LangGraph 核心知识速记手册 (八股文)

第一部分：核心概念与定位

1. LangGraph 是什么?

LangGraph 是一个用于构建**有状态、多智能体（multi-actor）*应用程序的框架。它基于 LangChain 构建，核心优势在于允许开发者创建*包含循环（cycles）的、图（Graph）结构**的工作流。这使得构建能够进行迭代、反思和动态决策的复杂 AI 代理（Agent）成为可能。

2. LangGraph 与 LangChain 的关系?

- **继承与扩展**：LangGraph 是 LangChain 生态的一部分，它**扩展**而非取代 LangChain。LangChain 提供了构建块（如模型、工具、提示词），而 LangGraph 提供了将这些构建块组织成**有状态、可循环的、更强大代理系统**的编排引擎。
- **核心区别**：
  - **LangChain**：擅长构建**线性**的、有向无环图（DAG）式的链（Chains）。
  - **LangGraph**：专注于构建**循环图**，通过显式的状态管理和节点间的边来控制复杂的、非线性的控制流。

3. LangGraph 的核心优势是什么?

- **循环与迭代**：原生支持循环图，这是实现 ReAct、反思（Reflection）、计划-执行等高级代理模式的基础。
- **状态管理**：引入了中心化的**状态（State）**对象，在图的节点之间传递和更新，使得记忆、持久化和容错成为可能。
- **人机协同 (HIL)**：内置在图的任意节点暂停执行、等待人工审批或输入，然后再继续执行的能力。
- **底层与可扩展性**：提供底层的图构建原语（State, Node, Edge），让开发者可以构建完全定制化的代理，而不是被固化的抽象所限制。
- **一流的流式处理**：支持令牌级和中间步骤的流式输出，为用户提供代理推理过程的实时可见性。

第二部分：核心构建模块 (State, Node, Edge)

构建一个 LangGraph 应用的核心就是定义这三个组件。

1. 状态 (State)

状态是图的“共享内存”或“黑板”，它是一个在所有节点间传递和演进的数据结构。

- **定义方式**：
  - `TypedDict` **(推荐)**：Python 标准库类型，轻量且清晰。
  - `Pydantic BaseModel`：提供运行时数据验证和默认值，更安全。
- **状态更新**：节点执行后返回一个字典，该字典中的值会更新到主状态中。
- **Reducer (合并器)**：可以为状态中的特定字段指定一个合并函数（Reducer），以控制更新行为。
  - **默认行为**：覆盖 (Overwrite)。
  - **自定义 Reducer**：例如 `operator.add` 可以将新项**追加**到列表中，而不是覆盖整个列表。通过 `typing.Annotated` 指定。

```python
from typing import Annotated
import operator

class AgentState(TypedDict):
    # messages 字段的更新方式是追加，而不是覆盖
    messages: Annotated[list, operator.add]
```

- `add_messages` **Reducer**：一个为处理对话历史（消息列表）专门优化的内置 Reducer，能智能地追加新消息或根据 ID 更新已存在的消息。

2. 节点 (Node)

节点是图中的基本计算单元，代表一个处理步骤。

- **实现方式**：可以是一个 Python 函数（同步或异步），也可以是一个 LangChain LCEL `Runnable` 对象。
- **函数签名**：节点的第一个参数总是当前的**图状态 (State)**。
- **返回值**：节点函数返回一个**字典**，其中包含对状态的更新。
- **特殊节点**：
  - `START`：图的虚拟入口点。
  - `END`：图的特殊终止节点。当执行流到达 `END`，该路径结束。

3. 边 (Edge)

边定义了节点之间的连接关系和执行流程。

- **普通边 (`add_edge`)**：无条件转换。当一个节点完成后，总是流向下一个指定的节点。
- **条件边 (`add_conditional_edges`)**：**实现循环和动态决策的核心。**
  - 它接收一个**源节点**和一个**路径函数 (path function)**。
  - **路径函数**会接收当前的状态 (State)，并返回一个字符串，该字符串决定了下一个要执行的节点的名称。
  - 这使得图可以根据 LLM 的输出、工具的结果或任何状态中的数据来动态地选择下一步的路径。

第三部分：图的执行与管理

1. 编译 (`compile()`)

- **作用**：将声明式定义的 `StateGraph` 转换为一个可执行的 `Runnable` 对象。
- **关键参数**：
  - `checkpointer`：传入一个检查点对象，用于实现持久化。
  - `interrupt_before` / `interrupt_after`：用于设置静态断点，实现人机协同。

2. 执行

编译后的图对象支持标准的 LangChain `Runnable` 接口：

- `invoke(input, config)`：同步执行，一次请求-响应。

- `stream(input, config, stream_mode)`: 流式执行，逐步返回中间状态或最终结果的令牌。`stream_mode` 参数可以控制流式输出的内容（如 `"values"`, `"updates"`, `"messages"`）。
- `batch(inputs, config)`: 并行处理多个输入。
- **异步方法**: `ainvoke`, `astream`, `abatch`。

3. 持久化 (Checkpointers)

持久化是实现有状态应用（如多轮对话）的基石。

- **机制**：通过在 `compile()` 时提供 `checkpointer` 对象实现。检查点记录器会在每个步骤后自动保存图状态的快照。
- `thread_id`：调用图时，必须在 `config` 中提供一个唯一的 `thread_id`，用于标识和关联一个独立的执行序列（例如，一个用户的单次对话）。
- 实现后端：

Checkpointer	存储介质	主要用途
MemorySaver	内存	测试、教程、快速原型
SqliteSaver	SQLite 文件	本地开发、小型项目
PostgresSaver	PostgreSQL 数据库	生产环境推荐，健壮、可扩展
RedisSaver	Redis	需要快速读写的分布式缓存场景

- **时间旅行 (Time Travel)**：配置了检查点后，可以获取历史状态 (`get_state_history`)，从任意一个历史快照恢复执行，或修改某个历史状态后开启新的分支。

第四部分：高级特性与 Agent 设计模式

1. 工具使用 (Tools)

- `ToolNode` **(from** `langgraph.prebuilt`**)**: 一个预构建的节点，专门用于执行 LLM 请求的工具。它会自动解析 `AIMessage` 中的 `tool_calls`，并行执行工具，并将结果包装成 `ToolMessage` 返回。
- `tools_condition` **(from** `langgraph.prebuilt`**)**: 一个用于条件边的实用函数，它检查最新消息中是否包含工具调用请求，从而决定是路由到 `ToolNode` 还是结束。这是实现 **ReAct 模式** 的关键。

2. 人机协同 (Human-in-the-Loop, HIL)

- **静态断点**: 在 `compile()` 时通过 `interrupt_before` 或 `interrupt_after` 参数在特定节点前后设置暂停点。
- **动态中断 (** `interrupt()` **)**: 在节点函数内部调用 `interrupt()` 函数，可以根据逻辑动态触发暂停。
- **恢复执行 (** `Command` **)**: 当图暂停后，通过 `graph.invoke(Command(resume=...))` 并传入人工输入来恢复执行。

3. 多智能体协作 (Multi-Agent)

LangGraph 是构建多智能体系统的理想框架。

- **主管-专员模式 (Supervisor-Worker)**:
  - 一个中心的“主管”Agent 负责分解任务和协调。
  - 多个“专员”Agent 负责执行具体的子任务。
  - 通信通常是 `主管 -> 专员 -> 主管`。
  - 使用 `langgraph-supervisor` 库可以快速搭建。
- **集群模式 (Swarm)**:
  - 一种更去中心化的模式，Agent 之间根据专长动态地相互“切换”控制权。
  - 使用 `langgraph-swarm` 库和 `create_handoff_tool` 实现。
- `Command.PARENT`: 子图中的节点可以使用此命令将控制权和状态更新交还给父图，是实现层级式 Agent 的关键。

4. 核心 Agent 设计模式

LangGraph 的循环和状态管理能力使其能够优雅地实现多种高级 Agent 设计模式:

设计模式	核心思想	LangGraph 实现关键
ReAct (推理-行动)	LLM 在“思考”和“调用工具”之间循环。	`ToolNode` + `tools_condition` 条件边。
计划-执行	先生成一个多步计划，然后逐一执行，并可根据结果重新规划。	状态中包含 `plan` 和 `past_steps`，通过循环和条件边控制执行与重新规划。
反思与自我批判	Agent 生成初步输出后，由另一个“批判者”LLM对其进行评估和提出修改建议，形成“生成-反思-再生成”的循环。	图中包含“生成器”和“反思器”两个节点，通过循环连接，直到满足终止条件。
投票/辩论	多个 Agent 并行处理同一问题，然后由一个“聚合器”或“裁判”Agent 综合它们的输出。	使用 `Send` API 并行分发任务，再用一个聚合节点收集所有结果。

5. 评估 (Evaluation)

- **与 LangSmith 集成**：LangGraph 与 LangSmith 无缝集成，可以自动追踪每一次运行的详细轨迹，包括节点输入输出、工具调用和状态变化。
- **系统性评估流程**:
  1. **创建数据集**：在 LangSmith 中创建包含输入和期望输出的“考卷”。
  2. **定义评估器**：使用内置的或自定义的评估器来为 Agent 的输出打分（例如，`correctness`）。
  3. **运行评估**：使用 `evaluate` 函数将你的 Agent 应用于整个数据集，并获得详细的性能报告。