Deepseek企业级Agent项目开发实战

Part 6. 使用 Python 的 FastAPI 构建 REST API

在《Part 2：Ollama REST API - api/generate 接口详解》和《Part 3：Ollama REST API - api/chat 接口详解》这两节内容中，我们介绍了如何使用`Ollama`的`REST API`接口，在`Python`中调用`Ollama`启动的模型服务，示例代码如下：

```
    # 设置 API 端点
    generate_url = "http://192.168.110.131:11434/api/generate"     # 这里需要根据实际情况进行修改

    # 示例数据
    generate_payload = {
        "model": "deepseek-r1:7b",    # 这里需要根据实际情况进行修改
        "prompt": "请生成一个关于人工智能的简短介绍。",  # 这里需要根据实际情况进行修改
        "stream": False,        # 默认使用的是True，如果设置为False，则返回的是一个完整的响应，而不是一个流式响应
    }

    # 调用生成接口
    response_generate = requests.post(generate_url, json=generate_payload)
    if response_generate.status_code == 200:
        generate_response = response_generate.json()
        print("生成响应:", json.dumps(generate_response, ensure_ascii=False, indent=2))
    else:
        print("生成请求失败:", response_generate.status_code, response_generate.text)
```

注意：`Ollama`提供的`OpenAPI`兼容接口，实质上就是`api/generate`和`api/chat`这两个接口的上层封装。

在应用的设计架构下，我们一直所说的前后端分离，其实就是指后端提供`REST API`接口，前端通过调用这些接口来获取数据并进行展示。 而后端服务，最简单的理解就是：通过类、函数、方法等，来实现业务逻辑的代码。比如我们对上述代码进一步封装成一个`Python`函数，则变成如下：

```
import requests  # type: ignore
import json

def generate_text(prompt: str, model: str = "deepseek-r1:32b", stream: bool = False) -> dict:
    """
    调用生成接口，返回生成的文本。

    :param prompt: 要生成文本的提示
    :param model: 使用的模型名称，默认为 "deepseek-r1:7b"
    :param stream: 是否使用流式响应，默认为 False
    :return: 生成的响应
    """
    # 设置 API 端点
```

```
    generate_url = "http://192.168.110.131:11434/api/generate"  # 这里需要根据实际情
况进行修改

    # 示例数据
    generate_payload = {
        "model": model,  # 使用传入的模型
        "prompt": prompt,  # 使用传入的提示
        "stream": False,
        'keep_alive': 0   # 调用完马上卸载模型
    }

    # 调用生成接口
    response_generate = requests.post(generate_url, json=generate_payload)
    if response_generate.status_code == 200:
        generate_response = response_generate.json()
        return generate_response
    else:
        raise Exception(f"生成请求失败: {response_generate.status_code},
{response_generate.text}")
```

进行调用:

```
prompt_text = "请生成一个关于人工智能的简短介绍。"  # 示例提示
response = generate_text(prompt_text)
print("生成响应:", json.dumps(response, ensure_ascii=False, indent=2))
```


```
生成响应: {
  "model": "deepseek-r1:32b",
  "created_at": "2025-02-21T04:33:38.534420627Z",
  "response": "<think>\n嗯，用户让我生成一个关于人工智能的简短介绍。首先，我得理解用户的需求
是什么。可能他需要这个介绍用于学习、演讲或者工作汇报之类的场合。\n\n接下来，我应该考虑内容结构。
通常，一个好的介绍应该涵盖定义、主要技术、应用领域以及一些伦理或社会影响的问题。这样可以让读者全
面了解AI的重要性和发展现状。\n\n然后，我需要用简单明了的语言来表达，避免太专业的术语，让更多人能
理解。比如，提到机器学习和深度学习时，可以稍微解释一下它们的基本概念。\n\n还要考虑用户可能没有说
出来的深层需求，比如他们是否需要最新的发展动态或者未来的趋势？不过这次是简短介绍，所以可能不需要
深入细节。\n\n最后，确保整个介绍逻辑清晰，层次分明，让读者能够快速抓住重点。这样写出来的内容既全
面又简洁，应该能满足用户的需求。\n</think>\n\n人工智能（Artificial Intelligence, AI）是指
通过计算机模拟人类智能的系统或技术。它涵盖机器学习、深度学习和自然语言处理等领域，使机器能够执行
复杂的任务，如图像识别、语音助手和自动驾驶等。AI正在改变我们的生活方式和工作方式，并在医疗、金
融、教育等多个领域展现出巨大潜力。然而，随着其快速发展，也引发了一些关于隐私、伦理和社会影响的讨
论。",
  "done": true,
  "done_reason": "stop",
  "context": [
    151644,
    14880,
    43959,
    46944,
    101888,
    104455,
    9370,
    98237,
    99534,
    100157,
    1773,
```

```
    151645,
    151648,
    198,
    106287,
    3837,
    20002,
    104029,
    43959,
    46944,
    101888,
    104455,
    9370,
    98237,
    99534,
    100157,
    1773,
    101140,
    3837,
    35946,
    49828,
    101128,
    20002,
    104378,
    102021,
    1773,
    87267,
    42411,
    85106,
    99487,
    100157,
    100751,
    100134,
    5373,
    105465,
    100631,
    99257,
    105032,
    106896,
    108430,
    1773,
    198,
    198,
    104326,
    3837,
    35946,
    99730,
    101118,
    43815,
    100166,
    1773,
    102119,
    3837,
    114128,
    100157,
    99730,
    102994,
    91282,
    5373,
```

```
    99558,
    99361,
    5373,
    99892,
    100650,
    101034,
    101883,
    112811,
    57191,
    99328,
    99564,
    103936,
    1773,
    99654,
    107366,
    104785,
    100011,
    99794,
    15469,
    101945,
    105178,
    99185,
    105044,
    1773,
    198,
    198,
    101889,
    3837,
    35946,
    118976,
    100405,
    30858,
    34187,
    109824,
    36407,
    102124,
    3837,
    101153,
    99222,
    104715,
    116925,
    3837,
    107863,
    17340,
    26232,
    101128,
    1773,
    101912,
    3837,
    104496,
    102182,
    100134,
    33108,
    102217,
    100134,
    13343,
    3837,
    73670,
```

```
    106683,
    104136,
    100158,
    104017,
    105166,
    101290,
    1773,
    198,
    198,
    104019,
    101118,
    20002,
    87267,
    80443,
    36587,
    104355,
    114246,
    100354,
    3837,
    101912,
    99650,
    64471,
    85106,
    108324,
    99185,
    104299,
    100631,
    105735,
    101226,
    11319,
    100632,
    103969,
    20412,
    98237,
    99534,
    100157,
    3837,
    99999,
    87267,
    104689,
    100403,
    104449,
    1773,
    198,
    198,
    100161,
    3837,
    103944,
    101908,
    100157,
    104913,
    104542,
    3837,
    100920,
    109556,
    3837,
    99258,
    104785,
```

```
    100006,
    101098,
    104609,
    99887,
    1773,
    99654,
    61443,
    104355,
    43815,
    99929,
    100011,
    99518,
    110485,
    3837,
    99730,
    112809,
    20002,
    104378,
    1773,
    198,
    151649,
    198,
    198,
    104455,
    9909,
    9286,
    16488,
    21392,
    11,
    15235,
    7552,
    104442,
    67338,
    104564,
    105717,
    103971,
    100168,
    9370,
    72448,
    57191,
    99361,
    1773,
    99652,
    102994,
    102182,
    100134,
    5373,
    102217,
    100134,
    33108,
    99795,
    102064,
    54542,
    106483,
    3837,
    32555,
    102182,
    100006,
```

```
    75117,
    106888,
    88802,
    3837,
    29524,
    107553,
    102450,
    5373,
    105761,
    110498,
    33108,
    109044,
    49567,
    1773,
    15469,
    96555,
    101933,
    103952,
    107142,
    33108,
    99257,
    75768,
    3837,
    104931,
    100182,
    5373,
    100015,
    5373,
    99460,
    108211,
    100650,
    112227,
    102334,
    102575,
    1773,
    103968,
    3837,
    101067,
    41146,
    106389,
    3837,
    74763,
    102361,
    105593,
    101888,
    107120,
    5373,
    112811,
    106640,
    99564,
    9370,
    104075,
    1773
  ],
  "total_duration": 80587533951,
  "load_duration": 67590369668,
  "prompt_eval_count": 13,
  "prompt_eval_duration": 3839000000,
```

```
  "eval_count": 274,
  "eval_duration": 9148000000
}
```

这其实就是一个非常典型的后端服务，即通过函数来实现业务逻辑。正如上述调用的示例，调用任意 `Python` 函数，其中只需要关注执行该函数应该传入的参数，以及函数返回的结果。

那么问题也随之而来，在本地代码环境下，我们通过函数来实现业务逻辑，这其实是非常容易的。但是，如果用户是在前端网页上点击一个按钮，我们希望它能够调用这个函数，并返回结果，这该如何实现呢？我们需要做的是，将本地函数封装成一个类似于 `Ollama` 的 `REST API` 接口一样，这样就可以通过某个 `URL` 地址来访问，并根据请求的参数即可执行对应的后端服务。而实现这个“链路”的关键，这就是本节内容要介绍的 `FastAPI`。

`FastAPI` 是一个开源项目，其作用是基于标准 `Python` 类型提示使用 `Python` 构建 `REST API`，使用 `ASGI` 的标准来构建 `Python Web` 框架和服务器。所有简单理解：`FastAPI` 是一个 `ASGI Web` 框架。官方Github：https://github.com/fastapi/fastapi

> 关于Python网络编程的扩展资料可以参考 `extend` 文件夹下的 `Python网络编程.pdf` 文件。

FastAPI framework, high performance, easy to learn, fast to code, ready for production

对 `FastAPI` 的两个概念理解：

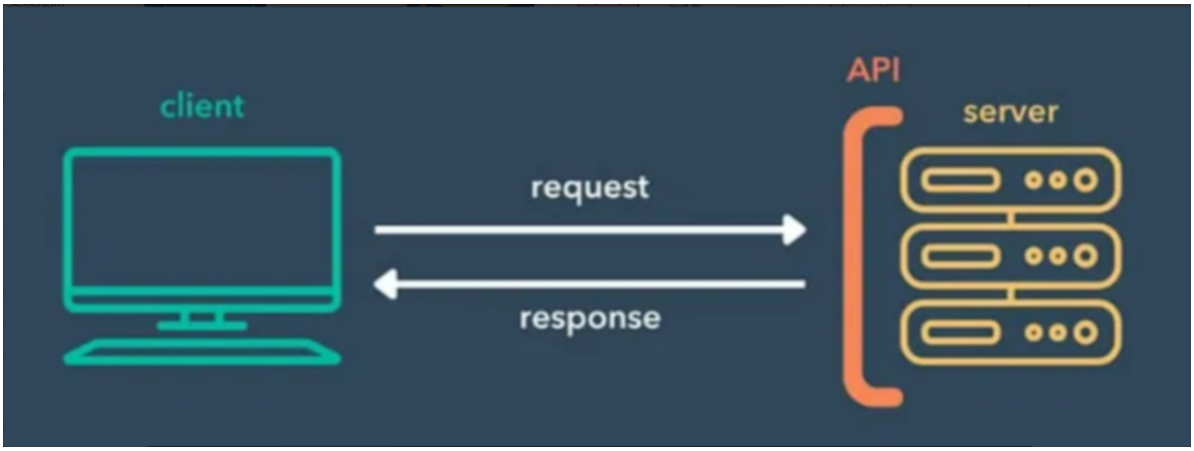
1. 它可以把一个 `Python` 函数，封装成一个 `REST API` 接口，并提供给前端调用。类似于 /api/generate 接口。
2. 它是一个 `ASGI Web` 框架，可以构建 `Python Web` 框架和服务器，解决前端和后端之间的通信问题。也就是可以让前端通过 `URL` 地址来访问后端服务，并根据请求的参数来执行对应的后端服务，并且前端还可以获取到后端服务的返回结果。

几个流行的基于ASGI的Web框架包括：

1. Starlette：一个轻量级的ASGI框架，它旨在成为构建高性能异步服务的基础框架。Starlette 可以独立使用，也可以作为其他框架的基础组件。
2. FastAPI：建立在Starlette之上的一个现代、快速（高性能）的Web框架，用于构建API。FastAPI 强调快速开发、类型安全和自动API文档生成。
3. Uvicorn：一个轻量级、超快的ASGI服务器，常用于运行ASGI应用。虽然Uvicorn自身不是一个Web框架，但它是运行基于ASGI的框架（如Starlette和FastAPI）的推荐服务器。

简单理解：`ASGI` 框架（`FastAPI` 和 `Starlette`）负责定义应用程序的逻辑和结构，服务器（`Uvicorn`）负责处理网络请求和响应。其中，`Uvicorn` 是 `FastAPI` 的默认服务器。

其数据流向如下图所示：



接下来，我们通过一个示例，来介绍如何使用`FastAPI`来构建一个`REST API`接口。

- 下载 fastapi 依赖包

使用 `pip install fastapi` 命令下载 `fastapi` 依赖包。

- 安装 uvicorn 依赖包

接下来，我们需要使用一个 `ASGI` 服务器来运行 `FastAPI` 应用程序。这里我们使用 `Uvicorn` 作为 `ASGI` 服务器。需要使用 `pip install uvicorn` 命令下载 `uvicorn` 依赖包。

- 创建一个 FastAPI 应用程序

创建一个 `FastAPI` 应用程序，并使用 `Uvicorn` 作为 `ASGI` 服务器来运行。这里需要说明的是，`jupyter notebook` 中是无法直接运行 `FastAPI` 应用程序的，需要使用 `Uvicorn` 作为 `ASGI` 服务器来运行。 因此，我们在`llm_backend/app/test/test_fastapi.py`文件中，来创建一个 `FastAPI` 应用程序，并使用 `Uvicorn` 作为 `ASGI` 服务器来运行。代码如下：

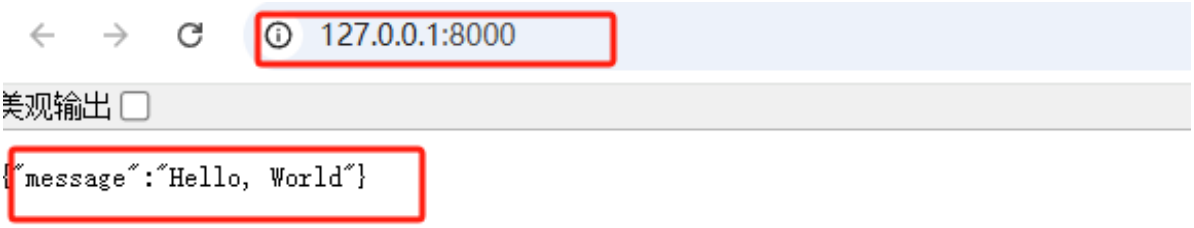
```
    from fastapi import FastAPI
    from pydantic import BaseModel

    app = FastAPI()  # 创建 FastAPI 应用实例

    class Item(BaseModel):
        name: str  # 项目的名称
        description: str = None  # 项目的描述，默认为 None

    @app.get("/")  # 定义 GET 请求的根路径
    async def read_root():
        """返回欢迎消息"""
        return {"message": "Hello, World"}

    @app.post("/items/")  # 定义 POST 请求的 /items 路径
    async def create_item(item: Item):
        """创建一个新项目并返回其数据"""
        return item  # 返回创建的项目数据
```

- 启动项目

在`llm_backend/app/test/test_fastapi.py`文件中，我们创建了一个 `FastAPI` 应用程序，并使用 `Uvicorn` 作为 `ASGI` 服务器来运行。接下来，我们启动项目，并访问 `http://127.0.0.1:8000/items/` 地址，来访问 `FastAPI` 应用程序。启动命令如下：

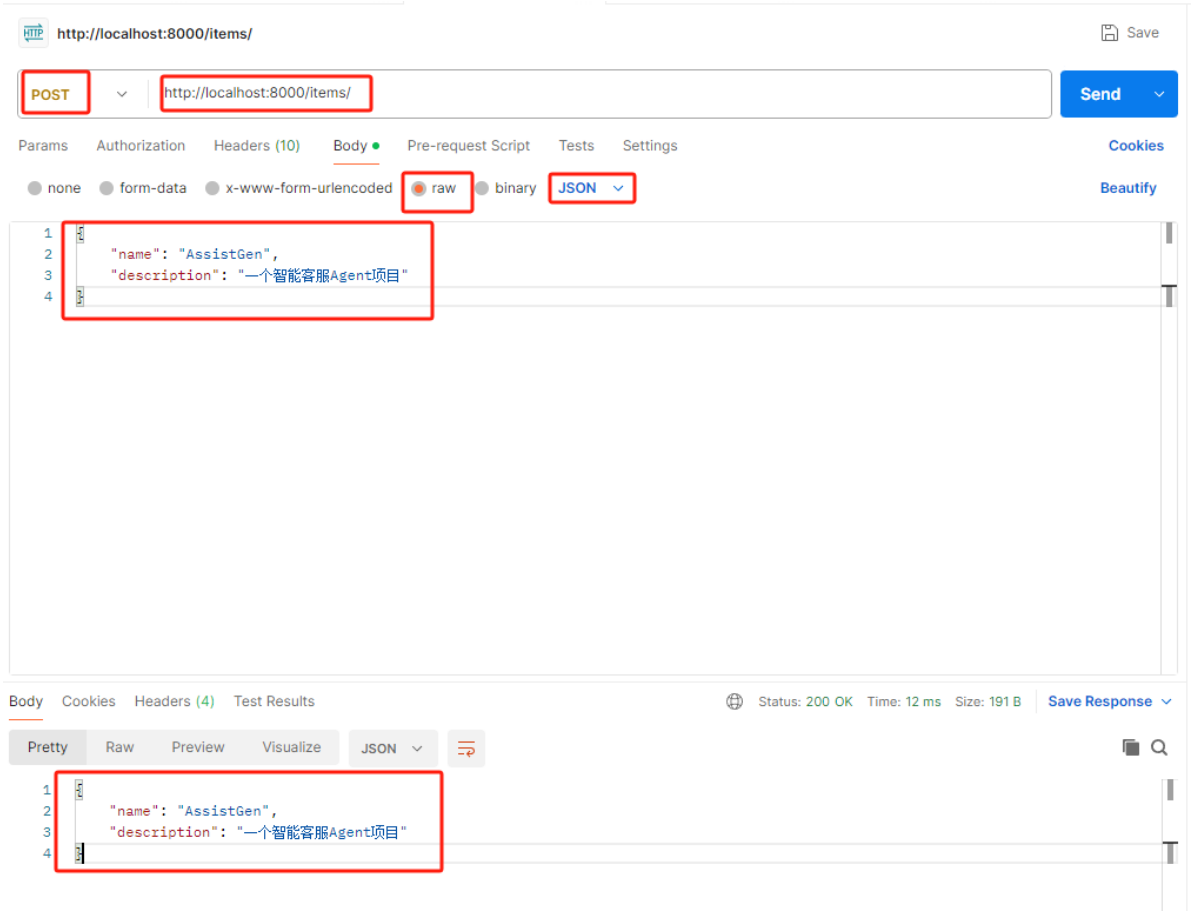
```
    uvicorn test_fastapi:app --reload
```

启动项目后，根据代码中定义的请求，当通过`localhost:8000/`地址访问时，会调用`read_root`函数，返回`Hello, World`消息。

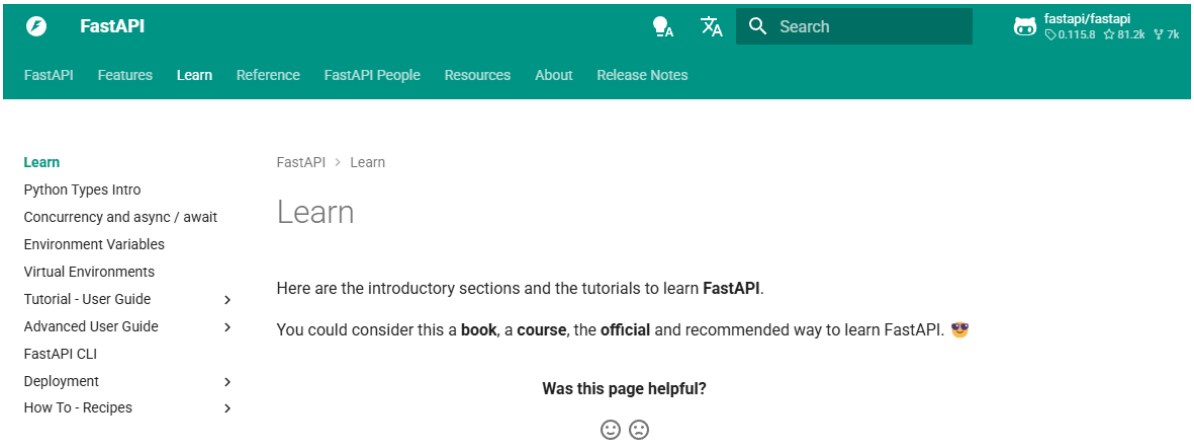


而当通过`localhost:8000/items/`地址访问时，会调用`create_item`函数，返回创建的项目数据。但是这里需要注意到是：`POST`请求是无法在浏览器的地址栏中直接访问的，因为`POST`请求需要发送请求体，而浏览器地址栏无法发送请求体。所有我们需要使用`curl`命令或者`Postman`工具来访问。

> PostMan 工具下载地址：https://www.postman.com/downloads/



整体实践也能够发现，`FastAPI`的基本使用并不是特别复杂，但是它能够帮助我们快速构建一个`REST API`接口，并提供给前端调用。但是其中也存在很多的细节，比如请求参数类型、返回结果类型、依赖注入、路由、安全、认证、日志等多方面的内容，这些内容我们会在接下来的实际案例中通过实际的代码来介绍。同时，大家也可以快速浏览`FastAPI`的官方文档，来了解更多关于`FastAPI`的详细内容，地址：https://fastapi.tiangolo.com/tutorial/



快速了解了`FastAPI`的基本使用后，我们就可以开始学习智能客服项目中的`FastAPI`接口代码了。入口文件是`llm_backend/main.py`文件。