Langgraph的记忆体调研

记忆体

- 实现多轮对话
- 通过列表维护历史聊天记录

Chain -> Graph

- Langchain中，需要自行定义ChatMessageHistory
  - 将历史记录作为文本块注入 Prompt，这种方式在处理复杂逻辑时存在局限性
- LangGraph 引入了 **"状态机（State Machine）"** 的概念，将记忆重构为两个核心层级
  - 短期记忆：Agent内部记忆，当前对话中的历史记忆，langgraph将他封装成CheckPoint
  - 长期记忆：Agent外部记忆，用第三方长久储存聊天信息，langgraph将封装成Store

[new] 记忆注入机制详解

> 本章节详细说明 LangGraph 记忆是如何传递给 LLM 的，包括上下文、提示词、消息列表等多种注入方式。

记忆的本质：LLM是无状态的

LLM 本身是**无状态的**——每次调用都是独立的，模型不会自动记住之前的对话。因此，所有的"记忆"功能本质上是通过以下方式实现的：

注入位置	说明	适用场景
消息列表 (Messages)	将历史对话作为消息序列传入 LLM	短期记忆、多轮对话
系统提示词 (System Prompt)	将长期记忆/用户偏好注入到系统提示中	长期记忆、个性化
上下文注入 (Context Injection)	在用户消息前后插入检索到的信息	语义记忆、RAG

短期记忆的注入方式：消息列表

核心机制

短期记忆（Checkpoint）的工作原理：

```
┌─────────────────────────────────────────────────────────┐
│  用户发送消息                                            │
│       ↓                                                 │
│  Checkpointer 根据 thread_id 加载历史状态                │
│       ↓                                                 │
│  state["messages"] 被填充为：                            │
│  [HumanMessage, AIMessage, HumanMessage, AIMessage, ...]  │
│       ↓                                                 │
│  开发者将 state["messages"] 传递给 LLM                   │
```

```
|       ↓                                                    |
|  LLM 生成回复                                              |
|       ↓                                                    |
|  新消息追加到 state["messages"]                            |
|       ↓                                                    |
|  Checkpointer 保存更新后的状态                             |
└────────────────────────────────────────────────────────────┘
```

代码实现细节

关键点：`state["messages"]` 存储了对话历史，但**开发者需要主动将这些消息传递给 LLM。**

```python
from langchain_core.messages import SystemMessage, HumanMessage

def llm_call(state: dict):
    """LLM 节点：将消息历史传递给模型"""

    # 方式1：直接传递所有消息
    response = llm.invoke(state["messages"])

    # 方式2：添加系统提示 + 消息历史
    response = llm.invoke([
        SystemMessage(content="你是一个有帮助的助手"),
        *state["messages"]  # 展开历史消息
    ])

    return {"messages": [response]}
```

使用 MessagesPlaceholder 模板

LangChain 提供 `MessagesPlaceholder` 来动态注入消息历史：

```python
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

# 创建 Prompt 模板
prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个智能助手，请根据对话历史回答问题。"),
    MessagesPlaceholder(variable_name="messages"),  # 历史消息插入点
    ("human", "{current_input}")
])

# 使用时
formatted_prompt = prompt.format_messages(
    messages=state["messages"],  # 从 state 获取历史
    current_input="用户的新问题"
)
response = llm.invoke(formatted_prompt)
```

MessagesPlaceholder 的作用：

- 在 Prompt 模板中预留一个"槽位"
- 运行时将消息列表动态插入该位置

- 支持 `n_messages` 参数限制消息数量

长期记忆的注入方式：提示词注入

核心机制

长期记忆（Store）的注入流程：

```
┌─────────────────────────────────────────────────────────────┐
│  用户发送消息                                                │
│       ↓                                                     │
│  从 Store 检索相关记忆（可使用语义搜索）                      │
│       ↓                                                     │
│  将检索结果注入到 System Prompt 中                           │
│       ↓                                                     │
│  构建完整的 Prompt：System + 历史消息 + 用户消息              │
│       ↓                                                     │
│  LLM 生成回复                                               │
└─────────────────────────────────────────────────────────────┘
```

代码实现细节

```python
from langgraph.store.memory import InMemoryStore

store = InMemoryStore()

def prompt_with_memory(state, store):
    """动态构建包含长期记忆的 Prompt"""

    # 1. 语义搜索：根据用户最新消息检索相关记忆
    user_message = state["messages"][-1].content
    memories = store.search(
        ("memories",),                    # 命名空间
        query=user_message,               # 搜索查询
        limit=5                           # 返回最相关的5条
    )

    # 2. 将记忆格式化为文本
    memory_context = "\n".join([
        f"- {mem.value['content']}"
        for mem in memories
    ])

    # 3. 构建包含记忆的 System Prompt
    system_msg = f"""你是一个购物助手，拥有持久记忆能力。

以下是关于用户的相关记忆：
{memory_context}

请根据这些记忆提供个性化的回答。"""

    return system_msg
```

记忆类型与注入位置

记忆类型	存储内容	注入位置	示例
语义记忆 (Semantic)	用户偏好、事实性知识	System Prompt	"用户喜欢素食"
情景记忆 (Episodic)	过去的交互经验	Few-shot 示例	"上次推荐X成功了"
程序记忆 (Procedural)	行为规则、指令	System Prompt 本身	Agent 的行为指南

记忆管理：避免上下文溢出

由于 LLM 有**上下文窗口限制**，需要对记忆进行管理：

```
from langchain_core.messages import trim_messages, SystemMessage

# Token 感知的消息修剪
trimmer = trim_messages(
    strategy="last",        # 保留最新的消息
    max_tokens=1200,        # 最大 token 数
    token_counter=llm       # 使用 LLM 计算 token
)

def respond_node(state):
    # 构建消息列表：系统提示 + 修剪后的历史
    system = SystemMessage(content="你是一个有帮助的助手")
    history = state.get("messages", [])

    # 修剪历史消息
    trimmed = trimmer.invoke([system, *history])

    # 调用 LLM
    response = llm.invoke(trimmed)
    return {"messages": [response]}
```

完整示例：短期 + 长期记忆组合

```
from langgraph.prebuilt import create_react_agent
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.store.memory import InMemoryStore

# 初始化存储
checkpointer = InMemorySaver()  # 短期记忆
store = InMemoryStore()           # 长期记忆

# 自定义 Prompt 函数：组合两种记忆
def custom_prompt(state, store):
    """动态生成包含长期记忆的系统提示"""

    # 从 Store 检索用户偏好
    user_id = state.get("user_id", "default")
```

```
    preferences = store.get(("users", user_id), "preferences")

    base_prompt = "你是一个个性化助手。"

    if preferences:
        base_prompt += f"\n\n用户偏好：\n{preferences.value}"

    return base_prompt

# 创建 Agent
agent = create_react_agent(
    model=llm,
    tools=[...],
    checkpointer=checkpointer,    # 短期记忆：自动管理对话历史
    store=store,                  # 长期记忆：跨会话持久化
    state_modifier=custom_prompt # 自定义 Prompt 生成
)

# 运行
config = {
    "configurable": {
        "thread_id": "session_001",     # 短期记忆隔离
        "user_id": "user_123"           # 长期记忆命名空间
    }
}
response = agent.invoke({"messages": [...]}, config)
```

记忆注入总结

关键要点:

1. **短期记忆**通过 `state["messages"]` 以**消息列表**形式传递给 LLM
2. **长期记忆**通过**语义搜索**检索后，注入到 **System Prompt** 中
3. **开发者需要主动**将记忆传递给 LLM，Checkpointer 只负责存储和加载状态
4. 需要通过 **trim/summarize** 管理上下文长度，避免超出 token 限制

Langgraph的短期记忆

机制

- 保存了 Graph 在每一步执行后的**完整状态快照**
  - **Values**: 当前状态通道中的实际数据（如 `messages`）
  - **Next**: 指示图在当前状态下计划执行的**下一个节点**（断点续传的关键）。
  - **Config**: 包含 `thread_id` 和 `checkpoint_id`（版本号）。
  - **Tasks**: 待执行的任务队列及错误信息（Error Information）。
- 线程隔离
  - `thread_id` 是持久化层的隔离边界。
  - Checkpointer 可同时服务数千个并发用户，只要 `thread_id` 不同，状态互不干扰。

基础代码实现（使用 `InMemorySaver` 进行内存级持久化）

```
import os
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.prebuilt import create_react_agent
from langchain_openai import ChatOpenAI


API_BASE = "https://api.api-store.store/v1"
API_KEY = "sk-LBT52Ag9pCmNcNEF6mC40IGENvOI6hokIKMcbG2GmQgooL44"

llm = ChatOpenAI(
    model="gemini-1.5-flash",
    openai_api_key=API_KEY,
    base_url=API_BASE,
    temperature=0
)

checkpointer = InMemorySaver()

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

# 构建agent时候组建一个checkpointer
agent = create_react_agent(
    model=llm,
    tools=[get_weather],
    checkpointer=checkpointer
```

```
)

# 运行agent
config = {"configurable": {"thread_id": "demo_thread_1"}}

print("--- 第一轮：问青岛 ---")
response_1 = agent.invoke(
    {"messages": [{"role": "user", "content": "青岛天气怎么样？"}]},
    config
)
print(response_1["messages"])

print("\n--- 第二轮：问杭州 (测试记忆) ---")
response_2 = agent.invoke(
    {"messages": [{"role": "user", "content": "杭州呢？"}]},
    config
)
print(response_2["messages"])
```

后端选择

- 实际项目中要根据不同情况选择储存后端
-

后端实现	适用场景	特点
InMemorySaver	本地调试/测试	进程重启即丢失，速度最快。
SqliteSaver	单机应用/轻量级	文件级存储，部署简单。
PostgresSaver	企业级生产环境	利用数据库事务，高可靠性，支持复杂查询。
RedisSaver	高频对话/实时系统	**读写性能最高**（快 12-30 倍），适合大规模并发。

Langgraph 管理短期记忆

- Summarization: 对记忆进行总结
- Trimming: 对短期记忆中最旧的消息进行删除

Summarization

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

# State 保存上一次总结的结果，不需要每次都需要重新总结一次
class State(AgentState):
```

```
    context: dict[str,Any]

checkpointer = InMemorySaver()

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

Trimming

```
def pre_model_hook(state):
    trimmed_messages = trim_messages(
        state["messages"],
        strategy = "last",
        token_counter = count_tokens_approximately,
        max_tokens = 384,
        start_on = "human",
        end_on = ("human", "tool"),
    )

    return {"llm_input_messages": trimmed_messages}

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

Langgraph管理长期记忆

- 主要通过Agent的store属性指定一个实现类
- 长期记忆通过namespace来区分不同的命名空间

```
# 定义
store = InMemoryStore()
# 测试数据
store.put(
    ("users",),
    "user_123",
    {
        "name": "大三",
        "age": "20"
    }
)
```

```python
# 定义工具
@tool(return_direct = True)
def fet_user_info(config: RunnableConfig) -> str:
    store = get_store()
    user_id = config["configurable"].get("user_id")
    user_info = store.get(("users",), user_id)
    return str(user_info.value) if user_info else "Unknown user"

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

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

NEW 记忆类型的认知科学视角

> LangGraph 的记忆系统设计借鉴了人类认知科学中的记忆分类理论。

三种核心记忆类型

记忆类型	人类认知类比	LangGraph 实现	典型应用
语义记忆 (Semantic)	知识性记忆：事实、概念	Store + 语义搜索	"用户喜欢素食"
情景记忆 (Episodic)	经验性记忆：事件、场景	Few-shot 示例存储	"上次这样处理成功了"
程序记忆 (Procedural)	技能性记忆：如何做	System Prompt	Agent 行为规则

语义记忆实现

```python
from langgraph.store.memory import InMemoryStore
from langchain_openai import OpenAIEmbeddings

# 带语义搜索能力的 Store
store = InMemoryStore(
    index={
        "dims": 1536,
        "embed": OpenAIEmbeddings(model="text-embedding-3-small"),
        "fields": ["content"]  # 指定要索引的字段
    }
)

# 存储语义记忆
store.put(
    ("facts", "user_123"),          # 命名空间
    "food_preference",               # 键
```

```
    {"content": "用户偏好素食，对花生过敏"}  # 值
)

# 语义检索
results = store.search(
    ("facts", "user_123"),
    query="用户的饮食习惯是什么？",    # 自然语言查询
    limit=5
)
```

情景记忆实现：Few-shot 学习

```
# 存储成功的交互作为示例
store.put(
    ("episodes", "user_123"),
    "successful_recommendation",
    {
        "situation": "用户询问餐厅推荐",
        "action": "推荐了三家素食餐厅并说明理由",
        "outcome": "用户表示满意"
    }
)

# 在 Prompt 中注入 Few-shot 示例
def inject_episodes(state, store):
    episodes = store.search(
        ("episodes", state["user_id"]),
        query=state["messages"][-1].content,
        limit=3
    )

    examples = "\n".join([
        f"情境：{ep.value['situation']}\n"
        f"行动：{ep.value['action']}\n"
        f"结果：{ep.value['outcome']}"
        for ep in episodes
    ])

    return f"以下是过去成功的交互案例，请参考：\n{examples}"
```

程序记忆实现：可进化的 System Prompt

```
# 存储可更新的行为规则
store.put(
    ("procedures",),
    "email_style",
    {"prompt": "写邮件时使用正式语气，每封邮件不超过200字"}
)

# 根据用户反馈更新规则
def update_procedure(feedback, store):
```

```
    current = store.get(("procedures",), "email_style")

    # 使用 LLM 根据反馈更新规则
    updated_prompt = llm.invoke(f"""
    当前规则：{current.value['prompt']}
    用户反馈：{feedback}
    请更新规则以满足用户需求。
    """)

    store.put(("procedures",), "email_style", {"prompt": updated_prompt})
```

人类监管 Human in the loop

- 手动干预大预言模型的过程

Langgraph的实现方法

- 人类利用 Checkpointer 的读写能力，对运行中的 Agent 进行**暂停、修改、回溯和分叉。**
- 通过interrupt() 来添加人类监督，监督时需要中断当前任务，和stream流式方法来配合

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

def book_hotel(hotel_name: str):
    """预定宾馆。这是一个敏感操作，需要人类审批。"""
    print(f"\n[Tool 内部] AI 正在尝试预定: {hotel_name}，正在请求人类介入...")
    # --- 关键点：触发中断 ---
    # 程序运行到这行代码会暂停！
    # 括号里的内容会作为中断信息返回给主程序
    human_feedback = interrupt({
        "message": f"正准备执行预定操作，目标: {hotel_name}。",
        "options": "请输入 'ok' 同意，或者输入 'edit' 修改名称。"
    })

    # --- 恢复后的逻辑 ---
    # 当人类发送 resume 指令后，程序会从上一行继续，human_feedback 就是人类输入的数据

    print(f"[Tool 内部] 收到人类反馈: {human_feedback}")

    action_type = human_feedback.get("type")

    if action_type == "ok":
        return f"成功！已为您预定 {hotel_name}。"

    elif action_type == "edit":
        new_name = human_feedback.get("new_name")
        return f"已根据人类指令修改：成功预定 {new_name}。"

    else:
```

```
        return f"操作被人类拒绝。"

# --- 3. 创建带记忆的 Agent ---
tools = [book_hotel]
checkpointer = InMemorySaver()

agent = create_react_agent(
    model=llm,
    tools=tools,
    checkpointer=checkpointer
)


# 必须指定 thread_id，否则无法断点续传
thread_config = {"configurable": {"thread_id": "human_supervision_demo_1"}}

inputs = {"messages": [{"role": "user", "content": "帮我在希尔顿酒店定一个房间"}]}
# 第一次运行
for update in agent.stream(inputs, config=thread_config):
    # 打印一些流式信息，看看它在干嘛
    if "__interrupt__" in update:
        # 这里捕捉到了中断信号
        print(f"\n🛑 检测到中断信号！AI 已暂停。")
        print(f"中断内容: {update['__interrupt__'][0].value}")
    else:
        pass # 正常打印其他消息

print("\n=== 阶段 2: 人类介入 (模拟前端操作) ===")

# 此时程序是停止状态。我们可以检查当前的 State
current_state = agent.get_state(thread_config)
# 确认确实有中断任务
if current_state.tasks and current_state.tasks[0].interrupts:
    print("系统提示：当前有一个挂起的审批任务。")

    # 模拟人类在控制台输入
    user_decision = input("请输入指令 (输入 'ok' 同意，或 'edit' 修改): ").strip()

    resume_payload = {}
    if user_decision == "edit":
        new_hotel = input("请输入新的酒店名称: ").strip()
        resume_payload = {"type": "edit", "new_name": new_hotel}
    else:
        resume_payload = {"type": "ok"}

    print(f"\n>>> 人类发送指令: {resume_payload}，恢复执行...")

    # --- 关键点：发送 Command 恢复执行 ---
    # 我们不传新的 messages，而是传 Command(resume=...)
    # 这会把数据直接塞回 tool 里的 interrupt() 函数返回值
    for update in agent.stream(Command(resume=resume_payload), config=thread_config):
        # 打印最后的结果
        if "agent" in update:
```

```
            print(f"\n🤖 AI 最终回复: {update['agent']['messages'][-1].content}")

else:
    print("没有检测到中断，任务可能已经直接完成了。")
```

动态中断

- Agent 发现参数缺失，或者工具执行前需要二次确认
- interrupt(value)

```
# 在工具内部
human_feedback = interrupt({"msg": "我要执行高危操作，请审批"})
# 此时程序挂起，等待人类发送 Command(resume=...)
```

静态中断

- 在构建图（Graph）时定义，无论 Agent 状态如何，走到特定节点前必须暂停。
- `interrupt_before` 或 `interrupt_after`

```
# 编译图时指定
graph = builder.compile(
    checkpointer=checkpointer,
    interrupt_before=["bank_transfer_node"] # 进入转账节点前强制暂停
)
```

状态干预

- 人类不仅仅能说"Yes/No"，还能直接修改 Agent 的"大脑"（State）

Data Correction

- 当人类发现 Agent 的中间变量错误时，可以直接在后台修改，而不需要 Agent 重试

```
graph.update_state(config, {"leaf_status": "semi-green"})
# Agent 的记忆被覆写
```

Mock Execution

- 人类可以替某个节点完成工作

```
graph.update_state(config, values, as_node="analysis_tool")
# 通过 as_node 参数，人类告诉系统："analysis_tool 这个节点已经跑完了，结果是 X"
# 系统跳过该节点的实际执行，直接流转到下一步。
```

Time Travel

- 允许对历史进行回溯和分叉

历史回溯

```
graph.get_state_history(config)
```

分叉执行

- 如果 Agent 在第 5 步犯错，人类可以在第 10 步发现后，选择回到第 5 步重新开始，分叉运行
- ```
  State A -> State B -> State C (这是第一次跑出来的烂结果，我们不要了)
                |
                +-> State D (这是我们分叉出来的新结果)
  ```
- 代码实现
  - 正常运行

    ```python
    import os
    from langgraph.checkpoint.memory import InMemorySaver
    from langgraph.prebuilt import create_react_agent
    from langchain_openai import ChatOpenAI

    # 1. 初始化
    checkpointer = InMemorySaver()
    llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0) # 假设用 GPT
    agent = create_react_agent(llm, tools=[], checkpointer=checkpointer)

    # 2. 设定 Thread ID
    config = {"configurable": {"thread_id": "story_1"}}

    # 3. 第一轮：写个开头
    print("--- Round 1: 开头 ---")
    agent.invoke({"messages": [{"role": "user", "content": "写个关于小猫的故事，第一句"}]}, config)
    # 假设 AI 回复： "从前有一只小白猫。" (这是 Checkpoint A)

    # 4. 第二轮：写第二句
    print("--- Round 2: 续写 ---")
    agent.invoke({"messages": [{"role": "user", "content": "继续"}]}, config)
    # 假设 AI 回复： "它变成了一只狗。" (这是 Checkpoint B -> 逻辑崩了！)
    ```
  - 查找存档
    ```

```
# 获取历史记录（倒序的，最新的在最前）
all_history = list(agent.get_state_history(config))

# all_history[0] 是 "变成狗" (最新)
# all_history[1] 是 "用户说继续"
# all_history[2] 是 "小白猫" (我们要回到这里！)

previous_checkpoint_config = all_history[2].config
print(f"找到存档点 ID: {previous_checkpoint_config['configurable']
['checkpoint_id']}")
```

- 分叉运行

```
# 核心操作：update_state
# 注意：我们传入的是 previous_checkpoint_config (旧存档的配置)
agent.update_state(
    previous_checkpoint_config,
    {"messages": [{"role": "user", "content": "突然后面来了一只大灰狼"}]},
    # as_node="agent" # 假装这是 agent 刚说完的话，或者作为新输入
)

# 此时，LangGraph 并没有删除"变成狗"的记录。
# 而是创建了一个新分支！
# 现在的树：
# 小白猫 -> 变成狗 (旧分支，还在数据库里)
#       |
#       +-> 遇到大灰狼 (新分支，当前激活)

print("--- Round 3: 分叉后的续写 ---")

# 使用 None，表示"就接着刚才修改的状态继续跑"
# 注意：这里 config 还是原来的 thread_id，系统会自动识别最新的那个分支
agent.invoke(None, config)

# 预期结果：AI 会基于"小白猫" + "遇到大灰狼" 继续续写。
# 它完全忘记了"变成狗"这回事。
```

为什么我们需要Langgraph的记忆体

传统Langchain memory的局限性

- 传统的 Chain 是单向的（输入 -> LLM -> 输出）
- 难以精准修改中间的某一个参数或变量
- 如果 Agent 在第 5 步犯错，传统架构只能重头再来；无法回到第 4 步修改状态后继续。

LangGraph 的解决方案：状态机 (State Machine)

- 记忆不再是外挂的文本日志，而是图（Graph）中的全局变量（State）。
- 即使服务器重启，只要有 `thread_id`，Agent 就能从上一次中断的地方无缝继续

在项目中的主要应用

Core Value Proposition

- 为什么引入记忆体： 本质上是为了解决大模型在生产落地中的**"认知断层"**问题。
    - 模型变成了"有经验的助手"，它能记住用户的反馈，从错误中学习，并随着时间推移越来越懂用户的特定需求。

对于动态判别标准，Langgraph的主要应用

- 使用长期记忆
    - 当用户纠正模型："这张图不是全绿，是半绿"时，Agent 将这条反馈存入 Store 的 `("user_A", "preferences")` 命名空间。
    - 下次再分析图片时，Agent 先检索 Store 中的用户偏好，动态调整 Prompt："注意，用户 A 对黄色的敏感度很高，稍微发黄请判定为半枯黄。"

与其他Agent记忆体做对比

- **Controllable Workflow**： 当模型把"半绿"判成"全枯"时，我们需要利用 LangGraph 的 **Checkpointer** 暂停程序，人工修改 `state["leaf_status"]` 变量，然后让它继续运行。这种**"修改程序内部变量"**的能力，是其他只存"聊天记录"的框架做不到的。
- 虽然 MemGPT 或 OpenAI 都有记忆，但它们存的主要是**文本历史**，适合做陪聊

Human-in-the-loop 可以解决"主观标准不统一"的问题

- 在"最终结果存入数据库"节点前，设置 `interrupt_before` -> 确保不会有错误的分析数据污染核心数据库
- 审核员发现LLM评判标准与人类不一样，直接调用update_state改为正常，无需重跑耗时的推理模型
- 发现某一批次图片因意外参数设置错误导致全部分析失败，利用 Time Travel 回滚到"参数配置"节点，修正光照参数后分叉运行，挽救该批次任务

🆕 附录：与其他记忆方案的对比

特性	LangGraph	MemGPT	OpenAI Assistants	原生 LangChain
短期记忆	Checkpoint（状态快照）	对话上下文	Thread（对话历史）	ConversationBufferMemory
长期记忆	Store（语义搜索）	外部存储	文件/向量存储	需自行实现
状态可修改	✅ update_state	❌	❌	❌
时间旅行	✅ 分叉/回溯	❌	❌	❌
Human-in-the-loop	✅ 原生支持	有限	有限	需自行实现
记忆注入方式	消息列表 + System Prompt	自动管理	自动管理	Prompt 模板