request 源码浅析

author: https://github.com/Ciyfly

版本是 request-2.22.0
init 初始化 一些信息 将各个包都导入

主要是这一句

```
from .api import request, get, head, post, patch, put, delete, options
```

这是从api 那里导入了 这么多的功能

api中导出的这些都是基于 request 方法的
get head等是对他的方法的封装
从其他模块导入到 `__init__` 这样直接就能 `from request import` 了
是在**api.py**下的
request方法 注释 参数很详细

request方法参数

```python
def request(method, url, **kwargs):
    """Constructs and sends a :class:`Request <Request>`.

    :param method: method for the new :class:`Request` object.
    :param url: URL for the new :class:`Request` object.
    :param params: (optional) Dictionary, list of tuples or bytes to send
        in the query string for the :class:`Request`.
    :param data: (optional) Dictionary, list of tuples, bytes, or file-like
        object to send in the body of the :class:`Request`.
    :param json: (optional) A JSON serializable Python object to send in the body of the :class:
    :param headers: (optional) Dictionary of HTTP Headers to send with the :class:`Request`.
    :param cookies: (optional) Dict or CookieJar object to send with the :class:`Request`.
    :param files: (optional) Dictionary of ``'name': file-like-objects`` (or ``{'name': file-tup
        ``file-tuple`` can be a 2-tuple ``('filename', fileobj)``, 3-tuple ``('filename', fileob
        or a 4-tuple ``('filename', fileobj, 'content_type', custom_headers)``, where ``'content
        defining the content type of the given file and ``custom_headers`` a dict-like object co
        to add for the file.
    :param auth: (optional) Auth tuple to enable Basic/Digest/Custom HTTP Auth.
    :param timeout: (optional) How many seconds to wait for the server to send data
        before giving up, as a float, or a :ref:`(connect timeout, read
        timeout) <timeouts>` tuple.
    :type timeout: float or tuple
    :param allow_redirects: (optional) Boolean. Enable/disable GET/OPTIONS/POST/PUT/PATCH/DELETE
    :type allow_redirects: bool
    :param proxies: (optional) Dictionary mapping protocol to the URL of the proxy.
    :param verify: (optional) Either a boolean, in which case it controls whether we verify
            the server's TLS certificate, or a string, in which case it must be a path
            to a CA bundle to use. Defaults to ``True``.
    :param stream: (optional) if ``False``, the response content will be immediately downloaded.
    :param cert: (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', '
    :return: :class:`Response <Response>` object
    :rtype: requests.Response
    Usage::

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

    # By using the 'with' statement we are sure the session is closed, thus we
    # avoid leaving sockets open which can trigger a ResourceWarning in some
    # cases, and look like a memory leak in others.
    with sessions.Session() as session:
        return session.request(method=method, url=url, **kwargs)
```

特别是有个json 可以 直接 json=data 这样就发送的就是直接将data转为json发送了

比如说 get 就是对rquest的method进行了封装

get方法参数

```
def get(self, url, **kwargs):
    r"""Sends a GET request. Returns :class:`Response` object.

    :param url: URL for the new :class:`Request` object.
    :param \*\*kwargs: Optional arguments that ``request`` takes.
    :rtype: requests.Response
    """

    kwargs.setdefault('allow_redirects', True)
    return self.request('GET', url, **kwargs)
```

根本还是 request里面的使用 session的request来创建

```
with sessions.Session() as session:
    return session.request(method=method, url=url, **kwargs)
```

打开一个会话 使用 with as 的方式
被with使用的里面一定写了 `__enter__` 和 `__exit__` 方法用来 提供给with自动关闭的

调用session.request方法传入对应值

那么我们这里先看下session这个对象

```
from . import sessions
```

是当前目录下的 `session.py`

首先Session 类

Session

在 340 行

```
class Session(SessionRedirectMixin):
    """A Requests session.

    Provides cookie persistence, connection-pooling, and configuration.

    Basic Usage::

      >>> import requests
      >>> s = requests.Session()
      >>> s.get('https://httpbin.org/get')
      <Response [200]>

    Or as a context manager::

      >>> with requests.Session() as s:
      >>>     s.get('https://httpbin.org/get')
      <Response [200]>
    """

    __attrs__ = [
        'headers', 'cookies', 'auth', 'proxies', 'hooks', 'params', 'verify',
        'cert', 'prefetch', 'adapters', 'stream', 'trust_env',
        'max_redirects',
    ]
```

这个用于提供cookie持久性、连接池和配置
都是实例化 Session来发起请求的

设置 包含哪些属性 `__attrs__`

接下来我们看 `__init__` 初始化方法

比较多

```
# 设置默认清头
self.headers = default_headers()
# default_headers
return CaseInsensitiveDict({
    'User-Agent': default_user_agent(),
    'Accept-Encoding': ', '.join(('gzip', 'deflate')),
    'Accept': '*/*',
    'Connection': 'keep-alive',
})

# 这里可以看到 返回的头部字典是一个 CaseInsensitiveDict 类型
# 这是一个不区分大小写的字典
# 实现了字典的所有方式 所有键都应该是字符串
# iter(instance), keys(), items(), iterkeys(), iteritems()都会返回原有的信息（即大小写敏感）, 而查询
# 详细的分析可以看这篇 https://zhuanlan.zhihu.com/p/53138686
# 这个就是如果直接str() 把request头转字符串会报错的问题根本原因


# 默认的身份验证
self.auth = None


# 代理
self.proxies = {}


# 事件动作钩子
self.hooks = default_hooks()

# 在./hooks.py
HOOKS = ['response']
def default_hooks():
    return {event: [] for event in HOOKS}
# 对响应hooks


# 参数
self.params = {}


# 默认响应流
self.stream = False


# 默认是对ssl验证
self.verify = True


# 默认证书
self.cert = None
```

```
# 允许的最大重定向数  30
self.max_redirects = DEFAULT_REDIRECT_LIMIT


# 代理的身份验证
self.trust_env = True


# cookiejar 对cookie的管理
# RequestsCookieJar
self.cookies = cookiejar_from_dict({})


# 默认适配器 是一个有序字典
self.adapters = OrderedDict()


# 将连接适配器注册到前缀适配器按前缀长度降序排序
#
self.mount('https://', HTTPAdapter())
self.mount('http://', HTTPAdapter())
```

重写了 **enter** 方法 和 __exit__方法
这样可用于 with open 来操作 也就是自动关闭

```
def __enter__(self):
    return self

def __exit__(self, *args):
    self.close()
```

以上就是 **init**
接下来

如果我们正常一个get请求是怎样的

```
def get(self, url, **kwargs):
    r"""Sends a GET request. Returns :class:`Response` object.

    :param url: URL for the new :class:`Request` object.
    :param \*\*kwargs: Optional arguments that ``request`` takes.
    :rtype: requests.Response
    """

    kwargs.setdefault('allow_redirects', True)
    return self.request('GET', url, **kwargs)
```

其中 设置 `kwargs.setdefault('allow_redirects', True)`
设置默认allow_redirects 为True 即默认运行重定向
返回 self.request 跟入

request 参数很多 如下:

```
def request(self, method, url,
        params=None, data=None, headers=None, cookies=None, files=None,
        auth=None, timeout=None, allow_redirects=True, proxies=None,
        hooks=None, stream=None, verify=None, cert=None, json=None):
```

如下注释

还可以看文档 https://2.python-requests.org//zh_CN/latest/api.html

```
"""
:param method: 请求模式 get post等
:param url: url
:param params: (可选)要在查询中发送的字典或字节
:param data:(可选)字典、元组列表、字节或类似文件
:param json:(可选) 发送的json
:param headers: (可选)字典请求头
:param cookies: (可选)cookie
:param files:(可选) "filename"字典: 类文件对象``
:param auth: (可选)) 验证元组或可调用以启用 基本/摘要/自定义http身份验证。
:param timeout: (可选) 超时时间 float
:param allow_redirects: (可选) 允许重定向 默认允许
:param proxies: (可选) 代理
:param stream: (可选) 默认为True, 获取body立即下载开关, 为False会立即下载响应头和响应体; 为True时会先
:param verify: (可选) 默认为True, 认证SSL证书开关; 为True时会验证SSL证书, 也可以使用cert参数提供一个C

:rtype: requests.Response
"""
```

创建一个 Request对象

```
req = Request(
    method=method.upper(),
    url=url,
    headers=headers,
    files=files,
    data=data or {},
    json=json,
    params=params or {},
    auth=auth,
    cookies=cookies,
    hooks=hooks,
)
```

Request

Request 是从 models导入的

`from .models import Request, PreparedRequest, DEFAULT_REDIRECT_LIMIT`

我们跟入查看下
在 198行位置

参数就是上面session中那些传入的参数

对于直接打印输出的是在这

```
    def __repr__(self):
        return '<Request [%s]>' % (self.method)
```

在session的request方法中 是直接获取初始化的request对象
后调用 prepare_request方法

`prep = self.prepare_request(req)`

这个方法会返回 requests.PreparedRequest
正常来说直接调用 Request的 prepare方法也是返回 PreparedRequest实例的对象
那为啥又要回到session创建呢
继续往下看
添加cookie, auth

然后创建 PreparedRequest 实例
区别是在创建的时候使用了方法 `merge_setting` 对一些参数处理

即是把 request的和session的参数合并到一起再创建这个 PreparedRequest实例

models下的Request 的创建 `PreparedRequest` 实例

```
def prepare(self):
    """Constructs a :class:`PreparedRequest <PreparedRequest>` for transmission and returns it."
    p = PreparedRequest()
    p.prepare(
        method=self.method,
        url=self.url,
        headers=self.headers,
        files=self.files,
        data=self.data,
        json=self.json,
        params=self.params,
        auth=self.auth,
        cookies=self.cookies,
        hooks=self.hooks,
    )
    return p
```

session下的的=创建 `PreparedRequest` 实例

```
p = PreparedRequest()
p.prepare(
    method=request.method.upper(),
    url=request.url,
    files=request.files,
    data=request.data,
    json=request.json,
    headers=merge_setting(request.headers, self.headers, dict_class=CaseInsensitiveDict),
    params=merge_setting(request.params, self.params),
    auth=merge_setting(auth, self.auth),
    cookies=merged_cookies,
    hooks=merge_hooks(request.hooks, self.hooks),
)
return p
```

为什么没有直接一起传入参数直接创建返回一个实例而是创建完的
reques再合并session的数据后再创建

主要的类就是这个了 `PreparedRequest` 是一个准备的请求类

初始化就是对之前传入的参数进行初始化
在 models的 272行

```python
class PreparedRequest(RequestEncodingMixin, RequestHooksMixin):
    """The fully mutable :class:`PreparedRequest <PreparedRequest>` object,
    containing the exact bytes that will be sent to the server.

    Generated from either a :class:`Request <Request>` object or manually.

    Usage::

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

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

返回 PreparedRequest 实例后
更新要发送的数据字典并调用 send方法
跟入send方法 (发送 PreparedRequest)
各种获取request的参数值

```
def send(self, request, **kwargs):
    """Send a given PreparedRequest.

    :rtype: requests.Response
    """
    # Set defaults that the hooks can utilize to ensure they always have
    # the correct parameters to reproduce the previous request.
    kwargs.setdefault('stream', self.stream)
    kwargs.setdefault('verify', self.verify)
    kwargs.setdefault('cert', self.cert)
    kwargs.setdefault('proxies', self.proxies)

    # It's possible that users might accidentally send a Request object.
    # Guard against that specific failure case.
    if isinstance(request, Request):
        raise ValueError('You can only send PreparedRequests.')

    # Set up variables needed for resolve_redirects and dispatching of hooks
    allow_redirects = kwargs.pop('allow_redirects', True)
    stream = kwargs.get('stream')
    hooks = request.hooks

    # Get the appropriate adapter to use
    adapter = self.get_adapter(url=request.url)

    # Start time (approximately) of the request
    start = preferred_clock()

    # Send the request
    r = adapter.send(request, **kwargs)
```

对于 start 即请求的开始时间如下获取

```
if sys.platform == 'win32':
    try:  # Python 3.4+
        preferred_clock = time.perf_counter
    except AttributeError:  # Earlier than Python 3.
        preferred_clock = time.clock
else:
    preferred_clock = time.time
```

然后调用 发起请求

```
r = adapter.send(request, **kwargs)
```

get_adapter 返回的是 adapters.BaseAdapter

调用的send方法就是 这个

```
class BaseAdapter(object):
    """The Base Transport Adapter"""

    def __init__(self):
        super(BaseAdapter, self).__init__()

    def send(self, request, stream=False, timeout=None, verify=True,
             cert=None, proxies=None):
        """Sends PreparedRequest object. Returns Response object.

        :param request: The :class:`PreparedRequest <PreparedRequest>` being sent.
        :param stream: (optional) Whether to stream the request content.
        :param timeout: (optional) How long to wait for the server to send
            data before giving up, as a float, or a :ref:`(connect timeout,
            read timeout) <timeouts>` tuple.
        :type timeout: float or tuple
        :param verify: (optional) Either a boolean, in which case it controls whether we verify
            the server's TLS certificate, or a string, in which case it must be a path
            to a CA bundle to use
        :param cert: (optional) Any user-provided SSL certificate to be trusted.
        :param proxies: (optional) The proxies dictionary to apply to the request.
        """
        raise NotImplementedError
```

调用的是这个发送

但是如果是基类的话怎么有这个send方法呢

那就要往回看
设置默认的适配器

```
# Default connection adapters.
self.adapters = OrderedDict()
self.mount('https://', HTTPAdapter())
self.mount('http://', HTTPAdapter())
```

对于前缀是如上的对应值为 HTTPAdapter 适配器
这个适配器是继承的 BaseAdapter 适配器

所以 get_adapter 方法 返回的适配器即是 HTTPAdapter

```
def get_adapter(self, url):
    """
    Returns the appropriate connection adapter for the given URL.

    :rtype: requests.adapters.BaseAdapter
    """
    for (prefix, adapter) in self.adapters.items():

        if url.lower().startswith(prefix.lower()):
            return adapter

    # Nothing matches :-/
    raise InvalidSchema("No connection adapters were found for '%s'" % url)
```

使用适配器的概念，匹配前缀url来使用对于的适配器来操作

这样 我们就要看 HTTPAdapter

```
>>> import requests
>>> s = requests.Session()
>>> a = requests.adapters.HTTPAdapter(max_retries=3)
>>> s.mount('http://', a)
```

可以看到使用就是这样的

可以看到可以获取的几个属性

```
    __attrs__ = ['max_retries', 'config', '_pool_connections', '_pool_maxsize',
                 '_pool_block']
```

初始化可以看到是一个连接池

```
self.init_poolmanager(pool_connections, pool_maxsize, block=pool_block)
```

初始化urllib3池管理器

对urlli3进行了连接池的管理

创建 PoolManager

```
self.poolmanager = PoolManager(num_pools=connections, maxsize=maxsize,block=block, strict=True,
```

这个 PoolManager 是urllib3中的

那么继续看下send方法

```
def send(self, request, stream=False, timeout=None, verify=True, cert=None, proxies=None):
    """Sends PreparedRequest object. Returns Response object.

    :param request: The :class:`PreparedRequest <PreparedRequest>` being sent.
    :param stream: (optional) Whether to stream the request content.
    :param timeout: (optional) How long to wait for the server to send
        data before giving up, as a float, or a :ref:`(connect timeout,
        read timeout) <timeouts>` tuple.
    :type timeout: float or tuple or urllib3 Timeout object
    :param verify: (optional) Either a boolean, in which case it controls whether
        we verify the server's TLS certificate, or a string, in which case it
        must be a path to a CA bundle to use
    :param cert: (optional) Any user-provided SSL certificate to be trusted.
    :param proxies: (optional) The proxies dictionary to apply to the request.
    :rtype: requests.Response
    """
```

首先创建一个连接对象

```
try:
    conn = self.get_connection(request.url, proxies)
except LocationValueError as e:
    raise InvalidURL(e, request=request)
```

使用urllib3的poolmanager的 connection_from_url方法
返回一个conn

```
conn = self.poolmanager.connection_from_url(url)
```

证书 请求头添加

```
self.cert_verify(conn, request.url, verify, cert)
url = self.request_url(request, proxies)
self.add_headers(request, stream=stream, timeout=timeout, verify=verify, cert=cert, proxies=prox
```

```
chunked = not (request.body is None or 'Content-Length' in request.headers)
```

如果 请求的body为空或者 请求头中有 Content-Length字段则如下

```
resp = conn.urlopen(
    method=request.method,
    url=url,
    body=request.body,
    headers=request.headers,
    redirect=False,
    assert_same_host=False,
    preload_content=False,
    decode_content=False,
    retries=self.max_retries,
    timeout=timeout
)
```

返回resp对象

否则

```
low_conn = conn._get_conn(timeout=DEFAULT_POOL_TIMEOUT)

try:
    low_conn.putrequest(request.method,
                        url,
                        skip_accept_encoding=True)

    for header, value in request.headers.items():
        low_conn.putheader(header, value)

    low_conn.endheaders()
    # 对请求的body进行换行处理
    for i in request.body:
        low_conn.send(hex(len(i))[2:].encode('utf-8'))
        low_conn.send(b'\r\n')
        low_conn.send(i)
        low_conn.send(b'\r\n')
    low_conn.send(b'0\r\n\r\n')

    # Receive the response from the server
    try:
        # For Python 2.7, use buffering of HTTP responses
        r = low_conn.getresponse(buffering=True)
    except TypeError:
        # For compatibility with Python 3.3+
        r = low_conn.getresponse()

    resp = HTTPResponse.from_httplib(
        r,
        pool=conn,
        connection=low_conn,
        preload_content=False,
        decode_content=False
    )
```

接下来各种异常捕获

很详细可以快速判断问题所在

最后返回

```
return self.build_response(request, resp)
```

build_response即是对响应的封装
最后返回response对象

一个完整的get请求。

下面重新梳理下

api中对request的get post 等封装
request的实质是session.request
再往里是因为HTTPAdapter适配器
里面实质还是使用的urllib3的请求
对请求和响应做了很多封装
来回导入调用很多

还有很多地方模块细节没有细读 留作下次再深入一些读