1. 视图

Django REST framwork 提供的视图的主要作用：

- 控制序列化器的执行（检验、保存、转换数据）
- 控制数据库查询的执行

1.1. 请求与响应

1.1.1 Request

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

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

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。

1.1.1.1 常用属性

1）.data

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

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

2）.query_params

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

1.1.2 Response

```
rest_framework.response.Response
```

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

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

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

1.1.2.1 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。

1.1.2.2 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

1.1.2.3 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
```

1.2 视图

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。

1.2.1 2个视图基类

1.2.1.1 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)
```

1.2.1.2 GenericAPIView[通用视图类]

```
rest_framework.generics.GenericAPIView
```

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

提供的关于序列化器使用的属性与方法

- 属性：
  - **serializer_class** 指明视图使用的序列化器
- 方法：
  - **get_serializer_class(self)**

    当出现一个视图类中调用多个序列化器时,那么可以通过条件判断在get_serializer_class方法中通过返回不同的序列化器类名就可以让视图方法执行不同的序列化器对象了。

    返回序列化器类，默认返回`serializer_class`，可以重写，例如：

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

  - **get_serializer(self, *args, *kwargs)**

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

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

    - **request** 当前视图的请求对象
    - **view** 当前请求的类视图对象
    - format 当前请求期望返回的数据格式

提供的关于数据库查询的属性与方法

- 属性：
  - **queryset** 指明使用的数据查询集
- 方法：
  - **get_queryset(self)**

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

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

- **get_object(self)**

  返回详情视图所需的模型类数据对象，主要用来提供给Mixin扩展类使用。

  在试图中可以调用该方法获取详情信息的模型类对象。

  若详情访问的模型类对象不存在，会返回404。

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

  举例：

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

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

其他可以设置的属性

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

1.2.2 5个视图扩展类

作用：

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

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

1）ListModelMixin

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

该Mixin的list方法会对数据进行过滤和分页。

源代码：

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

举例:

```
from rest_framework.mixins import ListModelMixin

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

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

2）CreateModelMixin

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

如果序列化器对前端发送的数据验证失败，返回400错误。

源代码:

```
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()
```

1.2.3 几个子类视图

1）CreateAPIView

提供 post 方法

继承自： GenericAPIView、CreateModelMixin

2）ListAPIView

提供 get 方法

继承自：GenericAPIView、ListModelMixin

3）RetrieveAPIView

提供 get 方法

继承自: GenericAPIView、RetrieveModelMixin

4）DestoryAPIView

提供 delete 方法

继承自：GenericAPIView、DestoryModelMixin

5）UpdateAPIView

提供 put 和 patch 方法

继承自：GenericAPIView、UpdateModelMixin

6）RetrieveUpdateAPIView

提供 get、put、patch方法

继承自： GenericAPIView、RetrieveModelMixin、UpdateModelMixin

7）RetrieveUpdateDestoryAPIView

提供 get、put、patch、delete方法

继承自：GenericAPIView、RetrieveModelMixin、UpdateModelMixin、DestoryModelMixin

1.3 视图集ViewSet

使用视图集ViewSet，可以将一系列逻辑相关的动作放到一个类中：

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

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

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

```
class BookInfoViewSet(viewsets.ViewSet):

    def list(self, request):
```

```
        books = BookInfo.objects.all()
        serializer = BookInfoSerializer(books, many=True)
        return Response(serializer.data)

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

在设置路由时，我们可以如下操作

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

1.3.1 常用视图集父类

1） ViewSet

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

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

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

2）GenericViewSet

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

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

举例：

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

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

url的定义

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

3）ModelViewSet

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

4）ReadOnlyModelViewSet

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

1.3.2 视图集中定义附加action动作

在视图集中，除了上述默认的方法动作外，还可以添加自定义动作。

举例：

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

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

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

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

1.3.3 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：

```
from rest_framework.viewsets import ModelViewSet,ReadOnlyModelViewSet
from booktest.models import BookInfo
from .serializers import BookInfoModelSerializer
from rest_framework.response import Response
class BookInfoModelViewSet(ModelViewSet):
    queryset = BookInfo.objects.all()
    serializer_class = BookInfoModelSerializer

    def get_top_5(self,request):
        """获取评论值最多的5条数据"""
        # 操作数据库
        print(self.action) # 获取本次请求的视图方法名
```

通过路由访问到当前方法中.可以看到本次的action就是请求的方法名

2. 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**

2.1 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))
]
```

2.2 视图集中附加action的声明

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

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

action装饰器可以接收两个参数：

- **methods**: 声明该action对应的请求方式，列表传递
- detail

  : 声明该action的路径是否与单一资源对应，及是否是

  ```
  xxx/<pk>/action方法名/
  ```

  - True 表示路径格式是`xxx/<pk>/action方法名/`

- False 表示路径格式是`xxx/action方法名/`

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

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

由路由器自动为此视图集自定义action方法形成的路由会是如下内容：

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

2.3 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。