🡪Django目前发展的比较好的REST框架为Django REST FRAMEWORK.可用Django版本为：1.11, 2.0, 2.1, Python版本的要求为2.7,3.4,3.6,3.7. 并且DRF（实际就是python包）有自己的版本：例如3.11.0.

DRF的一个好处是，在后端中采用DRF，提供API方式的数据支持，在前端便可以采用任意的框架, 框架并不局限于web框架，指的更是手机APP等应用。不同的终端，有不同的渲染技术，但是数据只有一个，所以RESEFUL API这样不局限于前台，返回数据提交数据，达到可以随意切换前台框架。达到前后台完全分离隔绝，使后台稳定。

测试DRF的工具有许多，可以直接浏览器访问，使用curl命令行，使用postman工具等。

🡪

DRF框架提供Serializer支持（例如ModelSerializer基类, 可以重写create，update方法）,用于支持序列化反序列化。

DRF框架提供View支持（例如APIVIEW基类，generics.ListCreateAPIView, Mixins基类， 可以重写get, post方法）。用于处理不同的restful请求。

对于一个Restful POST请求，我们先用view接受该请求，然后在view中使用serializer来验证数据更新数据库。

对于一个Restful GET请求，我们先用view接受该请求，然后在view中从数据库取出数据，使用serializer来序列化结果返回结果。

🡪

使用DRF的时候，一般和网页处理函数分开，更易于管理。可以在根urls.py中对Restful请求开头的所有url匹配放入对应app下的自身的urls.py中，并且匹配时候到对应app的单独views.py中。

🡪

Django Rest Framework提供特别的Request对象，其中的request数据，使用request.data来获取。Request.data可以处理来自于GET, POST, PUT请求的所有数据。

Django Rest Framework提供特别的Response对象，它接受还没有渲染的数据，并决定返回什么数据类型给client.

Django Rest Framework提供特别的Status模块，用于提供更明确的状态码。

Django Rest Framework提供三种特别的Wrapper，这些Wrapper将保证在处理函数中存在Request对象(这个Reques对象不是普通的Django下的HTTPRequest对象，而是专门针对DRF的Request对象) ，并且为返回的Response对象添加context用于Response返回内容类型时候的协商（这个Response对象也不是是普通的Django下的HttpResponse对象，而是专门针对DRF的HttpResponse对象）。（我们通过这些Wrapper去Wrap我们的处理函数，从而提供一些默认的功能。这是推荐的做法。从本质来说，我们也可直接写函数在views当中，但是不被任何wrapper, 也可以达到目的，但是此时很多其他的情况都需要考虑）：

第一个是@api\_view,用于装饰函数,默认只接受GET方法，POST等方法需要显示指明。

第二种是APIView基类。使用情况为完全自主定义行为

第三种是generics.\*基类，使用情况默认实现部分CRUD操作选择的方式。这些Generic类继承APIView, 并且扩充其方法。我们可以根据需要选择不同的Generic类，也可以在选择的基础上再去override一些方法。

第三是Viewsets基类，在APIVIEW的基础上，默认的提供`list`, `create`, `retrieve`,

`update` and `destroy`方法。也就是默认实现CRUD方法。在接下来的urls.py中将list, create, retrieve, update, destroy方法绑定到http的get, post, patch, delete请求即可。

且经常和Router类一起用。当使用了Router类以后，我们不需要在urls.py中显示的写出什么路径对应什么函数，DRF会根据我们在router传入的viewsets, 直接自动的构建出对应的url地址。这样的好处是所有自动构建的url地址都是由DRF自动生成的，所以具有非常高的统一性。但是不好的地方是，没有我们手动指定的语义明显。

🡪

我们可以在发送http请求中:

***Accept字段***：告诉服务器，我们需要什么类型的数据返回类型。

***Content-type字段***：告诉服务器，我们放在http请求中的携带数据类型。

主要的媒体格式类型如下：

* text/html ： HTML格式
* text/plain ：纯文本格式
* text/xml ： XML格式
* image/gif ：gif图片格式
* image/jpeg ：jpg图片格式
* image/png：png图片格式

   以application开头的媒体格式类型：

* application/xhtml+xml ：XHTML格式
* application/xml     ： XML数据格式
* application/atom+xml ：Atom XML聚合格式
* application/json    ： JSON数据格式
* application/pdf       ：pdf格式
* application/msword ： Word文档格式
* application/octet-stream ： 二进制流数据（如常见的文件下载）
* application/x-www-form-urlencoded ： <form encType=””>中默认的encType，form表单数据被编码为key/value格式发送到服务器（表单默认的提交数据的格式）

   另外一种常见的媒体格式是上传文件之时使用的：

* multipart/form-data ： 需要在表单中进行文件上传时，就需要使用该格式

当客户端发起请求，并且没有特别指定Accept字段时候，Django Rest Framework默认会返回HTML格式的数据。

🡪

面向资源是REST最明显的特征，对于同一个资源的一组不同的操作。资源是服务器上一个可命名的抽象概念，资源是以名词为核心来组织的，首先关注的是名词。REST要求，必须通过统一的接口来对资源执行各种操作。对于每个资源只能执行一组有限的操作。（7个HTTP方法：GET/POST/PUT/DELETE/PATCH/HEAD/OPTIONS）

并且REST架构风格并不是绑定在HTTP上，只不过目前HTTP是唯一与REST相关的实例

🡪

幂等与非幂等：

等幂性简单点说就是一次请求和多次请求，资源的状态是一样。比如GET和HEAD，不论你请求多少次，资源还是在那里。请注意，DELETE和PUT也是等幂的，以为对同一个资源删除一次或者多次，结果是一样的，就是资源被删除了，不存在了。为什么说PUT也是等幂的？当你PUT一个新资源的时候，资源被创建，再次PUT这个URI的时候，资源还是没变。当你PUT一个存在的资源时，更新了资源，再次PUT的时候，还是更新成这个样子。

我们设计Restful WS的时候，GET，HEAD, PUT, DELETE一定要设计成等幂的。由于网络是不可靠的，安全性和等幂性就显得特别重要。如果一次请求，服务器收到处理以后，客户端没有收到相应，客户端会再次请求，如果没有等幂性保障，就会发生意想不到的问题。

POST是不安全也不等幂的，还是拿weblog的例子，如果两次POST相同的博文，则会产生两个资源，URI可能是这样/weblogs/myweblog/entries/1和/weblogs/myweblog/entries/2，尽管他们的内容是一样的

|  |  |
| --- | --- |
| Verd | 描述 |
| HEAD（SELECT） | 只获取某个资源的头部信息，幂等 |
| GET（SELECT） | 获取资源，幂等 |
| POST（CREATE） | 创建资源，非幂等 |
| PATCH（UPDATE） | 更新资源的部分属性，幂等 |
| PUT（UPDATE） | 更新资源，客户端需要提供新建资源的所有属性，幂等 |
| OPTIONS | 用于url验证，显示接口信息查询服务信息，幂等 |
| DELETE（DELETE） | 删除资源，幂等 |

在HTTP报文构成中，有个字段很重要：status code。它说明请求的大致情况，是否正常处理、出现了什么错误等。状态码都是三位数，大概分为了一下几个区间：

|  |  |
| --- | --- |
| 状态码 | 描述 |
| 2XX | 请求正常处理并返回 |
| 3XX | 重定向，请求的资源位置发生变化 |
| 4XX | 客户端发送的请求有误 |
| 5XX | 服务器端的错误 |

🡪

generic.ListCreateAPIView基类，展示实例或者创建。支持GET和POST请求。

generic.RetrieveDestroyAPIView基类，获取单独实例或者删除他们。支持GET和DELETE请求。

generic.CreateAPIView基类，创建实例。支持POST请求。

--重写类中的get\_queryset方法可以为返回的结果添加筛选条件

--重写类中的post方法可以修改当接受POST请求时的行为

🡪

DRF提供多种Resful服务的认证服务。包括最流行的token认证，先使用账号密码登录然后获取token, 使用该token进行其他restful请求。

🡪

相比于普通的Django Framework, DRF需要url, view和序列器。url用于控制API的终点站在哪，view定义到数据库取数据的逻辑，序列器用于将我们需要的数据变为JSON格式进行传输。

🡪

当我们引入DRF到现有project的时候，我们可以为每个原本的现有app添加serializers.py, 在views.py中添加函数。但是最好的做法是，新建一个app, 这个app专门管理DRF, 在这个新建的app中一样可以读取其他apps的models, 为其指定序列器，views等。最后在这个app中采用router + viewsets的方式进行抽象。

🡪

在urls.py定义中format\_suffix\_patterns用于为url增加后缀，

例如增加以后，将从<http://example.com/api/items/4.json>也会被接受，而不止<http://example.com/api/items/4>

🡪

Class-based与function-based的对于GET, POST方法的不同。

如果是function-based的函数，在处理Request的不同请求方法时为：

if request.method == 'GET': -🡪 通过判断request.method的值

snippets = Snippet.objects.all()

serializer = SnippetSerializer(snippets, many=True)

return Response(serializer.data)

elif request.method == 'POST': -🡪 通过判断request.method的值

serializer = SnippetSerializer(data=request.data)

if serializer.is\_valid():

serializer.save()

如果是class-based的函数时，在处理Request的不同请求方法时为：

class ListUsers(APIView):

def get(self, request, format=None): -🡪 通过关键函数名GET, POST, PUT等

"""

Return a list of all users.

"""

usernames = [user.username for user in User.objects.all()]

return Response(usernames)

🡪

DRF中的browserable API需要用的JS以及css文件的支撑。这些文件在通过pip安装的DRF中都已经自己包含，不需要用户执行下载。我们要做的是确保我们的服务器可以serve到这些静态文件。

🡪

我们在views中所读到的requests.data是不可变的，所以无法直接通过requests.data[‘field\_name’] = “…” 去为requests.data增加或者改变值。

🡪

在对DRF进行PUT方向执行serilizers的update方法的时候，默认情况下，不需要提供所有modesl.py中定义的所有数据域，但是必须提供modesl.py中不允许为空的数据域，并且update默认改变提交数据中所包含的数据域，不改变其他数据域。

🡪

RESTful API规范：

GET： 从服务器取出资源

POST: 从服务器新建一个资源

PUT： 在服务器更新资源（客户端提供改变后的完整资源），提交更新的时候是提交完整的对象。

PATCH： 在服务器更新资源（客户端提供改变的属性），提交更新的时候只提交改变的数据域。

DELETE： 在服务器删除资源

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

🡪

DRF中的views或者serilizers改变是需要重启服务器进行生效的。

🡪

DRF的配置在settings.py中会专门以REST\_FRAMEWORK的字段，并且以字典的形式进行配置。其他的配置包括DRF的访问权限，分页等。

🡪

Mixins的用法。DRF提供了Mixin类，这个类中提供了许多REST中常用的操作，例如list展示所有存在数据库中的对象，create创建对象，delete删除对象等。

例子：

如果不采用mixins类，只采用APIView作为基类，则要列出所有记录时，通常会这样写：

class PatchTrackingList(APIView):  
 """  
 List all PatchTrackingList, or create a new PatchTracking Record.  
 """  
 def get(self, request, format=None):  
 patch\_tracking\_objs = rgr\_patch\_tracking\_tbl.objects.all()  
 serializer = PatchTrackingSerializer(patch\_tracking\_objs, many=True)  
 return Response(serializer.data)  
  
 def post(self, request, format=None):  
 serializer = PatchTrackingSerializer(data=request.data)  
 if serializer.is\_valid():  
 serializer.save()  
 return Response(serializer.data, status=status.HTTP\_201\_CREATED)  
 return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

这是列出所有PatchTrackingList的view函数，同样的对于其他需要操作的表执行的操作都相同，在get方法中都是先找到所有对象，然后序列化。对于post都是先给序列器加入数据，验证是否合法，然后调用序列器的save()方法。基于这样重复的行为，DRF在mixins中添加许多子类，用于对应这些常用的操作。

Mixins.ListModelMixin 🡪 提供list方法，用于展示所有的对象。

Mixins.CreateModelMixin 🡪 提供create方法，用于创建和保存一个新的对象。

Mixins.RetrieveModelMixin 🡪 用于查询一个在数据库中的对象。

Mixins.UpdateModelMixin 🡪 用于更新对象，更新的对象包括PUT更新，以及只上传部分更新域的PATCH更新。

Mixins.DestroyModelMixin 🡪 用于删除一个对象。

所以上面采用APIView的view函数可以改为：

class PatchTrackingList(mixins.ListModelMixin, 🡪表示可用于列出所有对象  
 mixins.CreateModelMixin, 🡪表示可以用于创建对象  
 generics.GenericAPIView): 🡪加入通用函数  
 """  
 List all PatchTrackingList, or create a new PatchTracking Record.  
 """  
 queryset = rgr\_patch\_tracking\_tbl.objects.all()🡪为queryset提供重写值  
 serializer\_class = PatchTrackingSerializer 🡪采用什么样的序列器  
  
 def get(self, request, \*args, \*\*kwargs):  
 return self.list(request, \*args, \*\*kwargs)  
  
 def post(self, request, \*args, \*\*kwargs):  
 return self.create(request, \*args, \*\*kwargs)

同样的我们可以改写针对每个对象的view方法：

原来：

class PatchTrackingDetail(APIView):  
 """  
 Retrieve, update or delete a rgr\_patch\_tracking\_tbl instance.  
 """  
 def get\_object(self, build\_slug):  
 try:  
 return rgr\_patch\_tracking\_tbl.objects.get(build\_\_slug=build\_slug)  
 except rgr\_patch\_tracking\_tbl.DoesNotExist:  
 raise Http404  
  
 def get(self, request, build\_slug, format=None):  
 patch\_tracking\_obj = self.get\_object(build\_slug)  
 serializer = PatchTrackingSerializer(patch\_tracking\_obj)  
 return Response(serializer.data)  
  
 def put(self, request, pk, format=None):  
 patch\_tracking\_obj = self.get\_object(pk)  
 serializer = PatchTrackingSerializer(patch\_tracking\_obj, data=request.data)  
 if serializer.is\_valid():  
 serializer.save()  
 return Response(serializer.data)  
 return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)  
  
 def patch(self, request, pk, format=None):  
 patch\_tracking\_obj = self.get\_object(pk)  
 serializer = PatchTrackingSerializer(patch\_tracking\_obj, data=request.data, partial=True)  
 if serializer.is\_valid():  
 serializer.save()  
 return Response(serializer.data)  
 return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)  
  
 def delete(self, request, pk, format=None):  
 patch\_tracking\_obj = self.get\_object(pk)  
 patch\_tracking\_obj.delete()  
 return Response(status=status.HTTP\_204\_NO\_CONTENT)

使用Mixins:

class PatchTrackingDetail(mixins.RetrieveModelMixin,🡪处理取得某个特定的object  
 mixins.UpdateModelMixin, ->更新某个特定的object  
 mixins.DestroyModelMixin, 🡪删除某个特定的object  
 generics.GenericAPIView): 🡪提供通用功能  
 """  
 Retrieve, update or delete a rgr\_patch\_tracking\_tbl instance.  
 """  
 queryset = rgr\_patch\_tracking\_tbl.objects.all()🡪为queryset提供重写值  
 serializer\_class = PatchTrackingSerializer 🡪 选择序列器  
  
 def get(self, request, \*args, \*\*kwargs):  
 return self.retrieve(request, \*args, \*\*kwargs)  
  
 def put(self, request, \*args, \*\*kwargs):  
 return self.update(request, \*args, \*\*kwargs)  
  
 def delete(self, request, \*args, \*\*kwargs):  
 return self.destroy(request, \*args, \*\*kwargs)

🡪

使用抽象程度更高的Generic类

我们通过使用Mixins,进行一定程度的抽象，view变为：

class PatchTrackingDetail(mixins.RetrieveModelMixin,🡪处理取得某个特定的object  
 mixins.UpdateModelMixin, ->更新某个特定的object  
 mixins.DestroyModelMixin, 🡪删除某个特定的object  
 generics.GenericAPIView): 🡪提供通用功能,这个是必须的因为RetrieveModelMixin,里面调用的方法retrieve里面使用get\_object方法，如果在我们不重写的情况下，使用的便是GenericAPIView里面的get\_object方法。  
 """  
 Retrieve, update or delete a rgr\_patch\_tracking\_tbl instance.  
 """  
 queryset = rgr\_patch\_tracking\_tbl.objects.all()🡪为queryset提供重写值  
 serializer\_class = PatchTrackingSerializer 🡪 选择序列器  
  
 def get(self, request, \*args, \*\*kwargs):  
 return self.retrieve(request, \*args, \*\*kwargs)  
  
 def put(self, request, \*args, \*\*kwargs):  
 return self.update(request, \*args, \*\*kwargs)  
  
 def delete(self, request, \*args, \*\*kwargs):  
 return self.destroy(request, \*args, \*\*kwargs)

这样的确比用APIView简单，但是对于每一个view都要输入mixins.xxx, mixins,xxx.所以DRF在Generics中提供的某些子类，可以直接让Generic的子类，直接继承这写Mixins类。

改写为：

class SnippetDetail(generics.RetrieveUpdateDestroyAPIView): 🡪这个类包含了mixins.Retrieve,

🡪mixins.Update, mixins.Destory,

🡪以及通用函数

queryset = Snippet.objects.all()

serializer\_class = SnippetSerializer

🡪

genericView中的RetrieveUpdateDestroyAPIView类，里面包含了使的对于单一的对象，可以进行查询，更新，删除。但是默认的get\_object行为是根据pk来查询，进而再进行更新等操作的。如果我们想根据其他域来查找特定的object时候，

我们可以在views.py中重写他的get\_object方法，那么其余操作会因为我们重写的这个方法，在寻找特定object的时候，采用不同于pk的查询方式。

--PrimaryKeyRelatedField

class AlbumSerializer(serializers.ModelSerializer):

tracks = serializers.PrimaryKeyRelatedField(many=True, read\_only=True) –>Album在models.py中没有这个域，这里额外定义一个域，并且read\_only=True,只会在序列化时显示，在反序列化时不会添加到Album的记录中。

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

Would serialize to a representation like this:

{

'album\_name': 'Undun',

'artist': 'The Roots',

'tracks': [

89,

90,

91,

...

]

}

默认情况下，域是read-write。

--HyperlinkedRelatedField

HyperlinkedRelatedField用超链接表示目标。

class AlbumSerializer(serializers.ModelSerializer):

tracks = serializers.HyperlinkedRelatedField(

many=True,

read\_only=True,

view\_name='track-detail'

)

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

Would serialize to a representation like this:

{

'album\_name': 'Graceland',

'artist': 'Paul Simon',

'tracks': [

'http://www.example.com/api/tracks/45/', 🡪以超链接的形式表示

'http://www.example.com/api/tracks/46/',

'http://www.example.com/api/tracks/47/',

...

]

}

--SlugRelatedField

SlugRelatedFile用于表示关系目标中的某个域

class AlbumSerializer(serializers.ModelSerializer):

tracks = serializers.SlugRelatedField(

many=True,

read\_only=True,

slug\_field='title' 🡪表示的是Track表中的title域

)

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

序列化为

{

'album\_name': 'Dear John',

'artist': 'Loney Dear',

'tracks': [

'Airport Surroundings',

'Everything Turns to You',

'I Was Only Going Out',

...

]

}

默认为read-write,***通常对应表中的域为unique=True的域***，并且必须带有参数slug\_field指定为表中的哪个域）。并且必须显示指定queryset或者指定read\_only=True

🡪

DRF中使用nested来表示关系，nested的关系意思为在序列器中的某个域是另一个序列器，而不是对应某个一个域，

例如：

class TrackSerializer(serializers.ModelSerializer):

class Meta:

model = Track

fields = ('order', 'title', 'duration')

class AlbumSerializer(serializers.ModelSerializer):

tracks = TrackSerializer(many=True, read\_only=True) 🡪这个序列器中的域，是另一个序列器

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

那么序列化时将变为：

{

'album\_name': 'The Grey Album',

'artist': 'Danger Mouse',

'tracks': [

{'order': 1, 'title': 'Public Service Announcement', 'duration': 245}, 🡪表示为整个对象

{'order': 2, 'title': 'What More Can I Say', 'duration': 264},

{'order': 3, 'title': 'Encore', 'duration': 159},

...

],

}

需要注意的时默认情况下，这种nested的域是read-only的。如果要支持写操作，那么我们必须显示的重写序列器中的create和udpate方法。例如对album的序列器：

class AlbumSerializer(serializers.ModelSerializer):

tracks = TrackSerializer(many=True)

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

def create(self, validated\_data): 🡪显式的重写create方法

tracks\_data = validated\_data.pop('tracks')

album = Album.objects.create(\*\*validated\_data) 🡪先创建album方法

for track\_data in tracks\_data:

Track.objects.create(album=album, \*\*track\_data) 🡪再逐一创建track方法，并且指定连到album中

return album

🡪

DRF中常用的一般类型的数据域 (<https://www.django-rest-framework.org/api-guide/fields/>) ：

--BooleanField （对应django.db.models.fields.BooleanField）

--NullBooleanField (对应django.db.models.fields.NullBooleanField)

--CharField (对应django.db.models.fields.CharField or django.db.models.fields.TextField)

--EmailField (对应django.db.models.fields.EmailField)

--RegexField (对应django.forms.fields.RegexField， 必须包含regex参数)

…..

当我们自行定义了某个域以后，相同域重写了该域在序列器中的定义。所以通过序列器自动提醒该域的属性都将消失。而是完全依照我们重写的域的定义。

🡪

我们通过DRF去创建一个对象，而其中的一个域想为其提供默认值的时候。可以在两个地方申明，我们可以通过在Django的models.py中对该域进行default的申明，也可以在序列器中对该域进行default的申明。都可以达到为该域提供默认值。并且由于这样，我们可以设置成当在Django中建立的时候，提供的Django的缺省值，在DRF REST中建立时候，提供DRF的缺省值。

🡪

改写显示外键域的方式。在默认的情况下，如果我们的models.py中有一个域是外键，那么在api返回中，在Web的browserable API中，返回的信息显示的将直接是外键所对象的对象的pk.原因时DRF默认将外键，使用DRF关系域中的PrimaryKeyRelatedField.

例如：

{

"id": 4,

"tracking\_history": "2019-01-02 16:32:08: Create Patch Tracking Record\r\n2019-01-03 14:17:23: All patch runs are finished sucessfully!\n2019-01-03 15:17:23: All patch runs are validated, patch validation done",

"comment": "Hui: Test patch tracking for NFMP\_18\_10\_SP2.",

"build": 10146

}

这个build显示的是外键的id,但是我们经常需要的不是要对象的id,要的是对象里面具有特征的某个值。所以需要改些，改写的方式有两种：

第一种是在Serilizers.py中选择不同的DRF的关系域，例如serializers.SlugRelatedField

build = serializers.SlugRelatedField(slug\_field="slug", queryset=rgr\_build\_tbl.objects.all())

slug\_field：是被引用表的unique=True性质的域

通过这样指定以后，build在DRF序列化时将以目标表，也就是所引用的表域来显示，同时在反序列化的时候也以提供slug，用来查找目标表中该slug域为该值的条目。

第二中是自行创建一个RelationField

例如：

class BuildField(serializers.RelatedField):  
  
 def to\_internal\_value(self, data):  
 return rgr\_build\_tbl.objects.get(slug=data)  
  
 def to\_representation(self, value):  
 return '%s' % value.slug  
  
build = BuildField(queryset=rgr\_build\_tbl.objects.all())

我们以serializers.RelatedField作为基类，根据自己要求自行创建一个表示关系域的Field. 要求是必须在这个类中，显式的定义to\_internal\_value以及to\_representation这两个方法。其中to\_internal\_value是用于反序列化，其中的data是unvalidate\_date中的build所指的值。而to\_represenation是用于序列化过程，value是被指用的对象，返回被引用对象的具体哪个域作为我们要显示的值。

🡪

DRF通常对URL需要添加两个额外的信息，一个头一个尾巴。意思是是头部将添加/api/的根目录，作用是展示出所有的可用api地址，称作api\_root. 一个是尾巴称作,suffix,即是满足,自动根据url判别需要的返回类型。

🡪

DRF虽然可以使用不同的Field去代表Relationships，但是本质是，无论如何表示，序列器必须有能力根据现有的Filed最后将变为数据库中的需要的更新对象。

例如外键关系，可以用SlugRelateField表示，也可以自定义表示，也可以用StringRelatedField表示，HyperLinkField表示,但是在Django ORM中外键域必须是被引用表的其中一个对象，所以序列器无论采用什么域，都必须最后反序列化传到数据库的时候，该域是一个引用表的其他中一个对像，最后也不能传字符串，pk作为该域的最终值。

🡪

DRF框架自带测试DRF RESTFUL API的包， APIRequestFactory, APIClient, APITestCase.大致流程为为每个测试用例设置user进行登录，然后指定进行的请求，参数以及期待的HTTPRESPONSE状态码。

🡪

Coreapi以及django-rest-swagger包可以为我们DRF API提供自动生成的文档。

🡪

Django Rest Framework中通过注册不同的Render可以提供不同的功能。

例如：

JSONRenderer将请求数据变为JSON对象，默认使用utf-8编码  
BrowsableAPIRenderer在网页上查阅API时候,拥有HTML样式。