1. typeddict是什么：

TypedDict 是 Python 中 typing 模块的一个类，用于定义字典的类型注解，它允许您指定字典中每个键的类型。

from typing import TypedDict

*# 定义一个 TypedDict 类型*

class Person(TypedDict):

name: str

age: int

is\_student: bool

*# 创建符合 Person 类型的字典*

person1: Person = {

"name": "Alice",

"age": 25,

"is\_student": True}

*# 这将导致类型检查错误，因为 'age' 的类型不匹配*

person2: Person = {

"name": "Bob",

"age": "30", *# 类型错误：str 不能赋值给 int*

"is\_student": False}

1. 定义类型:通过继承typeddict，可以定义一个类型，其中每个键都有指定的类型。
2. 类型检查：在支持类型检查的环境中，typedict可以确保字典的键和值的类型符合定义
3. 可选键：可以通过total=False参数指定某些键是可选的。

可选键示例：  
from typing import TypedDict, Optional

class Person(TypedDict, total=False):

name: str

age: int

is\_student: bool

address: str *# 可选键*

在这个示例中，address 是一个可选键，字典中可以存在或不存在该键。

1. typing模块：

typing是 Python 的一个内置模块，用于支持类型注解（Type Hints）。类型注解可以帮助开发者在编写代码时明确变量、函数参数和返回值的类型，提高代码的可读性和可维护性。

1. Annotated：  
   Annotated 是 typing 模块中的一个特殊类型注解，用于为现有类型添加额外的信息或元数据。它允许您在不改变类型本身的情况下，附加一些额外的注解或约束。

from typing import Annotatedfrom typing\_extensions import TypeGuard

*# 定义一个类型注解，表示一个正整数*

PositiveInt = Annotated[int, lambda x: x > 0]

def is\_positive(value: int) -> TypeGuard[PositiveInt]:

return value > 0

def process\_positive\_number(number: PositiveInt) -> None:

print(f"Processing positive number: {number}")

*# 测试*

process\_positive\_number(10) *# 正常调用*

process\_positive\_number(-5) *# 类型检查工具会警告，因为 -5 不是正数*

4.

from langchain\_anthropic import ChatAnthropic

llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")

def chatbot(state: State):

return {"messages": [llm.invoke(state["messages"])]}

*# The first argument is the unique node name# The second argument is the function or object that will be called whenever# the node is used.*

graph\_builder.add\_node("chatbot", chatbot)

这段代码展示了如何使用 langchain\_anthropic 库中的 ChatAnthropic 类来创建一个语言模型实例，并将其集成到一个状态图中，用于构建一个简单的聊天机器人。

4.1

定义聊天机器人函数：

def chatbot(state: State)：定义一个名为 chatbot 的函数，它接受一个 state 参数，这个参数是一个 State 类型的状态字典。

return {"messages": [llm.invoke(state["messages"])]}：在函数内部，使用语言模型实例 llm 的 invoke 方法处理状态中的消息，并将生成的响应作为新的消息列表返回。

4.2

graph\_builder.add\_node("chatbot", chatbot)：将 chatbot 函数作为节点添加到状态图构建器 graph\_builder 中。第一个参数 "chatbot" 是节点的唯一名称，第二个参数是当节点被使用时将被调用的函数。

5.

from IPython.display import Image, display try: display(Image(graph.get\_graph().draw\_mermaid\_png())) except Exception: # This requires some extra dependencies and is optional pass

5.1导入模块：

from IPython.display import Image, display：从 IPython.display 模块中导入 Image 和 display 函数。这些函数用于在 Jupyter Notebook 中显示图像。

5.2尝试显示图像：

try 块：尝试执行显示图像的代码。

graph.get\_graph().draw\_mermaid\_png()：调用 graph 对象的 get\_graph 方法获取图表对象，然后调用 draw\_mermaid\_png 方法生成 Mermaid.js 格式的图表图像数据。

Image(...)：将生成的图像数据封装为 Image 对象。

display(...)：显示 Image 对象。

5.3异常处理：

except Exception：如果在尝试显示图像时发生任何异常（例如缺少依赖项），捕获异常并进入 except 块。

pass：在 except 块中什么都不做，只是忽略错误。注释说明这需要一些额外的依赖项，并且是可选的。

6.graph.stream

graph.stream 方法，它接收一个包含用户输入的字典，并返回一个可迭代对象。

输入的格式：{"messages": [{"role": "user", "content": user\_input}]} 是一个标准的消息格式，符合许多对话系统的输入规范。

输出的处理：for event in graph.stream(...) 表明这是一个流式处理的过程，每次迭代可能返回一个事件对象，而事件中包含与用户输入相关的处理结果。

【参数 {"messages": [...]}：

参数是一个字典，键为 "messages"，值是一个列表，列表中包含一个字典，表示用户输入的消息。

消息字典的结构通常包括 "role" 和 "content" 两个键，分别表示消息的角色（这里是用户）和消息内容。】

可能的应用场景：结合上下文，这段代码可能用于对话系统或知识图谱的交互，用户输入被传递给某个处理模块，模块返回逐步的响应。

7.bind\_tools方法：用于将语言模型与一组工具绑定。

llm\_with\_tools:这是绑定后的对象，表示语言模型已经具备了调用这些工具的能力。

在后续的推理过程中，模型可以根据需要动态调用这些工具。

示例场景：  
假设你正在构建一个智能客服系统，语言模型需要调用以下工具：

查询天气的 API

订单查询的函数

支付处理的服务

通过 bind\_tools，你可以将这些工具与语言模型绑定，使模型在回答用户问题时能够调用这些工具来获取实时信息或执行操作。

*# 定义工具*def get\_weather(location):

*# 模拟查询天气的函数*

return f"晴天，温度 25°C，地点：{location}"

tools = [

{"name": "get\_weather", "description": "获取指定地点的天气信息", "func": get\_weather}]

*# 假设 llm 是一个语言模型对象*

llm\_with\_tools = llm.bind\_tools(tools)

*# 使用绑定后的模型*

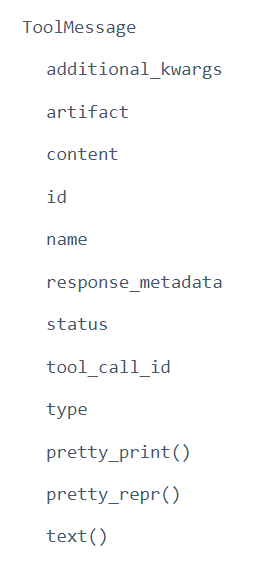
user\_input = "北京今天的天气如何？"

response = llm\_with\_tools.generate(user\_input)

print(response)

1. 类langchain\_core.message.tool

ToolMessages用于将执行工具的结果传递回模型的消息，通常结果在content字段内编码。

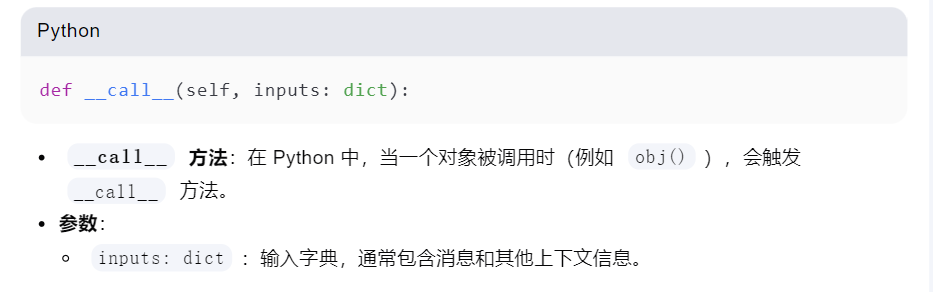
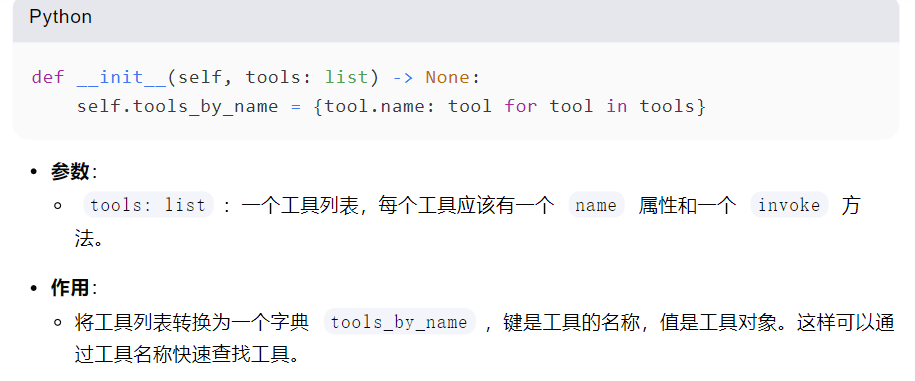


1. additional\_kwargs[dict]；表示额外的关键字参数，通常以字典的形式传递，键是参数名，值是对应的参数值
2. Artifact：any=none:工具执行的工件，不打算发给模型
3. Content：str | list[str | dict] [required]（可以是字符串，或者包含字符串和字典的列表。这个参数是必需的）
4. Id:str | None = None，消息的可选唯一标识符，类型为字符串或 None，默认为 None
5. Name：
6. Response\_metadata:目前从 BaseMessage 继承，但未使用。类型为字典，是可选的（[Optional]）
7. Status：Literal['success', 'error'] = 'success'含义：工具调用的状态，可以是 'success' 或 'error'，默认值为 'success'。

用途：表示工具调用是否成功。

1. Tool\_call\_id：
2. Type:Literal['tool'] = 'tool'
3. Pretty\_print()
4. Pretty\_repr()
5. Text()

\_--------3.10-------------------------

1. 





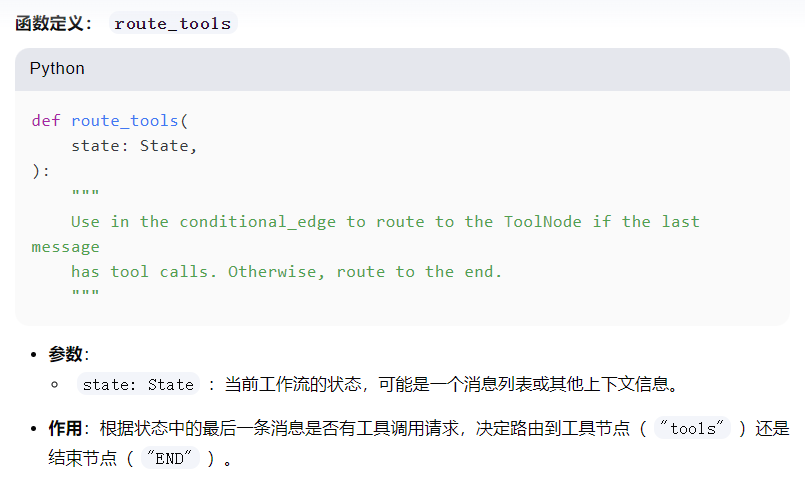
ToolMessage：创建一个 ToolMessage 对象，用于封装工具调用的结果。

content=json.dumps(tool\_result)：将工具结果转换为 JSON 格式的字符串，作为消息的内容。

name=tool\_call["name"]：设置消息的名称为工具的名称。

tool\_call\_id=tool\_call["id"]：设置消息的工具调用 ID，用于关联工具调用和结果。

outputs.append(...)：将创建的 ToolMessage 对象添加到输出列表 outputs 中。

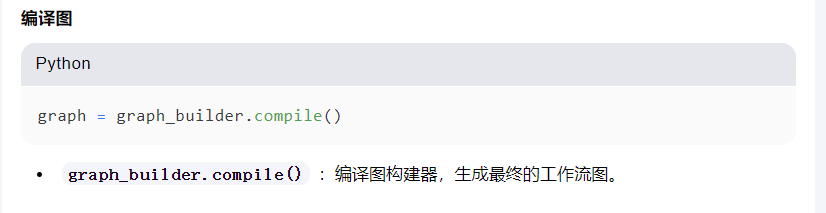












------------------3.11---------------------------------------

1.今日困难：  
Error occurred while generating graph image: ('Connection aborted.', ConnectionResetError(10054, '远程主机强迫关闭了一个现有的连

接。', None, 10054, None))

这是因为爬取页面次数过多导致，学习代码任务暂时停止。

现在更新错误原因：

第一个代码在生成图形图像时出现 ('Connection aborted.', ConnectionResetError(10054, '远程主机强迫关闭了一个现有的连接。', None, 10054, None)) 错误，这个错误通常是在与远程服务器建立连接时，远程服务器主动关闭了连接。结合代码来看，报错代码段是：try:

graph\_png = app.get\_graph().draw\_mermaid\_png()

with open("langgraph\_base.png", "wb") as f:

f.write(graph\_png)except Exception as e:

print(f"Error occurred while generating graph image: {e}")

【得出结论，是因为生成图片的函数，需要挂梯子才可以生成，所以再生成图片以后，就可以把梯子取消掉了】

1. 学习MemorySaver()

用于保存和管理聊天过程，通过将 MemorySaver 传递给 compile 方法，图的状态将在整个对话过程中保存。

总结:今天学习了如何生成langgrpha图片并且了解到，报错的原因是什么，并且调用了谯的tools对agent\_base中的寻求天气的智能体进行的代码的缝合，并且使其可以正常运行，解下来，明天的任务：看明白谯师兄的tools代码是怎么写的，尝试可以让天气agent也正常运行。并且尝试让生成的png可以保存在子目录下。

图片生成与访问次数无关，只是第一次生成图片的时候，是需要挂梯子，现在先不融入日志，记忆等等内容，先进行简单的langgraph生成，把agent\_base的tools改好，看到代码的内容逻辑之后，去b站把那两个视频给他看了。今天的任务就到此为止啦，辛苦啦！孙姐！

------------3.12------------

1.from rich.console import Console：从rich.console模块中导入Console类，rich是一个用于在终端中输出美观格式化文本的库，Console类可以方便地进行文本输出。

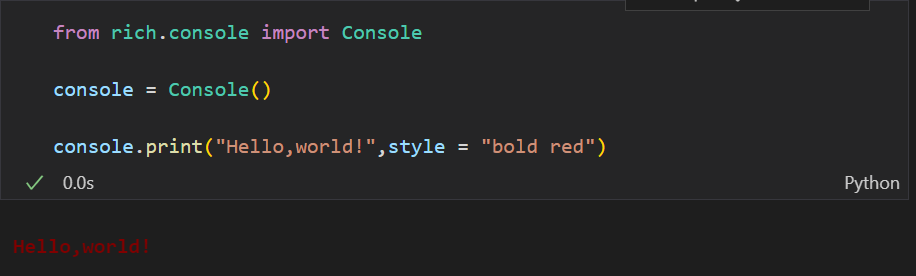
在 Python 中，Console类来自于rich库。rich是一个用于在终端中输出美观、格式化文本的第三方库，它支持多种样式设置，如颜色、加粗、斜体等，还能处理表格、进度条等复杂的显示内容。Console类是rich库的核心类之一，它提供了一个接口，用于在终端中以丰富的格式输出文本和其他内容。

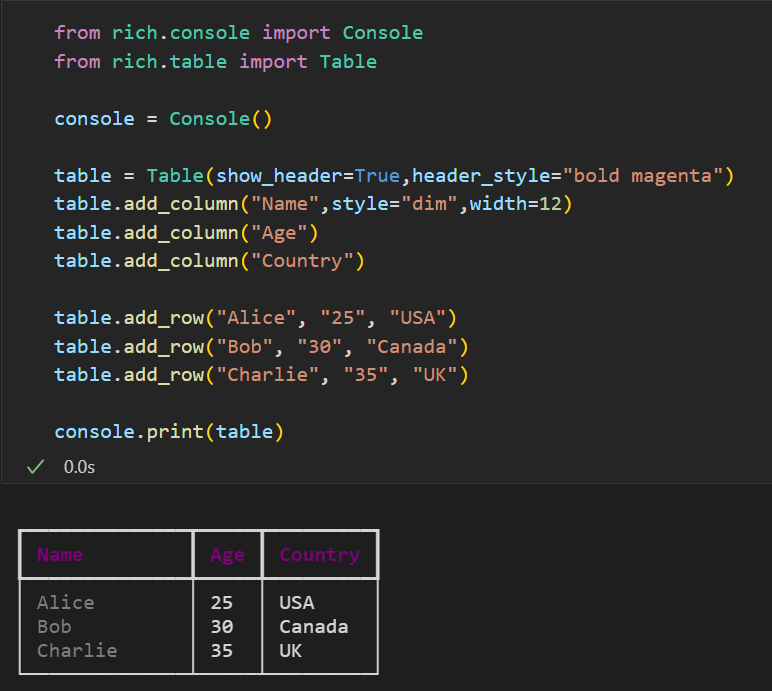
美化输出：可以为文本添加颜色、样式（如加粗、斜体、下划线等），使输出在终端中更加醒目和易读。

处理复杂内容：支持输出表格、进度条、树状结构等复杂的数据结构，方便用户展示和分析数据。

跨平台兼容性：确保在不同的操作系统（如 Windows、Linux、macOS）的终端上都能正常显示格式化的内容。

print方法：类似于 Python 内置的print函数，但支持丰富的样式设置





* ****性能问题****：虽然rich库可以美化输出，但在处理大量数据时，频繁使用rich的功能可能会影响性能。因此，在性能敏感的场景中需要谨慎使用。
* ****兼容性问题****：尽管rich库尽量保证跨平台兼容性，但在某些特殊的终端环境中，可能会出现显示异常的情况。可以通过调整Console类的参数（如force\_terminal）来解决部分兼容性问题。

**接着再来学习force\_terminal=False的核心作用**

该参数用于控制console是否强制使用终端模式，具体含义如下：

force\_terminal=True：强制使用终端模式，无论当前环境是否支持。

force\_terminal=False（默认值）：自动检测当前环境是否支持终端样式，若不支持则回退为普通文本输出。

1. 不加该参数的默认行为：console = Console() # 等同于 console = Console(force\_terminal=False)
2. 显式设置 force\_terminal=True 的行为，console=Console(force\_terminal=True)

| **参数值** | **行为描述** | **适用场景** |
| --- | --- | --- |
| force\_terminal=True | 强制终端模式，可能在不支持的环境中显示乱码。 | 已知环境支持 ANSI，且必须启用样式。 |
| force\_terminal=False（默认） | 自动检测环境，兼容性强，推荐使用。 | 通用场景，确保不同环境下都能正常输出。 |

**2.**

requests库：用于发送http请求，与网络接口进行交互；

json解码：JSON 是一种轻量级的数据交换格式json 库提供了 JSON 数据的编码（将 Python 对象转换为 JSON 字符串）和解码（将 JSON 字符串转换为 Python 对象）功能。

##### **JSON 编码（将 Python 对象转换为 JSON 字符串）**

import json

# Python 字典

data = {

'name': 'John',

'age': 30,

'city': 'New York'}

# 将 Python 字典转换为 JSON 字符串

json\_str = json.dumps(data)print(json\_str)

##### **JSON 解码（将 JSON 字符串转换为 Python 对象）**

import json

# JSON 字符串

json\_str = '{"name": "John", "age": 30, "city": "New York"}'

# 将 JSON 字符串转换为 Python 字典

data = json.loads(json\_str)print(data)

logging库：用于记录程序运行过程中的日志信息。在开发和调试过程中，日志信息可以帮助我们了解程序的运行状态，排查问题。logging 库提供了不同级别的日志记录（如 DEBUG、INFO、WARNING、ERROR、CRITICAL），可以根据需要设置日志的输出级别和输出格式。

##### **基本使用**

import logging

# 配置日志记录

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')

# 记录不同级别的日志

logging.debug('这是一个调试信息')

logging.info('这是一个普通信息')

logging.warning('这是一个警告信息')

logging.error('这是一个错误信息')

logging.critical('这是一个严重错误信息')

import logging

# 配置日志记录到文件

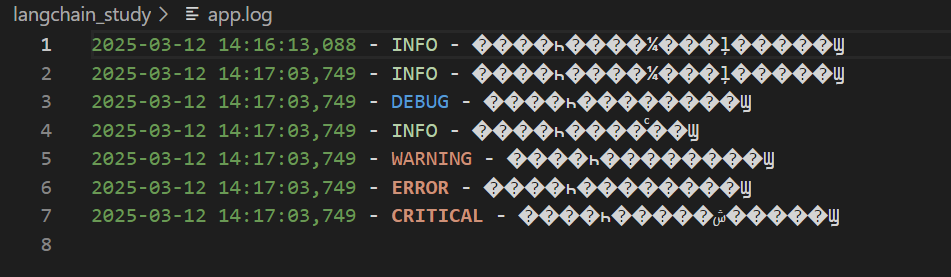
logging.basicConfig(filename='app.log', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')

# 记录日志

logging.info('这是一条记录到文件的信息')

logging.basicConfig 方法用于配置日志记录，level=logging.DEBUG 表示设置日志的输出级别为 DEBUG，即会记录所有级别的日志信息。format 参数用于设置日志的输出格式，%(asctime)s 表示日志记录的时间，%(levelname)s 表示日志的级别，%(message)s 表示日志的具体内容。

运行过后得到的结果如下：



应该是没有utf-8的缘故，故再次，先补充uft-8的意义是什么。

首先来了解以下utf-8的本质（Unicode Transformation Format 8-bit）

Python3的str类型默认使用的Unicode编码，单是文件读写时需要显式指定编码：

字节与字符串的转换

str.encode('utf-8')：将字符串转为 UTF-8 字节流

bytes.decode('utf-8')：将 UTF-8 字节流转为字符串

**为什么要配置utf-8?**

跨平台一致性

不同操作系统默认编码不同（Windows 默认为 GBK，Linux/macOS 默认为 UTF-8），显式指定 UTF-8 可避免：

中文乱码（如 Windows 打开 Linux 生成的文件）

非法字符错误（如读取非 ASCII 字符时）

网络传输标准

HTTP 协议、JSON 格式等现代通信标准均强制要求 UTF-8 编码：

python

# 网络请求示例

import requests

response = requests.get('https://api.example.com/data', headers={'Accept-Charset': 'utf-8'})

文件存储规范

国际标准文档（如 XML、CSV）和代码文件（.py）推荐使用 UTF-8，确保：

代码中的中文注释正常显示

数据文件在不同环境可正确解析

**配置时只需要在需要的地方加一个encoding="utf-8"**

**继续学习logging**

**1.level=logging.INFO**

含义：此参数设置了日志记录的级别。日志级别决定了哪些日志信息会被记录下来。logging 模块提供了多个日志级别，从低到高依次为 DEBUG、INFO、WARNING、ERROR、CRITICAL。当设置为 logging.INFO 时，只有 INFO、WARNING、ERROR 和 CRITICAL 级别的日志信息会被记录，而 DEBUG 级别的信息会被忽略。

import logging

logging.basicConfig(

level=logging.INFO

)

logging.debug("这是一条调试信息") # 不会被记录

logging.info("这是一条普通信息") # 会被记录

### **format="%(asctime)s - %(levelname)s - %(message)s"**

含义：该参数定义了日志信息的输出格式。格式字符串中可以包含一些占位符，用于表示不同的日志信息。

%(asctime)s：表示日志记录的时间，会显示具体的日期和时间。

%(levelname)s：表示日志的级别，如 DEBUG、INFO 等。

%(message)s：表示具体的日志内容。

import logging

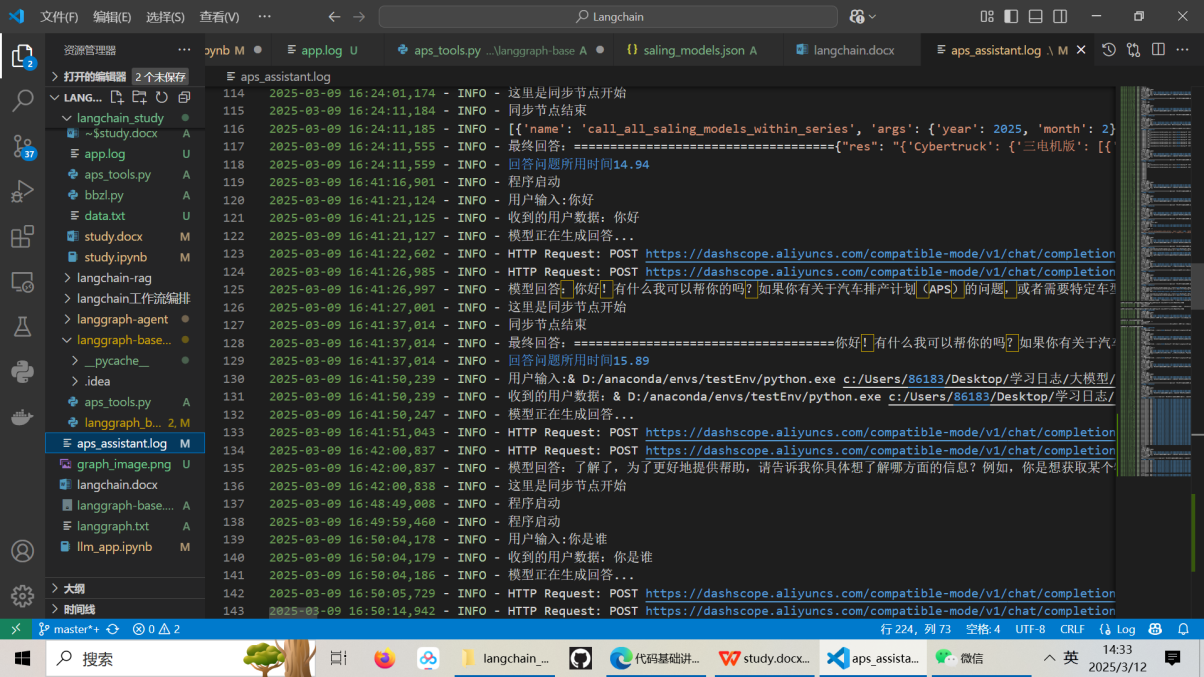
logging.basicConfig(

format="%(asctime)s - %(levelname)s - %(message)s"

)

logging.info("这是一条示例信息")

#输出2024-12-01 10:30:00,123 - INFO - 这是一条示例信息



**3.**

**Agent=LLM+观察+思考+行动+记忆**

from langchain.agents import create\_react\_agent

agent = create\_react\_agent(llm, tools, prompt)

所以我们想要创建一个agent，需要llm,tools，prompt

出现了问题：

在更改提示词模板以后，ai 可以按照提示词模板输出自己的自我认知，但是现在当我调入了更多的模型model\_series,导致了ai 的自我认知出现错误。

这是因为在调用模型的时候，把我的提示词给覆盖了，

1）在当前的 prompt\_template 中，用户在示例回答中写了 “我是 xsun，一个专门用来搜索日期的智能机器人”，但系统消息部分可能没有包含这一信息。正确的做法是在系统消息中明确说明 AI 的身份，而不是在示例回答中。

因此，解决方案应该是在提示词模板中添加系统消息，明确 AI 的身份为 xsun。同时，确保在创建 ChatPromptTemplate 时包含系统消息和工具描述。

2）另外，用户可能混淆了不同的提示词模板结构。原来的 react\_agent 可能期望特定的消息结构，包括系统消息、工具描述等。直接使用 from\_template 可能没有正确包含这些部分，导致系统消息被覆盖。

需要调整 prompt\_template，确保包含系统消息，并且在创建 ChatPromptTemplate 时正确组合消息部分。例如，使用 HumanMessagePromptTemplate 和 SystemMessage 来构建完整的提示词。

改正以后：

prompt = ChatPromptTemplate.from\_messages([

    system\_message,

    SystemMessagePromptTemplate.from\_template(tools\_description),

    HumanMessagePromptTemplate.from\_template(human\_template)

])

其中三个部分的内容分别如下：

system\_message = SystemMessagePromptTemplate.from\_template(

    "你是一个搜索日期的智能机器人xsun。必须使用提供的工具来回答问题。"

)

# 定义工具描述

tools\_description = """

你有以下工具可用：

{tools}

"""

# 定义用户消息模板

human\_template = """

Answer the following questions as best you can. Use the following format:

Question: the input question you must answer

Thought: you should always think about what to do

Action: the action to take, should be one of [{tool\_names}]

Action Input: the input to the action，如果其中有日期，请确保只输入日期，格式为:YYYY-MM-DD，不要有任何其它字符

Observation: the result of the action，如果其中有日期，请确保输出的日期格式为:YYYY-MM-DD，不要有任何其它字符

... (this Thought/Action/Action Input/Observation can repeat N times)

Thought: I now know the final answer

Final Answer: the final answer to the original input question

Begin! Let's think step by step. Take a deep breath.

Question: {input}

Thought:{agent\_scratchpad}

"""

4.

def weekday(date\_str: str) -> str:

    """Convert date to weekday name"""

    d = parser.parse(date\_str)

    return calendar.day\_name[d.weekday()]

def weekday(date\_str: str) -> str:：这是函数的定义部分。

date\_str: str 表示函数接受一个参数 date\_str，并且这个参数的类型应该是字符串（str）。

-> str 是函数的返回值类型注解，表明该函数将返回一个字符串。

"""Convert date to weekday name"""：这是函数的文档字符串（docstring），用于对函数的功能进行简要描述。当使用 help(weekday) 时，会显示这段描述信息。

parser.parse(date\_str)：parser 是从 dateutil.parser 模块导入的，parse 方法的作用是将输入的日期字符串 date\_str 解析为 datetime 对象。dateutil.parser 模块非常强大，它可以处理多种不同格式的日期字符串，例如 "2024-10-01"、"October 1, 2024" 等。

d 是一个 datetime 对象，它包含了日期和时间的信息

d.weekday()：weekday() 是 datetime 对象的一个方法，它返回一个整数，表示该日期是星期几。其中，0 表示星期一，1 表示星期二，以此类推，6 表示星期日。

calendar.day\_name：calendar 是 Python 的标准库模块，day\_name 是 calendar 模块中的一个元组，它包含了一周七天的名称，顺序为 ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')。

calendar.day\_name[d.weekday()]：通过 d.weekday() 得到的整数索引，从 calendar.day\_name 元组中获取对应的星期几的名称，最后将其作为函数的返回值。

**今日总结：学习了console()函数的具体用法，requestion,json,logging的用法，还有uft-8.解决了提示词模板的问题，可以让模型进行良好的自我认知。最后学习了toolS的建立以及使用方法，做了一个笨笨的ai搜索日期的模型。仍需下一步改进**

------------------3.13-------------------------------------------------

新的问题：不确定是否agent的内容都参与了，以及如何选择会话内容，去保存记忆。删除记忆等等功能应该如何实现。目前的代码已经传输到GitHub。

1. 解决：

def weekday(date\_str: str) -> str:

    """Convert date to weekday name"""

    print(f"Tool 'weekday' is called with date: {date\_str}")  # 添加打印语句

    d = parser.parse(date\_str)

    return calendar.day\_name[d.weekday()]

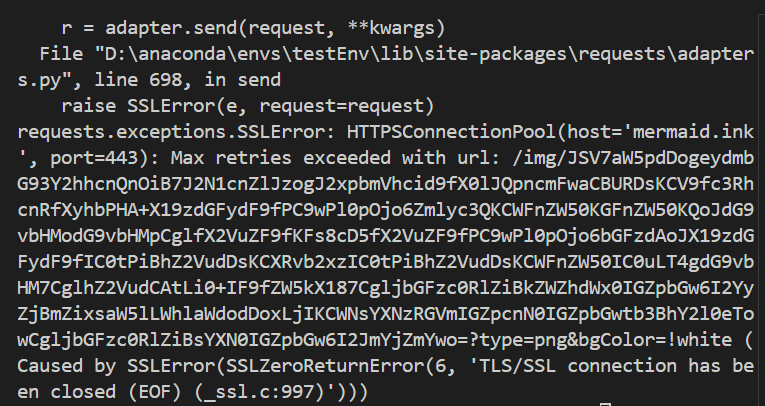
1. agent能够存储回忆并有历史记录：

from langgraph.checkpoint.memory import MemorySaver

memory = MemorySaver()

今日出现的新问题：

1. 响应时间很慢
2. 偶尔会报错如下：

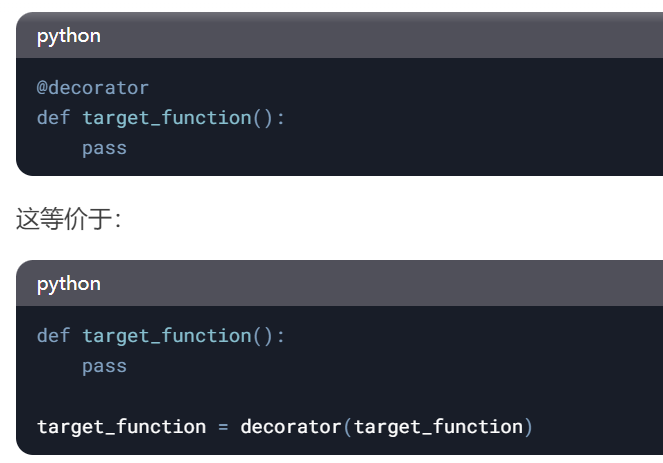


----------------------------------0314---------------------------

1. 装饰器学习：

@ 符号通常用于 装饰器（Decorator） 的声明。装饰器是一种特殊的函数或类，用于修改或扩展其他函数或类的行为。装饰器的核心思想是 在不修改原函数或类代码的情况下，为其添加额外的功能。

装饰器的本质是一个 高阶函数，它接受一个函数作为输入，并返回一个新的函数。通过 @ 符号，我们可以将装饰器应用到目标函数或类上。



在函数执行前自动记录日志：

def log\_decorator(func):

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

print(f"Calling function: {func.\_\_name\_\_}")

result = func(\*args, \*\*kwargs)

print(f"Finished calling: {func.\_\_name\_\_}")

return result

return wrapper

@log\_decorator

def greet(name):

print(f"Hello, {name}!")

greet("Alice")

输出：

Calling function: greet

Hello, Alice!

Finished calling: greet

2. 权限检查

在函数执行前检查用户权限：

python

复制

def check\_permission(func):

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

user = kwargs.get("user", "guest")

if user == "admin":

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

else:

raise PermissionError("Permission denied")

return wrapper

@check\_permission

def delete\_file(user):

print("File deleted")

delete\_file(user="admin") # 正常执行

delete\_file(user="guest") # 抛出 PermissionError

3. 缓存

缓存函数的返回值，避免重复计算：

python

复制

def cache\_decorator(func):

cache = {}

def wrapper(\*args):

if args in cache:

print("Returning cached result")

return cache[args]

result = func(\*args)

cache[args] = result

return result

return wrapper

@cache\_decorator

def expensive\_operation(x):

print("Performing expensive operation")

return x \* x

print(expensive\_operation(2)) # 第一次计算

print(expensive\_operation(2)) # 返回缓存结果

4. 性能测试

测量函数的执行时间：

python

复制

import time

def timing\_decorator(func):

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

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

print(f"{func.\_\_name\_\_} took {end\_time - start\_time:.2f} seconds")

return result

return wrapper

@timing\_decorator

def slow\_function():

time.sleep(2)

slow\_function()

5. 类装饰器

装饰器不仅可以用于函数，还可以用于类：

python

复制

def add\_method(cls):

def new\_method(self):

return "This is a new method"

cls.new\_method = new\_method

return cls

@add\_method

class MyClass:

pass

obj = MyClass()

print(obj.new\_method()) # 输出: This is a new method

装饰器的原理

装饰器的核心原理是 闭包（Closure）。闭包是指一个函数可以访问其定义时的作用域中的变量，即使该作用域已经结束。

例如：

python

复制

def decorator(func):

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

print("Before calling the function")

result = func(\*args, \*\*kwargs)

print("After calling the function")

return result

return wrapper

在这里，wrapper 函数可以访问 decorator 函数的参数 func，即使 decorator 已经执行完毕。

多层装饰器

装饰器可以叠加使用，执行顺序是从下往上：

python

复制

@decorator1

@decorator2

def my\_function():

pass

等价于：

python

复制

my\_function = decorator1(decorator2(my\_function))

带参数的装饰器

如果装饰器本身需要参数，可以再嵌套一层函数：

python

复制

def repeat(n):

def decorator(func):

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

for \_ in range(n):

result = func(\*args, \*\*kwargs)

return result

return wrapper

return decorator

@repeat(3)

def say\_hello():

print("Hello!")

say\_hello()

输出：

复制

Hello!

Hello!

Hello!

总结

@ 符号 用于声明装饰器，是一种语法糖，简化了装饰器的使用。

装饰器的作用 是动态扩展函数或类的功能，而不修改其原始代码。

装饰器的核心原理 是闭包和高阶函数。

常见用途 包括日志记录、权限检查、缓存、性能测试等。

通过装饰器，Python 提供了一种优雅且强大的方式来增强代码的功能和可维护性。

1. ReAct是什么意思：

ReAct 是一种结合了 推理（Reasoning） 和 行动（Action） 的框架，通常用于构建智能代理（Agent），使其能够通过与环境的交互来完成任务。ReAct 的核心思想是让代理在每一步中 动态地决定下一步的操作，并通过 推理 来指导行动。

ReAct 的全称是 Reasoning + Acting，它强调代理不仅要有行动能力，还要具备推理能力，以便在复杂任务中做出更智能的决策。

ReAct 的核心思想

推理（Reasoning）：

代理通过分析当前状态、任务目标和历史信息，生成下一步的计划或策略。

推理可以是逻辑推理、基于规则的推理，或者是基于模型的推理（如大语言模型的推理能力）。

行动（Action）：

代理根据推理结果执行具体的操作，例如调用工具、查询外部数据、与环境交互等。

行动可以是调用 API、执行函数、生成文本等。

循环迭代：

ReAct 是一个循环过程：代理在每一步中推理并行动，然后根据行动的结果更新状态，继续推理和行动，直到任务完成。

ReAct 的工作流程通常包括以下几个步骤：

观察：代理获取当前的环境状态或任务输入。

推理：代理分析当前状态，生成下一步的计划或策略。

行动：代理执行具体的操作（如调用工具、查询数据等）。

观察结果：代理获取行动的结果，更新状态。

迭代：重复上述步骤，直到任务完成。

ReAct 与 LangChain 的结合

在 LangChain 中，ReAct 框架通常用于构建 智能代理（Agent）。LangChain 提供了以下组件来支持 ReAct：

工具（Tools）：

代理可以调用的外部工具，例如搜索引擎、计算器、数据库等。

提示词模板（Prompt Templates）：

用于指导代理进行推理和行动的提示词。

大语言模型（LLM）：

提供推理能力，生成下一步的计划或策略。

代理执行器（Agent Executor）：

负责管理代理的推理和行动循环。