Checkpoint, Interrupt, Streaming

Sub-Graph

LangGraph 만들기

① Graph Builder 생성 – 그래프 설계 도구, Node, Edge, State 정의/추가

② compile() – 실행 가능한 app 생성

③ Sub-graph 연결 – 큰 그래프 안에 다른 그래프를 노드처럼 넣기

- START – 그래프 시작 노드, `graph.add_edge(START, "node")` ≈ `graph.set_entry_node("node")`
- END – 그래프 종료 노드

State, Node 함수 정의

```
class MyState(TypedDict):
    log: List[str]
    sub: str
    done: bool

def step1(state: MyState) -> MyState:
    state["log"].append("step1 done")
    return state

def step2(state: MyState) -> MyState:
    state["log"].append("step2 done")
    return state
```

Graph Builder / App 생성

```
builder = StateGraph(MyState)    # Graph Builder
builder.add_node("step1", step1)
builder.add_node("step2", step2)

builder.add_edge(START, "step1")
builder.add_edge("step1", "step2")
builder.add_edge("step2", END)

app = builder.compile()   # 실행 가능 app 생성

init_state: MyState = {"log": []}
print(app.invoke(init_state))  # 그래프 구동
```

Subgraph ① Subgraph를 구동 가능한 app로 만들어 놓는다.

```
sub = StateGraph(MyState)


def sub1(state: MyState) -> MyState:
    state["log"].append("sub1 in sub-graph")
    state["sub"] = "done"
    return state


sub.add_node(START, "sub1")
sub.add_node("sub1", sub1)
sub.add_edge("sub1", END)


sub_app = sub.compile()
```

Subgraph ② Main Graph Node 함수 정의

```
def start(state: MyState) -> MyState:
    state["log"].append("start in main")
    return state


def finish(state: MyState) -> MyState:
    state["log"].append("finish in main")
    state["done"] = True
    return state
```

Subgraph ③ Main Graph에 Subgraph 넣기

```
main = StateGraph(MyState)
main.add_node("start", start)
main.add_node("call_sub", sub_app)   # sub-graph를 하나의 노드처럼
main.add_node("finish", finish)

main.add_edge(START, "start")
main.add_edge("start", "call_sub")
main.add_edge("call_sub", "finish")
main.add_edge("finish", END)

main_app = main.compile()
init_state: MyState = {"log": []}
print(main_app.invoke(init_state))
```

```
{
  'log': [
    'start in main',
    'sub1 in sub-graph',
    'finish in main'
  ],
  'sub': 'done',
  'done': True
}
```

2

State & Reducer

LangGraph State

```
class MyState(TypedDict):
    messages: Annotated[List[str], add_messages(또는, operator.add)]
    sub: str
    done: bool
```

① Annotated 필드에 operator.add 를 사용하면 한 노드에서 현재 state만 반환해도 LangGraph가 history를 누적해서 관리해준다.

② Annotated 필드에 add_messages 를 사용하면 LangGraph가 history 관리 뿐만 아니라 특정 message를 id를 이용하여 update(지우고 새로 작성) 할 수 있게 해준다.(id를 명시하지 않으면 operator.add와 같은 기능)

- `import operator`
- `from langgraph.graph.message import add_messages`

Annotated(from typing import Annotated) Field 예제

```
# 1. 나만의 Reducer 함수 정의
def keep_last_3(existing: list, new: list) -> list:
    merged = existing + new
    return merged[-3:]

# 2. State에 적용
class MyState(TypedDict):
    # LangGraph에게, 무조건 keep_last_3 함수로 업데이트하라고 지시!
    short_memory: Annotated[list, keep_last_3]

# --- 결과 ---
# 노드가 계속 데이터를 추가해도, short_memory에는 항상 3개만 남게 됩니다.
```

State 데이터 종류

데이터 종류	특징	Annotated 필요 여부	예시 코드
대화 기록 (History)	계속 쌓여야 함 (Append)	필수 (add_messages 또는 operator.add)	`messages: Annotated[ list, add_messages]`
현재 상태 (Status)	최신 값만 중요함 (Update)	불필요 (그냥 Type만 씀)	`current_step: str`
최종 답변 (Output)	마지막 결과만 필요함	불필요	`final_answer: str`
검색 결과 (Docs)	매번 새로 갈아끼워야 함	불필요	`documents: list[str]`

Checkpoint

Checkpoint 핵심 개념

- LangGraph에서 그래프를 컴파일할 때 checkpointer를 지정하면, 그래프의 각 단계(Super-step)마다 State(상태) 스냅샷을 저장

① Persistence (영속성): 대화형 AI에서 이전 대화 내용을 기억 (Short-term Memory).

② Human-in-the-loop: 실행을 특정 지점에서 멈추고, 사람이 승인하거나 데이터를 수정한 뒤 다시 실행

③ Time Travel: 과거의 특정 시점(Checkpoint)으로 돌아가서 다른 경로로 재실행하거나 디버깅

④ Fault Tolerance: 에러가 발생했을 때, 마지막 성공 지점부터 재시작

- checkpointer는 LangGraph가 제공하는 in-memory를 사용할 수도 있고, 전통적인 DB(SQLite, PostgreSQL)을 사용할 수 있다.

① MemorySavor 인메모리 – 실험용

② SQLite – 작은 규모, 개인용

③ PostgreSQL – 기업/서비스 용

왜 모든 Step마다 State를 저장하는가?

- **스토리지 비용(저렴)" vs "컴퓨팅/LLM 비용(비쌈)“**

① **재시도 비용 절감 (Fault Tolerance):**

- 10단계 중 9단계에서 에러가 났을 때, 체크포인트가 없다면 1~8단계(LLM API 호출 비용 + 시간)를 다시 써야 합니다.
- 체크포인트가 있다면 9단계부터만 다시 하면 됩니다. 저장 공간 조금 쓰는 게 LLM 토큰 값보다 훨씬 쌉니다.

② **디버깅 시간 단축 (Time Travel):**

- 개발자가 에러 원인을 찾기 위해 과거 시점으로 돌아가서 상태 확인 가능 → 개발자 인건비와 시간 절약

- **State 설계 팁**

① **State에는 대화내역, 제어변수 등 가벼운 것만 넣는다.**

② **이미지, PDF 원문 등 무거운 데이터는 외부 저장소에 두고 링크만 State에 둔다.**

실행 및 저장 순서

- Checkpointer 설정 및 실행

① 인메모리 저장소인 MemorySaver를 사용하여 compile(checkpointer=memory)로 체크포인터 연결.

② 실행할 때,

```
config={"configurable": {"thread_id": "session_1"}}  # thread_id로 세션구분
graph.invoke(inputs, config=config)
```

- 실행 및 저장 순서 (Lifecycle)

① Node 실행: 노드 함수가 실행되고 결과값(return)을 반환

② State 업데이트: 만약 Annotated가 정의되어 있다면, Annotated에 정의된 리듀서(add_messages 등)가 작동하여 결과값을 State에 반영. 정의되어 있지 않다면 반환된 값을 그대로 State에 반영.

③ Checkpoint 저장: 현재의 완벽한 State 스냅샷을 DB(또는 메모리)

④ 다음 단계 이동: 저장이 안전하게 끝난 후, 다음 에지(Edge)를 따라 다음 노드로 이동

LangGraph의 State 자동 업데이트

```
class State(TypedDict):
    messages: Annotated[List[str], add_messages]
```

① 노드에서 state를 반환하면, return { "messages": ["새로운 메시지 입니다“] }

② LangGraph엔진이 반환된 값을 낚아챈다.

③ 엔진에서 기존 state와 현재 반환된 state를 add_message를 이용하여 reduce한다.

④ 합쳐진 결과를 자동으로 checkpoint에 저장하고 다음 노드로 넘어간다.

❖ Checkpoint에 저장되는 내용은 messages뿐만 아니라 메타정보 저장함(자세한 것은 LangGraph참조)

❖ Interrupt상황에서 자동이 아니라 수동으로 개발자가 state update를 할 수 있다. (Interrupt 슬라이드 참조)

4

Interrupt

Interrupt

- 목적: 그래프 엔진이 노드를 실행할 때, 사용자의 의견을 묻고 그 결과를 받아 진행할 필요가 있을 경우 interrupt를 사용한다. → 엔진이 그래프 실행을 멈추고 제어권을 그래프를 호출했던 바깥쪽 코드(client code)로 넘긴다.

- Interrupt를 하는 방식은 두 가지가 있다.

① interrupt_before

- 그래프를 compile할 때, 특정 노드 진입 전에(즉, 어떤 노드를 끝내고) 엔진이 그래프 실행을 멈추고 사용자 의견을 받는다.

```
graph = builder.compile(
    checkpointer=memory,
    interrupt_before=["sender"]  # “sender” 노드에서 interrupt 하라!
)
```

② interrupt()

- 노드 안에서 answer = interrupt(question)
- client code에서 interrupt를 capture함

① interrupt_before 방식

- 시나리오: "작성된 초안 수정하기“

① [Stop] Node A(초안 작성)가 끝나고 Node B(이메일 전송) 직전에 멈춤

② [Human] 사용자가 초안이 마음에 안 들어서 수정을 원함

③ [Action] 이때 개발자는 그래프의 update_state() 함수를 사용하여 State 자체를 바꿈

- 기존 State: {"msg": "엉망인 초안"}
- 수정된 State: {"msg": "완벽한 초안"}

④ [Resume] graph.invoke(None, config)으로 재개

⑤ [Next Node] Node B가 시작

- Node B는 State를 읽음
- Node B 입장: "앞 단계(Node A)가 '완벽한 초안'을 넘겨줬다"라고 생각 (사실은 사람이 바꾼 초안을 보고 있음)

- 결론: interrupt_before 방식에서 정보를 반영하는 방법은 "State를 바꿔치기(update_state) 하는 것임 "

update_state

```
class State(TypedDict):
    messages: Annotated[List[str], add_messages]


graph.update_state(
    config,
    {"messages": [updated_msg]},
    as_node="drafter"
)
```

- updated_msg는 client code에서 만들어진(사용자 입력) str
- drafter는 interrupt_before("node-name") 걸리기 직전의 노드이름이다. → 즉, 엔진이 resume할 때, state가 마치 drafter가 만든 state로 여겨지도록 하는 것임
- graph.invoke(None, config) # 그래프를 다시 시작 invoke 할 때, state는 None으로 함!

② interrupt() 방식

- 함수 내부에서 멈추는 interrupt()는 State를 건드리지 않고 "값(Value)"을 직접 전달

① [Stop] Node B 내부에서 user_input = interrupt() 줄에서 멈춤

② [Action] 개발자가 Command(resume="새로운 정보")를 전달

③ [Resume] user_input 변수에 "새로운 정보"가 할당되면서 코드가 이어짐

- user_input = interrup(arg) 를 하면, 클라이언트 코드에 arg를 전달할 수 있다.

- 그래프 재개는 다음과 같이 invoke한다.

```
result = graph.invoke(
        Command(resume="새로운 정보"),
        config
    )
```

interrup_before 동작 원리

① 첫 구동

- `class State(TypedDict):`
  `messages: Annotated[List[str], add_messages]`
- `graph = builder.compile(checkpointer=memory, interrupt_before=["sender"])`
- `config = {"configurable": {"thread_id": "session_1"}}`
- `result = graph.invoke(inputs, config)`

② 그래프에서 invoke를 구동 시키면, interrupt 직전 노드까지 수행되고 그 때의 state를 반환한다.

③ 엔진은 중단되고, 제어권이 클라이언트 코드로 넘어간다.

④ 클라이언트 코드에서 현재 상태를 확인할 수 있다.

- `snapshot = graph.get_state(config)`
- `current_msg = snapshot.values["messages"][-1]`

⑤ 클라이언트 코드에서 사용자 의견을 받아서 현재 상태를 업데이트 한다.

- `graph.update_state(config, {"messages": [updated_msg]}, as_node="drafter")`

⑥ 클라이언트 코드에서 그래프 실행을 재개한다.

- `result = graph.invoke(None, config=config)`

interrupt(arg) 동작 원리

① 첫 구동

- builder = StateGraph(dict) # 간단하게 dict state를 사용하였다.
- graph = builder.compile(checkpointer=memory)
- config = {"configurable": {"thread_id": "session_1"}}
- result = graph.invoke({"name": ""}, config)

② 엔진이 그래프를 실행하다가 노드 안에서 interrupt()를 만난다.

- user_name = interrupt("이름이 뭔가요?")

③ 엔진은 중단되고 interrupt정보는 result에 담고 제어권은 클라이언트 코드로 넘긴다.

④ 클라이언트 코드는 result에서 interrupt 정보를 얻고, 사용자 입력을 받아 그래프 실행을 재개한다.

- ```
  if "__interrupt__" in result:
      interrupt_info = result["__interrupt__"][0].value
      print(f"멈춤 감지! 요청 내용: '{interrupt_info}'")
      user_input = input(f">> {interrupt_info} : ")
      result = graph.invoke(Command(resume=user_input), config)
  ```

⑤ 엔진은 중단되었던 그 자리에서 user_input을 반환받고 user_name에 할당하고 계속 노드를 실행한다.
  ```

stream

5

Stream

- **LangGraph의 stream 메서드는 그래프의 실행 과정을 단계별(Step-by-Step)로 모니터링 해주는 기능**
  - 일반적인 함수 호출이 "결과가 나올 때까지 기다리는 것" 이라면, stream은 " 각 단계가 완료될 때마다 즉시 보고를 받는 것 " 과 같음
  - 이것은 긴 작업 흐름을 가진 Agent의 동작을 모니터링하거나, 사용자에게 중간 진행 상황을 보여 줄 때 사용할 수 있음

- **invoke vs stream 비교**
- invoke (동기 실행): 그래프의 시작부터 끝까지 다 실행한 뒤, 최종 결과(Final State)만 한 번에 반환(return)
- stream (스트리밍 실행): 그래프의 각 노드(Node)가 실행을 마칠 때마다, 그 결과를 즉시 반환(yield)
  - `graph.stream(inputs, config)`을 실행하면 Python의 generator를 반환 → for 문으로 순회하면 각 단계의 실행 결과를 실시간으로 받을 수 있음
  - 기본적으로 stream은 방금 실행된 노드의 이름과 그 노드가 업데이트한 상태(State)를 dict 형태로 반환
  - `{ "노드이름": { "업데이트된_상태_키": "값" } }`

간단 예시

```
# graph.stream()은 각 노드가 끝날 때마다 멈추고 결과를 yield 합니다.
for event in graph.stream(inputs):
    # event는 딕셔너리입니다. (key: 노드이름, value: 노드의 리턴값)
    for node_name, value in event.items():
        print(f"[{node_name} 실행됨]")
        print(f"  -> 업데이트 내용: {value}")
        print("---")
```

stream 모드(stream_mode)

- stream() 메서드는 stream_mode 파라미터를 통해 무엇을 출력할지 결정할 수 있음

① stream_mode="updates" (기본값):
  - 방금 실행된 노드가 변경한 부분만 보여줍니다. (위 예제와 동일)

② stream_mode="values":
  - 각 단계가 끝난 시점의 **전체 State(모든 누적 데이터)**를 보여줍니다.
  - 예: 1단계 때는 1단계 로그만, 2단계 때는 1+2단계 로그 전체를 보여줌.

- 주의사항: LLM 토큰 스트리밍과는 다름

- 여기서 말하는 graph.stream은 Graph의 노드 단위 스트리밍를 의미 (Node A 완료 -> Node B 완료).
- ChatGPT처럼 글자가 찍히는 효과는 Token Streaming이며, 이를 위해서 (a) LangGraph의 astream_events (비동기 이벤트 스트리밍)를 사용하거나 (b) LLM을 처리하는 노드 내부에서 별도의 처리 해주어야 함