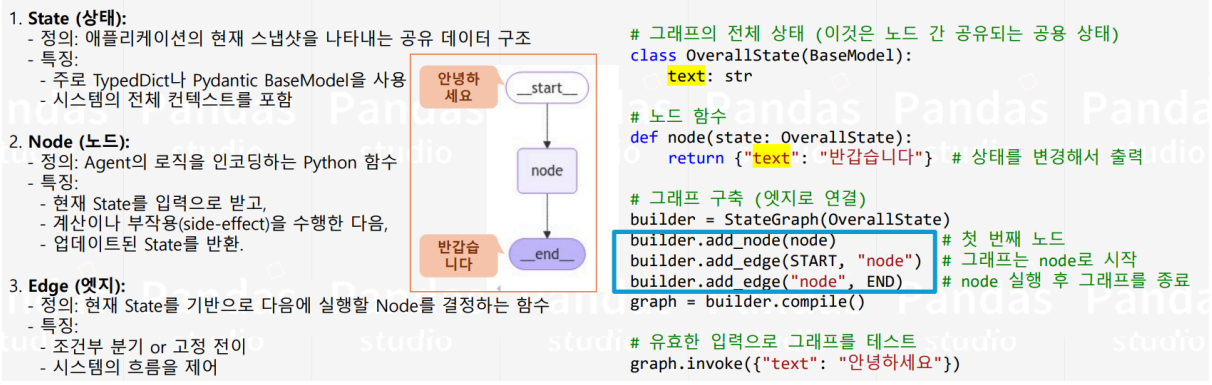
Langraph 기본기 다지기

StateGraph

- 상태(state)를 기반으로 작동하는 그래프 구조
- 복잡한 작업 흐름을 상태와 전이로 모델링해 유연하고 제어 가능한 시스템 구축
- 각 노드(node)가 특정 상태를 나타내며, edge가 상태 간 전이 조건을 정의함




```
from typing import TypedDict

# 데이터의 자료 구조 정의
# 상태 Schema 정의 - 사용자의 선호도, 추천된 메뉴, 그리고 메뉴 정보를 저장
class MenuState(TypedDict):
    user_preference: str
    recommended_menu: str
    menu_info: str
```

```
import random

def get_user_preference(state: MenuState) -> MenuState:
    print("---랜덤 사용자 선호도 생성---")
    preferences = ["육류", "해산물", "채식", "아무거나"]
```

```
    preference = random.choice(preferences)
    print(f"생성된 선호도: {preference}")
    # 업데이트할 field를 key값으로 사용해 dict 형태로 업데이트
    return {"user_preference": preference}

def recommend_menu(state: MenuState) -> MenuState:
    print("---메뉴 추천---")
    preference = state['user_preference']
    if preference == "육류":
        menu = "스테이크"
    elif preference == "해산물":
        menu = "랍스터 파스타"
    elif preference == "채식":
        menu = "그린 샐러드"
    else:
        menu = "오늘의 쉐프 특선"
    print(f"추천 메뉴: {menu}")
    return {"recommended_menu": menu}

def provide_menu_info(state: MenuState) -> MenuState:
    print("---메뉴 정보 제공---")
    menu = state['recommended_menu']
    if menu == "스테이크":
        info = "최상급 소고기로 만든 juicy한 스테이크입니다. 가
격: 30,000원"
    elif menu == "랍스터 파스타":
        info = "신선한 랍스터와 al dente 파스타의 조화. 가격: 2
8,000원"
    elif menu == "그린 샐러드":
        info = "신선한 유기농 채소로 만든 건강한 샐러드. 가격: 1
5,000원"
    else:
        info = "쉐프가 그날그날 엄선한 특별 요리입니다. 가격: 35,
000원"
    print(f"메뉴 정보: {info}")
    return {"menu_info": info}
```

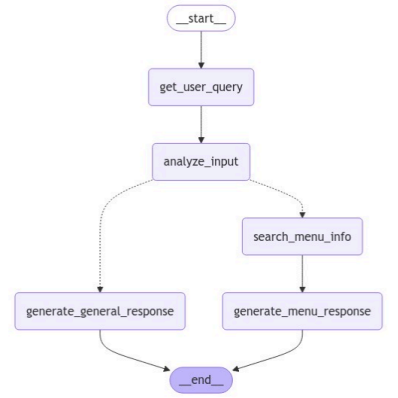
```
from langgraph.graph import StateGraph, START, END

# 그래프 빌더 생성
# MenuState를 기본 상태로 정의
builder = StateGraph(MenuState)

# 노드 추가
builder.add_node("get_preference", get_user_preference)
builder.add_node("recommend", recommend_menu)
builder.add_node("provide_info", provide_menu_info)

# 엣지 추가
builder.add_edge(START, "get_preference")
builder.add_edge("get_preference", "recommend")
builder.add_edge("recommend", "provide_info")
builder.add_edge("provide_info", END)

# 그래프 컴파일
graph = builder.compile()
```

```
# 그래프 실행

def print_result(result: MenuState):
    print("\n=== 결과 ===")
    print("선호도:", result['user_preference'])
    print("추천 메뉴:", result['recommended_menu'])
    print("메뉴 정보:", result['menu_info'])
    print("============\n")


# 초기 입력
# 초기에는 user_preference를 정의하지 않음
# random으로 정의될 것이기 때문
inputs = {"user_preference": ""}

# 여러 번 실행하여 테스트
for _ in range(2):
```

```
    result = graph.invoke(inputs)
    print_result(result)
    print("*"*100)
    print()
```

조건부 Edge를 활용한 분기 (Branching)

- 현재 상태나 입력에 따라 다음 노드를 동적으로 결정하는 연결
- 시스템의 흐름을 더 유연하고 상황에 맞게 제어
- 조건문을 사용해 다음 실행할 노드를 선택 (분기, Branching)




```python
def decide_next_step(state: MenuState):
    if state['is_menu_related']:
        return "search_menu_info"
    else:
        return "generate_general_response"
```

```python
# 조건부 엣지 추가
builder.add_conditional_edges(
    "analyze_input",
    decide_next_step,
    {
        "search_menu_info": "search_menu_info",
        "generate_general_response": "generate_general_response"
    }
)
```

```
from typing import List

# state 스키마
class MenuState(TypedDict):
    user_query: str
    is_menu_related: bool
    search_results: List[str]
    final_answer: str
```

```
from langchain_chroma import Chroma
from langchain_ollama  import OllamaEmbeddings

# 모델 초기화
embeddings_model = OllamaEmbeddings(model="bge-m3")
```

```
# Chroma 인덱스 로드
# 컬렉션 이름과 디렉토리 경로를 지정
vector_db = Chroma(
    embedding_function=embeddings_model,
    collection_name="restaurant_menu",
    persist_directory="./chroma_db",
)
```

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

# LLM 모델
llm = ChatOpenAI(model="gpt-4o-mini")

# 사용자가 입력한 텍스트로 user_query 업데이트
def get_user_query(state: MenuState) -> MenuState:
    user_query = input("무엇을 도와드릴까요? ")
    return {"user_query": user_query}

# 사용자 입력 분석
# state를 입력받아 업데이트 후 반환
def analyze_input(state: MenuState) -> MenuState:
    analyze_template = """
    사용자의 입력을 분석하여 레스토랑 메뉴 추천이나 음식 정보에 관한
질문인지 판단하세요.

    사용자 입력: {user_query}

    레스토랑 메뉴나 음식 정보에 관한 질문이면 "True", 아니면 "Fals
e"로 답변하세요.

    답변:
    """
    analyze_prompt = ChatPromptTemplate.from_template(analy
ze_template)
    analyze_chain = analyze_prompt | llm | StrOutputParser
```

```
()

    result = analyze_chain.invoke({"user_query": state['use
r_query']})
    is_menu_related = result.strip().lower() == "true"

    return {"is_menu_related": is_menu_related}

# 유사도 기반으로 답변 출력
# 검색 문서 중 리스트 축약을 통해 텍스트 부분만 추출
# 결과를 search_result field에 저장
def search_menu_info(state: MenuState) -> MenuState:
    # 벡터저장소에서 최대 2개의 문서를 검색
    results = vector_db.similarity_search(state['user_quer
y'], k=2)
    search_results = [doc.page_content for doc in results]
    return {"search_results": search_results}

# 메뉴 정보 생성
def generate_menu_response(state: MenuState) -> MenuState:
    response_template = """
    사용자 입력: {user_query}
    메뉴 관련 검색 결과: {search_results}

    위 정보를 바탕으로 사용자의 메뉴 관련 질문에 대한 상세한 답변을
생성하세요.
    검색 결과의 정보를 활용하여 정확하고 유용한 정보를 제공하세요.

    답변:
    """
    response_prompt = ChatPromptTemplate.from_template(resp
onse_template)
    response_chain = response_prompt | llm | StrOutputParse
r()

    final_answer = response_chain.invoke({"user_query": sta
te['user_query'], "search_results": state['search_result
s']})
```

```
    print(f"\n메뉴 어시스턴트: {final_answer}")

    return {"final_answer": final_answer}

# 일반적인 답변을 생성할 수 있도록 정의
def generate_general_response(state: MenuState) -> MenuStat
e:
    response_template = """
    사용자 입력: {user_query}

    위 입력은 레스토랑 메뉴나 음식과 관련이 없습니다.
    일반적인 대화 맥락에서 적절한 답변을 생성하세요.

    답변:
    """
    response_prompt = ChatPromptTemplate.from_template(resp
onse_template)
    response_chain = response_prompt | llm | StrOutputParse
r()

    final_answer = response_chain.invoke({"user_query": sta
te['user_query']})
    print(f"\n일반 어시스턴트: {final_answer}")

    return {"final_answer": final_answer}
```

```
from typing import Literal

# 함수명을 통해 출력을 하게 됨
# 메뉴 관련 질문일 경우 분기를 통해 답변을 출력하도록 정의
def decide_next_step(state: MenuState) -> Literal["search_m
enu_info", "generate_general_response"]:
    if state['is_menu_related']:
        return "search_menu_info"
    else:
        return "generate_general_response"
```

```
from langgraph.graph import StateGraph, START, END

# 그래프 구성
builder = StateGraph(MenuState)

# 노드 추가
builder.add_node("get_user_query", get_user_query)
builder.add_node("analyze_input", analyze_input)
builder.add_node("search_menu_info", search_menu_info)
builder.add_node("generate_menu_response", generate_menu_re
sponse)
builder.add_node("generate_general_response", generate_gene
ral_response)

# 엣지 추가
builder.add_edge(START, "get_user_query")
builder.add_edge("get_user_query", "analyze_input")

# 조건부 엣지 추가
builder.add_conditional_edges(
    "analyze_input",
    decide_next_step,
    {
        "search_menu_info": "search_menu_info",
        "generate_general_response": "generate_general_resp
onse"
    }
)

builder.add_edge("search_menu_info", "generate_menu_respons
e")
builder.add_edge("generate_menu_response", END)
builder.add_edge("generate_general_response", END)

# 그래프 컴파일
graph = builder.compile()
```

```
while True:
    initial_state = {'user_query':''}
    graph.invoke(initial_state)
    continue_chat = input("다른 질문이 있으신가요? (y/n): ").l
ower()
    if continue_chat != 'y':
        print("대화를 종료합니다. 감사합니다!")
        break
```