一、 路由器

有一些像Rails这样的Web框架提供自动生成Urls的功能。但是Django并没有。

REST framework为Django添加了这一功能，以一种简单、快速、一致的方式。

DRF的routers模块没有什么东西，主要包含下面几个类：

- BaseRouter：路由的基类
- SimpleRouter： 继承了BaseRouter，常用类之一
- **DefaultRouter**：继承了SimpleRouter，常用类之一
- DynamicDetailRoute
- DynamicListRoute
- RenameRouterMethods
- APIRootView

我们主要使用的其实就是SimpleRouter和DefaultRouter。

基本用法

下面是一个简单的例子，这里继承了 `SimpleRouter` 类。

```
from rest_framework import  routers

router = routers.SimpleRouter()
router.register(r'users', UserViewSet,  basename='user')
router.register(r'accounts', AccountViewSet)
urlpatterns = router.urls
```

1. 导入routers模块
2. 实例化一个SimpleRouter对象router
3. 使用router的register方法注册两条路由模式
4. 将router的urls属性赋值给Django的基本路由变量urlpatterns

对于 `register()` 方法，有两个必填参数:

- `prefix` ： 用户视图集的URL路径的前缀
- `viewset` ： 对应的视图集类

另外还有一个可选，但又特别重要的参数：

- `basename` ：URL的name属性的基础部分。如果没有显式指定这个参数，那么将以视图集的 `queryset` 属性的值，自动生成。如果视图集没 `queryset` 属性，那么你必须在有
  register方法里设置。

上面的例子将自动生成下面的URL模式：

- `users/` Name: `'user-list'`
- `users/<int:pk>/` Name: `'user-detail'`
- `accounts/` Name: `'account-list'`
- `accounts/<int:pk>/` Name: `'account-detail'`

`basename` 参数就是用于指定上面模式中，Name值对应的字符串的短横线的前面部分，也就是字符串 `user` 或者`account` 部分。

前面说了，通常我们不需要指定 `basename` 参数的值，但是如果你在视图集类中写了一个自定义的 `get_queryset` 方法，可能视图集就会缺少一个 `.queryset` 属性，这种情况下，如果你不指定 `basename` 参数的值，会导致下面的异常：

```
'basename' argument not specified, and could not automatically determine the
name from the viewset, as it does not have a '.queryset'  attribute.
```

在路由器的语法中使用 `include` 方法转发路由

其实，DRF路由器对象的urls属性，本质上是一个Django标准的URL模式对象，同样可以使用标准的Django路由语法和功能。

例如，你可以将 `router.urls` 追加到现有的urlpatterns里，也就是列表+列表的合并操作：

```
router = routers.SimpleRouter()
router.register(r'users', UserViewSet)
router.register(r'accounts',  AccountViewSet)

urlpatterns = [
    path('forgot-password/', ForgotPasswordFormView.as_view()),
]

# 注意这一行!
urlpatterns += router.urls
```

或者使用Django的 `include` 函数，如下所示：

```
urlpatterns = [
    path('forgot-password/', ForgotPasswordFormView.as view()),
    path('',  include(router.urls)),   # 转发到DRF的router
]
```

也可以提供一个app的命名空间参数：

```
urlpatterns = [
    path('forgot-password/', ForgotPasswordFormView.as_view()),
path('api/', include((router.urls, 'app_name'))), 4
]
```

甚至同时使用app和实例的命名空间：

```
urlpatterns = [
    path('forgot-password/', ForgotPasswordFormView.as_view()),
    path('api/', include((router.urls, 'app_name'),
namespace='instance_name')),
]
```

注意：如果使用了超链接序列化器时，使用命名空间要确保序列化器的 `view_name` 参数正确的映射到了命名空间。在上面的例子中，你就需要提供一个类似 `view_name='app_name:user-detail'` 的参数给序列化器的超链接字段，用于指向用户的详情视图。

默认情况下，自动生成的 `view_name` 属性是 `%(model_name)-detail` 这种格式的。

为额外的动作（actions）生成路由

DRF的视图集可以通过 `@action` 装饰器，为类的方法提供额外动作。路由器也会为这些动作自动生成URL模式。比如下面的例子：

```
from myapp.permissions import  IsAdminOrIsSelf
from rest_framework.decorators import action

class UserViewSet(ModelViewSet):
    ...
     # 参考视图集中action的帮助
    @action(methods=['post'], detail=True, permission_classes=
[IsAdminOrIsSelf])
    def set_password(self, request, pk=None):
        ...
```

自动生成的路由模式如下：

- `users/<int:pk>/set_password/` name: `'user-set-password'`

默认情况下，生成的URL模式以方法名为基础。而name参数的值则组合了 `ViewSet.basename` 和方法名，以短横线连接，比如 `'user-set-password'` 。

如果你不想用上面的生成规则，可以自己指定 `@action` 装饰器的 `url_path` 和 `url_name` 参数，如下所示：

```
from myapp.permissions import  IsAdminOrIsSelf
from rest_framework.decorators import action

class UserViewSet(ModelViewSet):
    ...
      # 注意参数名
    @action(methods=['post'], detail=True, permission_classes=
[IsAdminOrIsSelf],
            url_path='change-password', url_name='change_password')
    def set_password(self, request, pk=None):
        ...
```

然后，生成的URL模式就是这样的了：

- `users/<int:pk>/change-password/` name: `'user-change_password'`

二、API参考

SimpleRouter

SimpleRouter类包含标准的 `list`，`create`，`retrieve`，`update`，`partial_update` and `destroy` 这些操作所对应的url，以及 `@action` 装饰器带来的操作。 下表列出了对应的关系：

URL Style	HTTP Method	Action	URL Name
{prefix}/	GET	list	{basename}-list
{prefix}/	POST	create	{basename}-list
{prefix}/{url_path}/	GET, or as specified by `methods` argument	`@action(detail=False)` decorated method	{basename}-{url_name}
{prefix}/{lookup}/	GET	retrieve	{basename}-detail
{prefix}/{lookup}/	PUT	update	{basename}-detail
{prefix}/{lookup}/	PATCH	partial_update	{basename}-detail
{prefix}/{lookup}/	DELETE	destroy	{basename}-detail
{prefix}/{lookup}/{url_path}/	GET, or as specified by `methods` argument	`@action(detail=True)` decorated method	{basename}-{url_name}

默认情况下，`SimpleRouter` 将为每一条url添加一个斜杠后缀，这也是Django的通常做法。如果你不想这么做，可以在初始化的时候提供 `trailing_slash` 参数，并设置为 `False` ：

```
router = SimpleRouter(trailing_slash=False)
```

路由器将匹配包含除斜杠和句点字符之外的任何字符的匹配值。对于更严格（或宽松）的查找模式，请在视图集上设置 `lookup_value_regex` 属性。例如，可以将查找范围限制为有效的UUID：

```
class MyModelViewSet(mixins.RetrieveModelMixin, viewsets.GenericViewSet):
    lookup_field = 'my_model_id'
    lookup_value_regex = '[0-9a-f]{32}'
```

DefaultRouter

DefaultRouter类和 `SimpleRouter` 基本类似，不同之处在于它还会自动生成默认的API根视图的url路径。另外，它还可以为 `.json` 类型的请求生成对应的url

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}

同样的，也可以指定是否追加最后的那个斜杠：

```
1  router  = DefaultRouter(trailing_slash=False)
```

一、解析器

> 解析器是干什么的？因为前后端分离，因为可能采用json、xml、html等各种不同格式的内容，后端必须要有一个解析器来解析前端发送过来的数据，也就是翻译器！否则后端凭什么看懂前端的数据？对应地，后端也有一个渲染器Render，和解析器是相反的方向，将后端的数据翻译成前端能明白的数据格式。

REST框架提供了许多内置的Parser类，用来处理各种媒体类型的请求，比如json，比如xml。还支持自定义解析器，可以灵活地设计API接受的媒体类型。

Django原生的解析器对于post的数据，如果要从request.body中解析出来放到request.POST中，那么必须同时满足两个条件：

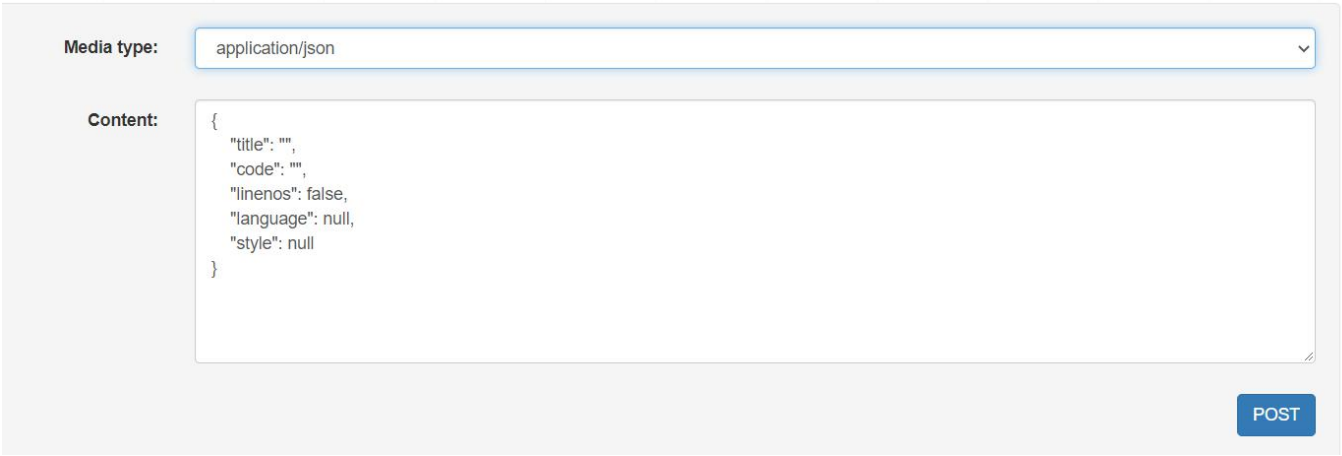
1. 请求头部 `Content_type = 'application/x-www-form-urlencoded'`
2. 数据格式必须是： `name=xxx&password=xxx&email=xxx.....`

而对于前端发送过来的例如JSON格式的数据则无法处理（当然你自己处理也是可以的）。DRF则不同，它提供了一些额外的解析器帮我们处理各种格式。

DRF的parsers模块非常简单，只定义了几个解析器类：

- BaseParser：解析器基类，以下四个类都直接继承它
- JSONParser
- FormParser
- MultiPartParser
- FileUploadParser

选择使用JSONParser解析器后，页面是只能提交json的数据



DRF在运行的时候如何知道该使用哪个解析器呢?

DRF将有效的解析器集定义为类的列表。当 `request.data` 被访问时，REST框架将检查请求头部的 `Content-Type` 属性，以此来确定要使用哪个解析器来解析数据。

所以，**要注意！解析器只有在请求**request.data**的时候才会被调用！**如果不需要data数据，那么就不用解析。

注意：在开发客户端应用程序时，务必确保在请求头部中包含 `Content-Type` 属性。如果未设置内容类型，则大多数客户端将默认使用 `'application/x-www-form-urlencoded'` 类型，但这可能不是你想要的。

例如，如果使用jQuery的ajax或者vue的axios方法发送 `json` 编码的数据，则应在请求头部中设置 `contentType: 'application/json'` 。

解析器的相关配置参数

可以在Django项目的settings.py文件中，使用 `DEFAULT_PARSER_CLASSES` 配置项，进行全局的解析器设置。例如，以下设置仅允许解析JSON格式的请求，而不是默认的JSON或表单数据：

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

当然，也可以同时支持多种解析器，比如下面的配置，**这也是DRF默认的解析器配置**：

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

几种解析器的写法没有先后顺序的要求，不像中间件那样的配置有顺序关系。

DRF支持视图级别的解析器，比如为基于APIView的类视图专门指定使用的解析器，核心是指定 `parser_classes` 属性为某个解析器：

```python
from rest_framework.parsers import  JSONParser
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    """
    A view that can accept POST requests with JSON content.
    """
    # 看这里
    parser_classes = (JSONParser,)

    def post(self, request, format=None):
        return Response({'received data':  request.data})
```

当然，也可以为使用 `@api_view` 装饰器改造的视图指定专用的解析器，核心是 `@parser_classes((JSONParser,))` 装饰器：

```python
from rest_framework.decorators import api_view
from rest_framework.decorators import parser_classes
from rest_framework.parsers import JSONParser

@api_view(['POST'])
@parser_classes((JSONParser,))   # 看这里
def example_view(request, format=None):
    """
    A view that can accept POST requests with JSON content.
    """
    return Response({'received data': request.data})
```

> 那么，问题来了！如果我在全局配置只允许JSON类型，但又在某个视图指定可以使用form表单类型，结果是什么？视图中的配置具有更高的优先级。

二、API参考

JSONParser

解析 `JSON` 格式的请求内容。

其.media_type属性值为 `application/json`

FormParser

解析HTML表单内容，使用QueryDict的数据填充request.data。这也是Django原生支持的解析方式。

通常我们希望同时支持FormParser和MultiPartParser两种解析器，以便完全支持HTML表单数据。

.media_type: `application/x-www-form-urlencoded`

MultiPartParser

解析多部分的HTML表单内容，支持文件上传。

.media_type: `multipart/form-data`

帮助：HTML的form表单的enctype属性规定了form表单在发送数据到服务器时的编码方式，有三种方式：

- application/x-www-form-urlencoded：默认的编码方式，常用于键值对数据。但是在用文本的传输和MP3等大型文件的时候，使用这种编码效率低下。
- multipart/form-data ： 指定传输数据为二进制类型，比如图片、mp3、文件。
- text/plain：纯文体的传输。空格转换为 “+” 加号，但不对特殊字符编码。使用较少。

这种情况下，接收二进制文件的例子：

```
class  FileUploadView(views.APIView):

    def post(self,request):
        print(request.body)
        with open(r'd:\temp', 'wb') as f:
            f.write(request.body)
        return  Response("200,ok")
```

FileUploadParser（项目里再看提交文件）

解析原始文件上传内容。此时，`request.data`属性将是一个字典，并且只包含一个键，这个键叫做`'file'`，对应的值包含上传的文件内容。

如果使用FileUploadParser解析器的视图，在被调用的时候URL中携带一个`filename`关键字参数，则该参数将被用作文件名。如果在没有这个关键字参数的情况下调用它，则客户端必须在HTTP头部的`Content-Disposition`中设置文件名。例如`Content-Disposition: attachment;filename=upload.jpg`。

.media_type: `*/*`

注意：

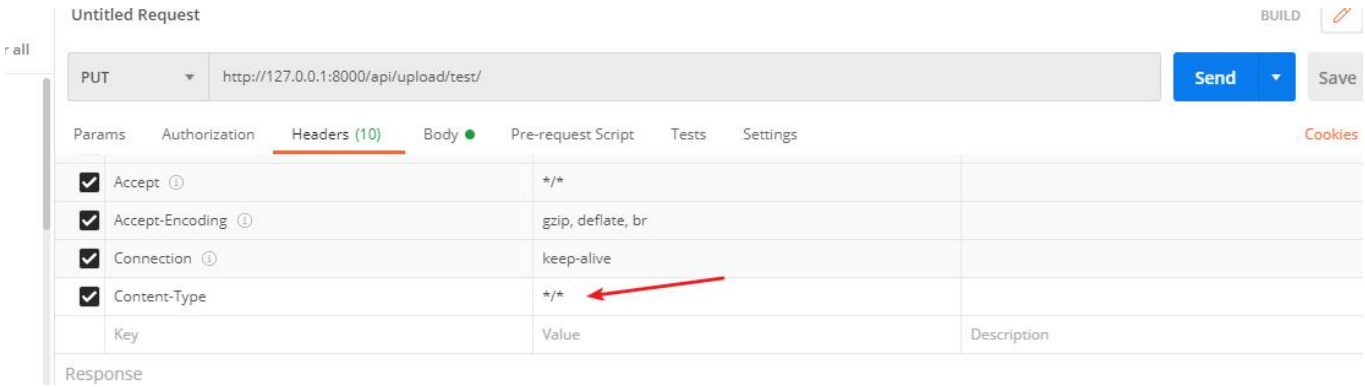
- `FileUploadParser`用于原生的文件上传请求。对于在浏览器中上传或者使用带有分段上传功能的客户端，请用`MultiPartParser`解析器。
- 由于`FileUploadParser`的`media_type`属性值是`*/*`，表示可以接受所有内容类型， 所以无需指定别的解析器，指定`FileUploadParser`就足够了。
- `FileUploadParser`遵守Django标准的`FILE_UPLOAD_HANDLERS`设置和`request.upload_handlers`属性。

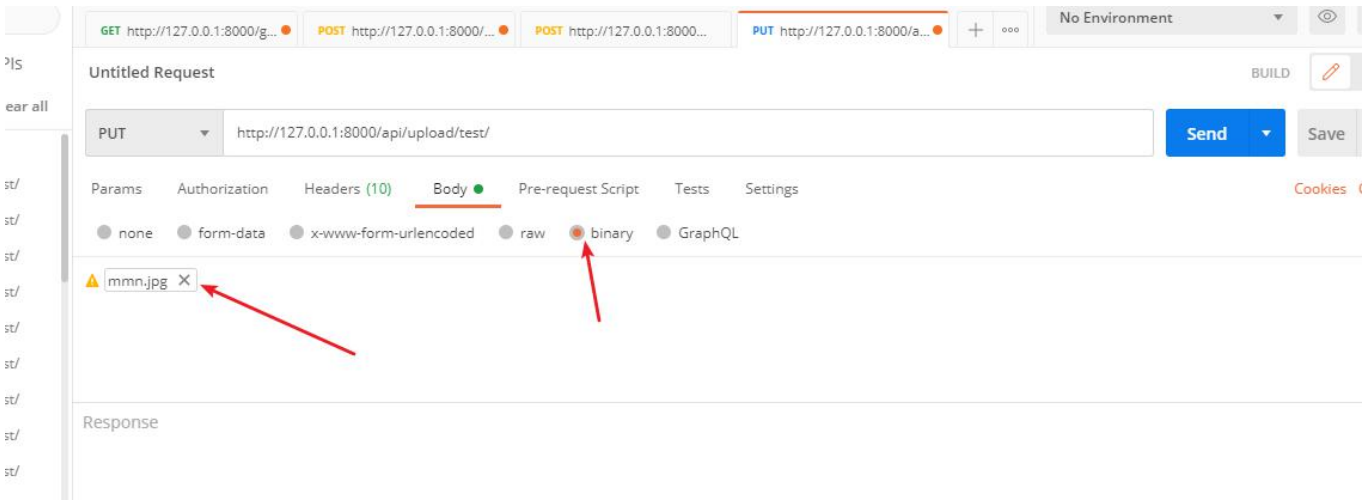
西面看一个具体的例子：

```
# views.py
class FileUploadView(views.APIView):
    parser_classes =  (FileUploadParser,)

    def put(self, request, filename, format=None):
        file_obj = request.data['file']
        # ...
        # 在这里处理你的文件内容
        # ...
        return Response(status=204)

# urls.py
urlpatterns = [
    # ...
    path('upload/<str:filename>/',  FileUploadView.as_view())
]
```





使用apifox

三、自定义解析器

要自定义解析器，必须继承 `BaseParser` 类，设置 `.media_type` 属性，并实现 `.parse(self, stream, media_type, parser_context)` 方法，该方法应返回将用于填充 `request.data` 属性的数据。

parse()方法的参数说明：

- stream

类似于流的对象，表示请求的主体。

- media_type

请求内容的媒体类型，可选

根据请求头部的 `Content-Type:` ，这可能比渲染器的 `media_type` 属性更具体，并且可能包括媒体类型参数。例如 `"text/plain;charset=utf-8"` 。

- parser_context

可选，字典格式。如果提供，将包含解析请求内容可能需要的任何其他上下文。

默认情况下，它包括以下的键： `view` ， `request` ， `args` ， `kwargs` 。

下面自定义了一个纯文本解析器，它将字符串形式表示的请求内容，填充到request.data属性。

```
class PlainTextParser(BaseParser):
    """
    纯文本解析器
    """
    media type = 'text/plain'

    def parse(self, stream, media_type=None, parser_context=None):
        """
        简单的返回一个请求主体内容的字符串形式
        """
        return stream.read()
```

四、第三方模块

下面这些第三方模块为DRF提供了额外的解析器。

YAML解析器

`djangorestframework-yaml` 包为我们提供了解析和渲染yaml格式的能力。它以前直接包含在REST框架包中，现在作为第三方包出现。

直接使用pip安装。

```
$ pip install djangorestframework-yaml
```

可以进行下面的配置：

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
    'rest_framework_yaml.parsers.YAMLParser',
    ),
    'DEFAULT_RENDERER_CLASSES': (
    'rest_framework_yaml.renderers.YAMLRenderer',
    ),
}
```

XML解析器

以前也是内置，现在也作为第三方包存在：

```
$ pip install djangorestframework-xml
```

配置方法：

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
    'rest_framework_xml.parsers.XMLParser',
    ),
    'DEFAULT_RENDERER_CLASSES': (
    'rest_framework_xml.renderers.XMLRenderer', ),
}
```

一、渲染器

所谓的渲染器（Renderer），其实就是将服务器生成的数据的格式转换为HTTP请求的格式。

REST框架包含许多内置的Renderer类，可以返回各种媒体类型的响应。还支持定义自定义渲染器，灵活地设计自己的媒体类型。

如何确定渲染器

在DRF配置参数中，可用的渲染器依然是作为一个类的列表进行定义。但与解析器不同的是，渲染器的列表是有顺序关系的。REST框架将对传入请求执行内容协商，根据请求的类型确定最合适的渲染器以满足类型要求。

内容协商过程会检查请求头部的 `Accept` **属性**，以确定客户期望的媒体类型。可选的，URL上的格式后缀可用于显式地请求特定的内容类型，例如 `http://example.com/api/users_count.json` 表明客户希望服务器返回JSON格式的数据。

配置渲染器的方法

使用 `DEFAULT_RENDERER_CLASSES` 设置全局渲染器集合。例如，以下设置将 `JSON` 用作主要媒体类型，也可以渲染可视化API，这也是DRF**的默认配置**。

```
1   REST_FRAMEWORK = {
2       'DEFAULT_RENDERER_CLASSES': (
3           'rest_framework.renderers.JSONRenderer',
4       'rest_framework.renderers.BrowsableAPIRenderer', 5  )
6   }
```

还可以为使用 `APIView` 基于类的视图，设置视图级别的，单独使用的渲染器。

```
from django.contrib.auth.models import User
from rest_framework.renderers import JSONRenderer
from rest_framework.response import Response
from rest_framework.views import APIView

```

```
class UserCountView(APIView):
    """
    A view that returns the count of active users in JSON.
    """
    # 注意这一行
    renderer_classes = (JSONRenderer, )

    def get(self, request, format=None):
        user_count =
        User.objects.filter(is_active=True).count() content =
        {'user_count': user_count}
```

或者对使用 `@api_view` 装饰器的视图，进行单独的渲染器指定。

```
@api_view(['GET'])
@renderer_classes((JSONRenderer,))      # 看这里!
def user_count_view(request, format=None):
    """
    A view that returns the count of active users in JSON.
    """
    user_count = User.objects.filter(active=True).count()
    content = {'user_count': user_count}
    return Response(content)
```

排序渲染器类的顺序

在为API指定渲染器类时，要考虑清楚为每种媒体类型分配什么样的优先级，这一点很重要。如果客户端未明确指定它可以接受的内容类型，例如发送 `Accept: */*` 属性值，或者根本没有 `Accept` 属性值，那么REST框架将选择settings列表中的第一个渲染器用于渲染响应内容。

如果你的API包含可根据请求的不同，同时提供常规网页和API响应的视图，请使用 `TemplateHTMLRenderer` 作为默认渲染器。

二、API参考

JSONRenderer

使用utf-8编码将响应内容渲染为json格式的数据。

默认样式包含unicode字符，并使用紧凑样式，没有不必要的空格，例如：

```
{"unicode  black star":"★","value":999}
```

如果客户端设置了 `'indent'` 缩进参数，返回的内容将缩进。例如 `Accept: application/json; indent=4` 。

```
{
    "unicode black star":  "★",
    "value": 999
}
```

可以使用 `UNICODE_JSON` 和 `COMPACT_JSON` 更改默认的JSON编码格式。

.media_type： `application/json`

.format： `'json'`

.charset： `None`

TemplateHTMLRenderer

使用Django的标准模板渲染器将数据渲染为HTML格式。与其他渲染器不同，此时传递给 `Response` 的数据不需要序列化。此外，可能还要使用 `template_name` 参数，指定你要渲染的HTML模板。

TemplateHTMLRenderer使用 `response.data` 作为上下文字典创建 `RequestContext` ，并确定要使用哪个模板。

模板的查询规则（按优先顺序）：

1. 显式指定的 `template_name` 参数。
2. 在TemplateHTMLRenderer上显式设置的 `.template_name` 属性。
3. `view.get_template_names()` 方法调用的返回结果。

下面是一个使用 `TemplateHTMLRenderer` 渲染器的示例：

```
class UserDetail(generics.RetrieveAPIView):
    """
    A view that returns a templated HTML representation of a given user.
    """
    queryset = User.objects.all()
    renderer_classes = (TemplateHTMLRenderer,)

    def get(self, request, *args, **kwargs):
        self.object =  self.get_object() return
        Response({'user':  self.object},
template name='user detail.html')
```

可以使用 `TemplateHTMLRenderer` 返回常规HTML页面，也可以返回API类型的响应。

如果你是与其他渲染器类一起混用的网站，则应将 `TemplateHTMLRenderer` 作为 `renderer_classes` 设置列表中的第一个类。

.media_type： `text/html`

.format： `'html'`

.charset： `utf-8`

StaticHTMLRenderer

一个简单的渲染器，只返回渲染好的HTML内容数据。与其他渲染器不同，传递给响应对象的数据应该是要返回内容的字符串形式。

例如：

```
@api_view(('GET',))
@renderer_classes((StaticHTMLRenderer,))
def simple_html_view(request):
    # 注意data变量是个字符串，虽然看起来像HTML代码
    data = '<html><body><h1>Hello, world</h1></body></html>'
    return Response(data)
```

既可以使用`utf-8icHTMLRenderer` .charset：

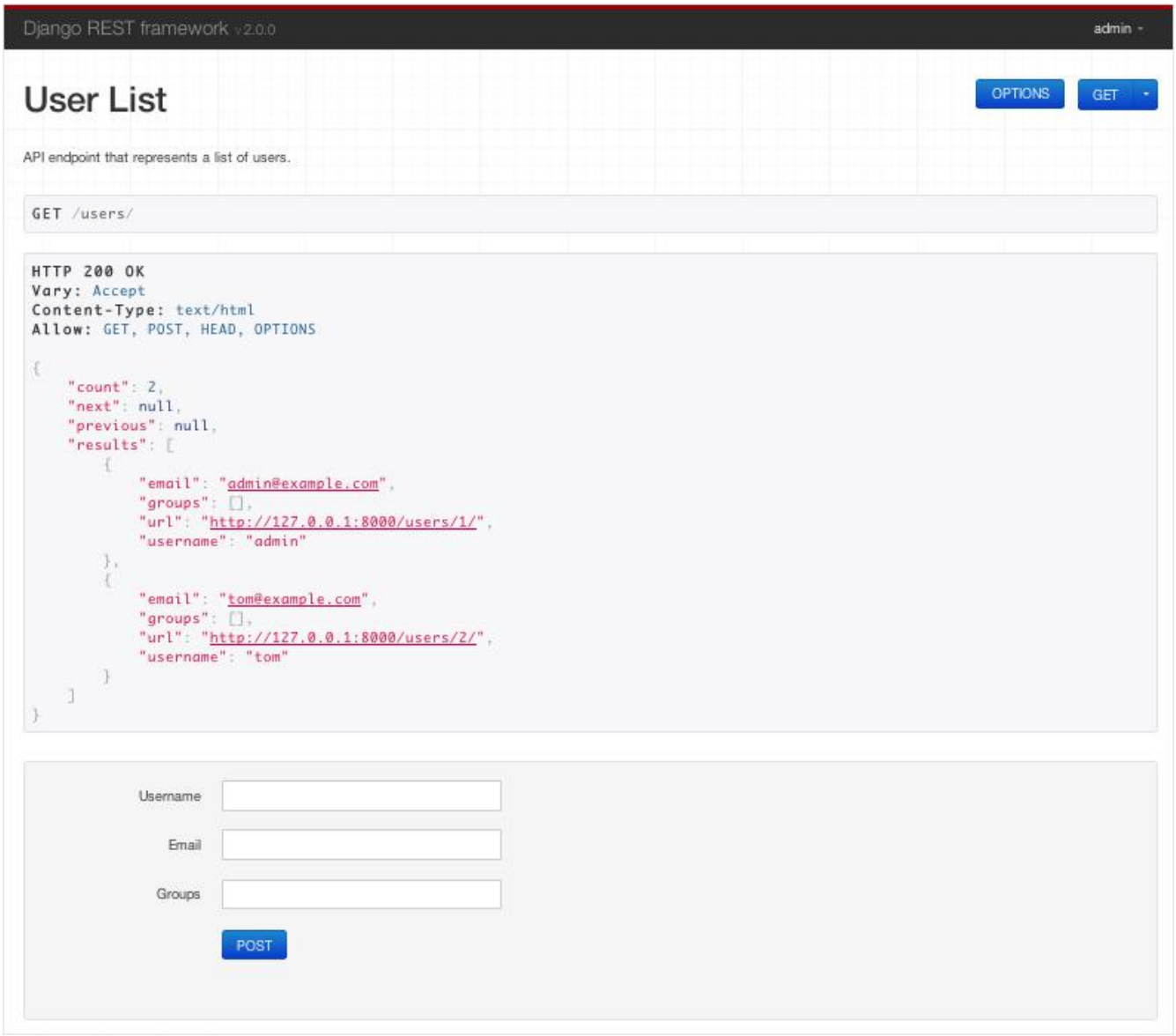
.media_type： `text/html`

.format： `'html'`

返回常规的HTML页面，也可以返回API响应。

BrowsableAPIRenderer

通过这个渲染器可以将数据渲染为HTML页面，提供可浏览的API页面:



BrowsableAPIRenderer会探测哪个其他渲染器被赋予了最高优先级，并使用该渲染器在HTML页面中显示API样式。

.media_type: `text/html`

.format: `'api'`

.charset: `utf-8`

.template: `'rest_framework/api.html'`

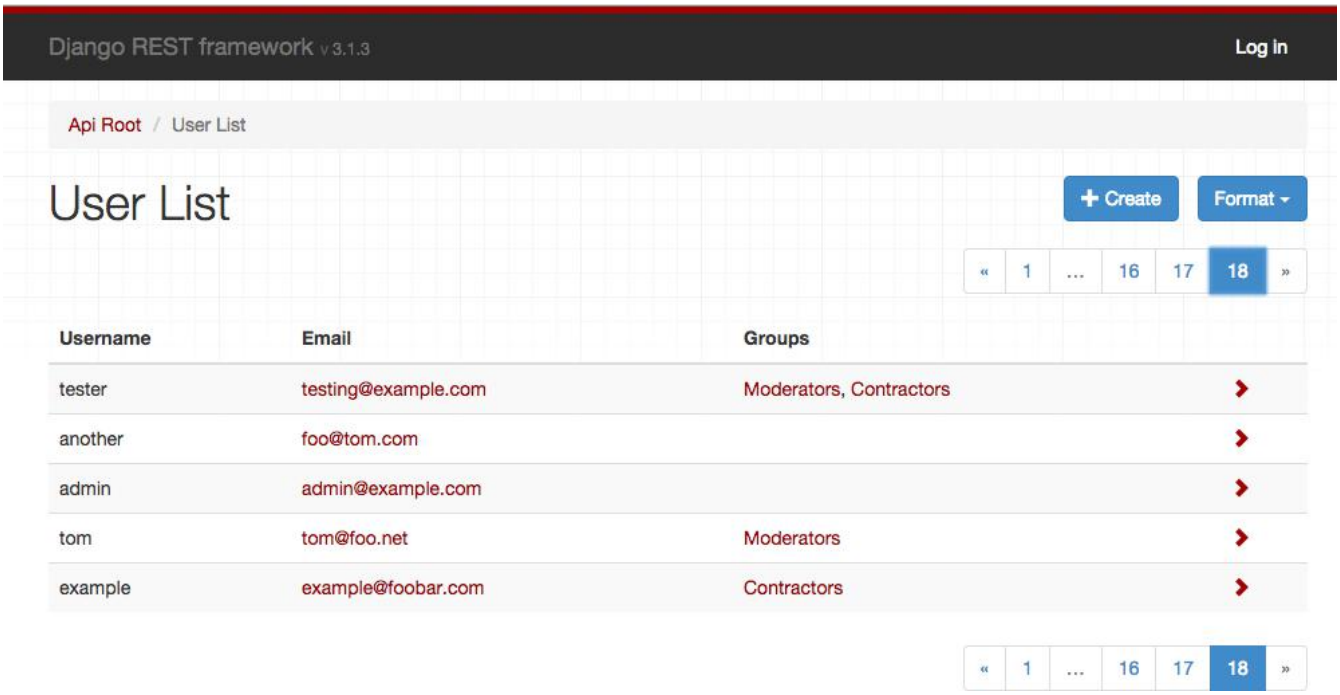
默认情况下，响应内容将使用除 `BrowsableAPIRenderer` 外，最高优先级的渲染器进行渲染。如果您需要自定义此行为，例如使用HTML作为默认返回格式，但在可浏览API中使用JSON。

可以采用下面的解决方法：覆盖 `get_default_renderer()` 方法。例如：

```
class CustomBrowsableAPIRenderer(BrowsableAPIRenderer):
    def get_default_renderer(self, view):
        return JSONRenderer()
```

AdminRenderer

将数据渲染为类似管理员后台界面的HTML页面，如下图所示：



请注意，具有嵌套序列化器或序列化器列表的视图将无法正常使用 `AdminRenderer` ，因为HTML表单无法正确支持它们。

注意： `AdminRenderer` 只有在数据中正确配置 `URL_FIELD_NAME` （ `url` 默认情况下）属性时，才能包含指向详细信息页面的链接。对于 `HyperlinkedModelSerializer` 无需担心，但是对于 `ModelSerializer` 或普通 `Serializer` 类，需要明确包含该字段。例如，我们使用模型 `get_absolute_url` 方法：

```
class AccountSerializer(serializers.ModelSerializer):
    url = serializers.CharField(source='get absolute url', read only=True)

    class Meta:
        model = Account
```

.media_type: `text/html`

.format: `'admin'`

.charset: `utf-8`

.template: `'rest_framework/admin.html'`

HTMLFormRenderer

将序列化程序返回的数据呈现为HTML表单。此渲染器的输出不包括封闭的 `<form></form>` 标签，隐藏的CSRF标签或任何提交按钮，这些都要你自己写。

此渲染器不建议直接使用，而是通过将序列化器的实例传递给 `render_form` 模板标记来在模板中使用。

```
{% load rest framework  %}

<form action="/submit-report/" method="post">
    {% csrf_token %}
    {% render_form serializer %}
    <input type="submit" value="Save" />
</form>
```

.media_type: `text/html`

.format: `'form'`

.charset: `utf-8`

.template: `'rest_framework/horizontal/form.html'`

MultiPartRenderer

此渲染器用于渲染HTML分成多个部分的表单数据。 **它不适合作为响应的渲染器**，而是用于创建测试请求，使用REST框架的测试客户端和测试请求工厂方法。了解即可，绝大多数场景下用不着。

.media_type: `multipart/form-data; boundary=BoUnDaRyStRiNg`

.format: `'multipart'`

.charset: `utf-8`

三、自定义渲染器

要自定义渲染器，首先要继承 `BaseRenderer` 类，然后设置 `.media_type` 和 `.format` 属性，最后实现 `.render(self, data, media_type=None, renderer_context=None)` 方法。

render()方法应返回一个bytestring对象，它将用作HTTP响应的主体。

`.render()` 方法的参数是：

- `data`

请求数据，由 `Response()` 实例化。

- `media_type=None`

可选参数。如果提供，则是内容协商的媒体类型。

- `renderer_context=None`

可选参数。如果提供，则是由视图提供的上下文信息的字典。

默认情况下，这个字典会包括以下的键： `view` ， `request` ， `response` ， `args` ， `kwargs` 。

范例：

自定义一个纯文本渲染器，它将返回一个将 `data` 参数作为内容的响应。

```
from django.utils.encoding import  smart_unicode
from rest framework import renderers

class PlainTextRenderer(renderers.BaseRenderer):
    media_type = 'text/plain'
    format = 'txt'

    def render(self, data, media_type=None, renderer_context=None):
        return data.encode(self.charset)
```

设置字符集

默认情况下，渲染器类使用 `UTF-8` 编码。要使用其他编码，请在渲染器上设置 `charset` 属性。

```
class PlainTextRenderer(renderers.BaseRenderer):
    media_type = 'text/plain'
    format = 'txt'
    charset = 'iso-8859-1'

    def render(self, data, media_type=None, renderer_context=None):
        return data.encode(self.charset)
```

请注意，如果渲染器返回unicode字符串，则响应内容将被强制转换为字节字符串 `Response` ，并在渲染器上设置 `charset` 属性以确定编码。

如果渲染器返回表示原始二进制内容的字节字符串，则应将字符集值设置为 `None` ，这将确保 `Content-Type` 响应的标头不会 `charset` 设置值。

在某些情况下，可能需要将 `render_style` 属性设置为 `'binary'` 。这样做将确保可浏览的API不会尝试将二进制内容显示为字符串。

```
class JPEGRenderer(renderers.BaseRenderer):
    media_type = 'image/jpeg'
    format = 'jpg'
    charset = None
    render style = 'binary'

    def render(self, data, media_type=None, renderer_context=None):
        return data
```

四、渲染器高级用法

可以使用REST框架的渲染器执行一些非常灵活的操作。比如：

- 根据请求的媒体类型，提供来自同一API的平坦或嵌套表示。
- 同时提供常规HTML网页和基于JSON的API响应。
- 为要使用的API客户端指定多种类型的HTML表示。
- 取消指定渲染器的媒体类型，例如使用 `media_type = 'image/*'` ，并使用 `Accept` 标头来改变响应的编码。

媒体类型的不同行为

在某些情况下，您可能希望视图使用不同的序列化器，具体取决于接受的媒体类型。如果需要这样做，你可以访问 `request.accepted_renderer` 以确定将用于响应的渲染器。

例如：

```
@api_view(('GET',))
@renderer_classes((TemplateHTMLRenderer, JSONRenderer))
def list_users(request):
    """
    A view that can return JSON or HTML representations
    of the users in the system.
    """
    queryset = Users.objects.filter(active=True)

    if request.accepted_renderer.format  ==  'html':
        # TemplateHTMLRenderer takes a context  dict,
        # and additionally requires a 'template_name'.
        # It does not require serialization.
        data = {'users': queryset}
        return Response(data, template_name='list_users.html')

    # JSONRenderer requires serialized data as normal.
    serializer =  UserSerializer(instance=queryset)
    data = serializer.data
    return Response(data)
```

未指定媒体类型

在某些情况下，您可能希望渲染器同时支持一系列媒体类型。在这种情况下，可以将 `media_type` 设置为 `*/*` 。

如果未指定渲染器的媒体类型，则应确保在使用该 `content_type` 属性返回响应时明确指定媒体类型。例如：

```
1   return Response(data, content_type='image/png')
```

HTML错误视图

通常情况下，渲染器的行为都是相同的，无论它是处理常规响应，还是由异常引起的响应（如 `Http404` 、 `PermissionDenied` 或 `APIException` 的子类）。

引发和处理的异常的HTML页面将尝试按优先顺序使用以下方法之一进行选择。

- 加载并渲染名为 `{status_code}.html` 的模板。
- 加载并渲染名为 `api_exception.html` 的模板。
- 返回HTTP状态代码和文本，例如“404 Not Found”。

模板将使用 `RequestContext` 包含 `status_code` 和 `details` 键的内容进行渲染。

注意：如果 `DEBUG=True` ，将显示Django的标准回溯错误页面，而不是呈现HTTP状态代码和文本。

五、第三方模块

DRF中，通过以下的第三方模块实现功能更强大的渲染器扩展：

YAML

它以前直接包含在REST框架中，现在作为第三方模块。

使用pip安装。

```
1   $ pip install djangorestframework-yaml
```

修改REST框架设置。

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
    'rest_framework_yaml.parsers.YAMLParser', 4
    ),
    'DEFAULT_RENDERER_CLASSES': (
    'rest_framework_yaml.renderers.YAMLRenderer', 7
    ),
}
```

XML

一种简单的非正式XML格式。它以前直接包含在REST框架中，现在作为第三方模块。

使用pip安装。

```
$ pip install djangorestframework-xml
```

修改REST框架设置。

```
REST_FRAMEWORK = {
    'DEFAULT_PARSER_CLASSES': (
    'rest_framework_xml.parsers.XMLParser', 4
    ),
    'DEFAULT_RENDERER_CLASSES': (
    'rest_framework_xml.renderers.XMLRenderer', 7),
}
```

JSONP

提供JSONP渲染支持。它以前直接包含在REST框架中，现在作为第三方模块。

警告：如果您需要跨域AJAX请求，通常应该使用更现代的CORS 方法作为 `JSONP` 的替代方案。有关更多详细信息，请参阅CORS**文档**。

使用pip安装。

```
$ pip install djangorestframework-jsonp
```

修改REST框架设置。

```
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': (
    'rest_framework_jsonp.renderers.JSONPRenderer', 4
    ),
}
```

XLSX

Microsoh OVice Excle的数据格式。

使用pip安装。

```
$ pip install drf-renderer-xlsx
```

修改REST框架设置。

```
REST_FRAMEWORK = {
    ...

    'DEFAULT_RENDERER_CLASSES': (
        'rest_framework.renderers.JSONRenderer',
        'rest_framework.renderers.BrowsableAPIRenderer',
        'drf_renderer_xlsx.renderers.XLSXRenderer',
    ),
}
```

为了避免文件流没有文件名（浏览器的默认文件名为“download”，没有扩展名），我们需要使用mixin来覆盖 `Content-Disposition` 标题。如果没有提供文件名，则默认为 `export.xlsx` 。例如：

```
from rest_framework.viewsets import ReadOnlyModelViewSet
from drf_renderer_xlsx.mixins import XLSXFileMixin
from drf_renderer_xlsx.renderers import  XLSXRenderer

from .models import MyExampleModel
from .serializers import MyExampleSerializer

class MyExampleViewSet(XLSXFileMixin, ReadOnlyModelViewSet):
    queryset = MyExampleModel.objects.all()
    serializer_class = MyExampleSerializer
    renderer_classes = (XLSXRenderer,)
    filename = 'my_export.xlsx'
```

Pandas（CSV，Excel，PNG）

Django REST Pandas提供了一个序列化器和渲染器，支持使用Pandas DataFrame API进行额外的数据处理和输出。Django REST Pandas这个模块包括用于Pandas风格的CSV文件、Excel工作簿（`.xls`和`.xlsx`）以及许多其他格式的渲染器。

此外，通过第三方模块，还可以支持CSV、LaTeX、UltraJSON、camelcaseJSON和MessagePack等格式。

序列化部分使用到的模型类

```
from datetime import datetime

class Comment(models.Model):

    email=models.EmailField()
    content = models.CharField(max_length=200)
    created = models.DateTimeField()


class Event(models.Model):
    description = models.CharField(max_length=100)
    start = models.DateTimeField()
    finish = models.DateTimeField()

class Event1(models.Model):
    name = models.CharField(max_length=50)
    room_number = models.IntegerField()
    date = models.DateField()


class Account(models.Model):
    name = models.CharField(max_length=50)
    owner =
```

```
models.ForeignKey('auth.User',related_name='accounts',on_delete=models.CASCADE,default=None,db_constraint=False)
```

一、Serializer

> 对于Serializer类，是比较底层和基础，可定制程度最高的类，需要编写代码量最大的继承方式，当你需要高度定制DRF的序列化器的时候选择它，否则请选择后面的类！

序列化器一边把像查询集queryset和模型实例instance这样复杂的基于Django自定义的数据类型转换为可以轻松转换成JSON，XML或其他内容类型的互联网通用的交换语言，让不知道后端到底是什么的前端能够轻松的解析数据内容，而不至于摸瞎。另一边，序列化器还提供反序列化功能，

在验证前端传入过来的数据之后，将它们转换成Django使用的模型实例等复杂数据类型，从而使用ORM与数据库进行交互，也就是CRUD。

序列化器存在的核心原因是前后端分离，前端不知道后端，后端又无法定义前端。为了数据交换，只好都说json话，都将自己的内容翻译成json格式。然而对于前端，json属于自带，对于后端，则需要使用序列化器将Django自定义的数据类型转换成json格式。（当然，也可以是别的格式。）

事实上Django本身也自带序列化功能的，参考 http://www.liujiangblog.com/course/django/171，只不过这里的序列化器不是专门为了前后端分离和API开发的，功能较弱。

REST framework中的serializers与Django的`Form`和`ModelForm`类非常像。DRF提供了一个最底层最基础的`Serializer`类，它类似Django的Form类，为我们提供了强大的通用方法来控制响应的输出。以及一个`ModelSerializer`类，类似Django的`ModelForm`类，为创建用于处理模型实例和查询集的序列化程序提供了快捷实现方式，但可自定义程度较低。

声明序列化器

让我们从创建一个简单的对象开始，我们可以使用下面的例子：

注意，下面是一个原生的Python类，而不是Django的model类，这里介绍的是序列化器的初步原理，不是最终做法：

```
from datetime import datetime
class Comment(object):
    def __init__(self, email, content, created=None):
        self.email = email
        self.content = content
        self.created = created or datetime.now()
comment = Comment(email='leila@example.com', content='foo bar')
```

下面声明一个序列化器，可以使用它来序列化和反序列化`Comment`对象。

声明一个序列化器在代码结构上看起来非常像声明一个Django的Form类：

```
from rest_framework import serializers
class CommentSerializer(serializers.Serializer):
    email = serializers.EmailField()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

序列化对象

现在，我们可以用`CommentSerializer`类去序列化一个comment对象或多个comment的列表。同样，使用`Serializer`类看起来很像使用`Form`类。

```
serializer = CommentSerializer(comment)serializer.data# {'email': 'leila@example.com',
'content': 'foo bar', 'created': '2016-01-27T15:17:10.375877'}
```

这样，我们就将一个comment实例转换为了Python原生的数据类型，看起来似乎是个字典类型，但type一下却不是。为了完成整个序列化过程，我们还需要使用渲染器将数据转化为成品json格式。

```
from rest_framework.renderers import JSONRenderer
json = JSONRenderer().render(serializer.data)json#
b'{"email":"leila@example.com","content":"foo bar","created":"2016-01-
27T15:17:10.375877"}'
```

注意结果里的b，在Python里，这表示Bytes类型，是Python3以后的数据传输格式。

反序列化对象

反序列化的过程大体类似，但差别还是不小。首先，为了举例方便我们将一个流解析为Python原生的数据类型，实际过程中不会有这种多此一举的做法：

```
import iofrom rest_framework.parsers import JSONParser
stream = io.BytesIO(json)data = JSONParser().parse(stream)
```

...然后我们将这些原生数据类型恢复到已验证数据的字典中。

```
serializer = CommentSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# {'content': 'foo bar', 'email': 'leila@example.com', 'created': datetime.datetime(2012,
08, 22, 16, 20, 09, 822243)}
```

注意：依然是调用CommentSerializer类的构造函数，但是给data参数传递数据，而不是第一位置参数，这表示反序列化过程。其次，数据有一个验证过程is_valid()。

这个步骤做完，只是从json变成了原生的Python数据类型，还不是前面自定义的Comment类的对象。

保存实例

如果我们想要返回基于验证数据的完整对象实例，我们需要实现.create()或者update()方法。例如：

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

    def create(self, validated_data):
```

```
        return Comment(**validated_data)

    def update(self, instance, validated_data):
        instance.email = validated_data.get('email', instance.email)
        instance.content = validated_data.get('content', instance.content)
        instance.created = validated_data.get('created', instance.created)
        return instance
```

上面的两个方法在其继承关系中的父类里定义了具体的参数形式，`instance`和`validated_data`都是由继承体系里定义了的，分别表示要返回的Comment对象和经过验证的数据。

如果你的对象实例，比如这里的Comment对象，假设它对应了某个Django的模型，你还需要将这些对象通过Django的ORM保存到数据库中。具体的方法如下所示：

```
    def create(self, validated_data):
        return Comment.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.email = validated_data.get('email', instance.email)
        instance.content = validated_data.get('content', instance.content)
        instance.created = validated_data.get('created', instance.created)
        instance.save()
        return instance
```

相比普通的Python的Comment类，对Django模型的保存多了一些ORM操作，比如save()方法。

等等，前面所作的改进只是在serializer层面的代码逻辑实现，在对应的视图操作中，我们还需要调用save()方法：

```
serializer = CommentSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# {'content': 'foo bar', 'email': 'leila@example.com', 'created': datetime.datetime(2012,
08, 22, 16, 20, 09, 822243)}
comment = serializer.save()  # if serializer.is_valid()
```

调用`.save()`方法将创建新实例或者更新现有实例，具体取决于实例化序列化器类的时候是否传递了一个现有实例：

```
# 这样的情况下.save() 将创建一个新实例
serializer = CommentSerializer(data=data)

# 这样将更新已有的comment实例
serializer = CommentSerializer(comment, data=data)
```

原则上`.create()`和`.update()`方法都是可选的。你可以根据你的序列化器类的用例不实现、实现它们之一或都实现，随你。但是新手，请千万不要随意。

传递附加属性到`.save()`

有时你会希望你的视图代码能够在保存实例时注入额外的数据。此额外数据可能是当前用户，当前时间或不是请求数据一部分的其他信息。

你可以通过在调用`.save()`时添加其他关键字参数来执行此操作。例如：

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

在`.create()`或`.update()`被调用时，所有其他的关键字参数将被包含在`validated_data`参数中。

重写`.save()`方法

在某些情况下`.create()`和`.update()`方法可能无意义。例如在联系人名单中，我们可能不会创建新的实例，而是发送电子邮件或其他消息。

在这些情况下，你可以选择直接重写`.save()`方法，也许更可读和有意义。

例如：

```
class ContactForm(serializers.Serializer):
    email = serializers.EmailField()
    message = serializers.CharField()

    def save(self):
        email = self.validated_data['email']
        message = self.validated_data['message']
        send_email(from=email, message=message)
```

请注意在上述情况下，我们需要直接访问serializer的`.validated_data`属性，因为没有写create或者update方法，没人给你传递`validated_data`参数，只能从self里面自己取。

重写save方法的例子给我们自定义保存数据做了个很好的演示，活学活用，会有很大的惊喜。

验证

反序列化数据的时候，你需要先调用`is_valid()`方法，然后再尝试去访问经过验证的数据或保存对象实例。如果发生任何验证错误，`.errors`属性将包含错误消息，以字典的格式。例如：

```
serializer = CommentSerializer(data={'email': 'foobar', 'content': 'baz'})
serializer.is_valid()
# False
serializer.errors
# {'email': ['Enter a valid e-mail address.'], 'created': ['This field is required.']}
```

字典里的每一个键都是字段名称，值是与该字段对应的错误消息的字符串列表。
`non_field_errors`键可能存在，它将列出任何一般验证错误信息。`non_field_errors`的名称可以通过REST framework设置中的`NON_FIELD_ERRORS_KEY`来自定义。 当对对象列表进行序列化时，返回的错误是每个反序列化项的字典列表。

内置无效数据的异常

.is_valid()方法具有raise_exception异常标志，如果存在验证错误将会抛出一个serializers.ValidationError异常。

这些异常由REST framework提供的默认异常处理程序自动处理，默认情况下将返回HTTP 400 Bad Request响应。

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

字段级别的验证

你可以通过向你的Serializer子类中添加validate_<field_name>方法来指定自定义字段级别的验证。类似于Django表单中的.clean_<field_name>方法。

这些方法采用单个参数，即需要验证的字段值。

validate_<field_name>方法应该返回一个验证过的数据或者抛出一个serializers.ValidationError异常。例如：

```
from rest_framework import serializers
class BlogPostSerializer(serializers.Serializer):
    title = serializers.CharField(max_length=100)
    content = serializers.CharField()

    def validate_title(self, value):
        """        这个例子检查博客是否和django有关。        """
        if 'django' not in value.lower():
            raise serializers.ValidationError("Blog post is not about Django")
        return value
```

注意： 如果你在序列化器中声明<field_name>的时候带有required=False参数，并且未给该字段提供参数，那么这个验证步骤不会执行。

对象级别的验证

要执行需要访问多个字段的任何其他验证，请添加一个.validate()方法到你的Serializer子类中。这个方法使用包含各个字段值的字典作为单个参数，错误情况下应该抛出一个 ValidationError异常，正常情况下应该返回经过验证的值。例如：

```
from rest_framework import serializers
class EventSerializer(serializers.Serializer):
    description = serializers.CharField(max_length=100)
    start = serializers.DateTimeField()
    finish = serializers.DateTimeField()
```

```
    def validate(self, data):
        """        检查开始时间是在结束时间之前。        """
        if data['start'] > data['finish']:
            raise serializers.ValidationError("finish must occur after start")
        return data
```

验证器参数

还可以通过在字段上声明验证器参数的方式为字段设置指定的验证器，例如：

```
def multiple_of_ten(value):
    if value % 10 != 0:
        raise serializers.ValidationError('Not a multiple of ten')
class GameRecord(serializers.Serializer):
    score = IntegerField(validators=[multiple_of_ten])
    ...
```

注意上面给参数提供了一个列表值，这说明可以同时使用多个验证器。

序列化器类还可以设置应用于一组字段的可重用的验证器。这些验证器要在内部的`Meta`类中声明，如下所示：

```
class EventSerializer(serializers.Serializer):
    name = serializers.CharField()
    room_number = serializers.IntegerField(choices=[101, 102, 103, 201])
    date = serializers.DateField()

    class Meta:
        # Each room only has one event per day.
        validators = [UniqueTogetherValidator(
            queryset=Event.objects.all(),
            fields=['room_number', 'date']
        )]
```

有专门的验证器章节，详细介绍这一部分内容。

访问初始数据和实例

将初始化对象或者查询集传递给序列化实例时，可以通过`.instance`访问原始对象。如果没有传递初始化对象，那么`.instance`属性值将是`None`。（正向，即序列化方向）

将数据传递给序列化器实例时，未修改的数据可以通过`.initial_data`获取。如果没有传递data关键字参数，那么`.initial_data`属性不存在。（反向，即反序列化方向）

部分更新

默认情况下，序列化器必须传递所有必填字段的值，否则就会引发验证错误。但是我们可以将 `partial`参数指定为True，来允许部分更新，而不至于产生错误，这很重要。

```
# 使用部分数据更新`comment`
serializer = CommentSerializer(comment, data={'content': 'foo bar'}, partial=True)
```

处理关系对象(从这里到ModelSerializer跳过，到序列化关系讲解)

前面的实例适用于处理只有简单数据类型的对象，但是有时候我们也需要表示更复杂的对象，其中对象的某些属性可能不是字符串、日期、整数这样简单的数据类型。

`Serializer`类本身也是一种`Field`，并且可以用来表示一个对象嵌套在另一个对象中的关系。也就是处理Django模型中的关系类型，一对一、多对一、多对多的字段。

```
class UserSerializer(serializers.Serializer):
    email = serializers.EmailField()
    username = serializers.CharField(max_length=100)
class CommentSerializer(serializers.Serializer):
    user = UserSerializer()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

如果嵌套字段可以接收 `None`值，也就是关联的对象可以为空，应该将 `required=False`标志传递给嵌套的序列化器。

```
class CommentSerializer(serializers.Serializer):
    user = UserSerializer(required=False)  # 有可能未登录
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

类似的，如果嵌套的关联字段可以接收一个列表，那么应该将`many=True`标志传递给嵌套的序列化器。也就是多对一外键和多对多关系的处理方式。

```
class CommentSerializer(serializers.Serializer):
    user = UserSerializer(required=False)
    edits = EditItemSerializer(many=True)  # edit'项的嵌套列表
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

关系字段数据的验证

当我们在处理包含关系字段的序列化过程中，如果关联字段的值不合法，同样会被检测出来，并且将错误信息保存在相应的位置，如下例，CommentSerializer中的user字段关联到auth的User表，但是提供了一个不合法的邮箱地址，sorry，你不能成功反序列化，DRF给出了明确的错误信息。类似的，`.validated_data` 属性也将包括关系字段的数据结构。

```
serializer = CommentSerializer(data={'user': {'email': 'foobar', 'username': 'doe'},
'content': 'baz'})
serializer.is_valid()
# False
serializer.errors
# {'user': {'email': ['Enter a valid e-mail address.']}, 'created': ['This field is
required.']}
```

为关系字段编写.create()方法

如果你对Django的ORM系统理解比较深刻，那么本章节的很多内容都能轻松理解。这些方法其实都是在打通序列化器和Django的ORM系统之间的通道。

对于关系型字段，我们需要编写.create()或.update()处理保存多个对象的方法。

下面的示例演示如何处理创建一个具有关联的概要信息对象的用户。

```
class UserSerializer(serializers.ModelSerializer):
    profile = ProfileSerializer()

    class Meta:
        model = User
        fields = ('username', 'email', 'profile')

    def create(self, validated_data):
        profile_data = validated_data.pop('profile')
        user = User.objects.create(**validated_data)
        Profile.objects.create(user=user, **profile_data)
        return user
```

官网的这个例子不太恰当，因为这里还没有讲解ModelSerializer，我们暂且放过它。重点是UserSerializer类内部的create方法。其代码的核心是，我们需要为关联的profile字段编写如何保存Profile对象的ORM语句！

为关系字段定义.update()方法

对于更新，你需要仔细考虑如何处理关联字段的更新。 例如，如果关联字段的值是None，或者没有提供，那么会发生下面哪种情况？

- 在数据库中将关联字段设置成NULL。
- 删除关联的实例。
- 忽略数据并保留这个实例。
- 抛出验证错误。

下面是我们之前UserSerializer类中update()方法的一个例子。

```
    def update(self, instance, validated_data):
        profile_data = validated_data.pop('profile')
        # Unless the application properly enforces that this field is
        # always set, the follow could raise a `DoesNotExist`, which
        # would need to be handled.
        profile = instance.profile

        instance.username = validated_data.get('username', instance.username)
        instance.email = validated_data.get('email', instance.email)
        instance.save()
```

```
        profile.is_premium_member = profile_data.get(
            'is_premium_member',
            profile.is_premium_member
        )
        profile.has_support_contract = profile_data.get(
            'has_support_contract',
            profile.has_support_contract
         )
        profile.save()

        return instance
```

时刻记住，update里的代码如何写，要根据你的业务逻辑来编写，不要忘了调用save方法，或者弹出异常。

因为关系字段的创建和更新行为可能不明确，并且可能需要关联模型间的复杂依赖关系，DRF从3.x版本后要求你始终明确的定义这些方法。并且，默认的`ModelSerializer` 类的`.create()`和`.update()`方法不包括对关联字段的支持，需要你自己实现，或者需求第三方模块的支持。

使用模型管理类保存关联对象

在序列化器中保存多个相关实例的另一种方法是编写处理创建正确实例的自定义模型管理器类，也就是Django原生的`models.Manager`。

例如，假设我们想确保`User`实例和`Profile`实例总是作为一对一起创建。我们可能会写一个类似这样的自定义管理器类：

```
class UserManager(models.Manager):
    ...

    def create(self, username, email, is_premium_member=False,
has_support_contract=False):
        user = User(username=username, email=email)
        user.save()
        profile = Profile(
            user=user,
            is_premium_member=is_premium_member,
            has_support_contract=has_support_contract
        )
        profile.save()
        return user
```

这个管理器类现使得用户实例和用户信息实例总是在同一时间创建。我们在序列化器类上的`.create()`方法现在可以用新的管理器方法重写一下。

```
def create(self, validated_data):
    return User.objects.create(
        username=validated_data['username'],
        email=validated_data['email']
        is_premium_member=validated_data['profile']['is_premium_member']
```

```
        has_support_contract=validated_data['profile']['has_support_contract']
    )
```

有关此方法的更多详细信息，请参阅Django文档中的 模型管理器

处理多个对象（看一看即可，后面讲解）

Serializer类还可以序列化或反序列化对象的列表。

同时序列化多个对象

为了能够序列化一个查询集或者一个对象列表而不是一个单独的对象，需要在实例化序列化器类的时候传一个many=True参数。这样就能序列化一个查询集或一个对象列表。

```
queryset = Book.objects.all()
serializer = BookSerializer(queryset, many=True)
serializer.data
# [
#     {'id': 0, 'title': 'The electric kool-aid acid test', 'author': 'Tom Wolfe'},
#     {'id': 1, 'title': 'If this is a man', 'author': 'Primo Levi'},
#     {'id': 2, 'title': 'The wind-up bird chronicle', 'author': 'Haruki Murakami'}
# ]
```

反序列化过程中默认支持多个对象的创建，但是不支持多个对象的同时更新。

有关如何支持或自定义这些情况的更多信息，请查阅ListSerializer类。

附加额外的上下文（了解）

在某些情况下，除了要序列化的对象之外，还需要为序列化程序提供额外的上下文。一个常见的情况是，如果你使用包含超链接关系的序列化程序，这需要序列化器能够访问当前的请求以便正确生成完全限定的URL。

可以在实例化序列化器的时候传递一个context参数来传递任意的附加上下文。例如：

```
serializer = AccountSerializer(account, context={'request': request})
serializer.data
# {'id': 6, 'owner': 'denvercoder9', 'created': datetime.datetime(2013, 2, 12, 09, 44,
56, 678870), 'details': 'http://example.com/accounts/6/details'}
```

这个上下文字典可以在任何序列化器字段的中使用，例如.to_representation()方法中可以通过访问self.context属性获取上下文字典。

二、 ModelSerializer

注：更高一级的封装，更少的代码，但也更低的可定制性

类似Django原生的ModelForm对model的引用。

`ModelSerializer`类能够让你自动创建一个具有模型中相应字段的`Serializer`类。

`ModelSerializer`类直接继承了`Serializer`类，不同的是：

- 它根据model模型的定义，自动生成默认字段。
- 它自动生成序列化器的验证器，比如unique_together验证器。
- 它实现了简单的`.create()`方法和`.update()`方法。

声明一个`ModelSerializer`类的方法如下：

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'users', 'created')
```

默认情况下，所有的Django模型类的字段都将映射到序列化器上相应的字段。（请抛弃本章节一开始对普通Python类的序列化概念，以下都是针对Django的模型相关。）

模型中所有的关系字段比如外键都将映射成`PrimaryKeyRelatedField`字段。

默认情况下不包括反向关联，除非像serializer relations文档中规定的那样显式包含。

如果你想确定ModelSerializer自动创建了哪些字段和验证器，可以打开Django shell。

执行 `python manage.py shell`，导入序列化器类，实例化它，并打印对象的表示：

```
>>> from myapp.serializers import AccountSerializer
>>> serializer = AccountSerializer()
>>> print(repr(serializer))
AccountSerializer():
    id = IntegerField(label='ID', read_only=True)
    name = CharField(allow_blank=True, max_length=100, required=False)
    owner = PrimaryKeyRelatedField(queryset=User.objects.all())
```

核心是这个`repr(serializer)`，在ModelSerializer中有专门的实现代码。事实上，你直接print也行，这些都是Python的特性。

指定要包括的字段

如果你希望在模型序列化器中只使用默认字段的一部分，你可以使用fields或exclude选项来执行此操作，就像使用ModelForm一样。强烈建议你使用fields属性显式的设置要序列化的字段。这样就不太可能因为你修改了模型而无意中暴露了数据。

例如：

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'users', 'created')
```

你还可以将fields属性设置成'__all__'来表明使用模型中的所有字段。

例如：

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = '__all__'
```

你可以将exclude属性设置成一个从序列化器中排除的字段列表。

例如：

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        exclude = ('users',)
```

在上面的例子中，如果Account模型有三个字段account_name，users和created，那么只有 account_name和created会被序列化。

在fields和exclude属性中的名称，通常会映射到模型类中的模型字段。

fields选项中的名称也可以映射到模型类中不存在任何参数的属性或方法。

最后，思考一个问题，如果序列化器没有映射Django模型中的必填字段，会发生什么？废话，肯定是出错呀，缺少了必填的字段的内容，Django的ORM怎么操作数据库的读写？所以，这里的fields和exclude要慎用，过度的浪就会浪出坑。

关系字段的序列化深度

ModelSerializer默认使用主键进行对象关联，但是你也可以使用depth选项轻松生成嵌套关联：

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
```

```
        fields = ('id', 'account_name', 'users', 'created')
        depth = 1
```

depth选项应该设置一个整数值，表明应该遍历的关联深度。

如果要自定义序列化的方式你需要自定义该字段。

明确指定字段

觉得全自动的字段不满足需求，也可以轻度定制，这就向基础的Serializer类靠拢了一点。

可以通过在ModelSerializer类上显式声明字段来增加额外的字段或者重写默认的字段，就和在Serializer类一样的。

```
class AccountSerializer(serializers.ModelSerializer):
    url = serializers.CharField(source='get_absolute_url', read_only=True)
    groups = serializers.PrimaryKeyRelatedField(many=True)

    class Meta:
        model = Account
```

额外的字段可以对应模型上任何属性或可调用的方法。

指定只读字段（演示）

你可能希望将某些字段指定为只读，而不是显式的逐一为每个字段添加read_only=True属性，这种情况你可以使用Meta的read_only_fields选项。

该选项应该是字段名称的列表或元组，并像下面这样声明：

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'users', 'created')
        read_only_fields = ('account_name',)
```

模型中设置editable=False的字段和AutoField字段默认被设置为只读属性，不需要额外添加到read_only_fields选项中。

注意: 有一种特殊情况，其中一个只读字段是模型级别unique_together约束的一部分。在这种情况下，序列化器需要该字段的值才能验证约束，但也是不能由用户编辑的。

处理此问题的正确方法是在序列化器上显式指定该字段，同时提供read_only=True和default=…关键字参数。

这种情况的一个例子就是对于一个和其他标识符unique_together的当前认证的User是只读的。在这种情况下你可以像下面这样声明user字段：

```
user = serializers.PrimaryKeyRelatedField(read_only=True,
default=serializers.CurrentUserDefault())
```

附加关键字参数

还可以通过使用extra_kwargs选项快捷地在字段上指定任意附加的关键字参数。这个选项是一个将具体字段名称当作键值的字典。例如：

```
class CreateUserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('email', 'username', 'password')
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User(
            email=validated_data['email'],
            username=validated_data['username']
        )
        user.set_password(validated_data['password'])
        user.save()
        return user
```

关系字段

在序列化模型实例的时候，你可以选择多种不同的方式来表示关联关系。对于ModelSerializer默认是使用相关实例的主键，也就是PrimaryKeyRelatedField。

替代的其他方法包括使用超链接序列化、序列化完整的嵌套表示或者使用自定义表示的序列化。

更多详细信息请查阅serializer relations章节

自定义字段映射（跳过，后面项目讲解）

ModelSerializer类还暴露了一个可以覆盖的API，以便在实例化序列化器时改变序列化器字段的自动确定方式。

通常情况下，如果ModelSerializer没有生成默认情况下你需要的字段，那么你应该将它们显式地添加到类中，或者直接使用常规的Serializer类。但是在某些情况下，你可能需要创建一个新的基类，定义任意模型创建序列化字段的方式。

- .serializer_field_mapping

将Django model的字段类型映射到REST framework serializer的字段类型。你可以覆写这个映射。

- `.serializer_related_field`

用于关联对象的字段类型。

对于`ModelSerializer`此属性默认是`PrimaryKeyRelatedField`。

对于`HyperlinkedModelSerializer`此属性默认是`serializers.HyperlinkedRelatedField`。

- `serializer_url_field`

`url`类型的字段类。默认是 `serializers.HyperlinkedIdentityField`

- `serializer_choice_field`

选择类型的字段类。默认是`serializers.ChoiceField`

The field_class和field_kwargs API

调用下面的方法来确定应该自动包含在序列化器类中每个字段的类和关键字参数。这些方法都应该返回 `(field_class, field_kwargs)`元组。

- `.build_standard_field(self, field_name, model_field)`

调用后生成对应标准模型字段的序列化器字段。

默认实现是根据`serializer_field_mapping`属性返回一个序列化器类。

- `.build_relational_field(self, field_name, relation_info)`

调用后生成对应关联模型字段的序列化器字段。

默认实现是根据`serializer_relational_field`属性返回一个序列化器类。

这里的`relation_info`参数是一个命名元组，包含`model_field`，`related_model`，`to_many`和`has_through_model`属性。

- `.build_nested_field(self, field_name, relation_info, nested_depth)`

当`depth`选项被设置时，被调用后生成一个对应到关联模型字段的序列化器字段。

默认实现是动态的创建一个基于`ModelSerializer`或`HyperlinkedModelSerializer`的嵌套的序列化器类。

nested_depth的值是depth的值减1。

relation_info参数是一个命名元组，包含 model_field，related_model，to_many和has_through_model属性。

- .build_property_field(self, field_name, model_class)

被调用后生成一个对应到模型类中属性或无参数方法的序列化器字段。

默认实现是返回一个ReadOnlyField类。

- .build_url_field(self, field_name, model_class)

被调用后为序列化器自己的url字段生成一个序列化器字段。默认实现是返回一个HyperlinkedIdentityField类。

- .build_unknown_field(self, field_name, model_class)

当字段名称没有对应到任何模型字段或者模型属性时调用。 默认实现会抛出一个错误，尽管子类可能会自定义该行为。

三、 HyperlinkedModelSerializer

进一步封装了ModelSerializer类，并且自动多出了一个url字段

HyperlinkedModelSerializer类直接继承ModelSerializer类，不同之处在于它使用超链接来表示关联关系而不是主键。

默认情况下，HyperlinkedModelSerializer序列化器将包含一个url字段而不是主键字段。

url字段将使用HyperlinkedIdentityField字段来表示，模型的任何关联都将使用HyperlinkedRelatedField字段来表示。

你可以通过将主键添加到fields选项中来显式的包含主键字段，例如下面的id：

```
class AccountSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Account
        fields = ('url', 'id', 'account_name', 'users', 'created')
```

这个类的源代码非常简单：

```
class HyperlinkedModelSerializer(ModelSerializer):
    # 下面这句是核心
```

```
    serializer_related_field = HyperlinkedRelatedField

    def get_default_field_names(self, declared_fields, model_info):
        # 覆写了ModelSerializer中的方法，在第一个变量处发生了变化，使用了url名字。
        return (
            [self.url_field_name] +
            list(declared_fields) +
            list(model_info.fields) +
            list(model_info.forward_relations)
        )

    def build_nested_field(self, field_name, relation_info, nested_depth):
        # 覆写了ModelSerializer中的方法，嵌套的子类依然继承的是HyperlinkedModelSerializer
        class NestedSerializer(HyperlinkedModelSerializer):
            class Meta:
                model = relation_info.related_model
                depth = nested_depth - 1
                fields = '__all__'

        field_class = NestedSerializer
        field_kwargs = get_nested_relation_kwargs(relation_info)

        return field_class, field_kwargs
```

绝对和相对URL

当实例化一个`HyperlinkedModelSerializer`时，你必须在序列化器的上下文中包含当前的`request`值，例如：

```
serializer = AccountSerializer(queryset, context={'request': request})
```

因为我们需要生成url字段的内容，需要request里关于请求的url路径信息，这样做将确保超链接可以包含恰当的主机名，生成完整的url路径，如下面的：

```
http://api.example.com/accounts/1/
```

而不是相对的URL，例如：

```
/accounts/1/
```

如果你真的要使用相对URL，你应该明确的在序列化器上下文中传递一个`{'request': None}`参数，而不是忽略不写。

如何确定超链接视图（后面在序列化器关系演示）

我们为什么要使用超链接的序列化器？因为默认的主键字段只是冰冷的数字id，也就是1，2，3等等，在前端你无法知道它具体的含义。使用超链接则返回的是对应对象的url访问地址，是可以点击跳转直达的，更加形象。

既然是点击跳转可以直达，那么就需要确定哪些视图能应用超链接到模型实例的方法，否则没有对应的视图来处理这些链接的请求，就会404了。

默认情况下，超链接期望对应到一个样式能匹配'{model_name}-detail'的视图，并通过pk关键字参数查找实例。

但是如果你想自己玩点花样，那么可以通过在extra_kwargs中设置view_name和lookup_field中的一个或两个来重写URL字段视图名称和查询字段。如下所示：

```
class AccountSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Account
        fields = ('account_url', 'account_name', 'users', 'created')
        extra_kwargs = {
            'url': {'view_name': 'accounts', 'lookup_field': 'account_name'},
            'users': {'lookup_field': 'username'}
        }
```

或者你可以显式的设置序列化器上的字段。例如：

```
class AccountSerializer(serializers.HyperlinkedModelSerializer):
    url = serializers.HyperlinkedIdentityField(
        view_name='accounts',
        lookup_field='slug'
    )
    users = serializers.HyperlinkedRelatedField(
        view_name='user-detail',
        lookup_field='username',
        many=True,
        read_only=True
    )

    class Meta:
        model = Account
        fields = ('url', 'account_name', 'users', 'created')
```

提示：正确匹配超链接表示和你的URL配置有时可能会有点困难。打印一个HyperlinkedModelSerializer实例的repr是一个特别有用的方式来检查关联关系映射的那些视图名称和查询字段。

更改URL字段名称

URL字段的名称默认为'url'。你可以在settings.py中修改URL_FIELD_NAME配置项进行全局性修改。

四、 ListSerializer

与前三者不同，ListSerializer继承的是BaseSerializer，属于Serializer的兄弟

ListSerializer类能够一次性序列化和验证多个对象。我们通常不要直接使用ListSerializer，而是应该在实例化一个序列化器时简单地传递一个many=True参数。

当一个序列化器在带有many=True选项被序列化时，实际将创建一个ListSerializer类的实例。该序列化器类将成为ListSerializer类的子类。（**和我们后面讲的功能一致**）

可以传递一个allow_empty参数给ListSerializer序列化器。这个参数的默认值是True，但是如果你不想把空列表当作有效输入的话可以把它设置成False。

自定义ListSerializer行为（一次创建多个对象，不重要，跳过）

碰到下面的情况，你可能需要自定制ListSerializer的一些行为：

- 希望提供列表的特定验证，例如检查一个元素是否与列表中的另外一个元素冲突。
- 想自定义多个对象的创建或更新行为。

对于这些情况，当你可以通过使用序列化器类的Meta类下面的list_serializer_class选项来修改当many=True时正在使用的类。

也就是说这个时候你必须写一个继承了serializers.ListSerializer类的子类，然后在需要用它的序列化器类的Meta中添加list_serializer_class条目。

例如：

```
class CustomListSerializer(serializers.ListSerializer):
    ...
class CustomSerializer(serializers.Serializer):
    ...
    class Meta:
        list_serializer_class = CustomListSerializer
```

自定义多个对象的创建

默认情况下，多个对象的创建是简单的对列表中每个对象调用.create()方法。如果要自定义实现，那么你需要自定义当被传递many=True参数时使用的ListSerializer类中的.create()方法。

例如：

```
class BookListSerializer(serializers.ListSerializer):
    def create(self, validated_data):
        books = [Book(**item) for item in validated_data]
        return Book.objects.bulk_create(books)
```

```
class BookSerializer(serializers.Serializer):
    ...
    class Meta:
        list_serializer_class = BookListSerializer
```

自定义多对象的更新

默认情况下，`ListSerializer`类不支持多对象的更新。这是因为插入和删除的预期行为是不明确的，每个人的业务逻辑都可能不同。

要支持多对象同时更新的话你需要自己明确地实现相关的代码。编写多时要注意以下几点：

- 如何确定数据列表中的每个元素应该对应更新哪个实例？
- 如何处理插入？它们是无效的？还是创建新对象？
- 移除应该如何处理？它们是要删除对象还是删除关联关系？它们应该被忽略还是提示无效操作？
- 排序如何处理？改变两个元素的位置是否意味着任何状态改变或者应该被忽视？

你需要向实例序列化器中显式添加一个`id`字段。默认隐式生成的`id`字段是`read_only`。这就导致它在更新时被删除。一旦你明确地声明它，它将在列表序列化器的`update`方法中可用。

下面是一个多对象更新的示例：

```
class BookListSerializer(serializers.ListSerializer):
    def update(self, instance, validated_data):
        # Maps for id->instance and id->data item.
        book_mapping = {book.id: book for book in instance}
        data_mapping = {item['id']: item for item in validated_data}

        # Perform creations and updates.
        ret = []
        for book_id, data in data_mapping.items():
            book = book_mapping.get(book_id, None)
            if book is None:
                ret.append(self.child.create(data))
            else:
                ret.append(self.child.update(book, data))

        # Perform deletions.
        for book_id, book in book_mapping.items():
            if book_id not in data_mapping:
                book.delete()

        return ret
class BookSerializer(serializers.Serializer):
    # We need to identify elements in the list using their primary key,
    # so use a writable field here, rather than the default which would be read-only.
    id = serializers.IntegerField()
    ...

    class Meta:
```

```
        list_serializer_class = BookListSerializer
```

自定义ListSerializer初始化

当带有many=True参数的序列化器被实例化时，我们需要确定哪些参数和关键字参数应该被传递给子Serializer类和父类ListSerializer的.__init__()方法。

默认实现是将所有参数都传递给两个类，除了validators和自定义的关键字参数，这些参数都假定用于子序列化器类。

有时候，你可能需要明确指定当被传递many=True参数时，子类和父类应该如何实例化。你可以使用many_init类方法来执行此操作。

```
@classmethod
  def many_init(cls, *args, **kwargs):
      # Instantiate the child serializer.
      kwargs['child'] = cls()
      # Instantiate the parent list serializer.
      return CustomListSerializer(*args, **kwargs)
```

五、 BaseSerializer（不重要，跳过）

> 这是DRF中其它序列化器的基类，一般我们不直接使用它。所以，下面的内容看看就好，非深度用户无需了解。

BaseSerializer 可以简单的用来替代序列化和反序列化的样式。

Serializer类直接继承了BaseSerializer类，所以两者具有基本相同的API：

- .data - 返回传出的原始数据。
- .is_valid() - 反序列化并验证传入的数据。
- .validated_data - 返回经过验证后的传入数据。
- .errors - 返回验证期间的错误。
- .save() - 将验证的数据保留到对象实例中。

它还有可以覆写的四种方法，具体取决于你想要序列化类支持的功能：

- .to_representation() - 重写此方法来改变读取操作的序列化结果。
- .to_internal_value() - 重写此方法来改变写入操作的序列化结果。
- .create() 和 .update() - 重写其中一个或两个来改变保存实例时的动作。

因为此类提供与Serializer类相同的接口，所以你可以将它与现有的基于类的通用视图一起使用，就像使用常规Serializer或ModelSerializer一样。

你需要注意到的唯一区别是BaseSerializer类并不会在可浏览的API页面中生成HTML表单。

- 只读的 `BaseSerializer` 类

要使用`BaseSerializer`类实现只读的序列化器，我们只需要覆写`.to_representation()`方法。让我们看一个简单的Django模型的示例：

```
class HighScore(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    player_name = models.CharField(max_length=10)
    score = models.IntegerField()
```

创建一个只读的序列化程序来将`HighScore`实例转换为原始数据类型非常简单。

```
class HighScoreSerializer(serializers.BaseSerializer):
    def to_representation(self, obj):
        return {
            'score': obj.score,
            'player_name': obj.player_name
        }
```

我们现在可以使用这个类来序列化单个`HighScore`实例：

```
@api_view(['GET'])
def high_score(request, pk):
    instance = HighScore.objects.get(pk=pk)
    serializer = HighScoreSerializer(instance)
    return Response(serializer.data)
```

或者使用它来序列化多个实例：

```
@api_view(['GET'])
def all_high_scores(request):
    queryset = HighScore.objects.order_by('-score')
    serializer = HighScoreSerializer(queryset, many=True)
    return Response(serializer.data)
```

- 可读写的`BaseSerializer`类

要创建一个读写都支持的序列化器，我们首先需要实现`.to_internal_value()`方法。这个方法返回用来构造对象实例的经过验证的值，如果提供的数据格式不正确，则可能引发`ValidationError`。

一旦你实现了`.to_internal_value()`方法，那些基础的验证API都会在序列化对象上可用了，你就可以使用`.is_valid()`, `.validated_data` 和 `.errors` 方法。

如果你还想支持`.save()`，你还需要实现`.create()`和`.update()`方法中的一个或两个。

下面是支持读、写操作的 `HighScoreSerializer` 完整示例:

```
class HighScoreSerializer(serializers.BaseSerializer):
    def to_internal_value(self, data):
        score = data.get('score')
        player_name = data.get('player_name')

        # Perform the data validation.
        if not score:
            raise serializers.ValidationError({
                'score': 'This field is required.'
            })
        if not player_name:
            raise serializers.ValidationError({
                'player_name': 'This field is required.'
            })
        if len(player_name) > 10:
            raise serializers.ValidationError({
                'player_name': 'May not be more than 10 characters.'
            })

        # Return the validated values. This will be available as
        # the `.validated_data` property.
        return {
            'score': int(score),
            'player_name': player_name
        }

    def to_representation(self, obj):
        return {
            'score': obj.score,
            'player_name': obj.player_name
        }

    def create(self, validated_data):
        return HighScore.objects.create(**validated_data)
```

- 创建一个新的基类

BaseSerializer类还可以用来创建新的通用序列化基类来处理特定的序列化样式或者用来整合备用存储后端。

下面这个类是一个可以将任意对象强制转换为基本表示的通用序列化类的示例。

```
class ObjectSerializer(serializers.BaseSerializer):
    """    A read-only serializer that coerces arbitrary complex objects    into
primitive representations.    """
    def to_representation(self, obj):
        for attribute_name in dir(obj):
            attribute = getattr(obj, attribute_name)
            if attribute_name('_'):
                # Ignore private attributes.
                pass
            elif hasattr(attribute, '__call__'):
```

```
            # Ignore methods and other callables.
            pass
        elif isinstance(attribute, (str, int, bool, float, type(None))):
            # Primitive types can be passed through unmodified.
            output[attribute_name] = attribute
        elif isinstance(attribute, list):
            # Recursively deal with items in lists.
            output[attribute_name] = [
                self.to_representation(item) for item in attribute
            ]
        elif isinstance(attribute, dict):
            # Recursively deal with items in dictionaries.
            output[attribute_name] = {
                str(key): self.to_representation(value)
                for key, value in attribute.items()
            }
        else:
            # Force anything else to its string representation.
            output[attribute_name] = str(attribute)
```

六、 serializer高级用法

重写序列化和反序列化行为（了解）

如果你需要自定义序列化类的序列化、反序列化或验证过程的行为，可以通过重写`.to_representation()`或`.to_internal_value()`方法来完成。

这么做的一些原因包括……

- 为新的序列化基类添加新行为。
- 对现有的类稍作修改。
- 提高频繁访问返回大量数据的API端点的序列化性能。

这些方法的签名如下：

- `.to_representation(self, obj)`

接收一个需要被序列化的对象实例并且返回一个序列化之后的表示。通常，这意味着返回内置Python数据类型的结构。可以处理的确切类型取决于你为API配置的渲染类。这是正向过程，由后端往前端。

- `.to_internal_value(self, data)`

将未经验证的传入数据作为输入，返回可以通过`serializer.validated_data`来访问的已验证的数据。如果在序列化类上调用`.save()`，则该返回值也将传递给`.create()`或`.update()`方法。这是反向的过程，由前端数据往后端保存。

如果任何验证条件失败，那么该方法会引发一个`serializers.ValidationError(errors)`。通常，此处的`errors`参数将是错误消息字典的一个映射字段，或者是`settings.NON_FIELD_ERRORS_KEY`设置的值。

传递给此方法的`data`参数通常是`request.data`的值，因此它提供的数据类型将取决于你为API配置的解析器类。

Serializer的继承

与Django表单类似，你可以通过继承扩展和重用序列化类。这允许你在父类上声明一组公共字段或方法，然后可以在许多序列化器中使用它们。举个例子，

```
class MyBaseSerializer(Serializer):
    my_field = serializers.CharField()

    def validate_my_field(self, value):
        ...
class MySerializer(MyBaseSerializer):
    ...
```

像Django的`Model`和`ModelForm`类一样，序列化器内部`Meta`类不会隐式地继承它的父元素内部的`Meta`类。如果你希望从父类继承`Meta`类，则必须明确地这样做。比如：

```
class AccountSerializer(MyBaseSerializer):
    class Meta(MyBaseSerializer.Meta):
        model = Account
```

但是，我们建议不要在内部Meta类上使用继承，而是明确声明所有选项。

此外，以下警告适用于序列化类的继承过程：

-

使用正常的Python名称解析规则。如果你有多个声明了`Meta`类的基类，则只使用第一个类。这意味要么是孩子的`Meta`（如果存在），否则就是第一个父类的`Meta`等。

-
-

通过在子类上将名称设置为`None`，可以声明性地删除从父类继承的`Field`。

-

``` class MyBaseSerializer(ModelSerializer): my_field = serializers.CharField()
```

class MySerializer(MyBaseSerializer): my_field = None ```

但是，你只能使用此黑科技去掉父类显式声明定义的字段；它不会阻止ModelSerializer生成的默认字段。

动态修改字段（了解）

访问或修改序列化器的fileds属性可以动态地修改字段。这可以实现一些有趣的操作，比如在运行过程中修改某些字段的参数。

比如下面的例子：

```
class DynamicFieldsModelSerializer(serializers.ModelSerializer):
    """    A ModelSerializer that takes an additional `fields` argument that    controls which fields should be displayed.    """

    def __init__(self, *args, **kwargs):
        # Don't pass the 'fields' arg up to the superclass       fields = kwargs.pop('fields', None)

        # Instantiate the superclass normally
        super(DynamicFieldsModelSerializer, self).__init__(*args, **kwargs)

        if fields is not None:
            # Drop any fields that are not specified in the `fields` argument.
            allowed = set(fields)
            existing = set(self.fields)
            for field_name in existing - allowed:
                self.fields.pop(field_name)
```

序列化字段

一 序列化器的字段

DRF在Django字段类型的基础上，派生出了自己的一系列字段类型以及字段参数。

序列化器的字段类型用于处理原始值和内部数据类型之间的转换。它们还用于验证输入值，以及从父对象检索和设置值。

注意：虽然序列化器的字段类型是在 `fields.py` 模块中声明的，但按照惯例，我们一般导入 `from rest_framework import serializers` ，并 `serializers.<FieldName>`

，然后使用 的方式引用字段类型。

一、字段的核心参数

下面是字段类型的通用参数，适合所有字段类型：

`read_only`

该参数默认为False，设置为True则将字段变为只读。

被设置成只读的字段可以包含在API输出中，但在创建或更新操作期间不应包含在输入中。即使你在序列化的时候为一个 `read_only` 字段提供值，也会被忽略，也就是说这个时候的 `validated_data` 中不会包括你发送过来的被设置为只读字段的数据。

`write_only`

默认为 `False` ，将其设置为 `True` 则表示只写不读。

`required`

如果反序列化期间未提供字段，通常会引发错误。如果反序列化过程中不需要此字段，则设置为false。

将此设置为，`False`还可以在序列化实例时从输出中省略对象属性或字典键。如果不存在该密钥，则它将不会直接包含在输出表示中。

默认为`True`。

allow_null

该参数的默认值为 `False`，表示该字段不允许为空。如果设置为True，表示字段的值可以是空的，或者接受一个None(JSON中是NULL)。类似于Django的blank为True

source

该参数是一种指定字段的值如何填充的方法。指定了这个参数，那么这个字段可能就不需要从请求中获取，也可以在序列化的时候自动获取对应的值。单词source应该理解为数据的来源，值从哪里获得的意思，通常是对应model模型中的某个字段。

这个参数的值可能是一个只接受 `self` 参数的方法，例如 `URLField(source='get_absolute_url')`，或者是使用点分表示来遍历属性，例如 `EmailField(source='user.email')`。使用点分表示法序列化字段时，如果对象不存在或在属性遍历的结果为空，则可能需要提供 `default` 参数值。

设置 `source='*'` 具有特殊含义，用于指示整个对象应该传递到该字段。这对于创建嵌套表示或者需要访问整个对象以确定输出表示的字段非常有用。

默认为字段的名称。

```
1  user_role = serializers.CharField(source='user_type')    # 获取model中user_type
   字段的内容
2  url = serializers.URLField(source='get_absolute_url')    # 获取完整url地址
3  group_id = serializers.CharField(source='group.id')    # 获取外键关联的组的id
4  students = serializers.CharField(source='student.all') # 获取多对多关联的所有学
   生，该做法有缺陷
5  serializers.CharField(source='user_type')
```

validators

该参数用于为字段指定专门的验证器，可以是多个验证器的列表。它会引发验证错误或只是简单地返回。

error_messages

错误消息的错误代码字典。也就是说，你可以自定义错误信息。

label

一个简短的文本字符串，可用作HTML表单或其他描述性元素中生成字段的标签。和Django原生的一个意思。

help_text

一个文本字符串，可用作HTML表单或其他描述性元素中对字段的描述文字。和Django原生的一个意思。

initial

该参数用于预先填充HTML表单字段的值，也就是给字段一个初始化值。也可以将一个callable可调用方法传递给它：(帮助后端开发进行测试方便)

```
import datetime
from rest_framework import serializers
class ExampleSerializer(serializers.Serializer):
    day = serializers.DateField(initial=datetime.date.today)
```

style

该参数的值必须为键值对的字典，可用于控制渲染器应如何渲染字段。也就是为字段添加额外的CSS或HTML控制。(为了方便后端测试)

这里的两个例子是 `'input_type'` 和 `'base_template'` ：

```
# Use <input type="password"> for the input.
password = serializers.CharField(
    style={'input_type': 'password'}
)

# Use a radio input instead of a select input.
color_channel = serializers.ChoiceField(
    choices=['red', 'green', 'blue'],
    style={'base_template':  'radio.html'}
)
```

下面开始介绍DRF中各种类型的字段。

二、布尔类型字段

BooleanField

普通的布尔字段。也就是值只能为True或者False。

对应于 `django.db.models.fields.BooleanField` 。

签名： `BooleanField()`

NullBooleanField

接 受`None` 作为有效值的布尔字段，也就是说这个字段也可以为空。

对应于 `django.db.models.fields.NullBooleanField` 。

签名： `NullBooleanField()`

三、字符串类型字段

CharField

最基本的文本类型。常用的参数是验证文本是否短于 `max_length` 和长于 `min_length` 。注意，不同于Django原生，这两个参数非必须。

对应于 `django.db.models.fields.CharField` 或 `django.db.models.fields.TextField` 。

签 名 ： `CharField(max_length=None, min_length=None, allow blank=False,`

- `max_length` - 字段最大长度
- `min_length` - 字段最小长度
- `allow_blank` - 字段是否可以为空，默认为 `False` 。
- `trim_whitespace` - 如果设置为 `True` 则会自动修剪字符串的前导和尾随的空格，相当于 `True`

自动应用了一个字符串的strip方法。默认为　　。

虽然 `allow_null` 参数也可用于字符串字段，但不鼓励同时设置 `allow_blank=True` 和 `allow_null=True` ，这可能会导致数据不一致和微妙的应用程序错误。

EmailField

文本格式，验证为有效的电子邮件地址。

对应于 `django.db.models.fields.EmailField`

签名： `EmailField(max_length=None, min_length=None, allow_blank=False)`

RegexField

文本格式，字段的值必须与regex参数指定的正则表达式相匹配。

对应于 `django.forms.fields.RegexField` 。

签 名 ： `RegexField(regex, max_length=None, min_length=None,`

必填的第一位置参数 `regex` 可以是字符串，也可以是编译过的python正则表达式对象。

此字段使用Django的 `django.core.validators.RegexValidator` 进行验证。

SlugField

一个 `RegexField` 字段，根据 `[a-zA-Z0-9_-]+` 正则模式验证输入。也就是定死了正则表达式。

对应于 `django.db.models.fields.SlugField` 。

签名： `SlugField(max_length=50, min_length=None, allow_blank=False)`

URLField

一个 `RegexField`类型字段，规定字段必须是个合法的URL类型的字符串，根据URL匹配模式验证输入。类似 `http://<host>/<path>` 的形式。

对应于 `django.db.models.fields.URLField` 。使用Django的 `django.core.validators.URLValidator` 进行验证。

签名： `URLField(max_length=200, min_length=None, allow_blank=False)`

UUIDField

确保输入的字段是有效的UUID字符串。`to_internal_value` 方法将返回一个 `uuid.UUID` 实例。在输出时，该字段将以规范的以短横线连接的格式返回一个字符串，例如：

```
"de305d54-75b4-431b-adb2-eb6b9e546013"
```

签名： `UUIDField(format='hex_verbose')`

- format：指定uuid值的表示格式
  - `'hex_verbose'` - 十六进制表示，包括连字符：`"5ce0e9a5-5ffa-654b-cee0-1238041fb31a"`
  - `'hex'` - UUID的紧凑十六进制表示形式，不包括连字符：`"5ce0e9a55ffa654bcee01238041fb31a"`
  - `'int'` - UUID的128位整数表示：`"123456789012312313134124512351145145114"`
  - `'urn'`-UUID的RFC4122URN表示：`"urn:uuid:5ce0e9a5-5ffa-654b-cee0-1238041fb31a"`

FilePathField

文件路径类型字段，其选择仅限于文件系统上某个目录中的文件名

对应于 `django.forms.fields.FilePathField` 。

签 名 ： `FilePathField(path, match=None, recursive=False, allow files=True,`

- `path` - 此FilePathField应从中选择的目录的绝对文件系统路径。
- `match` - 作为字符串的正则表达式，FilePathField用它过滤文件名。
- `recursive` - 指定是否应递归搜索路径的所有子目录。默认是 `False` 。
- `allow_files` - 指定是否应包括指定位置的文件。默认是 `True` 。
- `allow_folders` - 指定是否应包括指定位置的文件夹。默认是 `False` 。它和

`allow_files`必须有一个为 `True`。

IPAddressField

确保输入是有效的IPv4或IPv6的字符串。

对应于 `django.forms.fields.IPAddressField` 和 `django.forms.fields.GenericIPAddressField` 。

签名: `IPAddressField(protocol='both', unpack_ipv4=False, **options)`

- `protocol` ：接受协议的类型。可接受的值是“both”（默认）、“IPv4”或“IPv6”。匹配不区
- `unpack_ipv4` ：解包IPv4映射地址，如::ffff:192.0.2.1。如果启用此选项，则该地址将解压缩到192.0.2.1。默认为禁用。只能在协议设置为“both”时使用。

四、数字类型字段

IntegerField

整数。

对应于 `django.db.models.fields.IntegerField` ， `django.db.models.fields.SmallIntegerField` ， `django.db.models.fields.PositiveIntegerField` 和 `django.db.models.fields.PositiveSmallIntegerField` 。

签名: `IntegerField(max_value=None, min_value=None)`

- `max_value` 允许的最大值。
- `min_value` 允许的最小值。

FloatField

浮点数。对 应 于 **签名**:

- `django.db.models.fields.FloatField`

  `FloatField(max_value=None, min_value=None)`

- `min_value` 验证提供的数字是否不低于此值。

`max_value` 验证提供的数字不大于此值。

DecimalField

科学十进制表示，Python的 `Decimal` 实例。

对应于 `django.db.models.fields.DecimalField` 。

签 名 ： `DecimalField(max_digits, decimal_places, coerce to string=None,`

- `max_digits` 数字中允许的最大位数。它必须是 `None` 或大于或等于 `decimal_places` 的整数。
- `decimal_places` 与数字一起存储的小数位数。
- `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。
- `localize`：默认为 `False`。设置为 `True`，则根据当前区域设置启用输入和输出的本地化。这也将迫使 `coerce_to_string` 设置为 `True` 。请注意，如果已在设置文件中进行了 `USE_L10N=True` 设置，则会启用数据格式设置。
- `rounding`：设置量化到配置精度时使用的舍入模式。默认为 `None` 。

五、日期和时间类型字段

DateTimeField

日期和时间表示。

对应于 `django.db.models.fields.DateTimeField` 。

签名： `DateTimeField(format=api_settings.DATETIME_FORMAT, input_formats=None, default_timezone=None)`

- `format` - 时间字符串的格式。如果未指定，则默认为settings中的`DATETIME_FORMAT`设置。
- `input_formats` - 用于解析日期的输入格式的字符串列表。如果未指定，`DATETIME_INPUT_FORMATS`将使用默认设置`['iso-8601']`。
- `default_timezone` - 默认时区

使用 `ModelSerializer` 或 `HyperlinkedModelSerializer` 序列化器时，请注意，带有 `auto_now=True` 或 `auto_now_add=True` 的模型字段将自动设置 `read_only=True` 参数。如果要覆盖此行为，则需要在序列化类中显式声明一个 `DateTimeField` 字段。例如：

```
class  CommentSerializer(serializers.ModelSerializer):
    created = serializers.DateTimeField()

    class Meta:
        model = Comment
```

DateField

日期类型字段。

对应于 `django.db.models.fields.DateField`

签名： `DateField(format=api_settings.DATE_FORMAT, input_formats=None)`

TimeField

时间类型字段。

对应于 `django.db.models.fields.TimeField`

签名： `TimeField(format=api_settings.TIME_FORMAT, input_formats=None)`

DurationField

持续时间长度类型的字段。

对应于 `django.db.models.fields.DurationField`

如果你定义了这种类型的字段，那么在 `validated_data` 变量中将包含一个 `datetime.timedelta` 实例，以 `'[DD] [HH:[MM:]]ss[.uuuuuu]'` 格式的字符串形式表示。

签名： `DurationField(max_value=None, min_value=None)`

- `max_value` 验证提供的持续时间不大于此值。
- `min_value` 验证提供的持续时间不小于此值。

六、选择类型字段

ChoiceField

接受有限选择集中的值的字段。

如果相应的模型字段包含 `choices=…` 参数，则 `ModelSerializer` 会自动生成一个对应的字段。

签名： `ChoiceField(choices)`

- `choices` - 有效值列表或 `(key, display_name)` 元组列表。
- `allow_blank` - 如果设置为 `True` 则应将空字符串视为有效值。如果设置为 `False` 则空字符串被视为无效并将引发验证错误。默认为 `False`。
- `html_cutoff` - 如果设置，这将是HTML下拉列表标签将显示的最大选择数。
- `html_cutoff_text` - 如果设置，如果在HTML选择下拉列表中截断了最大项目数，则将显示文本指示符。默认为 `"More than {count} items…"`

`allow_blank` 与 `allow_null` 两个参数都可以用于 `ChoiceField` 字段，但建议只使用一个，而不是同时用两个。 `allow_blank` 应是文本选择的首选， `allow_null` 对应数字或其他非文本选择。

MultipleChoiceField

可多选的类型字段。

可以接受零个、一个或多个值的字段。 `to_internal_value` 方法可以返回包含所选值的集合。

签名： `MultipleChoiceField(choices)`

- `choices` - 有效值列表或 `(key, display_name)` 元组列表，必填参数。
- `allow_blank` - 如果设置为 `True` 则应将空字符串视为有效值。如果设置为 `False` 则空字符串被视为无效并将引发验证错误。默认为 `False`。
- `html_cutoff` - 同上

- `html_cutoff_text` - 同上

正如 `ChoiceField` , `allow_blank` 和 `allow_null` 只能二选一。

七、文件和图片字段

`FileField` 和 `ImageField` 字段仅适用于使用 `MultiPartParser` 或 `FileUploadParser` 解析器的情况。大多数解析器（例如JSON）不支持文件上载。Django原生的FILE_UPLOAD_HANDLERS用于上传文件的处理。

FileField

文件字段。执行Django标准的FileField验证。

对应于 `django.forms.fields.FileField` 。

签名： `FileField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)`

- `max_length` - 指定文件名的最大长度。
- `allow_empty_file` - 指定是否允许空文件。
- `use_url` - 如果设置为 `True` ，则URL字符串将用于输出表示。如果设置为 `False` ，则文件名字符串值将用于输出表示。默认为settings中 `UPLOADED_FILES_USE_URL` 设置的值，除非另有设置。

ImageField

图像字段。将上载的文件内容验证为与已知图像格式匹配。

对应于 `django.forms.fields.ImageField` 。

签 名 ： `ImageField(max_length=None, allow empty file=False,`

- `max_length` - 指定文件名的最大长度。
- `allow_empty_file` - 指定是否允许空文
- 件。`use_url` - 同上

`PIL` `Pillow` `PIL`

使用此字段需要提前安装 `Pillow` 包或　　包。建议使用　　，因为　　不再维护。

八、复合类型字段

ListField

对象列表的字段类型。

签名： `ListField(child=<A_FIELD_INSTANCE>, min length=None,`

- `child` - 用于验证列表中对象的字段实例。如果未提供此参数，则不会进行验证。
- `min_length` - 列表包含的元素数量不少于该值。
- `max_length` - 列表包含的元素个数不超过该值。

例如，要验证整数列表，可以使用以下内容：

```
1   scores = serializers.ListField(
2   child=serializers.IntegerField(min_value=0, max_value=100) 3 )
```

`ListField` 还支持编写可重用的列表字段类。

```
1   class StringListField(serializers.ListField):
2       child = serializers.CharField()
```

现在可以在整个应用程序中重用我们自定义的 `StringListField` 类，而无需为其提供 `child` 参数。

DictField

对象字典的字段类。 `DictField` 中的键始终假定为字符串值。

签名： `DictField(child=<A_FIELD_INSTANCE>)`

- `child` - 同上，但对元素个数没有要求

例如，要创建一个验证字符串到字符串的映射的字段，可以编写如下内容：

```
1   document = DictField(child=CharField())
```

也可以自定义字段类。例如：

```
class DocumentField(DictField):
    child = CharField()
```

HStoreField

预配置的 `DictField` 。与Django的postgres `HStoreField` 兼容。

签名： `HStoreField(child=<A_FIELD_INSTANCE>)`

- `child` - 用于验证字典中的值的字段实例。默认子字段接受空字符串和空值。

请注意，子字段**必须**是 `CharField` 实例。

JSONField

json字段类，用于验证传入数据是否有效的JSON结构。

签名： `JSONField(binary)`

- `binary` - 如果设置为 `True` 则该字段将输出并验证JSON编码的字符串，而不是原始数据结构。默认为 `False` 。

九、其他类型字段

ReadOnlyField

自读类型，它只返回字段的值而不进行修改。

默认情况下，此字段用于 `ModelSerializer` 包含与属性相关的字段名称而不是模型字段。

签名： `ReadOnlyField()`

例如，如果 `has_expired` 是 `Account` 模型上的属性，则以下序列化程序会自动将其生成为 `ReadOnlyField` ：

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'has_expired')
```

HiddenField

隐藏的字段，它不根据用户输入获取值，而是从默认值或可调用的值中获取值。

签名：`HiddenField()`

例如，要包含始终将当前时间作为序列化程序验证数据的一部分提供的字段：

```
modified = serializers.HiddenField(default=timezone.now)
```

ModelField

可以绑定到任意模型字段的通用字段。

此字段用于`ModelSerializer`对应于自定义模型字段类。一般我们不使用

签名：`ModelField(model_field=<Django ModelField instance>)`

SerializerMethodField

一个只读字段。它通过调用附加到的序列化类上的方法来获取值。可用于将任何类型的数据添加到序列化对象中。（项目里展示）

这相当于自定义字段数据。

签名：`SerializerMethodField(method_name=None)`

- `method_name` - 要调用的序列化程序上方法的名称。如果不包含，则默认为`get_<field_name>`。

序列化关系字段

一、序列化关系字段

关系字段用于表示模型之间的关联。在Django中存在 `ForeignKey` 、 `ManyToManyField` 和 `OneToOneField` 三种正向关系，以及反向关联和自定义关联。

注意: 在DRF框架，关系型字段的代码虽然定义在 `relations.py`模块中，但我们通常 `serializers` 从模块中导入关系型字段，也就是 `from rest_framework import serializers` ，然后使用 `serializers.<FieldName>` 的方式引用。

当继承 `ModelSerializer` 类的时候，包括关系型字段在内的所有字段会自动生成。我们可以查看这些字段。使用 `python manage.py shell` 命令，进入Django的shell环境，然后按下面的操作查看字段信息：

```
>>> from myapp.serializers import AccountSerializer
>>> serializer = AccountSerializer()
>>> print(repr(serializer))
AccountSerializer():
    id = IntegerField(label='ID', read_only=True)
    name = CharField(allow_blank=True, max_length=100, required=False)
    owner = PrimaryKeyRelatedField(queryset=User.objects.all())
```

二、API参考

为了解释不同类型的关系字段，我们使用下面的一组模型作为例子。Album是唱片，Track是每张唱片里的某首歌曲。

```
class Album(models.Model):
    album_name = models.CharField(max_length=100)  #唱片名字
    artist = models.CharField(max_length=100)  #唱片作者

    def __str__(self):
        return self.album_name

class Track(models.Model):
    album = models.ForeignKey(Album, related_name='tracks', on_delete=models.CASCADE)  #关联唱片
    order = models.IntegerField()  #歌曲的顺序
```

```
    title = models.CharField(max_length=100)  #歌曲的名字
    duration = models.IntegerField()  #歌曲的时间长度
```

StringRelatedField

`StringRelatedField` 使用对象的 `__str__` 方法来表示关联的对象。这个字段其实也就是将关联对象的字符串表示形式的信息拿来，放到自己的序列化类中，供API视图使用并渲染，然后传递给前端。

例如下面的序列化类：

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.StringRelatedField(many=True)    # 额外增加了一个字段，注意many参数

    class Meta:
        model = Album
        fields = ('album name', 'artist',  'tracks')
```

上面的操作，会获得下面的结果，注意'tracks'键：

```
{
    'album_name': 'Things We Lost In  The Fire',
    'artist': 'Low',
    'tracks': [
        '1: Sunflower',
        '2: Whitetail',
        '3: Dinosaur Act',
        ...
    ]
}
```

注意：该字段是只读的！

StringRelatedField**的参数**:

- `many` - 如果关联的是一个复数数量的对象，必须将此参数设置为 `True` 。

PrimaryKeyRelatedField

`PrimaryKeyRelatedField` 使用关联对象的主键id值来表示对象。

例如:

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True) # 一定要注意参数

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

序列化的结果如下，注意那些数字:

```
{
    'album_name': 'Undun',
    'artist': 'The Roots',
    'tracks': [
        89,
        90,
        91,
        ...
    ]
}
```

默认情况下，这种字段关联方式是可读可写的。可以通过添加 `read_only=True` 标识，变成只读!

PrimaryKeyRelatedField**参数**:

- `queryset` - 当对字段的输入数据进行验证的时候，用于查询模型实例的查询集。必须显式的提供这个参数，或者设置 `read_only=True` 。
- `many` - 如果关联的是一个复数数量的对象，必须将此参数设置为 `True` 。
- `allow_null` - 默认为False。如果设置 `True` ，在可空的关联上，该字段将可以接收为 `None` 或空字符串。
- `pk_field` - 指定序列化/反序列化过程中，使用的主键字段类型。例如，`pk_field=UUIDField(format='hex')` 将序列化一个UUID主键值。

HyperlinkedRelatedField

`HyperlinkedRelatedField` 使用关联对象的超链接形式来标识关联对象。也就是说，不使用字符串，也不用主键id数字，而是用一个可以点击跳转的url地址来表示关联的对象。

例如：

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks =  serializers.HyperlinkedRelatedField(
        many=True,
        read_only=True,
        view_name='track-detail'
    )  # 注意参数

    class Meta:
        model = Album
        fields = ('album name', 'artist',  'tracks')
```

序列化的结果;

```
{
    'album_name': 'Graceland',
    'artist': 'Paul Simon',
    'tracks': [
        'http://www.example.com/api/tracks/45/',
        'http://www.example.com/api/tracks/46/',
        'http://www.example.com/api/tracks/47/',
        ...
    ]
}
```

默认情况下，这种字段关联方式是可读可写的。可以通过添加 `read_only=True` 标识，变成只读!

注意：此字段是为那些映射到URL的对象设计的，该URL接受使用`lookup_field`和`lookup_url_kwarg`参数设置的单个URL关键字参数。

这种关系字段适合包含单独主键或者slug参数的URLs。

如果你需要更复杂的超链接表示形式，参考下面的自定义关系字段章节。

参数:

- `view_name` - 处理关联对象URL的视图。如果你使用的是标准的路由类，它必须是一个 **`<modelname>-detail`** 格式的字符串。此参数必填!

- `queryset` - 当对字段的输入数据进行验证的时候，用于查询模型实例的查询集。必须显式的提供这个参数，或者设置 `read_only=True` 。
- `many` - 如果关联的是一个复数数量的对象，必须将此参数设置为 `True` 。
- `allow_null` -默认为False。如果设置为 `True` ，在可空的关联上，该字段将可以接收 `None` 或空字符串。
- `lookup_field` - 该参数的默认值为 `'pk'` ，表示用主键id在视图种查找关联的对象，一般我们不需要修改这个参数。它必须和对应视图的URL关键字参数一致，你这里如果改了，在视图中也必须跟着改。
- `lookup_url_kwarg` - 指定上面参数值的名字，一般使用 `lookup_field` 。大多数情况下，我们不用修改这个参数，默认就好，除非你需要自定义一大堆，不要给自己挖坑。
- `format` - 如果URL种使用了格式后缀，超链接字段将使用同样的格式后缀，除非使用这个 `format` 参数另外指定后缀形式。

SlugRelatedField

`SlugRelatedField` 使用某个指定的字段的值作为关联对象的表示形式。比如拿对象的名字、或者邮箱、或者昵称、或者地址等等。

例如：

```
1   class AlbumSerializer(serializers.ModelSerializer):
2       tracks = serializers.SlugRelatedField(
3           many=True,
4           read_only=True,
5           slug_field='title'
6        )
7
8       class Meta:
9           model = Album
10          fields = ('album_name', 'artist',  'tracks')
```

序列化的结果

```
{
    'album_name': 'Dear John',
    'artist': 'Loney Dear',
    'tracks': [
        'Airport Surroundings',
        'Everything Turns to You',
        'I Was Only Going Out',
        ...
    ]
}
```

默认情况下，这种字段关联方式是可读可写的。可以通过添加 `read_only=True` 标识，变成只读!

如果 `SlugRelatedField` 作为一个可读写的字段，那么你必须确保对应的模型种的字段必须是 `unique=True` ，否则你用什么来区分是哪个对象?

参数:

- `slug_field` - 指定用来表示关联对象的字段，该参数必填。
- `queryset` - 同上
- `many` - 同上
- `allow_null` - 同上

HyperlinkedIdentityField

这种字段可以用作身份关联，使用较少。

```
class AlbumSerializer(serializers.HyperlinkedModelSerializer): # 注意继承的类变了
    track_listing = serializers.HyperlinkedIdentityField(view_name='track-list')

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'track_listing')
```

序列化结果:

```
{
    'album_name': 'The Eraser',
    'artist': 'Thom Yorke',
    'track_listing': 'http://www.example.com/api/track_list/12/',
}
```

这种字段只读!

参数:

- `view_name` - 同前。必填
- `lookup_field` - 同前
- `lookup_url_kwarg` - 同前
- `format` - 同前

三、嵌套关联

可以将序列化类作为字段，来表示嵌套关联。

如果关联的是一个复数数量的对象，必须将 `many` 参数设置为 `True` 。

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)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

序列化的过程例子如下:

```
>>> album = Album.objects.create(album_name="The Grey Album",
artist='Danger Mouse')
>>> Track.objects.create(album=album, order=1, title='Public Service
Announcement', duration=245)
<Track: Track object>
>>> Track.objects.create(album=album, order=2, title='What More Can I Say',
duration=264)
```

```
<Track: Track object>
>>> Track.objects.create(album=album, order=3, title='Encore',
duration=159)
<Track: Track object>
>>> serializer = AlbumSerializer(instance=album)
>>> serializer.data
{
    'album_name': 'The Grey Album',
    'artist': 'Danger Mouse',
    'tracks': [
        {'order': 1, 'title': 'Public Service Announcement',  'duration':
245},
        {'order': 2, 'title': 'What More Can I Say', 'duration':  264},
        {'order': 3, 'title': 'Encore', 'duration': 159},
        ...
    ],
}
```

默认情况下，嵌套关联是只读的！如果你想设置可读写的嵌套关联，你必须自己实现，`create()` 与/或 `update()` 方法，显式地指定如何保存子关系。例如：

```
class TrackSerializer(serializers.ModelSerializer):
    class Meta:
        model = Track
        fields = ('order', 'title', 'duration')

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackSerializer(many=True)（这里没有
    readonly）

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')

    def create(self, validated_data):
        tracks_data = validated_data.pop('tracks')
        album = Album.objects.create(**validated_data)
        for track_data in tracks_data:
            Track.objects.create(album=album, **track_data)
        return album

>>> data = {
    'album_name': 'The Grey Album',
    'artist': 'Danger Mouse'
```

```
            {'order': 1, 'title': 'Public Service Announcement', 'duration':
    245},
            {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
            {'order': 3, 'title': 'Encore', 'duration': 159},
        ],
    }
    >>> serializer =  AlbumSerializer(data=data)
    >>> serializer.is_valid()
    True
    >>>  serializer.save()
    <Album: Album object>
```

四、自定义关系类型字段

在很少的情况下，如果现有的关系样式都不适合所需要的表示，那么你可以实现一个完全自定义的关系字段，该字段准确描述了应该如何从模型实例生成输出表示。

要自定义关系类型字段，你必须先继承 `RelatedField` 类，然后实现 `.to_representation(self, value)` 方法。此方法将字段的目标作为“value”参数，并应返回对象序列化后的表示形式。“value”参数通常是模型实例。

如果你想实现可读写的关系类型字段，你还需要实现 `.to_internal_value(self, data)` 方法.

如果想提供一个基于 `context` 的动态查询集，你需要覆盖 `.get_queryset(self)` 方法。

例如：

```
import time

class  TrackListingField(serializers.RelatedField):
    def to_representation(self,  value):  # 看这里
        duration = time.strftime('%M:%S', time.gmtime(value.duration))
        return 'Track %d: %s (%s)' % (value.order, value.name, duration)

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackListingField(many=True)

    class Meta:
        model = Album
        fields = ('album name', 'artist', 'tracks')
```

自定义字段的序列化结果：

```
{
    'album_name': 'Sometimes I Wish We Were an Eagle',
    'artist': 'Bill Callahan',
    'tracks': [
        'Track 1: Jim Cain (04:39)',
        'Track 2: Eid Ma Clack Shaw (04:19)',
        'Track 3: The Wind and the Dove  (04:34)',
        ...
    ]
}
```

五、自定义超链接字段

首先需要继承HyperlinkedRelatedField`类，然后根据需要，选择性地覆写下面两个方法：

- get_url(self, obj, view_name, request, format)

此方法用于对象实例和它的URL表示之间的映射。

- get_object(self, view_name, view_args, view_kwargs)

这个方法的返回值必须和匹配的URL conf参数一致。

下面假设我们有一个顾客对象的URL，并接受两个参数：

```
/api/<organization_slug>/customers/<customer_pk>/
```

对于上面这种url，普通的超链接字段无法实现，因为超链接字段只能处理url中只有一个参数的情况，而上面有两个参数。只能自定义超链接字段，然后自己写代码处理了：

```
from rest_framework import serializers
from rest framework.reverse import  reverse

class CustomerHyperlink(serializers.HyperlinkedRelatedField):
    # 这里把下面两个变量作为类属性进行处理，这样就不需要传递参数了
    view_name = 'customer-detail'
    queryset = Customer.objects.all()

    def get_url(self, obj, view_name, request, format):
        url_kwargs = {
            'organization_slug': obj.organization.slug,
```

```
            'customer_pk': obj.pk
        }
        return reverse(view_name, kwargs=url_kwargs, request=request,
format=format)

    def get_object(self, view_name, view_args, view_kwargs):
        lookup_kwargs = {
           'organization slug': view_kwargs['organization_slug'],
        'pk': view_kwargs['customer_pk'] 20
        }
        return self.get_queryset().get(**lookup_kwargs)
```

如果你想将上面的自定义超链接字段和通用视图一起使用，那么你还需要在视图中重写 `.get_object` 方法，保持两者的一致。

六、注意事项

- `queryset` 参数

"queryset"参数仅对**可写**关系字段是必需的，在这种情况下，它用于执行从基本用户输入映射到模型实例的查找。

- 自定义HTML显示

当在HTML的可浏览API中，如果要显示包含 `choices` 属性的字段时，默认使用对象的 `__str__` 方法，也就是字符串表示形式。

如果要自定义这个形式，重写 `RelatedField` 的 `display_value()` 方法。该方法接收一个实例作为参数，并应当返回一个合适的表示形式，例如：

```
class TrackPrimaryKeyRelatedField(serializers.PrimaryKeyRelatedField):
    def display_value(self, instance):
        return 'Track: %s' % (instance.title)
```

- 反向关联

请注意，在 继承`ModelSerializer` 和 `HyperlinkedModelSerializer` 类时，不会自动生成反向关联，你必须显式地添加反向关联字段。例如：

```
class AlbumSerializer(serializers.ModelSerializer):
    class Meta:
        fields = ('tracks', ...)
```

一般情况下，我们会为反向关联设置一个 `related_name` 参数，作为反向关联时的字段名，例如：

```
class Track(models.Model):
    album = models.ForeignKey(Album, related_name='tracks',
on_delete=models.CASCADE)
    ...
```

如果没有指定，那么你只能使用Django原生给我们提供的关联名，也就是`modelname_set`：

```
class AlbumSerializer(serializers.ModelSerializer):
    class Meta:
        fields = ('track_set', ...)
```

For more information see the Django documentation on generic relations .

- 使用中间模型的ManyToManyFields

默认情况下，关联到一个使用了中间模型的ManyToManyFields的字段，需要设为只读，因此请确保为字段设置了 `read_only=True` 属性。

```
class  MySerializer(serializers.Serializer):

    extra_info = serializers.SerializerMethodField()

    def get_extra_info(self, obj): # 注意这个方法的名字,obj参数表示每一条数据库记
录
        return {
            'email': 'xxxx',
            'xxxx': 'xxxx',
            'status':obj.related_filed.some_filed,
        }
```