Python之Django-REST framework

原创

咕咕@ 于 2021-11-29 21:33:02 发布 666 收藏 7
分类专栏： python 文章标签： python rabbitmq batch

本文链接：https://blog.csdn.net/qq_40887840/article/details/121564690

前言-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

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
```

3. 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
```

4. 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。

5. 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。

6. 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"
}
```

7. 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

8. 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. 环境安装

1.1 安装

```
pip install djangorestframework
```

1.2 添加rest_framework应用

> 在Django工程的settings.py的INSTALLED_APPS中添加'rest_framework'。

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

2. 序列化器

2.1 定义序列化器

> 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

```
from django.db import models

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

```
from rest_framework import serializers

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=False)
    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。

详细字段及字段属性描述见文档

2.2 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

```
bSerializer = BookInfoSerializer(instance=book, context={'remarks':'这是一个备注'})
```

3. 序列化的基本使用

自己准备一些数据插入数据库

接下来的基本测试是使用python manage.py shell 打开shell脚本测试

3.1 查询出一个图书对象

```
>>> from book.models import BookInfo
>>> book = BookInfo.objects.get(id=3)
```

3.2 构造序列化器对象

```
>>> from start_drf.test import BookInfoSerializer
>>> serializer = BookInfoSerializer(instance=book, context={'remarks':'这里是备注'})
```

3.3 获取序列化数据

```
>>> serializer
BookInfoSerializer(context={'remarks': '这里是备注'}, instance=<BookInfo: BookInfo object (3)>):
    id = IntegerField(label='ID', read_only=True)
    btitle = CharField(label='名称', max_length=20)
    bpub_date = DateField(label='发布日期', required=False)
    bread = IntegerField(label='阅读量', required=False)
    bcomment = IntegerField(label='评论量', required=False)
    image = ImageField(label='图片', required=False)

# 通过data属性可以获取序列化后的数据
>>> serializer.data
{'id': 3, 'btitle': '红楼梦', 'bpub_date': '2021-11-26', 'bread': 5, 'bcomment': 3, 'image': '/booktest/5cb77315
e3ecb2e6891ce49211016233.jpeg'}
```

3.4 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_all = BookInfo.objects.all()
>>>
>>> serializer = BookInfoSerializer(instance = book_all, many=True)
>>> serializer.data
#[OrderedDict([('id', 1), ('btitle', '西游记'), ('bpub_date', '2021-11-26'), ('bread', 20), ('bcomment', 10), ('
image', '/booktest/bccc868599405390f744e3e46ddd0341.jpeg')]), OrderedDict([('id', 2), ('btitle', '水浒传'), ('bp
ub_date', '2021-11-26'), ('bread', 22), ('bcomment', 1), ('image', '/booktest/b319c3617681252f061105603d1ff19d.j
peg')]), OrderedDict([('id', 3), ('btitle', '红楼梦'), ('bpub_date', '2021-11-26'), ('bread', 5), ('bcomment', 3
), ('image', '/booktest/5cb77315e3ecb2e6891ce49211016233.jpeg')]), OrderedDict([('id', 4), ('btitle','三国演义'),
 ('bpub_date', '2021-11-26'), ('bread', 22), ('bcomment', 3), ('image', '/booktest/5d04644e95a430f8bbe3d98878c2a
7b7.jpg')])]
```

3.5 关联对象嵌套的序列化

3.5.1 多对一

> 一对一也可当做多对一处理

在models.py中添加一个英雄人物模型，并往数据库中写入一下信息

```python
# 在models.py中添加一个英雄人物模型，并往数据库中写入一下信息
class HeroInfo(models.Model):
    """英雄数据序列化器"""
    GENDER_CHOICES = (
        ('MALE', '男'),
        ('FEMALE', '女')
    )
    hname = models.CharField(verbose_name='名字', max_length=20)
    hgender = models.CharField(choices=GENDER_CHOICES, verbose_name='性别', max_length=6)
    hcomment = models.CharField(verbose_name='描述信息', max_length=200, null=True)
    hbook = models.ForeignKey(verbose_name='所属图书', to= BookInfo, on_delete=models.SET_NULL, null=True)

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

定义英雄人物的序列化器类

```python
class HeroInfoSerializer(serializers.Serializer):
    """英雄数据序列化器"""
    GENDER_CHOICES = (
        ('MALE', '男'),
        ('FEMALE', '女')
    )
    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)
    #hbook = serializers.PrimaryKeyRelatedField(label='所属图书', read_only=True)
    #hbook = serializers.StringRelatedField(label='所属图书')
    #hbook = BookInfoSerializer()
```

注意：不论是一对多还是多对一，序列化器中的外键命名需按照django模型类查询规范来命名
多对一查询：多的一方的模型类对象.一对应的模型类名小写
一对多查询：一对应的模型类对象.多对应的模型类名小写_set

故：在序列化器类中，
多对一查询应定义外键属性名为：一对应的模型类名小写；(不强制，因为这个在配置模型类时已指明)
一对多查询中应定义外键属性名为：多对应的模型类名小写_set (多对一必须这样命名)

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。

```python
hbook = serializers.PrimaryKeyRelatedField(label='所属图书', read_only=True)
```

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

使用效果如下：

```
>>> from start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> from book.models import BookInfo,HeroInfo
>>> hero = HeroInfo.objects.get(id=1)
>>> serializer = HeroInfoSerializer(hero)
>>> serializer.data
{'id': 1, 'hname': '孙悟空', 'hgender': 'MALE', 'hcomment': 'a monkey', 'hbook': 1}
```

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='所属图书')
```

使用效果如下：

```
>>> from book.models import BookInfo,HeroInfo
>>> from start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> hero = HeroInfo.objects.get(id=1)
>>> serializer = HeroInfoSerializer(hero)
>>> serializer.data
{'id': 1, 'hname': '孙悟空', 'hgender': 'MALE', 'hcomment': 'a monkey', 'hbook': '西游记'}
```

3) 使用关联对象的序列化器

在外键出定义外键属性为关联对象的序列化器

```
hbook = BookInfoSerializer()
```

使用效果

```
>>> from book.models import BookInfo,HeroInfo
>>> from start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> hero = HeroInfo.objects.get(id=1)
>>> serializer = HeroInfoSerializer(hero)
>>> serializer.data
{'id': 1, 'hname': '孙悟空', 'hgender': 'MALE', 'hcomment': 'a monkey', 'hbook': OrderedDict([('id', 1), ('btitl
e', '西游记'), ('bpub_date', '2021-11-26'), ('bread', 20), ('bcomment',
 10), ('image', '/booktest/bccc868599405390f744e3e46ddd0341.jpeg')])}
>>>
```

3.5.2 一对多

一对多时，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。

在图书序列化器中添加关联项

```
heroinfo_set = serializers.PrimaryKeyRelatedField(label='拥有英雄',read_only=True, many=True)
```

> 注意：
> django模型由一到多的访问语法规则：一对应的模型类对象.多对应的模型类名小写_set
> 故定义一外键属性名为：多对应的模型类名小写_set

使用效果

```
>>> from start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> from book.models import BookInfo,HeroInfo
>>> book = BookInfo.objects.get(id=1)
>>> serializer = BookInfoSerializer(book)
>>> serializer.data
{'id': 1, 'btitle': '西游记', 'bpub_date': '2021-11-26', 'bread': 20, 'bcomment': 10, 'image': '/booktest/bccc86
8599405390f744e3e46ddd0341.jpeg', 'heroinfo_set': [1, 2, 3, 4]}
```

4. 反序列化的基本使用

4.1 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=False)
    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，需将要反序列化的数据传递给data构造参数，进而进行验证。默认验证规则就是我们给每个字段的定义的选项参数。如：

```
>>> from start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> data = {'bpub_date': 123}
>>> serializer = BookInfoSerializer(data=data)
>>> serializer.is_valid()
False
>>> serializer.errors
{'btitle': [ErrorDetail(string='This field is required.', code='required')], 'bpub_date': [ErrorDetail(string='D
ate has wrong format. Use one of these formats instead: YYYY-MM-DD.', code='invalid')]}
>>> serializer.validated_data
{}
>>> data = {'btitle': 'python'}
>>> 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。

```
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

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

    def validate_btitle(self, value):
        """自定义字段属性验证"""
        if 'django' not in value.lower():
            raise serializers.ValidationError("图书不是关于Django的")
        return value
```

测试：

```
>>> from start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> 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

```python
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 start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> data = {'btitle': 'django基本使用', 'bread':2, 'bcomment':5}
>>> serializer = BookInfoSerializer(data=data)
>>> serializer.is_valid()
False
>>> serializer.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(需要有验证函数)，如

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

class BookInfoSerializer(serializers.Serializer):
    """图书数据序列化器"""
    ...
    btitle = serializers.CharField(label='名称', max_length=20, validators=[about_django])
    ...
```

测试

```
>>> from start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> data = {'btitle': 'Python'}
>>> serializer = BookInfoSerializer(data = data)
>>> serializer.is_valid()
False
>>> serializer.errors
{'btitle': [ErrorDetail(string='About_Django:图书不是关于Django的', code='invalid')]}
>>>
```

4.2 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。

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

    def create(self, validated_data):
        # 保存数据到数据库
        # return BookInfo.objects.create(validated_data)
        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)
        # 将数据保存到数据库
        #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。

```
# 无instance
>>> from start_drf.test import BookInfoSerializer, HeroInfoSerializer
>>> data = {'btitle':'封神演义'}
>>> serializer = BookInfoSerializer(data = data)
>>> serializer.is_valid()
True
>>> serializer.save()
<BookInfo: 封神演义>

# 有instance
>>> from book.models import BookInfo
>>> book = BookInfo.objects.get(id=5)
>>> book
<BookInfo: 封神演义>
>>> data = {'btitle':'倚天剑', 'bpub_date':'2021-11-29', 'bread': 99}
>>> serializer = BookInfoSerializer(instance = book, data = data)
>>> serializer.is_valid()
True
>>> serializer.save()
<BookInfo: 倚天剑>
>>> book
<BookInfo: 倚天剑>
```

4.3 说明

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

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


...
class BookInfoSerializer(serializers.Serializer):
 ...
 def create(self, validated_data):
     owner = validated_data.get('owner', None)
     # 保存数据到数据库
     return BookInfo.objects.crea5te(**validated_data)
```

5. 模型类序列化器ModelSerializer

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
- 基于模型类自动为Serializer生成validators，比如unique_together
- 包含默认的create()和update()的实现

5.1 定义

比如我们创建一个BookInfoSerializer：

```
from rest_framework import serializers
from book.models import BookInfo

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

    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 book.serializer 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)
>>>
```

5.2 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):
    """图书序列化器"""

    class Meta:
        model = BookInfo
        # fields = '__all__'
        fields = ('id', 'btitle', 'bpub_date')
```

2. 使用exclude可以明确排除掉哪些字段

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

    class Meta:
        model = BookInfo
        exclude = ('image',)
```

3. 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):
    """图书序列化器"""

    class Meta:
        model = BookInfo
        exclude = ('image',)
        read_only_fields = ('id', 'bread', 'bcomment')
```

5.3 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

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

    class Meta:
        model = BookInfo
        exclude = ('image',)
        read_only_fields = ('id', 'bread', 'bcomment')
        extra_kwargs = {
            'bread': {'min_value': 0, 'required': True},
            'bcomment': {'min_value': 0, 'required': True},
        }
```

```
>>> from book.serializer 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', min_value=0, read_only=True)
    bcomment = IntegerField(label='评论量', min_value=0, read_only=True)
```

6 视图

6.1 Request 对象

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。

6.2 Request对象的常用属性

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、request.body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RESTframework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。

6.3 Response对象

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。

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。

6.5 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

2）成功 - 2xx

3）重定向 - 3xx

4）客户端错误 - 4xx

5）服务器错误 - 5xx

状态码详情见官方文档

7. 视图说明

7.1 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 列表或元祖，身份认证类
- permissoin_classes 列表或元祖，权限检查类
- throttle_classes 列表或元祖，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。

```
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APIVIew，增加了对于列表视图和详情视图可能用到的通用支持方法。通常使用时，可搭配一个或多个Mixin扩展类。

支持定义的属性：

列表视图与详情视图通用：

- queryset 列表视图的查询集
- serializer_class 视图使用的序列化器

列表视图使用：

- pagination_class 分页控制类
- filter_backends 过滤控制后端

详情页视图使用：

- lookup_field 查询单一数据库对象时使用的条件字段，默认为'pk'
- lookup_url_kwarg 查询单一数据时URL中的参数关键字名称，默认与look_field相同

提供的方法：

列表视图与详情视图通用：

get_queryset(self)

返回视图使用的查询集，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_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，被其他视图或扩展类使用，如果我们在视图中想要获取序列化器对象，可以直接调用此方法。

注意，在提供序列化器对象的时候，REST framework会向对象的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。

详情视图使用：

get_object(self) 返回详情视图所需的模型类数据对象，默认使用lookup_field参数来过滤queryset。在视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。

```
# 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()
        serializer = self.get_serializer(book)
        return Response(serializer.data)
```

7.2 五个扩展类

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：

```python
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：

```python
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：

```
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：

```
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()
```

7.3 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）RetireveAPIView
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

8 视图集

8.1 视图集概念

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。

8.2 视图集使用

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):
        ...

    def retrieve(self, request, pk=None):
        ...
```

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'})
]
```

8.3 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
```

8.4 常用视图集父类

1） ViewSet
继承自APIView，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。

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
继承自GenericAPIView，作用也与GenericAPIView类似，提供了get_object、get_queryset等方法便于列表视图与详情信息视图的开发。

3）ModelViewSet
继承自GenericAPIView，同时包括了ListModelMixin、RetrieveModelMixin、CreateModelMixin、UpdateModelMixin、DestoryModelMixin。

4）ReadOnlyModelViewSet

继承自GenericAPIView，同时包括了ListModelMixin、RetrieveModelMixin。

8.5 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。

添加自定义动作需要使用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: 该action支持的请求方式，列表传递
- detail: 表示是action中要处理的是否是视图资源的对象（即是否通过url路径获取主键）
  - True 表示使用通过URL获取的主键对应的数据对象
  - False 表示不使用URL获取主键

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

    # detail为False 表示不需要处理具体的BookInfo对象
    @action(methods=['get'], detail=False)
    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)

    # detail为True，表示要处理具体与pk主键对应的BookInfo对象
    @action(methods=['put'], detail=True)
    def read(self, request, pk):
        """
        修改图书的阅读量数据
        """
        book = self.get_object()
        book.bread = request.data.get('read')
        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'})),
]
```

9 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

9.1 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))
]
```

9.2 视图集中包含附加action的

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

    @action(methods=['get'], detail=False)
    def latest(self, request):
        ...

    @action(methods=['put'], detail=True)
    def read(self, request, pk):
        ...
```

此视图集会形成的路由：

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

9.3 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。

10 其他功能（认证、权限、限流、过滤、排序、分页、异常、文档）

10.1 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

10.2 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

1. 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,)
    ...
```

2. 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

3. 例子

```
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]
```

10.3 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。

1. 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。

DEFAULT_THROTTLE_CLASSES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,)
    ...
```

2. 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：

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

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

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

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

3. 例子

```
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,)
```

10.4 过滤

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
```

10.4 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
```

10.5 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_class**属性来指明。

```
class LargeResultsSetPagination(PageNumberPagination):
 """自定义pagenation类"""
    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
```

可继承的分页器

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设置一致
- 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

```python
from rest_framework.pagination import LimitOffsetPagination

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
```

10.6 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

```python
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

```python
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

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

例如：

补充上处理关于数据库的异常

```python
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

10.7 接口文档生成

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'))
]
```

3) 文档描述说明的定义位置

a) 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.
    """
```

b）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.
    """
```

c）对于视图集ViewSet，仍在类视图的文档字符串中分开定义，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
    """
```

d) 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。