Requests: HTTP for Humans

发行版本 v1.1.0. (安装)

Requests 是使用 *Apache2 Licensed* 许可证的 HTTP 库。用 Python 编写，真正的为人类着想。

Python 标准库中的 **urllib2** 模块提供了你所需要的大多数 HTTP 功能，但是它的 API 太渣了。它是为另一个时代、另一个互联网所创建的。它需要巨量的工作，甚至包括各种方法覆盖，来完成最简单的任务。

在Python的世界里，事情不应该这么麻烦。

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

参见 未使用 Requests 的相似代码 [https://gist.github.com/973705].

Requests 使用的是 urllib3，因此继承了它的所有特性。Requests 支持 HTTP 连接保持和连接池，支持使用 cookie 保持会话，支持文件上传，支持自动确定响应内容的编码，支持国际化的 URL 和 POST 数据自动编码。现代、国际化、人性化。

Testimonials

Her Majesty's Government, Amazon, Google, Twilio, Mozilla, Heroku, PayPal, NPR, Obama for America, Transifex, Native Instruments, The Washington Post, Twitter, SoundCloud, Kippt, Readability, and Federal US Institutions use Requests internally. It has been downloaded over 2,000,000 times from PyPI.

Armin Ronacher
: Requests is the perfect example how beautiful an API can be with the right level of abstraction.

Matt DeBoard
: I'm going to get @kennethreitz's Python requests module tattooed on my body, somehow. The whole thing.

Daniel Greenfeld
: Nuked a 1200 LOC spaghetti code library with 10 lines of code thanks to @kennethreitz's request library. Today has been AWESOME.

Kenny Meyers
: Python HTTP: When in doubt, or when not in doubt, use Requests. Beautiful, simple, Pythonic.

功能特性

Requests 完全满足如今网络的需求。

- 国际化域名和 URLs
- Keep-Alive & 连接池
- 持久的 Cookie 会话
- 类浏览器式的 SSL 加密认证
- 基本/摘要式的身份认证
- 优雅的键/值 Cookies
- 自动解压
- Unicode 编码的响应体
- 多段文件上传
- 连接超时
- 支持 `.netrc`
- 适用于 Python 2.6—3.4
- 线程安全

用户指南

这部分文档主要介绍了 Requests 的背景，然后对于 Requests 的应用做了一步一步的要点介绍。

- 简介
  - 开发哲学
  - Apache2 协议
  - Requests 协议
- 安装
  - Distribute & Pip

    - 获得源码
- 快速上手
    - 发送请求
    - 为URL传递参数
    - 响应内容
    - 二进制响应内容
    - JSON响应内容
    - 原始响应内容
    - 定制请求头
    - 更加复杂的POST请求
    - POST一个多部分编码(Multipart-Encoded)的文件
    - 响应状态码
    - 响应头
    - Cookies
    - 重定向与请求历史
    - 超时
    - 错误与异常
- 高级用法
    - 会话对象
    - 请求与响应对象
    - Prepared Requests
    - SSL证书验证
    - 响应体内容工作流
    - 保持活动状态（持久连接）
    - 流式上传
    - 块编码请求
    - POST Multiple Multipart-Encoded Files
    - 事件挂钩
    - 自定义身份验证
    - 流式请求
    - 代理
    - 合规性
    - HTTP动词
    - 响应头链接字段
    - Transport Adapters
    - Blocking Or Non-Blocking?
    - Timeouts
    - CA Certificates
- 身份认证
    - 基本身份认证
    - 摘要式身份认证
    - OAuth 1 Authentication
    - 其他身份认证形式
    - 新的身份认证形式

社区指南

这部分文档主要详细地介绍了Requests的社区支持情况

- 常见问题
- 整合
- Articles & Talks
- 支持
- 更新

API 文档

如果你正在寻找一个特殊函数，类或者方法的话，这部分文档刚好满足你。

- 开发者接口
    - 主要接口
    - 请求会话
    - 迁移到 1.x

贡献者指南

如果你对这个项目有兴趣，想做一点贡献，请参考下面文档

- 开发理念
- 如何帮助
- 贡献者

简介

开发哲学

Requests 是以 **PEP 20** [http://www.python.org/dev/peps/pep-0020] 的习语为中心开发的

1. Beautiful is better than ugly.(美丽优于丑陋)
2. Explicit is better than implicit.(清楚优于含糊)
3. Simple is better than complex.(简单优于复杂)
4. Complex is better than complicated.(复杂优于繁琐)
5. Readability counts.(重要的是可读性)

对于 Requests 所有的贡献都应牢记这些重要的准则。

Apache2 协议

现在你找到的许多开源项目都是以 GPL Licensed [http://www.opensource.org/licenses/gpl-license.php] 发布的. 虽然 GPL 有它自己的时间和空间，但应该确定的是它不会是你下一个开源项目的发布协议。

即使一个项目发行于GPL协议也不能用于任何本身没开源的商业产品。

The MIT, BSD, ISC, and Apache2 licenses are great alternatives to the GPL that allow your open-source software to be used freely in proprietary, closed-source software.

Requests is released under terms of Apache2 License [http://opensource.org/licenses/Apache-2.0].

Requests 协议

安装

这部分文档包含了Requests的安装过程，使用任何软件的第一步就是合适的安装好它.

Distribute & Pip

使用 pip [http://www.pip-installer.org/] 安装Requests非常简单

```
$ pip install requests
```

或者使用 easy_install [http://pypi.python.org/pypi/setuptools] 安装

```
$ easy_install requests
```

但是你最好 不要这样 [http://www.pip-installer.org/en/latest/other-tools.html#pip-compared-to-easy-install].

获得源码

Requests 一直在Github上被积极的开发着，你可以在此获得代码，且总是 可用的 [https://github.com/kennethreitz/requests].

你也可以克隆公共版本库:

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

下载 源码 [https://github.com/kennethreitz/requests/tarball/master]:

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

或者下载 zipball [https://github.com/kennethreitz/requests/zipball/master]:

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

一旦你获得了复本，你就可以轻松的将它嵌入到你的python包里或者安装到你的site-packages:

```
$ python setup.py install
```

快速上手

迫不及待了吗？本页内容为如何入门Requests提供了很好的指引。其假设你已经安装了Requests。如果还没有， 去 *安装* 一节看看吧。

首先，确认一下：

- Requests *已安装*
- Requests是 *最新的*

让我们从一些简单的示例开始吧。

发送请求

使用Requests发送网络请求非常简单。

一开始要导入Requests模块:

```
>>> import requests
```

然后，尝试获取某个网页。本例子中，我们来获取Github的公共时间线

```
>>> r = requests.get('https://github.com/timeline.json')
```

现在，我们有一个名为 `r` 的 **Response** 对象。可以从这个对象中获取所有我们想要的信息。

Requests简便的API意味着所有HTTP请求类型都是显而易见的。例如，你可以这样发送一个HTTP POST请求:

```
>>> r = requests.post("http://httpbin.org/post")
```

漂亮，对吧？那么其他HTTP请求类型：PUT， DELETE， HEAD以及OPTIONS又是如何的呢？都是一样的简单:

```
>>> 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")
```

都很不错吧，但这也仅是Requests的冰山一角呢。

为URL传递参数

你也许经常想为URL的查询字符串(query string)传递某种数据。如果你是手工构建URL，那么数据会以键/值 对的形式置于URL中，跟在一个问号的后面。例如， `httpbin.org/get?key=val` 。 Requests允许你使用 `params` 关键字参数，以一个字典来提供这些参数。举例来说，如果你想传递 `key1=value1` 和 `key2=value2` 到 `httpbin.org/get` ，那么你可以使用如下代码:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.get("http://httpbin.org/get", params=payload)
```

通过打印输出该URL，你能看到URL已被正确编码:

```
>>> print(r.url)
http://httpbin.org/get?key2=value2&key1=value1
```

注意字典里值为 `None` 的键都不会被添加到 URL 的查询字符串里。

响应内容

我们能读取服务器响应的内容。再次以Github时间线为例:

```
>>> import requests
>>> r = requests.get('https://github.com/timeline.json')
>>> r.text
u'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

Requests会自动解码来自服务器的内容。大多数unicode字符集都能被无缝地解码。

请求发出后，Requests会基于HTTP头部对响应的编码作出有根据的推测。当你访问 `r.text` 之时，Requests会使用其推测的文本编码。你可以找出Requests使用了什么编码，并且能够使用 `r.encoding` 属性来改变它:

```
>>> r.encoding
'utf-8'
>>> r.encoding = 'ISO-8859-1'
```

如果你改变了编码，每当你访问 `r.text` ，Request都将会使用 `r.encoding` 的新值。你可能希望在使用特殊逻辑计算出文本的编码的情况下来修改编码。比如 HTTP 和 XML 自身可以指定编码。这样的话，你应该使用 `r.content` 来找到编码，然后设置 `r.encoding` 为相应的编码。这样就能使用正确的编码解析 `r.text` 了。

在你需要的情况下，Requests也可以使用定制的编码。如果你创建了自己的编码，并使用 `codecs` 模块进行注册，你就可以轻松地使用这个解码器名称作为 `r.encoding` 的值， 然后由Requests来为你处理编码。

二进制响应内容

你也能以字节的方式访问请求响应体，对于非文本请求:

```
>>> r.content
b'[{"repository":{"open_issues":0,"url":"https://github.com/...
```

Requests会自动为你解码 `gzip` 和 `deflate` 传输编码的响应数据。

例如，以请求返回的二进制数据创建一张图片，你可以使用如下代码:

```
>>> from PIL import Image
>>> from StringIO import StringIO
>>> i = Image.open(StringIO(r.content))
```

JSON响应内容

Requests中也有一个内置的JSON解码器，助你处理JSON数据:

```
>>> import requests
>>> r = requests.get('https://github.com/timeline.json')
>>> r.json()
[{u'repository': {u'open_issues': 0, u'url': 'https://github.com/...
```

如果JSON解码失败， `r.json` 就会抛出一个异常。例如，相应内容是 401 (Unauthorized)，尝试访问 `r.json` 将会抛出 `ValueError: No JSON object could be decoded` 异常。

原始响应内容

在罕见的情况下你可能想获取来自服务器的原始套接字响应，那么你可以访问 `r.raw` 。 如果你确实想这么干，那请你确保在初始请求中设置了 `stream=True` 。具体的你可以这么做:

```
>>> r = requests.get('https://github.com/timeline.json', stream=True)
>>> r.raw
<requests.packages.urllib3.response.HTTPResponse object at 0x101194810>
>>> r.raw.read(10)
'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'
```

但一般情况下，你应该下面的模式将文本流保存到文件:

```
with open(filename, 'wb') as fd:
    for chunk in r.iter_content(chunk_size):
        fd.write(chunk)
```

使用 `Response.iter_content` 将会处理大量你直接使用 `Response.raw` 不得不处理的。 当流下载时，上面是优先推荐的获取内容方式。

定制请求头

如果你想为请求添加HTTP头部，只要简单地传递一个 `dict` 给 `headers` 参数就可以了。

例如，在前一个示例中我们没有指定content-type:

```
>>> import json
>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}
>>> headers = {'content-type': 'application/json'}

>>> r = requests.post(url, data=json.dumps(payload), headers=headers)
```

更加复杂的POST请求

通常，你想要发送一些编码为表单形式的数据—非常像一个HTML表单。 要实现这个，只需简单地传递一个字典给 *data* 参数。你的数据字典在发出请求时会自动编码为表单形式:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.post("http://httpbin.org/post", data=payload)
>>> print r.text
{
  ...
  "form": {
    "key2": "value2",
    "key1": "value1"
  },
  ...
}
```

很多时候你想要发送的数据并非编码为表单形式的。如果你传递一个 `string` 而不是一个 `dict` ，那么数据会被直接发布出去。

例如，Github API v3接受编码为JSON的POST/PATCH数据:

```
>>> import json
>>> url = 'https://api.github.com/some/endpoint'
>>> payload = {'some': 'data'}

>>> r = requests.post(url, data=json.dumps(payload))
```

POST一个多部分编码(Multipart-Encoded)的文件

Requests使得上传多部分编码文件变得很简单:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': open('report.xls', 'rb')}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<censored...binary...data>"
  },
  ...
}
```

你可以显式地设置文件名，文件类型和请求头:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel', {'Expires': '0'})}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "<censored...binary...data>"
  },
  ...
}
```

如果你想，你也可以发送作为文件来接收的字符串:

```
>>> url = 'http://httpbin.org/post'
>>> files = {'file': ('report.csv', 'some,data,to,send\nanother,row,to,send\n')}

>>> r = requests.post(url, files=files)
>>> r.text
{
  ...
  "files": {
    "file": "some,data,to,send\\nanother,row,to,send\\n"
  },
  ...
}
```

如果你发送一个非常大的文件作为 `multipart/form-data` 请求，你可能希望流请求(?)。默认下 `requests` 不支持, 但有个第三方包支持 - `requests-toolbelt`. 你可以阅读 toolbelt 文档 [https://toolbelt.rtfd.org] 来了解使用方法。

在一个请求中发送多文件参考 *高级用法* 一节.

响应状态码

我们可以检测响应状态码:

```
>>> r = requests.get('http://httpbin.org/get')
>>> r.status_code
200
```

为方便引用，Requests还附带了一个内置的状态码查询对象:

```
>>> r.status_code == requests.codes.ok
True
```

如果发送了一个失败请求(非200响应)，我们可以通过 **`Response.raise_for_status()`** 来抛出异常:

```
>>> bad_r = requests.get('http://httpbin.org/status/404')
>>> bad_r.status_code
404

>>> bad_r.raise_for_status()
Traceback (most recent call last):
  File "requests/models.py", line 832, in raise_for_status
    raise http_error
requests.exceptions.HTTPError: 404 Client Error
```

但是，由于我们的例子中 r 的 status_code 是 200 ， 当我们调用 raise_for_status() 时，得到的是:

```
>>> r.raise_for_status()
None
```

一切都挺和谐哈。

响应头

我们可以查看以一个Python字典形式展示的服务器响应头:

```
>>> r.headers
{
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'nginx/1.0.4',
    'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json'
}
```

但是这个字典比较特殊：它是仅为HTTP头部而生的。根据 RFC 2616 [http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html] ， HTTP头部是大小写不敏感的。

因此，我们可以使用任意大写形式来访问这些响应头字段:

```
>>> r.headers['Content-Type']
'application/json'

>>> r.headers.get('content-type')
'application/json'
```

Cookies

如果某个响应中包含一些Cookie，你可以快速访问它们:

```
>>> url = 'http://example.com/some/cookie/setting/url'
>>> r = requests.get(url)

>>> r.cookies['example_cookie_name']
'example_cookie_value'
```

要想发送你的cookies到服务器，可以使用 cookies 参数:

```
>>> url = 'http://httpbin.org/cookies'
>>> cookies = dict(cookies_are='working')

>>> r = requests.get(url, cookies=cookies)
>>> r.text
'{"cookies": {"cookies_are": "working"}}'
```

重定向与请求历史

默认情况下，除了 HEAD, Requests会自动处理所有重定向。

可以使用响应对象的 history 方法来追踪重定向。

Response.history 是一个:class:*Response <requests.Response>* 对象的列表，为了完成请求而创建了这些对象。这个对象列表按照从最老到最近的请求进行排序。

例如，Github将所有的HTTP请求重定向到HTTPS。:

```
>>> r = requests.get('http://github.com')
>>> r.url
'https://github.com/'
>>> r.status_code
200
>>> r.history
[<Response [301]>]
```

如果你使用的是GET, OPTIONS, POST, PUT, PATCH 或者 DELETE,，那么你可以通过 allow_redirects 参数禁用重定向处理:

```
>>> r = requests.get('http://github.com', allow_redirects=False)
>>> r.status_code
301
>>> r.history
[]
```

如果你使用的是HEAD，你也可以启用重定向:

```
>>> r = requests.head('http://github.com', allow_redirects=True)
```

```
>>> r.url
'https://github.com/'
>>> r.history
[<Response [301]>]
```

超时

你可以告诉requests在经过以 `timeout` 参数设定的秒数时间之后停止等待响应:

```
>>> requests.get('http://github.com', timeout=0.001)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
requests.exceptions.Timeout: HTTPConnectionPool(host='github.com', port=80): Request timed out. (timeout=0.001)
```

注:

`timeout` 仅对连接过程有效，与响应体的下载无关。 `timeout` is not a time limit on the entire response download; rather, an exception is raised if the server has not issued a response for `timeout` seconds (more precisely, if no bytes have been received on the underlying socket for `timeout` seconds).

错误与异常

遇到网络问题（如：DNS查询失败、拒绝连接等）时，Requests会抛出一个 **`ConnectionError`** 异常。

遇到罕见的无效HTTP响应时，Requests则会抛出一个 **`HTTPError`** 异常。

若请求超时，则抛出一个 **`Timeout`** 异常。

若请求超过了设定的最大重定向次数，则会抛出一个 **`TooManyRedirects`** 异常。

所有Requests显式抛出的异常都继承自 **`requests.exceptions.RequestException`** 。

准备好学习更多内容了吗？去 *高级用法* 一节看看吧。

高级用法

本篇文档涵盖了Requests的一些更加高级的特性。

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cookies。

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cookies:

```
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。

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'}
```

Prepared Requests

Whenever you receive a **Response** object from an API call or a Session call, the `request` attribute is actually the `PreparedRequest` that was used. In some cases you may wish to do some extra work to the body or headers (or anything else really) before sending a request. The simple recipe for this is the following:

```
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)
```

Since you are not doing anything special with the `Request` object, you prepare it immediately and modify the `PreparedRequest` object. You then send that with the other parameters you would have sent to `requests.*` or `Session.*`.

However, the above code will lose some of the advantages of having a Requests **`Session`** object. In particular, **`Session`**-level state such as cookies will not get applied to your request. To get a **`PreparedRequest`** with that state applied, replace the call to **`Request.prepare()`** with a call to **`Session.prepare_request()`**, like this:

```
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
```

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。

If you set `stream` to `True` when making a request, Requests cannot release the connection back to the pool unless you consume all the data or call **Response.close**. This can lead to inefficiency with connections. If you find yourself partially reading request bodies (or not reading them at all) while using `stream=True`, you should consider using `contextlib.closing` (documented here [http://docs.python.org/2/library/contextlib.html#contextlib.closing]), like this:

```
from contextlib import closing

with closing(requests.get('http://httpbin.org/get', stream=True)) as r:
    # Do things with the response here.
```

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)
```

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(without a length)的迭代器）：

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

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

POST Multiple Multipart-Encoded Files

You can send multiple files in one request. For example, suppose you want to upload image files to an HTML form with a multiple file field 'images':

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

To do that, just set files to a list of tuples of (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',
  ...
}
```

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):
    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 [https://dev.twittercom/docs/streaming-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))
```

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/",
}
```

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 [http://pypi.python.org/pypi/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 [http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.7.1] 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和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是一个幂等的方法，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/a050faf084662f3a352dd1a941f2c7c9f886d4ac
```

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，我们来告诉这个叫肯尼思的家伙，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.
```

精彩！噢，不！我原本是想说等我一会，因为我得去喂一下我的猫。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肯尼思的家伙，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 [http://developer.github.com/v3/#pagination] 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
```

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'}
```

Transport Adapters

As of v1.0.0, Requests has moved to a modular internal design. Part of the reason this was done was to implement Transport Adapters, originally described here [http://www.kennethreitz.org/essays/the-future-of-python-http]. Transport Adapters provide a mechanism to define interaction methods for an HTTP service. In particular, they allow you to apply per-service configuration.

Requests ships with a single Transport Adapter, the **`HTTPAdapter`**. This adapter provides the default Requests interaction with HTTP and HTTPS using the powerful urllib3 [https://github.com/shazow/urllib3] library. Whenever a Requests **`Session`** is initialized, one of these is attached to the **`Session`** object for HTTP, and one for HTTPS.

Requests enables users to create and use their own Transport Adapters that provide specific functionality. Once created, a Transport Adapter can be mounted to a Session object, along with an indication of which web services it should apply to.

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

The mount call registers a specific instance of a Transport Adapter to a prefix. Once mounted, any HTTP request made using that session whose URL starts with the given prefix will use the given Transport Adapter.

Many of the details of implementing a Transport Adapter are beyond the scope of this documentation, but take a look at the next example for a simple SSL use- case. For more than that, you might look at subclassing `requests.adapters.BaseAdapter`.

Example: Specific SSL Version

The Requests team has made a specific choice to use whatever SSL version is default in the underlying library (urllib3 [https://github.com/shazow/urllib3]). Normally this is fine, but from time to time, you might find yourself needing to connect to a service-endpoint that uses a version that isn't compatible with the default.

You can use Transport Adapters for this by taking most of the existing implementation of HTTPAdapter, and adding a parameter *ssl_version* that gets passed-through to *urllib3*. We'll make a TA that instructs the library to use 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)
```

Blocking Or Non-Blocking?

With the default Transport Adapter in place, Requests does not provide any kind of non-blocking IO. The **`Response.content`** property will block until the entire response has been downloaded. If you require more granularity, the streaming features of the library (see *流式请求*) allow you to retrieve smaller quantities of the response at a time. However, these calls will still block.

If you are concerned about the use of blocking IO, there are lots of projects out there that combine Requests with one of Python's asynchronicity frameworks. Two excellent examples are grequests [https://github.com/kennethreitz/grequests] and requests-futures [https://github.com/ross/requests-futures].

Timeouts

Most requests to external servers should have a timeout attached, in case the server is not responding in a timely manner. Without a timeout, your code may hang for minutes or more.

The **connect** timeout is the number of seconds Requests will wait for your client to establish a connection to a remote machine (corresponding to the connect() [http://linux.die.net/man/2/connect]) call on the socket. It's a good practice to set connect timeouts to slightly larger than a multiple of 3, which is the default TCP packet retransmission window [http://www.hjp.at/doc/rfc/rfc2988.txt].

Once your client has connected to the server and sent the HTTP request, the **read** timeout is the number of seconds the client will wait for the server to send a response. (Specifically, it's the number of seconds that the client will wait *between* bytes sent from the server. In 99.9% of cases, this is the time before the server sends the first byte).

If you specify a single value for the timeout, like this:

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

The timeout value will be applied to both the `connect` and the `read` timeouts. Specify a tuple if you would like to set the values separately:

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

If the remote server is very slow, you can tell Requests to wait forever for a response, by passing None as a timeout value and then retrieving a

cup of coffee.

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

CA Certificates

By default Requests bundles a set of root CAs that it trusts, sourced from the Mozilla trust store [https://hg.mozilla.org/mozilla-central/raw-file/tip/security/nss/lib/ckfw/builtins/certdata.txt]. However, these are only updated once for each Requests version. This means that if you pin a Requests version your certificates can become extremely out of date.

From Requests version 2.4.0 onwards, Requests will attempt to use certificates from certifi [http://certifi.io/] if it is present on the system. This allows for users to update their trusted certificates without having to change the code that runs on their system.

For the sake of security we recommend upgrading certifi frequently!

身份认证

本篇文档讨论如何配合Requests使用多种身份认证方式。

许多web服务都需要身份认证，并且也有多种不同的认证类型。 以下，我们会从简单到复杂概述Requests中可用的几种身份认证形式。

基本身份认证

许多要求身份认证的web服务都接受HTTP Basic Auth。这是最简单的一种身份认证，并且Requests 对这种认证方式的支持是直接开箱即可用。

以HTTP Basic Auth发送请求非常简单:

```
>>> from requests.auth import HTTPBasicAuth
>>> requests.get('https://api.github.com/user', auth=HTTPBasicAuth('user', 'pass'))
<Response [200]>
```

事实上，HTTP Basic Auth如此常见，Requests就提供了一种简写的使用方式:

```
>>> requests.get('https://api.github.com/user', auth=('user', 'pass'))
<Response [200]>
```

像这样在一个元组中提供认证信息与前一个 `HTTPBasicAuth` 例子是完全相同的。

netrc Authentication

If no authentication method is given with the `auth` argument, Requests will attempt to get the authentication credentials for the URL's hostname from the user's netrc file.

If credentials for the hostname are found, the request is sent with HTTP Basic Auth.

摘要式身份认证

另一种非常流行的HTTP身份认证形式是摘要式身份认证，Requests对它的支持也是开箱即可用的:

```
>>> from requests.auth import HTTPDigestAuth
>>> url = 'http://httpbin.org/digest-auth/auth/user/pass'
>>> requests.get(url, auth=HTTPDigestAuth('user', 'pass'))
<Response [200]>
```

OAuth 1 Authentication

A common form of authentication for several web APIs is OAuth. The `requests-oauthlib` library allows Requests users to easily make OAuth authenticated requests:

```
>>> 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)
<Response [200]>
```

For more information on how to OAuth flow works, please see the official OAuth [http://oauth.net/] website. For examples and documentation on requests-oauthlib, please see the requests_oauthlib [https://github.com/requests/requests-oauthlib] repository on GitHub

其他身份认证形式

Requests被设计成允许其他形式的身份认证非常容易快速地插入其中。开源社区的成员 时常为更复杂或不那么常用的身份认证形式编写认证处理插件。其中一些最优秀的已被 收集在 Requests organization [https://github.com/requests] 页面中，包括:

- Kerberos [https://github.com/requests/requests-kerberos]
- NTLM [https://github.com/requests/requests-ntlm]

如果你想使用其中任何一种身份认证形式，直接去它们的GitHub页面，依照说明进行。

新的身份认证形式

如果你找不到所需要的身份认证形式的一个良好实现，你也可以自己实现它。Requests非常易于添加你自己的身份认证形式。

要想自己实现，就从 **`requests.auth.AuthBase`** 继承一个子类，并实现 `__call__()` 方法:

```
>>> import requests
>>> class MyAuth(requests.auth.AuthBase):
...     def __call__(self, r):
...         # Implement my authentication
...         return r
...
>>> url = 'http://httpbin.org/get'
>>> requests.get(url, auth=MyAuth())
<Response [200]>
```

当一个身份认证模块被附加到一个请求上，在设置request期间就会调用该模块。因此 `__call__` 方法必须完成使得身份认证生效的所有事情。一些身份认证形式会额外地添加钩子来提供进一步的功能。

可以在 Requests organization [https://github.com/requests] 页面的 `auth.py` 文件中找到示例。

常见问题

这部分的文档回答了有关requests的常见问题。

已编码的数据?

Requests 自动解压缩的gzip编码的响应体，并在可能的情况下尽可能的将响应内容解码为unicode.

如果需要的话，你可以直接访问原始响应内容（甚至是套接字）。

自定义 User-Agents?

Requests 允许你使用其它的HTTP Header来轻松的覆盖自带的User-Agent字符串。

怎么不是Httplib2?

Chris Adams 给出来一个很好的总结 Hacker News [http://news.ycombinator.com/item?id=2884406]:

> httplib2 is part of why you should use requests: it's far more respectable as a client but not as well documented and it still takes way too much code for basic operations. I appreciate what httplib2 is trying to do, that there's a ton of hard low-level annoyances in building a modern HTTP client, but really, just use requests instead. Kenneth Reitz is very motivated and he gets the degree to which simple things should be simple whereas httplib2 feels more like an academic exercise than something people should use to build production systems[1].
>
> Disclosure: I'm listed in the requests AUTHORS file but can claim credit for, oh, about 0.0001% of the awesomeness.
>
> 1. http://code.google.com/p/httplib2/issues/detail?id=96 is a good example: an annoying bug which affect many people, there was a fix available for months, which worked great when I applied it in a fork and pounded a couple TB of data through it, but it took over a year to make it into trunk and even longer to make it onto PyPI where any other project which required " httplib2" would get the working version.

支持 Python 3 吗?

恭喜！回答是肯定的。下面是官方支持的python平台列表:

- Python 2.6
- Python 2.7
- Python 3.1
- Python 3.2
- Python 3.3
- PyPy 1.9

整合

ScraperWiki

ScraperWiki [https://scraperwiki.com/] is an excellent service that allows you to run Python, Ruby, and PHP scraper scripts on the web. Now, Requests v0.6.1 is available to use in your scrapers!

To give it a try, simply:

```
import requests
```

Python for iOS

Requests is built into the wonderful Python for iOS [https://itunes.apple.com/us/app/python-2.7-for-ios/id485729872?mt=Python8] runtime!

To give it a try, simply:

```
import requests
```

Articles & Talks

- Python for the Web [http://gun.io/blog/python-for-the-web/] teaches how to use Python to interact with the web, using Requests.
- Daniel Greenfield's Review of Requests [http://pydanny.blogspot.com/2011/05/python-http-requests-for-humans.html]
- My 'Python for Humans' talk [http://python-for-humans.heroku.com] (audio [http://codeconf.s3.amazonaws.com/2011/pycodeconf/talks/PyCodeConf2011%20-%20Kenneth%20Reitz.m4a])
- Issac Kelly's 'Consuming Web APIs' talk [http://issackelly.github.com/Consuming-Web-APIs-with-Python-Talk/slides/slides.html]
- Blog post about Requests via Yum [http://arunsag.wordpress.com/2011/08/17/new-package-python-requests-http-for-humans/]
- Russian blog post introducing Requests [http://habrahabr.ru/blogs/python/126262/]
- French blog post introducing Requests [http://www.nicosphere.net/requests-urllib2-de-python-simplifie-2432/]

支持

对于Requests，如果你有疑问或者是问题的话，下面几种方法能解决：

发送推文

如果你的问题在140个字符内描述，欢迎在twitter上发送推文至 @kennethreitz [http://twitter.com/kennethreitz].

修复问题

如果你在Requests中注意到了一些意想不到的行为，或者希望看到一个新的功能支持，请查看 file an issue on GitHub [https://github.com/kennethreitz/requests/issues].

邮件

我更乐意回答关于Requests的个人或者更深入的问题。 Feel free to email requests@kennethreitz.com.

IRC

Requests 的官方Freenode 频道是 #python-requests

在 Freenode 上，你也能通过 **kennethreitz** 找到我。

更新

如果你想和社区以及开发版的 Requests 保持最新的联系， 这有几种方式:

GitHub

最好的方式是追踪Requests开发版本的 GitHub 库 [https://github.com/kennethreitz/requests].

Twitter

我经常推送关于Requests的新功能和新版本.

Follow @kennethreitz [https://twitter.com/kennethreitz] for updates.

邮件列表

对于 Requests 的讨论有一个小容量的邮件列表。发送邮件到 requests@librelist.org 订阅邮件。

开发者接口

这部分文档包含了Requests所有的接口。对于Requests依赖的外部库部分，我们介绍 一些比较重要的并提供规范文档的链接。

主要接口

Requests所有的功能都可以通过以下7个方法访问。 它们全部都会返回 **Response** 对象的一个实例。

`requests.`**`request`**(*method*, *url*, ***kwargs*)

Constructs and sends a **Request**. Returns **Response** object.

参数:
- **method** – method for the new **Request** object.
- **url** – URL for the new **Request** object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the **Request**.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- **headers** – (optional) Dictionary of HTTP Headers to send with the **Request**.
- **cookies** – (optional) Dict or CookieJar object to send with the **Request**.
- **files** – (optional) Dictionary of 'name': file-like-objects (or {'name': ('filename', fileobj)}) for multipart encoding upload.
- **auth** – (optional) Auth tuple to enable Basic/Digest/Custom HTTP Auth.
- **timeout** – (optional) Float describing the timeout of the request.
- **allow_redirects** – (optional) Boolean. Set to True if POST/PUT/DELETE redirect following is allowed.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **verify** – (optional) if `True`, the SSL cert will be verified. A CA_BUNDLE path can also be provided.
- **stream** – (optional) if `False`, the response content will be immediately downloaded.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

Usage:

```
>>> import requests
>>> req = requests.request('GET', 'http://httpbin.org/get')
<Response [200]>
```

`requests.`**`head`**(*url*, ***kwargs*)

Sends a HEAD request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`get`**(*url*, ***kwargs*)

Sends a GET request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`post`**(*url*, *data=None*, ***kwargs*)

Sends a POST request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`put`**(*url*, *data=None*, ***kwargs*)

Sends a PUT request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`patch`**(*url*, *data=None*, ***kwargs*)

Sends a PATCH request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

`requests.`**`delete`**(*url*, ***kwargs*)

Sends a DELETE request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.

- ****kwargs** – Optional arguments that `request` takes.

较低级别的类

class `requests.`**`Request`**(*method=None*, *url=None*, *headers=None*, *files=None*, *data={}*, *params={}*, *auth=None*, *cookies=None*, *hooks=None*)

A user-created **`Request`** object.

Used to prepare a **`PreparedRequest`**, which is sent to the server.

参数:
- **method** – HTTP method to use.
- **url** – URL to send.
- **headers** – dictionary of headers to send.
- **files** – dictionary of {filename: fileobject} files to multipart upload.
- **data** – the body to attach the request. If a dictionary is provided, form-encoding will take place.
- **params** – dictionary of URL parameters to append to the URL.
- **auth** – Auth handler or (user, pass) tuple.
- **cookies** – dictionary or CookieJar of cookies to attach to this request.
- **hooks** – dictionary of callback hooks, for internal usage.

Usage:

```
>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> req.prepare()
<PreparedRequest [GET]>
```

`deregister_hook`(*event*, *hook*)

Deregister a previously registered hook. Returns True if the hook existed, False if not.

`prepare`()

Constructs a **`PreparedRequest`** for transmission and returns it.

`register_hook`(*event*, *hook*)

Properly register a hook.

class `requests.`**`Response`**

The **`Response`** object, which contains a server's response to an HTTP request.

`apparent_encoding`

The apparent encoding, provided by the lovely Charade library (Thanks, Ian!).

`content`

Content of the response, in bytes.

`cookies` = *None*

A CookieJar of Cookies the server sent back.

`elapsed` = *None*

The amount of time elapsed between sending the request and the arrival of the response (as a timedelta)

`encoding` = *None*

Encoding to decode with when accessing r.text.

`headers` = *None*

Case-insensitive Dictionary of Response Headers. For example, `headers['content-encoding']` will return the value of a `'Content-Encoding'` response header.

`history` = *None*

A list of **`Response`** objects from the history of the Request. Any redirect responses will end up here. The list is sorted from the oldest to the most recent request.

`iter_content`(*chunk_size=1*, *decode_unicode=False*)

Iterates over the response data. This avoids reading the content at once into memory for large responses. The chunk size is the number of bytes it should read into memory. This is not necessarily the length of each item returned as decoding can take place.

`iter_lines`(*chunk_size=512*, *decode_unicode=None*)

Iterates over the response data, one line at a time. This avoids reading the content at once into memory for large responses.

`json`(***kwargs*)

Returns the json-encoded content of a response, if any.

参数: ****kwargs** – Optional arguments that `json.loads` takes.

links
Returns the parsed header links of the response, if any.

raise_for_status()
Raises stored **HTTPError**, if one occurred.

raw = *None*
File-like object representation of response (for advanced usage). Requires that ``stream=True` on the request.

status_code = *None*
Integer Code of responded HTTP Status.

text
Content of the response, in unicode.

if Response.encoding is None and chardet module is available, encoding will be guessed.

url = *None*
Final URL location of Response.

请求会话

class requests.**Session**
A Requests session.

Provides cookie persistience, connection-pooling, and configuration.

Basic Usage:

```
>>> import requests
>>> s = requests.Session()
>>> s.get('http://httpbin.org/get')
200
```

auth = *None*
Default Authentication tuple or object to attach to **Request**.

cert = *None*
SSL certificate default.

close()
Closes all adapters and as such the session

delete(*url*, ***kwargs*)
Sends a DELETE request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- ****kwargs** – Optional arguments that `request` takes.

get(*url*, ***kwargs*)
Sends a GET request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- ****kwargs** – Optional arguments that `request` takes.

get_adapter(*url*)
Returns the appropriate connnection adapter for the given URL.

head(*url*, ***kwargs*)
Sends a HEAD request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- ****kwargs** – Optional arguments that `request` takes.

headers = *None*
A case-insensitive dictionary of headers to be sent on each **Request** sent from this **Session**.

hooks = *None*
Event-handling hooks.

max_redirects = *None*

Maximum number of redirects to follow.

mount(*prefix*, *adapter*)
Registers a connection adapter to a prefix.

options(*url*, ***kwargs*)
Sends a OPTIONS request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- ****kwargs** – Optional arguments that `request` takes.

params = *None*
Dictionary of querystring data to attach to each **Request**. The dictionary values may be lists for representing multivalued query parameters.

patch(*url*, *data=None*, ***kwargs*)
Sends a PATCH request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

post(*url*, *data=None*, ***kwargs*)
Sends a POST request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

proxies = *None*
Dictionary mapping protocol to the URL of the proxy (e.g. {'http': 'foo.bar:3128'}) to be used on each **Request**.

put(*url*, *data=None*, ***kwargs*)
Sends a PUT request. Returns **Response** object.

参数:
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

resolve_redirects(*resp*, *req*, *stream=False*, *timeout=None*, *verify=True*, *cert=None*, *proxies=None*)
Receives a Response. Returns a generator of Responses.

send(*request*, ***kwargs*)
Send a given PreparedRequest.

stream = *None*
Stream response content default.

trust_env = *None*
Should we trust the environment?

verify = *None*
SSL Verification default.

异常

exception `requests`.**RequestException**
There was an ambiguous exception that occurred while handling your request.

exception `requests`.**ConnectionError**
A Connection error occurred.

exception `requests`.**HTTPError**(**args*, ***kwargs*)
An HTTP error occurred.

exception `requests`.**URLRequired**
A valid URL is required to make a request.

exception `requests`.**TooManyRedirects**
Too many redirects.

状态码查询

requests.**codes**()

Dictionary lookup object.

```
>>> requests.codes['temporary_redirect']
307

>>> requests.codes.teapot
418

>>> requests.codes['\o/']
200
```

Cookies

编码

类

class requests.**Response**

The **Response** object, which contains a server's response to an HTTP request.

apparent_encoding

The apparent encoding, provided by the lovely Charade library (Thanks, Ian!).

content

Content of the response, in bytes.

cookies = *None*

A CookieJar of Cookies the server sent back.

elapsed = *None*

The amount of time elapsed between sending the request and the arrival of the response (as a timedelta)

encoding = *None*

Encoding to decode with when accessing r.text.

headers = *None*

Case-insensitive Dictionary of Response Headers. For example, `headers['content-encoding']` will return the value of a `'Content-Encoding'` response header.

history = *None*

A list of **Response** objects from the history of the Request. Any redirect responses will end up here. The list is sorted from the oldest to the most recent request.

iter_content(*chunk_size=1*, *decode_unicode=False*)

Iterates over the response data. This avoids reading the content at once into memory for large responses. The chunk size is the number of bytes it should read into memory. This is not necessarily the length of each item returned as decoding can take place.

iter_lines(*chunk_size=512*, *decode_unicode=None*)

Iterates over the response data, one line at a time. This avoids reading the content at once into memory for large responses.

json(***kwargs*)

Returns the json-encoded content of a response, if any.

参数： ****kwargs** – Optional arguments that `json.loads` takes.

links

Returns the parsed header links of the response, if any.

raise_for_status()

Raises stored **HTTPError**, if one occurred.

raw = *None*

File-like object representation of response (for advanced usage). Requires that ``stream=True` on the request.

status_code = *None*

Integer Code of responded HTTP Status.

text

Content of the response, in unicode.

if Response.encoding is None and chardet module is available, encoding will be guessed.

url = *None*
Final URL location of Response.

class requests.**Request**(*method=None*, *url=None*, *headers=None*, *files=None*, *data={}*, *params={}*, *auth=None*, *cookies=None*, *hooks=None*)
A user-created **Request** object.

Used to prepare a **PreparedRequest**, which is sent to the server.

参数:
- **method** – HTTP method to use.
- **url** – URL to send.
- **headers** – dictionary of headers to send.
- **files** – dictionary of {filename: fileobject} files to multipart upload.
- **data** – the body to attach the request. If a dictionary is provided, form-encoding will take place.
- **params** – dictionary of URL parameters to append to the URL.
- **auth** – Auth handler or (user, pass) tuple.
- **cookies** – dictionary or CookieJar of cookies to attach to this request.
- **hooks** – dictionary of callback hooks, for internal usage.

Usage:

```
>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> req.prepare()
<PreparedRequest [GET]>
```

deregister_hook(*event*, *hook*)
Deregister a previously registered hook. Returns True if the hook existed, False if not.

prepare()
Constructs a **PreparedRequest** for transmission and returns it.

register_hook(*event*, *hook*)
Properly register a hook.

class requests.**PreparedRequest**
The fully mutable **PreparedRequest** object, containing the exact bytes that will be sent to the server.

Generated from either a **Request** object or manually.

Usage:

```
>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> r = req.prepare()
<PreparedRequest [GET]>

>>> s = requests.Session()
>>> s.send(r)
<Response [200]>
```

body = *None*
request body to send to the server.

deregister_hook(*event*, *hook*)
Deregister a previously registered hook. Returns True if the hook existed, False if not.

headers = *None*
dictionary of HTTP headers.

hooks = *None*
dictionary of callback hooks, for internal usage.

method = *None*
HTTP verb to send to the server.

path_url
Build the path URL to use.

prepare_auth(*auth*)
Prepares the given HTTP auth data.

prepare_body(*data*, *files*)

Prepares the given HTTP body data.

prepare_cookies(*cookies*)
Prepares the given HTTP cookie data.

prepare_headers(*headers*)
Prepares the given HTTP headers.

prepare_hooks(*hooks*)
Prepares the given hooks.

prepare_method(*method*)
Prepares the given HTTP method.

prepare_url(*url*, *params*)
Prepares the given HTTP URL.

register_hook(*event*, *hook*)
Properly register a hook.

url = *None*
HTTP URL to send the request to.

class requests.**Session**
A Requests session.

Provides cookie persistience, connection-pooling, and configuration.

Basic Usage:

```
>>> import requests
>>> s = requests.Session()
>>> s.get('http://httpbin.org/get')
200
```

auth = *None*
Default Authentication tuple or object to attach to Request.

cert = *None*
SSL certificate default.

close()
Closes all adapters and as such the session

delete(*url*, ***kwargs*)
Sends a DELETE request. Returns Response object.

参数:
- **url** – URL for the new Request object.
- ****kwargs** – Optional arguments that request takes.

get(*url*, ***kwargs*)
Sends a GET request. Returns Response object.

参数:
- **url** – URL for the new Request object.
- ****kwargs** – Optional arguments that request takes.

get_adapter(*url*)
Returns the appropriate connnection adapter for the given URL.

head(*url*, ***kwargs*)
Sends a HEAD request. Returns Response object.

参数:
- **url** – URL for the new Request object.
- ****kwargs** – Optional arguments that request takes.

headers = *None*
A case-insensitive dictionary of headers to be sent on each Request sent from this Session.

hooks = *None*
Event-handling hooks.

max_redirects = *None*

Maximum number of redirects to follow.

mount(*prefix*, *adapter*)
Registers a connection adapter to a prefix.

options(*url*, ***kwargs*)
Sends a OPTIONS request. Returns **Response** object.

参数：
- **url** – URL for the new **Request** object.
- ****kwargs** – Optional arguments that `request` takes.

params = *None*
Dictionary of querystring data to attach to each **Request**. The dictionary values may be lists for representing multivalued query parameters.

patch(*url*, *data=None*, ***kwargs*)
Sends a PATCH request. Returns **Response** object.

参数：
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

post(*url*, *data=None*, ***kwargs*)
Sends a POST request. Returns **Response** object.

参数：
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

proxies = *None*
Dictionary mapping protocol to the URL of the proxy (e.g. {'http': 'foo.bar:3128'}) to be used on each **Request**.

put(*url*, *data=None*, ***kwargs*)
Sends a PUT request. Returns **Response** object.

参数：
- **url** – URL for the new **Request** object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the **Request**.
- ****kwargs** – Optional arguments that `request` takes.

resolve_redirects(*resp*, *req*, *stream=False*, *timeout=None*, *verify=True*, *cert=None*, *proxies=None*)
Receives a Response. Returns a generator of Responses.

send(*request*, ***kwargs*)
Send a given PreparedRequest.

stream = *None*
Stream response content default.

trust_env = *None*
Should we trust the environment?

verify = *None*
SSL Verification default.

迁移到 1.x

本节详细介绍0.X和1.x的主要区别，减少升级带来的一些不便。

API 改变

- `Response.json` 现在是可调用的并且不再是响应体的属性。

```
import requests
r = requests.get('https://github.com/timeline.json')
r.json()   # This *call* raises an exception if JSON decoding fails
```

- `Session` API 也发生了变化. Sessions 对象不在需要参数了。 `Session` is also now capitalized,但为了向后兼容，它仍然能被小写的 `session` 实例化。

```
s = requests.Session()    # formerly, session took parameters
s.auth = auth
s.headers.update(headers)
r = s.get('http://httpbin.org/headers')
```

- 除了'response'，所有的请求挂钩已被移除。
- 认证助手已经被分解成单独的模块了. 参见 requests-oauthlib [https://github.com/requests/requests-oauthlib] and requests-kerberos [https://github.com/requests/requests-kerberos].
- 流请求的参数已从 `prefetch` 改变到 `stream`，并且逻辑也被颠倒。除此之外, `stream` 现在对于原始响应读取也是必需的。

```
# in 0.x, passing prefetch=False would accomplish the same thing
r = requests.get('https://github.com/timeline.json', stream=True)
r.raw.read(10)
```

- requests 方法的``config`` 参数已全部删除。 现在配置这些选项都在 `Session`，比如 keep-alive 和最大数目的重定向。 详细程度选项应当由配置日志来处理。

```
# Verbosity should now be configured with logging
my_config = {'verbose': sys.stderr}
requests.get('http://httpbin.org/headers', config=my_config)   # bad!
```

许可

有一个关键的与 API 无关的区别是开放许可从 ISC [http://opensource.org/licenses/ISC] 许可 变更到 Apache 2.0 [http://opensource.org/licenses/Apache-2.0] 许可. Apache 2.0 license 确保了对于requests的贡献也被涵盖在 Apache 2.0 许可内.

开发理念

Requests is an open but opinionated library, created by an open but opinionated developer.

Benevolent Dictator

Kenneth Reitz [http://kennethreitz.org] is the BDFL. He has final say in any decision related to Requests.

Values

- Simplicity is always better than functionality.
- Listen to everyone, then disregard it.
- The API is all that matters. Everything else is secondary.
- Fit the 90% use-case. Ignore the nay-sayers.

Semantic Versioning

For many years, the open source community has been plagued with version number dystonia. Numbers vary so greatly from project to project, they are practically meaningless.

Requests uses Semantic Versioning [http://semver.org]. This specification seeks to put an end to this madness with a small set of practical guidelines for you and your colleagues to use in your next project.

Standard Library?

Requests has no *active* plans to be included in the standard library. This decision has been discussed at length with Guido as well as numerous core developers.

Essentially, the standard library is where a library goes to die. It is appropriate for a module to be included when active development is no longer necessary.

Requests just reached v1.0.0. This huge milestone marks a major step in the right direction.

Linux Distro Packages

Distributions have been made for many Linux repositories, including: Ubuntu, Debian, RHEL, and Arch.

These distributions are sometimes divergent forks, or are otherwise not kept up-to-date with the latest code and bugfixes. PyPI (and its mirrors) and GitHub are the official distribution sources; alternatives are not supported by the requests project.

如何帮助

Requests is under active development, and contributions are more than welcome!

1. Check for open issues or open a fresh issue to start a discussion around a bug. There is a Contributor Friendly tag for issues that should be ideal for people who are not very familiar with the codebase yet.
2. Fork the repository [https://github.com/kennethreitz/requests] on Github andstart making your changes to a new branch.
3. Write a test which shows that the bug was fixed.
4. Send a pull request and bug the maintainer until it gets merged and published. :) Make sure to add yourself to AUTHORS [https://github.com/kennethreitz/requests/blob/master/AUTHORS.rst].

Feature Freeze

As of v1.0.0, Requests has now entered a feature freeze. Requests for new features and Pull Requests implementing those features will not be accepted.

Development Dependencies

You'll need to install py.test in order to run the Requests' test suite:

```
$ make test-deps
$ make test
py.test
platform darwin -- Python 2.7.3 -- pytest-2.3.4
collected 25 items

test_requests.py .........................
25 passed in 3.50 seconds
```

Runtime Environments

Requests currently supports the following versions of Python:

- Python 2.6
- Python 2.7
- Python 3.1
- Python 3.2
- Python 3.3
- PyPy 1.9

Support for Python 3.1 and 3.2 may be dropped at any time.

Google App Engine will never be officially supported. Pull requests for compatiblity will be accepted, as long as they don't complicate the codebase.

Are you crazy?

- SPDY support would be awesome. No C extensions.

贡献者

Requests is written and maintained by Kenneth Reitz and various contributors:

Development Lead

- Kenneth Reitz <me@kennethreitz.com>

Urllib3

- Andrey Petrov <andrey.petrov@shazow.net>

Patches and Suggestions

- Various Pocoo Members
- Chris Adams
- Flavio Percoco Premoli
- Dj Gilcrease
- Justin Murphy
- Rob Madole
- Aram Dulyan
- Johannes Gorset
- 村山めがね (Megane Murayama)
- James Rowe
- Daniel Schauenberg
- Zbigniew Siciarz
- Daniele Tricoli 'Eriol'
- Richard Boulton
- Miguel Olivares <miguel@moliware.com>
- Alberto Paro
- Jérémy Bethmont
- 潘旭 (Xu Pan)
- Tamás Gulácsi
- Rubén Abad
- Peter Manser
- Jeremy Selier
- Jens Diemer
- Alex <@alopatin>
- Tom Hogans <tomhsx@gmail.com>
- Armin Ronacher
- Shrikant Sharat Kandula
- Mikko Ohtamaa
- Den Shabalin
- Daniel Miller <danielm@vs-networks.com>
- Alejandro Giacometti
- Rick Mak
- Johan Bergström
- Josselin Jacquard
- Travis N. Vaught
- Fredrik Möllerstrand
- Daniel Hengeveld
- Dan Head
- Bruno Renié
- David Fischer
- Joseph McCullough
- Juergen Brendel
- Juan Riaza
- Ryan Kelly
- Rolando Espinoza La fuente
- Robert Gieseke
- Idan Gazit
- Ed Summers
- Chris Van Horne
- Christopher Davis
- Ori Livneh
- Jason Emerick
- Bryan Helmig
- Jonas Obrist
- Lucian Ursu
- Tom Moertel

- Frank Kumro Jr
- Chase Sterling
- Marty Alchin
- takluyver
- Ben Toews (mastahyeti)
- David Kemp
- Brendon Crawford
- Denis (Telofy)
- Cory Benfield (Lukasa)
- Matt Giuca
- Adam Tauber
- Honza Javorek
- Brendan Maguire <maguire.brendan@gmail.com>
- Chris Dary
- Danver Braganza <danverbraganza@gmail.com>
- Max Countryman
- Nick Chadwick
- Jonathan Drosdeck
- Jiri Machalek
- Steve Pulec
- Michael Kelly
- Michael Newman <newmaniese@gmail.com>
- Jonty Wareing <jonty@jonty.co.uk>
- Shivaram Lingamneni
- Miguel Turner
- Rohan Jain (crodjer)
- Justin Barber <barber.justin@gmail.com>
- Roman Haritonov <@reclosedev>
- Josh Imhoff <joshimhoff13@gmail.com>
- Arup Malakar <amalakar@gmail.com>
- Danilo Bargen (dbrgn)
- Torsten Landschoff
- Michael Holler (apotheos)
- Timnit Gebru
- Sarah Gonzalez
- Victoria Mo
- Leila Muhtasib
- Matthias Rahlf <matthias@webding.de>
- Jakub Roztocil <jakub@roztocil.name>
- Ian Cordasco <graffatcolmingov@gmail.com> @sigmavirus24
- Rhys Elsmore
- André Graf (dergraf)
- Stephen Zhuang (everbird)
- Martijn Pieters
- Jonatan Heyman
- David Bonner <dbonner@gmail.com> @rascalking
- Vinod Chandru
- Johnny Goodnow <j.goodnow29@gmail.com>
- Denis Ryzhkov <denisr@denisr.com>
- Wilfred Hughes <me@wilfred.me.uk> @dontYetKnow
- Dmitry Medvinsky <me@dmedvinsky.name>

Python 模块索引

r

r

requests

requests.models

索引

A | B | C | D | E | G | H | I | J | L | M | O | P | R | S | T | U | V

A

apparent_encoding (requests.Response 属性), [1]
auth (requests.Session 属性), [1]

B

body (requests.PreparedRequest 属性)

C

cert (requests.Session 属性), [1]
close() (requests.Session 方法), [1]
codes() (在 requests 模块中)
ConnectionError
content (requests.Response 属性), [1]
cookies (requests.Response 属性), [1]

D

delete() (requests.Session 方法), [1]
    (在 requests 模块中)
deregister_hook() (requests.PreparedRequest 方法)
    (requests.Request 方法), [1]

E

elapsed (requests.Response 属性), [1]
encoding (requests.Response 属性), [1]

G

get() (requests.Session 方法), [1]
    (在 requests 模块中)
get_adapter() (requests.Session 方法), [1]

H

head() (requests.Session 方法), [1]
    (在 requests 模块中)
headers (requests.PreparedRequest 属性)
    (requests.Response 属性), [1]
    (requests.Session 属性), [1]
history (requests.Response 属性), [1]
hooks (requests.PreparedRequest 属性)
    (requests.Session 属性), [1]
HTTPError

I

iter_content() (requests.Response 方法), [1]
iter_lines() (requests.Response 方法), [1]

J

json() (requests.Response 方法), [1]

L

links (requests.Response 属性), [1]

M

max_redirects (requests.Session 属性), [1]
method (requests.PreparedRequest 属性)
mount() (requests.Session 方法), [1]

O

options() (requests.Session 方法), [1]

P

params (requests.Session 属性), [1]
patch() (requests.Session 方法), [1]
    (在 requests 模块中)
path_url (requests.PreparedRequest 属性)
post() (requests.Session 方法), [1]
    (在 requests 模块中)
prepare() (requests.Request 方法), [1]
prepare_auth() (requests.PreparedRequest 方法)
prepare_body() (requests.PreparedRequest 方法)
prepare_cookies() (requests.PreparedRequest 方法)
prepare_headers() (requests.PreparedRequest 方法)
prepare_hooks() (requests.PreparedRequest 方法)
prepare_method() (requests.PreparedRequest 方法)
prepare_url() (requests.PreparedRequest 方法)
PreparedRequest (requests 中的类)
proxies (requests.Session 属性), [1]
put() (requests.Session 方法), [1]
    (在 requests 模块中)
Python 提高建议
    PEP 20

R

raise_for_status() (requests.Response 方法), [1]
raw (requests.Response 属性), [1]
register_hook() (requests.PreparedRequest 方法)
    (requests.Request 方法), [1]
Request (requests 中的类), [1]
request() (在 requests 模块中)
RequestException
requests (模块), [1]
requests.models (模块)
resolve_redirects() (requests.Session 方法), [1]
Response (requests 中的类), [1]

S

send() (requests.Session 方法), [1]
Session (requests 中的类), [1]
status_code (requests.Response 属性), [1]
stream (requests.Session 属性), [1]

T

text (requests.Response 属性), [1]
TooManyRedirects
trust_env (requests.Session 属性), [1]

U

url (requests.PreparedRequest 属性)
    (requests.Response 属性), [1]
URLRequired

V

verify (requests.Session 属性), [1]