django rest framework学习

原创

Slience_me 于 2021-01-10 22:34:17 发布 157 收藏 3

文章标签： django 数据库 python restful

本文链接：https://blog.csdn.net/Slience_me/article/details/112444684

文章目录

1）APIView

2）GenericAPIView

2. 五个扩展类

1）ListModelMixin

2）CreateModelMixin

3） RetrieveModelMixin

4）UpdateModelMixin

5）DestroyModelMixin

3. 几个可用子类视图

1） CreateAPIView

2）ListAPIView

3）RetrieveAPIView

4）DestoryAPIView

5）UpdateAPIView

6）RetrieveUpdateAPIView

7）RetrieveUpdateDestoryAPIView

4.4 视图集ViewSet

1. 常用视图集父类

1） ViewSet

2）GenericViewSet

3）ModelViewSet

4）ReadOnlyModelViewSet

2. 视图集中定义附加action动作

3. action属性

4. 视图集的继承关系

4.5 路由Routers

1. 使用方法

2. 视图集中附加action的声明

3. 路由router形成URL的方式

1） SimpleRouter

2）DefaultRouter

5. 其他功能

5.1 认证

认证Authentication

5.2 权限

权限Permissions

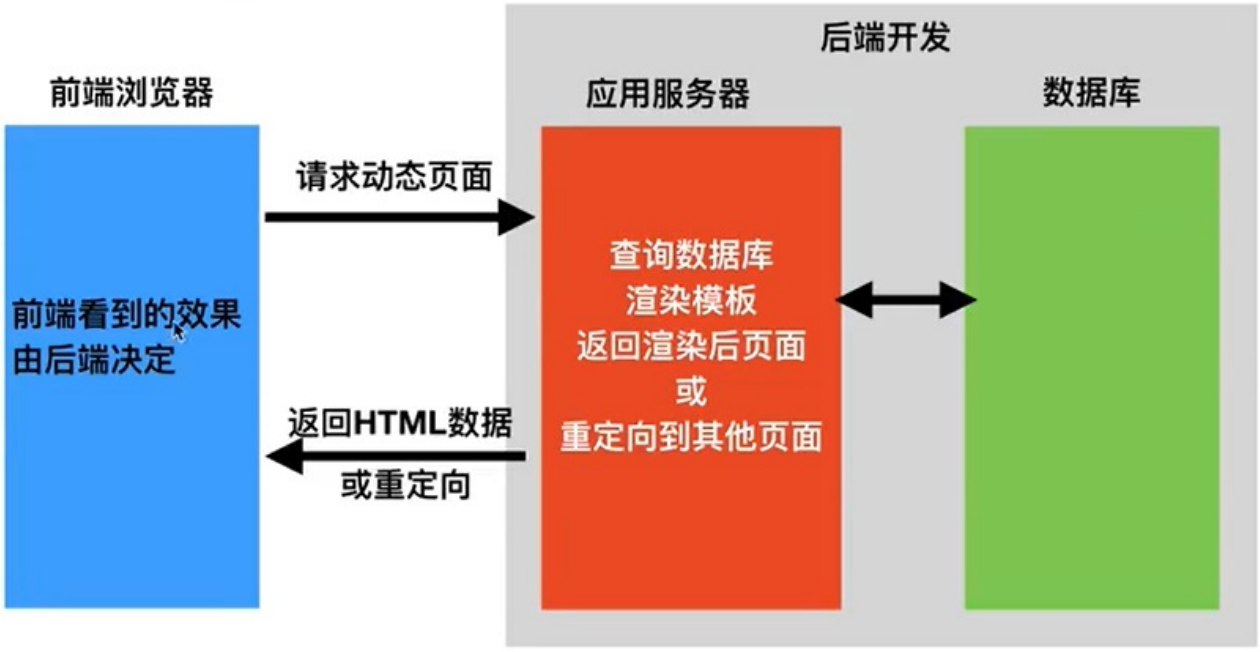
使用

1. 引入DjangoRESTframework

1.1 Web应用模式

前后端不分离

1 前后端不分离

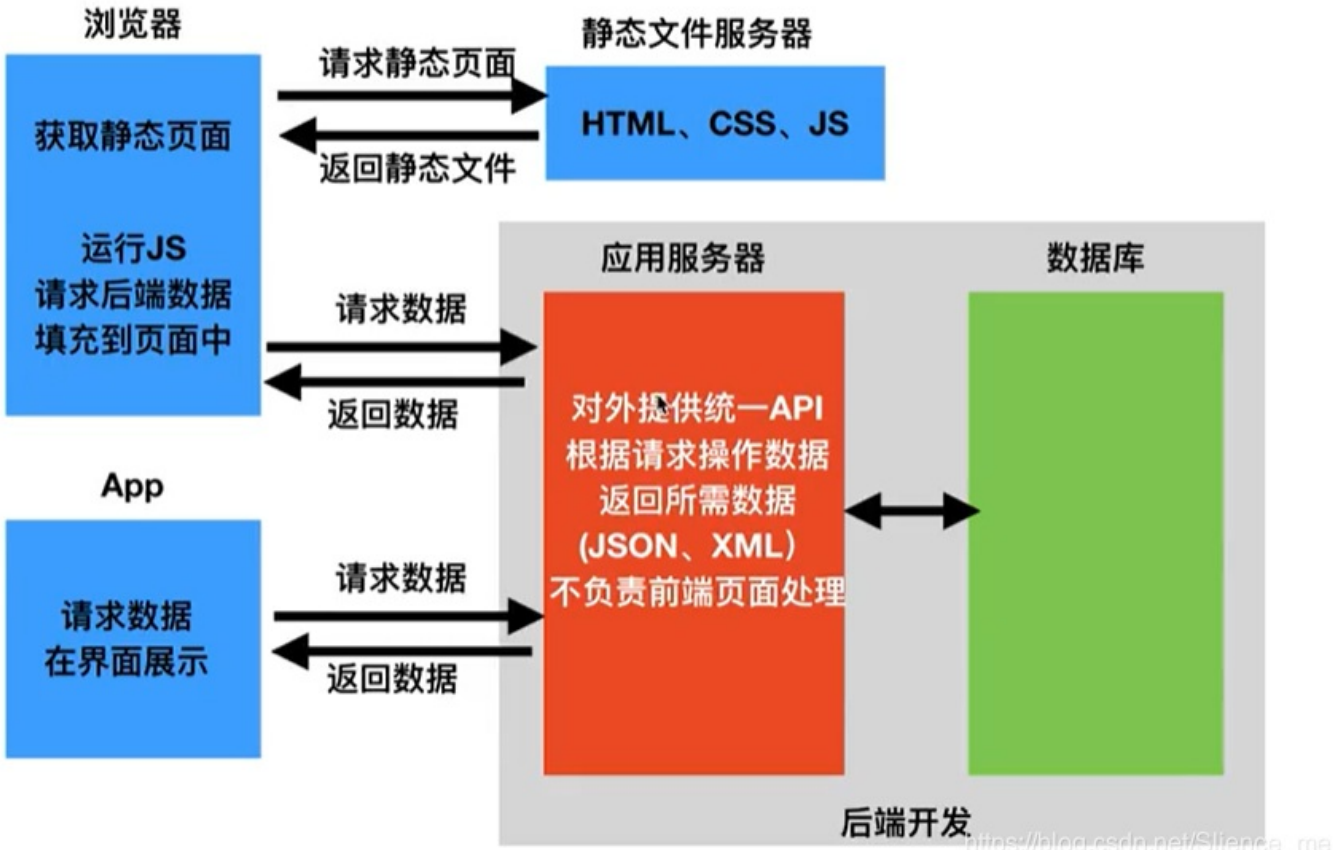


在前后端不分离的应用模式中，前端页面看到的效果都是由后端控制，由后端渲染页面或重定向，也就是后端需要控制前端的展示，前端与后端的耦合度很高。

这种应用模式比较适合纯网页应用，但是当后端对接App时，App可能并不需要后端返回一个HTML网页，而仅仅是数据本身，所以后端原本返回网页的接口不再适用于前端App应用，为了对接App后端还需再开发一套接口。

前后端分离

2 前后端分离



在前后端分离的应用模式中，后端仅返回前端所需的数据，不再渲染HTML页面，不再控制前端的效果。至于前端用户看到什么效果，从后端请求的数据如何加载到前端中，都由前端自己决定，网页有网页的处理方式，App有App的处理方式，但无论哪种前端，所需的数据基本相同，后端仅需开发一套逻辑对外提供数据即可。

在前后端分离的应用模式中 ，前端与后端的耦合度相对较低。

在前后端分离的应用模式中，我们通常将后端开发的每个视图都称为一个接口，或者API，前端通过访问接口来对数据进行增删改查。

1.2 认识RESTful

在前后端分离的应用模式里，API接口如何定义？

> 对于接口的请求方式与路径，每个后端开发人员可能都有自己的定义方式，风格迥异。

> 是否存在一种统一的定义方式，被广大开发人员接受认可的方式呢？

> 这就是被普遍采用的API的RESTful设计风格。

> 例如对于后端数据库中保存了商品的信息，前端可能需要对商品数据进行增删改查，那相应的每个操作后端都需要提供一个API接口：

请求方法	请求地址	后端操作
GET	/goods	获取所有商品
POST	/goods	增加商品
GET	/goods/1	获取编号为1的商品
PUT	/goods/1	修改编号为1的商品
DELETE	/goods/1	删除编号为1的商品

1.3 RESTful设计方法

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://example.org/api/
```

2. 版本（Versioning）

应该将API的版本号放入URL。

```
http://www.example.com/app/1.0/foo

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

http://www.example.com/app/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。

因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个URL。版本号可以在HTTP请求头信息的Accept字段中进行区分（参见Versioning REST Services）：

```
Accept: vnd.example-com.foo+json; version=1.0

Accept: vnd.example-com.foo+json; version=1.1

Accept: vnd.example-com.foo+json; version=2.0
```

3. 路径（Endpoint）

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) 资源作为网址，只能有名词，不能有动词，而且所用的名词往往与数据库的表名对应。

举例来说，以下是不好的例子:

```
/getProducts
/listOrders
/retreiveClientByOrder?orderId=1
```

对于一个简洁结构，你应该始终用名词。 此外，利用的HTTP方法可以分离网址中的资源名称的操作。

```
GET /products ：将返回所有产品清单
POST /products ：将产品新建到集合
GET /products/4 ：将获取产品 4
PATCH（或）PUT /products/4 ：将更新产品 4
```

(2) API中的名词应该使用复数。无论子资源或者所有资源。

举例来说，获取产品的API可以这样定义

> 获取单个产品：http://127.0.0.1:8080/AppName/rest/products/1 获取所有产品:
> http://127.0.0.1:8080/AppName/rest/products

4. HTTP动词

对于资源的具体操作类型，由HTTP动词表示。

常用的HTTP动词有下面四个（括号里是对应的SQL命令）。

> GET（SELECT）：从服务器取出资源（一项或多项）。
> POST（CREATE）：在服务器新建一个资源。
> PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）。
> DELETE（DELETE）：从服务器删除资源。

还有三个不常用的HTTP动词。

> PATCH（UPDATE）：在服务器更新(更新)资源（客户端提供改变的属性）。
> HEAD：获取资源的元数据。
> OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的。 下面是一些例子。
> GET /zoos：列出所有动物园
> POST /zoos：新建一个动物园（上传文件）
> GET /zoos/ID：获取某个指定动物园的信息
> PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）
> PATCH/zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）
> DELETE /zoos/ID：删除某个动物园
> GET/zoos/ID/animals：列出某个指定动物园的所有动物
> DELETE/zoos/ID/animals/ID：删除某个指定动物园的指定动物

5. 过滤信息（Filtering）

如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。

下面是一些常见的参数。

?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&per_page=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。
?animal_type_id=1：指定筛选条件

参数的设计允许存在冗余，即允许API路径和URL参数偶尔有重复。比如，GET /zoos/ID/animals 与 GET /animals?zoo_id=ID 的含义是相同的。

6. 状态码（Status Codes）

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的HTTP动词）。

200 OK - [GET]：服务器成功返回用户请求的数据
201 CREATED -[POST/PUT/PATCH]：用户新建或修改数据成功
202 Accepted - []：*表示一个请求已经进入后台排队（异步任务）*
204 NO CONTENT - [DELETE]：用户删除数据成功。
400 INVALID REQUEST - [POST/PUT/PATCH]：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
401 Unauthorized - []：表示用户没有权限（令牌、用户名、密码错误）。
403 Forbidden - [*]表示用户得到授权（与401错误相对），但是访问是被禁止的。*
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。
406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。
410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。
422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
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。 状态码的完全列表参见这里或这里。

7. 错误处理（Error handling）

如果状态码是4xx，服务器就应该向用户返回出错信息。一般来说，返回的信息中将error作为键名，出错信息作为键值即可。

```
{
error: “Invalid API key”
}
```

8. 返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

GET /collection：返回资源对象的列表（数组）
GET /collection/resource：返回单个资源对象
POST /collection：返回新生成的资源对象
PUT /collection/resource：返回完整的资源对象
PATCH /collection/resource：返回完整的资源对象
DELETE /collection/resource：返回一个空文档

9. 超媒体（Hypermedia API）

RESTful API最好做到Hypermedia（即返回结果中提供链接，连向其他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。

10. 其他

服务器返回的数据格式，应该尽量使用JSON，避免使用XML。

1.4 使用Django开发REST 接口

我们以在Django框架中使用的图书英雄案例来写一套支持图书数据增删改查的REST API接口，来理解REST API的开发。

在此案例中，前后端均发送JSON格式数据。

```python
# views.py

from datetime import datetime

class BooksAPIView(View):
    """
    查询所有图书、增加图书
    """
    def get(self, request):
        """
        查询所有图书
        路由：GET /books/
        """
        queryset = BookInfo.objects.all()
        book_list = []
        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)

    def post(self, request):
        """
        新增图书
        路由：POST /books/
        """
        json_bytes = request.body
        json_str = json_bytes.decode()
        book_dict = json.loads(json_str)

        # 此处详细的校验参数省略

        book = BookInfo.objects.create(
            btitle=book_dict.get('btitle'),
            bpub_date=datetime.strptime(book_dict.get('bpub_date'), '%Y-%m-%d').date()
        )

        return JsonResponse({
            '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 ''
        }, status=201)


class BookAPIView(View):
    def get(self, request, pk):
        """
        获取单个图书信息
        路由： GET  /books/<pk>/
        """
        try:
            book = BookInfo.objects.get(pk=pk)
        except BookInfo.DoesNotExist:
            return HttpResponse(status=404)

        return JsonResponse({
            '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 ''
        })

    def put(self, request, pk):
        """
        修改图书信息
        路由： PUT  /books/<pk>
        """
        try:
            book = BookInfo.objects.get(pk=pk)
        except BookInfo.DoesNotExist:
            return HttpResponse(status=404)

        json_bytes = request.body
```

```
        json_str = json_bytes.decode()
        book_dict = json.loads(json_str)

        # 此处详细的校验参数省略

        book.btitle = book_dict.get('btitle')
        book.bpub_date = datetime.strptime(book_dict.get('bpub_date'), '%Y-%m-%d').date()
        book.save()

        return JsonResponse({
            '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 ''
        })

    def delete(self, request, pk):
        """
        删除图书
        路由： DELETE /books/<pk>/
        """
        try:
            book = BookInfo.objects.get(pk=pk)
        except BookInfo.DoesNotExist:
            return HttpResponse(status=404)

        book.delete()

        return HttpResponse(status=204)
```

```
# urls.py

urlpatterns = [
    url(r'^books/$', views.BooksAPIVIew.as_view()),
    url(r'^books/(?P<pk>\d+)/$', views.BookAPIView.as_view())
]
```

测试

使用Postman测试上述接口

> 1） 获取所有图书数据
> GET 方式访问 http://127.0.0.1:8000/books/， 返回状态码200，数据如下

```
[
    {
        "id": 1,
        "btitle": "射雕英雄传",
        "bpub_date": "1980-05-01",
        "bread": 12,
        "bcomment": 34,
        "image": ""
    },
    {
        "id": 2,
        "btitle": "天龙八部",
        "bpub_date": "1986-07-24",
        "bread": 36,
        "bcomment": 40,
        "image": ""
    },
    {
        "id": 3,
        "btitle": "笑傲江湖",
        "bpub_date": "1995-12-24",
        "bread": 20,
        "bcomment": 80,
        "image": ""
    },
    {
        "id": 4,
        "btitle": "雪山飞狐",
        "bpub_date": "1987-11-11",
        "bread": 58,
        "bcomment": 24,
        "image": ""
    },
    {
        "id": 5,
        "btitle": "西游记",
        "bpub_date": "1988-01-01",
        "bread": 10,
        "bcomment": 10,
        "image": "booktest/xiyouji.png"
    },
    {
        "id": 6,
        "btitle": "水浒传",
        "bpub_date": "1992-01-01",
        "bread": 10,
        "bcomment": 11,
        "image": ""
    },
    {
        "id": 7,
        "btitle": "红楼梦",
        "bpub_date": "1990-01-01",
        "bread": 0,
        "bcomment": 0,
        "image": ""
    }
]
```

2）获取单一图书数据
GET 访问 http://127.0.0.1:8000/books/5/ ，返回状态码200， 数据如下

```
{
    "id": 5,
    "btitle": "西游记",
    "bpub_date": "1988-01-01",
    "bread": 10,
    "bcomment": 10,
    "image": "booktest/xiyouji.png"
}
```

GET 访问http://127.0.0.1:8000/books/100/，返回状态码404
3）新增图书数据
POST 访问http://127.0.0.1:8000/books/，发送JSON数据：

```
{
    "btitle": "三国演义",
    "bpub_date": "1990-02-03"
}
```

返回状态码201，数据如下

```
{
    "id": 8,
    "btitle": "三国演义",
    "bpub_date": "1990-02-03",
    "bread": 0,
    "bcomment": 0,
    "image": ""
}
```

4）修改图书数据 PUT 访问http://127.0.0.1:8000/books/8/，发送JSON数据：

```
{
    "btitle": "三国演义（第二版）",
    "bpub_date": "1990-02-03"
}
```

返回状态码200，数据如下

```
{
    "id": 8,
    "btitle": "三国演义（第二版）",
    "bpub_date": "1990-02-03",
    "bread": 0,
    "bcomment": 0,
    "image": ""
}
```

5）删除图书数据
DELETE 访问http://127.0.0.1:8000/books/8/，返回204状态码

源码

```
import json

from django.http import JsonResponse, HttpResponse
from django.shortcuts import render

"""
GET         /books/
POST        /books/
GET         /books/<pk>/
PUT         /books/<pk>/
DELETE      /books/<pk>/

响应数据  JSON
# 列表视图: 路由后边没有pk/ID
# 详情视图: 路由后面   pk/ID
"""
from django.views import View

from .models import BookInfo


class BookListView(View):

    def get(self, request):
        """查询所有图书接口"""
        # 1. 查询出所有图书模型
        books = BookInfo.objects.all()
        # 2. 遍历查询集，去除里边的每个书籍模型对象，把模型对象转换成字典
        # 定义一个列表保存所有字典
        book_list = []
        for book in books:
            book_dict = {
                'id': book.id,
                'btitle': book.btitle,
                'bput_date': book.bpub_date,
                'bread': book.bcomment,
                'image': book.image.url if book.image else '',
            }
            book_list.append(book_dict)  # 将转换好的字典添加到列表中
        # 3. 响应给前端
        # 如果book_;list 不是一个字典的话就需要将safe设置成False.
        return JsonResponse(book_list, safe=False)

    def post(self, request):
        """新增图书接口"""
        # 获取前端传入的请求体数据(json) request.body
        json_str_bytes = request.body
        # 把bytes类型的json字符串转换成json_str
        json_str = json_str_bytes.decode()
        # 利用json.loads将json字符串扎UN干哈UN从json（字典/列表）
        book_dict = json.loads(json_str)
        # 创建模型对象并保存（把字典转换成模型并储存）
        book = BookInfo(
```

```
            btitle=book_dict['btitle'],
            bpub_date=book_dict['bpub_date'],

        )
        book.save()

        # 把新增的模型转换成字典
        json_dict = {
            'id': book.id,
            'btitle': book.btitle,
            'bput_date': book.bpub_date,
            'bread': book.bread,
            'bcomment':book.bcomment,
            'image': book.image.url if book.image else '',
        }
        # 响应（把新增的数据再响应回去，201）
        return JsonResponse(json_dict,status=201)



class BookDetailView(View):
    """详情视图"""

    def get(self, request, pk):
        """查询指定某个图书馆接口"""
        # 1. 获取出指定pk的那个模型对象
        try:
            book = BookInfo.objects.get(id=pk)
        except BookInfo.DoesNotExist:
            return HttpResponse({'message': '查询的数据不存在'}, status=404)
        # 2. 模型对象转字典
        book_dict = {
            'id': book.id,
            'btitle': book.btitle,
            'bput_date': book.bpub_date,
            'bread': book.bread,
            'bcomment':book.bcomment,
            'image': book.image.url if book.image else '',
        }
        # 3. 响应
        return JsonResponse(book_dict)

    def put(self, request, pk):
        """修改指定图书馆接口"""
        # 先查询要修改的模型对象
        try:
            book = BookInfo.objects.get(pk=pk)
        except BookInfo.DoesNotExist:
            return HttpResponse({'message': '查询的数据不存在'}, status=404)
        # 获取前端传入的新数据（把数据转换成字典）
        # json_str_bytes = request.body
        # json_str = json_str_bytes.decode()
        # book_dict = json.loads(json_str)

        book_dict = json.loads(request.body.decode())
        # 重新给模型指定的属性赋值
        book.btitle = book_dict['btitle']
        book.bpub_date = book_dict['bpub_date']

        # 调用save方法进行修改操作
```

```
        # 调用save方法进行修改操作
        book.save()
        # 把修改后的模型再转换成字典
        json_dict = {
            'id': book.id,
            'btitle': book.btitle,
            'bput_date': book.bpub_date,
            'bread': book.bread,
            'bcomment': book.bcomment,
            'image': book.image.url if book.image else '',
        }
        # 响应
        return JsonResponse(json_dict)

    def delete(self, request, pk):
        """删除指定图书接口"""
        # 获取要删除的模型对象
        try:
            book = BookInfo.objects.get(id=pk)
        except BookInfo.DoesNotExist:
            return HttpResponse({'message':'查询的数据不存在'},status=404)
        # 删除指定模型对象
        book.delete()  # 物理删除（真正从数据库删除）
        # book.is_delete = True
        # book.save()  # （逻辑删除）
        # 响应：删除时不需要有响应体但要指定状态码为 204
        return HttpResponse(status=204)
```

```python
from django.db import models


# Create your models here.
# 定义图书模型类BookInfo
class BookInfo(models.Model):
    btitle = models.CharField(max_length=20, verbose_name='名称')
    bpub_date = models.DateField(verbose_name='发布日期')
    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='逻辑删除')
    # 注意,如果模型已经迁移建表并且表中如果已经有数据了,那么后新增的字段,必须给默认值或可以为空,不然迁移就报错
    # upload_to 指定上传到media_root配置项的目录中再创建booktest里面
    image = models.ImageField(upload_to='booktest', verbose_name='图片', null=True)


    class Meta:
        db_table = 'tb_books'  # 指明数据库表名
        verbose_name = '图书'  # 在admin站点中显示的名称
        verbose_name_plural = verbose_name  # 显示的复数名称

    def __str__(self):
        """定义每个数据对象的显示信息"""
        return self.btitle

    def pub_date_format(self):
        return self.bpub_date.strftime('%Y-%m-%d')
    # 修改方法名在列表界面的展示
    pub_date_format.short_description = '发布日期'
    # 指定自定义方法的排序依据
    pub_date_format.admin_order_field = 'bpub_date'


# 定义英雄模型类HeroInfo
class HeroInfo(models.Model):
    GENDER_CHOICES = (
        (0, 'female'),
        (1, 'male')
    )
    hname = models.CharField(max_length=20, verbose_name='名称')
    hgender = models.SmallIntegerField(choices=GENDER_CHOICES, default=0, verbose_name='性别')
    hcomment = models.CharField(max_length=200, null=True, verbose_name='描述信息')
    hbook = models.ForeignKey(BookInfo, on_delete=models.CASCADE, verbose_name='图书')  # 外键
    is_delete = models.BooleanField(default=False, verbose_name='逻辑删除')

    class Meta:
        db_table = 'tb_heros'
        verbose_name = '英雄'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.hname

    def read(self):
        return self.hbook.bread
    read.short_description = '阅读量'
    read.admin_order_field = 'hbook__bread'
    # HeroInfo.objects.filter(hbook__bread=xx)
```

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

urlpatterns = [
    # 列表视图的路由
    url(r'^books/$', views.BookListView.as_view()),
    # 详情视图的路由
    url(r'^books/(?P<pk>\d+)/$', views.BookDetailView.as_view()),
]
```

```
# 配置项目中静态文件存放/读取目录
STATICFILES_DIRS = [
    # http://127.0.0.1:8000/static/index.html
    # http://127.0.0.1:8000/static/mm03.jpg
    os.path.join(BASE_DIR, 'static_files'),
    os.path.join(BASE_DIR, 'static_files/good'),
]

# 指定上传文件存储目录
MEDIA_ROOT=os.path.join(BASE_DIR,"static_files/media")
```

1.5 明确REST接口开发的核心任务

分析一下上节的案例，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）

序列化Serialization

> 维基百科中对于序列化的定义：
> 序列化（serialization）在计算机科学的资料处理中，是指将数据结构或物件状态转换成可取用格式（例如存成档案，存于缓冲，或经由网络中传送），以留待后续在相同或另一台计算机环境中，能恢复原先状态的过程。依照序列化格式重新获取字节的结果时，可以利用它来产生与原始物件相同语义的副本。对于许多物件，像是使用大量参照的复杂物件，这种序列化重建的过程并不容易。面向对象中的物件序列化，并不概括之前原始物件所关联的函式。这种过程也称为物件编组（marshalling）。从一系列字节提取数据结构的反向操作，是反序列化（也称为解编组,
> deserialization, unmarshalling）。
> 序列化在计算机科学中通常有以下定义:
>
> 在数据储存与传送的部分是指将一个对象)存储至一个储存媒介，例如档案或是记忆体缓冲等，或者透过网络传送资料时进行编码的过程，可以是字节或是XML等格式。而字节的或XML编码格式可以还原完全相等的对象)。这程序被应用在不同应用程序之间传送对象)，以及服务器将对象)储存到档案或数据库。相反的过程又称为反序列化。
> 简而言之，我们可以将序列化理解为：
> 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中的模型类对象装换为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)
```

> 反之，将其他格式（字典、JSON、XML等）转换为程序中的数据，例如将JSON字符串转换为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=datetime.strptime(book_dict.get('bpub_date'), '%Y-%m-%d').date()
)
```

> 我们可以看到，在开发REST API时，视图中要频繁的进行序列化与反序列化的编写。
> 总结 在开发REST API接口时，我们在视图中需要做的最核心的事是：
>
> - 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为模型类对象，并保存到数据库中。

1.6 Django REST framework 简介

> 1. 在序列化与反序列化时，虽然操作的数据不尽相同，但是执行的过程却是相似的，也就是说这部分代码是可以复用简化编写的。
> 2. 在开发RESTAPI的视图中，虽然每个视图具体操作的数据不同，但增、删、改、查的实现流程基本套路化，所以这部分代码也是可以复用简化编写的：
>
> - 增：校验请求数据 -> 执行反序列化过程 -> 保存数据库 -> 将保存的对象序列化并返回
> - 删：判断要删除的数据是否存在 ->执行数据库删除
> - 改：判断要修改的数据是否存在 -> 校验请求的数据 -> 执行反序列化过程 -> 保存数据库 -> 将保存的对象序列化并返回
> - 查：查询数据库 -> 将数据序列化并返回

> Django REST framework可以帮助我们简化上述两部分的代码编写，大大提高REST API的开发速度。

认识Django REST framework

> 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 或 REST framework。
> 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。

特点

> - 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；
> - 提供了丰富的类视图、Mixin扩展类，简化视图的编写；
> - 丰富的定制层级：函数视图、类视图、视图集合到自动生成 API， 满足各种需要；
> - 多种身份认证和权限认证方式的支持；
> - 内置了限流系统；
> - 直观的 API web 界面；
> - 可扩展性，插件丰富

2. DRF工程搭建

环境安装与配置

> DRF需要以下依赖：

- Python (2.7, 3.4, 3.5, 3.6, 3.7)
- Django (1.11, 2.0, 2.1)

> DRF是以Django扩展应用的方式提供的，所以我们可以直接利用已有的Django环境而无需从新创建。（若没有Django环境，需要先创建环境安装Django）

> 1. 安装DRF

```
pip install djangorestframework
```

2. 添加rest_framework应用

我们利用在Django框架学习中创建的demo工程，在settings.py的INSTALLED_APPS中添加'rest_framework'。

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

接下来就可以使用DRF进行开发了。

见识DRF的魅力

我们仍以在学习Django框架时使用的图书英雄为案例，使用Django REST framework快速实现图书的REST API。

1. 创建序列化器

在booktest应用中新建serializers.py用于保存该应用的序列化器。
创建一个BookInfoSerializer用于序列化与反序列化。

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

- model 指明该序列化器处理的数据字段从模型类BookInfo参考生成
- fields 指明该序列化器包含模型类中的哪些字段，'all'指明包含所有字段

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 .serializers import BookInfoSerializer
from .models import BookInfo

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

- queryset 指明该视图集在查询数据时使用的查询集
- serializer_class 指明该视图在进行序列化或反序列化时使用的序列化器

3. 定义路由

在booktest应用的urls.py中定义路由信息。

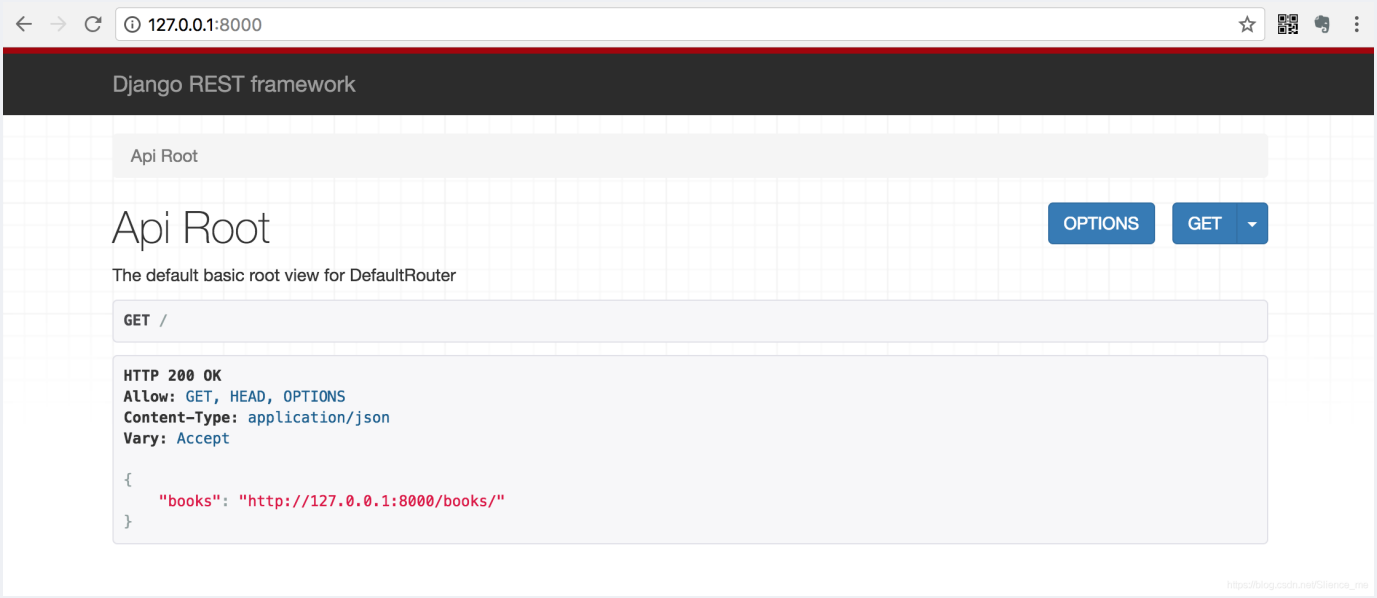
```
from . import views
from rest_framework.routers import DefaultRouter

urlpatterns = [
    ...
]

router = DefaultRouter()  # 可以处理视图的路由器
router.register(r'books', views.BookInfoViewSet)  # 向路由器中注册视图集

urlpatterns += router.urls  # 将路由器中的所以路由信息追到到django的路由列表中
```

4. 运行测试

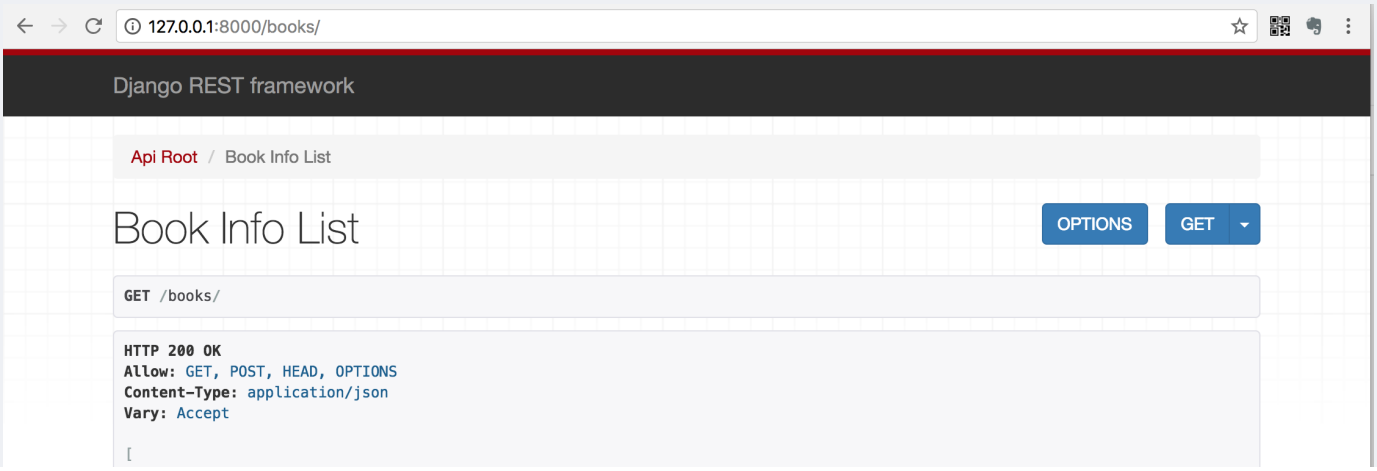
运行当前程序（与运行Django一样）

```
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：



1）点击链接127.0.0.1:8000/books/ 可以访问获取所有数据的接口，呈现如下页面：



```
    {
        "id": 1,
        "btitle": "射雕英雄传",
        "bpub_date": "1980-05-01",
        "bread": 12,
        "bcomment": 34,
        "image": null
    },
    {
        "id": 2,
        "btitle": "天龙八部",
        "bpub_date": "1986-07-24",
        "bread": 36,
        "bcomment": 40,
        "image": null
    },
    {
```



```
        "bpub_date": "1990-02-03",
        "bread": 0,
        "bcomment": 0,
        "image": null
    }
]
```


Raw data | HTML form

名称

发布日期 年 /月/日

阅读量

评论量

图片 选择文件 未选择任何文件

POST

2）在页面底下表单部分填写图书信息，可以访问添加新图书的接口，保存新书：

Raw data | HTML form

名称 神雕侠侣

发布日期 1996/01/01

阅读量

评论量

图片 选择文件 未选择任何文件

POST

点击POST后，返回如下页面信息：

Book Info List

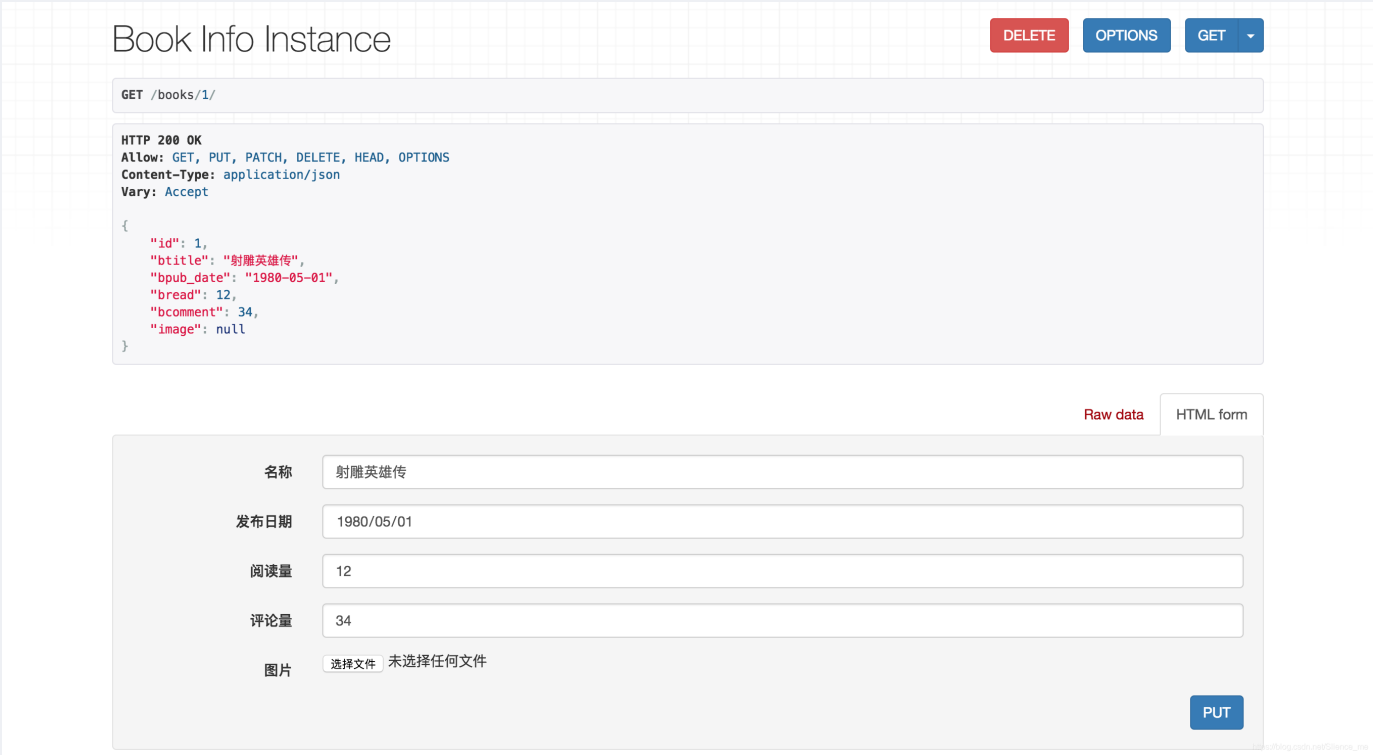
OPTIONS GET

```
POST /books/
```

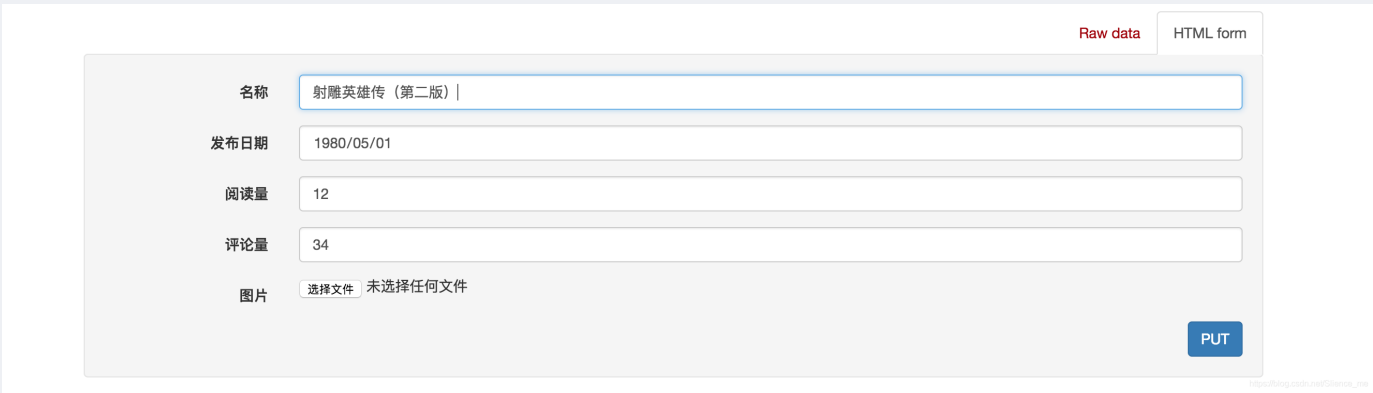
```
HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
```

```
{
    "id": 10,
    "btitle": "神雕侠侣",
    "bpub_date": "1996-01-01",
    "bread": 0,
    "bcomment": 0,
    "image": null
}
```

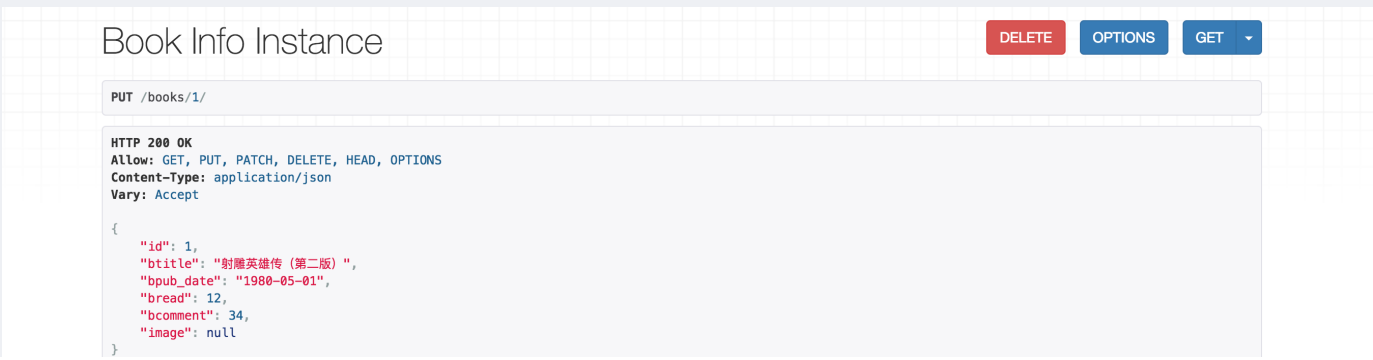

3）在浏览器中输入网址127.0.0.1:8000/books/1/，可以访问获取单一图书信息的接口（id为1的图书），呈现如下页面：



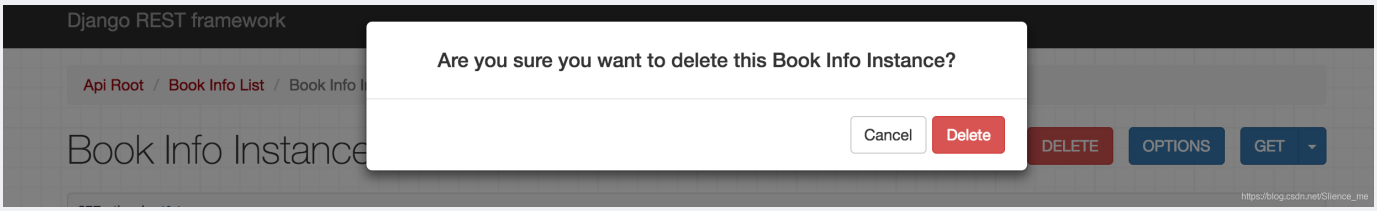
4）在页面底部表单中填写图书信息，可以访问修改图书的接口：



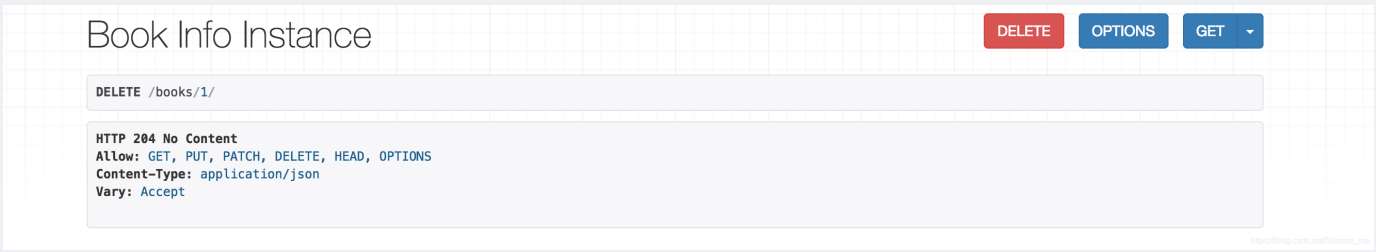
点击PUT，返回如下页面信息：



5）点击DELETE按钮，可以访问删除图书的接口：



返回，如下页面：



至此，是不是发现Django REST framework很好用！

3. Serializer序列化器

序列化器的作用：

1. 进行数据的校验
2. 对数据对象进行转换

3.1 定义Serializer

1. 定义方法

Django REST framework中的Serializer使用类来定义，须继承自rest_framework.serializers.Serializer。
例如，我们已有了一个数据库模型类BookInfo

```
class BookInfo(models.Model):
    btitle = models.CharField(max_length=20, verbose_name='名称')
    bpub_date = models.DateField(verbose_name='发布日期')
    bread = models.IntegerField(default=0, verbose_name='阅读量')
    bcomment = models.IntegerField(default=0, verbose_name='评论量')
    image = models.ImageField(upload_to='booktest', verbose_name='图片', null=True)
```

我们想为这个模型类提供一个序列化器，可以定义如下：

```python
class BookInfoSerializer(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', required=True)
    bread = serializers.IntegerField(label='阅读量', required=False)
    bcomment = serializers.IntegerField(label='评论量', required=False)
    image = serializers.ImageField(label='图片', required=False)
```

> 注意：serializer不是只能为数据库模型类定义，也可以为非数据库模型类的数据定义。serializer是独立于数据库之外的存在。

2. 字段与选项

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: `"12345678901231231313412451235114514 5114"` 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)
ListField	ListField(child=, min_length=None, max_length=None)
DictField	DictField(child=)

选项参数：

参数名称	作用
max_length	最大长度
min_lenght	最小长度
allow_blank	是否允许为空
trim_whitespace	是否截断空白字符
max_value	最小值
min_value	最大值

通用参数：

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	反序列化时使用的默认值
allow_null	表明该字段是否允许传入None，默认False
validators	该字段使用的验证器
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
help_text	用于HTML展示API页面时，显示的字段帮助提示信息

3. 创建Serializer对象

> 定义好Serializer类后，就可以创建Serializer对象了。

Serializer的构造方法为：

```
Serializer(instance=None, data=empty, **kwarg)
```

> 说明：
> 1）用于序列化时，将模型类对象传入instance参数
> 2）用于反序列化时，将要被反序列化的数据传入data参数
> 3）除了instance和data参数外，在构造Serializer对象时，还可通过context参数额外添加数据，如

```
serializer = AccountSerializer(account, context={'request': request})
```

> 通过context参数附加的数据，可以通过Serializer对象的context属性获取。

3.2 序列化使用

> 我们在django shell中来学习序列化器的使用。

```
python manage.py shell
```

1 基本使用

> 1） 先查询出一个图书对象

```
from booktest.models import BookInfo

book = BookInfo.objects.get(id=2)
```

> 2） 构造序列化器对象

```
from booktest.serializers import BookInfoSerializer

serializer = BookInfoSerializer(book)
```

> 3）获取序列化数据
> 通过data属性可以获取序列化后的数据

```
serializer.data
# {'id': 2, 'btitle': '天龙八部', 'bpub_date': '1986-07-24', 'bread': 36, 'bcomment': 40, 'image': None}
```

> 4）如果要被序列化的是包含多条数据的查询集QuerySet，可以通过添加many=True参数补充说明

```
book_qs = BookInfo.objects.all()
serializer = BookInfoSerializer(book_qs, many=True)
serializer.data
# [OrderedDict([('id', 2), ('btitle', '天龙八部'), ('bpub_date', '1986-07-24'), ('bread', 36), ('bcomment', 40),
('image', N]), OrderedDict([('id', 3), ('btitle', '笑傲江湖'), ('bpub_date', '1995-12-24'), ('bread', 20), ('bcom
ment', 80), ('image'ne)]), OrderedDict([('id', 4), ('btitle', '雪山飞狐'), ('bpub_date', '1987-11-11'), ('bread',
 58), ('bcomment', 24), ('ima None)]), OrderedDict([('id', 5), ('btitle', '西游记'), ('bpub_date', '1988-01-01')
, ('bread', 10), ('bcomment', 10), ('im', 'booktest/xiyouji.png')])]
```

2 关联对象嵌套序列化

> 如果需要序列化的数据中包含有其他关联对象，则对关联对象数据的序列化需要指明。
> 例如，在定义英雄数据的序列化器时，外键hbook（即所属的图书）字段如何序列化？
> 我们先定义HeroInfoSerialzier除外键字段外的其他部分

```
class HeroInfoSerializer(serializers.Serializer):
    """英雄数据序列化器"""
    GENDER_CHOICES = (
        (0, 'female'),
        (1, 'male')
    )
    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, allow_null=True)
```

> 对于关联字段，可以采用以下几种方式：

1 PrimaryKeyRelatedField

> 此字段将被序列化为关联对象的主键。

```
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：

- 包含read_only=True参数时，该字段将不能用作反序列化使用
- 包含queryset参数时，将被用作反序列化时参数校验使用

使用效果：

```
from booktest.serializers import HeroInfoSerializer
from booktest.models import HeroInfo
hero = HeroInfo.objects.get(id=6)
serializer = HeroInfoSerializer(hero)
serializer.data
# {'id': 6, 'hname': '乔峰', 'hgender': 1, 'hcomment': '降龙十八掌', 'hbook': 2}
```

2 StringRelatedField

此字段将被序列化为关联对象的字符串表示方式（即__str__方法的返回值）

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

使用效果

```
{'id': 6, 'hname': '乔峰', 'hgender': 1, 'hcomment': '降龙十八掌', 'hbook': '天龙八部'}
```

3 使用关联对象的序列化器

```
hbook = BookInfoSerializer()
```

使用效果

```
{'id': 6, 'hname': '乔峰', 'hgender': 1, 'hcomment': '降龙十八掌', 'hbook': OrderedDict([('id', 2), ('btitle', '天龙八部')te', '1986-07-24'), ('bread', 36), ('bcomment', 40), ('image', None)])}
```

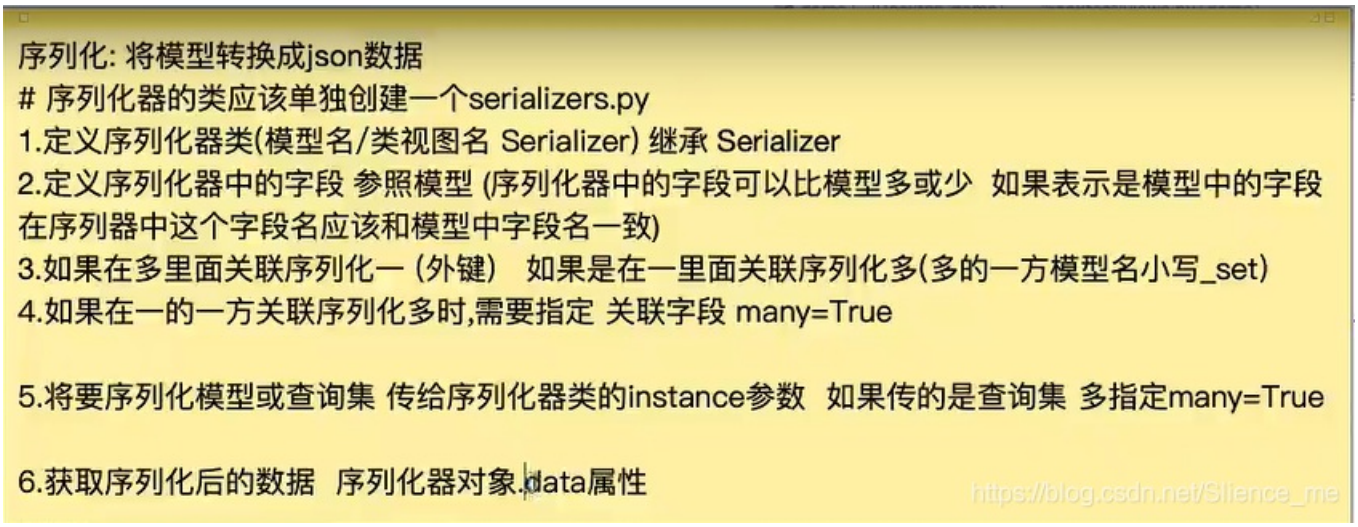
3 many参数

> 如果关联的对象数据不是只有一个，而是包含多个数据，如想序列化图书BookInfo数据，每个BookInfo对象关联的英雄HeroInfo对象可能有多个，此时关联字段类型的指明仍可使用上述几种方式，只是在声明关联字段时，多补充一个many=True参数即可。
> 此处仅拿PrimaryKeyRelatedField类型来举例，其他相同。
> 在BookInfoSerializer中添加关联字段：

```
class BookInfoSerializer(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', required=True)
    bread = serializers.IntegerField(label='阅读量', required=False)
    bcomment = serializers.IntegerField(label='评论量', required=False)
    image = serializers.ImageField(label='图片', required=False)
    heroinfo_set = serializers.PrimaryKeyRelatedField(read_only=True, many=True)  # 新增
```

使用效果：

```
from booktest.serializers import BookInfoSerializer
from booktest.models import BookInfo
book = BookInfo.objects.get(id=2)
serializer = BookInfoSerializer(book)
serializer.data
# {'id': 2, 'btitle': '天龙八部', 'bpub_date': '1986-07-24', 'bread': 36, 'bcomment': 40, 'image': None, 'heroinfo_set': [6,8, 9]}
```



序列化只是一个单纯 模型转字典

3.3 反序列化使用

1. 验证

> 使用序列化器进行反序列化时，需要对数据进行验证后，才能获取验证成功的数据或保存成模型类对象。

> 在获取反序列化的数据前，必须调用is_valid()方法进行验证，验证成功返回True，否则返回False。

> 验证失败，可以通过序列化器对象的errors属性获取错误信息，返回字典，包含了字段和字段的错误。如果是非字段错误，可以通过修改REST framework配置中的NON_FIELD_ERRORS_KEY来控制错误字典中的键名。

> 验证成功，可以通过序列化器对象的validated_data属性获取数据。

> 在定义序列化器时，指明每个字段的序列化类型和选项参数，本身就是一种验证行为。

> 如我们前面定义过的BookInfoSerializer

```
class BookInfoSerializer(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', required=True)
    bread = serializers.IntegerField(label='阅读量', required=False)
    bcomment = serializers.IntegerField(label='评论量', required=False)
    image = serializers.ImageField(label='图片', required=False)
```

> 通过构造序列化器对象，并将要反序列化的数据传递给data构造参数，进而进行验证

```
from booktest.serializers import BookInfoSerializer
data = {'bpub_date': 123}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()  # 返回False
serializer.errors
# {'btitle': [ErrorDetail(string='This field is required.', code='required')], 'bpub_date': [ErrorDetail(string=
'Date has wrong format. Use one of these formats instead: YYYY[-MM[-DD]].', code='invalid')]}
serializer.validated_data  # {}

data = {'btitle': 'python', 'bpub_date': '1998-12-1'}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()  # True
serializer.errors  # {}
serializer.validated_data  #  OrderedDict([('btitle', 'python')])
```

> is_valid()方法还可以在验证失败时抛出异常serializers.ValidationError，可以通过传递raise_exception=True参数开启，REST framework接收到此异常，会向前端返回HTTP 400 Bad Request响应。

```
# Return a 400 response if the data was invalid.
serializer.is_valid(raise_exception=True)
```

> 如果觉得这些还不够，需要再补充定义验证行为，可以使用以下三种方法：

1）validate_<field_name>

> 对<field_name>字段进行验证，如

```
class BookInfoSerializer(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
```

> 测试

```
from booktest.serializers import BookInfoSerializer
data = {'btitle': 'python'}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()  # False
serializer.errors
#  {'btitle': [ErrorDetail(string='图书不是关于Django的', code='invalid')]}
```

2）validate

> 在序列化器中需要同时对多个字段进行比较验证时，可以定义validate方法来验证，如

```
class BookInfoSerializer(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
```

> 测试

```
from booktest.serializers import BookInfoSerializer
data = {'btitle': 'about django', 'bpub_date': '1998-12-1', 'bread': 10, 'bcomment': 20}
s = BookInfoSerializer(data=data)
s.is_valid()  # False
s.errors
#  {'non_field_errors': [ErrorDetail(string='阅读量小于评论量', code='invalid')]}
```

3）validators

在字段中添加validators选项参数，也可以补充验证行为，如

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

class BookInfoSerializer(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', required=True)
    bread = serializers.IntegerField(label='阅读量', required=False)
    bcomment = serializers.IntegerField(label='评论量', required=False)
    image = serializers.ImageField(label='图片', required=False)
```

测试：

```
from booktest.serializers import BookInfoSerializer
data = {'btitle': 'python'}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()  # False
serializer.errors
#  {'btitle': [ErrorDetail(string='图书不是关于Django的', code='invalid')]}
```

2. 保存

如果在验证成功后，想要基于validated_data完成数据对象的创建，可以通过实现create()和update()两个方法来实现。

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

    def create(self, validated_data):
        """新建"""
        return BookInfo(**validated_data)

    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.bread = validated_data.get('bread', instance.bread)
        instance.bcomment = validated_data.get('bcomment', instance.bcomment)
        return instance
```

如果需要在返回数据对象的时候，也将数据保存到数据库中，则可以进行如下修改

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

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

    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.bread = validated_data.get('bread', instance.bread)
        instance.bcomment = validated_data.get('bcomment', instance.bcomment)
        instance.save()
        return instance
```

> 实现了上述两个方法后，在反序列化数据的时候，就可以通过save()方法返回一个数据对象实例了

```python
book = serializer.save()
```

> 如果创建序列化器对象的时候，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。

```python
from db.serializers import BookInfoSerializer
data = {'btitle': '封神演义', 'bpub_date': '1998-1-1'}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()  # True
serializer.save()  # <BookInfo: 封神演义>

from db.models import BookInfo
book = BookInfo.objects.get(id=2)
data = {'btitle': '倚天剑', 'bpub_date': '1998-12-18'}
serializer = BookInfoSerializer(book, data=data)
serializer.is_valid()  # True
serializer.save()  # <BookInfo: 倚天剑>
book.btitle  # '倚天剑'
```

两点说明：

> 1）在对序列化器进行save()保存时，可以额外传递数据，这些数据可以在create()和update()中的validated_data参数获取到

```python
serializer.save(owner=request.user)
```

> 2）默认序列化器必须传递所有required为True的字段，否则会抛出验证异常。但是我们可以使用partial参数来允许部分字段在更新操作时可以不传,而使用原有的数据

```python
serializer = BookInfoSerializer(instance=book, data={'btitle': 'hello django', 'bpub_date': book.bpub_date})
# 两种写法最终效果等价
serializer = BookInfoSerializer(instance=book, data={'btitle': 'hello django'}, partial=True)
```

```
In [1]: from booktest.serializers import BookInfoSerializer, HeroInfoSerializer

In [2]:

In [2]: data = {
   ...:     'btitle': '三国',
   ...:     'bpub_date': '1991-11-11'
   ...: }

In [3]: serializer = BookInfoSerializer(data=data)

In [4]: serializer.is_valid(raise_exception=True)
Out[4]: True

In [5]: serializer.validated_data
Out[5]: OrderedDict([('btitle', '三国'), ('bpub_date', datetime.date(1991, 11, 11))])

In [6]: serializer.save()
Out[6]: <BookInfo: 三国>
```


反序列化: 拿到前端传入的数据—>序列化器的data—>调用序列化器的.is_valid()方法进行校验—>调用序列化器的.save()方法
1.获取前端传入的json字典数据
2.创建序列化器 给序列化器的data参数传参(要以关键字方式传递)
3.调用序列化器的.is_valid(raise_exception=True) 进行校验,如果校验出错会自动抛出错误信息
4.调用序列化器的.save()方法, 调用save时会判断当初始创建序列化器时是否传入的instance
5.如果传了instance 也传了data 那么调用save实际调用序列化器中的update方法反之就是调用序列化器中的create方法

反序列化最后 会自动帮你完成序列化
Serializer.data

3.4 模型类序列化器ModelSerializer

```
ModelSerializer它继承Serializer:
1.ModelSerializer可以根据模型自动生成序列化器中的字段
2. ModelSerializer它里面已经帮我们实现了create和update方法
{'password1': 'xxxx', 'password2': 'xxxx'}
def create(self, validated_data):
    validated_data.pop('password2')

    return BookInfo.objects.create(**validated_data)

Def update(self, instance, validated_data)
    instance. password1 = validated_data['ps1']
instance. password2 = validated_data['ps1']
Insstance.save()
```


> 如果我们想要使用序列化器对应的是Django的模型类，DRF为我们提供了ModelSerializer模型类序列化器来帮助我们快速创建一个Serializer类。

> ModelSerializer与常规的Serializer相同，但提供了：

- 基于模型类自动生成一系列字段
- 包含默认的create()和update()的实现

1. 定义

> 比如我们创建一个BookInfoSerializer

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

- model 指明参照哪个模型类
- fields 指明为模型类的哪些字段生成

> 我们可以在python manage.py shell中查看自动生成的BookInfoSerializer的具体实现

```
>>> from booktest.serializers import BookInfoSerializer
>>> serializer = BookInfoSerializer()
>>> serializer
BookInfoSerializer():
    id = IntegerField(label='ID', read_only=True)
    btitle = CharField(label='名称', max_length=20)
    bpub_date = DateField(allow_null=True, label='发布日期', required=False)
    bread = IntegerField(label='阅读量', max_value=2147483647, min_value=-2147483648, required=False)
    bcomment = IntegerField(label='评论量', max_value=2147483647, min_value=-2147483648, required=False)
    image = ImageField(allow_null=True, label='图片', max_length=100, required=False)
```

2. 指定字段

1. 使用fields来明确字段，__all__表名包含所有字段，也可以写明具体哪些字段，如

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

2. 使用exclude可以明确排除掉哪些字段

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

3. 显示指明字段，如：

```
class HeroInfoSerializer(serializers.ModelSerializer):
    hbook = BookInfoSerializer()

    class Meta:
        model = HeroInfo
        fields = ('id', 'hname', 'hgender', 'hcomment', 'hbook')
```

4. 指明只读字段

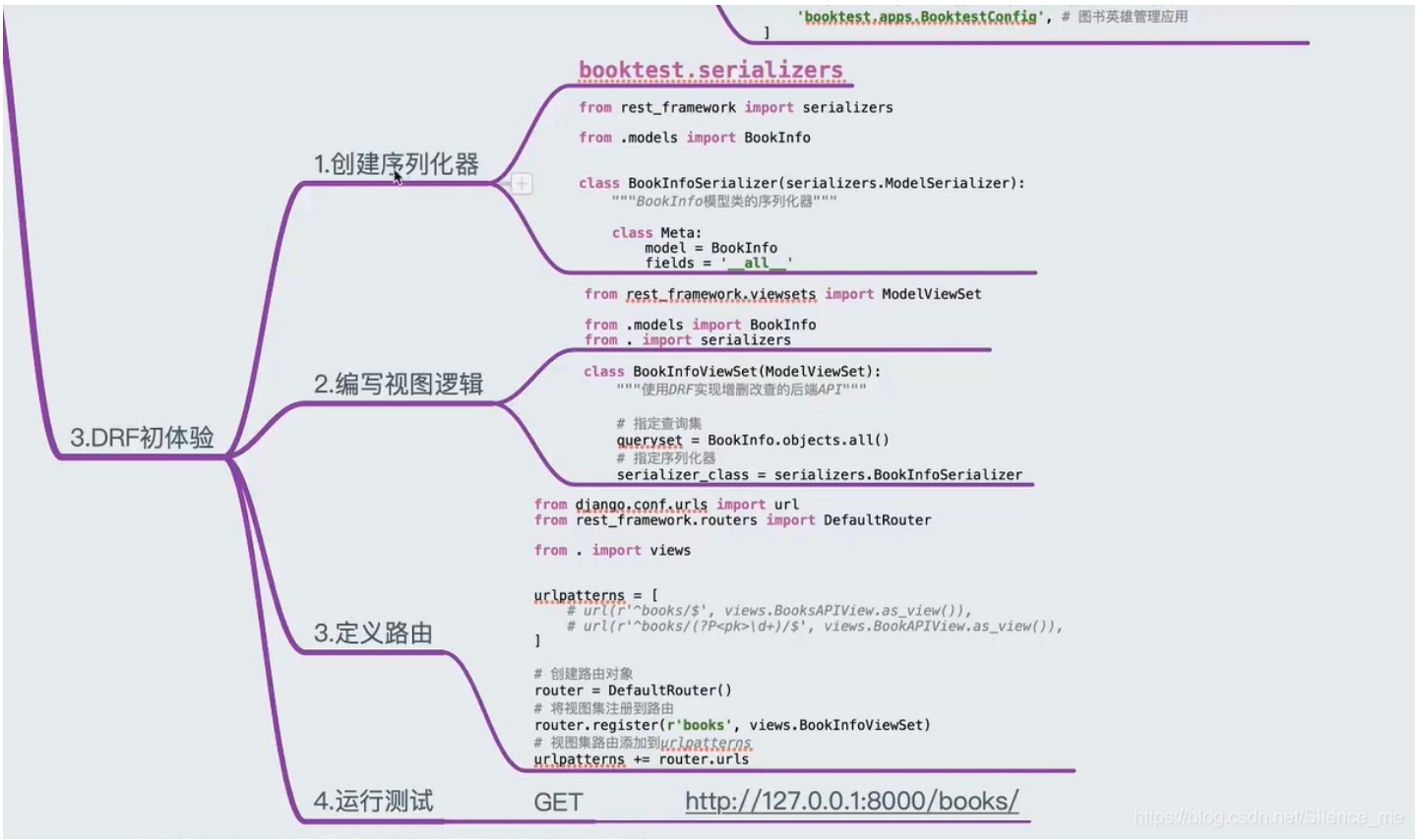
可以通过read_only_fields指明只读字段，即仅用于序列化输出的字段

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

3. 添加额外参数

我们可以使用extra_kwargs参数为ModelSerializer添加或修改原有的选项参数

```python
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},
        }

# BookInfoSerializer():
#     id = IntegerField(label='ID', read_only=True)
#     btitle = CharField(label='名称', max_length=20)
#     bpub_date = DateField(allow_null=True, label='发布日期', required=False)
#     bread = IntegerField(label='阅读量', max_value=2147483647, min_value=0, required=True)
#     bcomment = IntegerField(label='评论量', max_value=2147483647, min_value=0, required=True)
```



2.反序列化

1.概念

将其他格式（字典、JSON、XML等）转换为程序中的数据，例如将JSON字符串转换为Django中的模型类对象

2.反序列化行为

```
# 读取客户端传入的JSON数据
json_bytes = request.body
json_str = json_bytes.decode()
book_dict = json.loads(json_str)

# 创建新的记录
book = BookInfo.objects.create(
    btitle=book_dict.get('btitle'),
    bpub_date=datetime.strptime(book_dict.get('bpub_date'), '%Y-%m-%d').date(),
    bread = book_dict['bread'],
    bcomment = book_dict['bcomment']
)
```

3.反序列化时机

当需要将用户发送的数据存储到数据库之前，需要使用反序列化

1.序列化器的作用

1. 对数据进行转换　　序列化(输出、read_only)和反序列化(输入、write_onl
2. 进行数据的校验　　判断用户发送的数据是否合法　　is_valid(raise_

2.定义序列化器说明

1.模型类

```
#定义图书模型类BookInfo
class BookInfo(models.Model):
    btitle = models.CharField(max_length=20, verbose_name='名称')
    bpub_date = models.DateField(verbose_name='发布日期')
    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='逻辑删
    image = models.ImageField(upload_to='book', verbose_name='图书图片',
```

2.序列化器

```
class BookInfoSerializer(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, val
    bpub_date = serializers.DateField(label='发布日期', required=Tru
    bread = serializers.IntegerField(label='阅读量', required=False)
    bcomment = serializers.IntegerField(label='评论量', required=Fal
    image = serializers.ImageField(label='图片', required=False)
    heroinfo_set = serializers.PrimaryKeyRelatedField(read_only=Tru
```

```
heroinfo_set = serializers.PrimaryKeyRelatedField(read_only=True, many=True)
```

3.定义序列化器字段类型和选项

1.字段类型

类似于模型字段类型

2.选项参数

选项参数：

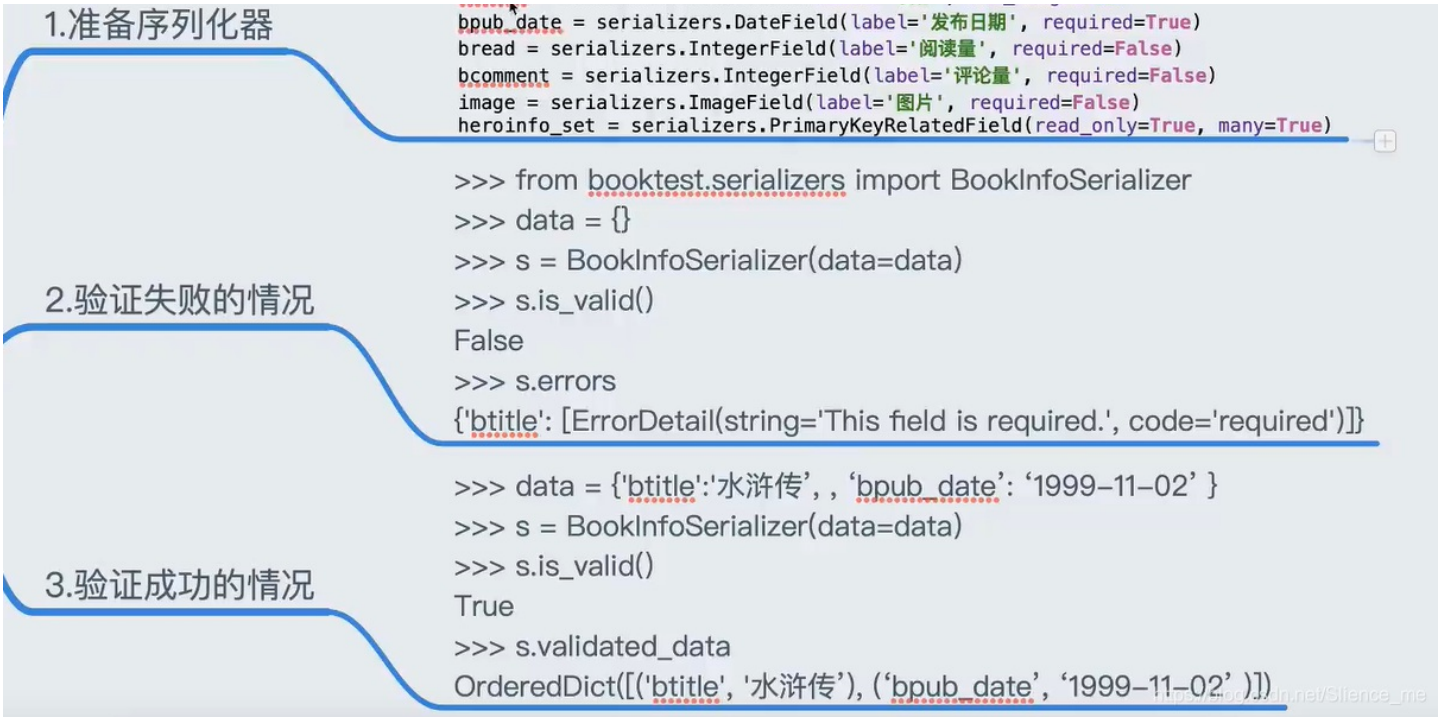
参数名称	作用
max_length	最大长度
min_lenght	最小长度
allow_blank	是否允许为空
trim_whitespace	是否截断空白字符
max_value	最小值
min_value	最大值

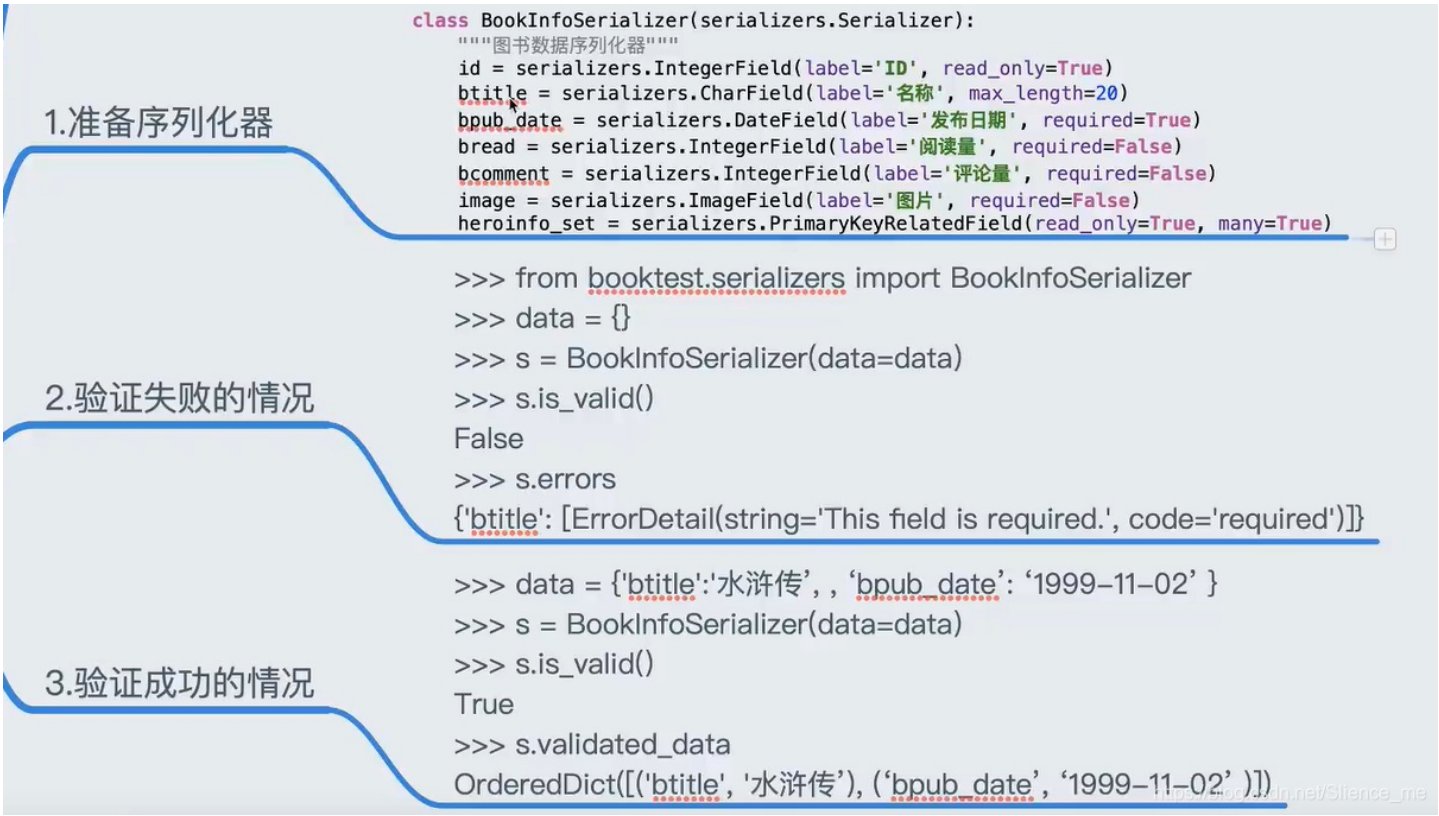
3.通用选项、参数

通用参数：

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	反序列化时使用的默认值
allow_null	表明该字段是否允许传入None，默认False
validators	该字段使用的验证器
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
help_text	用于HTML展示API页面时，显示的字段帮助提示信息


```
class BookInfoSerializer(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)
```





4. 视图

4.1 Request 与 Response

1. Request

> REST framework 传入视图的request对象不再是Django默认的HttpRequest对象，而是REST framework提供的扩展了HttpRequest类的Request类的对象。

> REST framework提供了Parser解析器，在接收到请求后会自动根据Content-Type指明的请求数据类型（如JSON、表单等）将请求数据进行parse解析，解析为类字典对象保存到Request对象中。

> Request对象的数据是自动根据前端发送数据的格式进行解析之后的结果。

> 无论前端发送的哪种格式的数据，我们都可以以统一的方式读取数据。

常用属性

1）.data

> request.data 返回解析之后的请求体数据。类似于Django中标准的request.POST和 request.FILES属性，但提供如下特性：
>
> - 包含了解析之后的文件和非文件数据
> - 包含了对POST、PUT、PATCH请求方式解析后的数据
> - 利用了REST framework的parsers解析器，不仅支持表单类型数据，也支持JSON数据

2）.query_params

> request.query_params与Django标准的request.GET相同，只是更换了更正确的名称而已。

2. Response

> rest_framework.response.Response

> REST framework提供了一个响应类Response，使用该类构造响应对象时，响应的具体数据内容会被转换（render渲染）成符合前端需求的类型。

> REST framework提供了Renderer渲染器，用来根据请求头中的Accept（接收数据类型声明）来自动转换响应数据到对应格式。如果前端请求中未进行Accept声明，则会采用默认方式处理响应数据，我们可以通过配置来修改默认响应格式。

```
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': (  # 默认响应渲染类
        'rest_framework.renderers.JSONRenderer',  # json渲染器
        'rest_framework.renderers.BrowsableAPIRenderer',  # 浏览API渲染器
    )
}
```

构造方式

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

> data数据不要是render处理之后的数据，只需传递python的内建类型数据即可，REST framework会使用renderer渲染器处理data。

> data不能是复杂结构的数据，如Django的模型类对象，对于这样的数据我们可以使用Serializer序列化器序列化处理后（转为了Python字典类型）再传递给data参数。
> 参数说明：
>
> - data: 为响应准备的序列化处理后的数据；
> - status: 状态码，默认200；
> - template_name: 模板名称，如果使用HTMLRenderer 时需指明；
> - headers: 用于存放响应头信息的字典；
> - content_type: 响应数据的Content-Type，通常此参数无需传递，REST framework会根据前端所需类型数据来设置该参数。

常用属性：

1）.data

> 传给response对象的序列化后，但尚未render处理的数据

2）.status_code

> 状态码的数字

3）.content

> 经过render处理后的响应数据

3. 状态码

> 为了方便设置状态码，REST framewrok在rest_framework.status模块中提供了常用状态码常量。

1）信息告知 - 1xx

```
HTTP_100_CONTINUE
HTTP_101_SWITCHING_PROTOCOLS
```

2）成功 - 2xx

```
HTTP_200_OK
HTTP_201_CREATED
HTTP_202_ACCEPTED
HTTP_203_NON_AUTHORITATIVE_INFORMATION
HTTP_204_NO_CONTENT
HTTP_205_RESET_CONTENT
HTTP_206_PARTIAL_CONTENT
HTTP_207_MULTI_STATUS
```

3）重定向 - 3xx

```
HTTP_300_MULTIPLE_CHOICES
HTTP_301_MOVED_PERMANENTLY
HTTP_302_FOUND
HTTP_303_SEE_OTHER
HTTP_304_NOT_MODIFIED
HTTP_305_USE_PROXY
HTTP_306_RESERVED
HTTP_307_TEMPORARY_REDIRECT
```

4）客户端错误 - 4xx

```
HTTP_400_BAD_REQUEST
HTTP_401_UNAUTHORIZED
HTTP_402_PAYMENT_REQUIRED
HTTP_403_FORBIDDEN
HTTP_404_NOT_FOUND
HTTP_405_METHOD_NOT_ALLOWED
HTTP_406_NOT_ACCEPTABLE
HTTP_407_PROXY_AUTHENTICATION_REQUIRED
HTTP_408_REQUEST_TIMEOUT
HTTP_409_CONFLICT
HTTP_410_GONE
HTTP_411_LENGTH_REQUIRED
HTTP_412_PRECONDITION_FAILED
HTTP_413_REQUEST_ENTITY_TOO_LARGE
HTTP_414_REQUEST_URI_TOO_LONG
HTTP_415_UNSUPPORTED_MEDIA_TYPE
HTTP_416_REQUESTED_RANGE_NOT_SATISFIABLE
HTTP_417_EXPECTATION_FAILED
HTTP_422_UNPROCESSABLE_ENTITY
HTTP_423_LOCKED
HTTP_424_FAILED_DEPENDENCY
HTTP_428_PRECONDITION_REQUIRED
HTTP_429_TOO_MANY_REQUESTS
HTTP_431_REQUEST_HEADER_FIELDS_TOO_LARGE
HTTP_451_UNAVAILABLE_FOR_LEGAL_REASONS
```

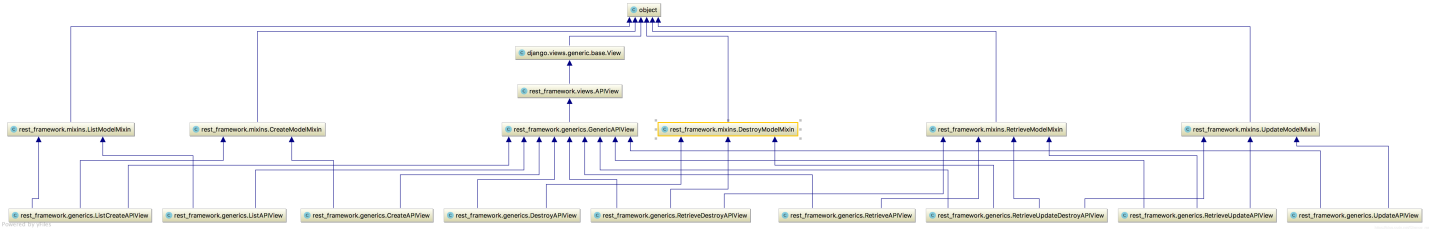
5）服务器错误 - 5xx

```
HTTP_500_INTERNAL_SERVER_ERROR
HTTP_501_NOT_IMPLEMENTED
HTTP_502_BAD_GATEWAY
HTTP_503_SERVICE_UNAVAILABLE
HTTP_504_GATEWAY_TIMEOUT
HTTP_505_HTTP_VERSION_NOT_SUPPORTED
HTTP_507_INSUFFICIENT_STORAGE
HTTP_511_NETWORK_AUTHENTICATION_REQUIRED
```

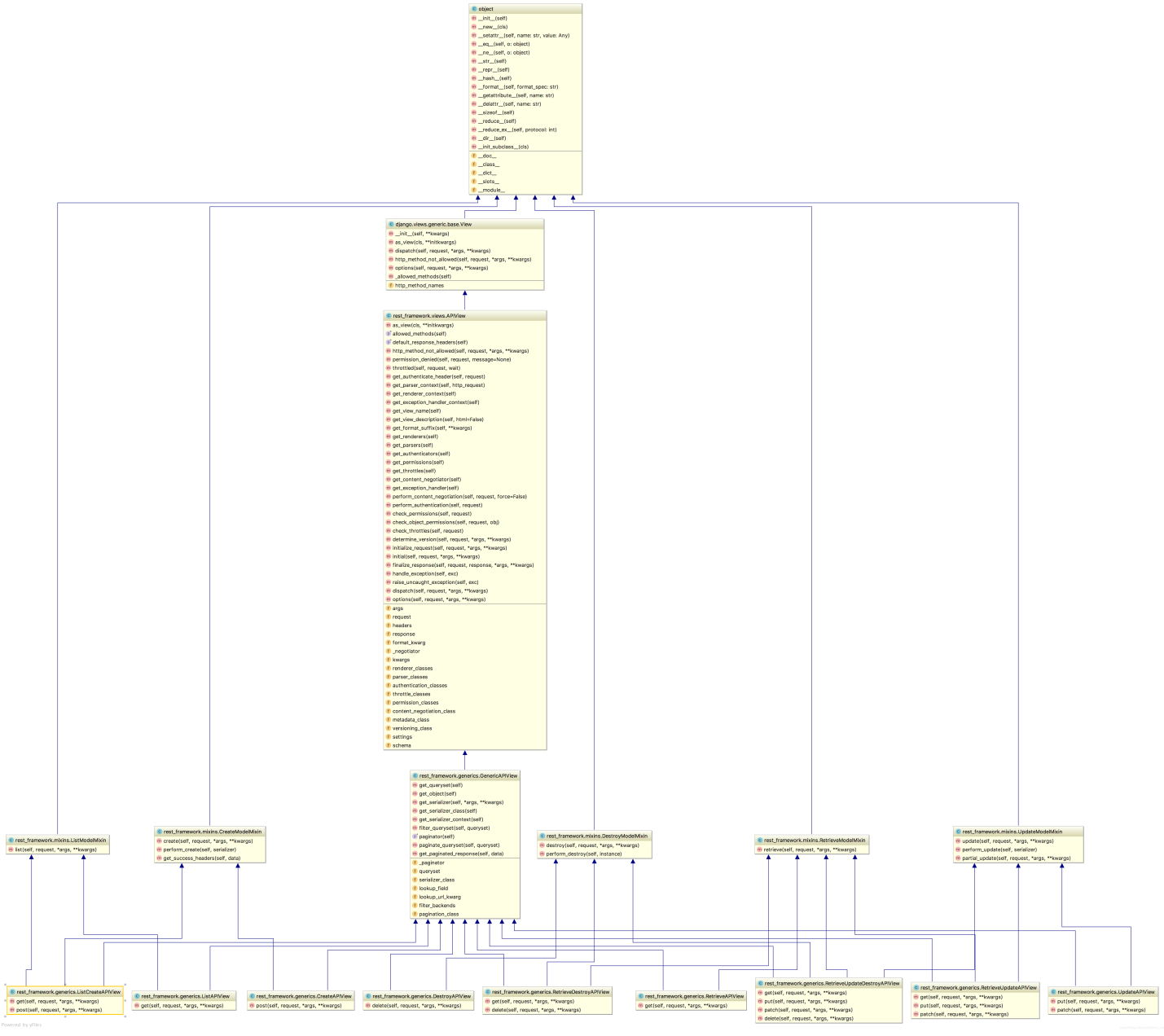
4.2 视图概览

REST framework 提供了众多的通用视图基类与扩展类，以简化视图的编写。

视图的继承关系：



视图的方法与属性：



4.3 视图说明

1. 两个基类

1）APIView

> rest_framework.views.APIView

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

> APIView与View的不同之处在于：
>
> - 传入到视图方法中的是REST framework的Request对象，而不是Django的HttpRequeset对象；
> - 视图方法可以返回REST framework的Response对象，视图会为响应数据设置（render）符合前端要求的格式；
> - 任何APIException异常都会被捕获到，并且处理成合适的响应信息；
> - 在进行dispatch()分发前，会对请求进行身份认证、权限检查、流量控制。

支持定义的属性：

> - authentication_classes 列表或元组，身份认证类
> - permissoin_classes 列表或元组，权限检查类
> - throttle_classes 列表或元组，流量控制类
>
> 在APIView中仍以常规的类视图定义方法来实现get() 、post() 或者其他请求方式的方法。

加粗样式举例：

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

# url(r'^books/$', views.BookListView.as_view()),
class BookListView(APIView):
    def get(self, request):
        books = BookInfo.objects.all()
        serializer = BookInfoSerializer(books, many=True)
        return Response(serializer.data)
```

2）GenericAPIView

> rest_framework.generics.GenericAPIView

> 继承自APIVlew，主要增加了操作序列化器和数据库查询的方法，作用是为下面Mixin扩展类的执行提供方法支持。通常在使用时，可搭配一个或多个Mixin扩展类。

> **提供的关于序列化器使用的属性与方法**
>
> 属性：
>
> - serializer_class 指明视图使用的序列化器
>
> 方法：
>
> - get_serializer_class(self) 返回序列化器类，默认返回serializer_class，可以重写，例如：

```
def get_serializer_class(self):
    if self.request.user.is_staff:
        return FullAccountSerializer
    return BasicAccountSerializer
```

get_serializer(self, args, *kwargs)
返回序列化器对象，主要用来提供给Mixin扩展类使用，如果我们在视图中想要获取序列化器对象，也可以直接调用此方法。

注意，该方法在提供序列化器对象的时候，会向序列化器对象的context属性补充三个数据：request、format、view，这三个数据对象可以在定义序列化器时使用。

request 当前视图的请求对象

view 当前请求的类视图对象

format 当前请求期望返回的数据格式

提供的关于数据库查询的属性与方法

属性：

- queryset 指明使用的数据查询集

方法：

- get_queryset(self)
- 返回视图使用的查询集，主要用来提供给Mixin扩展类使用，是列表视图与详情视图获取数据的基础，默认返回queryset属性，可以重写，例如：

```
def get_queryset(self):
    user = self.request.user
    return user.accounts.all()
```

- get_object(self)

  返回详情视图所需的模型类数据对象，主要用来提供给Mixin扩展类使用。
  在试图中可以调用该方法获取详情信息的模型类对象。

  若详情访问的模型类对象不存在，会返回404。

  该方法会默认使用APIView提供的check_object_permissions方法检查当前对象是否有权限被访问。

举例：

```
# url(r'^books/(?P<pk>\d+)/$', views.BookDetailView.as_view()),
class BookDetailView(GenericAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer

    def get(self, request, pk):
        book = self.get_object() # get_object()方法根据pk参数查找queryset中的数据对象
        serializer = self.get_serializer(book)
        return Response(serializer.data)
```

其他可以设置的属性

- pagination_class 指明分页控制类
- filter_backends 指明过滤控制后端

2. 五个扩展类

> **作用：** 提供了几种后端视图（对数据资源进行曾删改查）处理流程的实现，如果需要编写的视图属于这五种，则视图可以通过继承相应的扩展类来复用代码，减少自己编写的代码量。

> 这五个扩展类需要搭配GenericAPIView父类，因为五个扩展类的实现需要调用GenericAPIView提供的序列化器与数据库查询的方法。

1）ListModelMixin

> 列表视图扩展类，提供list(request, *args, **kwargs)方法快速实现列表视图，返回200状态码。

> 该Mixin的list方法会对数据进行过滤和分页。

源代码：

```
class ListModelMixin(object):
    """
    List a queryset.
    """
    def list(self, request, *args, **kwargs):
        # 过滤
        queryset = self.filter_queryset(self.get_queryset())
        # 分页
        page = self.paginate_queryset(queryset)
        if page is not None:
            serializer = self.get_serializer(page, many=True)
            return self.get_paginated_response(serializer.data)
        # 序列化
        serializer = self.get_serializer(queryset, many=True)
        return Response(serializer.data)
```

举例：

```
from rest_framework.mixins import ListModelMixin

class BookListView(ListModelMixin, GenericAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer

    def get(self, request):
        return self.list(request)
```

2）CreateModelMixin

> 创建视图扩展类，提供create(request, *args, **kwargs)方法快速实现创建资源的视图，成功返回201状态码。

> 如果序列化器对前端发送的数据验证失败，返回400错误。

源代码：

```python
class CreateModelMixin(object):
    """
    Create a model instance.
    """
    def create(self, request, *args, **kwargs):
        # 获取序列化器
        serializer = self.get_serializer(data=request.data)
        # 验证
        serializer.is_valid(raise_exception=True)
        # 保存
        self.perform_create(serializer)
        headers = self.get_success_headers(serializer.data)
        return Response(serializer.data, status=status.HTTP_201_CREATED, headers=headers)

    def perform_create(self, serializer):
        serializer.save()

    def get_success_headers(self, data):
        try:
            return {'Location': str(data[api_settings.URL_FIELD_NAME])}
        except (TypeError, KeyError):
            return {}
```

3） RetrieveModelMixin

> 详情视图扩展类，提供retrieve(request, *args, **kwargs)方法，可以快速实现返回一个存在的数据对象。

> 如果存在，返回200， 否则返回404。

源代码：

```python
class RetrieveModelMixin(object):
    """
    Retrieve a model instance.
    """
    def retrieve(self, request, *args, **kwargs):
        # 获取对象，会检查对象的权限
        instance = self.get_object()
        # 序列化
        serializer = self.get_serializer(instance)
        return Response(serializer.data)
```

举例：

```python
class BookDetailView(RetrieveModelMixin, GenericAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer

    def get(self, request, pk):
        return self.retrieve(request)
```

4）UpdateModelMixin

> 更新视图扩展类，提供update(request, *args, **kwargs)方法，可以快速实现更新一个存在的数据对象。

> 同时也提供partial_update(request, *args, **kwargs)方法，可以实现局部更新。

> 成功返回200，序列化器校验数据失败时，返回400错误。

源代码：

```
class UpdateModelMixin(object):
    """
    Update a model instance.
    """
    def update(self, request, *args, **kwargs):
        partial = kwargs.pop('partial', False)
        instance = self.get_object()
        serializer = self.get_serializer(instance, data=request.data, partial=partial)
        serializer.is_valid(raise_exception=True)
        self.perform_update(serializer)

        if getattr(instance, '_prefetched_objects_cache', None):
            # If 'prefetch_related' has been applied to a queryset, we need to
            # forcibly invalidate the prefetch cache on the instance.
            instance._prefetched_objects_cache = {}

        return Response(serializer.data)

    def perform_update(self, serializer):
        serializer.save()

    def partial_update(self, request, *args, **kwargs):
        kwargs['partial'] = True
        return self.update(request, *args, **kwargs)
```

5）DestroyModelMixin

> 删除视图扩展类，提供destroy(request, *args, **kwargs)方法，可以快速实现删除一个存在的数据对象。

> 成功返回204，不存在返回404。

源代码：

```
class DestroyModelMixin(object):
    """
    Destroy a model instance.
    """
    def destroy(self, request, *args, **kwargs):
        instance = self.get_object()
        self.perform_destroy(instance)
        return Response(status=status.HTTP_204_NO_CONTENT)

    def perform_destroy(self, instance):
        instance.delete()
```

3. 几个可用子类视图

1） CreateAPIView

> 提供 post 方法
> 继承自： GenericAPIView、CreateModelMixin

2）ListAPIView

> 提供 get 方法
> 继承自：GenericAPIView、ListModelMixin

3）RetrieveAPIView

> 提供 get 方法
> 继承自: GenericAPIView、RetrieveModelMixin

4）DestoryAPIView

> 提供 delete 方法
> 继承自：GenericAPIView、DestoryModelMixin

5）UpdateAPIView

> 提供 put 和 patch 方法
> 继承自：GenericAPIView、UpdateModelMixin

6）RetrieveUpdateAPIView

> 提供 get、put、patch方法
> 继承自： GenericAPIView、RetrieveModelMixin、UpdateModelMixin

7）RetrieveUpdateDestoryAPIView

> 提供 get、put、patch、delete方法
> 继承自：GenericAPIView、RetrieveModelMixin、UpdateModelMixin、DestoryModelMixin

4.4 视图集ViewSet

> 使用视图集ViewSet，可以将一系列逻辑相关的动作放到一个类中：
>
> - list() 提供一组数据
> - retrieve() 提供单个数据
> - create() 创建数据
> - update() 保存数据
> - destory() 删除数据

> ViewSet视图集类不再实现get()、post()等方法，而是实现动作 action 如 list() 、create() 等。

> 视图集只在使用as_view()方法的时候，才会将action动作与具体请求方式对应上。如：

```
class BookInfoViewSet(viewsets.ViewSet):

    def list(self, request):
        books = BookInfo.objects.all()
        serializer = BookInfoSerializer(books, many=True)
        return Response(serializer.data)

    def retrieve(self, request, pk=None):
        try:
            books = BookInfo.objects.get(id=pk)
        except BookInfo.DoesNotExist:
            return Response(status=status.HTTP_404_NOT_FOUND)
        serializer = BookInfoSerializer(books)
        return Response(serializer.data)
```

> 在设置路由时，我们可以如下操作

```
urlpatterns = [
    url(r'^books/$', BookInfoViewSet.as_view({'get':'list'}),
    url(r'^books/(?P<pk>\d+)/$', BookInfoViewSet.as_view({'get': 'retrieve'})
]
```

1. 常用视图集父类

1） ViewSet

> 继承自APIView与ViewSetMixin，作用也与APIView基本类似，提供了身份认证、权限校验、流量管理等。

> ViewSet主要通过继承ViewSetMixin来实现在调用as_view()时传入字典（如{‘get’:‘list’}）的映射处理工作。

> 在ViewSet中，没有提供任何动作action方法，需要我们自己实现action方法。

2）GenericViewSet

> 使用ViewSet通常并不方便，因为list、retrieve、create、update、destory等方法都需要自己编写，而这些方法与前面讲过的Mixin扩展类提供的方法同名，所以我们可以通过继承Mixin扩展类来复用这些方法而无需自己编写。但是Mixin扩展类依赖与GenericAPIView，所以还需要继承GenericAPIView。

> GenericViewSet就帮助我们完成了这样的继承工作，继承自GenericAPIView与ViewSetMixin，在实现了调用as_view()时传入字典（如{‘get’:‘list’}）的映射处理工作的同时，还提供了GenericAPIView提供的基础方法，可以直接搭配Mixin扩展类使用。

举例：

```
from rest_framework import mixins
from rest_framework.viewsets import GenericViewSet
from rest_framework.decorators import action

class BookInfoViewSet(mixins.ListModelMixin, mixins.RetrieveModelMixin, GenericViewSet):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
```

url的定义

```
urlpatterns = [
    url(r'^books/$', views.BookInfoViewSet.as_view({'get': 'list'})),
    url(r'^books/(?P<pk>\d+)/$', views.BookInfoViewSet.as_view({'get': 'retrieve'})),
]
```

3）ModelViewSet

> 继承自GenericViewSet，同时包括了ListModelMixin、RetrieveModelMixin、CreateModelMixin、UpdateModelMixin、DestoryModelMixin。

4）ReadOnlyModelViewSet

> 继承自GenericViewSet，同时包括了ListModelMixin、RetrieveModelMixin。

2. 视图集中定义附加action动作

> 在视图集中，除了上述默认的方法动作外，还可以添加自定义动作。

举例：

```
from rest_framework import mixins
from rest_framework.viewsets import GenericViewSet
from rest_framework.decorators import action

class BookInfoViewSet(mixins.ListModelMixin, mixins.RetrieveModelMixin, GenericViewSet):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer

    def latest(self, request):
        """
        返回最新的图书信息
        """
        book = BookInfo.objects.latest('id')
        serializer = self.get_serializer(book)
        return Response(serializer.data)

    def read(self, request, pk):
        """
        修改图书的阅读量数据
        """
        book = self.get_object()
        book.bread = request.data.get('bread')
        book.save()
        serializer = self.get_serializer(book)
        return Response(serializer.data)
```

url的定义

```
urlpatterns = [
    url(r'^books/$', views.BookInfoViewSet.as_view({'get': 'list'})),
    url(r'^books/latest/$', views.BookInfoViewSet.as_view({'get': 'latest'})),
    url(r'^books/(?P<pk>\d+)/$', views.BookInfoViewSet.as_view({'get': 'retrieve'})),
    url(r'^books/(?P<pk>\d+)/read/$', views.BookInfoViewSet.as_view({'put': 'read'})),
]
```

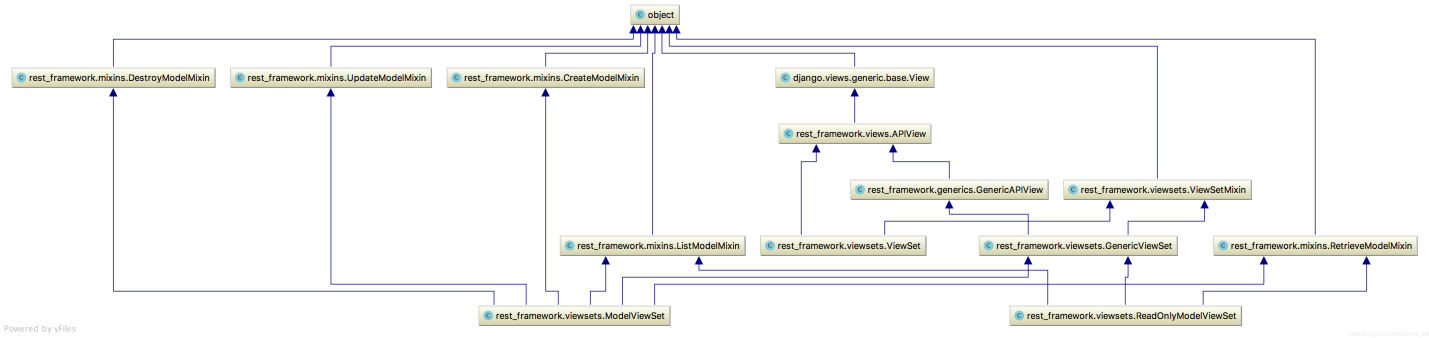
3. action属性

> 在视图集中，我们可以通过action对象属性来获取当前请求视图集时的action动作是哪个。

例如：

```
def get_serializer_class(self):
    if self.action == 'create':
        return OrderCommitSerializer
    else:
        return OrderDataSerializer
```

4. 视图集的继承关系



4.5 路由Routers

> 对于视图集ViewSet，我们除了可以自己手动指明请求方式与动作action之间的对应关系外，还可以使用Routers来帮助我们快速实现路由信息。

> REST framework提供了两个router
>
> - SimpleRouter
> - DefaultRouter

1. 使用方法

> 1） 创建router对象，并注册视图集，例如

```
from rest_framework import routers

router = routers.SimpleRouter()
router.register(r'books', BookInfoViewSet, base_name='book')
```

> register(prefix, viewset, base_name)
>
> - prefix 该视图集的路由前缀
> - viewset 视图集
> - base_name 路由名称的前缀

> 如上述代码会形成的路由如下：

```
^books/$    name: book-list
^books/{pk}/$   name: book-detail
```

> 2）添加路由数据

> 可以有两种方式：

```
urlpatterns = [
    ...
]
urlpatterns += router.urls
```

或

```
urlpatterns = [
    ...
    url(r'^', include(router.urls))
]
```

2. 视图集中附加action的声明

> 在视图集中，如果想要让Router自动帮助我们为自定义的动作生成路由信息，需要使用rest_framework.decorators.action装饰器。

> 以action装饰器装饰的方法名会作为action动作名，与list、retrieve等同。

> action装饰器可以接收两个参数：
>
> - methods: 声明该action对应的请求方式，列表传递
> - detail: 声明该action的路径是否与单一资源对应，及是否是xxx//action方法名/
>   - True 表示路径格式是xxx//action方法名/
>   - False 表示路径格式是xxx/action方法名/

举例：

```python
from rest_framework import mixins
from rest_framework.viewsets import GenericViewSet
from rest_framework.decorators import action

class BookInfoViewSet(mixins.ListModelMixin, mixins.RetrieveModelMixin, GenericViewSet):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer

    # detail为False 表示路径名格式应该为 books/latest/
    @action(methods=['get'], detail=False)
    def latest(self, request):
        """
        返回最新的图书信息
        """
        ...

    # detail为True, 表示路径名格式应该为 books/{pk}/read/
    @action(methods=['put'], detail=True)
    def read(self, request, pk):
        """
        修改图书的阅读量数据
        """
        ...
```

> 由路由器自动为此视图集自定义action方法形成的路由会是如下内容：

```
^books/latest/$    name: book-latest
^books/{pk}/read/$  name: book-read
```

3. 路由router形成URL的方式

1） SimpleRouter

URL Style	HTTP Method	Action	URL Name
{prefix}/	GET	list	{basename}-list
	POST	create	
{prefix}/{url_path}/	GET, or as specified by `methods` argument	`@action(detail=False)` decorated method	{basename}-{url_name}
{prefix}/{lookup}/	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{url_path}/	GET, or as specified by `methods` argument	`@action(detail=True)` decorated method	{basename}-{url_name}

2）DefaultRouter

URL Style	HTTP Method	Action	URL Name
[.format]	GET	automatically generated root view	api-root
{prefix}/[.format]	GET	list	{basename}-list
	POST	create	
{prefix}/{url_path}/[.format]	GET, or as specified by `methods` argument	`@action(detail=False)` decorated method	{basename}-{url_name}
{prefix}/{lookup}/[.format]	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{url_path}/[.format]	GET, or as specified by `methods` argument	`@action(detail=True)` decorated method	{basename}-{url_name}

> DefaultRouter与SimpleRouter的区别是，DefaultRouter会多附带一个默认的API根视图，返回一个包含所有列表视图的超链接响应数据。

5. 其他功能

5.1 认证

认证Authentication

> 可以在配置文件中配置全局默认的认证方案

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',   # 基本认证
        'rest_framework.authentication.SessionAuthentication',  # session认证
    )
}
```

> 也可以在每个视图中通过设置authentication_classess属性来设置

```
from rest_framework.authentication import SessionAuthentication, BasicAuthentication
from rest_framework.views import APIView

class ExampleView(APIView):
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    ...
```

> 认证失败会有两种可能的返回值：
>
> - 401 Unauthorized 未认证
> - 403 Permission Denied 权限被禁止

5.2 权限

权限Permissions

> 权限控制可以限制用户对于视图的访问和对于具体数据对象的访问。
>
> - 在执行视图的dispatch()方法前，会先进行视图访问权限的判断
> - 在通过get_object()获取具体对象时，会进行对象访问权限的判断

使用

> 可以在配置文件中设置默认的权限管理类，如

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    )
}
```

> 如果未指明，则采用如下默认配置

```
'DEFAULT_PERMISSION_CLASSES': (
   'rest_framework.permissions.AllowAny',
)
```

> 也可以在具体的视图中通过permission_classes属性来设置，如

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

class ExampleView(APIView):
    permission_classes = (IsAuthenticated,)
    ...
```

> ### 提供的权限
>
> - AllowAny 允许所有用户
> - IsAuthenticated 仅通过认证的用户
> - IsAdminUser 仅管理员用户
> - IsAuthenticatedOrReadOnly 认证的用户可以完全操作，否则只能get读取

举例

```
from rest_framework.authentication import SessionAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.generics import RetrieveAPIView

class BookDetailView(RetrieveAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticated]
```

自定义权限

> 如需自定义权限，需继承rest_framework.permissions.BasePermission父类，并实现以下两个任何一个方法或全部
>
> .has_permission(self, request, view) 是否可以访问视图， view表示当前视图对象
>
> .has_object_permission(self, request, view, obj) 是否可以访问数据对象， view表示当前视图， obj为数据对象

例如：

```
class MyPermission(BasePermission):
    def has_object_permission(self, request, view, obj):
        """控制对obj对象的访问权限，此案例决绝所有对对象的访问"""
        return False

class BookInfoViewSet(ModelViewSet):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
    permission_classes = [IsAuthenticated, MyPermission]
```

5.3 限流

限流Throttling

> 可以对接口访问的频次进行限制，以减轻服务器压力。

使用

> 可以在配置文件中，使用DEFAULT_THROTTLE_CLASSES 和 DEFAULT_THROTTLE_RATES进行全局配置，

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ),
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',
        'user': '1000/day'
    }
}
```

> DEFAULT_THROTTLE_RATES 可以使用 second, minute, hour 或day来指明周期。

> 也可以在具体视图中通过throttle_classess属性来配置，如

```
from rest_framework.throttling import UserRateThrottle
from rest_framework.views import APIView

class ExampleView(APIView):
    throttle_classes = (UserRateThrottle,)
    ...
```

可选限流类

> 1） AnonRateThrottle
> 限制所有匿名未认证用户，使用IP区分用户。
> 使用DEFAULT_THROTTLE_RATES['anon'] 来设置频次
> 2）UserRateThrottle
> 限制认证用户，使用User id 来区分。
> 使用DEFAULT_THROTTLE_RATES['user'] 来设置频次
> 3）ScopedRateThrottle
> 限制用户对于每个视图的访问频次，使用ip或user id。

例如：

```
class ContactListView(APIView):
    throttle_scope = 'contacts'
    ...

class ContactDetailView(APIView):
    throttle_scope = 'contacts'
    ...

class UploadView(APIView):
    throttle_scope = 'uploads'
    ...
```

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.ScopedRateThrottle',
    ),
    'DEFAULT_THROTTLE_RATES': {
        'contacts': '1000/day',
        'uploads': '20/day'
    }
}
```

实例

```
from rest_framework.authentication import SessionAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.generics import RetrieveAPIView
from rest_framework.throttling import UserRateThrottle

class BookDetailView(RetrieveAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticated]
    throttle_classes = (UserRateThrottle,)
```

5.4 过滤

过滤Filtering

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

```
pip install django-filter
```

> 在配置文件中增加过滤后端的设置：

```
INSTALLED_APPS = [
    ...
    'django_filters',  # 需要注册应用，
]

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

> 在视图中添加filter_fields属性，指定可以过滤的字段

```
class BookListView(ListAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
    filter_fields = ('btitle', 'bread')

# 127.0.0.1:8000/books/?btitle=西游记
```

5.5 排序

排序

> 对于列表数据，REST framework提供了OrderingFilter过滤器来帮助我们快速指明数据按照指定字段进行排序。

使用方法：

> 在类视图中设置filter_backends，使用rest_framework.filters.OrderingFilter过滤器，REST framework会在请求的查询字符串参数中检查是否包含了ordering参数，如果包含了ordering参数，则按照ordering参数指明的排序字段对数据集进行排序。

> 前端可以传递的ordering参数的可选字段值需要在ordering_fields中指明。

示例：

```
class BookListView(ListAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
    filter_backends = [OrderingFilter]
    ordering_fields = ('id', 'bread', 'bpub_date')

# 127.0.0.1:8000/books/?ordering=-bread
```

5.6 分页

分页Pagination

> REST framework提供了分页的支持。

我们可以在配置文件中设置全局的分页方式，如：

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':  'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 100  # 每页数目
}
```

也可通过自定义Pagination类，来为视图添加不同分页行为。在视图中通过pagination_clas属性来指明。

```
class LargeResultsSetPagination(PageNumberPagination):
    page_size = 1000
    page_size_query_param = 'page_size'
    max_page_size = 10000
```

```
class BookDetailView(RetrieveAPIView):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoSerializer
    pagination_class = LargeResultsSetPagination
```

注意：如果在视图内关闭分页功能，只需在视图内设置

```
pagination_class = None
```

可选分页器

1） PageNumberPagination

前端访问网址形式：
`GET http://api.example.org/books/?page=4`

可以在子类中定义的属性：

- page_size 每页数目
- page_query_param 前端发送的页数关键字名，默认为"page"
- page_size_query_param 前端发送的每页数目关键字名，默认为None
- max_page_size 前端最多能设置的每页数量

```
from rest_framework.pagination import PageNumberPagination

class StandardPageNumberPagination(PageNumberPagination):
    page_size_query_param = 'page_size'
    max_page_size = 10

class BookListView(ListAPIView):
    queryset = BookInfo.objects.all().order_by('id')
    serializer_class = BookInfoSerializer
    pagination_class = StandardPageNumberPagination

# 127.0.0.1/books/?page=1&page_size=2
```

2）LimitOffsetPagination

前端访问网址形式：
`GET http://api.example.org/books/?limit=100&offset=400`

可以在子类中定义的属性：

- default_limit 默认限制，默认值与PAGE_SIZE设置一直
- limit_query_param limit参数名，默认'limit'
- offset_query_param offset参数名，默认'offset'
- max_limit 最大limit限制，默认None

```
from rest_framework.pagination import LimitOffsetPaginati`from rest_framework.views import exception_handler

def custom_exception_handler(exc, context):
    # 先调用REST framework默认的异常处理方法获得标准错误响应对象
    response = exception_handler(exc, context)

    # 在此处补充自定义的异常处理
    if response is not None:
        response.data['status_code'] = response.status_code

    return response`on

class BookListView(ListAPIView):
    queryset = BookInfo.objects.all().order_by('id')
    serializer_class = BookInfoSerializer
    pagination_class = LimitOffsetPagination

# 127.0.0.1:8000/books/?offset=3&limit=2
```

5.7 异常处理

异常处理 Exceptions

REST framework提供了异常处理，我们可以自定义异常处理函数。

```
from rest_framework.views import exception_handler

def custom_exception_handler(exc, context):
    # 先调用REST framework默认的异常处理方法获得标准错误响应对象
    response = exception_handler(exc, context)

    # 在此处补充自定义的异常处理
    if response is not None:
        response.data['status_code'] = response.status_code

    return response
```

在配置文件中声明自定义的异常处理

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'my_project.my_app.utils.custom_exception_handler'
}
```

> 如果未声明，会采用默认的方式，如下

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'rest_framework.views.exception_handler'
}
```

> 例如：
> 补充上处理关于数据库的异常

```
from rest_framework.views import exception_handler as drf_exception_handler
from rest_framework import status
from django.db import DatabaseError

def exception_handler(exc, context):
    response = drf_exception_handler(exc, context)

    if response is None:
        view = context['view']
        if isinstance(exc, DatabaseError):
            print('[%s]: %s' % (view, exc))
            response = Response({'detail': '服务器内部错误'}, status=status.HTTP_507_INSUFFICIENT_STORAGE)

    return response
```

> REST framework提供了异常处理，我们可以自定义异常处理函数。
>
> ### REST framework定义的异常
>
> - APIException 所有异常的父类
> - ParseError 解析错误
> - AuthenticationFailed 认证失败
> - NotAuthenticated 尚未认证
> - PermissionDenied 权限决绝
> - NotFound 未找到
> - MethodNotAllowed 请求方式不支持
> - NotAcceptable 要获取的数据格式不支持
> - Throttled 超过限流次数
> - ValidationError 校验失败

5.8 自动生成接口文档

自动生成接口文档

> REST framework可以自动帮助我们生成接口文档。

> 接口文档以网页的方式呈现。

> 自动接口文档能生成的是继承自APIView及其子类的视图。

1. 安装依赖

> REST framewrok生成接口文档需要coreapi库的支持。

```
pip install coreapi
```

2. 设置接口文档访问路径

> 在总路由中添加接口文档路径。

> 文档路由对应的视图配置为
> rest_framework.documentation.include_docs_urls，

> 参数title为接口文档网站的标题。

```
from rest_framework.documentation import include_docs_urls

urlpatterns = [
    ...
    url(r'^docs/', include_docs_urls(title='My API title'))
]

# settings.py
'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema',
```

3. 文档描述说明的定义位置

> 1） 单一方法的视图，可直接使用类视图的文档字符串，如

```
class BookListView(generics.ListAPIView):
    """
    返回所有图书信息.
    """
```

> 2）包含多个方法的视图，在类视图的文档字符串中，分开方法定义，如

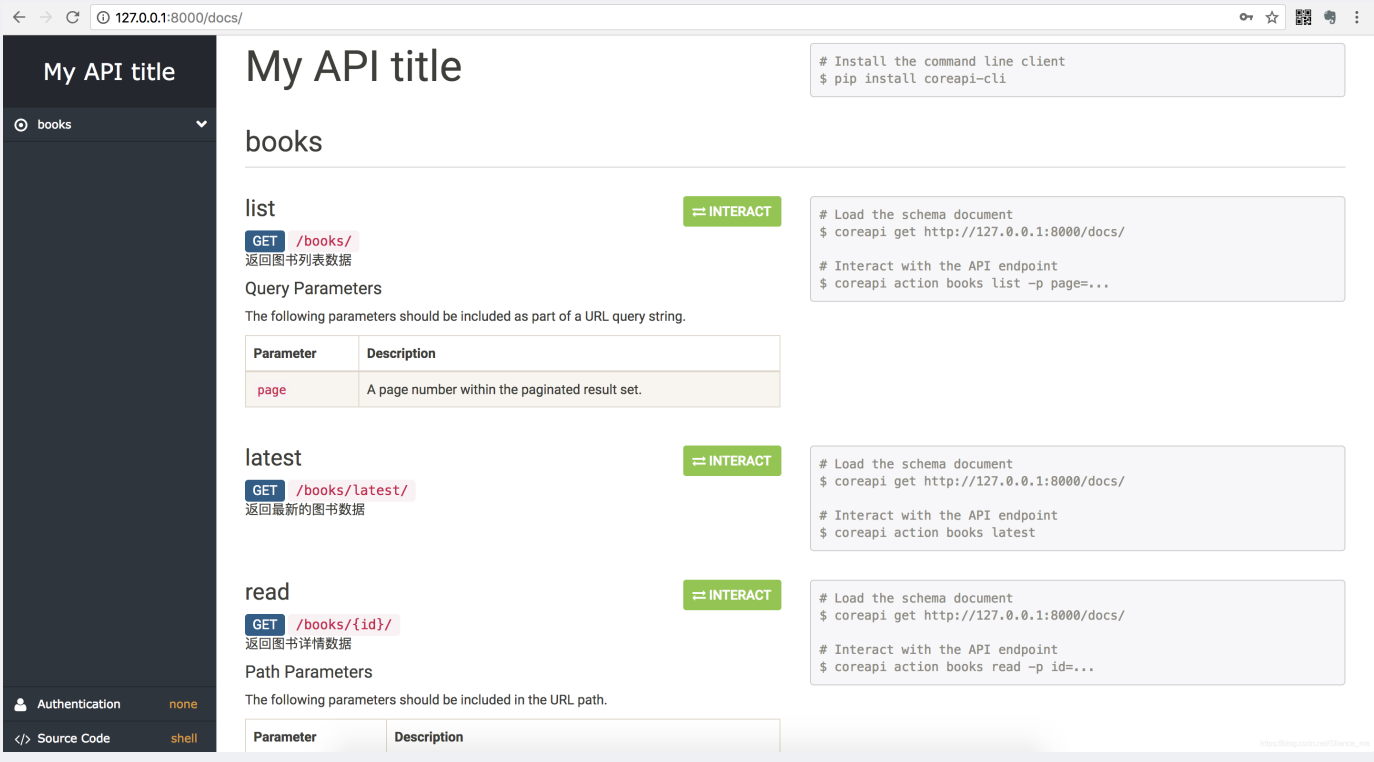
```
class BookListCreateView(generics.ListCreateAPIView):
    """
    get:
    返回所有图书信息.

    post:
    新建图书.
    """
```

3）对于视图集ViewSet，仍在类视图的文档字符串中封开定义，但是应使用action名称区分，如

```
class BookInfoViewSet(mixins.ListModelMixin, mixins.RetrieveModelMixin, GenericViewSet):
    """
    list:
    返回图书列表数据

    retrieve:
    返回图书详情数据

    latest:
    返回最新的图书数据

    read:
    修改图书的阅读量
    """
```

4. 访问接口文档网页

浏览器访问 127.0.0.1:8000/docs/，即可看到自动生成的接口文档。



> **两点说明：**
>
> 1） 视图集ViewSet中的retrieve名称，在接口文档网站中叫做read
> 2）参数的Description需要在模型类或序列化器类的字段中以help_text选项定义，如：

```
class BookInfo(models.Model):
    ...
    bread = models.IntegerField(default=0, verbose_name='阅读量', help_text='阅读量')
    ...
```

或

```
class BookReadSerializer(serializers.ModelSerializer):
    class Meta:
        model = BookInfo
        fields = ('bread', )
        extra_kwargs = {
            'bread': {
                'required': True,
                'help_text': '阅读量'
            }
        }
```