DRF

原创

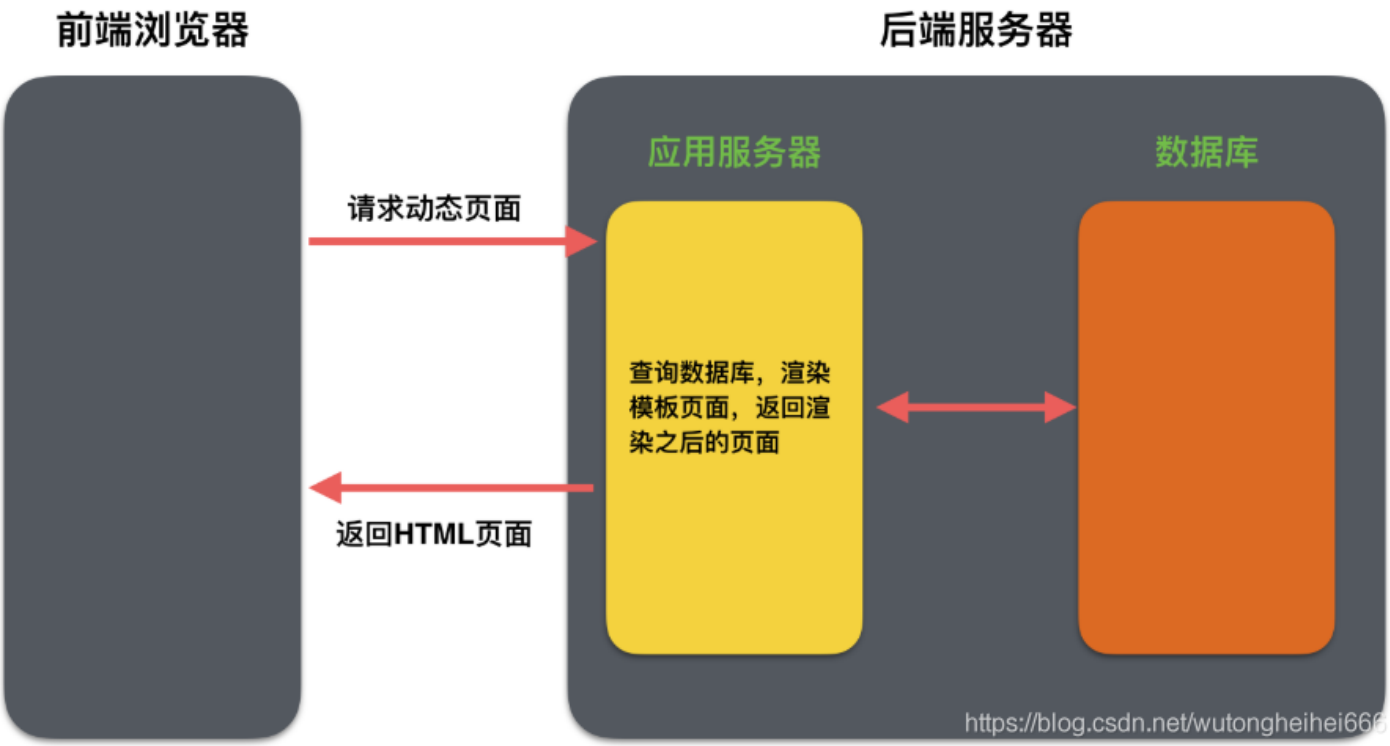
wutongheihei666 于 2020-06-05 22:37:04 发布

Web开发模式

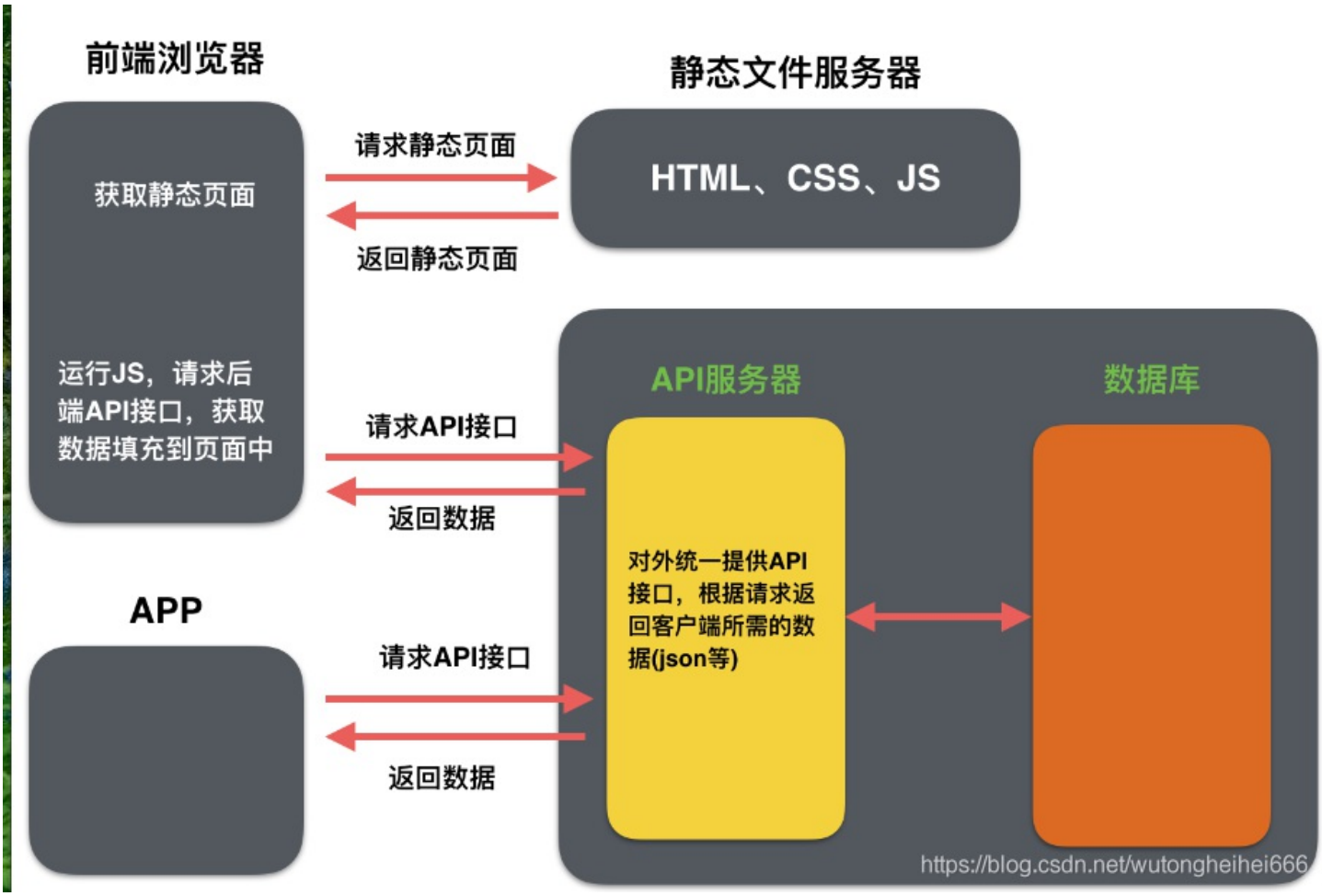
在Web项目开发中，有两种开发模式：

```
前后端不分离
前后端分离
```

1 前后端不分离



2前后端分离



注：

后端仅返回前端所需的数据，至于数据怎么进行展示，由前端自己进行控制，前端与后端的耦合度很低。
API: 在前后端分离开发模式中，我们通常将后端开发的每个视图都称为一个接口或者**API**。

3.开发模式对比

区别：

前后端不分离：完整的**html**页面是在后端生成的，后端给前端返回完整的页面，前端只是进行展示。
前后端分离：完整的**html**页面是在前端生成的，后端只给前端返回所需的数据，前端将数据填充在页面上。

优缺点：

开发模式	优点	缺点
前后端不分离	利于SEO(搜索引擎优化)	仅适合于纯网页的应用
前后端分离	可以针对于不同的客户端	不利于SEO(搜索引擎优化)

> SEO: 搜索引擎优化，是针对搜索引擎排名的一种优化手段，一个基础的优化点就是通过返回给客户端的页面内容上多加对应的关键字来提高搜索引擎排名。

小结：

前后端不分离：前端看到的效果是由后端控制的，后端模板渲染返回给客户端完整的页面。

前后端分离：后台只返回前端所需的数据，至于数据怎么展示，由前端自己进行控制。

前后端不分离适合于纯网页的应用，前后端分离可以适合于不同的客户端。

RESTful风格

```
在进行API接口设计时，不同的开发人员可能有不同的设计风格，风格迥异。

那是否存在一种统一的接口设计方式，被广大开发人员所接受呢？
```

答: 这就是被普遍采用的**RESTful API**设计风格。

1.Restful风格设计-关键点

1.1 URL路径

```
URL地址尽量使用名词复数，不要使用动词。

错误的例子：

 /getProducts
 /listOrders


正确的例子

  GET /products:将返回所有产品信息
  POST /products:将新建商品信息
  GET /products/4：将获取产品4
  PUT /products/4：将更新产品4


路径又称"终点"（endpoint），表示API的具体网址，每个网址代表一种资源（resource）
```

1.2 请求方式

```
访问同一个URL地址，采用不同的请求方式，代表要执行不同的操作。
常用的HTTP请求方式有下面四个：
```

请求方式	说明
GET	获取资源数据(单个或多个)
POST	新增资源数据
PUT	修改资源数据
DELETE	删除资源数据


```
例如：

 GET /books：列出所有图书数据
 POST /books：新建一本图书数据
 GET /books/<id>/：获取某个指定的图书数据
 PUT /books/<id>/：更新某个指定的图书数据
 DELETE /books/<id>/：删除某个指定的图书数据
```

1.3 过滤信息

过滤参数可以放在查询字符串中。

```
在访问API接口获取数据时，可能需要对数据进行过滤

常见的参数：

  ?limit=10：指定返回记录的数量
  ?offset=10：指定返回记录的开始位置。
  ?page=2&pagesize=100：指定第几页，以及每页的记录数。
  ?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序。
```

1.4 响应数据

针对不同操作，服务器向用户返回不同的响应数据。

```
一般遵循以下规范：

 1. 获取一组数据，返回一组数据
 2. 获取指定数据，返回指定数据
 3. 新增数据，返回新增的数据
 4. 修改数据，返回修改的数据
 5. 删除数据，返回空
```

1.5 响应数据格式

服务器返回的响应数据格式，应该尽量使用JSON

1.6 响应状态码

服务器向客户端返回的状态码和提示信息，常见的状态码如下：
  200 OK - [GET/PUT]：服务器成功返回用户请求的数据
  201 CREATED - [POST]：用户新建数据成功。
  204 NO CONTENT - [DELETE]：用户删除数据成功。
  400 INVALID REQUEST - [POST/PUT]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作
  404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。。
  500 INTERNAL SERVER ERROR - [*]：服务器发生错误，用户将无法判断发出的请求是否成功。

2.Restful风格设计-其他

2.1 域名

应该尽量将API部署在专用域名之下。

https://api.example.com
如果确定API很简单，不会有进一步扩展，可以考虑放在主域名下。

https://www.example.com/api/

2.2 版本

应该将API的版本号放入URL。

http://www.example.com/api/1.0/foo

http://www.example.com/api/1.1/foo

http://www.example.com/api/2.0/foo
另一种做法是，将版本号放在HTTP头信息中，但不如放入URL方便和直观。Github采用这种做法。

2.3 错误处理

如果状态码是4xx，服务器就应该向用户返回出错信息。

```
{
    error: "<error message>"
}
```

2.4 超媒体

RESTful API最好做到Hypermedia（即返回结果中提供链接，指向其他API方法），使得用户不查文档，也知道下一步应该做什么。

比如，Github的API就是这种设计，访问api.github.com会得到一个所有可用API的网址列表。

```
{
    "current_user_url": "https://api.github.com/user",
    "authorizations_url": "https://api.github.com/authorizations",
    // ...
}
```
从上面可以看到，如果想获取当前用户的信息，应该去访问api.github.com/user，然后就得到了下面结果。

```
{
    "message": "Requires authentication",
    "documentation_url": "https://developer.github.com/v3"
}
```
上面代码表示，服务器给出了提示信息，以及文档的网址。

3.总结

要点

```
url地址 url地址尽量使用名词复数，不要使用动词

请求方式 执行不同的操作，采用不同的请求方式：
GET(获取)
POST(新增)
PUT(修改)
DELETE(删除)

过滤参数 过滤参数放在查询字符串中

响应数据 获取多个数据时，返回对应的多个数据
获取单个数据时，返回对应的单个数据
新增数据时，将新增的数据返回
修改数据时，将修改的数据返回
删除数据时，返回空

响应数据格式 JSON

响应状态码 200(获取或修改成功)
201(新增成功)
204(删除成功)
404(资源找不到)
500(服务器出错)
```

接口设计

API设计思路

```
1.确定请求方式和URL地址
2.确定请求参数和格式
  1.通过url地址传递参数，如：/books/(?P<pk>\d+)/
  2通过查询字符串传参参数，如：/books/?page=<页码>&pagesize=<页容量>
  3.通过请求体传递参数，如：post表单数据和json数据
  通过请求头传递参数
3.确定响应数据和格式以及响应状态码
```

总结

API接口设计过程：

```
    请求方式和请求url地址
    请求参数的传递
    响应数据及响应状态码
```

REST接口开发的核心任务

REST接口开发的核心任务

```
分析上节图书管理的5个API接口，可以发现，在开发REST API接口时，视图中做的最主要有三件事：

  将请求的数据（如JSON格式）转换为模型类对象
  操作数据库
  将模型类对象转换为响应的数据（如JSON格式）
```

1.序列化Serialization

在以上操作中，涉及到两个概念：**序列化和反序列化。**

1.1 序列化

将程序中的一个数据结构类型转换为其他格式（字典、JSON、XML等）

例如将Django中的模型类对象转换为字典或JSON字符串，这个转换过程我们称为序列化

```
如：

queryset = BookInfo.objects.all()
book_list = []
# 序列化
for book in queryset:
    book_list.append({
        'id': book.id,
        'btitle': book.btitle,
        'bpub_date': book.bpub_date,
        'bread': book.bread,
        'bcomment': book.bcomment,
        'image': book.image.url if book.image else ''
    })
return JsonResponse(book_list, safe=False)
```

1.2 反序列化
将其他格式（字典、JSON、XML等）转换为程序中的数据

例如将JSON字符串或字典转换保存为Django中的模型类对象，这个过程我们称为反序列化

```
如：
json_bytes = request.body
json_str = json_bytes.decode()

# 反序列化
book_dict = json.loads(json_str)
book = BookInfo.objects.create(
    btitle=book_dict.get('btitle'),
    bpub_date=book_dict.get('bpub_date')
)
```

在开发REST API时，视图中要频繁的进行序列化与反序列化的操作。

2.RestAPI核心工作说明

开发REST API接口时，我们在视图中在做的最核心的事是：

**将数据库数据序列化为前端所需要的格式，并返回。
将前端发送的数据反序列化保存到模型类对象，并保存到数据库中。
**

3. 总结

**序列化：将对象转换为字典或者json的过程
反序列化：将字典或json转换保存到对象中的过程**

RestAPI核心工作：

**将数据库数据序列化为前端所需要的格式，并返回
将前端发送的数据反序列化保存到模型类对象，并保存到数据库中**

Django REST framework 简介

1.简介
Django REST framework 框架是一个用于构建Web API 的强大而又灵活的工具。通常简称为DRF框架

DRF框架是建立在Django框架基础之上，由Tom Christie大牛二次开发的开源项目。

2.作用

Django REST framework可以帮助我们大大提高REST API的开发速度。

注：DRF框架内容封装了很多东西，目的就是**简化开发代码的编写**

提高API接口的开发速度

```
举例：

1）在序列化与反序列化时，虽然操作的数据可能不同，但是过程却是相似的，这部分操作DRF框架进行了封装。

2）在开发REST API的视图时，虽然每个视图操作的数据可能不同，但增、删、改、查的基本流程是一样的，这部分代码DRF框架也进行了封装。

增：校验请求数据 -> 反序列化-将数据保存到对象中 -> 保存数据到数据库 -> 将保存的对象序列化返回
删：判断要删除的数据是否存在 -> 执行数据库删除 -> 返回响应
改：判断要修改的数据是否存在 -> 校验请求的数据 -> 反序列化-将数据保存到对象中 -> 保存数据库 -> 将保存的对象序列化返回
查(1个或多个)：查询数据库 -> 将数据序列化并返回
```

3. 特点

1.提供了定义序列化器Serializer的方法，可以快速根据 Django ORM 或者其它库自动序列化/反序列化
2.提供了丰富的类视图、Mixin扩展类、子类视图、视图集，简化视图代码的编写
3.多种身份认证和权限控制方式的支持
4.内置了限流系统
5.直观的 API web 界面
6.可扩展性，插件丰富

4. 总结

作用：快速开发RestAPI接口

特点：进行了大量封装，提高API开发速度

核心功能：序列化器和视图

环境安装与使用

1. 安装DRF框架

pip install djangorestframework

注：DRF框架依赖于Django，需要先安装Django环境，再安装djangorestframework

2. 添加rest_framework应用

```
在Django项目中使用DRF框架进行开发时，需要将rest_framework在INSTALLED_APPS中进行注册

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

DRF框架功能演示

注：此案例仅做DRF框架功能演示使用，现阶段不需要深究代码。

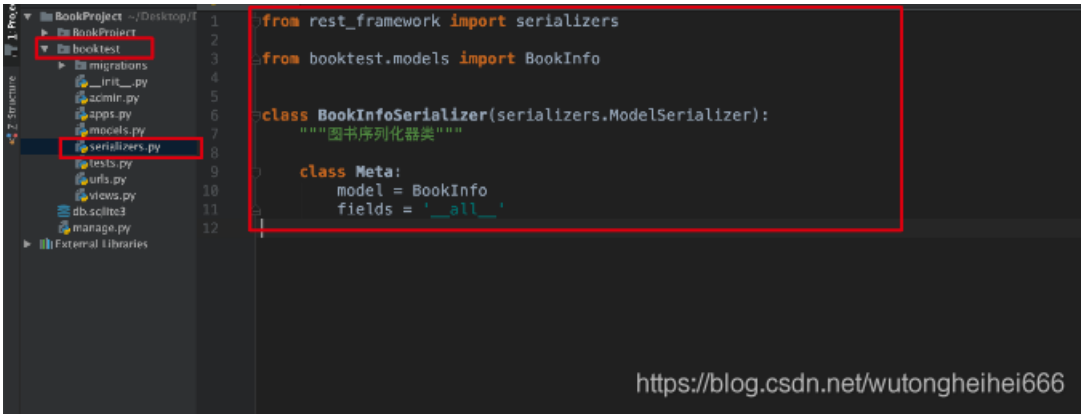
使用Django REST framework快速实现图书管理的5个RestAPI

1. 创建序列化器类

在booktest应用中新建serializers.py用于保存该应用的序列化器类

```
class BookInfoSerializer(serializers.ModelSerializer):
    """图书序列化器类"""
    class Meta:
        model = BookInfo
        fields = '__all__'
```

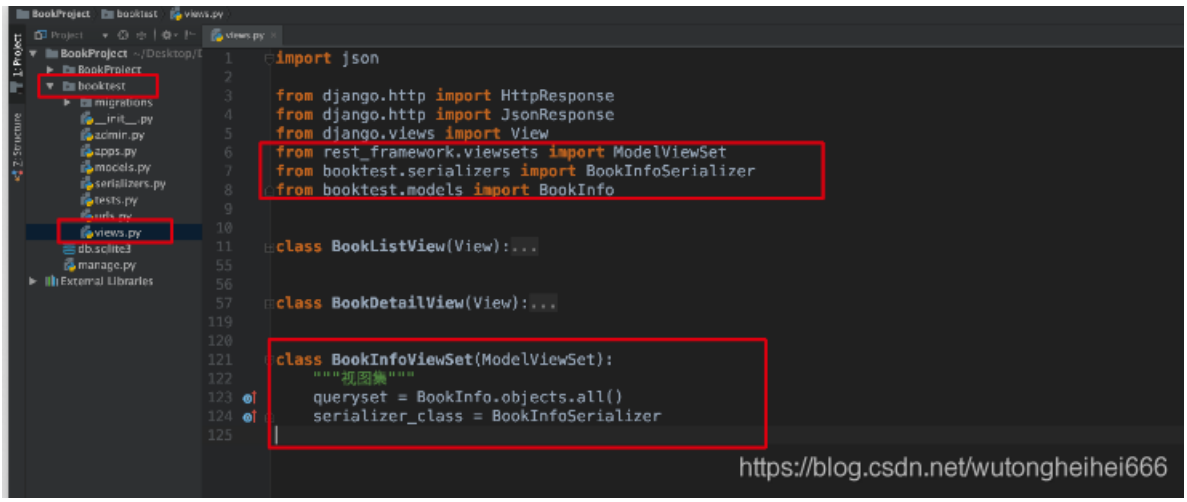
model：指定该序列化器类所对应的模型类
fields：指定依据模型类的哪些字段生成对应序列化器类的字段，**__all__**代表所有



2. 编写视图

```
在booktest应用的views.py中创建视图BookInfoViewSet，这是一个视图集合。


from rest_framework.viewsets import ModelViewSet
from booktest.serializers import BookInfoSerializer
from booktest.models import BookInfo

class BookInfoViewSet(ModelViewSet):
    """视图集"""
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
```

queryset：指定视图集在进行数据查询时所使用的查询集
serializer_class：指定视图集在进行序列化或反序列化时所使用的序列化器类



3. 定义路由

```
在booktest应用的urls.py中进行URL配置。

from booktest import views
from rest_framework.routers import DefaultRouter

urlpatterns = [
    ...
]

# 路由Router: 动态生成视图集中处理函数的url配置项
router = DefaultRouter()  # 路由Router
router.register('books', views.BookInfoViewSet)  # 向路由Router中注册视图集

urlpatterns += router.urls  # 将路由Router生成的url配置信息添加到django的路由列表中
```

4. 运行测试

```
运行当前程序（与运行Django一样）

python manage.py runserver
在浏览器中输入网址127.0.0.1:8000，可以看到DRF提供的API Web浏览页面：
```

5. 总结

视图集、序列化器类、路由Router

序列化器类

序列化器类定义

序列化器功能

序列化通俗直白的说：将实例对象----转换为---->字典（或者其他格式）
反序列化通俗直白的说：将字典（或者其他格式）----转换为------->实例对象

2.序列化器类定义
2.1 基本形式

```
想要完成Django使用序列化器，基本的方式如下

    定义模型类
    定义序列化器类
```

2.2 定义模型类

```
# 模型类定义
from django.db import models

class 模型类名(models.Model):
  # 模型类字段 = models.字段类型(选项参数)
  # ...
```

2.3 定义序列化器类

```
# 序列化器类定义
from rest_framework import serializers

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

2.4 demo

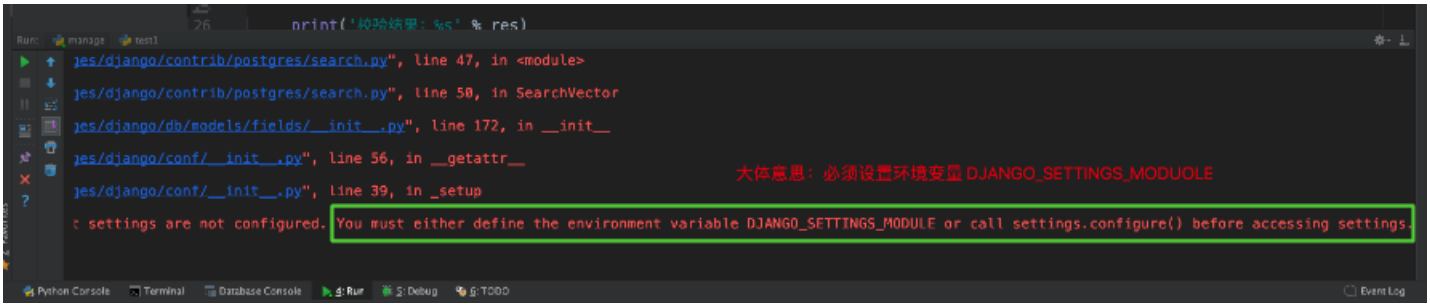
```
from rest_framework import serializers


class User(object):
    """用户类"""

    def __init__(self, name, age):
        self.name = name
        self.age = age


class UserSerializer(serializers.Serializer):
    """用户序列化器类"""
    name = serializers.CharField()
    age = serializers.IntegerField()
```

2.5 注意

运行一个单独的py文件，而该py文件中需要使用Django中的东西，则需要在py文件的开头添加以下代码：

```
# 设置Django运行所依赖的环境变量
import os
if not os.environ.get('DJANGO_SETTINGS_MODULE'):
  os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'drf_demo.settings')

# 让Django进行一次初始化
import django
django.setup()
```

如果不设定就会出现运行错误，如下



3. 序列化器基本使用

使用序列化器时需要先创建一个序列化器类的对象

```
# 使用自定义的序列化器类创建对象
序列化器对象 = 序列化器类(instance=None, data=empty, **kwarg)
```

说明：

用于序列化时，将实例对象传给**instance**参数
用于反序列化时，将要被反序列化的数据传给**data**参数

4. 操作演示

4.1 序列化操作

```
# 设置Django运行所依赖的环境变量
import os

if not os.environ.get('DJANGO_SETTINGS_MODULE'):
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'BookProject.settings')

from rest_framework import serializers


class User(object):
    """用户类"""

    def __init__(self, name, age):
        self.name = name
        self.age = age


class UserSerializer(serializers.Serializer):
    """用户序列化器类"""
    name = serializers.CharField()
    age = serializers.IntegerField()


# 创建User对象
user = User(name='smart', age=18)

# 使用UserSerializer将user对象序列化为如下字段数据：{'name': 'smart', 'age': 18}
serializer = UserSerializer(user)

# 获取序列化之后的数据
print(serializer.data)
```

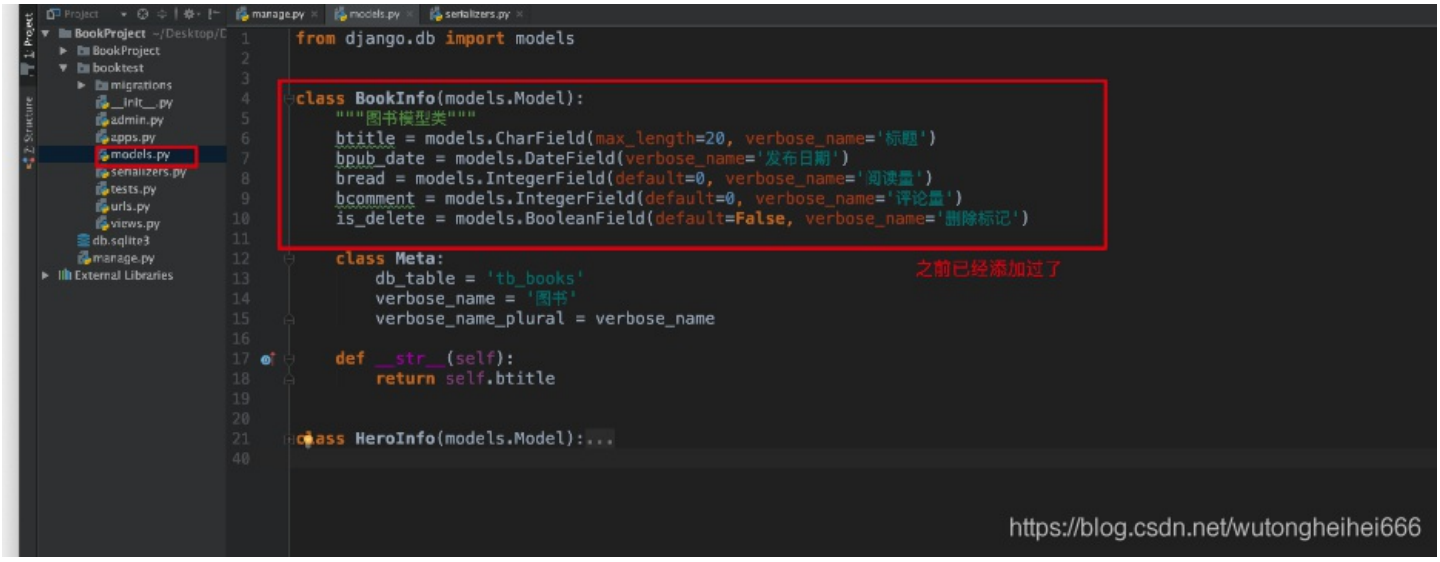
4.2 反序列化操作-数据校验

```
# 设置Django运行所依赖的环境变量
import os

if not os.environ.get('DJANGO_SETTINGS_MODULE'):
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'BookProject.settings')

from rest_framework import serializers


class User(object):
    """用户类"""

    def __init__(self, name, age):
        self.name = name
        self.age = age


class UserSerializer(serializers.Serializer):
    """用户序列化器类"""
    name = serializers.CharField()
    age = serializers.IntegerField()


# 准备数据
data = {'name': 'admin', 'age': 30}

# 使用UserSerializer对data中的数据进行反序列化校验
serializer = UserSerializer(data=data)

# 调用is_valid进行数据校验，成功返回True，失败返回False
print(serializer.is_valid())

# 获取校验失败之后的错误提示信息
print(serializer.errors)

# 获取校验通过之后的数据
print(serializer.validated_data)
```

序列化操作

1. 定义图书序列化器类

```
在BookProject/booktest/models.py中添加如下代码（注意之前如果已经存在则不用添加）

class BookInfo(models.Model):
    btitle = models.CharField(max_length=20, verbose_name='名称')
    bpub_date = models.DateField(verbose_name='发布日期', null=True)
    bread = models.IntegerField(default=0, verbose_name='阅读量')
    bcomment = models.IntegerField(default=0, verbose_name='评论量')
    is_delete = models.BooleanField(default=False, verbose_name='删除标记')
```



定义一个和BookInfo对应的序列化器类:

class BookInfoSerializer_v2(serializers.Serializer):
“”“图书数据序列化器”""
id = serializers.IntegerField(label=‘ID’, read_only=True)
btitle = serializers.CharField(label=‘名称’, max_length=20)
bpub_date = serializers.DateField(label=‘发布日期’)
bread = serializers.IntegerField(label=‘阅读量’, required=False)
bcomment = serializers.IntegerField(label=‘评论量’, required=False)

2.序列化单个对象

```
在终端中开启Django的shell，然后操作如下步骤

from booktest.models import BookInfo
from booktest.serializers import BookInfoSerializer_v2
# 查询获取图书对象
book = BookInfo.objects.get(id=2)
# 创建序列化器对象
serializer = BookInfoSerializer_v2(book)
# 获取序列化之后的数据
serializer.data
# {'id': 2, 'btitle': '天龙八部', 'bpub_date': '1986-07-24', 'bread': 36, 'bcomment': 40}
```

3.序列化多个对象

如果要被序列化的是包含多个对象的查询集QuerySet或list，在创建序列化器对象时，需要添加many=True参数。

```
from booktest.models import BookInfo
from booktest.serializers import BookInfoSerializer_v2
# 查询多个数据
books = BookInfo.objects.all()
# 创建序列化器对象
serializer = BookInfoSerializer_v2(books, many=True)
# 获取序列化之后的数据
serializer.data
# [
#   OrderedDict([('id', 2), ('btitle', '天龙八部'), ('bpub_date', '1986-07-24'), ('bread', 36), ('bcomment', 40)
),
#   OrderedDict([('id', 3), ('btitle', '笑傲江湖'), ('bpub_date', '1995-12-24'), ('bread', 20), ('bcomment', 80)
),
#   OrderedDict([('id', 4), ('btitle', '雪山飞狐'), ('bpub_date', '1987-11-11'), ('bread', 58), ('bcomment', 24)
),
#   OrderedDict([('id', 5), ('btitle', '西游记'), ('bpub_date', '1988-01-01'), ('bread', 10), ('bcomment', 10)]
)
# ]
```

OrderedDict是有序字典类型

4. 关联对象嵌套序列化

4.1 说明

如果在序列化对象数据时，需要将其关联的对象一并序列化
则定义序列化器类的字段时，需要再定义对应的关联对象嵌套序列化字段。

4.2 demo

在序列化英雄对象数据时，hbook(即和英雄关联的图书)字段如何序列化？

我们先定义HeroInfoSerialzier除关联对象嵌套序列化字段之外的其他部分

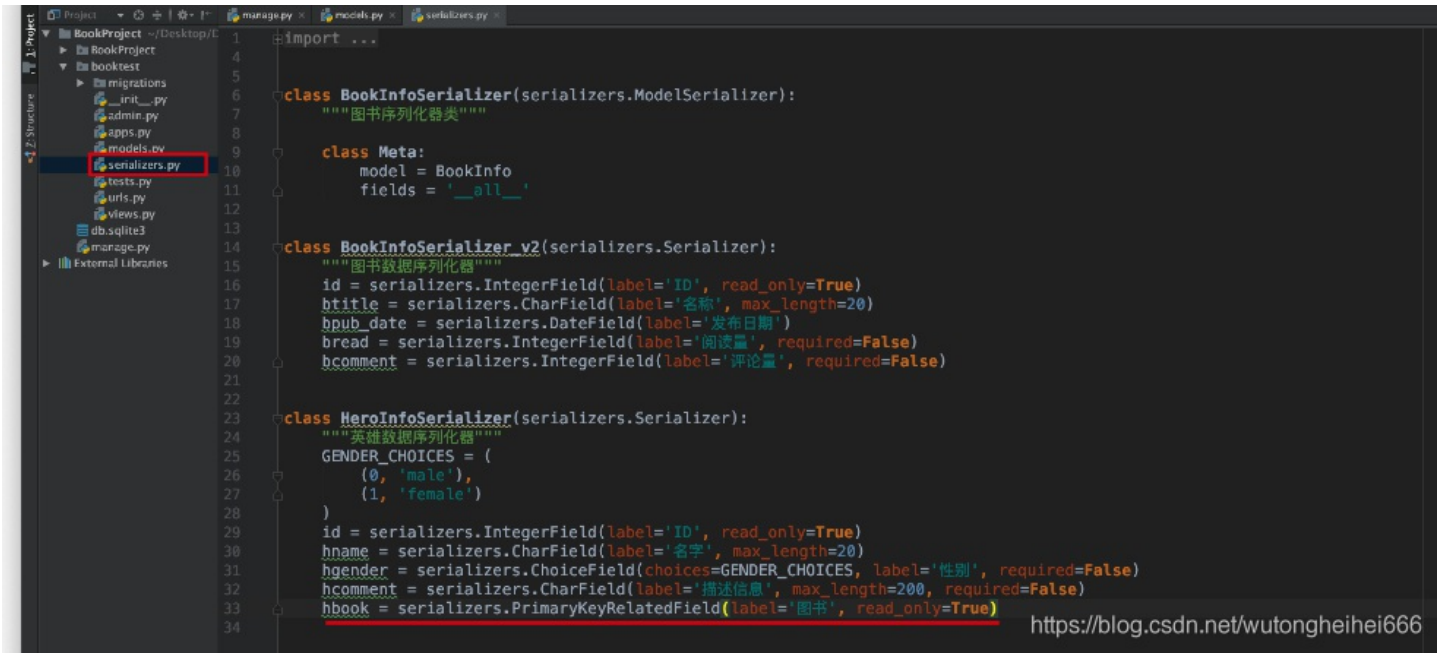
```
class HeroInfoSerializer(serializers.Serializer):
    """英雄数据序列化器"""
    GENDER_CHOICES = (
        (0, 'male'),
        (1, 'female')
    )
    id = serializers.IntegerField(label='ID', read_only=True)
    hname = serializers.CharField(label='名字', max_length=20)
    hgender = serializers.ChoiceField(choices=GENDER_CHOICES, label='性别', required=False)
    hcomment = serializers.CharField(label='描述信息', max_length=200, required=False
```

对于嵌套关联字段，可以采用以下3种方式进行定义：

4.2.1 PrimaryKeyRelatedField

将关联对象序列化为关联对象的主键

```
# 在HeroInfoSerializer中添加此字段
hbook = serializers.PrimaryKeyRelatedField(label='图书', read_only=True)
或
hbook = serializers.PrimaryKeyRelatedField(label='图书', queryset=BookInfo.objects.all())

指明字段时需要包含read_only=True或者queryset参数：

指定read_only=True参数时，该字段仅在序列化时使用。
指定queryset参数时，将被用作反序列化时参数校验使用。
```



注：使用**PrimaryKeyRelatedField**将和英雄关联的图书**hbook**序列化为了**hbook**的主键。

4.2.2 使用关联对象的序列化器

使用指定的序列化器类将关联对象进行序列化

```
hbook = BookInfoSerializer_v2()
```

4.2.3 StringRelatedField

```
将关联对象序列化为关联对象模型类__str__方法的返回值

hbook = serializers.StringRelatedField(label='图书')
```

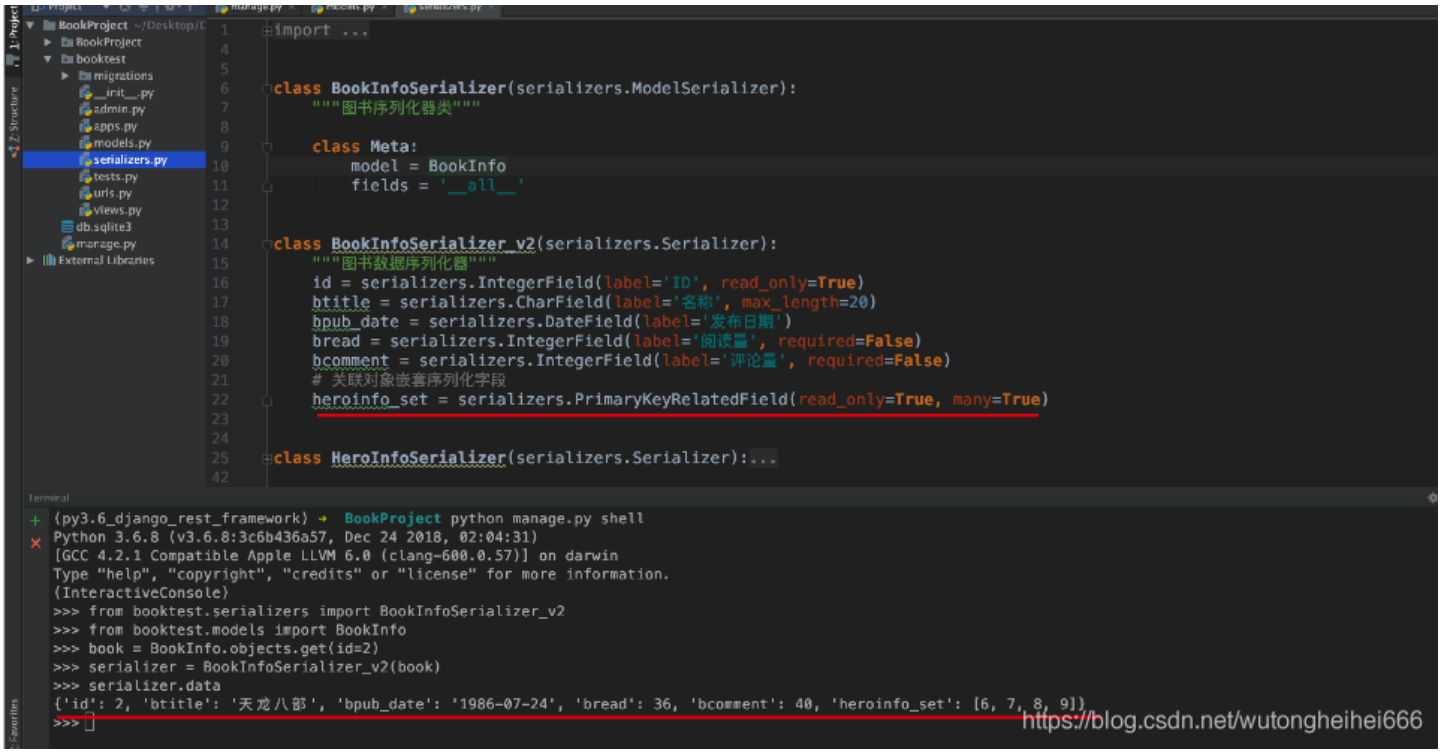
注：将和英雄关联的图书**hbook**序列化为图书模型类**__str__**方法的返回值

5. many参数

如果和一个对象关联的对象有多个，在序列化器类中定义嵌套序列化字段时，需要多添加一个many=True参数

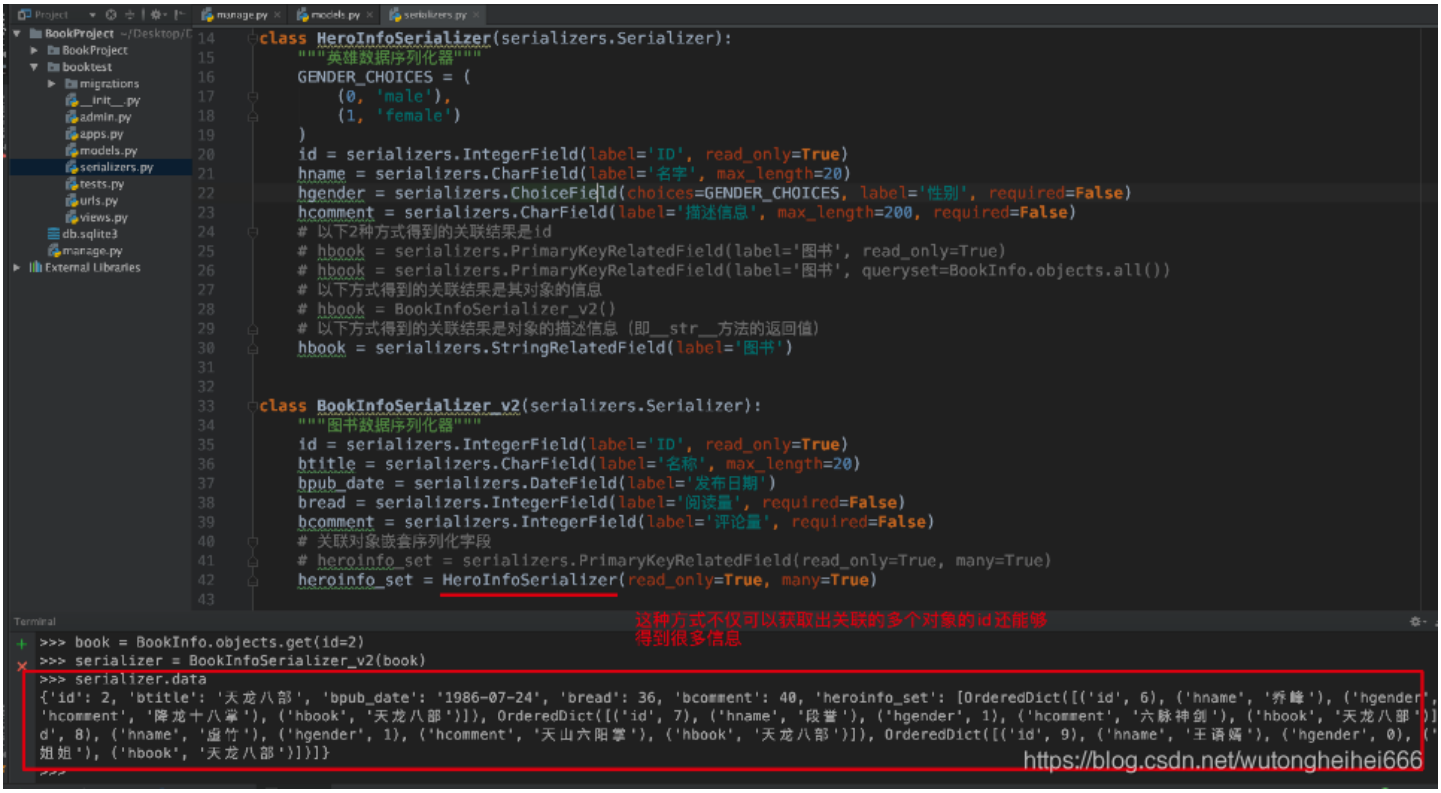
如在序列化图书对象时，将和图书关联的英雄一并进行嵌套序列化，需在BookInfoSerializer_v2中添加嵌套序列化字段：

```
class BookInfoSerializer_v2(serializers.Serializer):
    """图书数据序列化器"""
    id = serializers.IntegerField(label='ID', read_only=True)
    btitle = serializers.CharField(label='名称', max_length=20)
    bpub_date = serializers.DateField(label='发布日期')
    bread = serializers.IntegerField(label='阅读量', required=False)
    bcomment = serializers.IntegerField(label='评论量', required=False)
    # 关联对象嵌套序列化字段
    heroinfo_set = serializers.PrimaryKeyRelatedField(read_only=True, many=True)
```



最后的serializers.py文件可以修改为如下，可以测试通过一对多，找到多个数据

```python
from rest_framework import serializers

from booktest.models import BookInfo


class BookInfoSerializer(serializers.ModelSerializer):
    """图书序列化器类"""

    class Meta:
        model = BookInfo
        fields = '__all__'


class HeroInfoSerializer(serializers.Serializer):
    """英雄数据序列化器"""
    GENDER_CHOICES = (
        (0, 'male'),
        (1, 'female')
    )
    id = serializers.IntegerField(label='ID', read_only=True)
    hname = serializers.CharField(label='名字', max_length=20)
    hgender = serializers.ChoiceField(choices=GENDER_CHOICES, label='性别', required=False)
    hcomment = serializers.CharField(label='描述信息', max_length=200, required=False)
    # 以下2种方式得到的关联结果是id
    # hbook = serializers.PrimaryKeyRelatedField(label='图书', read_only=True)
    # hbook = serializers.PrimaryKeyRelatedField(label='图书', queryset=BookInfo.objects.all())
    # 以下方式得到的关联结果是其对象的信息
    # hbook = BookInfoSerializer_v2()
    # 以下方式得到的关联结果是对象的描述信息（即__str__方法的返回值）
    hbook = serializers.StringRelatedField(label='图书')


class BookInfoSerializer_v2(serializers.Serializer):
    """图书数据序列化器"""
    id = serializers.IntegerField(label='ID', read_only=True)
    btitle = serializers.CharField(label='名称', max_length=20)
    bpub_date = serializers.DateField(label='发布日期')
    bread = serializers.IntegerField(label='阅读量', required=False)
    bcomment = serializers.IntegerField(label='评论量', required=False)
    # 关联对象嵌套序列化字段
    # heroinfo_set = serializers.PrimaryKeyRelatedField(read_only=True, many=True)
    heroinfo_set = HeroInfoSerializer(read_only=True, many=True)
```



反序列化操作

1. 数据校验

1.1 基本使用方式

```
# 1. 创建序列化器对象
serializer = 序列化器类(data=<待校验字典数据>)

# 2. 数据校验：成功返回True，失败返回False
serializer.is_valid()

# 3. 获取校验成功之后的数据
serializer.validated_data

# 4. 如果校验失败，获取失败的错误提示信息
serializer.errors



# 1. 创建序列化器对象
serializer = 序列化器类(data=<待校验字典数据>)

# 2. 数据校验：成功返回True，失败返回False
serializer.is_valid()

# 3. 获取校验成功之后的数据
serializer.validated_data

# 4. 如果校验失败，获取失败的错误提示信息
serializer.errors
```

注：

调用**is_valid**时，会根据对应序列化类字段是否需要传递、字段类型以及一些选项参数对**data**中的数据进行校验。

1.2 操作**demo**

```
from booktest.serializers import BookInfoSerializer_v2

data = {'btitle': 'python', 'bpub_date': '2019-06-01'}
serializer = BookInfoSerializer_v2(data=data)
serializer.is_valid()  # 返回True
serializer.errors  # {}
serializer.validated_data
```

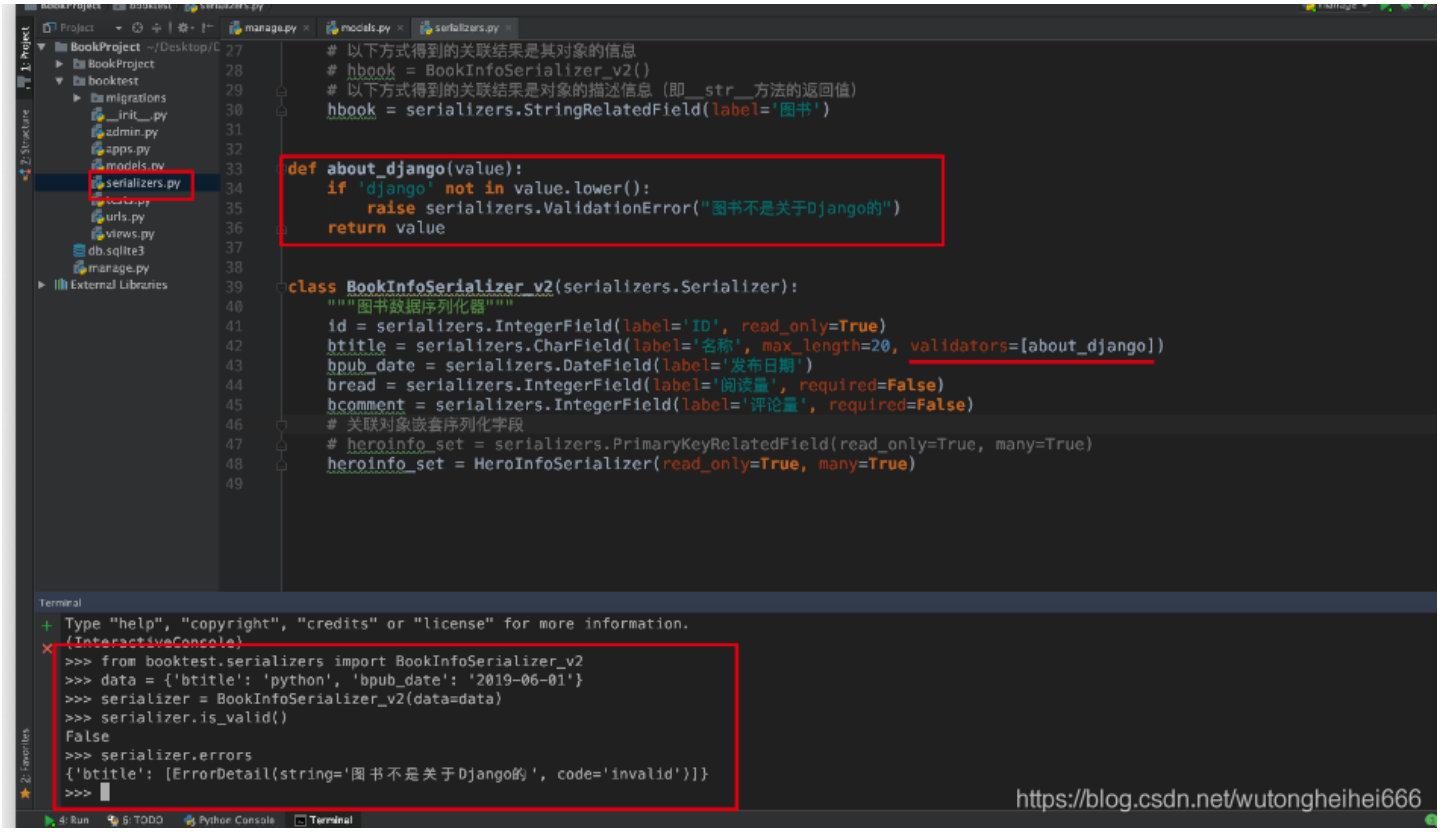
2. 补充验证

```
在调用is_valid进行数据校验时，除了一些基本的默认验证行为，可能还需要补充一些验证行为，比如有如下需求：

1）在进行btitle验证时，要求btitle的内容必须含有django

2）在进行bread和bcomment验证时，要求bread必须大于等于bcomment
想要实现上述功能，需要再补充验证行为

可以使用以下三种方法：
```

2.1 validators

针对指定序列化器字段添加**validators**选项参数补充校验

```
def about_django(value):
    if 'django' not in value.lower():
        raise serializers.ValidationError("图书不是关于Django的")
    return value

class BookInfoSerializer_v2(serializers.Serializer):
    """图书数据序列化器"""
    id = serializers.IntegerField(label='ID', read_only=True)
    btitle = serializers.CharField(label='名称', max_length=20, validators=[about_django])
    bpub_date = serializers.DateField(label='发布日期')
    bread = serializers.IntegerField(label='阅读量', required=False)
    bcomment = serializers.IntegerField(label='评论量', required=False)
```



2.2 validate_<field_name>

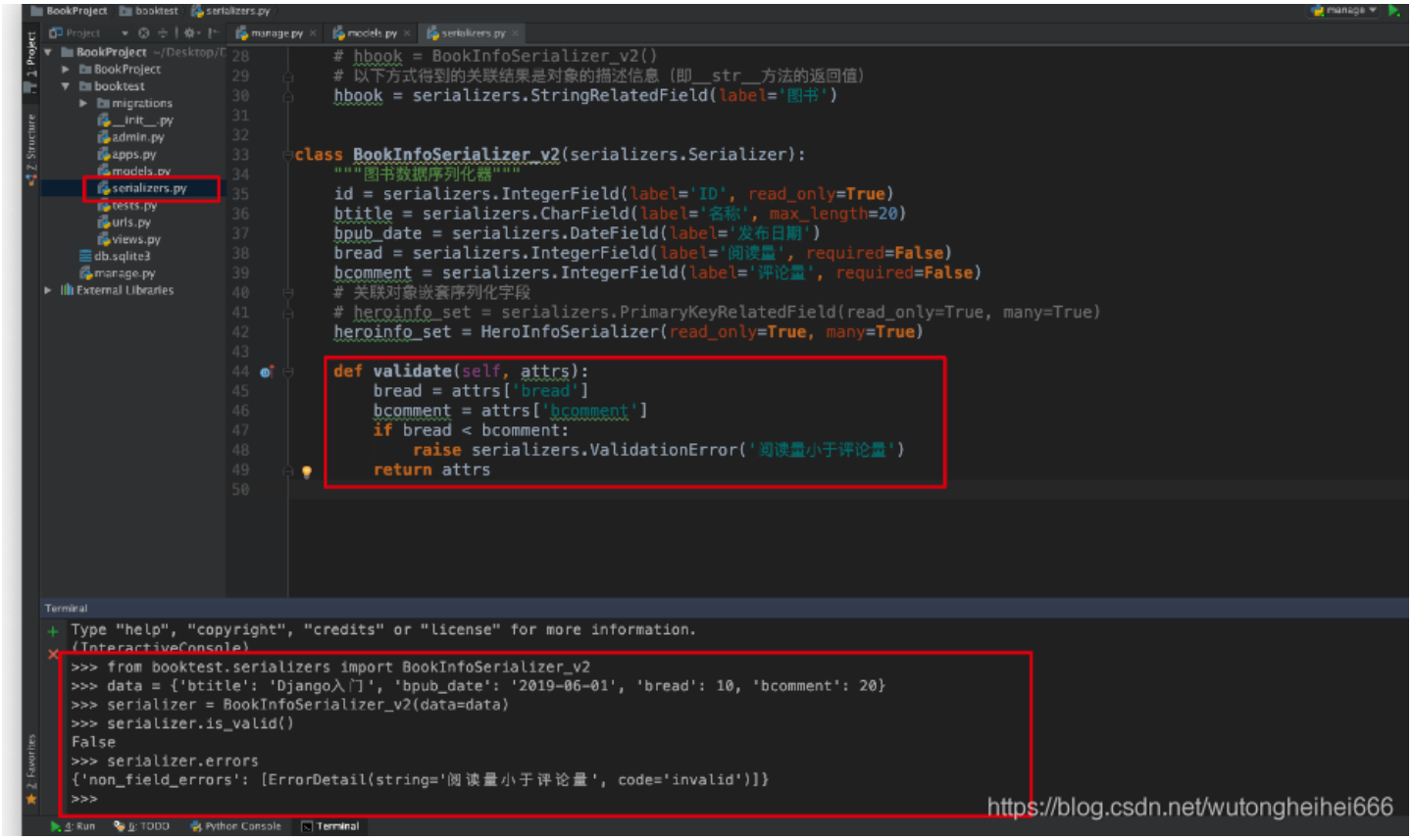
在序列化器类中定义特定方法validate_<field_name>，针对特定字段进行补充验证，如：

```
class BookInfoSerializer_v2(serializers.Serializer):
    """图书数据序列化器"""
    ...

    def validate_btitle(self, value):
        if 'django' not in value.lower():
            raise serializers.ValidationError("图书不是关于Django的")
        return value
```

2.3 validate

在序列化器类中定义validate方法针对多个字段的内容进行补充验证，如：

```
class BookInfoSerializer_v2(serializers.Serializer):
    """图书数据序列化器"""
    ...

    def validate(self, attrs):
        bread = attrs['bread']
        bcomment = attrs['bcomment']
        if bread < bcomment:
            raise serializers.ValidationError('阅读量小于评论量')
        return attrs
```


BookProject
booktest
serializers.py
manage.py
models.py
# hbook = BookInfoSerializer_v2()
# 以下方式得到的关联结果是对象的描述信息（即__str__方法的返回值）
hbook = serializers.StringRelatedField(label='图书')
class BookInfoSerializer_v2(serializers.Serializer):
"""图书数据序列化器"""
id = serializers.IntegerField(label='ID', read_only=True)
btitle = serializers.CharField(label='名称', max_length=20)
bpub_date = serializers.DateField(label='发布日期')
bread = serializers.IntegerField(label='阅读量', required=False)
bcomment = serializers.IntegerField(label='评论量', required=False)
# 关联对象嵌套序列化字段
# heroinfo_set = serializers.PrimaryKeyRelatedField(read_only=True, many=True)
heroinfo_set = HeroInfoSerializer(read_only=True, many=True)
def validate(self, attrs):
bread = attrs['bread']
bcomment = attrs['bcomment']
if bread < bcomment:
raise serializers.ValidationError('阅读量小于评论量')
return attrs
Terminal
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from booktest.serializers import BookInfoSerializer_v2
>>> data = {'btitle': 'Django入门', 'bpub_date': '2019-06-01', 'bread': 10, 'bcomment': 20}
>>> serializer = BookInfoSerializer_v2(data=data)
>>> serializer.is_valid()
False
>>> serializer.errors
{'non_field_errors': [ErrorDetail(string='阅读量小于评论量', code='invalid')]}
>>>
Run
TODO
Python Console
Terminal
https://blog.csdn.net/wutongheihei666

3. 数据保存

1）在数据校验通过之后，想要基于validated_data完成数据的保存，可以通过序列化器对象.save()进行数据的保存。

2）在save方法内部会调用序列化器类的create或update方法，可以在create方法中实现数据新增，update方法中实现数据更新。

3）创建序列化器对象的时候

如果没有传递instance实例，则调用save()方法的时候，create()被调用

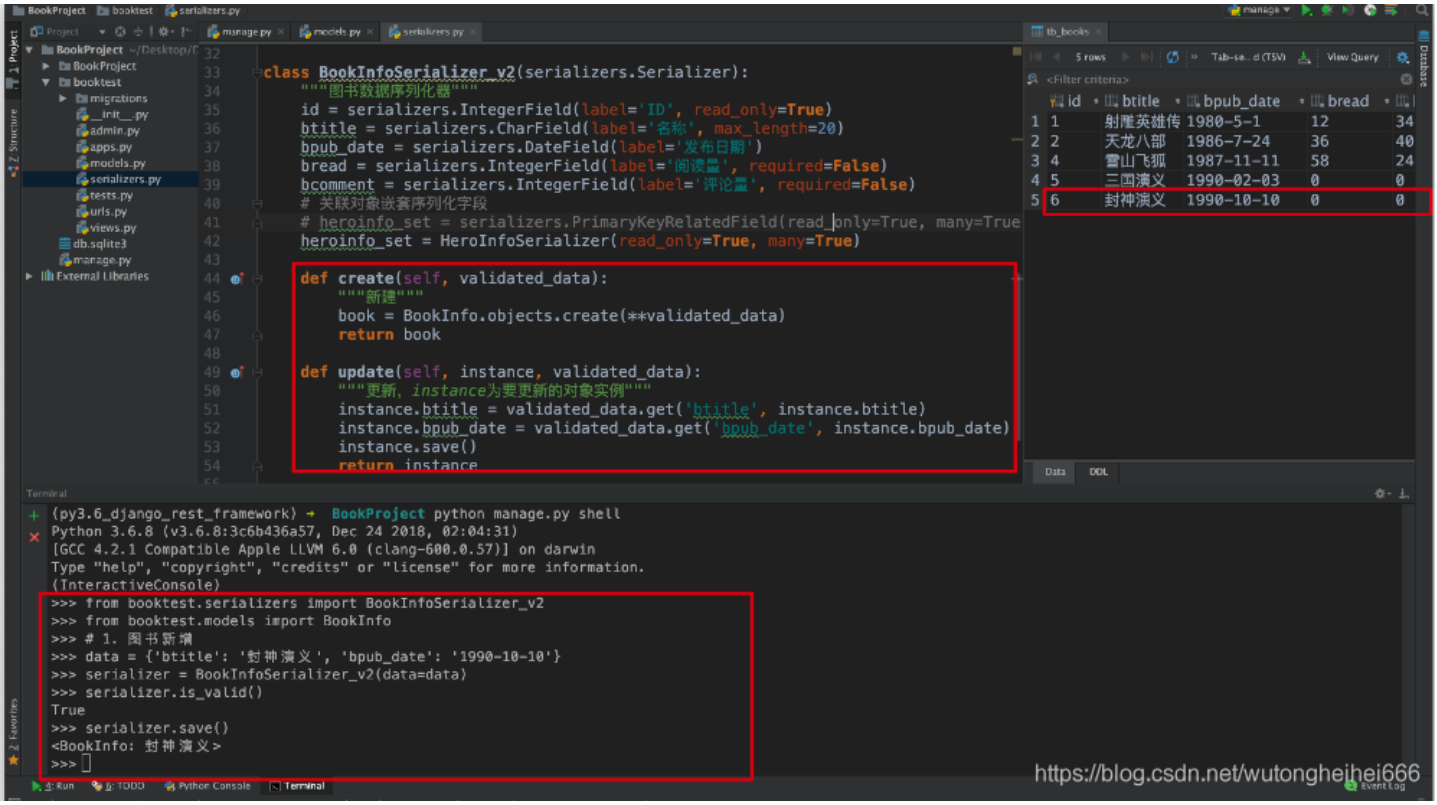
相反，如果传递了instance实例，则调用save()方法的时候，update()被调用

如：

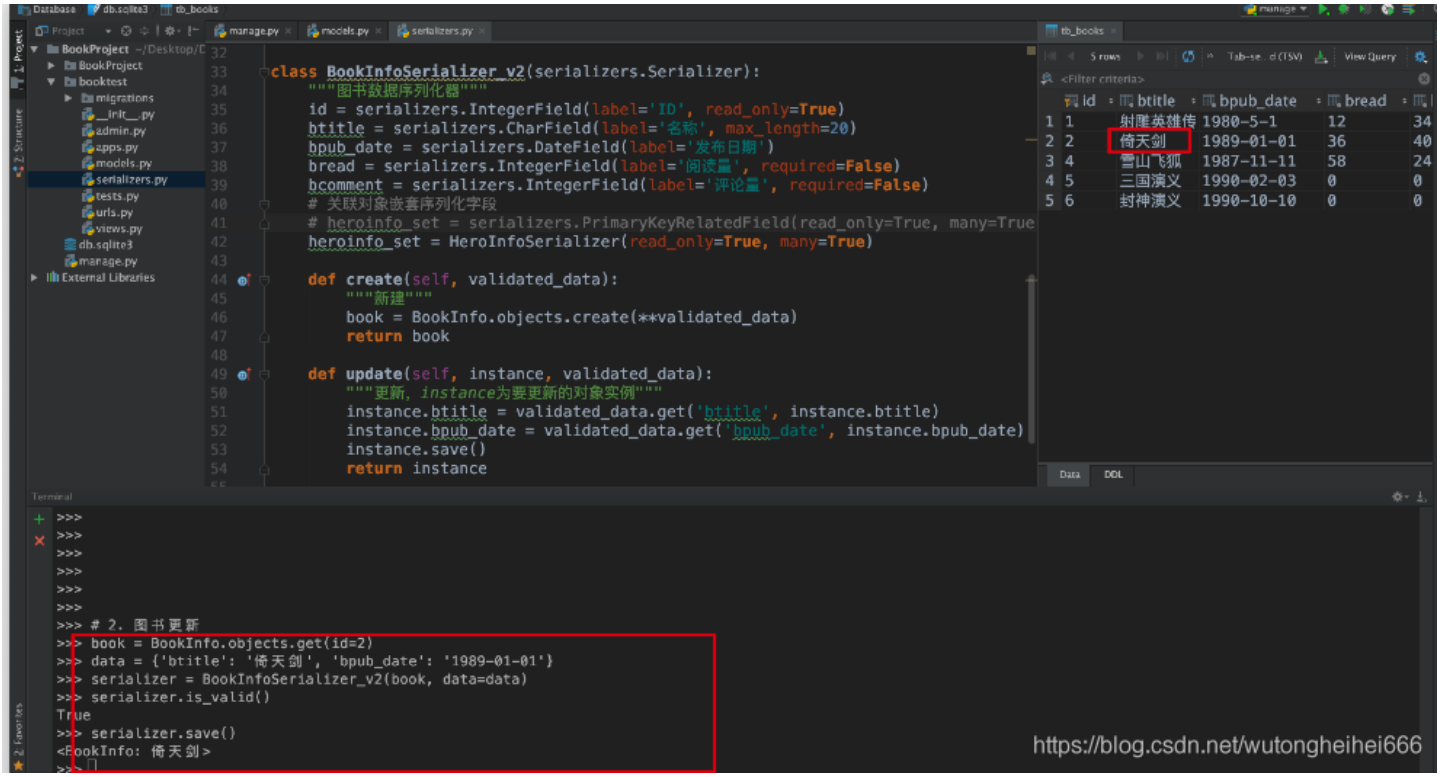
```
class BookInfoSerializer_v2(serializers.Serializer):
    """图书数据序列化器"""
    ...

    def create(self, validated_data):
        """新建"""
        book = BookInfo.objects.create(**validated_data)
        return book

    def update(self, instance, validated_data):
        """更新，instance为要更新的对象实例"""
        instance.btitle = validated_data.get('btitle', instance.btitle)
        instance.bpub_date = validated_data.get('bpub_date', instance.bpub_date)
        instance.save()
        return instance
```

测试：

```
from booktest.serializers import BookInfoSerializer_v2
from booktest.models import BookInfo

# 1. 图书新增
data = {'btitle': '封神演义', 'bpub_date': '1990-10-10'}
serializer = BookInfoSerializer_v2(data=data)
serializer.is_valid()  # True
serializer.save()  # 调用序列化器类的create方法，实现图书的新增
serializer.data # 获取新增图书序列化之后的数据
```




```
from booktest.serializers import
BookInfoSerializer_v2
from booktest.models import BookInfo

# 2. 图书更新
book = BookInfo.objects.get(id=2)
data = {'btitle': '倚天剑', 'bpub_date': '1989-01-01'}
serializer = BookInfoSerializer_v2(book, data=data)
serializer.is_valid()  # True
serializer.save()  # 调用序列化器类的update方法，实现图书的更新
serializer.data # 获取更新图书序列化之后的数据
```



4. 总结

反序列化-基本校验

```
创建序列化器对象，将字典数据传递给data，调用
序列化器的is_valid方法进行数据校验
```

反序列化-补充校验

```
1.针对指定字段添加validators选项参数添加补充验证函数
2.在序列化器类中定义特定方法 validate_<field_name>针对特定字段进行补充验证
3.在序列化器类中定义方法validate进行补充验证
```

反序列化-数据保存

```
1.数据校验通过之后，可以调用序列化对象的save方法进行数据保存
2.save方法内部会调用对应序列化器类中的create或update方法，可以在create中实现数据新增，在update中实现数据更新
```

ModelSerializer

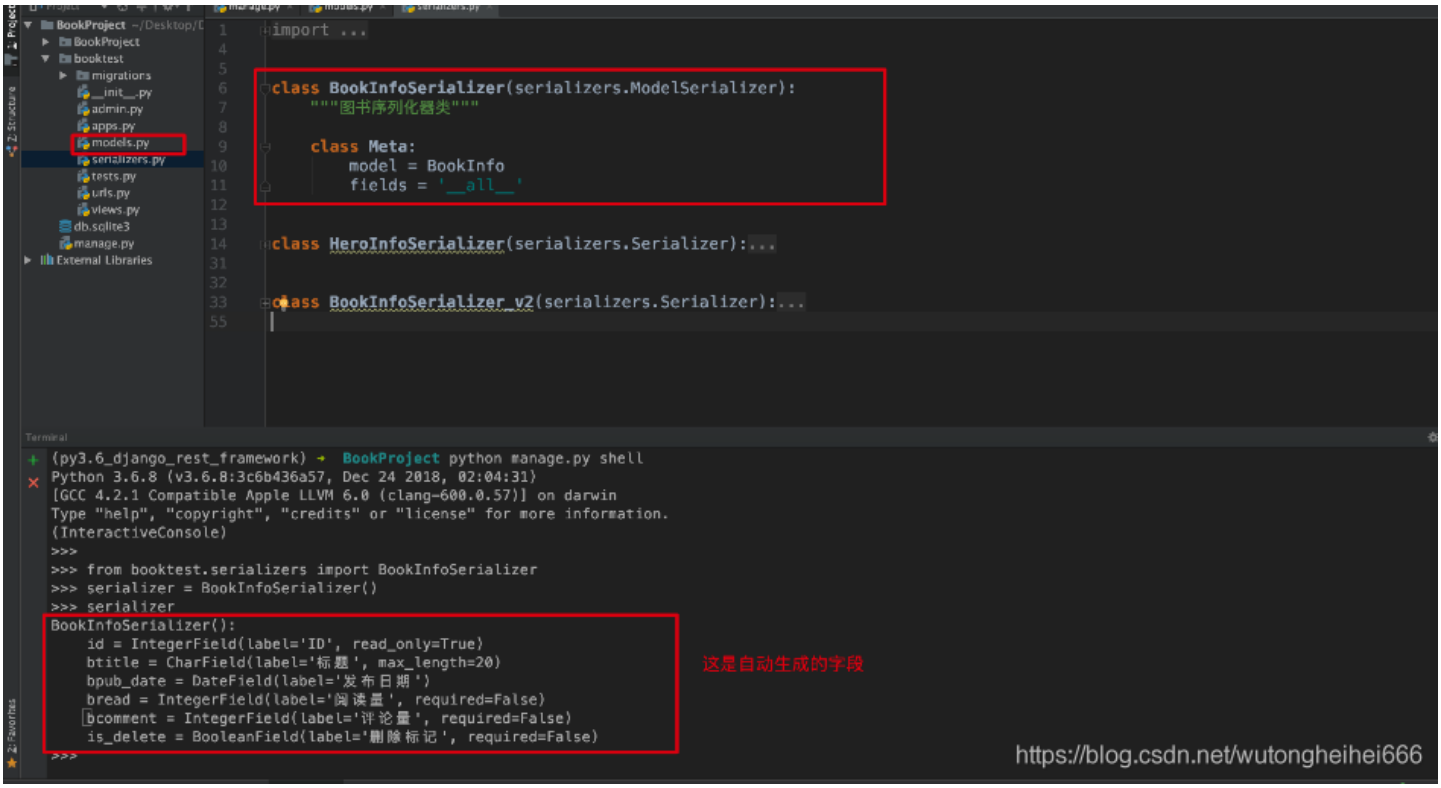
```
如果序列化器类对应的是Django的某个模型类，则定义序列化器类时，可以直接继承于ModelSerializer

ModelSerializer是Serializer类的子类，相对于Serializer，提供了以下功能：

基于模型类字段自动生成序列化器类的字段
包含默认的create()和update()方法的实现
```

1. 基本使用

创建一个BookInfoSerializer类：

```
class BookInfoSerializer(serializers.ModelSerializer):
    """图书数据序列化器"""
    class Meta:
        model = BookInfo
        fields = '__all__'
```

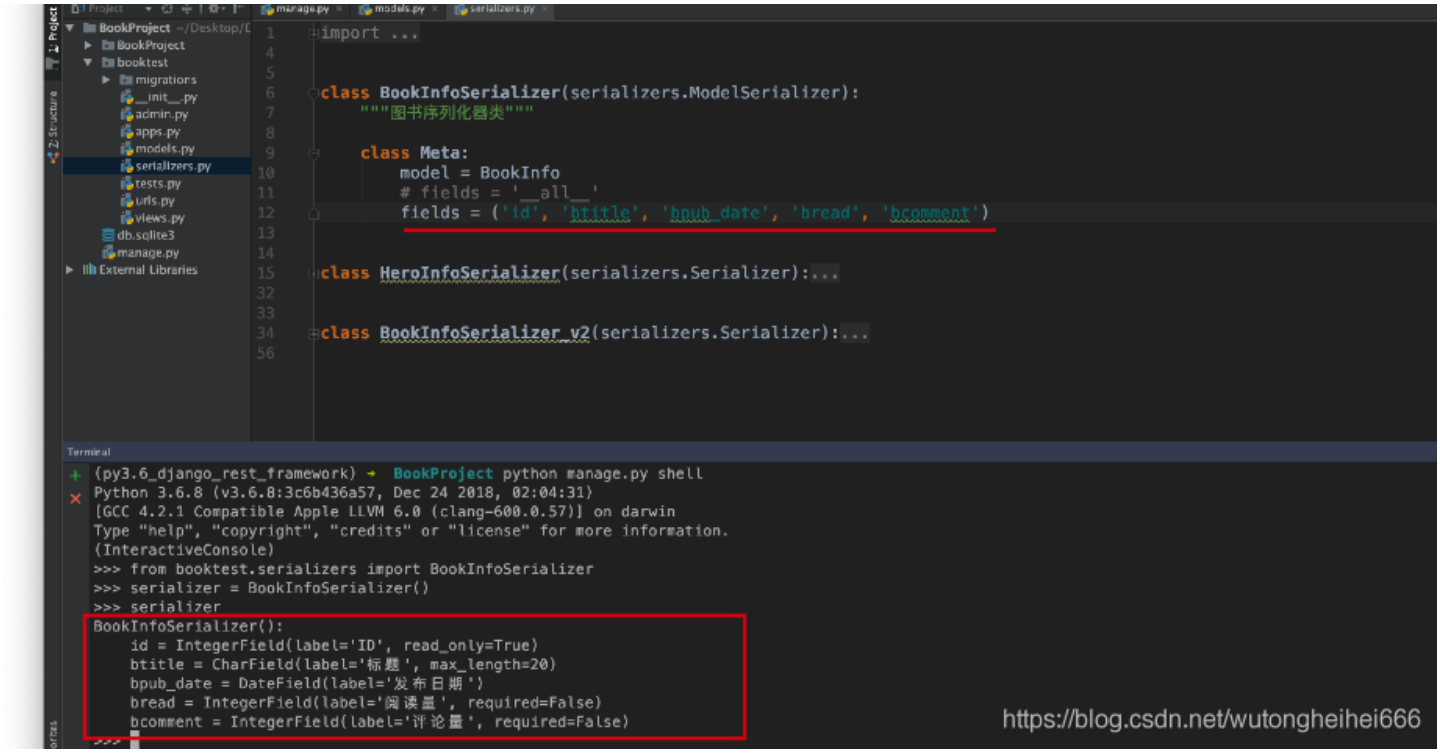
model：指明序列化器类对应的模型类
fields：指明依据模型类的哪些字段生成序列化器类的字段



2. 指定字段

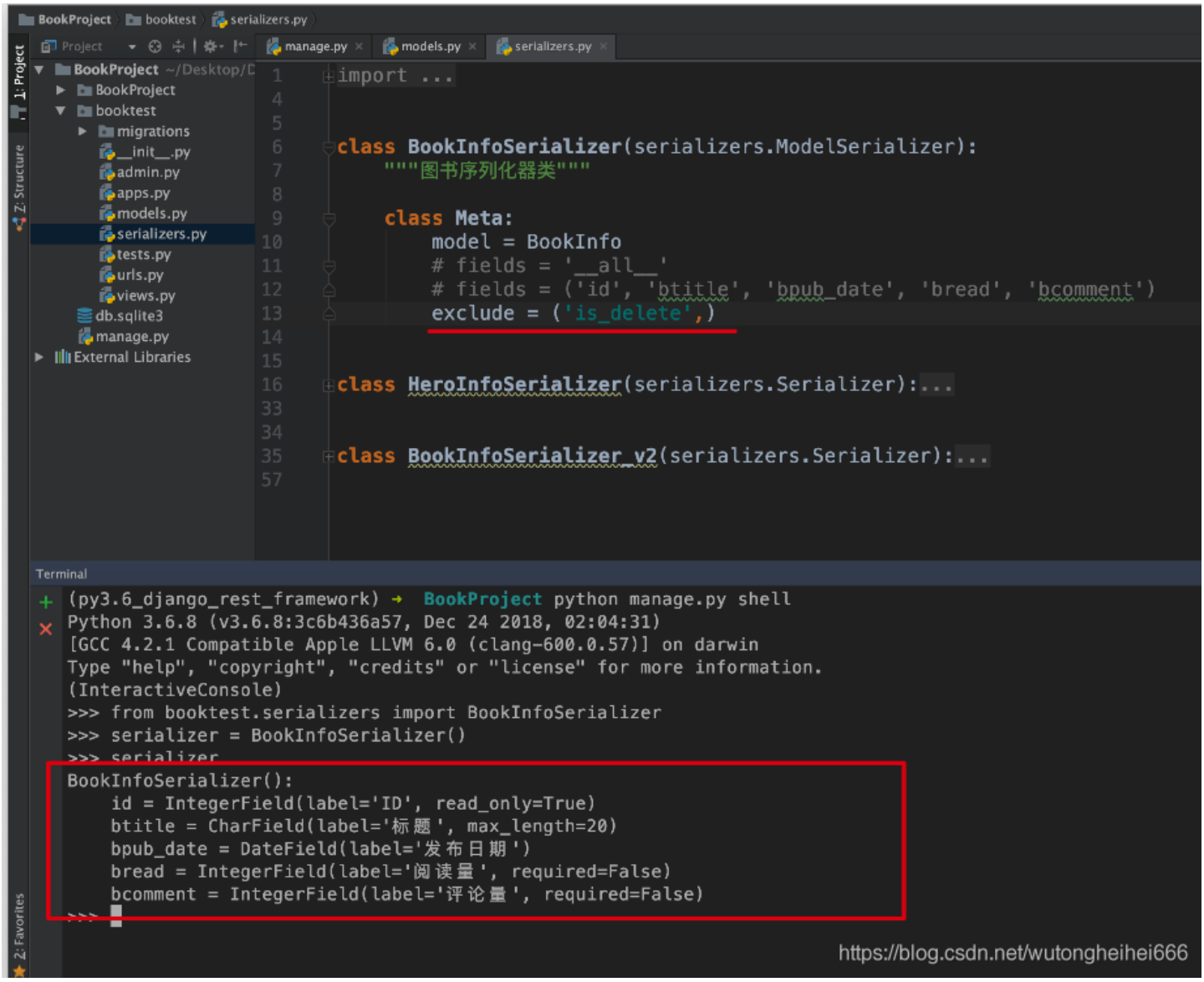
使用fields来指明依据模型类的哪些字段生成序列化器类的字段，__all__表明包含所有字段，也可以指明具体哪些字段，如：

```
class BookInfoSerializer(serializers.ModelSerializer):
    """图书数据序列化器"""
    class Meta:
        model = BookInfo
        fields = ('id', 'btitle', 'bpub_date', 'bread', 'bcomment')
```



使用exclude可以指明排除哪些字段，如：

```
class BookInfoSerializer(serializers.ModelSerializer):
    """图书数据序列化器"""
    class Meta:
        model = BookInfo
        exclude = ('is_delete',)
```



3.添加和修改选项参数

```
可以使用extra_kwargs参数为自动生成的序列化器类字段添加或修改原有的选项参数

class BookInfoSerializer(serializers.ModelSerializer):
"""图书数据序列化器"""
    class Meta:
        model = BookInfo
        fields = ('id', 'btitle', 'bpub_date', 'bread', 'bcomment')
        extra_kwargs = {
            'bread': {'min_value': 0, 'required': True},
            'bcomment': {'min_value': 0, 'required': True},
            'bpub_date': {'required': True}
        }
```

4.总结

```
ModelSerializer使用

如果序列化器类对应的是Django的模型类，可以直接继承ModelSerializer
继承ModelSerializer之后，可以依据模型类字段自动生成序列化器的字段
ModelSerializer中已经实现了create和update方法
```

序列化时字段类型和选项参数

1. 常用字段类型

字段	字段构造方式
BooleanField	BooleanField()
NullBooleanField	NullBooleanField()
CharField	CharField(max_length=None, min_length=None, allow_blank=False, trim_whitespace=True)
EmailField	EmailField(max_length=None, min_length=None, allow_blank=False)
RegexField	RegexField(regex, max_length=None, min_length=None, allow_blank=False)
SlugField	SlugField(max_length=50, min_length=None, allow_blank=False) 正则字段，验证正则模式 [-a-zA-Z0-9_-]+
URLField	URLField(max_length=200, min_length=None, allow_blank=False)
UUIDField	UUIDField(format='hex_verbose') format: 1) `'hex_verbose'` 如 `"5ce0e9a5-5ffa-654b-cee0-1238041fb31a"` 2) `'hex'` 如 `"5ce0e9a55ffa654bcee01238041fb31a"` 3) `'int'` - 如: `"123456789012312313134124512351145145114"` 4) `'urn'` 如: `"urn:uuid:5ce0e9a5-5ffa-654b-cee0-1238041fb31a"`

IPAddressField	IPAddressField(protocol='both', unpack_ipv4=False, **options)
IntegerField	IntegerField(max_value=None, min_value=None)
FloatField	FloatField(max_value=None, min_value=None)
DecimalField	DecimalField(max_digits, decimal_places, coerce_to_string=None, max_value=None, min_value=None) max_digits: 最多位数 decimal_palces: 小数点位置
DateTimeField	DateTimeField(format=api_settings.DATETIME_FORMAT, input_formats=None)
DateField	DateField(format=api_settings.DATE_FORMAT, input_formats=None)
TimeField	TimeField(format=api_settings.TIME_FORMAT, input_formats=None)
DurationField	DurationField()
ChoiceField	ChoiceField(choices) choices与Django的用法相同
MultipleChoiceField	MultipleChoiceField(choices)
FileField	FileField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)
ImageField	ImageField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)

2. 通用参数

无论哪种字段类型都可以使用的选项参数。

参数名称	说明
read_only	表明该字段仅用于序列化输出，默认False
write_only	表明该字段仅用于反序列化输入，默认False
required	表明该字段在反序列化时必须输入，默认True
default	序列化和反序列化时使用的默认值
error_messages	包含错误编号与错误信息的字典
label	用于HTML展示API页面时，显示的字段名称

定义序列化器类的字段时，如果没有指定`read_only`和`write_only`，则这两个参数默认值都为`False`，表明对应的字段既在序列化时使用，也在反序列化时使用

3. demo

```
#read_only
# 设置Django运行所依赖的环境变量
import os

if not os.environ.get('DJANGO_SETTINGS_MODULE'):
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'BookProject.settings')

from rest_framework import serializers


class User(object):
    """用户类"""

    def __init__(self, name, age):
        self.name = name
        self.age = age


class UserSerializer(serializers.Serializer):
    """序列化器类"""
    name = serializers.CharField()
    age = serializers.IntegerField(read_only=True)


if __name__ == "__main__":
    # 准备数据
    data = {'name': 'laowang', 'age': 18}

    # 数据校验
    serializer = UserSerializer(data=data)
    res = serializer.is_valid()
    print('校验结果：%s' % res)
    # 如果校验失败，获取校验失败的错误原因
    print(serializer.errors)
    # 如果校验成功，获取校验之后的数据
    print(serializer.validated_data)
```

常用参数：

参数名称	作用
max_length	最大长度
min_length	最小长度
max_value	最大值
min_value	最小值

> max_length和min_length是针对字符串类型的参数
> max_value和min_value是针对数字类型的参数。

视图部分

APIView

1. APIView介绍

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

APIView与**View**的区别：

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

2. Request对象

视图继承APIView之后，传入视图的request对象是DRF框架提供的Request类的对象，Request类的对象有两个属性：

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

3. Response对象

视图继承APIView之后，响应时可以统一返回Response对象，格式如下：

```
from rest_framework.response import Response

response = Response(<原始响应数据>)
```

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

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