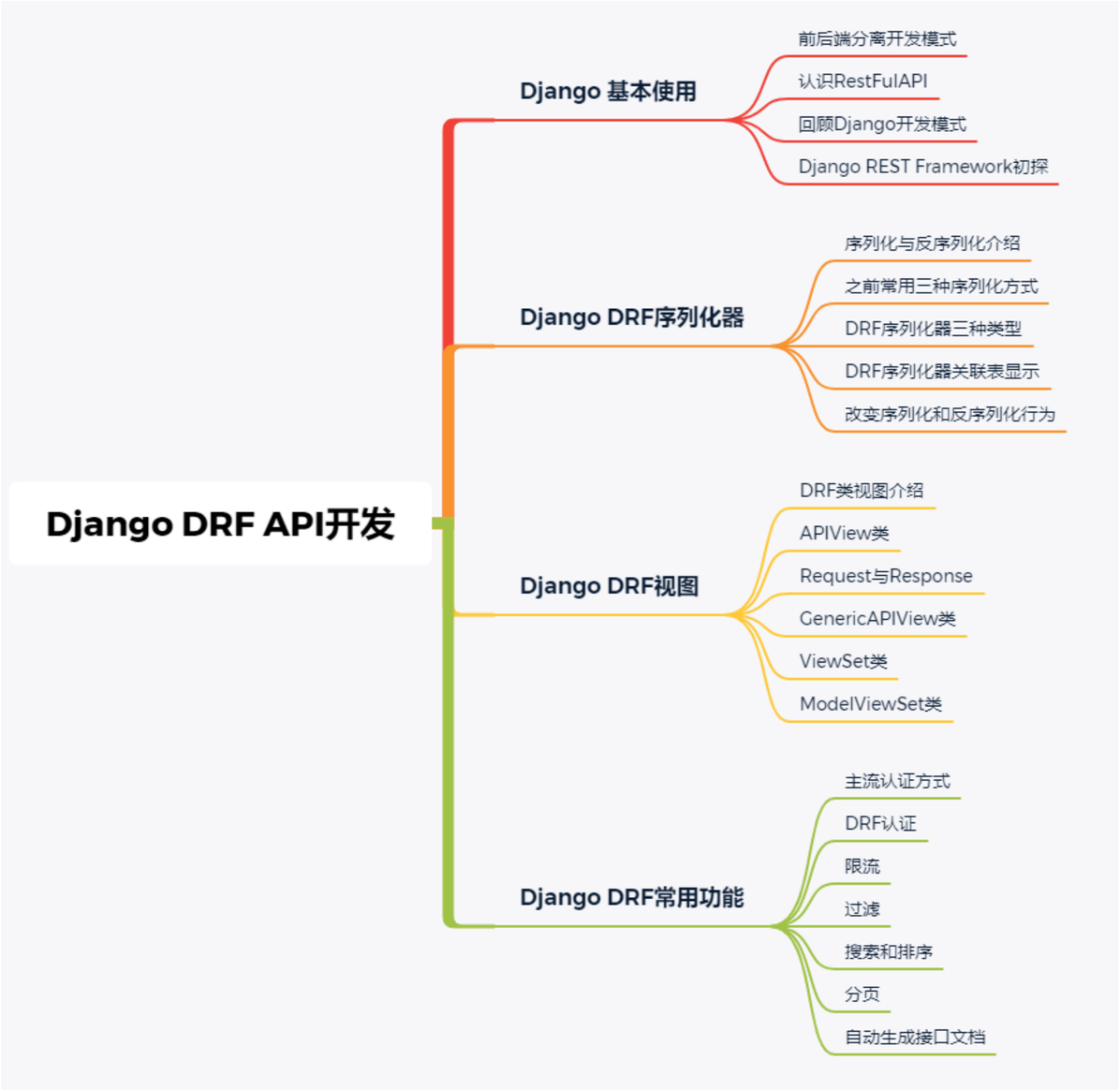
讲师：李振良（阿良）

今天课题：《Django DRF API开发》

学院官网：www.aliangedu.cn

阿良个人微信

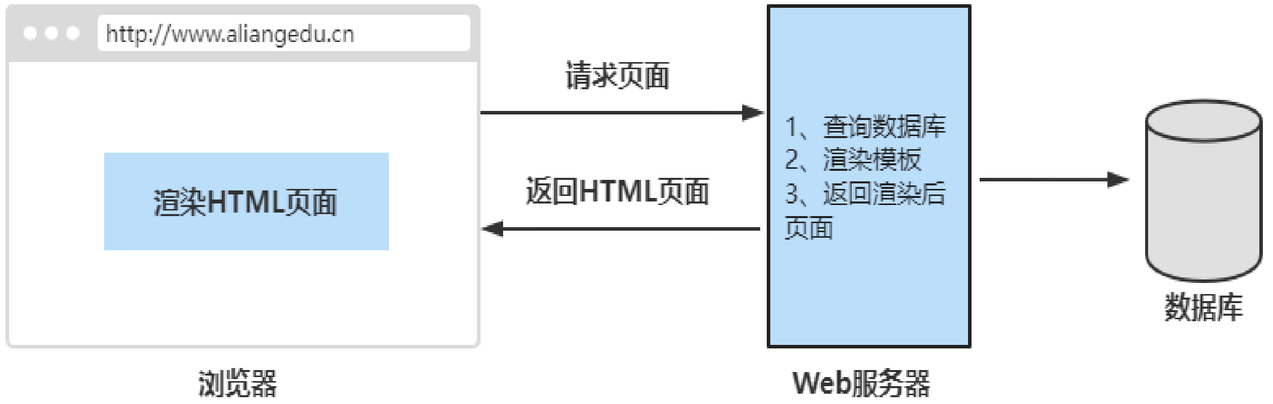
DevOps技术栈公众号


Django DRF API开发
Django 基本使用
前后端分离开发模式
认识RestFulAPI
回顾Django开发模式
Django REST Framework初探
Django DRF序列化器
序列化与反序列化介绍
之前常用三种序列化方式
DRF序列化器三种类型
DRF序列化器关联表显示
改变序列化和反序列化行为
Django DRF视图
DRF类视图介绍
APIView类
Request与Response
GenericAPIView类
ViewSet类
ModelViewSet类
Django DRF常用功能
主流认证方式
DRF认证
限流
过滤
搜索和排序
分页
自动生成接口文档

Django 基本使用

- 前后端分离开发模式
- 认识RestFulAPI
- 回顾Django开发模式
- Django REST Framework初探

前后端分离开发模式



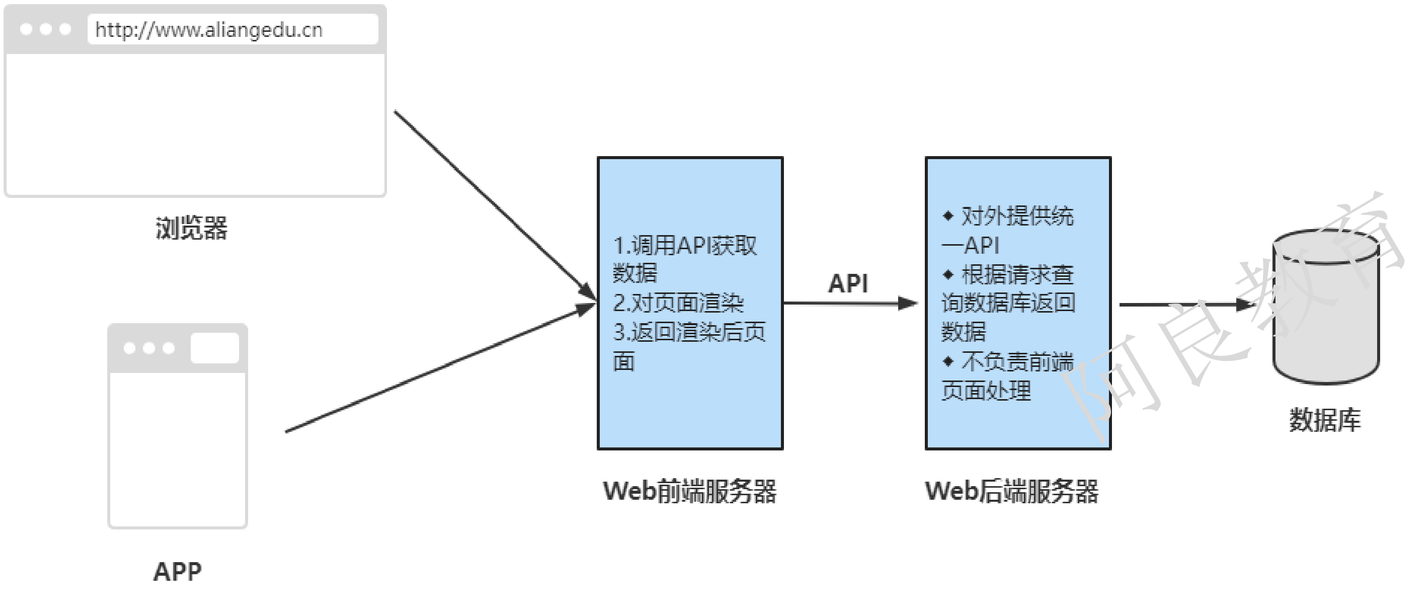
前后端分离前：前端页面看到的效果都是由后端控制，即后端渲染HTML页面，前端与后端的耦合度很高。

前后端分离开发模式

前后端分离前存在的问题：

- PC、APP、Pad等多端流行
- 前后端开发职责不清晰：各司其职，最大程度减少开发难度，方便协作
- 开发效率问题，一般后端开发需先等前端页面准备好，有时前端也一直配合后端，能力受限
- 前后端代码混在一起，日积月累，维护成本增加
- 后端开发语言和模板耦合

前后端分离开发模式



前后端分离后：后端仅返回前端所需要的数据，不再渲染HTML页面，不再控制前端的效果，至于前端展示什么效果，都由前端自己决定。

认识RestFulAPI

什么是RestfulAPI：

- REST（Representational State Transfer，表征状态转移）是一种Web服务的软件架构风格。描述网络中客户端与服务端的一种交互方式，它本身不常用，常用的是如何设计RestfulAPI（REST风格的网络接口）
- RestfulAPI风格就是把所有的数据都当做资源，对表的操作就是对资源操作
- 资源就是指的URL，基于URL对资源操作，Web服务在URL上支持一系列请求方法，如下表所示。

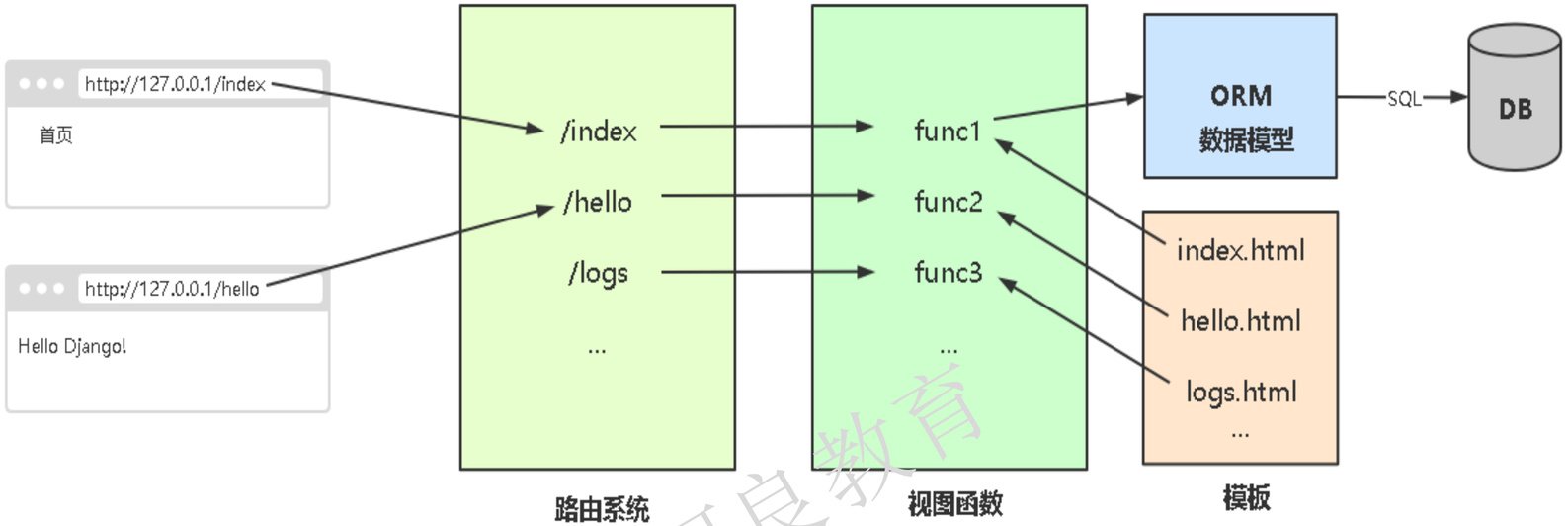
HTTP方法	数据处理	说明
POST	新增	新增一个资源
GET	获取	获取一个资源
PUT	更新	更新一个资源
DELETE	删除	删除一个资源

示例：

非REST的URL：http://ip/get_user?id=123

REST的URL：http://ip/user/123

回顾Django开发模式



工作流程图

回顾Django开发模式

通过一个用户信息管理案例回顾Django开发模式！

目标：

- 熟悉Django项目创建流程
- 熟悉Django与HTML模板渲染
- 熟悉Ajax前后端数据交互
- 熟悉ORM数据库操作

Django REST Framework初探

Django REST framework （简称：DRF）是一个强大而灵活的 Web API 工具。

遵循RESTFullAPI风格，功能完善，可快速开发API平台。

官网文档：https://www.django-rest-framework.org

Django REST Framework初探

Django REST framework 最新版使用要求：

- Python（3.6、3.7、3.8、3.9、3.10）
- Django（2.2、3.0、3.1、3.2、4.0）

安装：

```
pip install djangorestframework
```

添加rest_framework到INSTALLED_APPS设置中：

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

Django REST Framework初探

示例：实现用户增删改查

步骤：

1. 创建APP
2. 定义数据模型并同步数据库
3. 编写序列化器文件
4. 编写视图
5. 添加API路由

Django REST Framework初探

1、创建APP

```
python manage.py startapp myapp_api
```

2、定义数据模型并同步数据库

models.py

```python
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=30)
    city = models.CharField(max_length=30)
    sex = models.CharField(max_length=10)
    age = models.IntegerField()
```

```
python manage.py makemigrations
python manage.py migrate
```

3、编写序列化器文件

myapp_api/serializers.py

models.py | serializers.py

```python
from myapp_api.models import User
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User  # 指定数据模型
        fields = '__all__'  # 显示所有字段
```

4、编写视图

models.py | serializers.py | views.py

```python
from rest_framework import viewsets
from .serializers import UserSerializer
from myapp_api.models import User

class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()  # 指定操作的数据
    serializer_class = UserSerializer  # 指定序列化器
```

5、添加API路由

devops/urls.py

```python
from django.contrib import admin
from django.urls import path,re_path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    re_path('myapp/', include('myapp.urls')),
    re_path('myapp_api/', include('myapp_api.urls'))
]
```

myapp_api/urls.py

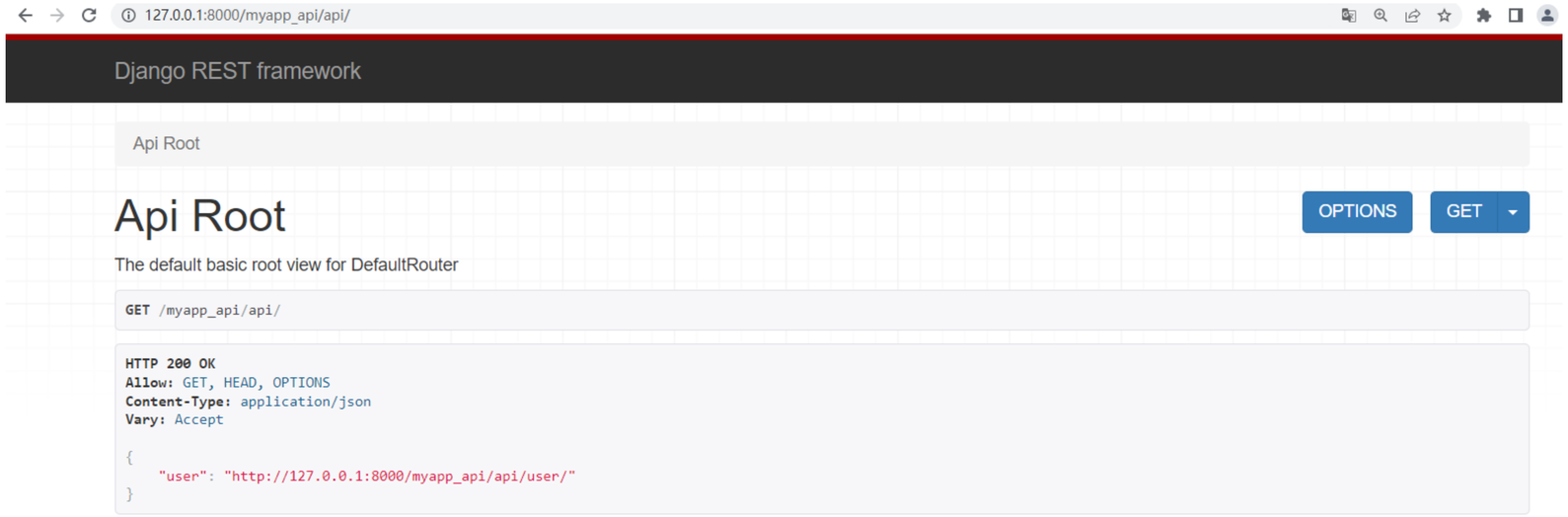
models.py | serializers.py | views.py | urls.py

```python
from django.urls import path, re_path, include
from myapp_api import views
from rest_framework import routers

# 自动注册路由
router = routers.DefaultRouter()
router.register(r'user', views.UserViewSet)

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

Django REST Framework初探



上图是访问/myapp_api/api/地址获得。

这是DRF自带的API调试系统，结果显示自动注册路由的API地址，可通过这个地址完成用户信息数据的增删改查：

增、查：http://127.0.0.1:8000/myapp_api/api/user/

删、改：http://127.0.0.1:8000/myapp_api/api/user/2/

DRF 序列化器

- 序列化与反序列化介绍
- 之前常用三种序列化方式
- DRF序列化器三种类型
- DRF序列化器关联表显示
- 改变序列化和反序列化行为

序列化与反序列化介绍

在日常开发中，会从别的API获取数据或者自己写API提供数据，数据格式一般都是采用JSON格式。这期间就会涉及两个专业术语：

- **序列化：**将python对象转json
- **反序列化：**将json转为python对象

之前常用三种序列化方式：JSON

之前经常用json模式完成序列化与反序列化操作：

- 序列化应用场景示例：用ORM查询数据，采用JSON格式API返回数据。
- 反序列化应用场景示例：从别的API获取数据，在Python里处理。

```
import json
# 序列化
computer = {"主机":5000,"显示器":1000,"鼠标":60,"键盘":150}
json.dumps(computer)
# 反序列化
json.loads(json_obj)
```

之前常用三种序列化方式： Django内置Serializers模块

Serializers是Django内置的一个序列化器，可直接将Python对象转为JSON格式，但不支持反序列化。

```
from django.core import serializers
obj = User.objects.all()
data = serializers.serialize('json', obj)
```

之前常用三种序列化方式： Django内置JsonResponse模块

JsonResponse模块自动将Python对象转为JSON对象并响应。

DRF序列化器

DRF中有一个serializers模块专门负责数据序列化，DRF提供的方案更先进、更高级别的序列化方案。

序列化器支持三种类型：

- **Serializer：**对Model（数据模型）进行序列化，需自定义字段映射。
- **ModelSerializer：**对Model进行序列化，会自动生成字段和验证规则，默认还包含简单的create()和update()方法。
- **HyperlinkedModelSerializer：**与ModelSerializer类似，只不过使用超链接来表示关系而不是主键ID。

DRF序列化器：Serializer

获取所有用户（查），接口地址：/myapp_api/api/user

1、定义序列化器

myapp_api/serializers.py

serializers.py

```python
from rest_framework import serializers

class UserSerializer(serializers.Serializer):
    # 这里字段必须与使用的模型字段对应
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=30)
    city = serializers.CharField(max_length=30)
    sex = serializers.CharField(max_length=30)
    age = serializers.IntegerField()
```

2、视图里使用序列化器

myapp_api/views.py

serializers.py views.py

```python
from rest_framework.views import APIView
from myapp_api.models import User
from .serializers import UserSerializer
from rest_framework.response import Response

class UserView(APIView):
    def get(self, request):
        # 获取所有用户
        queryset = User.objects.all()
        # 调用序列化器将queryset对象转为json
        user_ser = UserSerializer(queryset, many=True)  # 如果序列化多条数据，需要指定many=True
        # 从.data属性获取序列化结果
        return Response(user_ser.data)
```

3、定义路由

myapp_api/urls.py

serializers.py views.py urls.py

```python
from django.urls import path, re_path
from myapp_api import views

urlpatterns = [
    re_path('^api/user/$', views.UserView.as_view())
]
```

DRF序列化器：Serializer

获取单个用户（查），接口地址：/myapp_api/api/user/2

1、定义视图

views.py × urls.py ×

```python
from rest_framework.views import APIView
from myapp_api.models import User
from .serializers import UserSerializer
from rest_framework.response import Response

class UserView(APIView):
    def get(self, request, pk=None):
        if pk:
            # 获取单用户
            user_obj = User.objects.get(id=pk)
            user_ser = UserSerializer(user_obj)
        else:
            # 获取所有用户
            queryset = User.objects.all()
            # 调用序列化器将queryset对象转为json
            user_ser = UserSerializer(queryset, many=True)  # 如果序列化多条数据，需要指定many=True
        # 从.data属性获取序列化结果
        result = {'code': 200, 'msg': '获取用户成功', 'data': user_ser.data}
        return Response(result)
```

2、定义路由

views.py × urls.py ×

```python
from django.urls import path, re_path
from myapp_api import views

urlpatterns = [
    re_path('^api/user/$', views.UserView.as_view()),
    re_path('^api/user/(?P<pk>\d+)/$', views.UserView.as_view()),
]
```

DRF序列化器：Serializer

创建用户，接口地址：/myapp_api/api/user/

定义视图：增加post方法接收数据

```python
def post(self, request):
    # 调用序列化器将提交的数据进行反序列化
    user_ser = UserSerializer(data=request.data)
    # 获取反序列化验证是否通过
    if user_ser.is_valid():
        # 保存到数据库
        user_ser.save()
        msg = '创建用户成功'
        code = 200
    else:
        msg = '创建用户失败，数据格式不正确！'
        code = 400
    result = {'code': code, 'msg': msg}
    return Response(result)
```

定义create()方法：

views.py | serializers.py | urls.py

```python
from rest_framework import serializers

class UserSerializer(serializers.Serializer):
    # 这里字段必须与使用的模型字段对应
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=30)
    city = serializers.CharField(max_length=30)
    sex = serializers.CharField(max_length=30)
    age = serializers.IntegerField()

    def create(self, validated_data):  # validated_data 是提交的JSON数据
        return User.objects.create(**validated_data)
```

DRF序列化器：Serializer

更新用户，接口地址：/myapp_api/api/user/2

定义视图：增加put方法接收数据

```python
def put(self, request, pk=None):
    user_obj = User.objects.get(id=pk)
    # 调用序列化器，传入已有对象和提交的数据
    user_ser = UserSerializer(instance=user_obj, data=request.data)
    if user_ser.is_valid():
        user_ser.save()
        msg = '更新用户成功'
        code = 200
    else:
        msg = '更新用户失败，数据格式不正确！'
        code = 400
    result = {'code': code, 'msg': msg}
    return Response(result)
```

定义update()方法：

views.py | serializers.py | urls.py

```python
from rest_framework import serializers

class UserSerializer(serializers.Serializer):
    # 这里字段必须与使用的模型字段对应
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=30)
    city = serializers.CharField(max_length=30)
    sex = serializers.CharField(max_length=30)
    age = serializers.IntegerField()

    def create(self, validated_data):  # validated_data 提交的JSON数据
        return User.objects.create(**validated_data)

    def update(self, instance, validated_data):  # instance 当前操作的用户对象，validated_data 提交的JSON数据
        instance.name = validated_data.get('name')
        instance.city = validated_data.get('city')
        instance.sex = validated_data.get('sex')
        instance.age = validated_data.get('age')
        instance.save()
        return instance
```

DRF序列化器：Serializer

删除用户，接口地址：/myapp_api/api/user/2

```
def delete(self, request, pk=None):
    user_obj = User.objects.get(id=pk)
    try:
        # 直接删除，不再通过序列化器
        user_obj.delete()
        msg = '用户删除成功'
        code = 200
    except Exception as e:
        msg = '用户删除失败 %s' %e
        code = 400
    result = {'code': code, 'msg': msg}
    return Response(result)
```

DRF序列化器：小结

序列化器工作流程：

序列化（读数据）：视图里通过ORM从数据库获取数据查询集对象 -> 数据传入序列化器 -> 序列化器将数据进行序列化 -> 调用序列化器的.data获取数据 -> 响应返回前端

反序列化（写数据）：视图获取前端提交的数据 -> 数据传入序列化器 -> 调用序列化器的.is_valid方法进行效验 -> 调用序列化器的.save()方法保存数据

序列化器常用方法与属性：

- serializer.is_valid()：调用序列化器验证是否通过，传入raise_exception=True可以在验证失败时由DRF响应400异常。
- serializer.errors：获取反序列化器验证的错误信息
- serializer.data：获取序列化器返回的数据
- serializer.save()：将验证通过的数据保存到数据库（ORM操作）

DRF序列化器：序列化器参数

序列化器常用参数：

名称	作用
max_length	最大长度，适用于字符串、列表、文件
min_length	最小长度，适用于字符串、列表、文件
allow_blank	是否允许为空
trim_whitespace	是否截断空白字符
max_value	最大值，适用于数值
min_value	最小值，适用于数值

通用参数：

名称	作用
read_only	说明该字段仅用于序列化，默认False，若设置为True，反序列化可不传。
write_only	该字段仅用于反序列化，默认False
required	该字段在反序列化时必须输入，默认True
default	反序列化时使用的默认值
allow_null	是否允许为NULL，默认False
validators	指定自定义的验证器
error_message	包含错误编号与错误信息的字典

```python
class UserSerializer(serializers.Serializer):
    # 这里字段必须与使用的模型字段对应
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(max_length=30,
                                 error_messages={  # 设置每种错误的提示
                                     "blank": "请输入姓名",
                                     "required": "该字段必填",
                                     "max_length": "字符长度不超过30",
                                 })
    city = serializers.CharField(max_length=10,
                                 error_messages={
                                     "blank": "请输入城市",
                                     "required": "该字段必填",
                                     "max_length": "字符长度不超过10",
                                 })
    sex = serializers.CharField(max_length=10,
                                error_messages={
                                    "blank": "请输入性别",
                                    "required": "该字段必填",
                                    "max_length": "字符长度不超过10",
                                })
    age = serializers.IntegerField(min_value=16, max_value=100,
                                   error_messages={
                                       "blank": "请输入年龄",
                                       "required": "该字段必填",
                                       "min_value": "年龄不低于16岁",
                                       "max_value": "年龄不高于100岁",
                                   })
```

示例

DRF序列化器：扩展验证规则

如果常用参数无法满足验证要求时，可通过钩子方法扩展验证规则。

局部钩子：validate_字段名(self, 字段值)

全局钩子：validate(self, 所有校验的数据字典)

```python
# 局部钩子
# 姓名不能包含数字
def validate_name(self, attrs):  # attrs是该字段的值
    from re import findall
    if findall('\d+', attrs):
        raise serializers.ValidationError("姓名不能包含数字！")
    else:
        return attrs

# 全局钩子
def validate(self, attrs):  # attrs是所有字段组成的字典
    sex = attrs.get('sex')
    if sex != "男" and sex != "女":
        raise serializers.ValidationError("性别不能是人妖！")
    else:
        return attrs
```

DRF序列化器：扩展验证规则

如果钩子无法满足需要，可以自定义验证器，更灵活。

在序列化类外面定义验证器，使用validators参数指定验证器。

```
# 自定义验证器
def check_name(data):
    if data.startswith('x'):
        raise serializers.ValidationError('姓名不能以x开头！')
    else:
        return data

class UserSerializer(serializers.Serializer):
    # 这里字段必须与使用的模型字段对应
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(max_length=30, validators=[check_name],
                                 error_messages={  # 设置每种错误的提示
                                     "blank": "请输入姓名",
                                     "required": "该字段必填",
                                     "max_length": "字符长度不超过30",
                                 })
```

DRF序列化器：ModelSerializer

ModelSerializer 类型不需要自定义字段映射和定义create、update方法，使用起来方便很多！

views.py serializers.py

```python
from rest_framework import serializers
from myapp_api.models import User

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User  # 指定数据模型
        fields = '__all__'  # 显示所有字段

```

DRF序列化器：ModelSerializer

Meta类常用属性：

- fields：显示所有或指定字段
- exclude：排除某个字段，元组格式，不能与fields同时用
- read_only_fields：只读字段，即只用于序列化，不支持修改
- extra_kwargs：添加或修改原有的字段参数，字典格式
- depth：根据关联的数据递归显示，一般是多表

views.py serializers.py

```python
from rest_framework import serializers
from myapp_api.models import User

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User  # 指定数据模型
        fields = '__all__'  # 显示所有字段
        # exclude = ('id', )
        read_only_fields = ('id', )
        extra_kwargs = {
            'name': {'max_length': 10, 'required': True},
            'city': {'max_length': 10, 'required': True},
            'sex': {'max_length': 10, 'required': True},
            'age': {'min_value': 16,  'max_value': 100, 'required': True}
        }
```

示例

DRF序列化器：HyperModelSerializer

与MedelSerializer使用方法一样。只不过它使用超链接来表示关系而不是主键ID。

示例：基于前面用户管理修改

```
# 更改序列化器
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = "__all__ “

# 更改视图
user_ser = UserSerializer(queryset, many=True, context={'request': request})

# 更改路由
re_path('^api/user/$', views.UserView.as_view(), name="user-detail"),
re_path('^api/user/(?P<pk>\d+)/$', views.UserView.as_view(), name="user-detail"),
```

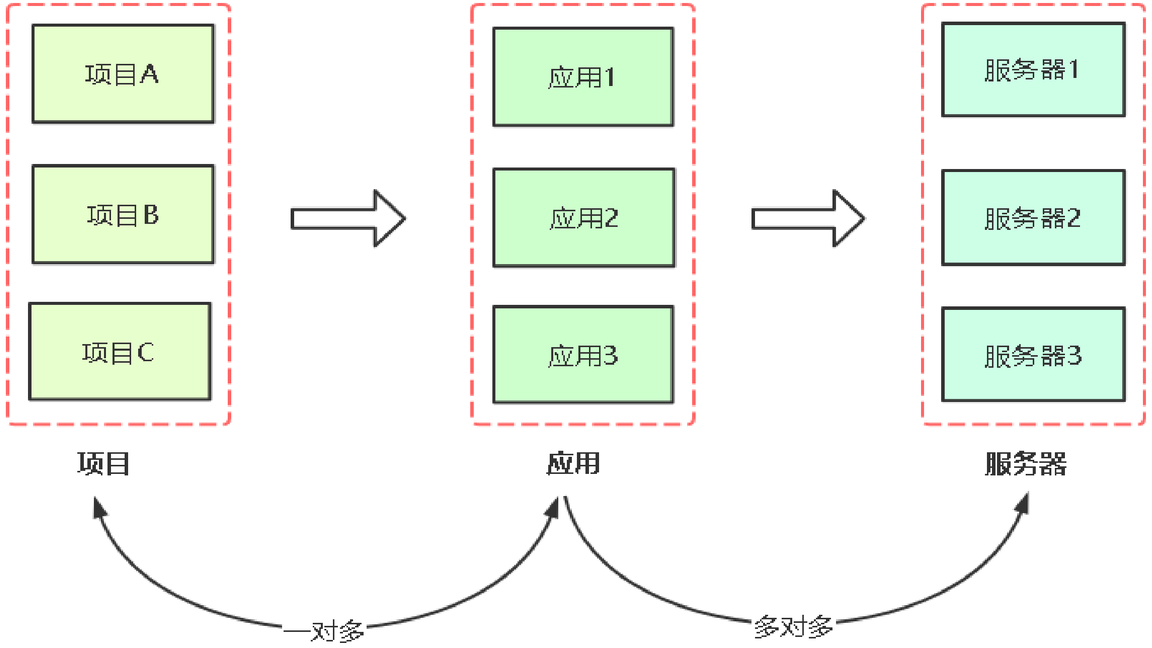
示例

DRF序列化器：关联表显示

例如：应用发布系统项目涉及表

一对多：一个项目有多个应用，一个应用只能属于一个项目

多对多：一个应用部署到多台服务器，一个服务器部署多个应用



DRF序列化器：关联表显示

1、定义数据模型

```
class Project(models.Model):
    name = models.CharField(max_length=30)

class App(models.Model):
    name = models.CharField(max_length=30)
    project = models.ForeignKey(Project, on_delete=models.CASCADE) # 一对多

class Server(models.Model):
    hostname = models.CharField(max_length=30)
    ip = models.GenericIPAddressField()
    app = models.ManyToManyField(App)  # 多对多
```

2、定义序列化器

```
class ProjectSerializer(serializers.ModelSerializer):
    class Meta:
        model = Project
        fields = "__all__"

class AppSerializer(serializers.ModelSerializer):
    class Meta:
        model = App
        fields = "__all__"

class ServerSerializer(serializers.ModelSerializer):
    class Meta:
        model = Server
        fields = "__all__"
```

3、定义路由

```
re_path('^api/project/$', views.ProjectView.as_view()),
re_path('^api/app/$', views.AppView.as_view()),
re_path('^api/server/$', views.ServerView.as_view()),
```

DRF序列化器：关联表显示

3、定义视图

```python
class ProjectView(APIView):
    def get(self, request):
        # 获取所有项目
        queryset = Project.objects.all()
        # 调用序列化器将queryset对象转为json，如果序列化多条数据，需要指定many=True
        project_ser = ProjectSerializer(queryset, many=True)
        # 从.data属性获取序列化结果
        result = {'code': 200, 'msg': '获取成功', 'data': project_ser.data}
        return Response(result)
    def post(self, request):
        project_ser = ProjectSerializer(data=request.data)
        project_ser.is_valid(raise_exception=True)
        project_ser.save()
        return Response(data=project_ser.data)
class AppView(APIView):
    def get(self, request):
        queryset = App.objects.all()
        app_ser = AppSerializer(queryset, many=True)
        result = {'code': 200, 'msg': '获取成功', 'data': app_ser.data}
        return Response(result)
    def post(self, request):
        app_ser = AppSerializer(data=request.data)
        app_ser.is_valid(raise_exception=True)
        app_ser.save()
        return Response(data=app_ser.data)
class ServerView(APIView):
    def get(self, request):
        queryset = Server.objects.all()
        server_ser = ServerSerializer(queryset, many=True)
        result = {'code': 200, 'msg': '获取成功', 'data': server_ser.data}
        return Response(result)
    def post(self, request):
        server_ser = ServerSerializer(data=request.data)
        server_ser.is_valid(raise_exception=True)
        server_ser.save()
        return Response(data=server_ser.data)
```

4、定义路由

```
re_path('^api/project/$', views.ProjectView.as_view()),
re_path('^api/app/$', views.AppView.as_view()),
re_path('^api/server/$', views.ServerView.as_view()),
```

5、添加测试数据

创建项目：

```
from myapp_api.models import Project, App, Server
Project.objects.create(name="电商")
Project.objects.create(name="教育")
```

创建应用并指定项目：

```
project_obj = Project.objects.get(name="电商")
App.objects.create(name="portal", project=project_obj)
App.objects.create(name="gateway", project=project_obj)
```

创建服务器：

```
Server.objects.create(hostname="test1", ip="192.168.31.10")
Server.objects.create(hostname="test2", ip="192.168.31.11")
```

将应用部署到服务器：

```
app_obj = App.objects.get(name="portal")
server_obj = Server.objects.get(hostname="test1")
server_obj.app.add(app_obj)
```

DRF序列化器：关联表显示

序列化器返回是当前模型中的字段，如果字段是外键时，返回的是外键对应id，如图所示，如果想要显示外键对应的详细信息如何做呢？

有两种方法：

- 定义字段为外键对应序列化类：例如project=ProjectSerializer(read_only=True)，这种适合针对某个外键字段。
- 序列化类中Meta类启用depth：深度获取关联表数据，这种所有外键都会显示出来。

DRF序列化器：关联表显示

一对多

```
# 方法1
class AppSerializer(serializers.ModelSerializer):
    project = ProjectSerializer(read_only=True) # 一对多，返回关联的项目详情
    class Meta:
        model = App
        fields = "__all__"

# 方法2
class AppSerializer(serializers.ModelSerializer):
    class Meta:
        model = App
        fields = "__all__"
        depth = 1
```

DRF序列化器：关联表显示

多对多

```
# 方法1
class ServerSerializer(serializers.ModelSerializer):
    app = AppSerializer(many=True)  # 多对多，返回关联的应用详情
    class Meta:
        model = Server
        fields = "__all__"

# 方法2
class ServerSerializer(serializers.ModelSerializer):
    class Meta:
        model = Server
        fields = "__all__"
        depth = 1
```

DRF序列化器：SerializerMethodField

DRF序列化器默认仅返回数据模型中已存在资源，如果想新增返回字段或者二次处理，该如何操作呢？用SerializerMethodFiled

示例：给项目API增加一个字段，这个字段数据可从别的表中获取

```python
class ProjectSerializer(serializers.ModelSerializer):
    app_count = serializers.SerializerMethodField()

    class Meta:
        model = Project
        fields = "__all__"

    # get_字段名
    def get_app_count(self, obj):
        return len(obj.app_set.all())
```

DRF序列化器：改变序列化和反序列化的行为

可以通过重写下面两个方法改变序列化和反序列化的行为：

- to_internal_value()：处理反序列化的输入数据，自动转换Python对象，方便处理。
- to_representation()：处理序列化数据的输出。

DRF序列化器：改变序列化和反序列化的行为

示例：如果提交API的数据与序列化器要求的格式不符合，序列化器就会出现错误。

这时就可以重写to_internal_value()方法只提取出我们需要的数据。

```
{
   "project_data": {
     "name": "测试",
     "describe": "测试。。",
   },
   "extra_info": {
       "msg": "hello world"
   }
}
```

提交的数据

```python
def to_internal_value(self, data):
    # data是未验证的数据
    # 提取数据
    project_data = data['project_data']
    return super().to_internal_value(project_data)
```

重写方法

DRF序列化器：改变序列化和反序列化的行为

示例：希望给返回的数据添加一个统计应用数量的字段

```python
def to_representation(self, instance):
    # 调用父类获取当前序列化数据，instance代表每个对象实例
    data = super().to_representation(instance)
    data['app_count'] = len(instance.app_set.all())
    return data
```

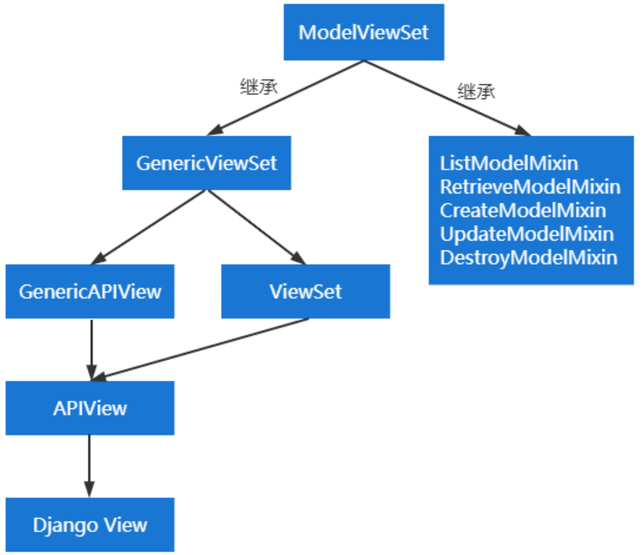
Django DRF视图

- DRF类视图介绍
- APIView类
- Request与Response
- GenericAPIView类
- ViewSet类
- ModelViewSet类

DRF类视图介绍

在DRF框架中提供了众多的通用视图基类与扩展类，以简化视图的编写。

- View：Django默认的视图基类，负责将视图连接到URL，HTTP请求方法的基本调度，之前写类视图一般都用这个。
- APIView：DRF提供的所有视图的基类，继承View并扩展，具备了身份认证、权限检查、流量控制等功能。
- GenericAPIView：对APIView更高层次的封装，例如增加分页、过滤器
- GenericViewSet：继承GenericAPIView和ViewSet
- ViewSet：继承APIView，并结合router自动映射路由
- ModelViewSet：继承GenericAPIView和五个扩展类，封装好各种请求，更加完善，业务逻辑基本不用自己写了。



APIView类

APIView：DRF提供的所有视图的基类，继承View并扩展，具备了身份认证、权限检查、流量控制等功能。

```
from rest_framework.views import APIView

class UserView(APIView):
    def get(self, request, pk=None):...
    def post(self, request):...
    def put(self, request, pk=None):...
    def delete(self, request, pk=None):...
```

Request与Response

DRF传入视图的request对象不再是Django默认的HttpRequest对象，而是基于HttpRequest类扩展后的Request类的对象。
Request对象的数据是自动根据前端发送的数据统一解析数据格式。

常用属性：

- request.data：返回POST提交的数据，与request.POST类似
- request.query_params：返回GET URL参数，与request.GET类似

Request与Response

DRF提供了一个响应类Reponse，响应的数据会自动转换符合前端的JSON数据格式。

导入：

```
from rest_framework.response import Response
```

格式：

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

- data：响应序列化处理后的数据，传递python对象
- status：状态码，默认200
- template_name：模板名称
- headers：用于响应头信息的字典
- content_type：响应数据的类型

Request与Response

使用方法：return Reponse(data=data, status=status.HTTP_404_NOT_FOUND)

为了方便设置状态码，rest_framework.status模块提供了所有HTTP状态码，以下是一些常用的：

- HTTP_200_OK：请求成功
- HTTP_301_MOVED_PERMANENTLY：永久重定向
- HTTP_302_FOUND：临时重定向
- HTTP_304_NOT_MODIFIED：请求的资源未修改
- HTTP_403_FORBIDDEN：没有权限访问
- HTTP_404_NOT_FOUND：页面没有发现
- HTTP_500_INTERNAL_SERVER_ERROR：服务器内部错误
- HTTP_502_BAD_GATEWAY：网关错误
- HTTP_503_SERVICE_UNAVAILABLE：服务器不可达
- HTTP_504_GATEWAY_TIMEOUT：网关超时

GenericAPIView类

GenericAPIView对APIView更高层次的封装，实现以下功能：

- 增加queryset属性，指定操作的数据，不用再将数据传给序列化器，会自动实现。
- 增加serializer_class属性，直接指定使用的序列化器
- 增加过滤器属性：filter_backends
- 增加分页属性：pagination_class
- 增加lookup_field属性和实现get_object()方法：用于获取单条数据，可自定义默认分组名（pk）

GenericAPIView类

```python
from rest_framework.generics import GenericAPIView
class UserView(GenericAPIView):
    queryset = User.objects.all()  # 指定操作的数据
    serializer_class = UserSerializer  # 指定序列化器

    def get(self, request, pk=None):
        if pk:
            user_obj = self.get_object()  # 从类方法调用指定数据（默认根据pk）
            user_ser = self.get_serializer(instance=user_obj)  # 从类方法调用序列化器
        else:
            queryset = self.get_queryset()  # 从类方法调用数据
            user_ser = self.get_serializer(instance=queryset, many=True)  # 从类方法调用序列化器
        result = {'code': 200, 'msg': '获取用户成功', 'data': user_ser.data}
        return Response(result)

    def put(self, request, pk=None):
        user_obj = self.get_object()
        user_ser = self.get_serializer(instance=user_obj, data=request.data)
        user_ser.is_valid(raise_exception=True)
        user_ser.save()
        result = {'code': 200, 'msg': '更新用户成功'}
        return Response(result)
```

ViewSet类

GenericAPIView已经完成了许多功能，但会有一个问题，获取所有用户列表和单个用户需要分别定义两个视图和URL路由，使用ViewSet可以很好解决这个问题，并且实现了路由自动映射。

ViewSet视图集不再实现get()、post()等方法，而是实现以下请求方法动作：

- list()：获取所有数据
- retrieve()：获取单个数据
- create()：创建数据
- update()：更新数据
- destory()：删除数据

ViewSet类

```
re_path('^api/user/$', views.UserViewSet.as_view({'get': 'list',
                                                  'post': 'create'
                                                  })),
re_path('^api/user/(?P<pk>\d+)/$', views.UserViewSet.as_view({'get': 'retrieve',
                                                              'put': 'update',
                                                              'delete': 'destory'
                                                              })),
```

URL路由

```
from rest_framework.viewsets import ViewSet
class UserViewSet(ViewSet):
    # 获取单个数据
    def retrieve(self, request, pk=None):...
    # 获取所有数据
    def list(self, request):...
    # 创建
    def create(self, request):...
    # 更新
    def update(self, request, pk=None):...
    # 删除
    def destory(self, request, pk=None):...
```

视图：逻辑定义与APIView一样

ViewSet类

在路由这块定义与之前方式一样，每个API接口都要写一条URL路由，但实际上我们用ViewSet后，就不用自己设计URL路由及绑定HTTP方法了，会自动处理URL路由映射。

```
from rest_framework import routers
from django.urls import include

# 自动生成URL路由
router = routers.DefaultRouter()
router.register(r'user', views.UserViewSet, basename='user') # 注册到路由，格式：路径，视图，名称

urlpatterns += [
    path('api/', include(router.urls))  # 将路由加入到django URL里
]
```

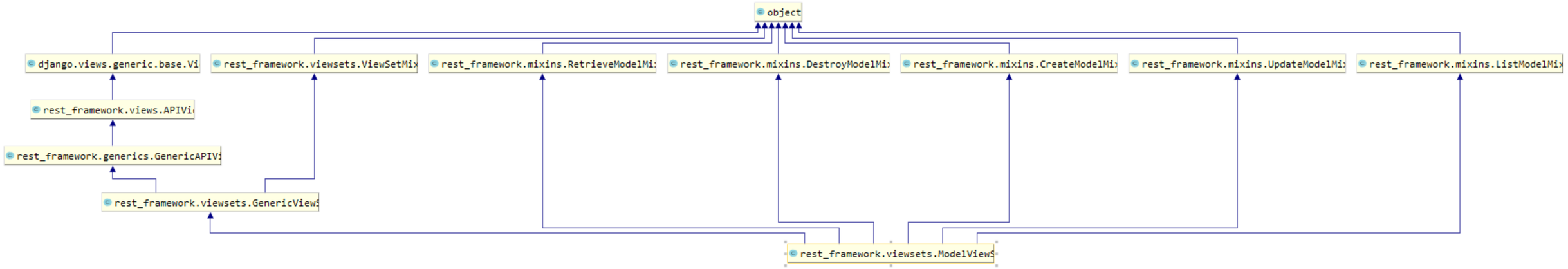
然后访问 http://ip/myapp_api/api 就可以看到自动生成的URL路由。

ModelViewSet类

ModelViewSet继承GenericAPIView和五个扩展类，封装好各种请求，更加完善，业务逻辑基本不用自己写了，只需要指定serializer_class和queryset，就可以直接进行增删改查。

```python
from rest_framework.viewsets import ModelViewSet
class UserViewSet(ModelViewSet):
    queryset = User.objects.all()  # 指定操作的数据
    serializer_class = UserSerializer  # 指定序列化器
```

ModelViewSet类



动作	类名	HTTP方法	说明	URL示例
retrieve	mixins.RetrieveModelMixin	GET	获取单条数据，需带pk	http://ip/api/user/123
list	mixins.ListModelMixin	GET	获取多条数据	http://ip/api/user/
create	mixins.CreateModelMixin	POST	创建数据	http://ip/api/user/
update	mixins.UpdateModelMixin	PUT	更新数据，需带pk	http://ip/api/user/123
destroy	mixins.DestroyModelMixin	DELETE	删除数据，需带pk	http://ip/api/user/123

ModelViewSet类

由于ModelViewSet有较高的抽象，实现自动增删改查功能。对于增、改在很多场景无法满足需求，这就需要重写对应方法了。

示例：重写create()方法，修改数据和响应内容格式

```
def create(self, request, *args, **kwargs):
    print(request.data)
    request.data['age'] += 1
    serializer = self.get_serializer(data=request.data)
    serializer.is_valid(raise_exception=True)
    self.perform_create(serializer)
    result = {'code': 200, 'msg': '创建用户成功'}
    return Response(result)
```

DRF 常用功能

- 主流认证方式
- DRF认证
- 限流
- 过滤
- 搜索和排序
- 分页
- 自动生成接口文档

主流认证方式

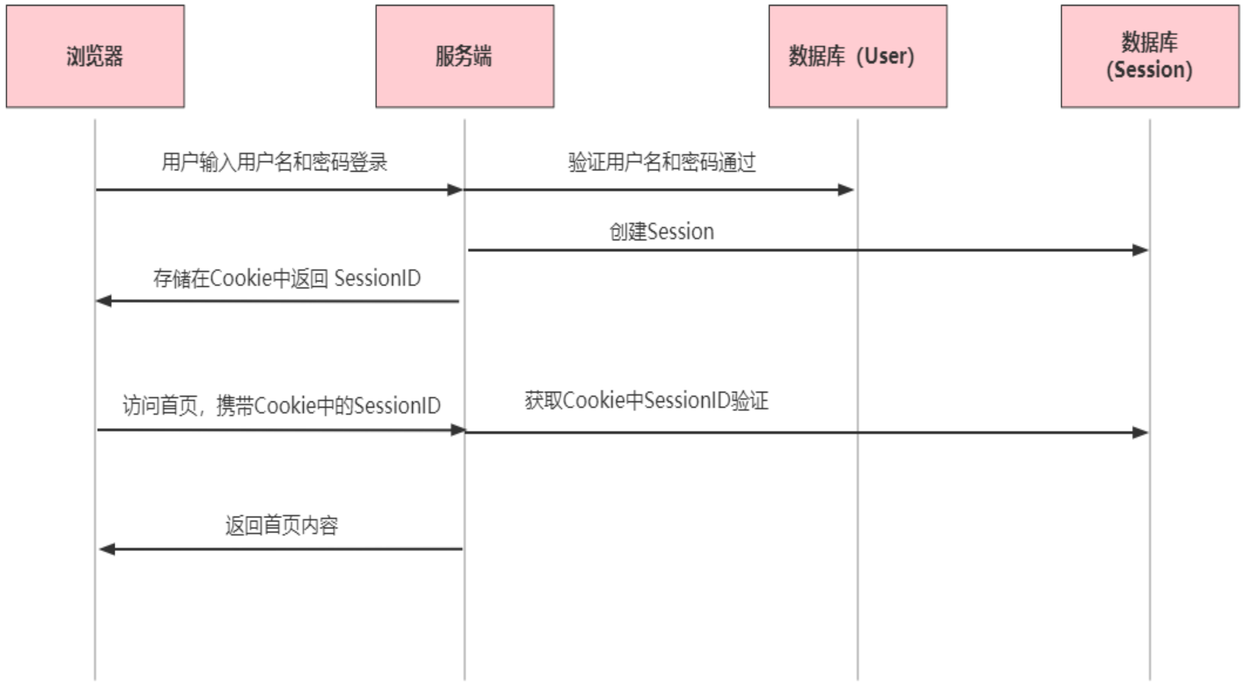
主流认证方式：

- Session
- Token
- JWT

主流认证方式：Session认证

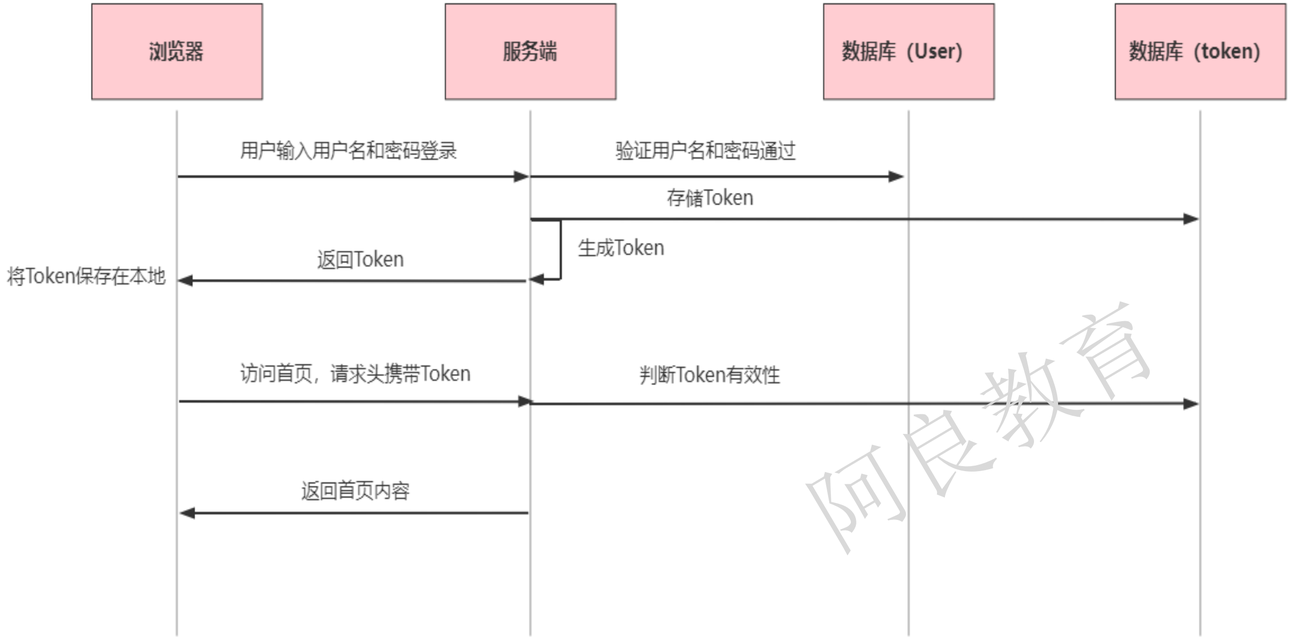
HTTP是一个无状态的协议，每次访问都是新的，早期主要用于浏览网页，随着时代发展，像在线购物网站的兴起，就面临着记录哪些人登录系统，哪些人购物车里放了商品。也就是必须每个人区分开，所以就有了用户名来标识，但每次访问页面都要登录，非常麻烦，这就有了会话保持。

Cookie+Session就是实现会话保持的技术。



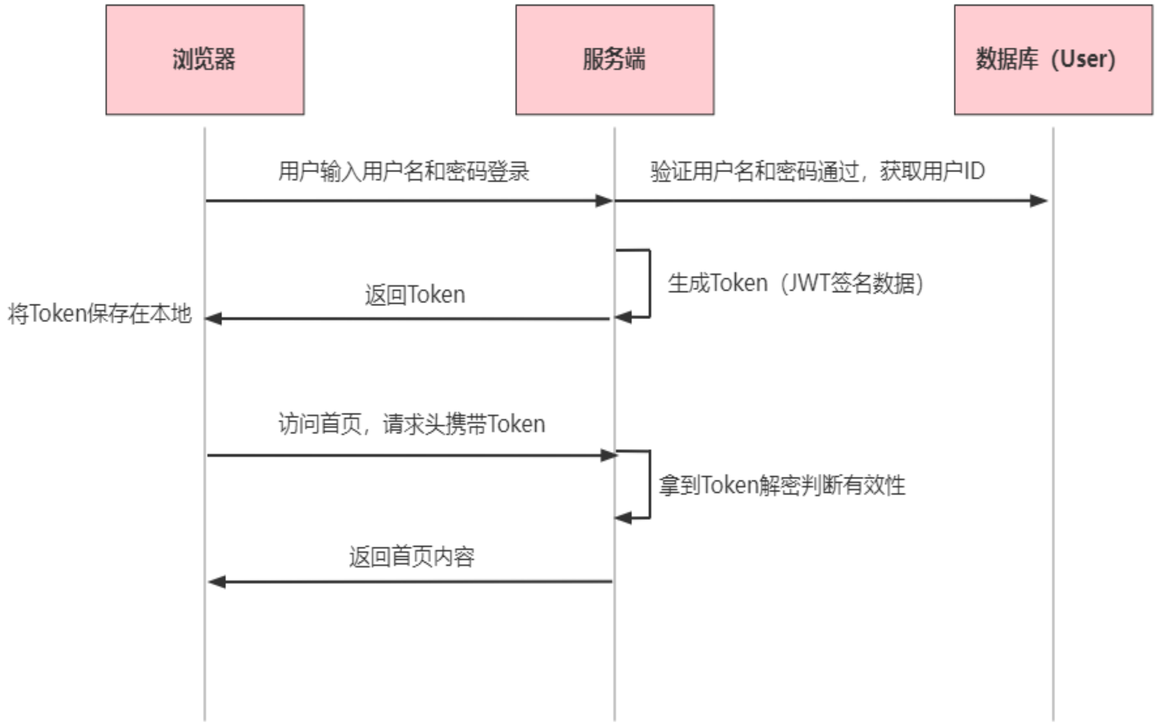
主流认证方式：Token认证

Cookie+Session通常在浏览器作为客户端的情况下比较通用，随着前后端分离开发模式的普及，会涉及到多端（PC、APP、Pad），特别是手机端，支持Cookie不友好，并且Cookie不支持跨域，因此基于这些局限性，Token逐渐主流。



主流认证方式：JWT认证

与普通Token一样，都是访问资源的令牌，区别是普通Token服务端验证token信息要查询数据库验证，JWT验证token信息不用查询数据库，只需要在服务端使用密钥效验。



DRF认证：DRF认证与权限

目前DRF可任意访问，没有任何限制，是不符合生产环境标准的，因此接下来学习认证实现访问控制。

DRF支持四种认证方式：

- BasicAuthentication：基于用户名和密码的认证，适用于测试
- SessionAuthentication：基于Session的认证
- TokenAuthentication：基于Token的认证
- RemoteUserAuthentication：基于远程用户的认证

DRF支持权限：

- IsAuthenticated：只有登录用户才能访问所有API
- AllowAny：允许所有用户
- IsAdminUser：仅管理员用户
- IsAuthenticatedOrReadOnly：登录的用户可以读写API，未登录用户只读

DRF认证：Session认证

所有视图（全局）启用认证：

```
# DRF配置
REST_FRAMEWORK = {
    # 认证
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.SessionAuthentication',
    ],
    # 权限
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
}
```

视图级别启用认证：

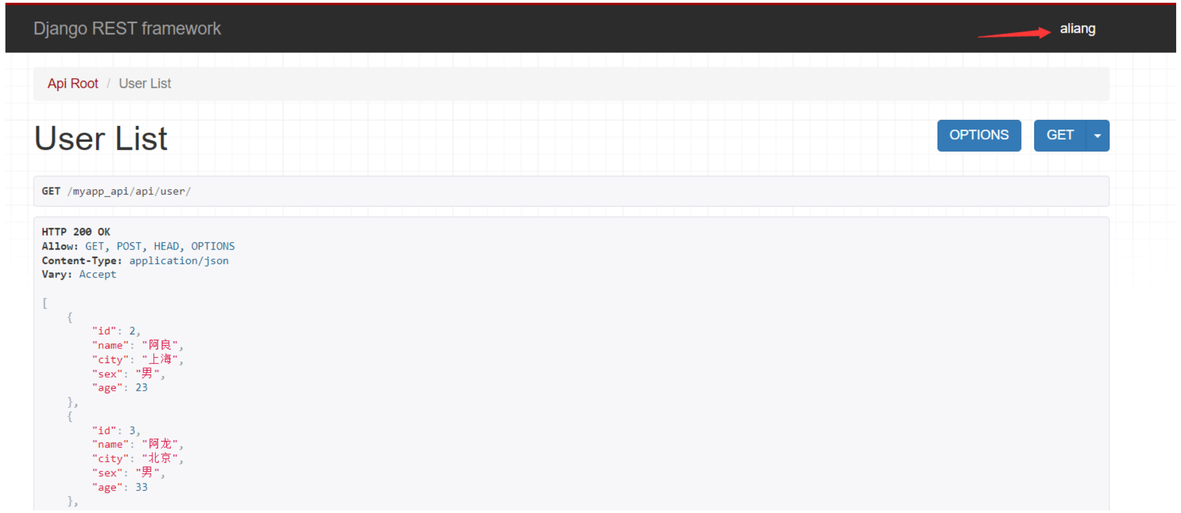
```
from rest_framework.viewsets import ModelViewSet
from rest_framework.authentication import SessionAuthentication
from rest_framework.permissions import IsAuthenticated

class UserViewSet(ModelViewSet):
    queryset = User.objects.all()  # 指定操作的数据
    serializer_class = UserSerializer  # 指定序列化器
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticated]
```

DRF认证：Session认证

由于Django默认提供Session存储机制，可直接通过登录内置管理后台进行验证。

当登录管理后台后，就有权限访问了。



DRF认证：Token认证

1、安装APP

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'myapp',
    'myapp_api',
    'rest_framework.authtoken'
]
```

2、启用Token认证

```
# DRF配置
REST_FRAMEWORK = {
    # 认证
    'DEFAULT_AUTHENTICATION_CLASSES': [
        # 'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.TokenAuthentication',
    ],
    # 权限
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
}
```

3、生成数据库表

```
python manage.py migrate
```

4、配置Token认证接口URL

```
from rest_framework.authtoken import views
urlpatterns += [
    re_path('^api-token-auth/', views.obtain_auth_token)
]
```

DRF认证：Token认证

使用用户名和密码登录测试，正常会返回token字符串

后面就可以直接token访问：把token字符串放到header

DRF认证：Token认证

默认的obtain_auth_token视图返回的数据是比较简单的，只有token一项，如果想返回更多的信息，例如用户名，可以重写ObtainAuthToken类的方法实现：

myapp_api/obtain_auth_token.py

```python
from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.authtoken.models import Token
from rest_framework.response import Response

class CustomAuthToken(ObtainAuthToken):

    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data,
                                           context={'request': request})
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']
        token, created = Token.objects.get_or_create(user=user)
        return Response({
            'token': token.key,
            'username': user.username
        })
```

URL指定自定义模块：

```python
urlpatterns += [
    path('api/', include(router.urls))  # 将路由加入到django URL里
]

from myapp_api.obtain_auth_token import CustomAuthToken
urlpatterns += [
    re_path('^api-token-auth/', CustomAuthToken.as_view())
]
```

限流

可以对接口访问的频率进行限制，以减轻服务器压力。

应用场景：投票、购买数量等

```python
# DRF配置
REST_FRAMEWORK = {
    # 认证
    'DEFAULT_AUTHENTICATION_CLASSES': [
        # 'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.TokenAuthentication',
    ],
    # 权限
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
    # 限流：范围
    'DEFAULT_THROTTLE_CLASSES': (
      'rest_framework.throttling.AnonRateThrottle',# 未登录用户
      'rest_framework.throttling.UserRateThrottle'  # 已登录用户
    ),
    # 限流：设置访问频率
    'DEFAULT_THROTTLE_RATES': {
      # 周期：second, minute, hour，day
      'anon': '3/minute',  # 针对未登录用户进行限制，每分钟最多访问3次，基于IP区分用户
      'user': '5/minute'   # 针对登录用户进行限制，每分钟最多访问5次，基于用户ID区分
    }
}
```

过滤

对于列表数据可能需要根据字段进行过滤，我们可以通过添加django-fitlter扩展来增强支持。

文档：https://www.django-rest-framework.org/api-guide/filtering/

安装：

```
pip install django-filter
```

添加APP：

```
INSTALLED_APPS = [
    ...
    'django_filters'
]
```

添加DRF配置：

```
REST_FRAMEWORK = {
    # 过滤
    'DEFAULT_FILTER_BACKENDS': ('django_filters.rest_framework.DjangoFilterBackend',)
}
```

过滤

在视图中指定过滤的字段：

```
class UserViewSet(ModelViewSet):
    queryset = User.objects.all()  # 指定操作的数据
    serializer_class = UserSerializer  # 指定序列化器
    filter_fields = ('name',)
```

测试：http://127.0.0.1:8001/myapp_api/api/user?name=阿良

搜索和排序

DRF提供过滤器帮助我们快速对字段进行搜索和排序。

```
from rest_framework import filters

class UserViewSet(ModelViewSet):
    queryset = User.objects.all()  # 指定操作的数据
    serializer_class = UserSerializer  # 指定序列化器
    filter_fields = ('name',)

    filter_backends = [filters.SearchFilter, filters.OrderingFilter]  # 指定过滤器
    search_fields = ('name', )  # 指定可以搜索的字段
```

搜索测试：http://127.0.0.1:8001/myapp_api/api/user?search=良

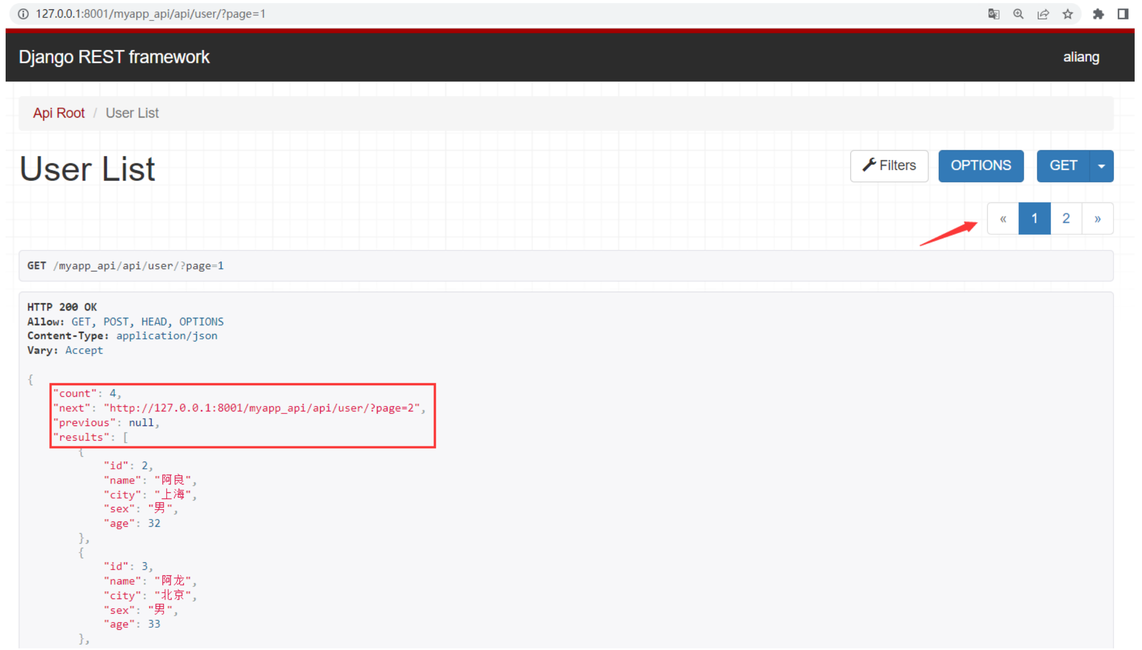
排序测试：http://127.0.0.1:8001/myapp_api/api/user?ordering=id

注：默认是正序排列，字段前面加横杠（例如-id）表示倒序排列

分页

分页是数据表格必备的功能，可以在前端实现，也可以在后端实现，为了避免响应数据过大，造成前端压力，一般在后端实现。

```
REST_FRAMEWORK = {
    # 分页
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 3  # 每页数目
}
```



分页

默认分页器灵活度不高，例如不能动态传递每页条数，可以通过重写PageNumberPagination类属性改变默认配置。

myapp_api/pagination.py

```python
from rest_framework.pagination import PageNumberPagination

class MyPagination(PageNumberPagination):
    page_size = 10 # 默认每页显示多少条数据
    page_query_param = 'page_num' # 指定URL查询第几页的关键字名称，默认为"page"
    page_size_query_param = 'page_size' # 指定URL查询（每页显示多少条数据）关键字名称，默认为None
    max_page_size = 50 # 每页最多显示多少条数据
```

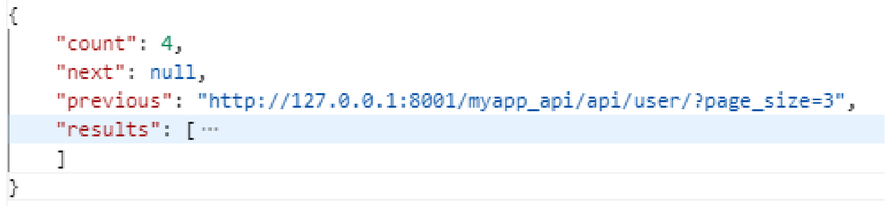
DRF配置指定模块路径：

```python
# 分页
# 'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
'DEFAULT_PAGINATION_CLASS': 'myapp_api.pagination.MyPagination',
# 'PAGE_SIZE': 3  # 每页数目，必须定义，否则分页无效
```

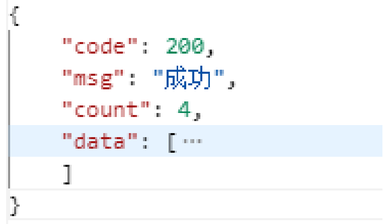
测试：http://127.0.0.1:8001/myapp_api/api/user?page_num=2&page_size=3

分页

默认返回的是一个固定格式JSON字符串，但这个格式与我们平时用的格式不太一样，所以希望把这个返回修改一下，可通过重写PageNumberPagination类实现。



修改前



修改后

```python
from rest_framework.pagination import PageNumberPagination
from collections import OrderedDict
from rest_framework.response import Response

class MyPagination(PageNumberPagination):
    page_size = 10 # 默认每页显示多少条数据
    page_query_param = 'page_num' # 指定URL查询第几页的关键字名称，默认为"page"
    page_size_query_param = 'page_size' # 指定URL查询（每页显示多少条数据）关键字名称，默认为None
    max_page_size = 50 # 每页最多显示多少条数据

    # 重写分页响应数据
    def get_paginated_response(self, data):
        code = 200
        msg = '成功'
        if not data:
            code = 404
            msg = "没有发现数据！"

        return Response(OrderedDict([
            ('code', code),
            ('msg', msg),
            ('count', self.page.paginator.count),
            # ('next', self.get_next_link()),
            # ('previous', self.get_previous_link()),
            ('data', data)
        ]))
```

自动生成接口文档

由于项目开发经验欠缺或着急上线，需求不断改动，项目设计阶段定义的接口已经面目全非，这给前端开发人员参考带来一定困难，如何改善这个问题呢?
Swagger来了，它是一个应用广泛的REST API文档自动生成工具，生成的文档可供前端人员查看。

文档参考：https://django-rest-swagger.readthedocs.io/en/latest/

自动生成接口文档

安装：

```
pip install django-rest-swagger
```

添加APP：

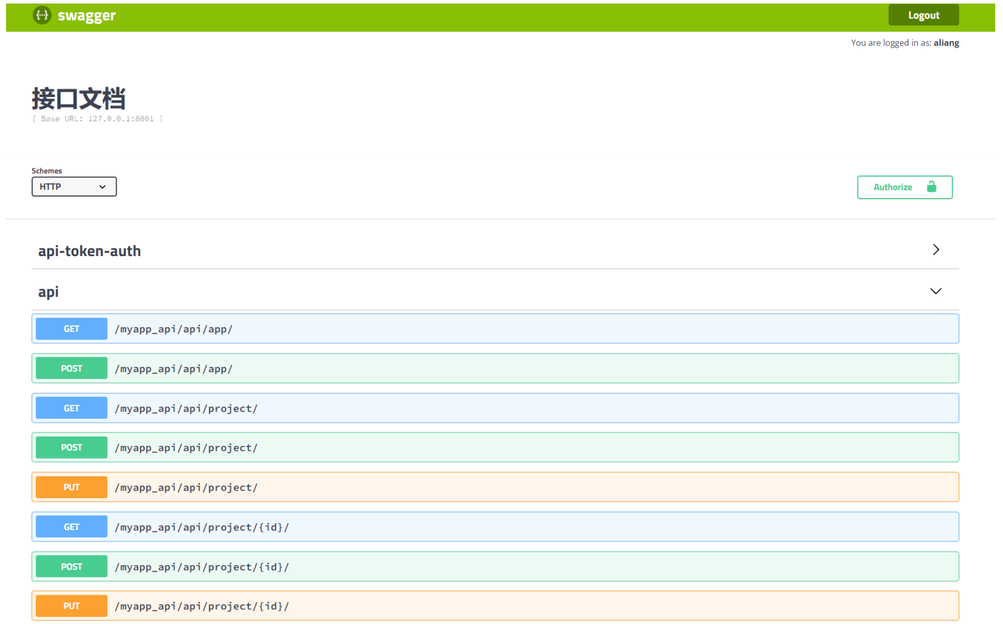
```
INSTALLED_APPS = [
    ...
    'rest_framework_swagger',
]
```

DRF配置：

```
REST_FRAMEWORK = {
    # API接口文档
    'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema',
}
```

URL路由：

```
from rest_framework_swagger.views import get_swagger_view
schema_view = get_swagger_view(title='接口文档')
urlpatterns += [
    re_path('^docs/$', schema_view),
]
```



谢谢

阿良个人微信

DevOps技术栈公众号

阿良教育：www.aliangedu.cn