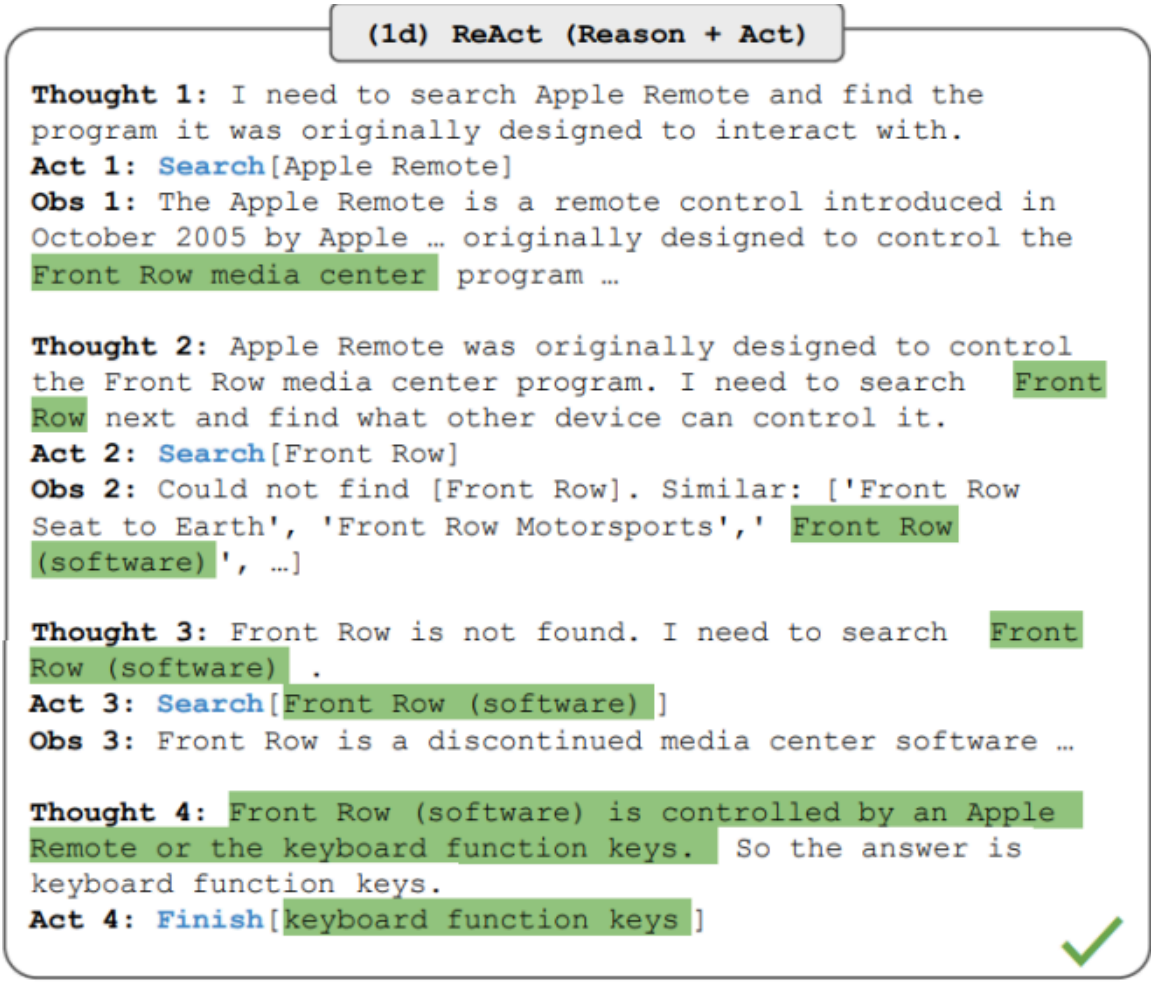
LangGraph ReAct 에이전트 활용

추론 - 행동 기반 ReAct 에이전트 이해하기

ReAct(Reasoning + Action) 개념

- 추론(Resoning)과 행동(Acting)을 결합한 접근 방식
- LLM이 단순히 텍스트를 생성하는 것을 넘어, 환경과 상호작용하며 복잡한 작업 수행
- 동작 방식
  1. 행동(Act)
     - 모델이 주어진 상황을 분석하고 적절한 도구를 선택
     - 선택한 도구를 호출하고 필요한 입력을 제공
  2. 관찰(Observe)
     - 호출된 도구의 실행 결과나 출력을 모델에 다시 전달
     - 에이전트가 자신의 행동 결과를 이해 / 학습
  3. 추론(Reason)
     - 이전 단계의 관찰 결과를 분석해 다음 행동 결정
     - 다른 도구 호출하거나, 또는 직접 응답 생성
     - 현재 상황을 평가하고 최선의 다음 단계 선택

  → 추론과 행동의 반복적인 사이클을 통해 복잡한 작업을 단계적으로 해결



- 사용자 정의 → @tool decorator

```
from langchain_chroma import Chroma
from langchain_ollama  import OllamaEmbeddings
from langchain_core.tools import tool
from typing import List

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

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

```
# Tool 정의
@tool
def search_menu(query: str) -> List[str]:
    """레스토랑 메뉴에서 정보를 검색합니다."""
    docs = vector_db.similarity_search(query, k=2)

    formatted_docs = "\n\n---\n\n".join(
        [
            f'<Document source="{doc.metadata["sourc
e"]}"/>\n{doc.page_content}\n</Document>'
            for doc in docs
        ]
    )

    if len(docs) > 0:
        return formatted_docs

    return "관련 메뉴 정보를 찾을 수 없습니다."
```

- LangChain 내장 도구
  - 웹 검색을 위한 Tavily 초기화

```
from langchain_community.tools import TavilySearchResult
s


# Tool 정의
@tool
def search_web(query: str) -> List[str]:
    """데이터베이스에 존재하지 않는 정보 또는 최신 정보를 인터넷
에서 검색합니다."""

    tavily_search = TavilySearchResults(max_results=3)
    docs = tavily_search.invoke(query)

    formatted_docs = "\n\n---\n\n".join(
        [
            f'<Document href="{doc["url"]}"/>\n{doc["con
```

```
tent"]}\n</Document>'
            for doc in docs
        ]
    )

    if len(docs) > 0:
        return formatted_docs

    return "관련 정보를 찾을 수 없습니다."
```

```
from langchain_openai import ChatOpenAI

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

# 도구 목록
tools = [search_menu, search_web]

# 모델에 도구를 바인딩
llm_with_tools = llm.bind_tools(tools=tools)
```

```
from langchain_core.messages import HumanMessage

# 도구 호출
tool_call = llm_with_tools.invoke([HumanMessage(content=
f"스테이크 메뉴의 가격은 얼마인가요?")])

# 결과 출력
print(tool_call.additional_kwargs)
```

```
# 도구 호출
tool_call = llm_with_tools.invoke([HumanMessage(content=
f"LangGraph는 무엇인가요?")])

# 결과 출력
print(tool_call.additional_kwargs)
```

도구 노드 (Tool Node)

- AI 모델이 필요한 도구(Tool) 호출을 실행하는 역할을 처리하는 LangGraph 컴포넌트
- 작동 방식
  - 가장 최근의 AI Message 에서 도구 호출 요청을 추출 (반드시, AIMessage는 반드시 tool_calls가 채워져 있어야 함)
  - 요청된 도구들을 병렬로 실행
  - 각 도구 호출에 대해 ToolMessage를 생성하여 반환
- 에이전트를 그래프로 구현할 때, 도구가 실행되는 노드를 별도로 함수로 구현할 수도 있지만 도구 호출을 직접적으로 실행하는 컴포넌트를 사용
- 병렬로 처리할 수 있도록 구현되어 있음

```
from langgraph.prebuilt import ToolNode

# 도구 노드 정의
tool_node = ToolNode(tools=tools)
```

```
# 도구 호출
tool_call = llm_with_tools.invoke([HumanMessage(content=
f"스테이크 메뉴의 가격은 얼마인가요?")])

tool_call
```

- 실행

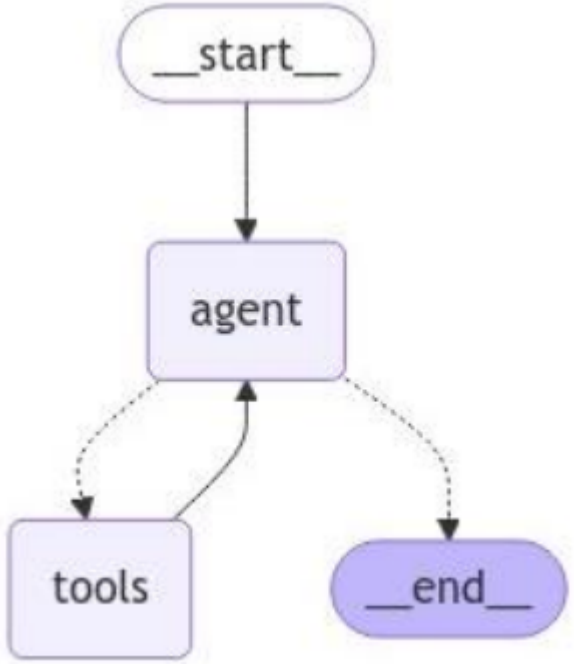
```
# 도구 호출 결과를 메시지로 추가하여 실행
results = tool_node.invoke({"messages": [tool_call]})

# 실행 결과 출력하여 확인
for result in results['messages']:
    print(result.content)
    print()
```

```
# LLM 모델을 이용하여 도구를 호출하여 실행
results = tool_node.invoke({"messages": [llm_with_tools.
```

```
invoke("LangGraph는 무엇인가요?")]})

# 실행 결과 출력하여 확인
for result in results['messages']:
    print(result.content)
    print()
```

랭그래프 (LangGraph)에 내장된 ReAct 에이전트 사용

- create_react_agent 함수
  - 가장 쉽게 ReAct 에이전트를 생성하는 방법
  - 주요 단계
    1. 필요한 라이브러리 임포트
    2. 언어 모델 (LLM) 설정
    3. 도구 정의
    4. ReAct 에이전트 생성
    5. 에이전트 실행



- 코드 예시

```
from langgraph.prebuilt import create_react_agent
# 그래프 생성
graph = create_react_agent(
    llm,
    tools = tools,
    state_modifier = system_prompt)

# 그래프 실행
inputs = {"messages": [HumanMessage(content = "스테이크 메뉴의 가격은 얼마인가요?")]}
messages = graph.invoke(inputs)
for m in messages['messages']:
    m.pretty_print()
```

- 랭그래프 내장 ReAct 에이전트 사용

```
from IPython.display import Image, display
from langgraph.prebuilt import create_react_agent
graph = create_react_agent(
    llm,
    tools=tools,
)

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

```
# 그래프 실행
inputs = {"messages": [HumanMessage(content="스테이크 메뉴의 가격은 얼마인가요?")]}
messages = graph.invoke(inputs)
for m in messages['messages']:
    m.pretty_print()
```

```
from langgraph.prebuilt import create_react_agent
from IPython.display import Image, display
```

```
# 시스템 프롬프트
system_prompt = dedent("""
You are an AI assistant designed to answer human questio
ns.
You can use the provided tools to help generate your res
ponses.

Follow these steps to answer questions:
    1. Carefully read and understand the question.
    2. Use the provided tools to obtain necessary inform
ation.
    3. Immediately after using a tool, cite the source u
sing the format below.
    4. Construct an accurate and helpful answer using th
e tool outputs and citations.
    5. Provide the final answer when you determine it's
complete.

When using tools, follow this format:
    Action: tool_name
    Action Input: input for the tool

Immediately after receiving tool output, cite the source
as follows:
    [Source: tool_name | document_title/item_name | url/
file_path]

For example:
    Action: search_menu
    Action Input: 스테이크

    (After receiving tool output)
    [Source: search_menu | 스테이크 | ./data/data.txt]
    스테이크에 대한 정보는 다음과 같습니다...

    Action: search_web
    Action Input: History of AI
```

```
    (After receiving tool output)
    [Source: search_web | AI History | https://en.wikipe
dia.org/wiki/History_of_artificial_intelligence]
    AI의 역사는 다음과 같이 요약됩니다...

If tool use is not necessary, answer directly.

Your final answer should be clear, concise, and directly
related to the user's question.
Ensure that every piece of factual information in your r
esponse is accompanied by a citation.

Remember: ALWAYS include these citations for all factual
information, tool outputs, and referenced documents in y
our response.
Do not provide any information without a corresponding c
itation.
""")

# 그래프 생성
graph = create_react_agent(
    llm,
    tools=tools,
    state_modifier=system_prompt,
    )

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

```
# 그래프 실행
inputs = {"messages": [HumanMessage(content="스테이크 메뉴
의 가격은 얼마인가요?")]}
messages = graph.invoke(inputs)
for m in messages['messages']:
    m.pretty_print()
```

- 조건부 엣지 함수를 사용자 정의

- should_continue 함수에서 도구 호출 여부에 따라 종료 여부 결정
- 도구 실행이 필요한 경우에는 그래프가 종료되지 않고 계속 실행

```
from langgraph.graph import MessagesState, StateGraph, S
TART, END
from langchain_core.messages import HumanMessage, System
Message
from langgraph.prebuilt import ToolNode
from IPython.display import Image, display

# LangGraph MessagesState 사용
class GraphState(MessagesState):
    pass

# 노드 구성
def call_model(state: GraphState):
    system_message = SystemMessage(content=system_promp
t)
    messages = [system_message] + state['messages']
    response = llm_with_tools.invoke(messages)
    return {"messages": [response]}

def should_continue(state: GraphState):
    last_message = state["messages"][-1]
    # 도구 호출이 있으면 도구 실행 노드로 이동
    if last_message.tool_calls:
        return "execute_tools"
    # 도구 호출이 없으면 답변 생성하고 종료
    return END

# 그래프 구성
builder = StateGraph(GraphState)
builder.add_node("call_model", call_model)
builder.add_node("execute_tools", ToolNode(tools))

builder.add_edge(START, "call_model")
builder.add_conditional_edges(
    "call_model",
```

```
    should_continue,
    {
        "execute_tools": "execute_tools",
        END: END
    }
)
builder.add_edge("execute_tools", "call_model")

graph = builder.compile()

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

```
# 그래프 실행
inputs = {"messages": [HumanMessage(content="스테이크 메뉴의 가격은 얼마인가요?")]}
messages = graph.invoke(inputs)
for m in messages['messages']:
    m.pretty_print()
```

- tools_condition 활용
  - LangGraph에서 제공하는 도구 사용을 위한 조건부 엣지 함수
  - 최신 메시지(결과)가 도구 호출이면 → tools_condition 이 도구로 라우팅
  - 최신 메시지(결과)가 도구 호출이 아니면 → tools_condition이 END로 라우팅

```
from langgraph.prebuilt import tools_condition

# 노드 함수 정의
def call_model(state: GraphState):
    system_message = SystemMessage(content=system_prompt)
    messages = [system_message] + state['messages']
    response = llm_with_tools.invoke(messages)
    return {"messages": [response]}
```

```
# 그래프 구성
builder = StateGraph(GraphState)

builder.add_node("agent", call_model)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "agent")

# tools_condition을 사용한 조건부 엣지 추가
builder.add_conditional_edges(
    "agent",
    tools_condition,
)

builder.add_edge("tools", "agent")

graph = builder.compile()

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

```
# 그래프 실행
inputs = {"messages": [HumanMessage(content="파스타에 어울리는 음료는 무엇인가요?")]}
messages = graph.invoke(inputs)
for m in messages['messages']:
    m.pretty_print()
```

StateGraph 구조를 사용해 ReAct 에이전트 만들기

- 조건부 엣지 함수를 사용자 정의
  - 조건부 엣지 함수를 사용하면 ReAct 에이전트의 동작을 더 세밀하게 제어 가능
  - 도구 호출 여부에 따라 실행 지속 여부 결정

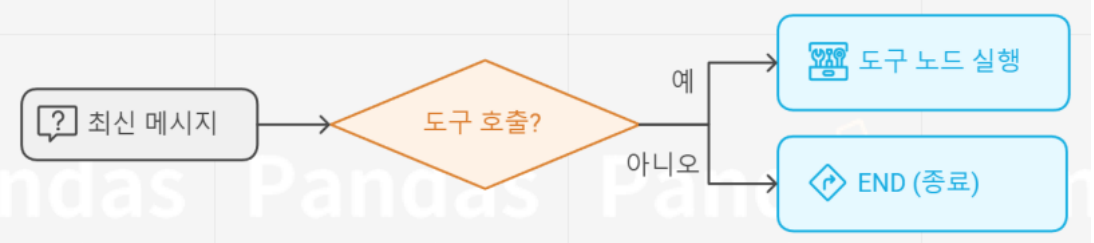
```
def should_continue(state: GraphState):
    last_message = state["messages"][-1]
    # 도구 호출이 있으면 도구 노드로 이동
    if last_message.tool_calls:
        return "excute_tools"
    # 도구 호출이 없으면 답변 생성하고 종료
    return END

# 그래프 구성
builder = StateGraph(GraphState)
builder.add_node("call_model", call_model)
builder.add_node("execute_tools", ToolNode(tools))

builder.add_edge(START, "call_model")
builder.add_conditional_edges(
    "call_model",
    should_continue,
)
builder.add_edge("execute_tools", "call_model")
graph = builder.compile()
```

- 도구 노드 ToolNode(tools)
  - 목적 : 모델이 요청한 도구 호출을 실제로 실행
  - 작동 방식
    1. 최신 AIMessage의 tool_calls 필드에서 도구 호출 정보 추출
    2. 추출된 도구 호출 요청들을 동시에(병렬로) 실행
    3. 각 도구 호출의 결과에 대해 ToolMessage를 생성 (결과 포함)
    4. ToolMessage는 다시 AI 모델에게 전달 → 답변 생성

```
from langgraph.prebuilt import ToolNode

# 도구 목록
tools = [search_menu, search_web]
```

```
# 도구 노드 정의
tool_node = ToolNode(tools)
```

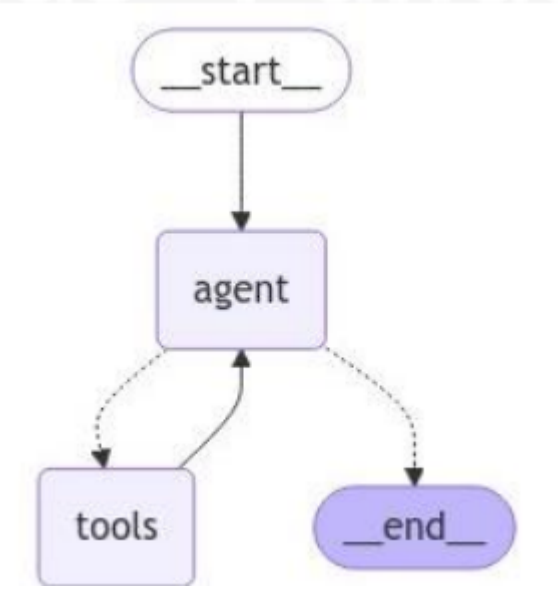
- 랭그래프 tools_condition 함수 활용
  - LangGraph에서 제공하는 매우 유용한 조건부 엣지 함수
  - 작동 방식
    1. 최신 메시지 (또는 결과)를 검사
    2. 도구 호출이 포함 : tools 노드에서 도구 실행
    3. 도구 호출이 없음 : END 노드에서 종료



  - 코드 예시

```
from langgraph.prebuilt import tools_condition

# 노드 함수 정의
def call_model(state: GraphState):
    system_prompt = SystemMessage(content = system_pr
ompt_template)
    messages = [system_prompt] + staet['messages']
    response = llm_with_tools.invoke(messages)
    return {"messages": [response]}

# 그래프 구성
builder = StateGraph(GraphState)

builder.add_node("agent", call_model)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "agent")
```

```
# tools_condition을 사용한 조건부 엣지 추가
builder.add_conditional_edges(
    "agent",
    tools_condition,
)
builder.add_edge("tools", "agent")
graph = builder.compile()
```



MemorySaver

- 그래프 각 단계 실행 후 자동으로 상태를 저장 (체크포인터 역할)
- 상태의 일시성(stateless) 문제 해결 (즉, 그래프는 각 실행마다 새로운 상태로 초기화 되는 문제)
- 대화의 연속성(멀티 턴), 대화 중단 후 복원 가능
- 독립적인 대화 스레드 관리
- Memory Saver 기능
    1. 체크포인터 : 그래프의 각 단계 실행 후 상태를 저장
    2. 인메모리 키-값 저장소 : 상태 검색 가능
    3. 지속성 제공 : 저장된 체크포인트로부터 실행 재개

```
from langgraph.checkpoint.memory import MemorySaver

# 메모리 초기화
memory = MemorySaver()

# 체크포인터 지정해 그래프 컴파일
graph_memory = builder.compile(checkpointer = memory)
```

- 체크 포인터 사용 방법
  1. 메모리 사용 시 thread_id를 지정
  2. 체크포인터는 그래프의 각 단계에서 상태를 기록(모든 상태 저장)
  3. 나중에 thread_id를 사용해 이 스레드에 접근 가능

```
config = {"configurable": {"thread_id": "1"}}
messages = [HumanMessage(content = "스테이크 메뉴의 가격은 얼마인가요?")]
messages = graph_memory.invoke({"messages": messages}, config)
for m in messages['messages']:
    m.pretty_print()

config = {"configurable": {"thread_id": "1"}}
messages = [HumanMessage(content = "둘 중에 더 저렴한 메뉴는 무엇인가요?")]
messages = graph_memory.invoke({"messages": messages}, config)
for m in messages['messages']:
    m.pretty_print()
```