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

Phiên bản #4
Được tạo 22 tháng 4 2025 03:39:40 bởi Đỗ Ngọc Tú
Được cập nhật 22 tháng 4 2025 05:27:51 bởi Đỗ Ngọc Tú