LangGraph

Agent 개발 Framework – https://www.langchain.com/langgraph

LangChain

LangGraph – AI 에이전트 개발 Framework

- **LangGraph는 다음을 목표로 하는 전문가용 프레임워크**

1. 멀티 에이전트
2. 파일 저장 기반 durable memory
3. LLM tool-calling + graph orchestration
4. 서버 배포 / 흐름 재개(Resumable) / 체크포인트
5. 브라우저 기반 시각화
6. 관측가능성(Observability)
7. Async 처리 / concurrency

→ 기업용 / 대규모 에이전트 개발을 위한 구조

LangGraph 구성 요소

1

핵심 구성 요소

- LangGraph는 Agent의 상태(State)를 가진 workflok를 직접 구성하는 framework
- LLM + Memory + Tool + Branching + Loop 이런 것들을 그래프 형태로 조립할 수 있게 해주는 시스템

1. State (상태 모델)
2. Node
3. Edges
4. Conditional Edge
5. Entry Point
6. Subgraph / Nested Graph
7. Interrupt / Streaming / Checkpointing

1. State(상태 모델) → 가장 중요

1. 그래프 전체가 공유하는 “데이터 저장소”

2. 개발자가 직접 설계함 (messages, memory, tool_result, metadata 등)

```
class State(TypedDict):
    messages: List[dict]
    memory: dict
    tool_result: dict | None
```

3. 노드들은 모두 이 State를 읽고 수정하여 반환함

- 모든 Node는 State 입력 → State 출력 구조

4. LangGraph의 본질 = ‘상태를 가진 에이전트 로직을 그래프로 표현’

- “State-driven agent”라는 철학이 핵심.

2. Node

- Node
- State를 입력받아 State를 반환하는 함수: State → State 함수
- LLM Node, Tool Node, Memory Node, Summary Node 등

```
def llm_node(state: State) -> State:
    msg = client.chat.completions.create(...)
    return { **state, "messages": state["messages"] + [msg] }
```

- 노드는 '함수 ' 이다.
- Graph도 하나의 node가 될 수 있다.
- 노드 이름은 문자열로 그래프에 등록한다.
- 한 노드에서 다른 노드로 가는 것은 에지 / 조건부 에지가 결정한다.
- 노드들을 조합하면 에이전트가 된다.

3. Edge

- Edge는 노드 간 연결 `graph.add_edge("tool", "llm")`

① tool_node(state_in) 실행

② tool_node가 새로운 state_out 을 반환

③ LangGraph는 이 state_out을 그대로 llm_node(state_out) 에 전달

④ llm_node는 그걸 가지고 다시 LLM 호출을 진행

```
tool_node:  State_in  →  State_out
llm_node:   (받은) State_out → State_next
```

4. Conditional Edge

- **Conditional Edge(조건 분기)**
- 가장 LangGraph스러운 부분 – 지능적 기능
- state를 인자로 받아 다음 노드를 선택하는 함수 작성.

```
def route(state: State):
    return "tool" if state["tool_result"] else END

graph.add_conditional_edges("llm", route)
```

- **Node 자체는 뇌, 조건분기는 뼈대.**

5-6. Entry Point, Subgraph / Nested Graph

- **Entry Point: 그래프의 첫 노드**
- ReAct라면 대부분 LLM을 entry로 설정: `graph.set_entry_point("llm")`

- **Subgraph / Nested Graph**
- LangGraph는 노드를 함수뿐 아니라 "하나의 그래프 전체를 노드처럼 넣을 수 있다."

```
planner = build_planner_graph()
graph.add_node("planner", planner)
```

- **복잡한 Agent → 여러 작은 그래프 조합**
- **실전 Agent 설계에서는 Subgraph가 사실상 필수.**

7. Interrupt / Streaming / Checkpointing

- **Interrupt**
  - 특정 노드에서 사람 입력을 기다리기(예: human-in-the-loop approval)

- **Streaming**
  - LLM 응답을 스트리밍으로 전달(일반 ChatGPT처럼)

- **Checkpointing**
  - 그래프 실행 과정 전체를 저장

→ 복구 가능

→ 여러 경로 탐색 가능

→ state snapshot 관리

  - 장시간 workflow에 필수

전체 요약

- State → 그래프 전체의 데이터 모델
- Node → 함수(State → State)
- Edge → 노드 간 연결
- Conditional Edge → State 기반 분기

- Entry → 시작점
- Subgraph → 모듈화된 그래프
- Checkpoint/Interrupt/Stream → 실전 기능

2

간단 예제 코드

1. import

```
from langgraph.graph import StateGraph, END
from langgraph.pregel import Pregel
from typing import TypedDict, List
import json
from openai import OpenAI


client = OpenAI()
```

2. State 정의

```
class State(TypedDict):
    messages: List[dict]
    tool_result: str | None
```

- **class State(TypedDict):**
  - LangGraph 그래프가 돌면서 계속 유지할 상태의 스키마를 정의.
  - TypedDict라서 "messages" / "tool_result" 키를 가진 dictionary 형태로 취급
- **messages: List[dict]**
  - 지금까지의 대화 히스토리를 모두 저장하는 리스트.
  - OpenAI chat 포맷의 각 메시지({"role": ..., "content": ...} 등)를 그대로 리스트에 저장
- **tool_result: str | None**
  - LLM이 “tool을 사용하라”고 했을 때,
  - 어떤 tool을 어떤 인자로 호출하라고 했는지 기록해두는 곳.
  - 없으면 None, 있으면 JSON 문자열로 직렬화된 tool_call 정보.

3. 그래프 구성

```
graph = StateGraph(State)

graph.add_node("llm", llm_node)
graph.add_node("tool", tool_node)

# 첫 노드는 llm
graph.set_entry_point("llm")

# edge 정의 – LLM이 tool을 호출하면 tool node로
def route(state: State):
    if state["tool_result"] is not None:
        return "tool"
    return END

graph.add_conditional_edges("llm", route)
graph.add_edge("tool", "llm")

app = graph.compile()
```

4. 실행 예시

```
initial_state = {
    "messages": [{"role": "user", "content": "123 * 987 계산해줘"}],
    "tool_result": None
}

result = app.invoke(initial_state)

print(result["messages"][-1]["content"])
```

5. LLM Node 정의

def llm_node(state: State):

① state로부터 messages를 만들고,

② llm에 tool calling을 하고,

③ msg를 llm으로부터 받은 후에,

④ 결과에 따라서 다시 state를 만들어 State를 반환하는 작업

- tool calls일 경우

```
return {
            "messages": messages + [msg.to_dict()],
            "tool_result": json.dumps(tool_call.to_dict())
        }
```

- final answer일 경우

```
return {
        "messages": messages + [msg.to_dict()],
        "tool_result": None
    }
```

6. Tool Node 정의

```
def tool_node(state: State):
    json_load_tool = json.loads(state["tool_result"])
```

① state로부터 tool_result를 액세스하여,

② tool을 실행하고,

③ 그 result를 observation 으로 만들고 기존 message와 함께 붙여서 State를 반환하는 노드

```
    observation = {
        "role": "tool",
        "content": result,
        "tool_call_id": json_load_tool["id"],
    }

    return {
        "messages": state["messages"] + [observation],
        "tool_result": None,
    }
```

7. 실제 Tool 정의

```
def calculator(expr: str) -> str:
    return str(eval(expr))
```

기존 툴 정의 함수와 동일하게 구현하면 됨