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 `ViewSet`）中。 在其他框架中，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 `View` 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。

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,
                            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

ViewSet

`ViewSet` 继承自 `views.APIView` 。你可以使用任何父类属性，
如 `permission_classes` ，`authentication_classes` 以便控制视图集上的 API 策略。

`ViewSet` 类不提供任何操作的实现。为了使用 `ViewSet` 类，你需要重写该类并显式地定义动作实现。

GenericViewSet

`GenericViewSet` 类继承 `GenericAPIView`，并提供 `get_object`， `get_queryset` 方法和其他通用视图基本行为的默认配置，但默认情况不包括任何操作。

ModelViewSet

`ModelViewSet` 又继承了 `GenericAPIView` ，但实现了基本的HTTP请求方法。它提供 `.list()` , `.retrieve()` , `.create()` , `.update()` , `.partial_update()` 和 `.destroy()` 操作。这是我们真正使用的类。

范例

至少需要提供 `queryset` 和 `serializer_class` 属性的值。

```
class AccountViewSet(viewsets.ModelViewSet):
    """
    A simple ViewSet for viewing and editing accounts.
    """
    queryset = Account.objects.all()
    serializer_class = AccountSerializer
    permission_classes = [IsAccountAdminOrReadOnly]
```

可以使用父类 `GenericAPIView` 所有的方法，比如：

```
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()
```

ReadOnlyModelViewSet

`ReadOnlyModelViewSet` 也继承 `GenericAPIView` 。与 `ModelViewSet` 相同的是，它也包括一些动作的实现。不同的是但是只提供只读的 `.list()` 和 `.retrieve()` 动作。

范例

与 `ModelViewSet` 类似，至少需要提供 `queryset` 和 `serializer_class` 属性的值。

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

Again, as with `ModelViewSet` , you can use any of the standard attributes and method overrides available to `GenericAPIView` .

三、自定义ViewSet基类

看下面的例子：

```
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
```