flask + rest api

Flask 是一个轻量级的 Web 开发框架，基于 Python 编程语言。它旨在使 Web 应用程序的开发变得简单、快捷和灵活。Flask 的原理主要基于 WSGI（Web Server Gateway Interface）协议，这是 Python 用来连接 Web 服务器和 Web 应用程序的标准接口。在这个框架下，Flask 应用程序接收来自客户端的 HTTP 请求，并返回相应的 HTTP 响应。

Flask 的核心是一个 WSGI 应用程序，它可以处理各种类型的 HTTP 请求。当一个请求到达时，Flask 将根据请求的路径选择对应的视图函数来处理请求。视图函数是一个 Python 函数，它接收请求并返回一个响应。视图函数通常会读取请求中的参数，执行必要的处理逻辑，然后生成响应并返回给客户端。

Flask 使用路由（route）来将 URL 映射到视图函数。路由规则由装饰器指定，例如 `@app.route('/hello')` 将 URL 路径 `/hello` 映射到对应的视图函数。当客户端发送一个 HTTP 请求时，Flask 会根据请求的 URL 找到匹配的路由规则，并调用对应的视图函数来处理请求。

同时，Flask 支持模板渲染，使开发者能够在 HTML 页面中使用模板语言来动态生成页面内容。模板引擎负责将模板文件中的变量替换为实际数值，并渲染为最终的 HTML 页面。这使开发者能够更方便地向客户端提供动态内容，常常用于前端页面的制作。

此外，Flask 还提供了插件系统，使开发者能够扩展框架的功能。插件可以用来实现各种功能，如身份验证、数据库集成、缓存管理等。开发者可以通过安装插件来添加新的功能，并将其集成到应用程序中。

举例：

```
from flask import Flask, jsonify


app = Flask(__name__)


@app.route('/predict')
def predict():

    return jsonify({"file_name": "pad_001757.jpg", "height": 340.0,
"width": 191.0, "id": 0})
```

```
if __name__ == '__main__':
    app.run()
```

上述简单的例子运行之后，会在http://127.0.0.1:5000/predict中返回json值。

对于rest api，REST（Representational State Transfer）是一种基于网络的软件架构风格，用于设计可伸缩性的分布式系统。REST API（RESTful API）是基于REST架构风格设计的API，用于实现不同系统之间的通信和数据交换。

REST API的原理主要包括以下几个方面：

1. 资源和标识符：在REST API中，资源是系统中的任何实体或信息，如用户、产品、订单等。每个资源都有一个唯一的标识符（URI）用于访问和操作资源。通过使用标准的HTTP方法（GET、POST、PUT、DELETE等）对资源进行操作，实现了资源的统一管理和操作；
2. 数据传输和格式：REST API使用标准的HTTP协议来传输数据，并且通常使用JSON或XML等格式对数据进行编码和解码。通过使用这些标准格式，不同系统之间可以方便地进行数据交换和解析，实现了系统之间的互操作性；
3. 状态转移：REST API通过定义一组状态转移规则来控制资源的访问和操作。在REST API中，每个资源都有一组状态，通过执行HTTP方法来对资源的状态进行转移和变化。通过定义这些状态转移规则，实现了资源的完整和一致性管理；
4. 无状态性：REST API是一种无状态的通信协议，每个请求都是独立的，服务器不会保存任何客户端的状态信息。这意味着客户端必须在每个请求中包含所有必要的信息，如认证信息、参数等。通过保持无状态性，提高了系统的可伸缩性和可靠性；
5. 缓存：REST API支持缓存机制，客户端可以缓存资源的响应，减少对服务器的请求次数，提高系统的性能和效率。通过使用HTTP的缓存头来控制缓存策略，实现了资源的高效利用和管理；
6. 统一接口：REST API使用统一的接口来访问和操作资源，即使用HTTP标准方法和URI来表示资源和资源之间的关系。通过统一接口，实现了系统之间的分离和解耦，提高了系统的可扩展性和灵活性。

REST API 部署模型是指将 REST API 部署到不同的环境中，使得客户端可以通过网络访问和调用 API 提供的服务。在实际应用中，REST API 部署模型可以按照不同的需求和场景进行选择和配置，以实现最佳的性能和可用性。

REST API 的部署模型通常包括以下几种方式：

1. 单节点部署：将 REST API 部署到单个服务器节点上，通过单个域名或 IP 地址来提供服务。这种部署模型适用于小型应用或简单的服务，它具有简单、易于管理和部署的优点，但在高负载情况下可能会出现性能瓶颈和单点故障的问题；
2. 多节点部署：将 REST API 部署到多个服务器节点上，通过负载均衡器或反向代理来分发请求和实现负载均衡。这种部署模型可以提高系统的性能和可扩展性，同时增加了系统的可用性和容错能力，适用于大型应用或高负载情况；
3. 微服务架构：将 REST API 拆分为多个微服务，每个微服务负责实现特定的功能或服务，通过服务注册发现和 API 网关来实现服务之间的通信和交互。微服务架构可以提高系统的灵活性和可维护性，同时降低了系统的复杂性和耦合度，适用于复杂的应用或分布式系统；
4. 云原生应用：将 REST API 部署到云平台上，通过容器化和自动化部署工具来实现快速部署和弹性扩展，实现高可用性和灵活的资源管理。云原生应用可以大大简化部署和运维的复杂度，提高系统的可伸缩性和弹性，适用于动态变化和需求快速变化的场景。

rest api 模型部署举例：yolov5

这里使用yolov5的官方文件`yolov5-master`来对rest api的模型部署进行讲解。

主要使用的文件：`yolov5-master\utils\flask_rest_api`

其中包含了`restapi.py`和`example_request.py`两个python文件以及一个`Readme.md`说明文档。

```python
# restapi.py
# YOLOv5 🚀 by Ultralytics, AGPL-3.0 license
"""Run a Flask REST API exposing one or more YOLOv5s models."""

import argparse
import io

import torch
from flask import Flask, request
from PIL import Image

app = Flask(__name__)
models = {}

DETECTION_URL = "/v1/object-detection/<model>"
```

```
@app.route(DETECTION_URL, methods=["POST"])
def predict(model):
    if request.method != "POST":
        return

    if request.files.get("image"):
        # Method 1
        # with request.files["image"] as f:
        #     im = Image.open(io.BytesIO(f.read()))

        # Method 2
        im_file = request.files["image"]
        im_bytes = im_file.read()
        im = Image.open(io.BytesIO(im_bytes))

        if model in models:
            results = models[model](im, size=640)  # reduce
size=320 for faster inference
            return
results.pandas().xyxy[0].to_json(orient="records")


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Flask API
exposing YOLOv5 model")
    parser.add_argument("--port", default=5000, type=int,
help="port number")
    parser.add_argument("--model", nargs="+", default=["yolov5s"],
help="model(s) to run, i.e. --model yolov5n yolov5s")
    opt = parser.parse_args()

    for m in opt.model:
        models[m] = torch.hub.load("./", m, source='local')

    app.run(host="0.0.0.0", port=opt.port)  # debug=True causes
Restarting with stat
```

```
# example_request.py
# YOLOv5 🚀 by Ultralytics, AGPL-3.0 license
"""Perform test request."""
```

```
import pprint

import requests

DETECTION_URL = "http://localhost:5000/v1/object-detection/yolov5s"
IMAGE = "data/images/zidane.jpg"

# Read image
with open(IMAGE, "rb") as f:
    image_data = f.read()

response = requests.post(DETECTION_URL, files={"image":
image_data}).json()

pprint.pprint(response)
```

首先，需要更改restapi调用模型的位置，原始文件调用模型的代码为`models[m] = torch.hub.load("ultralytics/yolov5", m, *force_reload*=True, *skip_validation*=True)`，这里我们改成从本地获取模型文件（需要提前下载）：`models[m] = torch.hub.load("./", m, source='local')`

然后更改要测试的图片路径：`IMAGE = "data/images/zidane.jpg"`

测试时，首先运行restapi.py，启动相关服务，然后运行example_request.py，对图片进行检测，输出json返回值。

> *[{'class': 0,*
> *'confidence': 0.8811399341,*
> *'name': 'person',*
> *'xmax': 1141.8494873047,*
> *'xmin': 742.9986572266,*
> *'ymax': 720.0,*
> *'ymin': 48.3657836914},*
> *{'class': 27,*
> *'confidence': 0.6753361821,*
> *'name': 'tie',*
> *'xmax': 496.6465148926,*
> *'xmin': 442.0031433105,*
> *'ymax': 709.985534668,*
> *'ymin': 437.5282592773},*
> *{'class': 0,*

> *'confidence': 0.6648373604,*
> *'name': 'person',*
> *'xmax': 715.6708374023,*
> *'xmin': 123.0640869141,*
> *'ymax': 719.6807861328,*
> *'ymin': 193.3258972168},*
> *{'class': 27,*
> *'confidence': 0.2599711418,*
> *'ymin': 308.4199829102}]*

至此，我们已了解了flask以及restapi的基本原理，并结合具体实例初步了解restapi部署模型的相关流程。