RESTful架构和DRF进阶

django restful framework教程大全

阅读目录

回到顶部

一. 什么是RESTful

1. REST与技术无关，代表的是一种软件架构风格，REST是Representational State Transfer的简称，中文翻译为“表征状态转移”
2. REST从资源的角度类审视整个网络，它将分布在网络中某个节点的资源通过URL进行标识，客户端应用通过URL来获取资源的表征，获得这些表征致使这些应用转变状态
3. REST与技术无关，代表的是一种软件架构风格，REST是Representational State Transfer的简称，中文翻译为“表征状态转移”
4. 所有的数据，不过是通过网络获取的还是操作（增删改查）的数据，都是资源，将一切数据视为资源是REST区别与其他架构风格的最本质属性
5. 对于REST这种面向资源的架构风格，有人提出一种全新的结构理念，即：面向资源架构（ROA：Resource Oriented Architecture）

回到顶部

二. RESTful API设计

- API与用户的通信协议，总是使用HTTPs协议。
- 域名
  - https://api.example.com　　　　尽量将API部署在专用域名（会存在跨域问题）
  - https://example.org/api/　　　　API很简单
- 版本
  - URL，如：https://api.example.com/v1/
  - 请求头　　　　　　　　　跨域时，引发发送多次请求
- 路径，视网络上任何东西都是资源，均使用名词表示（可复数）
  - https://api.example.com/v1/zoos

    - https://api.example.com/v1/animals
    - https://api.example.com/v1/employees
- method
    - GET ：从服务器取出资源（一项或多项）
    - POST ：在服务器新建一个资源
    - PUT ：在服务器更新资源（客户端提供改变后的完整资源）
    - PATCH：在服务器更新资源（客户端提供改变的属性）
    - DELETE：从服务器删除资源
- 过滤，通过在url上传参的形式传递搜索条件
    - https://api.example.com/v1/zoos?limit=10：指定返回记录的数量
    - https://api.example.com/v1/zoos?offset=10：指定返回记录的开始位置
    - https://api.example.com/v1/zoos?page=2&per_page=100：指定第几页，以及每页的记录数
    - https://api.example.com/v1/zoos?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序
    - https://api.example.com/v1/zoos?animal_type_id=1：指定筛选条件
- 状态码

```
OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
Accepted - [*]：表示一个请求已经进入后台排队（异步任务）
NO CONTENT - [DELETE]：用户删除数据成功。
INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
Unauthorized - [*]：表示用户没有权限（令牌、用户名、密码错误）。
Forbidden - [*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。
Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。
Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
INTERNAL SERVER ERROR - [*]：服务器发生错误，用户将无法判断发出的请求是否成功。

更多看这里：http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
```

- 错误处理，状态码是4xx时，应返回错误信息，error当做key。

```
{
    error: "Invalid API key"
}
```

- 返回结果，针对不同操作，服务器向用户返回的结果应该符合以下规范。

```
GET /collection：返回资源对象的列表（数组）
GET /collection/resource：返回单个资源对象
POST /collection：返回新生成的资源对象
PUT /collection/resource：返回完整的资源对象
PATCH /collection/resource：返回完整的资源对象
DELETE /collection/resource：返回一个空文档
```

- Hypermedia API，RESTful API最好做到Hypermedia，即返回结果中提供链接，连向其他API方法，使得用户不查文档，也知道下一步应该做什么。

```
{"link": {
  "rel":   "collection https://www.example.com/zoos",
  "href":  "https://api.example.com/zoos",
  "title": "List of zoos",
  "type":  "application/vnd.yourformat+json"
}}
```

回到顶部

三. 基于Django实现流程

路由系统：**urls.py**

```
urlpatterns = [
    url(r'^users', Users.as_view()),
]
```

CBV视图类:views.py

```
from django.views import View
from django.http import JsonResponse

class Users(View):
    def get(self, request, *args, **kwargs):
        result = {
            'status': True,
            'data': 'response data'
        }
        return JsonResponse(result, status=200)

    def post(self, request, *args, **kwargs):
        result = {
            'status': True,
            'data': 'response data'
        }
        return JsonResponse(result, status=200)
```

回到顶部

四. 基于Django Rest Framework框架实现流程

1.基本流程

路由系统：urls.py

```
from django.conf.urls import url, include
from web.views.s1_api import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

CBV视图：views.py

```
from rest_framework.views import APIView
from rest_framework.response import Response

class TestView(APIView):
    def dispatch(self, request, *args, **kwargs):
        """
        请求到来之后，都要执行dispatch方法，dispatch方法根据请求方式不同触发 get/post/put
等方法

        注意：APIView中的dispatch方法有好多好多的功能
        """
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, *args, **kwargs):
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

回到顶部

五.认证和授权

a. 用户url传入的token认证

```
from django.conf.urls import url, include
from web.viewsimport TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.authentication import BaseAuthentication
from rest_framework.request import Request
from rest_framework import exceptions

token_list = [
    'sfsfss123kuf3j123',
    'asijnfowerkkf9812',
]



class TestAuthentication(BaseAuthentication):
    # 以下两个方法可以看源码
    def authenticate(self, request):
        """
        用户认证，如果验证成功后返回元组： (用户,用户Token)
        :param request:
        :return:
            None,表示跳过该验证；
                如果跳过了所有认证，默认用户和Token和使用配置文件进行设置
                self._authenticator = None
                if api_settings.UNAUTHENTICATED_USER:
                    self.user = api_settings.UNAUTHENTICATED_USER()
                else:
                    self.user = None

                if api_settings.UNAUTHENTICATED_TOKEN:
                    self.auth = api_settings.UNAUTHENTICATED_TOKEN()
                else:
                    self.auth = None
            (user,token)表示验证通过并设置用户名和Token；
            AuthenticationFailed异常
        """
        val = request.query_params.get('token')
        if val not in token_list:
            raise exceptions.AuthenticationFailed("用户认证失败")

        return ('登录用户', '用户token')

    def authenticate_header(self, request):
        """
        Return a string to be used as the value of the `WWW-Authenticate`
        header in a `401 Unauthenticated` response, or `None` if the
        authentication scheme should return `403 Permission Denied` responses.
        """
        # 验证失败时，返回的响应头WWW-Authenticate对应的值
        pass


class TestView(APIView):
    authentication_classes = [TestAuthentication, ]
    permission_classes = []

    def get(self, request, *args, **kwargs):
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

b. 请求头认证

```
from django.conf.urls import url, include
from web.viewsimport TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.authentication import BaseAuthentication
from rest_framework.request import Request
from rest_framework import exceptions

token_list = [
    'sfsfss123kuf3j123',
    'asijnfowerkkf9812',
]


class TestAuthentication(BaseAuthentication):
    def authenticate(self, request):
        """
        用户认证，如果验证成功后返回元组： (用户,用户Token)
        :param request:
        :return:
            None,表示跳过该验证；
                如果跳过了所有认证，默认用户和Token和使用配置文件进行设置
                self._authenticator = None
                if api_settings.UNAUTHENTICATED_USER:
                    self.user = api_settings.UNAUTHENTICATED_USER()
                else:
                    self.user = None

                if api_settings.UNAUTHENTICATED_TOKEN:
                    self.auth = api_settings.UNAUTHENTICATED_TOKEN()
                else:
                    self.auth = None
            (user,token)表示验证通过并设置用户名和Token；
            AuthenticationFailed异常
        """
        import base64
        auth = request.META.get('HTTP_AUTHORIZATION', b'')
        if auth:
            auth = auth.encode('utf-8')
        auth = auth.split()
        if not auth or auth[0].lower() != b'basic':
            raise exceptions.AuthenticationFailed('验证失败')
        if len(auth) != 2:
            raise exceptions.AuthenticationFailed('验证失败')
        username, part, password = base64.b64decode(auth[1]).decode('utf-
8').partition(':')
        if username == 'alex' and password == '123':
            return ('登录用户', '用户token')
        else:
            raise exceptions.AuthenticationFailed('用户名或密码错误')

    def authenticate_header(self, request):
        """
        Return a string to be used as the value of the `WWW-Authenticate`
        header in a `401 Unauthenticated` response, or `None` if the
        authentication scheme should return `403 Permission Denied` responses.
        """
        return 'Basic realm=api'


class TestView(APIView):
    authentication_classes = [TestAuthentication, ]
    permission_classes = []

    def get(self, request, *args, **kwargs):
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

c. 多个认证规则

```
from django.conf.urls import url, include
from web.views.s2_auth import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.authentication import BaseAuthentication
from rest_framework.request import Request
from rest_framework import exceptions

token_list = [
    'sfsfss123kuf3j123',
    'asijnfowerkkf9812',
]


class Test1Authentication(BaseAuthentication):
    def authenticate(self, request):
        """
        用户认证，如果验证成功后返回元组： (用户,用户Token)
        :param request:
        :return:
            None,表示跳过该验证；
                如果跳过了所有认证，默认用户和Token和使用配置文件进行设置
                self._authenticator = None
                if api_settings.UNAUTHENTICATED_USER:
                    self.user = api_settings.UNAUTHENTICATED_USER() # 默认值为：匿
名用户
                else:
                    self.user = None

                if api_settings.UNAUTHENTICATED_TOKEN:
                    self.auth = api_settings.UNAUTHENTICATED_TOKEN()# 默认值为：
None
                else:
                    self.auth = None
            (user,token)表示验证通过并设置用户名和Token；
            AuthenticationFailed异常
        """
        import base64
        auth = request.META.get('HTTP_AUTHORIZATION', b'')
        if auth:
            auth = auth.encode('utf-8')
        else:
            return None
        print(auth,'xxxx')
        auth = auth.split()
        if not auth or auth[0].lower() != b'basic':
            raise exceptions.AuthenticationFailed('验证失败')
        if len(auth) != 2:
            raise exceptions.AuthenticationFailed('验证失败')
        username, part, password = base64.b64decode(auth[1]).decode('utf-
8').partition(':')
        if username == 'alex' and password == '123':
            return ('登录用户', '用户token')
        else:
            raise exceptions.AuthenticationFailed('用户名或密码错误')

    def authenticate_header(self, request):
        """
        Return a string to be used as the value of the `WWW-Authenticate`
        header in a `401 Unauthenticated` response, or `None` if the
        authentication scheme should return `403 Permission Denied` responses.
        """
        # return 'Basic realm=api'
        pass

class Test2Authentication(BaseAuthentication):
    def authenticate(self, request):
        """
        用户认证，如果验证成功后返回元组： (用户,用户Token)
        :param request:
        :return:
            None,表示跳过该验证；
                如果跳过了所有认证，默认用户和Token和使用配置文件进行设置
                self._authenticator = None
                if api_settings.UNAUTHENTICATED_USER:
                    self.user = api_settings.UNAUTHENTICATED_USER() # 默认值为：匿
名用户
                else:
                    self.user = None

                if api_settings.UNAUTHENTICATED_TOKEN:
                    self.auth = api_settings.UNAUTHENTICATED_TOKEN()# 默认值为：
None
                else:
                    self.auth = None
            (user,token)表示验证通过并设置用户名和Token；
            AuthenticationFailed异常
        """
        val = request.query_params.get('token')
        if val not in token_list:
            raise exceptions.AuthenticationFailed("用户认证失败")
        return ('登录用户', '用户token')

    def authenticate_header(self, request):
        """
        Return a string to be used as the value of the `WWW-Authenticate`
        header in a `401 Unauthenticated` response, or `None` if the
        authentication scheme should return `403 Permission Denied` responses.
        """
        pass


class TestView(APIView):
    authentication_classes = [Test1Authentication, Test2Authentication]
    permission_classes = []

    def get(self, request, *args, **kwargs):
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

d. 认证和权限

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.authentication import BaseAuthentication
from rest_framework.permissions import BasePermission

from rest_framework.request import Request
from rest_framework import exceptions

token_list = [
    'sfsfss123kuf3j123',
    'asijnfowerkkf9812',
]


class TestAuthentication(BaseAuthentication):
    def authenticate(self, request):
        """
        用户认证，如果验证成功后返回元组： (用户,用户Token)
        :param request:
        :return:
            None,表示跳过该验证；
                如果跳过了所有认证，默认用户和Token和使用配置文件进行设置
                self._authenticator = None
                if api_settings.UNAUTHENTICATED_USER:
                    self.user = api_settings.UNAUTHENTICATED_USER() # 默认值为：匿
名用户
                else:
                    self.user = None

                if api_settings.UNAUTHENTICATED_TOKEN:
                    self.auth = api_settings.UNAUTHENTICATED_TOKEN()# 默认值为：
None
                else:
                    self.auth = None
            (user,token)表示验证通过并设置用户名和Token；
            AuthenticationFailed异常
        """
        val = request.query_params.get('token')
        if val not in token_list:
            raise exceptions.AuthenticationFailed("用户认证失败")

        return ('登录用户', '用户token')

    def authenticate_header(self, request):
        """
        Return a string to be used as the value of the `WWW-Authenticate`
        header in a `401 Unauthenticated` response, or `None` if the
        authentication scheme should return `403 Permission Denied` responses.
        """
        pass


class TestPermission(BasePermission):
    message = "权限验证失败"

    def has_permission(self, request, view):
        """
        判断是否有权限访问当前请求
        Return `True` if permission is granted, `False` otherwise.
        :param request:
        :param view:
        :return: True有权限；False无权限
        """
        if request.user == "管理员":
            return True

    # GenericAPIView中get_object时调用
    def has_object_permission(self, request, view, obj):
        """
        视图继承GenericAPIView，并在其中使用get_object时获取对象时，触发单独对象权限验证
        Return `True` if permission is granted, `False` otherwise.
        :param request:
        :param view:
        :param obj:
        :return: True有权限；False无权限
        """
        if request.user == "管理员":
            return True


class TestView(APIView):
    # 认证的动作是由request.user触发
    authentication_classes = [TestAuthentication, ]

    # 权限
    # 循环执行所有的权限
    permission_classes = [TestPermission, ]

    def get(self, request, *args, **kwargs):
        # self.dispatch
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

e. 全局使用

上述操作中均是对单独视图进行特殊配置即局部使用，如果想要对全局进行配置，则需要再配置文件中写入即可。

```
REST_FRAMEWORK = {
    'UNAUTHENTICATED_USER': None,
    'UNAUTHENTICATED_TOKEN': None,
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "web.utils.TestAuthentication",  #自定义认证类路径
    ],
    "DEFAULT_PERMISSION_CLASSES": [
        "web.utils.TestPermission",  # 自定义权限类路径
    ],
}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response

class TestView(APIView):

    def get(self, request, *args, **kwargs):
        # self.dispatch
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

回到顶部

六. 用户访问次数/频率限制

a. 基于用户IP限制访问频率

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import time
from rest_framework.views import APIView
from rest_framework.response import Response

from rest_framework import exceptions
from rest_framework.throttling import BaseThrottle
from rest_framework.settings import api_settings

# 保存访问记录
RECORD = {
    '用户IP': [12312139, 12312135, 12312133, ]
}



class TestThrottle(BaseThrottle):
    ctime = time.time

    def get_ident(self, request):
        """
        根据用户IP和代理IP，当做请求者的唯一IP
        Identify the machine making the request by parsing HTTP_X_FORWARDED_FOR
        if present and number of proxies is > 0. If not use all of
        HTTP_X_FORWARDED_FOR if it is available, if not use REMOTE_ADDR.
        """
        xff = request.META.get('HTTP_X_FORWARDED_FOR')
        remote_addr = request.META.get('REMOTE_ADDR')
        num_proxies = api_settings.NUM_PROXIES

        if num_proxies is not None:
            if num_proxies == 0 or xff is None:
                return remote_addr
            addrs = xff.split(',')
            client_addr = addrs[-min(num_proxies, len(addrs))]
            return client_addr.strip()

        return ''.join(xff.split()) if xff else remote_addr

    def allow_request(self, request, view):
        """
        是否仍然在允许范围内
        Return `True` if the request should be allowed, `False` otherwise.
        :param request:
        :param view:
        :return: True，表示可以通过；False表示已超过限制，不允许访问
        """
        # 获取用户唯一标识（如：IP）

        # 允许一分钟访问10次
        num_request = 10
        time_request = 60

        now = self.ctime()
        ident = self.get_ident(request)
        self.ident = ident
        if ident not in RECORD:
            RECORD[ident] = [now, ]
            return True
        history = RECORD[ident]
        while history and history[-1] <= now - time_request:
            history.pop()
        if len(history) < num_request:
            history.insert(0, now)
            return True

    def wait(self):
        """
        多少秒后可以允许继续访问
        Optionally, return a recommended number of seconds to wait before
        the next request.
        """
        last_time = RECORD[self.ident][0]
        now = self.ctime()
        return int(60 + last_time - now)


class TestView(APIView):
    throttle_classes = [TestThrottle, ]

    def get(self, request, *args, **kwargs):
        # self.dispatch
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')

    def throttled(self, request, wait):
        """
        访问次数被限制时，定制错误信息
        """

        class Throttled(exceptions.Throttled):
            default_detail = '请求被限制.'
            extra_detail_singular = '请 {wait} 秒之后再重试.'
            extra_detail_plural = '请 {wait} 秒之后再重试.'

        raise Throttled(wait)
```

b. 基于用户IP限制访问频率（利于Django缓存）

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_RATES': {
        'test_scope': '10/m',
    },
}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response

from rest_framework import exceptions
from rest_framework.throttling import SimpleRateThrottle


class TestThrottle(SimpleRateThrottle):

    # 配置文件定义的显示频率的Key
    scope = "test_scope"

    def get_cache_key(self, request, view):
        """
        Should return a unique cache-key which can be used for throttling.
        Must be overridden.

        May return `None` if the request should not be throttled.
        """
        if not request.user:
            ident = self.get_ident(request)
        else:
            ident = request.user
        return self.cache_format % {
            'scope': self.scope,
            'ident': ident
        }


class TestView(APIView):
    throttle_classes = [TestThrottle, ]

    def get(self, request, *args, **kwargs):
        # self.dispatch
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')

    def throttled(self, request, wait):
        """
        访问次数被限制时，定制错误信息
        """

        class Throttled(exceptions.Throttled):
            default_detail = '请求被限制.'
            extra_detail_singular = '请 {wait} 秒之后再重试.'
            extra_detail_plural = '请 {wait} 秒之后再重试.'

        raise Throttled(wait)
```

c. view中限制请求频率

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_RATES': {
        'xxxxxx': '10/m',
    },
}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response

from rest_framework import exceptions
from rest_framework.throttling import ScopedRateThrottle
```

```python


# 继承 ScopedRateThrottle
class TestThrottle(ScopedRateThrottle):

    def get_cache_key(self, request, view):
        """
        Should return a unique cache-key which can be used for throttling.
        Must be overridden.

        May return `None` if the request should not be throttled.
        """
        if not request.user:
            ident = self.get_ident(request)
        else:
            ident = request.user

        return self.cache_format % {
            'scope': self.scope,
            'ident': ident
        }


class TestView(APIView):
    throttle_classes = [TestThrottle, ]

    # 在settings中获取 xxxxxx 对应的频率限制值
    throttle_scope = "xxxxxx"

    def get(self, request, *args, **kwargs):
        # self.dispatch
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')

    def throttled(self, request, wait):
        """
        访问次数被限制时，定制错误信息
        """

        class Throttled(exceptions.Throttled):
            default_detail = '请求被限制.'
            extra_detail_singular = '请 {wait} 秒之后再重试.'
            extra_detail_plural = '请 {wait} 秒之后再重试.'

        raise Throttled(wait)
```

d. 匿名时用IP限制+登录时用Token限制

```
REST_FRAMEWORK = {
    'UNAUTHENTICATED_USER': None,
    'UNAUTHENTICATED_TOKEN': None,
    'DEFAULT_THROTTLE_RATES': {
        'luffy_anon': '10/m',
        'luffy_user': '20/m',
    },
}
```

```
from django.conf.urls import url, include
from web.views.s3_throttling import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response

from rest_framework.throttling import SimpleRateThrottle


class LuffyAnonRateThrottle(SimpleRateThrottle):
    """
    匿名用户，根据IP进行限制
    """
    scope = "luffy_anon"

    def get_cache_key(self, request, view):
        # 用户已登录，则跳过 匿名频率限制
        if request.user:
            return None

        return self.cache_format % {
            'scope': self.scope,
            'ident': self.get_ident(request)
        }


class LuffyUserRateThrottle(SimpleRateThrottle):
    """
    登录用户，根据用户token限制
    """
    scope = "luffy_user"

    def get_ident(self, request):
        """
        认证成功时：request.user是用户对象；request.auth是token对象
        :param request:
        :return:
        """
        # return request.auth.token
        return "user_token"
    def get_cache_key(self, request, view):
        """
        获取缓存key
        :param request:
        :param view:
        :return:
        """
        # 未登录用户，则跳过 Token限制
        if not request.user:
            return None

        return self.cache_format % {
            'scope': self.scope,
            'ident': self.get_ident(request)
        }


class TestView(APIView):
    throttle_classes = [LuffyUserRateThrottle, LuffyAnonRateThrottle, ]

    def get(self, request, *args, **kwargs):
        # self.dispatch
        print(request.user)
        print(request.auth)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

e. 全局使用

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'api.utils.throttles.throttles.LuffyAnonRateThrottle',
        'api.utils.throttles.throttles.LuffyUserRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '10/day',
        'user': '10/day',
        'luffy_anon': '10/m',
        'luffy_user': '20/m',
    },
}
```

回到顶部

七. 版本控制

a. 基于url的get传参方式

如：/users?version=v1

```
REST_FRAMEWORK = {
    'DEFAULT_VERSION': 'v1',            # 默认版本
    'ALLOWED_VERSIONS': ['v1', 'v2'],   # 允许的版本
    'VERSION_PARAM': 'version'          # URL中获取值的key
}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view(),name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.versioning import QueryParameterVersioning


class TestView(APIView):
    versioning_class = QueryParameterVersioning

    def get(self, request, *args, **kwargs):

        # 获取版本
        print(request.version)
        # 获取版本管理的类
        print(request.versioning_scheme)

        # 反向生成URL
        reverse_url = request.versioning_scheme.reverse('test', request=request)
        print(reverse_url)

        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

b. 基于url的正则方式

如：/v1/users/

```
REST_FRAMEWORK = {
    'DEFAULT_VERSION': 'v1',            # 默认版本
    'ALLOWED_VERSIONS': ['v1', 'v2'],   # 允许的版本
    'VERSION_PARAM': 'version'          # URL中获取值的key
}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^(?P<version>[v1|v2]+)/test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.versioning import URLPathVersioning


class TestView(APIView):
    versioning_class = URLPathVersioning

    def get(self, request, *args, **kwargs):
        # 获取版本
        print(request.version)
        # 获取版本管理的类
        print(request.versioning_scheme)

        # 反向生成URL
        reverse_url = request.versioning_scheme.reverse('test', request=request)
        print(reverse_url)

        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

c. 基于 accept 请求头方式

如：Accept: application/json; version=1.0

```
REST_FRAMEWORK = {
    'DEFAULT_VERSION': 'v1',            # 默认版本
    'ALLOWED_VERSIONS': ['v1', 'v2'],   # 允许的版本
    'VERSION_PARAM': 'version'          # URL中获取值的key
}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
```

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.versioning import AcceptHeaderVersioning


class TestView(APIView):
    versioning_class = AcceptHeaderVersioning

    def get(self, request, *args, **kwargs):
        # 获取版本 HTTP_ACCEPT头
        print(request.version)
        # 获取版本管理的类
        print(request.versioning_scheme)
        # 反向生成URL
        reverse_url = request.versioning_scheme.reverse('test', request=request)
        print(reverse_url)

        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

d. 基于主机名方法

如：v1.example.com

```
ALLOWED_HOSTS = ['*']
REST_FRAMEWORK = {
    'DEFAULT_VERSION': 'v1',  # 默认版本
    'ALLOWED_VERSIONS': ['v1', 'v2'],  # 允许的版本
    'VERSION_PARAM': 'version'  # URL中获取值的key
}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.versioning import HostNameVersioning


class TestView(APIView):
    versioning_class = HostNameVersioning

    def get(self, request, *args, **kwargs):
        # 获取版本
        print(request.version)
        # 获取版本管理的类
        print(request.versioning_scheme)
        # 反向生成URL
        reverse_url = request.versioning_scheme.reverse('test', request=request)
        print(reverse_url)

        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

e. 基于django路由系统的namespace

如：example.com/v1/users/

```
REST_FRAMEWORK = {
    'DEFAULT_VERSION': 'v1',  # 默认版本
    'ALLOWED_VERSIONS': ['v1', 'v2'],  # 允许的版本
    'VERSION_PARAM': 'version'  # URL中获取值的key
}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'^v1/', ([
                      url(r'test/', TestView.as_view(), name='test'),
                  ], None, 'v1')),
    url(r'^v2/', ([
                      url(r'test/', TestView.as_view(), name='test'),
                  ], None, 'v2')),

]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.versioning import NamespaceVersioning


class TestView(APIView):
    versioning_class = NamespaceVersioning

    def get(self, request, *args, **kwargs):
        # 获取版本
        print(request.version)
        # 获取版本管理的类
        print(request.versioning_scheme)
        # 反向生成URL
        reverse_url = request.versioning_scheme.reverse('test', request=request)
        print(reverse_url)
        return Response('GET请求，响应内容')

    def post(self, request, *args, **kwargs):
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

f. 全局使用

```
REST_FRAMEWORK = {
    'DEFAULT_VERSIONING_CLASS':"rest_framework.versioning.URLPathVersioning",
    'DEFAULT_VERSION': 'v1',
    'ALLOWED_VERSIONS': ['v1', 'v2'],
    'VERSION_PARAM': 'version'
}
```

回到顶部

八. 解析器（parser）

根据请求头 content-type 选择对应的解析器就请求体内容进行处理。

a. 仅处理请求头content-type为application/json的请求体

```
from django.conf.urls import url, include
from web.views.s5_parser import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.request import Request
from rest_framework.parsers import JSONParser


class TestView(APIView):
    parser_classes = [JSONParser, ]

    def post(self, request, *args, **kwargs):
        print(request.content_type)

        # 获取请求的值，并使用对应的JSONParser进行处理
        print(request.data)

        # application/x-www-form-urlencoded 或 multipart/form-data时，
request.POST中才有值
        print(request.POST)
        print(request.FILES)

        return Response('POST请求，响应内容')
    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

b. 仅处理请求头content-type为application/x-www-form-urlencoded 的请求体

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.request import Request
from rest_framework.parsers import FormParser


class TestView(APIView):
    parser_classes = [FormParser, ]

    def post(self, request, *args, **kwargs):
        print(request.content_type)

        # 获取请求的值，并使用对应的JSONParser进行处理
        print(request.data)

        # application/x-www-form-urlencoded 或 multipart/form-data时，
request.POST中才有值
        print(request.POST)
        print(request.FILES)

        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

c. 仅处理请求头content-type为multipart/form-data的请求体

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.request import Request
from rest_framework.parsers import MultiPartParser


class TestView(APIView):
    parser_classes = [MultiPartParser, ]

    def post(self, request, *args, **kwargs):
        print(request.content_type)

        # 获取请求的值，并使用对应的JSONParser进行处理
        print(request.data)
        # application/x-www-form-urlencoded 或 multipart/form-data时，
request.POST中才有值
        print(request.POST)
        print(request.FILES)
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<form action="http://127.0.0.1:8000/test/" method="post"
enctype="multipart/form-data">
    <input type="text" name="user" />
    <input type="file" name="img">

    <input type="submit" value="提交">

</form>
</body>
</html>
```

d. 仅上传文件

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'test/(?P<filename>[^/]+)', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.request import Request
from rest_framework.parsers import FileUploadParser
```

```
class TestView(APIView):
    parser_classes = [FileUploadParser, ]

    def post(self, request, filename, *args, **kwargs):
        print(filename)
        print(request.content_type)

        # 获取请求的值，并使用对应的JSONParser进行处理
        print(request.data)
        # application/x-www-form-urlencoded 或 multipart/form-data时，
request.POST中才有值
        print(request.POST)
        print(request.FILES)
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<form action="http://127.0.0.1:8000/test/f1.numbers" method="post"
enctype="multipart/form-data">
    <input type="text" name="user" />
    <input type="file" name="img">

    <input type="submit" value="提交">

</form>
</body>
</html>
```

e. 同时多个Parser

当同时使用多个parser时，rest framework会根据请求头content-type自动进行比对，并使用对应parser

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.request import Request
from rest_framework.parsers import JSONParser, FormParser, MultiPartParser
```

```


class TestView(APIView):
    parser_classes = [JSONParser, FormParser, MultiPartParser, ]

    def post(self, request, *args, **kwargs):
        print(request.content_type)

        # 获取请求的值，并使用对应的JSONParser进行处理
        print(request.data)
        # application/x-www-form-urlencoded 或 multipart/form-data时，
request.POST中才有值
        print(request.POST)
        print(request.FILES)
        return Response('POST请求，响应内容')

    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

f. 全局使用

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES':[
        'rest_framework.parsers.JSONParser'
        'rest_framework.parsers.FormParser'
        'rest_framework.parsers.MultiPartParser'
    ]

}
```

```
from django.conf.urls import url, include
from web.views import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response


class TestView(APIView):
    def post(self, request, *args, **kwargs):
        print(request.content_type)

        # 获取请求的值，并使用对应的JSONParser进行处理
        print(request.data)
        # application/x-www-form-urlencoded 或 multipart/form-data时，
request.POST中才有值
        print(request.POST)
        print(request.FILES)
        return Response('POST请求，响应内容')


    def put(self, request, *args, **kwargs):
        return Response('PUT请求，响应内容')
```

注意：个别特殊的值可以通过Django的request对象 request._request 来进行获取

回到顶部

九. 序列化组件

序列化用于对用户请求数据进行验证和数据(query_set)进行序列化。

a. 自定义字段

```
from django.conf.urls import url, include
from web.views.s6_serializers import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers
from .. import models


class PasswordValidator(object):
    def __init__(self, base):
        self.base = base

    def __call__(self, value):
        if value != self.base:
            message = 'This field must be %s.' % self.base
            raise serializers.ValidationError(message)

    def set_context(self, serializer_field):
        """
        This hook is called by the serializer instance,
        prior to the validation call being made.
        """
        # 执行验证之前调用,serializer_fields是当前字段对象
        pass


class UserSerializer(serializers.Serializer):
    ut_title = serializers.CharField(source='ut.title')
    user = serializers.CharField(min_length=6)
    pwd = serializers.CharField(error_messages={'required': '密码不能为空'},
validators=[PasswordValidator('666')])


class TestView(APIView):
    def get(self, request, *args, **kwargs):

        # 序列化，将数据库查询字段序列化为字典
        data_list = models.UserInfo.objects.all()
        ser = UserSerializer(instance=data_list, many=True)
        # 或
        # obj = models.UserInfo.objects.all().first()
        # ser = UserSerializer(instance=obj, many=False)
        return Response(ser.data)

    def post(self, request, *args, **kwargs):
        # 验证，对请求发来的数据进行验证
        ser = UserSerializer(data=request.data)
        if ser.is_valid():
            print(ser.validated_data)
        else:
            print(ser.errors)

        return Response('POST请求，响应内容')
```

b. 基于Model自动生成字段

```
from django.conf.urls import url, include
from web.views.s6_serializers import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers
from .. import models


class PasswordValidator(object):
    def __init__(self, base):
        self.base = str(base)

    def __call__(self, value):
        if value != self.base:
            message = 'This field must be %s.' % self.base
            raise serializers.ValidationError(message)

    def set_context(self, serializer_field):
        """
        This hook is called by the serializer instance,
        prior to the validation call being made.
        """
        # 执行验证之前调用,serializer_fields是当前字段对象
        pass

class ModelUserSerializer(serializers.ModelSerializer):

    user = serializers.CharField(max_length=32)

    class Meta:
        model = models.UserInfo
        fields = "__all__"
        # fields = ['user', 'pwd', 'ut']
        depth = 2
        extra_kwargs = {'user': {'min_length': 6}, 'pwd': {'validators':
[PasswordValidator(666), ]}}
        # read_only_fields = ['user']


class TestView(APIView):
    def get(self, request, *args, **kwargs):

        # 序列化，将数据库查询字段序列化为字典
        data_list = models.UserInfo.objects.all()
        ser = ModelUserSerializer(instance=data_list, many=True)
        # 或
        # obj = models.UserInfo.objects.all().first()
        # ser = UserSerializer(instance=obj, many=False)
        return Response(ser.data)

    def post(self, request, *args, **kwargs):
        # 验证，对请求发来的数据进行验证
        print(request.data)
        ser = ModelUserSerializer(data=request.data)
        if ser.is_valid():
            print(ser.validated_data)
        else:
            print(ser.errors)

        return Response('POST请求，响应内容')
```

c. 生成URL

```
from django.conf.urls import url, include
from web.views.s6_serializers import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
    url(r'detail/(?P<pk>\d+)/', TestView.as_view(), name='detail'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers
from .. import models



class PasswordValidator(object):
    def __init__(self, base):
        self.base = str(base)

    def __call__(self, value):
        if value != self.base:
            message = 'This field must be %s.' % self.base
            raise serializers.ValidationError(message)

    def set_context(self, serializer_field):
        """
        This hook is called by the serializer instance,
        prior to the validation call being made.
        """
        # 执行验证之前调用,serializer_fields是当前字段对象
        pass


class ModelUserSerializer(serializers.ModelSerializer):
    ut = serializers.HyperlinkedIdentityField(view_name='detail')
    class Meta:
        model = models.UserInfo
        fields = "__all__"

        extra_kwargs = {
            'user': {'min_length': 6},
            'pwd': {'validators': [PasswordValidator(666),]},
        }



class TestView(APIView):
    def get(self, request, *args, **kwargs):

        # 序列化，将数据库查询字段序列化为字典
        data_list = models.UserInfo.objects.all()
        ser = ModelUserSerializer(instance=data_list, many=True, context=
{'request': request})
        # 或
        # obj = models.UserInfo.objects.all().first()
        # ser = UserSerializer(instance=obj, many=False)
        return Response(ser.data)

    def post(self, request, *args, **kwargs):
        # 验证，对请求发来的数据进行验证
        print(request.data)
        ser = ModelUserSerializer(data=request.data)
        if ser.is_valid():
            print(ser.validated_data)
        else:
            print(ser.errors)

        return Response('POST请求，响应内容')
```

d. 自动生成URL

```
from django.conf.urls import url, include
from web.views.s6_serializers import TestView

urlpatterns = [
    url(r'test/', TestView.as_view(), name='test'),
    url(r'detail/(?P<pk>\d+)/', TestView.as_view(), name='xxxx'),
]
```

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers
from .. import models


class PasswordValidator(object):
    def __init__(self, base):
        self.base = str(base)

    def __call__(self, value):
        if value != self.base:
            message = 'This field must be %s.' % self.base
            raise serializers.ValidationError(message)

    def set_context(self, serializer_field):
        """
        This hook is called by the serializer instance,
        prior to the validation call being made.
        """
        # 执行验证之前调用,serializer_fields是当前字段对象
        pass


class ModelUserSerializer(serializers.HyperlinkedModelSerializer):
    ll = serializers.HyperlinkedIdentityField(view_name='xxxx')
    tt = serializers.CharField(required=False)

    class Meta:
        model = models.UserInfo
        fields = "__all__"
        list_serializer_class = serializers.ListSerializer

        extra_kwargs = {
            'user': {'min_length': 6},
            'pwd': {'validators': [PasswordValidator(666), ]},
            'url': {'view_name': 'xxxx'},
            'ut': {'view_name': 'xxxx'},
        }


class TestView(APIView):
    def get(self, request, *args, **kwargs):
        # # 序列化，将数据库查询字段序列化为字典
        data_list = models.UserInfo.objects.all()
        ser = ModelUserSerializer(instance=data_list, many=True, context=
{'request': request})
        # # 如果Many=True
        # # 或
        # # obj = models.UserInfo.objects.all().first()
        # # ser = UserSerializer(instance=obj, many=False)
        return Response(ser.data)

    def post(self, request, *args, **kwargs):
        # 验证，对请求发来的数据进行验证
        print(request.data)
        ser = ModelUserSerializer(data=request.data)
        if ser.is_valid():
            print(ser.validated_data)
        else:
            print(ser.errors)

        return Response('POST请求，响应内容')
```

回到顶部

十. 分页器组件

a. 根据页码进行分页

```
from django.conf.urls import url, include
from rest_framework import routers
from web.views import s9_pagination

urlpatterns = [
    url(r'^test/', s9_pagination.UserViewSet.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework import serializers
from .. import models

from rest_framework.pagination import PageNumberPagination


class StandardResultsSetPagination(PageNumberPagination):
    # 默认每页显示的数据条数
    page_size = 1
    # 获取URL参数中设置的每页显示数据条数
    page_size_query_param = 'page_size'

    # 获取URL参数中传入的页码key
    page_query_param = 'page'

    # 最大支持的每页显示的数据条数
    max_page_size = 1


class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class UserViewSet(APIView):
    def get(self, request, *args, **kwargs):
        user_list = models.UserInfo.objects.all().order_by('-id')

        # 实例化分页对象，获取数据库中的分页数据
        paginator = StandardResultsSetPagination()
        page_user_list = paginator.paginate_queryset(user_list, self.request,
view=self)

        # 序列化对象
        serializer = UserSerializer(page_user_list, many=True)

        # 生成分页和数据
        response = paginator.get_paginated_response(serializer.data)
        return response
```

b. 位置和个数进行分页

```
from django.conf.urls import url, include
from web.views import s9_pagination

urlpatterns = [
    url(r'^test/', s9_pagination.UserViewSet.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework import serializers
from .. import models

from rest_framework.pagination import PageNumberPagination,LimitOffsetPagination


class StandardResultsSetPagination(LimitOffsetPagination):
    # 默认每页显示的数据条数
    default_limit = 10
    # URL中传入的显示数据条数的参数
    limit_query_param = 'limit'
    # URL中传入的数据位置的参数
    offset_query_param = 'offset'
    # 最大每页显得条数
    max_limit = None

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class UserViewSet(APIView):
    def get(self, request, *args, **kwargs):
        user_list = models.UserInfo.objects.all().order_by('-id')

        # 实例化分页对象，获取数据库中的分页数据
        paginator = StandardResultsSetPagination()
        page_user_list = paginator.paginate_queryset(user_list, self.request,
view=self)

        # 序列化对象
        serializer = UserSerializer(page_user_list, many=True)
        # 生成分页和数据
        response = paginator.get_paginated_response(serializer.data)
        return response
```

c. 游标分页

```
from django.conf.urls import url, include
from web.views import s9_pagination

urlpatterns = [
    url(r'^test/', s9_pagination.UserViewSet.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework import serializers
from .. import models

from rest_framework.pagination import PageNumberPagination,
LimitOffsetPagination, CursorPagination


class StandardResultsSetPagination(CursorPagination):
    # URL传入的游标参数
    cursor_query_param = 'cursor'
    # 默认每页显示的数据条数
    page_size = 2
    # URL传入的每页显示条数的参数
    page_size_query_param = 'page_size'
    # 每页显示数据最大条数
    max_page_size = 1000

    # 根据ID从大到小排列
    ordering = "id"



class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class UserViewSet(APIView):
    def get(self, request, *args, **kwargs):
        user_list = models.UserInfo.objects.all().order_by('-id')

        # 实例化分页对象，获取数据库中的分页数据
        paginator = StandardResultsSetPagination()
        page_user_list = paginator.paginate_queryset(user_list, self.request,
view=self)

        # 序列化对象
        serializer = UserSerializer(page_user_list, many=True)
        # 生成分页和数据
        response = paginator.get_paginated_response(serializer.data)
        return response
```

回到顶部

十一. 路由系统

a. 自定义路由

```
from django.conf.urls import url, include
from web.views import s11_render

urlpatterns = [
    url(r'^test/$', s11_render.TestView.as_view()),
    url(r'^test\.(?P<format>[a-z0-9]+)$', s11_render.TestView.as_view()),
    url(r'^test/(?P<pk>[^/.]+)/$', s11_render.TestView.as_view()),
    url(r'^test/(?P<pk>[^/.]+)\.(?P<format>[a-z0-9]+)$',
s11_render.TestView.as_view())
]
```

```
from rest_framework.views import APIView
from rest_framework.response import Response
from .. import models


class TestView(APIView):
    def get(self, request, *args, **kwargs):
        print(kwargs)
        print(self.renderer_classes)
        return Response('...')
```

b. 半自动路由

```
from django.conf.urls import url, include
from web.views import s10_generic

urlpatterns = [
    url(r'^test/$', s10_generic.UserViewSet.as_view({'get': 'list', 'post':
'create'})),
    url(r'^test/(?P<pk>\d+)/$', s10_generic.UserViewSet.as_view(
        {'get': 'retrieve', 'put': 'update', 'patch': 'partial_update',
'delete': 'destroy'})),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.viewsets import ModelViewSet
from rest_framework import serializers
from .. import models


class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class UserViewSet(ModelViewSet):
    queryset = models.UserInfo.objects.all()
    serializer_class = UserSerializer
```

c. 全自动路由(推荐使用)

```
from django.conf.urls import url, include
from rest_framework import routers
from web.views import s10_generic


router = routers.DefaultRouter()
router.register(r'users', s10_generic.UserViewSet)

urlpatterns = [
    url(r'^', include(router.urls)),
]
```

```
from rest_framework.viewsets import ModelViewSet
from rest_framework import serializers
from .. import models


class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class UserViewSet(ModelViewSet):
    queryset = models.UserInfo.objects.all()
    serializer_class = UserSerializer
```

回到顶部

十二.视图组件

a. GenericViewSet

```
from django.conf.urls import url, include
from web.views.s7_viewset import TestView

urlpatterns = [
    url(r'test/', TestView.as_view({'get':'list'}), name='test'),
    url(r'detail/(?P<pk>\d+)/', TestView.as_view({'get':'list'}), name='xxxx'),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework import viewsets
from rest_framework.response import Response
```

```
class TestView(viewsets.GenericViewSet):
    def list(self, request, *args, **kwargs):
        return Response('...')

    def add(self, request, *args, **kwargs):
        pass

    def delete(self, request, *args, **kwargs):
        pass

    def edit(self, request, *args, **kwargs):
        pass
```

b. ModelViewSet（自定义URL）

```
from django.conf.urls import url, include
from web.views import s10_generic

urlpatterns = [
    url(r'^test/$', s10_generic.UserViewSet.as_view({'get': 'list', 'post':
'create'})),
    url(r'^test/(?P<pk>\d+)/$', s10_generic.UserViewSet.as_view(
        {'get': 'retrieve', 'put': 'update', 'patch': 'partial_update',
'delete': 'destroy'})),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.viewsets import ModelViewSet
from rest_framework import serializers
from .. import models


class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class UserViewSet(ModelViewSet):
    queryset = models.UserInfo.objects.all()
    serializer_class = UserSerializer
```

c. ModelViewSet（rest framework路由）

```
from django.conf.urls import url, include
from rest_framework import routers
from app01 import views

router = routers.DefaultRouter()
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    url(r'^', include(router.urls)),
]
```

```
from rest_framework import viewsets
from rest_framework import serializers


class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = models.User
        fields = ('url', 'username', 'email', 'groups')


class GroupSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = models.Group
        fields = ('url', 'name')

class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserSerializer


class GroupViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
```

回到顶部

十三. 渲染器

根据 用户请求URL 或 用户可接受的类型，筛选出合适的 渲染组件。
用户请求URL：

- http://127.0.0.1:8000/test/?format=json
- http://127.0.0.1:8000/test.json

用户请求头：

- Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,/;q=0.8

a. json

访问URL：

- http://127.0.0.1:8000/test/?format=json
- http://127.0.0.1:8000/test.json
- http://127.0.0.1:8000/test/

```
from django.conf.urls import url, include
from web.views import s11_render

urlpatterns = [
    url(r'^test/$', s11_render.TestView.as_view()),
    url(r'^test\.(?P<format>[a-z0-9]+)', s11_render.TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers

from rest_framework.renderers import JSONRenderer

from .. import models


class TestSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class TestView(APIView):
    renderer_classes = [JSONRenderer, ]

    def get(self, request, *args, **kwargs):
        user_list = models.UserInfo.objects.all()
        ser = TestSerializer(instance=user_list, many=True)
        return Response(ser.data)
```

b. 表格

访问URL：

- http://127.0.0.1:8000/test/?format=admin
- http://127.0.0.1:8000/test.admin
- http://127.0.0.1:8000/test/

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers

from rest_framework.renderers import AdminRenderer

from .. import models


class TestSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class TestView(APIView):
    renderer_classes = [AdminRenderer, ]

    def get(self, request, *args, **kwargs):
        user_list = models.UserInfo.objects.all()
        ser = TestSerializer(instance=user_list, many=True)
        return Response(ser.data)
```

c. Form表单

访问URL：

- http://127.0.0.1:8000/test/?format=form
- http://127.0.0.1:8000/test.form
- http://127.0.0.1:8000/test/

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers

from rest_framework.renderers import JSONRenderer
from rest_framework.renderers import AdminRenderer
from rest_framework.renderers import HTMLFormRenderer

from .. import models


class TestSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class TestView(APIView):
    renderer_classes = [HTMLFormRenderer, ]

    def get(self, request, *args, **kwargs):
        user_list = models.UserInfo.objects.all().first()
        ser = TestSerializer(instance=user_list, many=False)
        return Response(ser.data)
```

d. 自定义显示模板

访问URL：

- http://127.0.0.1:8000/test/?format=html
- http://127.0.0.1:8000/test.html
- http://127.0.0.1:8000/test/

```
from django.conf.urls import url, include
from web.views import s11_render

urlpatterns = [
    url(r'^test/$', s11_render.TestView.as_view()),
    url(r'^test\.(?P<format>[a-z0-9]+)', s11_render.TestView.as_view()),
]
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers
from rest_framework.renderers import TemplateHTMLRenderer

from .. import models


class TestSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class TestView(APIView):
    renderer_classes = [TemplateHTMLRenderer, ]

    def get(self, request, *args, **kwargs):
        user_list = models.UserInfo.objects.all().first()
        ser = TestSerializer(instance=user_list, many=False)
        return Response(ser.data, template_name='user_detail.html')
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    {{ user }}
    {{ pwd }}
    {{ ut }}
</body>
</html>
```

e. 浏览器格式API+JSON

访问URL：

- http://127.0.0.1:8000/test/?format=api

- http://127.0.0.1:8000/test.api
- http://127.0.0.1:8000/test/

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import serializers

from rest_framework.renderers import JSONRenderer
from rest_framework.renderers import BrowsableAPIRenderer

from .. import models


class TestSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.UserInfo
        fields = "__all__"


class CustomBrowsableAPIRenderer(BrowsableAPIRenderer):
    def get_default_renderer(self, view):
        return JSONRenderer()


class TestView(APIView):
    renderer_classes = [CustomBrowsableAPIRenderer, ]

    def get(self, request, *args, **kwargs):
        user_list = models.UserInfo.objects.all().first()
        ser = TestSerializer(instance=user_list, many=False)
        return Response(ser.data, template_name='user_detail.html')
```

注意：如果同时多个存在时，自动根据URL后缀来选择渲染器。

阅读目录(Content)

回到顶部(go to top)

RESTful规范

REST风格

- 资源 网页中能看到的都是资源
- URI 统一资源标识符

  URL 统一资源定位符
- 统一资源接口

  对资源的操作根据HTTP请求方式的不同来进行不同的操作

  遵循HTTP请求方式的语义
- 前后端传输的是资源的表述
- 展现的是资源的状态

凡是遵循REST风格实现的前后端交互都叫RESTful架构

核心思想

- 面向资源去编程, url中尽量用名词不要用动词
- 根据HTTP请求方式的不同对资源进行不同的操作

在url中体现的

- 体现版本

  https://v3.bootcss.com/

  https://v2.bootcss.com/

- 体现是否是API

  https://v2.bootcss.com/api

- 有过滤条件

  https://v3.bootcss.com/css/#grid

- 尽量用HTTPS

在返回值中

- 携带状态码
- 返回值
  - get 返回查看的所有或者单条数据
  - post 返回新增的这条数据
  - put/patch 返回更新的这条数据
  - delete 返回值空
- 携带错误信息
- 携带超链接

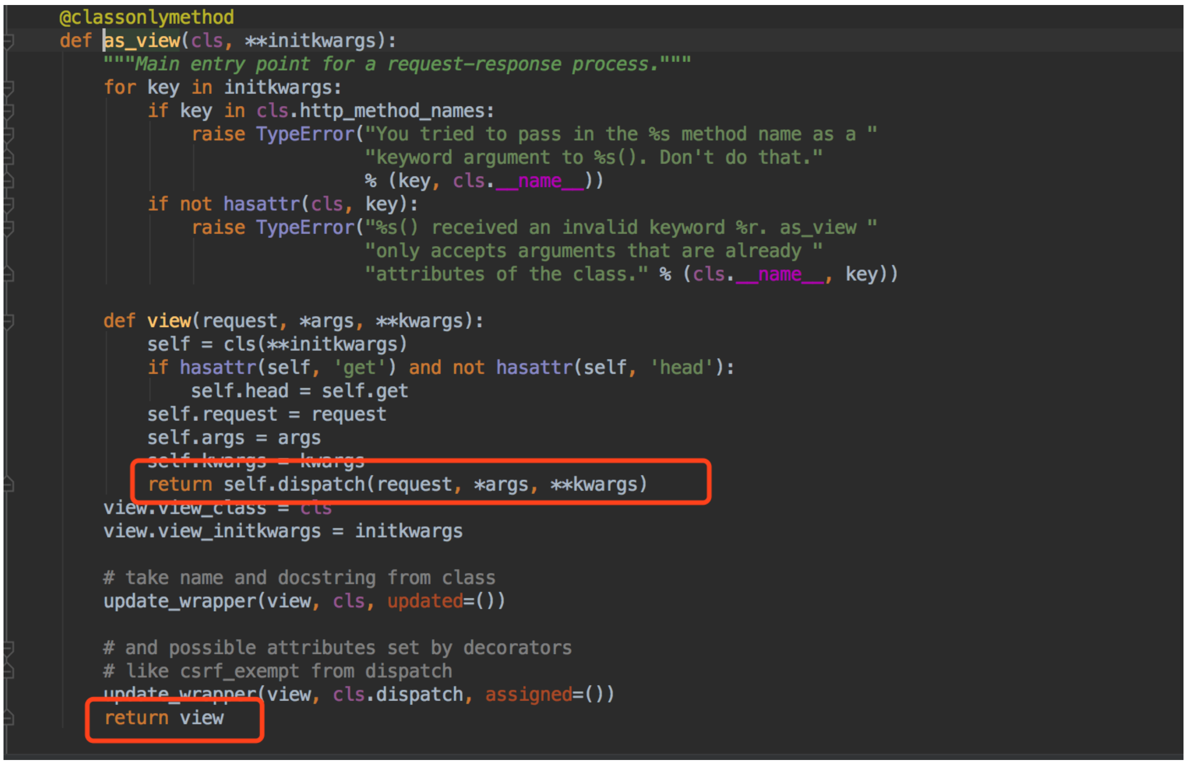
  前后端不分离的项目用的比较多

回到顶部(go to top)

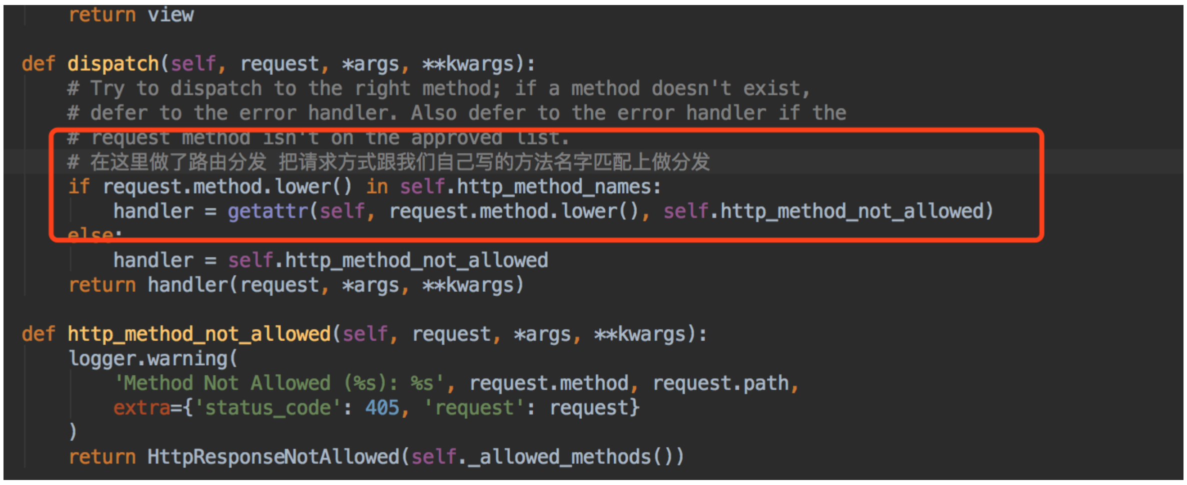
FBV和CBV区别

- FBV 基于函数的视图
- CBV 基于类的视图

```
Copyurlpatterns = [
    path('admin/', admin.site.urls),
    path('test_fbv', test_fbv),
    path('test_cbv', TestCBV.as_view())
]
Copydef test_fbv(request):
    return HttpResponse("ok")


class TestCBV(View):
    def get(self, request):
        return HttpResponse("ok")
```



```
    @classonlymethod
    def as_view(cls, **initkwargs):
        """Main entry point for a request-response process."""
        for key in initkwargs:
            if key in cls.http_method_names:
                raise TypeError("You tried to pass in the %s method name as a "
                                "keyword argument to %s(). Don't do that."
                                % (key, cls.__name__))
            if not hasattr(cls, key):
                raise TypeError("%s() received an invalid keyword %r. as_view "
                                "only accepts arguments that are already "
                                "attributes of the class." % (cls.__name__, key))

        def view(request, *args, **kwargs):
            self = cls(**initkwargs)
            if hasattr(self, 'get') and not hasattr(self, 'head'):
                self.head = self.get
            self.request = request
            self.args = args
            self.kwargs = kwargs
            return self.dispatch(request, *args, **kwargs)
        view.view_class = cls
        view.view_initkwargs = initkwargs

        # take name and docstring from class
        update_wrapper(view, cls, updated=())

        # and possible attributes set by decorators
        # like csrf_exempt from dispatch
        update_wrapper(view, cls.dispatch, assigned=())
        return view
```



```
        return view

    def dispatch(self, request, *args, **kwargs):
        # Try to dispatch to the right method; if a method doesn't exist,
        # defer to the error handler. Also defer to the error handler if the
        # request method isn't on the approved list.
        # 在这里做了路由分发 把请求方式跟我们自己写的方法名字匹配上做分发
        if request.method.lower() in self.http_method_names:
            handler = getattr(self, request.method.lower(), self.http_method_not_allowed)
        else:
            handler = self.http_method_not_allowed
        return handler(request, *args, **kwargs)

    def http_method_not_allowed(self, request, *args, **kwargs):
        logger.warning(
            'Method Not Allowed (%s): %s', request.method, request.path,
            extra={'status_code': 405, 'request': request}
        )
        return HttpResponseNotAllowed(self._allowed_methods())
```

CBV首先执行了as_view()方法, 然后在内部做了一个分发本质和FBV是一样的~

以后做接口开发的时候，就要用CBV, 因为需要用到类的调用以及一些特性

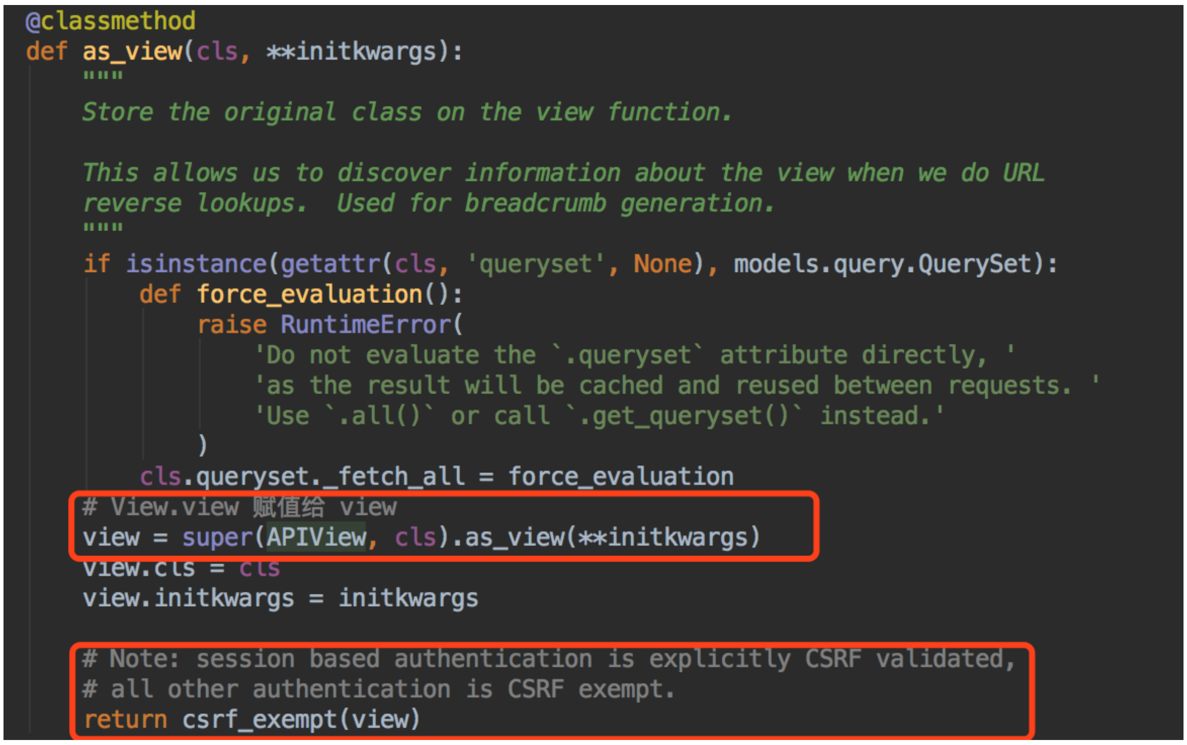
回到顶部(go to top)

APIView和View的区别

- APIView继承了View
    - APIView 重写了as_view以及 dispatch方法
    - 在dispatch里重新封装了request
        - request = Request()

          旧的request变成了_request
          get请求数据 ------> request.query_params
          post请求的数据------>request.data

我们django中写CBV的时候继承的是View，rest_framework继承的是APIView

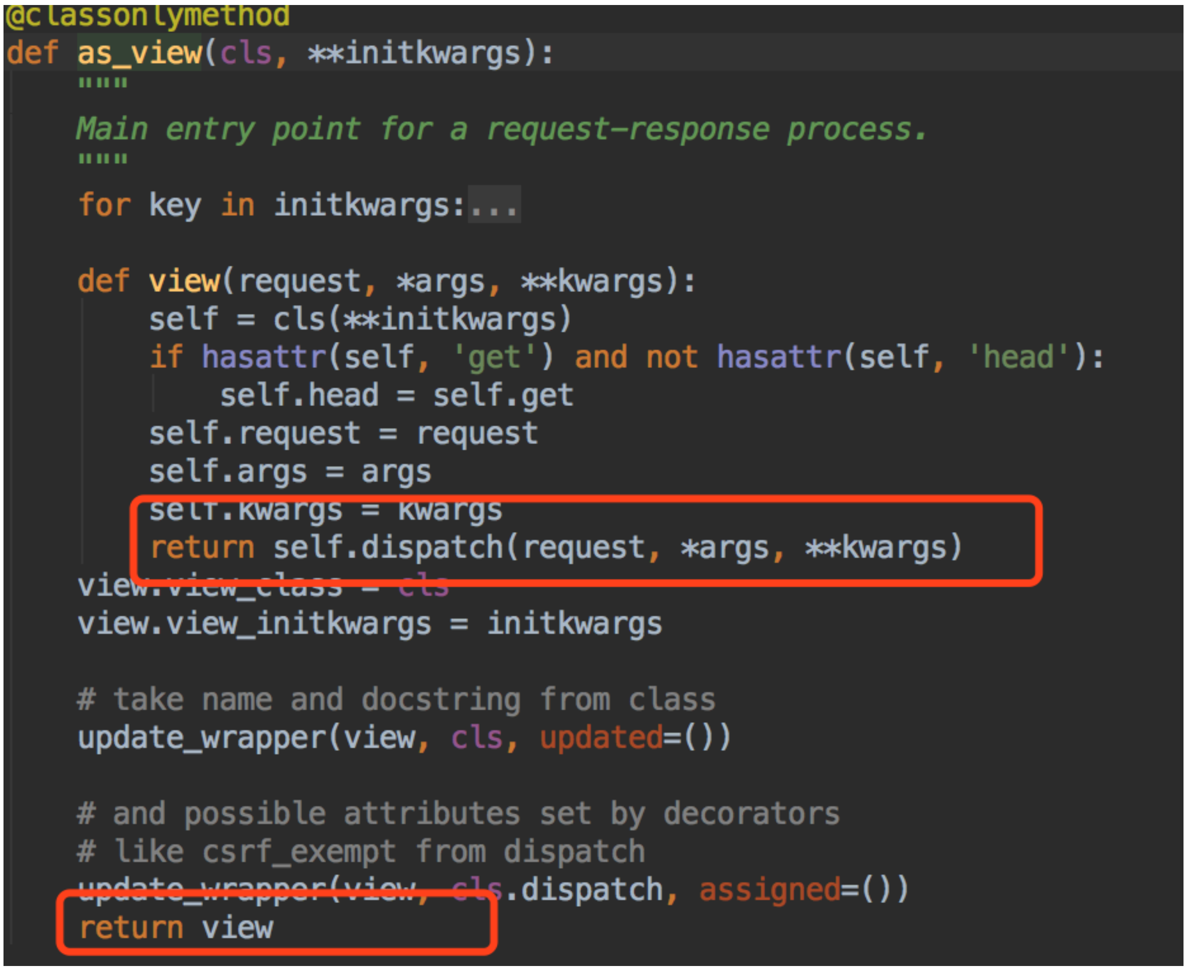


```
@classmethod
def as_view(cls, **initkwargs):
    """
    Store the original class on the view function.

    This allows us to discover information about the view when we do URL
    reverse lookups.  Used for breadcrumb generation.
    """
    if isinstance(getattr(cls, 'queryset', None), models.query.QuerySet):
        def force_evaluation():
            raise RuntimeError(
                'Do not evaluate the `.queryset` attribute directly, '
                'as the result will be cached and reused between requests. '
                'Use `.all()` or call `.get_queryset()` instead.'
            )
        cls.queryset._fetch_all = force_evaluation
    # View.view 赋值给 view
    view = super(APIView, cls).as_view(**initkwargs)
    view.cls = cls
    view.initkwargs = initkwargs

    # Note: session based authentication is explicitly CSRF validated,
    # all other authentication is CSRF exempt.
    return csrf_exempt(view)
```

APIView

我们django中写CBV的时候继承的是View，rest_framework继承的是APIView

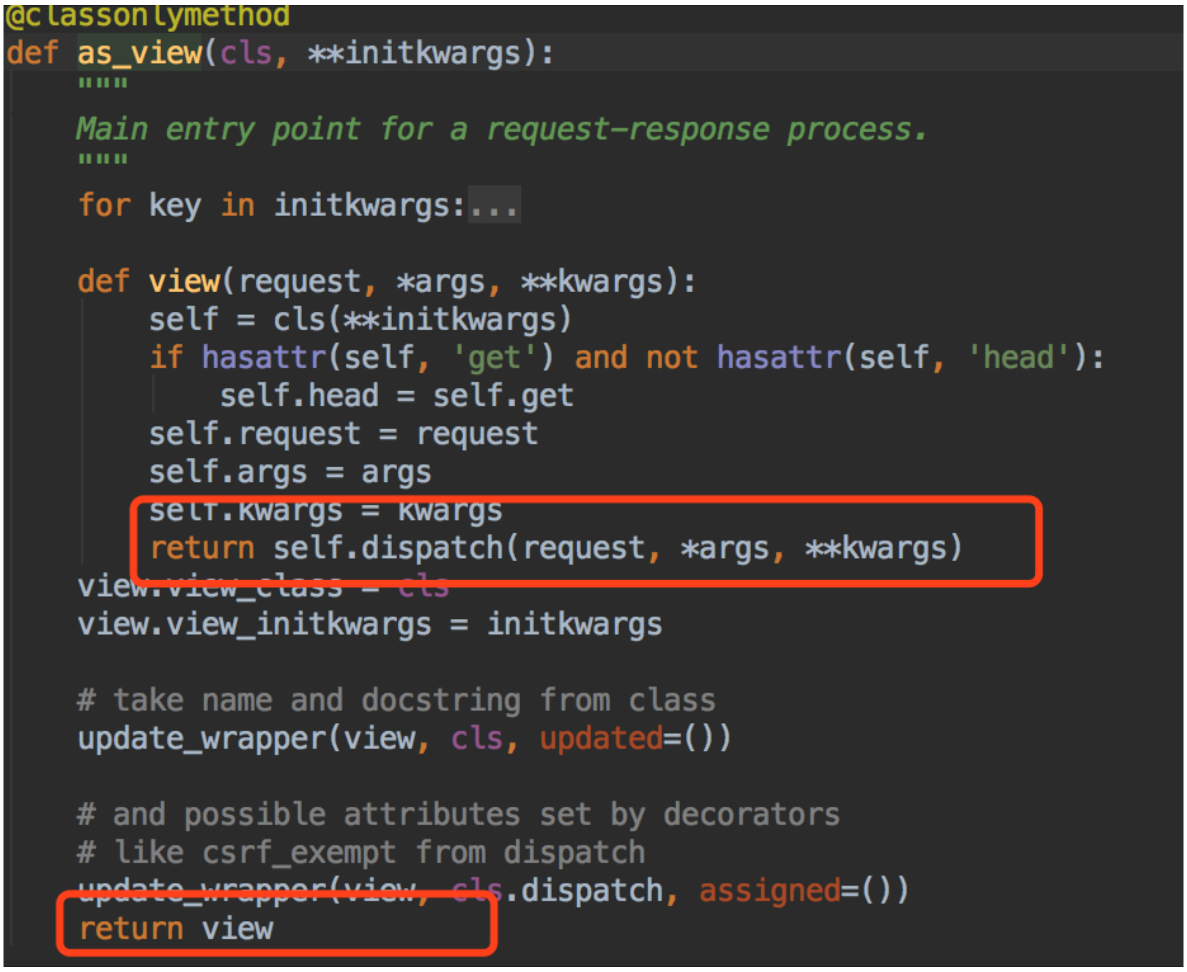
不管是View还是APIView最开始调用的都是as_view()方法



```
@classonlymethod
def as_view(cls, **initkwargs):
    """
    Main entry point for a request-response process.
    """
    for key in initkwargs:...

    def view(request, *args, **kwargs):
        self = cls(**initkwargs)
        if hasattr(self, 'get') and not hasattr(self, 'head'):
            self.head = self.get
        self.request = request
        self.args = args
        self.kwargs = kwargs
        return self.dispatch(request, *args, **kwargs)
    view.view_class = cls
    view.view_initkwargs = initkwargs

    # take name and docstring from class
    update_wrapper(view, cls, updated=())

    # and possible attributes set by decorators
    # like csrf_exempt from dispatch
    update_wrapper(view, cls.dispatch, assigned=())
    return view
```

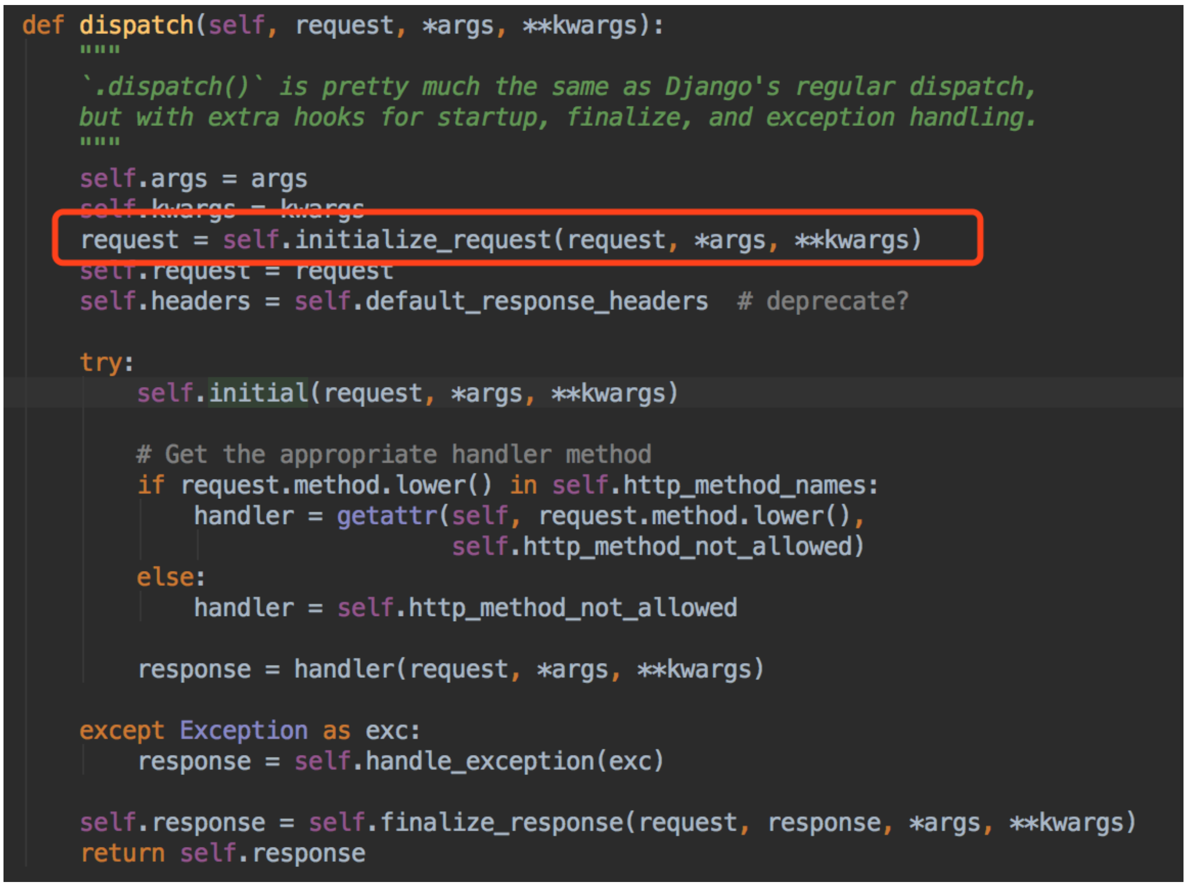
APIView继承了View, 并且执行了View中的as_view()方法，最后把view返回了，用csrf_exempt()方法包裹后去掉了csrf的认证

as_view()方法做了什么???



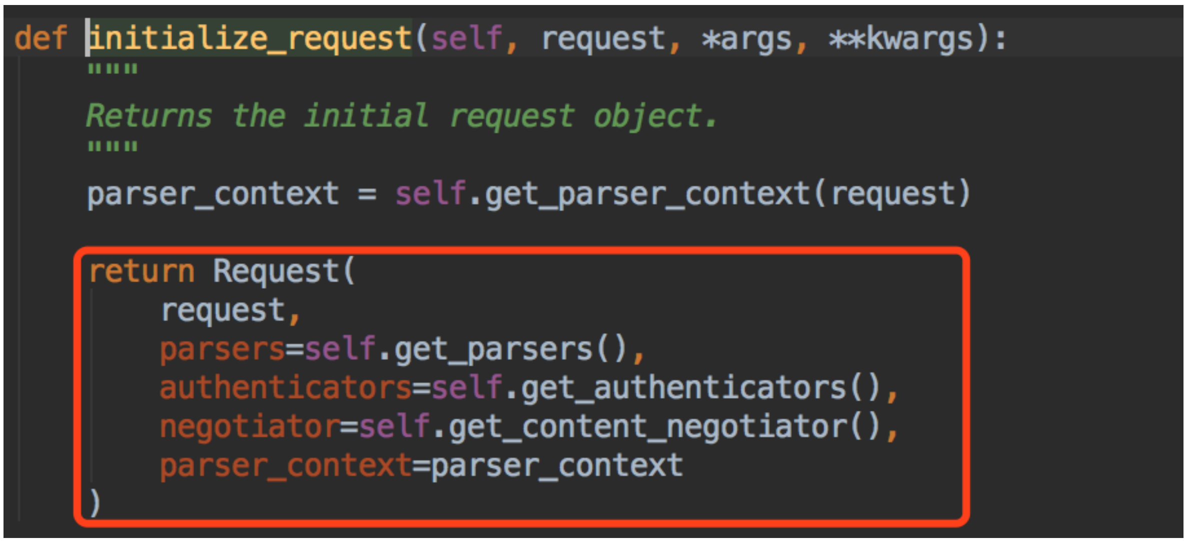
```
@classonlymethod
def as_view(cls, **initkwargs):
    """
    Main entry point for a request-response process.
    """
    for key in initkwargs:...

    def view(request, *args, **kwargs):
        self = cls(**initkwargs)
        if hasattr(self, 'get') and not hasattr(self, 'head'):
            self.head = self.get
        self.request = request
        self.args = args
        self.kwargs = kwargs
        return self.dispatch(request, *args, **kwargs)
    view.view_class = cls
    view.view_initkwargs = initkwargs

    # take name and docstring from class
    update_wrapper(view, cls, updated=())

    # and possible attributes set by decorators
    # like csrf_exempt from dispatch
    update_wrapper(view, cls.dispatch, assigned=())
    return view
```

在View中的as_view方法返回了view函数，而view函数执行了self.dispatch()方法~~但是这里的dispatch方法应该是我们APIView中的



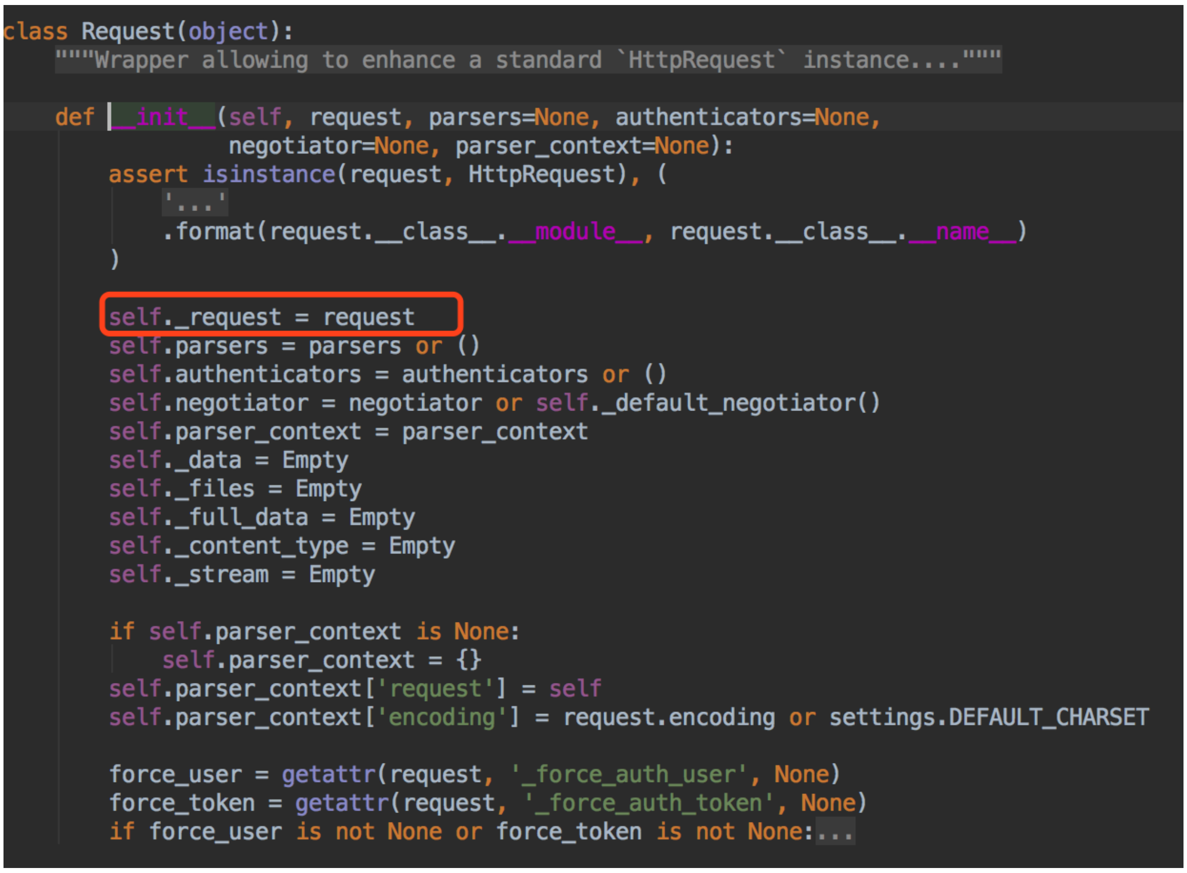
```python
def dispatch(self, request, *args, **kwargs):
    """
    `.dispatch()` is pretty much the same as Django's regular dispatch,
    but with extra hooks for startup, finalize, and exception handling.
    """
    self.args = args
    self.kwargs = kwargs
    request = self.initialize_request(request, *args, **kwargs)
    self.request = request
    self.headers = self.default_response_headers  # deprecate?

    try:
        self.initial(request, *args, **kwargs)

        # Get the appropriate handler method
        if request.method.lower() in self.http_method_names:
            handler = getattr(self, request.method.lower(),
                              self.http_method_not_allowed)
        else:
            handler = self.http_method_not_allowed

        response = handler(request, *args, **kwargs)

    except Exception as exc:
        response = self.handle_exception(exc)

    self.response = self.finalize_response(request, response, *args, **kwargs)
    return self.response
```

request这个变量是经过django封装的uwsgi协议的模块处理过的, 这里通过initialize_request方法再次处理这个变量, 然后赋值给了另一个同名的变量(不同的名称空间)request，并且赋值给了self.request, 也就是我们在视图中用的request.xxx到底是什么~~



```python
def initialize_request(self, request, *args, **kwargs):
    """
    Returns the initial request object.
    """
    parser_context = self.get_parser_context(request)

    return Request(
        request,
        parsers=self.get_parsers(),
        authenticators=self.get_authenticators(),
        negotiator=self.get_content_negotiator(),
        parser_context=parser_context
    )
```

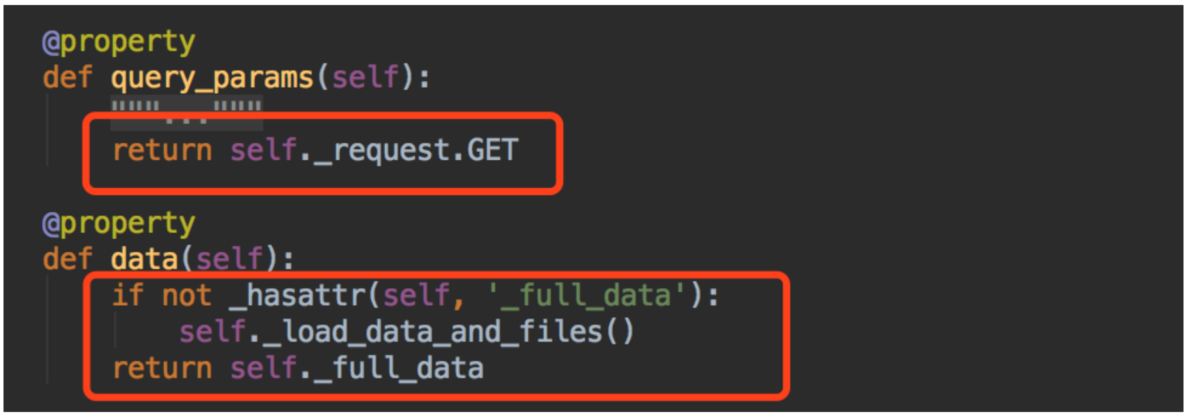
这个方法返回的是Request这个类的实例对象~~我们注意我们看下这个Request类中的第一个参数request，是我们走我们django的时候的原来的request~



```
class Request(object):
    """Wrapper allowing to enhance a standard `HttpRequest` instance...."""

    def __init__(self, request, parsers=None, authenticators=None,
                 negotiator=None, parser_context=None):
        assert isinstance(request, HttpRequest), (
            '...'
            .format(request.__class__.__module__, request.__class__.__name__)
        )

        self._request = request
        self.parsers = parsers or ()
        self.authenticators = authenticators or ()
        self.negotiator = negotiator or self._default_negotiator()
        self.parser_context = parser_context
        self._data = Empty
        self._files = Empty
        self._full_data = Empty
        self._content_type = Empty
        self._stream = Empty

        if self.parser_context is None:
            self.parser_context = {}
        self.parser_context['request'] = self
        self.parser_context['encoding'] = request.encoding or settings.DEFAULT_CHARSET

        force_user = getattr(request, '_force_auth_user', None)
        force_token = getattr(request, '_force_auth_token', None)
        if force_user is not None or force_token is not None:...
```

这个Request类把原来的request赋值给了`self._request`, 也就是说以后`_request`是我们老的request，新的request是我们这个Request类~~

所以, 现在的问题是,继承APIView之后请求来的数据都在哪????



```
@property
def query_params(self):
    """..."""
    return self._request.GET

@property
def data(self):
    if not _hasattr(self, '_full_data'):
        self._load_data_and_files()
    return self._full_data
```

当我们用了rest_framework框架以后，我们的request是重新封装的Request类~

request.query_params 存放的是我们get请求的参数

request.data 存放的是我们所有的数据，包括post请求的以及put，patch请求~~~

相比原来的django的request，我们现在的request更加精简，清晰了~~~

回到顶部(go to top)

序列化

查看数据

使得后端的数据能够被前端的浏览器识别

本质上就是将python数据转换成json数据类型

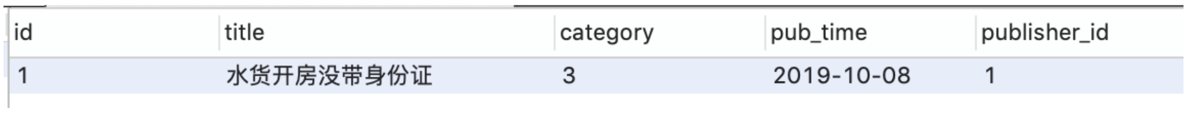
Json格式数据和python格式数据的对应关系

from `/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/json/encoder.py`

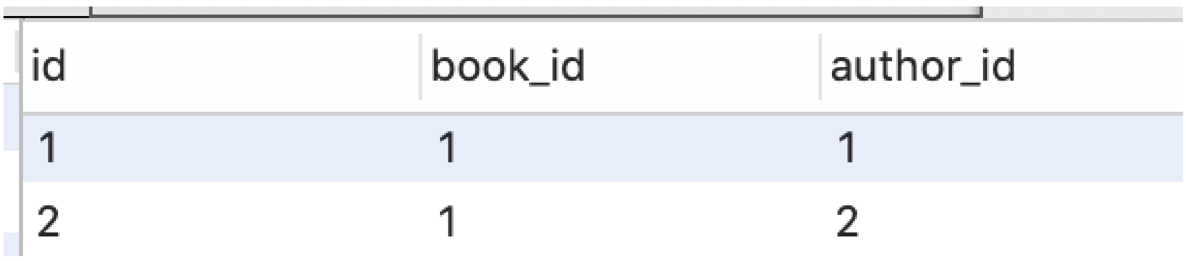
```
Copy+-------------------+---------------+
| Python            | JSON          |
+===================+===============+
| dict              | object        |
+-------------------+---------------+
| list, tuple       | array         |
+-------------------+---------------+
| str               | string        |
+-------------------+---------------+
| int, float        | number        |
+-------------------+---------------+
| True              | true          |
+-------------------+---------------+
| False             | false         |
+-------------------+---------------+
| None              | null          |
+-------------------+---------------+
```

```
/DRFDemo/djangoDemo/models.py
Copyfrom django.db import models


# Create your models here.

__all__= ["Book", "Publisher", "Author"]

class Book(models.Model):
    title = models.CharField(max_length=32)
    CHOICES = ((1, 'Python'), (2, 'Linux'), (3, 'go'))
    category = models.IntegerField(choices=CHOICES)
    pub_time = models.DateField()
    publisher = models.ForeignKey(to="Publisher")
    authors = models.ManyToManyField(to="Author")

class Publisher(models.Model):
    title = models.CharField(max_length=32)


class Author(models.Model):
    name = models.CharField(max_length=32)
```

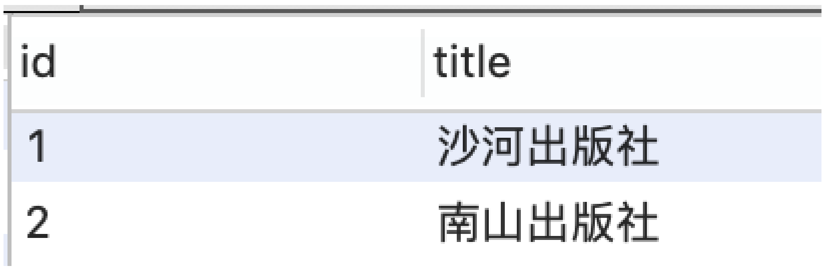
插入点数据



id	title	category	pub_time	publisher_id
1	水货开房没带身份证	3	2019-10-08	1



id	book_id	author_id
1	1	1
2	1	2



id	title
1	沙河出版社
2	南山出版社

```
/DRFDemo/DRFDemo/urls.py
Copyfrom django.conf.urls import url,include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^demo/', include("djangoDemo.urls")),
]
```

```
/DRFDemo/djangoDemo/urls.py
Copyfrom django.conf.urls import url
from .views import BookView

urlpatterns=[
    url(r'^book', BookView.as_view())
]
```

第一版 用values方法取数据

```
/DRFDemo/djangoDemo/views.py
Copyfrom django.http import JsonResponse
from django.shortcuts import render, HttpResponse
from django.views import View
from .models import Book,Publisher
import json
```

```
class BookView(View):
    def get(self, request):
        book_queryset = Book.objects.values("id", "title",
"pub_time",'publisher')
        book_list = list(book_queryset)
        '''python自带json模块不支持时间格式，需要对时间格式支持需要用django的
JsonResponse模块'''
        # ret = json.dumps(book_list, ensure_ascii=False)
        # return HttpResponse(ret)
        ret=[]
        for book in book_list:
            print(book)
            book['publisher'] = {
                'id': book['id'],
                'title':
Publisher.objects.filter(id=book['publisher']).first().title
            }
            ret.append(book)
        return JsonResponse(ret, safe=False, json_dumps_params={"ensure_ascii":
False})
```

输出结果

```
Copy[
    {
        "id": 1,
        "title": "水货开房没带身份证",
        "pub_time": "2019-10-08",
        "publisher": {
            "id": 1,
            "title": "沙河出版社"
        }
    }
]
```

第二版 使用django序列化组件获取数据

```
/DRFDemo/djangoDemo/views.py
Copyfrom django.shortcuts import render, HttpResponse
from django.views import View
from .models import Book,Publisher
from django.core import serializers

class BookView(View):
    def get(self,request):
        book_queryset = Book.objects.all()
        data = serializers.serialize("json",book_queryset, ensure_ascii=False)
        return HttpResponse(data)
```

输出结果

```
Copy[
    {
        "model": "djangoDemo.book",
        "pk": 1,
        "fields": {
            "title": "水货开房没带身份证",
            "category": 3,
            "pub_time": "2019-10-08",
            "publisher": 1,
            "authors": [
                1,
                2
            ]
        }
    }
]
```

第三版 使用djangorestframework

前面2版的缺点: 不能处理外键关系

```
新建一个app
Copypython3 manager.py startapp SerDemo
安装djangorestframework
Copypip3 install djangorestframework
注册rest_framework这个app, 如果不注册, 通过drf序列化的数据无法在页面上渲染
```

第一步: 新建序列化器

序列化器中的定义的字段对象必须和ORM字段的名字是一致的

```
/DRFDemo/SerDemo/serializers.py
Copyfrom rest_framework import serializers


class PublishSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=32)

class AuthorSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=32)

class BookSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=32)
    pub_time = serializers.DateField()
    category = serializers.CharField(source="get_category_display")

    # 外键关系
    publisher = PublishSerializer()
    authors = AuthorSerializer(many=True)
```

第二步: 使用自定义的序列化器序列化queryset

1. 把模型对象(ORM对象)放入序列化器进行字段匹配, 匹配上的字段进行序列化 匹配不上丢弃
2. 序列化好的数据在ser_obj.data中
3. 外键关系的序列化是嵌套的序列化器对象 注意: `many=True`

   只要指定了many=True, 则肯定是可迭代对象

```
/DRFDemo/DRFDemo/urls.py
Copyfrom django.conf.urls import url,include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^demo/', include("djangoDemo.urls")),
    url(r'^api/', include("SerDemo.urls")),
]
```

```
/DRFDemo/SerDemo/urls.py
Copyfrom django.conf.urls import url,include
from .views import BookView
urlpatterns = [
    url(r'^book/', BookView.as_view()),
]
```

```
/DRFDemo/SerDemo/views.py
Copyfrom django.shortcuts import render
from rest_framework.views import APIView
from djangoDemo.models import Book
from .serializers import BookSerializer
from rest_framework.response import Response
# Create your views here.

class BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
                '''
                用序列化器进行序列化
                加了many=True这个参数, 则会先循环book_queryset这个对象,把循环出来的每一
个对象放到序列化器中序列                化, 这些对象是ORM对象, 把这些对象和序列化器中的对象
进行匹配, 匹配上的则序列化, 匹配不上的则丢弃
                '''
        ser_obj = BookSerializer(book_queryset, many=True)
        '''
          BookSerializers(<QuerySet [<Book: 水货开房没带身份证>]>, many=True):
```

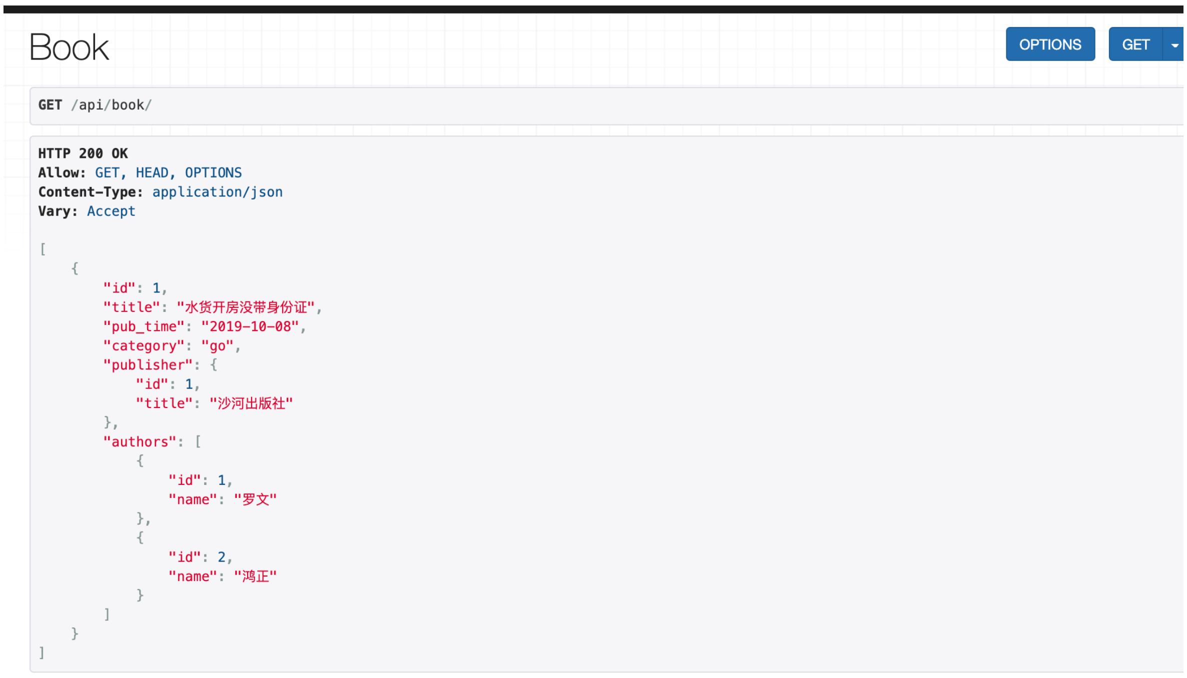
```
        id = IntegerField()
        title = CharField(max_length=32)
        category = CharField(source='get_category_display')
        pub_time = DateField()
        publisher = PublisherSerializers():
        id = IntegerField()
        name = CharField(max_length=32)
        authors = AuthorSerializers(many=True):
        id = IntegerField()
        name = CharField(max_length=32)
      '''
      return Response(ser_obj.data)
```

第三步: 注册rest_framwork app

使得页面中渲染rest_framwork的页面

```
/DRFDemo/DRFDemo/settings.py
CopyINSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'djangoDemo.apps.DjangodemoConfig',
    "SerDemo.apps.SerdemoConfig",
    "rest_framework.apps.RestFrameworkConfig",
]
```

输出结果



```
GET /api/book/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "title": "水货开房没带身份证",
        "pub_time": "2019-10-08",
        "category": "go",
        "publisher": {
            "id": 1,
            "title": "沙河出版社"
        },
        "authors": [
            {
                "id": 1,
                "name": "罗文"
            },
            {
                "id": 2,
                "name": "鸿正"
            }
        ]
    }
]
```

相关报错

报错1

```
CopyGot AttributeError when attempting to get a value for field `id` on
serializer `BookSerializers`.
The serializer field might be named incorrectly and not match any attribute or
key on the `QuerySet` instance.
Original exception text was: 'QuerySet' object has no attribute 'id'.
```

原因: 序列化queryset对象时候, 没有加 `many=True`

```
Copyclass BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
        ser_obj = serializers.BookSerializers(book_queryset)
        return Response(ser_obj.data)
```

解决: 加上 `many=True`

```
Copyclass BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
        ser_obj = serializers.BookSerializers(book_queryset,many=True)
        return Response(ser_obj.data)
```

报错2

```
CopyObject of type 'ListSerializer' is not JSON serializable
```

原因: 最后返回到页面的是一个序列化的对象, 并不是一个具体的数据

```
Copyclass BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
        ser_obj = serializers.BookSerializers(book_queryset,many=True)
        return Response(ser_obj)
```

解决: 返回序列化的数据

```
Copyclass BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
        ser_obj = serializers.BookSerializers(book_queryset,many=True)
        return Response(ser_obj.data)
```

[回到顶部(go to top)]

反序列化

新增数据

通过post方法写入数据库, 前端通过json格式传到后端, 后端反序列化成python数据类型进行处理写入数据库

思路

```
Copy-- 确定新增的数据结构
        -- 序列化器
            -- 正序和反序列化字段不统一
            -- required=False 只序列化不走校验
            -- read_only=True  只序列化用
            -- write_only=True  只反序列化用
            -- 重写create方法
        -- 验证通过返回ser_obj.validated_data
        -- 验证不通过返回ser_obj.errors
```

```
/DRFDemo/DRFDemo/urls.py
Copyfrom django.conf.urls import url,include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^demo/', include("djangoDemo.urls")),
    url(r'^api/', include("SerDemo.urls")),
]
```

```
/DRFDemo/SerDemo/urls.py
Copyfrom django.conf.urls import url
from .views import BookView, BookEditView
urlpatterns = [
    url(r'^book/', BookView.as_view()),
]
```

第一步: 确认字段

确认需要校验的序列化的字段和反序列的字段, 并重写create方法

```
/DRFDemo/SerDemo/serializers.py
Copyfrom rest_framework import serializers
from djangoDemo.models import Book
import copy


class PublishSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=32)

class AuthorSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=32)

class BookSerializer(serializers.Serializer):
    id = serializers.IntegerField(required=False)
    title = serializers.CharField(max_length=32)
    pub_time = serializers.DateField()

    # read_only=True  反序列化不校验该字段，序列化时校验该字段
    category = serializers.CharField(source="get_category_display",
read_only=True)

    # 外键关系
    publisher = PublishSerializer(read_only=True)
    # 内部通过外键关系的id找到了publish_obj
    # PublishSerializer(publish_obj)

    authors = AuthorSerializer(many=True, read_only=True)
    # 这里加上 many=True，因为多对多的缘故，内部通过外键关系的id找到的authors_obj是可迭代
对象的数据类型，因为有多个数据，所以需要通过可迭代的数据类型包裹


    # write_only=True  序列化时候不校验该字段，反序列化校验该字段
    post_category = serializers.IntegerField(write_only=True)
    publisher_id = serializers.IntegerField(write_only=True)
    author_list = serializers.ListField(write_only=True)

    def create(self, validated_data):
        # validated_data  校验通过的数据  就是book_obj
        # 通过ORM操作给Book表增加数据
        print(validated_data)
        '''
        前端传过来的数据:
        {   "title": "水货国庆节很寂寞",
            "pub_time": "2019-10-08",
            "post_category": 1,
            "publisher_id": 1,
            "author_list": [1,2]
            }
        '''
        cleaned_data = copy.deepcopy(validated_data)
        cleaned_data.pop('author_list')
        cleaned_data['category']=cleaned_data['post_category']
        cleaned_data.pop('post_category')
        book_obj = Book.objects.create(**cleaned_data)
        book_obj.authors.add(*validated_data['author_list'])
        return book_obj
```

第二步: 对前端的数据反序列化

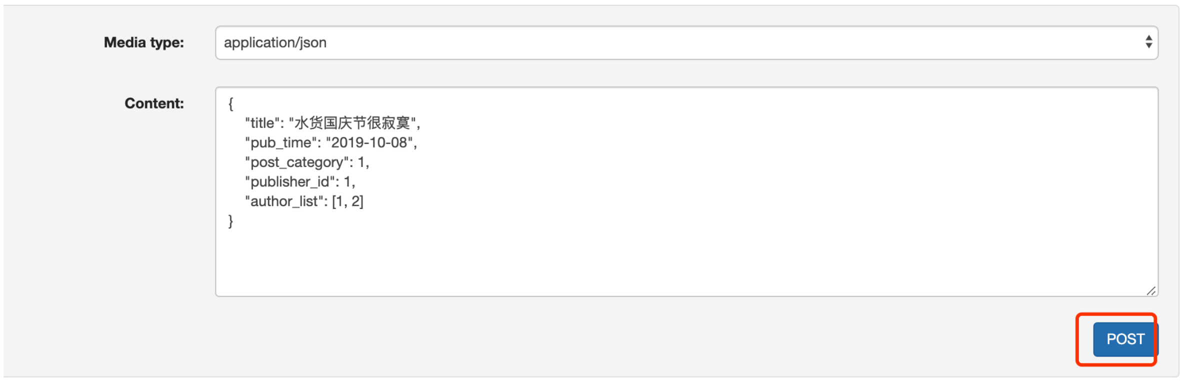
这里的反序列化完成了校验和转换2个步骤

- 校验前端传过来的字段的名字是否和序列化器中定义的需要反序列化的字段的名字是否一样
- 如果校验通过了, 则将 `json` 数据类型转换为 `python` 数据类型, 保存在序列化器的对象的属性 `validated_data` 中

```
Copyfrom django.shortcuts import render
from rest_framework.views import APIView
from djangoDemo.models import Book
from .serializers import BookSerializer
from rest_framework.response import Response
# Create your views here.

class BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
        # 用序列化器进行序列化
        ser_obj = BookSerializer(book_queryset, many=True)
        return Response(ser_obj.data)

    def post(self, request):
        # 确定数据类型以及数据以及数据结构
        # 对前端的数据进行校验
        book_obj = request.data
        ser_obj = BookSerializer(data=book_obj)
        if ser_obj.is_valid():
            ser_obj.save()
            return Response(ser_obj.validated_data)
        return Response(ser_obj.errors)
```

第三步: 新增数据

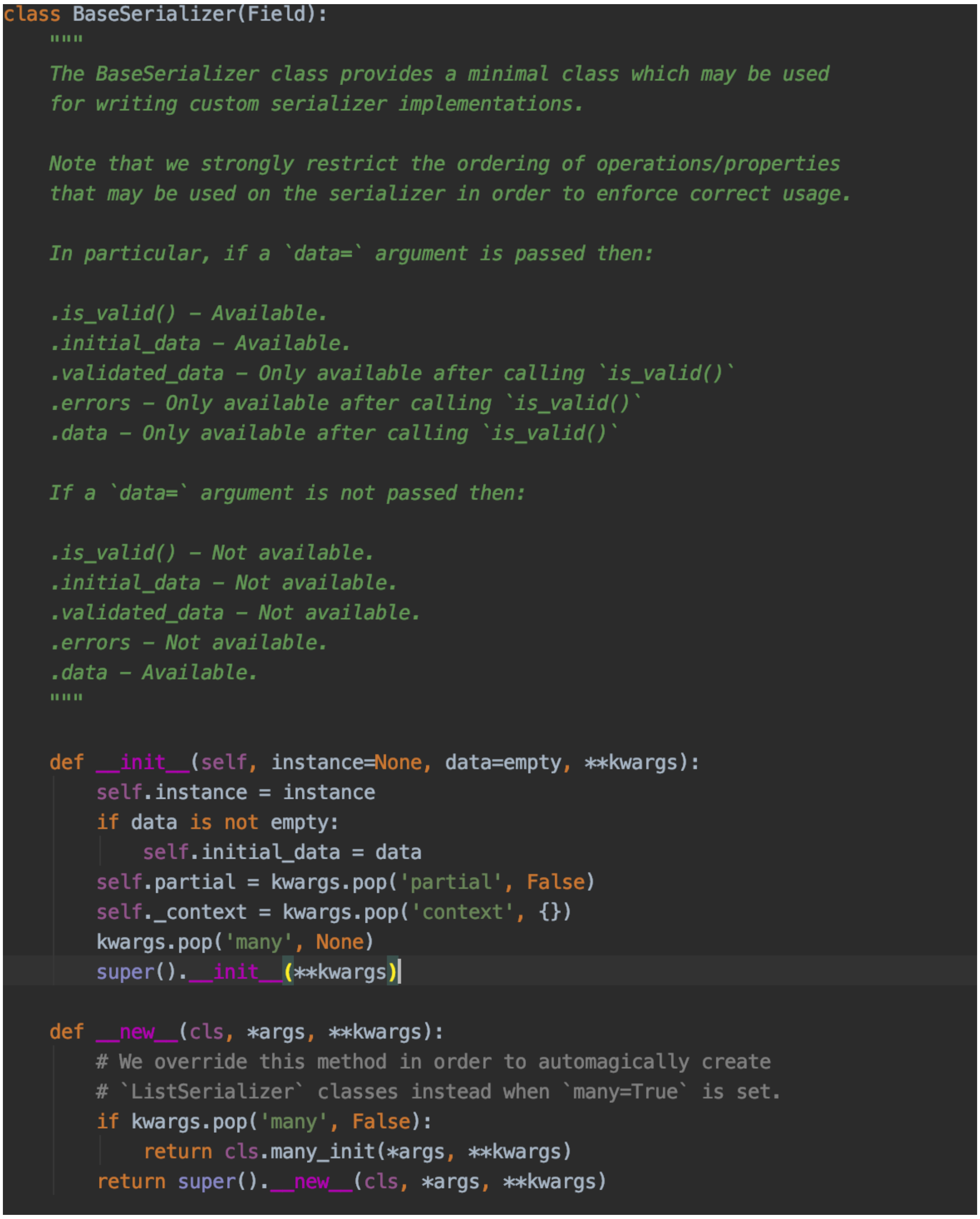
```
Copy{
    "title": "水货国庆节很寂寞",
    "pub_time": "2019-10-08",
    "post_category": 1,
    "publisher_id": 1,
    "author_list": [1, 2]
}
```



```
{
    "title": "水货国庆节很寂寞",
    "pub_time": "2019-10-08",
    "post_category": 1,
    "publisher_id": 1,
    "author_list": [1, 2]
}
```

注意!!! 序列化和反序列化的时候, 其实就是实例化继承了serializers.Serializer的序列化类进行相应的操作

Serializer的构造方法为:
Serializer(instance=None, data=empty, **kwargs)



```python
class BaseSerializer(Field):
    """
    The BaseSerializer class provides a minimal class which may be used
    for writing custom serializer implementations.

    Note that we strongly restrict the ordering of operations/properties
    that may be used on the serializer in order to enforce correct usage.

    In particular, if a `data=` argument is passed then:

    .is_valid() - Available.
    .initial_data - Available.
    .validated_data - Only available after calling `is_valid()`
    .errors - Only available after calling `is_valid()`
    .data - Only available after calling `is_valid()`

    If a `data=` argument is not passed then:

    .is_valid() - Not available.
    .initial_data - Not available.
    .validated_data - Not available.
    .errors - Not available.
    .data - Available.
    """

    def __init__(self, instance=None, data=empty, **kwargs):
        self.instance = instance
        if data is not empty:
            self.initial_data = data
        self.partial = kwargs.pop('partial', False)
        self._context = kwargs.pop('context', {})
        kwargs.pop('many', None)
        super().__init__(**kwargs)

    def __new__(cls, *args, **kwargs):
        # We override this method in order to automagically create
        # `ListSerializer` classes instead when `many=True` is set.
        if kwargs.pop('many', False):
            return cls.many_init(*args, **kwargs)
        return super().__new__(cls, *args, **kwargs)
```

- 用于序列化时，校验后端数据, 将python的数据类型转换为前端可以识别的json数据类型, 将模型类对象传入instance参数
- 用于反序列化时，校验前端数据, 并将json数据类型转换为python的数据类型, 将要被反序列化的数据传入data参数

相关报错

报错1

```
CopyAssertionError at /api/book/
Cannot call `.is_valid()` as no `data=` keyword argument was passed when
instantiating the serializer instance.
```

问题: 反序列化的时候, 将反序列化的数据传入了instance参数, 默认不用关键字传参是将数据传入instance参数

```
Copyclass BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
        ser_obj = BookSerializers(book_queryset, many=True)
        return Response(ser_obj.data)

    def post(self, request):
        book_queryset = request.data
        ser_obj = BookSerializers(book_queryset)
        if ser_obj.is_valid():
            ser_obj.save()
            return Response(ser_obj.validated_data)
        return Response(ser_obj.errors)
```

解决: 将反序列化的数据传入用关键字传参传给data参数

```
Copyclass BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
        ser_obj = BookSerializers(book_queryset, many=True)
        return Response(ser_obj.data)

    def post(self, request):
        book_queryset = request.data
        ser_obj = BookSerializers(data=book_queryset)
        if ser_obj.is_valid():
            ser_obj.save()
            return Response(ser_obj.validated_data)
        return Response(ser_obj.errors)
```

更新数据

第一步: 新建子路由关系

```
/DRFDemo/DRFDemo/urls.py
Copyfrom django.conf.urls import url
from .views import BookView, BookEditView
urlpatterns = [
    url(r'^book/', BookView.as_view()),
    url(r'^books/(?P<id>\d+)', BookEditView.as_view()),
]
```

第二步: 新建视图类

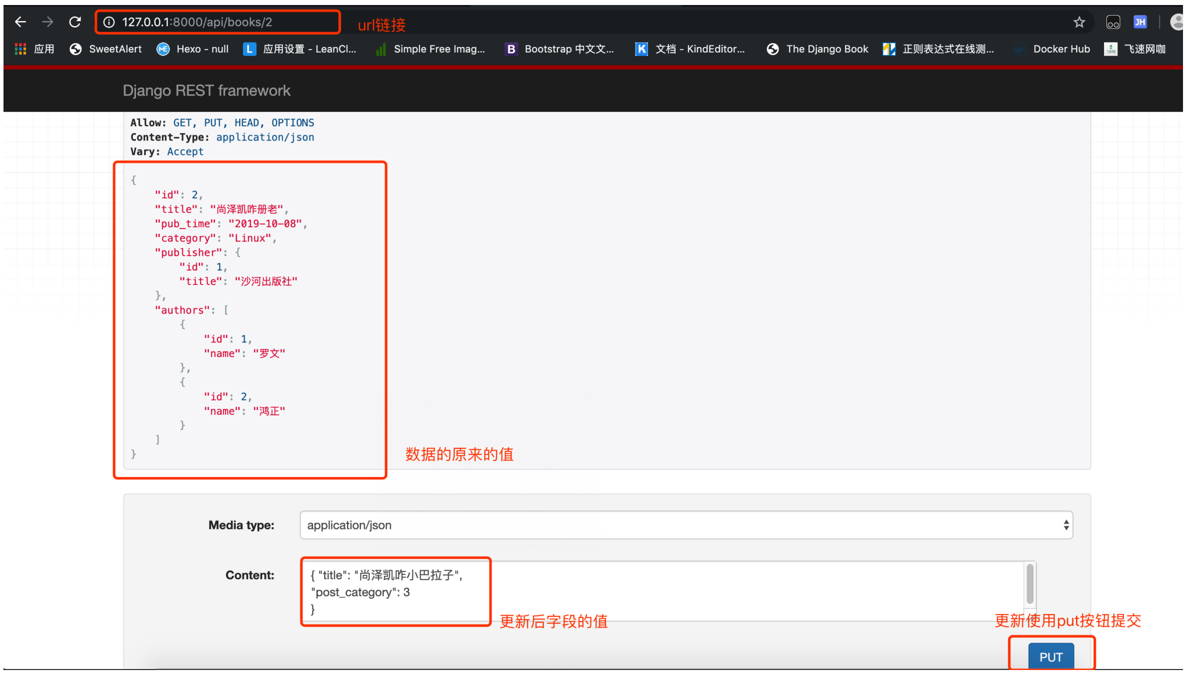
```
/DRFDemo/SerDemo/views.py
Copyfrom rest_framework.views import APIView
from djangoDemo.models import Book
from .serializers import BookSerializer
from rest_framework.response import Response

class BookEditView(APIView):
    def get(self, request, id):
        book_obj = Book.objects.filter(id=id).first()
        ser_obj = BookSerializer(book_obj)
        return Response(ser_obj.data)

    def put(self, request, id):
        book_obj = Book.objects.filter(id=id).first()
        # partial=True  对序列化器中的update方法中，只对部分字段进行校验
        ser_obj = BookSerializer(instance=book_obj, data=request.data,
partial=True)
        if ser_obj.is_valid():
            ser_obj.save()
            return Response(ser_obj.validated_data)
        return Response(ser_obj.errors)
```

第三步: 序列化器中重写`update`方法

```
/DRFDemo/SerDemo/serializers.py
Copyfrom rest_framework import serializers
from djangoDemo.models import Book
import copy

class BookSerializer(serializers.Serializer):
    id = serializers.IntegerField(required=False)
    title = serializers.CharField(max_length=32)
    pub_time = serializers.DateField()

    # read_only=True  反序列化不校验该字段，序列化时校验该字段
    category = serializers.CharField(source="get_category_display",
read_only=True)

    # 外键关系
    publisher = PublishSerializer(read_only=True)
    # 内部通过外键关系的id找到了publish_obj
    # PublishSerializer(publish_obj)
    authors = AuthorSerializer(many=True, read_only=True)

    # write_only=True  序列化时候不校验该字段，反序列化校验该字段
    post_category = serializers.IntegerField(write_only=True)
    publisher_id = serializers.IntegerField(write_only=True)
    author_list = serializers.ListField(write_only=True)

    def create(self, validated_data):
        # validated_data  校验通过的数据  就是book_obj
        # 通过ORM操作给Book表增加数据
        print(validated_data)
        '''
        前端传过来的数据:
        {'title': '水货开房没带身份证',
         'pub_time': datetime.date(2019, 10, 8),
         'post_category': 1,
         'publisher_id': 1,
         'author_list': [1, 2]}
        '''
        cleaned_data = copy.deepcopy(validated_data)
        cleaned_data.pop('author_list')
        cleaned_data['category'] = cleaned_data['post_category']
        cleaned_data.pop('post_category')
        book_obj = Book.objects.create(**cleaned_data)
        book_obj.authors.add(*validated_data['author_list'])
        return book_obj

    def update(self, instance, validated_data):
        # instance 更新的book_obj 对象
        # validated_data  前端传过来的并且校验通过的数据
        # validated data中的键值肯定是
title|pub_time|post_category|publisher_id|author_list|
        # ORM 做更新操作
        instance.title = validated_data.get('title', instance.title)
        instance.pub_time = validated_data.get('pub_time', instance.pub_time)
        instance.category = validated_data.get('post_category',
instance.category)
        instance.publisher_id = validated_data.get('publisher_id',
instance.publisher_id)
        if validated_data.get('author_list'):
            instance.author.set(validated_data['author_list'])
        instance.save()

        return instance
```

第四步: 更新数据

```
Copy{ "title": "尚泽凯咋小巴拉子",
"post_category": 3
}
```

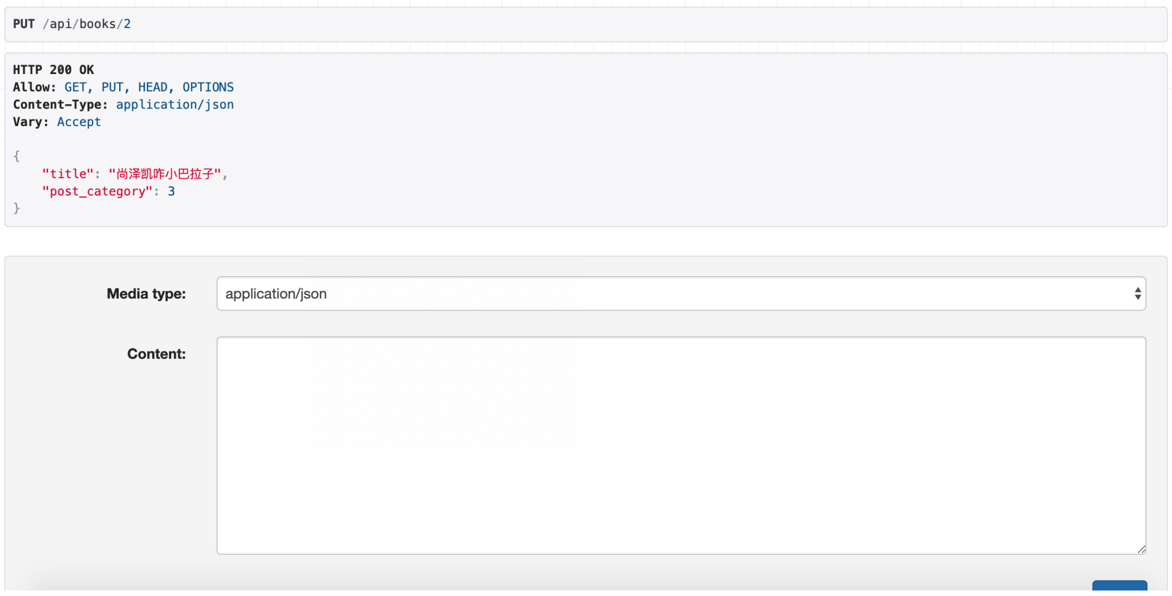


```
Allow: GET, PUT, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 2,
    "title": "尚泽凯咋册老",
    "pub_time": "2019-10-08",
    "category": "Linux",
    "publisher": {
        "id": 1,
        "title": "沙河出版社"
    },
    "authors": [
        {
            "id": 1,
            "name": "罗文"
        },
        {
            "id": 2,
            "name": "鸿正"
        }
    ]
}
```

Media type: application/json

Content:

```
{ "title": "尚泽凯咋小巴拉子",
"post_category": 3
}
```



```
PUT /api/books/2

HTTP 200 OK
Allow: GET, PUT, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "title": "尚泽凯咋小巴拉子",
    "post_category": 3
}
```

Media type: application/json

Content:

```
HTTP 200 OK
Allow: GET, PUT, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 2,
    "title": "尚泽凯咋小巴拉子",
    "pub_time": "2019-10-08",
    "category": "go",
    "publisher": {
        "id": 1,
        "title": "沙河出版社"
    },
    "authors": [
        {
            "id": 1,
            "name": "罗文"
        },
        {
            "id": 2,
            "name": "鸿正"
        }
    ]
}
```

进一步校验数据

```
/DRFDemo/SerDemo/serializers.py
Copyfrom rest_framework import serializers
from djangoDemo.models import Book
import copy


class PublishSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    title = serializers.CharField(max_length=32)
```

```
class AuthorSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=32)

'''自定义校验'''
def my_validate(value):
    print('11111111')
    if '敏感信息' in value.lower():
        raise serializers.ValidationError('有敏感词汇')

    return value

class BookSerializer(serializers.Serializer):
    id = serializers.IntegerField(required=False)
    title = serializers.CharField(max_length=32, validators=[my_validate, ])
    pub_time = serializers.DateField()

    # read_only=True  反序列化不校验该字段，序列化时校验该字段
    category = serializers.CharField(source="get_category_display",
read_only=True)

    # 外键关系
    publisher = PublishSerializer(read_only=True)
    # 内部通过外键关系的id找到了publish_obj
    # PublishSerializer(publish_obj)
    authors = AuthorSerializer(many=True, read_only=True)

    # write_only=True  序列化时候不校验该字段，反序列化校验该字段
    post_category = serializers.IntegerField(write_only=True)
    publisher_id = serializers.IntegerField(write_only=True)
    author_list = serializers.ListField(write_only=True)

    def create(self, validated_data):
        # validated_data  校验通过的数据  就是book_obj
        # 通过ORM操作给Book表增加数据
        print(validated_data)
        '''
        前端传过来的数据:
        {'title': '水货开房没带身份证',
         'pub_time': datetime.date(2019, 10, 8),
         'post_category': 1,
         'publisher_id': 1,
         'author_list': [1, 2]}
        '''
        cleaned_data = copy.deepcopy(validated_data)
        cleaned_data.pop('author_list')
        cleaned_data['category'] = cleaned_data['post_category']
        cleaned_data.pop('post_category')
        book_obj = Book.objects.create(**cleaned_data)
        book_obj.authors.add(*validated_data['author_list'])
        return book_obj

    def update(self, instance, validated_data):
        # instance 更新的book_obj 对象
        # validated_data  前端传过来的并且校验通过的数据
        # validated data中的键值肯定是
title|pub_time|post_category|publisher_id|author_list|
        # ORM 做更新操作
        instance.title = validated_data.get('title', instance.title)
        instance.pub_time = validated_data.get('pub_time', instance.pub_time)
        instance.category = validated_data.get('post_category',
instance.category)
        instance.publisher_id = validated_data.get('publisher_id',
instance.publisher_id)
        if validated_data.get('author_list'):
            instance.author.set(validated_data['author_list'])
        instance.save()

        return instance

    '''单个字段校验'''
    def validate_title(self, value):
        print(22222222)
        # value就是title的值
        if 'python' not in value.lower():
            raise serializers.ValidationError("标题必须含有python")
        return value


    '''多个字段校验'''
    def validate(self, attrs):
        print(333333333333)
        # attrs  字典有你传过来的所有字段
        print(attrs)
        if "python" in attrs['title'].lower() and attrs['post_category'] == 1:
            return attrs
        else:
            raise serializers.ValidationError('分类或标题不符合要求')
```

权重比较: 自定义校验的函数 (my_validate) > 单个字段校验的方法(validate_title) > 多个字段校验的方法(validate)

回到顶部(go to top)

ModelSerializers 序列化

```
/DRFDemo/djangoDemo/models.py
Copyfrom django.db import models

# Create your models here.

field = ['Book', 'Publisher', 'Author']

class Book(models.Model):
    title = models.CharField(max_length=32)
    CHOICES = ((1, 'go'), (2, 'linux'), (3, 'python'))
    category = models.IntegerField(choices=CHOICES)
    publisher = models.ForeignKey(to='Publisher')
    author = models.ManyToManyField(to='Author')
    def __str__(self):
        return self.title


class Publisher(models.Model):
    name = models.CharField(max_length=32)

    def __str__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=32)

    def __str__(self):
        return self.name
```

```
/DRFDemo/DRFDemo/admin.py
Copyfrom django.contrib import admin

# Register your models here.

from . import models


for table in models.field:
    admin.site.register(getattr(models, table))
```

```
/drfdemo2/drfdemo2/settings.py
CopyINSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'djangomodel.apps.DjangomodelConfig',
    'drfapp.apps.DrfappConfig',
    'rest_framework.apps.RestFrameworkConfig',
]
```

/DRFDemo/SerDemo/serializers.py`

```
Copyfrom rest_framework import serializers
from djangomodel.models import Book

def my_validate(value):
    if '敏感信息' in value.lower():
        raise serializers.ValidationError('有敏感词汇')
```

```
class BookSerializers(serializers.ModelSerializer):
    bookname = serializers.SerializerMethodField()

    def get_bookname(self, object):
        return object.title


    # 一旦定义的字段名使用 SerializerMethodField这个方法，则就可以接收下面定义的方法(get_
字段名)的返回值，默认是read_only=True
    category_info = serializers.SerializerMethodField()

    def get_category_info(self, object):
        # obj 就是序列化的每个book 对象
        # print('obj', obj)
        return object.get_category_display()  # get_字段名_display() 这个函数返回的
是字段里元组中的第二个元素

    author_info = serializers.SerializerMethodField()

    def get_author_info(self, object):
        author_obj = object.author.all()
        return [{'id': author.id, 'name': author.name} for author in author_obj]

    publish_info = serializers.SerializerMethodField()

    def get_publish_info(self, object):
        # object 就相当于是一本书
        publisher_obj = object.publisher
        return {'id': publisher_obj.id, 'name': publisher_obj.name}

    class Meta:
        model = Book
        # fileds指定字段，__all__显示所有字段，exclude排除字段， fields和exclude 不能
共存， 只能指定一个
        fields = '__all__'
        # exclude = ['id']
        # depth = 1 会将所有的外键字段加上选项 read_only = True，并且会拿到所有字段，一
般在开发中不用
        # depth = 1
        extra_kwargs = {'category': {'write_only': True},
                        'author': {'write_only': True},
                        'publisher': {'write_only': True},
                        'title': {'write_only': True, 'validators':
[my_validate, ]},

        }

        category_id = serializers.IntegerField(write_only=True)
        publisher_id = serializers.ListField(write_only=True)
```

```
/DRFDemo/SerDemo/urls.py
Copyfrom django.conf.urls import url
from . import views


urlpatterns = [
    url(r'^books/$', views.BookView.as_view()),
    url(r'^book/(?P<id>\d+)/$', views.BookEdit.as_view())
]
```

```
/DRFDemo/SerDemo/views.py
Copyfrom django.shortcuts import render

# Create your views here.
from rest_framework.views import APIView
from rest_framework.response import Response
from djangomodel.models import Book
from .serializers import BookSerializers


class BookView(APIView):
    def get(self, request):
        book_queryset = Book.objects.all()
        ser_obj = BookSerializers(book_queryset, many=True)
        return Response(ser_obj.data)

    def post(self, request):
        book_queryset = request.data
        '''
        前端数据>>>>
         {
        "title": "陈骏删库跑路了",
        "category": 1,
        "author": [1, 2],
        "publisher": 1
        }
        '''

        ser_obj = BookSerializers(data=book_queryset)
        if ser_obj.is_valid():
            ser_obj.save()
            return Response(ser_obj.validated_data)
        return Response(ser_obj.errors)



class BookEdit(APIView):
    def get(self, request, id):
        book_queryset = Book.objects.filter(id=id).first()
        ser_obj = BookSerializers(book_queryset)
        return Response(ser_obj.data)


    def patch(self, request, id):
        book_queryset = Book.objects.filter(id=id).first()
        '''
         {
        "title": "陈骏删库跑路了",
        "category": 1,
        "author": [1, 2],
        "publisher": 1
        }
        '''

        # partial=True  对序列化器中的update方法中，只对部分字段进行校验
        ser_obj = BookSerializers(instance=book_queryset, data=request.data, 
partial=True)
        if ser_obj.is_valid():
            ser_obj.save()
            return Response(ser_obj.validated_data)
        return Response(ser_obj.errors)
```

分类: django