**Flask**

**flask简介**

Python Flask框架是用于构建Web应用程序的轻量级框架。它提供了一个简单灵活的方式来处理HTTP请求、路由URL、渲染模板、处理表单数据等常见的Web开发任务。它结合了MVC（模型-视图-控制器）模式，适合于中小型网站或Web服务的开发。Flask自身不包含一些常见框架的默认功能，如表单验证，而是提供基础组件，让开发者根据项目需求自由配置。Flask的核心功能由[Werkzeug（路由模块）](https://www.baidu.com/s?wd=Werkzeug&usm=2&ie=utf-8&rsv_pq=b3f4f4b600c1ca00&oq=python%E7%9A%84flask%E6%A1%86%E6%9E%B6%E5%B9%B2%E5%95%A5%E7%9A%84&rsv_t=c2d2tlPOyPzy6RNhJO7QggnywIRSfYoTHFhIZaWmtl9VQ5NJSpI1zLlEyNw&sa=re_dqa_zy&icon=1" \t "_self)和[Jinja2（模板引擎）](https://www.baidu.com/s?wd=Jinja2&usm=2&ie=utf-8&rsv_pq=b3f4f4b600c1ca00&oq=python%E7%9A%84flask%E6%A1%86%E6%9E%B6%E5%B9%B2%E5%95%A5%E7%9A%84&rsv_t=21d7%2FoL3g8fNjyBtQNGaV1bS5TeyeAymw0p%2BT45bkI81BSsKOoeJeKRnhUQ&sa=re_dqa_zy&icon=1" \t "_self)两个库提供，Werkzeug负责处理业务逻辑和安全功能，而Jinja2是一个灵活的模板引擎，支持自动HTML转义，提高安全性。

Flask可用于构建各种Web应用程序，包括网站开发、系统开发、小程序等后端开发任务。并且Flask适合开发API服务，特别是当需要与不同的客户端（如移动应用、网页等）交互时。Flask还提供了对数据库的集成支持，如[SQLite](https://www.baidu.com/s?wd=SQLite&usm=2&ie=utf-8&rsv_pq=b3f4f4b600c1ca00&oq=python%E7%9A%84flask%E6%A1%86%E6%9E%B6%E5%B9%B2%E5%95%A5%E7%9A%84&rsv_t=21d7%2FoL3g8fNjyBtQNGaV1bS5TeyeAymw0p%2BT45bkI81BSsKOoeJeKRnhUQ&sa=re_dqa_zy&icon=1" \t "_self)、[MySQL](https://www.baidu.com/s?wd=MySQL&usm=2&ie=utf-8&rsv_pq=b3f4f4b600c1ca00&oq=python%E7%9A%84flask%E6%A1%86%E6%9E%B6%E5%B9%B2%E5%95%A5%E7%9A%84&rsv_t=21d7%2FoL3g8fNjyBtQNGaV1bS5TeyeAymw0p%2BT45bkI81BSsKOoeJeKRnhUQ&sa=re_dqa_zy&icon=1" \t "_self)和[MongoDB](https://www.baidu.com/s?wd=MongoDB&usm=2&ie=utf-8&rsv_pq=b3f4f4b600c1ca00&oq=python%E7%9A%84flask%E6%A1%86%E6%9E%B6%E5%B9%B2%E5%95%A5%E7%9A%84&rsv_t=21d7%2FoL3g8fNjyBtQNGaV1bS5TeyeAymw0p%2BT45bkI81BSsKOoeJeKRnhUQ&sa=re_dqa_zy&icon=1" \t "_self)等，并且可以通过扩展模块来增强其功能，例如通过Flask-Login处理用户身份验证和会话管理。

Flask框架的主要用途包括：

**构建Web应用程序：**Flask提供了一个简单的方式来构建Web应用程序，并且可以轻松地扩展和定制。

**处理HTTP请求：**Flask框架可以处理各种类型的HTTP请求，例如GET、POST等，并根据请求的URL路径和参数执行相应的操作。

**路由URL：**Flask框架可以定义路由规则，将特定的URL路径映射到相应的处理函数上，以便进行处理和返回响应。

**渲染模板：**Flask框架可以使用模板引擎来生成动态内容，并将其与静态内容结合在一起返回给客户端。

**处理表单数据：**Flask框架提供了方便的方式来处理表单数据，包括验证输入、处理文件上传等。

1. **使用flask框架**

创建一个虚拟环境：

Mkdir myproject -> cd myproject -> py -3 -m venv myvenv

激活：

myvenv\Scripts\activate

安装：

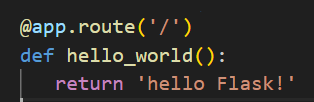
pip install flask

从flask包中导入Flask类:

from flask import Flask

创建一个Flask对象

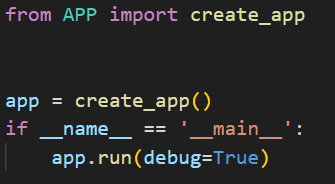
app = Flask(\_\_name\_\_)



@app.route: 是一个装饰器

@app.route('/')就是将url中 / 映射到hello\_world设个视图函数上面,访问网站的 / 目录的时候会执行hello\_world这个函数，然后将这个函数的返回值返回给浏览器。

启动这个WEB服务

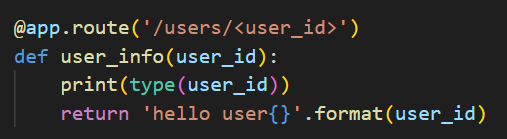


运行指令：flask run 打开调试模式：flask run –-debug

运行程序时，可以指定运行的主机IP地址，端口app.run(host="0.0.0.0", port=5000) ，例如127.0.0.1，可以不传，默认localhost，也可自己设置端口 (port=5000)

1. **URL与函数的映射**

比如，有一个请求访问的接口地址为 /users/11001 ，其中 11001 实际上为 具体的请求参数，表明请求 11001 号用户的信息。 此时如何从url中提取出 11001 的数据？



其中 ，尖括号是固定写法， variable 默认的 数据类型是字符串。 如果需要指定类型，则要写成 converter:variable ，其中 converter 就是类型名称，可以有以下几种：

**string:**如果没有指定具体的数据类型，那么默认就是使用 string 数据类型。

**int:**数据类型只能传递 int 类型。

**float:**数据类型只能传递 float 类型。

**path:**数据类型和 string 有点类似，都是可以接收任意的字符串，但是 path 可以接收路径，也就 是说可以包含斜杠。

**uuid:**数据类型只能接收符合 uuid 的字符串。 uuid 是一个全宇宙都唯一的字符串，一般可以用来 作为表的主键。

**any**:数据类型可以在一个 url 中指定多个路径。

1. **url\_for**

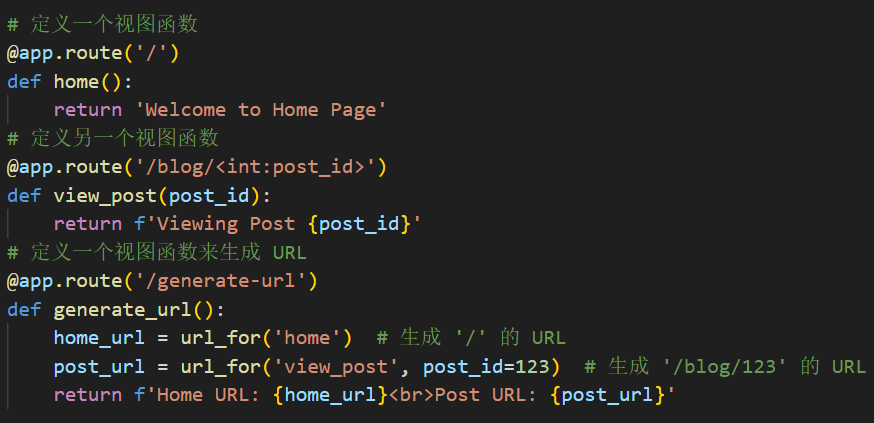
在 Flask 中，url\_for 是一个用于构建 URL 的辅助函数。它接受视图函数的名称作为参数，并返回与该视图函数对应的 URL。

url\_for 的使用有两个主要的优点：

**避免硬编码URL：** 使用 url\_for，您可以在代码中使用视图函数的名称而不是硬编码的URL，这样可以增强代码的可维护性。如果您更改了路由规则或 URL 结构，只需修改路由配置，而不需要在所有使用该 URL 的地方进行手动更改。

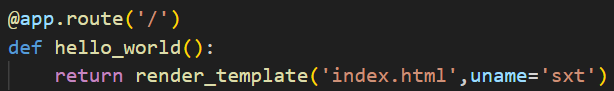
**支持反向路由：**url\_for 可以根据视图函数的名称和 URL 规则生成正确的 URL。这意味着您可以在应用程序中定义一次路由规则，然后使用 url\_for 来生成与该路由匹配的 URL，而无需手动在每个地方编写 URL。

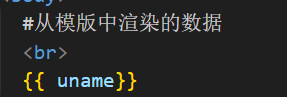
以下是一个使用 url\_for 的示例：



在上面的示例中，generate\_url 视图函数使用 url\_for 来生成两个 URL：主页的 URL 和带有特定文章 ID 的博客文章 URL。这样，无论您更改了路由规则或文章 ID，generate\_url 视图函数都会返回正确的 URL。

1. **模板传参**



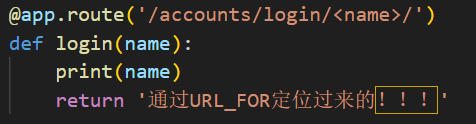


如果你的参数项过多，那么可以将所有的参数放到一个字典中， 然后在传这个字典参数的时候，使用两个星号，将字典打散成关键字参数（也叫命名参数）。



获取方式是： {{childrens.name}} 或者 {{childrens['name']}}

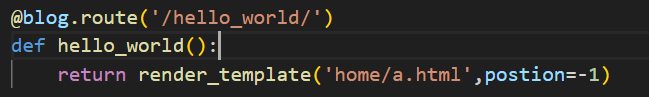
模版中也可使用 url\_for ,和后台视图函数中的 url\_for 使用起来基本是一模一样的。无论是路径参数还是查询式参数都可以直接传递

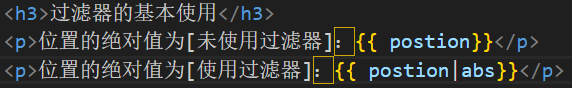




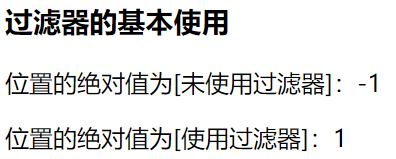
1. **过滤器**

有时候我们想要在模版中对一些变量进行处理，那么就必须需要类 似于Python中的函数一样，可以将这个值传到函数中，然后做一些操作。 在模版中，过滤器相当于是一个函数，把当前的变量传入到过滤器中，然后过滤器根据自己的功能，再返回相应的值，之后再将结果渲染到页面中





结果:



Jinja模板自带过滤器,常见有（abs,int,float.string,default,safe）等,也可自定义过滤器。

1. **循环结构**

列表：

<ul>

{% for user in users%}

<li>{{ user}}</li>

{% endfor %}

</ul>

字典：

<tr>

{% for key in person.keys() %}

<td>{{ key}}</td>

{% endfor %}

</tr>

<tr>

{% for val in person.values() %}

<td>{{ val}}</td>

{% endfor %}

</tr>

如果序列中没有值的时候，进入else 反向遍历用过滤器 reverse：

<ul>

{% for user in users|reverse %}

<li>{{ user}}</li>

{% else %}

<li>没有任何用户</li>

{% endfor %}

</ul>

在 jinja2 中的 for 循环，跟 python 中的 for 循环基本上是一模一样的 也是 for…in… 的形式。并且也可以遍历所有的序列以及迭代器，唯一不同的是： **jinja2 中的 for 循环没有 break 和 continue 语句。**

1. **导入模板**

1. 这个标签相当于是直接将指定的模版中的代码复制粘贴到当前位置。

2. include 标签，如果想要使用父模版中的变量，直接用就可以了，不需要使用 with context 。

3. include 的路径，也是跟 import 一样，直接从 templates 根目录下去找，不要以相对路径去找。

例子：

<body>

<!--通过include 引入头部log信息-->

{% include "common/head.html" %}

<div>

这是首页内容

{{ major }}

</div>

<hr>

<!--通过include 引入底部版权信息-->

{% include "common/footer.html" %}

</body>

1. **Set和with标签**

在模版中，可以使用 set 语句来定义变量：

<!--set语句来定义变量，之后，那么在后面的代码中，都可以使用这个变量-->

{% set uname='sxt'%}

<p>用户名：{{ uname }}</p>

一旦定义了这个变量，那么在后面的代码中，都可以使用这个变 量，就类似于Python的变量定义是一样的。

with 语句定义的变量，只能在 with 语句块中使用，超过了这个代码 块，就不能再使用了。

<!--with语句来定义变量，只有在指定区域 才能使用这个变量-->

{% with classroom='python202'%}

<p>班级：{{ classroom }}</p>

{% endwith %}

注意：

关于定义的变量， with 语句也不一定要跟一个变量， 可以定义一个空的 with 语句， 需要在指定的区域才能使用的情况，可以set与with组合使用。

{% with %}

{% set pname='lili' %}

<p>娱乐：{{ pname }}</p>

{% endwith %}

1. **Flask视图**

在 Flask 中，有两种常用的方式来定义视图函数：**add\_url\_rule** 和 **app.route。**

add\_url\_rule 方法：add\_url\_rule 是 Flask 应用程序对象的方法，用于通过显式地调用该方法来添加路由规则和视图函数的映射关系。

以下是一个使用 add\_url\_rule 的示例：

from flask import Flask

app = Flask(\_\_name\_\_)

# 添加路由规则和映射的视图函数

def hello():

return 'Hello, Flask!'

# 使用 add\_url\_rule 方法添加路由规则和视图函数的映射

app.add\_url\_rule('/hello', 'hello', hello)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()  
 app.route 方法则是前面介绍例子的写法，这两种方法都可以用来定义 Flask 应用程序的路由和视图函数映射关系，但是 app.route 装饰器方法更为常用和简便，因为它可以直接应用于视图函数，使代码更加简洁易读。

**类视图**



类视图的好处：

1.可以继承，把一些共性的东西抽取出来放到父视图中，子视图直接拿来用就可以了。

2.但是也不是说所有的视图都要使用类视图，这个要根据情况而定。视图函数用得最多。

**基于调度方法的类视图**

class LoginView(views.MethodView):

def get(self,error=None):

return render\_template('login.html',error=error)

def post(self):

uname = request.form['uname']

pwd = request.form['pwd']

if uname=="sxt" and pwd =="123":

return render\_template('index.html')

else:

return self.get(error="用户名或者密码错误")

# 注册类视图

app.add\_url\_rule('/login/',view\_func=LoginView.as\_view('my\_login'))

url方式的写法，需要在url.py中注册：

class VueView(Resource):

def get(self):

data = CatalogueModel.query.filter\_by(category='vue3').all()

return render\_template('home/vue.html',data=data)

def post(self):

try:

u = CatalogueModel()

u.name = request.form.get('name')

u.category = request.form.get('category')

u.file\_text = request.form.get('file\_text')

u.img = request.form.get('img')

db.session.add(u)

db.session.commit()

return 'success'

except Exception as e:

return e

在 Flask-RESTful 中，Resource 是一个基础类，用于创建 RESTful API 的资源类。

Resource 类提供了方便的方法和函数，帮助您定义和处理 API 资源相关的请求。它简化了处理不同 HTTP 方法（如 GET、POST、PUT、DELETE 等）的逻辑，并提供了一种组织和管理资源的方式。

**装饰器：**

定义装饰器:

def login\_required(func):

@wraps(func)

def wrapper(\*arg,\*\*kwargs):

uname = request.args.get('uname')

pwd = request.args.get('pwd')

if uname == 'zs' and pwd == '123':

logging.info(f'{uname}:登录成功')

return func(\*arg,\*\*kwargs)

else:

logging.info(f'{uname}:尝试登录，但没成功')

return '请先登录'

return wrapper

使用装饰器：

@app.route('/settings/')

@login\_requierd

def settings():

return '这是设置界面'

1. **蓝图**

在Flask中，使用蓝图Blueprint来分模块组织管理。蓝图实际可以理解为是存储一组视图方法的容器对象，蓝图的基本概念是：在蓝图被注册到应用之后，所要执行的操作的集合。当分配 请求时，Flask会把蓝图和视图函数关联起来，并生成两个端点之前的URL。其具有如下特点：

1．一个应用可以具有多个Blueprint。

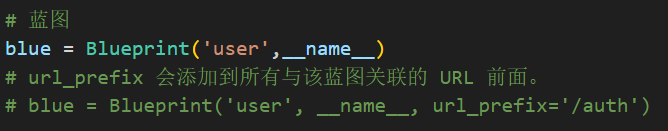
2．可以将一个Blueprint注册到任何一个未使用的URL下比如 “/user”、 “/goods”。

3．Blueprint可以单独具有自己的模板、静态文件或者其它的通用操作方法，它并不是必须要实现应 用的视图和函数的。

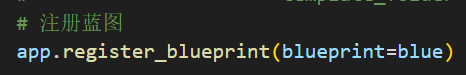
4．在一个应用初始化时，就应该要注册需要使用的Blueprint。

注意：Blueprint并不是一个完整的应用，它不能独立于应用运行，而必须要注册到某一个应用中

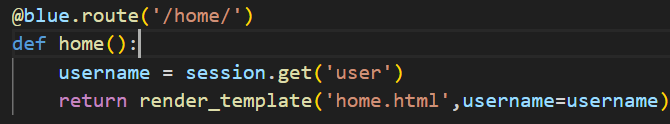
创建蓝图：



注册蓝图：



使用蓝图：



1. **Flask其它**

Flask\_线程隔离的g对象

保存为全局对象g对象的好处：

g对象是在整个Flask应用运行期间都是可以使用的。 并且也跟request一样，是线程隔离的。 这个对象是专门用来存储开发者自己定义的一些数据，方便在整个Flask程序中都可以使用。 一般使用就是，将一些经常会用到的数据绑定到上面，以后就直接从g上面取就可以了，而不需要通过传参的形式，这样更加方便，在 Flask 中，g对象是一个与每个客户端请求相关的线程隔离的全局对象。它允许您在请求处理过程中共享数据，而不必将数据作为参数传递给每个函数或方法。

以下是一个简单的示例，展示了如何在 Flask 中使用 g 对象：



1. **Flask钩子函数**

在 Flask 中，钩子函数是在请求处理过程中执行的特殊函数，它们允许您在请求的不同阶段插入自定义的逻辑。

以下是一些常见的 Flask 钩子函数：

**before\_request：**在每个请求开始之前执行。可以用于执行一些初始化任务，如设置数据库连接、身份验证等。

**after\_request：**在每个请求结束之后执行。可以用于对响应进行处理，如添加头信息、修改响应数据等。

**before\_first\_request：**在第一个请求到达时执行，用于执行应用程序级别的初始化任务，如加载配置、创建数据库表等。

**teardown\_request：**在每个请求结束后执行，无论是否发生异常。用于清理资源，如关闭数据库连接、释放内存等。

以下是一个使用钩子函数的简单示例：



before\_request：在每个请求之前打印 "Before Request"。

after\_request：在每个请求之后打印 "After Request"。

index：处理根路径的请求，打印 "Processing Request" 并返回一个简单的响应信息。

通过运行这个示例应用程序，您会看到在访问根路径时，钩子函数的输出顺序如下：

Before Request

Processing Request

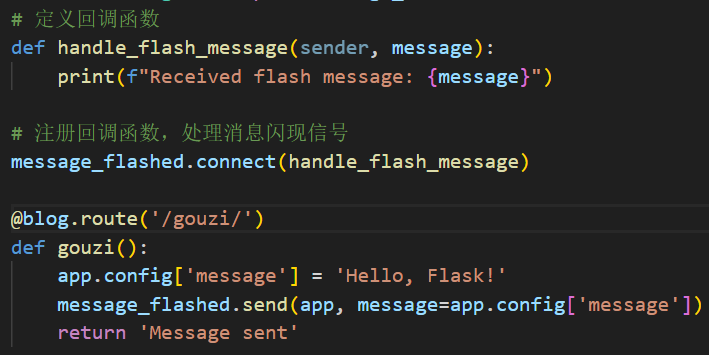
After Request

1. **flask信号机制**

在 Flask 中，信号是一种事件机制，允许您在应用程序中的特定事件发生时触发回调函数。

Flask 使用 Werkzeug 库提供的信号机制，可以让您在应用程序的不同阶段注册回调函数，以响应特定的事件。

以下是一个简单的例子，展示了如何在 Flask 中使用信号：



使用 message\_flashed 信号定义了一个回调函数 handle\_flash\_message。然后，我们通过 message\_flashed.connect() 方法将回调函数与 message\_flashed 信号关联起来。

在根路径的视图函数中，我们设置了一个message 配置变量，并通过 message\_flashed.send() 方法发送了一个消息闪现信号，并传递了相应的消息。

当应用程序运行时，当发送消息闪现信号时，回调函数会被触发，并打印收到的消息。

1. **Flask内置信号**

Flask内置了10个常用的信号：

1. template\_rendered：模版渲染完成后的信号。

2. before\_render\_template：模版渲染之前的信号。

3. request\_started：请求开始之前，在到达视图函数之前发送信号。

4. request\_finished：请求结束时，在响应发送给客户端之前发送信号。

5. request\_tearing\_down：请求对象被销毁时发送的信号，即使在请求过程中发生异常也会发送信号。

6. got\_request\_exception：在请求过程中抛出异常时发送信号，异常本身会通过exception传递到订 阅（监听）的函数中。一般可以监听这个信号，来记录网站异常信息。

7. appcontext\_tearing\_down：应用上下文被销毁时发送的信号。

8.appcontext\_pushed：应用上下文被推入到栈上时发送的信号。

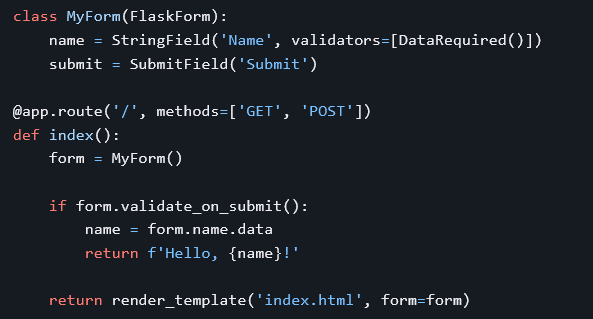
9. appcontext\_popped：应用上下文被推出栈时发送的信号。

10. message\_flashed：调用了Flask的 flash 方法时发送的信号。

1. **WTForms表单验证的基本使用**

WTForms 是一个用于处理表单验证和数据渲染的 Python 库。

基本的使用示例：



当用户提交表单时，使用 form.validate\_on\_submit() 验证表单数据是否有效。如果有效，提取姓名字段的数据，并返回欢迎消息。如果表单未通过验证，则继续渲染表单页面。

1. **WTForms常用验证器**

页面把数据提交上来，需要经过表单验证，进而需要借助验证器来 进行验证，以下是常用的内置验证器：

**1. Length：**字符串长度限制，有min和max两个值进行限制。

username = StringField(validators=[Length(min=3,max=10,message="用户名长度必须在3到10位之间")])

**2. EqualTo**：验证数据是否和另外一个字段相等，常用的就是密码 和确认密码两个字段是否相等。

password\_repeat=StringField(validators=[Length(min=6,max=10),EqualTo("password")])

**3. Email：**验证上传的数据是否为邮箱数据格式 如：223333@qq. com。

email = StringField(validators=[Email()])

**4. InputRequired：**验证该项数据为必填项，即要求该项非空。

username = StringField(validators=[input\_required()])

**5. NumberRange：**数值的区间，有min和max两个值限制，如果 处在这两个数字之间则满足。

age = IntegerField(validators=[NumberRange(12,18)])

**6. Regexp：**定义正则表达式进行验证，如验证手机号码。

phone = StringField(validators=[Regexp(r'1[34578]\d{9}')])

**7. URL：**必须是URL的形式 如http://www.bjsxt.com。

home\_page = StringField(validators=[URL()])

**8. UUID：**验证数据是UUID类型。

uuid = StringField(validators=[UUID()])

**机器学习**

**第3章 线性回归**

线性回归是一种用于建立模型和预测连续输出的监督学习方法。

线性回归模型通过最小化预测值与实际值之间的差异来找到最佳拟合直线。

最小二乘法是一种常用的解决线性回归问题的方法，通过最小化残差平方和来确定模型参数。

线性回归模型的评估指标包括平均绝对误差（MAE）、均方误差（MSE）、均方根误差（RMSE）等。

**第4章 k近邻算法**

k近邻算法是一种基于样本相似度的监督学习方法。

k近邻算法通过计算样本之间的距离来确定最接近的k个邻居，根据邻居的标签进行分类或预测。

距离度量可以使用欧氏距离、曼哈顿距离等。

k值的选择对算法的性能有重要影响，较小的k值容易受到噪声影响，较大的k值容易产生类别不平衡。

**第5章 朴素贝叶斯算法**

朴素贝叶斯算法是一种基于贝叶斯定理和特征独立性假设的监督学习方法。

朴素贝叶斯根据训练样本的特征和标签计算概率，通过最大化后验概率进行分类或预测。

朴素贝叶斯算法较简单且具有较好的性能，在文本分类、垃圾邮件过滤等领域有广泛应用。

**第6章 决策树算法**

决策树算法是一种基于树结构的监督学习方法。

决策树通过将数据集分成越来越小的子集来构建树，每个节点是一个特征，每个分支代表一个特征值，叶子节点表示结果。

决策树的构建可以使用不纯度或信息增益等指标来选择最佳特征进行分割。

决策树算法具有解释性强、易于理解和实现等优点，但容易过拟合。

**第7章 支持向量机算法**

支持向量机（SVM）是一种用于分类和回归的监督学习方法。

SVM通过在特征空间中找到一个超平面来最大化类之间的间隔，从而进行分类或预测。

SVM可以使用不同的核函数来处理非线性问题，如线性核、多项式核、径向基核等。

SVM的优化问题可以转化为凸二次规划问题，可以通过有效的优化算法求解。

**在TensorFlow中对应的一些处理：**

**数据预处理：**在使用机器学习算法之前，通常需要对数据进行预处理，例如特征缩放、数据标准化、特征选择等。TensorFlow提供了各种数据处理工具和函数，可以在数据准备阶段使用。

**特征工程：**特征工程是指将原始数据转换为更具有信息量的特征，以提升模型性能。在TensorFlow中，可以使用FeatureColumns进行特征的处理和组合，从而提供更好的输入数据给模型。

**模型构建：**使用TensorFlow可以方便地构建各种机器学习模型，包括线性回归、k近邻、朴素贝叶斯、决策树、支持向量机等。通过定义模型结构、损失函数和优化器，使用TensorFlow的API进行模型构建。

**模型训练**：TensorFlow提供了各种优化算法和训练方法，可以通过反向传播算法进行模型训练和参数更新。通过指定训练数据和目标变量，使用TensorFlow的优化器进行模型训练。

**模型评估：**在训练完成后，可以使用TensorFlow提供的评估指标，如平均绝对误差（MAE）、均方误差（MSE）、准确率、召回率等指标，对模型进行评估和性能分析。

**D R F**

**简介**

Django REST framework（DRF）是一个用于构建Web API的强大框架。DRF建立在Django基础上，提供了许多用于快速开发高质量Web API的功能和工具。

以下是一些DRF的核心功能：

1. 序列化（Serialization）：DRF提供了序列化器（Serializer）来处理数据的转换和验证。通过定义序列化器，可以将复杂的数据结构（如Django模型）转换为序列化的JSON、XML或其他格式，并反向将序列化数据转换为模型实例。
2. 视图（Views）：DRF提供丰富的视图类，支持一系列常见的API视图模式，例如基于类的视图（Class-based views）、视图集（Viewsets）和通用视图（Generic views）。这些视图类简化了API的开发，并提供了一些便捷的功能，如认证、权限控制、限流等。
3. 路由（Routing）：DRF的路由系统可以将URL与API视图进行映射。DRF支持传统的基于函数的路由和基于类的路由两种方式。通过路由系统，可以轻松地定义API的URL结构。
4. 认证和权限（Authentication & Permissions）：DRF提供了各种认证和权限类，用于在API中进行身份验证和权限控制。您可以根据需求选择合适的认证方式，如基于令牌的认证、基于Session的认证、OAuth认证等。同时，DRF也提供了灵活的权限控制机制，可以基于用户身份、对象所有权和其他条件来限制API的访问。
5. 响应（Response）：DRF提供了不同类型的响应类，用于返回API请求的数据。这些响应类包括常见的JSONResponse、XMLResponse等。DRF还支持内容协商（Content negotiation），它根据客户端的接受的内容类型来自动选择合适的响应类型。
6. 分页（Pagination）：DRF提供了实现分页的工具，用于处理大量数据的API响应。您可以选择不同类型的分页器，如基于游标的分页、基于页码的分页等。
7. 验证和数据校验（Validation & Data Validation）：DRF提供了丰富的验证器和校验方法，用于验证传入的数据是否符合规范。DRF的验证器可以用于字段级别的验证，也可以用于整体数据的校验。

Django REST framework（DRF）提供了丰富的功能和工具，使开发人员能够快速构建出高质量和强大的Web API。它是目前最受欢迎和广泛使用的Django框架的API开发扩展。

**一、序列化**

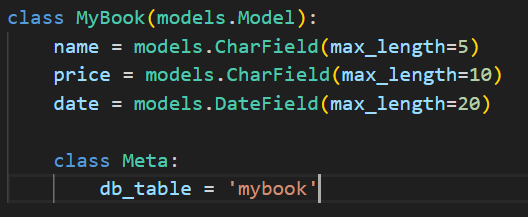
DRF中的序列化是将数据从 Python 对象转换为可传输或可存储的格式，通常是 JSON、XML 等格式。序列化在 API 中起着很重要的作用，它允许我们将复杂的数据结构转化为简单的数据格式，以便在网络上进行传输。

在 DRF 中，常用的序列化器有以下几种：

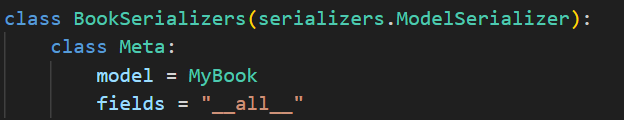
1. **ModelSerializer**：这是 DRF 提供的一个内置的序列化器，用于将 Django 模型转换为序列化的数据。它自动检测模型字段，并生成相应的序列化字段。
2. **Serializer**：这是一个通用的序列化器类，对于自定义序列化逻辑或序列化 Django 模型之外的其他对象非常有用。
3. **HyperlinkedModelSerializer：**它与 ModelSerializer 类似，但会生成超链接字段来代表模型之间的关系。

简单示例：

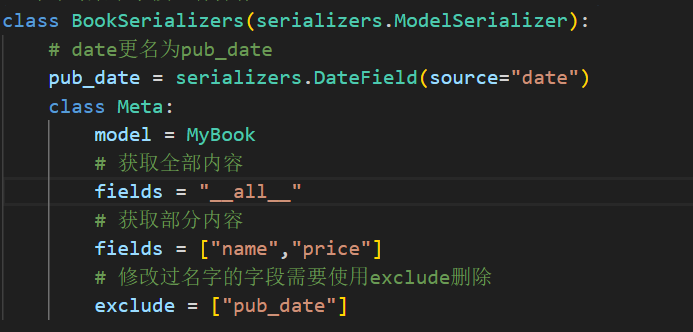
先创建一个模型



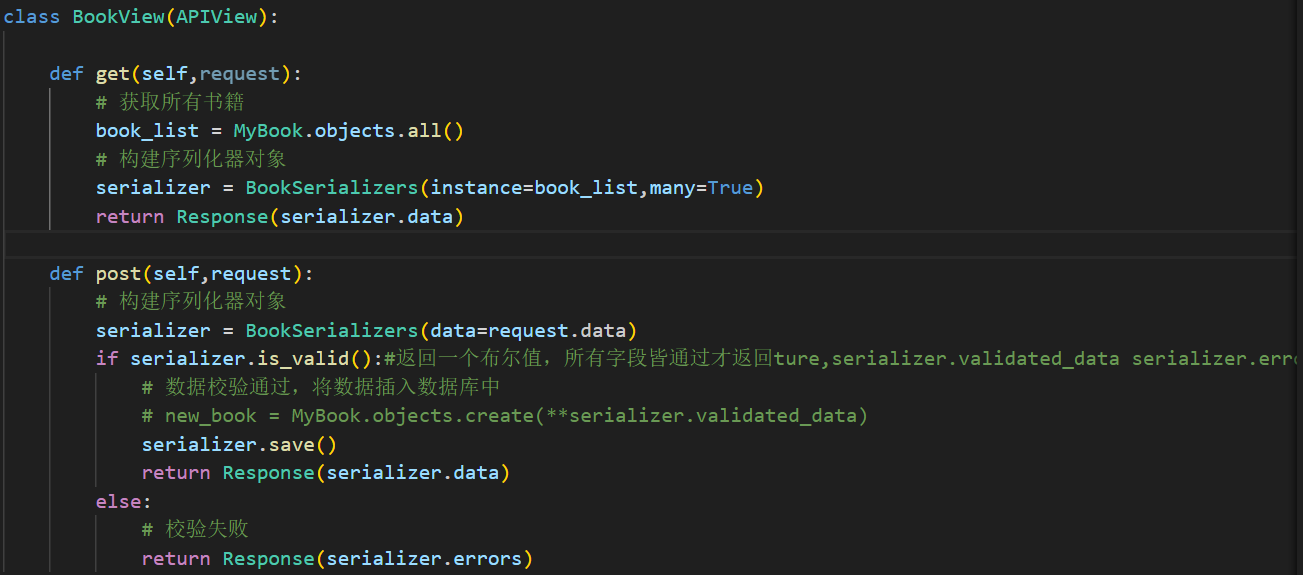
**ModelSerializer示例：**



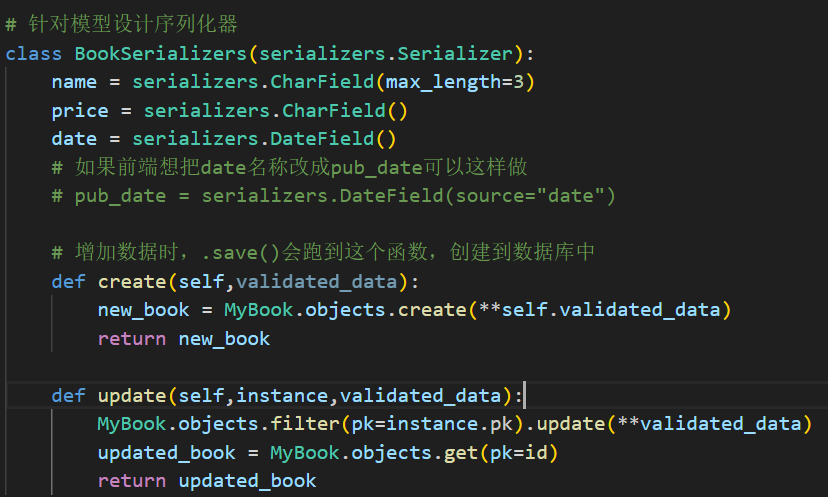
也可在针对模型中个别字段做操作：



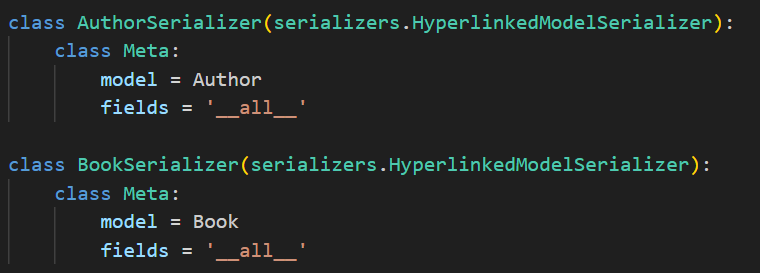
使用 ModelSerializer 进行序列化

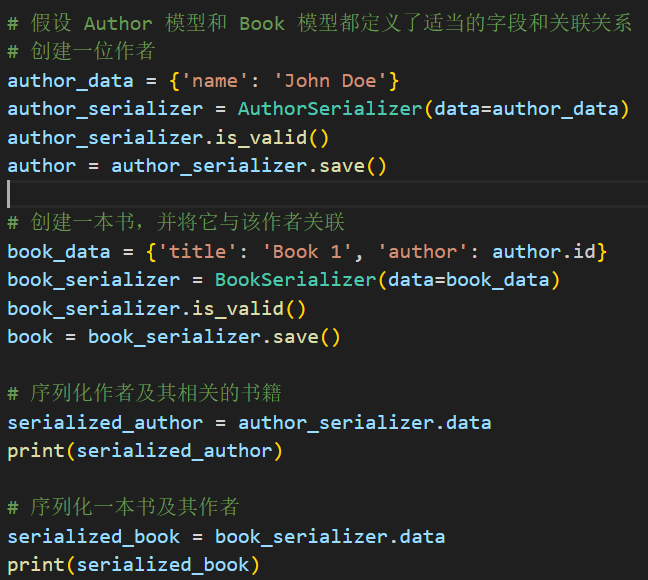


**Serializer 示例：**



**HyperlinkedModelSerializer 的示例：**





AuthorSerializer 和 BookSerializer 会自动创建超链接字段来代表模型之间的关联关系。例如，在 BookSerializer 中，如果定义了

author = serializers.HyperlinkedRelatedField(read\_only=True)

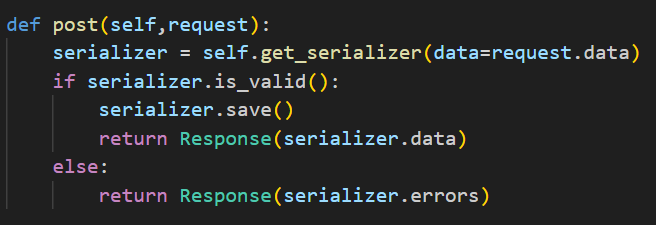
则会在序列化输出中包含一个名为 author 的超链接字段。序列化器会自动处理模型之间的关联关系，将其转化为超链接字段或其他适当的方式来表示。

**二、请求和响应**

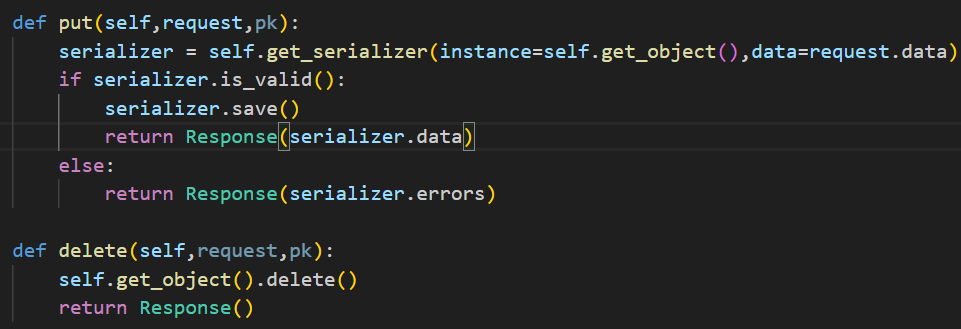
请求对象（Request objects）

REST框架引入了一个扩展了常规HttpRequest的Request对象，并提供了更灵活的请求解析。Request对象的核心功能是request.data属性，它与request.POST类似，但对于使用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'方法



响应对象（Response objects）

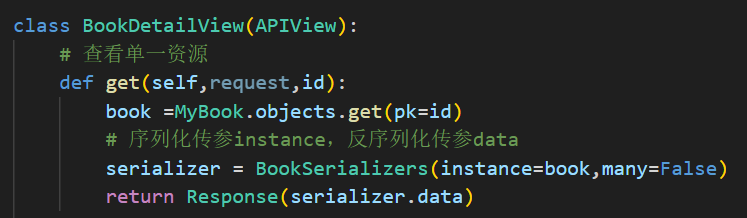
REST框架还引入了一个Response对象，这是一种获取未渲染（unrendered）内容的TemplateResponse类型，并使用内容协商来确定返回给客户端的正确内容类型。

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

**三、基于类的视图**

DRF 提供了多种类视图供开发者使用，以下是其中几种常用的类视图。

**1、APIView：**这是 DRF 提供的最基本的类视图，它允许开发者细粒度地控制 API 中每个 HTTP 方法的实现。可以通过方法重写来自定义每个方法的行为。



通过继承 APIView 类和自定义需要的方法，我们可以在视图中实现复杂的业务逻辑，并对每个 HTTP 方法的请求进行定制化的处理

**2、GenericAPIView：**这是 DRF 提供的一个通用类视图，它封装了大部分常见的 API 功能，如获取列表、创建、获取单个对象、更新和删除。



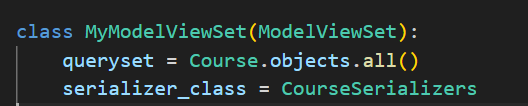
GenericAPIView 是 Django REST framework（DRF）提供的一个视图类，它是一个功能强大且灵活的通用视图。

GenericAPIView 类提供了许多常见的API视图功能和操作的默认实现，同时也可以让开发人员通过扩展和重载来自定义视图行为。

GenericAPIView 的一些常用功能：

1. **查询集（Queryset）**：GenericAPIView 提供了一个 queryset 属性，用于指定视图所使用的查询集合。这个查询集合可以是从数据库获取的对象列表或查询结果。
2. **序列化器（Serializer）**：通过指定 serializer\_class 属性，GenericAPIView 允许您定义与视图关联的序列化器类。序列化器负责将数据转换为合适的格式，以便在 API 响应中返回或在提交请求时进行数据验证。
3. **路由方法（HTTP Methods）**：GenericAPIView 为常见的 HTTP 方法（如 GET、POST、PUT、DELETE 等）提供了默认的实现。开发人员可以根据需要重写这些方法来自定义请求的处理逻辑。
4. **响应处理（Response handling）**：GenericAPIView 通过 Response 类提供了便捷的 HTTP 响应处理。您可以使用 Response 类来构建适合 API 返回的响应数据，例如序列化后的数据。
5. **认证和权限（Authentication & Permissions）**：GenericAPIView 提供了内置的认证和权限控制机制。您可以通过指定 authentication\_classes 和 permission\_classes 属性来指定所需的认证和权限。
6. **过滤器（Filtering）**：GenericAPIView 支持使用过滤器类来处理 API 数据的过滤。您可以指定 filter\_backend 属性来选择使用的过滤器类。

**3、ModelViewSet**：这是 DRF 提供的最为强大且便捷的类视图，它封装了常见的 CRUD 功能，同时还提供了额外的方法，如搜索、过滤、排序等，可用于处理与数据库模型相关的业务逻辑。



通过继承 ModelViewSet 类，我们自动获得了以下常见的方法和功能：

list：获取模型对象列表（GET 请求）。

create：创建新的模型对象（POST 请求）。

retrieve：获取单个模型对象的详细信息（GET 请求）。

update：更新单个模型对象的信息（PUT 请求）。

partial\_update：部分更新单个模型对象的信息（PATCH 请求）。

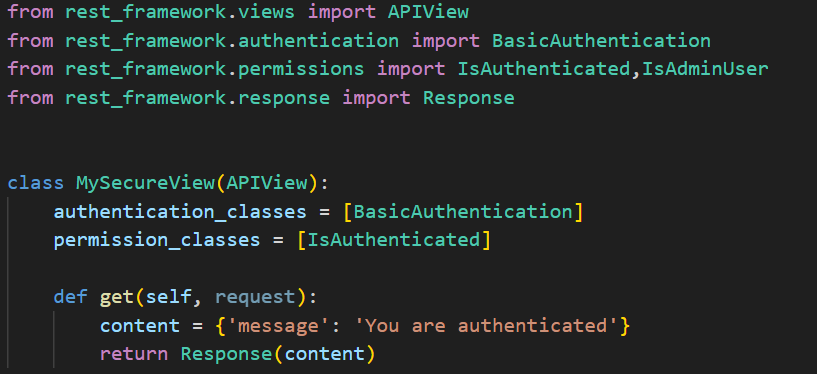
destroy：删除单个模型对象（DELETE 请求）。

除了以上常见的方法外，ModelViewSet 还提供了其他一些有用的方法和功能，如搜索、过滤、排序、分页等。它是一种非常强大和方便的类视图，适用于处理与数据库模型相关的 API 逻辑。

**四、认证和权限**

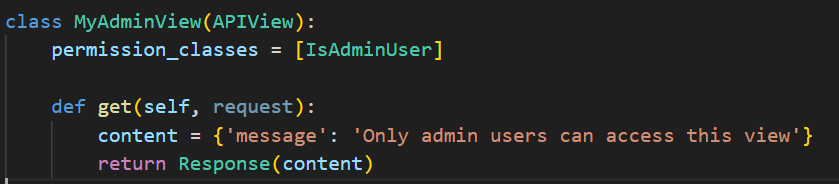
DRF 提供了认证（Authentication）和权限（Permissions）机制，用于在 API 中进行身份验证和权限控制。

认证（Authentication）示例：



定义了一个名为 MySecureView 的 APIView 类。在该类中，我们将 BasicAuthentication 设置为认证类，并将 IsAuthenticated 设置为权限类。这样，只有经过身份验证的用户才能访问视图中的 GET 请求。

权限（Permissions）示例：



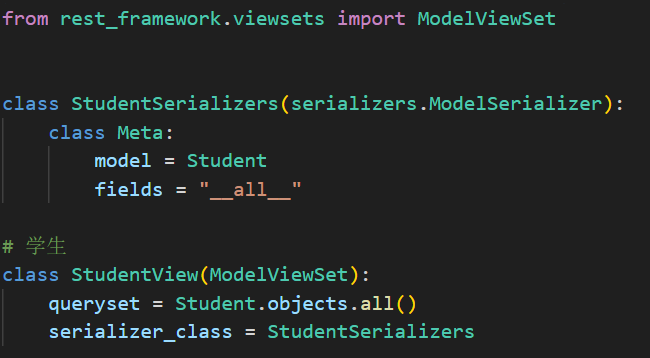
定义了一个名为 MyAdminView 的 APIView 类。将 IsAdminUser 设置为权限类，这样只有具有管理员权限的用户才能访问视图中的 GET 请求。

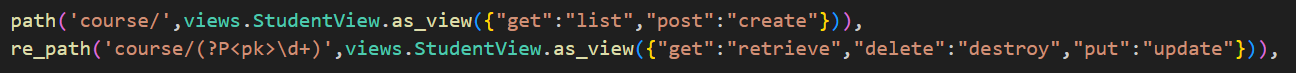
**五、ViewSet**

REST框架包括一个用于处理ViewSets的抽象，它允许开发人员集中精力对API的状态和交互进行建模，并根据常规约定自动处理URL构造。

ViewSet类与View类几乎相同，不同之处在于它们提供诸如read或update之类的操作，而不是get或put等方法处理程序。

最后一个ViewSet类只绑定到一组方法处理程序，当它被实例化成一组视图的时候，通常通过使用一个Router类来处理自己定义URL conf的复杂性。

  
路由控制：



"get":"list" ：指定 GET 请求将调用 CourseView 中名为 list 的方法，用于获取课程列表。

"post":"create" ：指定 POST 请求将调用 CourseView 中名为 create 的方法，用于创建新课程。

"get":"retrieve" ：指定 GET 请求将调用 CourseView 中名为 retrieve 的方法，用于获取特定课程的详细信息。

"delete":"destroy" ：指定 DELETE 请求将调用 CourseView 中名为 destroy 的方法，用于删除特定课程。

"put":"update" ：指定 PUT 请求将调用 CourseView 中名为 update 的方法，用于更新特定课程的信息。