
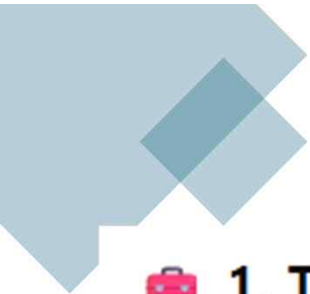


Tools 와 Agent



1. Tools






1. Tools 란 무엇인가

Tool(도구) 이란 LangChain Agent 가 작업을 수행할 때 실제로 호출할 수 있는 **외부 함수 또는 API 기능** 이다.

즉, LLM이 "생각(thought)" 단계에서 "이제 무엇을 해야 할까?" 판단했을 때 직접 실행할 수 있는 행동(action)을 정의한 것이다.

예를 들면 다음과 같다.

- "계산기를 사용해 수식을 계산한다."
 - "날씨 API를 호출해 현재 기온을 확인한다."
 - "데이터베이스에서 정보를 검색한다."
 - "파일 시스템에 기록하거나 읽는다."
- 

🌀 2. LangChain 1.0 에서 의 Tool 구조

LangChain v 1.0에서는 모든 Tool이 `Runnable` 객체로 추상화된다.
에이전트는 이 Runnable 도구를 LLM 내부의 함수 처럼 호출할 수 있다.

기본적인 정의 방식은 2가지이다.

① `@tool` 데코레이터 방식 (가장 간단하고 권장)

python

```
from langchain.tools import tool

@tool
def multiply(a: int, b: int) -> int:
    """두 수를 곱한 값을 반환한다."""
    return a * b
```


② 직접 Tool 객체로 정의

python

```
from langchain.tools import Tool

def get_weather(city: str) -> str:
    return f"{city}의 날씨는 맑음이다."


weather_tool = Tool.from_function(
    func=get_weather,
    name="get_weather",
    description="지정된 도시의 날씨를 가져온다."
)
```



3. Tool의 핵심 속성

속성	설명
<code>name</code>	도구의 고유 이름. LLM이 이 이름을 이용해 호출한다.
<code>description</code>	LLM에게 이 도구가 언제, 어떻게 쓰이는지 설명한다. (프롬프트에 포함됨)
<code>args_schema</code>	입력 데이터의 형식을 명시하는 Pydantic 모델(선택).
<code>return_direct</code>	도구 결과를 LLM을 거치지 않고 직접 반환할지 여부.
<code>coroutine</code>	비동기 실행용 함수(async 지원).






4. Agent 와 Tool 의 관계

에이전트는 다음 과정으로 Tool을 활용한다.

1. 사용자 질의 수신
2. LLM이 해당 질의에 필요한 도구 선택
3. 선택된 도구에 입력 값을 전달하고 결과 반환
4. 반환된 결과를 LLM 이 다시 해석하여 최종 답변 생성

예를 들면 "서울 날씨와 3x7 결과를 알려줘" 라는 입력이 들어오면,

LLM은 `get_weather` → `multiply` 를 순서대로 호출하고 최종 결과를 조합해 답을 낸다.



5. 전체 예제 (LangChain 1.0 표준 구조)

```
1 from langchain_core.tools import tool
2 from langchain_core.messages import SystemMessage
3 from datetime import datetime
4 import pytz # 시간대(timezone)를 다루기 위한 모듈
5
6 @tool # @tool 데코레이터를 사용하여 함수를 도구로 등록
7 def get_current_time(timezone: str, location: str) -> str:
8     """ 현재 시각을 반환하는 함수
9
10     Args:
11         timezone (str): 타임존 (예: 'Asia/Seoul') 실제 존재하는 타임존이어야 함
12         location (str): 지역명. 타임존이 모든 지명에 대응되지 않기 때문에 이후 llm 답변 생성에 사용됨
13     """
14     tz = pytz.timezone(timezone) # 특정 지역의 시간대를 객체로 생성
15     now = datetime.now(tz).strftime("%Y-%m-%d %H:%M:%S")
16     location_and_local_time = f'{timezone} ({location}) 현재시각 {now} ' # 타임존, 지역명, 현재시각을 문자열로 반환
17     print(location_and_local_time)
18     return location_and_local_time
19
20
21 # 도구를 tools 리스트에 추가하고, tool_dict에도 추가
22 tools = [get_current_time,]
23 tool_dict = {"get_current_time": get_current_time,}
```




```
25 # 도구를 모델에 바인딩: 모델에 도구를 바인딩하면, 도구를 사용하여 Llm 답변을 생성할 수 있음
26 llm_with_tools = model.bind_tools(tools)
27
28 # 사용자의 질문과 tools 사용하여 Llm 답변 생성
29 messages = [
30     SystemMessage("너는 사용자의 질문에 답변을 하기 위해 tools를 사용할 수 있다."),
31     HumanMessage("부산은 지금 몇시야?"),
32 ]
33
34 # llm_with_tools를 사용하여 사용자의 질문에 대한 Llm 답변 생성
35 response = llm_with_tools.invoke(messages)
36 messages.append(response)
37
38 # 생성된 Llm 답변 출력
39 print(messages)
```


```
1 for tool_call in response.tool_calls:
2     selected_tool = tool_dict[tool_call["name"]] # tool_dict를 사용하여 도구 함수를 선택
3     print(tool_call["args"]) # 도구 호출 시 전달된 인자 출력
4     tool_msg = selected_tool.invoke(tool_call) # 도구 함수를 호출하여 결과를 반환
5     messages.append(tool_msg)
6
7 messages
```

```
{'timezone': 'Asia/Seoul', 'location': '부산'}
Asia/Seoul (부산) 현재시각 2025-10-24 21:53:47
```



6. Tool 활용 팁

- 도구 이름과 설명은 자연어로 명확히 작성해야 LLM이 정확히 선택한다.
 - 도구가 많을 경우 **Tool kits** 로 그룹화할 수 있다 (예: Google Search toolkit, SQL toolkit).
 - 에이전트 내부 프롬프트에는 모든 도구의 이름과 설명이 자동으로 삽입된다.
 - 민감정보 처리 또는 안전한 API 호출이 필요하면 **Middleware** 계층에서 검증할 수 있다.
- 



7. 정리

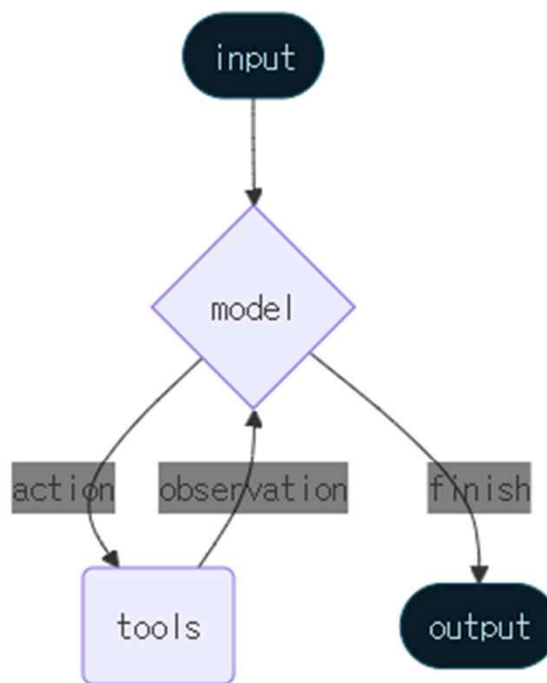
항목	내용
정의	에이전트가 LLM의 판단에 따라 실행할 수 있는 외부 기능 또는 함수
역할	LLM이 세상과 상호작용할 수 있도록 해주는 실행 수단
표준 정의법	<code>@tool</code> 데코레이터 또는 <code>Tool.from_function()</code>
연관 객체	<code>create_agent</code> , <code>Toolkits</code> , <code>AgentExecutor</code>
주요 변화	v 1.0 부터 LCEL/LangGraph 기반으로 추상화 통합



2. Agent

<https://docs.langchain.com/oss/python/langchain/agents>

An LLM Agent runs tools in a loop to achieve a goal. An agent runs until a stop condition is met - i.e., when the model emits a final output or an iteration limit is reached.



1. 에이전트(Agent)란

에이전트란, 고수준의 작업(task)을 받아서 언어모델(LLM)을 ****추론 엔진(reasoning engine)****으로 사용하고, 적절한 도구(tools)를 선택·실행하며 **반복적으로 행동(action) → 관찰(observation) → 추론(thought)** 과정을 거쳐 최종 결과를 도출하는 시스템이다. [docs.langchain.c... +2](#)

즉 단순히 질문에 답변하는 것이 아니라 “생각하고 → 도구를 실행하고 → 다시 생각하고 → 마무리 답변”하는 루프(loop)를 갖는다.


2. v 1.0에서의 주요 변화 및 특징

- v 1.0부터는 에이전트 생성 시 `create_agent` 함수가 표준 방식이다. [docs.langchain.c... +1](#)
- 내부적으로는 LangGraph 기반의 그래프(graph) 런타임 위에 에이전트가 구현되어 있다. 즉 노드(nodes)와 엣지(edges)를 이용해 “모델 호출 → 도구 호출 → 후처리” 등의 흐름을 표현한다. [docs.langchain.c... +1](#)
- 메시지 구조(message)나 출력 구조(output) 등이 좀 더 구조화되어 있다. 예컨대 `content_blocks` 같은 새로운 속성이 도입되었다. [LangChain Blog +1](#)
- 미들웨어(middleware)를 통해 에이전트 실행 흐름을 제어할 수 있는 기능이 강화되었다. 예컨대 요약, 개인정보 마스킹, 인간 검토(human-in-the-loop) 등이 가능하다. [docs.langchain.c...](#)

3. 구성 요소


에이전트 설계 시 고려해야 할 주요 구성 요소는 다음과 같다.

구성 요소	역할 및 설명
모델(Model; LLM)	에이전트의 추론 엔진이다. 주어진 입력(messages 등)을 받아서 어떤 행동(action)을 할지 결정한다. docs.langchain.c... +1
도구(Tools)	모델이 호출할 수 있는 기능 단위다. 예컨대 웹검색, 계산기, 데이터 베이스 질의, 이메일 발송 등이 도구가 된다. python.langchai... +1
프롬프트(Prompt) 또는 시스템 지시문	모델에게 에이전트로서 해야 할 역할이나 도구 사용 기준 등을 지시하는 부분이다.
루프 구조(Reason → Act → Observe)	모델이 생각(thought)을 서술하고, 행동(action)을 선택하고, 도구 실행 후 관찰(observation)을 모델에 다시 입력해 반복한다. docs.langchain.c...
중단 조건(Stopping condition)	반복 루프를 끝낼 조건이다. 예컨대, 모델이 최종 답변을 내놓으면 종료하거나 최대 반복 횟수를 정할 수 있다. python.langchai...
미들웨어(Middleware)	실행 흐름을 제어하거나 조작할 수 있는 중간 계층이다. 예: 대화 요약, 민감정보 검열, 인간 승인 등. docs.langchain.c...



4. 작동 흐름

에이전트의 일반적인 흐름은 다음과 같다.

1. 사용자가 입력을 보낸다 (예: "서울 날씨 알려줘", "3 곱하기 7 계산해줘").
 2. 모델이 해당 입력을 보고 어떤 도구를 사용할지 판단하며 자신의 생각(thought)을 생성한다.
 3. 모델이 도구(action)를 선택하여 실행한다 (예: 날씨 도구 호출, 계산 도구 호출).
 4. 도구가 실행되어 관찰(observation)을 반환한다.
 5. 모델이 그 관찰을 받아서 다시 생각하거나 추가 도구를 호출할지 결정한다.
 6. 위 과정을 반복하다가 중단 조건이 충족되면 최종 답변을 사용자에게 출력한다.
- 

5. 간단 코드 예제 (v 1.0 기준)

python

```
from langchain_openai import ChatOpenAI
from langchain.agents import create_agent
from langchain.tools import tool
```

```
@tool
```

```
def multiply(a: int, b: int) -> int:
    return a * b
```

```
@tool
```

```
def get_weather(city: str) -> str:
    return f"{city}의 날씨는 맑음이다."
```

```
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
```

```
agent = create_agent(  
    model=llm,  
    tools=[multiply, get_weather],  
    system_prompt="당신은 사용자의 요청을 해결하기 위해 필요한 도구를 적절히 이용하는 에이전트이다."  
)  
  
result = agent.invoke({  
    "messages": [  
        {"role": "user", "content": "서울의 날씨와 3 곱하기 7 결과를 알려줘"}  
    ]  
})  
  
print(result["messages"][-1].content)
```

서울의 날씨는 흐림입니다. 그리고 3×7 의 결과는 21입니다.

위 코드에서는 `create_agent` 를 사용했고, 두 가지 도구를 정의했으며 기본 시스템 프롬프트를 통해 역할을 지시했다.

6. 장점 및 활용 시 주의사항

장점

- 복잡한 작업이나 외부 API 호출이 필요한 작업을 자동화할 수 있다.
- 모델이 스스로 행동을 계획하고 실행하므로 유연성이 높다.
- 반복 추론 루프를 통해 보다 정확하고 신뢰성 있는 결과를 얻을 수 있다 (예: 도구 호출로 환각 (hallucination)을 줄일 수 있다). [docs.langchain.c...](#)

주의사항

- 잘못 설계된 도구 호출 흐름이나 프롬프트는 오작동이나 도구 남용을 초래할 수 있다.
- 중단 조건을 명확히 하지 않으면 무한 루프에 빠질 수 있다.
- 도구 설계 및 권한 관리, 민감정보 처리, 비용 관리 등이 필요하다—미들웨어 설계가 중요하다.
- v 1.0로 마이그레이션 할 경우 기존 `AgentExecutor`, `create_react_agent` 등을 사용하던 코드가 변경 요구될 수 있다. [docs.langchain.c...](#)

🧠 내부 동작 이해

`create_agent()` 함수는 내부적으로 아래 과정을 자동화한다.

SCSS


User **Input** → LLM Node → Tool **Node(s)** → LLM Node → Output Node

즉, 당신이 예시로 만든 다음 구조를 LangChain이 자동 생성해준다.

python

```
graph = StateGraph(StateSchema)
graph.add_node("llm", ChatOpenAI(...))
graph.add_node("tool", Tool(...))
graph.add_edge("llm", "tool")
graph.add_edge("tool", "llm")
graph.set_entry_point("llm")
graph.set_finish_point("llm")
```


이 과정을 수동으로 작성할 필요가 없다는 뜻이다.



두 접근법 비교

항목	<code>create_agent()</code>	StateGraph 직접 작성
목적	표준 Agent 구성 (LLM + Tool 자동 연결)	복잡한 워크플로우 설계, 커스텀 파이프라인
사용 난이도	쉬움	다소 복잡
노드 정의	내부 자동 생성	직접 정의 (함수, Runnable 등)
상태 관리	기본 메시지 히스토리 중심	TypedDict 등으로 명시적 상태 관리
예시	LLM + Tools 기반 에이전트	RAG, 다단계 QA, 멀티 Agent 협업
코드 길이	짧음	길어짐 (명시적 노드 구성 필요)





사용 목적

일반적인 LLM + Tools 기반의 “생각 → 도구 호출 → 답변” 에이전트

맞춤형 파이프라인, 다단계 노드 처리, 조건 분기, 복잡한 상태 공유


여러 Agent 간 협업(멀티에이전트), 병렬처리, 워크플로 제어


권장 방식

✓ `create_agent()` 사용 (LangGraph 내부 자동 구성)

✓ `StateGraph` 를 직접 작성

✓ LangGraph 직접 설계 필수





🧩 실제 사용 시 가이드

- 단일 Agent 프로젝트:


👉 `create_agent(model, tools)` 로 충분하다.

(LangGraph 내부적으로 자동 노드 구성)

- 복잡한 데이터 흐름 제어 / RAG / 다중 Agent 협업:

👉 `StateGraph` 와 `RunnableLambda`, `RunnableSequence` 등을 직접 사용해야 한다.

이 경우 LangGraph의 노드를 직접 구성하는 것이 필수이다.



정리

질문

답변

“create_agent를 쓸 때 노드를 직접 만들어야 하나요?”

❌ 필요 없음. 내부적으로 LangGraph 구조 자동 생성

“내가 워처럼 StateGraph를 직접 짜는 경우는?”

✅ 맞춤형 그래프(복잡한 단계, 병렬 처리, 멀티 Agent 협업 등)를 설계할 때

“둘 중 어떤 게 낫나요?”

일반적인 AI Agent는 `create_agent()` 로 충분하고, RAG/Workflow 엔진급 프로젝트는 LangGraph 직접 구성이 적합

요약하자면

LangGraph는 LangChain Agent의 엔진이다.

`create_agent()` 는 이 엔진을 자동으로 구성하고,

`StateGraph` 는 엔진을 직접 설계할 수 있는 저수준 API이다.



감사합니다