AI编程与智能体开发

林子雨　副教授　厦门大学
E-mail: ziyulin@xmu.edu.cn

2026年9月

主讲教师

- 2013年度厦门大学奖教金获得者
- 2017年度厦门大学奖教金获得者
- 2020年度厦门大学奖教金获得者
- 2023年度厦门大学奖教金获得者
- 2026年度厦门大学奖教金获得者
- 2018年国家精品在线开放课程
- 2020年国家级线上一流本科课程
- 2023年国家级线上一流本科课程
- 2020年福建省线上一流本科课程
- 2021年福建省线上线下混合一流本科课程
- 2018年福建省高等教育教学成果奖二等奖
- 2022年福建省高等教育教学成果奖特等奖
- 2024年福建省高等教育教学成果奖特等奖
- 2023年教育部国家智慧教育公共服务平台应用典型案例

配套教材

教材目录

第1章 大模型
第2章 AI编程概述
第3章 AI编程环境搭建
第4章 基于提示词的AI编程
第5章 基于规范的AI编程
第6章 流程驱动的AI编程
第7章 智能体概述
第8章 智能体开发框架LangChain
第9章 智能体开发框架LangGraph

教材特色

- 教材内容来自互联网大厂真实实践
- 以实践教学为依托
- 理论实践贯通
- 覆盖主流技术

《AI编程与智能体开发》

林子雨 编著

人民邮电出版社 预计2026年11月上市销售

教材官网提供详细信息和样书申请

官网： https://dblab.xmu.edu.cn/post/ai-programming-agent/

扫码访问教材官网
了解详情、获取资源、申请样书

第9章
智能体开发框架
LangGraph

目录

Part 01

LangGraph概述

9.1 LangGraph概述

从链式调用说起

LangGraph简介

LangGraph与传统链式编排的对比分析

LangGraph的典型应用场景

9.1.1 从链式调用说起

在大模型应用发展的早期，许多系统主要采用由提示词、大模型调用与后处理组成的线性链式结构。对于问答、摘要、分类等单步任务而言，这种结构开发成本低、理解门槛小，往往已经足够使用。

但是，当应用从单轮生成走向多步决策以后，线性链就会暴露出明显局限。例如，一个真实的智能体往往既要根据输入在多个路径之间进行选择，又要在需要时调用外部工具、保存中间状态、允许人工审核、在失败后继续恢复执行，甚至还要让多个子流程并行推进。此时，问题的核心不再只是如何生成下一步输出，而是系统当前处于什么状态、下一步应该去哪里，以及中断后如何继续。

9.1.1 从链式调用说起

从工程视角看，智能体应用的难点主要体现在以下几个方面：

第一，状态往往是跨轮次、跨节点持续存在的，不能只依赖函数局部变量。

第二，控制流不再是单一路径，而是分支、循环与并行结构的组合。

第三，执行时间可能很长，中途可能出现超时、失败或人工接管。

第四，开发者需要在运行时看到每一步发生了什么，并能够定位故障原因。

因此，智能体开发逐渐从链式拼接转向图式编排。图的价值并不在于把箭头画得更复杂，而在于它能够把状态与路由显式化，使开发者可以像设计业务流程那样设计智能体流程。LangGraph 正是在这一背景下出现的。

9.1.2 LangGraph 简介

1. LangGraph 属于 LangChain 生态低层编排框架与运行时，核心解决长时运行、状态持久化、人工介入、流式反馈、故障恢复等智能体工程问题；官方将其定位为搭建、管理、部署长生命周期、有状态智能体的基础设施，并非单纯提示词封装工具。

2. LangGraph 区别于各类高层智能体框架，不会预设提示模板、角色设定、固定智能体结构；仅要求开发者明确定义状态（State）、节点（Node）、边（Edge），节点支持大模型调用、普通 Python 代码、数据库查询、外部 API 访问多种形式。

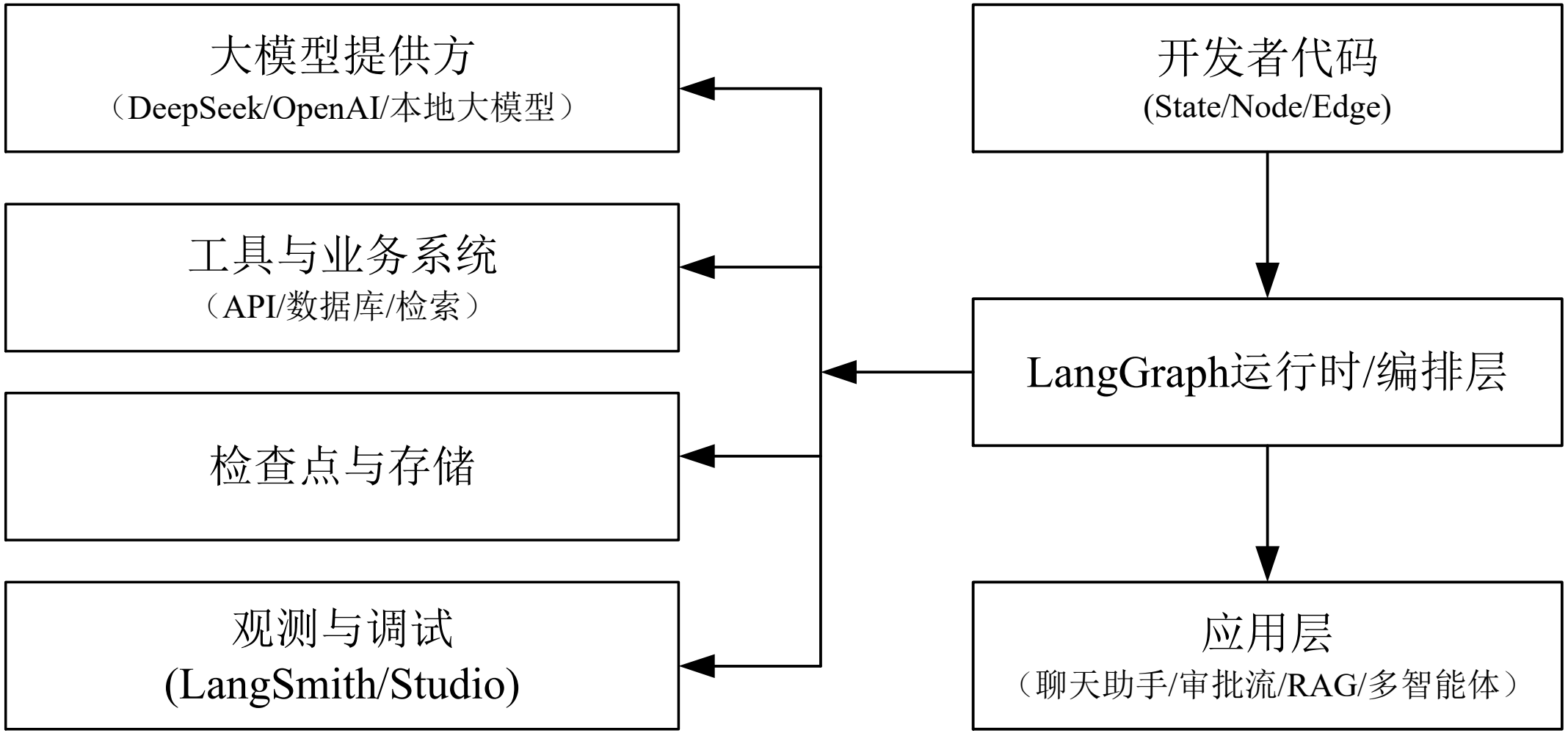
3. LangGraph 核心侧重点为可靠组织智能体执行流程，不限制开发者选用的大模型类型与提示词策略。

4. 依据官方文档，LangChain 内置 create_agent 高层接口底层依托 LangGraph 运行。

5. 开发者使用 LangChain 相关开发存在两种路径：可借助高层接口快速开发，有精细化控制需求时可下沉至 LangGraph 图级编排开发。

9.1.2 LangGraph 简介

为了更直观地说明这一定位，下图给出了 LangGraph 与周边组件的关系，它并不直接替代大模型、工具或存储，而是把这些能力组织在统一的编排框架之下。

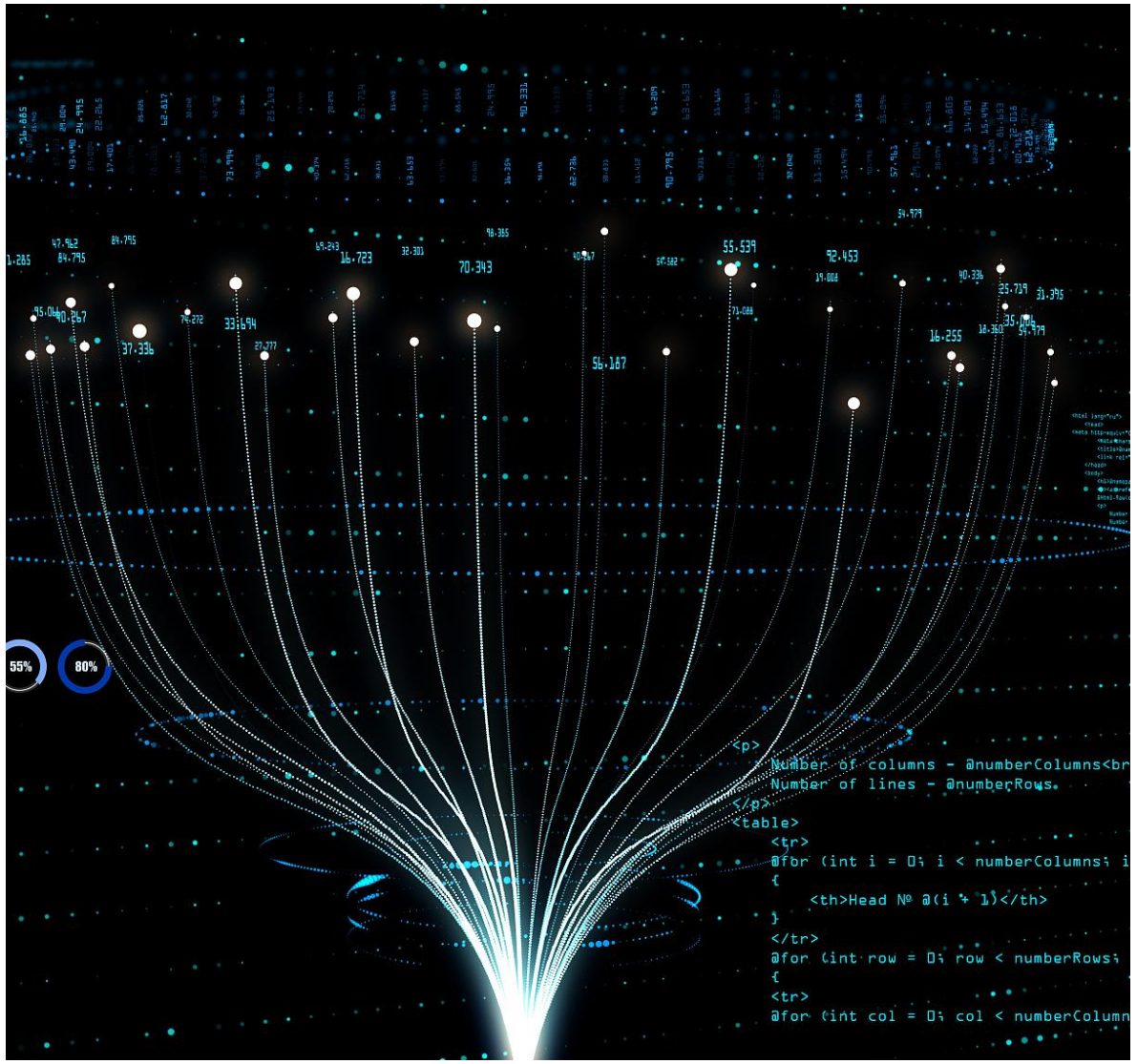


9.1.3 LangGraph 与传统链式编排的对比分析

LangGraph 与传统链式编排并不是先进替代落后的关系，而是分别面向不同复杂度的问题空间。

对于路径固定、状态短暂、失败代价较低的任务，简单链式结构依然是成本最低的方案。

只有当应用需要长期状态、复杂控制流与可恢复执行时，LangGraph 的价值才会真正显现。



9.1.3 LangGraph 与传统链式编排的对比分析

LangGraph 与传统链式编排的区别主要体现在以下几个方面：

控制流

线性链通常按既定顺序逐步执行，而 LangGraph 可以显式表示顺序、条件分支、循环、并行与子图嵌套，因此更适合真实业务流程。

状态管理

普通链式结构常依赖函数参数和临时变量传递上下文；LangGraph 则把共享状态作为核心对象，使状态的读取、更新与持久化成为一等公民。

中断恢复

链式调用一旦中途失败，通常只能从头重跑；LangGraph 可以借助检查点存储器在超步边界保存检查点，使系统在失败后从最近一次成功状态继续执行。

人机协同

在线性链中加入人工审核往往意味着在框架外手工打断流程；LangGraph 通过 interrupt() 与 Command(resume=...) 让人工成为图内部的正式节点。

观测调试

简单脚本通常只能依赖日志排查问题；LangGraph 可以结合 LangSmith /Studio展示节点执行轨迹、状态变化和中断位置，更适合复杂系统运维。

工程成本

LangGraph 的学习和建模成本高于普通链式结构，因此，不适合脱离实际需求而强行引入图结构。只有当流程复杂度确实上升时，额外的结构化成本才会转化为工程收益。

9.1.3 LangGraph 与传统链式编排的对比分析

为了把这些差异放在同一组维度下观察，下表对 LangGraph 与传统链式编排作了集中比较，重点落在控制流、状态管理、恢复能力和人工协同几个方面。

对比维度	简单链式编排	LangGraph
控制流	以顺序执行为主，少量 if/else	天然支持分支、循环、并行与子图
状态管理	多依赖局部变量或手工传参	以共享状态为核心，支持合并
恢复能力	失败后通常从头执行	可借助 检查点从最近状态继续
人工审核	常在框架外手工处理	interrupt() 可把人工审核纳入图执行
适用问题	线性、短时、单轮任务	长时运行、有状态、复杂路由任务

9.1.4 LangGraph的典型应用场景

LangGraph 适合构建具有状态管理、循环迭代、工具调用、持久执行与人工干预能力的智能体工作流。适合采用 LangGraph 的任务一般符合以下特征：

特征 1

任务包含多个处理步骤，无法依赖单次模型调用完成。

特征 2

需要保存中间状态、对话记忆，或支持中断后恢复执行。

特征 3

需要通过循环重试、质量评估或自我纠错持续改进输出。

特征 4

需要多个智能体或多种外部工具协同完成复杂目标。

特征 5

涉及发布、支付、删除、合规等需要人工审核的关键动作。

特征 6

需要以流式方式展示执行过程，并便于调试、追踪与部署。

9.1.4 LangGraph的典型应用场景

下表给出了 LangGraph的典型应用场景。

应用场景	典型任务	LangGraph 的适配点
智能客服与业务受理	问题解答、订单查询、退款申请、转人工处理	维护多轮状态；在退款等高风险操作前加入人工审批
检索增强生成（RAG）问答	企业知识库、论文或法规问答	组织查询改写、检索、相关性判断、补充检索与生成闭环
深度研究与报告生成	市场调研、竞品分析、文献综述、专题报告	拆分研究节点，汇总证据，并支持迭代审校
代码开发智能体	代码生成、执行测试、错误修复、代码审查	适配“分析—修改—测试—修复”的循环任务
数据分析智能体	SQL 查询、指标解释、可视化与经营分析	连接数据工具，并在敏感查询或操作前审核
多智能体协作系统	规划者、检索者、写作者、评审者协作	支持任务分派、并行执行、集中汇总与角色切换
人机协同的决策支持	合同审查、财务审核、内容发布审批	可暂停流程，等待人工确认或反馈后继续执行
内容生成与质量优化	翻译、摘要、营销文案、政策材料撰写	实现生成、评估、反馈与重新生成的优化循环
自动化办公与流程执行	邮件处理、工单流转、表单审批、日程安排	处理条件分支、工具调用、异常恢复与确认节点
事件响应与运维辅助	告警分析、故障排查、处置建议与复盘	适合保存执行状态并编排多工具协作
个性化长期助手	学习助手、个人助理、客户经理助手	持久化记忆可保存偏好、历史交互与未完成事项
合规与风控工作流	内容审核、风险判断、审计材料核查	组合规则判断、模型评估与人工复核降低风险

Part 02

LangGraph核心概念

9.2 LangGraph核心概念

状态

节点

边

9.2.1 状态

状态是整张图的共享数据载体。先定义状态，才能明确每个节点读取什么、写回什么，以及流程结束后能得到哪些结果。

为什么不直接用全局变量呢？你可能想到，让每个函数直接读写一个全局字典不也能传递数据吗？问题在于：全局变量没有类型声明，不知道哪些字段存在；多个节点并发运行时会产生竞争；最重要的是，LangGraph 的检查点（Checkpointer）机制无法追踪它——你失去了状态快照、断点恢复、会话隔离等所有能力。状态是被 LangGraph“托管”的数据结构，框架负责在节点间传递、合并、持久化它。

在 LangGraph 中，状态是贯穿整个工作流的共享数据结构。每当工作流从一个节点流转到另一个节点时，所携带的信息都保存在状态中。定义状态时，使用 Python 标准库中的 TypedDict 以确保类型安全。状态的更新是增量式的：节点返回部分字段时，LangGraph 将其与当前状态合并，生成完整的新状态。

9.2.2 节点

有了状态之后，就需要定义谁来处理状态。节点负责完成一个清晰的业务动作，并把局部结果写回状态。

为什么不直接调用函数呢？普通函数调用是你主动写 result = classify(input)，调用顺序硬编码在代码里。而 LangGraph 的节点是“注册”到图中的——框架知道节点叫什么名字、上游是谁、下游是谁。这带来三个好处：路由函数可以在运行时动态决定调用哪个节点；图结构可以被可视化；每个节点的输入输出都被框架记录，便于调试和回溯。

节点是 LangGraph 中进行实际工作的执行单元。每个节点是一个 Python 函数，接收当前完整状态作为输入，返回需要更新的状态片段（字典）。节点返回值只需包含待更新的字段，LangGraph 会自动与当前状态合并。节点函数既可以是同步的，也可以是异步的——涉及外部 API 调用或数据库访问时，推荐使用异步函数。

9.2.3 边

节点定义了“做什么”，边定义了“下一步去哪里”。理解边之后，图结构才真正从一组函数变成可运行流程。

1. 为什么不在节点内部写 if-else 决定下一步呢？把路由判断塞进节点函数，节点就同时承担”做事”和“决定去哪”两个职责，既难测试，也难修改——改一条路由规则就要改节点代码。边的设计把这两件事分开：节点只负责更新状态，边（路由函数）只负责读状态决定走向。分离后，每部分都可以独立测试，新增一条分支也只需添加节点和边，不动已有逻辑。

2. 边定义节点之间的连接关系，决定工作流从当前节点下一步流向何处。LangGraph 支持三种类型的边。普通边表示固定流转关系，工作流到达某节点后无条件流向下一指定节点。条件边（Conditional Edge）根据当前状态动态决定下一步，适用于需要分支判断的场景。入口边指定整个工作流的起始节点，使用特殊标识 __start__；工作流终点使用 END 标识。

3. 边定义在节点添加完成之后进行：通常先add_node 注册所有节点，再用add_edge与 add_conditional_edges 建立连接关系。

Part 03

LangGraph快速入门

9.3 LangGraph快速入门

LangGraph 常用 Python API 及应用说明

环境准备

第一个 LangGraph 应用

9.3.1 LangGraph 常用 Python API 及应用说明

为了完成后面的实例，需要先熟悉若干常用 Python API。这里不追求罗列全部接口，而是选取最常见、最能体现运行时思想的部分。下表列出了最常用的一组 Python API，基本覆盖了建图、执行、持久化和中断恢复几个关键环节。

名称	说明	常见写法
StateGraph	定义一张以某个状态模式为输入输出的图	builder = StateGraph(State)
START / END	虚拟入口与出口节点	builder.add_edge(START, "node_a")
add_node	向图中注册节点函数	builder.add_node("draft", draft_answer)
add_edge	添加固定流向的边	builder.add_edge("a", "b")
add_conditional_edges	根据状态动态决定后继节点	builder.add_conditional_edges("draft", route_fn)
compile	把图定义编译为可执行对象，并可注入 checkpointer	graph = builder.compile(checkpointer=cp)
invoke	执行一次图调用，或以 Command(resume=...) 恢复	graph.invoke(inputs, config=config)
stream	流式获取 updates / values / messages	graph.stream(inputs, config=config)
InMemorySaver	内存型 checkpointer，适合课堂和本地测试	cp = InMemorySaver()
interrupt	在节点内部暂停执行并等待外部输入	edited = interrupt(payload)
Command	恢复中断，或在节点内进行高级跳转	Command(resume={...})

从调用顺序看，一个最小 LangGraph 程序通常先定义 State，再用 StateGraph 注册节点和边，然后通过 compile 生成可执行图，最后用 invoke 或 stream 触发执行；当流程需要人工介入时，再通过 interrupt() 暂停，并用 Command(resume=...) 在原线程上恢复。这样一条调用链，正好把图定义、运行时和恢复机制串联起来。

9.3.2 环境准备

为了保证依赖管理清晰，实际开发中通常建议使用独立的Python虚拟环境，以免与其他实验冲突。推荐命令如下：

```
python -m venv langgraph-env  # 在当前目录下创建一个名字叫langgraph-env的独立 Python 环境，用来隔离项目依赖，不污染系统全局 Python
langgraph-env\Scripts\activate
pip install -U langgraph
```

9.3.3 第一个LangGraph应用

状态、节点和边，单独看都不复杂，关键是理解它们如何配合。下面通过协同视角把三者串起来，为快速入门代码做准备。这里通过一个最简示例展示状态、节点和边三者的协作方式。

状态在整个工作流中流转，节点读取并更新状态，边则决定状态应流向哪个节点。三者各司其职、紧密配合。以下代码演示完整的 LangGraph构建过程：

```
# state_node_edge.py
from typing import TypedDict
from langgraph.graph import StateGraph, END

# 1. 定义 State：三个字段，各司其职
class State(TypedDict):
    messages: list[str]   # 用户输入的消息列表
    intent: str | None    # 由第一个节点识别出的意图
    response: str | None  # 由第二个节点生成的回复
```

9.3.3 第一个LangGraph应用

```
# 2. 定义 Node：每个节点只做一件事，返回需要更新的字段
def classify_intent(state: State) -> dict:
    """读取最后一条消息，用关键词匹配识别意图"""
    text = state["messages"][-1] if state["messages"] else ""
    if any(kw in text for kw in ["计算", "多少", "分数", "成绩"]):
        intent = "calculation"
    elif any(kw in text for kw in ["规定", "政策", "要求"]):
        intent = "policy"
    else:
        intent = "knowledge"
    return {"intent": intent}  # 只更新 intent 字段，其余字段保持不变
```

9.3.3 第一个LangGraph应用

```
def generate_response(state: State) -> dict:
    """读取识别出的意图，生成对应回复"""
    intent = state.get("intent", "knowledge")
    question = state["messages"][-1] if state["messages"] else ""
    if intent == "calculation":
        response = f"这是一道计算题，让我帮你算一下：{question}"
    elif intent == "policy":
        response = f"关于规定的问题，我来查一下：{question}"
    else:
        response = f"关于您的问题「{question}」，这是我的解答..."
    return {"response": response}
```

9.3.3 第一个LangGraph应用

```
# 3. 构建工作流：先注册节点，再连接边
workflow = StateGraph(State)
workflow.add_node("classify", classify_intent)
workflow.add_node("respond", generate_response)

# 4. 定义边：固定顺序，classify → respond → END
workflow.add_edge("__start__", "classify")
workflow.add_edge("classify", "respond")
workflow.add_edge("respond", END)

# 5. 编译为可执行应用
app = workflow.compile()

# 6. 运行测试
result = app.invoke({"messages": ["期末考试成绩怎么计算？"], "intent": None, "response": None})
print(f"意图: {result['intent']}")    # 输出: 意图: calculation
print(f"回复: {result['response']}")  # 输出: 回复: 这是一道计算题，让我帮你算一下：期末考试成绩怎么计算?
```

Part 04

LangGraph的条件边与条件节点

9.4 LangGraph的条件边与条件节点

- 条件边的工作原理
- 多条件分支
- LLM驱动的路由
- 默认路由与健壮性
- 条件边的特殊用法：循环

9.4.1 条件边的工作原理

条件边（Conditional Edge）是 LangGraph 中处理分支逻辑的核心机制。其核心是路由函数，该函数接收当前状态，返回一个字符串标识符，再结合 add_conditional_edges 传入的映射表确定实际跳转的节点。

路由函数必须是纯函数（Pure Function），即只根据输入状态决定输出，不产生副作用。下面是智学助手最基础的两路分支的例子：

```
# two_edges.py
from typing import TypedDict
from langgraph.graph import StateGraph, END

class AssistantState(TypedDict):
    question: str
    intent: str | None
    answer: str | None

# 节点：处理知识问答
def knowledge_node(state: AssistantState) -> dict:
    return {"answer": f"关于「{state['question']}」，这是我的解答..."}
```

9.4.1 条件边的工作原理

```
# 节点：处理计算问题
def calc_node(state: AssistantState) -> dict:
    return {"answer": f"让我帮您计算：{state['question']}"}

# 识别意图的节点
def classify_node(state: AssistantState) -> dict:
    q = state["question"]
    if any(kw in q for kw in ["计算", "多少", "分数"]):
        return {"intent": "calculation"}
    return {"intent": "knowledge"}

# 路由函数：根据意图决定走哪个节点
def route_by_intent(state: AssistantState) -> str:
    intent = state.get("intent", "knowledge")
    if intent == "calculation":
        return "calc_node"
    return "knowledge_node"
```

9.4.1 条件边的工作原理

```
workflow = StateGraph(AssistantState)
workflow.add_node("classify", classify_node)
workflow.add_node("knowledge", knowledge_node)
workflow.add_node("calc", calc_node)

workflow.add_edge("__start__", "classify")
workflow.add_conditional_edges(
    "classify",          # 从 classify 节点出发
    route_by_intent,     # 路由函数
    {
        "knowledge_node": "knowledge",   # 返回 "knowledge_node" → 执行 knowledge 节点
        "calc_node": "calc",             # 返回 "calc_node" → 执行 calc 节点
    }
)
```

9.4.1 条件边的工作原理

```
workflow.add_edge("knowledge", END)
workflow.add_edge("calc", END)

app = workflow.compile()

result = app.invoke({"question": "期末成绩怎么计算？ ", "intent": None, "answer": None})
print(result["answer"])  # 输出: 让我帮您计算：期末成绩怎么计算?
```

9.4.2 多条件分支

理解两路分支后，就可以扩展到多路分支。多条件路由更接近智学助手中的真实需求——不同意图进入不同处理节点。

一个路由函数可以同时考察多个维度。在智学助手里，除了区分知识和计算，还要处理投诉类问题——投诉需要先安抚情绪，而不是直接回答。

在上一节代码基础上，这里新增第三条分支：

```
# three_edges.py
from typing import TypedDict
from langgraph.graph import StateGraph, END

class AssistantState(TypedDict):
    question: str
    intent: str | None
    sentiment: str | None   # 新增：用户情绪
    answer: str | None

# 新增节点：情绪安抚
def empathy_node(state: AssistantState) -> dict:
    return {"answer": "非常抱歉给您带来了不好的体验。我已记录您的反馈，会尽快跟进处理。"}
```

9.4.2 多条件分支

```
# 节点：处理知识问答
def knowledge_node(state: AssistantState) -> dict:
    return {"answer": f"关于「{state['question']}」，这是我的解答..."}

# 节点：处理计算问题
def calc_node(state: AssistantState) -> dict:
    return {"answer": f"让我帮您计算：{state['question']}"}

# 识别意图和情绪
def classify_node(state: AssistantState) -> dict:
    q = state["question"]
    if any(kw in q for kw in ["计算", "多少", "分数"]):
        return {"intent": "calculation", "sentiment": "neutral"}
    elif any(kw in q for kw in ["投诉", "不满", "太差了"]):
        return {"intent": "complaint", "sentiment": "negative"}
    return {"intent": "knowledge", "sentiment": "neutral"}
```

9.4.2 多条件分支

```
# 路由函数：同时看意图和情绪
def route_by_intent(state: AssistantState) -> str:
    intent = state.get("intent", "knowledge")
    sentiment = state.get("sentiment", "neutral")
    if intent == "complaint" and sentiment == "negative":
        return "empathy"        # 投诉且情绪负面，先安抚
    if intent == "calculation":
        return "calc"
    return "knowledge"
```

9.4.2 多条件分支

```
workflow = StateGraph(AssistantState)
workflow.add_node("empathy", empathy_node)
workflow.add_node("classify", classify_node)
workflow.add_node("knowledge", knowledge_node)
workflow.add_node("calc", calc_node)
workflow.add_edge("__start__", "classify")
workflow.add_conditional_edges(
    "classify",
    route_by_intent,
    {
        "empathy": "empathy",
        "calc": "calc",
        "knowledge": "knowledge",
    }
)
```

9.4.2 多条件分支

```
workflow.add_edge("empathy", END)
workflow.add_edge("knowledge", END)
workflow.add_edge("calc", END)
app = workflow.compile()
result = app.invoke({"question": "我投诉酒店房间卫生太差了？", "intent": None, "answer": None})
print(result["answer"])  # 输出: 非常抱歉给您带来了不好的体验。我已记录您的反馈，会尽快跟进处理。
```

9.4.3 LLM驱动的路由

前面的路由规则主要依赖关键词或确定性判断。当问题表达更自由时，可以让 LLM 参与意图识别，再由条件边执行确定性分发。

采用关键词匹配识别用户意图的方式会存在盲区，比如，用户说“我不明白这门课”，可能是情绪诉求，也可能是知识问题。这时，可以让 LLM 来判断意图，路由函数只负责解析 LLM 的输出并映射到节点。

在上一节代码基础上，这里把意图识别方式修改为由LLM判断意图：

```
# llm_intent.py
from typing import TypedDict
from langgraph.graph import StateGraph, END
from langchain_ollama import ChatOllama
from langchain_core.messages import SystemMessage, HumanMessage
import os
```

9.4.3 LLM驱动的路由

```
class AssistantState(TypedDict):
    question: str
    intent: str | None
    sentiment: str | None   # 新增：用户情绪
    answer: str | None

llm = ChatOllama(
    model=os.getenv("OLLAMA_MODEL", "gemma4:e4b"),
    base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),
    temperature=0
)
```

9.4.3 LLM驱动的路由

```
# 节点：情绪安抚
def empathy_node(state: AssistantState) -> dict:
    return {"answer": "非常抱歉给您带来了不好的体验。我已记录您的反馈，会尽快跟进处理。"}
# 节点：处理知识问答
def knowledge_node(state: AssistantState) -> dict:
    return {"answer": f"关于「{state['question']}」，这是我的解答..."}
# 节点：处理计算问题
def calc_node(state: AssistantState) -> dict:
    return {"answer": f"让我帮您计算：{state['question']}"}

# classify_node 改为调用 LLM 判断意图
def classify_node(state: AssistantState) -> dict:
    response = llm.invoke([
        SystemMessage(content="""分析学生的问题，返回以下意图之一（只返回单词，不要解释）：
knowledge（知识问答）、calculation（计算问题）、empathy（投诉反馈）"""),
        HumanMessage(content=state["question"])
    ])
    intent = response.content.strip().lower()  # llm.invoke() 返回 AIMessage，.content 取文本
    # 防御：LLM 返回意外值时回退到默认
    valid_intents = {"knowledge", "calculation", "empathy"}
    return {"intent": intent if intent in valid_intents else "knowledge"}
```

9.4.3 LLM驱动的路由

```
# 路由函数，只读取state["intent"]，因为LLM 的判断结果已经写进了状态
def route_by_intent(state: AssistantState) -> str:
    intent = state.get("intent", "knowledge")
    return intent   # 直接返回意图字符串，映射表用相同的 key

workflow = StateGraph(AssistantState)
workflow.add_node("empathy", empathy_node)
workflow.add_node("classify", classify_node)
workflow.add_node("knowledge", knowledge_node)
workflow.add_node("calc", calc_node)
workflow.add_edge("__start__", "classify")
workflow.add_conditional_edges(
    "classify",
    route_by_intent,
    {
        "empathy": "empathy",
        "calc": "calc",
        "knowledge": "knowledge",
    }
)
workflow.add_edge("empathy", END)
workflow.add_edge("knowledge", END)
workflow.add_edge("calc", END)
app = workflow.compile()
result = app.invoke({"question": "这个西瓜怎么这么难吃", "intent": None, "answer": None})
print(result["answer"])  # 输出: 非常抱歉给您带来了不好的体验。我已记录您的反馈，会尽快跟进处理。
```

9.4.4 默认路由与健壮性

LLM 路由并不总是完美，因此，工作流必须考虑无法识别或输出异常的情况。默认路由是保证系统可恢复的重要设计。

路由函数若返回映射表中不存在的 key，工作流将抛出异常。上一节的 LLM 驱动路由已展示了防御写法。规则总结如下：

```
def safe_router(state: AssistantState) -> str:
    intent = state.get("intent", "knowledge")
    # 明确列出所有合法值，非法值统一回退到默认路径
    return intent if intent in {"knowledge", "calculation", "empathy"} else "knowledge"
```

这里可以参考三条实践规则：

路由函数的返回值必须能在映射表中找到对应的key。

始终设置兜底分支（else 或回退逻辑）。

LLM 的输出先写入 State（状态）再路由，不要在路由函数里直接调用 LLM。

9.4.5 条件边的特殊用法：循环

条件边不仅能做分支，也能把流程带回前面的节点，形成循环。循环让智能体具备反复尝试、检查和修正的能力。

条件边不仅可以向前分支，还可以指回已有节点，形成循环。智学助手的“反复追问”场景就需要这种结构——如果学生的问题描述不清楚，就循环要求补充，直到信息足够才继续处理。

```
# condition_edge_cycle.py
from typing import TypedDict
from langgraph.graph import StateGraph, END

class ClarifyState(TypedDict):
    question: str
    clarify_count: int    # 记录已追问次数，防止无限循环
    is_clear: bool        # 问题是否已清晰
    answer: str | None

def check_clarity(state: ClarifyState) -> dict:
    """判断问题是否足够清晰（这里用关键词判断，实际可用LLM来判断）"""
    q = state["question"]
    is_clear = len(q) > 10 and not q.endswith("?")  # 简化判断：长度足够且不是模糊问法
    return {"is_clear": is_clear, "clarify_count": state.get("clarify_count", 0)}
```

9.4.5 条件边的特殊用法：循环

```
def ask_clarify(state: ClarifyState) -> dict:
    """追问节点：提示用户补充信息"""
    count = state.get("clarify_count", 0) + 1
    print(f"[第{count}次追问] 您的问题还不够具体，请补充更多细节。")
    return {"clarify_count": count}

def answer_node(state: ClarifyState) -> dict:
    if state.get("clarify_count", 0) >= 3:
        return {"answer": "非常抱歉，我已经无法进一步追问了，请重新描述您的问题。"}
    return {"answer": f"关于「{state['question']}」，这是我的解答..."}

def should_clarify(state: ClarifyState) -> str:
    if state["is_clear"] or state.get("clarify_count", 0) >= 3:
        return "answer"   # 问题清晰，或追问次数达到上限，继续处理
    return "clarify"      # 问题不清晰，再次追问
```

9.4.5 条件边的特殊用法：循环

```
workflow = StateGraph(ClarifyState)
workflow.add_node("check", check_clarity)
workflow.add_node("clarify", ask_clarify)
workflow.add_node("answer", answer_node)
workflow.add_edge("__start__", "check")
workflow.add_conditional_edges("check", should_clarify, {
    "clarify": "clarify",
    "answer": "answer",
})
workflow.add_edge("clarify", "check")  # 追问后回到检查节点，形成循环
workflow.add_edge("answer", END)
app = workflow.compile()
result = app.invoke({"question": "这门课程为什么不及格?", "clarify_count": 0, "is_clear": False,
"answer": None})
print(result["answer"])
```

Part 05

LangGraph的工具调用

9.5 LangGraph的工具调用

LangGraph 并不要求开发者重新学习一套工具定义方式。LangChain 章节中介绍过的 @tool 装饰器、工具说明和参数类型，在 LangGraph 中仍然可以继续使用。区别在于，LangChain 章节重点说明“工具如何定义、模型如何选择工具”，而 本章的LangGraph则重点说明“工具调用如何作为图中的节点参与状态流转、条件路由、循环执行和异常处理”。

LangGraph 内置的 ToolNode，可以将 LangChain 工具自动转换为 LangGraph 节点，省去手动处理工具调用输入输出的逻辑。在LangGraph中使用工具的完整用法分 4 步：

第一步	第二步	第三步	第四步
用 LangChain 的 @tool 定义工具。	把工具列表绑定给模型：llm.bind_tools(tools)。	把同一组工具交给 LangGraph 的 ToolNode(tools)。	用 tools_condition 判断模型是否请求调用工具：有工具调用就进入 tools 节点，工具执行完再回到 agent 节点。

9.5 LangGraph的工具调用

下面是一个完整代码示例：

```
# langgraph_toolnode_demo.py

from langchain_core.messages import HumanMessage
from langchain_core.tools import tool
from langchain_ollama import ChatOllama
from langgraph.graph import END, START, MessagesState, StateGraph
from langgraph.prebuilt import ToolNode, tools_condition


# 1. 用 LangChain 的 @tool 定义工具
@tool
def query_course_notice(topic: str) -> str:
    """根据主题查询课程通知。topic 可以是：实验报告、课程答疑、期末复习。"""
    notices = {
        "实验报告": "实验报告需在第 10 周周五 18:00 前提交到课程平台。",
        "课程答疑": "课程答疑安排在每周三 19:00-20:00，地点为线上会议室。",
        "期末复习": "期末复习资料将在第 16 周发布，以课程平台通知为准。",
    }
    return notices.get(topic, "未查询到该主题的课程通知。")
```

9.5 LangGraph的工具调用

```
@tool
def calculate_final_score(homework: float, experiment: float, exam: float) -> str:
    """按照课程规则计算总评成绩：作业 30%，实验 30%，期末 40%。"""
    score = homework * 0.3 + experiment * 0.3 + exam * 0.4
    return f"总评成绩为 {score:.1f} 分。"


tools = [query_course_notice, calculate_final_score]


# 2. 把工具绑定给大模型
llm = ChatOllama(
        model="gemma4:e4b",
        temperature=0,
        base_url="http://127.0.0.1:11434",
    )

llm_with_tools = llm.bind_tools(tools)
```

9.5 LangGraph的工具调用

```
# 3. 定义 agent 节点：模型决定是否调用工具
def agent_node(state: MessagesState):
    response = llm_with_tools.invoke(state["messages"])
    return {"messages": [response]}


# 4. 用 ToolNode 包装工具
tool_node = ToolNode(tools)


# 5. 构建 LangGraph
builder = StateGraph(MessagesState)

builder.add_node("agent", agent_node)
builder.add_node("tools", tool_node)

builder.add_edge(START, "agent")
```

9.5 LangGraph的工具调用

```
# agent 运行后：
# 如果模型返回 tool_calls，则进入 tools 节点；
# 如果模型没有返回 tool_calls，则结束。
builder.add_conditional_edges(
    "agent",
    tools_condition,
    {
        "tools": "tools",
        END: END,
    },
)

# 工具执行完后，把 ToolMessage 写回 messages，再回到 agent
builder.add_edge("tools", "agent")
```

9.5 LangGraph的工具调用

```
graph = builder.compile()


if __name__ == "__main__":
    result = graph.invoke(
        {
            "messages": [
                HumanMessage(content="请查询实验报告什么时候提交。")
            ]
        }
    )

    for message in result["messages"]:
        print(message.type, ":", message.content)
```

Part 06

LangGraph的状态和持久化

9.6 LangGraph的状态和持久化

为什么需要状态持久化

使用MessagesState保存对话历史

使用自定义状态保存会话信息

使用SQLite实现检查点持久化

状态历史、恢复与调试

使用建议

9.6.1 为什么需要状态持久化

如果一个问答系统只处理单轮问题，那么每次调用都从空白状态开始并没有明显问题。但是，一旦用户使用追问、补充信息或需要人工审核，系统就必须保存之前发生过的事情。例如，学生先问“实验报告什么时候交”，随后又问“需要提交哪些文件”，第二个问题中的“实验报告”依赖前一次对话；再如，补考申请流程在教师审核前被中断，系统必须在教师给出意见后从中断位置继续，而不是重新执行整个流程。

LangGraph解决这类问题的核心机制是检查点（Checkpoint），这个机制在8.9节有过简单介绍（因为LangChain使用了LangGraph的检查点机制来实现状态保存）。图在运行过程中会在合适的边界保存状态快照，后续调用只要使用相同的thread_id，就可以读取该线程已经保存的状态。因此，状态持久化并不是简单保存一段聊天文本，而是保存图运行过程中的状态版本，使系统能够恢复上下文、支持中断恢复，也便于调试和回溯。

可以把这套机制理解为三个层次：State表示图中流动的数据结构；Checkpointer表示保存和读取状态的组件；thread_id表示一次会话或一次任务的唯一标识。在简单应用中通常使用InMemorySaver；如果需要跨程序重启保存状态，则应使用SQLite、PostgreSQL等持久化存储。

9.6.2 使用MessagesState保存对话历史

对话历史是最常见的状态。LangGraph提供了MessagesState这一预定义状态结构，其中 messages字段用于保存消息列表。每次节点返回新的消息时，LangGraph会把新消息追加到已有消息之后，而不是覆盖原来的消息。这使得多轮对话可以自然地表示为状态中的消息序列。

下面的示例构建一个最小聊天图。图中只有一个chatbot节点，该节点读取当前messages，调用本地Ollama模型生成回复，再把模型回复写回messages。由于编译时传入了InMemorySaver，同一个 thread_id下的后续调用可以读取前一次调用保存的消息历史。具体代码如下：

```
# langgraph_conversation_memory.py
from dotenv import load_dotenv
from langchain_ollama import ChatOllama
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.graph import START, END, MessagesState, StateGraph
import os
```

9.6.2 使用MessagesState保存对话历史

```
llm = ChatOllama(
    model=os.getenv("OLLAMA_MODEL", "gemma4:e4b"),
    base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),
    temperature=0.2,
)

def chatbot(state: MessagesState):
    response = llm.invoke(state["messages"])
    return {"messages": [response]}

builder = StateGraph(MessagesState)
builder.add_node("chatbot", chatbot)
builder.add_edge(START, "chatbot")
builder.add_edge("chatbot", END)

memory = InMemorySaver()
```

9.6.2 使用MessagesState保存对话历史

```
graph = builder.compile(checkpointer=memory)

config = {"configurable": {"thread_id": "student-001"}}

result1 = graph.invoke(
    {"messages": [{"role": "user", "content": "我叫小林，正在学习LangGraph。"}]},
    config=config,
)
print(result1["messages"][-1].content)

result2 = graph.invoke(
    {"messages": [{"role": "user", "content": "我正在学习什么？"}]},
    config=config,
)
print(result2["messages"][-1].content)
```

9.6.3 使用自定义状态保存会话信息

MessagesState适合普通聊天，但很多图应用还需要保存额外字段。例如，课程助手除了消息历史，还可能需要保存当前课程名称、问题类型、检索到的材料和是否需要人工审核。这时可以使用TypedDict自定义状态，并通过Annotate 与add_messages指定messages字段采用追加合并方式。

下面的示例在状态中同时保存messages和course两个字段。remember_course节点从用户输入中提取课程名称，answer节点再利用状态中的课程名称生成回答。这个例子说明，LangGraph的状态不仅可以保存对话历史，也可以保存业务字段。具体代码如下：

```
# langgraph_custom_state_memory.py
from typing import Annotated, TypedDict

from langchain_core.messages import BaseMessage
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.graph import START, END, StateGraph
from langgraph.graph.message import add_messages
```

9.6.3 使用自定义状态保存会话信息

```
class CourseState(TypedDict):
    messages: Annotated[list[BaseMessage], add_messages]
    course: str

def remember_course(state: CourseState):
    last_message = state["messages"][-1].content
    if "数据库" in last_message:
        return {"course": "数据库系统"}
    if "人工智能" in last_message:
        return {"course": "人工智能导论"}
    return {}

def answer(state: CourseState):
    course = state.get("course", "当前课程")
    return {
        "messages": [
            {
                "role": "assistant",
                "content": f"我会结合《{course}》这门课来回答你的问题。"
            }
        ]
    }
```

9.6.3 使用自定义状态保存会话信息

```
builder = StateGraph(CourseState)
builder.add_node("remember_course", remember_course)
builder.add_node("answer", answer)
builder.add_edge(START, "remember_course")
builder.add_edge("remember_course", "answer")
builder.add_edge("answer", END)

graph = builder.compile(checkpointer=InMemorySaver())
config = {"configurable": {"thread_id": "course-thread-001"}}

result = graph.invoke(
    {
        "messages": [{"role": "user", "content": "我正在学习数据库，请帮我复习索引。"}],
        "course": "",
    },
    config=config,
)
print(result["messages"][-1].content)
print("当前课程：", result["course"])
```

9.6.4 使用SQLite实现检查点持久化

如果只使用内存检查点，程序关闭后会话历史就会丢失。为了让状态跨程序重启继续存在，可以使用 SQLite 检查点存储器。SQLite不需要单独启动数据库服务，适合作为学习阶段实验中的持久化方案；生产环境中则更常使用PostgreSQL等支持并发访问的数据库。

使用 SQLite 检查点前，需要安装额外依赖：

```
pip install langgraph-checkpoint-sqlite
```

9.6.4 使用SQLite实现检查点持久化

下面的示例与 9.6.2节基本相同，只是把InMemorySaver换成了SqliteSaver。第一次运行程序时，状态会写入 langgraph_state.db；再次运行程序时，只要使用相同数据库文件和相同 thread_id，就可以继续读取该线程的历史状态。具体代码如下：

```
# langgraph_sqlite_checkpoint.py
from dotenv import load_dotenv
from langchain_ollama import ChatOllama
from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import START, END, MessagesState, StateGraph
import os

load_dotenv()

llm = ChatOllama(
    model=os.getenv("OLLAMA_MODEL", "gemma4:e4b"),
    base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),
    temperature=0.2,
)
```

9.6.4 使用SQLite实现检查点持久化

```
def chatbot(state: MessagesState):
    response = llm.invoke(state["messages"])
    return {"messages": [response]}

builder = StateGraph(MessagesState)
builder.add_node("chatbot", chatbot)
builder.add_edge(START, "chatbot")
builder.add_edge("chatbot", END)

config = {"configurable": {"thread_id": "student-001"}}

with SqliteSaver.from_conn_string("langgraph_state.db") as checkpointer:
    graph = builder.compile(checkpointer=checkpointer)

    result = graph.invoke(
        {"messages": [{"role": "user", "content": "请记住：我的实验题目是课程问答助手。"}]},
        config=config,
    )
    print(result["messages"][-1].content)

    result = graph.invoke(
        {"messages": [{"role": "user", "content": "我的实验题目是什么？"}]},
        config=config,
    )
    print(result["messages"][-1].content)
```

9.6.5 状态历史、恢复与调试

检查点除了支持对话记忆，还能帮助开发者观察状态变化。当图运行多步以后，可以查看某个线程的状态快照，了解当前messages、业务字段和中间结果是什么。这对于调试条件分支、人工审核和长流程恢复特别有用。

下面的示例演示如何读取某个thread_id的当前状态。在前面代码运行后，可以通过get_state 查看该线程最近一次保存的状态：

```
config = {"configurable": {"thread_id": "student-001"}}

current_state = graph.get_state(config)
print("下一步节点：", current_state.next)
print("状态字段：", current_state.values.keys())
print("消息条数：", len(current_state.values["messages"]))
```

如果图中包含interrupt() 节点，检查点机制就更加关键。工作流在中断位置保存状态，外部系统拿到人工输入后，再用相同thread_id 调用 Command(resume=...)，LangGraph才能知道应从哪个位置继续。

9.6.6 使用建议

学习LangGraph的状态和持久化机制时，可以把握以下几条原则。

第一，能不用状态时不要强行引入状态，单轮任务可以直接执行；只要存在追问、人工审核、中断恢复或长流程，就应认真设计状态。

第二，对话历史适合使用MessagesState或 add_messages，业务字段则应根据需要选择覆盖、追加或自定义合并策略。

第三，学习阶段的实验可以使用InMemorySaver，跨重启或多人使用时应选择持久化检查点。

第四，thread_id是状态隔离的边界，必须由应用层稳定管理。

概括来说，LangGraph的“记忆”并不是一个神秘模块，而是状态、检查点和线程标识共同形成的工程机制。状态让图知道当前已经发生了什么，检查点让状态能够被保存和恢复，thread_id 让不同会话互相隔离。理解这一点以后，读者就能把简单聊天记忆、中断恢复、人工审核和生产环境持久化放在同一个框架下理解。

Part 07

LangGraph编程实践

9.7 LangGraph编程实践

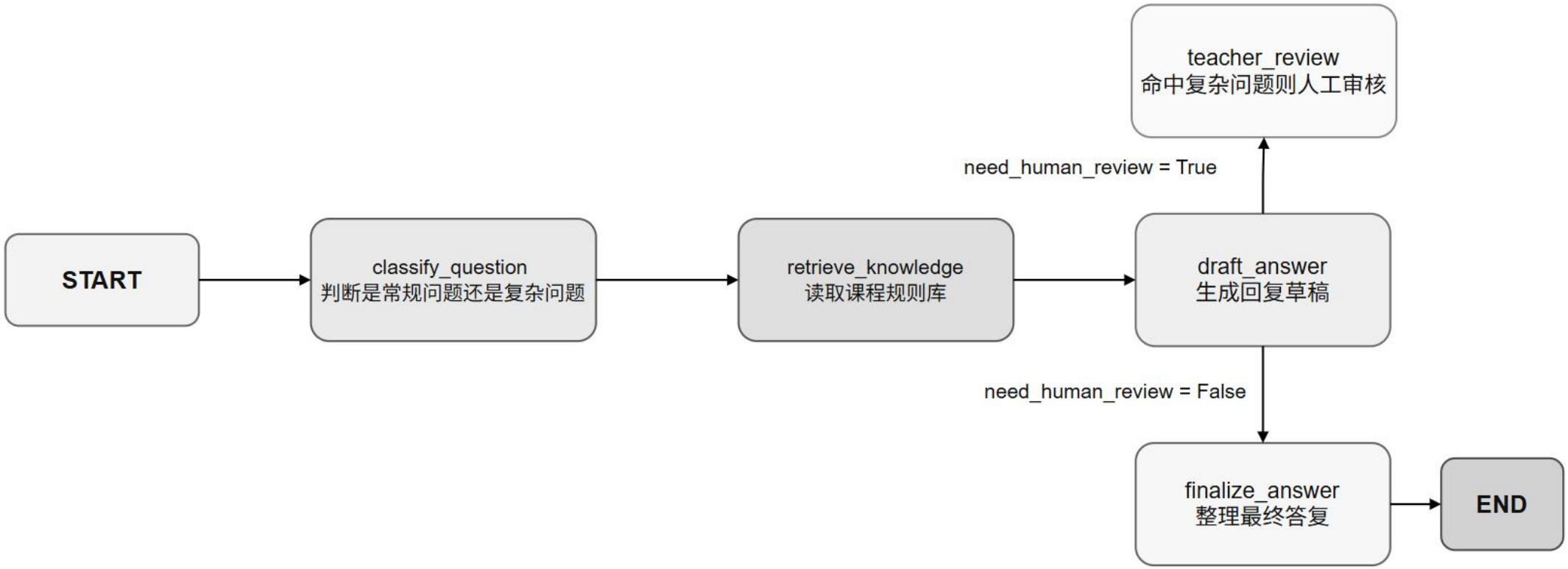
课程助教智能体案例（基础版）

课程助教智能体案例（高级版）

9.7.1 课程助教智能体案例（基础版）

■ 案例需求

假设系统需要处理学生关于课程安排、作业提交、补考申请等问题。若问题可以直接从课程规则中回答，则系统自动完成回复；若问题涉及补考、成绩争议等复杂情形，则必须交由教师复核。整体流程如图所示，依次包含问题分类、规则检索、答案起草、教师复核和最终输出几个环节。



9.7.1 课程助教智能体案例（基础版）

■ 状态设计

在编写节点函数之前，首先要确定图需要保存哪些状态。对本案例而言，至少需要保存原始问题、分类结果、命中的规则文本、答复草稿、是否需要人工复核、教师审核结果和最终答复。日志字段属于便于观察执行过程的增强信息。

这些状态字段之间的分工，可以结合下表来理解：从问题输入到最终答复，每一类信息都对应着不同的运行阶段。

字段	类型	作用
question	str	保存学生原始问题
category	str	记录问题类型，如 routine （常规）或 complex（复杂）
retrieved_docs	list[str]	保存命中的课程规则文本
draft_answer	str	保存系统生成的答复草稿
need_human_review	bool	标识是否需要教师审核
review_result	dict	保存教师的审核输入，如 approved（审核通过）、edited_answer（人工修改答案）
final_answer	str	保存最终输出给学生的答复
logs	list[str]	记录关键执行轨迹，便于观察节点推进过程

9.7.1 课程助教智能体案例（基础版）

■ 代码框架

下面先给出核心代码框架。这个案例最核心的 4 个节点是 classify_question、retrieve_knowledge、draft_answer 和 finalize_answer。状态中的 logs 字段以及 Annotated[list[str], add] 写法用于增强可观察性，不影响主流程对状态更新与路由机制的说明。具体代码如下：

```
from operator import add
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.types import Command, interrupt


class AssistantState(TypedDict):
    question: str  #学生问题
    category: str  # 问题分类，routine / complex
    retrieved_docs: list[str]  # 检索到的知识库文档
    draft_answer: str  # 草稿答案
    need_human_review: bool  # 是否需要人工审核
    review_result: dict  # 教师审核结果
    final_answer: str  # 最终答案
    logs: Annotated[list[str], add]  #工作流日志（自动累加）
```

9.7.1 课程助教智能体案例（基础版）

■ 代码框架

```
RULES = {
    "作业": [
        "作业须在截止时间前提交到课程平台。",
        "逾期提交按照课程说明扣分。"
    ],
    "调课": [
        "调课通知以课程群和教学平台公告为准。"
    ],
    "补考": [
        "补考申请由学院教学办公室统一处理。",
        "需在规定时间内提交材料。"
    ],
}
```

9.7.1 课程助教智能体案例（基础版）

■ 代码框架

上面代码导入了构建状态化工作流必需的工具：

TypedDict

定义工作流的状态结构（存所有数据）。

Annotated + add

实现列表数据累加（日志拼接）。

StateGraph

LangGraph 的核心，用来画工作流流程图。

interrupt

中断工作流，等待人工输入。

Command(resume)

恢复中断的工作流。

InMemorySaver

内存存储器，保存工作流执行状态（对话记忆）。

AssistantState用于定义工作流状态，所有字段清晰定义了工作流需要处理的所有数据，这是整个工作流的“共享账本”，所有节点都能读写这个状态里的数据。logs是工作流日志，会自动累加，不会覆盖。RULES是内置的问答知识库，程序会根据学生问题自动匹配这里的规则。

9.7.1 课程助教智能体案例（基础版）

■ 代码框架

接着实现各个节点函数，具体代码如下：

```
def classify_question(state: AssistantState):
    q = state["question"]
    if any(k in q for k in ["补考", "申诉", "成绩", "特殊情况"]):
        category = "complex"
    else:
        category = "routine"
    return {
        "category": category,
        "logs": [f"classify:{category}"]
    }


def retrieve_knowledge(state: AssistantState):
    q = state["question"]
    docs = []
    for keyword, values in RULES.items():
        if keyword in q:
            docs.extend(values)
    return {
        "retrieved_docs": docs,
        "logs": [f"retrieve:{len(docs)} docs"]
    }
```

9.7.1 课程助教智能体案例（基础版）

```
def draft_answer(state: AssistantState):
    docs = state.get("retrieved_docs", [])
    if docs:
        draft = "；".join(docs)
    else:
        draft = "当前规则库中未找到直接答案，请教师进一步确认。"

    need_review = (state["category"] == "complex") or (len(docs) == 0)
    return {
        "draft_answer": draft,
        "need_human_review": need_review,
        "logs": [f"draft:review={need_review}"]
    }


def route_after_draft(state: AssistantState) -> Literal["teacher_review", "finalize_answer"]:
    if state["need_human_review"]:
        return "teacher_review"
    return "finalize_answer"
```

9.7.1 课程助教智能体案例（基础版）

```
def teacher_review(state: AssistantState):
    payload = {
        "instruction": "请教师审核该复杂问题的答复草稿",
        "question": state["question"],
        "draft": state["draft_answer"],
        "docs": state["retrieved_docs"],
    }
    review = interrupt(payload)
    return {
        "review_result": review,
        "logs": ["review:completed"]
    }


def finalize_answer(state: AssistantState):
    review = state.get("review_result", {})
    if review and review.get("edited_answer"):
        answer = review["edited_answer"]
    else:
        answer = state["draft_answer"]

    return {
        "final_answer": answer,
        "logs": [f"final:{answer[:18]}"]
    }
```

9.7.1 课程助教智能体案例（基础版）

最后构建图并执行调用。当属于常规问题时，工作流可以一路运行到 END；若图命中了 teacher_review 节点，第一次 invoke会返回__interrupt__信息，外部系统需要在相同thread_id下再次调用 graph.invoke(Command(resume=...)) 才能继续。具体代码如下：

```
builder = StateGraph(AssistantState)
builder.add_node("classify_question", classify_question)
builder.add_node("retrieve_knowledge", retrieve_knowledge)
builder.add_node("draft_answer", draft_answer)
builder.add_node("teacher_review", teacher_review)
builder.add_node("finalize_answer", finalize_answer)

builder.add_edge(START, "classify_question")
builder.add_edge("classify_question", "retrieve_knowledge")
builder.add_edge("retrieve_knowledge", "draft_answer")
builder.add_conditional_edges("draft_answer", route_after_draft)
builder.add_edge("teacher_review", "finalize_answer")
builder.add_edge("finalize_answer", END)
```

9.7.1 课程助教智能体案例（基础版）

```
checkpointer = InMemorySaver()
graph = builder.compile(checkpointer=checkpointer)
config = {"configurable": {"thread_id": "course-demo-001"}}

initial = graph.invoke(
    {"question": "老师，我想申请补考，流程是什么？", "logs": []},
    config=config,
)
print(initial["__interrupt__"])

final_state = graph.invoke(
    Command(
        resume={
            "approved": True,
            "edited_answer": "请在教务系统提交补考申请，并于三日内补充证明材料。"
        }
    ),
    config=config,
)
print(final_state["final_answer"])
```

9.7.1 课程助教智能体案例（基础版）

■ **运行结果与分析**

在Trae中运行course-agent.py的输出结果如图所示。

问题 输出 调试控制台 终端 Python Debug Console

```
_projects\LinziyuAITest\AI-Coding\langgraph\course-agent.py'
[Interrupt(value={'instruction': '请教师审核该复杂问题的答复草稿', 'ques
tion': '老师，我想申请补考，流程是什么？', 'draft': '补考申请由学院教学
办公室统一处理。；需在规定时间内提交材料。', 'docs': ['补考申请由学院教
学办公室统一处理。', '需在规定时间内提交材料。']}, id='c78507536ba2a41c6
b6ce89c37f612d8')]
请在教务系统提交补考申请，并于三日内补充证明材料。
```

9.7.2 课程助教智能体案例（高级版）

■ 从基础版案例扩展到高级版案例

把第9.7.1节中的基础版课程助教智能体，扩展到高级版，主要变化是采用真实大模型（不再是基于规则库）。最直接的做法是：保留已有的状态模式和流程骨架，只把 retrieve_knowledge 替换为检索器或向量数据库调用，把 draft_answer 替换为模型生成节点。这样一来，图的工程语义基本不变，改变的只是节点内部能力。

基础版案例的价值在于，它已经把“分类—检索—起草—审核—恢复”这条主流程完整表达出来，因此，后续扩展并不是推倒重写，而是把原本由 if / else 和字符串拼接完成的部分，逐步替换为真实的大模型调用与检索增强模块。对于初学者而言，这种“状态不变、节点替换、能力升级”的演进方式，比一开始直接搭建完整 RAG 系统更容易理解，也更符合工程实践中渐进迭代的思路。

9.7.2 课程助教智能体案例（高级版）

■ 从基础版案例扩展到高级版案例

如果把 draft_answer 节点扩展为真实大模型生成节点，最常见的做法是在节点内部构造提示词，把学生问题、检索结果、回答约束和输出格式要求一起发送给聊天模型。此时，节点不再简单地把多条规则用分号拼接，而是由大模型根据上下文组织答案。为了减少幻觉，提示词中通常需要明确要求大模型“优先依据检索材料回答，不知道时直接说明”，必要时还应要求它给出引用片段或资料来源。

如果把 retrieve_knowledge 节点扩展为真实 RAG 检索节点，则需要把课程规则、实验说明、教学公告、答疑文档等资料整理成可检索语料。一个典型过程包括：首先对原始文档进行清洗，去除无关格式噪声；然后把文本切分为长度适中的片段；接着用嵌入模型把片段表示为向量，并写入向量数据库；在收到学生问题后，再把问题向量化并检索最相关的若干片段，作为后续生成节点的上下文输入。由此可见，RAG 的关键并不仅仅是“接一个向量库”，而是要把文档整理、切分策略、检索数量和上下文拼接方式统一考虑。

9.7.2 课程助教智能体案例（高级版）

■ 从基础版案例扩展到高级版案例

在流程结构上，真实的 RAG 系统往往也不止“检索一次、生成一次”这么简单。较完整的实现可以在 retrieve_knowledge 与 draft_answer 之间增加重排节点、过滤节点或问题改写节点。例如，当用户问题表达模糊时，系统可以先改写查询，再执行向量检索；当检索结果过多或质量较低时，可以增加重排步骤，只把最相关的片段送入生成节点；当检索结果为空时，也可以绕开生成环节，直接进入人工确认或澄清提问节点。这样一来，LangGraph 的图式优势就会进一步体现出来。

需要注意的是，把案例扩展到真实大模型和RAG以后，系统面对的问题将不再只是“代码能否运行”，而是“答案是否可靠”。这里常见的误区有三个：第一，误以为只要接入大模型，答案就一定更准确；第二，忽视文档切分和检索参数，导致模型读到的上下文并不真正相关；第三，只关注最终回答，而不检查中间检索结果和审核分支是否合理。因此，在做提高性实验时，除了看最终输出，更应观察 retrieved_docs 是否命中主题、draft_answer 是否引用了材料、need_human_review 的判定是否符合预期。

9.7.2 课程助教智能体案例（高级版）

■ 可以直接在本地计算机上运行的最小 RAG方案

本节给出一套可以直接在本地计算机上运行的最小 RAG 方案。建议读者先保留前文已经完成的基础版 LangGraph 案例，再新建一个本地 RAG 版本用于对照。为了便于说明，假设本节所有脚本都保存在同一目录下，并准备一份课程资料文件 course_docs_sample.txt（可以从教材官网下载）。

整个过程可以分为七步：

第一步

安装并启动 Ollama。

第二步

下载 Gemma4 和嵌入模型。

第三步

安装 Python 依赖。

第四步

把课程资料切分并写入 Chroma（一个本地的、轻量级的向量数据库）。

第五步

用 LangGraph 把检索节点和生成节点串起来。

第六步

运行基础版本地 RAG。

第七步

把人工审核节点重新接回图中，形成完整案例。

9.7.2 课程助教智能体案例（高级版）

■ 准备本地模型环境

首先，安装 Ollama，并确保本机可以通过命令行正常调用。如果 Ollama 服务尚未启动，可以在Windows系统的cmd命令行界面中执行以下命令启动Ollama：

```
ollama serve
```

然后，下载回答模型和嵌入模型。若机器显存或内存较充足，可以直接使用 gemma4；若是普通教学用笔记本，也可以改用较小规格的 gemma4 变体，例如 gemma4:e4b。嵌入模型的作用是把文本转化成向量，这里使用 nomic-embed-text。具体安装命令如下：

```
ollama pull gemma4:e4b  # 下载回答模型
ollama pull nomic-embed-text  # 下载嵌入模型
```

模型下载完成后，可用下面的命令检查本地模型列表：

```
ollama list
```

9.7.2 课程助教智能体案例（高级版）

■ 安装 Python 依赖

建议在独立虚拟环境中完成安装，以免与其他实验冲突。推荐命令如下：

```
python -m venv langgraph-env  # 在当前目录下创建一个名字叫langgraph-env的独立 Python 环境，用来隔离项目依赖，不污染系统全局 Python
langgraph-env\Scripts\activate
pip install -U langgraph langchain langchain-ollama langchain-chroma chromadb langchain-text-splitters
```

9.7.2 课程助教智能体案例（高级版）

■ 准备课程资料

为了让 RAG 真正发挥作用，系统必须先有一批可以检索的课程文本。本节示例中，可以先建立一个简单的纯文本文件 course_docs_sample.txt，其中保存课程成绩规则、补考说明、实验提交要求和调课通知等资料。一个可直接使用的示例如下（可以从教材官网下载）：

```
实验一要求学生完成 LangGraph 基础流程搭建，并提交运行截图与关键代码说明。
实验报告应在实验完成后一周内提交，迟交将按照课程说明扣减实验分数。
课程总评由平时成绩、实验成绩和期末考试成绩组成，其中实验成绩占 40%。
补考申请由学院教学办公室统一处理，学生需先在教务系统提交申请，再按要求补充材料。
调课通知以课程群和教学平台公告为准，教师口头通知应在平台上补发书面说明。
课程学习建议分为三个阶段：概念理解、代码实践和案例扩展。
当学生提出规则类问题时，系统应优先依据课程资料回答；若资料不足，应明确说明并建议联系教师。
```

与基础版案例相比，这一步的含义十分关键。基础版案例把知识写在 Python 字典中，适合演示流程；而真实 RAG 需要把知识转移到独立文档中，便于后续更新、扩充和统一管理。只要课程资料发生变化，重新建库即可，不必重写 LangGraph 的图结构。

9.7.2 课程助教智能体案例（高级版）

■ 构建本地向量库

接下来要把课程资料切分成多个文档片段，再用嵌入模型将它们写入 Chroma（一种轻量级向量数据库）。具体代码如下（可以从教材官网下载build_vector_db_ollama.py）：

```
# build_vector_db_ollama.py
from pathlib import Path
import shutil
from langchain_core.documents import Document
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_ollama import OllamaEmbeddings
from langchain_chroma import Chroma
BASE_DIR = Path(__file__).resolve().parent
SOURCE_FILE = BASE_DIR / 'course_docs_sample.txt'
PERSIST_DIR = BASE_DIR / 'chroma_db'
COLLECTION_NAME = 'course_rules'
EMBED_MODEL = 'nomic-embed-text'
```

9.7.2 课程助教智能体案例（高级版）

■ 构建本地向量库

```
def load_documents() -> list[Document]:
    text = SOURCE_FILE.read_text(encoding='utf-8')
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=120,
        chunk_overlap=20,
        separators=['\n', '。 ', '； ', '， ', ' ']
    )
    chunks = splitter.split_text(text)
    return [
        Document(page_content=chunk, metadata={'source': SOURCE_FILE.name, 'chunk_id': index})
        for index, chunk in enumerate(chunks)
    ]
```

9.7.2 课程助教智能体案例（高级版）

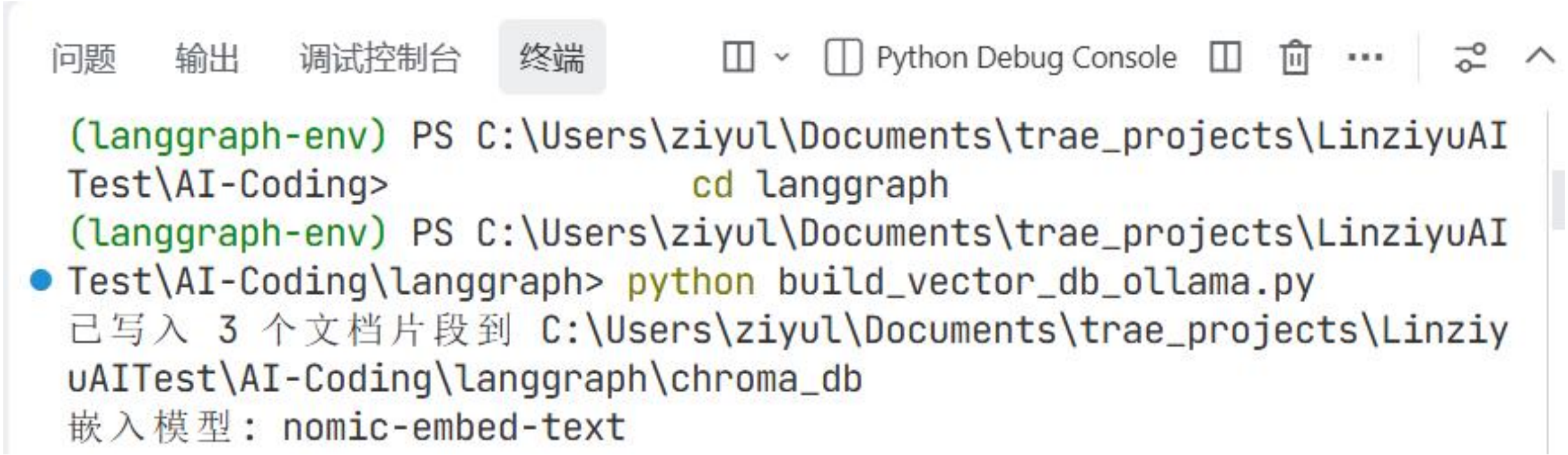
■ 构建本地向量库

```
def main() -> None:
    if not SOURCE_FILE.exists():
        raise FileNotFoundError(f'未找到课程资料文件: {SOURCE_FILE}')
    if PERSIST_DIR.exists():
        shutil.rmtree(PERSIST_DIR)
    documents = load_documents()
    embeddings = OllamaEmbeddings(model=EMBED_MODEL)
    Chroma.from_documents(
        documents=documents,
        embedding=embeddings,
        collection_name=COLLECTION_NAME,
        persist_directory=str(PERSIST_DIR),
    )
    print(f'已写入 {len(documents)} 个文档片段到 {PERSIST_DIR}')
    print(f'嵌入模型: {EMBED_MODEL}')
if __name__ == '__main__':
    main()
```

9.7.2 课程助教智能体案例（高级版）

■ **构建本地向量库**

可以在Trae中运行 build_vector_db_ollama.py，执行结果如图所示。



9.7.2 课程助教智能体案例（高级版）

■ 把 LangGraph 接入本地 RAG

完成建库后，需要把原来基础版案例中的 retrieve_knowledge 和 draft_answer 节点改造为“向量检索 + Gemma4 生成”。具体代码如下（可以到教材官网下载course_agent_ollama_rag.py）：

```
# course_agent_ollama_rag.py
from operator import add
from pathlib import Path
from typing import Annotated
from typing_extensions import TypedDict
from langchain_chroma import Chroma
from langchain_ollama import ChatOllama, OllamaEmbeddings
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.graph import END, START, StateGraph
```

9.7.2 课程助教智能体案例（高级版）

■ 把 LangGraph 接入本地 RAG

```
BASE_DIR = Path(__file__).resolve().parent
PERSIST_DIR = BASE_DIR / 'chroma_db'
COLLECTION_NAME = 'course_rules'
CHAT_MODEL = 'gemma4:e4b'
EMBED_MODEL = 'nomic-embed-text'
class RagState(TypedDict):
    question: str
    retrieved_docs: list[str]
    draft_answer: str
    final_answer: str
    logs: Annotated[list[str], add]
if not PERSIST_DIR.exists():
    raise FileNotFoundError(f'未找到向量库目录: {PERSIST_DIR}。请先运行 build_vector_db_ollama.py。')
llm = ChatOllama(model=CHAT_MODEL, temperature=0)
embeddings = OllamaEmbeddings(model=EMBED_MODEL)
```

9.7.2 课程助教智能体案例（高级版）

■ 把 LangGraph 接入本地 RAG

```
vector_store = Chroma(
    collection_name=COLLECTION_NAME,
    persist_directory=str(PERSIST_DIR),
    embedding_function=embeddings,
)
def retrieve_knowledge(state: RagState):
    docs = vector_store.similarity_search(state['question'], k=3)
    contents = [doc.page_content for doc in docs]
    return {'retrieved_docs': contents, 'logs': [f'retrieve:{len(contents)} docs']}
SYSTEM_PROMPT = '''
你是《AI编程与智能体开发》课程助教。
请严格依据给定材料回答问题。
如果材料不足，请明确说“知识库中未找到充分依据”，不要编造。
回答应尽量简洁、准确、适合本科生阅读。
'''.strip()
```

9.7.2 课程助教智能体案例（高级版）

■ 把 LangGraph 接入本地 RAG

```
def generate_answer(state: RagState):
    if not state['retrieved_docs']:
        return {
            'draft_answer': '知识库中未找到相关材料，请补充课程资料或转人工处理。',
            'logs': ['draft:no_docs']
        }
    context = '\n\n'.join(
        f'[材料{index + 1}] {text}'
        for index, text in enumerate(state['retrieved_docs'])
    )
    prompt = f'''{SYSTEM_PROMPT}
学生问题：
{state['question']}
参考材料：
{context}
请给出最终回答：'''
    answer = llm.invoke(prompt).content
    return {'draft_answer': answer, 'logs': ['draft:done']}
```

9.7.2 课程助教智能体案例（高级版）

■ 把 LangGraph 接入本地 RAG

```
def finalize_answer(state: RagState):
    return {'final_answer': state['draft_answer'], 'logs': [f"final:{state['draft_answer'][:18]}"]}
builder = StateGraph(RagState)
builder.add_node('retrieve_knowledge', retrieve_knowledge)
builder.add_node('generate_answer', generate_answer)
builder.add_node('finalize_answer', finalize_answer)
builder.add_edge(START, 'retrieve_knowledge')
builder.add_edge('retrieve_knowledge', 'generate_answer')
builder.add_edge('generate_answer', 'finalize_answer')
builder.add_edge('finalize_answer', END)
checkpointer = InMemorySaver()
```

9.7.2 课程助教智能体案例（高级版）

■ 把 LangGraph 接入本地 RAG

```
graph = builder.compile(checkpointer=checkpointer)
def main() -> None:
    config = {'configurable': {'thread_id': 'local-rag-demo-001'}}
    result = graph.invoke({'question': '补考申请流程是什么？', 'logs': []}, config=config)
    print('检索结果：')
    for index, doc in enumerate(result['retrieved_docs'], start=1):
        print(f'{index}. {doc}')
    print('\n最终回答：')
    print(result['final_answer'])
if __name__ == '__main__':
    main()
```

9.7.2 课程助教智能体案例（高级版）

■ 运行本地 RAG

完成脚本编写后，可以在Trae中运行course_agent_ollama_rag.py，执行结果如图所示。

问题　输出　调试控制台　终端

```
(langgraph-env) PS C:\Users\ziyul\Documents\trae_projects\LinziyuAITest\AI-Coding\langgraph
>
(langgraph-env) PS C:\Users\ziyul\Documents\trae_projects\LinziyuAITest\AI-Coding\langgraph
> python course_agent_ollama_rag.py
检索结果：
1. 当学生提出规则类问题时，系统应优先依据课程资料回答；若资料不足，应明确说明并建议联系教师
。
2. 补考申请由学院教学办公室统一处理，学生需先在教务系统提交申请，再按要求补充材料。
调课通知以课程群和教学平台公告为准，教师口头通知应在平台上补发书面 说明。
课程学习建议分为三个阶段：概念理解、代码实践和案例扩展。
代码说明。
实验报告应在实验完成后一周内提交，迟交将按照课程说明扣减实验分数。
课程总评由平时成绩、实验成绩和期末考试成绩组成，其中实验成绩占 40%。

最终回答：
补考申请流程如下：

1. **处理机构：** 补考申请由学院教学办公室统一处理。
2. **申请步骤：** 学生需先在教务系统提交申请，然后根据要求补充相关材料。
```

9.7.2 课程助教智能体案例（高级版）

若环境正常，终端会先打印检索到的若干条材料，再打印 Gemma4 生成的最终回答。此时可以把输入问题改为“实验报告什么时候提交”“调课通知以什么为准”“课程总评如何组成”等，观察检索内容和最终答案是否一致。可以看出，从基础版到高级版的智能体，这一扩展过程有三点特别值得把握。

第一，真实 RAG 并没有改变 LangGraph 的基本思想，依然是状态、节点和边在组织整个过程。

第二，RAG 的重点不只是“大模型回答”，而是“先检索，再生成”。

第三，把课程规则从代码中拆分到外部文档后，系统的可维护性会显著提高。

只要读者能够独立完成“准备资料—建向量库—接入 LangGraph—运行问答”这四个动作，就说明已经掌握了从规则版案例走向真实本地 LLM/RAG 的核心方法。

9.7.2 课程助教智能体案例（高级版）

■ **把人工审核节点重新接回本地 RAG 版本**

前面的 course_agent_ollama_rag.py 已经能够完成“检索—生成—输出”的基础流程，但它仍然属于全自动问答。若问题涉及补考、成绩、申诉、特殊情况，或者模型明确表示“知识库中未找到充分依据”，则更稳妥的做法是转入教师人工审核。此时，可以把前文讲过的 interrupt() 与 Command(resume=...) 重新接回图中，使本地 RAG 版本也具备人机协同能力。

实现思路并不复杂：

在状态中增加 need_human_review 和 review_result 两个字段。

在 generate_answer 节点中加入审核判定逻辑。

然后新增 route_after_generate 和 teacher_review 两个节点，其中 teacher_review 通过 interrupt(payload) 暂停工作流，把问题、检索材料和草稿答案交给教师。

教师审核后，再用 Command(resume=...) 在相同 thread_id 下恢复执行。

9.7.2 课程助教智能体案例（高级版）

■ 把人工审核节点重新接回本地 RAG 版本

具体代码如下（可以从教材官网下载course_agent_ollama_rag_review.py）：

```
# course_agent_ollama_rag_review.py
from operator import add
from pathlib import Path
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langchain_chroma import Chroma
from langchain_ollama import ChatOllama, OllamaEmbeddings
from langgraph.checkpoint.memory import InMemorySaver
from langgraph.graph import END, START, StateGraph
from langgraph.types import Command, interrupt
BASE_DIR = Path(__file__).resolve().parent
PERSIST_DIR = BASE_DIR / 'chroma_db'
COLLECTION_NAME = 'course_rules'
CHAT_MODEL = 'gemma4:e4b'
EMBED_MODEL = 'nomic-embed-text'
SENSITIVE_KEYWORDS = ['补考', '成绩', '申诉', '特殊情况']
```

9.7.2 课程助教智能体案例（高级版）

■ 把人工审核节点重新接回本地 RAG 版本

```
class ReviewRagState(TypedDict):
    question: str
    retrieved_docs: list[str]
    draft_answer: str
    need_human_review: bool
    review_result: dict
    final_answer: str
    logs: Annotated[list[str], add]
if not PERSIST_DIR.exists():
    raise FileNotFoundError(f'未找到向量库目录: {PERSIST_DIR}。请先运行 build_vector_db_ollama.py。')
llm = ChatOllama(model=CHAT_MODEL, temperature=0)
embeddings = OllamaEmbeddings(model=EMBED_MODEL)
vector_store = Chroma(
    collection_name=COLLECTION_NAME,
    persist_directory=str(PERSIST_DIR),
    embedding_function=embeddings,
)
```

9.7.2 课程助教智能体案例（高级版）

■ 把人工审核节点重新接回本地 RAG 版本

```
def retrieve_knowledge(state: ReviewRagState):
    docs = vector_store.similarity_search(state['question'], k=3)
    contents = [doc.page_content for doc in docs]
    return {'retrieved_docs': contents, 'logs': [f'retrieve:{len(contents)} docs']}
SYSTEM_PROMPT = '''
你是《AI编程与智能体开发》课程助教。
请严格依据给定材料回答问题。
如果材料不足，请明确说“知识库中未找到充分依据”，不要编造。
回答应尽量简洁、准确、适合本科生阅读。
'''.strip()
def generate_answer(state: ReviewRagState):
    if not state['retrieved_docs']:
        return {
            'draft_answer': '知识库中未找到相关材料，请补充课程资料或转人工处理。',
            'need_human_review': True,
            'logs': ['draft:no_docs']
        }
    context = '\n\n'.join(
        f'[材料{index + 1}] {text}'
        for index, text in enumerate(state['retrieved_docs'])
    )
```

9.7.2 课程助教智能体案例（高级版）

- 把人工审核节点重新接回本地 RAG 版本

```
    prompt = f'''{SYSTEM_PROMPT}
学生问题:
{state['question']}
参考材料:
{context}
请给出最终回答：'''
    answer = llm.invoke(prompt).content
    need_review = any(keyword in state['question'] for keyword in SENSITIVE_KEYWORDS)
    if '知识库中未找到充分依据' in answer:
        need_review = True
    return {
        'draft_answer': answer,
        'need_human_review': need_review,
        'logs': [f'draft:review={need_review}']
    }
```

9.7.2 课程助教智能体案例（高级版）

■ 把人工审核节点重新接回本地 RAG 版本

```
def route_after_generate(state: ReviewRagState) -> Literal['teacher_review', 'finalize_answer']:
    if state['need_human_review']:
        return 'teacher_review'
    return 'finalize_answer'
def teacher_review(state: ReviewRagState):
    payload = {
        'instruction': '请教师审核本地RAG课程助教的答复草稿',
        'question': state['question'],
        'draft_answer': state['draft_answer'],
        'retrieved_docs': state['retrieved_docs'],
    }
    review = interrupt(payload)
    return {'review_result': review, 'logs': ['review:completed']}
def finalize_answer(state: ReviewRagState):
    review = state.get('review_result', {})
    if review and review.get('edited_answer'):
        answer = review['edited_answer']
    else:
        answer = state['draft_answer']
    return {'final_answer': answer, 'logs': [f'final:{answer[:18]}']}
```

9.7.2 课程助教智能体案例（高级版）

- 把人工审核节点重新接回本地 RAG 版本

```
builder = StateGraph(ReviewRagState)
builder.add_node('retrieve_knowledge', retrieve_knowledge)
builder.add_node('generate_answer', generate_answer)
builder.add_node('teacher_review', teacher_review)
builder.add_node('finalize_answer', finalize_answer)
builder.add_edge(START, 'retrieve_knowledge')
builder.add_edge('retrieve_knowledge', 'generate_answer')
builder.add_conditional_edges('generate_answer', route_after_generate)
builder.add_edge('teacher_review', 'finalize_answer')
builder.add_edge('finalize_answer', END)
checkpointer = InMemorySaver()
graph = builder.compile(checkpointer=checkpointer)
```

9.7.2 课程助教智能体案例（高级版）

■ 把人工审核节点重新接回本地 RAG 版本

```
def main() -> None:
    config = {'configurable': {'thread_id': 'local-rag-review-demo-001'}}
    initial = graph.invoke({'question': '补考申请流程是什么？', 'logs': []}, config=config)
    print('第一次调用返回：')
    if '__interrupt__' in initial:
        print(initial['__interrupt__'])
    else:
        print(initial['final_answer'])
        return
    final_state = graph.invoke(
        Command(
            resume={
                'approved': True,
                'edited_answer': '请先在教务系统提交补考申请，再按学院要求在规定时间内补充证明材料。'
            }
        ),
        config=config,
    )
    print('\n恢复执行后的最终回答：')
    print(final_state['final_answer'])
if __name__ == '__main__':
    main()
```

9.7.2 课程助教智能体案例（高级版）

与前面的简单版本的 RAG 相比，这个改进后的RAG版本的关键变化有三个：第一，增加了审核状态字段；第二，在 generate_answer 之后增加了条件路由；第三，使用 interrupt() 让人工审核成为图内部的正式节点，而不是图外部的临时处理逻辑。对于初学者而言，这样的改造非常有代表性，因为它把“检索增强”和“人机协同”这两种能力放到了同一个 LangGraph 图中。

运行顺序也应相应调整。首先确保已经完成 ollama serve、模型下载和向量库构建；然后，可以在Trae中执行course_agent_ollama_rag_review.py，执行结果如图所示。

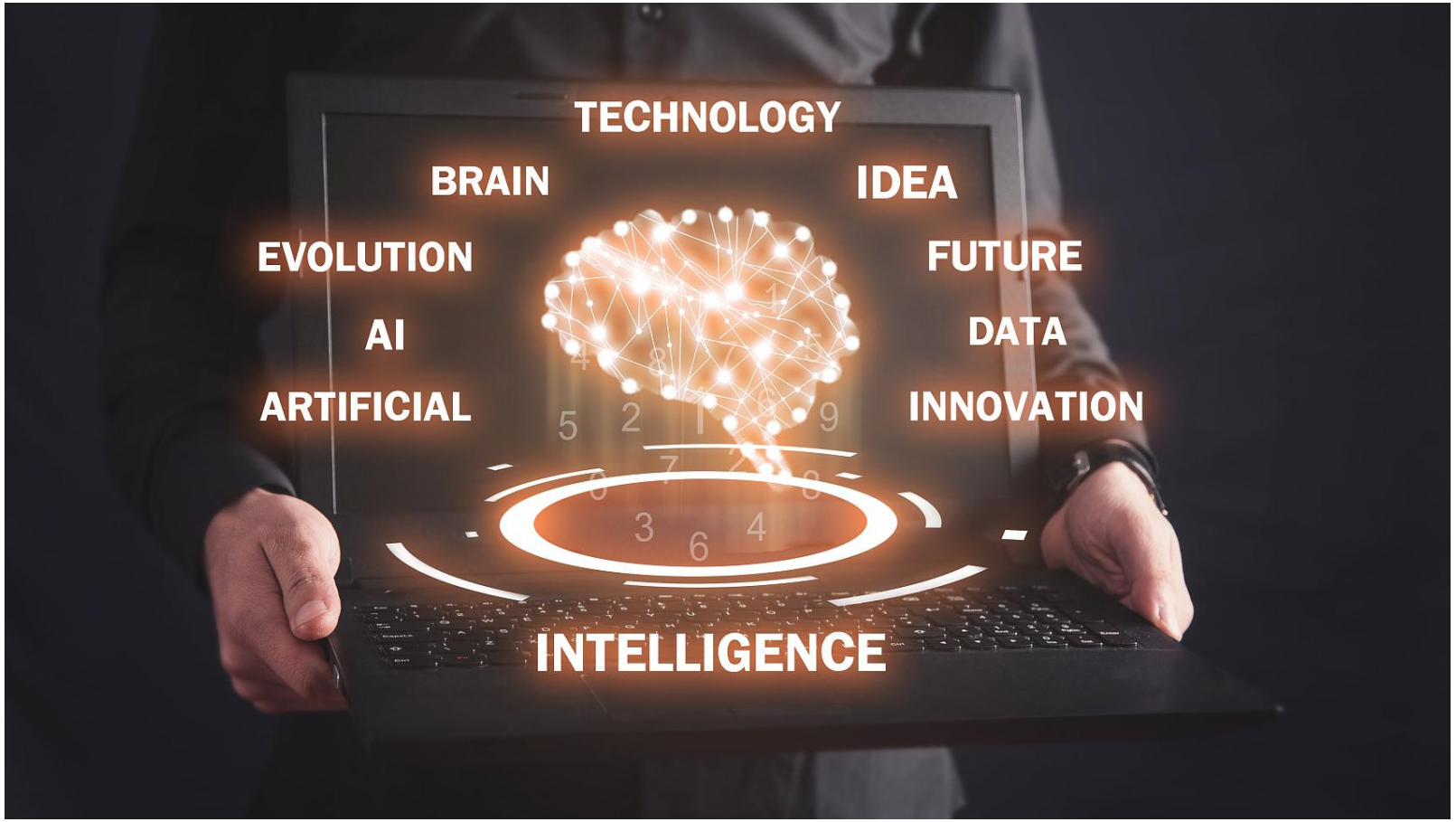
问题　输出　调试控制台　终端

```
(langgraph-env) PS C:\Users\ziyul\Documents\trae_projects\LinziyuAITest\AI-Coding\langgraph
> python course_agent_ollama_rag_review.py
第一次调用返回：
[Interrupt(value={'instruction': '请教师审核本地RAG课程助教的答复草稿', 'question': '补考申请流程是什么？', 'draft_answer': '补考申请流程如下：\n\n1. **处理机构：** 补考申请由学院教学办公室统一处理。\n2. **申请步骤：** 学生需先在教务系统提交申请，然后根据要求补充相关材料。', 'retrieved_docs': ['当学生提出规则类问题时，系统应优先依据课程资料回答；若资料不足，应明确说明并建议联系教师。', '补考申请由学院教学办公室统一处理，学生需先在教务系统提交申请，再按要求补充材料。\n调课通知以课程群和教学平台公告为准，教师口头通知应在平台上补发书面说明。\n课程学习建议分为三个阶段：概念理解、代码实践和案例扩展。', '实验一要求学生完成 LangGraph 基础流程搭建，并提交运行截图与关键代码说明。\n实验报告应在实验完成后一周内提交，迟交将按照课程说明扣减实验分数。\n课程总评由平时成绩、实验成绩和期末考试成绩组成，其中实验成绩占 40%。']}, id='cffbb05c213d85deb197114cf08f7df5')]

恢复执行后的最终回答：
请先在教务系统提交补考申请，再按学院要求在规定时间内补充证明材料。
```

9.7.3智能体部署到生产环境的基本流程

把一个 LangGraph 教学案例运行在本地终端中，与把它部署成可供真实用户访问的在线服务，是两个不同层次的问题。本章的实验代码关注的是“流程是否成立”，而生产系统关注的是“服务是否稳定、状态是否可恢复、知识是否可更新、风险是否可控制”。



9.7.3智能体部署到生产环境的基本流程

因此，在完成原型验证之后，通常还需要经过接口封装、状态持久化、知识库部署、监控告警和灰度发布等多个步骤，才能把智能体真正投入使用，具体如下：

（1）准备运行环境与配置管理。

1. 首先应固定 Python 版本和依赖包版本，使用 requirements.txt 或 pyproject.toml 管理运行环境。

2. 其次，应把模型 API Key、数据库连接串、向量库地址等敏感配置放入环境变量或专用配置中心，而不应直接写在代码文件中。

3. 对于教学场景中的演示代码，很多参数可以直接写死；但在生产环境中，这些配置必须做到可替换、可审计和可回滚。

9.7.3智能体部署到生产环境的基本流程

（2）把图应用封装成服务接口。

> **较常见的做法是使用 FastAPI、Flask 或其他 Web 框架，在服务端接收用户问题，生成或读取 thread_id，再调用 graph.invoke() 或 graph.stream() 执行图。**

> **这样可以把 LangGraph 从“脚本程序”变成“可被前端、移动端或其他系统访问的后端服务”。如果系统包含 interrupt() 节点，还需要提供恢复接口，使审核系统能够在拿到人工结果后，以相同 thread_id 调用 Command(resume=...) 继续执行。**

9.7.3智能体部署到生产环境的基本流程

（3）把检查点存储从内存改为持久化存储。

教学案例中常用 InMemorySaver()，它适合演示，但服务进程一旦重启，状态就会丢失。

进入生产环境后，通常需要把检查点保存到数据库或专门的持久化后端中，使长流程能够跨进程、跨重启恢复。

对于课程助教、审批助手、工单助手这类会话型系统，thread_id、checkpoint 和日志之间的映射关系尤为重要，因为它直接决定了系统能否在异常后接着跑，而不是从头重来。

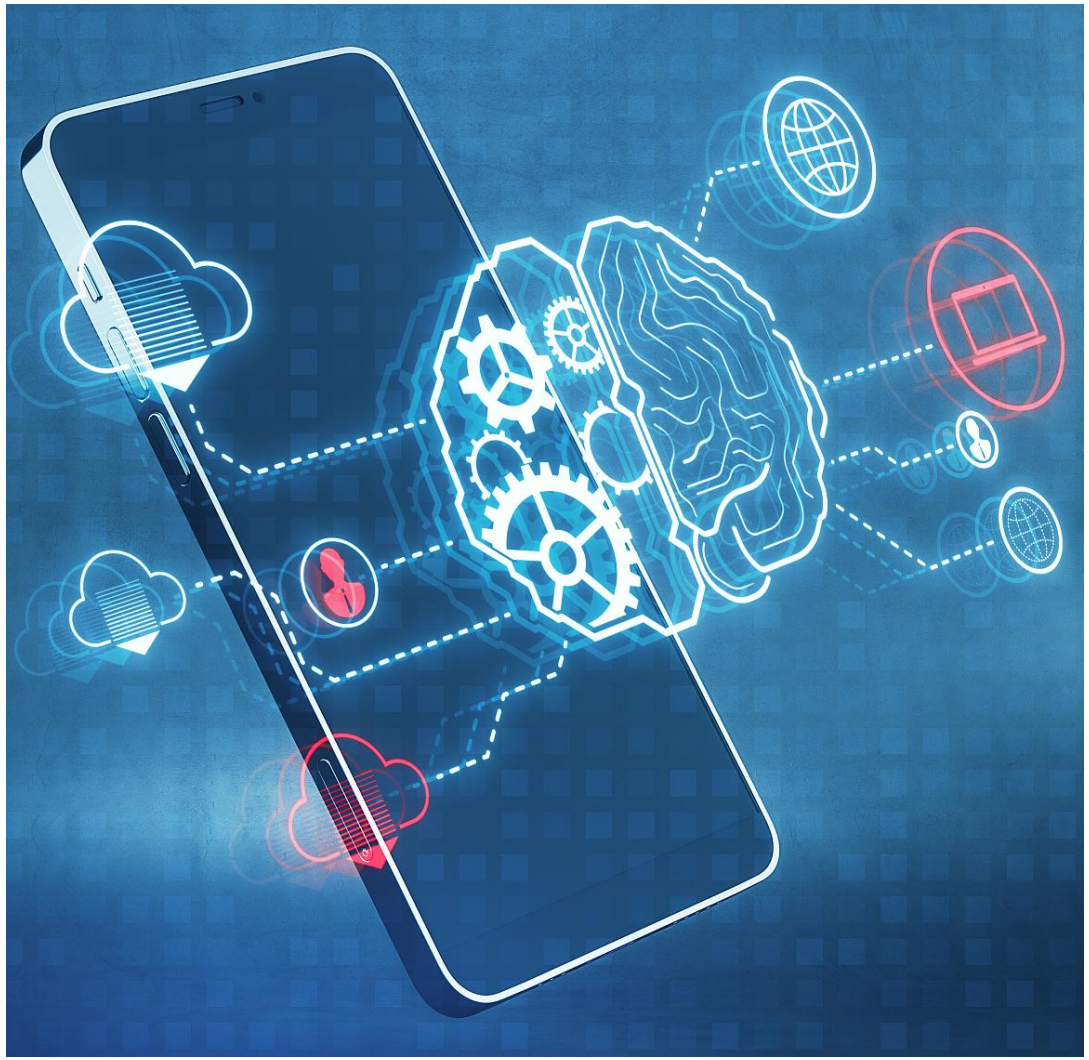
9.7.3智能体部署到生产环境的基本流程

（4）部署知识库与检索服务。

若系统采用 RAG，就不能只部署主程序，还必须同时部署文档处理流水线、嵌入模型、向量索引与检索接口。

常见做法是把“文档清洗—切分—向量化—写入索引”设计为离线更新流程，把“按问题检索 Top-k 片段”设计为在线服务流程。

这样做的好处是，课程通知、实验要求或制度文件一旦变化，只需更新知识库，不必改动图结构本身。



9.7.3智能体部署到生产环境的基本流程

（5）管理外部依赖和工具调用。

01

真实系统中的模型服务、搜索服务、数据库和第三方 API 都可能超时或失败，因此应为关键节点设置超时、重试、降级和异常日志。

02

例如，检索服务短时不可用时，可以返回“请稍后重试”或直接进入人工处理分支；模型调用失败时，可以记录失败原因并触发兜底答复。

03

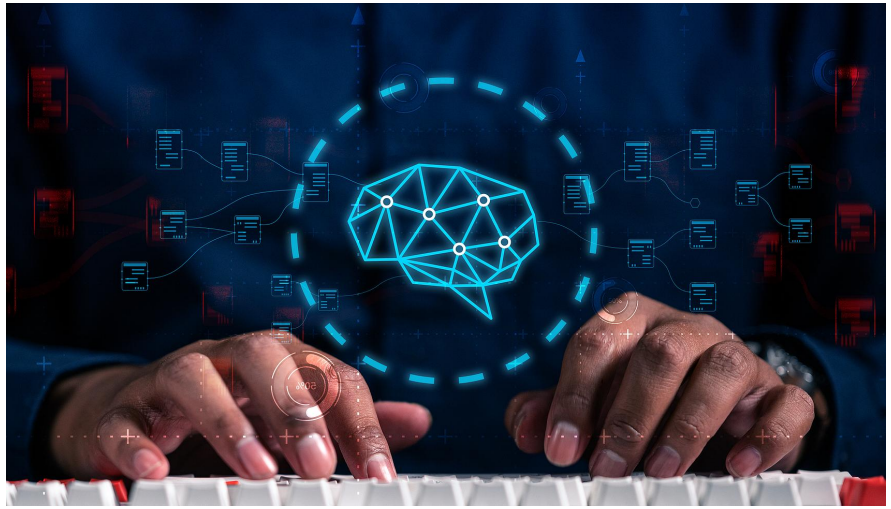
LangGraph 的优势在于，开发者可以把这些异常处理逻辑写成显式节点或显式分支，而不是散落在脚本的各个角落。

9.7.3智能体部署到生产环境的基本流程

（6）接入人工审核、权限控制与安全策略。

> **对于涉及成绩、证书、请假、补考、财务或内部资料的智能体，不能只追求“自动回复”，还必须考虑谁有权限查看、谁有权限恢复流程、哪些问题必须转人工。**

> **通常应在服务外围增加身份认证、访问控制、请求限流和敏感操作审计；在图内部保留 interrupt() 审核节点；在知识检索与生成阶段增加越权访问检查。这样才能把“可用的智能体”进一步提升为“可信的智能体”。**



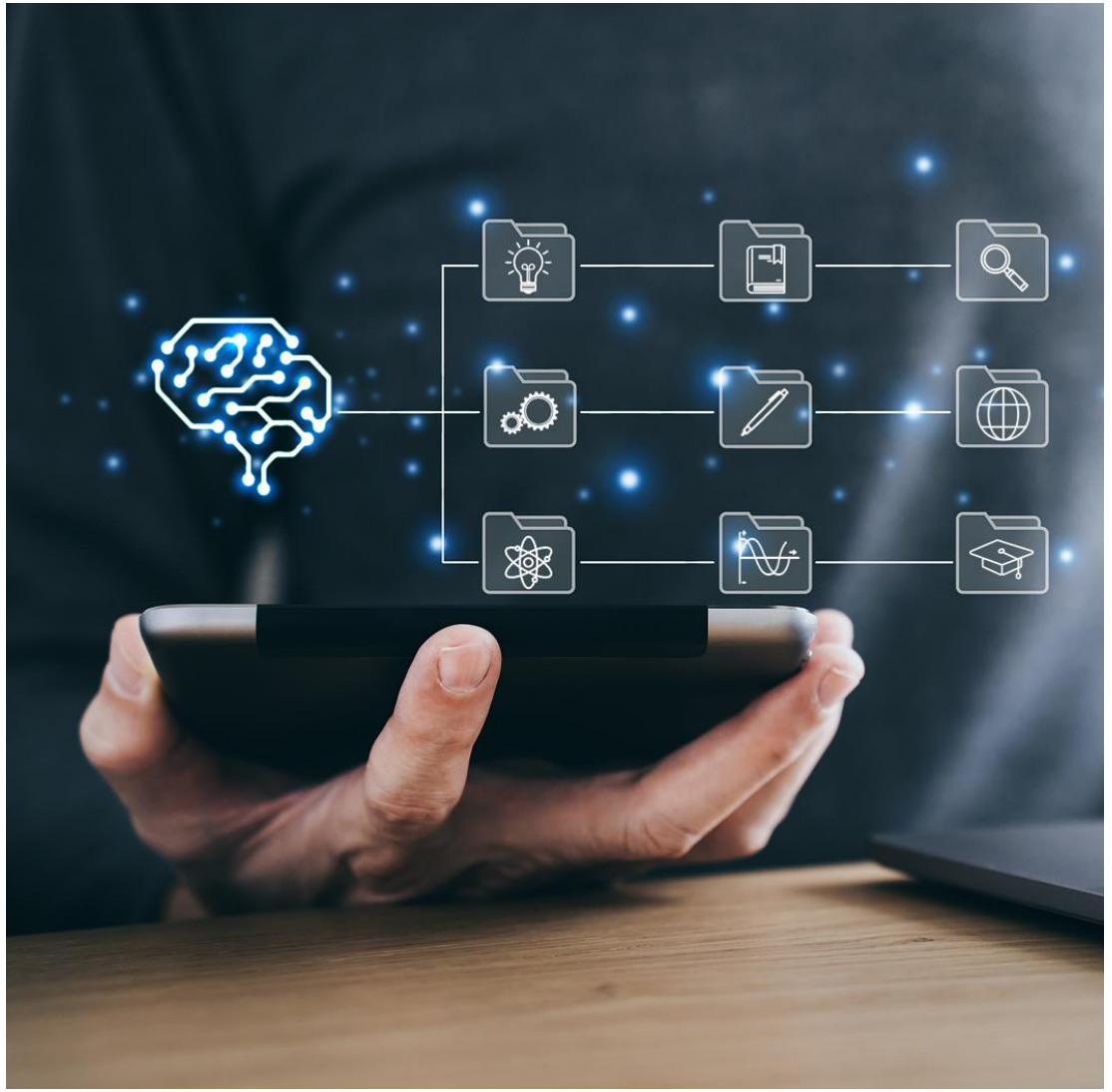
9.7.3智能体部署到生产环境的基本流程

(7) 建立日志、监控与评测机制。

生产环境中的问题往往不是“程序能不能跑”，而是“哪个节点慢了”“为什么这类问题最近答错增多”“恢复流程是否经常卡住”。

因此，应记录每次请求的 thread_id、节点执行顺序、耗时、异常信息、模型调用 token 用量以及人工审核比例；同时结合 LangSmith、Prometheus、Grafana 或平台自带监控工具，持续观察系统运行状态。

对于教学类或问答类智能体，还应准备一小批标准问题作为回归测试集，在每次升级后验证答案质量是否下降。



9.7.3智能体部署到生产环境的基本流程

（8）完成容器化、灰度发布与上线验证。

通常可以先把图应用和依赖服务封装为 Docker 镜像，在测试环境完成联调后，再发布到预发布环境和生产环境。

上线前应重点检查三类问题：

- **第一，常规请求是否能稳定返回。**
- **第二，人工审核与恢复流程是否能闭环。**
- **第三，知识库更新后是否会影响历史会话。**

正式发布时，较稳妥的方法是先对少量用户灰度开放，确认无明显错误后再逐步扩大范围；一旦发现严重问题，则应能快速回滚到上一版本。

9.8 本章小结

本章围绕 LangGraph 智能体开发框架展开，介绍了其应用场景、核心概念、快速入门、条件边与条件节点、 编程实践。

1. LangGraph 的核心价值不在于替代大模型，而在于把状态、控制流、持久化和人机协同组织成可靠的执行图。

2. 通过本章的理论说明与案例分析，可以把 LangGraph 理解为一种面向复杂智能体流程的运行时组织方式。

3. 从本章内容可以归纳出三个基础能力：能够定义一个简单状态；能够用节点、边和条件边搭建一个小图；能够用 interrupt() 与 Command(resume=...) 完成一次人工复核。

谢谢！

林子雨　副教授　厦门大学

附录A：主讲教师林子雨简介

主讲教师：林子雨

单位：厦门大学计算机科学与技术系

E-mail: ziyulin@xmu.edu.cn

个人网页：http://dblab.xmu.edu.cn/post/linziyu

数据库实验室网站：http://dblab.xmu.edu.cn

- 林子雨，男，1978年出生，博士（毕业于北京大学），全国高校知名大数据与人工智能教师，入选“2021年高校计算机专业优秀教师奖励计划”。现为厦门大学计算机科学与技术系副教授，厦门大学信息学院实验教学中心主任，曾任厦门大学信息科学与技术学院院长助理、晋江市发展和改革局副局长。中国计算机学会数据库专业委员会执行委员，中国计算机学会信息系统专业委员会执行委员。
- 国内高校首个“数字教师”提出者和建设者，厦门大学数据库实验室负责人，厦门大学云计算与大数据研究中心主要建设者和骨干成员，2013年度、2017年度、2020年度、2023年度和2026年度厦门大学教学类奖教金获得者，荣获2024年福建省高等教育教学成果奖特等奖（个人排名第七）、2022年福建省高等教育教学成果奖特等奖（个人排名第一）、2018年福建省高等教育教学成果奖二等奖（个人排名第一）、2018年国家精品在线开放课程、2021年国家级线上一流本科课程、2020年国家级线上一流本科课程。编著出版了20余本大数据与人工智能系列教材，被国内1000多所高校采用，3本教材入选教育部“十四五”普通高等教育本科国家级规划教材；建设了国内高校首个大数据课程公共服务平台，为教师教学和学生学习大数据课程提供全方位、一站式服务，年访问量超过400万次，累计访问量超过3200万次。大数据系列MOOC课程入选“2023年教育部国家智慧教育公共服务平台应用典型案例”。2025年发布4个大模型科普报告，全网累计浏览量超过1000万，并应邀为高校、政府和企业做170余场大模型讲座。
- 主要研究方向为数据库、数据仓库、数据挖掘、大数据、云计算和物联网，并以第一作者身份在《软件学报》《计算机学报》和《计算机研究与发展》等国家重点期刊以及国际学术会议上发表多篇学术论文。作为项目负责人主持的科研项目包括1项国家自然科学青年基金项目(No.61303004)、1项福建省自然科学青年基金项目(No.2013J05099)和1项中央高校基本科研业务费项目(No.2011121049)，主持的教改课题包括1项2016年福建省教改课题、1项2016年教育部产学协作育人项目、1项2024年教育部产学协作育人项目。

附录B：大数据学习路线图

大数据学习路线图访问地址：http://dblab.xmu.edu.cn/post/10164/

附录C：林子雨大数据系列教材

了解全部教材信息：http://dblab.xmu.edu.cn/post/bigdatabook/

附录D：林子雨人工智能通识系列教材

厦门大学林子雨人工智能通识系列教材

被众多高校采用，满足不同高校差异化教学需求

了解全部教材信息：http://dblab.xmu.edu.cn/post/ai-book/

附录E：林子雨Python系列教材

了解全部教材信息：https://dblab.xmu.edu.cn/post/python-books/

谢谢！

林子雨　副教授

厦门大学