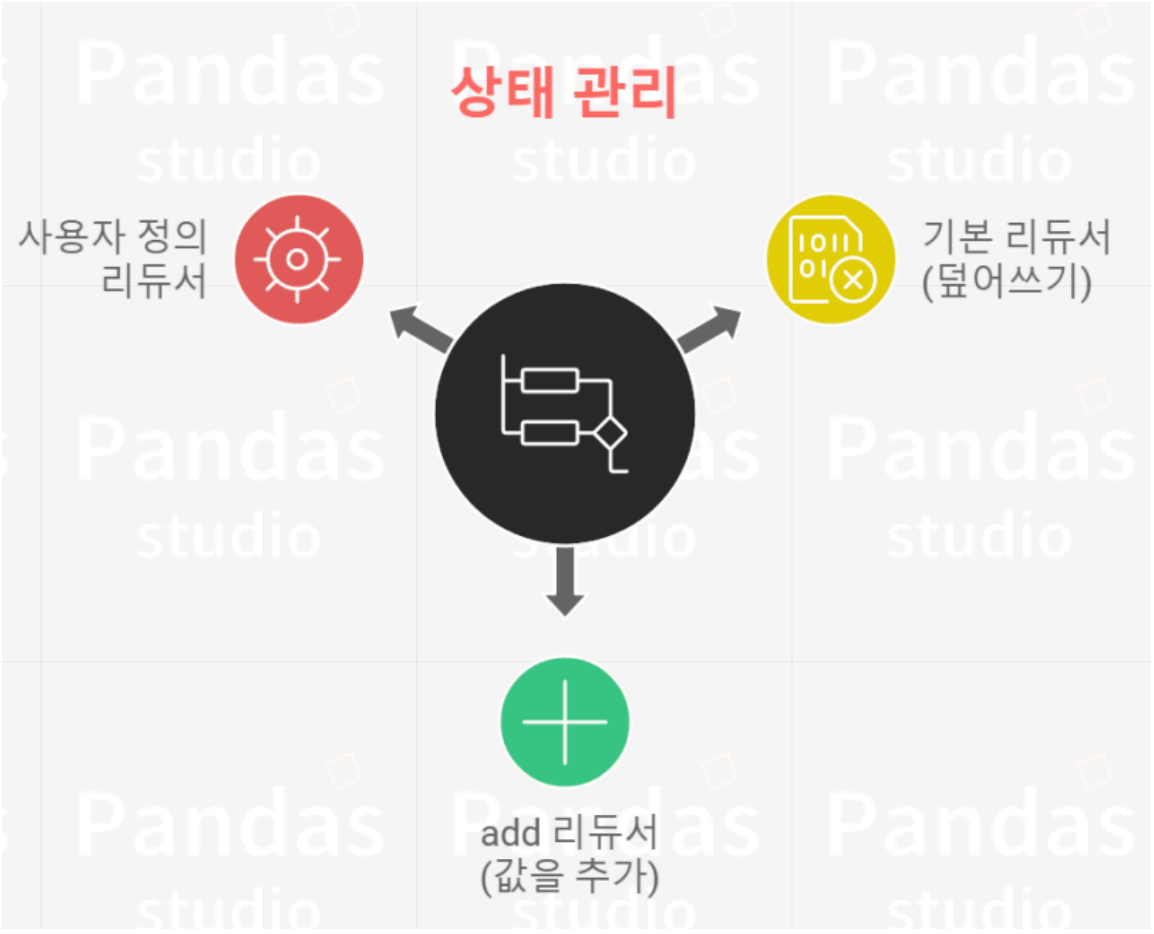
LangGraph 기본기 다지기 2

State Reducer

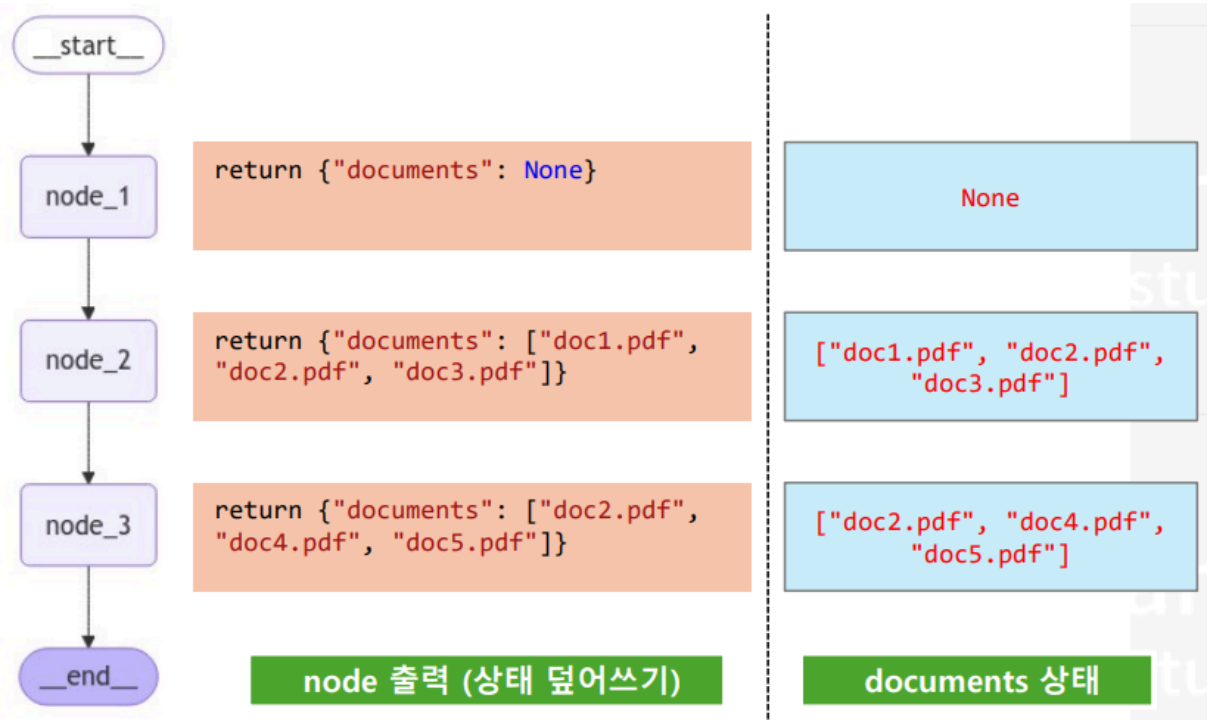
- 상태 업데이트를 관리하는 함수
- 그래프의 각 노드의 출력을 그래프의 전체 상태에 통합하는 방법을 정의
- 동작 방식
  - 각 노드 반환값은 해당 상태 키의 이전 값을 **덮어쓰기** 저장 (**기본** 리듀서)
  - 메시지 리스트 등에서 이전 상태에 새로운 값을 추가할 때 사용 (**add** 리듀서)
  - 중복 제거, 정렬 등 특수한 상태 관리를 하는 사용자 정의 리듀서 지원 (**custom** 리듀서)



Reducer를 별도로 지정하지 않은 경우

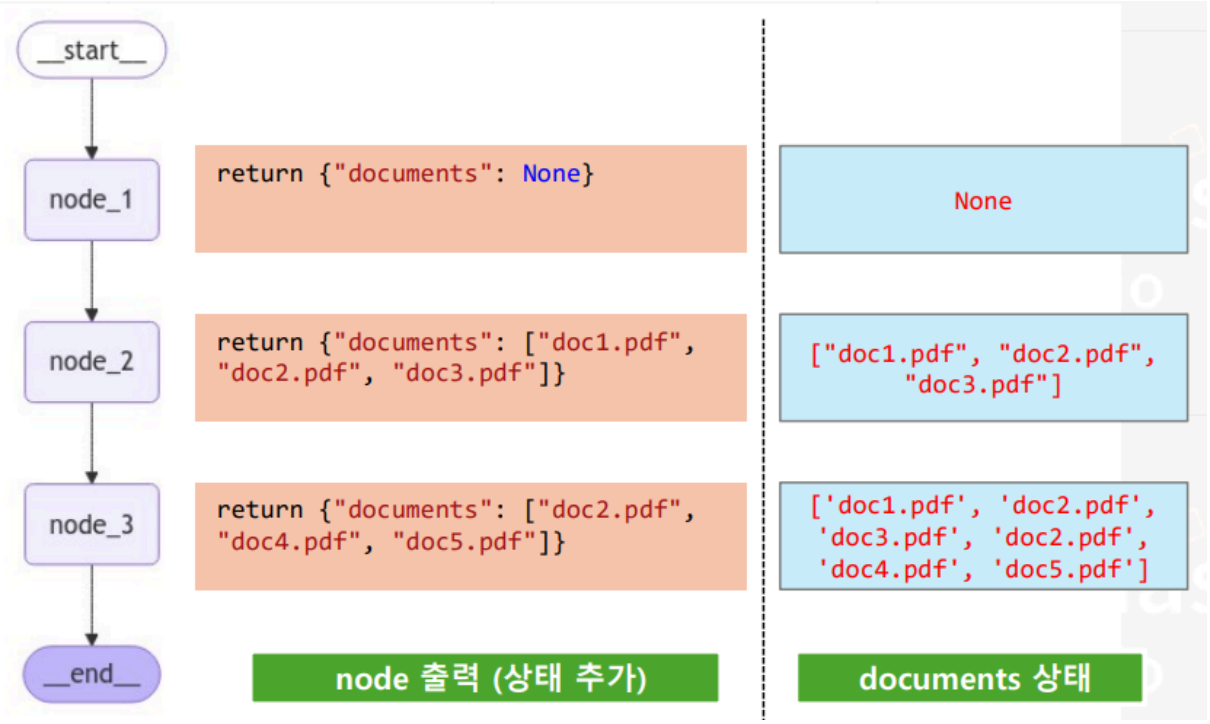
```
class DocumentState(TypedDict):
    query: str
    documents: List[str]
```

- 덮어쓰기 방식으로 관리할 경우 DocumentState라는 상태 정의



Reducer를 별도로 지정하는 경우

```
class ReducerSTate(TypedDict):
    query: str
    documents: Annotated[List[str], add]
```



- Annotated : 리듀서 추가하는 함수
- add 함수를 추가했기에 기존 state를 덮어쓰지 않고 마지막 부분에 리스트의 원소로 추가됨

Reducer 활용하기

1. Reducer를 별도로 지정하지 않은 경우

```
from typing import TypedDict, List
from langgraph.graph import StateGraph, START, END
from IPython.display import Image, display

# 상태 정의
class DocumentState(TypedDict):
    query: str
    documents: List[str]

# Node 1: query 업데이트
def node_1(state: DocumentState) -> DocumentState:
    print("---Node 1 (query update)---")
    query = state["query"]
```

```
    return {"query": query}

# Node 2: 검색된 문서 추가
def node_2(state: DocumentState) -> DocumentState:
    print("---Node 2 (add documents)---")
    return {"documents": ["doc1.pdf", "doc2.pdf", "doc3.
pdf"]}

# Node 3: 추가적인 문서 검색 결과 추가
def node_3(state: DocumentState) -> DocumentState:
    print("---Node 3 (add more documents)---")
    return {"documents": ["doc2.pdf", "doc4.pdf", "doc5.
pdf"]}


# 그래프 빌드
builder = StateGraph(DocumentState)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

# 논리 구성
builder.add_edge(START, "node_1")
builder.add_edge("node_1", "node_2")
builder.add_edge("node_2", "node_3")
builder.add_edge("node_3", END)

# 그래프 실행
graph = builder.compile()

# 그래프 시각화
display(Image(graph.get_graph().draw_mermaid_png()))
```

```
# 초기 상태
initial_state = {"query": "채식주의자를 위한 비건 음식을 추천
해주세요."}

# 그래프 실행
```

```
final_state = graph.invoke(initial_state)

# 최종 상태 출력
print("최종 상태:", final_state)
```

2. Reducer를 별도로 지정한 경우

```
from operator import add
from typing import Annotated, TypedDict

class ReducerState(TypedDict):
    query: str
    documents: Annotated[List[str], add]

# Node 1: query 업데이트
def node_1(state: ReducerState) -> ReducerState:
    print("---Node 1 (query update)---")
    query = state["query"]
    return {"query": query}

# Node 2: 검색된 문서 추가
def node_2(state: ReducerState) -> ReducerState:
    print("---Node 2 (add documents)---")
    return {"documents": ["doc1.pdf", "doc2.pdf", "doc3.
pdf"]}

# Node 3: 추가적인 문서 검색 결과 추가
def node_3(state: ReducerState) -> ReducerState:
    print("---Node 3 (add more documents)---")
    return {"documents": ["doc2.pdf", "doc4.pdf", "doc5.
pdf"]}

# 그래프 빌드
builder = StateGraph(ReducerState)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)
```

```
# 논리 구성
builder.add_edge(START, "node_1")
builder.add_edge("node_1", "node_2")
builder.add_edge("node_2", "node_3")
builder.add_edge("node_3", END)

# 그래프 실행
graph = builder.compile()

# 그래프 시각화
display(Image(graph.get_graph().draw_mermaid_png()))
```

```
# 초기 상태
initial_state = {"query": "채식주의자를 위한 비건 음식을 추천
해주세요."}

# 그래프 실행
final_state = graph.invoke(initial_state)

# 최종 상태 출력
print("최종 상태:", final_state)
```

3. Custom Reducer 사용
    - 상태 업데이트가 기본적인 덮어쓰기나 병합만으로 해결되지 않을 때 사용
    - 중복 제거, 최대/최소 값 유지, 조건부 병합 등의 특정 비즈니스 로직이 필요한 경우 사용

```
from typing import TypedDict, List, Annotated

# Custom reducer: 중복된 문서를 제거하며 리스트 병합
def reduce_unique_documents(left: list | None, right: li
st | None) -> list:
    """Combine two lists of documents, removing duplicat
es."""
    if not left:
        left = []
    if not right:
```

```
        right = []
    # 중복 제거: set을 사용하여 중복된 문서를 제거하고 다시 lis
t로 변환
    return list(set(left + right))

# 상태 정의 (documents 필드 포함)
class CustomReducerState(TypedDict):
    query: str
    documents: Annotated[List[str], reduce_unique_docume
nts]  # Custom Reducer 적용


# Node 1: query 업데이트
def node_1(state: CustomReducerState) -> CustomReducerSt
ate:
    print("---Node 1 (query update)---")
    query = state["query"]
    return {"query": query}

# Node 2: 검색된 문서 추가
def node_2(state: CustomReducerState) -> CustomReducerSt
ate:
    print("---Node 2 (add documents)---")
    return {"documents": ["doc1.pdf", "doc2.pdf", "doc3.
pdf"]}

# Node 3: 추가적인 문서 검색 결과 추가
def node_3(state: CustomReducerState) -> CustomReducerSt
ate:
    print("---Node 3 (add more documents)---")
    return {"documents": ["doc2.pdf", "doc4.pdf", "doc5.
pdf"]}

# 그래프 빌드
builder = StateGraph(CustomReducerState)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)
```

```
# 논리 구성
builder.add_edge(START, "node_1")
builder.add_edge("node_1", "node_2")
builder.add_edge("node_2", "node_3")
builder.add_edge("node_3", END)

# 그래프 실행
graph = builder.compile()

# 그래프 시각화
display(Image(graph.get_graph().draw_mermaid_png()))
```

```
# 초기 상태
initial_state = {"query": "채식주의자를 위한 비건 음식을 추천
해주세요.", "documents": []}

# 그래프 실행
final_state = graph.invoke(initial_state)

# 최종 상태 출력
print("최종 상태:", final_state)
```

MessageGraph

- LangChain의 ChatModel을 위한 특수한 형태의 StateGraph
- Message 객체 목록 (HumanMessage, AIMessage 등)을 입력으로 처리
- 대화 기록을 효과적으로 관리하고 활용 가능 (자연스러운 대화 흐름, 컨텍스트 활용 답변)
- Messages State 정의
  1. 대화 기록을 그래프 상태에 메시지 목록으로 저장
  2. Message 객체 목록을 저장하는 messages 키를 추가
  3. 이 키에 리듀서 함수를 추가해 메시지 업데이트를 관리

- operator.add: 새 메시지를 기존 목록에 단순히 추가
- add_messages : 기존 메시지 업데이트 처리(메시지 ID 추적)

```
from typing import Annotated
from langchain_core.messages import AnyMessage
from langgraph.graph.message import add_messages

# 기본 State 초기화 방법 사용
class GraphState(TypedDict):
    messages: Annotated[list[AnyMessage], add_messages]
```

```
# LangGraph MessagesState라는 미리 만들어진 상태를 사용
from langgraph.graph import MessagesState
from typing import List
from langchain_core.documents import Document

class GraphState(MessageState):
    # messages 키는 기본 제공
    # 다른 키를 추가하고 싶을 경우 아래 주석과 같이 적용 가능
    documents: List[Document]
    grade: float
    num_generation: int
```

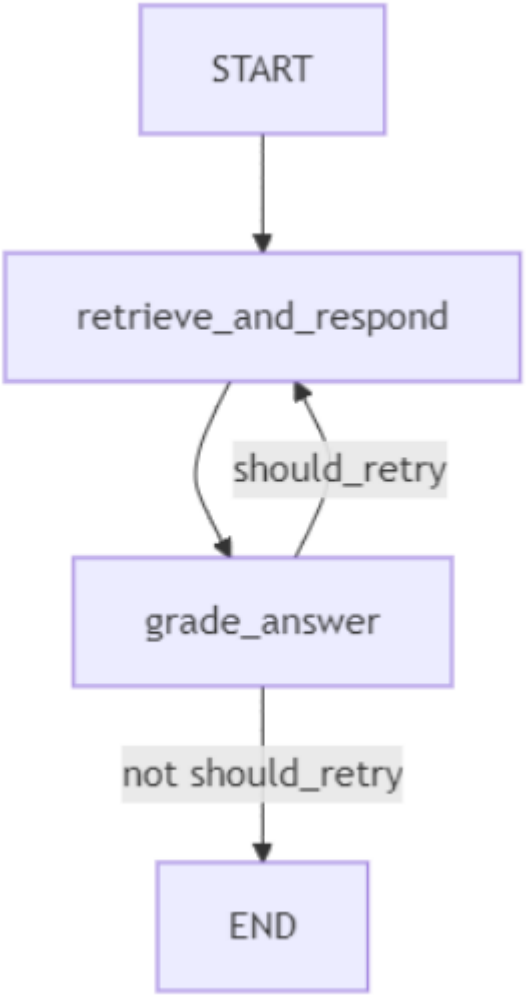
- 주의사항
  - 메모리 관리 : 대화 기록이 길어질 경우 오래된 메시지를 제거하는 로직을 추가
  - 프라이버시 : 민감한 정보가 포함된 메시지는 적절히 처리
  - 성능 최적화 : 메시지 처리 로직이 복잡해질 경우, 성능 최적화 필요

조건부 엣지(Edge)를 활용 : 반복(Iteration / Feddback Loop)

- 조건부 엣지를 활용해 반복적인 작업을 수행하는 그래프
- 평가 결과를 바탕으로 다시 응답을 생성하는 과정이 반복
- grade_answer에서 retrieve_and_respond로 돌아가는 구조는 피드백 루프를 형성

```
def should_retry(state: GraphState):
    if state["grade"] < 0.7:
        return "retrieve_and_respond"
    else:
        return "generate"
```

```
# 조건부 엣지 추가
builder.add_conditional_edges(
    "grade_answer",
    should_retry,
    {
        "retrieve_and_respond" : "retrieve_and_respond",
        "generate": END
    }
)
```



LangChain 기반으로 RAG chain 구성

- 메뉴 검색을 위한 벡터 저장소를 초기화 (기존 저장소를 로드)
- LangChain Runnable로 구현

```
from langchain_chroma import Chroma
from langchain_ollama  import OllamaEmbeddings
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, AIMessage
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough, R
unnableLambda
```

```
embeddings_model = OllamaEmbeddings(model="bge-m3")

# Chroma 인덱스 로드
vector_db = Chroma(
    embedding_function=embeddings_model,
    collection_name="restaurant_menu",
    persist_directory="./chroma_db",
)

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

# RAG 체인 구성
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

system = """
You are a helpful assistant. Use the following context to a
nswer the user's question:

[Context]
{context}
"""

prompt = ChatPromptTemplate.from_messages([
    ("system", system),
    ("human", "{question}")
])

# 검색기 정의
retriever = vector_db.as_retriever(
    search_kwargs={"k": 2}
)

# RAG 체인 구성
rag_chain = (
    {"context": retriever | format_docs, "question": Runnab
lePassthrough()}
```

```
    | prompt
    | llm
    | StrOutputParser()
)

# RAG 체인 실행
query = "채식주의자를 위한 메뉴를 추천해주세요."
response = rag_chain.invoke(query)

# 답변 출력
print(response)
```

Node

```
# RAG 수행 함수 정의
def retrieve_and_respond(state: GraphState):
    last_human_message = state['messages'][-1]

    # HumanMessage 객체의 content 속성에 접근
    query = last_human_message.content

    # 문서 검색
    retrieved_docs = retriever.invoke(query)

    # 응답 생성
    response = rag_chain.invoke(query)

    # 검색된 문서와 응답을 상태에 저장
    return {
        "messages": [AIMessage(content=response)],
        "documents": retrieved_docs
    }
```

```
from pydantic import BaseModel, Field

class GradeResponse(BaseModel):
    "A score for answers"
```

```
    score: float = Field(..., ge=0, le=1, description="A sc
ore from 0 to 1, where 1 is perfect")
    explanation: str = Field(..., description="An explanati
on for the given score")

# 답변 품질 평가 함수
def grade_answer(state: GraphState):
    messages = state['messages']
    question = messages[-2].content
    answer = messages[-1].content
    context = format_docs(state['documents'])

    grading_system = """You are an expert grader.
    Grade the following answer based on its relevance and a
ccuracy to the question, considering the given context.
    Provide a score from 0 to 1, where 1 is perfect, along
with an explanation."""

    grading_prompt = ChatPromptTemplate.from_messages([
        ("system", grading_system),
        ("human", "[Question]\n{question}\n\n[Context]\n{co
ntext}\n\n[Answer]\n{answer}\n\n[Grade]\n")
    ])

    grading_chain = grading_prompt | llm.with_structured_ou
tput(schema=GradeResponse)

    grade_response = grading_chain.invoke({
        "question": question,
        "context": context,
        "answer": answer
    })

    # 답변 생성 횟수를 증가
    num_generation = state.get('num_generation', 0)
    num_generation += 1
```

```
    return {"grade": grade_response.score, "num_generatio
n": num_generation}
```

Edge

```
from typing import Literal


def should_retry(state: GraphState) -> Literal["retrieve_an
d_respond", "generate"]:
    print("----GRADTING---")
    print("Grade Score: ", state["grade"])

    # 답변 생성 횟수가 3회 이상이면 "generate"를 반환
    if state["num_generation"] > 2:
        return "generate"

    # 답변 품질 평가점수가 0.7 미만이면 RAG 체인을 다시 실행
    if state["grade"] < 0.7:
        return "retrieve_and_respond"
    else:
        return "generate"
```

Graph 구성

```
# 그래프 설정
builder = StateGraph(GraphState)
builder.add_node("retrieve_and_respond", retrieve_and_respo
nd)
builder.add_node("grade_answer", grade_answer)

builder.add_edge(START, "retrieve_and_respond")
builder.add_edge("retrieve_and_respond", "grade_answer")
builder.add_conditional_edges(
    "grade_answer",
    should_retry,
    {
```

```
        "retrieve_and_respond": "retrieve_and_respond",
        "generate": END
    }
)

# 그래프 컴파일
graph = builder.compile()

# 그래프 시각화
display(Image(graph.get_graph().draw_mermaid_png()))
```

Graph 실행

```
# 초기 상태
initial_state = {
    "messages": [HumanMessage(content="채식주의자를 위한 메뉴를 추천해주세요.")],
}

# 그래프 실행
final_state = graph.invoke(initial_state)

# 최종 상태 출력
print("최종 상태:", final_state)
```

```
# 최종 답변만 출력
pprint(final_state['messages'][-1].content)
```