web开发框架之DRF

转载

dengzhichou9378 于 2019-02-20 23:57:00 发布 87 收藏
文章标签： 数据库 前端 json ViewUI
原文链接：http://www.cnblogs.com/wangxiongbing/p/10410036.html

RESTful架构解释：

```
# Representational State Transfer  表现层状态转化
```

到底什么是RESTful架构： 如果一个架构符合REST原则，就称它为RESTful架构。要理解RESTful架构，理解Representational State Transfer

这三个单词的意思。

具象的，就是指表现层，要表现的对象也就是“资源”，什么是资源呢？网站就是资源共享的东西，客户端（浏览器）访问web服务器，所获取的就叫资源。比如html，txt，

json，图片,视频等

表现，比如，文本可以用txt格式表现，也可以用HTML格式、XML格式、JSON格式表现，甚至可以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现。

浏览器通过URL确定一个资源，但是如何确定它的具体表现形式呢？应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"

的描述。

状态转换， 就是客户端和服务器互动的一个过程，在这个过程中，势必涉及到数据和状态的变化,这种变化叫做状态转换。

互联网通信协议HTTP协议，客户端访问必然使用HTTP协议，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。

HTTP协议实际上含有4个表示操作方式的动词，分别是GET,POST,PUT,DELETE,他们分别对应四种操作。GET用于获取资源，POST用于新建资源，PUT用于更新资源，

DElETE用于删除资源。GET和POST是表单提交的两种基本方式，比较常见，而PUT和DElETE不太常用。而且HTTP协议是一种无状态协议，这样就必须把所有的状态都保存在

服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）

表现具象的状态转化简言之一句话就是：客户端和服务器之间传递资源时是以什么数据格式来传递的

RESTful架构就是：

每一个URL代表一种资源；

客户端和服务器之间，传递这种资源的某种表现层；

客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"

RESTful设计方法

1. 域名

应该尽量将API部署在专用域名之下。

2. 版本

应该将API的版本号放入URL

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）路由设置的几种方法：

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

2)API中的名词应该使用复数,无论子资源或者所有资源

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

5.状态码
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。
405 NOT POST - [*] 表示发出的URL路径请求正确,但是未执行与该URl匹配的类视图函数
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
7.返回结果
8.超媒体

使用Django开发REST 接口
具体参见DRF_restful工程

在开发REST API接口时，视图中做的最主要有三件事：
1)将请求的数据（如JSON格式）转换为模型类对象
2)操作数据库
3)将模型类对象转换为响应的数据（如JSON格式）

序列化和反序列化的实现过程
序列化:将Python对象转换为json格式的字符串
从数据库中获取数据,并过滤,将前端所需要的数据返回给前端
输出
id的read_only设置为True
反序列化: 将json格式的字符串转换为Python对象
从前端界面获取数据,进行校验成功,将成功后的值返回给数据库
输入
id的write_only设置为True

在开发REST API接口时，我们在视图中需要做的最核心的事是：
1)将数据库数据序列化为前端所需要的格式，并返回；
2)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。

Django REST framework 简介
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。
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。
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

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

内置了限流系统；直观的 API web 界面；可扩展性，插件丰富

DRF工程搭建

环境安装和配置

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',
]
```

定义Serializer

1. 定义方法

Django RESTframework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', 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：

```
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

2)Serializer常用的字段

⊞⊟

```
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(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: "12345678901231231313412451235114514511 4"
                    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=)
```

View Code

```
            常用的参数设置
                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    该字段的最大值
                min_value    该字段的最小值
                read_only    表明该字段仅用于序列化输出(一般指往前端返回的数据),让某个属性只做过滤,不做验证(主键)
                write_only    表明该字段仅用于反序列化输入(一般需要保存在数据库中的数据),让某些属性只做验证,不做过滤(逻
辑删除)
                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
                auto_now_add  这个参数的默认值为false，设置为true时，会在model对象第一次被创建时，将字段的值设置为创
建时的时间，以后修改对象时，
                                字段的值不会再更新
                auto_now    这个参数的默认值为false，设置为true时，能够在保存该字段时，将其值设置为当前时间，并且每次
修改model，都会自动更新
```

修改model，都会自动更新。

因此这个参数在需要存储 "最后修改时间"的场景下，十分方便

```
related_name = "subs"  # 重新制定set_类名
on_delete  级联设置
```

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

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
```

结果为:

```
# {'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
```

结果为:

```
# [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),
('bcomment', 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。

外键一般定义字段的方法:

1）PrimaryKeyRelatedField

2） StringRelatedField

3）HyperlinkedRelatedField

4）SlugRelatedField

5）使用关联对象的序列化器

6）重写to_representation方法

```
many参数,一般用作查询集
book_qs = BookInfo.objects.all()
serializer = BookInfoSerializer(book_qs, many=True)
```

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。

```
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，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'}
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

自定义验证行为

```
serializer.is_valid(raise_exception=True)
```

1）validate_<field_name>

对<field_name>字段进行验证

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

3）validators在字段中添加validators选项参数，也可以补充验证行为

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

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

方法的时候，update()被调用

```
from db.serializers import BookInfoSerializer
data = {'btitle': '封神演义'}
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'}
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：

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

2）默认序列化器必须传递所有required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更新

```
serializer = CommentSerializer(comment, data={'content': u'foo bar'}, partial=True)
```

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：

1)基于模型类自动生成一系列字段

2)基于模型类自动为Serializer生成validators，比如unique_together

3)包含默认的create()和update()的实现

1. 定义

```
class BookInfoSerializer(serializers.ModelSerializer):
    """图书数据序列化器"""
    class Meta:
        # model 指明参照哪个模型类
        # fields 指明为模型类的哪些字段生成
        model = BookInfo
        fields = '__all__'
```

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

如：

```
>>> 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

如：

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

3）默认ModelSerializer使用主键作为关联字段，但是我们可以使用depth来简单的生成嵌套表示，depth应该是整数，表明嵌套的层级数量

如：

```
    class HeroInfoSerializer2(serializers.ModelSerializer):
        class Meta:
            model = HeroInfo
            fields = '__all__'
            depth = 1
```

形成的序列化器如下：

```
        HeroInfoSerializer():
            id = IntegerField(label='ID', read_only=True)
            hname = CharField(label='名称', max_length=20)
            hgender = ChoiceField(choices=((0, 'male'), (1, 'female')), label='性别',
required=False, validators=[<django.core.valators.MinValueValidator object>,
<django.core.validators.MaxValueValidator object>])
            hcomment = CharField(allow_null=True, label='描述信息', max_length=200,
required=False)
            hbook = NestedSerializer(read_only=True):
                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)
```

4）显示指明字段

如：

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

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

5）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

如：

```
    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

如:

```
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': {'max_value': 0, 'required': True}},
        }
```

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

```
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)
```

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RESTframework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：

1)包含了解析之后的文件和非文件数据

2)包含了对POST、PUT、PATCH请求方式解析后的数据

3)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。

RESTframework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。

如:

```
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

View Code

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。

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，

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，

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

扩展类和子类查看DRF框架中view的演变