

1.什么是Ollama

Ollama 是一个开源的大型语言模型（LLM）平台，Ollama 提供了简洁易用的命令行界面和服务端，使用户能够轻松下载、运行和管理各种开源 LLM，通过 Ollama，用户可以方便地加载和使用各种预训练的语言模型，支持文本生成、翻译、代码编写、问答等多种自然语言处理任务。

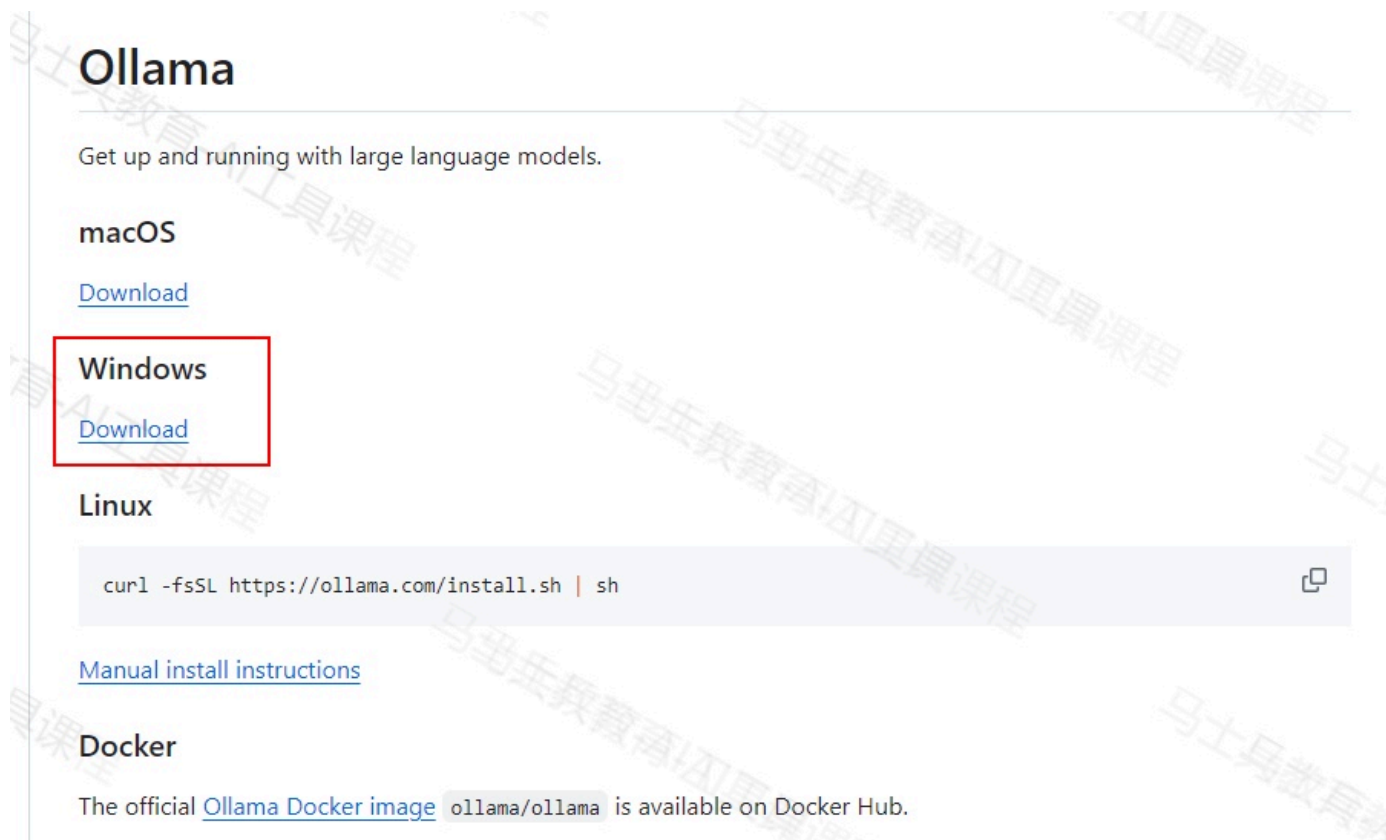
ollama官网：<https://ollama.com>

Ollama特点:

- 开源免费：Ollama 是一个开源的本地大型语言模型运行框架，用户可以自由使用、修改和分发，无需支付费用。
- 跨平台支持：Ollama 支持 macOS、Windows、Linux 以及 Docker多种操作系统环境下顺利部署和使用。
- 简单易用：通过命令行方式下载、运行、管理各种开源LLM，使非专业用户也能方便的管理运行各种复杂模型。
- 性能强大：充分利用本地资源，既可以使用GPU也可以使用CPU。
- 易于集成：Ollama 提供了命令行工具和 API，简化了与其他项目和服务的集成。

2.Ollama下载与安装

Ollama下载地址:<https://github.com/ollama/ollama>,这里以window中下载为例：



Ollama

Get up and running with large language models.

macOS

[Download](#)

Windows

[Download](#)

Linux

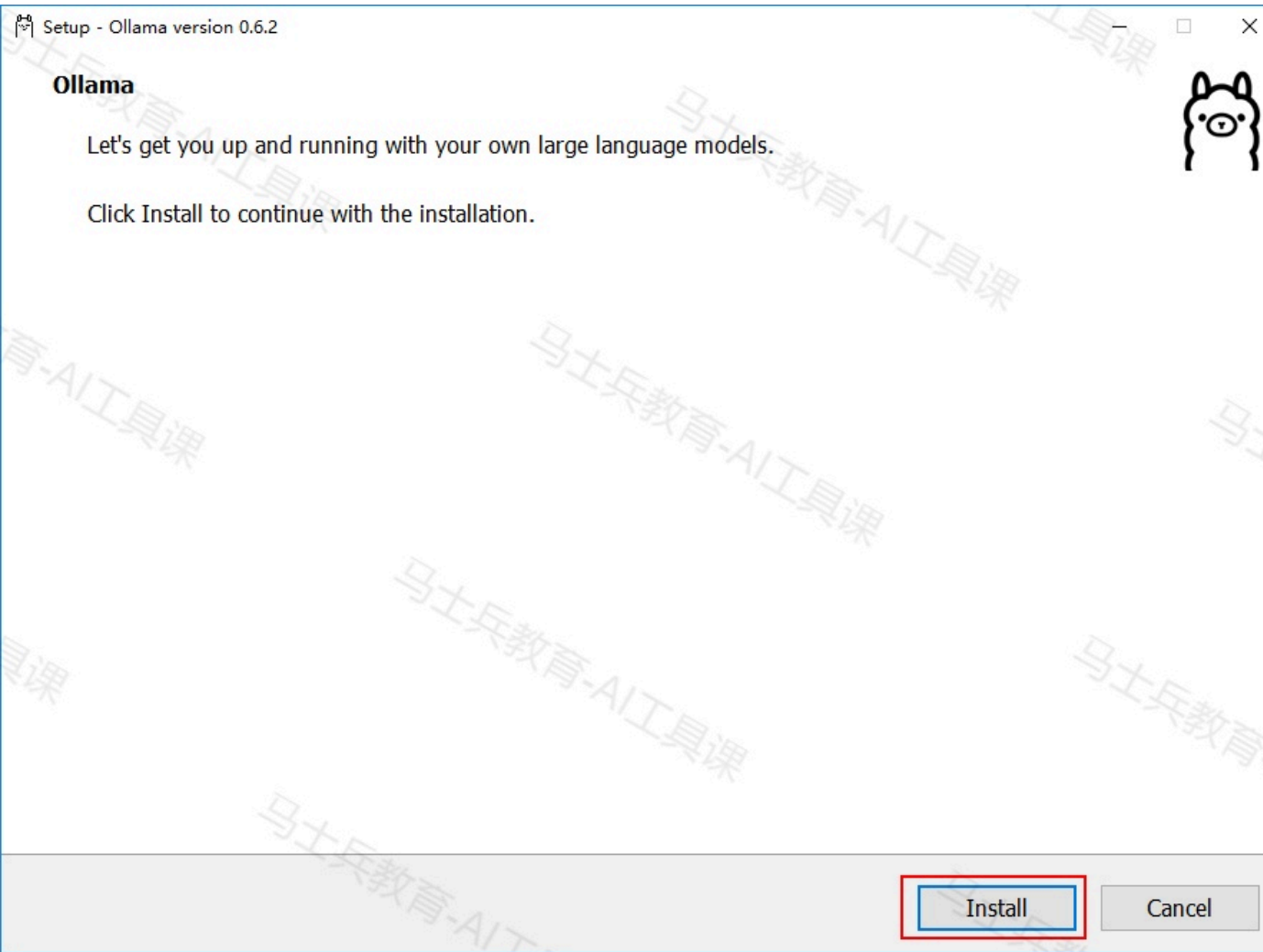
```
curl -fsSL https://ollama.com/install.sh | sh
```

[Manual install instructions](#)

Docker

The official [Ollama Docker image](#) `ollama/ollama` is available on Docker Hub.

下载完成后双击“OllamaSetup.exe” 进行安装，默认安装在“C:\users\{user}\AppData\Local\Programs” 目录下，建议C盘至少要有10G 剩余的磁盘空间，因为后续Ollama中还要下载其他模型到相应目录中。



安装完成后，电脑右下角自动有“Ollama” 图标并开机启动。

3.Ollama支持模型

Ollama支持很多模型，可以通过“<https://ollama.com/library>” 查看可用的模型列表，在本机Ollama运行7B模型至少需要8GB内存；运行13B模型至少需要16G内存；运行33B模型至少需要32G内存。

注意：“7B”、“13B” 和 “33B” 分别表示模型中参数的数量级，分别为 70 亿、130 亿和 330 亿个参数。模型的参数数量直接影响其对内存的需求，参数越多，模型越复杂，所需的内存也就越大，运行这些模型时，确保系统具有足够的内存。

如下是一些模型示例：

Model	Parameters	Size	Download

Gemma 3	1B	815MB	ollama run gemma3:1b
Gemma 3	4B	3.3GB	ollama run gemma3
Gemma 3	12B	8.1GB	ollama run gemma3:12b
Gemma 3	27B	17GB	ollama run gemma3:27b
QwQ	32B	20GB	ollama run qwq
DeepSeek-R1	7B	4.7GB	ollama run deepseek-r1
DeepSeek-R1	671B	404GB	ollama run deepseek-r1:671b
Llama 3.3	70B	43GB	ollama run llama3.3
Llama 3.2	1B	1.3GB	ollama run llama3.2:1b
Llama 3.2 Vision	11B	7.9GB	ollama run llama3.2-vision
Llama 3.2 Vision	90B	55GB	ollama run llama3.2-vision:90b
Llama 3.1	8B	4.7GB	ollama run llama3.1

推荐使用模型：大语言模型（Gemma、Deepseek、qwen2）、视觉模型（llava）。

备注：

- Gemma是由 Google DeepMind 团队开发的一系列先进、轻量级的开放模型，灵感源自 Google 的 Gemini 模型。Gemma 模型可用于自然语言处理、计算机视觉、跨模态任务等领域，适用于开发 AI 应用、研究和教育等多种场景。
- LLaVA(Large Language and Vision Assistant)，即大型语言和视觉助手，是一个端到端训练的大型多模态模型，将视觉编码器和大语言模型连接起来实现通用的视觉和语言理解。

4.Ollama 快速上手

可以通过“ollama run +模型”来运行模型，这里以运行“gemma”模型为例演示如何使用 Ollama。

1. 运行Gemma模型

打开cmd命令窗口，输入“ollama run gemma:2b”运行Gemma模型，如果Ollama没有该模型，会自动下载模型后再运行，后续运行不会重复下载模型。

```
ollama run gemma:2b
```

2. 使用Gemma模型

Ollama支持命令行方式使用模型，也支持API方式使用模型，这里先演示命令行方式使用模型，API方式使用模型参考后文。

```
>>> 你是谁？
我是一个 AI，人工智能的语言模型。我可以为你回答你的问题并进行对话。
>>> 你能做什么？
我能够提供各种语言处理服务，包括：

* 翻译文本
* 检索信息
* 生成创意文本
* 解答问题
* 对话

请问您还有其他问题吗？

>>> /clear
Cleared session context

>>> /bye
```

可以在与模型多次对话后输入“/clear”来清除上下文信息。最后使用ctrl+d 或者 输入 /bye 退出Ollama。

5.Ollama命令

可以在windows” 中打开cmd命令窗口，输入相应Ollama指令操作模型。

- **查看ollama已下载模型列表**

```
#命令 ollama list
ollama list
NAME ID SIZE MODIFIED
```

- **拉取模型**

```
#命令 ollama pull + 模型
ollama pull deepseek-r1:1.5b

#再次查看模型
ollama list
ollama list
NAME ID SIZE MODIFIED
deepseek-r1:1.5b a42b25d8c10a 1.1 GB 15 seconds ago
```

注意:ollama pull命令还可以用于更新本地模型，只会拉取diff不同的部分。

- **显示模型信息**

```
#命令 ollama show + 模型deepseek-r1:1.5b
```

```
ollama show deepseek-r1:1.5b
```

```
Model
```

```
architecture    qwen2
parameters      1.8B
context length   131072
embedding length 1536
quantization     Q4_K_M
```

```
Parameters
```

```
stop "< | begin_of_sentence | >"
stop "< | end_of_sentence | >"
stop "< | User | >"
stop "< | Assistant | >"
```

```
License
```

```
MIT License
Copyright (c) 2023 DeepSeek
```

- **运行模型进行会话**

```
#命令 ollama run +模型 deepseek-r1:1.5b
```

```
ollama run deepseek-r1:1.5b
```

```
>>> 你是谁？
```

```
<think>
```

```
</think>
```

```
您好！我是由中国的深度求索（DeepSeek）公司开发的智能助手DeepSeek-R1。如您有任何任何问题，我会尽我
```

```
>>>
```

使用run命令第一次运行模型，如果模型不存在会自动下载，然后进入模型交互窗口。后续使用会自动进入到交互窗口。

模型会话中支持多行输入，可以使用“"""”将文本括起来，进行对话。

```
>>> """
```

```
... 请给我讲一个故事
```

```
... 不超过100字
```

```
... """
```

```
<think>
```

```
</think>
```

```
当然可以，请听一下这个有趣的故事：
```

```
... ..
```

```
#退出模型
```

```
>>> /bye
```

- 将prompt作为参数传递给模型

在运行模型时，可以将prompt（提示词）作为参数传入给模型，无需进入与模型对话的交互式窗口即可获取模型返回内容。如下：

```
ollama run deepseek-r1:1.5b "你是谁"
<think>
</think>
您好！我是由中国的深度求索（DeepSeek）公司开发的智能助手DeepSeek-R1。如您有任何任何问题，我会尽我
```

- 查看模型执行效率

执行模型时，可以加入“--verbose”来查看每次对话后模型执行的效率细节。

```
ollama run deepseek-r1:1.5b --verbose
>>> 你是谁
<think>

</think>

您好！我是由中国的深度求索（DeepSeek）公司开发的智能助手DeepSeek-R1。如您有任何任何问题，我会尽我

total duration:    578.5214ms
load duration:     28.5914ms
prompt eval count: 5 token(s)
prompt eval duration: 37.5937ms
prompt eval rate:  133.00 tokens/s
eval count:        40 token(s)
eval duration:     511.8619ms
eval rate:         78.15 tokens/s
```

注意：

total duration：表示整个运行过程所花费的总时间。

load duration：表示加载模型所花费的时间，单位为毫秒。

prompt eval count：表示在处理提示（prompt）时评估的标记（token）数量。

prompt eval duration：表示评估提示所花费的时间，单位为毫秒。

prompt eval rate：表示评估提示时的速度，以每秒处理的标记数量表示。

eval count：表示在生成响应时评估的标记数量。

eval duration：表示生成响应所花费的时间，单位为毫秒。

eval rate：表示生成响应时的速度，以每秒处理的标记数量表示。

- **多模式模型**

不仅仅可以与模型进行对话，也可以让模型分析文本文件及图片内容。这里下载llava模型并进行图片内容分析。

如下：

```
#拉取模型
ollama pull llava:7b

#在C盘下准备pic.png图片，并通过该模型分析图片内容
C:\Users\wubai>ollama run llava:7b "请中文回复图片中是什么内容？ C:\pic1.png"
Added image 'C:\pic1.png'
这张照片显示了一名人，戴着头部上的葫芦叶服饰，手中也戴着一些装饰。他们身穿的衣服有多彩的元素，以及一些
```

- **删除模型**

```
#命令 ollama rm deepseek-r1:1.5b
```

6.自定义模型

Ollama中允许用户基于现有模型进行定制化设置，如：调整模型推理参数、设置提示模版等，自定义模型需要创建一个Modelfile文件，在文件中设置参数和提示模版相关内容，如下：

```
FROM llama3.2

# set the temperature to 1 [higher is more creative, lower is more coherent]，范围0-1
PARAMETER temperature 1

# set the system message
SYSTEM ""
You are Mario from Super Mario Bros. Answer as Mario, the assistant, only.
""
```

更多Modelfile设置的参数及内容参考：<https://github.com/ollama/ollama/blob/main/docs/modelfile.md>

下面基于deepseek-r1:1.5b模型来创建一个模型，设置推理参数及提示模版，步骤如下：

1) 创建Modelfile文件

在C盘中创建Modelfile，并写入如下内容：

```
FROM deepseek-r1:1.5b

# 设置 temperature 为 1（值越高，创造性越强；值越低，连贯性越强）
PARAMETER temperature 0.9
```



```
# 设置自定义系统消息，以指定聊天助手的行为
SYSTEM """
你是一位名为“小智”的虚拟助手，专长于文学和历史领域，喜欢以生动有趣的方式与用户交流，提供深入且易于理解的信息。
"""
```

2) 指定Modelfile创建并运行模型

```
C:\Users\wubai>ollama create my_deepseek -f C:\Modelfile
```

查看ollama中的模型：

```
C:\Users\wubai>ollama list
NAME                ID                SIZE    MODIFIED
my_deepseek:latest  4c0be70b4516     1.1 GB  8 seconds ago
llava:7b            8dd30f6b0cb1     4.7 GB  1 hours ago
gemma:2b            b50d6c999e59     1.7 GB  4 hours ago
deepseek-r1:1.5b    a42b25d8c10a     1.1 GB  4 hours ago
```

3) 使用模型

```
C:\Users\wubai>ollama run my_deepseek
>>> 你是谁？
... ..
>>> 给我介绍下清朝最勤劳的皇帝
... ..
```

7.Ollama可视化WebUI

下面介绍两款Ollama可视化WebUI工具：Open WebUI和Msty，方便用户使用与管理模型。

1. Open WebUI

Open WebUI 是一个功能丰富、用户友好的自托管 AI 平台，旨在完全离线运行,它支持多种大型语言模型（LLM）运行器，如 Ollama，并兼容 OpenAI API，内置的 RAG 推理引擎使其成为强大的 AI 部署解决方案,提供离线访问和流畅的用户界面。

安装Open WebUI需要python环境，使用“pip install open-webui”安装即可，python版本要求3.11版本。下面演示在Window中基于Anconda(默认已安装，不再讲解)安装python 3.11环境并在该环境中安装Open WebUI。

1) 在Anconda 中安装python3.11环境

```
#打开cmd,创建一个名为ollama_python311的环境，指定Python版本是3.11(此时默认安装python3.11的最新版)
conda create --name ollama_python311 python=3.11.11

#安装好后，使用activate激活某个环境
```



```
activate ollama_python311 # for Windows
source activate ollama_python311 # for Linux & Mac

#此时，查看python版本
python --version

#如果想返回默认的python 环境，运行
deactivate ollama_python311 # for Windows
source deactivate ollama_python311 # for Linux & Mac

#删除一个已有的环境
conda remove --name ollama_python311 --all
```

2) 安装open webui

```
#切换ollama_python311环境，安装open webui
C:\Users\wubai>activate ollama_python311
#默认安装open-webui的最新版，此时为0.5.20版本
(ollama_python311) C:\Users\wubai>pip install open-webui
```

3) 运行并访问open webui

安装open webui后可以通过 “open-webui serve” 来启动Open WebUI，通过 “http://localhost:8080” 来访问Open WebUI。

```
#启动Open WebUI
(ollama_python311) C:\Users\wubai>open-webui serve

#访问Open WebUI
http://localhost:8080/
```

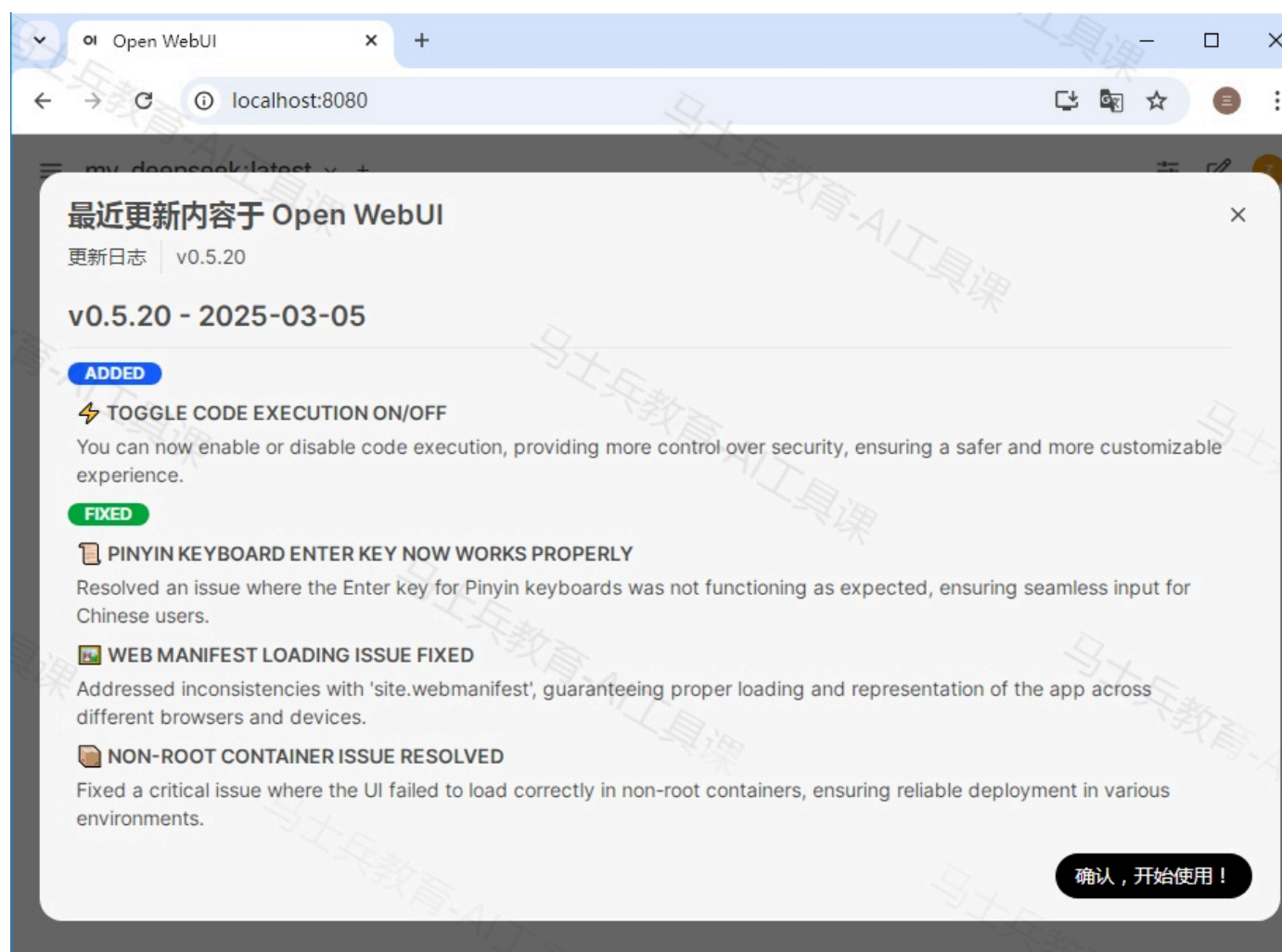
浏览器访问Open WebUI后，按照如下步骤来设置：



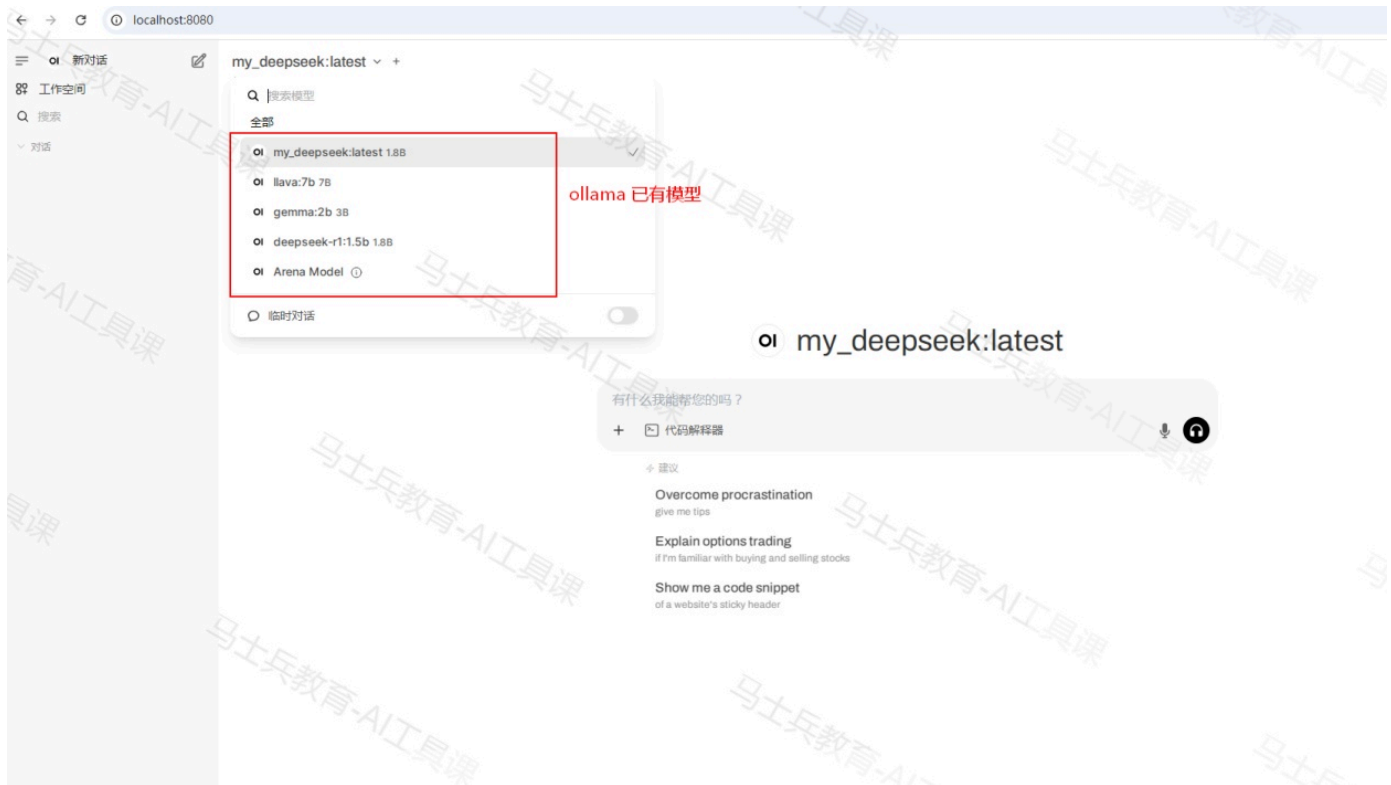
首次使用需要创建用户并设置用户、邮箱、密码(设置后记住电子邮箱和密码)：



创建完成后会连接 “api.openai.com” （连接不上也没关系），等待一会自动进入到WebUI界面：



以上Open WebUI安装完成后，可以自动识别Ollama（Ollama安装并已经运行），可以通过Open WebUI来与Ollama管理的模型进行对话、管理Ollama实例等操作。



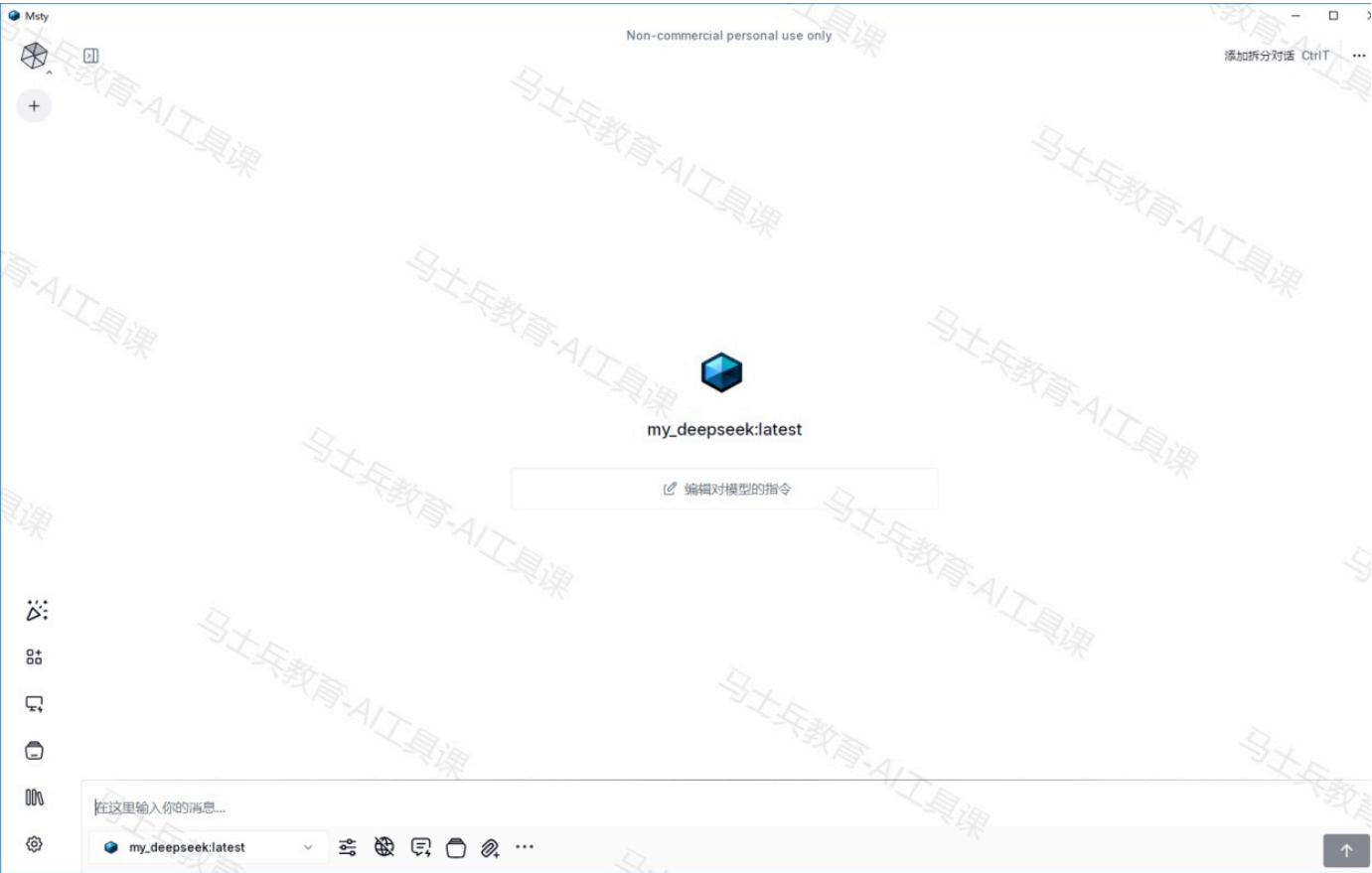
选择已有模型并对话：

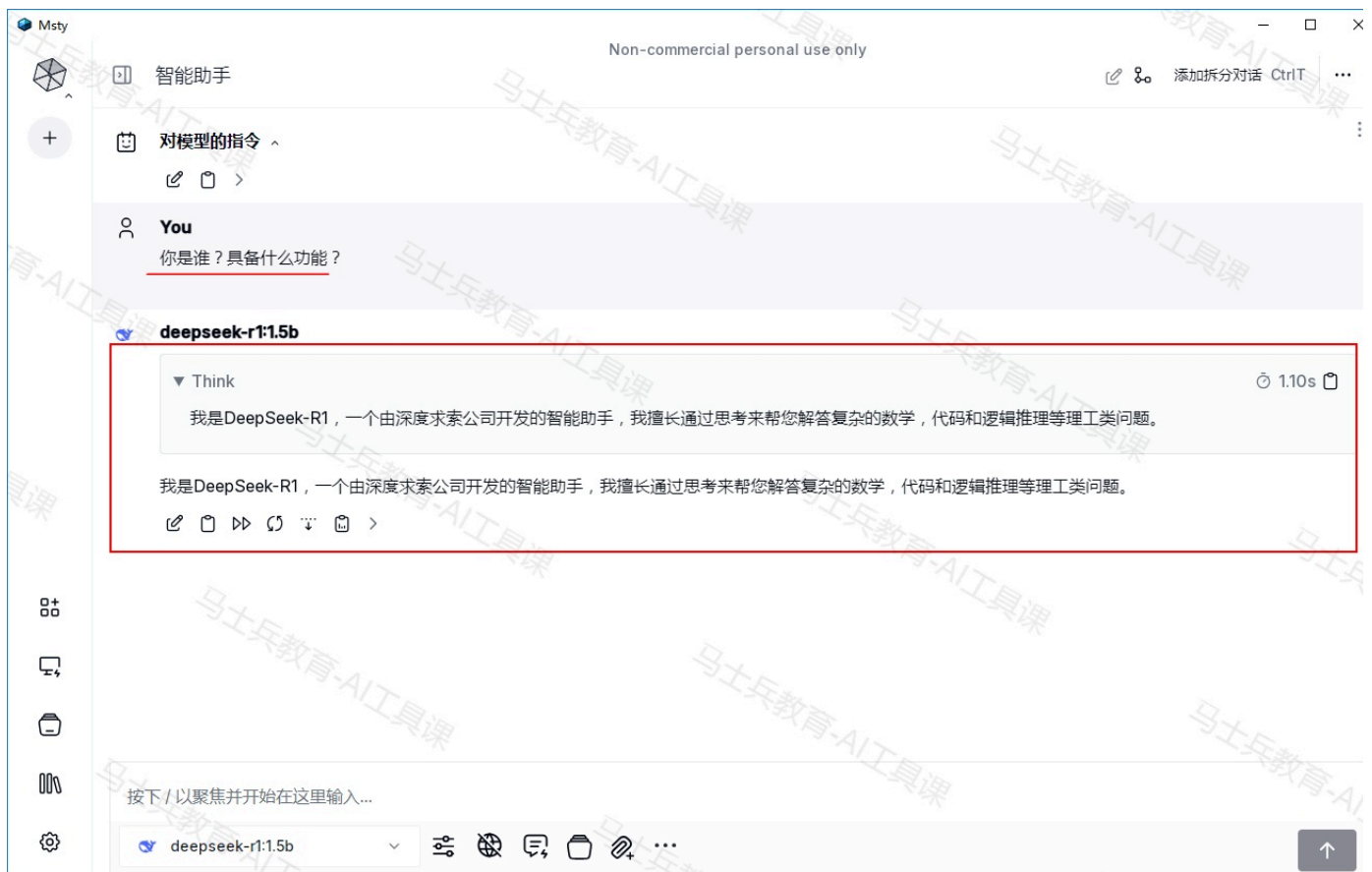
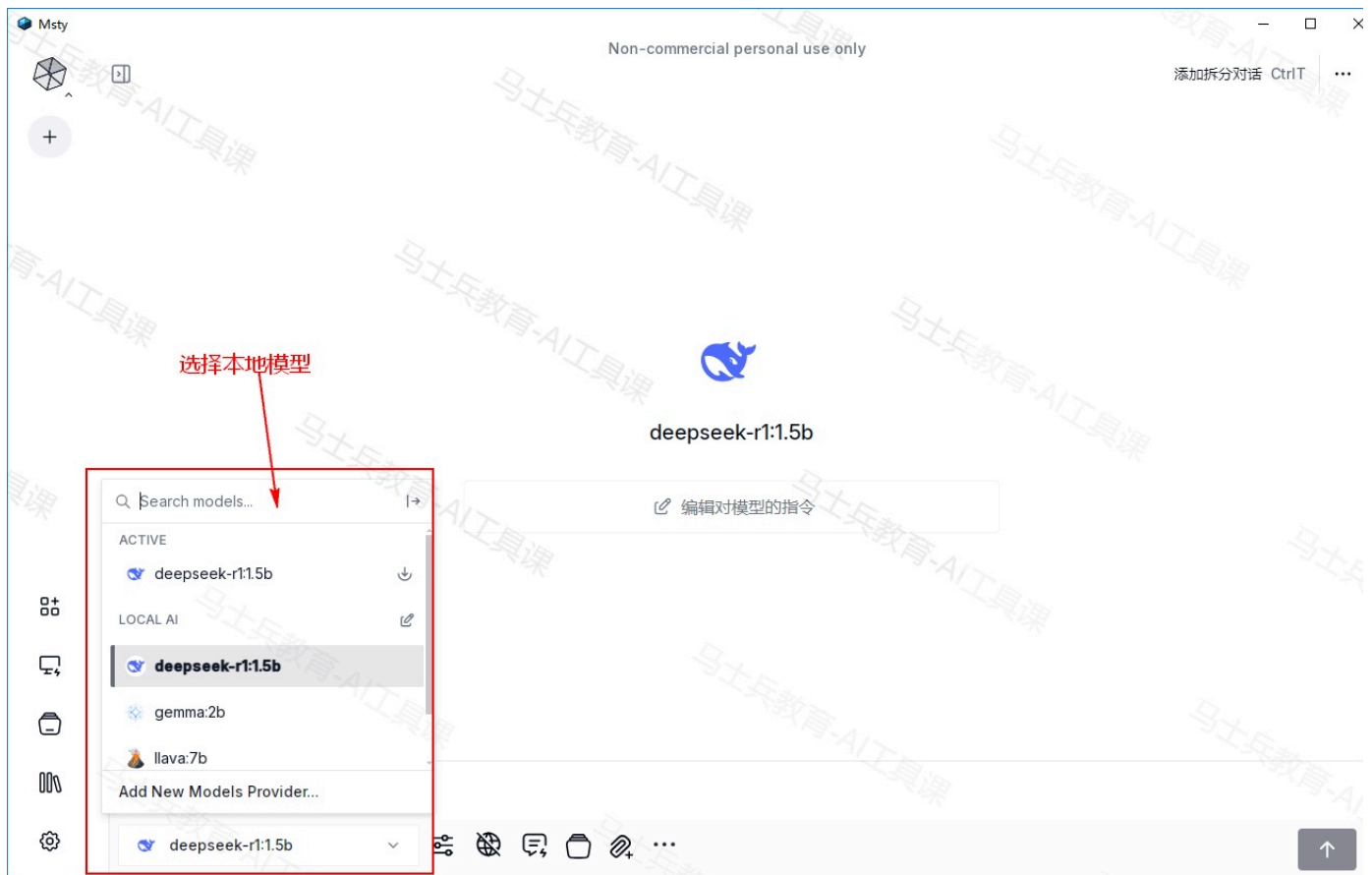


2. Msty

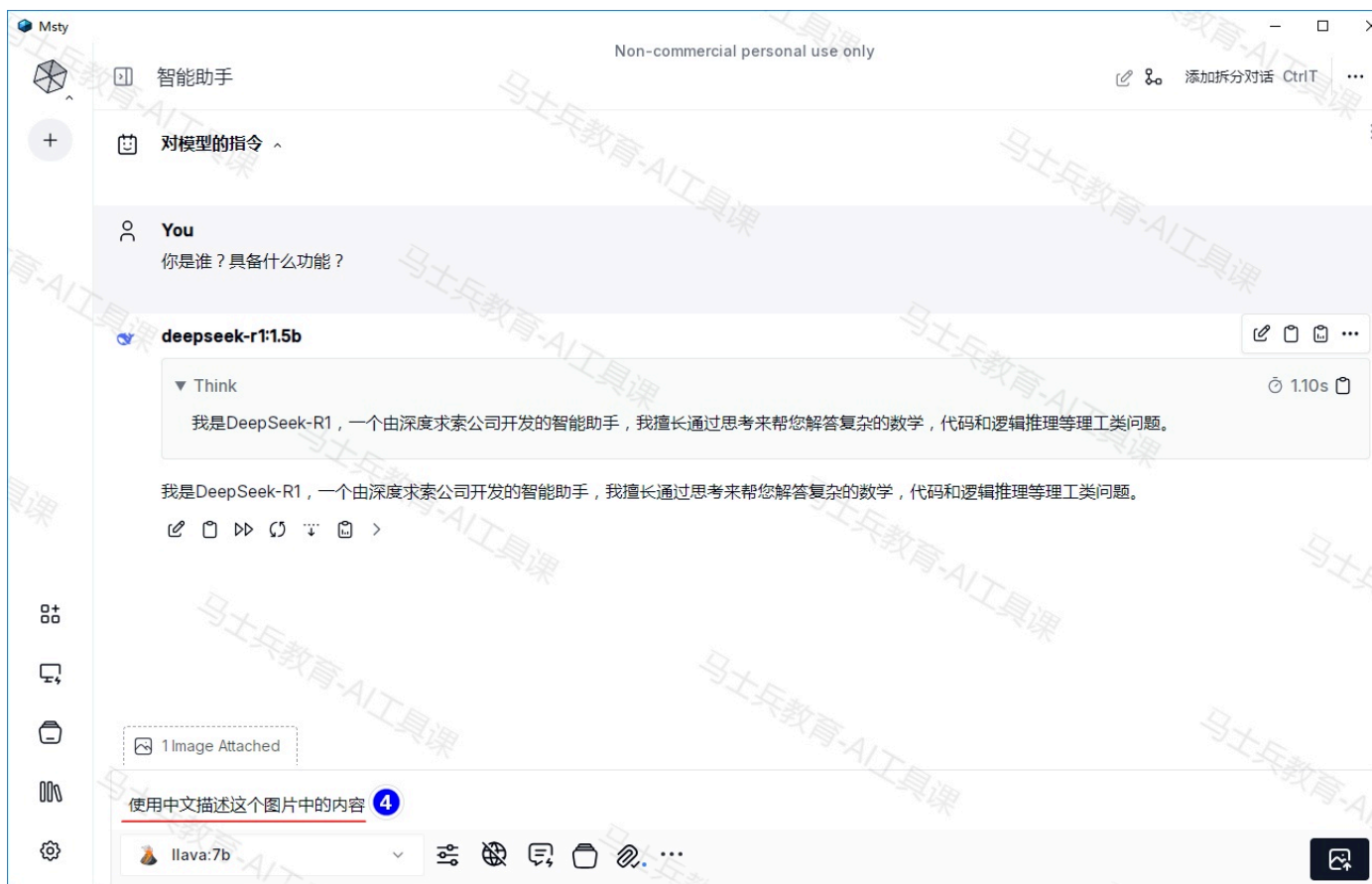
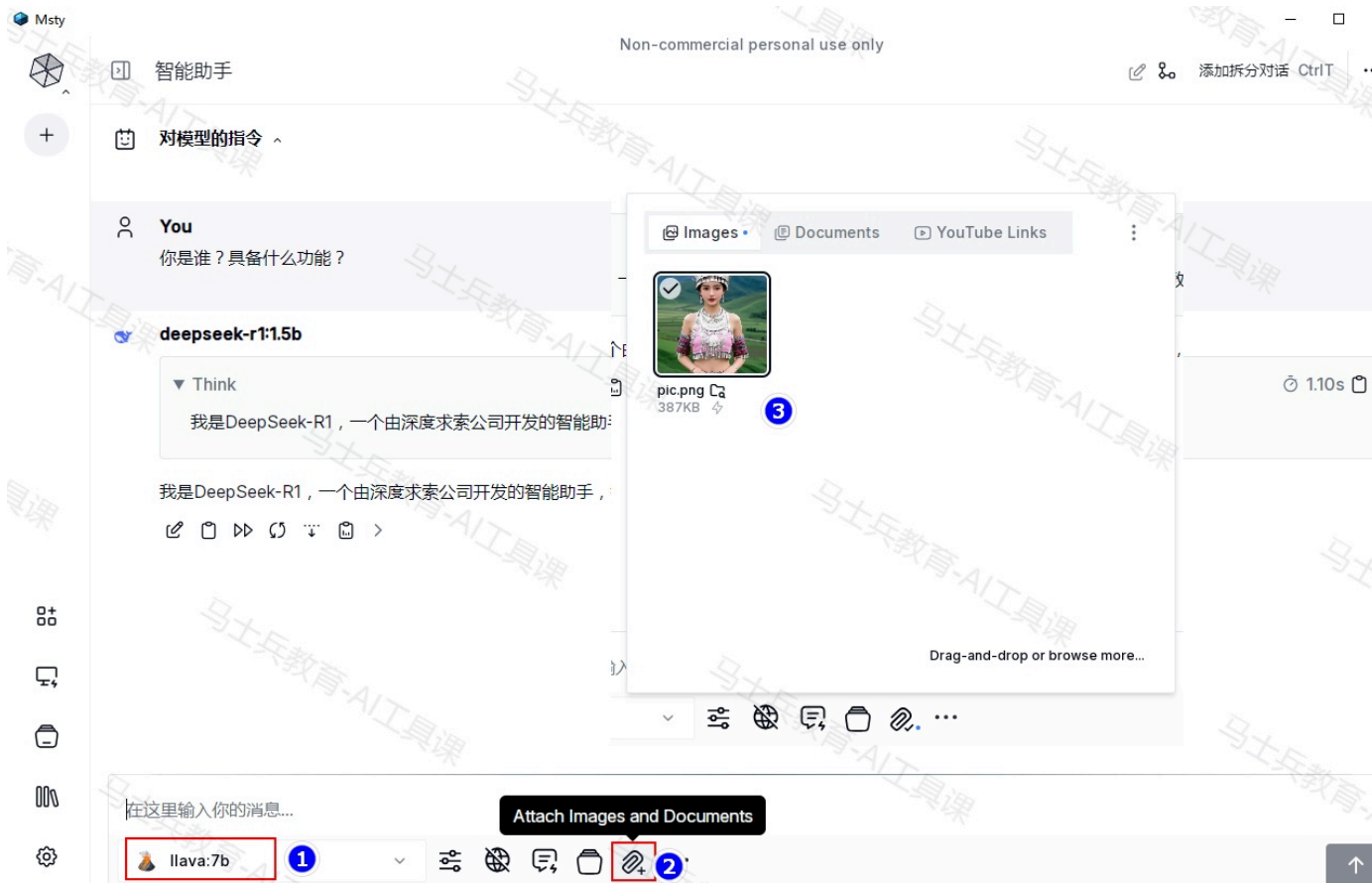
Msty官网：<https://msty.app/>，这里下载Window x64 GPU版本并安装，介绍Msty的使用。

下载Msty并双击“Msty_x64.exe”安装，安装完成后双击“Msty”图标进入到Msty，Msty会自动识别本机是否安装Ollama并导入Ollama相关模型。



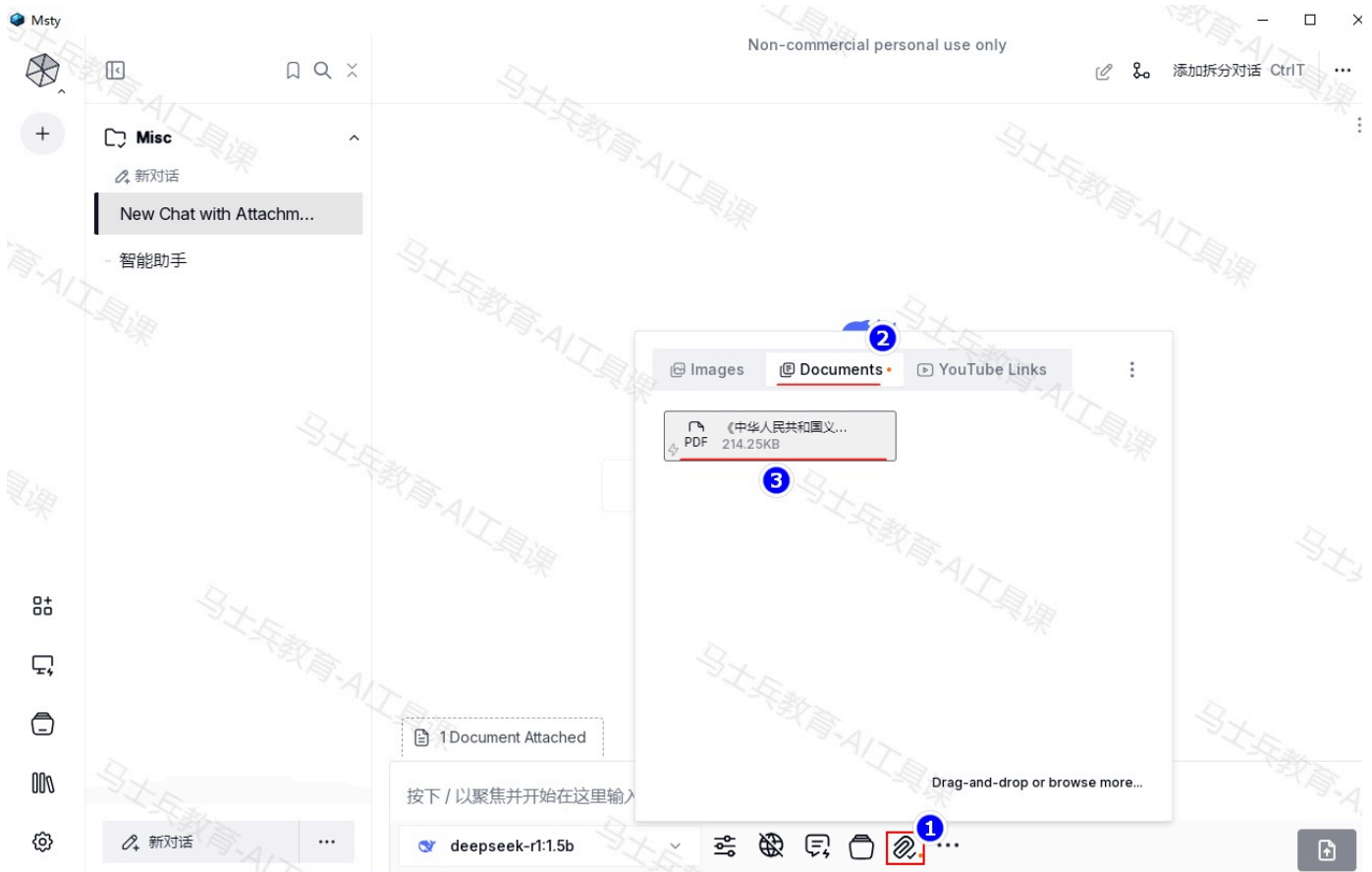


- 选择 “llava:7b” 模型，让模型识别图片，将 “pic.png” 图片上传





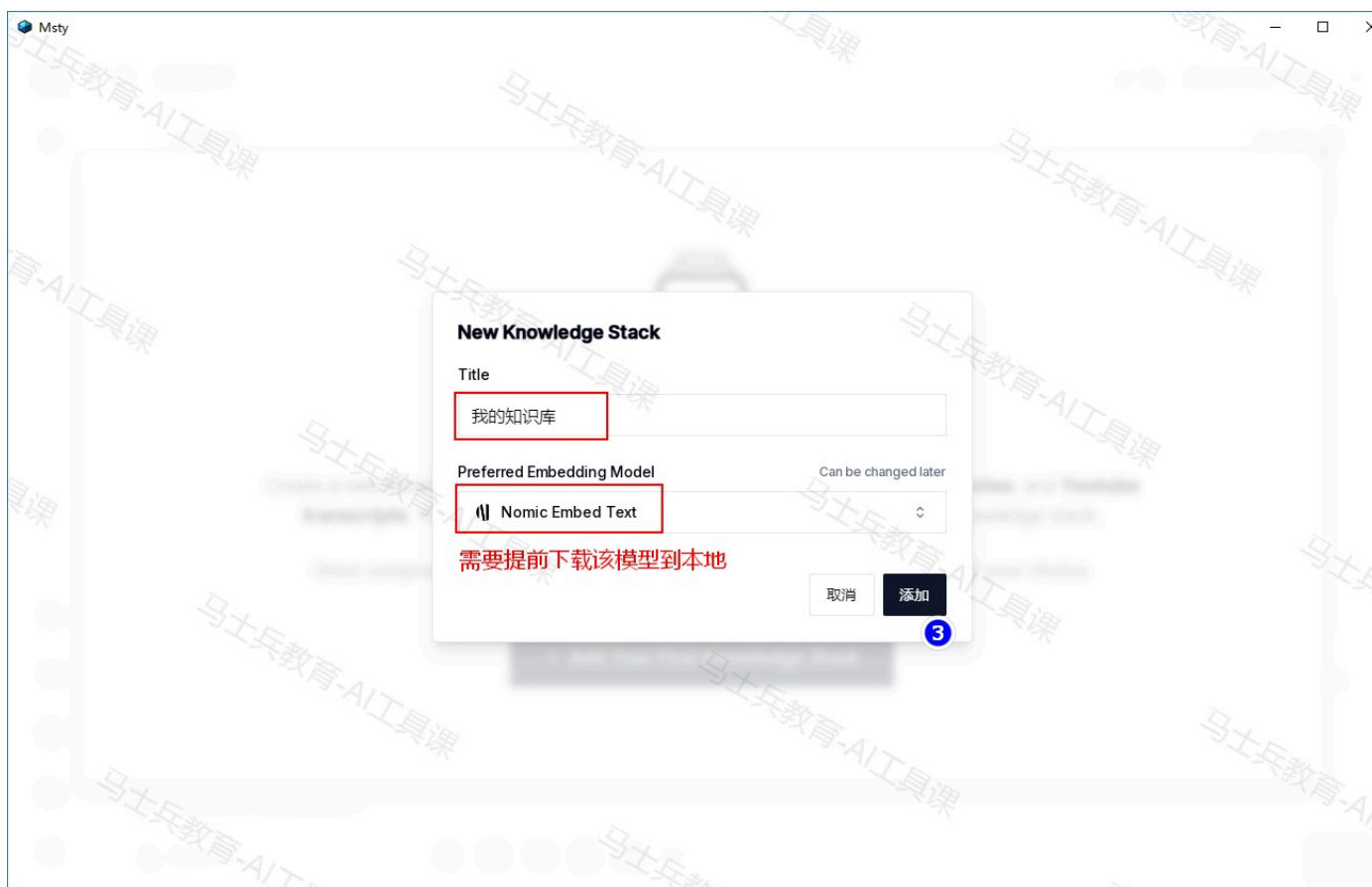
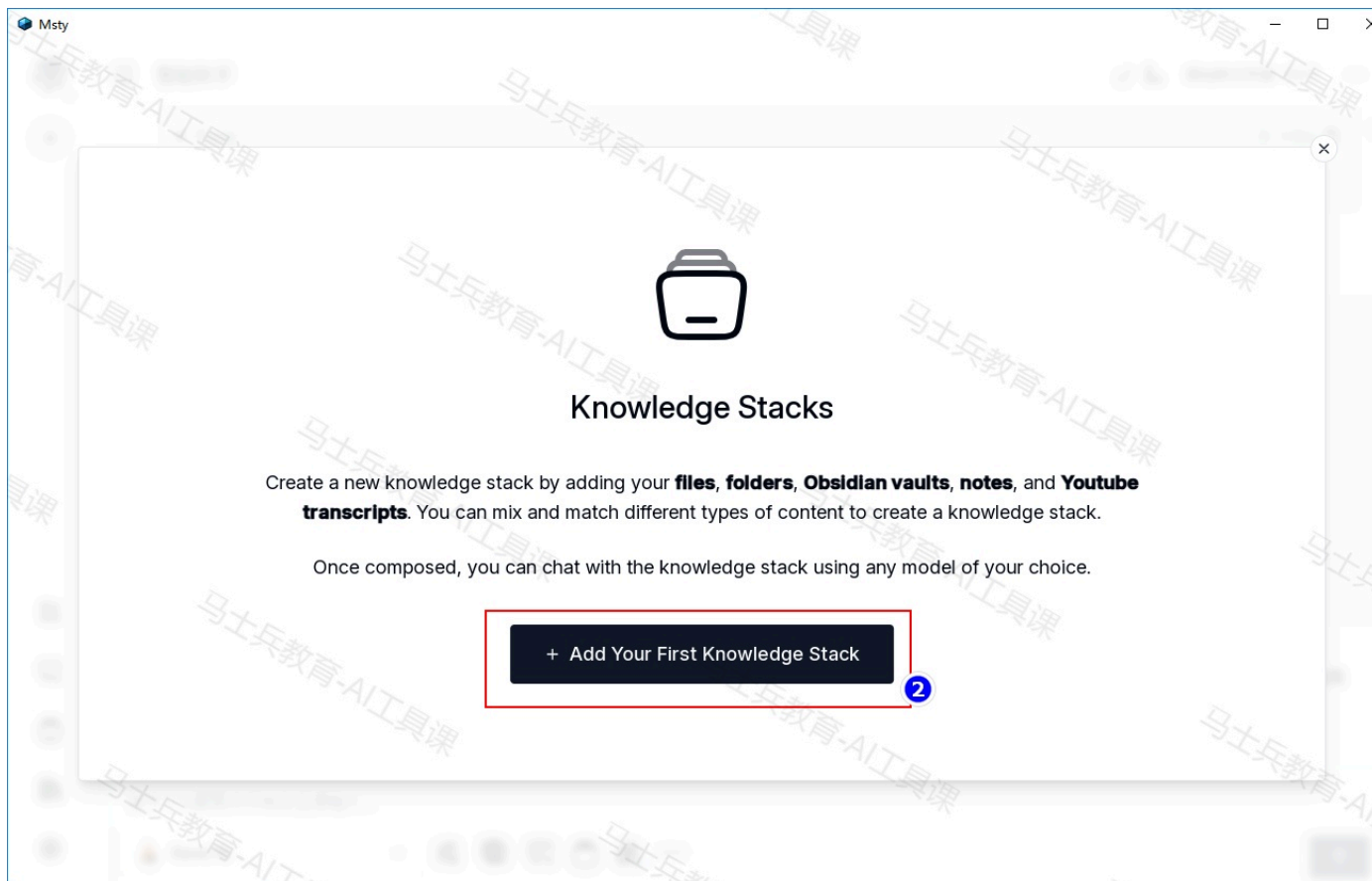
- 选择“deepseek-r1:1.5”模型，让模型识别上传文件



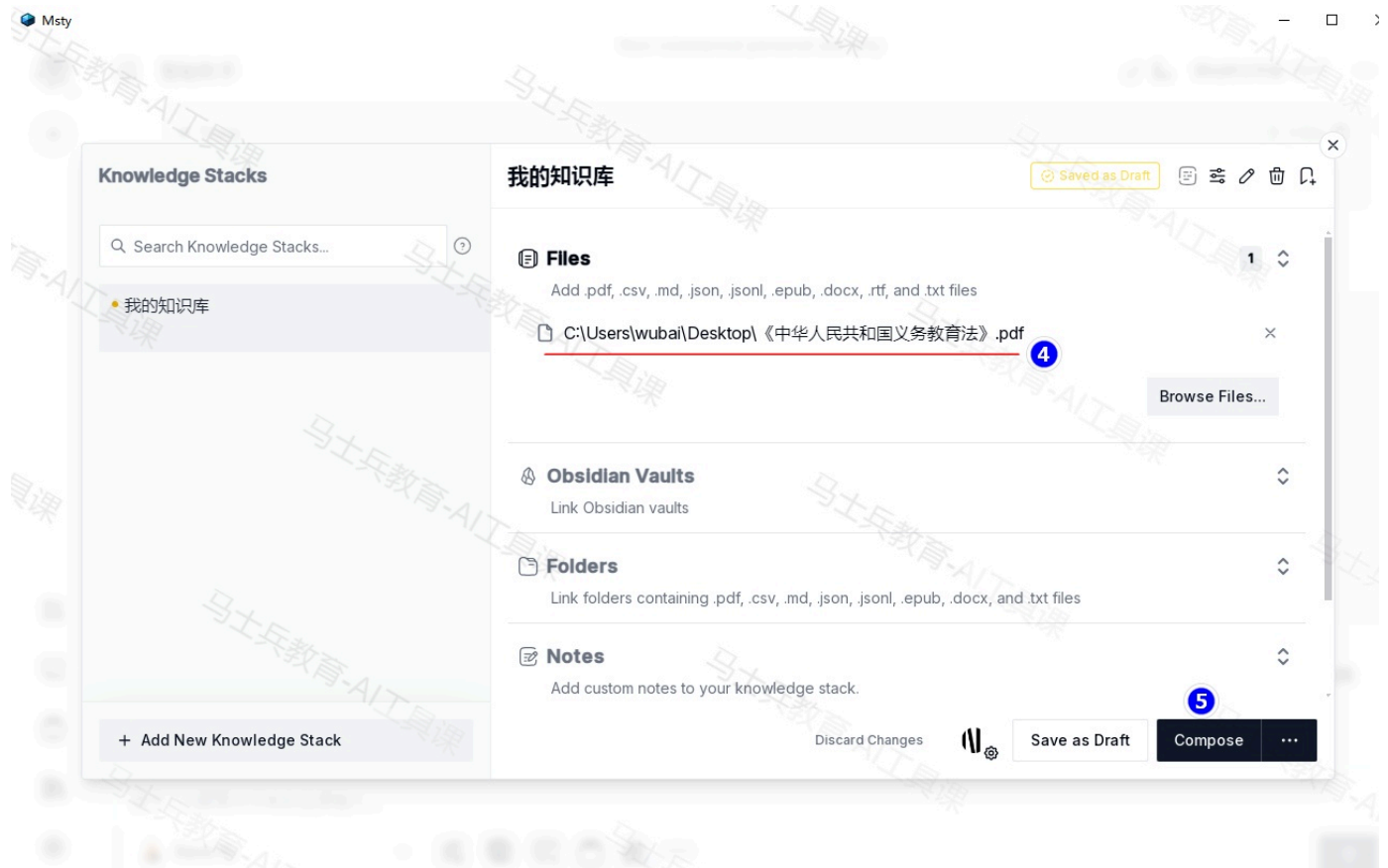


• 选择本地文件作为本地知识库使用





注意：Nomic Embed Text 模型是一个完全开源的英文文本嵌入模型，能够将文本转换为固定长度的向量表示，用于语义搜索、文本分类、聚类等自然语言处理任务。在ollama中下载该模型命令为 `"ollama pull nomic-embed-text"` ,大概274M。





以上对于文本分析需要选择支持的文本类模型，图像分析需要选择支持图像的模型。

8.Ollama Java API方式使用模型

如下是通过Rest API方式与模型进行对话示例，参考链接：<https://github.com/ollama/ollama/blob/main/docs/api.md>：

1) 与模型一次对话

```
#请求
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "why is the sky blue?"
    }
  ],
  "stream": false
}'

#响应
{
  "model": "llama3.2",
  "created_at": "2023-12-12T14:13:43.416799Z",
  "message": {
    "role": "assistant",
    "content": "Hello! How are you today?"
  },
  "done": true,
  "total_duration": 5191566416,
  "load_duration": 2154458,
  "prompt_eval_count": 26,
  "prompt_eval_duration": 383809000,
  "eval_count": 298,
  "eval_duration": 4799921000
}
```

2) 与模型多次对话（携带历史消息）

```
#请求
curl http://localhost:11434/api/chat -d '{
  "model": "llama3.2",
  "messages": [
    {
      "role": "user",
      "content": "why is the sky blue?"
    },
    {
      "role": "assistant",
      "content": "due to rayleigh scattering."
    }
  ]
}'
```

```

    },
    {
      "role": "user",
      "content": "how is that different than mie scattering?"
    }
  ]
}'

```

#响应

```

{
  "model": "llama3.2",
  "created_at": "2023-08-04T08:52:19.385406455-07:00",
  "message": {
    "role": "assistant",
    "content": "The"
  },
  "done": false
}

```

下面通过Java API方式演示使用方式实现 Ollama API方式与本地模型进行一次对话或者多次对话。

- 与模型一次对话

```

try {
  // 设置请求的URL
  URL url = new URL("http://localhost:11434/api/chat");
  HttpURLConnection connection = (HttpURLConnection) url.openConnection();

  // 设置请求方法为POST
  connection.setRequestMethod("POST");
  connection.setDoOutput(true);
  connection.setRequestProperty("Content-Type", "application/json; utf-8");

  // 创建JSON格式的请求体
  String jsonInputString = "{" +
    "\"model\": \"deepseek-r1:1.5b\", " +
    "\"messages\": [\n" +
    "  { \"role\": \"user\", \"content\": \"天空为什么是蓝色的?\" }\n" +
    " ], " +
    "\"stream\": false " + //是否流式返回结果，默认为true,流式返回结果
    "}";

  // 将请求体写入输出流
  try (OutputStream os = connection.getOutputStream()) {
    byte[] input = jsonInputString.getBytes(StandardCharsets.UTF_8);
    os.write(input, 0, input.length);
  }
}

```



```

// 读取响应
int responseCode = connection.getResponseCode();
System.out.println("Response Code: " + responseCode);

// 创建一个ObjectMapper实例
ObjectMapper objectMapper = new ObjectMapper();
StringBuilder fullResponse = new StringBuilder();

// 处理响应
try (BufferedReader br = new BufferedReader(new InputStreamReader(connection.getInputStream()), St
    String responseLine;
    while ((responseLine = br.readLine()) != null) {
        System.out.println("responseLine: " + responseLine);

        // 解析每一行JSON响应
        JsonNode jsonNode = objectMapper.readTree(responseLine);
        // 获取"message"中的"content"字段的值并拼接
        if (jsonNode.has("message") && jsonNode.get("message").has("content")) {
            fullResponse.append(jsonNode.get("message").get("content").asText());
        }
        // 检查"done"字段的值
        if (jsonNode.has("done") && jsonNode.get("done").asBoolean()) {
            break; // 如果"done"为true，结束循环
        }
    }
}

// 输出拼接后的完整响应
System.out.println("Full Response: " + fullResponse.toString());

} catch (Exception e) {
    e.printStackTrace();
}

```

• 与模型多次对话

与模型多次对话，这里获取用户在控制台输入内容作为每次与模型对话内容，需要携带历史消息，代码如下：

```

// 创建一个ObjectMapper实例
ObjectMapper objectMapper = new ObjectMapper();
// 用于存储对话历史记录
List<JsonNode> messages = new ArrayList<>();
// 创建一个Scanner对象以读取用户输入
Scanner scanner = new Scanner(System.in);

while (true) {
    try {
        // 提示用户输入

```

```
System.out.print("你: ");
String userInput = scanner.nextLine();

// 将用户输入添加到对话历史记录中
JsonNode userMessage = objectMapper.createObjectNode()
    .put("role", "user")
    .put("content", userInput);
messages.add(userMessage);

// 设置请求的URL
URL url = new URL("http://localhost:11434/api/chat");
URLConnection connection = (URLConnection) url.openConnection();

// 设置请求方法为POST
connection.setRequestMethod("POST");
connection.setDoOutput(true);
connection.setRequestProperty("Content-Type", "application/json; utf-8");

// 创建JSON格式的请求体
JsonNode requestBody = objectMapper.createObjectNode()
    .put("model", "deepseek-r1:1.5b")
    .putPOJO("messages", messages)
    .put("stream", false); // 设置为非流式返回结果，默认为true

// 将请求体对象转换为JSON字符串
String jsonString = objectMapper.writeValueAsString(requestBody);

// 输出请求体
System.out.println("Request Body: " + jsonString);

// 将请求体写入输出流
try (OutputStream os = connection.getOutputStream()) {
    byte[] input = objectMapper.writeValueAsBytes(requestBody);
    os.write(input, 0, input.length);
}

// 读取响应
int responseCode = connection.getResponseCode();
if (responseCode == HttpURLConnection.HTTP_OK) {
    StringBuilder fullResponse = new StringBuilder();

    // 处理响应
    try (BufferedReader br = new BufferedReader(new InputStreamReader(connection.getInputStream))) {
        String responseLine;
        while ((responseLine = br.readLine()) != null) {
            // 解析每一行JSON响应
            JsonNode jsonNode = objectMapper.readTree(responseLine);
            // 获取"message"中的"content"字段的值并拼接
            if (jsonNode.has("message") && jsonNode.get("message").has("content")) {
                String assistantResponse = jsonNode.get("message").get("content").asText();
            }
        }
    }
}
```

```

        fullResponse.append(assistantResponse);
        // 将助手的回复添加到对话历史记录中
        JsonNode assistantMessage = objectMapper.createObjectNode()
            .put("role", "assistant")
            .put("content", assistantResponse);
        messages.add(assistantMessage);
    }
    // 检查"done"字段的值
    if (jsonNode.has("done") && jsonNode.get("done").asBoolean()) {
        break; // 如果"done"为true，结束循环
    }
}

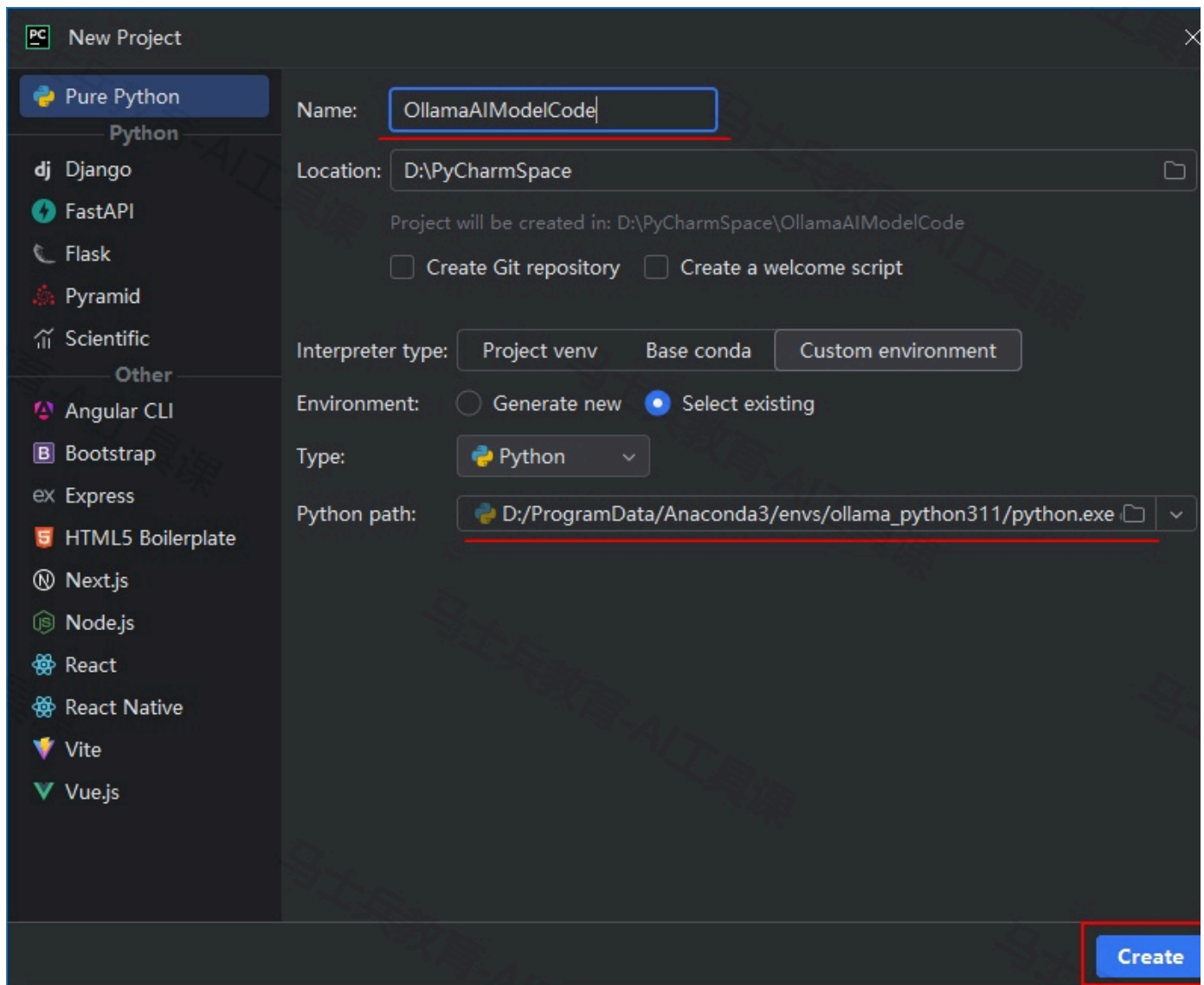
// 输出拼接后的完整响应
System.out.println("模型: " + fullResponse.toString());
} else {
    System.out.println("请求失败，响应代码: " + responseCode);
}

} catch (Exception e) {
    e.printStackTrace();
}
}

```

9.Ollama Python API方式使用模型

这里使用Pycharm创建Python项目，使用python环境为Anconda中的“ollama_python311”。



1. Rest API方式与模型交互

使用Rest API方式与模型交互，需要在python环境中安装requests：pip install requests==2.32.3。

- 与模型一次对话python代码 chat_with_model_test.py

```
# -*- coding: utf8 -*-

import requests
import json

url = "http://localhost:11434/api/chat"

data = {
    "model": "deepseek-r1:1.5b",
    "messages": [{"role": "user", "content": "给我讲一个童话故事"}]
}
```

```

response = requests.post(
    url, json=data, stream=True
) # stream=True 流式返回结果

# 检查返回的状态, 200正常
if response.status_code == 200:
    full_response = [] # 用于存储完整的回复内容
    # 遍历流式响应
    for line in response.iter_lines():
        if line:
            # 解码该行并解析JSON
            decoded_line = line.decode("utf-8")
            # print(f"接收到的行内容: {decoded_line}") # 打印每一行的内容
            try:
                result = json.loads(decoded_line)
                # 提取message中的content字段并添加到full_response列表
                content = result.get("message", {}).get("content", "")
                if content:
                    full_response.append(content)
                # 检查done字段的值
                if result.get("done", False):
                    break # 如果done为true, 结束循环
            except json.JSONDecodeError as e:
                print(f"JSON解析错误: {e}")
    # 输出完整的回复内容
    print("生成的文本:", "".join(full_response))
else:
    print("错误:", response.status_code, response.text)

```

- 与模型多次对话 (携带历史消息) python代码 chat_with_model_realtime.py

```

# -*- coding: utf-8 -*-

import requests
import json

# 定义API的URL
url = "http://localhost:11434/api/chat"

# 初始化对话历史记录
messages = []

def chat_with_model(user_input):
    # 将用户输入添加到对话历史
    messages.append({"role": "user", "content": user_input})

    # 构建请求数据
    data = {

```

```

    "model": "deepseek-r1:1.5b",
    "messages": messages,
    "stream": False # 设置为False以获取完整的响应
}

# 发送POST请求
response = requests.post(url, json=data)

# 检查响应状态码
if response.status_code == 200:
    result = response.json()
    # 提取模型的回复内容
    model_reply = result.get("message", {}).get("content", "")
    if model_reply:
        # 将模型的回复添加到对话历史
        messages.append({"role": "assistant", "content": model_reply})
        return model_reply
    else:
        return "模型未返回有效的回复。"
else:
    return f"请求失败，状态码: {response.status_code}, 错误信息: {response.text}"

def main():
    print("与模型对话开始，输入'退出'结束对话。")
    while True:
        user_input = input("您: ")
        if user_input.lower() == "退出":
            print("对话结束。")
            break
        response = chat_with_model(user_input)
        print(response)

if __name__ == "__main__":
    main()

```

2. 使用Python API方式与模型交互

使用python api方式与模型交互首先需要在python环境中安装ollama : `pip install ollama==0.4.7`。

```

# -*- coding: utf8 -*-
import ollama

# 列出ollama中所有模型
response = ollama.list()

print("ollama所有模型 :", response)

# == 与模型聊天案例 ==

```

```
res = ollama.chat(  
    model="deepseek-r1:1.5b",  
    messages=[  
        {"role": "user", "content": "天空为什么是蓝色的？"},  
    ],  
    stream=False  
)  
print(res["message"]["content"])  
  
# == 流式返回结果 ==  
res = ollama.chat(  
    model="deepseek-r1:1.5b",  
    messages=[  
        {  
            "role": "user",  
            "content": "大海为什么是蓝色的？",  
        },  
    ],  
    stream=True,  
)  
for chunk in res:  
    print(chunk["message"]["content"], end="", flush=True)
```

10.案例一：大模型内容分类

使用大语言模型将如下食物清单进行分类，食物清单文件grocery_list.txt内容如下：

```
苹果  
鸡胸肉  
牛奶  
面包  
胡萝卜  
橙汁  
鸡蛋  
菠菜  
酸奶  
牛肉末  
香蕉  
奶酪  
麦片  
西红柿  
意大利面  
大米  
黄油  
三文鱼  
西兰花  
杏仁  
土豆  
洋葱
```


生菜
草莓
咖啡
茶
糖
面粉
橄榄油
蜂蜜
花生酱
果酱
大蒜
甜椒
蘑菇
虾
香肠
燕麦棒
燕麦片
冰淇淋
汽水
薯片
巧克力
卫生纸
纸巾
洗洁精
洗衣液
洗发水
牙膏

python代码 categorizer.py :

```
import ollama
import os

model = "deepseek-r1:1.5b"

# 输入和输出文件的路径
input_file = "../data/grocery_list.txt"
output_file = "../data/categorized_grocery_list.txt"

# 检查输入文件是否存在
if not os.path.exists(input_file):
    print(f"输入文件 '{input_file}' 未找到。")
    exit(1)

# 从输入文件读取未分类的购物清单
with open(input_file, "r") as f:
    items = f.read().strip()

# 为模型准备提示信息
```

```
prompt = f"""
```

你是一个对购物清单进行分类和排序的助手。

以下是购物清单：

```
{items}
```

请将这些物品分类到适当的类别，例如：农产品、乳制品、肉类、烘焙食品、饮料等

```
"""
```

```
# 发送提示信息并获取响应
```

```
try:
```

```
    response = ollama.generate(model=model, prompt=prompt)
```

```
    generated_text = response.get("response", "")
```

```
    print("==== 分类后的列表: ===== \n")
```

```
    print(generated_text)
```

```
# 将分类后的列表写入输出文件
```

```
with open(output_file, "w") as f:
```

```
    f.write(generated_text.strip())
```

```
    print(f"分类后的购物清单已保存到 '{output_file}'。")
```

```
except Exception as e:
```

```
    print("发生错误:", str(e))
```

代码运行后分类结果如下：

根据购物清单中的物品，我们对它们进行了分类和排序：

```
### 分类结果：
```

```
1. **农产品**
```

- 苹果
- 胡萝卜
- 香肠（香肠食品）
- 土豆（蔬菜）
- 洋葱

```
2. **肉类**
```

- 鸡胸肉
- 牛肉末
- 牛肉
- 大肉（如牛排）
- 肉制品（如牛肉）

```
3. **乳制品**
```

- 牛奶
- 奶酪（奶酪）
- 面板酱

- 鸡胸肉（属肉类，但更精确归类为肉类及其制品）
- 黄油

4. **烘焙食品**

- 麦片

5. **蔬菜**

- 菠菜
- 苹果（属农产品）
- 橙汁、葡萄汁（果汁）
- 二月花
- 冬瓜
- 卵果类（如西兰花、冬瓜）
- 土豆（含淀粉和糖分，属于谷物类）
- 卤叶（属蔬菜）

6. **调味品**

- 糖（可进一步归为甜味剂）
- 橄榄油
- 蜡urate（如大蒜末）
- 蒸干粉（调味料）
- 酥糖（可进一步归为调味剂）
- 杏仁
- 苹果酱（调味品）
- 葡萄汁（属于果汁类）
- 甜椒

7. **饮料**

- 克罗鸡·原味饮料（碳酸饮料）
- 瓶装饮料（含糖分和营养物质）
- 面包（含乳制品）
- 牛奶（乳制品）
- 艾式雪碧
- 二氧化碳水（含糖分和水分）
- 咸菜（含糖分和脂肪）
- 玉米片（调味品）
- 意大利面（面条）
- 大麦片（调味品）
- 酱香料（调味品）

8. **其他**

- 苹果醋
- 苹果酱
- 葡萄汁
- 红糖

最终分类结果：

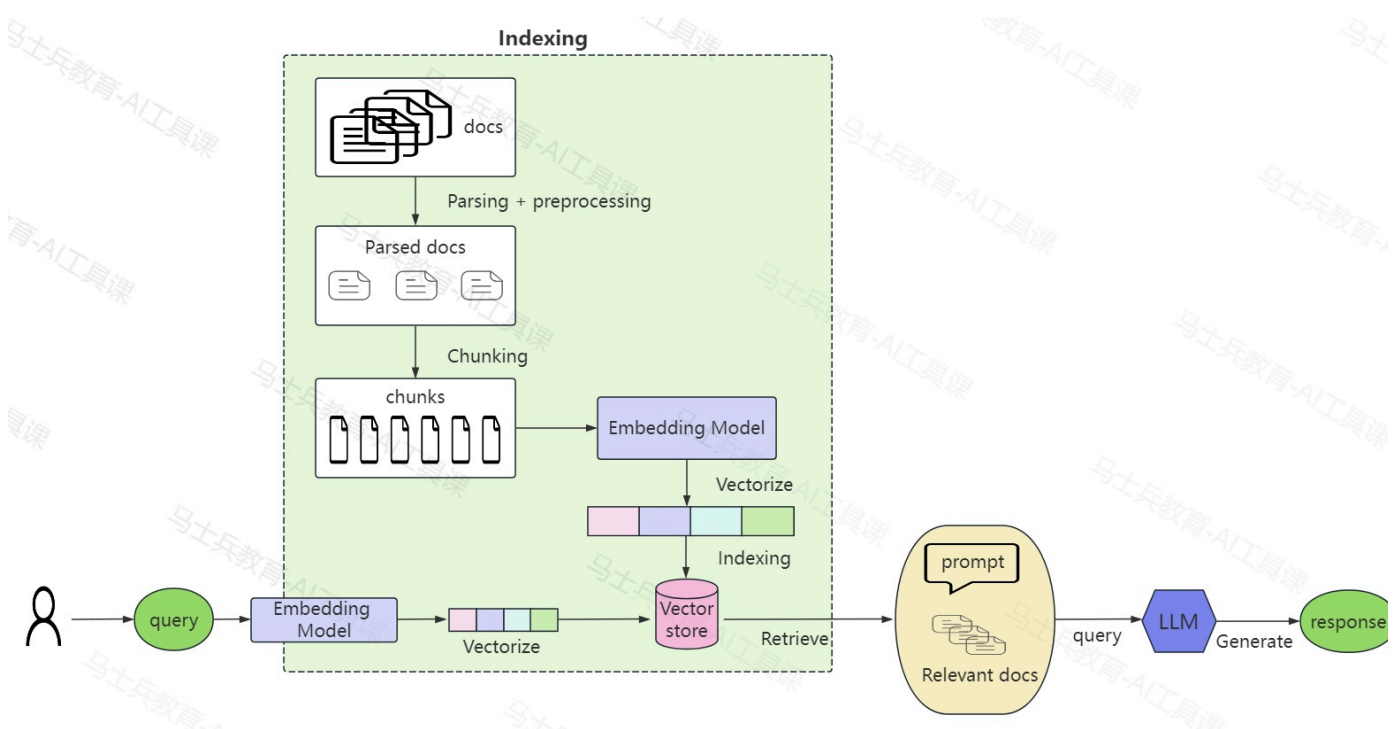
- 农产品：苹果、胡萝卜、香蕉、南瓜（假设），牛油果（如鸡蛋清）、奶酪（奶酪）。
- 肉类：鸡胸肉、牛肉末、牛肉、鸡肉、三文鱼。
- 乳制品：牛奶、酸奶、鸡蛋、豆奶（如黄油）、奶酪（奶酪）、豆腐酱（如醋、蜂蜜）。

- 培养材料：小麦片、酵母粉（调味料），面包、燕麦片、雪梅片（调味品）。
- 清洁产品（调味剂）：糖、橄榄油、蜂蜜、大蒜末、干果（如干盐）、干切面（调味剂），糖（作为甜味剂），
- 饮料类：碳酸饮料、啤酒、苏打水（含糖分和营养），碳酸乳、汽水、美式冰水（含糖分），可乐、橙汁、乳制品
- 咸菜：红辣椒、苦瓜、番茄、黄瓜（含糖分和脂肪）。
- 甜味剂：干果（如干果），干切面（调味料），干果粉（调味剂）。

11.案例二：基于Ollama+Deepseek构建RAG系统

检索增强生成（Retrieval-Augmented Generation，简称 RAG）是一种将大型语言模型（LLM）与外部知识源相结合的人工智能技术。通过在生成响应前检索相关信息，RAG 能够为模型提供最新且特定领域的知识，从而提高回答的准确性和相关性。

RAG的工作原理如下：



我们有自己的文档（url、pdf、txt、数据库等），这些文档就是用户本地的知识库，经过解析处理为chunks（小块）文档，然后这些文档通过嵌入模型（Embedding Model）将这些文本信息转换成向量（Vector），即转换成数字表示，这就是嵌入的含义，这些向量保存在向量数据库中，这个过程我们称为indexing(索引)。

当用户提出问题时，这些问题也会通过Embedding Model转换成向量，然后去向量数据库中去检索（Retrieve），检索到与问题相关的文档，然后把这些内容打包，结合提示词（Prompt）一起传递给大模型，这样大模型就有了如何回答该问题的上下文概念，结合大模型的推理能力，形成连贯答案返回给用户，这就是RAG工作流程。

下面我们实现使用Ollama Embedding和deepseek大语言模型的RAG系统。该系统通过Langchain来解析PDF文件，将PDF文件分割成多个chunks，然后使用Ollama 嵌入模型将这些小块向量化，将所有信息存储到ChromaDB数据库中。然后再次使用Langchain的查询检索器将

查询内容转换成嵌入向量，并在ChromaDB数据库中搜索所有嵌入内容，匹配最相似的信息片段，将这些内容结合prompt传递给大语言模型，最终形成我们想要的答案。以上系统中使用的大语言模型可以随意切换成其他的大语言模型，都是通用的。

备注：LangChain是一个开源框架，通过LangChain可以构建由大语言模型（LLM）驱动的应用程序，它提供了一套工具和接口，简化了与语言模型的集成过程，支持开发聊天机器人、文档分析、代码生成等多种应用。

按照如下步骤构建基于Ollama Embedding和deepseek大语言模型的RAG系统。

1) 在Anconda ollama_python311环境中安装依赖包

进入到\$ANCONDA_HOME/env/ollama_python311/Scripts目录中，在该路径中打开cmd，执行如下命令安装所需依赖。

```
pip install -r D:\PyCharmSpace\OllamaAIModelCode\requirements.txt
```

requirements.txt内容如下：

```
ollama==0.4.7
chromadb==0.6.2
langchain==0.3.19
langchain-core==0.3.47
langchain-community==0.3.18
langchain_ollama==0.3.0
unstructured==0.16.17
unstructured-client==0.31.3
sentence-transformers==3.3.1
```

2) 编写代码

```
### 步骤
# 1.导入PDF文件
# 2.从PDF文件中提取文本并将其拆分为chunk
# 3.将这些小块发送到embedding 嵌入模型
# 4.将生成的embeddings嵌入保存到向量数据库
# 5.在向量数据库上执行相似性搜索以查找相似文档
# 6.检索相似文档并将结果反馈给用户

## 安装所需的软件包：pip install -r requirements.txt

## 1. 导入所需的库
from langchain.document_loaders import PyPDFLoader
from langchain_ollama import OllamaEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain.prompts import ChatPromptTemplate, PromptTemplate
from langchain_core.output_parsers import StrOutputParser
```

```

from langchain_ollama import ChatOllama

from langchain_core.runnables import RunnablePassthrough
from langchain.retrievers.multi_query import MultiQueryRetriever

## 2. 设置文档路径和模型
doc_path = "../data/《中华人民共和国义务教育法》.pdf"
# model = "llama3.2"
model = "deepseek-r1:1.5b"
# model = "gemma:2b"
# model = "llava:7b"

## 3. 本地PDF文件加载
if doc_path:
    loader = PyPDFLoader(file_path=doc_path)
    data = loader.load()
    print("PDF文档加载完成....")
else:
    print("请上传PDF文件")

    # 预览第一页内容,打印前100个字符
content = data[0].page_content
print(content[:100])

# ==== 结束PDF加载 ====

# ==== 从PDF文件中提取文本并拆分成小块(Chunks) ====
# 拆分和分块,每个块的大小为1200字符,重叠部分为300字符
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1200, chunk_overlap=300)
chunks = text_splitter.split_documents(data)
print("文档 split 完成....")

print(f"chunks总数: {len(chunks)}")
# print(f"chunk[0]示例: {chunks[0]}")

# ===== 添加到向量数据库 =====
import ollama

# ollama.pull("nomic-embed-text")

vector_db = Chroma.from_documents(
    documents=chunks,
    embedding=OllamaEmbeddings(model="nomic-embed-text"), #使用 nomic-embed-text 模型生成文本嵌入
    collection_name="simple-rag", #存储在名为simple-rag的集合中
)
print("添加到向量数据库完成....")

```



```
## === 检索 ===
```

```
# 设置模型
```

```
llm = ChatOllama(model=model)
```

```
# 使用多重查询检索技术，根据用户问题生成多个查询，从向量数据库中检索相关文档
```

```
QUERY_PROMPT = PromptTemplate(  
    input_variables=["question"],  
    template="""你是一个人工智能语言模型助手，你的任务是生成五个不同版本的用户问题，  
    以便从向量数据库中检索相关文档。通过对用户问题的多角度改写，你可以帮助用户克服基于距离的相似性搜索  
    请提供这些不同版本的问题，并使用换行符分隔。  
    原始问题: {question}""",  
)
```

```
retriever = MultiQueryRetriever.from_llm(  
    vector_db.as_retriever(), llm, prompt=QUERY_PROMPT  
)
```

```
# RAG 提示模板
```

```
# {context} 是一个占位符，用于在运行时插入特定的上下文信息。
```

```
# 在 LangChain 框架中，提示模板（PromptTemplate）允许通过占位符来动态地构建提示，以适应不同的输入和
```

```
# 在实际应用中，{context} 通常由检索器（retriever）从向量数据库或其他数据源中获取的相关文档或信息填充。
```

```
# 当用户提出问题时，系统会检索与该问题相关的内容，并将这些内容作为 context 插入到提示模板中。
```

```
# 这样，语言模型就能够基于提供的上下文来生成更准确和相关的回答。
```

```
template = """请仅根据以下提供的上下文回答问题：
```

```
{context}
```

```
问题: {question}
```

```
"""
```

```
# 生成提示的对象
```

```
prompt = ChatPromptTemplate.from_template(template)
```

```
# 管道操作符（|），用于实现函数或对象的链式调用，参数解释如下：
```

```
# retriever:检索向量数据库，获取相关的文本内容
```

```
# RunnablePassthrough:直接传递用户输入的问题
```

```
# prompt:查询提示
```

```
# llm:使用之前定义的 ChatOllama 模型生成回答
```

```
# StrOutputParser：解析模型返回的字符串，并将其格式化为最终的输出
```

```
chain = (  
    {"context": retriever, "question": RunnablePassthrough()}  
    | prompt  
    | llm  
    | StrOutputParser()  
)
```

```
# 提问示例
```



```
# res = chain.invoke(input="你是谁 ? ")
# res = chain.invoke(input="你现在拥有什么知识 ? ")
# res = chain.invoke(input="第二十三条内容是什么 ? ")
# print(f"模型: {response}")
```

```
# 实时对话循环
```

```
print("请输入您的问题（输入 '退出' 结束对话）：")
```

```
while True:
```

```
    user_input = input("用户: ")
```

```
    if user_input.lower() in ['退出', 'exit']:
```

```
        print("结束对话。")
```

```
        break
```

```
    response = chain.invoke(input=user_input)
```

```
    print(f"模型: {response}")
```