Memory (Bộ nhớ)

Một trong những yếu tố quan trọng nhất để tạo nên sự khác biệt giữa một agent “bình thường” và một agent “đỉnh cao” sẵn sàng chạy trên môi trường thực tế (production).

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Giới Thiệu

Tiếp tục hành trình – Bước vào một chủ đề cực kỳ quan trọng: Memory (Bộ nhớ) trong LangGraph

Trong phần này, chúng ta sẽ đào sâu vào một chủ đề đóng vai trò sống còn khi xây dựng agent bằng LangGraph – đó chính là **memory (bộ nhớ)**. Và không chỉ nói đến **trí nhớ ngắn hạn(short term memory)**, mà cả **trí nhớ dài hạn(long term memory)** nữa!

Ngay từ đầu, bạn sẽ nhận ra: đây chính là **một trong những yếu tố quan trọng nhất để tạo nên sự khác biệt** giữa một agent "bình thường" và một agent "đỉnh cao" sẵn sàng chạy trên môi trường thực tế (production).

Nói đơn giản:
Ứng dụng của bạn có thật sự thông minh và đáng nhớ hay chỉ dừng lại ở mức "cho vui" – tất cả nằm ở cách bạn xử lý memory.

Vậy chúng ta sẽ học gì?

Chúng ta sẽ bắt đầu với phần dễ trước – **trí nhớ ngắn hạn(short term memory)**. Đây là loại bộ nhớ gắn liền với **cuộc trò chuyện hiện tại** giữa người dùng và agent.

Nếu bạn muốn agent của mình **hiểu được mạch hội thoại** (biết bạn vừa nói gì, nhớ câu hỏi trước, không bị "mất trí nhớ ngắn hạn"), thì bạn cần phải **kích hoạt trí nhớ dài hạn(long term memory)** cho nó.

Và bạn sẽ thấy: Việc này **cực kỳ đơn giản** với LangGraph.

Có những thứ bạn cần nắm trước

Trong tài liệu chính thức của LangGraph, khi họ bắt đầu nói về memory, họ thường đưa ra rất nhiều khái niệm như:

- cách định dạng schema của state,
- dùng reducer tùy chỉnh (custom reducers),
- hay sử dụng nhiều schema trạng thái trong một ứng dụng.

Thành thật mà nói, **cách LangGraph viết tài liệu không phải lúc nào cũng dễ hiểu**, thậm chí có thể gây rối nếu bạn mới bắt đầu.

Vì vậy, chúng ta sẽ làm rõ ba khái niệm đó trước – theo cách **đơn giản, trực quan và đúng bản chất**, giúp bạn hiểu sâu mà **không bị rối bởi quá nhiều dòng code**.

Mục tiêu của phần này là gì?

- Giúp bạn hiểu rõ **trí nhớ ngắn hạn(short term memory)** hoạt động như thế nào trong LangGraph.
- Làm quen với các khái niệm nâng cao: `state schema`, `custom reducer`, và `multiple schemas`.
- Tập trung vào phần **tư duy và logic**, không sa đà vào chi tiết lập trình.

Hãy sẵn sàng! Trong những bài tiếp theo, chúng ta sẽ cùng nhau khám phá kỹ hơn về 4 khái niệm trên, với những ví dụ thực tế và trực quan nhất.

Bạn chỉ cần nhớ: **đừng để tài liệu khó hiểu làm bạn nản lòng – chúng ta sẽ đi từng bước, và bạn chắc chắn sẽ làm được.**

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Thêm Bộ Nhớ Ngắn Hạn vào Agent trong LangGraph

Trong bài học này, chúng ta sẽ tìm hiểu cách thêm **bộ nhớ ngắn hạn (short-term memory)** vào agent để nó có thể ghi nhớ ngữ cảnh của cuộc trò chuyện. Điều này giúp agent hiểu được các yêu cầu liên tiếp từ người dùng một cách chính xác hơn.

Vấn đề của Agent Không Có Bộ Nhớ

Trong bài tập trước, chúng ta đã xây dựng một agent đơn giản có thể thực hiện các phép tính toán học bằng cách di chuyển qua các node **"assistant"** và **"tools"**. Tuy nhiên, agent này có một hạn chế lớn: **nó không có bộ nhớ**.

Ví dụ minh họa:

1. Người dùng yêu cầu:
   "*3 + 5.*"
   → Agent trả lời: "*Tổng của 3 + 5 = 8*"
2. Người dùng tiếp tục:
   "*nhân kết quả với 2*"

Nếu agent **không có bộ nhớ**, nó sẽ không hiểu "*kết quả*" ám chỉ kết quả trước đó và yêu cầu người dùng cung cấp lại số:
"*Tôi cần một số cụ thể để nhân với hai. Bạn vui lòng cung cấp số bạn muốn nhân*"

Điều này gây bất tiện vì người dùng phải lặp lại thông tin đã cung cấp trước đó.

Cách Thêm Bộ Nhớ Ngắn Hạn vào Agent

Trong **LangGraph**, việc thêm bộ nhớ ngắn hạn rất đơn giản bằng cách sử dụng module `MemorySaver`. Module này giúp lưu trữ thông tin tạm thời trong một phiên hội thoại cụ thể.

1. Sử dụng `MemorySaver` tại bước `compile`

Khi khởi tạo agent, chúng ta thêm bộ nhớ vào trong quá trình **compile**:

```
from langgraph.checkpoint.memory import MemorySaver


memory = MemorySaver()
```

```
react_graph_memory = builder.compile(checkpointer=memory)
```

2. Sử dụng `thread_id` để phân biệt các cuộc hội thoại

Để đảm bảo bộ nhớ chỉ áp dụng cho **một phiên trò chuyện cụ thể**, ta sử dụng **`thread_id`** khi gọi `invoke`:

```
config = {"configurable": {"thread_id": "1"}}

messages = [HumanMessage(content="Add 3 and 4.")]

messages = react_graph_memory.invoke({"messages": messages},config)
```

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Thực hành bộ nhớ ngắn hạn

Trong bài học này, chúng ta sẽ học cách thêm bộ nhớ ngắn hạn vào agent của mình.

1. Cài đặt

Tại console:

```
* cd project_name
* pyenv local 3.11.4
* poetry install
* poetry shell
* jupyter lab
```

2. Tạo file .env

```
* OPENAI_API_KEY=your_openai_api_key
* LANGCHAIN_TRACING_V2=true
* LANGCHAIN_ENDPOINT=https://api.smith.langchain.com
* LANGCHAIN_API_KEY=your_langchain_api_key
* LANGCHAIN_PROJECT=your_project_name
```

3. Kết nối với tệp .env nằm trong cùng thư mục của notebook

```
#pip install python-dotenv
```

```
import os
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())
openai_api_key = os.environ["OPENAI_API_KEY"]
```

3. Cài đặt LangChain

```
#!pip install langchain
#!pip install langchain-openai
```

```
from langchain_openai import ChatOpenAI
```

```
chatModel35 = ChatOpenAI(model="gpt-3.5-turbo-0125")
chatModel4o = ChatOpenAI(model="gpt-4o")
```

4. LLM kết hợp với nhiều công cụ hỗ trợ

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

@tool
def multiply(a: int, b: int) -> int:
    """Multiply a and b.

    Args:
        a: first int
        b: second int
    """
    return a * b

@tool
def add(a: int, b: int) -> int:
    """Adds a and b.

    Args:
        a: first int
        b: second int
    """
    return a + b

@tool
def divide(a: int, b: int) -> float:
    """Divide a and b.

    Args:
        a: first int
        b: second int
    """
    return a / b

tools = [add, multiply, divide]
llm_with_tools = chatModel4o.bind_tools(tools, parallel_tool_calls=False)
```

`parallel_tool_calls=True`

Đây là **gọi công cụ song song** (chạy nhiều công cụ cùng lúc).

- **Ý tưởng:** Nếu mô hình cần gọi nhiều công cụ để trả lời người dùng, nó có thể gọi tất cả các công cụ cùng một lúc, không cần chờ từng cái chạy xong.
- **Ưu điểm:** Nhanh hơn! Vì các công cụ chạy đồng thời.
- **Khi nào nên dùng:** Khi các công cụ **không phụ thuộc** vào kết quả của nhau. Ví dụ: bạn hỏi đồng thời "thời tiết ở TP. Hồ Chí Minh?" và "giá Bitcoin hôm nay?" — hai việc này độc lập nhau.

`parallel_tool_calls=False`

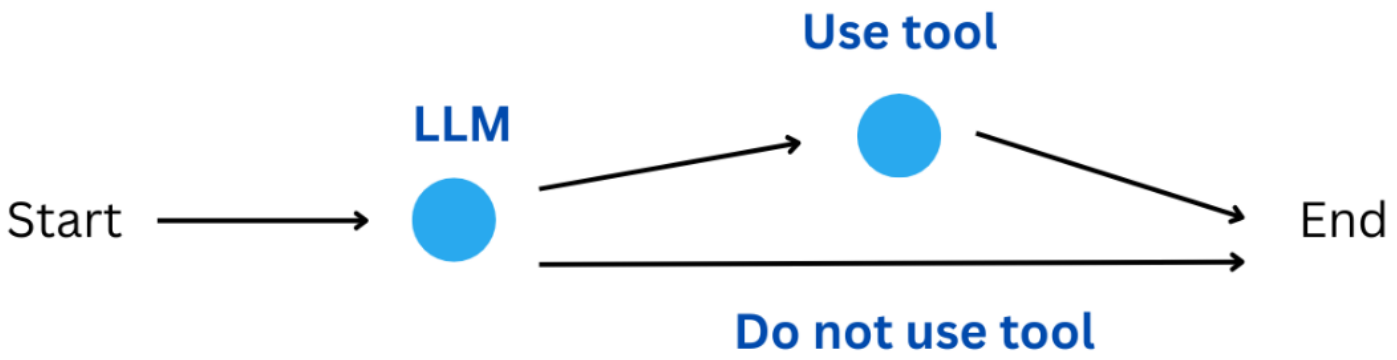
Đây là **gọi công cụ tuần tự** (chạy lần lượt từng cái một).

- **Ý tưởng:** Mỗi công cụ được gọi **theo thứ tự**, cái sau có thể dùng kết quả của cái trước.
- **Ưu điểm:** Giữ đúng logic nếu cần kết quả từng bước.
- **Khi nào nên dùng:** Khi các công cụ **phụ thuộc lẫn nhau**. Ví dụ:
  - Bước 1: Tính tổng
  - Bước 2: Dùng kết quả tổng để chia 2
    → Phải làm từng bước theo thứ tự.

Chi tiết tại

https://python.langchain.com/docs/how_to/tool_calling_parallel/

5. Xây dựng một ứng dụng (gọi là đồ thị trong langgraph) quyết định xem có nên trò chuyện bằng LLM hay sử dụng công cụ



5.1 Định nghĩa State schema

```
from langgraph.graph import MessagesState
```

```
class MessagesState(MessagesState):
    # Add any keys needed beyond messages, which is pre-built
    pass
```

5.2 Định nghĩa Node đầu tiên

```
from langgraph.graph import MessagesState
from langchain_core.messages import HumanMessage, SystemMessage

# Khai báo System message
sys_msg = SystemMessage(content="Bạn là trợ lý hữu ích có nhiệm vụ thực hiện phép tính số học trên một tập hợp dữ liệu đầu vào.")

# Định nghĩa Node
def assistant(state: MessagesState):
   return {"messages": [llm_with_tools.invoke([sys_msg] + state["messages"])]}
```

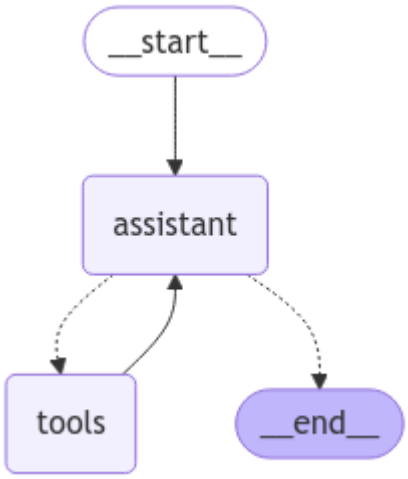
[sys_msg] + state["messages"]

- sys_msg là 1 **SystemMessage** định hướng vai trò AI, ví dụ: SystemMessage(content="Bạn là trợ lý toán học.")
- `state["messages"]`: là các tin nhắn trước đó (giữa người dùng và AI).

5.3. Kết hợp các Node và Edge để xây dựng Graph

```
from langgraph.graph import START, StateGraph
from langgraph.prebuilt import tools_condition
from langgraph.prebuilt import ToolNode
from IPython.display import Image, display

# Build graph
builder = StateGraph(MessagesState)

builder.add_node("assistant", assistant)
builder.add_node("tools", ToolNode(tools))

# Add the logic of the graph
builder.add_edge(START, "assistant")

builder.add_conditional_edges(
    "assistant",
```

```
    # Nếu câu trả lời gần nhất của assistant là một lời gọi tới tool, Thì điều kiện tools_condition sẽ quyết định đi tiếp đến node tools
    # Ngược lại, tools_condition sẽ quyết định chuyển sang node END (kết thúc luồng, trả kết quả ra ngoài).
    tools_condition,
)

# PAY ATTENTION HERE: from the tools node, back to the assistant
builder.add_edge("tools", "assistant")

# PAY ATTENTION: No END edgge.

# Compile the graph
react_graph = builder.compile()

# Visualize the graph
display(Image(react_graph.get_graph(xray=True).draw_mermaid_png()))
```



5.4 Chạy ứng dụng

```
messages = [HumanMessage(content="Add 3 and 4.")]

messages = react_graph.invoke({"messages": messages})

for m in messages['messages']:
    m.pretty_print()
```

================================ Human Message
================================ Add 3 and 4.

================================== Ai Message
================================== Tool Calls: add
(call_AKxpCcslEcimrNFvBEWZ2Ru3) Call ID: call_AKxpCcslEcimrNFvBEWZ2Ru3 Args: a: 3 b: 4
================================= Tool Message
================================= Name: add 7
================================== Ai Message
================================== The sum of 3 and 4 is 7.

5.5 Tiếp tục hỏi Agent

```
messages = [HumanMessage(content="Multiply that by 2.")]


messages = react_graph.invoke({"messages": messages})


for m in messages['messages']:
    m.pretty_print()
```

================================ Human Message
================================ Multiply that by 2.
================================== Ai Message
================================== I need a specific number to multiply by 2. Could you please provide the number you want to multiply?

> *Như bạn thấy, Agent yêu cầu cung cấp một số cụ thể để thực hiện phép tính nhân, nó không nhớ được kết quả của phép tính cộng đã thực hiện ở trên*

6. Thêm bộ nhớ

```
from langgraph.checkpoint.memory import MemorySaver


memory = MemorySaver()


react_graph_memory = builder.compile(checkpointer=memory)
```

```
# PAY ATTENTION HERE: see how we specify a thread
config = {"configurable": {"thread_id": "1"}}


# Enter the input
messages = [HumanMessage(content="Add 3 and 4.")]
```

```
# PAY ATTENTION HERE: see how we add config to referr to the thread_id
messages = react_graph_memory.invoke({"messages": messages},config)

for m in messages['messages']:
    m.pretty_print()
```

Kết quả

================================ Human Message
================================ Add 3 and 4.
================================== Ai Message
================================== Tool Calls: add
(call_7lH6zvBw4rMeTG3Zu66i53eW) Call ID: call_7lH6zvBw4rMeTG3Zu66i53eW Args: a: 3 b: 4
================================= Tool Message
================================= Name: add 7
================================== Ai Message
================================== The sum of 3 and 4 is 7.

Tiếp tục yêu cầu phép tính nhân

```
# PAY ATTENTION HERE: see how we check if the app has memory
messages = [HumanMessage(content="Multiply that by 2.")]

# Again, see how we use config here to referr to the thread_id
messages = react_graph_memory.invoke({"messages": messages}, config)

for m in messages['messages']:
    m.pretty_print()
```

================================ Human Message
================================ Multiply that by 2.
================================== Ai Message
================================== Tool Calls: multiply
(call_x3tKpGBsTHUCbVwzBr3PiNn1) Call ID: call_x3tKpGBsTHUCbVwzBr3PiNn1 Args:

...
14 ================================== Ai Message
================================== The result of multiplying 7 by 2 is 14.

Agent đã nhớ được kết quả của phép tính cộng và thực hiện phép tính nhân

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Định dạng state schema trong LangGraph từ TypeDict sang Pydantic

Trong ba bài học tiếp theo, chúng tôi sẽ giới thiệu cho bạn về một số khái niệm thú vị mà bạn sẽ gặp trong tài liệu của LangGraph, đặc biệt là trong phần *memo*.

Tuy nhiên, ở thời điểm này, những khái niệm này có thể gây xao nhãng cho bạn. Vì vậy, chúng tôi **không khuyến khích bạn đào sâu vào các chủ đề này ngay lúc này**. Mục tiêu là **giúp bạn hiểu khái niệm** và **biết nơi có thể tìm hiểu thêm khi cần thiết**.

Chủ đề đầu tiên là các cách định dạng state schema trong LangGraph.

Nếu bạn còn nhớ, khi tạo **state schema** trong các bài tập trước, chúng ta **luôn sử dụng loại** `TypeDict`.

`TypeDict` là một cách rất đơn giản để xây dựng state schema.
Tuy nhiên, **khi bạn làm việc với các ứng dụng ở cấp độ production**, cách này có thể **quá đơn giản** và **thiếu tính an toàn**, đặc biệt trong việc **phát hiện lỗi** hoặc **cảnh báo khi có điều gì đó sai** trong ứng dụng của bạn.

Trong bài học này, chúng tôi sẽ giải thích nội dung mà tài liệu của LangGraph đề cập về các cách định dạng state schema:

Bao gồm:

- Cách sử dụng `TypeDict` (cơ bản nhất)
- Sử dụng `Literal` để làm cho `TypeDict` mạnh mẽ hơn
- Sử dụng `dataclass` (tiến bộ hơn)
- Cuối cùng và quan trọng nhất: **sử dụng** `Pydantic`

I. TypeDict trong State Schema

`TypeDict` là một cách đơn giản và nhanh chóng để định nghĩa **state schema** – tức là cấu trúc trạng thái (state) mà LangGraph sẽ lưu trữ và truyền giữa các node (hoặc giữa LLM và công cụ).

Nó là một alias của `TypedDict` trong Python, được dùng để định nghĩa dictionary có kiểu rõ ràng cho từng key.

Khi nào nên dùng `TypeDict`?

- Khi bạn muốn tạo **state đơn giản**
- Phù hợp với các **ứng dụng nhỏ hoặc giai đoạn thử nghiệm**
- Không yêu cầu xác thực đầu vào quá nghiêm ngặt

Cách định nghĩa State Schema với `TypeDict`

```
from typing import TypedDict


class MyState(TypedDict):
    name: str
    mood: str
    count: int
```

Trong ví dụ trên, `MyState` định nghĩa một state có 3 trường:

- `name`: kiểu `str`
- `mood`: kiểu `str`
- `count`: kiểu `int`

Ví dụ trong LangGraph

```
from langgraph.graph import StateGraph


# Định nghĩa schema bằng TypeDict
class State(TypedDict):
    name: str
    mood: str


# Tạo Graph với state schema
builder = StateGraph(State)


# Add nodes, edges... (ví dụ đơn giản)
def greet(state: State) -> State:
    print(f"Hello, {state['name']}! You seem {state['mood']} today.")
    return state
```

```
builder.add_node("greet", greet)
builder.set_entry_point("greet")

app = builder.compile()

# Chạy thử
initial_state = {"name": "Alice", "mood": "happy"}
app.invoke(initial_state)
```

Lưu ý khi dùng TypeDict

- **Không kiểm tra giá trị hợp lệ** – ví dụ, nếu bạn muốn `mood` chỉ nhận `"happy"` hoặc `"sad"`, thì TypeDict **không báo lỗi** nếu bạn truyền `"angry"`.
- **Không hỗ trợ xác thực dữ liệu phức tạp**
- Nếu state sai kiểu (ví dụ `count="abc"`), nó vẫn có thể chạy mà không báo lỗi rõ ràng

Ưu điểm	Hạn chế
Dễ viết, nhanh chóng	Không xác thực dữ liệu
Phù hợp cho prototyping(giai đoạn thử nghiệm hoặc xây dựng bản mẫu)	Dễ gây lỗi trong ứng dụng lớn
Sử dụng chuẩn Python typing	Không cảnh báo khi sai dữ liệu

II. Literal

Trong Python, `Literal` được cung cấp bởi thư viện `typing` và cho phép bạn **ràng buộc giá trị của một biến chỉ được phép nằm trong một tập hợp cụ thể**. Đây là một cách tuyệt vời để **tăng tính an toàn** cho state schema, đảm bảo rằng người dùng hoặc ứng dụng chỉ có thể nhập những giá trị hợp lệ.

`Literal` trong State Schema

Khi định nghĩa **trạng thái (state)** trong LangGraph, bạn có thể muốn giới hạn một trường nào đó chỉ nhận một số giá trị nhất định.
Ví dụ: trạng thái "mood" chỉ được là `"happy"` hoặc `"sad"` — không được là `"angry"`, `"confused"`, v.v.

Ví dụ 1

```
from typing import TypedDict, Literal

class MyState(TypedDict):
    mood: Literal["happy", "sad"]
```

Trong ví dụ trên:

- Trường `mood` **chỉ được nhận một trong hai giá trị**: `"happy"` hoặc `"sad"`.
- Nếu bạn cố gắng gán giá trị khác như `"excited"` hoặc `"angry"`, thì trình phân tích tĩnh (hoặc các công cụ như `pydantic`) sẽ cảnh báo lỗi.

Ví dụ 2

```
from typing import TypedDict, Literal


class UserState(TypedDict):
    name: str
    mood: Literal["happy", "sad"]
```

Giá trị hợp lệ:

```
state: UserState = {
    "name": "Alice",
    "mood": "happy"
}
```

Giá trị không hợp lệ (trình kiểm tra type sẽ cảnh báo):

```
state: UserState = {
    "name": "Alice",
    "mood": "angry"  # □ Không nằm trong Literal["happy", "sad"]
}
```

Trong LangGraph, bạn có thể sử dụng `Literal` để đảm bảo rằng luồng logic của bạn hoạt động đúng như mong đợi.
Ví dụ:

```
class WorkflowState(TypedDict):
    step: Literal["start", "processing", "done"]
```

Với định nghĩa này, bạn có thể đảm bảo:

- Chỉ các bước `"start"`, `"processing"`, hoặc `"done"` mới được sử dụng.
- Tránh lỗi do nhập sai chuỗi hoặc thiếu kiểm tra logic.

III. `dataclass` để tạo state schema

Nhắc lại dataclass trong python

`dataclasses` giúp bạn **tự động tạo ra các phương thức** như:

- `__init__()` – hàm khởi tạo
- `__repr__()` – biểu diễn đối tượng
- `__eq__()` – so sánh bằng
- `__hash__()` – dùng cho tập hợp, từ điển
- `__lt__()`, `__le__()`... nếu bạn bật `order=True`

Ví dụ

```
from dataclasses import dataclass


@dataclass
class Person:
    name: str
    age: int
```

Python sẽ tự động sinh ra:

```
def __init__(self, name: str, age: int):
    self.name = name
    self.age = age
```

`dataclass` để tạo state schema(Chi tiết về dataclass tại đây)

```
from dataclasses import dataclass


@dataclass
class ConversationState:
    name: str
    mood: str
    count: int = 0  # Giá trị mặc định
```

`__post_init__`: logic sau khi khởi tạo

```
@dataclass
class State:
    name: str
    mood: str
```

```
    def __post_init__(self):
        if self.mood not in ["happy", "sad"]:
            raise ValueError("Mood must be 'happy' or 'sad'")
```

Hợp lệ

```
state = State(name="Lan", mood="happy")
print(state)
# Output: ConversationState(name='Lan', mood='happy')
```

Không hợp lệ

```
state = State(name="Lan", mood="angry")
# Output: ValueError: Mood must be 'happy' or 'sad'
```

IV. Dùng `Pydantic` làm State Schema trong LangGraph

Chi tiết về Pydantic

Trong LangGraph, **State Schema** định nghĩa cấu trúc dữ liệu sẽ được truyền và cập nhật giữa các node trong graph.
Pydantic là một thư viện mạnh mẽ giúp **xác thực dữ liệu, kiểm tra lỗi đầu vào và định kiểu rõ ràng**, rất phù hợp cho các ứng dụng thực tế (production-level).

Ưu điểm khi dùng Pydantic cho State Schema

- Tự động kiểm tra và xác thực dữ liệu đầu vào
- Xác định kiểu dữ liệu rõ ràng, dễ đọc
- Hiển thị lỗi rõ ràng và chi tiết khi dữ liệu không đúng định dạng
- Dễ mở rộng, dễ bảo trì khi project phức tạp hơn

Cách định nghĩa State Schema bằng Pydantic

Bạn sẽ tạo một lớp kế thừa từ `pydantic.BaseModel`.
Mỗi thuộc tính của lớp đại diện cho một phần trong trạng thái.

1. Cài đặt pydantic

```
pip install pydantic
```

2. Tạo State

```
from pydantic import BaseModel
from typing import Literal

class MyState(BaseModel):
    name: str
    age: int
    mood: Literal["happy", "sad"]
```

- `name`: bắt buộc phải là chuỗi (`str`)
- `age`: là số nguyên
- `mood`: chỉ chấp nhận `"happy"` hoặc `"sad"` → nếu giá trị khác sẽ báo lỗi

Ví dụ sử dụng trong LangGraph:

```
from langgraph.graph import StateGraph, END
from langgraph.checkpoint import MemorySaver

# Định nghĩa schema với Pydantic
class State(BaseModel):
    name: str
    mood: Literal["happy", "sad"]

# Node đơn giản: in trạng thái hiện tại
def print_state(state: State):
    print(f"□□ {state.name} đang cảm thấy {state.mood}")
    return state

# Tạo graph
builder = StateGraph(State)
builder.add_node("printer", print_state)
builder.set_entry_point("printer")
builder.set_finish_point("printer")

# Chạy
graph = builder.compile()
graph.invoke({"name": "An", "mood": "happy"})  # □ OK
graph.invoke({"name": "An", "mood": "angry"})  # □ Gây lỗi ValidationError
```

Khi có lỗi, bạn sẽ thấy:

```
ValidationError: 1 validation error for State
mood
  unexpected value; permitted: 'happy', 'sad' (type=value_error.const; given=angry; permitted=('happy', 'sad'))
```

Mở rộng với mặc định và tùy chỉnh:

```
from pydantic import Field


class State(BaseModel):
    name: str = Field(..., description="Tên người dùng")
    mood: Literal["happy", "sad"] = "happy"  # mặc định là happy
```

- Đây là cách bạn tạo một **State Schema** trong LangGraph bằng `Pydantic`.
- `BaseModel` là lớp cơ sở của Pydantic dùng để định nghĩa và kiểm tra kiểu dữ liệu.

`name: str = Field(..., description="Tên người dùng")`

- `name` là một **thuộc tính bắt buộc** phải là chuỗi (`str`).
- `Field(...)` với dấu ba chấm `...` có nghĩa là **giá trị này bắt buộc phải được truyền vào**.
- `description="Tên người dùng"` chỉ là mô tả – không bắt buộc, nhưng hữu ích khi dùng để sinh docs hoặc debug.

Nếu bạn không truyền `name`, bạn sẽ nhận được lỗi:

```
ValidationError: 1 validation error for State
name
  field required (type=value_error.missing)
```

`mood: Literal["happy", "sad"] = "happy"`

- `mood` là một **thuộc tính tùy chọn** với giá trị mặc định là `"happy"`.
- `Literal["happy", "sad"]` có nghĩa là: chỉ được phép là `"happy"` hoặc `"sad"`. Bất kỳ giá trị nào khác đều bị từ chối.
- `= "happy"` nghĩa là nếu người dùng không truyền giá trị `mood`, thì mặc định sẽ là `"happy"`.

```
State(name="An")  # OK, mood mặc định là "happy"
State(name="An", mood="sad")  # OK
State(name="An", mood="angry")  #  Lỗi vì "angry" không nằm trong ["happy", "sad"]
```

Tổng kết:

Thành phần	Giải thích

`Field(...)`	Đánh dấu giá trị là **bắt buộc**
`description`	Ghi chú mô tả, dùng cho docs
`Literal[...]`	Ràng buộc giá trị chỉ được nằm trong một tập hợp cố định
`= "giá trị"`	Thiết lập giá trị **mặc định** nếu không truyền vào

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Cách Tùy Chỉnh Cập Nhật Trạng Thái bằng Reducers

Trong LangGraph, **Reducers** là những hàm đặc biệt được sử dụng để **tổng hợp dữ liệu** hoặc **kết hợp trạng thái** khi có nhiều luồng song song (parallel branches) trả về kết quả khác nhau.

Nói cách khác:

> Khi bạn chạy nhiều node cùng lúc (parallel), và muốn gom kết quả từ chúng lại để đưa vào bước tiếp theo – thì bạn cần một **Reducer**.

Bạn cần dùng Reducer khi:

- Bạn có **các nhánh song song** trong graph.
- Mỗi nhánh thực hiện công việc riêng và trả về một phần kết quả.
- Bạn cần gom các kết quả đó về **một trạng thái chung** trước khi đi tiếp.

Mục đích	**Chi tiết**
Kết hợp dữ liệu	Khi có nhiều nhánh chạy song song
Nhận đầu vào	Một danh sách các trạng thái (dict)
Trả đầu ra	Một trạng thái duy nhất đã được tổng hợp
Ứng dụng	Tổng hợp kết quả LLM, kết hợp thông tin từ nhiều nguồn

Giả sử bạn có 3 nhánh song song:

- Node A xử lý phần tên.
- Node B xử lý địa chỉ.
- Node C xử lý số điện thoại.

Sau khi 3 node này chạy xong, bạn cần gộp kết quả lại thành:

```
{
  "name": "VHTSoft",
  "address": "TP. Hồ Chí Minh",
  "phone": "0123456789"
```

```
}
```

Reducer sẽ **gom và hợp nhất** những giá trị này thành một trạng thái chung.

Định nghĩa Reducer

Reducer là một hàm nhận vào nhiều trạng thái (dưới dạng danh sách) và trả về một trạng thái duy nhất.

```
def my_reducer(states: list[dict]) -> dict:
    result = {}
    for state in states:
        result.update(state)  # Gộp tất cả dict lại
    return result
```

Cách sử dụng trong LangGraph

Khi bạn xây dựng graph với `StateGraph`, bạn có thể chỉ định reducer cho một node nhất định như sau:

```
from langgraph.graph import StateGraph

builder = StateGraph(dict)
# Các node song song được thêm ở đây...

# Thêm reducer cho node tổng hợp
builder.add_reducer("merge_node", my_reducer)
```

Ví dụ thực tế

1. Khai báo các node

```
def node_1(state): return {"name": "VHTSoft"}
def node_2(state): return {"address": "TP.Hồ Chí Minh"}
def node_3(state): return {"phone": "0123456789"}
```

2. Tạo graph

```
builder = StateGraph(dict)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)
```

```
builder.add_node("merge_node", lambda x: x)  # Dummy node

# Thêm reducer
def my_reducer(states):  # states là list các dict
    result = {}
    for s in states:
        result.update(s)
    return result

builder.add_reducer("merge_node", my_reducer)

# Chạy 3 node song song → gộp về merge_node
builder.add_edge("node_1", "merge_node")
builder.add_edge("node_2", "merge_node")
builder.add_edge("node_3", "merge_node")

builder.set_entry_point("node_1")  # node_1 là điểm vào chính

graph = builder.compile()
```

Các hàm hữu ích về Reducers trong LangGraph

1. **Custom Logic trong Reducer**

Bạn không bị giới hạn bởi việc chỉ dùng `dict.update()`. Bạn có thể áp dụng **bất kỳ logic tùy chỉnh nào** để gộp dữ liệu:

Ví dụ: Lọc trạng thái hợp lệ

```
def filtered_reducer(states: list[dict]) -> dict:
    final_state = {}
    for state in states:
        # Bỏ qua những trạng thái thiếu dữ liệu cần thiết
        if "score" in state and state["score"] >= 0.8:
            final_state.update(state)
    return final_state
```

Dùng khi:

- Bạn chỉ muốn lấy kết quả “đủ tốt” từ các nhánh.
- Bạn có các nhánh xử lý có thể bị lỗi, hoặc chất lượng không đồng đều.

Trong ví dụ này, ta sẽ có 3 nhánh song song trả về thông tin khác nhau cùng với một chỉ số **độ tin cậy (`score`)**. Sau đó, ta sẽ dùng một `reducer` để **chỉ giữ lại những kết quả có `score >= 0.8`**.

Giả sử bạn có một graph với 3 agent xử lý một nhiệm vụ phân tích cảm xúc từ một đoạn văn. Mỗi agent trả về:

- `sentiment`: happy/sad/neutral
- `score`: độ tin cậy (confidence score)

Bạn chỉ muốn giữ lại kết quả của các agent **có độ tin cậy cao (score ≥ 0.8)**.

Bước 1: Tạo các node trả về kết quả

```
from langgraph.graph import StateGraph, END
from typing import TypedDict

class State(TypedDict):
    sentiment: str
    score: float

# Mỗi node trả về một kết quả khác nhau
def analyzer_1(state):
    return {"sentiment": "happy", "score": 0.9}

def analyzer_2(state):
    return {"sentiment": "sad", "score": 0.7}  # Không đạt

def analyzer_3(state):
    return {"sentiment": "neutral", "score": 0.85}
```

Bước 2: Tạo `filtered_reducer`

```
def filtered_reducer(states: list[dict]) -> dict:
    for state in states:
        if state.get("score", 0) >= 0.8:
            return state  # Trả về state đầu tiên đạt yêu cầu
    return {"sentiment": "unknown", "score": 0.0}
```

Bước 3: Dựng graph và chạy thử

```
graph = StateGraph(State)
graph.add_node("analyzer_1", analyzer_1)
```

```
graph.add_node("analyzer_2", analyzer_2)
graph.add_node("analyzer_3", analyzer_3)

graph.add_edge("analyzer_1", END)
graph.add_edge("analyzer_2", END)
graph.add_edge("analyzer_3", END)

graph.set_entry_point(["analyzer_1", "analyzer_2", "analyzer_3"])
graph.set_finish_point(END, filtered_reducer)

app = graph.compile()

result = app.invoke({})
print(result)
```

Kết quả mong đợi:

```
{'sentiment': 'happy', 'score': 0.9}
```

Vì analyzer_1 là node đầu tiên đạt yêu cầu `score >= 0.8`, nên nó được giữ lại. Các kết quả khác bị bỏ qua.

Biến thể nâng cao: Trả về danh sách tất cả các kết quả tốt

```
def filtered_reducer(states: list[dict]) -> dict:
    good_results = [s for s in states if s.get("score", 0) >= 0.8]
    return {"results": good_results}
```

Kết quả:

```
{
  'results': [
    {'sentiment': 'happy', 'score': 0.9},
    {'sentiment': 'neutral', 'score': 0.85}
  ]
}
```

2. **Merging trạng thái phức tạp (nested)**

Giả sử mỗi nhánh trả về trạng thái **dạng lồng nhau (nested)**:

```
{
  "agent": {
    "name": "Alice",
    "tasks": ["translate"]
  }
}
```

Bạn có thể merge có điều kiện, thậm chí hợp nhất danh sách:

```
def deep_merge_reducer(states: list[dict]) -> dict:
    merged = {"agent": {"name": "", "tasks": []}}
    for state in states:
        agent = state.get("agent", {})
        if "name" in agent:
            merged["agent"]["name"] = agent["name"]
        if "tasks" in agent:
            merged["agent"]["tasks"] += agent["tasks"]
    return merged
```

Giả sử bạn có một hệ thống thu thập thông tin sản phẩm từ nhiều nguồn. Mỗi node (agent) sẽ trả về thông tin chi tiết khác nhau về sản phẩm, như:

- `name`
- `price`
- `reviews`

Mỗi thông tin nằm trong một cấu trúc **nested dictionary**:

```
{
  "product": {
    "name": ...,
    "price": ...,
    "reviews": [...],
  }
```

```
}
```

Chúng ta sẽ merge lại tất cả thành một `state` hoàn chỉnh.

1. Định nghĩa state phức tạp

```
from typing import TypedDict, Optional
from langgraph.graph import StateGraph, END

class ProductInfo(TypedDict, total=False):
    name: Optional[str]
    price: Optional[float]
    reviews: list[str]

class State(TypedDict, total=False):
    product: ProductInfo
```

2. Các node thu thập thông tin khác nhau

```
def fetch_name(state):
    return {
        "product": {
            "name": "Smartphone X"
        }
    }

def fetch_price(state):
    return {
        "product": {
            "price": 799.99
        }
    }

def fetch_reviews(state):
    return {
        "product": {
            "reviews": ["Good value", "Excellent battery", "Fast delivery"]
        }
    }
```

3. Merge state phức tạp bằng custom reducer

LangGraph sẽ merge các dicts theo mặc định, **nhưng với dict lồng nhau**, bạn cần viết `custom reducer`.

```
def nested_merge_reducer(states: list[dict]) -> dict:
    merged = {"product": {}}
    for state in states:
        product = state.get("product", {})
        for key, value in product.items():
            if key == "reviews":
                merged["product"].setdefault("reviews", []).extend(value)
            else:
                merged["product"][key] = value
    return merged
```

4. Tạo Graph

```
graph = StateGraph(State)

graph.add_node("name", fetch_name)
graph.add_node("price", fetch_price)
graph.add_node("reviews", fetch_reviews)

graph.add_edge("name", END)
graph.add_edge("price", END)
graph.add_edge("reviews", END)

graph.set_entry_point(["name", "price", "reviews"])
graph.set_finish_point(END, nested_merge_reducer)

app = graph.compile()
result = app.invoke({})

print(result)
```

Kết quả mong đợi:

```
{
  "product": {
    "name": "Smartphone X",
    "price": 799.99,
    "reviews": [
```

```
      "Good value",
      "Excellent battery",
      "Fast delivery"
    ]
  }
}
```

Ghi chú:

- Nếu bạn không viết `reducer`, các dict lồng nhau sẽ **bị ghi đè**.
- `nested_merge_reducer` giúp **kết hợp thông minh** các phần tử trong `product`.

3. **Giữ thứ tự hoặc gán vai trò**

Trong một số trường hợp, bạn muốn biết được nhánh nào trả về cái gì, thay vì merge chung.

```
def role_based_reducer(states: list[dict]) -> dict:
    return {
        "summarizer_result": states[0]["output"],
        "validator_result": states[1]["output"],
        "rewriter_result": states[2]["output"],
    }
```

Ví Dụ

Bạn đang xây dựng một ứng dụng **hội thoại** với nhiều agent khác nhau tham gia đối thoại. Mỗi agent đóng một vai trò (ví dụ: Customer, Support, Bot), và bạn muốn giữ **thứ tự lời thoại** cùng với **vai trò** rõ ràng.

1. Định nghĩa `State`

```
from typing import TypedDict
from langgraph.graph import StateGraph, END

class Message(TypedDict):
    role: str
    content: str

class ChatState(TypedDict, total=False):
    messages: list[Message]
```

2. Các node đóng vai khác nhau

```
def customer_node(state):
    return {
        "messages": [
            {"role": "customer", "content": "Tôi muốn kiểm tra đơn hàng của mình."}
        ]
    }

def support_node(state):
    return {
        "messages": [
            {"role": "support", "content": "Chào anh/chị, vui lòng cung cấp mã đơn hàng ạ."}
        ]
    }

def bot_node(state):
    return {
        "messages": [
            {"role": "bot", "content": "Tôi có thể hỗ trợ những câu hỏi thường gặp. Bạn muốn hỏi gì?"}
        ]
    }
```

3. Custom reducer giữ thứ tự lời thoại

```
def ordered_reducer(states: list[dict]) -> dict:
    all_messages = []
    for state in states:
        msgs = state.get("messages", [])
        all_messages.extend(msgs)
    return {"messages": all_messages}
```

Lưu ý: `extend` giúp nối danh sách lời thoại từ các node theo **đúng thứ tự gọi**.

4. Xây dựng LangGraph

```
graph = StateGraph(ChatState)

graph.add_node("customer", customer_node)
graph.add_node("support", support_node)
graph.add_node("bot", bot_node)
```

```
graph.add_edge("customer", "support")
graph.add_edge("support", "bot")
graph.add_edge("bot", END)

graph.set_entry_point("customer")
graph.set_finish_point(END, reducer=ordered_reducer)

app = graph.compile()
result = app.invoke({})

from pprint import pprint
pprint(result)
```

Kết quả mong đợi:

```
{
 'messages': [
   {'role': 'customer', 'content': 'Tôi muốn kiểm tra đơn hàng của mình.'},
   {'role': 'support', 'content': 'Chào anh/chị, vui lòng cung cấp mã đơn hàng ạ.'},
   {'role': 'bot', 'content': 'Tôi có thể hỗ trợ những câu hỏi thường gặp. Bạn muốn hỏi gì?'}
 ]
}
```

4. Gộp theo trọng số(Weighted merge)

Khi bạn có nhiều nguồn dữ liệu và muốn **ưu tiên nguồn nào đó**, bạn có thể merge có trọng số.

```
def weighted_merge_reducer(states: list[dict]) -> dict:
    scores = [0.5, 0.3, 0.2]  # trọng số cho từng nhánh
    merged_output = ""
    for state, score in zip(states, scores):
        merged_output += f"{score:.1f} * {state['text']}\n"
    return {"combined_output": merged_output}
```

Ví dụ

Giả sử có 3 agent khác nhau đề xuất một câu trả lời cho cùng một câu hỏi, nhưng mỗi agent có **độ tin cậy khác nhau**. Bạn muốn:

- Gộp kết quả trả về của họ,
- Nhưng kết quả nào đáng tin hơn thì nên ảnh hưởng **nhiều hơn** đến kết quả cuối cùng.

1. Khai báo State

```
from typing import TypedDict

class State(TypedDict, total=False):
    suggestions: list[dict]  # Mỗi đề xuất có 'text' và 'score'
```

2. Tạo các node với "đề xuất" khác nhau

```
def agent_1(state):
    return {"suggestions": [{"text": "Trả lời từ agent 1", "score": 0.8}]}

def agent_2(state):
    return {"suggestions": [{"text": "Trả lời từ agent 2", "score": 0.6}]}

def agent_3(state):
    return {"suggestions": [{"text": "Trả lời từ agent 3", "score": 0.9}]}
```

3. Gộp theo trọng số(Weighted merge)

```
def weighted_merge(states: list[dict]) -> dict:
    from collections import defaultdict

    scores = defaultdict(float)

    for state in states:
        for suggestion in state.get("suggestions", []):
            text = suggestion["text"]
            score = suggestion.get("score", 1.0)
            scores[text] += score  # Cộng điểm nếu có trùng câu trả lời

    # Chọn text có tổng điểm cao nhất
    best_text = max(scores.items(), key=lambda x: x[1])[0]
    return {"final_answer": best_text}
```

defaultdict: xem chi tiết tại https://docs.vhterp.com/books/ky-thuat-lap-trinh-python/page/defaultdict

4. Xây dựng đồ thị

```
from langgraph.graph import StateGraph, END

graph = StateGraph(State)

graph.add_node("agent_1", agent_1)
graph.add_node("agent_2", agent_2)
graph.add_node("agent_3", agent_3)

graph.add_edge("agent_1", END)
graph.add_edge("agent_2", END)
graph.add_edge("agent_3", END)

graph.set_entry_point("agent_1")
graph.set_finish_point(END, reducer=weighted_merge)

app = graph.compile()
result = app.invoke({})

print(result)
```

Kết quả:

```
{'final_answer': 'Trả lời từ agent 3'}
```

5. Reducers + Pydantic = An toàn tuyệt đối

Khi reducer trả về một dict, bạn có thể dùng **Pydantic** để đảm bảo kết quả hợp lệ và chặt chẽ:

```
from pydantic import BaseModel

class FinalState(BaseModel):
    name: str
    summary: str
    mood: str

def reducer_with_validation(states):
    merged = {}
    for s in states:
        merged.update(s)
    return FinalState(**merged).dict()
```

Tình huống thực tế:

Giả sử bạn đang xây dựng một hệ thống trợ lý AI cho chăm sóc khách hàng. Bạn có nhiều "nhánh" (agents) cùng phân tích yêu cầu của người dùng để trích xuất thông tin: `name`, `email`, `issue`.

> Mỗi agent có thể phát hiện **một phần** thông tin. Bạn muốn:

- Gộp kết quả lại,
- Và đảm bảo kết quả cuối cùng **đúng định dạng**, **đầy đủ**, **an toàn**.

1. Định nghĩa `State` với Pydantic

```
from pydantic import BaseModel, EmailStr
from typing import Optional

class UserRequest(BaseModel):
    name: Optional[str]
    email: Optional[EmailStr]
    issue: Optional[str]
```

2. Tạo các Agent

```
def extract_name(state):
    return {"name": "Nguyễn Văn A"}

def extract_email(state):
    return {"email": "nguyenvana@example.com"}

def extract_issue(state):
    return {"issue": "Không đăng nhập được vào tài khoản"}
```

3. Tạo Reducer sử dụng Pydantic để kiểm tra tính hợp lệ

```
def safe_merge(states: list[dict]) -> dict:
    merged = {}
    for state in states:
        merged.update(state)

    # Kiểm tra hợp lệ bằng Pydantic
    validated = UserRequest(**merged)
```

```
    return validated.dict()
```

4. Xây dựng Graph

```
from langgraph.graph import StateGraph, END

graph = StateGraph(UserRequest)

graph.add_node("name", extract_name)
graph.add_node("email", extract_email)
graph.add_node("issue", extract_issue)

# Cho các node chạy song song
graph.add_edge("name", END)
graph.add_edge("email", END)
graph.add_edge("issue", END)

graph.set_entry_point("name")
graph.set_finish_point(END, reducer=safe_merge)

app = graph.compile()
result = app.invoke({})

print(result)
```

Kết quả

```
{
  'name': 'Nguyễn Văn A',
  'email': 'nguyenvana@example.com',
  'issue': 'Không đăng nhập được vào tài khoản'
}
```

6. Kết hợp với ToolCall

Nếu các nhánh là các công cụ (tools), bạn có thể dùng reducer để phân tích kết quả từ từng công cụ:

```
def tool_result_reducer(states):
    results = {}
```

```
    for state in states:
        tool_name = state["tool"]
        result = state["result"]
        results[tool_name] = result
    return {"tools_result": results}
```

Tình huống thực tế:

Bạn có một hệ thống AI assistant, khi người dùng hỏi:

> "Tôi tên là Minh, email của tôi là minh@example.com, tôi cần hỗ trợ về hóa đơn."

Bạn dùng LLM để tách thông tin đó, nhưng thay vì chỉ tách thủ công, bạn dùng **ToolCall** để giao cho từng tool xử lý riêng một phần.

1. Định nghĩa `Tool` và `State` với Pydantic

```
from pydantic import BaseModel, EmailStr
from typing import Optional
from langchain_core.tools import tool

class UserInfo(BaseModel):
    name: Optional[str]
    email: Optional[EmailStr]
    issue: Optional[str]
```

2. Định nghĩa các Tool dùng ToolCall

```
@tool
def extract_name_tool(text: str) -> dict:
    if "tên là" in text:
        return {"name": "Minh"}
    return {}

@tool
def extract_email_tool(text: str) -> dict:
```

```
    if "email" in text:
        return {"email": "minh@example.com"}
    return {}

@tool
def extract_issue_tool(text: str) -> dict:
    return {"issue": "hỗ trợ về hóa đơn"}
```

3. Dùng LangGraph gọi các Tool + Gộp lại bằng Reducer

```
from langgraph.graph import StateGraph, END
from langgraph.graph.message import ToolMessage
from langgraph.prebuilt.tool_node import ToolNode

# Khởi tạo ToolNode
tool_node = ToolNode(tools=[extract_name_tool, extract_email_tool, extract_issue_tool])

# Reducer sử dụng Pydantic để validate kết quả
def merge_tool_results(states: list[dict]) -> dict:
    merged = {}
    for state in states:
        merged.update(state)
    return UserInfo(**merged).dict()
```

4. Tạo Graph

```
graph = StateGraph(UserInfo)

# Tạo một node để gọi tool dựa vào user input
def prepare_tool_call(state):
    return ToolMessage(
        tool_name="extract_name_tool",  # Không cần quá cụ thể vì ToolNode sẽ tự chọn tool phù hợp
        tool_input={"text": "Tôi tên là Minh, email của tôi là minh@example.com, tôi cần hỗ trợ về hóa đơn."}
    )

graph.add_node("call_tool", prepare_tool_call)
graph.add_node("tool_node", tool_node)

graph.add_edge("call_tool", "tool_node")
graph.add_edge("tool_node", END)
```

```
graph.set_entry_point("call_tool")
graph.set_finish_point(END, reducer=merge_tool_results)

app = graph.compile()
result = app.invoke({})

print(result)
```

Kết quả

```
{
  "name": "Minh",
  "email": "minh@example.com",
  "issue": "hỗ trợ về hóa đơn"
}
```

Lợi ích của việc dùng ToolCall:

Tính năng	Lợi ích
Tách vai trò	Mỗi Tool xử lý một phần dữ liệu
Kết hợp logic	Dễ dàng mở rộng thêm Tool mà không phải viết lại toàn bộ
An toàn dữ liệu	`Pydantic` đảm bảo kết quả hợp lệ, đúng định dạng
Tự động	ToolNode sẽ **chọn tool phù hợp** dựa vào tên tool và input

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Public State, Private State và Multiple State Schemas trong LangGraph

Ví Dụ Minh Họa: Cửa Hàng Sửa Máy Tính

1. Public State - Giao Tiếp Đơn Giản Với Khách Hàng

- **Kịch bản**: Bạn mang máy tính bị hỏng đến cửa hàng sửa chữa.
- **Cuộc trò chuyện**:
  - Bạn: "Máy tính của tôi không hoạt động nữa, tôi không biết lý do."
  - Nhân viên: "Chúng tôi sẽ kiểm tra và báo lại cho bạn sau."
- **Đặc điểm**:
  - Thông tin **đơn giản**, dễ hiểu.
  - Không đi vào chi tiết kỹ thuật.

→ Đây chính là **public state** trong Landgraaf: trạng thái mà người dùng cuối (khách hàng) tương tác trực tiếp.

2. Private State - Giao Tiếp Nội Bộ Kỹ Thuật

- **Kịch bản**: Nhân viên chuyển máy tính cho đội kỹ thuật.
- **Cuộc trò chuyện**:
  - Kỹ thuật viên 1: "Kiểm tra nguồn điện, có thể do adapter."
  - Kỹ thuật viên 2: "Test RAM và ổ cứng, có thể bị bad sector."
- **Đặc điểm**:
  - Thông tin **kỹ thuật phức tạp**, chỉ dành cho nội bộ.
  - Khách hàng không cần biết chi tiết này.

→ Đây là **private state**: trạng thái nội bộ để xử lý logic phức tạp.

3. Kết Quả Cuối Cùng - Quay Lại Public State

- Kỹ thuật viên tổng hợp kết quả và báo lại cho nhân viên:
  - "Máy bị lỗi ổ cứng, cần thay thế với chi phí 3 triệu đồng."
- Nhân viên thông báo lại cho khách hàng:
  - "Máy của bạn bị hỏng ổ cứng, chúng tôi sẽ sửa trong 2 ngày."

→ Thông tin được **đơn giản hóa** từ private state trở lại public state.

Áp Dụng Vào LangGraph

1. Public State Schema

- **Vai trò**: Giao diện tương tác với người dùng.
- **Đặc điểm**:
  - Dữ liệu đầu vào/đầu ra đơn giản.
  - Ví dụ: Form đăng ký, kết quả tìm kiếm.

2. Private State Schema

- **Vai trò**: Xử lý logic nghiệp vụ phức tạp.
- **Đặc điểm**:
  - Chứa các biến và thuật toán kỹ thuật.
  - Ví dụ: Xử lý dữ liệu từ database, tính toán AI.

3. Chuyển Đổi Giữa Các Trạng Thái

- **Node 1 (Nhận Public State → Tạo Private State)**:
  - Nhận thông tin đơn giản từ người dùng.
  - **"Tăng độ phức tạp lên 1000 lần"** để xử lý nội bộ.
- **Node 2 (Nhận Private State → Trả Public State)**:
  - Tổng hợp kết quả phức tạp.
  - **"Giảm độ phức tạp 999 lần"** để trả về người dùng.

Tại Sao Điều Này Quan Trọng?

1. **Bảo Mật**: Private state giúp che giấu logic nhạy cảm khỏi người dùng.
2. **Hiệu Suất**: Tách biệt xử lý phức tạp khỏi giao diện người dùng.
3. **Dễ Bảo Trì**: Thay đổi logic kỹ thuật mà không ảnh hưởng đến trải nghiệm người dùng.

Lời Khuyên Khi Áp Dụng

- **Ghi nhớ khái niệm**: Hiểu rõ sự khác biệt giữa public/private state.
- **Đừng sa lầy vào chi tiết ngay**: Tập trung vào bài học thực hành trước.
- **Quay lại khi cần**: Khi xây dựng ứng dụng phức tạp, hãy tra cứu lại tài liệu này.

> **"LandGraph không chỉ dành cho siêu anh hùng - nó đơn giản nếu bạn hiểu cách tổ chức trạng thái!"**

Kết Luận

Việc phân chia **public/private state** giống như cách một cửa hàng sửa chữa hoạt động:

- **Public state** = Giao tiếp với khách hàng.
- **Private state** = Thảo luận nội bộ kỹ thuật.
  Hiểu được điều này sẽ giúp bạn thiết kế ứng dụng LangGraph mạnh mẽ và dễ bảo trì!

Multiple State Schemas Qua Ví Dụ Cửa Hàng Sửa Máy Tính

1. Multiple State Schemas Là Gì?

Trong LangGraph, **Multiple State Schemas** (Đa lược đồ trạng thái) là cách phân chia trạng thái ứng dụng thành nhiều "lớp" khác nhau, mỗi lớp có:

- **Mục đích riêng** (giao tiếp với người dùng vs xử lý nội bộ)
- **Độ phức tạp riêng** (đơn giản vs kỹ thuật)
- **Phạm vi truy cập riêng** (public vs private)

Giống như một cửa hàng có khu vực bán hàng (public) và xưởng sửa chữa (private).

2. Áp Dụng Vào Ví Dụ Cửa Hàng Sửa Máy Tính

Schema 1: Public State Schema

(Khu vực tiếp tân)

- **Đặc điểm**:
  - Ngôn ngữ: Đơn giản, không chuyên môn
  - Dữ liệu: Chỉ thông tin cần thiết cho khách hàng
- **Ví dụ**:
- ```
  // Public State Schema
  {
    customerName: "Nguyễn Văn A",
    device: "Laptop Dell XPS",
    problem: "Không khởi động được",
    status: "Đang chẩn đoán"
  }
  ```
  ```

Schema 2: Private State Schema

(Xưởng kỹ thuật)

- **Đặc điểm**:
  - Ngôn ngữ: Kỹ thuật, chi tiết
  - Dữ liệu: Đầy đủ thông tin để sửa chữa
- **Ví dụ**:
- ```
  // Private State Schema
  {
    deviceId: "DELL-XPS-2023-SN-12345",
    diagnosticLogs: [
      { test: "Power Supply", result: "12V rail unstable" },
      { test: "HDD S.M.A.R.T", result: "Reallocated sectors: 512" }
    ],
    repairPlan: [
      { action: "Replace PSU", part: "PSU-DELL-120W" },
      { action: "Clone HDD to SSD", tools: ["Acronis", "USB-SATA adapter"] }
    ]
  }
  ```

3. Cách Chuyển Đổi Giữa Các State Schema

Quy Trình Xử Lý

1. **Public → Private (Node 1)**
   - Nhận yêu cầu đơn giản từ khách
   - **"Mở rộng" thành thông số kỹ thuật**
   - ```
     def public_to_private(public_state):
         private_state = {
             'deviceId': lookup_device_id(public_state['device']),
             'diagnosticLogs': run_hardware_tests(),
             'repairPlan': generate_repair_options()
         }
         return private_state
     ```
2. **Private → Public (Node 2)**
   - Tổng hợp kết quả kỹ thuật
   - **"Rút gọn" thành thông báo dễ hiểu**

- ```
  def private_to_public(private_state):
      public_state_update = {
          'status': "Đã xác định lỗi",
          'solution': f"Thay {private_state['repairPlan'][0]['part']}",
          'cost': calculate_cost(private_state)
      }
      return public_state_update
  ```

4. Lợi Ích Của Multiple State Schemas

Tiêu Chí	Public State	Private State
Đối tượng	Khách hàng	Kỹ thuật viên
Độ phức tạp	1x (dễ hiểu)	1000x (chi tiết)
Bảo mật	Ai cũng thấy	Nội bộ
Ví dụ thực tế	App Mobile	Database + Microservices

Ưu điểm:

- Giảm tải thông tin cho người dùng cuối
- Dễ dàng thay đổi logic nghiệp vụ mà không ảnh hưởng giao diện
- Bảo mật thông tin nhạy cảm (giá vốn, lỗi hệ thống...)

5. Sai Lầm Cần Tránh

1. **Nhồi nhét private state vào public**
   → Làm người dùng bối rối với thông báo kiểu:
   "*Lỗi 0x7F do sector 512 bị bad, cần remap cluster*"
2. **Thiếu chuyển đổi giữa các schema**
   → Dẫn đến mất mát thông tin khi giao tiếp giữa các thành phần
3. **Dùng chung schema cho mục đích khác nhau**
   → Như bắt kỹ thuật viên dùng form đơn giản của khách để ghi chép

Câu Hỏi Thực Hành

Hãy thử thiết kế Multiple State Schemas cho các tình huống sau:

1. **Ứng dụng ngân hàng**
   - Public: Số dư tài khoản
   - Private: Lịch sử giao dịch chi tiết + thuật toán chống gian lận
  ```

2. **Trợ lý ảo y tế**
   - Public: Triệu chứng đơn giản ("Đau đầu 3 ngày")
   - Private: Dữ liệu EHR + mô hình chẩn đoán AI

> □□ **Mẹo**: Luôn tự hỏi "*Thông tin này có CẦN THIẾT cho người dùng cuối không?*" khi thiết kế schema.

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Thực hành Public State and Private State

1. Cài Đặt LangGraph

```
pip install langgraph
```

2. Triển Khai Code

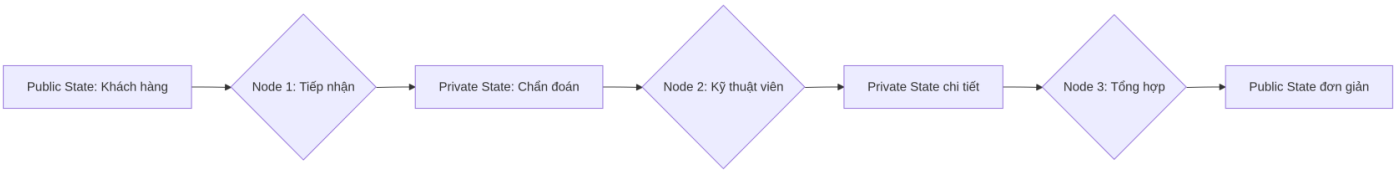
```
from langgraph.graph import Graph
from typing import Dict, Any

# ======================================
# ĐỊNH NGHĨA SCHEMAS
# ======================================

# Public State Schema (Dành cho khách hàng)
PublicState = Dict[str, Any]  # Ví dụ: {'customer_name': 'John', 'problem': 'Không khởi động'}

# Private State Schema (Dành cho kỹ thuật viên)
PrivateState = Dict[str, Any]  # Ví dụ: {'diagnostics': ['lỗi ổ cứng', 'pin hỏng'], 'repair_cost': 200}

# ======================================
# ĐỊNH NGHĨA CÁC NODE
# ======================================

def receive_complaint(state: PublicState) -> Dict[str, Any]:
    """Node 1: Tiếp nhận yêu cầu từ khách hàng (Public State)"""
    print(f"□□ Nhân viên nhận máy từ khách hàng: {state['customer_name']}")
    print(f"□□ Vấn đề mô tả: {state['problem']}")
    return {"internal_note": f"Khách hàng {state['customer_name']} báo: {state['problem']}"}

def diagnose_problem(state: Dict[str, Any]) -> PrivateState:
    """Node 2: Chuyển sang Private State để chẩn đoán"""
    print("\n□□ Kỹ thuật viên đang kiểm tra...")
```

```
    diagnostics = [
        "Test nguồn: OK",
        "Ổ cứng bị bad sector (LBA 1024-2048)",
        "Mainboard lỗi khe RAM"
    ]
    return {
        'diagnostics': diagnostics,
        'repair_cost': 350,
        'technician_notes': "Cần thay ổ cứng SSD 256GB + vệ sinh khe RAM"
    }

def summarize_report(state: PrivateState) -> PublicState:
    """Node 3: Chuyển kết quả về Public State cho khách hàng"""
    print("\n□□ Tổng hợp báo cáo đơn giản...")
    return {
        'customer_name': state['internal_note'].split()[2],  # Lấy tên từ internal_note
        'solution': "Thay ổ cứng SSD + bảo dưỡng",
        'cost': f"${state['repair_cost']}",
        'time_estimate': "2 ngày"
    }

# ========================================
# XÂY DỰNG GRAPH
# ========================================

workflow = Graph()

# Thêm các node vào graph
workflow.add_node("receive_complaint", receive_complaint)
workflow.add_node("diagnose_problem", diagnose_problem)
workflow.add_node("summarize_report", summarize_report)

# Kết nối các node
workflow.add_edge("receive_complaint", "diagnose_problem")
workflow.add_edge("diagnose_problem", "summarize_report")

# Biên dịch graph
chain = workflow.compile()

# ========================================
```

```
# CHẠY VÍ DỤ
# ========================================

# Khởi tạo Public State ban đầu
initial_state = {
    'customer_name': 'Nguyễn Văn A',
    'problem': 'Máy tính không lên nguồn',
    'status': 'received'
}

# Thực thi workflow
final_state = chain.invoke(initial_state)

# ========================================
# KẾT QUẢ
# ========================================
print("\n□□ Kết quả cuối cùng (Public State):")
print(final_state)
```

3. Giải Thích Hoạt Động

Luồng Dữ Liệu



Kết Quả Khi Chạy

```
□□ Nhân viên nhận máy từ khách hàng: Nguyễn Văn A
□□ Vấn đề mô tả: Máy tính không lên nguồn

□□ Kỹ thuật viên đang kiểm tra...

□□ Tổng hợp báo cáo đơn giản...

□□ Kết quả cuối cùng (Public State):
{
    'customer_name': 'Nguyễn',
    'solution': 'Thay ổ cứng SSD + bảo dưỡng',
```

```
    'cost': '$350',
    'time_estimate': '2 ngày'
}
```

4. Ghi nhớ

1. **Public State → Private State**
   - Node 1 nhận thông tin đơn giản, Node 2 mở rộng thành dữ liệu kỹ thuật.
2. **Private State → Public State**
   - Node 3 lọc bỏ chi tiết không cần thiết, chỉ giữ lại thông tin khách hàng cần biết.
3. **Ưu Điểm LangGraph**
   - Dễ dàng mở rộng thêm node (vd: thêm node "Xác nhận thanh toán")
   - Tách biệt rõ ràng giữa các tầng logic.

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft