【Day 5】LangChain (3) 實作 LLM 對話紀錄系統

在 LLM 應用程式中，每一次呼叫都是獨立的，導致缺乏上下文，實用性大打折扣。為了讓 LLM 能夠記住先前的對話內容，我們可以透過管理對話紀錄的方式來實作對話記憶功能。今天將介紹如何在 LangChain 中，利用 `ChatPromptTemplate` 和 `RunnableWithMessageHistory` 來建立一個能夠維持對話上下文的系統，並實作三種常見的對話紀錄：**Buffer**、**Buffer Window** 和 **Summary**。

1. 核心組件

1.1 `ChatPromptTemplate`

首先，我們利用 `ChatPromptTemplate` 將 System Prompt、對話歷史（`chat_history`）和使用者輸入（`{input}`）整合在一起，建立一個完整的提示模板。

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

prompt_template = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful assistant -- Gemini 2.5 flash lite."),
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)
```

1.2 `InMemoryChatMessageHistory()`

接著，我們使用 `InMemoryChatMessageHistory()` 來管理對話紀錄，它會把資料暫存在記憶體。為了讓每個對話 session 都能有獨立的歷史紀錄，我們可以建立一個字典 `store` 來儲存不同 session 的對話紀錄。

```
store = {}
def get_session_history(session_id: str) → BaseChatMessageHistory:
```

```
    if session_id not in store:
        store[session_id] = InMemoryChatMessageHistory()
    return store[session_id]
```

1.3 RunnableWithMessageHistory

在 LangChain 中，`RunnableWithMessageHistory` 是一個功能強大的工具，它能優雅地整合 Runnable（例如 `prompt | llm` ）與**管理對話紀錄**。早期版本中 LangChain 曾提供 `ConversationBufferMemory` 等封裝工具，但目前已移除。

傳入的 runnable 必須遵循一定的 **輸入/輸出格式限制**。一些時候，也需要手動傳入 `input_messages_key` 、 `output_messages_key` 和 `chat_messages_key` 。

1.3.1 Input 格式

必須符合以下其中一種：

- **一個 BaseMessages 的 list**
- **一個 dict，且僅包含所有訊息的單一 key**
  - 此時需要傳入 `input_messages_key`
- **一個 dict，分別包含目前輸入與歷史訊息**
  - 此時需要同時指定 `input_messages_key` 與 `chat_messages_key`
  - 若輸入 key 對應的是字串，系統會將其視為 `HumanMessage`

1.3.2 Output 格式

必須符合以下其中之一：

- 一個可轉換為 `AIMessage` 的字串
- 一個 BaseMessage 或 BaseMessages 的序列
- 一個 dict，其中包含 BaseMessage 或其序列
  - 此時需要指定 `output_messages_key`

1.3.3 執行流程

整體來說，`RunnableWithMessageHistory` 會自動完成以下步驟：

1. 接收輸入並記錄為 `HumanMessage`。

2. 將輸入與對話紀錄一併傳入 runnable：

```
{
   "{input_messages_key}": ...,
   "{chat_messages_key}": ...
}
```

3. 接收模型輸出的 `AIMessage`。
4. 將輸出寫入對話紀錄中，更新 chat history。

2. 三種對話紀錄策略

2.1 Conversation Buffer Memory

最基礎的方式，會保存完整的對話紀錄並傳入模型。缺點是當輪數過多時，token 容易超過限制，或導致成本增加。

```
chain = prompt_template | llm

chain_with_buffer_memory = RunnableWithMessageHistory(
    chain,
    get_session_history=get_session_history
    input_messages_key="input",
    history_messages_key="chat_history",
)
```

2.2 Conversation Buffer Window Memory

會保留**最後 k 個訊息**或**最後 k 個 tokens 的訊息**，相當於「裁剪」對話紀錄。在這裡，我們會用 `trim_messages` 這個函數，在傳入 LLM 前先修剪紀錄，控制上下文長度。

關於 `token_counter`：

- `len` 用訊息的數量裁剪對話歷史。
- `BaseLanguageModel` 會調用模型的 `.get_num_tokens_from_messages()` 方法來計算 token 數量。
- `count_tokens_approximately` 在文檔裡面推薦使用，計算大約的 token 數量。（在 langchain_core.messages.utils 裡）

```
from langchain_core.messages.utils import trim_messages

# 只處理 'chat_history' 所以先包裝一下函數
def trim_chat_history(x):
    trimmed_messages = trim_messages(
        x['chat_history'],
        max_tokens=2,
        strategy="last",
        token_counter=len,
        include_system=False,
        allow_partial=False
    )
    return trimmed_messages

# {"input": ..., "chat_history": messeges]}
# → {"input": ..., "chat_history": trimmed_messeges}

trimmer = RunnablePassthrough.assign(chat_history=trim_chat_history)
chain = trimmer | prompt_template | llm

# 接著像之前一樣定義 RunnableWithMessageHistory
```

2.3 Summary

我們也可以呼叫一個 LLM 來總結，一樣把總結的 chain 放在 `prompt | llm` 之前。

```
summarizer = GoogleGenerativeAI(model="gemini-2.5-flash-lite")
summary_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "Please summarize the following conversation history conci
sely. If the list is empty, just return nothing."),
        MessagesPlaceholder(variable_name="chat_history"),
    ]
)
main_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful AI assistant. The conversation so far is s
```

```
ummarized as follows: {summary}. If the list is empty, it means it's the start of the conversation."),
        ("human", "{input}"),
    ]
)

summarization_chain = summary_prompt | summarizer

# {"input": ..., "chat_history": messeges]}
# → {"input": ..., "chat_history": messeges, "summary": summarized_messeges]}
# addtional keys will be ignored

runnable = RunnablePassthrough.assign(
    summary=lambda x: summarization_chain.invoke({"chat_history": x["chat_history"]})
)

chain = runnable | main_prompt | llm

# 接著像之前一樣定義 RunnableWithMessageHistory
```

3. 總結

LLM 對話紀錄系統的核心在於 **如何管理 chat history**。

- **Buffer Memory**：完整保留對話，簡單直接但不適合長對話。
- **Buffer Window Memory**：只留最新的訊息，適合近期對話為主的應用。
- **Summary Memory**：透過摘要保留語境，適合長期、多回合的對話場景。

實際上，我們可以將這三種策略混合使用，例如先透過 Buffer Window 控制長度，再搭配 Summary 來保留重要資訊，來兼顧效能與上下文連貫性。