高端用法原出处

高级用法

本篇文档涵盖了 Requests 的一些高级特性。

会话对象

会话对象让你能够跨请求保持某些参数。它也会在同一个 Session 实例发出的所有请求之间保持 cookie， 期间使用 urllib3 的 connection pooling 功能。所以如果你向同一主机发送多个请求，底层的 TCP 连接将会被重用，从而带来显著的性能提升。 (参见 HTTP persistent connection).

会话对象具有主要的 Requests API 的所有方法。

我们来跨请求保持一些 cookie:

```
s = requests.Session()

s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
r = s.get("http://httpbin.org/cookies")

print(r.text)
# '{"cookies": {"sessioncookie": "123456789"}}'
```

会话也可用来为请求方法提供缺省数据。这是通过为会话对象的属性提供数据来实现的：

```
s = requests.Session()
s.auth = ('user', 'pass')
s.headers.update({'x-test': 'true'})

# both 'x-test' and 'x-test2' are sent
s.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
```

任何你传递给请求方法的字典都会与已设置会话层数据合并。方法层的参数覆盖会话的参数。

不过需要注意，就算使用了会话，方法级别的参数也不会被跨请求保持。下面的例子只会和第一个请求发送 cookie ，而非第二个：

```
s = requests.Session()

r = s.get('http://httpbin.org/cookies', cookies={'from-my': 'browser'})
print(r.text)
# '{"cookies": {"from-my": "browser"}}'

r = s.get('http://httpbin.org/cookies')
print(r.text)
# '{"cookies": {}}'
```

如果你要手动为会话添加 cookie，就是用 Cookie utility 函数 来操纵 **Session.cookies**。

会话还可以用作前后文管理器：

```
with requests.Session() as s:
    s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
```

这样就能确保 with 区块退出后会话能被关闭，即使发生了异常也一样。

从字典参数中移除一个值

有时你会想省略字典参数中一些会话层的键。要做到这一点，你只需简单地在方法层参数中将那个键的值设置为 None ，那个键就会被自动省略掉。

包含在一个会话中的所有数据你都可以直接使用。学习更多细节请阅读 会话 API 文档。

请求与响应对象

任何时候调用 requests.*() 你都在做两件主要的事情。其一，你在构建一个 *Request* 对象， 该对象将被发送到某

个服务器请求或查询一些资源。其二，一旦 requests 得到一个从 服务器返回的响应就会产生一个 Response 对象。该响应对象包含服务器返回的所有信息， 也包含你原来创建的 Request 对象。如下是一个简单的请求，从 Wikipedia 的服务器得到 一些非常重要的信息：

```
>>> r = requests.get('http://en.wikipedia.org/wiki/Monty_Python')
```

如果想访问服务器返回给我们的响应头部信息，可以这样做：

```
>>> r.headers
{'content-length': '56170', 'x-content-type-options': 'nosniff', 'x-cache':
'HIT from cp1006.eqiad.wmnet, MISS from cp1010.eqiad.wmnet', 'content-encoding':
'gzip', 'age': '3080', 'content-language': 'en', 'vary': 'Accept-Encoding,Cookie',
'server': 'Apache', 'last-modified': 'Wed, 13 Jun 2012 01:33:50 GMT',
'connection': 'close', 'cache-control': 'private, s-maxage=0, max-age=0,
must-revalidate', 'date': 'Thu, 14 Jun 2012 12:59:39 GMT', 'content-type':
'text/html; charset=UTF-8', 'x-cache-lookup': 'HIT from cp1006.eqiad.wmnet:3128,
MISS from cp1010.eqiad.wmnet:80'}
```

然而，如果想得到发送到服务器的请求的头部，我们可以简单地访问该请求，然后是该请求的头部：

```
>>> r.request.headers
{'Accept-Encoding': 'identity, deflate, compress, gzip',
'Accept': '*/*', 'User-Agent': 'python-requests/0.13.1'}
```

准备的请求 （Prepared Request）

当你从 API 或者会话调用中收到一个 **Response** 对象时，request 属性其实是使用了 PreparedRequest。有时在发送请求之前，你需要对 body 或者 header （或者别的什么东西）做一些额外处理，下面演示了一个简单的做法：

```
from requests import Request, Session

s = Session()
req = Request('GET', url,
    data=data,
    headers=header
)
prepped = req.prepare()

# do something with prepped.body
# do something with prepped.headers

resp = s.send(prepped,
    stream=stream,
    verify=verify,
    proxies=proxies,
    cert=cert,
    timeout=timeout
)

print(resp.status_code)
```

由于你没有对 Request 对象做什么特殊事情，你立即准备和修改了 PreparedRequest 对象，然后把它和别的参数一起发送到 requests.* 或者 Session.*。

然而，上述代码会失去 Requests **Session** 对象的一些优势， 尤其 **Session** 级别的状态，例如 cookie 就不会被应用到你的请求上去。要获取一个带有状态的 **PreparedRequest**， 请用 **Session.prepare_request()** 取代 **Request.prepare()** 的调用，如下所示：

```
from requests import Request, Session

s = Session()
req = Request('GET',  url,
    data=data
    headers=headers
```

```
)

prepped = s.prepare_request(req)

# do something with prepped.body
# do something with prepped.headers

resp = s.send(prepped,
    stream=stream,
    verify=verify,
    proxies=proxies,
    cert=cert,
    timeout=timeout
)

print(resp.status_code)
```

SSL 证书验证

Requests 可以为 HTTPS 请求验证 SSL 证书，就像 web 浏览器一样。要想检查某个主机的 SSL 证书，你可以使用 `verify` 参数：

```
>>> requests.get('https://kennethreitz.com', verify=True)
requests.exceptions.SSLError: hostname 'kennethreitz.com' doesn't match either of '*.herokuapp.com',
'herokuapp.com'
```

在该域名上我没有设置 SSL，所以失败了。但 Github 设置了 SSL：

```
>>> requests.get('https://github.com', verify=True)
<Response [200]>
```

对于私有证书，你也可以传递一个 CA_BUNDLE 文件的路径给 `verify`。你也可以设置`REQUEST_CA_BUNDLE` 环境变量。

如果你将 `verify` 设置为 False，Requests 也能忽略对 SSL 证书的验证。

```
>>> requests.get('https://kennethreitz.com', verify=False)
<Response [200]>
```

默认情况下， `verify` 是设置为 True 的。选项 `verify` 仅应用于主机证书。

你也可以指定一个本地证书用作客户端证书，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：

```
>>> requests.get('https://kennethreitz.com', cert=('/path/server.crt', '/path/key'))
<Response [200]>
```

如果你指定了一个错误路径或一个无效的证书：

```
>>> requests.get('https://kennethreitz.com', cert='/wrong_path/server.pem')
SSLError: [Errno 336265225] _ssl.c:347: error:140B0009:SSL routines:SSL_CTX_use_PrivateKey_file:PEM lib
```

警告

本地证书的私有 key 必须是解密状态。目前，Requests 不支持使用加密的 key。

CA 证书

Requests 默认附带了一套它信任的根证书，来自于 Mozilla trust store。然而它们在每次 Requests 更新时才会更新。这意味着如果你固定使用某一版本的 Requests，你的证书有可能已经 太旧了。

从 Requests 2.4.0 版之后，如果系统中装了 certifi 包，Requests 会试图使用它里边的 证书。这样用户就可以在不修改代码的情况下更新他们的可信任证书。

为了安全起见，我们建议你经常更新 certifi！

响应体内容工作流

默认情况下，当你进行网络请求后，响应体会立即被下载。你可以通过 `stream` 参数覆盖这个行为，推迟下载响应体直到访问 **`Response.content`** 属性：

```
tarball_url = 'https://github.com/kennethreitz/requests/tarball/master'
r = requests.get(tarball_url, stream=True)
```

此时仅有响应头被下载下来了，连接保持打开状态，因此允许我们根据条件获取内容：

```
if int(r.headers['content-length']) < TOO_LONG:
  content = r.content
  ...
```

你可以进一步使用 **Response.iter_content** 和 **Response.iter_lines** 方法来控制工作流，或者以 **Response.raw** 从底层 urllib3 的 **urllib3.HTTPResponse <urllib3.response.HTTPResponse** 读取。

如果你在请求中把 `stream` 设为 `True`，Requests 无法将连接释放回连接池，除非你 消耗了所有的数据，或者调用了 **Response.close**。 这样会带来连接效率低下的问题。如果你发现你在使用 `stream=True` 的同时还在部分读取请求的 body（或者完全没有读取 body），那么你就应该考虑使用 `contextlib.closing`（文档）， 如下所示：

```
from contextlib import closing

with closing(requests.get('http://httpbin.org/get', stream=True)) as r:
    # 在此处理响应。
```

保持活动状态（持久连接）

好消息——归功于 urllib3，同一会话内的持久连接是完全自动处理的！同一会话内你发出的任何请求都会自动复用恰当的连接！

注意：只有所有的响应体数据被读取完毕连接才会被释放为连接池；所以确保将 `stream` 设置为 `False` 或读取 `Response` 对象的 `content` 属性。

流式上传

Requests支持流式上传，这允许你发送大的数据流或文件而无需先把它们读入内存。要使用流式上传，仅需为你的请求体提供一个类文件对象即可：

```
with open('massive-body') as f:
    requests.post('http://some.url/streamed', data=f)
```

警告

我们强烈建议你用二进制模式（binary mode）打开文件。这是因为 requests 可能会为你提供 header 中的 `Content-Length`，在这种情况下该值会被设为文件的字节数。如果你用文本模式打开文件，就可能碰到错误。

块编码请求

对于出去和进来的请求，Requests 也支持分块传输编码。要发送一个块编码的请求，仅需为你的请求体提供一个生成器（或任意没有具体长度的迭代器）：

```
def gen():
    yield 'hi'
    yield 'there'

requests.post('http://some.url/chunked', data=gen())
```

对于分块的编码请求，我们最好使用 **Response.iter_content()** 对其数据进行迭代。在理想情况下，你的 request 会设置 `stream=True`，这样你就可以通过调用 `iter_content` 并将分块大小参数设为 `None`，从而进行分块的迭代。如果你要设置分块的最大体积，你可以把分块大小参数设为任意整数。

POST 多个分块编码的文件

你可以在一个请求中发送多个文件。例如，假设你要上传多个图像文件到一个 HTML 表单，使用一个多文件 field 叫做 "images"：

```
<input type="file" name="images" multiple="true" required="true"/>
```

要实现，只要把文件设到一个元组的列表中，其中元组结构为 `(form_field_name, file_info)`：

```
>>> url = 'http://httpbin.org/post'
>>> multiple_files = [
        ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
        ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]
```

```
>>> r = requests.post(url, files=multiple_files)
>>> r.text
{
  ...
  'files': {'images': ' ....'}
  'Content-Type': 'multipart/form-data; boundary=3131623adb2043caaeb5538cc7aa0b3a',
  ...
}
```

警告

我们强烈建议你用二进制模式（binary mode）打开文件。这是因为 requests 可能会为你提供 header 中的 `Content-Length`，在这种情况下该值会被设为文件的**字节数**。如果你用**文本模式**打开文件，就可能碰到错误。

事件挂钩

Requests有一个钩子系统，你可以用来操控部分请求过程，或信号事件处理。

可用的钩子:

`response`:

从一个请求产生的响应

你可以通过传递一个 `{hook_name: callback_function}` 字典给 `hooks` 请求参数 为每个请求分配一个钩子函数：

```
hooks=dict(response=print_url)
```

`callback_function` 会接受一个数据块作为它的第一个参数。

```
def print_url(r, *args, **kwargs):
    print(r.url)
```

若执行你的回调函数期间发生错误，系统会给出一个警告。

若回调函数返回一个值，默认以该值替换传进来的数据。若函数未返回任何东西， 也没有什么其他的影响。

我们来在运行期间打印一些请求方法的参数：

```
>>> requests.get('http://httpbin.org', hooks=dict(response=print_url))
http://httpbin.org
<Response [200]>
```

自定义身份验证

Requests 允许你使用自己指定的身份验证机制。

任何传递给请求方法的 `auth` 参数的可调用对象，在请求发出之前都有机会修改请求。

自定义的身份验证机制是作为 `requests.auth.AuthBase` 的子类来实现的，也非常容易定义。Requests 在 `requests.auth` 中提供了两种常见的的身份验证方案： `HTTPBasicAuth` 和 `HTTPDigestAuth` 。

假设我们有一个web服务，仅在 `X-Pizza` 头被设置为一个密码值的情况下才会有响应。虽然这不太可能，但就以它为例好了。

```
from requests.auth import AuthBase

class PizzaAuth(AuthBase):
    """Attaches HTTP Pizza Authentication to the given Request object."""
    def __init__(self, username):
        # setup any auth-related data here
        self.username = username

    def __call__(self, r):
        # modify and return the request
        r.headers['X-Pizza'] = self.username
        return r
```

然后就可以使用我们的PizzaAuth来进行网络请求:

```
>>> requests.get('http://pizzabin.org/admin', auth=PizzaAuth('kenneth'))
<Response [200]>
```

流式请求

使用 **requests.Response.iter_lines()** 你可以很方便地对流式 API （例如 Twitter 的流式 API ） 进行迭代。简单地设置 `stream` 为 `True` 便可以使用 **iter_lines()** 对相应进行迭代：

```
import json
import requests

r = requests.get('http://httpbin.org/stream/20', stream=True)

for line in r.iter_lines():

    # filter out keep-alive new lines
    if line:
        print(json.loads(line))
```

警告

iter_lines() 不保证重进入时的安全性。多次调用该方法 会导致部分收到的数据丢失。如果你要在多处调用它，就应该使用生成的迭代器对象：

```
lines = r.iter_lines()
# Save the first line for later or just skip it

first_line = next(lines)

for line in lines:
    print(line)
```

代理

如果需要使用代理，你可以通过为任意请求方法提供 `proxies` 参数来配置单个请求：

```
import requests

proxies = {
  "http": "http://10.10.1.10:3128",
  "https": "http://10.10.1.10:1080",
}

requests.get("http://example.org", proxies=proxies)
```

你也可以通过环境变量 `HTTP_PROXY` 和 `HTTPS_PROXY` 来配置代理。

```
$ export HTTP_PROXY="http://10.10.1.10:3128"
$ export HTTPS_PROXY="http://10.10.1.10:1080"

$ python
>>> import requests
>>> requests.get("http://example.org")
```

若你的代理需要使用HTTP Basic Auth，可以使用 *http://user:password@host/* 语法：

```
proxies = {
    "http": "http://user:pass@10.10.1.10:3128/",
}
```

要为某个特定的连接方式或者主机设置代理，使用 *scheme://hostname* 作为 key， 它会针对指定的主机和连接方式进行匹配。

```
proxies = {'http://10.20.1.128': 'http://10.10.1.10:5323'}
```

注意，代理 URL 必须包含连接方式。

SOCKS

2.10.0 新版功能.

除了基本的 HTTP 代理，Request 还支持 SOCKS 协议的代理。这是一个可选功能，若要使用， 你需要安装第三方库。

你可以用 `pip` 获取依赖:

```
$ pip install requests[socks]
```

安装好依赖以后，使用 SOCKS 代理和使用 HTTP 代理一样简单：

```
proxies = {
    'http': 'socks5://user:pass@host:port',
    'https': 'socks5://user:pass@host:port'
}
```

合规性

Requests 符合所有相关的规范和 RFC，这样不会为用户造成不必要的困难。但这种对规范的考虑导致一些行为对于不熟悉相关规范的人来说看似有点奇怪。

编码方式

当你收到一个响应时，Requests 会猜测响应的编码方式，用于在你调用 **`Response.text`** 方法时对响应进行解码。Requests 首先在 HTTP 头部检测是否存在指定的编码方式，如果不存在，则会使用charade 来尝试猜测编码方式。只有当 HTTP 头部不存在明确指定的字符集，并且 `Content-Type` 头部字段包含 `text` 值之时， Requests 才不去猜测编码方式。在这种情况下， RFC 2616 指定默认字符集必须是 `ISO-8859-1` 。Requests 遵从这一规范。如果你需要一种不同的编码方式，你可以手动设置 **`Response.encoding`** 属性，或使用原始的 **`Response.content`**。

HTTP动词

Requests 提供了几乎所有HTTP动词的功能：GET、OPTIONS、HEAD、POST、PUT、PATCH、DELETE。以下内容为使用 Requests 中的这些动词以及 Github API 提供了详细示例。

我将从最常使用的动词 GET 开始。HTTP GET 是一个幂等方法，从给定的 URL 返回一个资源。因而，当你试图从一个 web 位置获取数据之时，你应该使用这个动词。一个使用示例是尝试从 Github 上获取关于一个特定 commit 的信息。假设我们想获取Requests的commit `a050faf` 的信息。我们可以这样去做：

```
>>> import requests
>>> r =
requests.get('https://api.github.com/repos/kennethreitz/requests/git/commits/a050faf084662f3a352dd1a941f2c7c9f886d4ad')
```

我们应该确认 GitHub 是否正确响应。如果正确响应，我们想弄清响应内容是什么类型的。像这样去做：

```
>>> if (r.status_code == requests.codes.ok):
...     print r.headers['content-type']
...
application/json; charset=utf-8
```

可见，GitHub 返回了 JSON 数据，非常好，这样就可以使用 `r.json` 方法把这个返回的数据解析成 Python 对象。

```
>>> commit_data = r.json()

>>> print commit_data.keys()
[u'committer', u'author', u'url', u'tree', u'sha', u'parents', u'message']

>>> print commit_data[u'committer']
{u'date': u'2012-05-10T11:10:50-07:00', u'email': u'me@kennethreitz.com', u'name': u'Kenneth Reitz'}

>>> print commit_data[u'message']
makin' history
```

到目前为止，一切都非常简单。嗯，我们来研究一下 GitHub 的 API。我们可以去看看文档，但如果使用 Requests 来研究也许会更有意思一点。我们可以借助 Requests 的 OPTIONS 动词来看看我们刚使用过的 url 支持哪些 HTTP 方法。

```
>>> verbs = requests.options(r.url)
>>> verbs.status_code
```

```
500
```

额，这是怎么回事？毫无帮助嘛！原来 GitHub，与许多 API 提供方一样，实际上并未实现 OPTIONS 方法。这是一个恼人的疏忽，但没关系，那我们可以使用枯燥的文档。然而，如果 GitHub 正确实现了 OPTIONS，那么服务器应该在响应头中返回允许用户使用的 HTTP 方法，例如：

```
>>> verbs = requests.options('http://a-good-website.com/api/cats')
>>> print verbs.headers['allow']
GET,HEAD,POST,OPTIONS
```

转而去查看文档，我们看到对于提交信息，另一个允许的方法是 POST，它会创建一个新的提交。由于我们正在使用 Requests 代码库，我们应尽可能避免对它发送笨拙的 POST。作为替代，我们来玩玩 GitHub 的 Issue 特性。

本篇文档是回应 Issue #482 而添加的。鉴于该问题已经存在，我们就以它为例。先获取它。

```
>>> r = requests.get('https://api.github.com/repos/kennethreitz/requests/issues/482')
>>> r.status_code
200

>>> issue = json.loads(r.text)

>>> print(issue[u'title'])
Feature any http verb in docs

>>> print(issue[u'comments'])
3
```

Cool，有 3 个评论。我们来看一下最后一个评论。

```
>>> r = requests.get(r.url + u'/comments')
>>> r.status_code
200
>>> comments = r.json()
>>> print comments[0].keys()
[u'body', u'url', u'created_at', u'updated_at', u'user', u'id']
>>> print comments[2][u'body']
Probably in the "advanced" section
```

嗯，那看起来似乎是个愚蠢之处。我们发表个评论来告诉这个评论者他自己的愚蠢。那么，这个评论者是谁呢？

```
>>> print comments[2][u'user'][u'login']
kennethreitz
```

好，我们来告诉这个叫 Kenneth 的家伙，这个例子应该放在快速上手指南中。根据 GitHub API 文档，其方法是 POST 到该话题。我们来试试看。

```
>>> body = json.dumps({u"body": u"Sounds great! I'll get right on it!"})
>>> url = u"https://api.github.com/repos/kennethreitz/requests/issues/482/comments"

>>> r = requests.post(url=url, data=body)
>>> r.status_code
404
```

额，这有点古怪哈。可能我们需要验证身份。那就有点纠结了，对吧？不对。Requests 简化了多种身份验证形式的使用，包括非常常见的 Basic Auth。

```
>>> from requests.auth import HTTPBasicAuth
>>> auth = HTTPBasicAuth('fake@example.com', 'not_a_real_password')

>>> r = requests.post(url=url, data=body, auth=auth)
>>> r.status_code
201

>>> content = r.json()
>>> print(content[u'body'])
Sounds great! I'll get right on it.
```

太棒了！噢，不！我原本是想说等我一会，因为我得去喂我的猫。如果我能够编辑这条评论那就好了！幸运的

是，GitHub 允许我们使用另一个 HTTP 动词 PATCH 来编辑评论。我们来试试。

```
>>> print(content[u"id"])
5804413

>>> body = json.dumps({u"body": u"Sounds great! I'll get right on it once I feed my cat."})
>>> url = u"https://api.github.com/repos/kennethreitz/requests/issues/comments/5804413"

>>> r = requests.patch(url=url, data=body, auth=auth)
>>> r.status_code
200
```

非常好。现在，我们来折磨一下这个叫 Kenneth 的家伙，我决定要让他急得团团转，也不告诉他是我在捣蛋。这意味着我想删除这条评论。GitHub 允许我们使用完全名副其实的 DELETE 方法来删除评论。我们来清除该评论。

```
>>> r = requests.delete(url=url, auth=auth)
>>> r.status_code
204
>>> r.headers['status']
'204 No Content'
```

很好。不见了。最后一件我想知道的事情是我已经使用了多少限额（ratelimit）。查查看，GitHub 在响应头部发送这个信息，因此不必下载整个网页，我将使用一个 HEAD 请求来获取响应头。

```
>>> r = requests.head(url=url, auth=auth)
>>> print r.headers
...
'x-ratelimit-remaining': '4995'
'x-ratelimit-limit': '5000'
...
```

很好。是时候写个 Python 程序以各种刺激的方式滥用 GitHub 的 API，还可以使用 4995 次呢。

响应头链接字段

许多 HTTP API 都有响应头链接字段的特性，它们使得 API 能够更好地自我描述和自我显露。

GitHub 在 API 中为 分页 使用这些特性，例如：

```
>>> url = 'https://api.github.com/users/kennethreitz/repos?page=1&per_page=10'
>>> r = requests.head(url=url)
>>> r.headers['link']
'<https://api.github.com/users/kennethreitz/repos?page=2&per_page=10>; rel="next",
<https://api.github.com/users/kennethreitz/repos?page=6&per_page=10>; rel="last"'
```

Requests 会自动解析这些响应头链接字段，并使得它们非常易于使用：

```
>>> r.links["next"]
{'url': 'https://api.github.com/users/kennethreitz/repos?page=2&per_page=10', 'rel': 'next'}

>>> r.links["last"]
{'url': 'https://api.github.com/users/kennethreitz/repos?page=7&per_page=10', 'rel': 'last'}
```

传输适配器

从 v1.0.0 以后，Requests 的内部采用了模块化设计。部分原因是为了实现传输适配器（Transport Adapter），你可以看看关于它的最早描述。传输适配器提供了一个机制，让你可以为 HTTP 服务定义交互方法。尤其是它允许你应用服务前的配置。

Requests 自带了一个传输适配器，也就是 **HTTPAdapter**。 这个适配器使用了强大的 urllib3，为 Requests 提供了默认的 HTTP 和 HTTPS 交互。每当 **Session** 被初始化，就会有适配器附着在 **Session** 上，其中一个供 HTTP 使用，另一个供 HTTPS 使用。

Request 允许用户创建和使用他们自己的传输适配器，实现他们需要的特殊功能。创建好以后，传输适配器可以被加载到一个会话对象上，附带着一个说明，告诉会话适配器应该应用在哪个 web 服务上。

```
>>> s = requests.Session()
```

```
>>> s.mount('http://www.github.com', MyAdapter())
```

这个 mount 调用会注册一个传输适配器的特定实例到一个前缀上面。加载以后，任何使用该会话的 HTTP 请求，只要其 URL 是以给定的前缀开头，该传输适配器就会被使用到。

传输适配器的众多实现细节不在本文档的覆盖范围内，不过你可以看看接下来这个简单的 SSL 用例。更多的用法，你也许该考虑为``requests.adapters.BaseAdapter`` 创建子类。

示例：指定的 SSL 版本

Requests 开发团队刻意指定了内部库（urllib3）的默认 SSL 版本。一般情况下这样做没有问题，不过是不是你可能会需要连接到一个服务节点，而该节点使用了和默认不同的 SSL 版本。

你可以使用传输适配器解决这个问题，通过利用 HTTPAdapter 现有的大部分实现，再加上一个*ssl_version* 参数并将它传递到 `urllib3` 中。我们会创建一个传输适配器，用来告诉 `urllib3` 让它使用 SSLv3：

```
import ssl

from requests.adapters import HTTPAdapter
from requests.packages.urllib3.poolmanager import PoolManager


class Ssl3HttpAdapter(HTTPAdapter):
    """"Transport adapter" that allows us to use SSLv3."""

    def init_poolmanager(self, connections, maxsize, block=False):
        self.poolmanager = PoolManager(num_pools=connections,
                                       maxsize=maxsize,
                                       block=block,
                                       ssl_version=ssl.PROTOCOL_SSLv3)
```

阻塞和非阻塞

使用默认的传输适配器，Requests 不提供任何形式的非阻塞 IO。 **`Response.content`** 属性会阻塞，直到整个响应下载完成。如果你需要更多精细控制，该库的数据流功能（见 流式请求） 允许你每次接受少量的一部分响应，不过这些调用依然是阻塞式的。

如果你对于阻塞式 IO 有所顾虑，还有很多项目可以供你使用，它们结合了 Requests 和 Python 的某个异步框架。典型的优秀例子是 grequests 和 requests-futures。

Header 排序

在某些特殊情况下你也许需要按照次序来提供 header，如果你向 `headers` 关键字参数传入一个`OrderedDict`，就可以向提供一个带排序的 header。**然而**，Requests 使用的默认 header 的次序会被优先选择，这意味着如果你在 `headers` 关键字参数中覆盖了默认 header，和关键字参数中别的 header 相比，它们也许看上去会是次序错误的。

如果这个对你来说是个问题，那么用户应该考虑在 **`Session`** 对象上面设置默认 header，只要将 **`Session`** 设为一个定制的 `OrderedDict` 即可。这样就会让它成为优选的次序。

超时（timeout）

为防止服务器不能及时响应，大部分发至外部服务器的请求都应该带着 timeout 参数。如果没有 timeout，你的代码可能会挂起若干分钟甚至更长时间。

连接超时指的是在你的客户端实现到远端机器端口的连接时（对应的是`connect()`_），Request 会等待的秒数。一个很好的实践方法是把连接超时设为比 3 的倍数略大的一个数值，因为 TCP 数据包重传窗口 (TCP packet retransmission window) 的默认大小是 3。

一旦你的客户端连接到了服务器并且发送了 HTTP 请求，**读取**超时指的就是客户端等待服务器发送请求的时间。（特定地，它指的是客户端要等待服务器发送字节**之间**的时间。在 99.9% 的情况下这指的是服务器发送第一个字节之前的时间）。

如果你制订了一个单一的值作为 timeout，如下所示：

```
r = requests.get('https://github.com', timeout=5)
```

这一 timeout 值将会用作 `connect` 和 `read` 二者的 timeout。如果要分别制定，就传入一个元组：

```
r = requests.get('https://github.com', timeout=(3.05, 27))
```

如果远端服务器很慢，你可以让 Request 永远等待，传入一个 None 作为 timeout 值，然后就冲咖啡去吧。

```
r = requests.get('https://github.com', timeout=None)
```