DRF框架核心背记知识点

1. 序列化器

1.1 序列化器定义的基本形式

```
from rest_framework import serializers

class 序列化器类名(serializers.Serializer):
  # 字段名 = serializers.字段类型(选项参数)
  # 字段名 = serializers.字段类型(选项参数)
  # 字段名 = serializers.字段类型(选项参数)
  # ...
```

问题：序列化器类中的字段对应什么地方?

- 序列化操作时，字段对应的是被序列化对象的属性名；
- 反序列化操作时，字段对应的是传入的字典中的key。

1.2 创建序列化器对象的2个参数

```
# 创建序列化器对象
serializer = 序列化器类名(instance, data, **kwargs)
```

问题：instance和data参数什么时候使用?

- 序列化操作时，将被序列化的实例对象传递给instance
- 数据校验时，将待校验的字典数据传递给data

1.3 序列化器核心选项参数

read_only和write_only：不做设置时，默认值都为False，表明对应字段在序列化操作和反序列化操作时都会发挥作用。

read_only=True：表明指定字段只在序列化操作时发挥作用，反序化操作时会直接忽略此字段的存在。

write_only=True：表明指定字段只在反序列化操作时发挥作用，序化操作时会直接忽略此字段的存在。

1.4 序列化单个对象和多个对象的区别

序列化多个对象时，创建序列化器对象需要添加many=True的参数。

```
# 序列化单个对象
book = BookInfo.objects.get(id=1)
serializer = BookInfoSerializer(book)
serializer.data

# 序列化多个对象
books = BookInfo.objects.all()
serializer = BookInfoSerializer(books, many=True)
serializer.data
```

1.5 关联对象嵌套序列化的3种方式

在序列化一个对象时，将其关联的对象数据一并进行序列化，比如序列化hero对象时，嵌套序列化关联的hbook图书对象。

```
# 1. 将关联对象序列化为关联对象的主键
hbook = serializers.PrimaryKeyRelatedField(read_only=True)

# 2. 将关联对象使用指定的序列化器进行序列化
hbook = BookInfoSerializer()

# 3. 将关联对象序列化为关联对象所属类的__str__方法的返回值
hbook = serializers.StringRelatedField()
```

1.6 反序列化数据校验基本使用

```
serializer = 序列化器类(data=<待校验字典数据>)
# 进行数据校验
res = serializer.is_valid()

if res:
  print('校验通过：', serializer.validated_data)
else:
  print('校验失败：', serializer.errors)
```

is_valid默认校验：数据完整性、数据类型、是否满足指定选项参数的限制。

1.7 数据校验补充验证的2种方式

```
# 方式1：在序列化器类中定义 `valdate_<fieldname>` 方法，针对指定字段进行数据校验
def validate_<fieldname>(self, value):
  """
  value是指定字段的值
  """
  pass

# 方式2：在序列化器类中定义 `valdate` 方法，可以结合字段进行数据校验
def validate(self, attrs):
  """
  attrs是创建序列化器对象时，传入的字典数据
  """
  pass
```

注：校验通过之后，必须将对应的参数进行返回！！！

1.8 数据保存实现新增或更新

前提：**必须数据校验通过，才可以进行数据保存。**

保存：**序列化器对象.save()**

过程：

1）在save方法内部会调用序列化器类的create或update方法；

```
# ① 创建序列化器对象时，如果未传递instance参数，则save方法内部会调用对应序列化器类中的create方法
serializer = 序列化器类(data=<待校验字典数据>)
serializer.is_valid()
serializer.save()

# ② 创建序列化器对象时，如果传递了instance参数，则save方法内部会调用对应序列化器类中的update方法
serializer = 序列化器类(instance=<实例对象>, data=<待校验字典数据>)
serializer.is_valid()
serializer.save()
```

2）我们可以在create方法中实现数据新增，在update方法中实现数据更新；

注：如果序列化器类继承的是Serializer，create和update方法需要自己进行实现

1.9 ModelSerializer类的使用

简介：

ModelSerializer是Serializer类的子类，相对于Serializer，增加了以下功能：

- 基于模型类字段自动生成序列化器类中对应的字段
- 包含默认的create()和update()方法的具体代码实现

使用：

定义序列化器类时，如果序列化器类对应的是Django中的某个模型类，可以直接继承于ModelSerializer。

示例：

```
class BookInfoSerializer(serializers.ModelSerializer):
    """图书序列化器类"""
    class Meta:
        # 指明序列化器类对应的模型类
        model = BookInfo
        # 指明依据模型类的哪些字段自动生成序列化器类中对应的字段
        # fields = '__all__'
        # fields = ('id', 'btitle', 'bpub_date', 'bread', 'bcomment')
        exclude = ('is_delete', )

        # 为自动生成的序列化器类字段添加选项参数或修改原有的选项参数。
        extra_kwargs = {
            'bread': {
                'min_value': 0
            },
```

```
            'bcomment': {
                'min_value': 0
            },
            'btitle': {
                'min_length': 3
            }
        }
```

2. 视图类

2.1 APIView

APIView是REST framework提供的所有视图的基类，继承自Django的View类。

APIView与View的区别：

- **请求对象**：传入到视图中的request对象是REST framework的 Request 对象，而不再是Django原始的HttpRequest 对象；
- **响应对象**：视图可以统一返回REST framework的 Response 对象，响应数据会根据客户端请求头 Accpet 自动转换为对应的格式进行返回；
- **异常处理**：任何 APIException 的子异常都或 Http404 异常都会被DRF框架默认的异常处理机制处理成对应的响应信息返回给客户端；
- **其他功能**：认证、权限、限流。

Request请求对象：

属性名	说明
data	包含解析之后的请求体数据，已经解析为了字典或类字典，相当于Django原始request对象的body、POST、FILES属性的综合。
query_params	包含解析之后的查询字符串数据，相当于Django原始request对象的GET属性。

Response请求对象：

```
from rest_framework.response import Response
response = Response(<原始响应数据>)
```

原始的响应数据，会根据客户端请求头的 Accpet，自动转换为对应的格式并进行返回，默认支持返回json和网页，如：

Accept请求头	说明
application/json	服务器会将原始响应数据转换为json数据进行返回，没指定Accept时，默认返回json
text/html	服务器会将原始响应数据转换为html网页进行返回

2.2 GenericAPIView

GenericAPIView继承自APIView，在APIView功能基础上，主要增加了操作序列化器和数据库查询的属性和方法。

序列化器相关的属性和方法：

属性	**说明**
serializer_class	指定视图所使用的序列化器类
方法	**说明**
get_serializer_class	默认返回serializer_class指定的序列化器类
get_serializer	创建一个指定序列化器类的对象并返回

数据库查询相关的属性和方法：

属性	**说明**
queryset	指定视图所使用的查询集
方法	**说明**
get_queryset	默认返回queryset指定的查询集
get_object	从指定的查询集中查找一个指定的对象并返回，根据url地址中提取的pk参数的值进行查询

其他功能：过滤、排序、分页。

2.3 Mixin扩展类

Mixin扩展类来源：

视图继承了 GenericAPIView 之后，使用 GenericAPIView 中的属性和方法写出的代码变成了通用的代码流程，这些通用的代码被 DRF 框架封装成了 5 个类，就是 5 个 Mixin 扩展类。

5个Mixin扩展类：

扩展类	**封装方法**	**说明**
ListModelMixin	list	封装获取一组数据的通用代码流程
CreateModelMixin	create	封装新增一条数据的通用代码流程
RetrieveModelMixin	retrieve	封装获取指定数据的通用代码流程
UpdateModelMixin	update	封装更新指定数据的通用代码流程
DestroyModelMixin	destroy	封装删除指定数据的通用代码流程

2.4 子类视图类

子类视图类特点：

- 一定继承了GenericAPIView和对应的Mixin扩展类
- 实现了对应的请求处理方法(get、post等...)

子类视图类举例：

子类视图类	说明
ListAPIView	① 继承了GenericAPIView和ListModelMixin ② 同时提供了get方法

3. 视图集

问题：视图集和类视图的区别?

1. 直接继承的父类不同

   ```
   类视图：View、APIView、GenericAPIView、子类视图
   视图集：ViewSet、GenericViewSet、ReadOnlyModelViewSet、ModelViewSet
   ```

2. 处理方法的名称不同

   ```
   类视图：get、post、put、delete
   视图集：list、retrieve、create、update、destroy
   ```

3. URL 地址配置不同

   ```
   类视图：类视图.as_view()
   视图集：视图集.as_view({'请求方式': '处理方法', ...})
   ```

3.1 基本使用

概念：将一组相关的API接口放在同一个类中进行实现，这个类就是视图集。

基本使用：

```
1. 继承视图集的父类(ViewSet、GenericViewSet、ReadOnlyModelViewSet、ModelViewSet)
2. 视图集中的处理方法不再以对应的请求方式(get、post等)命名，而是以对应的操作命名(list、create等)
3. 进行url地址配置时，需要给as_view传递一个字典参数，明确指明请求方法和处理函数之间的对应关系
```

3.2 视图集父类

父类名称	说明
ViewSet	① 继承ViewSetMixin和APIView，基本不用 ② ViewSetMixin中重写了as_view方法，允许给as_view传递一个字典参数
GenericViewSet	① 继承ViewSetMixin和GenericAPIView，可以搭配Mixin扩展类进行使用
ReadOnlyModelViewSet	① 继承了GenericViewSet、ListModelMixin和RetrieveModelMixin
ModelViewSet	① 继承了GenericViewSet和5个Mixin扩展类

3.3 路由Router

作用：动态生成视图集中API接口的url配置项。

基本使用：

```
# ① 创建Router对象
from rest_framework.routers import SimpleRouter, DefaultRouter
router = SimpleRouter()
# ② 注册视图集
router.register('prefix', 'viewset', 'basename')
# ③ 添加路由数据
urlpatterns += router.urls
```

4. 序列化器类定义核心注意点

4.1 问题1：定义序列化器类时，如何确定序列化器类中需要定义几个字段?

答：定义序列化器类时，参考API接口设计，参数字段+响应字段，合起来就是定义序列化器类时所需的字段。

```
# 比如：美多后台管理员登录API接口
API：POST /meiduo_admin/authorizations/
参数：
  {
    "username": "用户名",
    "password": "密码"
  }
响应：
  {
    "id": "用户id",
    "username": "用户名",
    "token": "jwt token"
  }
```

```
# 说明：
# ① 根据上面的API，定义管理员登录序列化器类时，将参数和响应字段结合起来，序列化器类中一共需要4个字段：id、username、password、token
# ② 根据模型自动生成序列化器类的字段时，若某字段不能自动生成(即：模型中没有此字段)，则需要自己在序列化器类中添加定义。比如：User模型中没有token字段，序列化器类中的token字段无法生成，需要自己添加
# ③ 根据模型自动生成序列化器类的字段时，若生成的某字段不能满足需要(即：根据模型能生成，但是不满足需要)，则需要自己在序列化器类重新定义。比如：User模型中有username字段，但是生成序列化器类中的username字段不满足需要，需要自己重定义
```

4.2 问题2：定义序列化器类时，如何进行字段的write_only和read_only设置?

答：定义序列化器类时，参考API接口设计，只在参数中有的字段设置为write_only=True，只在响应中有的字段设置为read_only=True，参数和响应中都有的字段不需要单独设置。

```
# 比如：美多后台管理员登录API接口
API：POST /meiduo_admin/authorizations/
参数：
  {
    "username": "用户名",
    "password": "密码"
  }
响应：
  {
    "id": "用户id",
    "username": "用户名",
    "token": "jwt token"
  }

# 说明：
# ① username在参数和响应中都有，不需要单独设置write_only和read_only
# ② password只在参数中出现，需要设置为：write_only=True
# ③ id和token只在响应中出现，需要设置为：read_only=True
```

4.3 问题3：使用序列化器进行数据校验时，是否需要补充验证?

答：使用序列化器进行数据校验时，需要考虑序列化器默认验证是否能够完全满足业务数据校验的需求，如果不能完全满足校验数据的需求，则需求进行补充验证。注：is_valid默认校验包括参数完整性、数据类型、一些选项参数的限制。

```
# 比如，管理员登录序列化器类如下：
class AdminAuthSerializer(serializers.ModelSerializer):
    """管理员登录序列化器类"""
    # 注：模型中没有的字段，可以自己进行添加，也要放到fields属性中
    token = serializers.CharField(label='jwt token', read_only=True)
```

```
    # 注：定义序列化器类时，如果自动生成的字段不满足需要，可以重写
    username = serializers.CharField(label='用户名')

    class Meta:
        model = User
        # 注：自动生成字段时，id默认就是read_only=True
        fields = ('id', 'username', 'password', 'token')

        extra_kwargs = {
            'password': {
                'write_only': True
            }
        }

# 说明：
# ① 管理员登录进行数据校验时，需要补充用户名和密码是否正确的验证
class AdminAuthSerializer(serializers.ModelSerializer):
    """管理员登录序列化器类"""
    ...

    # 补充验证
    # validate_<fieldname>：针对单个字段进行校验
    # validate：结合多个字段的内容进行校验
    def validate(self, attrs):
        """
        attrs：创建序列化器对象时，传递给data的字段
        """
        # 获取username和password
        username = attrs.get('username')
        password = attrs.get('password') # None

        # 查询管理员是否存在
        try:
            user = User.objects.get(username=username, is_staff=True)
        except User.DoesNotExist:
            raise serializers.ValidationError('用户名或密码错误')
        else:
            # 校验密码是否正确
            if not user.check_password(password): # False
                raise serializers.ValidationError('用户名或密码错误')

        # 给attrs添加一个user，保存用户对象
        attrs['user'] = user

        # 返回attrs，必须返回！！！
        # attrs中有什么，validated_data中就有什么
        return attrs
```

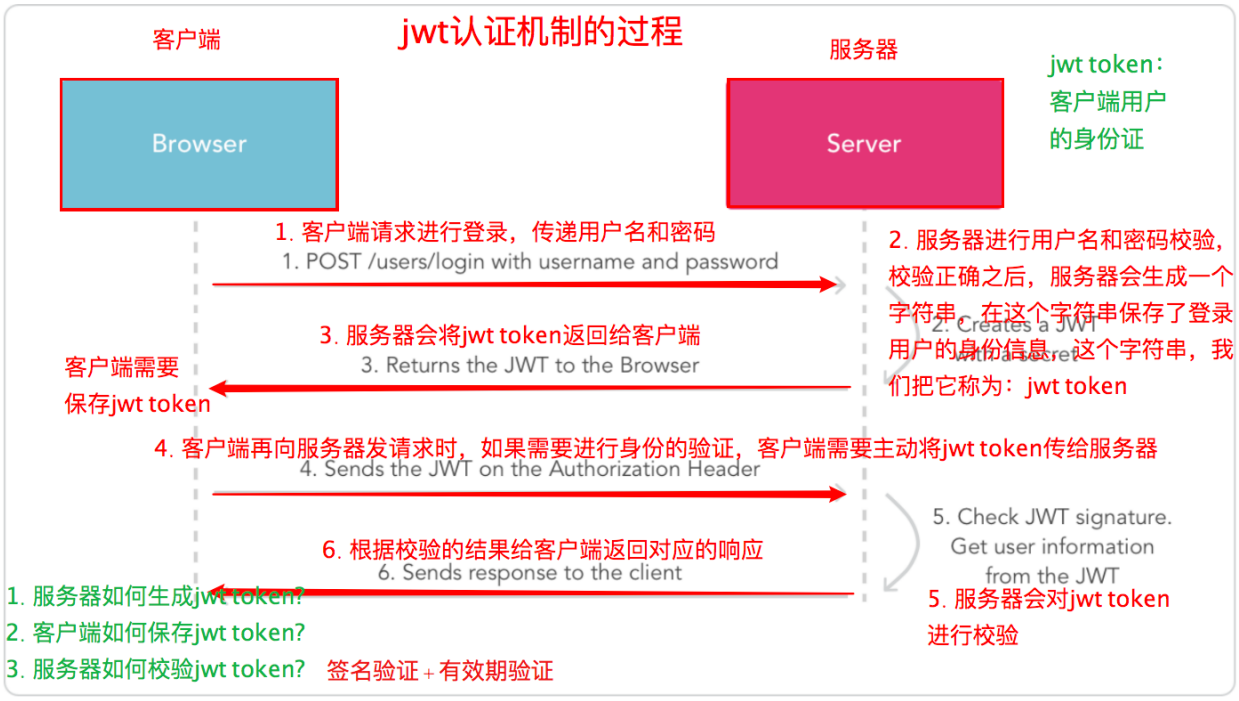
4.4 问题4：调用序列化器对象的save方法时，是否需要重写create和update?

答：调用序列化器对象的save方法时，需要考虑序列化器类中提供的create和update方法是否满足对应的业务需求，若不满足，则需要将序列化器类中的create或update进行重写。

```
# 比如：管理员登录时，我们将生成jwt token的代码封装到了管理员登录的序列化器类中，管理员登录序列化器类中的create不满足此处的业务需求，所以需要在管理员登录序列化器类进行create方法的重写。

class AdminAuthSerializer(serializers.ModelSerializer):
    """管理员登录序列化器类"""
    ...

    def create(self, validated_data):
        """抽取生成JWT token代码"""
        user = validated_data.get('user') # None

        from rest_framework_jwt.settings import api_settings

        jwt_payload_handler = api_settings.JWT_PAYLOAD_HANDLER
        jwt_encode_handler = api_settings.JWT_ENCODE_HANDLER

        payload = jwt_payload_handler(user)
        token = jwt_encode_handler(payload)

        # 给user对象临时增加一个token属性，保存jwt token的值
        # 注意：这里跟数据库操作没有关系，只是临时增加一个属性而已
        user.token = token

        return user
```

5. JWT认证机制的过程

5.1 JWT 认证机制的过程



5.2 JWT Token 的 3 部分组成

1）头部(header)：JSON 数据，保存 token 类型和签名加密的算法，生成时会进行 Base64 编码

2）载荷(payload)：JSON 数据，保存有效数据和 token 的有效时间，生成时会进行 Base64 编码

3）签名(signature)：签名字符串数据，防止 JWT token 被伪造

5.3 JWT 认证使用注意点

- 不要在 jwt 的 payload 部分存放敏感信息，该部分是客户端可解码的部分
- 服务器需要保存好自己使用的签名加密 secret 密钥
- 如果可以，请使用 https 协议(注：此方式不只是针对 JWT 认证机制，其他认证机制要想提高安全性也一样)

6. 不同身份用户权限控制表设计

6.1 不同身份用户权限控制需求

权限控制的层次：

① 用户登录还是未登录

比如：项目中某些API只允许登录过的用户进行访问

② 登录系统用户的权限控制

比如：登录系统之后，不同身份的用户也只能做指定的操作

举例：

以博学谷为例，其中有以下几个角色用户：

① 讲师：发布每日反馈、查看反馈结果…

② 助教：查看反馈结果…

③ 班主任：发布评分数据、查看学员信息...

④ 学员：提交每日反馈、参加阶段考试…

⑤ ……

思考：实现登录用户不同身份的权限控制，如何设计数据表?

6.2 不同身份用户权限控制表设计

1）用户表：保存每个用户的数据(用户名、密码、手机号...)

2）角色表(用户组表)：保存有哪些角色，哪些用户组

3）权限表：保存权限记录的数据，有哪些权限。

用户表和用户组表的关系：

一对多：一个用户只能属于一个组，而一个组可以包含多个用户。

多对多：一个用户可以属于多个组，同时一个组可以包含多个用户。

用户组表和权限表的关系：

多对多：一个组可以被分配很多权限，同一个权限也可以分配给不同的组

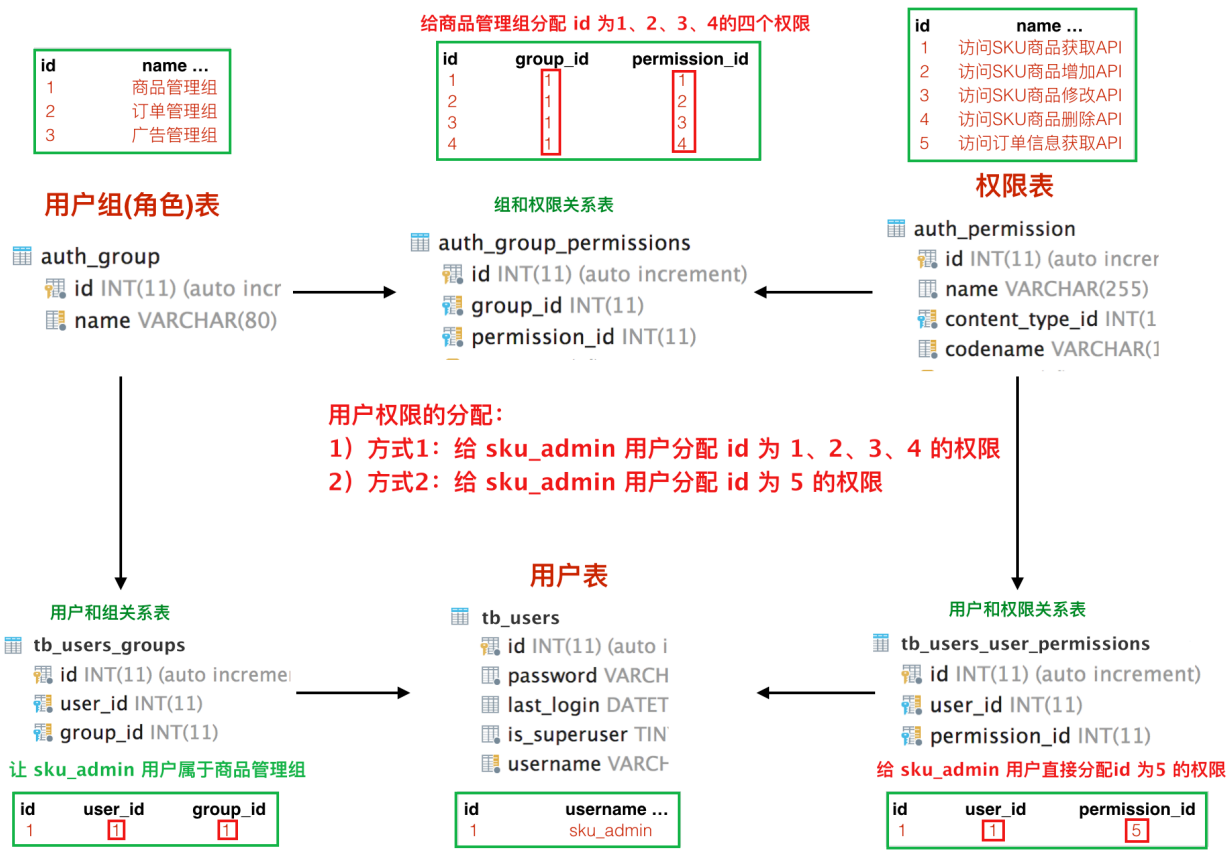
用户表和权限表的关系：

多对多：一个用户可以指定被分配很多权限，同一个权限也可以分配给不同的用户

6.3 用户的权限分配方式

1）方式1：先给用户分配组，然后给组分配权限，这个用户就拥有了所属组的权限

2）方式2：直接给用户分配权限


给商品管理组分配 id 为1、2、3、4的四个权限
id name ...
1 商品管理组
2 订单管理组
3 广告管理组
id group_id permission_id
1 1 1
2 1 2
3 1 3
4 1 4
id name ...
1 访问SKU商品获取API
2 访问SKU商品增加API
3 访问SKU商品修改API
4 访问SKU商品删除API
5 访问订单信息获取API
用户组(角色)表
组和权限关系表
权限表
auth_group
id INT(11) (auto incr
name VARCHAR(80)
auth_group_permissions
id INT(11) (auto increment)
group_id INT(11)
permission_id INT(11)
auth_permission
id INT(11) (auto increr
name VARCHAR(255)
content_type_id INT(1
codename VARCHAR(1
用户权限的分配:
1）方式1：给 sku_admin 用户分配 id 为 1、2、3、4 的权限
2）方式2：给 sku_admin 用户分配 id 为 5 的权限
用户表
用户和组关系表
用户和权限关系表
tb_users
id INT(11) (auto i
password VARCH
last_login DATET
is_superuser TIN
username VARCH
tb_users_groups
id INT(11) (auto increme
user_id INT(11)
group_id INT(11)
tb_users_user_permissions
id INT(11) (auto increment)
user_id INT(11)
permission_id INT(11)
让 sku_admin 用户属于商品管理组
给 sku_admin 用户直接分配id 为5 的权限
id user_id group_id
1 1 1
id username ...
1 sku_admin
id user_id permission_id
1 1 5