Introduction to LangGraph
LangGraph简介

In this tutorial, we will build a support chatbot in LangGraph that can:

在本教程中，我们将在 LangGraph 中构建一个支持聊天机器人，它可以：

- Answer common questions by searching the web
  通过搜索网络回答常见问题
- Maintain conversation state across calls
  保持通话期间的对话状态
- Route complex queries to a human for review
  将复杂的查询转交给人工审核
- Use custom state to control its behavior
  使用自定义状态来控制其行为
- Rewind and explore alternative conversation paths
  倒带并探索替代对话路径

We'll start with a basic chatbot and progressively add more sophisticated capabilities, introducing key LangGraph concepts along the way.

我们将从基本的聊天机器人开始，逐步添加更复杂的功能，并在此过程中引入关键的 LangGraph 概念。

Setup 设置

First, install the required packages:

首先，安装所需的软件包：

In [ ]: 在 [ ]:
```
%%capture --no-stderr
%pip install -U langgraph langsmith

# Used for this tutorial; not a requirement for LangGraph
%pip install -U langchain_anthropic
```

Next, set your API keys:

接下来，设置您的 API 密钥：

In [1]: 在[1]中:
```
import getpass
import os


def _set_env(var: str):
    if not os.environ.get(var):
        os.environ[var] = getpass.getpass(f"{var}: ")


_set_env("ANTHROPIC_API_KEY")
```

(Encouraged) LangSmith makes it a lot easier to see what's going on "under the hood."

（受到鼓励）LangSmith 让我们更容易了解“幕后”发生的事情。

In [2]: 在[2]中:
```
_set_env("LANGSMITH_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = "LangGraph Tutorial"
```

Part 1: Build a Basic Chatbot
第 1 部分：构建基本聊天机器人

We'll first create a simple chatbot using LangGraph. This chatbot will respond directly to user messages. Though simple, it will illustrate the core concepts of building with LangGraph. By the end of this section, you will have a built rudimentary chatbot.

我们首先使用 LangGraph 创建一个简单的聊天机器人。该聊天机器人将直接响应用户消息。虽然简单，但它将说明使用 LangGraph 构建的核心概念。到本节结束时，您将拥有一个构建的基本聊天机器人。

Start by creating a `StateGraph`. A `StateGraph` object defines the structure of our chatbot as a "state machine". We'll add `nodes` to represent the llm and functions our chatbot can call and `edges` to specify how the bot should transition between these functions.

首先创建一个 `StateGraph` 。 `StateGraph` 对象将我们的聊天机器人的结构定义为“状态机”。我们将添加 `nodes` 来表示 llm 以及我们的聊天机器人可以调用的函数，并添加 `edges` 来指定机器人应如何在这些函数之间转换。

In [3]: 在[3]中:
```
from typing import Annotated

from typing_extensions import TypedDict

from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages


class State(TypedDict):
    # Messages have the type "list". The `add_messages` function
    # in the annotation defines how this state key should be updated
    # (in this case, it appends messages to the list, rather than overwr
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)
```

Notice that we've defined our `State` as a TypedDict with a single key: `messages`. The `messages` key is annotated with the `add_messages` function, which tells LangGraph to append new messages to the existing list, rather than overwriting it.

请注意，我们已将 `State` 定义为具有单个键的 TypedDict： `messages` 。 `messages` 键用 `add_messages` 函数注释，该函数告诉 LangGraph 将新消息追加到现有列表中，而不是覆盖它。

So now our graph knows two things:

现在我们的图知道两件事：

1. Every `node` we define will receive the current `State` as input and return a value that updates that state.

   我们定义的每个 `node` 都会接收当前的 `State` 作为输入，并返回一个更新该状态的值。

2. `messages` will be *appended* to the current list, rather than directly overwritten. This is communicated via the prebuilt `add_messages` function in the `Annotated` syntax.

   `messages` 将被追加到当前列表中，而不是直接覆盖。这是通过 `Annotated` 语法中预构建的 `add_messages` 函数进行通信的。

Next, add a "`chatbot`" node. Nodes represent units of work. They are typically regular python functions.

接下来，添加一个“ `chatbot` ”节点。节点代表工作单元。它们通常是常规的 Python 函数。

In [4]: 在[4]中:

```python
from langchain_anthropic import ChatAnthropic

llm = ChatAnthropic(model="claude-3-haiku-20240307")


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 whene
# the node is used.
graph_builder.add_node("chatbot", chatbot)
```

Notice how the `chatbot` node function takes the current `State` as input and returns an updated `messages` list. This is the basic pattern for all LangGraph node functions.

请注意 `chatbot` 节点函数如何将当前 `State` 作为输入并返回更新的 `messages` 列表。这是所有 LangGraph 节点函数的基本模式。

The `add_messages` function in our `State` will append the llm's response messages to whatever messages are already in the state.

`State` 中的 `add_messages` 函数会将 llm 的响应消息附加到状态中已有的任何消息中。

Next, add an `entry` point. This tells our graph **where to start its work** each time we run it.

接下来，添加一个 `entry` 点。这告诉我们的图表每次运行时从哪里开始工作。

In [5]: 在[5]中:

```python
graph_builder.set_entry_point("chatbot")
```

Similarly, set a `finish` point. This instructs the graph **"any time this node is run, you can exit."**

同样，设置一个 `finish` 点。这指示图表“任何时候运行此节点，您都可以退出”。

In [6]: 在[6]中:

```python
graph_builder.set_finish_point("chatbot")
```

Finally, we'll want to be able to run our graph. To do so, call "`compile()`" on the graph builder. This creates a "`CompiledGraph`" we can use invoke on our state.

最后，我们希望能够运行我们的图表。为此，请在图形生成器上调用“ `compile()` ”。这将创建一个“ `CompiledGraph` ”，我们可以在状态上使用调用。

In [7]: 在[7]中:

```python
graph = graph_builder.compile()
```

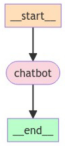
You can visualize the graph using the `get_graph` method and one of the "draw" methods, like `draw_ascii` or `draw_png`. The `draw` methods each require additional dependencies.

您可以使用 `get_graph` 方法和“绘制”方法之一（例如 `draw_ascii` 或 `draw_png` ）可视化图形。每个 `draw` 方法都需要额外的依赖项。

In [8]: 在[8]中:

```python
from IPython.display import Image, display

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
except:
    # This requires some extra dependencies and is optional
    pass
```



Now let's run the chatbot!

现在让我们运行聊天机器人！

Tip: You can exit the chat loop at any time by typing "quit", "exit", or "q".

提示：您可以随时输入“quit”、“exit”或“q”退出聊天循环。

In [9]: 在[9]中:

```python
while True:
    user_input = input("User: ")
    if user_input.lower() in ["quit", "exit", "q"]:
        print("Goodbye!")
        break
    for event in graph.stream({"messages": ("user", user_input)}):
        for value in event.values():
            print("Assistant:", value["messages"][-1].content)
```

```
User:  what's langgraph all about?
```

```
Assistant: Langgraph is a new open-source deep learning framework that focuses o
n enabling efficient training and deployment of large language models. Some key
things to know about Langgraph:

1. Efficient Training: Langgraph is designed to accelerate the training of large
language models by leveraging advanced optimization techniques and parallelizati
on strategies.

2. Modular Architecture: Langgraph has a modular architecture that allows for ea
sy customization and extension of language models, making it flexible for a vari
ety of NLP tasks.

3. Hardware Acceleration: The framework is optimized for both CPU and GPU hardwa
re, allowing for efficient model deployment on a wide range of devices.

4. Scalability: Langgraph is designed to handle large-scale language models with
billions of parameters, enabling the development of state-of-the-art NLP applica
tions.

5. Open-Source: Langgraph is an open-source project, allowing developers and res
earchers to collaborate, contribute, and build upon the framework.

6. Performance: The goal of Langgraph is to provide superior performance and eff
iciency compared to existing deep learning frameworks, particularly for training
and deploying large language models.

Overall, Langgraph is a promising new deep learning framework that aims to addre
ss the challenges of building and deploying advanced natural language processing
models at scale. It is an active area of research and development, with the pote
ntial to drive further advancements in the field of language AI.
User:  hm that doesn't seem right...
Assistant: I'm sorry, I don't have enough context to determine what doesn't seem
right. Could you please provide more details about what you're referring to? Tha
t would help me better understand and respond appropriately.
User:  q
Goodbye!
```

Congratulations! You've built your first chatbot using LangGraph. This bot can engage in basic conversation by taking user input and generating responses using an LLM. You can inspect a LangSmith Trace for the call above at the provided link.

恭喜！您已经使用 LangGraph 构建了第一个聊天机器人。该机器人可以通过获取用户输入并使用 LLM 生成响应来进行基本对话。您可以在提供的链接中检查上面调用的 LangSmith 跟踪。

However, you may have noticed that the bot's knowledge is limited to what's in its training data. In the next part, we'll add a web search tool to expand the bot's knowledge and make it more capable.

但是，您可能已经注意到，机器人的知识仅限于其训练数据中的内容。在下一部分中，我们将添加一个网络搜索工具来扩展机器人的知识并使其更加强大。

Below is the full code for this section for your reference:

以下是本节的完整代码，供您参考：

In [10]: 在[10]中:

```python
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from typing_extensions import TypedDict

from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


llm = ChatAnthropic(model="claude-3-haiku-20240307")


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 whe
# the node is used.
graph_builder.add_node("chatbot", chatbot)
graph_builder.set_entry_point("chatbot")
graph_builder.set_finish_point("chatbot")
graph = graph_builder.compile()
```

Part 2: Enhancing the Chatbot with Tools
第 2 部分：使用工具增强聊天机器人

To handle queries our chatbot can't answer "from memory", we'll integrate a web search tool. Our bot can use this tool to find relevant information and provide better responses.

为了处理我们的聊天机器人无法“凭记忆”回答的查询，我们将集成一个网络搜索工具。我们的机器人可以使用此工具查找相关信息并提供更好的响应。

Requirements 要求

Before we start, make sure you have the necessary packages installed and API keys set up:

在开始之前，请确保您已安装必要的软件包并设置 API 密钥：

First, install the requirements to use the Tavily Search Engine, and set your TAVILY_API_KEY.

首先，安装使用 Tavilly 搜索引擎的要求，并设置您的 TAVILY_API_KEY。

In [ ]: 在 [ ]:

```
%%capture --no-stderr
%pip install -U tavily-python
```

In [3]: 在[3]中:

```
_set_env("TAVILY_API_KEY")
```

Next, define the tool:

接下来，定义工具：

In [4]: 在[4]中:

```
from langchain_community.tools.tavily_search import TavilySearchResults

tool = TavilySearchResults(max_results=2)
tools = [tool]
tool.invoke("What's a 'node' in LangGraph?")
```

Out[4]: 输出[4]:

```
[{'url': 'https://medium.com/@cplog/introduction-to-langgraph-a-beginne
rs-guide-14f9be027141',
  'content': 'Nodes: Nodes are the building blocks of your LangGraph. E
ach node represents a function or a computation step. You define nodes
to perform specific tasks, such as processing input, making ...'},
 {'url': 'https://js.langchain.com/docs/langgraph',
  'content': "Assuming you have done the above Quick Start, you can bui
ld off it like:\nHere, we manually define the first tool call that we w
ill make.\nNotice that it does that same thing as agent would have done
(adds the agentOutcome key).\n LangGraph\n🦜🕸LangGraph.js\n Building
language agents as graphs \nOverview\u200b\nLangGraph is a library for
```

```
building stateful, multi-actor applications with LLMs, built on top of
(and intended to be used with) LangChain.js.\n Therefore, we will use a
n object with one key (messages) with the value as an object: { value:
Function, default?: () => any }\nThe default key must be a factory that
returns the default value for that attribute.\n Streaming Node Output\u
200b\nOne of the benefits of using LangGraph is that it is easy to stre
am output as it's produced by each node.\n What this means is that only
one of the downstream edges will be taken, and which one that is depend
s on the results of the start node.\n")]
```

The results are page summaries our chat bot can use to answer questions.

结果是我们的聊天机器人可以用来回答问题的页面摘要。

Next, we'll start defining our graph. The following is all **the same as in Part 1**, except we have added `bind_tools` on our LLM. This lets the LLM know the correct JSON format to use if it wants to use our search engine.

接下来，我们将开始定义我们的图表。以下内容与第 1 部分中的内容相同，只是我们在 LLM 上添加了 `bind_tools` 。这可以让 LLM 知道要使用我们的搜索引擎时要使用的正确 JSON 格式。

In [11]: 在[11]中:

```python
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from typing_extensions import TypedDict

from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


llm = ChatAnthropic(model="claude-3-haiku-20240307")
# Modification: tell the LLM which tools it can call
llm_with_tools = llm.bind_tools(tools)


def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}


graph_builder.add_node("chatbot", chatbot)
```

Next we need to create a function to actually run the tools if they are called. We'll do this by adding the tools to a new node.

接下来，我们需要创建一个函数，以便在调用工具时实际运行这些工具。我们将通过将工具添加到新节点来完成此操作。

Below, implement a `BasicToolNode` that checks the most recent message in the state and calls tools if the message contains `tool_calls. It relies on the LLM's` tool_calling` support, which is available in Anthropic, OpenAI, Google Gemini, and a number of other LLM providers.

下面，实现一个 `BasicToolNode` ，它检查状态中的最新消息，如果消息包含 `tool_calls. It relies on the LLM's` tool_calling` 支持，则调用工具，该支持在 Anthropic、OpenAI、Google Gemini 和许多其他LLM提供商。

We will later replace this with LangGraph's prebuilt ToolNode to speed things up, but building it ourselves first is instructive.

In [12]:

```python
import json

from langchain_core.messages import ToolMessage


class BasicToolNode:
    """A node that runs the tools requested in the last AIMessage."""

    def __init__(self, tools: list) -> None:
        self.tools_by_name = {tool.name: tool for tool in tools}

    def __call__(self, inputs: dict):
        if messages := inputs.get("messages", []):
            message = messages[-1]
        else:
            raise ValueError("No message found in input")
        outputs = []
        for tool_call in message.tool_calls:
            tool_result = self.tools_by_name[tool_call["name"]].invoke(
                tool_call["args"]
            )
            outputs.append(
                ToolMessage(
                    content=json.dumps(tool_result),
                    name=tool_call["name"],
                    tool_call_id=tool_call["id"],
                )
            )
        return {"messages": outputs}


tool_node = BasicToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)
```

With the tool node added, we can define the `conditional_edges`.

With the tool node added, we can define the `conditional_edges`.

添加工具节点后，我们可以定义 `conditional_edges` 。

Recall that **edges** route the control flow from one node to the next. **Conditional edges** usually contain "if" statements to route to different nodes depending on the current graph state. These functions receive the current graph `state` and return a string or list of strings indicating which node(s) to call next.

回想一下，边将控制流从一个节点路由到下一个节点。条件边通常包含“if”语句，根据当前图状态路由到不同的节点。这些函数接收当前图 `state` 并返回一个字符串或字符串列表，指示接下来要调用的节点。

Below, call define a router function called `route_tools`, that checks for tool_calls in the chatbot's output. Provide this function to the graph by calling `add_conditional_edges`, which tells the graph that whenever the `chatbot` node completes to check this function to see where to go next.

下面，调用定义一个名为 `route_tools` 的路由器函数，该函数检查聊天机器人输出中的 tool_calls。通过调用 `add_conditional_edges` 向图形提供此函数，这告诉图形每当 `chatbot` 节点完成时就检查此函数以了解下一步该去哪里。

The condition will route to `tools` if tool calls are present and " `__end__` " if not.

Later, we will replace this with the prebuilt tools_condition to be more concise, but implementing it ourselves first makes things more clear.

In [13]:

```python
from typing import Literal


def route_tools(
```

```
    state: State,
) -> Literal["tools", "__end__"]:
    """Use in the conditional_edge to route to the ToolNode if the last message

    has tool calls. Otherwise, route to the end."""
    if isinstance(state, list):
        ai_message = state[-1]
    elif messages := state.get("messages", []):
        ai_message = messages[-1]
    else:
        raise ValueError(f"No messages found in input state to tool_edge: {stat
    if hasattr(ai_message, "tool_calls") and len(ai_message.tool_calls) > 0:
        return "tools"
    return "__end__"


# The `tools_condition` function returns "tools" if the chatbot asks to use a too
# it is fine directly responding. This conditional routing defines the main agent
graph_builder.add_conditional_edges(
    "chatbot",
    route_tools,
    # The following dictionary lets you tell the graph to interpret the condition
```

```
# The `tools_condition` function returns "tools" if the chatbot asks to
# it is fine directly responding. This conditional routing defines the
graph_builder.add_conditional_edges(
    "chatbot",
    route_tools,
    # The following dictionary lets you tell the graph to interpret th
    # It defaults to the identity function, but if you
    # want to use a node named something else apart from "tools",
    # You can update the value of the dictionary to something else
    # e.g., "tools": "my_tools"
    {"tools": "tools", "__end__": "__end__"},
)
# Any time a tool is called, we return to the chatbot to decide the ne
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")
graph = graph_builder.compile()
```

Notice that conditional edges start from a single node. This tells the graph "any time the '`chatbot`' node runs, either go to 'tools' if it calls a tool, or end the loop if it responds directly.

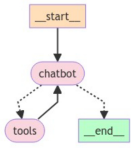
请注意，条件边从单个节点开始。这告诉图表“任何时候‘`chatbot`’节点运行，要么转到‘工具’（如果它调用工具），要么结束循环（如果它直接响应）。

Like the prebuilt `tools_condition`, our function returns the "`__end__`" string if no tool calls are made. When the graph transitions to `__end__`, it has no more tasks to complete and ceases execution. Because the condition can return `__end__`, we don't need to explicitly set a `finish_point` this time. Our graph already has a way to finish!

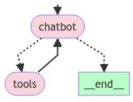
Let's visualize the graph we've built. The following function has some additional dependencies to run that are unimportant for this tutorial.

In [14]:
```
from IPython.display import Image, display

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
except:
    # This requires some extra dependencies and is optional
    pass
```



Now we can ask the bot questions outside its training data.



Now we can ask the bot questions outside its training data.

现在我们可以向机器人询问其训练数据之外的问题。

In [15]: 在[15]中:
```
from langchain_core.messages import BaseMessage

while True:
    user_input = input("User: ")
    if user_input.lower() in ["quit", "exit", "q"]:
        print("Goodbye!")
        break
    for event in graph.stream({"messages": [("user", user_input)]}):
        for value in event.values():
            if isinstance(value["messages"][-1], BaseMessage):
                print("Assistant:", value["messages"][-1].content)
```

```
User:  what's langgraph all about?
Assistant: [{'id': 'toolu_01L1TABSBXsHPsebWiMPNqf1', 'input': {'query': 'langgra
ph'}, 'name': 'tavily_search_results_json', 'type': 'tool_use'}]
Assistant: [{"url": "https://langchain-ai.github.io/langgraph/", "content": "Lan
gGraph is framework agnostic (each node is a regular python function). It extend
s the core Runnable API (shared interface for streaming, async, and batch calls)
to make it easy to: Seamless state management across multiple turns of conversat
ion or tool usage. The ability to flexibly route between nodes based on dynamic
criteria."}, {"url": "https://blog.langchain.dev/langgraph-multi-agent-workflow
s/", "content": "As a part of the launch, we highlighted two simple runtimes: on
e that is the equivalent of the AgentExecutor in langchain, and a second that wa
s a version of that aimed at message passing and chat models.\n It's important t
o note that these three examples are only a few of the possible examples we coul
d highlight - there are almost assuredly other examples out there and we look fo
rward to seeing what the community comes up with!\n LangGraph: Multi-Agent Workf
lows\nLinks\nLast week we highlighted LangGraph - a new package (available in bo
th Python and JS) to better enable creation of LLM workflows containing cycles,
which are a critical component of most agent runtimes. \"\nAnother key differenc
e between Autogen and LangGraph is that LangGraph is fully integrated into the L
angChain ecosystem, meaning you take fully advantage of all the LangChain integr
ations and LangSmith observability.\n As part of this launch, we're also excited
to highlight a few applications built on top of LangGraph that utilize the conce
pt of multiple agents.\n"}]
Assistant: Based on the search results, LangGraph is a framework-agnostic Python
and JavaScript library that extends the core Runnable API from the LangChain pro
ject to enable the creation of more complex workflows involving multiple agents
or components. Some key things about LangGraph:

- It makes it easier to manage state across multiple turns of conversation or to
ol usage, and to dynamically route between different nodes/components based on c
riteria.

- It is integrated with the LangChain ecosystem, allowing you to take advantage
of LangChain integrations and observability features.

- It enables the creation of multi-agent workflows, where different components o
r agents can be chained together in more flexible and complex ways than the stan
dard LangChain AgentExecutor.

- The core idea is to provide a more powerful and flexible framework for buildin
g LLM-powered applications and workflows, beyond what is possible with just the
core LangChain tools.

Overall, LangGraph seems to be a useful addition to the LangChain toolkit, focus
ed on enabling more advanced, multi-agent style applications and workflows power
ed by large language models.
User:  neat!
```

```
Assistant: I'm afraid I don't have enough context to provide a substantive respo
nse to "neat!". As an AI assistant, I'm designed to have conversations and provi
de information to users, but I need more details or a specific question from you
in order to give a helpful reply. Could you please rephrase your request or prov
ide some additional context? I'd be happy to assist further once I understand wh
at you're looking for.
User:  what?
Assistant: I'm afraid I don't have enough context to provide a meaningful respon
se to "what?". Could you please rephrase your request or provide more details ab
out what you are asking? I'd be happy to try to assist you further once I have a
clearer understanding of your query.
User:  q
Goodbye!
```

Congrats! You've created a conversational agent in langgraph that can use a search engine to retrieve updated information when needed. Now it can handle a wider range of user queries. To inspect all the steps your agent just took, check out this LangSmith trace.

恭喜！您已经在 langgraph 中创建了一个对话代理，它可以在需要时使用搜索引擎检索更新的信息。现在它可以处理更广泛的用户查询。要检查代理刚刚执行的所有步骤，请查看此 LangSmith 跟踪。

Our chatbot still can't remember past interactions on its own, limiting its ability to have coherent, multi-turn conversations. In the next part, we'll add **memory** to address this.

The full code for the graph we've created in this section is reproduced below, replacing our `BasicToolNode` for the prebuilt ToolNode, and our `route_tools` condition with the prebuilt tools_condition

In [17]:
```python
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)
```

```python
class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
llm_with_tools = llm.bind_tools(tools)


def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}


graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
# Any time a tool is called, we return to the chatbot to decide the ne
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")
graph = graph_builder.compile()
```

Part 3: Adding Memory to the Chatbot
第 3 部分：为聊天机器人添加内存

Our chatbot can now use tools to answer user questions, but it doesn't remember the context of previous interactions. This limits its ability to have coherent, multi-turn conversations.

LangGraph solves this problem through **persistent checkpointing**. If you provide a `checkpointer` when compiling the graph and a `thread_id` when calling your graph, LangGraph automatically saves the state after each step. When you invoke the graph again using the same `thread_id`, the graph loads its saved state, allowing the chatbot to pick up where it left off.

We will see later that **checkpointing** is *much* more powerful than simple chat memory - it lets you save and resume complex state at any time for error recovery, human-in-the-loop workflows, time travel interactions, and more. But before we get too ahead of ourselves, let's add checkpointing to enable multi-turn conversations.

状态，从而允许聊天机器人从中断处继续。

We will see later that **checkpointing** is *much* more powerful than simple chat memory - it lets you save and resume complex state at any time for error recovery, human-in-the-loop workflows, time travel interactions, and more. But before we get too ahead of ourselves, let's add checkpointing to enable multi-turn conversations.

稍后我们将看到检查点比简单的聊天内存强大得多 - 它可以让您随时保存和恢复复杂的状态，以进行错误恢复、人机交互工作流程、时间旅行交互等。但在我们太过超前之前，让我们添加检查点以启用多轮对话。

To get started, create a `SqliteSaver` checkpointer.

首先，创建一个 `SqliteSaver` 检查指针。

In [1]: 在[1]中:
```python
from langgraph.checkpoint.sqlite import SqliteSaver

memory = SqliteSaver.from_conn_string(":memory:")
```

Notice that we've specified `:memory` as the Sqlite DB path. This is convenient for our tutorial (it saves it all in-memory). In a production application, you would likely change this to connect to your own DB and/or use one of the other checkpointer classes.

请注意，我们已指定 `:memory` 作为 Sqlite DB 路径。这对于我们的教程来说很方便（它将所有内容保存在内存中）。在生产应用程序中，您可能会更改此设置以连接到

您自己的数据库和/或使用其他检查指针类之一。

Next define the graph. Now that you've already built your own `BasicToolNode`, we'll replace it with LangGraph's prebuilt `ToolNode` and `tools_condition`, since these do some nice things like parallel API execution. Apart from that, the following is all copied from Part 2.

In [2]:
```
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition


class State(TypedDict):
    messages: Annotated[list, add_messages]
```

```
class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
llm_with_tools = llm.bind_tools(tools)


def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}


graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
# Any time a tool is called, we return to the chatbot to decide the next
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")
```

```
/Users/wfh/code/lc/langchain/libs/core/langchain_core/_api/beta_decorator.py:87: LangChainBetaWarning: The method `ChatAnthropic.bind_tools` is in beta. It is actively being worked on, so the API may change.
  warn_beta(
```

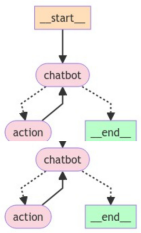
Finally, compile the graph with the provided checkpointer.

In [3]:
```
graph = graph_builder.compile(checkpointer=memory)
```

Notice the connectivity of the graph hasn't changed since Part 2. All we are doing is checkpointing the `State` as the graph works through each node.

In [6]:
```
from IPython.display import Image, display

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
except:
    # This requires some extra dependencies and is optional
    pass
```



Now you can interact with your bot! First, pick a thread to use as the key for this conversation.

现在您可以与您的机器人互动了！首先，选择一个线程作为本次对话的关键。

In [5]:
```
config = {"configurable": {"thread_id": "1"}}
```

Next, call your chat bot.

In [6]:
```
user_input = "Hi there! My name is Will."

# The config is the **second positional argument** to stream() or invoke()!
events = graph.stream(
    {"messages": [("user", user_input)]}, config, stream_mode="values"
)
for event in events:
    event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================

Hi there! My name is Will.
================================== Ai Message ==================================

It's nice to meet you, Will! I'm an AI assistant created by Anthropic. I'm here to help you with any questions or tasks you may have. Please let me know how I can assist you today.
```

Note: The config was provided as the **second positional argument** when calling our graph. It importantly is *not* nested within the graph inputs (`{'messages': []}`).

Let's ask a followup: see if it remembers your name.

In [8]:
```
user_input = "Remember my name?"

# The config is the **second positional argument** to stream() or invoke()!
events = graph.stream(
    {"messages": [("user", user_input)]}, config, stream_mode="values"
)
for event in events:
    event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================

Remember my name?
================================== Ai Message ==================================

Of course, your name is Will. It's nice to meet you again!
```

Notice that we are't the memory using an external list: it's all handled by the

```
================================ Human Message =================================
```

```
Remember my name?
================================== Ai Message ==================================

Of course, your name is Will. It's nice to meet you again!
```

Notice that we are't the memory using an external list: it's all handled by the checkpointer! You can inspect the full execution in this LangSmith trace to see what's going on.

请注意，我们不是使用外部列表的内存：它全部由检查指针处理！您可以检查 LangSmith 跟踪中的完整执行情况，看看发生了什么。

Don't believe me? Try this using a different config.

不相信我？使用不同的配置尝试此操作。

In [9]:
```python
# The only difference is we change the `thread_id` here to "2" instead of "1"
events = graph.stream(
    {"messages": [("user", user_input)]},
    {"configurable": {"thread_id": "2"}},
    stream_mode="values",
)
for event in events:
    event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================

Remember my name?
================================== Ai Message ==================================

I'm afraid I don't actually have the capability to remember your name. As an AI
assistant, I don't have a persistent memory of our previous conversations or int
eractions. I respond based on the current context provided to me. Could you plea
se restate your name or provide more information so I can try to assist you?
```

Notice that the **only** change we've made is to modify the `thread_id` in the config. See this call's LangSmith trace for comparison.

By now, we have made a few checkpoints across two different threads. But what goes into a checkpoint? To inspect a graph's `state` for a given config at any time, call `get_state(config)`.

In [10]:
```python
snapshot = graph.get_state(config)
snapshot
```

Out[10]:
```
StateSnapshot(values={'messages': [HumanMessage(content='Hi there! My name is Will.', id='aad97d7f-8845-4f9e-b723-2af3b7c97590'), AIMessage(content="It's nice to meet you, Will! I'm an AI assistant created by Anthropic. I'm here to help you with any questions or tasks you may have. Please let me know how I can assist you today.", response_metadata={'id': 'msg_01VCz7Y5jVmMZXibBtnECyvJ', 'model': 'claude-3-haiku-20240307', 'stop_reason': 'end_turn', 'stop_sequence': None, 'usage': {'input_tokens': 375, 'output_tokens': 49}}, id='run-66cf1695-5ba8-4fd8-a79d-ded9ee3c3b33-0'), HumanMessage(content='Remember my name?', id='ac1e9971-dbee-4622-9e63-5015dee05c20'), AIMessage(content="Of course, your name is Will. It's nice to meet you again!", response_metadata={'id': 'msg_01RsJ6GaQth7r9soxbF7TSpQ', 'model': 'claude-3-haiku-20240307', 'stop_reason': 'end_turn', 'stop_sequence': None, 'usage': {'input_tokens': 375, 'output_tokens': 49}}, id='run-66cf1695-5ba8-4fd8-a79d-ded9ee3c3b33-0'), HumanMessage(content='Remember my name?', id='ac1e9971-dbee-4622-9e63-5015dee05c20'), AIMessage(content="Of course, your name is Will. It's nice to meet you again!", response_metadata={'id': 'msg_01RsJ6GaQth7r9soxbF7TSpQ', 'model': 'claude-3-haiku-20240307', 'stop_reason': 'end_turn', 'stop_sequence': None, 'usage': {'input_tokens': 431, 'output_tokens': 19}}, id='run-890149d3-214f-44e8-9717-57ec4ef68224-0')]}, next=(), config={'configurable': {'thread_id': '1', 'thread_ts': '2024-05-06T22:23:20.430350+00:00'}}, parent_config=None)
```

In [11]: 在[11]中:
```python
snapshot.next  # (since the graph ended this turn, `next` is empty.
```

Out[11]:
```
()
```

The snapshot above contains the current state values, corresponding config, and the `next` node to process. In our case, the graph has reached an `__end__` state, so `next` is empty.

Congratulations! Your chatbot can now maintain conversation state across sessions thanks to LangGraph's checkpointing system. This opens up exciting possibilities for more natural, contextual interactions. LangGraph's checkpointing even handles **arbitrary complex graph states**, which is much more expressive and powerful than simple chat memory.

In the next part, we'll introduce human oversight to our bot to handle situations where it may need guidance or verification before proceeding.

Check out the code snippet below to review our graph from this section.

In [12]:
```python
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
llm_with_tools = llm.bind_tools(tools)
```

```python
from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
llm_with_tools = llm.bind_tools(tools)


def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}


graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
```

```
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")
graph = graph_builder.compile(checkpointer=memory)
```

Part 4: Human-in-the-loop

第 4 部分：人机交互

Agents can be unreliable and may need human input to successfully accomplish tasks. Similarly, for some actions, you may want to require human approval before running to ensure that everything is running as intended.

LangGraph supports `human-in-the-loop` workflows in a number of ways. In this section, we will use LangGraph's `interrupt_before` functionality to always break the tool node.

First, start from our existing code. The following is copied from Part 3.

In [1]:
```
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict
```

First, start from our existing code. The following is copied from Part 3.

首先，从我们现有的代码开始。以下内容是从第 3 部分复制的。

In [1]: 在[1]中:
```
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition

memory = SqliteSaver.from_conn_string(":memory:")


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
llm_with_tools = llm.bind_tools(tools)


def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}


graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")
```

```
/Users/wfh/code/lc/langchain/libs/core/langchain_core/_api/beta_decorator.py:87: LangChainBetaWarning: The method `ChatAnthropic.bind_tools` is in beta. It is actively being worked on, so the API may change.
  warn_beta(
```

Now, compile the graph, specifying to `interrupt_before` the `action` node.

现在，编译图形，指定 `interrupt_before` `action` 节点。

In [2]:
```
graph = graph_builder.compile(
    checkpointer=memory,
    # This is new!
    interrupt_before=["tools"],
    # Note: can also interrupt __after__ actions, if desired.
    # interrupt_after=["tools"]
    )
```

In [3]: 在[3]中:
```
user_input = "I'm learning LangGraph. Could you do some research on it for me?"
config = {"configurable": {"thread_id": "1"}}
# The config is the **second positional argument** to stream() or invoke()
events = graph.stream(
    {"messages": [("user", user_input)]}, config, stream_mode="values"
)
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================

I'm learning LangGraph. Could you do some research on it for me?
================================== Ai Message ==================================

[{'text': "Okay, let's do some research on LangGraph:", 'type': 'text'}, {'id': 'toolu_01Be7aRgMEv9cg6ezaFjiCry', 'input': {'query': 'LangGraph'}, 'name': 'tavily_search_results_json', 'type': 'tool_use'}]
Tool Calls:
  tavily_search_results_json (toolu_01Be7aRgMEv9cg6ezaFjiCry)
 Call ID: toolu_01Be7aRgMEv9cg6ezaFjiCry
  Args:
    query: LangGraph
```

Let's inspect the graph state to confirm it worked.

In [4]:
```
snapshot = graph.get_state(config)
snapshot.next
```

Out[4]:
```
('action',)
```

Notice that unlike last time, the "next" node is set to **'action'**. We've interrupted here! Let's check the tool invocation.

In [5]:
```
existing_message = snapshot.values["messages"][-1]
existing_message.tool_calls
```

Out[5]:
```
[{'name': 'tavily_search_results_json',
  'args': {'query': 'LangGraph'},
  'id': 'toolu_01Be7aRgMEv9cg6ezaFjiCry'}]
```

This query seems reasonable. Nothing to filter here. The simplest thing the human can do is just let the graph continue executing. Let's do that below.

Next, continue the graph! Passing in `None` will just let the graph continue where it left off, without adding anything new to the state.

In [6]:
```
# `None` will append nothing new to the current state, letting it resume as if it had never been interrupted
events = graph.stream(None, config, stream_mode="values")
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================= Tool Message =================================
Name: tavily_search_results_json
```

```
[{"url": "https://github.com/langchain-ai/langgraph", "content": "LangGraph is a
Python package that extends LangChain Expression Language with the ability to co
```

接下来继续上图！传入 `None` 只会让图表从中断处继续，而不向状态添加任何新内容。

```
In [6]: 在[6]中:
```

```
# `None` will append nothing new to the current state, letting it resume
events = graph.stream(None, config, stream_mode="values")
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================= Tool Message =================================
Name: tavily_search_results_json

[{"url": "https://github.com/langchain-ai/langgraph", "content": "LangGraph is a
Python package that extends LangChain Expression Language with the ability to co
ordinate multiple chains across multiple steps of computation in a cyclic manne
r. It is inspired by Pregel and Apache Beam and can be used for agent-like behav
iors, such as chatbots, with LLMs."}, {"url": "https://python.langchain.com/doc
s/langgraph/", "content": "LangGraph is a library for building stateful, multi-a
ctor applications with LLMs, built on top of (and intended to be used with) Lang
Chain . It extends the LangChain Expression Language with the ability to coordin
ate multiple chains (or actors) across multiple steps of computation in a cyclic
manner. It is inspired by Pregel and Apache Beam ."}]
================================== Ai Message ==================================

Based on the search results, LangGraph seems to be a Python library that extends
the LangChain library to enable more complex, multi-step interactions with large
language models (LLMs). Some key points:

- LangGraph allows coordinating multiple "chains" (or actors) over multiple step
s of computation, in a cyclic manner. This enables more advanced agent-like beha
viors like chatbots.
- It is inspired by distributed graph processing frameworks like Pregel and Apac
he Beam.
- LangGraph is built on top of the LangChain library, which provides a framework
for building applications with LLMs.

So in summary, LangGraph appears to be a powerful tool for building more sophist
icated applications and agents using large language models, by allowing you to c
oordinate multiple steps and actors in a flexible, graph-like manner. It extends
the capabilities of the base LangChain library.

Let me know if you need any clarification or have additional questions!
```

Review this call's LangSmith trace to see the exact work that was done in the above call. Notice that the state is loaded in the first step so that your chatbot can continue where it left off.

查看此调用的 LangSmith 跟踪以查看上述调用中完成的确切工作。请注意，状态是在第一步中加载的，以便您的聊天机器人可以从中断处继续。

Congrats! You've used an `interrupt` to add human-in-the-loop execution to your chatbot, allowing for human oversight and intervention when needed. This opens up the potential UIs you can create with your AI systems. Since we have already added a **checkpointer**, the graph can be paused **indefinitely** and resumed at any time as if nothing had happened.

Next, we'll explore how to further customize the bot's behavior using custom state updates.

恭喜！您已使用 `interrupt` 为聊天机器人添加人机交互执行，允许在需要时进行人工监督和干预。这开启了您可以使用 AI 系统创建的潜在 UI。由于我们已经添加了检查点，因此图表可以无限期暂停并随时恢复，就像什么也没发生一样。

Next, we'll explore how to further customize the bot's behavior using custom state updates.

接下来，我们将探索如何使用自定义状态更新进一步自定义机器人的行为。

Below is a copy of the code you used in this section. The only difference between this and the previous parts is the addition of the `interrupt_before` argument.

以下是您在本节中使用的代码的副本。这部分与前面部分之间的唯一区别是添加了 `interrupt_before` 参数。

```
In [7]: 在[7]中:
```

```
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import MessageGraph, StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
llm_with_tools = llm.bind_tools(tools)


def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}


graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")

memory = SqliteSaver.from_conn_string(":memory:")
graph = graph_builder.compile(
    checkpointer=memory,
    # This is new!
    interrupt_before=["tools"],
    # Note: can also interrupt __after__ actions, if desired.
    # interrupt_after=["tools"]
)
```

Part 5: Manually Updating the State

第 5 部分：手动更新状态

In the previous section, we showed how to interrupt a graph so that a human could inspect its actions. This lets the human `read` the state, but if they want to change they agent's course, they'll need to have `write` access.

Thankfully, LangGraph lets you **manually update state**! Updating the state lets you control the agent's trajectory by modifying its actions (even modifying the past!). This capability is particularly useful when you want to correct the agent's mistakes, explore alternative paths, or guide the agent towards a specific goal.

We'll show how to update a checkpointed state below. As before, first, define your graph. We'll reuse the exact same graph as before.

In [2]:
```
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import MessageGraph, StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
llm_with_tools = llm.bind_tools(tools)


def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}


graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")
memory = SqliteSaver.from_conn_string(":memory:")
graph = graph_builder.compile(
    checkpointer=memory,
    # This is new!
    interrupt_before=["tools"],
    # Note: can also interrupt **after** actions, if desired.
    # interrupt_after=["tools"]
)

user_input = "I'm learning LangGraph. Could you do some research on it
config = {"configurable": {"thread_id": "1"}}
# The config is the **second positional argument** to stream() or invoke(
events = graph.stream({"messages": [("user", user_input)]}, config)
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
/Users/wfh/code/lc/langchain/libs/core/langchain_core/_api/beta_decorator.py:87:
LangChainBetaWarning: The method `ChatAnthropic.bind_tools` is in beta. It is ac
tively being worked on, so the API may change.
  warn_beta(
```

In [3]:
```
snapshot = graph.get_state(config)
existing_message = snapshot.values["messages"][-1]
existing_message.pretty_print()
```

```
================================== Ai Message ==================================

[{'id': 'toolu_01DTyDpJ1kKdNps5yxv3AGJd', 'input': {'query': 'LangGraph'}, 'nam
e': 'tavily_search_results_json', 'type': 'tool_use'}]
Tool Calls:
  tavily_search_results_json (toolu_01DTyDpJ1kKdNps5yxv3AGJd)
 Call ID: toolu_01DTyDpJ1kKdNps5yxv3AGJd
  Args:
    query: LangGraph
```

So far, all of this is an *exact repeat* of the previous section. The LLM just requested to use the search engine tool and our graph was interrupted. If we proceed as before, the tool will be called to search the web.

But what if the user wants to intercede? What if we think the chat bot doesn't need to use the tool?

Let's directly provide the correct response!

In [4]:
```
from langchain_core.messages import AIMessage, ToolMessage

answer = (
    "LangGraph is a library for building stateful, multi-actor applications wit
)
new_messages = [
    # The LLM API expects some ToolMessage to match its tool call. We'll satisfy
```

让我们直接给出正确的答案吧！

In [4]:
```
from langchain_core.messages import AIMessage, ToolMessage

answer = (
    "LangGraph is a library for building stateful, multi-actor applicat
)
new_messages = [
    # The LLM API expects some ToolMessage to match its tool call. We'll
    ToolMessage(content=answer, tool_call_id=existing_message.tool_call
    # And then directly "put words in the LLM's mouth" by populating its
    AIMessage(content=answer),
]

new_messages[-1].pretty_print()
graph.update_state(
    # Which state to update
    config,
    # The updated values to provide. The messages in our `State` are "ap
    # to the existing state. We will review how to update existing messa
    {"messages": new_messages},
)

print("\n\nLast 2 messages;")
print(graph.get_state(config).values["messages"][-2:])
```

```
================================== Ai Message ==================================

LangGraph is a library for building stateful, multi-actor applications with LLM
s.


Last 2 messages;
[ToolMessage(content='LangGraph is a library for building stateful, multi-actor
applications with LLMs.', id='14589ef1-15db-4a75-82a6-d57c40a216d0', tool_call_i
d='toolu_01DTyDpJ1kKdNps5yxv3AGJd'), AIMessage(content='LangGraph is a library f
or building stateful, multi-actor applications with LLMs.', id='1c657bfb-7690-44
c7-a26d-d0d22453013d')]
```

Now the graph is complete, since we've provided the final response message! Since state updates simulate a graph step, they even generate corresponding traces. Inspec the LangSmith trace of the `update_state` call above to see what's going on.

Notice that our new messages is *appended* to the messages already in the state. Remember how we defined the `State` type?

```
class State(TypedDict):
    messages: Annotated[list, add_messages]
```

We annotated `messages` with the pre-built `add_messages` function. This instructs the graph to always append values to the existing list, rather than overwriting the list directly. The same logic is applied here, so the messages we passed to `update_state` were appended in the same way!

The `update_state` function operates as if it were one of the nodes in your

We annotated `messages` with the pre-built `add_messages` function. This instructs the graph to always append values to the existing list, rather than overwriting the list directly. The same logic is applied here, so the messages we passed to `update_state` were appended in the same way!

我们用预先构建的 `add_messages` 函数注释了 `messages` 。这指示图形始终将值附加到现有列表，而不是直接覆盖列表。这里应用了相同的逻辑，因此我们传递给 `update_state` 的消息以相同的方式附加！

The `update_state` function operates as if it were one of the nodes in your graph! By default, the update operation uses the node that was last executed, but you can manually specify it below. Let's add an update and tell the graph to treat it as if it came from the "chatbot".

`update_state` 函数的运行方式就好像它是图中的节点之一一样！默认情况下，更新操作使用最后执行的节点，但您可以在下面手动指定。让我们添加一个更新并告诉图表将其视为来自“聊天机器人”。

In [5]: 在[5]中:
```
graph.update_state(
    config,
    {"messages": [AIMessage(content="I'm an AI expert!")]},
    # Which node for this function to act as. It will automatically cont
    # processing as if this node just ran.
    as_node="chatbot",
)
```

Out[5]:
```
{'configurable': {'thread_id': '1',
  'thread_ts': '2024-05-06T22:27:57.350721+00:00'}}
```

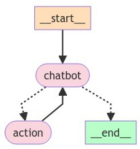
Check out the LangSmith trace for this update call at the provided link. **Notice** from the trace that the graph continues into the `tools_condition` edge. We just told the graph to treat the update `as_node="chatbot"` . If we follow the diagram below and start from the `chatbot` node, we naturally end up in the `tools_condition` edge and then `__end__` since our updated message lacks tool calls.

In [6]:
```
from IPython.display import Image, display

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
except:
    # This requires some extra dependencies and is optional
    pass
```

In [6]: 在[6]中:
```
from IPython.display import Image, display

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
except:
    # This requires some extra dependencies and is optional
    pass
```



Inspect the current state as before to confirm the checkpoint reflects our manual updates.

In [7]:
```
snapshot = graph.get_state(config)
print(snapshot.values["messages"][-3:])
print(snapshot.next)
```

```
[ToolMessage(content='LangGraph is a library for building stateful, multi-actor applications with LLMs.', id='14589ef1-15db-4a75-82a6-d57c40a216d0', tool_call_id='toolu_01DTyDpJ1kKdNps5yxv3AGJd'), AIMessage(content='LangGraph is a library for building stateful, multi-actor applications with LLMs.', id='1c657bfb-7690-44c7-a26d-d0d22453013d'), AIMessage(content="I'm an AI expert!", id='acd668e3-ba31-42c0-843c-00d0994d5885')]
()
```

Notice: that we've continued to add AI messages to the state. Since we are acting as the `chatbot` and responding with an AIMessage that doesn't contain `tool_calls` , the graph knows that it has entered a finished state (`next` is empty).

What if you want to overwrite existing messages?

The `add_messages` function we used to annotate our graph's `State` above controls how updates are made to the `messages` key. This function looks at any message IDs in the new `messages` list. If the ID matches a message in the existing state, `add_messages` overwrites the existing message with the new content.

As an example, let's update the tool invocation to make sure we get good results from our search engine! First, start a new thread:

The `add_messages` function we used to annotate our graph's `State` above controls how updates are made to the `messages` key. This function looks at any message IDs in the new `messages` list. If the ID matches a message in the existing state, `add_messages` overwrites the existing message with the new content.

我们用来注释上面图表的 `State` 的 `add_messages` 函数控制如何对 `messages` 键进行更新。此函数查看新的 `messages` 列表中的所有消息 ID。如果 ID 与现有状态的消息匹配， `add_messages` 就会用新内容覆盖现有消息。

As an example, let's update the tool invocation to make sure we get good results from our search engine! First, start a new thread:

作为示例，让我们更新工具调用，以确保我们从搜索引擎获得良好的结果！首先，启动一个新线程：

In [8]: 在[8]中:

```
user_input = "I'm learning LangGraph. Could you do some research on it
config = {"configurable": {"thread_id": "2"}}  # we'll use thread_id = 2
events = graph.stream(
    {"messages": [("user", user_input)]}, config, stream_mode="values"
)
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================

I'm learning LangGraph. Could you do some research on it for me?
================================== Ai Message ==================================

[{'id': 'toolu_013MvjoDHnv476ZGzyPFZhrR', 'input': {'query': 'LangGraph'}, 'nam
e': 'tavily_search_results_json', 'type': 'tool_use'}]
Tool Calls:
  tavily_search_results_json (toolu_013MvjoDHnv476ZGzyPFZhrR)
 Call ID: toolu_013MvjoDHnv476ZGzyPFZhrR
  Args:
    query: LangGraph
```

Next, let's update the tool invocation for our agent. Maybe we want to search for human-in-the-loop workflows in particular.

接下来，让我们更新代理的工具调用。也许我们特别想寻找人机交互的工作流程。

In [9]:

```
from langchain_core.messages import AIMessage

snapshot = graph.get_state(config)
existing_message = snapshot.values["messages"][-1]
print("Original")
print("Message ID", existing_message.id)
print(existing_message.tool_calls[0])
new_tool_call = existing_message.tool_calls[0].copy()
new_tool_call["args"]["query"] = "LangGraph human-in-the-loop workflow"
new_message = AIMessage(
    content=existing_message.content,
    tool_calls=[new_tool_call],
    # Important! The ID is how LangGraph knows to REPLACE the message in the sta
        id=existing_message.id,
    )

    print("Updated")
    print(new_message.tool_calls[0])
    print("Message ID", new_message.id)
    graph.update_state(config, {"messages": [new_message]})

    print("\n\nTool calls")
    graph.get_state(config).values["messages"][-1].tool_calls
```

```
Original
Message ID run-59283969-1076-45fe-bee8-ebfccab163c3-0
{'name': 'tavily_search_results_json', 'args': {'query': 'LangGraph'}, 'id': 'to
olu_013MvjoDHnv476ZGzyPFZhrR'}
Updated
{'name': 'tavily_search_results_json', 'args': {'query': 'LangGraph human-in-the
-loop workflow'}, 'id': 'toolu_013MvjoDHnv476ZGzyPFZhrR'}
Message ID run-59283969-1076-45fe-bee8-ebfccab163c3-0


Tool calls
```

Out[9]:

```
[{'name': 'tavily_search_results_json',
  'args': {'query': 'LangGraph human-in-the-loop workflow'},
  'id': 'toolu_013MvjoDHnv476ZGzyPFZhrR'}]
```

Notice that we've modified the AI's tool invocation to search for "LangGraph human-in-the-loop workflow" instead of the simple "LangGraph".

Check out the LangSmith trace to see the state update call - you can see our new message has successfully updated the previous AI message.

Resume the graph by streaming with an input of `None` and the existing config.

In [10]:

```
events = graph.stream(None, config, stream_mode="values")
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================= Tool Message =================================
Name: tavily_search_results_json

[{"url": "https://langchain-ai.github.io/langgraph/how-tos/human-in-the-loop/",
"content": "Human-in-the-loop\u00b6 When creating LangGraph agents, it is often
nice to add a human in the loop component. This can be helpful when giving them
access to tools. ... from langgraph.graph import MessageGraph, END # Define a ne
w graph workflow = MessageGraph # Define the two nodes we will cycle between wor
kflow. add_node (\"agent\", call_model) ..."}, {"url": "https://langchain-ai.git
hub.io/langgraph/how-tos/chat_agent_executor_with_function_calling/human-in-the-
loop/", "content": "Human-in-the-loop. In this example we will build a ReAct Age
nt that has a human in the loop. We will use the human to approve specific actio
ns. This examples builds off the base chat executor. It is highly recommended yo
u learn about that executor before going through this notebook. You can find doc
umentation for that example here."}]
================================== Ai Message ==================================

Based on the search results, LangGraph appears to be a framework for building AI
agents that can interact with humans in a conversational way. The key points I g
athered are:

- LangGraph allows for "human-in-the-loop" workflows, where a human can be invol
ved in approving or reviewing actions taken by the AI agent.
- This can be useful for giving the AI agent access to various tools and capabil
ities, with the human able to provide oversight and guidance.
- The framework includes components like "MessageGraph" for defining the convers
ational flow between the agent and human.

Overall, LangGraph seems to be a way to create conversational AI agents that can
leverage human input and guidance, rather than operating in a fully autonomous w
ay. Let me know if you need any clarification or have additional questions!
```

Check out the trace to see the tool call and later LLM response. **Notice** that now the graph queries the search engine using our updated query term - we were able to manually override the LLM's search here!

All of this is reflected in the graph's checkpointed memory, meaning if we continue the conversation, it will recall all the *modified* state.

In [15]:

```
events = graph.stream(
    {
        "messages": (
            "user",
            "Remember what I'm learning about?",
        )
    },
    config,
    stream_mode="values",
)
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================

Remember what I'm learning about?
================================== Ai Message ==================================

Ah yes, now I remember - you mentioned earlier that you are learning about LangG
raph.

LangGraph is the framework I researched in my previous response, which is for bu
```

```
ilding conversational AI agents that can incorporate human input and oversight.

So based on our earlier discussion, it seems you are currently learning about an
d exploring the LangGraph system for creating human-in-the-loop AI agents. Pleas
e let me know if I have the right understanding now.
```

Congratulations! You've used `interrupt_before` and `update_state` to manually modify the state as a part of a human-in-the-loop workflow. Interruptions and state modifications let you control how the agent behaves. Combined with persistent checkpointing, it means you can `pause` an action and `resume` at any point. Your user doesn't have to be available when the graph
Congratulations! You've used `interrupt_before` and `update_state` to manually modify the state as a part of a human-in-the-loop workflow. Interruptions and state modifications let you control how the agent behaves. Combined with persistent checkpointing, it means you can `pause` an action and `resume` at any point. Your user doesn't have to be available when the graph
interrupts!

恭喜！您已使用 `interrupt_before` 和 `update_state` 手动修改状态，作为人机交互工作流程的一部分。中断和状态修改让您可以控制代理的行为方式。与持久检查点相结合，这意味着您可以在任何时候 `pause` 一个操作和 `resume` 。当图表中断时，您的用户不必在线！

The graph code for this section is identical to previous ones. The key snippets to remember are to add `.compile(..., interrupt_before=[...])` (or `interrupt_after`) if you want to explicitly pause the graph whenever it reaches a node. Then you can use `update_state` to modify the checkpoint and control how the graph should proceed.

本节的图形代码与前面的相同。如果您想在图形到达节点时显式暂停图形，则要记住的关键片段是添加 `.compile(..., interrupt_before=[...])` （或 `interrupt_after` ）。然后您可以使用 `update_state` 修改检查点并控制图表应如何进行。

Part 6: Customizing State

So far, we've relied on a simple state (it's just a list of messages!). You can go far with this simple state, but if you want to define complex behavior without relying on the message list, you can add additional fields to the state. In this section, we will extend our chat bot with a new node to illustrate this.

In the examples above, we involved a human deterministically: the graph **always** interrupted whenever an tool was invoked. Suppose we wanted our chat bot to have the choice of relying on a human.

One way to do this is to create a passthrough "human" node, before which the graph will always stop. We will only execute this node if the LLM invokes a "human" tool. For our convenience, we will include an "ask_human" flag in our graph state that we will flip if the LLM calls this tool.

Below, define this new graph, with an updated `State`

在上面的示例中，我们确定性地涉及人类：每当调用工具时，图形总是会中断。假设我们希望我们的聊天机器人可以选择依赖人类。

One way to do this is to create a passthrough "human" node, before which the graph will always stop. We will only execute this node if the LLM invokes a "human" tool. For our convenience, we will include an "ask_human" flag in our graph state that we will flip if the LLM calls this tool.

实现此目的的一种方法是创建一个直通“人类”节点，在该节点之前图形将始终停止。仅当 LLM 调用“人类”工具时，我们才会执行此节点。为了方便起见，我们将在图形状态中包含一个“ask_ human”标志，如果 LLM 调用此工具，我们将翻转该标志。

Below, define this new graph, with an updated `State`

下面，使用更新的 `State` 定义这个新图表

In [1]: 在[1]中:

```python
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition


class State(TypedDict):
    messages: Annotated[list, add_messages]
    # This flag is new
    ask_human: bool
```

Next, define a schema to show the model to let it decide to request assistance.

In [2]:

```python
from langchain_core.pydantic_v1 import BaseModel


class RequestAssistance(BaseModel):
    """Escalate the conversation to an expert. Use this if you are unable to as

    To use this function, relay the user's 'request' so the expert can provide
    """

    request: str
```

Next, define the chatbot node. The primary modification here is flip the `ask_human` flag if we see that the chat bot has invoked the `RequestAssistance` flag.

In [3]:

```python
tool = TavilySearchResults(max_results=2)
tools = [tool]
```

`RequestAssistance` 标志，则翻转 `ask_human` 标志。

In [3]: 在[3]中:

```python
tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
# We can bind the llm to a tool definition, a pydantic model, or a json s
llm_with_tools = llm.bind_tools(tools + [RequestAssistance])


def chatbot(state: State):
    response = llm_with_tools.invoke(state["messages"])
    ask_human = False
    if (
        response.tool_calls
        and response.tool_calls[0]["name"] == RequestAssistance.__name
```

```
    ):
        ask_human = True
    return {"messages": [response], "ask_human": ask_human}
```

```
/Users/wfh/code/lc/langchain/libs/core/langchain_core/_api/beta_decorator.py:87: LangChainBetaWarning: The method `ChatAnthropic.bind_tools` is in beta. It is actively being worked on, so the API may change.
  warn_beta(
```

Next, create the graph builder and add the chatbot and tools nodes to the graph, same as before.

接下来，创建图形生成器并将聊天机器人和工具节点添加到图形中，与之前相同。

In [4]:
```
graph_builder = StateGraph(State)

graph_builder.add_node("chatbot", chatbot)
graph_builder.add_node("tools", ToolNode(tools=[tool]))
```

Next, create the "human" `node`. This `node` function is mostly a placeholder in our graph that will trigger an interrupt. If the human does **not** manually update the state during the `interrupt`, it inserts a tool message so the LLM knows the user was requested but didn't respond. This node also unsets the `ask_human` flag so the graph knows not to revisit the node unless further requests are made.

In [5]:
```
from langchain_core.messages import AIMessage, ToolMessage


def create_response(response: str, ai_message: AIMessage):
    return ToolMessage(
        content=response,
        tool_call_id=ai_message.tool_calls[0]["id"],
    )


def human_node(state: State):
    new_messages = []
    if not isinstance(state["messages"][-1], ToolMessage):
        # Typically, the user will have updated the state during the interrupt.
        # If they choose not to, we will include a placeholder ToolMessage to
        # let the LLM continue.
            new_messages.append(
                create_response("No response from human.", state["messages"
            )
        return {
            # Append the new messages
            "messages": new_messages,
            # Unset the flag
            "ask_human": False,
        }


graph_builder.add_node("human", human_node)
```

Next, define the conditional logic. The `select_next_node` will route to the `human` node if the flag is set. Otherwise, it lets the prebuilt `tools_condition` function choose the next node.

Recall that the `tools_condition` function simply checks to see if the `chatbot` has responded with any `tool_calls` in its response message. If so, it routes to the `action` node. Otherwise, it ends the graph.

In [6]:
```
def select_next_node(state: State):
    if state["ask_human"]:
        return "human"
    # Otherwise, we can route as before
    return tools_condition(state)


graph_builder.add_conditional_edges(
    "chatbot",
    select_next_node,
    {"human": "human", "tools": "tools", "__end__": "__end__"},
)
```

Finally, add the simple directed edges and compile the graph. These edges instruct the graph to **always** flow from node `a` -> `b` whenever `a` finishes executing.

In [7]:
```
# The rest is the same
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge("human", "chatbot")
graph_builder.set_entry_point("chatbot")
memory = SqliteSaver.from_conn_string(":memory:")
graph = graph_builder.compile(
    checkpointer=memory,
    # We interrupt before 'human' here instead.
    interrupt_before=["human"],
)
```

If you have the visualization dependencies installed, you can see the graph structure below:

In [8]:
```
from IPython.display import Image, display

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
```

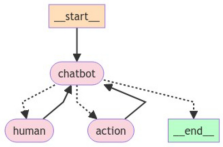
In [7]: 在[7]中:
```
# The rest is the same
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge("human", "chatbot")
graph_builder.set_entry_point("chatbot")
memory = SqliteSaver.from_conn_string(":memory:")
graph = graph_builder.compile(
    checkpointer=memory,
    # We interrupt before 'human' here instead.
    interrupt_before=["human"],
)
```

If you have the visualization dependencies installed, you can see the graph structure below:

如果您安装了可视化依赖项，您可以看到下面的图形结构：

In [8]: 在[8]中:
```
from IPython.display import Image, display

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
except:
    # This requires some extra dependencies and is optional
    pass
```



The chat bot can either request help from a human (chatbot->select->human), invoke the search engine tool (chatbot->select->action), or directly respond (chatbot->select->**end**). Once an action or request has been made, the graph will transition back to the `chatbot` node to continue operations.

Let's see this graph in action. We will request for expert assistance to illustrate our graph.

```
In [9]:
user_input = "I need some expert guidance for building this AI agent. Could you
config = {"configurable": {"thread_id": "1"}}
# The config is the **second positional argument** to stream() or invoke()!
events = graph.stream(
    {"messages": [("user", user_input)]}, config, stream_mode="values"
)
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================

I need some expert guidance for building this AI agent. Could you request assist
ance for me?
```

```
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================ Human Message =================================

I need some expert guidance for building this AI agent. Could you request assist
ance for me?
================================== Ai Message ==================================

[{'id': 'toolu_017XaQuVsoAyfXeTfDyv55Pc', 'input': {'request': 'I need some expe
rt guidance for building this AI agent.'}, 'name': 'RequestAssistance', 'type':
'tool_use'}]
Tool Calls:
  RequestAssistance (toolu_017XaQuVsoAyfXeTfDyv55Pc)
 Call ID: toolu_017XaQuVsoAyfXeTfDyv55Pc
  Args:
    request: I need some expert guidance for building this AI agent.
```

Notice: the LLM has invoked the "`RequestAssistance`" tool we provided it, and the interrupt has been set. Let's inspect the graph state to confirm.

```
In [10]:
snapshot = graph.get_state(config)
snapshot.next
```

```
Out[10]: ('human',)
```

The graph state is indeed **interrupted** before the `'human'` node. We can act as the "expert" in this scenario and manually update the state by adding a new ToolMessage with our input.

Next, respond to the chatbot's request by:

1. Creating a `ToolMessage` with our response. This will be passed back to the `chatbot`.
2. Calling `update_state` to manually update the graph state.

```
In [11]:
ai_message = snapshot.values["messages"][-1]
human_response = (
    "We, the experts are here to help! We'd recommend you check out LangGraph t
    " It's much more reliable and extensible than simple autonomous agents."
)
tool_message = create_response(human_response, ai_message)
graph.update_state(config, {"messages": [tool_message]})
```

```
Out[11]: {'configurable': {'thread_id': '1',
  'thread_ts': '2024-05-06T22:31:39.973392+00:00'}}
```

You can inspect the state to confirm our response was added.

```
In [12]:
graph.get_state(config).values["messages"]
```

```
Out[12]: [HumanMessage(content='I need some expert guidance for building this AI agent.
Could you request assistance for me?', id='ab75eb9d-cce7-4e44-8de7-b0b375a8697
2'),
 AIMessage(content=[{'id': 'toolu_017XaQuVsoAyfXeTfDyv55Pc', 'input': {'reques
t': 'I need some expert guidance for building this AI agent.'}, 'name': 'Reques
```

```
tool_message = create_response(human_response, ai_message)
graph.update_state(config, {"messages": [tool_message]})
```

```
Out[11]: 输出[11]:
{'configurable': {'thread_id': '1',
  'thread_ts': '2024-05-06T22:31:39.973392+00:00'}}
```

You can inspect the state to confirm our response was added.

您可以检查状态以确认我们的回复已添加。

```
In [12]: 在[12]中:
graph.get_state(config).values["messages"]
```

```
Out[12]: 输出[12]:
[HumanMessage(content='I need some expert guidance for building this
AI agent. Could you request assistance for me?', id='ab75eb9d-cce7-4e
44-8de7-b0b375a86972'),
 AIMessage(content=[{'id': 'toolu_017XaQuVsoAyfXeTfDyv55Pc', 'input':
{'request': 'I need some expert guidance for building this AI agen
t.'}, 'name': 'RequestAssistance', 'type': 'tool_use'}], response_met
adata={'id': 'msg_0199PiK6kmVAbeo1qmephKDq', 'model': 'claude-3-haiku
-20240307', 'stop_reason': 'tool_use', 'stop_sequence': None, 'usag
e': {'input_tokens': 486, 'output_tokens': 63}}, id='run-ff07f108-505
5-4343-8910-2fa40ead3fb9-0', tool_calls=[{'name': 'RequestAssistanc
e', 'args': {'request': 'I need some expert guidance for building thi
s AI agent.'}, 'id': 'toolu_017XaQuVsoAyfXeTfDyv55Pc'}]),
 ToolMessage(content="We, the experts are here to help! We'd recommen
d you check out LangGraph to build your agent. It's much more reliabl
e and extensible than simple autonomous agents.", id='19f2eb9f-a742-4
6aa-9847-60909c30e64a', tool_call_id='toolu_017XaQuVsoAyfXeTfDyv55P
c')]
```

Next, **resume** the graph by invoking it with `None` as the inputs.

```
In [13]:
events = graph.stream(None, config, stream_mode="values")
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
================================= Tool Message =================================

We, the experts are here to help! We'd recommend you check out LangGraph to buil
d your agent. It's much more reliable and extensible than simple autonomous agen
ts.
================================== Ai Message ==================================

It looks like the experts have provided some guidance on how to build your AI ag
ent. They suggested checking out LangGraph, which they say is more reliable and
extensible than simple autonomous agents. Please let me know if you need any oth
er assistance - I'm happy to help coordinate with the expert team further.
```

Notice that the chat bot has incorporated the updated state in its final response. Since **everything** was checkpointed, the "expert" human in the loop could perform the update at any time without impacting the graph's execution.

Congratulations! you've now added an additional node to your assistant graph to let the chat bot decide for itself whether or not it needs to interrupt execution. You did so by updating the graph `State` with a new `ask_human` field and modifying the interruption logic when compiling the graph. This lets you dynamically include a human in the loop while maintaining full **memory** every to let the chat bot decide for itself whether or not it needs to interrupt execution. You did so by updating the graph `State` with a new `ask_human` field and modifying the interruption logic when compiling the graph. This lets you dynamically include a human in the loop while maintaining full **memory** every time you execute the graph.

恭喜！现在，您已在助理图中添加了一个附加节点，以便聊天机器人自行决定是否需

要中断执行。您通过使用新的 `ask_human` 字段更新图形 `State` 并在编译图形时修改中断逻辑来完成此操作。这使您可以动态地将人员纳入循环中，同时在每次执行图形时保持完整的内存。

We're almost done with the tutorial, but there is one more concept we'd like to review before finishing that connects `checkpointing` and `state updates`.

我们即将完成本教程，但在完成之前我们还想回顾一下连接 `checkpointing` 和 `state updates` 的概念。

This section's code is reproduced below for your reference.

In [26]:
```python
from typing import Annotated, Union

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from langchain_core.pydantic_v1 import BaseModel
from typing_extensions import TypedDict

from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition


class State(TypedDict):
    messages: Annotated[list, add_messages]
    # This flag is new
    ask_human: bool


class RequestAssistance(BaseModel):
    """Escalate the conversation to an expert. Use this if you are unable to as

    To use this function, relay the user's 'request' so the expert can provide
    """

    request: str


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
# We can bind the llm to a tool definition, a pydantic model, or a json schema
llm_with_tools = llm.bind_tools(tools + [RequestAssistance])


def chatbot(state: State):
    response = llm_with_tools.invoke(state["messages"])
```

```python
def chatbot(state: State):
    response = llm_with_tools.invoke(state["messages"])
    ask_human = False
    if (
        response.tool_calls
        and response.tool_calls[0]["name"] == RequestAssistance.__nam
    ):
        ask_human = True
    return {"messages": [response], "ask_human": ask_human}


graph_builder = StateGraph(State)

graph_builder.add_node("chatbot", chatbot)
graph_builder.add_node("tools", ToolNode(tools=[tool]))


def create_response(response: str, ai_message: AIMessage):
    return ToolMessage(
        content=response,
        tool_call_id=ai_message.tool_calls[0]["id"],
    )


def human_node(state: State):
    new_messages = []
    if not isinstance(state["messages"][-1], ToolMessage):
        # Typically, the user will have updated the state during the i
        # If they choose not to, we will include a placeholder ToolMes
        # let the LLM continue.
        new_messages.append(
            create_response("No response from human.", state["message
        )
    return {
        # Append the new messages
        "messages": new_messages,
        # Unset the flag
        "ask_human": False,
    }


graph_builder.add_node("human", human_node)


def select_next_node(state: State):
    if state["ask_human"]:
        return "human"
    # Otherwise, we can route as before
    return tools_condition(state)


graph_builder.add_conditional_edges(
    "chatbot",
    select_next_node,
    {"human": "human", "tools": "tools", "__end__": "__end__"},
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge("human", "chatbot")
graph_builder.set_entry_point("chatbot")
memory = SqliteSaver.from_conn_string(":memory:")
graph = graph_builder.compile(
    checkpointer=memory,
    interrupt_before=["human"],
)
```

Part 7: Time Travel 第七部分：时间旅行

In a typical chat bot workflow, the user interacts with the bot 1 or more times to accomplish a task. In the previous sections, we saw how to add memory and a human-in-the-loop to be able to checkpoint our graph state and manually override the state to control future responses.

But what if you want to let your user start from a previous response and "branch off" to explore a separate outcome? Or what if you want users to be able to "rewind" your assistant's work to fix some mistakes or try a different strategy (common in applications like autonomous software engineers)?

You can create both of these experiences and more using LangGraph's built-in "time travel" functionality.

In this section, you will "rewind" your graph by fetching a checkpoint using the graph's `get_state_history` method. You can then resume execution at this previous point in time.

First, recall our chatbot graph. We don't need to make **any** changes from before:

In [2]:
```python
from typing import Annotated, Union, Literal

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import AIMessage, BaseMessage, ToolMessage
from langchain_core.pydantic_v1 import BaseModel
from typing_extensions import TypedDict
```

```
from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition


class State(TypedDict):
    messages: Annotated[list, add_messages]
    # This flag is new
    ask_human: bool


class RequestAssistance(BaseModel):
    """Escalate the conversation to an expert. Use this if you are unable to as

    To use this function, relay the user's 'request' so the expert can provide
    """

    request: str


tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-haiku-20240307")
```

```
    from langgraph.prebuilt import ToolNode, tools_condition


    class State(TypedDict):
        messages: Annotated[list, add_messages]
        # This flag is new
        ask_human: bool


    class RequestAssistance(BaseModel):
        """Escalate the conversation to an expert. Use this if you are unab

        To use this function, relay the user's 'request' so the expert can
        """

        request: str


    tool = TavilySearchResults(max_results=2)
    tools = [tool]
    llm = ChatAnthropic(model="claude-3-haiku-20240307")
    # We can bind the llm to a tool definition, a pydantic model, or a json s
    llm_with_tools = llm.bind_tools(tools + [RequestAssistance])


    def chatbot(state: State):
        response = llm_with_tools.invoke(state["messages"])
        ask_human = False
        if (
```