第 20 课：LangChain 核心概念，重点版

1. LangChain 是什么？

面试表达：LangChain 是一个大模型应用开发框架，主要解决模型调用、Prompt 管理、工具调用、结构化输出、RAG 检索、Agent 编排、记忆和可观测性等问题。它的价值不是让模型更聪明，而是让大模型应用更容易工程化。

更口语一点：如果直接调 OpenAI / DeepSeek / 通义 API，我要自己管理 prompt、history、tools、JSON 解析、RAG 检索、异常重试。LangChain 把这些常见模式封装成统一接口，方便快速搭建 LLM 应用和 Agent 应用。

和你项目的关系：你的 Policy KB Assistant 其实很多地方是“自己手写版 LangChain 思路”：

你项目里的东西	LangChain 里的对应概念
Agent Planner	Agent / Tool Calling
ToolPlan JSON	Structured Output
RAG 检索服务	Retriever / RAG Chain
Prompt 证据组织	Prompt / Context Engineering
trace_id / audit	LangSmith tracing 类似思想
L1/L3 记忆	Memory / Checkpoint
工单工具	Tools

所以你面试时可以说：我没有完全依赖 LangChain 黑盒，而是自己实现了一个更可控的 Agent + RAG 架构。后来学习 LangChain / LangGraph 后，我能把自己项目里的 Planner、ToolPlan、RAG Retriever、Memory、Audit 分别映射到框架概念上。

这个表达很重要，因为它比“我会用 LangChain”更有含金量。

2. ChatModel 是什么？

LangChain 里的 ChatModel 可以理解为“聊天模型接口”。官方文档里，Chat Model 是以一组 messages 作为输入，并返回 message 作为输出，而不是传统单字符串输入输出。

面试表达：ChatModel 是 LangChain 对聊天模型的统一封装。不同厂商的 API 格式可能不同，比如 OpenAI、Claude、Gemini、通义、智谱，但 LangChain 会尽量提供统一的调用方式，方便切换模型提供商。

你项目里对应的是：

```
用户问题
  -> system prompt
  -> user message
  -> 检索证据
  -> LLM 生成答案 / ToolPlan
```

在 LangChain 里，会更标准地表示成：

```
SystemMessage
HumanMessage
AIMessage
ToolMessage
```

LangChain 的 messages 是模型上下文的基本单位，通常包含 role、content 和 metadata。

面试官可能问：**为什么不用普通 LLM，而用 ChatModel?**

你可以答：现在主流大模型基本都是 chat completion 风格，需要区分 system、user、assistant、tool 等角色。ChatModel 更适合多轮对话、工具调用、Agent 状态管理和 RAG 问答。

普通 LLM 就像和一个**“输入-输出”**机器对话，你给一句完整的话（Prompt），它给你补全下文

模拟了一场**真正的对话**。输入是一个“消息列表”(有角色标签、上下文管理)，其中包含了用户问题、AI 历史回复，甚至系统指令等

3. PromptTemplate 是什么？

PromptTemplate 本质上是“提示词模板”。

普通写法：

```
prompt = f"""
你是企业政策助手。
请根据以下证据回答用户问题：

证据：
{context}

用户问题：
{question}
"""
```

LangChain 的 PromptTemplate 思路是把 {context}、{question} 这种变量抽出来，让 prompt 可复用、可组合、可管理。

面试表达：PromptTemplate 是把提示词参数化。它的作用是避免把 prompt 写死在代码里，让不同输入可以复用同一套提示词结构，比如 RAG 里的 context、question、user_role、ticket_id 都可以作为变量注入。

和你项目的关系：

你项目的 RAG prompt 里可能会有：

```
你只能基于证据回答。
证据不足时拒答。
必须返回 citation。
不要编造制度条款。
```

这就是典型 RAG PromptTemplate。

4. OutputParser / Structured Output 是什么？

早期 LangChain 常讲 OutputParser，也就是“把模型自然语言输出解析成结构化数据”。

但现在更推荐你重点掌握 **Structured Output**。官方文档说明，Structured Output 可以让 Agent 返回固定、可预测的数据格式，比如 JSON object、Pydantic model 或 dataclass，而不是让业务代码解析一大段自然语言。

面试表达：Structured Output 是让模型按指定 schema 输出结构化结果，常用于分类、信息抽取、工具参数生成和 Agent Planner。相比纯自然语言输出，它更适合后端系统处理，也更容易做校验、重试和审计。

你项目里最强的对应点就是：

```
用户说：“帮我把上一单催一下”
        ↓
Planner 输出 ToolPlan：
{
  "intent": "ticket_followup",
  "tool": "add_ticket_comment",
  "args": {
    "ticket_id": "...",
    "comment": "用户请求催办"
  }
}
```

这个就是结构化输出。

你的面试表达可以升级成：在我的项目里，Agent 不直接执行数据库操作，而是先生成结构化 ToolPlan。后端再对 ToolPlan 做 schema 校验、工具白名单校验、资源权限校验、状态校验、危险操作确认和审计日志记录。这样可以避免 LLM 直接产生不可控副作用。

这是非常高频、非常值钱的 Agent 工程表达。

5. Tool Calling 是什么？

Tool Calling 就是让模型在需要时调用外部工具。
LangChain 官方把 tools 描述为能扩展 Agent 能力的可调用函数，这些工具可以获取实时数据、执行代码、查询数据库，或者执行外部动作；工具有明确输入输出，模型根据上下文决定是否调用以及传什么参数。
面试表达：Tool Calling 是让大模型不只是生成文本，而是能调用后端函数完成动作。比如查工单、创建工单、查询知识库、调用搜索接口、发送通知等。模型负责判断调用哪个工具和生成参数，后端负责真正执行和安全校验。
你项目里的工具可以这样包装：

用户意图	Tool
查询政策	kb_search
创建工单	create_ticket
继续草稿	continue_ticket_draft
查工单	get_ticket
补充评论	add_ticket_comment
催办	followup_ticket
取消工单	cancel_ticket

但一定要记住：Tool Calling 不是让模型获得无限权限，而是让模型提出“调用建议”。真正执行必须由后端控制。
这是你前面 JWT / 权限 / Agent 安全知识的延伸。

6. Retriever / RAG Chain 是什么？

Retriever 是检索器，负责根据用户问题从知识库里取相关内容。
LangChain 的 RAG 文档里提到两种常见方式：一种是 **2-Step RAG**，先检索再把结果交给 LLM；另一种是 **Agentic RAG**，把检索器作为工具，让 Agent 自己决定什么时候检索。
你现在项目更接近：

```
用户问题
  -> 判断是知识问答
  -> 检索 Qdrant + BM25
  -> RRF 融合
  -> CrossEncoder rerank
  -> 组织证据
  -> LLM 回答
```

这属于比较典型的 **2-Step RAG / 工程化 RAG**。
如果变成 Agentic RAG，就是：

```
用户问题
  -> Agent 判断是否需要查知识库
  -> 调用 kb_search 工具
  -> 拿到证据
  -> 再回答
```

面试表达：简单 RAG 可以写成固定工作流：每次问题都先检索再回答。Agentic RAG 则把检索封装成工具，由 Agent 判断是否需要检索。前者稳定、可控、低成本；后者灵活，但更需要工具调用约束和评测。
这句话很适合面试。

7. LangChain Agent 是什么？

LangChain 现在的核心入口之一是 create_agent。官方示例里，一个 agent 可以由 model、tools、system_prompt 组成。
你可以把 Agent 理解成：

```
Agent = LLM + Prompt + Tools + Loop + State
```

更工程化一点：

用户输入
-> 模型理解意图
-> 判断是否需要工具
-> 生成工具参数
-> 执行工具
-> 读取工具结果
-> 继续推理
-> 输出最终答案

面试表达：Agent 和普通 ChatBot 的区别是：普通 ChatBot 主要回答文本，Agent 可以基于目标进行多步决策，并调用工具和外部系统完成任务。

但是你要补一句安全边界：在企业系统里，我不会让 Agent 直接操作数据库，而是让它生成结构化计划，由后端进行权限校验、参数校验、状态校验和审计后再执行。

这就是你的项目优势。

8. LangChain 和 LangGraph 的关系

框架	适合场景
LangChain	快速搭 LLM 应用、RAG、简单 Agent
LangGraph	复杂 Agent 流程、状态机、多节点、多分支、人类确认
LangSmith	调试、追踪、评测、可观测性

官方文档也明确说，LangGraph 是更底层的编排框架，适合高级需求，比如把确定性 workflow 和 agentic workflow 组合起来；LangChain 则适合用 create_agent 快速构建可定制 harness。

你项目更像 LangGraph 思路，因为你有：

planner node
memory node
draft node
confirm node
execution node
audit node

所以你后面学 LangGraph 会很顺。

面试表达：LangChain 更像上层封装，适合快速搭建 Agent；LangGraph 更像状态机和流程编排框架，适合复杂业务场景。我的项目里有 Planner、Memory、Confirm、Execution、Audit 等明确节点，所以更适合用 LangGraph 思路建模。

9. LangChain 不是必须用，但必须懂

中小公司面试官可能问：**你项目为什么没直接用 LangChain?**

你可以答：我的项目核心是企业知识库和工单系统，需要强权限校验、审计、确认机制和可控执行。如果完全依赖框架黑盒，安全边界不够清晰。所以我选择核心链路自研，但设计思想和 LangChain / LangGraph 是一致的：模型负责理解和规划，后端负责工具执行和安全约束。

再补一句：如果后续要快速接入更多模型、工具生态或做 Agent tracing，可以把现有的 RAG Retriever、Ticket Tool、Planner 输出逐步适配到 LangChain / LangGraph。

这就是高级回答。

10. 这一课你真正要掌握的 8 个核心点

概念	一句话
LangChain	LLM 应用开发框架
ChatModel	统一封装聊天模型调用
PromptTemplate	参数化管理提示词

Structured Output	让模型输出可校验结构化数据
Tool Calling	让模型调用外部工具
Retriever	根据问题检索知识库内容
RAG Chain	检索 + 证据组织 + 生成
Agent	模型基于目标调用工具完成任务

11. 面试高频问法与标准回答

Q1：LangChain 是什么？

答：LangChain 是一个大模型应用开发框架，主要用于封装模型调用、Prompt、工具调用、结构化输出、RAG、Agent 和记忆等能力。它的作用不是提升模型本身能力，而是帮助开发者更快构建可维护的大模型应用。

Q2：LangChain 和直接调用大模型 API 有什么区别？

答：直接调用 API 更轻量，但 prompt、history、tools、JSON 解析、RAG 检索、异常处理都要自己写。LangChain 提供统一抽象，可以更方便地组合模型、提示词、工具、检索器和 Agent 流程。简单项目可以直接调 API，复杂 Agent / RAG 应用用框架会更方便。

Q3：PromptTemplate 有什么用？

答：PromptTemplate 是把提示词模板化和参数化，比如把 question、context、user_role 作为变量传入。这样可以提高 prompt 的复用性，也方便统一维护 RAG 问答、分类、信息抽取等不同任务的提示词。

Q4：什么是 Structured Output？

答：

Structured Output 是让模型按指定 schema 输出结构化结果，比如 JSON 或 Pydantic 对象。它常用于 Agent Planner、工具参数生成和信息抽取。相比自然语言输出，结构化输出更适合后端校验、执行和审计。

Q5：Tool Calling 是什么？

答：

Tool Calling 是让模型根据用户意图选择外部工具，并生成调用参数。比如查知识库、查工单、创建工单、调用搜索接口等。但在企业系统里，模型只负责生成调用计划，真正执行必须由后端做权限、参数、状态和审计校验。

Q6：Retriever 在 RAG 里负责什么？

答：Retriever 负责根据用户问题从知识库中检索相关文档片段。检索结果会作为上下文交给 LLM 生成答案。我的项目里 Retriever 不只是向量检索，还包括 Dense + BM25 混合检索、RRF 融合和 CrossEncoder rerank。

Q7：Agent 和普通 RAG 有什么区别？

答：普通 RAG 通常是固定流程：问题进来后先检索，再生成答案。Agent 则可以根据任务目标决定是否调用工具、调用哪个工具、是否多步执行。RAG 更稳定可控，Agent 更灵活，但也更需要安全约束和评测。

Q8：你的项目和 LangChain 有什么关系？

答：

我的项目没有简单套 LangChain，而是自研了一个更可控的企业 Agent + RAG 架构。但它和 LangChain 的思想是对应的：RAG 检索器对应 Retriever，Planner 输出对应 Structured Output，工单操作对应 Tools，确认和审计对应 Agent 安全机制。这样既能体现我理解框架，也能体现我知道企业项目为什么要控制安全边界。

---

12. 你项目里最应该这样包装

你可以在简历或面试里说：

项目中实现了一个面向企业制度知识库和 IT 工单的 Agent + RAG 系统。RAG 侧完成 PDF 解析、结构化切分、metadata 保存、Dense + BM25 混合检索、RRF 融合、CrossEncoder 重排、citation 和拒答机制；Agent 侧没有让 LLM 直接操作数据库，而是让模型生成结构化 ToolPlan，再由后端进行 schema 校验、权限校验、状态校验、危险操作确认和审计日志记录。整体设计和 LangChain / LangGraph 的思想一致，但核心链路自研，安全边界更清晰。
这个回答可以直接背。

第 21 课：Tool Calling / Structured Output 深入版

这一课是你项目里 **Agent 安全、ToolPlan、Planner JSON 评测** 的核心。
先记住一句话：Structured Output 解决“模型输出能不能被程序稳定解析”；Tool Calling 解决“模型能不能提出调用外部工具的计划”；后端 Executor 解决“这个计划能不能安全执行”。

1. Tool Calling 是什么？

LangChain 官方对 tools 的描述是：工具可以扩展 Agent 的能力，比如获取实时数据、执行代码、查询数据库、执行外部动作；本质上，tool 是有明确输入输出的可调用函数，模型根据上下文决定是否调用，以及传什么参数。
面试表达：
Tool Calling 是让大模型根据用户意图选择外部工具，并生成结构化调用参数。比如用户说“帮我查一下上一单”，模型不应该直接回答，而应该生成调用 get_ticket 工具的请求。
你项目里的例子：

```
{
  "tool": "get_ticket",
  "args": {
    "ticket_id": "T20260617001"
  }
}
```

注意，这还不是执行数据库查询，只是**模型提出调用计划**。

2. Tool Calling 的完整链路

LangChain 的 agent loop 可以抽象成两步：第一步模型调用，模型返回普通回复或者工具执行请求；第二步执行工具，把工具结果再交回模型，这个循环会持续到模型决定结束。
工程链路可以理解成：

```
用户输入
  -> LLM 判断意图
  -> LLM 选择工具
  -> LLM 生成工具参数
  -> 后端校验工具名和参数
  -> 后端执行工具
  -> 工具结果返回给 LLM
  -> LLM 生成最终回复
```

你项目里的更安全版本是：

```
用户输入
  -> Planner 生成 ToolPlan
  -> schema 校验
  -> 工具白名单校验
  -> 资源权限校验
  -> 资源状态校验
  -> 高风险动作确认
  -> Executor 执行
  -> Audit 记录
  -> 返回用户
```

这就是你项目比“普通 Agent demo”强的地方。

3. Tool Call 里一般有什么？

一个工具调用通常至少包含三部分：

```
{
  "name": "create_ticket",
  "args": {
    "title": "无法登录系统",
    "category": "account",
    "priority": "high"
  },
  "id": "call_xxx"
}
```

LangChain 文档里也提到，agent 发出的工具调用通常包含工具名 name、结构化参数 args、以及用于关联调用结果的唯一 id。

你项目里的 ToolPlan 可以这样对照：

通用 Tool Calling	你项目 ToolPlan
name	tool
args	args
id	request_id / trace_id / tool_call_id
tool result	service 执行结果
final response	Agent 最终回复

4. Structured Output 是什么？

Structured Output 是让模型输出固定格式的数据，而不是一段随意自然语言。LangChain 文档说，它可以让 agent 返回可预测的数据格式，例如 JSON object、Pydantic model 或 dataclass，并且结果会被捕获、校验后放到 agent state 的 structured_response 里。

面试表达：Structured Output 是让模型按指定 schema 输出结构化结果，方便后端解析、校验、执行和审计。它适合分类、信息抽取、工具参数生成、Planner 输出等场景。

普通自然语言输出：

```
用户想创建一个高优先级账号问题工单。
```

Structured Output：

```
{
  "intent": "create_ticket",
  "need_confirm": false,
  "args": {
    "title": "账号无法登录",
    "category": "account",
    "priority": "high"
  }
}
```

后端更喜欢第二种，因为它能直接校验和执行。

5. Structured Output 和 Tool Calling 的区别

对比点	Structured Output	Tool Calling
核心问题	输出格式是否稳定	是否调用外部工具
输出目标	给程序消费	让工具执行

是否一定执行动作	不一定	通常会触发工具执行
典型用途	分类、抽取、Planner JSON	查数据库、建工单、检索知识库
你项目对应	ToolPlan JSON	Ticket tools / KB tools

6. Structured Output 的两种实现方式

现在常见有两类：

第一类是 **Provider-native Structured Output**。也就是模型供应商本身支持结构化输出，服务端会尽量按 schema 约束模型结果。

第二类是 **Tool-based Structured Output**。也就是把“返回结构化结果”伪装成一次工具调用，模型通过调用一个特殊 tool 来提交 JSON 参数。

LangChain 文档里也区分了 ProviderStrategy 和 ToolStrategy：前者使用模型供应商原生结构化输出，后者使用 tool calling 来实现结构化输出；直接传 schema 时，LangChain 会根据模型能力自动选择策略。

面试不用背 API 名字，但要知道本质：

有些模型/API 原生支持 JSON schema 约束；如果不支持，也可以通过 function/tool calling 的参数 schema 来间接实现结构化输出。

7. Tool Calling 为什么有风险？

因为模型可能会：

选错工具
填错参数
漏掉必要参数
把“查询”误判成“修改”
把“取消意向”误判成“立即取消”
引用了用户无权访问的 ticket_id
对高风险操作没有二次确认

所以企业系统里不能这样设计：

LLM -> 直接操作数据库

必须是：

LLM -> ToolPlan -> 后端校验 -> 后端执行

你面试时要强调：模型只负责生成计划，不能直接拥有数据库写权限。所有副作用动作都必须经过后端的工具白名单、参数校验、权限校验、状态校验、确认机制和审计日志。

这句话非常重要。

8. 你项目里的标准安全链路

你可以把你的项目讲成这个流程：

用户：“帮我取消上一单”
↓
Agent 解析“上一单”引用
↓
从 L1 memory 找到 last_ticket_id
↓
Planner 生成 ToolPlan
↓
判断 cancel_ticket 是高风险动作
↓
返回确认问题：“确认取消 Txxx 吗？”
↓
用户确认
↓
后端校验 ticket 属于当前用户 / 租户

↓
校验 ticket 状态是否允许取消
↓
Executor 执行取消
↓
写入 audit_log
↓
返回结果

这比简单回答“我用了 Tool Calling”强很多。

9. Tool Schema 应该怎么设计？

不要让工具太泛。

差的设计：

```
{
  "tool": "execute_sql",
  "args": {
    "sql": "delete from tickets where id = 1"
  }
}
```

这个非常危险，因为模型可以构造任意 SQL。

好的设计：

```
{
  "tool": "cancel_ticket",
  "args": {
    "ticket_id": "T20260617001",
    "reason": "用户主动取消"
  }
}
```

原因：

工具名受控
参数结构受控
业务语义明确
后端可以做权限校验
后端可以做状态校验
方便审计

面试表达：工具设计要尽量业务语义化，不要暴露底层数据库能力。比如不要给 Agent 一个 execute_sql 工具，而是提供 create_ticket、cancel_ticket、add_ticket_comment 这种受控工具。

10. 必填参数缺失怎么办？

比如用户说：

帮我建个工单

这时候缺少标题、问题描述、分类等信息。

错误做法：

模型自己脑补：标题=电脑坏了，分类=硬件

正确做法：

```
{
  "intent": "create_ticket",
  "action": "clarify",
  "missing_fields": ["title", "description"],
  "question": "请补充一下你遇到的问题现象。"
}
```

你项目里对应：

NEED_MORE_INFO

ticket_draft

continue_ticket_draft

面试表达：

对于缺失必填参数的工具调用，不应该让模型编造参数，而应该进入澄清或草稿状态。我的项目里会生成 ticket_draft，等用户补齐字段后再执行创建。

11. 高风险动作怎么办？

比如：

取消工单

删除记录

修改权限

提交审批

发送外部通知

这些动作不能直接执行。

应该进入确认机制：

```
{
  "intent": "cancel_ticket",
  "action": "confirm",
  "confirm_message": "确认取消工单 T20260617001 吗？",
  "pending_action": {
    "tool": "cancel_ticket",
    "args": {
      "ticket_id": "T20260617001"
    }
  }
}
```

你项目可以这样讲：我把取消工单这类动作设计成 pending_action，先记录待确认动作，用户确认后才执行。这样可以避免模型误判导致业务状态被错误修改。

12. Tool Calling 的评测应该测什么？

指标	说明
JSON Parse	能不能解析成 JSON
Schema Valid	是否符合 ToolPlan schema
Required Args	必填参数是否齐全
Intent Accuracy	意图是否识别正确
Tool Accuracy	工具是否选对
Unsafe Action Block	高风险动作是否被拦截
Final Business Accept	最终业务结果是否可接受
Latency	调用耗时
Retry Success	出错后修复是否成功