# LangGraph和LangSmith调研

## LangChain公司

Python包langchain和公司LangChain的目标是尽可能让开发者轻松构建具备推理能力的应用程序。虽然LangChain最初只是一个单一的开源包，但它已发展成为一家公司和一个完整的生态系统，其框架如下图所示。

**LangChain工具的架构（从整个公司产品角度）：**

# IMG_256

（1）LangChain:是一个用于构建大语言模型应用程序的框架。它通过提供一套可组合、模块化的抽象和工具集，将大语言模型与外部数据源、计算资源以及操作步骤连接起来，从而创建出功能强大的端到端应用。

（2）LangGraph**:**是一个低级别的编排框架，是LangChain的扩展，用于构建、管理和部署长时间运行、有状态、人机交互的智能体。

（官网：<https://www.langchain.com/langgraph）>

（3）Intergrations**:**是一系列独立的Python 包，每个包专门用于将 LangChain 核心框架与一个特定的第三方服务、提供商或工具（如 OpenAI、Google Cloud、PostgreSQL 等）进行连接和集成。

（列表：<https://python.langchain.com/docs/integrations/providers/>

详细API：<https://python.langchain.com/docs/integrations/providers/>）

（4）LangGraph Platform**:**LangGraph Platform 是一个用于在生产环境中部署和管理长周期、有状态智能体工作流的运行时平台。该平台提供执行、持久化、监控与扩展智能体应用的API支持。基于LangGraph或其他框架构建的智能体均可部署于此平台，并通过托管端点对外提供服务。

（5）LangSmith**:**一个开发人员平台，可让您调试、测试、评估和监控 LLM 应用程序。

（官网：<https://www.langchain.com/langsmith）>

（6）Langserve（上图中没有）**:**用于将 LangChain 链部署为 REST API 的包。可以轻松启动和运行生产就绪的API。

（文档：<https://python.langchain.com/docs/langserve/）>

**LangChain工具用于应用的生命周期（分工）：**

LangChain公司的产品简化了LLM应用程序生命周期的每个阶段。

开发阶段：使用 LangChain 的开源组件与第三方集成来构建您的应用程序。利用 LangGraph 构建具备一流数据流支持和人工干预功能的有状态智能体。

生产化阶段：运用 LangSmith 来检查、监控和评估您的应用程序，从而能够持续优化并充满信心地进行部署。

部署阶段：通过 LangGraph Platform 将您的 LangGraph 应用程序转化为可用于生产环境的 API 和智能助手。（LangServe则可用于LangChain的部署）

**工具开源与商业化：**

上述工具中，有四个部分是开源的分别是LangChain、LangGraph、LangServe和Intergrations。LangSmith和LangGraph Platform则为LangChain公司开发的商业产品。另外LangServe虽是开源的，但现只接受社区对LangServe的错误修复，而不再接受新功能贡献，并明确说明LangServe不适用于LangGraph的部署，建议采用LangGraph Platform。体现了LangChain公司的商业化发展。

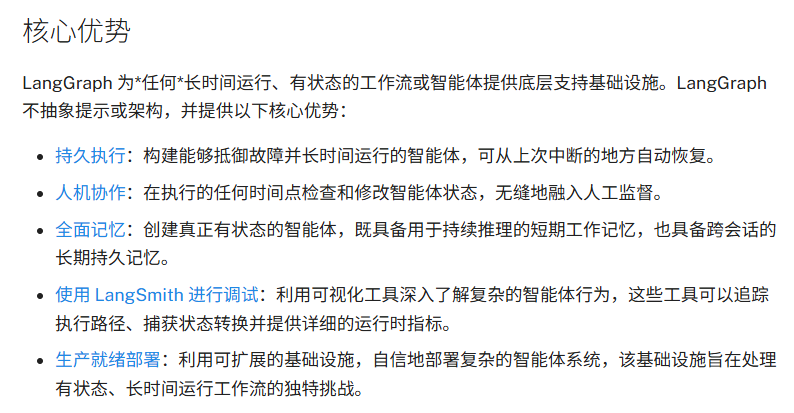
## LangGraph

LangGraph 是 LangChain 的一个扩展，旨在通过将步骤建模为图中的边和节点，构建健壮且有状态的基于LLM的**多智能体应用程序**。

（LangGraph is an extension of LangChain aimed at building robust and stateful multi-actor applications with LLMs by modeling steps as edges and nodes in a graph.）

**LangGraph的核心优势**：

从官方文档中（https://langgraph.com.cn/index.html#get-started）中提取到LangGraph的核心优势如图所示：



上图列出了五个核心优势，其中最后两条主要以来工具LangSmith和LangGraph Platform，前三条是LangGraph本身的属性。另外LangGraph核心是将工作步骤建模为图。所以从图、持久执行、人机协作、全面记忆四个角度来对LangGraph进行了解。

### 图

**基本概念：**

LangGraph 的核心是将代理工作流建模为图。可以使用三个关键组件来定义代理的行为：

（1）State：一个共享数据结构，表示应用程序的当前快照。它可以是任何 Python 类型，但通常是 TypedDict 或 Pydantic BaseModel。

（2）Nodes：编码代理逻辑的 Python 函数。它们接收当前的 State 作为输入，执行一些计算或副作用，并返回一个更新的 State。

（3）Edges：Python 函数，根据当前的 State 决定接下来执行哪个 Node。它们可以是条件分支或固定转换。

通过组合 Nodes 和 Edges，可以创建复杂、循环的工作流，使 State 随时间演进。然而，真正的力量来自于 LangGraph 如何管理 State。需要强调的是：Nodes 和 Edges 不过是 Python 函数——它们可以包含一个 LLM，也可以只是普通的 Python 代码。

LangGraph 的底层图算法使用消息传递来定义通用程序。当一个节点完成其操作时，它会沿着一条或多条边向其他节点发送消息。这些接收节点随后执行其函数，将结果消息传递给下一组节点，然后该过程继续。受 Google 的 Pregel 系统启发，程序以离散的“超步”进行。

超步可以被认为是图节点的一次迭代。并行运行的节点属于同一个超步，而顺序运行的节点属于不同的超步。在图执行开始时，所有节点都处于 inactive 状态。当节点在其任何传入边（或“通道”）上接收到新消息（状态）时，它就会变为 active。活跃节点随后运行其函数并返回更新。在每个超步结束时，没有传入消息的节点通过将自身标记为 inactive 来投票 halt（停止）。当所有节点都处于 inactive 状态且没有消息传输时，图执行终止。

StateGraph 类是主要的图类。它通过用户定义的 State 对象进行参数化。

要构建图，您首先定义状态，然后添加节点和边，最后编译它。编译是一个相当简单的步骤。它对图的结构进行一些基本检查（例如，没有孤立节点）。它也是可以指定运行时参数的地方，例如检查点和断点（后文会体现）。只需调用compile方法即可编译图。

**示例代码**

from typing import Annotated, List, TypedDict

from langgraph.graph import StateGraph, START, END

import operator

class State(TypedDict):

execution\_order: Annotated[List[str], operator.add]

def node\_a(state: State):return {"execution\_order": ["A"],}

def node\_b(state: State):return {"execution\_order": ["B"],}

def node\_c(state: State):return {"execution\_order": ["C"],}

# 1. 构建状态图

graph\_builder = StateGraph(State)

graph\_builder.add\_node("A", node\_a)

graph\_builder.add\_node("B", node\_b)

graph\_builder.add\_node("C", node\_c)

graph\_builder.add\_edge(START, "A")

graph\_builder.add\_edge("A", "B")

graph\_builder.add\_edge("A", "C")

graph\_builder.add\_edge("B", END)

graph\_builder.add\_edge("C", END)

graph=graph\_builder.compile()

result = graph.invoke({})

print(result)

上述示例代码建立了一个简单的状态图，用来展示图的建立和编译过程。首先定义状态，示例中状态为execution\_order是一个列表用于记录节点执行的顺序，利用状态即可定义状态图。定义状态图后向其中添加节点和边，示例包含三个节点，分别是node\_a、node\_b、node\_c每个节点执行时更新状态，将自生编号写入执行顺序列表。边的连接是从node\_a连接到node\_b和node\_c。定义完成后，进行图的编译,编译完成后可使用invoke或stream方法来执行。根据介绍的执行原理，A节点位于一个超步，B、C节点位于第二个超步，BC节点并行执行。

### 持久执行

持久执行是一种技术，其中进程或工作流在关键点保存其进度，使其能够暂停并在稍后从中断处精确恢复。这在需要人工干预的场景中特别有用，用户可以在继续之前检查、验证或修改进程，也适用于可能遇到中断或错误（例如，LLM 调用超时）的长时间运行任务。通过保留已完成的工作，持久执行使进程能够恢复，而无需重新处理之前的步骤——即使在长时间延迟（例如，一周后）之后也是如此。

**基本概念（检查点）：**

LangGraph有一个内置的持久化层，通过检查点器实现。当你使用检查点器编译图时，检查点器会在每个超级步骤保存图状态的checkpoint。这些检查点保存到thread中，可以在图执行后访问。因为 threads 允许在执行后访问图的状态，所以人机协作、记忆、时间旅行和容错等多种强大功能都成为可能。

检查点是每个超级步骤保存的图状态的快照，由具有以下关键属性的StateSnapshot 对象表示

config：与此检查点关联的配置。

metadata：与此检查点关联的元数据。

values：此时状态通道的值。

next：要在图中接下来执行的节点名称的元组。

tasks：包含要执行的后续任务信息的 PregelTask 对象元组。如果该步骤之前曾尝试过，它将包含错误信息。如果图在节点内部被动态中断，tasks 将包含与中断相关的附加数据。

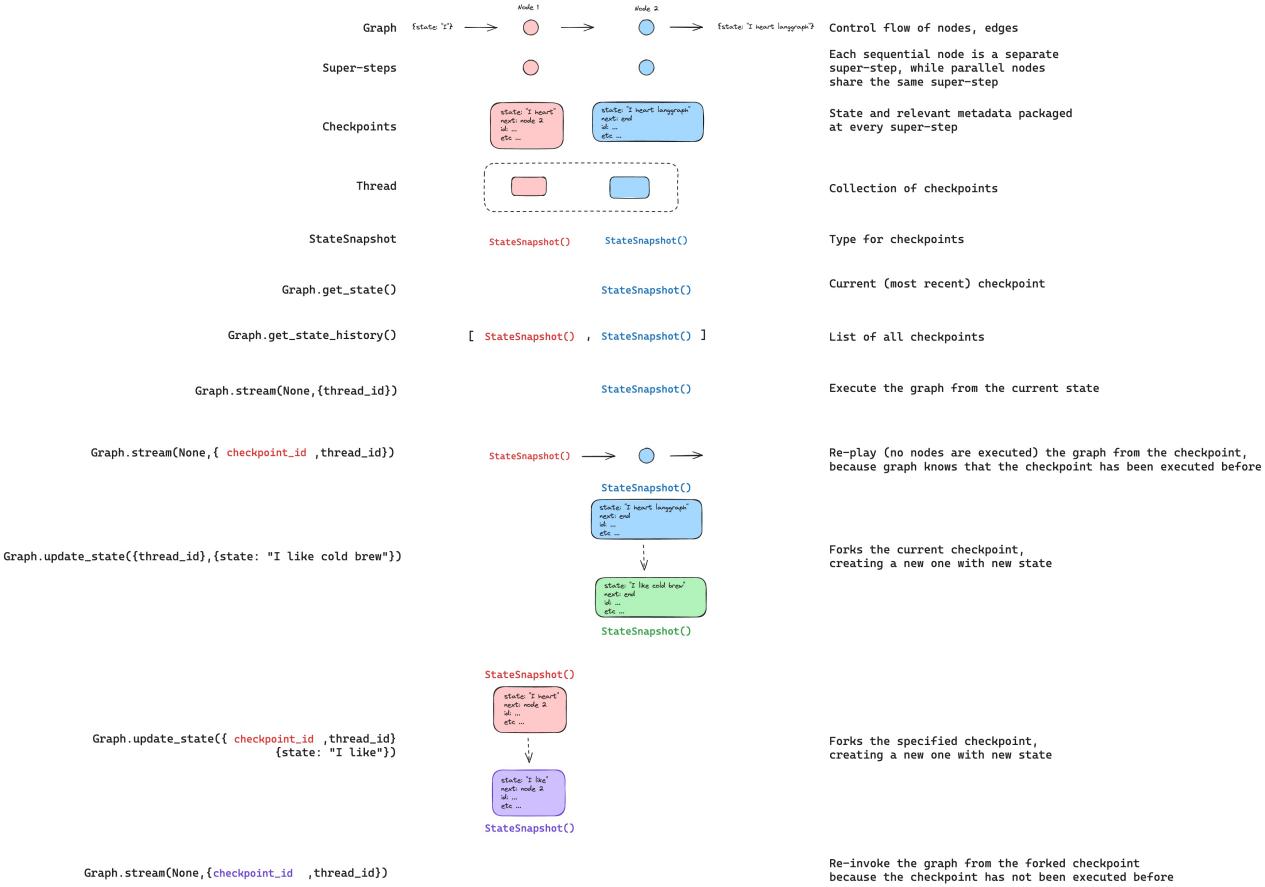
示例如下：



检查点的作用：

1. 获取状态历史
2. 回放（执行某检查点之后的步骤）
3. 更新状态：除了从特定的checkpoints回放图之外，还可以编辑图的状态。

下图展示了关于检查点应用的相关内容，包括get\_state()获取当前位置的快照、get\_state\_history()获取快照历史，Graph.stream()通过参数从特定检查点运行，Graph.update\_state()从特定检查点更新状态。



**使用持久执行的要求：**

要在 LangGraph 中利用持久执行，需要：

（1）通过指定一个将保存工作流进度的检查点器，在您的工作流中启用持久化。

（2）在执行工作流时指定一个会话标识符（thread\_id）。这将跟踪工作流特定实例的执行历史。

（3）将任何非确定性操作（例如，随机数生成）或具有副作用的操作（例如，文件写入、API 调用）封装在任务中，以确保当工作流恢复时，这些操作不会针对特定运行重复执行，而是从持久化层检索其结果。有关更多信息，请参阅确定性与一致性回放。

**恢复工作流：**

（1）暂停和恢复工作流：使用interrupt 函数在特定点暂停工作流，并使用Command 原语以更新的状态恢复它。（Interrupt是人机协同部分的关键，故在下一小结进行介绍）

（2）从失败中恢复：在异常（例如，LLM 提供商中断）后，自动从上次成功的检查点恢复工作流。这涉及到通过将None作为输入值来使用相同的会话标识符执行工作流。

**示例代码：**

主要通过示例代码来展示检查点的获取状态历史、回放、更新状态等功能，并示例如何从失败中恢复，体现LongGraph的持久执行特点。

from typing import Annotated, List, TypedDict

from langgraph.checkpoint.memory import InMemorySaver

from langgraph.graph import StateGraph, START, END

import operator

class State(TypedDict):

execution\_order: Annotated[List[str], operator.add]

error:bool

def node\_a(state: State):

if state["error"] == True:

raise ValueError("Failure")

return {"execution\_order": ["A"],}

def node\_b(state: State):return {"execution\_order": ["B"],}

def node\_c(state: State):return {"execution\_order": ["C"],}

# 1. 构建状态图

graph\_builder = StateGraph(State)

graph\_builder.add\_node("A", node\_a)

graph\_builder.add\_node("B", node\_b)

graph\_builder.add\_node("C", node\_c)

graph\_builder.add\_edge(START, "A")

graph\_builder.add\_edge("A", "B")

graph\_builder.add\_edge("A", "C")

graph\_builder.add\_edge("B", END)

graph\_builder.add\_edge("C", END)

checkpointer=InMemorySaver()

graph=graph\_builder.compile(checkpointer=checkpointer)

config = {"configurable": {"thread\_id": "1"}}

try:

result = graph.invoke({"error":True},config)

except ValueError as e:

print(e)

print("\ncurrent state:")

print(graph.get\_state(config).values)

print(graph.get\_state(config).config)

print("\nfirst fix:update state")

graph.update\_state(graph.get\_state(config).config,{"error":False})

print(graph.get\_state(config).config)

result = graph.invoke(None,graph.get\_state(config).config)

print("fixed\n")

print("history:")

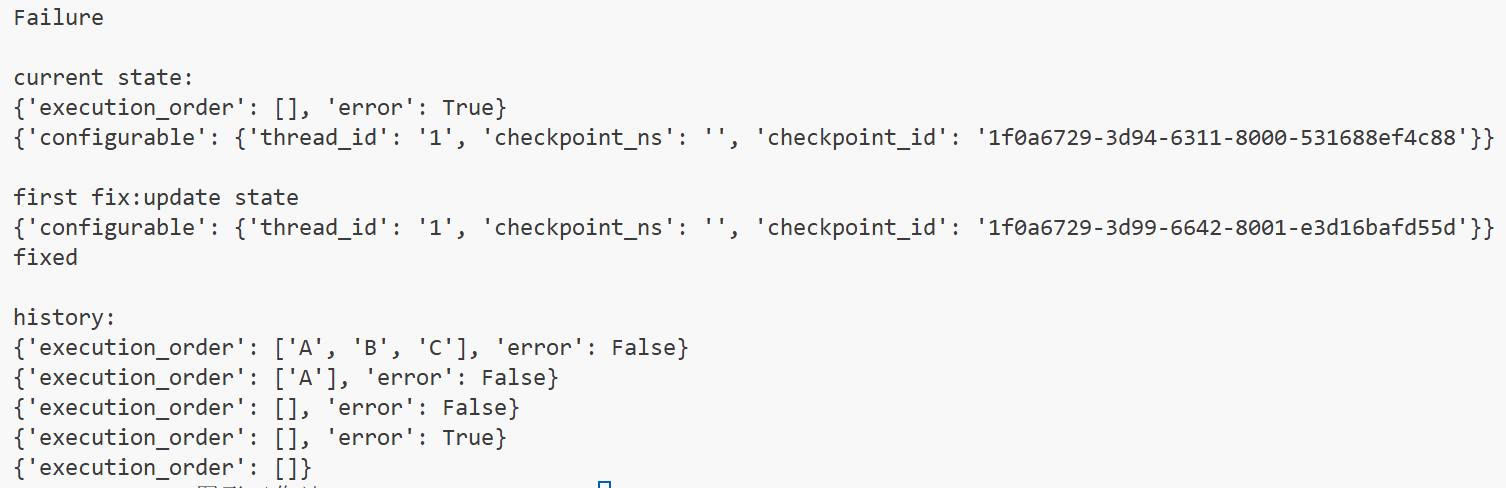
for history\_snapshot in list(graph.get\_state\_history(config)): print(history\_snapshot.values)

上述代码由上一小节的代码修改而来，主要修改如下：

1. State中增加了error，并在第一次运行时将其初始化为True，用来模拟程序运行出错。
2. 修改了node\_a，State中的error为True时，抛出一个ValueError。
3. 在最后的except ValueError部分，对错误进行修复，体现Langgraph支持持久运行的特质。

值得关注的是最后对错误进行修复的部分，图运行后，a节点由于State中的error设置为true会产生异常，进入错误修复程序。错误修复程序将更新状态将error设置为false，新建一个检查点，并从新检查点开始继续执行，从而修复错误。其中使用到的方法由：get\_state方法返回当前的检查点信息，update\_state方法从某检查点创建一个新的检查点，并更新状态值。Invoke从某个检查点开始执行。get\_state\_history返回历史检查点，上文的图中有对这些函数的直观介绍。

程序输出如下图所示。



首先输出报错Failure，然后输出get\_state的结果答应了检查点中的values和config值，可见State中的error为True，也可见当前检查点的checkpoint\_id。

接着开始尝试修复，执行update\_state方法，输出了新的检查点config值，可以发现建立了一个新的检查点。

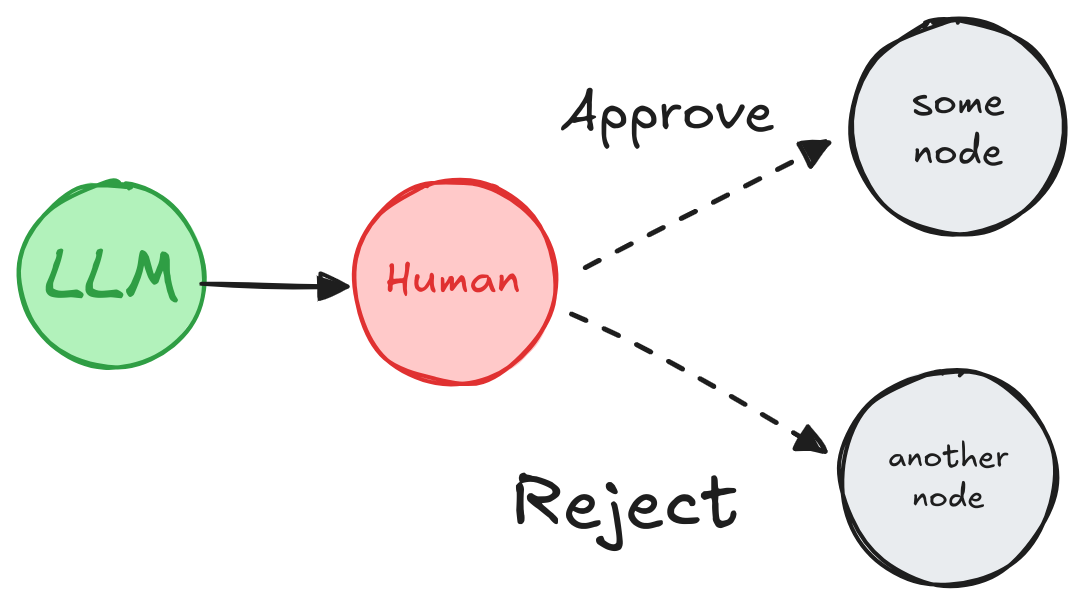
然后继续通过invoke从新的检查点执行图，执行完成后打印了所有历史检查点的values值。可以发现update\_state创建的检查点和之前的检查点都存在历史中，其中倒数第三条即为update\_state创建的检查点。正数第二条为A节点执行后创建的，正数第一条则为B、C节点执行完后创建，BC节点位于同一超步。

检查点机制，使得Langgraph可以获取状态历史、回放、更新状态，使其可以恢复和纠错，有助于持久执行智能体的开发。

### 人机协同

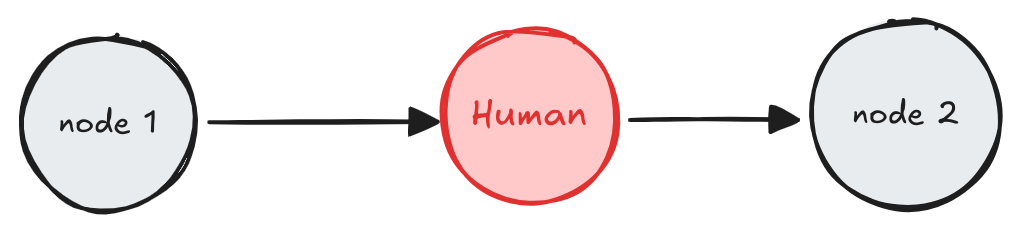
在执行过程中的任何时刻，通过检查和修改智能体状态，无缝地引入人类监督。常见的模式如下

1. 批准或拒绝



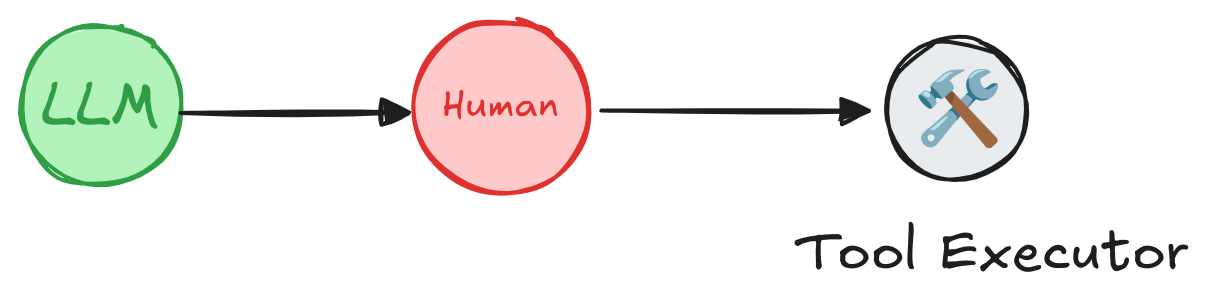
根据人工的批准或拒绝，图可以继续执行操作或采取替代路径

1. 审查和编辑状态



人工可以审查和编辑图的状态。这对于纠正错误或用附加信息更新状态很有用。

（3）审查工具调用



人工可以在继续之前审查和编辑来自 LLM 的输出。这在 LLM 请求的工具调用可能敏感或需要人工监督的应用中尤其关键。

**基本概念（中断与恢复）：**

支持在工作流中审查、编辑、批准工具的调用，以便在工作流的任何时刻进行人工干预。这在大型语言模型 (LLM) 驱动的应用中尤其有用，因为模型输出可能需要验证、修正或额外的上下文。  
 要审查、编辑和批准智能体或工作流中的工具调用，使用中断来暂停图并等待人工输入。中断使用 LangGraph 的持久化层，它会保存图的状态，从而无限期地暂停图的执行，直到恢复为止。

最终要的机制是Interrupt中断和Command恢复机制。

当在图中使用 interrupt 函数时，执行会在此处暂停并等待用户输入。

要恢复执行，需使用 Command 原语，它可以通过 invoke 或 stream 方法提供。图会从最初调用interrupt(...)的节点的开头恢复执行。这一次，interrupt函数将返回在Command(resume=value)中提供的值，而不会再次暂停。从节点开头到interrupt的所有代码都将被重新执行。

**示例代码：**

上述示例代码依旧与前文示例代码保持了尽可能小的改动。主要用于示范中断和恢复的过程，场景为前文提到的批准和拒绝。示例代码如下所示

from typing import Annotated, List, TypedDict

from langgraph.graph import StateGraph, START, END

import operator

from langgraph.checkpoint.memory import InMemorySaver

from langgraph.types import interrupt, Command

class State(TypedDict):

execution\_order: Annotated[List[str], operator.add]

def node\_a(state: State):return {"execution\_order": ["A"],}

def node\_people(state: State):

decision = interrupt("approve?")

if decision == "approve":

return Command(goto="B", update={"execution\_order":["approve"]})

else:

return Command(goto="C", update={"execution\_order":["reject"]})

def node\_b(state: State):return {"execution\_order": ["B"]}

def node\_c(state: State):return {"execution\_order": ["C"]}

# 1. 构建状态图

graph\_builder = StateGraph(State)

graph\_builder.add\_node("A", node\_a)

graph\_builder.add\_node("B", node\_b)

graph\_builder.add\_node("C", node\_c)

graph\_builder.add\_node("People", node\_people)

graph\_builder.add\_edge(START, "A")

graph\_builder.add\_edge("A","People")

graph\_builder.add\_edge("B", END)

graph\_builder.add\_edge("C", END)

checkpointer=InMemorySaver()

graph=graph\_builder.compile(checkpointer=checkpointer)

config = {"configurable": {"thread\_id": "1"}}

result = graph.invoke({},config=config)

print(result)

final\_result = graph.invoke(Command(resume="reject"), config=config)

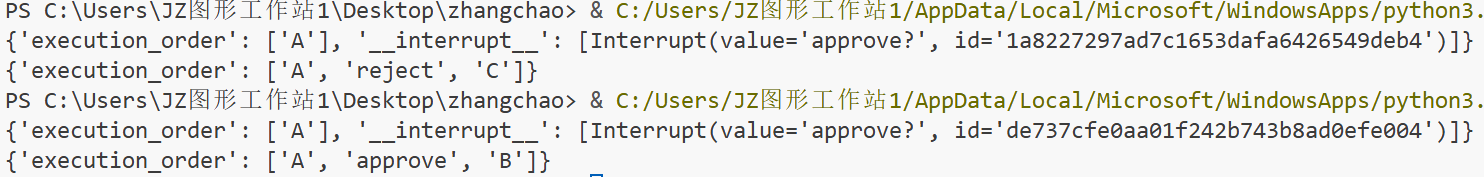
print(final\_result)

主要的代码修改如下：

1. 取消了节点a和b与c的连接，建立了新节点node\_people,将a节点连接到node\_people，但是node\_people不直接连接到b和c，node\_people后执行哪个节点由node\_people节点中的程序决定。
2. 新建立了node\_people节点，该节点的作用是决定下一个运行的节点是b还是c，当前，若用户意见为approve，则执行a，否则为b。
3. 最后执行图的部分，增加了中断后恢复的步骤。

具体执行过程为，第一次执行graph.invoke()，到达node\_people节点中的interrupt()函数后停止，打印出运行结果。然后进行第二次执行graph.invoke(Command（resume=“reject/approve”）)，本次执行将在上次node\_people节点的头部开始执行，当再次运行interrupt()时，节点获得resume中的输入并赋值给decision变量，并继续向下执行，根据decision决定下一个节点运行a还是b，Command(goto=?)方法决定了下一个运行的节点。

示例程序输出的结果如下图所示：



由示例程序的运行结果可知，当用户输入为approve时，执行了A->people->B，当输入为reject时执行了A->people->C，LangGraph的中断和恢复机制提供了有效的人机协同方法。

### 全面记忆

记忆是一种能记录过往交互信息的系统。对于人工智能（AI）智能体而言，记忆至关重要 —— 它能让智能体记住之前的交互内容、从反馈中学习，并根据用户偏好做出调整。当智能体需要处理更复杂的任务（这类任务往往涉及大量用户交互）时，记忆能力对于提升效率和用户满意度都不可或缺。

**基本概念：**

短期记忆（又称 “线程作用域记忆”）通过保存一个会话内的消息历史，来跟踪当前正在进行的对话。在 LangGraph（一种用于构建智能体工作流的框架）中，短期记忆是智能体“状态”（State）的一部分。系统会借助“检查点”（checkpointer）将状态持久化存储到数据库中，确保对话线程可在任意时间恢复。当调用该框架（graph）或完成某个步骤时，短期记忆会随之更新；而在每个步骤开始时，系统会读取当前的“状态”数据。

长期记忆用于存储跨会话的用户特定数据或应用级数据，且可在多个对话线程间共享。它能在任意时间、任意对话线程中被调用。长期记忆的作用域可自定义为任意命名空间（namespace），而非局限于单一的线程 ID。LangGraph 提供了专门的存储工具（参考文档见链接），支持用户保存和调用长期记忆。

LangGraph还提供了其他一些记忆相关的功能，例如采用语义搜索的方式来查找记忆，对于这些拓展功能还有待后续详细学习，暂时不在此描述。

**示例代码：**

通过下属代码主要测试LangGraph提供的短期和长期记忆，短期记忆通过图的状态和检查点实现，在线程内起作用主要通过State和InMemorySaver实现，长期记忆则可在多个线程间使用，主要通过InMemoryStore实现。

from typing import Annotated, List, TypedDict

from langgraph.graph import StateGraph, START, END

import operator

from langgraph.checkpoint.memory import InMemorySaver

from langgraph.store.memory import InMemoryStore

import uuid

class State(TypedDict):

    execution\_order: Annotated[List[str], operator.add]

def node\_a(state: State, store: InMemoryStore):

    namespace = ("node\_a","memorys")

    key = f"{uuid.uuid4()}"  # 避免冲突

    # 2. 准备要存储的原始数据（字典格式）

    data\_to\_store = {"node\_name": "A"}

    # 3. 向store写入数据（put需要3个参数：namespace, key, 原始数据）

    store.put(namespace, key, data\_to\_store)

    # 4. 从store读取数据（get返回Item对象，需通过.value获取原始数据）

    retrieved\_item = store.search(namespace)  # 返回Item对象

    print(retrieved\_item )

    # 返回节点A对状态的更新

    return {"execution\_order": ["A"]}

# 节点B

def node\_b(state: State):return {"execution\_order": ["B"]}

# 节点C（无修改，保持原逻辑）

def node\_c(state: State):return {"execution\_order": ["C"]}

# 1. 构建状态图

graph\_builder = StateGraph(State)

# 节点A通过lambda传递store实例（因add\_node的节点函数仅接收state参数）

graph\_builder.add\_node("A", lambda state: node\_a(state, store))

graph\_builder.add\_node("B", node\_b)

graph\_builder.add\_node("C", node\_c)

# 2. 定义节点间的边（流程：START→A，A→B/C，B/C→D，D→END）

graph\_builder.add\_edge(START, "A")

graph\_builder.add\_edge("A", "B")

graph\_builder.add\_edge("A", "C")

graph\_builder.add\_edge("B", END)

graph\_builder.add\_edge("C", END)

# 3. 初始化检查点（用于状态持久化）和存储（用于自定义数据存储）

checkpointer = InMemorySaver()

store = InMemoryStore()  # 实例化存储，供节点A使用

# 4. 编译图（仅需传入checkpointer，store已通过lambda传递给节点A）

graph = graph\_builder.compile(checkpointer=checkpointer)

# 5. 执行图（两个独立线程，验证存储隔离性）

initial\_state = {

    "execution\_order": [],

}

# 线程1执行

print("="\*50)

print("线程1执行第一次")

print("="\*50)

config\_thread = {"configurable": {"thread\_id": "1"}}  # 线程1的配置（含唯一ID）

# 首次invoke会执行到D节点的interrupt，第二次invoke继续执行（因checkpointer保存状态）

result\_thread1\_1 = graph.invoke(initial\_state, config\_thread)

print("="\*50)

print("线程1执行第二次")

print("="\*50)

result\_thread1\_2 = graph.invoke({}, config\_thread)  # 空输入表示继续上一状态

# 打印线程1结果

print(f"\n线程1最终执行顺序: {result\_thread1\_2['execution\_order']}")

# 线程2执行（独立于线程1，状态和存储数据隔离）

print("\n" + "="\*50)

print("线程2执行第一次")

print("="\*50)

config\_thread2 = {"configurable": {"thread\_id": "2"}}  # 线程2的配置（含唯一ID）

result\_thread2\_2 = graph.invoke({}, config\_thread2)

# 打印线程2结果

print(f"\n线程2最终执行顺序: {result\_thread2\_2['execution\_order']}")

# 6. 查看状态历史

print("\n" + "="\*50)

print("线程1状态历史")

print("="\*50)

for idx, snapshot in enumerate(graph.get\_state\_history(config\_thread)):

    print(f"历史快照{idx+1}:")

    print(f"  状态值: {snapshot.values}")

print("\n" + "="\*50)

print("线程2状态历史")

print("="\*50)

for idx, snapshot in enumerate(graph.get\_state\_history(config\_thread2)):

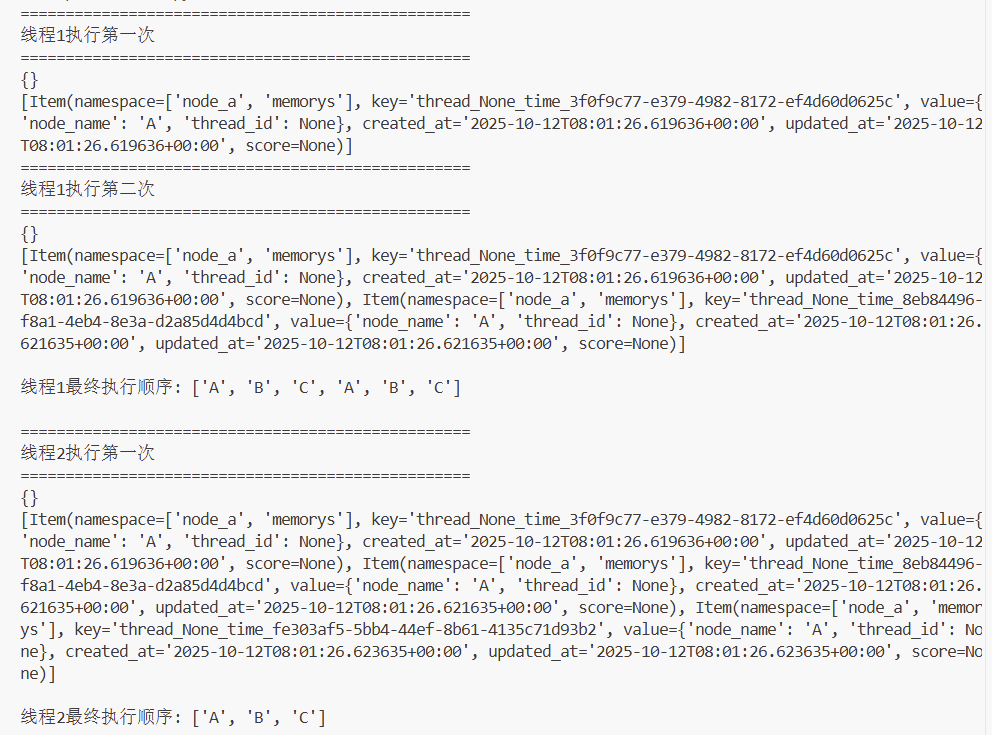
    print(f"历史快照{idx+1}:")

    print(f"  状态值: {snapshot.values}")

    print(f"  config: {snapshot.config}")

上述程序依旧采用三个节点的结构，主要修改是：增加了长期记忆，修改了A节点的功能，在A节点中向长期记忆以同一命名空间写入数据，写入后读取该命名空间中的所有数据并打印。

本程序首先采用线程ID为1对图运行两次，再使用线程ID为2对图运行两次，且只在线程ID为1运行第一次时初始化State的值。运行结果如下图所示。



首先看短期记忆，线程1两次执行后最终执行顺序输出为两次ABC，但是再执行一次线程2，最终的执行顺序却并非是三次ABC，而只是一次ABC。说明不同线程之间的State不可见，若打印其执行历史可发现其检查点也互相不可见。

再看长期记忆，每次A节点执行后都会向长期记忆的存储中写入数据，并打印所有已存在的数据。由上可见，虽运行图的线程号不同，但长期记忆中存在的数据在三次执行中依次增多，体现了长期记忆跨线程的特点。

## LangSmith

LangSmith 是一个用于构建生产级 LLM 应用程序的平台。监控和评估您的应用程序，以便您可以快速、自信地交付。

**LangSmith 与框架无关——你可以在有或没有 LangChain 的开源框架 langchain 和 langgraph 的情况下使用它。**

可观测性是使用大型语言模型（LLM）构建的应用程序的关键要求。LLM是不确定的，这意味着相同的提示可以产生不同的响应。这种行为使调试和监控比传统软件更具挑战性。

LangSmith通过提供对应用程序如何处理请求的端到端可见性来解决这一问题。每个请求都会生成一个追踪记录，其中完整记录了整个处理过程。在单个追踪记录中包含多个独立运行单元，即应用程序执行的具体操作（如LLM调用或检索步骤）。通过追踪这些运行单元，您可以检查、调试和验证应用程序的行为表现。

支持可视化的追踪、调试和监控LangChain应用，应用表现不佳时可以清晰看到每步的输入和输出，以及模型究竟在想什么，可用于优化提示词、分析错误、提升应用的性能。

LangChain：

教程：https://python.langchain.com/docs/how\_to/

<https://python.langchain.ac.cn/docs/how_to/>

LLM和提示词（Prompt）

链（Chain）

LCEL表达式

检索增强生产（RAG）

Agent智能体

Memory模型记忆

## 调研总结