```
'''写一个序列化的类，继承Serializer
   在类中要序列化的字段
'''
   from rest_framework import serializers

   class BookSer(serializers.Serializer):
      title = serializers.CharField() # 括号内写校验需求参数
      price = serializers.CharField()
      publish = serializers.CharField()
'''urls查询一个book的数据'''
   url(r'^books/(?P<pk>\d+)', views.TestOne.as_view())

'''在视图中导入使用，实例化，再将序列化对象传入'''
   from rest_framework.views import APIView
   from app01.serializerFile import BookSer
   from rest_framework.response import Response #  drf提供的响应对象，二次封装
                                                # 其实和JsonResponse类似，差别很小
                                                # 对浏览器及进行了优化
   class TestOne(APIView):
      def get(self, request, pk):
         book = models.Book.objects.filter(pk=pk).first()
         # 实例化序列化器，要序列化谁，就传谁
         book_ser = BookSer(book)
         return Response(book_ser.data)
         '''
         发送请求返回 book_ser.data 结果是一个字典
         {
            "title": "西游记",
            "price": "12.00",
            "publish": "东方出版社"
         }
         '''
      '''校验钩子函数，全局钩子/局部钩子（类似于Forms组件）'''
      # 局部钩子
      def validate_price(self, data): # 会把验证字段的数据传入,类型为字段类型，主要注意
         # 添加校验，如果价格小于十，校验不通过
         if float(data) > 10:
            return data #将数据返回
         else:
            raise ValidationError('价格低了')

      # 全局钩子
      def validate(self, validated_data): # validated_data内部含有所有的数据
         #内部逻辑
         return validated_data


'''其他的校验规则（很少用，可以忘记）'''
   def check_price(data):
      # 内部逻辑
      return data

   class BookSer(serializers.Serializer):
      title = serializers.CharField()
      price = serializers.CharField(validators=[check_price])
      publish = serializers.CharField()
```

"""**********序列化字段类型**********
BooleanField  BooleanField()

NullBooleanField  NullBooleanField()  多了一个null

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(maxlength=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)

ListField  ListField(child=, min_length=None, max_length=None)

DictField  DictField(child=)

```
**********************************校验字段参数名称 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（适用于那种在数字领域的控制）
min_value               最大值（适用于那种在数字领域的控制）

**********************************通用参数名称 作用**********************************
read_only               表明该字段仅用于序列化输出，默认False
                          如果设置成True，postman中可以看到该字段，修改时，不需要传入该字段

write_only              表明该字段仅用于反序列化输入，默认False
                          如果设置为True，postman中看不到该字段，修改时，该字段需要传入
                            （有带你类似于添加required=True）

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
'''

**********************************数据增删改查**********************************
'''新增数据'''
#views.py
    def post(self, request):
        response_msg = {
            'status': 1001,
            'msg': 'success'
        }
        # 修改时才有instance对象，新增不需要，只需要传入数据即可
        # book_ser = BookSer(request.data)  # 报错，因为第一个位置是instance
        book_ser = BookSer(data=request.data)
        if book_ser.is_valid():
            book_ser.save()
            response_msg["data"] = book_ser.data
        else:
            response_msg['status'] = 1002
            response_msg['msg'] = 'fail'
            response_msg['data'] = book_ser.errors
        return Response(response_msg)
#serializerFile.py(其他一样，列出重写的create方法)
# 想要反序列化新增数据必须在序列化器中自己重写create方法
    def create(self, validated_data):
        # models.Book.objects.create(title=...)  # 传统意义上的创建
        instance = models.Book.objects.create(**validated_data) # 只有数据对应才可以这样
        return instance  # 必须将刚创建的对象返回


'''删除数据'''
  def delete(self, request, pk):
    response_msg = {
        'status': 1001,
```

```
        'msg': 'success'
    }
    ret = models.Book.objects.filter(pk=pk).delete()
    return Response(response_msg)


'''修改数据put/dispatch'''
#views.py
    def put(self, request, pk):
        # 初始化默认返回信息
        response_msg = {
            'status':1001,
            'msg':'success'
        }
        # 先找到这个对象
        book_obj = models.Book.objects.filter(pk=pk).first()
        # 得到序列化类的对象
        # book_ser = BookSer(book_obj, request.data) # request.data 是要修改的数据
        book_ser = BookSer(instance=book_obj, data=request.data) # request.data 是要修改的数据
        # 验证提交的数据是否符合校验要求，符合则保存，返回，不成功，返回错误信息
        if book_ser.is_valid():
            book_ser.save()  # 直接调save方法会直接报错，需要在serializers中重写update方法
            # 添加返回数据信息
            response_msg["data"] = book_ser.data
            return Response(response_msg)
        else:
            response_msg['status'] = 1002
            response_msg['msg'] = 'fail'
            response_msg['data'] = book_ser.errors
            return Response(response_msg)

#serializerFile.py(其他一样，列出重写的update方法)
    def update(self, instance, validated_data):
        # instance是book对象
        # validated_data是校验后的数据
        instance.title = validated_data.get('title')
        instance.price = validated_data.get('price')
        instance.publish = validated_data.get('publish')
        instance.save() # book.save()  django的orm提供的
        return instance


'''查询所有数据'''
# 可选==》添加校验规则，在一个视图class的get中处理==》？？开辟新的url
  url(r'^books/$', views.Books.as_view())

  class Books(APIView):
  def get(self, request):
    response_msg = {
        'status': 1001,
        'msg': 'success'
    }
    books = models.Book.objects.all()
    book_ser = BookSer(books, many=True) #many=True代表序列化多条数据，一个不需要写
    response_msg["data"] = book_ser.data
    return Response(response_msg)
```

```
'''查询时，查询多个和查询一个的底层区别==》类不一样'''
    books = models.Book.objects.all()
    book_one = models.Book.objects.filter(pk=1)
    book_ser = BookSer(books, many=True) #many=True代表序列化多条数据，一个不需要写
    book_one_ser = BookSer(book_one)
    print(type(book_ser),type(book_one_ser))
    <class 'rest_framework.serializers.ListSerializer'>
    <class 'app01.serializerFile.BookSer'>


==========================================模型类序列化器==========================================
'''写一个序列化的类，继承ModelSerializer
更改数据时，如修改和新增等，不需要重写create和update方法
其他使用，和上面的一样
'''

class BookMolSer(serializers.ModelSerializer):
  class Meta:
    model = models.Book
    # 序列化所有字段
    # fields = '__all__'

    # 序列化需要的字段()\[]都可以
    fields = ('title', 'price')

    # exclude和fields不可以同时写，代表不需要的字段
    # exclude = ('title')

    #其他需要添加的条件，在字典extra_kwargs后面添加，格式如下:
    extra_kwargs = {
      'price':{'write_only': True, 'required': True, ...},
    }


==========================================高级用法==========================================
'''source'''&'''serializers.SerializerMethodField'''
'''source不可传参
serializers.SerializerMethodField需要配合其他方法使用
'''

# models.py
    from django.db import models

    class Book(models.Model):
      title=models.CharField(max_length=32)
      price=models.IntegerField()
      pub_date=models.DateField()
      # on_delete=models.CASCADE为Django2.0以后的版本必须增加的
      publish=models.ForeignKey("Publish",on_delete=models.CASCADE,null=True)
      # 自动创建关联表
      authors=models.ManyToManyField("Author")
      def __str__(self):
        return self.title

      # 看source的引用，此函数的返回值就是response显示的value
      def test(self):
        return 'xxx'
```

```python
class Publish(models.Model):
    name=models.CharField(max_length=32)
    email=models.EmailField()
    def __str__(self):
        return self.name

class Author(models.Model):
    name=models.CharField(max_length=32)
    age=models.IntegerField()
    def __str__(self):
        return self.name


# serializerFile.py
from rest_framework import serializers
from app01.models import Book

class BookSer(serializers.Serializer):
    id = serializers.CharField(read_only=True)

    title = serializers.CharField()

    '''修改response字段名'''
    '''其实在这样输出时，所查询的内容都可以看作是
        book.id、book.price、book.title
        source做的事情就是：
        book.xxx---->book.title
    '''
    # xxx = serializers.CharField(source = "title")
    # 上面这样输出的时候xxx变成key，但是value是title的value

    price = serializers.IntegerField()
    '''使用方法'''
    # price = serializers.IntegerField(source = "test")
    # 和models.py中的函数相互作用，此处price即为key

    pub_date = serializers.DateField()

    # publish=serializers.CharField(source="publish.name",read_only=True)
    publish=serializers.CharField(source="publish.name",default='null')

    '''跨表'''
    #可以直接取出publish的email
    # publish=serializers.CharField(source="publish.email")

    # 可以直接取出所有的作者
    #authors=serializers.CharField(source="authors.all")

    '''serializers.SerializerMethodField 与 方法get_字段名联合使用 ==>格式化输出'''
    authors=serializers.SerializerMethodField(read_only=True)
    def get_authors(self,instance):     #instance就是Book对象
        temp=[]
        authors =  instance.authors.all()
        for author in authors:
            temp.append({
                'name': author.name, 'age': auehors.age   # 定制格式化输出
            })
        return temp  #return的内容就是查询请求来之后response的内容
```

==========Request&Response==========

'''
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[QueryDict]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

对Request做二次封装的源码分析：
'''

```
class Request:
    ......
    ......
    def __init__(self, request, parsers=None, authenticators=None,
                 negotiator=None, parser_context=None):
        # 利用断言先判断是否是新的request
        assert isinstance(request, HttpRequest), (
            'The `request` argument must be an instance of '
            '`django.http.HttpRequest`, not `{}.{}`.'
            .format(request.__class__.__module__, request.__class__.__name__)
        )
        self._request = request

    def __getattr__(self, attr):
        """
        If an attribute does not exist on this instance, then we also attempt
        to proxy it to the underlying HttpRequest object.

        request代理
        """
        try:
            return getattr(self._request, attr)
        except AttributeError:
            return self.__getattribute__(attr)
    ......
    ......
```

```
# 请求对象.data   前端以三种编码格式传入的数据都可以取出来
    #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

# 请求对象.query_params   与Django标准的request.GET相同，只是更换正确名称
```

'''Response响应对象
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（如使用浏览器和postman发送的格式不一样）
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

1.全局配置：通过项目settings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
    )
}
......（还有很多的默认配置都可以修改，可以进源码看）
......
```

2. 局部配置：在视图函数中配置

```
from rest_framework.renderers import JSONRenderer

class TestOne(APIView):
    renderer_classes = [JSONRenderer]  # 可以配置多个
```

```
# drf有默认的renderer：先从视图类中找==》项目的settings中找==》找不到用默认的
```

```
'''
    def __init__(self, data=None, status=None,
                 template_name=None, headers=None,
                 exception=False, content_type=None):
                # status 响应状态码
                # headers 响应头                    是个字典，可以自己添加
                # template_name 模板                可自定义模板，但是无需了解
                # exception  异常的处理方式
                # content_type 响应的编码方式       text/html和application/json

                # data:             返回的数据，字典
                # status:           响应状态码，默认200  (from rest_framework import status)

    # .data
        # 传给response对象的序列化后，但尚未render处理的数据

    # .status_code
        # 状态码的数字

    # .content
        # 经过render处理后的响应数据

'''状态码(from rest_framework import status)
    信息告知 - 1xx
        HTTP_100_CONTINUE
        HTTP_101_SWITCHING_PROTOCOLS

    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

    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

  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

  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'''

==============================APIview&GenericAPIView==============================
'''代码简要写，错误的逻辑没有增加，返回最简单的数据'''
  from rest_framework.views import APIView
  from rest_framework.generics import GenericAPIView
  from app01 import serializersFile
  from app01 import models
  from rest_framework.response import Response

'''APIView'''
  class Books(APIView):
    def get(self, request):
      book_obj = models.Book.objects.all()
      books_ser = serializersFile.BooksSer(book_obj, many=True)
```

```
        return Response(books_ser.data)

    def post(self, request):
        book_ser = serializersFile.BooksSer(data=request.data)
        if book_ser.is_valid():
            book_ser.save()
            return Response(book_ser.data)


class Book(APIView):
    def get(self, request, pk):
        book_obj = models.Book.objects.filter(pk=pk).first()
        books_ser = serializersFile.BooksSer(book_obj)
        return Response(books_ser.data)

    def put(self, request, pk):
        book_obj = models.Book.objects.filter(pk=pk).first()
        book_ser = serializersFile.BooksSer(instance=book_obj, data=request.data)
        if book_ser.is_valid():
            book_ser.save()
            return Response(book_ser.data)

    def delete(self, request, pk):
        instance = models.Book.objects.filter(pk=pk).delete()
        return Response(instance)

'''GenericAPIView
    queryset = models.Book.objects.all()                # 无论是否带上all(),内部逻辑都会有
    serializer_class = serializersFile.BooksSer         # 选择合适的序列化器

    book_obj = self.get_queryset()                      # 获取多个数据对象
    books_ser = self.get_serializer(book_obj, many=True)    #增加数据

    book_ser = self.get_serializer(data=request.data)   # 修改数据传值不变

    book_obj = self.get_object()                        # 获取单个数据对象
    books_ser = self.get_serializer(book_obj)

    book_ser = self.get_serializer(book_obj, request.data)  # 修改数据传值不变

    instance = self.get_object().delete'''


    class Books2(GenericAPIView):
        queryset = models.Book.objects.all()
        serializer_class = serializersFile.BooksSer
        def get(self,request):
            book_obj = self.get_queryset()
            books_ser = self.get_serializer(book_obj, many=True)
            return Response(books_ser.data)

        def post(self, request):
            book_ser = self.get_serializer(data=request.data)
            if book_ser.is_valid():
                book_ser.save()
                return Response(book_ser.data)

    class Book2(GenericAPIView):
```

```
        queryset = models.Book.objects.all()
        serializer_class = serializersFile.BooksSer

        def get(self,request,pk):
            book_obj = self.get_object()
            books_ser = self.get_serializer(book_obj)
            return Response(books_ser.data)

        def put(self, request, pk):
            book_obj = self.get_object()
            book_ser = self.get_serializer(book_obj, request.data)
            if book_ser.is_valid():
                book_ser.save()
                return Response(book_ser.data)

        def delete(self, request, pk):
            instance = self.get_object().delete()
            return Response(instance)
    '''
    这样封装之后，只需要改变
        queryset =
        serializer_class =
        就可以实现这五个接口的直接复用，于是可以进一步对逻辑进行封装
    '''

'''
GenericAPIView提供了五个视图扩展类：
    (from rest_framework.mixins import ...)

    CreateModelMixin
    增加数据
    UpdateModelMixin
    更新数据
    DestroyModelMixin
    删除数据
    ListModelMixin
    查看多个数据
    RetrieveModelMixin
    查看单个数据
    '''

    from rest_framework.mixins import CreateModelMixin, UpdateModelMixin,
                        DestroyModelMixin, ListModelMixin, RetrieveModelMixin

    class Books3(GenericAPIView, ListModelMixin,CreateModelMixin):
        queryset = models.Book.objects.all()
        serializer_class = serializersFile.BooksSer
        def get(self,request):
            return self.list(request)

        def post(self, request):
            return self.create(request)

    class Book3(GenericAPIView, UpdateModelMixin, DestroyModelMixin, RetrieveModelMixin):
        queryset = models.Book.objects.all()
        serializer_class = serializersFile.BooksSer

        def get(self,request,pk):
```

```
        return self.retrieve(request,pk)

    def put(self, request, pk):
        return self.update(request, pk)

    def delete(self, request, pk):
        return self.destroy(request, pk)


'''GenericAPIView的9个视图子类
   (from rest_framework.generics import..... )'''

   '''
   ListAPIView = GenericAPIView + ListModelMixin

   CreateAPIView = GenericAPIView + CreateModelMixin

   DestroyAPIView = GenericAPIView + DestroyModelMixin

   RetrieveAPIView = GenericAPIView + RetrieveModelMixin

   UpdateAPIView = GenericAPIView + UpdateModelMixin
   '''
      from rest_framework.generics import ListAPIView, CreateAPIView, \
         DestroyAPIView, RetrieveAPIView, UpdateAPIView

      class Books4(ListAPIView, CreateAPIView):
         queryset = models.Book.objects.all()
         serializer_class = serializersFile.BooksSer

      class Book4(DestroyAPIView, RetrieveAPIView, UpdateAPIView):
         queryset = models.Book.objects.all()
         serializer_class = serializersFile.BooksSer

   '''
   以下不做代码演示了，具体应该都是符合规律的
   ListCreateAPIView = ListAPIView + CreateAPIView

   RetrieveUpdateAPIView = RetrieveAPIView + UpdateAPIView

   RetrieveDestroyAPIView = RetrieveAPIView + DestroyAPIView

   RetrieveUpdateDestroyAPIView = RetrieveAPIView + UpdateAPIView + DestroyAPIView


   '''
      from rest_framework.generics import ListCreateAPIView, RetrieveUpdateDestroyAPIView

      class Books5(ListCreateAPIView):
         queryset = models.Book.objects.all()
         serializer_class = serializersFile.BooksSer

      class Book5(RetrieveUpdateDestroyAPIView):
         queryset = models.Book.objects.all()
         serializer_class = serializersFile.BooksSer


'''ModelViewSet = ListAPIView + CreateAPIView + DestroyAPIView + RetrieveAPIView
```

```
                + UpdateAPIView + GenericViewSet'''

# urls.py
    url(r'^book6$', views.Books6.as_view(actions={
        'get': 'list','post': 'create'
    })),
    url(r'^book6/(?P<pk>\d+)', views.Books6.as_view(actions={
        'get': 'retrieve','put': 'update','delete': 'destroy'
    }))
# views.py
    from rest_framework.viewsets import ModelViewSet

    class Books6(ModelViewSet):
        queryset = models.Book.objects.all()
        serializer_class = serializersFile.BooksSer


'''源码分析（只摘取了重要部分）'''
    from rest_framework.viewsets import ModelViewSet

    # 点进ModelViewSet
    class GenericViewSet(ViewSetMixin, generics.GenericAPIView):
        pass

     # 点进ViewSetMixin  发现重写了as_view方法
    class ViewSetMixin:
        ......
        @classonlymethod
        def as_view(cls, actions=None, **initkwargs):
        ......
            # actions must not be empty
            if not actions:
            def view(request, *args, **kwargs):
                self = cls(**initkwargs)

                # 将action字典导入，假设为{'get': 'list'}
                self.action_map = actions

                # 循环遍历actions得到key，value
                # 循环结束之后，所有的method和action都一一对应
                for method, action in actions.items():
                    # method: get
                    # action: list

                    # 利用反射，将action对应的method查询到之后将函数地址传给handler
                    handler = getattr(self, action)
                    #handler: list
                    setattr(self, method, handler)
                    # 对象.get = list
        ......

'''ViewSetMixin重写路由执行函数的名称，可以多种组合使用'''
# views.py
    from rest_framework.viewsets import ViewSetMixin
    from rest_framework.views import APIView

    class Books7(ViewSetMixin, APIView):
    # ViewSetMixin一定要放在最左边
```

```
    # 查询父类时，第一个查询到ViewSetMixin内部重写的as_view方法
    def get_all_books(self, request):
        book_obj = models.Book.objects.all()
        books_ser = serializersFile.BooksSer(book_obj, many=True)
        return Response(books_ser.data)

# urls.py
    url(r'^book6$', views.Books6.as_view(actions={
            'get': 'get_all_books'
        })),


=====================================自动生成路由=====================================
# 1.导入模块
    from rest_framework import routers
# 2.有两个类，实例化得到对象
    #SimpleRouter
    #DefaultRouter  生成的路由比较多
    router = routers.SimpleRouter()
# router = routers.DefaultRouter()
# 3.注册
    # router.register('前缀','继承自ModelViewSet的视图类','别名(用于方向解析)')
    router.register(prefix='books',viewset=views.Books)  # 不用加斜杠了
# 3.5.看一眼路由的样子
    #print(router.urls)
    '''
    router = routers.SimpleRouter()生成的：
    [
    <RegexURLPattern book-list ^books/$>
    <RegexURLPattern book-detail ^books/(?P<pk>[^/.]+)/$>
     ]
     '''

    urlpatterns = [
        url(r'^admin/', admin.site.urls),
        # url(r'^books/', views.Books.as_view(actions={'get': 'list', 'post': 'create'})),
        # url(r'^books/(?P<pk>\d+)', views.Books.as_view(actions={'get': 'retrieve', 'put': 'update', 'delete': 'destroy'}))
    ]
    # 4.路由加进去，由于是列表，所以可以这样直接加
    urlpatterns += router.urls

    '''
    router = routers.DefaultRouter()生成：

    [
    <RegexURLPattern book-list ^books/$>        和Simple一样
    <RegexURLPattern book-list ^books\.(?P<format>[a-z0-9]+)/?$>
        路由可以.json转换显示格式

    <RegexURLPattern book-detail ^books/(?P<pk>[^/.]+)/$>       和Simple一样
    <RegexURLPattern book-detail ^books/(?P<pk>[^/.]+)\.(?P<format>[a-z0-9]+)/?$>
        路由可以/1.json转换显示格式

    <RegexURLPattern api-root ^$>   根路径
    <RegexURLPattern api-root ^\.(?P<format>[a-z0-9]+)/?$>
        路由可以.json转换显示格式
    ]
    '''
```

```
'''默认状态下的缺陷和解决方式'''
# 自己在ModelViewSet的视图类中添加到路由方法不能自动生成
# 使用一个装饰器，将自定义的路由方法也能够自动生成路由

    from rest_framework.decorators import action
    # （导入不全，仅有action装饰器）

    class Books(ModelViewSet, APIView):
        queryset = models.Book.objects.all()
        serializer_class = serializeFiles.BookSer
         '''
        detail为布尔类型
        1.会自动生成一个路由
            <RegexURLPattern book-get-book-two ^books/get_book_two/$>
        2.action装饰器的methods的请求方式是指的当get请求从上面的路由来的时候会执行所装饰的函数
        3.detail表示生成带pk的路由
        '''
        @action(methods='get', detail=False)
        def get_book_two(self,request):
            book_obj = self.get_queryset()[:2]
            book_ser = self.get_serializer(book_obj, many=True)
            return Response(book_ser.data)


==========================================路由认证方案==========================================
'''认证的实现'''
    # 1.写一个类，继承BaseAuthentication，重写authenticate方法，里面写认证逻辑
        # 认证通过，返回两个值，第一个给了Request的user对象
        # 认证失败，报异常：APIException或者AuthenticateFailed (继承了APIException)
    # 2.全局使用，局部使用

    # authenicateFiles.py
            from rest_framework.authentication import BaseAuthentication
            from rest_framework.exceptions import AuthenticationFailed
            from app01 import models


            class myAuth(BaseAuthentication):
                def authenticate(self, request):
                    token = request.GET.get('token')
                    if token:
                        user_token = models.User_token.objects.filter(token=token).first()
                        if user_token:
                            # 如果不return  user那么最后返回的是匿名用户
                            return user_token.user,token
                        else:
                            raise AuthenticationFailed('fail')
                    else:
                        raise AuthenticationFailed('请求没有带token')

    # utils.py
        class response_login:
            def __init__(self):
                self.code = 0
                self.msg = 'success'

            def ResponseData(self):
                return self.__dict__
```

```
# views.py
# 局部使用authentication_classes = [myAuth]
    from app01.authenicateFiles import myAuth（其他的模块自己导入）

    class Books(ModelViewSet, APIView):
        authentication_classes = [myAuth]
        queryset = models.Book.objects.all()
        serializer_class = serializeFiles.BookSer

    class Login(APIView):
    def post(self, request):
        Res = response_login()
        username = request.data.get('username')
        password = request.data.get('password')
        user_obj = models.User.objects.filter(username=username, password=password).first()
        if user_obj:
            Res.data = request.data
            Res.token = uuid.uuid4()
            # if not models.User_token.objects.filter(token=Res.token):
            #     models.User_token.objects.create(token=Res.token, user=user_obj)

            # update_or_create() 有就更新，没有就增加
            models.User_token.objects.update_or_create(defaults={'token':Res.token}, user=user_obj)
            return Response(Res.ResponseData())
        else:
            Res.code = 1
            Res.msg = 'fail'
            return Response(Res.ResponseData())

# 全局使用settings里面注册，可以注册多个，从左往右执行
    REST_FRAMEWORK={
        "DEFAULT_AUTHENTICATION_CLASSES":["app01.authenticateFiles.myAuth",]
    }

#局部禁用:     authentication_classes = []


'''权限验证的实现'''
# 1.写一个类，继承BasePermission，重写has_permission方法，里面写认证逻辑
    # 权限通过，返回True
    # 权限失败，返回False
# 2.全局使用，局部使用

# authenticateFiles.py
    class UserPermission(BasePermission):
        def has_permission(self, request, view):
            # 已经认证过了，request内部有user对象了，是当前的登录用户
            user = request.user
            # get_user_type_display()
            # get_字段名_display      这句话是因为user表中的choice字段的对应中文的显示
            # print(user.get_user_type_display())
            if user.user_type == 1:
                return True
            else:
                return False

# views.py
```

```python
    from app01.authenicateFiles import myAuth, UserPermission

    class Books(ModelViewSet, APIView):
        authentication_classes = [myAuth]
        permission_classes = [UserPermission]
        ......

# 全局配置/局部配置/局部禁用和上一个类似，权限名称去默认权限内部找
    # 全局使用settings里面注册，可以注册多个，从左往右执行
    REST_FRAMEWORK={
        "DEFAULT_PERMISSION_CLASSES":["app01.authenticateFiles.UserPermission",]
    }
    # 局部使用见上述代码
    # 局部禁用:     permission_classes = []

# 内置权限类

from rest_framework.permissions import AllowAny,IsAuthenticated,
                                       IsAdminUser,IsAuthenticatedOrReadOnly

- AllowAny                          # 允许所有用户
- IsAuthenticated                    # 仅通过认证的用户
- IsAdminUser                       # 仅管理员用户
- IsAuthenticatedOrReadOnly          # 已经登陆认证的用户可以对数据进行增删改操作
                                        # 没有登陆认证的只能查看数据


'''限流（频率限制）验证的实现'''
    # 自定义频率类
    class MyThrottles():
        VISIT_RECORD = {}
        def __init__(self):
            self.history=None
        def allow_request(self,request, view):
            #取出访问者ip
            # print(request.META)
            ip=request.META.get('REMOTE_ADDR')
            import time
            ctime=time.time()
            # 判断当前ip不在访问字典里，添加进去，并且直接返回True,表示第一次访问
            if ip not in self.VISIT_RECORD:
                self.VISIT_RECORD[ip]=[ctime,]
                return True
            self.history=self.VISIT_RECORD.get(ip)

            # 循环判断当前ip的列表，有值，并且当前时间减去列表的最后一个时间大于60s
            while self.history and ctime-self.history[-1]>60:
                # 把这种数据pop掉，这样列表中只有60s以内的访问时间
                self.history.pop()

            # 判断，当列表小于3，说明一分钟以内访问不足三次，把当前时间插入到列表第一个位置
                #返回True，顺利通过
            if len(self.history)<3:
                self.history.insert(0,ctime)
                return True
            # 当大于等于3，说明一分钟内访问超过三次，返回False验证失败
            else:
                return False
```

```
        def wait(self):
            import time
            ctime=time.time()
            return 60-(ctime-self.history[-1])

    # 全局使用
        REST_FRAMEWORK = {
            'DEFAULT_THROTTLE_CLASSES':['app01.utils.MyThrottles',],
        }
    # 局部使用
        throttle_classes = [MyThrottles,]


# 使用内部频率类
    #写一个类，继承自SimpleRateThrottle（根据ip限制）
        from rest_framework.throttling import SimpleRateThrottle
        class VisitThrottle(SimpleRateThrottle):
            scope = 'luffy'
            def get_cache_key(self, request, view):
                return self.get_ident(request)

        #在setting里配置：（一分钟访问三次）
        REST_FRAMEWORK = {
            'DEFAULT_THROTTLE_RATES':{
                'luffy':'3/m'  # key要跟类中的scop对应
            }
        }

            '''# 了解：错误信息中文显示
                class Course(APIView):
                    authentication_classes = [TokenAuth, ]
                    permission_classes = [UserPermission, ]
                    throttle_classes = [MyThrottles,]

                    def get(self, request):
                        return HttpResponse('get')

                    def post(self, request):
                        return HttpResponse('post')
                    def throttled(self, request, wait):
                        from rest_framework.exceptions import Throttled
                        class MyThrottled(Throttled):
                            default_detail = '傻逼啊'
                            extra_detail_singular = '还有 {wait} second.'
                            extra_detail_plural = '出了 {wait} seconds.'
                        raise MyThrottled(wait)'''

    # 限制匿名用户每分钟访问3次
        REST_FRAMEWORK = {
            'DEFAULT_THROTTLE_CLASSES': (
                'rest_framework.throttling.AnonRateThrottle',
            ),
            'DEFAULT_THROTTLE_RATES': {
                'anon': '3/m',
            }
        }
        # 使用 `second`, `minute`, `hour` 或`day`来指明周期。
        # 可以全局使用，局部使用
```

```
# 限制登陆用户每分钟访问10次
    REST_FRAMEWORK = {
        'DEFAULT_THROTTLE_CLASSES': (
            'rest_framework.throttling.UserRateThrottle'
        ),
        'DEFAULT_THROTTLE_RATES': {
            'user': '10/m'
        }
    }
    # 可以全局使用，局部使用

# 内部类小结，通常使用内部类较多
    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

    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

    ScopedRateThrottle
        限制用户对于每个视图的访问频次，使用ip或user id。
            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
    # 全局配置中设置访问频率
    'DEFAULT_THROTTLE_RATES': {
        'anon': '3/minute',
        'user': '10/minute'
    }

    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 StudentAPIView(RetrieveAPIView):
        queryset = Student.objects.all()
```

```
        serializer_class = StudentSerializer
        authentication_classes = [SessionAuthentication]
        permission_classes = [IsAuthenticated]
        throttle_classes = (UserRateThrottle,)


'''源码分析（只摘取了重要部分）'''
    # APIView==>dispatch==> self.initialize_request(request, *args, **kwargs)
    # ==> def initial(self, request, *args, **kwargs):
    '''
    三大认证模块
        self.perform_authentication(request)
        self.check_permissions(request)
        self.check_throttles(request)
    '''
        '''self.perform_authentication(request)
        认证组件：校验用户--游客、合法用户、非法用户
            游客：校验通过，直接进去下一步校验（权限校验）
            合法用户：校验通过，将用户存储在request.user中，再进行下一步校验
            非法用户：校验失败，抛出异常，返回403
        '''
        def perform_authentication(self, request):
            # 需要去Request.user中继续查找
            request.user

        @property
        def user(self):
            # 进行查找是否有用户的属性
            if not hasattr(self, '_user'):
                # 上下文管理器
                with wrap_attributeerrors():
                    # 进行用户认证
                    self._authenticate()
            # 返回认证之后的用户
            return self._user

        # self._authenticate()的_authenticate()方法
        def _authenticate(self):
            # self为request对象    找authenticators

            # self.authenticators一开始是我们所配置的认证类，在request二次封装时初始化
            # 经过内部转换之后
            # self.authenticators最后是自己所配置的认证类产生的对象所组成的list
            # 每次循环，都可以拿到一个认证类的对象
            for authenticator in self.authenticators:
                try:
                    user_auth_tuple = authenticator.authenticate(self)
                                        # 每一个对象都执行authenticate方法
                                        # 所以我们需要重写authenticate方法进行认证
                except exceptions.APIException:
                    self._not_authenticated()
                    raise

                if user_auth_tuple is not None:
                    self._authenticator = authenticator

                    self.user, self.auth = user_auth_tuple
                    # 对返回的元组进行解压
```

```
            # 解压后的值第一个给了Request的user对象，说明只要经过认证
            # 那么一定可以从Request的user对象拿到用户登录信息

            return self._not_authenticated()


    '''self.check_permissions(request)
    权限组件--校验用户权限-必须登录、所有用户、登陆读写游客只读、自定义用户角色
       认证通过：可以进行下一步校验（频率校验）
       认证失败：抛出异常，返回403
    '''
    # 是APIview的对象方法
    def check_permissions(self, request):
        """
        Check if the request should be permitted.
        Raises an appropriate exception if the request is not permitted.
        """
        # 遍历权限对象列表，得到一个权限对象（权限器）
        for permission in self.get_permissions():
        # 权限类一定要有一个has.permission权限方法，用来做权限认证
        # 参数：权限对象self、请求对象request，视图类对象self
        # 返回值：有权限返回True，没有权限返回False
            if not permission.has_permission(request, self):
                self.permission_denied(
                    request, message=getattr(permission, 'message', None)
                )


     '''self.check_throttles(request)
    频率组件--限制视图接口被访问的频率次数
          限制条件有：IP、id、唯一键、频率周期时间(s、m、h)、频率次数
       没有达到限次：正常访问接口
       认证失败：限制时间内不能访问，限制时间达到之后可以继续访问
    '''
    def check_throttles(self, request):
        """
        Check if request should be throttled.
        Raises an appropriate exception if the request is throttled.
        """
        throttle_durations = []

        for throttle in self.get_throttles():
            if not throttle.allow_request(request, self):
                throttle_durations.append(throttle.wait())

        if throttle_durations:
            # Filter out `None` values which may happen in case of config / rate
            # changes, see #1438
            durations = [
                duration for duration in throttle_durations
                if duration is not None
            ]

            duration = max(durations, default=None)
            self.throttled(request, duration)
```

```
'''过滤组件的使用djando-filter(需要自己安装)
使用的视图一定需要是 GenericAPIView（继承它也行）
'''
    # 组件注册
        INSTALLED_APPS = [
            ...
            'django_filters', # 需要注册应用，
        ]

    # 全局使用
        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
        from rest_framework.generics import ListAPIView
        class Books2(ListAPIView):
            queryset = models.Book.objects.all()
            serializer_class = serializeFiles.BookSer
            filter_fields = ('id',)

'''排序'''
    # REST framework提供了OrderingFilter过滤器来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

        class StudentListView(ListAPIView):
            queryset = Student.objects.all()
            serializer_class = StudentModelSerializer
            filter_backends = [OrderingFilter]
            ordering_fields = ('id', 'age')

        # 127.0.0.1:8000/books/?ordering=-age
            # -id 表示针对id字段进行倒序排序
            # id  表示针对id字段进行升序排序

    # 如果需要在过滤以后再次进行排序，则需要两者结合

        from rest_framework.generics import ListAPIView
        from students.models import Student
        from .serializers import StudentModelSerializer
        from django_filters.rest_framework import DjangoFilterBackend
        class Student3ListView(ListAPIView):
            queryset = Student.objects.all()
            serializer_class = StudentModelSerializer
            filter_fields = ('age', 'sex')
            # 因为局部配置会覆盖全局配置,所以需要重新把过滤组件核心类再次声明,
            # 否则过滤功能会失效
            filter_backends = [OrderingFilter,DjangoFilterBackend]
            ordering_fields = ('id', 'age')
```

```
'''异常处理，统一接口的返回'''
    # 自定义异常处理，统一错误的返回
    # 日志记录（重要）
    from rest_framework.views import exception_handler

    def custom_exception_handler(exc, context): # exc 是异常对象的信息，context是异常对象的具体信息
        response_data = response_login()
        # 一般来说会重新调用一下exception_handler函数，后面写自己的 处理逻辑
        response = exception_handler(exc, context)
        #
        上面的函数执行完成有两种情况，一个是None，drf不处理，一个是response对象，但是处理不符合需求
        if not response:
            response_data.code = 1
            response_data.msg = str(exc)
            return Response(response_data.ResponseData())
        else:
            # return response
            response_data.code = 1
            response_data.msg = response.data.get('detail')
            return Response(data=response_data.ResponseData())

'''补充，封装自己的Response，__dict__方法'''
    # 初级封装
        class myResponse:
            def __init__(self):
                self.code = 1
                self.msg = 'success'

            # def get_dic(self): # 在使用时取个见名知意的名字
            def responseData(self):
                return self.__dict__

        if __name__ == '__main__':
            res = myRespnse()
            print(res.responseData())   # {'code': 1, 'msg': 'success'}

            # 需要添加数据时
            res.data = ...
            # 最后调用responseData方法得到字典
            res.responseData()

    # 高级封装,之后再使用Resposne，直接使用新的就可以了
    class myResponse(Response):
        def __init__(self, code=100 ,msg='请求成功', data=None, status=None,headers=None, **kwargs):
            dic = {
                'code':code,
                'msg':msg
            }
            if  data:
                dic = {
                    'code': code,
                    'msg': msg,
                    'data': data
                }
            dic.update(kwargs)
            super().__init__(data=dic, status=status, headers=headers)
```