第 22 课：LangGraph 基础

这一课你先记一句话：

LangChain 更像“快速搭 Agent 的上层框架”，LangGraph 更像“把复杂 Agent 拆成状态机/流程图来控制的编排框架”。

LangGraph 官方现在强调的是：它是面向 Agent 编排的运行时，重点能力包括 **durable execution、streaming、human-in-the-loop、persistence**；并且 LangChain 的 agents 也是构建在 LangGraph 之上的。

1. 为什么有了 LangChain，还需要 LangGraph？

因为简单 Agent 可以这样写：

```
用户输入
  -> LLM 判断
  -> 调工具
  -> 返回结果
```

但你的项目不是简单 Agent。你的项目有：

```
用户输入
  -> Planner
  -> Memory
  -> Draft
  -> Confirm
  -> Execution
  -> Audit
  -> Response
```

这已经不是一条简单 chain，而是一个带状态、分支、暂停、恢复、确认、审计的复杂流程。

所以 LangGraph 解决的是：

当 Agent 不是简单“一问一答”，而是有多个步骤、多个分支、状态保存、人工确认、失败恢复时，用图结构来管理流程会更清晰。

官方文档也说，使用 LangGraph 构建 agent 时，通常会先把流程拆成离散步骤，也就是 nodes，然后描述节点之间的决策和转移，最后通过共享 state 把节点连接起来。

2. LangGraph 的 4 个核心概念

先只掌握这 4 个：

```
State：状态
Node：节点
Edge：边
Conditional Edge：条件边
```

不用一上来背太多 API。

3. State 是什么？

State 就是整个 Agent 流程共享的状态对象。

你可以理解成：

```
class AgentState:
    user_input: str
    intent: str
    tool_plan: dict
    retrieved_docs: list
    draft_id: str | None
    pending_action: dict | None
    final_answer: str | None
    trace_id: str
```

在 LangGraph 里，各个节点会读取和更新这个 state。官方文档把这种思路描述为：节点之间通过共享 state 连接，每个 node 可以读取 state，也可以写入 state

你项目状态	LangGraph State 里的字段
用户输入	user_input
Planner 结果	tool_plan
L1 记忆	last_ticket_id, last_draft_id
草稿	draft_id, missing_fields
确认动作	pending_action
执行结果	tool_result
审计追踪	trace_id
最终回复	final_answer

面试表达：State 是 LangGraph 中跨节点传递的共享状态。每个节点不应该乱传一堆零散参数，而是从 State 读取输入，再把自己的输出写回 State。这样复杂 Agent 的执行过程更容易追踪和恢复。

4. Node 是什么？

Node 就是流程里的一个处理步骤。

比如你的项目可以拆成这些节点：

```
planner_node
memory_node
rag_node
draft_node
confirm_node
execute_node
audit_node
response_node
```

每个 node 做一件明确的事。

例如：

```
planner_node：判断用户意图，生成 ToolPlan
memory_node：解析“上一单”“刚才那个草稿”
rag_node：执行知识库检索
confirm_node：判断是否需要用户确认
execute_node：执行后端工具
audit_node：写审计日志
response_node：组织最终回复
```

面试表达：Node 是 LangGraph 里的一个执行步骤，可以是一次 LLM 调用、一次工具调用、一次检索、一次权限校验，或者一次审计记录。好的设计是让每个 Node 职责单一，方便测试和排查问题。

这一点和你前面学的 service 分层类似：不要所有逻辑塞在一个函数里。

5. Edge 是什么？

Edge 就是节点之间的连接关系。

最简单的是固定流程：

```
START
  -> planner_node
  -> execute_node
  -> response_node
  -> END
```

也就是：A 执行完一定进入 B。

在普通 workflow 里，这种固定边很常见。

面试表达：Edge 表示节点之间的执行顺序。普通 edge 是固定流转，比如 planner 执行完一定进入 executor，executor 执行完一定进入 response。

6. Conditional Edge 是什么？

Conditional Edge 是条件边，也就是根据 State 决定下一步去哪里。

这是 LangGraph 最重要的点之一。

比如你的项目里，Planner 生成结果后，下一步不是固定的：

```
如果是 RAG 问答
  -> rag_node

如果是创建工单但缺字段
  -> draft_node

如果是取消工单
  -> confirm_node

如果是查单
  -> execute_node

如果意图不清楚
  -> clarify_node
```

这就是 conditional edge。

你可以画成：

```
planner_node
  ├── intent = kb_answer            -> rag_node
  ├── intent = create_ticket        -> draft_or_execute_node
  ├── intent = cancel_ticket        -> confirm_node
  ├── intent = get_ticket           -> execute_node
  └── intent = unclear              -> clarify_node
```

面试表达：Conditional Edge 是根据当前 State 决定下一个节点。它适合 Agent 里的意图分流、工具选择、缺字段澄清、高风险确认和异常处理。

这正好对应你的项目核心。

7. LangGraph 和普通工作流有什么区别？

普通工作流一般是确定性的：

```
A -> B -> C -> D
```

比如：

```
接收请求 -> 参数校验 -> 查询数据库 -> 返回响应
```

LangGraph 更适合这种场景：

```
A -> 根据状态选择 B/C/D
B -> 可能暂停等待用户确认
C -> 可能调用工具后再回到模型
D -> 可能失败重试
```

也就是：

```
有状态
有分支
有循环
可暂停
可恢复
可人工介入
```

LangGraph 官方强调了 durable execution、human-in-the-loop、persistence 等能力；其中 persistence 会

把 graph state 保存为 checkpoint，支持人工介入、会话记忆、调试和故障恢复。
面试表达：普通 workflow 更适合固定流程，LangGraph 更适合复杂 Agent。因为 Agent 需要根据模型输出和工具结果动态选择下一步，还可能需要人工确认、状态保存和失败恢复。

8. Human-in-the-loop 是什么？

Human-in-the-loop 就是人在关键节点介入。
比如用户说：

```
取消上一单
```

系统不能直接取消，而是：

```
planner_node
  -> confirm_node
  -> 暂停，等待用户确认
  -> 用户确认后继续 execute_node
```

LangGraph 的 interrupts 机制可以在某个节点暂停图执行，等待外部输入，然后基于持久化状态继续执行；这正适合审批、确认、高风险操作等场景。
你的项目里对应：

```
pending_action
用户确认
继续执行
audit_log
```

面试表达：Human-in-the-loop 是在高风险或不确定步骤中让用户或人工审核介入。比如取消工单、删除数据、提交审批这类操作，Agent 可以先生成 pending_action，暂停流程，等用户确认后再继续执行。

9. Checkpoint / Persistence 是什么？

这个先简单理解：

```
Checkpoint 就是把当前 State 保存下来。
Persistence 就是让这些状态可以持久化，不因为请求结束或服务异常就丢失。
```

LangGraph 的 persistence 文档说明，编译 graph 时如果配置 checkpointer，系统会在每个执行步骤保存 graph state 的快照，并按 thread 组织；这可以支持 human-in-the-loop、会话记忆、time travel debugging 和故障恢复。
你项目里的对应：

```
L1 session memory
ticket_draft
pending_action
trace_id
audit_log
```

虽然你不是完全用 LangGraph checkpoint，但思想类似：

```
用户说“继续刚才那个”
  -> 从 L1 / draft 表恢复上下文

用户确认“是的，取消”
  -> 从 pending_action 恢复上一步待执行动作
```

面试表达：Checkpoint 的价值是让 Agent 流程可以暂停和恢复。对于多轮任务，比如补全工单字段、等待用户确认、失败后重试，如果没有状态持久化，Agent 很容易丢上下文。

10. 你的项目如何用 LangGraph 思路包装？

你的 Policy KB Assistant 可以这样讲：

```
START
  -> parse_request_node
  -> memory_node
  -> planner_node
  -> conditional_edge
```

```
        ├── kb_answer -> rag_node -> answer_node
        ├── create_ticket  缺字段  -> draft_node -> clarify_node
        ├── create_ticket  字段齐全  -> execute_node -> audit_node
        ├── cancel_ticket -> confirm_node -> pending_action
        ├──  用户确认  -> execute_pending_node -> audit_node
        └── unclear -> clarify_node
    -> END
```

这就是一个典型 LangGraph 风格的 Agent。

你可以在面试里说：我的项目虽然没有完全依赖 LangGraph，但整体设计很接近 LangGraph 的状态机思想。我把 Agent 流程拆成 Planner、Memory、Draft、Confirm、Execution、Audit 等节点，通过状态字段传递 ToolPlan、draft_id、pending_action 和 trace_id，再根据 intent、missing_fields、need_confirm 等字段进行条件分流。

这段非常适合背。

11. LangGraph 和你前面学的 ToolPlan 怎么结合？

ToolPlan 是 Planner Node 的输出。

```
planner_node
  -> 输出 ToolPlan
```

然后 Conditional Edge 根据 ToolPlan 决定下一步：

```
if tool_plan.intent == "kb_answer":
    go to rag_node

if tool_plan.intent == "create_ticket" and missing_fields:
    go to draft_node

if tool_plan.need_confirm:
    go to confirm_node

if tool_plan.tool in allowed_tools:
    go to execute_node
```

也就是说：ToolPlan 决定 Agent 的下一步流向。

面试表达：在我的项目里，Planner 不是直接执行动作，而是生成 ToolPlan。Graph 根据 ToolPlan 里的 intent、tool、args、need_confirm 等字段做条件分流。这样可以把模型的不确定性限制在 Planner 输出里，后续执行节点仍然是确定性的后端逻辑。

这个表达很强。

12. LangGraph 适合解决哪些面试场景？

你可以答：

```
多步骤 Agent
需要工具调用
需要状态保存
需要人工确认
需要失败恢复
需要多轮任务
需要可观测和可测试
```

比如：

```
客服工单 Agent
企业知识库助手
审批 Agent
报销 Agent
数据分析 Agent
```

运维排障 Agent
代码修复 Agent

你的项目正好是：

企业知识库 + 工单 Agent

所以 LangGraph 是非常贴合的。

13. LangGraph 不是什么？

它不是让模型更强的东西。

它也不是必须替代你的后端 service。

它更像：Agent 流程编排层。

你仍然需要：

FastAPI
权限校验
数据库事务
SQLAlchemy
审计日志
Redis 限流
RAG 检索服务
工具执行 service

面试时不要说：我用了 LangGraph，所以权限安全就解决了。

要说：LangGraph 负责流程编排，后端 service 负责真正的业务执行、安全校验和数据一致性。

这和你前面 Tool Calling 的逻辑是一致的。

14. 面试高频问答

Q1：LangGraph 是什么？

答：LangGraph 是一个面向复杂 Agent 的流程编排框架，可以把 Agent 拆成 State、Node、Edge 和 Conditional Edge。它适合有状态、多分支、多工具、可暂停、可恢复、需要人工确认的 Agent 场景。

Q2：LangChain 和 LangGraph 有什么区别？

答：LangChain 更偏上层 Agent 和 LLM 应用开发框架，封装模型、工具、Prompt、Retriever 等能力；LangGraph 更偏底层编排运行时，用图结构管理复杂 Agent 的状态、节点、分支、循环、持久化和 human-in-the-loop。简单 Agent 可以用 LangChain，复杂业务 Agent 更适合用 LangGraph。LangChain 官方也说明其 agents 构建在 LangGraph 之上，以利用 durable execution、human-in-the-loop 和 persistence 等能力。

Q3：State 是什么？

答：State 是整个图执行过程中共享的状态对象。每个节点可以读取 State，也可以写入自己的执行结果。比如用户输入、意图、ToolPlan、draft_id、pending_action、tool_result、trace_id、final_answer 都可以放在 State 里。

Q4：Node 是什么？

答：Node 是图里的一个执行步骤，可以是一次 LLM 调用、一次 RAG 检索、一次工具调用、一次权限校验、一次确认处理或一次审计记录。好的 Node 应该职责单一，方便测试和排查。

Q5：Edge 和 Conditional Edge 有什么区别？

答：Edge 是固定流转，比如 A 执行完一定到 B。Conditional Edge 是根据当前 State 动态选择下一步，比如 Planner 输出 kb_answer 就进入 RAG，输出 cancel_ticket 就进入确认节点，缺字段就进入澄清或草稿节点。

Q6：为什么 Agent 需要 Human-in-the-loop？

答：因为有些动作有副作用或风险，比如取消工单、删除数据、提交审批，不能让模型误判后直接执行。

Human-in-the-loop 可以让流程暂停，等待用户或人工确认后再继续执行。LangGraph 的 interrupt/persistence 机制就适合这种暂停和恢复场景。

Q7：你的项目为什么适合用 LangGraph 思路表达？

答：因为我的项目不是简单问答，而是企业知识库 RAG + 工单 Agent。它有 Planner、Memory、Draft、Confirm、Execution、Audit 等多个步骤，还需要根据 intent、missing_fields、need_confirm 做条件分流。取消工单这类高风险动作还需要暂停等待确认。所以它非常适合用 LangGraph 的 State、Node、Conditional Edge 和 checkpoint 思路来建模。

15. 本课你必须背的项目表达

这段直接背：我的项目可以用 LangGraph 思路建模：State 里保存 user_input、ToolPlan、draft_id、pending_action、trace_id 等信息；Planner Node 负责识别意图并生成 ToolPlan；Memory Node 负责解析“上一单”“继续刚才”等上下文引用；Conditional Edge 根据 intent、missing_fields、need_confirm 决定进入 RAG、Draft、Confirm 或 Execution；Execution Node 调用后端 service 执行业务动作；Audit Node 写入审计日志。这样可以把复杂 Agent 拆成可测试、可追踪、可恢复的流程。

第 23 课：LangGraph State / Node / Edge 项目化表达

这一课目标很明确：

你不是要变成 LangGraph API 熟练工，而是要能在面试里把你的 Agent 项目讲成一张状态图。

官方对 LangGraph Graph API 的核心定义也很清楚：**State 是应用当前快照的共享数据结构；Nodes 是执行 Agent 逻辑的函数；Edges 决定下一个执行哪个 Node，可以是固定转移，也可以是条件分支**

1. 先把你的项目抽象成一张图

你的 Policy KB Assistant 可以讲成：

```
用户输入
  ↓
Memory Node：恢复上下文
  ↓
Planner Node：识别意图，生成 ToolPlan
  ↓
Validate Node：校验 ToolPlan
  ↓
Conditional Edge：根据 intent / missing_fields / need_confirm 分流
  ├── RAG Node：制度知识库问答
  ├── Draft Node：缺字段建单草稿
  ├── Confirm Node：高风险动作确认
  ├── Execute Node：执行工单工具
  └── Clarify Node：意图不清或缺对象时追问
  ↓
Audit Node：写审计日志
  ↓
Response Node：返回用户
```

面试里可以直接这样说：我的项目可以用 LangGraph 思路建模：先通过 Memory Node 恢复“上一单”“继续刚才”等上下文，再由 Planner Node 生成结构化 ToolPlan，Validate Node 做 schema 和工具白名单校验，然后 Conditional Edge 根据 intent、missing_fields、need_confirm 进入 RAG、Draft、Confirm、Execute 或 Clarify 节点，最后通过 Audit Node 记录审计日志，并由 Response Node 返回结果。

这就是你项目的核心架构表达。

2. 你的 Agent State 应该包含哪些字段？

请求信息：trace_id、user_id、tenant_id、user_input

上下文记忆：last_ticket_id、last_draft_id

Planner：intent、ToolPlan、tool_args、missing_fields、need_confirm

RAG：retrieved_docs、citations
草稿确认：draft_id、pending_action
执行结果：tool_result、final_answer
面试表达：State 里不只放 messages，还要放业务上下文。我的项目会放用户身份、租户、trace_id、ToolPlan、draft_id、pending_action、retrieved_docs、tool_result 等字段。这样每个节点都能基于同一个状态对象工作，也方便审计和失败恢复。

3. State 设计原则：不要把所有东西塞进 Prompt

这是很多人会犯的错。
差的做法：

把用户信息、权限、历史工单、草稿、审计信息全部拼进 prompt

好的做法：

State 里保存结构化信息
需要 LLM 的时候，再按需格式化成 prompt

LangGraph 的“Thinking in LangGraph”文档也强调：设计 State 时要考虑什么属于 State，并且保持 State 原始，在需要时再格式化 prompt。
你的项目里应该这样讲：我不会把所有业务状态都塞进 prompt，而是把它们结构化存到 State 里。比如 user_id、tenant_id 用于权限校验，不一定给模型看；retrieved_docs 用于 RAG 回答；pending_action 用于恢复确认流程。这样可以减少 prompt 膨胀，也能避免把敏感信息暴露给模型。
这个表达很高级。

4. Memory Node 怎么讲？

用户会说：

“查一下上一单”
“继续刚才那个”
“帮我催一下”
“取消上一单”

这些话里没有明确 ticket_id 或 draft_id，需要从记忆里解析。
Memory Node 做的事：

读取 user_id / session_id
读取 L1 session memory
解析 last_ticket_id / last_draft_id
把 resolved_ticket_id / resolved_draft_id 写回 State

项目表达：Memory Node 主要解决上下文引用问题，比如用户说“上一单”“刚才那个草稿”。它会从 L1 session memory 读取 last_ticket_id 或 last_draft_id，并写入 State。后续 Planner 或 Executor 不直接相信自然语言引用，而是使用解析后的结构化 ID。
面试补充：如果没有找到对应对象，就进入 Clarify Node，而不是让模型猜一个 ticket_id。

5. Planner Node 怎么讲？

Planner Node 是你的 Agent 大脑，但它不能直接执行动作。
输入：

user_input
resolved_ticket_id
resolved_draft_id
history summary
可用工具列表

输出：

```
{
  "intent": "cancel_ticket",
  "tool": "cancel_ticket",
  "args": {
    "ticket_id": "T20260618001"
```

```
  },
  "need_confirm": true,
  "missing_fields": []
}
```

面试表达：Planner Node 负责理解用户意图，并生成结构化 ToolPlan。ToolPlan 里包含 intent、tool、args、missing_fields、need_confirm 等字段。Planner 只负责规划，不负责执行；这样可以把 LLM 的不确定性限制在结构化输出阶段。

这句话是核心。

6. Validate Node 怎么讲？

Validate Node 是你项目区别于普通 demo 的关键。

它负责：

- JSON / Pydantic schema 校验
- 工具名是否在白名单
- 必填参数是否齐全
- 参数类型是否合法
- 是否引用了允许的资源 ID
- 是否需要确认

面试表达：Validate Node 会对 Planner 生成的 ToolPlan 做结构化校验，包括 schema 校验、工具白名单、必填参数、参数类型和动作风险等级。它不会代替最终权限校验，但可以在进入执行节点前先挡掉明显非法或危险的计划。

注意边界：

- Validate Node：校验 ToolPlan 形式和基本合法性
- Service 层：做真实权限、状态、事务和数据一致性校验

这个边界要记住。

7. Conditional Edge 怎么根据 ToolPlan 分流？

Conditional Edge 根据 ToolPlan 里的 intent、missing_fields、need_confirm、tool_name 来分流。知识问答进入 RAG；缺字段进入 Draft；高风险动作进入 Confirm；普通工具调用进入 Execute；意图不清或校验失败进入 Clarify。

这就是你 Agent 的“可控性”。

8. RAG Node 怎么讲？

RAG Node 只处理制度知识问答，不处理工单副作用动作。

流程：

- 读取 query / tenant_id / user_id
- 执行权限过滤
- Dense + BM25 检索
- RRF 融合
- CrossEncoder rerank
- 组织证据和 citation
- 生成回答或拒答
- 写回 final_answer / citations

面试表达：RAG Node 负责制度知识库问答。它会根据用户身份和租户做 metadata / payload 过滤，再执行 Dense + BM25 混合检索、RRF 融合和 CrossEncoder 重排，最后把证据组织进 prompt，并要求模型基于证据回答和返回 citation。证据不足时进入拒答。

这段就是你 RAG 项目亮点。

9. Draft Node 怎么讲？

Draft Node 处理缺字段任务。比如用户只说“帮我建个工单”，系统不会让模型编造标题或描述，而是创建 ticket_draft，记录已经抽取到的字段和 missing_fields，然后追问用户补充。用户后续说“就是 VPN 连不上”，Memory Node 可以恢复 last_draft_id，再继续补全草稿。

这比简单说“缺信息追问”强很多。

10. Confirm Node 怎么讲?

Confirm Node 处理高风险动作。

例如:

- 取消工单
- 删除草稿
- 提交审批
- 发送外部通知

Confirm Node 做:

- 创建 pending_action
- 生成确认问题
- 暂停流程
- 等待用户确认

LangGraph 的 interrupts 支持在图节点中暂停执行并等待外部输入; 触发 interrupt 时, 图状态会通过 persistence 保存, 之后可以恢复执行。

你的项目表达: Confirm Node 用来处理高风险动作, 比如取消工单。它不会直接执行, 而是创建 pending_action, 并返回确认问题。用户确认后, 系统从 pending_action 恢复原始 ToolPlan, 再进入 Execute Node。这个思想和 LangGraph 的 human-in-the-loop / interrupt 很像。

如果面试官问: **你项目没用 LangGraph interrupt 怎么办?**

你可以答: 我用 pending_actions 表实现了类似的暂停和恢复机制, 本质上也是把待执行动作持久化, 等待用户确认后再继续执行。

11. Execute Node 怎么讲?

Execute Node 是统一工具执行入口, 不建议每个工具都拆成一个 Node。

它做:

- 根据 tool_name 找到后端 service
- 再次做权限校验
- 再次做资源状态校验
- 开启数据库事务
- 执行 create_ticket / cancel_ticket / add_comment / followup
- commit / rollback
- 返回 tool_result

面试表达: Execute Node 不直接写 SQL, 而是调用后端 service。service 内部负责权限校验、资源状态校验、幂等控制、数据库事务和异常处理。这样 LangGraph 只负责编排, 真正的数据一致性和安全边界仍然在后端 service 层。

这句非常重要:

Graph 管流程, Service 管执行。

12. Audit Node 怎么讲?

Audit Node 是你项目的企业级亮点。

它记录:

- trace_id
- user_id
- tenant_id
- intent
- tool_name
- tool_args 摘要
- validation result
- permission result
- tool_result
- latency
- error

面试表达：Audit Node 负责把关键节点的决策和执行结果写入审计日志，比如 Planner 输出了什么 ToolPlan、是否触发确认、最终执行了哪个工具、结果是什么。所有日志通过 trace_id 串起来，方便排查问题、复盘模型误判和做安全追踪。
这能把你前面学的 trace_id、middleware、audit_logs 都串起来。

13. Response Node 怎么讲？

Response Node 负责组织最终返回。
它根据 State 判断：

```
如果 final_answer 存在 -> 返回 RAG 答案
如果 missing_fields 存在 -> 返回追问
如果 pending_action 存在 -> 返回确认问题
如果 tool_result 存在 -> 返回执行结果
如果 validation_errors 存在 -> 返回澄清/错误说明
```

面试表达：Response Node 不做复杂业务，只负责根据 State 组织用户可理解的回复。这样业务执行和用户表达可以解耦，方便后续支持不同前端或多语言输出。

14. 你项目的完整 LangGraph 讲法

你可以直接背这段：我的 Policy KB Assistant 可以用 LangGraph 思路建模。State 里保存 user_input、user_id、tenant_id、trace_id、ToolPlan、draft_id、pending_action、retrieved_docs、tool_result 等信息。Memory Node 先解析“上一单”“继续刚才”等上下文引用；Planner Node 生成结构化 ToolPlan；Validate Node 做 schema、工具白名单和必填参数校验；Conditional Edge 根据 intent、missing_fields、need_confirm 决定进入 RAG、Draft、Confirm、Execute 或 Clarify。RAG Node 负责制度问答，Draft Node 负责缺字段工单草稿，Confirm Node 负责高风险动作的 pending_action，Execute Node 调用后端 service 执行业务动作，Audit Node 用 trace_id 记录全链路。这样可以把复杂 Agent 拆成可测试、可追踪、可暂停和可恢复的流程。
15.面试官可能追问：Node 怎么测试？
每个 Node 都可以单独测试。比如 Planner Node 可以用 Planner JSON 评测测试 intent、tool、required args；Validate Node 可以构造非法工具名、缺参数、越权 ID 测试拦截；Draft Node 测缺字段追问；Confirm Node 测 pending_action；Execute Node 测权限、状态和事务；最后再做 Agent E2E 测试，覆盖完整用户场景。
这里可以接你已有的评测数据：
我项目里已经做了 Planner JSON 评测和 Agent E2E 评测，不只是看模型回答，而是看结构化计划和业务结果是否正确。

第 24 课：Checkpoint / Human-in-the-loop / 多轮恢复

Checkpoint 解决“状态怎么保存”；Human-in-the-loop 解决“危险动作怎么暂停等人确认”；多轮恢复解决“用户下一句话怎么接上刚才的任务”。

1. 为什么需要 Checkpoint？

普通接口请求是一次性的：

```
用户请求
  -> FastAPI
  -> service 执行
  -> 返回结果
```

请求结束后，很多临时状态就没了。
但 Agent 任务经常不是一次完成的：

```
用户：“帮我建个工单”
系统：“请补充问题描述”
用户：“VPN 连不上”
系统：“已创建工单”
```

或者：

```
用户：“取消上一单”
系统：“确认取消 T001 吗？”
用户：“确认”
```

系统：“已取消”

这种任务需要保存中间状态。

所以 Checkpoint 的作用就是：**把 Agent 当前 State 保存下来，让流程可以暂停、恢复、调试和容错。**

LangGraph 的 checkpointer 会在执行步骤中保存 graph state 的快照，并按 thread 组织；这些 checkpoint 可用于 human-in-the-loop、time travel debugging、fault-tolerant execution 和 conversational memory。

2. Checkpoint 和普通 Memory 有什么区别？

这两个容易混。

Memory

Memory 偏“长期或会话上下文”：

用户最近查过哪张工单
用户常用部门
最近草稿 ID
用户偏好

你项目里对应：

L1 Session Memory
L3 User Memory
last_ticket_id
last_draft_id

Checkpoint

Checkpoint 偏“流程执行快照”：

当前跑到哪个 node
当前 State 是什么
下一步要执行哪个 node
是否暂停等待确认
工具执行前的状态是什么

你项目里对应：

pending_action
ticket_draft
trace_id 关联的执行状态
一次 Agent workflow 的中间状态

面试表达：Memory 更像上下文记忆，Checkpoint 更像流程快照。Memory 帮 Agent 知道“用户之前说过什么、最近操作过什么”；Checkpoint 帮 Agent 知道“当前任务执行到哪一步，后面要怎么继续”。

3.为什么 Human-in-the-loop 必须依赖状态保存？

因为人工确认不是同一个瞬间完成的。

比如：

第 1 次请求：
用户：“取消上一单”

系统：
1. 解析 last_ticket_id = T001
2. 生成 ToolPlan = cancel_ticket(T001)
3. 判断 need_confirm = true
4. 保存 pending_action
5. 返回：“确认取消 T001 吗？”

用户过几秒或几分钟后才回复：

第 2 次请求：
用户：“确认”

这时候系统必须知道：

用户确认的是哪个动作？

对应哪个 ticket_id?
原始工具是什么?
参数是什么?
确认前状态是什么?
这个 pending_action 是否过期?

如果没有状态保存，系统只能重新猜，很危险。

LangGraph 的 interrupts 机制就是为这种模式设计的：在需要暂停的节点调用 interrupt，图执行会暂停并等待外部输入；使用 interrupt 需要 checkpointer 保存图状态，并且需要 thread ID 让运行时知道从哪个状态恢复。

面试表达：Human-in-the-loop 本质上是“暂停—等待人输入—恢复执行”。如果没有 checkpoint 或 pending_action 这种状态保存机制，系统在用户确认时就不知道要恢复哪个动作，也无法保证安全。

4. 你的项目里哪些地方需要多轮恢复?

第一类：缺字段建单

用户：“帮我建个工单”
系统：“请补充问题描述”
用户：“VPN 连不上”
系统：“已创建工单”

对应你项目：

ticket_draft
last_draft_id
continue_ticket_draft
missing_fields

第二类：高风险动作确认

用户：“取消上一单”
系统：“确认取消 T001 吗？”
用户：“确认”
系统：“已取消”

对应你项目：

pending_action
need_confirm
confirm_token / pending_action_id
execute_pending_action

第三类：上下文引用

用户：“查一下上一单”
用户：“帮我催一下”

对应：

last_ticket_id
resolved_ticket_id
L1 session memory

第四类：失败恢复

工具执行失败
数据库事务回滚
用户稍后重试
trace_id 排查

对应：

trace_id
audit_log
tool_result / error
workflow state

面试表达：我的项目里多轮恢复主要出现在缺字段建单、高风险确认、上下文引用和失败恢复。比如 ticket_draft 负责保存未完成的建单任务，pending_action 负责保存待确认动作，last_ticket_id 和 last_draft_id 负责恢复“上一单”“继续刚才”等上下文。

6. pending_action 是什么？

pending_action 是“待用户确认的高风险动作”。

用户确认后，系统通过 pending_action_id 恢复原始动作，再进入 Execute Node。

关键点：用户确认后，也不是直接执行。还要重新做权限校验、租户校验、工单状态校验和审计记录。

面试表达：pending_action 用来保存高风险动作的原始 ToolPlan。它支持流程暂停和恢复，用户确认后再恢复 pending_action 执行。确认只是增加人工确认环节，不会绕过后端权限校验和状态校验。

7. confirm token 该怎么理解？

你上一课提到了 confirm token，这个可以用，但要讲清楚。

confirm token 不是安全的全部，它只是一个“确认上下文标识”。

更推荐你说：pending_action_id / confirmation token

它的作用是：标识用户确认的是哪个 pending_action

避免把“确认”误绑定到错误动作

防止前端或多轮对话丢失上下文

但真正安全靠：

- pending_action 绑定 user_id / tenant_id
- pending_action 过期时间
- pending_action 状态校验
- 执行前权限校验
- 执行前资源状态校验
- audit_log

面试表达：confirmation token 只是用来定位待确认动作，不能代替权限校验。真正执行前仍然要验证 pending_action 属于当前用户和租户，状态是 waiting_confirm，没有过期，并且目标资源仍然允许执行该操作。

8. “确认”为什么不能直接用上一条消息？

错误做法：

- 系统：“确认取消 T001 吗？”
- 用户：“确认”
- 系统直接取消 last_ticket_id

问题是：

- 用户可能开了多个窗口
- 期间 last_ticket_id 可能变化
- pending_action 可能已过期
- 工单状态可能已被别人处理
- 用户可能没有权限了

正确做法：

- 用户：“确认”
- ↓
- 根据 pending_action_id 找到待确认动作
- ↓
- 校验 user_id / tenant_id
- ↓
- 校验 pending_action 状态和过期时间
- ↓
- 校验 ticket 权限和状态
- ↓
- 执行取消

↓

写 audit_log

面试表达：我不会只根据“上一条消息”来执行确认，而是通过 pending_action_id 恢复持久化的待确认动作。这样可以避免多窗口、上下文漂移、状态变化导致误操作。

这个点很高级。

11. Checkpoint / pending_action / ticket_draft 的关系

概念	解决什么问题	你项目对应
Checkpoint	保存图执行快照	workflow state / trace state
Memory	保存上下文引用	last_ticket_id / last_draft_id
ticket_draft	保存未完成建单任务	缺字段草稿
pending_action	保存待确认高风险动作	取消工单确认
audit_log	保存已发生事实	trace_id 全链路审计

面试表达：对话历史只能辅助模型理解语义，不能作为可靠的业务状态。涉及创建、取消、审批这类动作时，必须把状态结构化保存到数据库或 Redis，并通过后端 service 校验后执行。

13. 失败恢复怎么讲？

比如工具执行中断：

Planner 已经生成 ToolPlan

Confirm 已经通过

Execute Node 调用 service 时超时

这时候需要知道：

执行到哪一步？

有没有 commit？

能不能重试？

是否会重复创建？

所以要配合：

trace_id

idempotency_key

audit_log

pending_action status

ticket_draft status

数据库事务

面试表达：对失败恢复，我会通过 trace_id、audit_log、状态字段和幂等键来判断任务执行到哪一步。比如创建工单时使用 Idempotency-Key 防止用户重复提交，数据库操作用事务保证 commit/rollback，一旦失败可以通过 trace_id 定位问题。

这就把你之前学的 FastAPI 幂等、数据库事务、trace_id 都串起来了。

14. 和 LangGraph 官方能力怎么对应？

LangGraph 里可以通过 checkpointer 保存图状态，通过 interrupt 暂停等待人工输入，再根据 thread_id 恢复执行。我的项目虽然不一定完全依赖 LangGraph 内置机制，但用 ticket_draft、pending_action、last_ticket_id、trace_id 等业务状态实现了类似的暂停、恢复和审计能力。官方文档也说明，durable execution 需要通过 checkpointer 保存 workflow progress，并指定 thread identifier 来跟踪某个工作流实例。

15. 面试高频问答

Q1：Checkpoint 是什么？

答：Checkpoint 是 Agent 执行过程中的状态快照，用来保存当前 State、执行位置和中间结果。它让复杂 Agent 可以暂停、恢复、调试和失败重试。

Q2：Checkpoint 和 Memory 有什么区别？

答：Memory 主要保存上下文信息，比如用户最近操作过哪个工单；Checkpoint 保存的是流程执行快照，比如当前跑到哪个节点、下一步要执行什么、是否在等待确认。Memory 解决“记住什么”，Checkpoint 解决“执行到哪了”。

Q3：Human-in-the-loop 是什么？

答：Human-in-the-loop 是在关键节点让人介入，比如取消工单、删除记录、提交审批等高风险动作。系统先暂停流程，保存 pending_action，等用户确认后再恢复执行。

Q4：为什么用户说“确认”后不能直接执行？

答：因为“确认”本身没有完整业务上下文。系统必须通过 pending_action_id 恢复原始动作，并校验 pending_action 是否属于当前用户和租户、是否过期、是否已执行；然后还要校验目标资源权限和状态，最后才能执行并写审计日志。

Q5：你的 ticket_draft 解决什么问题？

答：ticket_draft 解决缺字段建单的多轮补全问题。字段不完整时系统不让模型编造，而是保存已抽取字段和 missing_fields，追问用户补充。用户后续补充时通过 last_draft_id 恢复草稿，字段齐全后再创建工单。

Q6：pending_action 解决什么问题？

答：pending_action 解决高风险动作确认问题。比如取消工单时，系统先保存原始 ToolPlan 和参数，返回确认问题；用户确认后恢复 pending_action，重新做权限、租户、状态和过期校验，再执行工具。

Q7：为什么不只靠 LLM 对话历史恢复上下文？

答：对话历史是非结构化文本，只适合辅助语义理解，不能作为可靠业务状态。涉及工单创建、取消、审批这类动作时，必须把 draft、pending_action、状态、过期时间和权限关联结构化保存，并由后端 service 校验后执行。

Q8：你的项目怎么体现 LangGraph 的 checkpoint / human-in-the-loop 思想？

答：我的项目里，ticket_draft 类似缺字段任务的状态保存，pending_action 类似高风险动作的暂停点，last_ticket_id 和 last_draft_id 用于上下文恢复，trace_id 和 audit_log 用于执行追踪。它们共同实现了类似 LangGraph checkpoint、interrupt 和 persistence 的能力，让 Agent 可以多轮补全、暂停确认、恢复执行和审计复盘。

16. 你必须背的项目表达

我的项目里，多轮恢复主要通过 ticket_draft、pending_action 和 session memory 实现。缺字段建单时，系统不会让模型编造参数，而是创建 ticket_draft，保存已抽取字段和 missing_fields，后续用户补充时通过 last_draft_id 恢复草稿。取消工单这类高风险动作会进入 Confirm Node，系统保存 pending_action，并返回确认问题；用户确认后根据 pending_action_id 恢复原始 ToolPlan，重新做用户、租户、过期、权限和资源状态校验，再由后端 service 执行。整个过程用 trace_id 和 audit_log 串起来，方便排查和审计。这本质上对应 LangGraph 的 checkpoint、interrupt 和 human-in-the-loop 思想。