在API视图的编写方法上，DRF为我们提供了很多种选择，比如基于类的视图。这是一个强大的模式，允许我们重用常用的功能，并帮助我们保持代码的DRY特性。

一、使用基于类的视图重写我们的API

我们又将再一次重写 `views.py` ，而且还会有下一次。代码如下：

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from django.http import Http404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status


class SnippetList(APIView):
    """
    List all snippets, or create a new snippet.
    """
    def get(self, request, format=None):
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = SnippetSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

注意类视图的名字规范，和基于函数的视图是不一样的，所以这里换了名字。

在原生的Django中编写类视图是这样的：

```python
# 具体的Django类视图知识可参考http://www.liujiangblog.com/blog/37/

from django.http import HttpResponse
from django.views import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
    def post(self,request):
        # <view logic>
        return HttpResponse('something')
```

比较一下区别，最主要的是我们导入了这个类：`from rest_framework.views import APIView`，在自己写的视图类中继承它。那么这个APIView是什么呢？它是DRF对`Django.view.View`类的高级封装，添加了很多DRF需要使用的特性，比如可浏览的API页面等。

当然，我们还需要更新`views.py`中的detail实例视图。

```python
class SnippetDetail(APIView):
    """
    Retrieve, update or delete a snippet instance.
    """
    def get_object(self, pk):
        try:
            return Snippet.objects.get(pk=pk)
        except Snippet.DoesNotExist:
            raise Http404

    def get(self, request, pk, format=None):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet)
        return Response(serializer.data)

    def put(self, request, pk, format=None):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

```
23
24      def delete(self, request, pk, format=None):
25          snippet = self.get_object(pk)
26          snippet.delete()
27          return Response(status=status.HTTP_204_NO_CONTENT)
```

看起来不错，但它现在仍然非常类似于基于函数的视图。（其实就是拆分了逻辑，实现可重用）

接下来，我们还需要重构我们的 `snippets.urls.py` ，因为Django对类视图有专门的url编写格式，不得不改：

```
1   # 注释了前面的内容，供大家对比参考
2   # from django.urls import path
3   # from snippets import views
4   #
5   # urlpatterns = [
6   #     path('snippets/', views.snippet_list),
7   #     path('snippets/<int:pk>/', views.snippet_detail),
8   # ]
9
10
11  from django.urls import path
12  from rest_framework.urlpatterns import format_suffix_patterns
13  from snippets import views
14
15  urlpatterns = [
16      path('snippets/', views.SnippetList.as_view()),
17      path('snippets/<int:pk>/', views.SnippetDetail.as_view()),
18  ]
19
20  urlpatterns = format_suffix_patterns(urlpatterns)
```

好了，阶段完成，如果你重新启动服务器，那么它应该像之前一样运行。

二、使用混合类（mixins）

等等，你以为DRF的视图体系到此就完了吗？你想得太简单了！一大波内容还在后面.....

使用基于类的视图，最大优势之一是创建可复用的代码。

到目前为止，我们使用的创建/获取/更新/删除操作中有一部分代码是非常类似的，完全可以抽象出来。DRF就这么干了，并把这部分代码放到mixin类系列中，然后作为父类供子类继承复用。

让我们来看看我们是如何通过使用mixin类编写视图的。打开 `views.py` 模块，又要重写这个文件了.....:

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import mixins
from rest_framework import generics

class SnippetList(mixins.ListModelMixin,
                  mixins.CreateModelMixin,
                  generics.GenericAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer

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

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

我们分析下这里的具体实现方式。我们自己写的SnippetList类继承了三个类，其中两个是mixin类，最后是 `GenericAPIView` 类。 `GenericAPIView` 类作为结构主父类，提供了基本的DRF的API类视图的功能， `GenericAPIView` 类直接继承了我们前面使用的 `APIView` 类。而 `ListModelMixin` 和 `CreateModelMixin` 提供 `.list()` 和 `.create()` 操作。

> 另外强调一点Python多继承的特点，继承父类的先后位置关系是有意义的，不可以随意调换顺序。

然后我们明确地将 `get` 和 `post` 方法绑定到适当的操作。

再修改一下我们的Detail类:

```
class SnippetDetail(mixins.RetrieveModelMixin,
                    mixins.UpdateModelMixin,
                    mixins.DestroyModelMixin,
                    generics.GenericAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer

```

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

和上面的那个类非常相似。

三、使用通用的基于类的视图

> 看完上面的内容，感觉又get到了新知识！以后就这么干了！

> 等等，DRF实际上又帮我们抽象了上面的代码，提供了一些通用的类似的类视图。WTF，那你前面还跟我啰嗦那么多？直接使用下面的方法就好了啊！

通过使用mixin类，我们使用更少的代码重写了这些视图，但我们还可以再进一步。REST框架提供了一组已经混合好（mixed-in）的通用的类视图，我们可以使用它来简化我们的 `views.py` 模块。

已经不记得是第几次修改了....

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics


class SnippetList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer


class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
```

你所需要做的，只是继承 `generics` 模块中的现成的某个通用类视图，比如 `ListCreateAPIView` 。然后在类里定义 `queryset` 和 `serializer_class` 两个属性的值，剩下的什么都不用写，全部交给DRF，它会帮你搞定。