# Request和responses

## Request

### Data

Request继承HttpRequest,提供更灵活的请求解析，核心功能是

Request.data属性，类似request.POST

request.POST # Only handles form data. Only works for 'POST' method.

request.data # Handles arbitrary data. Works for 'POST', 'PUT' and 'PATCH' methods.

支持解析post/put/patch方法的数据

### Parser

有一些内置的parser类，允许接受不同media types的请求。一个视图的parser通过parser类列表定义，当获取request.data时，rest\_framework将检测request的Content-type头并决定使用哪个parser解析request内容。

#### 设置parser

**全局默认parser这样设置：**

REST\_FRAMEWORK = {

'DEFAULT\_PARSER\_CLASSES': [

'rest\_framework.parsers.JSONParser',

]

}

类视图设置parser

from rest\_framework.parsers import JSONParser  
from rest\_framework.response import Response  
from rest\_framework.views import APIView  
class ExampleView(APIView):  
 *"""  
 A view that can accept POST requests with JSON content.  
 """* parser\_classes = [JSONParser]  
 def post(self, request, format=None):  
 return Response({'received data': request.data})

方法视图设置parser

@api\_view(['POST'])  
@parser\_classes([JSONParser])  
def example\_view(request, format=None):  
 *"""  
 A view that can accept POST requests with JSON content.  
 """* return Response({'received data': request.data})

### query\_params

request.GET的替代品，推荐中这个，因为不仅Get有参数

## Response

 a Response object, which is a type of TemplateResponse that takes unrendered content and uses content negotiation to determine the correct content type to return to the client.

return Response(data) # Renders to content type as requested by the client.

Response类可以根据客户端request把数据渲染成不同内容类型。是Django的SimpleTemplateResponse的子类。也可以返回HttpResponse或者SimpleTemplateResponse对象，Response类仅仅是提供更优雅的接口返回基于内容协商的WEB api response，它可以渲染成多种格式。

### 创建Response

Response(data, status=None, template\_name=None, headers=None, content\_type=None)

和HttpResponse不同，不需要使用渲染后的内容初始化Response。Response不能处理类似django模型实例的复杂类型，因此，需要把数据序列化成原生类型，可以使用Serializer类实现。

* data: The serialized data for the response.
* status: A status code for the response. Defaults to 200. See also [status codes](https://www.django-rest-framework.org/api-guide/status-codes/).
* template\_name: A template name to use if HTMLRenderer is selected.
* headers: A dictionary of HTTP headers to use in the response.
* content\_type: The content type of the response. Typically, this will be set automatically by the renderer as determined by content negotiation, but there may be some cases where you need to specify the content type explicitly.

### 属性

data

status\_code

content: 渲染后的数据

标准HttpResponse属性等

render()

## views

APIView类是Django View类的子类，但有不同：

1. 传递给处理方法的是Request实例而不是Django的HttpRequest实例
2. 处理方法返回Response实例，而不是HttpResponse实例

## 状态码

such as HTTP\_400\_BAD\_REQUEST in the status module. It's a good idea to use these throughout rather than using numeric identifiers.

## 包装views

1. The @api\_view decorator for working with function based views.
2. The APIView class for working with class-based views.

These wrappers provide a few bits of functionality such as making sure you receive Requestinstances in your view, and adding context to Response objects so that content negotiation can be performed.

The wrappers also provide behaviour such as returning 405 Method Not Allowed responses when appropriate, and handling any ParseError exception that occurs when accessing request.datawith malformed input.

## 添加可选的url后缀

视图函数添加可选参数format=None,然后追加一些format\_suffix\_patterns

路由urlpatterns = format\_suffix\_patterns(urlpatterns)。客户端则可以指定返回数据的格式：

http http://127.0.0.1:8000/snippets/ Accept:application/json # Request JSON

http http://127.0.0.1:8000/snippets/ Accept:text/html # Request HTML

或者

http http://127.0.0.1:8000/snippets.json # JSON suffix

http http://127.0.0.1:8000/snippets.api # Browsable API suffix

也可以使用Content-Type控制请求数据格式

# POST using form data

http --form POST http://127.0.0.1:8000/snippets/ code="print(123)"

{

"id": 3,

"title": "",

"code": "print(123)",

"linenos": false,

"language": "python",

"style": "friendly"

}

# POST using JSON

http --json POST http://127.0.0.1:8000/snippets/ code="print(456)"

{

"id": 4,

"title": "",

"code": "print(456)",

"linenos": false,

"language": "python",

"style": "friendly"

}

# 基于类的视图

更利于重用代码，把各http方法分开

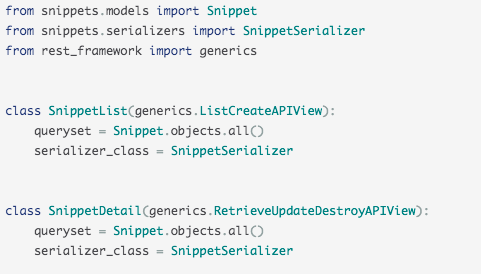
## 使用mixins

One of the big wins of using class-based views is that it allows us to easily compose reusable bits of behaviour.

The create/retrieve/update/delete operations that we've been using so far are going to be pretty similar for any model-backed API views we create. Those bits of common behaviour are implemented in REST framework's mixin classes.

## 范型views

REST framework provides a set of already mixed-in generic views that we can use



# 序列化器

<https://www.django-rest-framework.org/api-guide/serializers/#serializers>

序列化器可以把复杂的数据，例如querysets和模型实例或者非模型类转为python类型，然后渲染成json/xml或其他类型。也提供反序列化器。

也可以批量处理

## 非model序列化

serializers.py

from rest\_framework import serializers

class YourSerializer(serializers.Serializer):

"""Your data serializer, define your fields here."""

comments = serializers.IntegerField()

likes = serializers.IntegerField()

views.py

from rest\_framework import views

from rest\_framework.response import Response

from .serializers import YourSerializer

class YourView(views.APIView):

def get(self, request):

yourdata= [{"likes": 10, "comments": 0}, {"likes": 4, "comments": 23}]

results = YourSerializer(yourdata, many=True).data

return Response(results)

# 测试

# 参考

<https://www.cnblogs.com/yangxt90/articles/8746825.html>

<http://www.zijin.net/news/tech/1193942.html>

# 源码剖析视频

<https://study.163.com/course/courseMain.htm?courseId=1006514027&_trace_c_p_k2_=8b6733a395d1428d928a1d3f689e8ac9>

# 工具

## 返回json

### 使用django-simple-serializer

<https://www.jianshu.com/p/2d60bf3faf37>

pip install django-simple-serializer

from django.shortcuts import render  
from rest\_framework.views import APIView  
from dss.Serializer import serializer  
from django.http import HttpResponse, HttpRequest  
# Create your views here.  
def response\_as\_json(data, foreign\_penetrate=False):  
 jsonString = serializer(data=data, output\_type="json", foreign=foreign\_penetrate)  
 response = HttpResponse(  
 # json.dumps(dataa, cls=MyEncoder),  
 jsonString,  
 content\_type="application/json",  
 )  
 response["Access-Control-Allow-Origin"] = "\*"  
 return response  
  
  
def json\_response(data, code=200, foreign\_penetrate=False, \*\*kwargs):  
 data = {  
 "code": code,  
 "msg": "成功",  
 "data": data,  
 }  
 return response\_as\_json(data, foreign\_penetrate=foreign\_penetrate)  
  
  
def json\_error(error\_string="", code=500, \*\*kwargs):  
 data = {  
 "code": code,  
 "msg": error\_string,  
 "data": {}  
 }  
 data.update(kwargs)  
 return response\_as\_json(data)  
  
JsonResponse = json\_response  
JsonError = json\_error  
class ReturnJson(APIView):  
  
 def get(self, request, \*args, \*\*kwargs):  
 return JsonResponse("Hello world!!!!!!!!++++++中文测试")

urls.py

from django.conf.urls import url  
from django.contrib import admin  
from rest\_framework.schemas import get\_schema\_view  
from mytest.views import ReturnJson  
  
  
urlpatterns = [  
 url(r'^admin/', admin.site.urls),  
 url(r'^docs/', get\_schema\_view()),  
 url(r'^api/getjson', ReturnJson.as\_view()),  
]

### 使用drf-json-api

<https://github.com/kevin-brown/drf-json-api> main repository.

<https://django-rest-framework-json-api.readthedocs.io/en/stable/>

# 视图

fbv/cbv

# 教程

<https://segmentfault.com/a/1190000004401112>

# django补充

## 路由

### path()方法

路由系统中最重要的path()方法可以接收4个参数，其中2个是必须的：route和view，以及2个可选的参数：kwargs和name

route:路由正则表达式，需要注意的是，route不会匹配 GET 和 POST 参数或域名。例如，URLconf 在处理请求 https://www.example.com/myapp/时，它会尝试匹配 myapp/。处理请求 https://www.example.com/myapp/?page=3 时，也只会尝试匹配 myapp/。

使用正则表达式匹配路径参数，格式：<类型:参数名>，例如'articles/<int:year>/<int:month>/ 会捕获两个int类型的参数，django默认参数类型是str，有如下内置转换器：

* str：匹配任何非空字符串，但不含斜杠/，如果你没有专门指定转换器，那么这个是默认使用的；
* int：匹配0和正整数，返回一个int类型
* slug：可理解为注释、后缀、附属等概念，是url拖在最后的一部分解释性字符。该转换器匹配任何ASCII字符以及连接符和下划线，比如building-your-1st-django-site；
* uuid：匹配一个uuid格式的对象。为了防止冲突，规定必须使用破折号，所有字母必须小写，例如075194d3-6885-417e-a8a8-6c931e272f00。返回一个UUID对象；
* path：匹配任何非空字符串，重点是可以包含路径分隔符’/‘。这个转换器可以帮助你匹配整个url而不是一段一段的url字符串。要区分path转换器和path()方法。

View: view指的是处理当前url请求的视图函数。当Django匹配到某个路由条目时，自动将封装的HttpRequest对象作为第一个参数，被“捕获”的参数以关键字参数的形式，传递给该条目指定的视图view。

kwargs：任意数量的关键字参数可以作为一个字典传递给目标视图。

name对你的URL进行命名，让你能够在Django的任意处，尤其是模板内显式地引用它。这是一个非常强大的功能，相当于给URL取了个全局变量名，不会将url匹配地址写死。

### django如何处理请求

1. 决定要使用的根URLconf模块。通常，这是ROOT\_URLCONF设置的值，但是如果传入的HttpRequest对象具有urlconf属性（由中间件设置），则其值将被用于代替ROOT\_URLCONF设置。通俗的讲，就是你可以自定义项目入口url是哪个文件！
2. 加载该模块并寻找可用的urlpatterns。 它是django.urls.path()或者django.urls.re\_path()实例的一个列表。
3. 依次匹配每个URL模式，在与请求的URL相匹配的第一个模式停下来。也就是说，url匹配是从上往下的短路操作，所以url在列表中的位置非常关键。
4. 导入并调用匹配行中给定的视图，该视图是一个简单的Python函数（被称为视图函数）,或基于类的视图。 视图将获得如下参数:
5. 一个HttpRequest 实例。
6. 如果匹配的表达式返回了未命名的组，那么匹配的内容将作为位置参数提供给视图。
7. 关键字参数由表达式匹配的命名组组成，但是可以被django.urls.path()的可选参数kwargs覆盖。
8. 如果没有匹配到任何表达式，或者过程中抛出异常，将调用一个适当的错误处理视图。

### 路由转发

使用include()方法， 会去掉URL中匹配的部分，把剩下的字符串转发给include的URLConf做进一步处理，同时也会把匹配到的参数传递给视图。

#### 传递参数给视图

path('blog/<int:year>/', views.year\_archive, {'foo': 'bar'}),

#### 传递参数给include

**如果被include的URLconf的所有视图都需要某个参数，则可以直接传递参数给include：**

**方法一：**

path('blog/', include('inner'), {'blog\_id': 3}),

方法二：

在被include的urlconf中写

path('archive/', views.archive, {'blog\_id': 3}),

### 反向解析和命名空间

反向解析，其实就是给URl起一个名字，代码中可以通过这个名字获得这个url，这样即使url路径修改了，代码中也无需修改。

Python代码中使用reverse()函数可以获得指定名字的url地址。

但是如果项目中有很多命名url，很可能出现重名的情况，所以出现了命名空间，

通过在app的urls.py开头增加app\_name=’xx’指定URLname命名空间

## 自定义错误页面

Django默认的自带的错误视图包括400、403、404和500，分别表示请求错误、拒绝服务、页面不存在和服务器错误

handler400 —— django.conf.urls.handler400。

handler403 —— django.conf.urls.handler403。

handler404 —— django.conf.urls.handler404。

handler500 —— django.conf.urls.handler500。

这些值可以在根URLconf中设置。在其它app中的二级URLconf中设置这些变量无效。

urlpatterns = [

path('admin/', admin.site.urls),

]

# 增加的条目

handler400 = views.bad\_request

handler403 = views.permission\_denied

handler404 = views.page\_not\_found

handler500 = views.error

## 视图函数及快捷方式

返回html网页return HttpResponse(htmlstr)

返回错误码return HttpResponse(status=201)

### 快捷方式

Django.shortcuts模块中提供了快捷类和方法

#### render

render(request, template\_name, context=None, content\_type=None, status=None, using=None)[source]

结合一个给定的模板和一个给定的上下文字典，返回一个渲染后的HttpResponse对象。

必需参数：

request：视图函数处理的当前请求，封装了请求头的所有数据，其实就是视图参数request。

template\_name：要使用的模板的完整名称或者模板名称的列表。如果是一个列表，将使用其中能够查找到的第一个模板。

可选参数：

context：添加到模板上下文的一个数据字典。默认是一个空字典。可以将认可需要提供给模板的数据以字典的格式添加进去。这里有个小技巧，使用Python内置的locals()方法，可以方便的将函数作用于内的所有变量一次性添加。

content\_type：用于生成的文档的MIME类型。 默认为DEFAULT\_CONTENT\_TYPE设置的值。

status：响应的状态代码。 默认为200。

using：用于加载模板使用的模板引擎的NAME。

#### render\_to\_response()

render\_to\_response(template\_name, context=None, content\_type=None, status=None, using=None)[source]

此功能在引入render()之前进行，不推荐，以后可能会被弃用

#### redirect()

redirect(to, permanent=False, *args,*\*kwargs)[source]

根据传递进来的url参数，返回HttpResponseRedirect。

参数to可以是：

* 一个模型：将调用模型的get\_absolute\_url()函数，反向解析出目的url；
* 视图名称：可能带有参数：reverse()将用于反向解析url；
* 一个绝对的或相对的URL：将原封不动的作为重定向的目标位置。

默认情况下是临时重定向，如果设置permanent=True将永久重定向。

## 请求和响应

### QueryDict对象

在HttpRequest对象中，GET和POST属性都是一个django.http.QueryDict的实例。

## 常用命令

### migrate

python manage.py migrate

migrate命令将遍历INSTALLED\_APPS设置中的所有项目，在数据库中创建对应的表，并打印出每一条动作信息

### makemigrations

python manage.py makemigrations polls

通过运行makemigrations命令，Django 会检测你对模型文件的修改，也就是告诉Django你对模型有改动，并且你想把这些改动保存为一个“迁移(migration)”。

migrations是Django保存模型修改记录的文件，这些文件保存在磁盘上

python manage.py sqlmigrate polls 0001

可以查看SQL语句

### migrate

在数据库中执行建表操作

## 管理站点

给管理员使用的后台，提供model的增删改查，

创建：python manage.py createsuperuser

把app注册到admin中，就是让admin管理该app的model

在app的admin.py中添加：

admin.site.register(你的model)

## 模板语言

{{}} == print

## 模型

### 元数据

指的是“除了字段外的所有内容”，例如排序方式、数据库表名、人类可读的单数或者复数名等等

class Ox(models.Model):

horn\_length = models.IntegerField()

class Meta: # 注意，是模型的子类，要缩进！

ordering = ["horn\_length"]

verbose\_name\_plural = "oxen"

abstract :抽象模型

db\_table:指定生成的表名

db\_tablespace:数据库表空间的名字

ordering:

ordering = ['pub\_date'] # 表示按'pub\_date'字段进行升序排列

ordering = ['-pub\_date'] # 表示按'pub\_date'字段进行降序排列

ordering = ['-pub\_date', 'author'] # 表示先按'pub\_date'字段进行降序排列，再按`author`字段进行升序排列。

Unique\_together: 联合约束

### 模型继承

三种继承方式：

1/抽象基类，将子类共同数据抽离出来，不创建实际的表

元数据：abstract=True

2/多表继承，每个模型都有自己的数据库表

3/代理模型，不改模型字段，只会改模型的python层面的行为。

### 用包组织模型

创建models包，\_\_init\_\_.py中引入所需model文件。

### 查询操作

查询操作，需要基于模型类，通过管理器(Manager)构造一个查询结果集（QerySet），管理器默认是objects。只能通过模型类调用它，以此实现表级别操作和记录级别操作分离。

检索过程不是立即执行的，只有某些的查询才会提交执行。

>>> q = Entry.objects.filter(headline\_\_startswith="What")

>>> q = q.filter(pub\_date\_\_lte=datetime.date.today())

>>> q = q.exclude(body\_text\_\_icontains="food")

>>> print(q)

queryset类似python 的列表，可以切片：

Entry.objects.all()[5:10] # 返回第6个到第10个对象

字段查询：

字段查询其实就是filter()、exclude()和get()等方法的关键字参数。 其基本格式是：field\_\_lookuptype=value，**注意其中是双下划线**。 例如：

>>> Entry.objects.filter(pub\_date\_\_lte='2006-01-01')

#　相当于：

SELECT \* FROM blog\_entry WHERE pub\_date <= '2006-01-01';

详细的参考<http://www.liujiangblog.com/course/django/129>

#### F表达式引用模型的字段

同一个模型的不同字段的比较，

>>> from django.db.models import F

>>> Entry.objects.filter(n\_comments\_\_gt=F('n\_pingbacks'))

F表达式使用\_\_进行跨表查询

Entry.objects.filter(authors\_\_name=F('blog\_\_name'))

#### 使用原生SQL

使用extra

models.Book.objects.filter(publisher\_\_name='人民出版社').extra(where=['price>50'])

models.Book.objects.filter(publisher\_\_name='人民出版社', price\_\_gt=50) models.Book.objects.extra(select={'count': 'select count(\*) from hello\_Book'})

使用raw

Book.objects.raw('select \* from hello\_Book') # 返回模型实例

自定义SQL

from django.db import connection

cursor=connection.cursor()

# 插入操作

cursor.execute("insert into hello\_author(name) values('钱钟书')")

raw=cursor.fetchone() # 返回结果行游标直读向前，读取一条

cursor.fetchall() # 读取所有

#### 不返回queryset的API

| **方法名** | **解释** |
| --- | --- |
| **get()** | 获取单个对象 |
| **create()** | 创建对象，无需save() |
| **get\_or\_create()** | 查询对象，如果没有找到就新建对象 |
| **update\_or\_create()** | 更新对象，如果没有找到就创建对象 |
| bulk\_create() | 批量创建对象 |
| **count()** | 统计对象的个数 |
| in\_bulk() | 根据主键值的列表，批量返回对象 |
| iterator() | 获取包含对象的迭代器 |
| **latest()** | 获取最近的对象 |
| **earliest()** | 获取最早的对象 |
| **first()** | 获取第一个对象 |
| **last()** | 获取最后一个对象 |
| **aggregate()** | 聚合操作 |
| **exists()** | 判断queryset中是否有对象 |
| **update()** | 批量更新对象 |
| **delete()** | 批量删除对象 |

## 模版层

### 变量

量的值来自context中的数据字典, 类似于字典对象的keys到values的映射关系。变量是被{{和}}括起来的部分, **字典查询，属性查询和列表索引查找都是通过圆点符号.来实现**

{{ my\_dict.key }}

{{ my\_object.attribute }}

{{ my\_list.0 }}

### 标签

标签类似Python中的函数，功能多样，使用灵活。可以输出内容、控制结构，甚至可以访问其他的模板标签。标签是由%}和{%来定义的，

## 表单

Django的表单模块给我们提供了下面三个主要功能：

准备和重构数据用于页面渲染

为数据创建HTML表单元素

接收和处理用户从表单发送过来的数据

编写Django的form表单，非常类似我们在模型系统里编写一个模型。在模型中，一个字段代表数据表的一列，而form表单中的一个字段代表<form>中的一个<input>元素。

## 序列化

Django的序列化工具让你可以将Django的模型‘翻译’成其它格式的数据。通常情况下，这种其它格式的数据是基于文本的，并且用于数据交换\传输过程。

### 序列化指定字段

from django.core import serializers

data = serializers.serialize('xml', SomeModel.objects.all(), fields=('name','size'))

### 反序列化

for obj in serializers.deserialize("xml", data):

do\_something\_with(obj)

Python本身不支持序列化类到json格式，Django帮我们实现了它自己的模型类序列化为json的方法，但也仅限于此，如果你在Django内写了一个别的自定义类，一样无法序列化为json格式，除非你自己实现

# 其他

## 开源博客系统

<https://github.com/billvsme/vmaig_blog>