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。

```
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：

```python
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')
```

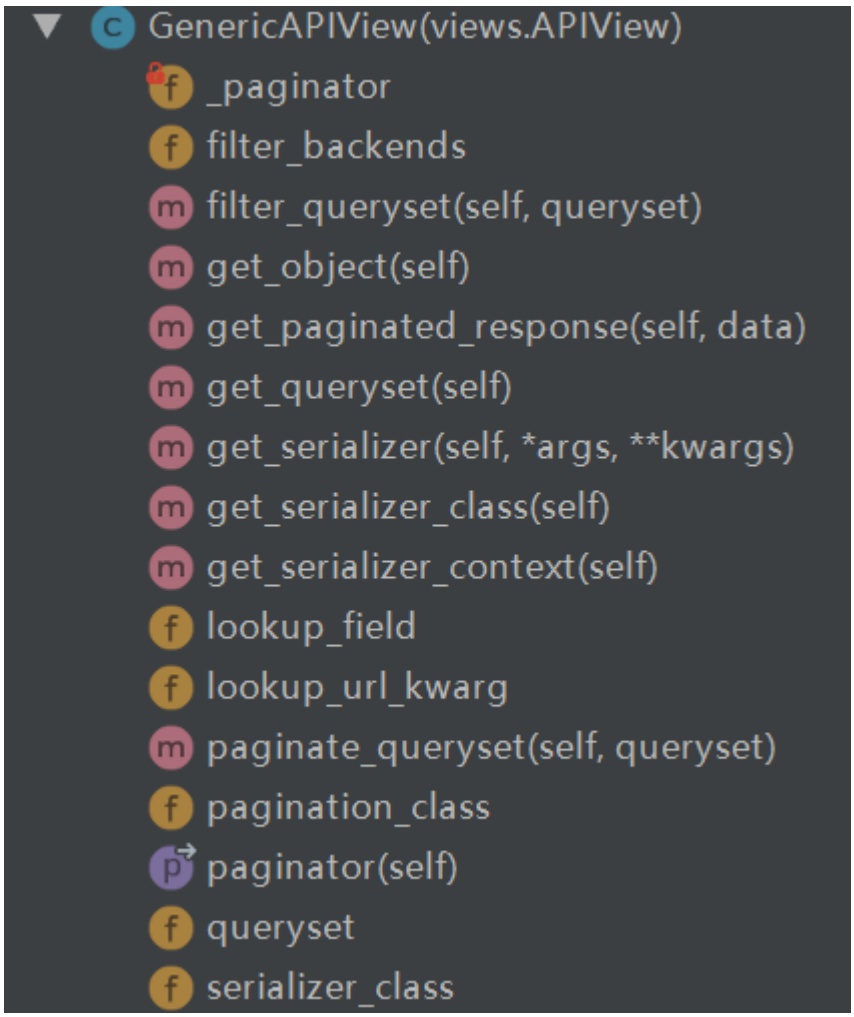
二、API参考

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 `DEFAULT_PAGINATION_CLASS` 设置的值，即 `'rest_framework.pagination.PageNumberPagination'` 。

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。

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。

`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:

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。

```
1   def perform_create(self, serializer):
2       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）。

```
1   def perform_update(self, serializer):
2       instance = serializer.save()
3       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:

```
1   def perform_create(self, serializer):
2       queryset = SignupRequest.objects.filter(user=self.request.user)
3       if queryset.exists():
4           raise ValidationError('You have already signed up')
5       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)
        if page is not None:
            serializer = self.get_serializer(page, many=True)
```

```
            return self.get_paginated_response(serializer.data)

        serializer = self.get_serializer(queryset, many=True)
        return Response(serializer.data)


class RetrieveModelMixin(object):

    def retrieve(self, request, *args, **kwargs):
        instance = self.get_object()
        serializer = self.get_serializer(instance)
        return Response(serializer.data)


class UpdateModelMixin(object):

    def update(self, request, *args, **kwargs):
        partial = kwargs.pop('partial', False)
        instance = self.get_object()
        serializer = self.get_serializer(instance, data=request.data,
partial=partial)
        serializer.is_valid(raise_exception=True)
        self.perform_update(serializer)

        if getattr(instance, '_prefetched_objects_cache', None):
            # If 'prefetch_related' has been applied to a queryset, we need
to
            # forcibly invalidate the prefetch cache on the instance.
            instance._prefetched_objects_cache = {}

        return Response(serializer.data)

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

    def partial_update(self, request, *args, **kwargs):
        kwargs['partial'] = True
        return self.update(request, *args, **kwargs)


class DestroyModelMixin(object):

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

```
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):
```

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

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


class RetrieveUpdateAPIView(mixins.RetrieveModelMixin,
                            mixins.UpdateModelMixin,
                            GenericAPIView):

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

    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 RetrieveDestroyAPIView(mixins.RetrieveModelMixin,
                             mixins.DestroyModelMixin,
                             GenericAPIView):

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

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


class RetrieveUpdateDestroyAPIView(mixins.RetrieveModelMixin,
                                   mixins.UpdateModelMixin,
                                   mixins.DestroyModelMixin,
                                   GenericAPIView):

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

    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)
```

```
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。