AI Agent

Session & Memory

1. Session 및 Memory 이해

LLM의 기억력

LLM은 기억이라는 개념 자체가 없는, 본질적으로 Stateless한 함수

- RNN/LSTM:
  - 순차적으로 토큰을 처리
  - 이전 타임스텝의 정보를 Hidden state라는 메모리 벡터에 지속적으로 누적
- Transformer (LLM의 기반):
  - 모든 입력 토큰을 한 번에 병렬로 처리
  - 이전 요청의 상태를 기억하는 별도의 메모리 네트워크가 없음
  - 모든 API 호출은 완전히 독립적인, 처음 보는 입력으로 간주

LLM의 기억력

챗봇이 이전 대화의 맥락을 이해하는 과정

- API 호출마다 지금까지의 모든 대화 기록 전체를 messages 리스트에 담아 재전송

```
User: "NVIDIA CEO는 누구야?"
App -> LLM: messages = [{"role": "user", "content": "NVIDIA CEO는 누구야?"}]
LLM -> App: {"role": "assistant", "content": "젠슨 황입니다."}
User: "그의 출생년도는?"
App -> LLM: messages = [
{"role": "user", "content": "NVIDIA CEO는 누구야?"},
{"role": "assistant", "content": "젠슨 황입니다."},
{"role": "user", "content": "그의 출생년도는?"} // ❗ 전체 대화 기록을 다시 보냄
```

- Agent의 기억은 LLM의 능력이 아님
- 애플리케이션단에서 대화 기록을 관리하고 프롬프트에 주입해주는 외부 메커니즘

Naive 접근법의 명확한 한계

복잡한 Agent 시나리오에서는 두 가지 심각한 문제에 직면

- 컨텍스트 윈도우 초과 (Context Window Overflow)
  - 한 번에 처리할 수 있는 토큰 한계
  - 대화나 작업이 길어지면, messages 리스트의 토큰 수가 이 한계를 초과
  - 더 이상 요청을 보낼 수 없거나, 오래된 정보를 강제로 삭제해야 함
- API 비용 및 지연 시간 폭증 API 비용은 총 토큰 수(입력+출력)에 비례
  - 대화가 길어질수록, 매번 동일한 과거 기록을 반복해서 전송
  - 비용이 기하급수적으로 증가
  - 입력 텍스트가 길어질수록 LLM의 처리 시간(Latency) 증가

Session, State, Memory

Session

- 대화의 단위를 구분
- 누구와의 대화인지,
- 어떤 프로젝트에 대한 작업인지를 식별
- 관련 없는 기억이 섞이지 않도록 함

State

- 현재 시점의 작업 기억 (Working Memory)
- 작업 계획, 중간 결과물 등 Agent 맥락을 담는 구조화된 스냅샷

Memory

- State를 영속적 저장 (Persistence)
- 대화가 중단되더라도 나중에 이어서 진행

Session, State, Memory

Session

- 사용자와 Agent 간의 상호작용 단위를 나타내는 논리적인 컨테이너
- 동시에 Agent와 상호작용하거나, 여러 개의 다른 작업을 동시에 진행할 때
- 각 상호작용의 기억이 서로 침범하지 않도록 격리하는 역할
- LangGraph에서는 thread_id라는 고유 식별자를 통해 세션을 구분

Session, State, Memory

State

- 그래프의 각 노드를 거칠 때마다 변화하는 Agent의 현재 상태 스냅샷
- 대화 기록(messages)만 저장하는 것이 아님
  - 모든 중간 데이터(작업 계획, 수집된 정보, 재시도 횟수)를 구조화된 형태로 관리
  - Agent가 현재 역할, 진행 작업, 향후 계획 등을 알 수 있음
- Python의 TypedDict를 사용하여 State의 설계도(Schema)를 정의
- Agent가 관리할 기억의 종류와 데이터 타입을 강제하여 안정성을 높임

Session, State, Memory

두 가지 방식을 조합하여 Agent의 기억을 동적으로 조작

- 덮어쓰기 (Overwrite)
  - my_key: str (일반 타입 힌트)
  - 노드가 반환한 값으로 기존 값을 완전히 대체
  - 작업의 현재 단계, 남은 작업 목록, 최종 목표 등 최신 상태를 유지
- 추가 (Append/Add)
  - Annotated[list, operator.add]
  - 노드가 반환한 값을 기존 리스트의 끝에 추가
  - 대화 기록, 작업 로그, 수집된 데이터 등 과거의 모든 기록을 보존

Session, State, Memory

Memory:

- State를 영속적으로 저장

- SessionService:
  - LangGraph의 checkpointer를 확장
  - Agent의 상태(기억)를 어디에, 어떻게 저장하고 불러올지를 정의하는 저장소 인터페이스
  - ADK는 다양한 SessionService 구현체를 제공

- InMemorySessionService (인메모리 방식):
  - Agent의 모든 세션 정보를 서버의 RAM에 저장
  - 빠르고 설정이 간편하나 서버가 재시작되면 모든 기억이 사라짐

- GCSSessionService (클라우드 스토리지 방식):
  - Agent의 세션 상태를 Google Cloud Storage(GCS) 버킷에 파일 저장
  - 서버가 재시작되어도 기억이 보존, 여러 서버 인스턴스에서 동일한 세션 정보 공유

Session, State, Memory

클라우드 기반 Stateful Agent 아키텍처

- 사용자 A가 서버 1에 첫 번째 요청
- 서버 1은 session_A를 생성하여 GCS에 저장
- 사용자 A의 두 번째 요청이 로드 밸런서에 의해 서버 2로 전달
- 서버 2는 session_A ID를 사용하여 GCS에서 상태를 불러옴
- 자신이 첫 번째 요청을 처리했던 것 마냥 대화를 이어감
- 코드 수정 없이 InMemory에서 GCS로 전환, 프로토타입을 프로덕션으로 변경 가능