

Django REST framework API 指南

书栈(BookStack.CN)

目 录

致谢
介绍
请求
响应
视图
通用视图
视图集
路由
解析
渲染
序列化
序列化·字段
序列化·关系
验证器
认证
权限
限流
过滤
分页
版本控制
内容协商
元数据
Schemas
Format 后缀
返回 URL
异常
状态码
测试
Settings

致谢

当前文档《Django REST framework API 指南》由 进击的皇虫 使用 书栈 (BookStack.CN) 进行构建，生成于 2018-06-02。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈 (BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/django-rest-framework-api-guide>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

- [Django REST framework API 指南](#)
- [目录](#)
- [来源](#)

Django REST framework API 指南



文章翻译自[官方网站](#)，部分内容有增删。

没看过官网 tutorial 的建议先看一遍，也有人翻译过：[Django Rest Framework 中文版教程](#)，少了最后一篇，不过影响不大。

本人能力有限，不足之处还有很多，欢迎大家 PR。

转载请注明出处。

目录

- [请求](#)
- [响应](#)
- [视图](#)
- [通用视图](#)
- [视图集](#)
- [路由](#)
- [解析](#)
- [渲染](#)
- [序列化](#)
- [序列化·字段](#)

- [序列化 · 关系](#)
- [验证器](#)
- [认证](#)
- [权限](#)
- [限流](#)
- [过滤](#)
- [分页](#)
- [版本控制](#)
- [内容协商](#)
- [元数据](#)
- [Schemas](#)
- [Format 后缀](#)
- [返回 URL](#)
- [异常](#)
- [状态码](#)
- [测试](#)
- [Settings](#)

来源

<https://github.com/jianshijiuyou/django-rest-framework-api-guide>

请求

- [Request 解析](#)
 - [.data](#)
 - [.query_params](#)
 - [.parsers](#)
- [内容协商](#)
 - [.accepted_renderer](#)
 - [.accepted_media_type](#)
- [认证 \(Authentication\)](#)
 - [.user](#)
 - [.auth](#)
 - [.authenticators](#)
- [浏览器增强](#)
 - [.method](#)
 - [.content_type](#)
 - [.stream](#)
- [标准的 HttpRequest 属性](#)

官方原文链接

如果你正在开发基于 *REST* 的 *web API* 服务..... 应该忽略 *request.POST*。

— *Malcom Tredinnick, Django* 开发组

REST framework 的 `Request` 类扩展与标准的 `HttpRequest`，并做了相应的增强，比如更加灵活的请求解析 (*request parsing*) 和认证 (*request authentication*)。

Request 解析

REST framework 的 `Request` 对象提供了灵活的请求解析，允许你使用 *JSON data* 或 其他 *media types* 像通常处理表单数据一样处理请求。

.data

`request.data` 返回请求主题的解析内容。这跟标准的 `request.POST` 和 `request.FILES` 类似，并且还具有以下特点：

- 包括所有解析的内容，文件 (*file*) 和 非文件 (*non-file inputs*)。
- 支持解析 `POST` 以外的 *HTTP method*，比如 `PUT`，`PATCH`。
- 更加灵活，不仅仅支持表单数据，传入同样的 *JSON 数据* 一样可以正确解析，并且不用做额外的处理（意思是前端不管提交的是表单数据，还是 *JSON 数据*，`.data` 都能够正确解析）。

`.data` 具体操作，以后再说 ~

`.query_params`

`request.query_params` 等同于 `request.GET`，不过其名字更加容易理解。

为了代码更加清晰可读，推荐使用 `request.query_params`，而不是 Django 中的 `request.GET`，这样能够让你的代码更加明显的体现出 —— 任何 HTTP method 类型都可能包含查询参数 (query parameters)，而不仅仅只是 ‘GET’ 请求。

`.parsers`

`APIView` 类或者 `@api_view` 装饰器将根据视图上设置的 `parser_classes` 或 `settings` 文件中的 `DEFAULT_PARSER_CLASSES` 设置来确保此属性 (`.parsers`) 自动设置为 `Parser` 实例列表。

通常不需要关注该属性.....

如果你非要看看它里面是什么，可以打印出来看看，大概长这样：

```
1. [<rest_framework.parsers.JSONParser object at 0x7fa850202d68>,
    <rest_framework.parsers.FormParser object at 0x7fa850202be0>,
    <rest_framework.parsers.MultiPartParser object at 0x7fa850202860>]
```

恩，包含三个解析器 `JSONParser`，`FormParser`，`MultiPartParser`。

注意：如果客户端发送格式错误的内容，则访问 `request.data` 可能会引发 `ParseError`。默认情况下，`REST framework` 的 `APIView` 类或者 `@api_view` 装饰器将捕获错误并返回 `400 Bad Request` 响应。

如果客户端发送的请求内容无法解析（不同于格式错误），则会引发 `UnsupportedMediaType` 异常，默认情况下会被捕获并返回 `415 Unsupported Media Type` 响应。

内容协商

该请求公开了一些属性，允许你确定内容协商阶段的结果。这使你可以实施一些行为，例如为不同媒体类型选择不同的序列化方案。

`.accepted_renderer`

渲染器实例是由内容协商阶段选择的。

`.accepted_media_type`

表示内容协商阶段接受的 media type 的字符串。

认证 (Authentication)

REST framework 提供了灵活的认证方式：

- 可以在 API 的不同部分使用不同的认证策略。
- 支持同时使用多个身份验证策略。
- 提供与传入请求关联的用户 (user) 和令牌 (token) 信息。

.user

`request.user` 通常会返回 `django.contrib.auth.models.User` 的一个实例，但其行为取决于正在使用的身份验证策略。

如果请求未经身份验证，则 `request.user` 的默认值是 `django.contrib.auth.models.AnonymousUser` 的实例（就是匿名用户）。

关于 `.user` 的更多内容，以后再说 ~

.auth

`request.auth` 返回任何附加的认证上下文 (authentication context)。`request.auth` 的确切行为取决于正在使用的身份验证策略，但它通常可能是请求经过身份验证的令牌 (token) 实例。

如果请求未经身份验证，或者没有附加上下文 (context)，则 `request.auth` 的默认值为 `None`。

关于 `.auth` 的更多内容，以后再说 ~

.authenticators

`APIView` 类或 `@api_view` 装饰器将确保根据视图上设置的 `authentication_classes` 或基于 `settings` 文件中的 `DEFAULT_AUTHENTICATORS` 设置将此属性 (`.authenticators`) 自动设置为 `Authentication` 实例列表。

通常不需要关注该属性.....

注意：调用 `.user` 或 `.auth` 属性时可能会引发 `WrappedAttributeError` 异常。这些错误源于 `authenticator` 作为一个标准的 `AttributeError`，为了防止它们被外部属性访问修改，有必要重新提升为不同的异常类型。Python 无法识别来自 `authenticator` 的 `AttributeError`，并会立即假定请求对象没有 `.user` 或 `.auth` 属性。`authenticator` 需要修复。

请求

多说几句

`.authenticators` 其实存的就是当前使用的认证器 (authenticator) 列表, 打印出来大概是这样:

```
1. [<rest_framework.authentication.SessionAuthentication object at 0x7f8ae4528710>, <rest_framework.authentication.BasicAuthentication object at 0x7f8ae45286d8>]
```

可以看到这里使用的认证器 (authenticator) 包括 `SessionAuthentication` 和 `BasicAuthentication`。

浏览器增强

REST framework 支持基于浏览器的 `PUT`, `PATCH`, `DELETE` 表单。

.method

`request.method` 返回请求 HTTP 方法的大写字符串表示形式。如 `GET`, `POST` ...。

透明地支持基于浏览器的 `PUT`, `PATCH` 和 `DELETE` 表单。

更多相关信息以后再说 ~

.content_type

`request.content_type` 返回表示 HTTP 请求正文的媒体类型 (media type) 的字符串对象 (比如: `text/plain`, `text/html` 等), 如果没有提供媒体类型, 则返回空字符串。

通常不需要直接访问此属性, 一般都依赖与 REST 框架的默认请求解析行为。

不建议使用 `request.META.get('HTTP_CONTENT_TYPE')` 来获取 content type。

更多相关信息以后再说 ~

.stream

`request.stream` 返回一个代表请求主体内容的流。

通常不需要直接访问此属性, 一般都依赖与 REST 框架的默认请求解析行为。

标准的 HttpRequest 属性

由于 REST framework 的 `Request` 扩展于 Django 的 `HttpRequest`，所有其他标准属性和方法也可用。例如 `request.META` 和 `request.session` 字典都可以正常使用。

请注意，由于实现原因，`Request` 类不会从 `HttpRequest` 类继承，而是使用组合扩展类（优先使用组合，而非继承，恩，老铁没毛病 0.0）

响应

- [Responses](#)
- 创建 response
 - [Response\(\)](#)
- 属性
 - [.data](#)
 - [.status_code](#)
 - [.content](#)
 - [.template_name](#)
 - [.accepted_renderer](#)
 - [.accepted_media_type](#)
 - [.renderer_context](#)
- 标准 [HttpResponse](#) 属性
 - [.render\(\)](#)

[官方原文链接](#)

Responses

与基本的 `HttpResponse` 对象不同，`TemplateResponse` 对象保留了视图提供的用于计算响应的上下文的详细信息。直到需要时才会计算最终的响应输出，也就是在后面的响应过程中进行计算。

– Django 文档

REST framework 通过提供一个 `Response` 类来支持 HTTP 内容协商，该类允许你根据客户端请求返回不同的表现形式（如：JSON，HTML 等）。

`Response` 类的子类是 Django 的 `SimpleTemplateResponse`。`Response` 对象使用数据进行初始化，数据应由 Python 对象（native Python primitives）组成。然后 REST framework 使用标准的 HTTP 内容协商来确定它应该如何渲染最终响应的内容。

当然，您也可以不使用 `Response` 类，直接返回常规 `HttpResponse` 或 `StreamingHttpResponse` 对象。使用 `Response` 类只是提供了一个更好的交互方式，它可以返回多种格式。

除非由于某种原因需要大幅度定制 REST framework，否则应该始终对返回 `Response` 对象的视图使用 `APIView` 类或 `@api_view` 装饰器。这样做可以确保视图执行内容协商，并在视图返回之前为响应选择适当的渲染器。

创建 response

Response()

与普通 `HttpResponse` 对象不同，您不会使用渲染的内容实例化 `Response` 对象。相反，您传递的是未渲染的数据，可能包含任何 Python 对象。

由于 `Response` 类使用的渲染器不能处理复杂的数据类型（比如 Django 的模型实例），所以需要在创建 `Response` 对象之前将数据序列化为基本的数据类型。

您可以使用 REST framework 的 `Serializer` 类来执行序列化的操作，也可以用自己的方式来序列化。

构造方法：`Response(data, status=None, template_name=None, headers=None, content_type=None)`

参数：

- `data`：响应的序列化数据。
- `status`：响应的状态代码。默认为200。
- `template_name`：选择 `HTMLRenderer` 时使用的模板名称。
- `headers`：设置 HTTP header，字典类型。
- `content_type`：响应的内容类型，通常渲染器会根据内容协商的结果自动设置，但有些时候需要手动指定。

属性

`.data`

还没有渲染，但已经序列化的响应数据。

`.status_code`

状态码

`.content`

将会返回的响应内容，必须先调用 `.render()` 方法，才能访问 `.content`。

`.template_name`

只有在 response 的渲染器是 `HTMLRenderer` 或其他自定义模板渲染器时才需要提供。

`.accepted_renderer`

用于将会返回的响应内容的渲染器实例。

从视图返回响应之前由 `APIView` 或 `@api_view` 自动设置。

`.accepted_media_type`

内容协商阶段选择的媒体类型。

从视图返回响应之前由 `APIView` 或 `@api_view` 自动设置。

`.renderer_context`

将传递给渲染器的 `.render()` 方法的附加的上下文信息字典。

从视图返回响应之前由 `APIView` 或 `@api_view` 自动设置。

标准 `HttpResponse` 属性

`Response` 类扩展于 `SimpleTemplateResponse`，并且响应中也提供了所有常用的属性和方法。例如，您可以用标准方式在响应中设置 `header`：

```
1. response = Response()
2. response['Cache-Control'] = 'no-cache'
```

`.render()`

与其他任何 `TemplateResponse` 一样，调用此方法将响应的序列化数据呈现为最终响应内容。响应内容将设置为在 `accepted_renderer` 实例上调用 `.render(data, accepted_media_type, renderer_context)` 方法的结果。

通常不需要自己调用 `.render()`，因为它是由 Django 处理的。

视图

- 基于类的视图
 - API 策略属性 (policy attributes)
 - API 策略实例方法 (policy instantiation methods)
 - API 策略实现方法 (policy implementation methods)
 - 方法调度
- 基于方法的视图
 - `@api_view()`
 - API 策略装饰器 (policy decorators)
 - View schema 装饰器

[官方原文链接](#)

基于类的视图

REST framework 提供了一个 `APIView` 类，它继承于 Django 的 `View` 类。

`APIView` 类与不同的 `View` 类有所不同：

- 传递给处理方法的 request 对象是 REST framework 的 `Request` 实例，而不是 Django 的 `HttpRequest` 实例。
- 处理方法可能返回 REST framework 的 `Response`，而不是 Django 的 `HttpResponse`。该视图将管理内容协商，并在响应中设置正确的渲染器。
- 任何 `APIException` 异常都会被捕获并进行适当的响应。
- 传入的请求会进行认证，在请求分派给处理方法之前将进行适当的权限检查（允许或限制）。

像往常一样，使用 `APIView` 类与使用常规 `View` 类非常相似，传入的请求被分派到适当的处理方法，如 `.get()` 或 `.post()`。此外，可以在类上设置许多属性（AOP）。

举个栗子：

```
1. from rest_framework.views import APIView
2. from rest_framework.response import Response
3. from rest_framework import authentication, permissions
4. from django.contrib.auth.models import User
5.
6. class ListUsers(APIView):
7.     """
8.     列出系统中的所有用户
9.
```

```

10.     * 需要 token 认证。
11.     * 只有 admin 用户才能访问此视图。
12.     """
13.     authentication_classes = (authentication.TokenAuthentication,)
14.     permission_classes = (permissions.IsAdminUser,)
15.
16.     def get(self, request, format=None):
17.         """
18.         Return a list of all users.
19.         """
20.         usernames = [user.username for user in User.objects.all()]
21.         return Response(usernames)

```

注意: *REST Framework* 中的 `APIView`, `GenericAPIView`, 各种 `Mixins` 和 `Viewsets` 包含许多方法和属性, 刚开始要全部理解是比较困难的。这里除了文档, 有一个 [Classy Django REST Framework](#) 资源, 它提供了一个可以在线浏览的参照, 包含所有属性和方法。

API 策略属性 (policy attributes)

以下属性用于增加扩展视图的功能, AOP。

`.renderer_classes`

设置渲染器

`.parser_classes`

设置解析器

`.authentication_classes`

设置认证器

`.throttle_classes`

`.permission_classes`

设置权限验证器

`.content_negotiation_class`

API 策略实例方法 (policy instantiation methods)

以下策略实例方法通常不需要我们重写。

`.get_renderers(self)`

`.get_parsers(self)`

视图

`.get_authenticators(self)`

`.get_throttles(self)`

`.get_permissions(self)`

`.get_content_negotiator(self)`

`.get_exception_handler(self)`

API 策略实现方法 (policy implementation methods)

在分派到处理方法之前会调用以下方法。

`.check_permissions(self, request)`

`.check_throttles(self, request)`

`.perform_content_negotiation(self, request, force=False)`

方法调度

以下方法由视图的 `.dispatch()` 方法直接调用。它们在调用处理方法 (`.get()`, `.post()`, `put()`, `patch()` 和 `.delete()`) 之前或者之后被调用。

`.initial(self, request, *args, **kwargs)`

用于执行处理方法被调用之前需要的任何操作。此方法用于强制执行权限和限流，并执行内容协商。

`.handle_exception(self, exc)`

处理方法抛出的任何异常都将传递给此方法，该方法返回一个 `Response` 实例，或者重新引发异常。

默认实现处理 `rest_framework.exceptions.APIException` 的任何子类，以及 Django 的 `Http404` 和 `PermissionDenied` 异常，并返回相应的错误响应。

如果需要自定义 API 返回的错误响应，应该重写此方法。

`.initialize_request(self, request, *args, **kwargs)`

确保传递给处理方法的请求对象是 `Request` 的一个实例，而不是通常的 Django `HttpRequest`。

通常不需要重写此方法。

`.finalize_response(self, request, response, *args, **kwargs)`

确保从处理方法返回的任何 `Response` 对象将被呈现为正确的内容类型，这由内容协商确定。

通常不需要重写此方法。

基于方法的视图

REST framework 也允许使用基于函数的视图。它提供了一套简单的装饰器来包装你的函数视图，以确保它们接收 `Request`（而不是 Django `HttpRequest`）实例并允许它们返回 `Response`（而不是 Django `HttpResponse`），并允许你配置该请求的处理方式。

@api_view()

签名: `@api_view(http_method_names=['GET'])`

`api_view` 是一个装饰器，用 `http_method_names` 来设置视图允许响应的 HTTP 方法列表，举个例子，编写一个简单的视图，手动返回一些数据。

```
1. from rest_framework.decorators import api_view
2.
3. @api_view()
4. def hello_world(request):
5.     return Response({"message": "Hello, world!"})
```

该视图将使用 `settings` 中指定的默认渲染器，解析器，认证类等。

默认情况下，只有 `GET` 方法会被接受。其他方法将以 `"405 Method Not Allowed"` 进行响应。要改变这种行为，请指定视图允许的方法，如下所示：

```
1. @api_view(['GET', 'POST'])
2. def hello_world(request):
3.     if request.method == 'POST':
4.         return Response({"message": "Got some data!", "data": request.data})
5.     return Response({"message": "Hello, world!"})
```

API 策略装饰器 (policy decorators)

为了覆盖默认设置，REST framework 提供了一系列可以添加到视图中的附加装饰器。这些必须在 `@api_view` 装饰器之后（下方）。例如，要创建一个使用 `throttle` 来确保它每天只能由特定用户调用一次的视图，请使用 `@throttle_classes` 装饰器，传递一个 `throttle` 类列表：

```
1. from rest_framework.decorators import api_view, throttle_classes
```

```

2. from rest_framework.throttling import UserRateThrottle
3.
4. class OncePerDayUserThrottle(UserRateThrottle):
5.     rate = '1/day'
6.
7. @api_view(['GET'])
8. @throttle_classes([OncePerDayUserThrottle])
9. def view(request):
10.     return Response({"message": "Hello for today! See you tomorrow!"})

```

这些装饰器对应于 `APIView` 上设置的策略属性。

可用的装饰器有：

- `@renderer_classes(...)`
- `@parser_classes(...)`
- `@authentication_classes(...)`
- `@throttle_classes(...)`
- `@permission_classes(...)`

每个装饰器都有一个参数，它必须是一个类列表或者一个类元组。

View schema 装饰器

要覆盖函数视图的默认 模式生成 (schema generation)，可以使用 `@schema` 装饰器。这必须在 `@api_view` 装饰器之后（下方）。例如：

```

1. from rest_framework.decorators import api_view, schema
2. from rest_framework.schemas import AutoSchema
3.
4. class CustomAutoSchema(AutoSchema):
5.     def get_link(self, path, method, base_url):
6.         # override view introspection here...
7.
8. @api_view(['GET'])
9. @schema(CustomAutoSchema())
10. def view(request):
11.     return Response({"message": "Hello for today! See you tomorrow!"})

```

该装饰器将采用一个 `AutoSchema` 实例，一个 `AutoSchema` 子类实例或 `ManualSchema` 实例，如 [Schemas 文档](#)（先放官链）中所述。您也可以传 `None` 以从 模式生成 (schema generation) 中排除视图。

```
1. @api_view(['GET'])
2. @schema(None)
3. def view(request):
4.     return Response({"message": "Will not appear in schema!"})
```

通用视图

- [通用视图](#)
 - [举个栗子](#)
- [API 参考](#)
 - [GenericAPIView](#)
 - [属性](#)
 - [基本设置](#)
 - [分页](#)
 - [过滤](#)
 - [方法](#)
 - [基本方法](#)
 - [保存和删除钩子 \(hook\)](#)
 - [其他方法](#)
- [Mixins](#)
 - [ListModelMixin](#)
 - [CreateModelMixin](#)
 - [RetrieveModelMixin](#)
 - [UpdateModelMixin](#)
 - [DestroyModelMixin](#)
- [内置视图类列表](#)
 - [CreateAPIView](#)
 - [ListAPIView](#)
 - [RetrieveAPIView](#)
 - [DestroyAPIView](#)
 - [UpdateAPIView](#)
 - [ListCreateAPIView](#)
 - [RetrieveUpdateAPIView](#)
 - [RetrieveDestroyAPIView](#)
 - [RetrieveUpdateDestroyAPIView](#)
- [自定义通用视图类](#)
 - [自定义 mixins](#)
 - [自定义基类](#)

[官方原文链接](#)

通用视图

基于类的视图的一个主要优点是它们允许你编写可重复使用的行为。 REST framework 通过提供大

量预构建视图来提供常用模式，从而充分利用了这一点。

REST framework 提供的通用视图允许您快速构建紧密映射到数据库模型的 API 视图。

如果通用视图不符合需求，可以使用常规的 `APIView` 类，或者利用 `mixin` 特性和基类组合出可重用的视图。

举个栗子

通常，在使用通用视图时，您需要继承该视图，并设置几个类属性。

```
1. from django.contrib.auth.models import User
2. from myapp.serializers import UserSerializer
3. from rest_framework import generics
4. from rest_framework.permissions import IsAdminUser
5.
6. class UserList(generics.ListCreateAPIView):
7.     queryset = User.objects.all()
8.     serializer_class = UserSerializer
9.     permission_classes = (IsAdminUser,)
```

对于更复杂的情况，您可能还想重写视图类中的各种方法。例如。

```
1. class UserList(generics.ListCreateAPIView):
2.     queryset = User.objects.all()
3.     serializer_class = UserSerializer
4.     permission_classes = (IsAdminUser,)
5.
6.     def list(self, request):
7.         # 注意使用`get_queryset()`而不是`self.queryset`
8.         queryset = self.get_queryset()
9.         serializer = UserSerializer(queryset, many=True)
10.        return Response(serializer.data)
```

对于非常简单的情况，您可能想要使用 `.as_view()` 方法来传递类属性。例如，您的 `URLconf` 可能包含类似于以下条目的内容：

```
1. url(r'^/users/', ListCreateAPIView.as_view(queryset=User.objects.all(),
    serializer_class=UserSerializer), name='user-list')
```

直接在 `URLconf` 中设置相关属性参数，这样连视图类都不用写了。

API 参考

GenericAPIView

`GenericAPIView` 类继承于 REST framework 的 `APIView` 类，为标准列表和详细视图添加了常见的行为。

内置的每一个具体的通用视图都是通过将 `GenericAPIView` 类和一个或多个 `mixxin` 类相互结合来构建的。

属性

基本设置

以下属性控制基本视图行为。

- `queryset` - 用于从此视图返回对象的查询集。通常，您必须设置此属性，或覆盖 `get_queryset()` 方法。如果你重写了一个视图方法，在视图方法中，你应该调用 `get_queryset()` 而不是直接访问这个属性，这一点很重要！因为 REST 会在内部对 `queryset` 的结果进行缓存用于后续所有请求。
- `serializer_class` - 用于验证和反序列化输入以及序列化输出的序列化类。通常，您必须设置此属性，或覆盖 `get_serializer_class()` 方法。
- `lookup_field` - 用于执行各个模型实例的对象查找的模型字段。默认为 `'pk'`。请注意，使用 hyperlinked API 时，如果需要使用自定义值，则需要确保 API 视图和序列化类设置了 `lookup field`。
- `lookup_url_kwarg` - 用于对象查找的 URL 关键字参数。URL conf 应该包含与此值相对应的关键字参数。如果未设置，则默认使用与 `lookup_field` 相同的值。

关于后两个参数，我这里附上对象查询的源码大家就应该了解了。

省略部分代码，方便理解

```
1. def get_object(self):
2.     # 先获取数据集
3.     queryset = self.filter_queryset(self.get_queryset())
4.     # 拿到查询参数的 key
5.     lookup_url_kwarg = self.lookup_url_kwarg or self.lookup_field
6.     # 组装成 {key:value} 的形式
7.     filter_kwargs = {self.lookup_field: self.kwargs[lookup_url_kwarg]}
8.     # 查询
9.     obj = get_object_or_404(queryset, **filter_kwargs)
10.    # 最后返回
11.    return obj
```

分页

与列表视图一起使用时，以下属性用于控制分页。

- `pagination_class` - 对列表进行分页时使用的分页类。默认值与 `DEFAULT_PAGINATION_CLASS` 设置相同，即 `rest_framework.pagination.PageNumberPagination`。设置 `pagination_class = None` 将禁用此视图的分页。

过滤

- `filter_backends` - 用于过滤查询集的过滤器类的列表。默认值与 `DEFAULT_FILTER_BACKENDS` 设置的值相同。

方法

基本方法

`get_queryset(self)`

应该返回列表视图的查询集，并应该将其用作查看详细视图的基础。默认返回由 `queryset` 属性指定的查询集。

应该始终使用此方法，而不是直接访问 `self.queryset`，因为 REST 会在内部对 `self.queryset` 的结果进行缓存用于后续所有请求。

可以覆盖以提供动态行为，例如针对不同用户的请求返回不同的数据。

举个栗子：

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

`get_object(self)`

应该返回详细视图的对象实例。默认使用 `lookup_field` 参数来过滤基本查询集。

可以覆盖以提供更复杂的行为，例如基于多个 URL kwarg 的对象查找。

举个栗子：

```
1. def get_object(self):
2.     queryset = self.get_queryset()
3.     filter = {}
```

```

4.     for field in self.multiple_lookup_fields:
5.         filter[field] = self.kwargs[field]
6.
7.     obj = get_object_or_404(queryset, **filter)
8.     self.check_object_permissions(self.request, obj)
9.     return obj

```

请注意，如果您的 API 不包含任何对象级权限，您可以选择排除

`self.check_object_permissions`，并简单地从 `get_object_or_404` 中查找返回对象。

`filter_queryset(self, queryset)`

给定一个查询集，使用过滤器进行过滤，返回一个新的查询集。

举个栗子：

```

1. def filter_queryset(self, queryset):
2.     filter_backends = (CategoryFilter,)
3.
4.     if 'geo_route' in self.request.query_params:
5.         filter_backends = (GeoRouteFilter, CategoryFilter)
6.     elif 'geo_point' in self.request.query_params:
7.         filter_backends = (GeoPointFilter, CategoryFilter)
8.
9.     for backend in list(filter_backends):
10.        queryset = backend().filter_queryset(self.request, queryset, view=self)
11.
12.    return queryset

```

`get_serializer_class(self)`

返回用于序列化的类。默认返回 `serializer_class` 属性。

可以被覆盖以提供动态行为，例如使用不同的序列化器进行读写操作，或为不同类型的用户提供不同的序列化器。

举个栗子：

```

1. def get_serializer_class(self):
2.     if self.request.user.is_staff:
3.         return FullAccountSerializer
4.     return BasicAccountSerializer

```

保存和删除钩子（hook）

以下方法由 `mixin` 类提供，可以很轻松的重写对象的保存和删除行为。

- `perform_create(self, serializer)` - 保存新对象实例时由 `CreateModelMixin` 调用。
- `perform_update(self, serializer)` - 在保存现有对象实例时由 `UpdateModelMixin` 调用。
- `perform_destroy(self, instance)` - 删除对象实例时由 `DestroyModelMixin` 调用。

这些钩子 (hook) 对设置请求中隐含的但不属于请求数据的属性特别有用。例如，您可以根据请求用户或基于 URL 关键字参数在对象上设置属性。

```
1. def perform_create(self, serializer):
2.     serializer.save(user=self.request.user)
```

这些覆盖点对于添加保存对象之前或之后发生的行为（如发送确认电子邮件或记录更新）也特别有用。

```
1. def perform_create(self, serializer):
2.     queryset = SignupRequest.objects.filter(user=self.request.user)
3.     if queryset.exists():
4.         raise ValidationError('You have already signed up')
5.     serializer.save(user=self.request.user)
```

注意：这些方法替代旧式版本 2.x `pre_save`，`post_save`，`pre_delete` 和 `post_delete` 方法，这些方法不再可用。

其他方法

通常不需要重写以下方法，但如果使用 `GenericAPIView` 编写自定义视图，则可能需要调用它们。

- `get_serializer_context(self)` - 返回包含应该提供给序列化的任何额外上下文的字典。默认包括 `'request'`，`'view'` 和 `'format'` 键。
- `get_serializer(self, instance=None, data=None, many=False, partial=False)` - 返回一个序列化器实例。
- `get_paginated_response(self, data)` - 返回分页样式的 `Response` 对象。
- `paginate_queryset(self, queryset)` - 根据需要为查询集分页，或者返回一个页面对象；如果没有为该视图配置分页，则为 `None`。
- `filter_queryset(self, queryset)` - 给定一个查询集，使用过滤器进行过滤，返回一个新的查询集。

Mixins

`mixin` 类用于提供基本视图行为的操作。请注意，`mixin` 类提供了操作方法，而不是直接定义处理方法，如 `.get()` 和 `.post()`。这允许更灵活的行为组合。

mixin 类可以从 `rest_framework.mixins` 中导入。

ListModelMixin

提供一个 `.list(request, *args, **kwargs)` 方法，实现了列出一个查询集。

如果查询集已填充，则返回 `200 OK` 响应，并将 `queryset` 的序列化表示形式作为响应的主体。响应数据可以设置分页。

CreateModelMixin

提供 `.create(request, *args, **kwargs)` 方法，实现创建和保存新模型实例。

如果创建了一个对象，则会返回一个 `201 Created` 响应，并将该对象的序列化表示形式作为响应的主体。如果表示包含名为 `url` 的键，则响应的 `Location` header 将填充该值。

如果为创建对象提供的请求数据无效，则将返回 `400 Bad Request` 响应，并将错误细节作为响应的主体。

RetrieveModelMixin

提供 `.retrieve(request, *args, **kwargs)` 方法，该方法实现在响应中返回现有的模型实例。

如果可以获取对象，则返回 `200 OK` 响应，并将对象的序列化表示作为响应的主体。否则，将返回一个 `404 Not Found`。

UpdateModelMixin

提供 `.update(request, *args, **kwargs)` 方法，实现更新和保存现有模型实例。还提供了一个 `.partial_update(request, *args, **kwargs)` 方法，它与更新方法类似，只是更新的所有字段都是可选的。这允许支持 HTTP `PATCH` 请求。

如果成功更新对象，则返回 `200 OK` 响应，并将对象的序列化表示形式作为响应的主体。

如果提供的用于更新对象的请求数据无效，则将返回 `400 Bad Request` 响应，并将错误细节作为响应的主体。

DestroyModelMixin

提供一个 `.destroy(request, *args, **kwargs)` 方法，实现现有模型实例的删除。

如果一个对象被删除，则返回一个 `204 No Content`，否则它将返回一个 `404 Not Found`。

内置视图类列表

以下类是具体的通用视图。通常情况下，你应该都是使用的它们，除非需要高度的自定义行为。

这些视图类可以从 `rest_framework.generics` 中导入。

CreateAPIView

仅用于创建实例。

提供一个 `post` 请求的处理方法。

继承自: `GenericAPIView` , `CreateModelMixin`

ListAPIView

仅用于读取模型实例列表。

提供一个 `get` 请求的处理方法。

继承自: `GenericAPIView` , `ListModelMixin`

RetrieveAPIView

仅用于查询单个模型实例。

提供一个 `get` 请求的处理方法。

继承自: `GenericAPIView` , `RetrieveModelMixin`

DestroyAPIView

仅用于删除单个模型实例。

提供一个 `delete` 请求的处理方法。

继承自: `GenericAPIView` , `DestroyModelMixin`

UpdateAPIView

仅用于更新单个模型实例。

提供 `put` 和 `patch` 请求的处理方法。

继承自: `GenericAPIView` , `UpdateModelMixin`

ListCreateAPIView

既可以获取也可以实例集合，也可以创建实例列表

提供 `get` 和 `post` 请求的处理方法。

继承自: `GenericAPIView` , `ListModelMixin` , `CreateModelMixin`

RetrieveUpdateAPIView

既可以查询也可以更新单个实例

提供 `get` , `put` 和 `patch` 请求的处理方法。

继承自: `GenericAPIView` , `RetrieveModelMixin` , `UpdateModelMixin`

RetrieveDestroyAPIView

既可以查询也可以删除单个实例

提供 `get` 和 `delete` 请求的处理方法。

继承自: `GenericAPIView` , `RetrieveModelMixin` , `DestroyModelMixin`

RetrieveUpdateDestroyAPIView

同时支持查询，更新，删除

提供 `get` , `put` , `patch` 和 `delete` 请求的处理方法。

继承自: `GenericAPIView` ,
`RetrieveModelMixin` , `UpdateModelMixin` , `DestroyModelMixin`

自定义通用视图类

通常你会想使用现有的通用视图，然后稍微定制一下行为。如果您发现自己在多个地方重复使用了一些自定义行为，则可能需要将行为重构为普通类，然后根据需要将其应用于任何视图或视图集。

自定义 mixins

例如，如果您需要根据 URL conf 中的多个字段查找对象，则可以创建一个 mixin 类。

举个栗子：

```

1. class MultipleFieldLookupMixin(object):
2.     """
3.     将此 mixin 应用于任何视图或视图集以获取多个字段过滤
4.     基于`lookup_fields`属性，而不是默认的单字段过滤。
5.     """
6.     def get_object(self):
7.         queryset = self.get_queryset() # 获取基本的查询集
8.         queryset = self.filter_queryset(queryset) # 使用过滤器
9.         filter = {}
10.        for field in self.lookup_fields:
11.            if self.kwargs[field]: # 忽略空字段
12.                filter[field] = self.kwargs[field]
13.        obj = get_object_or_404(queryset, **filter) # 查找对象
14.        self.check_object_permissions(self.request, obj)
15.        return obj

```

随后可以在需要应用自定义行为的任何时候，将该 mixin 应用于视图或视图集。

```

1. class RetrieveUserView(MultipleFieldLookupMixin, generics.RetrieveAPIView):
2.     queryset = User.objects.all()
3.     serializer_class = UserSerializer
4.     lookup_fields = ('account', 'username')

```

如果您需要使用自定义行为，那么使用自定义 mixins 是一个不错的选择。

自定义基类

如果您在多个视图中使用 mixin，您可以进一步创建自己的一组基本视图，然后在整个项目中使用它们。例如：

```

1. class BaseRetrieveView(MultipleFieldLookupMixin,
2.                        generics.RetrieveAPIView):
3.     pass
4.
5. class BaseRetrieveUpdateDestroyView(MultipleFieldLookupMixin,
6.                                     generics.RetrieveUpdateDestroyAPIView):
7.     pass

```

如果自定义行为始终需要在整个项目中的大量视图中重复使用，那么使用自定义基类是一个不错的选择。

视图集

- [视图集](#)
 - [举个栗子](#)
 - [操作视图集](#)
 - [标记额外的路由行为](#)
 - [action 跳转](#)
- [API 参考](#)
 - [ViewSet](#)
 - [GenericViewSet](#)
 - [ModelViewSet](#)
 - [ReadOnlyModelViewSet](#)
- [自定义视图集基类](#)

[官方原文链接](#)

视图集

在路由决定了哪个控制器用于请求后，控制器负责理解请求并产生适当的输出。

– *Ruby on Rails* 文档

Django REST framework 允许将一组相关视图的逻辑组合到一个称为 `ViewSet` 的类中。在其他框架中，您可能会发现概念上类似的实现，名为 “Resources” 或 “Controllers”。

`ViewSet` 类只是一种基于类的 View，它不提供任何处理方法，如 `.get()` 或 `.post()`，而是提供诸如 `.list()` 和 `.create()` 之类的操作。

`ViewSet` 只在用 `.as_view()` 方法绑定到最终化视图时做一些相应操作。

通常，不是在 `urlpatterns` 中的视图集中明确注册视图，而是使用路由器类注册视图集，这会自动为您确定 `urlpatterns`。

举个栗子

定义一个简单的视图集，可以用来列出或检索系统中的所有用户。

```
1. from django.contrib.auth.models import User
2. from django.shortcuts import get_object_or_404
3. from myapps.serializers import UserSerializer
4. from rest_framework import viewsets
5. from rest_framework.response import Response
6.
```

```

7. class UserViewSet(viewsets.ViewSet):
8.
9.     def list(self, request):
10.         queryset = User.objects.all()
11.         serializer = UserSerializer(queryset, many=True)
12.         return Response(serializer.data)
13.
14.     def retrieve(self, request, pk=None):
15.         queryset = User.objects.all()
16.         user = get_object_or_404(queryset, pk=pk)
17.         serializer = UserSerializer(user)
18.         return Response(serializer.data)

```

如果需要，可以将这个视图集成两个单独的视图，如下所示：

```

1. user_list = UserViewSet.as_view({'get': 'list'})
2. user_detail = UserViewSet.as_view({'get': 'retrieve'})

```

通常情况下，我们不会这样做，而是用路由器注册视图集，并允许自动生成 `urlpatterns`。

```

1. from myapp.views import UserViewSet
2. from rest_framework.routers import DefaultRouter
3.
4. router = DefaultRouter()
5. router.register(r'users', UserViewSet, base_name='user')
6. urlpatterns = router.urls

```

不用自己编写视图集，通常使用默认提供的现有基类。例如：

```

1. class UserViewSet(viewsets.ModelViewSet):
2.     """
3.     用于查看和编辑用户实例的视图。
4.     """
5.     serializer_class = UserSerializer
6.     queryset = User.objects.all()

```

使用 `ViewSet` 类比使用 `View` 类有两个主要优点。

- 重复的逻辑可以合并成一个类。在上面的例子中，我们只需要指定一次查询集，它将在多个视图中使用。
- 通过使用 `routers`，我们不再需要处理自己的 `URL` 配置。

这两者各有优缺点。使用常规视图和 URL 配置文件更加明确，并为您提供更多控制。如果想要更快速的开发出一个应用，或者需要使大型 API 的 URL 配置始终保持一致，视图集会非常有用。

操作视图集

REST framework 中包含的默认 routes 将为一组标准的 create / retrieve / update / destroy 风格 action 提供路由，如下所示：

```
1. class UserViewSet(viewsets.ViewSet):
2.     """
3.     这些方法将由路由器负责处理。
4.
5.     如果要使用后缀，请确保加上 `format = None` 关键字参数
6.     """
7.
8.     def list(self, request):
9.         pass
10.
11.    def create(self, request):
12.        pass
13.
14.    def retrieve(self, request, pk=None):
15.        pass
16.
17.    def update(self, request, pk=None):
18.        pass
19.
20.    def partial_update(self, request, pk=None):
21.        pass
22.
23.    def destroy(self, request, pk=None):
24.        pass
```

在调度期间，当前 action 的名称可以通过 `.action` 属性获得。您可以检查 `.action` 以根据当前 action 调整行为。

例如，您可以将权限限制为只有 admin 才能访问 `list` 以外的其他 action，如下所示：

```
1. def get_permissions(self):
2.     """
3.     实例化并返回此视图所需的权限列表。
4.     """
```



```

5.     if self.action == 'list':
6.         permission_classes = [IsAuthenticated]
7.     else:
8.         permission_classes = [IsAdmin]
9.     return [permission() for permission in permission_classes]

```

标记额外的路由行为

如果需要路由特定方法，则可以用 `@detail_route` 或 `@list_route` 装饰器进行修饰。

`@detail_route` 装饰器在其 URL 模式中包含 `pk`，用于支持需要获取单个实例的方法。`@list_route` 修饰器适用于在对象列表上操作的方法。

举个例子：

```

1.  from django.contrib.auth.models import User
2.  from rest_framework import status
3.  from rest_framework import viewsets
4.  from rest_framework.decorators import detail_route, list_route
5.  from rest_framework.response import Response
6.  from myapp.serializers import UserSerializer, PasswordSerializer
7.
8.  class UserViewSet(viewsets.ModelViewSet):
9.      """
10.     提供标准操作的视图集
11.     """
12.     queryset = User.objects.all()
13.     serializer_class = UserSerializer
14.
15.     @detail_route(methods=['post'])
16.     def set_password(self, request, pk=None):
17.         user = self.get_object()
18.         serializer = PasswordSerializer(data=request.data)
19.         if serializer.is_valid():
20.             user.set_password(serializer.data['password'])
21.             user.save()
22.             return Response({'status': 'password set'})
23.         else:
24.             return Response(serializer.errors,
25.                             status=status.HTTP_400_BAD_REQUEST)
26.
27.     @list_route()

```

```

28.     def recent_users(self, request):
29.         recent_users = User.objects.all().order('-last_login')
30.
31.         page = self.paginate_queryset(recent_users)
32.         if page is not None:
33.             serializer = self.get_serializer(page, many=True)
34.             return self.get_paginated_response(serializer.data)
35.
36.         serializer = self.get_serializer(recent_users, many=True)
37.         return Response(serializer.data)

```

另外，装饰器可以为路由视图设置额外的参数。例如...

```

1.     @detail_route(methods=['post'], permission_classes=[IsAdminOrIsSelf])
2.     def set_password(self, request, pk=None):
3.         ...

```

这些装饰器默认路由 `GET` 请求，但也可以使用 `methods` 参数接受其他 HTTP 方法。例如：

```

1.     @detail_route(methods=['post', 'delete'])
2.     def unset_password(self, request, pk=None):
3.         ...

```

这两个新操作将在 `urls` `^users/{pk}/set_password/$` 和 `^users/{pk}/unset_password/$` 上。

action 跳转

如果你需要获取 action 的 URL，请使用 `.reverse_action()` 方法。这是 `.reverse()` 的一个便捷包装，它会自动传递视图的请求对象，并将 `url_name` 与 `.basename` 属性挂接。

请注意，`basename` 是在 `ViewSet` 注册过程中由路由器提供的。如果您不使用路由器，则必须提供 `.as_view()` 方法的 `basename` 参数。

使用上一节中的示例：

```

1. >>> view.reverse_action('set-password', args=['1'])
2. 'http://localhost:8000/api/users/1/set_password'

```

`url_name` 参数应该与 `@list_route` 和 `@detail_route` 装饰器的相同参数匹配。另外，这可以用来反转默认 `list` 和 `detail` 路由。

API 参考

ViewSet

`ViewSet` 类继承自 `APIView`。您可以使用任何标准属性（如 `permission_classes`，`authentication_classes`）来控制视图上的 API 策略。

`ViewSet` 类不提供任何 action 的实现。为了使用 `ViewSet` 类，必须继承该类并明确定义 action 实现。

GenericViewSet

`GenericViewSet` 类继承自 `GenericAPIView`，并提供默认的 `get_object`，`get_queryset` 方法和其他通用视图基础行为，但默认情况下不包含任何操作。

为了使用 `GenericViewSet` 类，必须继承该类并混合所需的 `mixins` 类，或明确定义操作实现。

ModelViewSet

`ModelViewSet` 类继承自 `GenericAPIView`，并通过混合各种 `mixins` 类的行为来包含各种操作的实现。

`ModelViewSet` 提供的操作有 `.list()`，`.retrieve()`，`.create()`，`.update()`，`.partial_update()`，和 `.destroy()`。

举个栗子：

由于 `ModelViewSet` 类继承自 `GenericAPIView`，因此通常需要提供至少 `queryset` 和 `serializer_class` 属性。例如：

```
1. class AccountViewSet(viewsets.ModelViewSet):
2.     """
3.     用于查看和编辑 Account
4.     """
5.     queryset = Account.objects.all()
6.     serializer_class = AccountSerializer
7.     permission_classes = [IsAccountAdminOrReadOnly]
```

请注意，您可以覆盖 `GenericAPIView` 提供的任何标准属性或方法。例如，要动态确定它应该操作的查询集的 `ViewSet`，可以这样做：

```
1. class AccountViewSet(viewsets.ModelViewSet):
```

```

2.     """
3.     A simple ViewSet for viewing and editing the accounts
4.     associated with the user.
5.     """
6.     serializer_class = AccountSerializer
7.     permission_classes = [IsAccountAdminOrReadOnly]
8.
9.     def get_queryset(self):
10.         return self.request.user.accounts.all()

```

但请注意，从 `ViewSet` 中删除 `queryset` 属性后，任何关联的 `router` 将无法自动导出模型的 `base_name`，因此您必须将 `base_name` kwarg 指定为 `router` 注册的一部分。

还要注意，虽然这个类默认提供了完整的 `create` / `list` / `retrieve` / `update` / `destroy` 操作集，但您可以通过使用标准权限类来限制可用操作。

ReadOnlyModelViewSet

`ReadOnlyModelViewSet` 类也从 `GenericAPIView` 继承。与 `ModelViewSet` 一样，它也包含各种操作的实现，但与 `ModelViewSet` 不同的是它只提供“只读”操作，`.list()` 和 `.retrieve()`。

举个栗子：

与 `ModelViewSet` 一样，您通常需要提供至少 `queryset` 和 `serializer_class` 属性。例如：

```

1. class AccountViewSet(viewsets.ReadOnlyModelViewSet):
2.     """
3.     A simple ViewSet for viewing accounts.
4.     """
5.     queryset = Account.objects.all()
6.     serializer_class = AccountSerializer

```

同样，与 `ModelViewSet` 一样，您可以覆盖 `GenericAPIView` 可用的任何标准属性和方法。

自定义视图集基类

您可能需要使用没有完整 `ModelViewSet` 操作集的自定义 `ViewSet` 类，或其他自定义行为。

举个栗子：

要创建提供 `create` , `list` 和 `retrieve` 操作的基本视图集类, 请从 `GenericViewSet` 继承, 并混合 (mixin) 所需的操作:

```
1. from rest_framework import mixins
2.
3. class CreateListRetrieveViewSet(mixins.CreateModelMixin,
4.                                mixins.ListModelMixin,
5.                                mixins.RetrieveModelMixin,
6.                                viewsets.GenericViewSet):
7.     """
8.     A viewset that provides `retrieve`, `create`, and `list` actions.
9.
10.    To use it, override the class and set the `.queryset` and
11.    `.serializer_class` attributes.
12.    """
13.    pass
```

通过创建自己的基本 `ViewSet` 类, 能够提供可在 API 中的多个视图集中重用的常见行为。

路由

- [路由](#)
 - [用法](#)
 - [使用 `include` 与路由](#)
 - [额外的链接和操作](#)
- [API 参考](#)
 - [SimpleRouter](#)
 - [DefaultRouter](#)
- [自定义路由](#)
 - [自定义动态路由](#)
 - [举个栗子](#)
 - [自定义路由器进阶](#)
- [第三方软件包](#)
 - [DRF Nested Routers](#)
 - [ModelRouter \(wq.db.rest\)](#)
 - [DRF-extensions](#)

[官方原文链接](#)

路由

一些 Web 框架（如 Rails）提供了一种能够自动确定应用程序的 URL 如何映射到处理请求的功能。

REST framework 增加了对 Django 自动 URL 路由的支持，并提供了一种将视图逻辑连接到一组 URL 的简单，高效和一致的方式。

用法

下面是一个使用 `SimpleRouter` 的简单 URL 配置示例。

```
1. from rest_framework import routers
2.
3. router = routers.SimpleRouter()
4. router.register(r'users', UserViewSet)
5. router.register(r'accounts', AccountViewSet)
6. urlpatterns = router.urls
```

`register()` 方法有两个必须参数：

- `prefix` - 设置这组路由的前缀。
- `viewset` - 设置对应的视图集类。

或者，您也可以指定一个附加参数：

- `base_name` - 用于创建的 URL 名称的基础。如果未设置，将根据视图集的 `queryset` 属性自动生成。请注意，如果视图集不包含 `queryset` 属性，则在注册视图集时必须设置 `base_name`。

上面的例子会生成以下 URL 模式：

- URL pattern: `^users/$` Name: `'user-list'`
- URL pattern: `^users/{pk}/$` Name: `'user-detail'`
- URL pattern: `^accounts/$` Name: `'account-list'`
- URL pattern: `^accounts/{pk}/$` Name: `'account-detail'`

注意：`base_name` 参数用于指定视图名称模式的初始部分。在上面的例子中，是 `user` 或 `account` 部分。

通常，您不需要指定 `base_name` 参数，但是如果您有一个视图集定义了自定义 `get_queryset` 方法，那么该视图集可能没有设置 `.queryset` 属性。如果此时尝试注册该视图，则会看到如下所示的错误：

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

`'base_name'` 参数未指定，并且无法自动确定视图中的名称，因为它没有 `'.queryset'` 属性。

这时候就需要在注册视图集时显式设置 `base_name` 参数，因为它无法从模型名称中自动确定。

使用 `include` 与路由

路由实例上的 `.urls` 属性是一个标准的 URL patterns。关于如何包含这些 URL，有许多不同的样式。

例如，可以将 `router.urls` 附加到现有视图的列表中...

```
1. router = routers.SimpleRouter()
2. router.register(r'users', UserViewSet)
3. router.register(r'accounts', AccountViewSet)
4.
5. urlpatterns = [
6.     url(r'^forgot-password/$', ForgotPasswordFormView.as_view()),
7. ]
```

```
8.
9. urlpatterns += router.urls
```

另外,你也可以使用 Django 的 `include` 函数,比如...

```
1. urlpatterns = [
2.     url(r'^forgot-password/$', ForgotPasswordFormView.as_view()),
3.     url(r'^$', include(router.urls)),
4. ]
```

还可以设置 namespace。

```
1. urlpatterns = [
2.     url(r'^forgot-password/$', ForgotPasswordFormView.as_view()),
3.     url(r'^api/', include(router.urls, namespace='api')),
4. ]
```

如果对超链接序列化器使用命名空间,则还需要确保序列化器上的任何 `view_name` 参数都能正确反映命名空间。在上面的示例中,您需要为超链接到用户详细信息视图的序列化程序字段包含诸如 `view_name='api:user-detail'` 之类的参数。

额外的链接和操作

用 `@detail_route` 或 `@list_route` 装饰的 视图上的任何方法 也将被路由。例如,在 `UserViewSet` 类中给出这样的方法:

```
1. from myapp.permissions import IsAdminOrIsSelf
2. from rest_framework.decorators import detail_route
3.
4. class UserViewSet(ModelViewSet):
5.     ...
6.
7.     @detail_route(methods=['post'], permission_classes=[IsAdminOrIsSelf])
8.     def set_password(self, request, pk=None):
9.         ...
```

会生成以下URL模式:

- URL pattern: `^users/{pk}/set_password/$` Name: `'user-set-password'`

如果您不想使用默认生成的 URL 模式,则可以使用 `url_path` 参数对其进行自定义。

例如，如果您想将我们的自定义操作的URL更改为 `^users/{pk}/change-password/$`，则可以编写：

```
1. from myapp.permissions import IsAdminOrIsSelf
2. from rest_framework.decorators import detail_route
3.
4. class UserViewSet(ModelViewSet):
5.     ...
6.
7.     @detail_route(methods=['post'], permission_classes=[IsAdminOrIsSelf],
8.                   url_path='change-password')
9.     def set_password(self, request, pk=None):
10.         ...
```

上面的例子现在将生成以下URL模式：

- URL pattern: `^users/{pk}/change-password/$` Name: `'user-change-password'`

如果您不想使用生成的默认名称，则可以使用 `url_name` 参数对其进行自定义。

例如，如果您想将自定义操作的名称更改为 `'user-change-password'`，则可以编写：

```
1. from myapp.permissions import IsAdminOrIsSelf
2. from rest_framework.decorators import detail_route
3.
4. class UserViewSet(ModelViewSet):
5.     ...
6.
7.     @detail_route(methods=['post'], permission_classes=[IsAdminOrIsSelf],
8.                   url_name='change-password')
9.     def set_password(self, request, pk=None):
10.         ...
```

上面的例子现在将生成以下URL模式：

- URL pattern: `^users/{pk}/set_password/$` Name: `'user-change-password'`

可以同时使用 `url_path` 和 `url_name` 参数。

更多相关信息请看 [视图集：标记额外的路由行为](#)。

API 参考

SimpleRouter

`SimpleRouter` 包含标准的 `list` , `create` , `retrieve` , `update` , `partial_update` 和 `destroy` action。`SimpleRouter` 还支持视图集使用 `@detail_route` 或 `@list_route` 装饰器标记其他要路由的方法。

URL Style	HTTP Method	Action	URL Name
{prefix}/	GET	list	{basename}-list
	POST	create	
{prefix}/{methodname}/	GET, 或者由 <code>methods</code> 参数指定	<code>@list_route</code> 装饰的方法	{basename}-{methodname}
{prefix}/{lookup}/	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{methodname}/	GET, 或者由 <code>methods</code> 参数指定	<code>@detail_route</code> 装饰的方法	{basename}-{methodname}

默认情况下, 由 `SimpleRouter` 创建的 URL 附加了尾部斜杠。在实例化路由器时, 可以通过将 `trailing_slash` 参数设置为 `False` 来修改此行为。例如:

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

尾部斜杠在 Django 中是常规的, 但在其他一些框架 (如 Rails) 中默认不使用。选择使用哪种风格在很大程度上是一个偏好问题, 尽管一些 JavaScript 框架可能会期望特定的路由风格。

`SimpleRouter` 将匹配包含除斜杠和句点字符以外的任何字符的 `lookup` 值。对于更严格 (或宽松) 的 `lookup` pattern, 请在视图集上设置 `lookup_value_regex` 属性。例如, 您可以将 `lookup` 限制为有效的 UUID:

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

DefaultRouter

`DefaultRouter` 与上面的 `SimpleRouter` 相似，但还包含一个默认的 API 根视图，该视图返回一个包含指向所有列表视图的超链接的响应。它还可选的 `.json` 风格格式后缀生成路由。

URL Style	HTTP Method	Action	URL Name
[.format]	GET	自动生成的根视图	api-root
{prefix}/[.format]	GET	list	{basename}-list
	POST	create	
{prefix}/{methodname}/[.format]	GET, 或者由 <code>methods</code> 参数指定	<code>@list_route</code> 装饰的方法	{basename}-{methodname}
{prefix}/{lookup}/[.format]	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{methodname}/[.format]	GET, 或者由 <code>methods</code> 参数指定	<code>@detail_route</code> 装饰的方法	{basename}-{methodname}

注意：我在使用 3.7.7 版本时，发现要写成 `{prefix}[.format]/` 风格才能访问，`{prefix}/[.format]` 风格会报 404，不知道是我设置问题还是官方更新了。

与 `SimpleRouter` 一样，通过在实例化 `DefaultRouter` 时将 `trailing_slash` 参数设置为 `False`，可以删除 URL 路径上的尾部斜杠。

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

自定义路由

自定义路由并不是你经常需要做的事情，但是如果你对 API 的 URL 是如何构建的有特定的要求的话，它会很有用。这样做可以让你以可重用的方式封装 URL 结构，确保你不必为每个新视图明确编写 URL 模式。

实现自定义路由的最简单方法是对现有路由类之一进行子类化。`.routes` 属性用于对将映射到每个视图集的 URL 模式进行模板化。`.Routes` 属性是一个 `Route` 列表（`Route` 的是一个 `namedtuple`）。

`Route` 命名元组的参数：

url：表示要路由的 URL 的字符串。可以包含以下格式字符串：

- `{prefix}` - 路由的 URL 前缀。
- `{lookup}` - 匹配单个实例的 lookup field。
- `{trailing_slash}` - 可以是 `'/'` 或空字符串，具体取决于 `trailing_slash` 参数。

mapping：HTTP 方法名称与视图方法的映射

name：用于调用 `reverse` 时的 URL 的名称。可能包含以下格式字符串：

- `{basename}` - 创建的 URL 名称的基础。

initkwargs：实例化视图时应传递的任何其他参数的字典。注意，`suffix` 参数被保留用于标识视图集类型，在生成视图名称和 breadcrumb 链接时使用。

自定义动态路由

你还可以自定义 `@list_route` 和 `@detail_route` 装饰器的路由方式。要路由这两个装饰器中的一个或两个，请在 `.routes` 列表中包含一个 `DynamicListRoute` 和/或 `DynamicDetailRoute`（别忘了类型是 `namedtuple`）。

`DynamicListRoute` 和 `DynamicDetailRoute` 的参数是：

url：表示要路由的 URL 的字符串。可以包含与 `Route` 相同的格式字符串，并且还接受 `{methodname}` 和 `{methodnamehyphen}` 格式的字符串。

name：用于调用 `reverse` 时的名称。可以包含以下格式字符串：`{basename}`，`{methodname}` 和 `{methodnamehyphen}`。

initkwargs: 实例化视图时应传递的任何其他参数的字典。

举个栗子

以下示例只会路由 `list` 和 `retrieve` action，并且不使用尾部斜杠约定。

```
1. from rest_framework.routers import Route, DynamicDetailRoute, SimpleRouter
2.
3. class CustomReadOnlyRouter(SimpleRouter):
4.     """
5.     A router for read-only APIs, which doesn't use trailing slashes.
6.     """
7.     routes = [
8.         Route(
9.             url=r'^{prefix}$',
10.            mapping={'get': 'list'},
11.            name='{basename}-list',
12.            initkwargs={'suffix': 'List'}
13.        ),
14.        Route(
15.            url=r'^{prefix}/{lookup}$',
16.            mapping={'get': 'retrieve'},
17.            name='{basename}-detail',
18.            initkwargs={'suffix': 'Detail'}
19.        ),
20.        DynamicDetailRoute(
21.            url=r'^{prefix}/{lookup}/{methodnamehyphen}$',
22.            name='{basename}-{methodnamehyphen}',
23.            initkwargs={}
24.        )
25.    ]
```

让我们来看看 `CustomReadOnlyRouter` 为一个简单的视图集生成的路由。

`views.py` :

```
1. class UserViewSet(viewsets.ReadOnlyModelViewSet):
2.     """
3.     A viewset that provides the standard actions
4.     """
5.     queryset = User.objects.all()
6.     serializer_class = UserSerializer
```

```

7.     lookup_field = 'username'
8.
9.     @detail_route()
10.    def group_names(self, request, pk=None):
11.        """
12.        Returns a list of all the group names that the given
13.        user belongs to.
14.        """
15.        user = self.get_object()
16.        groups = user.groups.all()
17.        return Response([group.name for group in groups])

```

`urls.py` :

```

1. router = CustomReadOnlyRouter()
2. router.register('users', UserViewSet)
3. urlpatterns = router.urls

```

将生成以下映射...

URL	HTTP Method	Action	URL Name
/users	GET	list	user-list
/users/{username}	GET	retrieve	user-detail
/users/{username}/group-names	GET	group_names	user-group-names

有关设置 `.routes` 属性的另一个示例，请参阅 `SimpleRouter` 类的源代码。

自定义路由器进阶

如果想提供完全自定义的行为，可以继承 `BaseRouter` 并覆盖 `get_urls(self)` 方法。该方法应检查已注册的视图集并返回一组 URL 模式。可以通过访问 `self.registry` 属性来检查注册的 `prefix`, `viewset` 和 `basename` tuples。

你可能还想覆盖 `get_default_base_name(self, viewset)` 方法，或者在向路由注册视图集时始终显式设置 `base_name` 参数。

第三方软件包

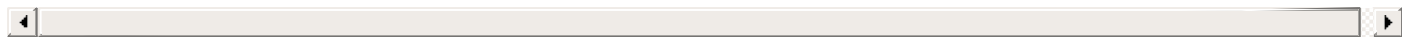
以下是可用的第三方包。

DRF Nested Routers

路由

ModelRouter (wq.db.rest)

DRF-extensions



解析

- [解析器](#)
 - [如何确定使用哪个解析器](#)
 - [设置解析器](#)
- [API 参考](#)
 - [JSONParser](#)
 - [FormParser](#)
 - [MultiPartParser](#)
 - [FileUploadParser](#)
- [自定义解析](#)
 - [stream](#)
 - [media_type](#)
 - [parser_context](#)
 - [举个栗子](#)
- [第三方包](#)
 - [YAML](#)
 - [安装和配置](#)
 - [XML](#)
 - [安装和配置](#)
 - [MessagePack](#)
 - [CamelCase JSON](#)
- [友情提示](#)

[官方原文链接](#)

解析器

REST framework 包含许多内置的解析器类，允许接受各种媒体类型 (media types) 的请求。还支持自定义解析器，这使你可以灵活地设计 API 接受的媒体类型。

如何确定使用哪个解析器

视图的有效解析器集始终定义为类列表。当访问 `request.data` 时，REST framework 将检查传入请求的 `Content-Type`，并确定使用哪个解析器来解析请求内容。

注意：在开发客户端应用程序时，请务必确保在 HTTP 请求中发送数据时设置了 `Content-Type`。如果你不设置 `content type`，大多数客户端将默认使用 `'application / x-www-form-urlencoded'`，这可能不是你想要的。

例如，如果你使用 `jquery` 和 `.ajax()` 方法发送 `json` 数据，则应确保包含 `contentType: 'application/json'` 设置。

设置解析器

可以使用 `DEFAULT_PARSER_CLASSES` 设置默认的全局解析器。例如，以下设置将只允许带有 `JSON` 内容的请求，而不是默认的 `JSON` 或表单数据。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_PARSER_CLASSES': (
3.         'rest_framework.parsers.JSONParser',
4.     )
5. }
```

还可以在基于类 (`APIView`) 的视图上设置单个视图或视图集的解析器。

```
1. from rest_framework.parsers import JSONParser
2. from rest_framework.response import Response
3. from rest_framework.views import APIView
4.
5. class ExampleView(APIView):
6.     """
7.     A view that can accept POST requests with JSON content.
8.     """
9.     parser_classes = (JSONParser,)
10.
11.     def post(self, request, format=None):
12.         return Response({'received data': request.data})
```

或者和 `@api_view` 装饰器一起使用。

```
1. from rest_framework.decorators import api_view
2. from rest_framework.decorators import parser_classes
3. from rest_framework.parsers import JSONParser
4.
5. @api_view(['POST'])
6. @parser_classes((JSONParser,))
7. def example_view(request, format=None):
8.     """
9.     A view that can accept POST requests with JSON content.
10.    """
11.    return Response({'received data': request.data})
```

API 参考

JSONParser

解析 JSON 请求内容。

`.media_type:` `application/json`

FormParser

解析 HTML 表单内容。 `request.data` 是一个 `QueryDict` 字典，包含所有表单参数。

通常需要同时使用 `FormParser` 和 `MultiPartParser`，以完全支持 HTML 表单数据。

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

MultiPartParser

解析文件上传的 multipart HTML 表单内容。 `request.data` 是一个 `QueryDict`（其中包含表单参数和文件）。

通常需要同时使用 `FormParser` 和 `MultiPartParser`，以完全支持 HTML 表单数据。

`.media_type:` `application/form-data`

FileUploadParser

解析文件上传内容。 `request.data` 是一个 `QueryDict`（只包含一个存有文件的 `'file'` key）。

如果与 `FileUploadParser` 一起使用的视图是用 `filename` URL 关键字参数调用的，那么该参数将用作文件名。

如果在没有 `filename` URL 关键字参数的情况下调用，则客户端必须在 `Content-Disposition` HTTP header 中设置文件名。例如 `Content-Disposition: attachment; filename=upload.jpg`。

`.media_type:` `*/*`

请注意：

- `FileUploadParser` 用于本地客户端，可以将文件作为原始数据请求上传。对于基于 web 的上传，或者对于具有分段上传支持的本地客户端，您应该使用 `MultiPartParser` 解析器。
- 由于此解析器的 `media_type` 与任何 content type 都匹配，因此 `FileUploadParser` 通常应该是在 API 视图上设置的唯一解析器。
- `FileUploadParser` 遵循 Django 的标准 `FILE_UPLOAD_HANDLERS` 设置和

`request.upload_handlers` 属性。有关更多详细信息，请参阅 Django 文档。

基本用法示例：

```
1. # views.py
2. class FileUploadView(views.APIView):
3.     parser_classes = (FileUploadParser,)
4.
5.     def put(self, request, filename, format=None):
6.         file_obj = request.data['file']
7.         # ...
8.         # do some stuff with uploaded file
9.         # ...
10.        return Response(status=204)
11.
12. # urls.py
13. urlpatterns = [
14.     # ...
15.     url(r'^upload/(?P<filename>[/\+]*)$', FileUploadView.as_view())
16. ]
```

自定义解析

要实现自定义解析器，应该继承 `BaseParser`，设置 `.media_type` 属性并实现 `.parse(self, stream, media_type, parser_context)` 方法。

该方法应该返回将用于填充 `request.data` 属性的数据。

传递给 `.parse()` 的参数是：

stream

表示请求正文的流式对象。

media_type

可选。如果提供，则这是传入请求内容的 media type。

根据请求的 `Content-Type:` header，可以比渲染器的 `media_type` 属性更具体，并且可能包含 media type 参数。比如 `"text/plain; charset=utf-8"`。

parser_context

可选。如果提供，则该参数将是一个包含解析请求内容可能需要的任何其他上下文的字典。

默认情况下，这将包括以下 key: `view` , `request` , `args` , `kwargs` 。

举个栗子

以下是一个示例纯文本解析器，它将使用表示请求正文的字符串填充 `request.data` 属性。

```
1. class PlainTextParser(BaseParser):
2.     """
3.     Plain text parser.
4.     """
5.     media_type = 'text/plain'
6.
7.     def parse(self, stream, media_type=None, parser_context=None):
8.         """
9.         Simply return a string representing the body of the request.
10.        """
11.        return stream.read()
```

第三方包

以下是可用的第三方包。

YAML

[REST framework YAML](#) 提供 YAML 解析和渲染支持。它以前直接包含在 REST framework 包中，现在作为第三方包。

安装和配置

使用pip安装。

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

修改 REST framework settings。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_PARSER_CLASSES': (
3.         'rest_framework_yaml.parsers.YAMLParser',
4.     ),
5.     'DEFAULT_RENDERER_CLASSES': (
```

```
6.         'rest_framework_yaml.renderers.YAMLRenderer',
7.     ),
8. }
```

XML

[REST Framework XML](#) 提供了一种简单的非正式 XML 格式。它以前直接包含在 REST framework 包中，现在作为第三方包。

安装和配置

使用pip安装。

```
1. $ pip install djangorestframework-xml
```

修改 REST framework settings。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_PARSER_CLASSES': (
3.         'rest_framework_xml.parsers.XMLParser',
4.     ),
5.     'DEFAULT_RENDERER_CLASSES': (
6.         'rest_framework_xml.renderers.XMLRenderer',
7.     ),
8. }
```

MessagePack

CamelCase JSON

友情提示

配合源码阅读效果更佳哦

渲染

- 渲染
 - 如何确定使用哪个渲染器
 - 设置渲染器
 - 渲染器类的优先级
- API 参考
 - JSONRenderer
 - TemplateHTMLRenderer
 - StaticHTMLRenderer
 - BrowsableAPIRenderer
 - AdminRenderer
 - HTMLFormRenderer
 - MultiPartRenderer
- 自定义渲染器
 - data
 - media_type=None
 - renderer_context=None
 - 举个栗子
 - 设置 charset
- 渲染器的高级用法
 - media type 的变化
 - 不明确的 media type
 - 设计你的 media type
 - HTML error 视图

官方原文链接

渲染

REST framework 包含许多内置的渲染器类，允许您使用各种 media type 返回响应。同时也支持自定义渲染器。

如何确定使用哪个渲染器

视图的渲染器集合始终被定义为类列表。当调用视图时，REST framework 将对请求内容进行分析，并确定最合适的渲染器以满足请求。内容分析的基本过程包括检查请求的 `Accept` header，以确定它在响应中期望的 media type。或者，用 URL 上的格式后缀明确表示。例如，URL

`http://example.com/api/users_count.json` 可能始终返回 JSON 数据。

设置渲染器

可以使用 `DEFAULT_RENDERER_CLASSES` 设置全局的默认渲染器集。例如，以下设置将使用JSON作为主要 media type，并且还包含自描述 API。

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

还可以使用基于 `APIView` 的视图类来设置单个视图或视图集的渲染器。

```
1. from django.contrib.auth.models import User
2. from rest_framework.renderers import JSONRenderer
3. from rest_framework.response import Response
4. from rest_framework.views import APIView
5.
6. class UserCountView(APIView):
7.     """
8.     A view that returns the count of active users in JSON.
9.     """
10.    renderer_classes = (JSONRenderer, )
11.
12.    def get(self, request, format=None):
13.        user_count = User.objects.filter(active=True).count()
14.        content = {'user_count': user_count}
15.        return Response(content)
```

或者是在基于 `@api_view` 装饰器的函数视图上设置。

```
1. @api_view(['GET'])
2. @renderer_classes((JSONRenderer,))
3. def user_count_view(request, format=None):
4.     """
5.     A view that returns the count of active users in JSON.
6.     """
7.     user_count = User.objects.filter(active=True).count()
8.     content = {'user_count': user_count}
9.     return Response(content)
```

渲染器类的优先级

在为 API 指定渲染器类时，需要考虑它们处理每种媒体类型时的优先级，这点很重要。如果客户端没有指定接受数据的表现形式，例如发送 `Accept: */*` header，或者根本不包含 `Accept` header，则 REST framework 将选择列表中的第一个渲染器用于响应。

例如，如果你的 API 提供 JSON 响应和可浏览的 HTML API，则可能需要将 `JSONRenderer` 作为默认渲染器，以便将 `JSON` 响应发送给未指定 `Accept` header 的客户端。

如果你的 API 包含可根据请求同时处理常规网页和 API 响应的视图，那么你可以考虑将 `TemplateHTMLRenderer` 设置为默认渲染器，以便与发送 broken accept headers 的老式浏览器很好地配合使用。

API 参考

JSONRenderer

使用 utf-8 编码将请求数据呈现为 `JSON`。

请注意，默认风格包含 unicode 字符，并使用紧凑风格呈现（没有多余的空白）响应：

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

客户端可能还会包含“缩进” media type 参数，在这种情况下，返回的 `JSON` 将会缩进。

比如：`Accept: application/json; indent=4`。

```
1. {
2.     "unicode black star": "★",
3.     "value": 999
4. }
```

使用 `UNICODE_JSON` 和 `COMPACT_JSON` 设置键可以更改默认的 JSON 编码风格。

`.media_type:` `application/json`

`.format:` `'.json'`

`.charset:` `None`

TemplateHTMLRenderer

使用 Django 的标准模板将数据呈现为 HTML。与其他渲染器不同，传递给 `Response` 的数据不

需要序列化。另外，创建 `Response` 时可能需要包含 `template_name` 参数。

`TemplateHTMLRenderer` 将创建一个 `RequestContext`，使用 `response.data` 作为上下文字典，并确定用于呈现上下文的模板名称。

模板名称由（按优先顺序）确定：

1. 传递给 `response` 的显式 `template_name` 参数。
2. 在此类上设置明确的 `.template_name` 属性。
3. 调用 `view.get_template_names()` 的返回结果。

使用 `TemplateHTMLRenderer` 的视图示例：

```
1. class UserDetails(generics.RetrieveAPIView):
2.     """
3.     A view that returns a templated HTML representation of a given user.
4.     """
5.     queryset = User.objects.all()
6.     renderer_classes = (TemplateHTMLRenderer,)
7.
8.     def get(self, request, *args, **kwargs):
9.         self.object = self.get_object()
10.        return Response({'user': self.object},
            template_name='user_detail.html')
```

你可以使用 `TemplateHTMLRenderer` 来使 REST framework 返回常规 HTML 页面，或者从单个端点 (a single endpoint) 返回 HTML 和 API 响应。

如果你正在构建使用 `TemplateHTMLRenderer` 以及其他渲染器类的网站，则应考虑将 `TemplateHTMLRenderer` 列为 `renderer_classes` 列表中的第一个类，以便即使对于发送格式错误的 `ACCEPT:` header 的浏览器，也会优先考虑它。

`.media_type:` `text/html`

`.format:` `'.html'`

`.charset:` `utf-8`

StaticHTMLRenderer

一个简单的渲染器，它只是返回预渲染的 HTML。与其他渲染器不同，传递给响应对象的数据应该是表示要返回的内容的字符串。

使用 `StaticHTMLRenderer` 的视图示例：

```
1. @api_view(('GET',))
2. @renderer_classes((StaticHTMLRenderer,))
3. def simple_html_view(request):
4.     data = '<html><body><h1>Hello, world</h1></body></html>'
5.     return Response(data)
```

你可以使用 `StaticHTMLRenderer` 来使 REST framework 返回常规 HTML 页面，或者从单个端点 (a single endpoint) 返回 HTML 和 API 响应。

`.media_type:` `text/html`

`.format:` `'.html'`

`.charset:` `utf-8`

BrowsableAPIRenderer

将数据呈现为可浏览的 HTML API:

Django REST framework v2.0.0

admin -

User List

OPTIONSGET

API endpoint that represents a list of users.

GET /users/

HTTP 200 OK
Vary: Accept
Content-Type: text/html
Allow: GET, POST, HEAD, OPTIONS

```
{
  "count": 2,
  "next": null,
  "previous": null,
  "results": [
    {
      "email": "admin@example.com",
      "groups": [],
      "url": "http://127.0.0.1:8000/users/1/",
      "username": "admin"
    },
    {
      "email": "tom@example.com",
      "groups": [],
      "url": "http://127.0.0.1:8000/users/2/",
      "username": "tom"
    }
  ]
}
```

Username

Email

Groups

POST

该渲染器将确定哪个其他渲染器被赋予最高优先级，并使用该渲染器在 HTML 页面中显示 API 风格响应。

`.media_type:` `text/html`

`.format:` `'api'`

`.charset:` `utf-8`

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

自定义 `BrowsableAPIRenderer`

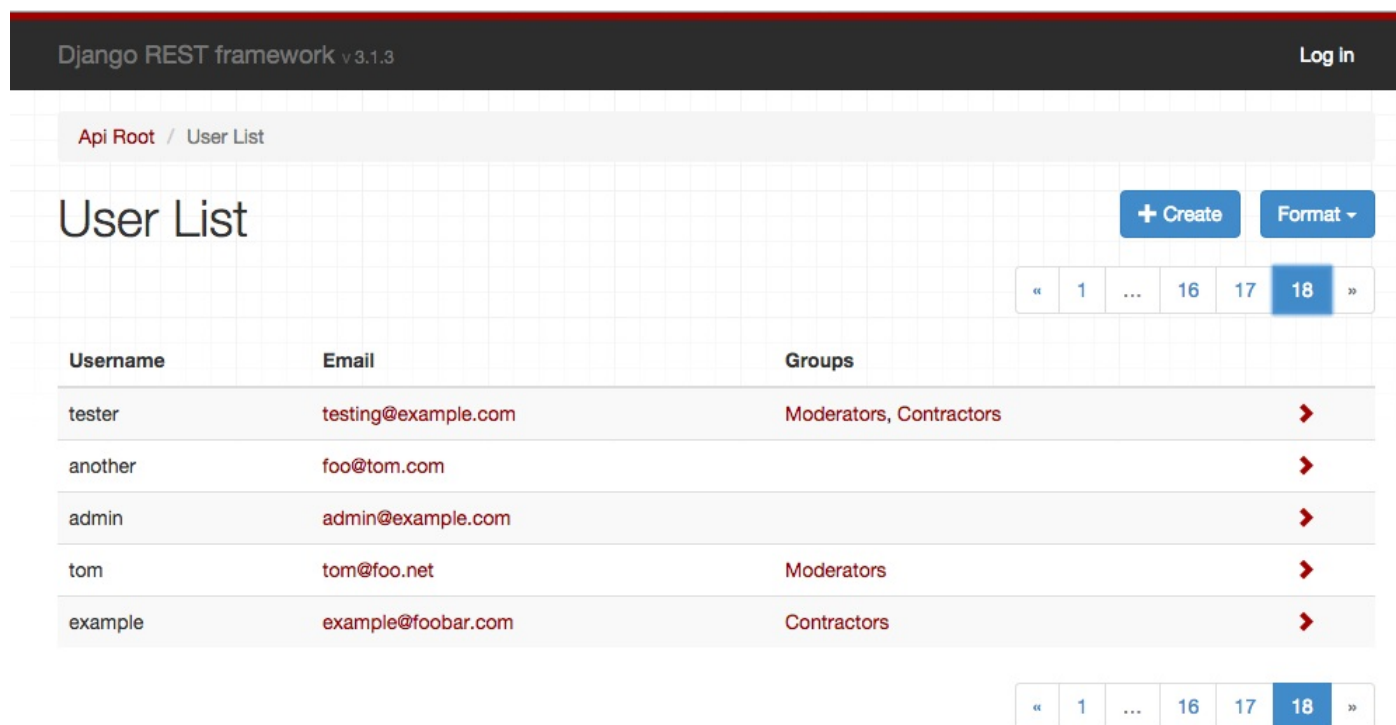
默认情况下，除 `BrowsableAPIRenderer` 之外，响应内容将使用最高优先级的渲染器渲染。如果你需要自定义此行为，例如，将 HTML 用作默认返回格式，但在可浏览的 API 中使用 JSON，则可以通过覆盖 `get_default_renderer()` 方法来实现。

例如：

```
1. class CustomBrowsableAPIRenderer(BrowsableAPIRenderer):
2.     def get_default_renderer(self, view):
3.         return JSONRenderer()
```

AdminRenderer

将数据呈现为 HTML，以显示类似管理员的内容：



该渲染器适用于 CRUD 风格的 Web API，该 API 还应提供用于管理数据的用户友好界面。

请注意，`AdminRenderer` 对于嵌套或列出序列化输入的视图不起作用，因为 HTML 表单无法正确支持它们。

注意：当数据中存在正确配置的 `URL_FIELD_NAME`（默认为 `url`）属性时，`AdminRenderer` 仅能够包含指向详细页面的链接。对于 `HyperlinkedModelSerializer`，情况就是这样，但对于 `ModelSerializer` 类或普通 `Serializer` 类，你需要确保明确包含该字段。例如，在这里我们使用模型 `get_absolute_url` 方法：

```
1. class AccountSerializer(serializers.ModelSerializer):
2.     url = serializers.CharField(source='get_absolute_url', read_only=True)
3.
4.     class Meta:
5.         model = Account
```

`.media_type: text/html``.format: '.admin'``.charset: utf-8``.template: 'rest_framework/admin.html'`

HTMLFormRenderer

将序列化返回的数据呈现为 HTML 表单。该渲染器的输出不包含封闭的 `<form>` 标签，隐藏的 CSRF 输入或任何提交按钮。

这个渲染器不是直接使用，而是可以通过将一个序列化器实例传递给 `render_form` 模板标签来在模板中使用。

```

1. {% load rest_framework %}
2.
3. <form action="/submit-report/" method="post">
4.     {% csrf_token %}
5.     {% render_form serializer %}
6.     <input type="submit" value="Save" />
7. </form>

```

`.media_type: text/html``.format: '.form'``.charset: utf-8``.template: 'rest_framework/horizontal/form.html'`

MultipartRenderer

该渲染器用于呈现 HTML multipart form 数据。它不适合作为响应渲染器，而是用于创建测试请求，使用 REST framework 的测试客户端和测试请求工厂。

`.media_type: multipart/form-data; boundary=BoUnDaRyStRiNg``.format: '.multipart'``.charset: utf-8`

自定义渲染器

要实现自定义渲染器，您应该继承 `BaseRenderer`，设置 `.media_type` 和 `.format` 属性，并实现 `.render(self, data, media_type=None, renderer_context=None)` 方法。

该方法应返回一个字符串，它将用作 HTTP 响应的主体。

传递给 `.render()` 方法的参数是：

`data`

请求数据，由 `Response()` 实例化时设置。

`media_type=None`

可选的。如果提供，这是接受的媒体类型，由内容协商 (content negotiation) 阶段确定。

依赖于客户端的 `Accept:` header，它可以比渲染器的 `media_type` 属性更具体，并且可能包含媒体类型参数。比如 `"application/json; nested=true"`。

`renderer_context=None`

可选的。如果提供，它是视图提供的上下文信息字典。

默认情况下，这将包括以下键：`view`，`request`，`response`，`args`，`kwargs`。

举个栗子

以下是一个示例纯文本渲染器，它将返回带有数据参数的响应作为响应的内容。

```
1. from django.utils.encoding import smart_unicode
2. from rest_framework import renderers
3.
4.
5. class PlainTextRenderer(renderers.BaseRenderer):
6.     media_type = 'text/plain'
7.     format = 'txt'
8.
9.     def render(self, data, media_type=None, renderer_context=None):
10.         return data.encode(self.charset)
```

设置 charset

默认情况下，渲染器类被假定为使用 UTF-8 编码。要使用不同的编码，请在渲染器上设置

`charset` 属性。

```

1. class PlainTextRenderer(renderers.BaseRenderer):
2.     media_type = 'text/plain'
3.     format = 'txt'
4.     charset = 'iso-8859-1'
5.
6.     def render(self, data, media_type=None, renderer_context=None):
7.         return data.encode(self.charset)

```

请注意，如果渲染器类返回一个 `unicode` 字符串，则响应内容将被 `Response` 类强制为一个 `bytestring`，请在渲染器上设置 `charset` 属性用于确定编码。

如果渲染器返回代表原始二进制内容的字符串，则应将 `charset` 值设置为 `None`，这将确保响应的 `Content-Type` header 不会设置 `charset` 值。

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

```

1. class JPEGRenderer(renderers.BaseRenderer):
2.     media_type = 'image/jpeg'
3.     format = 'jpg'
4.     charset = None
5.     render_style = 'binary'
6.
7.     def render(self, data, media_type=None, renderer_context=None):
8.         return data

```

渲染器的高级用法

你可以使用 REST framework 的渲染器来做一些非常灵活的事情。例如...

- 根据请求的 `media type`，提供来自同一端点的平面或嵌套 (`flat` or `nested`) 表示。
- 同时处理常规 HTML 网页和来自相同端点的基于 JSON 的 API 响应。
- 指定 API 客户端使用的多种 HTML 表示形式。
- 不用明确指定渲染器的 `media type`，例如使用 `media_type = 'image/*'`，并使用 `Accept` header 改变响应的编码。

media type 的变化

在某些情况下，可能希望视图根据接受的 `media type` 使用不同的序列化风格。如果需要这样做，可以访问 `request.accepted_renderer` 以确定将用于响应的协商 (`negotiate`) 渲染器。

例如：

```

1. @api_view(('GET',))
2. @renderer_classes((TemplateHTMLRenderer, JSONRenderer))
3. def list_users(request):
4.     """
5.     A view that can return JSON or HTML representations
6.     of the users in the system.
7.     """
8.     queryset = Users.objects.filter(active=True)
9.
10.    if request.accepted_renderer.format == 'html':
11.        # TemplateHTMLRenderer takes a context dict,
12.        # and additionally requires a 'template_name'.
13.        # It does not require serialization.
14.        data = {'users': queryset}
15.        return Response(data, template_name='list_users.html')
16.
17.    # JSONRenderer requires serialized data as normal.
18.    serializer = UserSerializer(instance=queryset)
19.    data = serializer.data
20.    return Response(data)

```

不明确的 media type

在某些情况下，可能需要渲染器来提供一系列 media type 。在这种情况下，可以通过使用

`media_type` 值（如 `image/*` 或 `*/*`）来指定它应该响应的 media type 。

如果没有明确指定渲染器的 media type ，则应确保在返回响应时使用 `content_type` 属性明确指定 media type 。比如：

```

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

```

设计你的 media type

出于许多 Web API 的目的，具有超链接关系的简单 JSON 响应可能就足够了。如果你想完全融入 RESTful 设计和 HATEOAS，则需要更详细地考虑 media type 的设计和使用。

HTML error 视图

通常情况下，无论处理常规响应还是引发异常的响应（例如 `Http404` 或 `PermissionDenied`

异常) 或 `APIException` 的子类引起的响应, 渲染器都会有相同的表现。

如果您使用的是 `TemplateHTMLRenderer` 或 `StaticHTMLRenderer`, 并且引发异常, 则行为会稍有不同, 反映了 Django 对错误视图的默认处理。

由 HTML 渲染器引发和处理的异常将尝试使用以下方法之一 (按优先顺序) 进行渲染。

- 加载并渲染模板 `{status_code}.html`。
- 加载并渲染模板 `api_exception.html`。
- 渲染 HTTP 状态码和文本, 例如 “404 Not Found”。

模板将使用 `RequestContext` 进行渲染, 其中包含 `status_code` 和 `details` 键。

注意: 如果 `DEBUG = True`, 则会显示 Django 的标准错误回溯页面, 而不是显示 HTTP 状态码和文本。

序列化

- Serializers

- 申明序列化类
- 序列化对象
- 反序列化对象
- 保存实例
 - 将附加属性传递给 `.save()`
 - 直接覆盖 `.save()`。
- 验证
 - 数据验证时抛出异常
 - 字段级验证
 - 对象级验证
 - 验证器
- 访问初始数据和实例
- 部分更新
- 处理嵌套对象
- 可写嵌套表示
 - 为嵌套表示书写 `.create()` 方法
 - 为嵌套表示书写 `.update()` 方法
 - 在模型管理器类中保存相关的实例
- 处理多个对象
 - 序列化多个对象
 - 反序列化多个对象
- 包含额外的上下文

- ModelSerializer

- 检查 `ModelSerializer`
- 指定要包含的字段
- 指定嵌套序列化
- 显式指定字段
- 指定只读字段
- 其他关键字参数
- 关系字段
- 自定义字段映射
 - `field_class` 和 `field_kwargs` API

- HyperlinkedModelSerializer

- 绝对和相对 URL
- 如何确定超链接视图
- 更改 URL 字段名称

- `ListSerializer`
 - `allow_empty`
 - 自定义 `ListSerializer` 行为
 - 自定义多个对象的创建
 - 自定义多个对象的更新
 - 自定义 `ListSerializer` 初始化
- `BaseSerializer`
 - Read-only `BaseSerializer` 类
 - Read-write `BaseSerializer` 类
 - 创建新的基类
- `Serializer` 使用进阶
 - 重写序列化和反序列化行为
 - 继承序列化类
 - 动态修改字段

[官方原文链接](#)

Serializers

序列化器允许将诸如查询集和模型实例之类的复杂数据转换为原生 Python 数据类型，然后可以将它们轻松地呈现为 `JSON`，`XML` 或其他内容类型。序列化器还提供反序列化，在首次验证传入数据之后，可以将解析的数据转换回复杂类型。

REST framework 中的序列化类与 Django 的 `Form` 和 `ModelForm` 类非常相似。我们提供了一个 `Serializer` 类，它提供了一种强大的通用方法来控制响应的输出，以及一个 `ModelSerializer` 类，它为创建处理模型实例和查询集的序列化提供了有效的快捷方式。

申明序列化类

首先创建一个简单的对象用于示例：

```
1. from datetime import datetime
2.
3. class Comment(object):
4.     def __init__(self, email, content, created=None):
5.         self.email = email
6.         self.content = content
7.         self.created = created or datetime.now()
8.
9. comment = Comment(email='leila@example.com', content='foo bar')
```

声明一个序列化类，使用它来序列化和反序列化与 `Comment` 对象相对应的数据。

声明一个序列化类看起来非常类似于声明一个表单：

```
1. from rest_framework import serializers
2.
3. class CommentSerializer(serializers.Serializer):
4.     email = serializers.EmailField()
5.     content = serializers.CharField(max_length=200)
6.     created = serializers.DateTimeField()
```

序列化对象

现在可以使用 `CommentSerializer` 来序列化评论或评论列表。同样，使用 `Serializer` 类看起来很像使用 `Form` 类。

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

此时已经将模型实例转换为 Python 原生数据类型。为了完成序列化过程，将数据渲染为 `json`。

```
1. from rest_framework.renderers import JSONRenderer
2.
3. json = JSONRenderer().render(serializer.data)
4. json
5. # b'{"email": "leila@example.com", "content": "foo bar", "created": "2016-01-27T15:17:10.375877"}'
```

反序列化对象

反序列化是相似的。首先我们将一个流解析为 Python 原生数据类型...

```
1. from django.utils.six import BytesIO
2. from rest_framework.parsers import JSONParser
3.
4. stream = BytesIO(json)
5. data = JSONParser().parse(stream)
```

...然后我们将这些原生数据类型恢复成通过验证的数据字典。

```

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

```

保存实例

如果希望能够基于验证的数据返回完整的对象实例，则需要实现 `.create()` 和 `.update()` 方法中的一个或两个。例如：

```

1. class CommentSerializer(serializers.Serializer):
2.     email = serializers.EmailField()
3.     content = serializers.CharField(max_length=200)
4.     created = serializers.DateTimeField()
5.
6.     def create(self, validated_data):
7.         return Comment(**validated_data)
8.
9.     def update(self, instance, validated_data):
10.        instance.email = validated_data.get('email', instance.email)
11.        instance.content = validated_data.get('content', instance.content)
12.        instance.created = validated_data.get('created', instance.created)
13.        return instance

```

如果对象实例与 Django 模型相对应，还需要确保这些方法将对象保存到数据库。如果 `Comment` 是一个 Django 模型，这些方法可能如下所示：

```

1.     def create(self, validated_data):
2.         return Comment.objects.create(**validated_data)
3.
4.     def update(self, instance, validated_data):
5.         instance.email = validated_data.get('email', instance.email)
6.         instance.content = validated_data.get('content', instance.content)
7.         instance.created = validated_data.get('created', instance.created)
8.         instance.save()
9.         return instance

```

现在，当反序列化数据时，我们可以调用 `.save()` 根据验证的数据返回一个对象实例。

```
1. comment = serializer.save()
```

调用 `.save()` 将创建一个新实例或更新现有实例，具体取决于在实例化序列化类时是否传递了现有实例：

```
1. # .save() will create a new instance.
2. serializer = CommentSerializer(data=data)
3.
4. # .save() will update the existing `comment` instance.
5. serializer = CommentSerializer(comment, data=data)
```

`.create()` 和 `.update()` 方法都是可选的。您可以都不实现，或者实现其中的一个或两个，具体取决于你的序列化类的用例。

将附加属性传递给 `.save()`

有时你会希望你的视图代码能够在保存实例的时候注入额外的数据。这些附加数据可能包含当前用户，当前时间或其他任何不属于请求数据的信息。

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

调用 `.create()` 或 `.update()` 时，任何其他关键字参数都将包含在 `validated_data` 参数中。

直接覆盖 `.save()`。

在某些情况下，`.create()` 和 `.update()` 方法名称可能没有意义。例如，在“联系人表单”中，我们可能不会创建新实例，而是发送电子邮件或其他消息。

在这些情况下，可以选择直接覆盖 `.save()`，因为它更具可读性和有意义性。

举个栗子：

```
1. class ContactForm(serializers.Serializer):
2.     email = serializers.EmailField()
3.     message = serializers.CharField()
4.
5.     def save(self):
6.         email = self.validated_data['email']
7.         message = self.validated_data['message']
8.         send_email(from=email, message=message)
```

请注意，在上面的情况下，必须直接访问 `serializer.validated_data` 属性。

验证

在反序列化数据时，你总是需要在尝试访问验证数据之前调用 `is_valid()`，或者保存对象实例。如果发生任何验证错误，那么 `.errors` 属性将包含一个代表错误消息的字典。例如：

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

字典中的每个键都是字段名称，值是与该字段相对应的错误消息（字符串列表）。`non_field_errors` 键也可能存在，并会列出任何常规验证错误。可以使用 `NON_FIELD_ERRORS_KEY`（在 `settings` 文件中设置）来定制 `non_field_errors` 关键字的名称。

反序列化 `item` 列表时，错误将作为代表每个反序列化 `item` 的字典列表返回。

数据验证时抛出异常

`.is_valid()` 方法带有一个可选的 `raise_exception` 标志，如果存在验证错误，将导致它引发 `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，并且默认情况下将返回 `HTTP 400 Bad Request`。

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

字段级验证

你可以通过向 `Serializer` 子类添加 `.validate_<field_name>` 方法来指定自定义字段级验证。这些与 Django 表单上的 `.clean_<field_name>` 方法类似。

这些方法只有一个参数，就是需要验证的字段值。

您的 `validate_<field_name>` 方法应返回验证值或引发 `serializers.ValidationError`。

例如：

```

1. from rest_framework import serializers
2.
3. class BlogPostSerializer(serializers.Serializer):
4.     title = serializers.CharField(max_length=100)
5.     content = serializers.CharField()
6.
7.     def validate_title(self, value):
8.         """
9.         Check that the blog post is about Django.
10.        """
11.        if 'django' not in value.lower():
12.            raise serializers.ValidationError("Blog post is not about Django")
13.        return value

```

注意：如果你的序列化程序中声明的 `<field_name>` 参数为 `required = False`，那么如果未包含该字段，则不会执行此验证步骤。

对象级验证

如果要对多个字段进行其他的验证，请将一个名为 `.validate()` 的方法添加到您的 `Serializer` 子类中。这个方法只有一个参数，它是一个字段值（`field` - `value`）的字典。如果有必要，它应该引发一个 `ValidationError`，或者只是返回验证的值。例如：

```

1. from rest_framework import serializers
2.
3. class EventSerializer(serializers.Serializer):
4.     description = serializers.CharField(max_length=100)
5.     start = serializers.DateTimeField()
6.     finish = serializers.DateTimeField()
7.
8.     def validate(self, data):
9.         """
10.        Check that the start is before the stop.
11.        """
12.        if data['start'] > data['finish']:
13.            raise serializers.ValidationError("finish must occur after start")
14.        return data

```

验证器

序列化器上的各个字段可以包含验证器，方法是在字段实例上声明它们，例如：


```

1. def multiple_of_ten(value):
2.     if value % 10 != 0:
3.         raise serializers.ValidationError('Not a multiple of ten')
4.
5. class GameRecord(serializers.Serializer):
6.     score = IntegerField(validators=[multiple_of_ten])
7.     ...

```

序列化类还可以包含应用于整个字段数据集的可重用验证器。这些验证器是通过在内部的 `Meta` 类中声明它们来包含的，如下所示：

```

1. class EventSerializer(serializers.Serializer):
2.     name = serializers.CharField()
3.     room_number = serializers.IntegerField(choices=[101, 102, 103, 201])
4.     date = serializers.DateField()
5.
6.     class Meta:
7.         # Each room only has one event per day.
8.         validators = UniqueTogetherValidator(
9.             queryset=Event.objects.all(),
10.            fields=['room_number', 'date']
11.        )

```

看不懂没关系哦，更多关于验证的内容，以后还会说到。

访问初始数据和实例

将初始对象或查询集传递给序列化类实例时，该对象将作为 `.instance` 提供。如果没有传递初始对象，则 `.instance` 属性将为 `None`。

将数据传递给序列化类实例时，未修改的数据将作为 `.initial_data` 提供。如果 `data` 关键字参数未被传递，那么 `.initial_data` 属性将不存在。

部分更新

默认情况下，序列化程序必须为所有必填字段传递值，否则会引发验证错误。您可以使用 `partial` 参数以允许部分更新。

```

1. # Update `comment` with partial data
2. serializer = CommentSerializer(comment, data={'content': u'foo bar'},
    partial=True)

```

处理嵌套对象

前面的例子适用于处理只具有简单数据类型的对象，但有时还需要能够表示更复杂的对象，其中对象的某些属性可能不是简单的数据类型，如字符串，日期或整数。

`Serializer` 类本身就是一种 `Field`，可以用来表示一个对象类型嵌套在另一个对象类型中的关系。

```
1. class UserSerializer(serializers.Serializer):
2.     email = serializers.EmailField()
3.     username = serializers.CharField(max_length=100)
4.
5. class CommentSerializer(serializers.Serializer):
6.     user = UserSerializer()
7.     content = serializers.CharField(max_length=200)
8.     created = serializers.DateTimeField()
```

如果嵌套对象可以是 `None` 值，则应将 `required = False` 标志传递给嵌套的序列化类。

```
1. class CommentSerializer(serializers.Serializer):
2.     user = UserSerializer(required=False) # May be an anonymous user.
3.     content = serializers.CharField(max_length=200)
4.     created = serializers.DateTimeField()
```

同样，如果嵌套对象是一个列表，则应将 `many = True` 标志传递给嵌套的序列化类。

```
1. class CommentSerializer(serializers.Serializer):
2.     user = UserSerializer(required=False)
3.     edits = EditItemSerializer(many=True) # A nested list of 'edit' items.
4.     content = serializers.CharField(max_length=200)
5.     created = serializers.DateTimeField()
```

可写嵌套表示

在处理支持反序列化数据的嵌套表示时，嵌套对象的任何错误都将嵌套在嵌套对象的字段名称下。

```
1. serializer = CommentSerializer(data={'user': {'email': 'foobar', 'username':
2.     'doe'}, 'content': 'baz'})
3. serializer.is_valid()
4. # False
5. serializer.errors
```

```
5. # {'user': {'email': [u'Enter a valid e-mail address.']}, 'created': [u'This field is required.']}
```

同样，`.validated_data` 属性将包含嵌套的数据结构。

为嵌套表示书写 `.create()` 方法

如果你支持可写嵌套表示，则需要编写处理保存多个对象的 `.create()` 或 `.update()` 方法。

以下示例演示如何处理使用嵌套配置文件对象创建用户。

```
1. class UserSerializer(serializers.ModelSerializer):
2.     profile = ProfileSerializer()
3.
4.     class Meta:
5.         model = User
6.         fields = ('username', 'email', 'profile')
7.
8.     def create(self, validated_data):
9.         profile_data = validated_data.pop('profile')
10.        user = User.objects.create(**validated_data)
11.        Profile.objects.create(user=user, **profile_data)
12.        return user
```

为嵌套表示书写 `.update()` 方法

对于更新，您需要仔细考虑如何处理关系更新。例如，如果关系的数据是 `None` 或没有提供，则应发生以下哪种情况？

- 在数据库中将关系设置为 `NULL`。
- 删除关联的实例。
- 忽略数据并保持原样。
- 引发验证错误。

以下是我们以前的 `UserSerializer` 类中的 `.update()` 方法的示例。

```
1.     def update(self, instance, validated_data):
2.         profile_data = validated_data.pop('profile')
3.         # Unless the application properly enforces that this field is
4.         # always set, the follow could raise a `DoesNotExist`, which
5.         # would need to be handled.
6.         profile = instance.profile
7.
```

```

8.         instance.username = validated_data.get('username', instance.username)
9.         instance.email = validated_data.get('email', instance.email)
10.        instance.save()
11.
12.        profile.is_premium_member = profile_data.get(
13.            'is_premium_member',
14.            profile.is_premium_member
15.        )
16.        profile.has_support_contract = profile_data.get(
17.            'has_support_contract',
18.            profile.has_support_contract
19.        )
20.        profile.save()
21.
22.        return instance

```

因为嵌套创建和更新的行为可能不明确，并且可能需要相关模型之间的复杂依赖关系，所以 REST framework 3 要求你始终明确写入这些方法。默认的 `ModelSerializer` 的 `.create()` 和 `.update()` 方法不包括对可写嵌套表示的支持。

不过，有第三方软件包可用，如支持自动可写嵌套表示的 [DRF Writable Nested](#)。

在模型管理器类中保存相关的实例

在序列化类中保存多个相关实例的另一种方法是编写自定义模型管理器类。

例如，假设我们希望确保 `User` 实例和 `Profile` 实例始终作为一对创建。我们可能会编写一个类似下面的自定义管理器类：

```

1. class UserManager(models.Manager):
2.     ...
3.
4.     def create(self, username, email, is_premium_member=False,
5.        has_support_contract=False):
6.         user = User(username=username, email=email)
7.         user.save()
8.         profile = Profile(
9.             user=user,
10.            is_premium_member=is_premium_member,
11.            has_support_contract=has_support_contract
12.        )
13.        profile.save()
14.        return user

```

此管理器类现在更好地封装了用户实例和配置文件实例始终在同一时间创建。现在可以重新编写序列化类上的 `.create()` 方法，以使用新的管理类方法。

```
1. def create(self, validated_data):
2.     return User.objects.create(
3.         username=validated_data['username'],
4.         email=validated_data['email']
5.         is_premium_member=validated_data['profile']['is_premium_member']
6.         has_support_contract=validated_data['profile']['has_support_contract']
7.     )
```

处理多个对象

`Serializer` 类还可以处理序列化或反序列化对象列表。

序列化多个对象

要序列化查询集或对象列表而不是单个对象实例，在实例化序列化类时，应该传递 `many=True` 标志。然后，您可以传递要序列化的查询集或对象列表。

```
1. queryset = Book.objects.all()
2. serializer = BookSerializer(queryset, many=True)
3. serializer.data
4. # [
5. #     {'id': 0, 'title': 'The electric kool-aid acid test', 'author': 'Tom
6.     wolfe'},
7. #     {'id': 1, 'title': 'If this is a man', 'author': 'Primo Levi'},
8. #     {'id': 2, 'title': 'The wind-up bird chronicle', 'author': 'Haruki
9.     Murakami'}
10. # ]
```

反序列化多个对象

反序列化多个对象的默认行为是支持多个对象创建，但不支持多个对象更新。

包含额外的上下文

除了被序列化的对象外，还有一些情况需要为序列化类提供额外的上下文。一种常见的情况是，如果你使用的是包含超链接关系的序列化类，则需要序列化类访问当前请求，以便它可以正确生成完全限定的 URL。

在实例化序列化对象时，你可以通过传递上下文参数来提供任意附加上下文。例如：

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

通过访问 `self.context` 属性，可以在任何序列化对象字段逻辑中使用上下文字典，例如自定义的 `.to_representation()` 方法。

ModelSerializer

通常你会想要序列化类紧密地映射到 Django 模型定义上。

`ModelSerializer` 类提供了一个快捷方式，可让你自动创建一个 `Serializer` 类，其中的字段与模型类字段对应。

`ModelSerializer` 类与常规 `Serializer` 类相同，不同之处在于：

- 它会根据模型自动生成一组字段。
- 它会自动为序列化类生成验证器，例如 `unique_together` 验证器。
- 它包含 `.create()` 和 `.update()` 的简单默认实现。

声明 `ModelSerializer` 如下所示：

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

默认情况下，该类中的所有模型类字段将被映射为相应的序列化类字段。

任何关系（如模型上的外键）都将映射到 `PrimaryKeyRelatedField`。除非在序列化关系文档中指定，否则默认不包括反向关系。

检查 `ModelSerializer`

序列化类能够生成一个表示字符串，可以让你充分检查其字段的状态。在使用 `ModelSerializer` 进行工作时，这是特别有用的，你需要确定它为你自动创建了哪些字段和验证器。

为此，使用 `python manage.py shell` 进入 Django shell，然后导入序列化类，实例化它并打印对象表示形式...

```

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

```

指定要包含的字段

如果你只希望在模型序列化程序中使用默认字段的子集，则可以使用 `fields` 或 `exclude` 选项来完成此操作，就像使用 `ModelForm` 一样。强烈建议你显式使用 `fields` 属性序列化的所有字段。这将使你不太可能在模型更改时无意中暴露数据。

举个栗子：

```

1. class AccountSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         model = Account
4.         fields = ('id', 'account_name', 'users', 'created')

```

你还可以将 `fields` 属性设置为特殊值 `'__all__'`，以指示应该使用模型中的所有字段。

```

1. class AccountSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         model = Account
4.         fields = '__all__'

```

你可以将 `exclude` 属性设置为从序列化程序中排除的字段列表。

```

1. class AccountSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         model = Account
4.         exclude = ('users',)

```

在上面的示例中，如果 `Account` 模型有3个字段 `account_name`，`users` 和 `created`，则会导致字段 `account_name` 和 `created` 被序列化。

`fields` 和 `exclude` 属性中的名称通常映射到模型类的模型字段。

或者 `fields` 选项中的名称可以映射成属性或方法。而不会变成模型类中的参数。

从版本 3.3.0 开始，必须提供其中一个属性 `fields` 或 `exclude`。

指定嵌套序列化

默认的 `ModelSerializer` 使用主键进行关联，但你也可以使用 `depth` 选项轻松生成嵌套表示（自关联）：

```
1. class AccountSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         model = Account
4.         fields = ('id', 'account_name', 'users', 'created')
5.         depth = 1
```

`depth` 选项应设置为一个整数值，该值指示在还原为平面表示之前应该遍历的关联的深度。

如果你想自定义序列化的方式，你需要自己定义字段。

显式指定字段

你可以将额外的字段添加到 `ModelSerializer`，或者通过在类上声明字段来覆盖默认字段，就像你对 `Serializer` 类所做的那样。

```
1. class AccountSerializer(serializers.ModelSerializer):
2.     url = serializers.CharField(source='get_absolute_url', read_only=True)
3.     groups = serializers.PrimaryKeyRelatedField(many=True)
4.
5.     class Meta:
6.         model = Account
```

额外的字段可以对应于模型上的任何属性或可调用的字段。

指定只读字段

你可能希望将多个字段指定为只读。不要显式给每个字段添加 `read_only = True` 属性，你可以使用快捷方式 `Meta` 选项 `read_only_fields`。

该选项应该是字段名称的列表或元组，声明如下：

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



```
5. read_only_fields = ('account_name',)
```

含有 `editable = False` 的模型字段，`AutoField` 字段默认设置为只读，并且不需要添加到 `read_only_fields` 选项。

注意：有一种特殊情况，只读字段是模型级别的 `unique_together` 约束的一部分。在这种情况下，序列化类需要验证约束该字段，但也不能由用户编辑。

处理这个问题的正确方法是在序列化类中明确指定字段，同时提供 `read_only = True` 和 `default = ...` 关键字参数。

其中一个例子是与当前认证 `User` 的只读关系，它与另一个标识符是 `unique_together`。在这种情况下，你会像这样声明用户字段：

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

关于验证以后还会再说

其他关键字参数

还有一个快捷方式允许你使用 `extra_kwargs` 选项在字段上指定任意附加关键字参数。与 `read_only_fields` 的情况一样，这意味着你不需要在序列化类中显式声明该字段。

该选项是一个字典，将字段名称映射到关键字参数字典。例如：

```
1. class CreateUserSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         model = User
4.         fields = ('email', 'username', 'password')
5.         extra_kwargs = {'password': {'write_only': True}}
6.
7.     def create(self, validated_data):
8.         user = User(
9.             email=validated_data['email'],
10.            username=validated_data['username']
11.        )
12.        user.set_password(validated_data['password'])
13.        user.save()
14.        return user
```

关系字段

序列化模型实例时，可以选择多种不同的方式来表示关系。`ModelSerializer` 的默认表示是使用相关实例的主键。

其他表示形式包括使用超链接序列化，序列化完整嵌套表示形式或使用自定义表示形式序列化。

自定义字段映射

`ModelSerializer` 类还公开了一个可以覆盖的 API，以便在实例化序列化对象时更改序列化对象的字段。

通常，如果 `ModelSerializer` 没办法生成默认需要的字段，那么你应该将它们明确地添加到类中，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，以定义如何为给定模型创建序列化对象的字段。

`.serializer_field_mapping`

Django 模型类到 REST framework 序列化类的映射。你可以重写此映射来更改应该用于每个模型类的默认序列化类。

`.serializer_related_field`

该属性应该是序列化器字段类，默认情况下用于关系字段。

对于 `ModelSerializer`，它默认为 `PrimaryKeyRelatedField`。

对于 `HyperlinkedModelSerializer`，它默认为 `serializers.HyperlinkedRelatedField`。

`serializer_url_field`

序列化器字段类，应该用于序列化类中的任何 `url` 字段。

默认是 `serializers.HyperlinkedIdentityField`。

`serializer_choice_field`

序列化器字段类，应该用于序列化程序中的任何选择字段。

默认是 `serializers.ChoiceField`。

field_class 和 field_kwargs API

调用以下方法来确定应该自动包含在序列化程序中的每个字段的类和关键字参数。这些方法都应返回两个元组 `(field_class, field_kwargs)`。

`.build_standard_field(self, field_name, model_field)`

调用以生成映射到标准模型字段的序列化器字段。

默认实现基于 `serializer_field_mapping` 属性返回序列化类。

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

调用以生成映射到关系模型字段的序列化器字段。

默认实现基于 `serializer_relational_field` 属性返回一个序列化类。

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

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

当 `depth` 选项已设置时，调用以生成映射到关系模型字段的序列化程序字段。

默认实现动态创建一个基于 `ModelSerializer` 或 `HyperlinkedModelSerializer` 的嵌套序列化类。

`nested_depth` 将是 `depth` 选项的值减 1。

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

```
.build_property_field(self, field_name, model_class)
```

调用以生成映射到模型类上的属性或零参数方法的序列化器字段。

默认实现返回一个 `ReadOnlyField` 类。

```
.build_url_field(self, field_name, model_class)
```

被调用来为序列化器自己的 `url` 字段生成一个序列化器字段。

默认实现返回一个 `HyperlinkedIdentityField` 类。

```
.build_unknown_field(self, field_name, model_class)
```

当字段名称未映射到任何模型字段或模型属性时调用。默认实现会引发错误。但是子类可以自定义这种行为。

HyperlinkedModelSerializer

`HyperlinkedModelSerializer` 类与 `ModelSerializer` 类相似，只不过它使用超链接来表示关系而不是主键。

默认情况下，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` 序列化器字段来表示，并且模型上的任何关系都将使用 `HyperlinkedRelatedField` 序列化器字段来表示。

你可以通过将主键添加到 `fields` 选项来明确包含主键，例如：

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

绝对和相对 URL

在实例化 `HyperlinkedModelSerializer` 时，必须在序列化上下文中包含当前请求，例如：

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

这样做将确保超链接可以包含适当的主机名，以便生成完全限定的 URL，例如：

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

而不是相对的 URL，例如：

```
1. /accounts/1/
```

如果你确实想要使用相对 URL，则应该在序列化上下文中显式传递 `{'request': None}`。

如何确定超链接视图

需要确定哪些视图应该用于超链接到模型实例。

默认情况下，超链接预期对应于与样式 `'{model_name}-detail'` 匹配的视图名称，并通过 `pk` 关键字参数查找实例。

您可以使用 `extra_kwargs` 设置中的 `view_name` 和 `lookup_field` 选项覆盖 URL 字段视图名称和查找字段，如下所示：

```
1. class AccountSerializer(serializers.HyperlinkedModelSerializer):
2.     class Meta:
3.         model = Account
4.         fields = ('account_url', 'account_name', 'users', 'created')
5.         extra_kwargs = {
6.             'url': {'view_name': 'accounts', 'lookup_field': 'account_name'},
```

```

7.         'users': {'lookup_field': 'username'}
8.     }

```

或者，可以显式设置序列化类中的字段。例如：

```

1. class AccountSerializer(serializers.HyperlinkedModelSerializer):
2.     url = serializers.HyperlinkedIdentityField(
3.         view_name='accounts',
4.         lookup_field='slug'
5.     )
6.     users = serializers.HyperlinkedRelatedField(
7.         view_name='user-detail',
8.         lookup_field='username',
9.         many=True,
10.        read_only=True
11.    )
12.
13.    class Meta:
14.        model = Account
15.        fields = ('url', 'account_name', 'users', 'created')

```

提示：正确地匹配超链接和 `URL conf` 有时可能有点困难。打印 `HyperlinkedModelSerializer` 实例的 `repr` 是一种特别有用的方法，可以准确检查这些关系预期映射的 `view name` 和 `lookup field`。

更改 URL 字段名称

URL 字段的名称默认为 `'url'`。可以使用 `URL_FIELD_NAME`（在 `settings` 文件）全局覆盖此设置。

ListSerializer

`ListSerializer` 类提供了一次序列化和验证多个对象的行为。你通常不需要直接使用 `ListSerializer`，而应该在实例化序列化类时简单地传递 `many=True`。

当一个序列化类被实例化并且 `many = True` 被传递时，一个 `ListSerializer` 实例将被创建。序列化类成为父级 `ListSerializer` 的子级

以下参数也可以传递给 `ListSerializer` 字段或传递了 `many = True` 的序列化类：

`allow_empty`

默认情况下为 `True`，但如果要禁止将空列表作为有效输入，则可将其设置为 `False`。

自定义 `ListSerializer` 行为

有几种情况可能需要自定义 `ListSerializer` 行为。例如：

- 希望提供对列表的特定验证，例如检查一个元素是否与列表中的另一个元素不冲突。
- 想要自定义多个对象的创建或更新行为。

对于这些情况，可以通过使用序列化类的 `Meta` 类中的 `list_serializer_class` 选项来修改传递了 `many=True` 时使用的类。

```
1. class CustomListSerializer(serializers.ListSerializer):
2.     ...
3.
4. class CustomSerializer(serializers.Serializer):
5.     ...
6.     class Meta:
7.         list_serializer_class = CustomListSerializer
```

自定义多个对象的创建

创建多个对象的默认实现是简单地为列表中的每个 `item` 调用 `.create()`。如果要自定义此行为，则需要在传递 `many=True` 时自定义 `ListSerializer` 类上的 `.create()` 方法。

```
1. class BookListSerializer(serializers.ListSerializer):
2.     def create(self, validated_data):
3.         books = [Book(**item) for item in validated_data]
4.         return Book.objects.bulk_create(books)
5.
6. class BookSerializer(serializers.Serializer):
7.     ...
8.     class Meta:
9.         list_serializer_class = BookListSerializer
```

自定义多个对象的更新

默认情况下，`ListSerializer` 类不支持多对象更新。这是因为插入和删除预期的行为是不明确的。

为了支持多对象更新，你需要重写 `update` 方法。在编写你的多对象更新代码时，一定要记住以下几点：

- 如何确定应该为数据列表中的每个 `item` 更新哪个实例？
- 插入应该如何处理？它们是无效的，还是创建新对象？

- 应该如何处理删除？它们是否暗示了对象删除，或者删除了一段关系？它们应该被忽略，还是无效？
- 如何处理排序？改变两个 `item` 的位置是否意味着状态的改变或者被忽略？

你需要为实例序列化类添加一个显式 `id` 字段。默认的隐式生成的 `id` 字段被标记为 `read_only`。这会导致它在更新时被删除。一旦你明确声明它，它将在列表序列化类的更新方法中可用。

下面是你如何选择实现多对象更新的示例：

```

1. class BookListSerializer(serializers.ListSerializer):
2.     def update(self, instance, validated_data):
3.         # Maps for id->instance and id->data item.
4.         book_mapping = {book.id: book for book in instance}
5.         data_mapping = {item['id']: item for item in validated_data}
6.
7.         # Perform creations and updates.
8.         ret = []
9.         for book_id, data in data_mapping.items():
10.            book = book_mapping.get(book_id, None)
11.            if book is None:
12.                ret.append(self.child.create(data))
13.            else:
14.                ret.append(self.child.update(book, data))
15.
16.         # Perform deletions.
17.         for book_id, book in book_mapping.items():
18.             if book_id not in data_mapping:
19.                 book.delete()
20.
21.         return ret
22.
23. class BookSerializer(serializers.Serializer):
24.     # We need to identify elements in the list using their primary key,
25.     # so use a writable field here, rather than the default which would be
26.     # read-only.
27.     id = serializers.IntegerField()
28.     ...
29.
30.     class Meta:
31.         list_serializer_class = BookListSerializer

```

自定义 ListSerializer 初始化

当具有 `many=True` 的序列化类实例化时，我们需要确定哪些参数和关键字参数应该传递给子级 `Serializer` 类和父级 `ListSerializer` 类的 `__init__()` 方法。

默认的实现是将所有参数传递给两个类，除了 `validators` 和任何自定义关键字参数，这两个参数都假定用于子序列化类。

有时你可能需要明确指定在传递 `many=True` 时如何实例化子类和父类。您可以使用 `many_init` 类方法来完成此操作。

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

BaseSerializer

`BaseSerializer` 类可以用来方便地支持其他序列化和反序列化风格。

这个类实现了与 `Serializer` 类相同的基本 API：

- `.data` - 返回传出的原始表示。
- `.is_valid()` - 反序列化并验证传入的数据。
- `.validated_data` - 返回验证的传入数据。
- `.errors` - 在验证期间返回错误。
- `.save()` - 将验证的数据保存到对象实例中。

有四种方法可以被覆盖，这取决于你希望序列化类支持的功能：

- `.to_representation()` - 重写此操作以支持序列化，用于读取操作。
- `.to_internal_value()` - 重写此操作以支持反序列化，以用于写入操作。
- `.create()` 和 `.update()` - 覆盖其中一个或两个以支持保存实例。

因为这个类提供了与 `Serializer` 类相同的接口，所以你可以像现有的常规 `Serializer` 或 `ModelSerializer` 一样，将它与基于类的通用视图一起使用。

唯一不同的是，`BaseSerializer` 类不会在可浏览的 API 中生成 HTML 表单。这是因为它们返回的数据不包含所有的字段信息，这些字段信息允许将每个字段渲染为合适的 HTML 输入。

Read-only `BaseSerializer` 类

要使用 `BaseSerializer` 类实现只读序列化类，我们只需重写 `.to_representation()` 方法。让我们来看一个使用简单的 Django 模型的示例：

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

创建用于将 `HighScore` 实例转换为基本数据类型的只读序列化类非常简单。

```
1. class HighScoreSerializer(serializers.BaseSerializer):
2.     def to_representation(self, obj):
3.         return {
4.             'score': obj.score,
5.             'player_name': obj.player_name
6.         }
```

我们现在可以使用这个类来序列化单个 `HighScore` 实例：

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

或者用它来序列化多个实例：

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

Read-write BaseSerializer 类

要创建一个可读写的序列化类，我们首先需要实现一个 `.to_internal_value()` 方法。此方法返回将用于构造对象实例的验证值，并且如果提供的数据格式不正确，则可能引发 `ValidationError`。

一旦实现 `.to_internal_value()`，基本验证 API 将在序列化器中可用，并且你将能够使用 `.is_valid()`，`.validated_data` 和 `.errors`。

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

以下是我们之前的 `HighScoreSerializer` 的一个完整示例，该示例已更新为支持读取和写入操作。

```

1. class HighScoreSerializer(serializers.BaseSerializer):
2.     def to_internal_value(self, data):
3.         score = data.get('score')
4.         player_name = data.get('player_name')
5.
6.         # Perform the data validation.
7.         if not score:
8.             raise ValidationError({
9.                 'score': 'This field is required.'
10.            })
11.         if not player_name:
12.             raise ValidationError({
13.                 'player_name': 'This field is required.'
14.            })
15.         if len(player_name) > 10:
16.             raise ValidationError({
17.                 'player_name': 'May not be more than 10 characters.'
18.            })
19.
20.         # Return the validated values. This will be available as
21.         # the `.validated_data` property.
22.         return {
23.             'score': int(score),
24.             'player_name': player_name
25.         }
26.
27.     def to_representation(self, obj):
28.         return {
29.             'score': obj.score,
30.             'player_name': obj.player_name
31.         }
32.
33.     def create(self, validated_data):
34.         return HighScore.objects.create(**validated_data)

```

创建新的基类

如果你希望实现新的泛型序列化类来处理特定的序列化风格，或者与可选的存储后端进行集成，那么

`BaseSerializer` 类也很有用。

以下类是可以处理将任意对象强制转换为基本表示形式的泛型序列化类的示例。

```
1. class ObjectSerializer(serializers.BaseSerializer):
2.     """
3.     A read-only serializer that coerces arbitrary complex objects
4.     into primitive representations.
5.     """
6.     def to_representation(self, obj):
7.         for attribute_name in dir(obj):
8.             attribute = getattr(obj, attribute_name)
9.             if attribute_name.startswith('_'):
10.                 # Ignore private attributes.
11.                 pass
12.             elif hasattr(attribute, '__call__'):
13.                 # Ignore methods and other callables.
14.                 pass
15.             elif isinstance(attribute, (str, int, bool, float, type(None))):
16.                 # Primitive types can be passed through unmodified.
17.                 output[attribute_name] = attribute
18.             elif isinstance(attribute, list):
19.                 # Recursively deal with items in lists.
20.                 output[attribute_name] = [
21.                     self.to_representation(item) for item in attribute
22.                 ]
23.             elif isinstance(attribute, dict):
24.                 # Recursively deal with items in dictionaries.
25.                 output[attribute_name] = {
26.                     str(key): self.to_representation(value)
27.                     for key, value in attribute.items()
28.                 }
29.             else:
30.                 # Force anything else to its string representation.
31.                 output[attribute_name] = str(attribute)
```

Serializer 使用进阶

重写序列化和反序列化行为

如果你需要更改序列化类的序列化或反序列化行为，可以通过覆盖 `.to_representation()` 或 `.to_internal_value()` 方法来实现。

以下原因可能需要重写这两个方法...

- 为新的序列化基类添加新行为。
- 稍微修改现有类的行为。
- 提高经常访问的 API 端点的序列化性能，以便返回大量数据。

这些方法的签名如下：

```
.to_representation(self, obj)
```

接受需要序列化的对象实例，并返回一个原始表示。通常这意味着返回一个内置 Python 数据类型的结构。可以处理的确切类型取决于您为 API 配置的渲染类。

可能会被重写以便修改表示风格。例如：

```
1. def to_representation(self, instance):
2.     """Convert `username` to lowercase."""
3.     ret = super().to_representation(instance)
4.     ret['username'] = ret['username'].lower()
5.     return ret
```

```
.to_internal_value(self, data)
```

将未验证的传入数据作为输入，并应返回将作为 `serializer.validated_data` 提供的验证数据。如果在序列化类上调用了 `.save()`，则返回值也将传递给 `.create()` 或 `.update()` 方法。

如果验证失败，则该方法会引发 `serializers.ValidationError(errors)`。 `errors` 参数应该是一个由字段名称（或 `settings.NON_FIELD_ERRORS_KEY`）映射到错误消息列表的字典。如果不需要改变反序列化行为，而是想提供对象级验证，则建议改为覆盖 `.validate()` 方法。

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。

继承序列化类

与 Django 表单类似，你可以通过继承来扩展和重用序列化类。这使你可以在父类上声明一组通用的字段或方法，然后可以在多个序列化类中使用它们。例如，

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

```

3.
4.     def validate_my_field(self):
5.         ...
6.
7. class MySerializer(MyBaseSerializer):
8.     ...

```

与 Django 的 `Model` 和 `ModelForm` 类一样，序列化类中的内部 `Meta` 类不会从其父类的内部 `Meta` 类中隐式继承。如果你想让 `Meta` 类继承父类，必须明确的指出。例如：

```

1. class AccountSerializer(MyBaseSerializer):
2.     class Meta(MyBaseSerializer.Meta):
3.         model = Account

```

通常我们建议不要在内部的 `Meta` 类中使用继承，而是显式声明所有选项。

动态修改字段

一旦序列化类初始化完毕，就可以使用 `.fields` 属性访问在序列化类中设置的字段字典。通过访问和修改这个属性可以达到动态地修改序列化类的目的。

直接修改 `fields` 参数允许你做一些有趣的事情，比如在运行时改变序列化字段的参数，而不是在声明序列化类的时候。

举个栗子：

例如，如果你希望能够设置序列化类在初始化时应使用哪些字段，你可以创建这样一个序列化类，如下所示：

```

1. class DynamicFieldsModelSerializer(serializers.ModelSerializer):
2.     """
3.     A ModelSerializer that takes an additional `fields` argument that
4.     controls which fields should be displayed.
5.     """
6.
7.     def __init__(self, *args, **kwargs):
8.         # Don't pass the 'fields' arg up to the superclass
9.         fields = kwargs.pop('fields', None)
10.
11.         # Instantiate the superclass normally
12.         super(DynamicFieldsModelSerializer, self).__init__(*args, **kwargs)
13.

```

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

这将允许你执行以下操作：

```
1.  >>> class UserSerializer(DynamicFieldsModelSerializer):
2.  >>>     class Meta:
3.  >>>         model = User
4.  >>>         fields = ('id', 'username', 'email')
5.  >>>
6.  >>> print UserSerializer(user)
7.  {'id': 2, 'username': 'jonwatts', 'email': 'jon@example.com'}
8.  >>>
9.  >>> print UserSerializer(user, fields=('id', 'email'))
10. {'id': 2, 'email': 'jon@example.com'}
```

序列化·字段

- `Serializer` 字段

- 核心参数

- `read_only`
 - `write_only`
 - `required`
 - `allow_null`
 - `default`
 - `source`
 - `validators`
 - `error_messages`
 - `label`
 - `help_text`
 - `initial`
 - `style`

- `Boolean` 字段

- `BooleanField`
 - `NullBooleanField`

- 字符串字段

- `CharField`
 - `EmailField`
 - `RegexField`
 - `SlugField`
 - `URLField`
 - `UUIDField`
 - `FilePathField`
 - `IPAddressField`

- 数字字段

- `IntegerField`
 - `FloatField`
 - `DecimalField`

- 用法示例

- 日期和时间字段

- `DateTimeField`
 - `DateTimeField` 格式化字符串。
 - `auto_now` 和 `auto_now_add` 模型字段。

- `DateField`

- `DateField` 格式化字符串

- `TimeField`
 - `TimeField` 格式化字符串
- `DurationField`
- 选择字段
 - `ChoiceField`
 - `MultipleChoiceField`
- 文件上传字段
 - 解析和文件上传。
- `FileField`
- `ImageField`
- 复合字段
 - `ListField`
 - `DictField`
 - `JSONField`
- 其他类型的字段
 - `ReadOnlyField`
 - `HiddenField`
 - `ModelField`
 - `SerializerMethodField`
- 自定义字段
 - 看几个栗子
 - 基本的自定义字段
 - 抛出验证错误
 - 使用 `source='*'`

[官方原文链接](#)

Serializer 字段

`Form` 类中的每个字段不仅负责验证数据，还负责“清洗”它 — 将其规范化为一致的格式。

– [Django 文档](#)

序列化字段处理基本数据类型和其他数据类型（比如自定义的类）之间的转换。它们还可以对数据进行验证，以及从其父对象中检索和设置值。

注意： 序列化字段都声明在 `fields.py` 中，但按照惯例，应该使用 `from rest_framework import serializers`，并用 `serializers.<FieldName>` 的方式引用。

核心参数

每个序列化字段类的构造函数都需要一些参数。某些字段类需要附加特定于该字段的参数，但应始终接受以下参数：

`read_only`

只读字段包含于输出 API 中，不应该包含在需要创建或更新操作的输入 API 中。在序列化类输入中错误的包含 ‘read_only’ 会被忽略。

将其设置为 `True` 可确保在序列化表示时使用该字段，但在反序列化期间创建或更新实例时不使用该字段。

默认为 `False`

`write_only`

将其设置为 `True` 以确保在更新或创建实例时可以使用该字段，但在序列化表示时不包括该字段。

默认为 `False`

`required`

如果在反序列化过程中没有该提供字段，通常会出现错误。如果在反序列化过程中不需要此字段，则应该设置为 `false`。

将此设置为 `False` 还允许在序列化实例时从输出中省略对象属性或字典密钥。如果密钥不存在，它将不会包含在输出表示中。

默认为 `True`

`allow_null`

如果把 `None` 传递给序列化字段，通常会引发错误。如果 `None` 应被视为有效值，则将此关键字参数设置为 `True`。

请注意，将此参数设置为 `True` 将意味着序列化输出的缺省值为 `null`，但并不意味着输入反序列化的缺省值。

默认为 `False`

`default`

如果设置，则会给出默认值，在没有提供输入值时，将使用该默认值。如果未设置，则默认行为是不填充该属性。

部分更新操作时不应该使用 `default`。因为有些情况下，只有传入数据中提供的字段才会返回验证值。

可以设置为函数或其他可调用的对象，在这种情况下，每次使用该值时都会对其进行调用。被调用时，它将不会收到任何参数。如果可调用对象具有 `set_context` 方法，那么在每次将字段实例作为参数获取值之前都会调用该方法。这与验证器的工作方式相同。

在序列化实例时，如果对象属性或字典关键字不存在于实例中，将使用缺省值。

请注意，设置默认值意味着该字段不是必需的。同时包括 `default` 和 `required` 的关键字参数都是无效的，会引发错误。

source

将用于填充字段的属性的名称。可以是一个只接受 `self` 参数的方法，如 `URLField(source='get_absolute_url')`，或者使用点符号来遍历属性，如 `EmailField(source='user.email')`。在使用点符号时，如果在属性遍历期间任何对象不存在或为空，则可能需要提供缺省值。

`source = '*'` 具有特殊含义，用于表示整个对象应该传递到该字段。这对创建嵌套表示或对于需要访问完整对象以确定输出表示的字段非常有用。

默认为该字段的名称。

validators

应该应用于传入字段输入的验证函数列表，该列表中的函数应该引发验证错误或仅返回。验证器函数通常应该引发 `serializers.ValidationError`，但 Django 的内置 `ValidationError` 也支持与 Django 代码库或第三方 Django 包中定义的验证器兼容。

error_messages

一个字典，key 是错误代码，value 是对应的错误信息。

label

一个简短的文本字符串，可用作 HTML 表单字段或其他描述性元素中字段的名称。

help_text

一个文本字符串，可用作 HTML 表单字段或其他描述性元素中字段的描述。

initial

应该用于预填充 HTML 表单字段的值。你可能会传递一个可调用对象，就像你对任何常规 Django `Field` 所做的一样：

```

1. import datetime
2. from rest_framework import serializers
3. class ExampleSerializer(serializers.Serializer):
4.     day = serializers.DateField(initial=datetime.date.today)

```

`style`

可用于控制渲染器渲染字段的键值对的字典。

这里有两个例子是 `'input_type'` 和 `'base_template'` :

```

1. # Use <input type="password"> for the input.
2. password = serializers.CharField(
3.     style={'input_type': 'password'}
4. )
5.
6. # Use a radio input instead of a select input.
7. color_channel = serializers.ChoiceField(
8.     choices=['red', 'green', 'blue'],
9.     style={'base_template': 'radio.html'}
10. )

```

Boolean 字段

BooleanField

表示一个 boolean 值。

使用 HTML 编码表单时需要注意，省略一个 boolean 值被视为将字段设置为 `False`，即使它指定了 `default=True` 选项。这是因为 HTML 复选框通过省略该值来表示未选中的状态，所以 REST framework 将省略看作是空的复选框。

请注意，将使用 `required=False` 选项生成默认的 `BooleanField` 实例（因为 Django `models.BooleanField` 始终为 `blank=True`）。如果想要更改此行为，请在序列化类上显式声明 `BooleanField`。

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

签名：`BooleanField()`

NullBooleanField

表示一个布尔值，它也接受 `None` 作为有效值。

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

签名: `NullBooleanField()`

字符串字段

CharField

表示文本。可以使用 `max_length` ， `min_length` 验证（或限定）文本的长短。

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

签名: `CharField(max_length=None, min_length=None, allow_blank=False, trim_whitespace=True)`

- `max_length` - 验证输入所包含的字符数不超过这个数目。
- `min_length` - 验证输入所包含的字符数不少于这个数目。
- `allow_blank` - 如果设置为 `True` ，则空字符串应被视为有效值。如果设置为 `False` ，那么空字符串被认为是无效的并会引发验证错误。默认为 `False` 。
- `trim_whitespace` - 如果设置为 `True` ，则前后空白将被删除。默认为 `True` 。

`allow_null` 选项也可用于字符串字段，尽管它相对于 `allow_blank` 来说不被推荐。同时设置 `allow_blank=True` 和 `allow_null=True` 是有效的，但这样做意味着字符串表示允许有两种不同类型的空值，这可能导致数据不一致和微妙的应用程序错误。

EmailField

表示文本，将文本验证为有效的电子邮件地址。

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

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

RegexField

表示文本，用于验证给定的值是否与某个正则表达式匹配。

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

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

强制的 `regex` 参数可以是一个字符串,也可以是一个编译好的 Python 正则表达式对象。

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

SlugField

一个根据模式 `[a-zA-Z0-9_-]+` 验证输入的 `RegexField` 。

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

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

URLField

一个根据 URL 匹配模式验证输入的 `RegexField` 。完全合格的 URL 格式为 `http://<host>/<path>` 。

对应与 `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,字段将以规范的连字符格式返回一个字符串,例如:

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

签名: `UUIDField(format='hex_verbose')`

- `format`: 确定 uuid 值的表示形式
 - `'hex_verbose'` - 权威的十六进制表示形式, 包含连字符: `"5ce0e9a5-5ffa-654b-cee0-1238041fb31a"`
 - `'hex'` - 紧凑的十六进制表示形式, 不包含连字符: `"5ce0e9a55ffa654bcee01238041fb31a"`
 - `'int'` - 128 位整数表示形式: `"123456789012312313134124512351145145114"`

- `'urn'` - RFC 4122 URN 表示形式: `"urn:5ce0e9a5-5ffa-654b-cee0-1238041fb31a"`
修改 `format` 仅影响表示值。所有格式都被 `to_internal_value` 接受。

FilePathField

一个其选项仅限于文件系统上某个目录中的文件名的字段。

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

签名: `FilePathField(path, match=None, recursive=False, allow_files=True, allow_folders=False, required=None, **kwargs)`

- `path` - `FilePathField` 应该从中选择的目录的绝对文件系统路径。
- `match` - 用来过滤文件名的正则表达式, `string` 类型。
- `recursive` - 指定是否应该包含路径的所有子目录。默认值是 `False`。
- `allow_files` - 是否应该包含指定位置的文件。默认值为 `True`。这个参数或 `allow_folders` 必须是 `True`。(两个属性必须有一个为 `true`)
- `allow_folders` - 是否应该包含指定位置的文件夹。默认值是 `False`。这个参数或 `allow_files` 必须是 `True`。(两个属性必须有一个为 `true`)

IPAddressField

确保输入是有效的 IPv4 或 IPv6 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` 将有效输入限制为指定的协议。接受的值是 `'both'` (默认), `'IPv4'` 或 `'IPv6'`。匹配不区分大小写。
- `unpack_ipv4` 解压 IPv4 映射的地址, 如 `:192.0.2.1`。如果启用此选项, 则该地址将解压到 `192.0.2.1`。默认是禁用的。只能在 `protocol` 设置为 `'both'` 时使用。

数字字段

IntegerField

表示整数。

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` 验证所提供的数字不大于这个值。
- `min_value` 验证所提供的数字不小于这个值。

FloatField

表示浮点。

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

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

- `max_value` 验证所提供的数字不大于这个值。
- `min_value` 验证所提供的数字不小于这个值。

DecimalField

表示十进制, 由 Python 用 `Decimal` 实例表示。

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

签名: `DecimalField(max_digits, decimal_places, coerce_to_string=None, max_value=None, min_value=None)`

- `max_digits` 允许的最大位数。它必须是 `None` 或大于等于 `decimal_places` 的整数。
- `decimal_places` 小数位数。
- `coerce_to_string` 如果应返回字符串值, 则设置为 `True` ; 如果应返回 `Decimal` 对象, 则设置为 `False` 。默认值与 `COERCE_DECIMAL_TO_STRING` settings key 的值相同, 除非被覆盖, 否则该值将为 `True` 。如果序列化对象返回 `Decimal` 对象, 则最终的输出格式将由渲染器决定。请注意, 设置 `localize` 将强制该值为 `True` 。
- `max_value` 验证所提供的数字不大于这个值。
- `min_value` 验证所提供的数字不小于这个值。
- `localize` 设置为 `True` 以启用基于当前语言环境的输入和输出本地化。这也会迫使 `coerce_to_string` 为 `True` 。默认为 `False` 。请注意, 如果你在 settings 文件中设置了 `USE_L10N=True` , 则会启用数据格式化。

- `rounding` 设置量化到配置精度时使用的舍入模式。有效值是 `decimal` 模块舍入模式。默认为 `None`。

用法示例

若要验证数字到999，精确到 2 位小数，应该使用：

```
1. serializers.DecimalField(max_digits=5, decimal_places=2)
```

用10位小数来验证数字不超过10亿：

```
1. serializers.DecimalField(max_digits=19, decimal_places=10)
```

这个字段还接受一个可选参数，`coerce_to_string`。如果设置为 `True`，则表示将以字符串形式输出。如果设置为 `False`，则表示将保留为 `Decimal` 实例，最终表示形式将由渲染器确定。

如果未设置，则默认设置为与 `COERCE_DECIMAL_TO_STRING` setting 相同的值，除非另行设置，否则该值为 `True`。

日期和时间字段

DateTimeField

表示日期和时间。

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

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

- `format` - 表示输出格式的字符串。如果未指定，则默认为与 `DATETIME_FORMAT` settings key 相同的值，除非设置，否则将为 `'iso-8601'`。设置为格式化字符串则表明 `to_representation` 返回值应该被强制为字符串输出。格式化字符串如下所述。将此值设置为 `None` 表示 Python `datetime` 对象应由 `to_representation` 返回。在这种情况下，日期时间编码将由渲染器确定。
- `input_formats` - 表示可用于解析日期的输入格式的字符串列表。如果未指定，则将使用 `DATETIME_INPUT_FORMATS` 设置，该设置默认为 `['iso-8601']`。

`DateTimeField` 格式化字符串。

格式化字符串可以是明确指定的 Python `strftime` 格式，也可以是使用 ISO 8601 风格

`datetime` 的特殊字符串 `iso-8601` 。（例如 `'2013-01-29T12:34:56.000000Z'`）

当一个 `None` 值被用于格式化 `datetime` 对象时，`to_representation` 将返回，最终的输出表示将由渲染器类决定。

`auto_now` 和 `auto_now_add` 模型字段。

使用 `ModelSerializer` 或 `HyperlinkedModelSerializer` 时，请注意，`auto_now=True` 或 `auto_now_add=True` 的模型字段默认情况下将使用 `read_only=True`。

如果想覆盖此行为，则需要在序列化类中明确声明 `DateTimeField`。例如：

```
1. class CommentSerializer(serializers.ModelSerializer):
2.     created = serializers.DateTimeField()
3.
4.     class Meta:
5.         model = Comment
```

DateField

表示日期。

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

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

- `format` - 表示输出格式的字符串。如果未指定，则默认为与 `DATE_FORMAT` settings key 相同的值，除非设置，否则将为 `'iso-8601'`。设置为格式化字符串则表明 `to_representation` 返回值应该被强制为字符串输出。格式化字符串如下所述。将此值设置为 `None` 表示 Python `date` 对象应由 `to_representation` 返回。在这种情况下，日期时间编码将由渲染器确定。
- `input_formats` - 表示可用于解析日期的输入格式的字符串列表。如果未指定，则将使用 `DATE_INPUT_FORMATS` 设置，该设置默认为 `['iso-8601']`。

`DateField` 格式化字符串

格式化字符串可以是明确指定的 Python `strftime` 格式，也可以是使用 ISO 8601 风格 `date` 的特殊字符串 `iso-8601` 。（例如 `'2013-01-29'`）

TimeField

表示时间。

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

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

- `format` - 表示输出格式的字符串。如果未指定，则默认为与 `TIME_FORMAT` settings key 相同的值，除非设置，否则将为 `'iso-8601'`。设置为格式化字符串则表明 `to_representation` 返回值应该被强制为字符串输出。格式化字符串如下所述。将此值设置为 `None` 表示 Python `time` 对象应由 `to_representation` 返回。在这种情况下，日期时间编码将由渲染器确定。
- `input_formats` - 表示可用于解析日期的输入格式的字符串列表。如果未指定，则将使用 `TIME_INPUT_FORMATS` 设置，该设置默认为 `['iso-8601']`。

`TimeField` 格式化字符串

格式化字符串可以是明确指定的 Python `strftime` 格式，也可以是使用 ISO 8601 风格 `time` 的特殊字符串 `iso-8601`。（例如 `'12:34:56.000000'`）

DurationField

表示持续时间。

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

这些字段的 `validated_data` 将包含一个 `datetime.timedelta` 实例。该表示形式是遵循格式 `'[DD] [HH:[MM:]]ss[.uuuuuu]'` 的字符串。

签名: `DurationField()`

选择字段

ChoiceField

可以从一个有限的选择中接受值的字段。

如果相应的模型字段包含 `choices=...` 参数，则由 `ModelSerializer` 自动生成字段。

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`，则空字符串应被视为有效值。如果设置为 `False`，

那么空字符串被认为是无效的并会引发验证错误。默认是 `False`。

- `html_cutoff` - 如果设置，这将是 HTML 选择下拉菜单中显示的选项的最大数量。可用于确保自动生成具有非常大可以选择的 `ChoiceField`，而不会阻止模板的渲染。默认是 `None`。
- `html_cutoff_text` - 指定一个文本指示器，在截断列表时显示，比如在 HTML 选择下拉菜单中已经截断了最大数量的项目。默认就会显示 `"More than {count} items..."`

`Allow_blank` 和 `allow_null` 都是 `ChoiceField` 上的有效选项，但强烈建议只使用一个而不是两个都用。对于文本选择，`allow_blank` 应该是首选，`allow_null` 应该是数字或其他非文本选项的首选。

MultipleChoiceField

可以接受一组零、一个或多个值的字段，从有限的一组选择中选择。采取一个必填的参数。

`to_internal_value` 返回一个包含选定值的 `set`。

签名: `MultipleChoiceField(choices)`

- `choices` - 有效值列表，或 `(key, display_name)` 元组列表。
- `allow_blank` - 如果设置为 `True`，则空字符串应被视为有效值。如果设置为 `False`，那么空字符串被认为是无效的并会引发验证错误。默认是 `False`。
- `html_cutoff` - 如果设置，这将是 HTML 选择下拉菜单中显示的选项的最大数量。可用于确保自动生成具有非常大可以选择的 `ChoiceField`，而不会阻止模板的渲染。默认是 `None`。
- `html_cutoff_text` - 指定一个文本指示器，在截断列表时显示，比如在 HTML 选择下拉菜单中已经截断了最大数量的项目。默认就会显示 `"More than {count} items..."`

`Allow_blank` 和 `allow_null` 都是 `ChoiceField` 上的有效选项，但强烈建议只使用一个而不是两个都用。对于文本选择，`allow_blank` 应该是首选，`allow_null` 应该是数字或其他非文本选项的首选。

文件上传字段

解析和文件上传。

`FileField` 和 `ImageField` 类只适用于 `MultiPartParser` 或 `FileUploadParser`。大多数解析器，例如，e.g. JSON 不支持文件上传。Django 的常规 `FILE_UPLOAD_HANDLERS` 用于处理上传的文件。

FileField

表示文件。执行 Django 的标准 `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 字符串值将用于输出表示。如果设置为 `False`，则文件名字符串值将用于输出表示。默认为 `UPLOADED_FILES_USE_URL` settings key 的值，除非另有设置，否则为 `True`。

ImageField

表示图片。验证上传的文件内容是否匹配已知的图片格式。

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

签名: `ImageField(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 字符串值将用于输出表示。如果设置为 `False`，则文件名字符串值将用于输出表示。默认为 `UPLOADED_FILES_USE_URL` settings key 的值，除非另有设置，否则为 `True`。

需要 `Pillow` 库或 `PIL` 库。 建议使用 `Pillow` 库。 因为 `PIL` 已经不再维护。

复合字段

ListField

验证对象列表的字段类。

签名: `ListField(child=<A_FIELD_INSTANCE>, min_length=None, max_length=None)`

- `child` - 应该用于验证列表中的对象的字段实例。如果未提供此参数，则列表中的对象将不会被验证。
- `min_length` - 验证列表中包含的元素数量不少于这个数。
- `max_length` - 验证列表中包含的元素数量不超过这个数。

例如，要验证整数列表：

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

`ListField` 类还支持一种声明式风格，允许编写可重用的列表字段类。

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

我们现在可以在我们的应用程序中重新使用我们自定义的 `StringListField` 类，而无需为其提供 `child` 参数。

DictField

验证对象字典的字段类。 `DictField` 中的键总是被假定为字符串值。

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

- `child` - 应该用于验证字典中的值的字段实例。如果未提供此参数，则映射中的值将不会被验证。

例如，要创建一个验证字符串到字符串映射的字段，可以这样写：

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

你也可以像使用 `ListField` 一样使用声明式风格。例如：

```
1. class DocumentField(DictField):
2.     child = CharField()
```

JSONField

验证传入的数据结构由有效 JSON 基元组成的字段类。在其二进制模式下，它将表示并验证 JSON 编码的二进制字符串。

签名： `JSONField(binary)`

- `binary` - 如果设置为 `True`，那么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。默认是 `False`。

其他类型的字段

ReadOnlyField

只是简单地返回字段的值而不进行修改的字段类。

当包含与属性相关的字段名而不是模型字段时，此字段默认与 `ModelSerializer` 一起使用。

签名： `ReadOnlyField()`

例如，如果 `has_expired` 是 `Account` 模型中的一个属性，则以下序列化程序会自动将其生成 `ReadOnlyField`：

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

HiddenField

不根据用户输入获取值的字段类，而是从默认值或可调用值中获取值。

签名： `HiddenField()`

例如，要包含始终提供当前时间的字段作为序列化类验证数据的一部分，则可以使用以下内容：

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

如果需要根据某些预先提供的字段值运行某些验证，则通常只需要 `HiddenField` 类，而不是将所有这些字段公开给最终用户。

ModelField

可以绑定到任意模型字段的通用字段。 `ModelField` 类将序列化/反序列化的任务委托给其关联的模型字段。该字段可用于为自定义模型字段创建序列化字段，而无需创建新的自定义序列化字段。

`ModelSerializer` 使用此字段来对应自定义模型字段类。

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

`ModelField` 类通常用于内部使用，但如果需要，可以由 API 使用。为了正确实例化 `ModelField`，必须传递一个附加到实例化模型的字段。例

如：`ModelField(model_field=MyModel()._meta.get_field('custom_field'))`

SerializerMethodField

这是一个只读字段。它通过调用它所连接的序列化类的方法来获得它的值。它可用于将任何类型的数据添加到对象的序列化表示中。

签名：`SerializerMethodField(method_name=None)`

- `method_name` - 要调用序列化对象的方法的名称。如果不包含，则默认为 `get_<field_name>`。

由 `method_name` 参数引用的序列化方法应该接受一个参数（除了 `self`），这是要序列化的对象。它应该返回你想要包含在对象的序列化表示中的任何内容。例如：

```
1. from django.contrib.auth.models import User
2. from django.utils.timezone import now
3. from rest_framework import serializers
4.
5. class UserSerializer(serializers.ModelSerializer):
6.     days_since_joined = serializers.SerializerMethodField()
7.
8.     class Meta:
9.         model = User
10.
11.     def get_days_since_joined(self, obj):
12.         return (now() - obj.date_joined).days
```

自定义字段

如果你想创建自定义字段，则需要对 `Field` 进行子类化，然后重写 `.to_representation()` 和 `.to_internal_value()` 方法中的一个或两个。这两个方法用于在初始数据类型和基本序列化数据类型之间进行转换。基本数据类型通常是 `number`，`string`，`boolean`，`date` / `time` / `datetime` 或 `None`。它们也可以是任何仅包含其他基本对象的列表或字典。其他类型可能会支持，但具体取决于你使用的渲染器。

调用 `.to_representation()` 方法将初始数据类型转换为基本的可序列化数据类型。

调用 `to_internal_value()` 方法将基本数据类型恢复为其内部 python 表示形式。如果数据无

效，此方法应该引发 `serializers.ValidationError` 异常。

请注意，2.x 版本中存在的 `WritableField` 类不再存在。 应此，如果字段需要支持数据输入，则应该继承 `Field` 并覆盖 `to_internal_value()`。

看几个栗子

基本的自定义字段

我们来看一个序列化代表 RGB 颜色值的类的例子：

```
1. class Color(object):
2.     """
3.     A color represented in the RGB colorspace.
4.     """
5.     def __init__(self, red, green, blue):
6.         assert(red >= 0 and green >= 0 and blue >= 0)
7.         assert(red < 256 and green < 256 and blue < 256)
8.         self.red, self.green, self.blue = red, green, blue
9.
10. class ColorField(serializers.Field):
11.     """
12.     Color objects are serialized into 'rgb(#, #, #)' notation.
13.     """
14.     def to_representation(self, obj):
15.         return "rgb(%d, %d, %d)" % (obj.red, obj.green, obj.blue)
16.
17.     def to_internal_value(self, data):
18.         data = data.strip('rgb(').rstrip(')')
19.         red, green, blue = [int(col) for col in data.split(',')]
20.         return Color(red, green, blue)
```

默认情况下，字段值被视为映射到对象的属性。如果需要自定义字段值的访问方式，则需要覆盖

`.get_attribute()` 和/或 `.get_value()`。

让我们创建一个可以用来表示被序列化对象的类名的字段：

```
1. class ClassNameField(serializers.Field):
2.     def get_attribute(self, obj):
3.         # We pass the object instance onto `to_representation`,
4.         # not just the field attribute.
5.         return obj
```



```

6.
7.     def to_representation(self, obj):
8.         """
9.         Serialize the object's class name.
10.        """
11.        return obj.__class__.__name__

```

抛出验证错误

我们上面的 `ColorField` 类目前不执行任何数据验证。为了表示无效数据，我们应该引发一个 `serializers.ValidationError`，如下所示：

```

1. def to_internal_value(self, data):
2.     if not isinstance(data, six.text_type):
3.         msg = 'Incorrect type. Expected a string, but got %s'
4.         raise ValidationError(msg % type(data).__name__)
5.
6.     if not re.match(r'^rgb\([0-9]+,[0-9]+,[0-9]+\)$', data):
7.         raise ValidationError('Incorrect format. Expected `rgb(##,##,##)`.')
8.
9.     data = data.strip('rgb(').rstrip(')')
10.    red, green, blue = [int(col) for col in data.split(',')]
11.
12.    if any([col > 255 or col < 0 for col in (red, green, blue)]):
13.        raise ValidationError('Value out of range. Must be between 0 and 255.')
14.
15.    return Color(red, green, blue)

```

`.fail()` 方法是引发 `ValidationError` 的快捷方式，它从 `error_messages` 字典中接收消息字符串。例如：

```

1. default_error_messages = {
2.     'incorrect_type': 'Incorrect type. Expected a string, but got
3.     {input_type}',
4.     'incorrect_format': 'Incorrect format. Expected `rgb(##,##,##)`.',
5.     'out_of_range': 'Value out of range. Must be between 0 and 255.'
6. }
7.
8. def to_internal_value(self, data):
9.     if not isinstance(data, six.text_type):
10.         self.fail('incorrect_type', input_type=type(data).__name__)

```

```

10.
11.     if not re.match(r'^rgb\[0-9]+\,[0-9]+\,[0-9]+\)$', data):
12.         self.fail('incorrect_format')
13.
14.     data = data.strip('rgb(').rstrip(')')
15.     red, green, blue = [int(col) for col in data.split(',')]
16.
17.     if any([col > 255 or col < 0 for col in (red, green, blue)]):
18.         self.fail('out_of_range')
19.
20.     return Color(red, green, blue)

```

这种风格让你的错误信息更清晰，并且与代码分离，应该是首选。

使用

`source='*'`

这里我们将举一个具有 `x_coordinate` 和 `y_coordinate` 属性的平面 `DataPoint` 模型的示例。

```

1. class DataPoint(models.Model):
2.     label = models.CharField(max_length=50)
3.     x_coordinate = models.SmallIntegerField()
4.     y_coordinate = models.SmallIntegerField()

```

使用自定义字段和 `source = '*'`，我们可以提供坐标对的嵌套表示形式：

```

1. class CoordinateField(serializers.Field):
2.
3.     def to_representation(self, obj):
4.         ret = {
5.             "x": obj.x_coordinate,
6.             "y": obj.y_coordinate
7.         }
8.         return ret
9.
10.    def to_internal_value(self, data):
11.        ret = {
12.            "x_coordinate": data["x"],
13.            "y_coordinate": data["y"],
14.        }
15.        return ret
16.

```

```

17.
18. class DataPointSerializer(serializers.ModelSerializer):
19.     coordinates = CoordinateField(source='*')
20.
21.     class Meta:
22.         model = DataPoint
23.         fields = ['label', 'coordinates']

```

请注意，此示例不处理验证。使用 `source = '*'` 的嵌套序列化类可以更好地处理坐标嵌套，并且带有两个 `IntegerField` 实例，每个实例都有自己的 `source` 指向相关字段。

然而，这个例子的关键点是：

- `to_representation` 传递整个 `DataPoint` 对象，并且会映射到所需的输出。

```

1. >>> instance = DataPoint(label='Example', x_coordinate=1, y_coordinate=2)
2. >>> out_serializer = DataPointSerializer(instance)
3. >>> out_serializer.data
4. ReturnDict([('label', 'Example'), ('coordinates', {'x': 1, 'y': 2})])

```

- 除非我们的字段是只读的，否则 `to_internal_value` 必须映射回适合更新目标对象的字典。使用 `source = '*'`，`to_internal_value` 的返回将更新根验证的数据字典，而不是单个键。

```

1. >>> data = {
2.     ...     "label": "Second Example",
3.     ...     "coordinates": {
4.     ...         "x": 3,
5.     ...         "y": 4,
6.     ...     }
7.     ... }
8. >>> in_serializer = DataPointSerializer(data=data)
9. >>> in_serializer.is_valid()
10. True
11. >>> in_serializer.validated_data
12. OrderedDict([('label', 'Second Example'),
13.              ('y_coordinate', 4),
14.              ('x_coordinate', 3)])

```

为了完整性，我们再次做同样的事情，但是使用上面建议的嵌套序列化方法：

```

1. class NestedCoordinateSerializer(serializers.Serializer):

```

```

2.     x = serializers.IntegerField(source='x_coordinate')
3.     y = serializers.IntegerField(source='y_coordinate')
4.
5.
6. class DataPointSerializer(serializers.ModelSerializer):
7.     coordinates = NestedCoordinateSerializer(source='*')
8.
9.     class Meta:
10.         model = DataPoint
11.         fields = ['label', 'coordinates']

```

这里，我们在 `IntegerField` 声明中处理目标和源属性对（`x` 和 `x_coordinate`，`y` 和 `y_coordinate`）之间的映射。这是使用了 `source = '*'` 的 `NestedCoordinateSerializer`。

新的 `DataPointSerializer` 展现出与自定义字段方法相同的行为。

序列化：

```

1. >>> out_serializer = DataPointSerializer(instance)
2. >>> out_serializer.data
3. ReturnDict([('label', 'testing'),
4.              ('coordinates', OrderedDict([('x', 1), ('y', 2)]))])

```

反序列化：

```

1. >>> in_serializer = DataPointSerializer(data=data)
2. >>> in_serializer.is_valid()
3. True
4. >>> in_serializer.validated_data
5. OrderedDict([('label', 'still testing'),
6.               ('x_coordinate', 3),
7.               ('y_coordinate', 4)])

```

虽然没有编写验证，但是可以使用内置的验证方式：

```

1. >>> invalid_data = {
2.     ...     "label": "still testing",
3.     ...     "coordinates": {
4.     ...         "x": 'a',
5.     ...         "y": 'b',
6.     ...     }

```

```
7. ... }
8. >>> invalid_serializer = DataPointSerializer(data=invalid_data)
9. >>> invalid_serializer.is_valid()
10. False
11. >>> invalid_serializer.errors
12. ReturnDict([('coordinates',
13.               {'x': ['A valid integer is required.'],
14.               'y': ['A valid integer is required.']}))])
```

出于这个原因，可以首先尝试嵌套序列化类方法。当嵌套序列化类变得不可行或过于复杂时，可以使用自定义字段方法。

序列化·关系

- [Serializer 关系](#)
 - [检查关系。](#)
- [API 参考](#)
 - [StringRelatedField](#)
 - [PrimaryKeyRelatedField](#)
 - [HyperlinkedRelatedField](#)
 - [SlugRelatedField](#)
 - [HyperlinkedIdentityField](#)
- [嵌套关系](#)
 - [举个栗子](#)
 - [可写嵌套序列化类](#)
- [自定义关系字段](#)
 - [举个栗子](#)
- [自定义超链接字段](#)
 - [举个栗子](#)
- [进一步说明](#)
 - `queryset` 参数
 - [自定义 HTML 显示](#)
 - [Select field cutoffs](#)
 - [反向关系](#)
 - [通用关系](#)
 - [具有 Through 模型的 ManyToManyFields](#)

[官方原文链接](#)

Serializer 关系

关系字段用于表示模型关系。它们可以应用于 `ForeignKey`，`ManyToManyField` 和 `OneToOneField` 关系，反向关系以及 `GenericForeignKey` 等自定义关系。

注意：关系字段在 `relations.py` 中声明，但按照惯例，你应该从 `serializers` 模块导入它们，使用 `from rest_framework import serializers` 引入，并像 `serializers.<FieldName>` 这样引用字段。

检查关系。

在使用 `ModelSerializer` 类时，将自动为你生成序列化字段和关系。检查这些自动生成的字段可以学习如何定制关系风格。

为此，使用 `python manage.py shell` 打开 Django shell，然后导入序列化类，实例化它并打印对象表示形式...

```
1. >>> from myapp.serializers import AccountSerializer
2. >>> serializer = AccountSerializer()
3. >>> print repr(serializer) # Or `print(repr(serializer))` in Python 3.x.
4. AccountSerializer():
5.     id = IntegerField(label='ID', read_only=True)
6.     name = CharField(allow_blank=True, max_length=100, required=False)
7.     owner = PrimaryKeyRelatedField(queryset=User.objects.all())
```

API 参考

为了解释各种类型的关系字段，我们将为我们的示例使用几个简单的模型。我们的模型将使用音乐专辑，以及每张专辑中列出的曲目。

```
1. class Album(models.Model):
2.     album_name = models.CharField(max_length=100)
3.     artist = models.CharField(max_length=100)
4.
5. class Track(models.Model):
6.     album = models.ForeignKey(Album, related_name='tracks',
7. on_delete=models.CASCADE)
8.     order = models.IntegerField()
9.     title = models.CharField(max_length=100)
10.    duration = models.IntegerField()
11.
12.    class Meta:
13.        unique_together = ('album', 'order')
14.        ordering = ['order']
15.
16.    def __unicode__(self):
17.        return '%d: %s' % (self.order, self.title)
```

StringRelatedField

`StringRelatedField` 用于使用 `__unicode__` 方法表示关系。

例如，下面的序列化类。

```

1. class AlbumSerializer(serializers.ModelSerializer):
2.     tracks = serializers.StringRelatedField(many=True)
3.
4.     class Meta:
5.         model = Album
6.         fields = ('album_name', 'artist', 'tracks')

```

将序列化为以下形式。

```

1. {
2.     'album_name': 'Things We Lost In The Fire',
3.     'artist': 'Low',
4.     'tracks': [
5.         '1: Sunflower',
6.         '2: Whitetail',
7.         '3: Dinosaur Act',
8.         ...
9.     ]
10. }

```

该字段是只读的。

参数：

- `many` - 如果是一对多的关系，就将此参数设置为 `True`。

PrimaryKeyRelatedField

`PrimaryKeyRelatedField` 用于使用其主键表示关系。

例如，以下序列化类：

```

1. class AlbumSerializer(serializers.ModelSerializer):
2.     tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True)
3.
4.     class Meta:
5.         model = Album
6.         fields = ('album_name', 'artist', 'tracks')

```

将序列化为这样的表示：


```

1.  {
2.      'album_name': 'Undun',
3.      'artist': 'The Roots',
4.      'tracks': [
5.          89,
6.          90,
7.          91,
8.          ...
9.      ]
10. }

```

默认情况下，该字段是可读写的，但您可以使用 `read_only` 标志更改此行为。

参数：

- `queryset` - 验证字段输入时用于模型实例查询的查询集。必须显式地设置查询集，或设置 `read_only=True`。
- `many` - 如果应用于一对多关系，则应将此参数设置为 `True`。
- `allow_null` - 如果设置为 `True`，那么该字段将接受 `None` 值或可为空的关系的空字符串。默认为 `False`。
- `pk_field` - 设置为一个字段来控制主键值的序列化/反序列化。例如，`pk_field=UUIDField(format='hex')` 会将 UUID 主键序列化为其紧凑的十六进制表示形式。

HyperlinkedRelatedField

`HyperlinkedRelatedField` 用于使用超链接来表示关系。

例如，以下序列化类：

```

1. class AlbumSerializer(serializers.ModelSerializer):
2.     tracks = serializers.HyperlinkedRelatedField(
3.         many=True,
4.         read_only=True,
5.         view_name='track-detail'
6.     )
7.
8.     class Meta:
9.         model = Album
10.        fields = ('album_name', 'artist', 'tracks')

```

将序列化为这样的表示：

```
1. {
2.     'album_name': 'Graceland',
3.     'artist': 'Paul Simon',
4.     'tracks': [
5.         'http://www.example.com/api/tracks/45/',
6.         'http://www.example.com/api/tracks/46/',
7.         'http://www.example.com/api/tracks/47/',
8.         ...
9.     ]
10. }
```

默认情况下，该字段是可读写的，但您可以使用 `read_only` 标志更改此行为。

注意：该字段是为映射到接受单个 URL 关键字参数的 URL 的对象而设计的，如使用

`lookup_field` 和 `lookup_url_kwarg` 参数设置的对象。

这适用于包含单个主键或 slug 参数作为 URL 一部分的 URL。

如果需要更复杂的超链接表示，你需要自定义该字段，稍后会详解。

参数：

- `view_name` - 用作关系目标的视图名称。如果你使用的是标准路由器类，则这将是一个格式为 `<modelname>-detail` 的字符串。必填。
- `queryset` - 验证字段输入时用于模型实例查询的查询集。必须显式地设置查询集，或设置 `read_only=True`。
- `many` - 如果应用于一对多关系，则应将此参数设置为 `True`。
- `allow_null` - 如果设置为 `True`，那么该字段将接受 `None` 值或可为空的关系的空字符串。默认为 `False`。
- `lookup_field` - 用于查找的目标字段。对应于引用视图上的 URL 关键字参数。默认是 `'pk'`。
- `lookup_url_kwarg` - 与查找字段对应的 URL conf 中定义的关键字参数的名称。默认使用与 `lookup_field` 相同的值。
- `format` - 如果使用格式后缀，则超链接字段将使用与目标相同的格式后缀，除非使用 `format` 参数进行覆盖。

SlugRelatedField

`SlugRelatedField` 用于使用目标上的字段来表示关系。

例如，以下序列化类：

```
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')
```

将序列化为这样的表示：

```
1. {
2.     'album_name': 'Dear John',
3.     'artist': 'Loney Dear',
4.     'tracks': [
5.         'Airport Surroundings',
6.         'Everything Turns to You',
7.         'I Was Only Going Out',
8.         ...
9.     ]
10. }
```

默认情况下，该字段是可读写的，但您可以使用 `read_only` 标志更改此行为。

将 `SlugRelatedField` 用作读写字段时，通常需要确保 slug 字段与 `unique=True` 的模型字段相对应。

参数：

- `slug_field` - 用来表示目标的字段。这应该是唯一标识给定实例的字段。例如，`username`。必填
- `queryset` - 验证字段输入时用于模型实例查询的查询集。必须显式地设置查询集，或设置 `read_only=True`。
- `many` - 如果应用于一对多关系，则应将此参数设置为 `True`。
- `allow_null` - 如果设置为 `True`，那么该字段将接受 `None` 值或可为空的关系的空字符串。默认为 `False`。

HyperlinkedIdentityField

此字段可以作为身份关系应用，例如 `HyperlinkedModelSerializer` 上的 `'url'` 字段。它也可以用于对象的属性。例如，以下序列化类：

```
1. class AlbumSerializer(serializers.HyperlinkedModelSerializer):
2.     track_listing = serializers.HyperlinkedIdentityField(view_name='track-
   list')
3.
4.     class Meta:
5.         model = Album
6.         fields = ('album_name', 'artist', 'track_listing')
```

将序列化为这样的表示：

```
1. {
2.     'album_name': 'The Eraser',
3.     'artist': 'Thom Yorke',
4.     'track_listing': 'http://www.example.com/api/track_list/12/',
5. }
```

该字段始终为只读。

参数：

- `view_name` - 用作关系目标的视图名称。如果你使用的是标准路由器类，则这将是一个格式为 `<modelname>-detail` 的字符串。必填。
- `lookup_field` - 用于查找的目标字段。对应于引用视图上的 URL 关键字参数。默认是 `'pk'`。
- `lookup_url_kwarg` - 与查找字段对应的 URL conf 中定义的关键字参数的名称。默认使用与 `lookup_field` 相同的值。
- `format` - 如果使用格式后缀，则超链接字段将使用与目标相同的格式后缀，除非使用 `format` 参数进行覆盖。

嵌套关系

嵌套关系可以通过使用序列化类作为字段来表达。

如果该字段用于表示一对多关系，则应将 `many=True` 标志添加到序列化字段。

举个栗子

例如，以下序列化类：

```

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

将序列化为这样的嵌套表示：

```

1. >>> album = Album.objects.create(album_name="The Grey Album", artist='Danger
    Mouse')
2. >>> Track.objects.create(album=album, order=1, title='Public Service
    Announcement', duration=245)
3. <Track: Track object>
4. >>> Track.objects.create(album=album, order=2, title='What More Can I Say',
    duration=264)
5. <Track: Track object>
6. >>> Track.objects.create(album=album, order=3, title='Encore', duration=159)
7. <Track: Track object>
8. >>> serializer = AlbumSerializer(instance=album)
9. >>> serializer.data
10. {
11.     'album_name': 'The Grey Album',
12.     'artist': 'Danger Mouse',
13.     'tracks': [
14.         {'order': 1, 'title': 'Public Service Announcement', 'duration': 245},
15.         {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
16.         {'order': 3, 'title': 'Encore', 'duration': 159},
17.         ...
18.     ],
19. }
```

可写嵌套序列化类

默认情况下，嵌套序列化类是只读的。如果要支持对嵌套序列化字段的写操作，则需要创建

`create()` 和/或 `update()` 方法，以明确指定应如何保存子关系。

```
1. class TrackSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         model = Track
4.         fields = ('order', 'title', 'duration')
5.
6. class AlbumSerializer(serializers.ModelSerializer):
7.     tracks = TrackSerializer(many=True)
8.
9.     class Meta:
10.         model = Album
11.         fields = ('album_name', 'artist', 'tracks')
12.
13.     def create(self, validated_data):
14.         tracks_data = validated_data.pop('tracks')
15.         album = Album.objects.create(**validated_data)
16.         for track_data in tracks_data:
17.             Track.objects.create(album=album, **track_data)
18.         return album
19.
20. >>> data = {
21.     'album_name': 'The Grey Album',
22.     'artist': 'Danger Mouse',
23.     'tracks': [
24.         {'order': 1, 'title': 'Public Service Announcement', 'duration': 245},
25.         {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
26.         {'order': 3, 'title': 'Encore', 'duration': 159},
27.     ],
28. }
29. >>> serializer = AlbumSerializer(data=data)
30. >>> serializer.is_valid()
31. True
32. >>> serializer.save()
33. <Album: Album object>
```

自定义关系字段

在极少数情况下，现有的关系类型都不符合您需要的表示形式，你可以实现一个完全自定义的关系字段，该字段准确描述应该如何从模型实例生成输出表示。

要实现自定义关系字段，您应该重写 `RelatedField`，并实现 `.to_representation(self, value)` 方法。此方法将字段的目标作为 `value` 参数，并返回应用于序列化目标的表示。`value` 参数通常是一个模型实例。

如果来实现读写关系字段，则还必须实现 `.to_internal_value(self, data)` 方法。

要提供基于 `context` 的动态查询集，还可以覆盖 `.get_queryset(self)`，而不是在类上指定 `.queryset` 或初始化该字段。

举个栗子

例如，我们可以定义一个关系字段，使用它的顺序，标题和持续时间将音轨序列化为自定义字符串表示。

```
1. import time
2.
3. class TrackListingField(serializers.RelatedField):
4.     def to_representation(self, value):
5.         duration = time.strftime('%M:%S', time.gmtime(value.duration))
6.         return 'Track %d: %s (%s)' % (value.order, value.name, duration)
7.
8. class AlbumSerializer(serializers.ModelSerializer):
9.     tracks = TrackListingField(many=True)
10.
11.     class Meta:
12.         model = Album
13.         fields = ('album_name', 'artist', 'tracks')
```

将序列化为这样的表示：

```
1. {
2.     'album_name': 'Sometimes I Wish We Were an Eagle',
3.     'artist': 'Bill Callahan',
4.     'tracks': [
5.         'Track 1: Jim Cain (04:39)',
6.         'Track 2: Eid Ma Clack Shaw (04:19)',
7.         'Track 3: The Wind and the Dove (04:34)',
8.         ...
9.     ]
}
```

```
10. }
```

自定义超链接字段

在某些情况下，您可能需要自定义超链接字段的行为，以表示需要多个查询字段的 URL。

您可以通过继承 `HyperlinkedRelatedField` 来实现此目的。有两个可以被覆盖的方法：

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

`get_url` 方法用于将对象实例映射到其 URL 表示。

如果 `view_name` 和 `lookup_field` 属性未配置为正确匹配 URL conf，可能会引发 `NoReverseMatch`。

`get_object(self, queryset, view_name, view_args, view_kwargs)`

如果您想支持可写的超链接字段，那么您还需要重写 `get_object`，以便将传入的 URL 映射回它们表示的对象。对于只读超链接字段，不需要重写此方法。

此方法的返回值应该是与匹配的 URL conf 参数对应的对象。

可能会引发 `ObjectDoesNotExist` 异常。

举个栗子

假设我们有一个带有两个关键字参数的 `customer` 对象的 URL，如下所示：

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

这没办法用仅接受单个查找字段的默认实现来表示。

在这种情况下，我们需要继承 `HyperlinkedRelatedField` 并重写其中的方法来获得我们想要的行为：

```
1. from rest_framework import serializers
2. from rest_framework.reverse import reverse
3.
4. class CustomerHyperlink(serializers.HyperlinkedRelatedField):
5.     # We define these as class attributes, so we don't need to pass them as
   arguments.
6.     view_name = 'customer-detail'
```



```

7.     queryset = Customer.objects.all()
8.
9.     def get_url(self, obj, view_name, request, format):
10.         url_kwargs = {
11.             'organization_slug': obj.organization.slug,
12.             'customer_pk': obj.pk
13.         }
14.         return reverse(view_name, kwargs=url_kwargs, request=request,
format=format)
15.
16.     def get_object(self, view_name, view_args, view_kwargs):
17.         lookup_kwargs = {
18.             'organization__slug': view_kwargs['organization_slug'],
19.             'pk': view_kwargs['customer_pk']
20.         }
21.         return self.get_queryset().get(**lookup_kwargs)

```

请注意，如果您想将此风格与通用视图一起使用，那么您还需要覆盖视图上的 `.get_object` 以获得正确的查找行为。

一般来说，我们建议尽可能在 API 表示方式下使用平面风格，但嵌套 URL 风格在适度使用时也是合理的。

进一步说明

queryset 参数

`queryset` 参数只对可写关系字段是必需的，在这种情况下，它用于执行模型实例查找，该查找从基本用户输入映射到模型实例。

在 2.x 版本中，如果正在使用 `ModelSerializer` 类，则序列化类有时会自动确定 `queryset` 参数。

此行为现在替换为始终为可写关系字段使用显式 `queryset` 参数。

这样做可以减少 `ModelSerializer` 提供的隐藏“魔术”数量（指 `ModelSerializer` 在内部帮我们完成的工作），使字段的行为更加清晰，并确保在使用 `ModelSerializer` 快捷方式（高度封装过，使用简单）或使用完全显式的 `Serializer` 类之间转换是微不足道的。

自定义 HTML 显示

模型内置的 `__str__` 方法用来生成用于填充 `choices` 属性的对象的字符串表示形式。这些 `choices` 用于在可浏览的 API 中填充选择的 HTML input。

要为这些 input 提供自定义表示，请重写 `RelatedField` 子类的 `display_value()` 方法。这个方法将接收一个模型对象，并且应该返回一个适合表示它的字符串。例如：

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

Select field cutoffs

在渲染可浏览的 API 关系字段时，默认只显示最多 1000 个可选 item。如果存在更多项目，则会显示 “More than 1000 items...” 的 disabled 选项。

此行为旨在防止由于显示大量关系而导致模板无法在可接受的时间范围内完成渲染。

有两个关键字参数可用于控制此行为：

- `html_cutoff` - 设置 HTML select 下拉菜单中显示的选项的最大数量。设置为 `None` 可以禁用任何限制。默认为 `1000`。
- `html_cutoff_text` - 设置一个文本字符串，在 HTML select 下拉菜单超出最大显示数量时显示。默认是 `"More than {count} items..."`

你还可以在 settings 中用 `HTML_SELECT_CUTOFF` 和 `HTML_SELECT_CUTOFF_TEXT` 来全局控制这些设置。

在强制执行 cutoff 的情况下，您可能希望改为在 HTML 表单中使用简单的 input 字段。你可以使用 `style` 关键字参数来做到这一点。例如：

```
1. assigned_to = serializers.SlugRelatedField(
2.     queryset=User.objects.all(),
3.     slug_field='username',
4.     style={'base_template': 'input.html'})
5. )
```

反向关系

请注意，反向关系不会自动包含在 `ModelSerializer` 和 `HyperlinkedModelSerializer` 类中。要包含反向关系，您必须明确将其添加到字段列表中。例如：

```

1. class AlbumSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         fields = ('tracks', ...)

```

通常需要确保已经在关系上设置了适当的 `related_name` 参数，可以将其用作字段名称。例如：

```

1. class Track(models.Model):
2.     album = models.ForeignKey(Album, related_name='tracks',
3.         on_delete=models.CASCADE)
4.     ...

```

如果你还没有为反向关系设置相关名称，则需要在 `fields` 参数中使用自动生成的相关名称。例如：

```

1. class AlbumSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         fields = ('track_set', ...)

```

通用关系

如果要序列化通用外键，则需要自定义字段，以明确确定如何序列化关系。

例如，给定一个以下模型的标签，该标签与其他任意模型具有通用关系：

```

1. class TaggedItem(models.Model):
2.     """
3.     Tags arbitrary model instances using a generic relation.
4.
5.     See: https://docs.djangoproject.com/en/stable/ref/contrib/contenttypes/
6.     """
7.     tag_name = models.SlugField()
8.     content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
9.     object_id = models.PositiveIntegerField()
10.    tagged_object = GenericForeignKey('content_type', 'object_id')
11.
12.    def __unicode__(self):
13.        return self.tag_name

```

以下两种模式可以用相关的标签：

```

1. class Bookmark(models.Model):
2.     """
3.     A bookmark consists of a URL, and 0 or more descriptive tags.
4.     """
5.     url = models.URLField()
6.     tags = GenericRelation(TaggedItem)
7.
8.
9. class Note(models.Model):
10.    """
11.    A note consists of some text, and 0 or more descriptive tags.
12.    """
13.    text = models.CharField(max_length=1000)
14.    tags = GenericRelation(TaggedItem)

```

我们可以定义一个可用于序列化标签实例的自定义字段，并使用每个实例的类型来确定它应该如何序列化。

```

1. class TaggedObjectRelatedField(serializers.RelatedField):
2.     """
3.     A custom field to use for the `tagged_object` generic relationship.
4.     """
5.
6.     def to_representation(self, value):
7.         """
8.         Serialize tagged objects to a simple textual representation.
9.         """
10.        if isinstance(value, Bookmark):
11.            return 'Bookmark: ' + value.url
12.        elif isinstance(value, Note):
13.            return 'Note: ' + value.text
14.        raise Exception('Unexpected type of tagged object')

```

如果你需要的关系具有嵌套表示，则可以在 `.to_representation()` 方法中使用所需的序列化类：

```

1.     def to_representation(self, value):
2.         """
3.         Serialize bookmark instances using a bookmark serializer,
4.         and note instances using a note serializer.
5.         """

```

```
6.         if isinstance(value, Bookmark):
7.             serializer = BookmarkSerializer(value)
8.         elif isinstance(value, Note):
9.             serializer = NoteSerializer(value)
10.        else:
11.            raise Exception('Unexpected type of tagged object')
12.
13.        return serializer.data
```

请注意，使用 `GenericRelation` 字段表示的反向通用键可以使用常规关系字段类型进行序列化，因为关系中目标的类型总是已知的。

具有 Through 模型的 ManyToManyFields

默认情况下，将指定带有 `through` 模型的 `ManyToManyField` 的关系字段设置为只读。

如果你要明确指定一个指向具有 `through` 模型的 `ManyToManyField` 的关系字段，请确保将 `read_only` 设置为 `True` 。

验证器

- 验证器
 - REST framework 中的验证
 - 举个栗子
- UniqueValidator
- UniqueTogetherValidator
- UniqueForDateValidator
- UniqueForMonthValidator
- UniqueForYearValidator
 - 与可写日期字段一起使用。
 - 与只读日期字段一起使用。
 - 与隐藏的日期字段一起使用。
- Advanced field defaults
 - CurrentUserDefault
 - CreateOnlyDefault
- 验证器的限制
 - 可选字段
 - 更新嵌套序列化类
 - 调试复杂的案例
- 编写自定义验证器
 - 基于函数
 - 字段级验证
- 基于类
 - 使用 `set_context()`

官方原文链接

验证器

大多数情况下，您在 REST framework 中处理验证时，只需依赖默认的字验证，或者在序列化类或字段类上编写明确的验证方法。

但是，有时你会希望将验证逻辑放置到可重用组件中，以便在整个代码库中轻松地重用它。这可以通过使用验证器函数和验证器类来实现。

REST framework 中的验证

Django REST framework 序列化器中的验证与 Django `ModelForm` 类中验证的工作方式有点

不同。

使用 `ModelForm`，验证一部分在表单上执行，一部分在模型实例上执行。使用 REST framework，验证完全在序列化类上执行。这是有优势的，原因如下：

- 它使问题适当的分离，让代码行为变的更加清晰。
- 使用快捷的 `ModelSerializer` 类和使用显式的 `Serializer` 类可以轻松切换。任何用于 `ModelSerializer` 的验证行为都很容易复制。
- 打印序列化类实例的 `repr` 将会显示它应用的验证规则。在模型实例上没有额外的隐藏验证行为（因为全在序列化类上）。

当你使用 `ModelSerializer` 时，所有验证都会自动为你处理。如果你想要改为使用 `Serializer` 类，那么你需要明确定义验证规则。

举个栗子

作为 REST framework 如何使用显式验证的示例，我们将采用一个简单的模型类，该类具有唯一性约束的字段。

```
1. class CustomerReportRecord(models.Model):
2.     time_raised = models.DateTimeField(default=timezone.now, editable=False)
3.     reference = models.CharField(unique=True, max_length=20)
4.     description = models.TextField()
```

下面是一个基本的 `ModelSerializer`，我们可以使用它来创建或更新 `CustomerReportRecord` 的实例：

```
1. class CustomerReportSerializer(serializers.ModelSerializer):
2.     class Meta:
3.         model = CustomerReportRecord
```

现在使用 `manage.py shell` 打开 Django shell

```
1. >>> from project.example.serializers import CustomerReportSerializer
2. >>> serializer = CustomerReportSerializer()
3. >>> print(repr(serializer))
4. CustomerReportSerializer():
5.     id = IntegerField(label='ID', read_only=True)
6.     time_raised = DateTimeField(read_only=True)
7.     reference = CharField(max_length=20, validators=
8.         [<UniqueValidator(queryset=CustomerReportRecord.objects.all())>])
9.     description = CharField(style={'type': 'textarea'})
```

这里有趣的是 `reference` 字段。我们可以看到唯一性约束由序列化字段上的验证器明确执行。

由于这种更明确的风格，REST framework 包含一些在核心 Django 中没有的验证器类。这些类在下面详细说明。

UniqueValidator

该验证器可用于在模型字段上强制实施 `unique=True` 约束。它需要一个必需的参数和一个可选的 `messages` 参数：

- `queryset` 必须 - 这是验证唯一性的查询集。
- `message` - 验证失败时使用的错误消息。
- `lookup` - 用于查找已验证值的现有实例。默认为 `'exact'`。

这个验证器应该应用于序列化字段，如下所示：

```
1. from rest_framework.validators import UniqueValidator
2.
3. slug = SlugField(
4.     max_length=100,
5.     validators=[UniqueValidator(queryset=BlogPost.objects.all())]
6. )
```

UniqueTogetherValidator

此验证器可用于在模型实例上强制实施 `unique_together` 约束。它有两个必需的参数和一个可选的 `messages` 参数：

- `queryset` 必须 - 这是验证唯一性的查询集。
- `fields` 必须 - 一个存放字段名称的列表或者元组，这个集合必须是唯一的（意思是集合中的字段代表的一组值不能同时出现在两条数据中）。这些字段必须都是序列化类中的字段。
- `message` - 验证失败时使用的错误消息。

验证器应该应用于序列化类，如下所示：

```
1. from rest_framework.validators import UniqueTogetherValidator
2.
3. class ExampleSerializer(serializers.Serializer):
4.     # ...
5.     class Meta:
6.         # TODO items belong to a parent list, and have an ordering defined
```



```

7.         # by the 'position' field. No two items in a given list may share
8.         # the same position.
9.         validators = [
10.             UniqueTogetherValidator(
11.                 queryset=ToDoItem.objects.all(),
12.                 fields=('list', 'position')
13.             )
14.         ]

```

注意： `UniqueTogetherValidation` 类总是施加一个隐式约束，即它所应用的所有字段都是按需处理的。具有 `default` 值的字段是一个例外，因为它们总是提供一个值，即使在用户输入中省略了这个值。

UniqueForDateValidator

UniqueForMonthValidator

UniqueForYearValidator

这些验证器可用于强制实施模型实例上的 `unique_for_date`，`unique_for_month` 和 `unique_for_year` 约束。他们有以下参数：

- `queryset` 必须 - 这是验证唯一性的查询集。
- `field` 必须 - 在给定日期范围内需要被验证唯一性的字段的名称。该字段必须是序列化类中的字段。
- `date_field` 必须 - 将用于确定唯一性约束的日期范围的字段名称。该字段必须是序列化类中的字段。
- `message` - 验证失败时使用的错误消息。

验证器应该应用于序列化类，如下所示：

```

1. from rest_framework.validators import UniqueForYearValidator
2.
3. class ExampleSerializer(serializers.Serializer):
4.     # ...
5.     class Meta:
6.         # Blog posts should have a slug that is unique for the current year.
7.         validators = [

```

```

8.         UniqueForYearValidator(
9.             queryset=BlogPostItem.objects.all(),
10.            field='slug',
11.            date_field='published'
12.        )
13.    ]

```

我解释下，上面例子的意思是，在 `published` 日期所在的年份中，`slug` 字段的值必须唯一，注意，不是要和 `published` 完全相等的日期，而是年份相等。 `unique_for_date`，`unique_for_month` 同理。

用于验证的日期字段应该始终存在于序列化类中。你不能简单地依赖模型类 `default=...`，因为默认值在验证运行之后才会生成。

你可能需要使用几种样式，具体取决于你希望 API 如何展现。如果你使用的是 `ModelSerializer`，可能只需依赖 REST framework 为你生成的默认值，但如果你使用的是 `Serializer` 或需要更明确的控制，请使用下面演示的样式。

与可写日期字段一起使用。

如果你希望日期字段是可写的，唯一值得注意的是你应该确保它始终可用于输入数据中，可以通过设置 `default` 参数或通过设置 `required=True` 来实现。

```
1. published = serializers.DateTimeField(required=True)
```

与只读日期字段一起使用。

如果你希望日期字段可见，但用户无法编辑，请设置 `read_only=True` 并另外设置 `default=...` 参数。

```
1. published = serializers.DateTimeField(read_only=True, default=timezone.now)
```

该字段对用户不可写，但默认值仍将传递给 `validated_data`。

与隐藏的日期字段一起使用。

如果你希望日期字段对用户完全隐藏，请使用 `HiddenField`。该字段类型不接受用户输入，而是始终将其默认值返回到序列化类中的 `validated_data`。

```
1. published = serializers.HiddenField(default=timezone.now)
```

注意：`UniqueFor<Range>Validation` 类总是施加一个隐式约束，即它所应用的所有字段都是按需

处理的。具有 `default` 值的字段是一个例外，因为它们总是提供一个值，即使在用户输入中省略了这个值。

Advanced field defaults

在序列化类的多个字段中应用的验证器有时不需要由 API 客户端提供的字段输入，但它可以用作验证器的输入。

有两种模式可能需要这种验证：

- 使用 `HiddenField`。该字段将出现在 `validated_data` 中，但不会用在序列化输出表示中。
- 使用 `read_only=True` 的标准字段，同时也包含 `default=...` 参数。该字段将用于序列化输出表示中，但不能由用户直接设置。

REST framework 包含一些在这种情况下可能有用的默认值。

CurrentUserDefault

可用于表示当前用户的默认类。为了使用它，在实例化序列化类时，`'request'` 必须作为上下文字典的一部分提供。

```
1. owner = serializers.HiddenField(  
2.     default=serializers.CurrentUserDefault()  
3. )
```

CreateOnlyDefault

可用于在 `create` 操作期间仅设置默认参数的默认类。在 `update` 期间，该字段被省略。

它接受一个参数，这是在 `create` 操作期间应该使用的默认值或可调用参数。

```
1. created_at = serializers.DateTimeField(  
2.     read_only=True,  
3.     default=serializers.CreateOnlyDefault(timezone.now)  
4. )
```

验证器的限制

有一些不明确的情况，你需要显示处理验证，而不是依赖 `ModelSerializer` 生成的默认序列化

类。

在这些情况下，你可能希望通过为序列化类 `Meta.validators` 属性指定一个空列表来禁用自动生成的验证器。

可选字段

默认情况下 “unique together” 验证强制所有字段都是 `required=True`。在某些情况下，你可能希望显式将 `required=False` 应用于其中一个字段，在这种情况下，验证所需的行为是不明确的。

在这种情况下，你通常需要从序列化类中排除验证器，并且在 `.validate()` 方法中或在视图中显式地编写验证逻辑。

例如：

```
1. class BillingRecordSerializer(serializers.ModelSerializer):
2.     def validate(self, data):
3.         # Apply custom validation either here, or in the view.
4.
5.     class Meta:
6.         fields = ('client', 'date', 'amount')
7.         extra_kwargs = {'client': {'required': 'False'}}
8.         validators = [] # Remove a default "unique together" constraint.
```

更新嵌套序列化类

将更新应用于现有实例时，唯一性验证器将从唯一性检查中排除当前实例。当前实例在唯一性检查的上下文中可用，因为它作为序列化程序中的一个属性存在，最初在实例化序列化类时已使用 `instance=...` 传递。

在嵌套序列化类上进行更新操作时，无法应用此排除，因为该实例不可用。

你可能又一次需要明确地从序列化类中移除验证器，并将验证约束的代码显式写入 `.validate()` 方法或视图中。

调试复杂的案例

如果你不确定 `ModelSerializer` 类的默认行为，那么运行 `manage.py shell` 并打印序列化类实例通常是一个好主意，以便你可以检查它自动生成的字段和验证器。

```

1. >>> serializer = MyComplexModelSerializer()
2. >>> print(serializer)
3. class MyComplexModelSerializer:
4.     my_fields = ...

```

还要记住，在复杂情况下，明确定义序列化类通常会更好，而不是依赖默认的行为。虽然这样会写更多的代码，但确保了最终的行为更加透明。

ModelSerializer

编写自定义验证器

你可以使用 Django 现有的验证器，也可以编写自定义验证器。

基于函数

验证器可以是任何可调用对象，在失败时引发 `serializers.ValidationError`。

```

1. def even_number(value):
2.     if value % 2 != 0:
3.         raise serializers.ValidationError('This field must be an even number.')

```

字段级验证

你可以通过向 `Serializer` 子类添加 `.validate_<field_name>` 方法来指定自定义字段级验证。

基于类

要编写一个基于类的验证器，请使用 `__call__` 方法。基于类的验证器很有用，因为它们允许参数化和重用行为。

```

1. class MultipleOf(object):
2.     def __init__(self, base):
3.         self.base = base
4.
5.     def __call__(self, value):
6.         if value % self.base != 0:
7.             message = 'This field must be a multiple of %d.' % self.base
8.             raise serializers.ValidationError(message)

```

使用

`set_context()`

在一些高级的情况下，你可能想要在验证器中获取正在被验证的序列化字段。这时，你可以通过在基于类的验证器上声明 `set_context` 方法来完成此操作。

```
1. def set_context(self, serializer_field):
2.     # Determine if this is an update or a create operation.
3.     # In `__call__` we can then use that information to modify the validation
   behavior.
4.     self.is_update = serializer_field.parent.instance is not None
```

认证

- 身份验证
 - 如何确定身份验证
 - 设置认证方案
 - 未经授权和禁止响应
 - Apache mod_wsgi 特定的配置
- API 参考
 - BasicAuthentication
 - TokenAuthentication
 - 生成令牌
 - 通过使用信号
 - 通过暴露一个 API 端点
 - 使用 Django admin
 - 使用 Django manage.py 命令
- SessionAuthentication
- RemoteUserAuthentication
- 自定义身份认证
 - 举个栗子

官方原文链接

前往掘金阅读

身份验证

身份验证是将传入请求与一组识别凭证（例如请求的用户或其签名的令牌）相关联的机制。然后，权限和限制策略可以使用这些凭据来确定请求是否应该被允许。

REST framework 提供了许多开箱即用的身份验证方案，同时也允许你实施自定义方案。

身份验证始终在视图的开始处运行，在执行权限和限制检查之前，在允许继续执行任何其他代码之前。

`request.user` 属性通常会设置为 `contrib.auth` 包的 `User` 类的一个实例。

`request.auth` 属性用于其他身份验证信息，例如，它可以用来表示请求已签名的身份验证令牌。

注意： 不要忘记， 身份验证本身不会（允许或不允许）传入的请求，它只是标识请求的凭据。

如何确定身份验证

认证方案总是被定义为一个类的列表。REST framework 将尝试使用列表中的每个类进行认证，并将使用成功认证的第一个类的返回值来设置 `request.user` 和 `request.auth`。

如果没有类进行身份验证，则将 `request.user` 设置为 `django.contrib.auth.models.AnonymousUser` 的实例，并将 `request.auth` 设置为 `None`。

可以使用 `UNAUTHENTICATED_USER` 和 `UNAUTHENTICATED_TOKEN` 设置修改未经身份验证的请求的 `request.user` 和 `request.auth` 的值。

设置认证方案

默认的认证方案可以使用 `DEFAULT_AUTHENTICATION_CLASSES` setting 全局设置。例如。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_AUTHENTICATION_CLASSES': (
3.         'rest_framework.authentication.BasicAuthentication',
4.         'rest_framework.authentication.SessionAuthentication',
5.     )
6. }
```

您还可以使用基于 `APIView` 类的视图，在每个视图或每个视图集的基础上设置身份验证方案。

```
1. from rest_framework.authentication import SessionAuthentication,
   BasicAuthentication
2. from rest_framework.permissions import IsAuthenticated
3. from rest_framework.response import Response
4. from rest_framework.views import APIView
5.
6. class ExampleView(APIView):
7.     authentication_classes = (SessionAuthentication, BasicAuthentication)
8.     permission_classes = (IsAuthenticated,)
9.
10.    def get(self, request, format=None):
11.        content = {
12.            'user': unicode(request.user), # `django.contrib.auth.User`
instance.
13.            'auth': unicode(request.auth), # None
14.        }
15.        return Response(content)
```


或者, 如果您将 `@api_view` 装饰器与基于函数的视图一起使用。

```
1. @api_view(['GET'])
2. @authentication_classes((SessionAuthentication, BasicAuthentication))
3. @permission_classes((IsAuthenticated,))
4. def example_view(request, format=None):
5.     content = {
6.         'user': unicode(request.user), # `django.contrib.auth.User` instance.
7.         'auth': unicode(request.auth), # None
8.     }
9.     return Response(content)
```

未经授权和禁止响应

当未经身份验证的请求被拒绝时, 有两种不同的错误代码可能是合适的。

- [HTTP 401 Unauthorized][http401]
- [HTTP 403 Permission Denied][http403]

HTTP 401 响应必须始终包含 `WWW-Authenticate` header, 该 header 指示客户端如何进行身份验证。 HTTP 403 响应不包含 `WWW-Authenticate` header。

将使用哪种响应取决于认证方案。尽管可能正在使用多种认证方案, 但只能使用一种方案来确定响应的类型。 在确定响应类型时使用视图上设置的第一个认证类。

请注意, 当请求可以成功进行身份验证时, 仍然可能会因为权限而被拒绝, 在这种情况下, 将始终使用 `403 Permission Denied` 响应, 而不管身份验证方案如何。

Apache mod_wsgi 特定的配置

请注意, 如果使用 mod_wsgi 部署到 Apache, 授权 header 默认情况下不会传递到 WSGI 应用程序, 因为它假定认证将由 Apache 处理, 而不是在应用程序级别处理。

如果您正在部署到 Apache 并使用任何基于非会话的身份验证, 则需要明确配置 mod_wsgi 以将所需的 headers 传递给应用程序。这可以通过在适当的上下文中指定 `WSGIPassAuthorization` 指令并将其设置为 `'On'` 来完成。

```
1. # this can go in either server config, virtual host, directory or .htaccess
2. WSGIPassAuthorization On
```

API 参考

BasicAuthentication

该认证方案使用 HTTP Basic Authentication，并根据用户的用户名和密码进行签名。Basic Authentication 通常只适用于测试。

如果成功通过身份验证，`BasicAuthentication` 将提供以下凭据。

- `request.user` 是一个 Django `User` 实例。
- `request.auth` 是 `None`。

未经身份验证的响应被拒绝将导致 `HTTP 401 Unauthorized` 的响应和相应的 `WWW-Authenticate` header。例如：

```
1. WWW-Authenticate: Basic realm="api"
```

注意：如果您在生产环境中使用 `BasicAuthentication`，则必须确保您的 API 仅可通过 `https` 访问。您还应该确保您的 API 客户端将始终在登录时重新请求用户名和密码，并且永远不会将这些详细信息存储到持久化存储中。

TokenAuthentication

此认证方案使用简单的基于令牌的 HTTP 认证方案。令牌身份验证适用于 client-server 架构，例如本机桌面和移动客户端。

要使用 `TokenAuthentication` 方案，您需要将认证类配置为包含 `TokenAuthentication`，并在 `INSTALLED_APPS` 设置中另外包含 `rest_framework.authtoken`：

```
1. INSTALLED_APPS = (  
2.     ...  
3.     'rest_framework.authtoken'  
4. )
```

注意：确保在更改设置后运行 `manage.py migrate`。 `rest_framework.authtoken` 应用程序提供 Django 数据库迁移。

您还需要为您的用户创建令牌。

```
1. from rest_framework.authtoken.models import Token
2.
3. token = Token.objects.create(user=...)
4. print token.key
```

对于客户端进行身份验证，令牌密钥应包含在 `Authorization` HTTP header 中。关键字应以字符串文字 “Token” 为前缀，用空格分隔两个字符串。例如：

```
1. Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

注意：如果您想在 header 中使用不同的关键字（例如 `Bearer` ），只需子类化 `TokenAuthentication` 并设置 `keyword` 类变量。

如果成功通过身份验证，`TokenAuthentication` 将提供以下凭据。

- `request.user` 是一个 Django `User` 实例。
- `request.auth` 是一个 `rest_framework.authtoken.models.Token` 实例。

未经身份验证的响应被拒绝将导致 `HTTP 401 Unauthorized` 的响应和相应的 `WWW-Authenticate` header。例如：

```
1. WWW-Authenticate: Token
```

`curl` 命令行工具可能对测试令牌认证的 API 有用。例如：

```
1. curl -X GET http://127.0.0.1:8000/api/example/ -H 'Authorization: Token
9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b'
```

注意：如果您在生产中使用 `TokenAuthentication` ，则必须确保您的 API 只能通过 `https` 访问。

生成令牌

通过使用信号

如果您希望每个用户都拥有一个自动生成的令牌，则只需捕捉用户的 `post_save` 信号即可。

```
1. from django.conf import settings
2. from django.db.models.signals import post_save
3. from django.dispatch import receiver
4. from rest_framework.authtoken.models import Token
```

```

5.
6. @receiver(post_save, sender=settings.AUTH_USER_MODEL)
7. def create_auth_token(sender, instance=None, created=False, **kwargs):
8.     if created:
9.         Token.objects.create(user=instance)

```

请注意，您需要确保将此代码片段放置在已安装的 `models.py` 模块或 Django 启动时将导入的其他某个位置。

如果您已经创建了一些用户，则可以为所有现有用户生成令牌，例如：

```

1. from django.contrib.auth.models import User
2. from rest_framework.authtoken.models import Token
3.
4. for user in User.objects.all():
5.     Token.objects.get_or_create(user=user)

```

通过暴露一个 API 端点

使用 `TokenAuthentication` 时，您可能希望为客户提供一种机制，以获取给定用户名和密码的令牌。REST framework 提供了一个内置的视图来支持这种行为。要使用它，请将 `obtain_auth_token` 视图添加到您的 URLconf 中：

```

1. from rest_framework.authtoken import views
2. urlpatterns += [
3.     url(r'^api-token-auth/', views.obtain_auth_token)
4. ]

```

请注意，模式的 URL 部分可以是任何你想使用的。

当使用表单数据或 JSON 将有效的 `username` 和 `password` 字段发布到视图时，`obtain_auth_token` 视图将返回 JSON 响应：

```

1. { 'token' : '9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b' }

```

请注意，缺省的 `obtain_auth_token` 视图显式使用 JSON 请求和响应，而不是使用你设置的默认的渲染器和解析器类。

默认情况下，没有权限或限制应用于 `obtain_auth_token` 视图。如果您希望应用 throttling，则需要重写视图类，并使用 `throttle_classes` 属性包含它们。

如果你需要自定义 `obtain_auth_token` 视图，你可以通过继承 `ObtainAuthToken` 视图类来实现，并在你的 url conf 中使用它。

例如，您可能会返回超出 `token` 值的其他用户信息：

```
1. from rest_framework.authtoken.views import ObtainAuthToken
2. from rest_framework.authtoken.models import Token
3. from rest_framework.response import Response
4.
5. class CustomAuthToken(ObtainAuthToken):
6.
7.     def post(self, request, *args, **kwargs):
8.         serializer = self.serializer_class(data=request.data,
9.                                             context={'request': request})
10.        serializer.is_valid(raise_exception=True)
11.        user = serializer.validated_data['user']
12.        token, created = Token.objects.get_or_create(user=user)
13.        return Response({
14.            'token': token.key,
15.            'user_id': user.pk,
16.            'email': user.email
17.        })
```

还有 `urls.py`：

```
1. urlpatterns += [
2.     url(r'^api-token-auth/', CustomAuthToken.as_view())
3. ]
```

使用 Django admin

也可以通过管理界面手动创建令牌。如果您使用的用户群很大，我们建议您对 `TokenAdmin` 类进行修补以根据需要对其进行定制，更具体地说，将 `user` 字段声明为 `raw_field`。

`your_app/admin.py`：

```
1. from rest_framework.authtoken.admin import TokenAdmin
2.
3. TokenAdmin.raw_id_fields = ('user',)
```

使用 Django manage.py 命令

从版本 3.6.4 开始，可以使用以下命令生成用户令牌：

```
1. ./manage.py drf_create_token <username>
```

此命令将返回给定用户的 API 令牌，如果它不存在则创建它：

```
1. Generated token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b for user user1
```

如果您想重新生成令牌（例如，它已被泄漏），则可以传递一个附加参数：

```
1. ./manage.py drf_create_token -r <username>
```

SessionAuthentication

此认证方案使用 Django 的默认 session 后端进行认证。Session 身份验证适用于与您的网站在同一会话环境中运行的 AJAX 客户端。

如果成功通过身份验证，则 `SessionAuthentication` 会提供以下凭据。

- `request.user` 是一个 Django `User` 实例。
- `request.auth` 是 `None`。

未经身份验证的响应被拒绝将导致 `HTTP 403 Forbidden` 响应。

如果您在 `SessionAuthentication` 中使用 AJAX 风格的 API，则需要确保为任何“不安全”的 HTTP 方法调用（例如 `PUT`，`PATCH`，`POST` 或 `DELETE` 请求）包含有效的 CSRF 令牌。

警告：创建登录页面时应该始终使用 Django 的标准登录视图。这将确保您的登录视图得到适当的保护。

REST framework 中的 CSRF 验证与标准 Django 略有不同，因为需要同时支持基于 session 和非基于 session 的身份验证。这意味着只有经过身份验证的请求才需要 CSRF 令牌，并且可以在没有 CSRF 令牌的情况下发送匿名请求。此行为不适用于应始终应用 CSRF 验证的登录视图。

RemoteUserAuthentication

这种身份验证方案允许您将身份验证委托给您的 Web 服务器，该服务器设置 `REMOTE_USER` 环境变量。

要使用它，你必须在你的 `AUTHENTICATION_BACKENDS` 设置中有

`django.contrib.auth.backends.RemoteUserBackend`（或者一个子类）。默认情况

下，`RemoteUserBackend` 为不存在的用户名创建 `User` 对象。要改变这个和其他行为，请参考 Django 文档。

如果成功通过身份验证，`RemoteUserAuthentication` 将提供以下凭据：

- `request.user` 是一个 Django `User` 实例。
- `request.auth` 是 `None`。

有关配置验证方法的信息，请参阅您的 Web 服务器的文档，例如：

- [Apache Authentication How-To](#)
- [NGINX \(Restricting Access\)](#)

自定义身份认证

要实现自定义身份验证方案，请继承 `BaseAuthentication` 并重写 `.authenticate(self, request)` 方法。如果认证成功，该方法应返回 `(user, auth)` 的二元组，否则返回 `None`。

在某些情况下，您可能想要从 `.authenticate()` 方法引发 `AuthenticationFailed` 异常而不是返回 `None`。

通常你应该采取的方法是：

- 如果不尝试认证，则返回 `None`。任何其他正在使用的身份验证方案仍将被检查。
- 如果尝试身份验证但失败了，请引发 `AuthenticationFailed` 异常。无论是否进行任何权限检查，都将立即返回错误响应，并且不再检查任何其他身份验证方案。

您也可以重写 `.authenticate_header(self, request)` 方法。如果实现，它应该返回一个字符串，该字符串将用作 `HTTP 401 Unauthorized` 响应中的 `WWW-Authenticate` header 的值。

如果未覆盖 `.authenticate_header()` 方法，那么当未经身份验证的请求被拒绝访问时，身份验证方案将返回 `HTTP 403 Forbidden` 响应。

Note: 当请求对象的 `.user` 或 `.auth` 属性调用您的自定义身份验证器时，您可能会看到 `AttributeError` 作为 `WrappedAttributeError` 被重新引发。这对于防止原始异常被外部属性访问所抑制是必要的。Python 不会识别 `AttributeError` 来自您的自定义身份验证器，而是会假设请求对象没有 `.user` 或 `.auth` 属性。这些错误应该由您的验证器修复或以其他方式处理。

举个栗子

下面的示例将根据名为 `"X_USERNAME"` 的自定义请求头中的用户名对任何传入请求进行身份验证。

```
1. from django.contrib.auth.models import User
2. from rest_framework import authentication
3. from rest_framework import exceptions
4.
5. class ExampleAuthentication(authentication.BaseAuthentication):
6.     def authenticate(self, request):
7.         username = request.META.get('X_USERNAME')
8.         if not username:
9.             return None
10.
11.         try:
12.             user = User.objects.get(username=username)
13.         except User.DoesNotExist:
14.             raise exceptions.AuthenticationFailed('No such user')
15.
16.         return (user, None)
```


权限

- [权限](#)
 - [如何确定权限](#)
 - [对象级权限](#)
 - [对象级权限的限制](#)
- [设置权限策略](#)
- [API 参考](#)
 - [AllowAny](#)
 - [IsAuthenticated](#)
 - [IsAdminUser](#)
 - [IsAuthenticatedOrReadOnly](#)
 - [DjangoModelPermissions](#)
 - 使用不包含 `queryset` 属性的视图。
- [DjangoModelPermissionsOrAnonReadOnly](#)
- [DjangoObjectPermissions](#)
- [自定义权限](#)
 - [举个栗子](#)

[官方原文链接](#)

权限

与 authentication 和 throttling 一起，permission 决定是应该接受还是拒绝访问请求。

权限检查总是在视图的最开始处运行，在任何其他代码被允许进行之前。权限检查通常会使用 `request.user` 和 `request.auth` 属性中的认证信息来确定是否允许传入请求。

权限用于授予或拒绝不同类别的用户访问 API 的不同部分。

最简单的权限是允许通过身份验证的用户访问，并拒绝未经身份验证的用户访问。这对应于 REST framework 中的 `IsAuthenticated` 类。

稍微宽松的权限会允许通过身份验证的用户完全访问，而未通过身份验证的用户只能进行只读访问。这对应于 REST framework 中的 `IsAuthenticatedOrReadOnly` 类。

如何确定权限

REST framework 中的权限总是被定义为权限类的列表。

在运行视图的主体之前，检查列表中的每个权限。如果任何权限检查失败，则会引发

`exceptions.PermissionDenied` 或 `exceptions.NotAuthenticated` 异常，并且视图的主体不会再运行。

当权限检查失败时，根据以下规则，将返回 “403 Forbidden” 或 “401 Unauthorized” 响应：

- 该请求已成功通过身份验证，但权限被拒绝。 – 将返回 *403 Forbidden* 响应。
- 该请求未成功通过身份验证，并且最高优先级身份验证类未添加 `WWW-Authenticate` header。 – 将返回 *403 Forbidden* 响应。
- 该请求未成功通过身份验证，不过最高优先级身份验证类添加了 `WWW-Authenticate` header。 – 返回一个 *HTTP 401 Unauthorized* 响应，并会带上一个适当的 `WWW-Authenticate` header。

对象级权限

REST framework 权限还支持对象级权限。对象级权限用于确定是否允许用户对特定对象进行操作，该特定对象通常是指模型实例。

`.get_object()` 被调用时，对象级权限由 REST framework 的通用视图执行。与视图级权限一样，如果用户不被允许对给定对象进行操作，则会引发 `exceptions.PermissionDenied` 异常。

如果您正在编写自己的视图并希望强制执行对象级权限，或者如果您在通用视图上重写了

`get_object` 方法，那么将需要显式地在你检索该对象时调用 `.check_object_permissions(request, obj)` 方法。

这将引发 `PermissionDenied` 或 `NotAuthenticated` 异常，或者只是在视图具有适当的权限时才返回。

例如：

```
1. def get_object(self):
2.     obj = get_object_or_404(self.get_queryset(), pk=self.kwargs["pk"])
3.     self.check_object_permissions(self.request, obj)
4.     return obj
```

对象级权限的限制

出于性能原因，通用视图在返回对象列表时不会自动将对象级权限应用于查询集中的每个实例。

通常，当您使用对象级权限时，您还需要适当地过滤查询集，以确保用户只能看到他们被允许查看的实例。

设置权限策略

默认权限策略可以使用 `DEFAULT_PERMISSION_CLASSES` setting 全局设置。例如。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_PERMISSION_CLASSES': (
3.         'rest_framework.permissions.IsAuthenticated',
4.     )
5. }
```

如果未指定，则此设置默认为允许无限制访问：

```
1. 'DEFAULT_PERMISSION_CLASSES': (
2.     'rest_framework.permissions.AllowAny',
3. )
```

您还可以在基于 `APIView` 类的视图上设置身份验证策略。

```
1. from rest_framework.permissions import IsAuthenticated
2. from rest_framework.response import Response
3. from rest_framework.views import APIView
4.
5. class ExampleView(APIView):
6.     permission_classes = (IsAuthenticated,)
7.
8.     def get(self, request, format=None):
9.         content = {
10.             'status': 'request was permitted'
11.         }
12.         return Response(content)
```

或者在基于 `@api_view` 装饰器的函数视图上设置。

```
1. from rest_framework.decorators import api_view, permission_classes
2. from rest_framework.permissions import IsAuthenticated
3. from rest_framework.response import Response
4.
5. @api_view(['GET'])
6. @permission_classes((IsAuthenticated, ))
7. def example_view(request, format=None):
8.     content = {
9.         'status': 'request was permitted'
10.     }
```

```
11.         return Response(content)
```

注意： 当你通过类属性或装饰器设置新的权限类时， `settings.py` 文件中的默认设置会被忽略。

API 参考

AllowAny

`AllowAny` 权限类将允许不受限制的访问，而不管该请求是否已通过身份验证或未经身份验证。

也不一定非要用此权限，可以通过为权限设置空列表或元组来实现相同的结果，但是你会发现，使用此权限使意图更加清晰。

IsAuthenticated

`IsAuthenticated` 权限类将拒绝任何未通过身份验证的用户的访问。

如果你希望 API 只能由注册用户访问，则可以使用此权限。

IsAdminUser

`IsAdminUser` 权限仅允许 `user.is_staff` 为 `True` 用户访问，其他任何用户都将被拒绝。

如果你希望 API 只能被部分受信任的管理员访问，则可以使用此权限。

IsAuthenticatedOrReadOnly

`IsAuthenticatedOrReadOnly` 允许通过身份验证的用户执行任何请求。未通过身份验证的用户只能请求“安全”的方法：`GET`，`HEAD` 或 `OPTIONS`。

如果你希望 API 允许匿名用户拥有读取权限，并且只允许对已通过身份验证的用户执行写入权限，则可以使用此权限。

DjangoModelPermissions

此权限类与 Django 的标准 `django.contrib.auth` 模型权限绑定。此权限只能应用于具有 `.queryset` 属性集的视图。只有在用户通过身份验证并分配了相关模型权限的情况下，才有权限访问。

- `POST` 请求要求用户在模型上具有 `add` 权限。
- `PUT` 和 `PATCH` 请求要求用户在模型上具有 `change` 权限。
- `DELETE` 请求要求用户在模型上具有 `delete` 权限。

默认行为也可以被重写以支持自定义模型权限。例如，你可能想要包含 `GET` 请求的 `view` 模型权限。

要自定义模型权限，请继承 `DjangoModelPermissions` 并设置 `.perms_map` 属性。有关详细信息，请参阅源代码。

使用不包含 `queryset` 属性的视图。

如果你将此权限与重写 `get_queryset()` 方法的视图一起使用，则视图上可能没有 `queryset` 属性。在这种情况下，我们建议使用 `sentinel` 查询集标记视图，以便此类可以确定所需的权限。例如：

```
1. queryset = User.objects.none() # Required for DjangoModelPermissions
```

DjangoModelPermissionsOrAnonReadOnly

与 `DjangoModelPermissions` 类似，但也允许未经身份验证的用户对 API 进行只读访问。

DjangoObjectPermissions

该权限类与 Django 的标准对象权限框架绑定，该框架允许对每个模型对象进行权限验证。为了使用此权限类，你还需要添加支持对象级权限的权限后端，例如 `django-guardian`。

与 `DjangoModelPermissions` 一样，此权限只能应用于具有 `.queryset` 属性或 `.get_queryset()` 方法的视图。只有在用户通过身份验证并且具有相关的每个对象权限和相关的模型权限后，才有权限访问。

- `POST` 请求要求用户对模型实例具有 `add` 权限。
- `PUT` 和 `PATCH` 请求要求用户对模型实例具有 `change` 权限。
- `DELETE` 请求要求用户对模型实例具有 `delete` 权限。

请注意，`DjangoObjectPermissions` 不需要 `django-guardian` 软件包，并且同样支持其他对象级别的后端。

与 `DjangoModelPermissions` 一样，你可以通过继承 `DjangoObjectPermissions` 并设置 `.perms_map` 属性来自定义模型权限。有关详细信息，请参阅源代码。

注意：如果你需要获取 `GET`，`HEAD` 和 `OPTIONS` 请求的对象级 `view` 权限，则还需要考虑添加 `DjangoObjectPermissionsFilter` 类，以确保列表端点只返回包含用户具有查看权限的对象的结果。

自定义权限

要实现自定义权限，请继承 `BasePermission` 并实现以下方法中的一个或两个：

- `.has_permission(self, request, view)`
- `.has_object_permission(self, request, view, obj)`

如果请求被授予访问权限，则方法应该返回 `True`，否则返回 `False`。

如果你需要测试一个请求是一个读操作还是一个写操作，你应该根据常量 `SAFE_METHODS` 检查请求方法，`SAFE_METHODS` 是一个包含 `'GET'`，`'OPTIONS'` 和 `'HEAD'` 的元组。例如：

```
1. if request.method in permissions.SAFE_METHODS:
2.     # Check permissions for read-only request
3. else:
4.     # Check permissions for write request
```

Note: 只有在视图级别 `has_permission` 检查已通过时才会调用实例级别的 `has_object_permission` 方法。还要注意，为了运行实例级检查，视图代码应该显式调用 `.check_object_permissions(request, obj)`。如果你使用的是通用视图，那么默认情况下会为您处理。（基于函数的视图将需要明确检查对象权限，在失败时引发 `PermissionDenied`。）

如果测试失败，自定义权限将引发 `PermissionDenied` 异常。要更改与异常相关的错误消息，请直接在你的自定义权限上实现 `message` 属性。否则将使用 `PermissionDenied` 的 `default_detail` 属性。

```
1. from rest_framework import permissions
2.
3. class CustomerAccessPermission(permissions.BasePermission):
4.     message = 'Adding customers not allowed.'
5.
6.     def has_permission(self, request, view):
7.         ...
```

举个栗子

以下是一个权限类的示例，该权限类将传入请求的 IP 地址与黑名单进行比对，并在 IP 被列入黑名单时拒绝该请求。

```
1. from rest_framework import permissions
2.
3. class BlacklistPermission(permissions.BasePermission):
4.     """
5.     Global permission check for blacklisted IPs.
6.     """
7.
8.     def has_permission(self, request, view):
9.         ip_addr = request.META['REMOTE_ADDR']
10.        blacklisted = Blacklist.objects.filter(ip_addr=ip_addr).exists()
11.        return not blacklisted
```

除了针对所有传入请求运行的全局权限，还可以创建对象级权限，这些权限仅针对影响特定对象实例的操作执行。例如：

```
1. class IsOwnerOrReadOnly(permissions.BasePermission):
2.     """
3.     Object-level permission to only allow owners of an object to edit it.
4.     Assumes the model instance has an `owner` attribute.
5.     """
6.
7.     def has_object_permission(self, request, view, obj):
8.         # Read permissions are allowed to any request,
9.         # so we'll always allow GET, HEAD or OPTIONS requests.
10.        if request.method in permissions.SAFE_METHODS:
11.            return True
12.
13.        # Instance must have an attribute named `owner`.
14.        return obj.owner == request.user
```

请注意，通用视图将检查适当的对象级权限，但如果你正在编写自己的自定义视图，则需要确保检查自己的对象级权限。您可以通过在拥有对象实例后从视图中调用

`self.check_object_permissions(request, obj)` 来完成此操作。如果任何对象级权限检查失败，此调用将引发适当的 `APIException`，否则将简单地返回。

另请注意，通用视图将仅检查单个模型实例的视图的对象级权限。如果你需要列表视图的对象级过滤，

权限

则需要单独过滤查询集。

限流

- [限流 \(Throttling\)](#)
 - [如何确定限流](#)
 - [设置限流策略](#)
 - [如何识别客户端](#)
 - [设置缓存](#)
- [API 参考](#)
 - [AnonRateThrottle](#)
 - [UserRateThrottle](#)
 - [ScopedRateThrottle](#)
- [自定义限流](#)
 - [举个栗子](#)

[官方原文链接](#)

限流 (Throttling)

限流与权限类似，因为它确定是否应该授权请求。限流阀指示临时状态，并用于控制客户端可以对API进行的请求速率。

与权限一样，可能会使用多种限流方式。你的 API 可能对未经身份验证的请求进行限流，对经过身份验证的请求限流较少。

如果你需要对 API 的不同部分使用不同的限流策略，由于某些服务特别占用资源，你可能想要使用同时有多种限流策略的另一种方案。

如果你想要同时实现爆发限流率和持续限流率，也可以使用多个限流阀。例如，你可能希望将用户限制为每分钟最多 60 个请求，并且每天最多 1000 个请求。

限流阀不一定只限制请求频率。例如，存储服务可能还需要对带宽进行限制，而付费数据服务可能希望对正在访问的某些记录进行限制。

如何确定限流

与权限和身份验证一样，REST framework 中的限流始终定义为类的列表。

在运行视图的主体之前，会检查列表中的每个限流阀。如果任何限流检查失败，将引发一个

`exceptions.Throttled` 异常，并且该视图的主体将不会再执行。

设置限流策略

可以使用 `DEFAULT_THROTTLE_CLASSES` 和 `DEFAULT_THROTTLE_RATES` setting 全局设置默认限流策略。例如：

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_THROTTLE_CLASSES': (
3.         'rest_framework.throttling.AnonRateThrottle',
4.         'rest_framework.throttling.UserRateThrottle'
5.     ),
6.     'DEFAULT_THROTTLE_RATES': {
7.         'anon': '100/day',
8.         'user': '1000/day'
9.     }
10. }
```

`DEFAULT_THROTTLE_RATES` 中使用的频率描述可能包括 `second` , `minute` , `hour` 或 `day` 作为限流期。

你还可以使用基于 `APIView` 类的视图，在每个视图或每个视图集的基础上设置限流策略。

```
1. from rest_framework.response import Response
2. from rest_framework.throttling import UserRateThrottle
3. from rest_framework.views import APIView
4.
5. class ExampleView(APIView):
6.     throttle_classes = (UserRateThrottle,)
7.
8.     def get(self, request, format=None):
9.         content = {
10.             'status': 'request was permitted'
11.         }
12.         return Response(content)
```

或者在基于 `@api_view` 装饰器的函数视图上设置。

```
1. @api_view(['GET'])
2. @throttle_classes([UserRateThrottle])
3. def example_view(request, format=None):
4.     content = {
5.         'status': 'request was permitted'
6.     }
7.     return Response(content)
```

如何识别客户端

`X-Forwarded-For` HTTP header 和 `REMOTE_ADDR` WSGI 变量用于唯一标识用于限流的客户端 IP 地址。如果存在 `X-Forwarded-For` header，则会使用它，否则将使用 WSGI 环境中的 `REMOTE_ADDR` 变量的值。

如果你需要严格标识唯一的客户端 IP 地址，则需要先通过设置 `NUM_PROXIES` setting 来配置 API 运行的应用代理的数量。该设置应该是一个零或更大的整数。如果设置为非零，则一旦任何应用程序代理 IP 地址首先被排除，客户端 IP 将被标识为 `X-Forwarded-For` header 中的最后一个 IP 地址。如果设置为零，则 `REMOTE_ADDR` 值将始终用作识别 IP 地址。

重要的是要理解，如果你配置了 `num_proxy` 设置，那么在一个唯一的 NAT 的网关后面的所有客户端将被当作一个单独的客户机来对待。

关于 `X-Forwarded-For` header 如何工作以及识别远程客户端 IP 的更多内容可以在[这里找到](#)。

设置缓存

REST framework 提供的限流类使用 Django 的缓存后端。你应该确保你已经设置了适当的缓存 setting。对于简单的设置，`LocMemCache` 后端的默认值应该没问题。有关更多详细信息，请参阅 [Django 的缓存文档](#)。

如果你需要使用 `'default'` 以外的缓存，则可以通过创建自定义限流类并设置 `cache` 属性来实现。例如：

```
1. class CustomAnonRateThrottle(AnonRateThrottle):
2.     cache = get_cache('alternate')
```

您需要记住还要在 `'DEFAULT_THROTTLE_CLASSES'` settings key 中设置自定义的限流类，或者使用 `throttle_classes` 视图属性。

API 参考

AnonRateThrottle

`AnonRateThrottle` 将永远限制未认证的用户。通过传入请求的 IP 地址生成一个唯一的密钥来进行限制。

允许的请求频率由以下之一决定（按优先顺序）。

- 类的 `rate` 属性，可以通过继承 `AnonRateThrottle` 并设置属性来提供。
- `DEFAULT_THROTTLE_RATES['anon']` 设置。

如果你想限制未知来源的请求频率，`AnonRateThrottle` 是合适的。

UserRateThrottle

`UserRateThrottle` 通过 API 将用户请求限制为给定的请求频率。用户标识用于生成一个唯一的密钥来加以限制。未经身份验证的请求将回退到使用传入请求的 IP 地址生成一个唯一的密钥来进行限制。

允许的请求频率由以下之一决定（按优先顺序）。

- 类的 `rate` 属性，可以通过继承 `UserRateThrottle` 并设置属性来提供。
- `DEFAULT_THROTTLE_RATES['user']` 设置。

一个 API 可能同时具有多个 `UserRateThrottles`。为此，请继承 `UserRateThrottle` 并为每个类设置一个唯一的“范围”。

例如，多个用户限流率可以通过使用以下类来实现.....

```
1. class BurstRateThrottle(UserRateThrottle):
2.     scope = 'burst'
3.
4. class SustainedRateThrottle(UserRateThrottle):
5.     scope = 'sustained'
```

...和以下设置。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_THROTTLE_CLASSES': (
3.         'example.throttles.BurstRateThrottle',
4.         'example.throttles.SustainedRateThrottle'
5.     ),
6.     'DEFAULT_THROTTLE_RATES': {
7.         'burst': '60/min',
8.         'sustained': '1000/day'
9.     }
10. }
```

如果希望对每个用户进行简单的全局速率限制，那么 `UserRateThrottle` 是合适的。

ScopedRateThrottle

`ScopedRateThrottle` 类可用于限制对 API 特定部分的访问。只有当正在访问的视图包含 `.throttle_scope` 属性时才会应用此限制。然后将请求的“范围”与唯一的用户标识或 IP 地址连接起来形成唯一的限流密钥。

允许的请求频率由 `DEFAULT_THROTTLE_RATES` setting 使用请求“范围”中的一个键确定。

例如，给出以下视图...

```
1. class ContactListView(APIView):
2.     throttle_scope = 'contacts'
3.     ...
4.
5. class ContactDetailView(APIView):
6.     throttle_scope = 'contacts'
7.     ...
8.
9. class UploadView(APIView):
10.    throttle_scope = 'uploads'
11.    ...
```

...和以下设置。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_THROTTLE_CLASSES': (
3.         'rest_framework.throttling.ScopedRateThrottle',
4.     ),
5.     'DEFAULT_THROTTLE_RATES': {
6.         'contacts': '1000/day',
7.         'uploads': '20/day'
8.     }
9. }
```

用户对 `ContactListView` 或 `ContactDetailView` 的请求将被限制为每天 1000 次。用户对 `UploadView` 的请求将被限制为每天 20 次。

自定义限流

要自定义限流，请继承 `BaseThrottle` 类并实现 `.allow_request(self, request, view)`

方法。如果请求被允许，该方法应该返回 `True`，否则返回 `False`。

或者，你也可以重写 `.wait()` 方法。如果实现，`.wait()` 应该返回建议的秒数，在尝试下一次请求之前等待，或者返回 `None`。如果 `.allow_request()` 先前已经返回 `False`，则只会调用 `.wait()` 方法。

如果 `.wait()` 方法被实现并且请求受到限制，那么 `Retry-After` header 将包含在响应中。

举个栗子

以下是限流的一个示例，随机地控制每 10 次请求中的 1 次。

```
1. import random
2.
3. class RandomRateThrottle(throttling.BaseThrottle):
4.     def allow_request(self, request, view):
5.         return random.randint(1, 10) != 1
```

过滤

- [过滤](#)
 - [根据当前用户进行过滤](#)
 - [根据 URL 进行过滤](#)
 - [根据查询参数进行过滤](#)
- [通用过滤器](#)
 - [设置过滤器后端](#)
 - [过滤和对象查找](#)
 - [覆盖初始查询集](#)
- [API 参考](#)
 - [DjangoFilterBackend](#)
 - [SearchFilter](#)
 - [OrderingFilter](#)
 - [指定可以根据哪些字段进行排序](#)
 - [指定默认顺序](#)
 - [DjangoObjectPermissionsFilter](#)
- [自定义通用过滤器](#)
 - [举个栗子](#)
 - [自定义接口](#)
 - [Pagination & schemas](#)

[官方原文链接](#)

过滤

REST framework 的通用列表视图的默认行为是从模型管理器返回整个查询集。通常你会希望 API 限制查询集返回的条目。

筛选 `GenericAPIView` 子类的查询集的最简单方法是重写 `.get_queryset()` 方法。

重写此方法允许你以多种不同方式自定义视图返回的查询集。

根据当前用户进行过滤

你可能需要过滤查询集，以确保只返回与当前通过身份验证的用户发出的请求相关的结果。

你可以基于 `request.user` 的值进行筛选来完成此操作。

例如：

```
1. from myapp.models import Purchase
2. from myapp.serializers import PurchaseSerializer
3. from rest_framework import generics
4.
5. class PurchaseList(generics.ListAPIView):
6.     serializer_class = PurchaseSerializer
7.
8.     def get_queryset(self):
9.         """
10.        This view should return a list of all the purchases
11.        for the currently authenticated user.
12.        """
13.        user = self.request.user
14.        return Purchase.objects.filter(purchaser=user)
```

根据 URL 进行过滤

另一种过滤方式可能涉及基于 URL 的某个部分限制查询集。

例如，如果你的 URL 配置包含这样的条目：

```
1. url('^purchases/(?P<username>+)/$', PurchaseList.as_view()),
```

然后，你可以编写一个视图，返回由 URL 的用户名部分过滤的 purchase 查询集：

```
1. class PurchaseList(generics.ListAPIView):
2.     serializer_class = PurchaseSerializer
3.
4.     def get_queryset(self):
5.         """
6.        This view should return a list of all the purchases for
7.        the user as determined by the username portion of the URL.
8.        """
9.        username = self.kwargs['username']
10.        return Purchase.objects.filter(purchaser__username=username)
```

根据查询参数进行过滤

过滤初始查询集的最后一个例子是根据 url 中的查询参数确定初始查询集。

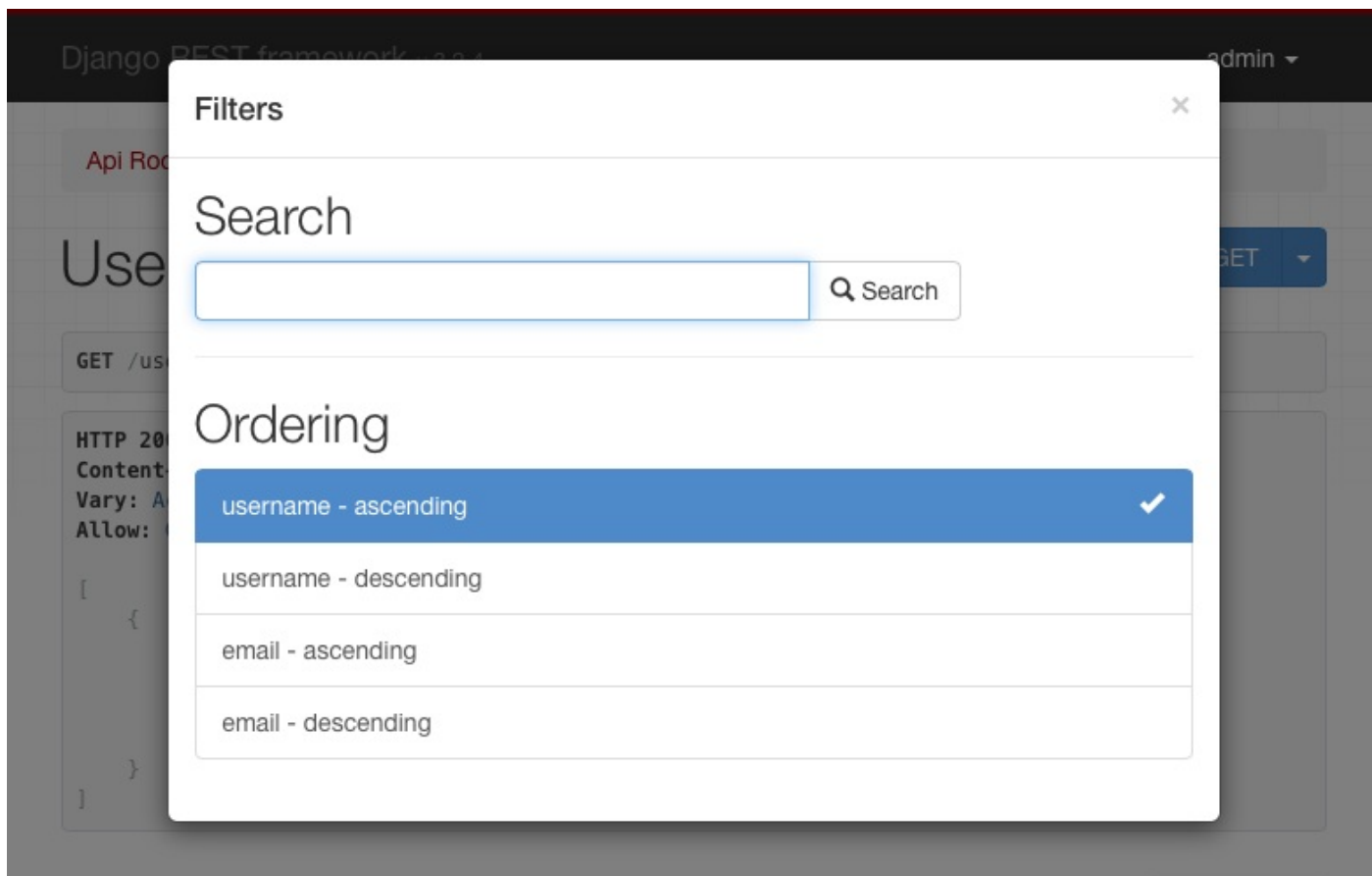
我们可以覆盖 `.get_queryset()` 来处理诸如 `http://example.com/api/purchases?username=denvercoder9` 的URL，并且只有在 URL 中包含 `username` 参数时才过滤查询集：

```
1. class PurchaseList(generics.ListAPIView):
2.     serializer_class = PurchaseSerializer
3.
4.     def get_queryset(self):
5.         """
6.         Optionally restricts the returned purchases to a given user,
7.         by filtering against a `username` query parameter in the URL.
8.         """
9.         queryset = Purchase.objects.all()
10.        username = self.request.query_params.get('username', None)
11.        if username is not None:
12.            queryset = queryset.filter(purchaser__username=username)
13.        return queryset
```

通用过滤器

除了能够覆盖默认的查询集外，REST framework 还包括对通用过滤后端的支持，使你可以轻松构建复杂的搜索和过滤器。

通用过滤器也可以在可浏览的 API 和管理 API 中将自己渲染为 HTML 控件。



设置过滤器后端

可以使用 `DEFAULT_FILTER_BACKENDS` setting 全局设置默认的过滤器后端。例如。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_FILTER_BACKENDS':
3.         ('django_filters.rest_framework.DjangoFilterBackend',)
```

你还可以使用基于 `GenericAPIView` 类的视图，在每个视图或视图集的基础上设置过滤器后端。

```
1. import django_filters.rest_framework
2. from django.contrib.auth.models import User
3. from myapp.serializers import UserSerializer
4. from rest_framework import generics
5.
6. class UserListView(generics.ListAPIView):
7.     queryset = User.objects.all()
8.     serializer_class = UserSerializer
9.     filter_backends = (django_filters.rest_framework.DjangoFilterBackend,)
```

过滤和对象查找

请注意，如果为一个视图配置了一个过滤器后端，那么除了用于筛选列表视图之外，它还将用于筛选返回单个对象的查询集。

例如，根据前面的示例以及 ID 为 `4675` 的产品，以下 URL 将返回相应的对象，或返回 404 响应，具体取决于给定产品实例是否满足过滤条件：

```
1. http://example.com/api/products/4675/?category=clothing&max_price=10.00
```

覆盖初始查询集

请注意，你可以同时重写的 `.get_queryset()` 和通用过滤，并且所有内容都将按预期工作。例如，如果产品与用户具有多对多关系，则可能需要编写一个如下所示的视图：

```
1. class PurchasedProductsList(generics.ListAPIView):
2.     """
3.     Return a list of all the products that the authenticated
4.     user has ever purchased, with optional filtering.
5.     """
6.     model = Product
7.     serializer_class = ProductSerializer
8.     filter_class = ProductFilter
9.
10.    def get_queryset(self):
11.        user = self.request.user
12.        return user.purchase_set.all()
```

API 参考

DjangoFilterBackend

`django-filter` 库包含一个 `DjangoFilterBackend` 类，它支持 REST framework 对字段过滤进行高度定制。

要使用 `DjangoFilterBackend`，首先安装 `django-filter`。然后将 `django_filters` 添加到 Django 的 `INSTALLED_APPS` 中

```
1. pip install django-filter
```

你现在应该将过滤器后端添加到设置中：

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_FILTER_BACKENDS':
3.     ('django_filters.rest_framework.DjangoFilterBackend',)
4. }
```

或者将过滤器后端添加到单个视图或视图集。

```
1. from django_filters.rest_framework import DjangoFilterBackend
2.
3. class UserListView(generics.ListAPIView):
4.     ...
5.     filter_backends = (DjangoFilterBackend,)
```

如果你只需要简单的基于等式的过滤，则可以在视图或视图集上设置 `filter_fields` 属性，列出你要过滤的一组字段。

```
1. class ProductList(generics.ListAPIView):
2.     queryset = Product.objects.all()
3.     serializer_class = ProductSerializer
4.     filter_backends = (DjangoFilterBackend,)
5.     filter_fields = ('category', 'in_stock')
```

这将自动为给定字段创建一个 `FilterSet` 类，并允许你发出如下请求：

```
1. http://example.com/api/products?category=clothing&in_stock=True
```

对于更高级的过滤要求，你应该在视图上指定 `FilterSet` 类。你可以在 [django-filter 文档](#) 中阅读有关 `FilterSet` 的更多信息。还建议你阅读 [DRF integration](#)。

SearchFilter

`SearchFilter` 类支持简单的基于单个查询参数的搜索，并且基于 Django 管理员的搜索功能。

在使用时，可浏览的 API 将包含一个 `SearchFilter` 控件：

Search

`SearchFilter` 类将仅在视图具有 `search_fields` 属性集的情况下应用。`search_fields` 属性应该是模型上文本类型字段的名称列表，例如 `CharField` 或 `TextField`。

```
1. class UserListView(generics.ListAPIView):
2.     queryset = User.objects.all()
3.     serializer_class = UserSerializer
4.     filter_backends = (filters.SearchFilter,)
5.     search_fields = ('username', 'email')
```

这将允许客户端通过查询来过滤列表中的项目，例如：

```
1. http://example.com/api/users?search=russell
```

你还可以使用查找 API 双下划线表示法对 `ForeignKey` 或 `ManyToManyField` 执行相关查找：

```
1. search_fields = ('username', 'email', 'profile__profession')
```

默认情况下，搜索将使用不区分大小写的部分匹配。搜索参数可能包含多个搜索词，它们应该是空格和（或）逗号分隔的。如果使用多个搜索条件，则只有在所有提供的条件匹配的情况下，对象才会返回到列表中。

搜索行为可以通过将各种字符预先添加到 `search_fields` 来限制。

- `'^'` 匹配起始部分。
- `'='` 完全匹配。
- `'@'` 全文搜索。（目前只支持 Django 的 MySQL 后端。）
- `'$'` 正则匹配。

例如：

```
1. search_fields = ('=username', '=email')
```

默认情况下，搜索参数被命名为 `'search'`，但这可能会被 `SEARCH_PARAM` setting 覆盖。

有关更多详细信息，请参阅 [Django 文档](#)。

OrderingFilter

`OrderingFilter`

类支持简单查询参数控制结果的排序。

Ordering

username - ascending	✓
username - descending	
email - ascending	
email - descending	

默认情况下，查询参数被命名为 `'ordering'`，但这可能会被 `ORDERING_PARAM` setting 覆盖。

例如，要通过 username 对用户排序：

```
1. http://example.com/api/users?ordering=username
```

客户端也可以通过在字段名称前加 `' - '` 来指定反向排序，如下所示：

```
1. http://example.com/api/users?ordering=-username
```

也可以指定多个排序：

```
1. http://example.com/api/users?ordering=account,username
```

指定可以根据哪些字段进行排序

建议你明确指定 API 应该允许在排序过滤器中使用哪些字段。你可以通过在视图上设置一个

`ordering_fields`

属性来完成此操作，如下所示：

```
1. class UserListView(generics.ListAPIView):
2.     queryset = User.objects.all()
3.     serializer_class = UserSerializer
4.     filter_backends = (filters.OrderingFilter,)
5.     ordering_fields = ('username', 'email')
```

这有助于防止意外的数据泄漏，例如允许用户根据密码哈希字段或其他敏感数据进行排序。

如果你未在视图上指定 `ordering_fields` 属性，则过滤器类将默认允许用户过滤由 `serializer_class` 属性指定的序列化类中的任何可读字段。

如果你确信视图使用的查询集不包含任何敏感数据，则还可以通过使用特殊值 `'__all__'` 明确指定视图允许在任何模型字段或查询集聚合上进行排序。

```
1. class BookingsListView(generics.ListAPIView):
2.     queryset = Booking.objects.all()
3.     serializer_class = BookingSerializer
4.     filter_backends = (filters.OrderingFilter,)
5.     ordering_fields = '__all__'
```

指定默认排序

如果在视图上设置了 `ordering` 属性，则将用作默认排序。

通常情况下，你应该通过在初始查询集上设置 `order_by` 来控制此操作，但是通过在视图上使用 `ordering` 参数，你可以指定排序方式，然后可以将其作为上下文自动传递到渲染的模板。这可以自动渲染列标题，如果它们用于排序结果。

```
1. class UserListView(generics.ListAPIView):
2.     queryset = User.objects.all()
3.     serializer_class = UserSerializer
4.     filter_backends = (filters.OrderingFilter,)
5.     ordering_fields = ('username', 'email')
6.     ordering = ('username',)
```

`ordering` 属性可以是一个字符串或者字符串列表（元组）。

DjangoObjectPermissionsFilter

`DjangoObjectPermissionsFilter` 旨在与 `django-guardian` 软件包一起使用，添加了自定义 `'view'` 的权限。过滤器将确保查询集仅返回用户具有适当查看权限的对象。

如果你使用的是 `DjangoObjectPermissionsFilter`，那么你可能还需要添加适当的对象权限类，以确保用户只有在具有适当对象权限的情况下才能对实例进行操作。做到这一点的最简单方法是继承 `DjangoObjectPermissions` 并为 `perms_map` 属性添加 `'view'` 权限。

使用 `DjangoObjectPermissionsFilter` 和 `DjangoObjectPermissions` 的完整示例可能如下

过滤

所示。

permissions.py:

```
1. class CustomObjectPermissions(permissions.DjangoObjectPermissions):
2.     """
3.     Similar to `DjangoObjectPermissions`, but adding 'view' permissions.
4.     """
5.     perms_map = {
6.         'GET': ['%(app_label)s.view_%(model_name)s'],
7.         'OPTIONS': ['%(app_label)s.view_%(model_name)s'],
8.         'HEAD': ['%(app_label)s.view_%(model_name)s'],
9.         'POST': ['%(app_label)s.add_%(model_name)s'],
10.        'PUT': ['%(app_label)s.change_%(model_name)s'],
11.        'PATCH': ['%(app_label)s.change_%(model_name)s'],
12.        'DELETE': ['%(app_label)s.delete_%(model_name)s'],
13.    }
```

views.py:

```
1. class EventViewSet(viewsets.ModelViewSet):
2.     """
3.     Viewset that only lists events if user has 'view' permissions, and only
4.     allows operations on individual events if user has appropriate 'view',
5.     'add',
6.     'change' or 'delete' permissions.
7.     """
8.     queryset = Event.objects.all()
9.     serializer_class = EventSerializer
10.    filter_backends = (filters.DjangoObjectPermissionsFilter,)
11.    permission_classes = (myapp.permissions.CustomObjectPermissions,)
```

自定义通用过滤器

你还可以提供自己的通用过滤器后端，或者编写一个可供其他开发人员使用的可安装应用程序。

为此，请继承 `BaseFilterBackend`，并覆盖 `.filter_queryset(self, request, queryset, view)` 方法。该方法应该返回一个新的，过滤的查询集。

除了允许客户端执行搜索和过滤外，通用过滤器后端可用于限制哪些对象应该对给定的请求或用户可见。

举个栗子

你可能需要限制用户只能看到他们创建的对象。

```
1. class IsOwnerFilterBackend(filters.BaseFilterBackend):
2.     """
3.     Filter that only allows users to see their own objects.
4.     """
5.     def filter_queryset(self, request, queryset, view):
6.         return queryset.filter(owner=request.user)
```

我们可以通过覆盖视图上的 `get_queryset()` 来实现相同的行为，但使用过滤器后端允许你更轻松地将此限制添加到多个视图，或者将其应用于整个 API。

自定义接口

通用过滤器也可以在可浏览的 API 中渲染接口。为此，你应该实现一个 `to_html()` 方法，该方法返回过滤器的渲染 HTML 表示。此方法应具有以下签名：

```
to_html(self, request, queryset, view)
```

该方法应该返回一个渲染的 HTML 字符串。

Pagination & schemas

通过实现 `get_schema_fields()` 方法，你还可以使过滤器控件可用于 REST framework 提供的模式自动生成。此方法应具有以下签名：

```
get_schema_fields(self, view)
```

该方法应该返回一个 `coreapi.Field` 实例列表。

分页

- [分页](#)
 - [设置分页样式](#)
 - [修改分页样式](#)
- [API 参考](#)
 - [PageNumberPagination](#)
 - [Setup](#)
 - [Configuration](#)
- [LimitOffsetPagination](#)
 - [Setup](#)
 - [Configuration](#)
- [CursorPagination](#)
 - [细节和限制](#)
 - [Setup](#)
 - [Configuration](#)
- [自定义分页样式](#)
 - [举个栗子](#)
 - [使用你的自定义分页类](#)
 - [分页和模式](#)
- [HTML 分页控件](#)
 - [自定义控件](#)
 - [低级 API](#)

[官方原文链接](#)

分页

REST framework 包含对可定制分页样式的支持。这使你可以将较大的结果集分成单独的数据页面。

分页 API 支持：

- 以分页链接的形式作为响应内容的一部分。
- 以分页链接的形式包含在响应的 header 中，如 `Content-Range` 或 `Link` 。

内置的样式目前是以分页链接的形式作为响应内容的一部分。使用可浏览的 API 时，此样式更易于访问。

分页仅在你使用通用视图或视图集时自动执行。如果你使用的是常规 `APIView`，则需要自己调用分页 API 以确保返回分页响应。示例请参阅 `mixins.ListModelMixin` 和 `generics.GenericAPIView` 类的源代码。

可以通过将分页类设置为 `None`，关闭分页。

设置分页样式

分页样式可以使用 `DEFAULT_PAGINATION_CLASS` 和 `PAGE_SIZE` setting key 全局设置。例如，要使用内置的 `limit/offset` 分页，你可以这样做：

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_PAGINATION_CLASS':
3.         'rest_framework.pagination.LimitOffsetPagination',
4.     'PAGE_SIZE': 100
5. }
```

请注意，你需要设置分页类和应使用的页面大小。默认情况下，`DEFAULT_PAGINATION_CLASS` 和 `PAGE_SIZE` 都是 `None`。

你还可以使用 `pagination_class` 属性在单个视图上设置分页类。通常，你希望在整个 API 中使用相同的分页样式，但你可能希望在每个视图的基础上更改分页的各个方面，例如默认或最大页面大小。

修改分页样式

如果要修改分页样式的特定方面，则需要继承其中一个分页类，并设置要更改的属性。

```
1. class LargeResultsSetPagination(PageNumberPagination):
2.     page_size = 1000
3.     page_size_query_param = 'page_size'
4.     max_page_size = 10000
5.
6. class StandardResultsSetPagination(PageNumberPagination):
7.     page_size = 100
8.     page_size_query_param = 'page_size'
9.     max_page_size = 1000
```

然后，你可以使用 `.pagination_class` 属性将新样式应用于视图：

```
1. class BillingRecordsView(generics.ListAPIView):
2.     queryset = Billing.objects.all()
3.     serializer_class = BillingRecordsSerializer
4.     pagination_class = LargeResultsSetPagination
```

或者使用 `DEFAULT_PAGINATION_CLASS` setting key 全局应用样式。例如：

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_PAGINATION_CLASS':
3.     'apps.core.pagination.StandardResultsSetPagination'
4. }
```

API 参考

PageNumberPagination

此分页样式在请求查询参数中接受一个页码值。

Request :

```
1. GET https://api.example.org/accounts/?page=4
```

Response :

```
1. HTTP 200 OK
2. {
3.     "count": 1023
4.     "next": "https://api.example.org/accounts/?page=5",
5.     "previous": "https://api.example.org/accounts/?page=3",
6.     "results": [
7.         ...
8.     ]
9. }
```

Setup

要全局启用 `PageNumberPagination` 样式，请使用以下配置，并根据需要设置 `PAGE_SIZE`：

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_PAGINATION_CLASS':
3.     'rest_framework.pagination.PageNumberPagination',
4.     'PAGE_SIZE': 100
5. }
```

如果使用的是 `GenericAPIView` 的子类，还可以设置 `pagination_class` 属性以在每个视图的基础上选择 `PageNumberPagination`。

Configuration

`PageNumberPagination` 类包含一些可以被覆盖以修改分页样式的属性。

要设置这些属性，你应该继承 `PageNumberPagination` 类，然后像上面那样启用你的自定义分页类。

- `django_paginator_class` - 要使用的 Django Paginator 类。默认是 `django.core.paginator.Paginator`，对于大多数用例来说应该没问题。
- `page_size` - 指定页面大小的数字值。如果设置，则会覆盖 `PAGE_SIZE` setting。默认值与 `PAGE_SIZE` setting key 相同。
- `page_query_param` - 一个字符串值，指定用于分页控件的查询参数的名称。
- `page_size_query_param` - 一个字符串值，指定查询参数的名称，允许客户端根据每个请求设置页面大小。默认为 `None`，表示客户端可能无法控制所请求的页面大小。
- `max_page_size` - 一个数字值，表示允许的最大页面大小。该属性仅在 `page_size_query_param` 也被设置时有效。
- `last_page_strings` - 字符串列表或元组，用于指定可能与 `page_query_param` 一起使用的值，用以请求集合中的最终页面。默认为 `('last',)`
- `template` - 在可浏览 API 中渲染分页控件时使用的模板的名称。可能会被覆盖以修改渲染样式，或设置为 `None` 以完全禁用 HTML 分页控件。默认为 `"rest_framework/pagination/numbers.html"`。

LimitOffsetPagination

这种分页样式反映了查找多个数据库记录时使用的语法。客户端包含 “limit” 和 “offset” 查询参数。limit 表示要返回的 item 的最大数量，并且等同于其他样式中的 `page_size`。offset 指定查询的起始位置与完整的未分类 item 集的关系。

Request:

```
1. GET https://api.example.org/accounts/?limit=100&offset=400
```

Response:

```
1. HTTP 200 OK
2. {
3.     "count": 1023
```

```

4.     "next": "https://api.example.org/accounts/?limit=100&offset=500",
5.     "previous": "https://api.example.org/accounts/?limit=100&offset=300",
6.     "results": [
7.         ...
8.     ]
9. }
```

Setup

要全局启用 `LimitOffsetPagination` 样式，请使用以下配置：

```

1. REST_FRAMEWORK = {
2.     'DEFAULT_PAGINATION_CLASS':
3.         'rest_framework.pagination.LimitOffsetPagination'
4. }
```

或者，你也可以设置一个 `PAGE_SIZE` 键。如果使用了 `PAGE_SIZE` 参数，则 `limit` 查询参数将是可选的，并且可能会被客户端忽略。

如果使用的是 `GenericAPIView` 子类，还可以设置 `pagination_class` 属性以基于每个视图选择 `LimitOffsetPagination`。

Configuration

`LimitOffsetPagination` 类包含一些可以被覆盖以修改分页样式的属性。

要设置这些属性，应该继承 `LimitOffsetPagination` 类，然后像上面那样启用你的自定义分页类。

- `default_limit` - 一个数字值，指定客户端在查询参数中未提供的 `limit`。默认值与 `PAGE_SIZE` setting key 相同。
- `limit_query_param` - 一个字符串值，指示 “limit” 查询参数的名称。默认为 `'limit'`。
- `offset_query_param` - 一个字符串值，指示 “offset” 查询参数的名称。默认为 `'offset'`。
- `max_limit` - 一个数字值，表示客户端可以要求的最大允许 `limit`。默认为 `None`。
- `template` - 在可浏览 API 中渲染分页控件时使用的模板的名称。可能会被覆盖以修改渲染样式，或设置为 `None` 以完全禁用 HTML 分页控件。默认为 `"rest_framework/pagination/numbers.html"`。

CursorPagination

基于游标的分页提供了一个不透明的“游标”指示器，客户端可以使用该指示器来翻阅结果集。此分页样式仅提供前向和反向控件，并且不允许客户端导航到任意位置。

基于游标的分页需要在结果集中存在唯一的，不变的 item 顺序。这种排序通常可以是记录上的创建时间戳，因为这确保了排序的一致性。

基于游标的分页比其他方案更复杂。它还要求结果集渲染固定顺序，并且不允许客户端任意索引结果集。但它确实提供了以下好处：

- 提供一致的分页视图。正确使用 `CursorPagination` 确保客户端在分页时不会看到同一个 item，即使在分页过程中其他客户端正在插入新 item。
- 支持使用非常大的数据集。使用极大数据集分页时，使用基于偏移量的分页样式可能会变得效率低下或无法使用。基于游标的分页方案具有固定时间属性，并且不会随着数据集大小的增加而减慢。

细节和限制

正确使用基于游标的分页需要稍微注意细节。你需要考虑希望将该方案应用于何种顺序。默认是按 `"-created"` 排序。这假设在模型实例上必须有一个“created”时间戳字段，并且会渲染一个“时间轴”样式分页视图，其中最近添加的 item 是第一个。

你可以通过重写分页类上的 `'ordering'` 属性或者将 `OrderingFilter` 过滤器类与 `CursorPagination` 一起使用来修改排序。与 `OrderingFilter` 一起使用时，你应该考虑限制用户可以排序的字段。

正确使用游标分页应该有一个满足以下条件的排序字段：

- 在创建时应该是一个不变的值，例如时间戳，slug，或其他只设置一次的字段。
- 应该是独特的，或几乎独一无二的。毫秒精度时间戳就是一个很好的例子。这种游标分页的实现使用了一种智能的“位置 + 偏移”风格，允许它正确地支持非严格唯一的值作为排序。
- 应该是可以强制为字符串的非空值。
- 不应该是一个 float。精度错误很容易导致错误的结果。提示：改用小数。（如果你已经有一个 float 字段并且必须对其进行分页，则可以在此处找到[使用小数来限定精度的示例](#)。）
- 该字段应该有一个数据库索引。

使用不满足这些约束条件的排序字段通常仍然有效，但是你将失去游标分页的一些好处。

Setup

要全局启用 `CursorPagination` 样式，请使用以下配置，根据需要修改 `PAGE_SIZE`：

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.CursorPagination',
3.     'PAGE_SIZE': 100
```

```
4. }
```

如果使用的是 `GenericAPIView` 子类，还可以设置 `pagination_class` 属性以基于每个视图选择 `CursorPagination`。

Configuration

`CursorPagination` 类包含一些可以被覆盖以修改分页样式的属性。

要设置这些属性，你应该继承 `CursorPagination` 类，然后像上面那样启用你的自定义分页类。

- `page_size` = 指定页面大小的数字值。如果设置，则会覆盖 `PAGE_SIZE` 设置。默认值与 `PAGE_SIZE` setting key 相同。
- `cursor_query_param` = 一个字符串值，指定“游标”查询参数的名称。默认为 `'cursor'`。
- `ordering` = 这应该是一个字符串或字符串列表，指定将应用基于游标的分页的字段。例如：`ordering = 'slug'`。默认为 `-created`。该值也可以通过在视图上使用 `OrderingFilter` 来覆盖。
- `template` = 在可浏览 API 中渲染分页控件时使用的模板的名称。可能会被覆盖以修改渲染样式，或设置为 `None` 以完全禁用 HTML 分页控件。默认为 `"rest_framework/pagination/previous_and_next.html"`。

自定义分页样式

要创建自定义分页序列化类，你应该继承 `pagination.BasePagination` 并覆盖 `paginate_queryset(self, queryset, request, view=None)` 和 `get_paginated_response(self, data)` 方法：

- `paginate_queryset` 方法被传递给初始查询集，并且应该返回一个只包含请求页面中的数据的可迭代对象。
- `get_paginated_response` 方法传递序列化的页面数据，并返回一个 `Response` 实例。

请注意，`paginate_queryset` 方法可以在分页实例上设置状态，而后 `get_paginated_response` 方法可以使用它。

举个栗子

假设我们想用一个修改后的格式替换默认的分页输出样式，该样式包含嵌套的“links” key（包含上一页，下一页链接）。我们可以像这样指定一个自定义分页类：

```
1. class CustomPagination(pagination.PageNumberPagination):
```



```

2.     def get_paginated_response(self, data):
3.         return Response({
4.             'links': {
5.                 'next': self.get_next_link(),
6.                 'previous': self.get_previous_link()
7.             },
8.             'count': self.page.paginator.count,
9.             'results': data
10.        })

```

然后我们需要在配置中设置自定义类：

```

1.  REST_FRAMEWORK = {
2.      'DEFAULT_PAGINATION_CLASS':
3.          'my_project.apps.core.pagination.CustomPagination',
4.      'PAGE_SIZE': 100
5.  }

```

请注意，如果你关心如何在可浏览的 API 中显示键的顺序，则可以在构建分页响应的主体时选择使用 `OrderedDict`，这是可选的。

使用你的自定义分页类

要默认使用你的自定义分页类，请使用 `DEFAULT_PAGINATION_CLASS` setting：

```

1.  REST_FRAMEWORK = {
2.      'DEFAULT_PAGINATION_CLASS':
3.          'my_project.apps.core.pagination.LinkHeaderPagination',
4.      'PAGE_SIZE': 100
5.  }

```

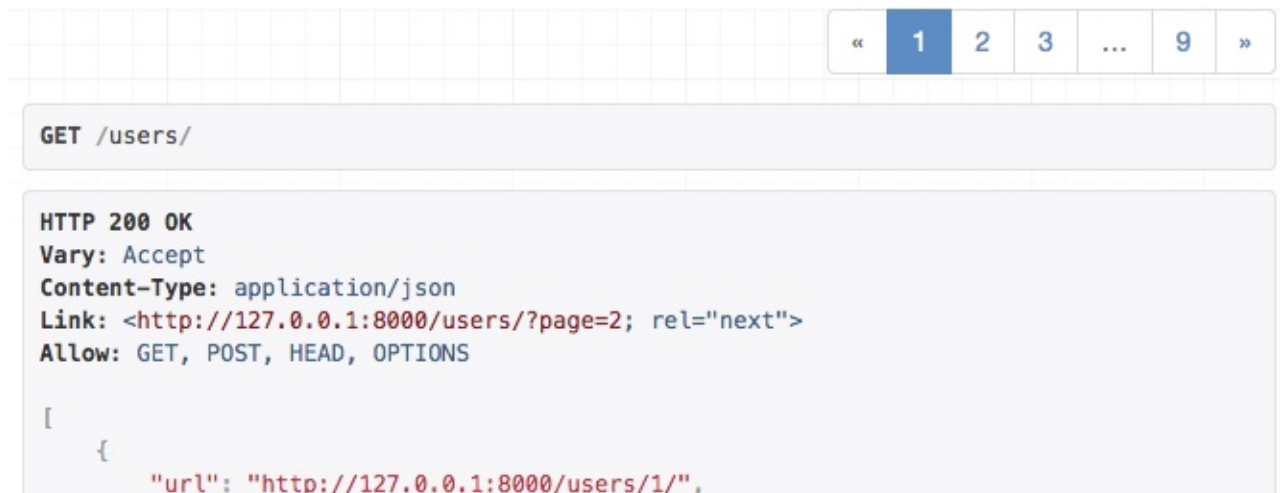
列表端点的 API 响应现在将包含一个 `Link` header，而不是将分页链接包含为响应主体的一部分。

分页和模式

通过实现 `get_schema_fields()` 方法，你还可以使分页控件可用于 REST framework 提供的模式自动生成。此方法应具有以下签名：

```
get_schema_fields(self, view)
```

该方法应该返回一个 `coreapi.Field` 实例列表。



自定义分页样式，使用 `'Link' header`

HTML 分页控件

默认情况下，使用分页类将导致 HTML 分页控件显示在可浏览的 API 中。有两种内置显示样式。

`PageNumberPagination` 和 `LimitOffsetPagination` 类显示包含上一页和下一页控件的页码列表。`CursorPagination` 类显示更简单的样式，只显示上一页和下一页控件。

自定义控件

你可以覆盖渲染 HTML 分页控件的模板。这两种内置式样是：

- `rest_framework/pagination/numbers.html`
- `rest_framework/pagination/previous_and_next.html`

在全局模板目录中提供具有这些路径的模板将覆盖相关分页类的默认渲染。

或者，你可以通过在现有类的子类上完全禁用 HTML 分页控件，将 `template=None` 设置为该类的属性。然后，你需要配置你的 `DEFAULT_PAGINATION_CLASS` setting key，以将你的自定义类用作默认分页样式。

低级 API

用于确定分页类是否应显示控件的低级 API 作为分页实例上的 `display_page_controls` 属性公开。如果需要显示 HTML 分页控件，自定义分页类应该在 `paginate_queryset` 方法中设置为 `True`。

`.to_html()` 和 `.get_html_context()` 方法也可以在自定义分页类中重写，以便进一步自定义控件的渲染方式。

版本控制

- 版本控制
 - 使用 REST framework 进行版本控制
 - 基于版本的变化行为
 - 反向解析版本化 API 的 URL
 - 版本化的 API 和超链接序列化类
- 配置版本控制方案
 - 其他版本设置
- API 参考
 - AcceptHeaderVersioning
 - 使用 accept header 和 vendor media type
- URLPathVersioning
- NamespaceVersioning
- HostNameVersioning
- QueryParameterVersioning
- 自定义版本控制方案
 - 举个栗子

[官方原文链接](#)

版本控制

API 版本控制允许你更改不同客户端之间的行为。REST framework 提供了许多不同的版本控制方案。

版本控制由传入的客户端请求决定，可能基于请求 URL 或请求 header。

有几种有效的方法来处理版本控制。非版本化的系统也可能是合适的，特别是如果你正在为超出控制范围的多个客户端的非常长期的系统进行工程设计。

使用 REST framework 进行版本控制

当启用 API 版本控制时，`request.version` 属性将包含一个对应于传入客户端请求的版本的字符串。

默认情况下，版本控制未启用，`request.version` 将始终返回 `None`。

基于版本的变化行为

如何改变 API 的行为取决于你，但你可能通常需要一个示例是在新版本中切换到不同的序列化样

式。例如：

```
1. def get_serializer_class(self):
2.     if self.request.version == 'v1':
3.         return AccountSerializerVersion1
4.     return AccountSerializer
```

反向解析版本化 API 的 URL

REST framework 包含的 `reverse` 函数与版本控制方案相关联。你需要确保将当前请求包含为关键字参数，如下所示。

```
1. from rest_framework.reverse import reverse
2.
3. reverse('bookings-list', request=request)
```

上述功能将应用适合请求版本的任何 URL 转换。例如：

- 如果正在使用 `NamespacedVersioning`，并且 API 版本为 'v1'，则使用的 URL lookup 将为 `'v1:bookings-list'`，可能会解析为像 `http://example.org/v1/bookings/` 这样的 URL。
- 如果正在使用 `QueryParameterVersioning`，并且 API 版本为 `1.0`，则返回的 URL 可能与 `http://example.org/bookings/?version=1.0` 类似

版本化的 API 和超链接序列化类

将超链接序列化样式与基于 URL 的版本控制方案一起使用时，请确保将请求作为上下文包含在序列化类中。

```
1. def get(self, request):
2.     queryset = Booking.objects.all()
3.     serializer = BookingsSerializer(queryset, many=True, context={'request':
4.         request})
5.     return Response({'all_bookings': serializer.data})
```

这样做将允许任何返回的 URL 包含合适的版本。

配置版本控制方案

版本控制方案由 `DEFAULT_VERSIONING_CLASS` setting key 定义。

```

1. REST_FRAMEWORK = {
2.     'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.NamespaceVersioning'
3. }

```

除非明确设置，否则 `DEFAULT_VERSIONING_CLASS` 的值将为 `None`。在这种情况下，`request.version` 属性将始终返回 `None`。

你还可以在单个视图上设置版本控制方案。通常，您不需要这样做，因为全局使用单个版本控制方案更有意义。如果你确实需要这样做，请使用 `versioning_class` 属性。

```

1. class ProfileList(APIView):
2.     versioning_class = versioning.QueryParameterVersioning

```

其他版本设置

以下 settings key 也用于控制版本控制：

- `DEFAULT_VERSION` . 当没有版本信息存在时，用于 `request.version` 的值。默认为 `None`。
- `ALLOWED_VERSIONS` . 如果设置，则此值将限制版本控制方案可能返回的版本集，如果提供的版本不在此集中，则会引发错误。请注意，用于 `DEFAULT_VERSION` 设置的值始终被认为是 `ALLOWED_VERSIONS` 集的一部分（除非它是 `None`）。默认为 `None`。
- `VERSION_PARAM` . 应该用于任何版本控制参数的字符串，例如媒体类型或 URL 查询参数。默认为 `'version'`。

你还可以通过定义自己的版本控制方案并使用 `default_version`，`allowed_versions` 和 `version_param` 类变量，在每个视图或每个视图集的基础上设置版本控制类以及这三个值。例如，如果你想使用 `URLPathVersioning`：

```

1. from rest_framework.versioning import URLPathVersioning
2. from rest_framework.views import APIView
3.
4. class ExampleVersioning(URLPathVersioning):
5.     default_version = ...
6.     allowed_versions = ...
7.     version_param = ...
8.
9. class ExampleView(APIView):
10.     versioning_class = ExampleVersioning

```

API 参考

AcceptHeaderVersioning

此方案要求客户端将版本指定为 `Accept` header 中媒体类型的一部分。该版本作为媒体类型参数包含在内，它补充了主要媒体类型。

这是一个使用 `accept header` 版本风格的示例 HTTP 请求。

```
1. GET /bookings/ HTTP/1.1
2. Host: example.com
3. Accept: application/json; version=1.0
```

在上面的示例请求中 `request.version` 属性将返回字符串 `'1.0'`。

基于 `Accept header` 的版本控制通常被认为是最佳实践，尽管其他样式可能更适合你的客户端需求。

使用 `accept header` 和 `vendor media type`

严格地说，`json` media type 不能被指定为包含附加参数。如果你正在构建精心指定的公共 API，则可以考虑使用 `vendor media type`。为此，请将你的渲染器配置为使用基于 JSON 的渲染器和自定义 media type：

```
1. class BookingsAPIRenderer(JSONRenderer):
2.     media_type = 'application/vnd.megacorp.bookings+json'
```

客户端请求现在看起来像这样：

```
1. GET /bookings/ HTTP/1.1
2. Host: example.com
3. Accept: application/vnd.megacorp.bookings+json; version=1.0
```

URLPathVersioning

该方案要求客户端将版本指定为 URL 路径的一部分。

```
1. GET /v1/bookings/ HTTP/1.1
2. Host: example.com
3. Accept: application/json
```

你的 URL conf 必须包含一个与 `'version'` 关键字参数相匹配的模式，以便版本控制方案可以使用此信息。

```
1. urlpatterns = [
2.     url(
3.         r'^(?P<version>(v1|v2))/bookings/$',
4.         bookings_list,
5.         name='bookings-list'
6.     ),
7.     url(
8.         r'^(?P<version>(v1|v2))/bookings/(?P<pk>[0-9]+)/$',
9.         bookings_detail,
10.        name='bookings-detail'
11.    )
12. ]
```

NamespaceVersioning

对于客户端来说，这个方案与 `URLPathVersioning` 相同。唯一的区别是，它是如何在 Django 应用程序中配置的，因为它使用 URL 命名空间而不是 URL 关键字参数。

```
1. GET /v1/something/ HTTP/1.1
2. Host: example.com
3. Accept: application/json
```

使用此方案，`request.version` 属性是根据与传入请求路径匹配的 `namespace` 确定的。

在下面的例子中，我们给出了一组视图两个不同的可能的 URL 前缀，每个前缀在不同的命名空间下：

```
1. # bookings/urls.py
2. urlpatterns = [
3.     url(r'^$', bookings_list, name='bookings-list'),
4.     url(r'^(?P<pk>[0-9]+)/$', bookings_detail, name='bookings-detail')
5. ]
6.
7. # urls.py
8. urlpatterns = [
9.     url(r'^v1/bookings/', include('bookings.urls', namespace='v1')),
10.    url(r'^v2/bookings/', include('bookings.urls', namespace='v2'))
11. ]
```


如果你只需要简单的版本控制方案，那么 `URLPathVersioning` 和 `NamespaceVersioning` 都是可以的。`URLPathVersioning` 方法可能更适合于小型项目，`NamespaceVersioning` 可能更容易管理大型项目。

HostNameVersioning

`hostname` 版本控制方案要求客户端将请求的版本指定为 URL 中 `hostname` 的一部分。

例如，以下是对 `http://v1.example.com/bookings/` URL 的 HTTP 请求：

```
1. GET /bookings/ HTTP/1.1
2. Host: v1.example.com
3. Accept: application/json
```

默认情况下，这个实现期望 `hostname` 匹配这个简单的正则表达式：

```
1. ^([a-zA-Z0-9]+\.[a-zA-Z0-9]+\.[a-zA-Z0-9]+)$
```

请注意，第一组用括号括起来，表示这是 `hostname` 的匹配部分。

`HostNameVersioning` 方案在调试模式下可能会很笨拙，因为你通常会访问诸如 `127.0.0.1` 的原始 IP 地址。

如果你有要求根据版本将传入请求路由到不同的服务器，那么基于 `hostname` 的版本控制就会特别有用，因为你可以为不同的 API 版本配置不同的 DNS 记录。

QueryParameterVersioning

该方案是一种简单的风格，其中包含版本作为 URL 中的查询参数。例如：

```
1. GET /something/?version=0.1 HTTP/1.1
2. Host: example.com
3. Accept: application/json
```

自定义版本控制方案

要实现自定义版本控制方案，请继承 `BaseVersioning` 并覆盖 `.determine_version` 方法。

举个栗子

以下示例使用自定义的 `X-API-Version` header 来确定所请求的版本。

```
1. class XAPIVersionScheme(versioning.BaseVersioning):  
2.     def determine_version(self, request, *args, **kwargs):  
3.         return request.META.get('HTTP_X_API_VERSION', None)
```

如果你的版本控制方案基于请求 URL，则还需要更改版本化 URL 的确定方式。为了做到这一点，你应该重写类的 `.reverse()` 方法。有关示例，请参阅源代码。

内容协商

- [内容协商](#)
 - [确定接受的渲染器](#)
- [自定义内容协商](#)
 - [举个栗子](#)
 - [设置内容协商](#)

[官方原文链接](#)

内容协商

内容协商是基于客户端或服务端偏好选择多种可能的表示之一以返回客户端的过程。

确定接受的渲染器

REST framework 根据可用的渲染器，每个渲染器的优先级以及客户端的 `Accept:` header，使用简单的内容协商风格来确定应将哪些媒体类型返回给客户端。所使用的风格部分由客户端驱动，部分由服务器驱动。

1. 更具体的媒体类型优先于较不特定的媒体类型。
2. 如果多种媒体类型具有相同的特性，则优先根据为给定视图配置的渲染器排序。

例如，给出以下 `Accept` header：

```
1. application/json; indent=4, application/json, application/yaml, text/html, */*
```

每种给定媒体类型的优先级为：

- `application/json; indent=4`
- `application/json` , `application/yaml` 和 `text/html`
- `*/*`

如果所请求的视图仅用 `YAML` 和 `HTML` 的渲染器配置，则 REST framework 将选择 `renderer_classes` 列表或 `DEFAULT_RENDERER_CLASSES` 设置中首先列出的渲染器。

注意：确定偏好时，REST framework 不会考虑 “q” 值。使用 “q” 值会对缓存产生负面影响，作者认为这是对内容协商的一种不必要和过于复杂的方法。

自定义内容协商

你不太可能希望为 REST framework 提供自定义内容协商方案，但如果需要，你可以这样做。要实现自定义内容协商方案，请覆盖 `BaseContentNegotiation`。

REST framework 的内容协商类处理选择适当的请求解析器和适当的响应渲染器，因此你应该实现

`.select_parser(request, parsers)` 和 `.select_renderer(request, renderers, format_suffix)` 方法。

`select_parser()` 方法应从可用解析器列表中返回一个解析器实例，如果没有任何解析器可以处理传入请求，则返回 `None`。

`select_renderer()` 方法应该返回（渲染器实例，媒体类型）的二元组，或引发 `NotAcceptable` 异常。

举个栗子

以下是自定义内容协商类，它在选择适当的解析器或渲染器时会忽略客户端请求。

```
1. from rest_framework.negotiation import BaseContentNegotiation
2.
3. class IgnoreClientContentNegotiation(BaseContentNegotiation):
4.     def select_parser(self, request, parsers):
5.         """
6.         Select the first parser in the `.parser_classes` list.
7.         """
8.         return parsers[0]
9.
10.    def select_renderer(self, request, renderers, format_suffix):
11.        """
12.        Select the first renderer in the `.renderer_classes` list.
13.        """
14.        return (renderers[0], renderers[0].media_type)
```

设置内容协商

默认内容协商类可以使用 `DEFAULT_CONTENT_NEGOTIATION_CLASS` setting 全局设置。例如，以下设置将使用我们的示例 `IgnoreClientContentNegotiation` 类。

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_CONTENT_NEGOTIATION_CLASS':
```

```
'myapp.negotiation.IgnoreClientContentNegotiation',  
3. }
```

你还可以使用 `APIView` 基于类的视图设置用于单个视图或视图集的内容协商。

```
1. from myapp.negotiation import IgnoreClientContentNegotiation  
2. from rest_framework.response import Response  
3. from rest_framework.views import APIView  
4.  
5. class NoNegotiationView(APIView):  
6.     """  
7.     An example view that does not perform content negotiation.  
8.     """  
9.     content_negotiation_class = IgnoreClientContentNegotiation  
10.  
11.     def get(self, request, format=None):  
12.         return Response({  
13.             'accepted media type': request.accepted_renderer.media_type  
14.         })
```

元数据

- [元数据](#)
 - [设置元数据 scheme](#)
 - [创建 schema 端点](#)
- [自定义元数据类](#)
 - [举个栗子](#)

[官方原文链接](#)

元数据

REST framework 包含一个可配置的机制，用于确定 API 如何响应 `OPTIONS` 请求。这使你可以返回 API schema 或其他资源信息。

对于 HTTP `OPTIONS` 请求应该返回哪种风格的响应，目前还没有任何被广泛采用的约定，所以我们提供了一种专门的风格来返回一些有用的信息。

下面是一个示例响应，演示默认返回的信息。

```
1. HTTP 200 OK
2. Allow: GET, POST, HEAD, OPTIONS
3. Content-Type: application/json
4.
5. {
6.     "name": "To Do List",
7.     "description": "List existing 'To Do' items, or create a new item.",
8.     "renders": [
9.         "application/json",
10.        "text/html"
11.    ],
12.    "pareses": [
13.        "application/json",
14.        "application/x-www-form-urlencoded",
15.        "multipart/form-data"
16.    ],
17.    "actions": {
18.        "POST": {
19.            "note": {
20.                "type": "string",
21.                "required": false,
22.                "read_only": false,
```

```

23.         "label": "title",
24.         "max_length": 100
25.     }
26. }
27. }
28. }
```

设置元数据 schema

你可以使用 `'DEFAULT_METADATA_CLASS'` settings key 全局设置元数据类：

```

1. REST_FRAMEWORK = {
2.     'DEFAULT_METADATA_CLASS': 'rest_framework.metadata.SimpleMetadata'
3. }
```

或者你可以单独设置一个视图的元数据类：

```

1. class APIRoot(APIView):
2.     metadata_class = APIRootMetadata
3.
4.     def get(self, request, format=None):
5.         return Response({
6.             ...
7.         })
```

REST framework 包只包含一个名为 `SimpleMetadata` 的元数据类实现。如果你想使用另一种风格，你需要实现一个自定义的元数据类。

创建 schema 端点

如果你对创建通过常规 GET 请求访问的 schema 端点有特定要求，则可以考虑重新使用元数据 API 来实现此目的。

例如，可以在视图集上使用以下附加路由来提供可链接的 schema 端点。

```

1. @list_route(methods=['GET'])
2. def schema(self, request):
3.     meta = self.metadata_class()
4.     data = meta.determine_metadata(request, self)
5.     return Response(data)
```

有几个原因可以选择采用这种方法，包括 `OPTIONS` 响应不能缓存。

自定义元数据类

如果你想提供一个自定义的元数据类，你应该继承 `BaseMetadata` 并且实现 `determine_metadata(self, request, view)` 方法。

你可能想要做的事情包括返回 `schema` 信息，使用 `JSON schema` 等格式，或将调试信息返回给管理员用户。

举个栗子

以下类可用于限定返回到 `OPTIONS` 请求的信息。

```
1. class MinimalMetadata(BaseMetadata):
2.     """
3.     Don't include field and other information for `OPTIONS` requests.
4.     Just return the name and description.
5.     """
6.     def determine_metadata(self, request, view):
7.         return {
8.             'name': view.get_view_name(),
9.             'description': view.get_view_description()
10.        }
```

然后配置你的设置以使用此自定义类：

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_METADATA_CLASS': 'myproject.apps.core.MinimalMetadata'
3. }
```


Schemas

- Schemas
 - 安装 Core API
 - 内部 schema 表示
 - Schema 输出格式
 - 备用 schema 格式
 - 举个栗子
 - Schemas vs 超媒体
- 创建一个 schema
 - 手动 Schema 规范
 - 自动 Schema 生成
 - 按视图 schema 自定义
- 添加 schema 视图
 - `get_schema_view` 快捷方式
 - `title`
 - `url`
 - `urlconf`
 - `renderer_classes`
 - `patterns`
 - `generator_class`
 - `authentication_classes`
 - `permission_classes`
- 使用显式 schema 视图
- 显式 schema 定义
- 静态 schema 文件
- 作为文档的 Schemas
 - 举个栗子
- API 参考
 - SchemaGenerator
 - `get_schema(self, request)`
 - `get_links(self, request)`
 - AutoSchema
 - `get_link(self, path, method, base_url)`
 - `get_description(self, path, method)`
 - `get_encoding(self, path, method)`
 - `get_path_fields(self, path, method):`
 - `get_serializer_fields(self, path, method)`
 - `get_pagination_fields(self, path, method)`

- `get_filter_fields(self, path, method)`
- `get_manual_fields(self, path, method)`
- `update_fields(fields, update_with)`
- `ManualSchema`
- `Core API`
 - `Document`
 - `title`
 - `url`
 - `content`
 - `Link`
 - `url`
 - `action`
 - `fields`
 - `description`
 - `Field`
 - `name`
 - `required`
 - `location`
 - `encoding`
 - `description`

[官方原文链接](#)

Schemas

API schema 是一个非常有用的工具，它允许一系列用例，包括生成参考文档，或者驱动可以与 API 交互的动态客户端库。

安装 Core API

你需要安装 `coreapi` 软件包才能为 REST framework 添加 schema 支持。

```
1. pip install coreapi
```

内部 schema 表示

REST framework 使用 Core API 以便以独立于格式的对 schema 信息建模。这些信息可以被渲染成各种不同的 schema 格式，或者用于生成 API 文档。

在使用 Core API 时, schema 表示为 `Document`, 它是有关 API 信息的顶级容器对象。可用的 API 交互使用 `Link` 对象表示。每个链接都包含一个 URL, HTTP 方法, 并且可能包含一个 `Field` 实例列表, 它描述了 API 端点可以接受的任何参数。 `Link` 和 `Field` 实例还可能包含描述, 允许将 API schema 渲染到用户文档中。

以下是包含单个搜索端点的 API 说明示例:

```
1. coreapi.Document(  
2.     title='Flight Search API',  
3.     url='https://api.example.org/',  
4.     content={  
5.         'search': coreapi.Link(  
6.             url='/search/',  
7.             action='get',  
8.             fields=[  
9.                 coreapi.Field(  
10.                    name='from',  
11.                    required=True,  
12.                    location='query',  
13.                    description='City name or airport code.'  
14.                ),  
15.                 coreapi.Field(  
16.                    name='to',  
17.                    required=True,  
18.                    location='query',  
19.                    description='City name or airport code.'  
20.                ),  
21.                 coreapi.Field(  
22.                    name='date',  
23.                    required=True,  
24.                    location='query',  
25.                    description='Flight date in "YYYY-MM-DD" format.'  
26.                )  
27.             ],  
28.             description='Return flight availability and prices.'  
29.         )  
30.     }  
31. )
```

Schema 输出格式

为了呈现在 HTTP 响应中，必须将内部表示渲染为响应中使用的实际字节。

Core JSON 被设计为与 Core API 一起使用的规范格式。REST framework 包含用于处理此媒体类型的渲染器类，该渲染器类可用作 `renderers.CoreJSONRenderer`。

备用 schema 格式

其他 schema 格式（如 Open API（“Swagger”），JSON HyperSchema 或 API Blueprint）也可以通过实现处理将 `Document` 实例转换为字符串表示形式的自定义渲染器类来支持。

如果有一个 Core API 编解码器包支持将编码格式化为你要使用的格式，则可以使用编解码器来实现渲染器类。

举个栗子

例如，`openapi_codec` 包提供对 Open API（“Swagger”）格式的编码或解码支持：

```
1. from rest_framework import renderers
2. from openapi_codec import OpenAPICodec
3.
4. class SwaggerRenderer(renderers.BaseRenderer):
5.     media_type = 'application/openapi+json'
6.     format = 'swagger'
7.
8.     def render(self, data, media_type=None, renderer_context=None):
9.         codec = OpenAPICodec()
10.        return codec.dump(data)
```

Schemas vs 超媒体

值得指出的是，Core API 也可以用来模拟超媒体响应，它为 API schema 提供了另一种交互风格。

通过 API schema，整个可用接口作为单个端点呈现在前端。然后，对各个 API 端点的响应通常会以普通数据的形式呈现，而不会在每个响应中包含任何进一步的交互。

使用超媒体，客户端会看到包含数据和可用交互的文档。每次交互都会生成一个新文档，详细说明当前状态和可用交互。

创建一个 schema

REST framework 包含用于自动生成 schema 的功能，或者允许你明确指定。

手动 Schema 规范

要手动指定 schema，请创建一个 Core API `Document`，与上例类似。

```
1. schema = coreapi.Document(  
2.     title='Flight Search API',  
3.     content={  
4.         ...  
5.     }  
6. )
```

自动 Schema 生成

自动 schema 生成由 `SchemaGenerator` 类提供。

`SchemaGenerator` 处理路由 URL patterns 列表并编译适当结构化的 Core API 文档。

基本用法只是为 schema 提供标题并调用 `get_schema()`：

```
1. generator = schemas.SchemaGenerator(title='Flight Search API')  
2. schema = generator.get_schema()
```

按视图 schema 自定义

默认情况下，查看自省是由可通过 `APIView` 上的 `schema` 属性访问的 `AutoSchema` 实例执行的。这为视图，请求方法和路径提供了适当的 Core API `Link` 对象：

```
1. auto_schema = view.schema  
2. coreapi_link = auto_schema.get_link(...)
```

（在编译模式时，`SchemaGenerator` 为每个视图，允许的方法和路径调用 `view.schema.get_link()`。）

注意：对于基本的 `APIView` 子类，默认自省本质上仅限于 URL kwarg 路径参数。对于包含所有基于类的视图的 `GenericAPIView` 子类，`AutoSchema` 将尝试自省序列化器，分页和过滤器字段，并提供更丰富的路径字段描述。（这里的关键钩子是相关的 `GenericAPIView` 属性和方法：`get_serializer`，`pagination_class`，`filter_backends` 等。）

要自定义 `Link` 生成, 你可以:

1) 使用 `manual_fields` kwarg 在你的视图上实例化 `AutoSchema` :

```
1. from rest_framework.views import APIView
2. from rest_framework.schemas import AutoSchema
3.
4. class CustomView(APIView):
5.     ...
6.     schema = AutoSchema(
7.         manual_fields=[
8.             coreapi.Field("extra_field", ...),
9.         ]
10.    )
```

这允许扩展最常见的情况而不需要子类化。

2) 提供具有更复杂定制的 `AutoSchema` 子类:

```
1. from rest_framework.views import APIView
2. from rest_framework.schemas import AutoSchema
3.
4. class CustomSchema(AutoSchema):
5.     def get_link(...):
6.         # Implement custom introspection here (or in other sub-methods)
7.
8. class CustomView(APIView):
9.     ...
10.    schema = CustomSchema()
```

这提供了对查看内省的完全控制。

3) 在你的视图上实例化 `ManualSchema` , 显式为视图提供 Core API `Fields` :

```
1. from rest_framework.views import APIView
2. from rest_framework.schemas import ManualSchema
3.
4. class CustomView(APIView):
5.     ...
6.     schema = ManualSchema(fields=[
7.         coreapi.Field(
8.             "first_field",
```

```

9.         required=True,
10.         location="path",
11.         schema=coreschema.String()
12.     ),
13.     coreapi.Field(
14.         "second_field",
15.         required=True,
16.         location="path",
17.         schema=coreschema.String()
18.     ),
19. ])

```

这允许手动指定某些视图的 `schema`，同时在别处保持自动生成。

通过将 `schema` 设置为 `None`，你可以禁用视图的 `schema` 生成：

```

1.     class CustomView(APIView):
2.         ...
3.         schema = None # Will not appear in schema

```

添加 schema 视图

有几种不同的方式可以将 `schema` 视图添加到你的 API 中，具体取决于你需要的内容。

get_schema_view 快捷方式

在你的项目中包含 `schema` 的最简单方法是使用 `get_schema_view()` 函数。

```

1. from rest_framework.schemas import get_schema_view
2.
3. schema_view = get_schema_view(title="Server Monitoring API")
4.
5. urlpatterns = [
6.     url('^$', schema_view),
7.     ...
8. ]

```

添加视图后，你将能够通过 API 请求来检索自动生成的 `schema` 定义。

```

1. $ http http://127.0.0.1:8000/ Accept:application/coreapi+json
2. HTTP/1.0 200 OK

```

```

3. Allow: GET, HEAD, OPTIONS
4. Content-Type: application/vnd.coreapi+json
5.
6. {
7.     "_meta": {
8.         "title": "Server Monitoring API"
9.     },
10.    "_type": "document",
11.    ...
12. }

```

`get_schema_view()` 的参数是：

`title`

可用于为 schema 定义提供描述性标题。

`url`

可用于为 schema 传递规范 URL。

```

1. schema_view = get_schema_view(
2.     title='Server Monitoring API',
3.     url='https://www.example.org/api/'
4. )

```

`urlconf`

表示要为其生成 API schema 的 URL conf 的导入路径的字符串。这默认为 Django 的 `ROOT_URLCONF` setting 的值。

```

1. schema_view = get_schema_view(
2.     title='Server Monitoring API',
3.     url='https://www.example.org/api/',
4.     urlconf='myproject.urls'
5. )

```

`renderer_classes`

可用于传递渲染 API 根端点的渲染器类列表。

```

1. from rest_framework.schemas import get_schema_view
2. from rest_framework.renderers import CoreJSONRenderer
3. from my_custom_package import APIBlueprintRenderer

```



```

4.
5. schema_view = get_schema_view(
6.     title='Server Monitoring API',
7.     url='https://www.example.org/api/',
8.     renderer_classes=[CoreJSONRenderer, APIBlueprintRenderer]
9. )

```

patterns

将 schema 内省限定为 url patterns 列表。如果你只想将 `myproject.api` url 公开在 schema 中：

```

1. schema_url_patterns = [
2.     url(r'^api/', include('myproject.api.urls')),
3. ]
4.
5. schema_view = get_schema_view(
6.     title='Server Monitoring API',
7.     url='https://www.example.org/api/',
8.     patterns=schema_url_patterns,
9. )

```

generator_class

可用于指定要传递给 `SchemaView` 的 `SchemaGenerator` 子类。

authentication_classes

可用于指定将应用于 schema 端点的认证类列表。默认为

`settings.DEFAULT_AUTHENTICATION_CLASSES`

permission_classes

可用于指定将应用于 schema 端点的权限类列表。默认为

`settings.DEFAULT_PERMISSION_CLASSES`

使用显式 schema 视图

如果你需要比 `get_schema_view()` 快捷方式更多的控制权，那么你可以直接使用 `SchemaGenerator` 类来自动生成 `Document` 实例，并从视图中返回该实例。

此选项使你可以灵活地设置 schema 端点，并使用你想要的任何行为。例如，你可以将不同的权限，限流或身份验证策略应用于 schema 端点。

以下是使用 `SchemaGenerator` 和视图一起返回 schema 的示例。

views.py:

```
1. from rest_framework.decorators import api_view, renderer_classes
2. from rest_framework import renderers, response, schemas
3.
4. generator = schemas.SchemaGenerator(title='Bookings API')
5.
6. @api_view()
7. @renderer_classes([renderers.CoreJSONRenderer])
8. def schema_view(request):
9.     schema = generator.get_schema(request)
10.    return response.Response(schema)
```

urls.py:

```
1. urlpatterns = [
2.     url('/', schema_view),
3.     ...
4. ]
```

你也可以为不同的用户提供不同的 schema，具体取决于他们拥有的权限。这种方法可以用来确保未经身份验证的请求以不同的模式呈现给已验证的请求，或者确保 API 的不同部分根据角色对不同用户可见。

为了呈现一个 schema，其中包含由用户权限过滤的端点，你需要将 `request` 参数传递给 `get_schema()` 方法，如下所示：

```
1. @api_view()
2. @renderer_classes([renderers.CoreJSONRenderer])
3. def schema_view(request):
4.     generator = schemas.SchemaGenerator(title='Bookings API')
5.     return response.Response(generator.get_schema(request=request))
```

显式 schema 定义

自动生成方法的替代方法是通过在代码库中声明 `Document` 对象来明确指定 API schema。这样做会多一点工作，但确保你完全控制 schema 表示。

```
1. import coreapi
```

```
2. from rest_framework.decorators import api_view, renderer_classes
3. from rest_framework import renderers, response
4.
5. schema = coreapi.Document(
6.     title='Bookings API',
7.     content={
8.         ...
9.     }
10. )
11.
12. @api_view()
13. @renderer_classes([renderers.CoreJSONRenderer])
14. def schema_view(request):
15.     return response.Response(schema)
```

静态 schema 文件

最后的选择是使用 Core JSON 或 Open API 等可用格式之一将你的 API schema 编写为静态文件。

然后你可以：

- 将模式定义写为静态文件，并直接提供静态文件。
- 编写一个使用 `Core API` 加载的 schema 定义，然后根据客户端请求将其渲染为多种可用格式之一。

作为文档的 Schemas

API schemas 的一个常见用法是使用它们来构建文档页面。

REST framework 中的 schema 生成使用文档字符串来自动填充 schema 文档中的描述。

这些描述将基于：

- 对应方法的文档字符串（如果存在）。
- 类文档字符串中的命名部分，可以是单行或多行。
- 类文档字符串。

举个栗子

一个 `APIView`，带有明确的方法文档字符串。

```

1. class ListUsernames(APIView):
2.     def get(self, request):
3.         """
4.         Return a list of all user names in the system.
5.         """
6.         usernames = [user.username for user in User.objects.all()]
7.         return Response(usernames)

```

一个 `ViewSet`，带有一个明确的 `action` 文档字符串。

```

1. class ListUsernames(ViewSet):
2.     def list(self, request):
3.         """
4.         Return a list of all user names in the system.
5.         """
6.         usernames = [user.username for user in User.objects.all()]
7.         return Response(usernames)

```

类文档字符串中带有 `action` 的通用视图，使用单行样式。

```

1. class UserList(generics.ListCreateAPIView):
2.     """
3.     get: List all the users.
4.     post: Create a new user.
5.     """
6.     queryset = User.objects.all()
7.     serializer_class = UserSerializer
8.     permission_classes = (IsAdminUser,)

```

使用多行样式的类文档字符串中带有 `action` 的通用视图集。

```

1. class UserViewSet(viewsets.ModelViewSet):
2.     """
3.     API endpoint that allows users to be viewed or edited.
4.
5.     retrieve:
6.     Return a user instance.
7.
8.     list:
9.     Return all users, ordered by most recently joined.
10.    """

```

```

11.     queryset = User.objects.all().order_by('-date_joined')
12.     serializer_class = UserSerializer

```

API 参考

SchemaGenerator

一个遍历路由 URL patterns 列表的类，为每个视图请求 schema 并整理生成的 CoreAPI 文档。

通常你会用一个参数实例化 `SchemaGenerator`，如下所示：

```
1. generator = SchemaGenerator(title='Stock Prices API')
```

参数：

- `title` 必需 - API 的名称。
- `url` - API schema 的 root URL。除非 schema 包含在路径前缀下，否则此选项不是必需的。
- `patterns` - 生成 schema 时要检查的 URL 列表。默认为项目的 URL conf。
- `urlconf` - 生成 schema 时使用的 URL conf 模块名称。默认为 `settings.ROOT_URLCONF`。

get_schema(self, request)

返回表示 API schema 的 `coreapi.Document` 实例。

```

1. @api_view
2. @renderer_classes([renderers.CoreJSONRenderer])
3. def schema_view(request):
4.     generator = schemas.SchemaGenerator(title='Bookings API')
5.     return Response(generator.get_schema())

```

`request` 参数是可选的，如果你希望将每个用户的权限应用于生成的 schema，则可以使用该参数。

get_links(self, request)

返回一个嵌套的字典，其中包含在 API schema 中的所有链接。

如果要修改生成的 `schema` 的结构，重写该方法很合适，因为你可以使用不同的布局构建新的字典。

AutoSchema

一个处理 `schema` 生成的个别视图内省的类。

`AutoSchema` 通过 `schema` 属性附加到 `APIView`。

`AutoSchema` 构造函数接受一个关键字参数 `manual_fields`。

`manual_fields`：将添加到生成的字段的 `coreapi.Field` 实例 `list`。具有匹配名称的生成字段将被覆盖。

```
1. class CustomView(APIView):
2.     schema = AutoSchema(manual_fields=[
3.         coreapi.Field(
4.             "my_extra_field",
5.             required=True,
6.             location="path",
7.             schema=coreschema.String()
8.         ),
9.     ])
```

对于通过继承 `AutoSchema` 来自定义 `schema` 生成。

```
1. class CustomViewSchema(AutoSchema):
2.     """
3.     Overrides `get_link()` to provide Custom Behavior X
4.     """
5.
6.     def get_link(self, path, method, base_url):
7.         link = super().get_link(path, method, base_url)
8.         # Do something to customize link here...
9.         return link
10.
11. class MyView(APIView):
12.     schema = CustomViewSchema()
```

以下方法可覆盖。

`get_link(self, path, method, base_url)`

返回与给定视图相对应的 `coreapi.Link` 实例。

这是主要的入口点。如果你需要为特定视图提供自定义行为，则可以覆盖此内容。

`get_description(self, path, method)`

返回用作链接描述的字符串。默认情况下，这基于上面的“作为文档的 Schemas” action 中描述的视图文档字符串。

`get_encoding(self, path, method)`

与给定视图交互时返回一个字符串，以指定任何请求主体的编码。例如 `'application/json'`。可能会返回一个空白字符串，以便查看不需要请求主体的视图。

`get_path_fields(self, path, method):`

返回 `coreapi.Link()` 实例列表。用于 URL 中的每个路径参数。

`get_serializer_fields(self, path, method)`

返回 `coreapi.Link()` 实例列表。用于视图使用的序列化类中的每个字段。

`get_pagination_fields(self, path, method)`

返回 `coreapi.Link()` 实例列表，该列表由 `get_schema_fields()` 方法返回给视图使用的分页类。

`get_filter_fields(self, path, method)`

返回 `coreapi.Link()` 实例列表，该列表是由视图所使用的过滤器类的 `get_schema_fields()` 方法返回的。

`get_manual_fields(self, path, method)`

返回 `coreapi.Field()` 实例列表以添加或替换生成的字段。默认为（可选）传递给 `AutoSchema` 构造函数的 `manual_fields`。

可以通过 `path` 或 `method` 覆盖自定义 manual field。例如，每个方法的调整可能如下所示：

```
1. def get_manual_fields(self, path, method):
2.     """Example adding per-method fields."""
3.
```

```

4.     extra_fields = []
5.     if method=='GET':
6.         extra_fields = # ... list of extra fields for GET ...
7.     if method=='POST':
8.         extra_fields = # ... list of extra fields for POST ...
9.
10.    manual_fields = super().get_manual_fields(path, method)
11.    return manual_fields + extra_fields

```

update_fields(fields, update_with)

实用的 `staticmethod`。封装逻辑以通过 `Field.name` 添加或替换列表中的字段。可能会被覆盖以调整替换标准。

ManualSchema

允许手动为 schema 提供 `coreapi.Field` 实例的列表，以及一个可选的描述。

```

1. class MyView(APIView):
2.     schema = ManualSchema(fields=[
3.         coreapi.Field(
4.             "first_field",
5.             required=True,
6.             location="path",
7.             schema=coreschema.String()
8.         ),
9.         coreapi.Field(
10.            "second_field",
11.            required=True,
12.            location="path",
13.            schema=coreschema.String()
14.        ),
15.    ]
16. )

```

ManualSchema 构造函数有两个参数：

fields : `coreapi.Field` 实例列表。必需。

description : 字符串描述。可选的。

Core API

本文档简要介绍了用于表示 API schema 的 `coreapi` 包内的组件。

请注意，这些类是从 `coreapi` 包导入的，而不是从 `rest_framework` 包导入的。

Document

表示 API schema 的容器。

`title`

API 的名称。

`url`

API 的规范 URL。

`content`

一个字典，包含 schema 的 `Link` 对象。

为了向 schema 提供更多结构，`content` 字典可以嵌套，通常是二层。例如：

```
1. content={
2.     "bookings": {
3.         "list": Link(...),
4.         "create": Link(...),
5.         ...
6.     },
7.     "venues": {
8.         "list": Link(...),
9.         ...
10.    },
11.    ...
12. }
```

Link

代表一个单独的 API 端点。

`url`

端点的 URL。可能是一个 URI 模板，例如 `/users/{username}/`。

action

与端点关联的 HTTP 方法。请注意，支持多个 HTTP 方法的 url 应该对应于每个 HTTP 方法的单个链接。

fields

Field 实例列表，描述输入上的可用参数。

description

对端点的含义和用途的简短描述。

Field

表示给定 API 端点上的单个输入参数。

name

输入的描述性名称。

required

boolean 值，表示客户端是否需要包含值，或者参数是否可以省略。

location

确定如何将信息编码到请求中。应该是以下字符串之一：

“path”

包含在模板化的 URI 中。例如，`/products/{product_code}/` 的 **url** 值可以与 **"path"** 字段一起使用，以处理 URL 路径中的 API 输入，例如 `/products/slim-fit-jeans/`。

这些字段通常与项目 URL conf 中的命名参数对应。

“query”

包含为 URL 查询参数。例如 `?search=sale`。通常用于 **GET** 请求。

这些字段通常与视图上的分页和过滤控件相对应。

“form”

包含在请求正文中，作为 JSON 对象或 HTML 表单的单个 item。例如 `{"colour": "blue", ...}`。通常用于 **POST**，**PUT** 和 **PATCH** 请求。多个 **"form"** 字段可能包含在单个链接上。

这些字段通常与视图上的序列化类字段相对应。

“body”

包含完整的请求主体。通常用于 `POST` , `PUT` 和 `PATCH` 请求。链接上不得存在超过一个 `"body"` 字段。不能与 `"form"` 字段一起使用。

这些字段通常对应于使用 `ListSerializer` 来验证请求输入或使用文件上传的视图。

`encoding`

“application/json”

JSON编码的请求内容。对应于使用 `JSONParser` 的视图。仅在 `Link` 中包含一个或多个 `location="form"` 字段或单个 `location="body"` 字段时有效。

“multipart/form-data”

Multipart 编码的请求内容。对应于使用 `MultiPartParser` 的视图。仅在 `Link` 中包含一个或多个 `location="form"` 字段时有效。

“application/x-www-form-urlencoded”

URL encode 的请求内容。对应于使用 `FormParser` 的视图。仅在 `Link` 中包含一个或多个 `location="form"` 字段时有效。

“application/octet-stream”

二进制上传请求内容。对应于使用 `FileUploadParser` 的视图。仅在 `Link` 中包含 `location="body"` 字段时有效。

`description`

对输入字段的含义和用途的简短描述。

Format 后缀

- [Format 后缀](#)
 - [format_suffix_patterns](#)
 - 与 `i18n_patterns` 一起使用
 - [查询参数 format](#)

[官方原文链接](#)

Format 后缀

Web API 的常见模式是在 URL 上使用文件扩展名来为给定的媒体类型提供端点。例如，`'http://example.com/api/users.json'` 用于提供 JSON 表示。

在 URLconf 中为你的 API 添加 `format-suffix` 模式是容易出错和非 DRY 的，因此 REST framework 提供了将这些模式添加到 URLconf 的快捷方式。

format_suffix_patterns

签名: `format_suffix_patterns(urlpatterns, suffix_required=False, allowed=None)`

返回一个 URL pattern 列表，其中包含附加到每个 URL pattern 的格式后缀模式。

参数:

- **urlpatterns:** 必需。一个 URL pattern 列表。
- **suffix_required:** 可选。一个 boolean 值，指定 URL 中的后缀是否可选或强制。默认为 `False`，这意味着后缀默认是可选的。
- **allowed:** 可选。有效格式后缀的列表或元组。如果没有提供，将使用通配符格式后缀模式。

例如:

```
1. from rest_framework.urlpatterns import format_suffix_patterns
2. from blog import views
3.
4. urlpatterns = [
5.     url(r'^$', views.appt_root),
6.     url(r'^comments/$', views.comment_list),
7.     url(r'^comments/(?P<pk>[0-9]+)/$', views.comment_detail)
8. ]
9.
10. urlpatterns = format_suffix_patterns(urlpatterns, allowed=['json', 'html'])
```

在使用 `format_suffix_patterns` 时，你必须确保将 `'format'` 关键字参数添加到相应的视图。例如：

```
1. @api_view(('GET', 'POST'))
2. def comment_list(request, format=None):
3.     # do stuff...
```

或者基于类视图：

```
1. class CommentList(APIView):
2.     def get(self, request, format=None):
3.         # do stuff...
4.
5.     def post(self, request, format=None):
6.         # do stuff...
```

所使用的 `kwargs` 的名称可以使用 `FORMAT_SUFFIX_KWARG` setting 进行修改。

另请注意，`format_suffix_patterns` 不支持降序包含 URL patterns。

与 `i18n_patterns` 一起使用

如果使用 Django 提供的 `i18n_patterns` 函数以及 `format_suffix_patterns`，则应确保将 `i18n_patterns` 函数用作最终或最外层函数。例如：

```
1. url_patterns = [
2.     ...
3. ]
4.
5. urlpatterns = i18n_patterns(
6.     format_suffix_patterns(urlpatterns, allowed=['json', 'html'])
7. )
```

查询参数 `format`

格式后缀的替代方法是将请求的 `format` 包含在查询参数中。REST framework 默认提供此选项，并且它在可浏览的 API 中用于在不同的可用表示之间切换。

要使用其短格式表示，请使用 `format` 查询参数。例如：

`http://example.com/organizations/?format=csv`。

此查询参数的名称可以使用 `URL_FORMAT_OVERRIDE` 设置进行修改。将该值设置为 `None` 以禁用此行为。

返回 URL

- [返回 URL](#)
 - [reverse](#)
 - [reverse_lazy](#)

[官方原文链接](#)

返回 URL

通常，从 Web API（例如 `http://example.com/foobar`）返回绝对 URI 可能是更好的做法，而不是返回相对 URI，例如 `/foobar`。

这样做的好处有：

- 它更明确。
- 它为你的 API 客户端留下更少的工作。
- 当字符串在诸如 JSON 这样的表示中没有本地 URI 类型时，它的含义是没有歧义的。
- 它使得使用超链接标记 HTML 表示等事情变得很容易。

REST framework 提供了两个实用函数，可以更简单地从 Web API 返回绝对 URI。

使用它们不是必须的，但是如果你这样做，自描述 API 将能够自动为你输出超链接，这使得浏览 API 变得更容易。

reverse

签名： `reverse(viewname, *args, **kwargs)`

具有与 `django.urls.reverse` 相同的行为，除了它返回一个完全限定的 URL，使用 request 来确定主机和端口。

你应该将 request 作为关键字参数包含在该函数中，例如：

```
1. from rest_framework.reverse import reverse
2. from rest_framework.views import APIView
3. from django.utils.timezone import now
4.
5. class APIRootView(APIView):
6.     def get(self, request):
7.         year = now().year
8.         data = {
```

```
9.         ...
10.         'year-summary-url': reverse('year-summary', args=[year],
    request=request)
11.     }
12.     return Response(data)
```

reverse_lazy

签名: `reverse_lazy(viewname, *args, **kwargs)`

具有与 `django.urls.reverse_lazy` 相同的行为，除了它返回一个完全限定的 URL，使用 `request` 来确定主机和端口。

与 `reverse` 函数一样，你应该将 `request` 作为关键字参数包含在函数中，例如：

```
1. api_root = reverse_lazy('api-root', request=request)
```


异常

- [异常](#)
 - [REST framework 视图中的异常处理](#)
 - [自定义异常处理](#)
- [API 参考](#)
 - [APIException](#)
 - [检查 API 异常](#)
- [ParseError](#)
- [AuthenticationFailed](#)
- [NotAuthenticated](#)
- [PermissionDenied](#)
- [NotFound](#)
- [MethodNotAllowed](#)
- [NotAcceptable](#)
- [UnsupportedMediaType](#)
- [Throttled](#)
- [ValidationError](#)

[官方原文链接](#)

异常

REST framework 视图中的异常处理

REST framework 的视图处理各种异常，并返回适当的错误响应。

需要处理的异常情况有：

- 在 REST framework 内引发的 `APIException` 的子类。
- Django 的 `Http404` 异常。
- Django 的 `PermissionDenied` 异常。

在每种情况下，REST framework 都会返回一个带有适当状态码和内容类型的响应。响应的主体将包含有关错误性质的其他细节。

大多数错误响应将包含响应正文中的关键 `detail`。

例如，以下请求：

```
1. DELETE http://api.example.com/foo/bar HTTP/1.1
```

```
2. Accept: application/json
```

可能会收到错误响应，指出在该资源上不允许使用 `DELETE` 方法：

```
1. HTTP/1.1 405 Method Not Allowed
2. Content-Type: application/json
3. Content-Length: 42
4.
5. {"detail": "Method 'DELETE' not allowed."}
```

验证错误的处理方式稍有不同，但都是将字段名称作为响应中的关键字。如果验证错误不是特定于某个字段的，那么它将使用 “non_field_errors” 键，或者为 `NON_FIELD_ERRORS_KEY` setting 设置的字符串值。

示例验证错误可能如下所示：

```
1. HTTP/1.1 400 Bad Request
2. Content-Type: application/json
3. Content-Length: 94
4.
5. {"amount": ["A valid integer is required."], "description": ["This field may not be blank."]}
```

自定义异常处理

你可以通过创建处理函数来实现自定义异常处理，该函数将 API 视图中引发的异常转换为响应对象。这使你可以控制 API 错误响应的样式。

该函数必须带有一对参数，第一个是要处理的异常，第二个是包含任何额外上下文（例如当前正在处理的视图）的字典。异常处理函数应该返回一个 `Response` 对象，或者如果无法处理异常，则返回 `None`。如果处理程序返回 `None`，那么异常将被重新抛出，Django 将返回一个标准的 HTTP 500 ‘server error’ 响应。

例如，你可能希望确保所有错误响应都包含响应正文中的 HTTP 状态码，如下所示：

```
1. HTTP/1.1 405 Method Not Allowed
2. Content-Type: application/json
3. Content-Length: 62
4.
5. {"status_code": 405, "detail": "Method 'DELETE' not allowed."}
```

为了改变响应的风格，你可以编写下面的自定义异常处理程序：

```
1. from rest_framework.views import exception_handler
2.
3. def custom_exception_handler(exc, context):
4.     # Call REST framework's default exception handler first,
5.     # to get the standard error response.
6.     response = exception_handler(exc, context)
7.
8.     # Now add the HTTP status code to the response.
9.     if response is not None:
10.         response.data['status_code'] = response.status_code
11.
12.     return response
```

`context` 参数不被默认处理程序使用，但是如果异常处理程序需要更多信息，例如当前正在处理的视图（可以作为 `context['view']` 访问），则该参数可能很有用。

异常处理程序还必须使用 `EXCEPTION_HANDLER` setting key 在你的设置中进行配置。例如：

```
1. REST_FRAMEWORK = {
2.     'EXCEPTION_HANDLER': 'my_project.my_app.utils.custom_exception_handler'
3. }
```

如果未指定，则 `'EXCEPTION_HANDLER'` setting 默认为由 REST framework 提供的标准异常处理程序：

```
1. REST_FRAMEWORK = {
2.     'EXCEPTION_HANDLER': 'rest_framework.views.exception_handler'
3. }
```

请注意，异常处理程序只会根据由异常产生的响应调用。它不会用于视图直接返回的任何响应，例如在序列化验证失败时通用视图返回的 `HTTP_400_BAD_REQUEST` 响应。

API 参考

APIException

签名： `APIException()`

在 `APIView` 类或 `@api_view` 中引发的所有异常的基类。

要自定义异常，请继承 `APIException`，并在该类上设置 `.status_code`，`.default_detail` 和 `default_code` 属性。

例如，如果你的 API 依赖于可能无法访问的第三方服务，则可能需要为 “503 Service Unavailable” HTTP 响应码封装异常。你可以这样做：

```
1. from rest_framework.exceptions import APIException
2.
3. class ServiceUnavailable(APIException):
4.     status_code = 503
5.     default_detail = 'Service temporarily unavailable, try again later.'
6.     default_code = 'service_unavailable'
```

检查 API 异常

有许多不同的属性可用于检查 API 异常的状态。你可以使用它们为你的项目构建自定义异常处理。

可用的属性和方法有：

- `.detail` - 返回错误的文本描述。
- `.get_codes()` - 返回错误的代码标识符。
- `.get_full_details()` - 返回文本描述和代码标识符。

在大多数情况下，错误详情将是一个简单的 item：

```
1. >>> print(exc.detail)
2. You do not have permission to perform this action.
3. >>> print(exc.get_codes())
4. permission_denied
5. >>> print(exc.get_full_details())
6. {'message': 'You do not have permission to perform this
    action.', 'code': 'permission_denied'}
```

在验证错误的情况下，错误详情将是 item 列表或字典：

```
1. >>> print(exc.detail)
2. {"name": "This field is required.", "age": "A valid integer is required."}
3. >>> print(exc.get_codes())
4. {"name": "required", "age": "invalid"}
5. >>> print(exc.get_full_details())
6. {"name": {"message": "This field is required.", "code": "required"}, "age":
```

```
{"message": "A valid integer is required.", "code": "invalid"}}
```

ParseError

签名: `ParseError(detail=None, code=None)`

在访问 `request.data` 时包含格式错误的数据则会引发此异常。

默认情况下，此异常会导致 HTTP 状态码 “400 Bad Request” 的响应。

AuthenticationFailed

签名: `AuthenticationFailed(detail=None, code=None)`

当传入的请求包含不正确的身份验证时引发。

默认情况下，此异常会导致 HTTP 状态码 “401 Unauthenticated” 的响应，但也可能会导致 “403 Forbidden” 响应，具体取决于所使用的身份验证方案。

NotAuthenticated

签名: `NotAuthenticated(detail=None, code=None)`

当未经身份验证的请求未通过权限检查时引发。

默认情况下，此异常会导致 HTTP 状态码 “401 Unauthenticated” 的响应，但也可能会导致 “403 Forbidden” 响应，具体取决于所使用的身份验证方案。

PermissionDenied

签名: `PermissionDenied(detail=None, code=None)`

当经过身份验证的请求未通过权限检查时引发。

默认情况下，此异常会导致 HTTP 状态码 “403 Forbidden” 的响应。

NotFound

签名: `NotFound(detail=None, code=None)`

当资源不存在于给定的 URL 时引发。这个异常相当于标准的 `Http404` Django 异常。

默认情况下，此异常会导致 HTTP 状态码为 “404 Not Found” 的响应。

MethodNotAllowed

签名: `MethodNotAllowed(method, detail=None, code=None)`

当请求发生时，找不到视图上对应的处理方法时引发。

默认情况下，此异常会导致 HTTP 状态码为 “405 Method Not Allowed” 的响应。

NotAcceptable

签名: `NotAcceptable(detail=None, code=None)`

当请求发生时，任何可用渲染器都不符合 `Accept` header 时引发。

默认情况下，此异常会导致 HTTP 状态码为 “406 Not Acceptable” 的响应。

UnsupportedMediaType

签名: `UnsupportedMediaType(media_type, detail=None, code=None)`

如果在访问 `request.data` 时没有可以处理请求数据的内容类型的解析器，就会引发。

默认情况下，此异常会导致 HTTP 状态码 “415 Unsupported Media Type” 的响应。

Throttled

签名: `Throttled(wait=None, detail=None, code=None)`

传入的请求未通过限流检查时引发。

默认情况下，此异常会导致 HTTP 状态码 “429 Too Many Requests” 的响应。

ValidationError

签名: `ValidationError(detail, code=None)`

`ValidationError` 异常与其他 `APIException` 类略有不同：

- `detail` 参数是必需的，不是可选的。
- `detail` 参数可以是错误详情列表或字典，也可以是嵌套的数据结构。

- 按照惯例，你应该导入 `serializers` 模块并使用完全限定的 `ValidationError` 样式，以区别于 Django 内置的验证错误。例如：

```
raise serializers.ValidationError('This field must be an integer value.')
```

`ValidationError` 类应该用于序列化类和字段验证以及验证器类。使用 `raise_exception` 关键字参数调用 `serializer.is_valid` 时也会引发此问题：

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

通用视图使用 `raise_exception=True` 标志，意味着你可以在 API 中全局覆盖验证错误响应的样式。为此，请使用自定义异常处理程序，如上所述。

默认情况下，此异常会导致 HTTP 状态码 “400 Bad Request” 的响应。

状态码

- 状态码
 - 信息 - 1xx
 - 成功 - 2xx
 - 重定向 - 3xx
 - 客户端错误 - 4xx
 - 服务器错误 - 5xx
 - 帮助函数

[官方原文链接](#)

状态码

不建议在你的响应中使用裸露（直接使用数字）的状态码。 REST framework 包含一组命名常量，你可以使用它们使代码更加清晰易读。

```
1. from rest_framework import status
2. from rest_framework.response import Response
3.
4. def empty_view(self):
5.     content = {'please move along': 'nothing to see here'}
6.     return Response(content, status=status.HTTP_404_NOT_FOUND)
```

下面列出了 `status` 模块中包含的全部 HTTP 状态码。

该模块还包含一组帮助函数，用于测试状态码是否在给定范围内。

```
1. from rest_framework import status
2. from rest_framework.test import APITestCase
3.
4. class ExampleTestCase(APITestCase):
5.     def test_url_root(self):
6.         url = reverse('index')
7.         response = self.client.get(url)
8.         self.assertTrue(status.is_success(response.status_code))
```

信息 - 1xx

这类状态码表示临时响应。默认情况下，REST framework 中没有使用 1xx 状态码。

1. HTTP_100_CONTINUE
2. HTTP_101_SWITCHING_PROTOCOLS

成功 - 2xx

这类状态码表明客户端的请求已被成功接收，理解和接受。

1. HTTP_200_OK
2. HTTP_201_CREATED
3. HTTP_202_ACCEPTED
4. HTTP_203_NON_AUTHORITATIVE_INFORMATION
5. HTTP_204_NO_CONTENT
6. HTTP_205_RESET_CONTENT
7. HTTP_206_PARTIAL_CONTENT
8. HTTP_207_MULTI_STATUS

重定向 - 3xx

这类状态码表明用户代理需要采取进一步行动来完成请求。

1. HTTP_300_MULTIPLE_CHOICES
2. HTTP_301_MOVED_PERMANENTLY
3. HTTP_302_FOUND
4. HTTP_303_SEE_OTHER
5. HTTP_304_NOT_MODIFIED
6. HTTP_305_USE_PROXY
7. HTTP_306_RESERVED
8. HTTP_307_TEMPORARY_REDIRECT

客户端错误 - 4xx

4xx 状态码适用于客户端出现错误的情况。除了响应 HEAD 请求之外，服务器应该包含一个错误情况说明的实体，以及它是临时的还是永久的。

1. HTTP_400_BAD_REQUEST
2. HTTP_401_UNAUTHORIZED
3. HTTP_402_PAYMENT_REQUIRED
4. HTTP_403_FORBIDDEN
5. HTTP_404_NOT_FOUND

6. HTTP_405_METHOD_NOT_ALLOWED
7. HTTP_406_NOT_ACCEPTABLE
8. HTTP_407_PROXY_AUTHENTICATION_REQUIRED
9. HTTP_408_REQUEST_TIMEOUT
10. HTTP_409_CONFLICT
11. HTTP_410_GONE
12. HTTP_411_LENGTH_REQUIRED
13. HTTP_412_PRECONDITION_FAILED
14. HTTP_413_REQUEST_ENTITY_TOO_LARGE
15. HTTP_414_REQUEST_URI_TOO_LONG
16. HTTP_415_UNSUPPORTED_MEDIA_TYPE
17. HTTP_416_REQUESTED_RANGE_NOT_SATISFIABLE
18. HTTP_417_EXPECTATION_FAILED
19. HTTP_422_UNPROCESSABLE_ENTITY
20. HTTP_423_LOCKED
21. HTTP_424_FAILED_DEPENDENCY
22. HTTP_428_PRECONDITION_REQUIRED
23. HTTP_429_TOO_MANY_REQUESTS
24. HTTP_431_REQUEST_HEADER_FIELDS_TOO_LARGE
25. HTTP_451_UNAVAILABLE_FOR_LEGAL_REASONS

服务器错误 - 5xx

5xx 表示服务器意识到其发生错误或无法执行请求的情况。除了响应 HEAD 请求之外，服务器应该包含一个错误情况说明的实体，以及它是临时的还是永久的。

1. HTTP_500_INTERNAL_SERVER_ERROR
2. HTTP_501_NOT_IMPLEMENTED
3. HTTP_502_BAD_GATEWAY
4. HTTP_503_SERVICE_UNAVAILABLE
5. HTTP_504_GATEWAY_TIMEOUT
6. HTTP_505_HTTP_VERSION_NOT_SUPPORTED
7. HTTP_507_INSUFFICIENT_STORAGE
8. HTTP_511_NETWORK_AUTHENTICATION_REQUIRED

帮助函数

以下帮助函数可用于识别响应代码的类别。

1. `is_informational()` # 1xx

```
2. is_success()      # 2xx
3. is_redirect()     # 3xx
4. is_client_error() # 4xx
5. is_server_error() # 5xx
```

测试

- 测试
- APIRequestFactory
 - 创建测试请求
 - 使用 `format` 参数
 - 显式编码请求主体
 - PUT 和 PATCH 与表单数据
- 强制认证
- 强制 CSRF 验证
- APIClient
 - 发出请求
 - 认证
 - `.login(**kwargs)`
 - `.credentials(**kwargs)`
 - `.force_authenticate(user=None, token=None)`
- CSRF 验证
- RequestsClient
 - `RequestsClient` 与数据库一起工作
 - Headers & 身份验证
 - CSRF
 - Live tests
- CoreAPIClient
 - Headers & 身份验证
- API Test cases
 - 举个栗子
- URLPatternsTestCase
 - 例如
- 测试响应
 - 检查响应数据
 - 渲染响应
- 配置
 - 设置默认格式
 - 设置可用的格式

官方原文链接

测试

REST framework 包含一些扩展 Django 现有测试框架的助手类，并改进对 API 请求的支持。

APIRequestFactory

扩展了 Django 现有的 `RequestFactory` 类。

创建测试请求

`APIRequestFactory` 类支持与 Django 的标准 `RequestFactory` 类几乎完全相同的 API。这意味着标准的 `.get()`，`.post()`，`.put()`，`.patch()`，`.delete()`，`.head()` 和 `.options()` 方法都可用。

```
1. from rest_framework.test import APIRequestFactory
2.
3. # Using the standard RequestFactory API to create a form POST request
4. factory = APIRequestFactory()
5. request = factory.post('/notes/', {'title': 'new idea'})
```

使用 `format` 参数

创建请求主体（如 `post`，`put` 和 `patch`）的方法包括 `format` 参数，这使得使用除 `multipart` 表单数据以外的内容类型生成请求变得容易。例如：

```
1. # Create a JSON POST request
2. factory = APIRequestFactory()
3. request = factory.post('/notes/', {'title': 'new idea'}, format='json')
```

默认情况下，可用的格式是 `'multipart'` 和 `'json'`。为了与 Django 现有的 `RequestFactory` 兼容，默认格式是 `'multipart'`。

显式编码请求主体

如果你需要显式编码请求正文，则可以通过设置 `content_type` 标志来完成。例如：

```
1. request = factory.post('/notes/', json.dumps({'title': 'new idea'}),
    content_type='application/json')
```

PUT 和 PATCH 与表单数据

Django 的 `RequestFactory` 和 REST framework 的 `APIRequestFactory` 之间值得注意的一个区别是 `multipart` 表单数据将被编码为除 `.post()` 以外的方法。

例如，使用 `APIRequestFactory`，你可以像这样做一个表单 PUT 请求：

```
1. factory = APIRequestFactory()
2. request = factory.put('/notes/547/', {'title': 'remember to email dave'})
```

使用 Django 的 `RequestFactory`，你需要自己显式编码数据：

```
1. from django.test.client import encode_multipart, RequestFactory
2.
3. factory = RequestFactory()
4. data = {'title': 'remember to email dave'}
5. content = encode_multipart('BoUnDaRyStRiNg', data)
6. content_type = 'multipart/form-data; boundary=BoUnDaRyStRiNg'
7. request = factory.put('/notes/547/', content, content_type=content_type)
```

强制认证

当使用请求工厂直接测试视图时，能够直接验证请求通常很方便，而不必构造正确的验证凭证。

要强制验证请求，请使用 `force_authenticate()` 方法。

```
1. from rest_framework.test import force_authenticate
2.
3. factory = APIRequestFactory()
4. user = User.objects.get(username='olivia')
5. view = AccountDetail.as_view()
6.
7. # Make an authenticated request to the view...
8. request = factory.get('/accounts/django-superstars/')
9. force_authenticate(request, user=user)
10. response = view(request)
```

该方法的签名是 `force_authenticate(request, user=None, token=None)`。调用时，可以设置 `user` 和 `token` 中的任一个或两个。

例如，当使用令牌强行进行身份验证时，你可能会执行以下操作：

```
1. user = User.objects.get(username='olivia')
2. request = factory.get('/accounts/django-superstars/')
3. force_authenticate(request, user=user, token=user.auth_token)
```

注意： `force_authenticate` 直接将 `request.user` 设置为内存中的 `user` 实例。如果跨多个测试重新使用同一个 `user` 实例来更新已保存的 `user` 状态，则可能需要在测试之间调用 `refresh_from_db()`。

注意：使用 `APIRequestFactory` 时，返回的对象是 Django 的标准 `HttpRequest`，而不是 REST framework 的 `Request` 对象，只有在调用视图后才会生成该对象。

这意味着直接在请求对象上设置属性可能并不总是有你期望的效果。例如，直接设置 `.token` 将不起作用，并且仅在使用会话身份验证时直接设置 `.user` 才会起作用。

```
1. # Request will only authenticate if `SessionAuthentication` is in use.
2. request = factory.get('/accounts/django-superstars/')
3. request.user = user
4. response = view(request)
```

强制 CSRF 验证

默认情况下，使用 `APIRequestFactory` 创建的请求在传递给 REST framework 视图时不会应用 CSRF 验证。如果你需要明确打开 CSRF 验证，则可以通过在实例化工厂时设置 `enforce_csrf_checks` 标志来实现。

```
1. factory = APIRequestFactory(enforce_csrf_checks=True)
```

注意：值得注意的是，Django 的标准 `RequestFactory` 不需要包含这个选项，因为当使用常规的 Django 时，CSRF 验证发生在中间件中，当直接测试视图时该中间件不运行。在使用 REST framework 时，CSRF 验证发生在视图内，因此请求工厂需要禁用视图级 CSRF 检查。

APIClient

扩展了 Django 现有的 `Client` 类。

发出请求

`APIClient` 类支持与 Django 标准 `Client` 类相同的请求接口。这意味着标准的 `.get()`，`.post()`，`.put()`，`.patch()`，`.delete()`，`.head()` 和 `.options()` 方法都可用。例如：

```

1. from rest_framework.test import APIClient
2.
3. client = APIClient()
4. client.post('/notes/', {'title': 'new idea'}, format='json')

```

认证

.login(**kwargs)

`login` 方法的功能与 Django 的常规 `Client` 类一样。这使你可以对任何包含 `SessionAuthentication` 的视图进行身份验证。

```

1. # Make all requests in the context of a logged in session.
2. client = APIClient()
3. client.login(username='lauren', password='secret')

```

要登出，请照常调用 `logout` 方法。

```

1. # Log out
2. client.logout()

```

`login` 方法适用于测试使用会话认证的 API，例如包含 AJAX 与 API 交互的网站。

.credentials(**kwargs)

`credentials` 方法可用于设置 header，这些 header 将包含在测试客户端的所有后续请求中。

```

1. from rest_framework.authtoken.models import Token
2. from rest_framework.test import APIClient
3.
4. # Include an appropriate `Authorization:` header on all requests.
5. token = Token.objects.get(user__username='lauren')
6. client = APIClient()
7. client.credentials(HTTP_AUTHORIZATION='Token ' + token.key)

```

请注意，第二次调用 `credentials` 会覆盖任何现有凭证。你可以通过调用没有参数的方法来取消任何现有的凭证。

```

1. # Stop including any credentials
2. client.credentials()

```


`credentials` 方法适用于测试需要验证 header 的 API，例如 basic 验证，OAuth1 和 OAuth2 验证以及简单令牌验证方案。

`.force_authenticate(user=None, token=None)`

有时你可能想完全绕过认证，强制测试客户端的所有请求被自动视为已认证。

如果你正在测试 API 但是不想构建有效的身份验证凭据以进行测试请求，则这可能是一个有用的捷径。

```
1. user = User.objects.get(username='lauren')
2. client = APIClient()
3. client.force_authenticate(user=user)
```

要对后续请求进行身份验证，请调用 `force_authenticate` 将 `user` 和(或) `token` 设置为 `None`。

```
1. client.force_authenticate(user=None)
```

CSRF 验证

默认情况下，使用 `APIClient` 时不应用 CSRF 验证。如果你需要明确启用 CSRF 验证，则可以通过在实例化客户端时设置 `enforce_csrf_checks` 标志来实现。

```
1. client = APIClient(enforce_csrf_checks=True)
```

像往常一样，CSRF 验证将仅适用于任何会话验证视图。这意味着 CSRF 验证只有在客户端通过调用 `login()` 登录后才会发生。

RequestsClient

REST framework 还包含一个客户端，用于使用流行的 Python 库 `requests` 与应用程序进行交互。这可能是有用的，如果：

- 你期望主要从另一个 Python 服务与 API 进行交互，并且希望在与客户端相同的级别测试该服务。
- 你希望以这样的方式编写测试，以便它们也可以在分段或实时环境中运行。

它暴露了与直接使用请求会话完全相同的接口。

```
1. client = RequestsClient()
2. response = client.get('http://testserver/users/')
3. assert response.status_code == 200
```

请注意，requests client 要求你传递完全限定的 URL。

RequestsClient

与数据库一起工作

如果你想编写仅与服务接口交互的测试，则 `RequestsClient` 类很有用。这比使用标准的 Django 测试客户端要严格一些，因为这意味着所有的交互必须通过 API。

如果你使用的是 `RequestsClient`，你需要确保测试设置和结果断言以常规 API 调用的方式执行，而不是直接与数据库模型进行交互。例如，不是检查 `Customer.objects.count() == 3`，而是列出 customers 端点，并确保它包含三条记录。

Headers & 身份验证

自定义 header 和身份验证凭证的提供方式与使用标准 `requests.Session` 实例时的方式相同。

```
1. from requests.auth import HTTPBasicAuth
2.
3. client.auth = HTTPBasicAuth('user', 'pass')
4. client.headers.update({'x-test': 'true'})
```

CSRF

如果你使用 `SessionAuthentication`，那么你需要为 `POST`，`PUT`，`PATCH` 或 `DELETE` 请求包含一个 CSRF 令牌。

你可以通过遵循基于 JavaScript 的客户端使用的相同流程来实现。首先进行 `GET` 请求以获取 CSRF 令牌，然后在以下请求中呈现该令牌。

例如...

```
1. client = RequestsClient()
2.
3. # Obtain a CSRF token.
4. response = client.get('/homepage/')
5. assert response.status_code == 200
6. csrftoken = response.cookies['csrftoken']
```

```
7.  
8. # Interact with the API.  
9. response = client.post('/organisations/', json={  
10.     'name': 'MegaCorp',  
11.     'status': 'active'  
12. }, headers={'X-CSRFToken': csrftoken})  
13. assert response.status_code == 200
```

Live tests

使用 `RequestsClient` 和 `CoreAPIClient` 可以编写在开发环境中运行的测试用例，也可以直接根据测试服务器或生产环境运行测试用例。

使用这种风格来创建几个核心功能的基本测试是验证你的实时服务的有效方法。这样做可能需要仔细注意安装和卸载 (setup and teardown)，以确保测试的运行方式不会直接影响客户数据。

CoreAPIClient

CoreAPIClient 允许你使用 Python `coreapi` 客户端库与你的 API 进行交互。

```
1. # Fetch the API schema  
2. client = CoreAPIClient()  
3. schema = client.get('http://testserver/schema/')  
4.  
5. # Create a new organisation  
6. params = {'name': 'MegaCorp', 'status': 'active'}  
7. client.action(schema, ['organisations', 'create'], params)  
8.  
9. # Ensure that the organisation exists in the listing  
10. data = client.action(schema, ['organisations', 'list'])  
11. assert(len(data) == 1)  
12. assert(data == [{'name': 'MegaCorp', 'status': 'active'}])
```

Headers & 身份验证

自定义 header 和身份验证可以与 `RequestsClient` 类似的方式和 `CoreAPIClient` 一起使用。

```
1. from requests.auth import HTTPBasicAuth
```

```
2.
3. client = CoreAPIClient()
4. client.session.auth = HTTPBasicAuth('user', 'pass')
5. client.session.headers.update({'x-test': 'true'})
```

API Test cases

REST framework 包含以下测试用例类，它们类似现有的 Django 测试用例类，但使用 `APIClient` 而不是 Django 的默认 `Client`。

- `APISimpleTestCase`
- `APITransactionTestCase`
- `APITestCase`
- `APILiveServerTestCase`

举个栗子

你可以像使用常规 Django 测试用例类一样使用任何 REST framework 的测试用例类。
`self.client` 属性将是一个 `APIClient` 实例。

```
1. from django.urls import reverse
2. from rest_framework import status
3. from rest_framework.test import APITestCase
4. from myproject.apps.core.models import Account
5.
6. class AccountTests(APITestCase):
7.     def test_create_account(self):
8.         """
9.         Ensure we can create a new account object.
10.        """
11.        url = reverse('account-list')
12.        data = {'name': 'DabApps'}
13.        response = self.client.post(url, data, format='json')
14.        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
15.        self.assertEqual(Account.objects.count(), 1)
16.        self.assertEqual(Account.objects.get().name, 'DabApps')
```

URLPatternsTestCase

REST framework 还提供了一个用于隔离每个类的 `urlpatterns` 的测试用例类。请注意，它继承自 Django 的 `SimpleTestCase`，并且很可能需要与另一个测试用例类混合使用。

例如

```
1. from django.urls import include, path, reverse
2. from rest_framework.test import APITestCase, URLPatternsTestCase
3.
4.
5. class AccountTests(APITestCase, URLPatternsTestCase):
6.     urlpatterns = [
7.         path('api/', include('api.urls')),
8.     ]
9.
10.    def test_create_account(self):
11.        """
12.        Ensure we can create a new account object.
13.        """
14.        url = reverse('account-list')
15.        response = self.client.get(url, format='json')
16.        self.assertEqual(response.status_code, status.HTTP_200_OK)
17.        self.assertEqual(len(response.data), 1)
```

测试响应

检查响应数据

在检查测试响应的有效性时，检查响应的创建数据通常比较方便，而不是检查完全渲染的响应。

例如，检查 `response.data` 更容易：

```
1. response = self.client.get('/users/4/')
2. self.assertEqual(response.data, {'id': 4, 'username': 'lauren'})
```

而不是检查解析 `response.content` 的结果：

```
1. response = self.client.get('/users/4/')
2. self.assertEqual(json.loads(response.content), {'id': 4, 'username': 'lauren'})
```

渲染响应

如果你使用 `APIRequestFactory` 直接测试视图，则返回的响应将不会渲染，因为模板响应的渲染由 Django 的内部请求 - 响应循环执行。为了访问 `response.content`，你首先需要渲染响应。

```
1. view = UserDetails.as_view()
2. request = factory.get('/users/4')
3. response = view(request, pk='4')
4. response.render() # Cannot access `response.content` without this.
5. self.assertEqual(response.content, '{"username": "lauren", "id": 4}')
```

配置

设置默认格式

用于创建测试请求的默认格式可以使用 `TEST_REQUEST_DEFAULT_FORMAT` setting key 进行设置。例如，默认情况下总是对测试请求使用 JSON 而不是标准的 multipart 表单请求，请在 `settings.py` 文件中设置以下内容：

```
1. REST_FRAMEWORK = {
2.     ...
3.     'TEST_REQUEST_DEFAULT_FORMAT': 'json'
4. }
```

设置可用的格式

如果你需要使用除 multipart 或 json 请求之外的其他方法来测试请求，则可以通过设置 `TEST_REQUEST_RENDERER_CLASSES` setting 来完成。

例如，要在测试请求中添加对 `format = 'html'` 的支持，您可能在 `settings.py` 文件中有这样的内容。

```
1. REST_FRAMEWORK = {
2.     ...
3.     'TEST_REQUEST_RENDERER_CLASSES': (
4.         'rest_framework.renderers.MultiPartRenderer',
5.         'rest_framework.renderers.JSONRenderer',
6.         'rest_framework.renderers.TemplateHTMLRenderer'
```

测试

7.)

8. }

Settings

- [Settings](#)
 - [访问 settings](#)
- [API 参考](#)
 - [API 策略设置](#)
 - [DEFAULT_RENDERER_CLASSES](#)
 - [DEFAULT_PARSER_CLASSES](#)
 - [DEFAULT_AUTHENTICATION_CLASSES](#)
 - [DEFAULT_PERMISSION_CLASSES](#)
 - [DEFAULT_THROTTLE_CLASSES](#)
 - [DEFAULT_CONTENT_NEGOTIATION_CLASS](#)
 - [DEFAULT_SCHEMA_CLASS](#)
- [Generic view settings](#)
 - [DEFAULT_FILTER_BACKENDS](#)
 - [PAGE_SIZE](#)
- [SEARCH_PARAM](#)
 - [ORDERING_PARAM](#)
- [版本控制设置](#)
 - [DEFAULT_VERSION](#)
 - [ALLOWED_VERSIONS](#)
 - [VERSION_PARAM](#)
- [认证设置](#)
 - [UNAUTHENTICATED_USER](#)
 - [UNAUTHENTICATED_TOKEN](#)
- [测试设置](#)
 - [TEST_REQUEST_DEFAULT_FORMAT](#)
 - [TEST_REQUEST_RENDERER_CLASSES](#)
- [Schema 生成控制](#)
 - [SCHEMA_COERCE_PATH_PK](#)
 - [SCHEMA_COERCE_METHOD_NAMES](#)
- [内容类型控制](#)
 - [URL_FORMAT_OVERRIDE](#)
 - [FORMAT_SUFFIX_KWARG](#)
- [日期和时间格式](#)
 - [DATETIME_FORMAT](#)
 - [DATETIME_INPUT_FORMATS](#)
 - [DATE_FORMAT](#)
 - [DATE_INPUT_FORMATS](#)

- `TIME_FORMAT`
- `TIME_INPUT_FORMATS`
- 编码
 - `UNICODE_JSON`
 - `COMPACT_JSON`
 - `STRICT_JSON`
 - `COERCE_DECIMAL_TO_STRING`
- View names and descriptions
 - `VIEW_NAME_FUNCTION`
 - `VIEW_DESCRIPTION_FUNCTION`
- HTML Select 字段截取
 - `HTML_SELECT_CUTOFF`
 - `HTML_SELECT_CUTOFF_TEXT`
- 杂项设置
 - `EXCEPTION_HANDLER`
 - `NON_FIELD_ERRORS_KEY`
 - `URL_FIELD_NAME`
 - `NUM_PROXIES`

[官方原文链接](#)

Settings

Namespaces are one honking great idea - let's do more of those!

– [The Zen of Python][cite]

REST framework 的配置是在名为 `REST_FRAMEWORK` 的单个 Django 设置中的所有命名空间。

例如，你的项目的 `settings.py` 文件可能包含如下所示的内容：

```
1. REST_FRAMEWORK = {
2.     'DEFAULT_RENDERER_CLASSES': (
3.         'rest_framework.renderers.JSONRenderer',
4.     ),
5.     'DEFAULT_PARSER_CLASSES': (
6.         'rest_framework.parsers.JSONParser',
7.     )
8. }
```

访问 settings

如果你需要访问项目中 REST framework 的 API 设置值，则应使用 `api_settings` 对象。例如。

```
1. from rest_framework.settings import api_settings
2.
3. print api_settings.DEFAULT_AUTHENTICATION_CLASSES
```

`api_settings` 对象将检查用户定义的设置，否则将回退到默认值。任何使用字符串导入路径引用类的设置都会自动导入并返回被引用的类，而不是字符串文字。

API 参考

API 策略设置

以下设置控制基本 API 策略，并应用于每个基于 `APIView` 类的视图或基于 `@api_view` 函数的视图。

DEFAULT_RENDERER_CLASSES

渲染器类的列表或元组，用于确定返回 `Response` 对象时可能使用的默认渲染器集。

默认值：

```
1. (
2.     'rest_framework.renderers.JSONRenderer',
3.     'rest_framework.renderers.BrowsableAPIRenderer',
4. )
```

DEFAULT_PARSER_CLASSES

解析器类的列表或元组，用于确定访问 `request.data` 属性时使用的默认解析器集。

默认值：

```
1. (
2.     'rest_framework.parsers.JSONParser',
3.     'rest_framework.parsers.FormParser',
4.     'rest_framework.parsers.MultiPartParser'
5. )
```

DEFAULT_AUTHENTICATION_CLASSES

身份验证类的列表或元组，用于确定在访问 `request.user` 或 `request.auth` 属性时使用的默认身份验证集。

默认值：

```
1. (
2.     'rest_framework.authentication.SessionAuthentication',
3.     'rest_framework.authentication.BasicAuthentication'
4. )
```

DEFAULT_PERMISSION_CLASSES

权限类的列表或元组，用于确定在视图开始时检查的默认权限集。权限必须由列表中的每个类授予。

默认值：

```
1. (
2.     'rest_framework.permissions.AllowAny',
3. )
```

DEFAULT_THROTTLE_CLASSES

限流类的列表或元组，用于确定在视图开始时检查的默认限流类集。

默认值： `()`

DEFAULT_CONTENT_NEGOTIATION_CLASS

内容协商类，用于确定如何为响应选择渲染器，并给定传入请求。

默认值： `'rest_framework.negotiation.DefaultContentNegotiation'`

DEFAULT_SCHEMA_CLASS

将用于 `schema` 生成的视图检查类。

默认值： `'rest_framework.schemas.AutoSchema'`

Generic view settings

以下设置控制通用基于类的视图的行为。

DEFAULT_FILTER_BACKENDS

用于通用过滤的过滤器后端类列表。如果设置为 `None`，则禁用通用过滤。

PAGE_SIZE

用于分页的默认页面大小。如果设置为 `None`，则默认情况下禁用分页。

默认值: `None`

SEARCH_PARAM

查询参数的名称，可用于指定 `SearchFilter` 使用的搜索词。

默认值: `search`

ORDERING_PARAM

排序参数的名称，可用于指定 `OrderingFilter` 返回结果的排序。

默认值: `ordering`

版本控制设置

DEFAULT_VERSION

当没有版本信息存在时，默认的 `request.version` 值。

默认值: `None`

ALLOWED_VERSIONS

如果设置，则此值将限制版本控制方案可能返回的版本集合，如果提供的版本不在此集合中，则会引发错误。

默认值: `None`

VERSION_PARAM

用于版本控制参数的字符串，例如媒体类型或 URL 查询参数。

默认值: `'version'`

认证设置

以下设置控制未经身份验证的请求的行为。

UNAUTHENTICATED_USER

用于初始化未经身份验证的请求的 `request.user` 的类。（如果要完全删除验证，可以通过从 `INSTALLED_APPS` 中除去 `django.contrib.auth`，将 `UNAUTHENTICATED_USER` 设置为 `None`。）

默认值: `django.contrib.auth.models.AnonymousUser`

UNAUTHENTICATED_TOKEN

用于初始化未经身份验证的请求的 `request.auth` 的类。

默认值: `None`

测试设置

以下设置控制 `APIRequestFactory` 和 `APIClient` 的行为

TEST_REQUEST_DEFAULT_FORMAT

进行测试请求时应使用的默认格式。

这应该与 `TEST_REQUEST_RENDERER_CLASSES` 设置中的其中一个渲染器类的格式相匹配。

默认值: `'multipart'`

TEST_REQUEST_RENDERER_CLASSES

构建测试请求时支持的渲染器类。

构建测试请求时可以使用任何这些渲染器类的格式，例如: `client.post('/users', {'username': 'jamie'}, format='json')`

默认值:

```
1. (
2.     'rest_framework.renderers.MultiPartRenderer',
3.     'rest_framework.renderers.JSONRenderer'
4. )
```

Schema 生成控制

SCHEMA_COERCE_PATH_PK

如果设置，则在生成 schema 路径参数时，会将 URL conf 中的 `'pk'` 标识符映射到实际字段名称上。通常这将是 `'id'`。由于“primary key”是实现细节，因此这提供了更适合的表示，而“identifier”是更一般的概念。

默认值: `True`

SCHEMA_COERCE_METHOD_NAMES

如果设置，则用于将内部视图方法名称映射到 schema 生成中使用的外部操作名称。这使我们能够生成比代码库中内部使用的名称更适合外部表示的名称。

默认值: `{'retrieve': 'read', 'destroy': 'delete'}`

内容类型控制

URL_FORMAT_OVERRIDE

通过在请求 URL 中使用 `format=...` 查询参数，可用于覆盖默认内容协商 `Accept` header 行为的 URL 参数的名称。

例如: `http://example.com/organizations/?format=csv`

如果此设置的值为 `None`，则 URL 格式覆盖将被禁用。

默认值: `'format'`

FORMAT_SUFFIX_KWARG

URL conf 中用于提供格式后缀的参数名称。使用 `format_suffix_patterns` 包含后缀 URL patterns 时应用此设置。

例如: `http://example.com/organizations.csv/`

默认值: `'format'`

日期和时间格式

以下设置用于控制如何分析和渲染日期和时间表示。

DATETIME_FORMAT

格式字符串，默认情况下用于渲染 `DateTimeField` 序列化字段的输出。如果为 `None`，那么 `DateTimeField` 序列化字段将返回 Python `datetime` 对象，并且日期时间编码将由渲染器决定。

可以是 `None`，`'iso-8601'` 或 Python `strftime` 格式字符串中的任何一个。

默认值: `'iso-8601'`

DATETIME_INPUT_FORMATS

默认使用的格式字符串列表，用于解析 `DateTimeField` 序列化字段的输入。

可以是包含字符串 `'iso-8601'` 或 Python `strftime` 格式字符串的列表。

默认值: `['iso-8601']`

DATE_FORMAT

格式字符串，默认情况下用于渲染 `DateField` 序列化字段的输出。如果为 `None`，那么 `DateField` 序列化字段将返回 Python `date` 对象，并且日期编码将由渲染器决定。

可以是 `None`，`'iso-8601'` 或 Python `strftime` 格式字符串中的任何一个。

默认值: `'iso-8601'`

DATE_INPUT_FORMATS

默认使用的格式字符串列表，用于解析 `DateField` 序列化字段的输入。

可以是包含字符串 `'iso-8601'` 或 Python `strftime` 格式字符串的列表。

默认值: `['iso-8601']`

TIME_FORMAT

格式字符串，默认情况下用于渲染 `TimeField` 序列化字段的输出。如果为 `None`，那么 `TimeField` 序列化字段将返回 Python `time` 对象，并且时间编码将由渲染器决定。

可以是 `None`，`'iso-8601'` 或 Python `strftime` 格式字符串中的任何一个。

默认值: `'iso-8601'`

TIME_INPUT_FORMATS

默认使用的格式字符串列表，用于解析 `TimeField` 序列化字段的输入。

可以是包含字符串 `'iso-8601'` 或 Python `strftime` 格式字符串的列表。

默认值: `['iso-8601']`

编码

UNICODE_JSON

设置为 `True` 时，JSON 响应将允许 unicode 字符作为响应。例如：

```
1. {"unicode black star": "★"}
```

当设置为 `False` 时，JSON 响应将转义非ascii字符，如下所示：

```
1. {"unicode black star": "\u2605"}
```

两种样式都符合 RFC 4627，并且在语法上都是有效 JSON。在检查 API 响应时，unicode 样式更受用户欢迎。

默认值: `True`

COMPACT_JSON

当设置为 `True` 时，JSON 响应将返回紧凑表示，`':'` 和 `','` 字符之后没有间隔。例如：

```
1. {"is_admin":false,"email":"jane@example"}
```

当设置为 `False` 时，JSON 响应将返回稍微更冗长的表示，如下所示：

```
1. {"is_admin": false, "email": "jane@example"}
```

默认样式是返回缩小的响应，符合 Heroku 的 API 设计准则。

默认值: `True`

STRICT_JSON

当设置为 `True` 时，JSON 渲染和解析只会遵循语法上有效的 JSON，而 Python 的 `json` 模块接受的扩展浮点值（`nan`，`inf`，`-inf`）会引发异常。这是推荐的设置，因为通常不支

持这些值。例如，Javascript 的 `JSON.Parse` 和 PostgreSQL 的 JSON 数据类型都不接受这些值。

当设置为 `False` 时，JSON 的渲染和解析将是宽松的。但是，这些值仍然无效，需要在代码中专门处理。

默认值: `True`

COERCE_DECIMAL_TO_STRING

在不支持原生十进制类型的 API 表示形式中返回小数对象时，通常最好将该值作为字符串返回。这样可以避免二进制浮点实现所带来的精度损失。

设置为 `True` 时，序列化 `DecimalField` 类将返回字符串而不是 `Decimal` 对象。设置为 `False` 时，序列化将返回 `Decimal` 对象，默认 JSON 编码器将作为浮点数返回。

默认值: `True`

View names and descriptions

以下设置用于生成视图名称和描述，如 `OPTIONS` 请求的响应中所使用的，以及可浏览 API 中使用的设置。

VIEW_NAME_FUNCTION

表示生成视图名称时应使用的函数的字符串。

这应该是一个具有以下签名的函数：

```
1. view_name(cls, suffix=None)
```

- `cls`：视图类。通常，名称函数会在生成描述性名称时通过访问 `cls.__name__` 来检查类的名称。
- `suffix`：区分视图中各个视图时使用的可选后缀。

默认值: `'rest_framework.views.get_view_name'`

VIEW_DESCRIPTION_FUNCTION

表示生成视图描述时应使用的函数的字符串。

此设置可以更改为支持除默认 markdown 以外的标记样式。例如，你可以使用它支持在可浏览的 API 中输出的视图文档字符串中的 `rst` 标记。

这应该是一个具有以下签名的函数：

```
1. view_description(cls, html=False)
```

- `cls`：视图类。通常，`description` 函数会在生成描述时检查类的文档字符串，方法是访问 `cls.__doc__`
- `html`：指示是否需要 HTML 输出的布尔值。在可浏览的 API 中使用时为 `true`，在生成 `OPTIONS` 响应时使用 `False`。

默认值： `'rest_framework.views.get_view_description'`

HTML Select 字段截取

用于在可浏览 API 中渲染关系字段的 `select` 字段截取的全局设置。

HTML_SELECT_CUTOFF

`html_cutoff` 值的全局设置。必须是整数。

默认值： 1000

HTML_SELECT_CUTOFF_TEXT

代表 `html_cutoff_text` 全局设置的字符串。

默认值： `"More than {count} items..."`

杂项设置

EXCEPTION_HANDLER

一个字符串，表示在返回任何给定异常的响应时应使用的函数。如果该函数返回 `None`，则会引发 500 错误。

此设置可以更改为支持默认 `{"detail": "Failure..."}` 响应以外的错误响应。例如，你可以使用它来提供 API 响应，如 `{"errors": [{"message": "Failure...", "code": ""} ...]}`。

这应该是一个具有以下签名的函数：

```
1. exception_handler(exc, context)
```

- `exc`：异常。

默认值: `'rest_framework.views.exception_handler'`

NON_FIELD_ERRORS_KEY

表示应该用于序列化错误的关键字的字符串，该字符串不引用特定字段，而代表一般错误。

默认值: `'non_field_errors'`

URL_FIELD_NAME

一个字符串，表示用于由 `HyperlinkedModelSerializer` 生成的 URL 字段的键

默认值: `'url'`

NUM_PROXIES

一个 0 或更大的整数，可用于指定 API 在后台运行的应用程序代理的数量。这允许限流器更准确地识别客户端 IP 地址。如果设置为 `None`，那么限流器将使用宽松的 IP 匹配方式。

默认值: `None`