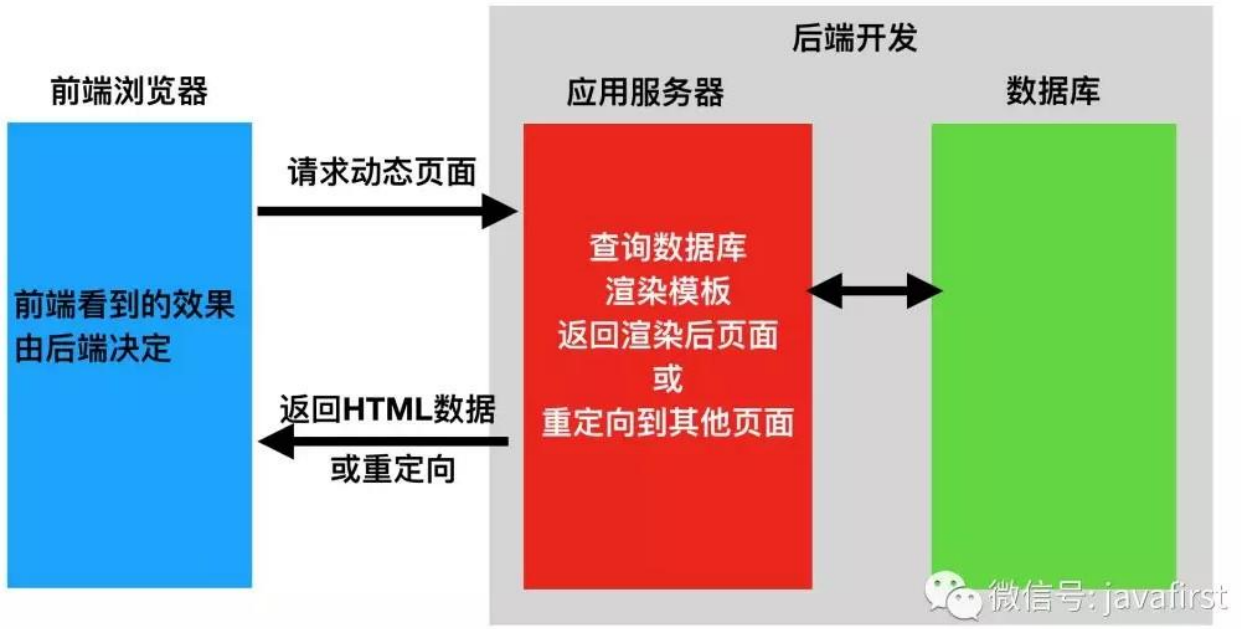
当前的Web开发领域，我们经常会听到**前后端分离**这个技术名词，顾名思义，就是前端页面的开发与后端服务器的开发分离开。这个技术方案的实现要借助API服务模式，API简单说就是开发人员提供编程接口被其他人调用，调用之后后端服务器会返回数据供调用者使用，或者接受调用者发来的数据，修改后端的状态。

让我们来详细了解一下：

前后端不分离

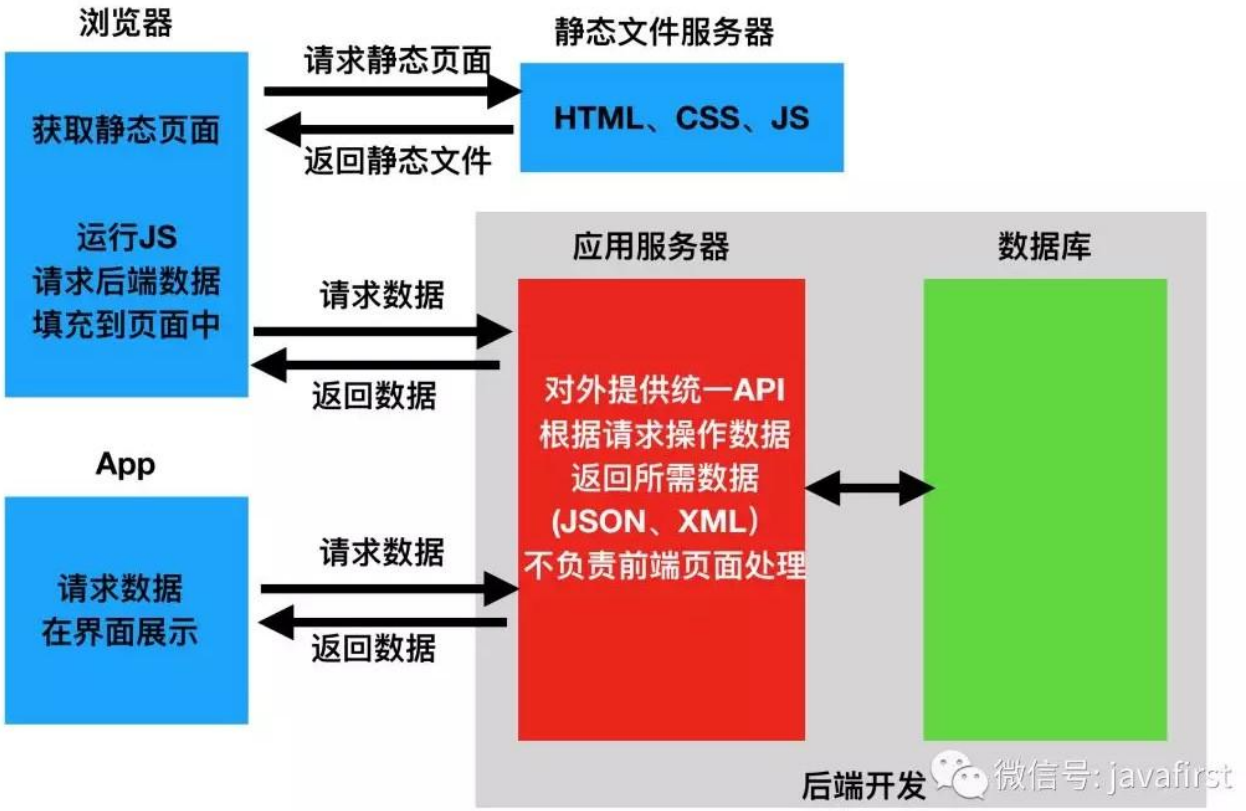
在前后端不分离的应用模式中，前端页面看到的效果都是由后端控制，由后端渲染页面或重定向，也就是后端需要控制前端的展示，前端与后端的耦合度很高。

这种应用模式比较适合纯网页应用，但是当后端对接App时，App可能并不需要后端返回一个HTML网页，而仅仅是数据本身，所以后端原本返回网页的接口不适用于前端App应用，为了对接App后端还需再开发一套接口。



前后端分离

在前后端分离的应用模式中，后端仅返回前端所需的数据，不再渲染HTML页面，不再控制前端的效果。至于前端用户看到什么效果，从后端请求的数据如何加载到前端中，都由前端自己决定，网页有网页的处理方式，App有App的处理方式，**但无论哪种前端，所需的数据基本相同，后端仅需开发一套逻辑对外提供数据即可。**



在前后端分离的应用模式中,我们通常将后端开发的每一视图都称为一个接口，或者API，前端通过访问接口来对数据进行增删改查。

API的风格有多种，但是现在比较主流且实用的就是本文要说的RESTful API。

RESTful

RESTful是一种软件架构风格、**设计风格**，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

REST全称是Representational State Transfer，中文意思是表征状态转移 。 它首次出现在2000年Roy Fielding的博士论文中，**Roy Fielding是HTTP规范的主要编写者之一。** 他在论文中提到："我这篇文章的写作目的，就是想在符合架构原理的前提下，理解和评估以网络为基础的应用软件的架构设计，得到一个功能强、性能好、适宜通信的架构。REST指的是一组架构约束条件和原则。" 如果一个架构符合REST的约束条件和原则，我们就称它为RESTful架构。

要好好地设计我们的url，不要乱搞！

REST本身并没有创造新的技术、组件或服务，而隐藏在RESTful背后的理念就是使用Web的现有特征和能力， 更好地使用现有Web标准中的一些准则和约束。虽然REST本身受Web技术的影响很深， 但是理论上REST架构风格并不是绑定在HTTP上，只不过目前HTTP是唯一与REST相关的实例。 所以我们这里描述的REST也是通过HTTP实现的REST。

RESTful的核心操作：URL定位资源，用HTTP动词（GET,POST,PUT,DELETE）描述操作。

那这种风格的接口有什么好处呢？可以前后端分离。前端拿到数据只负责展示和渲染，不对数据做任何处理。后端处理数据并以JSON格式传输出去，定义这样一套统一的接口，在web，ios，android三端都可以用相同的接口。

关于RESTful的技术内涵，阅读下面三篇博文即可：

https://www.runoob.com/w3cnote/restful-architecture.html

https://blog.csdn.net/qq_21383435/article/details/80032375

http://www.ruanyifeng.com/blog/2018/10/restful-api-best-practices.html

关于RESTful的核心，其实就是如何设计URL！！！

RESTful实践

- API与用户的通信协议，尽量使用HTTPs协议。
- 域名
  - https://api.example.com 最好不要用这种（会存在跨域问题）
  - https://example.org/api/ API很简单
- 版本
  - URL，如： https://api.example.com/v1/
  - 包含在请求头中
- url，任何东西都是资源，均使用名词表示（可复数）
  - https://api.example.com/v1/zoos
  - https://api.example.com/v1/animals
  - https://api.example.com/v1/employees
- method
  - GET ：从服务器取出资源（一项或多项）

- POST ： 在服务器新建一个资源
- PUT ： 在服务器更新资源（客户端提供改变后的完整资源），完整更新

  - PATCH ： 在服务器更新资源（客户端提供改变的属性），局部更新，可能不支持
  - DELETE ： 从服务器删除资源
- 过滤，通过在url上传参的形式传递搜索条件
  - https://api.example.com/v1/zoos?limit=10 ： 指定返回记录的数量
  - https://api.example.com/v1/zoos?oGset=10 ： 指定返回记录的开始位置
  - https://api.example.com/v1/zoos?page=2&per_page=100 ： 指定第几页，以及每页的记录 数
  - https://api.example.com/v1/zoos?sortby=name&order=asc：指定返回结果按照哪个属性 排 序 ， 以 及 排 序 顺 序
  - https://api.example.com/v1/zoos?animal_type_id=1 ： 指定筛选条件
- 状态码 + code信息

```
200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
202 Accepted - [*]：表示一个请求已经进入后台排队（异步任务）
204 NO CONTENT - [DELETE]：用户删除数据成功。
400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
401 Unauthorized - [*]：表示用户没有权限（令牌、用户名、密码错误）。
403 Forbidden - [*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
有XML格式）。
410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。
422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
500 INTERNAL SERVER ERROR - [*]：服务器发生错误，用户将无法判断发出的请求是否成功。
```

- 错误处理，状态码是4xx时，应返回错误信息。

```
{
    "error": "Invalid API  key"
}
```

- 返回结果，针对不同操作，服务器向用户返回的结果应该符合以下规范。

```
GET /collection：返回资源对象的列表（数组）
GET /collection/resource：返回单个资源对象
POST /collection：返回新生成的资源对象
PUT /collection/resource：返回完整的资源对象
PATCH /collection/resource：返回完整的资源对象
DELETE /collection/resource：返回一个空文档
```

- Hypermedia API，RESTful API最好做到Hypermedia，即返回结果中提供链接，连向其他API方法，使得用户不查文档，也知道下一步应该做什么。

```
{"link":
  { "rel": "collection  https://www.example.com/zoos",
  "href":  "https://api.example.com/zoos",
  "title": "List of  zoos",
  "type":  "application/vnd.yourformat+json"
}}
```

HTTP请求方法详解

请求方法：指定了客户端想对指定的资源/服务器作何种操作

根据HTTP标准，HTTP请求可以使用多种请求方法。

HTTP1.0定义了三种请求方法： GET, POST 和 HEAD方

法。

HTTP1.1新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法。

序号	方法	描述
1	GET	请求指定的页面信息，并返回实体主体。
2	HEAD	类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头
3	POST	向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。
4	PUT	从客户端向服务器传送的数据取代指定的文档的内容。
5	DELETE	请求服务器删除指定的页面。
6	CONNECT	HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。
7	OPTIONS	允许客户端查看服务器的性能。
8	TRACE	回显服务器收到的请求，主要用于测试或诊断。

GET：获取资源

GET方法用来请求已被URI识别的资源。指定的资源经服务器端解析后返回响应内容（也就是说，如果请求的资源是文本，那就保持原样返回；如果是CGI[通用网关接口]那样的程序，则返回经过执行后的输出结果）。 最常用于向服务器查询某些信息。必要时，可以将查询字符串参数追加到URL末尾，以便将信息发送给服务器。

POST：传输实体文本

POST方法用来传输实体的主体。 虽然用GET方法也可以传输实体的主体，但一般不用GET方法进行传输，而是用POST方法；虽然GET方法和POST方法很相似，但是POST的主要目的并不是获取响应的主体内容。 POST请求的主体可以包含非常多的数据，而且格式不限。

GET方法和POST方法本质上的区别：

1、GET方法用于信息获取，它是安全的（安全：指非修改信息，如数据库方面的信息），而POST方法是用于修改服务器上资源的请求；

2、GET请求的数据会附在URL之后，而POST方法提交的数据则放置在HTTP报文实体的主体里，所以POST方法的安全性比GET方法要高；

3、GET方法传输的数据量一般限制在2KB，其原因在于：GET是通过URL提交数据，而URL本身对于数据没有限制，但是不同的浏览器对于URL是有限制的，比如IE浏览器对于URL的限制为2KB，而Chrome，FireFox浏览器理论上对于URL是没有限制的，它真正的限制取决于操作系统本身；POST方法对于数据大小是无限制的，真正影响到数据大小的是服务器处理程序的能力。

HEAD：获得报文首部

HEAD方法和GET方法一样，只是不返回报文的主体部分，用于确认URI的有效性及资源更新的日期时间等。 具体来说：1、判断类型； 2、查看响应中的状态码，看对象是否存在（响应：请求执行成功了，但无数据返回）； 3、测试资源是否被修改过 HEAD方法和GET方法的区别：GET方法有实体，HEAD方法无实体。

PUT：传输文件

PUT方法用来传输文件，就像FTP协议的文件上传一样，要求在请求报文的主体中包含文件内容，然后保存在请求URI指定的位置。但是HTTP/1.1的PUT方法自身不带验证机制，任何人都可以上传文件，存在安全问题，故一般不用。

POST和PUT的区别

POST 方法用来传输实体的主体，PUT方法用来传输文件，自身不带验证机制。

这两个方法看起来都是讲一个资源附加到服务器端的请求，但其实是不一样的。一些狭窄的意见认为，POST方法用来创建资源，而PUT方法则用来更新资源。这个说法本身没有问题，但是并没有从根本上解释了二者的区别。事实上，它们最根本的区别就是：POST方法不是幂等的，而PUT方法则有幂等性。那这又衍生出一个问题，什么是幂等？

幂等（idempotent、idempotence）是一个抽象代数的概念。在计算机中，可以这么理解，一个幂等操作的特点就是其任意多次执行所产生的影响均与依次一次执行的影响相同。

POST在请求的时候，服务器会每次都创建一个文件，但是在PUT方法的时候只是简单地更新，而不是去重新创建。因此PUT是幂等的。

DELETE：删除资源

指明客户端想让服务器删除某个资源，与PUT方法相反，按URI删除指定资源

OPTIONS：询问支持的方法

OPTIONS方法用来查询针对请求URI指定资源支持的方法（客户端询问服务器可以提交哪些请求方法）

TRACE：追踪路径

客户端可以对请求消息的传输路径进行追踪，TRACE方法是让Web服务器端将之前的请求通信还给客户端的方法

CONNECT：要求用隧道协议连接代理

CONNECT方法要求在与代理服务器通信时建立隧道，实现用隧道协议进行TCP通信。主要使用SSL（安全套接层）和TLS（传输层安全）协议把通信内容加密后经网络隧道传输。

02-Django-Rest-Framework简介和安装

一、简介

官网地址： https://www.django-rest-framework.org/

> Django Rest Framework是一个开源的，由**合作资助的项目**。如果在商业上使用REST框架，建议**注册付费计划**，使之可以持续开发，健康发展。

Django Rest Framework是一个强大且灵活的工具包，主要用以构建RESTful风格的Web API。

Django REST Framework（以后简称DRF）可以在Django的基础上迅速实现API，并且自身还带有基于WEB的测试和浏览页面，可以方便的测试自己的API。DRF几乎是Django生态中进行前后端分离开发的默认库。

Django REST Framework具有以下功能和特性：

- 自带基于Web的可浏览的API，对于开发者非常有帮助
- 支持OAuth1a 和OAuth2认证策略
- 支持ORM或非ORM数据源的序列化
- 高可自定制性，多种视图类型可选
- 自动生成符合 RESTful 规范的API
- 支持 OPTION、HEAD、POST、GET、PATCH、PUT、DELETE等HTTP方法

- 根据 Content-Type 来动态的返回数据类型（如HTML、
- json） 细粒度的权限管理（可到对象级别）
- 丰富的文档和强大的社区支持
- Mozilla、 Red Hat、 Heroku和Eventbrite等知名公司正在使用

二、安装依赖

当前时间2019年10月，DRF版本3.9.2，依赖及支持如下:

- Python (2.7, 3.4, 3.5, 3.6, 3.7)
- Django (1.11, 2.0, 2.1, 2.2)

通常我们都是使用最新稳定版本的Python和Django，比如当下的Django2.2和

Python3.7.3。以下软件包是可选的：

- coreapi（1.32.0+） - 支持模式生成和coreapi命令行工
- 具。Markdown（2.1.0+） - 可浏览API的Markdown支持。
- django-filter（1.0.1+） - 过滤支持。
- django-crispy-forms - 改进的HTML显示过滤。
- django-guardian（1.1.1+） - 对象级别的权限支持。

三、安装方法

直接使用pip就可以安装：

```
pip install  djangorestframework
pip install markdown        # Markdown
pip install django-filter     # 可 选
pip install coreapi             # 可 选
```

或者通过git下载：

```
git clone https://github.com/cskaoyan/django-rest-framework
```

安装完毕，在项目配置文件中，注册app：

```
INSTALLED_APPS =
(
    ...
    'rest_framework',
)
```

如果你想使用基于浏览器的可视化的API目录，并且希望获得一个登录登出功能，那么可以在根路由下添加下面的路由，这个功能类似Django自带的admin后台：

```
urlpatterns =
[
    ...
    path('api-auth/', include('rest_framework.urls'))
]
```

为了登录操作，也许你还要生成数据表，创建超级用户。这个步骤可选：

```
python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser
```

就可以访问http://127.0.0.1:8000/api-auth/login/去登陆

在Windows中，DRF的目录结构如下：

四、简单的使用

在项目的根路由urls.py文件中，写入下面的代码：

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include("drf_rumen.urls")),#给app分配路由
    #配置下面路由链接才有登录效果
    re_path(r'^api-auth/', include('rest_framework.urls')),
]
```

新增serializers.py

```
from rest_framework import serializers
from booktest.models import BookInfo
class BookInfoSerializer(serializers.ModelSerializer):
```

```
    """专门用于对图书进行进行序列化和反序列化的类：序列化器类"""
    class Meta:
        # 当前序列化器在序列化数据的时候,使用哪个模型
        model = BookInfo
        # fields = ["id","btitle"] # 多个字段可以使用列表声明,如果是所有字段都要转换,则使用 '__all__'
        fields = '__all__' # 多个字段可以使用列表声明,如果是所有字段都要转换,则使用 '__all__'
```

新增urls.py

```
from rest_framework.routers import DefaultRouter
from .views import BookInfoAPIView
urlpatterns = []

# 创建路由对象
routers = DefaultRouter()
# 通过路由对象对视图类进行路由生成
routers.register("books",BookInfoAPIView)

urlpatterns+=routers.urls
```

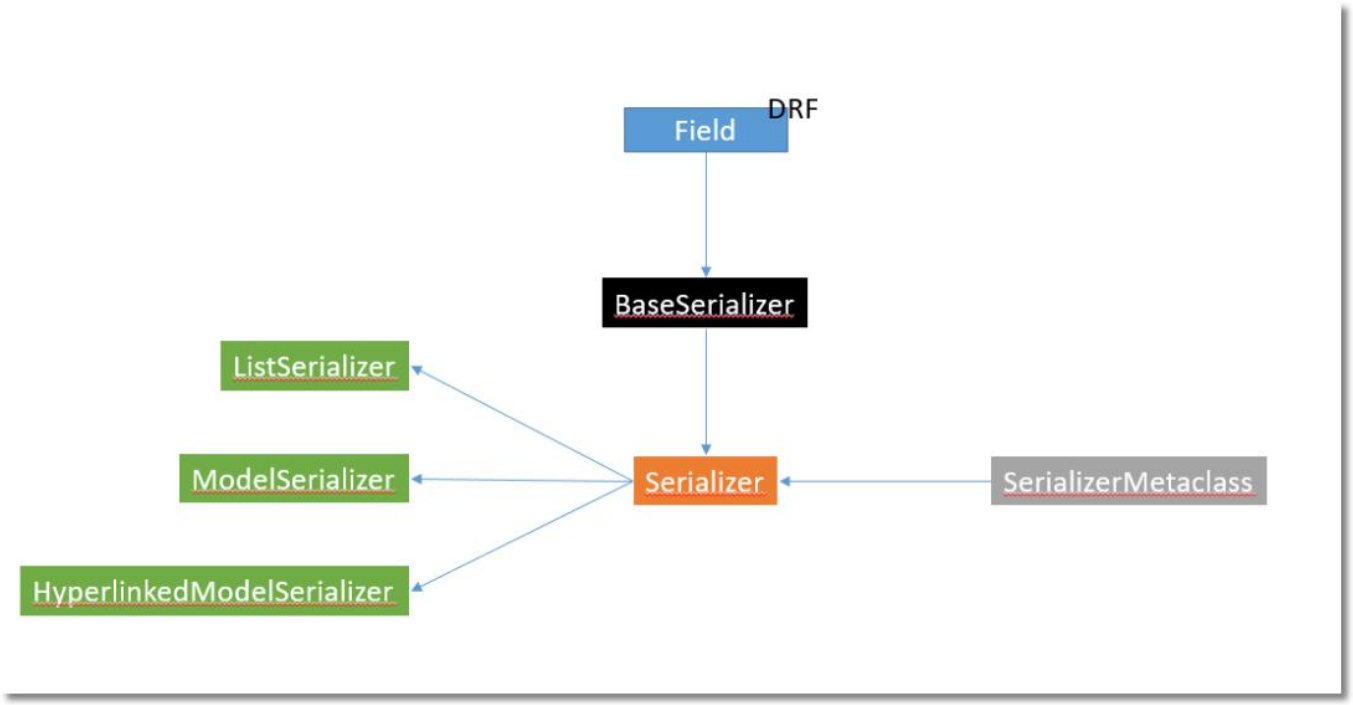
models中增加

```
from django.db import models

# Create your models here.
from django.db import models

#定义图书模型类BookInfo
class BookInfo(models.Model):
    btitle = models.CharField(max_length=20, verbose_name='图书标题')
    bpub_date = models.DateField(verbose_name='出版时间')
    bread = models.IntegerField(default=0, verbose_name='阅读量')
    bcomment = models.IntegerField(default=0, verbose_name='评论量')
    is_delete = models.BooleanField(default=False, verbose_name='逻辑删除')
```

```
    class Meta:
        db_table = 'tb_books'  # 指明数据库表名
        verbose_name = '图书'  # 在admin站点中显示的名称
        verbose_name_plural = verbose_name  # 显示的复数名称

    def __str__(self):
        """定义每个数据对象的显示信息"""
        return "图书:《"+self.btitle+"》"

#定义英雄模型类HeroInfo
class HeroInfo(models.Model):
    GENDER_CHOICES = (
        (0, 'female'),
        (1, 'male')
    )
    hname = models.CharField(max_length=20, verbose_name='名称')
    hgender = models.SmallIntegerField(choices=GENDER_CHOICES, default=0, verbose_name='性
别')
    hcomment = models.CharField(max_length=200, null=True, verbose_name='描述信息')
    hbook = models.ForeignKey(BookInfo, on_delete=models.CASCADE, verbose_name='图书')  #
外键
    is_delete = models.BooleanField(default=False, verbose_name='逻辑删除')

    class Meta:
        db_table = 'tb_heros'
        verbose_name = '英雄'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.hname
```

views中增加

```
from rest_framework.viewsets import ModelViewSet
from drf_rumen.models import BookInfo
from .serializers import BookInfoSerializer
# Create your views here.
class BookInfoAPIView(ModelViewSet):
    # 当前视图类所有方法使用得数据结果集是谁?
    queryset = BookInfo.objects.all()
    # 当前视图类使用序列化器类是谁
    serializer_class = BookInfoSerializer
```



03-快速入门1--序列化

一、概述

本教程将介绍如何创建一个简单的对代码片段进行高亮展示的Web API。这个过程中，将会介绍组成DRF框架的各个组件，并让你大概了解各个组件是如何一起工作的。

这个教程是相当深入的，可能需要结合后面的API，反复揣摩。

二、创建虚拟环境

我们先用virtualenv创建一个新的虚拟环境。这样就能确保与我们正在开展的任何其他项目保持良好的隔离。

进入virtualenv环境后，安装我们需要的包。

```
pip install django   #当前为2.2版本
pip install  djangorestframework    #当然为3.10版
pip install pygments   # 代码高亮插件
```

DRF在导入时的名字为 `rest_framework` ，不要搞错了。

注意： 要随时退出virtualenv环境，只需输入 `deactivate` 。

三、 创建项目

好了，我们现在要开始写代码了。 首先，创建一个新的项目。

```
cd ~
django-admin.py startproject tutorial
cd tutorial
```

再创建一个app，名字叫做snippets，这个单词的意思是'片段'，理解为代码片段。

```
python manage.py startapp snippets
```

需要将新建的snippets放入apps

```
INSTALLED_APPS =
(
    ...
    'rest_framework',
'snippets.apps.SnippetsConfig',
)
```

好了，我们的准备工作做完了。

四、编写model模型

为了实现本教程的目的，我们将开始创建一个用于存储代码片段的简单的 model模型。打开 `snippets/models.py` 文件，并写入下面的代码：

```
from django.db import models
from pygments.lexers import get_all_lexers
from pygments.styles import get_all_styles

# 下面的几行代码是处理代码高亮的，不好理解，但没关系，它不重要。
LEXERS = [item for item in get_all_lexers() if item[1]]
LANGUAGE_CHOICES = sorted([(item[1][0], item[0]) for item in LEXERS])
STYLE_CHOICES = sorted((item, item) for item in get_all_styles())

class Snippet(models.Model):
created = models.DateTimeField(auto_now_add=True)
title = models.CharField(max_length=100, blank=True, default='')
code = models.TextField()
linenos = models.BooleanField(default=False)
        language = models.CharField(choices=LANGUAGE_CHOICES, default='python',
max_length=100)
        style = models.CharField(choices=STYLE_CHOICES, default='friendly',
max_length=100)

class Meta:
ordering = ('created',)
```

然后，使用下面的命令创建数据表：

```
python manage.py makemigrations
python manage.py migrate
```

五、创建序列化类

开发Web API的第一件事是为我们的代码片段对象创建一种序列化和反序列方法，将其与诸如 `json` 格式进行互相转换。具体方法是声明与Django forms非常相似的序列化器（serializers）来实现。 在 `snippets` 的目录下创建一个名为 `serializers.py` 文件，并添加以下内容。

> 为每一个你需要序列化的model创建一个对应的序列化类。

```
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES


class SnippetSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)  # 序列化时使用，反序列化时不用
    title = serializers.CharField(required=False, allow_blank=True, max_length=100)
    code = serializers.CharField(style={'base_template': 'textarea.html'})
    linenos = serializers.BooleanField(required=False)
    language = serializers.ChoiceField(choices=LANGUAGE_CHOICES, default='python')
    style = serializers.ChoiceField(choices=STYLE_CHOICES, default='friendly')
    # 注意，没有为model的created创建对应的序列化字段
    def create(self, validated_data):
        """
        使用验证后的数据，创建一个代码片段对象。使用的是Django的ORM的语法。
        """
        return Snippet.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """
        使用验证过的数据，更新并返回一个已经存在的‘代码片段’对象。依然使用的是Django的
```

```
    ORM的语法。
22          """
23          instance.title = validated_data.get('title', instance.title)
```

```
24            instance.code = validated_data.get('code', instance.code)
25            instance.linenos = validated_data.get('linenos', instance.linenos)
26            instance.language = validated_data.get('language',
    instance.language)
27            instance.style = validated_data.get('style', instance.style)
28            instance.save()
29            return instance
```

Serializer是DRF提供的序列化基本类，供我们继承使用，它位于DRF的serializers包中。使用这个类，你需要自己编写所有的字段以及create和update方法，比较底层，抽象度较低，接近Django 的form表单类的层次。

让我们看一下上面的代码。SnippetSerializer类的第一部分定义了序列化/反序列化过程中需要的字段。 `create()` 和 `update()` 方法定义了在调用 `serializer.save()` 时如何创建和修改实例。

序列化类与Django 的 `Form` 类非常相似，并在各种字段中包含类似的验证标志，例如 `required` ， `max_length` 和 `default` 。这里暂时不讲解各种字段的含义，以及它们包含的参数的用法，详见API。

字段参数可以控制serializer在某些情况下如何显示，比如渲染HTML的时候。上面的 `style={'base_template': 'textarea.html'}` 等同于在Django 的 `Form` 类中使用 `widget=widgets.Textarea` ，也就是使用文本输入框标签。这对于控制如何显示可浏览的API特别有用。

实际上也可以通过使用 `ModelSerializer` 类来节省一些编写代码的时间，就像教程后面会用到的那样。但是现在还继续使用我们刚才定义的serializer。

面向切面编程
https://www.zhihu.com/question/24863332
https://www.cnblogs.com/ajianbeyourself/p/3665996.html

六、序列化器的基本使用

我们先来熟悉一下Serializer类的基本使用方法。输入下面的命令进入Django shell。

```
1  python manage.py shell
```

在命令行中，像下面一样导入几个模块，然后创建一些代码片段：

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser

snippet = Snippet(code='foo = "bar"\n')
snippet.save()

snippet = Snippet(code='print("hello, world")\n')
snippet.save()
```

我们现在已经有2个代码片段实例了，让我们将第二个实例序列化：

```
serializer = SnippetSerializer(snippet)
serializer.data
# {'id': 2, 'title': '', 'code': 'print("hello, world")\n', 'linenos':
False, 'language': 'python', 'style': 'friendly'}
```

此时，我们将模型实例对象转换为了Python的原生数据类型。

```
>>> type(serializer)
<class 'snippets.serializers.SnippetSerializer'>
>>> type(serializer.data)
<class 'rest_framework.utils.serializer_helpers.ReturnDict'>
>>> issubclass(type(serializer.data), dict)
True
```

但是，要完成最终的序列化过程，我们还需要将数据转换成 `json` 格式，这样的话，客户端才可以理解。

```
content = JSONRenderer().render(serializer.data)
content
# b'{"id": 2, "title": "", "code": "print(\\"hello, world\\")\\n",
"linenos": false, "language": "python", "style": "friendly"}'

type(content)
#<class 'bytes'>
```

以上是序列化过程，也就是使用ORM从数据库中读取对象，然后序列化为DRF的某种格式，再转换为json格式（为什么上面是bytes类型，这是和Python语言相关的），最后将json数据通过HTTP发送给客户端。

反序列化则是上面过程的逆向。首先我们使用Python内置的io模块将我们前面生成的content转换为一个流（stream）对象，模拟从前端发送过来的json格式的请求数据，然后将数据解析为Python原生数据类型：

```
import io

stream =  io.BytesIO(content)
data = JSONParser().parse(stream)
type(data)
# <class 'dict'>
```

然后我们要将数据类型转换成模型对象实例并保存。（实际保存要使用Snippet.objects.create(**serializer.validated_data)）

serializer.data是未经过验证的数据，经过验证后，就会得到serializer.validated_data，如果验证失败，serializer.validated_data就会为空，详见下面链接

https://stackoverflow.com/questions/42000687/what-are-the-differences-between-data-and-validated-data

```
serializer = SnippetSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# OrderedDict([('title', ''), ('code', 'print("hello, world")\n'),
('linenos', False), ('language', 'python'), ('style', 'friendly')])
serializer.save()
# <Snippet: Snippet object (3)>
```

You cannot call `.save()` after accessing `serializer.data`."
AssertionError: You cannot call `.save()` after accessing `serializer.data`.If you need to access data before committing to the database then inspect 'serializer.validated_data' instead.

上面的操作都是在shell中进行的，实际中我们不会这么麻烦。

可以看到序列化器的API和Django的表单(forms)是多么相似。

也可以序列化查询结果集（querysets）而不是单个模型实例，也就是同时序列化多个对象。只需要为serializer添加一个 `many=True` 标志。（这个功能是比较重要的）

```
serializer = SnippetSerializer(Snippet.objects.all(), many=True)
serializer.data
# [OrderedDict([('id', 1), ('title', ''), ('code', 'foo = "bar"\n'),
('linenos', False), ('language', 'python'), ('style',  'friendly')]),
OrderedDict([('id', 2), ('title', ''), ('code', 'print("hello, world")\n'),
('linenos', False), ('language', 'python'), ('style', 'friendly')]),
OrderedDict([('id', 3), ('title', ''), ('code', 'print("hello,  world")'),
('linenos', False), ('language', 'python'), ('style', 'friendly')])]
```

七、使用ModelSerializers类

除了前面的Serializer类，DRF还给我们提供了几种别的可以继承的序列化类，ModelSerializers就是常用的一个。

前面我们写的的 `SnippetSerializer` 类中重复了很多包含在 `Snippet` 模型类（model）中的信息。如果能自动生成这些内容，像Django的 `ModelForm` 那样就更好了。事实上REST framework的 `ModelSerializer` 类就是这么一个类，它会根据指向的model，自动生成默认的字段和简单的create及update方法。

让我们来看看如何使用 `ModelSerializer` 类重构我们的序列化类。再次打开 `snippets/serializers.py` 文件，并将 `SnippetSerializer` 类替换为以下内容。

```
1   class SnippetSerializer(serializers.ModelSerializer):
2       class Meta:
3           model = Snippet
4           fields = ('id', 'title', 'code', 'linenos', 'language',  'style')
```

额外的提示一下，DRF的序列化类有一个repr属性可以通过打印序列化器类实例的结构(representation)查看它的所有字段。以下操作在命令行中进行：

```
1   from snippets.serializers import SnippetSerializer
2   serializer = SnippetSerializer()
3   print(repr(serializer))
4   # SnippetSerializer():
5   #     id = IntegerField(label='ID', read_only=True)
6   #     title = CharField(allow_blank=True, max_length=100, required=False)
7   #     code = CharField(style={'base_template': 'textarea.html'})
8   #     linenos = BooleanField(required=False)
9   #     language = ChoiceField(choices=[('Clipper', 'FoxPro'), ('Cucumber',
    'Gherkin'), ('RobotFramework', 'RobotFramework'), ('abap', 'ABAP'), ('ada',
    'Ada')...
10  #     style = ChoiceField(choices=[('autumn', 'autumn'),  ('borland',
    'borland'), ('bw', 'bw'), ('colorful', 'colorful')...
```

注意： `ModelSerializer` 类并不会做任何特别神奇的事情，它们只是创建序列化器类的快捷方式：

- 一组自动确定的字段。
- 默认简单实现的 `create()` 和 `update()` 方法。

这个 `ModelSerializer` 类帮我们节省了很多代码，但同时，又降低了可定制性，如何取舍，取决于你的业务逻辑。

没有谁规定必须用 `ModelSerializer` 类，不能用前面的更基础的Serializer类，实际上在复杂的业务逻辑中，定制性更高的Serializer类，反而是更实用的。 `ModelSerializer`

类感觉比较鸡肋。

八、编写常规的Django视图

让我们看看如何使用我们新的Serializer类编写一些API视图。目前我们不会使用任何REST框架的其他功能，我们只需将视图作为常规Django视图编写。

编辑 `snippets/views.py` 文件，并且添加以下内容：

```
from django.http import HttpResponse, JsonResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
```

我们API的根视图的功能是列出所有的snippets或创建一个新的snippet。

```
@csrf_exempt    # 防止403
def snippet_list(request):
    """
    列出所有的代码片段或者创建新的。
    """
    if request.method ==  'GET': snippets
        =  Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True) #注意many参数
        # 实用Django自带方法，响应json格式的数据
        return JsonResponse(serializer.data, safe=False)

    elif request.method == 'POST':
        data = JSONParser().parse(request)
        serializer = SnippetSerializer(data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data, status=201)
        return JsonResponse(serializer.errors,  status=400)
```

请注意，因为我们后面会使用没有CSRF令牌的客户端对此视图进行POST测试，因此我们需要为视图增加 `csrf_exempt` 装饰器，跳过csrf的检测，避免403。事实上DRF有专门应对的策略。

另外，我们还需要写一个与单个snippet对象相应的detail视图，用于获取，更新和删除这个snippet。

```
@csrf_exempt
def snippet_detail(request, pk):
```

```
    """
    获取、更新和删除指定的某个代码片段。
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return HttpResponse(status=404)

    if request.method == 'GET':
        serializer = SnippetSerializer(snippet)
        return JsonResponse(serializer.data)

    elif request.method == 'PUT':
        data = JSONParser().parse(request)
        serializer = SnippetSerializer(snippet, data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data)
        return JsonResponse(serializer.errors, status=400)

    elif request.method == 'DELETE':
        snippet.delete()
        return HttpResponse(status=204)
```

注意视图函数的名称！注意两个视图函数各自支持的HTTP操作！注意区分POST和PUT方法！注意，同时只能创建或更新一个对象，暂时不支持批量更新或创建！但是读取可以批量!

视图有了，序列化器有了，模型有了，我们还差编写路由把请求和视图链接起来。创建一个 `snippets/urls.py` 文件:

```
from django.urls import  path
from snippets import views

urlpatterns = [
    path('snippets/', views.snippet_list),
    path('snippets/<int:pk>/',  views.snippet_detail),
]
```

上面是snippets这个app自己的二级路由文件，我们还需要在项目根URL配置 `tutorial/urls.py` 文件中，添加我们的snippet应用的include语句。

```
from django.urls import path,  include

urlpatterns = [
    path('',  include('snippets.urls')),
]
```

注：原来的admin路由，保留与否，随意。

这样，`127.0.0.1:8000/snippets/` 将访问 `snippet_list` 视图。
值得注意的是，目前我们还没有正确处理好几种特殊情况。比如假设我们发送格式错误的 `json` 数据，或者使用视图不处理的HTTP方法发出请求，那么我们最终会出现一个500“服务器错误”响应。不过，暂时没有关系。

九、测试前面的工作（直接看下面截图来测试）

现在退出所有的shell...，启动Django开发服务器：

```
Watching for file changes with  StatReloader
Performing system checks...

System check identified no issues (0 silenced).
April 28, 2019 - 10:51:57
Django version 2.2, using settings 'tutorial.settings'
Starting development server at  http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

打开一个终端窗口，我们在命令行下测试服务器。

我们可以使用curl或httpie测试我们的服务器。Httpie是用Python编写的用户友好的http客户端，我们安装它。

可以使用pip来安装httpie：

```
pip install httpie
```

访问下面的url可以得到所有snippet的列表：

```
输入命令：http  http://127.0.0.1:8000/snippets/

结果如下：
HTTP/1.1 200 OK
Content-Length: 354
```

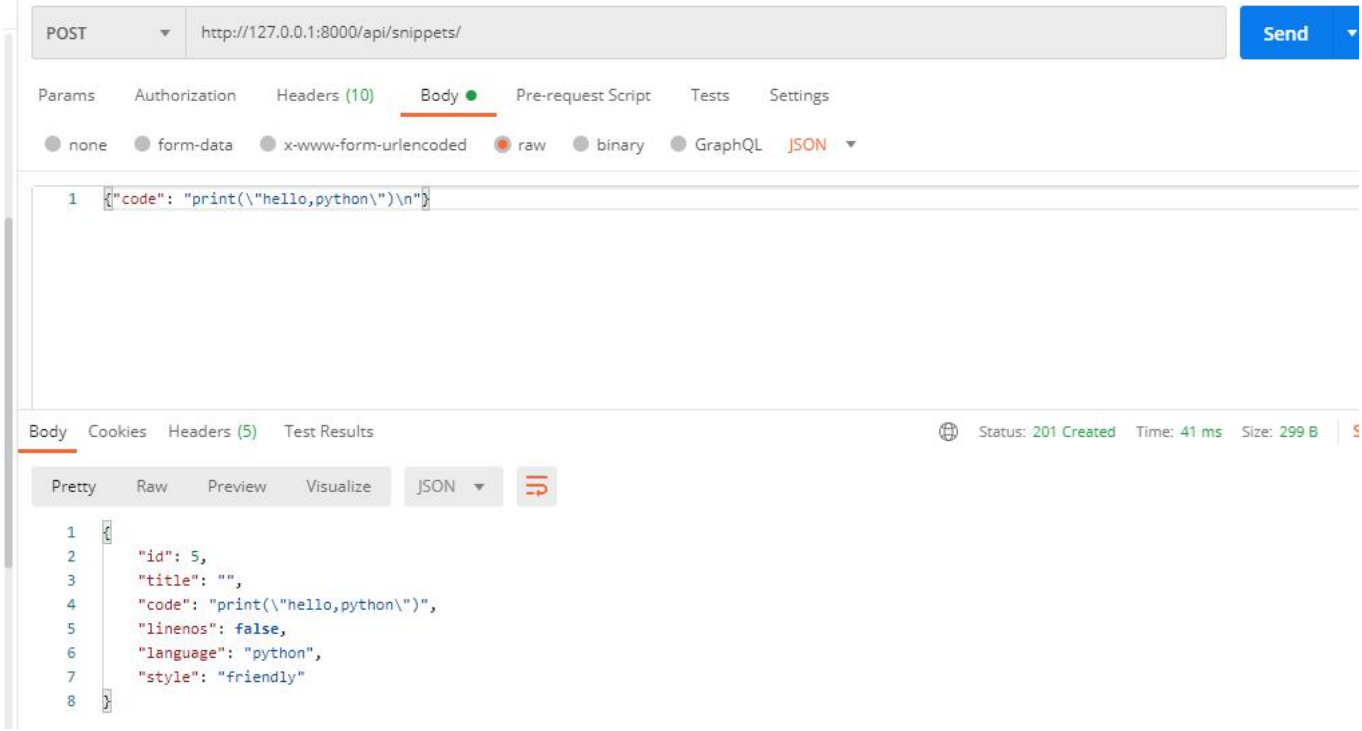
```
Content-Type: application/json
Date: Sun, 28 Apr 2019 02:53:42  GMT
Server: WSGIServer/0.2  CPython/3.7.3
X-Frame-Options: SAMEORIGIN

[
    {
        "code": "foo = \"bar\"\n",
        "id": 1,
        "language": "python",
        "linenos": false,
        "style": "friendly",
        "title": ""
    },
    {
        "code": "print(\"hello, world\")\n",
        "id": 2,
        "language": "python",
        "linenos": false,
        "style": "friendly",
        "title": ""
    },
    {
        "code": "print(\"hello, world\")",
        "id": 3,
        "language": "python",
        "linenos": false,
        "style": "friendly",
        "title": ""
    }
]
```

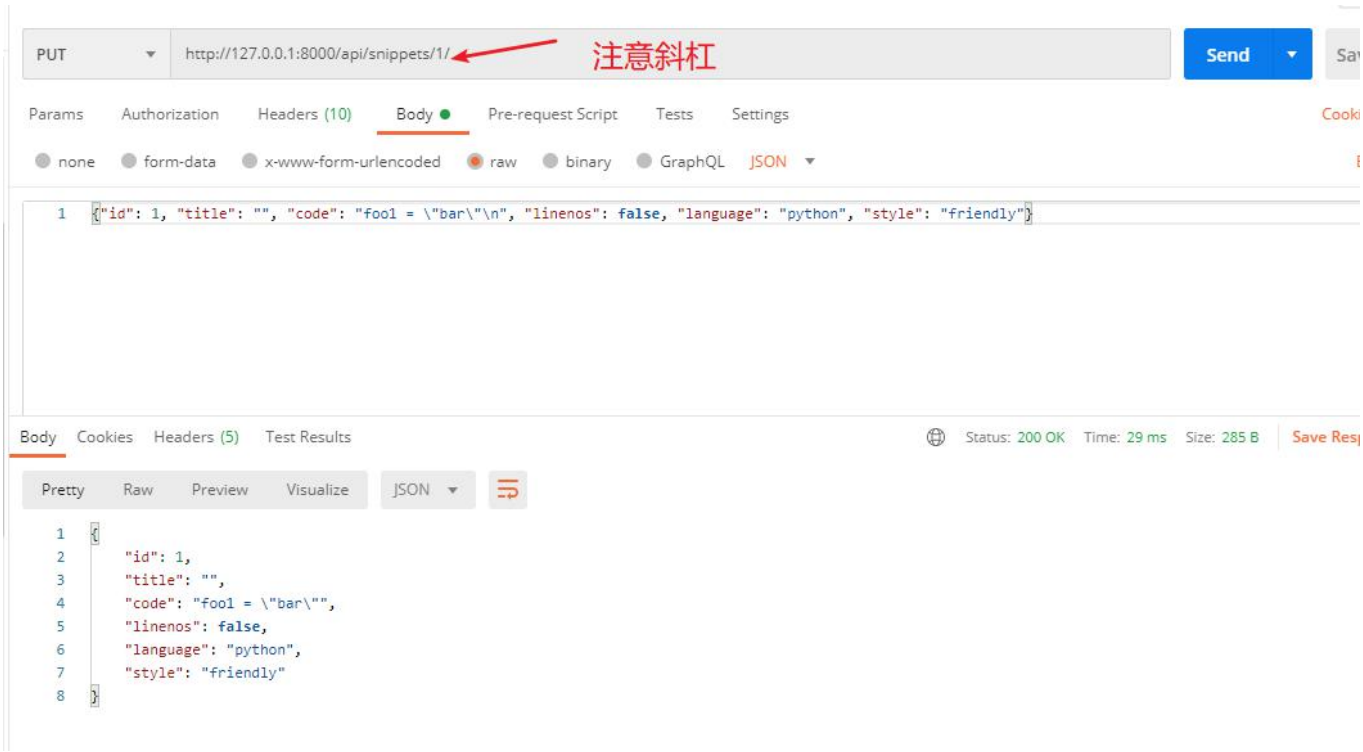
或者我们可以指定id来获取特定snippet的detail信息：

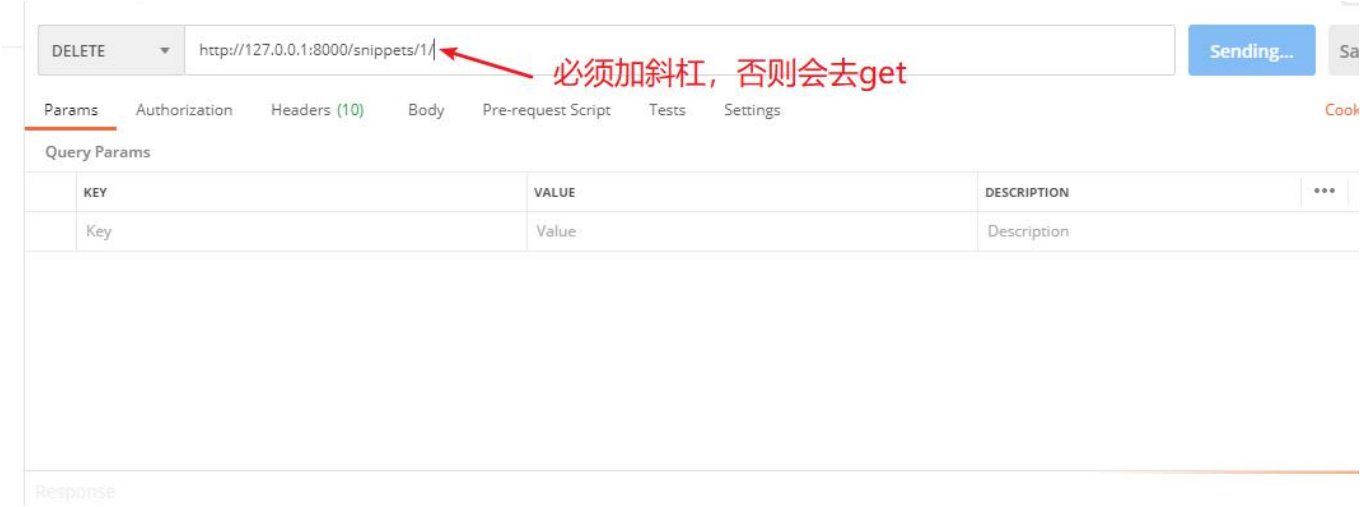
```
http  http://127.0.0.1:8000/snippets/2/

HTTP/1.1 200 OK
Content-Length: 120
Content-Type: application/json
Date: Sun, 28 Apr 2019 02:54:21  GMT
Server: WSGIServer/0.2 CPython/3.7.3
X-Frame-Options: SAMEORIGIN

{
```

```
        "code": "print(\"hello, world\")\n",
        "id": 2,
        "language": "python",
        "linenos": false,
        "style": "friendly",
        "title": ""
    }
```

当然，也可以在浏览器中访问这些URL来显示相同的json。


POST
http://127.0.0.1:8000/api/snippets/
Send
Params
Authorization
Headers (10)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
1 {"code": "print(\"hello,python\")\n"}
Body
Cookies
Headers (5)
Test Results
Status: 201 Created
Time: 41 ms
Size: 299 B
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "id": 5,
3 "title": "",
4 "code": "print(\"hello,python\")",
5 "linenos": false,
6 "language": "python",
7 "style": "friendly"
8 }


PUT
http://127.0.0.1:8000/api/snippets/1/
注意斜杠
Send
Params
Authorization
Headers (10)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
1 {"id": 1, "title": "", "code": "foo1 = \"bar\"\n", "linenos": false, "language": "python", "style": "friendly"}
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 29 ms
Size: 285 B
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "id": 1,
3 "title": "",
4 "code": "foo1 = \"bar\"",
5 "linenos": false,
6 "language": "python",
7 "style": "friendly"
8 }



04-快速入门2--请求和响应

从现在开始，我们将真正开始接触REST框架的核心。 我们来介绍几个基本的构建模块。

一、请求对象（Request objects）

DRF引入了一个扩展Django常规 `HttpRequest` 对象的 `Request` 对象，并提供了更灵活的请求解析能力。 `Request` 对象的核心功能是 `request.data` 属性，它 `request.POST` 类似，但与
对于使用Web API更为有用。

```
request.POST  # 只处理表单数据  只适用于'POST'方法
request.data  # 处理任意数据  适用于'POST', 'PUT'和'PATCH'等方法
```

在DRF中，请始终使用`request.data`，不要使用`request.POST`。

二、响应对象（Response objects）

DRF同时还引入了一个 `Response` 对象，这是一种尚未对内容进行渲染 `TemplateResponse` 类型，并使用内容协商的结果来确定返回给客户端正确的内容类型。

```
return Response(data)  # 渲染成客户端请求的内容类型。
```

三、状态码（Status codes）

在前后端分离的RESTful模式中，我们不能简单、随意地返回响应，而是需要使用HTTP规定的，大家都认可的

状态码的形式。

然而，在视图中使用纯数字的HTTP 状态码并不总是那么容易被理解，很容易被忽略。REST框架为 `status` 模块中的每个状态代码（如 `HTTP_400_BAD_REQUEST` ）提供了更明确的标识符。使用它们来代替纯数字的HTTP状态码是个很好的主意。

四、封装API视图

REST框架提供了三种可用于编写API视图的包装器（wrappers）。

1. 基于函数视图的 `@api_view`装饰器
2. 基于类视图的 `APIView`类系列

3 基于 `viewset` 视图集的类系列

这些包装器提供了一些功能，例如确保你在视图中接收到 `Request` 实例，并将上下文添加到 `Response`，以便可以执行内容协商的约定。

包装器还提供了诸如在适当时候返回 `405 Method Not Allowed` 之类的响应，并处理在使用格式错误的输入来访问 `request.data` 时发生的任何 `ParseError` 异常。

> 视图体系是DRF最复杂，最晦涩的部分，文档写得特别不清晰，各种用法穿插，没有统一的逻辑。每种编写视图的方法，又分别对应不同代码细节，非常难以记忆和理解。

五、修改成真正的API视图

下面我们开始使用新的组件来写几个视图。

首先，我们使用 `@api_view` 装饰器来将一个传统的Django视图改造成DRF的API视图。

删除 `views.py` 中原来所有的内容，写入下面的代码：（**JsonResponse改为Response**）

```
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from snippets.models import Snippet
from snippets.serializers import  SnippetSerializer


@api_view(['GET', 'POST'])
def snippet_list(request):
    """
    List all code snippets, or create a new snippet.
    """
    if request.method ==  'GET': snippets
        =  Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = SnippetSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
```

```
23          return Response(serializer.errors,
    status=status.HTTP_400_BAD_REQUEST)
```

当前的视图比前面的示例有所改进，它稍微简洁一点。在装饰器的参数位置指定视图支持的HTTP方法类型。不需要你额外处理csrf问题，也不使用Django的JSONRsponse方法了，而是使用DRF的Response方法。此外，我们还使用了命名状态代码status，这使得响应意义更加明显。你可以对比一下代码前后的变化，加深理解和印象。

同样，我们重写 `views.py` 模块中snippet的detail视图。

```
1   @api_view(['GET', 'PUT', 'DELETE'])
2   def snippet_detail(request, pk):
3       """
4       Retrieve, update or delete a code snippet.
5       """
6       try:
7           snippet = Snippet.objects.get(pk=pk)
8       except Snippet.DoesNotExist:
9           return  Response(status=status.HTTP 404 NOT FOUND)
10
11      if request.method == 'GET':
12          serializer = SnippetSerializer(snippet)
13          return Response(serializer.data)
14
15      elif request.method ==  'PUT':
16          serializer = SnippetSerializer(snippet, data=request.data)
17          if serializer.is_valid():
18              serializer.save()
19              return Response(serializer.data)
20          return Response(serializer.errors,
    status=status.HTTP_400_BAD_REQUEST)
21
22      elif request.method == 'DELETE':
23          snippet.delete()
24          return  Response(status=status.HTTP_204_NO_CONTENT)
```

目前为止，我们写的API视图和普通的Django视图没有什么太大区别，还是很好理解的。

注意，我们不再显式地将请求或响应绑定到给定的内容类型。 `request.data` 可以处理传入的 `json` 请求，也可以处理其他格式。同样，我们返回带有数据的响应对象，但允许REST框架将响应给渲染成正确的内容类型，比如json。

使用drf就会有界面，有界面是因为请求的Accept　　text/html，如果想要JSON格式的数据，请求时Accept　　application/json，得到的就是json格式的数据

六、 为url添加可选的后缀

> 其实这部分内容不应该出现在这里，它是和主干无关的细节。

在DRF的机制中，响应数据的格式不再与单一内容类型连接，可以同时响应json格式或HTML格式。我们可以为API路径添加对格式后缀的支持。使用格式后缀给我们明确指定了给定格式的URL，这意味着我们的API将能够处理诸如 `http://example.com/api/items/4.json` 之类的URL。

像下面这样在这两个视图中添加一个 `format` 关键字参数。

```
def snippet_list(request, format=None):
```

和

```
def snippet_detail(request, pk, format=None):
```

仅仅在视图中添加format参数还不够，还需要在路由中进行设置。现在更新 `snippets/urls.py` 文件，为现有的URL后面添加一组 `format_suffix_patterns` 。

```
from django.urls import path
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

urlpatterns = [
    path('snippets/', views.snippet_list),
    path('snippets/<int:pk>',  views.snippet_detail),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

注意上面最后一句代码，它的意思是用 `format_suffix_patterns` 来封装urlpatterns，这样每一个带有 `.json` 等后缀的url都能被正确解析。

以上操作都是可选的，我们不一定需要添加这些额外的url模式，但它给了我们一个简单，清晰的方式来引用特定的格式。

七、测试我们的工作

从命令行开始测试API，就像我们在前面所做的那样。

可以像以前一样获取所有snippet的列表。

```
执行命令：http http://127.0.0.1:8000/snippets/


HTTP/1.1 200 OK
Allow: GET, OPTIONS, POST
Content-Length: 319
Content-Type: application/json
Date: Sun, 28 Apr 2019 04:32:47 GMT
Server: WSGIServer/0.2 CPython/3.7.3
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

[
    {
        "code": "foo = \"bar\"\n",
        "id": 1,
        "language": "python",
        "linenos": false,
        "style": "friendly",
    "title": ""
    },
    {
        "code": "print(\"hello, world\")\n",
        "id": 2,
        "language": "python",
        "linenos": false,
        "style": "friendly",
    "title": ""
    },
    {
        "code": "print(\"hello, world\")",
        "id": 3,
        "language": "python",
        "linenos": false,
        "style": "friendly",
    "title": ""
    }
```

```
39    ]
40
```

可以通过使用 Accept标头来控制我们回复的响应格式：

```
http  http://127.0.0.1:8000/snippets/  Accept:application/json  # 请求JSON
http  http://127.0.0.1:8000/snippets/  Accept:text/html         # 请求HTML
```

或者通过附加后缀的方式：

```
http  http://127.0.0.1:8000/snippets.json  # JSON后缀
http  http://127.0.0.1:8000/snippets.api   # 可浏览API后缀
```

类似地，可以使用 `Content-Type` 头控制我们发送的请求的格式。

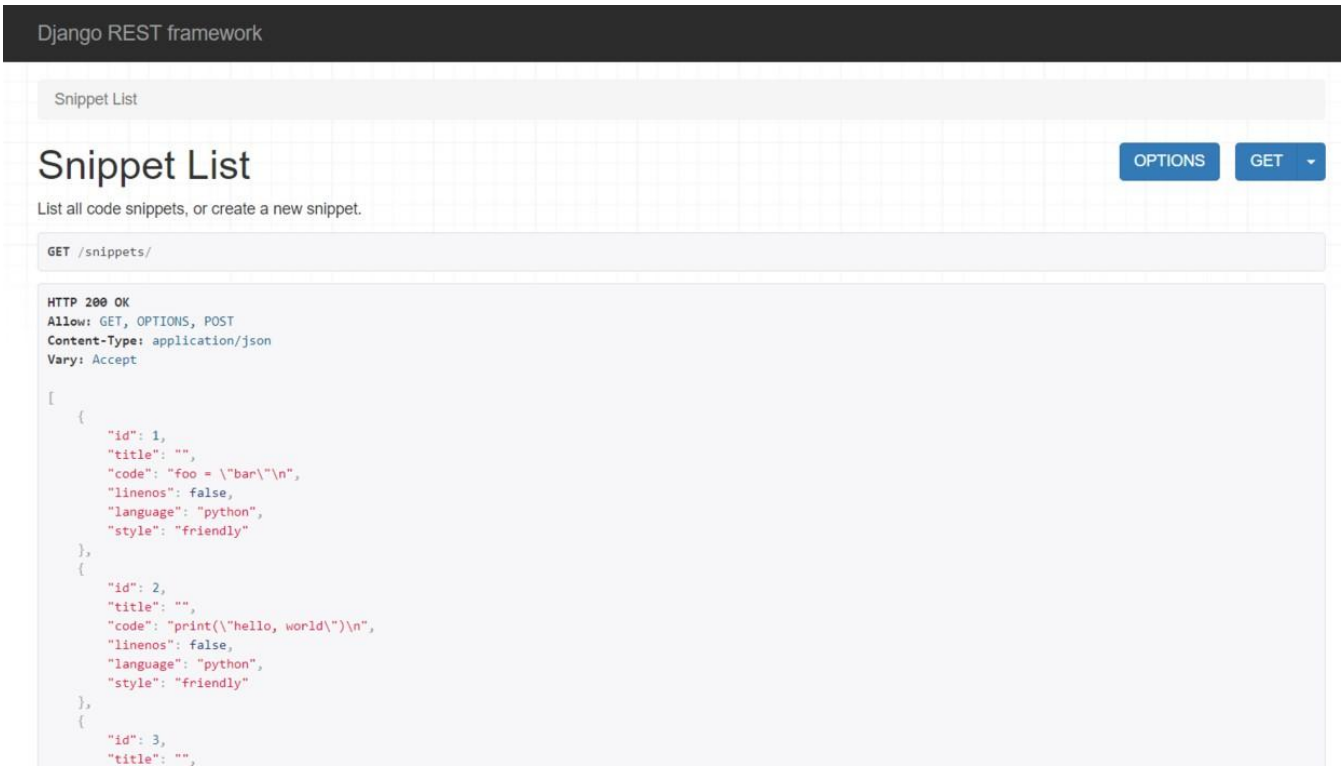
```
# POST表单数据
http --form POST http://127.0.0.1:8000/snippets/ code="print 123"

HTTP/1.1 201 Created
Allow: GET, OPTIONS, POST
Content-Length: 93
Content-Type: application/json
Date: Sun, 28 Apr 2019 04:35:02 GMT
Server: WSGIServer/0.2 CPython/3.7.3
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "code": "print 123",
    "id": 4,
    "language": "python",
    "linenos": false,
    "style": "friendly",
"title": "" 20
}


# POST JSON数据
http --json POST http://127.0.0.1:8000/snippets/ code="print 456"

HTTP/1.1 201 Created
Allow: GET, OPTIONS, POST
Content-Length: 94
Content-Type: application/json
Date: Sun, 28 Apr 2019 04:35:39 GMT
Server: WSGIServer/0.2 CPython/3.7.3
```

```
32  Vary: Accept, Cookie
33  X-Frame-Options: SAMEORIGIN
34
```

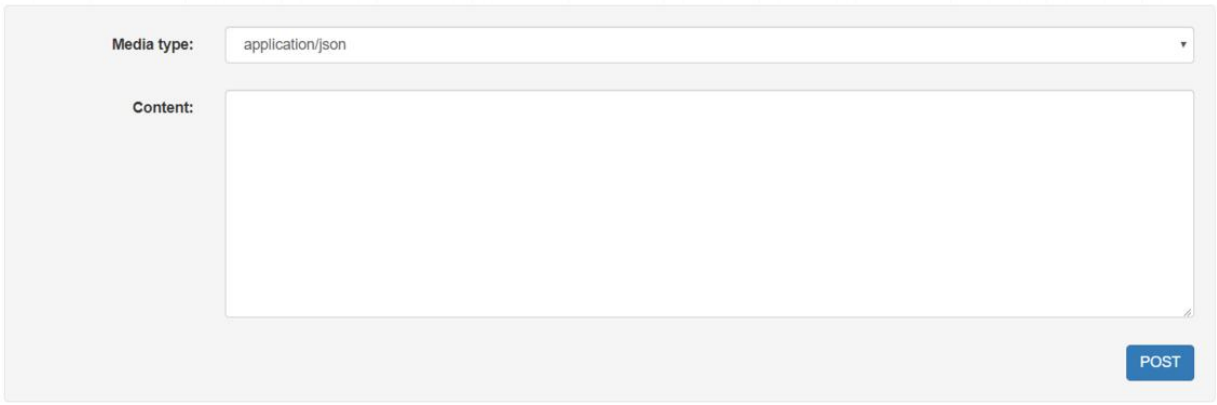
```
35    {
36        "code": "print(456)",
37        "id": 5,
38        "language": "python",
39        "linenos": false,
40        "style": "friendly",
41        "title": ""
42    }
```

如果你向上述 `http` 请求添加 `--debug` 参数，则可以在请求标头中查看更详细的内容。

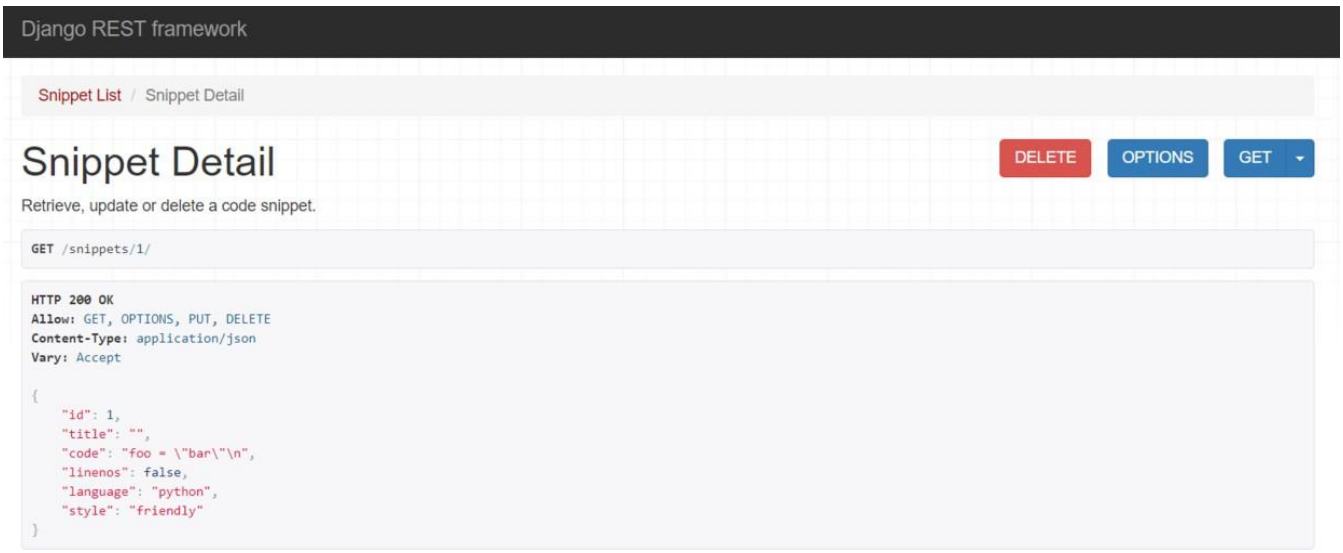
最关键的是：现在可以在浏览器中访问 `http://127.0.0.1:8000/snippets/` ，可以看到下面的页面：



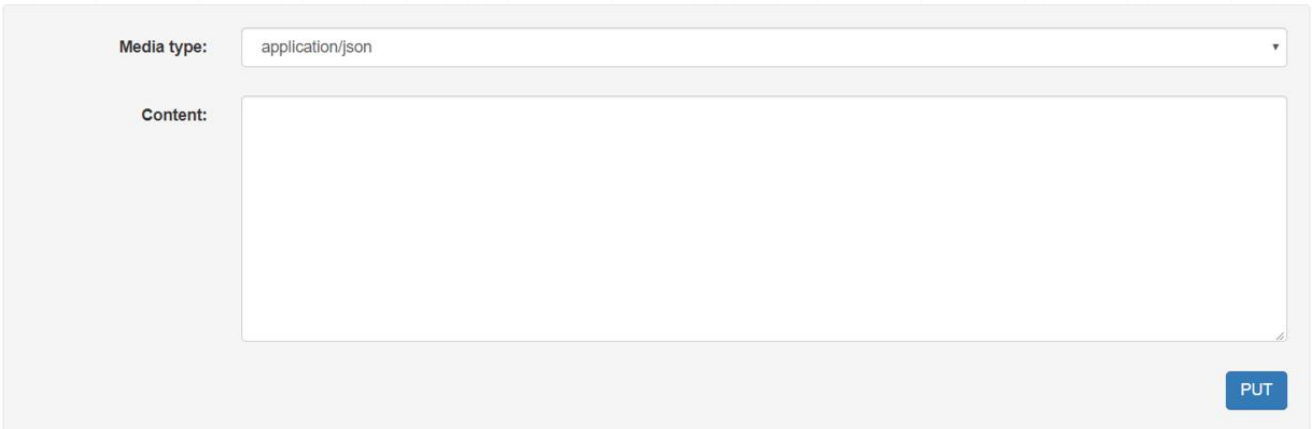
并且下面还有一个可以创建新sinppet的表单：



同样，访问 `http://127.0.0.1:8000/snippets/1/` ，会看到下面的序号为1的snippet的具体内容：



并且也有一个更新内容的表单：



你可能会疑惑，这么高端大气的页面怎么来的，我们没有编写这个页面HTML啊。这是DRF安利

给我们的，内置的。

DRF的API视图会根据客户端请求的响应内容类型返回对应类型的数据，因此当Web浏览器请求snippets时，它实际请求的是HTML格式，而不是json等格式，API会按要求返回HTML格式的表示，这个过程是DRF早就写好了的，在源码中实现了的。

DRF的这个功能，比较类似Django的admin后台的理念，大大降低了开发人员检查和使用API的障碍，可视化后更直接、更清晰、更便捷。

05-快速入门3--基于类的视图

在API视图的编写方法上，DRF为我们提供了很多种选择，比如基于类的视图。这是一个强大的模式，允许我们重用常用的功能，并帮助我们保持代码的DRY特性。

一、使用基于类的视图重写我们的API

我们又将再一次重写 `views.py`，而且还会有下一次。代码如下：

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from django.http import Http404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status


class SnippetList(APIView):
    """
    List all snippets, or create a new snippet.
    """
    def get(self, request, format=None):
        snippets =  Snippet.objects.all()
        serializer = SnippetSerializer(snippets,  many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = SnippetSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

是在APIView的dispath函数判断的GET请求，就会调用get方法

注意类视图的名字规范，和基于函数的视图是不一样的，所以这里换了名字。

在原生的Django中编写类视图是这样的：

https://docs.djangoproject.com/zh-hans/2.2/topics/class-based-views/

使用GenericAPIView产生新增修改的多行的模板

```
1
2
3   from django.http import HttpResponse
4   from django.views import View
5
6   class MyView(View):
7       def get(self, request):
8           # <view logic>
9           return HttpResponse('result')
10      def post(self,request):
11          # <view logic>
12          return  HttpResponse('something')
```

比较一下区别，最主要的是我们导入了这个类：`from rest_framework.views import APIView`，在自己写的视图类中继承它。那么这个APIView是什么呢？它是DRF对`Django.view.View`类的高级封装，添加了很多DRF需要使用的特性，比如可浏览的API页面等。

当然，我们还需要更新`views.py`中的detail实例视图。

```
1   class SnippetDetail(APIView):
2       """
3       Retrieve, update or delete a snippet instance.
4       """
5       def get_object(self, pk):
6           try:
7               return Snippet.objects.get(pk=pk)
8           except Snippet.DoesNotExist:
9               raise Http404
10
11      def get(self, request, pk, format=None):
12          snippet = self.get_object(pk)
13          serializer = SnippetSerializer(snippet)
14          return Response(serializer.data)
15
16      def put(self, request, pk, format=None):
17          snippet = self.get_object(pk)
18          serializer = SnippetSerializer(snippet, data=request.data)
19          if serializer.is_valid():
20              serializer.save()
21              return Response(serializer.data)
22          return Response(serializer.errors,
    status=status.HTTP_400_BAD_REQUEST)
```

```
23
24      def delete(self, request, pk, format=None):
25          snippet = self.get_object(pk)
26          snippet.delete()
27          return  Response(status=status.HTTP_204_NO_CONTENT)
```

看起来不错，但它现在仍然非常类似于基于函数的视图。（其实就是拆分了逻辑，实现可重用）

接下来，我们还需要重构我们的 `snippets.urls.py` ，因为Django对类视图有专门的url编写格式，不得不改：

```
1   # 注释了前面的内容，供大家对比参考
2   # from django.urls import path
3   # from snippets import views
4   #
5   # urlpatterns = [
6   #     path('snippets/', views.snippet_list),
7   #     path('snippets/<int:pk>/',  views.snippet_detail),
8   # ]
9
10
11  from django.urls import path
12  from rest_framework.urlpatterns import format_suffix_patterns
13  from snippets import views
14
15  urlpatterns = [
16      path('snippets/', views.SnippetList.as_view()),
17      path('snippets/<int:pk>/', views.SnippetDetail.as_view()),
18  ]
19
20  urlpatterns = format suffix patterns(urlpatterns)
```

http://192.168.1.111:8000/snippets.json

为什么使用`format_suffix_patterns`，请看

https://www.django-rest-framework.org/api-guide/format-suffixes/

好了，阶段完成，如果你重新启动服务器，那么它应该像之前一样运行。

视图类中的as_view方法会判断当请求的method是get时，就会调用get方法，是post，就调用post方法

二、使用混合类（mixins）

等等，你以为DRF的视图体系到此就完了吗？你想得太简单了！一大波内容还在后面.....

使用基于类的视图，最大优势之一是创建可复用的代码。

到目前为止，我们使用的创建/获取/更新/删除操作中有一部分代码是非常类似的，完全可以抽象出来。DRF就这么干了，并把这部分代码放到mixin类系列中，然后作为父类供子类继承复用。

让我们来看看我们是如何通过使用mixin类编写视图的。打开 `views.py` 模块，又要重写这个文件了…：

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import mixins
from rest_framework import generics

class SnippetList(mixins.ListModelMixin,
                  mixins.CreateModelMixin,
                  generics.GenericAPIView):
    queryset = Snippet.objects.all()
    serializer_class =  SnippetSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args,  **kwargs)

    def post(self, request, *args, **kwargs):  return
        self.create(request, *args,  **kwargs)
```

我们分析下这里的具体实现方式。我们自己写的SnippetList类继承了三个类，其中两个是mixin类，最后是 `GenericAPIView` 类。 `GenericAPIView` 类作为结构主父类，提供了基本的DRF 的API类视图的功能， `GenericAPIView` 类直接继承了我们前面使用的 `APIView` 类。而`ListModelMixin` 和 `CreateModelMixin` 提供 `.list()` 和 `.create()` 操作。

> 另外强调一点Python多继承的特点，继承父类的先后位置关系是有意义的，不可以随意调换顺序。

然后我们明确地将 `get` 和 `post` 方法绑定到适当的操作。

再修改一下我们的Detail类：

```
class SnippetDetail(mixins.RetrieveModelMixin,
                    mixins.UpdateModelMixin,
                    mixins.DestroyModelMixin,
                    generics.GenericAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer

```

```
    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args,  **kwargs)

    def put(self, request, *args, **kwargs):
        return self.update(request, *args,  **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args,  **kwargs)
```

和上面的那个类非常相似。

三、使用通用的基于类的视图

> 看完上面的内容，感觉又get到了新知识！以后就这么干了！

> 等等，DRF实际上又帮我们抽象了上面的代码，提供了一些通用的类似的类视图。 那你前面还跟我啰嗦那么多？直接使用下面的方法就好了啊！

通过使用mixin类，我们使用更少的代码重写了这些视图，但我们还可以再进一步。REST框架提供了一组已经混合好（mixed-in）的通用的类视图，我们可以使用它来简化我们的 `views.py` 模块。

已经不记得是第几次修改了....

```
from snippets.models import  Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics


class SnippetList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer


class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
```

你所需要做的，只是继承 `generics` 模块中的现成的某个通用类视图，比如 `ListCreateAPIView` ，`queryset` ，`serializer_class`

。然后在类里定义　　和　　两个属性的值，剩下的什么都不用写，全部交给DRF，它会帮你搞定。