

二、使用LangGraph构建Agent智能体

这一章节开始，我们将使用LangGraph来重新构建一个专属的聊天机器人。并以此为案例，逐步构建一个功能强大、又安全可控的聊天Agent智能体。再以此为基础，为后续LangGraph构建多智能体提供基础支撑。

- 使用LangGraph构建Agent智能体
- Agent智能体增加Tools工具调用机制
- Agent智能体消息记忆管理功能
- Human-In-Loop人类监督功能

说明：LangGraph是和LangChain一体的。我觉得，任何抛开LangChain独立讲解LangGraph的，都是耍流氓。所以，后续内容设计都是在LangChain的基础上构建，并且最好是能够跟上我之前分享LangChain的思路。

一、什么是Agent?

Agent智能体，是LangGraph中的一个核心概念。很多朋友也应该经常在网上听说过Agent智能体，但是，到底什么是智能体呢？

这个问题并没有标准答案。而LangGraph给我们勾勒出了一个好的Agent智能体的形象。这个Agent，可以类比于一个好的员工。什么是好的员工呢？自然是希望这个员工即能力强大，可以放心的独立完成任务，而不用过多干预实现的细节。同时，这个员工又要能够听从安排，关键节点要及时跟领导请示，并及时的根据领导的指示调整自己的工作进度。

具体到LangGraph的实现中，Agent即需要拥有封装与大模型交互的所有基础能力，包括访问大模型、调用Tools、保存ChatMemory等等这些基础的能力。可以独立的完成一系列基于大模型构建的任务。同时，我们又可以随时干预Agent执行进度，对关键步骤随时做出调整。

二、构建聊天Agent智能体

要如何构建这样强大并且听话的Agent智能体呢？我们先从基础的大模型聊天开始。

在LangChain中，提供了ChatModels的接口，用于访问大模型。我们可以通过调用ChatModels的接口来访问大模型。

```
from config.load_key import load_key
from langchain_community.chat_models import ChatTongyi
# 构建阿里云百炼大模型客户端
llm = ChatTongyi(
    model="qwen-plus",
    api_key=load_key("BAILIAN_API_KEY"),
)

llm.invoke("你是谁？能帮我解决什么问题？")
```

```
AIMessage(content='你好！我是通义千问，阿里巴巴集团旗下的超大规模语言模型。我能够帮助你解决各种各样的问题，包括但不限于：\n\n- **信息查询**：我可以为你提供各种领域的知识和信息，比如科学、技术、文化、历史等。
\n- **写作辅助**：无论是写故事、公文、邮件还是剧本，我都可以提供帮助。
\n- **逻辑推理**：如果你有复杂的逻辑问题或需要分析的情况，我可以尝试帮你理清思路。
\n- **编程支持**：我可以理解并生成多种编程语言的代码，帮助你解决编程中的问题。
\n- **多语言交流**：我支持多种语言，可以帮助你进行跨语言的沟通和翻译。
\n- **日常对话**：如果你想聊天或者分享想法，我也很乐意陪伴你。
\n\n如果你有任何具体的需求或问题，随时告诉我，我会尽力提供帮助！', additional_kwargs={}, response_metadata={'model_name': 'qwen-plus', 'finish_reason': 'stop', 'request_id': '11dbf031-46b3-9946-84bf-450fee0eff7c', 'token_usage': {'input_tokens': 17, 'output_tokens': 177, 'total_tokens': 194, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-805f3453-2235-4a00-9261-5cd2b10fb419-0')
```

而LangGraph只要将这个ChatModel进行简单的封装，就可以完成与大模型的交互。

```
from langgraph.prebuilt import create_react_agent

agent = create_react_agent(
    model=llm,
    tools=[],
    prompt="You are a helpful assistant",
)

agent.invoke({"messages": [{"role": "user", "content": "你是谁？能帮我解决什么问题？"}]})
```

```
{'messages': [HumanMessage(content='你是谁？能帮我解决什么问题？', additional_kwargs={}, response_metadata={}, id='f4159810-3c0c-49a5-81cf-cb1901555892'),
  AIMessage(content='你好！我是通义千问，阿里巴巴集团旗下的超大规模语言模型。我可以帮助你解决各种问题，包括但不限于：\n\n1. **回答问题**：无论是学术问题、生活常识、技术问题还是趣味知识，我都可以提供详细的解答。
\n2. **创作文字**：我可以帮你写故事、公文、邮件、剧本等各类文本。
\n3. **逻辑推理**：我可以进行复杂的逻辑推理和数学计算，帮助你解决难题。
\n4. **编程**：支持多种编程语言，可以帮助你编写代码、调试程序或解释编程概念。
\n5. **多语言支持**：除了中文，我还支持英文、德语、法语、西班牙语等多种语言，满足国际交流需求。
\n6. **表达观点**：可以就某一话题发表见解，参与讨论。
\n7. **玩游戏**：如猜谜语、成语接龙等互动游戏。
\n\n如果你有任何需要帮助的地方，随时告诉我！我会尽力为你提供支持和帮助。😊', additional_kwargs={}, response_metadata={'model_name': 'qwen-plus', 'finish_reason': 'stop', 'request_id': '0f94b967-ae36-9b15-b79d-1a2e456ace7c', 'token_usage': {'input_tokens': 27, 'output_tokens': 202, 'total_tokens': 229, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-aa7eeb8a-9062-4446-8da4-017021d66e86-0')])
```

当然，Agent也支持常用的Stream流式输出的方式。

```
for chunk in agent.stream(
    {"messages": [{"role": "user", "content": "你是谁？能帮我解决什么问题？"}]},
    # stream_mode="messages"
):
    print(chunk)
    print("\n")
```

```
{'agent': {'messages': [AIMessage(content='你好！我是通义千问，阿里巴巴集团旗下的超大规模语言模型。我能够帮助你解决各种问题，包括但不限于：\n\n1. **知识查询**：我可以提供广泛领域的信息，例如科学、技术、历史、文化等。
2. **问题解答**：无论是学术问题、工作中的难题，还是日常生活中的疑惑，我都可以尝试为你解答。
3. **创意支持**：如果你需要灵感，比如写故事、诗歌、邮件或策划活动，我可以提供帮助。
4. **语言翻译**：我支持多种语言的互译，帮助你跨越语言障碍。
5. **对话交流**：随时可以和我聊天解闷，分享想法或感受。
6. **技术支持**：对于编程和技术相关的问题，我也能提供一定的帮助和建议。
\n\n如果你有任何具体的需求或问题，请随时告诉我！我会尽力提供帮助。😊', additional_kwargs={}, response_metadata={'model_name': 'qwen-plus', 'finish_reason': 'stop', 'request_id': 'ad3f2900-91ef-9384-bf54-8e05495c5e53', 'token_usage': {'input_tokens': 27, 'output_tokens': 179, 'total_tokens': 206, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-ddd26b7a-d05c-469e-830b-3a662be094fc-0')]]}
```

这里stream_mode有三种选项：

- updates：流式输出每个工具调用的每个步骤。
- messages：流式输出大语言模型回复的Token。
- values：一次拿到所有的chunk。默认值。
- custom：自定义输出。主要是可以在工具内部使用get_stream_writer获取输入流，添加自定义的内容。

关于流式输出的这几种选项，在后面结合Graph，会体现出更大的作用。

三、增加Tools工具调用

Tools工具机制是大语言模型中的一个重要机制，它可以让大模型调用外部工具，从而实现更加复杂的功能。通常，一个完整的工具调用流程需要以下几个步骤：

1. 客户端定义工具类，实现工具的功能。
2. 客户端请求大语言模型，带上问题以及工具的描述信息。
3. 大语言模型综合判断问题，并决定是否调用工具。
4. 如果大语言模型判断需要使用工具，就会向客户端返回一个带有tool_calls工具调用信息的AiMessage。
5. 客户端根据工具调用信息，调用工具，并将结果返回给大语言模型。
6. 大语言模型根据工具调用的结果，生成最终的回答。

使用LangChain，我们需要实现完整的流程。而使用LangGraph后，工具成了Agent的标配。只需要定义工具类，Agent中会自行完成工具调用的流程。

```
import datetime
from langgraph.prebuilt import create_react_agent
```

```
def get_current_date():
    """获取今天日期"""
    return datetime.datetime.today().strftime("%Y-%m-%d")

agent = create_react_agent(
    model=llm,
    tools=[get_current_date],
    prompt="You are a helpful assistant",
)

agent.invoke({"messages":[{"role":"user","content":"今天是几月几号"}]})
```

```
{'messages': [HumanMessage(content='今天是几月几号', additional_kwargs={},
response_metadata={}, id='fe2f895d-e671-48fd-b923-8167dde338ca'),
  AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0, 'id':
'call_0490226b1df34522b263b3', 'type': 'function', 'function': {'name':
'get_current_date', 'arguments': '{}'}}]}, response_metadata={'model_name': 'qwen-plus',
'finish_reason': 'tool_calls', 'request_id': 'e783d687-a346-9067-bb96-51d478210fb6',
'token_usage': {'input_tokens': 155, 'output_tokens': 13, 'total_tokens': 168,
'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-53e18787-5de0-4721-a418-
9c82b25fc976-0', tool_calls=[{'name': 'get_current_date', 'args': {}, 'id':
'call_0490226b1df34522b263b3', 'type': 'tool_call'}]),
  ToolMessage(content='2025-06-10', name='get_current_date', id='cf29db0c-c366-4cf2-9bf0-
d63912f91366', tool_call_id='call_0490226b1df34522b263b3'),
  AIMessage(content='今天是2025年6月10日.', additional_kwargs={}, response_metadata=
{'model_name': 'qwen-plus', 'finish_reason': 'stop', 'request_id': '4195f991-d335-9a17-
9f06-d149f5d678b4', 'token_usage': {'input_tokens': 187, 'output_tokens': 15,
'total_tokens': 202, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-d09839ac-
d602-4ca8-9518-272c37b49b23-0')]]}
```

从返回的结果就能看到，LangGraph的Agent完整的封装了工具调用的整个流程。而我们所需要关注的，只是构建出带有工具信息的Agent，然后把用户的问题交给Agent去处理就行了。

在定义工具时，除了可以从工具函数的注释中获取工具描述信息外，LangGraph也同样兼容了LangChain中使用@tool注解声明工具的方式。

如果工具执行时出错了，LangGraph也提供了主动处理异常信息的能力。

```
from langchain_core.tools import tool
from langgraph.prebuilt import ToolNode
# 定义工具 return_direct=True 表示直接返回工具的结果
@tool("devide_tool",return_direct=True)
def devide(a : int,b : int) -> float:
    """计算两个整数的除法。
    Args:
```

```

        a (int): 除数
        b (int): 被除数"""
# 自定义错误
if b == 1:
    raise ValueError("除数不能为1")
return a/b

print(devide.name)
print(devide.description)
print(devide.args)

# 定义工具调用错误处理函数
def handle_tool_error(error: Exception) -> str:
    """处理工具调用错误。
    Args:
        error (Exception): 工具调用错误"""
    if isinstance(error, ValueError):
        return "除数为1没有意义, 请重新输入一个除数和被除数。"
    elif isinstance(error, ZeroDivisionError):
        return "除数不能为0, 请重新输入一个除数和被除数。"
    return f"工具调用错误: {error}"

tool_node = ToolNode(
    [devide],
    handle_tool_errors=handle_tool_error
)

agent_with_error_handler = create_react_agent(
    model=llm,
    tools=tool_node
)

result = agent_with_error_handler.invoke({"messages":[{"role":"user","content":"10除以5等于多少? "}]}))
# 打印最后的返回结果
# print(result["messages"][-1].content)
print(result)

```

devide_tool

计算两个整数的除法。

Args:

a (int): 除数

b (int): 被除数

```
{'a': {'title': 'A', 'type': 'integer'}, 'b': {'title': 'B', 'type': 'integer'}}
{'messages': [HumanMessage(content='10除以5等于多少?', additional_kwargs={},
response_metadata={}, id='3bafa315-16ca-4d06-86e3-d90cee86d88d'), AIMessage(content='',
additional_kwargs={'tool_calls': [{'index': 0, 'id': 'call_c3f3ed186bf54ad4b9dbd8',
'type': 'function', 'function': {'name': 'devide_tool', 'arguments': '{"a": 10, "b":
5}'}]}], response_metadata={'model_name': 'qwen-plus', 'finish_reason': 'tool_calls',
'request_id': '5e11f8e4-2cf9-9a59-a050-e8c150640bda', 'token_usage': {'input_tokens': 208,
'output_tokens': 25, 'total_tokens': 233, 'prompt_tokens_details': {'cached_tokens': 0}}},
id='run-e07a7f59-7046-4881-9e68-03730589262d-0', tool_calls=[{'name': 'devide_tool',
'args': {'a': 10, 'b': 5}, 'id': 'call_c3f3ed186bf54ad4b9dbd8', 'type': 'tool_call'}]),
ToolMessage(content='2.0', name='devide_tool', id='978472a5-41e5-4bfa-a549-e90ed20320fd',
tool_call_id='call_c3f3ed186bf54ad4b9dbd8') ]}
```

四、增加消息记忆

对大模型的交互信息进行保存，这是实现多轮对话的关键。

Qwen 3

长沙今天天气怎么样？给我一个简短的介绍



● 已完成搜索

今天是2025年5月27日，星期二，长沙的天气预报为多云转大雨，气温在19°C到29°C之间，风向为东南风，风力等级为2级。湿度方面，早晨的湿度较高，达到了100%，但随着一天中的时间推移，湿度会有所下降。由于今天有雨，出门记得携带雨具，并注意行车安全。紫外线指数较弱，不需要特别防护。空气质量状况良好，适合户外活动，但在降雨期间可能需要调整计划。请根据实际天气变化适时调整着装和活动安排。

参考来源 · 4 >



需要将与大模型之前的聊天记录全部传给大模型，
这样大模型才能理解这个问题。

北京呢？



● 已完成搜索

今天是2025年5月27日，北京的天气情况如下：

- 天气状况：晴
- 温度：最高温度达到32°C
- 风向：南风
- 风力：预计风力为2、3级

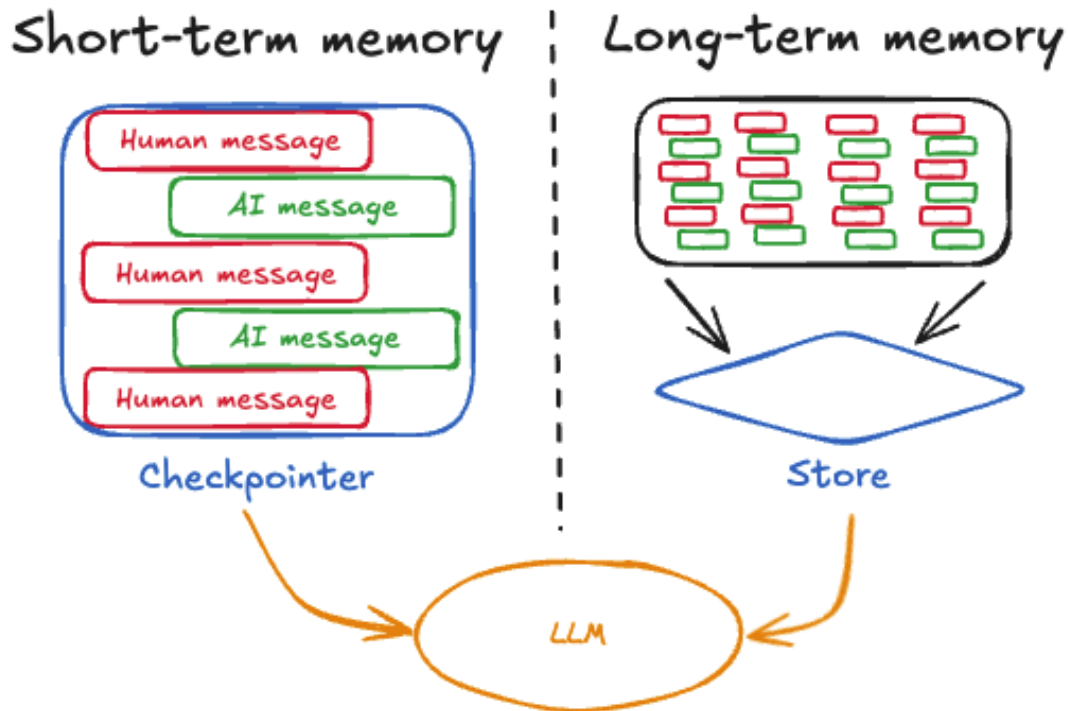


在LangChain中，我们需要自行定义ChatMessageHistory，并且自行保存每一轮的消息记录，然后在调用大模型时，将其作为参数传入。

LangGraph中，实现消息记录的流程，也完整的封装到了Agent当中。

LangGraph将消息记忆分为了短期记忆与长期记忆。

- 短期记忆是指Agent内部的记忆，用于当前对话中的历史记忆信息。LangGraph将他封装成CheckPoint。
- 长期记忆是指Agent外部的记忆，用第三方存储长久的保存用户级别或者应用级别的聊天信息。LangGraph将它封装成Store。



*Both **short-term** and **long-term** memory require persistent storage to maintain continuity across LLM interactions. In production environments, this data is typically stored in a database.*

对于记忆管理，我认为LangGraph的管理方式或许比具体实现更有参考价值。

在具体实现时，LangGraph都默认提供了InMemorySaver和InMemoryStore，也同样都可以转移到其他外部存储当中。不过，短期记忆通常是代表那些会话级别的小内存，而长期记忆通常是代表那些用户级别或者应用级别的大内存。

短期记忆的内存比较紧张，所以需要更频繁的清理内存，并对已有的消息记录进行总结，从而减少内存占用。而长期记忆的内存比较充足，所以不太需要频繁的清理内存。更需要关注的，是如何对已有的消息进行检索。

4.1 短期记忆 CheckPoint

在LangGraph的Agent中，只需要指定checkpointer属性，就可以实现短期记忆。具体传入的属性需要是BaseCheckpointSaver的子类。

LangGraph中默认提供了InMemorySaver，用于将短期记忆信息保存在内存中。当然，也可以采用Redis、SQLite等第三方存储来实现长期记忆。不过当前版本的LangGraph并没有提供具体的实现，需要自行实现。(如果不会写，交给AI)

另外，使用checkpointer时，需要制定一个单独的thread_id来区分不同的对话。


```

from langgraph.checkpoint.memory import InMemorySaver
from langgraph.prebuilt import create_react_agent

checkpointer = InMemorySaver()

def get_weather(city: str) -> str:
    """获取某个城市的天气"""
    return f"城市: {city}, 天气一直都是晴天! "

agent = create_react_agent(
    model=llm,
    tools=[get_weather],
    checkpointer=checkpointer
)

# Run the agent
config = {
    "configurable": {
        "thread_id": "1"
    }
}

cs_response = agent.invoke(
    {"messages": [{"role": "user", "content": "长沙天气怎么样? "}]},
    config
)

print(cs_response)

# Continue the conversation using the same thread_id
bj_response = agent.invoke(
    {"messages": [{"role": "user", "content": "北京呢? "}]},
    config
)

bj_response

```



```
{'messages': [HumanMessage(content='长沙天气怎么样? ', additional_kwargs={},
response_metadata={}, id='e517580b-f433-4a85-86d1-cc2dfc69ab7e'), AIMessage(content='',
additional_kwargs={'tool_calls': [{'index': 0, 'id': 'call_f1c8dadefbcb48df8d6996',
'type': 'function', 'function': {'name': 'get_weather', 'arguments': '{"city": "长
沙"}'}}]}, response_metadata={'model_name': 'qwen-plus', 'finish_reason': 'tool_calls',
'request_id': '806fe716-1dd2-9e99-bc0f-44eab4c64fa4', 'token_usage': {'input_tokens': 161,
'output_tokens': 17, 'total_tokens': 178, 'prompt_tokens_details': {'cached_tokens': 0}}},
id='run-4ff8ef85-3ab3-4ad7-937e-ebe2467be5af-0', tool_calls=[{'name': 'get_weather',
'args': {'city': '长沙'}, 'id': 'call_f1c8dadefbcb48df8d6996', 'type': 'tool_call'}]),
ToolMessage(content='城市: 长沙, 天气一直都是晴天! ', name='get_weather', id='88e8254a-6eee-
407d-bdac-cf73cadb9d7a', tool_call_id='call_f1c8dadefbcb48df8d6996'),
AIMessage(content='长沙的天气一直都是晴天! ', additional_kwargs={}, response_metadata=
{'model_name': 'qwen-plus', 'finish_reason': 'stop', 'request_id': '317e8854-3d29-9148-
b713-5884844abaea', 'token_usage': {'input_tokens': 198, 'output_tokens': 11,
'total_tokens': 209, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-67eaf402-
8792-4f11-af01-30aa93b8eb8a-0')]]}
```

```
{'messages': [HumanMessage(content='长沙天气怎么样? ', additional_kwargs={},
response_metadata={}, id='e517580b-f433-4a85-86d1-cc2dfc69ab7e'),
  AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0, 'id':
'call_f1c8dadefbcb48df8d6996', 'type': 'function', 'function': {'name': 'get_weather',
'arguments': '{"city": "长沙"}'}]}], response_metadata={'model_name': 'qwen-plus',
'finish_reason': 'tool_calls', 'request_id': '806fe716-1dd2-9e99-bc0f-44eab4c64fa4',
'token_usage': {'input_tokens': 161, 'output_tokens': 17, 'total_tokens': 178,
'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-4ff8ef85-3ab3-4ad7-937e-
ebe2467be5af-0', tool_calls=[{'name': 'get_weather', 'args': {'city': '长沙'}, 'id':
'call_f1c8dadefbcb48df8d6996', 'type': 'tool_call'}]),
  ToolMessage(content='城市: 长沙, 天气一直都是晴天! ', name='get_weather', id='88e8254a-6eee-
407d-bdac-cf73cadb9d7a', tool_call_id='call_f1c8dadefbcb48df8d6996'),
  AIMessage(content='长沙的天气一直都是晴天! ', additional_kwargs={}, response_metadata=
{'model_name': 'qwen-plus', 'finish_reason': 'stop', 'request_id': '317e8854-3d29-9148-
b713-5884844abaea', 'token_usage': {'input_tokens': 198, 'output_tokens': 11,
'total_tokens': 209, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-67eaf402-
8792-4f11-af01-30aa93b8eb8a-0'),
  HumanMessage(content='北京呢? ', additional_kwargs={}, response_metadata={},
id='e91c5d64-d24b-45d5-92b2-435479d998f2'),
  AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0, 'id':
'call_ab3f937745e645e7894761', 'type': 'function', 'function': {'name': 'get_weather',
'arguments': '{"city": "北京"}'}]}], response_metadata={'model_name': 'qwen-plus',
'finish_reason': 'tool_calls', 'request_id': '5c5918c1-2f81-9ebe-9d6c-47c43166b472',
'token_usage': {'input_tokens': 222, 'output_tokens': 17, 'total_tokens': 239,
'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-ccb45704-fbe0-4680-b5c9-
73703920539b-0', tool_calls=[{'name': 'get_weather', 'args': {'city': '北京'}, 'id':
'call_ab3f937745e645e7894761', 'type': 'tool_call'}]),
  ToolMessage(content='城市: 北京, 天气一直都是晴天! ', name='get_weather', id='3f35cc19-87c6-
45bc-9112-9cee38b5ecd4', tool_call_id='call_ab3f937745e645e7894761'),
  AIMessage(content='北京的天气也一直都是晴天! ', additional_kwargs={}, response_metadata=
{'model_name': 'qwen-plus', 'finish_reason': 'stop', 'request_id': '5f775221-5695-9668-
807c-7ef9adfb30e', 'token_usage': {'input_tokens': 259, 'output_tokens': 11,
'total_tokens': 270, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-f233a772-
7177-426e-8570-a66bbfb52564-0')]]
```

从结果可以看到，当前会话当中和大模型的每次交互记录，包括工具调用的信息，都保存在了短期记忆当中。当然，目前的实现是保存在内存中，所以程序结束后就释放了。生产环境中，LangGraph建议是保存到外部存储当中，例如数据库、文件系统等。这样每次启动程序时，都可以从外部存储中加载历史记录。

短期记忆通常认为还是比较紧张的，所以需要定期做清理，防止历史消息过多。

LangGraph的Agent中，提供了一个pre_model_hook属性，可以在每次调用大模型之前触发。通过这个hook，就可以来定期管理短期记忆。

LangGraph中管理短期记忆的方法主要有两种：

- Summarization 总结：用大模型的方式，对短期记忆进行总结，然后再把总结的结果作为新的短期记忆。
- Trimming 删除：直接把短期记忆中最旧的消息删除掉。

LangGraph提供了SummarizationNode函数，用于使用大模型的方式对短期记忆进行总结。

```

from langmem.short_term import SummarizationNode
from langchain_core.messages.utils import count_tokens_approximately
from langgraph.prebuilt import create_react_agent
from langgraph.prebuilt.chat_agent_executor import AgentState
from langgraph.checkpoint.memory import InMemorySaver
from typing import Any

# 使用大模型对历史信息进行总结
summarization_node = SummarizationNode(
    token_counter=count_tokens_approximately,
    model=llm,
    max_tokens=384,
    max_summary_tokens=128,
    output_messages_key="llm_input_messages",
)

class State(AgentState):
    # 注意：这个状态管理的作用是为了能够保存上一次总结的结果。这样就可以防止每次调用大模型时，都要重新总结历史信息。
    # 这是一个比较常见的优化方式，因为大模型的调用是比较耗时的。
    context: dict[str, Any]

checkpointer = InMemorySaver()

agent = create_react_agent(
    model=llm,
    tools=tools,
    pre_model_hook=summarization_node,
    state_schema=State,
    checkpointer=checkpointer,
)

```

另外，还提供了trim_messages函数，用于定期清理短期记忆。

```

from langchain_core.messages.utils import (
    trim_messages,
    count_tokens_approximately
)

from langgraph.prebuilt import create_react_agent

# This function will be called every time before the node that calls LLM
def pre_model_hook(state):
    trimmed_messages = trim_messages(
        state["messages"],
        strategy="last",
        token_counter=count_tokens_approximately,
        max_tokens=384,
        start_on="human",
        end_on=("human", "tool"),
    )

```

```

        return {"llm_input_messages": trimmed_messages}

checkpointer = InMemorySaver()
agent = create_react_agent(
    model=llm,
    tools=[],
    pre_model_hook=pre_model_hook,
    checkpointer=checkpointer,
)

```

实现了基础的短期记忆管理后，LangGraph还提供了状态管理机制，用于保存处理过程中的中间结果。而且，这些状态数据，还可以在Tools工具中使用。

```

from typing import Annotated
from langgraph.prebuilt import InjectedState, create_react_agent
from langgraph.prebuilt.chat_agent_executor import AgentState
from langchain_core.tools import tool

class CustomState(AgentState):
    user_id: str

@tool(return_direct=True)
def get_user_info(
    state: Annotated[CustomState, InjectedState]
) -> str:
    """查询用户信息."""
    user_id = state["user_id"]
    return "user_123用户的姓名: 楼兰。" if user_id == "user_123" else "未知用户"

agent = create_react_agent(
    model=llm,
    tools=[get_user_info],
    state_schema=CustomState,
)

agent.invoke({
    "messages": "查询用户信息",
    "user_id": "user_123"
})

```

```
{'messages': [HumanMessage(content='查询用户信息', additional_kwargs={}, response_metadata={}, id='90ae5eec-a789-4483-9beb-7fc9ff492073'),
  AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0, 'id': 'call_e92899f36e174aa08b13e6', 'type': 'function', 'function': {'name': 'get_user_info', 'arguments': '{}'}}]}, response_metadata={'model_name': 'qwen-plus', 'finish_reason': 'tool_calls', 'request_id': '6c077d3b-7148-9958-9293-93145d091da3', 'token_usage': {'input_tokens': 148, 'output_tokens': 13, 'total_tokens': 161, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-0bc2c569-d532-4b8d-ade2-473810955ace-0', tool_calls=[{'name': 'get_user_info', 'args': {}, 'id': 'call_e92899f36e174aa08b13e6', 'type': 'tool_call'}]),
  ToolMessage(content='user_123用户的姓名: 楼兰。', name='get_user_info', id='82d759ab-d828-445c-b551-2f9ff62b2a1b', tool_call_id='call_e92899f36e174aa08b13e6')],
  'user_id': 'user_123'}
```

4.2 长期记忆

长期记忆通常认为还是比较充足的记忆空间，因此使用时，可以比短期记忆更加粗犷，不太需要实时关注内存空间大小。

至于使用方式，和短期记忆差不太多。主要是通过Agent的store属性指定一个 实现类就可以了。与短期记忆最大的区别在于，短期记忆通过thread_id来区分不同的对话，而长期记忆则通过namespace来区分不同的命名空间。

```
from langchain_core.runnables import RunnableConfig
from langgraph.config import get_store
from langgraph.prebuilt import create_react_agent
from langgraph.store.memory import InMemoryStore
from langchain_core.tools import tool

# 定义长期存储
store = InMemoryStore()
# 添加一些测试数据。 users是命名空间，user_123是key，后面的JSON数据是value
store.put(
    ("users",),
    "user_123",
    {
        "name": "楼兰",
        "age": "33",
    }
)
#定义工具
@tool(return_direct=True)
def get_user_info(config: RunnableConfig) -> str:
    """查找用户信息"""
    # 获取长期存储。获取到了后，这个存储组件可读也可写
    store = get_store()
    # store.put(
```

```

#      ("users",),
#      "user_456",
#      {
#          "name": "楼兰",
#          "age": "33",
#      }
# )
# 获取配置中的用户ID
user_id = config["configurable"].get("user_id")
user_info = store.get(("users",), user_id)
return str(user_info.value) if user_info else "Unknown user"

agent = create_react_agent(
    model=llm,
    tools=[get_user_info],
    store=store
)

# Run the agent
agent.invoke(
    {"messages": [{"role": "user", "content": "查找用户信息"}]},
    config={"configurable": {"user_id": "user_123"}}
)

```

```

{'messages': [HumanMessage(content='查找用户信息', additional_kwargs={}, response_metadata={}, id='7eb4ab13-7772-4419-91a5-597600b67ee4'),
  AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0, 'id': 'call_938d3874f073475abb949c', 'type': 'function', 'function': {'name': 'get_user_info', 'arguments': '{}'}}]}, response_metadata={'model_name': 'qwen-plus', 'finish_reason': 'tool_calls', 'request_id': '8ed0f272-5377-9a31-b41e-3332d2ede94e', 'token_usage': {'input_tokens': 148, 'output_tokens': 13, 'total_tokens': 161, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-f4df912a-7d1d-4430-8739-7322a63d9c84-0', tool_calls=[{'name': 'get_user_info', 'args': {}, 'id': 'call_938d3874f073475abb949c', 'type': 'tool_call'}]),
  ToolMessage(content="{ 'name': '楼兰', 'age': '33' }", name='get_user_info', id='b675906d-0851-4a84-a594-36d33f96de7f', tool_call_id='call_938d3874f073475abb949c'))]

```

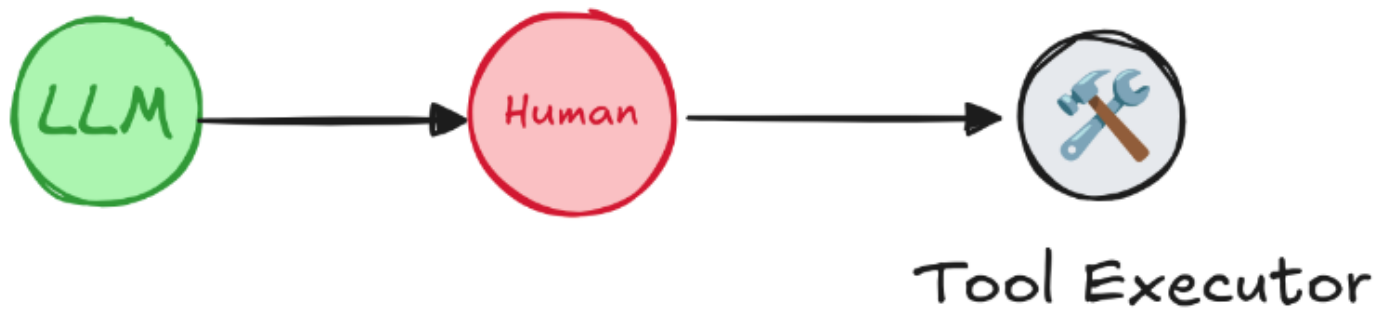
五、Human-in-the-loop人类监督

这也是LangGraph的Agent中非常核心的一个功能。

在Agent的工作过程中，有一个问题是非常致命的。就是Agent可以添加Tools工具，但是要不要调用工具，却完全是由Agent自己决定的。

这就会导致Agent在面对一些问题时，可能会出现错误的判断。

为了解决这个问题，LangGraph提供了Human-in-the-loop的功能。在Agent进行工具调用的过程中，允许用户进行监督。这就需要中断当前的执行任务，等待用户输入后，再重新恢复任务。



A human can review and edit the output from the agent before proceeding. This is particularly critical in applications where the tool calls requested may be sensitive or require human oversight.

在实现时，LangGraph提供了interrupt()方法添加人类监督。监督时需要中断当前任务，所以通常是和stream流式方法配合使用。

```
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.types import interrupt
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import tool

# An example of a sensitive tool that requires human review / approval
@tool(return_direct=True)
def book_hotel(hotel_name: str):
    """预定宾馆"""
    response = interrupt(
        f"正准备执行'book_hotel'工具预定宾馆，相关参数名:  {{'hotel_name': {hotel_name}}}. "
        "请选择OK，表示同意，或者选择edit，提出补充意见."
    )
    if response["type"] == "OK":
        pass
    elif response["type"] == "edit":
        hotel_name = response["args"]["hotel_name"]
    else:
        raise ValueError(f"Unknown response type: {response['type']}")
    return f"成功在 {hotel_name} 预定了一个房间."

checkpointer = InMemorySaver()

agent = create_react_agent(
    model=llm,
    tools=[book_hotel],
    checkpointer=checkpointer,
)

config = {
    "configurable": {
        "thread_id": "1"
    }
}
```



```
for chunk in agent.stream(
    {"messages": [{"role": "user", "content": "帮我在图灵宾馆预定一个房间"}]},
    config
):
    print(chunk)
    print("\n")
```

```
{'agent': {'messages': [AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0, 'id': 'call_c9978e4809f447d286914b', 'type': 'function', 'function': {'name': 'book_hotel', 'arguments': '{"hotel_name": "图灵宾馆"}'}]}], response_metadata={'model_name': 'qwen-plus', 'finish_reason': 'tool_calls', 'request_id': 'e2578a15-e306-9d94-b23b-f8b17ac33c9b', 'token_usage': {'input_tokens': 171, 'output_tokens': 21, 'total_tokens': 192, 'prompt_tokens_details': {'cached_tokens': 0}}}, id='run-82d4fb3c-73c0-453b-a64a-e5878d3e47cc-0', tool_calls=[{'name': 'book_hotel', 'args': {'hotel_name': '图灵宾馆'}, 'id': 'call_c9978e4809f447d286914b', 'type': 'tool_call'}])}]}
```

```
{'__interrupt__': (Interrupt(value="正准备执行'book_hotel'工具预定宾馆，相关参数名：{'hotel_name': 图灵宾馆}。请选择OK，表示同意，或者选择edit，提出补充意见。", resumable=True, ns=['tools:3e143aac-9c42-9f34-162c-c0e42c08165b']),)}
```

执行完成后，会在book_hotel执行过程中，输出一个Interrupt响应，表示当前正在等待用户输入确认。

接下来，可以通过Agent提交一个Command请求，来继续完成之前的任务。

需要注意的是，在这个示例中，Agent只会一直等待用户输入。如果等待时间过长，后续请求就无法恢复了。

```
from langgraph.types import Command

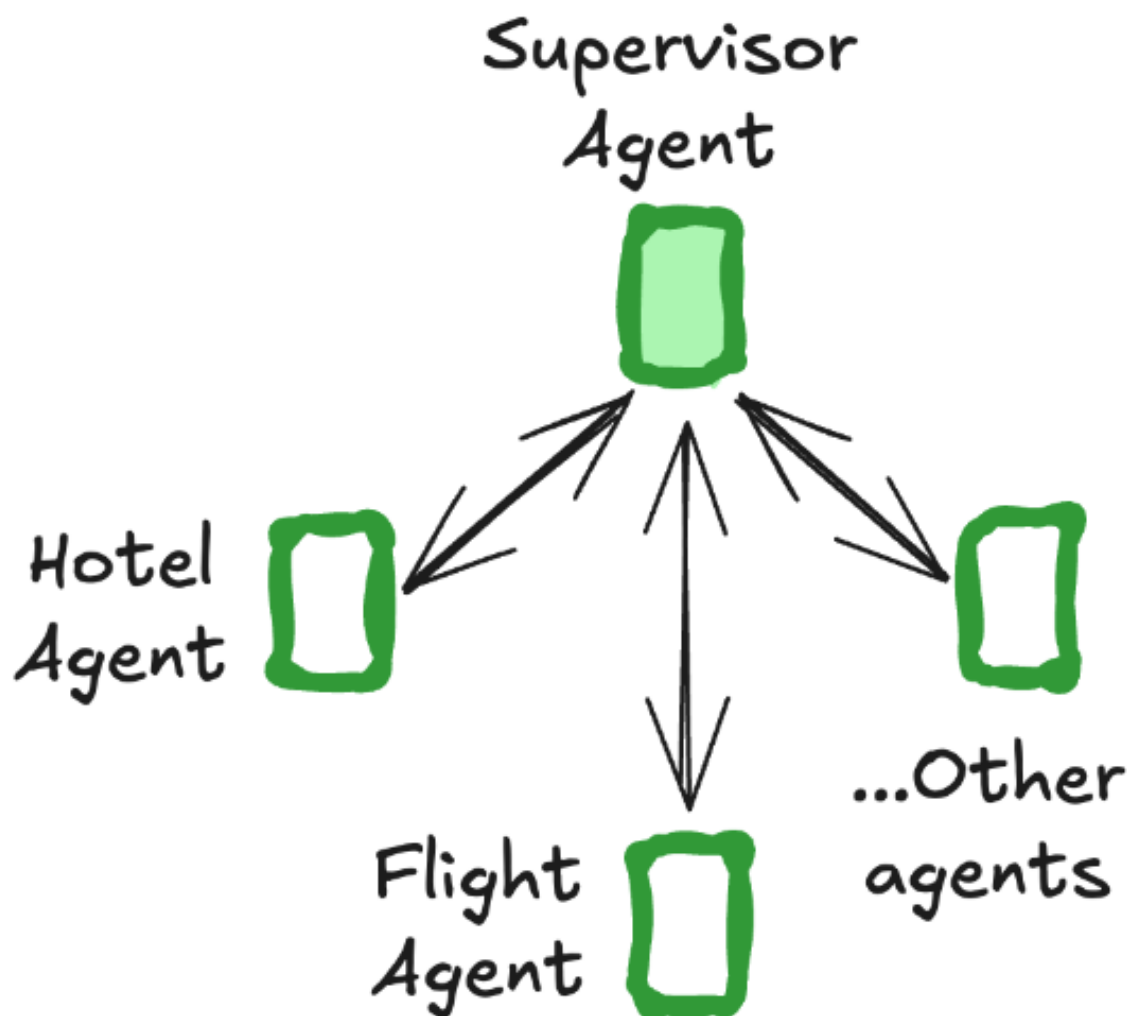
for chunk in agent.stream(
    # Command(resume={"type": "OK"}),
    Command(resume={"type": "edit", "args": {"hotel_name": "三号宾馆"}}),
    config
):
    print(chunk)
    print(chunk['tools'][0]['messages'][-1].content)
    print("\n")
```

```
{'tools': {'messages': [ToolMessage(content='成功在 三号宾馆 预定了一个房间.', name='book_hotel', id='97532b7c-f383-46ce-813e-c27891214c4a', tool_call_id='call_c9978e4809f447d286914b')]}},
成功在 三号宾馆 预定了一个房间。
```

五、多Agent应用构建

应该是当前版本不兼容。案例执行报错。

通常我们希望一个Agent能够专注干好一件事情。但是，如果是面对一些复杂的任务，我们可能需要多个Agent协同完成。例如一种典型的多Agent系统会是这样的：



由一个Supervisor Agent对任务进行分发。然后，交由另一个Agent来处理具体的事情。这样，这多个Agent就可以成为一个同时处理多个任务的系统。

LangGraph中单独提供一个langgraph-supervisor依赖库来实现这种类型的多Agent系统。

当然，LangGraph的核心是用Graph图的方式来实现多Agent协作。所以，这个库更多是作为基础了解。

```
# 安装langgraph-supervisor依赖库
!pip install langgraph-supervisor --upgrade
```

```
from langgraph.prebuilt import create_react_agent
from langgraph_supervisor import create_supervisor

def book_hotel(hotel_name: str):
    """Book a hotel"""
    return f"Successfully booked a stay at {hotel_name}."
```

```

def book_flight(from_airport: str, to_airport: str):
    """Book a flight"""
    return f"Successfully booked a flight from {from_airport} to {to_airport}."

flight_assistant = create_react_agent(
    model=llm,
    tools=[book_flight],
    prompt="You are a flight booking assistant",
    name="flight_assistant"
)

hotel_assistant = create_react_agent(
    model=llm,
    tools=[book_hotel],
    prompt="You are a hotel booking assistant",
    name="hotel_assistant"
)

supervisor = create_supervisor(
    agents=[flight_assistant, hotel_assistant],
    model=llm,
    prompt=(
        "You manage a hotel booking assistant and a"
        "flight booking assistant. Assign work to them."
    )
).compile()

for chunk in supervisor.stream(
    {
        "messages": [
            {
                "role": "user",
                "content": "book a flight from BOS to JFK and a stay at McKittrick Hotel"
            }
        ]
    }
):
    print(chunk)
    print("\n")

```

六、LangGraph的Agent总结

Agent，这是LangGraph后续构建Graph图的基础。但其实Agent并不是LangGraph框架当中独有的。甚至Agent并不是一种技术，而是我们设想的一种理想的大模型工作模式。那么到底什么是Agent？或者说AI行业心目中理想的Agent应该是什么样子呢？LangGraph实际上给我们提供了一种理解。而这个理解，或许比具体实现更重要。

这一章节，我们重点在演练LangGraph中Agent的功能。我们介绍了LangGraph的Agent功能，以及如何使用LangGraph的Agent功能来构建一个简单的应用。对于LangGraph框架来说，由于有了LangChain作为支撑，Agent智能体或许并不是他的重点。后续，LangGraph使用Graph图的方式协调管理多个Agent，或许是更大的价值所在。

- 使用LangGraph构建Agent智能体
- Agent智能体增加Tools工具调用机制
- Agent智能体消息记忆管理功能
- Human-In-Loop人类监督功能

但是，LangGraph对于Agent的功能封装却给大模型应用落地提供了非常好的思想指导。通过Agent，我们不再需要关注应用的实现细节，而是可以更专注于应用的功能设计。而Agent，绝不仅仅只是LangGraph所需要构建的，即便脱离LangGraph框架，如何构建一个能力强大又听话懂事的Agent，或许是我们后续都需要思考的问题。