LangGraph入門

ノード・ステート・エージェント設計を学ぶ

© 2025 LangGraph Study

□□□□□

1. **LangGraph□□**
2. □□□□□□□□□□□□□□□□□□□□□
3. □□□□□□
    - ○ □□□□□□□□□□□□□□□
    - ○ □□□□□□□□□□□□
    - ○ □□□□□□□□□□□
    - ○ □□□□□□□□□□□□□□□□□□□
4. □□□□□□□□□
5. □□□□□□□□□□

© 2025 LangGraph Study

LangGraphとは

LangGraphは、LangChainベースのステートフルなマルチアクターアプリケーションを構築するフレームワークです。

特徴

- グラフ構造でワークフローを定義
- ステート管理が組み込み
- 条件分岐やループが簡単に実装可能
- マルチエージェントシステムの構築に最適

© 2025 LangGraph Study

□□LangGraph□□□□□□

□□□LangChain

- □□□□□□□□□□A → B → C□
- □□□□□□□□□□□□□

LangGraph□□□

- ✅ □□□□□□□□□□
- ✅ □□□□□□□□□□□□□
- ✅ □□□□□□□□□□□□□□□
- ✅ □□□□□□□□□

© 2025 LangGraph Study

[illegible]

© 2025 LangGraph Study

ステート（**State**）の概念

ステートは、グラフ全体で共有される状態情報です。

```python
from typing import Annotated, TypedDict
import operator

class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
    count: Annotated[int, operator.add]
```

ポイント

- `TypedDict` で型を定義
- `Annotated` で集約方法を指定（`operator.add` で追加）
- ノード間で状態を受け渡し

© 2025 LangGraph Study

ステートの集約戦略

`operator.add` の挙動

```
# 初期状態
state = {"messages": [msg1], "count": 1}

# ノードが返す値
return {"messages": [msg2], "count": 1}

# 集約後の状態
# messages: [msg1, msg2]  # リストに追加
# count: 2                # 数値を加算
```

重要: `operator.add` により、値が上書きではなく追加/加算されます。

© 2025 LangGraph Study

□□□□Node□□□□

□□□□□□□□□□□□□□□□□

```python
def my_node(state: AgentState) -> dict:
    """□□□□□"""
    # □□□□□□□
    messages = state["messages"]

    # □□□□□
    result = do_something(messages)

    # □□□□□□□□□
    return {"messages": [result]}
```

□□□□

- □□□□□□□□□State□
- □□□□□□□□□□□□

© 2025 LangGraph Study

□□□□Edge□□□□

□□□□□□□□□□□□□□□□□□□

□□□□□□

```
workflow.add_edge("NodeA", "NodeB")  # A → B
```

□□□□□□□

```
def should_continue(state):
    if state["done"]:
        return "end"
    return "continue"

workflow.add_conditional_edges(
    "NodeA",
    should_continue,
    {"continue": "NodeB", "end": END}
)
```

[illegible]Graph[illegible]

```
from langgraph.graph import StateGraph, END

# 1. [illegible]
workflow = StateGraph(AgentState)

# 2. [illegible]
workflow.add_node("NodeA", node_a_function)
workflow.add_node("NodeB", node_b_function)

# 3. [illegible]
workflow.add_edge("NodeA", "NodeB")
workflow.add_edge("NodeB", END)

# 4. [illegible]
workflow.set_entry_point("NodeA")

# 5. [illegible]
graph = workflow.compile()
```

© 2025 LangGraph Study

[illegible]

```
# [illegible]
result = graph.invoke({
    "messages": [],
    "count": 0
})

# [illegible]
print(result["messages"])
print(result["count"])
```

実例1: 基本的なグラフ

貯金箱アプリ（**work_1.py**）

© 2025 LangGraph Study

□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□

□□□

1. □□□Deposit□
2. □□□□□□
    - □□ → □□□Full□
    - □□□ → □□□□
3. □□□□□Full□

ステート定義

```python
from typing import Annotated, TypedDict
import operator

class PiggyBankState(TypedDict):
    total: Annotated[int, operator.add]       # 合計金額（加算）
    count: Annotated[int, operator.add]       # 貯金回数（加算）
    last_deposit: int                          # 最後の入金額
```

ポイント

- `total` と `count` は `operator.add` で累積
- `last_deposit` は常に上書き（Annotatedなし）

© 2025 LangGraph Study

ノード実装

```python
def deposit(state: PiggyBankState) -> dict:
    """入金ノード"""
    amount = int(input("Enter the amount to deposit: "))
    return {
        "total": amount,      # totalに加算される
        "count": 1,           # countに加算される
        "last_deposit": amount  # 上書きされる
    }

def finalize(state: PiggyBankState) -> dict:
    """終了ノード"""
    print(f"{state['count']}回の貯金で目標金額に到達しました。")
    print(f"{state['total']}円貯まっています。")
    return {"total": 0}
```

© 2025 LangGraph Study

条件分岐の実装

```python
def check_goal(state: PiggyBankState, goal: int) -> str:
    """目標金額をチェック"""
    if state["total"] >= goal:
        return "full"      # 目標達成
    else:
        return "continue"  # 継続
```

ポイント：戻り値は文字列（エッジ名）

© 2025 LangGraph Study

グラフ構造

```
graph TD
    Start([開始]) --> Deposit[入金]
    Deposit --> Check{目標達成?}
    Check -->|未達成| Deposit
    Check -->|達成| Full[終了処理]
    Full --> End([終了])
```

© 2025 LangGraph Study

□□□□□□

```
import functools

def piggy_bank(goal: int):
    workflow = StateGraph(PiggyBankState)

    # □□□□□
    workflow.add_node("Deposit", deposit)
    workflow.add_node("Full", finalize)

    # □□□□□□□
    workflow.add_conditional_edges(
        "Deposit",
        functools.partial(check_goal, goal=goal),
        {"continue": "Deposit", "full": "Full"}
    )

    workflow.add_edge("Full", END)
    workflow.set_entry_point("Deposit")

    return workflow.compile()
```

© 2025 LangGraph Study

[illegible]

```
Enter the amount to deposit: 300
Enter the amount to deposit: 400
Enter the amount to deposit: 500
3[illegible]
1200[illegible]
[illegible]500[illegible]
```

[illegible]

- [illegible]`operator.add`[illegible]
- [illegible]`add_conditional_edges`[illegible]
- [illegible]

© 2025 LangGraph Study

実例2: ツールを使うエージェント

シェルコマンド実行（**work_2.py**）

© 2025 LangGraph Study

シナリオ: シェルコマンド実行エージェント

LLMがツール（シェルコマンド）を使って質問に答えるエージェント

フロー

1. エージェント: ユーザーの質問を理解し、必要に応じてツールを呼び出す
2. ツール実行: シェルコマンドを実行
3. エージェント: 結果を元に回答を生成

© 2025 LangGraph Study

□□□□□□

```
from langchain_core.tools import tool
import subprocess

@tool
def exec_command(shell_command: str) -> str:
    """□□□□□□□□□□□□□□
    shell_command: Linux□□□□□□□
    """
    result = subprocess.run(
        shell_command,
        shell=True,
        capture_output=True
    )
    return result.stdout.decode("utf-8") + \
           result.stderr.decode("utf-8")
```

□□□□□□

```
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage
import operator

class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
```

□□□□

- □□□□□□□□□
- `operator.add`□□□□□□□□□□□

© 2025 LangGraph Study

LLM□□□□□□

```python
from langchain_openai import AzureChatOpenAI

llm = AzureChatOpenAI(
    azure_deployment=os.environ.get("AZURE_OPENAI_CHAT_DEPLOYMENT"),
    api_key=os.environ.get("AZURE_OPENAI_API_KEY"),
    azure_endpoint=os.environ.get("AZURE_OPENAI_ENDPOINT"),
    api_version=os.environ.get("AZURE_OPENAI_VERSION"),
    temperature=0,
)

# □□□□□□□□
llm_with_tool = llm.bind_tools([exec_command])
```

© 2025 LangGraph Study

エージェントノード

```python
def agent_node(state: AgentState):
    """エージェントノード: LLMを呼び出す"""
    messages = state["messages"]
    response = llm_with_tool.invoke(messages)
    return {"messages": [response]}
```

動作

1. 現在のメッセージ履歴を取得
2. LLMに送信（ツール呼び出しの判断も含む）
3. LLMの応答をメッセージとして返す

© 2025 LangGraph Study

□□□□□□

```
from langchain_core.messages import ToolMessage

def tool_node(state: AgentState):
    """□□□□□□□□□□□□□"""
    messages = state["messages"]
    last_message = messages[-1]

    tool_messages = []
    for call in last_message.tool_calls:
        if call["name"] == "exec_command":
            value = exec_command.invoke(call["args"])
            tool_message = ToolMessage(
                content=value,
                name=call["name"],
                tool_call_id=call["id"],
            )
            tool_messages.append(tool_message)

    return {"messages": tool_messages}
```

条件分岐：ツール呼び出しの判定

```python
def should_continue(state: AgentState):
    """ツールを呼び出すか、終了するかを判定"""
    messages = state["messages"]
    last_message = messages[-1]

    if last_message.tool_calls:
        return "tool"   # ツールを呼び出す
    else:
        return "end"    # 終了
```

グラフ構造

```
graph TD
    Start([開始]) --> Agent[エージェント]
    Agent --> Check{ツール呼び出し?}
    Check -->|Yes| Tool[ツール実行]
    Tool --> Agent
    Check -->|No| End([終了])
```

ポイント: エージェント ⇄ ツールのループ

© 2025 LangGraph Study

□□□□□□

```
workflow = StateGraph(AgentState)

# □□□□□
workflow.add_node("Agent", agent_node)
workflow.add_node("Tool", tool_node)

# □□□□□□□
workflow.add_conditional_edges(
    "Agent",
    should_continue,
    {"tool": "Tool", "end": END}
)

# □□□ → □□□□□□
workflow.add_edge("Tool", "Agent")

workflow.set_entry_point("Agent")
graph = workflow.compile()
```

□□□□□

```
query: □□□□□□□□□□□□□□□□□□□□

[□□□□]
1. Agent: "ls | wc -l" □□□□□□□□□
2. Tool: □□□□□□ → "42"
3. Agent: □□□□□□□□□

[□□]
□□□□□□□□□□□42□□□□□□□□□□□□
```

□□□□□

- □□□□□□
- □□□□□□ ⇄ □□□□□□□
- □□□□□□□□□□

© 2025 LangGraph Study

実例**3:** マルチエージェント対話

2エージェント対話（**work_3.py**）

© 2025 LangGraph Study

[illegible]

[illegible]

[illegible]

- [illegible]: [illegible]
- [illegible]: [illegible]

[illegible]

1. [illegible]
2. [illegible]
3. [illegible] or [illegible]

プロンプトテンプレート

```python
from langchain_core.prompts import ChatPromptTemplate

salesman_prompt = ChatPromptTemplate.from_messages([
    SystemMessage(
        "あなたは熱意ある壺のベテラン訪問販売員、坪田壺夫です。"
        "営業が終了したら、「FINISH」と回答してください。"
        "壺が売れるか、売れる見込みがない場合に営業を終了してください。"
    ),
    HumanMessage(content="こんにちは。どちら様でしょうか？"),
    ("placeholder", "{messages}"),
])

shed_prompt = ChatPromptTemplate.from_messages([
    SystemMessage("あなたは堅実な主夫の堅木実です。"),
    ("placeholder", "{messages}"),
])
```

© 2025 LangGraph Study

エージェント作成

```
from langchain_openai import AzureChatOpenAI

llm = AzureChatOpenAI(...)

salesman_agent = salesman_prompt | llm
shed_agent = shed_prompt | llm
```

ポイント

- プロンプトとLLMをパイプで接続（`|`）
- 各エージェントに異なるキャラクター設定

© 2025 LangGraph Study

ステート定義

```python
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
```

シンプル：メッセージ履歴のみを管理

© 2025 LangGraph Study

[illegible]

```python
from langchain_core.messages import AIMessage

def agent_node(state, agent, name):
    """[illegible]"""
    result = agent.invoke(state)
    message = AIMessage(
        **result.model_dump(exclude={"type", "name"}),
        name=name
    )
    print(f"{name}: {message.content}")
    return {"messages": [message]}
```

[illegible]

- `functools.partial` [illegible]
- [illegible] `name` [illegible]

□□□□□□□□

```python
import functools

salesman_node = functools.partial(
    agent_node,
    agent=salesman_agent,
    name="Salesman"
)

shed_node = functools.partial(
    agent_node,
    agent=shed_agent,
    name="SHED"
)
```

`functools.partial`□□□□□□□□

© 2025 LangGraph Study

終了条件の判定

```python
def route(state):
    """販売員の応答に基づいて次の行動を決定"""
    messages = state["messages"]
    last_message = messages[-1]

    if "FINISH" in last_message.content:
        return "finish"   # 営業終了
    return "continue"     # 対話継続
```

© 2025 LangGraph Study

グラフ構造

```
graph TD
    Start([開始]) --> Salesman[販売員]
    Salesman --> Check{FINISH?}
    Check -->|No| SHED[主夫]
    SHED --> Salesman
    Check -->|Yes| End([終了])
```

ポイント: 販売員 ⇄ 主夫のループ

□□□□□□

```
workflow = StateGraph(AgentState)

# □□□□□
workflow.add_node("Salesman", salesman_node)
workflow.add_node("SHED", shed_node)

# □□□□□□□
workflow.add_conditional_edges(
    "Salesman",
    route,
    {"continue": "SHED", "finish": END}
)

# □□ → □□□
workflow.add_edge("SHED", "Salesman")

workflow.set_entry_point("Salesman")
graph = workflow.compile()
```

© 2025 LangGraph Study

実行結果例

```
Salesman: こんにちは！素晴らしい壺をご紹介に参りました！
SHED: すみません、今は結構です。
Salesman: この壺は運気を上げると評判なんですよ！
SHED: 興味ないです。お帰りください。
Salesman: かしこまりました。FINISH
```

学んだこと

- 複数エージェントの協調
- プロンプトによるキャラクター設定
- 明確な終了条件の重要性

© 2025 LangGraph Study

実例**4:** 高度なマルチエージェント

開発チーム（**work_4.py**）

© 2025 LangGraph Study

シナリオ: ソフトウェア開発チーム

4つの役割を持つエージェントが協力してコードを開発

チームメンバー

- **Leader**: チームの進行管理、役割割り当て
- **Programmer**: 仕様に基づいてコードを作成
- **TestWriter**: テストコードを作成
- **Evaluator**: コードとテストを実行、結果を分析

© 2025 LangGraph Study

ステート定義

```python
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
    next: str        # 次に実行するメンバー
    task: str        # 開発タスク
```

ポイント

- `messages`：会話履歴
- `next`：リーダーが決定する次の担当者
- `task`：開発する機能の仕様

© 2025 LangGraph Study

Structured Outputの活用

```
from pydantic import BaseModel, Field

class LeaderResponse(BaseModel):
    reasoning: str = Field(description="決定の背後にある理由。")
    next: Union[Literal["Finish"], str] = Field(
        description="次にアクションを行うチームメンバ。"
    )
    instructions: str = Field(description="次のチームメンバへの指示。")
```

ポイント: LLMの出力を構造化し、確実に `next` を取得

© 2025 LangGraph Study

[illegible]

```python
@tool
def evaluate(code: str, test: str) -> tuple[str, str]:
    """
    [illegible]
    """
    with open("product.py", "w") as f:
        f.write(code)
    with open("test_product.py", "w") as f:
        f.write(test)
    result = subprocess.run(
        ["pytest", "test_product.py"],
        capture_output=True
    )
    return result.stdout.decode(), result.stderr.decode()
```

Evaluator[illegible]

□□□□□□□□

```
def create_agent(llm, name: str):
    """□□□□□□□□□□□□□□"""
    prompt = ChatPromptTemplate.from_messages([
        SystemMessagePromptTemplate.from_template(
            "□□□□□□□□□□□□□□□□□□□{name}□□□"
            "□□□□□□: {members}\n\n"
            "{member_roles}"
        ),
        HumanMessagePromptTemplate.from_template("task: {task}\n"),
        MessagesPlaceholder("messages"),
    ]).partial(name=name, members=members, member_roles=member_roles)

    return prompt | llm
```

© 2025 LangGraph Study

[illegible]

```
llm = AzureChatOpenAI(...)

# Leader[illegible]
leader_agent = create_agent(
    llm.with_structured_output(LeaderResponse),
    "Leader"
)

programmer_agent = create_agent(llm, "Programmer")
tester_agent = create_agent(llm, "TestWriter")
evaluator_agent = create_agent(
    llm.bind_tools([evaluate]),
    "Evaluator"
)
```

リーダーノード

```python
def leader_node(state: AgentState) -> dict:
    """リーダエージェント"""
    response = leader_agent.invoke(state)
    return {
        "messages": [
            HumanMessage(content=response.instructions, name="Leader")
        ],
        "next": response.next,  # 次の担当者を決定
    }
```

ポイント: `next` を更新して次のメンバーを指定

© 2025 LangGraph Study

□□□□□□□

```
def member_node(state: AgentState, agent, name: str) -> dict:
    """□□□□□□□□□"""
    result = agent.invoke(state)

    # □□□□□□□□□□□□□□□□□□□

    return {
        "messages": [
            AIMessage(**result.model_dump(exclude={"type", "name"}), name=name)
        ]
    }

# □□□□□□□□□□□
programmer_node = functools.partial(member_node, agent=programmer_agent, name="Programmer")
tester_node = functools.partial(member_node, agent=tester_agent, name="TestWriter")
evaluator_node = functools.partial(member_node, agent=evaluator_agent, name="Evaluator")
```

© 2025 LangGraph Study

ツールノード（**ToolNode**）

```python
from langgraph.prebuilt import ToolNode

# ツールノードを作成
tool_node = ToolNode([evaluate])
```

ポイント: `ToolNode` はツール呼び出しを自動処理

© 2025 LangGraph Study

[illegible]

```python
def router(state: AgentState) -> str:
    """Evaluator[illegible]"""
    messages = state["messages"]
    last_message = messages[-1]

    if last_message.tool_calls:
        return "call_tool"
    return "continue"
```

□□□□□

```
graph TD
    Start([□□]) --> Leader[Leader]
    Leader --> Decision{next}
    Decision -->|Programmer| Prog[Programmer]
    Decision -->|TestWriter| Test[TestWriter]
    Decision -->|Evaluator| Eval[Evaluator]
    Decision -->|Finish| End([□□])
    Prog --> Leader
    Test --> Leader
    Eval --> ToolCheck{□□□□□□□?}
    ToolCheck -->|Yes| Tool[□□□□□]
    ToolCheck -->|No| Leader
    Tool --> Eval
```

[illegible]

```
workflow = StateGraph(AgentState)

# [illegible]
workflow.add_node("Leader", leader_node)
workflow.add_node("Evaluator", evaluator_node)
workflow.add_node("Tool", ToolNode([evaluate]))
workflow.add_node("Programmer", programmer_node)
workflow.add_node("TestWriter", tester_node)

# Evaluator[illegible]
workflow.add_conditional_edges(
    "Evaluator",
    router,
    {"continue": "Leader", "call_tool": "Tool"}
)

# Leader[illegible]
workflow.add_conditional_edges(
    "Leader",
    lambda x: x["next"],
    {
        "Programmer": "Programmer",
        "TestWriter": "TestWriter",
        "Evaluator": "Evaluator",
        "Finish": END,
    },
)

workflow.add_edge("Programmer", "Leader")
workflow.add_edge("TestWriter", "Leader")
workflow.add_edge("Tool", "Evaluator")

workflow.set_entry_point("Leader")
```

実行結果例

```
task: フィボナッチ数列のN番目を返す関数を実装してください

Leader: Programmerさん、フィボナッチ関数を実装してください
Programmer: [コード実装]
Leader: TestWriterさん、テストを書いてください
TestWriter: [テスト実装]
Leader: Evaluatorさん、実行して確認してください
Evaluator: [ツール呼び出し] → テスト成功
Leader: 完了しました。Finish
```

© 2025 LangGraph Study

学んだこと

- 役割分担: 各エージェントに明確な責任
- **Structured Output**: LLMの出力を構造化
- **ToolNode**: ツール呼び出しの簡略化
- 動的ルーティング: `lambda x: x["next"]` でリーダーの判断に基づく分岐
- 複雑なワークフロー: 複数のノードと条件分岐の組み合わせ

© 2025 LangGraph Study

ベストプラクティス

© 2025 LangGraph Study

ステート設計のコツ

1. 必要最小限の情報

```python
# ❌ 過剰
class State(TypedDict):
    messages: List[BaseMessage]
    message_count: int  # messagesから計算可能
    last_message: str   # messagesから取得可能

# ✅ シンプル
class State(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
```

2. 適切な集約戦略

- 追加/加算: `operator.add`
- 上書き: Annotatedなし

□□□□□□

1. □□□□□□

```python
# ❌ □□□□□
def process_node(state):
    data = fetch_data()       # □□□□□
    result = analyze(data)    # □□
    save(result)              # □□
    return {"result": result}

# ✅ □□□□□
def fetch_node(state): ...
def analyze_node(state): ...
def save_node(state): ...
```

© 2025 LangGraph Study

ノード設計のコツ（続き）

2. 純粋関数として設計

```python
# ❌ 副作用あり（グローバル変数の変更）
global_counter = 0
def bad_node(state):
    global global_counter
    global_counter += 1
    return {"count": global_counter}

# ✅ 純粋関数
def good_node(state):
    return {"count": 1}  # operator.addで加算される
```

ポイント：ステートのみを使い、外部状態に依存しない

エージェント設計のコツ

1. 明確な役割定義

```
MEMBERS = {
    "Leader": "チームの進行を管理し、役割を割り当てる",
    "Programmer": "仕様に基づいてコードを書く",
    "TestWriter": "テストを書く",
    "Evaluator": "コードとテストを実行し、結果を分析する",
}
```

役割が明確であるほど、エージェントは効果的に動作します。

© 2025 LangGraph Study

エージェント設計のコツ（続き）

2. 適切な終了条件

```python
# ✅ 明確な終了条件
def should_continue(state):
    if state["goal_reached"]:
        return "end"
    if state["max_iterations"] >= 10:
        return "end"  # 無限ループ防止
    return "continue"
```

重要：無限ループを防ぐため、必ず終了条件を設定

© 2025 LangGraph Study

デバッグとテスト

1. 状態の可視化

```python
result = graph.invoke(initial_state)

# ステップごとの状態を確認
for step in result:
    print(f"Step: {step}")
    print(f"State: {result[step]}")
```

2. ノード単体のテスト

```python
# ノードを個別にテスト
state = {"messages": [HumanMessage(content="test")]}
result = my_node(state)
assert "messages" in result
```

© 2025 LangGraph Study

[illegible]

© 2025 LangGraph Study

学んだこと

1. ステート（**State**）：TypedDict + Annotatedで状態管理
2. ノード（**Node**）：処理単位、入力はState、出力は更新する状態
3. エッジ（**Edge**）：ノード間の接続、条件分岐も可能
4. グラフ（**Graph**）：StateGraph → add_node/add_edge → compile

実例で学んだパターン

- 基本的なループと条件分岐（貯金箱）
- ツールを使うエージェント（シェルコマンド）
- マルチエージェント対話（販売員と主夫）
- 高度なマルチエージェント（開発チーム）

© 2025 LangGraph Study

次のステップ

1. 公式ドキュメント
   - LangGraph Documentation
   - より高度な機能を学ぶ
2. より高度なパターン
   - **Persistence**: グラフの状態を永続化
   - **Human-in-the-loop**: 人間の承認を含むワークフロー
   - **Streaming**: リアルタイムな出力
   - **Subgraphs**: グラフのネスト
3. 実践
   - □□□□□□□ースに適用

© 2025 LangGraph Study

Q&A

ご質問はありますか?

© 2025 LangGraph Study

ありがとうございました

Happy Coding with LangGraph!

© 2025 LangGraph Study