

---

# 如何利用 urllib 包获取网络资源

发行版本 3.12.11

Guido van Rossum and the Python development team

九月 29, 2025

Python Software Foundation  
Email: docs@python.org

## Contents

1 概述	1
2 获取 URL 资源	2
2.1 数据	3
2.2 HTTP 头部信息	3
3 异常的处理	4
3.1 URLError	4
3.2 HTTPError	4
3.3 总之	6
4 info 和 geturl 方法	7
5 Opener 和 Handler	7
6 基本认证	7
7 代理	8
8 套接字与分层	9
9 备注	9
索引	10

---

作者

Michael Foord

## 1 概述

### Related Articles

关于如何用 Python 获取 web 资源，以下文章或许也很有用：

- 基本身份认证

基本认证的教程，带有一些 Python 示例。

`urllib.request` 是用于获取 URL（统一资源定位符）的 Python 模块。它以 `urlopen` 函数的形式提供了一个非常简单的接口，能用不同的协议获取 URL。同时它还还为处理各种常见情形提供了一个稍微复杂一些的接口——比如：基础身份认证、cookies、代理等等。这些功能是由名为 `handlers` 和 `opener` 的对象提供的。

`urllib.request` 支持多种“URL 方案”（通过 URL 中 “:” 之前的字符串加以区分——如 “`ftp://python.org/`” 中的 “`ftp`”）即为采用其关联网络协议（FTP、HTTP 之类）的 URL 方案。本教程重点关注最常用的 HTTP 场景。

对于简单场景而言，`urlopen` 用起来十分容易。但只要在打开 HTTP URL 时遇到错误或非常情况，就需要对超文本传输协议有所了解才行。最全面、最权威的 HTTP 参考是 [RFC 2616](#)。那是一份技术文档，并没有追求可读性。本文旨在说明 `urllib` 的用法，为了便于阅读也附带了足够详细的 HTTP 信息。本文并不是为了替代 `urllib.request` 文档，只是其补充说明而已。

## 2 获取 URL 资源

`urllib.request` 最简单的使用方式如下所示：

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
    html = response.read()
```

如果想通过 URL 获取资源并临时存储一下，可以采用 `shutil.copyfileobj()` 和 `tempfile.NamedTemporaryFile()` 函数：

```
import shutil
import tempfile
import urllib.request

with urllib.request.urlopen('http://python.org/') as response:
    with tempfile.NamedTemporaryFile(delete=False) as tmp_file:
        shutil.copyfileobj(response, tmp_file)

with open(tmp_file.name) as html:
    pass
```

`urllib` 的很多用法就是这么简单（注意 URL 不仅可以 `http:` 开头，还可以是 `ftp:`、`file:` 等）。不过本教程的目的是介绍更加复杂的应用场景，重点还是关注 HTTP。

HTTP 以请求和响应为基础——客户端生成请求，服务器发送响应。`urllib.request` 用 `Request` 对象来表示要生成的 HTTP 请求。最简单的形式就是创建一个 `Request` 对象，指定了想要获取的 URL。用这个 `Request` 对象作为参数调用 `urlopen`，将会返回该 URL 的响应对象。响应对象类似于文件对象，就是说可以对其调用 `.read()` 之类的命令：

```
import urllib.request

req = urllib.request.Request('http://python.org/')
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

请注意，`urllib.request` 用同一个 `Request` 接口处理所有 URL 方案。比如可生成 FTP 请求如下：

```
req = urllib.request.Request('ftp://example.com/')
```

就 HTTP 而言，`Request` 对象能够做两件额外的事情：首先可以把数据传给服务器。其次，可以将有关数据或请求本身的额外信息（`metadata`）传给服务器——这些信息将会作为 HTTP “头部” 数据发送。下面依次看下。

## 2.1 数据

有时需要向某个 URL 发送数据，通常此 URL 会指向某个 CGI（通用网关接口）脚本或其他 web 应用。对于 HTTP 而言，这通常会用所谓的 **POST** 请求来完成。当要把 Web 页填写的 HTML 表单提交时，浏览器通常会执行此操作。但并不是所有的 POST 都来自表单：可以用 POST 方式传输任何数据到自己的应用上。对于通常的 HTML 表单，数据需要以标准的方式编码，然后作为 data 参数传给 Request 对象。编码过程是用 urllib.parse 库的函数完成的：

```
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.parse.urlencode(values)
data = data.encode('ascii') # 数据应为字节串
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

请注意，有时还需要采用其他编码，比如由 HTML 表单上传文件——更多细节请参见 [HTML 规范](#)，提交表单。

如果不传递 data 参数，urllib 将采用 **GET** 请求。GET 和 POST 请求有一点不同，POST 请求往往具有“副作用”，他们会以某种方式改变系统的状态。例如，从网站下一个订单，购买一大堆罐装垃圾并运送到家。尽管 HTTP 标准明确指出 POST 总是要导致副作用，而 GET 请求从来不会导致副作用。但没有什么办法能阻止 GET 和 POST 请求的副作用。数据也可以在 HTTP GET 请求中传递，只要把数据编码到 URL 中即可。

做法如下所示：

```
>>> import urllib.request
>>> import urllib.parse
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
>>> data['language'] = 'Python'
>>> url_values = urllib.parse.urlencode(data)
>>> print(url_values) # The order may differ from below.
name=Somebody+Here&language=Python&location=Northampton
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib.request.urlopen(full_url)
```

请注意，完整的 URL 是通过在其中添加 ? 创建的，后面跟着经过编码的数据。

## 2.2 HTTP 头部信息

下面介绍一个具体的 HTTP 头部信息，以此说明如何在 HTTP 请求加入头部信息。

有些网站<sup>1</sup>不愿被程序浏览到，或者要向不同的浏览器发送不同版本<sup>2</sup>的网页。默认情况下，urllib 将自身标识为“Python-urllib/xy”（其中 x、y 是 Python 版本的主、次版本号，例如 Python-urllib/2.5），这可能会让网站不知所措，或者干脆就使其无法正常工作。浏览器是通过头部信息 User-Agent<sup>3</sup>来标识自己的。在创建 Request 对象时，可以传入字典形式的头部信息。以下示例将生成与之前相同的请求，只是将自身标识为某个版本的 Internet Explorer<sup>4</sup>：

<sup>1</sup> 例如 Google。

<sup>2</sup> 对于网站设计而言，探测不同的浏览器是非常糟糕的做法——更为明智的做法是采用 web 标准构建网站。不幸的是，很多网站依然向不同的浏览器发送不同版本的网页。

<sup>3</sup> MSIE 6 的 user-agent 信息是 “Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)”

<sup>4</sup> 有关 HTTP 请求的头部信息，详情请参阅 [Quick Reference to HTTP Headers](#)。

```
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)'
values = {'name': 'Michael Foord',
          'location': 'Northampton',
          'language': 'Python' }
headers = {'User-Agent': user_agent}

data = urllib.parse.urlencode(values)
data = data.encode('ascii')
req = urllib.request.Request(url, data, headers)
with urllib.request.urlopen(req) as response:
    the_page = response.read()
```

响应对象也有两个很有用的方法。请参阅有关[info](#) 和 [geturl](#) 部分，了解出现问题时会发生什么。

## 3 异常的处理

当 `urlopen` 无法处理响应信息时将会引发 `URLError` (当然与 Python API 通常的情况一样，也可能会引发如 `ValueError`, `TypeError` 之类的内置异常)。

`HTTPError` 是在 HTTP URL 的特定情况下引发的 `URLError` 的子类。

上述异常类是从 `urllib.error` 模块中导出的。

### 3.1 URLError

触发 `URLError` 的原因，通常是网络不通（或者没有到指定服务器的路由），或者指定的服务器不存在。这时触发的异常会带有一个 `reason` 属性，是一个包含错误代码和文本错误信息的元组。

例如：

```
>>> req = urllib.request.Request('http://www.pretend_server.org')
>>> try: urllib.request.urlopen(req)
... except urllib.error.URLError as e:
...     print(e.reason)
...
(4, 'getaddrinfo failed')
```

### 3.2 HTTPError

来自服务器的每个 HTTP 响应都包含一个数字形式的“状态码”。有时该状态码表明服务器无法完成请求。默认的处理程序将会为你处理其中的部分响应（例如，当响应为要求客户端从另一 URL 获取文档的“重定向”响应时，`urllib` 将为你处理该响应）。对于无法处理的响应，`urlopen` 将会引发 `HTTPError`。典型的错误包括“404”（页面未找到）、“403”（请求遭拒）和“401”（需要身份认证）等。

全部的 HTTP 错误码请参阅 [RFC 2616](#)。

被引发的 `HTTPError` 实例将有一个整数形式的 `'code'` 属性，对应于服务器发送的错误信息。

#### 错误代码

由于默认处理函数会自行处理重定向（300 以内的错误码），而且 100--299 的状态码表示成功，因此通常只会出现 400--599 的错误码。

`http.server.BaseHTTPRequestHandler.responses` 是很有用的响应码字典，其中给出了 [RFC 2616](#) 用到的所有响应代码。为方便起见，将此字典转载如下：

```

# 响应代码到消息的映射表；条目的形式为
# {代码: (简短消息, 详细消息)}。
responses = {
    100: ('Continue', 'Request received, please continue'),
    101: ('Switching Protocols',
        'Switching to new protocol; obey Upgrade header'),

    200: ('OK', 'Request fulfilled, document follows'),
    201: ('Created', 'Document created, URL follows'),
    202: ('Accepted',
        'Request accepted, processing continues off-line'),
    203: ('Non-Authoritative Information', 'Request fulfilled from cache'),
    204: ('No Content', 'Request fulfilled, nothing follows'),
    205: ('Reset Content', 'Clear input form for further input.'),
    206: ('Partial Content', 'Partial content follows.'),

    300: ('Multiple Choices',
        'Object has several resources -- see URI list'),
    301: ('Moved Permanently', 'Object moved permanently -- see URI list'),
    302: ('Found', 'Object moved temporarily -- see URI list'),
    303: ('See Other', 'Object moved -- see Method and URL list'),
    304: ('Not Modified',
        'Document has not changed since given time'),
    305: ('Use Proxy',
        'You must use proxy specified in Location to access this '
        'resource.'),
    307: ('Temporary Redirect',
        'Object moved temporarily -- see URI list'),

    400: ('Bad Request',
        'Bad request syntax or unsupported method'),
    401: ('Unauthorized',
        'No permission -- see authorization schemes'),
    402: ('Payment Required',
        'No payment -- see charging schemes'),
    403: ('Forbidden',
        'Request forbidden -- authorization will not help'),
    404: ('Not Found', 'Nothing matches the given URI'),
    405: ('Method Not Allowed',
        'Specified method is invalid for this server.'),
    406: ('Not Acceptable', 'URI not available in preferred format.'),
    407: ('Proxy Authentication Required', 'You must authenticate with '
        'this proxy before proceeding.'),
    408: ('Request Timeout', 'Request timed out; try again later.'),
    409: ('Conflict', 'Request conflict.'),
    410: ('Gone',
        'URI no longer exists and has been permanently removed.'),
    411: ('Length Required', 'Client must specify Content-Length.'),
    412: ('Precondition Failed', 'Precondition in headers is false.'),
    413: ('Request Entity Too Large', 'Entity is too large.'),
    414: ('Request-URI Too Long', 'URI is too long.'),
    415: ('Unsupported Media Type', 'Entity body in unsupported format.'),
    416: ('Requested Range Not Satisfiable',
        'Cannot satisfy request range.'),
    417: ('Expectation Failed',
        'Expect condition could not be satisfied.'),

    500: ('Internal Server Error', 'Server got itself in trouble'),
    501: ('Not Implemented',
        'Server does not support this operation'),
    502: ('Bad Gateway', 'Invalid responses from another server/proxy.'),
    503: ('Service Unavailable',

```

(续下页)

(接上页)

```
'The server cannot process the request due to a high load'),
504: ('Gateway Timeout',
      'The gateway server did not receive a timely response'),
505: ('HTTP Version Not Supported', 'Cannot fulfill request.'),
}
```

当错误被引发时服务器会通过返回 HTTP 错误码 和 错误页面进行响应。你可以在返回的页面上使用 `HTTPError` 实例作为响应。这意味着除了 `code` 属性之外，它还像 `urllib.response` 模块: 所返回对象那样具有 `read`, `geturl` 和 `info` 等方法:

```
>>> req = urllib.request.Request('http://www.python.org/fish.html')
>>> try:
...     urllib.request.urlopen(req)
... except urllib.error.HTTPError as e:
...     print(e.code)
...     print(e.read())
...
404
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n<html
...
<title>Page Not Found</title>\n
...
'
```

### 3.3 总之

因此当你想为 `HTTPError` 或 `URLError` 做好准备时有两种基本的方案。我更倾向使用第二种方案。

#### 第一种方案

```
from urllib.request import Request, urlopen
from urllib.error import URLError, HTTPError
req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print('The server couldn\'t fulfill the request.')
    print('Error code: ', e.code)
except URLError as e:
    print('We failed to reach a server.')
    print('Reason: ', e.reason)
else:
    # 一切正常
```

#### 备注

`except HTTPError` 必须首先被处理，否则 `except URLError` 将会同时捕获 `HTTPError`。

#### 第二种方案

```
from urllib.request import Request, urlopen
from urllib.error import URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
```

(续下页)

```

    print('We failed to reach a server.')
    print('Reason: ', e.reason)
    elif hasattr(e, 'code'):
        print('The server couldn\'t fulfill the request.')
        print('Error code: ', e.code)
else:
    # 一切正常

```

## 4 info 和 geturl 方法

`urlopen` 返回的响应（或 `HTTPError` 实例）包含两个有用的方法 `info()` 和 `geturl()` 并且是在 `urllib.response` 模块中定义的。

- **geturl** ——返回所获取页面的真实 URL。该方法很有用，因为 `urlopen`（或 `opener` 对象）可能已经经过了一次重定向。已获取页面的 URL 未必就是所请求的 URL。
- **info** - 该方法返回一个类似字典的对象，描述了所获取的页面，特别是由服务器送出的头部信息（headers）。目前它是一个 `http.client.HTTPMessage` 实例。

典型的标头包括 'Content-length', 'Content-type' 等等。请参阅 [HTTP 标头快速参考](#) 获取 HTTP 标头的完整列表及其含义和用法的简要说明。

## 5 Opener 和 Handler

当你获取 URL 时会使用一个 `opener`（名称可能有些令人困惑的 `urllib.request.OpenerDirector` 的实例）。通常我们会使用默认的 `opener` ——通过 `urlopen` ——但你也可以创建自定义的 `opener`。`opener` 还会用到 `handler`。所有“繁重工作”都是由 `handler` 来完成的。每种 `handler` 都知道要以何种 URL 方案（http, ftp 等等）来打开特定的 URL，或是如何处理 URL 打开时的特定操作，例如 HTTP 重定向或 HTTP cookie 等。

若要用已安装的某个 `handler` 获取 URL，需要创建一个 `opener` 对象，例如处理 cookie 的 `opener`，或对重定向不做处理的 `opener`。

若要创建 `opener`，请实例化一个 `OpenerDirector`，然后重复调用 `.add_handler(some_handler_instance)`。

或者也可以用 `build_opener`，这是个用单次调用创建 `opener` 对象的便捷函数。`build_opener` 默认会添加几个 `handler`，不过还提供了一种快速添加和/或覆盖默认 `handler` 的方法。

可能还需要其他类型的 `handler`，以便处理代理、身份认证和其他常见但稍微特殊的情况。

`install_opener` 可用于让 `opener` 对象成为（全局）默认 `opener`。这意味着调用 `urlopen` 时会采用已安装的 `opener`。

`opener` 对象带有一个 `.open` 方法，可供直接调用以获取 url，方式与 `urlopen` 函数相同。除非是为了调用方便，否则没必要去调用 `install_opener`。

## 6 基本认证

为了说明 `handler` 的创建和安装过程我们将使用 `HTTPBasicAuthHandler`。有关该主题的更详细讨论 -- 包括对基本身份认证的原理的阐述 -- 请参阅 [Basic Authentication Tutorial](#)。

如果需要身份认证，服务器会发送一条请求身份认证的头部信息（以及 401 错误代码）。这条信息中指明了身份认证方式和“安全区域（realm）”。格式如下所示：WWW-Authenticate: SCHEME realm="REALM"

例如

```
WWW-Authenticate: Basic realm="cPanel Users"
```



然后，客户端应重试发起请求，请求数据中的头部信息应包含安全区域对应的用户名和密码。这就是“基本身份认证”。为了简化此过程，可以创建 HTTPBasicAuthHandler 的一个实例及使用它的 opener。

HTTPBasicAuthHandler 用一个名为密码管理器的对象来管理 URL、安全区域与密码、用户名之间的映射关系。如果知道确切的安全区域（来自服务器发送的身份认证头部信息），那就可以用到 HTTPPasswordMgr。通常人们并不关心安全区域是什么，这时用 HTTPPasswordMgrWithDefaultRealm 就很方便，允许为 URL 指定默认的用户名和密码。当没有为某个安全区域提供用户名和密码时，就会用到默认值。下面用 None 作为 add\_password 方法的安全区域参数，表明采用默认用户名和密码。

首先需要身份认证的是顶级 URL。比传给 add\_password() 的 URL 级别“更深”的 URL 也会得以匹配：

```
# 创建一个密码管理器
password_mgr = urllib.request.HTTPPasswordMgrWithDefaultRealm()

# 添加用户名和密码。
# 如果我们知道域，可以用它代替 None。
top_level_url = "http://example.com/foo/"
password_mgr.add_password(None, top_level_url, username, password)

handler = urllib.request.HTTPBasicAuthHandler(password_mgr)

# 创建 "opener" (OpenerDirector 实例)
opener = urllib.request.build_opener(handler)

# 使用 opener 获取一个 URL
opener.open(a_url)

# 安装 opener。
# 现在所有对 urllib.request.urlopen 的调用都将使用此 opener。
urllib.request.install_opener(opener)
```

#### 备注

在上面的救命中我们只向 build\_opener 提供了 HTTPBasicAuthHandler。在默认情况下 opener 会包含针对常见状况的处理器 -- ProxyHandler (如果设置了代理如设置了 http\_proxy 环境变量), UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler, DataHandler, HTTPErrorProcessor。

top\_level\_url 其实要么是一条完整的 URL（包括“http:”部分和主机名及可选的端口号），比如“http://example.com/”，要么是一条“访问权限”（即主机名，及可选的端口号），比如“example.com”或“example.com:8080”（后一个示例包含了端口号）。访问权限不得包含“用户信息”部分——比如“joe:password@example.com”就不正确。

## 7 代理

urllib 将自动检测并使用代理设置。这是通过 ProxyHandler 实现的，当检测到代理设置时，是正常 handler 链中的一部分。通常这是一件好事，但有时也可能会无效<sup>5</sup>。一种方案是配置自己的 ProxyHandler，不要定义代理。设置的步骤与 Basic Authentication handler 类似：

```
>>> proxy_support = urllib.request.ProxyHandler({})
>>> opener = urllib.request.build_opener(proxy_support)
>>> urllib.request.install_opener(opener)
```

#### 备注

<sup>5</sup> 本人必须使用代理才能在工作中访问互联网。如果尝试通过代理获取 localhost URL，将会遭到阻止。IE 设置为代理模式，urllib 就会获取到配置信息。为了用 localhost 服务器测试脚本，我必须阻止 urllib 使用代理。



目前 `urllib.request` 尚不支持通过代理抓取 `https` 链接地址。但此功能可以通过扩展 `urllib.request` 来启用，如以下例程所示<sup>6</sup>。

#### 备注

如果设置了 `REQUEST_METHOD` 变量，则会忽略 `HTTP_PROXY`；参阅 `getproxies()` 文档。

## 8 套接字与分层

Python 获取 Web 资源的能力是分层的。`urllib` 用到的是 `http.client` 库，而后者又用到了套接字库。

从 Python 2.3 开始，可以指定套接字等待响应的超时时间。这对必须要读到网页数据的应用程序会很有用。默认情况下，套接字模块不会超时并且可以挂起。目前，套接字超时机制未暴露给 `http.client` 或 `urllib.request` 层使用。不过可以为所有套接字应用设置默认的全局超时。

```
import socket
import urllib.request

# 超时秒数
timeout = 10
socket.setdefaulttimeout(timeout)

# 这个对 urllib.request.urlopen 的调用现在将使用
# 我们在 socket 模块中设置的默认超时值
req = urllib.request.Request('http://www.voidspace.org.uk')
response = urllib.request.urlopen(req)
```

## 9 备注

这篇文档由 John Lee 审订。

---

<sup>6</sup> `urllib` 的 SSL 代理 opener (CONNECT 方法): [ASPEN Cookbook Recipe](#)。

## 索引

### R

RFC

RFC 2616, 2, 4