django-rest-framework

一、序列化

序列化可以把查询集和模型对象转换为json、xml或其他类型，也提供反序列化功能。，也就是把转换后的类型转换为对象或查询集。

REST框架中的序列化程序与Django `Form`和`ModelForm`类的工作方式非常相似。我们提供了一个`Serializer`类，它为您提供了一种强大的通用方法来控制响应的输出，以及一个`ModelSerializer`类，它提供了一个有用的快捷方式来创建处理模型实例和查询集的序列化程序。

1.1声明序列化器

```
from rest_framework import serializers

class xxxSerializer(serializers.Serializer):
    email = serializers.EmailField()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

1.2 常用field类

- 核心参数

参数名	缺省值	说明
read_only	False	表明该字段仅用于序列化输出，默认False
required	True	如果在反序列化期间未提供字段，则会引发错误。如果在反序列化期间不需要此字段，则设置为false。
default		缺省值，部分更新时不支持
allow_null	False	表明该字段是否允许传入None，默认False
validators	[]	验证器，一个函数列表，验证通不过引起serializers.ValidationError
error_messages	{}	一个错误信息字典

- 常用字段

字段名	说明
BooleanField	对应于django.db.models.fields.BooleanField
CharField	CharField(max_length=None, min_length=None, allow_blank=False,trim_whitespace=True) max_length - 验证输入包含的字符数不超过此数量。 min_length - 验证输入包含不少于此数量的字符。allow_blank- 如果设置为True则则应将空字符串视为有效值。如果设置为False则则空字符串被视为无效并将引发验证错误。默认为False。trim_whitespace- 如果设置为Truethen则会修剪前导和尾随空格。默认为True
EmailField	EmailField(max_length=None, min_length=None, allow_blank=False)
IntegerField	IntegerField(max_value=None, min_value=None) max_value 验证提供的数字是否不大于此值。min_value 验证提供的数字是否不低于此值。
FloatField	FloatField(max_value=None, min_value=None)
DateTimeField	DateTimeField(format=api_settings.DATETIME_FORMAT, input_formats=None, default_timezone=None) format格式字符串可以是显式指定格式的Python strftime格式 input_formats - 表示可用于解析日期的输入格式的字符串列表

1.3 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 = UserSerializer(User, 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。

- 实例

```
#models.py
class User(models.Model):
    username = models.CharField(max_length=30)
    password_hash =
models.CharField(max_length=20,db_column='password')
    age = models.IntegerField(default=0)

    class Meta:
        db_table = 'user'


#serializers.py
def validate_password(password):
    if re.match(r'\d+$',password):
        raise serializers.ValidationError("密码不能是纯数字")

class UserSerializer(serializers.Serializer):
    # id = serializers.IntegerField()
    username = serializers.CharField(max_length=30)
    password_hash = serializers.CharField(min_length=3,validators=
[validate_password])
    age = serializers.IntegerField()
```

```
    def create(self, validated_data):
        return User.objects.create(**validated_data)
    def update(self, instance, validated_data):
        instance.username =
validated_data.get('username',instance.username)
        instance.password_hash =
validated_data.get('password_hash',instance.password_hash)
        instance.age = validated_data.get('age',instance.age)
        instance.save()
        return instance
```

```
#view.py
class ExampleView(APIView):
    def get(self,request):
        # 1.查询数据
        user = User.objects.get(pk=1)
        # 2.构造序列化器
        serializer = UserSerializer(instance=user)
        # 获取序列化数据，通过data属性可以获取序列化后的数据
        print(serializer.data)
        return Response(serializer.data)
```

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

```
data = User.objects.all()
serializer = UserSerializer(instance=data,many=True)
print(serializer.data)
```

1.4 ModelSerializer

ModelSerializer类能够让你自动创建一个具有模型中相应字段的Serializer类。这个ModelSerializer类和常规的Serializer类一样，不同的是：

- 它根据模型自动生成一组字段。
- 它自动生成序列化器的验证器，比如unique_together验证器。
- 它默认简单实现了.create()方法和.update()方法。

1. **定义**

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

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

2. 指定字段

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

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

2) 使用**exclude**可以明确排除掉哪些字段

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

3) 指明只读字段

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

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

3. 添加额外参数

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

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

1.5 反向序列化

- 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。

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

如果觉得这些还不够，需要再补充定义验证行为，可以使用以下三种方法：

1）`validate_<field_name>`

对`<field_name>`字段进行验证，如

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

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

2）validate

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

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

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

3）validators

在字段中添加validators选项参数，也可以补充验证行为，如

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

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

2. 保存

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

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

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

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

```
        return instance
```

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

```
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。

```
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'
```

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)
```

1.6 关联对象的序列化

如果模型中两个模型有外键关系，如果需要序列化的模型中包含有其他关联对象，则对关联对象数据的序列化需要指明。例如，在定义英雄数据的序列化器时，外键hbook（即所属的图书）字段如何序列化?

我们先定义HeroInfoSerialzier除外键字段外的其他部分：

```
# models.py
```

```
class BookInfo(models.Model):
    id = models.AutoField(primary_key=True)
    btitle = models.CharField(max_length=200)
    bpub_date = models.DateField(auto_now_add=True,null=True)
    bread = models.IntegerField(default=0)
    bcomment = models.IntegerField(default=0)
    bimage = models.CharField(max_length=200,null=True)

    class Meta:
        db_table = 'bookinfo'

class HeroInfo(models.Model):
    hid = models.AutoField(primary_key=True)
    hname = models.CharField(max_length=50)
    book =
models.ForeignKey(BookInfo,db_column='bid',related_name='heros',o
n_delete=models.CASCADE)

    class Meta:
        db_table = 'heroinfo'

    def __str__(self):
        return "{}:{}".format(self.hid,self.hname)
```

对于外键关联字段，可以采用以下方式：

1. PrimaryKeyRelatedField

此字段将被序列化为关联对象的主键。

```
class HeroInforSerializer(serializers.ModelSerializer):
    class Meta:
        model = HeroInfo
        fields = "__all__"

class BookInfoSerializer(serializers.ModelSerializer):
    heros = PrimaryKeyRelatedField(many=True,read_only=True)
    class Meta:
        model = BookInfo
        fields = '__all__'
```

- many=True，表示可以有多个主键值

- read_only=True,该字段将不能用作反序列化使用

效果：

```
[
    {
        "id": 1,
        "heros": [1,4 ],
        "btitle": "射雕英雄传","bpub_date": "2020-02-18","bread":
30,
        "bcomment": 80,"bimage": null
    },
]
```

2. StringRelatedField

此字段将被序列化为关联对象的字符串表示方式（即模型对象`__str__`方法的返回值）

```
hbook = serializers.StringRelatedField(many=True,read_only=True)
```

效果：

```
[
    {
        "id": 1,
        "heros": ["1:郭靖","4:老顽童"],
        "btitle": "射雕英雄传", "bpub_date": "2020-02-18",
"bread": 30,"bcomment": 80,
        "bimage": null
    },
]
```

3. 使用关联对象的序列化器

```
heros = HeroInforSerializer(many=True,read_only=True)
```

效果：

```
[
   {"id": 1,
   "heros": [{"hid": 1, "hname": "郭靖", "book": 1},{"hid": 4,
"hname": "老顽童", "book": 1}],
  "btitle": "射雕英雄传","bpub_date": "2020-02-18","bread": 30,
"bcomment": 80, "bimage": null
  },
]
```

二、Request和Response

1.Request

REST框架引入了一个扩展了常规HttpRequest的Request对象，并提供了更灵活的请求解析。Request对象的核心功能是request.data属性，它与request.POST类似，但对于使用Web API更为有用。

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。

1) data属性

`request.data` 返回解析之后的请求体数据。类似于Django中标准的 `request.POST` 和 `request.FILES` 属性，但提供如下特性：

- 包含了解析之后的文件和非文件数据
- 包含了对POST、PUT、PATCH请求方式解析后的数据
- 利用了REST framework的parsers解析器，不仅支持表单类型数据，也支持JSON数据

```
request.POST  # 只处理表单数据  只适用于'POST'方法
request.data  # 处理任意数据  适用于'POST', 'PUT'和'PATCH'方法
```

2）query_params 查询参数

`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。

3）method

request.method 返回请求的HTTP方法的 大写字符串表示形式。

4）自定义解析器

REST framework的请求对象提供灵活的请求解析，允许你以与通常处理表单数据相同的方式使用JSON数据或其他媒体类型处理请求。

可以使用DEFAULT_PARSER_CLASSES设置全局默认的解析器集。例如，以下设置将仅允许具有JSON内容的请求，而不是JSON或表单数据的默认值。

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
        'rest_framework.parsers.JSONParser',
    )
}
```

```
你还可以设置用于单个视图或视图集的解析器， 使用APIView类视图。

    from rest_framework.parsers import JSONParser
    from rest_framework.response import Response
    from rest_framework.views import APIView
    class ExampleView(APIView):
        """
        可以接收JSON内容POST请求的视图。
        """
        parser_classes = (JSONParser,)
        def post(self, request, format=None):
            return Response({'received data': request.data})
或者，如果你使用基于方法的视图的@api_view装饰器。

from rest_framework.decorators import api_view
from rest_framework.decorators import parser_classes

@api_view(['POST'])
@parser_classes((JSONParser,))
def example_view(request, format=None):
    """
```

```
    可以接收JSON内容POST请求的视图
    """
    return Response({'received data': request.data})
```

2.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。

```
from rest_framework.response import Response
Response(data, status=None, template_name=None, headers=None,
content_type=None)
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。
```

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。

3. 状态码（Status codes）

在你的视图（views）中使用纯数字的HTTP 状态码并不总是那么容易被理解。而且如果错误代码出错，很容易被忽略。REST框架为status模块中的每个状态代码（如HTTP_400_BAD_REQUEST）提供更明确的标识符。使用它们来代替纯数字的HTTP状态码是个很好的主意。

1）信息告知 - 1xx

```
HTTP_100_CONTINUE
HTTP_101_SWITCHING_PROTOCOLS
```

2）成功 - 2xx

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

3）重定向 - 3xx

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

4）客户端错误 - 4xx

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

```
HTTP_424_FAILED_DEPENDENCY
HTTP_428_PRECONDITION_REQUIRED
HTTP_429_TOO_MANY_REQUESTS
HTTP_431_REQUEST_HEADER_FIELDS_TOO_LARGE
HTTP_451_UNAVAILABLE_FOR_LEGAL_REASONS
```

5）服务器错误 - 5xx

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

4. wrapping

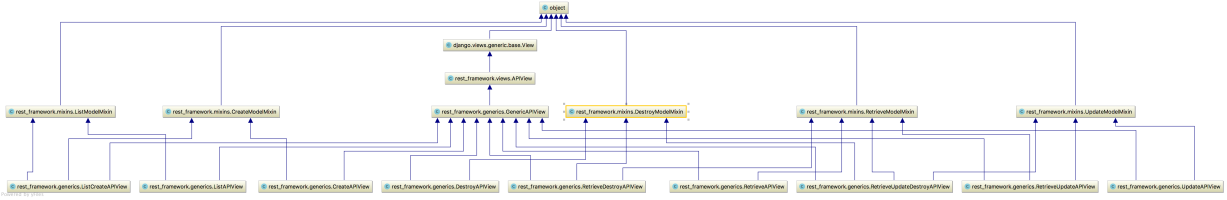
REST框架提供了两个可用于编写API视图的包装器（wrappers）。

- 用于基于函数视图的@api_view装饰器。
- 用于基于类视图的APIView类。

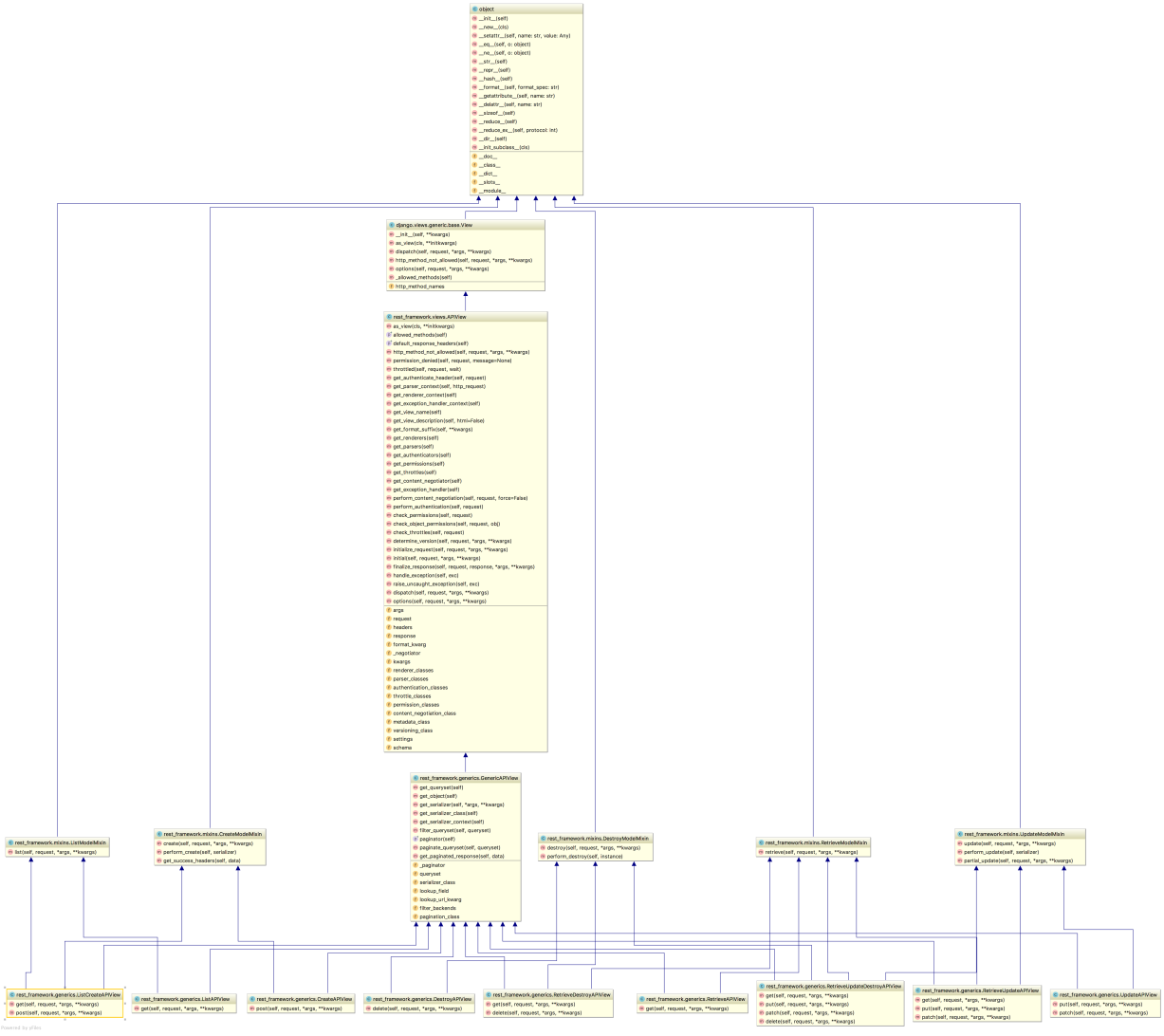
```
from rest_framework.decorators import api_view

@api_view(['GET', 'POST'])
def snippet_list(request):
    pass
```

三、基于类的视图（CBV）

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

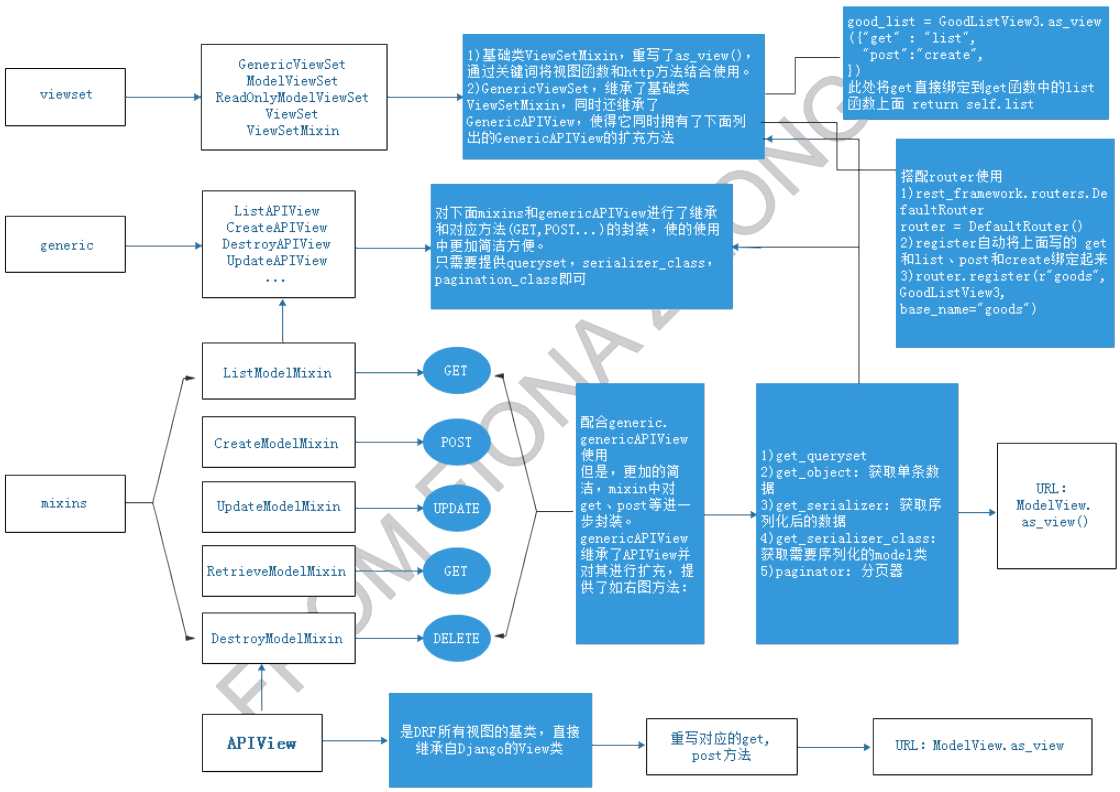


视图的方法与属性：



1.APIView

APIView是DRF的基类，它支持GET POST PUT DELETE等请求类型,且各种类型的请求之间，有了更好的分离在进行dispatch分发之前，会对请求进行身份认证，权限检查，流量控制。APIView有两个：



`rest_framework.views.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.mixins

使用基于类的视图的一个最大的好处就是我们可以灵活的选择各种View，使我们的开发更加的简洁。mixins里面对应了ListModelMixin，CreateModelMixin，RetrieveModelMixin，UpdateModelMixin，DestroyModelMixin。

- 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。

  保存方法perform_create(self, serializer),

  成功获取请求头的方法：get_success_headers(self, data)

  ```
  # 源码
  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 {}
```

- 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。

```
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)
```

- 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)
```

实例：

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

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

- 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)
```

- DestroyModelMixin

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

  成功返回204，不存在返回404。

  源代码：

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

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

3.使用generic

restframework给我们提供了一组混合的generic视图，可以使我们的代码 大大简化。

GenericAPIView：继承自APIView，增加了对列表视图或者详情视图可能用到的通用支持方法。通常使用时，可搭配一个或者多个Mixin扩展类。支持定义的属性：

- 列表视图与详情视图通用：
  - queryset（视图的查询集），
  - serializer_class（视图使用的序列化器）
- 列表视图专用：
  - pagination_classes（分页控制类）
  - filter_backends（过滤控制）
- 详情页视图专用：
  - 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： 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，默认返回serializer_class，可以重写:

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

- get_serializer: 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： 根据传入的查询参数（lookup_url_kwarg or lookup_field参数)获取查询对象，然后返回， 一般进行联合查询时， 需要重写此方法。

  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)
```

列表视图专用：

- paginate_queryset： 进行分页，返回分页后的单页结果集

提供了5个扩展类：

- generics.ListAPIView
- generics.CreateAPIView
- generic.RetrieveAPIView
- generic.UpdateAPIView
- generic.DestroyAPIView

四、ViewSet

使用视图集ViewSet，可以将一系列逻辑相关的动作放到一个类中：

- list() 提供一组数据
- retrieve() 提供单个数据
- create() 创建数据
- update() 保存数据
- destory() 删除数据

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

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

```
class BookInfoViewSet(viewsets.ViewSet):

    def list(self, request):
        ...

    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'})
]
```

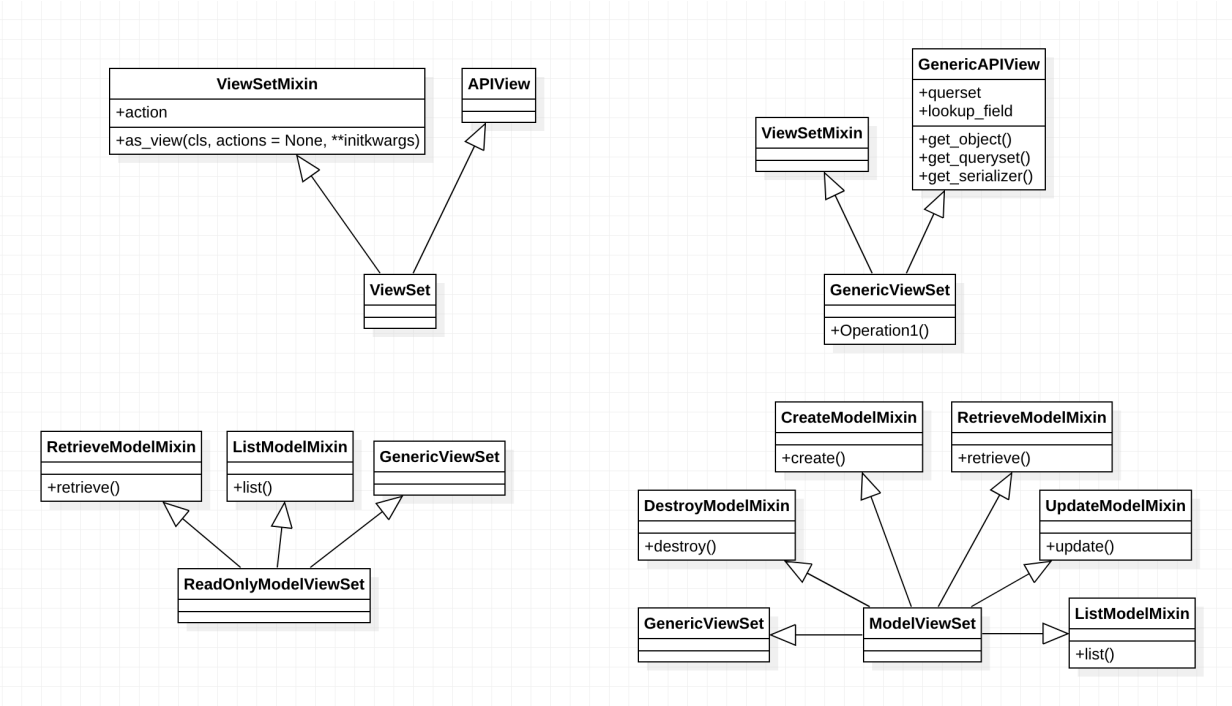
action属性

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
```

一个ViewSet 类当被实例化成一组视图时, 通常会通过使用一个路由类(Router class)来帮你处理复杂的URL定义，最终绑定到一组处理方法。



路由Routers

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

REST framework提供了两个router

- **SimpleRouter**
- **DefaultRouter**

1. 使用方法

1） 创建router对象，并注册视图集，例如

```
from rest_framework import routers

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

register(prefix, viewset, base_name)

- prefix 该视图集的路由前缀
- viewset 视图集
- base_name 路由名称的前缀

如上述代码会形成的路由如下：

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

2）添加路由数据

可以有两种方式：

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

或

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

2. 视图集中包含附加action的

```
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
```