资源的增删改查

上节课我们演示了REST框架的运行原理，序列化过程和查询数据的能力。这节课开始我们来学习修改资源的能力。

新增资源

继续更新视图,sqtp/views.py ,编辑request_list视图，增加添加方法

```
@api_view(['GET','POST'])  # 允许的请求方法
def request_list(request,format=None):
    if request.method == 'GET':
        # 构造序列化器
        serializer = RequestSerializer(Request.objects.all(),many=True)
        # 返回json格式数据
        return Response(serializer.data)  # 将python原生格式转成json数据 safe=False是为了支持{}以外的python对象转json
    elif request.method == 'POST':
        serializer = RequestSerializer(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)
```

刷新浏览器，测试新增方法,输入以下内容后，点POST

```
{"step":null,"method":0,"url":"/demo1","params":null,"headers":null,"cookies":null,"json":{"name":"方方","age":18,"addr":"nanjing"},"data":null}
```

成功返回

```
HTTP 201 Created
Allow: POST, GET, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "step": null,
    "method": 0,
    "url": "/demo1",
    "params": null,
    "headers": null,
    "data": null,
    "json": {
        "name": "方方",
        "age": 18,
        "addr": "nanjing"
    }
}
```

修改资源

修改资源往往是针对1个数据的，所以接下来修改requset_detail视图。修改使用的是PUT方法，增加装饰器可用的方法和请求方法判断

```
@api_view(['GET','PUT'])
def requset_detail(request,_id,format=None):
    try:
        req_obj = Request.objects.get(pk=_id) # 根据id查找单个数据
    except Exception:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method=='GET':
        serializer = RequestSerializer(req_obj)
        return Response(serializer.data)
    elif request.method == 'PUT':  # 修改使用PUT方法
        serializer = RequestSerializer(req_obj,data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        # 如果数据不合法就返回400
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

测试修改方法，访问单个资源页面http://127.0.0.1:8000/requests/1，输入以下数据

```
{
    "step": null,
    "method": 0,
    "url": "/example/demo2",
    "params": {
        "age": 66,
        "addr": "nanjing",
        "name": "哈哈哈"
    },
    "headers": {"content_type":"json"},
    "data": {},
    "json": null
}
```

成功返回修改后的数据内容

```
HTTP 200 OK
Allow: OPTIONS, GET, PUT
Content-Type: application/json
Vary: Accept

{
    "step": null,
    "method": 0,
    "url": "/example/demo2",
    "params": {
        "age": 66,
        "addr": "nanjing",
        "name": "哈哈哈"
    },
    "headers": {
```

```
        "content_type": "json"
    },
    "data": {},
    "json": null
}
```

删除资源

与修改一样，删除也是针对1个数据的，接下来修改requset_detail视图，增加删除部分，同样，删除采用的是DELETE方法，所以增加准入方法和DELETE判断。

```
# 返回单个数据
@api_view(['GET','PUT','DELETE'])
def requset_detail(request,_id,format=None):
    try:
        req_obj = Request.objects.get(pk=_id) # 根据id查找单个数据
    except Exception:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method=='GET':
        serializer = RequestSerializer(req_obj)
        return Response(serializer.data)
    elif request.method == 'PUT':  # 修改使用PUT方法
        serializer = RequestSerializer(req_obj,data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        # 如果数据不合法就返回400
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        req_obj.delete() # 直接调用数据对象的delete方法
        return Response(status=status.HTTP_204_NO_CONTENT)
```

测试删除方法，访问单个资源页面http://127.0.0.1:8000/requests/1，发现多了DELETE按钮，点击即可完成删除

基于类的视图

作为开发要不断思考如何让代码保持高内聚，低耦合，因此优化代码的道路上一直都不停歇。目前我们开发的视图是基于函数形式的，特点是灵活，缺点是功能冗余性大，面对常见的增删改查往往要写重复的代码。

REST框架可以用基于类的视图来优化代码结构，我们来一探究竟：

用类重写视图，继承REST框架的APIView类

```
from rest_framework.views import APIView
class RequestList(APIView):
    """
    列出所有的requests或者创建一个新的request.
    """
    def get(self,request,format=None):
        # 构造序列化器
        serializer = RequestSerializer(Request.objects.all(),many=True)
        # 返回json格式数据
```

```
        return Response(serializer.data)  # 将python原生格式转成json数据 safe=False 是为了支持{}以外的python对象
    def post(self,request):
        serializer = RequestSerializer(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)
```

以上代码完成了查询列表和新增单个数据的功能，我们发现除了变成了类的形式，新增了get（处理get请求）和post（处理post请求）方法，其他的几乎都没变。

同时，也更改下详情视图，用类视图重构

```
class RequestDetail(APIView):
    # 覆盖父类的get_object方法
    def get_object(self,_id):
        try:
            return Request.objects.get(pk=_id) # 根据id查找单个数据
        except Exception:
            return Response(status=status.HTTP_404_NOT_FOUND)

    def get(self,request,_id,format=None):
        req_obj = Request.objects.get(pk=_id)
        serializer = RequestSerializer(req_obj)
        return Response(serializer.data)

    def put(self,request,_id,format=None):
        req_obj = Request.objects.get(pk=_id)
        serializer = RequestSerializer(req_obj,data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        # 如果数据不合法就返回400
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self,request,_id,format=None):
        req_obj = Request.objects.get(pk=_id)
        req_obj.delete() # 直接调用数据对象的delete方法
        return Response(status=status.HTTP_204_NO_CONTENT)
```

启动服务之前，修改下sqtp/urls.py,此时我们使用的是基于类的视图

```
urlpatterns = [
    path('requests/', views.RequestList.as_view()),  # 需要调用类视图的as_view方法
    path('requests/<int:_id>', views.RequestDetail.as_view())
]
urlpatterns = format_suffix_patterns(urlpatterns)
```

测试一下增删改查，OK。

通用类视图代码优化1

使用类视图的一个好处就是可以复用相同的功能，只需传入指定的参数即可。在上面的案例中，增删改查的逻辑行为都是确定的。REST框架为我们封装好了逻辑，我们可以用更少的代码来封装视图，比如采用generics模块的通用视图。

```
class RequestList(generics.ListCreateAPIView):
    """
    列出所有的requests或者创建一个新的request.
    """
    queryset = Request.objects.all()
    serializer_class = RequestSerializer


class RequestDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Request.objects.all()
    serializer_class = RequestSerializer
```

修改路由：

```
path('requests/<int:pk>',views.RequsetDetail.as_view())
```

重启服务，测试增删改查，依然OK!

并且我们发现，数据详情的编辑页面改成了按字段进行编辑了，这是通用类视图提供的效果。

上例中用到的两个通用类视图从名字可以知道其功能

ListCreateAPIView：提供列出所有和创建数据功能

RetrieveUpdateDestroyAPIView：提供查询、修改和删除数据功能

通用类视图代码优化2

我们发现，优化后的视图代码依然存在重复的部分，那么这个部分能不能继续优化呢，答案是可以的，我们用ViewSet（视图集）代替View类重构视图。

```
# 删除其余的视图函数
from rest_framework import viewsets

class RequestViewSet(viewsets.ModelViewSet):
    queryset = Request.objects.all()
    serializer_class = RequestSerializer
```

使用REST `ViewSets` 的抽象后，开发人员可以集中精力对API的状态和交互进行建模，并根据常规约定自动处理URL构造。

`ViewSet` 类与 `View` 类几乎相同，不同之处在于它们提供诸如 `read` 或 `update` 之类的操作，而不是 `get` 或 `put` 等方法处理程序。

一个 `ViewSet` 类只绑定到一组方法处理程序，当它被实例化成一组视图的时候，通常通过使用一个 `Router` 类来代替自己定义复杂的URL。

路由设计的自动化

因为我们使用的是`ViewSet`类而不是`View`类，所以连常规的URL设计我们都可以偷懒了。利用rest框架的router，可以帮助我们自动生成路由列表。

重构sqtp/urls.py

```
from rest_framework.routers import DefaultRouter
# 创建路由器并注册我们的视图。
router = DefaultRouter()
router.register(r'requests', views.RequestViewSet)

urlpatterns = [
    path('',include(router.urls)),
]
```

router的作用是根据你注册的路由前缀，帮助你自动生成诸如此类的路由列表

```
^requests/$ [name='request-list']
^requests\.(?P<format>[a-z0-9]+)/?$ [name='request-list']
^requests/(?P<pk>[^/.]+)/$ [name='request-detail']
^requests/(?P<pk>[^/.]+)\.(?P<format>[a-z0-9]+)/?$ [name='request-detail']
```

自定义接口规范

目前我们使用的是REST框架默认的返回格式，类似这种：

```
[
    {
        "id": 1,
        "method": 0,
        "url": "/demo1",
        "params": null,
        "headers": null,
        "data": {
            "age": 18,
            "addr": "nanjing",
            "name": "小明"
        },
        "json": null
    },
    ...
]
```

这样并不利于前端的渲染，我们希望接口在正确的时候返回

```
{"msg":"success","retcode":200,"retlist":[...]}
```

错误的时候返回

```
{"msg":"error","retcode":404,"error":error_msg}
```

想要弄成类似这样的效果需要自定义drf异常返回和自定义数据返回格式，REST框架为我们提供了该技术

在settings.py中添加drf全局配置

```
# rest框架配置
REST_FRAMEWORK = {
    # 默认的渲染器
    'DEFAULT_RENDERER_CLASSES': (
        # 'rest_framework.renderers.JSONRenderer',
        # 'rest_framework.renderers.BrowsableAPIRenderer',
        'utils.renderers.MyRenderer',
    )
}
```

utils.renderers.MyRenderer对应的是你文件路径，路径起点为项目根目录，因此你需要创建utils目录然后在下面新建renderers.py文件

```
from rest_framework.renderers import JSONRenderer
# 继承空返回JSON的渲染器
class MyRenderer(JSONRenderer):
    # 重构 render方法
    def render(self, data, accepted_media_type=None, renderer_context=None):
        print(renderer_context['response'])
        status_code = renderer_context['response'].status_code  #响应状态码
        if str(status_code).startswith('2'): # 以2开头表示响应正常
            res = {'msg':'success','retcode':status_code,'retlist':data}
            # 返回父类方法
            return super().render(res,accepted_media_type,renderer_context)
        else: # 异常请情况
            res = {'msg':'error','retcode':status_code}
            return super().render(res,accepted_media_type,renderer_context)
```