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://api.github.com/events')
```

现在，我们有一个名为 `r` 的 `Response` 对象。我们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', data =
{'key':'value'})
```

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', data =
{'key':'value'})

>>> 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。

传递 URL 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，以一个字符串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。

你还可以将一个列表作为值传入：

```
>>> payload = {'key1': 'value1', 'key2': ['value2', 'value3']}

>>> r = requests.get('http://httpbin.org/get', params=payload)
>>> print(r.url)
http://httpbin.org/get?key1=value1&key2=value2&key2=value3
```

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://api.github.com/events')
>>> 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 io import BytesIO


>>> i = Image.open(BytesIO(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://api.github.com/events')
>>> 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，响应内容是 401 (Unauthorized)，尝试访问 `r.json()` 将会抛出 `ValueError: No JSON object could be decoded` 异常。

需要注意的是，成功调用 `r.json()` 并**不**意味着响应的成功。有的服务器会在失败的响应中包含一个 JSON 对象（比如 HTTP 500 的错误细节）。这种 JSON 会被解码返回。要检查请求是否成功，请使用 `r.raise_for_status()` 或者检查 `r.status_code` 是否和你的期望相同。

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`。具体你可以这么做：

```
>>> r = requests.get('https://api.github.com/events', 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，你应该以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。 Note that `chunk_size` can be freely adjusted to a number that may better fit your use cases.

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:

```
>>> url = 'https://api.github.com/some/endpoint'
>>> headers = {'user-agent': 'my-app/0.0.1'}

>>> r = requests.get(url, headers=headers)
```

注意：定制 header 的优先级低于某些特定的信息源，例如：

- 如果在 `.netrc` 中设置了用户认证信息，使用 *headers=* 设置的授权就不会生效。而如果设置了 `auth=` 参数，``.netrc`` 的设置就无效了。
- 如果被重定向到别的主机，授权 header 就会被删除。
- 代理授权 header 会被 URL 中提供的代理身份覆盖掉。
- 在我们能判断内容长度的情况下，header 的 Content-Length 会被改写。

更进一步讲，Requests 不会基于定制 header 的具体情况改变自己的行为。只不过在最后的请求中，所有的 header 信息都会被传递进去。

注意：所有的 header 值必须是 `string`、bytestring 或者 unicode。尽管传递 unicode header 也是允许的，但不建议这样做。

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"
  },
  ...
}
```

你还可以为 `data` 参数传入一个元组列表。在表单中多个元素使用同一 key 的时候，这种方式尤其有效：

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

```
  "form": {
    "key1": [
      "value1",
      "value2"
    ]
  },
  ...
}
```

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))
```

此处除了可以自行对 `dict` 进行编码，你还可以使用 `json` 参数直接传递，然后它就会被自动编码。这是 2.4.2 版的新加功能：

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

>>> r = requests.post(url, json=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，你可能希望将请求做成数据流。默认下 `requests` 不支持，但有个第三方包 `requests-toolbelt` 是支持的。你可以阅读 toolbelt 文档 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。

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 可能会试图为你提供 `Content-Length` header，在它这样做的时候，这个值会被设为文件的字节数（*bytes*）。如果用文本模式(text mode)打开文件，就可能会发生错误。

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
```

如果发送了一个错误请求(一个 4XX 客户端错误，或者 5XX 服务器错误响应)，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 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'
```

它还有一个特殊点，那就是服务器可以多次接受同一 header，每次都使用不同的值。但 Requests 会将它们合并，这样它们就可以用一个映射来表示出来，参见 RFC 7230:

A recipient MAY combine multiple header fields with the same field name into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field value to the combined field value in order, separated by a comma.

接收者可以合并多个相同名称的 header 栏位，把它们合为一个 "field-name: field-value" 配对，将每个后续的栏位值依次追加到合并的栏位值中，用逗号隔开即可，这样做不会改变信息的语义。

Cookie

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"}}'
```

Cookie 的返回对象为 `RequestsCookieJar`，它的行为和字典类似，但接口更为完整，适合跨域名跨路径使用。你还可以把 Cookie Jar 传到 Requests 中：

```
>>> jar = requests.cookies.RequestsCookieJar()
>>> jar.set('tasty_cookie', 'yum', domain='httpbin.org',
path='/cookies')
>>> jar.set('gross_cookie', 'blech', domain='httpbin.org',
path='/elsewhere')
>>> url = 'http://httpbin.org/cookies'
>>> r = requests.get(url, cookies=jar)
>>> r.text
'{"cookies": {"tasty_cookie": "yum"}}'
```

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 `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
[]
```

如果你使用了 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。基本上所有的生产代码都应该使用这一参数。如果不使用，你的程序可能会永远失去响应：

```
>>> 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)
```

注意

`timeout` 仅对连接过程有效，与响应体的下载无关。 `timeout` 并不是整个下载响应的时间限制，而是如果服务器在 `timeout` 秒内没有应答，将会引发一个异常（更精确地说，是在 `timeout` 秒内没有从基础套接字上接收到任何字节的数据时）If no timeout is specified explicitly, requests do not time out.

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。

如果 HTTP 请求返回了不成功的状态码， `Response.raise_for_status()` 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` 。