数Server获取了响应头中的Server值，结果是nginx，意思是服务器是用Nginx搭建的。

利用最基本的urlopen()方法，可以完成最基本的简单网页的GET请求抓取。

如果想给链接传递一些参数，该怎么实现呢？首先看一下urlopen()函数的API：

|  |  |
| --- | --- |
| 1 | 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，我们还可以传递其他内容，比如data（附加数据）、timeout（超时时间）等。

data参数

data参数是可选的。如果要添加该参数，并且如果它是字节流编码格式的内容，即bytes类型，则需要通过bytes()方法转化。另外，如果传递了这个参数，则它的请求方式就不再是GET方式，而是POST方式。

下面用实例来看一下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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()) |

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，该方法的第一个参数需要是str（字符串）类型，需要用urllib.parse模块里的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，这里指定为utf8。

timeout参数

timeout参数用于设置超时时间，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。如果不指定该参数，就会使用全局默认时间。它支持HTTP、HTTPS、FTP请求。

下面用实例来看一下：

|  |  |
| --- | --- |
| 1  2  3  4 | import urllib.request    response = urllib.request.urlopen('http://httpbin.org/get', timeout=1)  print(response.read()) |

运行结果如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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> |

这里我们设置超时时间是1秒。程序1秒过后，服务器依然没有响应，于是抛出了URLError异常。该异常属于urllib.error模块，错误原因是超时。

因此，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。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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') |

这里我们请求了[http://httpbin.org/get](http://httpbin.org/get" \o ")测试链接，设置超时时间是0.1秒，然后捕获了URLError异常，接着判断异常是socket.timeout类型（意思就是超时异常），从而得出它确实是因为超时而报错，打印输出了TIME OUT。

运行结果如下：

|  |  |
| --- | --- |
| 1 | TIME OUT |

按照常理来说，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。

通过设置timeout这个参数来实现超时处理，有时还是很有用的。

除了data参数和timeout参数外，还有context参数，它必须是ssl.SSLContext类型，用来指定SSL设置。

此外，cafile和capath这两个参数分别指定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，其默认值为False。

前面讲解了urlopen()方法的用法，通过这个最基本的方法，我们可以完成简单的请求和网页抓取。若需更加详细的信息，可以参见官方文档：[https://docs.python.org/3/library/urllib.request.html](https://docs.python.org/3/library/urllib.request.html" \o ")。

我们知道利用urlopen()方法可以实现最基本请求的发起，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，就可以利用更强大的Request类来构建。

首先，我们用实例来感受一下Request的用法：

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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，只不过这次该方法的参数不再是URL，而是一个Request类型的对象。通过构造这个数据结构，一方面我们可以将请求独立成一个对象，另一方面可更加丰富和灵活地配置参数。

下面我们看一下Request可以通过怎样的参数来构造，它的构造方法如下：

|  |  |
| --- | --- |
| 1 | **class** 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。

添加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，我们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：

|  |  |
| --- | --- |
| 1 | Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11 |

* 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。

下面我们传入多个参数构建请求来看一下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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')) |

另外，headers也可以用add\_header()方法来添加：

|  |  |
| --- | --- |
| 1  2 | req = request.Request(url=url, data=data, method='POST')  req.add\_header('User-Agent', 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)') |

更强大的工具Handler登场了。简而言之，我们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HTTP请求中所有的事情。

首先，介绍一下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，例如default\_open()、protocol\_request()等。

接下来，就有各种Handler子类继承这个BaseHandler类，举例如下。

* **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。

另外，还有其他的Handler类，这里就不一一列举了，详情可以参考官方文档：[https://docs.python.org/3/library/urllib.request.html#urllib.request.BaseHandler](https://docs.python.org/3/library/urllib.request.html" \l "urllib.request.BaseHandler" \o ")。

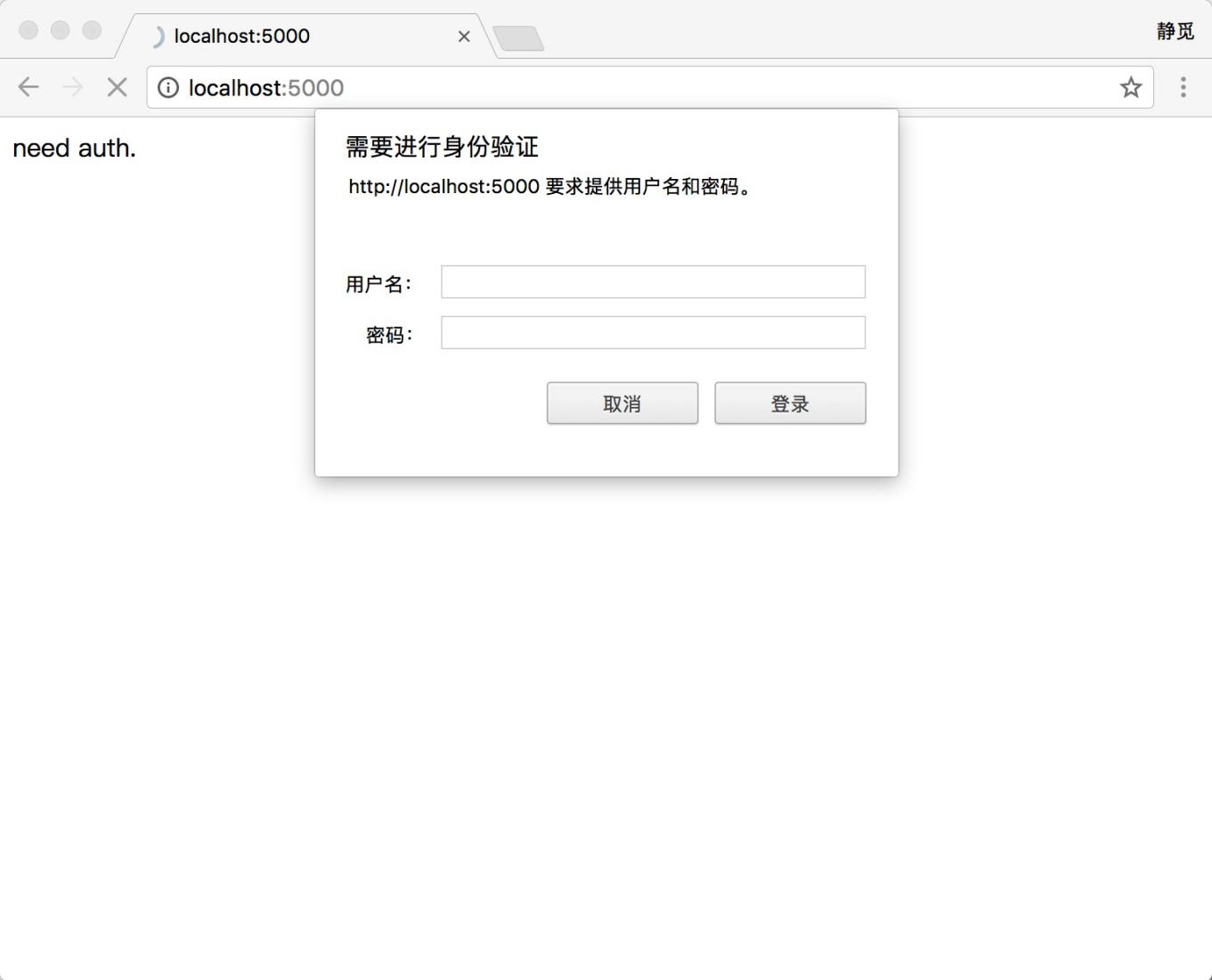
另一个比较重要的类就是OpenerDirector，我们可以称为Opener。我们之前用过urlopen()这个方法，实际上它就是urllib为我们提供的一个Opener。

那么，为什么要引入Opener呢？因为需要实现更高级的功能。之前使用的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，但是现在不一样了，我们需要实现更高级的功能，所以需要深入一层进行配置，使用更底层的实例来完成操作，所以这里就用到了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有什么关系呢？简而言之，就是利用Handler来构建Opener。

### 验证

有些网站在打开时就会弹出提示框，直接提示你输入用户名和密码，验证成功后才能查看页面，如图3-2所示。

图3-2 验证页面

那么，如果要请求这样的页面，该怎么办呢？借助HTTPBasicAuthHandler就可以完成，相关代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | from urllib.request import HTTPPasswordMgrWithDefaultRealm, HTTPBasicAuthHandler, build\_opener  from urllib.error import URLError    username = 'username'  password = 'password'  url = 'http://localhost:5000/'    p = HTTPPasswordMgrWithDefaultRealm()  p.add\_password(None, url, username, password)  auth\_handler = HTTPBasicAuthHandler(p)  opener = build\_opener(auth\_handler)    **try**:      result = opener.open(url)      html = result.read().decode('utf-8')      print(html)  except URLError **as** e:      print(e.reason) |

这里首先实例化HTTPBasicAuthHandler对象，其参数是

HTTPPasswordMgrWithDefaultRealm对象，它利用add\_password()添加进去用户名和密码，这样就建立了一个处理验证的Handler。

接下来，利用这个Handler并使用build\_opener()方法构建一个Opener，这个Opener在发送请求时就相当于已经验证成功了。

接下来，利用Opener的open()方法打开链接，就可以完成验证了。这里获取到的结果就是验证后的页面源码内容。

### 代理

在做爬虫的时候，免不了要使用代理，如果要添加代理，可以这样做：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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) |

这里我们在本地搭建了一个代理，它运行在9743端口上。

这里使用了ProxyHandler，其参数是一个字典，键名是协议类型（比如HTTP或者HTTPS等），键值是代理链接，可以添加多个代理。

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

### Cookies

**Cookies的处理就需要相关的Handler了。**

**我们先用实例来看看怎样将网站的Cookies获取下来，相关代码如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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，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()函数即可。**

**运行结果如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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**) |

**这时CookieJar就需要换成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，比如读取和保存Cookies，可以将Cookies保存成Mozilla型浏览器的Cookies格式。**

**运行之后，可以发现生成了一个cookies.txt文件，其内容如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | # 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    05AE39B5F56C1DEC474325CDA522D44F:FG=1  .baidu.com    **TRUE**    /    **FALSE**    3622386254    BIDUPSID    05AE39B5F56C1DEC474325CDA522D44F  .baidu.com    **TRUE**    /    **FALSE**        H\_PS\_PSSID    19638\_1453\_17710\_18240\_21091\_18560\_17001\_21191\_21161  .baidu.com    **TRUE**    /    **FALSE**    3622386254    PSTM    1474902606  www.baidu.com    **FALSE**    /    **FALSE**        BDSVRTM    0  www.baidu.com    **FALSE**    /    **FALSE**        BD\_HOME    0 |

**另外，LWPCookieJar同样可以读取和保存Cookies，但是保存的格式和MozillaCookieJar不一样，它会保存成libwww-perl(LWP)格式的Cookies文件。**

**要保存成LWP格式的Cookies文件，可以在声明时就改为：**

|  |  |
| --- | --- |
| 1 | cookie = http.cookiejar.LWPCookieJar(filename) |

**此时生成的内容如下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #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 |

**那么，生成了Cookies文件后，怎样从文件中读取并利用呢？**

**下面我们以LWPCookieJar格式为例来看一下：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | cookie = http.cookiejar.LWPCookieJar()  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')) |

**可以看到，这里调用load()方法来读取本地的Cookies文件，获取到了Cookies的内容。不过前提是我们首先生成了LWPCookieJar格式的Cookies，并保存成文件，然后读取Cookies之后使用同样的方法构建Handler和Opener即可完成操作。**

**运行结果正常的话，会输出百度网页的源代码。**

**通过上面的方法，我们可以实现绝大多数请求功能的设置了。**

**这便是urllib库中request模块的基本用法，如果想实现更多的功能，可以参考官方文档的说明：[https://docs.python.org/3/library/urllib.request.html#basehandler-objects](https://docs.python.org/3/library/urllib.request.html" \l "basehandler-objects" \o ")。**

## 1. URLError

URLError类来自urllib库的error模块，它继承自OSError类，是error异常模块的基类，由request模块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。

下面用一个实例来看一下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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，照理来说应该会报错，但是这时我们捕获了URLError这个异常，运行结果如下：

|  |  |
| --- | --- |
| 1 | **Not** Found |

程序没有直接报错，而是输出了如上内容，这样通过如上操作，我们就可以避免程序异常终止，同时异常得到了有效处理。

## 2. HTTPError

它是URLError的子类，专门用来处理HTTP请求错误，比如认证请求失败等。它有如下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。

下面我们用几个实例来看看：

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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') |

运行结果如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **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，这里捕获了HTTPError异常，输出了reason、code和headers属性。

因为URLError是HTTPError的父类，所以可以先选择捕获子类的错误，再去捕获父类的错误，所以上述代码更好的写法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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') |

这样就可以做到先捕获HTTPError，获取它的错误状态码、原因、headers等信息。如果不是HTTPError异常，就会捕获URLError异常，输出错误原因。最后，用else来处理正常的逻辑。这是一个较好的异常处理写法。

有时候，reason属性返回的不一定是字符串，也可能是一个对象。再看下面的实例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import socket  import urllib.request  import urllib.error    **try**:      response = urllib.request.urlopen('https://www.baidu.com', timeout=0.01)  except urllib.error.URLError **as** e:      print(type(e.reason))  **if** isinstance(e.reason, socket.timeout):          print('TIME OUT') |

这里我们直接设置超时时间来强制抛出timeout异常。

运行结果如下：

|  |  |
| --- | --- |
| 1  2 | <**class** 'socket.timeout'>  TIME OUT |

可以发现，reason属性的结果是socket.timeout类。所以，这里我们可以用isinstance()方法来判断它的类型，作出更详细的异常判断。

本节中，我们讲述了error模块的相关用法，通过合理地捕获异常可以做出更准确的异常判断，使程序更加稳健。

## 1. urlparse()

该方法可以实现URL的识别和分段，这里先用一个实例来看一下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import urlparse    result = urlparse('http://www.baidu.com/index.html;user?id=5#comment')  print(type(result), result) |

这里我们利用urlparse()方法进行了一个URL的解析。首先，输出了解析结果的类型，然后将结果也输出出来。

运行结果如下：

|  |  |
| --- | --- |
| 1  2 | <**class** 'urllib.parse.ParseResult'>  ParseResult(scheme='http', netloc='www.baidu.com', path='/index.html', params='user', query='id=5', fragment='comment') |

可以看到，返回结果是一个ParseResult类型的对象，它包含6部分，分别是scheme、netloc、path、params、query和fragment。

观察一下该实例的URL：

|  |  |
| --- | --- |
| 1 | http:*//www.baidu.com/index.html;user?id=5#comment* |

可以发现，urlparse()方法将其拆分成了6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，代表协议；第一个/前面便是netloc，即域名；分号;前面是params，代表参数。

所以，可以得出一个标准的链接格式，具体如下：

|  |  |
| --- | --- |
| 1 | scheme:*//netloc/path;parameters?query#fragment* |

一个标准的URL都会符合这个规则，利用urlparse()方法可以将它拆分开来。

除了这种最基本的解析方式外，urlparse()方法还有其他配置吗？接下来，看一下它的API用法：

|  |  |
| --- | --- |
| 1 | urllib.parse.urlparse(urlstring, scheme='', allow\_fragments=**True**) |

可以看到，它有3个参数。

* **urlstring**：这是必填项，即待解析的URL。
* **scheme**：它是默认的协议（比如http或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。我们用实例来看一下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import urlparse    result = urlparse('www.baidu.com/index.html;user?id=5#comment', scheme='https')  print(result) |

运行结果如下：

|  |  |
| --- | --- |
| 1 | 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URL没有包含最前面的scheme信息，但是通过指定默认的scheme参数，返回的结果是https。

假设我们带上了scheme：

|  |  |
| --- | --- |
| 1 | result = urlparse('http://www.baidu.com/index.html;user?id=5#comment', scheme='https') |

则结果如下：

|  |  |
| --- | --- |
| 1 | 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URL中不包含scheme信息时才生效。如果URL中有scheme信息，就会返回解析出的scheme。

* **allow\_fragments**：即是否忽略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，而fragment部分为空。下面我们用实例来看一下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import urlparse    result = urlparse('http://www.baidu.com/index.html;user?id=5#comment', allow\_fragments=**False**)  print(result) |

运行结果如下：

|  |  |
| --- | --- |
| 1 | ParseResult(scheme='http', netloc='www.baidu.com', path='/index.html', params='user', query='id=5#comment', fragment='') |

假设URL中不包含params和query，我们再通过实例看一下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import urlparse    result = urlparse('http://www.baidu.com/index.html#comment', allow\_fragments=**False**)  print(result) |

运行结果如下：

|  |  |
| --- | --- |
| 1 | ParseResult(scheme='http', netloc='www.baidu.com', path='/index.html#comment', params='', query='', fragment='') |

可以发现，当URL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，我们可以用索引顺序来获取，也可以用属性名获取。示例如下：

|  |  |
| --- | --- |
| 1  2  3  4 | 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') |

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

|  |  |
| --- | --- |
| 1  2  3  4 | http  http  www.baidu.com  www.baidu.com |

可以发现，二者的结果是一致的，两种方法都可以成功获取。

## 2. urlunparse()

有了urlparse()，相应地就有了它的对立方法urlunparse()。它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。先用一个实例看一下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import urlunparse    data = ['http', 'www.baidu.com', 'index.html', 'user', 'a=6', 'comment']  print(urlunparse(data)) |

这里参数data用了列表类型。当然，你也可以用其他类型，比如元组或者特定的数据结构。

运行结果如下：

|  |  |
| --- | --- |
| 1 | http:*//www.baidu.com/index.html;user?a=6#comment* |

这样我们就成功实现了URL的构造。

## 3. urlsplit()

这个方法和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。上面例子中的params会合并到path中。示例如下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import urlsplit    result = urlsplit('http://www.baidu.com/index.html;user?id=5#comment')  print(result) |

运行结果如下：

|  |  |
| --- | --- |
| 1 | SplitResult(scheme='http', netloc='www.baidu.com', path='/index.html;user', query='id=5', fragment='comment') |

可以发现，返回结果是SplitResult，它其实也是一个元组类型，既可以用属性获取值，也可以用索引来获取。示例如下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import urlsplit    result = urlsplit('http://www.baidu.com/index.html;user?id=5#comment')  print(result.scheme, result[0]) |

运行结果如下：

|  |  |
| --- | --- |
| 1 | http http |

## 4. urlunsplit()

与urlunparse()类似，它也是将链接各个部分组合成完整链接的方法，传入的参数也是一个可迭代对象，例如列表、元组等，唯一的区别是长度必须为5。示例如下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import urlunsplit    data = ['http', 'www.baidu.com', 'index.html', 'a=6', 'comment']  print(urlunsplit(data)) |

运行结果如下：

|  |  |
| --- | --- |
| 1 | http:*//www.baidu.com/index.html?a=6#comment* |

## 5. 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。

此外，生成链接还有另一个方法，那就是urljoin()方法。我们可以提供一个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。

下面通过几个实例看一下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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')) |

运行结果如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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和path。如果这3项在新的链接里不存在，就予以补充；如果新的链接存在，就使用新的链接的部分。而base\_url中的params、query和fragment是不起作用的。

通过urljoin()方法，我们可以轻松实现链接的解析、拼合与生成。

## 6. urlencode()

这里我们再介绍一个常用的方法——urlencode()，它在构造GET请求参数的时候非常有用，示例如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from urllib.parse import urlencode    params = {      'name': 'germey',      'age': 22  }  base\_url = 'http://www.baidu.com?'  url = base\_url + urlencode(params)  print(url) |

这里首先声明了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。

运行结果如下：

|  |  |
| --- | --- |
| 1 | http:*//www.baidu.com?name=germey&age=22* |

可以看到，参数就成功地由字典类型转化为GET请求参数了。

这个方法非常常用。有时为了更加方便地构造参数，我们会事先用字典来表示。要转化为URL的参数时，只需要调用该方法即可。

## 7. parse\_qs()

有了序列化，必然就有反序列化。如果我们有一串GET请求参数，利用parse\_qs()方法，就可以将它转回字典，示例如下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import parse\_qs    query = 'name=germey&age=22'  print(parse\_qs(query)) |

运行结果如下：

|  |  |
| --- | --- |
| 1 | {'name': ['germey'], 'age': ['22']} |

可以看到，这样就成功转回为字典类型了。

## 8. parse\_qsl()

另外，还有一个parse\_qsl()方法，它用于将参数转化为元组组成的列表，示例如下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import parse\_qsl    query = 'name=germey&age=22'  print(parse\_qsl(query)) |

运行结果如下：

|  |  |
| --- | --- |
| 1 | [('name', 'germey'), ('age', '22')] |

可以看到，运行结果是一个列表，而列表中的每一个元素都是一个元组，元组的第一个内容是参数名，第二个内容是参数值。

## 9. quote()

该方法可以将内容转化为URL编码的格式。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，示例如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | from urllib.parse import quote    keyword = '壁纸'  url = 'https://www.baidu.com/s?wd=' + quote(keyword)  print(url) |

这里我们声明了一个中文的搜索文字，然后用quote()方法对其进行URL编码，最后得到的结果如下：

|  |  |
| --- | --- |
| 1 | https:*//www.baidu.com/s?wd=%E5%A3%81%E7%BA%B8* |

## 10. unquote()

有了quote()方法，当然还有unquote()方法，它可以进行URL解码，示例如下：

|  |  |
| --- | --- |
| 1  2  3  4 | from urllib.parse import unquote    url = 'https://www.baidu.com/s?wd=%E5%A3%81%E7%BA%B8'  print(unquote(url)) |

这是上面得到的URL编码后的结果，这里利用unquote()方法还原，结果如下：

|  |  |
| --- | --- |
| 1 | https:*//www.baidu.com/s?wd=壁纸* |

可以看到，利用unquote()方法可以方便地实现解码。

本节中，我们介绍了parse模块的一些常用URL处理方法。有了这些方法，我们可以方便地实现URL的解析和构造，建议熟练掌握。