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8章

智能体开发框架

LangChain

目录

01 概述

02 LangChain的核心组成

03 LangChain的工作流程

04 LangChain的典型应用场景

05 LangChain快速入门

06 LangChain的消息

07 LangChain的提示词

08 LangChain的工具

09 LangChain的状态和持久化

10 案例：课程资料问答助手

Part 01

概述

8.1 概述

LangChain的由来与定位

LangChain在大模型应用开发中的作用

LangChain与直接调用大模型API的对比分析

LangChain与LangGraph的关系

LangChain与RAG的关系

8.1.1 LangChain的由来与定位

1. **大语言模型能力快速提升**，开发者依托其开发智能问答、写作辅助、智能客服、文档检索、代码解释等各类应用

2. **早期 LLM 应用开发方式**：程序向大模型发送提示词，展示模型返回文本结果，该方式简单直观

3. **早期开发模式存在短板**：应用若需调用本地文档、数据库、Web接口等工具，需额外编写大量流程控制代码；应用规模扩大后，提示词构造、上下文拼接、异常处理、状态传递、结果解析等各类问题会不断累积

4. **LangChain 诞生背景**：为解决早期 LLM 应用开发存在的各类问题应运而生

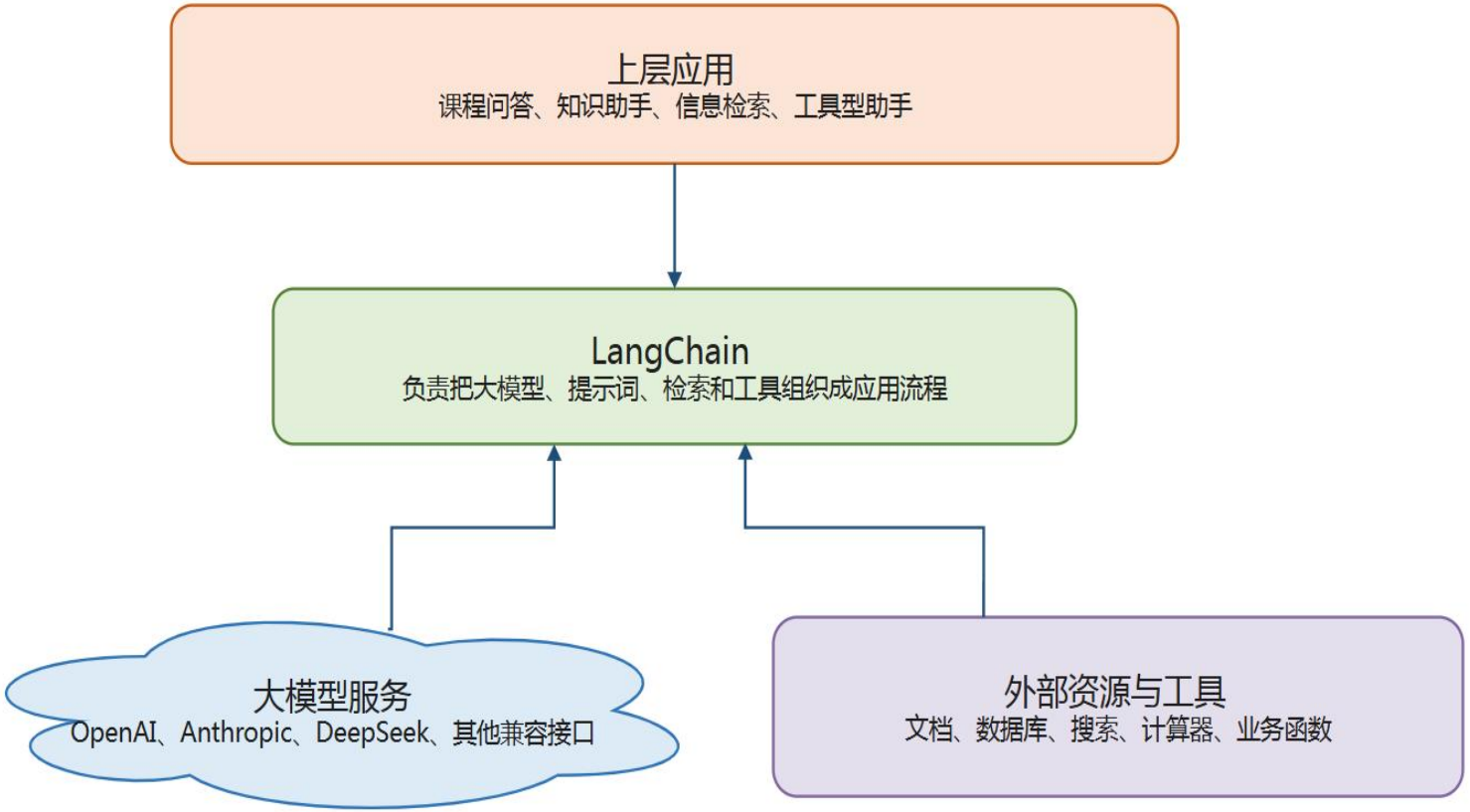
5. **LangChain 定位**：面向大模型应用开发共性需求，提供统一抽象层与可组合模块的 LLM 应用开发框架

6. **LangChain 核心目标**：帮助开发者高效将 LLM 和外部数据、工具、API、记忆系统结合，搭建智能可交互应用（智能问答、对话系统、自动化助手、文档检索分析系统等）

7. **LangChain 极简概括**：作为中间层框架，连接大语言模型与现实世界

8.1.1 LangChain的由来与定位

LangChain的重点是帮助开发者快速构建基于大模型和工具的应用与智能体，并通过标准化接口降低不同大模型提供商之间的切换成本。换言之，LangChain的定位不是替代大模型，而是充当大模型与应用之间的“组织层”。下图展示了LangChain在大模型应用开发体系中的位置，它位于上层应用与底层模型、工具资源之间，承担流程组织与能力编排的作用。



8.1.2 LangChain在大模型应用开发中的作用

LangChain在开发中的首要作用是降低系统组织难度。以一个课程问答系统为例，开发者通常至少需要完成文档读取、文本切分、向量化、相似度检索、提示词模板构造、大模型调用和结果输出等多个环节。如果这些操作全部手工拼接，程序会很快被大量细节填满。LangChain把这些步骤包装成相对独立的组件，使开发者可以把注意力更多地放在业务逻辑而非底层胶水代码上。

LangChain的第二个作用是增强应用的可扩展性。由于模型接口被统一封装，开发者可以在不大幅重写业务代码的情况下切换不同的大模型服务；由于提示词模板、检索器和工具本身是独立模块，当应用需要从“简单问答”升级为“带知识库的问答”或“带工具调用的助手”时，原有代码结构也更容易扩展。这种模块化设计有助于把复杂系统拆解为若干边界清晰的小单元。

8.1.3 LangChain与直接调用大模型API的对比分析

直接调用大模型API的方式并非没有价值。对于非常简单的任务，例如单轮文本生成、短文本改写或固定模板问答，直接调用大模型API往往是最直接的方案。它的优点在于依赖少、程序短、上手快。然而，当应用从简单调用扩展到实际业务流程时，往往很快就会遇到知识受限、输出格式不稳定、业务流程难组织等问题。

与之相比，LangChain的优势在于结构化。它并不一定让每个程序都更短，但能够让程序更清楚地表达“输入如何处理”、“上下文来自哪里”、“模型如何调用”、“输出如何解析”和“何时使用工具”等逻辑。因而，当任务涉及多步骤处理、检索增强生成或智能体式问题求解时，LangChain往往更适合。下表给出了两种开发方式的对比。

对比维度	直接调用大模型API	LangChain
程序入口	通常从一次大模型请求开始	从组件装配和流程组织开始
适用任务	简单生成、单轮问答	多步骤应用、检索增强、工具调用
上下文管理	开发者手工维护	可通过模板、链和检索器组织
模型切换	常需修改调用细节	标准化接口降低切换成本
代码结构	短但容易快速膨胀	初始稍复杂但更利于扩展
使用特点	适合理解大模型调用	适合理解应用系统组织

8.1.4 LangChain与LangGraph的关系

LangChain是面向智能体和大模型应用的高层框架，而LangGraph则是更底层的编排与运行时框架。LangChain提供较容易上手的抽象和预构建能力，适合快速开发；LangGraph则更强调对流程、状态和执行路径的细粒度控制，适合复杂场景下的高级定制。

这里不介绍LangGraph的底层细节（将在第8章介绍），但需要指出，当前LangChain智能体能力并不是脱离底层运行时独立存在的，而是建立在图式执行框架基础之上。这有助于理解现代大模型应用开发中“高层封装”和“底层编排”之间的关系。

如果把大模型应用比作一个完整的软件系统，那么LangChain更像是面向开发者的高层接口层，它负责降低常见应用模式的实现门槛；LangGraph则更像是负责状态流转、节点执行和流程控制的底层调度层。前者强调“如何更快搭起可用应用”，后者强调“如何更精细地控制执行过程”。理解这种分层关系，有助于把“框架易用性”和“系统可控性”放到统一视角下考察。

8.1.5 LangChain与RAG的关系

1	RAG 定义与核心原理	全称为检索增强生成，核心是大语言模型回答问题前先查询外部知识库，结合检索内容生成答案。
2	RAG 两大运行阶段	一是离线建库，将文档切片、转为向量后存入向量数据库；二是在线问答，对用户问题向量化，检索相关文本片段，拼接问题与片段输入大模型生成回答。
3	RAG 核心优势	摆脱大模型固有参数静态知识限制，可调用最新、最相关、私有领域数据，有效降低模型幻觉，提升答案准确性、可追溯性与时效性。
4	RAG 本质分工	检索系统负责精准查找相关资料，生成模型负责梳理语言、输出作答内容。
5	RAG 适用场景	适配企业知识库问答、论文助手、客服系统、法律与医疗文档查询等场景。

8.1.5 LangChain与RAG的关系

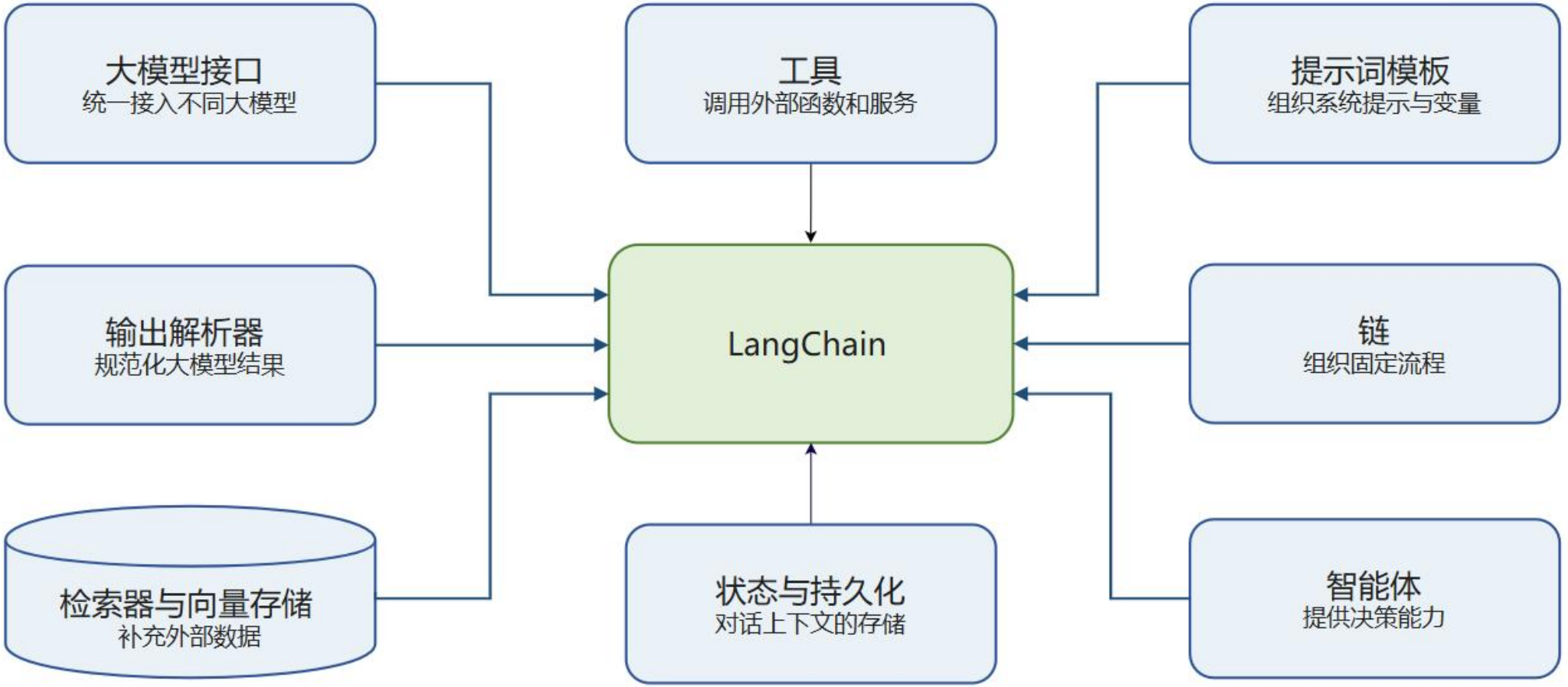
6	LangChain 与 RAG 核心关系	RAG 是解决模型知识静态、上下文有限问题的应用方法论，LangChain 是落地 RAG 的常用工程化工具框架。
7	LangChain 的核心作用	可快速搭建文档处理、向量存储、检索、生成回答的完整 RAG 流程，模块化封装 RAG 体系，支持灵活替换各类组件、接入对话记忆与多步推理，提升项目扩展性。
8	二者从属关键说明	RAG 不依赖 LangChain，可通过原生 Python、向量数据库 SDK、其他框架实现搭建。
9	二者核心定位区别	RAG 解决模型如何调用外部知识的问题，LangChain 解决如何高效将 RAG 能力落地为应用的问题。

Part 02

LangChain的核心组成

8.2 LangChain的核心组成

LangChain的核心思想是把大模型应用拆分为若干可组合模块。不同版本和不同教程的命名方式可能略有变化，但整体上可以把它概括为大模型接口、提示词模板、输出解析器、工具、检索器与向量存储、链、智能体、状态与持久化等几个部分（如图所示）。这些部分之间既相互独立，又可以组合形成完整流程。



8.2 LangChain的核心组成

下表对LangChain核心组件及其作用进行了归纳，有助于从功能角度理解各模块的职责分工。

组件	主要作用
大模型接口	封装不同大模型服务的调用方式
提示词模板	组织系统提示词、上下文和用户问题
输出解析器	把大模型结果变成期望格式
工具	让大模型可调用外部函数或服务
检索器与向量存储	从外部知识源获取相关内容
链	按固定顺序连接多个步骤
智能体	根据任务动态决定是否及如何调用工具
状态与持久化	在链或智能体的多次调用之间持久化状态，实现对话的上下文记忆

8.2 LangChain的核心组成

LangChain各个模块的具体功能如下：

（1）大模型接口。

1. **不同大模型提供商在请求格式、返回结果、参数命名和鉴权方式上往往存在差异。如果应用直接依赖这些差异化接口，那么在更换大模型服务时就可能需要重写大量代码。**

2. **LangChain通过统一的大模型接口，把“向大模型发送输入并获得输出”的过程抽象为比较稳定的编程形式。这样一来，开发者可以更多关注输入提示和输出结果，而不必频繁处理不同平台的适配问题。**

3. **大模型接口通常可以从两个层面来理解。其一是“调用层”，即如何把问题传给大模型；其二是“工程层”，即如何保证未来可以切换大模型而不破坏原有应用。前者强调大模型调用本身，后者强调系统结构的可替换性。**

8.2 LangChain的核心组成

LangChain各个模块的具体功能如下：

（2）提示词模板。

它的作用是把原本散乱的提示词组织成结构化输入。一个完整的大模型请求通常不仅包含用户问题，还包含角色设定、任务说明、回答约束、外部上下文等内容。

如果这些内容全部通过字符串拼接实现，程序既难维护，也容易出错。LangChain的提示词模板机制可以将固定部分和变量部分分离，从而使提示构造更清晰。

8.2 LangChain的核心组成

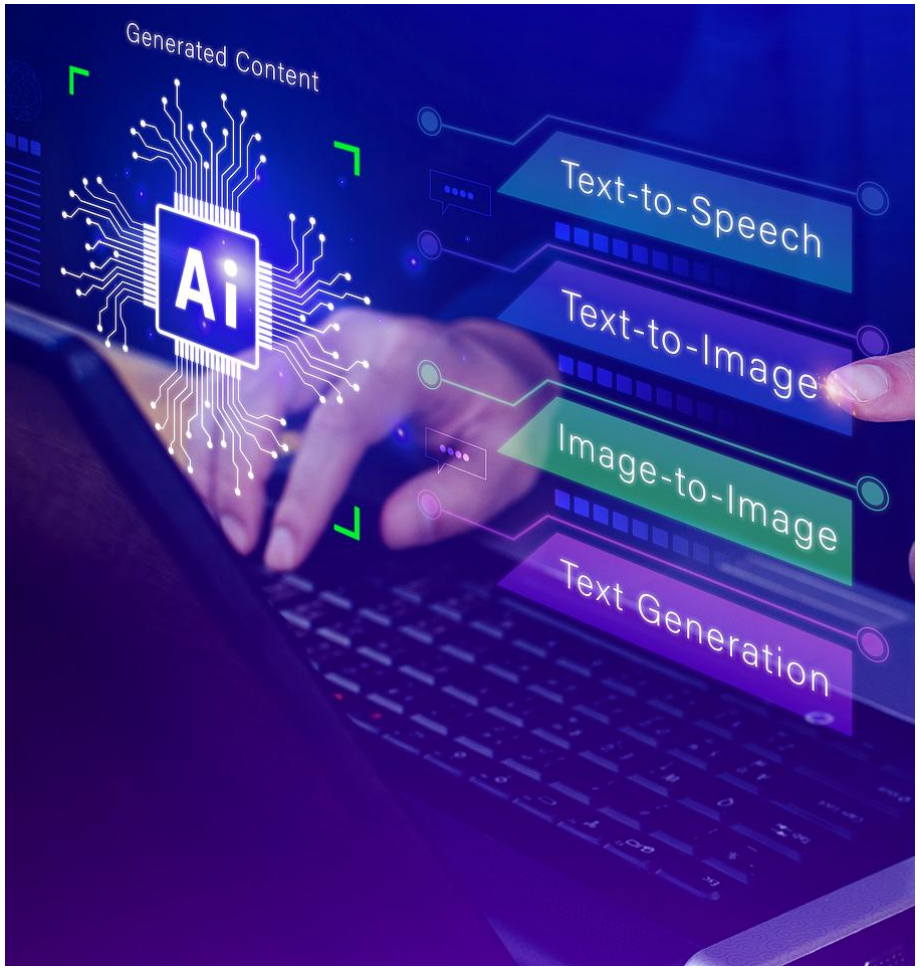
LangChain各个模块的具体功能如下：

（3）输出解析器。

它用于把大模型给出的结果转换成应用需要的形式。对于简单任务，大模型返回自然语言文本即可。

但对于信息抽取、分类、结构化问答等任务，开发者往往希望得到列表、键值对或其他可程序处理的结构。

通过输出解析机制，可以把“大模型生成结果”进一步变成“程序可消费结果”。这也是从“能聊天”走向“能构建应用”的关键一步。



8.2 LangChain的核心组成

LangChain各个模块的具体功能如下：

（4）工具。

大模型虽然具备较强的语言生成能力，但它并不能天然访问程序运行环境之外的资源。

为了让大模型能够查询数据库、调用搜索接口、读取文件或执行计算，就需要为其提供工具。

工具本质上是由开发者定义的外部能力入口，模型根据任务需要决定是否调用某个工具，从而让系统具备与外部世界交互的能力。

8.2 LangChain的核心组成

LangChain各个模块的具体功能如下：

（5）检索器与向量存储。

在知识密集型应用中，检索器和向量存储尤为重要。检索器负责按照用户问题从知识库中挑选相关内容，向量存储负责保存文本片段的向量表示并支持相似度查询，它们构成了检索增强生成的基础。通过这一路径，系统可以在生成回答前先补充与问题相关的外部资料，从而降低模型“凭空编造”的风险。

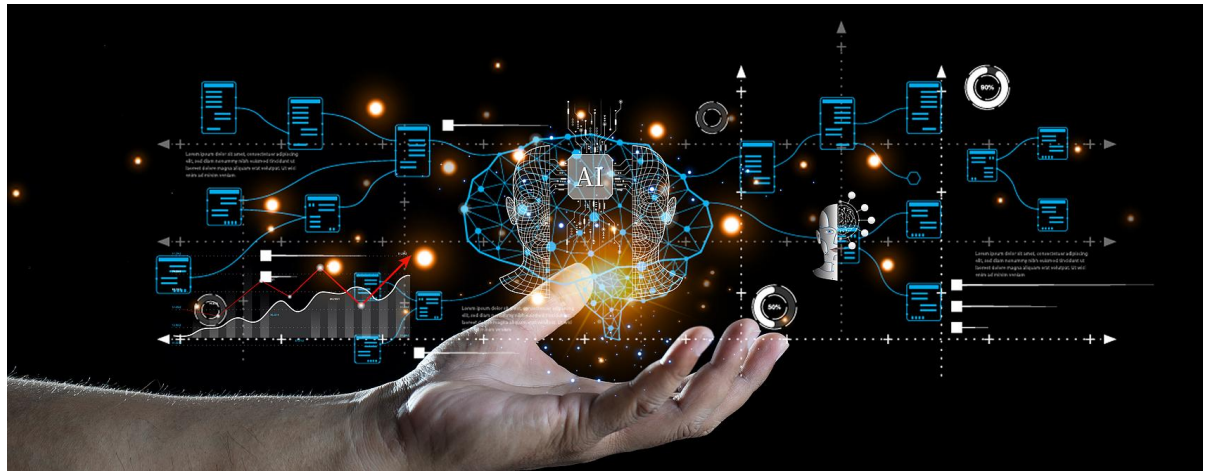
8.2 LangChain的核心组成

LangChain各个模块的具体功能如下：

（6）链。

> **链可以理解为“固定顺序的处理流程”。例如，先把用户问题送入检索器获得相关文档，再把文档与问题一起放入提示词模板，最后交给大模型生成答案。**

> **这种处理顺序在程序设计时已经明确，因此，适合用链来组织。链的特点是路径清晰、执行稳定、行为可预期，尤其适用于检索问答、格式化输出和固定步骤的数据处理任务。**



8.2 LangChain的核心组成

LangChain各个模块的具体功能如下：

（7）智能体。

01 **智能体更强调动态决策。面对复杂问题时，系统不一定事先知道应该先做什么、调用哪个工具，也不一定能在一步之内得到答案。**

02 **智能体的基本思想是让模型在执行过程中不断判断“下一步做什么”，必要时调用工具、读取外部信息、继续推理，直到形成最终结果。**

03 **和链相比，智能体更加灵活，但实现和调试也更复杂。LangChain的一个重要方向是帮助开发者更快速地构建智能体应用，而其智能体运行能力又建立在LangGraph提供的底层执行框架之上。**

04 **因此，在学习时可以先掌握链式组织，再理解智能体的决策循环，这样更符合认知规律。**

8.2 LangChain的核心组成

LangChain各个模块的具体功能如下：

（8）状态与持久化。

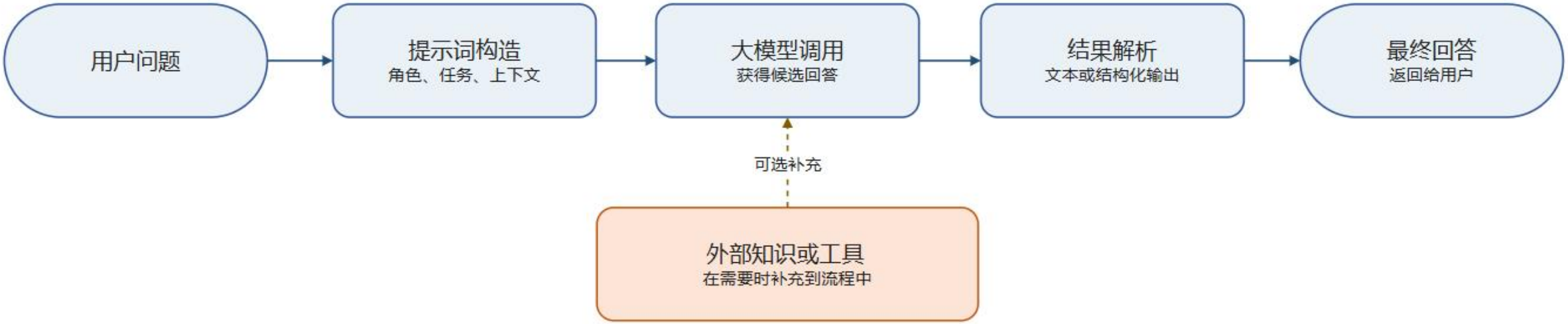
智能体执行过程中需要保存消息历史、工具结果和必要的业务信息。对于同一会话中的短期记忆，可通过线程标识和检查点机制保存状态，使后续调用能够延续先前上下文；对于跨会话的长期记忆，则需要使用独立的存储机制保存用户偏好或长期事实。

Part 03

LangChain的工作流程

8.3 LangChain的工作流程

从系统实现角度看，LangChain的价值不仅体现在“有哪些组件”，更体现在“这些组件如何形成流程”。无论是固定的链式调用，还是动态的智能体调用，LangChain都试图把原本散乱的应用逻辑转化为可描述、可维护、可扩展的运行流程。下图给出了LangChain基本工作流程，从中可以看到用户输入、提示词构造、大模型调用、结果解析与最终输出之间的顺序关系。



8.3 LangChain的工作流程

链式调用流程

智能体调用工具流程

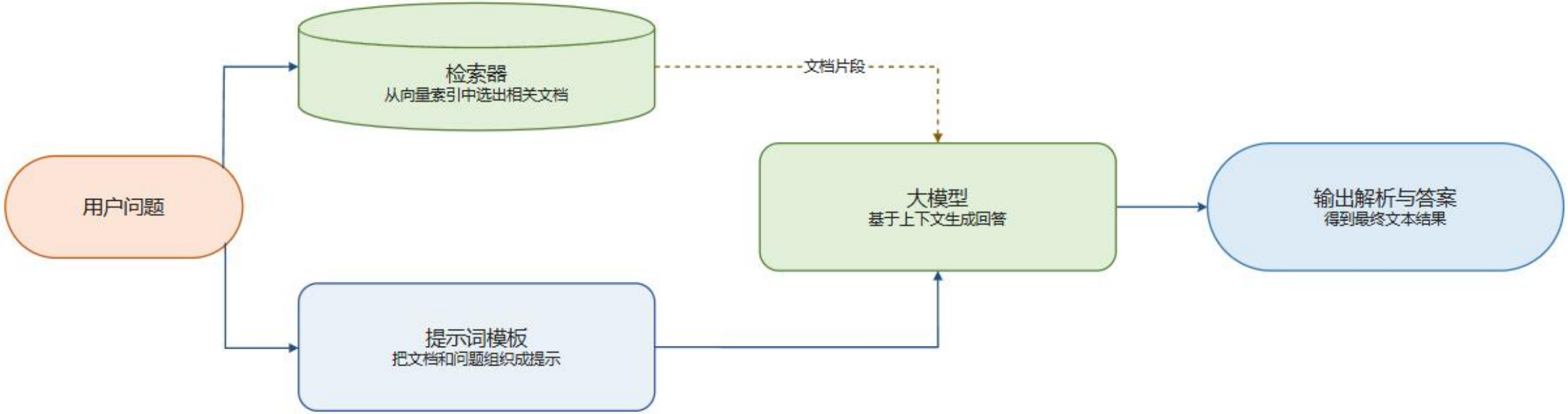
结构化输出与数据校验

Runnable与LCEL的组合执行机制

运行时上下文、状态与调试跟踪

8.3.1 链式调用流程

链式调用通常从用户输入开始，下图展示了一个典型的检索问答链流程。系统首先接收问题，然后根据业务需求构造提示词内容。如果任务涉及外部知识，就先通过检索器获取若干相关文档，再把这些文档片段与用户问题一起注入提示词模板，最后调用大模型生成回答。如果需要进一步对结果进行清洗或结构化处理，还可以在大模型之后接输出解析步骤。



链式调用的优点是执行路径稳定。这种方式更容易理解，也更便于测试和调试。因为，每个步骤的输入输出都比较清楚，开发者能够单独检查“检索结果是否相关”、“提示词模板是否合理”以及“大模型输出是否符合要求”。

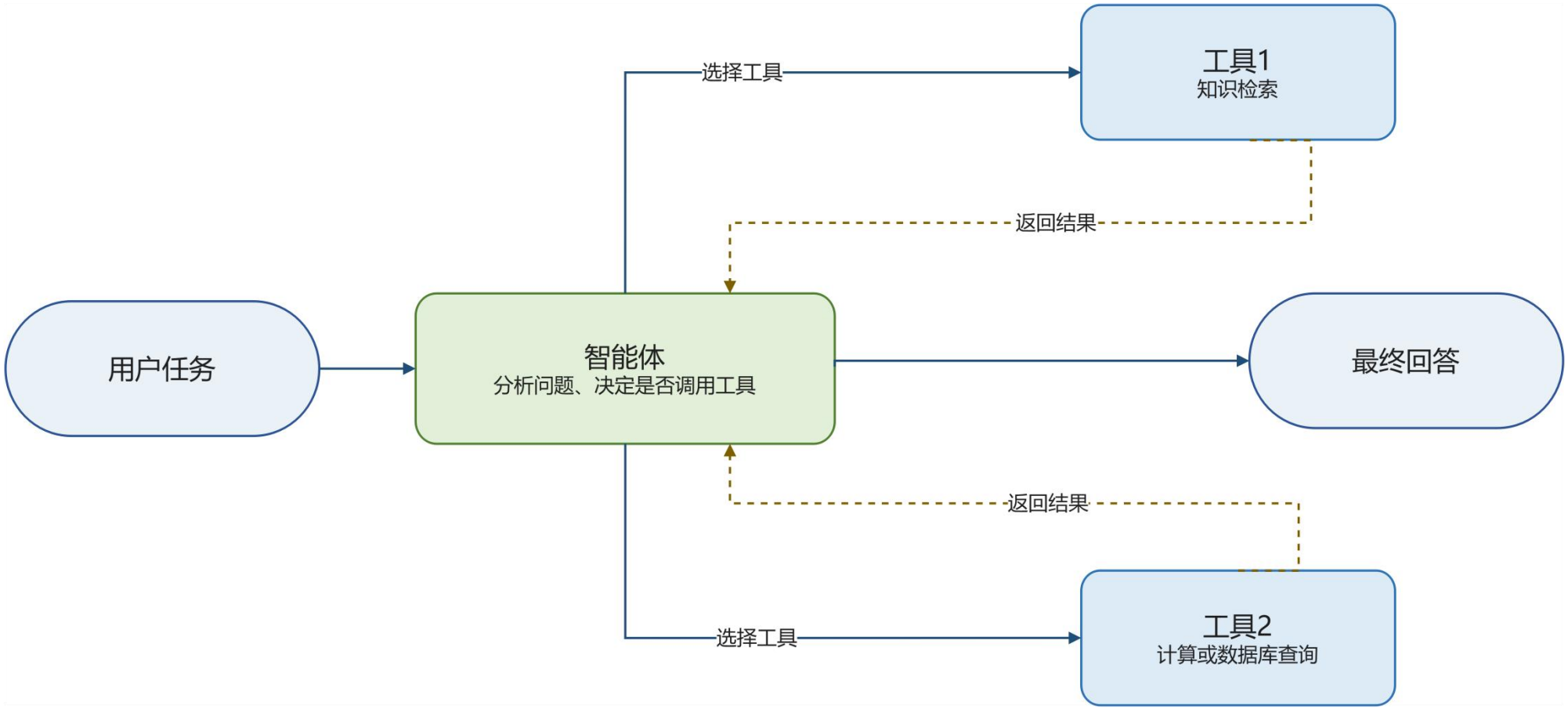
8.3.2 智能体调用工具流程

智能体流程与链式调用的主要区别在于处理路径并不预先完全固定。系统在接收到用户问题后，会先由大模型进行分析，判断当前任务是否需要使用工具。如果只靠大模型自身即可回答，就可以直接给出结果；如果需要额外信息或外部操作，大模型就会选择适当的工具，例如检索知识库、查询网页、执行计算或调用业务函数。

工具执行完成后，其返回结果会再次进入大模型上下文，大模型据此继续判断是否还需要新的工具调用，或者已经能够形成最终回答。这个过程本质上是“分析—调用—观察—再分析”的循环。正因为存在动态决策，智能体往往比固定链更强大，但也更依赖合理的提示词设计、工具定义和执行限制。

8.3.2 智能体调用工具流程

可以把智能体理解为一种“能够自己决定下一步动作的应用程序”，而不必把它神秘化。只要把握住它的核心机制（大模型负责选择动作，工具负责与环境交互），就能对其工作方式形成清晰认识。下图给出了智能体调用工具的基本流程。



8.3.3 结构化输出与数据校验

在许多大模型应用中，系统不仅需要生成一段自然语言回答，还需要把模型输出继续交给程序处理。

例如，课程通知处理程序需要从教师发布的通知中提取通知标题、适用对象、截止时间、需准备材料和建议动作，然后据此生成提醒、写入数据库或触发后续流程。如果模型只是返回一段自由文本，程序就很难稳定地识别其中的关键信息。

> 结构化输出（Structured Output）的作用，是要求模型按照预先规定的数据结构返回结果。开发者可以先定义输出需要包含哪些字段、各字段采用什么数据类型以及哪些字段允许为空，再由LangChain调用模型并把结果转换成可被程序直接使用的数据对象。

> 数据校验（Data Validation）则用于检查返回结果是否满足结构要求，例如截止时间是否为字符串、材料列表是否为列表、必填字段是否缺失等。

因此，结构化输出并不是单纯让答案“排版更整齐”，而是把模型的语言生成结果转化为可验证、可保存、可参与业务流程的数据。它**是大模型应用从“能够回答问题”走向“能够可靠地驱动程序”的重要一步**。

8.3.4 Runnable与LCEL的组合执行机制

在当前LangChain的编程模型中，许多组件都可以被看作Runnable，即“可被调用、可被组合、可被批处理、可被流式输出的执行单元”。大模型、提示词模板、检索器、输出解析器乃至一些自定义函数，都可以在这一统一抽象下组织起来。这样做的意义在于，开发者不必分别记忆每一类对象的独特执行接口，而是可以围绕统一的调用与组合思维来构建应用。

01

LangChain 表达式语言（LangChain Expression Language，简称 LCEL）则是在Runnable基础上形成的组合表达方式。最常见的写法是用管道符把多个步骤按顺序连接起来，例如“提示词模板→大模型→输出解析器”。这种写法的价值不只是代码更短，更重要的是它把数据流向直接呈现在程序结构中（上一步输出什么、下一步接收什么），逻辑关系可以从代码本身读出来。

02

8.3.4 Runnable与LCEL的组合执行机制

LCEL是LangChain框架中的一个核心组件，旨在提供一种简洁、灵活的方式来定义和操作语言模型的工作流。

LCEL允许开发者以声明式的方式构建复杂的大语言模型应用，而无需编写大量的样板代码。

LCEL是一种强大的工作流编排工具，可以从基本组件构建复杂任务链条，并支持诸如流式处理、并行处理和日志记录等开箱即用的功能。

8.3.4 Runnable与LCEL的组合执行机制

LCEL的主要优势包括：

01 声明式语法：LCEL是一种声明式语法，专为LangChain框架设计，旨在通过一种直观的方式定义和组合不同类型的组件链。其核心目标是简化复杂应用程序的开发过程，使开发者能够专注于逻辑而非底层实现细节；

02 异步、批处理和流支持：采用LCEL构建的任何链都将自动、完全地支持同步、异步、批处理和流等能力；

03 错误处理能力：由于大模型的非确定性，使得具备优雅地处理错误的能力变得很重要；

04 并行性：由于大模型应用涉及（有时长期的）API调用，因此，支持并行处理很重要；

05 无缝集成LangSmith：使用LCEL，所有步骤都会自动记录到LangSmith中，可以最大限度的实现可观察性和可调试性；

06 统一的接口：为了支持灵活的组件组合，LCEL 提供了一个统一接口设计方案。该方案允许所有组件共享相同的调用模式，无论是代理、工具还是数据源，均可以通过一致的方法进行交互。这种一致性显著降低了学习成本，并提高了系统的可扩展性和易维护性。

8.3.4 Runnable与LCEL的组合执行机制

从工程角度看，Runnable与LCEL的组合机制还带来几个额外好处。

- 第一，许多链在统一抽象下天然支持同步、异步、批量处理与流式输出，减少了重复封装。
- 第二，当某个步骤需要替换时，开发者通常只需替换链中的局部组件，而不必重写整个应用。
- 第三，链式结构更容易与观测、重试、回退和调试工具结合。

因此，LCEL的价值并不只在于“写法优雅”，而在于它为复杂应用提供了一种更稳定的组装方法。

8.3.5 运行时上下文、状态与调试跟踪

1. LangChain 运行时层面负责信息管理，关注输入输出、上下文、存储、流式写出、执行信息等要素；其中运行时上下文是单次调用程序层面携带的依赖信息（用户标识、数据库连接、配置参数、运行期资源等），并非仅提示词文本上下文，将其纳入运行时可避免依赖散落在全局变量。

2. 状态管理对智能体场景至关重要，智能体依靠多轮循环完成任务，系统需记录执行步骤、中间结果、停止判定条件；智能体运行状态包含消息历史、工具返回结果、执行次数、线程标识等运行信息，智能体本质是有状态运行过程。

3. 运行时视角支撑链与智能体应用的调试、评估工作；简单脚本仅需校验最终答案，链和智能体应用还需查看检索内容、工具调用记录、每轮输入输出合规性、是否出现无效循环等全流程信息。

Part 04

LangChain的典型应用场景

8.4 LangChain的典型应用场景

LangChain的应用场景并不局限于单一方向，只要任务同时涉及大模型调用和流程组织，就可能从中受益。下表归纳了LangChain的典型应用场景及其特点，可用于比较不同场景下更适合采用的框架能力。

应用场景	核心特征	适合采用的LangChain能力
问答系统	用户提出问题并获得解释性回答	提示词模板、大模型接口、输出解析器
检索增强生成	回答必须参考外部知识	文档加载、文本切分、检索器、向量存储
工具调用型助手	需要访问计算器、搜索、数据库等外部工具	工具、智能体
多步骤任务处理	任务包含多个阶段和中间状态	链、智能体、流程编排

8.4.1 问答系统

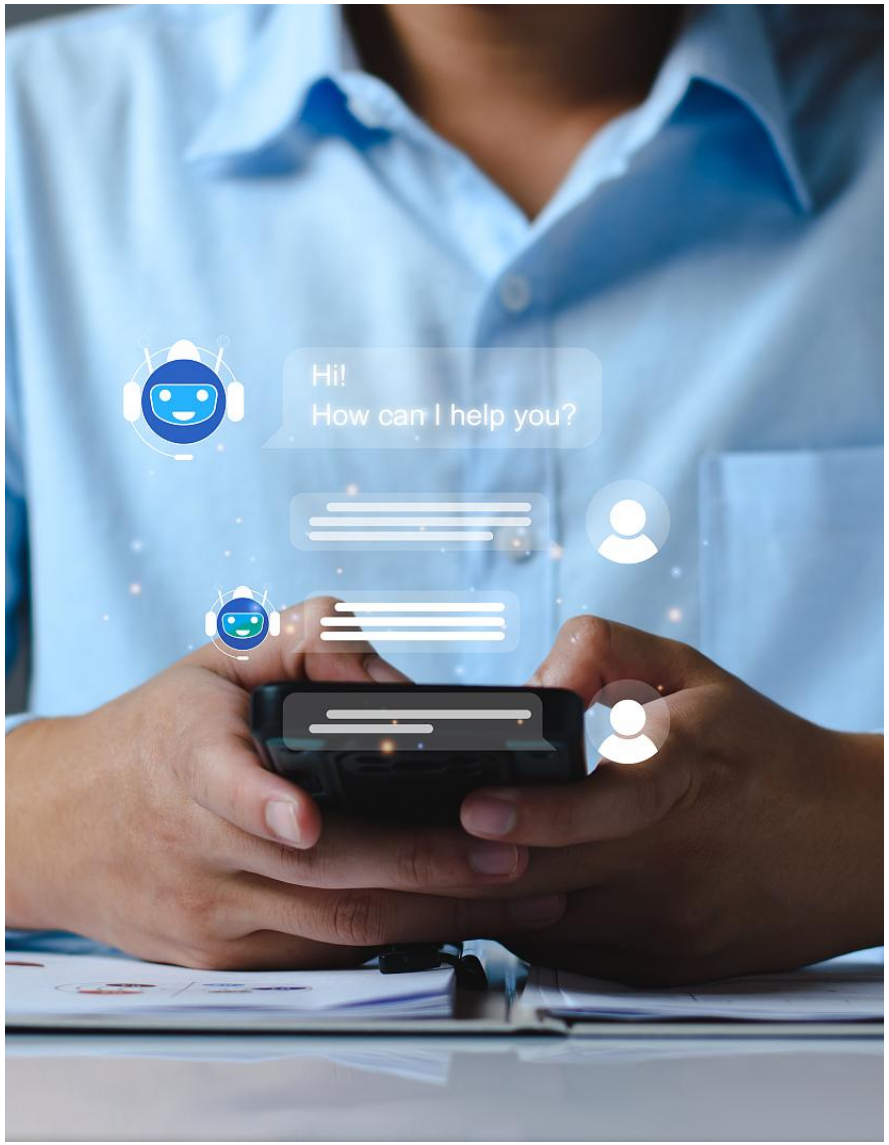
问答系统属于基础、易理解的大模型应用场景，运行流程为用户提问→依托提示词模板组装模型输入→大模型生成答案。

纯大模型问答系统无外部知识支撑，回答仅依靠模型训练存量知识，适配通用解释、写作辅助、概念问答类任务。

LangChain 在问答场景的核心价值不在于使用复杂框架，而是统一整合提示词组织、大模型切换、输出处理三类通用需求。

单轮问答场景仅需 LangChain 链式写法即可满足需求；多轮上下文、工具查询、来源约束类拓展问答，才需启用 LangChain 复杂能力。

问答系统是学习、理解 LangChain 最自然的入门场景，但无法完整展现 LangChain 的全部功能价值。



8.4.2 检索增强生成

RAG 核心定义与价值

检索增强生成（RAG）是大模型应用重要方向，核心思路为回答问题前先检索相关资料，再将资料作为上下文交付大模型，可显著提升文档问答、企业知识库问答、法规问答等场景回答的针对性。

RAG 主流架构分类

依据 LangChain 官方文档，RAG 架构可分为 2-Step RAG 与 Agentic RAG 两类核心思路，核心区别为前者是固定流程，后者是动态决策流程。

2-Step RAG 核心特性

遵循 “检索总是发生在生成之前” 的固定流程，始终先检索、后生成；具备流程简单、执行稳定、响应延迟可预测的特点，易实现、易测试与维护。

2-Step RAG 适用场景

适配知识来源明确、检索步骤稳定、强调效率的场景，包括 FAQ 问答、课程资料问答、教材辅助问答、制度文档检索、文档机器人等。

8.4.2 检索增强生成

Agentic RAG 核心特性

将检索作为可调用工具交由智能体，由大模型在推理过程中自主判断是否检索、何时检索、检索次数，核心是大模型驱动智能体动态决策检索行为；优势为灵活性高，适配多轮查找、多工具调用的复杂任务，无需预设固定流程；短板是执行路径不稳定、响应时延波动大、系统控制性较弱。

Agentic RAG 适用场景

适配研究助手、综合信息分析、多工具问答等复杂任务，对应无需固定检索、可能需要多次检索的问答场景。

两类 RAG 核心差异总结

2-Step RAG 主打固定流程，核心优势是结构清晰、流程可控；Agentic RAG 主打动态推理决策，核心优势是灵活性强、适配复杂任务。

学习与设计进阶逻辑

入门优先掌握 2-Step RAG，可清晰掌握检索器、提示词模板、大模型调用、输出解析器的顺序关系；在此基础上学习 Agentic RAG，可理解大模型系统从固定链向智能体的演进路径。

8.4.3 工具调用型助手

当系统不仅需要“说”，还需要“做”时，就进入了工具调用型助手的场景。例如，课程助手除了回答概念问题，还可能需要调用计算器完成分数统计，或访问日历服务查询实验安排。LangChain中的工具机制使大模型能够把“语言理解”与“外部操作”结合起来，从而构成更实用的应用程序。

此类场景下，LangChain的重点不再只是组织文本上下文，而是建立“大模型如何受控地接触外部世界”的边界。开发者需要明确哪些能力可以被暴露为工具、工具输入输出采用何种格式、失败时如何处理、大模型在什么条件下应停止继续调用。也就是说，工具调用型助手虽然功能更强，但设计难点不在于多写几个函数，而在于约束大模型决策空间，使系统既有行动能力，又保持可控。

8.4.4 多步骤任务处理

一些任务并不能通过一次大模型调用完成，例如生成报告、完成信息汇总、分阶段撰写内容或对复杂问题逐步求解。此时，开发者可以用链把固定步骤串联起来，或者用智能体让系统自主决定下一步操作。多步骤任务处理体现了LangChain在流程组织上的优势，也说明大模型应用开发已经从“单次对话”走向“任务执行”。

与单轮问答相比，多步骤任务处理更接近传统软件工程中的工作流设计。它要求开发者思考阶段划分、中间结果表示、错误恢复和最终输出整合等问题。LangChain在这里的价值，恰恰是把这些步骤表达为可读、可拆分、可验证的流程，而不是把所有逻辑塞进一次冗长提示词中。因此，多步骤任务是连接“简单链式应用”和“复杂智能体系统”的中间层，也是非常值得展开说明的一类场景。

Part 05

LangChain快速入门

8.5 LangChain快速入门

环境准备

第一个LangChain程序

一个简易的智能体实例

8.5.1 环境准备

LangChain应用程序需要调用大模型，可以调用云端大模型，也可以调用本地大模型，这里采用Ollama提供的本地大模型服务（如何在本地计算机上安装Ollama可以参考教材官网）。

Ollama负责在本机启动大模型服务并暴露HTTP接口，LangChain通过ChatOllama与OllamaEmbeddings完成文本生成和向量化调用。**典型的本地服务地址为http://127.0.0.1:11434。**

若本机已经通过ollama pull下载所需大模型，应用代码只需指定大模型名称和服务地址即可运行。这里统一采用聊天大模型gemma4:e4b（具体安装方法可以参考教材官网），**向量模型采用nomic-embed-text**（具体安装方法可以参考教材官网）。

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。

8.5.1 环境准备

除了在Python虚拟环境中安装LangChain核心包之外，开发者还需要根据所选大模型提供商、文档加载方式和向量检索方式安装附加包。下表给出了本章案例所使用的主要依赖包，可以使用Python的pip工具安装这些依赖包。

依赖包	作用说明
langchain	LangChain核心框架
langchain-ollama	Ollama大模型接口适配
langchain-community	社区扩展组件，如向量存储等
langchain-text-splitters	文本切分工具
faiss-cpu	本地向量索引与相似度检索

8.5.1 环境准备

推荐使用如下命令构建Python虚拟环境并安装依赖包：

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

8.5.2 第一个LangChain程序

最基础的LangChain程序通常只包含大模型接口和一次调用。下面的示例使用ChatOllama创建大模型对象，然后调用invoke方法完成一次简单问答。虽然这个示例还没有体现链和智能体的完整优势，但它展示了LangChain在“统一大模型接口”层面的基本写法。具体代码如下：

```
# langchain_call_model.py
from langchain_ollama import ChatOllama
model = ChatOllama(
    model="gemma4:e4b",
    temperature=0,
    base_url="http://127.0.0.1:11434"
)
response = model.invoke("请用一句话解释什么是检索增强生成。")
print(response.content)
```

8.5.2 第一个LangChain程序

可以在Trae中运行该代码，程序成功运行后的结果类似如下：

```
检索增强生成（RAG）可以理解为：
**在让大型语言模型（LLM）生成答案之前，先让它去外部的、可靠的知识库中搜
索相关信息，然后根据搜索到的最新资料来生成答案，从而确保答案的准确性和时效性。**
```

需要注意的是，程序运行以后，因为需要调用大模型，大模型会“思考”一段时间以后才会给出结果，所以，需要等待一段时间，程序运行才会出现结果。

8.5.2 第一个LangChain程序

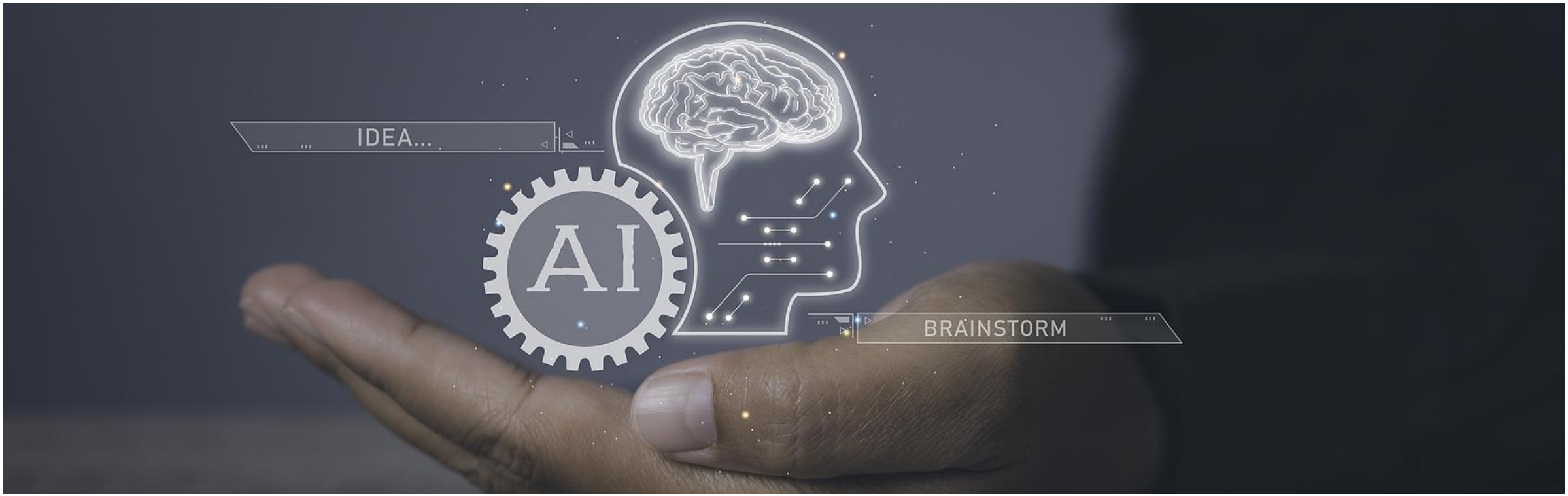
如果采用云端大模型，比如阿里的千问大模型，则可以采用类似如下代码：

```
# langchain_cloud_qwen.py
from langchain_community.chat_models import ChatTongyi

model = ChatTongyi(
    model="qwen-turbo",
    temperature=0,
    api_key="你的阿里云API_KEY"
)

response = model.invoke("请用一句话解释什么是检索增强生成。")
print(response.content)
```

8.5.3 一个简易的智能体实例

开发者可以通过create_agent方便地构建一个具备工具调用能力的简易智能体。与固定链不同，智能体会根据问题内容决定是否需要调用工具，因此，更适合处理“先判断、再行动”的任务。例如，一个课程助手除了可以解释概念，还可以调用查询函数查找课程通知、实验安排和作业要求。

需要注意的是，智能体并不意味着系统“完全自主”。更准确地说，开发者仍然负责定义系统目标、可用工具和执行边界，大模型只是在这些边界内决定下一步动作。



8.5.3 一个简易的智能体实例

下面给出了一个简化的课程通知查询智能体示例：大模型根据用户问题决定是否调用课程通知查询函数，具体代码如下：

```
# langchain_simple_agent.py
from langchain.agents import create_agent
from langchain_ollama import ChatOllama

def search_notice(topic: str) -> str:
    """
    查询课程相关通知信息
    根据输入的主题关键词，返回对应的课程通知内容
    Args:
        topic: 要查询的通知主题（如：实验报告、课程答疑）
    Returns:
        对应的通知详情字符串
    """
    notices = {
        "实验报告": "实验报告需在本周五18:00前提交到教学平台。",
        "课程答疑": "课程答疑安排在周三下午 4 点，地点为实验楼302。",
    }
    for keyword, notice in notices.items():
        if keyword in topic:
            return notice
    return notices.get(topic, "暂未查询到相关通知。")
```

8.5.3 一个简易的智能体实例

```
def build_agent():
    model = ChatOllama(
        model="gemma4:e4b",
        temperature=0,
        base_url="http://127.0.0.1:11434",
    )
    return create_agent(
        model=model,
        tools=[search_notice],
        system_prompt="你是一名课程助理，必要时调用工具查询课程通知。",
    )
```

8.5.3 一个简易的智能体实例

```
def main() -> None:
    agent = build_agent()
    result = agent.invoke(
        {"messages": [{"role": "user", "content": "请帮我查询实验报告的提交要求。"}]}
    )
    print(result)

if __name__ == "__main__":
    main()
```

Part 06

LangChain的消息

8.6 LangChain的消息

消息组件的核心定位

核心消息类型

消息实操

消息在智能体中的流转

8.6.1 消息组件的核心定位

消息是LangChain中实现多轮对话、上下文交互的核心组件

1. 消息的本质是封装了”发送者身份 + 消息内容”的结构化数据，用于在模型、用户、智能体之间传递信息。

2. LangChain将消息分为不同类型，每种类型对应不同的交互场景。

3. 理解消息的本质后，关键是区分“谁在说话”和“这条消息承担什么职责”。不同消息类型正是为这种角色划分服务的。

8.6.2 核心消息类型

LangChain中常见消息可以按发送者和用途分为四类。阅读时可以把它们对应到一次对话流程中：用户提出问题、系统设定规则、模型生成回复、工具返回结果，如表所示。

消息类型	说明	典型场景
HumanMessage	用户发送的消息	用户提问、指令输入
AIMessage	AI/智能体发送的消息	模型回答、工具调用结果
SystemMessage	系统级引导消息	定义智能体角色、行为规范
ToolMessage	工具调用的相关信息	工具参数、工具返回结果

8.6.3 消息实操

消息类型只有放到真实调用中才容易理解。这里先从普通多轮对话入手，观察消息列表如何帮助模型理解上下文。

以下示例演示多轮对话中消息的创建与使用，场景模拟了教师与助手的连续提问。具体代码如下：

```
# langchain_messages_demo.py
from langchain_ollama import ChatOllama
from dotenv import load_dotenv
import os

load_dotenv()

llm = ChatOllama(
    model=os.getenv("OLLAMA_MODEL", "gemma4:e4b"),
    base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),
    temperature=0.7
)

# 构建多轮对话消息列表
messages = [
    {"role": "system", "content": "你是一个专业的Python教学助手，回答简洁明了。"},
    {"role": "user", "content": "什么是装饰器？"},
    {"role": "assistant", "content": "装饰器是Python中用于修改函数或类行为的特殊语法。"},
    {"role": "user", "content": "请举例说明"}
]

response = llm.invoke(messages)
print(f"AI响应: {response.content}")
```

8.6.4 消息在智能体中的流转

普通对话只涉及用户与模型之间的消息传递。智能体场景更进一步：模型可能先发出工具调用指令，工具执行后再把结果返回给模型，因此，需要观察更完整的消息流转过程。

智能体的一条典型交互链路如下：

简单问答

用户HumanMessage →模型分析→ 直接回复AIMessage。

工具调用

用户HumanMessage →模型判断需调用工具→生成含ToolCall的AIMessage →工具执行产生ToolMessage →模型整合后输出AIMessage。

8.6.4 消息在智能体中的流转

以下示例完整演示四种消息类型在一次工具调用中的流转过程：

```
# langchain_agent_messages_demo.py
from dotenv import load_dotenv
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage,
ToolMessage
from langchain_core.tools import tool
from langchain_ollama import ChatOllama
import os

load_dotenv()
```

8.6.4 消息在智能体中的流转

```
@tool
def calculate(a: float, b: float, operator: str) -> str:
    """
    执行基础数学计算

    参数:
        a: 第一个数字
        b: 第二个数字
        operator: 运算符，支持 "+"、"-"、"*"、"/"
    """
    if operator == "+":
        return str(a + b)
    elif operator == "-":
        return str(a - b)
    elif operator == "*":
        return str(a * b)
    elif operator == "/" and b != 0:
        return str(a / b)
    return "不支持的操作"
```

8.6.4 消息在智能体中的流转

```
llm = ChatOllama(
    model=os.getenv("OLLAMA_MODEL", "gemma4:e4b"),
    base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),
    temperature=0.7
)
llm_with_tools = llm.bind_tools([calculate])

# ── 步骤 1：用户输入封装为 HumanMessage ──────────────────────────
messages = [
    SystemMessage(content="你是计算助手，使用 calculate 工具完成计算。"),
    HumanMessage(content="计算 15 加 27 等于多少"),
]
print(f"步骤1 消息数: {len(messages)}，类型: {[type(m).__name__ for m in messages]}")
```

8.6.4 消息在智能体中的流转

```
# ── 步骤 2：模型判断需要调用工具，返回含 tool_calls 的 AIMessage ──
ai_response = llm_with_tools.invoke(messages)
print(f"\n步骤2 模型回复类型: {type(ai_response).__name__}")
print(f"  工具调用指令: {ai_response.tool_calls}")

# ── 步骤 3：执行工具，结果封装为 ToolMessage ──────────────────
tool_call = ai_response.tool_calls[0]
tool_result = calculate.invoke(tool_call["args"])
tool_message = ToolMessage(content=tool_result, tool_call_id=tool_call["id"])
print(f"\n步骤3 工具执行结果: {tool_message.content}")

# ── 步骤 4：把完整历史交给模型，生成自然语言最终回复 ──────────
messages_full = messages + [ai_response, tool_message]
final_response = llm_with_tools.invoke(messages_full)
print(f"\n步骤4 最终回复: {final_response.content}")
print(f"\n完整消息链类型: {[type(m).__name__ for m in messages_full]}")
```

Part 07

LangChain的提示词

8.7 LangChain的提示词

- 提示词的核心价值
- 提示词的设计原则
- 提示词模板
- LCEL：把提示词串进链式调用

8.7.1 提示词的核心价值

提示词（Prompt）是连接开发者意图与大语言模型的核心桥梁。在LangChain中，提示词并非简单的文本输入，而是结合模型特性、任务需求设计的结构化指令，其质量直接决定模型的响应效果。

提示词的核心价值体现在：

精准控制输出

通过指令明确告知模型期望的输出格式与内容

引导模型行为

定义智能体的角色、能力边界、响应规范

注入上下文信息

将相关背景、限制条件融入提示词

明确提示词的重要性后，还需要把经验性的写法整理成可执行的设计原则，否则，提示词很容易变成随手写出的自然语言描述。

8.7.2 提示词的设计原则

提示词设计中最容易出问题的地方，通常集中在目标、上下文、格式和示例四个方面。读者可先看“反例”为什么模糊，再看“正例”如何降低模型误解的概率，如表所示。

原则	说明	反例	正例
指令清晰	明确告知任务目标	“处理一下”	“将以下文本总结为3个要点，每点不超过20字”
上下文完整	提供必要背景信息	直接提问	“基于上文提到的LangChain框架，说明...”
输出规范	指定输出格式	“写点什么”	“以JSON格式输出，包含title和content字段”
示例引导	通过示例说明期望	无示例	“例如：输入‘北京天气’输出‘晴，20-25℃’”

8.7.3 提示词模板

如果每次调用都手写完整提示词，代码会很快变得重复且难维护。**LangChain提供了提示词模板，就是为了把固定指令和动态变量分离开。**

提示词模板本质上跟我们平时使用的邮件模板、短信模板没什么区别，就是一个字符串模板，模板可以包含一组模板参数，通过模板参数值可以替换模板对应的参数。一个提示词模板可以包含下面内容：

发给大模型的指令	一组问答示例，以提醒AI以什么格式返回请求	发给大模型的问题

提示模板在LangChain中的作用是定义一个结构化的输入格式，该格式包含了大模型在生成输出时所需的所有信息。通过使用提示模板，开发者可以确保向大模型提供的输入是清晰、一致且易于理解的，从而提高大模型的输出质量。

8.7.3 提示词模板

■ 字符串提示词模板

在LangChain中，可以使用 PromptTemplate类创建简单的提示词。提示词模板可以内嵌任意数量的模板参数，然后通过参数值格式化模板内容。下面是一个代码实例：

```
# langchain_string_prompt.py
from langchain_core.prompts import PromptTemplate
# 定义一个提示模板，包含adjective和content两个模板变量，模板变量使用{}包括起来
prompt_template = PromptTemplate.from_template(
    "给我讲一个关于{content}的{adjective}故事。"
)
# 通过模板参数格式化提示模板
result = prompt_template.format(adjective="童话", content="一千零一夜")
print(result)
```

8.7.3 提示词模板

■ 聊天消息提示词模板

聊天消息提示词模板ChatPromptTemplate是智能体开发中最常用的模板形式，适用于多轮对话、有明确角色划分的场景。每条消息带有角色标签（system、human、ai），能够更准确地表达对话的结构和意图。

聊天模型（Chat Model）以聊天消息列表作为输入，这个聊天消息列表的消息内容也可以通过提示词模板进行管理。这些聊天消息与原始字符串不同，因为每个消息都与“角色(role)”相关联。

在OpenAI的聊天模型中，给不同的聊天消息定义了三种角色类型分别是助手(assistant)、人类（human）或系统（system）角色，具体如下：

助手(assistant)

助手(assistant) 消息指的是当前消息是AI回答的内容

用户（user）

用户（user）消息指的是你发给AI的内容

系统（system）

系统（system）消息通常是用来给AI身份进行描述

8.7.3 提示词模板

■ 聊天消息提示词模板

下面是一个关于聊天消息提示词模板的具体代码实例：

```
# langchain_chat_prompt.py
from langchain_core.prompts import ChatPromptTemplate
chat_template = ChatPromptTemplate.from_messages(
    [
        ("system", "你是一位人工智能助手，你的名字是{name}。"),
        ("human", "你能做什么"),
        ("ai", "设计课程，谢谢！"),
        ("human", "{user_input}"),
    ]
)
messages = chat_template.format_messages(name="教学助手",
                                         user_input="你的名字叫什么？")
print(messages)
```

8.7.3 提示词模板

■ MessagesPlaceholder

MessagesPlaceholder提示词模板负责在特定位置添加消息列表。在上面的 ChatPromptTemplate中，我们看到了如何格式化两条消息，每条消息都是一个字符串。如果我们希望用户传入一个消息列表，就可以通过MessagesPlaceholder的方式将其插入到特定位置。下面是一个具体代码实例：

```
# langchain_messagesplaceholder_prompt.py
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.prompts import MessagesPlaceholder
from langchain_core.messages import HumanMessage
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant"),
    #可以传入一组消息
    MessagesPlaceholder("msgs")
])
result = prompt_template.invoke({"msgs": [HumanMessage(content="您好!"),
                                          HumanMessage(content="langchain!")]})
print(result)
```

8.7.4 LCEL：把提示词串进链式调用

学完ChatPromptTemplate，可以自然引出 LCEL（LangChain Expression Language），它用管道符“|”把提示词、模型、输出解析器串成一条“流水线”，是LangChain推荐的现代写法。

这里对比两种不同的写法：

```
# 传统写法：手动分三步
messages = chat_prompt.format_messages(role="Python工程师", question="装饰器")
response = llm.invoke(messages)
result = response.content          # 手动取 .content

# LCEL写法：用管道符“|”串成链，一步完成
from langchain_core.output_parsers import StrOutputParser
chain = chat_prompt | llm | StrOutputParser()
result = chain.invoke({"role": "Python工程师", "question": "装饰器"})
```

8.7.4 LCEL：把提示词串进链式调用

两段代码效果完全一样，LCEL版的优势在于：省去手动取 .content，并且同一条链支持四种调用方式。同一条LCEL链可以根据运行场景采用不同调用方式。选择方法时，先判断任务是单次、流式、批量还是异步，再选择对应接口，如表所示：

方法	用途	示例
chain.invoke()	单次调用，返回完整结果	chain.invoke({"role": "..."})
chain.stream()	流式输出，逐字打印	for chunk in chain.stream(...)
chain.batch()	批量处理多条输入	chain.batch([{...}, {...}])
chain.ainvoke()	异步调用	await chain.ainvoke(...)

8.7.4 LCEL：把提示词串进链式调用

以下示例演示流式输出，在需要“打字机效果”的界面中常用：

```
# langchain_stream_demo.py
from dotenv import load_dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_ollama import ChatOllama
import os

load_dotenv()

llm = ChatOllama(
    model=os.getenv("OLLAMA_MODEL", "gemma4:e4b"),
    base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),
    temperature=0.7
)
```

8.7.4 LCEL：把提示词串进链式调用

```
chain = ChatPromptTemplate.from_messages([
    ("system", "你是一个简洁的学习助手，回答不超过100字。"),
    ("human", "{question}")
]) | llm | StrOutputParser()

# 流式输出：逐块打印，不等待全部生成完成
print("助手：", end="", flush=True)
for chunk in chain.stream({"question": "什么是递归？用一句话解释"}):
    print(chunk, end="", flush=True)
print()
```

在Trae中运行该程序，会看到字符像打字机效果一样被连续动态输出，屏幕上会一个字一个字被打印出来。

何时用LCEL，何时不用呢？简单的单次调用，llm.invoke()直接写就很清晰。当你需要流式输出、批量处理、或者把多个步骤串成一条可复用的链时，LCEL是更好的选择。

Part 08

LangChain的工具

8.8 LangChain的工具

- 工具的核心定位
- 工具的分类
- 自定义工具实操
- 工具在智能体中的调用
- MCP：标准化的工具与上下文接入协议

8.8.1 工具的核心定位

工具（Tool）是LangChain中扩展智能体能力边界的核心组件。大语言模型在精确计算、实时数据查询、外部系统交互等方面存在天然局限，工具组件通过标准化接口封装外部功能，实现“模型决策+工具执行”的协同模式。从能力边界看，凡是模型自身不擅长、但程序可以确定完成的任务，都适合被封装为工具，典型能力扩展方向如表所示。

能力扩展	示例
精确计算	数学运算、日期计算
实时查询	天气查询、股价查询
外部交互	数据库操作、API调用
本地操作	文件读写、命令执行

不同工具背后的复杂度不同：有的只是本地函数，有的需要访问外部服务，有的会操作本地文件或系统资源。先分类，再实现，能帮助读者判断工具的风险和适用边界。

8.8.2 工具的分类

工具还可以按照依赖的资源类型进一步分类。分类的目的不是制造新概念，而是帮助读者评估工具实现的复杂度和课堂复现难度，如表所示。

类型	说明	典型示例
基础工具	本地计算，无需外部依赖	计算器、日期工具
外部服务工具	需对接第三方API	天气API、搜索API
本地工具	操作本地环境	文件读写、命令执行

8.8.3 自定义工具实操

在 LangChain中，自定义工具的开发遵循“装饰器+文档字符串+类型注解”的三要素原则：

@tool 装饰器	将普通函数转换为LangChain可识别的工具对象。
文档字符串	描述工具的功能与参数，模型读取这些信息以理解工具用途。
类型注解	帮助LangChain进行参数验证和转换。

8.8.3 自定义工具实操

1.基础计算工具

以下示例演示如何封装一个支持四则运算的计算器工具，并通过invoke方法验证正常与异常两种情况。

```
# langchain_tool_calculator.py
from langchain_core.tools import tool
from typing import Union

@tool
def calculate(a: Union[int, float], b: Union[int, float], operator: str) -> str:
    """
    执行基础数学计算，支持加减乘除。

    参数:
        a: 第一个数字
        b: 第二个数字
        operator: 运算符，可选值: "+", "-", "*", "/"

    返回:
        计算结果字符串，格式: "结果: {数值}"

    示例:
        calculate(10, 5, "+") → "结果: 15"
    """
```

8.8.3 自定义工具实操

1.基础计算工具

```
    try:
        if operator == "+":
            result = a + b
        elif operator == "-":
            result = a - b
        elif operator == "*":
            result = a * b
        elif operator == "/":
            if b == 0:
                return "错误：除数不能为零"
            result = a / b
        else:
            return f"错误：不支持的运算符 '{operator}'"
        return f"结果: {result}"
    except Exception as e:
        return f"错误：{str(e)}"

print(calculate.invoke({"a": 10, "b": 5, "operator": "+"}))
print(calculate.invoke({"a": 10, "b": 0, "operator": "/"}))
```

8.8.3 自定义工具实操

2.日期计算工具

以下示例演示如何封装日期加减运算工具，同时兼容YYYY-MM-DD和YYYY/MM/DD 两种常见日期格式。

```
# langchain_date_tool.py
from langchain_core.tools import tool
from datetime import datetime, timedelta
from typing import Union

@tool
def calculate_date(date_str: str, days: int) -> str:
    """
    计算指定日期加减天数后的日期。
    参数:
        date_str: 原始日期，格式支持 YYYY-MM-DD 或 YYYY/MM/DD
        days: 要加减的天数（正数加，负数减）
    返回:
        计算后的日期字符串，格式："YYYY-MM-DD"
    示例:
        calculate_date("2026-01-01", 10) → "2026-01-11"
        calculate_date("2026-01-01", -5) → "2025-12-27"
    """
```

8.8.3 自定义工具实操

2.日期计算工具

```
formats = ["%Y-%m-%d", "%Y/%m/%d"]
    for fmt in formats:
        try:
            original = datetime.strptime(date_str, fmt)
            target = original + timedelta(days=days)
            return target.strftime("%Y-%m-%d")
        except ValueError:
            continue
    return f"错误：日期格式无效，请使用 YYYY-MM-DD 或 YYYY/MM/DD"

print(calculate_date.invoke({"date_str": "2026-04-18", "days": 7}))
print(calculate_date.invoke({"date_str": "2026/04/18", "days": -30}))
```

8.8.3 自定义工具实操

3.文件写入工具

以下示例演示文件写入工具的实现，包括目录自动创建、追加/覆盖两种写入模式，以及权限异常的友好处理：

```
# langchain_file_tool.py
from langchain_core.tools import tool
from pathlib import Path

@tool
def write_to_file(file_path: str, content: str, mode: str = "append") -> str:
    """
    将文本内容写入本地文件。
    参数:
        file_path: 文件路径（支持相对路径和绝对路径）
        content: 要写入的文本内容
        mode: 写入模式，"append"（追加）或 "overwrite"（覆盖）
    返回:
        操作结果字符串
    示例:
        write_to_file("test.txt", "Hello", "overwrite") → "成功写入test.txt"
    """
```

8.8.3 自定义工具实操

3.文件写入工具

```
    try:
        path = Path(file_path)
        path.parent.mkdir(parents=True, exist_ok=True)  # 自动创建缺失的目录层级

        write_mode = "a" if mode == "append" else "w"
        with open(file_path, write_mode, encoding="utf-8") as f:
            f.write(content + "\n")

        return f"成功：{'追加写入' if mode == 'append' else '覆盖写入'}{file_path}"
    except PermissionError:
        return f"错误：权限不足，无法写入{file_path}"
    except Exception as e:
        return f"错误：{str(e)}"

print(write_to_file.invoke({
    "file_path": "output/test.txt",
    "content": "LangChain工具测试",
    "mode": "overwrite"
}))
```

8.8.4 工具在智能体中的调用

前面三个示例只是验证“工具本身能运行”。真正的智能体还需要让模型根据用户问题判断是否调用工具，并把工具结果组织成自然语言回复。

这里给出一个包含工具调用的最小可运行智能体示例。虽然该智能体只有一个工具，但已经具备“理解问题、决定是否调用工具、返回最终答案”的基本能力。具体代码如下：

```
# langchain_agent_tool.py
from langchain.agents import create_agent
from langchain.tools import tool
from langchain_ollama import ChatOllama
SYSTEM_PROMPT = """
你是高校《AI编程和智能体开发》课程助教。
请准确回答学生问题；涉及分数计算时必须调用工具。
"""
```

8.8.4 工具在智能体中的调用

```
@tool
def calculate_final_score(attendance: float, lab: float, exam: float) -> str:
    """按照总评 = 平时考勤*0.1 + 实验*0.4 + 期末考试*0.5 计算成绩。"""
    total = attendance * 0.1 + lab * 0.4 + exam * 0.5
return f"根据课程规则，总评成绩为 {total:.2f} 分。"

def build_agent():
    model = ChatOllama(
        model="gemma4:e4b",
        temperature=0,
        base_url="http://127.0.0.1:11434",
    )
    return create_agent(
        model=model,
        tools=[calculate_final_score],
        system_prompt=SYSTEM_PROMPT,
)
```

8.8.4 工具在智能体中的调用

```
agent = build_agent()
response = agent.invoke(
    {
        "messages": [
            {
                "role": "user",
                "content": "我的考勤 95，实验 88，期末 90，请计算总评。"
            }
        ]
    }
)
print(response["messages"][-1].content)
```

8.8.5 MCP：标准化的工具与上下文接入协议

前面已经介绍了LangChain中的工具分类。无论是计算器、日期处理，还是文件写入工具，它们都有一个共同特点：工具通常由应用开发者在当前项目中直接定义，然后交给智能体调用。这种方式便于学习和快速实验，但在真实系统中会遇到新的问题：如果同一个数据库查询能力、文件检索能力或课程管理系统接口，需要同时被多个智能体应用、多个IDE助手或多个桌面客户端使用，难道每个应用都重新写一套工具封装吗?

MCP（Model Context Protocol，模型上下文协议）正是为了解决这类问题而出现的开放协议。可以把MCP理解为“大模型应用连接外部能力的一种标准接口”。它并不替代LangChain，也不替代大模型本身，而是规定外部系统如何把工具、资源和提示词等能力以统一方式暴露出来。这样，智能体应用就不必关心某个能力背后到底是本地函数、数据库、文件系统，还是远程业务服务，只需要通过协议发现和调用这些能力。

8.8.5 MCP：标准化的工具与上下文接入协议

从LangChain的角度看，MCP最直接的价值在于扩展工具来源。传统写法是在Python代码中使用@tool装饰器定义工具；采用MCP之后，工具可以运行在独立的MCP服务器中，再由LangChain客户端把这些工具加载为LangChain可识别的Tool对象。换句话说，MCP服务器负责“提供能力”，LangChain智能体负责“根据任务选择和调用能力”。这使工具的开发、复用和权限控制都更容易工程化。

8.8.5 MCP：标准化的工具与上下文接入协议

下表给出了本地自定义工具与MCP工具接入方式的对比

比较维度	本地自定义工具	MCP工具接入
工具位置	通常写在当前LangChain项目代码中	通常由独立MCP server暴露，可以被多个客户端复用
适用场景	适合课堂实验、单个应用和简单业务函数	适合企业系统、跨应用工具生态、IDE助手和多客户端复用
接入方式	使用@tool装饰器、类型注解和文档字符串定义	通过MCP协议发现工具，再由适配器转换为LangChain工具
主要优势	实现直接，调试简单，依赖少	接口标准化，工具与应用解耦，便于复用和治理
主要代价	工具容易和应用代码耦合，复用成本较高	需要理解MCP服务器、传输方式和客户端配置

8.8.5 MCP：标准化的工具与上下文接入协议

■ MCP的基本组成

为了理解MCP与LangChain工具的关系，可以先把MCP拆成三个角色。第一，MCP服务器是能力提供方，负责把某个外部系统中的工具、资源或提示词暴露出来。例如，一个课程资料服务器可以提供“查询课程通知”“读取实验要求”“获取课程大纲”等能力。第二，MCP客户端是能力使用方，负责连接MCP服务器、发现可用能力并发起调用。第三，智能体框架负责把这些能力纳入自己的执行流程，让模型在合适的时候选择工具。

MCP服务器通常可以暴露三类内容。工具（tools）表示可执行动作，如查询通知、写入文件、调用业务接口；资源（resources）表示可读取的上下文材料，如课程大纲、文档片段、配置文件；提示词（prompts）表示可复用的提示模板。对初学者而言，最容易落地的是工具，因为它们与前面学习过的LangChain工具概念最接近。理解工具以后，再学习资源和提示词会更自然。

8.8.5 MCP：标准化的工具与上下文接入协议

■ MCP工具服务器示例

下面给出一个简化的课程通知MCP服务器。它把“课程通知查询”封装成一个MCP工具。为了便于学习者复现，示例使用内存字典模拟课程通知数据；在真实系统中，工具函数内部可以改为查询数据库、调用教务系统接口或读取文件。运行示例前，需要在当前Python环境中安装MCP SDK、LangChain MCP适配器以及本章前面使用过的LangChain和模型接入包，具体安装命令如下：

```
# 安装依赖示例
pip install mcp langchain-mcp-adapters
```

8.8.5 MCP：标准化的工具与上下文接入协议

■ MCP工具服务器示例

构建课程通知MCP服务器的代码如下：

```
# mcp_course_server.py
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("course-assistant")
COURSE_NOTICES = {
    "实验报告": "实验报告需在本周五22:00前提交，文件名格式为：学号-姓名-实验4.docx。",
    "课程答疑": "本周三19:30在线答疑，主题为LangChain工具调用与RAG案例。",
    "期末复习": "期末复习资料将在第16周发布，重点包括RAG、工具调用和智能体状态管理。"
}

@mcp.tool()
```

8.8.5 MCP：标准化的工具与上下文接入协议

■ MCP工具服务器示例

```
def query_course_notice(topic: str) -> str:
    """
    根据主题查询课程通知。
    参数:
        topic: 通知主题，例如“实验报告”“课程答疑”或“期末复习”
    返回:
        与主题相关的课程通知；若没有命中，则返回可查询主题列表。
    """
    if topic in COURSE_NOTICES:
        return COURSE_NOTICES[topic]
    topics = "、".join(COURSE_NOTICES.keys())
    return f"未找到主题“{topic}”。当前可查询主题包括：{topics}。"

if __name__ == "__main__":
    mcp.run(transport="stdio")
```

8.8.5 MCP：标准化的工具与上下文接入协议

■ MCP工具服务器示例

这段代码中：

01 FastMCP用于创建一个MCP服务器。

02 @mcp.tool()把普通Python函数注册为MCP工具。

03 函数的名称、参数类型和文档字符串会成为客户端理解工具能力的重要信息。

04 这里的设计与LangChain本地工具很相似，都是通过“函数签名+说明文字”告诉模型工具能做什么。区别在于，该工具现在不直接写在LangChain应用内部，而是由一个独立服务器对外提供。

需要注意的是，这段代码不要运行，后面将要介绍的代码文件langchain_use_mcp_tool.py运行时，会自动运行python mcp_course_server.py。

8.8.5 MCP：标准化的工具与上下文接入协议

■ 在LangChain中调用MCP工具

有了MCP服务器以后，LangChain应用可以通过MCP适配器加载工具。下面的示例演示如何连接上面的课程通知服务器，并把MCP工具交给LangChain智能体使用。为了与本章前面的示例保持一致，这里仍以本地Ollama模型为例；如果读者使用其他大模型服务，只需要替换模型初始化部分。具体代码如下：

```
# langchain_use_mcp_tool.py
import asyncio

from langchain.agents import create_agent
from langchain_mcp_adapters.client import MultiServerMCPClient
from langchain_ollama import ChatOllama

async def main():
    client = MultiServerMCPClient({
        "course": {
            "command": "python",
            "args": ["mcp_course_server.py"],
            "transport": "stdio",
        }
    })
```

8.8.5 MCP：标准化的工具与上下文接入协议

■ 在LangChain中调用MCP工具

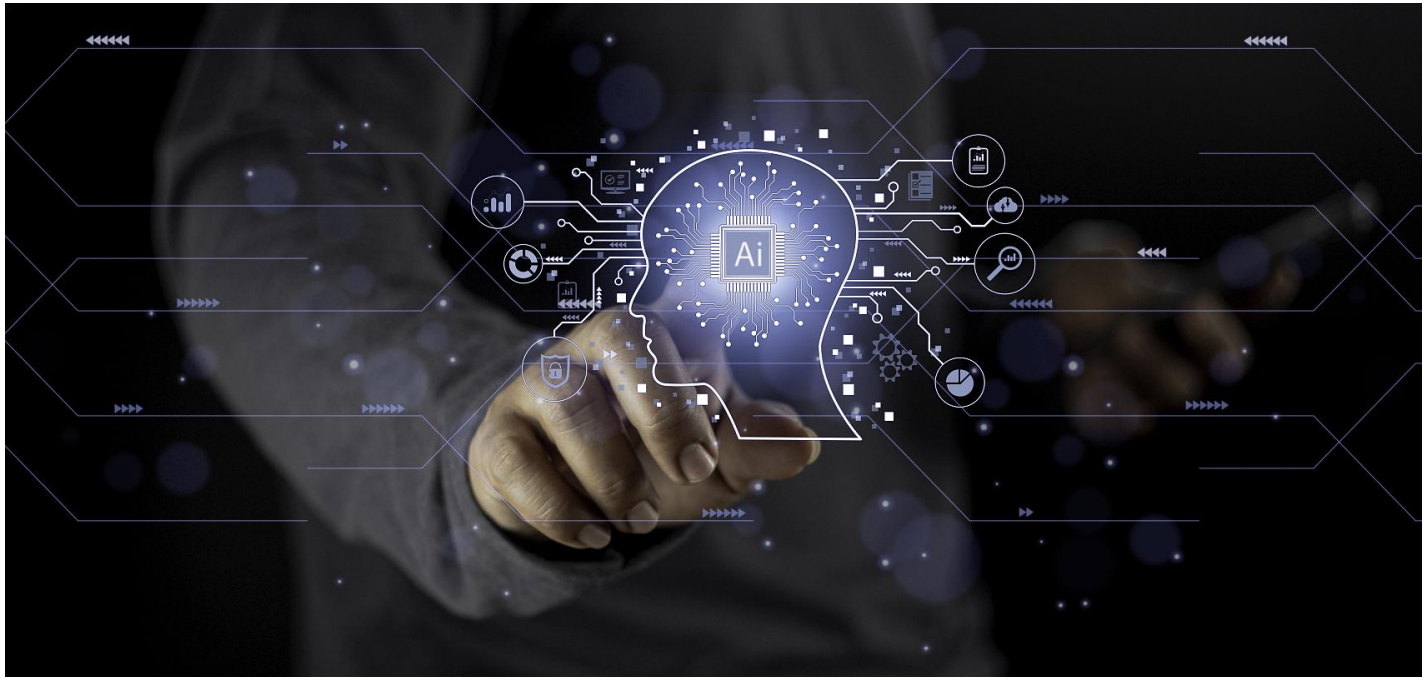
```
    tools = await client.get_tools()
    print("已加载工具：", [tool.name for tool in tools])

    model = ChatOllama(model="gemma4:e4b")
    agent = create_agent(model=model, tools=tools)

    result = await agent.ainvoke({
        "messages": [
            {
                "role": "user",
                "content": "请查询实验报告的提交要求，并用一句话提醒学生。"
            }
        ]
    })
    print(result["messages"][-1].content)

if __name__ == "__main__":
    asyncio.run(main())
```

8.8.5 MCP：标准化的工具与上下文接入协议

■ 在LangChain中调用MCP

从这个例子可以看出，MCP并没有改变智能体“模型决策+工具执行”的基本机制。模型仍然负责理解用户问题、判断是否需要工具以及组织最终回答；工具仍然负责执行确定性操作。MCP改变的是工具的提供方式：工具不再必须由当前应用本地定义，而可以由独立服务器标准化提供。

需要说明的是，这个例子只是简单演示了对本地自定义MCP服务器的调用，实际上，还可以支持对各类第三方MCP服务器的调用，比如调用百度天气MCP服务器或者百度地图MCP服务器等。



8.8.5 MCP：标准化的工具与上下文接入协议

■ MCP与上下文接入

MCP名称中包含Context，说明它关注的不只是“调用函数”，也关注“给模型提供上下文”。在智能体应用中，上下文既可能来自用户输入和历史消息，也可能来自外部文件、数据库、知识库或业务系统。若每个应用都用自己的方式读取这些上下文，系统之间就很难复用。MCP通过统一协议描述外部资源，使不同AI应用能够以较一致的方式发现和读取上下文材料。

不过，在学习顺序上，不建议一开始就把MCP、RAG、向量数据库和智能体状态全部混在一起。更清晰的路径是：先掌握LangChain本地工具，理解工具描述如何影响模型选择；再学习MCP如何把工具从本地代码中抽离出来；最后再把MCP资源、RAG检索和长期记忆等能力组合进更完整的智能体系统。这样既能看到MCP的工程价值，也不会把它误解成另一个“大模型框架”。

8.8.5 MCP：标准化的工具与上下文接入协议

■ 使用MCP时的设计建议

第一，只有具有复用价值的能力才值得优先做成MCP服务器。简单计算器工具直接写成本地工具即可；需要被多个应用共同使用的课程资料查询、文件检索、数据库访问和企业系统接口，更适合通过MCP暴露。

第二，MCP工具的说明文字仍然非常重要。模型选择工具时会读取工具名称、参数和文档字符串。如果说明含糊，例如只写“查询信息”，模型很难判断何时调用；如果说明明确写出“根据主题查询课程通知”，模型就更容易把用户问题映射到正确工具。

第三，要注意安全边界。MCP让外部能力更容易接入智能体，也意味着工具权限需要被认真管理。凡是涉及文件写入、数据库修改、网络请求或系统命令的工具，都应限制参数范围、记录调用日志，并在必要时加入人工确认机制。工具越接近真实业务系统，越不能只关注“能不能调用”，还要关注“是否应该调用”和“调用后如何追踪”。

总的来说，MCP可以看作LangChain工具机制的工程化延伸。LangChain负责组织智能体的推理和执行流程，MCP负责让外部工具与上下文以标准方式进入这个流程。对于初学者而言，先理解本地工具，再理解MCP工具，是从单应用开发走向工具生态开发的一条自然路径。

Part 09

LangChain的状态和持久化

8.9 LangChain的状态和持久化

前面介绍消息、提示词和工具时，主要关注的是一次调用内部的信息组织。但在真实应用中，用户往往不会只问一个问题，智能体也不会只执行一步操作。例如，学生先询问“本周实验内容是什么”，随后又追问“需要提交哪些文件”，系统必须知道第二个问题中的“本周实验”指的是前一次对话中已经提到的内容。这就引出了LangChain中的状态和持久化问题。

早期LangChain版本中常用Memory机制来保存对话历史，例如ConversationBufferMemory和ConversationSummaryMemory等。但这些旧式Memory更像是“把聊天历史组织进提示词”的组件，主要服务于早期Chain API。到了LangChain 1.x之后，智能体状态管理逐渐转向采用LangGraph的checkpoint体系，因为智能体不仅要记住聊天内容，还要保存工具调用、中断位置、节点状态和恢复信息。

在较新的LangChain版本中，开发者不再把“记忆”简单理解为自动拼接聊天记录，而是需要区分短期状态、检查点持久化和长期记忆三类问题。短期状态负责让同一会话能够连续对话；检查点持久化负责把运行过程保存下来，以便程序重启后继续使用；长期记忆则更接近业务数据，例如用户偏好、课程身份、常用资料等，需要按业务规则保存和读取。

8.9 LangChain的状态和持久化

从Memory到状态管理

短期状态：让同一会话能够连续对话

持久化检查点：让程序重启后仍能延续状态

长期记忆：不要把所有业务信息都放进聊天历史

状态和持久化的使用建议

8.9.1 从Memory到状态管理

旧版 LangChain Memory

机制本质是将聊天历史塞入提示词，简单聊天程序可正常使用，但无法适配智能体场景；智能体除聊天记录外，还需留存工具调用结果、执行中间步骤、线程标识、执行断点信息。

新版 LangChain

核心引入运行状态（state）、检查点（checkpoint）两大概念。

运行状态（state）

指应用单次运行全程需留存的全部信息；基础内容为用户消息、模型回复、工具消息构成的消息历史；智能体场景下还包含已调用工具、工具返回数据、是否产出最终答案等信息。

检查点（checkpoint）

是运行状态的存储结果，等同于智能体执行流程在某一时刻的快照，支撑流程断点续跑。

核心可控能力

状态、检查点机制优化后划分边界更清晰，支持两类核心可控能力：①开发者可手动指定会话线程标识，限定上下文仅在对应线程内延续；②可切换检查点存储方案，开发环境用内存存储，线上部署用数据库存储。

新旧机制核心差异

旧版是框架自动隐性存储信息；新版实现显式状态管理，程序可明确管控存储内容、存储位置、数据读取时机。

8.9.2 短期状态：让同一会话能够连续对话

短期状态主要解决同一用户在同一会话中的上下文延续问题。在LangChain智能体中，通常通过检查点保存器和thread_id共同完成这一点。检查点保存器负责保存状态，thread_id用于区分不同会话。只要两次调用使用相同的thread_id，后一次调用就可以读取前一次调用留下的消息历史。

下面的示例使用InMemorySaver在内存中保存状态。它适合演示和本地调试，但程序退出后状态会丢失。具体代码如下：

```
# langchain_short_term_state_demo.py
from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_ollama import ChatOllama
from langgraph.checkpoint.memory import InMemorySaver
import os


load_dotenv()
```

8.9.2 短期状态：让同一会话能够连续对话

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

# InMemorySaver把状态保存在当前Python进程的内存中。
checkpointer = InMemorySaver()
agent = create_agent(
    model=llm,
    tools=[],
    checkpointer=checkpointer,
    system_prompt="你是一个课程学习助手，回答要简洁准确。",
)
```

8.9.2 短期状态：让同一会话能够连续对话

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

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

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

8.9.2 短期状态：让同一会话能够连续对话

第二次调用中，用户并没有再次说明自己正在学习什么，模型仍然能够回答，原因在于两次调用使用了相同的thread_id。如果把第二次调用中的thread_id改为student-002，系统会把它当作另一个会话，也就不能从student-001的消息历史中读取信息。这说明状态不是全局共享的，而是通过线程标识进行隔离的。

01

需要注意的是，短期状态并不等于把所有历史内容无限制地塞进提示词。对话越长，上下文窗口压力越大，模型也越容易被无关历史干扰。实际系统通常需要对历史消息进行裁剪、摘要或按任务选择性保留，而不是机械保存所有内容。

02

8.9.3 持久化检查点：让程序重启后仍能延续状态

内存保存器的优点是简单，但它只能在当前进程中工作。如果程序关闭、服务器重启或部署到多进程环境中，内存中的状态就会丢失。因此，实际应用通常需要把检查点保存到数据库或其他持久化存储中。对于入门示例，可以使用SQLite保存本地检查点；在生产系统中，则通常会选择PostgreSQL等更适合并发访问的数据库。

使用SQLite检查点前，需要安装额外依赖。可以在虚拟环境中执行如下命令：

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

8.9.3 持久化检查点：让程序重启后仍能延续状态

下面的示例把检查点写入本地SQLite数据库文件。第一次运行程序时，用户告诉助手自己的课程；再次运行程序时，只要使用相同的数据库文件和thread_id，助手就可以继续读取先前保存的会话状态。具体代码如下：

```
# langchain_persistent_state_demo.py
from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_ollama import ChatOllama
from langgraph.checkpoint.sqlite import SqliteSaver
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,
)
```

8.9.3 持久化检查点：让程序重启后仍能延续状态

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

with SqliteSaver.from_conn_string("langchain_state.db") as checkpointer:
    agent = create_agent(
        model=llm,
        tools=[],
        checkpointer=checkpointer,
        system_prompt="你是一个课程学习助手，回答要简洁准确。",
    )

    result = agent.invoke(
        {"messages": [{"role": "user", "content": "我这学期选了人工智能导论。"}]},
        config=config,
    )
    print(result["messages"][-1].content)

    result = agent.invoke(
        {"messages": [{"role": "user", "content": "请提醒我刚才提到的课程名称。"}]},
        config=config,
    )
    print(result["messages"][-1].content)
```

8.9.3 持久化检查点：让程序重启后仍能延续状态

与InMemorySaver相比，SQLite保存器的关键区别在于状态被写入磁盘文件。因此，程序重新启动以后，只要继续使用同一个数据库文件和同一个thread_id，就可以恢复对应线程的上下文。这就是持久化检查点的基本作用：它保存的不是某一句回答文本，而是应用在某个线程中的运行状态。

在设计持久化状态时，还需要注意线程标识的选择。thread_id不应随意写成固定字符串，否则多个用户可能共享同一段历史。实际系统通常会把用户ID、会话ID或任务ID组合成线程标识，从而保证不同用户、不同会话之间的状态互不干扰。

8.9.4 长期记忆：不要把所有业务信息都放进聊天历史

短期状态解决的是“这轮会话刚刚发生了什么”，长期记忆解决的则是“系统长期应该知道什么”。二者不能混为一谈。例如，学生的姓名、常用编程语言、所在课程、作业提交偏好等信息，更适合保存在数据库或用户画像表中，而不是长期堆在对话历史里。这样做既便于更新，也便于权限控制和数据清理。

一种常见做法是：把稳定的用户信息保存在业务存储中，每次调用智能体前只取出与当前任务相关的少量信息，放入系统提示词或运行时上下文。

8.9.4 长期记忆：不要把所有业务信息都放进聊天历史

下面用一个简化的字典模拟长期用户信息，说明长期记忆如何与短期会话状态配合使用：

```
# langchain_profile_context_demo.py
from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_ollama import ChatOllama
from langgraph.checkpoint.memory import InMemorySaver
import os

load_dotenv()

user_profiles = {
    "student-001": {
        "name": "小林",
        "course": "人工智能导论",
        "level": "大学本科二年级",
    }
}
```

8.9.4 长期记忆：不要把所有业务信息都放进聊天历史

```
def build_system_prompt(user_id: str) -> str:
    profile = user_profiles.get(user_id, {})
    return (
        "你是一个课程学习助手。"
        f"当前学生姓名：{profile.get('name', '未知')}；"
        f"课程：{profile.get('course', '未知')}；"
        f"学习阶段：{profile.get('level', '未知')}。"
        "回答时结合这些背景，但不要编造未提供的信息。"
    )

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

user_id = "student-001"
```

8.9.4 长期记忆：不要把所有业务信息都放进聊天历史

```
agent = create_agent(
    model=llm,
    tools=[],
    checkpointer=InMemorySaver(),
    system_prompt=build_system_prompt(user_id),
)

result = agent.invoke(
    {"messages": [{"role": "user", "content": "请给我一个适合当前课程的复习建议。"}]},
    config={"configurable": {"thread_id": user_id}},
)
print(result["messages"][-1].content)
```

8.9.5 状态和持久化的使用建议

学习LangChain的状态机制时，可以把握以下几点。

第一，单轮问答通常不需要状态；只有当后续输入依赖前文时，才需要保存消息历史。

第二，课堂实验和临时脚本可以使用InMemorySaver，便于理解机制；需要跨程序运行保存上下文时，应使用SQLite、PostgreSQL等持久化检查点。

第三，thread_id是状态隔离的关键，真实系统必须为不同用户或会话设置不同标识。

第四，长期记忆不应简单等同于长聊天记录，而应作为业务数据单独建模、单独保存。

概括来说，旧版Memory机制强调“让模型记住对话”，而当前LangChain中的状态和持久化更强调“让应用可控地保存和恢复运行过程”。这种变化体现了大模型应用从简单聊天程序走向工程化智能体系统的趋势。开发者只有明确区分消息历史、检查点和长期业务数据，才能构建既能连续对话、又便于调试和部署的LangChain应用。

Part 10

案例：课程资料问答助手

8.10 案例：课程资料问答助手

- 案例目标与功能说明
- 案例总体架构
- 数据准备与文档切分
- 向量索引构建
- 检索问答链实现
- 运行结果与效果分析

8.10.1案例目标与功能说明

本案例的目标是说明一个基于LangChain的知识问答系统是如何搭建起来的。系统不追求复杂的人机交互界面，而是强调后端处理流程，包括文档加载、文本切分、向量化、相似度检索、上下文拼接和答案生成等环节。通过这一案例，可以把“组件”与“流程”两个层面的知识统一起来。

从功能上看，该系统至少应具备以下能力：

01 一是能够读取课程资料并建立本地知识索引。

02 二是能够根据用户问题检索与之最相关的文档片段。

03 三是能够将检索内容与用户问题一起送入大模型，从而生成更有针对性的回答。

04 四是能够在结果分析部分说明该系统的局限性，例如回答质量依赖检索效果、大模型仍可能误解问题等。

8.10.2案例总体架构

课程资料问答助手的总体架构可以分为离线构建和在线问答两个阶段。

01

离线构建阶段

系统把课程文本读入内存，按照一定长度切分为多个片段，并通过嵌入模型生成向量表示后写入向量索引。

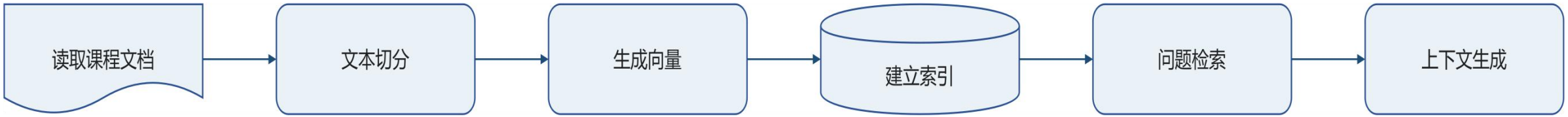
02

在线问答阶段

系统接收用户问题，先在向量索引中检索相关片段，再把这些片段作为上下文交给大模型生成回答。

8.10.3数据准备与文档切分

文档准备是知识问答系统的第一步。若直接把整份课程文档一次性交给大模型，不仅会带来上下文长度压力，也不利于精确检索。因此，通常需要先将文档划分为多个较小片段，再分别建立向量表示。切分长度并不是越短越好，因为过短会导致语义信息被破坏；也不是越长越好，因为过长会使检索粒度变粗。这一过程本质上是在上下文完整性和检索精度之间寻找平衡。下图展示了课程资料问答助手中的文档处理与检索流程，从中可以看出文本切分、索引构建与问题检索之间的衔接关系。



LangChain提供了专门的文本切分工具，开发者可以设定片段大小与重叠长度。重叠的作用在于减少关键信息被切断的概率。例如，一段概念说明可能跨越两个片段，如果完全无重叠，就容易在检索时丢失一部分语义。

8.10.3数据准备与文档切分

这里先以四个自制的txt文档为例，被放在项目目录的course_docs子目录下，分别为01_hbase_mapreduce.txt、02_hbase_data_model.txt、03_hbase_region.txt和04_hbase_storage_structure.txt，其中，01_hbase_mapreduce.txt的部分内容如下：

1. HBase是BigTable的开源实现，主要面向海量数据存储与随机读写场景。

2. 在Hadoop生态中，HBase与MapReduce、HDFS和ZooKeeper关系紧密。

3. HDFS用于提供底层分布式存储能力，是HBase保存数据文件的重要基础。

4. ZooKeeper用于维护集群协调信息，并帮助系统完成故障检测与状态同步。

（备注：以下内容省略，可以到教材官网获取完整内容）

8.10.3数据准备与文档切分

02_hbase_data_model.txt内容如下：

1. HBase采用列族数据模型，而不是传统关系数据库中的固定二维表结构。

2. HBase数据模型的核心概念包括表、行键、列族、列限定符、单元格和时间戳。

3. 行键用于唯一标识一行记录，也是HBase最重要的索引方式。

4. 行键设计会直接影响数据分布、读取效率和热点问题，因此在建模时必须认真考虑。

（备注：以下内容省略，可以到教材官网获取完整内容）

8.10.3数据准备与文档切分

03_hbase_region.txt内容如下：

1. HBase表会按照行键范围被划分为多个Region，以支持分布式存储与横向扩展。

2. Region可以理解为表在物理层面上的一个分片，每个Region负责一段连续的行键范围。

3. 当一个Region增长到一定规模后，系统会自动将其分裂为两个新的Region。

4. Region分裂的目的在于避免单个分片过大，并提升系统的并行处理能力。

（备注：以下内容省略，可以到教材官网获取完整内容）

8.10.3数据准备与文档切分

文档加载和文本切分的示例代码如下：

```
# langchain_text_load_split.py
from pathlib import Path
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
data_dir = Path("course_docs")
docs = []
for path in data_dir.glob("*.txt"):
    loader = TextLoader(str(path), encoding="utf-8")
    docs.extend(loader.load())
splitter = RecursiveCharacterTextSplitter(
    chunk_size=350,
    chunk_overlap=80
)
splits = splitter.split_documents(docs)
print(f"原始文档数：{len(docs)}")
print(f"切分后片段数：{len(splits)}")
```

8.10.4向量索引构建

文本切分完成后，需要把每个文本片段映射为向量表示。所谓向量，可以理解为文本语义在高维空间中的数值表达。语义相近的文本，其向量位置往往也更接近。基于这一性质，系统就可以在用户提出问题时，用问题向量去检索最相近的文本片段。

在本案例中，使用FAISS作为本地向量索引工具。它不依赖额外数据库服务，便于在个人计算机上复现。需要指出的是，向量索引并不直接生成答案，它只负责找出最相关的知识片段。真正的答案仍由大模型在阅读这些片段后生成。索引构建过程如下所示：

```
# langchain_index_build.py
from langchain_community.vectorstores import FAISS
from langchain_ollama import OllamaEmbeddings
from pathlib import Path
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

8.10.4向量索引构建

```
data_dir = Path("course_docs")
docs = []
for path in data_dir.glob("*.txt"):
    loader = TextLoader(str(path), encoding="utf-8")
    docs.extend(loader.load())
splitter = RecursiveCharacterTextSplitter(
    chunk_size=350,
    chunk_overlap=80
)
splits = splitter.split_documents(docs)
print(f"原始文档数：{len(docs)}")
print(f"切分后片段数：{len(splits)}")
embeddings = OllamaEmbeddings(model="nomic-embed-text", base_url="http://127.0.0.1:11434")
vector_store = FAISS.from_documents(
    documents=splits,
    embedding=embeddings
)
retriever = vector_store.as_retriever(search_kwargs={"k": 5})
print(retriever)
```

8.10.5检索问答链实现

完成索引构建之后，就可以把检索过程与大模型生成过程连接起来，形成完整的问答链。实现思路是：先由检索器根据问题返回若干文本片段，再把这些片段拼接成上下文，与用户问题共同填入提示词模板，最后调用大模型生成回答。为了让结果更易于程序处理，示例中继续使用字符串输出解析器。

这一实现方式本质上属于2-Step RAG，即检索步骤先于生成步骤且流程固定。它的优点是结构清晰、执行稳定。若未来需要把系统升级为更灵活的知识智能体，还可以把检索器进一步封装为工具，由智能体决定何时检索。检索问答链的核心代码如下：

```
# langchain_find_answer_chain.py
from langchain_community.vectorstores import FAISS
from langchain_ollama import OllamaEmbeddings
from pathlib import Path
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_ollama import ChatOllama
```

8.10.5检索问答链实现

```
data_dir = Path("course_docs")
docs = []
for path in data_dir.glob("*.txt"):
    loader = TextLoader(str(path), encoding="utf-8")
    docs.extend(loader.load())
splitter = RecursiveCharacterTextSplitter(
    chunk_size=350,
    chunk_overlap=80
)
splits = splitter.split_documents(docs)
print(f"原始文档数：{len(docs)}")
print(f"切分后片段数：{len(splits)}")
embeddings = OllamaEmbeddings(model="nomic-embed-text", base_url="http://127.0.0.1:11434")
vector_store = FAISS.from_documents(
    documents=splits,
    embedding=embeddings
)
```

8.10.5检索问答链实现

```
retriever = vector_store.as_retriever(search_kwargs={"k": 5})
print(retriever)

def format_docs(documents):
    return "\n\n".join(doc.page_content for doc in documents)

prompt = ChatPromptTemplate.from_template(
    """你是一名课程助教。请结合给定资料回答问题。
如果资料中没有足够信息，请明确说明“资料中未给出充分信息”。
资料内容：
{context}
用户问题：
{question}
"""
)
model = ChatOllama(model="gemma4:e4b", temperature=0, base_url="http://127.0.0.1:11434")
parser = StrOutputParser()
```

8.10.5检索问答链实现

```
rag_chain = (
        {
            "context": retriever | format_docs,
            "question": RunnablePassthrough()
        }
        | prompt
        | model
        | parser
)
answer = rag_chain.invoke("MapReduce 在课程资料中被描述为什么？")
print(answer)
```

8.10.6运行结果与效果分析

从应用效果看，课程资料问答助手的回答质量主要取决于两个因素：

一是检索结果是否真正包含与问题相关的片段。

二是大模型是否能够基于给定资料进行准确概括。

如果检索阶段选出的片段与问题偏离，即使大模型能力较强，也难以生成高质量回答；如果检索片段本身信息充分，大模型往往能够给出较准确且具有解释性的结果。

可以设计若干测试问题，对系统表现进行分析，例如概念解释类问题、细节定位类问题和超出资料范围的问题。对于超出资料范围的问题，一个设计良好的问答链应尽量提示“资料不足”，而不是盲目编造答案。

8.10.6运行结果与效果分析

下表给出了案例中可采用的测试问题与观察重点。

问题类型	示例问题	观察重点
概念解释	什么是列式存储?	回答是否抓住核心概念
细节定位	课程资料中提到的HBase组件有哪些?	是否能准确引用检索内容
对比分析	HBase与关系数据库有何差异?	是否能综合多个片段
越界问题	课程资料是否介绍了Transformer的训练过程?	是否能够说明资料不足

需要指出的是，本案例仍然是一个基础系统。它没有处理多轮会话记忆、文档更新自动同步、答案来源高亮显示、复杂权限控制等问题。但正因为边界清晰，它适合作为进一步扩展的起点。在此基础上，可以继续尝试加入来源引用、Web界面、会话历史管理或工具调用能力，从而逐步把链式问答系统扩展成更完整的智能体应用。

8.11 本章小结

本章围绕智能体开发框架LangChain展开介绍。

1. 首先说明了LangChain的定位，即它并不是模型本身，而是面向大模型应用开发的组织框架。与直接调用模型API相比，LangChain更强调模块化和流程化，适合构建包含外部知识、工具调用和多步骤处理的应用系统。

2. 随后，本章分析了LangChain的核心组成、消息、提示词、工具、状态和持久化。

3. 最后，本章通过课程资料问答助手案例，演示了文档加载、文本切分、向量索引构建和检索问答链实现的完整过程。该案例表明，LangChain的真正价值不只在于“调用模型”，更在于帮助开发者把模型、知识和流程整合成可维护的应用系统。

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