源代码位于：request.py

对于DRF的request对象，有下面的主要属性：

- .data
- .query_params
- .parsers
- .accepted_renderer
- .accepted_media_type
- .user
- .auth
- .authenticators
- .method
- .content_type
- .stream

> 如果你正在做基于REST的Web服务...你最好忽略request.POST。
>
> — Malcom Tredinnick, Django **开发组成员**

REST framework的 `Request` 类扩展了Django标准的 `HttpRequest`，添加了对REST framework请求解析和身份验证的支持。

源代码片段：

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

    Kwargs:
        - request(HttpRequest). The original request instance.
        - parsers_classes(list/tuple). The parsers to use for parsing the
          request content.
        - authentication_classes(list/tuple). The authentications used to
try
          authenticating the request's  user.
    """

      def_init_(self, request, parsers=None, authenticators=None,
                  negotiator=None, parser_context=None):
```

```
15          assert isinstance(request, HttpRequest), (
16              'The `request` argument must be an instance of '
17              '`django.http.HttpRequest`, not `{}.{}`.'
18              .format(request.__class__.__module__,
    request.__class__.__name__)
19          )
20
21          self._request = request
22          self.parsers = parsers or  ()
23          self.authenticators = authenticators or ()
24          self.negotiator = negotiator or self._default_negotiator()
25          self.parser_context = parser_context
26          self._data = Empty
27          self._files = Empty
28          self._full_data = Empty
29          self._content_type =  Empty
30          self. stream = Empty
```

请求解析相关

REST framework的request请求对象以通常处理表单数据相同的方式使用JSON数据或其他媒体类型处理请求。下面的属性是request中数据主体的部分。

.data

`request.data` 返回请求正文的解析内容，这与标准的 `request.POST` 和 `request.FILES` 属性类似，除了下面的区别：

- `request.data` 包含所有解析的内容，包括 文件或非文件输入。
- `request.data` 支持除 `POST` 之外的HTTP方法，这意味着你可以访问 `PUT` 和 `PATCH` 请求的内容。
- `request.data` 支持更灵活的请求解析，而不仅仅是表单数据。 例如，你可以与处理表单数据相同的方式处理传入的JSON数据。

```
1       @property
2       def data(self):
3           if not _hasattr(self, '_full_data'):
4               self._load_data_and_files()
5           return self. full data
```

.query_params

`request.query_params` 是 `request.GET` 的一个更准确的同义词，也就是在DRF中的替代品。

为了让你的代码清晰明了，我们建议使用 `request.query_params` 而不是Django标准的 `request.GET` 。这样做有助于保持代码库更加正确和明了。通俗地说，就是不要把DRF的属性/方法名和Django原生的属性/方法名混用。

```
1       def query_params(self):
2           """
3           More semantically correct name for request.GET.
4           """
5           return self._request.GET
```

.parsers

`APIView` 类或 `@api_view` 装饰器将根据view中设置的 `parser_classes` 值或 `DEFAULT_PARSER_CLASSES` 配置参数进行设置，确保此属性自动设置为 `Parser` 实例列表。通常并不需要访问这个属性。

注意：如果客户端发送格式错误的内容，则访问`request.data`可能会引发`ParseError`。默认情况下，REST框架的`APIView`类或`@api_view`装饰器将捕获错误并返回`400 Bad Request`响应。

如果客户端发送的请求的内容类型无法解析，则将`UnsupportedMediaType`引发异常，默认情况下将捕获该异常并返回`415 Unsupported Media Type`响应。

内容协商相关

request请求对象还提供了一些属性允许你确定内容协商阶段的结果。这允许你实现具体的行为，例如为不同的媒体类型选择不同的序列化方案。

.accepted_renderer

内容协商阶段选择的renderer实例。

.accepted_media_type

内容协商阶段接受的媒体类型的字符串。

认证相关

requset对象提供了灵活的，每次请求都进行验证的认证功能，并支持下面的特性：

- 对API的不同部分使用不同的身份验证策略。
- 支持同时使用多种身份验证策略
- 提供与传入请求相关联的用户和令牌信息

.user

`request.user` 默认情况下使用的是Django自带的auth框架的用户模型，返回一个 `django.contrib.auth.models.User` 实例。但该行为取决于你所使用的的认证策略，很显然当你使用第三方认证模块时，就不一样了。

如果请求未认证则 `request.user` 的默认值为 `django.contrib.auth.models.AnonymousUser` 的一个实例。

.auth

`request.auth` 返回所有附加的身份验证上下文。 其实际行为取决于所使用的具体认证策略。如果请求未认证或者没有其他上下文，则 `request.auth` 的默认值为 `None` 。

.authenticators

使用的认证策略。通常并不需要访问此属性。

```
    def _authenticate(self):
        """
        Attempt to authenticate the request using each authentication instance
        in turn.
        """
        for authenticator in self.authenticators:
            try:
                user_auth_tuple = authenticator.authenticate(self)
            except exceptions.APIException:
                self._not_authenticated()
                raise

            if user_auth_tuple is not None:
                self._authenticator = authenticator
                self.user, self.auth = user_auth_tuple
                return

        self._not_authenticated()
```

浏览器相关

REST framework 支持一些浏览器增强功能，例如基于浏览器的 `PUT`，`PATCH` 和 `DELETE` 表单功能。

下面是request对象中关于浏览器增强相关的一些属性：

.method

`request.method` 返回请求的HTTP方法的 **大写** 字符串表示形式。

隐含支持基于浏览器的 `PUT`，`PATCH` 和 `DELETE` 方法。

.content_type

`request.content_type` 返回HTTP请求正文的媒体类型的字符串对象，如果未提供媒体类型，则返回空字符串。也就是正文的格式。

一般我们不使用这个属性，但如果真要在DRF中访问请求的内容类型，请务必使用 `.content_type` 属性，而不是使用 `request.META.get('HTTP_CONTENT_TYPE')`，因为前者为基于浏览器的非表单内容提供了隐含的支持。

.stream

`request.stream` 返回请求主体内容的流形表示。

我们通常不需要直接访问请求的内容主体，而是依赖REST framework默认的请求解析行为。

request._request

如果你强烈需要使用Django原生的request对象，请这么调用： `request._request`

由于 REST framework 的 `Request` 对象扩展了 Django的 `HttpRequest`对象, 所以所有Django原生的标准属性和方法也是可用的。例如 `request.META` 和 `request.session` 字典正常可用。

请注意，由于实现原因，`Request`该类不继承自`HttpRequest`类，而是使用composition扩展了该类。

类并不是直接继承Django原生的

11-响应（Responses）

REST框架通过提供一个`Response`类来支持HTTP内容协商，该类允许您根据客户端请求返回可以呈现为多种内容类型的内容。

在`Response`类的子类Django的`SimpleTemplateResponse`。 `Response`对象使用数据初始化，该数据应包含本地Python原语。然后，REST框架使用标准的HTTP内容协商来确定应如何呈现最终响应内容。

不需要使用`Response`该类，也可以根据需要从视图中返回常规`HttpResponse`或`StreamingHttpResponse`对象。使用`Response`该类只是为返回内容协商的Web API响应提供了一个更好的接口，该响应可以呈现为多种格式。

除非出于某种原因要大量定制REST框架，否则应始终对返回对象的视图使用`APIView`类或`@api_view`函数`Response`。这样做可确保在视图返回视图之前，视图可以执行内容协商并为响应选择适当的渲染器。

创建 responses

Response()

签 名：`Response(data, status=None, template_name=None, headers=None, content_type=None)`

与常规的 `HttpResponse` 对象不同，我们不使用渲染过的内容来实例化一个 `Response` 对象，而是传递未渲染的数据，这些数据可以是任何Python基本数据类型。

`Response` 类使用的渲染器无法自行处理像 Django模型实例这样的复杂数据类型，因此你需要在创建 `Response` 对象之前将数据序列化为基本数据类型，比如json。

1 注：这段话我们要深刻理解一下！由于前后端分离了，前端页面的代码不是后端人员写的，前端不知道django模型（甚至不知道后端是什么语言、什么框架、什么服务、什么数据库），不知道数据库的表结构，没有Python中model的代码签名，无法自己反序列化一个Django的模型的对象，就像Python的内置json模块无法序列化一个自定义的Python类一样。
2 然而，由于API形式的前后端分离，通过HTTP传输的内容格式通常都是json或者xml这些类型。为了将Python，也就是Django中的模型中的具体数据内容可以json化，我们需要自己写serializer，也就是序列化器，进行模型model对象的序列化和反序列化。如果不这么做，那前端看着后端发来的数据会懵逼的，后端拿着前端post过来的数据也不知道该怎么存到数据库里去。
3 这完全不同于Django本身的全栈式开发模式，因为在前后端都是一个人编写的情况下，编写者掌握全局，无论前端页面的渲染还是后端数据库的CRUD都由此人掌控，原理上随便怎么定义数据传输接口都可以。

你可以使用 REST framework的 `Serializer` 类来执行此类数据的序列化，或者使用你自定义的序列化器。

参数说明:

- `data` ：要响应的已经序列化了的数据
- `status` ：响应的状态码。默认是
- 200。`template_name` ：当选择 `HTMLRenderer` 渲染器时，指定要使用的模板的名称。
- `headers` ：一个字典，包含响应的HTTP头部信息。
- `content_type` ：响应的内容类型。通常由渲染器自行设置，由协商内容确定，但是在某些情况下，你需要明确指定内容类型。

属性

.data

未渲染的，已经序列化的要响应的数据

.status_code

HTTP 响应的数字状态码。

.template_name

`template_name` 只有在使用 `HTMLRenderer` 或者其他自定义模板作为响应的渲染器时才具有该属性。

标准的HttpResponse 属性

DRF的 `Response` 类扩展了 Django原生的`SimpleTemplateResponse` ，所有原生的属性和方法都是提供的。比如你可以使用标准的方法设置响应的header信息：

```
response = Response()

response['Cache-Control'] = 'no-cache'
```

.render()

和其他的 `TemplateResponse` 一样，调用该方法将序列化的数据渲染为最终的响应内容。当 `.render()` 被调用时， 响应的内容将被设置成在 `accepted_renderer` 实例上调用 `.render(data, accepted_media_type, renderer_context)` 方法返回的结果。

我通常并不需要自己调用 `.render()` ，因为它是由Django的标准响应过程来处理的。

我们只要记住下面几个知识点就可以了：

- 扩展了Django原生的SimpleTemplateResponse
- 可以使用标准的方法设置响应的头部信息
- 会根据内容协商的结果自动渲染成指定的类型
- 执行render()将序列化的数据渲染为最终响应的内容
- 始终使用DRF提供的视图系统，并调用DRF提供的Response

代码查看

```
from rest_framework.views import APIView
class Test(APIView):
    def get(self, request, *args, **kwargs):
        # url 拼接的参数
        print(request._request.GET)  # 二次封装request
        print(request.GET)  # 兼容
        print(request.query_params)  # 扩展,GET请求拼接的参数这里都有
```

```
        return Response('drf get ok')

    def post(self, request, *args, **kwargs):
        # 请求携带的数据包
        print(request._request.POST)  # 二次封装方式,没有json方式的数据
        print(request.POST)  # 兼容,没有json方式的数据
        print(request.data)  # 拓展，兼容性最强，三种数据方式都可以
        print(request.query_params)  # post拼接的数据也可以接受到
        return Response('drf post ok')


path('test/',views.Test.as_view())
```

APIView

REST framework提供了一个 `APIView` 类，它是Django的 `View` 类的子类。

`APIView` 类和Django原生的类视图的 `View` 类有以下不同：

- 传入的请求对象不是Django原生的 `HttpRequest` 类的实例，而是REST framework的 `Request` 类的实例。
- 视图返回的是REST framework的 `Response` 响应，而不是Django的 `HttpResponse` 。视图会管理内容协商的结果，给响应设置正确的渲染器。
- 任何 `APIException` 异常都会被捕获，并且传递给合适的响应。
- 请求对象会经过合法性检验，权限验证，或者阈值检查后，再被派发到相应的视图。

使用 `APIView` 类和使用一般的 `View` 类非常相似，通常，进入的请求会被分发到合适的处理方法比如 `.get()` ，或者 `.post` 。

比如：

```
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):
    """
    View to list all users in the system.

    * Requires token authentication.
    * Only admin users are able to access this view.
    """
    authentication_classes = (authentication.TokenAuthentication,)
    permission_classes = (permissions.IsAdminUser,)

    def get(self, request, format=None):
        """
        Return a list of all users.
        """
        usernames = [user.username for user in User.objects.all()]
        return Response(usernames)
```

APIView的所有属性和方法：

- allowed_methods
- as_view
- authentication_classes
- check_object_permissions
- check_permissions
- check_throttles
- content_negotiation_class
- default_response_headers
- determine_version
- dispatch
- finalize_response
- get_authenticate_header
- get_authenticators
- get_content_negotiator
- get_exception_handler
- get_exception_handler_context
- get_format_suVix
- get_parser_context
- get_parsers
- get_permissions
- get_renderer_context
- get_renderers
- get_throttles
- get_view_description
- get_view_name
- handle_exception
- http_method_names
- http_method_not_allowed
- initial
- initialize_request
- metadata_class
- options
- parser_classes
- perform_authentication
-

-

perform_content_negotiation

permission_classes

- permission_denied
- raise_uncaught_exception
- renderer_classes
- schema
- settings
- setup
- throttle_classes
- throttled
- versioning_class

API视图的属性

了解即可。

- .renderer_classes
- .parser_classes
- .authentication_classes
- .throttle_classes
- .permission_classes
- .content_negotiation_class

API实例化方法属性

通常不需要重写这些方法，了解即可。

- .get_renderers(self)
- .get_parsers(self)
- .get_authenticators(self)
- .get_throttles(self)
- .get_permissions(self)
- .get_content_negotiator(self)
- .get_exception_handler(self)

API实现方法

下面这些方法会在请求被分发到具体的处理方法之前调用。了解即可。

- .check_permissions(self, request)
- .check_throttles(self, request)
- .perform_content_negotiation(self, request, force=False)

分发dispatch()

下面这些方法会被视图的 `.dispatch()` 方法直接调用。它们在调用 `.get` 、 `.post()` 、 `put()` 、`patch()`和`delete()` 之类的请求处理方法之前或者之后执行任何需要执行的操作。

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

在处理方法调用之前进行任何需要的动作。 这个方法用于执行权限认证和限制，并且执行内容协商，通常不需要重写此方法。

.handle_exception(self, exc)

任何被处理请求的方法抛出的异常都会被传递给这个方法，这个方法既不返回 `Response` 的实例，也不重新抛出异常。

默会处理 `rest_framework.expceptions.APIException` 的子类异常，以及Django的 `Http404`和 `PermissionDenied`异常，并且返回一个适当的错误响应。

如果你需要在自己的API中自定义返回的错误响应，你可以重写这个方法。

.initialize_request(self, request, *args,**kwargs)

这个方法确保传递给视图的请求对象是 `Request` 的实例，而不是原生的Django `HttpResquest` 的实例。通常不需要重写这个方法。

.finalize_response(self, request, response, *args, **kwargs)

确保任何从处理请求的方法返回的 `Response` 对象被渲染到由内容协商决定的正确内容类型。通常不需要重写这个方法。

@api_view()

在REST framework中，也可以使用常规的基于函数的视图。DRF提供了一组简单的装饰器，用来包装你的视图函数，以确保视图函数会收到 `Request` （而不是Django原生的 `HttpRequest`)对象，并且返回 `Response` （而不是Django的 `HttpResponse` ）对象，同时允许你设置这个请求的处理方式。

@api_view()装饰器

签名: `@api_view(http_method_names=['GET'], exclude_from_schema=False)`

`api_view` 装饰器的主要参数是响应的HTTP方法的列表。 比如，你可以像这样写一个返回一些数据的非常简单的视图。

```
from rest framework.decorators import  api view

@api_view()
def hello_world(request):
    return Response({"message": "Hello,  world!"})
```

这个视图会使用settings中指定的默认的渲染器，解析器，认证类等等。

默认的情况下，只有 `GET` 请求会被接受。其他的请求方法会得到一个"405 Method Not Allowed"响应。可以像下面的示例代码一样改变默认行为：

```
@api_view(['GET', 'POST'])
def hello_world(request):
    if request.method == 'POST':
        return Response({"message": "Got some data!", "data": request.data})
    return Response({"message": "Hello, world!"})
```

API 访问策略装饰器

REST framework提供了一组可以加到视图上的装饰器来重写一些默认设置。这些装饰器必须放在 `@api_view` 装饰器的 后 (下)面。比如，要创建一个使用限制器确保特定用户每天只能调用一次的视图，可以用 `@throttle_classes` 装饰器并给它传递一个限制器类的列表。

```
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 today! See you tomorrow!"})
```

这些装饰器和前文中的 `APIView` 的子类中设置的属性相对应。

可用的装饰器有：

- `@renderer_classes(...)`
- `@parser_classes(...)`
- `@authentication_classes(...)`
- `@throttle_classes(...)`
- `@permission_classes(...)`

这些装饰器都只接受一个参数，这个参数必须是类的列表或元组。

视图模式装饰器

要重写默认的基于函数的视图生成的模式，你需要使用 `@schema` 装饰器。它必须放在 `@api_view` 装饰器后面，例如：(不重要，不使用)

```
from rest_framework.decorators import api_view, schema
from rest_framework.schemas import AutoSchema

class CustomAutoSchema(AutoSchema):
    def get_link(self, path, method, base_url):
        # override view introspection  here...

@api_view(['GET'])
@schema(CustomAutoSchema())
def view(request):
    return Response({"message": "Hello for today! See you tomorrow!"})
```

如果给装饰器传递一个 `None` 参数值，那么会将函数排除在模式生成之外。

```
@api_view(['GET'])
@schema(None)
def view(request):
    return Response({"message": "Will not appear in schema!"})
```

13-通用视图（Generic-views）

一、概述

基于类的视图的主要优点之一是它们允许你组合一些可重用的行为，并将代码抽象出来，称为可重用的对象。 REST framework同样提供了许多预先构建的通用视图，快速构建与数据库模型密切映射的API视图。

如果通用视图不适合你的API的需求，你可以选择使用常规 `APIView` 类，或重用通用视图使用的mixins和基类来组成你自己的一组可重用的通用视图。

范例

通常在使用通用视图时，你将覆盖视图，并设置多个类属性。

```python
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, )
```

- 首先要注意UserList继承的是谁
- 其次要在类里获取查询集
- 然后要指定使用的序列化器
- 最后还要考虑认证、权限、限流、分页等功能

对于更复杂的情况，可能还想重写视图类上的各种方法。比如：

```
class UserList(generics.ListCreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission classes = (IsAdminUser,)

    def list(self, request):
        # 注意下方使用了self.get_queryset()
        queryset = self.get_queryset()
        serializer = UserSerializer(queryset,  many=True)
        return Response(serializer.data)
```

在urls.py中我们使用 `.as_view()` 方法，表明这是一个类视图。 比如：你的URLconf可能如下：

```
path('users/', ListCreateAPIView.as_view\(queryset=User.objects.all(),
serializer_class=UserSerializer), name='user-list')
```

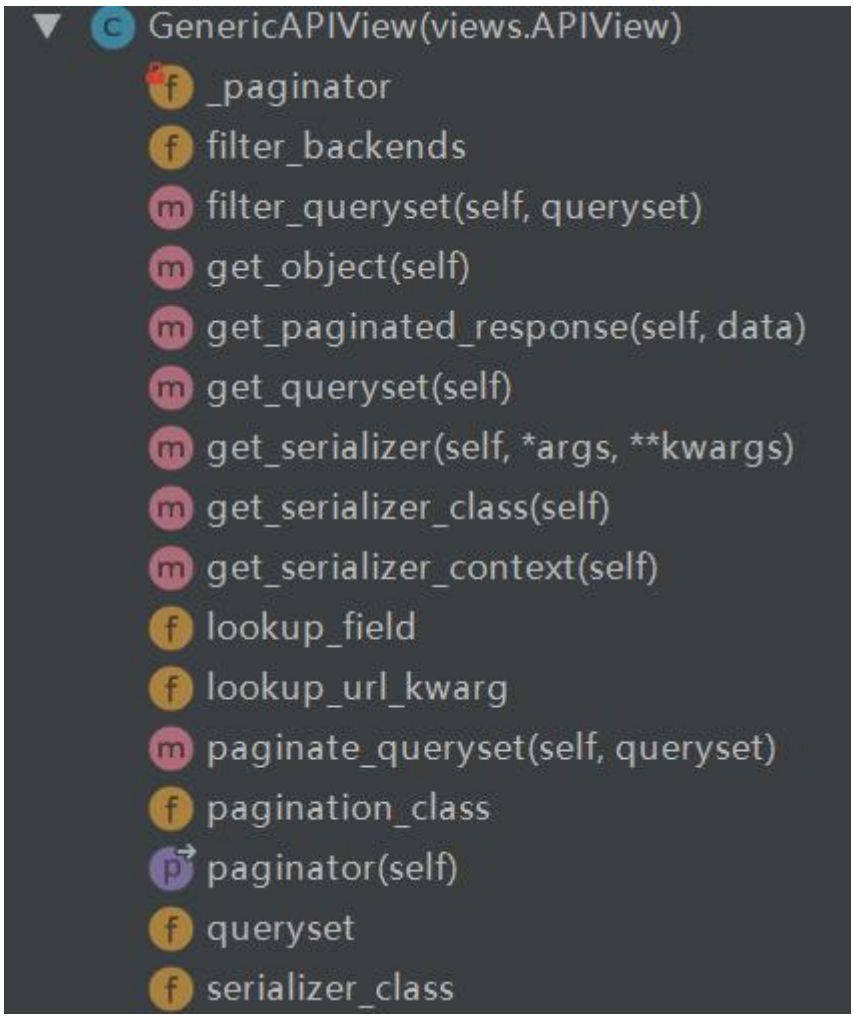
二、API参考

GenericAPIView

DRF通过多父类继承的方式，实现了各个不同的功能类。父类主要有两种，一种是mixin，一个是GenericAPIView。

GenericAPIView是此后所有具体APIView类的结构主父类，此类扩展了REST框架的 `APIView` 类，为标准list和detail视图添加了一般性的操作。

下面列出了 `GenericAPIView` 所有提供的方法和属性：



当你不知道如何使用GenericAPIView**的时候，请来查询上图，或许能给你一些启发。**

属性

基本设置:

以下属性控制着视图的基本行为。

- `queryset` - **必须指定！**用于从视图返回对象的查询结果集。通常，你必须设置此属性或者重写 `get_queryset()` 方法。如果你重写了一个视图的方法，你应该调用 `get_queryset()` 方法而不是直接访问该属性，因为 `queryset` 将被计算一次，这些结果将为后续请求缓存起来。
- `serializer_class` - 用于验证和反序列化输入以及用于序列化输出的Serializer类。 通常，你必须设置此属性或者重写 `get_serializer_class()` 方法。
- `lookup_field` - 用于执行各个model实例的对象查找的model字段。默认为 `'pk'` 。 请注意，在使用超链接API时，如果需要使用自定义的值，你需要确保在API视图和序列化类中都设置查找字段。

- `lookup_url_kwarg` - 应用于对象查找的URL关键字参数。它的 URL conf 应该包括一个与这个值相对应的关键字参数。如果取消设置，默认情况下使用与 `lookup_field` 相同的值。

```
class  PostDetail(mixins.RetrieveModelMixin,
                  mixins.UpdateModelMixin,
                  mixins.DestroyModelMixin,
                  generics.GenericAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
    lookup_url_kwarg =  'sn'                  # 看这里

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args,  **kwargs)

 #urls.py
 path('posts/<int:sn>/',  views.PostDetail.as_view()),    # 注意其中的参数名
```

Pagination:

以下属性用于在与列表视图一起使用时控制分页功能。

- `pagination_class` - 使用的分页类。默认值为 设置的 `DEFAULT_PAGINATION_CLASS`

Filtering:

- `filter_backends` - 用于过滤查询集的过滤器后端类的列表。默认值为 `DEFAULT_FILTER_BACKENDS` 设置的值。

方法

`get_queryset(self)`

返回list视图中使用的查询集，该查询集还用作detail视图中的查找基础。默认返回由 `queryset` 属性指定的查询集。 平时我们应该多使用这个方法，而不是直接访问 `self.queryset`， 因为 `self.queryset` 只会被提交一次（Django的ORM的缓存特性），然后这些结果将为后续的请求缓存起来。 该方法可能会被重写以提供动态行为。

源码:

```
    def get_queryset(self):
        assert self.queryset is not None, (
            "'%s' should either include a `queryset` attribute, "
            "or override the `get_queryset()` method."
            % self.__class__.___name___
        )

        queryset = self.queryset
        if isinstance(queryset, QuerySet):
            # Ensure queryset is re-evaluated on each request.
            queryset = queryset.all()
        return queryset
```

```
def get_queryset(self):
    user = self.request.user
    return user.accounts.all()
```

`get_object(self)`

返回用于detail视图的对象实例。默认使用 `lookup_field` 参数过滤基本的查询集。

get_object(self) 返回详情视图所需的模型类数据对象，默认使用lookup_field参数来过滤queryset。 在视图中可以调用该方法获取详情信息的模型类对象。

若详情访问的模型类对象不存在，会返回404。

该方法会默认使用APIView提供的check_object_permissions方法检查当前对象是否有权限被访问。

该方法可以被重写以提供更复杂的行为，例如基于多个 URL 参数的对象查找。源码：

```
    def get_object(self):

        queryset = self.filter_queryset(self.get_queryset())

        # Perform the lookup filtering.
        lookup_url_kwarg = self.lookup_url_kwarg or  self.lookup_field

        assert lookup_url_kwarg in self.kwargs, (
            'Expected view %s to be called with a URL keyword argument '
            'named "%s". Fix your URL conf, or set the `.lookup_field` '
            'attribute on the view correctly.' %
            (self.__class__.__name__, lookup_url_kwarg)
        )

        filter_kwargs = {self.lookup_field: self.kwargs[lookup_url_kwarg]}
        obj = get_object_or_404(queryset, **filter_kwargs)

```

```
            # May raise a permission denied
            self.check_object_permissions(self.request,  obj)

            return obj
```

例如:

```
    def get_object(self):
        queryset = self.get_queryset()
        filter = {}
        for field in  self.multiple_lookup_fields:
            filter[field] = self.kwargs[field]

        obj = get_object_or_404(queryset, **filter)
        self.check_object_permissions(self.request, obj)
        return obj
```

请注意，如果你的API不包含任何对象级的权限控制，你可以选择不执行 `self.check_object_permissions` ，简单的返回 `get_object_or_404` 查找的对象即可。

https://www.django-rest-framework.org/tutorial/3-class-based-views/

`filter_queryset(self, queryset)`

给定一个queryset，使用任何过滤器后端进行过滤，返回一个新的queryset。

源码:

```
        def filter_queryset(self, queryset):

            for backend in list(self.filter_backends):
                queryset = backend().filter_queryset(self.request, queryset,
    self)
            return queryset
```

例如:

```
def filter_queryset(self, queryset):
    filter backends =  (CategoryFilter,)

    if 'geo_route' in self.request.query_params:
        filter_backends = (GeoRouteFilter,  CategoryFilter)
    elif 'geo_point' in self.request.query_params:
        filter_backends = (GeoPointFilter,  CategoryFilter)

    for backend in list(filter_backends):
        queryset = backend().filter_queryset(self.request, queryset,
view=self)

    return queryset
```

get_serializer_class(self)

选择你想要使用的序列化类。默认返回 `serializer_class` 属性的值。

可以被重写以提供动态的行为，例如对于读取和写入操作使用不同的序列化器，或者为不同类型的用户提供不同的序列化器。

源码：

```
    def  get serializer class(self):

        assert self.serializer_class is not None, (
            "'%s' should either include a `serializer_class` attribute, "
            "or override the `get_serializer_class()` method."
            % self.__class__.___name___
        )

        return self.serializer class
```

例如:

```
def get_serializer_class(self):
    if self.request.user.is_staff:
        return FullAccountSerializer
    return BasicAccountSerializer
```

保存与删除操作中提供的钩子（回调函数：这个函数是给别人调用的）：

以下方法由mixin类提供，并提供对象保存或删除行为的简单重写。

- `perform_create(self, serializer)` - 在保存新对象实例时由 `CreateModelMixin`调 用 。
- `perform_update(self, serializer)` - 在保存现有对象实例时由 `UpdateModelMixin`调 用 。
- `perform_destroy(self, instance)` - 在删除对象实例时由 `DestroyModelMixin` 调用。

这些钩子对于设置请求中隐含的但不是请求数据的一部分的属性特别有用。例如，你可以根据请求用户或基于URL关键字参数在对象上设置属性。

```
def perform_create(self, serializer):
    serializer.save(user=self.request.user)
```

这些钩子对于在保存对象之前或之后添加一些你想要的的操作特别有用（例如通过电子邮件发送确认或记录更新日志）。

```
def perform_update(self, serializer):
    instance = serializer.save()
    send_email_confirmation(user=self.request.user, modified=instance)
```

你还可以使用这些钩子来提供额外的验证，如果不通过则抛出 `ValidationError()` 。当你需要在数据库保存时应用一些验证逻辑时，这会很有用。 例如:

```
def perform_create(self, serializer):
    queryset = SignupRequest.objects.filter(user=self.request.user)
    if queryset.exists():
        raise ValidationError('You have already signed up')
    serializer.save(user=self.request.user)
```

其他方法:

通常并不需要重写以下方法，虽然在你使用 `GenericAPIView` 编写自定义视图的时候可能会调用它们。

- `get_serializer_context(self)` - 返回包含应该提供给序列化程序的任何额外上下文的字典。默认包含 `'request'` ,`'view'` 和 `'format'` 这些键。
- `get_serializer(self, instance=None, data=None, many=False, partial=False)` - 返回一个序列化器的实例。
- `get_paginated_response(self, data)` - 返回分页样式的 `Response` 对象。
- `paginate_queryset(self, queryset)` - 如果需要分页查询，返回页面对象，如果没有为此视图配置分页，则返回 `None` 。
- `filter_queryset(self, queryset)` - 给定查询集，使用任何过滤器后端进行过滤，

返回一个新的查询集。

三、Mixins

Mixin 类用于提供视图的基本操作行为。注意mixin类提供动作方法，而不是直接定义处理程序方法，例如 `.get()` 和 `.post()`， 这允许更灵活的自定义。

Mixin 类可以从 `rest_framework.mixins` 导入。

这个模块的代码相当简单只有不到100行：

```
from__ future___import unicode_literals
from rest_framework import status
from rest_framework.response import Response
from rest_framework.settings import api_settings


class CreateModelMixin(object):

    def create(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        self.perform_create(serializer)
        headers = self.get_success_headers(serializer.data)
        return Response(serializer.data, status=status.HTTP_201_CREATED,
headers=headers)

    def perform_create(self, serializer):
        serializer.save()

    def get_success_headers(self, data):
        try:
            return {'Location': str(data[api_settings.URL_FIELD_NAME])}
        except (TypeError, KeyError):
            return {}


class ListModelMixin(object):

    def list(self, request, *args, **kwargs):
        queryset = self.filter_queryset(self.get_queryset())

        page = self.paginate_queryset(queryset)
```

```
32            if page is not None:
33                serializer = self.get_serializer(page, many=True)
```

```
34              return self.get_paginated_response(serializer.data)
35
36          serializer = self.get_serializer(queryset, many=True)
37          return Response(serializer.data)
38
39
40  class RetrieveModelMixin(object):
41
42      def retrieve(self, request, *args, **kwargs):
43          instance = self.get_object()
44          serializer = self.get_serializer(instance)
45          return Response(serializer.data)
46
47
48  class UpdateModelMixin(object):
49
50      def update(self, request, *args, **kwargs):
51          partial = kwargs.pop('partial', False)
52          instance = self.get_object()
53          serializer = self.get_serializer(instance, data=request.data,
    partial=partial)
54          serializer.is_valid(raise_exception=True)
55          self.perform_update(serializer)
56
57          if getattr(instance, '_prefetched_objects_cache', None):
58              # If 'prefetch_related' has been applied to a queryset, we need
    to
59              # forcibly invalidate the prefetch cache on the  instance.
60              instance._prefetched_objects_cache = {}
61
62          return Response(serializer.data)
63
64      def perform_update(self, serializer):
65          serializer.save()
66
67      def partial_update(self, request, *args, **kwargs):
68          kwargs['partial'] = True
69          return self.update(request, *args, **kwargs)
70
71
72  class DestroyModelMixin(object):
```

```
73
    74    def destroy(self, request, *args, **kwargs):
75            instance = self.get_object()
```

```
76            self.perform_destroy(instance)
77            return  Response(status=status.HTTP_204_NO_CONTENT)
78
79        def perform_destroy(self,  instance):
80            instance.delete()
```

下面我们看一下具体的mixin类：

ListModelMixin

提供一个 `.list(request, *args, **kwargs)` 方法，返回查询结果的列表。

如果查询集被填充了数据，则返回 `200 OK` 响应，将查询集的序列化表示作为响应的主体。相应数据可以任意分页。

CreateModelMixin

提供 `.create(request, *args, **kwargs)` 方法，实现创建和保存一个新model实例的功能。

如果创建了一个对象，这将返回一个 `201 Created` 响应，将该对象的序列化表示作为响应的主体。如果序列化的表示中包含名为 `url` 的键，则响应的 `Location` 头将填充该值。

如果为创建对象提供的请求数据无效，将返回 `400 Bad Request` ，其中错误详细信息作为响应的正文。

RetrieveModelMixin

提供一个 `.retrieve(request, *args, **kwargs)` 方法，返回响应中现有模型的实例。

如果可以检索对象，则返回 `200 OK` 响应，将该对象的序列化表示作为响应的主体。否则将返回 `404 Not Found` 。

UpdateModelMixin

提供 `.update(request, *args, **kwargs)` 方法，实现更新和保存现有模型实例的功能。

同时还提供了一个 `.partial_update(request, *args, **kwargs)` 方法，这个方法和 `update` 方法类似，但更新的所有字段都是可选的。这允许支持 HTTP `PATCH` 请求。

如果一个对象被更新，这将返回一个 `200 OK` 响应，并将对象的序列化表示作为响应的主体。

如果为更新对象提供的请求数据无效，将返回一个 `400 Bad Request` 响应，错误详细信息作为响应的正文。

DestroyModelMixin

提供一个 `.destroy(request, *args, **kwargs)` 方法，实现删除现有模型实例的功能。

如果成功删除对象，则返回 `204 No Content` 响应，否则返回 `404 Not Found` 。

DRF对mixin类的设计是让它们可以尽量的组合使用，不是一次只能继承一个mixin，可以同时继承多个mixin。

四、具体的通用类视图

以下类是具体的通用视图，也是我们平时真正使用的类，除非你需要深度定制，否则不要直接使用上面的父类。

这些视图类可以从 `rest_framework.generics` 导入。

CreateAPIView

仅用于创建功能的视图。提供 `post` 方法。

我们看看它的源码：

```
class  CreateAPIView(mixins.CreateModelMixin,
                     GenericAPIView):

    def post(self, request, *args, **kwargs):
        return self.create(request, *args,  **kwargs)
```

其实就是先继承了CreateModelMixin，然后继承GenericAPIView，最后留个post方法的坑。后面的类的构造，基本也是这个套路。

ListAPIView

以只读的方式列出某些查询对象的集合。提供 `get` 方法。

```
class ListAPIView(mixins.ListModelMixin, GenericAPIView):

    def get(self, request, *args, **kwargs):
        return self.list(request, *args,  **kwargs)
```

RetrieveAPIView

以只读的形式获取某个对象。提供 `get` 方法。

以下部分所有的源码：

```python
class RetrieveAPIView(mixins.RetrieveModelMixin, GenericAPIView):

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)


class DestroyAPIView(mixins.DestroyModelMixin, GenericAPIView):

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)


class UpdateAPIView(mixins.UpdateModelMixin, GenericAPIView):

    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def patch(self, request, *args, **kwargs):
        return self.partial_update(request, *args,  **kwargs)


class ListCreateAPIView(mixins.ListModelMixin,
                        mixins.CreateModelMixin,
                        GenericAPIView):

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

```
27          return self.list(request, *args, **kwargs)
28
29      def post(self, request, *args, **kwargs):
30          return self.create(request, *args, **kwargs)
31
32
33  class RetrieveUpdateAPIView(mixins.RetrieveModelMixin,
34                              mixins.UpdateModelMixin,
35                              GenericAPIView):
36
37      def get(self, request, *args, **kwargs):
38          return self.retrieve(request, *args, **kwargs)
39
40      def put(self, request, *args, **kwargs):
41          return self.update(request, *args, **kwargs)
42
43      def patch(self, request, *args, **kwargs):
44          return self.partial_update(request, *args, **kwargs)
45
46
47  class RetrieveDestroyAPIView(mixins.RetrieveModelMixin,
48                               mixins.DestroyModelMixin,
49                               GenericAPIView):
50
51      def get(self, request, *args, **kwargs):
52          return self.retrieve(request, *args, **kwargs)
53
54      def delete(self, request, *args, **kwargs):
55          return self.destroy(request, *args, **kwargs)
56
57
58  class RetrieveUpdateDestroyAPIView(mixins.RetrieveModelMixin,
59                                     mixins.UpdateModelMixin,
60                                     mixins.DestroyModelMixin,
61                                     GenericAPIView):
62
63      def get(self, request, *args, **kwargs):
64          return self.retrieve(request, *args, **kwargs)
65
66      def put(self, request, *args, **kwargs):
67          return self.update(request, *args, **kwargs)
```

```
68
    69    def patch(self, request, *args, **kwargs):
70            return self.partial_update(request, *args, **kwargs)
```

```
71
72      def delete(self, request, *args, **kwargs):
73          return self.destroy(request, *args,  **kwargs)
```

DestroyAPIView

删除单个模型实例。提供 `delete` 方法。

UpdateAPIView

更新单个模型实例。提供 `put` 和 `patch` 方法。

ListCreateAPIView

创建或者列出模型实例的集合。提供 `get` 和 `post` 方法。同时继承了ListModelMixin和CreateModelMixin两个mixin类，以及基类 GenericAPIView。

RetrieveUpdateAPIView

读取或更新单个模型实例。提供 `get` , `put` 和 `patch` 方法的占坑。

RetrieveDestroyAPIView

读取或删除单个模型实例。提供 `get` 和 `delete` 方法。

RetrieveUpdateDestroyAPIView

读写删除单个模型实例。提供 `get` , `put` , `patch` 和 `delete` 方法。

五、自定义通用视图

创建自定义 mixins

如果你需要基于 URL conf中的多个字段查找对象，则可以创建一个如下所示的 mixin类：

```
class MultipleFieldLookupMixin(object):

    def get_object(self):
        queryset = self.get_queryset()                # 获取基础的查询集
        queryset = self.filter_queryset(queryset)     # 先把默认的过滤查询做了
        filter = {}
        for field in self.lookup_fields:
            if self.kwargs[field]: # Ignore empty fields.
                filter[field] = self.kwargs[field]
        obj = get object or 404(queryset, **filter)   # 实施自定义的过滤
        self.check_object_permissions(self.request, obj)   #检查以下权限
        return obj
```

然后，你可以在需要应用自定义行为的视图类中继承此mixin类。

```
class RetrieveUserView(MultipleFieldLookupMixin, generics.RetrieveAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_fields = ('account', 'username')
```

自定义基类

如果你在多个视图中使用了同一个mixin，你可以创建你自己的一组基本视图，然后在整个项目中使用。举个例子:

```
class BaseRetrieveView(MultipleFieldLookupMixin,
                       generics.RetrieveAPIView):
    pass

class BaseRetrieveUpdateDestroyView(MultipleFieldLookupMixin,
                                    generics.RetrieveUpdateDestroyAPIView):
    pass
```

如果你的自定义行为始终需要在整个项目中的大量视图中重复，使用自定义基类是一个不错的选择。

一、概述

Django REST framework允许你将一组相关视图的逻辑组合在单个类（称为其 `ViewSet` 在他框架中，你也可以找到类似于 'Resources' 或 'Controllers'的概念。

`ViewSet` 只是**一种基于类的视图，它不提供任何方法处理程序**（如 `.get()` 或 `.post()`），而是提供诸如 `.list()` 和 `.create()` 之类的操作。

`ViewSet` 是比前面的通用类视图更深入的封装，简化了更多的代码。它并不高大上，也没有提高性能，用于不用，取决于你的需求，既不重点推荐也不强制使用。

范例

让我们定义一个简单的视图集，可以用来列出或检索系统中的所有用户。

```
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)
```

如果我们需要，我们可以将这个viewset绑定到两个单独的视图，也就是将传统的get、 post、 put、 delete这些HTTP方法的名字映射成list、 create、 update、 retrieve、 destroy等方法名，像这样：

```
user_list = UserViewSet.as_view({'get': 'list'})
user_detail = UserViewSet.as_view({'get': 'retrieve'})
```

这么做，为的是将DRF的视图中的方法区分开，不至于混淆。

通常我们不会这么做，我们会用一个router来注册我们的viewset，让urlconf自动生成。

```
from myapp.views import UserViewSet
from rest framework.routers import  DefaultRouter

router = DefaultRouter()
router.register(r'users', UserViewSet, basename='user')
urlpatterns = router.urls
```

你会发现DRF为viewset设计了专门的路由模式编写方法，WTF，能不能简单一点？不要搞这么复杂！完全没有统一的设计思维，换一种视图就换一个套路....

然后，文档的编写者的思路又直接跳到这里了：

你不需要编写自己的视图集，直接使用提供默认行为的现有基类即可。例如：

```
class UserViewSet(viewsets.ModelViewSet):
    """
    用于查看和编辑用户实例的视图集。
    """
    serializer_class = UserSerializer
    queryset = User.objects.all()
```

> 前面的ViewSet基类还没说清楚，又冒出来一个ModelViewSet类。这文档写得够烂，对新手太不友好。

与使用前面章节的 类相比，使用 `ViewSet` 类有两个主要优点。

- 重复的逻辑可以组合成一个类。在上面的例子中，我们只需要指定一次 `queryset` ，它将在多个视图中使用。
- 通过使用 routers，不再需要自己处理URLconf。

`View`

这两者都有一个权衡。使用常规的 views 和 URL confs 更明确，也能够提供更多的控制。ViewSets有助于快速启动和运行，或者当你有大型的API，并且希望在整个过程中执行一致的URL 配置。

自带路由，无需额外添加

REST framework 中包含的默任 routes为标准的 create/retrieve/update/destroy 操作提供了路由，也就是说你可以不用写了（省略了什么蛋用，差你这点代码吗？但带来的学习成本，太高了）。如下所示：

```
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
```

在dispatch过程中，下列属性可用于 `ViewSet` ：

- `basename` - 根url路径
- `action` - 当前动作类型(例如 `list` `create`).
- `detail` - 用于指示当前动作是针对一个列表还是一个对象detail的布尔指示器
- `suffix` - viewset类型的前缀
- `name` - viewset的名字
- `description` - 详细描述

可以使用上面的属性来做一些展示和调整。例如下面的例子，重写了获取权限的钩子方法，根据请求的不同，需要不同的权限，list只需要普通登录即可，但其它的则需要管理员身份：

```
def get_permissions(self):
    """
    Instantiates and returns the list of permissions that this view requires.
    """
    if self.action == 'list':
        permission_classes = [IsAuthenticated]
    else:
        permission_classes = [IsAdmin]
    return [permission() for permission in permission_classes]
```

为路由增加额外的方法

如果你有需要被路由用到的额外方法，可以使用 `@action` 装饰器将进行标记。

例如：

```
from django.contrib.auth.models import User
from rest_framework import status, viewsets
from rest_framework.decorators import action
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

    @action(detail=True, methods=['post'])
    def set_password(self, request, pk=None):  # POST /users/<int:pk>/set_password/
        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,
```

```
24                                  status=status.HTTP_400_BAD_REQUEST)
```

```
25
26      @action(detail=False)
27      def recent users(self,  request):      # GET  /users/recent users/
28          recent_users = User.objects.all().order_by('-last_login')
29
30          page = self.paginate_queryset(recent_users)
31          if page is not None:
32              serializer = self.get_serializer(page, many=True)
33              return  self.get_paginated_response(serializer.data)
34
35          serializer = self.get_serializer(recent_users, many=True)
36          return Response(serializer.data)
```

装饰器可以另外获取为路由视图设置的额外参数。例如...

```
1       @action(detail=True, methods=['post'], permission_classes=
    [IsAdminOrIsSelf])
2       def set_password(self, request, pk=None):
3       ...
```

action装饰器将默认路由 `GET` 请求，但也可以通过使用 `methods` 参数接受其他 HTTP 方法。例如：

```
1       @action(detail=True, methods=['post', 'delete'])
2       def unset_password(self, request, pk=None):
3          ...
```

这两个新动作将在 urls `users/<int:pk>/set_password/` 和 `users/<int:pk>/unset_password/` 上可用。

二 API参考

视图集

将`ViewSet`类从继承`APIView`。您可以使用任何标准的属性，如`permission_classes`，`authentication_classes`以控制在视图集的API政策。

本`ViewSet`类不提供任何操作实现。为了使用一个`ViewSet`类，您将覆盖该类并显式定义动作实现。

通用视图集

在`GenericViewSet`从类继承`GenericAPIView`，并提供了默认设置`get_object`，`get_queryset`方法及其他通用视图基地的行为，但不包括默认情况下，任何动作。

为了使用一个`GenericViewSet`类，您将覆盖该类并混合所需的mixin类，或显式定义动作实现。

模型视图集

在`ModelViewSet`从类继承`GenericAPIView`，并包括用于各种动作实现方式中，通过在各种混入类的行为混合。

由提供的动作`ModelViewSet`类是`.list()`，`.retrieve()`，`.create()`，`.update()`，`.partial_update()`，和`.destroy()`。

因为`ModelViewSet`extends `GenericAPIView`，通常需要至少提供`queryset`and `serializer_class`属性。例如：

```
class AccountViewSet(viewsets.ModelViewSet):

    """

    A simple ViewSet for viewing and editing accounts.

    """

    queryset = Account.objects.all()

    serializer_class = AccountSerializer

    permission_classes = [IsAccountAdminOrReadOnly]
```

请注意，您可以使用所提供的任何标准属性或方法替代`GenericAPIView`。例如，要使用`ViewSet`动态确定其应操作的查询集的，您可以执行以下操作：

```
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()
```

但是请注意，`queryset`从您的属性中删除该属性后`ViewSet`，任何关联的路由器都将无法自动派生Model的基本名称，因此您必须`basename`在路由器注册中指定kwarg 。

还要注意，尽管默认情况下此类提供了完整的create / list / retrieve / update / destroy操作集，但是您可以使用标准权限类来限制可用的操作。

ReadOnlyModelViewSet

该`ReadOnlyModelViewSet`班也继承`GenericAPIView`。与一样，`ModelViewSet`它也包含各种动作的实现，但与`ModelViewSet`仅提供“只读”动作不同，`.list()`和`.retrieve()`。

与一样`ModelViewSet`，您通常至少需要提供`queryset`and `serializer_class`属性。例如：

```
class AccountViewSet(viewsets.ReadOnlyModelViewSet):
    """
    A simple ViewSet for viewing accounts.
    """
    queryset = Account.objects.all()
    serializer_class = AccountSerializer
```

同样，`ModelViewSet`您可以使用的任何标准属性和方法替代`GenericAPIView`。

三 自定义ViewSet基类

您可能需要提供`ViewSet`没有完整`ModelViewSet`动作集或以其他方式自定义行为的自定义类。

要创建基础视图集类，提供`create`， `list`和`retrieve`操作，继承`GenericViewSet`和混入所需的操作：

```
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
```

通过创建自己的基`ViewSet`类，您可以提供可以在您的API的多个视图集中重用的常见行为。