第 6 章

智能体设计模式3：记忆系统

在智能体从“工具化响应”向“类人化协作”演进的过程中，记忆与规划能力的构建成为突破瓶颈的核心范式——这正是LangGraph框架的核心优势所在。

作为LangChain生态下专注复杂智能体开发的开源工具，LangGraph以图结构为骨架，通过结构化记忆管理与动态规划机制的深度耦合，让智能体摆脱了“单次交互即失忆”的局限，具备了连续认知与目标导向的执行能力。

其记忆系统借鉴人类记忆分层逻辑，构建了短期会话记忆与长期经验记忆的双重体系，既通过线程级持久化保障多轮对话的上下文连贯性，又依托跨线程存储与智能修剪策略实现经验的高效沉淀；而规划系统则借助节点与条件边组成的灵活图结构，支持任务的动态拆解、工具调用的有序协调及循环推理，让智能体在复杂场景中可自主规划执行路径。

这种“记忆为规划提供依据，规划引导记忆高效调用”的协同模式，成为新一代智能体设计的核心方法论。智能体的组成如图6-1所示。

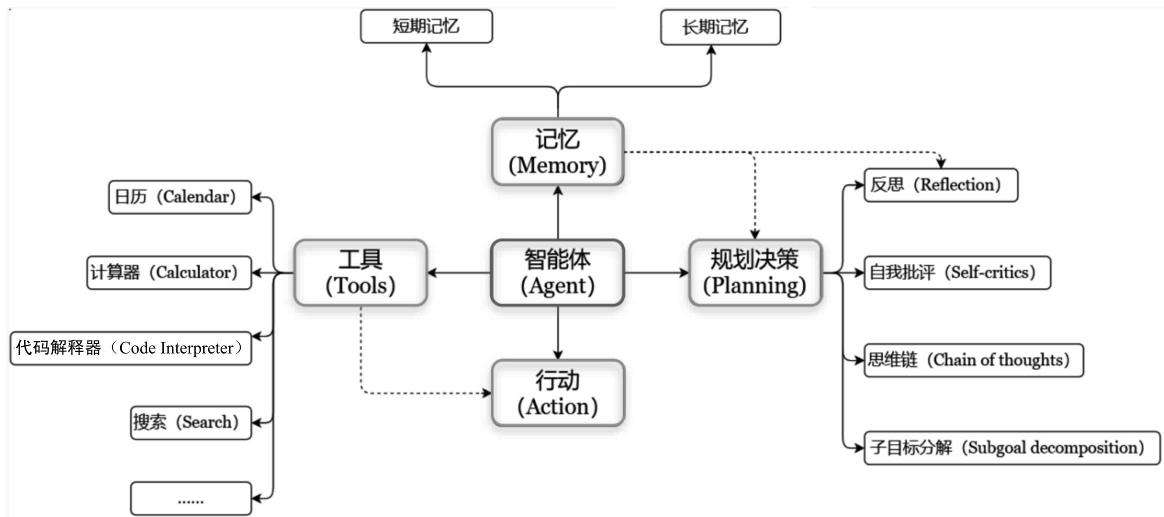

图 6-1　智能体的组成

6.1　LangGraph中的记忆存储

在LangGraph构建的AI智能体系统中，记忆模组是支撑智能体实现“类人交互”的核心组件，它打破了传统对话模型“单次交互即失忆”的局限，让智能体能够通过记录历史信息、沉淀经验知识，实现连贯的多轮对话与复杂任务处理。不同于简单的信息缓存，LangGraph的记忆模组是一套结构化的系统架构，其核心设计思路围绕智能体的交互场景需求，精准区分了短期记忆与长期记忆两大核心类型，确保信息的存储、调用与更新既高效又贴合实际应用场景。

短期记忆与长期记忆的协同工作构成了LangGraph智能体记忆能力的基础框架。短期记忆聚焦于当前会话的即时信息管理，如同智能体在单一对话线程中的“即时记事本”，实时跟踪交互过程中的上下文动态；而长期记忆则承担着跨场景、跨会话的信息沉淀功能，像是智能体的“永久知识库”，让分散对话中的关键信息得以留存和复用。这两种记忆类型并非孤立存在，而是通过LangGraph的结构化设计形成互补，共同为智能体的决策与响应提供全面的信息支撑。

作为智能体状态系统的重要组成部分，记忆模组的设计直接决定了智能体的交互连贯性与任务适应性。短期记忆通过与智能体状态深度绑定实现实时更新与持久化，确保当前对话的上下文不中断；长期记忆则借助灵活的命名空间与存储组件，打破对话线程的限制，让用户偏好、任务规则等核心信息能够跨场景调用。深入理解这套记忆架构的设计逻辑与实现方式，是掌握LangGraph智能体开发的关键环节。

6.1.1　基于内存的短期记忆

基于内存的短期记忆是LangGraph中短期记忆最基础、最常用的实现形式，其核心特征是将记忆数据直接存储在智能体运行进程的内存中，完全依赖当前会话的生命周期完成数据的创建、更新与销毁。这种设计模式天然契合短期记忆“即时性”与“临时性”的需求，无须依赖外部数据库等存储介质，从根本上保障了信息读取与写入的极低延迟，让智能体能够快速调用上下文信息响应用户需求。

首先，我们使用一个无记忆功能的智能体来帮助用户完成任务，代码如下：

```
from langchain_core.tools import tool

# 定义工具
@tool("get_weather_info")
def get_weather_info(city: str) -> str:
    """根据城市名称查询当前天气状况（含温度、天气状况）"""
    weather_data = {
        "上海": "城市：上海 | 天气：晴 | 温度：18℃ | 风向：东南风 | 湿度：65%",
        "南京": "城市：南京 | 天气：多云 | 温度：15℃ | 风向：北风 | 湿度：70%",
        "杭州": "城市：杭州 | 天气：小雨 | 温度：12℃ | 风向：西北风 | 湿度：80%",
    }
    return weather_data.get(city, f"未查询到{city}的天气信息")

@tool
def get_weather_tips(city: str) -> str:
    """根据城市名称返回天气相关生活建议（不包含温度和天气状况）"""
    tips_data = {
```

```
            "上海": "天气晴朗，适合户外活动；紫外线较强，注意防晒；早晚温差大",
            "南京": "多云天气，适宜出行；风力不大，适合晾晒衣物；空气较干燥",
            "杭州": "有小雨，出门记得带伞；路面湿滑，注意行车安全；气温较低，适当添衣",
        }
        return tips_data.get(city, f"暂无{city}的相关天气建议")

    tools = [get_weather_info,get_weather_tips]

    import bigmodel
    llm = bigmodel.llm

    from langchain.messages import AIMessage
    from langchain.agents.middleware.types import AgentMiddleware, AgentState,
ModelResponse

    from langchain.agents import create_agent
    agent = create_agent(
        model=llm,
        tools=tools,
        system_prompt="你是一个有用的智能体，可以借助工具帮助用户完成任务。",
    )

    def chatbot(state:bigmodel.BaseAgentState):
        messages = state.messages
        reply = agent.invoke({"messages": messages})
        reply_content = (reply["messages"][-1].content)
        return {"messages": reply_content}

    from langgraph.graph import StateGraph,add_messages
    graph_builder = StateGraph(bigmodel.BaseAgentState)
    graph_builder.add_node(chatbot)

    graph_builder.set_entry_point("chatbot")
    graph = graph_builder.compile()

    if __name__ == '__main__':
        inps = ["帮我查一下南京的天气","你能给什么建议吗？"]
        for inp in inps:
            reply = graph.invoke(input={"messages": inp})
            print(reply["messages"][-1].content)
            print("-"*50)
```

输出结果如下：

```
南京当前天气为多云，温度是15℃，风向为北风，湿度为70%。
--------------------------------------------------
请告诉我您所在的城市或您关心的城市名称，这样我才能为您提供相关的天气建议！
--------------------------------------------------
```

在上面的输出结果中，我们仅提供了对当前内容的查询，而没有加载前期的查询内容，即使当前的智能体运行在同一个状态空间State中。

这显然不是我们所需要的，想要解决这个问题，我们需要提供一个额外的内存空间。在LangGraph的开发框架中，基于内存的短期记忆通常与智能体的“状态（State）”紧密绑定，通过定义结构化的

状态类来承载记忆数据，即作为一个载体对当前状态空间的内容建立整体联系，从而使得智能体具有“短期记忆”。

例如，开发者可以在State中声明一个“对话历史”列表字段，每当智能体完成一次交互（接收用户输入、生成响应），就会通过状态更新逻辑将本次交互的关键信息（如用户提问内容、智能体回答要点、交互时间戳等）追加到该列表中。当智能体需要生成下一次响应时，只需从当前状态的“对话历史”字段中读取数据，即可快速掌握前序交互脉络，避免出现“上下文断裂”的问题。

下面是一个添加记忆功能到智能体的示例，代码如下：

```
...
from langgraph.checkpoint.memory import InMemorySaver
memory = InMemorySaver()
graph = graph_builder.compile(checkpointer=memory)
...
```

此时可以看到，我们在智能体图的构建过程中，主动传入了一个初始化的memory，这是基于内存的存储空间，供智能体上下文使用。完整代码如下：

```
from langchain_core.tools import tool

# 定义工具
@tool
def get_weather_info(city: str) -> str:
    """根据城市名称查询当前天气状况（含温度、天气状况）"""
    weather_data = {
        "上海": "城市：上海 | 天气：晴 | 温度：18℃ | 风向：东南风 | 湿度：65%",
        "南京": "城市：南京 | 天气：多云 | 温度：15℃ | 风向：北风 | 湿度：70%",
        "杭州": "城市：杭州 | 天气：小雨 | 温度：12℃ | 风向：西北风 | 湿度：80%",
    }
    return weather_data.get(city, f"未查询到{city}的天气信息")

@tool
def get_weather_tips(city: str) -> str:
    """根据城市名称返回天气相关生活建议（不包含温度和天气状况）"""
    tips_data = {
        "上海": "天气晴朗，适合户外活动；紫外线较强，注意防晒；早晚温差大",
        "南京": "多云天气，适宜出行；风力不大，适合晾晒衣物；空气较干燥",
        "杭州": "有小雨，出门记得带伞；路面湿滑，注意行车安全；气温较低，适当添衣",
    }
    return tips_data.get(city, f"暂无{city}的相关天气建议")

tools = [get_weather_info,get_weather_tips]

import bigmodel
llm = bigmodel.llm

from langchain.agents import create_agent
agent = create_agent(
    model=llm,
    tools=tools,
    system_prompt="你是一个有用的智能体，可以借助工具帮助用户完成任务。",
)

def chatbot(state:bigmodel.BaseAgentState):
    messages = state.messages
```

```
    reply = agent.invoke({"messages": messages})
    reply_content = (reply["messages"][-1].content)
    return {"messages": reply_content}

from langgraph.graph import StateGraph,add_messages
graph_builder = StateGraph(bigmodel.BaseAgentState)
graph_builder.add_node(chatbot)

graph_builder.set_entry_point("chatbot")

from langgraph.checkpoint.memory import InMemorySaver
memory = InMemorySaver()
graph = graph_builder.compile(checkpointer=memory)

if __name__ == '__main__':

    config = {
        "configurable": {
            "thread_id": "wx929"  # 每个用户/对话用唯一ID，多轮对话依赖此保存历史
        }
    }
    inps = ["帮我查一下南京的天气","你能给什么建议吗？"]
    for inp in inps:
        reply = graph.invoke(input={"messages": inp},config=config)
        print(reply["messages"][-1].content)
        print("-"*50)
```

输出结果如下：

```
南京当前天气为多云，温度是15℃，风向为北风，湿度为70%。
--------------------------------------------------
根据南京当前的天气情况，以下是一些建议：

- 多云天气适宜出行，可以安排户外活动。
- 风力不大，适合晾晒衣物。
- 空气较干燥，注意适当补水，保持皮肤湿润。
--------------------------------------------------
```

对比可以看到，我们通过在智能体的构建过程中显式地传入一个基于内存的存储空间，可以对当前对话进行保留，从而使得智能体具有短期的记忆功能。

可以看到，这种记忆形式的优势不仅体现在高效性上，还在于其极强的易用性和灵活性。对于快速原型开发、简单对话场景或轻量级智能体而言，基于内存的短期记忆无须额外配置存储服务，开发者仅需通过几行代码定义状态结构和记忆更新规则，就能实现基础的上下文管理功能。同时，其支持根据场景需求灵活定义记忆数据的格式——既可以存储完整的对话文本，也可以仅保留经过提炼的关键信息（如用户核心诉求、任务进度节点等），从而在“上下文完整性”与“内存占用效率”之间找到平衡。

6.1.2　基于硬存储的长期记忆存储

基于硬存储的长期记忆的核心目标是将人机交互过程中产生的每一次交流结果持久化保存，摆脱内存存储的临时性限制，实现对话数据的长期留存、追溯与复用。在具体使用上，我们可以直接对节

点进行操作，也可以使用after_model中间件作为模型推理流程的后置钩子（Hook）。钩子是实现这一目标的核心载体——它能在模型生成并返回响应后，自动触发数据处理与存储逻辑，确保每次交流结果以标准化的格式落地到硬存储介质（本地文件、数据库等）中。

本小节将依托智能体状态空间State，之后以JSON格式的形式对需要存储的记忆进行定义。

1. 状态空间State格式定义

在具体使用上，我们为保证数据的结构化和可解析性，需先定义统一的状态空间State，覆盖交流全量关键信息，简单的格式如下：

```
from datetime import datetime

from pydantic import BaseModel,Field
class ChatMemoryState(BaseModel):
    human_message: str = Field(default="",description="用户提出的问题和交流")
    AI_message:str = Field(default="",description="AI对当前问题进行回复的时间")
    datetime_now:datetime = (datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
```

2. 交流Agent的定义与使用示例

下面需要完成Agent的初始化和使用，仿照前面章节的内容，我们定义一个Agent的实例，通过传入工具完成用户的查询，示例如下：

```
from langchain.agents.middleware import before_model
from datetime import datetime

from pydantic import BaseModel,Field
class ChatMemoryState(BaseModel):
    human_message: str = Field(default="",description="用户提出的问题和交流")
    AI_message:str = Field(default="",description="AI对当前问题进行回复的时间")
    datetime_now:datetime = (datetime.now().strftime("%Y-%m-%d %H:%M:%S"))

from langchain_core.tools import tool

# 定义工具
@tool
def get_weather_info(city: str) -> str:
    """根据城市名称查询当前天气状况（含温度、天气状况）"""
    weather_data = {
        "上海": "城市：上海 | 天气：晴 | 温度：18℃ | 风向：东南风 | 湿度：65%",
        "南京": "城市：南京 | 天气：多云 | 温度：15℃ | 风向：北风 | 湿度：70%",
        "杭州": "城市：杭州 | 天气：小雨 | 温度：12℃ | 风向：西北风 | 湿度：80%",
    }
    return weather_data.get(city, f"未查询到{city}的天气信息")

@tool
def get_weather_tips(city: str) -> str:
    """根据城市名称返回天气相关生活建议（不包含温度和天气状况）"""
    tips_data = {
        "上海": "天气晴朗，适合户外活动；紫外线较强，注意防晒；早晚温差大",
        "南京": "多云天气，适宜出行；风力不大，适合晾晒衣物；空气较干燥",
        "杭州": "有小雨，出门记得带伞；路面湿滑，注意行车安全；气温较低，适当添衣",
```

```
        }
        return tips_data.get(city, f"暂无{city}的相关天气建议")

    tools = [get_weather_info,get_weather_tips]

    import bigmodel

    from langchain.agents import create_agent
    agent = create_agent(
        model=bigmodel.llm,
        tools=[get_weather_info, get_weather_tips],
        system_prompt="""你需要回答用户的提问，必要时调用工具完成任务。回答需清晰准确,并以JSON的
格式回复。""",
        response_format=ChatMemoryState,
    )

    # 调用智能体
    result = agent.invoke({"messages": "南京的天气是什么？"})
    print(result["structured_response"])
```

读者可以自行运行代码验证。

3. 核心after_model中间件函数

after_model函数作为模型响应后的钩子，接收用户输入、模型输出、会话时间数据等核心参数，依托这些参数，该函数可以先加载存储数据作为背景知识，再完成对后续内容的存储操作。代码如下：

```
    from langgraph.runtime import Runtime
    from langchain.agents.middleware import after_agent
    from pathlib import Path
    import json

    @after_agent
    def log_after_agent(state: ChatMemoryState | dict, runtime: Runtime):
        state = state["structured_response"]
        try:

            human_msg = state.human_message
            ai_msg = state.AI_message
            dt_now = state.datetime_now

            # 构造存储数据（确保datetime转为字符串）
            storage_data = {
                "human_message": human_msg,
                "AI_message": ai_msg,
                "datetime_now": dt_now.strftime("%Y-%m-%d %H:%M:%S") if
isinstance(dt_now, datetime) else dt_now
            }

            # 标准JSON数组存储（支持正常解析）
            storage_file = Path("./memory.json")
            if storage_file.exists():
                # 读取已有数据（JSON数组）
                with open(storage_file, "r", encoding="utf-8") as f:
```

```
                try:
                    memory_list = json.load(f)  # 加载已有数组
                except json.JSONDecodeError:
                    memory_list = []  # 若文件损坏，重置为空数组
        else:
            memory_list = []  # 新文件初始化数组

        # 追加新记录并写入（保持JSON数组格式）
        memory_list.append(storage_data)
        with open(storage_file, "w", encoding="utf-8") as f:
            json.dump(memory_list, f, ensure_ascii=False, indent=2)  # indent=2 格式
化显示，便于阅读

        print(f"✅ 交互记录已追加到 {storage_file}")
        print(f"📝 新增记录：{json.dumps(storage_data, ensure_ascii=False,
indent=2)}\n")

    except Exception as e:
        print(f"❌ 存储失败：{str(e)}")

    return
```

4. 可对交互内容进行存储的智能体完整实现

下面我们实现对实时交互内容进行完整存储的智能体，其核心目标是在响应用户需求的同时，通过after_agent中间件自动持久化交互记录，确保每一次用户提问与AI回复都以标准JSON格式存储，支持后续追溯、历史查询与长期复用。该实现兼顾了结构化响应、稳定存储与灵活扩展，适用于需要留存交互轨迹的场景（如客服对话、工具使用记录等）。

```
from langchain.agents.middleware import before_model
from datetime import datetime

from pydantic import BaseModel,Field
class ChatMemoryState(BaseModel):
    human_message: str = Field(default="",description="用户提出的问题和交流")
    AI_message:str = Field(default="",description="AI对当前问题进行回复")
    datetime_now:datetime = (datetime.now().strftime("%Y-%m-%d %H:%M:%S"))

from langgraph.runtime import Runtime
from langchain.agents.middleware import after_agent
from pathlib import Path
import json

@after_agent
def log_after_agent(state: ChatMemoryState | dict, runtime: Runtime):
    state = state["structured_response"]
    try:

        human_msg = state.human_message
        ai_msg = state.AI_message
        dt_now = state.datetime_now

        # 构造存储数据（确保datetime转为字符串）
```

```
            storage_data = {
                "human_message": human_msg,
                "AI_message": ai_msg,
                "datetime_now": dt_now.strftime("%Y-%m-%d %H:%M:%S") if
isinstance(dt_now, datetime) else dt_now
            }

            # 标准JSON数组存储（支持正常解析）
            storage_file = Path("./memory.json")
            if storage_file.exists():
                # 读取已有数据（JSON数组）
                with open(storage_file, "r", encoding="utf-8") as f:
                    try:
                        memory_list = json.load(f)  # 加载已有数组
                    except json.JSONDecodeError:
                        memory_list = []  # 若文件损坏，重置为空数组
            else:
                memory_list = []  # 新文件初始化数组

            # 追加新记录并写入（保持JSON数组格式）
            memory_list.append(storage_data)
            with open(storage_file, "w", encoding="utf-8") as f:
                json.dump(memory_list, f, ensure_ascii=False, indent=2)  # indent=2 格式
化显示，便于阅读

            print(f"✅ 交互记录已追加到 {storage_file}")
            print(f"📝 新增记录：{json.dumps(storage_data, ensure_ascii=False,
indent=2)}\n")

        except Exception as e:
            print(f"❌ 存储失败：{str(e)}")

        return

    from langchain_core.tools import tool
    # 定义工具
    @tool
    def get_weather_info(city: str) -> str:
        """根据城市名称查询当前天气状况（含温度、天气状况）"""
        weather_data = {
            "上海": "城市：上海 | 天气：晴 | 温度：18℃ | 风向：东南风 | 湿度：65%",
            "南京": "城市：南京 | 天气：多云 | 温度：15℃ | 风向：北风 | 湿度：70%",
            "杭州": "城市：杭州 | 天气：小雨 | 温度：12℃ | 风向：西北风 | 湿度：80%",
        }
        return weather_data.get(city, f"未查询到{city}的天气信息")

    @tool
    def get_weather_tips(city: str) -> str:
        """根据城市名称返回天气相关生活建议（不包含温度和天气状况）"""
        tips_data = {
            "上海": "天气晴朗，适合户外活动；紫外线较强，注意防晒；早晚温差大",
            "南京": "多云天气，适宜出行；风力不大，适合晾晒衣物；空气较干燥",
            "杭州": "有小雨，出门记得带伞；路面湿滑，注意行车安全；气温较低，适当添衣",
        }
```

```
        return tips_data.get(city, f"暂无{city}的相关天气建议")

    tools = [get_weather_info,get_weather_tips]

    import bigmodel

    from langchain.agents import create_agent
    agent = create_agent(
        model=bigmodel.llm,
        tools=[get_weather_info, get_weather_tips],
        system_prompt="""你需要回答用户的提问，必要时调用工具完成任务。回答需清晰准确，并以JSON
的格式回复。""",
        middleware=[log_after_agent],
        response_format=ChatMemoryState,
    )

    # 调用智能体
    result = agent.invoke({"messages": "南京的天气是什么？"})
    print(result["structured_response"])
```

此时可以看到，memory.json以标准JSON数组格式存储所有交互记录，可直接用记事本、JSON 编辑器打开，读者可以自行查阅。

对于有一定基础的读者，可以将本地文件替换为数据库（如MongoDB、SQLite、MySQL），只需修改中间件的读写逻辑，示例如下：

```
    # 替换为MongoDB存储（需安装pymongo）
    from pymongo import MongoClient
    client = MongoClient("mongodb://localhost:27017/")
    db = client["chat_memory_db"]
    collection = db["interaction_records"]

    # 中间件中替换文件读写为数据库插入
    collection.insert_one(new_record)
```

读者可以自行尝试完成。

6.2　LangGraph中的长期记忆载入与整理

长期记忆存储在使用中并非仅作为静态数据沉淀，预载入机制是其发挥价值的核心环节——当智能体启动或新会话开始时，系统会自动从硬存储介质（如本地memory.json文件或数据库）中读取目标用户的历史交互记录，将其转换为智能体可识别的上下文信息，完成长期记忆的“唤醒”。这一过程能让智能体在响应当前问题前，已掌握用户过往的提问偏好、核心需求甚至未完成的任务，例如用户此前曾查询南京天气，预载入后智能体可直接关联历史天气数据，无须用户重复说明场景。

预载入的实现需依托结构化的存储格式，前文定义的JSON数组记录可通过关键词匹配（如用户标识）或会话ID筛选，快速提取有效历史数据；同时需控制预载入的范围，避免过多冗余信息增加模型推理负担，通常优先载入最近一段时间或与当前问题主题相关的记忆片段，实现存储价值与交互效率的平衡。

6.2.1　基于硬存储的长期记忆预载入1：拼接提示词

长期记忆存储在使用中并非仅作为静态数据沉淀，预载入机制是其发挥价值的核心环节——当智能体启动或新会话开始时，系统会自动从硬存储介质（如本地memory.json文件或数据库）中读取目标用户的历史交互记录，将其转换为智能体可识别的上下文信息，完成长期记忆的“唤醒”。这一过程能让智能体在响应当前问题前，已掌握用户过往的提问偏好、核心需求甚至未完成的任务，例如用户此前曾查询南京天气，预载入后智能体可直接关联历史天气数据，无须用户重复说明场景。

预载入的实现需依托结构化的存储格式，前文定义的JSON数组记录可通过关键词匹配（如用户标识）或会话ID筛选，快速提取有效历史数据；同时需控制预载入的范围，避免过多冗余信息增加模型推理负担，通常优先载入最近一段时间或与当前问题主题相关的记忆片段，实现存储价值与交互效率的平衡。

对于记忆内容的处理，一个最简单的方案就是智能体在回复内容时，结合以往的所有信息内容进行回复。下面是作者实现的一个预载入的长期记忆，具体在使用时将硬存储中的内容从硬件上载入，通过拼接Propmt的形式将长期记忆进行载入，从而使得模型能够在使用时有所参考，代码如下：

```
    def get_history(file_path: str,max_list = 1024) -> str:
        """
        从指定JSON文件读取历史交互记录，转为智能体可识别的文本格式
        :param file_path: 历史记录存储文件路径
        :return: 格式化后的历史记录字符串（空字符串表示无记录）
        """
        storage_file = Path(file_path)
        # 处理文件不存在的情况
        if not storage_file.exists():
            return "暂无历史交互记录"
        try:
            # 读取JSON数组格式的历史记录
            with open(storage_file, "r", encoding="utf-8") as f:
                memory_list = json.load(f) if storage_file.stat().st_size > 0 else []
            # 控制预载入范围：仅取最近5条记录，避免冗余
            recent_history = memory_list[-max_list:] if len(memory_list) > max_list else
memory_list
            # 格式化历史记录（让大模型清晰识别用户与AI的对话逻辑）
            history_str = "\n".join([
                f"用户：{item['human_message']} | AI：{item['AI_message']}
({item['datetime_now']})"
                for item in recent_history
            ])
            return history_str if history_str else "暂无历史交互记录"
        except json.JSONDecodeError:
            # 处理文件损坏场景
            return "历史记录文件格式异常，无法读取"
        except Exception as e:
            return f"读取历史记录失败：{str(e)}"
```

这里作者实现了get_history函数作为长期记忆预载入的核心工具，其核心作用是从指定JSON存储文件中读取历史交互记录，并将其转换为智能体可清晰识别的文本格式，为智能体提供过往交互上下文支持。该函数包含两个参数：file_path用于指定历史记录的存储文件路径；max_list作为可选参数，

默认值为1024，用于控制预载入历史记录的最大条数，避免因记录过多增加模型推理负担。

函数执行时先将传入的文件路径转换为Path对象，若文件不存在，则直接返回“暂无历史交互记录”，确保流程不中断；若文件存在，则尝试读取内容，通过判断文件大小处理空文件场景——空文件时将记忆列表初始化为空，非空则用json.load解析为JSON数组格式的记忆列表。

为进一步优化性能，函数会根据max_list截取最近的历史记录，当已有记录数超过max_list时，取末尾最新的max_list条，否则保留全部记录，有效过滤冗余信息。随后函数将筛选后的每条记录按“用户：[提问内容] | AI：[回复内容]（[交互时间]）”的固定格式拼接，用换行符分隔多条记录，形成结构化的文本字符串，便于智能体快速识别对话逻辑；若格式化后无有效内容，则返回“暂无历史交互记录”。

完整的使用长期记忆进行交流的代码如下：

```
from langchain.agents.middleware import before_model
from datetime import datetime

from pydantic import BaseModel,Field
class ChatMemoryState(BaseModel):
    human_message: str = Field(default="",description="用户提出的问题和交流")
    AI_message:str = Field(default="",description="AI对当前问题进行回复")
    datetime_now:datetime = (datetime.now().strftime("%Y-%m-%d %H:%M:%S"))

from langgraph.runtime import Runtime
from langchain.agents.middleware import after_agent
from langchain.agents.middleware import before_agent,before_model

from pathlib import Path
import json
# -------------------------补充 log_after_agent 存储逻辑 -------------------------
@after_agent
def log_after_agent(state: ChatMemoryState | dict, runtime: Runtime):
    state = state["structured_response"]
    try:

        human_msg = state.human_message
        ai_msg = state.AI_message
        dt_now = state.datetime_now

        # 构造存储数据（确保datetime转为字符串）
        storage_data = {
            "human_message": human_msg,
            "AI_message": ai_msg,
            "datetime_now": dt_now.strftime("%Y-%m-%d %H:%M:%S") if 
isinstance(dt_now, datetime) else dt_now
        }

        # 标准JSON数组存储（支持正常解析）
        storage_file = Path("./memory.json")
        if storage_file.exists():
            # 读取已有数据（JSON数组）
            with open(storage_file, "r", encoding="utf-8") as f:
                try:
```

```
                    memory_list = json.load(f)  # 加载已有数组
                except json.JSONDecodeError:
                    memory_list = []  # 若文件损坏，重置为空数组
            else:
                memory_list = []  # 新文件初始化数组

            # 追加新记录并写入（保持JSON数组格式）
            memory_list.append(storage_data)
            with open(storage_file, "w", encoding="utf-8") as f:
                json.dump(memory_list, f, ensure_ascii=False, indent=2)  # indent=2 格式
化显示，便于阅读

            print(f"✅ 交互记录已追加到 {storage_file}")
            print(f"📝 新增记录：{json.dumps(storage_data, ensure_ascii=False,
indent=2)}\n")

        except Exception as e:
            print(f"❌ 存储失败：{str(e)}")

        return

    from langchain_core.tools import tool
    # 1. 定义工具
    @tool
    def get_weather_info(city: str) -> str:
        """根据城市名称查询当前天气状况（含温度、天气状况）"""
        weather_data = {
            "上海": "城市：上海 | 天气：晴 | 温度：18℃ | 风向：东南风 | 湿度：65%",
            "南京": "城市：南京 | 天气：多云 | 温度：15℃ | 风向：北风 | 湿度：70%",
            "杭州": "城市：杭州 | 天气：小雨 | 温度：12℃ | 风向：西北风 | 湿度：80%",
        }
        return weather_data.get(city, f"未查询到{city}的天气信息")

    @tool
    def get_weather_tips(city: str) -> str:
        """根据城市名称返回天气相关生活建议（不包含温度和天气状况）"""
        tips_data = {
            "上海": "天气晴朗，适合户外活动；紫外线较强，注意防晒；早晚温差大",
            "南京": "多云天气，适宜出行；风力不大，适合晾晒衣物；空气较干燥",
            "杭州": "有小雨，出门记得带伞；路面湿滑，注意行车安全；气温较低，适当添衣",
        }
        return tips_data.get(city, f"暂无{city}的相关天气建议")

    tools = [get_weather_info,get_weather_tips]


    # 2. 补全get_history函数：读取硬存储中的历史交互记录
    def get_history(file_path: str,max_list = 1024) -> str:
        """
        从指定JSON文件读取历史交互记录，转为智能体可识别的文本格式
        :param file_path: 历史记录存储文件路径
        :return: 格式化后的历史记录字符串（空字符串表示无记录）
        """
        storage_file = Path(file_path)
```

```
        # 处理文件不存在的情况
        if not storage_file.exists():
            return "暂无历史交互记录"
        try:
            # 读取JSON数组格式的历史记录
            with open(storage_file, "r", encoding="utf-8") as f:
                memory_list = json.load(f) if storage_file.stat().st_size > 0 else []
            # 控制预载入范围：仅取最近5条记录，避免冗余
            recent_history = memory_list[-max_list:] if len(memory_list) > max_list else
memory_list
            # 格式化历史记录（让大模型清晰识别用户与AI的对话逻辑）
            history_str = "\n".join([
                f"用户：{item['human_message']} | AI：{item['AI_message']}
({item['datetime_now']})"
                for item in recent_history
            ])
            return history_str if history_str else "暂无历史交互记录"
        except json.JSONDecodeError:
            # 处理文件损坏场景
            return "历史记录文件格式异常，无法读取"
        except Exception as e:
            return f"读取历史记录失败：{str(e)}"


    import bigmodel
    from langchain.agents import create_agent
    agent = create_agent(
        model=bigmodel.llm,
        tools=[get_weather_info, get_weather_tips],
        system_prompt=f"""你有关于用户的记忆{get_history('./memory.json')}，你需要回答用户
的提问，必要时调用工具完成任务。回答需清晰准确，并以JSON的格式回复。""",
        middleware=[log_after_agent],
        response_format=ChatMemoryState,
    )

    # 调用智能体
    result = agent.invoke({"messages": "我查询过哪里的天气，天气情况如何"})
    print(result["structured_response"])
```

读者可以自行运行代码验证。

6.2.2　基于硬存储的长期记忆预载入2：before_agent中间件的使用

前文我们通过直接拼接提示词的方式实现了长期记忆信息的预载入，但这种静态拼接的方式存在明显局限性（例如，记忆内容无法随交互动态更新、易因提示词冗余造成Token浪费、初始化后无法灵活调整记忆范围等）。

为此，我们可以借助LangChain提供的before_agent中间件来优化这一过程——该中间件能够在智能体执行核心推理逻辑前，动态完成硬存储中历史记忆的读取、筛选与上下文注入，让记忆载入更灵活、高效。

接下来，我们将具体展示如何通过before_agent中间件完成长期记忆的动态预载入。实现代码如下：

```
@before_agent
def inject_history_before_agent(state: dict, runtime: Runtime) -> dict:
    """
    在智能体执行前动态读取并注入历史交互记录到上下文
    修复：适配LangChain消息对象（而非字典）的处理逻辑
    """
    try:
        # 读取历史记录（限制最近5条，避免上下文过长）
        history_str = get_history("./memory.json", max_list=1024)

        # 构建包含历史记录的系统提示词
        base_system_prompt = """你需要结合用户的历史交互记录回答问题，必要时调用工具完成任务。
            回答需清晰准确，并严格以JSON格式回复。

            【用户历史交互记录】：
            {history}"""

        # 替换占位符，生成最终系统提示词
        updated_system_prompt = base_system_prompt.format(history=history_str)

        # 适配不同版本的LangChain Agent状态结构（核心修复点）
        if "system_prompt" in state:
            # 方式1：直接更新system_prompt字段
            state["system_prompt"] = updated_system_prompt
        elif "messages" in state:
            # 方式2：处理LangChain消息对象（而非字典）
            # 过滤掉已有的SystemMessage，保留用户/AI消息
            new_messages = [msg for msg in state["messages"] if not isinstance(msg, 
SystemMessage)]
            # 插入新的SystemMessage（对象形式，而非字典）
            new_messages.insert(0, SystemMessage(content=updated_system_prompt))
            state["messages"] = new_messages

        print(f"✅ 成功注入历史记录：\n{history_str}\n")
    except Exception as e:
        print(f"❌ 注入历史记录失败：{str(e)}")
    return state

# -------------------------- 历史记录读取函数--------------------------
def get_history(file_path: str, max_list: int = 5) -> str:
    """
    从指定JSON文件读取历史交互记录，转换为智能体可识别的文本格式
    """
    storage_file = Path(file_path)
    if not storage_file.exists():
        return "暂无历史交互记录"

    try:
        with open(storage_file, "r", encoding="utf-8") as f:
            memory_list = json.load(f) if storage_file.stat().st_size > 0 else []

        # 仅取最近max_list条记录
```

```
            recent_history = memory_list[-max_list:] if len(memory_list) > max_list else
memory_list
            if not recent_history:
                return "暂无历史交互记录"

            # 格式化历史记录（让大模型清晰识别对话逻辑）
            history_str = "\n".join([
                f"[{item['datetime_now']}] 用户：{item['human_message']}\n  AI：
{item['AI_message']}"
                for item in recent_history
            ])

            return history_str
        except json.JSONDecodeError:
            return "历史记录文件格式异常，无法读取"
        except Exception as e:
            return f"读取历史记录失败：{str(e)}"
```

上面两段代码共同实现了在智能体执行核心推理逻辑前，从本地JSON硬存储文件中动态读取并注入历史交互记录的核心功能。其中，inject_history_before_agent作为before_agent中间件承担调度与上下文注入的核心角色，get_history则负责具体的历史记录读取与格式化工作。

被@before_agent装饰器标记的inject_history_before_agent函数是在智能体执行前触发的中间件，它接收智能体执行前的状态字典state和LangGraph运行时对象runtime作为入参，最终返回修改后的状态字典。

函数内部通过try-except异常处理块保障执行稳定性，首先调用get_history函数读取./memory.json路径下的历史交互记录，并限制最多读取1024条（可通过参数调整），避免因历史记录过长导致上下文冗余。随后构建包含历史记录占位符的基础系统提示词模板，将读取到的格式化历史记录填充到模板中，生成包含历史信息的完整系统提示词。

为了适配不同版本LangChain Agent的状态存储结构，函数做了双重兼容处理：若状态字典中存在system_prompt字段，直接将该字段更新为新的系统提示词；若状态字典以messages字段存储对话消息，则先过滤掉已有的SystemMessage类型消息，再将新的系统提示词以SystemMessage对象的形式插入消息列表首位，这一处理修正了此前将LangChain原生消息对象当作普通字典操作导致的属性错误问题。函数执行成功时会打印注入的历史记录日志，若出现异常，则捕获并输出具体错误信息，确保中间件的执行不会中断智能体的整体流程。

作者创建了一个记录日常回复的文件memory.json，基于此历史记录，完成查询的历史代码如下：

```
from langchain.agents.middleware import before_model, before_agent, after_agent
from datetime import datetime
from pydantic import BaseModel, Field
from langgraph.runtime import Runtime
from pathlib import Path
import json
from langchain_core.tools import tool
# 新增：导入LangChain消息对象类型，用于判断消息角色
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage
import bigmodel  # 确保你的bigmodel模块已正确定义LLM
from langchain.agents import create_agent
```

```
# --------------------------- 1. 修正数据模型定义 ---------------------------
class ChatMemoryState(BaseModel):
    human_message: str = Field(default="", description="用户提出的问题和交流")
    AI_message: str = Field(default="", description="AI对当前问题进行回复")
    # 修正：字段类型与赋值统一为字符串，使用default_factory动态生成当前时间
    datetime_now: str = Field(
        default_factory=lambda: datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        description="交互时间"
    )

# ---------------------- 2. before_agent中间件：注入历史记录 ----------------------
@before_agent
def inject_history_before_agent(state: dict, runtime: Runtime) -> dict:
    """
    在智能体执行前动态读取并注入历史交互记录到上下文
    修复：适配LangChain消息对象（而非字典）的处理逻辑
    """
    try:
        # 读取历史记录（限制最近5条，避免上下文过长）
        history_str = get_history("./memory.json", max_list=1024)

        # 构建包含历史记录的系统提示词
        base_system_prompt = """你需要结合用户的历史交互记录回答问题，必要时调用工具完成任务。
            回答需清晰准确，并严格以JSON格式回复。

            【用户历史交互记录】：
            {history}"""

        # 替换占位符，生成最终系统提示词
        updated_system_prompt = base_system_prompt.format(history=history_str)

        # 适配不同版本的LangChain Agent状态结构（核心修复点）
        if "system_prompt" in state:
            # 方式1：直接更新system_prompt字段
            state["system_prompt"] = updated_system_prompt
        elif "messages" in state:
            # 方式2：处理LangChain消息对象（而非字典）
            # 过滤掉已有的SystemMessage，保留用户/AI消息
            new_messages = [msg for msg in state["messages"] if not isinstance(msg, 
SystemMessage)]
            # 插入新的SystemMessage（对象形式，而非字典）
            new_messages.insert(0, SystemMessage(content=updated_system_prompt))
            state["messages"] = new_messages

        print(f"✅ 成功注入历史记录：\n{history_str}\n")
    except Exception as e:
        print(f"❌ 注入历史记录失败：{str(e)}")
    return state

# -------------------------------- 3. 工具定义 ----------------------------
@tool
def get_weather_info(city: str) -> str:
    """根据城市名称查询当前天气状况（含温度、天气状况）"""
    weather_data = {
```

```
            "上海": "城市：上海 | 天气：晴 | 温度：18℃ | 风向：东南风 | 湿度：65%",
            "南京": "城市：南京 | 天气：多云 | 温度：15℃ | 风向：北风 | 湿度：70%",
            "杭州": "城市：杭州 | 天气：小雨 | 温度：12℃ | 风向：西北风 | 湿度：80%",
        }
        return weather_data.get(city, f"未查询到{city}的天气信息")

    @tool
    def get_weather_tips(city: str) -> str:
        """根据城市名称返回天气相关生活建议（不包含温度和天气状况）"""
        tips_data = {
            "上海": "天气晴朗，适合户外活动；紫外线较强，注意防晒；早晚温差大",
            "南京": "多云天气，适宜出行；风力不大，适合晾晒衣物；空气较干燥",
            "杭州": "有小雨，出门记得带伞；路面湿滑，注意行车安全；气温较低，适当添衣",
        }
        return tips_data.get(city, f"暂无{city}的相关天气建议")

    tools = [get_weather_info, get_weather_tips]

    # -------------------------- 4. 历史记录读取函数--------------------------
    def get_history(file_path: str, max_list: int = 5) -> str:
        """
        从指定JSON文件读取历史交互记录，转为智能体可识别的文本格式
        """
        storage_file = Path(file_path)
        if not storage_file.exists():
            return "暂无历史交互记录"

        try:
            with open(storage_file, "r", encoding="utf-8") as f:
                memory_list = json.load(f) if storage_file.stat().st_size > 0 else []

            # 仅取最近max_list条记录
            recent_history = memory_list[-max_list:] if len(memory_list) > max_list else
memory_list
            if not recent_history:
                return "暂无历史交互记录"

            # 格式化历史记录（让大模型清晰识别对话逻辑）
            history_str = "\n".join([
                f"[{item['datetime_now']}] 用户：{item['human_message']}\n  AI：
{item['AI_message']}"
                for item in recent_history
            ])

            return history_str
        except json.JSONDecodeError:
            return "历史记录文件格式异常，无法读取"
        except Exception as e:
            return f"读取历史记录失败：{str(e)}"

    # -------------------------- 5. 创建智能体（注入完整中间件）--------------------------
    agent = create_agent(
        model=bigmodel.llm,
        tools=tools,
```

```
        # 基础系统提示词（历史记录由before_agent动态注入）
        system_prompt="你需要回答用户的提问，必要时调用工具完成任务。回答需清晰准确，并以JSON的格式回复。",
        # 中间件顺序：先注入历史 → 执行Agent → 保存记录
        middleware=[inject_history_before_agent],
        response_format=ChatMemoryState,
    )

    # -------------------------- 6. 测试调用--------------------------
    if __name__ == "__main__":

        result = agent.invoke({"messages": "我之前问过防流感食谱还记得吗？是怎么做的？什么时间问的？"})
        print("调用结果：", result["structured_response"])
```

输出结果如下：

```
    ...
    调用结果： human_message='上次说的防流感食谱还记得吗？' AI_message='11月18日推荐「葱白姜枣汤」：葱白5段+生姜3片+红枣6颗，煮沸10分钟。南京今日气温适宜饮用。' datetime_now='2025-11-20 10:44:33'
```

从上面的结果中可以看到，通过调用硬存储中的历史交互记录，我们在回应用户问题时，能更精准地追溯过往对话信息，从而避免用户重复表述已知需求，让回答更具连贯性和针对性。这种依托before_agent中间件的动态注入方式，无须在智能体初始化时静态拼接提示词，既能根据实际需求灵活控制历史记忆的加载范围，又能确保每次交互都能调用最新的存储数据，让智能体的“长期记忆” 更高效、更贴合用户实时场景，显著提升对话的自然度和实用性。

6.2.3　基于硬存储的长期记忆预载入3：before_model与检索整理

前面我们在进行历史记忆注入时，查询并打印新生成的State，除了对结果的回复外，还可以看到如下内容：

```
    {'messages':[SystemMessage(content='你需要结合用户的历史交互记录回答问题，必要时调用工具完成任务。\n            回答需清晰准确，并严格以JSON格式回复。\n            \n            【用户历史交互记录】：\n            [2025-11-20 09:30:15] 用户：今天南京地铁2号线末班车几点？\n
    ...
    ...
    第二次调用结果： human_message='上次说的防流感食谱还记得吗？' AI_message='11月18日推荐「葱白姜枣汤」：葱白5段+生姜3片+红枣6颗，煮沸10分钟。南京今日气温适宜饮用。' datetime_now='2025-11-20 10:44:33'
```

之所以采用这种方式呈现，核心是在智能体（Agent）启动前，就将所有历史记忆一次性输入其中，让智能体在生成回复前，能从完整记忆库中检索关联信息，进而输出更精准的最优回复。这样做的好处在于无须在交互过程中额外触发记忆读取流程，能减少实时计算开销，同时确保智能体掌握完整的历史上下文，避免关键对话信息的遗漏，让回复更具连贯性。

但由于实际场景中历史记录往往会随交互不断累积，长度变长。若使用过程中历史数据过多，则会导致输入模型的提示词体积急剧膨胀，不仅可能超出大模型的上下文窗口限制，还会大幅消耗Token资源，拖慢智能体的推理响应速度，甚至因冗余信息过多干扰模型对核心问题的判断，反而影响回复的精准度。

下面我们修改原有的代码，期望能够在模型进行下一步处理之前，只包含最相关的几条内容，而不加载不相关的内容。

BM25（Best Matching 25）是经典的文本相关性检索算法，作为TF-IDF的改进版，核心作用是量化“查询文本”与“文档集合”中每个文档的相关性，通过得分排序快速筛选出最匹配的内容，在对话历史匹配等场景中尤为实用。相比“时间最近”“全量加载”这类粗放方式，它能基于关键词的语义和频次特征，精准定位与用户当前提问最相关的历史记录，既减少大模型的上下文长度、降低Token消耗，又能避免长文档或高频无意义词的干扰，让模型回复更精准——这得益于它引入的“词频饱和”和“文档长度归一化”机制，解决了TF-IDF线性加权的固有缺陷。

具体来看，假如我们有一系列的文档Doc，现在要查询问题Query。BM25的思想是，对Query进行语素解析，生成语素Q；然后对于每个搜索文档 D_i 计算每个语素 Q_i 与文档 D_j 的相关性，最后将所有的语素 Q_i 与 D_j 进行加权求和，从而最终计算出Query与 D_j 的相似性得分。将BM25算法总结如下：

$$\text{Score}(\text{Query}, D_i) = \sum_{i}^{n} W_i \cdot R(Q_i, D_j)$$

在中文中，我们通常将每一个词语当作 Q_i，W_i 表示语素 Q_i 的权重，$R(Q_i, D_j)$ 表示语素 Q_i 与文档 D_i 的相关性得分关系。

BM25的核心是一套相关性得分计算公式，虽然无须死记硬背，但理解关键逻辑能更好地应用：公式会结合单个分词在文档中的出现次数（词频TF）、分词的稀缺程度（逆文档频率IDF，比如“防流感”比“的”更有区分度），再通过$k1$（调节词频饱和度，通常为1.2～2.0，避免高频词过度加权）、b（调节文档长度影响，通常为0.75，平衡长短期文档权重）等参数，综合计算出每条文档与查询的相关性得分，得分越高，则匹配度越强。

在中文场景落地时，BM25需要针对性处理：首先要进行数据清洗，过滤历史列表中的空字符串、全空格等无效数据，同时清理用户提问中的冗余信息；由于BM25原生适配英文，中文需用jieba等工具分词，将“防流感食谱怎么做”拆分为["防流感","食谱","怎么","做"] 这类词汇列表，还可以选择过滤“的、了、吗”等无意义停用词，减少噪声干扰。之后将所有有效历史记录的分词结果组成语料库，初始化BM25Okapi（最常用的变体），模型会自动计算分词IDF值、文档平均长度等关键数据；接着对用户当前提问分词，调用模型计算每条历史记录的相关性得分，最后将得分与历史记录绑定、按得分降序排序，过滤掉得分为0的无匹配内容，按需选取前N条结果即可。

下面我们将实现使用BM算法完成相似度查找，需要安装两个库：

```
pip install rank_bm25 jieba
```

而结合了BM25查找算法的历史记录获取代码如下：

```
def inject_history_before_agent(state: dict, runtime: Runtime) -> dict:
    """
    在智能体执行前动态读取并注入历史交互记录到上下文
    修复：适配LangChain消息对象（而非字典）的处理逻辑
    """
    try:
        human_message = state["messages"][-1].content

        # 读取历史记录（限制最近5条，避免上下文过长）
        history_str = get_history("./memory.json", max_list=1024)
```

```
            history_list = history_str.split("[")
            topk_result = bm25_retrieve_top3(human_message,history_list)
            print(topk_result)
            print("------------")
            # 构建包含历史记录的系统提示词
            base_system_prompt = """你需要结合用户的历史交互记录回答问题，必要时调用工具完成任务。
                回答需清晰准确，并严格以JSON格式回复。

                【用户历史交互记录】：
                {history}"""

            # 替换占位符，生成最终系统提示词
            updated_system_prompt = base_system_prompt.format(history=history_str)

            # 适配不同版本的LangChain Agent状态结构（核心修复点）
            if "system_prompt" in state:
                # 方式1：直接更新system_prompt字段
                state["system_prompt"] = updated_system_prompt
            elif "messages" in state:
                # 方式2：处理LangChain消息对象（而非字典）
                # 过滤掉已有的SystemMessage，保留用户/AI消息
                new_messages = [msg for msg in state["messages"] if not isinstance(msg, 
SystemMessage)]
                # 插入新的SystemMessage（对象形式，而非字典）
                new_messages.insert(0, SystemMessage(content=updated_system_prompt))
                state["messages"] = new_messages

        except Exception as e:
            print(f"❌ 注入历史记录失败：{str(e)}")
        return state


    from rank_bm25 import BM25Okapi
    import jieba
    import warnings


    def bm25_retrieve_top3(
            human_message: str,
            history_list: list[str],
            top_k: int = 3
    ) -> list[str]:
        """
        使用BM25算法从字符串列表中检索与用户提问最相似的top3内容
        :param human_message: 用户当前提问（待匹配的字符串）
        :param history_list: 历史字符串列表（待检索的数据源）
        :param top_k: 检索结果数量，默认3条
        :return: 与用户提问最相似的top3字符串列表（按相似度降序排列，无匹配则返回空列表）
        """
        # ------------- 1. 输入校验与预处理 -------------
        # 过滤历史列表中的空字符串/全空格字符串，避免无效数据
        valid_history = [h.strip() for h in history_list if isinstance(h, str) and 
h.strip()]
        if not valid_history:
```

```
            return []  # 无有效历史数据，返回空列表

        # 过滤用户提问的空值
        query = human_message.strip()
        if not query:
            return []  # 用户提问为空，返回空列表

        # ------------- 2.中文分词（BM25适配中文核心步骤） -------------
        # 对历史列表每个字符串分词，构建BM25语料库
        corpus_seg = [jieba.lcut(history) for history in valid_history]
        # 对用户提问分词，作为检索词
        query_seg = jieba.lcut(query)

        # ------------- 3. BM25相似度计算与排序 -------------
        # 初始化BM25模型
        bm25 = BM25Okapi(corpus_seg)
        # 计算每个历史字符串与用户提问的相似度得分
        scores = bm25.get_scores(query_seg)

        # ------------- 4. 筛选top3相似结果（过滤相似度为0的无效匹配） -------------
        # 构建“得分-索引-原始字符串”的三元组，便于排序
        score_with_history = list(zip(scores, range(len(valid_history)),
valid_history))
        # 按相似度得分降序排序
        score_with_history_sorted = sorted(score_with_history, key=lambda x: x[0],
reverse=True)
        # 筛选得分大于0的结果，取前top_k条
        top3_results = [
            item[2] for item in score_with_history_sorted
            if item[0] > 0  # 过滤无相似度的内容
        ][:top_k]

        return top3_results

    # -------------------------- 5. 历史记录读取函数--------------------------
    def get_history(file_path: str, max_list: int = 5) -> str:
        """
        从指定JSON文件读取历史交互记录，转为智能体可识别的文本格式
        """
        storage_file = Path(file_path)
        if not storage_file.exists():
            return "暂无历史交互记录"

        try:
            with open(storage_file, "r", encoding="utf-8") as f:
                memory_list = json.load(f) if storage_file.stat().st_size > 0 else []

            # 仅取最近max_list条记录
            recent_history = memory_list[-max_list:] if len(memory_list) > max_list else
memory_list
            if not recent_history:
                return "暂无历史交互记录"

            # 格式化历史记录（让大模型清晰识别对话逻辑）
```

```
            history_str = "\n".join([
                f"[{item['datetime_now']}] 用户：{item['human_message']}\n  AI：
{item['AI_message']}"
                for item in recent_history
            ])

            return history_str
        except json.JSONDecodeError:
            return "历史记录文件格式异常，无法读取"
        except Exception as e:
            return f"读取历史记录失败：{str(e)}"
```

可以看到，这三段代码共同构成了智能体执行前的历史记录检索与注入逻辑，核心是通过BM25算法从本地JSON存储的历史交互记录中精准匹配与用户当前提问最相关的内容，并将其动态注入智能体的上下文，替代了传统全量或按时间加载历史的方式，大幅提升智能体回复的精准度。

这里我们主要介绍BM25算法，其中bm25_retrieve_top3是代码中实现精准检索的核心工具函数，专门用于从历史字符串列表中筛选出与用户提问最相似的3条内容。函数首先对输入进行严格校验与预处理，过滤历史列表中的空字符串、全空格字符串以及非字符串类型数据，得到有效的历史记录列表，同时过滤用户提问中的空值，若任一输入无效，则直接返回空列表。由于BM25算法原生适配英文，函数针对中文场景做了关键处理——使用jieba分词工具将每条有效历史记录和用户当前提问拆分为独立词汇，分别构建BM25检索所需的语料库和检索词列表。

随后初始化BM25Okapi模型，传入分词后的历史语料库，模型会自动计算词汇的逆文档频率、文档平均长度等关键参数，接着调用模型的get_scores方法计算每条历史记录与用户提问的相似度得分。为筛选出最相关的内容，函数将得分、历史记录索引和原始字符串绑定为三元组，按得分降序排序后，过滤掉得分为0的无匹配内容，最终返回前3条有效结果，确保返回的内容都是与用户提问真正相关的。

完整代码如下：

```
from langchain.agents.middleware import before_model, before_agent, after_agent
from datetime import datetime
from pydantic import BaseModel, Field
from langgraph.runtime import Runtime
from pathlib import Path
import json
from langchain_core.tools import tool
# 新增：导入LangChain消息对象类型，用于判断消息角色
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage
import bigmodel  # 确保你的bigmodel模块已正确定义LLM
from langchain.agents import create_agent

# -------------------------- 1. 修正数据模型定义 --------------------------
class ChatMemoryState(BaseModel):
    human_message: str = Field(default="", description="用户提出的问题和交流")
    AI_message: str = Field(default="", description="AI对当前问题进行回复")
    # 修正：字段类型与赋值统一为字符串，使用default_factory动态生成当前时间
    datetime_now: str = Field(
        default_factory=lambda: datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        description="交互时间"
    )
```

```
# ---------------------- 2. before_agent中间件：注入历史记录 ----------------------
@before_model
def inject_history_before_agent(state: dict, runtime: Runtime) -> dict:
    """
    在智能体执行前动态读取并注入历史交互记录到上下文
    修复：适配LangChain消息对象（而非字典）的处理逻辑
    """
    try:
        human_message = state["messages"][-1].content

        # 读取历史记录（限制最近5条，避免上下文过长）
        history_str = get_history("./memory.json", max_list=1024)
        history_list = history_str.split("[")
        topk_result = bm25_retrieve_top3(human_message,history_list)
        print(topk_result)
        print("------------")
        # 构建包含历史记录的系统提示词
        base_system_prompt = """你需要结合用户的历史交互记录回答问题，必要时调用工具完成任务。
            回答需清晰准确，并严格以JSON格式回复。

            【用户历史交互记录】：
            {history}"""

        # 替换占位符，生成最终系统提示词
        updated_system_prompt = base_system_prompt.format(history=history_str)

        # 适配不同版本的LangChain Agent状态结构（核心修复点）
        if "system_prompt" in state:
            # 方式1：直接更新system_prompt字段
            state["system_prompt"] = updated_system_prompt
        elif "messages" in state:
            # 方式2：处理LangChain消息对象（而非字典）
            # 过滤掉已有的SystemMessage，保留用户/AI消息
            new_messages = [msg for msg in state["messages"] if not isinstance(msg, 
SystemMessage)]
            # 插入新的SystemMessage（对象形式，而非字典）
            new_messages.insert(0, SystemMessage(content=updated_system_prompt))
            state["messages"] = new_messages

    except Exception as e:
        print(f"✗ 注入历史记录失败：{str(e)}")
    return state

from rank_bm25 import BM25Okapi
import jieba
import warnings

def bm25_retrieve_top3(
        human_message: str,
        history_list: list[str],
        top_k: int = 3
) -> list[str]:
    """
    使用BM25算法从字符串列表中检索与用户提问最相似的top3内容
    :param human_message: 用户当前提问（待匹配的字符串）
    :param history_list: 历史字符串列表（待检索的数据源）
```

```
        :param top_k: 检索结果数量，默认3条
        :return: 与用户提问最相似的top3字符串列表（按相似度降序排列，无匹配则返回空列表）
        """
        # ------------- 1. 输入校验与预处理 -------------
        # 过滤历史列表中的空字符串/全空格字符串，避免无效数据
        valid_history = [h.strip() for h in history_list if isinstance(h, str) and
h.strip()]
        if not valid_history:
            return []  # 无有效历史数据，返回空列表

        # 过滤用户提问的空值
        query = human_message.strip()
        if not query:
            return []  # 用户提问为空，返回空列表

        # ------------- 2.中文分词（BM25适配中文核心步骤） -------------
        # 对历史列表每个字符串分词，构建BM25语料库
        corpus_seg = [jieba.lcut(history) for history in valid_history]
        # 对用户提问分词，作为检索词
        query_seg = jieba.lcut(query)

        # ------------- 3. BM25相似度计算与排序 -------------
        # 初始化BM25模型
        bm25 = BM25Okapi(corpus_seg)
        # 计算每个历史字符串与用户提问的相似度得分
        scores = bm25.get_scores(query_seg)

        # ------------- 4. 筛选top3相似结果（过滤相似度为0的无效匹配） -------------
        # 构建“得分-索引-原始字符串”的三元组，便于排序
        score_with_history = list(zip(scores, range(len(valid_history)),
valid_history))
        # 按相似度得分降序排序
        score_with_history_sorted = sorted(score_with_history, key=lambda x: x[0],
reverse=True)
        # 筛选得分大于0的结果，取前top_k条
        top3_results = [
            item[2] for item in score_with_history_sorted
            if item[0] > 0  # 过滤无相似度的内容
        ][:top_k]

        return top3_results


    # -------------------------- 5. 历史记录读取函数--------------------------
    def get_history(file_path: str, max_list: int = 5) -> str:
        """
        从指定JSON文件读取历史交互记录，转为智能体可识别的文本格式
        """
        storage_file = Path(file_path)
        if not storage_file.exists():
            return "暂无历史交互记录"

        try:
            with open(storage_file, "r", encoding="utf-8") as f:
```

```
                memory_list = json.load(f) if storage_file.stat().st_size > 0 else []

            # 仅取最近max_list条记录
            recent_history = memory_list[-max_list:] if len(memory_list) > max_list else
memory_list
            if not recent_history:
                return "暂无历史交互记录"

            # 格式化历史记录（让大模型清晰识别对话逻辑）
            history_str = "\n".join([
                f"[{item['datetime_now']}] 用户：{item['human_message']}\n  AI：
{item['AI_message']}"
                for item in recent_history
            ])

            return history_str
        except json.JSONDecodeError:
            return "历史记录文件格式异常，无法读取"
        except Exception as e:
            return f"读取历史记录失败：{str(e)}"

    # -------------------------- 6. 创建智能体（注入完整中间件） --------------------------
    agent = create_agent(
        model=bigmodel.llm,
        tools=None,
        # 基础系统提示词（历史记录由before_agent动态注入）
        system_prompt="你需要回答用户的提问，必要时调用工具完成任务。回答需清晰准确，并以JSON的格
式回复。",
        # 中间件顺序：注入历史 → 执行Agent → 保存记录
        middleware=[inject_history_before_agent],
        response_format=ChatMemoryState,
    )

    # -------------------------- 7. 测试调用--------------------------
    if __name__ == "__main__":
        result = agent.invoke({"messages": "我之前问过防流感食谱还记得吗？是怎么做的？什么时
间问的？"})
        print("调用结果：", result["structured_response"])
```

运行结果如下：

```
    ['] 用户：我之前问过防流感食谱还记得吗？是怎么做的？什么时间问的？\n  AI：', '2025-11-20
10:44:33] 用户：上次说的防流感食谱还记得吗？\n  AI：11月18日推荐「葱白姜枣汤」：葱白5段+生姜3片+红
枣6颗，煮沸10分钟。南京今日气温适宜饮用。', '2025-11-20 10:26:44] 用户：昨天查的电影场次是什么时间？
\n  AI：11月19日18:30您查询了「IMAX厅《海洋奇缘2》」：19:20/21:45两场，新街口德基广场影院。']
    ------------
    调用结果： human_message='我之前问过防流感食谱还记得吗？是怎么做的？什么时间问的？'
AI_message='您于2025年11月20日10:44:33询问过防流感食谱，当时推荐的是「葱白姜枣汤」，做法是：取葱
白5段、生姜3片、红枣6颗，加水煮沸10分钟即可。该食谱具有驱寒暖身、增强免疫力的作用，适合当前南京气温
饮用。' datetime_now='2025-11-20 12:26:00'
```

这里作者额外打印了智能体在大模型回复之前在运行过程中传入的信息，从中可以看到，相对于前期我们将所有的记忆体内容一次性传入，此次传入的内容仅仅保留与我们的提问最相关的部分，从而实现在解答时提升效率。

6.2.4 before_agent与before_model的使用差异

before_agent与before_model的核心区别聚焦于触发时机与作用场景，二者均为LangChain智能体的中间件装饰器，但钩子（使用位置）节点和功能定位完全不同。

- 触发时机差异：before_agent是智能体整个流程启动前的前置钩子，在智能体初始化、接收请求后立即执行；before_model是模型推理调用前的前置钩子，仅在智能体决定调用大模型生成响应时触发（晚于before_agent，且可能在工具调用后执行）。
- 作用场景差异：before_agent 侧重智能体全局初始化，比如结合前文的长期记忆预载入——在智能体启动时读取memory.json中的历史记录，初始化会话状态；还可用于注册额外工具、校验输入合法性。before_model则聚焦模型输入优化，比如将before_agent预载入的历史记忆整理为模型可识别的上下文格式，过滤冗余信息以降低推理负担，或补充当前请求的元数据（如用户ID）。

可以看到，前文的天气智能体中，before_agent负责从硬存储加载用户过往天气查询记录；before_model则将这些记录精简为“用户曾查南京天气”的关键信息，作为模型生成响应的参考。

6.3 本章小结

本章讲解了智能体记忆模块的设计范式，从直接依赖智能体原生短期记忆的基础模式入手，逐步延伸至基于硬存储的长期记忆持久化存储与动态调用，完整覆盖了从临时记忆到长期记忆的全流程实现逻辑。

同时，我们针对性解决了长期记忆预载入过程中的核心痛点——历史记忆冗余导致的上下文臃肿、Token消耗过高及匹配精准度不足的问题，通过引入BM25文本相关性检索算法，从海量历史记录中精准筛选出与用户当前提问最相近的3条核心信息，从而既避免了全量载入历史记录造成的资源浪费和推理干扰，又确保智能体能够高效调用关键过往交互信息。

这种“短期记忆打底 + 长期记忆精准补位”的设计，让智能体的记忆管理既具备实时性又兼顾高效性，不仅大幅提升了回复的连贯性与针对性，还降低了大模型的上下文压力，为复杂场景下的持续交互提供了稳定可靠的记忆支撑，让智能体真正具备“记关键、用得当”的记忆能力。