**Django Rest Framework API指南**

Django Rest Framework 所有API如下：

* [Request 请求](https://www.cnblogs.com/victorwu/p/7418368.html#1)
* [Response 响应](https://www.cnblogs.com/victorwu/p/7418368.html#2)
* [View 视图](https://www.cnblogs.com/victorwu/p/7418368.html#3)
* [Generic view 通用视图](https://www.cnblogs.com/victorwu/p/7418368.html#4)
* [Viewsets 视图集](https://www.cnblogs.com/victorwu/p/7418368.html#5)
* [Router 路由](https://www.cnblogs.com/victorwu/p/7418368.html#6)
* [Parser 解析器](https://www.cnblogs.com/victorwu/p/7418368.html#7)
* [Renderer 渲染器](https://www.cnblogs.com/victorwu/p/7418368.html#8)
* [Serializer 序列化器](https://www.cnblogs.com/victorwu/p/7418368.html#9)
* [Serializer fields 序列化字段](https://www.cnblogs.com/victorwu/p/7418368.html#10)
* [Serializer relations 序列化字段关系](https://www.cnblogs.com/victorwu/p/7418368.html#11)
* [Validators 验证器](https://www.cnblogs.com/victorwu/p/7418368.html#12)
* [Authentication 认证](https://www.cnblogs.com/victorwu/p/7418368.html#13)
* [Permissions 权限](https://www.cnblogs.com/victorwu/p/7418368.html#14)
* [Throttling 节流](https://www.cnblogs.com/victorwu/p/7418368.html#15)
* [Filtering 过滤器](https://www.cnblogs.com/victorwu/p/7418368.html#16)
* [Pagination 分页](https://www.cnblogs.com/victorwu/p/7418368.html#17)
* [Versioning 版本控制](https://www.cnblogs.com/victorwu/p/7418368.html#18)
* [Content negotiation 内容协商](https://www.cnblogs.com/victorwu/p/7418368.html#19)
* [Metadata 元数据](https://www.cnblogs.com/victorwu/p/7418368.html#20)
* [Schemas 模式](https://www.cnblogs.com/victorwu/p/7418368.html#21)
* [Format suffixes 格式化后缀](https://www.cnblogs.com/victorwu/p/7418368.html#22)
* [Returning URLs 返回URL](https://www.cnblogs.com/victorwu/p/7418368.html#23)
* [Exceptions 异常](https://www.cnblogs.com/victorwu/p/7418368.html#24)
* [Status codes 状态码](https://www.cnblogs.com/victorwu/p/7418368.html#25)
* [Testing 测试](https://www.cnblogs.com/victorwu/p/7418368.html#26)
* [Settings 配置](https://www.cnblogs.com/victorwu/p/7418368.html#27)

**一、请求（Request）**  
REST\_FRAMEWORK 中的 Request 扩展了标准的HttpRequest,为 REST\_FRAMEWORK增加了灵活的request解析和request认证。  
1、请求  
**.data**: 获取请求的主体，相当于request.POST和request.FILES，支持解析除POST之外的HTTP方法的内容，即PUT和PATCH方法。  
**.query\_params**:   request.GET的重命名，但是比标准的request.GET更准确，任何HTTP方法类型可能包括查询参数，而不仅仅是GET请求。  
.parsers:   APIView类或@api\_view装饰将确保这个属性将自动设置为一个解析器实例列表，通常不需要使用。  
2、内容协商  
.accepted\_render: 接受渲染一个对象内容协商  
.accepted\_media\_type: 接受的媒体类型  
3、身份验证  
**.user**:  通常返回django.contrib.auth.models.user的对象。虽然行为取决于所使用的身份验证策略。  
如果请求用户没有被认证，request.user是一个django.contrib.auth.models.Anoymoser对象。  
**.auth**:  被认证的后才会被使用，返回任何额外身份验证的信息，未验证返回None，前端报错HTTP 401 未经授权  
.authenticators:  APIView类或@api\_view装饰将确保这个属性将自动设置为一个认证实例列表  
4、浏览器增强方法  
django rest framework提供一些附加的表单提交方法:PUT,PATCH,DELETE  
**.method**: 返回一个大写的HTTP请求方法字符串，包含了PUT,PATCH,DELETE等。  
.content\_type: 返回一个字符串对象代表HTTP请求的媒体类型的身体,或如果没有媒体类型提供一个空字符串。通常不需要直接访问请求的内容类型，  
因为通常依赖于REST框架的默认请求解析行为，同样可以使用：request.META.get(‘HTTP\_CONTENT\_TYPE’)  
.stream: 返回一个请求主体类容的流，通常不需要访问。  
5、 支持其他标准的HttpResquest属性  
.META/.session/等等

**二、Response（响应）**  
使用APIView类或@api\_view函数返回response对象的views  
1、创建Response  
from rest\_framework.response import Response  
实例化: Response(data, status=None, template\_name=None, headers=None, content\_type=None)

注意：Response类无法处理诸如Django模型实例之类的复杂数据类型，因此需要在创建Response对象之前将数据序列化为原始数据类型，可以使用REST框架的Serializer类来执行数据序列化。

参数说明：  
**data:** 响应serialized(序列化)后的数据  
status: 定义响应状态码，默认200  
template\_name: 如果HTMLRender被选择，返回一个模板  
headers: 是关于HTTP请求头的一个字典  
content\_type: 响应内容类型，一般会根据内容自动协商。

2、其他标准的HttpResponse属性  
response = Response()  
response[‘Cache-Control’] = ‘no-cache’  
.render(): 响应序列化的数据。

**三、基本视图类 (APIView/@api\_view)**  
1、类基础视图（APIView）  
DRF不同于django常规的view类，有如下几点优点：  
\* 提供了更好用的request对象，不同于普通的django HttpRequest更好用。  
\* 封装了Response对象，代替了原有的django HttpResponse，视图将管理内容协商并设置正确的渲染器的响应。  
\* 任何APIException异常将会被捕捉，并做适当的响应。  
\* 传入的请求将身份验证和适当的权限和节流检查将之前运行调度请求处理程序的方法。  
一个小例子：

[复制代码](javascript:void(0);)

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import authentication, permissions

from django.contrib.auth.models import User

class ListUsers(APIView):

"""

展示系统中所有的用户

\* 需要令牌认证。

\* 只有admin用户能够访问这一个视图

"""

authentication\_classes = (authentication.TokenAuthentication,)

permission\_classes = (permissions.IsAdminUser,)

def get(self, request, format=None):

"""

返回一个用户列表

"""

usernames = [user.username for user in User.objects.all()]

return Response(usernames)

[复制代码](javascript:void(0);)

1） API策略属性  
以下的API策略属性应用于APIView，控制视图的策略：  
renderer\_classes： 渲染器类  
parser\_classes： 解释器类  
**authentication\_classes**： 权限类  
throttle\_classes：节流类  
**permission\_classes**： 权限类  
content\_negotiation\_class： 内容协商类  
2）API 策略方法  
以下的策略方法用在API的策略，通常不用重写它：  
get\_renderers(self):  获取渲染器方法  
get\_parsers(self): 获取解释器方法  
get\_authenticators(self): 获取认证方法  
get\_throttles(self): 获取节流方法  
get\_permissions(self):  获取权限方法  
get\_content\_negotiator(self): 获取内容协商方法  
3）API策略实施方法  
下列方法之前被称为调度处理程序方法：  
check\_permissions(self, request): 检查权限  
check\_throttles(self, request): 检查节流  
check\_content\_negotiation(self, request, force=False): 检查内容协商  
4）调度方法  
这些执行任何操作,需要发生之前或之后调用处理程序方法等.  
initial(self, request, \*args, \*\*kwargs): 执行任何操作,需要发生在处理程序方法之前被调用。这个方法是用来执行权限和节流,并执行内容协商。  
handle\_exception(self, exc)：抛出的任何异常处理程序方法将被传递给这个方法,而返回响应实例,或者re-raises异常。  
initialize\_request(self, request, \*args, \*\*kwargs)：确保请求对象传递给处理程序方法是request的一个实例，而不是django的HttpRequest  
finalize\_response(self, request, response, \*args, \*\*kwargs)：确保任何响应处理程序方法返回的对象将被呈现到正确的内容类型  
2、函数基础视图（@api\_view()）  
DRF同样提供了另外一种函数基础视图来装饰django的普通视图，我们同样可以使用request来接受请求和response响应。  
一个小例子：

from rest\_framework.decorators import api\_view

@api\_view

def hello\_world(request):

return Response({“message”:”Hello world!”})

1）api\_view()使用方法  
这个视图将使用默认渲染器、解析器、身份验证设置中指定的类等。通常默认只有GET方法，其他请求方法会报405错误，我们可以手动添加方法为这  
装饰器指定request方法。像这样：

@api\_view([‘GET’, ‘POST’])

def hello\_world(request):

　　if request.method == ‘POST’:

　　　　return Response({“message”:”Got some data”, “data”: request.data})

　　return Response({“messsage”:”Hello world!”})

2）API策略装饰器  
DRF提供了很多附加的装饰器，我们可以添加到@api\_view()后面，例如要加入一个节流的装饰器来确保特定用户每天只能一次通过这个视图，我  
我们就要用到@throttle\_classes装饰器：

[复制代码](javascript:void(0);)

from rest\_framework.decorators import api\_view, throttle\_classes

from rest\_framework.throttling import UserRateThrottle

class OncePerDayUserThrottle(UserRateThrottle):

　　rate = “1/day”

@api\_view([‘GET’])

@throttle\_classes([OncePerDayUserThrottle])

def view(request):

　　return Response({“message”:”Hello for to day! see you tomorrow!”})

[复制代码](javascript:void(0);)

其他可用API的装饰器：  
@renderer\_classes(…)  
@parser\_classes(…)  
@authentication\_classes(…)  
@throttle\_classes(…)  
@permission\_classes(…)

**四、 通用视图（Generic views）**  
基于类视图的主要好处之一是他们允许您编写可重用的行为.  
REST框架提供的通用视图允许您快速构建API观点紧密地映射到您的数据库模型.  
如果通用视图不适合你的需要API,您可以使用常规APIView类,或重用mixin和基类使用的通用视图来组成自己的组可重用通用视图。  
1、通用视图  
同样我们可以设置一些类属性在通用视图内，也可以根据特殊需要重写它的内部视图方法。一个小例子：

[复制代码](javascript:void(0);)

from django.contrib.auth.models import User

from myapp.serializers import UserSerializer

from rest\_framework import generics

from rest\_framework.permissions import IsAdminUser

class UserList(generics.ListCreateAPIView):

　　queryset = User.objects.all()

　　serializer\_class = UserSerializer

　　permission\_classes = (IsAdminUser,)

　　def list(self, request):

　　　　# Note the use of get\_queryset() instead of self.queryset

　　　　queryset = self.get\_queryset()

　　　　serializer = UserSerializer(queryset, many=True)

　　　　return Response(serializer.data)

[复制代码](javascript:void(0);)

在urls配置中，我们可以使用.as\_views()来转换成视图函数，当然也可以配置一些属性

url(r’^/users/’, ListCreateAPIView.as\_view(queryset=User.objects.all(), serializer\_class=UserSerializer), name=’user-list’)

2、GenericAPIView通用视图API参考：  
GenericAPIView继承了DRF的APIView类，为list和detail视图增加了一些一般需求行为方法（提供queryset）。  
**1）属性**  
基本属性：  
**queryset**: 用于返回查询对象集合，也可以使用**get\_queryset()**方法。  
**serializer\_class**: 序列化器类,应该用于输入进行验证和反序列化,并用于序列化输出。通常情况下,你必须设置这个属性,或重写**get\_serializer\_class()**方法。  
lookup\_field: 用于对单个模型实例的对象查找的字段，默认为'pk'。请注意，使用超链接API时，如果需要使用自定义值，则需要确保API视图和序列化器类都设置该查找字段。  
lookup\_url\_kwarg:对象查找的URL关键字参数。URL conf应包含与该值相对应的关键字参数。 如果取消设置，则默认使用与lookup\_field相同的值。

分页属性：  
**pagination\_class**: 用于返回一个分页列表视图的分页类，默认与settings中设置的**DEFAULT\_PAGINATION\_CLASS** 值相同，  
可以通过**’rest\_framework.pagination.PageNumberPagination’**设置分页数  
过滤器属性：  
**filter\_backends**: 过滤queryset的类列表，和在settings中设置**DEFAULT\_FILTER\_BACKENDS** 一样  
**2）方法**  
基本方法：  
**get\_queryset()**: 返回queryset。（详情见官网http://www.django-rest-framework.org/api-guide/generic-views/）  
**get\_object()**:获取某一个具体的model实例对象。  
保存与删除挂钩方法：  
以下方法是mixins类提供，提供简单的对象保存和删除的行为重写：  
**perform\_create(self, serializer)**: CreateModelMixin 当要保存对象时候会被调用  
**perform\_update(self, serializer)**:UpdateModelMixin 当要更新对象时候会被调用  
**perform\_destroy(self, instance)**: DestoryModelMixin 当药删除对象时候会被调用   
其他方法：  
通常不需要覆盖以下方法，但可能会调用。  
get\_serializer\_context(self)：返回序列化程序的上下文的字典。  
get\_serializer(self, instance=None, data=None, many=False, partial=False)：返回一个序列化实例  
get\_paginated\_response(self, data)：返回一个分页样式的response对象  
paginate\_queryset(self, queryset)：分页查询  
filter\_queryset(self, queryset)：不理解

**3、Mixins**  
mixin类提供用于提供基础视图的操作行为。注意,mixin类提供操作方法而不是定义处理程序方法,比如. get()和. post(),直接。这允许更灵活的组合的行为。  
通过rest\_framework.mixins引用。  
**ListModelMixin：提供list方法，列出queryset**  
**CreateModelMixin： 提供create方法，创建和保存一个Model对象**  
**RetrieveModelMixin：提供retrieve方法，检索一个存在的model对象**  
**UpdateModelMixin： 提供Update方法，更改一个模型对象**  
**DestroyModelMixin：提供destroy方法，删除一个模型对象**  
**4）Generic view**  
Generic通用视图类提供具体操作的通用视图类，可以理解为Generic.GenericAPIView和mixin类的合体，通过rest\_framework.generic.调用  
.CreateAPIView：创建一个模型实例，提供post方法的处理器，继承于：GenericAPIView，CreateModelMixin  
.ListAPIView：模型实例的集合，提供get方法处理器，继承于：GenericAPIView，ListModelMixin  
.RetrieveAPIView：一个模型实例，提供get方法处理器，继承于：GenericAPIView，RetrieveModelMixin  
.DestoryAPIView：删除一个模型实例，提供delete方法处理器，继承于：GenericAPIView，DestroyModelMixin  
.UpdateAPIView：修改模型实例，提供put和patch方法处理器，继承于：GenericAPIView，UpdateModelMixi  
.ListCreateAPIView：创建和展示一个模型实例集合，提供get和post处理器，继承于：GenericAPIView，ListModelMixin，CreateModelMixin  
.RetrieveUpdateAPIView：读和改一个模型实例，提供get,put,patch处理器，继承于：GenericAPIView，RetrieveModelMixin，UpdateModelMixin  
.RetrieveDestoryAPIView：读和删除一个模型实例，提供get和delete处理器，继承于：GenericAPIView，RetrieveModelMixin，DestroyModelMixin  
.RetrieveUpdateDestroyAPIView：读、改和删一个模型实例，get, put, patch,delete处理器，继承于：GenericAPIView，RetrieveModelMixin，UpdateModelMixin，DestroyModelMixin

**五、视图集合ViewSets**  
Django REST框架允许将一组相关的逻辑视图在一个类ViewSets中，将重复的逻辑组合成一个类。  
一个小例子：

[复制代码](javascript:void(0);)

from django.contrib.auth.models import User

from django.shortcuts import get\_object\_or\_404

from myapps.serializers import UserSerializer

from rest\_framework import viewsets

from rest\_framework.response import Response

class UserViewSet(viewsets.ViewSet):

"""

A simple ViewSet for listing or retrieving users.

"""

def list(self, request):

queryset = User.objects.all()

serializer = UserSerializer(queryset, many=True)

return Response(serializer.data)

def retrieve(self, request, pk=None):

queryset = User.objects.all()

user = get\_object\_or\_404(queryset, pk=pk)

serializer = UserSerializer(user)

return Response(serializer.data)

[复制代码](javascript:void(0);)

1）关于ViewSets  
如果我们需要，我们需要为两个GET请求分开绑定视图：  
user\_list = UserViewSet.as\_view({“GET”:’list’})  
user\_detail = UserViewSet.as\_view({‘GET’:’retrieve’})

但是通常我们不会这样做，而是通过注册路由来配置url：

from myapp.views import UserViewSet

from rest\_framework.routers import DefaultRouter

router = DefaultRouter()

router.register(r’users’, UserViewSet)

urlpatterns = router.urls

或许你会经常用到自己模型数据，而不是手动写视图集合，这样就要用到模型视图集合ModelViewSet:

class UserViewSet(viewsets.ModelViewSet):

"""

A viewset for viewing and editing user instances.

"""

serializer\_class = UserSerializer

queryset = User.objects.all()

REST框架附带的默认路由器将为标准的create / retrieve / update / destroy样式操作提供路由：

[复制代码](javascript:void(0);)

class UserViewSet(viewsets.ViewSet):

"""

Example empty viewset demonstrating the standard

actions that will be handled by a router class.

If you're using format suffixes, make sure to also include

the `format=None` keyword argument for each action.

"""

def list(self, request):

pass

def create(self, request):

pass

def retrieve(self, request, pk=None):

pass

def update(self, request, pk=None):

pass

def partial\_update(self, request, pk=None):

pass

def destroy(self, request, pk=None):

pass

[复制代码](javascript:void(0);)

如果有需要被路由到的其他方法,可以将它们标记为需要路由使用@detail\_route或@list\_route修饰符。  
@detail\_route装饰器在其URL模式中包含pk，用于单个实例（有pk）的方法，  
@list\_route装饰器适用于在对象列表上操作的方法

[复制代码](javascript:void(0);)

from django.contrib.auth.models import User

from rest\_framework import status

from rest\_framework import viewsets

from rest\_framework.decorators import detail\_route, list\_route

from rest\_framework.response import Response

from myapp.serializers import UserSerializer, PasswordSerializer

class UserViewSet(viewsets.ModelViewSet):

"""

A viewset that provides the standard actions

"""

queryset = User.objects.all()

serializer\_class = UserSerializer

@detail\_route(methods=['post'])

def set\_password(self, request, pk=None):

user = self.get\_object()

serializer = PasswordSerializer(data=request.data)

if serializer.is\_valid():

user.set\_password(serializer.data['password'])

user.save()

return Response({'status': 'password set'})

else:

return Response(serializer.errors,

status=status.HTTP\_400\_BAD\_REQUEST)

@list\_route()

def recent\_users(self, request):

recent\_users = User.objects.all().order('-last\_login')

page = self.paginate\_queryset(recent\_users)

if page is not None:

serializer = self.get\_serializer(page, many=True)

return self.get\_paginated\_response(serializer.data)

serializer = self.get\_serializer(recent\_users, many=True)

return Response(serializer.data)

[复制代码](javascript:void(0);)

2）ViewSets的API参考  
.ViewSet：继承自APIView，可以使用任何标准的属性例如permission\_class,authentication\_classes来控制视图上的API策略。ViewSet同样不提供具体  
行为方法的实现，你可以重写和定义一些请求处理方法。而代替了原来APIVIew的POST，GET等方法，取而代之的是list，create等方法。  
.GenericViewSet：继承自GenericAPIView，提供了默认的get\_queryset()和get\_object()等方法来获取model数据，但不提供任何请求处理方法，为了使用需要混合mixin类，或明确定义操作实现。  
**.ModelViewSet**：继承了GenericAPIView，并通过混入各种mixin类的行为来包含各种操作的实现，请求处理方法有list()、retrieve()、create()、update()、partial\_update()和destroy()。通常需要至少提供queryset和serializer\_class属性，也可以使用get\_queryset()方法动态确定查询集，但也注意从ViewSet中删除queryset属性后，任何关联的路由器将无法自动导出模型的base\_name，因此必须制定base\_name作为路由器注册。  
例子：

[复制代码](javascript:void(0);)

class AccountViewSet(viewsets.ModelViewSet):

"""

A simple ViewSet for viewing and editing the accounts

associated with the user.

"""

serializer\_class = AccountSerializer

permission\_classes = [IsAccountAdminOrReadOnly]

def get\_queryset(self):

return self.request.user.accounts.all()

[复制代码](javascript:void(0);)

.ReadOnlyModelViewSet：继承了GenericAPIView，只提供了只读的请求处理方法list()和retrieve()  
自定义View类：只需要继承GenericAPIView, 增加mixins的相关处理方法。如：

[复制代码](javascript:void(0);)

from rest\_framework import mixins

class CreateListRetrieveViewSet(mixins.CreateModelMixin,

mixins.ListModelMixin,

mixins.RetrieveModelMixin,

viewsets.GenericViewSet):

"""

A viewset that provides `retrieve`, `create`, and `list` actions.

To use it, override the class and set the `.queryset` and

`.serializer\_class` attributes.

"""

pass

[复制代码](javascript:void(0);)

**六、路由(routers)**  
1、REST框架还支持自动Django URL路由,并为您提供了一个简单的、快速的和一致的方式连接视图逻辑的URL。  
如果设置basename将自动生成基于queryset viewset的属性,如果它有一个。注意,如果viewset不包括queryset属性然后注册时必须设置base\_name viewset。

from rest\_framework import routers

router = routers.SimpleRouter()

router.register(r'users', UserViewSet)

router.register(r'accounts', AccountViewSet)

urlpatterns = router.urls

注意：router.register有三个参数属性：

\*prefix：这是url匹配正则表达式。

\*viewset: 指定视图集合

\*base\_name: 基于基础的url名字设置。这个特别要注意，如果不加这个参数，那么视图集合必须要包含queryset这个属性。

2、使用路由routers

router = routers.SimpleRouter()

router.register(r’users’, UserViewSet)

urlpatterns = [

url(r’^api/’, include(router.urls, namespace=’api’)),

]

或者：urlpatterns +=router.urls

3、@detail\_route / @list\_route指定路由  
@detail\_route(methods=[‘post’], permission\_classes=[IsAdminOrIsSelf])  
->指定post方法，权限为IsAdminOrIsSelf

**七、解释器(parse)**  
1、设置解释器  
设置成全局：

REST\_FRAMEWORK ={

‘DEFAULT\_PARSER\_CLASSES’: (‘rest\_framework.parsers.JSONParser’,）

}

也可以在视图APIView中设置：

from rest\_framework.parsers import JSONParser

parser\_class = (JSONParser,)

如果是@api\_view装饰的视图：

@api\_view([‘POST’])

@parser\_classes((JSONParser,))

2、API 参考  
JSONParse: application/json  
FormParse: application/x-www-form-urlencoded  
MultiPartParser:  multipart/form-data  
FileUploadParser: \*/\*  
–>file\_obj = request.data[‘file’]

**八、渲染器(renders)**  
1、设置渲染器  
设置成全局：

REST\_FRAMEWORK ={

‘rest\_framework.renderers.JSONRenderer’,

‘rest\_framework.renderers.BrowsableAPIRenderer’,

}

也可以在视图APIView中设置：

from rest\_framework.renderers import JSONRenderer

renderer\_classes = (JSONRenderer, )

如果是@api\_view装饰的视图：

@api\_view([‘GET’])

@renderer\_classes((JSONRenderer,))

**九、序列化器serializers**  
序列化器可以将python的复杂数据格式渲染成json，xml等格式，同时也提供了反序列化器，可以将渲染成的数据解释成python的数据模型对象。它和django中  
的from表单用法很相似。  
1、渲染序列化对象：  
可以将对象序列化成json,xml格式等

serializer = CommentSerializer(comment)

serializer.data

# {’email’: ‘leila@example.com’, ‘content’: ‘foo bar’, ‘created’: ‘2016-01-27T15:17:10.375877′}

json = JSONRenderer().render(serializer.data) # 渲染成json格式

# b'{“email”:”leila@example.com”,”content”:”foo bar”,”created”:”2016-01-27T15:17:10.375877″}’

2、反序列化

[复制代码](javascript:void(0);)

from django.utils.six import BytesIO

from rest\_framework.parsers import JSONParser

stream = BytesIO(json)

data = JSONParser().parse(stream) # 解析json格式

serializer.is\_valid()

# True

serializer.validated\_data

[复制代码](javascript:void(0);)

3、保存实例  
可以利用**.create**()和**.update()**方法  
调用.save()方法会执行上面两个方法，自动保存或者更新实例对象  
有时候，需要保存其他的信息，可以在参数中添加，如：  
**serializer.save(owner=request.user)**  
4、验证validation  
当需要保存反序列化的数据时，我们往往需要在之前用**is\_valid()**验证数据的合法性，如果不合法可以用  
**serializer.errors**得到该错误信息。这个错误信息可以根据**error\_message**来进行自定义  
5、模型序列化ModelSerializer  
模型序列化与字段序列化用法类似，只需在类的属性Meta中指定相应的模型，就会根据模型自动生成相应的字段  
class AccountSerializer(serializers.ModelSerializer):  
　　class Meta:  
　　　　model = Account  
　　　　fields = (‘id’, ‘account\_name’, ‘users’, ‘created’)  
6、超链接模型序列化HyperlinkedModelSerializer  
与模型序列化类似，只不过给了一个可链接的形式。

**十、序列化字段serializer fields**  
1、核心参数  
\*read\_only: 只读形式，只能序列化到api输出，不能被反序列化数据更新修改操作。默认False  
\*wirte\_only: 只写形式，只能被反序列化数据更新和创建，不能序列化表示。默认False  
\***required**: 必须字段，通常出现在反序列化的过程中，该字段必须有值。默认True  
\*allow\_null：是否允许为空，默认False  
\***default**: 默认值，  
\***source**：字段来源，为字段指定一个字段来源，如(source=’user.email’).  
\*validators: 合法性函数列表，支持django的ValidationError包。  
\*error\_message: 一个错误信息的字典，包含null, blank, invalid, invalid\_choice, unique等键值。  
\*label:一篇简短的文本字符串,可以用作在HTML表单字段的名称或其他描述性的元素  
\*help\_text: 一个文本字符串,可以用作描述字段的HTML表单字段或其他描述性的元素。  
\*initial: 用于预先填充HTML表单字段的值。  
\*style: 字典的键-值对,可以用来控制渲染器应该如何呈现。  
2、字段类型  
\*布尔字段**BooleanField**: BoolenField(defalut=True)  
\*字符字段**CharField**: CharField(max\_length=None, min\_length=None, allow\_blank=False, trim\_whitespace=True)  
\*邮箱字段**EmailField**: EmailField(max\_length=None, min\_length=None, allow\_blank=False)  
\*正则字段**RegexField**: RegexField(regex, max\_length=None, min\_length=None, allow\_blank=False)  
\*新闻标题字段**SlugField**：SlugField(max\_length=50, min\_length=None, allow\_blank=False)  
\*URL地址字段 **UrlField**: URLField(max\_length=200, min\_length=None, allow\_blank=False)  
\*UUID字段 **UUIDField**: UUIDField(format=’hex\_verbose’)   —“de305d54-75b4-431b-adb2-eb6b9e546013″  
\*文件路径字段**FilePathField**：FilePathField(path, match=None, recursive=False, allow\_files=True, allow\_folders=False, required=None, \*\*kwargs)  
\*IP地址字段**IPAddressField**：IPAddressField(protocol=’both’, unpack\_ipv4=False, \*\*options)  
\*数字型字段：**IntegerField、FloatField、DecimalField**  
\*时间和日期字段：**DateTimeField**：DateTimeField(format=None, input\_formats=None)  
**DateField**：  
**TimeField**：  
**DurationField**：  
\*选择字段：**ChoiceField**  
\*文件字段：**FileField**：FileField(max\_length=None, allow\_empty\_file=False, use\_url=UPLOADED\_FILES\_USE\_URL)  
\*图片字段：**ImageField**(max\_length=None, allow\_empty\_file=False, use\_url=UPLOADED\_FILES\_USE\_URL)  
\*模型字段：**ModelField**：ModelField(model\_field=<Django ModelField instance>)  
\*只读字段：**ReadOnlyField**：

**十一、字段关系serializer relations**  
多对一：**ForeignKey**  
多对多: **ManyToManyField**  
一对一：**OneToOneField**  
1、api参考  
tracks = serializers.StringRelatedField(many=True)  # 字符串形式关系字段  
tracks = serializers.PrimaryKeyRelatedField(many=True, read\_only=True)# 主键形式关系字段  
tracks = serializers.HyperlinkedRelatedField(many=True, read\_only=True, view\_name=’track-detail’) # 超链接形式关系字段  
tracks = serializers.SlugRelatedField(many=True, read\_only=True, slug\_field=’title’)  # 以小标题形式关系  
track\_listing = serializers.HyperlinkedIdentityField(view\_name=’track-list’)  # 超链接身份关系  
2、嵌套关系  
序列化类中可以直接引用其他序列化化类

**十二、合法性验证(Validation)**  
1、数据的唯一性**UniqueValidator**  
在模型中，只需设置字段的**unique=True**就会在该字段自动生成改唯一性判断  
例:slug = SlugField(max\_length=100, validators=[UniqueValidator(queryset=BlogPost.objects.all(), messsage=”)])  
\*queryset：模型数据对象集合  
\*message: 异常提示信息  
2、数据字段的不重复性UniqueTogetherValidator

[复制代码](javascript:void(0);)

class Meta:

validators = [

UniqueTogetherValidator(

queryset=ToDoItem.objects.all(),

fields=(‘list’, ‘position’)

)

]

\*queryset：模型数据对象集合

\*fields: 字段的集合元祖

\*message: 异常提示信息

[复制代码](javascript:void(0);)

**十三、认证Authentication**  
1、设置认证方案  
\*可以在settings文件中配置DEFAULT\_AUTHENTICATION\_CLASSES：

REST\_FRAMEWORK = {

‘DEFAULT\_AUTHENTICATION\_CLASSES’: (

‘rest\_framework.authentication.BasicAuthentication’,

‘rest\_framework.authentication.SessionAuthentication’,

)

}

\*也可以在视图中设置  
(APIView)：

authentication\_classes = (SessionAuthentication, BasicAuthentication)

permission\_classes = (IsAuthenticated,)

(@api\_view)

@authentication\_classes((SessionAuthentication, BasicAuthentication))

@permission\_classes((IsAuthenticated,))

**十四、权限**  
1、设置权限方案  
\*可以在settings文件中配置DEFAULT\_PERMISSION\_CLASSES

REST\_FRAMEWORK = {

‘DEFAULT\_PERMISSION\_CLASSES’: (

‘rest\_framework.permissions.IsAuthenticated’,

# ‘rest\_framework.permissions.AllowAny’,

)

}

\*视图中设置：  
（APIView）

permission\_classes = (IsAuthenticated,)

（@api\_view）

@permission\_classes((IsAuthenticated, ))

2、API引用  
\*AllowAny：允许任何  
\***IsAuthenticated**：是否被认证  
\***IsAdminUser**：是管理用户  
\***IsAuthenticatedOrReadOnly**：是被认证或者只读  
\***DjangoModelPermissions**：标准的django模型设置权限  
\***DjangoModelPermissionsOrAnonReadOnly**：标准的django模型处理或者只读权限  
\***DjangoObjectPermissions**：相比模型权限，只提供queryset和get\_queryset方法。  
**\*自定义权限：**  
**继承BasePermission;**  
重写：.**has\_permission**(self, request, view)  
.**has\_object\_permission**(self, request, view, obj)

**十五、节流Trottling**  
限制请求量  
1、设置节流方案  
\*可以在settings文件中配置DEFAULT\_THROTTLE\_CLASSES 和 DEFAULT\_THROTTLE\_RATES

[复制代码](javascript:void(0);)

‘DEFAULT\_THROTTLE\_CLASSES’: (

‘rest\_framework.throttling.AnonRateThrottle’,

‘rest\_framework.throttling.UserRateThrottle’

),

‘DEFAULT\_THROTTLE\_RATES’: {

‘anon’: ‘100/day’,

‘user’: ‘1000/day’

}

[复制代码](javascript:void(0);)

\*可以在视图中设置：  
（APIView）

throttle\_classes = (UserRateThrottle,)

（@api\_view）

@throttle\_classes([UserRateThrottle])

**十六、过滤器Filtering**  
通常drf的默认的通用list视图类要处理的是全部模型的查询集合集，如果我们要指定结果集，  
这就必须用到过滤器了。通常的方法是重写get\_queryset(self)方法。  
例如：

[复制代码](javascript:void(0);)

class PurchaseList(generics.ListAPIView):

　　serializer\_class = PurchaseSerializer

　　# 根据用户过滤

　　def get\_queryset(self):

　　　　user = self.request.user

　　　　return Purchase.objects.filter(purchase = user)

　　# 根据url请求过滤

　　#def get\_queryset(self):

　　#　　username = self.kwargs[‘username’]

　　#　　return Purchase.objects.filter(purchaser\_\_username=username)

　　# 根据请求参数过滤同上

　　#def get\_queryset(self):

　　#　　queryset = Purchase.objects.all()

　　#　　username = self.request.query\_params.get(‘username’, None)

　　#　　if username is not None:

　　#　　　　queryset = queryset.filter(purchaser\_\_username=username)

　　#　　return queryset

[复制代码](javascript:void(0);)

1、generic filter过滤  
\*通过settings文件中配置过滤

REST\_FRAMEWORK = {

‘DEFAULT\_FILTER\_BACKENDS’: (‘rest\_framework.filters.DjangoFilterBackend’,)

}

\*也可通过视图中过滤  
filter\_backends = (filters.DjangoFilterBackend,)

2、API向导

* **DjangoFilterBackend**

django-filter库包括一个**DjangoFilterBackend**类，它支持REST框架的高度可定制的字段过滤。  
首先安装django-filter， 然后将django\_filters添加到Django的INSTALLED\_APPS。  
pip install django-filter  
在settings.py加上如下配置：

REST\_FRAMEWORK = {

'DEFAULT\_FILTER\_BACKENDS': ('django\_filters.rest\_framework.DjangoFilterBackend',)

}

或者将过滤器加到单个View或ViewSet中：

from django\_filters.rest\_framework import DjangoFilterBackend

class UserListView(generics.ListAPIView):

...

filter\_backends = (DjangoFilterBackend,)

**如果要允许对某些字段进行过滤，可以使用filter\_fields属性。**

class ProductList(generics.ListAPIView):

queryset = Product.objects.all()

serializer\_class = ProductSerializer

filter\_backends = (DjangoFilterBackend,)

filter\_fields = ('category', 'in\_stock')

对应的列表过滤请求：http://example.com/api/products?category=clothing&in\_stock=True

* **SearchFilter**

**如果要明确指定可以对哪些字段进行搜索，可以使用search\_fields属性，**默认为可以对serializer\_class属性指定的串行器上的任何可读字段进行搜索**：**

class UserListView(generics.ListAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

filter\_backends = (filters.SearchFilter,)

search\_fields = ('username', 'email')

对应的列表搜索请求：http://example.com/api/users?search=russell  
也可以使用双下划线在Foreign Key或ManyToManyField上执行相关查找：

search\_fields = ('username', 'email', 'profile\_\_profession')

默认情况下，搜索将使用不区分大小写的部分匹配。 搜索参数可以包含多个搜索项，它们应该是空格和/或逗号分隔。 如果使用多个搜索项，则仅当所有提供的条款匹配时，才会在列表中返回对象。默认情况下，搜索参数被命名为“search”，但这可能会被SEARCH\_PARAM设置覆盖。  
The search behavior may be restricted by prepending various characters to the search\_fields.  
可以通过在search\_fields中加入一些字符来限制搜索行为，如下：

* **'^'**  ：以xx字符串开始搜索
* **'='**：完全匹配
* **'@'**：全文搜索（目前只支持Django的MySQL后端）
* **'$'**：正则表达式搜索

例如：

search\_fields = ('=username', '=email')

* **OrderingFilter**

**OrderingFilter类支持简单的查询参数控制结果排序。**  
默认情况下，查询参数被命名为“ordering”，但这可能会被ORDERING\_PARAM设置覆盖。  
可以使用**ordering\_fields**属性**明确指定**可以对哪些字段执行排序，这有助于防止意外的数据泄露，例如允许用户对密码散列字段或其他敏感数据进行排序。  
如果不指定**ordering\_fields**属性，则默认为可以对serializer\_class属性指定的串行器上的任何可读字段进行过滤，同：ordering\_fields = '\_\_all\_\_'

class UserListView(generics.ListAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

filter\_backends = (filters.OrderingFilter,)

ordering\_fields = ('username', 'email')

使用**ordering**属性设置默认排序：

class UserListView(generics.ListAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

filter\_backends = (filters.OrderingFilter,)

ordering\_fields = ('username', 'email')

ordering = ('username',)

3、**自定义过滤类，再用filter\_class指定过滤集合类**

[复制代码](javascript:void(0);)

import django\_filters

from myapp.models import Product

from myapp.serializers impoert ProductSerializer

from rest\_framework import filters

from rest\_framework import generics

class ProductFilter(filters.FilterSet):

　　min\_price = django\_filters.NumberFilter(name=’price’, lookup\_type=”gte”)

　　max\_price = django\_filters.NumberFilter(name=’price’, lookup\_type=”lte”)

　　class Meta:

　　　　model = Product

　　　　fields = [‘category’, ‘in\_stock’, ‘min\_price’, ‘max\_price’]

class ProductList(generics.ListAPIView):

　　queryset = Product.object.all()

　　serializer\_class = ProductSerializer

　　filter\_backends = (filter.DjangoFilterBackend,)

　　filter\_class = ProductFilter

[复制代码](javascript:void(0);)

请求示例： http://example.com/api/products?category=clothing&max\_price=10.00

**十七、分页Pagination**  
1、设置自带的分页  
\*可以在settings使用**DEFAULT\_PAGINATION\_CLASS**和**PAGE\_SIZE**设置键全局默认分页样式。

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.LimitOffsetPagination',

'PAGE\_SIZE': 100

}

2、修改分页样式，如果要修改分页样式的特定方面，则需要覆盖其中一个分页类，并设置要更改的属性。

[复制代码](javascript:void(0);)

class LargeResultsSetPagination(PageNumberPagination):

　　page\_size = 1000

　　page\_size\_query\_param = ‘page\_size’

　　max\_page\_size = 10000

class StandardResultsSetPagination(PageNumberPagination):

　　page\_size = 100

　　page\_size\_query\_param = ‘page\_size’

　　max\_page\_size = 1000

[复制代码](javascript:void(0);)

应用到视图中

class BillingRecordsView(generics.ListAPIView):

　　queryset = Billing.objects.all()

　　serializer = BillingRecordsSerializer

　　pagination\_class = LargeResultsSetPagination

或者设置为全局分页样式：

class BillingRecordsView(generics.ListAPIView):

queryset = Billing.objects.all()

serializer\_class = BillingRecordsSerializer

pagination\_class = LargeResultsSetPagination

3、API参考

**PageNumberPagination**：此分页样式在请求查询参数中接受单个编号页码。

　　请求示例：GET https://api.example.org/accounts/?page=4

　　全局设置：

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 100

}

　　PageNumberPagination类包括可以覆盖以修改分页样式的许多属性，要设置这些属性，应覆盖PageNumberPagination类，然后如上所示启用自定义分页类。

* django\_paginator\_class：使用的Django Paginator类，默认是django.core.paginator.Paginator，对大部分用例是适用的。
* **page\_size**：数值，页面大小，默认是全局PAGE\_SIZE的值。
* **page\_query\_param**：字符串，查询参数的名称，默认是'page'
* **page\_size\_query\_param**：字符串，请求设置页面大小的参数名称，默认是None，表示客户端可能无法控制请求的页面大小。
* **max\_page\_size**：字符串，最大允许请求的页面大小， 此属性仅在page\_size\_query\_param也被设置时有效。
* last\_page\_strings：字符串列表或者元组，默认是('last',)
* template：分页控件使用的模板的名称，可以覆盖或设置为None，默认为"rest\_framework/pagination/numbers.html"

**LimitOffsetPagination**：查找多个数据库记录时使用的语法，客户端包括“limit”和“offset”查询参数，limit指示要返回的最大项目数，并且相当于其他样式中的page\_size，offset表示查询起始位置。

　　请求示例：GET https://api.example.org/accounts/?limit=100&offset=400

　　全局设置：

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.LimitOffsetPagination'

}

**十八、 版本控制（Versioning）**

启用API版本控制时，**request.version**属性将包含与传入客户端请求中请求的版本相对应的字符串。  
默认情况下，版本控制未启用，request.version将始终返回None。  
1、基于版本的不同，改变API的行为，常见的一种用法是在较新版本中切换到其他序列化样式。 例如：

def get\_serializer\_class(self):

if self.request.version == 'v1':

return AccountSerializerVersion1

return AccountSerializer

2、REST框架包含的reverse（反向）功能与版本控制方案有关，参数加上request：

from rest\_framework.reverse import reverse

reverse('bookings-list', request=request)

上述功能用于请求版本的任何URL转换，如果使用的是NamespacedVersioning，API版本是“v1”，则使用的URL查找将是“v1：bookings-list”，这可能会解析为http://example.org/v1/bookings/这样的URL，如果使用QueryParameterVersioning，API版本为1.0，则返回的URL类似于http://example.org/bookings/?version=1.0。3、当使用超链接序列化样式和基于URL的版本控制方案时，请确保将request作为context包含在序列化实例化中，这样做将允许返回的URL都包含适当的版本。

def get(self, request):

queryset = Booking.objects.all()

serializer = BookingsSerializer(queryset, many=True, context={'request': request})

return Response({'all\_bookings': serializer.data})

4、settings配置

REST\_FRAMEWORK = {

'DEFAULT\_VERSIONING\_CLASS': 'rest\_framework.versioning.NamespaceVersioning'

}

除非明确设置，DEFAULT\_VERSIONING\_CLASS的值将为“None”。 在这种情况下，request.version属性将始终返回None。  
也可以在单个视图中设置版本控制方案， 但通常不需要这样做，因为在全局范围内使用单个版本控制方案更有意义， 如果确实需要这样做，请使用versioning\_class属性。

class ProfileList(APIView):

versioning\_class = versioning.QueryParameterVersioning

5、其他配置

* DEFAULT\_VERSION
* ALLOWED\_VERSIONS
* VERSION\_PARAM

6、API参考

* AcceptHeaderVersioning
* URLPathVersioning
* HostNameVersioning
* QueryParameterVersioning

7、自定义版本控制模式

要实现一个自定义版本控制方案，子类继承BaseVersioning并覆盖.determine\_version方法。

class XAPIVersionScheme(versioning.BaseVersioning):

def determine\_version(self, request, \*args, \*\*kwargs):

return request.META.get('HTTP\_X\_API\_VERSION', None)

**十九、 内容协商（Content negotiation）**

Content negotiation是根据客户端或服务器的首选项 从多个可能的表示选择一个 返回给客户端的过程。

**二十、 元数据（Metadata）**

REST框架包括一个可配置的机制，用于确定API应如何响应OPTIONS请求，允许返回API模式或其他资源信息。目前还没有任何广泛采用的惯例来确定HTTP OPTIONS请求应该返回什么样的响应，因此我们提供一种特殊的样式，返回一些有用的信息。  
这是一个示例响应，演示了默认情况下返回的信息。

[复制代码](javascript:void(0);)

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

Content-Type: application/json

{

"name": "To Do List",

"description": "List existing 'To Do' items, or create a new item.",

"renders": [

"application/json",

"text/html"

],

"parses": [

"application/json",

"application/x-www-form-urlencoded",

"multipart/form-data"

],

"actions": {

"POST": {

"note": {

"type": "string",

"required": false,

"read\_only": false,

"label": "title",

"max\_length": 100

}

}

}

}

[复制代码](javascript:void(0);)

全局配置：

REST\_FRAMEWORK = {

'DEFAULT\_METADATA\_CLASS': 'rest\_framework.metadata.SimpleMetadata'

}

**二十一、 架构（Schemas）**

API模式是一个非常有用的工具，它允许一系列用例，包括生成参考文档，或者驱动与API进行交互的动态客户端库。

1、内部模式表示REST框架使用Core API用于以格式无关的表示来模拟模式信息。 该信息可以呈现为各种不同的模式格式，或用于生成API文档。  
当使用Core API时，模式为文档，有关API的信息的顶级容器对象。 可用的API交互通过链接对象来表示，每个链接都包含一个URL，HTTP方法，并且可以包括一个Field实例的列表，它描述API端点可以接受的任何参数。 链接和字段实例还可以包括允许将API模式呈现为用户文档的描述。示例**：**

[复制代码](javascript:void(0);)

coreapi.Document(

title='Flight Search API',

url='https://api.example.org/',

content={

'search': coreapi.Link(

url='/search/',

action='get',

fields=[

coreapi.Field(

name='from',

required=True,

location='query',

description='City name or airport code.'

),

coreapi.Field(

name='to',

required=True,

location='query',

description='City name or airport code.'

),

coreapi.Field(

name='date',

required=True,

location='query',

description='Flight date in "YYYY-MM-DD" format.'

)

],

description='Return flight availability and prices.'

)

}

)

[复制代码](javascript:void(0);)

2、模式输出格式

为了在HTTP响应中呈现，内部表示必须是呈现为响应中使用的实际字节。  
Core JSON被设计为与Core API一起使用的规范格式。 REST框架包括一个用于处理这种媒体类型的渲染器类，即renderers.CoreJSONRenderer。  
其他模式格式，如Open API（“Swagger”）、JSON HyperSchema或API Blueprint，也可以通过实现自定义渲染器类来支持。

3、添加模式

安装：pip install coreapiREST框架包括自动生成模式的功能，也允许明确指定一个模式，有多个方法用于在API来添加一个模式  
**get\_schema\_view()：**最简单的方法

[复制代码](javascript:void(0);)

from rest\_framework.schemas import get\_schema\_view

schema\_url\_patterns = [

url(r'^api/', include('myproject.api.urls')),

]

schema\_view = get\_schema\_view(

title='Server Monitoring API',

url='https://www.example.org/api/',

urlconf='myproject.urls',

renderer\_classes=[CoreJSONRenderer, APIBlueprintRenderer],

patterns=schema\_url\_patterns,

)

urlpatterns = [ url('^$', schema\_view), ... ]

[复制代码](javascript:void(0);)

参数说明：

* **title**：模式的描述性标题
* url：模式的URL地址
* urlconf ：URL配置文件路径，默认为Django的 ROOT\_URLCONF
* renderer\_classes ：渲染器类
* patterns ：
* generator\_class ：

**二十二、 格式化后缀（Format suffixes）**

**二十三、 Returning URLs**

通常情况下，最好从Web API返回绝对URI，例如http://example.com/foobar，而不是返回相对的URI，如/ foobar。这样做的好处是：  
     它更明确  
     它为您的API客户端减少了工作。  
     当在不具有本机URI类型的表示形式（如JSON）中找到字符串的含义时，这种含义不会有歧义。  
     它使得超链接进行标记HTML表示等操作变得容易。REST框架提供了两个实用功能，可以更简单地从Web API返回绝对URI。  
**1、reverse(viewname, \*args, \*\*kwargs)**同django.urls.reverse，返回完全限定的URL，使用request确定主机和端口。  
需将request作为关键字参数包含在函数中：

[复制代码](javascript:void(0);)

from rest\_framework.reverse import reverse

from rest\_framework.views import APIView

from django.utils.timezone import now

class APIRootView(APIView):

def get(self, request):

year = now().year

data = {

...

'year-summary-url': reverse('year-summary', args=[year], request=request)

}

return Response(data)

[复制代码](javascript:void(0);)

**2、reverse\_lazy(viewname, \*args, \*\*kwargs)**

同django.urls.reverse\_lazy，返回一个完全限定的URL，使用request确定主机和端口。用法同reverse

**二十四、 异常（Exceptions）**

1、REST框架视图处理各种异常，并处理返回适当的错误响应。处理的异常有：

* REST框架内引发的APIException子类
* Django的Http404异常
* Django的PermissionDenied异常

2、自定义异常处理  
自定义异常处理函数必须使用一对参数，第一个是要处理的异常，第二个是包含任何额外的上下文的字典，例如当前正在处理的视图。异常处理函数应该返回一个Response对象，如果异常不能被处理，返回None。 如果处理程序返回None，则异常将被重新引发，Django将返回标准的HTTP 500'服务器错误'响应。  
例如：

[复制代码](javascript:void(0);)

from rest\_framework.views import exception\_handler

def custom\_exception\_handler(exc, context):

# Call REST framework's default exception handler first,

# to get the standard error response.

response = exception\_handler(exc, context)

# Now add the HTTP status code to the response.

if response is not None:

response.data['status\_code'] = response.status\_code

return response

[复制代码](javascript:void(0);)

必须在settings配置EXCEPTION\_HANDLER：

REST\_FRAMEWORK = {

'EXCEPTION\_HANDLER': 'my\_project.my\_app.utils.custom\_exception\_handler'

}

如果未指定，则默认为REST框架提供的标准异常处理程序：

REST\_FRAMEWORK = {

'EXCEPTION\_HANDLER': 'rest\_framework.views.exception\_handler'

}

请注意，异常处理程序将只会由引发异常生成的响应进行调用。 它不会用于视图直接返回的任何响应，例如当序列化器验证失败时通用视图返回的HTTP\_400\_BAD\_REQUEST响应。

3、API参考

* APIException()
* ParseError(detail=None, code=None)
* AuthenticationFailed(detail=None, code=None)
* NotAuthenticated(detail=None, code=None)
* PermissionDenied(detail=None, code=None)
* NotFound(detail=None, code=None)
* MethodNotAllowed(method, detail=None, code=None)
* NotAcceptable(detail=None, code=None)
* UnsupportedMediaType(media\_type, detail=None, code=None)
* Throttled(wait=None, detail=None, code=None)
* ValidationError(detail, code=None)

**二十五、 状态码（Status Code）**

不建议在响应中使用裸状态代码，REST框架包括一组命名常量，可用于使更多的代码更加明显和可读。

例如：

from rest\_framework import status

from rest\_framework.response import Response

def empty\_view(self):

content = {'please move along': 'nothing to see here'}

return Response(content, status=status.HTTP\_404\_NOT\_FOUND)

所有状态码分类如下：

**信息--1XX**

此类状态码表示临时响应。 默认情况下，REST框架中没有使用1xx状态代码。

HTTP\_100\_CONTINUE

HTTP\_101\_SWITCHING\_PROTOCOLS

**成功--2XX**

此类状态码表示客户端的请求已成功接收，理解并被接受。

[复制代码](javascript:void(0);)

HTTP\_200\_OK

HTTP\_201\_CREATED

HTTP\_202\_ACCEPTED

HTTP\_203\_NON\_AUTHORITATIVE\_INFORMATION

HTTP\_204\_NO\_CONTENT

HTTP\_205\_RESET\_CONTENT

HTTP\_206\_PARTIAL\_CONTENT

HTTP\_207\_MULTI\_STATUS

[复制代码](javascript:void(0);)

**重定向--3XX**

此类状态码表示用户代理需要采取进一步的操作才能完成请求

[复制代码](javascript:void(0);)

HTTP\_300\_MULTIPLE\_CHOICES

HTTP\_301\_MOVED\_PERMANENTLY

HTTP\_302\_FOUND

HTTP\_303\_SEE\_OTHER

HTTP\_304\_NOT\_MODIFIED

HTTP\_305\_USE\_PROXY

HTTP\_306\_RESERVED

HTTP\_307\_TEMPORARY\_REDIRECT

[复制代码](javascript:void(0);)

**客户端错误--4XX**

4xx类的状态码适用于客户端似乎有错误的情况。 除了响应HEAD请求之外，服务器应该包含一个包含错误情况说明的实体，以及它是一个临时的还是永久的。

[复制代码](javascript:void(0);)

HTTP\_400\_BAD\_REQUEST

HTTP\_401\_UNAUTHORIZED

HTTP\_402\_PAYMENT\_REQUIRED

HTTP\_403\_FORBIDDEN

HTTP\_404\_NOT\_FOUND

HTTP\_405\_METHOD\_NOT\_ALLOWED

HTTP\_406\_NOT\_ACCEPTABLE

HTTP\_407\_PROXY\_AUTHENTICATION\_REQUIRED

HTTP\_408\_REQUEST\_TIMEOUT

HTTP\_409\_CONFLICT

HTTP\_410\_GONE

HTTP\_411\_LENGTH\_REQUIRED

HTTP\_412\_PRECONDITION\_FAILED

HTTP\_413\_REQUEST\_ENTITY\_TOO\_LARGE

HTTP\_414\_REQUEST\_URI\_TOO\_LONG

HTTP\_415\_UNSUPPORTED\_MEDIA\_TYPE

HTTP\_416\_REQUESTED\_RANGE\_NOT\_SATISFIABLE

HTTP\_417\_EXPECTATION\_FAILED

HTTP\_422\_UNPROCESSABLE\_ENTITY

HTTP\_423\_LOCKED

HTTP\_424\_FAILED\_DEPENDENCY

HTTP\_428\_PRECONDITION\_REQUIRED

HTTP\_429\_TOO\_MANY\_REQUESTS

HTTP\_431\_REQUEST\_HEADER\_FIELDS\_TOO\_LARGE

HTTP\_451\_UNAVAILABLE\_FOR\_LEGAL\_REASONS

[复制代码](javascript:void(0);)

**服务器错误--5XX**

以数字“5”开头的响应状态代码表示服务器知道它已经发生错误或不能执行请求的情况。 除了响应HEAD请求之外，服务器应该包含一个包含错误情况说明的实体，以及它是一个临时的还是永久的。

[复制代码](javascript:void(0);)

HTTP\_500\_INTERNAL\_SERVER\_ERROR

HTTP\_501\_NOT\_IMPLEMENTED

HTTP\_502\_BAD\_GATEWAY

HTTP\_503\_SERVICE\_UNAVAILABLE

HTTP\_504\_GATEWAY\_TIMEOUT

HTTP\_505\_HTTP\_VERSION\_NOT\_SUPPORTED

HTTP\_507\_INSUFFICIENT\_STORAGE

HTTP\_511\_NETWORK\_AUTHENTICATION\_REQUIRED

[复制代码](javascript:void(0);)

**以下帮助函数可用于识别响应代码的类别。**

is\_informational() # 1xx

is\_success() # 2xx

is\_redirect() # 3xx

is\_client\_error() # 4xx

is\_server\_error() # 5xx

**二十六、 测试（Testing）**

**二十七、 配置（settings）**

REST框架的配置在Django settings中命名为REST\_FRAMEWORK。  
可以做的配置如下：  
1、API策略配置  
以下设置控制基本API策略，并应用于基于APIView类的每个视图或基于@api\_view功能的视图。

* DEFAULT\_RENDERER\_CLASSES
* DEFAULT\_PARSER\_CLASSES
* DEFAULT\_AUTHENTICATION\_CLASSES
* DEFAULT\_PERMISSION\_CLASSES
* DEFAULT\_THROTTLE\_CLASSES
* DEFAULT\_CONTENT\_NEGOTIATION\_CLASS

2、通用视图配置

以下设置控制基于类的通用视图的行为。

* DEFAULT\_FILTER\_BACKENDS
* PAGE\_SIZE
* SEARCH\_PARAM
* ORDERING\_PARAM

3、版本控制配置

* DEFAULT\_VERSION
* ALLOWED\_VERSIONS
* VERSION\_PARAM

4、认证配置

以下设置控制未经身份验证的请求的行为。

* UNAUTHENTICATED\_USER
* UNAUTHENTICATED\_TOKEN

5、测试配置

以下设置控制APIRequestFactory和APIClient的行为

* TEST\_REQUEST\_DEFAULT\_FORMAT
* TEST\_REQUEST\_RENDERER\_CLASSES

6、模式生成配置

* SCHEMA\_COERCE\_PATH\_PK
* SCHEMA\_COERCE\_METHOD\_NAMES

7、Content type配置

* URL\_FORMAT\_OVERRIDE
* FORMAT\_SUFFIX\_KWARG

8、日期和时间格式配置

* DATETIME\_FORMAT
* DATETIME\_INPUT\_FORMATS
* DATE\_FORMAT
* DATE\_INPUT\_FORMATS
* TIME\_FORMAT
* TIME\_INPUT\_FORMATS

9、编码配置

* UNICODE\_JSON
* COMPACT\_JSON
* COERCE\_DECIMAL\_TO\_STRING