CH05 - 메모리(Memory)

I. 대화 버퍼 메모리(ConversationBufferMemory)

1. 역할

- **대화 기록 전체를 메모리에 그대로 저장하는 가장 기본적인 메모리**
- 이전 메시지를 “문자열” 또는 “메시지 객체 리스트”로 반환하여 LLM Prompt에 포함시킬 수 있음

---

2. 주요 기능

(1) save_context(inputs, outputs)

- 사용자의 입력과 AI의 출력을 대화 기록으로 저장
- 형식:

```
memory.save_context(
    inputs={"human": "..."},
    outputs={"ai": "..."}
)
```

(2) load_memory_variables({})

- 저장된 대화 기록(history)을 불러옴
- 기본 반환값:

```
{"history": "Human: ...\nAI: ..."}
```

- return_messages=True 설정 시:

```
{"history": [HumanMessage(...), AIMessage(...)]}
```

---

3. return_messages=True

- history를 하나의 긴 문자열이 아니라

  LangChain 메시지 객체 형식(HumanMessage, AIMessage) 으로 돌려줌

- Retrieval, Multi-turn QA, Agent 등에서 선호됨

---

4. ConversationChain과 함께 사용하기

```
conversation = ConversationChain(
    llm=llm,
    memory=ConversationBufferMemory()
)
```

- ConversationChain 사용 시 자동으로

  입력 → LLM → 출력 → 메모리 저장 구조가 됨
- 이후 메시지에서 “이전 대화 기반” 요청을 하면

  ConversationBufferMemory가 이전 기록을 Prompt에 삽입해줌

---

5. 장점 & 단점

장점

- 가장 단순하고 직관적
- 전체 대화 맥락을 그대로 유지
- 디버깅·튜토리얼·기본 챗봇 구현에 적합

단점

- 대화가 길어지면 Prompt가 비대해짐
- 중요 정보 압축이나 요약 없음
- 장기 대화나 긴 문서 처리에는 적합하지 않음

---

6. 언제 쓰는 메모리인가?

- 간단한 상담 챗봇
- 튜토리얼, 테스트용 대화 흐름
- 짧고 맥락 유지가 중요한 태스크

> 반대로, 대화가 길어지는 경우에는

> **ConversationBufferWindowMemory / ConversationSummaryMemory / CombinedMemory** 를 사용해야 함.

II. 대화 버퍼 윈도우 메모리 (ConversationBufferWindowMemory)

1. 역할

- 전체 대화를 모두 저장하는 것이 아니라

  “최근 K개 상호작용만 저장하는 슬라이딩 윈도우 메모리”

- 대화가 길어져도 프롬프트가 너무 커지지 않도록 관리하는 목적

---

2. 특징

- 최근 K개의 상호작용만 기억
  - K=2 → “Human- AI” 한 쌍을 기준으로 **최근 2개의 Human ↔ AI 대화만 유지**
  - 오래된 대화는 자동으로 삭제됨
- 프롬프트 길이를 자동으로 제한
  - 비용 절약, 응답 시간 단축
  - LLM의 컨텍스트 오버플로우(too long prompt) 방지

---

3. 주요 파라미터

k (windown size)

- 저장할 최근 상호작용 개수

  예:

```
memory = ConversationBufferWindowMemory(k=2)
```

return_messages=True

- 문자열이 아닌 메시지 객체(HumanMessage/AIMessage)로 반환

---

4. 기본 사용 패턴

```
from langchain.memory import ConversationBufferWindowMemory

memory = ConversationBufferWindowMemory(k=2, return_messages=Tru
e)

memory.save_context(
    inputs={"human": "..."},
    outputs={"ai": "..."}
)
```

저장된 기록 불러오기:

```
memory.load_memory_variables({})["history"]
```

---

5. 반환 결과 (예시)

K=2일 경우:

```
[
  HumanMessage(content='정보를 모두 입력했습니다. 다음 단계는 무엇인가요?'),
  AIMessage(content='입력해 주신 정보를 확인했습니다...'),
  HumanMessage(content='모든 절차를 완료했습니다. 계좌가 개설된 건가요?'),
  AIMessage(content='네, 계좌 개설이 완료되었습니다...')
]
```

→ 가장 최근 상호작용 2개(= Human-AI 2쌍)만 남음

---

6. 언제 사용하는 메모리인가?

적합한 상황

- 대화가 매우 길어질 수 있는 앱
- 최근 문맥만 중요하고 과거 맥락은 필요 없는 챗봇
- 비용/속도를 아끼고 싶을 때

- Agent / Tool 호출 시 긴 Prompt를 피하고 싶을 때

적합하지 않음

- 장기적인 맥락이 필요한 작업(예: 스토리텔링, 상담)
- 전체 히스토리를 활용해야 하는 QA

III. 대화 토큰 버퍼 메모리 (ConversationTokenBufferMemory)

1. 역할

- **최근 대화를 토큰 단위로 관리하는 메모리**
- 저장할 대화량을 "메시지 개수(K)"가 아니라

  전체 토큰 수(max_token_limit) 기준으로 제한
- 전체 토큰 수가 limit를 넘으면, **가장 오래된 대화부터 자동 삭제(Flush)**

---

2. 특징

토큰 기반 제한

- 대화가 길어지면 자동으로 앞부분을 잘라냄

  → 프롬프트가 길어져 LLM 비용이 증가하는 문제 해결

  → LLM context window 내에서 안전하게 유지 가능

정교한 관리

- BufferWindowMemory는 "K개 메시지" 기준
- TokenBufferMemory는 "토큰 수(T)" 기준

  → 긴 문장/짧은 문장에 따라 실제 프롬프트 길이를 훨씬 정확하게 제어 가능

---

3. 주요 파라미터

max_token_limit

- 저장할 최대 토큰 수

예:

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

llm

- 토큰 계산을 위해 **LLM 객체를 반드시 입력해야 함**

return_messages=True

- 문자열이 아닌 HumanMessage / AIMessage 객체 배열로 반환

---

4. 동작 방식

1. save_context로 대화를 저장
2. 저장된 모든 메시지를 token length로 계산
3. `max_token_limit`보다 커지면

   → **가장 오래된 메시지부터 제거 (Flush)**
4. 최신 대화만 기억한 상태로 유지

---

5. 예시

토큰 제한 150으로 설정 후 여러 메시지 저장 →

마지막 대화만 남음:

```
[
  HumanMessage("감사합니다, 도움이 많이 되었어요!"),
  AIMessage("언제든지 도와드릴 준비가 되어 있습니다...")
]
```

→ 앞의 모든 대화는 token limit 초과로 자동 삭제됨

---

6. 언제 사용하는 메모리인가?

적합한 상황

- 대화가 길어질 수 있는 서비스
- Prompt 비용을 관리해야 함
- Context Window 초과 방지 필요
- 메시지 개수가 아닌 “실제 토큰 길이” 기준으로 정교하게 조절하고 싶은 경우

적합하지 않음

- 전체 대화를 반드시 유지해야 하는 상담/스토리 서비스
- 과거 맥락이 중요하고 토큰으로 잘리면 안 되는 경우

Ⅳ. 대화 엔티티 메모리(ConversationEntityMemory)

1. 역할

- 대화 속에서 등장하는 **엔티티(Entity: 사람, 회사, 장소 등)**에 대한 정보를 **자동으로 추출하고 기억**하는 메모리.
- 단순히 기록을 저장하는 것이 아니라

  LLM이 문맥 속 사실 정보만 추출해 엔티티별로 정리해 저장함.

---

2. 특징

사실(Fact) 중심 메모리

- Human/AI의 메시지를 그대로 저장하는 것이 아니라

  “테디 = 개발자”, “셜리 = 디자이너” 같은 구조화된 지식을 쌓음

누적 학습

- 동일 엔티티가 다시 등장하면

  → 이전 정보를 불러와

  → LLM이 업데이트하여 더 풍부한 설명을 저장함

LLM을 사용하여 지식 추출

- 메시지를 엔티티로 변환하는 작업 자체도 모델이 수행
- ConversationBufferMemory 같은 단순 저장 방식이 아님

3. 기본 사용 패턴

1) Entity Memory 프롬프트 사용

```
from langchain.memory.prompt import ENTITY_MEMORY_CONVERSATION
_TEMPLATE
```

2) ConversationChain 구성

```
conversation = ConversationChain(
    llm=llm,
    prompt=ENTITY_MEMORY_CONVERSATION_TEMPLATE,
    memory=ConversationEntityMemory(llm=llm),
)
```

3) 대화를 입력하면 자동으로 엔티티 추출 및 저장

```
conversation.predict(
    input="테디는 개발자이고 셜리는 디자이너입니다. 둘은 창업을 준비 중입니다."
)
```

4) 엔티티 정보 확인

```
conversation.memory.entity_store.store
```

예시 결과:

```
{
  '테디': '테디는 개발자이고 셜리와 함께 회사를 차릴 계획.',
  '셜리': '셜리는 디자이너이며 테디와 함께 창업 준비 중.'
}
```

4. Entity Memory의 동작 방식

1. **대화 입력 → 프롬프트 전달**

2. LLM이 입력 내용에서 **엔티티 인식(NER)**
3. 엔티티별 “사실 정보” 추출
4. 기존 store에 추가 또는 업데이트
5. 이후 대화에서 자동으로 그 사실을 활용하여 응답

---

5. 언제 사용하는 메모리인가?

적합한 상황

- 사람/장소/제품 등 **특정 대상에 대한 정보가 축적되는** 서비스
- 예:
  - 장기 상담/코칭
  - 게임 NPC 설정 저장
  - CRM 고객 정보 기억
  - 인물 설정 유지되는 스토리 생성기

부적합한 상황

- 단순히 최근 맥락 기억만 필요할 때
- 사실 추출이 필요하지 않은 일반 챗봇
- 대화가 매우 길고 엔티티가 계속 추가되는 경우(비용↑)

V. 대화 지식그래프 메모리(ConversationKGMemory)

1. 역할

- 대화에서 **지식 그래프(Knowledge Graph)** 형태로 정보를 추출하여
  개체(Entity) 간 관계(Relationship) 를 저장하는 메모리.
- 단순히 “사람 → 직업” 같은 사실을 저장하는 것이 아니라
  'A는 B에 있다', 'A는 B의 동료다', 'A는 C 회사에 소속' 같은
  연결 구조까지 LLM이 자동으로 그래프 형태로 정리함.

---

2. 핵심 특징

관계 중심(Relationship-Centric) 메모리

- EntityMemory는 엔티티의 속성(attribute) 중심
- KGMemory는 **엔티티 간 관계(relation)** 중심 저장

예시:

> On Pangyo: Pangyo has resident 김셜리씨.
> On 김셜리씨: 김셜리씨 is a 신입 디자이너. 김셜리씨 is in 우리 회사.

→ '김셜리씨'가 어떤 장소(Pangyo), 어떤 조직(우리 회사)과 어떻게 연결되는지 저장됨.

LLM이 관계 추출 자동 수행

대화 입력 →

LLM이 관계를 분석 →

지식 그래프 형태로 저장:

예:

- "Shirley is a new designer at our company."

  → "Shirley is a new designer."

  → "Shirley is at company."

관계 기반 질의가 가능해짐

사용자가 "Shirley는 누구야?" 라고 물으면

KGMemory는 저장된 'Shirley'와 관련된 관계들을 모두 모아서 제공:

> On Shirley: Shirley is a coworker.
> Shirley is a new designer.
> Shirley is at company.

---

3. 기본 사용 패턴

1) 메모리 생성

```
memory = ConversationKGMemory(llm=llm, return_messages=True)
```

2) 대화 저장

```
memory.save_context(
    {"input": "이쪽은 Pangyo에 거주중인 김셜리씨입니다."},
    {"output": "김셜리씨는 누구시죠?"}
)
memory.save_context(
    {"input": "김셜리씨는 우리 회사의 신입 디자이너입니다."},
    {"output": "만나서 반갑습니다."}
)
```

3) 지식 그래프 조회

```
memory.load_memory_variables({"input": "김셜리씨는 누구입니까?"})
```

결과 예시:

```
[
  SystemMessage(content="On Pangyo: Pangyo has resident 김셜리씨."),
  SystemMessage(content="On 김셜리씨: 김셜리씨 is a 신입 디자이너. 김셜리씨 is in 우리 회사.")
]
```

---

4. ConversationChain에 통합 사용

ConversationChain에 메모리를 넣으면

AI는 Relevant Information 안의 KG 정보를 기반으로만 답변하게 됨.

```
conversation_with_kg = ConversationChain(
    llm=llm,
    prompt=prompt,
    memory=ConversationKGMemory(llm=llm)
)
```

대화 예:

입력

“My name is Teddy. Shirley is a coworker and a new designer.”

이후, KG 조회:

```
conversation_with_kg.memory.load_memory_variables(
    {"input": "who is Shirley?"}
)
```

결과:

```
On Shirley: Shirley is a coworker. Shirley is a new designer. Shirley is at co
mpany.
```

---

5. 언제 사용하는 메모리인가?

이런 상황에 최적

- 스토리텔링(특정 인물들과 관계 연결)
- RPG/NPC 설정 유지
- 인물/장소/조직 간 연결 관계가 많은 세계관
- 복잡한 조직 구조/지식 관계를 기억해야 하는 챗봇
- 상담/코칭에서 관계도 추적 (가족/직장 구조 등)

이런 경우는 부적합

- 단순 대화 기록만 필요할 때 → BufferMemory
- 엔티티 속성만 기억하면 될 때 → EntityMemory
- LLM 비용이 낮아야 하는 경우 (KGMemory는 LLM에 의존해 정보 추출)

VI. 대화 요약 메모리(ConversationSummaryMemory)

1. 역할

- 대화가 길어질수록 모든 과거 메시지를 저장하지 않고

  LLM이 자동으로 요약(summary) 하여 압축된 컨텍스트로 유지하는 메모리.
- 긴 대화를 효율적으로 유지하면서도 핵심 정보는 남기는 방식.

---

2. 특징

과거 대화를 요약본으로 저장

- 이전 대화들은 한 문단의 요약으로 바뀜
- 최신 대화는 그대로 유지됨

긴 대화에서도 토큰 사용량 최소화

- "과거 맥락 유지 + 토큰 절약"이라는 목적에 최적화

프롬프트에 요약문만 들어가므로 비용 절감

---

3. 기본 사용 패턴

```
memory = ConversationSummaryMemory(
    llm=ChatOpenAI(temperature=0),
    return_messages=True
)
```

대화 저장

```
memory.save_context(
    inputs={"human": "여행자 보험은 포함되어 있나요?"},
    outputs={"ai": "네, 기본 여행자 보험을 제공합니다..."}
)
```

요약본 조회

```
memory.load_memory_variables({})["history"]
```

---

4. 결과 예시

(Few turns later...)

```
SystemMessage(
  "The human asks about the price of a European travel package...
   ...The human asks about the reservation deposit and cancellation polic
y..."
)
```

→ 전체 대화가 깔끔하게 한 단락으로 요약됨

---

5. 언제 사용하는 메모리인가?

적합한 상황

- 상담/고객센터/챗봇 등 **대화가 길어지는 서비스**
- 전체 맥락을 유지해야 하지만 token 사용량을 줄여야 할 때
- "분량은 적게, 맥락은 유지"가 목표일 때

부적합한 상황

- 모든 대화를 그대로 기억해야 하는 경우

  (요약 과정에서 일부 세부 정보가 사라질 수 있음)

ConversationSummaryBufferMemory 핵심 정리

(*SummaryMemory + TokenBufferMemory 의 하이브리드*)

1. 역할

- 두 기능을 결합:
  1. **최근 대화는 원문 그대로 유지 (Buffer)**
  2. 오래된 대화는 **LLM이 요약해서 압축 저장 (Summary)**
- 최종 목표:

  "지금 대화는 상세하게 + 오래된 대화는 요약 + 전체 토큰 길이 제한"

---

2. 특징

Token 기반으로 요약 시점 결정

- `max_token_limit` 초과 시

  → 오래된 대화를 Summary로 압축

  → 최신 메시지 1~2개는 원문 그대로 유지

긴 대화에 가장 효율적

- BufferMemory보다 훨씬 똑똑하고
- SummaryMemory보다 문맥 손실이 적음

---

3. 기본 사용 패턴

```
memory = ConversationSummaryBufferMemory(
    llm=llm,
    max_token_limit=200,
    return_messages=True
)
```

대화 저장 → 자동 요약 & 버퍼 관리

```
memory.save_context({"human": "호텔 등급은?"}, {...})
```

history 조회

```
memory.load_memory_variables({})["history"]
```

---

4. 작동 결과 예시

max_token_limit를 넘으면:

```
[SystemMessage(
    "Summary: 사용자가 가격, 관광지, 보험, 비즈니스 업그레이드 등 질문..."),
 HumanMessage("패키지에 포함된 호텔의 등급은 어떻게 되나요?"),
 AIMessage("이 패키지에는 4성급 호텔 숙박이 포함되어 있습니다...")
]
```

→ 과거 대화는 Summary로 압축

→ 최근 1~2개의 대화는 그대로 유지

---

5. 언제 사용하는 메모리인가?

추천되는 상황

- 장기 대화
- 상담/고객 응대/교육 코칭
- "과거 맥락 필요 + 토큰 절감 + 최근 대화는 그대로 유지"가 모두 필요한 경우

피해야 하는 상황

- 매우 정밀한 과거 데이터가 필요할 때

  (요약 과정에서 세부 사항 일부 손실 가능)

VII. 벡터저장소 검색 메모리 (VectorStoreRetrieverMemory)

1. 역할

- 대화 내용을 **벡터 임베딩(embedding)** 으로 저장하고

  이후 사용자가 질문할 때, **의미적으로 가장 관련성 높은 문서(K개)** 를 검색하여 반환하는 메모리.
- 기존 메모리들과 달리 **대화 순서를 저장하지 않고**,

  "의미 연관성 기반 검색"이라는 점이 핵심.

---

2. 특징

순서가 아닌 의미적 유사도 기반 메모리

- BufferMemory: "시간 순서" 기반
- ❗ VectorStoreRetrieverMemory: "의미적 유사도" 기반

  → 대화를 아무리 오래 했어도, LLM은 "현재 질문과 가장 관련 있는 내용"만 다시 불러옴

벡터 DB를 사용하여 장기 기억 가능

- 벡터 스토어에 저장되기 때문에

대화량이 늘어나도 오래된 내용도 검색 가능

- 마치 “장기 기억(Long-term memory)” 같은 구조

최고의 장점: 정확한 문맥 검색

- 이전에 말했던 내용을 순서와 상관없이 의미적으로 다시 꺼낼 수 있음

---

3. 기본 아키텍처

```
사용자 메시지 → 임베딩 → 벡터 스토어 저장
      ↑                    ↓
   Retrieval(검색) ← 현재 질문 임베딩
```

---

4. 핵심 코드 패턴

1) 벡터 스토어 초기화

```
embeddings_model = OpenAIEmbeddings()
index = faiss.IndexFlatL2(1536)
vectorstore = FAISS(embeddings_model, index, InMemoryDocstore({}), {})
```

2) RetrieverMemory 생성

```
retriever = vectorstore.as_retriever(search_kwargs={"k": 1})
memory = VectorStoreRetrieverMemory(retriever=retriever)
```

3) 대화 저장

```
memory.save_context(
    inputs={"human": "..."},
    outputs={"ai": "..."}
)
```

4) 질문 → 관련성 높은 대화 검색

```
memory.load_memory_variables({"prompt": "면접자 전공은 무엇인가요?"})["history"]
```

결과 예시:

human: 자기소개 부탁드립니다.
ai: 저는 컴퓨터 과학을 전공했습니다.

---

5. 주요 장점

의미적 검색 능력 (Semantic Search)

→ 사용자가 “이전에 말했던 전공이 뭐였지?”라고 물으면,

가장 관련성 높은 메시지를 자동 검색

대규모 대화에도 효율적

→ 순서를 저장하지 않기 때문에, 오래된 내용도 검색 가능

RAG/Retriever 기반 에이전트와 완벽 호환

---

6. 단점

순서 정보 없음

→ 흐름 기반 대화에는 부적합

→ “이전에 내가 뭐라 했지?” 같은 질문은 의미적으로 처리되지만

“바로 직전에 한 말” 같은 순서 기반 문맥은 유지되지 않음

임베딩 비용 / 벡터 저장소 필요

→ 저장 시마다 임베딩(embedding) 호출됨

→ 비용 증가

---

7. 언제 사용하는 메모리인가?

적합한 상황

- RAG 기반 챗봇
- FAQ/면접/학습 내용 기반 대화
- 대화량이 많고 오래된 내용까지 모두 검색해야 하는 경우
- 의미적으로 연관된 이전 정보만 필요할 때

부적합한 상황

- 연속된 대화 흐름이 중요할 때

  → BufferMemory / WindowMemory 사용

VIII. LCEL Chain 에 메모리 추가

1. 개념 요약

- LCEL(Runnable 기반 Chain)에서는 ConversationChain처럼 자동으로 메모리를 연결해주지 않음.
- 따라서 **메모리(load → prompt → model → save)** 과정을 직접 체인에 연결해야 함.
- 핵심은:

  input → memory.load → prompt → model → memory.save

---

2. 메모리 구조 흐름 (핵심 다이어그램)

```
사용자 입력(input)
    ↓
memory.load_memory_variables()
(과거 대화 기록 chat_history 로 변환)
    ↓
ChatPromptTemplate (MessagesPlaceholder에 삽입)
    ↓
LLM(ChatOpenAI)
    ↓
응답 생성(response)
    ↓
memory.save_context(inputs, outputs)
(대화 기록 업데이트)
```

이 구조를 LCEL로 직접 구성하는 것이 핵심.

---

3. 기본 구성 요소

1) ChatPromptTemplate

```
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful chatbot"),
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)
```

2) Memory 인스턴스 생성

```
memory = ConversationBufferMemory(
    return_messages=True,
    memory_key="chat_history"
)
```

3) 메모리를 LCEL 체인에 연결

- RunnablePassthrough.assign
- RunnableLambda(memory.load_memory_variables)
- itemgetter(memory.memory_key)

```
runnable = RunnablePassthrough.assign(
    chat_history=RunnableLambda(memory.load_memory_variables)
                | itemgetter("chat_history")
)
```

이렇게 해서 prompt에서 필요한 `{chat_history}`가 자동으로 채워짐.

---

4. LCEL 파이프라인 전체 흐름

```
chain = runnable | prompt | model
```

→ chain.invoke({"input": "내용"})

→ 메모리로부터 이전 대화가 prompt에 삽입됨

---

5. 응답 후 메모리 저장

메모리는 자동 저장 X

직접 호출해야 함 O

```
memory.save_context(
    {"human": user_input},
    {"ai": response.content}
)
```

이 부분이 LCEL에서 가장 중요한 차이점.

---

6. 이름 기억 예시 (작동 흐름)

1. 첫 질문

```
input: "만나서 반갑습니다. 제 이름은 테디입니다."
→ 결과 출력
→ memory.save_context 로 대화 저장
```

1. 다음 질문

```
input: "제 이름이 무엇이었는지 기억하세요?"
→ chat_history에 저장된 대화가 prompt에 삽입됨
→ 모델이 “테디”라고 답함
```

---

7. 커스텀 ConversationChain 구현 (핵심 구조)

구조

```
class MyConversationChain(Runnable):
    def __init__(self, llm, prompt, memory):
        self.chain = (
            RunnablePassthrough.assign(
                chat_history = RunnableLambda(memory.load_memory_variables)
                            | itemgetter(memory.memory_key)
            )
            | prompt
            | llm
            | StrOutputParser()
```

```
        )

    def invoke(self, query):
        answer = self.chain.invoke({"input": query})
        memory.save_context({"human": query}, {"ai": answer})
        return answer
```

사용 예

```
conversation_chain.invoke("안녕하세요? 제 이름은 테디입니다.")
conversation_chain.invoke("제 이름이 뭐죠?")
→ "테디라고 하셨죠?"
conversation_chain.invoke("앞으로 영어로만 답해주세요")
conversation_chain.invoke("제 이름이 뭐죠?")
→ "Your name is Teddy."
```

IX. SQLite에 대화내용 저장

1. 목적

- 대화 내용을 **SQLite 같은 RDBMS에 영구 저장**하고

  이후 특정 user_id / conversation_id 기준으로 다시 불러오는 방법.

- LangChain에서 이를 담당하는 클래스:

  `SQLChatMessageHistory`

---

2. SQLChatMessageHistory 기본 사용

생성

```
from langchain_community.chat_message_histories import SQLChatMessa
geHistory

chat_message_history = SQLChatMessageHistory(
    session_id="sql_history",
```

```
    connection="sqlite:///sqlite.db"
)
```

메시지 저장

```
chat_message_history.add_user_message("안녕? 만나서 반가워...")
chat_message_history.add_ai_message("안녕 테디!")
```

메시지 조회

```
chat_message_history.messages
```

---

3. LCEL Chain에 SQLite 기반 메모리 적용

Prompt 정의

```
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful assistant."),
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{question}"),
    ]
)
```

Chain 생성

```
chain = prompt | ChatOpenAI(model_name="gpt-4o") | StrOutputParser()
```

---

4. SQLite DB에서 대화 기록 가져오는 함수

```
def get_chat_history(user_id, conversation_id):
    return SQLChatMessageHistory(
        table_name=user_id,
        session_id=conversation_id,
        connection="sqlite:///sqlite.db",
    )
```

- **user_id → DB 테이블 이름**
- **conversation_id → 세션 ID**

---

5. config 필드 정의 (ConfigurableFieldSpec)

```
config_fields = [
    ConfigurableFieldSpec(
        id="user_id", annotation=str,
        name="User ID", description="Unique identifier for a user.",
        default="", is_shared=True,
    ),
    ConfigurableFieldSpec(
        id="conversation_id", annotation=str,
        name="Conversation ID", description="Unique identifier for a convers
ation.",
        default="", is_shared=True,
    ),
]
```

---

6. RunnableWithMessageHistory로 체인 감싸기

```
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",
    history_factory_config=config_fields,
)
```

---

7. config 설정 후 호출

첫 번째 대화 (conversation1)

```
config = {"configurable": {"user_id": "user1", "conversation_id": "conversation1"}}

chain_with_history.invoke({"question": "안녕 반가워, 내 이름은 테디야"}, config)
→ "안녕하세요, 테디!"
```

이전 기록 기반 답변 확인

```
chain_with_history.invoke({"question": "내 이름이 뭐라고?"}, config)
→ "당신의 이름은 테디라고 하셨죠."
```

---

8. conversation_id 변경 시 새로운 기록으로 취급됨

같은 user_id, 다른 conversation_id

```
config = {"configurable": {"user_id": "user1", "conversation_id": "conversation2"}}

chain_with_history.invoke({"question": "내 이름이 뭐라고?"}, config)
→ "죄송하지만, 당신의 이름을 알 수 없습니다."
```

→ session_id가 다르면 “새로운 대화”로 인식

→ 이전 기록 사용 X

X. RunnableWithMessageHistory에 ChatMessageHistory추가

1. 목적

- LCEL 기반 체인에서 **이전 대화를 자동으로 기억하는 multi-turn QA 챗봇** 만들기.
- Memory 클래스를 사용하지 않고

  ChatMessageHistory + RunnableWithMessageHistory 조합으로 직접 구성.

2. 기본 구조

```
ChatMessageHistory(메모리 저장)
      ↓
RunnableWithMessageHistory(메모리 읽기 + 쓰기)
      ↓
Prompt → LLM → OutputParser
```

`session_id` 기준으로 대화를 구분해 관리함.

3. Prompt 정의

```
prompt = ChatPromptTemplate.from_messages(
   [
      ("system", "당신은 Question-Answering 챗봇입니다. 주어진 질문에 대한 답변을 제공해주세요."),
      MessagesPlaceholder(variable_name="chat_history"),
      ("human", "#Question:\n{question}"),
   ]
)
```

- `chat_history`는 대화 기록이 들어가는 자리 → 반드시 유지 추천
- `question`은 사용자 입력 변수

4. LLM + 체인 구성

```
llm = ChatOpenAI(model_name="gpt-4o")
chain = prompt | llm | StrOutputParser()
```

5. ChatMessageHistory 저장소 준비

store 딕셔너리: session_id → ChatMessageHistory 매핑

```
store = {}

def get_session_history(session_id):
```

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

---

6. RunnableWithMessageHistory 적용

```
chain_with_history = RunnableWithMessageHistory(
    chain,
    get_session_history,
    input_messages_key="question",
    history_messages_key="chat_history",
)
```

- `input_messages_key`: 프롬프트에서 사용자 입력이 들어가는 key
- `history_messages_key`: MessagesPlaceholder로 연결될 key

---

7. 대화 실행 (multi-turn 테스트)

첫 번째 질문

```
chain_with_history.invoke(
    {"question": "나의 이름은 테디입니다."},
    config={"configurable": {"session_id": "abc123"}},
)
```

응답:

```
'안녕하세요, 테디님! 무엇을 도와드릴까요?'
```

이어서 질문 (기억 확인)

```
chain_with_history.invoke(
    {"question": "내 이름이 뭐라고?"},
    config={"configurable": {"session_id": "abc123"}},
)
```

응답:

```
'당신의 이름은 테디입니다.'
```

→ 같은 session_id이므로 대화 연속성 유지됨

---

8. 다른 session_id 사용 시 새로운 대화 시작

```
chain_with_history.invoke(
    {"question": "내 이름이 뭐라고?"},
    config={"configurable": {"session_id": "abc1234"}},
)
```

응답:

```
'죄송하지만, 당신의 이름은 알 수 없습니다.'
```

→ 세션 ID가 바뀌면 완전히 새로운 대화로 인식