Ch.05 Memory

Date	@2025년 11월 15일
Category	GDGoC
done	☑
day of the week	Sat

1. 개요

대화형 인공지능 시스템에서 “기억(Memory)”은 단순 응답 생성 이상의 핵심 기능이다.

LangChain은 LLM을 활용한 애플리케이션에서 다양한 메모리 구조를 제공하여 **맥락 유지, 장기 기억, 요약, 검색 기반 회상** 등을 손쉽게 구현할 수 있다.

이 문서에서는 LangChain이 제공하는 10종의 대표적인 메모리 클래스를 ① 개념, ② 코드 구조, ③ 활용 목적 중심으로 통합 정리한다.

2. Conversation Memory 계열

2.1 ConversationBufferMemory

- **핵심 개념**: 모든 대화 히스토리를 순차적으로 저장하는 가장 단순한 형태의 메모리.
- **구조**

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory()
memory.save_context(inputs={"human": "안녕하세요"}, outputs={"ai": "반갑습니다!"})
memory.load_memory_variables({})
```

- **특징**
  - `save_context(inputs, outputs)`로 대화 저장
  - `load_memory_variables({})["history"]`로 전체 히스토리 반환

- `return_messages=True` 설정 시 `HumanMessage`, `AIMessage` 객체로 반환

- **활용**: 고객센터, 상담, 인터뷰 등 전체 대화 맥락이 중요한 경우.
- **Chain 예시**

```
from langchain_openai import ChatOpenAI
from langchain.chains import ConversationChain

conversation = ConversationChain(
    llm=ChatOpenAI(temperature=0),
    memory=ConversationBufferMemory()
)
```

2.2 ConversationBufferWindowMemory

- **핵심 개념**: 전체 히스토리 대신 **최근 k회 대화만 유지**하는 슬라이딩 윈도우 메모리.
- **구조**

```
from langchain.memory import ConversationBufferWindowMemory
memory = ConversationBufferWindowMemory(k=2, return_messages=
True)
```

- **특징**
  - 매개변수 `k`로 최근 n개의 상호작용만 보존
  - 긴 대화의 과도한 토큰 사용을 방지
- **활용**: 챗봇·헬프데스크 등 최근 대화 맥락만 중요할 때.

2.3 ConversationTokenBufferMemory

- **핵심 개념**: "대화 개수"가 아니라 **토큰 길이 기준으로 히스토리 제한**
- **코드**

```
from langchain.memory import ConversationTokenBufferMemory
from langchain_openai import ChatOpenAI
```

```
memory = ConversationTokenBufferMemory(
    llm=ChatOpenAI(), max_token_limit=150, return_messages=True)
```

- **특징**
  - 토큰 초과 시 자동으로 오래된 기록 삭제
  - 모델별 토큰 한도를 안정적으로 유지
- **활용**: 긴 응답을 생성하는 기술 지원·설명형 챗봇

3. 구조적 기억 메모리

3.1 ConversationEntityMemory

- **핵심 개념**: 대화 중 등장하는 **인물·장소·사물(엔티티)**의 속성 및 관계를 기억
- **코드**

```
from langchain.memory import ConversationEntityMemory
from langchain_openai import ChatOpenAI
memory = ConversationEntityMemory(llm=ChatOpenAI())
```

- **동작 원리**
  - LLM을 이용해 대화 중 엔티티를 추출
  - 각 엔티티별 정보 저장 (`memory.entity_store.store`)
- **활용 예시**

```
{'테디': '개발자이며 셜리와 회사를 창업할 계획',
 '셜리': '디자이너이며 테디와 함께 창업 예정'}
```

- **적용**

```
from langchain.chains import ConversationChain
conversation = ConversationChain(
    llm=ChatOpenAI(),
```

```
    memory=ConversationEntityMemory(llm=ChatOpenAI())
)
```

3.2 ConversationKGMemory (Knowledge Graph Memory)

- **핵심 개념**: 대화에서 추출한 개체 간 관계를 **지식 그래프 형태**로 저장
- **코드**

```
from langchain.memory import ConversationKGMemory
memory = ConversationKGMemory(llm=ChatOpenAI(), return_messages=True)
```

- **예시**

```
Input: 김셜리씨는 판교에 거주하며 우리 회사의 신입 디자이너입니다.
→ On 김셜리씨: 김셜리씨 is a 신입 디자이너. 김셜리씨 is in 우리 회사.
```

- **활용**
  - 캐릭터 관계 추적, 기업 조직도, 사용자 프로필 그래프 구축

4. 요약 기반 메모리

4.1 ConversationSummaryMemory

- **핵심 개념**: 모든 대화를 요약하여 **압축된 대화 맥락**으로 저장
- **코드**

```
from langchain.memory import ConversationSummaryMemory
memory = ConversationSummaryMemory(llm=ChatOpenAI(), return_messages=True)
```

- **특징**
  - 과거 대화를 자동 요약 → 긴 히스토리도 효율 관리

- 프롬프트에 포함할 토큰 절감

- **활용**
  - 장시간 고객 지원, 상담 로그 요약, 회의록 자동 생성

4.2 ConversationSummaryBufferMemory

- **핵심 개념**: "요약 + 최신 대화 유지"를 결합한 하이브리드 메모리
- **코드**

```
from langchain.memory import ConversationSummaryBufferMemory
memory = ConversationSummaryBufferMemory(
    llm=ChatOpenAI(), max_token_limit=200, return_messages=True)
```

- **결과**
  - 오래된 대화 → 요약 저장
  - 최신 1~2회 대화 → 원문 유지
- **활용**
  - 컨설팅/튜터링 등 최근 발화의 정확한 참조가 필요한 장기 대화

5. 검색 기반 메모리

5.1 VectorStoreRetrieverMemory

- **핵심 개념**: 대화 내용을 **벡터 임베딩으로 저장** 후, 질의 시 의미적으로 가장 유사한 기록을 검색
- **코드**

```
import faiss
from langchain.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
from langchain.memory import VectorStoreRetrieverMemory
```

```
embeddings = OpenAIEmbeddings()
index = faiss.IndexFlatL2(1536)
vectorstore = FAISS(embeddings, index)
retriever = vectorstore.as_retriever(search_kwargs={"k":1})
memory = VectorStoreRetrieverMemory(retriever=retriever)
```

- **활용**
  - 문서 QA, 지식 검색 챗봇, 시맨틱 회상 기반 대화

6. Chain 통합 및 확장

6.1 LCEL 기반 메모리 통합

LCEL(RunnableChain) 구조에서는 메모리를 직접 체인에 주입할 수 있다.

```
from operator import itemgetter
from langchain.memory import ConversationBufferMemory
from langchain_core.prompts import ChatPromptTemplate, MessagesPlace
holder
from langchain_core.runnables import RunnableLambda, RunnablePassthro
ugh
from langchain_openai import ChatOpenAI

model = ChatOpenAI()
memory = ConversationBufferMemory(return_messages=True, memory_ke
y="chat_history")
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful chatbot"),
    MessagesPlaceholder(variable_name="chat_history"),
    ("human", "{input}")
])
chain = (RunnablePassthrough.assign(
            chat_history=RunnableLambda(memory.load_memory_variables)
            | itemgetter("chat_history"))
        | prompt | model)
```

6.2 커스텀 ConversationChain 구현

```
class MyConversationChain:
    def __init__(self, llm, prompt, memory):
        self.llm, self.prompt, self.memory = llm, prompt, memory
    def invoke(self, query):
        answer = (self.prompt | self.llm).invoke({"input": query})
        self.memory.save_context({"human": query}, {"ai": answer})
        return answer
```

7. SQL 기반 대화 저장

7.1 SQLChatMessageHistory

LangChain은 `SQLAlchemy`를 활용해 대화 히스토리를 관계형 DB에 저장할 수 있다.

```
from langchain_community.chat_message_histories import SQLChatMessa
geHistory
chat_history = SQLChatMessageHistory(session_id="user1", connection="s
qlite:///chat.db")
chat_history.add_user_message("안녕, 나는 테디야.")
chat_history.add_ai_message("안녕 테디, 만나서 반가워!")
```

7.2 RunnableWithMessageHistory 적용

```
from langchain_core.runnables.history import RunnableWithMessageHistor
y
chain_with_history = RunnableWithMessageHistory(
    chain, get_chat_history,
    input_messages_key="question",
```

```
    history_messages_key="chat_history"
)
```

- **user_id / conversation_id**로 다중 세션 관리 가능
- **활용**: 고객별 세션 저장, 챗봇 개인화, 장기 로그 분석

8. 세션 기반 Runnable 메모리

8.1 ChatMessageHistory 기반

```
from langchain_community.chat_message_histories import ChatMessageHi
story
store = {}
def get_session_history(sid):
    if sid not in store:
        store[sid] = ChatMessageHistory()
    return store[sid]
```

→ `session_id` 단위로 대화 지속성 유지.

→ LangSmith 로그를 통해 히스토리 추적 가능.

9. 결론

LangChain의 메모리 시스템은 **대화형 LLM의 상태 관리 계층**을 구성한다.

Buffer, Token, Entity, Graph, Summary, Vector, SQL 등은 각각 "**시간적 기억 / 구조적 기억 / 요약 기억 / 검색 기억 / 외부 저장**"의 역할을 담당한다.

이러한 구조적 메모리의 적절한 조합은 단순한 질의응답을 넘어 **지속적 학습, 개인화된 상호작용, 맥락 기반 의사결정**을 가능하게 한다.