django 基础

1.安装django

pip install django

安装django会出现两个

- django 框架源码
- django-admin.exe 工具，用于创建django项目中的文件和文件夹

2.创建项目

> django项目中会有一些默认的文件和默认的文件夹

在终端

- 打开终端
- 进入某个（项目）目录
- 执行命令创建项目

> ...\django-admin.exe startproject 项目名称
>
> ```
> django-admin startproject mysite
> ```

在pycharm

- 直接选择创建

> 特殊说明：
>
> - 命令行：创建的项目是标准的
> - pycharm，在标准的基础上默认加了东西
>   - templates目录
>   - setting.py文件中添加了东西

> 默认项目的文件介绍
>
> ```
> project_name
>     manage.py        【项目的管理、启动项目、创建app、数据管理】
>     project_name
>         __init__.py
>         settings.py 【项目配置文件】【经常操作】
>         urls.py      【URL和函数的对应关系】【经常操作】
>         asgi.py      【接收网络请求】【不要动】
>         wsgi.py      【接受网络请求】【不要动】
> ```

3.APP

```
-项目
    -app，用户管理【表结构，函数，html模板，css】
    -app，订单管理【表结构，函数，html模板，css】
    -app，网站【表结构，函数，html模板，css】
    。。。


注意：本项目比较简单，无需多app
```

创建app命令

```
python manage.py startapp app01
```

app目录文件介绍

```
app01
    -__init__.py
    -admin.py           【django默认提供了admin后台管理】
    -apps.py            【app启动类】
    migrations          【数据库变更记录】
        __init__.py
    models.py           【重要，对数据库操作（默认有个数据库）】
    tests.py            【单元测试，写业务不用】
    views.py            【重要，视图函数】
```

4.快速上手

下面必须确保

- app已经注册（不是创建

  ```
  app01.apps.App01Config'
  ```

- 编写URL和视图函数的对应关系【urls.py中编写】
  - 先导入app的视图py
  - 后写入对应关系
- 编写视图函数【view.py】
- 启动
  - 命令行启动 python manage.py runserver
  - pycharm启动

4.1页面

需要写的内容（仅返回字符串

- url对应关系【urls.py】
- view.py视图函数编写【view.py】

仅返回字符串，仅仅使用HttpResponse("文本")

4.2templates

如果需要返回html文件

则需要返回 render(reqest,"xxx.html")

自动寻找templates目录下寻找

在视图函数中写的html文件

```
#1.首先去settings.py中的DIRS中寻找
#2.然后根据app的注册顺序去寻找templates
```

4.3静态文件

- 在开发过程中一般将：图片、css、js当作静态文件处理
- 静态文件默认在app里面的static中。
- 静态文件引入一般使用`{%load static%}`

4.4 CBV和FBV

在Django中，视图(View)分为基于函数的视图（Function-Based Views，FBV）和基于类的视图（Class-Based Views，CBV）。这两种视图均用于处理请求和返回响应，但在组织代码和重用方面有所不同。以下是它们的调用和使用方法的对比：

基于函数的视图 (FBV)

定义和使用：

FBV是最为简单直接的方式，它是一个Python函数，接收一个`HttpRequest`对象并返回一个`HttpResponse`对象。

示例代码：

```
from django.http import HttpResponse
from django.shortcuts import render

def my_view(request):
    if request.method == 'POST':
        # 处理POST请求
        return HttpResponse('This is a POST request')
    else:
        # 处理GET请求
        return HttpResponse('This is a GET request')

# 在urls.py中配置URL：
from django.urls import path
from .views import my_view

urlpatterns = [
    path('my-view/', my_view, name='my_view'),
```

```
]
```

基于类的视图 (CBV)

定义和使用：

CBV使用Python类实现视图，通过类的方法来处理请求。它提供了更好的代码组织和更灵活的重用性。

示例代码：

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

class MyView(View):
    def get(self, request):
        # 处理GET请求
        return HttpResponse('This is a GET request')

    def post(self, request):
        # 处理POST请求
        return HttpResponse('This is a POST request')

# 在urls.py中配置URL：
from django.urls import path
from .views import MyView

urlpatterns = [
    path('my-view/', MyView.as_view(), name='my_view'),
]
```

优缺点对比

特性	基于函数的视图 (FBV)	基于类的视图 (CBV)
代码简洁性	简单直观，适合小型和简单的视图逻辑	代码相对复杂，但有助于组织和重用逻辑
重用性	困难，需要手工重用代码	通过继承和Mixin机制实现代码重用
类继承	不适用	支持类的继承，有利于构建复杂的视图层次结构
装饰器使用	可直接使用Django提供的函数装饰器	需要使用 `method_decorator` 将装饰器应用于方法
可读性	对简单和单一功能的视图，易于理解	对于复杂逻辑，类结构有助于分离职责和逻辑
扩展性	扩展能力有限，复杂逻辑需要更多的代码和方法	提供更多可扩展的框架，支持多种Pre-build功能

- **FBV**适合简单的小型应用模块，快速实现简单功能。
- **CBV**适合需要多功能、复杂逻辑的大型项目，可以通过面向对象的特性来组织和简化视图代码。

根据项目的特定需求和复杂性，合理选择使用FBV或CBV，以提高开发效率和代码的可维护性。

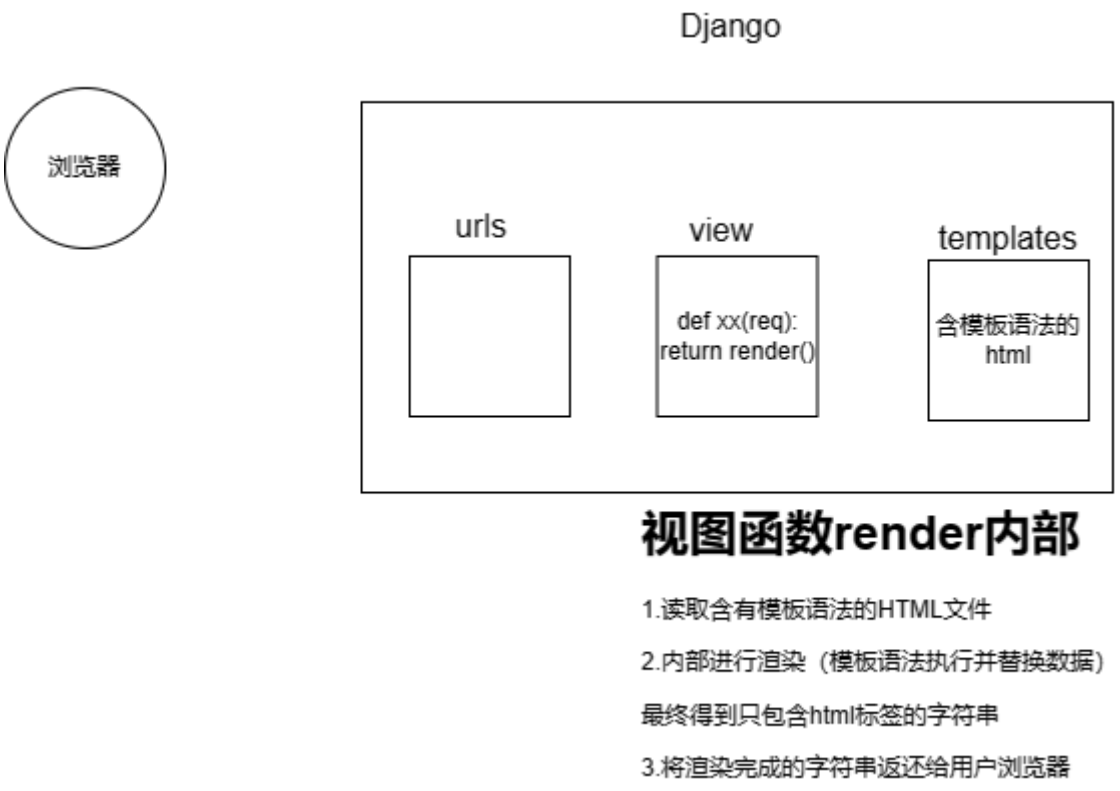
5.模板语法

本质上：在HTML中写一些占位符，由数据对这些占位符进行替换和处理

这个是django开发的模板语法

模板语法

- 变量`{{}}`
- 循环`{%for %} {%endfor%}`
- 条件`{% if %} {% elif %} {% endif %}`
- 其他注意事项：
  - 在模板中，列表元素通过.来访问，而不是[]



6.请求和响应

三个响应

- 返回字符串 HttpResponse
- 返回html内容 render(request,'xx.html',{:})
- 返回重定向地址 redirect('https://')

> 关于重定向，其实django是告诉浏览器目的地址，从而让浏览器自行去访问，而不是django请求目的地址得到html返回给浏览器

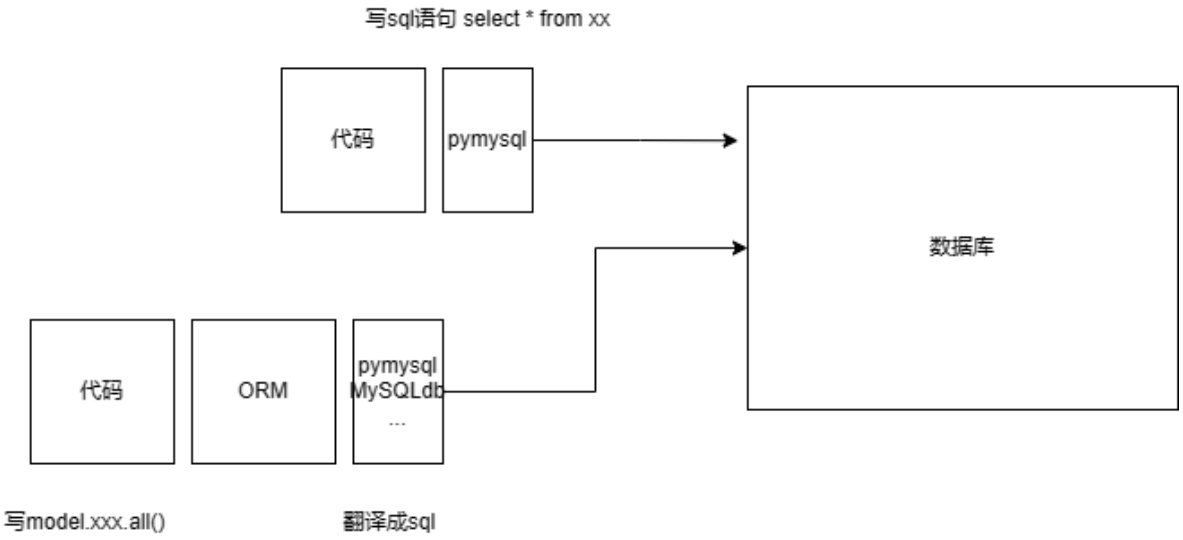
请求相关

- 获取请求方式 request.method
- 在URL上传递值，接受这个参数 request.GET
- 获取POST请求的所有参数 request.POST

7.数据库操作

可以使用的方法

- mysql数据库+pymysql
- django开发操作数据库，可以通过内部的ORM框架（而不是通过pymysql）



需要安装第三方模块

pip install mysqlclient

7.1 ORM

ORM 可以帮助我们做两件事：

- 创建、修改、删除数据库中的表（不用自己写sql语句）【无法创建数据库】
- 操作表中的数据（不用写sql语句）

7.1.1自己创建数据库

- 启动mysql服务
- 自导创建数据库

7.1.2django连接数据库

在settings.py中进行配置和修改

7.1.3django操作表

操作表可以分成

- 创建表
- 删除表
- 修改表

上述操作应该在models.py文件中

创建表步骤如下：

注意：app需要事先注册，否则不会提交数据库

1.model.py文件中创建类

```
# Create your models here.
class UserInfo(models.Model):
    name = models.CharField(max_length=32)
    password = models.CharField(max_length=64)
    age = models.IntegerField()


"""
创建了上述类，可以帮助生成下面的语句
create table app01_userinfo(
    id bigint auto_crement primary key,
    name varchar(32),
    passwrod varchar(64),
    age int)
"""
```

2. 运行两条指令（正式创建数据库表
    1. 生成迁移文件 python manage.py makemigrations
    2. 应用迁移文件 python manage.py migrate

对于数据库表的删除字段和删除表，可以对类直接注释，但是如果是在类中进行字段添加，需要有一定的限制（下面有三种选择）

- 命令行设置所有值（手动写
- 添加默认值（代码default
- 可以为空（null = True

> 综上所述，如果想对表结构进行跳跳转
>
> - 在models.py文件中操作类
> - 在执行两条命令

7.1.4django操作表中数据

```
from app01.models import Department,UserInfo
def orm(request):

#1.下面六条即为添加
    # Department.objects.create(title="销售部")
    # Department.objects.create(title="IT部")
    # Department.objects.create(title="运营部")
    # UserInfo.objects.create(name="zhu",password="123")
    # UserInfo.objects.create(name="zhu",password="123")
    # UserInfo.objects.create(name="zhu",password="123")

#2.下面为删除,先筛选后删除
    UserInfo.objects.filter(id=3).delete()
    Department.objects.all().delete()

#3.下面为获取数据
    #data_list = {行，行，行} data_list获取的是QuerySet类型
    data_list = UserInfo.objects.all() #相当于select * from
    for obj in data_list:
        print(obj.id,obj.name,obj.password)

    data_list2  = UserInfo.objects.filter(id=1)
    row_obj = data_list2.first()
    print(row_obj.id,row_obj.name,row_obj.password)


#4.更新数据
    UserInfo.objects.all().update(password = 999)
    UserInfo.objects.filter(id=2).update(password = 999)
    UserInfo.objects.filter(name = "朱弘飞").update(age=999)


    return HttpResponse("成功")
```

Django 的模型数据可以通过两种方式添加到数据库：

1. **使用 `.create()` 方法**：这是最简洁的方法，它会立即创建并保存数据到数据库，无需额外调用 `.save()` 方法。

   ```
   user = CustomUser.objects.create(username="user1", email="user1@example.com")
   ```

2. **类实例化 + `.save()` 方法**：这方式先实例化模型对象，然后再调用 `.save()` 方法将数据保存到数据库。这样可以在 `.save()` 之前对实例进行额外操作（如密码加密）。

   ```
   user = CustomUser(username="user1", email="user1@example.com")
   user.save()
   ```

这两种方法的区别是，`.create()` 是一步完成的，而实例化 + `.save()` 可以提供更多灵活性，比如允许你在保存之前设置额外的字段或执行一些逻辑。

> 此外对于序列化器，可以采用

> a = 序列化器类(data=xx)
>
> a.is_valid()
> a.save()
> 来添加数据

通过上述内容可以看出，QuerySet对象可以自动动态的修改数据表中的数据

> ## 执行 SQL 语句
>
> - 当你定义一个 `QuerySet`（例如通过 `UserInfo.objects.filter(name="朱弘飞")`），你实际上是在构建一个 SQL 查询，但这个查询在这一刻还没有被执行。`QuerySet` 是懒惰的，它仅在需要评估结果（如迭代、访问元素、调用 `.count()`、`.update()` 等）时才真正执行对应的 SQL 语句。
> - 特定的操作，如 `.update()` 和 `.delete()`，会立即执行相应的 SQL `UPDATE` 或 `DELETE` 语句来修改数据库中的数据，而不需要将数据加载到 Python 内存中。

7.1.5 Form和Modelform操作表中数据

当然，我将详细描述如何使用Django的 `Form` 和 `ModelForm` 类来完成数据库的数据"增删改查"（CRUD）操作，并逐步展示代码实现。

Form

Form类不直接关联数据库，因此我们需要手动处理数据的增删改查。

添加数据

使用Form创建记录时，我们需要手动从表单的数据中提取值，然后使用ORM来创建对象：

```
from django import forms
from app01.models import UserInfo

class UserInfoForm(forms.Form):
    name = forms.CharField(max_length=100)
    password = forms.CharField(widget=forms.PasswordInput)

def form_create_view(request):
    if request.method == 'POST':
        form = UserInfoForm(request.POST)
        if form.is_valid():
            # 从表单中提取数据并手动创建记录
            UserInfo.objects.create(
                name=form.cleaned_data['name'],
                password=form.cleaned_data['password']
            )
            return redirect('/success/')
    else:
        form = UserInfoForm()
    return render(request, 'form_template.html', {'form': form})
```

删除数据

删除记录时，需要根据某种条件手动查询并删除对象：

```
def form_delete_view(request, user_id):
    UserInfo.objects.filter(id=user_id).delete()  # 使用ID删除记录
    return redirect('/success/')
```

修改数据

修改已有记录时，需要加载现有数据，进行必要的改变后保存数据：

```
def form_update_view(request, user_id):
    user_instance = UserInfo.objects.filter(id=user_id).first()
    if request.method == 'POST':
        form = UserInfoForm(request.POST)
        if form.is_valid():
            user_instance.name = form.cleaned_data['name']
            user_instance.password = form.cleaned_data['password']
            user_instance.save()
            return redirect('/success/')
    else:
        # 初始化表单以现有数据
        form = UserInfoForm(initial={'name': user_instance.name, 'password':
user_instance.password})
    return render(request, 'form_template.html', {'form': form})
```

查询数据

查询数据一般在视图中直接使用ORM方法：

```
def form_read_view(request):
    users = UserInfo.objects.all()  # 查询所有记录
    return render(request, 'user_list.html', {'users': users})
```

ModelForm

ModelForm自动绑定到模型，使得数据处理更加简洁。

添加数据

使用ModelForm可以直接处理表单并保存数据：

```
from django import forms
from app01.models import UserInfo

class UserInfoModelForm(forms.ModelForm):
    class Meta:
        model = UserInfo
        fields = ['name', 'password']

def modelform_create_view(request):
    if request.method == 'POST':
        form = UserInfoModelForm(request.POST)
        if form.is_valid():
            form.save()  # 表单验证通过自动创建记录
            return redirect('/success/')
```

```
    else:
        form = UserInfoModelForm()
    return render(request, 'form_template.html', {'form': form})
```

删除数据

ModelForm本身不处理删除，需要手动通过ORM调用：

```
def modelform_delete_view(request, user_id):
    UserInfo.objects.filter(id=user_id).delete()  # 使用ID删除记录
    return redirect('/success/')
```

修改数据

ModelForm支持直接对现有对象进行更新：

```
def modelform_update_view(request, user_id):
    user_instance = UserInfo.objects.filter(id=user_id).first()
    if request.method == 'POST':
        form = UserInfoModelForm(request.POST, instance=user_instance)
        if form.is_valid():
            form.save()  # 保存对现有对象的更改
            return redirect('/success/')
    else:
        form = UserInfoModelForm(instance=user_instance)
    return render(request, 'form_template.html', {'form': form})
```

查询数据

与Form相同，直接使用ORM来进行数据查询：

```
def modelform_read_view(request):
    users = UserInfo.objects.all()  # 获取所有用户
    return render(request, 'user_list.html', {'users': users})
```

总结

- **添加（Create）**：ModelForm更简洁，因为它绑定到模型并可以直接调用`save()`。
- **删除（Delete）**：都需要通过ORM手动操作；删除并不是Form或ModelForm的任务。
- **修改（Update）**：ModelForm通过绑定实例简化了更新操作。
- **查询（Read）**：ORM方法直接实现查询，与表单无关。

下表从不同方面对比了使用普通数据库操作与通过`Form`和`ModelForm`在Django中处理数据时的区别：

操作/特性	普通数据库操作	`Form`	`ModelForm`
数据校验	手动校验	自动字段校验，并支持自定义校验逻辑	自动字段校验，并支持自定义校验逻辑
数据绑定	手动绑定输入数据到模型	需手动处理输入数据与表单字段绑定	自动绑定表单字段与模型字段

操作/特性	普通数据库操作	`Form`	`ModelForm`
对象创建/保存	手动创建/保存模型实例	表单验证通过后手动保存数据	通过 `save()` 便捷创建/更新模型实例
安全性	自行处理安全性及CSRF防护	内置CSRF防护	内置CSRF防护
HTML生成	自己编写HTML表单结构	自动生成表单HTML	自动生成与模型字段匹配的表单HTML
代码简洁性	代码量多，处理细节繁琐	中等，提供较为自动化的数据验证	简洁，自动处理与模型同步的多项任务
灵活性	高，可完全控制DB交互和逻辑细节	中等，需要自己定义表单逻辑	稍低，受限于模型定义的约束
适用场景	复杂业务逻辑或需要精确控制时	需要自定义表单但不直接绑定模型时	主要用于标准的CRUD操作和模型数据编辑

> 解释
>
> - **普通数据库操作**：
> - 灵活度最高，但需要开发者自行进行所有数据校验、验证和安全措施。
> - 适用于复杂的业务需求和逻辑要求较高的场景。
> - **Form**：
> - 提供了一定程度的自动化校验功能，但不直接绑定到模型，需要手动实现数据保存等操作。
> - 适合需要高度自定义的表单，或者在没有直接相关数据库模型时使用。
> - **ModelForm**：
> - 大多数自动化处理与模型相关的任务，特别是常见的数据库交互，如创建和更新操作由`ModelForm`简化处理。
> - 非常适合标准CRUD需求，在需要快速实施与模型关联的表单时尤为便利。
>
> 通过此对比，我们可以根据实际需要选择适当的方式来进行数据操作，以在代码效率、灵活性和安全性之间取得良好平衡。

7.2 数据库关联

在 Django 中，数据模型的关联关系主要包括以下几种：

1. **一对多关系（Many-to-One）：**
   - 使用 `ForeignKey` 实现。
   - 一个对象与多个对象相关，但多个对象只能与一个对象关联。
2. **多对多关系（Many-to-Many）：**
   - 使用 `ManyToManyField` 实现。

- 一个对象可以与多个对象关联，多个对象也可以与一个对象关联。

3. **一对一关系（One-to-One）：**
   - 使用 `OneToOneField` 实现。
   - 一个对象与另一个对象相互唯一关联。

7.2.1 一对多关系

在一对多关系中，一个对象可以与多个对象相关，但多个对象只能与一个对象关联。比如，一个客户可以下多个订单。

示例：客户和订单

```
pythonCopy code# app/models.py
from django.db import models

class Customer(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Order(models.Model):
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
    order_date = models.DateField()

    def __str__(self):
        return f"Order for {self.customer} on {self.order_date}"
```

解释：

- `customer` 字段是 `ForeignKey`，实现了多对一关系。
- `on_delete=models.CASCADE` 表示当关联的 `Customer` 被删除时，相关的 `Order` 也将被删除。

数据操作：

```
pythonCopy code# 创建一个客户
customer = Customer.objects.create(name="John Doe")

# 创建多个订单
order1 = Order.objects.create(customer=customer, order_date="2024-05-01")
order2 = Order.objects.create(customer=customer, order_date="2024-05-02")

# 获取客户的所有订单
orders = Order.objects.filter(customer=customer)
print([order.order_date for order in orders])  # 输出: ['2024-05-01', '2024-05-
02']
```

7.2.2 多对多关系

在多对多关系中，一个对象可以与多个对象关联，多个对象也可以与一个对象关联。比如，一个学生可以选修多门课程，每门课程也可以有多名学生。

示例：学生和课程

```
pythonCopy code# app/models.py
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Course(models.Model):
    name = models.CharField(max_length=100)
    students = models.ManyToManyField(Student, related_name='courses')

    def __str__(self):
        return self.name
```

解释：

- `students` 字段是 `ManyToManyField`，实现了多对多关系。
- `related_name` 定义了反向查询的名称，可以通过 `student.courses.all()` 获取学生的所有课程。

数据操作：

```
pythonCopy code# 创建多个学生
alice = Student.objects.create(name="Alice")
bob = Student.objects.create(name="Bob")

# 创建一门课程
course1 = Course.objects.create(name="Math 101")

# 将学生添加到课程
course1.students.add(alice,bob)

# 获取课程的所有学生
students = course1.students.all()
print([student.name for student in students])  # 输出: ['Alice', 'Bob']

# 获取学生的所有课程
courses = alice.courses.all()
print([course.name for course in courses])  # 输出: ['Math 101']
```

7.2.3一对一关系

在一对一关系中，一个对象与另一个对象相互唯一关联。比如，一个用户可以有一个唯一的用户档案。

示例：用户和用户档案

```
pythonCopy code# app/models.py
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
```

```
        return self.name

class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    address = models.CharField(max_length=200)
    phone_number = models.CharField(max_length=15)

    def __str__(self):
        return f"Profile of {self.user}"
```

解释：

- `user` 字段是 `OneToOneField`，实现了一对一关系。
- `on_delete=models.CASCADE` 表示当关联的 `User` 被删除时，相关的 `UserProfile` 也将被删除。

数据操作：

```
pythonCopy code# 创建一个用户
user = User.objects.create(name="Jane Doe")

# 创建关联的用户档案
profile = UserProfile.objects.create(user=user, address="456 Elm Street",
phone_number="987-654-3210")

# 获取用户的档案
print(profile.address)  # 输出: '456 Elm Street'
print(user.userprofile.phone_number)  # 输出: '987-654-3210'
```

7.2.4 总结

1. **一对多关系（Many-to-One）：**
   - 使用 `ForeignKey`。
   - 适用于如客户与订单、作者与书籍的关系。
2. **多对多关系（Many-to-Many）：**
   - 使用 `ManyToManyField`。
   - 适用于如学生与课程、演员与电影的关系。
3. **一对一关系（One-to-One）：**
   - 使用 `OneToOneField`。
   - 适用于如用户与档案、员工与工位的关系。

7.2.5 其他相关关系

- **自关联：**
  - 一个模型与自身关联，比如树形结构中的父子节点关系。
  - 使用 `ForeignKey` 或 `ManyToManyField` 实现。
- **抽象基类：**
  - 允许创建一个共享字段和方法的抽象模型类，子类继承它们。
  - 使用 `abstract = True`。

- **多表继承：**
    - 子类继承父类模型的字段并具有自己的独立表。
    - 使用 `Multi-table inheritance`。

模型之间的关联不仅可以发生在同一个文件或应用中，还可以跨应用关联，这通常称为**跨文件关联**。主要的关联方式仍然是使用 `ForeignKey`、`OneToOneField` 和 `ManyToManyField`。它们与在同一个应用中的关联方式几乎完全相同。

8.身份认证

8.1 介绍

故事背景

小明是一位业余摄影师，他热爱在网上分享他的摄影作品。为了展示和销售他的作品，小明决定创建一个线上博客网站。这个网站不仅允许小明上传和管理他的照片，还提供一个商城功能，让访问者可以购买他的作品。为了保护他的作品并确保只有他才能管理这些内容，身份认证成为了网站必要的功能之一。

身份认证是验证用户身份的过程，确保用户是他们所声称的那个人。在Web应用程序中，身份认证通常涉及以下几个步骤：

1. **用户输入凭据**：
    - 用户使用用户名/密码组合、OAuth令牌或其他形式的凭据登录。
2. **验证凭据**：
    - 应用服务器检查用户提交的凭据是否与其数据库中存储的数据匹配。
3. **创建会话或令牌**：
    - 验证成功后，服务器通常会创建一个会话，或生成一个JWT（JSON Web Token）或其他令牌，以标识用户。这个令牌会在后续请求中用于验证用户身份。
4. **访问控制**：
    - 根据用户的身份及其权限等级，决定是否允许访问特定资源或功能。

使用场景

- **Web应用登录**：
    - 用户通过身份认证登录到电商网站、社交媒体或企业应用。
- **API访问**：
    - 使用API密钥或OAuth令牌进行身份认证，保障API资源的安全。
- **企业级应用**：
    - 结合LDAP、SAML等方案，实现复杂的企业用户认证和单点登录（SSO）。

Django中的身份认证

在Django中，身份认证是通过`django.contrib.auth`模块进行管理的，其中包括：

- **用户模型（User Model）**：用于存储用户信息和凭据。
- **认证后端（Authentication Backends）**：处理凭据验证的逻辑，实现自定义认证方案。
- **中间件**：自动管理用户登录状态和会话。
- **装饰器和权限系统**：控制视图访问权限，确保只有授权用户能够访问。

Django还支持集成第三方认证机制，如社会化登录（例如Google、Facebook登录）和第三方身份提供商的OAuth或OpenID连接。这些机制通常需要第三方库如`django-allauth`或`social-auth-app-django`。这些扩展大大简化了集成多种身份认证方法的过程。

8.2 身份验证vs表单验证

身份认证和表单验证虽然都涉及用户数据的处理，但它们的目的、流程和作用略有不同。

表单验证

目的：

- 确保用户提交的表单数据符合预期格式和要求。

流程：

1. 用户提交表单数据。
2. 服务器端进行验证，检查数据类型（如字符串、数字等）以及格式（如电子邮件格式、电话号码格式）。
3. 验证通过：数据被进一步处理（如存储到数据库）。
4. 验证失败：表单返回错误，用户需进行修改。

作用：

- 保证数据完整性及安全性，防止错误或恶意输入（如SQL注入）。

身份认证

目的：

- 确认用户的身份，确保他们有权访问系统中的资源或功能。

流程：

1. 用户提交凭据（如用户名和密码）。
2. 服务器查验凭据与存储信息是否匹配。
3. 匹配成功：创建会话或令牌，用户被识别为已登录。
4. 匹配失败：拒绝访问，并可能要求重新输入凭据。

作用：

- 确保系统资源和信息的安全，防止未经授权的访问。

相似性和差异

相似性:

- 两者都涉及用户提交的数据。
- 都需要对输入进行验证检查。
- 在Web应用中都依赖后端逻辑处理。

差异:

- 表单验证专注于检查字段的格式和完整性，身份认证专注于验证用户身份。
- 表单验证通常是身份认证的前一道关卡（确保凭据格式正确），但身份验证包含逻辑更为复杂。（检查凭据与系统信息是否匹配）

虽然身份认证涉及表单数据，但它的目标和工作流程超出了单纯验证数据的正确性，强调用户身份的合法性和安全性。

当然！以下是身份认证和表单验证在使用场景以及Django中的实现方式的对比表格：

特性	**身份认证**	**表单验证**
目的	验证用户身份，确保用户是其声称的对象	验证表单输入的正确性和格式完整性
使用场景	用户登录、权限管理、API访问控制	用户注册、信息提交、数据输入校验
Django实现	- 使用`django.contrib.auth`模块	- 使用`django.forms`模块
	- 使用默认用户模型或自定义用户模型	- 定义自定义的表单类，包含字段及其验证逻辑
	- 认证后端（如LDAP、OAuth）	- 使用内置验证器或自定义验证器进行字段验证
	- 使用中间件管理用户会话和身份状态	- 表单类负责整个验证逻辑，通常在视图中使用
成功结果	创建会话或返回JWT令牌，完成身份认证，提供访问权限	数据验证通过，通常会继续被保存或处理，如存入数据库
失败结果	拒绝访问，返回身份认证错误信息	返回错误信息，要求用户修改其输入并重新提交
相关组件	- 用户模型（User Model）	- Form类与ModelForm类
	- 认证后端、权限装饰器和中间件	- 各种字段验证器及自定义验证逻辑
安全性	保护敏感资源及数据，防止未经授权的访问	保证数据输入的完整性和正确性，避免恶意输入或错误

这张表格清晰地展示了身份认证和表单验证在使用目的、应用场景和Django实现方式上的不同之处。通过理解这些，可以更好地针对不同需求选择合适的技术实现。

8.3 身份认证和权限控制

身份认证和权限控制在Django中是息息相关但又有所不同的两部分。以下是两者的详细区别和联系：

身份认证

身份认证（Authentication）：指的是确认用户的身份。主要关注的是“你是谁”。

Django中的实现：

- Django提供了`django.contrib.auth`模块用于处理用户的身份认证。
- 用户需要输入用户名和密码来验证自己的身份。
- 认证成功后，通常会创建一个用户会话或JWT令牌以维持用户的登录状态，确保只有经过身份认证的用户才能执行特定操作或访问某些页面。

场景：

- 用户登录时，系统检查凭据是否匹配。
- 对于需要确保用户身份的API请求，每个请求都可能需要用户认证信息。

权限控制

权限控制（Authorization）：是在身份认证之后进一步确认用户可以做什么。主要关注的是“你可以做什么”。

Django中的实现：

- 使用权限系统来定义不同用户或用户组能够访问哪些资源和执行哪些操作。
- Django内置了一个权限系统与用户模型关联，可以为用户分配组和特定的权限。
- 可以使用Django的`@permission_required`装饰器或者`PermissionRequiredMixin`类来保护特定视图，使得只有具有特定权限的用户才能访问这些视图。

场景：

- 在管理后台界面，只有管理员用户可以添加或删除其他用户。
- 在博客平台中，普通用户可以创建和编辑自己的文章，但只有具有编辑作者权限的用户可以审核和发布其他用户的文章。

联系与整合

1. **先身份认证，后权限控制**：
    - 一个用户必须首先被认证（通过登录或其他方式确认身份），然后其权限才会被检查以确认其是否有权访问某些资源。
2. **权限基于身份**：
    - 用户的权限是附加在其身份上的，即在认证其身份后，系统会根据其身份分配的权限来授予或拒绝访问。
3. **统一管理**：

- Django通过`django.contrib.auth`模块整合了身份认证和基本的权限管理，使得开发者可以较为轻松地维护应用的安全性。

通过理解身份认证和权限控制的区别与联系，可以更有效地设计安全可靠的Web应用，确保只让合适的人访问合适的资源。这个体系在Django中被高度集成并且可扩展，以满足各种复杂应用的需求。

8.4 身份认证的实现

当然，我可以通过表格的形式来比较Django身份认证中的`auth`和中间件在实现身份认证时的优缺点。

特点	Django `auth`	Django 中间件
概述	`auth`是Django内置的身份认证系统，提供标准且安全的方法进行用户验证、登录、登出等功能。	中间件是Django请求处理过程的一部分，可以自定义认证逻辑。
使用便捷性	非常简便，提供一整套的认证和权限管理功能，只需简单配置即可使用。	需要自定义中间件，增加一定的复杂性，适合于需要复杂条件和额外处理的场景。
安全性	经过社区长期验证，安全机制较为健全，默认采用哈希密码存储等安全策略。	取决于自定义实现，安全性由开发人员负责，容易出错。
灵活性	提供基本和大多数常用的功能，支持扩展但需要遵循一定的接口。	高度灵活，可以完全根据需求自定义处理流程，但实现相对复杂。
配置难度	较低，只需配置认证后端和使用现有视图或表单即可。	较高，需要编写自定义中间件类以及处理请求和响应。
维护难度	较低，由于`auth`模块被广泛使用和测试，问题较少。	较高，自定义实现可能导致可维护性差，需要进行大量的测试。
适用场景	适用于大多数Web应用的标准用户认证需求，如用户登录、注册。	适合需要复杂的请求处理逻辑、额外验证步骤或多种认证方式结合的场景。
性能影响	标准实现效率较高，对性能影响较小。	如果实现不当，可能对请求处理性能有一定影响。

- **使用 Django `auth`**:
  - 比较适合于常规项目，可以快速集成并提供良好的安全保障。
  - 在不需要特殊处理的情况下优先使用，因其稳定性和社区支持。
- **使用 Django 中间件**:
  - 如果你的项目具有特殊的认证需求，中间件提供了灵活性，可以在请求的各个阶段进行自定义处理。
  - 适用于需要预处理请求或需要支持多种认证流的环境，但需注意安全和维护成本。

根据项目的具体需求选择合适的身份认证方案，可以提升开发效率并确保安全性和可维护性。

8.4.1 auth模块

`django.contrib.auth`模块是Django内置的身份认证系统。这一模块实现了一整套用户身份验证和权限管理功能，非常适合开发中小型Web应用。

常见功能

1. **用户管理**:
    - 用户创建、更新、删除。
    - 用户分组及权限管理。
    - 超级用户（管理员）和普通用户的区分。
2. **身份验证**:
    - 提供用于认证用户的功能（如登录和登出）。
    - 支持会话管理系统。
3. **密码管理**:
    - 提供安全的密码加密和验证方法。
    - 提供密码重设等功能。
4. **权限管理**:
    - 基于用户和组的权限系统。
    - 可以限制用户执行特定操作的权限。

常用的组件和方法

1. `User` 模型

功能:

- Django内置的用户模型，用于存储和管理用户信息。

常用字段:

- `username`: 用户名，唯一标识。
- `password`: 加密存储的密码。
- `email`: 用户电子邮件。
- `is_staff`: 是否具有管理员权限。
- `is_active`: 用户是否处于活跃状态。
- `is_superuser`: 是否为超级用户，拥有所有权限。

常用方法:

- `create_user(username, password=None, **extra_fields)`: 创建一个新用户并将其保存到数据库中。
- `set_password(raw_password)`: 设置用户的密码，自动处理哈希运算。

- `check_password(raw_password)`：验证布贺or密码是否与存储的哈希密码匹配。

示例：

```
from django.contrib.auth.models import User

# 创建一个用户
user = User.objects.create_user(username='john', password='secret',
email='john@example.com')

# 验证密码
if user.check_password('secret'):
    print("Password is correct!")

# 设置新密码
user.set_password('new_password')
user.save()
```

2. `authenticate()` 方法

功能：

- 验证提供的用户名和密码是否正确。

实现机制：

- 搜索匹配的用户并检查密码。
- 调用时可能需要传递`request`对象和关键词参数。

示例：

```
from django.contrib.auth import authenticate

user = authenticate(username='john', password='secret')
if user is not None:
    print("Authenticated successfully!")
else:
    print("Authentication failed.")
```

3. `login()` 方法

功能：

- 将用户标记为已登录，并建立会话。

实现机制：

- 通过Django的会话框架管理用户会话。

示例：

```python
from django.contrib.auth import login

def user_login(request):
    username = request.POST.get('username')
    password = request.POST.get('password')
    user = authenticate(request, username=username, password=password)
    if user is not None:
        login(request, user)  # 开启用户会话
        return redirect('home')
    else:
        print("Invalid login attempt")
```

4. `logout()` 方法

功能:

- 注销用户并结束会话。

实现机制:

- 清除当前用户的会话数据。

示例:

```python
from django.contrib.auth import logout

def user_logout(request):
    logout(request)
    return redirect('home')
```

5. `@login_required` 装饰器

功能:

- 限制视图函数只能被已登录用户访问。

实现机制:

- 如果用户未登录，自动重定向到登录页面。

示例:

```python
from django.contrib.auth.decorators import login_required

@login_required
def protected_view(request):
    return render(request, 'protected.html')
```

不常用的组件和方法

1. Permissions and Groups

- **Permissions**:

  功能:

  - 设置和检验用于访问不同功能和资源的权限。

  示例:

```
user.has_perm('app_label.permission_codename')
```

- **Groups**:

  功能:

  - 将用户分组，并为组赋予权限以简化权限管理。

  示例:

```
from django.contrib.auth.models import Group

# 创建一个新的用户组
editors_group, created = Group.objects.get_or_create(name='Editors')

# 将用户加入该组
user.groups.add(editors_group)
```

2. 自定义用户模型

- **AbstractUser**:

  功能:

  - 对内置用户模型的简单扩展。

  使用场景:

  - 添加额外字段，如年龄、地址等，同时保留密码和会话管理。

  示例:

```
from django.contrib.auth.models import AbstractUser

class CustomUser(AbstractUser):
    bio = models.TextField(blank=True, null=True)
```

- **AbstractBaseUser**:

  功能:

  - 更灵活的基类，允许完全自定义用户模型，需自行实现用户名字段唯一性和身份验证方法。

  示例:

```
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        email = self.normalize_email(email)
```

```
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

class CustomUser(AbstractBaseUser):
    email = models.EmailField(unique=True)
    USERNAME_FIELD = 'email'
    objects = CustomUserManager()
```

3. 信号 (Signals)

功能:

- 监控用户登录事件。

常用信号:

- `user_logged_in`, `user_logged_out`, `user_login_failed`.

示例:

```
from django.contrib.auth.signals import user_logged_in
from django.dispatch import receiver

@receiver(user_logged_in)
def on_user_logged_in(sender, request, user, **kwargs):
    print(f"{user.username} has logged in.")
```

auth基本使用流程

以下是一般使用Django身份认证系统的标准流程。

1. 设置和配置

确保项目中已经包含`django.contrib.auth`模块，且在`settings.py`的`INSTALLED_APPS`中已激活。

2. 用户模型

- Django默认的用户模型可以满足大多数基本需求，但如果有特殊需求，可以扩展或替代默认的用户模型。
- 对用户模型的操作，例如创建用户、检查用户密码等，通过Django的`User`模型来进行。

3. 用户注册和认证

- **用户注册**：通常你需要允许用户自己注册账户。这可以通过创建一个自定义视图处理用户输入的注册信息。
- **登录和登出**：使用Django内置的视图函数`auth.views.LoginView`和`auth.views.LogoutView`简单实现登录和登出功能。

4. 实现登录视图

下面是如何用Django实现一个简单的登录视图：

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login
from django.contrib import messages

def user_login(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('home')
        else:
            messages.error(request, 'Invalid username or password.')
    return render(request, 'accounts/login.html')
```

5. 授权管理

基于权限的访问控制可以通过给用户分配权限或分组来实现。在视图中，你可以使用装饰器或者类来检查用户权限：

- 使用装饰器`@login_required`确保视图只能被认证用户访问。
- 使用装饰器`@permission_required`确保用户具有特定权限。

6. 登出功能

使用Django的`logout`函数来实现用户登出功能：

```
from django.contrib.auth import logout

def user_logout(request):
    logout(request)
    return redirect('home')
```

> 为什么Django的auth系统有效
>
> 1. **安全性**：
>   - 使用安全的密码存储方法（例如加盐的散列）。
>   - 内置防护策略（如CSRF保护）和会话管理机制。
> 2. **易用性**：
>   - 提供现成的类和函数，可以很容易定制。
>   - 管理后台带有强大的用户和权限管理界面。
> 3. **集成性**：
>   - 自带应用能与Django其他功能无缝集成，比如中间件、模板系统等。

Django的auth系统提供了一个全面且可扩展的框架来管理用户认证和权限控制。在此基础上，你可以根据需求增加自定义功能，使得Web应用的用户管理变得既安全又高效。了解其基本流程对开发更复杂的认证机制至关重要。

8.5 身份认证的应用

需求场景

小明的网站需要一个用户账户系统，通过它能够区分普通访问者和他这个管理员。普通访问者可以浏览和购买照片，而小明需要拥有更高权限的管理员账户，以便能够：

1. 登录后台上传、编辑或删除照片。
2. 查看购买订单和发货管理。
3. 管理访问者的评论和留言。

为了使得这个账户系统不仅便利，还有助于保护小明的作品，网站需要实现严格的身份认证机制。

为何需要身份认证

- **保护管理功能**：小明是唯一的管理员，他需要通过身份认证来防止其他人未经授权地访问管理员功能。
- **个性化用户体验**：提供给用户的个性化视图，使管理员看到不同于普通访问者的界面。
- **保障安全交易**：确保支付和订单信息被妥善保护，避免数据被盗用。

Django身份认证实现

为了满足这个需求，小明的网站使用Django内置的身份认证系统。以下是实现的关键步骤示例：

1. **用户登录视图**

   小明的网站需要一个登录页面，只有通过正确的邮箱和密码认证后，才能访问管理员后台：

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login
from django.contrib import messages

def admin_login(request):
    if request.method == 'POST':
        email = request.POST.get('email')
        password = request.POST.get('password')
        user = authenticate(request, username=email, password=password)
        if user is not None and user.is_staff:  # 确保是管理员账户
            login(request, user)
            return redirect('admin_dashboard')  # 重定向到管理员后台
        else:
            messages.error(request, 'Invalid credentials or you do not have
permission.')
    return render(request, 'accounts/login.html')
```

2. **URL配置**

   配置使得登录页面可以通过访问特定的URL：

```
from django.urls import path
from . import views

urlpatterns = [
    path('login/', views.admin_login, name='admin_login'),
]
```

3. **用户角色配置**

   在用户管理系统中，确保小明的账户具备管理员权限（`is_staff=True`），以便他能够通过身份认证访问管理功能。

8.6 auth认证 vs drf认证

以下是对 Django `auth` 系统和 DRF `authentication` 系统的更详细描述和比较，包括它们的定义、适用场景、优缺点、使用示例等。

1. 定义

- **Django `auth` 系统**：
  Django 提供了一个内置的用户认证系统，用于用户管理、权限控制和用户会话管理。它支持基本的用户注册、登录、注销、密码管理等功能。
- **DRF `authentication` 系统**：
  Django Rest Framework 提供了多种认证机制，专门为 RESTful API 设计。它支持多种认证方法，如 Token 认证、Session 认证、Basic 认证、JWT 认证等，方便 API 的用户身份验证。

2. 适用场景

- **Django `auth` 系统**：
  - 适用于传统的 Web 应用。
  - 用于需要用户登录、注销和会话管理的场景。
  - 适合那些使用 Django 的模板系统进行页面渲染的项目。
- **DRF `authentication` 系统**：
  - 适用于 RESTful API。
  - 用于需要提供用户身份验证的 API 接口。
  - 适合前后端分离的项目，尤其是需要跨域请求的场景。

3. 返回形式

- **Django `auth` 系统**：
  - 通常通过 `request.user` 来访问当前登录的用户对象。
  - 返回的是 `User` 对象，包含用户的所有信息。

- **DRF `authentication` 系统**：
  - 通过认证类返回用户对象，返回值可以是 `User` 对象或 `None`（如果未认证）。
  - 可以通过 `request.user` 访问当前用户。

4. 优缺点

特性	Django `auth` 系统	DRF `authentication` 系统
优点	1. 内置于 Django，使用简单 2. 提供全面的用户管理和权限控制	1. 支持多种认证方式 2. 更灵活，适应不同 API 需求
缺点	1. 主要支持 session 认证，不适合 API 2. 不适合跨域请求	1. 需要配置，增加复杂性 2. 不同认证方式的实现可能较繁琐

5. 使用示例

Django `auth` 系统 示例：

这是一个基本的登录视图，用户通过表单提交用户名和密码，进行认证。

```
from django.contrib.auth import authenticate, login
from django.shortcuts import render, redirect

def my_login_view(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('home')  # 重定向到主页
        else:
            return render(request, 'login.html', {'error': 'Invalid
credentials'})
    return render(request, 'login.html')
```

DRF `authentication` 系统 示例：

这是一个使用 Token 认证的 API 视图，只有经过身份验证的用户才能访问。

```python
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated

class MyProtectedView(APIView):
    authentication_classes = [TokenAuthentication]  # 使用 Token 认证
    permission_classes = [IsAuthenticated]  # 只有认证用户才能访问

    def get(self, request):
        content = {'message': 'Hello, {}'.format(request.user.username)}
        return Response(content)
```

在 Django 中，DRF 认证返回的用户（`request.user`）和 Django `auth` 系统中的用户（`django.contrib.auth.models.User`）实际上是同一个用户对象。这是因为 DRF 认证系统是基于 Django 的 `auth` 系统构建的。

1. **用户模型**：
   - Django 的 `auth` 系统使用 `User` 模型来表示用户。该模型包含用户的基本信息（如用户名、密码、电子邮件等）。
2. **DRF 认证**：
   - 当使用 DRF 的认证机制（如 Token 认证或 Session 认证）时，认证过程会将请求的用户信息解析为 `User` 对象。
   - 通过 `request.user` 可以访问当前认证的用户，该对象是 Django `auth` 系统中的同一个 `User` 实例。

Django `auth` 系统和 DRF `authentication` 系统各自针对不同的应用场景提供了认证机制。Django `auth` 更适合传统 Web 应用，而 DRF `authentication` 则为 RESTful API 提供了灵活的认证方式。根据项目需求，开发者可以选择合适的认证机制。

> 在 Django REST Framework (DRF) 中，用户的登录状态通常是通过认证类来管理的。DRF 提供了多种认证方式，以便您可以根据需求选择适合您的项目的方案。下面是一些常见的认证方式以及如何控制用户的登录状态。
>
> 1. 使用 Session Authentication
>
> 如果希望使用与 Django 登录系统相同的会话认证，可以使用 DRF 的 `SessionAuthentication`。
>
> 安装并配置
>
> 首先，确保您已经安装了 DRF：
>
> ```
> pip install djangorestframework
> ```
>
> 在您的 Django 项目的 `settings.py` 文件中，您可以使用如下配置来启用会话认证：

```
REST_FRAMEWORK = {
 'DEFAULT_AUTHENTICATION_CLASSES': [
     'rest_framework.authentication.SessionAuthentication',
     'rest_framework.authentication.BasicAuthentication',
 ],
 # 其他配置...
}
```

使用

当用户登录后，可以通过 DRF 的视图处理请求，在视图中可以使用 `request.user` 来获取当前认证的用户。

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated

class ExampleView(APIView):
 permission_classes = [IsAuthenticated]  # 确保用户已登录

 def get(self, request):
     content = {'message': 'Hello, {}'.format(request.user.username)}
     return Response(content)
```

在这个例子中，只有经过认证的用户才能访问 `ExampleView`，如果用户未登录，它将返回 401 Unauthorized 错误。

2. 使用 Token Authentication

如果您希望实现基于 Token 的认证（例如，适用于移动应用或单页面应用），可以使用 DRF 的 `TokenAuthentication`。

安装并配置

首先，确保您已安装 `djangorestframework.authtoken`：

```
pip install djangorestframework
```

接着，在 `settings.py` 中将 `TokenAuthentication` 添加到认证类中，并确保在 `INSTALLED_APPS` 中添加 `rest_framework.authtoken`：

```
INSTALLED_APPS = [
 # ... 其他应用 ...
 'rest_framework',
 'rest_framework.authtoken',
]

REST_FRAMEWORK = {
 'DEFAULT_AUTHENTICATION_CLASSES': [
     'rest_framework.authentication.TokenAuthentication',
 ],
 # 其他配置...
}
```

创建 Token

您可以使用现成的创建 Token 的视图，或者根据需求自定义。例如，您可以创建一个登录视图来生成 token:

```
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)
     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})
```

使用 Token

在 API 请求中，用户需要在请求头中包含 Bearer Token，以访问保护的资源：

```
Authorization: Token your_token_here
```

您可以在视图中使用 `request.user` 检查用户的登录状态。

```
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response

class ExampleTokenView(APIView):
 permission_classes = [IsAuthenticated]  # 确保用户已登录

 def get(self, request):
     content = {'message': 'Hello, {}'.format(request.user.username)}
     return Response(content)
```

> 总结
>
> 在 DRF 中，用户登录状态的控制主要通过认证类实现。当用户成功通过认证后，您可以在视图中通过 `request.user` 来访问当前认证的用户信息。根据您的需求选择合适的认证方式，可以有效地保护 API 端点并控制用户的访问权限。如果您还有其他问题或需要进一步的解释，请随时提问！

8.6 其他

自定义用户模型

如果需要自定义存储用户数据，可以通过自定义用户模型来实现，Django提供了扩展及替换默认用户模型的灵活性。以下是自定义用户模型的一般步骤：

步骤一：自定义用户模型

1. **使用 `AbstractUser` 扩展**

   继承 `AbstractUser` 可以在保留现有功能的基础上拓展字段。

```python
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    # 添加自定义字段
    bio = models.TextField(blank=True, null=True)
    birth_date = models.DateField(null=True, blank=True)
```

2. **使用 `AbstractBaseUser` 创建全新用户模型**

   如果需要完全自定义，可以继承 `AbstractBaseUser`，同时需要指定 `UserManager`。

```python
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager
from django.db import models

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError('The Email field must be set')
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        return self.create_user(email, password, **extra_fields)
```

```
class CustomUser(AbstractBaseUser):
    email = models.EmailField(unique=True)
    name = models.CharField(max_length=50)
    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['name']

    objects = CustomUserManager()
```

步骤二：修改Django设置

- 在项目的`settings.py`中，将默认的用户模型替换为自定义模型。

```
AUTH_USER_MODEL = 'yourapp.CustomUser'
```

  - 这里`yourapp`是包含`CustomUser`模型的应用名称。

步骤三：应用迁移

- 确保数据库结构符合自定义用户模型的定义。

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

步骤四：使用自定义用户模型

- 在整个项目中始终使用`get_user_model()`来获取用户模型，不要直接引用`User`模型。

```
from django.contrib.auth import get_user_model

User = get_user_model()
```

小结

- **扩展用户模型**: 若仅需添加字段，推荐继承`AbstractUser`。
- **重写用户模型**: 需要完全自定义身份认证的结构时，继承`AbstractBaseUser`。
- **确保一致性**: 一旦定义自定义用户模型，应在整个项目中始终使用，以确保数据一致性和功能的正常运作。

利用自定义用户模型，你可以更好地适应项目的特定需求，同时保持与Django 的`auth`认证系统的完美集成。

create_user说明

在Django中，对于创建用户实例，`create_user`方法与`create`方法的区别主要在于安全性和功能性，特别是在处理用户密码时：

差异与原因

1. **密码哈希处理**:
    - `create_user`: 该方法会自动调用`set_password`来对用户的密码进行哈希处理。Django使用PBKDF2等安全的哈希算法来存储密码，以防止明文密码被存储。
    - `create`: 使用`create`方法时，密码不会自动哈希处理。传递给`create`的密码将直接以明文形式存储在数据库中，在大多数情况下，这会导致安全隐患。
2. **内置的额外逻辑**:
    - `create_user`: 除了处理密码外，还可能包含额外的用户相关初始化逻辑，比如设置默认值、信号发送等。
    - `create`: 主要功能是直接创建并保存数据库记录，适用于那些不需要额外处理的普通模型实例。

使用场景

- `create_user`: 应用于创建用户时，以确保密码安全且遵循用户模型创建的最佳实践。
- `create`: 用于创建不涉及到特殊处理逻辑的其他模型实例。

示例

在创建用户时应使用`create_user`方法:

```
user = User.objects.create_user(username='john', password='secure_password',
email='john@example.com')
```

这确保密码通过Django的密码哈希系统存储，确保安全性。

自定义扩展

如果你自定义了用户模型并实现自定义管理器，你仍然可以定义类似的方法:

```
class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError('The Email field must be set')
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user
```

这里的`set_password`确保密码哈希化并安全存储。

@login_required默认登录url

在Django的`@login_required`装饰器中，重定向到登录页面的URL是通过Django的设置来确定的，主要通过`LOGIN_URL`和`LOGIN_REDIRECT_URL`进行配置。在默认情况下，如果你使用`@login_required`装饰器限制访问，Django会自动将未登录用户重定向到配置的登录页面。

确定登录页面的URL

1. **LOGIN_URL 设置**
   - `LOGIN_URL`是一个常量，你可以在你的Django项目的`settings.py`文件中配置它。
   - 它指定了未登录用户在访问需要登录的页面时将被重定向到的URL。

```
# settings.py

LOGIN_URL = '/accounts/login/'  # 这里的URL是示例，请根据你的项目路径设置
```

2. **示例应用**

   在视图函数中使用`@login_required`装饰器：

```
from django.contrib.auth.decorators import login_required
from django.http import HttpResponse

@login_required
def my_protected_view(request):
    return HttpResponse("You are logged in and can see this content!")
```

3. **默认行为**
   - 当一个未登录用户试图访问被`@login_required`修饰的视图时，Django会查找`settings.py`中的`LOGIN_URL`的值。
   - 如果未设置`LOGIN_URL`，Django将使用`/accounts/login/`作为默认值。
   - 要重定向到自定义登录页面，只需将`LOGIN_URL`设置为所需的登录页面的URL路径。
4. **LOGIN_REDIRECT_URL 设置**
   - 此设置指定了一旦登录成功后，用户将被重定向到的URL。
   - 默认情况下，这是`/accounts/profile/`，但可以在`settings.py`中更改。

```
# settings.py

LOGIN_REDIRECT_URL = '/dashboard/'  # 登录成功后重定向的页面
```

通过配置`LOGIN_URL`和`LOGIN_REDIRECT_URL`，你可以灵活地控制未登录用户被重定向到哪个页面进行登录，以及登录成功后他们将被导向到哪个页面。确保将这些URL配置为与你项目的URL匹配的路径，以确保用户体验的顺畅。

常见的认证方式

在 Django REST Framework (DRF) 中，身份认证是一个重要的组成部分。DRF 提供了多种身份认证方式，允许开发者根据具体需求选择合适的认证机制。以下是对`JWTAuthentication`（JSON Web Token 认证）及其他一些常见认证方式的总结和对比。

1. **JWT Authentication (JSON Web Token 认证)**:
   - 使用 JWT 令牌来验证用户的身份。用户在登录成功后获得一个令牌，后续请求将此令牌作为认证标识。

- JWT 是一种自包含的令牌格式，包含用户身份信息和有效期。由于是无状态的，服务器无需存储会话。

2. **Session Authentication (会话认证)**:
   - 使用 Django 的会话框架进行身份验证。用户登录后，系统将用户信息存储在服务器的会话中，并在浏览器中设置 sessionid Cookie。
   - 每次请求时，DRF 会检查该 Cookie，以验证用户身份。
3. **Token Authentication (令牌认证)**:
   - 用户登录后获得一个 token，该 token 需要在后续请求中作为请求头提供。
   - 该 token 是一个单独的字符串，一般存储在数据库中。
4. **Basic Authentication (基本认证)**:
   - 使用 HTTP 基本身份认证。在请求的 `Authorization` 头中包含用户的基本信息（以 `username:password` 编码）。
   - 简单，但相对不安全，因为没有加密，因此常与 HTTPS 结合使用。

认证方式对比

认证方式	描述	优点	缺点	适用场景
JWT Authentication	使用 JSON Web Token 进行身份验证。	无状态认证、跨域支持和分布式系统友好	令牌泄露风险、处理复杂性	移动应用、单页应用、微服务架构。
Session Authentication	使用 Django session 进行身份验证。	易于实现和使用，Django 原生支持	需要存储会话数据，状态保持	Web 应用、传统的服务器渲染应用。
Token Authentication	使用 Token 进行用户身份验证。	简单、无状态，无需存储会话	常常需要额外的数据库查询	API 接口、手机应用、轻量级服务端。
Basic Authentication	使用 HTTP Basic Auth 进行身份验证。	简单易用	不安全，需要 HTTPS	简化的 API、快速原型开发（受限于不暴露敏感数据）。

9.权限控制

9.1 介绍

背景：公司内部文件管理系统

在一家快速成长的技术公司里，员工需要频繁地创建、共享和编辑各类项目文档。文件管理系统是支持日常运营的重要工具。然而，随着公司规模的扩大，问题逐渐显现。

使用权限控制前：

1. **访问混乱**：
    - 所有员工都能查看和编辑所有文档，导致敏感信息泄漏的风险。
    - 误删或误改关键文件的情况时有发生，造成项目进度延误。
2. **责任不清**：
    - 因为缺乏访问日志，修改文档的责任难以追踪。
    - 管理层无法判断是谁删除或更改了重要文件。
3. **效率低下**：
    - 员工经常需要手工确认同一文件的最新版本，浪费大量时间。

使用权限控制后：

1. **访问权限明确**：
    - 每个员工只可访问与自己相关的项目文件，增强了信息安全性。
    - 管理员可以为不同部门设置不同的权限，保护敏感数据。
2. **清晰的责任划分**：
    - 改动记录会详细标注修改者和时间，使责任明确。
    - 相应的权限日志可以追踪文件的所有访问和更改行为。
3. **提高工作效率**：
    - 版本控制和权限分配结合，使得文件能在权限允许的范围内自动更新。
    - 员工只需关注其负责的内容，避免信息过载，提高专注度。

通过权限控制的实施，公司建立了一个更安全和高效的文件管理系统，不仅保护了敏感信息，还显著提升了全体员工的工作效率。

权限控制是一个确保系统中资源的安全性和完整性的关键概念。它包括对用户访问系统功能和数据的限制与管理。以下是权限控制的一些核心概念：

核心概念

1. **认证（Authentication）**：
    - 认证是验证用户身份的过程，通常通过用户名和密码组合来实现。它回答“你是谁？”的问题。
2. **授权（Authorization）**：
    - 授权是在身份验证之后，决定哪些资源可以由用户访问及如何操作的过程。它回答“你可以做什么？”的问题。
3. **访问控制（Access Control）**：
    - 访问控制是实现授权的一种机制，定义用户如何访问某些资源。
    - 主要涉及两个方面：用户身份（身份识别的对象）和角色（权限划分的对象）。

权限控制模型

1. **强制访问控制 (Mandatory Access Control, MAC)**：

    - 系统中央权威机构控制对资源的访问。用户无法改变访问控制设置，它们由安全策略决定。
    - 常见于高安全需求环境，如政府或军事系统。
2. **自主访问控制 (Discretionary Access Control, DAC)**:
    - 资源的所有者有权决定谁可以访问资源以及如何访问。
    - 用户可以进一步分配或撤消其他用户对其拥有资源的访问权限。
3. **基于角色的访问控制 (Role-Based Access Control, RBAC)**:
    - 用户通过其在系统中的角色获得权限，而角色则对应于特定的访问级别。
    - 提供了一种通过角色进行权限管理的简化方法。
4. **基于属性的访问控制 (Attribute-Based Access Control, ABAC)**:
    - 权限决策基于属性（主体、资源、环境属性等）的综合评估。
    - 提供了灵活且细粒度的权限管理。

关键组件

1. **用户/主体（Subject）**：
    - 系统中的实体，可以是用户、设备或程序，试图访问资源。
2. **资源/对象（Object）**：
    - 系统中需要保护的实体，比如文件、记录或服务。
3. **操作（Action）**：
    - 用户对资源执行的动作，比如读取、修改、删除等。
4. **策略（Policy）**：
    - 规则集，定义用户在何种情况下可对资源执行哪些操作。

实施权限控制的策略和最佳实践

1. **最小权限原则（Principle of Least Privilege）**：
    - 用户只应拥有执行其工作所需的最少权限。
2. **分离职责（Separation of Duties）**：
    - 任务划分，应控制权限，使得单个人无法独自完成整个过程，减少舞弊风险。
3. **定期审计和更新权限**:
    - 定期检查并更新权限以确保其仍然符合当前业务需求和安全标准。
4. **最小暴露原则**:
    - 仅应向用户暴露必须的信息和功能来减少访问不必要资源的风险。

通过以上策略和模型，权限控制不仅可以确保信息和操作的安全性，还可以提高系统的治理和管理效率。

9.2 权限和认证

权限（Authorization）和认证（Authentication）是信息安全领域中的两个关键概念，它们在确保系统安全和管理访问方面扮演着重要角色。虽然两者常常在一个整体的安全框架中共存，但它们是不同的阶段和过程。

	认证（Authentication）	**权限（Authorization）**
定义	验证用户身份的过程，确保用户是其声称的身份	决定经过认证的用户可以访问哪些资源并执行哪些操作
目的	确保用户身份的真实性	管理用户对资源的访问权限，确保安全地分配资源
实现方式	- 用户名与密码 - 生物识别 - 数字证书 - 双因素认证 (2FA)	- 访问控制列表 (ACL) - 角色权限 (RBAC) - 属性权限 (ABAC)
例子	- 输入密码登录电邮 - 使用指纹解锁手机	- 普通用户可以查看文件 - 管理员可以修改系统设置
执行顺序	先于权限，必须在进行权限判断前完成	随着认证之后执行，决定通过认证的用户可以执行的操作
主要关注对象	用户身份的验证	用户行为和资源访问的控制与约束

关系与区别

1. **执行顺序**:
   - **认证是首先执行的步骤。**在知道用户是谁之后，系统才能决定给予该用户什么权限。因此，认证先于授权。
2. **目的**:
   - **认证**: 确保用户身份的真实性。
   - **权限**: 确保用户只能访问其被允许的资源和功能。
3. **操作对象**:
   - **认证**: 处理用户身份的数据。
   - **权限**: 处理用户能执行哪些操作及访问哪些资源的规则和权限。
4. **实现场景**:
   - **认证**: 针对用户和系统之间的信任协议。
   - **权限**: 基于业务逻辑和安全策略进行资源分配和访问控制。

方面	ACL（访问控制列表）	RBAC（基于角色的访问控制）	ABAC（基于属性的访问控制）
基本概念	权限直接与用户或用户组关联，每个资源有自己的权限列表。	根据用户所属角色来分配权限，角色与权限直接关联。	基于用户、资源和环境属性的组合来决定访问权限。
对象	用户或用户组，权限与具体实体关联。	角色，用户通过角色获取权限。	属性，允许用属性和规则动态地评估访问权限。
灵活性	灵活性较低，修改复杂，需要具体设定。	具有更高的灵活性，通过少量角色变动即可反映在大量用户上。	高度灵活，可根据多种属性进行详细权限控制。
管理难度	随资源或用户数量的增加而变得复杂，管理成本较高。	因角色通常少于用户，简化了管理，减少错误风险。	需要管理属性和规则，设计和维护复杂度高。
调控方式	为每个用户设置特定权限。	分配角色来控制用户权限，通过角色定义一组权限。	通过制定复杂的规则来结合属性进行访问决策。
应用场景	适合小规模、简单的资源权限管理，如网络设备、文件系统。	适用于大中型组织，尤其是有明晰角色划分的场景，如企业管理信息系统。	适合复杂、动态环境的细粒度控制，如云计算和多样化用户群。
动态决策能力	静态决策，依赖于明确的用户和权限设置。	静态角色分配，但角色职责的变化能灵活调整用户权限。	动态决策，基于实际情况实时应用权限规则。
规则基础	主要基于列表条目，对每个资源直接定义访问者。	以组织角色为中心，通过角色定义访问权限。	基于策略语言的规则，支持复杂的条件和组合。
典型实现	文件系统权限控制，网络设备访问控制列表。	企业内部管理系统，ERP权限管理系统。	云服务平台，动态业务环境中的数据访问控制。

9.3 django权限和drf权限

9.3.1 django权限

以下是关于 Django 权限控制的精简总结表格：

要素	描述
用户（User）	每个用户对象有验证和权限属性。
组（Group）	一个组是权限的集合，多用户可以共享组权限。
权限（Permissions）	包括模型的增删改查权限，允许自定义。

操作	方法
检查权限	`user.has_perm('app_label.permission_codename')`
视图装饰器	`@permission_required('app_label.permission_codename')`
模板中检查权限	`{% if user.has_perm 'app_label.permission_codename' %}`
为用户添加权限	`user.user_permissions.add(permission)`
将用户添加到组	`user.groups.add(group)`

管理方式	说明
Django Admin	通过后台界面管理用户和组的权限。
编程方式	直接在代码中管理用户、组和权限，如增加或检查权限。

> Django 的权限控制是通过 Django 自带的认证系统（`django.contrib.auth`）实现的。这个系统支持用户、组和权限的细粒度管理，能够让你更灵活地控制用户对不同功能的访问。以下是 Django 权限控制的一些关键要素和实现方式：
>
> 权限控制概念
>
> 1. **用户（User）**：
>   - 每个用户都有一个用户模型实例，可以用来进行登录和认证。
>   - 用户可以拥有特定权限，可以通过用户对象来直接检查个体权限。
> 2. **组（Group）**：
>   - 组是用户权限的集合。
>   - 将用户分配到一个或多个组中，继而赋予用户组权限。
>   - 使用组可以更高效地管理一组用户的权限。
> 3. **权限（Permissions）**：
>   - 权限是一个预定义的权限集，通常与数据模型相关。
>   - 默认情况下，Django 会为每个模型自动创建以下权限：
>   - `add_<modelname>`
>   - `change_<modelname>`
>   - `delete_<modelname>`
>   - `view_<modelname>`（对于 Django 2.1 及以上版本）
>   - 你也可以定义自定义权限。
>
> 权限控制实现
>
> 使用权限
>
> 1. **指定权限**：
>   - 在模型中通过 `Meta` 类的 `permissions` 属性可以添加自定义权限示例：

```
from django.db import models

class MyModel(models.Model):
    # 模型字段定义
    ...

    class Meta:
        permissions = [
            ("can_publish", "Can Publish Content"),
        ]
```

2. **检查权限**：
   - 通过用户对象的方法来检查权限：

```
if user.has_perm('myapp.can_publish'):
    # 用户具有发布内容的权限
    ...
```

   - 在视图中可以使用 Django 的装饰器：

```
from django.contrib.auth.decorators import permission_required

@permission_required('myapp.can_publish', login_url='/login/')
def my_view(request):
    ...
```

3. **在模板中检查权限**：
   - 使用 `if` 模板标签来检查权限：

```
{% if request.user.has_perm 'myapp.can_publish' %}
    <!-- 用户可以发布内容 -->
    <a href="/publish/">Publish</a>
{% endif %}
```

管理权限

1. **在 Admin 界面管理**：
   - 通过 Django Admin 界面，可以为用户分配权限和组。
2. **使用代码管理**：
   - 可以通过编程的方式对用户、组和权限进行操作。如分配权限给用户：

```
from django.contrib.auth.models import User, Permission

user = User.objects.get(username='john')
permission = Permission.objects.get(codename='can_publish')
user.user_permissions.add(permission)
```

3. **组管理**：
   - 将用户添加到某个组：

```
from django.contrib.auth.models import Group

group = Group.objects.get(name='Editors')
user.groups.add(group)
```

- 通过组来检查用户是否有某些权限：

```
if user.groups.filter(name='Editors').exists():
    # 用户属于 "Editors" 组
    ...
```

小结

- **Django 的权限系统是基于用户、组和权限的三层结构设计**。它既可以实现简单的权限控制，也能支持复杂的权限策略。
- **权限检查**既可以在视图中实现，也可以在模板中实现，为开发者提供了多种控制途径。
- **通过编程的方式或者 Django Admin**界面，可以方便地管理用户、组和权限。
- **结合中间件和装饰器**，Django 能够实现复杂的认证和权限控制逻辑，确保应用的安全和稳定性。

这样，一个完整的权限管理系统让开发者能够对用户权限进行精细化的管理。

9.3.2 drf 权限

Django REST Framework (DRF) 是 Django 的一个强大的库，用于构建 Web APIs。DRF 提供了一些功能强大的组件来管理 API 的身份验证和权限控制。以下是关于 DRF 权限控制的详细描述：

1. **权限类**

DRF 使用权限类（Permission Classes）来检查视图请求是否具有执行特定动作的权限。每个权限类必须实现以下两个方法：

- `has_permission(self, request, view)`：检查整个视图的入口权限。
- `has_object_permission(self, request, view, obj)`：针对对象级别的权限检查。

2. **内置权限类**

DRF 提供了一些内置的权限类，包括：

- `AllowAny`：允许所有请求（默认）。
- `IsAuthenticated`：仅允许经过认证的用户访问。
- `IsAdminUser`：仅允许管理员用户访问。
- `IsAuthenticatedOrReadOnly`：未认证的用户可以读取信息，认证用户可以执行写操作。

3. **自定义权限类**

你可以借助权限类来自定义逻辑。例如：

```python
from rest_framework.permissions import BasePermission

class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.owner == request.user
```

这个自定义权限类 `IsOwner` 确保只有对象的所有者能执行修改或删除操作。

如何使用权限类

权限类可以在视图或视图集（ViewSet）中使用，通常设置在 `permission_classes` 属性中。例如：

```python
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated

class MyView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request):
        # 视图逻辑
        pass
```

全局设置

你可以在项目的全局设置文件中设置默认权限类：

```python
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
}
```

结合身份验证

- **Token Authentication & JWT**: DRF 支持多种身份验证机制，如 Token Authentication 和 JWT，通常这些机制会和权限类结合使用。
- **Session Authentication**: 使用 Django 的会话框架进行身份验证，并结合权限类管理用户的访问。

现实应用

- **安全性**: 通过将权限类应用于敏感资源以限制访问。
- **灵活性**: 利用自定义权限类实现复杂的访问策略。
- **简化管理**: 通过全局权限设置，规范化整个应用的访问控制。

DRF 权限控制是一个强大且灵活的工具，可以帮助开发者根据需求构建安全、健壮的 API，以确保敏感数据的安全访问。

9.3.3 权限对比

以下是关于Django和Django REST Framework (DRF) 权限控制的总结，对比这两者的特性和用法。

权限控制对比

特性	Django	DRF
基本权限模型	内建`User`、`Group`和`Permission`模型	自定义`Permission`类
权限应用层次	每个视图函数、类或模板级别	每个视图或视图集
对象级权限	限制性支持，需要自定义	支持通过`permission_classes`
内建权限类型	可用READ、WRITE等基本权限	可以自定义各种复杂权限类
扩展性	需要深度定制和模型扩展	高度可扩展，使用简洁
第三方支持	集成`django-guardian`等支持对象级权限	使用 Mixins、多个权限类
授权逻辑	手动检查用户权限	自动化，通过`has_permission`和`has_object_permission`
常见用法	角色控制、数据库查询限制	基于API请求的访问控制

Django 权限控制

1. **模型基础**：依赖于内建的`User`、`Group`和`Permission`模型，使用简单，适合基础的权限控制。
2. **应用场景**：多用于后台管理系统，传统MVC架构下的权限控制。
3. **扩展性**：需要手动扩展`User`模型和自定义逻辑来实现更复杂的需求。
4. **对象级权限**：可以通过第三方库如`django-guardian`获得更细粒度的权限控制。

DRF 权限控制

1. **灵活性**：利用DRF的`permission_classes`，开发者可以通过自定义权限类来实现复杂的API访问控制。
2. **API设计**：专为RESTful API设计，更适合前后端分离的项目。
3. **对象级权限**：天然支持，允许基于每个对象实例进行权限判断。
4. **综合性**：结合序列化和视图逻辑，使权限检查成为请求处理中的一部分。

综合对比

- **简便性**：Django的系统适合快速、简单的用户及组权限控制，适合不复杂的应用。
- **复杂场景支持**：DRF提供了更强的灵活性和适配复杂权限需求的能力，尤其是在API中，可以通过自定义权限类轻松实现复杂逻辑。

- **选择依据**：选择哪种方式是基于项目需求的复杂性和结构，在传统应用中Django的实现足够，而在现代API主导的应用中，DRF则提供了更合适的权限控制框架。

通过此对比表格和总结，你可以选择最适合你的项目需求的权限控制方案，充分利用各自的优点进行开发。

9.4 权限控制的实现

9.4.1 django权限控制概念

Django 的权限管理系统是通过 `django.contrib.auth` 内置应用实现的，这个应用提供了一整套用于用户认证和权限授权的机制。理解 Django 权限控制涉及到多个核心概念和模块。以下是这些核心概念的详细介绍：

对象和相关方法

1. User（用户）

描述: `User` 对象表示在 Django 应用程序中的一个用户。每个用户都有自己的用户名、密码等信息，还可以直接分配权限或者通过组获得权限。

相关方法:

- `user_permissions`: 一个用于直接管理用户权限的 Many-to-Many 关系，允许为用户添加或移除具体权限。

```
permission = Permission.objects.get(codename='add_article')
user.user_permissions.add(permission)  # 添加权限
user.user_permissions.remove(permission)  # 移除权限
```

- `has_perm(perm, obj=None)`: 检查用户是否拥有特定权限。

```
if user.has_perm('app_label.add_article'):
    print("User can add articles")
```

- `get_user_permissions(obj=None)`: 返回用户直接拥有的权限。
- `get_group_permissions(obj=None)`: 返回用户通过组继承的权限。
- `get_all_permissions(obj=None)`: 返回用户所有拥有的权限（直接的或通过组获得的）。

2. Group（组）

描述: `Group` 对象表示权限管理中的一个角色。通过为组分配权限，能够轻松管理和更新多个用户的权限。

相关方法:

- `permissions`: 一个用于管理组权限的 Many-to-Many 关系，用户通过加入组来继承这些权限。

```
permission = Permission.objects.get(codename='change_article')
group.permissions.add(permission)  # 为组添加权限
group.permissions.remove(permission)  # 为组移除权限
```

- `add(user)`: 将用户添加到组。

```
user.groups.add(group)  # 用户加入组
user.groups.remove(group)  # 用户退出组
```

3. Permission（权限）

描述: `Permission` 对象定义了可以分配给用户或组的权限。每一个权限与一个特定模型关联，用于描述某种操作或能力。

字段:

- `name`: 权限的描述性名称，用户在管理后台查看。
- `codename`: 用于标识权限的代码名称，格式上为 `"app_label.codename"`。
- `content_type`: 指出该权限所关联的具体模型。

方法:

- 虽然 `Permission` 没有直接的方法，但它在 `User` 和 `Group` 中通过 `user_permissions` 和 `permissions` 起作用。这些字段允许直接管理权限的增加和减少。

4. ContentType（内容类型）

描述: `ContentType` 用于标识权限与哪些模型相关联。在权限系统中，这是一个关键组件，它使得权限定义变得具体化和关联化。

使用:

- 通常与 `Permission` 对象共同使用，用于指出某个权限作用于哪个具体模型。
- `ContentType` 实例通过与其他模型和权限结合使用，实现细粒度的权限控制。

通过合并这些对象和方法，我们可以清晰地看到 Django 如何通过一系列关联操作来管理用户和组的权限，从而有效地控制对资源的访问和操作权限。

三种权限分配

权限分配方式的表格和总结：

权限分配方式	描述	优点	缺点
用户特定权限	权限直接分配给用户个体。 适用于需要为特定用户设置独特权限场景。	提供极高的灵活性，适合精细化、个性化管理。	管理难度较大，特别是在用户数量庞大时。
组权限	将权限赋予组，用户通过加入组来获得组的权限。 适合角色基于的权限管理。	简化权限管理，便于统一更新和维护,尤其是多个用户共享权限时。	个性化调整较困难，可能需要结合用户特定权限使用。

权限分配方式	描述	优点	缺点
混合使用	结合用户和组权限，通过组给予通用权限，再通过用户权限进行个性化调整。	结合两者优势，既可简化管理，又能实现个性化。	复杂度提升，需要关注两个维度上的权限设置同步。

总结

- **用户特定权限**: 适用于需要为某些用户提供特定的、个性化的权限控制场景。用户权限直接绑定用户对象，使得个性化权限管理成为可能。
- **组权限**: 可以看作是一种角色基于的权限管理方法。通过为组分配权限，任何一个加入该组的用户就会自动继承这类权限。即权限的分配可以通过管理组（角色）来实现，便于权限的大规模统一管理。
- **混合方式**: 在实际应用中，通常会结合两者：通过组实现大多数共享权限管理，通过用户权限进行个性化权限实现。这种方式允许在不失灵活性的情况下保持易于管理的结构。

使用混合策略可以同时享受到简化管理和精细控制的优点，是组织较大、权限需求复杂的应用中常见的最佳实践。

以下是展示三种权限分配方式的示例代码：

1. 用户特定权限

在这种方式中，我们直接将权限分配给一个特定用户。

```
from django.contrib.auth.models import User, Permission

# 获取或创建用户
user = User.objects.get(username='john_doe')

# 获取权限对象
add_article_perm = Permission.objects.get(codename='add_article')

# 分配权限给用户
user.user_permissions.add(add_article_perm)

# 检查用户是否拥有特定权限
if user.has_perm('app_label.add_article'):
    print("User has permission to add articles")
```

2. 组权限

在这种方式中，权限被赋予一个组，用户可以通过加入该组来继承组的权限。

```
from django.contrib.auth.models import User, Group, Permission

# 创建或获取组
editors_group, created = Group.objects.get_or_create(name='Editors')

# 获取权限对象
change_article_perm = Permission.objects.get(codename='change_article')
```

```
# 为组分配权限
editors_group.permissions.add(change_article_perm)

# 创建用户并将其添加到组
user = User.objects.create_user(username='jane_doe', password='password')
user.groups.add(editors_group)

# 检查用户是否通过组获得了权限
if user.has_perm('app_label.change_article'):
    print("User can change articles")
```

3. 混合使用

结合用户和组权限：大多数权限通过组进行管理，而特殊权限直接分配给用户。

```
from django.contrib.auth.models import User, Group, Permission

# 创建或获取组
writers_group, created = Group.objects.get_or_create(name='Writers')

# 创建权限对象
delete_article_perm = Permission.objects.get(codename='delete_article')
publish_article_perm = Permission.objects.get(codename='publish_article')

# 向组中添加权限
writers_group.permissions.add(delete_article_perm)

# 创建用户并分配权限
user = User.objects.create_user(username='alice_doe', password='password')
user.groups.add(writers_group)

# 为用户直接分配特殊权限
user.user_permissions.add(publish_article_perm)

# 检查用户通过组和单独分配获得的权限
if user.has_perm('app_label.delete_article'):
    print("User can delete articles")

if user.has_perm('app_label.publish_article'):
    print("User can publish articles")
```

总结

- **用户特定权限**: 用于精细化权限分派。
- **组权限**: 用于大规模、多用户的统一权限管理。
- **混合使用**: 在复杂的权限需求下，通过将共享权限与个别权限管理结合使用，实现最佳效果。

9.4.2 django权限控制实现

在Django中，权限管理通过内置的`auth`框架实现，该框架提供了用户、组和权限的管理功能。这里详细介绍Django权限控制的实现，包括组的划分和权限的配置。

> 默认权限的简单使用
>
> 使用 Django 的默认权限涉及到几个步骤：创建模型、分配权限、检查权限、以及使用这些权限控制用户访问。以下是如何使用默认权限的指南：
>
> 1. 确保模型自动创建默认权限
>
> Django 自动为每个模型创建默认的 `add`, `change`, 和 `delete` 权限。当你创建一个新的模型时，Django 在数据库中设置这些权限。
>
> 例如，一个简单的模型：
>
> ```
> from django.db import models
>
> class Article(models.Model):
>     title = models.CharField(max_length=100)
>     content = models.TextField()
> ```
>
> Django 自动为 `Article` 模型创建以下权限：
>
> - `add_article`
> - `change_article`
> - `delete_article`
>
> 2. 分配权限
>
> 通过管理后台
>
> - 登录 Django 管理后台，找到用户或用户组管理界面。
> - 编辑用户或用户组，将所需的权限分配给他们。
>
> 通过代码
>
> 你可以通过代码为用户分配权限。以下示例展示如何给用户授予添加、修改和删除 `Article` 的权限：
>
> ```
> from django.contrib.auth.models import User, Permission
>
> user = User.objects.get(username='your_username')
> add_perm = Permission.objects.get(codename='add_article')
> change_perm = Permission.objects.get(codename='change_article')
> delete_perm = Permission.objects.get(codename='delete_article')
>
> # 为用户添加权限
> user.user_permissions.add(add_perm, change_perm, delete_perm)
> user.save()
> ```
>
> 3. 检查权限
>
> 在视图或模板中，你可以检查用户是否具有某个权限，以决定是否允许执行某操作。

```python
from django.http import HttpResponseForbidden

def my_view(request):
    if not request.user.has_perm('yourapp.add_article'):
        return HttpResponseForbidden("你没有权限添加文章")
    # 其他业务逻辑
```

4. 使用权限

- **控制访问**：在视图函数中使用权限来控制用户对各种操作的访问。例如，只有具备 `add_article` 权限的用户才能看到添加文章的按钮或表单。
- **自定义装饰器**：为某些视图添加访问控制，可以自定义装饰器来简化权限检查。

总结

把权限系统集成进你的业务逻辑，以确保只有具备适当权限的用户可以访问相应的功能和数据。这使得应用更加安全，并符合业务需求。通过管理后台或代码编辑，确保为用户或组分配合适的权限。

1.用户和组

用户

在Django中，用户是通过 `User` 模型表示的，`User` 模型包含如下的重要字段：

- **用户名** (`username`)
- **密码** (`password`)
- **邮箱** (`email`)
- **是否活跃** (`is_active`)
- **是否管理员** (`is_staff`)
- **是否超级用户** (`is_superuser`)

超级用户（superuser）具有系统的所有权限，包括访问Django管理后台。

组

组是权限的集合，允许将多个权限赋予多个用户：

- **创建组**: 可以通过Django管理后台或编程方式创建组。
- **组与权限**: 权限可以赋予组（就像是角色），用户加入某个组则自动获得该组的所有权限。

通过依赖于组，可以简化用户权限的分配，当需要某一类用户共享权限时，使用组是非常有效的。

2. 权限管理

权限的基础

每个Django模型默认生成 `add`，`change`，`delete` 三个权限。除此之外，你可以定义自定义权限。

自定义权限

在模型中，可以通过`Meta`类的`permissions`属性定义自定义权限：

```
class Article(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()

    class Meta:
        permissions = [
            ("can_publish_article", "Can publish an article"),
            ("can_archive_article", "Can archive an article"),
        ]
```

操作权限

- **分配权限**: 权限可以被分配到用户或组。

  ```
  from django.contrib.auth.models import User, Group, Permission

  # 获取用户
  user = User.objects.get(username='john')

  # 获取或创建组
  editors_group, created = Group.objects.get_or_create(name='Editors')

  # 获取权限实例
  publish_permission = Permission.objects.get(codename='can_publish_article')

  # 赋予用户权限
  user.user_permissions.add(publish_permission)

  # 赋予组权限
  editors_group.permissions.add(publish_permission)

  # 将用户加入到组
  user.groups.add(editors_group)
  ```

- **检查权限**:

  ```
  # 检查用户的权限
  if user.has_perm('app_label.can_publish_article'):
      print("User can publish articles")
  ```

3. 使用Django Admin管理权限

Django提供了强大的管理后台，可以用于管理用户、组和权限：

- **创建/编辑用户和组**: 通过管理界面，管理员可以给用户分配或者取消权限，以及将用户添加到组或从组中移除。
- **管理权限**: 管理员可以查看和编辑每个组和用户的具体权限。

- **模型权限控制**: 在管理后台权限部分，指定哪个用户或组有权限访问、添加、修改或删除某个模型的数据。

4. 高级权限管理

扩展权限

可以通过继承 Django 的 `AbstractUser` 创建自定义用户模型来添加更多的用户属性并在用户模型中进行权限扩展。

对象级别权限

使用第三方库如Django Guardian可以实现对象级别的权限控制，这种方式可以对单个对象的访问权限进行详细的划分。

示例代码

假设你有一个博客系统，希望不同角色（如撰稿人、编辑和管理员）有不同的权限。你可以创建多个组并分配不同的权限：

```
from django.contrib.auth.models import Group, Permission

# 创建组
contributors_group, created = Group.objects.get_or_create(name='Contributors')
editors_group, created = Group.objects.get_or_create(name='Editors')
admin_group, created = Group.objects.get_or_create(name='Administrators')

# 分配权限，例如文章的添加、编辑权限
add_permission = Permission.objects.get(codename='add_article')
change_permission = Permission.objects.get(codename='change_article')

# 为撰稿人分配权限
contributors_group.permissions.add(add_permission)

# 为编辑分配权限
editors_group.permissions.add(add_permission, change_permission)

# 管理员默认有全部权限，也可以自定义添加更多专属权限
```

通过这些功能，Django 提供了灵活的权限管理系统，使开发者能够将复杂的用户角色和权限要求集成到Web应用程序中。根据应用场景，你可以设计适合的权限与组，确保资源的安全性和访问控制的可维护性。

5. RBAC实现-应用

详细说明如何在 PyCharm 中实现一个基于角色的访问控制（RBAC）系统。

项目需求背景

假设我们正在开发一个在线学习平台，名为“EduPlatform”，这个平台需要以下功能和权限管理：

- **角色和权限**:

- **学生（Student）**：可以查看课程、完成作业。
- **导师（Instructor）**：可以创建课程、发布公告、评阅作业。
- **管理员（Admin）**：可以管理用户、角色和系统设置。

第一步：在 PyCharm 中创建 Django 项目

1. **安装 Django**
   - 打开 PyCharm 终端并输入：

     ```
     pip install django
     ```

2. **创建项目**
   - 在 PyCharm 中，选择 "File" -> "New Project"。
   - 在项目类型中选择 "Django"，并为项目命名为 "EduPlatform"。
   - 确保选择创建新的虚拟环境。
   - 点击 "Create" 创建项目。

第二步：创建 Django 应用

1. **启动 "accounts" 应用**
   - 在项目根目录下的终端输入：

     ```
     python manage.py startapp accounts
     ```

2. **注册应用**
   - 在 `EduPlatform/settings.py` 中的 `INSTALLED_APPS` 中添加 `'accounts'`。

第三步：定义角色与权限模型

根据需求在 `accounts/models.py` 中建立相关模型：

```
from django.db import models
from django.contrib.auth.models import User

class Permission(models.Model):
    name = models.CharField(max_length=100, unique=True)
    codename = models.CharField(max_length=100, unique=True)

    def __str__(self):
        return self.name

class Role(models.Model):
    name = models.CharField(max_length=100, unique=True)
    permissions = models.ManyToManyField(Permission, blank=True)

    def __str__(self):
        return self.name
```

```
class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    roles = models.ManyToManyField(Role, blank=True)

    def __str__(self):
        return self.user.username

    def has_permission(self, permission_codename):
        for role in self.roles.all():
            if role.permissions.filter(codename=permission_codename).exists():
                return True
        return False
```

第四步：数据库初始化

1. **迁移数据库**
   - 执行以下命令生成并应用迁移：

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

第五步：创建超级用户

- 在项目根目录终端输入：

  ```
  python manage.py createsuperuser
  ```

第六步：初始化权限和角色数据

在 `accounts/admin.py` 或管理命令中初始化数据：

```
# accounts/admin.py

from django.contrib import admin
from .models import Permission, Role, UserProfile

admin.site.register(Permission)
admin.site.register(Role)
admin.site.register(UserProfile)
```

在 Django 管理后台或使用管理命令添加权限和角色：

```
# 创建权限示例
view_course = Permission.objects.create(name="View Course",
codename="view_course")
complete_assignment = Permission.objects.create(name="Complete Assignment",
codename="complete_assignment")
```

```
create_course = Permission.objects.create(name="Create Course",
codename="create_course")
post_announcement = Permission.objects.create(name="Post Announcement",
codename="post_announcement")
review_assignment = Permission.objects.create(name="Review Assignment",
codename="review_assignment")
manage_users = Permission.objects.create(name="Manage Users",
codename="manage_users")

# 创建角色并分配权限
student_role = Role.objects.create(name="Student")
student_role.permissions.set([view_course, complete_assignment])

instructor_role = Role.objects.create(name="Instructor")
instructor_role.permissions.set([view_course, create_course, post_announcement,
review_assignment])

admin_role = Role.objects.create(name="Admin")
admin_role.permissions.set([manage_users])
```

第七步：视图逻辑中引入权限控制

在 `accounts/views.py` 中实现权限检查：

```
from django.http import HttpResponse, HttpResponseForbidden
from .models import UserProfile

def course_detail_view(request):
    user_profile = request.user.userprofile

    if not user_profile.has_permission('view_course'):
        return HttpResponseForbidden("You do not have permission to view this
course.")

    return HttpResponse("Course Details: Welcome, Student!")

def create_course_view(request):
    user_profile = request.user.userprofile

    if not user_profile.has_permission('create_course'):
        return HttpResponseForbidden("You do not have permission to create a
course.")

    return HttpResponse("Course Creation: Welcome, Instructor!")
```

####

在 `accounts/views.py` 中，编写一个简单的登录视图：

```
from django.contrib.auth import authenticate, login
from django.shortcuts import render, redirect
from django.http import HttpResponse
```

```python
def login_view(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')

        # 使用 Django 的 auth 系统进行认证
        user = authenticate(request, username=username, password=password)

        if user is not None:
            # 登录用户，创建会话
            login(request, user)
            return redirect('home')  # 登录后重定向到首页或其他
        else:
            return HttpResponse("Invalid username or password")

    # GET 请求时显示登录表单
    return render(request, 'login.html')
```

在 `accounts/templates/accounts/` 中创建 `login.html` 文件：

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form method="post" action="{% url 'login' %}">
        {% csrf_token %}
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
        <br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
        <br>
        <button type="submit">Login</button>
    </form>
</body>
</html>
```

在 `EduPlatform/urls.py` 中添加登录视图的路由：

```
from django.contrib import admin
from django.urls import path
from accounts.views import course_detail_view, create_course_view, login_view

urlpatterns = [
    path('admin/', admin.site.urls),
    path('login/', login_view, name='login'),
    path('course_detail/', course_detail_view, name='course_detail'),
    path('create_course/', create_course_view, name='create_course'),
]
```

设置重定向页面

在 `EduPlatform/settings.py` 中，设置登录后的重定向页面：

```
LOGIN_REDIRECT_URL = '/'  # 或者根据需要设置为具体页面，这个是登录成功跳转
```

第八步：视图逻辑中引入权限控制

（与您之前的步骤类似，但将视图逻辑应用于认证用户。）

在 `accounts/views.py` 中为每个视图引入权限检查：

```
from django.http import HttpResponse, HttpResponseForbidden
from django.contrib.auth.decorators import login_required

@login_required
def course_detail_view(request):
    user_profile = request.user.userprofile

    if not user_profile.has_permission('view_course'):
        return HttpResponseForbidden("You do not have permission to view this
course.")

    return HttpResponse("Course Details: Welcome, Student!")

@login_required
def create_course_view(request):
    user_profile = request.user.userprofile

    if not user_profile.has_permission('create_course'):
        return HttpResponseForbidden("You do not have permission to create a
course.")

    return HttpResponse("Course Creation: Welcome, Instructor!")
```

第九步：运行和测试项目

1. **启动开发服务器**

```
python manage.py runserver
```

2. **测试登录和访问控制**
   - 访问 `http://127.0.0.1:8000/login/` 并使用创建的用户进行登录。
   - 登录成功后，尝试访问 `http://127.0.0.1:8000/course_detail` 和 `http://127.0.0.1:8000/create_course` 页面，验证权限控制是否生效。

通过这样的集成，您就可以在 Django 项目中添加简易的登录功能，以配合您的 RBAC 系统进行用户的权限管理和控制。这样，用户必须登录并拥有适当的角色和权限才能访问受保护的资源。

> 下面我将为你展示如何在用户注册时使用 `UserProfile` 模型进行处理，并确保你可以在注册时记录用户的角色（`roles`）。
>
> 1. 定义更新后的 `UserProfile` 模型
>
> 首先，你的 `UserProfile` 模型已经正确定义了用户与角色的关系。这里我们将确保在表单中能够处理角色的选择。
>
> ```
> # models.py
> from django.contrib.auth.models import User
> from django.db import models
>
> class Role(models.Model):
>     name = models.CharField(max_length=100)
>     permissions = models.ManyToManyField('Permission', blank=True)  # 假设有一个 Permission 模型
>
>     def __str__(self):
>         return self.name
>
>
> class UserProfile(models.Model):
>     user = models.OneToOneField(User, on_delete=models.CASCADE)
>     roles = models.ManyToManyField(Role, blank=True)
>
>     def __str__(self):
>         return self.user.username
>
>     def has_permission(self, permission_codename):
>         for role in self.roles.all():
>             if role.permissions.filter(codename=permission_codename).exists():
>                 return True
>         return False
> ```
>
> 2. 创建用户注册表单
>
> 我们需要更新注册表单，以便在注册时选择角色。你可以使用 `ModelChoiceField` 来让用户选择角色：
>
> ```
> # forms.py
> from django import forms
> from django.contrib.auth.forms import UserCreationForm
> from django.contrib.auth.models import User
> from .models import UserProfile, Role
> ```

```
class CustomRegistrationForm(UserCreationForm):
    roles = forms.ModelMultipleChoiceField(
        queryset=Role.objects.all(),
        widget=forms.CheckboxSelectMultiple,
        required=False
    )

    class Meta:
        model = User
        fields = ('username', 'email', 'password1', 'password2', 'roles')

    def save(self, commit=True):
        user = super().save(commit=False)
        if commit:
            user.save()
            # 创建或更新用户的 UserProfile
            user_profile = UserProfile.objects.create(user=user)
            user_profile.roles.set(self.cleaned_data.get('roles'))  # 设置用
户角色
            user_profile.save()
        return user
```

3. 视图函数

视图函数不变，所以我们仍然使用 `CustomRegistrationForm`：

```
# views.py
from django.shortcuts import render, redirect
from .forms import CustomRegistrationForm

def register_view(request):
    if request.method == 'POST':
        form = CustomRegistrationForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('success_url')  # 注册成功后重定向
    else:
        form = CustomRegistrationForm()

    return render(request, 'registration/register.html', {'form': form})
```

4. 数据库迁移

确保你的数据库反映出所有的变更，如果你还没有运行过迁移的话，别忘了执行：

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

5. 模板

在你的模板中，确保渲染 `roles` 字段以便用户可以选择角色：

```
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">注册</button>
</form>
```

总结

通过以上步骤，你可以在用户注册时，一并录入用户的角色信息。这种方法保证了用户基本信息与其关联的角色信息能够一起被存储。同时，使用 `ModelMultipleChoiceField` 使得角色选择的过程更加友好和清晰。这样，你的用户模型将更加完整，并能够支持更复杂的权限管理系统。

下面是补充的注册功能实现

仅使用 `ModelForm` 来处理用户注册，包括用户和用户角色的所有字段实现。

1. 定义模型

先确保你的 `UserProfile` 和 `Role` 模型已经定义好：

```
# models.py
from django.contrib.auth.models import User
from django.db import models

class Role(models.Model):
    name = models.CharField(max_length=100)
    permissions = models.ManyToManyField('Permission', blank=True)  # 假设有一个
Permission 模型

    def __str__(self):
        return self.name

class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    roles = models.ManyToManyField(Role, blank=True)

    def __str__(self):
        return self.user.username

    def has_permission(self, permission_codename):
        for role in self.roles.all():
            if role.permissions.filter(codename=permission_codename).exists():
                return True
        return False
```

2. 创建 `ModelForm`

我们将创建一个只使用 `ModelForm` 的注册表单，包含用户字段和用户角色字段。

```
# forms.py
from django import forms
from django.contrib.auth.models import User
from .models import UserProfile, Role
```

```python
class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput, required=True)
    password_confirm = forms.CharField(widget=forms.PasswordInput, required=True,
label='Confirm Password')
    roles = forms.ModelMultipleChoiceField(
        queryset=Role.objects.all(),
        widget=forms.CheckboxSelectMultiple,
        required=False
    )

    class Meta:
        model = User
        fields = ('username', 'email', 'password', 'password_confirm', 'roles')

    def clean(self):
        cleaned_data = super().clean()
        password = cleaned_data.get('password')
        password_confirm = cleaned_data.get('password_confirm')

        # 验证密码和确认密码是否相同
        if password and password_confirm and password != password_confirm:
            self.add_error('password_confirm', "The two password fields didn't
match.")

        return cleaned_data

    def save(self, commit=True):
        user = super().save(commit=False)
        user.set_password(self.cleaned_data['password'])  # 处理密码加密
        if commit:
            user.save()
            # 创建 UserProfile 对象并设置角色
            user_profile = UserProfile.objects.create(user=user)
            user_profile.roles.set(self.cleaned_data.get('roles'))  # 设置用户角色
            user_profile.save()
        return user
```

3. 视图函数

在视图中使用这个 `ModelForm` 来处理用户注册：

```python
# views.py
from django.shortcuts import render, redirect
from .forms import UserRegistrationForm

def register_view(request):
    if request.method == 'POST':
        form = UserRegistrationForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('success_url')  # 成功后的重定向
    else:
        form = UserRegistrationForm()
```

```
    return render(request, 'registration/register.html', {'form': form})
```

4. 模板更新

确保在模板中渲染这个表单，允许用户输入注册信息和选择角色：

```
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">注册</button>
</form>
```

5. 数据库迁移

如果你修改了模型，请确保执行迁移：

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

总结

以上示例展示了如何使用 `ModelForm` 来实现用户注册功能，包括用户信息和其关联的角色。通过这个方式，你可以实现不依赖于 `UserCreationForm` 的用户注册，同时将所有的字段通过模型表单进行管理。

这种方式清晰且灵活，能够很方便地扩展字段与功能，并且完全基于 Django 的表单与模型。这也是处理用户注册的一个非常有效的方法。

> 使用UserCreationForm可以简化密码加密和密码一致性验证
>
> - 即password1 和 password2
> - user.set_password(self.cleaned_data['password'])

9.4.3 drf权限控制实现

基础知识

基础开始介绍Django REST Framework（DRF）中的权限控制，包括相关的对象和方法。

DRF 权限控制的基础

在DRF中，权限控制主要通过`permissions`模块来实现。权限控制的核心是在视图中定义哪些用户可以访问哪些资源或进行哪些操作。DRF提供了一系列现成的权限类，你也可以基于这些类自定义自己的权限。

1. 主要对象

a. Permissions类

DRF的核心是`permissions`模块，其中包含了多个权限类。每个类都以不同的方式实现访问控制。以下是一些常用的权限类：

- `AllowAny`：允许所有用户访问，不管是认证用户还是匿名用户。
- `IsAuthenticated`：仅允许经过认证的用户访问。
- `IsAdminUser`：仅允许管理员用户访问。
- `IsAuthenticatedOrReadOnly`：认证用户可以进行任何操作，匿名用户只能进行读取（GET）操作。

b. 权限类的使用

权限类通常在视图中作为`permission_classes`属性来指定。例如：

```
from rest_framework import permissions
from rest_framework.views import APIView
from rest_framework.response import Response

class MyView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request):
        return Response({"message": "Welcome, authenticated user!"})
```

2.自定义权限类

如果内置的权限类不能满足需求，您可以自定义权限类。自定义的权限类需要从`BasePermission`继承，并重写`has_permission`或`has_object_permission`方法。

a. 自定义权限类示例

```
from rest_framework.permissions import BasePermission

class IsAdminUser(BasePermission):
    def has_permission(self, request, view):
        # 仅允许管理员用户访问
        return request.user and request.user.is_staff
```

3.主要方法

DRF权限类通常包含以下两个方法：

a. has_permission(self, request, view)

该方法用于检查用户是否有权限执行视图的操作。

- **参数**：
    - `request`：包含请求信息的请求对象。

    - `view`：当前请求的视图对象。
- **返回值**：返回 `True` 表示用户有权限，返回 `False` 表示用户没有权限。

b. has_object_permission(self, request, view, obj)

如果权限与具体对象相关（例如，编辑或删除特定资源），则使用此方法。

- **参数**：
    - `request`：包含请求信息的请求对象。
    - `view`：当前请求的视图对象。
    - `obj`：待检查权限的对象。
- **返回值**：返回 `True` 表示用户有权限访问该对象，返回 `False` 表示用户没有权限。

4.全局权限设置

DRF允许通过全局设置来指定一组默认权限。可以在`settings.py`中配置`DEFAULT_PERMISSION_CLASSES`：

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
}
```

5.结合其他功能

DRF的权限控制可以与其他功能结合使用，例如：

- **身份验证**：在判断用户权限前，通常需要先确认用户身份。DRF支持多种身份验证方法，如Token Authentication、Session Authentication等。
- **序列化**：在处理CRUD操作时，权限设置与序列化也可以结合起来，确保只有特定用户才能创建或修改序列化数据。

总结

- DRF中权限控制的核心是权限类，允许开发者定义用户能否执行特定操作。
- 内置的权限类提供了常用的权限控制方法，开发者可以自定义权限逻辑以适应特定需求。
- 使用权限控制的同时，可以结合身份验证和序列化实现更加严密的安全控制。

如果你希望深入某个具体的方面或者有其他相关问题，请告诉我！

RBAC实现-应用

项目名称：用户管理系统

项目需求概述

在中型企业中，进行有效的用户管理和权限控制是至关重要的。该系统需要支持不同用户角色（如Admin、Editor、Viewer），并确保用户根据其角色访问相应的资源。管理员能创建、查看、更新和删除用户；而编辑者只能更新用户信息，查看权限的用户只能查看用户信息。

基础实现

包含CRUD，角色权限说明

在我们的用户管理系统中，我们定义了以下角色及其相应的权限：

1. **Admin（管理员）**
    - 权限：
        - 创建用户
        - 查看所有用户
        - 更新所有用户的信息
        - 删除任何用户
        - 创建角色
        - 查看所有角色
        - 更新任何角色的信息
        - 删除任何角色

    描述：管理员拥有系统的完全控制权，能够管理用户和角色的所有操作，适用于负责管理系统的人员。
2. **Editor（编辑者）**
    - 权限：
        - 查看所有用户信息
        - 更新自己的信息
        - 更新其他用户的信息（但通常不允许更新用户的角色或删除用户）

    描述：编辑者通常负责更新内容和用户的信息，但不具备删除用户和管理角色的权限，适用于内容编辑或管理的人员。
3. **Viewer（观察者）**
    - 权限：
        - 查看自己的信息
        - 查看其他用户的基本信息（通常有限制，仅限于某些字段）

    描述：观察者仅有读取权限，专注于查看信息而不进行任何修改，适用于普通用户或初级员工。

项目结构

我们将设置如下的项目结构：

```
myproject/
|
├── myapp/
|   ├── migrations/
|   ├── __init__.py
|   ├── admin.py
|   ├── apps.py
|   ├── models.py
|   ├── permissions.py
|   ├── serializers.py
|   ├── tests.py
|   └── views.py
|
├── myproject/
|   ├── __init__.py
|   ├── settings.py
|   ├── urls.py
|   └── wsgi.py
|
└── manage.py
```

第一步：设置Django项目

1. 创建一个新的Django项目：

```
django-admin startproject myproject
cd myproject
```

2. 创建一个新的Django应用：

```
python manage.py startapp myapp
```

3. 在`myproject/settings.py`中注册新应用和DRF及Token：

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

添加 REST Framework 的设置（全局的，注册和登录注意不要加）：

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
}
```

第二步：建立用户和角色模型

在`myapp/models.py`中定义用户和角色模型，以及用户与角色的关系。

```
from django.contrib.auth.models import AbstractUser
from django.db import models

class Role(models.Model):
    name = models.CharField(max_length=50, unique=True)

    def __str__(self):
        return self.name

class CustomUser(AbstractUser):
    roles = models.ManyToManyField(Role, related_name='users')

    def __str__(self):
        return self.username
```

第三步：创建权限类

在`myapp/permissions.py`中定义基于角色的访问控制权限。各类权限实现如下：

```
from rest_framework.permissions import BasePermission

class IsAdminUser(BasePermission):
    def has_permission(self, request, view):
        return request.user.is_authenticated and
request.user.roles.filter(name='Admin').exists()

class IsEditorUser(BasePermission):
    def has_permission(self, request, view):
        return request.user.is_authenticated and
request.user.roles.filter(name='Editor').exists()

class IsViewerUser(BasePermission):
    def has_permission(self, request, view):
        return request.user.is_authenticated and
request.user.roles.filter(name='Viewer').exists()
```

第四步：创建序列化器

在`myapp/serializers.py`中定义用户和角色序列化器，以及用户注册序列化器。

```
from rest_framework import serializers
from .models import CustomUser, Role

class RoleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Role
        fields = ['id', 'name']

from rest_framework import serializers
from .models import CustomUser, Role

class RoleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Role
        fields = ['id', 'name']

class UserSerializer(serializers.ModelSerializer):
    roles = serializers.PrimaryKeyRelatedField(queryset=Role.objects.all(),
many=True, write_only=True)
    roles_detail = RoleSerializer(source='roles', many=True, read_only=True)

    class Meta:
        model = CustomUser
        fields = ['id', 'username', 'password', 'roles', 'roles_detail']
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        roles_data = validated_data.pop('roles', None)
        user = CustomUser.objects.create(**validated_data)
        user.set_password(validated_data['password'])
        user.save()
        if roles_data:
            user.roles.set(roles_data)
        return user

    def update(self, instance, validated_data):
        roles_data = validated_data.pop('roles', None)
        for attr, value in validated_data.items():
            setattr(instance, attr, value)
        if roles_data:
            instance.roles.set(roles_data)
        instance.save()
        return instance


class UserLoginSerializer(serializers.ModelSerializer):
    username = serializers.CharField(max_length=150)
    class Meta:
        model = CustomUser
        fields = ['username', 'password']
```

第五步：实现视图逻辑

在`myapp/views.py`中实现用户和角色的增删改查API的视图逻辑。

```
from rest_framework import status
from rest_framework.response import Response
from rest_framework.views import APIView
from rest_framework.authtoken.models import Token
from .models import CustomUser, Role
from .serializers import UserSerializer, UserLoginSerializer, RoleSerializer
from .permissions import IsAdminUser, IsEditorUser
from rest_framework.permissions import IsAuthenticated


class RegisterUser(APIView):
    def post(self, request):
        """
        注册新用户。
        """
        serializer = UserSerializer(data=request.data)
        if serializer.is_valid():
            user = serializer.save()
            return Response({'id': user.id, 'username': user.username},
status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)


class LoginUser(APIView):
    def post(self, request):
        """
        用户登录，返回Token。
        """
        serializer = UserLoginSerializer(data=request.data)
        if serializer.is_valid():
            username = serializer.validated_data['username']
            password = serializer.validated_data['password']
            user = CustomUser.objects.filter(username=username).first()
            if user and user.check_password(password):
                token, created = Token.objects.get_or_create(user=user)
                return Response({'token': token.key}, status=status.HTTP_200_OK)
            return Response({'detail': 'Invalid credentials'},
status=status.HTTP_401_UNAUTHORIZED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)


class LogoutView(APIView):
    permission_classes = [IsAuthenticated]  # 确保只有已认证的用户可以登出

    def post(self, request):
        # 获取当前用户的 Token
        token = Token.objects.get(user=request.user)
        token.delete()  # 删除 Token，以实现登出
```

```
        return Response({"message": "You have been logged out."},
status=status.HTTP_200_OK)


class UserList(APIView):
    permission_classes = [IsAdminUser]  # 只有管理员可以创建用户

    def get(self, request):
        """
        获取所有用户的列表。
        """
        users = CustomUser.objects.all()
        serializer = UserSerializer(users, many=True)
        return Response(serializer.data)

    def post(self, request):
        """
        创建一个新的用户。
        """
        serializer = UserSerializer(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)




class UserDetail(APIView):
    permission_classes = [IsAdminUser | IsEditorUser]  # 只有管理员和编辑者可以查看和
修改用户

    def get_object(self, pk):
        """
        获取特定用户的对象。
        """
        try:
            return CustomUser.objects.get(pk=pk)
        except CustomUser.DoesNotExist:
            return None

    def get(self, request, pk):
        """
        获取特定用户的详细信息。
        """
        user = self.get_object(pk)
        if user is None:
            return Response({"detail": "Not found."},
status=status.HTTP_404_NOT_FOUND)

        serializer = UserSerializer(user)
        return Response(serializer.data)

    def put(self, request, pk):
        """
```

```
        更新特定用户的信息。
        """
        user = self.get_object(pk)
        if user is None:
            return Response({"detail": "Not found."},
status=status.HTTP_404_NOT_FOUND)

        serializer = UserSerializer(user, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
        """
        删除特定用户。
        """
        user = self.get_object(pk)
        if user is None:
            return Response({"detail": "Not found."},
status=status.HTTP_404_NOT_FOUND)

        user.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)


class RoleList(APIView):
    permission_classes = [IsAdminUser]  # 只有管理员可以管理角色

    def get(self, request):
        """
        获取所有角色的列表。
        """
        roles = Role.objects.all()
        serializer = RoleSerializer(roles, many=True)
        return Response(serializer.data)

    def post(self, request):
        """
        创建新角色。
        """
        serializer = RoleSerializer(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)


class RoleDetail(APIView):
    permission_classes = [IsAdminUser]  # 只有管理员可以查看和修改角色

    def get_object(self, pk):
        """
        获取特定角色的对象。
        """
        try:
```

```
            return Role.objects.get(pk=pk)
        except Role.DoesNotExist:
            return None

    def get(self, request, pk):
        """
        获取特定角色的详细信息。
        """
        role = self.get_object(pk)
        if role is None:
            return Response({"detail": "Not found."},
status=status.HTTP_404_NOT_FOUND)

        serializer = RoleSerializer(role)
        return Response(serializer.data)

    def put(self, request, pk):
        """
        更新特定角色的信息。
        """
        role = self.get_object(pk)
        if role is None:
            return Response({"detail": "Not found."},
status=status.HTTP_404_NOT_FOUND)

        serializer = RoleSerializer(role, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
        """
        删除特定角色。
        """
        role = self.get_object(pk)
        if role is None:
            return Response({"detail": "Not found."},
status=status.HTTP_404_NOT_FOUND)

        role.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

第六步：配置URLs

在`myproject/urls.py`中配置API端点。

```
from django.contrib import admin
from django.urls import path
from myapp.views import RegisterUser, LoginUser, UserList, UserDetail, RoleList,
RoleDetail

urlpatterns = [
    path('admin/', admin.site.urls),
```

```
    path('register/', RegisterUser.as_view(), name='register'),
    path('login/', LoginUser.as_view(), name='login'),
    path('users/', UserList.as_view(), name='user-list'),
    path('users/<int:pk>/', UserDetail.as_view(), name='user-detail'),
    path('roles/', RoleList.as_view(), name='role-list'),
    path('roles/<int:pk>/', RoleDetail.as_view(), name='role-detail'),
    path('logout/', LogoutView.as_view(), name='logout'),
]
```

第七步：进行数据库迁移

ps:要先把继承的user在setting中设置

```
AUTH_USER_MODEL = 'myapp.CustomUser'
```

执行以下命令以创建数据库表：

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

第八步：创建超级用户和角色

创建一个超级用户以便进行测试（你可以分配角色）：

```
python manage.py createsuperuser
```

通过 Django 管理后台（http://127.0.0.1:8000/admin/）创建角色，例如 Admin、Editor 和 Viewer。

ps:注意要在admin中注册模型

第九步：运行Django开发服务器

启动开发服务器：

```
python manage.py runserver
```

第十步：测试API

使用工具（如Postman或cURL）测试API端点：

1. **POST /api/register/**: 注册新用户，示例请求体：

```
{
    "username": "newuser",
    "password": "password123",
    "roles": [1,2]
}
```

2. **POST /api/login/**: 用户登录，示例请求体：

```
{
    "username": "newuser",
    "password": "password123"
}
```

成功后将返回 Token。

3. **GET /users/**: 获取所有用户的列表。仅管理员可访问该端点。
4. **POST /users/**: 创建新用户，示例请求体：

```
{
    "username": "anotheruser",
    "password": "password456",
    "roles": [2]  # 请根据角色ID调整
}
```

5. **GET /users/{id}/**: 获取特定用户的详情。
6. **PUT /users/{id}/**: 更新用户信息，示例请求体：

```
{
    "username": "updateduser",
    "roles": [2]  # 请根据角色ID调整
}
```

7. **DELETE /users/{id}/**: 删除特定用户。
8. **GET /roles/**: 获取所有角色的列表。
9. **POST /roles/**: 创建新角色，示例请求体：

```
{
    "name": "Editor"
}
```

10. **GET /roles/{id}/**: 获取特定角色的详情。
11. **PUT /roles/{id}/**: 更新角色名称，示例请求体：

```
{
    "name": "Updated Editor"
}
```

这将更新指定 ID 的角色的名称。

12. **DELETE /roles/{id}/**: 删除特定角色。

您可以使用此请求来删除特定的角色，示例请求为：

```
DELETE /roles/{id}/
```

此请求将删除具有指定 ID 的角色。

总结

本项目展示了一个完整的用户管理系统，包括以下功能：

- **用户注册**：新用户可以注册帐户并选择角色。
- **用户登录**：用户使用用户名和密码进行身份验证，获取 Token。
- **用户 CRUD**：管理员可以创建、查看、更新和删除用户。
- **角色 CRUD**：管理员可以创建、查看、更新和删除角色。
- **角色权限管理**：根据角色，为不同用户分配适当的权限，从而控制他们可以执行的操作。

角色权限说明综述

- **Admin（管理员）**：拥有最高权限，能够对用户和角色进行完全管理。
- **Editor（编辑者）**：可以查看和更新用户信息，但不能创建或删除用户和角色。
- **Viewer（观察者）**：只能查看自己的信息和其他用户的基本信息，没有修改权限。

可能的扩展与改进

1. **动态角色分配**：实现 API 让 Admin 有能力在用户创建后直接为其分配角色。
2. **登录失败限制**：实现登录失败次数限制，以提高安全性。
3. **权限细化**：为 Editor 和 Viewer 提供更细致的权限控制。
4. **邮箱验证**：在注册过程中添加电子邮件验证功能。
5. **密码重置机制**：为用户提供重置密码的功能，以增强用户体验。

您可以在此基础上根据需求进行扩展和改进。如果您有任何进一步的问题或需要其他帮助，请随时告诉我！

> from rest_framework.authtoken.models import Token
>
> 这个引入的token是默认的一种模型
>
> 这个模型字段有user、key和create_time，其中key保证了其唯一性，创建只需要传入user模型即可，

补充功能

1.设置token有效时间，有效期内免登录

想在不引入其他库的前提下使用 Django REST Framework 的 Token 身份验证功能，并且希望为 Token 设置有效时间（即过期时间），您可以通过修改 Token 模型并进行必要的逻辑处理来实现这一点。以下是一种简单的方法：

第一 创建自定义 Token 模型

首先，您需要创建自定义 Token 模型并添加有效期字段：

```
from django.contrib.auth.models import AbstractUser
from django.conf import settings
from django.db import models
from django.utils import timezone
from datetime import timedelta
from rest_framework.authtoken.models import Token

class CustomToken(Token):
    expires_at = models.DateTimeField()

    def save(self, *args, **kwargs):
        if not self.pk:  # 创建新 Token 时设置过期时间
            self.key = self.generate_key()
            self.expires_at = timezone.now() + timedelta(hours=1)  # 默认过期时间 1
小时
        return super().save(*args, **kwargs)

    def is_expired(self):
        return timezone.now() > self.expires_at  # 判断 Token 是否过期
```

第二 创建自定义 Token 认证类

创建一个自定义 Token 认证类，用于处理 Token 的验证和有效性检验。

```
# yourapp/authentication.py
from rest_framework.authentication import TokenAuthentication as
BaseTokenAuthentication
from rest_framework.exceptions import AuthenticationFailed
from .models import CustomToken  # 导入您的自定义 Token 模型

class CustomTokenAuthentication(BaseTokenAuthentication):
    def authenticate(self, request):
        auth = super().authenticate(request)
        if auth is None:
            return None

        user, token = auth
        if token.is_expired():
            raise AuthenticationFailed('Token has expired.')
```

```
        return (user, token)
```

第三 创建自定义认证视图

然后，您可以创建一个自定义视图处理用户登录并生成 Token。

```
# yourapp/views.py
from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.response import Response
from django.utils import timezone
from datetime import timedelta
from .models import CustomToken  # 导入自定义 Token 模型


from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.response import Response
from django.utils import timezone
from datetime import timedelta
from .models import CustomToken  # 根据需要导入您的 Token 模型

class CustomAuthToken(ObtainAuthToken):
    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data)
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']

        # 获取或创建 Token
        token, created = CustomToken.objects.get_or_create(user=user)

        # 每次登录时更新 Token 的过期时间
        token.expires_at = timezone.now() + timedelta(hours=1)  # 更新过期时间
        token.save()

        return Response({
            'token': token.key,
            'expires_at': token.expires_at,
        })
```

第四 更新 URL 路由

确保将自定义认证视图添加到 URL 路由中，以便用户可以对其进行调用。

```python
# yourapp/urls.py
from django.urls import path
from .views import CustomAuthToken  # 导入自定义 Token 认证视图

urlpatterns = [
    path('api-token-auth/', CustomAuthToken.as_view(), name='api_token_auth'),  # 新增的 Token 认证路由
]
```

第五 使用自定义认证在受保护的视图中

在需要身份验证的视图中，您需要确保使用自定义的 Token 认证。

```python
# yourapp/views.py
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from .authentication import CustomTokenAuthentication

class SomeProtectedView(APIView):
    permission_classes = [IsAuthenticated]
    authentication_classes = [CustomTokenAuthentication]  # 使用自定义 Token 认证类

    def get(self, request):
        return Response({"message": "You are authenticated!"})
```

总结

通过以上步骤，您的项目应当正确配置自定义 Token 模型和认证类：

- **创建自定义 Token 模型**：添加 `expires_at` 字段以处理 Token 的过期。
- **在 `settings.py` 中设置**：指定自定义 Token 模型并使用自定义认证类。
- **实现登录功能**：使用自定义视图产生 Token。
- **保护需要认证的视图**：使用自定义 Token 认证类进行认证。

工作流程分析

1. **用户访问登录页面**：
    - 在访问登录页面时，首先检查本地存储中是否有 Token。
2. **有 Token 的情况**：
    - **发送请求**：向后端 API `verifyToken` 发送 `POST` 请求以检查 Token 的有效性。
        - **如果 Token 有效**：用户将被重定向到主页。
        - **如果 Token 无效**：用户将被重定向到登录页面，执行步骤 3。
3. **没有 Token 的情况**：
    - 直接跳转到登录页面，用户需要输入用户名和密码进行登录。
4. **用户登录**：

- 用户在登录页面输入凭据后，向后端 API `CustomAuthToken` 发送 `POST` 请求以进行身份验证。
  - **如果登录成功**：用户将被重定向到主页。
  - **如果登录失败**：捕获异常并向用户展示错误信息（例如，用户名或密码错误）。

流程图示

这里是您的流程图示意：

```
用户访问登录页
    └─ 检查 Token
        ├─ 有 Token
        |      └─ VerifyToken 检查有效性
        |           ├─ 有效：跳转到主页
        |           └─ 无效：跳转到登录页
        └─ 没有 Token：跳转到登录页

登录流程
  └─  CustomAuthToken 进行登录
        ├─ 成功：跳转到主页
        └─ 失败：展示错误信息
```

drf功能说明

1.user模型和自定义AbstractUser继承的模型以及request.user关系

2.drf的token模型和permission权限中的IsAuthenticated权限类

3.修改drf的token模型并替换原有token模型（添加有效时间等字段）

4.注意在一个序列器类中如果要使用另一个序列器类对应数据库中已有数据的操作

```
roles = serializers.PrimaryKeyRelatedField(queryset=Role.objects.all(),
many=True)

而不是
roles = RoleSerializer(many=True)
```

5.设置数据库字段数据唯一（unique=True）

6.序列器类的is_valid()在登录和注册的时候都会进行唯一性校验，应该注意注册需要，登录不需要。

为了方便您逐个测试 API，请参考以下测试表格。您可以在每个测试用例的输入、预期输出以及实际输出部分记录结果。此表格提供了不同 API 端点的描述以及测试用例。

API 测试表格

测试用例编号	API 端点	请求方法	输入数据	预期输出	实际输出	备注
TC1	`/api/register/`	POST	`{"username": "testuser", "password": "testpass", "roles": [1, 2]}`	`HTTP 201`，返回用户 ID 和用户名	成功	注册新用户
TC2	`/api/register/`	POST	`{"username": "testuser", "password": "testpass", "roles": [1]}`	`HTTP 400`，错误信息，用户名已存在	成功	重复注册同一个用户
TC3	`/api/login/`	POST	`{"username": "testuser", "password": "testpass"}`	`HTTP 200`，返回令牌（token）	成功	用户登录
TC4	`/api/login/`	POST	`{"username": "testuser", "password": "wrongpass"}`	`HTTP 401`，错误信息，凭据无效	成功	错误密码登录
TC5	`/api/users/`	GET	无	`HTTP 200`，返回用户列表	成功	获取用户列表
TC6	`/api/users/<user_id>/`	GET	无	`HTTP 200`，返回指定用户的详细信息	成功	获取用户详细信息
TC7	`/api/users/<user_id>/`	PATCH	`{"username": "newusername"}`	`HTTP 200`，返回更新后的用户详细信息	成功	更新用户信息
TC8	`/api/users/<user_id>/`	DELETE	无	`HTTP 204`，无返回体	成功	删除用户
TC9	`/api/roles/`	GET	无	`HTTP 200`，返回角色列表	成功	获取角色列表
TC10	`/api/roles/`	POST	`{"name": "New Role"}`	`HTTP 201`，返回新角色的详细信息	成功	创建新角色
TC11	`/api/roles/<role_id>/`	GET	无	`HTTP 200`，返回指定角色的详细信息	成功	获取角色详细信息

测试用例编号	API 端点	请求方法	输入数据	预期输出	实际输出	备注
TC12	`/api/roles/<role_id>/`	PUT	`{"name": "Updated Role"}`	`HTTP 200`，返回更新后的角色详细信息	成功	更新角色
TC13	`/api/roles/<role_id>/`	DELETE	无	`HTTP 204`，无返回体	成功	删除角色
TC14	`/api/token-auth/`	POST	`{"username": "testuser", "password": "testpass"}`	`HTTP 200`，返回测试用户的令牌和过期时间	成功	获取身份认证 token
TC15	`/api/token-verify/`	GET	`Authorization: Token <token>`	`HTTP 200`，返回令牌有效信息	成功	验证 Token 是否有效
TC16	`/api/logout/`	POST	`Authorization: Token <token>`	`HTTP 200`，返回“您已注销。”	成功	用户注销

> 在 HTTP 协议中，每一个响应都会包含一个状态码（status code），这个状态码用来表示请求的结果，帮助客户端理解请求是成功了、失败了，还是需要执行某种额外的操作。下面是一些常见的 HTTP 状态码类别及其具体含义，这些状态码对于理解和测试您的 API 非常关键：
>
> 1xx: 信息响应
>
> - **100 Continue**：服务器已接受请求的初始部分，客户端应继续发送请求的其余部分。
>
> 2xx: 成功
>
> 这些状态码表示请求被成功接收、理解和接受。
>
> - **200 OK**：请求成功，对 GET 请求，响应将包含请求的资源；对 POST 请求，响应将包含一些描述或操作结果的消息。
> - **201 Created**：请求成功并且服务器创建了新的资源，通常用于 POST 或 PUT 请求后的响应。
> - **204 No Content**：请求成功，但服务器不需要返回任何实体内容；可能只需要发送更新的元信息。
>
> 3xx: 重定向
>
> 这类状态码表示客户端需要采取进一步操作才能完成请求。
>
> - **301 Moved Permanently**：请求的资源已永久移动到新位置，服务器返回此响应时也会提供新的 URL。
> - **302 Found**：请求的资源现在临时从不同的 URI 响应请求。
>
> 4xx: 客户端错误

> 这些状态码表示请求可能出错，妨碍了服务器的处理。
>
> - **400 Bad Request**：服务器无法理解请求的格式，客户端不应该尝试再次使用相同的内容进行请求，而是修改请求数据。
> - **401 Unauthorized**：请求没有进行身份验证或验证未通过。
> - **403 Forbidden**：客户端没有权利访问所请求的内容。
> - **404 Not Found**：服务器找不到请求的资源，常用于无效的 URL 请求。
> - **405 Method Not Allowed**：指定的请求 HTTP 方法被服务器知道但被禁止使用。
>
> 5xx: 服务器错误
>
> 表示服务器在尝试处理请求时发生了错误。
>
> - **500 Internal Server Error**：服务器遇到了一个阻止它为请求提供服务的错误。
> - **502 Bad Gateway**：服务器作为网关或代理，从上游服务器收到无效响应。
> - **503 Service Unavailable**：服务器当前无法处理请求，这可能是由于超载或维护。

10.Form系列

10.1 介绍

Django 的 form 系列主要包括 `forms.Form` 和 `forms.ModelForm`。它们是 Django 的一部分，主要用于处理网页表单数据的验证和清理，以及对模型数据的直接操作。下面详细介绍这些表单的产生原因和使用场景：

1. **产生原因**：
   - **数据验证**：处理用户输入时，验证数据的正确性是非常重要的。Django form 提供了一个系统化的方法来进行数据验证。
   - **避免重复代码**：在 Web 开发中，处理表单是一个常见的任务。Django form 系统允许开发者通过声明性的表单字段来重用代码和逻辑，简化表单处理。
   - **安全性**：表单处理涉及到跨站请求伪造（CSRF）和注入攻击等安全问题。Django form 提供了内建的防护措施，例如 CSRF 令牌的管理。
2. **使用场景**：
   - **用户输入的创建和更新**：当需要从用户那里接收数据来创建或更新数据库中的记录时，`ModelForm` 可以自动将表单字段映射到模型字段。
   - **数据验证**：在用户提交表单数据之前验证数据的合法性，例如检查邮箱是否有效，或者密码是否符合特定格式。
   - **处理复杂的表单逻辑**：比如动态地改变表单字段、基于某些输入调整其他字段的需求等。

在没有使用 Django 的 form 系统的情况下，开发者可能会遇到以下一些问题或缺点：

1. **增加错误率**：

- 手动处理表单数据可能会引入错误，特别是在数据验证和清理阶段。每次都需要手动编写验证逻辑，易于遗漏或错误实现。

2. **代码重复**：
   - 没有使用 form 的情况下，可能需要在多个视图中重复相似的数据处理逻辑。这不仅增加了维护的难度，也违反了 DRY（Don't Repeat Yourself）原则。
3. **安全风险**：
   - 开发者需要手动管理安全性问题，如 CSRF 防护和 SQL 注入防护等，容易由于疏忽或缺乏经验导致安全漏洞。
4. **维护困难**：
   - 在没有表单系统的帮助下，对表单逻辑的更改可能会变得复杂和易错，尤其是在表单逻辑交织在业务逻辑中时。
5. **用户体验**：
   - 表单错误处理可能不一致或者实现不当，影响用户体验。例如，错误信息可能不清晰或者不易于理解，或者表单提交后用户需要重新输入大量数据。

Django 的表单系统提供了一个强大而灵活的工具，用于处理网页表单的数据输入、验证和处理。它简化了开发过程，增强了安全性，提高了代码的重用性和可维护性。虽然直接处理 POST 数据或使用其他方法（如直接使用序列化器处理 JSON 数据）在某些 API-重的应用中可能是可行的，但在传统的动态网页应用中，Django 的表单系统仍然是处理用户输入的首选方法。

10.2 CRUD操作

1. Form

`Form` 主要用于定义表单字段和验证逻辑，不直接与模型关联，因此在 CRUD 操作时，我们需要手动处理模型的数据赋值。

创建（Create）

```
from django import forms
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .models import Contact

class ContactForm(forms.Form):  # 特有：使用 Form 而非 ModelForm
    name = forms.CharField()
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)

    def clean_message(self):
        data = self.cleaned_data['message']
        if "spam" in data.lower():
            raise forms.ValidationError("No spam please!")
        return data

def contact_create_view(request):
    if request.method == 'POST':
```

```
        form = ContactForm(request.POST)
        if form.is_valid():
            # 特有：手动创建数据
            Contact.objects.create(**form.cleaned_data)
            return HttpResponseRedirect('/success/')
    else:
        form = ContactForm()

    return render(request, 'contact_form.html', {'form': form})
```

读取（Read）

```
from django.shortcuts import get_object_or_404

def contact_detail_view(request, pk):
    contact = get_object_or_404(Contact, pk=pk)  # 特有：手动获取对象
    return render(request, 'contact_detail.html', {'contact': contact})

def contact_list_view(request):
    contacts = Contact.objects.all()  # 特有：手动查询所有对象
    return render(request, 'contact_list.html', {'contacts': contacts})
```

更新（Update）

```
def contact_update_view(request, pk):
    contact = get_object_or_404(Contact, pk=pk)  # 特有：手动获取对象
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # 特有：手动更新数据
            contact.name = form.cleaned_data['name']
            contact.email = form.cleaned_data['email']
            contact.message = form.cleaned_data['message']
            contact.save()
            return HttpResponseRedirect('/success/')
    else:
        form = ContactForm(initial={
            'name': contact.name,
            'email': contact.email,
            'message': contact.message
        })

    return render(request, 'contact_form.html', {'form': form})
```

删除（Delete）

```
def contact_delete_view(request, pk):
    contact = get_object_or_404(Contact, pk=pk)  # 特有：手动获取对象
    if request.method == 'POST':
        contact.delete()
        return HttpResponseRedirect('/success/')
    return render(request, 'contact_confirm_delete.html', {'contact': contact})
```

2.ModelForm

`ModelForm` 直接与模型关联，可以自动处理创建、更新的逻辑，从而减少了大量手动代码。

创建（Create）

```
from django.forms import ModelForm
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .models import Contact

class ContactModelForm(ModelForm):  # 特有：使用 ModelForm
    class Meta:
        model = Contact
        fields = ['name', 'email', 'message']

def contact_create_view(request):
    if request.method == 'POST':
        form = ContactModelForm(request.POST)
        if form.is_valid():
            form.save()  # 特有：直接保存对象
            return HttpResponseRedirect('/success/')
    else:
        form = ContactModelForm()

    return render(request, 'contact_form.html', {'form': form})
```

读取（Read）

```
from django.shortcuts import get_object_or_404

def contact_detail_view(request, pk):
    contact = get_object_or_404(Contact, pk=pk)  # 与 Form 相同
    return render(request, 'contact_detail.html', {'contact': contact})

def contact_list_view(request):
    contacts = Contact.objects.all()  # 与 Form 相同
    return render(request, 'contact_list.html', {'contacts': contacts})
```

更新（Update）

```
def contact_update_view(request, pk):
    contact = get_object_or_404(Contact, pk=pk)  # 与 Form 相同
    if request.method == 'POST':
        form = ContactModelForm(request.POST, instance=contact)  # 特有：使用
instance 参数
        if form.is_valid():
            form.save()  # 特有：直接保存更新的对象
            return HttpResponseRedirect('/success/')
    else:
        form = ContactModelForm(instance=contact)  # 特有：自动填充实例数据

    return render(request, 'contact_form.html', {'form': form})
```

删除（Delete）

```
def contact_delete_view(request, pk):
    contact = get_object_or_404(Contact, pk=pk)  # 与 Form 相同
    if request.method == 'POST':
        contact.delete()
        return HttpResponseRedirect('/success/')
    return render(request, 'contact_confirm_delete.html', {'contact': contact})
```

3. 纯 Django ORM

`Form` 和 `ModelForm` 在原始的 Django ORM 基础的 CRUD 操作之上，增加了数据校验和处理的能力，其中最重要的一部分就是 `is_valid()` 方法。

Django 的 ORM 提供了直观且高效的方法来进行数据库的 CRUD（创建、读取、更新和删除）操作。以下是更完整的 CRUD 操作示例。

创建（Create）

1. **使用模型实例创建数据**：

```
from django.contrib.auth.models import User

user = User(username='newuser', email='user@example.com')
user.set_password('password123')  # 设置加密密码
user.save()  # 保存数据到数据库
```

2. **使用 `create()` 方法创建数据**：

```
user = User.objects.create(username='anotheruser', email='another@example.com')
user.set_password('password456')
user.save()
```

> `create()` 方法会直接在数据库中插入记录，但如果需要对字段进行额外处理（如加密密码），通常仍需手动调用 `save()`。

读取（Read）

1. **获取单个对象**：

```
user = User.objects.get(username='newuser')  # 如果用户名唯一，返回单个用户
```

> 注意：如果找不到用户或有多个用户匹配，将抛出 `DoesNotExist` 或 `MultipleObjectsReturned` 异常。

2. **获取多个对象（查询集）**：

```
users = User.objects.all()  # 获取所有用户
active_users = User.objects.filter(is_active=True)  # 只获取活跃用户
```

3. **使用查询集中的字段进行筛选**：

```
users_with_email = User.objects.filter(email__icontains='example.com')  # 邮箱包含 'example.com' 的用户
```

更新（Update）

1. **使用实例更新单个对象**：

```
user = User.objects.get(username='newuser')
user.email = 'newemail@example.com'  # 修改 email 字段
user.save()  # 保存更新后的数据到数据库
```

2. **使用 `update()` 方法批量更新**：

```
User.objects.filter(is_active=True).update(is_active=False)  # 将所有活跃用户的状态改为不活跃
```

> `update()` 方法适用于批量更新。需要注意，它不会调用 `save()` 方法，因此模型中的一些 `save` 相关逻辑（例如 `pre_save`、`post_save` 信号）不会触发。

删除（Delete）

1. **删除单个对象**：

```
user = User.objects.get(username='newuser')
user.delete()  # 删除这个用户
```

2. **批量删除多个对象**：

```
User.objects.filter(is_active=False).delete()  # 删除所有不活跃的用户
```

> `delete()` 方法可以用于批量删除。需要注意，删除操作会立即从数据库中移除数据。

操作类型	操作方法	示例代码
创建	`create()`	`User.objects.create(username='example', email='example@example.com')`
	手动创建实例并保存	`user = User(username='example')` `user.save()`
读取	获取单个对象（`get()`）	`User.objects.get(username='example')`

操作类型	操作方法	示例代码
	获取多个对象（`filter()`，`all()`）	`User.objects.filter(is_active=True)`

更新	手动修改并保存（`save()`）	`user = User.objects.get(username='example')`
		`user.email = 'newemail@example.com'`
		`user.save()`

	批量更新（`update()`）	`User.objects.filter(is_active=True).update(is_active=False)`
删除	单个删除（`delete()`）	`user = User.objects.get(username='example')` `user.delete()`
	批量删除	`User.objects.filter(is_active=False).delete()`

1. **`create()` 与 `save()` 的区别**：
   - `create()` 方法直接在数据库中插入一条记录，适用于快速创建对象的场景。
   - `save()` 方法可以进行更细粒度的控制，比如设置默认值、调用信号等。
2. **`update()` 与实例级 `save()`**：
   - 使用 `update()` 进行批量更新不会触发模型的 `save()` 方法中的逻辑（如信号）。
   - `save()` 适用于单个实例的保存，可以调用信号和自定义保存逻辑。
3. **异常处理**：
   - 使用 `get()` 方法时要注意异常，如果查询不到数据或查询到多个对象，将抛出 `DoesNotExist` 或 `MultipleObjectsReturned` 异常。因此在实际使用时需要加上异常处理逻辑。

希望这份重新整理的版本能帮助您全面理解 Django ORM 在 CRUD 操作上的完整性和灵活性。它提供了多个方式来满足各种不同的需求，从创建、读取、更新到删除，Django 的 ORM 都尽可能地用 Pythonic 的方式进行抽象，让开发者可以高效地管理数据。

4.序列化器实现

序列化器的定义

首先，定义一个序列化器来处理您的模型。假设您有一个简单的 `User` 模型：

```
# models.py
from django.db import models

class User(models.Model):
    username = models.CharField(max_length=100)
    email = models.EmailField()
    is_active = models.BooleanField(default=True)
```

对应的序列化器可能如下所示：

```
# serializers.py
from rest_framework import serializers
from .models import User

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username', 'email', 'is_active']
```

增（Create）

您可以使用序列化器来创建新的模型实例。以下是在视图中如何使用序列化器创建新用户的示例：

```
from rest_framework import status
from rest_framework.response import Response
from rest_framework.views import APIView
from .serializers import UserSerializer

class UserCreate(APIView):
    def post(self, request):
        serializer = UserSerializer(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)
```

删（Delete）

删除操作通常涉及检索一个特定的实例，然后调用其 `delete()` 方法：

```
class UserDelete(APIView):
    def delete(self, request, pk):
        user = User.objects.get(pk=pk)
        user.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

改（Update）

更新操作涉及检索模型实例并使用序列化器来更新字段：

```python
class UserUpdate(APIView):
    def put(self, request, pk):
        user = User.objects.get(pk=pk)
        serializer = UserSerializer(user, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

查（Read）

读取操作可以分为读取单一资源和读取资源列表：

```python
class UserDetail(APIView):
    def get(self, request, pk):
        user = User.objects.get(pk=pk)
        serializer = UserSerializer(user)
        return Response(serializer.data)

class UserList(APIView):
    def get(self, request):
        users = User.objects.all()
        serializer = UserSerializer(users, many=True)
        return Response(serializer.data)
```

总结

使用 DRF 的序列化器进行 CRUD 操作不仅简化了代码，并保持了数据处理的一致性和安全性。通过序列化器，您可以轻松地进行数据验证和字段级别的自定义，使得 API 开发变得更加高效和可控。

> 序列化器（Serializer）和模型表单（ModelForm）都是 Django 生态中处理模型数据的强大工具，但它们的使用场景和功能重点有所不同。以下是一个详细的对比表格，帮助您理解二者的主要差异：

特性	序列化器 (Serializer)	模型表单 (ModelForm)
主要用途	主要用于构建 RESTful APIs，处理 JSON, XML 等数据格式的输入输出。	主要用于构建网页表单，处理 HTML 表单的提交。
数据格式	可以处理多种格式，通常用于处理非HTML格式的数据，如JSON、XML。	专门处理HTML表单数据。
环境	通常用于 Django REST Framework 中，为 API 开发优化。	Django 的标准组件，用于处理服务器渲染的表单。
验证	提供全面的数据验证功能，可以轻易自定义验证逻辑和错误处理。	提供强大的表单验证，易于扩展和定制。
安全性	支持复杂的数据结构，易于实现安全措施，如 OAuth 和 Token 认证。	内建防护措施如 CSRF 保护，适用于网页表单。

特性	序列化器 (Serializer)	模型表单 (ModelForm)
客户端独立性	与客户端技术无关，可以服务于任何消费 REST API 的客户端（如移动应用、前端框架等）。	通常与 Django 模板一起使用，依赖于 Django 的模板系统。
CRUD 操作支持	通过 APIView 或视图集直接支持 CRUD 操作，与 HTTP 方法（GET, POST, PUT, DELETE）紧密集成。	主要通过视图和表单处理来实现 CRUD，通常需要手动配置 URL 和视图逻辑。
性能	设计用于处理大规模数据交互，优化了数据序列化和反序列化的性能。	主要优化了用户交互体验和表单渲染速度。

- **序列化器** 适合用于构建灵活的、与客户端独立的 API。它们支持复杂的数据结构和多种数据格式，使得在多客户端环境中共享数据变得简单和一致。
- **模型表单** 更适用于传统的 Django 网站开发，特别是需要服务器渲染的 HTML 表单。它们内建支持 Django 的安全机制，如 CSRF 保护，并且可以直接与 Django 模板集成。

选择使用哪一个取决于您的具体需求——是否重点在于构建 API 还是创建交互式的 Web 页面。

Django REST Framework (DRF) 提供了多种序列化器，从基本的 `Serializer` 类到更高级和专门的类，如 `ModelSerializer`。下面我们将通过一个表格和详细的文字解释来对这些序列化器进行分类和解释。

序列化器类型表格

序列化器类型	用途与特点
Serializer	基础序列化器。提供最大的灵活性，需要手动定义字段，适用于复杂的数据结构或特殊的验证需求。
ModelSerializer	自动将模型字段映射到序列化器字段，简化了创建和更新模型实例的过程。适用于直接与 Django 模型交互的场景。
HyperlinkedModelSerializer	类似于 `ModelSerializer`，但使用超链接来表示关系，而不是使用主键。适用于创建 HATEOAS 风格的 RESTful API。
ListSerializer	用于处理对象列表的序列化和反序列化，通常由框架自动使用，可以手动用于特定的需求。
BaseSerializer	更底层的序列化器基类，通常不直接使用，但可以扩展来创建完全定制的序列化器。

1. **Serializer**
    - 这是所有序列化器的基础，提供了最基本的序列化功能。
    - 开发者需要手动定义每个字段和对应的验证方法。

- 非常灵活，适合需要自定义处理逻辑的复杂场景。
- 示例：

```
from rest_framework import serializers

class AccountSerializer(serializers.Serializer):
    email = serializers.EmailField()
    username = serializers.CharField(max_length=100)
    date_joined = serializers.DateTimeField()

    def validate_username(self, value):
        if 'admin' in value.lower():
            raise serializers.ValidationError("Username may not
contain 'admin'")
        return value
```

2. **ModelSerializer**
   - 自动根据模型生成序列化器字段，简化了序列化器的定义过程。
   - 自动实现了默认的创建和更新的方法。
   - 适合直接与模型数据交互的 API，减少了大量的重复代码。
   - 示例：

```
from django.contrib.auth.models import User
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username', 'email']
```

3. **HyperlinkedModelSerializer**
   - 在 `ModelSerializer` 的基础上，使用超链接来表示与其他记录的关系。
   - 适用于客户端需要遵循链接进行操作的 API 设计风格。
   - 自动包括一个 `url` 字段，使用 DRF 的 `HyperlinkedIdentityField`。
   - 示例：

```
class
UserHyperlinkedSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ['url', 'username', 'email']
```

4. **ListSerializer**
   - 处理数据列表的序列化。
   - 通常由 DRF 在处理 `many=True` 时自动使用。
   - 可以自定义来对列表进行特殊处理。
   - 示例：

```python
class BulkUserSerializer(serializers.ListSerializer):
    def create(self, validated_data):
        users = [User(**item) for item in validated_data]
        return User.objects.bulk_create(users)
```

5. **BaseSerializer**
    - 最底层的序列化器类，提供了序列化和反序列化的核心功能，但没有默认的字段处理。
    - 一般用于非常定制的序列化需求，需要开发者完全重写序列化和反序列化的逻辑。
    - 示例：

```python
class CustomSerializer(serializers.BaseSerializer):
    def to_internal_value(self, data):
        return CustomObject(**data)

    def to_representation(self, obj):
        return {'data': obj.data}
```

10.3 序列化器vsModelForm

在 Django 和 Django REST Framework (DRF) 中，`ModelForm` 和 `Serializer` 都提供了强大的数据处理能力，特别是在数据验证和保存方面。尽管它们有许多相似的功能，如自动从模型生成字段，它们的使用场景和优缺点却有所不同。以下是对两者的比较：

ModelForm 和 Serializer 的优缺点

特性	ModelForm	Serializer
设计目的	主要设计用于 Django 的表单处理，特别是在网页中处理表单数据。	设计用于处理 JSON/XML 等数据格式，适用于构建 API。
数据支持	主要处理 HTML 表单数据。	支持多种格式，更适合处理复杂的数据结构和网络数据。
使用场景	用于网页应用，直接与 Django 模板集成。	用于构建 RESTful API，通常与 DRF 一起使用。
客户端独立性	依赖于 Django 的前端模板和视图系统。	与客户端技术无关，可以为任何客户端提供数据，包括移动应用、前端框架等。
安全性	内建了针对 CSRF 和其他网页表单相关安全问题的保护。	提供了对认证和权限的广泛支持，适合 API 安全需求。
灵活性	在处理标准表单交互方面非常高效，但在非表单交互的复杂数据处理上可能不够灵活。	高度灵活，可以定制复杂的数据验证和字段级逻辑。
性能	针对同步处理表单数据进行了优化。	为异步和高效的数据序列化/反序列化而设计。

特性	ModelForm	Serializer
复杂性处理	直接与 Django 模型集成，但在非表单数据处理上可能需要额外的定制。	处理复杂数据和自定义行为时更为灵活和强大。

使用场景分析

ModelForm

- **Web表单**：`ModelForm` 是处理 Django 网页应用中表单数据的理想选择。如果你在开发一个需要表单输入并直接创建或更新数据库记录的传统动态网站，使用 `ModelForm` 可以简化很多工作，因为它可以自动生成表单字段，并直接从 POST 请求中创建或更新模型实例。
- **表单验证**：`ModelForm` 提供了一个简洁的方式来定义和执行验证逻辑，这些验证直接基于模型的定义，如字段类型、长度限制、必填字段等。
- **直接与模板集成**：在 Django 的视图和模板中使用 `ModelForm` 可以轻松地渲染表单并处理表单提交，支持快速开发。

Serializer

- **API开发**：当你需要构建一个供移动应用或前端框架（如 React、Angular 或 Vue.js）使用的 API 时，`Serializer` 提供了一个强大的工具，尤其是与 DRF 配合使用。它不仅支持数据的序列化和反序列化，还能处理复杂的数据结构和关系。
- **数据格式多样性**：`Serializer` 适用于需要支持多种数据表示（如 JSON, XML）的应用。它可以轻松地与不同格式的数据交互，为各种客户端提供支持。
- **自定义数据处理**：在需要复杂的数据验证或当数据保存逻辑需要脱离单一模型简单映射的场景下，`Serializer` 提供了足够的灵活性来实现这些需求。

如果您正在开发一个前后端分离的项目，通常使用 Django REST Framework (DRF) 提供的 **序列化器 (Serializer)** 可以完全替代 **ModelForm**，因为它们都可以用于数据验证和处理，尤其在前后端分离的情况下，序列化器更加适合。以下是为什么序列化器可以完全替代 ModelForm 的详细分析，以及它们在不同使用场景下的优劣比较。

为什么序列化器可以替代 ModelForm?

1. **适合前后端分离的项目架构**
    - 在前后端分离的项目中，前端通常通过 RESTful API 与后端交互。这些 API 请求和响应通常是 JSON 格式的数据，这也是序列化器的强项。
    - **Serializer** 专门用于处理 JSON 等数据格式，可以轻松处理来自前端的数据，执行数据的验证、清理，并将其转换为适用于 Django 模型的 Python 数据类型。而 **ModelForm** 通常用于网页 HTML 表单的交互，不太适合与 JSON 格式的 API 请求直接交互。
2. **序列化和反序列化**
    - 序列化器在 REST API 开发中扮演双重角色：不仅可以将模型实例转换为 JSON 格式的数据以响应前端（**序列化**），还可以将从前端传入的 JSON 格式的数据验证并保存到数据库中（**反序列化**）。

- **ModelForm** 的作用相对更单一，它只能用来从 HTML 表单获取数据并保存到模型中，或者从模型中填充数据到 HTML 表单中。而对于 API 开发，ModelForm 并不能直接将数据转换为 JSON 格式。

3. **更灵活的字段定义和验证逻辑**
   - **Serializer** 允许对数据进行非常细粒度的定制和验证。例如，您可以很容易地在序列化器中实现复杂的字段验证逻辑、自定义字段表示和跨字段验证逻辑。
   - 虽然 **ModelForm** 也可以通过自定义方法（例如 `clean_field_name()` 或 `clean()`）实现数据验证，但它主要是为简化基于 HTML 表单的输入验证而设计的，因此它不如序列化器在处理复杂数据结构方面灵活。
4. **可以处理嵌套关系和多对多关系**
   - 序列化器可以轻松地处理模型之间的嵌套关系和外键关系。例如，您可以在序列化器中定义嵌套序列化器来表示外键关系的数据结构，这非常适合用于多表关联的数据操作。
   - **ModelForm** 虽然也可以处理多对多关系，但实现起来往往较为复杂且代码冗长，而序列化器可以更加直接和简洁地处理这些情况。
5. **用于权限和认证**
   - 序列化器能够与 **Django REST Framework** 的认证和权限系统无缝集成，例如 Token、JWT 认证、权限类等，从而更好地支持 RESTful 风格的 API 开发。
   - **ModelForm** 主要用于表单的交互和验证，不具备直接与 DRF 认证和权限系统集成的能力。

序列化器 vs ModelForm

特性	序列化器 (Serializer)	ModelForm
设计目的	用于 API 开发，处理 JSON 等数据格式的序列化和反序列化	用于传统的 HTML 表单与 Django 模板结合
适用场景	前后端分离、移动端开发、RESTful API	传统的 Django 服务器端渲染的表单
数据格式支持	支持 JSON, XML 等多种格式	只处理 HTML 表单数据
验证和清理	提供复杂的验证、跨字段验证、嵌套验证	提供标准的表单验证，适用于简单的数据输入
嵌套关系处理	支持嵌套序列化、外键、多对多关系等复杂数据结构	对于嵌套关系的支持有限，代码比较冗长
与前端交互	与 JSON 格式前端交互无缝结合	主要与 HTML 表单直接交互
数据保存	可以保存数据到数据库，类似于 ModelForm 的 `save()`	提供 `save()` 方法直接保存表单数据
序列化功能	可以将模型数据转换为 JSON 形式返回给前端	不能将模型数据直接转换为 JSON

什么时候应该选择序列化器而不是 ModelForm?

- **前后端分离的项目**：前端使用框架如 React、Vue.js、Angular，而后端用 Django REST Framework 提供 API，这种情况下应该使用 **Serializer**。
- **需要复杂的数据验证**：比如跨字段验证、嵌套数据结构处理、与其他表的关联等，**Serializer** 可以更灵活地实现这些功能。
- **需要 JSON 数据**：如果您的系统需要以 JSON 格式来交换数据（无论是 API 请求还是响应），序列化器是更适合的工具。
- **开发 RESTful API**：序列化器可以直接与 DRF 视图、认证和权限系统集成，可以轻松构建 RESTful 风格的 API。

ModelForm 仍然适合哪些场景?

尽管 **Serializer** 非常强大，但在某些情况下 **ModelForm** 仍然是合适的选择：

- **传统的网页应用**：如果您的项目是一个传统的服务器端渲染的 Django 网站，用户通过 HTML 表单与服务器交互，这时 **ModelForm** 可以为您提供便捷的表单生成、表单验证，以及与模型之间的关联。
- **简单的管理后台**：例如在 Django Admin 后台或自定义的基于 HTML 的管理工具中，**ModelForm** 可以很方便地用来生成表单并处理表单提交。

总结

- **序列化器 (Serializer)** 在处理 RESTful API 开发中可以完全替代 **ModelForm**，因为它能更好地适应前后端分离的开发模式，支持 JSON 格式数据的验证和处理，并能实现复杂的嵌套关系处理。
- **ModelForm** 更适合传统的服务器渲染的网页应用，用于从 HTML 表单到模型实例的处理，非常简洁。

如果您正在开发一个前后端分离的系统，建议全面使用 **序列化器** 来进行数据的验证和处理。它能帮助您更好地适应现代的开发需求，简化数据的验证、序列化和反序列化的逻辑，实现更加灵活和强大的 API。

11.视图类

视图类是 Django 和 Django REST Framework (DRF) 中处理 HTTP 请求和生成 HTTP 响应的核心组件。它们允许开发者以面向对象的方式定义应用程序的行为和逻辑。以下是关于视图类的一些基本概念和定义：

什么是视图类?

1. **定义**：
    - 视图类是一种用于处理特定 URL 请求的 Python 类。它封装了与请求和响应相关的逻辑，使得代码更易于管理和重用。
2. **功能**：
    - 视图类包含处理特定请求（如 GET、POST、PUT、DELETE）的逻辑。
    - 它们负责执行业务逻辑、查询数据库、处理用户输入和返回正确的 HTTP 响应。
    - 支持数据序列化，可以将 Python 对象转换为 JSON 或其他格式，以便在 API 中返回。

3. **类型**：
    - 在 Django 中，视图可以分为两大类：
        - **基于函数的视图 (Function-Based Views, FBV)**：以函数的形式定义视图逻辑。
        - **基于类的视图 (Class-Based Views, CBV)**：以类的形式定义视图逻辑，这种方式更灵活，支持继承和重用。
4. **优点**：
    - **组织性**：通过类将相关的逻辑组织在一起，便于理解和维护。
    - **重用性**：可以通过继承和组合实现代码的重用，减少重复代码。
    - **可扩展性**：可以方便地扩展和定制视图行为。

视图类的基本结构

在 Django 中创建一个基于类的视图通常包括以下步骤：

1. **导入视图类**：
   导入 Django 或 DRF 提供的视图类。
2. **定义类**：
   创建一个新的类，继承自 Django 或 DRF 提供的基本视图类。
3. **定义请求处理方法**：
   在类中定义处理请求的方法，例如 `get`、`post`、`put` 和 `delete`。
4. **返回响应**：
   在方法中执行相关逻辑，并返回一个 HTTP 响应对象。

以下是一个简单的 Django 视图类示例：

```
from django.http import JsonResponse
from django.views import View

class MyView(View):
    def get(self, request):
        data = {
            "message": "Hello, world!"
        }
        return JsonResponse(data)

    def post(self, request):
        # 处理 POST 请求数据
        # request.body 可以获取请求的原始数据
        return JsonResponse({"received": request.body}, status=201)
```

在这个示例中，我们定义了一个 `MyView` 类来处理 HTTP 请求。当收到 GET 请求时，它会返回一个 JSON 响应；当收到 POST 请求时，它会返回请求体的数据。

视图类是构建 Django 和 DRF 应用程序的重要组成部分，可以帮助开发者有效地处理 HTTP 请求、执行业务逻辑并返回响应。通过使用视图类，开发者可以组织代码、重用逻辑并提高可维护性。

11.1 常见视图类

Django 视图类详细说明

1. **View**
   - **描述**: 所有基于类的视图的基类，允许自定义请求处理逻辑。
   - **特点**:
     - 基础类，支持多种 HTTP 方法。
     - 可以使用装饰器或中间件进行额外的响应处理。
   - **示例代码**:

```python
from django.http import JsonResponse
from django.views import View

class CustomView(View):
    def get(self, request):
        data = {"message": "Hello, world!"}
        return JsonResponse(data)

    def post(self, request):
        return JsonResponse({"received": request.POST})
```

2. **TemplateView**
   - **描述**: 用于渲染 HTML 模板并返回静态网页。
   - **特点**:
     - 处理 GET 请求并渲染指定的 HTML 模板。
     - 可以通过 `get_context_data` 方法传递上下文数据。
   - **示例代码**:

```python
from django.views.generic import TemplateView

class HomePageView(TemplateView):
    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['greeting'] = "Hello, world!"
        return context
```

3. **ListView**
   - **描述**: 显示模型对象列表的视图。
   - **特点**:
     - 自动支持分页和过滤。
     - 使用 `get_queryset` 方法自定义要展示的数据。
   - **示例代码**:

```python
from django.views.generic import ListView
from .models import Book

class BookListView(ListView):
    model = Book
    template_name = "book_list.html"
    context_object_name = "books"
    paginate_by = 10  # 每页显示10本书

    def get_queryset(self):
        return Book.objects.filter(is_published=True)
```

4. **DetailView**
   - **描述**: 显示单个对象的详细信息。
   - **特点**:
     - 自动从 URL 中获取参数并查找对象。
     - 提供详细信息的视图，处理对象不存在的情况。
   - **示例代码**:

```python
from django.views.generic import DetailView
from .models import Book

class BookDetailView(DetailView):
    model = Book
    template_name = "book_detail.html"
    context_object_name = "book"
```

Django REST Framework 视图类详细说明

1. **APIView**
   - **描述**: DRF 中的基础视图，用于处理 RESTful API 请求。
   - **特点**:
     - 为处理 HTTP 方法提供基础支持。
     - 支持数据序列化和反序列化，处理权限和认证。
   - **示例代码**:

```python
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status

class HelloWorldView(APIView):
    def get(self, request):
        return Response({"message": "Hello, world!"})

    def post(self, request):
        return Response({"received": request.data},
status=status.HTTP_201_CREATED)
```

2. **GenericAPIView**
   - **描述**: 通用的 API 视图基础，提供快速实现 CRUD 操作的能力。

- **特点**:
  - 支持 QuerySet 和序列化，自定义数据处理。
  - 允许用户重写请求处理和数据序列化逻辑。
- **示例代码**:

```
from rest_framework.generics import GenericAPIView
from rest_framework.mixins import ListModelMixin, CreateModelMixin
from .models import Book
from .serializers import BookSerializer

class BookListCreateView(GenericAPIView, ListModelMixin,
CreateModelMixin):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

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

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

上面的代码除了GenericAPIView，还使用了Mixin

> 知乎资料：16.Django · DRF入门 - 通用视图类 GenericAPIView DRF超强视图封装！ - 知乎 (zhihu.com)

3. **ListAPIView**
   - **描述**: 用于返回模型对象列表的 API 视图。
   - **特点**:
     - 自动支持分页、过滤和排序。
     - 简单地扩展以满足特定需求。
   - **示例代码**:

```
from rest_framework.generics import ListAPIView
from .models import Book
from .serializers import BookSerializer

class BookListView(ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    pagination_class = YourPaginationClass  # 自定义分页类
```

4. **CreateAPIView**
   - **描述**: 用于创建新对象的 API 视图。
   - **特点**:
     - 自动处理请求数据的验证和序列化。
     - 返回创建的对象，支持见外部响应格式。
   - **示例代码**:

```python
from rest_framework.generics import CreateAPIView
from .models import Book
from .serializers import BookSerializer

class BookCreateView(CreateAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

5. **RetrieveAPIView**
   - **描述**: 用于检索单个对象的 API 视图。
   - **特点**:
     - 支持通过 URL 参数来自动查找对象。
     - 处理对象不存在时返回404错误。
   - **示例代码**:

```python
from rest_framework.generics import RetrieveAPIView
from .models import Book
from .serializers import BookSerializer

class BookDetailView(RetrieveAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

6. **UpdateAPIView**
   - **描述**: 用于更新现有对象的 API 视图。
   - **特点**:
     - 支持 PUT 和 PATCH 方法进行全量更新和部分更新。
     - 自动处理对象不存在时的异常。
   - **示例代码**:

```python
from rest_framework.generics import UpdateAPIView
from .models import Book
from .serializers import BookSerializer

class BookUpdateView(UpdateAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

7. **DestroyAPIView**
   - **描述**: 用于删除对象的 API 视图。
   - **特点**:
     - 提供简单的 API 接口来删除单个对象。
     - 自动处理找不到对象时返回404错误。
   - **示例代码**:

```python
from rest_framework.generics import DestroyAPIView
from .models import Book

class BookDeleteView(DestroyAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer  # 可选，通常不需要
```

8. **RetrieveUpdateDestroyAPIView**
   - **描述**: 同时处理检索、更新和删除操作的 API 视图。
   - **特点**:
     - 整合了多个基本操作的功能，提高了代码的复用性。
     - 适合需要在一个视图中同时支持查询和修改的场景。
   - **示例代码**:

```python
from rest_framework.generics import RetrieveUpdateDestroyAPIView
from .models import Book
from .serializers import BookSerializer

class BookDetailUpdateDeleteView(RetrieveUpdateDestroyAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

视图类实现细节

GenericAPIView封装

GenericAPIView是继承基础视图类的APIView的通用视图类，只需配置好类熟悉，就可以实现一整套的增删改查

原本的APIView实现增删改查

```python
# 使用APIView实现

class BookDetailView(APIView):
    """
        单个Book查询，Url中传入pk(主键id)
    """
    def get(self, request, pk):
        book = Book.objects.get(pk=pk) # 用了一次objects.get
        serializer = BookSerializer(instance=book)
        return Response(serializer.data)

    def patch(self, request, pk):
        book = Book.objects.get(pk=pk) # 又用了一次objects.get
        serializer = BookSerializer(instance=book,data=request.data)
        if serializer.is_valid():
```

```
            serializer.save()
            return Response(serializer.data)
        else:
            return Response(serializer.errors,status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
        Book.objects.get(pk=pk).delete() # 又用了一次objects.get
        return Response()


class BookListView(APIView):
    """
        书籍列表查询,不传入id,直接创建/查询所有
    """
    def get(self, request):
        books = Book.objects.all()
        serializer = BookSerializer(instance=books,many=True)
        return Response(serializer.data)

    def post(self, request):
        data = request.data
        serializer = BookSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
            return
Response(serializer.errors.values(),status=status.HTTP_400_BAD_REQUEST)
```

对于上述代码存在非常多重复代码，且逻辑为增删改查

对于不同字段实现增删改查区别在于：

- 模型的区别
- 序列化器的区别

此外逻辑没有任何区别

因此有了GenericApIView通用视图类

GenericAPIView实现增删改查

```
class BookListView(GenericAPIView):
    """
        多个书籍的类视图
    """

    # 使用GenericAPIView，需要定义两个类属性(名字不能错)
    # queryset: 模型类对象的.all()
    queryset = Book.objects.all()
    # serializer_class: 序列化器类
    serializer_class = BookSerializer

    # 这就相当于，把上面我们说的模型以及序列化类进行了类级别的定义
    # 我们后面的逻辑代码，实现一次，后面再去继承的时候，只需要把类属性改了就行了
```

```
    def get(self, request):
        # self.get_serializer ---> 返回的就是我们类属性中的 serializer_class 的实例化对
象
        serializer = self.get_serializer(instance=self.get_queryset(), many=True)
        # self.self.get_queryset() ---> 返回的就是 类属性中的queryset --->
Book.objects.all()
        return Response(serializer.data)

    def post(self, request):
        serializer = self.get_serializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)


class BookDetailView(GenericAPIView):
    # 和上面一样的定义方法
    queryset = Book.objects.all()
    serializer_class = BookSerializer

    def get(self, request, pk):
        serializer = self.get_serializer(instance=self.get_object())
        # get_object就等同于Book.objects.all().filter(pk=pk)
        # 相当于帮你完成了查询操作
        return Response(serializer.data)
    def put(self, request, pk):
        serializer = self.get_serializer(instance=self.get_object(),
data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

可以看到，配置了模型和序列化器，使其可以进行通用配置

GenericAPIView中有些自定义的方法

- self.get_queryset()，返回lei属性中的queryset
- self.get_serializer()，类属性中的序列化器的实例化对象传入的数据的返回结果
- self.get_object()，相当于原本的queryset又进行了get查询（因此需要提供pk）

Mixin封装

Mixin是一种插件，脱离于所有子类数据

使用Mixin可以提供增删改查的各种方法

使用Mixin实现增删改查

```
from rest_framework.mixins import CreateModelMixin, UpdateModelMixin,
ListModelMixin, RetrieveModelMixin,DestroyModelMixin
class AuthorListView(CreateModelMixin, ListModelMixin, GenericAPIView):
    """
        因为ListView需要使用两种方式:
            1.get请求获取全部数据
            2.post创建数据
        所以,直接继承CreateModelMixin(创建数据插件),ListModelMixin(列表模型插件)
    """
    queryset = Author.objects.all()
    serializer_class = AuthorSerializer

    def get(self, requset):
        """
            使用了Mixin，我们直接返回一个self.list就可以
        """
        return self.list(requset)

    def post(self, request):
        # 直接返回self.create就是实现了我们的创建数据
        return self.create(request)
```

Mixin源码解析

Post->CreateModelMixin源码->create方法

```
class CreateModelMixin:
    """
    Create a model instance.
    翻译: 创建一个模型示例
    """
    def create(self, request, *args, **kwargs):
        """
            其实可以看到，这里的实现，和我们之前自己去create是类似的
            所以其实就是帮我们直接把创建的方法封装起来了
        """
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        self.perform_create(serializer)
        headers = self.get_success_headers(serializer.data)
        # 最后返回的就是一个Response
        return Response(serializer.data, status=status.HTTP_201_CREATED,
headers=headers)
```

get->ListModelMixin源码->list方法

```
class ListModelMixin:
    """
        ListModelMixin和我们之前自己写的GET也是非常类似
    """
    def list(self, request, *args, **kwargs):
        queryset = self.filter_queryset(self.get_queryset())

        page = self.paginate_queryset(queryset)
        if page is not None:
            serializer = self.get_serializer(page, many=True)
            return self.get_paginated_response(serializer.data)

        serializer = self.get_serializer(queryset, many=True)
        return Response(serializer.data)
```

1. CreateModelMixin 创建数据的Mixin(一般对应POST请求)
   ---> 返回 self.create(request) 即可
2. ListModelMixin 获取多个数据的Mixin(对应GET请求无主键id的情况)
   ---> 返回 self.list(request) 即可
3. UpdateModelMixin更新数据的Mixin(对于PUT/PATCH请求)
   ---> 返回 self.update(request) 即可
4. RetrieveModelMixin 获取单个数据的Mixin(对应GET请求有主键id的情况)
   ----> 返回 self.retrieve(request) 即可
5. DestroyModelMixin 删除数据的Mixin(对应DELETE请求)
   ----> 返回 self.destroy(request) 即可

为了避免在继承的时候写太多，drf有APIView+Mixin的类（其实是GenericAPIView）

- LiistCreateAPIView
- RetrieveUpdateDestroyAPIView

记住List、Create、Retrieve、Update、Destroy这几个对应功能即可

ViewSet

这个ViewSet的出现是为了方便接口的编写

> **之前我们使用普通的或者封装的APIView开发接口时，总是需要分两类情况处理**
>
> 1. 有主键 /user/20/ (id=20的用户)
> 2. 无主键 /user/ (实现get所有或者post创建)
>
> **我们总是需要分两种接口去写，然后还要进行两次类的封装，其实代码的冗余还是比较高，那有没有办法，直接写一个类就能实现呢?**
>
> **ViewSet类就是DRF给我们的解决方案!**

首先定义url

```
from django.urls import path,re_path
from BookManage.views import *
urlpatterns = [
 path("booknew/<int:pk>/",BookView.as_view({"get":"get_one_data","delete":"delete
_date","put":"update_data"})),
 # 对于有pk的，我们 get -> get_one_data | delete -> delete_date | put ->
update_data
 path("booknew/",BookView.as_view({"get":"get_all_data","post":"create_data"})),
 # 对于没有pk的，get -> get_all_data | post -> create_data
]
```

其次定义视图类

```
from rest_framework.viewsets import ViewSet # 导入ViewSet
class BookViewSet(ViewSet):
    """
        继承ViewSet
    """
    def get_all_data(self, request): # 对应get无pk方法(查询多个)
        return Response('返回所有书籍')

    def create_data(self, request): # 对应post创建方法
        return Response('创建一个书籍')

    def get_one_data(self, request, pk): # 对应get有pk方法(查询单个)
        return Response(f'返回pk={pk}的书籍')

    def update_data(self, request, pk): # 对应put/patch有pk更新方法
        return Response(f'更新pk={pk}的书籍')

    def delete_date(self, request, pk): # 对应delete有pk删除
        return Response(f'删除pk={pk}的书籍')
```

注意的是ViewSet的其中的方法是没有简化的，即没有继承GenericAPIView和Mixin

ModelViewSet

> **其实ViewSet与APIView类似，他也有对应的GenericViewSet，并且也支持继承Mixin**
>
> **所以为了我们的方法便，这里直接引入ModelViewSet**
>
> **其实ModelViewSet本质上就是帮我们继承了所有的Mixins，让我们只需要在定义路由的时候，分别定义不同的方法就够了！**

```
class ModelViewSet(mixins.CreateModelMixin,
                   mixins.RetrieveModelMixin,
                   mixins.UpdateModelMixin,
                   mixins.DestroyModelMixin,
                   mixins.ListModelMixin,
                   GenericViewSet):
    # 它就是帮我们把五种Mixins全部继承，并且继承了GenericViewSet
    # GenericViewSet和我们的GenericAPIView几乎一样，就是多了可以自定义不同方法使用哪个函数
    pass
```

首先创建路由url

```
from rest_framework import routers # 导入路由函数
route = routers.DefaultRouter() # 创建路由
route.register('book', viewset=BookView)
# 注册路由，viewset是我们创建的ViewSet视图(必须是继承自viewset/派生类的视图，不能是APIView)
# 注册的第一个参数填写名称即可
# 填写book则会关联 /book/ 以及 /book/<pk>/

urlpatterns = []
urlpatterns += route.urls # 将创建的路由添加到列表



--------也可以这样--------


urlpatterns = [
    path('',include(route.urls)), # 用include进行路由分发,可以用到名称空间
]
```

然后编写视图类

```
from rest_framework.viewsets import ModelViewSet
class BookViewSet(ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

    def get_all_data(self, request):  # 对应get无pk方法(查询多个)
        return self.list(request)
```

```
    def create_data(self, request):  # 对应post创建方法
        return self.create(request)

    def get_one_data(self, request, pk):  # 对应get有pk方法(查询单个)
        return self.retrieve(request)

    def update_data(self, request, pk):  # 对应put/patch有pk更新方法
        return self.update(request)

    def delete_date(self, request, pk):  # 对应delete有pk删除
        return self.destroy(request)
```

下面是Minxi插件实现的CRUD的对应

简化方法	HTTP 方法	功能描述
create	POST	用于创建一个新的资源。例如，添加新的投票、问题或评论。
list	GET	用于获取资源的列表。例如，获取所有问题、所有投票记录或所有评论。
retrieve	GET	用于获取单个资源的详细信息。例如，获取某个特定问题的详细信息。
update	PUT/PATCH	用于更新现有资源的所有或部分信息。例如，更新投票类型或问题标题。
destroy	DELETE	用于删除资源。例如，删除特定的投票、问题或评论。

功能细节

1. **create（POST）**：
    - **功能**: 创建一个新的资源。通常在请求体中包含要创建的对象的详细信息。
    - **示例**: 用户提交新的投票数据，发送请求到端点 `/votes/`。
2. **list（GET）**：
    - **功能**: 返回资源的列表，通常提供分页功能。
    - **示例**: 请求 `/questions/` 来获取所有问题的列表。
3. **retrieve（GET）**：
    - **功能**: 返回单个资源的详细信息，通过资源的 ID 定位。
    - **示例**: 请求 `/questions/1/` 获取 ID 为 1 的问题的详细信息。
4. **update（PUT/PATCH）**：
    - **功能**: 更新现有资源的数据，`PUT` 通常要求提供完整数据，`PATCH` 允许部分更新。
    - **示例**: 通过发送数据到 `/votes/1/` 来更新 ID 为 1 的投票的类型。
5. **destroy（DELETE）**：

- **功能**: 删除指定的资源。
- **示例**: 请求 `/votes/1/` 来删除 ID 为 1 的投票。

以下是一些不适合使用 Django REST Framework 自带的 CRUD 的场景，并详细解释了每种情况的原因。

1. 复杂的业务逻辑

不适合使用的原因:

- **默认逻辑不足**: 自带的CRUD方法通常是直接操作数据库的，无法处理复杂的业务规则（如用户资格验证、字段依赖等）。
- **自定义性差**: 当业务逻辑涉及多个步骤或条件时，默认的 `create`、`update` 方法无法满足条件。

示例场景: 在一个银行系统中，转账操作需要检查账户余额是否足够、转账的合法性等。这些逻辑无法通过简单的 CRUD 方法实现。

2. 自定义字段或响应格式

不适合使用的原因:

- **返回格式固定**: 默认的CRUD方法返回的是标准化的响应，这可能会限制灵活性，例如需要返回额外的统计信息或额外的元数据。
- **定制性限制**: 当需要的响应格式与标准不符时，必须重写方法以达到特定的输出效果。

示例场景: 在电商平台的产品列表页面，可能需要同时返回产品的数量、分类信息等附加数据，而默认的 `list` 方法只返回列表数据。

3. 多模型交互

不适合使用的原因:

- **事务性问题**: 默认的CRUD方法通常只针对单一模型的操作，无法处理多个模型之间的关系，特别是在涉及到原子性（事务）时。
- **复杂的依赖关系**: 当创建或更新某个模型需要同时操作其他模型（如减少库存、更新状态等）时，默认的方法不支持这种复杂交互。

示例场景: 在一个订单处理系统中，创建订单时需要同时更新产品的库存数量，如果库存不足则需要取消订单。自带的CRUD方法无法处理这种情况下的复杂逻辑。

4. 安全与权限控制

不适合使用的原因:

- **权限检测不足**: 默认的权限和认证机制无法满足特定安全性需求，尤其是在多角色应用中，需要根据用户角色动态决定权限。
- **灵活性差**: 自带的CRUD方法可能不允许细粒度的权限控制，无法针对不同用户、角色或请求场景进行精细化的权限设计。

示例场景: 在企业内部管理系统中，普通用户只应能够查看自己的数据，而管理员可以管理所有数据，必须实施更严密的权限检查。

5. 性能优化

不适合使用的原因:

- **性能瓶颈**: 默认的CRUD方法可能未针对性能问题进行优化，特别是对于大量数据的处理。
- **复杂查询**: 当需要进行复杂的数据库查询（如联接、聚合、分页等）时，默认方法可能不适用。

示例场景: 在一个社交媒体应用中，检索用户的朋友列表可能需要复杂的关联合并（JOIN），而直接使用自带的 `list` 方法会导致性能问题。

6. 特殊需求的端点

不适合使用的原因:

- **不满足标准操作**: 默认CRUD方法定义了一套标准的RESTful操作，但一些功能性需求可能并不符合这个标准，如报告生成、数据导出等。
- **单一职责原则**: 当某些功能与模型操作逻辑脱节时，使用原 CRUD 逻辑可能会导致接口变得臃肿和杂乱。

示例场景: 在一个数据分析平台上，用户可能需要下载分析的报告，这个操作与数据的基本创建和删除逻辑没有直接关系，适合定义一个独立的API。

7. 组合逻辑

不适合使用的原因:

- **缺乏灵活性**: 默认的CRUD操作适合简单的分开处理，无法支持多项操作的组合，比如同时处理多个资源的创建。
- **业务场景限制**: 批量操作、合并操作等复杂逻辑必须自定义实现，以保证业务的完整性和正确性。

示例场景: 在员工管理系统中，可能需要同时上传和创建多个员工的资料，这时可自定义一个批量创建的方法，而不是分别调用 `create` 方法。