

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 để 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 **nhều 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.vhinterp.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	<code>Pydantic</code> 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**

Phiên bản #5

Được tạo 22 tháng 4 2025 13:34:23 bởi Đỗ Ngọc Tú

Được cập nhật 22 tháng 4 2025 15:14:11 bởi Đỗ Ngọc Tú