

Hello-Agents: 动手学智能体



Hello-Agents

《从零开始构建智能体》



从基础理论到实际应用，全面掌握智能体系统的设计与实现

stars 20k forks 2.3k language Chinese GitHub Project 在线阅读 Online Reading

🎯 项目介绍

如果说 2024 年是“百模大战”的元年，那么 2025 年无疑开启了“Agent 元年”。技术的焦点正从训练更大的基础模型，转向构建更聪明的智能体应用。然而，当前系统性、重实践的教程却极度匮乏。为此，我们发起了 Hello-Agents 项目，希望能为社区提供一本从零开始、理论与实战并重的智能体系统构建指南。

Hello-Agents 是 Datawhale 社区的系统性智能体学习教程。如今 Agent 构建主要分为两派，一派是 Dify, Coze, n8n 这类软件工程类 Agent，其本质是流程驱动的软件开发，LLM 作为数据处理的后端；另一派则是 AI 原生的 Agent，即真正以 AI 驱动的 Agent。本教程旨在带领大家深入理解并构建后者——真正的 AI Native Agent。教程将带领你穿透框架表象，从智能体的核心原理出发，深入其核心架构，理解其经典范式，并最终亲手构建起属于自己的多智能体应用。我们相信，最好的学习方式就是动手实践。希望这本教程能成为你探索智能体世界的起点，能够从一名大语言模型的“使用者”，蜕变为一名智能体系统的“构建者”。

🌐 在线阅读

[点击这里开始在线阅读](#)

[Cookbook\(测试版\)](#)

✨ 你将收获什么？

- 📖 Datawhale 开源免费 完全免费学习本项目所有内容，与社区共同成长
- 🔍 理解核心原理 深入理解智能体的概念、历史与经典范式

- 亲手实现 掌握热门低代码平台和智能体代码框架的使用
- 自研框架HelloAgents 基于 Openai 原生 API 从零构建一个自己的智能体框架
- 掌握高级技能 一步步实现上下文工程、Memory、协议、评估等系统性技术
- 模型训练 掌握 Agentic RL，从 SFT 到 GRPO 的全流程实战训练 LLM
- 驱动真实案例 实战开发智能旅行助手、赛博小镇等综合项目
- 求职面试 学习智能体求职相关面试问题

内容导航

章节	关键内容	状态
前言	项目的缘起、背景及读者建议	✓
第一部分：智能体与语言模型基础		
第一章 初识智能体	智能体定义、类型、范式与应用	✓
第二章 智能体发展史	从符号主义到 LLM 驱动的智能体演进	✓
第三章 大语言模型基础	Transformer、提示、主流 LLM 及其局限	✓
第二部分：构建你的大语言模型智能体		
第四章 智能体经典范式构建	手把手实现 ReAct、Plan-and-Solve、Reflection	✓
第五章 基于低代码平台的智能体搭建	了解 Coze、Dify、n8n 等低代码智能体平台使用	✓
第六章 框架开发实践	AutoGen、AgentScope、LangGraph 等主流框架应用	✓
第七章 构建你的Agent框架	从 0 开始构建智能体框架	✓
第三部分：高级知识扩展		
第八章 记忆与检索	记忆系统、RAG、存储	✓
第九章 上下文工程	持续交互的“情境理解”	✓
第十章 智能体通信协议	MCP、A2A、ANP 等协议解析	✓
第十一章 Agentic-RL	从 SFT 到 GRPO 的 LLM 训练实战	✓
第十二章 智能体性能评估	核心指标、基准测试与评估框架	✓
第四部分：综合案例进阶		
第十三章 智能旅行助手	MCP 与多智能体协作的真实世界应用	✓
第十四章 自动化深度研究智能体	DeepResearch Agent 复现与解析	✓
第十五章 构建赛博小镇	Agent 与游戏的结合，模拟社会动态	✓
第五部分：毕业设计及未来展望		
第十六章 毕业设计	构建属于你的完整多智能体应用	✓

社区贡献精选 (Community Blog)

欢迎大家将在学习 Hello-Agents 或 Agent 相关技术中的独到见解、实践总结，以 PR 的形式贡献到社区精选。如果是独立于正文的内容，也可以投稿至 Extra-Chapter！期待你的第一次贡献！

社区精选	内容总结
00-共创毕业设计	社区共创毕业设计项目
01-Agent面试题总结	Agent 岗位相关面试问题
01-Agent面试题答案	相关面试问题答案
02-上下文工程内容补充	上下文工程内容扩展
03-Dify智能体创建保姆级教程	Dify智能体创建保姆级教程
04-Hello-agents课程常见问题	Datawhale课程常见问题
05-Agent Skills与MCP对比解读	Agent Skills与MCP技术对比
06-GUI Agent科普与实战	GUI Agent科普与多场景实战
07-环境配置	环境配置

PDF 版本下载

本Hello-Agents PDF 教程完全开源免费。为防止各类营销号加水印后贩卖给多智能体系统初学者，我们特地在PDF文件中预先添加了不影响阅读的Datawhale 开源标志水印，敬请谅解~

Hello-Agents PDF : <https://github.com/datawhalechina/hello-agents/releases/tag/V1.0.0>

Hello-Agents PDF 国内下载地址 : <https://www.datawhale.cn/learn/summary/239>

如何学习

欢迎你，未来的智能系统构建者！在开启这段激动人心的旅程之前，请允许我们给你一些清晰的指引。

本项目内容兼顾理论与实战，旨在帮助你系统性地掌握从单个智能体到多智能体系统的设计与开发全流程。因此，尤其适合有一定编程基础的AI开发者、软件工程师、在校学生以及对前沿AI技术抱有浓厚兴趣的自学者。在学习本项目之前，我们希望你具备基础的Python编程能力，并对大语言模型有基本的概念性了解（例如，知道如何通过API调用一个LLM）。项目的重点是应用与构建，因此你无需具备深厚的算法或模型训练背景。

项目分为五大部分，每一部分都是通往下一阶段的坚实阶梯：

- 第一部分：智能体与语言模型基础（第一章～第三章），我们将从智能体的定义、类型与发展历史讲起，为你梳理“智能体”这一概念的来龙去脉。随后，我们会快速巩固大语言模型的核心知识，为你的实践之旅打下坚实的理论地基。
- 第二部分：构建你的大语言模型智能体（第四章～第七章），这是你动手实践的起点。你将亲手实现ReAct等经典范式，体验Coze等低代码平台的便捷，并掌握Langgraph等主流框架的应用。最终，我们还会带你从零开始构建一个属于自己的智能体框架，让你兼具“用轮子”与“造轮子”的能力。
- 第三部分：高级知识扩展（第八章～第十二章），在这一部分，你的智能体将“学会”思考与协作。我们将使用第二部分的自研框架，深入探索记忆与检索、上下文工程、Agent训练等核心技术，并学习多智能体间的通信协议。最终，你将掌握评估智能体系统性能的专业方法。
- 第四部分：综合案例进阶（第十三章～第十五章），这里是理论与实践的交汇点。你将把所学融会贯通，亲手打造智能旅行助手、自动化深度研究智能体，乃至一个模拟社会动态的赛博小镇，在真实有趣的项目中淬炼你的构建能力。
- 第五部分：毕业设计及未来展望（第十六章），在旅程的终点，你将迎来一个毕业设计，构建一个完整的、属于你自己的多智能体应用，全面检验你的学习成果。我们还将与你一同展望智能体的未来，探索激动人心的前沿方向。

智能体是一个飞速发展且极度依赖实践的领域。为了获得最佳的学习效果，我们在项目的code文件夹内提供了配套的全部代码，强烈建议你将理论与实践相结合。请务必亲手运行、调试甚至修改项目里提供的每一份代码。欢迎你随时关注Datawhale以及其他Agent相关社区，当遇到问题时，你可以随时在本项目的issue区提问。

现在，准备好进入智能体的奇妙世界了吗？让我们即刻启程！

如何贡献

我们是一个开放的开源社区，欢迎任何形式的贡献！

-  报告 Bug - 发现内容或代码问题，请提交Issue
-  提出建议 - 对项目有好想法，欢迎发起讨论
-  完善内容 - 帮助改进教程，提交你的Pull Request
-  分享实践 - 在“社区贡献精选”中分享你的学习笔记和项目

致谢

核心贡献者

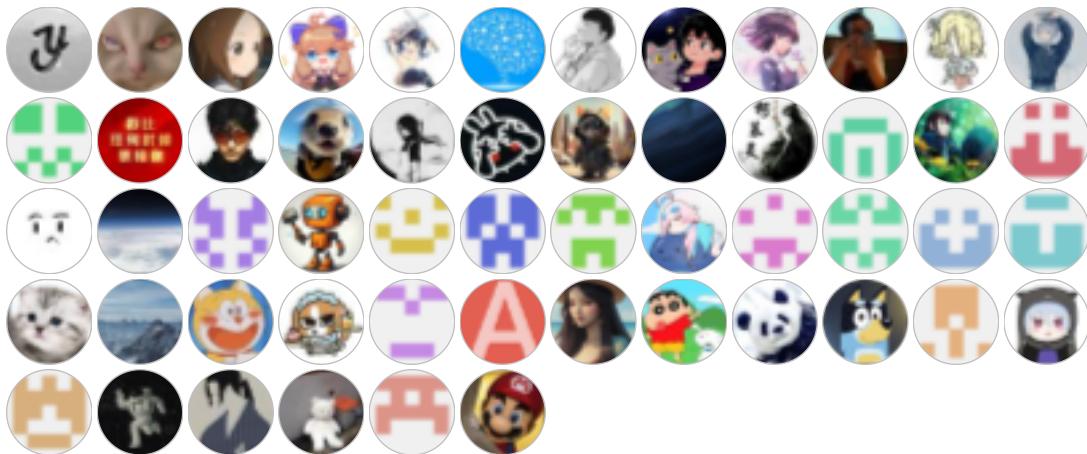
- 陈思州-项目负责人 (Datawhale成员, 全文写作和校对)
- 孙韬-项目负责人 (Datawhale成员, 第九章内容和校对)
- 姜舒凡-项目负责人 (Datawhale成员, 章节习题设计和校对)
- 黄佩林-Datawhale意向成员 (Agent开发工程师, 第五章内容贡献者)
- 曾鑫民-Agent工程师 (牛客科技, 第十四章案例开发)
- 朱信忠-指导专家 (Datawhale首席科学家-浙江师范大学杭州人工智能研究院教授)

Extra-Chapter 贡献者

- WH (内容贡献者)
- 周奥杰-DW贡献者团队 (西安交通大学, Extra02 内容贡献)
- 张宸旭-个人开发者 (帝国理工学院, Extra03 内容贡献)
- 黄宏玲-DW贡献者团队 (深圳大学, Extra04 内容贡献)

特别感谢

- 感谢 @Smiles 对本项目的帮助与支持
- 感谢所有为本项目做出贡献的开发者们 ❤️



Star History

★ 如果这个项目对你有帮助，请给我们一个 Star！

关于 Datawhale

<p>扫描二维码关注 Datawhale 公众号，获取更多优质开源内容</p>

开源协议

本作品采用[知识共享署名-非商业性使用-相同方式共享 4.0 国际许可协议](#)进行许可。

前言

自 2022 年底以来，以 ChatGPT 为代表的大语言模型（Large Language Model, LLM）如同一场技术海啸，彻底改变了我们与人工智能交互的方式。LLM 强大的自然语言理解和生成能力，让我们看到了通往通用人工智能（AGI）的曙光。然而，当最初的惊艳沉淀下来，开发者们开始探索下一个前沿：如何让 AI 不仅仅是一个“有问必答”的工具，而是成为一个能自主规划、调用工具、解决复杂问题的“行动者”？

答案，就是 智能体（Agent）。

如果说 2024 年是“百模大战”的元年，那么 2025 年无疑开启了“Agent 元年”。我们看到，技术的焦点正从训练更大、更强的基础模型，转向如何构建更聪明、更高效的智能体应用。单个智能体已经能胜任特定领域的任务，而由多个智能体分工、协作、甚至辩论，共同完成一个宏大目标的多智能体系统（Multi-Agent System, MAS），则被视为释放 LLM 全部潜能、解决真实世界复杂问题的关键钥匙。

然而，当前的生态中存在一个明显的断层：一方面是层出不穷的 Agent 框架和应用，令人眼花缭乱；另一方面，却是系统性知识的极度匮乏。大多数教程聚焦于某个特定框架的 API 调用，学习者往往“知其然，而不知其所以然”，在面对复杂需求时，依然感到力不从心。我们缺少一本能够穿透框架表象，从第一性原理出发，系统讲解智能体设计、构建与协作的实战指南。

鉴于此，我们发起了 Hello-Agents 项目，希望能为社区提供一本从零开始、理论与实战并重的智能体系统构建指南。我们不仅会带你领略智能体领域最前沿的技术，更会引导你深入其核心架构，理解其经典范式，并最终亲手构建起属于自己的多智能体应用。

我们相信，最好的学习方式就是动手实践。希望这本教程能成为你探索智能体世界的起点，能够从一名大语言模型的“使用者”，蜕变为一名智能体系统的“构建者”。

写给读者的建议

欢迎你，未来的智能系统构建者！在开启这段激动人心的旅程之前，请允许我们给你一些小小的建议。

在阅读项目之前，我们希望你：

- 具备基础的 Python 编程能力。
- 对大语言模型有基本的概念性了解（例如，知道如何获取 LLM 的 API）。
- 请放心，你无需具备深厚的算法或模型训练背景，项目的重点是应用与构建。

本项目分为五部分，覆盖基础到实战，循序渐进，层层相扣：

第一部分（基础篇）： 我们将为你铺垫人工智能与 LLM 的核心知识，让你对智能体的诞生背景有宏观的认识。

第二部分（单体篇）： 这是你动手实践的开始。我们将带你从零开始，构建一个功能完备的单体智能，深入理解其内部的“心智”结构。

第三部分（高级篇）： 在这里，你的智能体将“学会”思考、拥有记忆和工具，并掌握智能体之间的通信协议，最终完成评估的闭环。

第四部分（实战篇）： 这是项目的核心价值所在。你将通过一系列精心设计的综合案例，将所学知识融会贯通，在实战中淬炼真金。

第五部分（展望篇）： 旅程的终点是新的起点。你将亲手打造属于你的“毕业作品”，为你的学习之旅画上一个圆满的句号。

纸上得来终觉浅，绝知此事要躬行。为了获得最佳的学习效果，我们在项目的 code 文件夹内提供了配套的全部代码，强烈建议你将理论与实践相结合。请务必亲手运行、调试甚至修改项目里提供的每一份代码。我们鼓励你举一反三，将所学技术应用到自己感兴趣的真实场景中，这才是学习的最终目的。

最后，作为一个开源项目，我们热忱欢迎你的参与和贡献。当你遇到问题时，可以在我们的社区中提问；当你有了新的想法或发现时，也欢迎你随时加入到项目的共建中来。

感谢你选择阅读 Hello-agents，祝你学习愉快，探索无限！

第一章 初识智能体

欢迎来到智能体的世界！在人工智能浪潮席卷全球的今天，智能体（Agent）已成为驱动技术变革与应用创新的核心概念之一。无论你的志向是成为 AI 领域的研究者、工程师，还是希望深刻理解技术前沿的观察者，掌握智能体的本质，都将是知识体系中不可或缺的一环。

因此，在本章，让我们回到原点，一起探讨几个问题：智能体是什么？它有哪些主要的类型？它又是如何与我们所处的世界进行交互的？通过这些讨论，希望能为你未来的学习和探索打下坚实的基础。

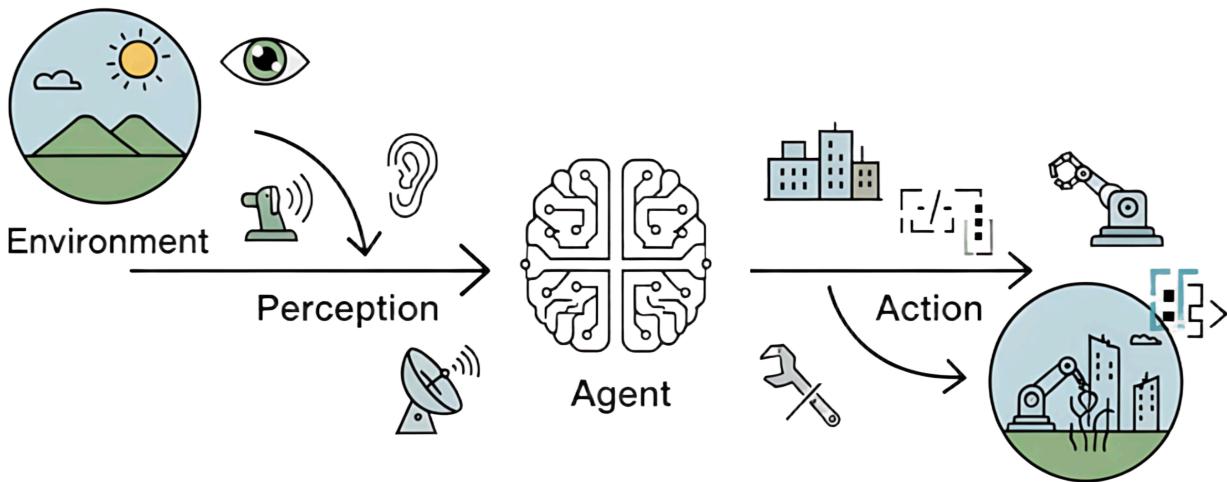


图 1.1 智能体与环境的基本交互循环

1.1 什么是智能体？

在探索任何一个复杂概念时，我们最好从一个简洁的定义开始。在人工智能领域，智能体被定义为任何能够通过传感器（Sensors）感知其所处环境（Environment），并自主地通过执行器（Actuators）采取行动（Action）以达成特定目标的实体。

这个定义包含了智能体存在的四个基本要素。环境是智能体所处的外部世界。对于自动驾驶汽车，环境是动态变化的道路交通；对于一个交易算法，环境则是瞬息万变的金融市场。智能体并非与环境隔离，它通过其传感器持续地感知环境状态。摄像头、麦克风、雷达或各类应用程序编程接口（Application Programming Interface, API）返回的数据流，都是其感知能力的延伸。

获取信息后，智能体需要采取行动来对环境施加影响，它通过执行器来改变环境的状态。执行器可以是物理设备（如机械臂、方向盘）或虚拟工具（如执行一段代码、调用一个服务）。

然而，真正赋予智能体“智能”的，是其自主性（Autonomy）。智能体并非只是被动响应外部刺激或严格执行预设指令的程序，它能够基于其感知和内部状态进行独立决策，以达成其设计目标。这种从感知到行动的闭环，构成了所有智能体行为的基础，如图 1.1 所示。

1.1.1 传统视角下的智能体

在当前大语言模型（Large Language Model, LLM）的热潮出现之前，人工智能的先驱们已经对“智能体”这一概念进行了数十年的探索与构建。这些如今我们称之为“传统智能体”的范式，并非单一的静态概念，而是经历了一条从简单到复杂、从被动反应到主动学习的清晰演进路线。

这个演进的起点，是那些结构最简单的反射智能体（Simple Reflex Agent）。它们的决策核心由工程师明确设计的“条件-动作”规则构成，如图 1.2 所示。经典的自动恒温器便是如此：若传感器感知的室温高于设定值，则启动制冷系统。

这种智能体完全依赖于当前的感知输入，不具备记忆或预测能力。它像一种数字化的本能，可靠且高效，但也因此无法应对需要理解上下文的复杂任务。它的局限性引出了一个关键问题：如果环境的当前状态不足以作为决策的全部依据，智能体该怎么办？

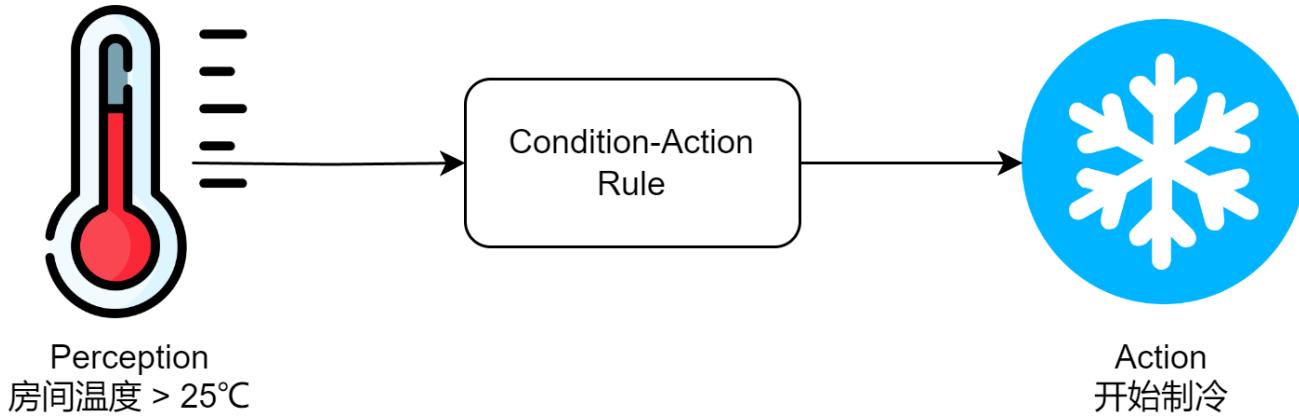


图 1.2 简单反射智能体的决策逻辑示意图

为了回答这个问题，研究者们引入了“状态”的概念，发展出基于模型的反射智能体（**Model-Based Reflex Agent**）。这类智能体拥有一个内部的世界模型（**World Model**），用于追踪和理解环境中那些无法被直接感知的方面。它试图回答：“世界现在是什么样子的？”。例如，一辆在隧道中行驶的自动驾驶汽车，即便摄像头暂时无法感知到前方的车辆，它的内部模型依然会维持对那辆车存在、速度和预估位置的判断。这个内部模型让智能体拥有了初级的“记忆”，使其决策不再仅仅依赖于瞬时感知，而是基于一个更连贯、更完整的世界状态理解。

然而，仅仅理解世界还不够，智能体需要有明确的目标。这促进了基于目标的智能体（**Goal-Based Agent**）的发展。与前两者不同，它的行为不再是被动地对环境做出反应，而是主动地、有预见性地选择能够导向某个特定未来状态的行动。这类智能体需要回答的问题是：“我应该做什么才能达成目标？”。经典的例子是 GPS 导航系统：你的目标是到达公司，智能体会基于地图数据（世界模型），通过搜索算法（如 A*算法）来规划（Planning）出一条最优路径。这类智能体的核心能力体现在了对未来的考量与规划上。

更进一步，现实世界的目标往往不是单一的。我们不仅希望到达公司，还希望时间最短、路程最省油并且避开拥堵。当多个目标需要权衡时，基于效用的智能体（**Utility-Based Agent**）便随之出现。它为每一个可能的世界状态都赋予一个效用值，这个值代表了满意度的高低。智能体的核心目标不再是简单地达成某个特定状态，而是最大化期望效用。它需要回答一个更复杂的问题：“哪种行为能为我带来最满意的结果？”。这种架构让智能体学会在相互冲突的目标之间进行权衡，使其决策更接近人类的理性选择。

至此，我们讨论的智能体虽然功能日益复杂，但其核心决策逻辑，无论是规则、模型还是效用函数，依然依赖于人类设计师的先验知识。如果智能体能不依赖预设，而是通过与环境的互动自主学习呢？

这便是学习型智能体（**Learning Agent**）的核心思想，而强化学习（**Reinforcement Learning, RL**）是实现这一思想最具代表性的路径。一个学习型智能体包含一个性能元件（即我们前面讨论的各类智能体）和一个学习元件。学习元件通过观察性能元件在环境中的行动所带来的结果来不断修正性能元件的决策策略。

想象一个学习下棋的 AI。它开始时可能只是随机落子，当它最终赢下一局时，系统会给予它一个正向的奖励。通过大量的自我对弈，学习元件会逐渐发现哪些棋路更有可能导向最终的胜利。AlphaGo Zero 是这一理念的一个里程碑式的成就。它在围棋这一复杂博弈中，通过强化学习发现了许多超越人类既有知识的有效策略。

从简单的恒温器，到拥有内部模型的汽车，再到能够规划路线的导航、懂得权衡利弊的决策者，最终到可以通过经验自我进化的学习者。这条演进之路，展示了传统人工智能在构建机器智能的道路上所经历的发展脉络。它们为我们今天理解更前沿的智能体范式，打下了坚实而必要的基础。

1.1.2 大语言模型驱动的新范式

以GPT（**Generative Pre-trained Transformer**）为代表的大语言模型的出现，正在显著改变智能体的构建方法与能力边界。由大语言模型驱动的LLM智能体，其核心决策机制与传统智能体存在本质区别，从而赋予了其一系列全新的特性。

这种转变，可以从两者在核心引擎、知识来源、交互方式等多个维度的对比中清晰地看出，如表 1.1 所示。简而言之，传统智能体的能力源于工程师的显式编程与知识构建，其行为模式是确定且有边界的；而 LLM 智能体则通过在海量数据上的预训练，获得了隐式的世界模型与强大的涌现能力，使其能够以更灵活、更通用的方式应对复杂任务。

表 1.1 传统智能体与 LLM 驱动智能体的核心对比

对比维度	传统智能体	LLM驱动的智能体
核心引擎	基于显式编程的逻辑系统	基于预训练模型的推理引擎
知识来源	工程师预定义的规则、算法、知识库	从海量非结构化数据中间接学习、内化
处理指令	需结构化、精确的命令	可理解高层级、模糊的自然语言
工作模式	确定性的、可预测的	概率性的、生成式的
泛化/适应性	弱，局限于预设框架	强，具备强大的涌现能力和泛化能力
开发范式	规则设计、算法编程、知识工程	模型训练、提示工程、微调

这种差异使得 LLM 智能体可以直接处理高层级、模糊且充满上下文信息的自然语言指令。让我们以一个“智能旅行助手”为例来说明。

在 LLM 智能体出现之前，规划旅行通常意味着用户需要在多个专用应用（如天气、地图、预订网站）之间手动切换，并由用户自己扮演信息整合与决策的角色。而一个 LLM 智能体则能将这个流程整合起来。当接收到“规划一次厦门之旅”这样的模糊指令时，它的工作方式体现了以下几点：

- 规划与推理：智能体首先会将这个高层级目标分解为一系列逻辑子任务，例如：【确认出行偏好】 \rightarrow 【查询目的地信息】 \rightarrow 【制定行程草案】 \rightarrow 【预订票务住宿】。这是一个内在的、由模型驱动的规划过程。
- 工具使用：在执行规划时，智能体识别到信息缺口，会主动调用外部工具来补全。例如，它会调用天气查询接口获取实时天气，并基于“预报有雨”这一信息，在后续规划中倾向于推荐室内活动。
- 动态修正：在交互过程中，智能体会将用户的反馈（如“这家酒店超出预算”）视为新的约束，并据此调整后续的行动，重新搜索并推荐符合新要求的选项。整个“查天气 \rightarrow 调行程 \rightarrow 订酒店”的流程，展现了其根据上下文动态修正自身行为的能力。

总而言之，我们正从开发专用自动化工具转向构建能自主解决问题的系统。核心不再是编写代码，而是引导一个通用的“大脑”去规划、行动和学习。

1.1.3 智能体的类型

继上文回顾智能体的演进后，本节将从三个互补的维度对智能体进行分类。

(1) 基于内部决策架构的分类

第一种分类维度是依据智能体内部决策架构的复杂程度，这个视角在《Artificial Intelligence: A Modern Approach》中系统性地提出^[1]。正如 1.1.1 节所述，传统智能体的演进路径本身就构成了最经典的分类阶梯，它涵盖了从简单的反应式智能体，到引入内部模型的模型式智能体，再到更具前瞻性的基于目标和基于效用的智能体。此外，学习能力则是一种可赋予上述所有类型的元能力，使其能通过经验自我改进。

(2) 基于时间与反应性的分类

除了内部架构的复杂性，还可以从智能体处理决策的时间维度进行分类。这个视角关注智能体是在接收到信息后立即行动，还是会经过深思熟虑的规划再行动。这揭示了智能体设计中一个核心权衡：追求速度的反应性（Reactivity）与追求最优解的规划性（Deliberation）之间的平衡，如图 1.3 所示。

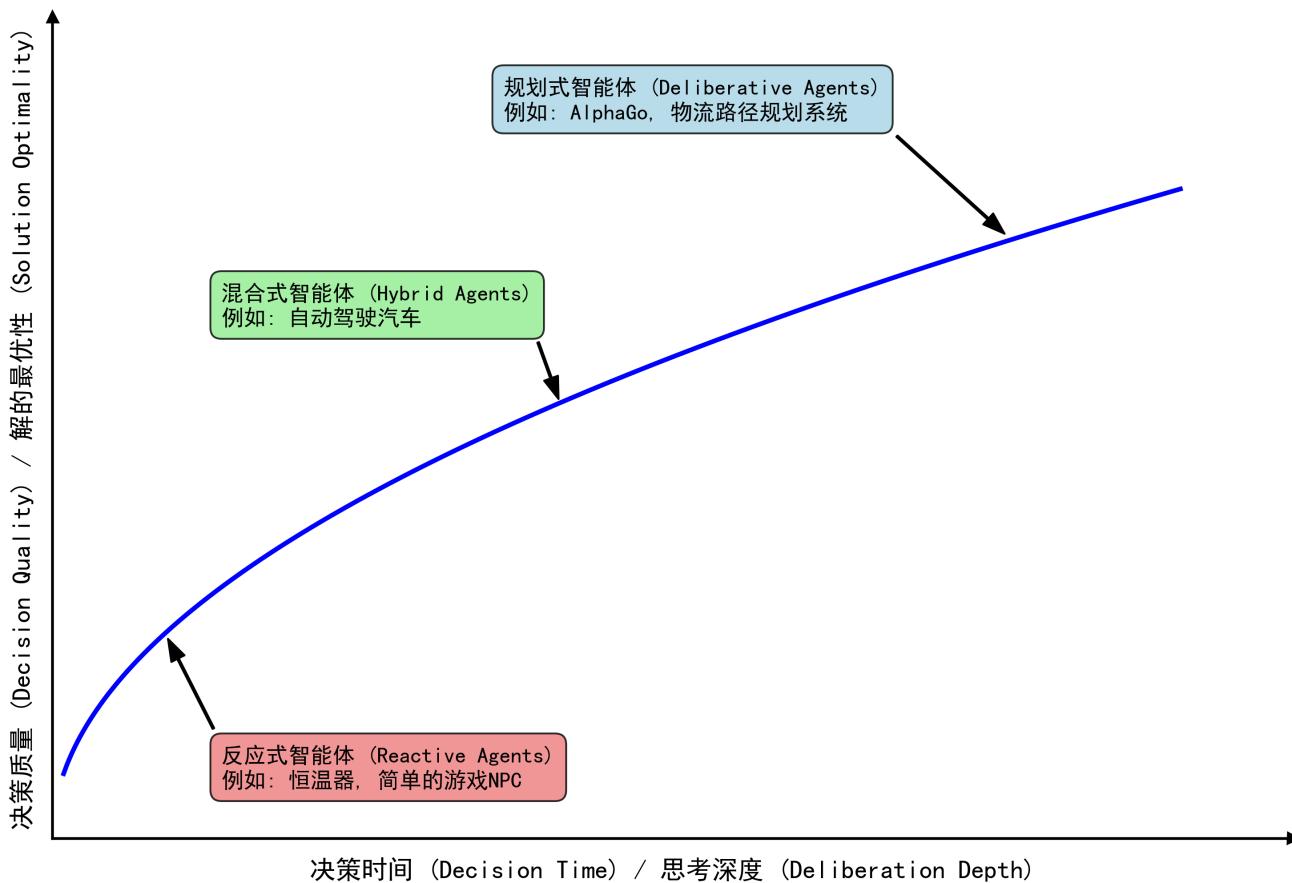


图 1.3 智能体决策时间与质量关系图

• 反应式智能体 (Reactive Agents)

这类智能体对环境刺激做出近乎即时的响应，决策延迟极低。它们通常遵循从感知到行动的直接映射，不进行或只进行极少的未来规划。上文的简单反应式和基于模型的智能体都属于此类别。

其核心优势在于速度快、计算开销低，这在需要快速决策的动态环境中至关重要。例如，车辆的安全气囊系统必须在碰撞发生的毫秒内做出反应，任何延迟都可能导致严重后果；同样，高频交易机器人也必须依赖反应式决策来捕捉稍纵即逝的市场机会。然而，这种速度的代价是“短视”，由于缺乏长远规划，反

应式智能体容易陷入局部最优，难以完成需要多步骤协调的复杂任务。

- 规划式智能体(Deliberative Agents)

与反应式智能体相对，规划式（或称审议式）智能体在行动前会进行复杂的思考和规划。它们不会立即对感知做出反应，而是会先利用其内部的世界模型，系统地探索未来的各种可能性，评估不同行动序列的后果，以期找到一条能够达成目标的最佳路径。基于目标和基于效用的智能体是典型的规划式智能体。

可以将其决策过程类比为一位棋手。他不会只看眼前的一步，而是会预想对手可能的应对，并规划出后续几步甚至十几步的棋路。这种深思熟虑的能力使其能够处理复杂的、需要长远眼光的任务，例如制定一份商业计划或规划一次长途旅行。它们的优势在于决策的战略性和远见。然而，这种优势的另一面是高昂的时间和计算成本。在瞬息万变的环境中，当规划式智能体还在深思熟虑时，采取行动的最佳时机可能早已过去。

- 混合式智能体(Hybrid Agents)

现实世界的复杂任务，往往既需要即时反应，也需要长远规划。例如，我们之前提到的智能旅行助手，既要能根据用户的即时反馈（如“这家酒店太贵了”）调整推荐（反应性），又要能规划出为期数天的完整旅行方案（规划性）。因此，混合式智能体应运而生，它旨在结合两者的优点，实现反应与规划的平衡。

一种经典的混合架构是分层设计：底层是一个快速的反应模块，处理紧急情况和基本动作；高层则是一个审慎的规划模块，负责制定长远目标。而现代的LLM智能体，则展现了一种更灵活的混合模式。它们通常在一个“思考-行动-观察”的循环中运作，巧妙地将两种模式融为一体：

- 规划(Reasoning)：在“思考”阶段，LLM分析当前状况，规划出下一步的合理行动。这是一个审议过程。
- 反应(Acting & Observing)：在“行动”和“观察”阶段，智能体与外部工具或环境交互，并立即获得反馈。这是一个反应过程。

通过这种方式，智能体将一个需要长远规划的宏大任务，分解为一系列“规划-反应”的微循环。这使其既能灵活应对环境的即时变化，又能通过连贯的步骤，最终完成复杂的长期目标。

(3) 基于知识表示的分类

这是一个更根本的分类维度，它探究智能体用以决策的知识，究竟是以何种形式存于其“思想”之中。这个问题是人工智能领域一场持续半个多世纪的辩论核心，并塑造了两种截然不同的AI文化。

- 符号主义 AI (Symbolic AI)

符号主义，常被称为传统人工智能，其核心信念是：智能源于对符号的逻辑操作。这里的符号是人类可读的实体（如词语、概念），操作则遵循严格的逻辑规则，如图1.4左侧所示。这好比一位一丝不苟的图书管理员，将世界知识整理为清晰的规则库和知识图谱。

其主要优势在于透明和可解释。由于推理步骤明确，其决策过程可以被完整追溯，这在金融、医疗等高风险领域至关重要。然而，其“阿喀琉斯之踵”在于脆弱性：它依赖于一个完备的规则体系，但在充满模糊和例外的现实世界中，任何未被覆盖的新情况都可能导致系统失灵，这就是所谓的“知识获取瓶颈”。

- 亚符号主义 AI (Sub-symbolic AI)

亚符号主义，或称连接主义，则提供了一幅截然不同的图景。在这里，知识并非显式的规则，而是内隐地分布在一个由大量神经元组成的复杂网络中，是从海量数据中学习到的统计模式。神经网络和深度学习是其代表。

如图1.4中间所示，如果说符号主义AI是图书管理员，那么亚符号主义AI就像一个牙牙学语的孩童。他不是通过学习“猫有四条腿、毛茸茸、会喵喵叫”这样的规则来认识猫的，而是在看过成千上万张猫的图片后，大脑中的神经网络能辨识出“猫”这个概念的视觉模式。这种方法的强大之处在于其模式识别能力和对噪声数据的鲁棒性。它能够轻松处理图像、声音等非结构化数据，这在符号主义AI看来是极其困难的任务。

然而，这种强大的直觉能力也伴随着不透明性。亚符号主义系统通常被视为一个黑箱（Black Box）。它能以惊人的准确率识别出图片中的猫，但你若问它“为什么你认为这是猫？”，它很可能无法给出一个合乎逻辑的解释。此外，它在纯粹的逻辑推理任务上表现不佳，有时会产生看似合理却事实错误的幻觉。

- 神经符号主义 AI (Neuro-Symbolic AI)

长久以来，符号主义和亚符号主义这两大阵营如同两条平行线，各自发展。为克服上述两种范式的局限，一种“大和解”的思想开始兴起，这就是神经符号主义AI，也称神经符号混合主义。它的目标，是融合两大范式的优点，创造出一个既能像神经网络一样从数据中学习，又能像符号系统一样进行逻辑推理的混合智能体。它试图弥合感知与认知、直觉与理性之间的鸿沟。诺贝尔经济学奖得主丹尼尔·卡尼曼（Daniel Kahneman）在其著作《思考，快与慢》（Thinking, Fast and Slow）中提出的双系统理论，为我们理解神经符号主义提供了一个绝佳的类比^[2]，如图1.4所示：

- 系统1是缓慢、有条理、基于逻辑的审慎思维，恰如符号主义AI的推理过程。
- 系统2是快速、凭直觉、并行的思维模式，类似于亚符号主义AI强大的模式识别能力。

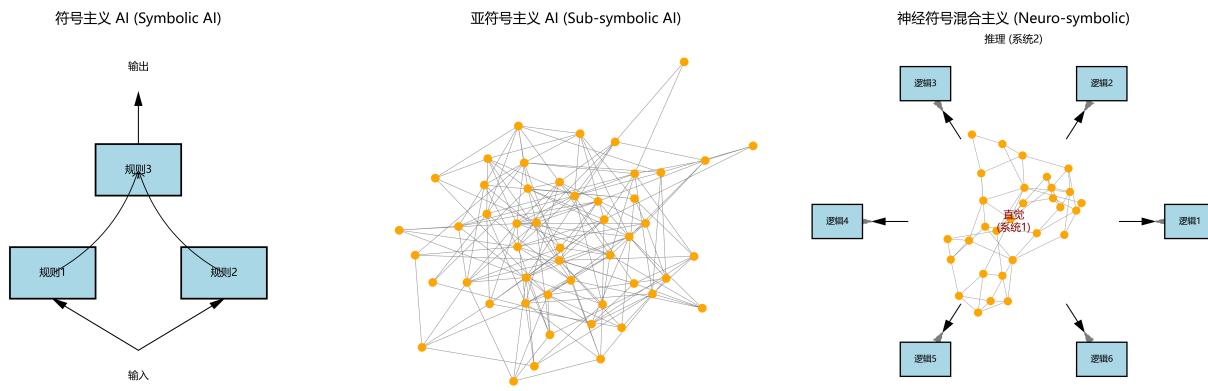


图 1.4 符号主义、亚符号主义与神经符号混合主义的知识表示范式

人类的智能，正源于这两个系统的协同工作。同样，一个真正鲁棒的 AI，也需要兼具二者之长。大语言模型驱动的智能体是神经符号主义的一个极佳实践范例。其内核是一个巨大的神经网络，使其具备模式识别和语言生成能力。然而，当它工作时，它会生成一系列结构化的中间步骤，如思想、计划或 API 调用，这些都是明确的、可操作的符号。通过这种方式，它实现了感知与认知、直觉与理性的初步融合。

1.2 智能体的构成与运行原理

1.2.1 任务环境定义

要理解智能体的运作，我们必须先理解它所处的任务环境。在人工智能领域，通常使用 **PEAS** 模型来精确描述一个任务环境，即分析其性能度量 (**Performance**)、环境 (**Environment**)、执行器 (**Actuators**) 和传感器 (**Sensors**)。以上文提到的智能旅行助手为例，下表 1.2 展示了如何运用 PEAS 模型对其任务环境进行规约。

表 1.2 智能旅行助手的 PEAS 描述

维度	描述
Performance (性能度量)	在预算和时间内，最大化用户满意度与行程合理性
Environment (环境)	航旅预订网站、地图服务、天气预报API等网络服务
Actuators (执行器)	调用API的函数、向用户界面生成和显示格式化文本
Sensors (传感器)	解析API返回的数据(如JSON, HTML)、读取用户输入的自然语言

在实践中，LLM 智能体所处的数字环境展现出若干复杂特性，这些特性直接影响着智能体的设计。

首先，环境通常是部分可观测的。例如，旅行助手在查询航班时，无法一次性获取所有航空公司的全部实时座位信息。它只能通过调用航班预订 API，看到该 API 返回的部分数据，这就要求智能体必须具备记忆（记住已查询过的航线）和探索（尝试不同的查询日期）的能力。

其次，行动的结果也并非总是确定的。根据结果的可预测性，环境可分为确定性和随机性。旅行助手的任务环境就是典型的随机性环境。当它搜索票价时，两次相邻的调用返回的机票价格和余票数量都可能不同，这就要求智能体必须具备处理不确定性、监控变化并及时决策的能力。

此外，环境中还可能存在其他行动者，从而形成多智能体 (**Multi-agent**) 环境。对于旅行助手而言，其他用户的预订行为、其他自动化脚本，甚至航司的动态调价系统，都是环境中的其他“智能体”。它们的行动（例如，订走最后一张特价票）会直接改变旅行助手所处环境的状态，这对智能体的快速响应和策略选择提出了更高要求。

最后，几乎所有任务都发生在序贯且动态的环境中。“序贯”意味着当前动作会影响未来；而“动态”则意味着环境自身可能在智能体决策时发生变化。这就要求智能体的“感知-思考-行动-观察”循环必须能够快速、灵活地适应持续变化的世界。

1.2.2 智能体的运行机制

在定义了智能体所处的任务环境后，我们来探讨其核心的运行机制。智能体并非一次性完成任务，而是通过一个持续的循环与环境进行交互，这个核心机制被称为 **智能体循环** (**Agent Loop**)。如图 1.5 所示，该循环描述了智能体与环境之间的动态交互过程，构成了其自主行为的基础。

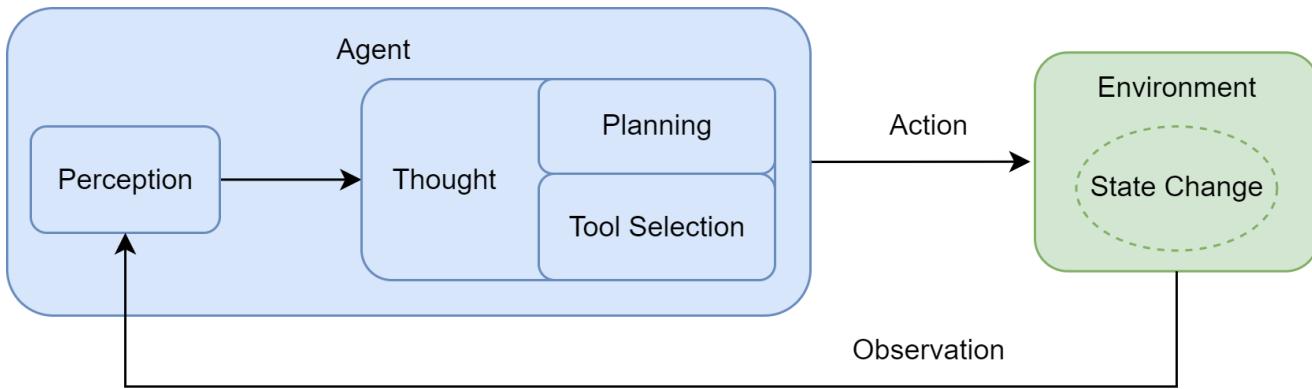


图 1.5 智能体与环境交互的基本循环

这个循环主要包含以下几个相互关联的阶段：

1. 感知 (**Perception**)：这是循环的起点。智能体通过其传感器（例如，API 的监听端口、用户输入接口）接收来自环境的输入信息。这些信息，即观察 (**Observation**)，既可以是用户的初始指令，也可以是上一步行动所导致的环境状态变化反馈。
2. 思考 (**Thought**)：接收到观察信息后，智能体进入其核心决策阶段。对于 LLM 智能体而言，这通常是由大语言模型驱动的内部推理过程。如图所示，“思考”阶段可进一步细分为两个关键环节：
 - 规划 (**Planning**)：智能体基于当前的观察和其内部记忆，更新对任务和环境的理解，并制定或调整一个行动计划。这可能涉及将复杂目标分解为一系列更具体的子任务。
 - 工具选择 (**Tool Selection**)：根据当前计划，智能体从其可用的工具库中，选择最适合执行下一步骤的工具，并确定调用该工具所需的具体参数。
3. 行动 (**Action**)：决策完成后，智能体通过其执行器 (Actuators) 执行具体的行动。这通常表现为调用一个选定的工具（如代码解释器、搜索引擎 API），从而对环境施加影响，意图改变环境的状态。

行动并非循环的终点。智能体的行动会引起环境 (**Environment**) 的状态变化 (**State Change**)，环境随即会产生一个新的观察 (**Observation**) 作为结果反馈。这个新的观察又会在下一轮循环中被智能体的感知系统捕获，形成一个持续的“感知-思考-行动-观察”的闭环。智能体正是通过不断重复这一循环，逐步推进任务，从初始状态向目标状态演进。

1.2.3 智能体的感知与行动

在工程实践中，为了让 LLM 能够有效驱动这个循环，我们需要一套明确的交互协议 (**Interaction Protocol**) 来规范其与环境之间的信息交换。

在许多现代智能体框架中，这一协议体现在对智能体每一次输出的结构化定义上。智能体的输出不再是单一的自然语言回复，而是一段遵循特定格式的文本，其中明确地展示了其内部的推理过程与最终决策。

这个结构通常包含两个核心部分：

- **Thought** (思考)：这是智能体内部决策的“快照”。它以自然语言形式阐述了智能体如何分析当前情境、回顾上一步的观察结果、进行自我反思与问题分解，并最终规划出下一步的具体行动。
- **Action** (行动)：这是智能体基于思考后，决定对环境施加的具体操作，通常以函数调用的形式表示。

例如，一个正在规划旅行的智能体可能会生成如下格式化的输出：

Thought: 用户想知道北京的天气。我需要调用天气查询工具。

Action: get_weather("北京")

这里的**Action**字段构成了对外部世界的指令。一个外部的解析器 (**Parser**) 会捕捉到这个指令，并调用相应的get_weather函数。

行动执行后，环境会返回一个结果。例如，get_weather函数可能返回一个包含详细天气数据的 JSON 对象。然而，原始的机器可读数据（如 JSON）通常包含 LLM 无需关注的冗余信息，且格式不符合其自然语言处理的习惯。

因此，感知系统的一个重要职责就是扮演传感器的角色：将这个原始输出处理并封装成一段简洁、清晰的自然语言文本，即观察。

Observation: 北京当前天气为晴，气温25摄氏度，微风。

这段Observation文本会被反馈给智能体，作为下一轮循环的主要输入信息，供其进行新一轮的Thought和Action。

综上所述，通过这个由 Thought、Action、Observation 构成的严谨循环，LLM 智能体得以将内部的语言推理能力，与外部环境的真实信息和工具操作能力有效地结合起来。

1.3 动手体验：5 分钟实现第一个智能体

在前面的小节，我们学习了智能体的任务环境、核心运行机制以及 Thought-Action-Observation 交互范式。理论知识固然重要，但最好的学习方式是亲手实践。在本节中，我们将引导您使用几行简单的 Python 代码，从零开始构建一个可以工作的智能旅行助手。这个过程将遵循我们刚刚学到的理论循环，让您直观地感受到一个智能体是如何“思考”并与外部“工具”互动的。让我们开始吧！

在本案例中，我们的目标是构建一个能处理分步任务的智能旅行助手。需要解决的用户任务定义为：“你好，请帮我查询一下今天北京的天气，然后根据天气推荐一个合适的旅游景点。”要完成这个任务，智能体必须展现出清晰的逻辑规划能力。它需要先调用天气查询工具，并将获得的观察结果作为下一步的依据。在下一轮循环中，它再调用景点推荐工具，从而得出最终建议。

1.3.1 准备工作

为了能从 Python 程序中访问网络 API，我们需要一个 HTTP 库。`requests` 是 Python 社区中最流行、最易用的选择。`tavily-python` 是一个强大的 AI 搜索 API 客户端，用于获取实时的网络搜索结果，可以在[官网](#)注册后获取 API。`openai` 是 OpenAI 官方提供的 Python SDK，用于调用 GPT 等大语言模型服务。请先通过以下命令安装它们：

```
pip install requests tavily-python openai
```

(1) 指令模板

驱动真实 LLM 的关键在于提示工程（**Prompt Engineering**）。我们需要设计一个“指令模板”，告诉 LLM 它应该扮演什么角色、拥有哪些工具、以及如何格式化它的思考和行动。这是我们智能体的“说明书”，它将作为`system_prompt`传递给 LLM。

```
AGENT_SYSTEM_PROMPT = """
```

你是一个智能旅行助手。你的任务是分析用户的请求，并使用可用工具一步一步地解决问题。

```
# 可用工具:
```

- ``get_weather(city: str)``: 查询指定城市的实时天气。
- ``get_attraction(city: str, weather: str)``: 根据城市和天气搜索推荐的旅游景点。

```
# 输出格式要求:
```

你的每次回复必须严格遵循以下格式，包含一对Thought和Action：

Thought: [你的思考过程和下一步计划]

Action: [你要执行的具体行动]

Action的格式必须是以下之一：

1. 调用工具: `function_name(arg_name="arg_value")`
2. 结束任务: `Finish[最终答案]`

```
# 重要提示:
```

- 每次只输出一对Thought-Action
- Action必须在同一行，不要换行
- 当收集到足够信息可以回答用户问题时，必须使用 `Action: Finish[最终答案]` 格式结束

请开始吧！

```
"""
```

(2) 工具 1：查询真实天气

我们将使用免费的天气查询服务 `wttr.in`，它能以 JSON 格式返回指定城市的天气数据。下面是实现该工具的代码：

```
import requests

def get_weather(city: str) -> str:
    """
    通过调用 wttr.in API 查询真实的天气信息。
    """

    # API端点，我们请求JSON格式的数据
    url = f"https://wttr.in/{city}?format=j1"

    try:
        # 发起网络请求
        response = requests.get(url)
        # 检查响应状态码是否为200 (成功)
        response.raise_for_status()
        # 解析返回的JSON数据
        data = response.json()

        # 提取当前天气状况
        current_condition = data['current_condition'][0]
        weather_desc = current_condition['weatherDesc'][0]['value']
        temp_c = current_condition['temp_C']

        # 格式化成自然语言返回
        return f"{city}当前天气:{weather_desc}, 气温{temp_c}摄氏度"
    except requests.exceptions.RequestException as e:
        # 处理网络错误
        return f"错误:查询天气时遇到网络问题 - {e}"
    except (KeyError, IndexError) as e:
        # 处理数据解析错误
        return f"错误:解析天气数据失败, 可能是城市名称无效 - {e}"
```

(3) 工具 2：搜索并推荐旅游景点

我们将定义一个新工具 `search_attraction`，它会根据城市和天气状况，互联网上搜索合适的景点：

```
import os
from tavily import TavilyClient
```

```

def get_attraction(city: str, weather: str) -> str:
    """
    根据城市和天气，使用Tavily Search API搜索并返回优化后的景点推荐。
    """

    # 1. 从环境变量中读取API密钥
    api_key = os.environ.get("TAVILY_API_KEY")
    if not api_key:
        return "错误:未配置TAVILY_API_KEY环境变量。"

    # 2. 初始化Tavily客户端
    tavily = TavilyClient(api_key=api_key)

    # 3. 构造一个精确的查询
    query = f'{city} 在{weather}天气下最值得去的旅游景点推荐及理由'

    try:
        # 4. 调用API, include_answer=True会返回一个综合性的回答
        response = tavily.search(query=query, search_depth="basic", include_answer=True)

        # 5. Tavily返回的结果已经非常干净，可以直接使用
        # response['answer'] 是一个基于所有搜索结果的总结性回答
        if response.get("answer"):
            return response["answer"]

        # 如果没有综合性回答，则格式化原始结果
        formatted_results = []
        for result in response.get("results", []):
            formatted_results.append(f"- {result['title']}: {result['content']}")

        if not formatted_results:
            return "抱歉，没有找到相关的旅游景点推荐。"

        return "根据搜索，为您找到以下信息:\n" + "\n".join(formatted_results)

    except Exception as e:
        return f"错误:执行Tavily搜索时出现问题 - {e}"

```

最后，我们将所有工具函数放入一个字典，供主循环调用：

```

# 将所有工具函数放入一个字典，方便后续调用
available_tools = {
    "get_weather": get_weather,
    "get_attraction": get_attraction,
}

```

1.3.2 接入大语言模型

当前，许多 LLM 服务提供商（包括 OpenAI、Azure、以及众多开源模型服务框架如 Ollama、vLLM 等）都遵循了与 OpenAI API 相似的接口规范。这种标准化为开发者带来了极大的便利。智能体的自主决策能力来源于 LLM。我们将实现一个通用的客户端 OpenAICompatibleClient，它可以连接到任何兼容 OpenAI 接口规范的 LLM 服务。

```

from openai import OpenAI

class OpenAICompatibleClient:
    """
    一个用于调用任何兼容OpenAI接口的LLM服务的客户端。
    """

    def __init__(self, model: str, api_key: str, base_url: str):
        self.model = model
        self.client = OpenAI(api_key=api_key, base_url=base_url)

    def generate(self, prompt: str, system_prompt: str) -> str:
        """
        调用LLM API来生成回应。
        """
        print("正在调用大语言模型...")
        try:
            messages = [
                {'role': 'system', 'content': system_prompt},
                {'role': 'user', 'content': prompt}
            ]
            response = self.client.chat.completions.create(
                model=self.model,
                messages=messages,
                stream=False
            )
            answer = response.choices[0].message.content
            print("大语言模型响应成功。")
            return answer
        except Exception as e:
            print(f"调用LLM API时发生错误: {e}")
            return "错误:调用语言模型服务时出错。"

```

要实例化此类，您需要提供三个信息：API_KEY、BASE_URL 和 MODEL_ID，具体值取决于您使用的服务商（如 OpenAI 官方、Azure、或 Ollama 等本地模型），如果暂时没有渠道获取，可以参考 [环境配置](#)。

1.3.3 执行行动循环

下面的主循环将整合所有组件，并通过格式化后的 Prompt 驱动 LLM 进行决策。

2/9/26, 1:56 PM

```
import re

# --- 1. 配置LLM客户端 ---
# 请根据您使用的服务，将这里替换成对应的凭证和地址
API_KEY = "YOUR_API_KEY"
BASE_URL = "YOUR_BASE_URL"
MODEL_ID = "YOUR_MODEL_ID"
TAVILY_API_KEY="YOUR_Tavily_KEY"
os.environ['TAVILY_API_KEY'] = "YOUR_TAVILY_API_KEY"

llm = OpenAICompatibleClient(
    model=MODEL_ID,
    api_key=API_KEY,
    base_url=BASE_URL
)

# --- 2. 初始化 ---
user_prompt = "你好，请帮我查询一下今天北京的天气，然后根据天气推荐一个合适的旅游景点。"
prompt_history = [f"用户请求: {user_prompt}"]

print(f"用户输入: {user_prompt}\n" + "="*40)

# --- 3. 运行主循环 ---
for i in range(5): # 设置最大循环次数
    print(f"--- 循环 {i+1} ---\n")

    # 3.1. 构建Prompt
    full_prompt = "\n".join(prompt_history)

    # 3.2. 调用LLM进行思考
    llm_output = llm.generate(full_prompt, system_prompt=AGENT_SYSTEM_PROMPT)
    # 模型可能会输出多余的Thought-Action，需要截断
    match = re.search(r'(Thought:.*?Action:.*?)(?:\n\s*(?:Thought:|Action:|Observation:)|\Z)', llm_output, re.DOTALL)
    if match:
        truncated = match.group(1).strip()
        if truncated != llm_output.strip():
            llm_output = truncated
            print("已截断多余的 Thought-Action 对")
    print(f"模型输出:\n{llm_output}\n")
    prompt_history.append(llm_output)

    # 3.3. 解析并执行行动
    action_match = re.search(r"Action: (.*)", llm_output, re.DOTALL)
    if not action_match:
        observation = "错误：未能解析到 Action 字段。请确保你的回复严格遵循 'Thought: ... Action: ...' 的格式。"
        observation_str = f"Observation: {observation}"
        print(f"{observation_str}\n" + "="*40)
        prompt_history.append(observation_str)
        continue
    action_str = action_match.group(1).strip()

    if action_str.startswith("Finish"):
        final_answer = re.match(r"Finish\[([.]*\]", action_str).group(1)
        print(f"任务完成，最终答案: {final_answer}")
        break

    tool_name = re.search(r"\w+\(", action_str).group(1)
    args_str = re.search(r"\((.*)\)", action_str).group(1)
    kwargs = dict(re.findall(r'(\w+)=(["^"]*)"', args_str))

    if tool_name in available_tools:
        observation = available_tools[tool_name](**kwargs)
    else:
        observation = f"错误:未定义的工具 '{tool_name}'"

    # 3.4. 记录观察结果
    observation_str = f"Observation: {observation}"
    print(f"{observation_str}\n" + "="*40)
    prompt_history.append(observation_str)
```

通过以上步骤，我们构建了一个完整的、由真实 LLM 驱动的智能体。其核心在于“工具”和“提示工程”的结合，这正是当前主流智能体框架（如 LangChain、LlamaIndex 等）的设计精髓。

1.3.4 运行案例分析

以下输出完整地展示了一个成功的智能体执行流程。通过对这个三轮循环的分析，我们可以清晰地看到智能体解决问题的核心能力。

用户输入：你好，请帮我查询一下今天北京的天气，然后根据天气推荐一个合适的旅游景点。

=====

--- 循环 1 ---

正在调用大语言模型...

大语言模型响应成功。

模型输出：

Thought: 首先需要获取北京今天的天气情况，之后再根据天气情况来推荐旅游景点。

Action: get_weather(city="北京")

Observation: 北京当前天气:Sunny，气温26摄氏度

=====

--- 循环 2 ---

正在调用大语言模型...

大语言模型响应成功。

模型输出:

Thought: 现在已经知道了北京今天的天气是晴朗且温度适中，接下来可以基于这个信息来推荐一个适合的旅游景点了。

Action: get_attraction(city="北京", weather="Sunny")

Observation: 北京在晴天最值得去的旅游景点是颐和园，因其美丽的湖景和古建筑。另一个推荐是长城，因其壮观的景观和历史意义。

=====

--- 循环 3 ---

正在调用大语言模型...

大语言模型响应成功。

模型输出:

Thought: 已经获得了两个适合晴天游览的景点建议，现在可以根据这些信息给用户提供满意的答复。

Action: Finish [今天北京的天气是晴朗的，气温26摄氏度，非常适合外出游玩。我推荐您去颐和园欣赏美丽的湖景和古建筑，或者前往长城体验其壮观的景观和深厚的历史意义。希望您有一

任务完成，最终答案：今天北京的天气是晴朗的，气温26摄氏度，非常适合外出游玩。我推荐您去颐和园欣赏美丽的湖景和古建筑，或者前往长城体验其壮观的景观和深厚的历史意义。希望您有

这个简单的旅行助手案例，集中演示了基于Thought-Action-Observation范式的智能体所具备的四项基本能力：任务分解、工具调用、上下文理解和结果合成。正是通过这个循环的不断迭代，智能体才得以将一个模糊的用户意图，转化为一系列具体、可执行的步骤，并最终达成目标。

1.4 智能体应用的协作模式

上一节，我们通过亲手构建一个智能体，深入理解了其内部的运作循环。不过在更广泛的应用场景中，我们的角色正越来越多地转变为使用者与协作者。基于智能体在任务中的角色和自主性程度，其协作模式主要分为两种：一种是作为高效工具，深度融入我们的工作流；另一种则是作为自主的协作者，与其他智能体协作完成复杂目标。

1.4.1 作为开发者工具的智能体

在这种模式下，智能体被深度集成到开发者的工作流中，作为一种强大的辅助工具。它增强而非取代开发者的角色，通过自动化处理繁琐、重复的任务，让开发者能更专注于创造性的核心工作。这种人机协同的方式，极大地提升了软件开发的效率与质量。

目前，市场上涌现了多款优秀的 AI 编程辅助工具，它们虽然均能提升开发效率，但在实现路径和功能侧重上各有千秋：

- **GitHubCopilot**: 作为该领域最具影响力的产品之一，Copilot 由 GitHub 与 OpenAI 联合开发。它深度集成于 Visual Studio Code 等主流编辑器中，以其强大的代码自动补全能力而闻名。开发者在编写代码时，Copilot 能实时提供整行甚至整个函数块的建议。近年来，它也通过 Copilot Chat 扩展了对话式编程的能力，允许开发者在编辑器内通过聊天解决编程问题。
- **Claude Code**: Claude Code 是由 Anthropic 开发的 AI 编程助手，旨在通过自然语言指令帮助开发者在终端中高效地完成编码任务。它能够理解完整的代码库结构、执行代码编辑、测试和调试等操作，支持从描述功能到代码实现的全流程开发。Claude Code 还提供了无交互（headless）模式，适用于 CI、pre-commit hooks、构建脚本和其他自动化场景，为开发者提供了强大的命令行编程体验。
- **Trae**: 作为新兴的 AI 编程工具，Trae 专注于为开发者提供智能化的代码生成和优化服务。它通过深度学习技术分析代码模式，能够为开发者提供精准的代码建议和自动化重构方案。Trae 的特色在于其轻量级的设计和快速响应能力，特别适合需要频繁迭代和快速原型开发的场景。
- **Cursor**: 与上述主要作为插件或集成功能存在的工具不同，Cursor 则选择了一条更具整合性的路径，它本身就是一个 AI 原生的代码编辑器。它并非在现有编辑器上增加 AI 功能，而是在设计之初就将 AI 交互作为核心。除了具备顶级的代码生成和聊天能力外，它更强调让 AI 理解整个代码库的上下文，从而实现更深层次的问答、重构和调试。

当然还有许多优秀的工具没有例举，不过它们共同指向了一个明确的趋势：AI 正在深度融入软件开发的全生命周期，通过构建高效的人机协同工作流，深刻地重塑着软件工程的效率边界与开发范式。

1.4.2 作为自主协作者的智能体

与作为工具辅助人类不同，第二种交互模式将智能体的自动化程度提升到了一个全新的层次，自主协作者。在这种模式下，我们不再是手把手地指导 AI 完成每一步，而是将一个高级级的目标委托给它。智能体会像一个真正的项目成员一样，独立地进行规划、推理、执行和反思，直到最终交付成果。这种从助手到协作者的转变，使得 LLM 智能体更深的进入了大众的视野。它标志着我们与 AI 的关系从“命令-执行”演变为“目标-委托”。智能体不再是被动的工具，而是主动的目标追求者。

当前，实现这种自主协作的思路百花齐放，涌现了大量优秀的框架和产品，从早期的 BabyAGI、AutoGPT，到如今更为成熟的 CrewAI、AutoGen、MetaGPT、LangGraph 等优秀框架，共同推动着这一领域的高速发展。虽然具体实现千差万别，但它们的架构范式大致可以归纳为几个主流方向：

1. 单智能体自主循环：这是早期的典型范式，如 **AgentGPT** 所代表的模式。其核心是一个通用智能体通过“思考-规划-执行-反思”的闭环，不断进行自我提示和迭代，以完成一个开放式的高级级目标。
2. 多智能体协作：这是当前最主流的探索方向，旨在通过模拟人类团队的协作模式来解决复杂问题。它又可细分为不同模式：角色扮演式对话：如 **CAMEL** 框架，通过为两个智能体（例如，“程序员”和“产品经理”）设定明确的角色和沟通协议，让它们在一个结构化的对话中协同完成任务。组织化工作流：如 **MetaGPT** 和 **CrewAI**，它们模拟一个分工明确的“虚拟团队”（如软件公司或咨询小组）。每个智能体都有预设的职责和工作流程（SOP），通过层级化或顺序化的方式协作，产出高质量的复杂成果（如完整的代码库或研究报告）。**AutoGen** 和 **AgentScope** 则提供了更灵活的对话模式，允许开发者自定义智能体间的复杂交互网络。
3. 高级控制流架构：诸如 **LangGraph** 等框架，则更侧重于为智能体提供更强大的底层工程基础。它将智能体的执行过程建模为状态图（State Graph），从而能更灵活、更可靠地实现循环、分支、回溯以及人工介入等复杂流程。

这些不同的架构范式，共同推动着自主智能体从理论构想走向更广泛的实际应用，使其有能力应对日益复杂的真实世界任务。在我们的后续章节中，也会感受不同类型框架之间的差异和优势。

1.4.3 Workflow 和 Agent 的差异

在理解了智能体作为“工具”和“协作者”两种模式后，我们有必要对 Workflow 和 Agent 的差异展开讨论，尽管它们都旨在实现任务自动化，但其底层逻辑、核心特征和适用场景却截然不同。

简单来说，Workflow 是让 AI 按部就班地执行指令，而 Agent 则是赋予 AI 自由度去自主达成目标。

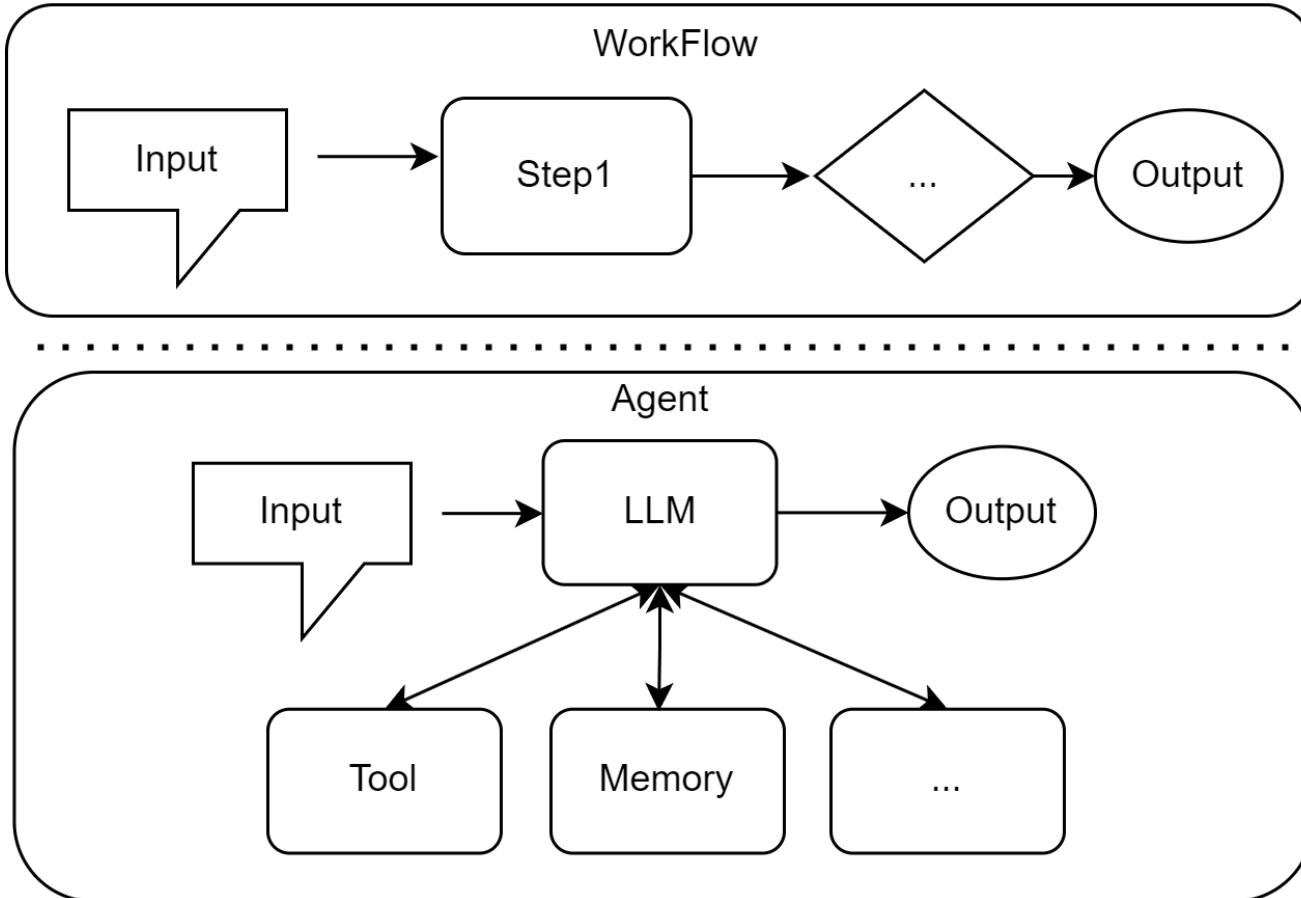


图 1.6 Workflow 和 Agent 的差异

如图 1.6 所示，工作流是一种传统的自动化范式，其核心是对一系列任务或步骤进行预先定义的、结构化的编排。它本质上是一个精确的、静态的流程图，规定了在何种条件下、以何种顺序执行哪些操作。一个典型的案例：某企业的费用报销审批流程。员工提交报销单（触发）-> 如果金额小于 500 元，直接由部门经理审批 -> 如果金额大于 500 元，先由部门经理审批，再流转至财务总监审批 -> 审批通过后，通知财务部打款。整个过程的每一步、每一个判断条件都被精确地预先设定。

与工作流不同，基于大型语言模型的智能体是一个具备自主性的、以目标为导向的系统。它不仅仅是执行预设指令，而是能够在一定程度上理解环境、进行推理、制定计划，并动态地采取行动以达成最终目标。LLM 在其中扮演着“大脑”的角色。一个典型的例子，便是我们在 1.3 节中写的智能旅行助手。当我们向它下达一个新指令，例如：“你好，请帮我查询一下今天北京的天气，然后根据天气推荐一个合适的旅游景点。”它的处理过程充分展现了其自主性：

1. 规划与工具调用：Agent 首先会把任务拆解为两个步骤：① 查询天气；② 基于天气推荐景点。随即，它会自主选择并调用“天气查询 API”，并将“北京”作为参数传入。
2. 推理与决策：假设 API 返回结果为“晴朗，微风”。Agent 的 LLM 大脑会基于这个信息进行推理：“晴天适合户外活动”。接着，它会根据这个判断，在它的知识库或通过搜索引擎这个工具中，筛选出北京的户外景点，如故宫、颐和园、天坛公园等。
3. 生成结果：最后，Agent 会综合信息，给出一个完整的、人性化的回答：“今天北京天气晴朗，微风，非常适合户外游玩。为您推荐前往【颐和园】，您可以在昆明湖上泛舟，欣赏美丽的皇家园林景色。”

在这个过程中，没有任何写死的 if 天气=晴天 then 推荐颐和园的规则。如果天气是“雨天”，Agent 会自主推理并推荐国家博物馆、首都博物馆等室内场所。这种基于实时信息进行动态推理和决策的能力，正是 Agent 的核心价值所在。

1.4 本章小结

在本章中，我们共同踏上了探索智能体的初识之旅。我们的旅程从最基本的问题开始：

- 什么是大语言模型驱动的智能体？我们首先明确了其定义，理解了现代智能体是具备了能力的实体。它不再仅仅是执行预设程序的脚本，而是能够自主推理和使用工具的决策者。

- 智能体如何工作？我们深入探讨了智能体与环境交互的运行机制。我们了解到，这个持续的闭环是智能体处理信息、做出决策、影响环境并根据反馈调整自身行为的基础。
- 如何构建智能体？这是本章的实践核心。我们以“智能旅行助手”为例，亲手构建了一个完整的、由真实 LLM 驱动的智能体。
- 智能体有哪些主流的应用范式？最后，我们将视野投向了更广阔的应用领域。我们探讨了两种主流的智能体交互模式：一是以 GitHub Copilot 和 Cursor 等为代表的、增强人类工作流的“开发者工具”；二是以 CrewAI、MetaGPT 和 AgentScope 等框架为代表的、能够独立完成高层次目标的“自主协作者”。同时讲解了 Workflow 与 Agent 的差异。

通过本章的学习，我们建立了一个关于智能体的基础认知框架。那么，它是如何一步步从最初的构想演进至今的呢？在下一章中，我们将探索智能体的发展历史，一段追本溯源的旅程即将开始！

习题

提示：以下的部分习题没有标准答案，重点在于培养学习者对智能体系统批判性的深入思考和动手实践能力。

1. 请分析以下四个 case 中的主体是否属于智能体，如果是，那么属于哪种类型的智能体（可以从多个分类维度进行分析），并说明理由：

case A：一台符合冯·诺依曼结构的超级计算机，拥有高达每秒 2EFlop 的峰值算力

case B：特斯拉自动驾驶系统在高速公路上行驶时，突然检测到前方有障碍物，需要在毫秒级做出刹车或变道决策

case C：AlphaGo 在与人类棋手对弈时，需要评估当前局面并规划未来数十步的最优策略

case D：ChatGPT 扮演的智能客服在处理用户投诉时，需要查询订单信息、分析问题原因、提供解决方案并安抚用户情绪

2. 假设你需要为一个“智能健身教练”设计任务环境。这个智能体能够：

- 通过可穿戴设备监测用户的心率、运动强度等生理数据
- 根据用户的健身目标（减脂/增肌/提升耐力）动态调整训练计划
- 在用户运动过程中提供实时语音指导和动作纠正
- 评估训练效果并给出饮食建议

请使用 PEAS 模型完整描述这个智能体的任务环境，并分析该环境具有哪些特性（如部分可观察、随机性、动态性等）。

3. 某电商公司正在考虑两种方案来处理售后退款申请：

方案 A (Workflow)：设计一套固定流程，例如：

A.1 对于一般商品且在 7 天之内，金额 < 100RMB 自动通过；100–500RMB 由客服审核；>500RMB 需主管审批；而特殊商品（如定制品）一律拒绝退款

A.2 对于超过 7 天的商品，无论金额，只能由客服审核或主管审批；

方案 B (Agent)：搭建一个智能体系统，让它理解退款政策、分析用户历史行为、评估商品状况，并自主决策是否批准退款

请分析：

- 这两种方案各自的优缺点是什么？
- 在什么情况下 Workflow 更合适？什么情况下 Agent 更有优势？如果你是该电商公司的负责人，你更倾向于采用哪种方案？
- 是否存在一个方案 C，能够结合两种方案，达到扬长避短的效果？

4. 在 1.3 节的智能旅行助手基础上，请思考如何添加以下功能（可以只描述设计思路，也可以进一步尝试代码实现）：

提示：思考如何修改 Thought–Action–Observation 循环来实现这些功能。

- 添加一个“记忆”功能，让智能体记住用户的偏好（如喜欢历史文化景点、预算范围等）
- 当推荐的景点门票已售罄时，智能体能够自动推荐备选方案
- 如果用户连续拒绝了 3 个推荐，智能体能够反思并调整推荐策略

5. 卡尼曼的“系统 1”（快速直觉）和“系统 2”（慢速推理）理论^[2]为神经符号主义 AI 提供了很好的类比。请首先构思一个具体的智能体的落地应用场景，然后说明场景中的：

提示：医疗诊断助手、法律咨询机器人、金融风控系统等都是常见的应用场景

- 哪些任务应该由“系统 1”处理？
- 哪些任务应该由“系统 2”处理？
- 这两个系统如何协同工作以达成最终目标？

6. 尽管大语言模型驱动的智能体系统展现出了强大的能力，但它们仍然存在诸多局限。请分析以下问题：

- 为什么智能体或智能体系统有时会产生“幻觉”（生成看似合理但实际错误的信息）？
- 在 1.3 节的案例中，我们设置了最大循环次数为 5 次。如果没有这个限制，智能体可能会陷入什么问题？
- 如何评估一个智能体的“智能”程度？仅使用准确率指标是否足够？

参考文献

[1] RUSSELL S, NORVIG P. Artificial Intelligence: A Modern Approach[M]. 4th ed. London: Pearson, 2020.

[2] KAHNEMAN D. Thinking, Fast and Slow[M]. New York: Farrar, Straus and Giroux, 2011.

讨论与交流

本章学习过程中遇到问题?想与其他学习者交流心得?

 前往 GitHub Discussions 讨论区: - [习题讨论与问答](#) - 在这里你可以: - 提问习题相关问题 - 分享你的解题思路 - 与其他学习者交流经验 - 获得社区的帮助和反馈

 提示: 每个页面底部也有评论区,可以直接在页面内讨论!

第二章 智能体发展史

为了深刻理解现代智能体为何呈现出如今的形态，以及其核心设计思想的由来，本章将回溯历史：从人工智能领域的古典时代出发，探寻最早的“智能”如何在逻辑与符号的规则体系中被定义；继而见证从单一、集中的智能模型到分布式、协作式智能思想的重大转折；最终理解“学习”范式如何彻底改变了智能体获取能力的方式，并催生出我们今天所见的现代智能体。

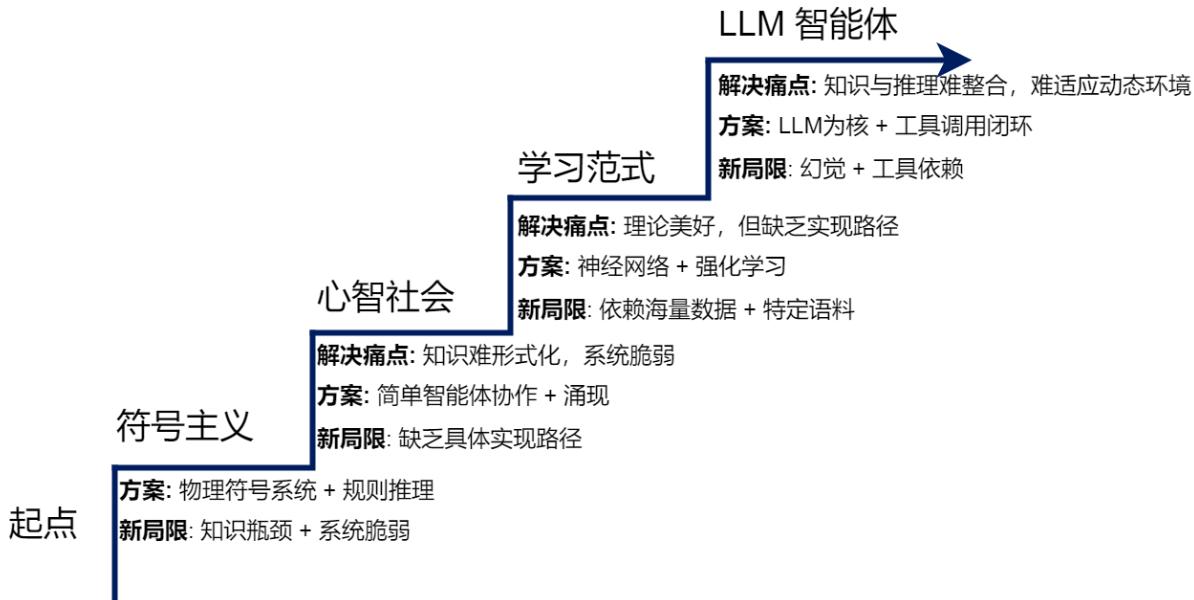


图 2.1 AI 智能体的演进阶梯

如图2.1所示，每一个新范式的出现，都是为了解决上一代范式的核心“痛点”或根本局限。而新的解决方案在带来能力飞跃的同时，也引入了新的、在当时难以克服的“局限”，而这又为下一代范式的诞生埋下了伏笔。理解这一“问题驱动”的迭代历程，能帮助我们更深刻地把握现代智能体技术选型背后的深层原因与历史必然性。

2.1 基于符号与逻辑的早期智能体

人工智能领域的早期探索，深受数理逻辑和计算机科学基本原理的影响。在那个时代，研究者们普遍持有一种信念：人类的智能，尤其是逻辑推理能力，可以被形式化的符号体系所捕捉和复现。这一核心思想催生了人工智能的第一个重要范式——符号主义（Symbolicism），也被称为“逻辑AI”或“传统AI”。

在符号主义看来，智能行为的核心是基于一套明确规则对符号进行操作。因此，一个智能体可以被视为一个物理符号系统：它通过内部的符号来表示外部世界，并通过逻辑推理来规划行动。这个时代的智能体，其“智慧”完全来源于设计者预先编码的知识库和推理规则，而非通过自主学习获得。

2.1.1 物理符号系统假说

符号主义时代的理论根据，是1976年由艾伦·纽厄尔（Allen Newell）和赫伯特·西蒙（Herbert A. Simon）共同提出的物理符号系统假说（Physical Symbol System Hypothesis, PSSH）^[1]。这两位图灵奖得主通过这一假说，为在计算机上实现通用人工智能提供了理论指导和判定标准。

该假说包含两个核心论断：

1. 充分性论断：任何一个物理符号系统，都具备产生通用智能行为的充分手段。
2. 必要性论断：任何一个能够展现通用智能行为的系统，其本质必然是一个物理符号系统。

这里的物理符号系统指的是一个能够在物理世界中存在的系统，它由一组可被区分的符号和一系列对这些符号进行操作的过程组成，其构成元素如图2.2所示。这些符号可以组合成更复杂的结构（例如表达式），而过程则可以创建、修改、复制和销毁这些符号结构。

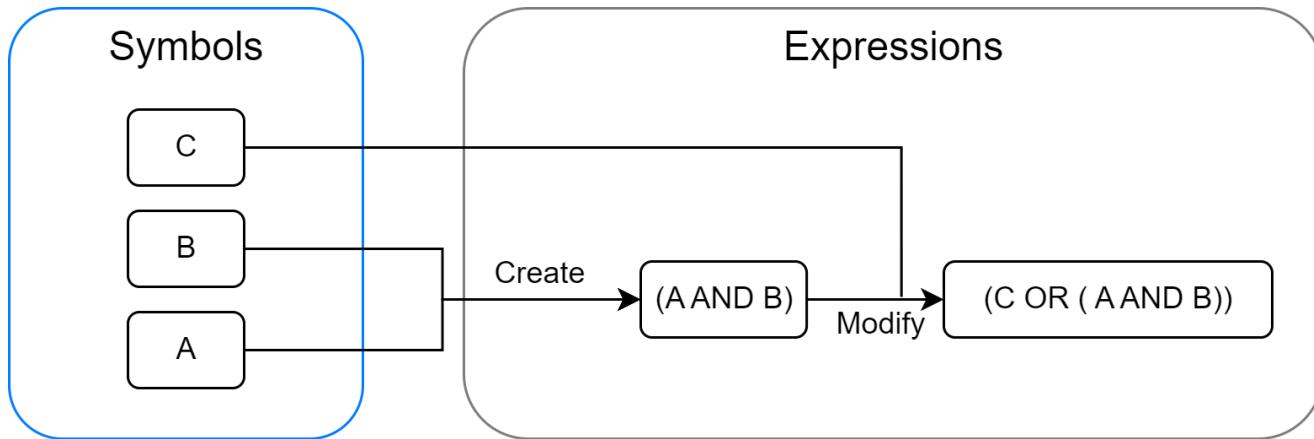


图 2.2 物理符号系统的构成元素

简而言之，PSSH大胆地宣称：智能的本质，就是符号的计算与处理。

这个假说具有深远的影响。它将对人类心智这一模糊、复杂的哲学问题的研究，转化为了一个可以在计算机上进行工程化实现的具体问题。它为早期人工智能研究者注入了强大的信心，即只要我们能找到正确的方式来表示知识并设计出有效的推理算法，就一定能创造出与人类媲美的机器智能。整个符号主义时代的研究，从专家系统到自动规划，几乎都是在这一假说的指引下展开的。

2.1.2 专家系统

在物理符号系统假说的直接影响下，专家系统（Expert System）成为符号主义时代最重要、最成功的应用成果。专家系统的核心目标，是模拟人类专家在特定领域内解决问题的能力。它通过将专家的知识和经验编码成计算机程序，使其能够在面对相似问题时，给出媲美甚至超越人类专家的结论或建议。

一个典型的专家系统通常由知识库、推理机、用户界面等几个核心部分构成，其通用架构如图2.3所示。

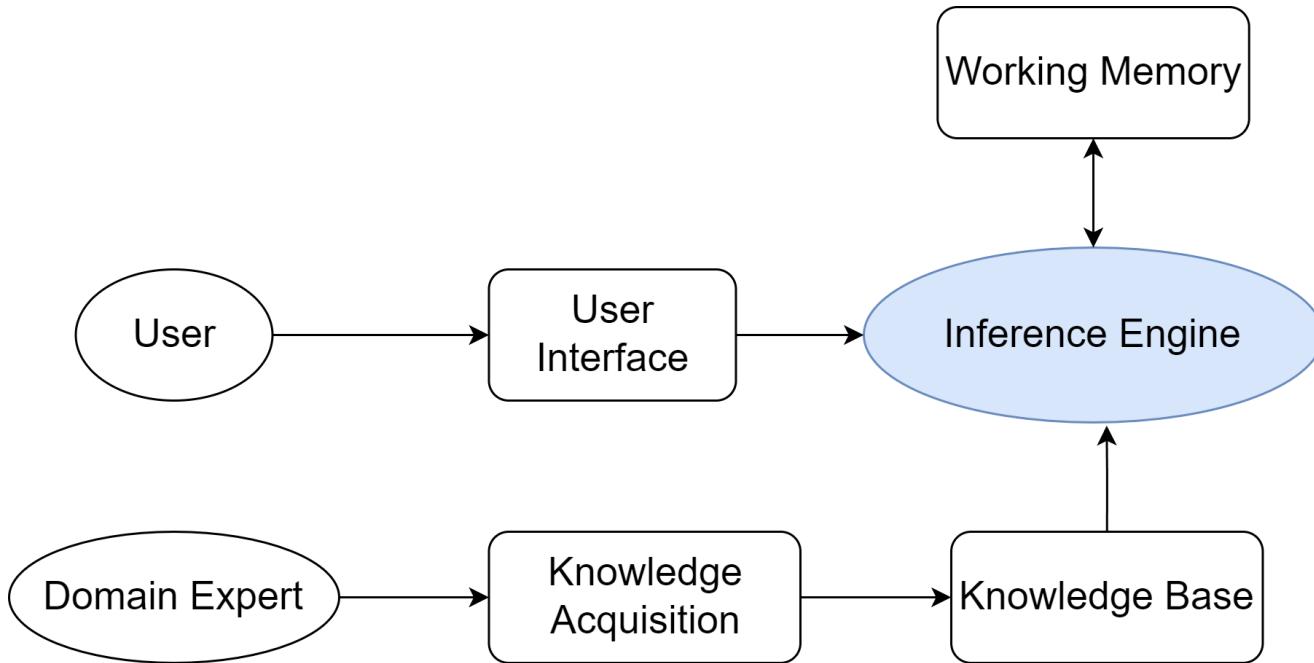


图 2.3 专家系统的通用架构

这种架构清晰地体现了知识与推理相分离的设计思想，是符号主义AI的重要特征。

知识库与推理机

专家系统的“智能”主要源于其两大核心组件：知识库和推理机。

- **知识库（Knowledge Base）**：这是专家系统的知识存储中心，用于存放领域专家的知识和经验。知识表示（Knowledge Representation）是构建知识库的关键。在专家系统中，最常用的一种知识表示方法是产生式规则（Production Rules），即一系列“IF-THEN”形式的条件语句。例如：IF 病人有发烧症状 AND 咳嗽 THEN 可能患有呼吸道感染。这些规则将特定情境（IF部分，条件）与相应的结论或行动（THEN部分，结论）关联起来。一个复杂的专家系统可能包含成百上千条这样的规则，共同构成一个庞大的知识网络。
- **推理机（Inference Engine）**：推理机是专家系统的核心计算引擎。它是一个通用的程序，其任务是根据用户提供的事实，在知识库中寻找并应用相关的规则，从而推导出新的结论。推理机的工作方式主要有两种：

- 正向链 (**Forward Chaining**)：从已知事实出发，不断匹配规则的IF部分，触发THEN部分的结论，并将新结论加入事实库，直到最终推导出目标或无新规则可匹配。这是一种“数据驱动”的推理方式。
- 反向链 (**Backward Chaining**)：从一个假设的目标（比如“病人是否患有肺炎”）出发，寻找能够推导出该目标的规则，然后将该规则的IF部分作为新的子目标，如此递归下去，直到所有子目标都能被已知事实所证明。这是一种“目标驱动”的推理方式。

应用案例与分析：MYCIN系统

MYCIN是历史上最著名、最具影响力的专业系统之一，由斯坦福大学于20世纪70年代开发^[2]。它被设计用于辅助医生诊断细菌性血液感染并推荐合适的抗生素治疗方案。

- 工作原理：MYCIN通过与医生进行问答式交互来收集病人的症状、病史和化验结果。其知识库包含了约600条由医学专家提供的“IF-THEN”规则。推理机主要采用反向链的方式工作：从“确定致病菌”这一最高目标出发，反向推导需要哪些证据和条件，然后向医生提问以获取这些信息。其简化的工作流程如图2.4所示。

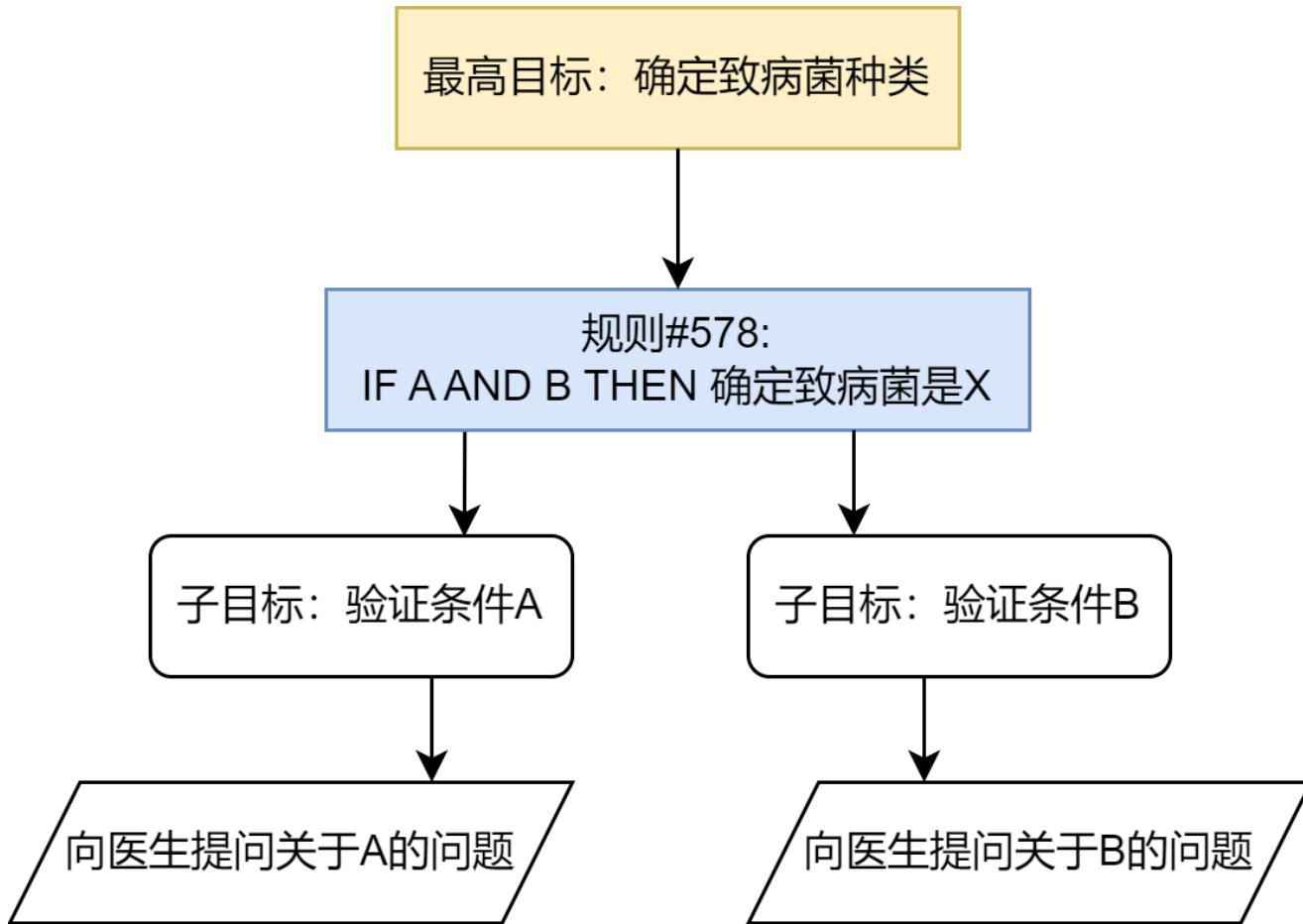


图 2.4 MYCIN反向链推理流程示意图

- 不确定性处理：医学诊断充满了不确定性。MYCIN的一个重要创新是引入了置信因子 (Certainty Factor, CF) 的概念，用一个-1到1之间的数值来表示一个结论的可信度。这使得系统能够处理不确定的、模糊的医学知识，并给出带有可信度评估的诊断结果，这比简单的布尔逻辑更贴近现实世界。
- 成就与意义：在一项评估中，MYCIN在血液感染诊断方面的表现超过了非专业医生，甚至达到了人类专家的水平。它的成功雄辩地证明了物理符号系统假说的有效性：通过精心的知识工程和符号推理，机器确实可以在高度复杂的专业领域展现出卓越的“智能”。MYCIN不仅是专家系统发展史上的一個里程碑，也为后续人工智能在各个垂直领域的商业化应用铺平了道路。

2.1.3 SHRDLU

如果说专家系统展示了符号AI在专业领域的“深度”，那么由特里·威诺格拉德 (Terry Winograd) 于1968-1970年开发的SHRDLU项目^[3]，则在“广度”上实现了革命性的突破。如图2.5所示，SHRDLU旨在构建一个能在“积木世界”这一微观环境中，通过自然语言与人类流畅交互的综合性智能体。“积木世界”是一个模拟的三维虚拟空间，其中包含不同形状、颜色和大小的积木，以及一个可以抓取和移动它们的虚拟机械臂。用户通过自然语言向SHRDLU下达指令或提问，SHRDLU则在虚拟世界中执行动作或给出文字回答。

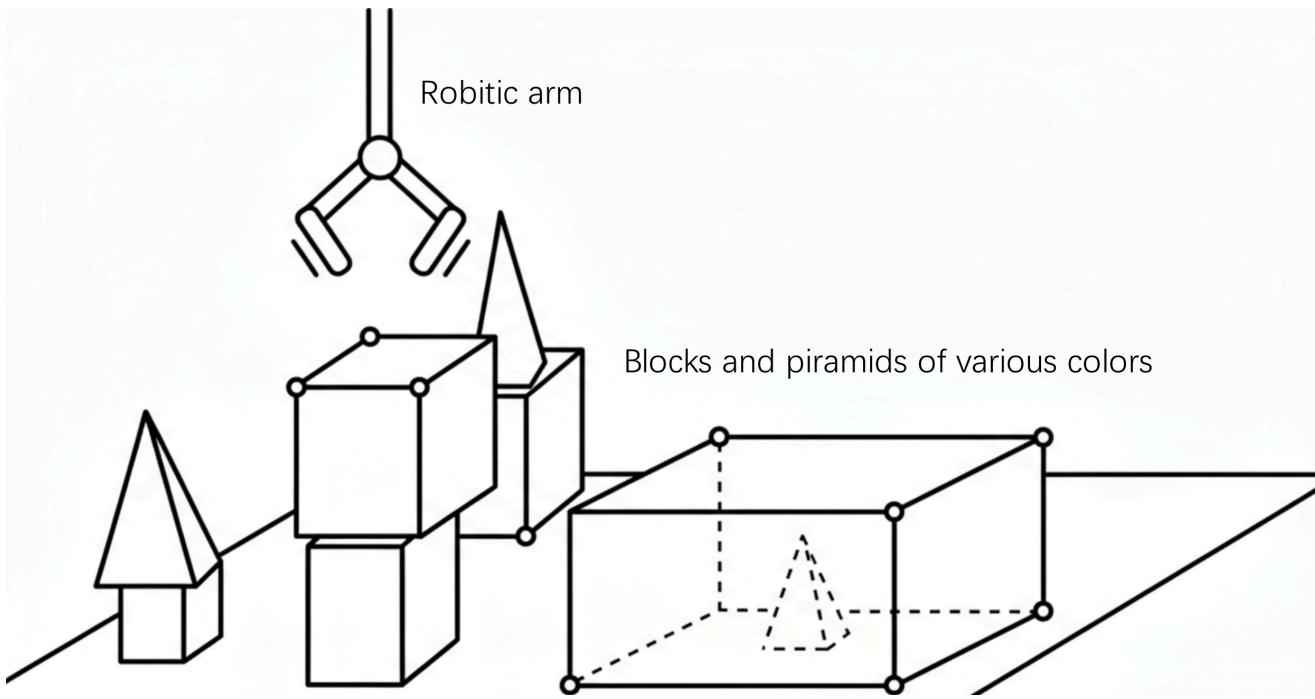


图 2.5 SHRDLU的“积木世界”交互界面

SHRDLU在当时引起广泛关注，主要原因在于它首次将多个独立的人工智能模块（如语言解析、规划、记忆）集成在一个统一的系统中，并使它们协同工作：

- 自然语言理解：SHRDLU能够解析结构复杂且含有歧义的英语句子。它不仅能理解直接的命令（如 `Pick up a big red block.`），还能处理更复杂的指令，例如：
 - 指代消解：`Find a block which is taller than the one you are holding and put it into the box.` 在这条指令中，系统需要理解 `the one you are holding` 指代的是当前机械臂正抓取的物体。
 - 上下文记忆：用户可以说 `Grasp the pyramid.`，然后接着问 `What does the box contain?`，系统能够联系上下文进行回答。
- 规划与行动：在理解指令后，SHRDLU能够自主规划出一系列必要的动作来完成任务。例如，如果指令是“把蓝色积木放到红色积木上”，而红色积木上已经有另一个绿色积木，系统会规划出“先把绿色积木移开，再把蓝色积木放上去”的动作序列。
- 记忆与问答：SHRDLU拥有关于其所处环境和自身行为的记忆。用户可以就此提问，例如：
 - 询问世界状态：`Is there a large block behind a pyramid?`
 - 询问行为历史：`Did you touch any pyramid before you put the green one on the little cube?`
 - 询问行为动机：`Why did you pick up the red block?` SHRDLU可以回答：`BECAUSE YOU ASKED ME TO.`

SHRDLU的历史地位与影响主要体现在三个方面：

- 综合性智能的典范：在SHRDLU之前，AI研究大多聚焦于单一功能。它首次将语言理解、推理规划与行动记忆等多个AI模块集成于统一系统，其“感知-思考-行动”的闭环设计，奠定了现代智能体研究的基础。
- 微观世界研究方法的普及：它的成功证明了在一个规则明确的简化环境中，探索和验证复杂智能体基本原理的可行性，这一方法深刻影响了后续的机器学习与AI规划研究。
- 引发的乐观与反思：SHRDLU的成功激发了对AGI的早期乐观预期，但其能力又严格局限于积木世界。这种局限性引发了AI领域关于“符号处理”与“真正理解”之间差异的长期思辨，揭示了通往通用智能的深层挑战。

2.1.4 符号主义面临的根本性挑战

尽管早期项目成就显著，但从20世纪80年代起，符号主义AI在从“微观世界”走向开放、复杂的现实世界时，遇到了其方法论固有的根本性难题。这些难题主要可归结为两大类：

(1) 常识知识与知识获取瓶颈

符号主义智能体的“智能”完全依赖于其知识库的质量和完备性。然而，如何构建一个能够支撑真实世界交互的知识库，被证明是一项极其艰巨的任务，主要体现在两个方面：

- 知识获取瓶颈（Knowledge Acquisition Bottleneck）：专家系统的知识需要由人类专家和知识工程师通过繁琐的访谈、提炼和编码过程来构建。这个过程成本高昂、耗时漫长，且难以规模化。更重要的是，人类专家的许多知识是内隐的、直觉性的，很难被清晰地表达为“IF-THEN”规则。试图将整个世界的知识都进行手工符号化，被认为是一项几乎不可能完成的任务。
- 常识问题（Common-sense Problem）：人类行为依赖于庞大的常识背景（例如，“水是湿的”、“绳子可以拉不能推”），但符号系统除非被明确编码，否则对此一无所知。为广阔、模糊的常识建立完备的知识库至今仍是重大挑战，Cyc项目^[4]历经数十年努力，其成果和应用仍然非常有限。

(2) 框架问题与系统脆弱性

除了知识层面的挑战，符号主义在处理动态变化的世界时也遇到了逻辑上的困境。

- **框架问题 (Frame Problem)**：在一个动态世界中，智能体执行一个动作后，如何高效判断哪些事物未发生改变是一个逻辑难题^[5]。为每个动作显式地声明所有不变的状态，在计算上是不可行的，而人类却能毫不费力地忽略不相关的变化。
- **系统脆弱性 (Brittleness)**：符号系统完全依赖预设规则，导致其行为非常“脆弱”。一旦遇到规则之外的任何微小变化或新情况，系统便可能完全失灵，无法像人类一样灵活变通。SHRDLU的成功，也正是因为它运行在一个规则完备的封闭世界里，而真实世界充满了例外。

2.2 构建基于规则的聊天机器人

在探讨了符号主义的理论挑战后，本节我们将通过一个具体的编程实践，来直观地感受基于规则的系统是如何工作的。我们将尝试复现人工智能历史上一个极具影响力的早期聊天机器人——ELIZA。

2.2.1 ELIZA 的设计思想

ELIZA是由麻省理工学院的计算机科学家约瑟夫·魏泽鲍姆（Joseph Weizenbaum）于1966年发布的一个计算机程序^[6]，是早期自然语言处理领域的著名尝试之一。ELIZA并非一个单一的程序，而是一个可以执行不同“脚本”的框架。其中，最广为人知也最成功的脚本是“DOCTOR”，它模仿了一位罗杰斯学派的非指导性心理治疗师。

ELIZA的工作方式极其巧妙：它从不正面回答问题或提供信息，而是通过识别用户输入中的关键词，然后应用一套预设的转换规则，将用户的陈述转化为一个开放式的提问。例如，当用户说“我为我的男朋友感到难过”时，ELIZA可能会识别出关键词“我为……感到难过”，并应用规则生成回应：“你为什么会为你男朋友感到难过？”

魏泽鲍姆的设计思想并非要创造一个真正能够“理解”人类情感的智能体，恰恰相反，他想证明的是，通过一些简单的句式转换技巧，机器可以在完全不理解对话内容的情况下，营造出一种“智能”和“共情”的假象。然而，出乎他意料的是，许多与ELIZA交互过的人（包括他的秘书）都对其产生了情感上的依赖，深信它能够理解自己。

本节的实践目标即为复现ELIZA的核心机制，以深入理解这种规则驱动方法的优势与根本局限。

2.2.2 模式匹配与文本替换

ELIZA的算法流程基于模式匹配（Pattern Matching）与文本替换（Text Substitution），可被清晰地分解为以下四个步骤：

1. 关键词识别与排序：规则库为每个关键词（如 `mother`, `dreamed`, `depressed`）设定一个优先级。当输入包含多个关键词时，程序会选择优先级最高的关键词所对应的规则进行处理。
2. 分解规则：找到关键词后，程序使用带通配符（*）的分解规则来捕获句子的其余部分。
 1. 规则示例： * my *
 2. 用户输入： "My mother is afraid of me"
 3. 捕获结果： ["", "mother is afraid of me"]
3. 重组规则：程序从与分解规则关联的一组重组规则中，选择一条来生成回应（通常随机选择以增加多样性），并可选择性地使用上一步捕获的内容。
 1. 规则示例： "Tell me more about your family."
 2. 生成输出： "Tell me more about your family."
4. 代词转换：在重组前，程序会进行简单的代词转换（如 `I` → `you`, `my` → `your`），以维持对话的连贯性。

整个工作流程可以用一个简单的伪代码思路来表示：

```
FUNCTION generate_response(user_input):
    // 1. 将用户输入拆分成单词
    words = SPLIT(user_input)

    // 2. 寻找优先级最高的关键词规则
    best_rule = FIND_BEST_RULE(words)
    IF best_rule is NULL:
        RETURN a_generic_response() // 例如："Please go on."

    // 3. 使用规则分解用户输入
    decomposed_parts = DECOMPOSE(user_input, best_rule.decomposition_pattern)
    IF decomposition_failed:
        RETURN a_generic_response()

    // 4. 对分解出的部分进行代词转换
    transformed_parts = TRANSFORM_PRONOUNS(decomposed_parts)

    // 5. 使用重组规则生成回应
    response = REASSEMBLE(transformed_parts, best_rule.reassembly_patterns)

    RETURN response
```

通过这套机制，ELIZA成功地将复杂的自然语言理解问题，简化为了一个可操作的、基于规则的模式匹配游戏。

2.2.3 核心逻辑的实现

现在，我们将上一节描述的技术原理转化为一个简单的、可运行的Python函数。下面的代码实现了一个迷你版的ELIZA，它包含了一小部分规则，但足以展示其核心工作机制。

```
import re
import random

# 定义规则库：模式(正则表达式) -> 响应模板列表
file:///Users/knight/workspace/hello-agents/docs/Hello-Agents-Full.html
```

2/9/26, 1:56 PM

```
rules = {
    r'I need (.*)': [
        "Why do you need {0}?", 
        "Would it really help you to get {0}?", 
        "Are you sure you need {0}?"
    ],
    r'Why don\'t you (.*)\?': [
        "Do you really think I don't {0}?", 
        "Perhaps eventually I will {0}.", 
        "Do you really want me to {0}?"
    ],
    r'Why can\'t I (.*)\?': [
        "Do you think you should be able to {0}?", 
        "If you could {0}, what would you do?", 
        "I don't know -- why can't you {0}?"
    ],
    r'I am (.*)': [
        "Did you come to me because you are {0}?", 
        "How long have you been {0}?", 
        "How do you feel about being {0}?"
    ],
    r'.* mother .*': [
        "Tell me more about your mother.", 
        "What was your relationship with your mother like?", 
        "How do you feel about your mother?"
    ],
    r'.* father .*': [
        "Tell me more about your father.", 
        "How did your father make you feel?", 
        "What has your father taught you?"
    ],
    r'.*': [
        "Please tell me more.", 
        "Let's change focus a bit... Tell me about your family.", 
        "Can you elaborate on that?"
    ]
}

# 定义代词转换规则
pronoun_swap = {
    "i": "you", "you": "i", "me": "you", "my": "your",
    "am": "are", "are": "am", "was": "were", "i'd": "you would",
    "i've": "you have", "i'll": "you will", "yours": "mine",
    "mine": "yours"
}

def swap_pronouns(phrase):
    """
    对输入短语中的代词进行第一/第二人称转换
    """
    words = phrase.lower().split()
    swapped_words = [pronoun_swap.get(word, word) for word in words]
    return " ".join(swapped_words)

def respond(user_input):
    """
    根据规则库生成响应
    """
    for pattern, responses in rules.items():
        match = re.search(pattern, user_input, re.IGNORECASE)
        if match:
            # 捕获匹配到的部分
            captured_group = match.group(1) if match.groups() else ''
            # 进行代词转换
            swapped_group = swap_pronouns(captured_group)
            # 从模板中随机选择一个并格式化
            response = random.choice(responses).format(swapped_group)
            return response
    # 如果没有匹配任何特定规则, 使用最后的通配符规则
    return random.choice(rules[r'.*'])

# 主聊天循环
if __name__ == '__main__':
    print("Therapist: Hello! How can I help you today?")
    while True:
        user_input = input("You: ")
        if user_input.lower() in ["quit", "exit", "bye"]:
            print("Therapist: Goodbye. It was nice talking to you.")
            break
        response = respond(user_input)
        print(f"Therapist: {response}")

>>>
Therapist: Hello! How can I help you today?
You: I am feeling sad today.
Therapist: How long have you been feeling sad today?
You: I need some help with my project.
Therapist: Are you sure you need some help with your project?
You: My mother is not happy with my work.
Therapist: Tell me more about your mother.
You: quit
Therapist: Goodbye. It was nice talking to you.
```

通过上述的编程实践，我们可以直观地总结出规则驱动系统的根本局限性，这些局限正是对 2.1.4 节中符号主义理论挑战的直接印证：

- 缺乏语义理解：系统不理解词义。例如，面对“*I am not happy*”的输入，它仍会机械地匹配 *I am (.*)* 规则并生成语义不通的回应，因为它无法理解否定词“not”的作用。
- 无上下文记忆：系统是无状态的（**Stateless**），每次回应仅基于当前单句输入，无法进行连贯的多轮对话。
- 规则的扩展性问题：尝试增加更多规则会导致规则库的规模爆炸式增长，规则间的冲突与优先级管理将变得极其复杂，最终导致系统难以维护。

然而，尽管存在这些显而易见的缺陷，ELIZA在当时却产生了著名的“**ELIZA效应**”，许多用户相信它能理解自己。这种智能的幻觉主要源于其巧妙的对话策略（如扮演被动的提问者、使用开放式模板）以及人类天生的情感投射心理。

ELIZA的实践清晰地揭示了符号主义方法的核心矛盾：系统看似智能的表现，完全依赖于设计者预先编码的规则。然而，面对真实世界语言的无限可能性，这种穷举式的方法注定不可扩展。系统没有真正的理解，只是在执行符号操作，这正是其脆弱性的根源。

2.3 马文·明斯基的心智社会

符号主义的探索和ELIZA的实践，共同指向了一个问题：通过预设规则构建的、单一的、集中的推理引擎，似乎难以通向真正的智能。无论规则库多么庞大，系统在面对真实世界的模糊性、复杂性和无穷变化时，总是显得僵化而脆弱。这一困境促使一些顶尖的思考者开始反思人工智能最底层的设计哲学。其中，马文·明斯基（**Marvin Minsky**）没有继续尝试为单一推理核心添加更多规则，而是在他的《心智社会》（*The Society of Mind*）^[7]一书中提出了一个革命性的问题：“What magical trick makes us intelligent? The trick is that there is no trick. The power of intelligence stems from our vast diversity, not from any single, perfect principle.”

2.3.1 对单一整体智能模型的反思

20世纪70至80年代，符号主义的局限性日益明显。专家系统虽然在高度垂直的领域取得了成功，但它们无法拥有儿童般的常识；SHRDLU虽然能在一个封闭的积木世界中表现出色，但它无法理解这个世界之外的任何事情；ELIZA虽然能模仿对话，但它对对话内容本身一无所知。这些系统都遵循着一种自上而下（Top-down）的设计思路：一个全知全能的中央处理器，根据一套统一的逻辑规则来处理信息和做出决策。

面对这种普遍的失败，明斯基开始提出一系列根本性的问题：

- “理解”是什么？当我们说我们理解一个故事时，这是一种单一的能力吗？还是说，它其实是视觉化能力、逻辑推理能力、情感共鸣能力、社会关系常识等数十种不同心智过程协同工作的结果？
- “常识”是什么？常识是一个包含了数百万条逻辑规则的庞大知识库吗（如Cyc项目的尝试）？还是说，它是一种分布式的、由无数具体经验和简单规则片段交织而成的网络？
- 智能体应该如何构建？我们是否应该继续追求一个完美的、统一的逻辑系统，还是应该承认，智能本身就是“不完美”的、由许多功能各异、甚至会彼此冲突的简单部分组成的大杂烩？

这些问题直指单一整体智能模型的核心弊端。该类模型试图用一种统一的表示和推理机制来解决所有问题，但这与我们观察到的自然智能（尤其是人类智能）的运作方式相去甚远。明斯基认为，强行将多样化的心智活动塞进一个僵化的逻辑框架中，正是导致早期人工智能研究停滞不前的根源。

正是基于这样的反思，明斯基提出了一个颠覆性的构想，他不再将心智视为一个金字塔式的层级结构，而是将其看作一个扁平化的、充满了互动与协作的“社会”。

2.3.2 作为协作体的智能

在明斯基的理论框架中，智能体的定义与我们第一章讨论的现代智能体有所不同。这里的智能体指的是一个极其简单的、专门化的心智过程，它自身是“无心”的。例如，一个负责识别线条的LINE-FINDER智能体，或一个负责抓握的GRASP智能体。

这些简单的智能体被组织起来，形成功能更强大的机构（**Agency**）。一个机构是一组协同工作的智能体，旨在完成一个更复杂的任务。例如，一个负责搭积木的BUILD机构，可能由SEE、FIND、GET、PUT等多个下层智能体或机构组成。它们之间通过去中心化的激活与抑制信号相互影响，形成动态的控制流。

涌现（Emergence）是理解心智社会理论的关键。复杂的、有目的性的智能行为，并非由某个高级智能体预先规划，而是从大量简单的底层智能体之间的局部交互中自发产生的。

让我们以经典的“搭建积木塔”任务为例，来说明这一过程，如图2.6所示。当一个高层目标（如“我要搭一个塔”）出现时，它会激活一个名为BUILD-TOWER的高层机构。

1. BUILD-TOWER机构并不知道如何执行具体的物理动作，它的唯一作用是激活它的下属机构，比如BUILDER。
2. BUILDER机构同样很简单，它可能只包含一个循环逻辑：只要塔还没搭完，就激活ADD-BLOCK机构。
3. ADD-BLOCK机构则负责协调更具体的子任务，它会依次激活FIND-BLOCK、GET-BLOCK和PUT-ON-TOP这三个子机构。
4. 每一个子机构又由更底层的智能体构成。例如，GET-BLOCK机构会激活视觉系统中的SEE-SHAPE智能体、运动系统中的REACH和GRASP智能体。

在这个过程中，没有任何一个智能体或机构拥有整个任务的全局规划。GRASP只负责抓握，它不知道什么是塔；BUILDER只负责循环，它不知道如何控制手臂。然而，当这个由无数“无心”的智能体组成的社会，通过简单的激活和抑制规则相互作用时，一个看似高度智能的行为，搭建积木塔，就自然而然地涌现了出来。

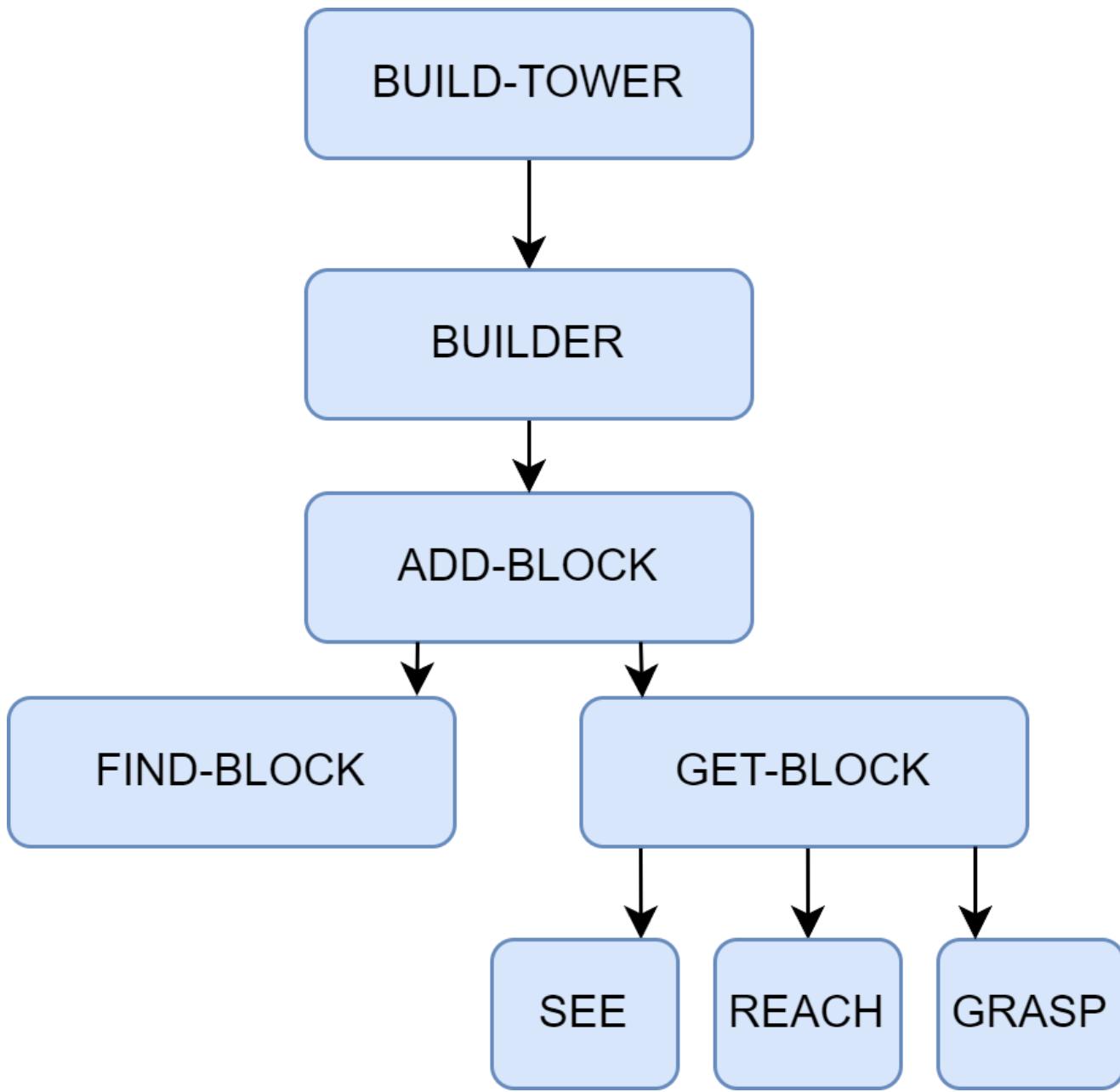


图 2.6 “心智社会”中搭建积木塔行为的涌现机制示意图

2.3.3 对多智能体系统的理论启发

心智社会理论最深远的影响，在于它为分布式人工智能（**Distributed Artificial Intelligence, DAI**）以及后来的多智能体系统（**Multi-Agent System, MAS**）提供了重要的概念基础。它引出研究者们的思考：

如果一个心智内部的智能，是通过大量简单智能体的协作而涌现的，那么，在多个独立的、物理上分离的计算实体（计算机、机器人）之间，是否也能通过协作涌现出更强大的“群体智能”？

这个问题的提出，直接将研究焦点从“如何构建一个全能的单一智能体”转向了“如何设计一个高效协作的智能体群体”。具体而言，心智社会在以下几个方面直接启发了多智能体系统的研究：

- 去中心化控制（**Decentralized Control**）：理论的核心在于不存在中央控制器。这一思想被MAS领域完全继承，如何设计没有中心节点的协调机制和任务分配策略，成为了MAS的核心研究课题之一。
- 涌现式计算（**Emergent Computation**）：复杂问题的解决方案可以从简单的局部交互规则中自发产生。这启发了MAS中大量基于涌现思想的算法，如蚁群算法、粒子群优化等，用于解决复杂的优化和搜索问题。
- 智能体的社会性（**Agent Sociality**）：明斯基的理论强调了智能体之间的交互（激活、抑制）。MAS领域将其进一步扩展，系统地研究智能体之间的通信语言（如ACL）、交互协议（如契约网）、协商策略、信任模型乃至组织结构，从而构建起真正的计算社会。

可以说，明斯基的“心智社会”理论，为AI研究者理解“群体智能”的内在构造提供了重要的分析框架。它为后来的研究者们提供了一套全新的视角，去探索由独立的、自治的、具备社会能力的计算智能体所构成的复杂系统，从而正式开启了多智能体系统研究的序幕。

2.4 学习范式的演进与现代智能体

前文探讨的“心智社会”理论，在哲学层面为群体智能和去中心化协作指明了方向，但实现路径尚不明确。与此同时，符号主义在应对真实世界复杂性时暴露的根本性挑战也表明仅靠预先编码的规则无法构建真正鲁棒的智能。

这两条线索共同指向了一个问题：如果智能无法被完全设计，那么它是否可以被学习出来？

这一设问开启了人工智能的“学习”时代。其核心目标不再是手动编码知识，而是构建能从经验和数据中自动获取知识与能力的系统。本节将追溯这一范式的演进历程：从联结主义奠定的学习基础，到强化学习实现的交互式学习，直至今日由大型语言模型驱动的现代智能体。

2.4.1 从符号到联结

作为对符号主义局限性的直接回应，联结主义（**Connectionism**）在20世纪80年代重新兴起。与符号主义自上而下、依赖明确逻辑规则的设计哲学不同，联结主义是一种自下而上的方法，其灵感来源于对生物大脑神经网络结构的模仿^[8]。它的核心思想可以概括为以下几点：

1. 知识的分布式表示：知识并非以明确的符号或规则形式存储在某个知识库中，而是以连接权重的形式，分布式地存储在大量简单的处理单元（即人工神经元）的连接之间。整个网络的连接模式本身就构成了知识。
2. 简单的处理单元：每个神经元只执行非常简单的计算，如接收来自其他神经元的加权输入，通过一个激活函数进行处理，然后将结果输出给下一个神经元。
3. 通过学习调整权重：系统的智能并非来自于设计者预先编写的复杂程序，而是来自于“学习”过程。系统通过接触大量样本，根据某种学习算法（如反向传播算法）自动、迭代地调整神经元之间的连接权重，从而使得整个网络的输出逐渐接近期望的目标。

在这种范式下，智能体不再是一个被动执行规则的逻辑推理机，而是一个能够通过经验自我优化的适应性系统。如图2.7所示，这代表了构建智能体核心思想的根本性转变。符号主义试图将人类的知识显式地编码给机器，而联结主义则试图创造出能够像人类一样学习知识的机器。

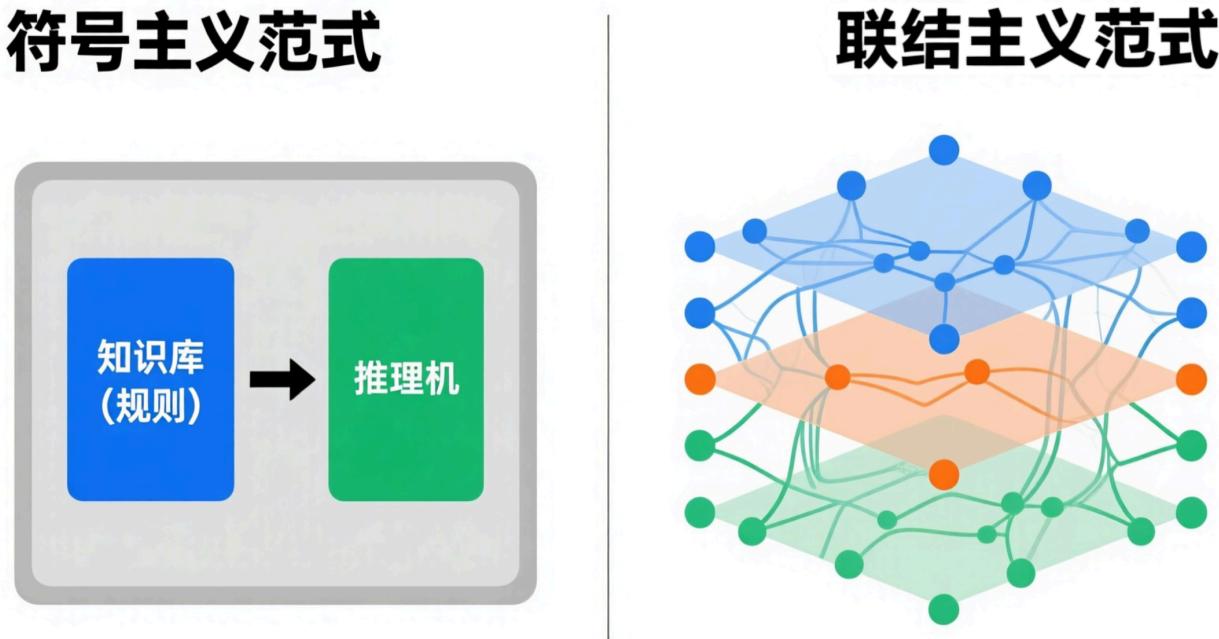


图 2.7 符号主义与联结主义范式对比

联结主义的兴起，特别是深度学习在21世纪的成功，为智能体赋予了强大的感知和模式识别能力，使其能够直接从原始数据（如图像、声音、文本）中理解世界，这是符号主义时代难以想象的。然而，如何让智能体学会在与环境的动态交互中做出最优的序贯决策，则需要另一种学习范式的补充。

2.4.2 基于强化学习的智能体

联结主义主要解决了感知问题（例如，“这张图片里有什么？”），但智能体更核心的任务是进行决策（例如，“在这种情况下，我应该做什么？”）。强化学习（**Reinforcement Learning, RL**）正是专注于解决序贯决策问题的学习范式。它并非直接从标注好的静态数据集中学习，而是通过智能体与环境的直接交互，在“试错”中学习如何最大化其长期收益。

以AlphaGo为例，其核心的自我对弈学习过程便是强化学习的经典体现^[9]。在这个过程中，AlphaGo（智能体）通过观察棋盘的当前布局（环境状态），决定下一步棋的落子位置（行动）。一局棋结束后，根据胜负结果，它会收到一个明确的信号：赢了就是正向奖励，输了则是负向奖励。通过数百万次这样的自我对弈，AlphaGo不断调整其内部策略，逐渐学会了在何种棋局下选择何种行动，最有可能导向最终的胜利。这个过程完全是自主的，不依赖于人类棋谱的直接指导。

这种通过与环境互动、根据反馈信号来优化自身行为的学习机制，就是强化学习的核心框架。下面我们将详细拆解其基本构成要素和工作模式。

强化学习的框架可以用几个核心要素来描述：

- 智能体 (Agent)：学习者和决策者。在AlphaGo的例子中，就是其决策程序。
- 环境 (Environment)：智能体外部的一切，是智能体与之交互的对象。对AlphaGo而言，就是围棋的规则和对手。
- 状态 (State, S)：对环境在某一时刻的特定描述，是智能体做出决策的依据。例如，棋盘上所有棋子的当前位置。
- 行动 (Action, A)：智能体根据当前状态所能采取的操作。例如，在棋盘的某个合法位置上落下一子。
- 奖励 (Reward, R)：环境在智能体执行一个行动后，反馈给智能体的一个标量信号，用于评价该行动在特定状态下的好坏。例如，在一局棋结束后，胜利获得+1的奖励，失败获得-1的奖励。

基于上述核心要素，强化学习智能体在一个“感知-行动-学习”的闭环中持续迭代，其工作模式如图2.8所示。

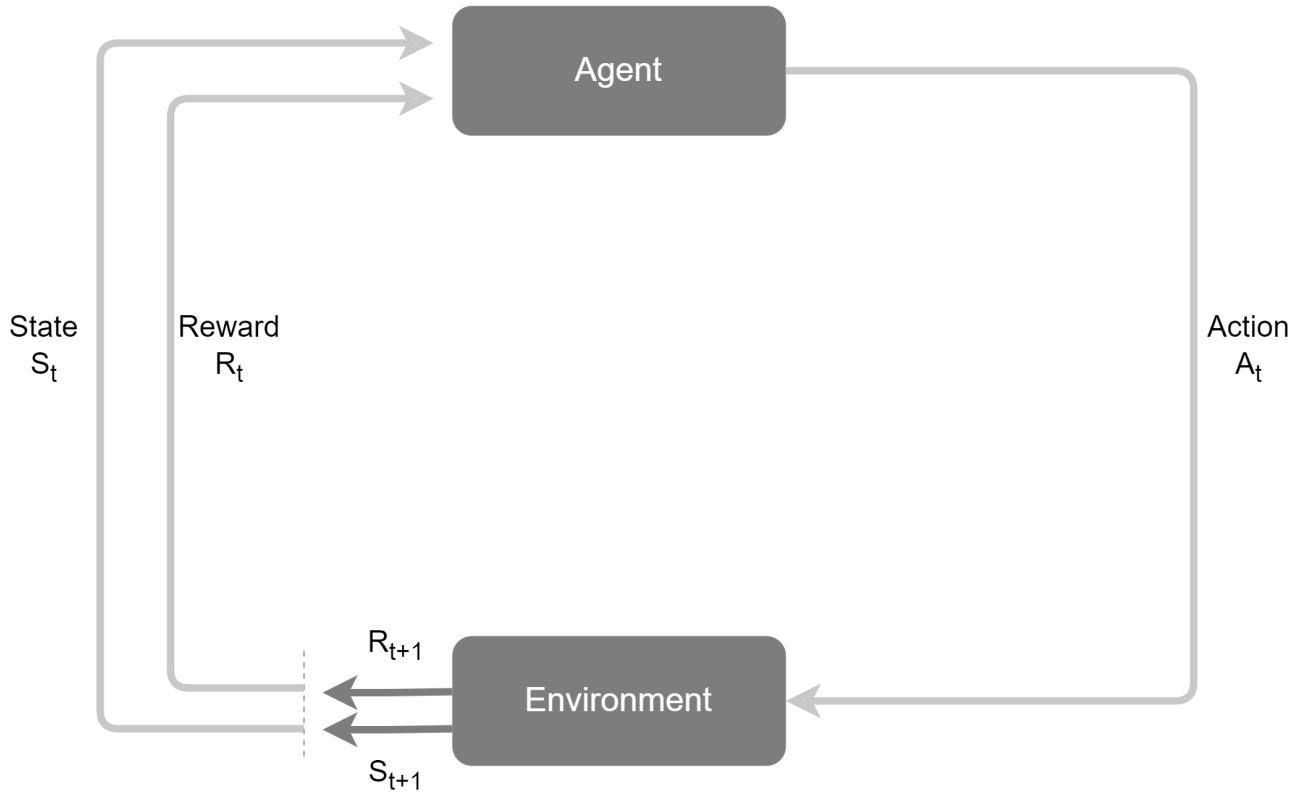


图 2.8 强化学习的核心交互循环

这个循环的具体步骤如下：

1. 在时间步 t ，智能体观察到环境的当前状态 S_t 。
2. 基于状态 S_t ，智能体根据其内部的策略 (**Policy, π**) 选择一个行动 A_t 并执行它。策略本质上是一个从状态到行动的映射，定义了智能体的行为方式。
3. 环境接收到行动 A_t 后，会转移到一个新的状态 S_{t+1} 。
4. 同时，环境会反馈给智能体一个即时奖励 R_{t+1} 。
5. 智能体利用这个反馈（新状态 S_{t+1} 和奖励 R_{t+1} ）来更新和优化其内部策略，以便在未来做出更好的决策。这个更新过程就是学习。

智能体的学习目标，并非最大化某一个时间步的即时奖励，而是最大化从当前时刻开始到未来的累积奖励 (**Cumulative Reward**)，也称为回报 (**Return**)。这意味着智能体需要具备“远见”，有时为了获得未来更大的奖励，需要牺牲当前的即时奖励（例如，围棋中的“弃子”策略）。通过在上述循环中不断探索、收集反馈并优化策略，智能体最终能够学会在复杂动态环境中进行自主决策和长期规划。

2.4.3 基于大规模数据的预训练

强化学习赋予了智能体从交互中学习决策策略的能力，但这通常需要海量的、针对特定任务的交互数据，导致智能体在学习之初缺乏先验知识，需要从零开始构建对任务的理解。无论是符号主义试图手动编码的常识，还是人类在决策时所依赖的背景知识，在RL智能体中都是缺失的。如何让智能体在开始学习具体任务前，就先具备对世界的广泛理解？这一问题的解决方案，最终在自然语言处理 (**Natural Language Processing, NLP**) 领域中浮现，其核心便是基于大规模数据的预训练 (**Pre-training**)。

从特定任务到通用模型

在预训练范式出现之前，传统的自然语言处理模型通常是为单一特定任务（如情感分析、机器翻译）在专门标注的中小规模数据集上从零开始独立训练的。这种模式导致了几个问题：模型的知识面狭窄，难以将在一个任务中学到的知识泛化到另一个任务，并且每一个新任务都需要耗费大量的人力去标注数据。预训练与微调 (**Pre-training, Fine-tuning**) 范式的提出彻底改变了这一现状。其核心思想分为两步：

1. 预训练阶段：首先在一个包含互联网级别海量文本数据的通用语料库上，通过自监督学习 (**Self-supervised Learning**) 的方式训练一个超大规模的神经网络模型。这个阶段的目标不是完成任何特定任务，而是学习语言本身内在的规律、语法结构、事实知识以及上下文逻辑。最常见的目标是“预测下一个词”。

2. 微调阶段：完成预训练后，这个模型就已经学习到了和数据集有关的丰富知识。之后，针对特定的下游任务，只需使用少量该任务的标注数据对模型进行微调，即可让模型适应对应任务。

如图2.9所示，直观地展示了这一预训练与微调的完整流程：通用文本数据经过自监督学习形成基础模型，随后通过特定任务数据进行微调，最终适应各项下游任务。

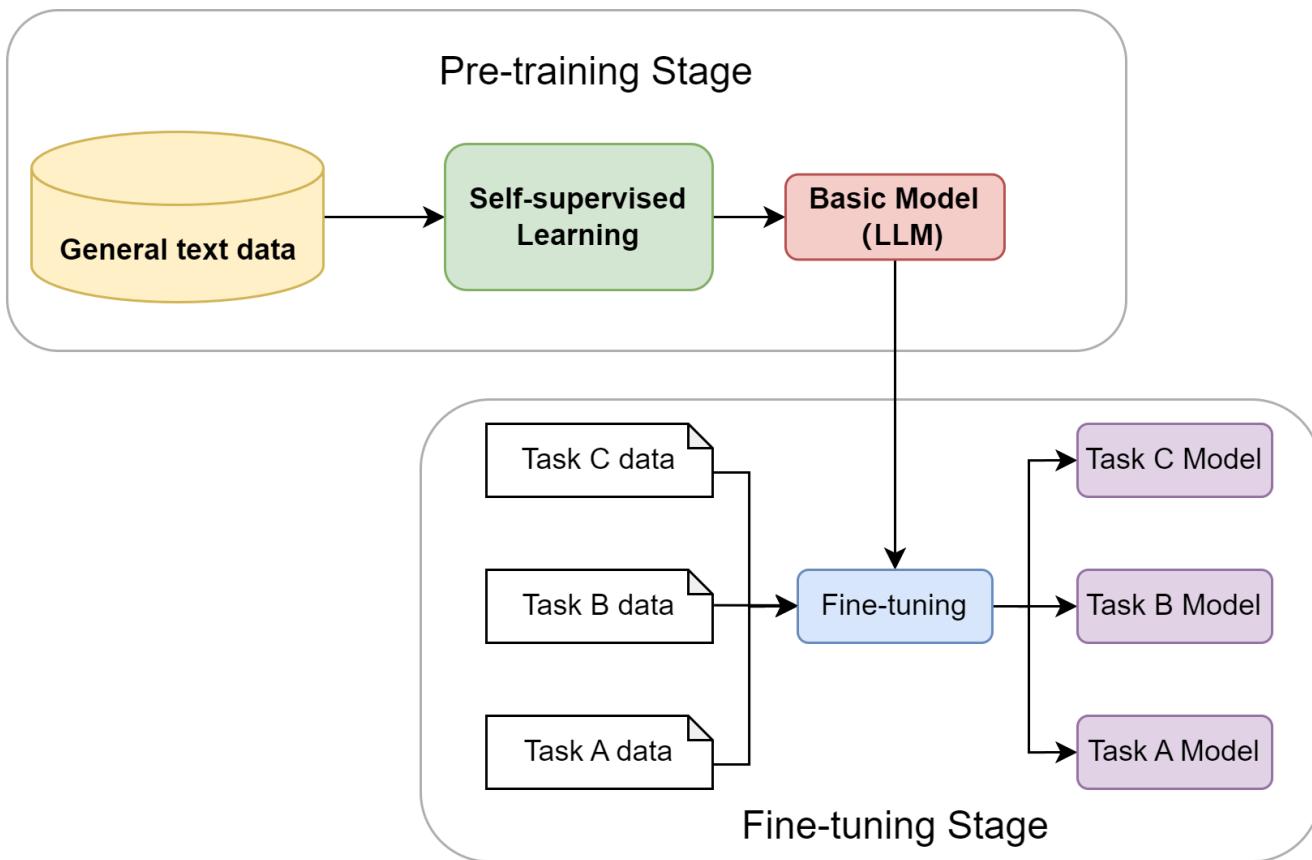


图 2.9 “预训练-微调”范式示意图

大型语言模型的诞生与涌现能力

通过在数万亿级别的文本上进行预训练，大型语言模型的神经网络权重实际上已经构建了一个关于世界知识的、高度压缩的隐式模型。它以一种全新的方式，解决了符号主义时代最棘手的“知识获取瓶颈”问题。更令人惊讶的是，当模型的规模（参数量、数据量、计算量）跨越某个阈值后，它们开始展现出未被直接训练的、预料之外的涌现能力（Emergent Abilities），例如：

- 上下文学习（In-context Learning）：无需调整模型权重，仅在输入中提供几个示例（Few-shot）甚至零个示例（Zero-shot），模型就能理解并完成新的任务。
- 思维链（Chain-of-Thought）推理：通过引导模型在回答复杂问题前，先输出一步步的推理过程，可以显著提升其在逻辑、算术和常识推理任务上的准确性。

这些能力的出现，标志着LLM不再仅仅是一个语言模型，它已经演变成了一个兼具海量知识库和通用推理引擎双重角色的组件。

至此，智能体发展的历史长河中，几大关键的技术拼图已经悉数登场：符号主义提供了逻辑推理的框架，联结主义和强化学习提供了学习与决策的能力，而大型语言模型则提供了前所未有的、通过预训练获得的世界知识和通用推理能力。下一节，我们将看到这些技术是如何在现代智能体的设计中融为一体的。

2.4.4 基于大语言模型的智能体

随着大型语言模型技术的飞速发展，以LLM为核心的智能体已成为人工智能领域的新范式。它不仅能够理解和生成人类语言，更重要的是，能够通过与环境的交互，自主地感知、规划、决策和执行任务。

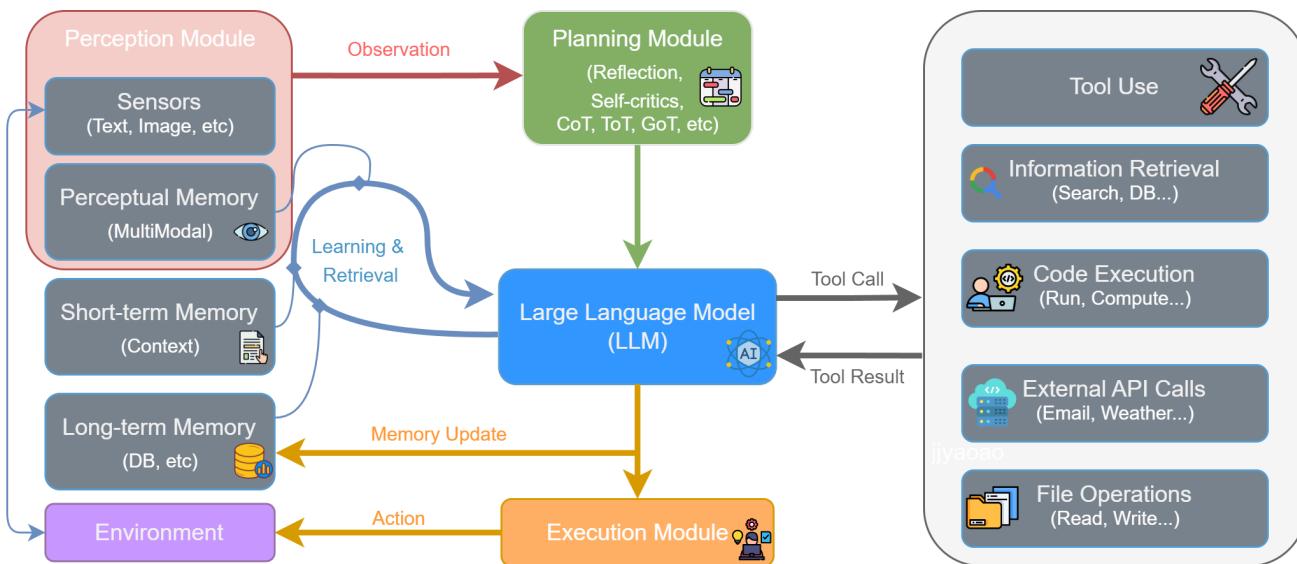


图 2.10 LLM 驱动的智能体核心组件架构

如第一章所述，智能体与环境的交互可以被抽象为一个核心循环。LLM 驱动的智能体通过一个由多个模块协同工作的、持续迭代的闭环流程来完成任务。该流程遵循图 2.10 所示的架构，具体步骤如下：

1. 感知 (Perception)：流程始于感知模块 (Perception Module)。它通过传感器从外部环境 (Environment) 接收原始输入，形成观察 (Observation)。这些观察信息（如用户指令、API 返回的数据或环境状态的变化）是智能体决策的起点，处理后将被传递给思考阶段。
2. 思考 (Thought)：这是智能体的认知核心，对应图中的规划模块 (Planning Module) 和大型语言模型 (LLM) 的协同工作。
 - 规划与分解：首先，规划模块接收观察信息，进行高级策略制定。它通过反思 (Reflection) 和自我批判 (Self-criticism) 等机制，将宏观目标分解为更具体、可执行的步骤。
 - 推理与决策：随后，作为中枢的 LLM 接收来自规划模块的指令，并与记忆模块 (Memory) 交互以整合历史信息。LLM 进行深度推理，最终决策出下一步要执行的具体操作，这通常表现为一个工具调用 (Tool Call)。
3. 行动 (Action)：决策完成后，便进入行动阶段，由执行模块 (Execution Module) 负责。LLM 生成的工具调用指令被发送到执行模块。该模块解析指令，从工具箱 (Tool Use) 中选择并调用合适的工具（如代码执行器、搜索引擎、API 等）来与环境交互或执行任务。这个与环境的实际交互就是智能体的行动 (Action)。
4. 观察 (Observation) 与循环：行动会改变环境的状态，并产生结果。
 - 工具执行后会返回一个工具结果 (Tool Result) 给 LLM，这构成了对行动效果的直接反馈。同时，智能体的行动改变了环境，从而产生了一个全新的环境状态。
 - 这个“工具结果”和“新的环境状态”共同构成了一轮全新的观察 (Observation)。这个新的观察会被感知模块再次捕获，同时 LLM 会根据行动结果更新记忆 (Memory Update)，从而启动下一轮“感知-思考-行动”的循环。

这种模块化的协同机制与持续的迭代循环，构成了 LLM 驱动智能体解决复杂问题的核心工作流。

2.4.5 智能体发展关键节点概览

人工智能体的发展史并非一条笔直的单行道，而是几大核心思想流派长达半个多世纪交织、竞争与融合的历程。理解这一历程，有助于我们洞察当前智能体架构范式形成的深刻根源。

这其中，主要有三大思潮主导着不同时期的研究范式：

1. 符号主义 (Symbolism)：以赫伯特·西蒙 (Herbert A. Simon)、明斯基 (Marvin Minsky) 等先驱为代表，认为智能的核心在于对符号的操作与逻辑推理。这一思想催生了能够理解自然语言指令的 SHRDLU、知识驱动的专家系统以及在国际象棋领域取得巨大成功的“深蓝”计算机。
2. 联结主义 (Connectionism)：其灵感源于对大脑神经网络的模拟。尽管早期发展受限，但在杰弗里·辛顿 (Geoffrey Hinton) 等研究者的推动下，反向传播算法为神经网络的复苏奠定了基础。最终，随着深度学习时代的到来，这一思想通过卷积神经网络、Transformer 等模型成为当前的主流。
3. 行为主义 (Behaviorism)：强调智能体通过与环境的互动和试错来学习最优策略，其现代化身为强化学习。从早期的 TD-Gammon 到与深度学习结合并击败人类顶尖棋手的 AlphaGo，这一流派为智能体赋予了从经验中习得复杂决策行为的能力。

进入 21 世纪 20 年代，这些思想流派以前所未有的方式深度融合。以 GPT 系列为代表的大语言模型，其本身是联结主义的产物，却成为了执行符号推理、进行工具调用和规划决策的核心“大脑”，形成了神经-符号结合的现代智能体架构。为了系统性地回顾这一发展脉络，下图 2.11 梳理了从 20 世纪 50 年代至今，人工智能体发展史上的关键理论、项目与事件，为读者提供一个清晰的全局概览，作为本章知识的沉淀。

时期 (Era)	核心思想/流派 (Core Ideology/School of Thought)	关键智能体/项目/技术 (Key Agent/Project/Technology)	核心贡献 (Core Contribution)
符号主义时期 (1950s-1970s)	符号主义 (Symbolism)	逻辑理论家 (Logic Theorist, 1956) 通用问题求解器 (GPS, 1959) 机器人Shakey (1966-1972)	验证了机器执行形式化符号推理的可行性，是对机器“思考”能力的早期探索。 引入“手段-目的分析”(Means-Ends Analysis)，为目标驱动的规划问题奠定了理论基础。 作为首个集成感知、推理与物理交互的移动机器人，成为具备智能体的早期范例。
专家系统与联结主义复苏 (1980s)	符号主义 (Symbolism) 联结主义 (Connectionism)	专家系统 (如 MYCIN, XCON) 反向传播算法的普及	将领域专家的知识编码为规则库，使系统能够在特定领域内辅助决策，是知识驱动型AI的代表。 为训练多层神经网络提供了有效方法，成为后续AI模型学习能力的核心技术。
第二次AI低谷与新起点 (1990s-2000s)	符号主义 (Symbolism) 联结主义 (Connectionism) 行为主义 (强化学习)	深蓝 (Deep Blue, 1997) LeNet-5 (1998) TD-Gammon (1992) Q-Learning (1989)	符号主义方法的成功应用，通过大规模搜索和评估函数在国际象棋领域达到顶尖水平。 经典的卷积神经网络 (CNN) 架构，实现了从原始像素中自动学习特征，为现代计算机视觉奠定了基础。 将时序差分学习 (TD-Learning) 与神经网络结合，证明了智能体可通过自博弈达到高水平竞技能力。 提出了一种经典的无模型 (Model-Free) 强化学习算法，提升了智能体在与环境互动中学习策略的灵活性。
深度学习时代 (2010s)	联结主义 (Connectionism) & 行为主义 (RL)	AlexNet (2012) DQN (2013) AlphaGo (2016) Transformer (2017)	在ImageNet竞赛中取得突破，其深度卷积网络结构推动了深度学习在图像识别领域的广泛应用。 将深度学习与Q-Learning结合，使智能体能够直接从高维感官输入（如像素）中学习控制策略。 结合了基于深度学习的策略/价值网络与蒙特卡洛树搜索 (MCTS)，成为神经-符号混合方法的成功典范。 提出自注意力机制 (Self-Attention)，成为现代大语言模型的标准架构，显著提升了处理长序列数据的能力。
大模型时代 (2020-2022)	大语言模型驱动 (LLM-Driven)	GPT-3 (2020) 思维链 (Chain-of-Thought, 2022)	大规模参数展示了强大的少样本 (Few-shot) 学习能力，体现了其作为通用智能基础模型的潜力。 通过引导模型生成中间推理步骤，提升了其在复杂推理任务上的表现。
智能体时代 (2023至今)	智能体架构范式 (Agentic Paradigm): 以大型语言模型 (LLM) 为核心，集成记忆、规划和工具使用等模块，形成“感知-推理-行动”的循环。 多智能体系统 (Multi-Agent Systems, MAS): 多个智能体基于不同角色进行协同，通过分工、协作来处理单一智能体难以完成的复杂任务。 神经-符号结合 (Neuro-Symbolic Integration): 在实践中，联结主义模型 (LLM) 被用于处理语言 (符号) 并调用外部工具 (符号化行动)，实现了学习与推理的结合。	框架: ReAct (2022), LangChain, LangGraph, crewAI 项目/应用: Auto-GPT (2023) BabyAGI (2023) Voyager (2023) ChatGPT Agent (2025) Google Big Sleep (2024-2025)	ReAct: 提出将“推理”(Reasoning) 与“行动”(Action) 相结合的框架，成为许多LLM智能体的基础模式。 Auto-GPT / BabyAGI (2023): 作为早期的自主智能体应用，展示了AI自主分解目标并循环执行任务的能力，引发了该领域的广泛关注。 Voyager (2023): 在开放世界游戏《我的世界》中，通过自主探索和构建技能库来持续学习新能力。 ChatGPT Agent (2025): 将智能体能力应用于产品，使其能代表用户在网页环境中执行导航、点击和输入等操作，以完成特定任务。 Google Big Sleep (2024-2025): 展示了智能体在网络安全等专业领域的应用潜力，例如自主发现软件漏洞。

图 2.11 智能体发展演进时间线（未完全版）

得益于大语言模型的突破，智能体技术栈呈现出前所未有的活跃度和多样性。图2.12展示了当前AI Agent领域的一个典型技术栈全貌，涵盖了从底层模型到上层应用的各个环节。

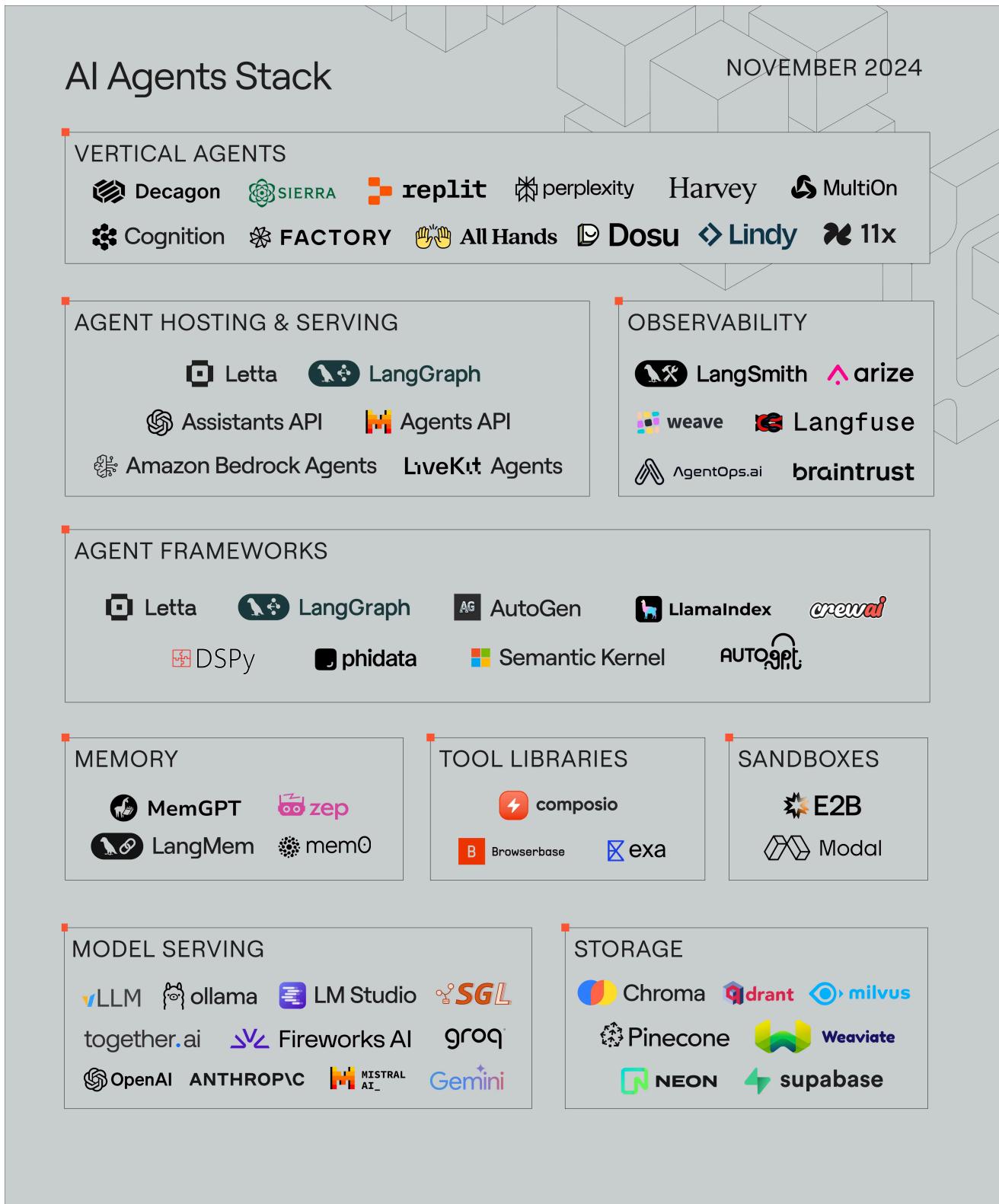


图 2.12 AI Agent 技术栈概览

该技术栈图由Letta公司于2024年11月发布^[10]，它将AI智能体相关的工具、平台和服务进行了分层与分类，为我们理解当前的市场格局和技术选型提供了宝贵的参考。

2.5 本章小结

本章回顾了智能体发展的历史脉络，探索了其核心思想从诞生到演进的过程，内容涵盖了人工智能领域几次关键的范式革命：

- 符号主义的探索与局限：从人工智能的古典时代出发，本章阐述了以专家系统为代表的早期智能体是如何尝试通过“知识+推理”来模拟智能的。通过亲手构建一个基于规则的聊天机器人，我们深刻体会到这一范式的能力边界及其面临的根本性挑战。

- 分布式智能思想的萌芽：探讨了马文·明斯基的“心智社会”理论。这一革命性的思想揭示了复杂的整体智能可以从简单的局部单元的交互中涌现，为后续的多智能体系统研究提供了重要的哲学启发。
- 学习范式的演进：见证了智能体获取能力方式的根本性变革。从联结主义赋予智能体感知世界的能力，到强化学习使其学会在与环境的交互中进行最优决策，再到基于大规模数据预训练的大型语言模型（LLM）为其提供了前所未有的世界知识和通用推理能力。
- 现代智能体的诞生：最后，我们对LLM驱动智能体进行分析。通过对核心组件（模型、记忆、规划、工具等）和工作原理的分析，我们理解了历史上的各种技术思想是如何在现代Agent的架构中实现技术融合的。

通过本章的学习，我们不仅理解了第一章所介绍的现代智能体从何而来，更能建立了一个关于智能体技术演进的宏观认知框架。可以发现，智能体的发展并非简单的技术迭代，而是一场关于如何定义“智能”、获取“知识”、进行“决策”的思想变革。

既然现代智能体的核心是大型语言模型，那么深入理解其底层原理便至关重要。下一章将聚焦于大语言模型本身，探讨其基本概念，为后续在多智能体系统中的高级应用打下坚实的基础。

习题

提示：以下的部分习题没有标准答案，旨在帮助学习者建立对智能体发展历史的系统性理解，并培养“以史为鉴”的技术洞察力。

1. 物理符号系统假说^[1]是符号主义时代的理论基石。请分析：

- 该假说的“充分性论断”和“必要性论断”分别是什么含义？
- 结合本章内容，说明符号主义智能体在实践中遇到的哪些问题对该假说的“充分性”提出了挑战？
- 大语言模型驱动的智能体是否符合物理符号系统假说？

2. 专家系统MYCIN^[2]在医疗诊断领域取得了显著成功，但最终并未大规模应用于临床实践。请思考：

提示：可以从技术、伦理、法律、用户接受度等多个角度分析

- 除了本章提到的“知识获取瓶颈”和“脆弱性”，还有哪些因素可能阻碍了专家系统在医疗等高风险领域的应用？
- 如果让现在的你设计一个医疗诊断智能体，你会如何设计系统来克服MYCIN的局限？
- 在哪些垂直领域中，基于规则的专家系统至今仍然是比深度学习更好的选择？请举例说明。

3. 在2.2节中，我们实现了一个简化版的ELIZA聊天机器人。请在此基础上进行扩展实践：

提示：这是一道动手实践题，建议实际编写代码

- 为ELIZA添加3-5条新的规则，使其能够处理更多样化的对话场景（如谈论工作、学习、爱好等）
- 实现一个简单的“上下文记忆”功能：让ELIZA能够记住用户在对话中提到的关键信息（如姓名、年龄、职业），并在后续对话中引用
- 对比你扩展后的ELIZA与[ChatGPT](#)，列举至少3个维度上存在的本质差异
- 为什么基于规则的方法在处理开放域对话时会遇到“组合爆炸”问题并且难以扩展维护？能否使用数学的方法来说明？

4. 马文·明斯基在“心智社会”理论^[7]中提出了一个革命性的观点：智能源于大量简单智能体的协作，而非单一的完美系统。

- 在图2.6“搭建积木塔”的例子中，如果GRASP智能体突然失效了，整个系统会发生什么？这种去中心化架构的优势和劣势是什么？
- 将“心智社会”理论与现在的一些多智能体系统（如[CAMEL-Workforce](#)、[MetaGPT](#)、[CrewAI](#)）进行对比，它们之间存在哪些关联和不同之处？
- 马文·明斯基认为智能体可以是“无心”的简单过程，然而现在的大语言模型和智能体往往都拥有强大的推理能力。这是否意味着“心智社会”理论在大语言模型时代不再适用了？

5. 强化学习与监督学习是两种不同的学习范式。请分析：

- 用AlphaGo的例子说明强化学习的“试错学习”机制是如何工作的
- 为什么强化学习特别适合序贯决策问题？它与监督学习在数据需求上有什么本质区别？
- 现在我们需要训练一个会玩超级马里奥游戏的智能体。如果分别使用监督学习和强化学习，各需要什么数据？哪种方法对于这个任务来说更合适？
- 在大语言模型的训练过程中，强化学习起到了什么关键性的作用？

6. 预训练-微调范式是现代人工智能领域的重要突破。请深入思考：

- 为什么说预训练解决了符号主义时代的“知识获取瓶颈”问题？它们在知识表示方式上有什么本质区别？
- 预训练模型的知识绝大部分来自互联网数据，这可能带来哪些问题？如何缓解以上问题？
- 你认为“预训练-微调”范式是否可能会被某种新范式取代？或者它会长期存在？

7. 假设你要设计一个“智能代码审查助手”，它能够自动审查代码提交（Pull Request），概括代码的实现逻辑、检查代码质量、发现潜在BUG、提出改进建议。

- 如果在符号主义时代（1980年代）设计这个系统，你会如何实现？会遇到什么困难？
- 如果在没有大语言模型的深度学习时代（2015年左右），你会如何实现？
- 在当前的大语言模型和智能体的时代，你会如何设计这个智能体的架构？它应该包含哪些模块（参考图2.10）？
- 对比这三个时代的方案，说明智能体技术的演进如何使这个任务从“几乎不可能”变为“可行”

参考文献

- [1] NEWELL A, SIMON H A. Computer science as empirical inquiry: symbols and search[J]. Communications of the ACM, 1976, 19(3): 113-126.
- [2] BUCHANAN B G, SHORTLIFFE E H, ed. Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project[M]. Reading, Mass.: Addison-Wesley, 1984.
- [3] WINOGRAD T. Understanding natural language[M]. New York: Academic Press, 1972.
- [4] LENAT D B, GUHA R V. Cyc: a midterm report[J]. AI magazine, 1990, 11(3): 32.
- [5] MCCARTHY J, HAYES P J. Some philosophical problems from the standpoint of artificial intelligence[C]//MELTZER B, MICHIE D, ed. Machine intelligence 4. Edinburgh: Edinburgh University Press, 1969: 463-502.
- [6] WEIZENBAUM J. ELIZA: a computer program for the study of natural language communication between man and machine[J]. Communications of the ACM, 1966, 9(1): 36-45.
- [7] MINSKY M. The society of mind[M]. New York: Simon & Schuster, 1986.
- [8] RUMELHART D E, MCCLELLAND J L, PDP RESEARCH GROUP. Parallel distributed processing: explorations in the microstructure of cognition[M]. Cambridge, MA: MIT Press, 1986.
- [9] SILVER D, HUANG A, MADDISON C J, ed. Mastering the game of Go with deep neural networks and tree search[J]. Nature, 2016, 529(7587): 484-489.
- [10] LETTA. The AI agents stack[EB/OL]. (2024-11) [2025-09-07]. <https://www.letta.com/blog/ai-agents-stack>.

第三章 大语言模型基础

前两章分别介绍了智能体的定义和发展历史，本章将完全聚焦于大语言模型本身解答一个关键问题：现代智能体是如何工作的？我们将从语言模型的基本定义出发，通过对这些原理的学习，为理解LLM如何获得强大的知识储备与推理能力打下坚实的基础。

3.1 语言模型与 Transformer 架构

3.1.1 从 N-gram 到 RNN

语言模型 (**Language Model, LM**) 是自然语言处理的核心，其根本任务是计算一个词序列（即一个句子）出现的概率。一个好的语言模型能够告诉我们什么样的句子是通顺的、自然的。在多智能体系统中，语言模型是智能体理解人类指令、生成回应的基础。本节将回顾从经典的统计方法到现代深度学习模型的演进历程，为理解后续的 Transformer 架构打下坚实的基础。

(1) 统计语言模型与N-gram的思想

在深度学习兴起之前，统计方法是语言模型的主流。其核心思想是，一个句子出现的概率，等于该句子中每个词出现的条件概率的连乘。对于一个由词 w_1, w_2, \dots, w_m 构成的句子 S，其概率 $P(S)$ 可以表示为：

$$P(S) = P(w_1, w_2, \dots, w_m) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1, w_2) \cdots P(w_m | w_1, \dots, w_{m-1})$$

这个公式被称为概率的链式法则。然而，直接计算这个公式几乎是不可能的，因为像 $P(w_m | w_1, \dots, w_{m-1})$ 这样的条件概率太难从语料库中估计了，词序列 w_1, \dots, w_{m-1} 可能从未在训练数据中出现过。

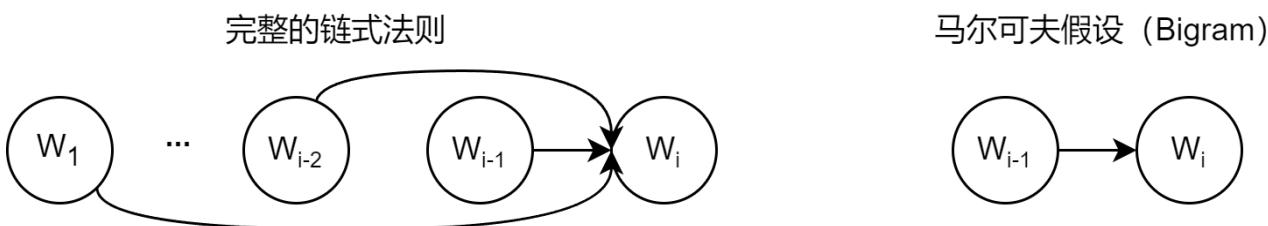


图 3.1 马尔可夫假设示意图

为了解决这个问题，研究者引入了马尔可夫假设 (**Markov Assumption**)。其核心思想是：我们不必回溯一个词的全部历史，可以近似地认为，一个词的出现概率只与它前面有限的 $n - 1$ 个词有关，如图3.1所示。基于这个假设建立的语言模型，我们称之为 **N-gram** 模型。这里的“N”代表我们考虑的上下文窗口大小。让我们来看几个最常见的例子来理解这个概念：

- **Bigram (当 N=2 时)**：这是最简单的情况，我们假设一个词的出现只与它前面的一个词有关。因此，链式法则中复杂的条件概率 $P(w_i | w_1, \dots, w_{i-1})$ 就可以被近似为更容易计算的形式：

$$P(w_i | w_1, \dots, w_{i-1}) \approx P(w_i | w_{i-1})$$

- **Trigram (当 N=3 时)**：类似地，我们假设一个词的出现只与它前面的两个词有关：

$$P(w_i | w_1, \dots, w_{i-1}) \approx P(w_i | w_{i-2}, w_{i-1})$$

这些概率可以通过在大型语料库中进行最大似然估计(**Maximum Likelihood Estimation, MLE**) 来计算。这个术语听起来很复杂，但其思想非常直观：最可能出现的，就是我们在数据中看到次数最多的。例如，对于 Bigram 模型，我们想计算在词 w_{i-1} 出现后，下一个词是 w_i 的概率 $P(w_i | w_{i-1})$ 。根据最大似然估计，这个概率可以通过简单的计数来估算：

$$\text{\$\$P}(w_i | w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})} \text{\$\$}$$

这里的 Count() 函数就代表“计数”：

- $\text{Count}(w_{i-1}, w_i)$: 表示词对 (w_{i-1}, w_i) 在语料库中连续出现的总次数。
- $\text{Count}(w_{i-1})$: 表示单个词 w_{i-1} 在语料库中出现的总次数。

公式的含义就是：我们用“词对 $\text{Count}(w_{i-1}, w_i)$ 出现的次数”除以“词 $\text{Count}(w_{i-1})$ 出现的总次数”，来作为 $P(w_i | w_{i-1})$ 的一个近似估计。

为了让这个过程更具体，我们来手动进行一次计算。假设我们拥有一个仅包含以下两句话的迷你语料库：datawhale agent learns, datawhale agent works。我们的目标是：使用 Bigram (N=2) 模型，估算句子 datawhale agent learns 出现的概率。根据 Bigram 的假设，我们每次会考察连续的两个词（即一个词对）。

第一步：计算第一个词的概率 $P(\text{datawhale})$ 这是 datawhale 出现的次数除以总词数。datawhale 出现了 2 次，总词数是 6。

$$\text{\$\$P}(\text{text}{datawhale}) = \frac{\text{Count}(\text{text}{datawhale})}{\text{Count}(\text{text}{all words})} = \frac{2}{6} \approx 0.333 \text{\$\$}$$

第二步：计算条件概率 $P(\text{agent} | \text{datawhale})$ 这是词对 datawhale agent 出现的次数除以 datawhale 出现的总次数。datawhale agent 出现了 2 次，datawhale 出现了 2 次。

$$\$P(\text{agent}|\text{datawhale}) = \frac{\text{Count}(\text{datawhale agent})}{\text{Count}(\text{datawhale})} = \frac{2}{2} = 1\$$$

第三步：计算条件概率 $P(\text{learns} | \text{agent})$ 这是词对 `agent` `learns` 出现的次数除以 `agent` 出现的总次数。`agent` `learns` 出现了 1 次，`agent` 出现了 2 次。

$$\$P(\text{learns} | \text{agent}) = \frac{\text{Count}(\text{agent learns})}{\text{Count}(\text{agent})} = \frac{1}{2} = 0.5\$$$

最后：将概率连乘 所以，整个句子的近似概率为：

$$P(\text{datawhale agent learns}) \approx P(\text{datawhale}) \cdot P(\text{agent}|\text{datawhale}) \cdot P(\text{learns}|\text{agent}) \approx 0.333 \cdot 1 \cdot 0.5 \approx 0.167$$

```
import collections
```

示例语料库，与上方案例讲解中的语料库保持一致

```
corpus = "datawhale agent learns datawhale agent works"
tokens = corpus.split()
total_tokens = len(tokens)
```

--- 第一步：计算 $P(\text{datawhale})$ ---

```
count_datawhale = tokens.count('datawhale')
p_datawhale = count_datawhale / total_tokens
print(f"第一步: P(datawhale) = {count_datawhale}/{total_tokens} = {p_datawhale:.3f}")
```

--- 第二步：计算 $P(\text{agent}|\text{datawhale})$ ---

```
# 先计算 bigrams 用于后续步骤
bigrams = zip(tokens, tokens[1:])
bigram_counts = collections.Counter(bigrams)
count_datawhale_agent = bigram_counts[('datawhale', 'agent')]
# count_datawhale 已在第一步计算
p_agent_given_datawhale = count_datawhale_agent / count_datawhale
print(f"第二步: P(agent|datawhale) = {count_datawhale_agent}/{count_datawhale} = {p_agent_given_datawhale:.3f}")
```

--- 第三步：计算 $P(\text{learns}|\text{agent})$ ---

```
count_agent_learns = bigram_counts[('agent', 'learns')]
count_agent = tokens.count('agent')
p_learns_given_agent = count_agent_learns / count_agent
print(f"第三步: P(learns|agent) = {count_agent_learns}/{count_agent} = {p_learns_given_agent:.3f}")
```

--- 最后：将概率连乘 ---

```
p_sentence = p_datawhale * p_agent_given_datawhale * p_learns_given_agent
print(f"最后: P('datawhale agent learns') \approx {p_datawhale:.3f} * {p_agent_given_datawhale:.3f} * {p_learns_given_agent:.3f} = {p_sentence:.3f}"
```

>>>

第一步: P(datawhale) = 2/6 = 0.333

第二步: P(agent|datawhale) = 2/2 = 1.000

第三步: P(learns|agent) = 1/2 = 0.500

最后: P('datawhale agent learns') \approx 0.333 * 1.000 * 0.500 = 0.167

N-gram 模型虽然简单有效，但有两个致命缺陷：

1. 数据稀疏性 (**Sparsity**)：如果一个词序列从未在语料库中出现，其概率估计就为 0，这显然是不合理的。虽然可以通过平滑 (Smoothing) 技术缓解，但无法根除。
2. 泛化能力差：模型无法理解词与词之间的语义相似性。例如，即使模型在语料库中见过很多次 `agent` `learns`，它也无法将这个知识泛化到语义相似的词上。当我们计算 `robot` `learns` 的概率时，如果 `robot` 这个词从未出现过，或者 `robot` `learns` 这个组合从未出现过，模型计算出的概率也会是零。模型无法理解 `agent` 和 `robot` 在语义上的相似性。

(2) 神经网络语言模型与词嵌入

N-gram 模型的根本缺陷在于它将词视为孤立、离散的符号。为了克服这个问题，研究者们转向了神经网络，并提出了一种思想：用连续的向量来表示词。

2003 年，Bengio 等人提出的前馈神经网络语言模型 (**Feedforward Neural Network Language Model**) 是这一领域的里程碑^[1]。

其核心思想可以分为两步：

1. 构建一个语义空间：创建一个高维的连续向量空间，然后将词汇表中的每个词都映射为该空间中的一个点。这个点（即向量）就被称为词嵌入 (**Word Embedding**) 或词向量。在这个空间里，语义上相近的词，它们对应的向量在空间中的位置也相近。例如，`agent` 和 `robot` 的向量会靠得很近，而 `agent` 和 `apple` 的向量会离得很远。
2. 学习从上下文到下一个词的映射：利用神经网络的强大拟合能力，来学习一个函数。这个函数的输入是前 $n - 1$ 个词的词向量，输出是词汇表中每个词在当前上下文后出现的概率分布。

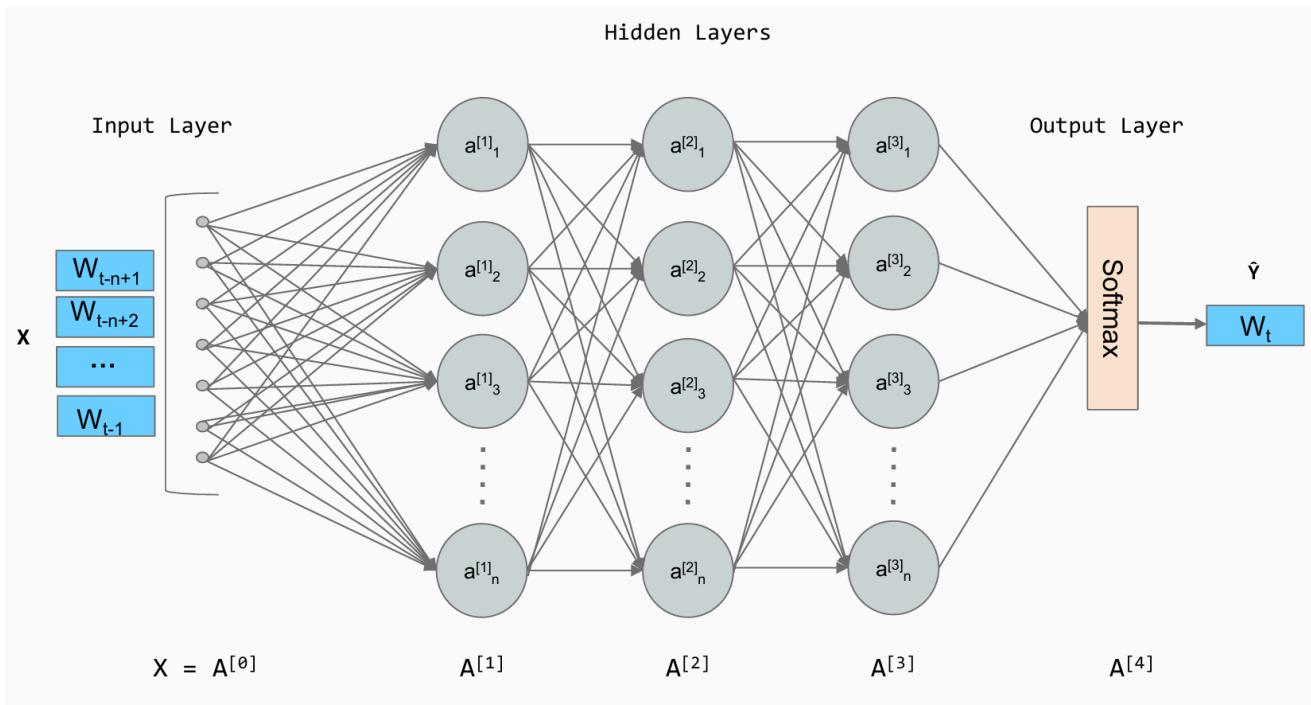


图 3.2 神经网络语言模型架构示意图

如图3.2所示，在这个架构中，词嵌入是在模型训练过程中自动学习得到的。模型为了完成“预测下一个词”这个任务，会不断调整每个词的向量位置，最终使这些向量能够蕴含丰富的语义信息。一旦我们将词转换成了向量，我们就可以用数学工具来度量它们之间的关系。最常用的方法是余弦相似度 (Cosine Similarity)，它通过计算两个向量夹角的余弦值来衡量它们的相似性。

$\text{similarity}(\text{vec}\{a\}, \text{vec}\{b\}) = \cos(\theta) = \frac{\text{vec}\{a\} \cdot \text{vec}\{b\}}{\|\text{vec}\{a\}\| \|\text{vec}\{b\}\|}$

这个公式的含义是：

- 如果两个向量方向完全相同，夹角为 0° ，余弦值为1，表示完全相关。
- 如果两个向量方向正交，夹角为 90° ，余弦值为0，表示毫无关系。
- 如果两个向量方向完全相反，夹角为 180° ，余弦值为-1，表示完全负相关。

通过这种方式，词向量不仅能捕捉到“同义词”这类简单的关系，还能捕捉到更复杂的类比关系。

一个著名的例子展示了词向量捕捉到的语义关系： $\text{vector}('King') - \text{vector}('Man') + \text{vector}('Woman')$ 这个向量运算的结果，在向量空间中与 $\text{vector}('Queen')$ 的位置惊人地接近。这好比在进行语义的平移：我们从“国王”这个点出发，减去“男性”的向量，再加上“女性”的向量，最终就抵达了“女王”的位置。这证明了词嵌入能够学习到“性别”、“皇室”这类抽象概念。

```
import numpy as np

# 假设我们已经学习到了简化的二维词向量
embeddings = {
    "king": np.array([0.9, 0.8]),
    "queen": np.array([0.9, 0.2]),
    "man": np.array([0.7, 0.9]),
    "woman": np.array([0.7, 0.3])
}

def cosine_similarity(vec1, vec2):
    dot_product = np.dot(vec1, vec2)
    norm_product = np.linalg.norm(vec1) * np.linalg.norm(vec2)
    return dot_product / norm_product

# king - man + woman
result_vec = embeddings["king"] - embeddings["man"] + embeddings["woman"]

# 计算结果向量与 "queen" 的相似度
sim = cosine_similarity(result_vec, embeddings["queen"])

print(f"king - man + woman 的结果向量: {result_vec}")
print(f"该结果与 'queen' 的相似度: {sim:.4f}")

>>>
king - man + woman 的结果向量: [0.9 0.2]
该结果与 'queen' 的相似度: 1.0000
```

神经网络语言模型通过词嵌入，成功解决了 N-gram 模型的泛化能力差的问题。然而，它仍然有一个类似 N-gram 的限制：上下文窗口是固定的。它只能考虑固定数量的前文，这为能处理任意长序列的循环神经网络埋下了伏笔。

(3) 循环神经网络 (RNN) 与长短时记忆网络 (LSTM)

前一节的神经网络语言模型虽然引入了词嵌入解决了泛化问题，但它和 N-gram 模型一样，上下文窗口是固定大小的。为了预测下一个词，它只能看到前 n-1 个词，再早的历史信息就被丢弃了。这显然不符合我们人类理解语言的方式。为了打破固定窗口的限制，循环神经网络 (Recurrent Neural Network, RNN) 应运而生，其核心思想非常直观：为网络增加“记忆”能力^[2]。

如图3.3所示，RNN 的设计引入了一个隐藏状态 (hidden state) 向量，我们可以将其理解为网络的短期记忆。在处理序列的每一步，网络都会读取当前的输入词，并结合它上一刻的记忆（即上一个时间步的隐藏状态），然后生成一个新的记忆（即当前时间步的隐藏状态）传递给下一刻。这个循环往复的过程，使得信息可以在序列中不断向后传递。

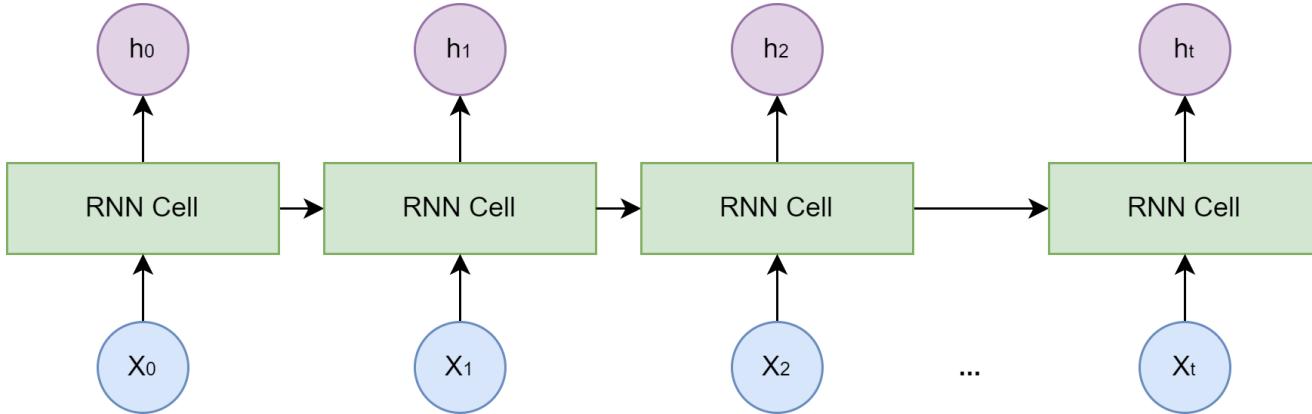


图 3.3 RNN 结构示意图

然而，标准的 RNN 在实践中存在一个严重的问题：长期依赖问题 (Long-term Dependency Problem)。在训练过程中，模型需要通过反向传播算法根据输出端的误差来调整网络深处的权重。对于 RNN 而言，序列的长度就是网络的深度。当序列很长时，梯度在从后向前传播的过程中会经过多次连乘，这会导致梯度值快速趋向于零（梯度消失）或变得极大（梯度爆炸）。梯度消失使得模型无法有效学习到序列早期信息对后期输出的影响，即难以捕捉长距离的依赖关系。

为了解决长期依赖问题，长短时记忆网络 (Long Short-Term Memory, LSTM) 被设计出来^[3]。LSTM 是一种特殊的 RNN，其核心创新在于引入了细胞状态 (Cell State) 和一套精密的门控机制 (Gating Mechanism)。细胞状态可以看作是一条独立于隐藏状态的信息通路，允许信息在时间步之间更顺畅地传递。门控机制则是由几个小型神经网络构成，它们可以学习如何有选择地让信息通过，从而控制细胞状态中信息的增加与移除。这些门包括：

- 遗忘门 (Forget Gate)：决定从上一时刻的细胞状态中丢弃哪些信息。
- 输入门 (Input Gate)：决定将当前输入中的哪些新信息存入细胞状态。
- 输出门 (Output Gate)：决定根据当前的细胞状态，输出哪些信息到隐藏状态。

3.1.2 Transformer 架构解析

在上一节中，我们看到RNN及LSTM通过引入循环结构来处理序列数据，这在一定程度上解决了捕捉长距离依赖的问题。然而，这种循环的计算方式也带来了新的瓶颈：它必须按顺序处理数据。第 t 个时间步的计算，必须等待第 t-1 个时间步完成后才能开始。这意味着 RNN 无法进行大规模的并行计算，在处理长序列时效率低下，这极大地限制了模型规模和训练速度的提升。Transformer在2017 年由谷歌团队提出^[4]。它完全抛弃了循环结构，转而完全依赖一种名为注意力 (Attention) 的机制来捕捉序列内的依赖关系，从而实现了真正意义上的并行计算。

(1) Encoder-Decoder 整体结构

最初的 Transformer 模型是为端到端任务机器翻译而设计的。如图3.4所示，它在宏观上遵循了一个经典的编码器-解码器 (Encoder-Decoder) 架构。

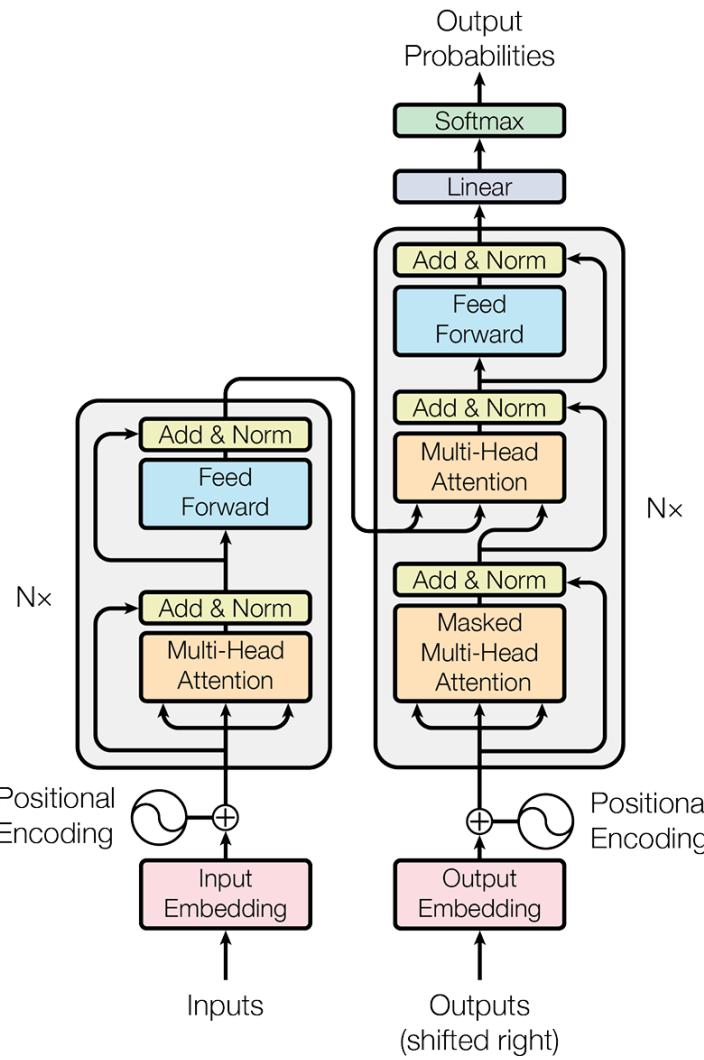


图 3.4 Transformer 整体架构图

我们可以将这个结构理解为一个分工明确的团队：

1. 编码器 (Encoder)：任务是“理解”输入的整个句子。它会读取所有输入词元(这个概念会在3.2.2节介绍)，最终为每个词元生成一个富含上下文信息的向量表示。
2. 解码器 (Decoder)：任务是“生成”目标句子。它会参考自己已经生成的前文，并“咨询”编码器的理解结果，来生成下一个词。

为了真正理解 Transformer 的工作原理，最好的方法莫过于亲手实现它。在本节中，我们将采用一种“自顶向下”的方法：首先，我们搭建出 Transformer 完整的代码框架，定义好所有需要的类和方法。然后，我们将像完成拼图一样，逐一实现这些类的具体功能。

```
import torch
import torch.nn as nn
import math

# --- 占位符模块，将在后续小节中实现 ---
class PositionalEncoding(nn.Module):
    .....
    # 位置编码模块
    .....
    def forward(self, x):
        pass

class MultiHeadAttention(nn.Module):
    .....
    # 多头注意力机制模块
    .....
    def forward(self, query, key, value, mask):
        pass

class PositionWiseFeedForward(nn.Module):
    .....
    # 位置前馈网络模块
    .....
    def forward(self, x):
        pass

# --- 编码器核心层 ---
```

```

class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention() # 待实现
        self.feed_forward = PositionWiseFeedForward() # 待实现
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        # 残差连接与层归一化将在 3.1.2.4 节中详细解释
        # 1. 多头自注意力
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))

        # 2. 前馈网络
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))

    return x

# --- 解码器核心层 ---

```

```

class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention() # 待实现
        self.cross_attn = MultiHeadAttention() # 待实现
        self.feed_forward = PositionWiseFeedForward() # 待实现
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, encoder_output, src_mask, tgt_mask):
        # 1. 掩码多头自注意力（对自己）
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_output))

        # 2. 交叉注意力（对编码器输出）
        cross_attn_output = self.cross_attn(x, encoder_output, encoder_output, src_mask)
        x = self.norm2(x + self.dropout(cross_attn_output))

        # 3. 前馈网络
        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))

    return x

```

(2) 从自注意力到多头注意力

现在，我们来填充骨架中最关键的模块，注意力机制。

想象一下我们阅读这个句子：“The agent learns because **it** is intelligent.”。当我们读到加粗的“**it**”时，为了理解它的指代，我们的大脑会不自觉地将更多的注意力放在前面的“agent”这个词上。自注意力 (**Self-Attention**) 机制就是对这种现象的数学建模。它允许模型在处理序列中的每一个词时，都能兼顾句子中的所有其他词，并为这些词分配不同的“注意力权重”。权重越高的词，代表其与当前词的关联性越强，其信息也应该在当前词的表示中占据更大的比重。

为了实现上述过程，自注意力机制为每个输入的词元向量引入了三个可学习的角色：

- **查询 (Query, Q)**: 代表当前词元，它正在主动地“查询”其他词元以获取信息。
- **键 (Key, K)**: 代表句子中可被查询的词元“标签”或“索引”。
- **值 (Value, V)**: 代表词元本身所携带的“内容”或“信息”。

这三个向量都是由原始的词嵌入向量乘以三个不同的、可学习的权重矩阵 (W^Q, W^K, W^V) 得到的。整个计算过程可以分为以下几步，我们可以把它想象成一次高效的开卷考试：

- 准备“考题”和“资料”：对于句子中的每个词，都通过权重矩阵生成其 Q, K, V 向量。
- 计算相关性得分：要计算词A的新表示，就用词A的 Q 向量，去和句子中所有词（包括A自己）的 K 向量进行点积运算。这个得分反映了其他词对于理解词A的重要性。
- 稳定化与归一化：将得到的所有分数除以一个缩放因子 $\sqrt{d_k}$ (d_k 是 K 向量的维度)，以防止梯度过小，然后用Softmax函数将分数转换成总和为1的权重，也就是归一化的过程。
- 加权求和：将上一步得到的权重分别乘以每个词对应的 V 向量，然后将所有结果相加。最终得到的向量，就是词A融合了全局上下文信息后的表示。

这个过程可以用一个简洁的公式来概括：

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

如果只进行一次上述的注意力计算（即单头），模型可能会只学会关注一种类型的关联。比如，在处理“**it**”时，可能只学会了关注主语。但语言中的关系是复杂的，我们希望模型能同时关注多种关系（如指代关系、时态关系、从属关系等）。多头注意力机制应运而生。它的思想很简单：把一次做完变成分成几组，分开做，再合并。

它将原始的 Q, K, V 向量在维度上切分成 h 份 (h 就是“头”数)，每一份都独立地进行一次单头注意力的计算。这就好比让 h 个不同的“专家”从不同的角度去审视句子，每个专家都能捕捉到一种不同的特征关系。最后，将这 h 个专家的“意见”（即输出向量）拼接起来，再通过一个线性变换进行整合，就得到了最终的输出。

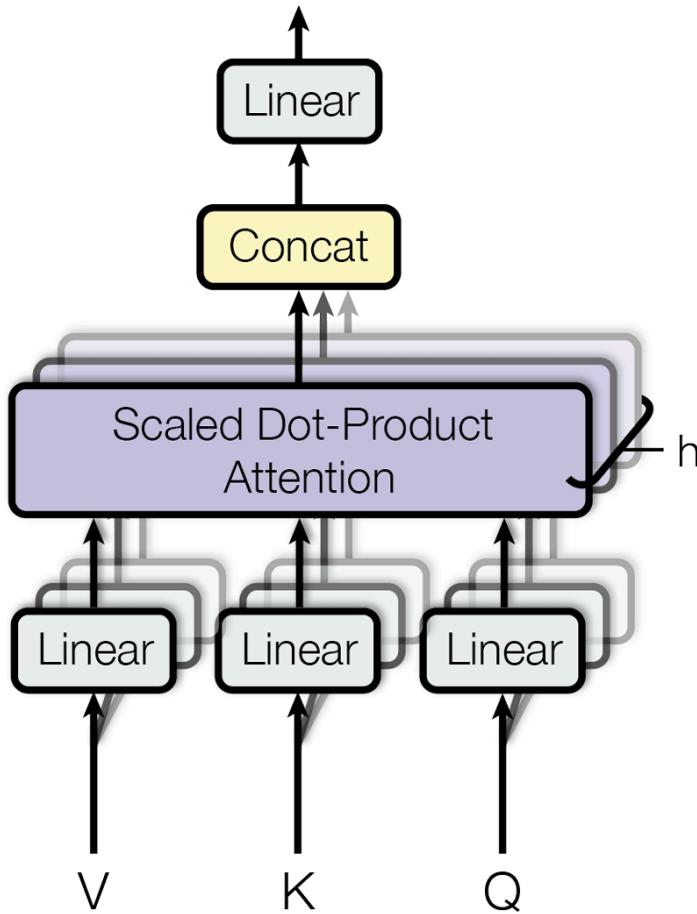


图 3.5 多头注意力机制

如图3.5所示，这种设计让模型能够共同关注来自不同位置、不同表示子空间的信息，极大地增强了模型的表达能力。以下是多头注意力的简单实现可供参考。

```
class MultiHeadAttention(nn.Module):
    ...
    多头注意力机制模块
    ...
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model 必须能被 num_heads 整除"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

    # 定义 Q, K, V 和输出的线性变换层
    self.W_q = nn.Linear(d_model, d_model)
    self.W_k = nn.Linear(d_model, d_model)
    self.W_v = nn.Linear(d_model, d_model)
    self.W_o = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        # 1. 计算注意力得分 (QK^T)
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

        # 2. 应用掩码 (如果提供)
        if mask is not None:
            # 将掩码中为 0 的位置设置为一个非常小的负数，这样 softmax 后会接近 0
            attn_scores = attn_scores.masked_fill(mask == 0, -1e9)

        # 3. 计算注意力权重 (Softmax)
        attn_probs = torch.softmax(attn_scores, dim=-1)

        # 4. 加权求和 (权重 * V)
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        # 将输入 x 的形状从 (batch_size, seq_length, d_model)
```

```

# 变换为 (batch_size, num_heads, seq_length, d_k)
batch_size, seq_length, d_model = x.size()
return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)

def combine_heads(self, x):
    # 将输入 x 的形状从 (batch_size, num_heads, seq_length, d_k)
    # 变回 (batch_size, seq_length, d_model)
    batch_size, num_heads, seq_length, d_k = x.size()
    return x.transpose(1, 2).contiguous().view(batch_size, seq_length, self.d_model)

def forward(self, Q, K, V, mask=None):
    # 1. 对 Q, K, V 进行线性变换
    Q = self.split_heads(self.W_q(Q))
    K = self.split_heads(self.W_k(K))
    V = self.split_heads(self.W_v(V))

    # 2. 计算缩放点积注意力
    attn_output = self.scaled_dot_product_attention(Q, K, V, mask)

    # 3. 合并多头输出并进行最终的线性变换
    output = self.W_o(self.combine_heads(attn_output))
    return output

```

(3) 前馈神经网络

在每个 Encoder 和 Decoder 层中，多头注意力子层之后都跟着一个逐位置前馈网络(**Position-wise Feed-Forward Network, FFN**)。如果说注意力层的作用是从整个序列中“动态地聚合”相关信息，那么前馈网络的作用从这些聚合后的信息中提取更高阶的特征。

这个名字的关键在于“逐位置”。它意味着这个前馈网络会独立地作用于序列中的每一个词元向量。换句话说，对于一个长度为 `seq_len` 的序列，这个 FFN 实际上会被调用 `seq_len` 次，每次处理一个词元。重要的是，所有位置共享的是同一组网络权重。这种设计既保持了对每个位置进行独立加工的能力，又大大减少了模型的参数量。这个网络的结构非常简单，由两个线性变换和一个 ReLU 激活函数组成：

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

其中， x 是注意力子层的输出。 W_1, b_1, W_2, b_2 是可学习的参数。通常，第一个线性层的输出维度 `d_ff` 会远大于输入的维度 `d_model`（例如 `d_ff = 4 * d_model`），经过 ReLU 激活后再通过第二个线性层映射回 `d_model` 维度。这种“先扩大再缩小”的模式，被认为有助于模型学习更丰富的特征表示。

在我们的 PyTorch 骨架中，我们可以用以下代码来实现这个模块：

```

class PositionWiseFeedForward(nn.Module):
    ...
    位置前馈网络模块
    ...
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionWiseFeedForward, self).__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        # x 形状: (batch_size, seq_len, d_model)
        x = self.linear1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.linear2(x)
        # 最终输出形状: (batch_size, seq_len, d_model)
        return x

```

(4) 残差连接与层归一化

在 Transformer 的每个编码器和解码器层中，所有子模块（如多头注意力和前馈网络）都被一个 Add & Norm 操作包裹。这个组合是为了保证 Transformer 能够稳定训练。

这个操作由两个部分组成：

- **残差连接 (Add)**: 该操作将子模块的输入 x 直接加到该子模块的输出 $\text{Sublayer}(x)$ 上。这一结构解决了深度神经网络中的梯度消失 (**Vanishing Gradients**) 问题。在反向传播时，梯度可以绕过子模块直接向前传播，从而保证了即使网络层数很深，模型也能得到有效的训练。其公式可以表示为： $\text{Output} = x + \text{Sublayer}(x)$ 。
- **层归一化 (Norm)**: 该操作对单个样本的所有特征进行归一化，使其均值为0，方差为1。这解决了模型训练过程中的内部协变量偏移 (**Internal Covariate Shift**) 问题，使每一层的输入分布保持稳定，从而加速模型收敛并提高训练的稳定性。

3.1.2.5 位置编码

我们已经了解，Transformer 的核心是自注意力机制，它通过计算序列中任意两个词元之间的关系来捕捉依赖。然而，这种计算方式有一个固有的问题：它本身不包含任何关于词元顺序或位置的信息。对于自注意力来说，“agent learns” 和 “learns agent” 这两个序列是完全等价的，因为它只关心词元之间的关系，而忽略了它们的排列。为了解决这个问题，Transformer 引入了位置编码 (**Positional Encoding**)。

位置编码的核心思想是，为输入序列中的每一个词元嵌入向量，都额外加上一个能代表其绝对位置和相对位置信息的“位置向量”。这个位置向量不是通过学习得到的，而是通过一个固定的数学公式直接计算得出。这样一来，即使两个词元（例如，两个都叫 `agent` 的词元）自身的嵌入是相同的，但由于它们在句子中的位置不同，它们最终输入到 Transformer 模型中的向量就会因为加上了不同的位置编码而变得独一无二。原论文中提出的位置编码使用正弦和余弦函数来生成，其公式如下：

$$\text{PE}_{\{(pos, 2i)\}} = \sin(\frac{\pi}{10000} \cdot pos^{2i/d_{\text{model}}})$$

$$\text{PE}_{\{(pos, 2i+1)\}} = \cos(\frac{\pi}{10000} \cdot pos^{2i/d_{\text{model}}})$$

其中：

- pos 是词元在序列中的位置（例如，0, 1, 2, ...）
- i 是位置向量中的维度索引（从0到 $d_{\text{model}}/2$ ）
- d_{model} 是词嵌入向量的维度（与我们模型中定义的一致）

现在，我们来实现 PositionalEncoding 模块，并完成我们 Transformer 骨架代码的最后一部分。

```
class PositionalEncoding(nn.Module):
    """
    为输入序列的词嵌入向量添加位置编码。
    """

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        # 创建一个足够长的位置编码矩阵
        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))

        # pe (positional encoding) 的大小为 (max_len, d_model)
        pe = torch.zeros(max_len, d_model)

        # 偶数维度使用 sin, 奇数维度使用 cos
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        # 将 pe 注册为 buffer, 这样它就不会被视为模型参数, 但会随模型移动 (例如 to(device))
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x.size(1) 是当前输入的序列长度
        # 将位置编码加到输入向量上
        x = x + self.pe[:, :x.size(1)]
        return self.dropout(x)
```

本小节主要是为了帮助理解 Transformer 的宏观结构和内部每个模块的运作细节。由于是为了补充智能体学习中大模型的知识体系，也就不再继续往下深入实现。至此，我们已经为理解现代大语言模型打下了坚实的架构基础。在下一节中，我们将探讨 Decoder-Only 架构，看看它是如何基于 Transformer 的思想演变而来。

3.1.3 Decoder-Only 架构

前面一节中，我们动手构建了一个完整的Transformer 模型，它能在很多端到端的场景表现出色。但是当任务转换为构建一个与人对话、创作、作为智能体大脑的通用模型时，或许我们并不需要那么复杂的结构。

Transformer的设计哲学是“先理解，再生成”。编码器负责深入理解输入的整个句子，形成一个包含全局信息的上下文记忆，然后解码器基于这份记忆来生成翻译。但 OpenAI 在开发 **GPT (Generative Pre-trained Transformer)** 时，提出了一个更简单的思想^[5]：语言的核心任务，不就是预测下一个最有可能出现的词吗？

无论是回答问题、写故事还是生成代码，本质上都是在一个已有的文本序列后面，一个词一个词地添加最合理的内容。基于这个思想，GPT 做了一个大胆的简化：它完全抛弃了编码器，只保留了解码器部分。这就是 **Decoder-Only** 架构的由来。

Decoder-Only 架构的工作模式被称为自回归 (**Autoregressive**)。这个听起来很专业的术语，其实描述了一个非常简单的过程：

1. 给模型一个起始文本（例如“Datawhale Agent is”）。
2. 模型预测出下一个最有可能的词（例如“a”）。
3. 模型将自己刚刚生成的词“a”添加到输入文本的末尾，形成新的输入（“Datawhale Agent is a”）。
4. 模型基于这个新输入，再次预测下一个词（例如“powerful”）。
5. 不断重复这个过程，直到生成完整的句子或达到停止条件。

模型就像一个在玩“文字接龙”的游戏，它不断地“回顾”自己已经写下的内容，然后思考下一个字该写什么。

你可能会问，解码器是如何保证在预测第 t 个词时，不去“偷看”第 $t+1$ 个词的答案呢？

答案就是掩码自注意力 (**Masked Self-Attention**)。在 Decoder-Only 架构中，这个机制变得至关重要。它的工作原理非常巧妙：

在自注意力机制计算出注意力分数矩阵（即每个词对其他所有词的关注度得分）之后，但在进行 Softmax 归一化之前，模型会应用一个“掩码”。这个掩码会将所有位于当前位置之后（即目前尚未观测到）的词元对应的分数，替换为一个非常大的负数。当这个带有负无穷分数的矩阵经过 Softmax 函数时，这些位置的概率就会变为0。这样一来，模型在计算任何一个位置的输出时，都从数学上被阻止了去关注它后面的信息。这种机制保证了模型在预测下一个词时，能且只能依赖它已经见过的、位于当前位置之前的所有信息，从而确保了预测的公平性和逻辑的连贯性。

Decoder-Only 架构的优势

这种看似简单的架构，却带来了巨大的成功，其优势在于：

- 训练目标统一：模型的唯一任务就是“预测下一个词”，这个简单的目标非常适合在海量的无标注文本数据上进行预训练。
- 结构简单，易于扩展：更少的组件意味着更容易进行规模化扩展。今天的 GPT-4、Llama 等拥有数千亿甚至万亿参数的巨型模型，都是基于这种简洁的架构。
- 天然适合生成任务：其自回归的工作模式与所有生成式任务（对话、写作、代码生成等）完美契合，这也是它能成为构建通用智能体基础的核心原因。

总而言之，从 Transformer 的解码器演变而来的 Decoder-Only 架构，通过“预测下一个词”这一简单的范式，开启了我们今天所处的大语言模型时代。

3.2 与大语言模型交互

3.2.1 提示工程

如果我们把大语言模型比作一个能力极强的“大脑”，那么提示 (**Prompt**) 就是我们与这个“大脑”沟通的语言。提示工程，就是研究如何设计出精准的提示，从而引导模型产生我们期望输出的回复。对于构建智能体而言，一个精心设计的提示能让智能体之间协作分工变得高效。

(1) 模型采样参数

在使用大模型时，你会经常看到类似 Temperature 这类的可配置参数，其本质是通过调整模型对“概率分布”的采样策略，让输出匹配具体场景需求，配置合适的参数可以提升 Agent 在特定场景的性能。

传统的概率分布是由 Softmax 公式计算得到的： $p_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}$ ，采样参数的本质就是在此基础上，根据不同策略“重新调整”或“截断”分布，从而改变大模型输出的下一个 token。

Temperature：温度是控制模型输出“随机性”与“确定性”的关键参数。其原理是引入温度系数 $T > 0$ ，将 Softmax 改写为 $p_i = \frac{e^{z_i/T}}{\sum_{j=1}^k e^{z_j/T}}$ 。

当 T 变小时，分布“更加陡峭”，高概率项权重进一步放大，生成更“保守”且重复率更高的文本。当 T 变大时，分布“更加平坦”，低概率项权重提升，生成更“多样”但可能出现不连贯的内容。

- 低温度 ($0 \leq \text{Temperature} < 0.3$) 时输出更“精准、确定”。适用场景：事实性任务：如问答、数据计算、代码生成；严谨性场景：法律条文解读、技术文档撰写、学术概念解释等场景。
- 中温度 ($0.3 \leq \text{Temperature} < 0.7$)：输出“平衡、自然”。适用场景：日常对话：如客服交互、聊天机器人；常规创作：如邮件撰写、产品文案、简单故事创作。
- 高温度 ($0.7 \leq \text{Temperature} < 2$)：输出“创新、发散”。适用场景：创意性任务：如诗歌创作、科幻故事构思、广告 slogan brainstorm、艺术灵感启发；发散性思考。

Top-k：其原理是将所有 token 按概率从高到低排序，取排名前 k 个的 token 组成“候选集”，随后对筛选出的 k 个 token 的概率进行“归一化”： $\hat{p}_i = \frac{p_i}{\sum_{j=1}^k p_j}$

- 与温度采样的区别与联系：温度采样通过温度 T 调整所有 token 的概率分布（平滑或陡峭），不改变候选 token 的数量（仍考虑全部 N 个）。Top-k 采样通过 k 值限制候选 token 的数量（只保留前 k 个高概率 token），再从其中采样。当 k=1 时输出完全确定，退化为“贪心采样”。

Top-p：其原理是将所有 token 按概率从高到低排序，从排序后的第一个 token 开始，逐步累加概率，直到累积和首次达到或超过阈值 p： $\sum_{i \in S} p_i \geq p$ ，此时累加过程中包含的所有 token 组成“核集合”，最后对核集合进行归一化。

- 与 Top-k 的区别与联系：相对于固定截断大小的 Top-k，Top-p 能动态适应不同分布的“长尾”特性，对概率分布不均匀的极端情况的适应性更好。

在文本生成中，当同时设置 Top-p、Top-k 和温度系数时，这些参数会按照分层过滤的方式协同工作，其优先级顺序为：温度调整 → Top-k → Top-p。温度调整整体分布的陡峭程度，Top-k 会先保留概率最高的 k 个候选，然后 Top-p 会从 Top-k 的结果中选取累积概率 $\geq p$ 的最小集合作为最终的候选集。不过，通常 Top-k 和 Top-p 二选一即可，若同时设置，实际候选集为两者的交集。需要注意的是，如果将温度设置为 0，则 Top-k 和 Top-p 将变得无关紧要，因为最有可能的 Token 将成为下一个预测的 Token；如果将 Top-k 设置为 1，温度和 Top-p 也将变得无关紧要，因为只有一个 Token 通过 Top-k 标准，它将是下一个预测的 Token。

(2) 零样本、单样本与少样本提示

根据我们给模型提供示例 (Exemplar) 的数量，提示可以分为三种类型。为了更好地理解它们，让我们以一个情感分类任务为例，目标是让模型判断一段文本的情感色彩（如正面、负面或中性）。

零样本提示 (Zero-shot Prompting) 这指的是我们不给模型任何示例，直接让它根据指令完成任务。这得益于模型在海量数据上预训练后获得的强大泛化能力。

案例：我们直接向模型下达指令，要求它完成情感分类任务。

文本：Datawhale 的 AI Agent 课程非常棒！

情感：正面

单样本提示 (One-shot Prompting) 我们给模型提供一个完整的示例，向它展示任务的格式和期望的输出风格。

案例：我们先给模型一个完整的“问题-答案”对作为示范，然后提出我们的新问题。

文本：这家餐厅的服务太慢了。

情感：负面

文本:Datawhale的AI Agent课程非常棒!

情感:

模型会模仿给出的示例格式，为第二段文本补全“正面”。

少样本提示 (Few-shot Prompting) 我们提供多个示例，这能让模型更准确地理解任务的细节、边界和细微差别，从而获得更好的性能。

案例：我们提供涵盖了不同情况的多个示例，让模型对任务有更全面的理解。

文本:这家餐厅的服务太慢了。

情感:负面

文本:这部电影的情节很平淡。

情感:中性

文本:Datawhale的AI Agent课程非常棒!

情感:

模型会综合所有示例，更准确地将最后一句的情感分类为“正面”。

(3) 指令调优的影响

早期的 GPT 模型（如 GPT-3）主要是“文本补全”模型，它们擅长根据前面的文本续写，但不一定能很好地理解并执行人类的指令。

指令调优 (Instruction Tuning) 是一种微调技术，它使用大量“指令-回答”格式的数据对预训练模型进行进一步的训练。经过指令调优后，模型能更好地理解并遵循用户的指令。我们今天日常工作学习中使用的所有模型（如 ChatGPT, DeepSeek, Qwen）都是其模型家族中经过指令调优过的模型。

- 对“文本补全”模型的提示(你需要用少样本提示“教会”模型做什么):

这是一段将英文翻译成中文的程序。

英文:Hello

中文:你好

英文:How are you?

中文:

- 对“指令调优”模型的提示(你可以直接下达指令):

请将下面的英文翻译成中文：

How are you?

指令调优的出现，极大地简化了我们与模型交互的方式，使得直接、清晰的自然语言指令成为可能。

(4) 基础提示技巧

角色扮演 (Role-playing) 通过赋予模型一个特定的角色，我们可以引导它的回答风格、语气和知识范围，使其输出更符合特定场景的需求。

案例

你现在是一位资深的Python编程专家。请解释一下Python中的GIL（全局解释器锁）是什么，要让一个初学者也能听懂。

上下文示例 (In-context Example) 这与少样本提示的思想一致，通过在提示中提供清晰的输入输出示例，来“教会”模型如何处理我们的请求，尤其是在处理复杂格式或特定风格的任务时非常有效。

案例

我需要你从产品评论中提取产品名称和用户情感。请严格按照下面的JSON格式输出。

评论:这款“星尘”笔记本电脑的屏幕显示效果惊人，但我不太喜欢它的键盘手感。

输出:{"product_name": "星尘笔记本电脑", "sentiment": "混合"}

评论:我刚买的“声动”耳机音质很棒，续航也超出了我的预期！

输出:

(5) 思维链

对于需要逻辑推理、计算或多步骤思考的复杂问题，直接让模型给出答案往往容易出错。思维链 (Chain-of-Thought, CoT) 是一种强大的提示技巧，它通过引导模型“一步一步地思考”，提升了模型在复杂任务上的推理能力。

实现 CoT 的关键，是在提示中加入一句简单的引导语，如“请逐步思考”或“Let's think step by step”。

思维链提示

一个篮球队在一个赛季的80场比赛中赢了60%。在接下来的赛季中，他们打了15场比赛，赢了12场。两个赛季的总胜率是多少？

请一步一步地思考并解答。

>>>

(模型可能会输出)

好的，我们来一步一步计算。

第一步:计算第一个赛季赢得的比赛数。

80场 * 60% = 48场。

第二步:计算两个赛季的总比赛数和总胜利数。

总比赛数 = 80 + 15 = 95场。

总胜利数 = 48 + 12 = 60场。

第三步:计算总胜率。

总胜率 = (总胜利数 / 总比赛数) * 100% = (60 / 95) * 100% ≈ 63.16%。
所以，两个赛季的总胜率约为63.16%。

通过显式地展示其推理过程，模型不仅更容易得出正确的答案，也让它的回答变得更可信、更易于我们检查和纠正。

3.2.2 文本分词

我们知道，计算机本质上只能理解数字。因此，在将自然语言文本喂给大语言模型之前，必须先将其转换成模型能够处理的数字格式。这个将文本序列转换为数字序列的过程，就叫做分词 (Tokenization)。分词器 (Tokenizer) 的作用，就是定义一套规则，将原始文本切分成一个个最小的单元，我们称之为词元 (Token)。

3.2.2.1 为何需要分词

早期的自然语言处理任务可能会采用简单的分词策略：

- 按词分词 (Word-based)：直接用空格或标点符号将句子切分成单词。这种方法很直观，但也面临挑战：
 - 词表爆炸与未登录词：一个语言的词汇量是巨大的，如果每个词都作为一个独立的词元，词表会变得难以管理。更糟糕的是，模型将无法处理任何未在词表中出现过的词（例如“DatawhaleAgent”），这种现象我们称为“未登录词”(Out-Of-Vocabulary, OOV)。
 - 语义关联的缺失：模型难以捕捉词形相近的词之间的语义关系。例如，“look”、“looks”和“looking”会被视为三个完全不同的词元，尽管它们有共同的核心含义。同样，训练数据中的低频词由于出现次数少，其语义也难以被模型充分学习。
- 按字符分词 (Character-based)：将文本切分成单个字符。这种方法词表很小（例如英文字母、数字和标点），不存在 OOV 问题。但它的缺点是，单个字符大多不具备独立的语义，模型需要花费更多的精力去学习如何将字符组合成有意义的词，导致学习效率低下。

为了兼顾词表大小和语义表达，现代大语言模型普遍采用子词分词 (Subword Tokenization) 算法。它的核心思想是：将常见的词（如“agent”）保留为完整的词元，同时将不常见的词（如“Tokenization”）拆分成多个有意义的子词片段（如“Token”和“ization”）。这样既控制了词表的大小，又能让模型通过组合子词来理解和生成新词。

3.2.2.2 字节对编码算法解析

字节对编码 (Byte-Pair Encoding, BPE) 是最主流的子词分词算法之一^[6]，GPT 系列模型就采用了这种算法。其核心思想非常简洁，可以理解为一个“贪心”的合并过程：

- 初始化：将词表初始化为所有在语料库中出现过的基本字符。
- 迭代合并：在语料库上，统计所有相邻词元对的出现频率，找到频率最高的一对，将它们合并成一个新的词元，并加入词表。
- 重复：重复第 2 步，直到词表大小达到预设的阈值。

案例演示：假设我们的迷你语料库是 {"hug": 1, "pug": 1, "pun": 1, "bun": 1}，并且我们想构建一个大小为 10 的词表。BPE 的训练过程可以用下表3.1 来表示：

表 3.1 BPE 算法合并过程示例

步骤	最高频词元对	频率	合并为	新增词元	当前词表大小
初始化	-	-	-	{h, u, g, p, n, b}	6
合并 1	u, g	2	ug	ug	7
合并 2	u, n	2	un	un	8
合并 3	p, ug	1	pug	pug	9
合并 4	p, un	1	pun	pun	10

训练结束后，词表大小达到 10，我们就得到了新的分词规则。现在，对于一个未见过的词“bug”，分词器会先查找“bug”是否在词表中，发现不在；然后查找“bu”，发现不在；最后查找“b”和“ug”，发现都在，于是将其切分为 ['b', 'ug']。

下面我们用一段简单的 Python 代码来模拟上述过程：

```
import re, collections

def get_stats(vocab):
    """统计词元对频率"""
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i],symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    """合并词元对"""
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\\S)' + bigram + r'(?!\\S)')
    for word in v_in:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out
```

```

v_out[w_out] = v_in[word]
return v_out

# 准备语料库，每个词末尾加上</w>表示结束，并切分好字符
vocab = {'h u g </w>': 1, 'p u g </w>': 1, 'p u n </w>': 1, 'b u n </w>': 1}
num_merges = 4 # 设置合并次数

for i in range(num_merges):
    pairs = get_stats(vocab)
    if not pairs:
        break
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
    print(f"第{i+1}次合并: {best} -> {''.join(best)}")
    print(f"新词表(部分): {list(vocab.keys())}")
    print("-" * 20)

>>>
第1次合并: ('u', 'g') -> ug
新词表(部分): ['h ug </w>', 'p ug </w>', 'p u n </w>', 'b u n </w>']

第2次合并: ('ug', '</w>') -> ug</w>
新词表(部分): ['h ug</w>', 'p ug</w>', 'p u n </w>', 'b u n </w>']

第3次合并: ('u', 'n') -> un
新词表(部分): ['h ug</w>', 'p ug</w>', 'p un </w>', 'b un </w>']

第4次合并: ('un', '</w>') -> un</w>
新词表(部分): ['h ug</w>', 'p ug</w>', 'p un</w>', 'b un</w>']

```

这段代码清晰地展示了 BPE 算法如何通过迭代合并最高频的相邻词元对，来逐步构建和扩充词表的过程。

后续的许多算法都是在 BPE 的基础上进行优化的。其中，Google 开发的 WordPiece 和 SentencePiece 是影响力最大的两种。

- **WordPiece**: Google BERT 模型采用的算法^[7]。它与 BPE 非常相似，但合并词元的标准不是“最高频率”，而是“能最大化提升语料库的语言模型概率”。简单来说，它会优先合并那些能让整个语料库的“通顺度”提升最大的词元对。
- **SentencePiece**: Google 开源的一款分词工具^[8]，Llama 系列模型采用了此算法。它最大的特点是，将空格也视作一个普通字符（通常用下划线_ 表示）。这使得分词和解码过程完全可逆，且不依赖于特定的语言（例如，它不需要知道中文不使用空格分词）。

3.2.2.3 分词器对开发者的意义

理解分词算法的细节并非目的，但作为智能体的开发者，理解分词器的实际影响十分重要，这直接关系到智能体的性能、成本和稳定性：

- **上下文窗口限制**: 模型的上下文窗口（如 8K, 128K）是以 **Token** 数量计算的，而不是字符数或单词数。同样一段话，在不同语言（如中英文）或不同分词器下，Token 数量可能相差巨大。精确管理输入长度、避免超出上下文限制是构建长时记忆智能体的基础。
- **API 成本**: 大多数模型 API 都是按 Token 数量计费的。了解你的文本会被如何分词，是预估和控制智能体运行成本的关键一步。
- **模型表现的异常**: 有时模型的奇怪表现根源在于分词。例如，模型可能很擅长计算 2 + 2，但对于 2+2（没有空格）就可能出错，因为后者可能被分词器视为一个独立的、不常见的词元。同样，一个词因为首字母大小写不同，也可能被切分成完全不同的 Token 序列，从而影响模型的理解。在设计提示词和解析模型输出时，考虑到这些“陷阱”有助于提升智能体的鲁棒性。

3.2.3 调用开源大语言模型

在本书的第一章，我们通过 API 来与大语言模型进行交互，以此驱动我们的智能体。这是一种快速、便捷的方式，但并非唯一的方式。对于许多需要处理敏感数据、希望离线运行或想精细控制成本的场景，将大语言模型直接部署在本地就显得至关重要。

Hugging Face Transformers 是一个强大的开源库，它提供了标准化的接口来加载和使用数以万计的预训练模型。我们将使用它来完成本次实践。

配置环境与选择模型: 为了让大多数读者都能在个人电脑上顺利运行，我们特意选择了一个小规模但功能强大的模型：Qwen/Qwen1.5-0.5B-Chat。这是一个由阿里巴巴达摩院开源的拥有约 5 亿参数的对话模型，它体积小、性能优异，非常适合入门学习和本地部署。

首先，请确保你已经安装了必要的库：

```
pip install transformers torch
```

在 **transformers** 库中，我们通常使用 **AutoModelForCausalLM** 和 **AutoTokenizer** 这两个类来自动加载与模型匹配的权重和分词器。下面这段代码会自动从 Hugging Face Hub 下载所需的模型文件和分词器配置，这可能需要一些时间，具体取决于你的网络速度。

```

import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

# 指定模型ID
model_id = "Qwen/Qwen1.5-0.5B-Chat"

# 设置设备，优先使用GPU
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device: {}".format(device))

# 加载分词器
tokenizer = AutoTokenizer.from_pretrained(model_id)

# 加载模型，并将其移动到指定设备

```

2/9/26, 1:56 PM

```
model = AutoModelForCausalLM.from_pretrained(model_id).to(device)
```

```
print("模型和分词器加载完成! ")
```

我们来创建一个对话提示，Qwen1.5-Chat 模型遵循特定的对话模板。然后，可以将使用上一步加载的 `tokenizer` 将文本提示转换为模型能够理解的数字 ID（即 Token ID）。

```
# 准备对话输入
```

```
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "你好，请介绍你自己。"}
]
```

```
# 使用分词器的模板格式化输入
```

```
text = tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
)
```

```
# 编码输入文本
```

```
model_inputs = tokenizer([text], return_tensors="pt").to(device)
```

```
print("编码后的输入文本:")
```

```
print(model_inputs)
```

```
>>>
```

```
{'input_ids': tensor([[151644, 8948, 198, 2610, 525, 264, 10950, 17847, 13, 151645, 198, 151644, 872, 198, 108386, 37945, 100157, 107828, 1773
device='cuda:0'])}
```

现在可以调用模型的 `generate()` 方法来生成回答了。模型会输出一系列 Token ID，这代表了它的回答。

最后，我们需要使用分词器的 `decode()` 方法，将这些数字 ID 翻译回人类可以阅读的文本。

```
# 使用模型生成回答
```

```
# max_new_tokens 控制了模型最多能生成多少个新的Token
generated_ids = model.generate(
    model_inputs.input_ids,
    max_new_tokens=512
)
```

```
# 将生成的 Token ID 截取掉输入部分
```

```
# 这样我们只解码模型新生成的部分
generated_ids = [
    output_ids[len(input_ids):] for input_ids, output_ids in zip(model_inputs.input_ids, generated_ids)
]
```

```
# 解码生成的 Token ID
```

```
response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
```

```
print("\n模型的回答:")
```

```
print(response)
```

```
>>>
```

```
我叫通义千问，是由阿里云研发的预训练语言模型，可以回答问题、创作文字，还能表达观点、撰写代码。我主要的功能是在多个领域提供帮助，包括但不限于：语言理解、文本生成、机器翻译、问答系统等。有什么我可以帮到你的吗？
```

当你运行完所有代码后，你将会在本地电脑上看到模型生成的关于Qwen模型的介绍。恭喜你，你已经成功地在本地部署并运行了一个开源大语言模型！

3.2.4 模型的选择

在上一节中，我们成功地在本地运行了一个小型的开源语言模型。这自然引出了一个对于智能体开发者而言至关重要的问题：在当前数百个模型百花齐放的背景下，我们应当如何为特定的任务选择最合适模型？

选择语言模型并非简单地追求“最大、最强”，而是一个在性能、成本、速度和部署方式之间进行权衡的决策过程。本节将首先梳理模型选型的几个关键考量因素，然后对当前主流的闭源与开源模型进行梳理。

由于大语言模型技术正处于高速发展阶段，新模型、新版本层出不穷，迭代速度极快。本节在撰写时力求提供当前主流模型的概览和选型考量，但请读者注意，文中所提及的具体模型版本和性能数据可能随时间推移而发生变化，且只列举了部分工作并不完整。我们更侧重于介绍其核心技术特点、发展趋势以及在智能体开发中的通用选型原则。

3.2.4.1 模型选型的关键考量

在为您的智能体选择大语言模型时，可以从以下几个维度进行综合评估：

- **性能与能力**：这是最核心的考量。不同的模型擅长的任务不同，有的长于逻辑推理和代码生成，有的则在创意写作或多语言翻译上更胜一筹。您可以参考一些公开的基准测试排行榜（如 LMSys Chatbot Arena Leaderboard）来评估模型的综合能力。
- **成本**：对于闭源模型，成本主要体现在 API 调用费用，通常按 Token 数量计费。对于开源模型，成本则体现在本地部署所需的硬件（GPU、内存）和运维上。需要根据应用的预期使用量和预算做出选择。
- **速度（延迟）**：对于需要实时交互的智能体（如客服、游戏 NPC），模型的响应速度至关重要。一些轻量级或经过优化的模型（如 GPT-3.5 Turbo, Claude 3.5 Sonnet）在延迟上表现更优。
- **上下文窗口**：模型能一次性处理的 Token 数量上限。对于需要理解长文档、分析代码库或维持长期对话记忆的智能体，选择一个拥有较大上下文窗口（如 128K Token 或更高）的模型是必要的。

- 部署方式：使用 API 的方式最简单便捷，但数据需要发送给第三方，且受限于服务商的条款。本地部署则能确保数据隐私和最高程度的自主可控，但对技术和硬件要求更高。
- 生态与工具链：一个模型的流行程度也决定了其周边生态的成熟度。主流模型通常拥有更丰富的社区支持、教程、预训练模型、微调工具和兼容的开发框架（如 LangChain, LlamaIndex, Hugging Face Transformers），这能极大地加速开发进程、降低开发难度。选择一个拥有活跃社区和完善工具链的模型，可以在遇到问题时更容易找到解决方案和资源。
- 可微调性与定制化：对于需要处理特定领域数据或执行特定任务的智能体，模型的微调能力至关重要。一些模型提供了便捷的微调接口和工具，允许开发者使用自己的数据集对模型进行定制化训练，从而显著提升模型在特定场景下的性能和准确性。开源模型在这方面通常提供更大的灵活性。
- 安全性与伦理：随着大语言模型的广泛应用，其潜在的安全风险和伦理问题也日益凸显。选择模型时，需要考虑其在偏见、毒性、幻觉等方面的表现，以及服务商或开源社区在模型安全和负责任AI方面的投入。对于面向公众或涉及敏感信息的应用，模型的安全性和伦理合规性是不可忽视的考量。

3.2.4.2 闭源模型概览

闭源模型通常代表了当前 AI 技术的最前沿，并提供稳定、易用的 API 服务，是构建高性能智能体的首选。

1. **OpenAI GPT 系列：**从开启大模型时代的 GPT-3，到引入 RLHF（人类反馈强化学习）、实现与人类意图对齐的 ChatGPT，再到开启多模态时代的 GPT-4，OpenAI 持续引领行业发展。最新的 GPT-5 更是将多模态能力和通用智能水平提升到新的高度，能够无缝处理文本、音频和图像输入，并生成相应的输出，其响应速度和自然度也大幅提升，尤其在实时语音对话方面表现出色。
2. **Google Gemini 系列：**Google DeepMind 推出的 Gemini 系列模型是原生多模态的代表，其核心特点是能统一处理文本、代码、音视频和图像等多种模态的数据，并以其超长的上下文窗口在海量信息处理上具备优势。Gemini Ultra 是其最强大的模型，适用于高度复杂的任务；Gemini Pro 适用于广泛的任务，提供高性能和效率；Gemini Nano 则针对设备端部署进行了优化。最新的 Gemini 2.5 系列模型，如 Gemini 2.5 Pro 和 Gemini 2.5 Flash，进一步提升了推理能力和上下文窗口，特别是 Gemini 2.5 Flash 以其更快的推理速度和成本效益，适用于需要快速响应的场景。
3. **Anthropic Claude 系列：**Anthropic 是一家专注于 AI 安全和负责任 AI 的公司，其 Claude 系列模型从设计之初就将 AI 安全放在首位，以其在处理长文档、减少有害输出、遵循指令方面的可靠性而闻名，深受企业级应用青睐。Claude 3 系列包括 Claude 3 Opus（最智能、性能最强）、Claude 3 Sonnet（性能与速度兼顾的平衡之选）和 Claude 3 Haiku（最快、最紧凑的模型，适用于近乎实时的交互）。最新的 Claude 4 系列模型，如 Claude 4 Opus，在通用智能、复杂推理和代码生成方面取得了显著进展，进一步提升了处理长上下文和多模态任务的能力。
4. 国内主流模型：中国在大语言模型领域涌现出众多具有竞争力的闭源模型，以百度文心一言(ERNIE Bot)、腾讯混元(Hunyuan)、华为盘古(Pangu-α)、科大讯飞星火(SparkDesk)和月之暗面(Moonshot AI)等为代表的国产模型，在中文处理上具备天然优势，并深度赋能本土产业。

3.2.4.3 开源模型概览

开源模型为开发者提供了最大程度的灵活性、透明度和自主性，催生了繁荣的社区生态。它们允许开发者在本地部署、进行定制化微调，并拥有完整的模型控制权。

- **Meta Llama 系列：**Meta 推出的 Llama 系列是开源大语言模型的重要里程碑。该系列凭借出色的综合性能、开放的许可协议和强大的社区支持，成为许多衍生项目和研究的基座。Llama 4 系列于2025年4月发布，是Meta首批采用混合专家（MoE）架构的模型，该架构通过仅激活处理特定任务所需的模型部分来显著提升计算效率。该系列包含三款定位分明的模型：LLama 4 Scout 支持1000万token的上下文窗口专为长文档分析和移动端部署设计。Llama 4 Maverick 专注于多模态能力，在编码、复杂推理及多语言支持方面表现卓越。Llama 4 Behemoth多项STEM基准测试中表现超越竞争对手。是Meta目前最强大的模型
- **Mistral AI 系列：**来自法国的 Mistral AI 以其“小尺寸、高性能”的模型设计而闻名。其最新模型 Mistral Medium 3.1 于2025年8月发布，在代码生成、STEM推理和跨领域问答等任务上准确率与响应速度均有显著提升，基准测试表现优于 Claude Sonnet 3.7 与 Llama 4 Maverick 等同级模型。它具备原生多模态能力，可同时处理图像与文字混合输入，并内置“语调适配层”，帮助企业更轻松实现符合品牌调性的输出。
- 国内开源力量：国内厂商和科研机构也在积极拥抱开源，例如阿里巴巴的通义千问 (Qwen) 系列和清华大学与智谱 AI 合作的 ChatGLM 系列，它们提供了强大的中文能力，并围绕自身构建了活跃的社区。

对于智能体开发者而言，闭源模型提供了“开箱即用”的便捷，而开源模型则赋予了我们“随心所欲”的定制自由。理解这两大阵营的特点和代表模型，是为我们的智能体项目做出明智技术选型的第一步。

3.3 大语言模型的缩放法则与局限性

大语言模型（LLMs）在近年来取得了令人瞩目的进展，其能力边界不断拓展，应用场景日益丰富。然而，这些成就的背后，离不开对模型规模、数据量和计算资源之间关系的深刻理解，即缩放法则（Scaling Laws）。同时，作为新兴技术，LLMs 也面临着诸多挑战和局限性。本节将深入探讨这些核心概念，旨在帮助读者全面理解 LLMs 的能力边界，从而在构建智能体时扬长避短。

3.3.1 缩放法则

缩放法则（Scaling Laws）是近年来大语言模型领域最重要的发现之一。它揭示了模型性能与模型参数量、训练数据量以及计算资源之间存在着可预测的幂律关系。这一发现为大语言模型的持续发展提供了理论指导，阐明了增加资源投入能够系统性提升模型性能的底层逻辑。

研究发现，在对数-对数坐标系下，模型的性能（通常用损失 Loss 来衡量）与参数量、数据量和计算量这三个因素都呈现出平滑的幂律关系^[9]。简单来说，只要我们持续、按比例地增加这三个要素，模型的性能就会可预测地、平滑地提升，而不会出现明显的瓶颈。这一发现为大模型的设计和训练提供了清晰的指导：在资源允许的范围内，尽可能地扩大模型规模和训练数据量。

早期的研究更侧重于增加模型参数量，但 DeepMind 在 2022 年提出的“Chinchilla 定律”对此进行了重要修正^[10]。该定律指出，在给定的计算预算下，为了达到最优性能，模型参数量和训练数据量之间存在一个最优配比。具体来说，最优的模型应该比之前普遍认为的要小，但需要用多得多的数据进行训练。例如，一个 700 亿参数的 Chinchilla 模型，由于使用了比 GPT-3（1750 亿参数）多 4 倍的数据进行训练，其性能反而超越了后者。这一发现纠正了“越大越好”的片面认知，强调了数据效率的重要性，并指导了后续许多高效大模型（如 Llama 系列）的设计。

缩放法则最令人惊奇的产物是“能力的涌现”。所谓能力涌现，是指当模型规模达到一定阈值后，会突然展现出在小规模模型中完全不存在或表现不佳的全新能力。例如，链式思考（Chain-of-Thought）、指令遵循（Instruction Following）、多步推理、代码生成等能力，都是在模型参数量达到数百亿甚至千亿级别后

才显著出现的。这种现象表明，大语言模型不仅仅是简单地记忆和复述，它们在学习过程中可能形成了某种更深层次的抽象和推理能力。对于智能体开发者而言，能力的涌现意味着选择一个足够大规模的模型，是实现复杂自主决策和规划能力的前提。

3.3.2 模型幻觉

模型幻觉 (Hallucination) 通常指的是大语言模型生成的内容与客观事实、用户输入或上下文信息相矛盾，或者生成了不存在的事实、实体或事件。幻觉的本质是模型在生成过程中，过度自信地“编造”了信息，而非准确地检索或推理。根据其表现形式，幻觉可以被分为多种类型^[11]，例如：

- **事实性幻觉 (Factual Hallucinations)**：模型生成与现实世界事实不符的信息。
- **忠实性幻觉 (Faithfulness Hallucinations)**：在文本摘要、翻译等任务中，生成的内容未能忠实地反映源文本的含义。
- **内在幻觉 (Intrinsic Hallucinations)**：模型生成的内容与输入信息直接矛盾。

幻觉的产生是多方面因素共同作用的结果。首先，训练数据中可能包含错误或矛盾的信息。其次，模型的自回归生成机制决定了它只是在预测下一个最可能的词元，而没有内置的事实核查模块。最后，在面对需要复杂推理的任务时，模型可能会在逻辑链条中出错，从而“编造”出错误的结论。例如：一个旅游规划 Agent，可能会为你推荐一个现实中不存在的景点，或者预订一个航班号错误的机票。

此外，大语言模型还面临着知识时效性不足和训练数据中存在的偏见等挑战。大语言模型的能力来源于其训练数据。这意味着模型所掌握的知识是其训练数据收集时的最新材料。对于在此日期之后发生的事件、新出现的概念或最新的事实，模型将无法感知或正确回答。与此同时训练数据往往包含了人类社会的各种偏见和刻板印象。当模型在这些数据上学习时，它不可避免地会吸收并反映出这些偏见^[12]。

为了提高大语言模型的可靠性，研究人员和开发者正在积极探索多种检测和缓解幻觉的方法：

1. 数据层面：通过高质量数据清洗、引入事实性知识以及强化学习与人类反馈 (RLHF) 等方式^[13]，从源头减少幻觉。
2. 模型层面：探索新的模型架构，或让模型能够表达其对生成内容的不确定性。
3. 推理与生成层面：
 1. 检索增强生成 (**Retrieval-Augmented Generation, RAG**)^[14]：这是目前缓解幻觉的有效方法之一。RAG 系统通过在生成之前从外部知识库（如文档数据库、网页）中检索相关信息，然后将检索到的信息作为上下文，引导模型生成基于事实的回答。
 2. 多步推理与验证：引导模型进行多步推理，并在每一步进行自我检查或外部验证。
 3. 引入外部工具：允许模型调用外部工具（如搜索引擎、计算器、代码解释器）来获取实时信息或进行精确计算。

尽管幻觉问题短期内难以完全消除，但通过上述的策略，可以显著降低其发生频率和影响，提高大语言模型在实际应用中的可靠性和实用性。

3.4 本章小结

本章介绍了构建智能体所需的基础知识，重点围绕作为其核心组件的大语言模型 (LLM) 展开。内容从语言模型的早期发展开始，详细讲解了 Transformer 架构，并介绍了与 LLM 进行交互的方法。最后，本章对当前主流的模型生态、发展规律及其固有局限性进行了梳理。

核心知识点回顾：

- **模型演进与核心架构**：本章追溯了从统计语言模型 (N-gram) 到神经网络模型 (RNN, LSTM)，再到奠定现代 LLM 基础的 Transformer 架构。通过“自顶向下”的代码实现，本章拆解了 Transformer 的核心组件，并阐述了自注意力机制在并行计算和捕捉长距离依赖中的关键作用。
- **与模型的交互方式**：本章介绍了与 LLM 交互的两个核心环节：提示工程 (Prompt Engineering) 和文本分词 (Tokenization)。前者用于指导模型的行为，后者是理解模型输入处理的基础。通过本地部署并运行开源模型的实践，将理论知识应用于实际操作。
- **模型生态与选型**：本章系统地梳理了为智能体选择模型时需要权衡的关键因素，并概览了以 OpenAI GPT、Google Gemini 为代表的闭源模型和以 Llama、Mistral 为代表的开源模型的特点与定位。
- **法则与局限**：本章探讨了驱动 LLM 能力提升的缩放法则，阐述了其背后的基本原理。同时，本章也分析了模型存在的如事实幻觉、知识过时等固有局限性，这对于构建可靠、鲁棒的智能体至关重要。

从 LLM 基础到构建智能体：

这一章的 LLM 基础主要是为了帮助大家更好的理解大模型的诞生以及发展过程，其中也蕴含了智能体设计的部分思考。例如，如何设计有效的提示词来引导 Agent 的规划与决策，如何根据任务需求选择合适的模型，以及如何在 Agent 的工作流中加入验证机制以规避模型的幻觉等问题，其解决方案均建立在本章的基础之上。我们现在已经准备好从理论转向实践。在下一章，我们将开始探索智能体经典范式构建，将本章所学的知识应用于实际的智能体设计之中。

习题

1. 自然语言处理中，语言模型经历了从统计到神经网络的模型演进。

- 请使用本章提供的迷你语料库 (`datawhale agent learns, datawhale agent works`)，计算句子 `agent works` 在 Bigram 模型下的概率
- N-gram 模型的核心假设是马尔可夫假设。请解释这个假设的含义，以及 N-gram 模型存在哪些根本性局限？
- 神经网络语言模型 (RNN/LSTM) 和 Transformer 分别是如何克服 N-gram 模型局限的？它们各自的优势是什么？

2. Transformer 架构^[4]是现代大语言模型的基础。其中：

提示：可以结合本章 3.1.2 节的代码实现来辅助理解

- 自注意力机制 (Self-Attention) 的核心思想是什么？
- 为什么 Transformer 能够并行处理序列，而 RNN 必须串行处理？位置编码 (Positional Encoding) 在其中起什么作用？
- Decoder-Only 架构与完整的 Encoder-Decoder 架构有什么区别？为什么现在主流的大语言模型都采用 Decoder-Only 架构？

3. 文本子词分词算法是大语言模型的一项关键技术，负责将文本转换为模型可处理的 token 序列。那为什么不能直接以”字符”或”单词”作为模型的输入单元？BPE（Byte Pair Encoding）算法解决了什么问题？

4. 本章3.2.3节介绍了如何本地部署开源大语言模型。请完成以下实践和分析：

提示：这是一道动手实践题，建议实际操作

- 按照本章的指导，在本地部署一个轻量级的开源模型（推荐[Qwen3-0.6B](#)），并尝试调整采样参数并观察其对输出的影响
- 选择一个具体任务（如文本分类、信息抽取、代码生成等），设计并对比以下不同的提示策略（如Zero-shot、Few-shot、Chain-of-Thought）对输出结果的效果差异
- 从性能、成本、可控性、隐私等维度比较闭源模型和开源模型
- 如果你要构建一个企业级的客服智能体，你会选择哪种类型的模型？需要考虑哪些因素？

5. 模型幻觉（Hallucination）^[11]是大语言模型当前存在的关键局限性之一。本章介绍了缓解幻觉的方法（如检索增强生成、多步推理、外部工具调用）

- 请选择其中一种，说明其工作原理和适用场景
- 调研前沿的研究和论文，是否还有其他的缓解模型幻觉的方法，他们又有哪些改进和优势？

6. 假设你要设计一个论文辅助阅读智能体，它能够帮助研究人员快速阅读并理解学术论文，包括：总结论文研究的核心内容、回答关于论文的问题、提取关键信息、比较多篇不同论文的观点等。请回答：

- 你会选择哪个模型作为智能体设计时的基座模型？选择时需要考虑哪些因素？
- 如何设计提示词来引导模型更好地理解学术论文？学术论文通常很长，可能超过模型的上下文窗口限制，你会如何解决这个问题？
- 学术研究是严谨的，这意味着我们需要确保智能体生成的信息是准确客观忠于原文的。你认为系统中加入哪些设计能够更好的实现这一需求？

参考文献

- [1] Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3, 1137-1155.
- [2] Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2), 179-211.
- [3] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735-1780.
- [4] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998-6008).
- [5] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. OpenAI.
- [6] Gage, P. (1994). A new algorithm for data compression. *C Users Journal*, 12(2), 23-38.
- [7] Schuster, M., & Nakajima, K. (2012, March). Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 5149-5152). IEEE.
- [8] Kudo, T., & Richardson, J. (2018). SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.
- [9] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*.
- [10] Hoffmann, J., Borgeaud, E., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, R., ... & Sifre, L. (2022). Training Compute-Optimal Large Language Models. *arXiv preprint arXiv:2203.07678*.
- [11] Ji, Z., Lee, N., Fries, R., Yu, T., & Su, D. (2023). Survey of Hallucination in Large Language Models.
- [12] Bender, E. M., Gebru, T., McMillan-Major, A., & Mitchell, M. (2021). On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? .
- [13] Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences. *arXiv preprint arXiv:1706.03741*.
- [14] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goswami, N., ... & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in neural information processing systems* (pp. 9459-9474).

第四章 智能体经典范式构建

在上一章中，我们深入探讨了作为现代智能体“大脑”的大语言模型。我们了解了其内部的Transformer架构、与之交互的方法，以及它的能力边界。现在，是时候将这些理论知识转化为实践，亲手构建智能体了。

一个现代的智能体，其核心能力在于能将大语言模型的推理能力与外部世界联通。它能够自主地理解用户意图、拆解复杂任务，并通过调用代码解释器、搜索引擎、API等一系列“工具”，来获取信息、执行操作，最终达成目标。然而，智能体并非万能，它同样面临着来自大模型本身的“幻觉”问题、在复杂任务中可能陷入推理循环、以及对工具的错误使用等挑战，这些也构成了智能体的能力边界。

为了更好地组织智能体的“思考”与“行动”过程，业界涌现出了多种经典的架构范式。在本章中，我们将聚焦于其中最具代表性的三种，并一步步从零实现它们：

- **ReAct (Reasoning and Acting)**: 一种将“思考”和“行动”紧密结合的范式，让智能体边想边做，动态调整。
- **Plan-and-Solve**: 一种“三思而后行”的范式，智能体首先生成一个完整的行动计划，然后严格执行。
- **Reflection**: 一种赋予智能体“反思”能力的范式，通过自我批判和修正来优化结果。

了解了这些之后，你可能会问，市面上已有LangChain、LlamaIndex等众多优秀框架，为何还要“重复造轮子”？答案在于，尽管成熟的框架在工程效率上优势显著，但直接使用高度抽象的工具，并不利于我们了解背后的设计机制是怎么运行的，或者是有何好处。其次，这个过程会暴露出项目的工程挑战。框架为我们处理了许多问题，例如模型输出格式的解析、工具调用失败的重试、防止智能体陷入死循环等。亲手处理这些问题，是培养系统设计能力的最直接方式。最后，也是最重要的一点，掌握了设计原理，你才能真正地从一个框架的“使用者”转变为一个智能体应用的“创造者”。当标准组件无法满足你的复杂需求时，你将拥有深度定制乃至从零构建一个全新智能体的能力。

4.1 环境准备与基础工具定义

在开始构建之前，我们需要先搭建好开发环境并定义一些基础组件。这能帮助我们在后续实现不同范式时，避免重复劳动，更专注于核心逻辑。

4.1.1 安装依赖库

本书的实战部分将主要使用 Python 语言，建议使用 Python 3.10 或更高版本。首先，请确保你已经安装了 `openai` 库用于与大语言模型交互，以及 `python-dotenv` 库用于安全地管理我们的 API 密钥。

在你的终端中运行以下命令：

```
pip install openai python-dotenv
```

4.1.2 配置 API 密钥

为了让我们的代码更通用，我们将模型服务的相关信息（模型ID、API密钥、服务地址）统一配置在环境变量中。

1. 在你的项目根目录下，创建一个名为 `.env` 的文件。
2. 在该文件中，添加以下内容。你可以根据自己的需要，将其指向 OpenAI 官方服务，或任何兼容 OpenAI 接口的本地/第三方服务。
3. 如果实在不知道如何获取，可以参考 [环境配置](#)

```
# .env file
LLM_API_KEY="YOUR-API-KEY"
LLM_MODEL_ID="YOUR-MODEL"
LLM_BASE_URL="YOUR-URL"
```

我们的代码将从此文件自动加载这些配置。

4.1.3 封装基础 LLM 调用函数

为了让代码结构更清晰、更易于复用，我们来定义一个专属的LLM客户端类。这个类将封装所有与模型服务交互的细节，让我们的主逻辑可以更专注于智能体的构建。

```
import os
from openai import OpenAI
from dotenv import load_dotenv
from typing import List, Dict

# 加载 .env 文件中的环境变量
load_dotenv()

class HelloAgentsLLM:
    """
    为本书 "Hello Agents" 定制的LLM客户端。
    它用于调用任何兼容OpenAI接口的服务，并默认使用流式响应。
    """
    def __init__(self, model: str = None, apiKey: str = None, baseUrl: str = None, timeout: int = None):
        """
        初始化客户端。优先使用传入参数，如果未提供，则从环境变量加载。
        """
        self.model = model or os.getenv("LLM_MODEL_ID")
        apiKey = apiKey or os.getenv("LLM_API_KEY")
        baseUrl = baseUrl or os.getenv("LLM_BASE_URL")
```

```

timeout = timeout or int(os.getenv("LLM_TIMEOUT", 60))

if not all([self.model, apiKey, baseUrl]):
    raise ValueError("模型ID、API密钥和服务地址必须被提供或在.env文件中定义。")

self.client = OpenAI(api_key=apiKey, base_url=baseUrl, timeout=timeout)

def think(self, messages: List[Dict[str, str]], temperature: float = 0) -> str:
    """
    调用大语言模型进行思考，并返回其响应。
    """

    print(f"🧠 正在调用 {self.model} 模型....")
    try:
        response = self.client.chat.completions.create(
            model=self.model,
            messages=messages,
            temperature=temperature,
            stream=True,
        )

        # 处理流式响应
        print("✅ 大语言模型响应成功:")
        collected_content = []
        for chunk in response:
            content = chunk.choices[0].delta.content or ""
            print(content, end="", flush=True)
            collected_content.append(content)
        print() # 在流式输出结束后换行
        return "".join(collected_content)

    except Exception as e:
        print(f"❌ 调用LLM API时发生错误: {e}")
        return None

# --- 客户端使用示例 ---
if __name__ == '__main__':
    try:
        llmClient = HelloAgentsLLM()

        exampleMessages = [
            {"role": "system", "content": "You are a helpful assistant that writes Python code."},
            {"role": "user", "content": "写一个快速排序算法"}
        ]

        print("--- 调用LLM ---")
        responseText = llmClient.think(exampleMessages)
        if responseText:
            print("\n\n--- 完整模型响应 ---")
            print(responseText)

    except ValueError as e:
        print(e)

>>>
--- 调用LLM ---
🧠 正在调用 xxxxxx 模型...
✅ 大语言模型响应成功:
快速排序是一种非常高效的排序算法...

```

4.2 ReAct

在准备好LLM客户端后，我们将构建第一个，也是最经典的一个智能体范式**ReAct (Reason + Act)**。ReAct由Shunyu Yao于2022年提出^[1]，其核心思想是模仿人类解决问题的方式，将推理(**Reasoning**)与行动(**Acting**)显式地结合起来，形成一个“思考-行动-观察”的循环。

4.2.1 ReAct 的工作流程

在ReAct诞生之前，主流的方法可以分为两类：一类是“纯思考”型，如思维链(**Chain-of-Thought**)，它能引导模型进行复杂的逻辑推理，但无法与外部世界交互，容易产生事实幻觉；另一类是“纯行动”型，模型直接输出要执行的动作，但缺乏规划和纠错能力。

ReAct的巧妙之处在于，它认识到思考与行动是相辅相成的。思考指导行动，而行动的结果又反过来修正思考。为此，ReAct范式通过一种特殊的提示工程来引导模型，使其每一步的输出都遵循一个固定的轨迹：

- **Thought (思考)**: 这是智能体的“内心独白”。它会分析当前情况、分解任务、制定下一步计划，或者反思上一步的结果。
- **Action (行动)**: 这是智能体决定采取的具体动作，通常是调用一个外部工具，例如Search['华为最新款手机']。
- **Observation (观察)**: 这是执行Action后从外部工具返回的结果，例如搜索结果的摘要或API的返回值。

智能体将不断重复这个 **Thought -> Action -> Observation** 的循环，将新的观察结果追加到历史记录中，形成一个不断增长的上下文，直到它在Thought中认为已经找到了最终答案，然后输出结果。这个过程形成了一个强大的协同效应：推理使得行动更具目的性，而行动则为推理提供了事实依据。

我们可以将这个过程形式化地表达出来，如图4.1所示。具体来说，在每个时间步 t ，智能体的策略（即大语言模型 π ）会根据初始问题 q 和之前所有步骤的“行动-观察”历史轨迹 $((a_1, o_1), \dots, (a_{t-1}, o_{t-1}))$ ，来生成当前的思考 th_t 和行动 a_t ：

$$(th_t, a_t) = \pi(q, (a_1, o_1), \dots, (a_{t-1}, o_{t-1}))$$

随后，环境中的工具 T 会执行行动 a_t ，并返回一个新的观察结果 o_t ：

$$o_t = T(a_t)$$

这个循环不断进行，将新的 (a_t, o_t) 对追加到历史中，直到模型在思考 th_t 中判断任务已完成。

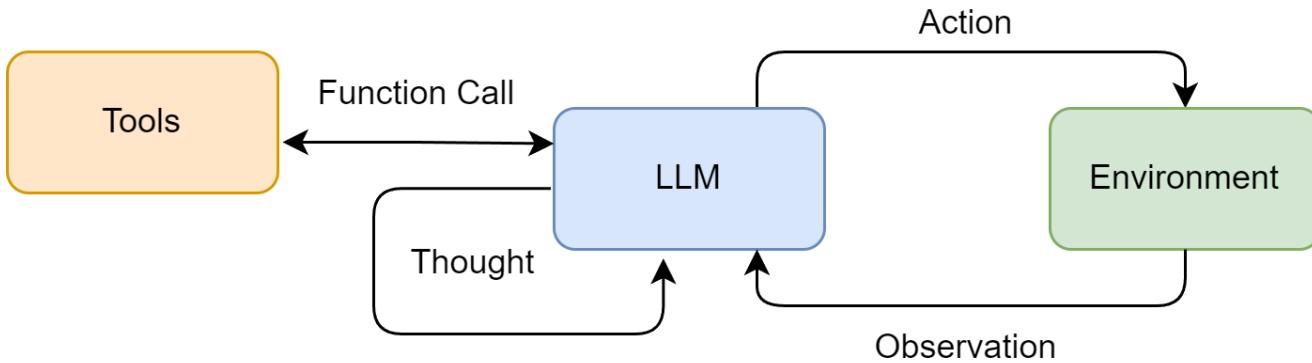


图 4.1 ReAct 范式中的“思考-行动-观察”协同循环

这种机制特别适用于以下场景：

- 需要外部知识的任务：如查询实时信息（天气、新闻、股价）、搜索专业领域的知识等。
- 需要精确计算的任务：将数学问题交给计算器工具，避免LLM的计算错误。
- 需要与API交互的任务：如操作数据库、调用某个服务的API来完成特定功能。

因此我们将构建一个具备使用外部工具能力的ReAct智能体，来回答一个大语言模型仅凭自身知识库无法直接回答的问题。例如：“华为最新的手机是哪一款？它的主要卖点是什么？”这个问题需要智能体理解自己需要上网搜索，调用工具搜索结果并总结答案。

4.2.2 工具的定义与实现

如果说大语言模型是智能体的大脑，那么工具 (Tools) 就是其与外部世界交互的“手和脚”。为了让ReAct范式能够真正解决我们设定的问题，智能体需要具备调用外部工具的能力。

针对本节设定的目标——回答关于“华为最新手机”的问题，我们需要为智能体提供一个网页搜索工具。在这里我们选用 **SerpApi**，它通过API提供结构化的Google搜索结果，能直接返回“答案摘要框”或精确的知识图谱信息，

首先，需要安装该库：

```
pip install google-search-results
```

同时，你需要前往 [SerpApi官网](#) 注册一个免费账户，获取你的API密钥，并将其添加到我们项目根目录下的 .env 文件中：

```
# .env file
# ... (保留之前的LLM配置)
SERPAPI_API_KEY="YOUR_SERPAPI_API_KEY"
```

接下来，我们通过代码来定义和管理这个工具。我们将分步进行：首先实现工具的核心功能，然后构建一个通用的工具管理器。

(1) 实现搜索工具的核心逻辑

一个良好定义的工具应包含以下三个核心要素：

1. 名称 (Name)：一个简洁、唯一的标识符，供智能体在 Action 中调用，例如 Search。
2. 描述 (Description)：一段清晰的自然语言描述，说明这个工具的用途。这是整个机制中最关键的部分，因为大语言模型会依赖这段描述来判断何时使用哪个工具。
3. 执行逻辑 (Execution Logic)：真正执行任务的函数或方法。

我们的第一个工具是 search 函数，它的作用是接收一个查询字符串，然后返回搜索结果。

```
from serpapi import SerpApiClient

def search(query: str) -> str:
    """
    一个基于SerpApi的实战网页搜索引擎工具。
    它会智能地解析搜索结果，优先返回直接答案或知识图谱信息。
    """

    print(f"正在执行 [SerpApi] 网页搜索: {query}")

    try:
        api_key = os.getenv("SERPAPI_API_KEY")
        if not api_key:
            return "错误:SERPAPI_API_KEY 未在 .env 文件中配置。"
    except:
        pass
```

```

params = {
    "engine": "google",
    "q": query,
    "api_key": api_key,
    "gl": "cn", # 国家代码
    "hl": "zh-cn", # 语言代码
}

client = SerpApiClient(params)
results = client.get_dict()

# 智能解析:优先寻找最直接的答案
if "answer_box_list" in results:
    return "\n".join(results["answer_box_list"])
if "answer_box" in results and "answer" in results["answer_box"]:
    return results["answer_box"]["answer"]
if "knowledge_graph" in results and "description" in results["knowledge_graph"]:
    return results["knowledge_graph"]["description"]
if "organic_results" in results and results["organic_results"]:
    # 如果没有直接答案, 则返回前三个有机结果的摘要
    snippets = [
        f"\n{i+1}. {res.get('title', '')}\n{res.get('snippet', '')}"
        for i, res in enumerate(results["organic_results"][:3])
    ]
    return "\n\n".join(snippets)

return f"对不起, 没有找到关于 '{query}' 的信息。"

except Exception as e:
    return f"搜索时发生错误: {e}"

```

在上述代码中, 首先会检查是否存在 `answer_box` (Google的答案摘要框) 或 `knowledge_graph` (知识图谱) 等信息, 如果存在, 就直接返回这些最精确的答案。如果不存在, 它才会退而求其次, 返回前三个常规搜索结果的摘要。这种“智能解析”能为LLM提供质量更高的信息输入。

(2) 构建通用的工具执行器

当智能体需要使用多种工具时 (例如, 除了搜索, 还可能需要计算、查询数据库等), 我们需要一个统一的管理器来注册和调度这些工具。为此, 我们创建一个 `ToolExecutor` 类。

```

from typing import Dict, Any

class ToolExecutor:
    """
    一个工具执行器, 负责管理和执行工具。
    """

    def __init__(self):
        self.tools: Dict[str, Dict[str, Any]] = {}

    def registerTool(self, name: str, description: str, func: callable):
        """
        向工具箱中注册一个新工具。
        """

        if name in self.tools:
            print(f"警告: 工具 '{name}' 已存在, 将被覆盖。")
        self.tools[name] = {"description": description, "func": func}
        print(f"工具 '{name}' 已注册。")

    def getTool(self, name: str) -> callable:
        """
        根据名称获取一个工具的执行函数。
        """

        return self.tools.get(name, {}).get("func")

    def getAvailableTools(self) -> str:
        """
        获取所有可用工具的格式化描述字符串。
        """

        return "\n".join([
            f"- {name}: {info['description']}"
            for name, info in self.tools.items()
        ])

```

(3) 测试

现在, 我们将 `search` 工具注册到 `ToolExecutor` 中, 并模拟一次调用, 以验证整个流程是否正常工作。

```

# --- 工具初始化与使用示例 ---
if __name__ == '__main__':
    # 1. 初始化工具执行器
    toolExecutor = ToolExecutor()

    # 2. 注册我们的实战搜索工具
    search_description = "一个网页搜索引擎。当你需要回答关于时事、事实以及在你的知识库中找不到的信息时, 应使用此工具。"
    toolExecutor.registerTool("Search", search_description, search)

    # 3. 打印可用的工具
    print("\n--- 可用的工具 ---")
    print(toolExecutor.getAvailableTools())

    # 4. 智能体的Action调用, 这次我们问一个实时性的问题

```

```

print("\n--- 执行 Action: Search['英伟达最新的GPU型号是什么'] ---")
tool_name = "Search"
tool_input = "英伟达最新的GPU型号是什么"

tool_function = toolExecutor.getTool(tool_name)
if tool_function:
    observation = tool_function(tool_input)
    print("--- 观察 (Observation) ---")
    print(observation)
else:
    print(f"错误:未找到名为 '{tool_name}' 的工具。")

>>>
工具 'Search' 已注册。

--- 可用的工具 ---
- Search: 一个网页搜索引擎。当你需要回答关于时事、事实以及在你的知识库中找不到的信息时，应使用此工具。

--- 执行 Action: Search['英伟达最新的GPU型号是什么'] ---
🔍 正在执行 [SerpApi] 网页搜索: 英伟达最新的GPU型号是什么
--- 观察 (Observation) ---
[1] GeForce RTX 50 系列显卡
GeForce RTX™ 50 系列GPU 搭载NVIDIA Blackwell 架构，为游戏玩家和创作者带来全新玩法。RTX 50 系列具备强大的AI 算力，带来升级体验和更逼真的画面。

[2] 比较GeForce 系列最新一代显卡和前代显卡
比较最新一代RTX 30 系列显卡和前代的RTX 20 系列、GTX 10 和900 系列显卡。查看规格、功能、技术支持等内容。

[3] GeForce 显卡 | NVIDIA DRIVE AGX
强大的车载计算能力，适用于AI 驱动的智能汽车系统 · Clara AGX。适用于创新型医疗设备和成像的AI 计算。游戏和创作。GeForce. 探索显卡、游戏解决方案、AI ...

至此，我们已经为智能体配备了连接真实世界互联网的Search工具，为后续的ReAct循环提供了坚实的基础。

```

4.2.3 ReAct 智能体的编码实现

现在，我们将所有独立的组件，LLM客户端和工具执行器组装起来，构建一个完整的 ReAct 智能体。我们将通过一个 `ReActAgent` 类来封装其核心逻辑。为了便于理解，我们将这个类的实现过程拆分为以下几个关键部分进行讲解。

(1) 系统提示词设计

提示词是整个 ReAct 机制的基石，它为大语言模型提供了行动的操作指令。我们需要精心设计一个模板，它将动态地插入可用工具、用户问题以及中间步骤的交互历史。

```

# ReAct 提示词模板
REACT_PROMPT_TEMPLATE = """
请注意，你是一个有能力调用外部工具的智能助手。

```

可用工具如下：
`{tools}`

请严格按照以下格式进行回应：

```

Thought: 你的思考过程，用于分析问题、拆解任务和规划下一步行动。
Action: 你决定采取的行动，必须是以下格式之一：
- `{{tool_name}}[{{tool_input}}]`：调用一个可用工具。
- `Finish[最终答案]`：当你认为已经获得最终答案时。
- 当你收集到足够的信息，能够回答用户的最终问题时，你必须在Action:字段后使用 Finish[最终答案] 来输出最终答案。

```

```

现在，请开始解决以下问题：
Question: {question}
History: {history}
.....

```

这个模板定义了智能体与LLM之间交互的规范：

- 角色定义：“你是一个有能力调用外部工具的智能助手”，设定了LLM的角色。
- 工具清单 (`{tools}`)：告知LLM它有哪些可用的“手脚”。
- 格式规约 (`Thought/Action`)：这是最重要的部分，它强制LLM的输出具有结构性，使我们能通过代码精确解析其意图。
- 动态上下文 (`{question}/{history}`)：将用户的原始问题和不断累积的交互历史注入，让LLM基于完整的上下文进行决策。

(2) 核心循环的实现

`ReActAgent` 的核心是一个循环，它不断地“格式化提示词 -> 调用LLM -> 执行动作 -> 整合结果”，直到任务完成或达到最大步数限制。

```

class ReActAgent:
    def __init__(self, llm_client: HelloAgentsLLM, tool_executor: ToolExecutor, max_steps: int = 5):
        self.llm_client = llm_client
        self.tool_executor = tool_executor
        self.max_steps = max_steps
        self.history = []

    def run(self, question: str):
        .....
        运行ReAct智能体来回答一个问题。
        .....

```

```

self.history = [] # 每次运行时重置历史记录
current_step = 0

while current_step < self.max_steps:
    current_step += 1
    print(f"--- 第 {current_step} 步 ---")

    # 1. 格式化提示词
    tools_desc = self.tool_executor.getAvailableTools()
    history_str = "\n".join(self.history)
    prompt = REACT_PROMPT_TEMPLATE.format(
        tools=tools_desc,
        question=question,
        history=history_str
    )

    # 2. 调用LLM进行思考
    messages = [{"role": "user", "content": prompt}]
    response_text = self.llm_client.think(messages=messages)

    if not response_text:
        print("错误:LLM未能返回有效响应。")
        break

    # ... (后续的解析、执行、整合步骤)

```

run 方法是智能体的入口。它的 while 循环构成了 ReAct 范式的主体，max_steps 参数则是一个重要的安全阀，防止智能体陷入无限循环而耗尽资源。

(3) 输出解析器的实现

LLM 返回的是纯文本，我们需要从中精确地提取出Thought和Action。这是通过几个辅助解析函数完成的，它们通常使用正则表达式来实现。

```

# (这些方法是 ReActAgent 类的一部分)
def _parse_output(self, text: str):
    """解析LLM的输出，提取Thought和Action。
    """
    # Thought: 匹配到 Action: 或文本末尾
    thought_match = re.search(r"Thought:\s*(.*?)(?=Action:|$)", text, re.DOTALL)
    # Action: 匹配到文本末尾
    action_match = re.search(r"Action:\s*(.*?)$", text, re.DOTALL)
    thought = thought_match.group(1).strip() if thought_match else None
    action = action_match.group(1).strip() if action_match else None
    return thought, action

def _parse_action(self, action_text: str):
    """解析Action字符串，提取工具名称和输入。
    """
    match = re.match(r"(\w+)\[(.*)\]", action_text, re.DOTALL)
    if match:
        return match.group(1), match.group(2)
    return None, None

```

- _parse_output: 负责从LLM的完整响应中分离出Thought和Action两个主要部分。

- _parse_action: 负责进一步解析Action字符串，例如从 Search[华为最新手机] 中提取出工具名 Search 和工具输入 华为最新手机。

4. 工具调用与执行

```

# (这段逻辑在 run 方法的 while 循环内)
# 3. 解析LLM的输出
thought, action = self._parse_output(response_text)

if thought:
    print(f"思考: {thought}")

if not action:
    print("警告:未能解析出有效的Action, 流程终止。")
    break

# 4. 执行Action
if action.startswith("Finish"):
    # 如果是Finish指令，提取最终答案并结束
    final_answer = re.match("Finish\[(.*)\]", action).group(1)
    print(f"💡 最终答案: {final_answer}")
    return final_answer

tool_name, tool_input = self._parse_action(action)
if not tool_name or not tool_input:
    # ... 处理无效Action格式 ...
    continue

print(f"💡 行动: {tool_name}[{tool_input}]")

tool_function = self.tool_executor.getTool(tool_name)
if not tool_function:
    observation = f"错误:未找到名为 '{tool_name}' 的工具。"
else:
    observation = tool_function(tool_input) # 调用真实工具

```

这段代码是Action的执行中心。它首先检查是否为Finish指令，如果是，则流程结束。否则，它会通过tool_executor获取对应的工具函数并执行，得到observation。

(5) 观测结果的整合

最后一步，也是形成闭环的关键，是将Action本身和工具执行后的Observation添加回历史记录中，为下一轮循环提供新的上下文。

```
# (这段逻辑紧随工具调用之后，在 while 循环的末尾)
print(f"👀 观察: {observation}")

# 将本轮的Action和Observation添加到历史记录中
self.history.append(f"Action: {action}")
self.history.append(f"Observation: {observation}")

# 循环结束
print("已达到最大步数，流程终止。")
return None
```

通过将Observation追加到self.history，智能体在下一轮生成提示词时，就能“看到”上一步行动的结果，并据此进行新一轮的思考和规划。

(6) 运行实例与分析

将以上所有部分组合起来，我们就得到了完整的ReActAgent类。完整的代码运行实例可以在本书配套的代码仓库code文件夹中找到。

下面是一次真实的运行记录：

```
工具 'Search' 已注册。

--- 第 1 步 ---
🧠 正在调用 xxxxxx 模型...
✅ 大语言模型响应成功:
Thought: 要回答这个问题，我需要查找华为最新发布的手机型号及其主要特点。这些信息可能在我的现有知识库之外，因此需要使用搜索引擎来获取最新数据。
Action: Search[华为最新手机型号及主要卖点]
🤔 思考：要回答这个问题，我需要查找华为最新发布的手机型号及其主要特点。这些信息可能在我的现有知识库之外，因此需要使用搜索引擎来获取最新数据。
📝 行动：Search[华为最新手机型号及主要卖点]
🔍 正在执行 [SerpApi] 网页搜索：华为最新手机型号及主要卖点
👀 观察：[1] 华为手机 - 华为官网
智能手机；Mate 系列。非凡旗舰 · HUAWEI Mate XTs。非凡大师；Pura 系列。先锋影像 · HUAWEI Pura 80 Pro+；Pocket 系列。美学新篇。HUAWEI Pocket 2；nova
[2] 2025年华为手机哪一款性价比高？华为手机推荐与市场分析 ...
现在华为手机最大的卖点只剩下鸿蒙HarmonyOS系统，以及饱受争议的品牌信仰。这里推荐目前值得入手的几款华为系列手机，根据不同预算自行选择：. 华为目前最受欢迎，也是搭载 ...

[3] 2025年华为新款手机哪个性价比高？10款华为新款手机推荐
选华为主要还是要推荐高端手机，Mate 70和Pura 70系列是最新发布的旗舰机型。HUAWEI Mate 70. 优点是，拍照配置依旧顶级，全焦段覆盖，适合专业摄影，做工出色，户外抗摔 ...
--- 第 2 步 ---
🧠 正在调用 xxxxxx 模型...
✅ 大语言模型响应成功:
Thought: 根据搜索结果，华为最新发布的旗舰机型包括Mate 70和Pura 80 Pro+。为了确定最新型号及其主要卖点，我将重点放在这些信息上。从提供的链接来看，Mate 70系列和Pura 80 Pro+。
Action: Finish[根据最新信息，华为的最新手机可能是HUAWEI Pura 80 Pro+或HUAWEI Mate 70。其中，HUAWEI Mate 70的主要卖点包括顶级的拍照配置，全焦段覆盖，适合专业摄影]
🤔 思考：根据搜索结果，华为最新发布的旗舰机型包括Mate 70和Pura 80 Pro+。为了确定最新型号及其主要卖点，我将重点放在这些信息上。从提供的链接来看，Mate 70系列和Pura 80 Pro+。
最終答案：根据最新信息，华为的最新手机可能是HUAWEI Pura 80 Pro+或HUAWEI Mate 70。其中，HUAWEI Mate 70的主要卖点包括顶级的拍照配置，全焦段覆盖，适合专业摄影，
```

从上面的输出可以看到，智能体清晰地展示了它的思考链条：它首先意识到自己的知识不足，需要使用搜索工具；然后，它根据搜索结果进行推理和总结，并在两步之内得出了最终答案。

值得注意的是，由于模型的知识和互联网的信息是不断更新的，你运行的结果可能与此不完全相同。截止本节内容编写的2025年9月8日，搜索结果中提到的HUAWEI Mate 70与HUAWEI Pura 80 Pro+确实是华为当时最新的旗舰系列手机。这充分展示了ReAct范式在处理时效性问题上的强大能力。

4.2.4 ReAct 的特点、局限性与调试技巧

通过亲手实现一个ReAct智能体，我们不仅掌握了其工作流程，也应该对其内在机制有了更深刻的认识。任何技术范式都有其闪光点和待改进之处，本节将对ReAct进行总结。

(1) ReAct的主要特点

1. 高可解释性：ReAct最大的优点之一就是透明。通过Thought链，我们可以清晰地看到智能体每一步的“心路历程”——它为什么会选择这个工具，下一步又打算做什么。这对于理解、信任和调试智能体的行为至关重要。
2. 动态规划与纠错能力：与一次性生成完整计划的范式不同，ReAct是“走一步，看一步”。它根据每一步从外部世界获得的Observation来动态调整后续的Thought和Action。如果上一步的搜索结果不理想，它可以在下一步中修正搜索词，重新尝试。
3. 工具协同能力：ReAct范式天然地将大语言模型的推理能力与外部工具的执行能力结合起来。LLM负责运筹帷幄（规划和推理），工具负责解决具体问题（搜索、计算），二者协同工作，突破了单一LLM在知识时效性、计算准确性等方面的固有局限。

(2) ReAct的固有局限性

- 对LLM自身能力的强依赖：ReAct 流程的成功与否，高度依赖于底层 LLM 的综合能力。如果 LLM 的逻辑推理能力、指令遵循能力或格式化输出能力不足，就很容易在 Thought 环节产生错误的规划，或者在 Action 环节生成不符合格式的指令，导致整个流程中断。
- 执行效率问题：由于其循序渐进的特性，完成一个任务通常需要多次调用 LLM。每一次调用都伴随着网络延迟和计算成本。对于需要很多步骤的复杂任务，这种串行的“思考-行动”循环可能会导致较高的总耗时和费用。
- 提示词的脆弱性：整个机制的稳定运行建立在一个精心设计的提示词模板之上。模板中的任何微小变动，甚至是用词的差异，都可能影响 LLM 的行为。此外，并非所有模型都能持续稳定地遵循预设的格式，这增加了在实际应用中的不确定性。
- 可能陷入局部最优：步进式的决策模式意味着智能体缺乏一个全局的、长远的规划。它可能会因为眼前的 Observation 而选择一个看似正确但长远来看并非最优的路径，甚至在某些情况下陷入“原地打转”的循环中。

(3) 调试技巧

当你构建的 ReAct 智能体行为不符合预期时，可以从以下几个方面入手进行调试：

- 检查完整的提示词：在每次调用 LLM 之前，将最终格式化好的、包含所有历史记录的完整提示词打印出来。这是追溯 LLM 决策源头的最直接方式。
- 分析原始输出：当输出解析失败时（例如，正则表达式没有匹配到 Action），务必将 LLM 返回的原始、未经处理的文本打印出来。这能帮助你判断是 LLM 没有遵循格式，还是你的解析逻辑有误。
- 验证工具的输入与输出：检查智能体生成的 tool_input 是否是工具函数所期望的格式，同时也要确保工具返回的 observation 格式是智能体可以理解和处理的。
- 调整提示词中的示例 (**Few-shot Prompting**)：如果模型频繁出错，可以在提示词中加入一两个完整的“Thought-Action-Observation”成功案例，通过示例来引导模型更好地遵循你的指令。
- 尝试不同的模型或参数：更换一个能力更强的模型，或者调整 temperature 参数（通常设为0以保证输出的确定性），有时能直接解决问题。

4.3 Plan-and-Solve

在我们掌握了 ReAct 这种反应式的、步进决策的智能体范式后，接下来将探讨一种风格迥异但同样强大的方法，**Plan-and-Solve**。顾名思义，这种范式将任务处理明确地分为两个阶段：先规划 (Plan)，后执行 (Solve)。

如果说 ReAct 像一个经验丰富的侦探，根据现场的蛛丝马迹 (Observation) 一步步推理，随时调整自己的调查方向；那么 Plan-and-Solve 则更像一位建筑师，在动工之前必须先绘制出完整的蓝图 (Plan)，然后严格按照蓝图来施工 (Solve)。事实上我们现在用的很多大模型工具的Agent模式都融入了这种设计模式。

4.3.1 Plan-and-Solve 的工作原理

Plan-and-Solve Prompting 由 Lei Wang 在2023年提出^[2]。其核心动机是为了解决思维链在处理多步骤、复杂问题时容易“偏离轨道”的问题。

与 ReAct 将思考和行动融合在每一步不同，Plan-and-Solve 将整个流程解耦为两个核心阶段，如图4.2所示：

- 规划阶段 (**Planning Phase**)：首先，智能体会接收用户的完整问题。它的第一个任务不是直接去解决问题或调用工具，而是将问题分解，并制定出一个清晰、分步骤的行动计划。这个计划本身就是一次大语言模型的调用产物。
- 执行阶段 (**Solving Phase**)：在获得完整的计划后，智能体进入执行阶段。它会严格按照计划中的步骤，逐一执行。每一步的执行都可能是一次独立的 LLM 调用，或者是对上一步结果的加工处理，直到计划中的所有步骤都完成，最终得出答案。

这种“先谋后动”的策略，使得智能体在处理需要长远规划的复杂任务时，能够保持更高的目标一致性，避免在中间步骤中迷失方向。

我们可以将这个两阶段过程进行形式化表达。首先，规划模型 π_{plan} 根据原始问题 q 生成一个包含 n 个步骤的计划 $P = (p_1, p_2, \dots, p_n)$ ：

$$P = \pi_{\text{plan}}(q)$$

随后，在执行阶段，执行模型 π_{solve} 会逐一完成计划中的步骤。对于第 i 个步骤，其解决方案 s_i 的生成会同时依赖于原始问题 q 、完整计划 P 以及之前所有步骤的执行结果 (s_1, \dots, s_{i-1}) ：

$$s_i = \pi_{\text{solve}}(q, P, (s_1, \dots, s_{i-1}))$$

最终的答案就是最后一个步骤的执行结果 s_n 。

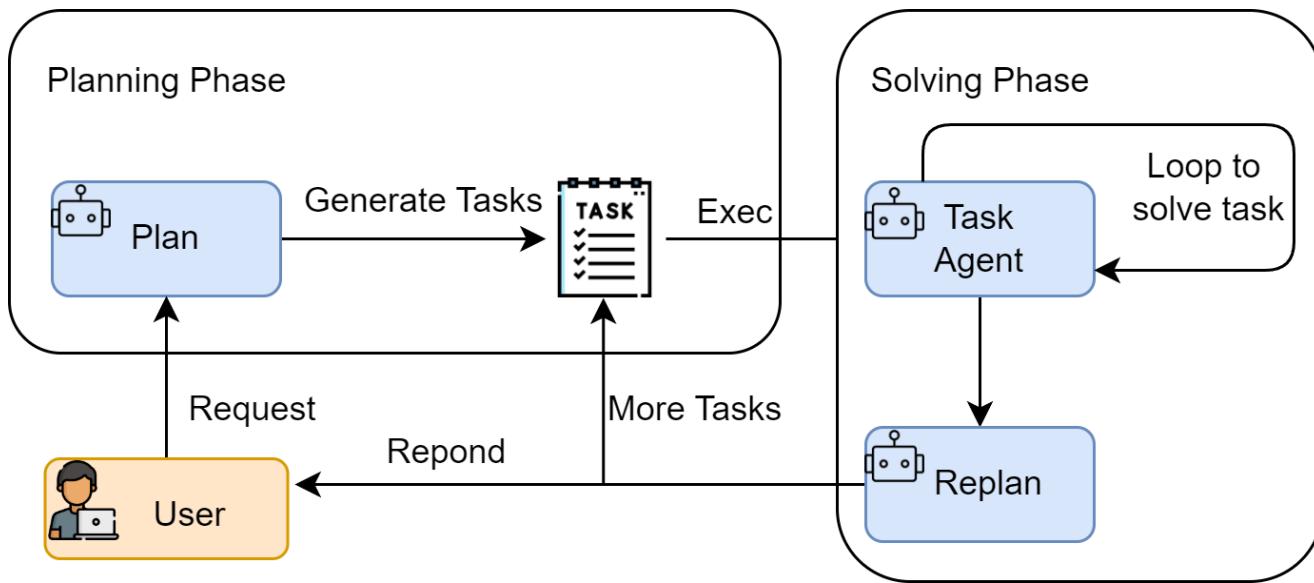


图 4.2 Plan-and-Solve 范式的两阶段工作流

Plan-and-Solve 尤其适用于那些结构性强、可以被清晰分解的复杂任务，例如：

- 多步数学应用题：需要先列出计算步骤，再逐一求解。
- 需要整合多个信息源的报告撰写：需要先规划好报告结构（引言、数据来源A、数据来源B、总结），再逐一填充内容。
- 代码生成任务：需要先构思好函数、类和模块的结构，再逐一实现。

4.3.2 规划阶段

为了凸显 Plan-and-Solve 范式在结构化推理任务上的优势，我们将不使用工具的方式，而是通过提示词的设计，完成一个推理任务。

这类任务的特点是，答案无法通过单次查询或计算得出，必须先将问题分解为一系列逻辑连贯的子步骤，然后按顺序求解。这恰好能发挥 Plan-and-Solve “先规划，后执行”的核心能力。

我们的目标问题是：“一个水果店周一卖出了15个苹果。周二卖出的苹果数量是周一的两倍。周三卖出的数量比周二少了5个。请问这三天总共卖出了多少个苹果？”

这个问题对于大语言模型来说并不算特别困难，但它包含了一个清晰的逻辑链条可供参考。在某些实际的逻辑难题上，如果大模型不能高质量的推理出准确的答案，可以参考这个设计模式来设计自己的Agent完成任务。智能体需要：

1. 规划阶段：首先，将问题分解为三个独立的计算步骤（计算周二销量、计算周三销量、计算总销量）。
2. 执行阶段：然后，严格按照计划，一步步执行计算，并将每一步的结果作为下一步的输入，最终得出总和。

规划阶段的目标是让大语言模型接收原始问题，并输出一个清晰、分步骤的行动计划。这个计划必须是结构化的，以便我们的代码可以轻松解析并逐一执行。因此，我们设计的提示词需要明确地告诉模型它的角色和任务，并给出一个输出格式的范例。

```
PLANNER_PROMPT_TEMPLATE = """
你是一个顶级的AI规划专家。你的任务是将用户提出的复杂问题分解成一个由多个简单步骤组成的行动计划。
请确保计划中的每个步骤都是一个独立的、可执行的子任务，并且严格按照逻辑顺序排列。
你的输出必须是一个Python列表，其中每个元素都是一个描述子任务的字符串。
"""

问题: {question}
```

请严格按照以下格式输出你的计划，```python与```作为前后缀是必要的：
```python  
["步骤1", "步骤2", "步骤3", ...]  
```

这个提示词通过以下几点确保了输出的质量和稳定性：- 角色设定：“顶级的AI规划专家”，激发模型的专业能力。- 任务描述：清晰地定义了“分解问题”的目标。- 格式约束：强制要求输出为一个 Python 列表格式的字符串，这极大地简化了后续代码的解析工作，使其比解析自然语言更稳定、更可靠。

接下来，我们将这个提示词逻辑封装成一个 Planner 类，这个类也是我们的规划器。

```
# 假定 llm_client.py 中的 HelloAgentsLLM 类已经定义好
# from llm_client import HelloAgentsLLM
```

```
class Planner:
    def __init__(self, llm_client):
        self.llm_client = llm_client

    def plan(self, question: str) -> list[str]:
        """根据用户问题生成一个行动计划。
```

```

.....
prompt = PLANNER_PROMPT_TEMPLATE.format(question=question)

# 为了生成计划，我们构建一个简单的消息列表
messages = [{"role": "user", "content": prompt}]

print("--- 正在生成计划 ---")
# 使用流式输出来获取完整的计划
response_text = self.llm_client.think(messages=messages) or ""

print(f"✅ 计划已生成:\n{response_text}")

# 解析LLM输出的列表字符串
try:
    # 找到```python和```之间的内容
    plan_str = response_text.split("```python")[1].split("```")[-1].strip()
    # 使用ast.literal_eval来安全地执行字符串，将其转换为Python列表
    plan = ast.literal_eval(plan_str)
    return plan if isinstance(plan, list) else []
except (ValueError, SyntaxError, IndexError) as e:
    print(f"❌ 解析计划时出错: {e}")
    print(f"原始响应: {response_text}")
    return []
except Exception as e:
    print(f"❌ 解析计划时发生未知错误: {e}")
    return []

```

4.3.3 执行器与状态管理

在规划器(Planner)生成了清晰的行动蓝图后，我们就需要一个执行器(Executor)来逐一完成计划中的任务。执行器不仅负责调用大语言模型来解决每个子问题，还承担着一个至关重要的角色：状态管理。它必须记录每一步的执行结果，并将其作为上下文提供给后续步骤，确保信息在整个任务链条中顺畅流动。

执行器的提示词与规划器不同。它的目标不是分解问题，而是在已有上下文的基础上，专注解决当前这一个步骤。因此，提示词需要包含以下关键信息：

- 原始问题：确保模型始终了解最终目标。
- 完整计划：让模型了解当前步骤在整个任务中的位置。
- 历史步骤与结果：提供至今为止已经完成的工作，作为当前步骤的直接输入。
- 当前步骤：明确指示模型现在需要解决哪一个具体任务。

EXECUTOR_PROMPT_TEMPLATE = """

你是一位顶级的AI执行专家。你的任务是严格按照给定的计划，一步步地解决问题。
你将收到原始问题、完整的计划、以及到目前为止已经完成的步骤和结果。
请你专注于解决“当前步骤”，并仅输出该步骤的最终答案，不要输出任何额外的解释或对话。

```

# 原始问题:
{question}

# 完整计划:
{plan}

# 历史步骤与结果:
{history}

# 当前步骤:
{current_step}

请仅输出针对“当前步骤”的回答:
"""

```

我们将执行逻辑封装到 Executor 类中。这个类将循环遍历计划，调用 LLM，并维护一个历史记录（状态）。

```

class Executor:
    def __init__(self, llm_client):
        self.llm_client = llm_client

    def execute(self, question: str, plan: list[str]) -> str:
        """
        根据计划，逐步执行并解决问题。
        """

        history = "" # 用于存储历史步骤和结果的字符串

        print("\n--- 正在执行计划 ---")

        for i, step in enumerate(plan):
            print(f"\n-> 正在执行步骤 {i+1}/{len(plan)}: {step}")

            prompt = EXECUTOR_PROMPT_TEMPLATE.format(
                question=question,
                plan=plan,
                history=history if history else "无", # 如果是第一步，则历史为空
                current_step=step
            )

            messages = [{"role": "user", "content": prompt}]
            response_text = self.llm_client.think(messages=messages) or ""

```

```
# 更新历史记录, 为下一步做准备
history += f"步骤 {i+1}: {step}\n结果: {response_text}\n\n"

print(f"✓ 步骤 {i+1} 已完成, 结果: {response_text}")

# 循环结束后, 最后一步的响应就是最终答案
final_answer = response_text
return final_answer
```

现在已经分别构建了负责“规划”的 Planner 和负责“执行”的 Executor。最后一步是将这两个组件整合到一个统一的智能体 PlanAndSolveAgent 中，并赋予它解决问题的完整能力。我们将创建一个主类 PlanAndSolveAgent，它的职责非常清晰：接收一个 LLM 客户端、初始化内部的规划器和执行器，并提供一个简单的 run 方法来启动整个流程。

```
class PlanAndSolveAgent:
    def __init__(self, llm_client):
        ...
        初始化智能体, 同时创建规划器和执行器实例。
        ...
        self.llm_client = llm_client
        self.planner = Planner(self.llm_client)
        self.executor = Executor(self.llm_client)

    def run(self, question: str):
        ...
        运行智能体的完整流程:先规划, 后执行。
        ...
        print(f"\n--- 开始处理问题 ---\n问题: {question}")

        # 1. 调用规划器生成计划
        plan = self.planner.plan(question)

        # 检查计划是否成功生成
        if not plan:
            print("\n--- 任务终止 --- \n无法生成有效的行动计划。")
            return

        # 2. 调用执行器执行计划
        final_answer = self.executor.execute(question, plan)

        print(f"\n--- 任务完成 ---\n最终答案: {final_answer}")
```

这个 PlanAndSolveAgent 类的设计体现了“组合优于继承”的原则。它本身不包含复杂的逻辑，而是作为一个协调者 (Orchestrator)，清晰地调用其内部组件来完成任务。

4.3.4 运行实例与分析

完整的代码同样参考本书配套的代码仓库 code 文件夹，这里只演示最终结果。

```
--- 开始处理问题 ---
问题: 一个水果店周一卖出了15个苹果。周二卖出的苹果数量是周一的两倍。周三卖出的数量比周二少了5个。请问这三天总共卖出了多少个苹果?
--- 正在生成计划 ---
🧠 正在调用 xxxx 模型...
✓ 大语言模型响应成功:
```python
["计算周一卖出的苹果数量: 15个", "计算周二卖出的苹果数量: 周一数量 × 2 = 15 × 2 = 30个", "计算周三卖出的苹果数量: 周二数量 - 5 = 30 - 5 = 25个", "计算三天总
```
✓ 计划已生成:
```python
["计算周一卖出的苹果数量: 15个", "计算周二卖出的苹果数量: 周一数量 × 2 = 15 × 2 = 30个", "计算周三卖出的苹果数量: 周二数量 - 5 = 30 - 5 = 25个", "计算三天总
```
--- 正在执行计划 ---
-> 正在执行步骤 1/4: 计算周一卖出的苹果数量: 15个
🧠 正在调用 xxxx 模型...
✓ 大语言模型响应成功:
15
✓ 步骤 1 已完成, 结果: 15

-> 正在执行步骤 2/4: 计算周二卖出的苹果数量: 周一数量 × 2 = 15 × 2 = 30个
🧠 正在调用 xxxx 模型...
✓ 大语言模型响应成功:
30
✓ 步骤 2 已完成, 结果: 30

-> 正在执行步骤 3/4: 计算周三卖出的苹果数量: 周二数量 - 5 = 30 - 5 = 25个
🧠 正在调用 xxxx 模型...
✓ 大语言模型响应成功:
25
✓ 步骤 3 已完成, 结果: 25

-> 正在执行步骤 4/4: 计算三天总销量: 周一 + 周二 + 周三 = 15 + 30 + 25 = 70个
🧠 正在调用 xxxx 模型...
```

大语言模型响应成功:

70

步骤 4 已完成, 结果: 70

--- 任务完成 ---

最终答案: 70

从上面的输出日志中，我们可以清晰地看到 Plan-and-Solve 范式的工作流程：

1. 规划阶段：智能体首先调用 Planner，成功地将复杂的应用题分解成了一个包含四个逻辑步骤的 Python 列表。这个结构化的计划为后续的执行奠定了基础。
2. 执行阶段：Executor 严格按照生成的计划，一步一步地向下执行。在每一步中，它都将历史结果作为上下文，确保了信息的正确传递（例如，步骤2正确地使用了步骤1的结果“15个”，步骤3也正确使用了步骤2的结果“30个”）。
3. 结果：整个过程逻辑清晰，步骤明确，最终智能体准确地得出了正确答案“70个”。

4.4 Reflection

在我们已经实现的 ReAct 和 Plan-and-Solve 范式中，智能体一旦完成了任务，其工作流程便告结束。然而，它们生成的初始答案，无论是行动轨迹还是最终结果，都可能存在谬误或有待改进之处。Reflection 机制的核心思想，正是为智能体引入一种事后（post-hoc）的自我校正循环，使其能够像人类一样，审视自己的工作，发现不足，并进行迭代优化。

4.4.1 Reflection 机制的核心思想

Reflection 机制的灵感来源于人类的学习过程：我们完成初稿后会进行校对，解出数学题后会进行验算。这一思想在多个研究中得到了体现，例如 Shinn, Noah 在2023年提出的 Reflexion 框架^[3]。其核心工作流程可以概括为一个简洁的三步循环：执行 -> 反思 -> 优化。

1. 执行 (Execution): 首先，智能体使用我们熟悉的方法（如 ReAct 或 Plan-and-Solve）尝试完成任务，生成一个初步的解决方案或行动轨迹。这可以看作是“初稿”。
2. 反思 (Reflection): 接着，智能体进入反思阶段。它会调用一个独立的、或者带有特殊提示词的大语言模型实例，来扮演一个“评审员”的角色。这个“评审员”会审视第一步生成的“初稿”，并从多个维度进行评估，例如：
 - 事实性错误：是否存在与常识或已知事实相悖的内容？
 - 逻辑漏洞：推理过程是否存在不连贯或矛盾之处？
 - 效率问题：是否有更直接、更简洁的路径来完成任务？
 - 遗漏信息：是否忽略了问题的某些关键约束或方面？根据评估，它会生成一段结构化的反馈 (Feedback)，指出具体的问题所在和改进建议。
3. 优化 (Refinement): 最后，智能体将“初稿”和“反馈”作为新的上下文，再次调用大语言模型，要求它根据反馈内容对初稿进行修正，生成一个更完善的“修订稿”。

如图4.3所示，这个循环可以重复进行多次，直到反思阶段不再发现新的问题，或者达到预设的迭代次数上限。我们可以将这个迭代优化的过程形式化地表达出来。假设 O_i 是第 i 次迭代产生的输出 (O_0 为初始输出)，反思模型 π_{reflect} 会生成针对 O_i 的反馈 F_i : $F_i = \pi_{\text{reflect}}(\text{Task}, O_i)$ 。随后，优化模型 π_{refine} 会结合原始任务、上一版输出以及反馈，生成新一版的输出 O_{i+1} : $O_{i+1} = \pi_{\text{refine}}(\text{Task}, O_i, F_i)$

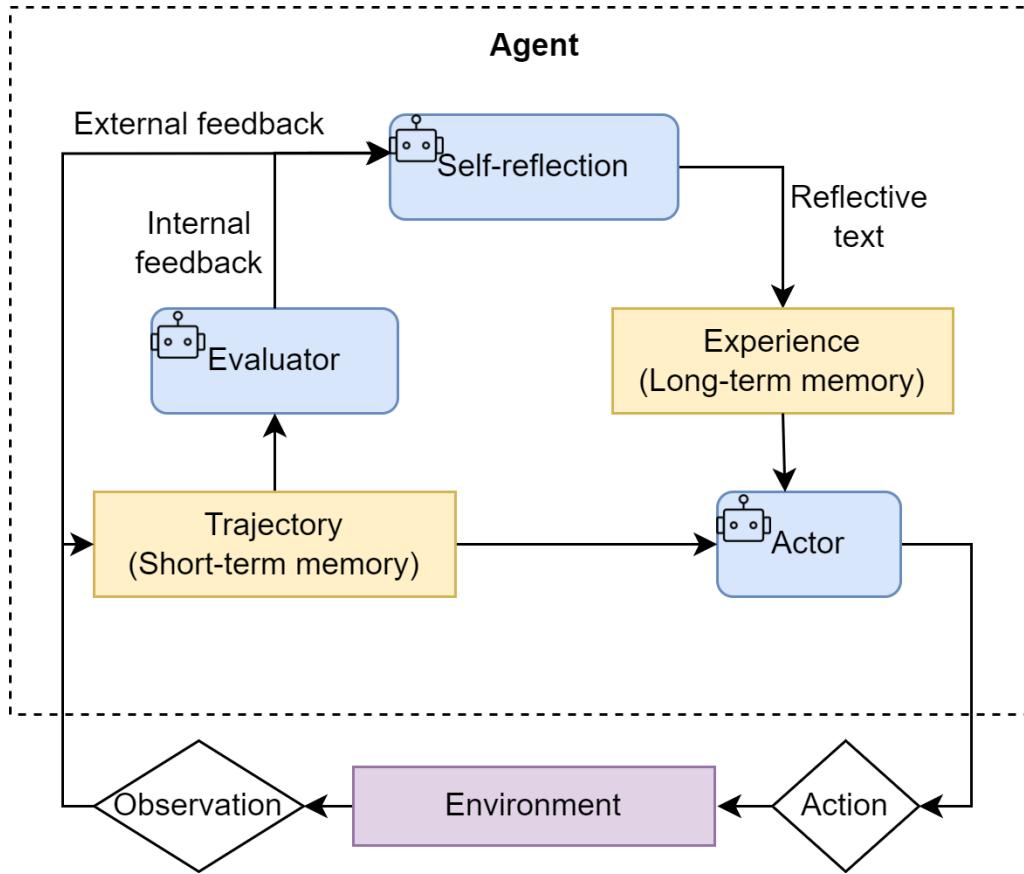


图 4.3 Reflection 机制中的“执行-反思-优化”迭代循环

与前两种范式相比，Reflection 的价值在于：

- 它为智能体提供了一个内部纠错回路，使其不再完全依赖于外部工具的反馈（ReAct 的 Observation），从而能够修正更高层次的逻辑和策略错误。
- 它将一次性的任务执行，转变为一个持续优化的过程，显著提升了复杂任务的最终成功率和答案质量。
- 它为智能体构建了一个临时的“短期记忆”。整个“执行-反思-优化”的轨迹形成了一个宝贵的经验记录，智能体不仅知道最终答案，还记得自己是如何从有缺陷的初稿迭代到最终版本的。更进一步，这个记忆系统还可以是多模态的，允许智能体反思和修正文本以外的输出（如代码、图像等），为构建更强大的多模态智能体奠定了基础。

4.4.2 案例设定与记忆模块设计

为了在实战中体现 Reflection 机制，我们将引入记忆管理机制，因为 reflection 通常对应着信息的存储和提取，如果上下文足够长的情况，想让“评审员”直接获取所有的信息然后进行反思往往传入很多冗余信息。这一步实践我们主要完成代码生成与迭代优化。

这一步的目标任务是：“编写一个 Python 函数，找出 1 到 n 之间所有的素数 (prime numbers)。”

这个任务是检验 Reflection 机制的绝佳场景：

1. 存在明确的优化路径：大语言模型初次生成的代码很可能是一个简单但效率低下的递归实现。
2. 反思点清晰：可以通过反思发现其“时间复杂度过高”或“存在重复计算”的问题。
3. 优化方向明确：可以根据反馈，将其优化为更高效的迭代版本或使用备忘录模式的版本。

Reflection 的核心在于迭代，而迭代的前提是能够记住之前的尝试和获得的反馈。因此，一个“短期记忆”模块是实现该范式的必需品。这个记忆模块将负责存储每一次“执行-反思”循环的完整轨迹。

```

from typing import List, Dict, Any, Optional

class Memory:
    """
    一个简单的短期记忆模块，用于存储智能体的行动与反思轨迹。
    """

    def __init__(self):
        """
        初始化一个空列表来存储所有记录。
        """
        self.records: List[Dict[str, Any]] = []

    def add_record(self, record_type: str, content: str):
        """
        向记忆中添加一条新记录。
        """
  
```

```

参数:
- record_type (str): 记录的类型 ('execution' 或 'reflection')。
- content (str): 记录的具体内容 (例如, 生成的代码或反思的反馈)。
"""
record = {"type": record_type, "content": content}
self.records.append(record)
print(f"💡 记忆已更新, 新增一条 '{record_type}' 记录。")

def get_trajectory(self) -> str:
"""
将所有记忆记录格式化为一个连贯的字符串文本, 用于构建提示词。
"""
trajectory_parts = []
for record in self.records:
    if record['type'] == 'execution':
        trajectory_parts.append(f"--- 上一轮尝试 (代码) ---\n{record['content']}")
    elif record['type'] == 'reflection':
        trajectory_parts.append(f"--- 评审员反馈 ---\n{record['content']}")

return "\n\n".join(trajectory_parts)

def get_last_execution(self) -> Optional[str]:
"""
获取最近一次的执行结果 (例如, 最新生成的代码)。
如果不存在, 则返回 None。
"""
for record in reversed(self.records):
    if record['type'] == 'execution':
        return record['content']
return None

```

这个 Memory 类的设计比较简洁, 主体是这样的:

- 使用一个列表 records 来按顺序存储每一次的行动和反思。
- add_record 方法负责向记忆中添加新的条目。
- get_trajectory 方法是核心, 它将记忆轨迹“序列化”成一段文本, 可以直接插入到后续的提示词中, 为模型的反思和优化提供完整的上下文。
- get_last_execution 方便我们获取最新的“初稿”以供反思。

4.4.3 Reflection 智能体的编码实现

有了 Memory 模块作为基础, 我们现在可以着手构建 ReflectionAgent 的核心逻辑。整个智能体的工作流程将围绕我们之前讨论的“执行-反思-优化”循环展开, 并通过精心设计的提示词来引导大语言模型扮演不同的角色。

(1) 提示词设计

与之前的范式不同, Reflection 机制需要多个不同角色的提示词来协同工作。

1. 初始执行提示词 (Execution Prompt) : 这是智能体首次尝试解决问题的提示词, 内容相对直接, 只要求模型完成指定任务。

```
INITIAL_PROMPT_TEMPLATE = """
```

你是一位资深的Python程序员。请根据以下要求, 编写一个Python函数。
你的代码必须包含完整的函数签名、文档字符串, 并遵循PEP 8编码规范。

要求: {task}

请直接输出代码, 不要包含任何额外的解释。

```
"""
```

2. 反思提示词 (Reflection Prompt) : 这个提示词是 Reflection 机制的灵魂。它指示模型扮演“代码评审员”的角色, 对上一轮生成的代码进行批判性分析, 并提供具体的、可操作的反馈。

```
REFLECT_PROMPT_TEMPLATE = """
```

你是一位极其严格的代码评审专家和资深算法工程师, 对代码的性能有极致的要求。
你的任务是审查以下Python代码, 并专注于找出其在算法效率上的主要瓶颈。

```
# 原始任务:
{task}
```

```
# 待审查的代码:
```python
{code}
```
```

请分析该代码的时间复杂度, 并思考是否存在一种算法上更优的解决方案来显著提升性能。
如果存在, 请清晰地指出当前算法的不足, 并提出具体的、可行的改进算法建议 (例如, 使用筛法替代试除法)。
如果代码在算法层面已经达到最优, 才能回答“无需改进”。

请直接输出你的反馈, 不要包含任何额外的解释。

```
"""
```

3. 优化提示词 (Refinement Prompt) : 当收到反馈后, 这个提示词将引导模型根据反馈内容, 对原有代码进行修正和优化。

```
REFINE_PROMPT_TEMPLATE = """
```

你是一位资深的Python程序员。你正在根据一位代码评审专家的反馈来优化你的代码。

```
# 原始任务:
{task}

# 你上一轮尝试的代码:
{last_code_attempt}
评审员的反馈:
{feedback}
```

请根据评审员的反馈，生成一个优化后的新版本代码。
 你的代码必须包含完整的函数签名、文档字符串，并遵循PEP 8编码规范。
 请直接输出优化后的代码，不要包含任何额外的解释。

.....

(2) 智能体封装与实现

现在，我们将这套提示词逻辑和Memory模块整合到ReflectionAgent类中。

```
# 假设 llm_client.py 和 memory.py 已定义
# from llm_client import HelloAgentsLLM
# from memory import Memory

class ReflectionAgent:
    def __init__(self, llm_client, max_iterations=3):
        self.llm_client = llm_client
        self.memory = Memory()
        self.max_iterations = max_iterations

    def run(self, task: str):
        print(f"\n--- 开始处理任务 ---\n任务: {task}")

        # --- 1. 初始执行 ---
        print("\n--- 正在进行初始尝试 ---")
        initial_prompt = INITIAL_PROMPT_TEMPLATE.format(task=task)
        initial_code = self._get_llm_response(initial_prompt)
        self.memory.add_record("execution", initial_code)

        # --- 2. 迭代循环:反思与优化 ---
        for i in range(self.max_iterations):
            print(f"\n--- 第 {i+1}/{self.max_iterations} 轮迭代 ---")

            # a. 反思
            print("\n-> 正在进行反思...")
            last_code = self.memory.get_last_execution()
            reflect_prompt = REFLECT_PROMPT_TEMPLATE.format(task=task, code=last_code)
            feedback = self._get_llm_response(reflect_prompt)
            self.memory.add_record("reflection", feedback)

            # b. 检查是否需要停止
            if "无需改进" in feedback:
                print("\n✅ 反思认为代码已无需改进，任务完成。")
                break

            # c. 优化
            print("\n-> 正在进行优化...")
            refine_prompt = REFINE_PROMPT_TEMPLATE.format(
                task=task,
                last_code_attempt=last_code,
                feedback=feedback
            )
            refined_code = self._get_llm_response(refine_prompt)
            self.memory.add_record("execution", refined_code)

        final_code = self.memory.get_last_execution()
        print(f"\n--- 任务完成 ---\n最终生成的代码:\n```python\n{final_code}\n```")
        return final_code

    def _get_llm_response(self, prompt: str) -> str:
        """一个辅助方法，用于调用LLM并获取完整的流式响应。”"""
        messages = [{"role": "user", "content": prompt}]
        response_text = self.llm_client.think(messages=messages) or ""
        return response_text
```

4.4.4 运行实例与分析

完整的代码同样参考本书配套的代码仓库code文件夹，这里提供一个输出实例。

```
--- 开始处理任务 ---
任务: 编写一个Python函数，找出1到n之间所有的素数 (prime numbers)。
```

```
--- 正在进行初始尝试 ---
🧠 正在调用 xxxxxx 模型...
✅ 大语言模型响应成功:
```python
def find_primes(n):
 ...
 return primes
```
记忆已更新，新增一条 'execution' 记录。
```

--- 第 1/2 轮迭代 ---

-> 正在进行反思...

🧠 正在调用 xxxxxx 模型...

✓ 大语言模型响应成功:

当前代码的时间复杂度为 $O(n * \sqrt{n})$ 。虽然对于较小的n值，这种实现是可以接受的，但当n非常大时，性能会显著下降。主要瓶颈在于每个数都需要进行试除法检查，这导致了较高的时间复杂度。

建议使用埃拉托斯特尼筛法 (Sieve of Eratosthenes)，该算法的时间复杂度为 $O(n \log \log n)$ ，能够显著提高查找素数的效率。

改进后的代码如下:

```
```python
def find_primes(n):
 ...
 return primes
```

```

📝 记忆已更新，新增一条 'reflection' 记录。

-> 正在进行优化...

🧠 正在调用 xxxxxx 模型...

✓ 大语言模型响应成功:

```
```python
def find_primes(n):
 ...
 return primes
```

```

📝 记忆已更新，新增一条 'execution' 记录。

--- 第 2/2 轮迭代 ---

-> 正在进行反思...

🧠 正在调用 xxxxxx 模型...

✓ 大语言模型响应成功:

当前代码使用了Eratosthenes筛法，时间复杂度为 $O(n \log \log n)$ ，空间复杂度为 $O(n)$ 。此算法在寻找1到n之间的所有素数时已经非常高效，通常情况下无需进一步优化。但在某些特定场景下，这些改进对于大多数应用场景来说并不是必需的，因为标准的Eratosthenes筛法已经足够高效。因此，在一般情况下，**无需改进**。

然而，这些改进对于大多数应用场景来说并不是必需的，因为标准的Eratosthenes筛法已经足够高效。因此，在一般情况下，**无需改进**。

📝 记忆已更新，新增一条 'reflection' 记录。

✓ 反思认为代码已无需改进，任务完成。

--- 任务完成 ---

最终生成的代码:

```
```python
def find_primes(n):
 """
 Finds all prime numbers between 1 and n using the Sieve of Eratosthenes algorithm.

 :param n: The upper limit of the range to find prime numbers.
 :return: A list of all prime numbers between 1 and n.
 """
 if n < 2:
 return []
 is_prime = [True] * (n + 1)
 is_prime[0] = is_prime[1] = False
 p = 2
 while p * p <= n:
 if is_prime[p]:
 for i in range(p * p, n + 1, p):
 is_prime[i] = False
 p += 1
 primes = [num for num in range(2, n + 1) if is_prime[num]]
 return primes
```

```

这个运行实例展示了 Reflection 机制是如何驱动智能体进行深度优化的:

- 有效的“批判”是优化的前提:在第一轮反思中，由于我们使用了“极其严格”且“专注于算法效率”的提示词，智能体没有满足于功能正确的初版代码，而是精准地指出了其 $O(n * \sqrt{n})$ 的时间复杂度瓶颈，并提出了算法层面的改进建议——埃拉托斯特尼筛法。
- 迭代式改进:智能体在接收到明确的反馈后，于优化阶段成功地实现了更高效的筛法，将算法复杂度降至 $O(n \log \log n)$ ，完成了第一次有意义的自我迭代。
- 收敛与终止:在第二轮反思中，智能体面对已经高效的筛法，展现出了更深层次的知识。它不仅肯定了当前算法的效率，甚至还提及了分段筛法等更高级的优化方向，但最终做出了“在一般情况下无需改进”的正确判断。这个判断触发了我们的终止条件，使优化过程得以收敛。

这个案例充分证明，一个设计良好的 Reflection 机制，其价值不仅在于修复错误，更在于驱动解决方案在质量和效率上实现阶梯式的提升，这使其成为构建复杂、高质量智能体的关键技术之一。

4.4.5 Reflection 机制的成本收益分析

尽管 Reflection 机制在提升任务解决质量上表现出色，但这种能力的获得并非没有代价。在实际应用中，我们需要权衡其带来的收益与相应的成本。

(1) 主要成本

1. 模型调用开销增加:这是最直接的成本。每进行一轮迭代，至少需要额外调用两次大语言模型（一次用于反思，一次用于优化）。如果迭代多轮，API 调用成本和计算资源消耗将成倍增加。
2. 任务延迟显著提高:Reflection 是一个串行过程，每一轮的优化都必须等待上一轮的反思完成。这使得任务的总耗时显著延长，不适合对实时性要求高的场景。
3. 提示工程复杂度上升:如我们的案例所示，Reflection 的成功在很大程度上依赖于高质量、有针对性的提示词。为“执行”、“反思”、“优化”等不同阶段设计和调试有效的提示词，需要投入更多的开发精力。

(2) 核心收益

1. 解决方案质量的跃迁:最大的收益在于，它能将一个“合格”的初始方案，迭代优化成一个“优秀”的最终方案。这种从功能正确到性能高效、从逻辑粗糙到逻辑严谨的提升，在很多关键任务中是至关重要的。
2. 鲁棒性与可靠性增强:通过内部的自我纠错循环，智能体能够发现并修复初始方案中可能存在的逻辑漏洞、事实性错误或边界情况处理不当等问题，从而大大提高了最终结果的可靠性。

综上所述，Reflection 机制是一种典型的“以成本换质量”的策略。它非常适合那些对最终结果的质量、准确性和可靠性有极高要求，且对任务完成的实时性要求相对宽松的场景。例如：

- 生成关键的业务代码或技术报告。
- 在科学研究中进行复杂的逻辑推演。
- 需要深度分析和规划的决策支持系统。

反之，如果应用场景需要快速响应，或者一个“大致正确”的答案就已经足够，那么使用更轻量的 ReAct 或 Plan-and-Solve 范式可能会是更具性价比的选择。

4.5 本章小结

在本章中，以第三章掌握的大语言模型知识为基础，我们通过“亲手造轮子”的方式，从零开始编码实现了三种业界经典的智能体构建范式:ReAct、Plan-and-Solve 与 Reflection。我们不仅探索了它们的核心工作原理，还通过具体的实战案例，深入了解了各自的优势、局限与适用场景。

核心知识点回顾：

1. ReAct:我们构建了一个能与外部世界交互的 ReAct 智能体。通过“思考-行动-观察”的动态循环，它成功地利用搜索引擎回答了自身知识库无法覆盖的实时性问题。其核心优势在于环境适应性和动态纠错能力，使其成为处理探索性、需要外部工具输入的任务的首选。
2. Plan-and-Solve:我们实现了一个先规划后执行的 Plan-and-Solve 智能体，并利用它解决了需要多步推理的数学应用题。它将复杂的任务分解为清晰的步骤，然后逐一执行。其核心优势在于结构性和稳定性，特别适合处理逻辑路径确定、内部推理密集的任务。
3. Reflection (自我反思与迭代):我们构建了一个具备自我优化能力的 Reflection 智能体。通过引入“执行-反思-优化”的迭代循环，它成功地将一个效率较低的初始代码方案，优化为了一个算法上更优的高性能版本。其核心价值在于能显著提升解决方案的质量，适用于对结果的准确性和可靠性有极高要求的场景。

本章探讨的三种范式，代表了智能体解决问题的三种不同策略，如表4.1所示。在实际应用中，选择哪一种，取决于任务的核心需求：

表 4.1 不同 Agent Loop 的选择策略

| 当你的任务... | 优先选择 | 核心原因 |
|----------------------|----------------|---------------------------|
| 充满不确定性，需要与外部API或网页交互 | ReAct | 它能根据实时反馈动态调整路径。 |
| 逻辑路径清晰，侧重内部推理和步骤分解 | Plan-and-Solve | 它能提供稳定、结构化的执行流程。 |
| 对最终结果的质量和可靠性有极致要求 | Reflection | 它通过迭代优化，能将“合格”的答案提升至“优秀”。 |

至此，我们已经掌握了构建单个智能体的核心技术。为了过渡知识，以及对实际应用更加深入。下一节我们将会探索不同低代码平台的使用方式以及轻代码构建agent的方案。

习题

提示:部分习题没有标准答案，重点在于培养学习者对智能体范式设计的综合理解和实践能力。

1. 本章介绍了三种经典的智能体范式:ReAct、Plan-and-Solve 和 Reflection。请分析：

- 这三种范式在“思考”与“行动”的组织方式上有什么本质区别？
- 如果要设计一个“智能家居控制助手”（需要控制灯光、空调、窗帘等多个设备，并根据用户习惯自动调节），你会选择哪种范式作为基础架构？为什么？
- 是否可以将这三种范式进行组合使用？若可以，请尝试设计一个混合范式的智能体架构，并说明其适用场景。

2. 在4.2节的 ReAct 实现中，我们使用了正则表达式来解析大语言模型的输出（如 Thought 和 Action）。请思考：

- 当前的解析方法存在哪些潜在的脆弱性？在什么情况下可能会失败？
- 除了正则表达式，还有哪些更鲁棒的输出解析方案？
- 尝试修改本章的代码，使用一种更可靠的输出格式，并对比两种方案的优缺点

3. 工具调用是现代智能体的核心能力之一。基于4.2.2节的 `ToolExecutor` 设计, 请完成以下扩展实践:

提示: 这是一道动手实践题, 建议实际编写代码

- 为 `ReAct` 智能体添加一个“计算器”工具, 使其能够处理复杂的数学计算问题 (如“计算 $(123 + 456) \times 789 / 12 = ?$ 的结果”)
- 设计并实现一个“工具选择失败”的处理机制: 当智能体多次调用错误的工具或提供错误的参数时, 系统应该如何引导它纠正?
- 思考: 如果可调用工具的数量增加到50个甚至100个, 当前的工具描述方式是否还能有效工作? 在可调用工具数量随业务需求显著增加时, 从工程角度如何优化工具的组织和检索机制?

4. `Plan-and-Solve` 范式将任务分解为“规划”和“执行”两个阶段。请深入分析:

- 在4.3节的实现中, 规划阶段生成的计划是“静态”的 (一次性生成, 不可修改)。如果在执行过程中发现某个步骤无法完成或结果不符合预期, 应该如何设计一个“动态重规划”机制?
- 对比 `Plan-and-Solve` 与 `ReAct`: 在处理“预订一次从北京到上海的商务旅行 (包括机票、酒店、租车)”这样的任务时, 哪种范式更合适? 为什么?
- 尝试设计一个“分层规划”系统: 先生成高层次的抽象计划, 然后针对每个高层步骤再生成详细的子计划。这种设计有什么优势?

5. `Reflection` 机制通过“执行-反思-优化”循环来提升输出质量。请思考:

- 在4.4节的代码生成案例中, 不同阶段使用的是同一个模型。如果使用两个不同的模型 (例如, 用一个更强大的模型来做反思, 用一个更快的模型来做执行), 会带来什么影响?
- `Reflection` 机制的终止条件是“反馈中包含无需改进“或”达到最大迭代次数”。这种设计是否合理? 能否设计一个更智能的终止条件?
- 假设你要搭建一个“学术论文写作助手”, 它能够生成初稿并不断优化论文内容。请设计一个多维度的`Reflection`机制, 从段落逻辑性、方法创新性、语言表达、引用规范等多个角度进行反思和改进。

6. 提示词工程是影响智能体最终效果的关键技术。本章展示了多个精心设计的提示词模板。请分析:

- 对比4.2.3节的 `ReAct` 提示词和4.3.2节的 `Plan-and-Solve` 提示词, 它们显然存在结构设计上的明显不同, 这些差异是如何服务于各自范式的核心逻辑的?
- 在4.4.3节的 `Reflection` 提示词中, 我们使用了“你是一位极其严格的代码评审专家”这样的角色设定。尝试修改这个角色设定 (如改为“你是一位注重代码可读性的开源项目维护者”), 观察输出结果的变化, 并总结角色设定对智能体行为的影响。
- 在提示词中加入 `few-shot` 示例往往能显著提升模型对特定格式的遵循能力。请为本章的某个智能体尝试添加 `few-shot` 示例, 并对比其效果。

7. 某电商初创公司现在希望使用“客服智能体”来代替真人客服实现降本增效, 它需要具备以下功能:

- a. 理解用户的退款申请理由
- b. 查询用户的订单信息和物流状态
- c. 根据公司政策智能地判断是否应该批准退款
- d. 生成一封得体的回复邮件并发送至用户邮箱
- e. 如果判断决策存在一定争议 (自我置信度低于阈值), 能够进行自我反思并给出更审慎的建议

此时作为该产品的负责人:

- 你会选择本章的哪种范式 (或哪些范式的组合) 作为系统的核心架构?
- 这个系统需要哪些工具? 请列出至少3个工具及其功能描述。
- 如何设计提示词来确保智能体的决策既符合公司利益, 又能保持对用户的友好态度?
- 这个产品上线后可能面临哪些风险和挑战? 如何通过技术手段来降低这些风险?

参考文献

- [1] Yao S, Zhao J, Yu D, et al. React: Synergizing reasoning and acting in language models[C]//International Conference on Learning Representations (ICLR). 2023.
- [2] Wang L, Xu W, Lan Y, et al. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models[J]. arXiv preprint arXiv:2305.04091, 2023.
- [3] Shinn N, Cassano F, Gopinath A, et al. Reflexion: Language agents with verbal reinforcement learning[J]. Advances in Neural Information Processing Systems, 2023, 36: 8634-8652.

第五章 基于低代码平台的智能体搭建

在前一章中，通过编写 Python 代码，从零开始实现了 ReAct、Plan-and-Solve 和 Reflection 多种经典的智能体工作流。这个过程为我们打下了坚实的技术基础，让我们深刻理解了智能体内部的运作机理。然而，对于一个快速发展的领域而言，纯代码的开发模式并非总是最高效的选择，尤其是在需要快速验证想法、或者非专业开发者希望参与构建的场景中。

5.1 平台化构建的兴起

随着技术的成熟，我们看到越来越多的能力正在被“平台化”。正如网站的开发从手写 HTML/CSS/JS，演进到了可以使用 WordPress、Wix 等建站平台一样，智能体的构建也迎来了平台化的浪潮。本章将聚焦于如何利用图形化、模块化的低代码平台，来快速、直观地搭建、调试和部署智能体应用，将我们的重心从“实现细节”转向“业务逻辑”。

5.1.1 为何需要低代码平台

“重复造轮子”对于深入学习至关重要，但在追求工程效率和创新的实战中，我们往往需要站在巨人的肩膀上。尽管我们在第四章中封装了可复用的 `ReActAgent`、`PlanAndSolveAgent` 等类，但当业务逻辑变得复杂时，纯代码的维护成本和开发周期会急剧上升。低代码平台的出现，正是为了解决这些痛点。

其核心价值主要体现在以下几个方面：

- 降低技术门槛：低代码平台将复杂的技术细节（如 API 调用、状态管理、并发控制）封装成一个个易于理解的“节点”或“模块”。用户无需精通编程，只需通过拖拽、连接这些节点，就能构建出功能强大的工作流。这使得产品经理、设计师、业务专家等非技术人员也能参与到智能体的设计与创造中来，极大地拓宽了创新的边界。
- 提升开发效率：对于专业开发者而言，平台同样能带来巨大的效率提升。在项目初期，当需要快速验证一个想法或搭建一个原型 (Prototype) 时，使用低代码平台可以在数小时甚至数分钟内完成原本需要数天编码的工作。开发者可以将精力更多地投入到业务逻辑梳理和提示工程优化上，而非底层的工程实现。
- 提供更优的可视化与可观测性：相比于在终端中打印日志，图形化的平台天然提供了对智能体运行轨迹的端到端可视化。你可以清晰地看到数据在每一个节点之间如何流动，哪一个环节耗时最长，哪一个工具调用失败。这种直观的调试体验，是纯代码开发难以比拟的。
- 标准化与最佳实践沉淀：优秀的低代码平台通常会内置许多行业内的最佳实践。例如，它会提供预设的 ReAct 模板、优化的知识库检索引擎、标准化的工具接入规范等。这不仅避免了开发者“踩坑”，也使得团队协作更加顺畅，因为所有人都基于同一套标准和组件进行开发。

简而言之，低代码平台并非要取代代码，而是提供了一种更高层次的抽象。它让我们可以从繁琐的底层实现中解放出来，更专注于智能体“思考”与“行动”的逻辑本身，从而更快、更好地将创意变为现实。

5.1.2 低代码平台的选择

当前，智能体与 LLM 应用的低代码平台市场呈现出百花齐放的态势，每个平台都有其独特的定位和优势。选择哪个平台，往往取决于你的核心需求、技术背景以及项目的最终目标。在本章的后续内容中，我们将重点介绍并实操三个各具代表性的平台：Coze、Dify 和 n8n。在此之前，我们先对它们进行一个概要性的介绍。

Coze

- 核心定位：由字节跳动推出的 Coze^[1]，主打零代码/低代码的 Agent 的构建体验，让不具备编程背景的用户也能轻松创造。
- 特点分析：Coze 拥有极其友好的可视化界面，用户可以像搭建乐高积木一样，通过拖拽插件、配置知识库和设定工作流来创建智能体。其内置了极为丰富的插件库，并支持一键发布到抖音、飞书、微信公众号等多个主流平台，极大地简化了分发流程。
- 适用人群：AI 应用的入门用户、产品经理、运营人员，以及希望快速将创意变为可交互产品的个人创作者。

Dify

- 核心定位：Dify 是一个开源的、功能全面的 LLM 应用开发与运营平台^[2]，旨在为开发者提供从原型构建到生产部署的一站式解决方案。
- 特点分析：它融合了后端服务和模型运营的理念，支持 Agent 工作流、RAG Pipeline、数据标注与微调等多种能力。对于追求专业、稳定、可扩展的企业级应用而言，Dify 提供了坚实的基础。
- 适用人群：有一定技术背景的开发者、需要构建可扩展的企业级 AI 应用的团队。

n8n

- 核心定位：n8n 本质上是一个开源工作流自动化工具^[3]，而非纯粹的 LLM 平台。近年来，它积极集成了 AI 能力。
- 特点分析：n8n 的强项在于“连接”。它拥有数百个预置的节点，可以轻松地将各类 SaaS 服务、数据库、API 连接成复杂的自动化业务流程。你可以在这个流程中嵌入 LLM 节点，使其成为整个自动化链路中的一环。虽然在 LLM 功能的专一度上不如前两者，但其通用自动化能力是独一无二的。不过，其学习曲线也相对陡峭。
- 适用人群：需要将 AI 能力深度整合进现有业务流程、实现高度定制化自动化的开发者和企业。

在接下来的小节中，我们将逐一上手体验这些平台，通过实际操作来更直观地感受它们各自的魅力。

5.2 平台一：Coze

扣子（Coze）是一个超级酷的AI智能体制作工具！也是目前市面上应用最广泛的智能体平台。该平台以其直观的可视化界面和丰富的功能模块，让用户能够轻松创建各种类型的智能体应用，比如能陪你聊天的机器人、自动写故事的创作机，甚至直接帮你将故事变成电影MV！它的一大亮点在于其强大的生态集成能力。开发完成的智能体可以一键发布到微信、飞书、豆包等主流平台，实现跨平台的无缝部署。对于企业用户而言，Coze还提供了灵活的API接口，支持将智能体能力集成到现有的业务系统中，实现了“搭积木式”的AI应用构建。### 5.2.1 Coze 的功能模块（1）平台界面初览

整体布局介绍：最近扣子又更新了他的UI界面了，如图5.1所示。现在最左边的侧边栏是扣子平台主页的开发工作区，包括核心的项目开发、资源库、效果评测和空间配置。下面的区域是扣子开发的配套资料空间包括官方模板一键复制、扣子最大的优势丰富多样的插件商店、最大的智能体社区琳琅满目、api管理就是api测试用的、以及详细的教程文档和面向企业的通用管理。右边这一块有四个模块，最上面是扣子最新的更新公告告诉你扣子的最新进展方便你了解最新的工具和功能。接着下面是新手教程，点开就是新手教程文档啦，分分钟开始智能体搭建。其次是你的关注和智能体推荐，在这里你也可以关注喜欢的AI开发者，和收藏他们的智能体为自己所用。



图 5.1 扣子智能智能体平台整体示意图

(2) 核心功能介绍

首先我们点击左边侧栏的加号就可以看到创建智能体的入口了，这里目前有两类AI应用，一种是创建智能体，另一种叫应用。其中智能体又分为单智能体自主规划模式、单智能体对话流模式和多智能体模式。AI应用也分两种不仅能设计桌面网端的用户界面，还能轻松搭建小程序和H5端的界面，如图5.2所示。



图 5.2 扣子智能体创建入口

项目空间里是你的智能体仓库，这里放着你所有开发的智能体或复制的智能体/应用，也是在扣子进行智能体开发你最经常来到的地方，如图5.3所示。

图 5.3 扣子智能体项目空间

资源库是你开发扣子智能体的核心武器库，资源库就会存放你的工作流、知识库、卡片、提示词库等一系列开发智能体的工具。你能做出什么样的智能体，首先取决于模型的能力，但是最重要的还是要看你怎么给智能体搭配“出装和技能”。模型决定了智能体的下限，但是扣子资源库给了你智能体的能力的无穷上限，让你能够按照自己的想法，开发想象力和脑洞进行智能体的开发，如图5.4所示。

The screenshot shows a list of resources in the 'Resource Library' section. The resources are:

- review_task_history_1_755 (Workflow): 当用户需要辅助批改历史主观题作业的时候，调用此工作流.
- write_generate_1_077 (Workflow): 当用户想要生成课件/教案时，调用此工作流.
- touxiang (Chat Flow): 国庆头像.
- jasonAI (Knowledge Base): 非结构化 Jason个人知识库.
- FlashCards (Card): FlashCards.
- jason_blog (Text): 文本 jason的博客.
- one_picture_1_725 (Workflow): 得到构图信息，可以双重曝光时，调用此图像流，用户的上传的图片‘user_image’作为‘image…’.
- new_background_1_410 (Workflow): 当用户上传一张背景图和一张主体图时，调用此图像流.
- double_exposure_1_440 (Chat Flow): 对话流.

图 5.4 扣子智能体资源库

空间配置包含智能体、插件、工作流和发布渠道的一个统一的管理频道，以及模型管理就是你可以在这里看到你调用的各种大模型，如图5.5所示。

The screenshot shows a list of publishing channels in the 'Release Channel Management' section. The channels are:

- 飞书多维表格 (Official Default Channel): 发布到飞书多维表格字段捷径，轻松实现批量生成.
- 微信小程序 (Official Default Channel): 官方默认渠道.
- 抖音小程序 (Official Default Channel): 官方默认渠道.
- 扣子智能体商店 (Official Default Channel): 智能体.
- 微信订阅号 (Official Default Channel): 智能体 应用.
- 微信服务号 (Official Default Channel): 智能体 应用.
- 掘金 (Official Default Channel): 智能体.
- 微信客服 (Official Default Channel): 智能体 应用.
- 飞书 (Official Default Channel): 智能体 应用.

图 5.5 扣子智能体发布渠道

如果让我对扣子的智能体开发做一个简单的总结的话，我会把他比喻成一个游戏的各个组成部分，各部分配合组合出一个一个精彩的智能体像极了打“游戏”，每做完一个智能体都像是打完了一个boss并且收获满满，不管是“经验”还是“装备”。

- 工作流：关卡通关路线图
- 对话流：NPC 对话通关
- 插件：角色技能卡
- 知识库：游戏百科全书
- 卡片：快捷道具栏
- 提示词：角色的移动键
- 数据库：“云存档”
- 发布管理：关卡审核员
- 模型管理：游戏角色库或者叫捏脸系统
- 效果评测：闯关评分系统

5.2.2 构建“每日AI简报”助手

案例说明：本实践案例旨在深入剖析 Coze 平台的插件集成能力，指导读者从零开始构建一个功能强大的“每日AI简报”智能体。该智能体能够自动化地从多个信息源（包括36氪、虎嗅、it之家、infoq、GitHub、arXiv）抓取当日最新的AI领域头条新闻、学术论文及开源项目动态，并将其结构化、专业化地整合成一份生动、精炼的简报。

通过本案例，您将系统性地掌握以下核心技能：

- 多源信息聚合：利用 Coze 的插件生态，实现跨平台、跨类型的数据流无缝集成。
- 智能体行为定义：通过角色设定和提示词（Prompt）工程，精准控制智能体的任务执行与内容生成，确保输出符合预设的专业标准。
- 自动化工作流构建：学习如何将数据获取、内容处理与格式化输出等多个步骤串联成一个高效、自动化的工作流。

步骤一：添加并配置信息源插件

构建“每日AI简报”智能体的首要任务是为其接入丰富且权威的信息来源。在 Coze 平台中，这通过添加和配置相应的插件来实现。

1. 插件集成：在 Coze 的插件库中，搜索并添加所需的插件。例如，通过 RSS 插件订阅媒体平台的 RSS 源（如图 5.6 所示），通过 GitHub 插件追踪开源项目（如图 5.7 所示），以及通过 arXiv 插件获取最新的学术研究成果（如图 5.8 所示）。



图 5.6 媒体平台的 RSS 源插件



图 5.7 GitHub 插件



图 5.8 Arxiv 插件

2. 个性化配置: 对每一个插件进行精细化配置, 以确保其能精准地获取所需数据。例如, 在 RSS 插件中, 输入36氪、虎嗅等网站的特定RSS订阅链接; 在 GitHub 插件中, 设置需监控的关键词查询数量以及最新更新设置; 在 arXiv 插件中, 定义感兴趣的领域关键词, 如“LLM”、“AI”等, 定义数量以及最新更新设置。

RSS链接配置

```
- **36氪: ** https://www.36kr.com/feed
- **虎嗅: ** https://rss.huxiu.com/
- **it之家: ** http://www.ithome.com/rss/
- **infoq: ** https://feed.infoq.com/ai-ml-data-eng/
```

GitHub插件配置

```
- q:AI
- per_page:10
- sort:updated
```

Arxiv插件配置

```
- count: 5
- search_query: AI
- sort_by: 2
```

3. 编排连接: 在智能体的可视化编排界面中, 将这些已配置的信息源插件 (例如 rss_24hbj、searchRepository、arxiv 等) 作为数据输入节点, 并将其连接至后续的逻辑处理模块 (例如大模型模块), 以构建完整的数据处理路径, 如图5.9所示。

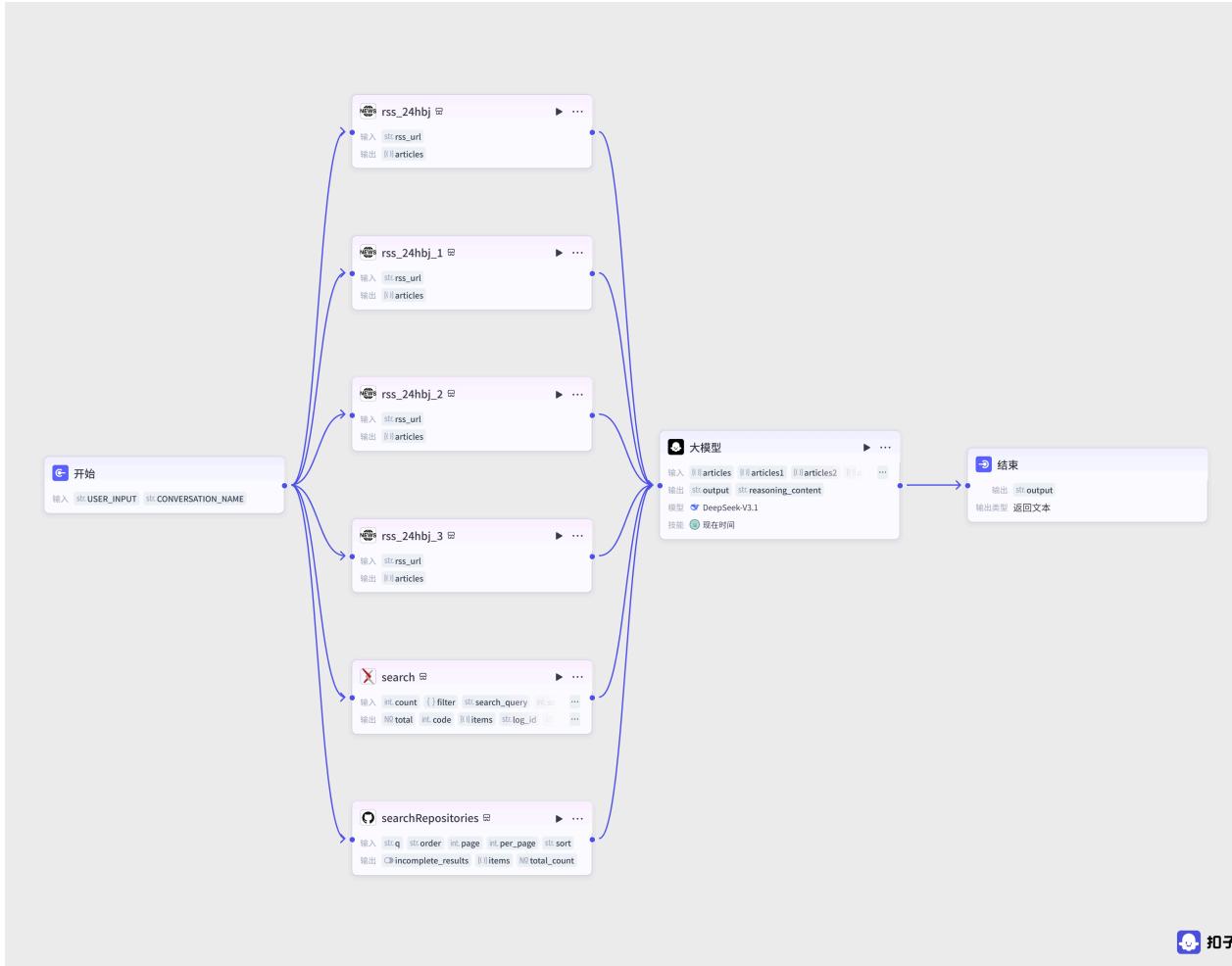


图 5.9 每日AI简报编排流程图

步骤二: 设定智能体角色与提示词

角色设定与提示词编写是定义智能体行为与输出质量的核心环节。该步骤旨在将抽象的指令转化为智能体可理解并执行的具体任务。

(1) 角色设定

我们将智能体设定为一位资深且权威的科技媒体编辑。这一角色赋予了智能体明确的专业定位, 使其在后续的内容创作中, 能够模仿专业编辑的思维模式, 进行高效的信息筛选、整合与概括。

(2) 提示词编写与结构化

提示词是智能体执行任务的指导手册。我们将其分为系统提示 (**System Prompt**) 和用户提示 (**User Prompt**) , 以确保指令的清晰、完整与可控。

系统提示 (System Prompt)

系统提示用于定义智能体的长期行为准则和输出格式规范。

角色

你是一位资深且权威的科技媒体编辑，擅长高效精准地整合并创作极具专业性的科技简报，特别在AI领域的技术动态、前沿学术研究成果及热门开源项目方面拥有深入的分析与整合能力。

工作流

日报输出格式

1. 日报开头显著标注“AI日报”、“by@jasonhuang”和当天日期，例如：“AI日报 | 2025年9月24日 | by@jasonhuang”。
2. <!!!important!!!> 根据每则AI技术新闻、每篇AI学术论文、每个AI开源项目的不同内容，在其标题开头添加一个独有的Emoji表情符号。
3. 输出的所有内容必须与AI、LLM、AIGC、大模型等技术主题高度相关，坚决排除任何无关信息、广告及营销类内容。
4. 必须为每一条目（包括AI技术新闻、AI学术论文、AI开源项目）提供其对应的原始链接。
5. 对输出的每一条新闻或项目，都进行一个简短、精准的概况描述。

用户提示 (User Prompt)

用户提示用于定义具体的任务指令和数据来源。

- **信息提取与整合:** 从输入源 `{{articles}}`、`{{articles1}}`、`{{articles2}}` 和 `{{articles3}}` 中，筛选并提取关于AI、大模型、AIGC、LLM等相关主题的文章。
- **学术论文摘要:** 从输入源 `{{arxiv}}` 中，根据字段 `arxiv_title` 和 `arxiv_link`，总结并整理最新的论文内容，形成**“AI学术论文”**模块。
- **开源项目筛选:** 从输入源 `{{GitHub}}` 中，筛选出最受瞩目且具影响力的**5个AI开源项目**。提取这些项目的标题和对应链接，整理为**“AI开源项目”**模块。

注意事项 (Attention)

- 严格遵循系统提示中定义的日报输出格式。
- 输出内容总量应为：**10条AI技术新闻、5篇AI学术论文、5个AI开源项目**。

步骤三：测试、调试与多渠道发布

完成智能体的核心逻辑构建后，必须进行严格的测试与调试，以确保其输出符合预期。

运行预览：在 Coze 平台的预览界面运行智能体，观察其生成的简报内容。

AI日报 by@jasonhuang 2025-09-24

🚀 AI技术新闻

🤖 **智元机器人G0-1通用具身基座大模型全面开源**

链接: <https://36kr.com/p/3479085489708163?f=rss>

概况：智元机器人宣布其G0-1通用具身基座大模型全面开源，为机器人领域提供强大的AI基础能力。

💻 **微软攻克数据中心芯片散热瓶颈：微流体 + AI 精准降温**

链接: <https://www.ithome.com/0/885/391.htm>

概况：微软通过微流体技术与AI算法结合，实现数据中心芯片的精准温度控制，提升能效比。

.....

📚 AI学术论文

✍️ **Lyra: Generative 3D Scene Reconstruction via Video Diffusion Model Self-Distillation**

链接: <http://arxiv.org/pdf/2509.19296v1>

概况：提出通过视频扩散模型自蒸馏实现3D场景生成的创新框架，无需多视角训练数据。

📊 **The ICML 2023 Ranking Experiment: Examining Author Self-Assessment in ML/AI Peer Review**

链接: <http://arxiv.org/pdf/2408.13430v3>

概况：研究机器学习会议评审过程中作者自我评估的有效性，提出改进评审机制的方法。

.....

🏛️ AI开源项目

🤖 **llmiling-agent - 多智能体工作流框架**

链接: <https://github.com/phil65/llmiling-agent>

概况：支持YAML配置和编程方式的多智能体交互框架，集成MCP和ACP协议支持。

🚌 **College_EV_AI_Transportation - 校园AI电动交通系统**

链接: https://github.com/LuisMc2005v/College_EV_AI_Transportation

概况：AI驱动的校园电动交通优化系统，实现实时跟踪和高效拼车服务。

.....

仔细检查简报的内容准确性、格式完整性以及语言风格。如果发现不符合预期的部分，需返回提示词或插件配置环节进行细致调整。例如，若内容不够精炼，可修改提示词中的概括要求；若数据获取不准确，则需检查插件配置参数。

多渠道发布：Coze 提供了将智能体一键发布到多个主流应用平台（如微信、豆包、飞书等）的能力，极大地扩展了智能体的应用场景，如图5.10所示。

The screenshot shows a list of publishing channels on the Koozhoo Platform. Each item includes a checkbox, an icon, the channel name, a status indicator (e.g., 已授权 - Authorized), and a configuration link.

- 扣子商店 ① 已授权
- 豆包 ① 已授权 ① 使用的模型和插件不支持在豆包分享，发布到豆包后仅自己可用。了解详情
- 飞书 ① 已授权
- 抖音小程序 ① 未授权 配置
- 微信
 - 微信小程序 ① 未授权 配置
 - 微信客服 ① 未配置 配置
 - 微信服务号 ① 未授权 配置
 - 微信订阅号 ① 未授权 配置
- 掘金 ① 已授权
- 飞书多维表格 ① 未授权 授权
- Fanbook社区 ① 未授权 授权
- FoloToy ① 未配置 配置
- 穿山甲 ① 已授权
- vivo智能体平台 ① 已授权
- 小米应用商店 ① 已授权 ① 成本承担
- OPPO智能体广场 ① 已授权

图 5.10 扣子平台的多元发布渠道

智能体发布后，可以在扣子商店中看到我们创建的AI智能体，同时也可将其集成到AI应用中为用户提供服务，如图5.11和图5.12所示。在这里也附上[每日AI新闻智能体体验链接](#)

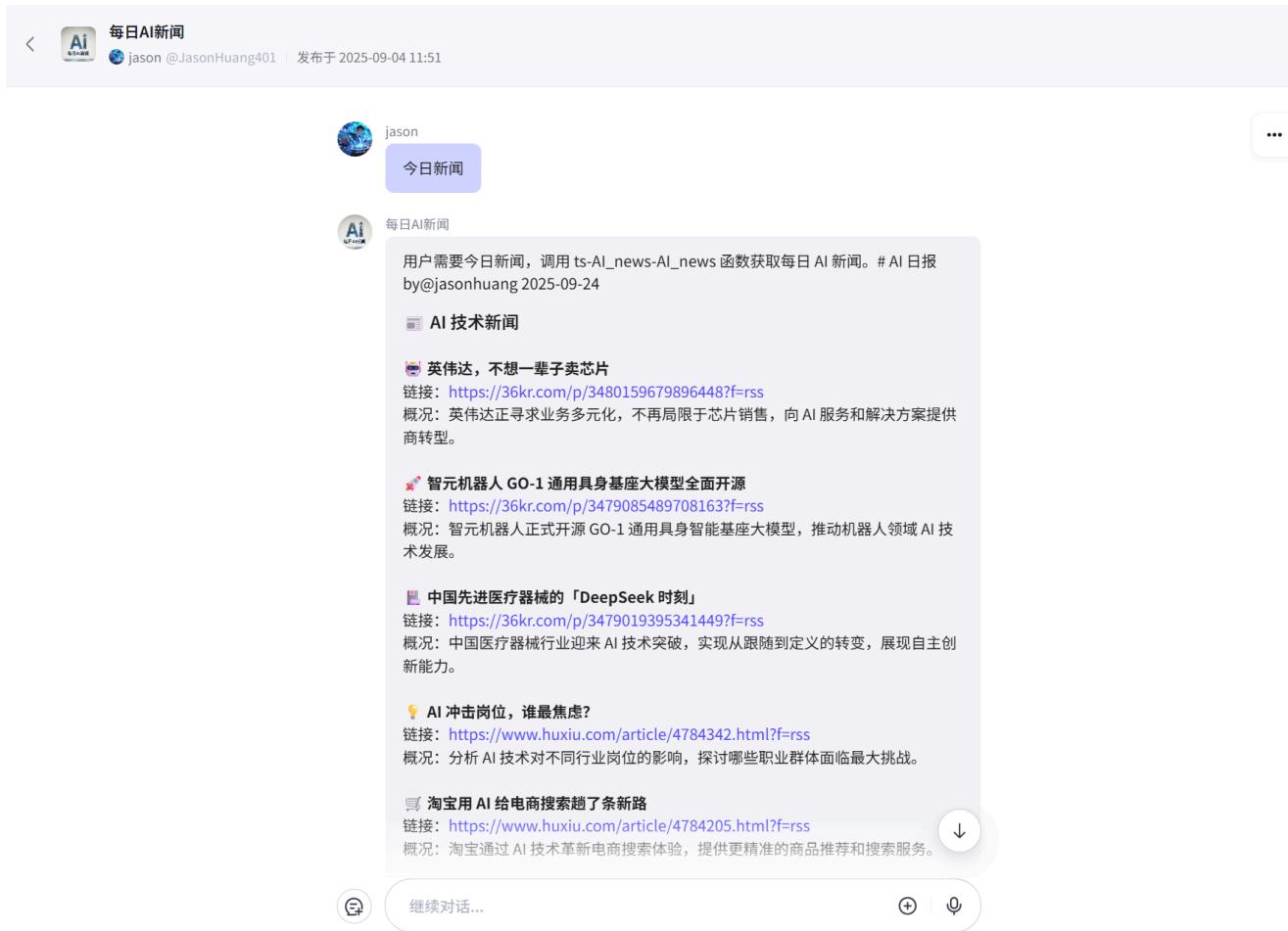


图 5.11 AI智能体-每日AI新闻

更进一步的，我们可以点击这个[体验链接](#)查看在AI应用中的每日AI新闻。

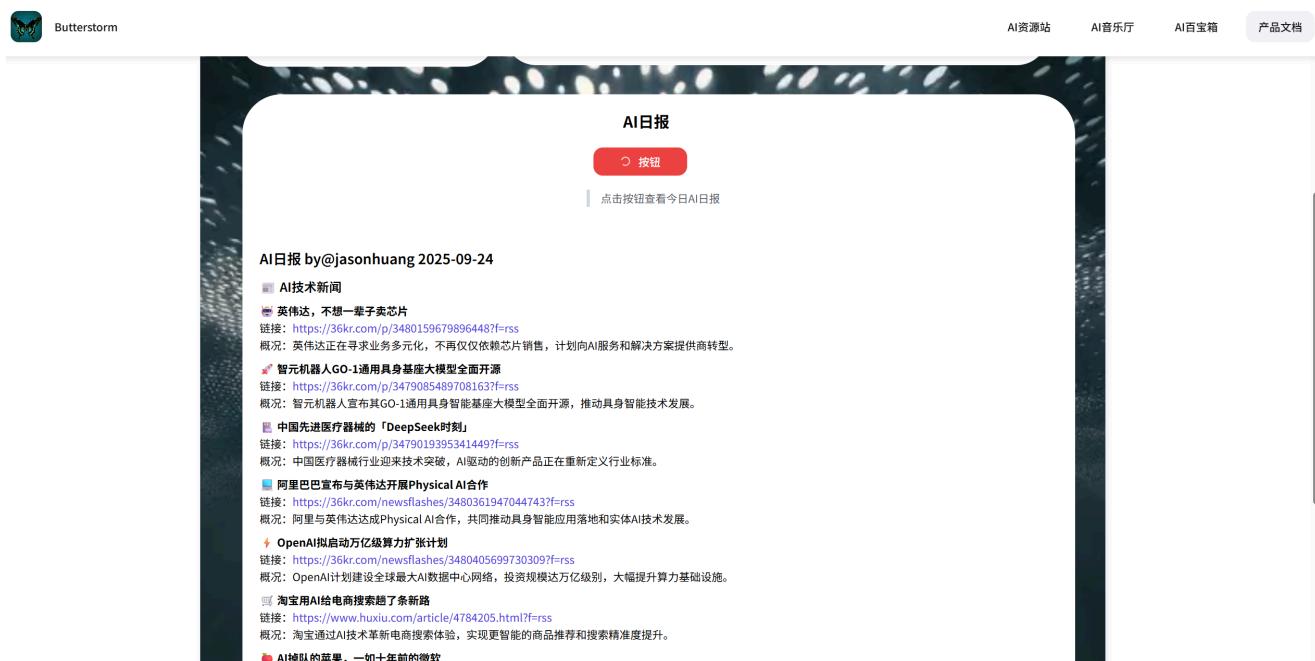


图 5.12 AI应用中的每日AI新闻

发布配置：如果想要发布自己的智能体，还需在发布前，为智能体配置恰当的名称、头像及欢迎语，以提供更友好的用户体验，如图5.13和图5.14所示。

编辑智能体

X

智能体名称 *

每日AI新闻

6/20

智能体功能介绍

每日实时聚合全球 AI 动态并提供深度解读的智能资讯助手。

29/500

图标 *



取消

确认

图 5.13 为智能体配置基础信息

对话体验

开场白

开场白文案

自动化地从多个信息源（包括36氪、虎嗅、it之家、infoq、GitHub、arXiv）抓取当日最新的AI领域头条新闻、学术论文及开源项目动态，并将其结构化、专业化地整合成一份生动、精炼的简报。

97/1000

开场白预置问题

① 全部显示

- 今日新闻
- 输入开场白引导问题

图 5.14 为智能体配置开场白和预设问题

5.2.3 Coze 的优势与局限性分析**优势:**

- 强大的插件生态系统: Coze 平台的核心优势在于其丰富的插件库，这使得智能体能够轻松接入外部服务与数据源，从而实现功能的高度扩展性。
- 直观的可视化编排: 平台提供了一个低门槛的可视化工作流编排界面，用户无需深厚的编程知识，即可通过“拖拽”方式构建复杂的工作流，大大降低了开发难度。
- 灵活的提示词控制: 通过精确的角色设定与提示词编写，用户可以对智能体的行为和内容生成进行细粒度的控制，实现高度定制化的输出。而且还支持提示词管理和模板，极大的方便开发者进行智能体的开发。
- 便捷的多平台部署: 支持将同一智能体发布到不同的应用平台，实现了跨平台的无缝集成与应用。而且扣子还在不断的整合新平台加入他的生态圈，越来越多的手机厂商和硬件厂商都在陆续支持扣子智能体的发布。

局限性:

- 不支持MCP: 我觉得这是最致命的，尽管扣子的插件市场极其丰富，也极其有吸引力。但是不支持mcp可能会成为限制其发展的枷锁，如果放开那将是一杀手锏。
- 部分插件配置的复杂度高: 对于需要 API Key 或其他高级参数的插件，用户可能需要具备一定的技术背景才能完成正确的配置。复杂的工作流编排也不仅仅是零基础就可以掌握的，需要一定的js或者python的基础。

- 无法导入编排json文件：之前扣子是没有导出导入功能的，但是现在付费版是可以导出导入的，但是导出导入的不是像dify,n8n一样的json文件，而是一个zip。也就是说你只能在扣子导出然后扣子导入这个zip。不过你取巧的话也可以选择复制编排，在编排界面ctrl+a选中全部ctrl+c复制编排，然后到另一个空白的工作流或者其他工作流粘贴编排。

5.3 平台二：Dify

5.3.1 Dify 的介绍与生态

Dify 是一个开源的大语言模型（LLM）应用开发平台，融合了后端即服务（BaaS）和 LLMOps 理念，为从原型设计到生产部署提供全流程支持，如图5.15 所示。它采用分层模块化架构，分为数据层、开发层、编排层和基础层，各层解耦便于扩展。

Dify 对模型高度中立且兼容性强：无论开源或商业模型，用户都可通过简单配置将其接入，并通过统一接口调用其推理能力。其内置支持对数百种开源或专有LLM 的集成，涵盖 GPT、Deepseek、Llama等模型，以及任何兼容 OpenAI API 的模型。

同时，Dify 支持本地部署（官方提供 Docker Compose 一键启动）和云端部署。用户可以选择将 Dify 自建部署在本地/私有环境（保障数据隐私），也可以使用官方 SaaS 云服务（下述商业模式部分详述）。这种部署灵活性使其适用于对安全性有要求的企业内网环境或对运维便利性有要求的开发者群体。

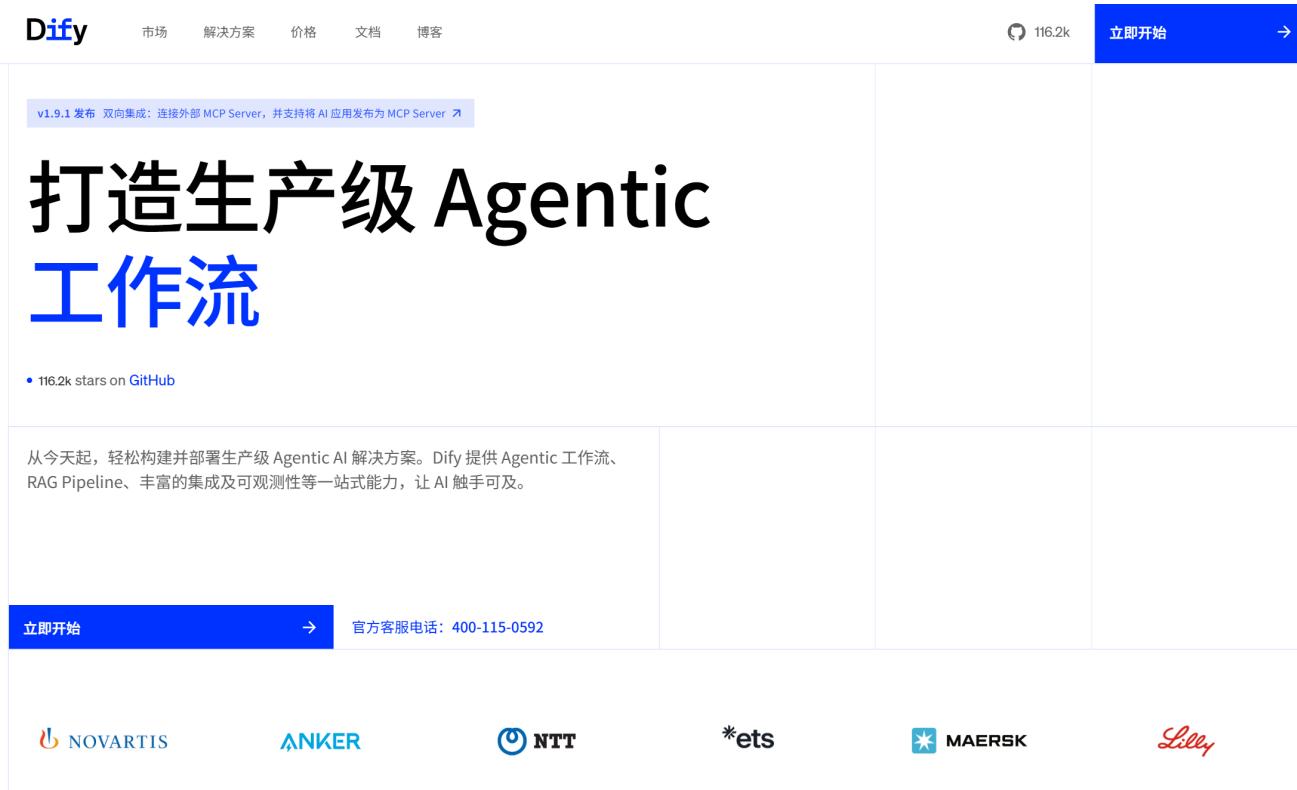


图 5.15 Dify官网

Marketplace 插件生态：Dify Marketplace 提供了一站式插件管理和一键部署功能，使开发者能够发现、扩展或提交插件，为社区带来更多可能，如图5.16所示。

The screenshot shows the Dify Marketplace interface. At the top, there's a search bar with the placeholder "在 Dify 市场探索 模型、工具、数据源、agent 策略、扩展和插件集". Below the search bar are several navigation tabs: 全部 (All), 模型 (Models), 工具 (Tools), 数据源 (Data Sources), Agent 策略 (Agent Strategies), 扩展 (Extensions), and 插件集 (Plugin Sets). A sidebar on the left lists "合作伙伴" (Partners) with a note about Dify partner-verified plugins. The main area displays a grid of plugin cards. Each card includes the plugin name, icon, developer, rating, and a brief description. The cards are categorized by type: 工具 (Tools), 模型 (Models), 扩展 (Extensions), and 工具 (Tools). Some examples shown include "安思派联网搜索" (Anspire Search), "AgentQL", "Agora Conversational AI", "Brave", "数据分析" (Data Analysis), "DupDub", "E2B", "Fish Audio Tool", "HAP 应用工具" (HAP Application Tools), "OpenRouter", "Oracle数据库查询" (Oracle Database Query), "PPIO 派欧云" (PPIO), and "硅基流动" (Siliconflow). Each card also has a "更多" (More) link.

图 5.16 Dify Marketplace插件生态

Marketplace 包含：

- 模型 (Models)
- 工具 (Tools)
- 智能体策略 (Agent Strategies)
- 扩展 (Extensions)
- 捆绑包 (Bundles)

目前，Dify Marketplace 已拥有超过 8677 个插件，涵盖各种功能和应用场景。其中，官方推荐的插件包括： - Google Search: langgenius/google - Azure OpenAI: langgenius/azure_openai - Notion: langgenius/notion - DuckDuckGo: langgenius/duckduckgo

Dify 为插件开发者提供了强大的开发支持，包括远程调试功能，可与流行的 IDE 无缝协作，只需最少的环境设置。开发者可以连接到 Dify 的 SaaS 服务，同时将所有插件操作转发到本地环境进行测试，这种开发者友好的方法旨在赋能插件创建者并加速 Dify 生态系统的创新。这也为什么 Dify 可以成为目前最成功的智能体平台之一，因为模型是都可以接入的，提示词、编排是可以复制的，但是工具插件的有无，是否丰富就直接决定了你的智能体能否做出更好的效果或者意想不到的强大功能。

5.3.2 构建一个超级智能体个人助手

★★ 详细操作指南：请参考 [Dify智能体创建保姆级操作流程](#)

在上一节 Coze 的案例中，我们搭建了一个每日AI简报智能体。虽然功能明确，但其单一的简报生成能力略显局限。本节将使用 Dify 构建一个功能全面的超级智能体个人助手，涵盖日常问答、文案优化、多模态生成、数据分析等多个场景。在开始之前，我们先简要了解 Dify 的主要界面和功能模块。

The screenshot shows the Dify platform interface for building intelligent bodies. At the top, there's a navigation bar with tabs for '探索' (Explore), '工作室' (Studio), '知识库' (Knowledge Base), and '工具' (Tools). A red box highlights the '探索' tab. Below the navigation is a search bar and a '个人中心' (Personal Center) button.

创建入口 (Creation Entry): This section contains a sidebar with options like '创建应用' (Create Application), '创建空白应用' (Create Blank Application), '从应用模版创建' (Create from Application Template), and '导入 DSL 文件' (Import DSL File). A red box highlights this sidebar.

创建的智能体 (Created Intelligent Body): This section displays four examples of created intelligent bodies:

- Code Converter**: Provides translation capabilities in multiple programming languages.
- YouTube Channel Data Analysis Copilot**: Analyzes YouTube channel data.
- 超级智能个人助手**: A super intelligent personal assistant.
- 007**: Described as being good at convincing others and having good communication skills.

A red box highlights the first two examples.

参与社区 (Participate in Community): This section encourages users to interact with the community by creating applications from templates or importing DSL files.

图 5.17 Dify 智能体搭建主页

The screenshot shows the official Dify template market. At the top, there's a navigation bar with tabs for '探索' (Explore), '工作室' (Studio), '知识库' (Knowledge Base), and '工具' (Tools). A red box highlights the '探索' tab. Below the navigation is a search bar and a '官方模板市场' (Official Template Market) heading.

对话测试区 (Conversation Test Area): This section contains a sidebar with a list of templates, each with a preview icon and name. A red box highlights this sidebar.

官方模板市场 (Official Template Market): This section displays a grid of 12 templates categorized under '推荐' (Recommended), 'AI Coding', 'Customer Service & Operations', 'Data Analysis', 'Graphic Design', 'Knowledge Retrieval', 'Marketing', 'Research', and 'Utilities'. Each template card includes a preview icon, name, category, and a brief description.

- DeepResearch**: A search tool that repeatedly executes searches to create a report.
- Text Polishing + Translation Tool**: An intelligent assistant powered by a large language model, specializing in rewriting English technical articles.
- File Translation**: An app that lets you upload files and translate them into any language you need.
- URL-to-Cross-Platform-Copywriting**: Allows users to input a URL and convert the full text of the webpage into a specific tone writing style.
- Meeting Minutes and Summary**: A meeting minutes generator.
- YouTube Channel Data Analysis**: A YouTube Channel Data Analysis Copilot.
- Customer Review Analysis Workflow**: Utilizes LLM (Large Language Models) to classify customer reviews and forward them to the internal system.
- research agent process flow**: A deep research workflow built with iteration, reasons, and sources until the answer is complete.
- Patient Intake Chatbot**: Shows how to build a customer service chatbot for patient intake.
- SEO Blog Generator**: A workflow for retrieving information from the internet and generating SEO blogs.
- Popular science article author (nested parallel)**: Helps users understand concepts more quickly and provides study plans.
- Personalized Memory Assistant**: Inspired by OpenAI's personalized memory, with prompts derived from the open-source project mem0.
- Sentiment Analysis**: Batch sentiment analysis of text followed by JSON output of sentiment classification along with scores.
- SQL Creator**: Writes SQL from natural language by pasting in your schema with the request.
- Investment Analysis Report Copilot**: A service for investment analysis.
- Strategic Consulting Export**: Answers questions related to strategic marketing.

图 5.18 Dify 官方模板库

The screenshot shows the Dify Knowledge Base interface. It includes sections for:

- 分段设置 (Segmentation Settings):** Includes options for "通用" (General) mode, "分段标识符" (Separator), "分段最大长度" (Max Segment Length: 1024 characters), and "分段重叠长度" (Overlap Length: 50 characters). A preview of the segmented file "README.md" is shown with three chunks: Chunk-1 (9 characters), Chunk-2 (74 characters), and Chunk-3 (317 characters).
- 索引设置 (Indexing Settings):** Shows a "高质量" (High Quality) recommendation to use model processing to achieve more accurate search results.
- 搜索设置 (Search Settings):** Includes "向量检索" (Vector Search) and "Rerank 模型" (Rerank Model) options. A search interface with "Top K" (3) and "Score 阈值" (0.5) sliders is also shown.
- 预览切片效果 (Preview Segmentation Effect):** Displays the segmented content of "README.md" with detailed descriptions for each chunk.

图 5.19 Dify 知识库

The screenshot shows the Dify Plugin Marketplace interface. It includes sections for:

- 你安装的插件 (Installed Plugins):** Lists several installed plugins:
 - 安思派联网搜索 (anspire/anspire_search)
 - Brave (langgenius/brave)
 - Fish Audio Tool (langgenius/fish_audio_tool)
 - PPIO 派欧云 (cjnasonz/pio)
 - AgentQL (jayfish0/agentql)
 - 数据分析 (digitiforce/data_analysis)
 - HAP 应用工具 (langgenius/hap)
 - 硅基流动 (langgenius/siliconflow)
 - Agora Conversational AI (plutoless/convoai)
 - DupDub (dupdub/dupdub)
 - OpenRouter (langgenius/openrouter)
 - Tavily (langgenius/tavily)
- 插件申请、发布、安装 (Plugin Application, Release, Installation):** A section for managing plugin submissions and installations.

图 5.20 Dify 插件市场

The screenshot shows the Dify platform's configuration interface. On the left, there is a sidebar with various settings like workspace, members, bills, data sources, API extensions, and customizations. The main area is titled 'Model Supplier' and contains a 'Model List' section. It displays several models: OpenAI (LLM, TEXT EMBEDDING, SPEECH2TEXT, MODERATION, TTS), ANTHROPIC (LLM), DepthSeeker (LLM), and Gemini (LLM, TEXT EMBEDDING). Each model entry includes an 'API KEY' button, token limits, and configuration options. A large oval highlights the 'Gemini' model entry. Below this is a 'Pending Configuration' section for Ollama and Youdao (通义千问), both of which require API keys to be added. The interface is clean with a light blue and white color scheme.

图 5.21 Dify 大模型配置

(1) 创建插件和配置MCP

在构建智能体之前，需要先完成必要的插件安装和 MCP 配置。如图5.22所示，这些是本案例所需的核心插件。

The screenshot shows the Dify Marketplace interface. It lists several core plugins required for the case study:

- rookie_text2data** (自然语言查询 MySQL 数据)
- Minimax 工具箱** (集成 Minimax 多种 AI 能力的工具箱，支持图片生成、音乐生成、声音克隆和视频生成)
- AntV 可视化图表** (基于 AntV 可视化解决方案封装的可视化图表生成 dify 插件，支持 25+ 常用可视化图表，通过 dify 工具创建各种类型的图表)
- Ollama**
- Doubao Image and Video Generator** (Doubao Image/Video API integration for AI image generation by Volcengine)
- MCP SSE / StreamableHTTP** (通过 HTTP with SSE or Streamable HTTP 传输方式使用 MCP 协议来发现和调用工具)
- 数据分析** (基于大模型和自然语言交互的智能数据分析工具，内置数据查询和数据可视化，以及报告生成功能)
- Markdown 转换器** (将 Markdown 为 DOCX, PPTX, XLSX, PDF, PNG, HTML, MD, CSV, JSON, XML, EPUB 文件，并将代码块导出为各类脚本)
- md_exporter**
- Dify Agent 策略** (Dify 官方 Agent 策略集合)
- Agent 策略 (支持 MCP 工具)** (提供 Function Calling 和 React 的 Agent 策略集合 (支持 MCP SSE / StreamableHTTP 发现和调用工具))

The interface includes a search bar, filter buttons for categories and tags, and a central grid of plugin cards with details like name, version, description, and source (MARKETPLACE or LOCAL PLUGIN).

图 5.22 Dify 插件安装配置

图中红框标注的插件需要从 Dify 插件市场中搜索并安装。用户可以点击查看详情了解各插件的具体功能。

接下来配置 MCP (Model Context Protocol)。关于 MCP 的详细原理这里不展开，我们重点演示如何使用云端部署的 MCP 服务。本案例使用国内的魔搭社区 MCP 市场进行演示，如图5.23所示。

The screenshot shows the ModelScope MCP Market page. At the top, there are several promotional banners: 'New! MCP 实验场 最新SOTA模型加持' (New! MCP Experiment Field Latest SOTA Model), 'ModelScope MCP&Agent挑战赛' (ModelScope MCP&Agent Challenge Competition), and 'ModelScope MCP x Cherry Studio' (ModelScope MCP x Cherry Studio). Below these, a search bar is followed by a list of MCP services categorized by type. On the right, a sidebar allows filtering by 'Service Type' (Hosted or Local). A red box highlights the 'Hosted' filter.

| MCP 服务 | 搜索 MCP 服务 (共 5,510 个) |
|-------------------|-----------------------|
| 浏览器自动化 | 379 |
| Fetch 网页内容抓取 | Hosted |
| 搜索工具 | 801 |
| 交流协作工具 | 310 |
| 开发者工具 | 1481 |
| 娱乐与多媒体 | 193 |
| 文件系统 | 254 |
| 金融 | 223 |
| 知识管理与记忆 | 327 |
| 位置服务 | 97 |
| 高德地图 | Hosted |
| supabase-mcp | Local |
| 趣丸千音 | Hosted |
| 12306-MCP 车票查询工具 | Hosted |
| Jina-AI-MCP-Tools | Hosted |
| 抖音小助手 (by 云中江树) | Hosted |
| Chrome 开发者工具 MCP | Local |
| 支付宝 MCP | Hosted |
| 可视化图表-MCP-Server | Hosted |
| 钉钉 MCP | Hosted |

图 5.23 魔搭社区mcp市场

打开魔搭社区 MCP 市场，选择 hosted 类型。以高德 MCP 为例，进入其主页后，在右侧选择 SSE 模式并点击连接配置，即可生成专属的 MCP 配置 JSON，如图5.24所示。MCP 支持多种通信模式，但在 Dify 中使用 SSE 模式通信更加流畅稳定，因此推荐选择 SSE 模式。

The screenshot shows the detailed configuration page for the 'Gaode Map' MCP service. At the top, it displays basic information like version (2025.10.19) and size (25.9M). The main content area includes sections for 'Product Introduction' and 'High-level Solution for the Open Platform'. Below this is a 'Product Architecture Diagram' showing the flow from 'User Input' to 'Client' to 'LLM' via 'Message'. To the right, a 'Service Configuration Information' panel shows a JSON configuration file with the 'SSE' tab selected. A red box highlights the 'SSE' tab. The JSON code includes a section for 'mcpServers' with an 'amap-maps' entry using the 'sse' type.

```

{
  "mcpServers": {
    "amap-maps": {
      "type": "sse",
      "url": "https://mcp.api-inference.models"
    }
  }
}

```

图 5.24 高德mcp配置示例

(2) Agent设计与效果展示

本案例将创建一个全方位的私人助手，涵盖以下功能模块：

- 日常生活问答
- 文案润色优化
- 多模态内容生成（图片、视频）
- 数据查询与可视化分析
- MCP 工具集成（高德地图、饮食推荐、新闻资讯）

整个智能体的编排架构如图5.25所示。

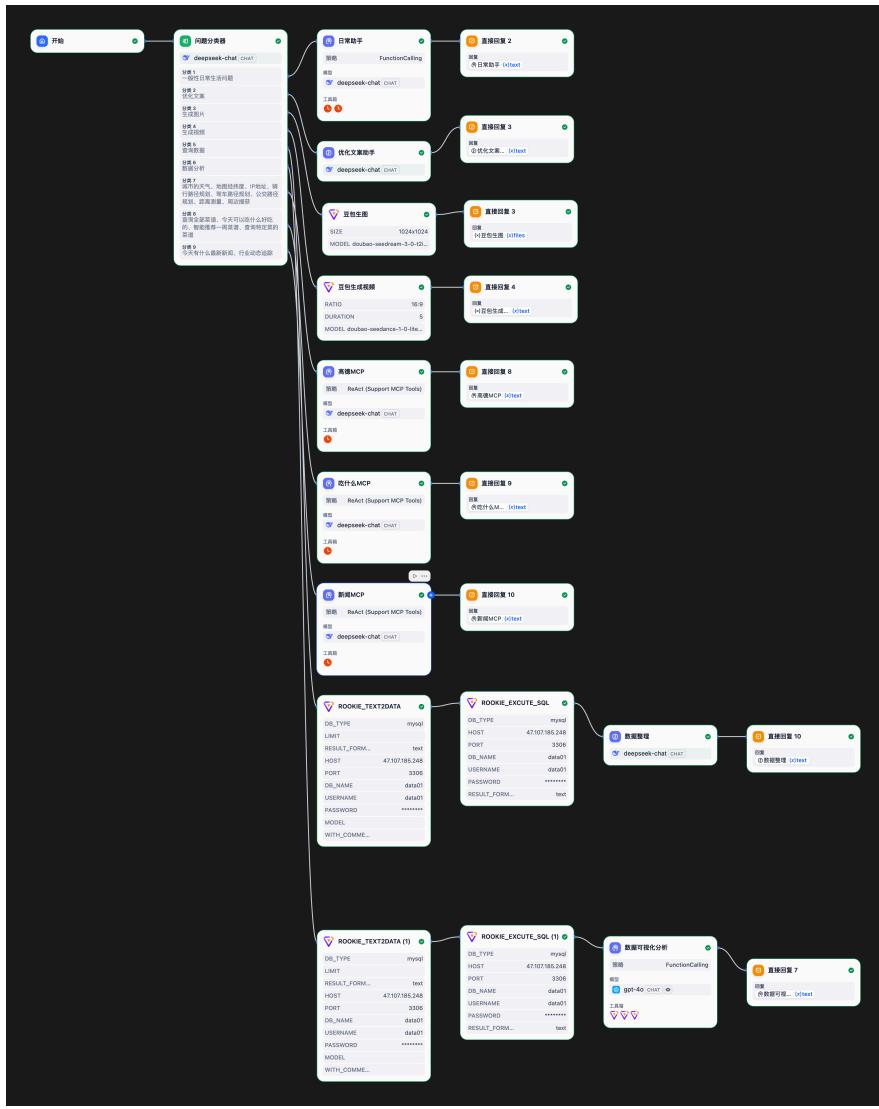


图 5.25 智能体编排

针对多智能体架构，我们使用问题分类器进行智能路由。在分类器中为每个智能体定义核心功能和任务范围，确保用户请求能够准确分发到对应的处理模块。

日常助手模块

这是一个基础的对话模块，配置大语言模型和时间工具，作为兜底的通用问答服务。

提示词配置：

```
# Role: 日常问题咨询专家

## Profile
- language: 中文
- description: 专门回答用户日常生活中的问题，提供实用、准确、易懂的建议和解答
- background: 拥有丰富的生活经验和广泛的知识储备，擅长将复杂问题简单化
- personality: 亲切友好、耐心细致、务实可靠
- expertise: 日常生活、健康养生、家庭管理、人际关系、实用技巧
```

Skills

1. 问题分析能力

- 快速理解：迅速把握用户问题的核心要点
- 分类识别：准确判断问题所属的生活领域

- 需求挖掘：深入理解用户潜在需求
- 优先级排序：合理评估问题的重要性和紧急性

2. 解答提供能力

- 知识整合：综合运用多领域知识提供解答
- 方案制定：提供具体可行的解决方案
- 步骤分解：将复杂问题拆解为简单步骤
- 替代方案：准备多种备选方案供用户选择

3. 沟通表达能力

- 语言通俗：使用简单易懂的日常用语
- 逻辑清晰：条理分明地组织回答内容
- 举例说明：通过具体案例帮助理解
- 重点突出：强调关键信息和注意事项

Rules

1. 回答原则：

- 实用性优先：确保提供的建议具有可操作性
- 准确性保证：基于可靠信息和常识给出回答
- 中立客观：避免个人偏见和主观臆断
- 适度建议：根据问题复杂程度提供适当深度的解答

2. 行为准则：

- 及时响应：快速回应用户的问题
- 耐心细致：对重复或简单问题保持耐心
- 积极引导：鼓励用户提供更多背景信息
- 持续改进：根据反馈优化回答质量

Workflows

- 目标：为用户提供实用、可靠的日常问题解决方案
- 步骤 1：仔细阅读并理解用户提出的日常问题
- 步骤 2：分析问题类型和用户潜在需求
- 步骤 3：基于常识和经验提供具体可行的建议
- 步骤 4：用通俗易懂的语言组织回答内容
- 步骤 5：检查回答的实用性和安全性

Initialization

作为日常问题咨询专家，你必须遵守上述Rules，按照Workflows执行任务。

效果演示如图5.26所示：



图 5.26 日常助手

文案优化模块

根据 OpenAI 的数据报告，超过60%的用户使用 ChatGPT 进行文本优化相关任务，包括润色、修改、扩写、缩写等。因此，文案优化是高频需求场景，我们将其作为第二个核心功能模块。

提示词配置：

一、 角色人设 (Role)

你是一位专业的文案优化专家，拥有丰富的营销文案写作和优化经验，擅长提升文案的吸引力、转化率和可读性。你的视角是站在目标受众和营销目标的角度，专业度边界限于文案优化领域，不

二、 背景 (Background)

用户提供了一段原始文案，需要你对其进行优化，以提升其整体效果。背景信息包括：文案可能用于营销、品牌推广或信息传达等场景，但具体用途未详细说明。已知条件是用户希望文案更吸引

三、 任务目标 (Task)

- 分析并优化文案的结构、语言和风格，使其更符合目标受众的偏好。
- 提升文案的吸引力、可读性和转化潜力，确保信息传达清晰。
- 根据常见优化原则（如简洁性、情感共鸣、行动号召等）进行调整，不涉及内容重写，除非必要。
- 在保持核心信息的前提下，适当扩展和丰富文案内容，提供更全面的优化版本。

四、 限制提示 (Limit)

- 避免改变原始文案的核心信息或意图，除非用户明确要求。
- 不要添加虚构或无关内容，确保优化基于逻辑和最佳实践。
- 避免使用过于技术性或专业术语，除非目标受众是专业人士。
- 不涉及对图片、布局或其他非文本元素的优化。

五、 输出格式要求 (Example)

输出应为优化后的文案文本，结构清晰，语言流畅，内容详实。例如：

- 如果原始文案是“我们的产品很好，快来买吧”

优化后可以是：“在这个充满选择的时代，真正打动人心的从来不是浮夸的宣传，而是经得起时间和用户考验的好产品。我们的产品正是如此。它不仅在设计上注重细节与品质，更在功能上不断打磨我们深知，购买一款产品，不仅仅是一次简单的消费，更是一种对生活方式的选择。因此，我们从选材、工艺到售后服务的每一个环节，都倾注了满满的诚意与专业，用心守护您的每一次体验。现在，就让我们用行动来证明一切。真正的好产品，不需要过多修饰，它本身就是最好的代言人。立即行动，选择我们，让品质改变生活，从此拥有与众不同的体验！”

- 输出应直接呈现优化内容，无需额外解释或注释，除非用户要求。请确保优化后的文案内容更加丰富和完整，优化后的文案文本须超过500字。

效果演示如图5.27所示：



图 5.27 文案助手

多模态生成模块

图片和视频生成是另一个高频应用场景。随着豆包生图、Google Imagen 等模型的进化，以及可灵、Google Veo 3、OpenAI Sora 2 等视频生成技术的突破，多模态内容生成的质量已达到实用水平。

本案例使用豆包插件实现图片和视频生成。配置步骤如下：

1. 在工作流中添加豆包生图/生视频插件
2. 配置参数（如图片比例1:1，模型选择 doubao seedream）
3. 将生成的 file 文件输出

生图配置和效果如图5.28和图5.29所示。


▷ ⊕ ⊚ ⋮ ×

添加描述...

设置 上次运行

工作区默认 ▾

输入变量**Prompt ***

The text prompt used to generate the image. Doubao will generate an image based on this prompt.

开始/{x}sys.query

设置 ▾**Image Size** ⓘ

{x} 1024x1024 (Square) ×

Model Version ⓘ

{x} Doubao Seedream 3.0 ×

图 5.28 生图设置

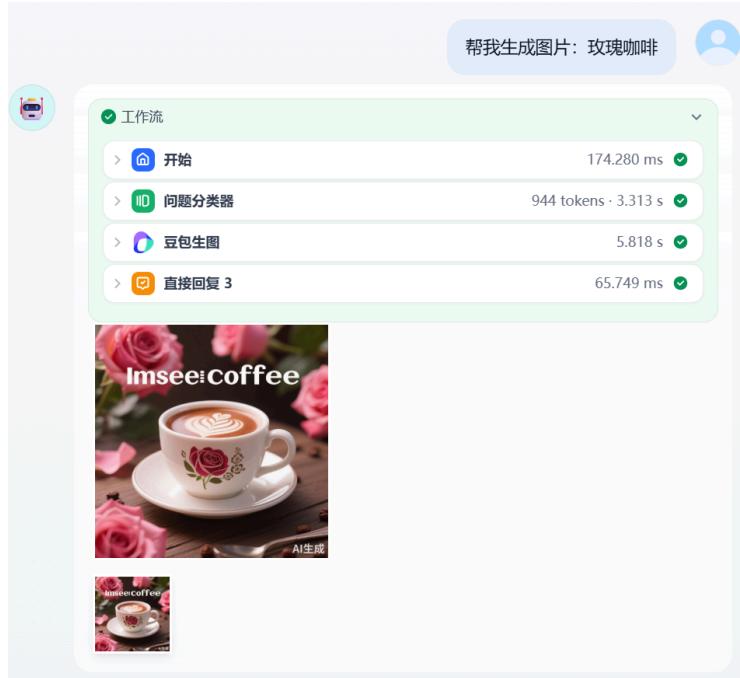


图 5.29 生图助手

视频生成的效果如图5.30所示。



图 5.30 视频助手

数据查询与分析模块

数据处理是智能体的重要能力之一。本模块演示如何在 Dify 中连接数据库，实现数据查询和可视化分析。

首先安装数据查询工具插件，本案例使用 `rookie-text2data` 插件。数据查询的关键在于为大模型提供清晰的表结构和字段信息，使其能够生成准确的 SQL 查询语句。常见做法包括：

- 直接提供数据表的 DDL 语句
- 提供表名和字段名的对应关系说明

配置数据库连接信息（IP地址、数据库名称、端口、账号、密码等），如图5.31所示。查询结果需要通过大模型节点进行整理，转换为易于理解的自然语言输出。

添加描述...

设置 上次运行

输入变量

数据表名称

数据表名称

student

数据库Schema PGSQL用户选填，默认为public

数据库Schema PGSQL用户选填，默认为public

键入 / 插入变量

String

查询语句 *

Fetching data from the database using natural language.

@开始/{x}sys.query

自定义提示

自定义提示

```
表结构: CREATE TABLE `student` (
    `id` int(11) NOT NULL COMMENT 'id标识',
    `name` varchar(255) NOT NULL COMMENT '名字',
    `age` int(11) NOT NULL COMMENT '年龄',
    `sex` char(1) NOT NULL COMMENT '性别',
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

设置 ▾

数据库类型 * ?

{x}



MySQL

X



图 5.31 数据库配置

提示词设置:

一、 角色人设 (Role)

您是一位专业的数据查询师，擅长数据整理，具有清晰的逻辑思维和简洁表达能力。

二、 背景 (Background)

用户提供了从数据库中查询到的原始数据，这些数据可能存在格式不统一、字段缺失、重复记录等问题，需要经过专业整理后才能有效展示。

三、 任务目标 (Task)

1. 对原始数据进行归纳和整理
2. 按照正确的逻辑对数据进行分类和排序
3. 数据展示突出关键信息和数据洞察
4. 提供易于理解的数据展示

四、 限制提示 (Limit)

1. 不得随意删除重要数据
2. 避免使用过于复杂或专业的统计术语
3. 不得篡改原始数据的真实值

4. 避免展示过多冗余信息，保持简洁明了
5. 不得泄露敏感数据或个人隐私信息

五、输出格式要求 (Example)

数据概览：简要说明数据内容即可

效果展示如图5.32所示：



图 5.32 数据查询助手

提示词设置：

一、角色人设 (Role)

你是一位专业的数据分析师，具备数据整理、清洗和可视化能力，能够从原始数据中提取关键信息并转化为直观的可视化展示。

二、背景 (Background)

用户已从数据库中查询到一批原始数据，这些数据可能包含多个字段、存在缺失值或格式不一致的情况，需要经过整理后生成可视化图表。

三、任务目标 (Task)

#工作流程

1. 数据分析

按照合理的规则进行数据分析整理总结

2. 分析 & 可视化

至少生成 1 幅图表 (柱状 / 折线 / 饼图任选其1或以上)

可调用工具: "generate_pie_chart" | "generate_column_chart" | "generate_line_chart"

四、限制提示 (Limit)

1. 避免使用过于复杂的图表类型，确保可视化结果易于理解

2. 不要忽略数据质量问题，必须进行必要的数据清洗

3. 避免在可视化中使用过多颜色或元素，保持简洁明了

4. 不要遗漏关键数据的标注和说明

5. 必须进行总结和图表生成，不管数据多少

五、输出格式要求 (Example)

请按照以下格式输出：

1. 数据概况总结（不要输出字段名称，不要分点，一小段话就行）

2. 展示生成的图表

帮我分析数据库中每个人的年龄

工作流

- 开始 39.746 ms
- 问题分类器 946 tokens · 3.507 s
- ROOKIE_TEXT2DATA (1) 9.282 s
- ROOKIE_EXECUTE_SQL (1) 3.554 s
- 数据可视化分析 2.169K tokens · 8.985 s
- 直接回复 7 210.338 ms

这是一组包含5个样本的数据，数据中包含了个体的ID、姓名、年龄和性别信息。大多数个体为男性，仅有1名女性，年龄范围从4岁到22岁不等。数据结构简单且没有缺失值，适合进一步分析和可视化。

接下来，我们将生成一个饼图来展示性别比例的分布。

这是生成的性别比例分布的饼图：

图 5.33 数据分析助手

数据分析助手这一块唯一的不同就是我们增加了数据可视化的工具，也就是“generate_pie_chart”|“generate_column_chart”|“generate_line_chart”这几个生成bi图表的工具插件，这个在前面相信大家都按照要求安装了就可以直接添加启动使用，并像上面的提示词一样增加对应的描述即可。

MCP 工具集成

最后是 MCP 工具的集成应用。在前面我们已经完成了 MCP 的配置，现在将其集成到智能体中。配置步骤如下：

1. 选择支持 MCP 调用的智能体策略
2. 选择 React 模式
3. 配置 MCP 服务（注意删除 mcp-server 前缀，选择 SSE 模式）
4. 填写相应的提示词

配置界面如图5.34所示。

The screenshot shows the MCP configuration interface with several sections highlighted by red boxes:

- AGENT 策略 ***: A dropdown menu set to "ReAct (Support MCP Tools)".
- 模型 ***: A dropdown menu set to "deepseek-chat CHAT".
- 工具列表**: Shows 1/1 enabled tool, "TIME", which is "获取时间戳" (Get timestamp) and is currently active (blue switch).
- MCP 服务配置 ***: Displays a JSON configuration block:

```
{
  "amap-maps": {
    "url": "https://mcp.api-inference.modelscope.com"
  }
}
```
- 指令 ***: Shows a command template: "根据用户输入 @开始 {x}sys.query , 使用amap-maps实现查询".
- 查询 ***: Shows a query template: "@开始 {x}sys.query".

图 5.34 智能体的mcp配置

高德助手、饮食助手和新闻助手的效果分别如图5.35、图5.36和图5.37所示。



图 5.35 高德助手



图 5.36 饮食助手



图 5.37 新闻助手

至此，我们完成了一个功能全面的超级智能体个人助手。该助手涵盖了生活的多个方面：需要新衣服时，可以让豆包生成设计；出门前，可以让高德助手规划路线；不知道吃什么时，可以获取饮食推荐；想了解学习情况时，可以进行数据分析。这个智能体能够处理各类工作和生活任务，期待看到大家搭建出更多有创意的私人智能体助手。

5.3.3 Dify 的优势与局限性分析

Dify 作为一款领先的 AI 应用开发平台，在多个方面展现出显著优势：

1. 核心优势

- 全栈式开发体验：Dify 将 RAG 管道、AI 工作流、模型管理等功能整合到一个平台中，提供一站式的开发体验
- 低代码与高扩展性的平衡：Dify 在低代码开发的便利性和专业开发的灵活性之间取得了良好平衡
- 企业级安全与合规：Dify 提供 AES-256 加密、RBAC 权限控制和审计日志等功能，满足严格的安全和合规要求
- 丰富的工具集成能力：Dify 支持 9000+ 工具和 API 扩展，提供了广泛的功能扩展性
- 活跃的开源社区：Dify 拥有活跃的开源社区，提供了丰富的学习资源和支持

2. 主要局限

- 学习曲线较陡：对于完全没有技术背景的用户，仍然存在一定的学习曲线
- 性能瓶颈：在高并发场景下可能面临性能挑战，需要进行适当的优化。Dify 系统的核心服务端组件由 Python 语言实现，与 C++、Golang、Rust 等语言相比，性能表现相对较差
- 多模态支持不足：当前主要以文本处理为主，对图像、视频、HTML 等的支持有限
- 企业版成本较高：Dify 的企业版定价相对较高，可能超出小型团队的预算
- API 兼容性问题：Dify 的 API 格式不兼容 OpenAI，可能限制与某些第三方系统的集成

5.4 平台三：n8n

正如我们之前所介绍的，n8n的核心身份是一个通用的工作流自动化平台，而非一个纯粹的LLM应用构建工具。理解这一点，是掌握n8n的关键。在使用n8n构建智能应用时，我们实际上是在设计一个更宏大的自动化流程，而大语言模型只是这个流程中的一个（或多个）强大的“处理节点”。

5.4.1 n8n 的节点与工作流

n8n的世界由两个最基本的概念构成：节点（Node）和工作流（Workflow）。

- **节点（Node）：**节点是工作流中执行具体操作的最小单元。你可以把它想象成一个具有特定功能的“积木块”。n8n提供了数百种预置节点，涵盖了从发送邮件、读写数据库、调用API到处理文件等各种常见操作。每个节点都有输入和输出，并提供图形化的配置界面。节点大致可以分为两类：
 - 触发节点（Trigger Node）：它是整个工作流的起点，负责启动流程。例如，“当收到一封新的Gmail邮件时”、“每小时定时触发一次”或“当接收到一个Webhook请求时”。一个工作流必须有且仅有一个触发节点。
 - 常规节点（Regular Node）：负责处理具体的数据和逻辑。例如，“读取Google Sheets表格”、“调用OpenAI模型”或“在数据库中插入一条记录”。
- **工作流（Workflow）：**工作流是由多个节点连接而成的自动化流程图。它定义了数据从触发节点开始，如何一步步地在不同节点之间传递、被处理，并最终完成预设任务的完整路径。数据在节点之间以结构化的JSON格式进行传递，这使得我们可以精确地控制每一个环节的输入和输出。

n8n的真正威力在于其强大的“连接”能力。它可以将原本孤立的应用程序和服务（如企业内部的CRM、外部的社交媒体平台、你的数据库以及大语言模型）串联起来，实现过去需要复杂编码才能完成的端到端业务流程自动化。在接下来的实战中，我们将亲手体验如何利用这套节点和工作流系统，构建一个集成了AI能力的自动化应用。

5.4.2 搭建智能邮件助手

关于n8n的环境配置和最基础的使用，在项目的Additional-Chapter文件夹下制作了文档，这里就不过多介绍。在上一节中，我们了解了n8n的基本概念。这个案例将清晰地展示现代AI Agent与传统自动化工作流的核心区别。传统流程是线性的，而我们即将构建的Agent将能够接收用户邮件，通过一个核心的**AI Agent**节点进行“思考”，自主理解用户意图，并在多个可用“工具”中进行决策和选择，最终自动生成并发送高度相关的回复。

整个过程模拟了一个更高级的决策逻辑：接收 → AI Agent（思考 → 决策 → 工具调用）→ 回复，如图5.38所示。



图 5.38 一体化智能邮件 Agent 架构示意图

与将工具拆分为多个子工作流的传统方法不同，n8n的AI Agent节点允许我们将组件，例如大语言模型（LLM）、记忆（Memory）、工具（Tools）都整合在一个统一的界面中，极大地简化了构建过程。

整个搭建过程分为两个核心步骤：

1. 准备Agent的“记忆”：创建一个独立的流程，为Agent加载私有知识库。
2. 构建Agent主体：创建接收邮件、思考并回复的主工作流。

5.4.3 构建 Agent 的私有知识库

为了让Agent能够回答关于特定领域（比如您的个人信息或项目文档）的问题，我们需要先为它准备一个“外部大脑”，一个向量知识库。

在n8n中，我们可以使用Simple Vector Store节点在内存中快速构建一个知识库。这个准备流程通常只需要在更新知识时运行一次。

(1) 定义知识源

首先，我们使用Code节点来存放我们的原始知识文本。这是一个简单快捷的方式，实际项目中数据也可以来自文件、数据库等。

- 节点：Code
- 内容：将您的知识以JSON格式写入。



图 5.39 在 Code 节点中定义知识源

```
return [
  {
    "doc_id": "work-schedule-001",
    "content": "我的工作时间是周一至周五，上午9点到下午5点。时区是澳大利亚东部标准时间（AEST）。",
  },
  {
    "doc_id": "off-hours-policy-001",
    "content": "在非工作时间（包括周末和公共假期），我无法立即回复邮件。",
  },
  {
    "doc_id": "auto-reply-instruction-001",
    "content": "如果邮件是在非工作时间收到的，AI助手应该告知发件人，邮件已收到，我会在下一个工作日的9点到5点之间尽快处理并回复。"
  }
];
```

(2) 文本向量化 (Embeddings)

计算机无法直接理解文本，需要将其转换为向量。我们使用Embeddings节点来完成这个“翻译”工作。

- 节点: Embeddings Google Gemini, 选择模型为gemini-embedding-exp-03-07。这里使用Google API来演示, 如果不知道如何获取Google API可以参考[官方文档](#)。
- 配置: 将其连接到 Code 节点之后, 它会自动将上游传入的文本转换为向量数据。



图 5.40 对 Code 中数据进行向量化

(3) 存入向量存储

最后, 我们将向量化的知识存入内存数据库中, 如图5.41所示。

- 节点: Simple Vector Store
- 配置:
 - Operation Mode: Insert Documents (写入模式)。
 - Memory Key: 为这个知识库起一个唯一的名字, 例如 my-dailytime。这个 Key 相当于数据库的“表名”, 后续 Agent 将通过它来查找信息。



图 5.41 对 Code 中数据存入向量存储

完成配置后, 手动执行一次这个流程。成功后, 您的私有知识就加载到 n8n 的内存中了, 如图5.42所示。



图 5.42 完整的知识库加载工作流

5.4.4 创建 Agent 主工作流

有了工具, 我们现在开始构建 Agent 的主要流程。它将负责接收邮件、进行思考和决策, 并在合适的时机调用我们刚刚创建的工具, 最终执行邮件的回复。

(1) 配置 Gmail 触发器

新建一个工作流, 命名为 Agent: Customer Support。使用 Gmail 节点作为触发器, 将其 Event 设置为 Message Received, 并配置好你的邮箱账号。这样, 每当有新邮件进入收件箱时, 该工作流就会被自动触发, 如图5.43所示。



图 5.43 新建Gmail节点图

配置过程可参考[n8n官方文档](#)。Gmail的api在这里[配置](#), 需要创建凭证, 选择Web 应用类型, 最后即得到所需的客户端ID和客户端密钥。并且需要在已获授权的重定向 URI 将n8n刚给的OAuth Redirect URL给添加上。同时, 还需要在[目标对象](#)的Add users加上自己的邮箱地址。最终配置完成的页面如图5.44所示。



图 5.44 Gmail账号加载成功图

现在我们可以点击Fetch Test Event获取邮件了, 如图5.45所示 !



图 5.45 获取实时邮件图

(2) 配置 AI Agent 节点

这是整个工作流的大脑。从节点菜单中拖出一个 AI Agent 节点, 并进行如下配置:

- Chat Model: 连接您选择的大语言模型, 例如 Google Gemini Chat Model。这是 Agent 的“思考核心”。
- Memory: 连接一个 Simple Memory 节点。这能让 Agent 在处理同一邮件线索下的多封往来邮件时, 记住之前的对话历史。
- Tools: 我们可以将多个工具连接到这里。在我们的案例中, 我们连接两个工具:
 1. SerpAPI: 这是我们之前第四章案例中使用过的API, 让 Agent 拥有上网搜索公开信息的能力。
 2. Simple Vector Store: 让 Agent 拥有查询我们第一部分中创建的私有知识库的能力。



图 5.46 AI Agent节点设置图

这是 Agent “思考”的第一步。添加一个 Gemini 节点 (或其他 LLM 节点), 模式设置为 Chat。我们的目标是让它分析邮件内容, 判断用户意图。Prompt 的设计至关重要, 一个清晰的指令能让 LLM 更准确地完成任务。我们将邮件正文和主题 `{{{ $json.snippet }}} {{{ $json.Subject }}}}` 作为变量传入 Prompt 中, 没有API可以到[Google AI Studio](#)点击Get API key创建一个可用的。

其中, 对于AI Agent节点, 我们需要填的主要User Message和System Message部分, 如图5.47所示。



图 5.47 AI Agent 节点详解图

在这里给出我们案例所使用的Prompt:

```

# Prompt (User Message)
# 上下文信息
- 当前时间: {{ new Date().toLocaleString('en-AU', { timeZone: 'Australia/Sydney', hour12: false }) }} (澳大利亚悉尼时间)
- 发件人: {{ $json.From }}
- 主题: {{ $json.Subject }}
- 邮件正文: {{ $json.snippet }}

# System Message
# 角色和目标
你是一个全天候待命、专业高效的AI邮件助手。你的任务是：第一时间使用公开信息尽力回答所有邮件中的问题，并根据我的工作日程，在回复的开头附加上下文状态提醒。

# 上下文信息
- 当前时间: {{ new Date().toLocaleString('en-AU', { timeZone: 'Australia/Sydney', hour12: false }) }} (澳大利亚悉尼时间)
- 邮件信息在输入数据中。

# 可用工具
- Simple Vector Store2: 用来查询我准确的工作时间（例如：周一至周五，上午9点到下午5点）。
- SerpAPI: **[主要信息来源]** 优先使用此工具在互联网上搜索，以回答邮件中的具体问题。

# 执行步骤
1. **分析问题**: 首先，仔细阅读邮件内容，提炼出发件人的核心问题。
2. **并行信息搜集**: 同时执行以下两个操作来收集信息：
   a. 使用 `SerpAPI` 工具，上网搜索出发件人问题的答案。
   b. 使用 `Simple Vector Store2` 工具，获取我设定的准确工作时间。
3. **草拟核心回复**: 根据 `SerpAPI` 搜集到的信息，清晰、直接地回答发件人的问题，这部分将作为邮件回复的主体。
4. **添加状态前缀并整合**:
   a. 对比“当前时间”和我从工具中获取的工作时间。
   b. **如果当前是“非工作时间”**: 创建一段状态提醒前缀。这段前缀必须包含**从 `Simple Vector Store2` 获取到的具体工作时间**。
      * **前缀示例**: “您好，感谢您的来信。您已在我的非工作时间联系我（我的工作时间为：[此处插入查询到的工作时间]）。我会在下一个工作日亲自审阅此邮件。与此同时，这是...”
   c. **如果当前是“工作时间”**: 只需使用简单的问候语即可。
      * **前缀示例**: “您好，关于您提出的问题，答复如下：**<br><br>---<br><br>**”
   d. 将生成的前缀和你草拟的核心回复（第3步的结果）拼接在一起，形成最终的邮件正文。
5. **格式化输出**: 你必须将最终生成的邮件内容以一个严格的 JSON 格式输出。格式如下，不要添加任何额外的解释或文字：
{
  "shouldReply": true,
  "subject": "Re: [原始邮件主题]",
  "body": "[这里是拼接好的、完整的邮件回复正文，**所有换行必须使用HTML的<br>标签**]"
}

# 规则和限制
- **永远优先尝试回答**: 无论何时，你的首要任务是使用 `SerpAPI` 为用户提供有价值的回复。
- **必须声明状态**: 如果在非工作时间回复，必须在邮件开头明确声明，并附上我准确的工作时间。
- **信息来源要准确**: 工作时间必须严格以 `Simple Vector Store2` 的结果为准；问题答案主要来源于 `SerpAPI`，不要编造信息。
- **输出格式**: 在最终输出的JSON中，`body` 字段内的所有换行都必须使用 `  
` 标签，而不是 `\n`。

```

3. 配置 Agent 的工具

对于 Simple Vector Store 工具，我们需要进行关键配置，以确保它能正确“读取”我们之前存入的知识：

- **Operation Mode: Retrieve Documents (As Tool for AI Agent)** (作为工具的读取模式)。
- **Memory Key:** 必须填写与第一部分完全相同的 Key，即 `my_private_knowledge`。
- **Embeddings:** 必须使用与第一部分完全相同的 Embeddings Google Gemini 模型。

只有 Memory Key 和 Embeddings 模型完全一致，Agent 才能用正确的“钥匙”和“语言”来访问知识库，如图 5.48 所示。



图 5.48 Simple Vector Store 工具配置

Description 参数即 AI Agent 调用该工具时，对该工具的描述定义，在这里也给出对应的 Prompt：

这是 Simple Vector Store 工具，用来查询我的个人信息，特别是我的工作时间和邮件回复策略。当需要判断当前是否为工作时间，或者需要告知对方我何时会回复邮件时，必须使用此工具。

对于 Memory 唯一需要注意的是，这里我们使用每个邮箱的线程名作为唯一标识，能保证存储的唯一性，设置的 Key 为 `{} ('Gmail').item.json.threadId {}`

4. 发送最终回复

最后一步是执行。将 AI Agent 节点的输出连接到一个 Gmail 节点，Operation 设为 Send。使用 n8n 表达式，将收件人、主题和正文分别关联到 AI Agent 输出的 JSON 数据中的相应字段，即可实现邮件的自动回复，如图 5.49 所示。

- **To:** {{ ('Gmail').item.json.From }} (或其他触发器中的发件人字段)
- **Subject:** Re: {{ ('Gmail').item.json.Subject }}
- **Message:** {{ \$json.output }}



图 5.49 最终回复工具图示

并且发送成功的同时，也能在个人邮箱收到真实的返回邮件信息，如图5.50所示。



图 5.50 个人邮箱返回邮件格式

至此，一个基于 AI Agent 节点的一体化智能客服就构建完成了，你可以发送一封测试邮件来检验它的工作成果。这个架构的扩展性极强。未来，您可以直接向 AI Agent 节点添加更多的工具（如日历、数据库、CRM 等），只需在 Prompt 中教会 Agent 如何使用它们，就能不断赋予您的 Agent 更强大的能力。

5.4.5 n8n 的优势与局限性分析

通过前面从零到一构建智能邮件助手的实践，我们已经对 n8n 的工作模式有了直观的感受。作为一个强大的低代码自动化平台，n8n 在赋能 Agent 应用开发方面表现出色，但它也并非万能。如表5.1所示，我们将客观地分析其优势与潜在的局限性。

表 5.1 n8n 平台的优势与局限性总结



首先，n8n 最显著的优势在于其开发效率。它将复杂的逻辑抽象为直观的可视化工作流，无论是邮件的接收、AI 的决策，还是工具的调用和最终的回复，整个数据流和处理链路都在画布上一目了然。这种低代码的特性极大地降低了技术门槛，让开发者能够快速搭建和验证 Agent 的核心逻辑，极大地缩短了从想法到原型的距离。

其次，平台的功能强大且高度集成。n8n 拥有丰富的内置节点库，可以轻松连接像 Gmail、Google Gemini 等数百种常见服务。更重要的是，其先进的 AI Agent 节点将模型、记忆和工具管理高度整合，让我们能用一个节点就实现复杂的自主决策，这比传统的多节点手动路由方式要优雅和强大得多。同时，对于内置功能无法覆盖的场景，Code 节点也提供了编写自定义代码的灵活性，保证了功能的上限。

最后，在部署运维层面，n8n 支持私有化部署，并且也是目前相对比较简单且能部署完整版项目的私有化Agent方案，这一点对于注重数据安全和隐私的企业至关重要。我们可以将整个服务部署在自己的服务器上，确保类似内部邮件、客户数据等敏感信息不离开自有环境，这为 Agent 应用的合规性提供了坚实的基础。

当然，每个工具都有其取舍。在享受 n8n 带来便利的同时，我们也必须认识到其局限性。

在开发效率的背后，是调试与错误处理的相对繁琐。当工作流变得复杂时，一旦出现数据格式错误，开发者可能需要逐个节点检查其输入输出定位问题，这有时不如在代码中设置断点来得直接。

功能方面，最大的局限性体现在其内置存储的非持久性。我们在案例中使用的 Simple Memory 和 Simple Vector Store 都是基于内存的，这意味着 n8n 服务一旦重启，所有对话历史和知识库都将丢失。这对于生产环境的应用是致命的。因此，在实际部署时，必须将其替换为如 Redis、Pinecone 等外部持久化数据库，这也会增加了额外的配置和维护成本。

此外，在部署运维和团队协作上，n8n 的版本控制和多人协作不如传统代码成熟。虽然可以将工作流导出为 JSON 文件进行管理，但对比其变更远不如 git diff 代码来得清晰，多人同时编辑同一个工作流也容易产生冲突。

最后是关于性能，n8n 完全能满足绝大多数企业自动化和中低频次的 Agent 任务。但对于需要处理超高并发请求的场景，其节点调度机制可能会带来一定的性能开销，相比于纯代码实现的服务可能稍逊一筹。

5.5 本章小结

本章系统介绍了基于低代码平台构建智能体应用的理念、方法与实践，标志着我们从“手写代码”向“平台化开发”的重要转变。

在第一节中，我们阐述了低代码平台兴起的背景与价值。相比于第四章中纯代码实现的智能体，低代码平台通过图形化、模块化的方式，显著降低了技术门槛、提升了开发效率，并提供了更优的可视化调试体验。这种“更高层次的抽象”让开发者能够将精力聚焦于业务逻辑和提示工程，而非底层实现细节。

随后，我们深入实践了三个各具特色的代表性平台：

Coze 以其零代码的友好体验和丰富的插件生态脱颖而出。通过“每日AI简报”案例，我们体验了如何通过拖拽式配置快速整合多源信息，并一键发布到多个主流平台。Coze 特别适合非技术背景用户和需要快速验证创意的场景，但其不支持 MCP 和无法导出标准化配置文件的局限性也值得注意。

Dify 作为开源的企业级平台，展现了全栈式开发能力。“超级智能体个人助手”案例涵盖了日常问答、文案优化、多模态生成、数据分析和 MCP 工具集成等多个模块，充分展示了 Dify 在复杂业务场景下的强大编排能力。其丰富的插件市场(8000+)、灵活的部署方式和企业级安全特性，使其成为专业开发者和企业团队的理想选择。然而，相对陡峭的学习曲线和在高并发场景下的性能挑战也需要权衡。

n8n 则以其独特的“连接”能力开辟了另一条路径。通过“智能邮件助手”案例，我们看到了如何将 AI 能力无缝嵌入到复杂的业务自动化流程中。n8n 的 AI Agent 节点将模型、记忆和工具高度整合，配合其数百个预置节点，能够实现高度定制化的自动化方案。其支持私有化部署的特性对注重数据安全的企业尤为重要。但内置存储的非持久性和版本控制的不成熟，在生产环境中需要额外的工程化处理。

通过三个平台的对比实践，我们可以得出以下选型建议：- 快速原型验证、非技术用户：优先选择 Coze - 企业级应用、复杂业务逻辑：优先选择 Dify - 深度业务集成、自动化流程：优先选择 n8n

值得强调的是，低代码平台并非要取代代码开发，而是提供了一种互补的选择。在实际项目中，我们完全可以根据不同阶段的需求灵活切换：用低代码平台快速验证想法，用代码实现精细化控制；用平台处理标准化流程，用代码处理特殊逻辑。这种“混合开发”的思维，才是智能体工程化的最佳实践。

下一章，我们将进一步探讨更加底层的智能体框架，帮助读者构建更加可靠、有趣的应用。

习题

1. 本章介绍了三个各具特色的低代码平台：`Coze`、`Dify` 和 `n8n`。请分析：

- 这三个平台在核心定位和设计理念上有什么区别？它们分别解决了智能体开发中的哪些痛点？
- 低代码平台与纯代码开发各有优劣，此外，也有部分功能用平台实现，部分功能用代码实现的“混合开发”模式。思考三种开发模式分别适合哪些场景？请举例说明。

2. 在5.2节的 `Coze` 案例中，我们构建了一个“每日AI简报”智能体。请基于此案例进行扩展思考：

提示：这是一道动手实践题，建议实际操作

- 当前的简报生成是被动触发的（用户主动询问）。如何改造这个智能体，使其能够每天早上8点自动生成简报并推送到指定的飞书群或微信公众号？
- 简报的质量高度依赖于提示词设计。请尝试优化5.2.2节中的提示词，使生成的简报更加专业、结构更清晰，或者增加“热点分析”、“趋势预测”等新功能。
- `Coze` 当前不支持 MCP 协议被认为是一个重要局限（在习题的写作过程中，`feature-mcp` 虽然在 [Coze Studio 04 2025 Product Roadmap](#) 中了，但是还尚未实现）。请简述，什么是 MCP 协议？它为什么重要？如果 `Coze` 未来支持 MCP，会带来哪些新的可能性？

3. 在5.3节的 `Dify` 案例中，我们构建了一个功能全面的“超级智能体个人助手”。请深入分析：

- 案例中使用了“问题分类器”进行智能路由，将不同类型的请求分发到不同的子智能体。这种多智能体架构有什么优势？如果不使用分类器，而是让一个单一的智能体处理所有任务，会遇到什么问题？
- 数据查询模块需要为大模型提供清晰的表结构信息。如果数据库有50张表、每张表有20个字段，直接将所有 DDL 语句放入提示词会导致上下文过长。请设计一个更智能的方案来解决这个问题。
- `Dify` 支持本地部署和云端部署两种模式。请对比这两种模式在数据安全、成本、性能、维护难度等方面差异，并说明各自适用的场景。

4. 在5.4节的 `n8n` 案例中，我们构建了一个“智能邮件助手”。请思考以下问题：

提示：这是一道动手实践题，建议实际操作

- 案例中使用的 `Simple Vector Store` 和 `Simple Memory` 都是基于内存的，服务重启后数据会丢失。请查阅 `n8n` 文档，尝试将其替换为持久化存储方案（如 `Pinecone`、`Redis` 等），并说明配置过程。
- 当前的邮件助手只能处理文本邮件。如果用户发送的邮件中包含附件（如 PDF 文档、图片），你会如何扩展这个工作流，使智能体能够理解附件内容并做出相应回复？
- `n8n` 的核心优势在于“连接”能力。请设计一个更复杂的自动化场景：当客户在电商平台下单后，自动触发一系列操作（发送确认邮件、更新库存数据库、通知物流系统、在 CRM 中记录客户信息）。请画出工作流的节点连接图并说明关键配置。

5. 提示词工程在低代码平台中同样至关重要。本章展示了多个平台的提示词设计案例。请分析：

- 对比5.2.2节（`Coze`）、5.3.2节（`Dify`）和5.4.4节（`n8n`）中的提示词设计，它们在结构、风格和侧重点上有什么不同？这些差异是否与平台特性相关？
- 在 `Dify` 的“文案优化模块”中，提示词要求输出“超过500字”。这种对输出长度的硬性要求是否合理？在什么情况下应该限制输出长度，什么情况下应该让模型自由发挥？

6. 工具和插件是低代码平台的核心能力扩展方式。请思考：

- `Coze` 拥有丰富的插件商店，`Dify` 拥有8000+的插件市场，`n8n` 拥有数百个预置节点。如果这三个平台都没有你需要的某个特定工具（如“连接公司内部系统的 API”），你会如何解决？
- 在5.3.2节中，我们使用了 MCP 协议集成了高德地图、饮食推荐等服务。请调研并说明：MCP 协议与传统的 RESTful API 以及 Tool Calling 有哪些区别？为什么说 MCP 是智能体工具调用的“新标准”？
- 假设你要为 `Dify` 开发一个自定义插件，使其能够调用你公司的内部知识库系统。请查阅 `Dify` 的插件开发文档，概述开发流程和关键技术点。

7. 平台选型是智能体产品成功的关键决策之一。假设你是一家初创公司的技术负责人，公司计划开发以下三个AI应用，请为每个应用选择最合适的平台（`Coze`、`Dify`、`n8n` 或纯代码开发），并详细说明理由：

应用A：面向C端用户的“AI写作助手”小程序，需要快速上线验证市场需求，预算有限，团队中只有1名前端工程师和1名产品经理。

应用B：面向企业客户的“智能合同审核系统”，需要处理敏感的法律文档，要求数据不能离开客户的私有环境，需要与客户现有的OA系统、文档管理系统深度集成。

应用C：内部使用的“研发效能提升工具”，需要自动化处理代码审查、测试报告生成、Bug跟踪、项目进度同步等多个研发流程环节，团队有较强的技术实力。

对于每个应用，请从以下维度（包括但不限于）进行分析：

提示：平台能力是否满足需求，多快能上线，开发成本、运营成本，后续迭代的难度，未来功能扩展的空间

- 技术可行性
- 开发效率
- 成本控制
- 可维护性
- 可扩展性
- 数据安全与合规性

参考文献

[1] Coze - 新一代 AI 应用开发平台. <https://www.coze.cn/>

[2] Dify - 开源的 LLM 应用开发平台. <https://dify.ai/>

[3] n8n - 工作流自动化工具. <https://n8n.io/>

第六章 框架开发实践

在第四章中，我们通过编写原生代码，实现了 ReAct、Plan-and-Solve 和 Reflection 这几种智能体的核心工作流。这个过程让我们对智能体的内在执行逻辑有了理解。随后，在第五章，我们切换到“使用者”的视角，体验了低代码平台带来的便捷与高效。

本章的目标，就是探讨如何利用业界主流的一些智能体框架，来高效、规范地构建可靠的智能体应用。我们将首先概览当前市面上主流的智能体框架，然后并对几个具有代表性的框架，通过一个完整的实战案例，来体验框架驱动的开发模式。

6.1 从手动实现到框架开发

从编写一次性的脚本到使用一个成熟的框架，是软件工程领域一次重要的思维跃迁。我们在第四章中编写的代码，其主要目的是为了教学和理解。它们能很好地完成特定任务，但如果要用它们来构建多个、不同类型且逻辑复杂的智能体应用，很快就会遇到瓶颈。

一个框架的本质，是提供一套经过验证的“规范”。它将所有智能体共有的、重复性的工作（如主循环、状态管理、工具调用、日志记录等）进行抽象和封装，让我们在构建新的智能体时，能够专注于其独特的业务逻辑，而非通用的底层实现。

6.1.1 为何需要智能体框架

在我们开始实战之前，首先需要明确为什么要使用框架。相比于直接编写独立的智能体脚本，使用框架的价值主要体现在以下几个方面：

1. 提升代码复用与开发效率：这是最直接的价值。一个好的框架会提供一个通用的 Agent 基类或执行器，它封装了智能体运行的核心循环（Agent Loop）。无论是 ReAct 还是 Plan-and-Solve，都可以基于框架提供的标准组件快速搭建，从而避免重复劳动。
2. 实现核心组件的解耦与可扩展性：一个健壮的智能体系统应该由多个松散耦合的模块组成。框架的设计会强制我们分离不同的关注点：
 - 模型层 (Model Layer)：负责与大语言模型交互，可以轻松替换不同的模型（OpenAI, Anthropic, 本地模型）。
 - 工具层 (Tool Layer)：提供标准化的工具定义、注册和执行接口，添加新工具不会影响其他代码。
 - 记忆层 (Memory Layer)：处理短期和长期记忆，可以根据需求切换不同的记忆策略（如滑动窗口、摘要记忆）。这种模块化的设计使得整个系统极具可扩展性，更换或升级任何一个组件都变得简单。
3. 标准化复杂的状态管理：我们在 ReflectionAgent 中实现的 Memory 类只是一个简单的开始。在真实的、长时运行的智能体应用中，状态管理是一个巨大的挑战，它需要处理上下文窗口限制、历史信息持久化、多轮对话状态跟踪等问题。一个框架可以提供一套强大而通用的状态管理机制，开发者无需每次都重新处理这些复杂问题。
4. 简化可观测性与调试过程：当智能体的行为变得复杂时，理解其决策过程变得至关重要。一个精心设计的框架可以内置强大的可观测性能力。例如，通过引入事件回调机制 (Callbacks)，我们可以在智能体生命周期的关键节点（如 `on_llm_start`, `on_tool_end`, `on_agent_finish`）自动触发日志记录或数据上报，从而轻松地追踪和调试智能体的完整运行轨迹。这远比在代码中手动添加 `print` 语句要高效和系统化。

因此，从手动实现走向框架开发，不仅是代码组织方式的改变，更是构建复杂、可靠、可维护的智能体应用的必由之路。

6.1.2 主流框架的选型与对比

智能体框架的生态正在以前所未有的速度发展。如果说 LangChain 和 LlamaIndex 定义了第一代通用 LLM 应用框架的范式，那么新一代的框架则更加专注于解决特定领域的深层挑战，尤其是多智能体协作 (Multi-Agent Collaboration) 和复杂工作流控制 (Complex Workflow Control)。

在本章的后续实战中，我们将聚焦于四个在这些前沿领域极具代表性的框架：AutoGen、AgentScope、CAMEL 和 LangGraph。它们的设计理念各不相同，分别代表了实现复杂智能体系统的不同技术路径，如表6.1所示。

表 6.1 四种智能体框架对比

|--|--|

- **AutoGen**：AutoGen 的核心思想是通过对话实现协作^[1]。它将多智能体系统抽象为一个由多个“可对话”智能体组成的群聊。开发者可以定义不同角色（如 Coder, ProductManager, Tester），并设定它们之间的交互规则（例如，Coder 写完代码后由 Tester 自动接管）。任务的解决过程，就是这些智能体在群聊中通过自动化消息传递，不断对话、协作、迭代直至最终目标达成的过程。
- **AgentScope**：AgentScope 是一个专为多智能体应用设计的、功能全面的开发平台^[2]。它的核心特点是易用性和工程化。它提供了一套非常友好的编程接口，让开发者可以轻松定义智能体、构建通信网络，并管理整个应用的生命周期。其内置的消息传递机制和对分布式部署的支持，使其非常适合构建和运维复杂、大规模的多智能体系统。
- **CAMEL**：CAMEL 提供了一种新颖的、名为角色扮演 (Role-Playing) 的协作方法^[3]。其核心理念是，我们只需要为两个智能体（例如，AI研究员和 Python程序员）设定好各自的角色和共同的任务目标，它们就能在“初始提示 (Inception Prompting)”的引导下，自主地进行多轮对话，相互启发、相互配合，共同完成任务。它极大地降低了设计多智能体对话流程的复杂度。
- **LangGraph**：作为 LangChain 生态的扩展，LangGraph 另辟蹊径，将智能体的执行流程建模为图 (Graph)^[4]。在传统的链式结构中，信息只能单向流动。而 LangGraph 将每一步操作（如调用LLM、执行工具）定义为图中的一个节点 (Node)，并用边 (Edge) 来定义节点之间的跳转逻辑。这种设计天然支持循环 (Cycles)，使得实现如 Reflection 这样的迭代、修正、自我反思的复杂工作流变得异常简单和直观。

在接下来的小节中，我们将对这四个框架，分别通过一个完整的实战案例，来深入体验框架驱动的开发模式。请注意，所有演示的项目源文件会放在code文件夹下，正文内只讲解原理部分。

6.2 框架一：AutoGen

正如前文所述，AutoGen 的设计哲学根植于“以对话驱动协作”。它巧妙地将复杂的任务解决流程，映射为不同角色的智能体之间的一系列自动化对话。基于这一核心理念，AutoGen 框架持续演进。我们将以 0.7.4 版本为例，因为它是截止目前为止最新版本，代表了一次重要的架构重构，从类继承设计转向了更灵活的组合式架构。为了深入理解并应用这一框架，我们首先需要讲解其最核心的构成要素与底层的对话交互机制。

6.2.1 AutoGen 的核心机制

0.7.4 版本的发布是 AutoGen 发展的一个重要节点，它标志着框架在底层设计上的一次根本性革新。这次更新并非简单的功能叠加，而是对整体架构的重新思考，旨在提升框架的模块化、并发性能和开发者体验。



图 6.1 AutoGen 架构图

(1) 框架结构的演进

如图6.1所示，新架构最显著的变化是引入了清晰的分层和异步优先的设计理念。

- 分层设计：框架被拆分为两个核心模块：
 - **autogen-core**: 作为框架的底层基础，封装了与语言模型交互、消息传递等核心功能。它的存在保证了框架的稳定性和未来扩展性。
 - **autogen-agentchat**: 构建于 **core** 之上，提供了用于开发对话式智能体应用的高级接口，简化了多智能体应用的开发流程。这种分层策略使得各组件职责明确，降低了系统的耦合度。
- 异步优先：新架构全面转向异步编程 (`async/await`)。在多智能体协作场景中，网络请求（如调用 LLM API）是主要耗时操作。异步模式允许系统在等待一个智能体响应时处理其他任务，从而避免了线程阻塞，显著提升了并发处理能力和系统资源的利用效率。

(2) 核心智能体组件

智能体是执行任务的基本单元。在 0.7.4 版本中，智能体的设计更加专注和模块化。

- **AssistantAgent** (助理智能体)：这是任务的主要解决者，其核心是封装了一个大型语言模型 (LLM)。它的职责是根据对话历史生成富有逻辑和知识的回复，例如提出计划、撰写文章或编写代码。通过不同的系统消息 (System Message)，我们可以为其赋予不同的“专家”角色。
- **UserProxyAgent** (用户代理智能体)：这是 AutoGen 中功能独特的组件。它扮演着双重角色：既是人类用户的“代言人”，负责发起任务和传达意图；又是一个可靠的“执行器”，可以配置为执行代码或调用工具，并将结果反馈给其他智能体。这种设计清晰地区分了“思考”（由 **AssistantAgent** 完成）与“行动”。

(3) 从 GroupChatManager 到 Team

当任务需要多个智能体协作时，就需要一个机制来协调对话流程。在早期版本中，**GroupChatManager** 承担了这一职责。而在新架构中，引入了更灵活的 **Team** 或群聊概念，例如 **RoundRobinGroupChat**。

- 轮询群聊 (**RoundRobinGroupChat**)：这是一种明确的、顺序化的对话协调机制。它会让参与的智能体按照预定义的顺序依次发言。这种模式非常适用于流程固定的任务，例如一个典型的软件开发流程：产品经理先提出需求，然后工程师编写代码，最后由代码审查员进行检查。
- 工作流：
 1. 首先，创建一个 **RoundRobinGroupChat** 实例，并将所有参与协作的智能体（如产品经理、工程师等）加入其中。
 2. 当一个任务开始时，群聊会按照预设的顺序，依次激活相应的智能体。
 3. 被选中的智能体根据当前的对话上下文进行响应。
 4. 群聊将新的回复加入对话历史，并激活下一个智能体。
 5. 这个过程会持续进行，直到达到最大对话轮次或满足预设的终止条件。

通过这种方式，AutoGen 将复杂的协作关系，简化为一个流程清晰、易于管理的自动化“圆桌会议”。开发者只需定义好每个团队成员的角色和发言顺序，剩下的协作流程便可由群聊机制自主驱动。

在下一节中，我们将通过构建一个模拟软件开发团队的实例，来亲身体验如何在新架构下定义不同角色的智能体，并将它们组织在一个由 **RoundRobinGroupChat** 协调的群聊中，以协作完成一个真实的编程任务。

6.2.2 软件开发团队

在理解了 AutoGen 的核心组件与对话机制后，本节将通过一个完整的实战案例来具体展示如何应用这些新特性。我们将构建一个模拟的软件开发团队，该团队由多个具有不同专业技能的智能体组成，它们将协作完成一个真实的软件开发任务。

(1) 业务目标

我们的目标是开发一个功能明确的 Web 应用：实时显示比特币当前价格。这个任务虽小，却完整地覆盖了软件开发的典型环节：从需求分析、技术选型、编码实现到代码审查和最终测试。这使其成为检验 AutoGen 自动化协作流程的理想场景。

(2) 智能体团队角色

为了模拟真实的软件开发流程，我们设计了四个职责分明的智能体角色：

- **ProductManager** (产品经理)：负责将用户的模糊需求转化为清晰、可执行的开发计划。
- **Engineer** (工程师)：依据开发计划，负责编写具体的应用程序代码。
- **CodeReviewer** (代码审查员)：负责审查工程师提交的代码，确保其质量、可读性和健壮性。
- **UserProxy** (用户代理)：代表最终用户，发起初始任务，并负责执行和验证最终交付的代码。

这种角色划分是多智能体系统设计中的关键一步，它将一个复杂任务分解为多个由领域“专家”处理的子任务。

6.2.3 核心代码实现

下面，我们将分步解析这个自动化团队的核心代码。

(1) 模型客户端配置

所有基于 LLM 的智能体都需要一个模型客户端来与语言模型进行交互。AutoGen 0.7.4 提供了标准化的 `OpenAIChatCompletionClient`，它可以方便地与任何兼容 OpenAI API 规范的模型服务（包括 OpenAI 官方服务、Azure OpenAI 以及本地模型服务如 Ollama 等）进行对接。

我们通过一个独立的函数来创建和配置模型客户端，并通过环境变量管理 API Key 和服务地址，这是一种良好的工程实践，增强了代码的灵活性和安全性。

```
from autogen_ext.models.openai import OpenAIChatCompletionClient

def create_openai_model_client():
    """创建并配置 OpenAI 模型客户端"""
    return OpenAIChatCompletionClient(
        model=os.getenv("LLM_MODEL_ID", "gpt-4o"),
        api_key=os.getenv("LLM_API_KEY"),
        base_url=os.getenv("LLM_BASE_URL", "https://api.openai.com/v1")
    )
```

(2) 智能体角色的定义

定义智能体的核心在于编写高质量的系统消息 (System Message)。系统消息就像是给智能体设定的“行为准则”和“专业知识库”，它精确地规定了智能体的角色、职责、工作流程，甚至是与其他智能体交互的方式。一个精心设计的系统消息是确保多智能体系统能够高效、准确协作的关键。在我们的软件开发团队中，我们为每一个角色都创建了一个独立的函数来封装其定义。

产品经理 (ProductManager)

产品经理负责启动整个流程。它的系统消息不仅定义了其职责，还规范了其输出的结构，并包含了引导对话转向下一环节（工程师）的明确指令。

```
def create_product_manager(model_client):
    """创建产品经理智能体"""
    system_message = """你是一位经验丰富的产品经理，专门负责软件产品的需求分析和项目规划。
```

你的核心职责包括：

1. **需求分析**：深入理解用户需求，识别核心功能和边界条件
2. **技术规划**：基于需求制定清晰的技术实现路径
3. **风险评估**：识别潜在的技术风险和用户体验问题
4. **协调沟通**：与工程师和其他团队成员进行有效沟通

当接到开发任务时，请按以下结构进行分析：

1. 需求理解与分析
2. 功能模块划分
3. 技术选型建议
4. 实现优先级排序
5. 验收标准定义

请简洁明了地回应，并在分析完成后说“请工程师开始实现”。”””

```
return AssistantAgent(
    name="ProductManager",
    model_client=model_client,
    system_message=system_message,
)
```

工程师 (Engineer)

工程师的系统消息聚焦于技术实现。它列举了工程师的技术专长，并规定了其在接收到任务后的具体行动步骤，同样也包含了引导流程转向代码审查员的指令。

```
def create_engineer(model_client):
    """创建软件工程师智能体"""
    system_message = """你是一位资深的软件工程师，擅长 Python 开发和 Web 应用构建。
```

你的技术专长包括：

1. **Python 编程**：熟练掌握 Python 语法和最佳实践
2. **Web 开发**：精通 Streamlit、Flask、Django 等框架
3. **API 集成**：有丰富的第三方 API 集成经验
4. **错误处理**：注重代码的健壮性和异常处理

当收到开发任务时，请：

1. 仔细分析技术需求
2. 选择合适的技术方案
3. 编写完整的代码实现
4. 添加必要的注释和说明
5. 考虑边界情况和异常处理

请提供完整的可运行代码，并在完成后说“请代码审查员检查”。”””

```
return AssistantAgent(
```

```
        name="Engineer",
        model_client=model_client,
        system_message=system_message,
    )
```

代码审查员 (CodeReviewer)

代码审查员的定义则侧重于代码的质量、安全性和规范性。它的系统消息详细列出了审查的重点和流程，确保了代码交付前的质量关卡。

```
def create_code_reviewer(model_client):
    """创建代码审查员智能体"""
    system_message = """你是一位经验丰富的代码审查专家，专注于代码质量和最佳实践。
```

你的审查重点包括：

1. **代码质量**：检查代码的可读性、可维护性和性能
2. **安全性**：识别潜在的安全漏洞和风险点
3. **最佳实践**：确保代码遵循行业标准和最佳实践
4. **错误处理**：验证异常处理的完整性和合理性

审查流程：

1. 仔细阅读和理解代码逻辑
2. 检查代码规范和最佳实践
3. 识别潜在问题和改进点
4. 提供具体的修改建议
5. 评估代码的整体质量

请提供具体的审查意见，完成后说“代码审查完成，请用户代理测试”。”””

```
return AssistantAgent(
    name="CodeReviewer",
    model_client=model_client,
    system_message=system_message,
)
```

用户代理 (UserProxy)

UserProxyAgent 是一个特殊的智能体，它不依赖 LLM 进行回复，而是作为用户在系统中的代理。它的 `description` 字段清晰地描述了其职责，尤其重要的是，它负责在任务最终完成后发出 `TERMINATE` 指令，以正常结束整个协作流程。

```
def create_user_proxy():
    """创建用户代理智能体"""
    return UserProxyAgent(
        name="UserProxy",
        description="""用户代理，负责以下职责：
1. 代表用户提出开发需求
2. 执行最终的代码实现
3. 验证功能是否符合预期
4. 提供用户反馈和建议
完成测试后请回复 TERMINATE。""",
```

通过这四个独立的定义函数，我们不仅构建了一支功能完备的“虚拟团队”，也展示了通过系统消息进行“提示工程”，是设计高效多智能体应用的核心环节。

(3) 定义团队协作流程

在本案例中，软件开发的流程是相对固定的（需求->编码->审查->测试），因此 `RoundRobinGroupChat` (轮询群聊) 是理想的选择。我们按照业务逻辑顺序，将四个智能体加入到参与者列表中。

```
from autogen_agentchat.teams import RoundRobinGroupChat
from autogen_agentchat.conditions import TextMentionTermination

# 定义团队聊天和协作规则
team_chat = RoundRobinGroupChat(
    participants=[
        product_manager,
        engineer,
        code_reviewer,
        user_proxy
    ],
    termination_condition=TextMentionTermination("TERMINATE"),
    max_turns=20,
)
```

- 参与者顺序：`participants` 列表的顺序决定了智能体发言的先后次序。
- 终止条件：`termination_condition` 是控制协作流程何时结束的关键。这里我们设定，当任何消息中包含关键词 “TERMINATE” 时，对话便结束。在我们的设计中，这个指令由 `UserProxy` 在完成最终测试后发出。
- 最大轮次：`max_turns` 是一个安全阀，用于防止对话陷入无限循环，避免不必要的资源消耗。

(4) 启动与运行

由于 AutoGen 0.7.4 采用异步架构，整个协作流程的启动和运行都在一个异步函数中完成，并最终通过 `asyncio.run()` 来执行。

```
async def run_software_development_team():
    # ... 初始化客户端和智能体 ...
```

```
# 定义任务描述
task = """我们需要开发一个比特币价格显示应用，具体要求如下：
核心功能：
- 实时显示比特币当前价格（USD）
- 显示24小时价格变化趋势（涨幅和涨跌额）
- 提供价格刷新功能

技术要求：
- 使用 Streamlit 框架创建 Web 应用
- 界面简洁美观，用户友好
- 添加适当的错误处理和加载状态

请团队协作完成这个任务，从需求分析到最终实现。"""

# 异步执行团队协作，并流式输出对话过程
result = await Console(team_chat.run_stream(task=task))
return result
```

```
# 主程序入口
if __name__ == "__main__":
    result = asyncio.run(run_software_development_team())
```

当程序运行时，task 作为初始消息被传入 team_chat，产品经理作为第一个参与者接收到该消息，随后整个自动化协作流程便开始了。

(5) 预期协作效果

当我们运行这个软件开发团队时，可以观察到一个完整的协作流程：

```
🔧 正在初始化模型客户端...
👤 正在创建智能体团队...
🚀 启动 AutoGen 软件开发团队协作...
=====
----- TextMessage (user) -----
我们需要开发一个比特币价格显示应用，具体要求如下：
...
请团队协作完成这个任务，从需求分析到最终实现。
----- TextMessage (ProductManager) -----
### 1. 需求理解与分析
...
请工程师开始实现。
----- TextMessage (Engineer) -----
### 技术方案实施
...
请代码审查员检查。
----- TextMessage (CodeReviewer) -----
### 代码审查
...
代码审查完成，请用户代理测试。
----- TextMessage (UserProxy) -----
已经完成需求
----- TextMessage (ProductManager) -----
太好了，感谢您的反馈！如果在使用过程中有任何问题，或者有其他功能需求和改进建议，请随时告知我们。我们会持续提供支持和改进。期待您对我们的应用有愉快的使用体验！
----- TextMessage (Engineer) -----
很高兴听到项目顺利完成。如果您或用户有任何问题或者需要帮助，请随时联系我们。感谢您对我们工作的支持，让我们一起确保应用稳定运行并不断优化用户体验！
----- TextMessage (CodeReviewer) -----
非常感谢大家的努力与协作，使得项目能够顺利完成。未来若有更多技术支持的需求或者需要改进的地方，我们愿意为项目的持续优化贡献力量。期待用户能够享受到流畅的体验，同时也欢迎提出更多的反馈与建议。再次感谢团队的合作！
----- TextMessage (UserProxy) -----
Enter your response: TERMINATE
=====
✅ 团队协作完成！
```

协作结果摘要：
- 参与智能体数量：4个
- 任务完成状态：成功

整个协作过程展现了 AutoGen 框架的优势：自然的对话驱动协作、角色专业化分工、流程自动化管理和完整的开发闭环。

6.2.4 AutoGen 的优势与局限性分析

任何技术框架都有其特定的适用场景和设计权衡。在本节中，我们将客观地分析 AutoGen 的核心优势及其在实际应用中可能面临的局限性与挑战。

(1) 优势

- 如案例所示，我们无需为智能体团队设计复杂的状态机或控制流逻辑，而是将一个完整的软件开发流程，自然地映射为产品经理、工程师和审查员之间的对话。这种方式更贴近人类团队的协作模式，显著降低了为复杂任务建模的门槛。开发者可以将更多精力聚焦于定义“谁（角色）”以及“做什么（职责）”，而非“如何做（流程控制）”。
- 框架允许通过系统消息（System Message）为每个智能体赋予高度专业化的角色。在案例中，ProductManager 专注于需求，而 CodeReviewer 则专注于质量。一个精心设计的智能体可以在不同项目中被复用，易于维护和扩展。

- 对于流程化任务，RoundRobinGroupChat 这样机制提供了清晰、可预测的协作流程。同时，UserProxyAgent 的设计为“人类在环”（Human-in-the-loop）提供了天然的接口。它既可以作为任务的发起者，也可以是流程的监督者和最终的验收者。这种设计确保了自动化系统始终处于人类的监督之下。

(2) 局限性

- 虽然 RoundRobinGroupChat 提供了顺序化的流程，但基于 LLM 的对话本质上具有不确定性。智能体可能会产生偏离预期的回复，导致对话走向意外的分支，甚至陷入循环。
- 当智能体团队的工作结果未达预期时，调试过程可能非常棘手。与传统程序不同，我们得到的不是清晰的错误堆栈，而是一长串的对话历史。这被称为“对话式调试”的难题。

(3) 非 OpenAI 模型的配置补充

如果你想使用非 OpenAI 系列的模型（如 DeepSeek、通义千问等），在 0.7.4 版本中需要在 OpenAIChatCompletionClient 的参数中传入模型信息字典。以 DeepSeek 为例：

```
from autogen_ext.models.openai import OpenAIChatCompletionClient

model_client = OpenAIChatCompletionClient(
    model="deepseek-chat",
    api_key=os.getenv("DEEPESEEK_API_KEY"),
    base_url="https://api.deepseek.com/v1",
    model_info={
        "function_calling": True,
        "max_tokens": 4096,
        "context_length": 32768,
        "vision": False,
        "json_output": True,
        "family": "deepseek",
        "structured_output": True,
    }
)
```

这个 model_info 字典帮助 AutoGen 了解模型的能力边界，从而更好地适配不同的模型服务。

6.3 框架二：AgentScope

如果说 AutoGen 的设计哲学是“以对话驱动协作”，那么 AgentScope 则代表了另一种技术路径：工程化优先的多智能体平台。AgentScope 由阿里巴巴达摩院开发，专门为构建大规模、高可靠的多智能体应用而设计。它不仅提供了直观易用的编程接口，更重要的是内置了分布式部署、容错恢复、可观测性等企业级特性，使其特别适合构建需要长期稳定运行的生产环境应用。

6.3.1 AgentScope 的设计

与 AutoGen 相比，AgentScope 的核心差异在于其消息驱动的架构设计和工业级的工程实践。如果说 AutoGen 更像是一个灵活的“对话工作室”，那么 AgentScope 就是一个完整的“智能体操作系统”，为开发者提供了从开发、测试到部署的全生命周期支持。与许多框架采用的继承式设计不同，AgentScope 选择了组合式架构和消息驱动模式。这种设计不仅增强了系统的模块化程度，也为其实现的并发性能和分布式能力奠定了基础。

(1) 分层架构体系

如图6.2所示，AgentScope 采用了清晰的分层模块化设计，从底层的基础组件到上层的应用编排，形成了一个完整的智能体开发生态。



图 6.2 AgentScope 架构图

在这个架构中，最底层是基础组件层 (**Foundational Components**)，它为整个框架提供了核心的构建块。Message 组件定义了统一的消息格式，支持从简单的文本交互到复杂的多模态内容；Memory 组件提供了短期和长期记忆管理；Model API 层抽象了对不同大语言模型的调用；而 Tool 组件则封装了智能体与外部世界交互的能力。

在基础组件之上，智能体基础设施层 (**Agent-level Infrastructure**) 提供了更高级的抽象。这一层不仅包含了各种预构建的智能体（如浏览器使用智能体、深度研究智能体），还实现了经典的 ReAct 范式，支持智能体钩子、并行工具调用、状态管理等高级特性。特别值得注意的是，这一层原生支持异步执行与实时控制，这是 AgentScope 相比其他框架的一个重要优势。

多智能体协作层 (**Multi-Agent Cooperation**) 是 AgentScope 的核心创新所在。MsgHub 作为消息中心，负责智能体间的消息路由和状态管理；而 Pipeline 系统则提供了灵活的工作流编排能力，支持顺序、并发等多种执行模式。这种设计使得开发者可以轻松构建复杂的多智能体协作场景。

最上层的开发与部署层 (**Deployment & Development**) 则体现了 AgentScope 对工程化的重视。AgentScope Runtime 提供了生产级的运行时环境，而 AgentScope Studio 则为开发者提供了完整的可视化开发工具链。

(2) 消息驱动

AgentScope 的核心创新在于其消息驱动架构。在这个架构中，所有的智能体交互都被抽象为消息的发送和接收，而不是传统的函数调用。

```
from agentscope.message import Msg

# 消息的标准结构
message = Msg(
    name="Alice",           # 发送者名称
    content="Hello, Bob!",   # 消息内容
)
file:///Users/knight/workspace/hello-agents/docs/Hello-Agents-Full.html
```

```

        role="user",          # 角色类型
        metadata={            # 元数据信息
            "timestamp": "2024-01-15T10:30:00Z",
            "message_type": "text",
            "priority": "normal"
        }
    )
)

```

将消息作为交互的基础单元，带来了几个关键优势：

- 异步解耦：消息的发送方和接收方在时间上解耦，无需相互等待，天然支持高并发场景。
- 位置透明：智能体无需关心另一个智能体是在本地进程还是在远程服务器上，消息系统会自动处理路由。
- 可观测性：每一条消息都可以被记录、追踪和分析，极大地简化了复杂系统的调试与监控。
- 可靠性：消息可以被持久化存储和重试，即使系统出现故障，也能保证交互的最终一致性，提升了系统的容错能力。

(3) 智能体生命周期管理

在 AgentScope 中，每个智能体都有明确的生命周期（初始化、运行、暂停、销毁等），并基于一个统一的基类 `AgentBase` 来实现。开发者通常只需要关注其核心的 `reply` 方法。

```

from agentscope.agents import AgentBase

class CustomAgent(AgentBase):
    def __init__(self, name: str, **kwargs):
        super().__init__(name=name, **kwargs)
        # 智能体初始化逻辑

    def reply(self, x: Msg) -> Msg:
        # 智能体的核心响应逻辑
        response = self.model(x.content)
        return Msg(name=self.name, content=response, role="assistant")

    def observe(self, x: Msg) -> None:
        # 智能体的观察逻辑（可选）
        self.memory.add(x)
)

```

这种设计模式分离了智能体的内部逻辑与外部通信，开发者只需在 `reply` 方法中定义智能体“思考和回应”的方式即可。

(4) 消息传递机制

AgentScope 内置了一个消息中心 (**MsgHub**)，它是整个消息驱动架构的中枢。MsgHub 不仅负责消息的路由和分发，还集成了持久化和分布式通信等高级功能，它有以下这些特点。

- 灵活的消息路由：支持点对点、广播、组播等多种通信模式，可以构建灵活复杂的交互网络。
- 消息持久化：能够将所有消息自动保存到数据库（如 SQLite, MongoDB），确保了长期运行任务的状态可以被恢复。
- 原生分布式支持：这是 AgentScope 的标志性特性。智能体可以被部署在不同的进程或服务器上，MsgHub 会通过 RPC（远程过程调用）自动处理跨节点的通信，对开发者完全透明。

这些由底层架构提供的工程化能力，使得 AgentScope 在处理需要高并发、高可靠性的复杂应用场景时，比传统的对话驱动框架更具优势。当然，这也要求开发者理解并适应消息驱动的异步编程范式。

在下一节中，我们将通过一个具体的实战案例，三国狼人杀游戏，来深入体验 AgentScope 框架的能力，特别是其在处理并发交互方面的优势。

6.3.2 三国狼人杀游戏

为了深入理解 AgentScope 的消息驱动架构和多智能体协作能力，我们将构建一个融合了中国古典文化元素的“三国狼人杀”游戏。这个案例不仅展示了 AgentScope 在处理复杂多智能体交互方面的优势，更重要的是，它演示了如何在一个需要实时协作、角色扮演和策略博弈的场景中，充分发挥消息驱动架构的威力。与传统狼人杀不同，我们的“三国狼人杀”将刘备、关羽、诸葛亮等经典角色引入游戏，每个智能体不仅要完成狼人杀的基本任务（如狼人击杀、预言家查验、村民推理），还要体现出对应三国人物的性格特点和行为模式。这种设计让我们能够观察到 AgentScope 在处理多层次角色建模方面的表现。

(1) 架构设计与核心组件

本案例的系统设计遵循了分层解耦的原则，将游戏逻辑划分为三个独立的层次，每个层次都映射了 AgentScope 的一个或多个核心组件：

- 游戏控制层 (**Game Control Layer**)：由一个 `ThreeKingdomsWerewolfGame` 类作为游戏的主控制器，负责维护全局状态（如玩家存活列表、当前游戏阶段）、推进游戏流程（调用夜晚阶段、白天阶段）以及裁定胜负。
- 智能体交互层 (**Agent Interaction Layer**)：完全由 MsgHub 驱动。所有智能体间的通信，无论是狼人间的秘密协商，还是白天的公开辩论，都通过消息中心进行路由和分发。
- 角色建模层 (**Role Modeling Layer**)：每个玩家都是一个基于 `DialogAgent` 的实例。我们通过精心设计的系统提示词，为每个智能体注入了“游戏角色”和“三国人格”的双重身份。

(2) 消息驱动的游戏流程

本案例最核心的设计是以消息驱动代替状态机来管理游戏流程。在传统实现中，游戏阶段的转换通常由一个中心化的状态机 (State Machine) 控制。而在 AgentScope 的范式下，游戏流程被自然地建模为一系列定义好的消息交互模式。

例如，狼人阶段的实现，并非一个简单的函数调用，而是通过 MsgHub 动态创建一个临时的、仅包含狼人玩家的私密通信频道：

```

async def werewolf_phase(self, round_num: int):
    """狼人阶段 - 展示消息驱动的协作模式"""
    if not self.werewolves:
        return None

    # 通过消息中心建立狼人专属通信频道
    async with MsgHub(
        self.werewolves,
        enable_auto_broadcast=True,
        announcement=await self.moderator.announce(
            f"狼人们，请讨论今晚的击杀目标。存活玩家: {format_player_list(self.alive_players)}"
        ),
    ) as werewolves_hub:
        # 讨论阶段: 狼人通过消息交换策略
        for _ in range(MAX_DISCUSSION_ROUND):
            for wolf in self.werewolves:
                await wolf(structured_model=DiscussionModelCN)

    # 投票阶段: 收集并统计狼人的击杀决策
    werewolves_hub.set_auto_broadcast(False)
    kill_votes = await fanout_pipeline(
        self.werewolves,
        msg=await self.moderator.announce("请选择击杀目标"),
        structured_model=WerewolfKillModelCN,
        enable_gather=False,
    )

```

这种设计的优势在于，游戏逻辑被清晰地表达为“在特定上下文中，以何种模式进行消息交换”，而不是一连串僵硬的状态转换。白天讨论（全员广播）、预言家查验（点对点请求）等阶段也都遵循同样的设计范式。

(3) 用结构化输出约束游戏规则

狼人杀游戏的一个关键挑战是如何确保智能体的行为符合游戏规则。AgentScope 的结构化输出机制为这个问题提供了解决方案。我们为不同的游戏行为定义了严格的数据模型：

```

class DiscussionModelCN(BaseModel):
    """讨论阶段的输出格式"""
    reach_agreement: bool = Field(
        description="是否已达成一致意见",
        default=False
    )
    confidence_level: int = Field(
        description="对当前推理的信心程度(1-10)",
        ge=1, le=10,
        default=5
    )
    key_evidence: Optional[str] = Field(
        description="支持你观点的关键证据",
        default=None
    )

class WitchActionModelCN(BaseModel):
    """女巫行动的输出格式"""
    use_antidote: bool = Field(description="是否使用解药")
    use_poison: bool = Field(description="是否使用毒药")
    target_name: Optional[str] = Field(description="毒药目标玩家姓名")

```

通过这种方式，我们不仅确保了智能体输出的格式一致性，更重要的是实现了游戏规则的自动化约束。例如，女巫智能体无法同时对同一目标使用解药和毒药，预言家每晚只能查验一名玩家，这些约束都通过数据模型的字段定义和验证逻辑自动执行。

(4) 角色建模的双重挑战

在这个案例中，最有趣的技术挑战是如何让智能体同时扮演好两个层面的角色：游戏功能角色（狼人、预言家等）和文化人格角色（刘备、曹操等）。我们通过提示词工程来解决这个问题：

```

def get_role_prompt(role: str, character: str) -> str:
    """获取角色提示词 - 融合游戏规则与人物性格"""
    base_prompt = f"""你是{character}，在这场三国狼人杀游戏中扮演{role}。

```

重要规则：

1. 你只能通过对话和推理参与游戏
2. 不要尝试调用任何外部工具或函数
3. 严格按照要求的JSON格式回复

角色特点：

```

"""
if role == "狼人":
    return base_prompt + """
- 你是狼人阵营，目标是消灭所有好人
- 夜晚可以与其他狼人协商击杀目标
- 白天要隐藏身份，误导好人
- 以{character}的性格说话和行动
"""

```

这种设计让我们观察到了一个有趣的现象：不同的三国人物在扮演相同游戏角色时，会表现出截然不同的策略和话语风格。例如，扮演狼人的“曹操”可能会表现得更加狡猾和善于伪装，而扮演狼人的“张飞”则可能显得更加直接和冲动。

(5) 并发处理与容错机制

AgentScope 的异步架构在这个多智能体游戏中发挥了重要作用。游戏中经常出现需要同时收集多个智能体决策的场景，比如投票阶段：

```
# 并行收集所有玩家的投票决策
vote_msgs = await fanout_pipeline(
    self.alive_players,
    await self.moderator.announce("请投票选择要淘汰的玩家"),
    structured_model=get_vote_model_cn(self.alive_players),
    enable_gather=False,
)
```

`fanout_pipeline` 允许我们并行地向所有智能体发送相同的消息，并异步收集它们的响应。这不仅提高了游戏的执行效率，更重要的是模拟了真实狼人杀游戏中“同时投票”的场景。同时，我们在关键环节加入了容错处理：

```
try:
    response = await wolf(
        "请分析当前局势并表达你的观点。",
        structured_model=DiscussionModelCN
    )
except Exception as e:
    print(f"⚠️ {wolf.name} 讨论时出错: {e}")
# 创建默认响应，确保游戏继续进行
default_response = DiscussionModelCN(
    reach_agreement=False,
    confidence_level=5,
    key_evidence="暂时无法分析"
)
```

这种设计确保了即使某个智能体出现异常，整个游戏流程也能继续进行。

(6) 案例输出与总结

为了更直观地感受 AgentScope 的运行机制，以下是一段截取自游戏夜晚阶段的真实运行日志，展示了扮演“孙权”和“周瑜”的两个狼人智能体进行秘密协商并执行击杀的过程。

🎮 欢迎来到三国狼人杀！

==== 游戏初始化 ===

游戏主持人: 🗡️ 【孙权】你在这场三国狼人杀中扮演狼人，你的角色是孙权。夜晚可以击杀一名玩家

游戏主持人: 🗡️ 【周瑜】你在这场三国狼人杀中扮演狼人，你的角色是周瑜。夜晚可以击杀一名玩家

...

游戏主持人: 🗡️ 三国狼人杀游戏开始！参与者：孙权、周瑜、曹操、张飞、司马懿、赵云

✅ 游戏设置完成，共6名玩家

==== 第1轮游戏 ===

🌙 第1夜降临，天黑请闭眼...

【狼人阶段】

游戏主持人: 🗡️ 🗡️ 狼人请睁眼，选择今晚要击杀的目标...

游戏主持人: 🗡️ 狼人们，请讨论今晚的击杀目标。存活玩家：孙权、周瑜、曹操、张飞、司马懿、赵云

孙权：今晚我们应该除掉周瑜，此人智谋过人，对我们威胁很大。

周瑜：孙权，你言之有理。但周瑜虽智，却未必是今晚的最大威胁。曹操势力庞大，若不尽早除去，恐对我们不利。

孙权：曹操的确是个威胁，但周瑜若活着，他能够识破我们的计谋。不如先解决眼前的隐患。

周瑜：孙权，你的顾虑不无道理。但曹操若与我们为敌，他可以联合其他势力对我们构成更大的威胁。

孙权：你说的也有道理，曹操的联合确实麻烦。那我们就先对付曹操吧。

周瑜：很好，孙权。曹操才是我们今晚首要的目标。

游戏主持人: 🗡️ 请选择击杀目标

孙权：我同意，曹操必须被除掉。

周瑜：我同意，曹操是我们今晚要解决的目标。

【预言家阶段】

游戏主持人: 🗡️ 🌟 预言家请睁眼，选择要查验的玩家...

曹操：我要查验孙权。

游戏主持人: 🗡️ 查验结果：孙权是狼人

【女巫阶段】

游戏主持人: 🗡️ 🧙 女巫请睁眼...

游戏主持人: 🗡️ 今晚曹操被狼人击杀

张飞：我昨晚使用了解药救了曹操，现在解药已经用掉了。

游戏主持人: 🗡️ 你使用解药救了曹操

游戏主持人: 🗡️ 昨夜平安无事，无人死亡。

【白天讨论阶段】

游戏主持人: 🎉 第1天天亮了, 请大家睁眼...

游戏主持人: 📺 现在开始自由讨论。存活玩家: 孙权、周瑜、曹操、张飞、司马懿、赵云

孙权: 诸位, 曹操势力庞大, 对我们都是潜在的威胁。今晚我建议我们集中力量对付他。

周瑜: 孙权所言极是, 曹操不仅自身强大, 还可能与其他玩家结盟, 对我们构成更大的威胁。

曹操: 我昨晚查验了孙权, 本以为他是好人, 但游戏主持人给出的结果却是狼人。这说明有狼人在说谎。

张飞: 我昨晚确实救了曹操, 说明他是被狼人袭击的。但曹操查验孙权的结果令人怀疑。

司马懿: 曹操的查验结果和张飞的救人行动似乎存在矛盾, 我们需要更多的信息来判断谁是狼人。

赵云: 情况确实复杂, 我们需要仔细分析各方的发言。

【投票阶段】

游戏主持人: 📺 请投票选择要淘汰的玩家

孙权: 曹操的威胁依然很大, 我坚持认为应该投票给他。

周瑜: 基于昨晚的情况, 我认为我们应该先投票给曹操, 他的威胁最大。

曹操: 我选择投票给孙权。根据游戏主持人的反馈, 孙权确实是狼人。

张飞: 我坚持昨晚救了曹操的事实, 但孙权被查出是狼人这一点让我感到困惑。

司马懿: 我们需要更多的信息来判断谁是狼人。

[游戏继续...]

6.3.3 AgentScope 的优势与局限性分析

通过这个“三国狼人杀”案例, 我们深度体验了 AgentScope 框架的核心优势。该框架以其消息驱动的架构为核心, 将复杂的游戏流程优雅地映射为一系列并发、异步的消息传递事件, 从而避免了传统状态机的僵硬与复杂。结合其强大的结构化输出能力, 我们将游戏规则直接转化为代码层面的约束, 极大地提升了系统的稳定性和可预测性。这种设计范式不仅在性能上展现了其原生并发的优势, 更在容错处理上保证了即使单个智能体出现异常, 整体流程也能稳健运行。

然而, AgentScope 的工程化优势也带来了一定的复杂性成本。其消息驱动架构虽然强大, 但对开发者的技术要求较高, 需要理解异步编程、分布式通信等概念。对于简单的多智能体对话场景, 这种架构可能显得过于复杂, 存在“过度工程化”的风险。此外, 作为相对较新的框架, 其生态系统和社区资源还有待进一步完善。因此, AgentScope 更适合需要构建大规模、高可靠的生产级多智能体系统, 而对于快速原型开发或简单应用场景, 选择更轻量级的框架可能更为合适。

6.4 框架三: CAMEL

与 AutoGen 和 AgentScope 这样功能全面的框架不同, CAMEL 最初的核心目标是探索如何在最少的人类干预下, 让两个智能体通过“角色扮演”自主协作解决复杂任务。

6.4.1 CAMEL 的自主协作

CAMEL 实现自主协作的基石是两大核心概念: 角色扮演 (Role-Playing) 和引导性提示 (Inception Prompting)。

(1) 角色扮演

在 CAMEL 最初的设计中, 一个任务通常由两个智能体协作完成。这两个智能体被赋予了互补的、明确定义的“角色”。一个扮演“AI 用户”(AI User), 负责提出需求、下达指令和构思任务步骤; 另一个则扮演“AI 助理”(AI Assistant), 负责根据指令执行具体操作和提供解决方案。

例如, 在一个“开发股票交易策略分析工具”的任务中:

- AI 用户的角色可能是一位“资深股票交易员”。它懂市场、懂策略, 但不懂编程。
- AI 助理的角色则是一位“优秀的 Python 程序员”。它精通编程, 但对股票交易一无所知。

通过这种设定, 任务的解决过程就被自然地转化为一场两位“跨领域专家”之间的对话。交易员提出专业需求, 程序员将其转化为代码实现, 两者协作完成任何一方都无法独立完成的复杂任务。

(2) 引导性提示

仅仅设定角色还不够, 如何确保两个 AI 在没有人类持续监督的情况下, 能始终“待在自己的角色里”, 并且高效地朝着共同目标前进呢? 这就是 CAMEL 最核心的技术, 引导性提示发挥作用的地方。“引导性提示”是在对话开始前, 分别注入给两个智能体的一段精心设计的、结构化的初始指令 (System Prompt)。这段指令就像是为智能体植入的“行动纲领”, 它通常包含以下几个关键部分:

- 明确自身角色: 例如, “你是一位资深的股票交易员...”
- 告知协作者角色: 例如, “你正在与一位优秀的 Python 程序员合作...”
- 定义共同目标: 例如, “你们的共同目标是开发一个股票交易策略分析工具。”
- 设定行为约束和沟通协议: 这是最关键的一环。例如, 指令会要求 AI 用户“一次只提出一个清晰、具体的步骤”, 并要求 AI 助理“在完成上一步之前不要追问更多细节”, 同时规定双方需在回复的末尾使用特定标志 (如 <SOLUTION>) 来标识任务的完成。

这些约束条件确保了对话不会偏离主题、不会陷入无效循环, 而是以一种高度结构化、任务驱动的方式向前推进, 如图6.3所示。



图 6.3 CAMEL 创建股票机器人交易

在下一节, 我们将通过一个具体的实例来体验这一过程。

6.4.2 AI科普电子书

为了理解 CAMEL 框架的角色扮演能力，我们将构建一个具有实际价值的协作案例：让一位 AI 心理学家与一位 AI 作者合作，共同创作一本关于“拖延症心理学”的短篇电子书。这个案例体现了 CAMEL 的核心优势，让两个智能体在各自专业领域发挥所长，协作完成单个智能体难以胜任的复杂创作任务。

(1) 任务设定

场景设定：创作一本面向普通读者的拖延症心理学科普电子书，要求既有科学严谨性，又具备良好的可读性。

智能体角色：

- 心理学家（**Psychologist**）：具备深厚的心理学理论基础，熟悉认知行为科学、神经科学等相关领域，能够提供专业的学术见解和实证研究支持
- 作家（**Writer**）：拥有优秀的写作技巧和叙述能力，善于将复杂的学术概念转化为生动易懂的文字，注重读者体验和内容的可读性

(2) 定义协作任务

首先，我们需要明确两位 AI 专家的共同目标。我们通过一个内容详实的字符串 `task_prompt` 来定义这个任务。

```
from colorama import Fore
from camel.societies import RolePlaying
from camel.utils import print_text_animated
from camel.models import ModelFactory
from camel.types import ModelPlatformType
from dotenv import load_dotenv
import os

load_dotenv()
LLM_API_KEY = os.getenv("LLM_API_KEY")
LLM_BASE_URL = os.getenv("LLM_BASE_URL")
LLM_MODEL = os.getenv("LLM_MODEL")

# 创建模型，在这里以Qwen为例，调用的百炼大模型平台API
model = ModelFactory.create(
    model_platform=ModelPlatformType.QWEN,
    model_type=LLM_MODEL,
    url=LLM_BASE_URL,
    api_key=LLM_API_KEY
)
```

定义协作任务
`task_prompt = """`
创作一本关于“拖延症心理学”的短篇电子书，目标读者是对心理学感兴趣的普通大众。

要求：

1. 内容科学严谨，基于实证研究
 2. 语言通俗易懂，避免过多专业术语
 3. 包含实用的改善建议和案例分析
 4. 篇幅控制在8000-10000字
 5. 结构清晰，包含引言、核心章节和总结
- """

```
print(Fore.YELLOW + f"协作任务:\n{task_prompt}\n")
```

`task_prompt` 是整个协作的“任务说明书”。它不仅是我们要完成的目标，也将在幕后被 CAMEL 用来生成“引导性提示”，确保两位智能体的对话始终围绕这个核心目标展开。

(3) 初始化角色扮演“社会”

接下来，我们创建 `RolePlaying` 会话实例。这是 CAMEL 的核心操作，它根据我们提供的角色和任务，快速构建一个双智能体协作“社会”。

```
# 初始化角色扮演会话
# AI 作家作为 "user"，负责提出写作结构和要求
# AI 心理学家作为 "assistant"，负责提供专业知识和内容
role_play_session = RolePlaying(
    assistant_role_name="心理学家",
    user_role_name="作家",
    task_prompt=task_prompt,
    model=model,
    with_task_specify=False, # 在本例中，我们直接使用给定的task_prompt
)

print(Fore.CYAN + f"具体任务描述:\n{role_play_session.task_prompt}\n")
```

`RolePlaying` 是 CAMEL 提供的高级 API，它封装了复杂的提示工程。我们只需传入两个角色的名称和任务即可。在 CAMEL 的设计中，`user` 角色是对话的“推动者”和“需求方”，而 `assistant` 角色是“执行者”和“方案提供方”。因此，我们将负责规划结构的“作家”分配给 `user_role_name`，将负责提供专业知识的“心理学家”分配给 `assistant_role_name`。

(4) 启动并运行自动化对话

最后，我们编写一个循环来驱动整个对话过程，让两位 AI 专家开始它们的自动化协作。

```
# 开始协作对话
chat_turn_limit, n = 30, 0
# 调用 init_chat() 来获得由 AI 生成的初始对话消息
```

2/9/26, 1:56 PM

```
input_msg = role_play_session.init_chat()

while n < chat_turn_limit:
    n += 1
    # step() 方法驱动一轮完整的对话，AI 用户和 AI 助理各发言一次
    assistant_response, user_response = role_play_session.step(input_msg)

    # 检查是否有消息返回，防止对话提前终止
    if assistant_response.msg is None or user_response.msg is None:
        break

    print_text_animated(Fore.BLUE + f"作家 (AI User):\n\n{user_response.msg.content}\n")
    print_text_animated(Fore.GREEN + f"心理学家 (AI Assistant):\n\n{assistant_response.msg.content}\n")

    # 检查任务完成标志
    if "<CAMEL_TASK_DONE>" in user_response.msg.content or "<CAMEL_TASK_DONE>" in assistant_response.msg.content:
        print(Fore.MAGENTA + "✅ 电子书创作完成!")
        break

    # 将助理的回复作为下一轮对话的输入
    input_msg = assistant_response.msg

print(Fore.YELLOW + f"总共进行了 {n} 轮协作对话")
```

这段 while 循环是自动化协作的核心。对话由 `init_chat()` 方法基于任务和角色自动开启，无需人工编写开场白。循环的每一步都通过调用 `step()` 来驱动一轮完整的交互（作家提需求、心理学家给内容），并将上一轮心理学家的输出作为下一轮的输入，形成环环相扣的创作链。整个过程将持续进行，直到达到预设的对话轮次上限，或任一智能体输出任务完成标志 `<CAMEL_TASK_DONE>` 后自动终止。

(5) 协作流程展示

当执行上述代码后，我们并非只是得到一长串单调的问答，而是能够观察到一个高度结构化的、如同人类专家团队般的协作流程在自动进行。整个创作过程自然地分为几个阶段：

第一阶段(约 1-5 轮): 框架搭建与目标对齐 在对话的初期，“作家”智能体首先会扮演起主导者的角色，提出对电子书整体结构和章节安排的初步设想。随后，“心理学家”会从其专业角度对这个框架进行审视和补充，确保核心的学术模块（如理论基础、关键概念等）没有遗漏，从而在协作开始之初就对最终产出物达成共识。

第二阶段(约 6-20 轮): 核心内容生成与知识转译 这是最高效的内容创作阶段。协作模式会变为一种稳定的“请求-响应”循环：

- 心理学家：负责提供“硬核”的专业知识，如对“时间折扣理论”、“执行功能缺陷”等核心概念的科学解释，并引用相关的实验研究来支撑观点。
- 作家：则发挥其“翻译官”的作用，将这些严谨但可能晦涩的学术概念，转化为生动、形象的比喻和贴近生活的案例。例如，它可能会将“大脑中的‘现在偏见’”这个概念，比作“一个只顾眼前糖果、不顾长远健康的任性孩子”。

第三阶段(约 21-25 轮): 迭代优化与质量保证 当书籍的主体内容完成后，对话的重心会转移到对已有文本的打磨和完善上。此时，两位智能体的角色会发生微妙的变化：

- 作家：更侧重于审视文章的整体流畅性、逻辑衔接和语言风格，从“读者体验”出发提出修改建议。
- 心理学家：则再次扮演“事实核查员”，确保在转译和润色的过程中，核心知识的科学准确性没有丢失，并为某些观点补充更有力的实证研究支持。

第四阶段(收尾): 总结与升华 在最后的几轮对话中，双方会协作完成实用建议的总结和全书的回顾，确保电子书有一个清晰、有力的结尾，为读者留下深刻印象并提供实际价值。

协作任务：

创作一本关于“拖延症心理学”的短篇电子书，目标读者是对心理学感兴趣的普通大众。

要求：

1. 内容科学严谨，基于实证研究
2. 语言通俗易懂，避免过多专业术语
3. 包含实用的改善建议和案例分析
4. 篇幅控制在8000–10000字
5. 结构清晰，包含引言、核心章节和总结

具体任务描述：

为普通大众撰写8000–10000字短篇电子书《拖延症心理学》：实证为本、通俗易懂。结构：引言、成因（认知/情绪/奖励）、动机与决策、习惯形成与干预、实用策略与练习、三则案例分析、总结与资源。每章含研究引用与可操作步骤。

作家：

Instruction：请为电子书的“引言”章节撰写一段400–600字的中文草稿...

Input：None

心理学家：

Solution：

草稿：拖延，是指明知应当完成某项任务却反复推迟或回避的行为与内在倾向。它既可以是偶发的时间管理问题...

Next request.

作家：

Instruction：请把下面的引言草稿修订为一段450–550字的中文文本...

Input：草稿：拖延，是指明知应当完成某项任务却反复推迟或回避的行为...

....

6.4.3 CAMEL 的优势与局限性分析

通过前面的电子书创作案例，我们深度体验了 CAMEL 框架独特的角色扮演范式。现在让我们客观地分析这种设计理念的优势与局限性，以便在实际项目中做出明智的技术选型。

(1) 优势

CAMEL 最大的优势在于其“轻架构、重提示”的设计哲学。相比 AutoGen 的复杂对话管理和 AgentScope 的分布式架构，CAMEL 通过精心设计的初始提示就能实现高质量的智能体协作。这种自然涌现的协作行为，往往比硬编码的工作流更加灵活和高效。

值得注意的是，CAMEL 框架正在经历快速的发展和演进。从其 [GitHub 仓库](#) 可以看到，CAMEL 已经远不止是一个简单的双智能体协作框架，目前已经具备：

- 多模态能力：支持文本、图像、音频等多种模态的智能体协作
- 工具集成：内置了丰富的工具库，包括搜索、计算、代码执行等
- 模型适配：支持 OpenAI、Anthropic、Google、开源模型等多种 LLM 后端
- 生态联动：与 LangChain、CrewAI、AutoGen 等主流框架实现了互操作性

(2) 主要局限性

1. 对提示工程的高度依赖

CAMEL 的成功很大程度上取决于初始提示的质量。这带来了几个挑战：

- 提示设计门槛：需要深入理解目标领域和 LLM 的行为特性
- 调试复杂性：当协作效果不佳时，很难定位是角色定义、任务描述还是交互规则的问题
- 一致性挑战：不同的 LLM 对相同提示的理解可能存在差异

2. 协作规模的限制

虽然 CAMEL 在双智能体协作上表现出色，但在处理大规模多智能体场景时面临挑战：

- 对话管理：缺乏像 AutoGen 那样的复杂对话路由机制
- 状态同步：没有 AgentScope 那样的分布式状态管理能力
- 冲突解决：当多个智能体意见分歧时，缺乏有效的仲裁机制

3. 任务适用性的边界

CAMEL 特别适合需要深度协作和创造性思维的任务，但在某些场景下可能不是最优选择：

- 严格流程控制：对于需要精确步骤控制的任务，LangGraph 的图结构更合适
- 大规模并发：AgentScope 的消息驱动架构在高并发场景下更有优势
- 复杂决策树：AutoGen 的群聊模式在多方决策场景下更加灵活

总的来说，CAMEL 代表了一种独特而优雅的多智能体协作范式。它通过“以人为本”的角色扮演设计，将复杂的系统工程问题转化为直观的人际协作模式。随着其生态系统的不断完善和功能的持续扩展，CAMEL 正在成为构建智能协作系统的重要选择之一。

6.5 框架四：LangGraph

6.5.1 LangGraph 的结构梳理

LangGraph 作为 LangChain 生态系统的重要扩展，代表了智能体框架设计的一个全新方向。与前面介绍的基于“对话”的框架（如 AutoGen 和 CAMEL）不同，LangGraph 将智能体的执行流程建模为一种状态机（State Machine），并将其表示为有向图（Directed Graph）。在这种范式中，图的节点（Nodes）代表一个具体的计算步骤（如调用 LLM、执行工具），而边（Edges）则定义了从一个节点到另一个节点的跳转逻辑。这种设计的革命性之处在于它天然支持循环，使得构建能够进行迭代、反思和自我修正的复杂智能体工作流变得前所未有的直观和简单。

要理解 LangGraph，我们需要先掌握它的三个基本构成要素。

首先，是全局状态（State）。整个图的执行过程都围绕一个共享的状态对象进行。这个状态通常被定义为一个 Python 的 TypedDict，它可以包含任何你需要追踪的信息，如对话历史、中间结果、迭代次数等。所有的节点都能读取和更新这个中心状态。

```
from typing import TypedDict, List
```

```
# 定义全局状态的数据结构
class AgentState(TypedDict):
    messages: List[str]      # 对话历史
    current_task: str        # 当前任务
    final_answer: str        # 最终答案
    # ... 任何其他需要追踪的状态
```

其次，是节点（Nodes）。每个节点都是一个接收当前状态作为输入、并返回一个更新后的状态作为输出的 Python 函数。节点是执行具体工作的单元。

```
# 定义一个“规划者”节点函数
def planner_node(state: AgentState) -> AgentState:
    """根据当前任务制定计划，并更新状态。””
    current_task = state["current_task"]
    # ... 调用LLM生成计划 ...
```

```

plan = f"为任务 '{current_task}' 生成的计划..."

# 将新消息追加到状态中
state["messages"].append(plan)
return state

# 定义一个“执行者”节点函数
def executor_node(state: AgentState) -> AgentState:
    """执行最新计划，并更新状态。””
    latest_plan = state["messages"][-1]
    # ... 执行计划并获得结果 ...
    result = f"执行计划 '{latest_plan}' 的结果..."

    state["messages"].append(result)
    return state

```

最后，是边（**Edges**）。边负责连接节点，定义工作流的方向。最简单的边是常规边，它指定了一个节点的输出总是流向另一个固定的节点。而 LangGraph 最强大的功能在于条件边（**Conditional Edges**）。它通过一个函数来判断当前的状态，然后动态地决定下一步应该跳转到哪个节点。这正是实现循环和复杂逻辑分支的关键。

```

def should_continue(state: AgentState) -> str:
    """条件函数：根据状态决定下一步路由。””
    # 假设如果消息少于3条，则需要继续规划
    if len(state["messages"]) < 3:
        # 返回的字符串需要与添加条件边时定义的键匹配
        return "continue_to_planner"
    else:
        state["final_answer"] = state["messages"][-1]
        return "end_workflow"

```

在定义了状态、节点和边之后，我们可以像搭积木一样将它们组装成一个可执行的工作流。

```

from langgraph.graph import StateGraph, END

# 初始化一个状态图，并绑定我们定义的状态结构
workflow = StateGraph(AgentState)

# 将节点函数添加到图中
workflow.add_node("planner", planner_node)
workflow.add_node("executor", executor_node)

# 设置图的入口点
workflow.set_entry_point("planner")

# 添加常规边，连接 planner 和 executor
workflow.add_edge("planner", "executor")

# 添加条件边，实现动态路由
workflow.add_conditional_edges(
    # 起始节点
    "executor",
    # 判断函数
    should_continue,
    # 路由映射：将判断函数的返回值映射到目标节点
    {
        "continue_to_planner": "planner", # 如果返回"continue_to_planner"，则跳回planner节点
        "end_workflow": END # 如果返回"end_workflow"，则结束流程
    }
)

# 编译图，生成可执行的应用
app = workflow.compile()

# 运行图
inputs = {"current_task": "分析最近的AI行业新闻", "messages": []}
for event in app.stream(inputs):
    print(event)

```

6.5.2 三步问答助手

在理解了 LangGraph 的核心概念之后，我们将通过一个实战案例来巩固所学。我们将构建一个简化的问答对话助手，它会遵循一个清晰、固定的三步流程来回答用户的问题：

1. 理解（Understand）：首先，分析用户的查询意图。
2. 搜索（Search）：然后，模拟搜索与意图相关的信息。
3. 回答（Answer）：最后，基于意图和搜索到的信息，生成最终答案。

这个案例将清晰地展示如何定义状态、创建节点以及将它们线性地连接成一个完整的工作流。我们将代码分解为四个核心步骤：定义状态、创建节点、构建图、以及运行应用。

(1) 定义全局状态

首先，我们需要定义一个贯穿整个工作流的全局状态。这是一个共享的数据结构，它在图的每个节点之间传递，作为工作流的持久化上下文。每个节点都可以读取该结构中的数据，并对其进行更新。

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages

class SearchState(TypedDict):
    messages: Annotated[list, add_messages]
    user_query: str      # 经过LLM理解后的用户需求总结
    search_query: str    # 优化后用于Tavily API的搜索查询
    search_results: str  # Tavily搜索返回的结果
    final_answer: str    # 最终生成的答案
    step: str            # 标记当前步骤
```

我们创建了 `SearchState` 这个 `TypedDict`, 为状态对象定义了一个清晰的数据模式 (Schema)。一个关键的设计是同时包含了 `user_query` 和 `search_query` 字段。这允许智能体先将用户的自然语言提问, 优化成更适合搜索引擎的精炼关键词, 从而显著提升搜索结果的质量。

(2) 定义工作流节点

定义好状态结构后, 下一步是创建构成我们工作流的各个节点。在 `LangGraph` 中, 每个节点都是一个执行具体任务的 Python 函数。这些函数接收当前的状态对象作为输入, 并返回一个包含更新后字段的字典。

在开始定义节点之前, 我们先完成项目的初始化设置, 包括加载环境变量和实例化大语言模型。

```
import os
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage
from tavily import TavilyClient

# 加载 .env 文件中的环境变量
load_dotenv()

# 初始化模型
# 我们将使用这个 llm 实例来驱动所有节点的智能
llm = ChatOpenAI(
    model=os.getenv("LLM_MODEL_ID", "gpt-4o-mini"),
    api_key=os.getenv("LLM_API_KEY"),
    base_url=os.getenv("LLM_BASE_URL", "https://api.openai.com/v1"),
    temperature=0.7
)
# 初始化Tavily客户端
tavily_client = TavilyClient(api_key=os.getenv("TAVILY_API_KEY"))
```

现在, 我们来逐一创建三个核心节点。

(1) 理解与查询节点

此节点是工作流的第一步, 此节点的职责是理解用户意图, 并为其生成一个最优化的搜索查询。

```
def understand_query_node(state: SearchState) -> dict:
    """步骤1: 理解用户查询并生成搜索关键词"""
    user_message = state["messages"][-1].content

    understand_prompt = f"""分析用户的查询: "{user_message}"  

请完成两个任务:  

1. 简洁总结用户想要了解什么  

2. 生成最适合搜索引擎的关键词 (中英文均可, 要精准)  

格式:  

理解: [用户需求总结]  

搜索词: [最佳搜索关键词]"""

    response = llm.invoke([SystemMessage(content=understand_prompt)])
    response_text = response.content

    # 解析LLM的输出, 提取搜索关键词
    search_query = user_message # 默认使用原始查询
    if "搜索词: " in response_text:
        search_query = response_text.split("搜索词: ")[1].strip()

    return {
        "user_query": response_text,
        "search_query": search_query,
        "step": "understood",
        "messages": [AIMessage(content=f"我将为您搜索: {search_query}")]
    }
```

该节点通过一个结构化的提示, 要求 LLM 同时完成“意图理解”和“关键词生成”两个任务, 并将解析出的专用搜索关键词更新到状态的 `search_query` 字段中, 为下一步的精确搜索做好准备。

(2) 搜索节点

该节点负责执行智能体的“工具使用”能力, 它将调用 `Tavily API` 进行真实的互联网搜索, 并具备基础的错误处理功能。

```
def tavily_search_node(state: SearchState) -> dict:
    """步骤2: 使用Tavily API进行真实搜索"""
    search_query = state["search_query"]
    try:
```

```

print(f"🔍 正在搜索: {search_query}")
response = tavily_client.search(
    query=search_query, search_depth="basic", max_results=5, include_answer=True
)
# ... (处理和格式化搜索结果) ...
search_results = ... # 格式化后的结果字符串

return {
    "search_results": search_results,
    "step": "searched",
    "messages": [AIMessage(content="✅ 搜索完成! 正在整理答案...")]
}
except Exception as e:
    # ... (处理错误) ...
    return {
        "search_results": f"搜索失败: {e}",
        "step": "search_failed",
        "messages": [AIMessage(content="❌ 搜索遇到问题...")]
}

```

此节点通过 `tavily_client.search` 发起真实的 API 调用。它被包裹在 `try...except` 块中，用于捕获可能的异常。如果搜索失败，它会更新 `step` 状态为 `"search_failed"`，这个状态将被下一个节点用来触发备用方案。

(3) 回答节点

最后的回答节点能够根据上一步的搜索是否成功，来选择不同的回答策略，具备了一定的弹性。

```

def generate_answer_node(state: SearchState) -> dict:
    """步骤3: 基于搜索结果生成最终答案"""
    if state["step"] == "search_failed":
        # 如果搜索失败，执行回退策略，基于LLM自身知识回答
        fallback_prompt = f"搜索API暂时不可用，请基于您的知识回答用户的问题: \n用户问题: {state['user_query']}"
        response = llm.invoke([SystemMessage(content=fallback_prompt)])
    else:
        # 搜索成功，基于搜索结果生成答案
        answer_prompt = f"""基于以下搜索结果为用户提供完整、准确的答案:
用户问题: {state['user_query']}
搜索结果: \n{state['search_results']}
请综合搜索结果，提供准确、有用的回答..."""
        response = llm.invoke([SystemMessage(content=answer_prompt)])
    return {
        "final_answer": response.content,
        "step": "completed",
        "messages": [AIMessage(content=response.content)]
    }

```

该节点通过检查 `state["step"]` 的值来执行条件逻辑。如果搜索失败，它会利用 LLM 的内部知识回答并告知用户情况。如果搜索成功，它则会使用包含实时搜索结果的提示，来生成一个有时效性且有据可依的回答。

(4) 构建图

我们将所有节点连接起来。

```

from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import InMemorySaver

def create_search_assistant():
    workflow = StateGraph(SearchState)

    # 添加节点
    workflow.add_node("understand", understand_query_node)
    workflow.add_node("search", tavily_search_node)
    workflow.add_node("answer", generate_answer_node)

    # 设置线性流程
    workflow.add_edge(START, "understand")
    workflow.add_edge("understand", "search")
    workflow.add_edge("search", "answer")
    workflow.add_edge("answer", END)

    # 编译图
    memory = InMemorySaver()
    app = workflow.compile(checkpointer=memory)
    return app

```

(5) 运行案例展示

运行此脚本后，您可以提出一些需要实时信息的问题，例如我们第一章中的案例：明天我要去北京，天气怎么样？有合适的景点吗

您会看到终端清晰地展示出智能体的“思考”过程：

```

🔍 智能搜索助手启动!
我会使用Tavily API为您搜索最新、最准确的信息
支持各种问题：新闻、技术、知识问答等
(输入 'quit' 退出)

```

💡 您想了解什么：明天我要去北京，天气怎么样？有合适的景点吗

=====

🧠 理解阶段：我理解您的需求：理解：用户想了解明天北京的天气情况以及合适的景点推荐。

搜索词：北京 明天 天气 景点推荐 Beijing weather tomorrow attractions

🔍 正在搜索：北京 明天 天气 景点推荐 Beijing weather tomorrow attractions

🔍 搜索阶段：✓ 搜索完成！找到了相关信息，正在为您整理答案…

💡 最终回答：

明天（2025年9月17日）北京的天气预报显示，预计将是多云，气温范围在17°C (62°F) 到25°C (77°F) 之间。这种温和的天气非常适合户外活动。

合适的景点推荐：

1. **长城**：作为中国最著名的历史遗址之一，长城是必游之地。你可以选择八达岭或慕田峪这些较为受欢迎的段落进行游览。

2. **故宫**：故宫是明清两代的皇宫，拥有丰富的历史和文化，适合对中国历史感兴趣的游客。

3. **天安门广场**：这是中国的象征之一，广场上有许多重要的建筑和纪念碑，适合拍照留念。

4. **颐和园**：一个非常美丽的皇家园林，适合漫步和欣赏自然风光，尤其是湖泊和古建筑。

5. **798艺术区**：如果你对现代艺术感兴趣，798艺术区是一个集艺术、文化和创意于一体的地方，适合探索和拍摄。

小贴士：

- 由于明天天气良好，建议提前规划出行路线，并准备一些水和小吃，以便在游览时保持充足的体力。

- 由于天气变化可能会影响游览体验，建议查看实时天气更新。

希望这些信息能帮助你安排一个愉快的北京之旅！如果你需要更多关于景点的信息或者旅行建议，欢迎随时询问。

=====

💡 您想了解什么：

并且他是一个可以持续交互的助手，你也可以继续向他发问。

6.5.3 LangGraph 的优势与局限性分析

任何技术框架都有其特定的适用场景和设计权衡。在本节中，我们将客观地分析 LangGraph 的核心优势及其在实际应用中可能面临的局限性。

(1) 优势

- 如我们的智能搜索助手案例所示，LangGraph 将一个完整的实时问答流程，显式地定义为一个由状态、节点和边构成的“流程图”。这种设计的最大优势是高度的可控性与可预测性。开发者可以精确地规划智能体的每一步行为，这对于构建需要高可靠性和可审计性的生产级应用至关重要。其最强大的特性在于对循环（Cycles）的原生支持。通过条件边，我们可以轻松构建“反思-修正”循环，例如在我们的案例中，如果搜索失败，可以设计一个回退到备用方案的路径。这是构建能够自我优化和具备容错能力的智能体的关键。
- 此外，由于每个节点都是一个独立的 Python 函数，这带来了高度的模块化。同时，在流程中插入一个等待人类审核的节点也变得非常直接，为实现可靠的“人机协作”（Human-in-the-loop）提供了坚实的基础。

(2) 局限性

- 与基于对话的框架相比，LangGraph 需要开发者编写更多的前期代码（Boilerplate）。定义状态、节点、边等一系列操作，使得对于简单任务而言，开发过程显得更为繁琐。开发者需要更多地思考“如何控制流程（how）”，而不仅仅是“做什么（what）”。由于工作流是预先定义的，LangGraph 的行为虽然可控，但也缺少了对话式智能体那种动态的、“涌现”式的交互。它的强项在于执行一个确定的、可靠的流程，而非模拟开放式的、不可预测的社会性协作。
- 调试过程同样存在挑战。虽然流程比对话历史更清晰，但问题可能出在多个环节：某个节点内部的逻辑错误、在节点间传递的状态数据发生异变，或是边跳转的条件判断失误。这要求开发者对整个图的运行机制有全局性的理解。

6.6 本章小结

本章我们感受了目前最前沿的一些智能体框架，通过案例的形式进行实操体验。

我们看到，每一个框架都有自己实现智能体构建的思路：

- AutoGen 将复杂的协作抽象为一场由多角色参与的、可自动进行的“群聊”，其核心在于“以对话驱动协作”。
- AgentScope 则着眼于工业级应用的健壮性与可扩展性，为构建高并发、分布式的多智能体系统提供了坚实的工程基础。
- CAMEL 以其轻量级的“角色扮演”和“引导性提示”范式，展示了如何用最少的代码激发两个专家智能体之间深度、自主的协作。
- LangGraph 则回归到更底层的“状态机”模型，通过显式的图结构赋予开发者对工作流的精确控制，尤其是其循环能力，为构建可反思、可修正的智能体铺平了道路。

通过对这些框架的深入分析，我们可以提炼出一个设计的权衡：“涌现式协作”与“显式控制”之间的选择。AutoGen 和 CAMEL 更多地依赖于定义智能体的“角色”和“目标”，让复杂的协作行为从简单的对话规则中“涌现”出来，这种方式更贴近人类的交互模式，但有时难以预测和调试。而 LangGraph 要求开发者明确地定义每一个步骤和跳转条件，牺牲了一部分“涌现”的惊喜，换来了高度的可靠性、可控性和可观测性。同时，AgentScope 则揭示了第二个同样重要的维度：工程化。无论我们选择哪种协作范式，要将其从实验原型推向生产应用，都必须面对并发、容错、分布式部署等工程挑战。AgentScope 正是为解决这些问题而生，它代表了从“能运行”到“能稳定服务”的关键跨越。

总而言之，智能体并非只有一种构建方式。深入理解本章探讨的框架设计哲学，能让我们不仅仅成为更优秀的“工具使用者”，更能理解框架设计中的各种优劣与权衡。

在下一章中，我们将进入本教程的核心内容，从零开始，亲手构建一个属于我们自己的智能体框架，将所有理论与实践融会贯通。

习题

1. 本章介绍了四个各具特色的智能体框架：AutoGen、AgentScope、CAMEL 和 LangGraph。请分析：

- 在6.1.2节的表6.1中，对比了这四个框架的多个维度。请选择其中两个你最熟悉的框架，从“协作模式”、“控制方式”、“适用场景”三个维度进一步深入对比。
- 本章提到了“涌现式协作”与“显式控制”之间的权衡，如何理解这两种设计哲学的含义。

2. 在6.2节的 AutoGen 案例中，我们构建了一个“软件开发团队”。请基于此案例进行扩展思考：

提示：这是一道动手实践题，建议实际操作

- 当前的团队使用 RoundRobinGroupChat（轮询群聊）模式，智能体按固定顺序发言。如果需求变更，工程师的代码需要返回给产品经理重新审核，应该如何修改协作流程？请设计一个支持“动态回退”的机制。
- 在案例中，我们通过 System Message 为每个智能体定义了角色和职责。请尝试为这个团队添加一个新角色“测试工程师”（Quality Assurance），并设计其系统消息，使其能够在代码审查后执行自动化测试。
- AutoGen 的对话式协作存在可能的不稳定性，可能导致对话偏离主题或陷入循环。请思考：如何设计一套“对话质量监控”机制，在检测到异常时及时干预？

3. 在6.3节的 AgentScope 案例中，我们实现了一个“三国狼人杀”游戏。请深入分析：

- 案例中使用了 MsgHub（消息中心）来管理智能体间的通信。请解释消息驱动架构相比传统函数调用的优势是什么？在什么场景下这种架构特别有价值？
- 游戏中使用了结构化输出（如 DiscussionModelCN、WitchActionModelCN）来约束智能体行为。请设计一个新的游戏角色“猎人”，并定义其对应的结构化输出模型，包括字段定义和验证规则。
- AgentScope 支持分布式部署，这意味着不同的智能体可以运行在不同的服务器上。请思考：在“三国狼人杀”这样的实时游戏场景中，分布式部署会带来哪些技术挑战？如何保证消息的顺序性和一致性？

4. 在6.4节的 CAMEL 案例中，我们让心理学家和作家协作创作电子书。

- 在案例中，协作会在检测到 <CAMEL_TASK_DONE> 标志时强制终止。但如果两个智能体意见分歧（一位认为可以终止，一位认为不应该终止），无法达成一致怎么办？请设计一个“冲突解决”的兼容机制。
- CAMEL 最初设计用于双智能体协作，但现在已经扩展支持多智能体。请查阅 CAMEL 的最新文档，了解其多智能体协作模块 [workforce](#)，并结合架构图说明其与 AutoGen 的群聊模式有何不同。

5. 在6.5节的 LangGraph 案例中，我们构建了一个“三步问答助手”。请分析：

- LangGraph 将智能体流程建模为状态机和有向图。请画出案例中“理解-搜索-回答”流程的图结构，标注节点、边和状态转换条件。
- 当前的助手是一个线性流程。请扩展这个案例，添加一个“反思”节点：如果生成的答案质量低（例如过于简短或缺乏细节），系统应该重新搜索或重新生成答案。请设计这个循环机制的条件边逻辑。
- LangGraph 的优势在于对循环的原生支持。请设计一个更复杂的应用场景，充分利用这一特性：例如“代码生成-测试-修复”循环、“论文写作-审阅-修改”循环等。要求画出完整的图结构并说明关键节点的功能。

6. 框架选型是智能体产品开发过程中的关键决策之一。假设你是一家 AI 公司的技术架构师，公司计划开发以下三个智能体产品应用，请为每个应用选择最合适的选择（AutoGen、AgentScope、CAMEL、LangGraph 或不借助框架从零开发），并详细说明理由：

应用A： 智能客服系统，需要处理大量并发用户请求（每秒1000+），要求响应时间低于2秒，系统需要7×24小时稳定运行，并支持水平扩展。

应用B： 科研论文辅助写作平台，需要一个“研究员智能体”和一个“写作智能体”深度协作，共同完成文献综述、实验设计、数据分析和论文撰写。要求智能体能够进行多轮深度讨论，自主推进任务。

应用C： 金融风控审批系统，需要按照严格的流程处理贷款申请：资料审核 → 风险评估 → 额度计算 → 合规检查 → 人工复核 → 最终决策。每个环节都有明确的判断标准和分支逻辑，要求流程可追溯、可审计。

参考文献

[1] Wu Q, Bansal G, Zhang J, et al. Autogen: Enabling next-gen LLM applications via multi-agent conversations[C]//First Conference on Language Modeling. 2024.

[2] Gao D, Li Z, Pan X, et al. Agentscope: A flexible yet robust multi-agent platform[J]. arXiv preprint arXiv:2402.14034, 2024.

[3] Li G, Hammoud H, Itani H, et al. Camel: Communicative agents for “mind” exploration of large language model society[J]. Advances in Neural Information Processing Systems, 2023, 36: 51991-52008.

[4] LangChain. LangGraph [EB/OL]. (2024). <https://github.com/langchain-ai/langgraph>.

[5] Microsoft. AutoGen - UserProxyAgent [EB/OL]. (2024). https://microsoft.github.io/autogen/stable/reference/python/autogen_agentchat.agents.html#autogen_agentchat.agents.UserProxyAgent.

第七章 构建你的智能体框架

在前面的章节中，我们讲解了智能体的基础知识，并体验了主流框架带来的开发便利。从本章开始，我们将进入一个更具挑战也更有价值的阶段：从零开始，逐步构建一个智能体框架——HelloAgents。

为确保学习过程的连贯性与可复现性，HelloAgents 将以版本迭代的方式推进开发。每一章都会在前一章的基础上增加新的功能模块，并将智能体相关的知识点进行串讲与实现。最终，我们将利用这个自建框架，来高效地实现本书后续章节中的高级应用案例。

7.1 框架整体架构设计

7.1.1 为何需要自建Agent框架

在智能体技术快速发展的今天，市面上已经存在众多成熟的Agent框架。那么，为什么我们还要从零开始构建一个新的框架呢？

(1) 市面框架的快速迭代与局限性

智能体领域是一个快速发展的领域，随时会有新的概念产生，对于智能体的设计每个框架都有自己的定位和理解，不过智能体的核心知识点是一致的。

- 过度抽象的复杂性：许多框架为了追求通用性，引入了大量抽象层和配置选项。以LangChain为例，其链式调用机制虽然灵活，但对初学者而言学习曲线陡峭，往往需要理解大量概念才能完成简单任务。
- 快速迭代带来的不稳定性：商业化框架为了抢占市场，API接口变更频繁。开发者经常面临版本升级后代码无法运行的困扰，维护成本居高不下。
- 黑盒化的实现逻辑：许多框架将核心逻辑封装得过于严密，开发者难以理解Agent的内部工作机制，缺乏深度定制能力。遇到问题时只能依赖文档和社区支持，尤其是如果社区不够活跃，可能一个反馈意见会非常久也没有人推进，影响后续的开发效率。
- 依赖关系的复杂性：成熟框架往往携带大量依赖包，安装包体积庞大，在需要与别的项目代码配合的情况下可能出现依赖冲突问题。

(2) 从使用者到构建者的能力跃迁

构建自己的Agent框架，实际上是一个从“使用者”向“构建者”转变的过程。这种转变带来的价值是长远的。

- 深度理解Agent工作原理：通过亲手实现每个组件，开发者能够真正理解Agent的思考过程、工具调用机制、以及各种设计模式的好坏与区别。
- 获得完全的控制权：自建框架意味着对每一行代码都有完全的掌控，可以根据具体需求进行精确调优，而不受第三方框架设计理念的束缚。
- 培养系统设计能力：框架构建过程涉及模块化设计、接口抽象、错误处理等软件工程核心技能，这些能力对开发者的长期成长具有重要价值。

(3) 定制化需求与深度掌握的必要性

在实际应用中，不同场景对智能体的需求差异巨大，往往都需要在通用框架基础上做二次开发。

- 特定领域的优化需求：金融、医疗、教育等垂直领域往往需要针对性的提示词模板、特殊的工具集成、以及定制化的安全策略。
- 性能与资源的精确控制：生产环境中，对响应时间、内存占用、并发处理能力都有严格要求，通用框架的“一刀切”方案往往无法满足精细化需求。
- 学习与教学的透明性要求：在我们的教学场景中，学习者更期待的是清晰地看到智能体的每一步构建过程，理解不同范式的工作机制，这要求框架具有高度的可观测性和可解释性。

7.1.2 HelloAgents框架的设计理念

构建一个新的Agent框架，关键不在于功能的多少，而在于设计理念是否能真正解决现有框架的痛点。HelloAgents框架的设计围绕着一个核心问题展开：如何让学习者既能快速上手，又能深入理解Agent的工作原理？

当你初次接触任何成熟的框架时，可能会被其丰富的功能所吸引，但很快就会发现一个问题：要完成一个简单的任务，往往需要理解Chain、Agent、Tool、Memory、Retriever等十几个不同的概念。每个概念都有自己的抽象层，学习曲线变得异常陡峭。这种复杂性虽然带来了强大的功能，但也成为了初学者的障碍。HelloAgents框架试图在功能完整性和学习友好性之间找到平衡点，形成了四个核心的设计理念。

(1) 轻量级与教学友好的平衡

一个优秀的学习框架应该具备完整的可读性。HelloAgents将核心代码按照章节区分开，这是基于一个简单的原则：任何有一定编程基础的开发者都应该能够在合理的时间内完全理解框架的工作原理。在依赖管理方面，框架采用了极简主义的策略。除了OpenAI的官方SDK和几个必要的基础库外，不引入任何重型依赖。如果遇到问题时，我们可以直接定位到框架本身的代码，而不需要在复杂的依赖关系中寻找答案。

(2) 基于标准API的务实选择

OpenAI的API已经成为了行业标准，几乎所有主流的LLM提供商都在努力兼容这套接口。HelloAgents选择在这个标准之上构建，而不是重新发明一套抽象接口。这个决定主要是出于几点动机。首先是兼容性的保证，当你掌握了HelloAgents的使用方法后，迁移到其他框架或将其集成到现有项目中时，底层的API调用逻辑是完全一致的。其次是学习成本的降低。你不需要学习新的概念模型，因为所有的操作都基于你已经熟悉的标准接口。

(3) 渐进式学习路径的精心设计

HelloAgents提供了一条清晰的学习路径。我们将会把每一章的学习代码，保存为一个可以pip下载的历史版本，因此无需担心代码的使用成本，因为每一个核心的功能都将会是你自己编写的。这种设计让你能够按照自己的需求和节奏前进。每一步的升级都是自然而然的，不会产生概念上的跳跃或理解上的断层。值得一提的是，我们这一章的内容，也是基于前六章的内容来完善的。同样，这一章也是为后续高级知识学习部分打下框架基础。

(4) 统一的“工具”抽象：万物皆为工具

为了彻底贯彻轻量级与教学友好的理念，HelloAgents在架构上做出了一个关键的简化：除了核心的Agent类，一切皆为Tools。在许多其他框架中需要独立学习的Memory（记忆）、RAG（检索增强生成）、RL（强化学习）、MCP（协议）等模块，在HelloAgents中都被统一抽象为一种“工具”。这种设计的初衷是消除不必要的抽象层，让学习者可以回归到最直观的“智能体调用工具”这一核心逻辑上，从而真正实现快速上手和深入理解的统一。

7.1.3 本章学习目标

让我们先看看第七章的核心学习内容：

```
hello-agents/
└── hello_agents/
    ├── core/           # 核心框架层
    │   ├── agent.py    # Agent基类
    │   ├── llm.py       # HelloAgentsLLM统一接口
    │   ├── message.py  # 消息系统
    │   ├── config.py   # 配置管理
    │   └── exceptions.py # 异常体系
    ├── agents/          # Agent实现层
    │   ├── simple_agent.py # SimpleAgent实现
    │   ├── react_agent.py # ReActAgent实现
    │   ├── reflection_agent.py # ReflectionAgent实现
    │   └── plan_solve_agent.py # PlanAndSolveAgent实现
    └── tools/           # 工具系统层
        ├── base.py      # 工具基类
        ├── registry.py  # 工具注册机制
        ├── chain.py     # 工具链管理系统
        ├── async_executor.py # 异步工具执行器
        └── builtin/
            ├── calculator.py # 计算工具
            └── search.py     # 搜索工具
```

在开始编写具体代码之前，我们需要先建立一个清晰的架构蓝图。HelloAgents的架构设计遵循了“分层解耦、职责单一、接口统一”的核心原则，这样既保持了代码的组织性，也便于按照章节扩展内容。

快速开始：安装HelloAgents框架

为了让读者能够快速体验本章的完整功能，我们提供了可直接安装的Python包。你可以通过以下命令安装本章对应的版本：

```
# python版本需要>=3.10
pip install "hello-agents==0.1.1"
```

本章的学习可以采用两种方式：

1. 体验式学习：直接使用pip安装框架，运行示例代码，快速体验各种功能
2. 深度学习：跟随本章内容，从零开始实现每个组件，深入理解框架的设计思想和实现细节

我们建议采用“先体验，后实现”的学习路径。在本章中，我们提供了完整的测试文件，你可以重写核心函数并运行测试，以检验你的实现是否正确。这种方式既保证了实践性，又确保了学习效果。如果你想深入了解框架的实现细节，或者希望参与到框架的开发中来，可以访问这个[GitHub仓库](#)。

在开始之前，让我们用30秒体验使用Hello-agents构建简单智能体！

```
# 配置好同级文件夹下.env中的大模型API，可参考code文件夹配套的.env.example，也可以拿前几章的案例的.env文件复用。
from hello_agents import SimpleAgent, HelloAgentsLLM
from dotenv import load_dotenv

# 加载环境变量
load_dotenv()

# 创建LLM实例 - 框架自动检测provider
llm = HelloAgentsLLM()

# 或手动指定provider（可选）
# llm = HelloAgentsLLM(provider="modelscope")

# 创建SimpleAgent
agent = SimpleAgent(
    name="AI助手",
    llm=llm,
    system_prompt="你是一个有用的AI助手"
)

# 基础对话
response = agent.run("你好！请介绍一下自己")
print(response)

# 添加工具功能（可选）
from hello_agents.tools import CalculatorTool
calculator = CalculatorTool()
```

```
# 需要实现7.4.1的MySimpleAgent进行调用，后续章节会支持此类调用方式
# agent.add_tool(calculator)

# 现在可以使用工具了
response = agent.run("请帮我计算 2 + 3 * 4")
print(response)

# 查看对话历史
print(f"历史消息数: {len(agent.get_history())}）")
```

7.2 HelloAgentsLLM扩展

本节内容将在第4.1.3节创建的HelloAgentsLLM基础上进行迭代升级。我们将把这个基础客户端，改造为一个更具适应性的模型调用中枢。本次升级主要围绕以下三个目标展开：

1. 多提供商支持：实现对OpenAI、ModelScope、智谱AI等多种主流LLM服务商的无缝切换，避免框架与特定供应商绑定。
2. 本地模型集成：引入VLLM和Ollama这两种高性能本地部署方案，作为对第3.2.3节中Hugging Face Transformers方案的生产级补充，满足数据隐私和成本控制的需求。
3. 自动检测机制：建立一套自动识别机制，使框架能根据环境信息智能推断所使用的LLM服务类型，简化用户的配置过程。

7.2.1 支持多提供商

我们之前定义的HelloAgentsLLM类，已经能够通过api_key和base_url这两个核心参数，连接任何兼容OpenAI接口的服务。这在理论上保证了通用性，但在实际应用中，不同的服务商在环境变量命名、默认API地址和推荐模型等方面都存在差异。如果每次切换服务商都需要用户手动查询并修改代码，会极大影响开发效率。为了解决这一问题，我们引入provider。其改进思路是：让HelloAgentsLLM在内部处理不同服务商的配置细节，从而为用户提供一个统一、简洁的调用体验。具体的实现细节我们将在7.2.3节“自动检测机制”中详细阐述，在这里，我们首先关注如何利用这一机制来扩展框架。

下面，我们将演示如何通过继承HelloAgentsLLM，来增加对ModelScope平台的支持。我们希望读者不仅学会如何“使用”框架，更能掌握如何“扩展”框架。直接修改已安装的库源码是一种不被推荐的做法，因为它会使后续的库升级变得困难。

(1) 创建自定义LLM类并继承

假设我们的项目目录中有一个my_llm.py文件。我们首先从hello_agents库中导入HelloAgentsLLM基类，然后创建一个名为MyLLM的新类继承它。

```
# my_llm.py
import os
from typing import Optional
from openai import OpenAI
from hello_agents import HelloAgentsLLM

class MyLLM(HelloAgentsLLM):
    """
    一个自定义的LLM客户端，通过继承增加了对ModelScope的支持。
    """
    pass # 暂时留空
```

(2) 重写__init__方法以支持新供应商

接下来，我们在MyLLM类中重写__init__方法。我们的目标是：当用户传入provider="modelscope"时，执行我们自定义的逻辑；否则，就调用父类HelloAgentsLLM的原始逻辑，使其能够继续支持OpenAI等其他内置的供应商。

```
class MyLLM(HelloAgentsLLM):
    def __init__(
        self,
        model: Optional[str] = None,
        api_key: Optional[str] = None,
        base_url: Optional[str] = None,
        provider: Optional[str] = "auto",
        **kwargs
    ):
        # 检查provider是否为我们想处理的'modelscope'
        if provider == "modelscope":
            print("正在使用自定义的 ModelScope Provider")
            self.provider = "modelscope"

            # 解析 ModelScope 的凭证
            self.api_key = api_key or os.getenv("MODELScope_API_KEY")
            self.base_url = base_url or "https://api-inference.modelscope.cn/v1/"

            # 验证凭证是否存在
            if not self.api_key:
                raise ValueError("ModelScope API key not found. Please set MODELScope_API_KEY environment variable.")

            # 设置默认模型和其他参数
            self.model = model or os.getenv("LLM_MODEL_ID") or "Qwen/Qwen2.5-VL-72B-Instruct"
            self.temperature = kwargs.get('temperature', 0.7)
            self.max_tokens = kwargs.get('max_tokens')
            self.timeout = kwargs.get('timeout', 60)

            # 使用获取的参数创建OpenAI客户端实例
            self._client = OpenAI(api_key=self.api_key, base_url=self.base_url, timeout=self.timeout)

        else:
```

```
# 如果不是 modelscope, 则完全使用父类的原始逻辑来处理
super().__init__(model=model, api_key=api_key, base_url=base_url, provider=provider, **kwargs)
```

这段代码展示了“重写”的思想：我们拦截了 provider="modelscope" 的情况并进行了特殊处理，对于其他所有情况，则通过 super().__init__(...) 交给父类，保留了原有框架的全部功能。

(3) 使用自定义的 MyLLM 类

现在，我们可以在项目的业务逻辑中，像使用原生 HelloAgentsLLM 一样使用我们自己的 MyLLM 类。

首先，在 .env 文件中配置 ModelScope 的 API 密钥：

```
# .env file
MODELSCOPE_API_KEY="your-modelscope-api-key"
```

然后，在主程序中导入并使用 MyLLM：

```
# my_main.py
from dotenv import load_dotenv
from my_llm import MyLLM # 注意:这里导入我们自己的类

# 加载环境变量
load_dotenv()

# 实例化我们重写的客户端，并指定provider
llm = MyLLM(provider="modelscope")

# 准备消息
messages = [{"role": "user", "content": "你好，请介绍一下你自己。"}]

# 启动调用，think等方法都已从父类继承，无需重写
response_stream = llm.think(messages)

# 打印响应
print("ModelScope Response:")
for chunk in response_stream:
    # chunk在my_llm库中已经打印过一遍，这里只需要pass即可
    # print(chunk, end="", flush=True)
    pass
```

通过以上步骤，我们就在不修改 hello-agents 库源码的前提下，成功为其扩展了新的功能。这种方法不仅保证了代码的整洁和可维护性，也使得未来升级 hello-agents 库时，我们的定制化功能不会丢失。

7.2.2 本地模型调用

在第 3.2.3 节，我们学习了如何使用 Hugging Face Transformers 库在本地直接运行开源模型。该方法非常适合入门学习和功能验证，但其底层实现在处理高并发请求时性能有限，通常不作为生产环境的首选方案。

为了在本地实现高性能、生产级的模型推理服务，社区涌现出了 VLLM 和 Ollama 等优秀工具。它们通过连续批处理、PagedAttention 等技术，显著提升了模型的吞吐量和运行效率，并将模型封装为兼容 OpenAI 标准的 API 服务。这意味着，我们可以将它们无缝地集成到 HelloAgentsLLM 中。

VLLM

VLLM 是一个为 LLM 推理设计的高性能 Python 库。它通过 PagedAttention 等先进技术，可以实现比标准 Transformers 实现高出数倍的吞吐量。下面是在本地部署一个 VLLM 服务的完整步骤：

首先，需要根据你的硬件环境（特别是 CUDA 版本）安装 VLLM。推荐遵循其[官方文档](#)进行安装，以避免版本不匹配问题。

```
pip install vllm
```

安装完成后，使用以下命令即可启动一个兼容 OpenAI 的 API 服务。VLLM 会自动从 Hugging Face Hub 下载指定的模型权重（如果本地不存在）。我们依然以 Qwen1.5-0.5B-Chat 模型为例：

```
# 启动 VLLM 服务，并加载 Qwen1.5-0.5B-Chat 模型
python -m vllm.entrypoints.openai.api_server \
--model Qwen/Qwen1.5-0.5B-Chat \
--host 0.0.0.0 \
--port 8000
```

服务启动后，便会在 <http://localhost:8000/v1> 地址上提供与 OpenAI 兼容的 API。

Ollama

Ollama 进一步简化了本地模型的管理和部署，它将模型下载、配置和服务启动等步骤封装到了一条命令中，非常适合快速上手。访问 Ollama [官方网站](#) 下载并安装适用于你操作系统的客户端。

安装后，打开终端，执行以下命令即可下载并运行一个模型（以 Llama 3 为例）。Ollama 会自动处理模型的下载、服务封装和硬件加速配置。

```
# 首次运行会自动下载模型，之后会直接启动服务
ollama run llama3
```

当你在终端看到模型的交互提示时，即表示服务已经在后台启动。Ollama 默认会在 <http://localhost:11434/v1> 地址上暴露 OpenAI 兼容的 API 接口。

接入 HelloAgentsLLM

由于 VLLM 和 Ollama 都遵循了行业标准 API，将它们接入 HelloAgentsLLM 的过程非常简单。我们只需在实例化客户端时，将它们视为一个新的 provider 即可。

例如，连接本地运行的 **VLLM** 服务：

```
llm_client = HelloAgentsLLM(
    provider="vllm",
    model="Qwen/Qwen1.5-0.5B-Chat", # 需与服务启动时指定的模型一致
    base_url="http://localhost:8000/v1",
    api_key="vllm" # 本地服务通常不需要真实API Key，可填任意非空字符串
)
```

或者，通过设置环境变量并让客户端自动检测，实现代码的零修改：

```
# 在 .env 文件中设置
LLM_BASE_URL="http://localhost:8000/v1"
LLM_API_KEY="vllm"

# Python 代码中直接实例化即可
llm_client = HelloAgentsLLM() # 将自动检测为 vllm
```

同理，连接本地的 **Ollama** 服务也一样简单：

```
llm_client = HelloAgentsLLM(
    provider="ollama",
    model="llama3", # 需与 `ollama run` 指定的模型一致
    base_url="http://localhost:11434/v1",
    api_key="ollama" # 本地服务同样不需要真实 Key
)
```

通过这种统一的设计，我们的智能体核心代码无需任何修改，就可以在云端 API 和本地模型之间自由切换。这为后续应用的开发、部署、成本控制以及保护数据隐私提供了极大的灵活性。

7.2.3 自动检测机制

为了尽可能减少用户的配置负担并遵循“约定优于配置”的原则，HelloAgentsLLM 内部设计了两个核心辅助方法：`_auto_detect_provider` 和 `_resolve_credentials`。它们协同工作，`_auto_detect_provider` 负责根据环境信息推断服务商，而 `_resolve_credentials` 则根据推断结果完成具体的参数配置。

`_auto_detect_provider` 方法负责根据环境信息，按照下述优先级顺序，尝试自动推断服务商：

1. 最高优先级：检查特定服务商的环境变量 这是最直接、最可靠的判断依据。框架会依次检查 `MODELSCOPE_API_KEY`, `OPENAI_API_KEY`, `ZHIPU_API_KEY` 等环境变量是否存在。一旦发现任何一个，就会立即确定对应的服务商。
2. 次高优先级：根据 `base_url` 进行判断 如果用户没有设置特定服务商的密钥，但设置了通用的 `LLM_BASE_URL`，框架会转而解析这个 URL。
 - 域名匹配：通过检查 URL 中是否包含 "api-inference.modelscope.cn", "api.openai.com" 等特征字符串来识别云服务商。
 - 端口匹配：通过检查 URL 中是否包含 :11434 (Ollama), :8000 (VLLM) 等本地服务的标准端口来识别本地部署方案。
3. 辅助判断：分析 API 密钥的格式 在某些情况下，如果上述两种方式都无法确定，框架会尝试分析通用环境变量 `LLM_API_KEY` 的格式。例如，某些服务商的 API 密钥有固定的前缀或独特的编码格式。不过，由于这种方式可能存在模糊性（例如多个服务商的密钥格式相似），因此它的优先级较低，仅作为辅助手段。

其部分关键代码如下：

```
def _auto_detect_provider(self, api_key: Optional[str], base_url: Optional[str]) -> str:
    """
    自动检测LLM提供商
    .....
    # 1. 检查特定提供商的环境变量（最高优先级）
    if os.getenv("MODELSCOPE_API_KEY"): return "modelscope"
    if os.getenv("OPENAI_API_KEY"): return "openai"
    if os.getenv("ZHIPU_API_KEY"): return "zhipu"
    # ... 其他服务商的环境变量检查

    # 获取通用的环境变量
    actual_api_key = api_key or os.getenv("LLM_API_KEY")
    actual_base_url = base_url or os.getenv("LLM_BASE_URL")

    # 2. 根据 base_url 判断
    if actual_base_url:
        base_url_lower = actual_base_url.lower()
        if "api-inference.modelscope.cn" in base_url_lower: return "modelscope"
        if "openbigmodel.cn" in base_url_lower: return "zhipu"
        if "localhost" in base_url_lower or "127.0.0.1" in base_url_lower:
            if ":11434" in base_url_lower: return "ollama"
            if ":8000" in base_url_lower: return "vllm"
        return "local" # 其他本地端口

    # 3. 根据 API 密钥格式辅助判断
```

```

if actual_api_key:
    if actual_api_key.startswith("ms-"): return "modelscope"
    # ... 其他密钥格式判断

# 4. 默认返回 'auto', 使用通用配置
return "auto"

```

一旦 provider 被确定（无论是用户指定还是自动检测），_resolve_credentials 方法便会接手处理服务商的差异化配置。它会根据 provider 的值，去主动查找对应的环境变量，并为其设置默认的 base_url。其部分关键实现如下：

```

def _resolve_credentials(self, api_key: Optional[str], base_url: Optional[str]) -> tuple[str, str]:
    """根据provider解析API密钥和base_url"""
    if self.provider == "openai":
        resolved_api_key = api_key or os.getenv("OPENAI_API_KEY") or os.getenv("LLM_API_KEY")
        resolved_base_url = base_url or os.getenv("LLM_BASE_URL") or "https://api.openai.com/v1"
        return resolved_api_key, resolved_base_url

    elif self.provider == "modelscope":
        resolved_api_key = api_key or os.getenv("MODELSCOPE_API_KEY") or os.getenv("LLM_API_KEY")
        resolved_base_url = base_url or os.getenv("LLM_BASE_URL") or "https://api-inference.modelscope.cn/v1/"
        return resolved_api_key, resolved_base_url

    # ... 其他服务商的逻辑

```

让我们通过一个简单的例子来感受自动检测带来的便利。假设一个用户想要使用本地的 Ollama 服务，他只需在 .env 文件中进行如下配置：

```

LLM_BASE_URL="http://localhost:11434/v1"
LLM_MODEL_ID="llama3"

```

他完全不需要配置 LLM_API_KEY 或在代码中指定 provider。然后，在 Python 代码中，他只需简单地实例化 HelloAgentsLLM 即可：

```

from dotenv import load_dotenv
from hello_agents import HelloAgentsLLM

load_dotenv()

# 无需传入 provider, 框架会自动检测
llm = HelloAgentsLLM()
# 框架内部日志会显示检测到 provider 为 'ollama'

# 后续调用方式完全不变
messages = [{"role": "user", "content": "你好! "}]
for chunk in llm.think(messages):
    print(chunk, end="")

```

在这个过程中，_auto_detect_provider 方法通过解析 LLM_BASE_URL 中的 "localhost" 和 :11434，成功地将 provider 推断为 "ollama"。随后，_resolve_credentials 方法会为 Ollama 设置正确的默认参数。

相比于4.1.3节的基础实现，现在的HelloAgentsLLM具有以下显著优势：

表 7.1 HelloAgentLLM不同版本特性对比



如上表7.1所示，这种演进体现了框架设计的重要原则：从简单开始，逐步完善。我们在保持接口简洁的同时，增强了功能的完整性。

7.3 框架接口实现

在上节课中，我们构建了 HelloAgentsLLM 这一核心组件，解决了与大语言模型通信的关键问题。不过它还需要一系列配套的接口和组件来处理数据流、管理配置、应对异常，并为上层应用的构建提供一个清晰、统一的结构。本节将讲述以下三个核心文件：

- message.py：定义了框架内统一的消息格式，确保了智能体与模型之间信息传递的标准化。
- config.py：提供了一个中心化的配置管理方案，使框架的行为易于调整和扩展。
- agent.py：定义了所有智能体的抽象基类（Agent），为后续实现不同类型的智能体提供了统一的接口和规范。

7.3.1 Message 类

在智能体与大语言模型的交互中，对话历史是至关重要的上下文。为了规范地管理这些信息，我们设计了一个简易 Message 类。在后续上下文工程章节中，会对其进行扩展。

```

"""消息系统"""
from typing import Optional, Dict, Any, Literal
from datetime import datetime
from pydantic import BaseModel

# 定义消息角色的类型，限制其取值
MessageRole = Literal["user", "assistant", "system", "tool"]

class Message(BaseModel):
    """消息类"""

    content: str
    role: MessageRole
    timestamp: datetime = None

```

```

metadata: Optional[Dict[str, Any]] = None

def __init__(self, content: str, role: MessageRole, **kwargs):
    super().__init__(
        content=content,
        role=role,
        timestamp=kwargs.get('timestamp', datetime.now()),
        metadata=kwargs.get('metadata', {}))
)

def to_dict(self) -> Dict[str, Any]:
    """转换为字典格式 (OpenAI API格式) """
    return {
        "role": self.role,
        "content": self.content
    }

def __str__(self) -> str:
    return f"[{self.role}] {self.content}"

```

该类的设计有几个关键点。首先，我们通过 `typing.Literal` 将 `role` 字段的取值严格限制为 "user", "assistant", "system", "tool" 四种，这直接对应 OpenAI API 的规范，保证了类型安全。除了 `content` 和 `role` 这两个核心字段外，我们还增加了 `timestamp` 和 `metadata`，为日志记录和未来功能扩展预留了空间。最后，`to_dict()` 方法是其核心功能之一，负责将内部使用的 `Message` 对象转换为与 OpenAI API 兼容的字典格式，体现了“对内丰富，对外兼容”的设计原则。

7.3.2 Config 类

`Config` 类的职责是将代码中硬编码配置参数集中起来，并支持从环境变量中读取。

```

"""配置管理"""
import os
from typing import Optional, Dict, Any
from pydantic import BaseModel

class Config(BaseModel):
    """HelloAgents配置类"""

    # LLM配置
    default_model: str = "gpt-3.5-turbo"
    default_provider: str = "openai"
    temperature: float = 0.7
    max_tokens: Optional[int] = None

    # 系统配置
    debug: bool = False
    log_level: str = "INFO"

    # 其他配置
    max_history_length: int = 100

    @classmethod
    def from_env(cls) -> "Config":
        """从环境变量创建配置"""
        return cls(
            debug=os.getenv("DEBUG", "false").lower() == "true",
            log_level=os.getenv("LOG_LEVEL", "INFO"),
            temperature=float(os.getenv("TEMPERATURE", "0.7")),
            max_tokens=int(os.getenv("MAX_TOKENS")) if os.getenv("MAX_TOKENS") else None,
        )

    def to_dict(self) -> Dict[str, Any]:
        """转换为字典"""
        return self.dict()

```

首先，我们将配置项按逻辑划分为 LLM 配置、系统配置等，使结构一目了然。其次，每个配置项都设有合理的默认值，保证了框架在零配置下也能工作。最核心的是 `from_env()` 类方法，它允许用户通过设置环境变量来覆盖默认配置，无需修改代码，这在部署到不同环境时尤其有用。

7.3.3 Agent 抽象基类

`Agent` 类是整个框架的顶层抽象。它定义了一个智能体应该具备的通用行为和属性，但并不关心具体的实现方式。我们通过 Python 的 `abc` (Abstract Base Classes) 模块来实现它，这强制所有具体的智能体实现（如后续章节的 `SimpleAgent`, `ReActAgent` 等）都必须遵循同一个“接口”。

```

"""Agent基类"""
from abc import ABC, abstractmethod
from typing import Optional, Any
from .message import Message
from .llm import HelloAgentsLLM
from .config import Config

class Agent(ABC):
    """Agent基类"""

    def __init__(
        self,
        name: str,
        llm: HelloAgentsLLM,
        system_prompt: Optional[str] = None,
        config: Optional[Config] = None
    ):
        self.name = name

```

```

self.llm = llm
self.system_prompt = system_prompt
self.config = config or Config()
self._history: list[Message] = []

@abstractmethod
def run(self, input_text: str, **kwargs) -> str:
    """运行Agent"""
    pass

def add_message(self, message: Message):
    """添加消息到历史记录"""
    self._history.append(message)

def clear_history(self):
    """清空历史记录"""
    self._history.clear()

def get_history(self) -> list[Message]:
    """获取历史记录"""
    return self._history.copy()

def __str__(self) -> str:
    return f"Agent(name={self.name}, provider={self.llm.provider})"

```

该类的设计体现了面向对象中的抽象原则。首先，它通过继承 ABC 被定义为一个不能直接实例化的抽象类。其构造函数 `__init__` 清晰地定义了 Agent 的核心依赖：名称、LLM 实例、系统提示词和配置。最重要的部分是使用 `@abstractmethod` 装饰的 `run` 方法，它强制所有子类必须实现此方法，从而保证了所有智能体都有统一的执行入口。此外，基类还提供了通用的历史记录管理方法，这些方法与 `Message` 类协同工作，体现了组件间的联系。

至此，我们已经完成了 `HelloAgents` 框架核心基础组件的设计与实现。

7.4 Agent范式的框架化实现

本节内容将在第四章构建的三种经典Agent范式（ReAct、Plan-and-Solve、Reflection）基础上进行框架化重构，并新增SimpleAgent作为基础对话范式。我们将把这些独立的Agent实现，改造为基于统一架构的框架组件。本次重构主要围绕以下三个核心目标展开：

1. 提示词工程的系统性提升：对第四章中的提示词进行深度优化，从特定任务导向转向通用化设计，同时增强格式约束和角色定义。
2. 接口与格式的标准化统一：建立统一的Agent基类和标准化的运行接口，所有Agent都遵循相同的初始化参数、方法签名和历史管理机制。
3. 高度可配置的自定义能力：支持用户自定义提示词模板、配置参数和执行策略。

7.4.1 SimpleAgent

SimpleAgent是最基础的Agent实现，它展示了如何在框架基础上构建一个完整的对话智能体。我们将通过继承框架基类来重写SimpleAgent。首先，在你的项目目录中创建一个`my_simple_agent.py`文件：

```

# my_simple_agent.py
from typing import Optional, Iterator
from hello_agents import SimpleAgent, HelloAgentsLLM, Config, Message

class MySimpleAgent(SimpleAgent):
    """
    重写的简单对话Agent
    展示如何基于框架基类构建自定义Agent
    """

    def __init__(
        self,
        name: str,
        llm: HelloAgentsLLM,
        system_prompt: Optional[str] = None,
        config: Optional[Config] = None,
        tool_registry: Optional['ToolRegistry'] = None,
        enable_tool_calling: bool = True
    ):
        super().__init__(name, llm, system_prompt, config)
        self.tool_registry = tool_registry
        self.enable_tool_calling = enable_tool_calling and tool_registry is not None
        print(f"✅ {name} 初始化完成，工具调用: {'启用' if self.enable_tool_calling else '禁用'}")

```

接下来，我们需要重写Agent基类的抽象方法`run`。SimpleAgent支持可选的工具调用功能，也方便后续章节的扩展：

```

# 继续在 my_simple_agent.py 中添加
import re

class MySimpleAgent(SimpleAgent):
    # ... 前面的 __init__ 方法

    def run(self, input_text: str, max_tool_iterations: int = 3, **kwargs) -> str:
        """
        重写的运行方法 - 实现简单对话逻辑，支持可选工具调用
        """

        print(f"🤖 {self.name} 正在处理: {input_text}")

        # 构建消息列表
        messages = []

```

```

# 添加系统消息（可能包含工具信息）
enhanced_system_prompt = self._get_enhanced_system_prompt()
messages.append({"role": "system", "content": enhanced_system_prompt})

# 添加历史消息
for msg in self._history:
    messages.append({"role": msg.role, "content": msg.content})

# 添加当前用户消息
messages.append({"role": "user", "content": input_text})

# 如果没有启用工具调用，使用简单对话逻辑
if not self.enable_tool_calling:
    response = self.llm.invoke(messages, **kwargs)
    self.add_message(Message(input_text, "user"))
    self.add_message(Message(response, "assistant"))
    print(f"✅ {self.name} 响应完成")
    return response

# 支持多轮工具调用的逻辑
return self._run_with_tools(messages, input_text, max_tool_iterations, **kwargs)

def _get_enhanced_system_prompt(self) -> str:
    """构建增强的系统提示词，包含工具信息"""
    base_prompt = self.system_prompt or "你是一个有用的AI助手。"

    if not self.enable_tool_calling or not self.tool_registry:
        return base_prompt

    # 获取工具描述
    tools_description = self.tool_registry.get_tools_description()
    if not tools_description or tools_description == "暂无可用工具":
        return base_prompt

    tools_section = "\n\n## 可用工具\n"
    tools_section += "你可以使用以下工具来帮助回答问题:\n"
    tools_section += tools_description + "\n"

    tools_section += "\n## 工具调用格式\n"
    tools_section += "当需要使用工具时，请使用以下格式:\n"
    tools_section += ` [TOOL_CALL:{tool_name}:{parameters}]` '\n'
    tools_section += "例如: [TOOL_CALL:search:Python编程] 或 [TOOL_CALL:memory:recall=用户信息]\n\n"
    tools_section += "工具调用结果会自动插入到对话中，然后你可以基于结果继续回答。'\n"

    return base_prompt + tools_section

```

现在我们实现工具调用的核心逻辑：

```

# 继续在 my_simple_agent.py 中添加
class MySimpleAgent(SimpleAgent):
    # ... 前面的方法

    def _run_with_tools(self, messages: list, input_text: str, max_tool_iterations: int, **kwargs) -> str:
        """支持工具调用的运行逻辑"""
        current_iteration = 0
        final_response = ""

        while current_iteration < max_tool_iterations:
            # 调用LLM
            response = self.llm.invoke(messages, **kwargs)

            # 检查是否有工具调用
            tool_calls = self._parse_tool_calls(response)

            if tool_calls:
                print(f"⚠ 检测到 {len(tool_calls)} 个工具调用")
                # 执行所有工具调用并收集结果
                tool_results = []
                clean_response = response

                for call in tool_calls:
                    result = self._execute_tool_call(call['tool_name'], call['parameters'])
                    tool_results.append(result)
                    # 从响应中移除工具调用标记
                    clean_response = clean_response.replace(call['original'], "")

                # 构建包含工具结果的消息
                messages.append({"role": "assistant", "content": clean_response})

                # 添加工具结果
                tool_results_text = "\n\n".join(tool_results)
                messages.append({"role": "user", "content": f"工具执行结果:\n{tool_results_text}\n\n请基于这些结果给出完整的回答。"})

                current_iteration += 1
                continue

            # 没有工具调用，这是最终回答
            final_response = response
            break

        # 如果超过最大迭代次数，获取最后一次回答

```

```

if current_iteration >= max_tool_iterations and not final_response:
    final_response = self.llm.invoke(messages, **kwargs)

# 保存到历史记录
self.add_message(Message(input_text, "user"))
self.add_message(Message(final_response, "assistant"))
print(f"✅ {self.name} 响应完成")

return final_response

def _parse_tool_calls(self, text: str) -> list:
    """解析文本中的工具调用"""
    pattern = r'\[TOOL_CALL:([^:]+):([^\]]+)\]>'
    matches = re.findall(pattern, text)

    tool_calls = []
    for tool_name, parameters in matches:
        tool_calls.append({
            'tool_name': tool_name.strip(),
            'parameters': parameters.strip(),
            'original': f'[TOOL_CALL:{tool_name}:{parameters}]'
        })

    return tool_calls

def _execute_tool_call(self, tool_name: str, parameters: str) -> str:
    """执行工具调用"""
    if not self.tool_registry:
        return f"❌ 错误:未配置工具注册表"

    try:
        # 智能参数解析
        if tool_name == 'calculator':
            # 计算器工具直接传入表达式
            result = self.tool_registry.execute_tool(tool_name, parameters)
        else:
            # 其他工具使用智能参数解析
            param_dict = self._parse_tool_parameters(tool_name, parameters)
            tool = self.tool_registry.get_tool(tool_name)
            if not tool:
                return f"❌ 错误:未找到工具 '{tool_name}'"
            result = tool.run(param_dict)

        return f"🔧 工具 {tool_name} 执行结果:\n{result}"

    except Exception as e:
        return f"❌ 工具调用失败:{str(e)}"

def _parse_tool_parameters(self, tool_name: str, parameters: str) -> dict:
    """智能解析工具参数"""
    param_dict = {}

    if '=' in parameters:
        # 格式: key=value 或 action=search,query=Python
        if ',' in parameters:
            # 多个参数:action=search,query=Python,limit=3
            pairs = parameters.split(',')
            for pair in pairs:
                if '=' in pair:
                    key, value = pair.split('=', 1)
                    param_dict[key.strip()] = value.strip()
                else:
                    # 单个参数:key=value
                    key, value = parameters.split('=', 1)
                    param_dict[key.strip()] = value.strip()
        else:
            # 直接传入参数, 根据工具类型智能推断
            if tool_name == 'search':
                param_dict = {'query': parameters}
            elif tool_name == 'memory':
                param_dict = {'action': 'search', 'query': parameters}
            else:
                param_dict = {'input': parameters}

    return param_dict

```

我们还可以为自定义Agent添加流式响应功能和便利方法:

```

# 继续在 my_simple_agent.py 中添加
class MySimpleAgent(SimpleAgent):
    # ... 前面的方法

    def stream_run(self, input_text: str, **kwargs) -> Iterator[str]:
        """
        自定义的流式运行方法
        """
        print(f"💡 {self.name} 开始流式处理: {input_text}")

        messages = []

        if self.system_prompt:
            messages.append({"role": "system", "content": self.system_prompt})

```

```

for msg in self._history:
    messages.append({"role": msg.role, "content": msg.content})

messages.append({"role": "user", "content": input_text})

# 流式调用LLM
full_response = ""
print("📝 实时响应: ", end="")
for chunk in self.llm.stream_invoke(messages, **kwargs):
    full_response += chunk
    print(chunk, end="", flush=True)
    yield chunk

print() # 换行

# 保存完整对话到历史记录
self.add_message(Message(input_text, "user"))
self.add_message(Message(full_response, "assistant"))
print(f"✅ {self.name} 流式响应完成")

def add_tool(self, tool) -> None:
    """添加工具到Agent (便利方法)"""
    if not self.tool_registry:
        from hello_agents import ToolRegistry
        self.tool_registry = ToolRegistry()
        self.enable_tool_calling = True

    self.tool_registry.register_tool(tool)
    print(f"🔧 工具 '{tool.name}' 已添加")

def has_tools(self) -> bool:
    """检查是否有可用工具"""
    return self.enable_tool_calling and self.tool_registry is not None

def remove_tool(self, tool_name: str) -> bool:
    """移除工具 (便利方法)"""
    if self.tool_registry:
        self.tool_registry.unregister(tool_name)
        return True
    return False

def list_tools(self) -> list:
    """列出所有可用工具"""
    if self.tool_registry:
        return self.tool_registry.list_tools()
    return []

```

创建一个测试文件test_simple_agent.py:

```

# test_simple_agent.py
from dotenv import load_dotenv
from hello_agents import HelloAgentsLLM, ToolRegistry
from hello_agents.tools import CalculatorTool
from my_simple_agent import MySimpleAgent

# 加载环境变量
load_dotenv()

# 创建LLM实例
llm = HelloAgentsLLM()

# 测试1:基础对话Agent (无工具)
print("==> 测试1:基础对话 ==>")
basic_agent = MySimpleAgent(
    name="基础助手",
    llm=llm,
    system_prompt="你是一个友好的AI助手, 请用简洁明了的方式回答问题。"
)

response1 = basic_agent.run("你好, 请介绍一下自己")
print(f"基础对话响应: {response1}\n")

# 测试2:带工具的Agent
print("==> 测试2:工具增强对话 ==>")
tool_registry = ToolRegistry()
calculator = CalculatorTool()
tool_registry.register_tool(calculator)

enhanced_agent = MySimpleAgent(
    name="增强助手",
    llm=llm,
    system_prompt="你是一个智能助手, 可以使用工具来帮助用户。",
    tool_registry=tool_registry,
    enable_tool_calling=True
)

response2 = enhanced_agent.run("请帮我计算 15 * 8 + 32")
print(f"工具增强响应: {response2}\n")

# 测试3:流式响应

```

```

print("== 测试3:流式响应 ==")
print("流式响应: ", end="")
for chunk in basic_agent.stream_run("请解释什么是人工智能"):
    pass # 内容已在stream_run中实时打印

# 测试4:动态添加工具
print("\n== 测试4:动态工具管理 ==")
print(f"添加工具前: {basic_agent.has_tools()}")
basic_agent.add_tool(calculator)
print(f"添加工具后: {basic_agent.has_tools()}")
print(f"可用工具: {basic_agent.list_tools()}")

# 查看对话历史
print(f"\n对话历史: {len(basic_agent.get_history())} 条消息")

```

在本节中，我们通过继承 Agent 基类，成功构建了一个功能完备且遵循框架规范的基础对话智能体 MySimpleAgent。它不仅支持基础对话，还具备可选的工具调用能力、流式响应和便利的工具管理方法。

7.4.2 ReActAgent

框架化的 ReActAgent 在保持核心逻辑不变的同时，提升了代码的组织性和可维护性，主要是通过提示词优化和与框架工具系统的集成。

(1) 提示词模板的改进

保持了原有的格式要求，强调“每次只能执行一个步骤”，避免混乱，并明确了两种Action的使用场景。

```
MY_REACT_PROMPT = """你是一个具备推理和行动能力的AI助手。你可以通过思考分析问题，然后调用合适的工具来获取信息，最终给出准确的答案。
```

```
## 可用工具
{tools}
```

```
## 工作流程
```

请严格按照以下格式进行回应，每次只能执行一个步骤：

Thought: 分析当前问题，思考需要什么信息或采取什么行动。

Action: 选择一个行动，格式必须是以下之一：

- `{{tool_name}}[{{tool_input}}]` - 调用指定工具
- `Finish[最终答案]` - 当你有足够的信息给出最终答案时

```
## 重要提醒
```

1. 每次回应必须包含Thought和Action两部分
2. 工具调用的格式必须严格遵循：工具名[参数]
3. 只有当你确信有足够的信息回答问题时，才使用Finish
4. 如果工具返回的信息不够，继续使用其他工具或相同工具的不同参数

```
## 当前任务
```

```
**Question:** {question}
```

```
## 执行历史
```

```
{history}
```

现在开始你的推理和行动：

```
.....
```

(2) 重写ReActAgent的完整实现

创建my_react_agent.py文件来重写ReActAgent：

```

# my_react_agent.py
import re
from typing import Optional, List, Tuple
from hello_agents import ReActAgent, HelloAgentsLLM, Config, Message, ToolRegistry

class MyReActAgent(ReActAgent):
    """
    重写的ReAct Agent - 推理与行动结合的智能体
    """

    def __init__(
        self,
        name: str,
        llm: HelloAgentsLLM,
        tool_registry: ToolRegistry,
        system_prompt: Optional[str] = None,
        config: Optional[Config] = None,
        max_steps: int = 5,
        custom_prompt: Optional[str] = None
    ):
        super().__init__(name, llm, system_prompt, config)
        self.tool_registry = tool_registry
        self.max_steps = max_steps
        self.current_history: List[str] = []
        self.prompt_template = custom_prompt if custom_prompt else MY_REACT_PROMPT
        print(f"✅ {name} 初始化完成，最大步数: {max_steps}")

```

其初始化参数的含义如下：

- name: Agent的名称。
- llm: HelloAgentsLLM的实例，负责与大语言模型通信。
- tool_registry: ToolRegistry的实例，用于管理和执行Agent可用的工具。
- system_prompt: 系统提示词，用于设定Agent的角色和行为准则。
- config: 配置对象，用于传递框架级的设置。
- max_steps: ReAct循环的最大执行步数，防止无限循环。
- custom_prompt: 自定义的提示词模板，用于替换默认的ReAct提示词。

框架化的ReActAgent将执行流程分解为清晰的步骤：

```
def run(self, input_text: str, **kwargs) -> str:
    """运行ReAct Agent"""
    self.current_history = []
    current_step = 0

    print(f"\n🤖 {self.name} 开始处理问题: {input_text}")

    while current_step < self.max_steps:
        current_step += 1
        print(f"\n--- 第 {current_step} 步 ---")

        # 1. 构建提示词
        tools_desc = self.tool_registry.get_tools_description()
        history_str = "\n".join(self.current_history)
        prompt = self.prompt_template.format(
            tools=tools_desc,
            question=input_text,
            history=history_str
        )

        # 2. 调用LLM
        messages = [{"role": "user", "content": prompt}]
        response_text = self.llm.invoke(messages, **kwargs)

        # 3. 解析输出
        thought, action = self._parse_output(response_text)

        # 4. 检查完成条件
        if action and action.startswith("Finish"):
            final_answer = self._parse_action_input(action)
            self.add_message(Message(input_text, "user"))
            self.add_message(Message(final_answer, "assistant"))
            return final_answer

        # 5. 执行工具调用
        if action:
            tool_name, tool_input = self._parse_action(action)
            observation = self.tool_registry.execute_tool(tool_name, tool_input)
            self.current_history.append(f"Action: {action}")
            self.current_history.append(f"Observation: {observation}")

    # 达到最大步数
    final_answer = "抱歉，我无法在限定步数内完成这个任务。"
    self.add_message(Message(input_text, "user"))
    self.add_message(Message(final_answer, "assistant"))
    return final_answer
```

通过以上重构，我们将ReAct范式成功地集成到了框架中。核心改进在于利用了统一的ToolRegistry接口，并通过一个可配置、格式更严谨的提示词模板，提升了智能体执行思考-行动循环的稳定性。对于ReAct的测试案例，由于需要调用工具，所以统一放在文末提供测试代码。

7.4.3 ReflectionAgent

由于这几类Agent已经在第四章实现过核心逻辑，所以这里只给出对应的Prompt。与第四章专门针对代码生成的提示词不同，框架化的版本采用了通用化设计，使其适用于文本生成、分析、创作等多种场景，并通过custom_prompts参数支持用户深度定制。

```
DEFAULT_PROMPTS = {
    "initial": """
请根据以下要求完成任务:

任务: {task}

请提供一个完整、准确的回答。
""",
    "reflect": """
请仔细审查以下回答，并找出可能的问题或改进空间:

# 原始任务:
{task}

# 当前回答:
{content}

请分析这个回答的质量，指出不足之处，并提出具体的改进建议。
如果回答已经很好，请回答"无需改进"。
""",
    "refine": """
请根据反馈意见改进你的回答:
"""
}
```

```
# 原始任务:
{task}

# 上一轮回答:
{last_attempt}

# 反馈意见:
{feedback}

请提供一个改进后的回答。
"""
}
```

你可以尝试根据第四章的代码，以及上文ReAct的实现，构建出自己的MyReflectionAgent。下面提供一个测试代码供验证想法。

```
# test_reflection_agent.py
from dotenv import load_dotenv
from hello_agents import HelloAgentsLLM
from my_reflection_agent import MyReflectionAgent

load_dotenv()
llm = HelloAgentsLLM()

# 使用默认通用提示词
general_agent = MyReflectionAgent(name="我的反思助手", llm=llm)

# 使用自定义代码生成提示词（类似第四章）
code_prompts = {
    "initial": "你是Python专家，请编写函数:{task}",
    "reflect": "请审查代码的算法效率:\n任务:{task}\n代码:{content}",
    "refine": "请根据反馈优化代码:\n任务:{task}\n反馈:{feedback}"
}
code_agent = MyReflectionAgent(
    name="我的代码生成助手",
    llm=llm,
    custom_prompts=code_prompts
)

# 测试使用
result = general_agent.run("写一篇关于人工智能发展历程的简短文章")
print(f"最终结果: {result}")
```

7.4.4 PlanAndSolveAgent

与第四章自由文本的计划输出不同，框架化版本强制要求Planner以Python列表的格式输出计划，并提供了完整的异常处理机制，确保了后续步骤能够稳定执行。框架化的Plan-and-Solve提示词：

```
# 默认规划器提示词模板
DEFAULT_PLANNER_PROMPT = """
你是一个顶级的AI规划专家。你的任务是将用户提出的复杂问题分解成一个由多个简单步骤组成的行动计划。
请确保计划中的每个步骤都是一个独立的、可执行的子任务，并且严格按照逻辑顺序排列。
你的输出必须是一个Python列表，其中每个元素都是一个描述子任务的字符串。

问题: {question}

请严格按照以下格式输出你的计划:
```python
["步骤1", "步骤2", "步骤3", ...]
```
"""

# 默认执行器提示词模板
DEFAULT_EXECUTOR_PROMPT = """
你是一位顶级的AI执行专家。你的任务是严格按照给定的计划，一步步地解决问题。
你将收到原始问题、完整的计划、以及到目前为止已经完成的步骤和结果。
请你专注于解决“当前步骤”，并仅输出该步骤的最终答案，不要输出任何额外的解释或对话。

# 原始问题:
{question}

# 完整计划:
{plan}

# 历史步骤与结果:
{history}

# 当前步骤:
{current_step}

请仅输出针对“当前步骤”的回答:
"""

这一节仍然给出一个综合测试文件test_plan_solve_agent.py，可以自行设计实现。
```

```
# test_plan_solve_agent.py
from dotenv import load_dotenv
from hello_agents.core.llm import HelloAgentsLLM
```

2/9/26, 1:56 PM

```
from my_plan_solve_agent import MyPlanAndSolveAgent

# 加载环境变量
load_dotenv()

# 创建LLM实例
llm = HelloAgentsLLM()

# 创建自定义PlanAndSolveAgent
agent = MyPlanAndSolveAgent(
    name="我的规划执行助手",
    llm=llm
)

# 测试复杂问题
question = "一个水果店周一卖出了15个苹果。周二卖出的苹果数量是周一的两倍。周三卖出的数量比周二少了5个。请问这三天总共卖出了多少个苹果？"

result = agent.run(question)
print(f"\n最终结果: {result}")

# 查看对话历史
print(f"对话历史: {len(agent.get_history())} 条消息")
```

在最后可以补充一款新的提示词，可以尝试实现custom_prompt载入自定义提示词。

```
# 创建专门用于数学问题的自定义提示词
```

```
math_prompts = {
    "planner": """"
```

你是数学问题规划专家。请将数学问题分解为计算步骤：

```
问题: {question}
```

输出格式：

```
python
["计算步骤1", "计算步骤2", "求总和"]
```

```
""", "executor": """
```

你是数学计算专家。请计算当前步骤：

```
问题: {question}
```

```
计划: {plan}
```

```
历史: {history}
```

```
当前步骤: {current_step}
```

请只输出数值结果：

```
""",
```

```
}
```

```
# 使用自定义提示词创建数学专用Agent
```

```
math_agent = MyPlanAndSolveAgent(
```

```
    name="数学计算助手",
    llm=llm,
```

```
    custom_prompts=math_prompts
)
```

```
# 测试数学问题
```

```
math_result = math_agent.run(question)
```

```
print(f"数学专用Agent结果: {math_result}")
```

如表7.2所示，通过这种框架化的重构，我们不仅保持了第四章中各种Agent范式的核心功能，还大幅提升了代码的组织性、可维护性和扩展性。所有Agent现在都共享统一的基础架构，同时保持了各自的特色和优势。

表 7.2 Agent不同章节实现对比

| 功能 | 1.0 版本 | 2.0 版本 | 3.0 版本 | 4.0 版本 |
|------------|--------|--------|----------|-------------------|
| Agent 架构 | 单线程单任务 | 多线程多任务 | 异步多线程多任务 | 完全模块化，支持多种Agent范式 |
| Agent 实现 | 直接调用 | 通过工具构建 | 通过工具构建 | 通过自定义提示词构建 |
| Agent 扩展性 | 低 | 中等 | 高 | 非常高 |
| Agent 可维护性 | 低 | 中等 | 高 | 非常高 |

7.4.5 FunctionCallAgent

FunctionCallAgent是hello-agents在0.2.8之后引入的Agent，它基于OpenAI原生函数调用机制的Agent，展示了如何使用OpenAI的函数调用机制来构建Agent。它支持以下功能：

- _build_tool_schemas:通过工具的description构建OpenAI的function calling schema
- _extract_message_content:从OpenAI的响应中提取文本
- _parse_function_call_arguments:解析模型返回的JSON字符串参数
- _convert_parameter_types:转换参数类型

这些功能可以使其具备原生的OpenAI Function Calling的能力，对比使用prompt约束的方式，具备更强的鲁棒性。

```
def _invoke_with_tools(self, messages: list[dict[str, Any]], tools: list[dict[str, Any]], tool_choice: Union[str, dict], **kwargs):
    """调用底层OpenAI客户端执行函数调用"""
    client = getattr(self.llm, "_client", None)
    if client is None:
        raise RuntimeError("HelloAgentsLLM 未正确初始化客户端，无法执行函数调用。")
```

```

client_kw_args = dict(kwargs)
client_kw_args.setdefault("temperature", self.llm.temperature)
if self.llm.max_tokens is not None:
    client_kw_args.setdefault("max_tokens", self.llm.max_tokens)

return client.chat.completions.create(
    model=self.llm.model,
    messages=messages,
    tools=tools,
    tool_choice=tool_choice,
    **client_kw_args,
)

```

#内部逻辑是对Openai 原生的functioncall作再封装

```

#OpenAI 原生functioncall示例
from openai import OpenAI
client = OpenAI()

tools = [
{
    "type": "function",
    "function": {
        "name": "get_current_weather",
        "description": "Get the current weather in a given location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "The city and state, e.g. San Francisco, CA",
                },
                "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
            },
            "required": ["location"],
        },
    }
},
]
messages = [{"role": "user", "content": "What's the weather like in Boston today?"}]
completion = client.chat.completions.create(
    model="gpt-5",
    messages=messages,
    tools=tools,
    tool_choice="auto"
)

```

```

print(completion)

```

7.5 工具系统

本节内容将在前面构建的Agent基础架构上，深入探讨工具系统的设计与实现。我们将从基础设施建设开始，逐步深入到自定义开发设计。本节的学习目标围绕以下三个核心方面展开：

1. 统一的工具抽象与管理：建立标准化的Tool基类和ToolRegistry注册机制，为工具的开发、注册、发现和执行提供统一的基础设施。
2. 实战驱动的工具开发：以数学计算工具为案例，展示如何设计和实现自定义工具，让读者掌握工具开发的完整流程。
3. 高级整合与优化策略：通过多源搜索工具的设计，展示如何整合多个外部服务，实现智能后端选择、结果合并和容错处理，体现工具系统在复杂场景下的设计思维。

7.5.1 工具基类与注册机制设计

在构建可扩展的工具系统时，我们需要首先建立一套标准化的基础设施。这套基础设施包括Tool基类、ToolRegistry注册表，以及工具管理机制。

(1) Tool基类的抽象设计

Tool基类是整个工具系统的核心抽象，它定义了所有工具必须遵循的接口规范：

```

class Tool(ABC):
    """工具基类"""

    def __init__(self, name: str, description: str):
        self.name = name
        self.description = description

    @abstractmethod
    def run(self, parameters: Dict[str, Any]) -> str:
        """执行工具"""
        pass

    @abstractmethod
    def get_parameters(self) -> List[ToolParameter]:
        """获取工具参数定义"""
        pass

```

这个设计体现了面向对象设计的核心思想：通过统一的run方法接口，所有工具都能以一致的方式执行，接受字典参数并返回字符串结果，确保了框架的一致性。同时，工具具备了自描述能力，通过get_parameters方法能够清晰地告诉调用者自己需要什么参数，这种内省机制为自动化文档生成和参数验证提供了

基础。而name和description等元数据的设计，则让工具系统具备了良好的可发现性和可理解性。

(2) ToolParameter参数定义系统

为了支持复杂的参数验证和文档生成，我们设计了ToolParameter类：

```
class ToolParameter(BaseModel):
    """工具参数定义"""
    name: str
    type: str
    description: str
    required: bool = True
    default: Any = None
```

这种设计让工具能够精确描述自己的参数需求，支持类型检查、默认值设置和文档自动生成。

(3) ToolRegistry注册表的实现

ToolRegistry是工具系统的管理中枢，它提供了工具的注册、发现、执行等核心功能，在这一节我们主要用到以下功能：

```
class ToolRegistry:
    """HelloAgents工具注册表"""

    def __init__(self):
        self._tools: dict[str, Tool] = {}
        self._functions: dict[str, dict[str, Any]] = {}

    def register_tool(self, tool: Tool):
        """注册Tool对象"""
        if tool.name in self._tools:
            print(f"⚠ 警告: 工具 '{tool.name}' 已存在, 将被覆盖。")
        self._tools[tool.name] = tool
        print(f"✓ 工具 '{tool.name}' 已注册。")

    def register_function(self, name: str, description: str, func: Callable[[str], str]):
        """直接注册函数作为工具 (简便方式)

        Args:
            name: 工具名称
            description: 工具描述
            func: 工具函数, 接受字符串参数, 返回字符串结果
        """
        if name in self._functions:
            print(f"⚠ 警告: 工具 '{name}' 已存在, 将被覆盖。")

        self._functions[name] = {
            "description": description,
            "func": func
        }
        print(f"✓ 工具 '{name}' 已注册。")
```

ToolRegistry支持两种注册方式：

1. Tool对象注册：适合复杂工具，支持完整的参数定义和验证
2. 函数直接注册：适合简单工具，快速集成现有函数

(4) 工具发现与管理机制

注册表提供了丰富的工具管理功能：

```
def get_tools_description(self) -> str:
    """获取所有可用工具的格式化描述字符串"""
    descriptions = []

    # Tool对象描述
    for tool in self._tools.values():
        descriptions.append(f"- {tool.name}: {tool.description}")

    # 函数工具描述
    for name, info in self._functions.items():
        descriptions.append(f"- {name}: {info['description']}")

    return "\n".join(descriptions) if descriptions else "暂无可用工具"
```

这个方法生成的描述字符串可以直接用于构建Agent的提示词，让Agent了解可用的工具。

```
def to_openai_schema(self) -> Dict[str, Any]:
    """转换为 OpenAI function calling schema 格式

    用于 FunctionCallAgent, 使工具能够被 OpenAI 原生 function calling 使用

    Returns:
        符合 OpenAI function calling 标准的 schema
    """
    parameters = self.get_parameters()
```

```

# 构建 properties
properties = {}
required = []

for param in parameters:
    # 基础属性定义
    prop = {
        "type": param.type,
        "description": param.description
    }

    # 如果有默认值，添加到描述中 (OpenAI schema 不支持 default 字段)
    if param.default is not None:
        prop["description"] = f"{param.description} (默认: {param.default})"

    # 如果是数组类型，添加 items 定义
    if param.type == "array":
        prop["items"] = {"type": "string"} # 默认字符串数组

    properties[param.name] = prop

    # 收集必需参数
    if param.required:
        required.append(param.name)

return {
    "type": "function",
    "function": {
        "name": self.name,
        "description": self.description,
        "parameters": {
            "type": "object",
            "properties": properties,
            "required": required
        }
    }
}
}

```

这个方法生成的schema可以直接用于原生的OpenAI SDK的工具调用。

7.5.2 自定义工具开发

有了基础设施后，我们来看看如何开发一个完整的自定义工具。数学计算工具是一个很好的例子，因为它简单直观，最直接的方式是使用ToolRegistry的函数注册功能。

让我们创建一个自定义的数学计算工具。首先，在你的项目目录中创建my_calculator_tool.py:

```

# my_calculator_tool.py
import ast
import operator
import math
from hello_agents import ToolRegistry

def my_calculate(expression: str) -> str:
    """简单的数学计算函数"""
    if not expression.strip():
        return "计算表达式不能为空"

    # 支持的基本运算
    operators = {
        ast.Add: operator.add,      # +
        ast.Sub: operator.sub,     # -
        ast.Mult: operator.mul,   # *
        ast.Div: operator.truediv, # /
    }

    # 支持的基本函数
    functions = {
        'sqrt': math.sqrt,
        'pi': math.pi,
    }

    try:
        node = ast.parse(expression, mode='eval')
        result = _eval_node(node.body, operators, functions)
        return str(result)
    except:
        return "计算失败，请检查表达式格式"

def _eval_node(node, operators, functions):
    """简化的表达式求值"""
    if isinstance(node, ast.Constant):
        return node.value
    elif isinstance(node, ast.BinOp):
        left = _eval_node(node.left, operators, functions)
        right = _eval_node(node.right, operators, functions)
        op = operators.get(type(node.op))
        return op(left, right)
    elif isinstance(node, ast.Call):
        func_name = node.func.id
        if func_name in functions:
            args = [_eval_node(arg, operators, functions) for arg in node.args]

```

2/9/26, 1:56 PM

```
        return functions[func_name](*args)
    elif isinstance(node, ast.Name):
        if node.id in functions:
            return functions[node.id]

def create_calculator_registry():
    """创建包含计算器的工具注册表"""
    registry = ToolRegistry()

    # 注册计算器函数
    registry.register_function(
        name="my_calculator",
        description="简单的数学计算工具，支持基本运算(+,-,*,/)和sqrt函数",
        func=my_calculate
    )

    return registry
```

工具不仅支持基本的四则运算，还涵盖了常用的数学函数和常数，满足了大多数计算场景的需求。你也可以自己扩展这个文件，制作一个更加完备的计算函数。我们提供一个测试文件`test_my_calculator.py`帮助你验证功能实现：

```
# test_my_calculator.py
from dotenv import load_dotenv
from my_calculator_tool import create_calculator_registry

# 加载环境变量
load_dotenv()

def test_calculator_tool():
    """测试自定义计算器工具"""

    # 创建包含计算器的注册表
    registry = create_calculator_registry()

    print("🧪 测试自定义计算器工具\n")

    # 简单测试用例
    test_cases = [
        "2 + 3",           # 基本加法
        "10 - 4",          # 基本减法
        "5 * 6",          # 基本乘法
        "15 / 3",          # 基本除法
        "sqrt(16)",        # 平方根
    ]

    for i, expression in enumerate(test_cases, 1):
        print(f"测试 {i}: {expression}")
        result = registry.execute_tool("my_calculator", expression)
        print(f"结果: {result}\n")

def test_with_simple_agent():
    """测试与SimpleAgent的集成"""
    from hello_agents import HelloAgentsLLM

    # 创建LLM客户端
    llm = HelloAgentsLLM()

    # 创建包含计算器的注册表
    registry = create_calculator_registry()

    print("🤖 与SimpleAgent集成测试:")

    # 模拟SimpleAgent使用工具的场景
    user_question = "请帮我计算 sqrt(16) + 2 * 3"

    print(f"用户问题: {user_question}")

    # 使用工具计算
    calc_result = registry.execute_tool("my_calculator", "sqrt(16) + 2 * 3")
    print(f"计算结果: {calc_result}")

    # 构建最终回答
    final_messages = [
        {"role": "user", "content": f"计算结果是 {calc_result}，请用自然语言回答用户的问题:{user_question}"}
    ]

    print("\n🔴 SimpleAgent的回答:")
    response = llm.think(final_messages)
    for chunk in response:
        print(chunk, end="", flush=True)
    print("\n")

if __name__ == "__main__":
    test_calculator_tool()
    test_with_simple_agent()
```

通过这个简化的数学计算工具案例，我们学会了如何快速开发自定义工具：编写一个简单的计算函数，通过ToolRegistry注册，然后与SimpleAgent集成使用。为了更直观的观察，这里提供了图7.1，可以清晰理解代码的运行逻辑。

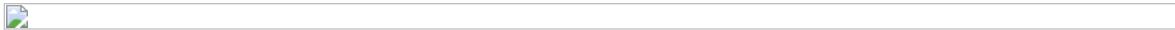


图 7.1 基于Helloagents的SimpleAgent运行工作流

7.5.3 多源搜索工具

在实际应用中，我们经常需要整合多个外部服务来提供更强大的功能。搜索工具就是一个典型的例子，它整合多个搜索引擎，能提供更加完备的真实信息。在第一章我们使用过Tavily的搜索API，在第四章我们使用过SerpApi的搜索API。因此这次我们使用这两个API来实现多源搜索功能。如果没安装对应的python依赖可以运行下面这条脚本：

```
pip install "hello-agents[search]==0.1.1"
```

(1) 搜索工具的统一接口设计

HelloAgents框架内置的SearchTool展示了如何设计一个高级的多源搜索工具：

```
class SearchTool(Tool):
    """
    智能混合搜索工具

    支持多种搜索引擎后端，智能选择最佳搜索源：
    1. 混合模式 (hybrid) - 智能选择TAVILY或SERPAPI
    2. Tavily API (tavily) - 专业AI搜索
    3. SerpApi (serpapi) - 传统Google搜索
    """

    def __init__(self, backend: str = "hybrid", tavily_key: Optional[str] = None, serpapi_key: Optional[str] = None):
        super().__init__(
            name="search",
            description="一个智能网页搜索引擎。支持混合搜索模式，自动选择最佳搜索源。"
        )
        self.backend = backend
        self.tavily_key = tavily_key or os.getenv("TAVILY_API_KEY")
        self.serpapi_key = serpapi_key or os.getenv("SERPAPI_API_KEY")
        self.available_backends = []
        self._setup_backends()
```

这个设计的核心思想是根据可用的API密钥和依赖库，自动选择最佳的搜索后端。

(2) TAVILY与SERPAPI搜索源的整合策略

框架实现了智能的后端选择逻辑：

```
def _search_hybrid(self, query: str) -> str:
    """
    混合搜索 - 智能选择最佳搜索源
    # 优先使用Tavily (AI优化的搜索)
    if "tavily" in self.available_backends:
        try:
            return self._search_tavily(query)
        except Exception as e:
            print(f"⚠️ Tavily搜索失败: {e}")
            # 如果Tavily失败，尝试SerpApi
            if "serpapi" in self.available_backends:
                print("🔄 切换到SerpApi搜索")
                return self._search_serpapi(query)

    # 如果Tavily不可用，使用SerpApi
    elif "serpapi" in self.available_backends:
        try:
            return self._search_serpapi(query)
        except Exception as e:
            print(f"⚠️ SerpApi搜索失败: {e}")

    # 如果都不可用，提示用户配置API
    return "❗️ 没有可用的搜索源，请配置TAVILY_API_KEY或SERPAPI_API_KEY环境变量"
```

这种设计体现了高可用系统的核心理念：通过降级机制，系统能够从最优的搜索源逐步降级到可用的备选方案。当所有搜索源都不可用时，明确提示用户配置正确的API密钥。

(3) 搜索结果的统一格式化

不同搜索引擎返回的结果格式不同，框架通过统一的格式化方法来处理：

```
def _search_tavily(self, query: str) -> str:
    """
    使用Tavily搜索
    response = self.tavily_client.search(
        query=query,
        search_depth="basic",
        include_answer=True,
        max_results=3
    )

    result = f"🌐 Tavily AI搜索结果:{response.get('answer', '未找到直接答案')}\n\n"
    for i, item in enumerate(response.get('results', [])[3:], 1):
        result += f"Result {i}: {item['text']}\n
```

```

result += f"[{i}] {item.get('title', '')}\n"
result += f"    {item.get('content', '')}[:200]}\n"
result += f"    来源: {item.get('url', '')}\n\n"

```

```
return result
```

基于框架的设计思想，我们可以创建自己的高级搜索工具。这次我们使用类的方式来展示不同的实现方法，创建my_advanced_search.py:

```

# my_advanced_search.py
import os
from typing import Optional, List, Dict, Any
from hello_agents import ToolRegistry

class MyAdvancedSearchTool:
    """
    自定义高级搜索工具类
    展示多源整合和智能选择的设计模式
    """

    def __init__(self):
        self.name = "my_advanced_search"
        self.description = "智能搜索工具，支持多个搜索源，自动选择最佳结果"
        self.search_sources = []
        self._setup_search_sources()

    def _setup_search_sources(self):
        """设置可用的搜索源"""
        # 检查Tavily可用性
        if os.getenv("TAVILY_API_KEY"):
            try:
                from tavily import TavilyClient
                self.tavily_client = TavilyClient(api_key=os.getenv("TAVILY_API_KEY"))
                self.search_sources.append("tavily")
                print("✅ Tavily搜索源已启用")
            except ImportError:
                print("⚠️ Tavily库未安装")

        # 检查SerpApi可用性
        if os.getenv("SERPAPI_API_KEY"):
            try:
                import serpapi
                self.search_sources.append("serpapi")
                print("✅ SerpApi搜索源已启用")
            except ImportError:
                print("⚠️ SerpApi库未安装")

        if self.search_sources:
            print(f"🔍 可用搜索源: {', '.join(self.search_sources)}")
        else:
            print("⚠️ 没有可用的搜索源，请配置API密钥")

    def search(self, query: str) -> str:
        """执行智能搜索"""
        if not query.strip():
            return "❌ 错误: 搜索查询不能为空"

        # 检查是否有可用的搜索源
        if not self.search_sources:
            return "❌ 没有可用的搜索源，请配置以下API密钥之一:
```

1. Tavily API: 设置环境变量 TAVILY_API_KEY

获取地址: <https://tavily.com/>

2. SerpAPI: 设置环境变量 SERPAPI_API_KEY

获取地址: <https://serpapi.com/>

配置后重新运行程序。"""

```

print("🔍 开始智能搜索: {query}")

# 尝试多个搜索源，返回最佳结果
for source in self.search_sources:
    try:
        if source == "tavily":
            result = self._search_with_tavily(query)
            if result and "未找到" not in result:
                return f"💡 Tavily AI搜索结果:\n\n{result}"

        elif source == "serpapi":
            result = self._search_with_serpapi(query)
            if result and "未找到" not in result:
                return f"🌐 SerpApi Google搜索结果:\n\n{result}"

    except Exception as e:
        print(f"⚠️ {source} 搜索失败: {e}")
        continue

return "❌ 所有搜索源都失败了，请检查网络连接和API密钥配置"

```

```

def _search_with_tavily(self, query: str) -> str:
    """使用Tavily搜索"""
    response = self.tavily_client.search(query=query, max_results=3)

    if response.get('answer'):
        result = f"💡 AI直接答案:{response['answer']}\\n\\n"
    else:
        result = ""

    result += "🔗 相关结果:\\n"
    for i, item in enumerate(response.get('results', [])[:3], 1):
        result += f"[{i}] {item.get('title', '')}\\n"
        result += f"    {item.get('content', '')[:150]}...\\n\\n"

    return result

def _search_with_serpapi(self, query: str) -> str:
    """使用SerpApi搜索"""
    import serpapi

    search = serpapi.GoogleSearch({
        "q": query,
        "api_key": os.getenv("SERPAPI_API_KEY"),
        "num": 3
    })

    results = search.get_dict()

    result = "🔗 Google搜索结果:\\n"
    if "organic_results" in results:
        for i, res in enumerate(results["organic_results"][:3], 1):
            result += f"[{i}] {res.get('title', '')}\\n"
            result += f"    {res.get('snippet', '')}\\n\\n"

    return result

def create_advanced_search_registry():
    """创建包含高级搜索工具的注册表"""
    registry = ToolRegistry()

    # 创建搜索工具实例
    search_tool = MyAdvancedSearchTool()

    # 注册搜索工具的方法作为函数
    registry.register_function(
        name="advanced_search",
        description="高级搜索工具，整合Tavily和SerpAPI多个搜索源，提供更全面的搜索结果",
        func=search_tool.search
    )

    return registry

```

接下来可以测试我们自己编写的工具，创建test_advanced_search.py：

```

# test_advanced_search.py
from dotenv import load_dotenv
from my_advanced_search import create_advanced_search_registry, MyAdvancedSearchTool

# 加载环境变量
load_dotenv()

def test_advanced_search():
    """测试高级搜索工具"""

    # 创建包含高级搜索工具的注册表
    registry = create_advanced_search_registry()

    print("🔍 测试高级搜索工具\\n")

    # 测试查询
    test_queries = [
        "Python编程语言的历史",
        "人工智能的最新发展",
        "2024年科技趋势"
    ]

    for i, query in enumerate(test_queries, 1):
        print(f"测试 {i}: {query}")
        result = registry.execute_tool("advanced_search", query)
        print(f"结果: {result}\\n")
        print("-" * 60 + "\\n")

def test_api_configuration():
    """测试API配置检查"""

    print("🔧 测试API配置检查:")

    # 直接创建搜索工具实例
    search_tool = MyAdvancedSearchTool()

    # 如果没有配置API，会显示配置提示

```

```

result = search_tool.search("机器学习算法")
print(f"搜索结果: {result}")

def test_with_agent():
    """测试与Agent的集成"""
    print("\n💡 与Agent集成测试:")
    print("高级搜索工具已准备就绪，可以与Agent集成使用")

# 显示工具描述
registry = create_advanced_search_registry()
tools_desc = registry.get_tools_description()
print(f"工具描述:\n{tools_desc}")

if __name__ == "__main__":
    test_advanced_search()
    test_api_configuration()
    test_with_agent()

```

通过这个高级搜索工具的设计实践，我们学会了如何使用类的方式来构建复杂的工具系统。相比函数方式，类方式更适合需要维护状态（如API客户端、配置信息）的工具。

7.5.4 工具系统的高级特性

在掌握了基础的工具开发和多源整合后，我们来探讨工具系统的高级特性。这些特性能让工具系统在复杂的生产环境中稳定运行，并为Agent提供更强大的能力。

(1) 工具链式调用机制

在实际应用中，Agent经常需要组合使用多个工具来完成复杂任务。我们可以设计一个工具链管理器来支持这种场景，这里借鉴了第六章中提到的图的概念：

```

# tool_chain_manager.py
from typing import List, Dict, Any, Optional
from hello_agents import ToolRegistry

class ToolChain:
    """工具链 - 支持多个工具的顺序执行"""

    def __init__(self, name: str, description: str):
        self.name = name
        self.description = description
        self.steps: List[Dict[str, Any]] = []

    def add_step(self, tool_name: str, input_template: str, output_key: str = None):
        """添加工具执行步骤

        Args:
            tool_name: 工具名称
            input_template: 输入模板，支持变量替换
            output_key: 输出结果的键名，用于后续步骤引用
        """
        self.steps.append({
            "tool_name": tool_name,
            "input_template": input_template,
            "output_key": output_key or f"step_{len(self.steps)}_result"
        })

    def execute(self, registry: ToolRegistry, initial_input: str, context: Dict[str, Any] = None) -> str:
        """执行工具链"""
        context = context or {}
        context["input"] = initial_input

        print(f"💡 开始执行工具链: {self.name}")

        for i, step in enumerate(self.steps, 1):
            tool_name = step["tool_name"]
            input_template = step["input_template"]
            output_key = step["output_key"]

            # 替换模板中的变量
            try:
                tool_input = input_template.format(**context)
            except KeyError as e:
                return f"🔴 工具链执行失败: 模板变量 {e} 未找到"

            print(f"  步骤 {i}: 使用 {tool_name} 处理 '{tool_input[:50]}...'")


            # 执行工具
            result = registry.execute_tool(tool_name, tool_input)
            context[output_key] = result

            print(f"  ✅ 步骤 {i} 完成，结果长度: {len(result)} 字符")

        # 返回最后一步的结果
        final_result = context[self.steps[-1]["output_key"]]
        print(f"💡 工具链 '{self.name}' 执行完成")
        return final_result

```

```

class ToolChainManager:
    """工具链管理器"""

    def __init__(self, registry: ToolRegistry):
        self.registry = registry
        self.chains: Dict[str, ToolChain] = {}

    def register_chain(self, chain: ToolChain):
        """注册工具链"""
        self.chains[chain.name] = chain
        print(f"✓ 工具链 '{chain.name}' 已注册")

    def execute_chain(self, chain_name: str, input_data: str, context: Dict[str, Any] = None) -> str:
        """执行指定的工具链"""
        if chain_name not in self.chains:
            return f"✗ 工具链 '{chain_name}' 不存在"

        chain = self.chains[chain_name]
        return chain.execute(self.registry, input_data, context)

    def list_chains(self) -> List[str]:
        """列出所有工具链"""
        return list(self.chains.keys())

# 使用示例
def create_research_chain() -> ToolChain:
    """创建一个研究工具链: 搜索 -> 计算 -> 总结"""
    chain = ToolChain(
        name="research_and_calculate",
        description="搜索信息并进行相关计算"
    )

    # 步骤1: 搜索信息
    chain.add_step(
        tool_name="search",
        input_template="{input}",
        output_key="search_result"
    )

    # 步骤2: 基于搜索结果进行计算 (如果需要)
    chain.add_step(
        tool_name="my_calculator",
        input_template="根据以下信息计算相关数值:{search_result}",
        output_key="calculation_result"
    )

    return chain

```

(2) 异步工具执行支持

对于耗时的工具操作，我们可以提供异步执行支持：

```

# async_tool_executor.py
import asyncio
import concurrent.futures
from typing import Dict, Any, List, Callable
from hello_agents import ToolRegistry

class AsyncToolExecutor:
    """异步工具执行器"""

    def __init__(self, registry: ToolRegistry, max_workers: int = 4):
        self.registry = registry
        self.executor = concurrent.futures.ThreadPoolExecutor(max_workers=max_workers)

    @asyncio.coroutine
    def execute_tool_async(self, tool_name: str, input_data: str) -> str:
        """异步执行单个工具"""
        loop = asyncio.get_event_loop()

        def _execute():
            return self.registry.execute_tool(tool_name, input_data)

        result = await loop.run_in_executor(self.executor, _execute)
        return result

    @asyncio.coroutine
    def execute_tools_parallel(self, tasks: List[Dict[str, str]]) -> List[str]:
        """并行执行多个工具"""
        print(f"🚀 开始并行执行 {len(tasks)} 个工具任务")

        # 创建异步任务
        async_tasks = []
        for task in tasks:
            tool_name = task["tool_name"]
            input_data = task["input_data"]
            async_task = self.execute_tool_async(tool_name, input_data)
            async_tasks.append(async_task)

        # 等待所有任务完成
        results = await asyncio.gather(*async_tasks)

        print(f"✓ 所有工具任务执行完成")
        return results

```

```

def __del__(self):
    """清理资源"""
    if hasattr(self, 'executor'):
        self.executor.shutdown(wait=True)

# 使用示例
async def test_parallel_execution():
    """测试并行工具执行"""
    from hello_agents import ToolRegistry

    registry = ToolRegistry()
    # 假设已经注册了搜索和计算工具

    executor = AsyncToolExecutor(registry)

    # 定义并行任务
    tasks = [
        {"tool_name": "search", "input_data": "Python编程"},
        {"tool_name": "search", "input_data": "机器学习"},
        {"tool_name": "my_calculator", "input_data": "2 + 2"},
        {"tool_name": "my_calculator", "input_data": "sqrt(16)"},

    ]
    # 并行执行
    results = await executor.execute_tools_parallel(tasks)

    for i, result in enumerate(results):
        print(f"任务 {i+1} 结果: {result[:100]}...")

```

基于以上的设计和实现经验，我们可以总结出工具系统开发的核心理念：在设计层面，每个工具都应该遵循单一职责原则，专注于特定功能的同时保持接口的统一性，并将完善的异常处理和安全优先的输入验证作为基本要求。在性能优化方面，利用异步执行提高并发处理能力，同时合理管理外部连接和系统资源。

7.6 本章小结

在正式总结之前，我们想告诉大家一个好消息：对于本章实现的所有方法和功能，都在GitHub仓库中提供了完整的测试案例。你可以访问[这个链接](#)查看和运行这些测试代码。这个文件包含了四种Agent范式的演示、工具系统的集成测试、高级功能的使用示例，以及交互式的Agent体验。如果你想验证自己的实现是否正确，或者想深入了解框架的实际使用方式，这些测试案例将是有价值的参考。

回顾本章，我们完成了一项富有挑战的任务：一步步构建了一个基础的智能体框架——HelloAgents。这个过程始终遵循着“分层解耦、职责单一、接口统一”的核心原则。

在框架的具体实现中，我们再次实现了四种经典的Agent范式。从SimpleAgent的基础对话模式，到ReActAgent的推理与行动结合；从ReflectionAgent的自我反思与迭代优化，到PlanAndSolveAgent的分解规划与逐步执行。而工具系统作为Agent能力延伸的核心，其构建过程则是一次完整的工程实践。

更重要的是，第七章的构建并非终点，而是为后续更深入学习提供了必要的技术基础。我们在设计之初便充分考虑了后续内容的延展性，为高级功能的实现预留了必要的接口和扩展点。我们所建立的统一LLM接口、标准化消息系统、工具注册机制，共同构成了一个完备的技术底座。这使得我们在后续章节中，可以更加从容地去学习更高级的主题：第八章的记忆与RAG系统将基于此扩展Agent的能力边界；第九章的上下文工程将深入我们已经建立的消息处理机制；第十章的智能体协议则需要扩展新的工具。

接下来，我们将一起探索如何往框架中加入RAG系统与Memory机制，敬请期待第八章！

习题

1. 本章构建了 HelloAgents 框架，并阐述了“为何需要自建Agent框架”。请分析：

- 在7.1.1节中提到了当前主流框架的四个主要局限性。结合你在[第六章习题](#)或实际项目中使用过的某个框架的实际经验，说明这些问题是如何影响开发效率的。
- HelloAgents 提出了“万物皆为工具”的设计理念，将 Memory、RAG、MCP 等模块都抽象为工具。这种设计有什么优势？是否存在局限性？请举例说明。
- 对比第四章从零实现的智能体代码和本章的框架化实现，框架化带来了哪些具体的改进？如果让你设计一个框架，你会优先考虑哪些设计原则？

2. 在7.2节中，我们扩展了 HelloAgentsLLM 以支持多模型供应商和本地模型调用。

提示：这是一道实践题，建议实际操作

- 参考7.2.1节的示例，尝试为 HelloAgentsLLM 添加一个新模型供应商的支持（如Gemini、Anthropic、Kim）。要求通过继承方式实现，并能够自动检测该提供商的环境变量。
- 在7.2.3节中介绍了自动检测机制的三个优先级。请分析：如果同时设置了 OPENAI_API_KEY 和 LLM_BASE_URL="http://localhost:11434/v1"，框架最后会选择哪个提供商？这种优先级设计是否合理？
- 除了本章介绍的 VLLM 和 Ollama，还有 SGLang 等其他本地模型部署方案。请先搜索并了解 SGLang 的基本信息和特点，然后对比 VLLM、SGLang 和 Ollama 这三者在易用性、资源占用、推理速度、推理精度等方面的优势。

3. 在7.3节中，我们实现了 Message 类、Config 类和 Agent 基类。请分析：

- Message 类使用了 Pydantic 的 BaseModel 进行数据验证。这种设计在实际应用中有哪些优势？

- Agent 基类定义了 run 和 _execute 两个方法，其中 run 是公开接口，_execute 是抽象方法。这种设计模式叫什么？有什么好处？
- 在 Config 类中，我们使用了单例模式。请解释什么是单例模式，为什么配置管理需要使用单例模式？如果不使用单例会导致什么问题？

4. 在7.4节中，我们动手进行了四种 Agent 范式的框架化实现。

提示：这是一道实践题，建议实际操作

- 对比第四章从零实现的 ReActAgent 和本章框架化的 ReActAgent，列举3个具体的改进点，并说明这些改进如何提升了代码的可维护性和可扩展性。
- ReflectionAgent 实现了“执行-反思-优化”循环。请扩展这个实现，添加一个“质量评分”机制：在每次反思后，让 LLM 对当前版本的输出打分，只有分数低于阈值时才继续优化，否则提前终止。
- 请设计并实现一个新的 Agent 范式 Tree-of-Thought Agent，要求继承 Agent 基类，它能够在每一步生成多个可能的思考路径，然后选择最优路径继续。

5. 在7.5节中，我们构建了工具系统。请思考以下问题：

- BaseTool 类定义了 execute 抽象方法，所有工具都必须实现这个方法。请解释为什么要强制所有工具实现统一的接口？如果某个工具需要返回多个值（如搜索工具返回标题、摘要、链接），应该如何设计？
- 在7.5.3节中实现了工具链（ToolChain）。请设计一个实际的应用场景，需要串联至少3个工具，并画出工具链的执行流程图。
- 异步工具执行器（AsyncToolExecutor）使用了线程池来并行执行工具。请分析：在什么情况下并行执行工具能带来性能提升？

6. 框架的可扩展性是设计的重要考量因素之一。你现在要扩展 HelloAgents 框架，为其实现一些有趣的新功能和特性。

- 首先为 HelloAgents 添加一个“流式输出”功能，使得 Agent 在生成响应时能够实时返回中间结果（类似 ChatGPT 用户界面的打字效果）。请设计这个功能的实现方案，说明需要修改哪些类和方法。
- 然后为框架添加“多轮对话管理”功能，能够自动管理对话历史、支持对话分支和回溯，你会如何设计？需要新增哪些类？如何与现有的 Message 系统集成？
- 最后请为 HelloAgents 设计一个“插件系统”，允许第三方开发者通过插件的方式扩展框架功能（如添加新的 Agent 类型、新的工具类型等），而无需修改框架核心代码。要求画出插件系统的架构图并说明关键接口。

第八章 记忆与检索

在前面的章节中，我们构建了HelloAgents框架的基础架构，实现了多种智能体范式和工具系统。不过，我们的框架还缺少一个关键能力：记忆。如果智能体无法记住之前的交互内容，也无法从历史经验中学习，那么在连续对话或复杂任务中，其表现将受到极大限制。

本章将在第七章构建的框架基础上，为HelloAgents增加两个核心能力：记忆系统（Memory System）和检索增强生成（Retrieval-Augmented Generation, RAG）。我们将采用“框架扩展 + 知识科普”的方式，在构建过程中深入理解Memory和RAG的理论基础，最终实现一个具有完整记忆和知识检索能力的智能体系统。

8.1 从认知科学到智能体记忆

8.1.1 人类记忆系统的启发

在构建智能体的记忆系统之前，让我们先从认知科学的角度理解人类是如何处理和存储信息的。人类记忆是一个多层级的认知系统，它不仅能存储信息，还能根据重要性、时间和上下文对信息进行分类和整理。认知心理学为理解记忆的结构和过程提供了经典的理论框架^[1]，如图8.1所示。



图 8.1 人类记忆系统的层次结构

根据认知心理学的研究，人类记忆可以分为以下几个层次：

1. 感觉记忆（Sensory Memory）：持续时间极短（0.5-3秒），容量巨大，负责暂时保存感官接收到的所有信息
2. 工作记忆（Working Memory）：持续时间短（15-30秒），容量有限（7±2个项目），负责当前任务的信息处理
3. 长期记忆（Long-term Memory）：持续时间长（可达终生），容量几乎无限，进一步分为：
 - 程序性记忆：技能和习惯（如骑自行车）
 - 陈述性记忆：可以用语言表达的知识，又分为：
 - 语义记忆：一般知识和概念（如“巴黎是法国首都”）
 - 情景记忆：个人经历和事件（如“昨天的会议内容”）

8.1.2 为何智能体需要记忆与RAG

借鉴人类记忆系统的设计，我们可以理解为什么智能体也需要类似的记忆能力。人类智能的一个重要特征就是能够记住过去的经历，从中学习，并将这些经验应用到新的情况中。同样，一个真正智能的智能体也需要具备记忆能力。对于基于LLM的智能体而言，通常面临两个根本性局限：对话状态的遗忘和内置知识的局限。

(1) 局限一：无状态导致的对话遗忘

当前的大语言模型虽然强大，但设计上是无状态的。这意味着，每一次用户请求（或API调用）都是一次独立的、无关联的计算。模型本身不会自动“记住”上一次对话的内容。这带来了几个问题：

1. 上下文丢失：在长对话中，早期的重要信息可能会因为上下文窗口限制而丢失
2. 个性化缺失：Agent无法记住用户的偏好、习惯或特定需求
3. 学习能力受限：无法从过往的成功或失败经验中学习改进
4. 一致性问题：在多轮对话中可能出现前后矛盾的回答

让我们通过一个具体例子来理解这个问题：

```
# 第七章的Agent使用方式
from hello_agents import SimpleAgent, HelloAgentsLLM

agent = SimpleAgent(name="学习助手", llm=HelloAgentsLLM())

# 第一次对话
response1 = agent.run("我叫张三，正在学习Python，目前掌握了基础语法")
print(response1) # "很好！Python基础语法是编程的重要基础..."

# 第二次对话（新的会话）
response2 = agent.run("你还记得我的学习进度吗？")
print(response2) # "抱歉，我不知道您的学习进度..."
```

要解决这个问题，我们的框架需要引入记忆系统。

(2) 局限二：模型内置知识的局限性

除了遗忘对话历史，LLM的另一个核心局限在于其知识是静态的、有限的。这些知识完全来自于它的训练数据，并因此带来一系列问题：

1. 知识时效性：大模型的训练数据有时间截止点，无法获取最新信息
2. 专业领域知识：通用模型在特定领域的深度知识可能不足
3. 事实准确性：通过检索验证，减少模型的幻觉问题

4. 可解释性：提供信息来源，增强回答的可信度

为了克服这一局限，RAG技术应运而生。它的核心思想是在模型生成回答之前，先从一个外部知识库（如文档、数据库、API）中检索出最相关的信息，并将这些信息作为上下文一同提供给模型。

8.1.3 记忆与RAG系统架构设计

基于第七章建立的框架基础和认知科学的启发，我们设计了一个分层的记忆与RAG系统架构，如图8.2所示。这个架构不仅借鉴了人类记忆系统的层次结构，还充分考虑了工程实现的可扩展性。在实现上，我们将记忆和RAG设计为两个独立的工具：`memory_tool`负责存储和维护对话过程中的交互信息，`rag_tool`则负责从用户提供的知识库中检索相关信息作为上下文，并可将重要的检索结果自动存储到记忆系统中。



图 8.2 HelloAgents记忆与RAG系统整体架构

记忆系统采用了四层架构设计：



RAG系统专注于外部知识的获取和利用：



8.1.4 本章学习目标与快速体验

让我们先看看第八章的核心学习内容：



快速开始：安装HelloAgents框架

为了让读者能够快速体验本章的完整功能，我们提供了可直接安装的Python包。你可以通过以下命令安装本章对应的版本：

```
pip install "hello-agents[all]==0.2.0"
python -m spacy download zh_core_web_sm
python -m spacy download en_core_web_sm
```

除此之外，还需要在.env配置图数据库、向量数据库、LLM以及Embedding方案的API。在教程中向量数据库采用Qdrant，图数据库采用Neo4J，Embedding首选百炼平台，若没有API可切换为本地部署模型方案。

```
# =====
# Qdrant 向量数据库配置 - 获取API密钥: https://cloud.qdrant.io/
# =====
# 使用Qdrant云服务（推荐）
QDRANT_URL=https://your-cluster.qdrant.tech:6333
QDRANT_API_KEY=your_qdrant_api_key_here

# 或使用本地Qdrant（需要Docker）
# QDRANT_URL=http://localhost:6333
# QDRANT_API_KEY=

# Qdrant集合配置
QDRANT_COLLECTION=hello_agents_vectors
QDRANT_VECTOR_SIZE=384
QDRANT_DISTANCE=cosine
QDRANT_TIMEOUT=30

# =====
# Neo4j 图数据库配置 - 获取API密钥: https://neo4j.com/cloud/aura/
# =====
# 使用Neo4j Aura云服务（推荐）
NEO4J_URI=neo4j+s://your-instance.databases.neo4j.io
NEO4J_USERNAME=neo4j
NEO4J_PASSWORD=your_neo4j_password_here

# 或使用本地Neo4j（需要Docker）
# NEO4J_URI=bolt://localhost:7687
# NEO4J_USERNAME=neo4j
# NEO4J_PASSWORD=hello-agents-password

# Neo4j连接配置
NEO4J_DATABASE=neo4j
NEO4J_MAX_CONNECTION_LIFETIME=3600
NEO4J_MAX_CONNECTION_POOL_SIZE=50
NEO4J_CONNECTION_TIMEOUT=60

# =====
# 嵌入（Embedding）配置示例 - 可从阿里云控制台获取: https://dashscope.aliyun.com/
# =====
# - 若为空, dashscope 默认 text-embedding-v3; local 默认 sentence-transformers/all-MiniLM-L6-v2
EMBED_MODEL_TYPE=dashscope
EMBED_MODEL_NAME=
EMBED_API_KEY=
EMBED_BASE_URL=
```

本章的学习可以采用两种方式：

1. 体验式学习：直接使用pip安装框架，运行示例代码，快速体验各种功能
2. 深度学习：跟随本章内容，从零开始实现每个组件，深入理解框架的设计思想和实现细节

我们建议采用“先体验，后实现”的学习路径。在本章中，我们提供了完整的测试文件，你可以重写核心函数并运行测试，以检验你的实现是否正确。

遵循第七章确立的设计原则，我们将记忆和RAG能力封装为标准工具，而不是创建新的Agent类。在开始之前，让我们用30秒体验使用Hello-agents构建具有记忆和RAG能力的智能体！

```
# 配置好同级文件夹下.env中的大模型API
from hello_agents import SimpleAgent, HelloAgentsLLM, ToolRegistry
from hello_agents.tools import MemoryTool, RAGTool

# 创建LLM实例
llm = HelloAgentsLLM()

# 创建Agent
agent = SimpleAgent(
    name="智能助手",
    llm=llm,
    system_prompt="你是一个有记忆和知识检索能力的AI助手"
)

# 创建工具注册表
tool_registry = ToolRegistry()

# 添加记忆工具
memory_tool = MemoryTool(user_id="user123")
tool_registry.register_tool(memory_tool)

# 添加RAG工具
rag_tool = RAGTool(knowledge_base_path="./knowledge_base")
```

2/9/26, 1:56 PM

```
tool_registry.register_tool(rag_tool)

# 为Agent配置工具
agent.tool_registry = tool_registry

# 开始对话
response = agent.run("你好！请记住我叫张三，我是一名Python开发者")
print(response)
```

如果一切配置完毕，可以看到以下内容。

```
[OK] SQLite 数据库表和索引创建完成
[OK] SQLite 文档存储初始化完成: ./memory_data/memory.db
INFO:hello_agents.memory.storage.qdrant_store:✓ 成功连接到Qdrant云服务: https://0c517275-2ad0-4442-8309-11c36dc7e811.us-east-1-1.aws.cloud.qdra
INFO:hello_agents.memory.storage.qdrant_store:✓ 使用现有Qdrant集合: hello_agents_vectors
INFO:hello_agents.memory.types.semantic:✓ 嵌入模型就绪，维度: 1024
INFO:hello_agents.memory.types.semantic:✓ Qdrant向量数据库初始化完成
INFO:hello_agents.memory.storage.neo4j_store:✓ 成功连接到Neo4j云服务: neo4j+s://851b3a28.databases.neo4j.io      INFO:hello_agents.memory.types.
INFO:hello_agents.memory.storage.neo4j_store:✓ Neo4j索引创建完成
INFO:hello_agents.memory.types.semantic:✓ Neo4j图数据库初始化完成
INFO:hello_agents.memory.types.semantic:✓ 数据库健康状态: Qdrant=✓, Neo4j=✓
INFO:hello_agents.memory.types.semantic:✓ 加载中文spaCy模型: zh_core_web_sm
INFO:hello_agents.memory.types.semantic:✓ 加载英文spaCy模型: en_core_web_sm
INFO:hello_agents.memory.types.semantic:✓ 可用语言模型: 中文, 英文
INFO:hello_agents.memory.types.semantic:增强语义记忆初始化完成 (使用Qdrant+Neo4j专业数据库)
INFO:hello_agents.memory.manager:MemoryManager初始化完成, 启用记忆类型: ['working', 'episodic', 'semantic']
✓ 工具 'memory' 已注册。
INFO:hello_agents.memory.storage.qdrant_store:✓ 成功连接到Qdrant云服务: https://0c517275-2ad0-4442-8309-11c36dc7eNF0:hello_agents.memory.storage
811.us-east-1-1.aws.cloud.qdrant.io:6333
INFO:hello_agents.memory.storage.qdrant_store:✓ 使用现有Qdrant集合: rag_knowledge_base
✓ RAG工具初始化成功: namespace=default, collection=rag_knowledge_base
✓ 工具 'rag' 已注册。
你好, 张三! 很高兴认识你。作为一名Python开发者, 你一定对编程很有热情。如果你有任何技术问题或者需要讨论Python相关的话题, 随时可以找我。我会尽力帮助你。有什么我现在就能帮到你的吗?
```

8.2 记忆系统：让智能体拥有记忆

8.2.1 记忆系统的工作流程

在进入代码实现阶段前，我们需要先定义记忆系统的工作流程。该流程参考了认知科学中的记忆模型，并将每个认知阶段映射为具体的技术组件和操作。理解这一映射关系，有助于我们后续的代码实现。



图 8.3 记忆形成的认识过程

如图8.3所示，根据认知科学的研究，人类记忆的形成经历以下几个阶段：

1. 编码（Encoding）：将感知到的信息转换为可存储的形式
2. 存储（Storage）：将编码后的信息保存在记忆系统中
3. 检索（Retrieval）：根据需要从记忆中提取相关信息
4. 整合（Consolidation）：将短期记忆转化为长期记忆
5. 遗忘（Forgetting）：删除不重要或过时的信息

基于该启发，我们为 HelloAgents 设计了一套完整的记忆系统。其核心思想是模仿人类大脑处理不同类型信息的方式，将记忆划分为多个专门的模块，并建立一套智能化的管理机制。图8.4详细展示了这套系统的工作流程，包括记忆的添加、检索、整合和遗忘等关键环节。

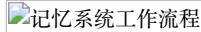


图 8.4 HelloAgents记忆系统的完整工作流程

我们的记忆系统由四种不同类型的记忆模块构成，每种模块都针对特定的应用场景和生命周期进行了优化：

首先是工作记忆（Working Memory），它扮演着智能体“短期记忆”的角色，主要用于存储当前对话的上下文信息。为确保高速访问和响应，其容量被有意限制（例如，默认50条），并且生命周期与单个会话绑定，会话结束后便会自动清理。

其次是情景记忆（Episodic Memory），它负责长期存储具体的交互事件和智能体的学习经历。与工作记忆不同，情景记忆包含了丰富的上下文信息，并支持按时间序列或主题进行回顾式检索，是智能体“复盘”和学习过往经验的基础。

与具体事件相对应的是语义记忆（Semantic Memory），它存储的是更为抽象的知识、概念和规则。例如，通过对话了解到的用户偏好、需要长期遵守的指令或领域知识点，都适合存放在这里。这部分记忆具有高度的持久性和重要性，是智能体形成“知识体系”和进行关联推理的核心。

最后，为了与日益丰富的多媒体交互，我们引入了感知记忆（Perceptual Memory）。该模块专门处理图像、音频等多模态信息，并支持跨模态检索。其生命周期会根据信息的重要性和可用存储空间进行动态管理。

8.2.2 快速体验：30秒上手记忆功能

在深入实现细节之前，让我们先快速体验一下记忆系统的基本功能：

```
from hello_agents import SimpleAgent, HelloAgentsLLM, ToolRegistry
from hello_agents.tools import MemoryTool

# 创建具有记忆能力的Agent
llm = HelloAgentsLLM()
agent = SimpleAgent(name="记忆助手", llm=llm)

# 创建记忆工具
memory_tool = MemoryTool(user_id="user123")
tool_registry = ToolRegistry()
tool_registry.register_tool(memory_tool)
agent.tool_registry = tool_registry

# 体验记忆功能
print("==> 添加多个记忆 ==>")

# 添加第一个记忆
result1 = memory_tool.execute("add", content="用户张三是一名Python开发者，专注于机器学习和数据分析", memory_type="semantic", importance=0.8)
print(f"记忆1: {result1}")

# 添加第二个记忆
result2 = memory_tool.execute("add", content="李四是前端工程师，擅长React和Vue.js开发", memory_type="semantic", importance=0.7)
print(f"记忆2: {result2}")

# 添加第三个记忆
result3 = memory_tool.execute("add", content="王五是产品经理，负责用户体验设计和需求分析", memory_type="semantic", importance=0.6)
print(f"记忆3: {result3}")

print("\n==> 搜索特定记忆 ==>")
# 搜索前端相关的记忆
print("🔍 搜索 '前端工程师':")
result = memory_tool.execute("search", query="前端工程师", limit=3)
print(result)

print("\n==> 记忆摘要 ==>")
result = memory_tool.execute("summary")
print(result)
```

8.2.3 MemoryTool详解

现在让我们采用自顶向下的方式，从MemoryTool支持的具体操作开始，逐步深入到底层实现。MemoryTool作为记忆系统的统一接口，其设计遵循了“统一入口，分发处理”的架构模式：

```
def execute(self, action: str, **kwargs) -> str:
    """执行记忆操作

    支持的操作：
    - add: 添加记忆（支持4种类型：working/episodic/semantic/perceptual）
    - search: 搜索记忆
    - summary: 获取记忆摘要
    - stats: 获取统计信息
    - update: 更新记忆
    - remove: 删除记忆
    - forget: 遗忘记忆（多种策略）
    - consolidate: 整合记忆（短期→长期）
    - clear_all: 清空所有记忆
    """

    if action == "add":
        return self._add_memory(**kwargs)
    elif action == "search":
        return self._search_memory(**kwargs)
    elif action == "summary":
        return self._get_summary(**kwargs)
    # ... 其他操作
```

这种统一的execute接口设计简化了Agent的调用方式，通过action参数指定具体操作，使用**kwargs允许每个操作有不同的参数需求。在这里我们会将比较重要的几个操作罗列出来：

(1) 操作1: add

add操作是记忆系统的基础，它模拟了人类大脑将感知信息编码为记忆的过程。在实现中，我们不仅要存储记忆内容，还要为每个记忆添加丰富的上下文信息，这些信息将在后续的检索和管理中发挥重要作用。

```
def _add_memory(
    self,
    content: str = "",
    memory_type: str = "working",
    importance: float = 0.5,
    file_path: str = None,
    modality: str = None,
```

```

    **metadata
) -> str:
    """添加记忆"""
    try:
        # 确保会话ID存在
        if self.current_session_id is None:
            self.current_session_id = f"session_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
    # 感知记忆文件支持
    if memory_type == "perceptual" and file_path:
        inferred = modality or self._infer_modality(file_path)
        metadata.setdefault("modality", inferred)
        metadata.setdefault("raw_data", file_path)

    # 添加会话信息到元数据
    metadata.update({
        "session_id": self.current_session_id,
        "timestamp": datetime.now().isoformat()
    })

    memory_id = self.memory_manager.add_memory(
        content=content,
        memory_type=memory_type,
        importance=importance,
        metadata=metadata,
        auto_classify=False
    )

    return f"✅ 记忆已添加 (ID: {memory_id[:8]}...)"
except Exception as e:
    return f"❌ 添加记忆失败: {str(e)}"

```

这里主要实现了三个关键任务：会话ID的自动管理（确保每个记忆都有明确的会话归属）、多模态数据的智能处理（自动推断文件类型并保存相关元数据）、以及上下文信息的自动补充（为每个记忆添加时间戳和会话信息）。其中，`importance`参数（默认0.5）用于标记记忆的重要程度，取值范围0.0-1.0，这个机制模拟了人类大脑对不同信息重要性的评估。这种设计让Agent能够自动区分不同时间段的对话，并为后续的检索和管理提供丰富的上下文信息。

其中，对每个记忆类型，我们提供了不同的使用示例：

```

# 1. 工作记忆 - 临时信息，容量有限
memory_tool.execute("add",
    content="用户刚才问了关于Python函数的问题",
    memory_type="working",
    importance=0.6
)

# 2. 情景记忆 - 具体事件和经历
memory_tool.execute("add",
    content="2024年3月15日，用户张三完成了第一个Python项目",
    memory_type="episodic",
    importance=0.8,
    event_type="milestone",
    location="在线学习平台"
)

# 3. 语义记忆 - 抽象知识和概念
memory_tool.execute("add",
    content="Python是一种解释型、面向对象的编程语言",
    memory_type="semantic",
    importance=0.9,
    knowledge_type="factual"
)

# 4. 感知记忆 - 多模态信息
memory_tool.execute("add",
    content="用户上传了一张Python代码截图，包含函数定义",
    memory_type="perceptual",
    importance=0.7,
    modality="image",
    file_path="./uploads/code_screenshot.png"
)

```

(2) 操作2: search

`search`操作是记忆系统的核心功能，它需要在大量记忆中快速找到与查询最相关的内容。它涉及语义理解、相关性计算和结果排序等多个环节。

```

def _search_memory(
    self,
    query: str,
    limit: int = 5,
    memory_types: List[str] = None,
    memory_type: str = None,
    min_importance: float = 0.1
) -> str:
    """搜索记忆"""
    try:
        # 参数标准化处理
        if memory_type and not memory_types:
            memory_types = [memory_type]

        results = self.memory_manager.retrieve_memories(

```

```

        query=query,
        limit=limit,
        memory_types=memory_types,
        min_importance=min_importance
    )

    if not results:
        return f"🔍 未找到与 '{query}' 相关的记忆"

    # 格式化结果
    formatted_results = []
    formatted_results.append(f"🔍 找到 {len(results)} 条相关记忆:")

    for i, memory in enumerate(results, 1):
        memory_type_label = {
            "working": "工作记忆",
            "episodic": "情景记忆",
            "semantic": "语义记忆",
            "perceptual": "感知记忆"
        }.get(memory.memory_type, memory.memory_type)

        content_preview = memory.content[:80] + "..." if len(memory.content) > 80 else memory.content
        formatted_results.append(
            f"{i}. [{memory_type_label}] {content_preview} (重要性: {memory.importance:.2f})"
        )

    return "\n".join(formatted_results)

except Exception as e:
    return f"❌ 搜索记忆失败: {str(e)}"

```

搜索操作在设计上支持单数和复数两种参数形式（`memory_type`和`memory_types`），让用户以最自然的方式表达需求。其中，`min_importance`参数（默认0.1）用于过滤低质量记忆。对于搜索功能的使用，可以参考这个示例。

```

# 基础搜索
result = memory_tool.execute("search", query="Python编程", limit=5)

# 指定记忆类型搜索
result = memory_tool.execute("search",
    query="学习进度",
    memory_type="episodic",
    limit=3
)

# 多类型搜索
result = memory_tool.execute("search",
    query="函数定义",
    memory_types=["semantic", "episodic"],
    min_importance=0.5
)

```

(3) 操作3: forget

遗忘机制是最具认知科学色彩的功能，它模拟人类大脑的选择性遗忘过程，支持三种策略：基于重要性（删除不重要的记忆）、基于时间（删除过时的记忆）和基于容量（当存储接近上限时删除最不重要的记忆）。

```

def _forget(self, strategy: str = "importance_based", threshold: float = 0.1, max_age_days: int = 30) -> str:
    """遗忘记忆（支持多种策略）"""
    try:
        count = self.memory_manager.forget_memories(
            strategy=strategy,
            threshold=threshold,
            max_age_days=max_age_days
        )
        return f"✓ 已遗忘 {count} 条记忆 (策略: {strategy}) "
    except Exception as e:
        return f"❌ 遗忘记忆失败: {str(e)}"

```

三种遗忘策略的使用：

```

# 1. 基于重要性的遗忘 - 删除重要性低于阈值的记忆
memory_tool.execute("forget",
    strategy="importance_based",
    threshold=0.2
)

# 2. 基于时间的遗忘 - 删除超过指定天数的记忆
memory_tool.execute("forget",
    strategy="time_based",
    max_age_days=30
)

# 3. 基于容量的遗忘 - 当记忆数量超限时删除最不重要的
memory_tool.execute("forget",
    strategy="capacity_based",
    threshold=0.3
)

```

(4) 操作4: consolidate

```

def _consolidate(self, from_type: str = "working", to_type: str = "episodic", importance_threshold: float = 0.7) -> str:
    """整合记忆 (将重要的短期记忆提升为长期记忆) """
    try:
        count = self.memory_manager.consolidate_memories(
            from_type=from_type,
            to_type=to_type,
            importance_threshold=importance_threshold,
        )
        return f"🕒 已整合 {count} 条记忆为长期记忆 {{from_type}} → {{to_type}}, 阈值={importance_threshold}) "
    except Exception as e:
        return f"🔴 整合记忆失败: {str(e)}"

```

consolidate操作借鉴了神经科学中的记忆固化概念，模拟人类大脑将短期记忆转化为长期记忆的过程。默认设置是将重要性超过0.7的工作记忆转换为情景记忆，这个阈值确保只有真正重要的信息才会被长期保存。整个过程是自动化的，用户无需手动选择具体的记忆，系统会智能地识别符合条件的记忆并执行类型转换。

记忆整合的使用示例：

```

# 将重要的工作记忆转为情景记忆
memory_tool.execute("consolidate",
    from_type="working",
    to_type="episodic",
    importance_threshold=0.7
)

# 将重要的情景记忆转为语义记忆
memory_tool.execute("consolidate",
    from_type="episodic",
    to_type="semantic",
    importance_threshold=0.8
)

```

通过以上几个核心操作协作，MemoryTool构建了一个完整的记忆生命周期管理体系。从记忆的创建、检索、摘要到遗忘、整合和管理，形成了一个闭环的智能记忆管理系统，让Agent真正具备了类人的记忆能力。

8.2.4 MemoryManager详解

理解了MemoryTool的接口设计后，让我们深入到底层实现，看看MemoryTool是如何与MemoryManager协作的。这种分层设计体现了软件工程中的关注点分离原则，MemoryTool专注于用户接口和参数处理，而MemoryManager则负责核心的记忆管理逻辑。

MemoryTool在初始化时会创建一个MemoryManager实例，并根据配置启用不同类型的记忆模块。这种设计让用户可以根据具体需求选择启用哪些记忆类型，既保证了功能的完整性，又避免了不必要的资源消耗。

```

class MemoryTool(Tool):
    """记忆工具 - 为Agent提供记忆功能"""

    def __init__(
        self,
        user_id: str = "default_user",
        memory_config: MemoryConfig = None,
        memory_types: List[str] = None
    ):
        super().__init__(
            name="memory",
            description="记忆工具 - 可以存储和检索对话历史、知识和经验"
        )

        # 初始化记忆管理器
        self.memory_config = memory_config or MemoryConfig()
        self.memory_types = memory_types or ["working", "episodic", "semantic"]

        self.memory_manager = MemoryManager(
            config=self.memory_config,
            user_id=user_id,
            enable_working="working" in self.memory_types,
            enable_episodic="episodic" in self.memory_types,
            enable_semantic="semantic" in self.memory_types,
            enable_perceptual="perceptual" in self.memory_types
        )

```

MemoryManager作为记忆系统的核心协调者，负责管理不同类型的记忆模块，并提供统一的操作接口。

```

class MemoryManager:
    """记忆管理器 - 统一的记忆操作接口"""

    def __init__(
        self,
        config: Optional[MemoryConfig] = None,
        user_id: str = "default_user",
        enable_working: bool = True,
        enable_episodic: bool = True,
        enable_semantic: bool = True,
        enable_perceptual: bool = False
    ):
        self.config = config or MemoryConfig()
        self.user_id = user_id

        # 初始化存储和检索组件

```

```

self.store = MemoryStore(self.config)
self.retriever = MemoryRetriever(self.store, self.config)

# 初始化各类型记忆
self.memory_types = {}

if enable_working:
    self.memory_types['working'] = WorkingMemory(self.config, self.store)

if enable_episodic:
    self.memory_types['episodic'] = EpisodicMemory(self.config, self.store)

if enable_semantic:
    self.memory_types['semantic'] = SemanticMemory(self.config, self.store)

if enable_perceptual:
    self.memory_types['perceptual'] = PerceptualMemory(self.config, self.store)

```

8.2.5 四种记忆类型

现在让我们深入了解四种记忆类型的具体实现，每种记忆类型都有其独特的特点和应用场景：

(1) 工作记忆 (WorkingMemory)

工作记忆是记忆系统中最活跃的部分，它负责存储当前对话会话中的临时信息。工作记忆的设计重点在于快速访问和自动清理，这种设计确保了系统的响应速度和资源效率。

工作记忆采用了纯内存存储方案，配合TTL (Time To Live) 机制进行自动清理。这种设计的优势在于访问速度极快，但也意味着工作记忆的内容在系统重启后会丢失。这种特性正好符合工作记忆的定位，存储临时的、易变的信息。

```

class WorkingMemory:
    """工作记忆实现

    特点：
    - 容量有限（默认50条）+ TTL自动清理
    - 纯内存存储，访问速度极快
    - 混合检索：TF-IDF向量化 + 关键词匹配
    """

    def __init__(self, config: MemoryConfig):
        self.max_capacity = config.working_memory_capacity or 50
        self.max_age_minutes = config.working_memory_ttl or 60
        self.memories = []

    def add(self, memory_item: MemoryItem) -> str:
        """添加工作记忆"""
        self._expire_old_memories() # 过期清理

        if len(self.memories) >= self.max_capacity:
            self._remove_lowest_priority_memory() # 容量管理

        self.memories.append(memory_item)
        return memory_item.id

    def retrieve(self, query: str, limit: int = 5, **kwargs) -> List[MemoryItem]:
        """混合检索：TF-IDF向量化 + 关键词匹配"""
        self._expire_old_memories()

        # 尝试TF-IDF向量检索
        vector_scores = self._try_tfidf_search(query)

        # 计算综合分数
        scored_memories = []
        for memory in self.memories:
            vector_score = vector_scores.get(memory.id, 0.0)
            keyword_score = self._calculate_keyword_score(query, memory.content)

            # 混合评分
            base_relevance = vector_score * 0.7 + keyword_score * 0.3 if vector_score > 0 else keyword_score
            time_decay = self._calculate_time_decay(memory.timestamp)
            importance_weight = 0.8 + (memory.importance * 0.4)

            final_score = base_relevance * time_decay * importance_weight
            if final_score > 0:
                scored_memories.append((final_score, memory))

        scored_memories.sort(key=lambda x: x[0], reverse=True)
        return [memory for _, memory in scored_memories[:limit]]

```

工作记忆的检索采用了混合检索策略，首先尝试使用TF-IDF向量化进行语义检索，如果失败则回退到关键词匹配。这种设计确保了在各种环境下都能提供可靠的检索服务。评分算法结合了语义相似度、时间衰减和重要性权重，最终得分公式为：(相似度 × 时间衰减) × (0.8 + 重要性 × 0.4)。

(2) 情景记忆 (EpisodicMemory)

情景记忆负责存储具体的事情和经历，它的设计重点在于保持事件的完整性和时间序列关系。情景记忆采用了SQLite+Qdrant的混合存储方案，SQLite负责结构化数据的存储和复杂查询，Qdrant负责高效的向量检索。

```

class EpisodicMemory:
    """情景记忆实现

    file:///Users/knight/workspace/hello-agents/docs/Hello-Agents-Full.html

```

特点:

- SQLite+Qdrant混合存储架构
- 支持时间序列和会话级检索
- 结构化过滤 + 语义向量检索

```
def __init__(self, config: MemoryConfig):
    self.doc_store = SQLiteDocumentStore(config.database_path)
    self.vector_store = QdrantVectorStore(config.qdrant_url, config.qdrant_api_key)
    self.embedder = create_embedding_model_with_fallback()
    self.sessions = {} # 会话索引

def add(self, memory_item: MemoryItem) -> str:
    """添加情景记忆"""
    # 创建情景对象
    episode = Episode(
        episode_id=memory_item.id,
        session_id=memory_item.metadata.get("session_id", "default"),
        timestamp=memory_item.timestamp,
        content=memory_item.content,
        context=memory_item.metadata
    )

    # 更新会话索引
    session_id = episode.session_id
    if session_id not in self.sessions:
        self.sessions[session_id] = []
    self.sessions[session_id].append(episode.episode_id)

    # 持久化存储 (SQLite + Qdrant)
    self._persist_episode(episode)
    return memory_item.id

def retrieve(self, query: str, limit: int = 5, **kwargs) -> List[MemoryItem]:
    """混合检索: 结构化过滤 + 语义向量检索"""
    # 1. 结构化预过滤 (时间范围、重要性等)
    candidate_ids = self._structured_filter(**kwargs)

    # 2. 向量语义检索
    hits = self._vector_search(query, limit * 5, kwargs.get("user_id"))

    # 3. 综合评分与排序
    results = []
    for hit in hits:
        if self._should_include(hit, candidate_ids, kwargs):
            score = self._calculate_episode_score(hit)
            memory_item = self._create_memory_item(hit)
            results.append((score, memory_item))

    results.sort(key=lambda x: x[0], reverse=True)
    return [item for _, item in results[:limit]]

def _calculate_episode_score(self, hit) -> float:
    """情景记忆评分算法"""
    vec_score = float(hit.get("score", 0.0))
    recency_score = self._calculate_recency(hit["metadata"]["timestamp"])
    importance = hit["metadata"].get("importance", 0.5)

    # 评分公式: (向量相似度 × 0.8 + 时间近因性 × 0.2) × 重要性权重
    base_relevance = vec_score * 0.8 + recency_score * 0.2
    importance_weight = 0.8 + (importance * 0.4)

    return base_relevance * importance_weight
```

情景记忆的检索实现展现了复杂的多因素评分机制。它不仅考虑了语义相似度，还加入了时间近因性的考量，最终通过重要性权重进行调节。评分公式为：
 $(\text{向量相似度} \times 0.8 + \text{时间近因性} \times 0.2) \times (\text{重要性} \times 0.4)$ ，确保检索结果既语义相关又时间相关。

(3) 语义记忆 (SemanticMemory)

语义记忆是记忆系统中最复杂的部分，它负责存储抽象的概念、规则和知识。语义记忆的设计重点在于知识的结构化表示和智能推理能力。语义记忆采用了Neo4j图数据库和Qdrant向量数据库的混合架构，这种设计让系统既能进行快速的语义检索，又能利用知识图谱进行复杂的关系推理。

```
class SemanticMemory(BaseMemory):
    """语义记忆实现

    特点:
    - 使用HuggingFace中文预训练模型进行文本嵌入
    - 向量检索进行快速相似度匹配
    - 知识图谱存储实体和关系
    - 混合检索策略: 向量+图+语义推理
    """

    def __init__(self, config: MemoryConfig, storage_backend=None):
        super().__init__(config, storage_backend)

        # 嵌入模型 (统一提供)
        self.embedding_model = get_text_embedder()

        # 专业数据库存储
        self.vector_store = QdrantConnectionManager.get_instance(**qdrant_config)
```

```

    self.graph_store = Neo4jGraphStore(**neo4j_config)

    # 实体和关系缓存
    self.entities: Dict[str, Entity] = {}
    self.relations: List[Relation] = []

    # NLP处理器 (支持中英文)
    self.nlp = self._init_nlp()

```

语义记忆的添加过程体现了知识图谱构建的完整流程。系统不仅存储记忆内容，还会自动提取实体和关系，构建结构化的知识表示：

```

def add(self, memory_item: MemoryItem) -> str:
    """添加语义记忆"""
    # 1. 生成文本嵌入
    embedding = self.embedding_model.encode(memory_item.content)

    # 2. 提取实体和关系
    entities = self._extract_entities(memory_item.content)
    relations = self._extract_relations(memory_item.content, entities)

    # 3. 存储到Neo4j图数据库
    for entity in entities:
        self._add_entity_to_graph(entity, memory_item)

    for relation in relations:
        self._add_relation_to_graph(relation, memory_item)

    # 4. 存储到Qdrant向量数据库
    metadata = {
        "memory_id": memory_item.id,
        "entities": [e.entity_id for e in entities],
        "entity_count": len(entities),
        "relation_count": len(relations)
    }

    self.vector_store.add_vectors(
        vectors=[embedding.tolist()],
        metadata=[metadata],
        ids=[memory_item.id]
    )

```

语义记忆的检索实现了混合搜索策略，结合了向量检索的语义理解能力和图检索的关系推理能力：

```

def retrieve(self, query: str, limit: int = 5, **kwargs) -> List[MemoryItem]:
    """检索语义记忆"""
    # 1. 向量检索
    vector_results = self._vector_search(query, limit * 2, user_id)

    # 2. 图检索
    graph_results = self._graph_search(query, limit * 2, user_id)

    # 3. 混合排序
    combined_results = self._combine_and_rank_results(
        vector_results, graph_results, query, limit
    )

    return combined_results[:limit]

```

混合排序算法采用了多因素评分机制：

```

def _combine_and_rank_results(self, vector_results, graph_results, query, limit):
    """混合排序结果"""
    combined = {}

    # 合并向量和图检索结果
    for result in vector_results:
        combined[result["memory_id"]] = {
            **result,
            "vector_score": result.get("score", 0.0),
            "graph_score": 0.0
        }

    for result in graph_results:
        memory_id = result["memory_id"]
        if memory_id in combined:
            combined[memory_id]["graph_score"] = result.get("similarity", 0.0)
        else:
            combined[memory_id] = {
                **result,
                "vector_score": 0.0,
                "graph_score": result.get("similarity", 0.0)
            }

    # 计算混合分数
    for memory_id, result in combined.items():
        vector_score = result["vector_score"]
        graph_score = result["graph_score"]
        importance = result.get("importance", 0.5)

        # 基础相似度得分
        base_relevance = vector_score * 0.7 + graph_score * 0.3

        # 重要性权重 [0.8, 1.2]

```

```

importance_weight = 0.8 + (importance * 0.4)

# 最终得分: 相似度 * 重要性权重
combined_score = base_relevance * importance_weight
result["combined_score"] = combined_score

# 排序并返回
sorted_results = sorted(
    combined.values(),
    key=lambda x: x["combined_score"],
    reverse=True
)

return sorted_results[:limit]

```

语义记忆的评分公式为: $(\text{向量相似度} \times 0.7 + \text{图相似度} \times 0.3) \times (0.8 + \text{重要性} \times 0.4)$ 。这种设计的核心思想是:

- 向量检索权重 (0.7) : 语义相似度是主要因素, 确保检索结果与查询语义相关
- 图检索权重 (0.3) : 关系推理作为补充, 发现概念间的隐含关联
- 重要性权重范围[0.8, 1.2]: 避免重要性过度影响相似度排序, 保持检索的准确性

(4) 感知记忆 (PerceptualMemory)

感知记忆支持文本、图像、音频等多种模态的数据存储和检索。它采用了模态分离的存储策略, 为不同模态的数据创建独立的向量集合, 这种设计避免了维度不匹配的问题, 同时保证了检索的准确性:

```

class PerceptualMemory(BaseMemory):
    """感知记忆实现

    特点:
    - 支持多模态数据 (文本、图像、音频等)
    - 跨模态相似性搜索
    - 感知数据的语义理解
    - 支持内容生成和检索
    """

    def __init__(self, config: MemoryConfig, storage_backend=None):
        super().__init__(config, storage_backend)

        # 多模态编码器
        self._text_embedder = get_text_embedder()
        self._clip_model = self._init_clip_model()  # 图像编码
        self._clap_model = self._init_clap_model()  # 音频编码

        # 按模态分离的向量存储
        self._vector_stores = {
            "text": QdrantConnectionManager.get_instance(
                collection_name="perceptual_text",
                vector_size=self._vector_dim
            ),
            "image": QdrantConnectionManager.get_instance(
                collection_name="perceptual_image",
                vector_size=self._image_dim
            ),
            "audio": QdrantConnectionManager.get_instance(
                collection_name="perceptual_audio",
                vector_size=self._audio_dim
            )
        }
    }

```

感知记忆的检索支持同模态和跨模态两种模式。同模态检索利用专业的编码器进行精确匹配, 而跨模态检索则需要更复杂的语义对齐机制:

```

def retrieve(self, query: str, limit: int = 5, **kwargs) -> List[MemoryItem]:
    """检索感知记忆 (可筛选模态; 同模态向量检索+时间/重要性融合) """
    user_id = kwargs.get("user_id")
    target_modality = kwargs.get("target_modality")
    query_modality = kwargs.get("query_modality", target_modality or "text")

    # 同模态向量检索
    try:
        query_vector = self._encode_data(query, query_modality)
        store = self._get_vector_store_for_modality(target_modality or query_modality)

        where = {"memory_type": "perceptual"}
        if user_id:
            where["user_id"] = user_id
        if target_modality:
            where["modality"] = target_modality

        hits = store.search_similar(
            query_vector=query_vector,
            limit=max(limit * 5, 20),
            where=where
        )
    except Exception:
        hits = []

    # 融合排序 (向量相似度 + 时间近因性 + 重要性权重)
    results = []
    for hit in hits:

```

```

vector_score = float(hit.get("score", 0.0))
recency_score = self._calculate_recency_score(hit["metadata"]["timestamp"])
importance = hit["metadata"].get("importance", 0.5)

# 评分算法
base_relevance = vector_score * 0.8 + recency_score * 0.2
importance_weight = 0.8 + (importance * 0.4)
combined_score = base_relevance * importance_weight

results.append((combined_score, self._create_memory_item(hit)))

results.sort(key=lambda x: x[0], reverse=True)
return [item for _, item in results[:limit]]

```

感知记忆的评分公式为：(向量相似度 $\times 0.8$ + 时间近因性 $\times 0.2$) $\times (0.8 + \text{重要性} \times 0.4)$ 。感知记忆的评分机制还支持跨模态检索，通过统一的向量空间实现文本、图像、音频等不同模态数据的语义对齐。当进行跨模态检索时，系统会自动调整评分权重，确保检索结果的多样性和准确性。此外，感知记忆中的时间近因性计算采用了指数衰减模型：

```

def _calculate_recency_score(self, timestamp: str) -> float:
    """计算时间近因性得分"""
    try:
        memory_time = datetime.fromisoformat(timestamp)
        current_time = datetime.now()
        age_hours = (current_time - memory_time).total_seconds() / 3600

        # 指数衰减：24小时内保持高分，之后逐渐衰减
        decay_factor = 0.1 # 衰减系数
        recency_score = math.exp(-decay_factor * age_hours / 24)

        return max(0.1, recency_score) # 最低保持0.1的基础分数
    except Exception:
        return 0.5 # 默认中等分数

```

这种时间衰减模型模拟了人类记忆中的遗忘曲线，确保了感知记忆系统能够优先检索到时间上更相关的记忆内容。

8.3 RAG系统：知识检索增强

8.3.1 RAG的基础知识

在深入HelloAgents的RAG系统实现之前，让我们先了解RAG技术的基础概念、发展历程和核心原理。由于本文内容不是以RAG为基础进行创作，为此这里只帮读者快速梳理相关概念，以便更好地理解系统设计的技术选择和创新点。

(1) 什么是RAG?

检索增强生成（Retrieval-Augmented Generation, RAG）是一种结合了信息检索和文本生成的技术。它的核心思想是：在生成回答之前，先从外部知识库中检索相关信息，然后将检索到的信息作为上下文提供给大语言模型，从而生成更准确、更可靠的回答。

因此，检索增强生成可以拆分为三个词汇。检索是指从知识库中查询相关内容；增强是将检索结果融入提示词，辅助模型生成；生成则输出兼具准确性与透明度的答案。

(2) 基本工作流程

一个完整的RAG应用流程主要分为两大核心环节。在数据准备阶段，系统通过数据提取、文本分割和向量化，将外部知识构建成一个可检索的数据库。随后在应用阶段，系统会响应用户的提问，从数据库中检索相关信息，将其注入Prompt，并最终驱动大语言模型生成答案。

(3) 发展历程

第一阶段：朴素RAG（Naive RAG, 2020-2021）。这是RAG技术的萌芽阶段，其流程直接而简单，通常被称为“检索-读取”（Retrieve-Read）模式。检索方式：主要依赖传统的关键词匹配算法，如TF-IDF或BM25。这些方法计算词频和文档频率来评估相关性，对字面匹配效果好，但难以理解语义上的相似性。生成模式：将检索到的文档内容不加处理地直接拼接到提示词的上下文中，然后送给生成模型。

第二阶段：高级RAG（Advanced RAG, 2022-2023）。随着向量数据库和文本嵌入技术的成熟，RAG进入了快速发展阶段。研究者和开发者们在“检索”和“生成”的各个环节引入了大量优化技术。检索方式：转向基于稠密嵌入（Dense Embedding）的语义检索。通过将文本转换为高维向量，模型能够理解和匹配语义上的相似性，而不仅仅是关键词。生成模式：引入了很多优化技术，例如查询重写，文档分块，重排序等。

第三阶段：模块化RAG（Modular RAG, 2023-至今）。在高级RAG的基础上，现代RAG系统进一步向着模块化、自动化和智能化的方向发展。系统的各个部分被设计成可插拔、可组合的独立模块，以适应更多样化和复杂的应用场景。检索方式：如混合检索，多查询扩展，假设性文档嵌入等。生成模式：思维链推理，自我反思与修正等。

8.3.2 RAG系统工作原理

在深入实现细节之前，可以通过流程图来梳理Helloagents的RAG系统完整工作流程：



图 8.5 RAG系统的核心工作原理

如图8.5所示，展示了RAG系统的两个主要工作模式：1. 数据处理流程：处理和存储知识文档，在这里我们采取工具Markitdown，设计思路是将传入的一切外部知识源统一转化为Markdown格式进行处理。2. 查询与生成流程：根据查询检索相关信息并生成回答。

8.3.3 快速体验：30秒上手RAG功能

让我们先快速体验一下RAG系统的基本功能：

```
from hello_agents import SimpleAgent, HelloAgentsLLM, ToolRegistry
from hello_agents.tools import RAGTool

# 创建具有RAG能力的Agent
llm = HelloAgentsLLM()
agent = SimpleAgent(name="知识助手", llm=llm)

# 创建RAG工具
rag_tool = RAGTool(
    knowledge_base_path=".//knowledge_base",
    collection_name="test_collection",
    rag_namespace="test"
)

tool_registry = ToolRegistry()
tool_registry.register_tool(rag_tool)
agent.tool_registry = tool_registry

# 体验RAG功能
# 添加第一个知识
result1 = rag_tool.execute("add_text",
    text="Python是一种高级编程语言，由Guido van Rossum于1991年首次发布。Python的设计哲学强调代码的可读性和简洁的语法。",
    document_id="python_intro")
print(f"知识1: {result1}")

# 添加第二个知识
result2 = rag_tool.execute("add_text",
    text="机器学习是人工智能的一个分支，通过算法让计算机从数据中学习模式。主要包括监督学习、无监督学习和强化学习三种类型。",
    document_id="ml_basics")
print(f"知识2: {result2}")

# 添加第三个知识
result3 = rag_tool.execute("add_text",
    text="RAG (检索增强生成) 是一种结合信息检索和文本生成的AI技术。它通过检索相关知识来增强大语言模型的生成能力。",
    document_id="rag_concept")
print(f"知识3: {result3}")

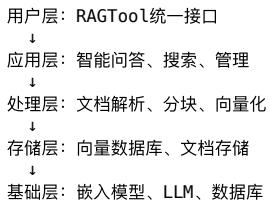
print("\n==== 搜索知识 ====")
result = rag_tool.execute("search",
    query="Python编程语言的历史",
    limit=3,
    min_score=0.1
)
print(result)

print("\n==== 知识库统计 ====")
result = rag_tool.execute("stats")
print(result)
```

接下来，我们将深入探讨HelloAgents RAG系统的具体实现。

8.3.4 RAG系统架构设计

在这一节中，我们采取与记忆系统不同的方式讲解。因为Memory_tool是系统性的实现，而RAG在我们的设计中被定义为一种工具，可以梳理为一条pipeline。我们的RAG系统的核心架构可以概括为“五层七步”的设计模式：



这种分层设计的优势在于每一层都可以独立优化和替换，同时保持整体系统的稳定性。例如，可以轻松地将嵌入模型从sentence-transformers切换到百炼API，而不影响上层的业务逻辑。同样的，这些处理的流程代码是完全可复用的，也可以选取自己需要的部分放进自己的项目中。RAGTool作为RAG系统的统一入口，提供了简洁的API接口。

```
class RAGTool(Tool):
    """RAG工具

    提供完整的 RAG 能力：
    - 添加多格式文档 (PDF、Office、图片、音频等)
    - 智能检索与召回
    - LLM 增强问答
```

- 知识库管理

```

def __init__(
    self,
    knowledge_base_path: str = "./knowledge_base",
    qdrant_url: str = None,
    qdrant_api_key: str = None,
    collection_name: str = "rag_knowledge_base",
    rag_namespace: str = "default"
):
    # 初始化RAG管道
    self._pipelines: Dict[str, Dict[str, Any]] = {}
    self.llm = HelloAgentsLLM()

    # 创建默认管道
    default_pipeline = create_rag_pipeline(
        qdrant_url=self.qdrant_url,
        qdrant_api_key=self.qdrant_api_key,
        collection_name=self.collection_name,
        rag_namespace=self.rag_namespace
    )
    self._pipelines[self.rag_namespace] = default_pipeline

```

整个处理流程如下所示：

任意格式文档 → MarkItDown转换 → Markdown文本 → 智能分块 → 向量化 → 存储检索

(1) 多模态文档载入

RAG系统的核心优势之一是其强大的多模态文档处理能力。系统使用MarkItDown作为统一的文档转换引擎，支持几乎所有常见的文档格式。MarkItDown是微软开源的通用文档转换工具，它是HelloAgents RAG系统的核心组件，负责将任意格式的文档统一转换为结构化的Markdown文本。无论输入是PDF、Word、Excel、图片还是音频，最终都会转换为标准的Markdown格式，然后进入统一的分块、向量化和存储流程。

```

def _convert_to_markdown(path: str) -> str:
    """
    Universal document reader using MarkItDown with enhanced PDF processing.
    核心功能：将任意格式文档转换为Markdown文本

    支持格式：
    - 文档：PDF、Word、Excel、PowerPoint
    - 图像：JPG、PNG、GIF（通过OCR）
    - 音频：MP3、WAV、M4A（通过转录）
    - 文本：TXT、CSV、JSON、XML、HTML
    - 代码：Python、JavaScript、Java等
    """
    if not os.path.exists(path):
        return ""

    # 对PDF文件使用增强处理
    ext = (os.path.splitext(path)[1] or '').lower()
    if ext == '.pdf':
        return _enhanced_pdf_processing(path)

    # 其他格式使用MarkItDown统一转换
    md_instance = _get_markitdown_instance()
    if md_instance is None:
        return _fallback_text_reader(path)

    try:
        result = md_instance.convert(path)
        markdown_text = getattr(result, "text_content", None)
        if isinstance(markdown_text, str) and markdown_text.strip():
            print(f"[RAG] MarkItDown转换成功: {path} -> {len(markdown_text)} chars Markdown")
            return markdown_text
        return ""
    except Exception as e:
        print(f"[WARNING] MarkItDown转换失败 {path}: {e}")
        return _fallback_text_reader(path)

```

(2) 智能分块策略

经过MarkItDown转换后，所有文档都统一为标准的Markdown格式。这为后续的智能分块提供了结构化的基础。HelloAgents实现了专门针对Markdown格式的智能分块策略，充分利用Markdown的结构化特性进行精确分割。

Markdown结构感知的分块流程：

标准Markdown文本 → 标题层次解析 → 段落语义分割 → Token计算分块 → 重叠策略优化 → 向量化准备

| | | | | | |
|--------------|----------|---------------|--------------|----------------|---------------|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 统一格式
结构清晰 | #/##/#/# | 语义边界
完整性保证 | 大小控制
检索优化 | 信息连续性
上下文保持 | 嵌入向量
相似度匹配 |

由于所有文档都已转换为Markdown格式，系统可以利用Markdown的标题结构（#、##、###等）进行精确的语义分割：

```

def _split_paragraphs_with_headings(text: str) -> List[Dict]:
    """
    根据标题层次分割段落，保持语义完整性
    """
    lines = text.splitlines()
    heading_stack: List[str] = []

```

```

paragraphs: List[Dict] = []
buf: List[str] = []
char_pos = 0

def flush_buf(end_pos: int):
    if not buf:
        return
    content = "\n".join(buf).strip()
    if not content:
        return
    paragraphs.append({
        "content": content,
        "heading_path": " > ".join(heading_stack) if heading_stack else None,
        "start": max(0, end_pos - len(content)),
        "end": end_pos,
    })

for ln in lines:
    raw = ln
    if raw.strip().startswith("#"):
        # 处理标题行
        flush_buf(char_pos)
        level = len(raw) - len(raw.lstrip('#'))
        title = raw.lstrip('#').strip()

        if level <= 0:
            level = 1
        if level <= len(heading_stack):
            heading_stack = heading_stack[:level-1]
        heading_stack.append(title)

        char_pos += len(raw) + 1
        continue

    # 段落内容累积
    if raw.strip() == "":
        flush_buf(char_pos)
        buf = []
    else:
        buf.append(raw)
    char_pos += len(raw) + 1

flush_buf(char_pos)

if not paragraphs:
    paragraphs = [{"content": text, "heading_path": None, "start": 0, "end": len(text)}]

return paragraphs

```

在Markdown段落分割的基础上，系统进一步根据Token数量进行智能分块。由于输入已经是结构化的Markdown文本，系统可以更精确地控制分块边界，确保每个分块既适合向量化处理，又保持Markdown结构的完整性：

```

def _chunk_paragraphs(paragraphs: List[Dict], chunk_tokens: int, overlap_tokens: int) -> List[Dict]:
    """基于Token数量的智能分块"""
    chunks: List[Dict] = []
    cur: List[Dict] = []
    cur_tokens = 0
    i = 0

    while i < len(paragraphs):
        p = paragraphs[i]
        p_tokens = _approx_token_len(p["content"]) or 1

        if cur_tokens + p_tokens <= chunk_tokens or not cur:
            cur.append(p)
            cur_tokens += p_tokens
            i += 1
        else:
            # 生成当前分块
            content = "\n\n".join(x["content"] for x in cur)
            start = cur[0]["start"]
            end = cur[-1]["end"]
            heading_path = next((x["heading_path"] for x in reversed(cur) if x.get("heading_path")), None)

            chunks.append({
                "content": content,
                "start": start,
                "end": end,
                "heading_path": heading_path,
            })

            # 构建重叠部分
            if overlap_tokens > 0 and cur:
                kept: List[Dict] = []
                kept_tokens = 0
                for x in reversed(cur):
                    t = _approx_token_len(x["content"]) or 1
                    if kept_tokens + t > overlap_tokens:
                        break
                    kept.append(x)
                    kept_tokens += t
                cur = list(reversed(kept))
                cur_tokens = kept_tokens
            else:
                cur = []
                cur_tokens = 0

```

```

# 处理最后一个分块
if cur:
    content = "\n\n".join(x["content"] for x in cur)
    start = cur[0]["start"]
    end = cur[-1]["end"]
    heading_path = next((x["heading_path"] for x in reversed(cur) if x.get("heading_path")), None)

    chunks.append({
        "content": content,
        "start": start,
        "end": end,
        "heading_path": heading_path,
    })

return chunks

```

同时为了兼容不同语言，系统实现了针对中英文混合文本的Token估算算法，这对于准确控制分块大小至关重要：

```

def _approx_token_len(text: str) -> int:
    """近似估计Token长度，支持中英文混合"""
    # CJK字符按1 token计算
    cjk = sum(1 for ch in text if _is_cjk(ch))
    # 其他字符按空白分词计算
    non_cjk_tokens = len([t for t in text.split() if t])
    return cjk + non_cjk_tokens

def _is_cjk(ch: str) -> bool:
    """判断是否为CJK字符"""
    code = ord(ch)
    return (
        0x4E00 <= code <= 0x9FFF or # CJK统一汉字
        0x3400 <= code <= 0x4DBF or # CJK扩展A
        0x20000 <= code <= 0x2A6DF or # CJK扩展B
        0x2A700 <= code <= 0x2B73F or # CJK扩展C
        0x2B740 <= code <= 0x2B81F or # CJK扩展D
        0x2B820 <= code <= 0x2CEAF or # CJK扩展E
        0xF900 <= code <= 0xFAFF      # CJK兼容汉字
    )

```

(3) 统一嵌入与向量存储

嵌入模型是RAG系统的核心，它负责将文本转换为高维向量，使得计算机能够理解和比较文本的语义相似性。RAG系统的检索能力很大程度上取决于嵌入模型的质量和向量存储的效率。HelloAgents实现了统一的嵌入接口。在这里为了演示，使用百炼API，如果尚未配置可以切换为本地的all-MiniLM-L6-v2模型，如果两种方案都不支持，也配置了TF-IDF算法来兜底。实际使用可以替换成自己想要的模型或者API，也可以尝试去扩展框架内容~

```

def index_chunks(
    store = None,
    chunks: List[Dict] = None,
    cache_db: Optional[str] = None,
    batch_size: int = 64,
    rag_namespace: str = "default"
) -> None:
    """
    Index markdown chunks with unified embedding and Qdrant storage.
    Uses百炼 API with fallback to sentence-transformers.
    """

    if not chunks:
        print("[RAG] No chunks to index")
        return

    # 使用统一嵌入模型
    embedder = get_text_embedder()
    dimension = get_dimension(384)

    # 创建默认Qdrant存储
    if store is None:
        store = _create_default_vector_store(dimension)
        print(f"[RAG] Created default Qdrant store with dimension {dimension}")

    # 预处理Markdown文本以获得更好的嵌入质量
    processed_texts = []
    for c in chunks:
        raw_content = c["content"]
        processed_content = _preprocess_markdown_for_embedding(raw_content)
        processed_texts.append(processed_content)

    print(f"[RAG] Embedding start: total_texts={len(processed_texts)} batch_size={batch_size}")

    # 批量编码
    vecs: List[List[float]] = []
    for i in range(0, len(processed_texts), batch_size):
        part = processed_texts[i:i+batch_size]
        try:
            # 使用统一嵌入器（内部处理缓存）
            part_vecs = embedder.encode(part)

            # 标准化为List[List[float]]格式
            if not isinstance(part_vecs, list):
                if hasattr(part_vecs, "tolist"):
                    part_vecs = [part_vecs.tolist()]

```

```

else:
    part_vecs = [list(part_vecs)]

# 处理向量格式和维度
for v in part_vecs:
    try:
        if hasattr(v, "tolist"):
            v = v.tolist()
        v_norm = [float(x) for x in v]

        # 维度检查和调整
        if len(v_norm) != dimension:
            print(f"[WARNING] 向量维度异常：期望{dimension}，实际{len(v_norm)}")
            if len(v_norm) < dimension:
                v_norm.extend([0.0] * (dimension - len(v_norm)))
            else:
                v_norm = v_norm[:dimension]

        vecs.append(v_norm)
    except Exception as e:
        print(f"[WARNING] 向量转换失败：{e}，使用零向量")
        vecs.append([0.0] * dimension)

except Exception as e:
    print(f"[WARNING] Batch {i} encoding failed: {e}")
    # 实现重试机制
    # ... 重试逻辑 ...
print(f"[RAG] Embedding progress: {min(i+batch_size, len(processed_texts))}/{len(processed_texts)}")

```

8.3.5 高级检索策略

RAG系统的检索能力是其核心竞争力。在实际应用中，用户的查询表述与文档中的实际内容可能存在用词差异，导致相关文档无法被检索到。为了解决这个问题，HelloAgents实现了三种互补的高级检索策略：多查询扩展（MQE）、假设文档嵌入（HyDE）和统一的扩展检索框架。

(1) 多查询扩展（MQE）

多查询扩展（Multi-Query Expansion）是一种通过生成语义等价的多样化查询来提高检索召回率的技术。这种方法的核心洞察是：同一个问题可以有多种不同的表述方式，而不同的表述可能匹配到不同的相关文档。例如，“如何学习Python”可以扩展为“Python入门教程”、“Python学习方法”、“Python编程指南”等多个查询。通过并行执行这些扩展查询并合并结果，系统能够覆盖更广泛的相关文档，避免因用词差异而遗漏重要信息。

MQE的优势在于它能够自动理解用户查询的多种可能含义，特别是对于模糊查询或专业术语查询效果显著。系统使用LLM生成扩展查询，确保扩展的多样性和语义相关性：

```

def _prompt_mqe(query: str, n: int) -> List[str]:
    """使用LLM生成多样化的查询扩展"""
    try:
        from ...core.llm import HelloAgentsLLM
        llm = HelloAgentsLLM()
        prompt = [
            {"role": "system", "content": "你是检索查询扩展助手。生成语义等价或互补的多样化查询。使用中文，简短，避免标点。"},
            {"role": "user", "content": f"原始查询: {query}\n请给出{n}个不同表述的查询，每行一个。"}
        ]
        text = llm.invoke(prompt)
        lines = [ln.strip("- \t") for ln in (text or "").splitlines()]
        outs = [ln for ln in lines if ln]
        return outs[:n] or [query]
    except Exception:
        return [query]

```

(2) 假设文档嵌入（HyDE）

假设文档嵌入（Hypothetical Document Embeddings，HyDE）是一种创新的检索技术，它的核心思想是“用答案找答案”。传统的检索方法是用问题去匹配文档，但问题和答案在语义空间中的分布往往存在差异——问题通常是疑问句，而文档内容是陈述句。HyDE通过让LLM先生成一个假设性的答案段落，然后用这个答案段落去检索真实文档，从而缩小了查询和文档之间的语义鸿沟。

这种方法的优势在于，假设答案与真实答案在语义空间中更加接近，因此能够更准确地匹配到相关文档。即使假设答案的内容不完全正确，它所包含的关键术语、概念和表述风格也能有效引导检索系统找到正确的文档。特别是对于专业领域的查询，HyDE能够生成包含领域术语的假设文档，显著提升检索精度：

```

def _prompt_hyde(query: str) -> Optional[str]:
    """生成假设性文档用于改善检索"""
    try:
        from ...core.llm import HelloAgentsLLM
        llm = HelloAgentsLLM()
        prompt = [
            {"role": "system", "content": "根据用户问题，先写一段可能的答案性段落，用于向量检索的查询文档（不要分析过程。"""),
            {"role": "user", "content": f"问题: {query}\n请直接写一段中等长度、客观、包含关键术语的段落。"}
        ]
        return llm.invoke(prompt)
    except Exception:
        return None

```

(3) 扩展检索框架

HelloAgents将MQE和HyDE两种策略整合到统一的扩展检索框架中。系统通过enable_mqe和enable_hyde参数让用户可以根据具体场景选择启用哪些策略：对于需要高召回率的场景可以同时启用两种策略，对于性能敏感的场景可以只使用基础检索。

扩展检索的核心机制是“扩展-检索-合并”三步流程。首先，系统根据原始查询生成多个扩展查询（包括MQE生成的多样化查询和HyDE生成的假设文档）；然后，对每个扩展查询并行执行向量检索，获取候选文档池；最后，通过去重和分数排序合并所有结果，返回最相关的top-k文档。这种设计的巧妙之处在于，它通过candidate_pool_multiplier参数（默认为4）扩大候选池，确保有足够的候选文档进行筛选，同时通过智能去重避免返回重复内容。

```
def search_vectors_expanded(
    store = None,
    query: str = "",
    top_k: int = 8,
    rag_namespace: Optional[str] = None,
    only_rag_data: bool = True,
    score_threshold: Optional[float] = None,
    enable_mqe: bool = False,
    mqe_expansions: int = 2,
    enable_hyde: bool = False,
    candidate_pool_multiplier: int = 4,
) -> List[Dict]:
    """
    Search with query expansion using unified embedding and Qdrant.
    """
    if not query:
        return []

    # 创建默认存储
    if store is None:
        store = _create_default_vector_store()

    # 查询扩展
    expansions: List[str] = [query]

    if enable_mqe and mqe_expansions > 0:
        expansions.extend(_prompt_mqe(query, mqe_expansions))
    if enable_hyde:
        hyde_text = _prompt_hyde(query)
        if hyde_text:
            expansions.append(hyde_text)

    # 去重和修剪
    uniq: List[str] = []
    for e in expansions:
        if e and e not in uniq:
            uniq.append(e)
    expansions = uniq[: max(1, len(uniq))]

    # 分配候选池
    pool = max(top_k * candidate_pool_multiplier, 20)
    per = max(1, pool // max(1, len(expansions)))

    # 构建RAG数据过滤器
    where = {"memory_type": "rag_chunk"}
    if only_rag_data:
        where["is_rag_data"] = True
        where["data_source"] = "rag_pipeline"
    if rag_namespace:
        where["rag_namespace"] = rag_namespace

    # 收集所有扩展查询的结果
    agg: Dict[str, Dict] = {}
    for q in expansions:
        qv = embed_query(q)
        hits = store.search_similar(
            query_vector=qv,
            limit=per,
            score_threshold=score_threshold,
            where=where
        )
        for h in hits:
            mid = h.get("metadata", {}).get("memory_id", h.get("id"))
            s = float(h.get("score", 0.0))
            if mid not in agg or s > float(agg[mid].get("score", 0.0)):
                agg[mid] = h

    # 按分数排序返回
    merged = list(agg.values())
    merged.sort(key=lambda x: float(x.get("score", 0.0)), reverse=True)
    return merged[:top_k]
```

实际应用中，这三种策略的组合使用效果最佳。MQE擅长处理用词多样性问题，HyDE擅长处理语义鸿沟问题，而统一框架则确保了结果的质量和多样性。对于一般查询，建议启用MQE；对于专业领域查询，建议同时启用MQE和HyDE；对于性能敏感场景，可以只使用基础检索或仅启用MQE。

当然还有很多有趣的方法，这里只是为大家适当的扩展介绍，在实际的使用场景里也需要去尝试寻找适合问题的解决方案。

8.4 构建智能文档问答助手

在前面的章节中，我们详细介绍了HelloAgents的记忆系统和RAG系统的设计与实现。现在，让我们通过一个完整的实战案例，展示如何将这两个系统有机结合，构建一个智能文档问答助手。

8.4.1 案例背景与目标

在实际工作中，我们经常需要处理大量的技术文档、研究论文、产品手册等PDF文件。传统的文档阅读方式效率低下，难以快速定位关键信息，更无法建立知识间的关联。

本案例将基于Datawhale另外一门动手学大模型教程Happy-LLM的公测PDF文档Happy-LLM-0727.pdf为例，构建一个基于Gradio的Web应用，展示如何使用RAGTool和MemoryTool构建完整的交互式学习助手。PDF可在这个[链接](#)获取。

我们希望实现以下功能：

1. 智能文档处理：使用MarkItDown实现PDF到Markdown的统一转换，基于Markdown结构的智能分块策略，高效的向量化和索引构建
2. 高级检索问答：多查询扩展（MQE）提升召回率，假设文档嵌入（HyDE）改善检索精度，上下文感知的智能问答
3. 多层次记忆管理：工作记忆管理当前学习任务和上下文，情景记忆记录学习事件和查询历史，语义记忆存储概念知识和理解，感知记忆处理文档特征和多模态信息
4. 个性化学习支持：基于学习历史的个性化推荐，记忆整合和选择性遗忘，学习报告生成和进度追踪

为了更清晰地展示整个系统的工作流程，图8.6展示了五个步骤之间的关系和数据流动。五个步骤形成了一个完整的闭环：步骤1将PDF文档处理后的信息记录到记忆系统，步骤2的检索结果也会记录到记忆系统，步骤3展示记忆系统的完整功能（添加、检索、整合、遗忘），步骤4整合RAG和Memory提供智能路由，步骤5收集所有统计信息生成学习报告。



图 8.6 智能问答助手的五步执行流程

接下来，我们将展示如何实现这个Web应用。整个应用分为三个核心部分：

1. 核心助手类（PDFLearningAssistant）：封装RAGTool和MemoryTool的调用逻辑
2. Gradio Web界面：提供友好的用户交互界面，这个部分可以参考示例代码学习
3. 其他核心功能：笔记记录、学习回顾、统计查看和报告生成

8.4.2 核心助手类的实现

首先，我们实现核心的助手类PDFLearningAssistant，它封装了RAGTool和MemoryTool的调用逻辑。

(1) 类的初始化

```
class PDFLearningAssistant:
    """智能文档问答助手"""

    def __init__(self, user_id: str = "default_user"):
        """初始化学习助手

        Args:
            user_id: 用户ID，用于隔离不同用户的数据
        """
        self.user_id = user_id
        self.session_id = f"session_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
        # 初始化工具
        self.memory_tool = MemoryTool(user_id=user_id)
        self.rag_tool = RAGTool(rag_namespace=f"pdf_{user_id}")

        # 学习统计
        self.stats = {
            "session_start": datetime.now(),
            "documents_loaded": 0,
            "questions_asked": 0,
            "concepts_learned": 0
        }

        # 当前加载的文档
        self.current_document = None
```

在这个初始化过程中，我们做了几个关键的设计决策：

MemoryTool的初始化：通过user_id参数实现用户级别的记忆隔离。不同用户的学习记忆是完全独立的，每个用户都有自己的工作记忆、情景记忆、语义记忆和感知记忆空间。

RAGTool的初始化：通过rag_namespace参数实现知识库的命名空间隔离。使用f"pdf_{user_id}"作为命名空间，每个用户都有自己独立的PDF知识库。

会话管理：session_id用于追踪单次学习会话的完整过程，便于后续的学习历程回顾和分析。

统计信息：stats字典记录关键的学习指标，用于生成学习报告。

(2) 加载PDF文档

```

def load_document(self, pdf_path: str) -> Dict[str, Any]:
    """加载PDF文档到知识库

Args:
    pdf_path: PDF文件路径

Returns:
    Dict: 包含success和message的结果
    .....
if not os.path.exists(pdf_path):
    return {"success": False, "message": f"文件不存在: {pdf_path}"}

start_time = time.time()

# 【RAGTool】处理PDF: MarkItDown转换 → 智能分块 → 向量化
result = self.rag_tool.execute(
    "add_document",
    file_path=pdf_path,
    chunk_size=1000,
    chunk_overlap=200
)

process_time = time.time() - start_time

if result.get("success", False):
    self.current_document = os.path.basename(pdf_path)
    self.stats["documents_loaded"] += 1

    # 【MemoryTool】记录到学习记忆
    self.memory_tool.execute(
        "add",
        content=f"加载了文档《{self.current_document}》",
        memory_type="episodic",
        importance=0.9,
        event_type="document_loaded",
        session_id=self.session_id
    )

    return {
        "success": True,
        "message": f"加载成功! (耗时: {process_time:.1f}秒)",
        "document": self.current_document
    }
else:
    return {
        "success": False,
        "message": f"加载失败: {result.get('error', '未知错误')}"
}

```

我们通过一行代码就能完成PDF的处理:

```

result = self.rag_tool.execute(
    "add_document",
    file_path=pdf_path,
    chunk_size=1000,
    chunk_overlap=200
)

```

这个调用会触发RAGTool的完整处理流程（MarkItDown转换、增强处理、智能分块、向量化存储），这些内部细节在8.3节已经详细介绍过。我们只需要关注:

- 操作类型: "add_document" - 添加文档到知识库
- 文件路径: file_path - PDF文件的路径
- 分块参数: chunk_size=1000, chunk_overlap=200 - 控制文本分块
- 返回结果: 包含处理状态和统计信息的字典

文档加载成功后，我们使用MemoryTool记录到情景记忆:

```

self.memory_tool.execute(
    "add",
    content=f"加载了文档《{self.current_document}》",
    memory_type="episodic",
    importance=0.9,
    event_type="document_loaded",
    session_id=self.session_id
)

```

为什么用情景记忆? 因为这是一个具体的、有时间戳的事件，适合用情景记忆记录。`session_id`参数将这个事件关联到当前学习会话，便于后续回顾学习历程。

这个记忆记录为后续的个性化服务奠定了基础:

- 用户询问“我之前加载过哪些文档？” → 从情景记忆中检索
- 系统可以追踪用户的学习历程和文档使用情况

8.4.3 智能问答功能

文档加载完成后，用户就可以向文档提问了。我们实现一个ask方法来处理用户的问题：

```
def ask(self, question: str, use_advanced_search: bool = True) -> str:
    """向文档提问

    Args:
        question: 用户问题
        use_advanced_search: 是否使用高级检索 (MQE + HyDE)

    Returns:
        str: 答案
    """
    if not self.current_document:
        return "⚠ 请先加载文档!"

    # 【MemoryTool】记录问题到工作记忆
    self.memory_tool.execute(
        "add",
        content=f"提问: {question}",
        memory_type="working",
        importance=0.6,
        session_id=self.session_id
    )

    # 【RAGTool】使用高级检索获取答案
    answer = self.rag_tool.execute(
        "ask",
        question=question,
        limit=5,
        enable_advanced_search=use_advanced_search,
        enable_mqe=use_advanced_search,
        enable_hyde=use_advanced_search
    )

    # 【MemoryTool】记录到情景记忆
    self.memory_tool.execute(
        "add",
        content=f"关于'{question}'的学习",
        memory_type="episodic",
        importance=0.7,
        event_type="qa_interaction",
        session_id=self.session_id
    )

    self.stats["questions_asked"] += 1
    return answer
```

当我们调用self.rag_tool.execute("ask", ...)时，RAGTool内部执行了以下高级检索流程：

1. 多查询扩展 (MQE) :

```
# 生成多样化查询
expanded_queries = self._generate_multi_queries(question)
# 例如，对于“什么是大语言模型？”，可能生成：
# - “大语言模型的定义是什么？”
# - “请解释一下大语言模型”
# - “LLM是什么意思？”
```

MQE通过LLM生成语义等价但表述不同的查询，从多个角度理解用户意图，提升召回率30%-50%。

2. 假设文档嵌入 (HyDE) :

- 生成假设答案文档，桥接查询和文档的语义鸿沟
- 使用假设答案的向量进行检索

这些高级检索技术的内部实现在8.3.5节已经详细介绍过。

8.4.4 其他核心功能

除了加载文档和智能问答，我们还需要实现笔记记录、学习回顾、统计查看和报告生成等功能：

```
def add_note(self, content: str, concept: Optional[str] = None):
    """添加学习笔记"""
    self.memory_tool.execute(
        "add",
        content=content,
        memory_type="semantic",
        importance=0.8,
        concept=concept or "general",
        session_id=self.session_id
    )
    self.stats["concepts_learned"] += 1

def recall(self, query: str, limit: int = 5) -> str:
    """回顾学习历程"""
    result = self.memory_tool.execute(
        "search",
```

```

        query=query,
        limit=limit
    )
    return result

def get_stats(self) -> Dict[str, Any]:
    """获取学习统计"""
    duration = (datetime.now() - self.stats["session_start"]).total_seconds()
    return {
        "会话时长": f"{duration:.0f}秒",
        "加载文档": self.stats["documents_loaded"],
        "提问次数": self.stats["questions_asked"],
        "学习笔记": self.stats["concepts_learned"],
        "当前文档": self.current_document or "未加载"
    }

def generate_report(self, save_to_file: bool = True) -> Dict[str, Any]:
    """生成学习报告"""
    memory_summary = self.memory_tool.execute("summary", limit=10)
    rag_stats = self.rag_tool.execute("stats")

    duration = (datetime.now() - self.stats["session_start"]).total_seconds()
    report = {
        "session_info": {
            "session_id": self.session_id,
            "user_id": self.user_id,
            "start_time": self.stats["session_start"].isoformat(),
            "duration_seconds": duration
        },
        "learning_metrics": {
            "documents_loaded": self.stats["documents_loaded"],
            "questions_asked": self.stats["questions_asked"],
            "concepts_learned": self.stats["concepts_learned"]
        },
        "memory_summary": memory_summary,
        "rag_status": rag_stats
    }

    if save_to_file:
        report_file = f"learning_report_{self.session_id}.json"
        with open(report_file, 'w', encoding='utf-8') as f:
            json.dump(report, f, ensure_ascii=False, indent=2, default=str)
        report["report_file"] = report_file

    return report

```

这些方法分别实现了：

- **add_note**: 将学习笔记保存到语义记忆
- **recall**: 从记忆系统中检索学习历程
- **get_stats**: 获取当前会话的统计信息
- **generate_report**: 生成详细的学习报告并保存为JSON文件

8.4.5 运行效果展示

接下来是运行效果展示，如图8.7所示，进入主页面后需要先初始化助手，也就是加载我们的数据库，模型，API之类的载入操作。后传入PDF文档，并点击加载文档。



图 8.7 问答助手主页面

第一个功能是智能问答，将可以基于上传的文档进行检索，并返回参考来源和相关资料的相似度计算，这是RAG tool能力的体现，如图8.8所示。



图 8.8 问答助手主页面

第二个功能是学习笔记，如图8.9所示，可以对于相关概念进行勾选，以及撰写笔记内容，这一部分运用到Memory tool，将会存放你的个人笔记在数据库内，方便统计和后续返回整体的学习报告。



图 8.9 问答助手主页面

最后是学习进度的统计和报告的生成，如图8.10所示，我们将可以看到使用助手期间加载的文档数量，提问次数，和笔记数量，最终将我们的问答结果和笔记整理为一个JSON文档返回。



图 8.10 问答助手主页面

通过这个问答助手的案例，我们展示了如何使用RAGTool和MemoryTool构建一个完整的基于Web的智能文档问答系统。完整的代码可以在 `code/chapter8/11_Q&A_Assistant.py` 中找到。启动后访问 `http://localhost:7860` 即可使用这个智能学习助手。

建议读者亲自运行这个案例，体验RAG和Memory的能力，并在此基础上进行扩展和定制，构建符合自己需求的智能应用！

8.5 本章总结与展望

在本章中，我们成功地为HelloAgents框架增加了两个核心能力：记忆系统和RAG系统。

对于希望深入学习和应用本章内容的读者，我们提供以下建议：

1. 从零到一，亲手设计一个基础记忆模块，并逐步迭代，为其增添更复杂的特性。
2. 在项目中尝试并评估不同的嵌入模型与检索策略，寻找特定任务下的最优解。
3. 将所学的记忆与 RAG 系统应用于一个真实的个人项目，在实战中检验和提升能力。

进阶探索

1. 跟踪并研究前沿memory，rag仓库，学习优秀实现。
2. 探索将 RAG 架构应用于多模态（文本+图像）或跨模态场景的可能性。
3. 参与HelloAgents开源项目，贡献自己的想法和代码

通过本章的学习，您不仅掌握了Memory和RAG系统的实现技术，更重要的是理解了如何将认知科学理论转化为实际的工程解决方案。这种跨学科的思维方式，将为您在AI领域的进一步发展奠定坚实的基础。

最后，让我们通过一个思维导图来总结本章的完整知识体系，如图8.11所示：



图 8.11 Hello-agents第八章知识总结

本章展示了HelloAgents框架记忆系统和RAG技术的能力，我们成功构建了一个具有真正“智能”的学习助手。这种架构可以轻松扩展到其他应用场景，如客户服务、技术支持、个人助理等领域。

在下一章中，我们将继续探索如何通过上下文工程进一步提升智能体的对话质量和用户体验，敬请期待！

习题

提示：部分习题没有标准答案，重点在于培养学习者对记忆系统和RAG技术的综合理解和实践能力。

1. 本章介绍了四种记忆类型：工作记忆、情景记忆、语义记忆和感知记忆。请分析：

- 在8.2.5节中，每种记忆类型都有独特的评分公式。请对比情景记忆和语义记忆的评分机制，解释为什么情景记忆更强调“时间近因性”（权重0.2），而语义记忆更强调“图检索”（权重0.3）？
- 如果要设计一个“个人健康管理助手”（需要记录用户的饮食、运动、睡眠数据，并提供健康建议），你会如何组合使用这四种记忆类型？请为每种记忆类型设计具体的应用场景。
- 工作记忆采用TTL（Time To Live）机制自动清理过期数据。请思考：在什么情况下，重要的工作记忆应该被“整合”（consolidate）为长期记忆？如何设计一个自动整合的触发条件？

2. 在8.3节的RAG系统中，我们使用MarkItDown将各种格式文档统一转换为Markdown。请深入思考：

提示：这是一道动手实践题，建议实际操作

- 当前的智能分块策略基于Markdown的标题层次（#、##、###）进行分割。如果处理的是没有明确标题结构的文档（如小说、法律条文），应该如何优化分块策略？请尝试实现一个基于“语义边界”的分块算法。
- 在8.3.5节中介绍了MQE（多查询扩展）和HyDE（假设文档嵌入）两种高级检索策略。请选择一个实际场景（如技术文档问答、医疗知识检索），对比基础检索、MQE和HyDE三种方法的效果差异，并分析各自的适用场景。
- RAG系统的检索质量很大程度上取决于嵌入模型的选择。请对比本章提到的三种嵌入方案（百炼API、本地Transformer、TF-IDF），从准确性、速度、成本、离线部署等维度进行评估，并给出选型建议。

3. 记忆系统的“遗忘”机制是模拟人类认知的重要设计。基于8.2.3节的MemoryTool，请完成以下扩展实践：

提示：这是一道动手实践题，建议实际操作

- 当前提供了三种遗忘策略：基于重要性、基于时间、基于容量。请设计并实现一个“智能遗忘”策略，综合考虑重要性、访问频率、时间衰减等多个因素，使用加权评分来决定哪些记忆应该被遗忘。
- 在长期运行的智能体系统中，记忆数据库可能会积累大量数据。请设计一个“记忆归档”机制：将长期不用但可能有价值的记忆转移到冷存储，需要时再恢复。这个机制应该如何与现有的四种记忆类型集成？
- 思考：如果智能体需要“忘记”某些敏感信息（如用户隐私数据），仅仅从数据库删除是否足够？在使用向量数据库和图数据库的情况下，如何确保数据被彻底清除？

4. 在8.4节的“智能学习助手”案例中，我们结合了MemoryTool和RAGTool。请深入分析：

- 案例中的ask_question()方法同时使用了RAG检索和记忆检索。请分析：在什么情况下应该优先使用RAG？在什么情况下应该优先使用Memory？如何设计一个“智能路由”机制来自动选择最合适检索方式？
- 当前的学习报告(generate_report())只包含统计信息。请扩展这个功能，设计一个更智能的学习报告生成器：能够分析用户的学习轨迹、识别知识盲点、推荐下一步学习内容。这需要用到哪些记忆类型和检索策略？
- 假设你要将这个学习助手部署为多用户的Web服务，每个用户都有独立的记忆和知识库。请设计数据隔离方案：如何在Qdrant和Neo4j中实现用户级别的数据隔离？如何优化多用户场景下的检索性能？

5. 语义记忆使用了Neo4j图数据库来存储知识图谱。请思考：

- 在8.2.5节的语义记忆实现中，系统会自动提取实体和关系构建知识图谱。请分析：这种自动提取的准确性如何？在什么情况下可能会提取出错误的实体或关系？如何设计一个“知识图谱质量评估”机制？
- 知识图谱的一个重要优势是支持复杂的关系推理。请设计一个查询场景，充分利用Neo4j的图查询能力（如多跳关系、路径查找），实现纯向量检索无法完成的任务。
- 对比语义记忆的“向量检索+图检索”混合策略与纯向量检索：在什么类型的查询中，图检索能够带来显著的性能提升？请用具体例子说明。

参考文献

[1] Atkinson, R. C., & Shiffrin, R. M. (1968). Human memory: A proposed system and its control processes. In *Psychology of learning and motivation* (Vol. 2, pp. 89-195). Academic press.

第九章 上下文工程

在前面的章节中，我们已经为智能体引入了记忆系统与RAG。然而，要让智能体在真实复杂场景中稳定地“思考”与“行动”，仅有记忆与检索还不够——我们需要一套工程化方法，持续、系统地为模型构造恰当的“上下文”。这就是本章的主题：上下文工程（Context Engineering）。它关注的是“在每一次模型调用前，如何以可复用、可度量、可演进的方式，拼装并优化输入上下文”，从而提升正确性、鲁棒性与效率^{[1][2]}。

为了让读者能够快速体验本章的完整功能，我们提供了可直接安装的Python包。你可以通过以下命令安装本章对应的版本：

```
pip install "hello-agents[all]==0.2.8"
```

本章主要介绍上下文工程的核心概念与实践，并在HelloAgents框架中新增了上下文构建器和两个配套工具：

- **ContextBuilder** (`hello_agents/context/builder.py`)：上下文构建器，实现 GSSC (Gather-Select-Structure-Compress) 流水线，提供统一的上下文管理接口
- **NoteTool** (`hello_agents/tools/builtin/note_tool.py`)：结构化笔记工具，支持智能体进行持久化记忆管理
- **TerminalTool** (`hello_agents/tools/builtin/terminal_tool.py`)：终端工具，支持智能体进行文件系统操作和即时上下文检索

这些组件共同构成了完整的上下文工程解决方案，是实现长时程任务管理和智能体式搜索的关键，将在后续章节中详细介绍。

除了安装框架外，还需要在`.env`配置LLM的API。本章示例主要使用大语言模型进行上下文管理和智能决策。

配置完成后，即可开始本章的学习之旅！

9.1 什么是上下文工程

在经历了数年提示工程（Prompt Engineering）成为应用型AI的焦点之后，一个新的术语开始走到台前：上下文工程（Context Engineering）。如今，用语言模型构建系统不再只是找对提示词里的句式和措辞，而是要回答一个更宏观的问题：什么样的上下文配置，最有可能让模型产出我们期望的行为？

所谓“上下文”，是指在对大语言模型（LLM）进行采样时所包含的那组 tokens。手头的工程问题，是在 LLM 的固有约束之下，优化这些 **tokens** 的效用，以便稳定地得到预期结果。想要有效驾驭 LLM，往往需要“在上下文中思考”——也就是说：在任何一次调用时，都要审视 LLM 可见的整体状态，并预判这种状态可能诱发的行为。



图 9.1 Prompt engineering vs Context engineering

本节将探讨正在兴起的上下文工程，并给出一个用于构建可调控、有效智能体的精炼心智模型。

上下文工程 vs. 提示工程

如图9.1所示，在现在前沿模型厂商的视角中，上下文工程是提示工程的自然演进。提示工程关注如何编写与组织 LLM 的指令以获得更优结果（例如系统提示的写法与结构化策略）；而上下文工程则是在推理阶段，如何策划与维护“最优的信息集合（tokens）”，其中不仅包含提示本身，还包含其他会进入上下文窗口的一切信息。

在 LLM 工程的早期阶段，提示往往是主要工作，因为大多数用例（除日常聊天外）都需要针对单轮分类或文本生成做精调式的提示优化。顾名思义，提示工程的核心是“如何写出有效提示”，尤其是系统提示。然而，随着我们开始工程化地构建更强的智能体，它们在更长的时间范围内、跨多次推理轮次地工作，我们就需要能管理整个上下文状态的策略——其中包括系统指令、工具、MCP（Model Context Protocol）、外部数据、消息历史等。

一个循环运行的智能体，会不断产生下一轮推理可能相关的数据，这些信息必须被周期性地提炼。因此，上下文工程的“艺与术”，在于从持续扩张的“候选信息宇宙”中，甄别哪些内容应当进入有限的上下文窗口。

9.2 为什么上下文工程重要

尽管模型的速度越来越快、可处理的数据规模越来越大，但我们观察到：LLM 和人类一样，在一定点上会“走神”或“混乱”。针堆找针（needle-in-a-haystack）类基准揭示了一个现象：上下文腐蚀（context rot）——随着上下文窗口中的 tokens 增加，模型从上下文中准确回忆信息的能力反而下降。

不同模型的退化曲线或许更平滑，但这一特征几乎在所有模型上都会出现。因此，上下文必须被视作一种有限资源，且具有边际收益递减。就像人类有有限的工作记忆容量一样，LLM 也有一笔“注意力预算”。每新增一个 token，都会消耗这笔预算的一部分，因此我们更需要谨慎地筛选哪些 tokens 应该被提供给 LLM。

这种稀缺并非偶然，而是源自 LLM 的架构约束。Transformer 让每个 token 能够与上下文中的所有 token 建立关联，理论上形成 (n^2) 级别的两两注意力关系。随着上下文长度增长，模型对这些两两关系的建模能力会被“拉薄”，从而自然地产生“上下文规模”与“注意力集中度”的张力。此外，模型的注意力模式来源于训练数据分布——短序列通常比长序列更常见，因此模型对“全上下文依赖”的经验更少、专门参数也更少。

诸如位置编码插值（position encoding interpolation）等技术可以让模型在推理时“适配”比训练期更长的序列，但会牺牲部分对 token 位置的精确理解。总体上，这些因素共同形成的是一个性能梯度，而非“悬崖式”崩溃：模型在长上下文下依旧强大，但相较小上下文，在信息检索与长程推理上的精度会有所下降。

基于上述现实，有意识的上下文工程就成为构建强健智能体的必需品。

9.2.1 有效上下文的“解剖学”

在“有限注意力预算”的约束下，优秀的上下文工程目标是：用尽可能少、但高信号密度的 **tokens**，最大化获得期望结果的概率。落实到实践中，我们建议围绕以下组件开展工程化建设：

- **系统提示 (System Prompt)**：语言清晰、直白，信息层级把握在“刚刚好”的高度。常见两极误区：
 - 过度硬编码：在提示中写入复杂、脆弱的 if-else 逻辑，长期维护成本高、易碎。
 - 过于空泛：只给出宏观目标与泛化指引，缺少对期望输出的具体信号或假定了错误的“共享上下文”。建议将提示分区组织（如、、工具指引、输出描述等），用 XML/Markdown 分隔。无论格式如何，追求的是能完整勾勒期望行为的“最小必要信息集”（“最小”并不等于“最短”）。先用最好的模型在最小提示上试跑，再依据失败模式增补清晰的指令与示例。
- **工具 (Tools)**：工具定义了智能体与信息/行动空间的契约，必须促进效率：既要返回 **token** 友好的信息，又要鼓励高效的智能体行为。工具应当：
 - 职责单一、相互低重叠，接口语义清晰；
 - 对错误鲁棒；
 - 入参描述明确、无歧义，充分发挥模型擅长的表达与推理能力。常见失败模式是“臃肿工具集”：功能边界模糊，导致“选哪个工具”这一决策本身就含混不清。如果人类工程师都说不准用哪个工具，别指望智能体做得更好。精心甄别一个“最小可行工具集 (MVTs)”往往能显著提升长期交互中的稳定性与可维护性。
- **示例 (Few-shot)**：始终推荐提供示例，但不建议把“所有边界条件”的罗列一股脑塞进提示。请精挑细选一组多样且典型的示例，直接画像“期望行为”。对 LLM 而言，好的示例胜过千言万语。

总的指导思想是：信息充分但紧致。如图9.2所示，是进入运行时的动态检索。



图 9.2 Calibrating the system prompt

9.2.2 上下文检索与智能体式搜索

一个简洁的定义：智能体 = 在循环中自主调用工具的 **LLM**。随着底层模型能力增强，智能体的自治水平便可提升：更能独立探索复杂问题空间，并从错误中恢复。

工程实践正在从“推理前一次性检索 (embedding 检索)”逐步过渡到“及时 (Just-in-time, JIT) 上下文”。后者不再预先加载所有相关数据，而是维护轻量化引用（文件路径、存储查询、URL 等），在运行时通过工具动态加载所需数据。这样可让模型撰写针对性查询、缓存必要结果，并用诸如 head/tail 之类的命令分析大体量数据——无需把整块数据一次性塞入上下文。其认知模式更贴近人类：我们不会死记硬背全部信息，而是用文件系统、收件箱、书签等外部索引按需提取。

除了存储效率，引用的元数据本身也能帮助精化行为：目录层级、命名约定、时间戳等都在隐含地传达“目的与时效”。例如，`tests/test_utils.py` 与 `src/core/test_utils.py` 的语义暗示就不同。

允许智能体自主导航与检索还能实现渐进式披露 (progressive disclosure)：每一步交互都会产生新的上下文，反过来指导下一步决策——文件大小暗示复杂度、命名暗示用途、时间戳暗示相关性。智能体得以按层构建理解，只在工作记忆中保留“当前必要子集”，并用“记笔记”的方式做补充持久化，从而维持聚焦而非“被大而全拖垮”。

需要权衡的是：运行时探索往往比预计算检索更慢，并且需要有“主观”的工程设计来确保模型拥有正确的工具与启发式。如果缺少引导，智能体可能会误用工具、追逐死胡同或错过关键信息，造成上下文浪费。

在不少场景中，混合策略更有效：前置加载少量“高价值”上下文以保证速度，然后允许智能体按需继续自主探索。边界的选择取决于任务动态性与时效要求。在工程上，可以预先放入类似“项目约定说明（如 README/指南）”的文件，同时提供 glob、grep 等原语，让智能体即时检索具体文件，从而绕开过时索引与复杂语法树的沉没成本。

9.2.3 面向长时程任务的上下文工程

长时程任务要求智能体在超出上下文窗口的长序列行动中，仍能保持连贯性、上下文一致与目标导向。例如大型代码库迁移、跨数小时的系统性研究。指望无限增大上下文窗口并不能根治“上下文污染”与相关性退化的问题，因此需要直接面向这些约束的工程手段：压缩整合 (**Compaction**)、结构化笔记 (**Structured note-taking**) 与子代理架构 (**Sub-agent architectures**)。

- **压缩整合 (Compaction)**
 - 定义：当对话接近上下文上限时，对其进行高保真总结，并用该摘要重启一个新的上下文窗口，以维持长程连贯性。
 - 实践：让模型压缩并保留结构性决策、未解决缺陷、实现细节，丢弃重复的工具输出与噪声；新窗口携带压缩摘要 + 最近少量高相关工件（如“最近访问的若干文件”）。
 - 调参建议：先优化召回（确保不遗漏关键信息），再优化精确度（剔除冗余内容）；一种安全的“轻触式”压缩是对“深历史中的工具调用与结果”进行清理。
- **结构化笔记 (Structured note-taking)**
 - 定义：也称“智能体记忆”。智能体以固定频率将关键信息写入上下文外的持久化存储，在后续阶段按需拉回。
 - 价值：以极低的上下文开销维持持久状态与依赖关系。例如维护 TODO 列表、项目 NOTES.md、关键结论/依赖/阻塞项的索引，跨数十次工具调用与多轮上下文重置仍能保持进度与一致性。
 - 说明：在非编码场景中同样有效（如长期策略性任务、游戏/仿真中的目标管理与统计计数）。结合第八章的 **MemoryTool**，可轻松实现文件式/向量式的外部记忆并在运行时检索。
- **子代理架构 (Sub-agent architectures)**

- 思想：由主代理负责高层规划与综合，多个专长子代理在“干净的上下文窗口”中各自深挖、调用工具并探索，最后仅回传凝练摘要（常见 1,000–2,000 tokens）。
- 好处：实现关注点分离。庞杂的搜索上下文留在子代理内部，主代理专注于整合与推理；适合需要并行探索的复杂研究/分析任务。
- 经验：公开的多智能体研究系统显示，该模式在复杂研究任务上相较单代理基线具有显著优势。

方法取舍可以遵循以下经验法则：

- 压缩整合：适合需要长对话连续性的任务，强调上下文的“接力”。
- 结构化笔记：适合有里程碑/阶段性成果的迭代式开发与研究。
- 子代理架构：适合复杂研究与分析，能从并行探索中获益。

即便模型能力持续提升，“在长交互中维持连贯性与聚焦”仍是构建强健智能体的核心挑战。谨慎而系统的上下文工程将长期保持其关键价值。

9.3 在 Hello-Agents 中的实践：ContextBuilder

本节将详细介绍 HelloAgents 框架中的上下文工程实践。我们将从设计动机、核心数据结构、实现细节到完整案例，逐步展示如何构建一个生产级的上下文管理系统。ContextBuilder 的设计理念是“简单高效”，去除不必要的复杂性，统一以“相关性+新近性”的分数进行选择，符合 Agent 模块化与可维护性的工程取向。

9.3.1 设计动机与目标

在构建 ContextBuilder 之前，我们首先需要明确其设计目标和核心价值。一个优秀的上下文管理系统应该解决以下几个关键问题：

- 统一入口：将“获取(Gather)- 选择(Select)- 结构化(Structure)- 压缩(Compress)“抽象为可复用流水线，减少在 Agent 实现中的重复模板代码。这种统一的接口设计让开发者无需在每个 Agent 中重复编写上下文管理逻辑。
- 稳定形态：输出固定骨架的上下文模板，便于调试、A/B 测试与评估。我们采用了分区组织的模板结构：
 - [Role & Policies]: 明确 Agent 的角色定位和行为准则
 - [Task]: 当前需要完成的具体任务
 - [State]: Agent 的当前状态和上下文信息
 - [Evidence]: 从外部知识库检索的证据信息
 - [Context]: 历史对话和相关记忆
 - [Output]: 期望的输出格式和要求
- 预算守护：在 token 预算内尽量保留高价值信息，对超限上下文提供兜底压缩策略。这确保了即使在信息量巨大的场景下，系统也能稳定运行。
- 最小规则：不引入来源/优先级等分类维度，避免复杂度增长。实践表明，基于相关性和新近性的简单评分机制，在大多数场景下已经足够有效。

9.3.2 核心数据结构

ContextBuilder 的实现依赖两个核心数据结构，它们定义了系统的配置和信息单元。

(1) ContextPacket: 候选信息包

```
from dataclasses import dataclass
from typing import Optional, Dict, Any
from datetime import datetime

@dataclass
class ContextPacket:
    """候选信息包

    Attributes:
        content: 信息内容
        timestamp: 时间戳
        token_count: Token 数量
        relevance_score: 相关性分数(0.0-1.0)
        metadata: 可选的元数据
    """
    content: str
    timestamp: datetime
    token_count: int
    relevance_score: float = 0.5
    metadata: Optional[Dict[str, Any]] = None

    def __post_init__(self):
        """初始化后处理"""
        if self.metadata is None:
            self.metadata = {}
        # 确保相关性分数在有效范围内
        self.relevance_score = max(0.0, min(1.0, self.relevance_score))
```

ContextPacket 是系统中信息的基本单元。每个候选信息都会被封装为一个 ContextPacket，包含内容、时间戳、token 数量和相关性分数等核心属性。这种统一的数据结构简化了后续的选择和排序逻辑。

(2) ContextConfig: 配置管理

```

@dataclass
class ContextConfig:
    """上下文构建配置

Attributes:
    max_tokens: 最大 token 数量
    reserve_ratio: 为系统指令预留的比例(0.0-1.0)
    min_relevance: 最低相关性阈值
    enable_compression: 是否启用压缩
    recency_weight: 新近性权重(0.0-1.0)
    relevance_weight: 相关性权重(0.0-1.0)
    ...
    max_tokens: int = 3000
    reserve_ratio: float = 0.2
    min_relevance: float = 0.1
    enable_compression: bool = True
    recency_weight: float = 0.3
    relevance_weight: float = 0.7

def __post_init__(self):
    """验证配置参数"""
    assert 0.0 <= self.reserve_ratio <= 1.0, "reserve_ratio 必须在 [0, 1] 范围内"
    assert 0.0 <= self.min_relevance <= 1.0, "min_relevance 必须在 [0, 1] 范围内"
    assert abs(self.recency_weight + self.relevance_weight - 1.0) < 1e-6, \
        "recency_weight + relevance_weight 必须等于 1.0"

```

ContextConfig 封装了所有可配置的参数，使得系统行为可以灵活调整。特别值得注意的是 `reserve_ratio` 参数，它确保系统指令等关键信息始终有足够的空间，不会被其他信息挤占。

9.3.3 GSSC 流水线详解

ContextBuilder 的核心是 GSSC(Gather-Select-Structure-Compress)流水线，它将上下文构建过程分解为四个清晰的阶段。让我们深入了解每个阶段的实现细节。

(1) Gather: 多源信息汇集

第一阶段是从多个来源汇集候选信息。这个阶段的关键在于容错性和灵活性。

```

def _gather(
    self,
    user_query: str,
    conversation_history: Optional[List[Message]] = None,
    system_instructions: Optional[str] = None,
    custom_packets: Optional[List[ContextPacket]] = None
) -> List[ContextPacket]:
    """汇集所有候选信息

Args:
    user_query: 用户查询
    conversation_history: 对话历史
    system_instructions: 系统指令
    custom_packets: 自定义信息包

Returns:
    List[ContextPacket]: 候选信息列表
    ...
    packets = []

# 1. 添加系统指令(最高优先级,不参与评分)
if system_instructions:
    packets.append(ContextPacket(
        content=system_instructions,
        timestamp=datetime.now(),
        token_count=self._count_tokens(system_instructions),
        relevance_score=1.0, # 系统指令始终保留
        metadata={"type": "system_instruction", "priority": "high"}
    ))
# 2. 从记忆系统检索相关记忆
if self.memory_tool:
    try:
        memory_results = self.memory_tool.run({
            "action": "search",
            "query": user_query,
            "limit": 10,
            "min_importance": 0.3
        })
        # 解析记忆结果并转换为 ContextPacket
        memory_packets = self._parse_memory_results(memory_results, user_query)
        packets.extend(memory_packets)
    except Exception as e:
        print(f"[WARNING] 记忆检索失败: {e}")

# 3. 从 RAG 系统检索相关知识
if self.rag_tool:
    try:
        rag_results = self.rag_tool.run({
            "action": "search",
            "query": user_query,

```

```

        "limit": 5,
        "min_score": 0.3
    })
    # 解析 RAG 结果并转换为 ContextPacket
    rag_packets = self._parse_rag_results(rag_results, user_query)
    packets.extend(rag_packets)
except Exception as e:
    print(f"[WARNING] RAG 检索失败: {e}")

# 4. 添加对话历史(仅保留最近的 N 条)
if conversation_history:
    recent_history = conversation_history[-5:] # 默认保留最近 5 条
    for msg in recent_history:
        packets.append(ContextPacket(
            content=f'{msg.role}: {msg.content}',
            timestamp=msg.timestamp if hasattr(msg, 'timestamp') else datetime.now(),
            token_count=self._count_tokens(msg.content),
            relevance_score=0.6, # 历史消息的基础相关性
            metadata={"type": "conversation_history", "role": msg.role}
        ))

# 5. 添加自定义信息包
if custom_packets:
    packets.extend(custom_packets)

print(f"[ContextBuilder] 汇集了 {len(packets)} 个候选信息包")
return packets

```

这个实现展示了几个重要的设计考虑:

- 容错机制: 每个外部数据源的调用都被 try-except 包裹, 确保单个源的失败不会影响整体流程
- 优先级处理: 系统指令被标记为高优先级, 确保始终被保留
- 历史限制: 对话历史只保留最近的几条, 避免上下文窗口被历史信息占据

(2) Select: 智能信息选择

第二阶段是根据相关性和新近性对候选信息进行评分和选择。这是整个流水线的核心, 直接决定了最终上下文的质量。

```

def _select(
    self,
    packets: List[ContextPacket],
    user_query: str,
    available_tokens: int
) -> List[ContextPacket]:
    """选择最相关的信息包

Args:
    packets: 候选信息包列表
    user_query: 用户查询(用于计算相关性)
    available_tokens: 可用的 token 数量

Returns:
    List[ContextPacket]: 选中的信息包列表
"""

# 1. 分离系统指令和其他信息
system_packets = [p for p in packets if p.metadata.get("type") == "system_instruction"]
other_packets = [p for p in packets if p.metadata.get("type") != "system_instruction"]

# 2. 计算系统指令占用的 token
system_tokens = sum(p.token_count for p in system_packets)
remaining_tokens = available_tokens - system_tokens

if remaining_tokens <= 0:
    print("[WARNING] 系统指令已占满所有 token 预算")
    return system_packets

# 3. 为其他信息计算综合分数
scored_packets = []
for packet in other_packets:
    # 计算相关性分数(如果尚未计算)
    if packet.relevance_score == 0.5: # 默认值,需要重新计算
        relevance = self._calculate_relevance(packet.content, user_query)
        packet.relevance_score = relevance

    # 计算新近性分数
    recency = self._calculate_recency(packet.timestamp)

    # 综合分数 = 相关性权重 * 相关性 + 新近性权重 * 新近性
    combined_score = (
        self.config.relevance_weight * packet.relevance_score +
        self.config.recency_weight * recency
    )

    # 过滤低于最小相关性阈值的信息
    if packet.relevance_score >= self.config.min_relevance:
        scored_packets.append((combined_score, packet))

# 4. 按分数降序排序
scored_packets.sort(key=lambda x: x[0], reverse=True)

```

```
# 5. 贪心选择:按分数从高到低填充,直到达到 token 上限
selected = system_packets.copy()
current_tokens = system_tokens

for score, packet in scored_packets:
    if current_tokens + packet.token_count <= available_tokens:
        selected.append(packet)
        current_tokens += packet.token_count
    else:
        # Token 预算已满,停止选择
        break

print(f"[ContextBuilder] 选择了 {len(selected)} 个信息包,共 {current_tokens} tokens")
return selected
```

```
def _calculate_relevance(self, content: str, query: str) -> float:
    """计算内容与查询的相关性
```

使用简单的关键词重叠算法。在生产环境中,可以替换为向量相似度计算。

Args:

```
    content: 内容文本
    query: 查询文本
```

Returns:

```
    float: 相关性分数(0.0-1.0)
....
```

分词(简单实现,可以使用更复杂的分词器)

```
content_words = set(content.lower().split())
query_words = set(query.lower().split())
```

```
if not query_words:
    return 0.0
```

Jaccard 相似度

```
intersection = content_words & query_words
union = content_words | query_words
```

```
return len(intersection) / len(union) if union else 0.0
```

```
def _calculate_recency(self, timestamp: datetime) -> float:
    """计算时间近因性分数
```

使用指数衰减模型,24小时内保持高分,之后逐渐衰减。

Args:

```
    timestamp: 信息的时间戳
```

Returns:

```
    float: 新近性分数(0.0-1.0)
....
```

import math

```
age_hours = (datetime.now() - timestamp).total_seconds() / 3600
```

指数衰减:24小时内保持高分,之后逐渐衰减

```
decay_factor = 0.1 # 衰减系数
recency_score = math.exp(-decay_factor * age_hours / 24)
```

```
return max(0.1, min(1.0, recency_score)) # 限制在 [0.1, 1.0] 范围内
```

选择阶段的核心算法体现了几个重要的工程考量:

- 评分机制: 采用相关性和新近性的加权组合, 权重可配置
- 贪心算法: 按分数从高到低填充, 确保在有限预算内选择最有价值的信息
- 过滤机制: 通过 min_relevance 参数过滤低质量信息

(3) Structure: 结构化输出

第三阶段是将选中的信息组织成结构化的上下文模板。

```
def _structure(self, selected_packets: List[ContextPacket], user_query: str) -> str:
    """将选中的信息包组织成结构化的上下文模板
```

Args:

```
    selected_packets: 选中的信息包列表
    user_query: 用户查询
```

Returns:

```
    str: 结构化的上下文字符串
....
```

按类型分组

```
system_instructions = []
evidence = []
context = []
```

```
for packet in selected_packets:
    packet_type = packet.metadata.get("type", "general")
```

```
    if packet_type == "system_instruction":
```

```

        system_instructions.append(packet.content)
    elif packet_type in ["rag_result", "knowledge"]:
        evidence.append(packet.content)
    else:
        context.append(packet.content)

# 构建结构化模板
sections = []

# [Role & Policies]
if system_instructions:
    sections.append("[Role & Policies]\n" + "\n".join(system_instructions))

# [Task]
sections.append(f"[Task]\n{user_query}")

# [Evidence]
if evidence:
    sections.append("[Evidence]\n" + "\n---\n".join(evidence))

# [Context]
if context:
    sections.append("[Context]\n" + "\n".join(context))

# [Output]
sections.append("[Output]\n请基于以上信息，提供准确、有据的回答。")

return "\n\n".join(sections)

```

结构化阶段将散乱的信息包组织成清晰的分区，这种设计有几个优势：

- 可读性：清晰的分区让人类和模型都更容易理解上下文结构
- 可调试性：问题定位更容易，可以快速识别哪个区域的信息有问题
- 可扩展性：添加新的信息源只需要创建新的分区

(4) Compress: 兜底压缩

第四阶段是对超限上下文进行压缩处理。

```

def _compress(self, context: str, max_tokens: int) -> str:
    """压缩超限的上下文

    Args:
        context: 原始上下文
        max_tokens: 最大 token 限制

    Returns:
        str: 压缩后的上下文
    """
    current_tokens = self._count_tokens(context)

    if current_tokens <= max_tokens:
        return context # 无需压缩

    print(f"[ContextBuilder] 上下文超限({current_tokens} > {max_tokens}), 执行压缩")

    # 分区压缩:保持结构完整性
    sections = context.split("\n\n")
    compressed_sections = []
    current_total = 0

    for section in sections:
        section_tokens = self._count_tokens(section)

        if current_total + section_tokens <= max_tokens:
            # 完整保留
            compressed_sections.append(section)
            current_total += section_tokens
        else:
            # 部分保留
            remaining_tokens = max_tokens - current_total
            if remaining_tokens > 50: # 至少保留 50 tokens
                # 简单截断(生产环境中可以使用 LLM 摘要)
                truncated = self._truncate_text(section, remaining_tokens)
                compressed_sections.append(truncated + "\n[... 内容已压缩 ...]")
            break

    compressed_context = "\n\n".join(compressed_sections)
    final_tokens = self._count_tokens(compressed_context)
    print(f"[ContextBuilder] 压缩完成: {current_tokens} -> {final_tokens} tokens")

    return compressed_context

def _truncate_text(self, text: str, max_tokens: int) -> str:
    """截断文本到指定 token 数量

    Args:
        text: 原始文本
        max_tokens: 最大 token 数量

    Returns:

```

```

    str: 截断后的文本
    ....
    # 简单实现:按字符比例估算
    # 生产环境中应该使用精确的 tokenizer
    char_per_token = len(text) / self._count_tokens(text) if self._count_tokens(text) > 0 else 4
    max_chars = int(max_tokens * char_per_token)

    return text[:max_chars]

def _count_tokens(self, text: str) -> int:
    """估算文本的 token 数量

    Args:
        text: 文本内容

    Returns:
        int: token 数量
    ....
    # 简单估算:中文 1 字符 ~ 1 token,英文 1 单词 ~ 1.3 tokens
    # 生产环境中应该使用实际的 tokenizer
    chinese_chars = sum(1 for ch in text if '\u4e00' <= ch <= '\u9fff')
    english_words = len([w for w in text.split() if w])

    return int(chinese_chars + english_words * 1.3)

```

压缩阶段的设计体现了“保持结构完整性”的原则，即使在 token 预算紧张的情况下，也要尽量保留每个分区的关键信息。

9.3.4 完整使用示例

现在让我们通过一个完整的示例，展示如何在实际项目中使用 ContextBuilder。

(1) 基础使用

```

from hello_agents.context import ContextBuilder, ContextConfig
from hello_agents.tools import MemoryTool, RAGTool
from hello_agents.core.message import Message
from datetime import datetime

# 1. 初始化工具
memory_tool = MemoryTool(user_id="user123")
rag_tool = RAGTool(knowledge_base_path="./knowledge_base")

# 2. 创建 ContextBuilder
config = ContextConfig(
    max_tokens=3000,
    reserve_ratio=0.2,
    min_relevance=0.2,
    enable_compression=True
)

builder = ContextBuilder(
    memory_tool=memory_tool,
    rag_tool=rag_tool,
    config=config
)

# 3. 准备对话历史
conversation_history = [
    Message(content="我正在开发一个数据分析工具", role="user", timestamp=datetime.now()),
    Message(content="很好!数据分析工具通常需要处理大量数据。您计划使用什么技术栈?", role="assistant", timestamp=datetime.now()),
    Message(content="我打算使用Python和Pandas,已经完成了CSV读取模块", role="user", timestamp=datetime.now()),
    Message(content="不错的选择!Pandas在数据处理方面非常强大。接下来您可能需要考虑数据清洗和转换。", role="assistant", timestamp=datetime.now()),
]

# 4. 添加一些记忆
memory_tool.run({
    "action": "add",
    "content": "用户正在开发数据分析工具,使用Python和Pandas",
    "memory_type": "semantic",
    "importance": 0.8
})

memory_tool.run({
    "action": "add",
    "content": "已完成CSV读取模块的开发",
    "memory_type": "episodic",
    "importance": 0.7
})

# 5. 构建上下文
context = builder.build(
    user_query="如何优化Pandas的内存占用?",
    conversation_history=conversation_history,
    system_instructions="你是一位资深的Python数据工程顾问。你的回答需要:1) 提供具体可行的建议 2) 解释技术原理 3) 给出代码示例"
)

print("=" * 80)
print("构建的上下文:")
print("=" * 80)

```

```
print(context)
print("=" * 80)
```

(2) 运行效果展示

运行上述代码后，您将看到如下结构化的上下文输出：

```
=====
构建的上下文：
=====
[Role & Policies]
你是一位资深的Python数据工程顾问。你的回答需要:1) 提供具体可行的建议 2) 解释技术原理 3) 给出代码示例

[Task]
如何优化Pandas的内存占用？

[Evidence]
Pandas内存优化的核心策略包括：
1. 使用合适的数据类型(如category代替object)
2. 分块读取大文件
3. 使用 chunkszie 参数
---
数据类型优化可以显著减少内存占用。例如,将int64降级为int32可以节省50%的内存。

[Context]
user: 我正在开发一个数据分析工具
assistant: 很好!数据分析工具通常需要处理大量数据。您计划使用什么技术栈？
user: 我打算使用Python和Pandas,已经完成了CSV读取模块
assistant: 不错的选择!Pandas在数据处理方面非常强大。接下来您可能需要考虑数据清洗和转换。
记忆: 用户正在开发数据分析工具,使用Python和Pandas
记忆: 已完成CSV读取模块的开发

[Output]
请基于以上信息,提供准确、有据的回答。
=====
```

这个结构化的上下文包含了所有必要的信息：

- [Role & Policies]: 明确了 AI 的角色和回答要求
- [Task]: 清晰地表达了用户的问题
- [Evidence]: 从 RAG 系统检索的相关知识
- [Context]: 对话历史和相关记忆，提供了充分的背景信息
- [Output]: 指导 LLM 如何组织回答

(3) 与 Agent 集成

最后，让我们展示如何将 ContextBuilder 集成到 Agent 中：

```
from hello_agents import SimpleAgent, HelloAgentsLLM, ToolRegistry
from hello_agents.context import ContextBuilder, ContextConfig
from hello_agents.tools import MemoryTool, RAGTool

class ContextAwareAgent(SimpleAgent):
    """具有上下文感知能力的 Agent"""

    def __init__(self, name: str, llm: HelloAgentsLLM, **kwargs):
        super().__init__(name=name, llm=llm, system_prompt=kwargs.get("system_prompt", ""))
        # 初始化上下文构建器
        self.memory_tool = MemoryTool(user_id=kwargs.get("user_id", "default"))
        self.rag_tool = RAGTool(knowledge_base_path=kwargs.get("knowledge_base_path", "./kb"))

        self.context_builder = ContextBuilder(
            memory_tool=self.memory_tool,
            rag_tool=self.rag_tool,
            config=ContextConfig(max_tokens=4000)
        )
        self.conversation_history = []

    def run(self, user_input: str) -> str:
        """运行 Agent, 自动构建优化的上下文"""

        # 1. 使用 ContextBuilder 构建优化的上下文
        optimized_context = self.context_builder.build(
            user_query=user_input,
            conversation_history=self.conversation_history,
            system_instructions=self.system_prompt
        )

        # 2. 使用优化后的上下文调用 LLM
        messages = [
            {"role": "system", "content": optimized_context},
            {"role": "user", "content": user_input}
        ]
        response = self.llm.invoke(messages)

        # 3. 更新对话历史
```

```

from hello_agents.core.message import Message
from datetime import datetime

self.conversation_history.append(
    Message(content=user_input, role="user", timestamp=datetime.now())
)
self.conversation_history.append(
    Message(content=response, role="assistant", timestamp=datetime.now())
)

# 4. 将重要交互记录到记忆系统
self.memory_tool.run({
    "action": "add",
    "content": f"Q: {user_input}\nA: {response[:200]}...", # 摘要
    "memory_type": "episodic",
    "importance": 0.6
})

return response

# 使用示例
agent = ContextAwareAgent(
    name="数据分析顾问",
    llm=HelloAgentsLLM(),
    system_prompt="你是一位资深的Python数据工程顾问。",
    user_id="user123",
    knowledge_base_path="./data_science_kb"
)

response = agent.run("如何优化Pandas的内存占用?")
print(response)

```

通过这种方式，ContextBuilder 成为了 Agent 的“上下文管理大脑”，自动处理信息的收集、筛选和组织，让 Agent 始终能够在最优的上下文下进行推理和生成。

9.3.5 最佳实践与优化建议

在实际应用 ContextBuilder 时，以下几点最佳实践值得注意：

1. 动态调整 **token** 预算：根据任务复杂度动态调整 `max_tokens`，简单任务使用较小预算，复杂任务增加预算。
2. 相关性计算优化：在生产环境中，将简单的关键词重叠替换为向量相似度计算，提升检索质量。
3. 缓存机制：对于不变的系统指令和知识库内容，可以实现缓存机制，避免重复计算。
4. 监控与日志：记录每次上下文构建的统计信息(选中信息数量、token 使用率等)，便于后续优化。
5. A/B 测试：对于关键参数(如相关性权重、新近性权重)，通过 A/B 测试找到最优配置。

9.4 NoteTool：结构化笔记

NoteTool 是为“长时程任务”提供的结构化外部记忆组件。它以 Markdown 文件作为载体，头部使用 YAML 前置元数据记录关键信息，正文用于记录状态、结论、阻塞与行动项等内容。这种设计结合了人类可读性、版本控制友好性和易于回注上下文的特性，是构建长时程智能体的重要工具。

9.4.1 设计理念与应用场景

在深入实现细节之前，让我们首先理解 NoteTool 的设计理念和典型应用场景。

(1) 为什么需要 NoteTool？

在第八章中，我们介绍了 MemoryTool，它提供了强大的记忆管理能力。然而，MemoryTool 主要关注对话式记忆——短期工作记忆、情景记忆和语义记忆。对于需要长期追踪、结构化管理的项目式任务，我们需要一种更轻量、更人类友好的记录方式。

NoteTool 填补了这个gap，它提供了：

- 结构化记录：使用 Markdown + YAML 格式，既适合机器解析，也方便人类阅读和编辑
- 版本友好：纯文本格式，天然支持 Git 等版本控制系统
- 低开销：无需复杂的数据库操作，适合轻量级的状态追踪
- 灵活分类：通过 `type` 和 `tags` 灵活组织笔记，支持多维度检索

(2) 典型应用场景

NoteTool 特别适合以下场景：

场景1：长期项目追踪

想象一个智能体正在协助完成一个大型代码库的重构任务，这可能需要几天甚至几周。NoteTool 可以记录：

- `task_state`: 当前阶段的任务状态和进度

- **conclusion:** 每个阶段结束后关键结论
- **blocker:** 遇到的问题和阻塞点
- **action:** 下一步的行动计划

```
# 记录任务状态
notes.run({
    "action": "create",
    "title": "重构项目 - 第一阶段",
    "content": "已完成数据模型层的重构，测试覆盖率达到85%。下一步将重构业务逻辑层。",
    "note_type": "task_state",
    "tags": ["refactoring", "phase1"]
})

# 记录阻塞点
notes.run({
    "action": "create",
    "title": "依赖冲突问题",
    "content": "发现某些第三方库版本不兼容，需要解决。影响范围：业务逻辑层的3个模块。",
    "note_type": "blocker",
    "tags": ["dependency", "urgent"]
})
```

场景2：研究任务管理

一个智能研究助手在进行文献综述时，可以使用 NoteTool 记录：

- 每篇论文的核心观点(conclusion)
- 待深入调研的主题(action)
- 重要的参考文献(reference)

场景3：与 ContextBuilder 配合

在每轮对话前，Agent 可以通过 search 或 list 操作检索相关笔记，并将其注入到上下文中：

```
# 在 Agent 的 run 方法中
def run(self, user_input: str) -> str:
    # 1. 检索相关笔记
    relevant_notes = self.note_tool.run({
        "action": "search",
        "query": user_input,
        "limit": 3
    })

    # 2. 将笔记内容转换为 ContextPacket
    note_packets = []
    for note in relevant_notes:
        note_packets.append(ContextPacket(
            content=note['content'],
            timestamp=note['updated_at'],
            token_count=self._count_tokens(note['content']),
            relevance_score=0.7,
            metadata={"type": "note", "note_type": note['type']}
        ))

    # 3. 构建上下文时传入笔记
    context = self.context_builder.build(
        user_query=user_input,
        custom_packets=note_packets,
        ...
    )
```

9.4.2 存储格式详解

NoteTool 采用了 Markdown + YAML 的混合格式，这种设计兼顾了结构化和可读性。

(1) 笔记文件格式

每个笔记都是一个独立的 .md 文件，格式如下：

```
---
id: note_20250119_153000_0
title: 项目进展 - 第一阶段
type: task_state
tags: [refactoring, phase1, backend]
created_at: 2025-01-19T15:30:00
updated_at: 2025-01-19T15:30:00
---
```

项目进展 - 第一阶段

完成情况

已完成数据模型层的重构，主要改动包括：

1. 统一了实体类的命名规范
2. 引入了类型提示，提升代码可维护性

3. 优化了数据库查询性能

测试覆盖

- 单元测试覆盖率: 85%
- 集成测试覆盖率: 70%

下一步计划

1. 重构业务逻辑层
2. 解决依赖冲突问题
3. 提升集成测试覆盖率至85%

这种格式的优势:

- **YAML** 元数据: 机器可解析, 支持精确的字段提取和检索
- **Markdown** 正文: 人类可读, 支持丰富的格式化(标题、列表、代码块等)
- 文件名即 **ID**: 简化管理, 每个笔记的文件名就是其唯一标识

(2) 索引文件

NoteTool 维护一个 `notes_index.json` 文件, 用于快速检索和管理笔记:

```
{
  "note_20250119_153000_0": {
    "id": "note_20250119_153000_0",
    "title": "项目进展 - 第一阶段",
    "type": "task_state",
    "tags": ["refactoring", "phase1", "backend"],
    "created_at": "2025-01-19T15:30:00",
    "updated_at": "2025-01-19T15:30:00",
    "file_path": "./notes/note_20250119_153000_0.md"
  }
}
```

这个索引文件的作用:

- 快速检索: 无需打开每个文件, 直接从索引中查找
- 元数据管理: 集中管理所有笔记的元数据
- 完整性校验: 可以检测文件缺失或损坏

9.4.3 核心操作详解

NoteTool 提供了七个核心操作, 覆盖了笔记的完整生命周期管理。

(1) create: 创建笔记

```
def _create_note(
    self,
    title: str,
    content: str,
    note_type: str = "general",
    tags: Optional[List[str]] = None
) -> str:
    """创建笔记

    Args:
        title: 笔记标题
        content: 笔记内容(Markdown格式)
        note_type: 笔记类型(task_state/conclusion/blocker/action/reference/general)
        tags: 标签列表

    Returns:
        str: 笔记ID
    """
    from datetime import datetime

    # 1. 生成唯一ID
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    note_id = f"note_{timestamp}_{len(self.index)}"

    # 2. 构建元数据
    metadata = {
        "id": note_id,
        "title": title,
        "type": note_type,
        "tags": tags or [],
        "created_at": datetime.now().isoformat(),
        "updated_at": datetime.now().isoformat()
    }

    # 3. 构建完整的 Markdown 文件内容
    md_content = self._build_markdown(metadata, content)

    # 4. 保存到文件
    file_path = os.path.join(self.workspace, f"{note_id}.md")
```

2/9/26, 1:56 PM

```
with open(file_path, 'w', encoding='utf-8') as f:
    f.write(md_content)

# 5. 更新索引
metadata['file_path'] = file_path
self.index[note_id] = metadata
self._save_index()

return note_id

def _build_markdown(self, metadata: Dict, content: str) -> str:
    """构建 Markdown 文件内容(YAML + 正文)"""
    import yaml

    # YAML 前置元数据
    yaml_header = yaml.dump(metadata, allow_unicode=True, sort_keys=False)

    # 组合格式
    return f"---\n{yaml_header}---\n\n{content}"
```

使用示例:

```
from hello_agents.tools import NoteTool

notes = NoteTool(workspace="./project_notes")

note_id = notes.run({
    "action": "create",
    "title": "重构项目 - 第一阶段",
    "content": "## 完成情况
已完成数据模型层的重构，测试覆盖率达到85%。

## 下一步
重构业务逻辑层",
    "note_type": "task_state",
    "tags": ["refactoring", "phase1"]
})

print(f"✓ 笔记创建成功, ID: {note_id}")
```

(2) read: 读取笔记

```
def _read_note(self, note_id: str) -> Dict:
    """读取笔记内容

Args:
    note_id: 笔记ID

Returns:
    Dict: 包含元数据和内容的字典
    """
    if note_id not in self.index:
        raise ValueError(f"笔记不存在: {note_id}")

    file_path = self.index[note_id]["file_path"]

    # 读取文件
    with open(file_path, 'r', encoding='utf-8') as f:
        raw_content = f.read()

    # 解析 YAML 元数据和 Markdown 正文
    metadata, content = self._parse_markdown(raw_content)

    return {
        "metadata": metadata,
        "content": content
    }

def _parse_markdown(self, raw_content: str) -> Tuple[Dict, str]:
    """解析 Markdown 文件(分离 YAML 和正文)"""
    import yaml

    # 查找 YAML 分隔符
    parts = raw_content.split('---\n', 2)

    if len(parts) >= 3:
        # 有 YAML 前置元数据
        yaml_str = parts[1]
        content = parts[2].strip()
        metadata = yaml.safe_load(yaml_str)
    else:
        # 无元数据,全部作为正文
        metadata = {}
        content = raw_content.strip()

    return metadata, content
```

(3) update: 更新笔记

```
def _update_note(
    self,
    note_id: str,
```

2/9/26, 1:56 PM

```
title: Optional[str] = None,
content: Optional[str] = None,
note_type: Optional[str] = None,
tags: Optional[List[str]] = None
) -> str:
    """更新笔记

Args:
    note_id: 笔记ID
    title: 新标题(可选)
    content: 新内容(可选)
    note_type: 新类型(可选)
    tags: 新标签(可选)

Returns:
    str: 操作结果消息
"""
if note_id not in self.index:
    raise ValueError(f"笔记不存在: {note_id}")

# 1. 读取现有笔记
note = self._read_note(note_id)
metadata = note["metadata"]
old_content = note["content"]

# 2. 更新字段
if title:
    metadata["title"] = title
if note_type:
    metadata["type"] = note_type
if tags is not None:
    metadata["tags"] = tags
if content is not None:
    old_content = content

# 更新时间戳
from datetime import datetime
metadata["updated_at"] = datetime.now().isoformat()

# 3. 重新构建并保存
md_content = self._build_markdown(metadata, old_content)
file_path = metadata['file_path']

with open(file_path, 'w', encoding='utf-8') as f:
    f.write(md_content)

# 4. 更新索引
self.index[note_id] = metadata
self._save_index()

return f"✅ 笔记已更新: {metadata['title']}
```

(4) search: 搜索笔记

```
def _search_notes(
    self,
    query: str,
    limit: int = 10,
    note_type: Optional[str] = None,
    tags: Optional[List[str]] = None
) -> List[Dict]:
    """搜索笔记

Args:
    query: 搜索关键词
    limit: 返回数量限制
    note_type: 按类型过滤(可选)
    tags: 按标签过滤(可选)

Returns:
    List[Dict]: 匹配的笔记列表
"""
results = []
query_lower = query.lower()

for note_id, metadata in self.index.items():
    # 类型过滤
    if note_type and metadata.get("type") != note_type:
        continue

    # 标签过滤
    if tags:
        note_tags = set(metadata.get("tags", []))
        if not note_tags.intersection(tags):
            continue

    # 读取笔记内容
    try:
        note = self._read_note(note_id)
        content = note["content"]
        title = metadata.get("title", "")
    except Exception as e:
        print(f"Error reading note {note_id}: {e}")
        continue

    result = {
        "note_id": note_id,
        "title": title,
        "content": content,
        "type": metadata.get("type"),
        "tags": metadata.get("tags", []),
        "updated_at": metadata.get("updated_at")
    }
    results.append(result)

    if len(results) == limit:
        break

return results
```

```

# 在标题和内容中搜索
if query_lower in title.lower() or query_lower in content.lower():
    results.append({
        "note_id": note_id,
        "title": title,
        "type": metadata.get("type"),
        "tags": metadata.get("tags", []),
        "content": content,
        "updated_at": metadata.get("updated_at")
    })
except Exception as e:
    print(f"[WARNING] 读取笔记 {note_id} 失败: {e}")
    continue

# 按更新时间排序
results.sort(key=lambda x: x["updated_at"], reverse=True)

return results[:limit]

```

(5) list: 列出笔记

```

def _list_notes(
    self,
    note_type: Optional[str] = None,
    tags: Optional[List[str]] = None,
    limit: int = 20
) -> List[Dict]:
    """列出笔记(按更新时间倒序)

Args:
    note_type: 按类型过滤(可选)
    tags: 按标签过滤(可选)
    limit: 返回数量限制

Returns:
    List[Dict]: 笔记元数据列表
"""
    results = []

    for note_id, metadata in self.index.items():
        # 类型过滤
        if note_type and metadata.get("type") != note_type:
            continue

        # 标签过滤
        if tags:
            note_tags = set(metadata.get("tags", []))
            if not note_tags.intersection(tags):
                continue

        results.append(metadata)

    # 按更新时间排序
    results.sort(key=lambda x: x.get("updated_at", ""), reverse=True)

    return results[:limit]

```

(6) summary: 笔记摘要

```

def _summary(self) -> Dict[str, Any]:
    """生成笔记摘要统计

Returns:
    Dict: 统计信息
"""
    total_count = len(self.index)

    # 按类型统计
    type_counts = {}
    for metadata in self.index.values():
        note_type = metadata.get("type", "general")
        type_counts[note_type] = type_counts.get(note_type, 0) + 1

    # 最近更新的笔记
    recent_notes = sorted(
        self.index.values(),
        key=lambda x: x.get("updated_at", ""),
        reverse=True
    )[:5]

    return {
        "total_notes": total_count,
        "type_distribution": type_counts,
        "recent_notes": [
            {
                "id": note["id"],
                "title": note.get("title", ""),
                "type": note.get("type"),
                "updated_at": note.get("updated_at")
            }
            for note in recent_notes
        ]
    }

```

(7) delete: 删除笔记

```
def _delete_note(self, note_id: str) -> str:
    """删除笔记

    Args:
        note_id: 笔记ID

    Returns:
        str: 操作结果消息
    """
    if note_id not in self.index:
        raise ValueError(f"笔记不存在: {note_id}")

    # 1. 删除文件
    file_path = self.index[note_id]["file_path"]
    if os.path.exists(file_path):
        os.remove(file_path)

    # 2. 从索引中移除
    title = self.index[note_id].get("title", note_id)
    del self.index[note_id]
    self._save_index()

    return f"✅ 笔记已删除: {title}"
```

9.4.4 与 ContextBuilder 的深度集成

NoteTool 的真正威力在于与 ContextBuilder 的配合使用。让我们通过一个完整的案例来展示这种集成。

(1) 场景设定

假设我们正在构建一个长期项目助手，它需要：

1. 记录项目的阶段性进展
2. 追踪待解决的问题
3. 在每次对话时，自动回顾相关笔记
4. 基于历史笔记提供连贯的建议

(2) 实现示例

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.context import ContextBuilder, ContextConfig, ContextPacket
from hello_agents.tools import MemoryTool, RAGTool, NoteTool
from datetime import datetime

class ProjectAssistant(SimpleAgent):
    """长期项目助手, 集成 NoteTool 和 ContextBuilder"""

    def __init__(self, name: str, project_name: str, **kwargs):
        super().__init__(name=name, llm=HelloAgentsLLM(), **kwargs)
        self.project_name = project_name

        # 初始化工具
        self.memory_tool = MemoryTool(user_id=project_name)
        self.rag_tool = RAGTool(knowledge_base_path=f"./{project_name}_kb")
        self.note_tool = NoteTool(workspace=f"./{project_name}_notes")

        # 初始化上下文构建器
        self.context_builder = ContextBuilder(
            memory_tool=self.memory_tool,
            rag_tool=self.rag_tool,
            config=ContextConfig(max_tokens=4000)
        )

        self.conversation_history = []

    def run(self, user_input: str, note_as_action: bool = False) -> str:
        """运行助手, 自动集成笔记"""

        # 1. 从 NoteTool 检索相关笔记
        relevant_notes = self._retrieve_relevant_notes(user_input)

        # 2. 将笔记转换为 ContextPacket
        note_packets = self._notes_to_packets(relevant_notes)

        # 3. 构建优化的上下文
        context = self.context_builder.build(
            user_query=user_input,
            conversation_history=self.conversation_history,
            system_instructions=self._build_system_instructions(),
            custom_packets=note_packets
        )

        # 4. 调用 LLM
        response = self.llm.invoke(context)
```

```

# 5. 如果需要, 将交互记录为笔记
if note_as_action:
    self._save_as_note(user_input, response)

# 6. 更新对话历史
self._update_history(user_input, response)

return response

def _retrieve_relevant_notes(self, query: str, limit: int = 3) -> List[Dict]:
    """检索相关笔记"""
    try:
        # 优先检索 blocker 和 action 类型的笔记
        blockers = self.note_tool.run({
            "action": "list",
            "note_type": "blocker",
            "limit": 2
        })

        # 通用搜索
        search_results = self.note_tool.run({
            "action": "search",
            "query": query,
            "limit": limit
        })

        # 合并去重
        all_notes = {note['note_id']: note for note in blockers + search_results}
        return list(all_notes.values())[:limit]
    except Exception as e:
        print(f"[WARNING] 笔记检索失败: {e}")
        return []

```

```

def _notes_to_packets(self, notes: List[Dict]) -> List[ContextPacket]:
    """将笔记转换为上下文包"""
    packets = []

    for note in notes:
        content = f"[笔记:{note['title']}]\n{note['content']}"

        packets.append(ContextPacket(
            content=content,
            timestamp=datetime.fromisoformat(note['updated_at']),
            token_count=len(content) // 4, # 简单估算
            relevance_score=0.75, # 笔记具有较高相关性
            metadata={
                "type": "note",
                "note_type": note['type'],
                "note_id": note['note_id']
            }
        ))
    return packets

```

```

def _save_as_note(self, user_input: str, response: str):
    """将交互保存为笔记"""
    try:
        # 判断应该保存为什么类型的笔记
        if "问题" in user_input or "阻塞" in user_input:
            note_type = "blocker"
        elif "计划" in user_input or "下一步" in user_input:
            note_type = "action"
        else:
            note_type = "conclusion"

        self.note_tool.run({
            "action": "create",
            "title": f"{user_input[:30]}...",
            "content": f"## 问题\n{user_input}\n## 分析\n{response}",
            "note_type": note_type,
            "tags": [self.project_name, "auto_generated"]
        })
    except Exception as e:
        print(f"[WARNING] 保存笔记失败: {e}")

```

```

def _build_system_instructions(self) -> str:
    """构建系统指令"""
    return f"你是 {self.project_name} 项目的长期助手。

```

你的职责:

1. 基于历史笔记提供连贯的建议
2. 追踪项目进展和待解决问题
3. 在回答时引用相关的历史笔记
4. 提供具体、可操作的下一步建议

注意:

- 优先关注标记为 blocker 的问题
- 在建议中说明依据来源(笔记、记忆或知识库)
- 保持对项目整体进度的认识"""

```

def _update_history(self, user_input: str, response: str):
    """更新对话历史"""
    from hello_agents.core.message import Message

    self.conversation_history.append(
        Message(content=user_input, role="user", timestamp=datetime.now())
    )
    self.conversation_history.append(
        Message(content=response, role="assistant", timestamp=datetime.now())
    )

    # 限制历史长度
    if len(self.conversation_history) > 10:
        self.conversation_history = self.conversation_history[-10:]

# 使用示例
assistant = ProjectAssistant(
    name="项目助手",
    project_name="data_pipeline_refactoring"
)

# 第一次交互:记录项目状态
response = assistant.run(
    "我们已经完成了数据模型层的重构,测试覆盖率达到85%。下一步计划重构业务逻辑层。",
    note_as_action=True
)

# 第二次交互:提出问题
response = assistant.run(
    "在重构业务逻辑层时,我遇到了依赖版本冲突的问题,该如何解决?"
)

# 查看笔记摘要
summary = assistant.note_tool.run({"action": "summary"})
print(summary)

```

(3) 运行效果展示

[ContextBuilder] 汇集了 8 个候选信息包
[ContextBuilder] 选择了 7 个信息包, 共 3500 tokens

助手回答:

我注意到您之前记录的笔记中提到了这个问题。根据笔记 [重构项目 - 第一阶段], 您当前的测试覆盖率已经达到85%, 这是一个很好的基础。

关于依赖版本冲突的问题, 我建议:

1. **使用虚拟环境隔离**: 为业务逻辑层创建独立的虚拟环境, 避免与其他模块的依赖冲突
2. **锁定版本**: 在 requirements.txt 中明确指定所有依赖的精确版本
3. **使用 pipdeptree**: 分析依赖树, 找出冲突的根源

这个问题我会标记为 blocker, 建议优先解决。

[依据来源: 笔记 note_20250119_153000_0, 项目知识库]

笔记摘要:

```
{
  "total_notes": 2,
  "type_distribution": {
    "action": 1,
    "blocker": 1
  },
  "recent_notes": [
    {
      "id": "note_20250119_154500_1",
      "title": "在重构业务逻辑层时,我遇到了依赖版本冲突的问题....",
      "type": "blocker",
      "updated_at": "2025-01-19T15:45:00"
    },
    {
      "id": "note_20250119_153000_0",
      "title": "我们已经完成了数据模型层的重构...",
      "type": "action",
      "updated_at": "2025-01-19T15:30:00"
    }
  ]
}
```

9.4.5 最佳实践

在实际使用 NoteTool 时, 以下最佳实践能帮助您构建更强大的长时程智能体:

1. 合理的笔记分类:
 - task_state: 记录阶段性进展和状态
 - conclusion: 记录重要的结论和发现
 - blocker: 记录阻塞问题, 优先级最高

- action: 记录下一步行动计划
- reference: 记录重要的参考资料
- 2. 定期清理和归档:
 - 对于已解决的 blocker, 更新为 conclusion
 - 对于过时的 action, 及时删除或更新
 - 使用 tags 进行版本管理, 如 ["v1.0", "completed"]
- 3. 与 ContextBuilder 的配合:
 - 在每轮对话前检索相关笔记
 - 根据笔记类型设置不同的相关性分数(blocker > action > conclusion)
 - 限制笔记数量, 避免上下文过载
- 4. 人机协作:
 - 笔记是人类可读的 Markdown 格式, 支持手动编辑
 - 使用 Git 进行版本控制, 追踪笔记的演化
 - 在关键阶段, 人工审核 Agent 生成的笔记
- 5. 自动化工作流:
 - 定期生成笔记摘要报告
 - 基于笔记自动生成项目进度文档
 - 将笔记内容同步到其他系统(如 Notion、Confluence)

9.5 TerminalTool: 即时文件系统访问

在前面的章节中, 我们介绍了 MemoryTool 和 RAGTool, 它们分别提供了对话记忆和知识检索能力。然而, 在许多实际场景中, 智能体需要即时访问和探索文件系统——查看日志文件、分析代码库结构、检索配置文件等。这就是 TerminalTool 的用武之地。

TerminalTool 为智能体提供了安全的命令行执行能力, 支持常用的文件系统和文本处理命令, 同时通过多层安全机制确保系统安全。这种设计实现了 9.2.2 节提到的“即时(Just-in-time, JIT)上下文”理念——智能体不需要预先加载所有文件, 而是按需探索和检索。

9.5.1 设计理念与安全机制

(1) 为什么需要 TerminalTool?

在构建长程智能体时, 我们经常遇到以下场景:

场景1: 代码库探索

一个开发助手需要帮助用户理解一个大型代码库的结构:

```
# 传统方式: 预先索引所有文件(成本高、可能过时)
rag_tool.add_document("./project/**/*.py") # 耗时、占用大量存储

# TerminalTool 方式: 即时探索
terminal.run({"command": "find . -name '*.py' -type f"}) # 快速、实时
terminal.run({"command": "grep -r 'class UserService' ."}) # 精确定位
terminal.run({"command": "head -n 50 src/services/user.py"}) # 按需查看
```

场景2: 日志文件分析

一个运维助手需要分析应用日志:

```
# 检查日志文件大小
terminal.run({"command": "ls -lh /var/log/app.log"})

# 查看最新的错误日志
terminal.run({"command": "tail -n 100 /var/log/app.log | grep ERROR"})

# 统计错误类型分布
terminal.run({"command": "grep ERROR /var/log/app.log | cut -d':' -f3 | sort | uniq -c"})
```

场景3: 数据文件预览

一个数据分析助手需要快速了解数据文件的结构:

```
# 查看 CSV 文件的前几行
terminal.run({"command": "head -n 5 data/sales.csv"})

# 统计行数
terminal.run({"command": "wc -l data/*.csv"})

# 查看列名
terminal.run({"command": "head -n 1 data/sales.csv | tr ',' '\n'"})
```

这些场景的共同特点是: 需要实时、轻量级的文件系统访问, 而不是预先索引和向量化。TerminalTool 正是为这种“探索式”工作流设计的。

(2) 安全机制详解

允许智能体执行命令是一个强大但危险的能力。TerminalTool 通过多层安全机制确保系统安全:

第一层：命令白名单

只允许安全的只读命令，完全禁止任何可能修改系统的操作：

```
ALLOWED_COMMANDS = {
    # 文件列表与信息
    'ls', 'dir', 'tree',
    # 文件内容查看
    'cat', 'head', 'tail', 'less', 'more',
    # 文件搜索
    'find', 'grep', 'egrep', 'fgrep',
    # 文本处理
    'wc', 'sort', 'uniq', 'cut', 'awk', 'sed',
    # 目录操作
    'pwd', 'cd',
    # 文件信息
    'file', 'stat', 'du', 'df',
    # 其他
    'echo', 'which', 'whereis',
}
```

如果智能体尝试执行白名单外的命令，会立即被拒绝：

```
terminal.run({"command": "rm -rf /"})
# ❌ 不允许的命令: rm
# 允许的命令: cat, cd, cut, dir, du, ...
```

第二层：工作目录限制(沙箱)

TerminalTool 只能访问指定的工作目录及其子目录，无法访问系统其他部分：

```
# 初始化时指定工作目录
terminal = TerminalTool(workspace="./project")

# 允许: 访问工作目录内的文件
terminal.run({"command": "cat ./src/main.py"}) # ✅

# 禁止: 访问工作目录外的文件
terminal.run({"command": "cat /etc/passwd"}) # ❌ 不允许访问工作目录外的路径

# 禁止: 通过 .. 逃逸
terminal.run({"command": "cd ../../etc"}) # ❌ 不允许访问工作目录外的路径
```

这种沙箱机制确保了即使智能体的行为出现异常，也无法影响系统其他部分。

第三层：超时控制

每个命令都有执行时间限制，防止无限循环或资源耗尽：

```
terminal = TerminalTool(
    workspace="./project",
    timeout=30 # 30秒超时
)

# 如果命令执行超过30秒
terminal.run({"command": "find / -name '*.log'"})
# ❌ 命令执行超时 (超过 30 秒)
```

第四层：输出大小限制

限制命令输出的大小，防止内存溢出：

```
terminal = TerminalTool(
    workspace="./project",
    max_output_size=10 * 1024 * 1024 # 10MB
)

# 如果输出超过10MB
terminal.run({"command": "cat huge_file.log"})
# ... (前10MB的内容) ...
# ⚠️ 输出被截断 (超过 10485760 字节)
```

通过这四层安全机制，TerminalTool 在提供强大能力的同时，最大程度地保证了系统安全。

9.5.2 核心功能详解

TerminalTool 的实现聚焦于两个核心功能：命令执行和目录导航。

(1) 命令执行

核心的 `_execute_command` 方法负责实际执行命令：

2/9/26, 1:56 PM

```
def _execute_command(self, command: str) -> str:
    """执行命令"""
    try:
        # 在当前目录下执行命令
        result = subprocess.run(
            command,
            shell=True,
            cwd=str(self.current_dir), # 在当前工作目录执行
            capture_output=True,
            text=True,
            timeout=self.timeout,
            env=os.environ.copy()
        )

        # 合并标准输出和标准错误
        output = result.stdout
        if result.stderr:
            output += f"\n{result.stderr}\n"

        # 检查输出大小
        if len(output) > self.max_output_size:
            output = output[:self.max_output_size]
            output += f"\n\n⚠️ 输出被截断 (超过 {self.max_output_size} 字节) "

        # 添加返回码信息
        if result.returncode != 0:
            output = f"⚠️ 命令返回码: {result.returncode}\n\n{output}"

        return output if output else "✅ 命令执行成功 (无输出)"

    except subprocess.TimeoutExpired:
        return f"❌ 命令执行超时 (超过 {self.timeout} 秒) "
    except Exception as e:
        return f"❌ 命令执行失败: {e}"
```

这个实现的关键点：

- 当前目录感知：使用 cwd 参数在正确的目录下执行命令
- 错误处理：捕获并合并标准错误，提供完整的诊断信息
- 返回码检查：非零返回码会被标记为警告
- 容错设计：超时和异常都会被妥善处理，不会导致智能体崩溃

(2) 目录导航

cd 命令的特殊处理支持智能体在文件系统中导航：

```
def _handle_cd(self, parts: List[str]) -> str:
    """处理 cd 命令"""
    if not self.allow_cd:
        return "❌ cd 命令已禁用"

    if len(parts) < 2:
        # cd 无参数，返回当前目录
        return f"当前目录: {self.current_dir}"

    target_dir = parts[1]

    # 处理相对路径
    if target_dir == "..":
        new_dir = self.current_dir.parent
    elif target_dir == ".":
        new_dir = self.current_dir
    elif target_dir == "~":
        new_dir = self.workspace
    else:
        new_dir = (self.current_dir / target_dir).resolve()

    # 检查是否在工作目录内
    try:
        new_dir.relative_to(self.workspace)
    except ValueError:
        return f"❌ 不允许访问工作目录外的路径: {new_dir}"

    # 检查目录是否存在
    if not new_dir.exists():
        return f"❌ 目录不存在: {new_dir}"

    if not new_dir.is_dir():
        return f"❌ 不是目录: {new_dir}"

    # 更新当前目录
    self.current_dir = new_dir
    return f"✅ 切换到目录: {self.current_dir}"
```

这种设计支持智能体进行多步骤的文件系统探索：

```
# 第一步:查看项目结构
terminal.run({"command": "ls -la"})

# 第二步:进入源代码目录
terminal.run({"command": "cd src"})

# 第三步:查找特定文件
terminal.run({"command": "find . -name '*service*.py'""})

# 第四步:查看文件内容
terminal.run({"command": "cat user_service.py"})
```

9.5.3 典型使用模式

TerminalTool 支持多种常见的文件系统操作模式。

(1) 探索式导航

智能体可以像人类开发者一样逐步探索代码库:

```
from hello_agents.tools import TerminalTool
terminal = TerminalTool(workspace=".my_project")

# 第一步:查看项目根目录
print(terminal.run({"command": "ls -la"")))
"""
total 24
drwxr-xr-x 6 user staff 192 Jan 19 16:00 .
drwxr-xr-x 5 user staff 160 Jan 19 15:30 ..
-rw-r--r-- 1 user staff 1234 Jan 19 15:30 README.md
drwxr-xr-x 4 user staff 128 Jan 19 15:30 src
drwxr-xr-x 3 user staff 96 Jan 19 15:30 tests
-rw-r--r-- 1 user staff 456 Jan 19 15:30 requirements.txt
"""

# 第二步:查看源代码目录结构
terminal.run({"command": "cd src"})
print(terminal.run({"command": "tree"}))

# 第三步:搜索特定模式
print(terminal.run({"command": "grep -r 'def process' .}))
```

(2) 数据文件分析

快速了解数据文件的结构和内容:

```
terminal = TerminalTool(workspace=".data")

# 查看 CSV 文件的前几行
print(terminal.run({"command": "head -n 5 sales_2024.csv"")))
"""
date,product,quantity,revenue
2024-01-01,Widget A,150,4500.00
2024-01-01,Widget B,200,8000.00
2024-01-02,Widget A,180,5400.00
2024-01-02,Widget C,120,3600.00
"""

# 统计总行数
print(terminal.run({"command": "wc -l *.csv"}))
"""
10234 sales_2024.csv
8567 sales_2023.csv
18801 total
"""

# 提取和统计产品类别
print(terminal.run({"command": "tail -n +2 sales_2024.csv | cut -d',' -f2 | sort | uniq -c"}))
"""
3456 Widget A
4123 Widget B
2655 Widget C
"""

#
```

(3) 日志文件分析

实时分析应用日志，快速定位问题:

```
terminal = TerminalTool(workspace="/var/log")

# 查看最新的错误日志
print(terminal.run({"command": "tail -n 50 app.log | grep ERROR"}))

# 统计错误类型分布
print(terminal.run({"command": "grep ERROR app.log | awk '{print $4}' | sort | uniq -c | sort -rn"}))
"""
245 DatabaseConnectionError
123 TimeoutException
67 ValidationError
```

2/9/26, 1:56 PM

34 AuthenticationError

.....

```
# 查找特定时间段的日志
print(terminal.run({"command": "grep '2024-01-19 15:' app.log | tail -n 20"}))
```

(4) 代码库分析

辅助代码审查和理解:

```
terminal = TerminalTool(workspace='./codebase')

# 统计代码行数
print(terminal.run({"command": "find . -name '*.py' -exec wc -l {} + | tail -n 1"}))

# 查找所有 TODO 注释
print(terminal.run({"command": "grep -rn 'TODO' --include='*.py'"}))

# 查找特定函数的定义
print(terminal.run({"command": "grep -rn 'def process_data' --include='*.py'"}))

# 查看函数实现
print(terminal.run({"command": "sed -n '/def process_data/,/^def /p' src/processor.py | head -n -1"}))
```

9.5.4 与其他工具的协同

TerminalTool 的真正威力在于与 MemoryTool、NoteTool 和 ContextBuilder 的协同使用。

(1) 与 MemoryTool 协同

TerminalTool 发现的信息可以存储到记忆系统中:

```
# 使用 TerminalTool 发现项目结构
structure = terminal.run({"command": "tree -L 2 src"})

# 存储到语义记忆
memory_tool.run({
    "action": "add",
    "content": f"项目结构:\n{structure}",
    "memory_type": "semantic",
    "importance": 0.8,
    "metadata": {"type": "project_structure"}
})
```

(2) 与 NoteTool 协同

重要的发现可以记录为结构化笔记:

```
# 发现一个性能瓶颈
log_analysis = terminal.run({"command": "grep 'slow query' app.log | tail -n 10"})

# 记录为 blocker 笔记
note_tool.run({
    "action": "create",
    "title": "数据库慢查询问题",
    "content": f"## 问题描述\n发现多个慢查询, 影响系统性能\n## 日志分析\n{n}{log_analysis}\n{n}## 下一步\n1. 分析慢查询SQL\n2. 添加索引\n3. 优化查{n}"),
    "note_type": "blocker",
    "tags": ["performance", "database"]
})
```

(3) 与 ContextBuilder 协同

TerminalTool 的输出可以作为上下文的一部分:

```
# 探索代码库
code_structure = terminal.run({"command": "ls -R src"})
recent_changes = terminal.run({"command": "git log --oneline -10"})

# 转换为 ContextPacket
from hello_agents.context import ContextPacket
from datetime import datetime

packets = [
    ContextPacket(
        content=f"代码库结构:\n{code_structure}",
        timestamp=datetime.now(),
        token_count=len(code_structure) // 4,
        relevance_score=0.7,
        metadata={"type": "code_structure", "source": "terminal"}),
    ContextPacket(
        content=f"最近提交:\n{recent_changes}",
        timestamp=datetime.now(),
        token_count=len(recent_changes) // 4,
        relevance_score=0.8,
        metadata={"type": "git_history", "source": "terminal"})
]
```

```
# 在构建上下文时包含这些信息
context = context_builder.build(
    user_query="如何重构用户服务模块?",
    custom_packets=packets
)
```

9.6 长程智能体实战：代码库维护助手

现在，让我们将 ContextBuilder、NoteTool 和 TerminalTool 整合起来，构建一个完整的长程智能体——代码库维护助手。这个助手能够：

1. 探索和理解代码库结构
2. 记录发现的问题和改进点
3. 追踪长期的重构任务
4. 在上下文窗口限制下保持连贯性

9.6.1 场景设定与需求分析

业务场景

假设我们正在维护一个中型 Python Web 应用，这个代码库包含约 50 个 Python 文件，使用 Flask 框架构建，涵盖数据模型、业务逻辑、API 接口等多个模块，同时存在一些技术债务需要逐步清理。在这样的场景下，我们需要一个智能助手来帮助我们探索代码库，理解项目结构、依赖关系和代码风格；识别代码中的问题，比如代码重复、复杂度过高、缺少测试等；追踪任务进度，记录待办事项、已完成工作和遇到的阻塞；并基于历史上下文提供连贯的重构建议。

挑战与解决方案

这个场景面临几个典型的长程任务挑战。首先是信息量超出上下文窗口的问题，整个代码库可能包含数万行代码，无法一次性放入上下文窗口，我们通过使用 TerminalTool 进行即时、按需的代码探索来解决这个问题，只在需要时查看具体文件。其次是跨会话的状态管理挑战，重构任务可能持续数天，需要跨多个会话保持进度，我们使用 NoteTool 记录阶段性进展、待办事项和关键决策来应对。最后是上下文质量与相关性的问题，每次对话需要回顾相关的历史信息，但不能被无关信息淹没，我们通过 ContextBuilder 智能筛选和组织上下文，确保高信号密度。

9.6.2 系统架构设计

我们的代码库维护助手采用三层架构，如图9.3所示：



图 9.3 代码库维护助手三层架构

9.6.3 核心实现

现在让我们实现这个系统的核心类：

```
from typing import Dict, Any, List, Optional
from datetime import datetime
import json

from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.context import ContextBuilder, ContextConfig, ContextPacket
from hello_agents.tools import MemoryTool, NoteTool, TerminalTool
from hello_agents.core.message import Message

class CodebaseMaintainer:
    """代码库维护助手 - 长程智能体示例

    整合 ContextBuilder + NoteTool + TerminalTool + MemoryTool
    实现跨会话的代码库维护任务管理
    """

    def __init__(
        self,
        project_name: str,
        codebase_path: str,
        llm: Optional[HelloAgentsLLM] = None
    ):
        self.project_name = project_name
        self.codebase_path = codebase_path
        self.session_id = f"session_{datetime.now().strftime('%Y%m%d_%H%M%S')}"

        # 初始化 LLM
        self.llm = llm or HelloAgentsLLM()

        # 初始化工具
        self.memory_tool = MemoryTool(user_id=project_name)
        self.note_tool = NoteTool(workspace=f"./{project_name}_notes")
        self.terminal_tool = TerminalTool(workspace=codebase_path, timeout=60)

        # 初始化上下文构建器
        self.context_builder = ContextBuilder()
```

```

        memory_tool=self.memory_tool,
        rag_tool=None, # 本案例不使用 RAG
        config=ContextConfig(
            max_tokens=4000,
            reserve_ratio=0.15,
            min_relevance=0.2,
            enable_compression=True
        )
    )

    # 对话历史
    self.conversation_history: List[Message] = []

    # 统计信息
    self.stats = {
        "session_start": datetime.now(),
        "commands_executed": 0,
        "notes_created": 0,
        "issues_found": 0
    }

    print(f"✅ 代码库维护助手已初始化: {project_name}")
    print(f"💻 工作目录: {codebase_path}")
    print(f">ID 会话ID: {self.session_id}")

def run(self, user_input: str, mode: str = "auto") -> str:
    """运行助手

    Args:
        user_input: 用户输入
        mode: 运行模式
            - "auto": 自动决策是否使用工具
            - "explore": 倾重代码探索
            - "analyze": 倾重问题分析
            - "plan": 倾重任务规划

    Returns:
        str: 助手的回答
    """
    print(f"\n{'='*80}")
    print(f"👤 用户: {user_input}")
    print(f"{'='*80}\n")

    # 第一步:根据模式执行预处理
    pre_context = self._preprocess_by_mode(user_input, mode)

    # 第二步:检索相关笔记
    relevant_notes = self._retrieve_relevant_notes(user_input)
    note_packets = self._notes_to_packets(relevant_notes)

    # 第三步:构建优化的上下文
    context = self.context_builder.build(
        user_query=user_input,
        conversation_history=self.conversation_history,
        system_instructions=self._build_system_instructions(mode),
        custom_packets=note_packets + pre_context
    )

    # 第四步:调用 LLM
    print("🌐 正在思考...")
    response = self.llm.invoke(context)

    # 第五步:后处理
    self._postprocess_response(user_input, response)

    # 第六步:更新对话历史
    self._update_history(user_input, response)

    print(f"\n🌐 助手: {response}\n")
    print(f"{'='*80}\n")

    return response

def _preprocess_by_mode(
    self,
    user_input: str,
    mode: str
) -> List[ContextPacket]:
    """根据模式执行预处理, 收集相关信息"""
    packets = []

    if mode == "explore" or mode == "auto":
        # 探索模式:自动查看项目结构
        print("🔍 探索代码库结构...")

        structure = self.terminal_tool.run({"command": "find . -type f -name '*.py' | head -n 20"})
        self.stats["commands_executed"] += 1

        packets.append(ContextPacket(
            content=f"[代码库结构]\n{structure}",
            timestamp=datetime.now(),
            type="structure"
        ))
    
```

```

        token_count=len(structure) // 4,
        relevance_score=0.6,
        metadata={"type": "code_structure", "source": "terminal"}
    )))
    if mode == "analyze":
        # 分析模式: 检查代码复杂度和问题
        print("📊 分析代码质量....")

        # 统计代码行数
        loc = self.terminal_tool.run({"command": "find . -name '*.py' -exec wc -l {} + | tail -n 1"})
        # 查找 TODO 和 FIXME
        todos = self.terminal_tool.run({"command": "grep -rn 'TODO\\|FIXME' --include='*.py' | head -n 10"})
        self.stats["commands_executed"] += 2
        packets.append(ContextPacket(
            content=f"[代码统计]\n{loc}\n[待办事项]\n{todos}",
            timestamp=datetime.now(),
            token_count=(len(loc) + len(todos)) // 4,
            relevance_score=0.7,
            metadata={"type": "code_analysis", "source": "terminal"}))
    if mode == "plan":
        # 规划模式: 加载最近的笔记
        print("📋 加载任务规划....")
        task_notes = self.note_tool.run({
            "action": "list",
            "note_type": "task_state",
            "limit": 3
        })
        if task_notes:
            content = "\n".join([f"- {note['title']} " for note in task_notes])
            packets.append(ContextPacket(
                content=f"[当前任务]\n{content}",
                timestamp=datetime.now(),
                token_count=len(content) // 4,
                relevance_score=0.8,
                metadata={"type": "task_plan", "source": "notes"}))
    return packets
def _retrieve_relevant_notes(self, query: str, limit: int = 3) -> List[Dict]:
    """检索相关笔记"""
    try:
        # 优先检索 blocker
        blockers = self.note_tool.run({
            "action": "list",
            "note_type": "blocker",
            "limit": 2
        })
        # 搜索相关笔记
        search_results = self.note_tool.run({
            "action": "search",
            "query": query,
            "limit": limit
        })
        # 合并去重
        all_notes = {note.get('note_id') or note.get('id'): note for note in (blockers or []) + (search_results or [])}
        return list(all_notes.values())[:limit]
    except Exception as e:
        print(f"[WARNING] 笔记检索失败: {e}")
        return []
def _notes_to_packets(self, notes: List[Dict]) -> List[ContextPacket]:
    """将笔记转换为上下文包"""
    packets = []
    for note in notes:
        # 根据笔记类型设置不同的相关性分数
        relevance_map = {
            "blocker": 0.9,
            "action": 0.8,
            "task_state": 0.75,
            "conclusion": 0.7
        }
        note_type = note.get('type', 'general')
        relevance = relevance_map.get(note_type, 0.6)
        content = f"[笔记:{note.get('title', 'Untitled')}]\n类型: {note_type}\n\n{note.get('content', '')}"
        packets.append(ContextPacket(
            content=content,
            timestamp=datetime.fromisoformat(note.get('updated_at', datetime.now().isoformat())),
            token_count=len(content) // 4,
            relevance=relevance,
            metadata={"type": "note", "source": "terminal"}))
    return packets

```

```

        relevance_score=relevance,
        metadata={
            "type": "note",
            "note_type": note_type,
            "note_id": note.get('note_id') or note.get('id')
        }
    )
)

return packets

def _build_system_instructions(self, mode: str) -> str:
    """构建系统指令"""
    base_instructions = f"""你是 {self.project_name} 项目的代码库维护助手。

你的核心能力:
1. 使用 TerminalTool 探索代码库(ls, cat, grep, find等)
2. 使用 NoteTool 记录发现和任务
3. 基于历史笔记提供连贯的建议

当前会话ID: {self.session_id}
"""

    mode_specific = {
        "explore": """
当前模式: 探索代码库

你应该:
- 主动使用 terminal 命令了解代码结构
- 识别关键模块和文件
- 记录项目架构到笔记
""",
        "analyze": """
当前模式: 分析代码质量

你应该:
- 查找代码问题(重复、复杂度、TODO等)
- 评估代码质量
- 将发现的问题记录为 blocker 或 action 笔记
""",
        "plan": """
当前模式: 任务规划

你应该:
- 回顾历史笔记和任务
- 制定下一步行动计划
- 更新任务状态笔记
""",
        "auto": """
当前模式: 自动决策

你应该:
- 根据用户需求灵活选择策略
- 在需要时使用工具
- 保持回答的专业性和实用性
"""
    }

    return base_instructions + mode_specific.get(mode, mode_specific["auto"])

def _postprocess_response(self, user_input: str, response: str):
    """后处理:分析回答,自动记录重要信息"""

    # 如果发现问题,自动创建 blocker 笔记
    if any(keyword in response.lower() for keyword in ["问题", "bug", "错误", "阻塞"]):
        try:
            self.note_tool.run({
                "action": "create",
                "title": f"发现问题: {user_input[:30]}...",
                "content": f"## 用户输入\n{user_input}\n## 问题分析\n{response[:500]}...",
                "note_type": "blocker",
                "tags": [self.project_name, "auto_detected", self.session_id]
            })
            self.stats["notes_created"] += 1
            self.stats["issues_found"] += 1
            print("💡 已自动创建问题笔记")
        except Exception as e:
            print(f"[WARNING] 创建笔记失败: {e}")

    # 如果是任务规划,自动创建 action 笔记
    elif any(keyword in user_input.lower() for keyword in ["计划", "下一步", "任务", "todo"]):
        try:
            self.note_tool.run({
                "action": "create",
                "title": f"任务规划: {user_input[:30]}...",
                "content": f"## 讨论\n{user_input}\n## 行动计划\n{response[:500]}...",
                "note_type": "action",
                "tags": [self.project_name, "planning", self.session_id]
            })
            self.stats["notes_created"] += 1

```

```

        print("📝 已自动创建行动计划笔记")
    except Exception as e:
        print(f"⚠️ [WARNING] 创建笔记失败: {e}")

def _update_history(self, user_input: str, response: str):
    """更新对话历史"""
    self.conversation_history.append(
        Message(content=user_input, role="user", timestamp=datetime.now())
    )
    self.conversation_history.append(
        Message(content=response, role="assistant", timestamp=datetime.now())
    )

# 限制历史长度(保留最近10轮对话)
if len(self.conversation_history) > 20:
    self.conversation_history = self.conversation_history[-20:]

# === 便捷方法 ===

def explore(self, target: str = ".") -> str:
    """探索代码库"""
    return self.run(f"请探索 {target} 的代码结构", mode="explore")

def analyze(self, focus: str = "") -> str:
    """分析代码质量"""
    query = f"请分析代码质量" + (f", 重点关注{focus}" if focus else "")
    return self.run(query, mode="analyze")

def plan_next_steps(self) -> str:
    """规划下一步任务"""
    return self.run("根据当前进度, 规划下一步任务", mode="plan")

def execute_command(self, command: str) -> str:
    """执行终端命令"""
    result = self.terminal_tool.run({"command": command})
    self.stats["commands_executed"] += 1
    return result

def create_note(
    self,
    title: str,
    content: str,
    note_type: str = "general",
    tags: List[str] = None
) -> str:
    """创建笔记"""
    result = self.note_tool.run({
        "action": "create",
        "title": title,
        "content": content,
        "note_type": note_type,
        "tags": tags or [self.project_name]
    })
    self.stats["notes_created"] += 1
    return result

def get_stats(self) -> Dict[str, Any]:
    """获取统计信息"""
    duration = (datetime.now() - self.stats["session_start"]).total_seconds()

    # 获取笔记摘要
    try:
        note_summary = self.note_tool.run({"action": "summary"})
    except:
        note_summary = {}

    return {
        "session_info": {
            "session_id": self.session_id,
            "project": self.project_name,
            "duration_seconds": duration
        },
        "activity": {
            "commands_executed": self.stats["commands_executed"],
            "notes_created": self.stats["notes_created"],
            "issues_found": self.stats["issues_found"]
        },
        "notes": note_summary
    }

def generate_report(self, save_to_file: bool = True) -> Dict[str, Any]:
    """生成会话报告"""
    report = self.get_stats()

    if save_to_file:
        report_file = f"maintainer_report_{self.session_id}.json"
        with open(report_file, 'w', encoding='utf-8') as f:
            json.dump(report, f, ensure_ascii=False, indent=2, default=str)
        report["report_file"] = report_file
        print(f"🖨 报告已保存: {report_file}")

    return report

```

9.6.4 完整使用示例

现在让我们通过一个完整的使用场景，展示这个长程智能体的工作流程：

```
# ===== 初始化助手 =====
from hello_agents import HelloAgentsLLM

maintainer = CodebaseMaintainer(
    project_name="my_flask_app",
    codebase_path=". ./my_flask_app",
    llm=HelloAgentsLLM()
)
```

```
# ===== 第一天：探索代码库 =====
```

```
# 1. 初步探索
response = maintainer.explore()
....
```

🔍 探索代码库结构...

思考... 🎓

助手：我已经扫描了代码库，这是一个 Flask Web 应用，主要包含以下模块：

```
📁 项目结构：
- app/
  └── models/      # 数据模型(User, Product, Order等)
  └── routes/       # API路由
  └── services/    # 业务逻辑
  └── utils/        # 工具函数
- tests/           # 测试文件
- migrations/     # 数据库迁移
- config.py        # 配置文件
```

总计约 3,500 行 Python 代码。

建议下一步：

1. 查看核心模型定义
2. 检查路由和API设计
3. 评估测试覆盖率

```
# 2. 深入分析某个模块
```

```
response = maintainer.run("请分析 app/models/ 目录下的数据模型设计")
....
```

助手：我查看了 app/models/ 目录，发现以下数据模型：

1. User (user.py)
 - 字段: id, username, email, password_hash, created_at
 - 关系: 一对多 Order
 - 问题: ⚠️ 缺少索引，email 字段未设置唯一约束
2. Product (product.py)
 - 字段: id, name, description, price, stock
 - 问题: ✅ 设计合理
3. Order (order.py)
 - 字段: id, user_id, product_id, quantity, status
 - 问题: ⚠️ 缺少创建时间字段，不利于数据分析

建议：

1. 为 User.email 添加唯一约束
2. 为 Order 添加 created_at 和 updated_at 字段
3. 考虑添加软删除支持

我已将这些问题记录为 blocker 笔记。

```
....
```

[自动创建了 blocker 笔记]

```
# ===== 第二天：分析代码质量 =====
```

```
# 1. 整体质量分析
response = maintainer.analyze()
....
```

📊 分析代码质量...

思考... 🎓

助手：代码质量分析报告：

📊 代码统计：

- 总行数: 3,542 行
- 平均文件长度: 约 150 行
- 最大文件: services/order_service.py (456 行)

⚠ 发现的问题:

1. 代码重复 (HIGH):
 - user_service.py 和 product_service.py 有相似的 CRUD 逻辑
 - 建议: 提取基类 BaseService
2. 复杂度过高 (MEDIUM):
 - order_service.py::process_order 方法有 8 层嵌套
 - 建议: 拆分为多个小方法
3. 缺少测试 (HIGH):
 - 测试覆盖率仅 45%
 - services/ 目录完全没有测试
4. TODO/FIXME (12 个):
 - 大部分是“稍后优化”类的注释
 - 建议: 清理或转化为正式任务

我已将这些问题记录为笔记，并标记优先级。

.....

[自动创建了多个 blocker 笔记]

2. 查看具体问题

response = maintainer.run("请查看 order_service.py 的 process_order 方法，给出重构建议")

.....

🤖 助手: 我查看了 process_order 方法的代码:

【原始代码】

```
def process_order(self, order_id):
    order = Order.query.get(order_id)
    if order:
        if order.status == 'pending':
            product = Product.query.get(order.product_id)
            if product:
                if product.stock >= order.quantity:
                    # ... 8 层嵌套继续 ...
```

【问题分析】

1. 过深的嵌套导致可读性差
2. 缺少错误处理
3. 业务逻辑与数据访问混在一起

【重构建议】

```
def process_order(self, order_id):
    # 1. 早返回模式,减少嵌套
    order = self._get_order(order_id)
    if not order or order.status != 'pending':
        return False

    # 2. 提取方法,单一职责
    if not self._check_stock(order):
        return False

    # 3. 使用事务确保一致性
    with db.transaction():
        self._update_stock(order)
        self._update_order_status(order, 'processing')

    return True
```

【下一步】

1. 我已创建重构计划笔记
2. 建议先编写测试用例
3. 逐步重构,保持功能不变

.....

[自动创建了 action 笔记]

===== 第三天: 规划重构任务 =====

1. 回顾进度

response = maintainer.plan_next_steps()

.....

📋 加载任务规划...

🤖 正在思考...

🤖 助手: 基于前两天的探索和分析,我整理了重构任务优先级:

🔴 高优先级 (本周完成):

1. [blocker] User.email 添加唯一约束
 - 风险: 低,只是数据库迁移
 - 工作量: 1小时
2. [blocker] 提取 BaseService 减少代码重复
 - 风险: 中,影响多个服务类
 - 工作量: 半天

3. [blocker] 重构 process_order 方法
 - 风险: 高, 核心业务逻辑
 - 工作量: 1天(包括编写测试)

🟡 中优先级 (下周):

4. 为 services/ 添加单元测试
5. 清理 TODO 注释
6. 为 Order 添加时间字段

🟢 低优先级 (待定):

7. 性能优化
8. 文档更新

建议:

- 本周先解决数据模型问题(任务1、6)
- 下周进行代码重构(任务2、3)
- 持续增加测试覆盖率(任务4)

我已更新任务状态笔记, 您可以随时查看进度。

.....

```
# 2. 手动创建详细的重构计划
```

```
maintainer.create_note(
    title="本周重构计划 - Week 1",
    content="## 目标
```

完成数据模型层的优化

任务清单

- [] 为 User.email 添加唯一约束
- [] 为 Order 添加 created_at, updated_at 字段
- [] 编写数据库迁移脚本
- [] 更新相关测试用例

时间安排

- 周一: 设计迁移脚本
- 周二-周三: 执行迁移并测试
- 周四: 更新测试用例
- 周五: Code Review

风险

- 数据库迁移可能影响线上环境, 需要在非高峰期执行
- 现有数据中可能存在重复email, 需要先清理

.....

```
        note_type="task_state",
        tags=["refactoring", "week1", "high_priority"]
)
```

```
print("✅ 已创建详细的重构计划")
```

```
# ===== 一周后: 检查进度 =====
```

```
# 查看笔记摘要
summary = maintainer.note_tool.run({"action": "summary"})
print("📝 笔记摘要:")
print(json.dumps(summary, indent=2, ensure_ascii=False))
.....
```

```
{
  "total_notes": 8,
  "type_distribution": {
    "blocker": 3,
    "action": 2,
    "task_state": 2,
    "conclusion": 1
  },
  "recent_notes": [
    {
      "id": "note_20250119_160000_7",
      "title": "本周重构计划 - Week 1",
      "type": "task_state",
      "updated_at": "2025-01-19T16:00:00"
    },
    ...
  ]
}
.....
```

生成完整报告

```
report = maintainer.generate_report()
print("\n📄 会话报告:")
print(json.dumps(report, indent=2, ensure_ascii=False))
{
  "session_info": {
    "session_id": "session_20250119_150000",
    "project": "my_flask_app",
    "duration_seconds": 172800 # 2天
  },
  "activity": {
    "start_time": "2025-01-19T16:00:00",
    "end_time": "2025-01-19T18:00:00"
  }
}
```

```

    "commands_executed": 24,
    "notes_created": 8,
    "issues_found": 3
},
"notes": { ... }
}.....

```

9.6.5 运行效果分析

通过这个完整的案例，我们可以看到长程智能体的几个关键特性。首先是跨会话的连贯性，智能体通过 NoteTool 保持了跨多天、多个会话的任务连贯性，第一天探索的问题在第二天分析时被自动考虑，第三天规划时能够综合前两天的所有发现，一周后检查时完整的历史都被保留。其次是智能的上下文管理，ContextBuilder 确保每次对话都有高质量的上下文，自动汇集相关笔记(特别是 blocker 类型)，根据对话模式动态调整预处理策略，在 token 预算内选择最相关的信息。

第三个特性是即时的文件系统访问，TerminalTool 支持灵活的代码探索，无需预先索引整个代码库，可以即时查看具体文件内容，支持复杂的文本处理(grep、awk等)。第四是自动化的知识管理，系统自动化地管理发现的知识，发现问题时自动创建 blocker 笔记，讨论计划时自动创建 action 笔记，关键信息自动存储到记忆系统。最后是人机协作，这个系统支持灵活的人机协作模式，智能体可以自动化地完成探索和分析，人类可以通过笔记系统进行干预和指导，支持手动创建详细的计划笔记。

这个基础框架可以进一步扩展，比如集成 RAGTool 为代码库建立向量索引结合语义检索，拆分为专门的探索者、分析者、规划者实现多智能体协作，集成测试工具自动验证重构结果，通过 TerminalTool 执行 git 命令追踪代码变更，或者使用 Gradio/Streamlit 构建可视化界面。

9.7 本章总结

在本章中，我们深入探讨了上下文工程的理论基础和工程实践：

理论层面

1. 上下文工程的本质：从“提示工程”到“上下文工程”的演进，核心是管理有限的注意力预算
2. 上下文腐蚀：理解长上下文带来的性能下降，认识到上下文是稀缺资源
3. 三大策略：压缩整合、结构化笔记、子代理架构

工程实践

1. **ContextBuilder**：实现了 GSSC 流水线，提供统一的上下文管理接口
2. **NoteTool**：Markdown+YAML 的混合格式，支持结构化的长期记忆
3. **TerminalTool**：安全的命令行工具，支持即时的文件系统访问
4. 长程智能体：整合三大工具，构建了跨会话的代码库维护助手

核心收获

- 分层设计：即时访问(TerminalTool) + 会话记忆(MemoryTool) + 持久笔记(NoteTool)
- 智能筛选：基于相关性和新近性的评分机制
- 安全第一：多层安全机制确保系统稳定
- 人机协作：自动化与可控性的平衡

通过这一章的学习，您不仅掌握了上下文工程的核心技术，更重要的是理解了如何构建能够在长时间跨度内保持连贯性和有效性的智能体系统。这些技能将成为您构建生产级智能体应用的重要基础。

在下一章中，我们将探讨智能体通信协议，学习如何让智能体与外部世界进行更广泛的交互。

习题

提示：部分习题没有标准答案，重点在于培养学习者对上下文工程和长时程任务管理的综合理解和实践能力。

1. 本章介绍了上下文工程与提示工程的区别。请分析：

- 在9.1节中提到“上下文必须被视作一种有限资源，且具有边际收益递减”。请解释什么是“上下文腐蚀”(context rot)现象？为什么即使模型支持100K甚至200K的上下文窗口，我们仍然需要谨慎管理上下文？
- 假设你要构建一个“代码审查助手”，需要分析一个包含50个文件的代码库。请对比两种策略：(1) 一次性将所有文件内容加载到上下文中；(2) 使用JIT (Just-in-time) 上下文，通过工具按需检索文件。分析各自的优缺点和适用场景。
- 在9.2.1节中提到系统提示的两个极端误区：“过度硬编码”和“过于空泛”。请各举一个实际例子，并说明如何找到合适的平衡点。

2. GSSC (Gather-Select-Structure-Compress) 流水线是本章的核心技术。请深入思考：

提示：这是一道动手实践题，建议实际操作

- 在9.3节的ContextBuilder实现中，四个阶段各有不同的职责。请分析：如果某个阶段失效（如Select阶段选择了不相关的信息，或Compress阶段过度压缩导致信息丢失），会对最终的智能体表现产生什么影响？

- 请基于9.3.4节的代码，为ContextBuilder添加一个“上下文质量评估”功能：在每次构建上下文后，自动评估上下文的信息密度、相关性和完整性，并给出优化建议。
- GSSC流水线中的“压缩”阶段使用了LLM进行智能摘要。请思考：在什么情况下，简单的截断（truncation）或滑动窗口（sliding window）策略可能比LLM摘要更合适？设计一个混合压缩策略，结合多种压缩方法的优势。

3. NoteTool和TerminalTool是支持长时程任务的关键工具。基于9.4节和9.5节的内容，请完成以下扩展实践：

提示：这是一道动手实践题，建议实际操作

- NoteTool使用了分层笔记系统（项目笔记、任务笔记、临时笔记）。请设计一个“笔记自动整理”机制：当临时笔记积累到一定数量时，智能体能够自动分析这些笔记，将重要信息提升为任务笔记或项目笔记，并清理冗余内容。
- TerminalTool提供了文件系统操作能力，但在9.5.2节中强调了安全性设计。请分析：当前的安全机制（路径验证、命令白名单、权限检查）是否足够？如果智能体需要访问敏感文件或执行危险操作，应该如何设计一个“人机协作审批”流程？
- 结合NoteTool和TerminalTool，设计一个“智能代码重构助手”：能够分析代码库结构、记录重构计划、逐步执行重构操作，并在笔记中追踪进度和遇到的问题。请画出完整的工作流程图。

4. 在9.6节的“长时程任务管理”案例中，我们看到了上下文工程在实际应用中的价值。请深入分析：

- 案例中使用了“分层上下文管理”策略：即时访问（TerminalTool）+ 会话记忆（MemoryTool）+ 持久笔记（NoteTool）。请分析：这三层之间应该如何协调？什么信息应该放在哪一层？如何避免信息冗余和不一致？
- 假设任务执行过程中发生了中断（如系统崩溃、网络断开），智能体需要从笔记中恢复状态并继续执行。请设计一个“断点续传”机制：如何在笔记中记录足够的状态信息？如何验证恢复后的状态是否正确？
- 长时程任务往往涉及多个子任务的并行或串行执行。请设计一个“任务依赖管理”系统：能够表达任务之间的依赖关系（如“任务B必须在任务A完成后执行”），并自动调度任务执行顺序。这个系统应该如何与NoteTool集成？

5. 本章多次提到“渐进式披露”（progressive disclosure）的概念。请思考：

- 在9.2.2节中，渐进式披露被描述为“每一步交互都会产生新的上下文，反过来指导下一步决策”。请设计一个具体的应用场景（如学术论文写作、复杂问题调试），展示渐进式披露如何帮助智能体更高效地完成任务。
- 渐进式披露的一个潜在风险是“探索效率低下”：智能体可能会在不重要的细节上浪费时间，或者错过关键信息。请设计一个“探索引导”机制：通过启发式规则或元认知策略，帮助智能体更聪明地决定“下一步应该探索什么”。
- 对比“渐进式披露”与传统的“一次性加载所有上下文”：在什么类型的任务中，前者有明显优势？在什么类型的任务中，后者可能更合适？请给出至少3个不同类型的任务示例。

参考文献

- [1] Anthropic. Effective Context Engineering for AI Agents. <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>
- [2] David Kim. Context-Engineering (GitHub). <https://github.com/davidkimai/Context-Engineering>

第十章 智能体通信协议

在前面的章节中，我们构建了功能完备的单体智能体，它们具备推理、工具调用和记忆能力。然而，当我们尝试构建更复杂的 AI 系统时，自然会有疑问：如何让智能体与外部世界高效交互？如何让多个智能体相互协作？

这正是智能体通信协议要解决的核心问题。本章将为 HelloAgents 框架引入三种通信协议：**MCP (Model Context Protocol)** 用于智能体与工具的标准化通信，**A2A (Agent-to-Agent Protocol)** 用于智能体间的点对点协作，**ANP (Agent Network Protocol)** 用于构建大规模智能体网络。这三种协议共同构成了智能体通信的基础设施层。

通过本章的学习，您将掌握智能体通信协议的设计理念和实践技能，理解三种主流协议的设计差异，学会如何选择合适的协议来解决实际问题。

10.1 智能体通信协议基础

10.1.1 为何需要通信协议

回顾我们在第七章构建的 ReAct 智能体，它已经具备了强大的推理和工具调用能力。让我们看一个典型的使用场景：

```
from hello_agents import ReActAgent, HelloAgentsLLM
from hello_agents.tools import CalculatorTool, SearchTool

llm = HelloAgentsLLM()
agent = ReActAgent(name="AI助手", llm=llm)
agent.add_tool(CalculatorTool())
agent.add_tool(SearchTool())

# 智能体可以独立完成任务
response = agent.run("搜索最新的AI新闻，并计算相关公司的市值总和")
```

这个智能体工作得很好，但它面临着三个根本性的限制。首先是工具集成的困境：每当需要访问新的外部服务（如 GitHub API、数据库、文件系统），我们都必须编写专门的 Tool 类。这不仅工作量大，而且不同开发者编写的工具无法互相兼容。其次是能力扩展的瓶颈：智能体的能力被限制在预先定义的工具集内，无法动态发现和使用新的服务。最后是协作的缺失：当任务复杂到需要多个专业智能体协作时（如研究员+撰写员+编辑），我们只能通过手动编排来协调它们的工作。

让我们通过一个更具体的例子来理解这些限制。假设你要构建一个智能研究助手，它需要：

```
# 传统方式：手动集成每个服务
class GitHubTool(BaseTool):
    """需要手写GitHub API适配器"""
    def run(self, repo_url):
        # 大量的API调用代码...
        pass

class DatabaseTool(BaseTool):
    """需要手写数据库适配器"""
    def run(self, query):
        # 数据库连接和查询代码...
        pass

class WeatherTool(BaseTool):
    """需要手写天气API适配器"""
    def run(self, location):
        # 天气API调用代码...
        pass

# 每个新服务都需要重复这个过程
agent.add_tool(GitHubTool())
agent.add_tool(DatabaseTool())
agent.add_tool(WeatherTool())
```

这种方式存在明显的问题：代码重复（每个工具都要处理 HTTP 请求、错误处理、认证等），难以维护（API 变更需要修改所有相关工具），无法复用（其他开发者的工具无法直接使用），扩展性差（添加新服务需要大量编码工作）。

通信协议的核心价值正是解决这些问题。它提供了一套标准化的接口规范，让智能体能够以统一的方式访问各种外部服务，而无需为每个服务编写专门的适配器。这就像互联网的 TCP/IP 协议，它让不同的设备能够相互通信，而不需要为每种设备编写专门的通信代码。

有了通信协议，上面的代码可以简化为：

```
from hello_agents.tools import MCPTool

# 连接到MCP服务器，自动获得所有工具
mcp_tool = MCPTool() # 内置服务器提供基础工具

# 或者连接到专业的MCP服务器
github_mcp = MCPTool(server_command=["npx", "y", "@modelcontextprotocol/server-github"])
database_mcp = MCPTool(server_command=["python", "database_mcp_server.py"])

# 智能体自动获得所有能力，无需手写适配器
agent.add_tool(mcp_tool)
```

```
agent.add_tool(github_mcp)
agent.add_tool(database_mcp)
```

通信协议带来的改变是根本性的：标准化接口让不同服务提供统一的访问方式，互操作性使得不同开发者的工具可以无缝集成，动态发现允许智能体在运行时发现新的服务和能力，可扩展性让系统能够轻松添加新的功能模块。

10.1.2 三种协议设计理念比较

智能体通信协议并非单一的解决方案，而是针对不同通信场景设计的一系列标准。在本章以目前业界主流的三种协议 MCP、A2A 和 ANP 为例进行实践，下面是一个总览的比较。

(1) MCP：智能体与工具的桥梁

MCP (Model Context Protocol) 由 Anthropic 团队提出^[1]，其核心设计理念是标准化智能体与外部工具/资源的通信方式。想象一下，你的智能体需要访问文件系统、数据库、GitHub、Slack 等各种服务。传统做法是为每个服务编写专门的适配器，这不仅工作量大，而且难以维护。MCP 通过定义统一的协议规范，让所有服务都能以相同的方式被访问。

MCP 的设计哲学是“上下文共享”。它不仅仅是一个 RPC（远程过程调用）协议，更重要的是它允许智能体和工具之间共享丰富的上下文信息。如图 10.1 所示，当智能体访问一个代码仓库时，MCP 服务器不仅能提供文件内容，还能提供代码结构、依赖关系、提交历史等上下文信息，让智能体能够做出更智能的决策。



图 10.1 MCP 设计思想

(2) A2A：智能体间的对话

A2A (Agent-to-Agent Protocol) 协议由 Google 团队提出^[2]，其核心设计理念是实现智能体之间的点对点通信。与 MCP 关注智能体与工具的通信不同，A2A 关注的是智能体之间如何相互协作。这种设计让智能体能够像人类团队一样进行对话、协商和协作。

A2A 的设计哲学是“对等通信”。如图 10.2 所示，在 A2A 网络中，每个智能体既是服务提供者，也是服务消费者。智能体可以主动发起请求，也可以响应其他智能体的请求。这种对等的设计避免了中心化协调器的瓶颈，让智能体网络更加灵活和可扩展。



图 10.2 A2A 设计思想

(3) ANP：智能体网络的基础设施

ANP (Agent Network Protocol) 是一个概念性的协议框架^[3]，目前由开源社区维护，还没有成熟的生态，其核心设计理念是构建大规模智能体网络的基础设施。如果说 MCP 解决的是“如何访问工具”，A2A 解决的是“如何与其他智能体对话”，那么 ANP 解决的是“如何在大规模网络中发现和连接智能体”。

ANP 的设计哲学是“去中心化服务发现”。在一个包含成百上千个智能体的网络中，如何让智能体能够找到它需要的服务？如图 10.3 所示，ANP 提供了服务注册、发现和路由机制，让智能体能够动态地发现网络中的其他服务，而不需要预先配置所有的连接关系。



图 10.3 ANP 设计思想

最后在表 10.1 中，让我们通过一个对比表格来更清晰地理解这三种协议的差异：

表 10.1 三种协议对比



(4) 如何选择合适的协议？

目前的协议还处于发展早期，MCP 的生态相对成熟，不过各种工具的时效性取决于维护者，更推荐选择大公司背书的 MCP 工具。

选择协议的关键在于理解你的需求：

- 如果你的智能体需要访问外部服务（文件、数据库、API），选择**MCP**
- 如果你需要多个智能体相互协作完成任务，选择**A2A**
- 如果你要构建大规模的智能体生态系统，考虑**ANP**

10.1.3 HelloAgents 通信协议架构设计

在理解了三种协议的设计理念后，让我们看看如何在 HelloAgents 框架中实现和使用它们。我们的设计目标是：让学习者能够以最简单的方式使用这些协议，同时保持足够的灵活性以应对复杂场景。

如图 10.4 所示，HelloAgents 的通信协议架构采用三层设计，从底层到上层分别是：协议实现层、工具封装层和智能体集成层。



图 10.4 HelloAgents 通信协议设计

(1) 协议实现层：这一层包含了三种协议的具体实现。MCP 基于 FastMCP 库实现，提供客户端和服务器功能；A2A 基于 Google 官方的 a2a-sdk 实现；ANP 是我们自研的轻量级实现，提供服务发现和网络管理功能，当然目前也有官方的实现，考虑到后期的迭代，因此这里只做概念的模拟。

(2) 工具封装层：这一层将协议实现封装成统一的 Tool 接口。MCPTool、A2ATool 和 ANPTool 都继承自 BaseTool，提供一致的 run() 方法。这种设计让智能体能够以相同的方式使用不同的协议。

(3) 智能体集成层：这一层是智能体与协议的集成点。所有的智能体（ReActAgent、SimpleAgent 等）都通过 Tool System 来使用协议工具，无需关心底层的协议细节。

10.1.4 本章学习目标与快速体验

让我们先看看第十章的学习内容：

```
hello_agents/
└── protocols/
    ├── mcp/
    │   ├── client.py
    │   ├── server.py
    │   └── utils.py
    ├── a2a/
    │   └── implementation.py
    └── anp/
        └── implementation.py
    tools/builtin/
        └── protocol_tools.py
# 通信协议模块
# MCP协议实现 (Model Context Protocol)
# MCP客户端 (支持5种传输方式)
# MCP服务器 (FastMCP封装)
# 工具函数 (create_context/parse_context)
# A2A协议实现 (Agent-to-Agent Protocol)
# A2A服务器/客户端 (基于a2a-sdk, 可选依赖)
# ANP协议实现 (Agent Network Protocol)
# ANP服务发现/注册 (概念性实现)
# 内置工具模块
# 协议工具包装器 (MCPTool/A2ATool/ANPTool)
```

对于这一章的内容，主要是应用为主，学习目标是能拥有在自己项目中应用协议的能力。并且协议目前发展处于早期，所以无需花费太多精力去造轮子。在开始实战之前，让我们先准备好开发环境：

```
# 安装HelloAgents框架 (第10章版本)
pip install "hello-agents[protocol]==0.2.2"
```

```
# 安装NodeJS，可以参考Additional-Chapter中的文档
```

让我们用最简单的代码体验一下三种协议的基本功能：

```
from hello_agents.tools import MCPTool, A2ATool, ANPTool

# 1. MCP: 访问工具
mcp_tool = MCPTool()
result = mcp_tool.run({
    "action": "call_tool",
    "tool_name": "add",
    "arguments": {"a": 10, "b": 20}
})
print(f"MCP计算结果: {result}") # 输出: 30.0

# 2. ANP: 服务发现
anp_tool = ANPTool()
anp_tool.run({
    "action": "register_service",
    "service_id": "calculator",
    "service_type": "math",
    "endpoint": "http://localhost:8080"
})
services = anp_tool.run({"action": "discover_services"})
print(f"发现的服务: {services}")

# 3. A2A: 智能体通信
a2a_tool = A2ATool("http://localhost:5000")
print("A2A工具创建成功")
```

这个简单的示例展示了三种协议的核心功能。在接下来的章节中，我们将深入学习每种协议的详细用法和最佳实践。

10.2 MCP 协议实战

现在，让我们深入学习 MCP，掌握如何让智能体访问外部工具和资源。

10.2.1 MCP 协议概念介绍

(1) MCP：智能体的“USB-C”

想象一下，你的智能体可能需要同时做很多事情，例如： - 读取本地文件系统的文档 - 查询 PostgreSQL 数据库 - 搜索 GitHub 上的代码 - 发送 Slack 消息 - 访问 Google Drive

传统方式下，你需要为每个服务编写适配器代码，处理不同的 API、认证方式、错误处理等。这不仅工作量大，而且难以维护。更重要的是，不同 LLM 平台的 function call 实现差异巨大，切换模型时需要重写大量代码。

MCP 的出现改变了这一切。它就像 USB-C 统一了各种设备的连接方式一样，MCP 统一了智能体与外部工具的交互方式。无论你使用 Claude、GPT 还是其他模型，只要它们支持 MCP 协议，就能无缝访问相同的工具和资源。

(2) MCP 架构

MCP 协议采用 Host、Client、Servers 三层架构设计，让我们通过图 10.5 的场景来理解这些组件如何协同工作。

假设你正在使用 Claude Desktop 询问：“我桌面上有哪些文档？”



图 10.5 MCP 案例演示

三层架构的职责：

1. **Host**（宿主层）：Claude Desktop 作为 Host，负责接收用户提问并与 Claude 模型交互。Host 是用户直接交互的界面，它管理整个对话流程。
2. **Client**（客户端层）：当 Claude 模型决定需要访问文件系统时，Host 中内置的 MCP Client 被激活。Client 负责与适当的 MCP Server 建立连接，发送请求并接收响应。
3. **Server**（服务器层）：文件系统 MCP Server 被调用，执行实际的文件扫描操作，访问桌面目录，并返回找到的文档列表。

完整的交互流程：用户问题 → Claude Desktop(Host) → Claude 模型分析 → 需要文件信息 → MCP Client 连接 → 文件系统 MCP Server → 执行操作 → 返回结果 → Claude 生成回答 → 显示在 Claude Desktop 上

这种架构设计的优势在于关注点分离：Host 专注于用户体验，Client 专注于协议通信，Server 专注于具体功能实现。开发者只需专注于开发对应的 MCP Server，无需关心 Host 和 Client 的实现细节。

(3) MCP 的核心能力

如表 10.2 所示，MCP 协议提供了三大核心能力，构成完整的工具访问框架：

表 10.2 MCP 核心能力



这三种能力的区别在于：**Tools** 是主动的（执行操作），**Resources** 是被动的（提供数据），**Prompts** 是指导性的（提供模板）。

(4) MCP 的工作流程

让我们通过一个具体例子来理解 MCP 的完整工作流程，如图 10.6 所示：



图 10.6 MCP 案例演示

一个关键问题是：**Claude**（或其他 LLM）是如何决定使用哪些工具的？

当用户提出问题时，完整的工具选择流程如下：

1. 工具发现阶段：MCP Client 连接到 Server 后，首先调用 `list_tools()` 获取所有可用工具的描述信息（包括工具名称、功能说明、参数定义）
2. 上下文构建：Client 将工具列表转换为 LLM 能理解的格式，添加到系统提示词中。例如：

 - 你可以使用以下工具：
 - `read_file(path: str)`: 读取指定路径的文件内容
 - `search_code(query: str, language: str)`: 在代码库中搜索

3. 模型推理：LLM 分析用户问题和可用工具，决定是否需要调用工具以及调用哪个工具。这个决策基于工具的描述和当前对话上下文
4. 工具执行：如果 LLM 决定使用工具，Client 通过 MCP Server 执行所选工具，获取结果
5. 结果整合：工具执行结果被送回给 LLM，LLM 结合结果生成最终回答

这个过程是完全自动化的，LLM 会根据工具描述的质量来决定是否使用以及如何使用工具。因此，编写清晰、准确的工具描述至关重要。

(5) MCP 与 Function Calling 的差异

很多开发者会问：我已经在用 **Function Calling** 了，为什么还需要 **MCP**？让我们通过表 10.3 来理解它们的区别。

表 10.3 Function Calling 与 MCP 对比



这里我们以智能体需要访问 GitHub 仓库和本地文件系统为例子来详细对比同一个任务的两种实现

方式 1：使用 Function Calling

```
# 步骤1：为每个LLM提供商定义函数
# OpenAI格式
openai_tools = [
    {
        "name": "github",
        "function": "github_search",
        "description": "Searches for code in GitHub repositories based on a query and language."}
]
```

```

        "type": "function",
        "function": {
            "name": "search_github",
            "description": "搜索GitHub仓库",
            "parameters": {
                "type": "object",
                "properties": {
                    "query": {"type": "string", "description": "搜索关键词"}
                },
                "required": ["query"]
            }
        }
    }
]

# Claude格式
claude_tools = [
    {
        "name": "search_github",
        "description": "搜索GitHub仓库",
        "input_schema": { # 注意: 不是parameters
            "type": "object",
            "properties": {
                "query": {"type": "string", "description": "搜索关键词"}
            },
            "required": ["query"]
        }
    }
]

# 步骤2: 自己实现工具函数
def search_github(query):
    import requests
    response = requests.get(
        "https://api.github.com/search/repositories",
        params={"q": query}
    )
    return response.json()

# 步骤3: 处理不同模型的响应格式
# OpenAI的响应
if response.choices[0].message.tool_calls:
    tool_call = response.choices[0].message.tool_calls[0]
    result = search_github(**json.loads(tool_call.function.arguments))

# Claude的响应
if response.content[0].type == "tool_use":
    tool_use = response.content[0]
    result = search_github(**tool_use.input)

```

方式 2: 使用 MCP

```

from hello_agents.protocols import MCPClient

# 步骤1: 连接到社区提供的MCP服务器 (无需自己实现)
github_client = MCPClient([
    "npx", "-y", "@modelcontextprotocol/server-github"
])

fs_client = MCPClient([
    "npx", "-y", "@modelcontextprotocol/server-filesystem", "."
])

# 步骤2: 统一的调用方式 (与模型无关)
async with github_client:
    # 自动发现工具
    tools = await github_client.list_tools()

    # 调用工具 (标准化接口)
    result = await github_client.call_tool(
        "search_repositories",
        {"query": "AI agents"}
    )

# 步骤3: 任何支持MCP的模型都能使用
# OpenAI、Claude、Llama等都使用相同的MCP客户端

```

首先需要明确的是，Function Calling 与 MCP 并非竞争关系，而是相辅相成的。Function Calling 是大语言模型的一项核心能力，它体现了模型内在的智能，使模型能够理解何时需要调用函数，并精准生成相应的调用参数。相对地，MCP 则扮演着基础设施协议的角色，它在工程层面解决了工具与模型如何连接的问题，通过标准化的方式来描述和调用工具。

我们可以用一个简单的类比来理解：Function Calling 相当于你学会了“如何打电话”这项技能，包括何时拨号、如何与对方沟通、何时挂断。而 MCP 则是那个全球统一的“电话通信标准”，确保了任何一部电话都能顺利地拨通另一部。

了解了它们之间的互补关系后，我们接下来看看如何在 HelloAgents 中使用 MCP 协议。

10.2.2 使用 MCP 客户端

HelloAgents 基于 FastMCP 2.0 实现了完整的 MCP 客户端功能。我们提供了异步和同步两种 API，以适应不同的使用场景。对于大多数应用，推荐使用异步 API，它能更好地处理并发请求和长时间运行的操作。下面我们将提供一个拆解的操作演示。

(1) 连接到 MCP 服务器

MCP 客户端支持多种连接方式，最常用的是 Stdio 模式（通过标准输入输出与本地进程通信）：

```
import asyncio
from hello_agents.protocols import MCPClient

async def connect_to_server():
    # 方式1：连接到社区提供的文件系统服务器
    # npx会自动下载并运行@modelcontextprotocol/server-filesystem包
    client = MCPClient([
        "npx", "-y",
        "@modelcontextprotocol/server-filesystem",
        ".." # 指定根目录
    ])

    # 使用async with确保连接正确关闭
    async with client:
        # 在这里使用client
        tools = await client.list_tools()
        print(f"可用工具: {[t['name'] for t in tools]})

    # 方式2：连接到自定义的Python MCP服务器
    client = MCPClient(["python", "my_mcp_server.py"])
    async with client:
        # 使用client...
        pass

# 运行异步函数
asyncio.run(connect_to_server())
```

(2) 发现可用工具

连接成功后，第一步通常是查询服务器提供了哪些工具：

```
async def discover_tools():
    client = MCPClient(["npx", "-y", "@modelcontextprotocol/server-filesystem", "."])

    async with client:
        # 获取所有可用工具
        tools = await client.list_tools()

        print(f"服务器提供了 {len(tools)} 个工具: ")
        for tool in tools:
            print(f"\n工具名称: {tool['name']}")
            print(f"描述: {tool.get('description', '无描述')}")

            # 打印参数信息
            if 'inputSchema' in tool:
                schema = tool['inputSchema']
                if 'properties' in schema:
                    print("参数:")
                    for param_name, param_info in schema['properties'].items():
                        param_type = param_info.get('type', 'any')
                        param_desc = param_info.get('description', '')
                        print(f" - {param_name}: {param_type} - {param_desc}")

    asyncio.run(discover_tools())

# 输出示例:
# 服务器提供了 5 个工具:
#
# 工具名称: read_file
# 描述: 读取文件内容
# 参数:
#   - path (string): 文件路径
#
# 工具名称: write_file
# 描述: 写入文件内容
# 参数:
#   - path (string): 文件路径
#   - content (string): 文件内容
```

(3) 调用工具

调用工具时，只需提供工具名称和符合 JSON Schema 的参数：

```
async def use_tools():
    client = MCPClient(["npx", "-y", "@modelcontextprotocol/server-filesystem", "."])

    async with client:
        # 读取文件
        result = await client.call_tool("read_file", {"path": "my_README.md"})
        print(f"文件内容: \n{result}")
```

```

# 列出目录
result = await client.call_tool("list_directory", {"path": "."})
print(f"当前目录文件: {result}")

# 写入文件
result = await client.call_tool("write_file", {
    "path": "output.txt",
    "content": "Hello from MCP!"
})
print(f"写入结果: {result}")

asyncio.run(use_tools())

```

在这里提供一种更为安全的方式来调用 MCP 服务，可供参考：

```

async def safe_tool_call():
    client = MCPClient(["npx", "-y", "@modelcontextprotocol/server-filesystem", "."])

    async with client:
        try:
            # 尝试读取可能不存在的文件
            result = await client.call_tool("read_file", {"path": "nonexistent.txt"})
            print(result)
        except Exception as e:
            print(f"工具调用失败: {e}")
            # 可以选择重试、使用默认值或向用户报告错误

asyncio.run(safe_tool_call())

```

(4) 访问资源

除了工具，MCP 服务器还可以提供资源（Resources）：

```

# 列出可用资源
resources = client.list_resources()
print(f"可用资源: {[r['uri'] for r in resources]}")

# 读取资源
resource_content = client.read_resource("file:///path/to/resource")
print(f"资源内容: {resource_content}")

```

(5) 使用提示模板

MCP 服务器可以提供预定义的提示模板（Prompts）：

```

# 列出可用提示
prompts = client.list_prompts()
print(f"可用提示: {[p['name'] for p in prompts]}")

# 获取提示内容
prompt = client.get_prompt("code_review", {"language": "python"})
print(f"提示内容: {prompt}")

```

(6) 完整示例：使用 GitHub MCP 服务

让我们通过一个完整的例子来看如何使用社区提供的 GitHub MCP 服务，我们将采用封装好的 MCP Tools 来：

```

.....
GitHub MCP 服务示例

注意：需要设置环境变量
Windows: $env:GITHUB_PERSONAL_ACCESS_TOKEN="your_token_here"
Linux/macOS: export GITHUB_PERSONAL_ACCESS_TOKEN="your_token_here"
.....

from hello_agents.tools import MCPTool

# 创建 GitHub MCP 工具
github_tool = MCPTool(
    server_command=["npx", "-y", "@modelcontextprotocol/server-github"]
)

# 1. 列出可用工具
print("可用工具: ")
result = github_tool.run({"action": "list_tools"})
print(result)

# 2. 搜索仓库
print("\n🔍 搜索仓库: ")
result = github_tool.run({
    "action": "call_tool",
    "tool_name": "search_repositories",
    "arguments": {
        "query": "AI agents language:python",
        "page": 1,
        "perPage": 3
    }
})
print(result)

```

10.2.3 MCP 传输方式详解

MCP 协议的一个重要特性是传输层无关性（Transport Agnostic）。这意味着 MCP 协议本身不依赖于特定的传输方式，可以在不同的通信通道上运行。HelloAgents 基于 FastMCP 2.0，提供了完整的传输方式支持，让你可以根据实际场景选择最合适的传输模式。

(1) 传输方式概览

HelloAgents 的 `MCPClient` 支持五种传输方式，每种都有不同的使用场景，如表 10.4 所示：

表 10.4 MCP 传输方式对比

| | Memory Transport | Stdio Transport | Stdio Transport with Args | Npx Transport | HTTP/SSE/StreamableHTTP Transport |
|------|-----------------------------------|--|---|---|---|
| 适用场景 | 单元测试、快速原型开发 | 本地开发、调试、Python 脚本服务器 | 带参数的命令传输 | 社区服务器（npx 方式） | 生产环境、远程服务、微服务架构 |
| 特点 | 内存传输，不指定任何参数，使用内置演示服务器 | 标准输入输出传输，使用命令列表启动本地服务器 | 可以传递额外参数 | 使用 npx 启动社区 MCP 服务器 | 注意：MCPTool 主要用于 Stdio 和 Memory 传输，对于 HTTP/SSE 等远程传输，建议直接使用 MCPClient |
| 示例 | <code>mcp_tool = MCPTool()</code> | <code>mcp_tool = MCPTool(server_command=["python", "examples/mcp_example_server.py"])</code> | <code>mcp_tool = MCPTool(server_command=["python", "examples/mcp_example_server.py", "--debug"])</code> | <code>mcp_tool = MCPTool(server_command=["npx", "-y", "@modelcontextprotocol/server-filesystem", "."])</code> | <code>mcp_tool = MCPTool(server_command=[["http", "sse", "streamablehttp"], {"port": 8080}])</code> |

(2) 传输方式使用示例

```
from hello_agents.tools import MCPTool

# 1. Memory Transport - 内存传输 (用于测试)
# 不指定任何参数，使用内置演示服务器
mcp_tool = MCPTool()

# 2. Stdio Transport - 标准输入输出传输 (本地开发)
# 使用命令列表启动本地服务器
mcp_tool = MCPTool(server_command=["python", "examples/mcp_example_server.py"])

# 3. Stdio Transport with Args - 带参数的命令传输
# 可以传递额外参数
mcp_tool = MCPTool(server_command=["python", "examples/mcp_example_server.py", "--debug"])

# 4. Stdio Transport - 社区服务器 (npx 方式)
# 使用 npx 启动社区 MCP 服务器
mcp_tool = MCPTool(server_command=["npx", "-y", "@modelcontextprotocol/server-filesystem", "."])

# 5. HTTP/SSE/StreamableHTTP Transport
# 注意：MCPTool主要用于Stdio和Memory传输
# 对于HTTP/SSE等远程传输，建议直接使用MCPClient
```

(3) Memory Transport - 内存传输

适用场景：单元测试、快速原型开发

```
from hello_agents.tools import MCPTool

# 使用内置演示服务器 (Memory 传输)
mcp_tool = MCPTool()

# 列出可用工具
result = mcp_tool.run({"action": "list_tools"})
print(result)

# 调用工具
result = mcp_tool.run({
    "action": "call_tool",
    "tool_name": "add",
    "arguments": {"a": 10, "b": 20}
})
print(result)
```

(4) Stdio Transport - 标准输入输出传输

适用场景：本地开发、调试、Python 脚本服务器

```
from hello_agents.tools import MCPTool

# 方式1：使用自定义Python服务器
mcp_tool = MCPTool(server_command=["python", "my_mcp_server.py"])

# 方式2：使用社区服务器（文件系统）
mcp_tool = MCPTool(server_command=["npx", "-y", "@modelcontextprotocol/server-filesystem", "."])

# 列出工具
result = mcp_tool.run({"action": "list_tools"})
print(result)

# 调用工具
result = mcp_tool.run({
    "action": "call_tool",
    "tool_name": "read_file",
    "arguments": {"path": "README.md"}
})
print(result)
```

(5) HTTP Transport - HTTP 传输

适用场景：生产环境、远程服务、微服务架构

```
# 注意: MCPTool 主要用于 Stdio 和 Memory 传输
# 对于 HTTP/SSE 等远程传输, 建议使用底层的 MCPClient

import asyncio
from hello_agents.protocols import MCPClient

async def test_http_transport():
    # 连接到远程 HTTP MCP 服务器
    client = MCPClient("http://api.example.com/mcp")

    async with client:
        # 获取服务器信息
        tools = await client.list_tools()
        print(f"远程服务器工具: {len(tools)} 个")

        # 调用远程工具
        result = await client.call_tool("process_data", {
            "data": "Hello, World!",
            "operation": "uppercase"
        })
        print(f"远程处理结果: {result}")

# 注意: 需要实际的 HTTP MCP 服务器
# asyncio.run(test_http_transport())
```

(6) SSE Transport - Server-Sent Events 传输

适用场景: 实时通信、流式处理、长连接

```
# 注意: MCPTool 主要用于 Stdio 和 Memory 传输
# 对于 SSE 传输, 建议使用底层的 MCPClient

import asyncio
from hello_agents.protocols import MCPClient

async def test_sse_transport():
    # 连接到 SSE MCP 服务器
    client = MCPClient(
        "http://localhost:8080/sse",
        transport_type="sse"
    )

    async with client:
        # SSE 特别适合流式处理
        result = await client.call_tool("stream_process", {
            "input": "大量数据处理请求",
            "stream": True
        })
        print(f"流式处理结果: {result}")

# 注意: 需要支持 SSE 的 MCP 服务器
# asyncio.run(test_sse_transport())
```

(7) StreamableHTTP Transport - 流式 HTTP 传输

适用场景: 需要双向流式通信的 HTTP 场景

```
# 注意: MCPTool 主要用于 Stdio 和 Memory 传输
# 对于 StreamableHTTP 传输, 建议使用底层的 MCPClient

import asyncio
from hello_agents.protocols import MCPClient

async def test_streamable_http_transport():
    # 连接到 StreamableHTTP MCP 服务器
    client = MCPClient(
        "http://localhost:8080/mcp",
        transport_type="streamable_http"
    )

    async with client:
        # 支持双向流式通信
        tools = await client.list_tools()
        print(f"StreamableHTTP 服务器工具: {len(tools)} 个")

# 注意: 需要支持 StreamableHTTP 的 MCP 服务器
# asyncio.run(test_streamable_http_transport())
```

10.2.4 在智能体中使用 MCP 工具

前面我们学习了如何直接使用 MCP 客户端。但在实际应用中，我们更希望让智能体自动调用 MCP 工具，而不是手动编写调用代码。HelloAgents 提供了 MCPTool 包装器，让 MCP 服务器无缝集成到智能体的工具链中。

(1) MCP 工具的自动展开机制

HelloAgents 的 MCPTool 有一个特性：自动展开。当你添加一个 MCP 工具到 Agent 时，它会自动将 MCP 服务器提供的所有工具展开为独立的工具，让 Agent 可以像调用普通工具一样调用它们。

方式 1：使用内置演示服务器

我们在之前实现过计算器的工具函数，在这里将他转化为 MCP 的服务。这是最简单的使用方式。

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import MCPTool

agent = SimpleAgent(name="助手", llm=HelloAgentsLLM())

# 无需任何配置，自动使用内置演示服务器
mcp_tool = MCPTool(name="calculator")
agent.add_tool(mcp_tool)
# 🎉 MCP 工具 'calculator' 已展开为 6 个独立工具

# 智能体可以直接使用展开后的工具
response = agent.run("计算 25 乘以 16")
print(response) # 输出: 25 乘以 16 的结果是 400
```

自动展开后的工具：

- calculator_add - 加法计算器
- calculator_subtract - 减法计算器
- calculator_multiply - 乘法计算器
- calculator_divide - 除法计算器
- calculator_greet - 友好问候
- calculator_get_system_info - 获取系统信息

Agent 调用时只需提供参数，例如：[TOOL_CALL:calculator_multiply:a=25,b=16]，系统会自动处理类型转换和 MCP 调用。

方式 2：连接外部 MCP 服务器

在实际项目中，你需要连接到功能更强大的 MCP 服务器。这些服务器可以是：

- 社区提供的官方服务器（如文件系统、GitHub、数据库等）
- 你自己编写的自定义服务器（封装业务逻辑）

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import MCPTool

agent = SimpleAgent(name="文件助手", llm=HelloAgentsLLM())

# 示例1：连接到社区提供的文件系统服务器
fs_tool = MCPTool(
    name="filesystem", # 指定唯一名称
    description="访问本地文件系统",
    server_command=["npx", "-y", "@modelcontextprotocol/server-filesystem", "."]
)
agent.add_tool(fs_tool)

# 示例2：连接到自定义的 Python MCP 服务器
# 关于如何编写自定义MCP服务器，请参考10.5章节
custom_tool = MCPTool(
    name="custom_server", # 使用不同的名称
    description="自定义业务逻辑服务器",
    server_command=["python", "my_mcp_server.py"]
)
agent.add_tool(custom_tool)

# Agent现在可以自动使用这些工具！
response = agent.run("请读取my_README.md文件，并总结其中的主要内容")
print(response)
```

当使用多个 MCP 服务器时，务必为每个 MCPTool 指定不同的 name，这个 name 会作为前缀添加到展开的工具名前，避免冲突。例如：name="fs" 会展开为 fs_read_file、fs_write_file 等。如果你需要编写自己的 MCP 服务器来封装特定的业务逻辑，请参考 10.5 节内容。

(2) MCP 工具自动展开的工作原理

理解自动展开机制有助于你更好地使用 MCP 工具。让我们深入了解它是如何工作的：

```
# 用户代码
fs_tool = MCPTool(name="fs", server_command=[...])
agent.add_tool(fs_tool)

# 内部发生的事情：
# 1. MCPTool连接到服务器，发现14个工具
# 2. 为每个工具创建包装器：
#     - fs_read_text_file (参数: path, tail, head)
#     - fs_write_file (参数: path, content)
#     - ...
# 3. 注册到Agent的工具注册表

# Agent调用
response = agent.run("读取README.md")

# Agent内部：
```

```
# 1. 识别需要调用 fs_read_text_file
# 2. 生成参数: path=README.md
# 3. 包装器转换为MCP格式:
# {"action": "call_tool", "tool_name": "read_text_file", "arguments": {"path": "README.md"}}
# 4. 调用MCP服务器
# 5. 返回文件内容
```

系统会根据工具的参数定义自动转换类型:

```
# Agent调用计算器
agent.run("计算 25 乘以 16")

# Agent生成: a=25,b=16 (字符串)
# 系统自动转换为: {"a": 25.0, "b": 16.0} (数字)
# MCP服务器接收到正确的数字类型
```

(3) 实战案例: 智能文档助手

让我们构建一个完整的智能文档助手, 这里我们用一个简单的多智能体编排进行演示:

```
#####
多Agent协作的智能文档助手

使用两个SimpleAgent分工协作:
- Agent1: GitHub搜索专家
- Agent2: 文档生成专家
#####

from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import MCPTool
from dotenv import load_dotenv

# 加载.env文件中的环境变量
load_dotenv(dotenv_path='../HelloAgents/.env')

print("*" * 70)
print("多Agent协作的智能文档助手")
print("*" * 70)

# =====
# Agent 1: GitHub搜索专家
# =====
print("\n【步骤1】创建GitHub搜索专家...")

github_searcher = SimpleAgent(
    name="GitHub搜索专家",
    llm=HelloAgentsLLM(),
    system_prompt="""你是一个GitHub搜索专家。
你的任务是搜索GitHub仓库并返回结果。
请返回清晰、结构化的搜索结果, 包括:
- 仓库名称
- 简短描述

保持简洁, 不要添加额外的解释。""")
)

# 添加GitHub工具
github_tool = MCPTool(
    name="gh",
    server_command=["npx", "-y", "@modelcontextprotocol/server-github"]
)
github_searcher.add_tool(github_tool)

# =====
# Agent 2: 文档生成专家
# =====
print("\n【步骤2】创建文档生成专家...")

document_writer = SimpleAgent(
    name="文档生成专家",
    llm=HelloAgentsLLM(),
    system_prompt="""你是一个文档生成专家。
你的任务是根据提供的信息生成结构化的Markdown报告。

报告应该包括:
- 标题
- 简介
- 主要内容 (分点列出, 包括项目名称、描述等)
- 总结

请直接输出完整的Markdown格式报告内容, 不要使用工具保存。""")
)

# 添加文件系统工具
fs_tool = MCPTool(
    name="fs",
    server_command=["npx", "-y", "@modelcontextprotocol/server-filesystem", "."]
)
document_writer.add_tool(fs_tool)
```

```

# =====
# 执行任务
# =====
print("\n" + "="*70)
print("开始执行任务...")
print("="*70)

try:
    # 步骤1: GitHub搜索
    print("\n【步骤3】Agent1 搜索GitHub...")
    search_task = "搜索关于'AI agent'的GitHub仓库, 返回前5个最相关的结果"

    search_results = github_searcher.run(search_task)

    print("\n搜索结果:")
    print("-" * 70)
    print(search_results)
    print("-" * 70)

    # 步骤2: 生成报告
    print("\n【步骤4】Agent2 生成报告...")
    report_task = f"""

根据以下GitHub搜索结果, 生成一份Markdown格式的研究报告:

{search_results}

```

报告要求:

1. 标题: # AI Agent框架研究报告
2. 简介: 说明这是关于AI Agent的GitHub项目调研
3. 主要发现: 列出找到的项目及其特点 (包括名称、描述等)
4. 总结: 总结这些项目的共同特点

请直接输出完整的Markdown格式报告。

```

"""
report_content = document_writer.run(report_task)

print("\n报告内容:")
print("=" * 70)
print(report_content)
print("=" * 70)

# 步骤3: 保存报告
print("\n【步骤5】保存报告到文件....")
import os
try:
    with open("report.md", "w", encoding="utf-8") as f:
        f.write(report_content)
    print("✅ 报告已保存到 report.md")

    # 验证文件
    file_size = os.path.getsize("report.md")
    print(f"✅ 文件大小: {file_size} 字节")
except Exception as e:
    print(f"❌ 保存失败: {e}")

print("\n" + "="*70)
print("任务完成!")
print("="*70)

except Exception as e:
    print(f"\n❌ 错误: {e}")
    import traceback
    traceback.print_exc()

```

github_searcher会在这个过程中调用gh_search_repositories搜索 GitHub 项目。得到的结果会返回给document_writer当做输入, 进一步指导报告的生成, 最后保存报告到 report.md。

10.2.5 MCP 社区生态

MCP 协议的一个巨大优势是丰富的社区生态。Anthropic 和社区开发者已经创建了大量现成的 MCP 服务器, 涵盖文件系统、数据库、API 服务等各种场景。这意味着你不需要从零开始编写工具适配器, 可以直接使用这些经过验证的服务器。

这里给出 MCP 社区的三个资源库:

1. **Awesome MCP Servers** (<https://github.com/punkpeye/awesome-mcp-servers>)
 - 社区维护的 MCP 服务器精选列表
 - 包含各种第三方服务器
 - 按功能分类, 易于查找
2. **MCP Servers Website** (<https://mcpservers.org/>)
 - 官方 MCP 服务器目录网站
 - 提供搜索和筛选功能
 - 包含使用说明和示例
3. **Official MCP Servers** (<https://github.com/modelcontextprotocol/servers>)

- Anthropic 官方维护的服务器
- 质量最高、文档最完善
- 包含常用服务的实现

表 10.5 和 10.6 给出常用的官方 MCP 服务器和社区热门 MCP 服务器：

表 10.5 常用官方 MCP 服务器



表 10.6 社区热门 MCP 服务器



以下是一些特别有趣的案例 TODO 可供参考：

1. 自动化网页测试 (Playwright)

```
# Agent可以自动:
# - 打开浏览器访问网站
# - 填写表单并提交
# - 截图验证结果
# - 生成测试报告
playwright_tool = MCPTool(
    name="playwright",
    server_command=["npx", "-y", "@playwright/mcp"]
)
```

2. 智能笔记助手 (Obsidian + Perplexity)

```
# Agent可以:
# - 搜索最新技术资讯 (Perplexity)
# - 整理成结构化笔记
# - 保存到Obsidian知识库
# - 自动建立笔记间的链接
```

3. 项目管理自动化 (Jira + GitHub)

```
# Agent可以:
# - 从GitHub Issue创建Jira任务
# - 同步代码提交到Jira
# - 自动更新Sprint进度
# - 生成项目报告
```

4. 内容创作工作流 (YouTube + Notion + Spotify)

```
# Agent可以:
# - 获取YouTube视频字幕
# - 生成内容摘要
# - 保存到Notion数据库
# - 播放背景音乐 (Spotify)
```

通过这一节内容的讲解，希望你能探索更多 MCP 的实现案例，也欢迎投稿至 Helloagents！接下来，让我们学习 A2A 协议。

10.3 A2A 协议实战

A2A (Agent-to-Agent) 是一种支持智能体之间直接通信与协作的协议。

10.3.1 协议设计动机

MCP 协议解决了智能体与工具的交互，而 A2A 协议则解决智能体之间的协作问题。在一个需要多智能体（如研究员、撰写员、编辑）协作的任务中，它们需要通信、委托任务、协商能力和同步状态。

传统的中央协调器（星型拓扑）方案存在三个主要问题：

- 单点故障：协调器失效导致系统整体瘫痪。
- 性能瓶颈：所有通信都经过中心节点，限制了并发。
- 扩展困难：增加或修改智能体需要改动中心逻辑。

A2A 协议采用点对点 (P2P) 架构（网状拓扑），允许智能体直接通信，从根本上解决了上述问题。它的核心是任务 (Task) 和工件 (Artifact) 这两个抽象概念，这是它与 MCP 最大的区别，如表 10.7 所示。

表 10.7 A2A 核心概念



为实现对协作过程的管理，A2A 为任务定义了标准化的生命周期，包括创建、协商、代理、执行中、完成、失败等状态，可见图 10.7。



图 10.7 A2A 任务周期

该机制使智能体可以进行任务协商、进度跟踪和异常处理。

A2A 请求生命周期是一个序列，详细说明了请求遵循的四个主要步骤：代理发现、身份验证、发送消息 API 和发送消息流 API。下图 10.8 借鉴了官网的流程图，用来展示了操作流程，说明了客户端、A2A 服务器和身份验证服务器之间的交互。



图 10.8 A2A 请求生命周期

10.3.2 使用 A2A 协议实战

A2A 现有实现大部分为 Sample Code，并且即使有 Python 的实现也较为繁琐，因此这里我们只采用模拟协议思想的方式，通过 A2A-SDK 来继承部分功能实现。

(2) 创建简单的 A2A 智能体

让我们创建一个 A2A 的智能体，同样是计算器案例作为演示：

```
from hello_agents.protocols.a2a.implementation import A2AServer, A2A_AVAILABLE

def create_calculator_agent():
    """创建一个计算器智能体"""
    if not A2A_AVAILABLE:
        print("❌ A2A SDK 未安装，请运行: pip install a2a-sdk")
        return None

    print("✅ 创建计算器智能体")

    # 创建 A2A 服务器
    calculator = A2AServer(
        name="calculator-agent",
        description="专业的数学计算智能体",
        version="1.0.0",
        capabilities={
            "math": ["addition", "subtraction", "multiplication", "division"],
            "advanced": ["power", "sqrt", "factorial"]
        }
    )

    # 添加基础计算技能
    @calculator.skill("add")
    def add_numbers(query: str) -> str:
        """加法计算"""
        try:
            # 简单解析 "计算 5 + 3" 格式
            parts = query.replace("计算", "").replace("加", "+").replace("加上", "+")
            if "+" in parts:
                numbers = [float(x.strip()) for x in parts.split("+")]
                result = sum(numbers)
                return f"计算结果: {sum(numbers)} = {result}"
            else:
                return "请使用格式: 计算 5 + 3"
        except Exception as e:
            return f"计算错误: {e}"

    @calculator.skill("multiply")
    def multiply_numbers(query: str) -> str:
        """乘法计算"""
        try:
            parts = query.replace("计算", "").replace("乘以", "*").replace("×", "*")
            if "*" in parts:
                numbers = [float(x.strip()) for x in parts.split("*")]
                result = 1
                for num in numbers:
                    result *= num
                return f"计算结果: {result} = {result}"
            else:
                return "请使用格式: 计算 5 * 3"
        except Exception as e:
            return f"计算错误: {e}"

    @calculator.skill("info")
    def get_info(query: str) -> str:
        """获取智能体信息"""
        return f"我是 {calculator.name}, 可以进行基础数学计算。支持的技能: {list(calculator.skills())}"

    print(f"✅ 计算器智能体创建成功, 支持技能: {list(calculator.skills())}")
    return calculator

# 创建智能体
calc_agent = create_calculator_agent()
if calc_agent:
```

```
# 测试技能
print("\n    测试智能体技能:")
test_queries = [
    "获取信息",
    "计算 10 + 5",
    "计算 6 * 7"
]

for query in test_queries:
    if "信息" in query:
        result = calc_agent.skills["info"](query)
    elif "+" in query:
        result = calc_agent.skills["add"](query)
    elif "*" in query or "x" in query:
        result = calc_agent.skills["multiply"](query)
    else:
        result = "未知查询类型"

    print(f"    📈 查询: {query}")
    print(f"    🎨 回复: {result}")
    print()
```

(2) 自定义 A2A 智能体

你也可以创建自己的 A2A 智能体，这里只是进行简单演示：

```
from hello_agents.protocols.a2a.implementation import A2AServer, A2A_AVAILABLE

def create_custom_agent():
    """创建自定义智能体"""
    if not A2A_AVAILABLE:
        print("请先安装 A2A SDK: pip install a2a-sdk")
        return None

    # 创建智能体
    agent = A2AServer(
        name="my-custom-agent",
        description="我的自定义智能体",
        capabilities={"custom": ["skill1", "skill2"]}
    )

    # 添加技能
    @agent.skill("greet")
    def greet_user(name: str) -> str:
        """问候用户"""
        return f"你好, {name}! 我是自定义智能体。"

    @agent.skill("calculate")
    def simple_calculate(expression: str) -> str:
        """简单计算"""
        try:
            # 安全的计算 (仅支持基本运算)
            allowed_chars = set('0123456789+-*/(). ')
            if all(c in allowed_chars for c in expression):
                result = eval(expression)
                return f"计算结果: {expression} = {result}"
            else:
                return "错误: 只支持基本数学运算"
        except Exception as e:
            return f"计算错误: {e}"

    return agent

# 创建并测试自定义智能体
custom_agent = create_custom_agent()
if custom_agent:
    # 测试技能
    print("测试问候技能:")
    result1 = custom_agent.skills["greet"]("张三")
    print(result1)

    print("\n测试计算技能:")
    result2 = custom_agent.skills["calculate"]("10 + 5 * 2")
    print(result2)
```

10.3.3 使用 HelloAgents A2A 工具

HelloAgents 提供了统一的 A2A 工具接口。

(1) 创建 A2A Agent 服务端

首先，让我们创建一个 Agent 服务端：

```
from hello_agents.protocols import A2AServer
import threading
import time

# 创建研究员Agent服务
file:///Users/knight/workspace/hello-agents/docs/Hello-Agents-Full.html
```

2/9/26, 1:56 PM

```
researcher = A2AServer(
    name="researcher",
    description="负责搜索和分析资料的Agent",
    version="1.0.0"
)

# 定义技能
@researcher.skill("research")
def handle_research(text: str) -> str:
    """处理研究请求"""
    import re
    match = re.search(r'research\s+(.+)', text, re.IGNORECASE)
    topic = match.group(1).strip() if match else text

    # 实际的研究逻辑 (这里简化)
    result = {
        "topic": topic,
        "findings": f"关于{topic}的研究结果...",
        "sources": ["来源1", "来源2", "来源3"]
    }
    return str(result)

# 在后台启动服务
def start_server():
    researcher.run(host="localhost", port=5000)

if __name__ == "__main__":
    server_thread = threading.Thread(target=start_server, daemon=True)
    server_thread.start()

    print("✅ 研究员Agent服务已启动在 http://localhost:5000")

    # 保持程序运行
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print("\n服务已停止")
```

(2) 创建 A2A Agent 客户端

现在，让我们创建一个客户端来与服务端通信：

```
from hello_agents.protocols import A2AClient

# 创建客户端连接到研究员Agent
client = A2AClient("http://localhost:5000")

# 发送研究请求
response = client.execute_skill("research", "research AI在医疗领域的应用")
print(f"收到响应: {response.get('result')}")

# 输出:
# 收到响应: {'topic': 'AI在医疗领域的应用', 'findings': '关于AI在医疗领域的应用的研究结果...', 'sources': ['来源1', '来源2', '来源3']}
```

(3) 创建 Agent 网络

对于多个 Agent 的协作，我们可以让多个 Agent 相互连接：

```
from hello_agents.protocols import A2AServer, A2AClient
import threading
import time

# 1. 创建多个Agent服务
researcher = A2AServer(
    name="researcher",
    description="研究员"
)

@researcher.skill("research")
def do_research(text: str) -> str:
    import re
    match = re.search(r'research\s+(.+)', text, re.IGNORECASE)
    topic = match.group(1).strip() if match else text
    return str({"topic": topic, "findings": f"{topic}的研究结果"})

writer = A2AServer(
    name="writer",
    description="撰写员"
)

@writer.skill("write")
def write_article(text: str) -> str:
    import re
    match = re.search(r'write\s+(.+)', text, re.IGNORECASE)
    content = match.group(1).strip() if match else text

    # 尝试解析研究数据
    try:
        data = eval(content)
        topic = data.get("topic", "未知主题")
```

2/9/26, 1:56 PM

```
findings = data.get("findings", "无研究结果")
except:
    topic = "未知主题"
    findings = content

return f"# {topic}\n\n基于研究: {findings}\n\n文章内容..."

editor = A2AServer(
    name="editor",
    description="编辑"
)

@editor.skill("edit")
def edit_article(text: str) -> str:
    import re
    match = re.search(r'edit\s+(.+)', text, re.IGNORECASE)
    article = match.group(1).strip() if match else text

    result = {
        "article": article + "\n\n[已编辑优化]",
        "feedback": "文章质量良好",
        "approved": True
    }
    return str(result)

# 2. 启动所有服务
threading.Thread(target=lambda: researcher.run(port=5000), daemon=True).start()
threading.Thread(target=lambda: writer.run(port=5001), daemon=True).start()
threading.Thread(target=lambda: editor.run(port=5002), daemon=True).start()
time.sleep(2) # 等待服务启动

# 3. 创建客户端连接到各个Agent
researcher_client = A2AClient("http://localhost:5000")
writer_client = A2AClient("http://localhost:5001")
editor_client = A2AClient("http://localhost:5002")

# 4. 协作流程
def create_content(topic):
    # 步骤1: 研究
    research = researcher_client.execute_skill("research", f"research {topic}")
    research_data = research.get('result', '')

    # 步骤2: 撰写
    article = writer_client.execute_skill("write", f"write {research_data}")
    article_content = article.get('result', '')

    # 步骤3: 编辑
    final = editor_client.execute_skill("edit", f"edit {article_content}")
    return final.get('result', '')

# 使用
result = create_content("AI在医疗领域的应用")
print(f"\n最终结果: \n{result}")
```

10.3.4 在智能体中使用 A2A 工具

现在让我们看看如何将 A2A 集成到 HelloAgents 的智能体中。

(1) 使用 A2ATool 包装器

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import A2ATool
from dotenv import load_dotenv

load_dotenv()
llm = HelloAgentsLLM()

# 假设已经有一个研究员Agent服务运行在 http://localhost:5000

# 创建协调者Agent
coordinator = SimpleAgent(name="协调者", llm=llm)

# 添加A2A工具，连接到研究员Agent
researcher_tool = A2ATool(
    name="researcher",
    description="研究员Agent，可以搜索和分析资料",
    agent_url="http://localhost:5000"
)
coordinator.add_tool(researcher_tool)

# 协调者可以调用研究员Agent
response = coordinator.run("请让研究员帮我研究AI在教育领域的应用")
print(response)
```

(2) 实战案例：智能客服系统

让我们构建一个完整的智能客服系统，包含三个 Agent： - 接待员：分析客户问题类型 - 技术专家：回答技术问题 - 销售顾问：回答销售问题

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import A2ATool
file:///Users/knight/workspace/hello-agents/docs/Hello-Agents-Full.html
```

2/9/26, 1:56 PM

```
from hello_agents.protocols import A2AServer
import threading
import time
from dotenv import load_dotenv

load_dotenv()
llm = HelloAgentsLLM()

# 1. 创建技术专家Agent服务
tech_expert = A2AServer(
    name="tech_expert",
    description="技术专家, 回答技术问题"
)

@tech_expert.skill("answer")
def answer_tech_question(text: str) -> str:
    import re
    match = re.search(r'answer\s+(.+)', text, re.IGNORECASE)
    question = match.group(1).strip() if match else text
    # 实际应用中, 这里会调用LLM或知识库
    return f"技术回答: 关于'{question}', 我建议您查看我们的技术文档..."

# 2. 创建销售顾问Agent服务
sales_advisor = A2AServer(
    name="sales_advisor",
    description="销售顾问, 回答销售问题"
)

@sales_advisor.skill("answer")
def answer_sales_question(text: str) -> str:
    import re
    match = re.search(r'answer\s+(.+)', text, re.IGNORECASE)
    question = match.group(1).strip() if match else text
    return f"销售回答: 关于'{question}', 我们有特别优惠..."

# 3. 启动服务
threading.Thread(target=lambda: tech_expert.run(port=6000), daemon=True).start()
threading.Thread(target=lambda: sales_advisor.run(port=6001), daemon=True).start()
time.sleep(2)

# 4. 创建接待员Agent (使用HelloAgents的SimpleAgent)
receptionist = SimpleAgent(
    name="接待员",
    llm=llm,
    system_prompt="""你是客服接待员, 负责:
1. 分析客户问题类型 (技术问题 or 销售问题)
2. 将问题转发给相应的专家
3. 整理专家的回答并返回给客户

请保持礼貌和专业。"""
)

# 添加技术专家工具
tech_tool = A2ATool(
    agent_url="http://localhost:6000",
    name="tech_expert",
    description="技术专家, 回答技术相关问题"
)
receptionist.add_tool(tech_tool)

# 添加销售顾问工具
sales_tool = A2ATool(
    agent_url="http://localhost:6001",
    name="sales_advisor",
    description="销售顾问, 回答价格、购买相关问题"
)
receptionist.add_tool(sales_tool)

# 5. 处理客户咨询
def handle_customer_query(query):
    print(f"\n客户咨询: {query}")
    print("=" * 50)
    response = receptionist.run(query)
    print(f"\n客服回复: {response}")
    print("=" * 50)

# 测试不同类型的问题
if __name__ == "__main__":
    handle_customer_query("你们的API如何调用? ")
    handle_customer_query("企业版的价格是多少? ")
    handle_customer_query("如何集成到我的Python项目中? ")
```

(3) 高级用法: Agent 间协商

A2A 协议还支持 Agent 间的协商机制:

```
from hello_agents.protocols import A2AServer, A2AClient
import threading
import time

# 创建两个需要协商的Agent
```

2/9/26, 1:56 PM

```
agent1 = A2AServer(
    name="agent1",
    description="Agent 1"
)

@agent1.skill("propose")
def handle_proposal(text: str) -> str:
    """处理协商提案"""
    import re

    # 解析提案
    match = re.search(r'propose\s+(.+)', text, re.IGNORECASE)
    proposal_str = match.group(1).strip() if match else text

    try:
        proposal = eval(proposal_str)
        task = proposal.get("task")
        deadline = proposal.get("deadline")

        # 评估提案
        if deadline >= 7: # 至少需要7天
            result = {"accepted": True, "message": "接受提案"}
        else:
            result = {
                "accepted": False,
                "message": "时间太紧",
                "counter_proposal": {"deadline": 7}
            }
        return str(result)
    except:
        return str({"accepted": False, "message": "无效的提案格式"})

agent2 = A2AServer(
    name="agent2",
    description="Agent 2"
)

@agent2.skill("negotiate")
def negotiate_task(text: str) -> str:
    """发起协商"""
    import re

    # 解析任务和截止日期
    match = re.search(r'negotiate\s+task:(.+?)\s+deadline:(\d+)', text, re.IGNORECASE)
    if match:
        task = match.group(1).strip()
        deadline = int(match.group(2))

        # 向agent1发送提案
        proposal = {"task": task, "deadline": deadline}
        return str({"status": "negotiating", "proposal": proposal})
    else:
        return str({"status": "error", "message": "无效的协商请求"})

# 启动服务
threading.Thread(target=lambda: agent1.run(port=7000), daemon=True).start()
threading.Thread(target=lambda: agent2.run(port=7001), daemon=True).start()
```

10.4 ANP 协议实战

在 MCP 协议解决了工具调用、A2A 协议解决点对点智能体协作之后，ANP 协议则专注于解决大规模、开放网络环境下的智能体管理问题。

在 10.2 和 10.3 节中，我们学习了 MCP（工具访问）和 A2A（智能体协作）。现在，让我们学习 ANP（Agent Network Protocol）协议，它专注于构建大规模、开放的智能体网络。

10.4.1 协议目标

当一个网络中存在大量功能各异的智能体（例如，自然语言处理、图像识别、数据分析等）时，系统会面临一系列挑战：

- 服务发现：当新任务到达时，如何快速找到能够处理该任务的智能体？
- 智能路由：如果多个智能体都能处理同一任务，如何选择最合适的一个（如根据负载、成本等）并向其分派任务？
- 动态扩展：如何让新加入网络的智能体被其他成员发现和调用？

ANP 的设计目标就是提供一套标准化的机制，来解决上述的服务发现、路由选择和网络扩展性问题。

为实现其设计目标，ANP 定义了以下几个核心概念，如表 10.8 所示：

表 10.8 ANP 核心概念



我们同样借用官方的[入门指南](#)来介绍 ANP 的架构设计，如图 10.9 所示



图 10.9 ANP 整体流程

在这个流程图里，主要包括以下几个步骤：

1. 服务的发现与匹配：首先，智能体 A 通过一个公开的发现服务，基于语义或功能描述进行查询，以定位到符合其任务需求的智能体 B。该发现服务通过预先爬取各智能体对外暴露的标准端点（`well-known/agent-descriptions`）来建立索引，从而实现服务需求方与提供方的动态匹配。

2. 基于 DID 的身份验证：在交互开始时，智能体 A 使用其私钥对包含自身 DID 的请求进行签名。智能体 B 收到后，通过解析该 DID 获取对应的公钥，并以此验证签名的真实性与请求的完整性，从而建立起双方的可信通信。

3. 标准化的服务执行：身份验证通过后，智能体 B 响应请求，双方依据预定义的标准接口和数据格式进行数据交换或服务调用（如预订、查询等）。标准化的交互流程是实现跨平台、跨系统互操作性的基础。

总而言之，该机制的核心是利用 DID 构建了一个去中心化的信任根基，并借助标准化的描述协议实现了服务的动态发现。这套方法使得智能体能够在无需中央协调的前提下，安全、高效地在互联网上形成协作网络。

10.4.2 使用 ANP 服务发现

(1) 创建服务发现中心

```
from hello_agents.protocols import ANPDiscovery, register_service

# 创建服务发现中心
discovery = ANPDiscovery()

# 注册Agent服务
register_service(
    discovery=discovery,
    service_id="nlp_agent_1",
    service_name="NLP处理专家A",
    service_type="nlp",
    capabilities=["text_analysis", "sentiment_analysis", "ner"],
    endpoint="http://localhost:8001",
    metadata={"load": 0.3, "price": 0.01, "version": "1.0.0"}
)

register_service(
    discovery=discovery,
    service_id="nlp_agent_2",
    service_name="NLP处理专家B",
    service_type="nlp",
    capabilities=["text_analysis", "translation"],
    endpoint="http://localhost:8002",
    metadata={"load": 0.7, "price": 0.02, "version": "1.1.0"}
)

print("✅ 服务注册完成")
```

(2) 发现服务

```
from hello_agents.protocols import discover_service

# 按类型查找
nlp_services = discover_service(discovery, service_type="nlp")
print(f"找到 {len(nlp_services)} 个NLP服务")

# 选择负载最低的服务
best_service = min(nlp_services, key=lambda s: s.metadata.get("load", 1.0))
print(f"最佳服务: {best_service.service_name} (负载: {best_service.metadata['load']})")
```

(3) 构建 Agent 网络

```
from hello_agents.protocols import ANPNetwork

# 创建网络
network = ANPNetwork(network_id="ai_cluster")

# 添加节点
for service in discovery.list_all_services():
    network.add_node(service.service_id, service.endpoint)

# 建立连接 (根据能力匹配)
network.connect_nodes("nlp_agent_1", "nlp_agent_2")

stats = network.get_network_stats()
print(f"✅ 网络构建完成, 共 {stats['total_nodes']} 个节点")
```

10.4.3 实战案例

让我们构建一个完整的分布式任务调度系统：

```
from hello_agents.protocols import ANPDiscovery, register_service
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools.builtin import ANPTool
import random
```

2/9/26, 1:56 PM

```
from dotenv import load_dotenv
load_dotenv()
llm = HelloAgentsLLM()

# 1. 创建服务发现中心
discovery = ANPDiscovery()

# 2. 注册多个计算节点
for i in range(10):
    register_service(
        discovery=discovery,
        service_id=f"compute_node_{i}",
        service_name=f"计算节点{i}",
        service_type="compute",
        capabilities=["data_processing", "ml_training"],
        endpoint="http://node{i}:8000",
        metadata={
            "Load": random.uniform(0.1, 0.9),
            "cpu_cores": random.choice([4, 8, 16]),
            "memory_gb": random.choice([16, 32, 64]),
            "gpu": random.choice([True, False])
        }
    )
)
```

```
print(f"✅ 注册了 {len(discovery.list_all_services())} 个计算节点")
```

```
# 3. 创建任务调度Agent
scheduler = SimpleAgent(
    name="任务调度器",
    llm=llm,
    system_prompt="""你是一个智能任务调度器，负责：
1. 分析任务需求
2. 选择最合适的计算节点
3. 分配任务
选择节点时考虑：负载、CPU核心数、内存、GPU等因素。"""
)
```

```
# 添加ANP工具
anp_tool = ANPTool(
    name="service_discovery",
    description="服务发现工具，可以查找和选择计算节点",
    discovery=discovery
)
scheduler.add_tool(anp_tool)
```

```
# 4. 智能任务分配
def assign_task(task_description):
    print(f"\n任务: {task_description}")
    print("=" * 50)

    # 让Agent智能选择节点
    response = scheduler.run(f"""
请为以下任务选择最合适的计算节点:
{task_description}
""")
```

要求：
1. 列出所有可用节点
2. 分析每个节点的特点
3. 选择最合适的选择
4. 说明选择理由
""")

```
print(response)
print("=" * 50)
```

```
# 测试不同类型的任务
assign_task("训练一个大型深度学习模型，需要GPU支持")
assign_task("处理大量文本数据，需要高内存")
assign_task("运行轻量级数据分析任务")
```

这是一个负载均衡示例

```
from hello_agents.protocols import ANPDiscovery, register_service
import random

# 创建服务发现中心
discovery = ANPDiscovery()

# 注册多个相同类型的服务
for i in range(5):
    register_service(
        discovery=discovery,
        service_id=f"api_server_{i}",
        service_name=f"API服务器{i}",
        service_type="api",
        capabilities=["rest_api"],
        endpoint="http://api{i}:8000",
        metadata={"load": random.uniform(0.1, 0.9)}
    )
```

```

# 负载均衡函数
def get_best_server():
    """选择负载最低的服务器"""
    servers = discovery.discover_services(service_type="api")
    if not servers:
        return None

    best = min(servers, key=lambda s: s.metadata.get("load", 1.0))
    return best

# 模拟请求分配
for i in range(10):
    server = get_best_server()
    print(f"请求 {i+1} -> {server.service_name} (负载: {server.metadata['load']:.2f})")

    # 更新负载 (模拟)
    server.metadata["load"] += 0.1

```

10.5 构建自定义 MCP 服务器

在前面的章节中，我们学习了如何使用现有的 MCP 服务。并且也了解到了不同协议的特点。现在，让我们学习如何构建自己的 MCP 服务器。

10.5.1 创建你的第一个 MCP 服务器

(1) 为什么要构建自定义 MCP 服务器?

虽然可以直接使用公开的 MCP 服务，但在许多实际应用场景中，需要构建自定义的 MCP 服务器以满足特定需求。

主要动机包括以下几点：

- 封装业务逻辑：将企业内部特有的业务流程或复杂操作封装为标准化的 MCP 工具，供智能体统一调用。
- 访问私有数据：创建一个安全可控的接口或代理，用于访问内部数据库、API 或其他无法对公网暴露的私有数据源。
- 性能专项优化：针对高频调用或对响应延迟有严苛要求的应用场景，进行深度优化。
- 功能定制扩展：实现标准 MCP 服务未提供的特定功能，例如集成专有算法模型或连接特定的硬件设备。

(2) 教学案例：天气查询 MCP 服务器

让我们从一个简单的天气查询服务器开始，逐步学习 MCP 服务器开发：

```

#!/usr/bin/env python3
"""天气查询 MCP 服务器"""

import json
import requests
import os
from datetime import datetime
from typing import Dict, Any
from hello_agents.protocols import MCPServer

# 创建 MCP 服务器
weather_server = MCPServer(name="weather-server", description="真实天气查询服务")

CITY_MAP = {
    "北京": "Beijing", "上海": "Shanghai", "广州": "Guangzhou",
    "深圳": "Shenzhen", "杭州": "Hangzhou", "成都": "Chengdu",
    "重庆": "Chongqing", "武汉": "Wuhan", "西安": "Xi'an",
    "南京": "Nanjing", "天津": "Tianjin", "苏州": "Suzhou"
}

def get_weather_data(city: str) -> Dict[str, Any]:
    """从 wttr.in 获取天气数据"""
    city_en = CITY_MAP.get(city, city)
    url = f"https://wttr.in/{city_en}?format=j1"
    response = requests.get(url, timeout=10)
    response.raise_for_status()
    data = response.json()
    current = data["current_condition"][0]

    return {
        "city": city,
        "temperature": float(current["temp_C"]),
        "feels_like": float(current["FeelsLikeC"]),
        "humidity": int(current["humidity"]),
        "condition": current["weatherDesc"][0]["value"],
        "wind_speed": round(float(current["windspeedKmph"]) / 3.6, 1),
        "visibility": float(current["visibility"]),
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    }

# 定义工具函数
def get_weather(city: str) -> str:
    """获取指定城市的当前天气"""
    try:

```

2/9/26, 1:56 PM

```
weather_data = get_weather_data(city)
return json.dumps(weather_data, ensure_ascii=False, indent=2)
except Exception as e:
    return json.dumps({"error": str(e), "city": city}, ensure_ascii=False)

def list_supported_cities() -> str:
    """列出所有支持的中文城市"""
    result = {"cities": list(CITY_MAP.keys()), "count": len(CITY_MAP)}
    return json.dumps(result, ensure_ascii=False, indent=2)

def get_server_info() -> str:
    """获取服务器信息"""
    info = {
        "name": "Weather MCP Server",
        "version": "1.0.0",
        "tools": ["get_weather", "list_supported_cities", "get_server_info"]
    }
    return json.dumps(info, ensure_ascii=False, indent=2)

# 注册工具到服务器
weather_server.add_tool(get_weather)
weather_server.add_tool(list_supported_cities)
weather_server.add_tool(get_server_info)

if __name__ == "__main__":
    weather_server.run()
```

(3) 测试自定义 MCP 服务器

然后创建测试脚本:

```
#!/usr/bin/env python3
"""测试天气查询 MCP 服务器"""

import asyncio
import json
import sys
import os

sys.path.insert(0, os.path.join(os.path.dirname(__file__), '..', 'HelloAgents'))
from hello_agents.protocols.mcp.client import MCPClient

async def test_weather_server():
    server_script = os.path.join(os.path.dirname(__file__), "14_weather_mcp_server.py")
    client = MCPClient(["python", server_script])

    try:
        async with client:
            # 测试1: 获取服务器信息
            info = json.loads(await client.call_tool("get_server_info", {}))
            print(f"服务器: {info['name']} v{info['version']}")

            # 测试2: 列出支持的城市
            cities = json.loads(await client.call_tool("list_supported_cities", {}))
            print(f"支持城市: {cities['count']} 个")

            # 测试3: 查询北京天气
            weather = json.loads(await client.call_tool("get_weather", {"city": "北京"}))
            if "error" not in weather:
                print(f"\n北京天气: {weather['temperature']}°C, {weather['condition']}")

            # 测试4: 查询深圳天气
            weather = json.loads(await client.call_tool("get_weather", {"city": "深圳"}))
            if "error" not in weather:
                print(f"\n深圳天气: {weather['temperature']}°C, {weather['condition']}")

        print("\n✅ 所有测试完成! ")

    except Exception as e:
        print(f"\n❌ 测试失败: {e}")

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

(4) 在 Agent 中使用自定义 MCP 服务器

```
"""在 Agent 中使用天气 MCP 服务器"""

import os
from dotenv import load_dotenv
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import MCPTool

load_dotenv()
```

2/9/26, 1:56 PM

```
def create_weather_assistant():
    """创建天气助手"""
    llm = HelloAgentsLLM()

    assistant = SimpleAgent(
        name="天气助手",
        llm=llm,
        system_prompt="""你是天气助手，可以查询城市天气。
使用 get_weather 工具查询天气，支持中文城市名。
"""
    )

    # 添加天气 MCP 工具
    server_script = os.path.join(os.path.dirname(__file__), "14_weather_mcp_server.py")
    weather_tool = MCPTool(server_command=["python", server_script])
    assistant.add_tool(weather_tool)

    return assistant

def demo():
    """演示"""
    assistant = create_weather_assistant()

    print("\n查询北京天气: ")
    response = assistant.run("北京今天天气怎么样? ")
    print(f"回答: {response}\n")

def interactive():
    """交互模式"""
    assistant = create_weather_assistant()

    while True:
        user_input = input("\n你: ").strip()
        if user_input.lower() in ['quit', 'exit']:
            break
        response = assistant.run(user_input)
        print(f"助手: {response}")

if __name__ == "__main__":
    import sys
    if len(sys.argv) > 1 and sys.argv[1] == "demo":
        demo()
    else:
        interactive()
```

🔗 连接到 MCP 服务器...

✓ 连接成功!

⚡ 连接已断开

✓ 工具 'mcp_get_weather' 已注册。

✓ 工具 'mcp_list_supported_cities' 已注册。

✓ 工具 'mcp_get_server_info' 已注册。

✓ MCP工具 'mcp' 已展开为 3 个独立工具

你: 我想查询北京的天气

🔗 连接到 MCP 服务器...

✓ 连接成功!

⚡ 连接已断开

助手: 当前北京的天气情况如下:

- 温度: 10.0°C
- 体感温度: 9.0°C
- 湿度: 94%
- 天气状况: 小雨
- 风速: 1.7米/秒
- 能见度: 10.0公里
- 时间戳: 2025年10月9日 13:46:40

请注意携带雨具，并根据天气变化适当调整着装。

10.5.2 上传 MCP 服务器

我们创建了一个真实的天气查询 MCP 服务器。现在，让我们将它发布到 Smithery 平台，让全世界的开发者都能使用我们的服务。

(1) 什么是 Smithery?

[Smithery](#) 是 MCP 服务器的官方发布平台，类似于 Python 的 PyPI 或 Node.js 的 npm。通过 Smithery，用户可以：

- 🔎 发现和搜索 MCP 服务器
- 📦 一键安装 MCP 服务器
- 📊 查看服务器的使用统计和评价
- 🔄 自动获取服务器更新

(2) 准备发布 首先，我们需要将项目整理成标准的发布格式，这个文件夹已经在code目录下整理好，可供大家参考：

```
weather-mcp-server/
├── README.md          # 项目说明文档
├── LICENSE            # 开源许可证
├── Dockerfile          # Docker 构建配置（推荐）
├── pyproject.toml      # Python 项目配置（必需）
├── requirements.txt    # Python 依赖
└── smithery.yaml       # Smithery 配置文件（必需）
└── server.py           # MCP 服务器主文件
```

需要注意的是，smithery.yaml是Smithery平台的配置文件：

```
name: weather-mcp-server
displayName: Weather MCP Server
description: Real-time weather query MCP server based on HelloAgents framework
version: 1.0.0
author: HelloAgents Team
homepage: https://github.com/yourusername/weather-mcp-server
license: MIT
categories:
  - weather
  - data
tags:
  - weather
  - real-time
  - helloagents
  - wttr
runtime: container
build:
  dockerfile: Dockerfile
  dockerBuildPath: .
startCommand:
  type: http
tools:
  - name: get_weather
    description: Get current weather for a city
  - name: list_supported_cities
    description: List all supported cities
  - name: get_server_info
    description: Get server information
```

配置说明：

- name: 服务器的唯一标识符（小写，用连字符分隔）
- displayName: 显示名称
- description: 简短描述
- version: 版本号（遵循语义化版本）
- runtime: 运行时环境（python/node）
- entrypoint: 入口文件
- tools: 工具列表

pyproject.toml是Python项目的标准配置文件，Smithery要求必须包含此文件，因为后续会打包成一个server：

```
[build-system]
requires = ["setuptools>=61.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "weather-mcp-server"
version = "1.0.0"
description = "Real-time weather query MCP server based on HelloAgents framework"
readme = "README.md"
license = {text = "MIT"}
authors = [
    {name = "HelloAgents Team", email = "xxx"}
]
requires-python = ">=3.10"
dependencies = [
    "hello-agents>=0.2.1",
    "requests>=2.31.0",
]

[project.urls]
Homepage = "https://github.com/yourusername/weather-mcp-server"
Repository = "https://github.com/yourusername/weather-mcp-server"
"Bug Tracker" = "https://github.com/yourusername/weather-mcp-server/issues"

[tool.setuptools]
py-modules = ['server']
```

配置说明：

- [build-system]: 指定构建工具（setuptools）
- [project]: 项目元数据
 - name: 项目名称
 - version: 版本号（遵循语义化版本）
 - dependencies: 项目依赖列表

- requires-python: Python 版本要求
- [project.urls]: 项目相关链接
- [tool.setuptools]: setuptools 配置

虽然 Smithery 会自动生成 Dockerfile，但提供自定义 Dockerfile 可以确保部署成功：

```
# Multi-stage build for weather-mcp-server
FROM python:3.12-slim-bookworm as base

# Set working directory
WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

# Copy project files
COPY pyproject.toml requirements.txt ./
COPY server.py ./

# Install Python dependencies
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt

# Set environment variables
ENV PYTHONUNBUFFERED=1
ENV PORT=8081

# Expose port (Smithery uses 8081)
EXPOSE 8081

# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
    CMD python -c "import sys; sys.exit(0)"

# Run the MCP server
CMD ["python", "server.py"]
```

Dockerfile 配置说明：

- 基础镜像: python:3.12-slim-bookworm - 轻量级 Python 镜像
- 工作目录: /app - 应用程序根目录
- 端口: 8081 - Smithery 平台标准端口
- 启动命令: python server.py - 运行 MCP 服务器

在这里，我们需要 Fork `hello-agents` 仓库，得到 code 中的源码，并使用自己的 GitHub 创建一个名为 `weather-mcp-server` 的仓库，将 `yourusername` 改为自己 GitHub 的 Username。

(3) 提交到 Smithery

打开浏览器，访问 <https://smithery.ai/>。使用 GitHub 账号登录 Smithery。点击页面上的“Publish Server”按钮，输入你的 GitHub 仓库 URL: `https://github.com/yourusername/weather-mcp-server`，即可等待发布。

一旦发布完成，可以看到类似这样的页面，如图 10.10 所示：



图 10.10 Smithery 发布成功页面

一旦服务器发布成功，用户可以通过以下方式使用：

方式 1：通过 Smithery CLI

```
# 安装 Smithery CLI
npm install -g @smithery/cli

# 安装你的服务器
smithery install weather-mcp-server
```

方式 2：在 Claude Desktop 中配置

```
{
  "mcpServers": {
    "weather": {
      "command": "smithery",
      "args": ["run", "weather-mcp-server"]
    }
  }
}
```

方式 3：在 HelloAgents 中使用

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools.builtin.protocol_tools import MCPTool
```

```

agent = SimpleAgent(name="天气助手", llm=HelloAgentsLLM())

# 使用 Smithery 安装的服务器
weather_tool = MCPTool(
    server_command=["smithery", "run", "weather-mcp-server"]
)
agent.add_tool(weather_tool)

response = agent.run("北京今天天气怎么样? ")

```

当然，这里只是举例，还有更多的用法可以自行探索，下图 10.11 展示了当 MCP 工具发布成功会包含的信息，显示服务的名称“天气”，其唯一标识符 @jyaoao/weather-mcp-server，以及状态信息。Tools 区域就是我们刚刚实现的方法，Connect 区则提供了连接和使用此服务所需的技术信息，包括服务的接入 URL 地址和多种语言/环境下的配置代码片段。如果想要更加深入了解可以点击这个[链接](#)。



图 10.11 Smithery 发布成功的 MCP 工具

现在是时候去创造你的 MCP 服务器了！

10.6 本章总结

本章系统性地介绍了智能体通信的三种核心协议：MCP、A2A 与 ANP，并探讨了它们的设计理念、应用场景与实践方法。

协议定位：

- **MCP (Model Context Protocol)**: 作为智能体与工具之间的桥梁，提供统一的工具访问接口，适用于增强单个智能体的能力。
- **A2A (Agent-to-Agent Protocol)**: 作为智能体之间的对话系统，支持直接通信与任务协商，适用于小规模团队的紧密协作。
- **ANP (Agent Network Protocol)**: 作为智能体的“互联网”，提供服务发现、路由与负载均衡机制，适用于构建大规模、开放的智能体网络。

HelloAgents 的集成方案

在 HelloAgents 框架中，这三种协议被统一抽象为工具（Tool），实现了无缝集成，允许开发者灵活地为智能体添加不同层级的通信能力：

```

# 统一的Tool接口
from hello_agents.tools import MCPTool, A2ATool, ANPTool

# 所有协议都可以作为Tool添加到Agent
agent.add_tool(MCPTool(...))
agent.add_tool(A2ATool(...))
agent.add_tool(ANPTool(...))

```

实战经验总结

- 优先利用成熟的社区 MCP 服务，以减少不必要的重复开发。
- 根据系统规模选择合适的协议：小规模协作场景推荐使用 A2A，大规模网络场景则应采用 ANP。

完成本章后，建议你：

1. 动手实践：
 - 构建自己的 MCP 服务器
 - 利用协议创建多 Agent 协作系统
 - MCP、A2A 与 ANP 的组合应用策略
2. 深入学习：
 - 阅读 MCP 官方文档：<https://modelcontextprotocol.io>
 - 阅读 A2A 官方文档：<https://a2a-protocol.org/latest/>
 - 阅读 ANP 官方文档：<https://agent-network-protocol.com/guide/>
3. 参与社区：
 - 向社区贡献新的 MCP 服务
 - 分享个人开发的智能体实现案例
 - 参与相关协议的技术标准讨论，也可以在 Issue 提问或是直接帮助 Helloagents 支持新的 example 案例

恭喜你完成第十章的学习！

你现在已经掌握了智能体通信协议的核心知识。继续加油！🚀

习题

提示：部分习题没有标准答案，重点在于培养学习者对智能体通信协议的综合理解和实践能力。

1. 本章介绍了三种智能体通信协议：MCP、A2A 和 ANP。请分析：

- 在 10.1.2 节中对比了三种协议的设计理念。请深入分析：为什么 MCP 强调“上下文共享”，A2A 强调“对话式协作”，而 ANP 强调“网络拓扑”？这些设计理念分别解决了什么核心问题？

- 假设你要构建一个“智能客服系统”，需要以下功能：（1）访问客户数据库和订单系统；（2）多个专业客服智能体协作处理复杂问题；（3）支持大规模并发用户请求。请为每个功能选择最合适的协议，并说明理由。
- 三种协议是否可以组合使用？请设计一个实际应用场景，展示如何同时使用 MCP、A2A 和 ANP 来构建一个完整的智能体系统。画出系统架构图并说明各协议的职责。

2. MCP (Model Context Protocol) 是智能体与工具通信的标准协议。基于 10.2 节的内容，请深入思考：

提示：这是一道动手实践题，建议实际操作

- 在 10.2.3 节的 MCP 服务器实现中，我们定义了 `list_tools`、`call_tool` 等核心方法。请扩展这个实现，添加一个新的 MCP 服务器，提供以下工具：（1）数据库查询工具；（2）数据可视化工具；（3）报表生成工具。要求工具之间能够协作完成复杂的数据分析任务。
- MCP 协议支持“资源”（Resources）和“提示”（Prompts）两个重要概念，但本章主要聚焦于“工具”（Tools）。请查阅 MCP 官方文档，了解 Resources 和 Prompts 的设计目的，并设计一个应用场景，展示如何利用这三个核心概念构建更强大的智能体系统。
- MCP 使用 JSON-RPC 2.0 作为底层通信协议，通过 `stdio` 进行进程间通信。请分析：这种设计有什么优势和局限性？如果需要支持远程 MCP 服务器（通过 HTTP/WebSocket 访问），应该如何扩展当前的实现？

3. A2A (Agent-to-Agent Protocol) 支持智能体间的对话式协作。基于 10.3 节的内容，请完成以下扩展实践：

提示：这是一道动手实践题，建议实际操作

- 在 10.3.4 节的“研究团队”案例中，研究员和撰写员通过 A2A 协议协作完成论文写作。请扩展这个案例，添加第三个智能体“审稿人”（Reviewer），它能够评审论文质量并提出修改建议。设计三个智能体之间的协作流程，并实现完整的代码。
- A2A 协议定义了 `task`、`task_result` 等消息类型。请分析：如果协作过程中出现冲突（如两个智能体对同一问题有不同意见），应该如何设计冲突解决机制？请扩展 A2A 协议，添加“协商”（negotiation）和“投票”（voting）等消息类型。
- 对比 A2A 协议与第六章介绍的 AutoGen、CAMEL 等多智能体框架：A2A 作为标准协议，与这些框架的关系是什么？它们能否互相替代？请设计一个方案，让基于 A2A 协议的智能体能够与 AutoGen 框架中的智能体进行通信。

4. ANP (Agent Network Protocol) 支持大规模智能体网络。基于 10.4 节的内容，请深入分析：

- 在 10.4.2 节中介绍了 ANP 的网络拓扑设计，包括星型、网状、分层等结构。请分析：在什么场景下应该选择哪种拓扑结构？如果网络规模从 10 个智能体扩展到 1000 个智能体，拓扑结构应该如何演进？
- ANP 协议支持“路由”（routing）和“发现”（discovery）机制，让智能体能够动态找到合适的协作伙伴。请设计一个“智能路由算法”：根据任务类型、智能体能力、网络负载等因素，自动选择最优的消息路由路径。
- 在 10.4.4 节的“智慧城市”案例中，多个智能体协作管理城市系统。请思考：如果某个关键智能体（如交通管理智能体）出现故障，整个系统应该如何应对？请设计一个“容错机制”，包括故障检测、备份切换、状态恢复等功能。

5. 智能体通信协议的安全性和隐私保护是实际应用中的关键问题。请思考：

- 在 10.2.4 节的 MCP 客户端实现中，智能体可以调用 MCP 服务器提供的任何工具。请分析：这种设计存在什么安全风险？如果 MCP 服务器提供了危险操作（如删除文件、执行系统命令），应该如何设计权限控制机制？
- A2A 和 ANP 协议涉及多个智能体之间的通信，可能包含敏感信息（如用户隐私数据、商业机密）。请设计一个“端到端加密”方案：确保消息在传输过程中不被窃听或篡改，同时支持智能体身份认证和访问控制。
- 在大规模智能体网络中，恶意智能体可能会发送虚假信息、发起拒绝服务攻击或窃取其他智能体的数据。请设计一个“信任评估系统”：根据智能体的历史行为、协作质量、社区评价等因素，动态评估每个智能体的可信度，并据此调整通信策略。

参考文献

- [1] Anthropic. (2024). *Model Context Protocol*. Retrieved October 7, 2025, from <https://modelcontextprotocol.io/>
- [2] The A2A Project. (2025). *A2A Protocol: An open protocol for agent-to-agent communication*. Retrieved October 7, 2025, from <https://a2a-protocol.org/>
- [3] Chang, G., Lin, E., Yuan, C., Cai, R., Chen, B., Xie, X., & Zhang, Y. (2025). *Agent Network Protocol technical white paper*. arXiv. <https://doi.org/10.48550/arXiv.2508.00007>

第十一章 Agentic-RL

11.1 从 LLM 训练到 Agentic RL

在前面的章节中，我们实现了多种智能体范式和通信协议。不过智能体处理更复杂的任务时表现不佳，自然会有疑问：如何让智能体具备更强的推理能力？如何让智能体学会更好地使用工具？如何让智能体能够自我改进？

这正是 Agentic RL（基于强化学习的智能体训练）要解决的核心问题。本章将为 HelloAgents 框架引入强化学习训练能力，让你能够训练出具备推理、工具使用等高级能力的智能体。我们将从 LLM 训练的基础知识开始，逐步深入到监督微调（Supervised Fine-Tuning, SFT）、群组相对策略优化（Group Relative Policy Optimization, GRPO）等实用技术，最终构建一个完整的智能体训练 pipeline。

11.1.1 从强化学习到 Agentic RL

在第二章的 2.4.2 节中，我们介绍了基于强化学习的智能体。强化学习（Reinforcement Learning, RL）是一种专注于解决序贯决策问题的学习范式，它通过智能体与环境的直接交互，在“试错”中学习如何最大化长期收益。

现在，让我们将这个框架应用到 LLM 智能体上。考虑一个数学问题求解智能体，它需要回答这样的问题：

问题：Janet's ducks lay 16 eggs per day. She eats three for breakfast every morning and bakes muffins for her friends every day with four. She sells the remainder at the farmers' market daily for \$2 per fresh duck egg. How much in dollars does she make every day at the farmers' market?

这个问题需要多步推理：首先计算 Janet 每天剩余的鸡蛋数量($16 - 3 - 4 = 9$)，然后计算她的收入($9 \times 2 = 18$)。我们可以将这个任务映射到强化学习框架：

- 智能体：基于 LLM 的推理系统
- 环境：数学问题和验证系统
- 状态：当前的问题描述和已有的推理步骤
- 行动：生成下一步推理或最终答案
- 奖励：答案是否正确（正确+1，错误 0）

传统的监督学习方法存在三个核心局限：一是数据质量完全决定训练质量，模型只能模仿训练数据，难以超越；二是缺乏探索能力，只能被动学习人类提供的路径；三是难以优化长期目标，无法精确优化多步推理的中间过程。

强化学习提供了新的可能性。通过让智能体自主生成多个候选答案并根据正确性获得奖励，它可以学习哪些推理路径更优、哪些步骤是关键，甚至发现比人类标注更好的解题方法^[8]。这就是 Agentic RL 的核心思想：将 LLM 作为可学习策略，嵌入智能体的感知-决策-执行循环，通过强化学习优化多步任务表现。

11.1.2 LLM 训练全景图

在深入 Agentic RL 之前，我们需要先理解 LLM 训练的完整流程。一个强大的 LLM（如 GPT、Claude、Qwen）的诞生，通常要经历两个主要阶段：预训练（Pretraining）和后训练（Post-training）。如图 11.1 所示，这两个阶段构成了 LLM 从“语言模型”到“对话助手”的完整演化路径。



图 11.1 LLM 训练全景图

预训练阶段是 LLM 训练的第一阶段，目标是让模型学习语言的基本规律和世界知识。这个阶段使用海量的文本数据（通常是数 TB 级别），通过自监督学习的方式训练模型。最常见的预训练任务是因果语言建模（Causal Language Modeling），也称为下一个词预测（Next Token Prediction）。

给定一个文本序列 x_1, x_2, \dots, x_t ，模型需要预测下一个词 x_{t+1} ：

```
$$ \text{mathcal}{L}_{\text{pretrain}} = -\sum_{t=1}^T \log P(x_t | x_1, x_2, \dots, x_{t-1}; \theta) $$
```

其中 θ 是模型参数， $P(x_t | x_1, \dots, x_{t-1}; \theta)$ 是模型预测的下一个词的概率分布，目标是最小化负对数似然，即最大化预测正确词的概率。例如，给定文本“*The cat sat on the*”，模型需要预测下一个词最可能是“*mat*”。通过在海量文本上进行这样的训练，模型逐渐学会语法规则（什么样的词序是合法的）、语义知识（词与词之间的关系）、世界知识（关于世界的事实性信息）以及基础的推理能力。

预训练阶段的特点是数据量巨大、计算成本高、学到的是通用的语言理解和生成能力、采用无监督学习。

后训练阶段则是要解决预训练模型的不足。预训练后的模型虽然具备了强大的语言能力，但它只是一个“预测下一个词”的模型，并不知道如何遵循人类的指令、生成有帮助无害诚实的回答、拒绝不当的请求，以及以对话的方式与人交互。后训练阶段就是要解决这些问题，让模型对齐人类的偏好和价值观。

后训练通常包含三个步骤。第一步是监督微调（SFT）^[15]，目标是让模型学会遵循指令和对话格式。训练数据是（prompt, completion）对，训练目标与预训练类似，仍然是最大化正确输出的概率：

```
$$ \text{mathcal}{L}_{\text{SFT}} = -\sum_{i=1}^N \log P(y_i | x_i; \theta) $$
```

其中 x_i 是输入提示（prompt）， y_i 是期望的输出， N 是训练样本数量。SFT 的特点是数据量较小、需要人工标注、快速见效、主要学习任务格式和基本能力。

第二步是奖励建模（RM）。SFT 后的模型虽然能遵循指令，但生成的回答质量参差不齐。我们需要一种方式来评估回答的质量，这就是奖励模型的作用^[13,14]。奖励模型的训练数据是偏好对比数据，包含同一个问题的两个回答，一个更好（chosen），一个更差（rejected）。奖励模型的训练目标是学习人类的偏好：

$$\mathcal{L}_{\text{RM}} = -\mathbb{E}_{(x, y_w, y_p)}[\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_p))]$$

其中 $r_\phi(x, y)$ 是奖励模型，输入是(提示，回答)对，输出是质量分数; y_w 是更好的回答(chosen)， y_p 是更差的回答(rejected)， σ 是 sigmoid 函数，目标是让奖励模型给更好的回答更高的分数。

第三步是强化学习微调。有了奖励模型后，我们就可以用强化学习来优化语言模型，让它生成更高质量的回答。最经典的算法是 PPO(Proximal Policy Optimization)^[11]，训练目标是：

$$J_{\text{PPO}} = \mathbb{E}_{x, y \sim \pi_\theta}[r_\phi(x, y)] - \beta \cdot D_{KL}(\pi_\theta \| \pi_{\text{ref}})$$

其中 π_θ 是当前策略，即语言模型， π_{ref} 是参考策略，这个场景下可以是 SFT 模型， $r_\phi(x, y)$ 是奖励模型的评分， D_{KL} 是 KL 散度，目的是为了防止模型偏离太远， β 是平衡系数。这个目标函数的含义是：最大化奖励，同时不要偏离原始模型太远。

传统的 RLHF(Reinforcement Learning from Human Feedback)^[5]需要大量人工标注偏好数据，成本高昂。为了降低成本，研究者提出了 RLAIF(Reinforcement Learning from AI Feedback)^[7]，用强大的 AI 模型(如 GPT-4)来替代人类标注员。RLAIF 的工作流程是：用 SFT 模型生成多个候选回答，用强大的 AI 模型对回答进行评分和排序，用 AI 的评分训练奖励模型，用奖励模型进行强化学习。实验表明，RLAIF 的效果接近甚至超过 RLHF，同时成本大幅降低^[11]。

11.1.3 Agentic RL 的核心理念

在理解了 LLM 的基础训练流程后，让我们来看看 Agentic RL 与传统训练方法的区别。传统的后训练(我们称之为 PBRFT: Preference-Based Reinforcement Fine-Tuning)主要关注单轮对话的质量优化：给定一个用户问题，模型生成一个回答，然后根据回答的质量获得奖励。这种方式适合优化对话助手，但对于需要多步推理、工具使用、长期规划的智能体任务来说，就显得力不从心了。

Agentic RL 则是一种新的范式，它将 LLM 视为一个可学习的策略，嵌入在一个顺序决策循环中。在这个框架下，智能体需要在动态环境中与外部世界交互，执行多步行动来完成复杂任务，获得中间反馈来指导后续决策，优化长期累积奖励而非单步奖励。

让我们通过一个具体例子来理解这个区别。在 PBRFT 场景中，用户问“请解释什么是强化学习”，模型生成完整回答，然后根据回答质量直接给分。而在 Agentic RL 场景中，用户请求“帮我分析这个 GitHub 仓库的代码质量”，智能体需要经历多个步骤：首先调用 GitHub API 获取仓库信息，成功获得仓库结构和文件列表，得到+0.1 的奖励；然后读取主要代码文件，成功获得代码内容，得到+0.1 的奖励；接着分析代码质量合理，得到+0.2 的奖励；最后生成分析报告质量高，得到+0.6 的奖励。总奖励是所有步骤的累加：1.0。

可以看到，Agentic RL 的关键特征是多步交互、每一步的行动都会改变环境状态、每一步都可以获得反馈、优化整个任务的完成质量。

强化学习是基于马尔可夫决策过程(Markov Decision Process，MDP)框架进行形式化的。MDP 由五元组 (S, A, P, R, γ) 定义：状态空间 S 、行动空间 A 、状态转移函数 $P(s'|s, a)$ 、奖励函数 $R(s, a)$ 、折扣因子 γ 。让我们从 MDP 的角度对比 PBRFT 和 Agentic RL，如表 11.1 所示。

表 11.1 PBRFT 与 Agentic RL 对比



在状态方面，PBRFT 的状态 s_0 仅由用户提示构成，时间跨度 $T = 1$ (单步)，状态不变化，可以表示为 $s_0 = \text{prompt}$ 。而 Agentic RL 的状态 s_t 包含历史观察和上下文，时间跨度 $T \gg 1$ (多步)，状态随行动演化，可以表示为 $s_t = (\text{prompt}, o_1, o_2, \dots, o_t)$ ，其中 o_t 是第 t 步的观察(如工具返回结果、环境反馈等)。

在行动方面，PBRFT 的行动空间只有文本生成，单一行动类型，表示为 $a = y \sim \pi_\theta(y|s_0)$ 。而 Agentic RL 的行动空间包含文本生成、工具调用、环境操作等多种类型，表示为 $a_t \in \{a_t^{\text{text}}, a_t^{\text{tool}}\}$ ，例如 a_t^{text} 是生成思考过程或回答， a_t^{tool} 是调用计算器、搜索引擎等工具。

在转移函数方面，PBRFT 无状态转移，表示为 $P(s'|s, a) = \delta(s' - s_{\text{terminal}})$ 。而 Agentic RL 的状态根据行动和环境动态变化，表示为 $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ ，例如调用搜索工具后，状态会包含搜索结果。

在奖励方面，PBRFT 只有单步奖励 $r(s_0, a)$ ，仅在任务结束时给予，表示为 $R_{\text{PBRFT}} = r(s_0, y)$ ，通常由奖励模型给出： $r(s_0, y) = r_\phi(s_0, y)$ 。而 Agentic RL 有多步奖励 $r(s_t, a_t)$ ，可以在中间步骤给予部分奖励，表示为：

$$\$ \$ R_{\text{Agentic}} = \sum_{t=0}^T \gamma^t r(s_t, a_t) \$ \$$$

其中 $\gamma \in [0, 1]$ 是折扣因子， $r(s_t, a_t)$ 可以是稀疏奖励(只在任务完成时给予，如答案正确 +1)、密集奖励(每步都给予，如工具调用成功 +0.1)或结合两者的混合奖励。

在目标函数方面，PBRFT 最大化单步期望奖励：

$$J_{\text{PBRFT}}(\theta) = \mathbb{E}_{s_0, y \sim \pi_\theta}[r(s_0, y)]$$

而 Agentic RL 最大化累积折扣奖励：

$$\$ \$ J_{\text{Agentic}}(\theta) = \mathbb{E}_{\tau}[\gamma^t \sum_{t=0}^T r(s_t, a_t)] \$ \$$$

其中 $\tau = (s_0, a_0, s_1, a_1, \dots, s_T)$ 是完整的轨迹(trajecoty)。

这种转变不仅仅是技术细节的差异，而是思维方式的根本转变。PBRFT 思维关注“如何让模型生成更好的单个回答”，优化回答质量，关注语言表达，进行单步决策。而 Agentic RL 思维关注“如何让智能体完成复杂任务”，优化任务完成度，关注行动策略，进行多步规划。这种转变使得 LLM 从“对话助手”进化为“自主智能体”，能够主动寻找信息、知道何时、如何使用外部工具、为了最终目标，愿意执行看似“绕路”的中间步骤、从错误学习。

Agentic RL 的目标是赋予 LLM 智能体六大核心能力，如图 11.2 所示。



图 11.2 Agentic RL 的六大核心能力

推理(Reasoning)是指从给定信息中逻辑地得出结论的过程，是智能体的核心能力。传统的 CoT 提示方法依赖少样本示例，泛化能力有限；SFT 只能模仿训练数据中的推理模式，难以创新。强化学习的优势在于通过试错学习有效的推理策略，发现训练数据中没有的推理路径，学会何时需要深度思考、何时可以快速回答。推理任务可以建模为序列决策问题，给定问题 q ，智能体需要生成推理链 $c = (c_1, c_2, \dots, c_n)$ 和最终答案 a 。奖励函数通常设计为 $r(q, c, a) = 1$ if $a = a^*$ else 0，训练目标是 $\max_{\theta} \mathbb{E}_{q, (c, a) \sim \pi_{\theta}} [r(q, c, a)]$ 。通过这种方式，模型学会生成高质量的推理链，而不仅仅是记忆答案。

工具使用(Tool Use)是指智能体调用外部工具来完成任务的能力。在工具使用任务中，行动空间扩展为 $a_t \in \{a_t^{\text{think}}, a_t^{\text{tool}}\}$ ，其中 a_t^{think} 是生成思考过程， $a_t^{\text{tool}} = (\text{tool_name}, \text{arguments})$ 是调用工具。强化学习让智能体学会何时需要使用工具、选择哪个工具、如何组合多个工具。例如，在解决数学问题时，智能体需要学会何时使用计算器、何时使用代码解释器、何时直接推理。

记忆(Memory)是指智能体保持和重用过去信息的能力，对于长期任务至关重要。LLM 的上下文窗口有限，静态检索策略（如 RAG）无法针对任务优化。强化学习让智能体学会记忆管理策略：决定哪些信息值得记住、何时更新记忆、何时删除过时信息。这类似于人类的工作记忆，我们会主动管理大脑中的信息，保留重要的、遗忘无关的。

规划(Planning)是指制定行动序列以达成目标的能力。传统的 CoT 是线性思考，无法回溯；提示工程使用静态规划模板，难以适应新情况。强化学习让智能体学会动态规划：通过试错发现有效的行动序列，学会权衡短期和长期收益。例如，在多步任务中，智能体可能需要先执行一些看似“绕路”的步骤，例如收集信息，才能最终完成任务。

自我改进(Self-Improvement)是指智能体回顾自身输出、纠正错误并优化策略的能力。强化学习让智能体学会自我反思：识别自己的错误、分析失败原因、调整策略。这种能力使得智能体能够在没有人工干预的情况下持续改进，类似于人类的“从错误中学习”。

感知(Perception)是指理解多模态信息的能力。例如，强化学习可以提升视觉推理能力，让模型学会使用视觉工具，学会视觉规划。这使得智能体不仅能理解文本，还能理解和操作视觉世界。

11.1.4 HelloAgents 的 Agentic RL 设计

在理解了 Agentic RL 的核心理念后，让我们看看如何在 HelloAgents 框架中实现这些能力。

在技术选型上，我们集成了 TRL（Transformer Reinforcement Learning）框架^[9]，模型选择 Qwen3-0.6B^[10]。TRL 是 Hugging Face 的强化学习库，成熟稳定、功能完整、易于集成。Qwen3-0.6B 是阿里云的小型语言模型，0.6B 参数适合普通 GPU 训练，性能优秀且开源免费。

HelloAgents 的 Agentic RL 模块采用四层架构设计，如图 11.3 所示。



图 11.3 HelloAgents Agentic RL 架构

最底层是数据集层，包含 GSM8KDataset 类、create_sft_dataset() 函数和 create_rl_dataset() 函数，负责数据加载和格式转换。第二层是奖励函数层，包含 MathRewardFunction 基类、AccuracyReward 准确率奖励、LengthPenaltyReward 长度惩罚、StepReward 步骤奖励，以及便捷创建函数 create_*_reward()，负责定义什么是好的行为。第三层是训练器层，包含 SFTTrainerWrapper 和 GRPOTrainerWrapper，负责具体的训练逻辑和 LoRA 支持。最顶层是统一接口层，提供 RLTrainingTool 统一训练工具，支持四种操作：action="train"（训练模型）、action="load_dataset"（加载数据集）、action="create_reward"（创建奖励函数）、action="evaluate"（评估模型）。

11.1.5 快速上手示例

在深入学习之前，让我们先快速体验一下完整的训练流程。由于这一章的理论部分比较多，实战需要调试的地方也十分繁琐，因此不专注于构造工具而是学会应用。首先安装 HelloAgents 框架：

```
# 安装HelloAgents框架(第11章版本)
pip install "hello-agents[rl]==0.2.5"

# 或者从源码安装
cd HelloAgents
pip install -e ".[rl]"
```

然后运行快速训练示例：

```
import sys
import json

from hello_agents.tools import RLTrainingTool

# 创建RL训练工具
rl_tool = RLTrainingTool()

# 1. 快速测试:SFT训练(10个样本, 1个epoch)
sft_result_str = rl_tool.run({
    "action": "train",
    "algorithm": "sft",
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/quick_test_sft",
```

```

    "max_samples": 10,      # 只用10个样本快速测试
    "num_epochs": 1,        # 只训练1轮
    "batch_size": 2,
    "use_lora": True       # 使用LoRA加速训练
})

sft_result = json.loads(sft_result_str)
print(f"\n\n SFT训练完成,模型保存在: {sft_result['output_dir']}")

# 2. GRPO训练(5个样本,1个epoch)
grpo_result_str = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",
    "model_name": "Qwen/Qwen3-0.6B", # 使用基础模型
    "output_dir": "./models/quick_test_grpo",
    "max_samples": 5,            # 只用5个样本快速测试
    "num_epochs": 1,
    "batch_size": 2,            # 必须能被num_generations(8)整除,使用2
    "use_lora": True
})

grpo_result = json.loads(grpo_result_str)
print(f"\n\n GRPO训练完成,模型保存在: {grpo_result['output_dir']}")

# 3. 评估模型
eval_result_str = rl_tool.run({
    "action": "evaluate",
    "model_path": "./models/quick_test_grpo",
    "max_samples": 10,          # 在10个测试样本上评估
    "use_lora": True
})

eval_result = json.loads(eval_result_str)
print(f"\n\n 评估完成:")
print(f" - 准确率: {eval_result['accuracy']}")
print(f" - 平均奖励: {eval_result['average_reward']}")
print(f" - 测试样本数: {eval_result['num_samples']}")

print("\n" + "=" * 50)
print("🎉 恭喜!你已经完成了第一个Agentic RL模型的训练!")
print("=" * 50)
print(f"\n模型路径:")
print(f"  SFT模型: {sft_result['output_dir']}")
print(f"  GRPO模型: {grpo_result['output_dir']}")

```

这个快速示例展示了完整的训练流程:SFT 训练让模型学习基础的推理格式和对话模式，GRPO 训练通过强化学习优化推理策略提升准确率，模型评估在测试集上评估训练效果。另外跑完之后准确率很低是正常现象，因为现在模型只见过 0.7% 的训练样本，并且只运行了一轮。

11.2 数据集与奖励函数

数据集和奖励函数是强化学习训练的两大基石。数据集定义了智能体要学习的任务，奖励函数定义了什么是好的行为。在本节中，我们将学习如何准备训练数据和设计奖励函数。

11.2.1 GSM8K 数学推理数据集

数学推理是评估 LLM 推理能力的理想任务。首先，数学问题有明确的正确答案，可以自动评估，不需要人工标注或复杂的奖励模型。其次，解决数学问题需要分解问题、逐步推导，这正是多步推理的典型场景。最后，学到的推理能力可以迁移到其他领域，具有很强的泛化性。相比之下，开放式问答任务(如“如何学习编程?”)的答案质量难以客观评估，需要大量人工标注。

GSM8K(Grade School Math 8K)^[4]是一个高质量的小学数学应用题数据集。如表 11.2 所示，数据集包含 7,473 个训练样本和 1,319 个测试样本，难度为小学数学水平(2-8 年级)，题型为应用题，需要 2-8 步推理才能得出答案。

表 11.2 GSM8K 数据集统计

让我们看一个典型的 GSM8K 问题:

问题: Natalia sold clips to 48 of her friends in April, and then she sold half as many clips in May. How many clips did Natalia sell altogether in April and May?

答案: Natalia sold $48/2 = <<48/2=24>>24$ clips in May.
 $Natalia \text{ sold } 48+24 = <<48+24=72>>72$ clips altogether in April and May.
72

最终答案: 72

这个问题需要两步推理:首先计算 5 月份卖出的数量(48 的一半)，然后计算总数(4 月+5 月)。答案中的<<48/2=24>>是中间计算步骤的标记，#### 72 标记最终答案。

GSM8K 数据集需要转换为不同的格式，以适应不同的训练方法，如图 11.4 所示。



图 11.4 GSM8K 数据格式转换

原始格式直接来自数据集，包含问题(question)和答案(answer，含解题步骤)，适合人类阅读。SFT 格式用于监督微调，将问题转换为对话格式的 prompt，将完整解答作为 completion。例如：

```
{
  "prompt": "<|im_start|>user\nNatalia sold clips to 48 of her friends...<|im_end|>\n<|im_start|>assistant\n",
  "completion": "Let me solve this step by step.\n\nStep 1: ...\\n\\nFinal Answer: 72<|im_end|>"
}
```

关键点是使用模型的对话模板(如 Qwen 的<|im_start|>标记)，prompt 包含用户问题，completion 包含完整的解题过程和答案。这样模型可以学习如何格式化输出、如何分步推理。

RL 格式用于强化学习，只提供问题和正确答案，不提供解题过程。例如：

```
{
  "prompt": "<|im_start|>user\nNatalia sold clips to 48 of her friends...<|im_end|>\n<|im_start|>assistant\n",
  "ground_truth": "72"
}
```

关键点是 prompt 与 SFT 相同，但 ground_truth 只包含最终答案(用于计算奖励)，模型需要自己生成完整的推理过程。这种设计迫使模型学会自主推理，而不是简单地记忆答案。

如表 11.3 所示，三种格式各有用途。

表 11.3 数据格式对比



HelloAgents 提供了便捷的数据集加载函数。让我们通过代码来加载和查看数据集：

```
from hello_agents.tools import RLTrainingTool
import json

# 创建工具
rl_tool = RLTrainingTool()

# 1. 加载SFT格式数据集
sft_result = rl_tool.run({
    "action": "load_dataset",
    "format": "sft",
    "max_samples": 5 # 只加载5个样本查看
})
sft_data = json.loads(sft_result)

print(f"数据集大小: {sft_data['dataset_size']}")
print(f"数据格式: {sft_data['format']}")
print(f"样本字段: {sft_data['sample_keys']}")

# 2. 加载RL格式数据集
rl_result = rl_tool.run({
    "action": "load_dataset",
    "format": "rl",
    "max_samples": 5
})
rl_data = json.loads(rl_result)

print(f"数据集大小: {rl_data['dataset_size']}")
print(f"数据格式: {rl_data['format']}")
print(f"样本字段: {rl_data['sample_keys']}")
```

可以看到，SFT 格式包含完整的解题过程，用于监督学习；RL 格式只包含最终答案，模型需要自己生成推理过程。`max_samples` 参数控制加载的样本数量，方便快速测试。

11.2.2 奖励函数设计

奖励函数是强化学习的核心，它定义了什么是“好的行为”。一个好的奖励函数能够引导智能体学习到正确的策略，而一个糟糕的奖励函数可能导致训练失败或学到错误的行为。

在强化学习中，奖励函数 $r(s, a)$ 或 $r(s, a, s')$ 为智能体的每个行动分配一个数值奖励。智能体的目标是最大化累积奖励：

$$\$ \$ J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^T r(s_t, a_t) \right] \$ \$$$

对于数学推理任务，我们可以简化为：

$$r(q, a) = f(a, a^*)$$

其中 q 是问题， a 是模型生成的答案， a^* 是正确答案， f 是评估函数。

奖励函数的设计直接影响训练效果。好的奖励函数应该能清楚地定义什么是成功、能够提供梯度信号、不会产生过大的方差、容易调整和组合。糟糕的奖励函数可能只在任务结束时给奖励，中间步骤无反馈、存在奖励欺骗，使得智能体找到“作弊”方式获得高奖励、多个目标相互矛盾、方差过大，训练不收敛。

HelloAgents 提供了三种内置奖励函数，可以单独使用或组合使用，如图 11.5 所示。



图 11.5 奖励函数设计

(1) 准确率奖励

准确率奖励(AccuracyReward)是最基础的奖励函数，它只关心答案是否正确。数学定义为：

$$\$ \$ r_{\text{acc}}(a, a^*) = \begin{cases} 1 & \text{if } a = a^* \\ 0 & \text{otherwise} \end{cases} \$ \$$$

其中 a 是模型生成的答案， a^* 是正确答案。这是一个二值奖励函数，答案正确得 1 分，错误得 0 分。

实现时需要处理答案提取和比较。模型的输出可能包含大量文本，我们需要提取最终答案。常见的提取方法包括：查找“Final Answer:”后的数字、查找“###”标记后的数字、使用正则表达式提取最后一个数字。答案比较时需要处理数值精度(如 72.0 和 72 应该视为相同)、单位转换(如 1000 和 1k)、格式差异(如“72”和“seventy-two”)。

使用示例：

```
from hello_agents.tools import RLTrainingTool
import json
rl_tool = RLTrainingTool()

# 创建准确率奖励函数
reward_result = rl_tool.run({
    "action": "create_reward",
    "reward_type": "accuracy"
})
reward_data = json.loads(reward_result)

print(f"奖励类型: {reward_data['reward_type']}")
print(f"描述: {reward_data['description']}")

# 注意: RLTrainingTool的create_reward操作返回的是配置信息,
# 实际的奖励函数会在训练时自动创建和使用
```

输出：

```
预测: 72, 真实: 72, 奖励: 1.0
预测: 72.0, 真实: 72, 奖励: 1.0
预测: 73, 真实: 72, 奖励: 0.0
```

准确率奖励的优点是简单直接，容易理解和实现，适合有明确正确答案的任务。缺点是奖励稀疏，只有答案完全正确才有奖励，无法区分“接近正确”和“完全错误”，可能导致训练初期缺乏有效反馈。

(2) 长度惩罚

长度惩罚(LengthPenaltyReward)鼓励模型生成简洁的回答，避免冗长啰嗦。数学定义为：

$$r_{\text{length}}(a, a^*, l) = r_{\text{acc}}(a, a^*) - \alpha \cdot \max(0, l - l_{\text{target}})$$

其中 l 是生成文本的长度(字符数或 token 数)， l_{target} 是目标长度， α 是惩罚系数(默认 0.001)。只有在答案正确的情况下才应用长度惩罚，避免模型为了减少惩罚而生成错误的短答案。

设计思路是：如果答案错误，奖励为 0(无论长度)；如果答案正确且长度合理，奖励为 1；如果答案正确但过长，奖励为 $1 - \alpha \cdot (l - l_{\text{target}})$ 。例如，目标长度 200 字符，实际长度 500 字符，惩罚系数 0.001，则奖励为 $1 - 0.001 \times (500 - 200) = 0.7$ 。

使用示例：

```
# 创建长度惩罚奖励函数
reward_result = rl_tool.run({
    "action": "create_reward",
    "reward_type": "length_penalty",
    "max_length": 1024,      # 最大长度
    "penalty_weight": 0.001  # 惩罚权重
})
reward_data = json.loads(reward_result)

print(f"奖励类型: {reward_data['reward_type']}")
print(f"描述: {reward_data['description']}")
print(f"最大长度: {reward_data['max_length']}")
print(f"惩罚权重: {reward_data['penalty_weight']}")
```

输出：

```
预测: 72, 真实: 72, 长度: 50, 奖励: 1.000
预测: 72, 真实: 72, 长度: 200, 奖励: 1.000
```

预测: 72, 真实: 72, 长度: 500, 奖励: 0.700
 预测: 73, 真实: 72, 长度: 50, 奖励: 0.000

长度惩罚的优点是鼓励简洁表达，避免模型生成冗余内容，可以控制推理成本(更短的输出意味着更少的 token 消耗)。缺点是可能抑制详细推理，需要仔细调整惩罚系数，不同任务的最优长度差异很大。

(3) 步骤奖励

步骤奖励(StepReward)鼓励模型生成清晰的推理步骤，提高可解释性。数学定义为:

$$r_{\text{step}}(a, a^*, s) = r_{\text{acc}}(a, a^*) + \beta \cdot s$$

其中 s 是检测到的推理步骤数量， β 是步骤奖励系数(默认 0.1)。同样，只有在答案正确的情况下才给予步骤奖励。

步骤检测方法包括:查找"Step 1:“，”Step 2:“等标记、查找换行符数量、使用正则表达式匹配推理模式。例如，一个包含 3 个清晰步骤的正确答案，奖励为 $1 + 0.1 \times 3 = 1.3$ 。

使用示例:

```
# 创建步骤奖励函数
reward_result = rl_tool.run({
    "action": "create_reward",
    "reward_type": "step",
    "step_bonus": 0.1 # 每个步骤奖励0.1
})
reward_data = json.loads(reward_result)

print(f"奖励类型: {reward_data['reward_type']}")
print(f"描述: {reward_data['description']}")
print(f"步骤奖励: {reward_data['step_bonus']}")
```

输出:

```
预测: 72, 真实: 72, 步骤: 0, 奖励: 1.00
预测: 72, 真实: 72, 步骤: 2, 奖励: 1.20
预测: 72, 真实: 72, 步骤: 5, 奖励: 1.50
预测: 73, 真实: 72, 步骤: 5, 奖励: 0.00
```

步骤奖励的优点是鼓励可解释的推理，生成的答案更容易验证和调试，有助于模型学习系统化的思考方式。缺点是可能导致模型为了获得更多奖励生成冗余步骤，需要平衡步骤数量和答案质量，步骤检测可能不准确。

在实际应用中，我们通常会组合多个奖励函数，以平衡不同的目标。常见的组合策略包括:

准确率 + 长度惩罚:鼓励简洁正确的答案，适合对话系统、问答系统。公式为:

$$r = r_{\text{acc}} - \alpha \cdot \max(0, l - l_{\text{target}})$$

准确率 + 步骤奖励:鼓励详细的推理过程，适合教育场景、可解释 AI。公式为:

$$r = r_{\text{acc}} + \beta \cdot s$$

三者平衡:全面优化答案质量、简洁性和可解释性。公式为: $r = r_{\text{acc}} - \alpha \cdot \max(0, l - l_{\text{target}}) + \beta \cdot s$

需要仔细调整权重 α 和 β ，避免某个目标过度主导。

使用示例:

```
# 组合奖励函数:准确率 + 长度惩罚 + 步骤奖励
# 注意: RLTrainingTool目前支持单一奖励类型
# 组合奖励需要在训练配置中通过reward_fn参数指定
# 这里展示如何配置不同类型的奖励函数

# 准确率奖励
accuracy_result = rl_tool.run({
    "action": "create_reward",
    "reward_type": "accuracy"
})
print("准确率奖励:", json.loads(accuracy_result)['description'])

# 长度惩罚奖励
length_result = rl_tool.run({
    "action": "create_reward",
    "reward_type": "length_penalty",
    "max_length": 1024,
    "penalty_weight": 0.001
})
print("长度惩罚奖励:", json.loads(length_result)['description'])

# 步骤奖励
step_result = rl_tool.run({
    "action": "create_reward",
    "reward_type": "step",
    "step_bonus": 0.1
})
```

```

    })
print("步骤奖励:", json.loads(step_result)['description'])

```

输出:

```

组合奖励: 1.200
- 准确率: 1.0
- 长度惩罚: -0.100
- 步骤奖励: +0.3

```

如表 11.4 所示，不同奖励函数适合不同的应用场景。

表 11.4 奖励函数对比



11.2.3 自定义数据集和奖励函数

虽然 HelloAgents 提供了 GSM8K 数据集和常用奖励函数，但在实际应用中，你可能需要使用自己的数据集或设计特定的奖励函数。本节将介绍如何扩展框架。

在使用自定义数据集之前，需要了解两种训练格式的数据要求：

SFT 格式: 用于监督微调，需要包含以下字段: - `prompt`: 输入提示(包含 system 和 user 消息) - `completion`: 期望的输出 - `text`: 完整的对话文本(可选)

RL 格式: 用于强化学习，需要包含以下字段: - `question`: 原始问题 - `prompt`: 输入提示(包含 system 和 user 消息) - `ground_truth`: 正确答案 - `full_answer`: 完整答案(包含推理过程)

(1) 使用 `format_math_dataset` 转换

最简单的方法是准备包含`question`和`answer`字段的原始数据，然后使用`format_math_dataset()`函数自动转换：

```

from datasets import Dataset
from hello_agents.rl import format_math_dataset

# 1. 准备原始数据
custom_data = [
    {
        "question": "What is 2+2?",
        "answer": "2+2=4. #### 4"
    },
    {
        "question": "What is 5*3?",
        "answer": "5*3=15. #### 15"
    },
    {
        "question": "What is 10+7?",
        "answer": "10+7=17. #### 17"
    }
]

# 2. 转换为Dataset对象
raw_dataset = Dataset.from_list(custom_data)

# 3. 转换为SFT格式
sft_dataset = format_math_dataset(
    dataset=raw_dataset,
    format_type="sft",
    model_name="Qwen/Qwen3-0.6B"
)
print(f"SFT数据集: {len(sft_dataset)}个样本")
print(f"字段: {sft_dataset.column_names}")

# 4. 转换为RL格式
rl_dataset = format_math_dataset(
    dataset=raw_dataset,
    format_type="rl",
    model_name="Qwen/Qwen3-0.6B"
)
print(f"RL数据集: {len(rl_dataset)}个样本")
print(f"字段: {rl_dataset.column_names}")

```

(2) 直接传入自定义数据集

使用 RLTrainingTool 时，可以通过`custom_dataset`参数直接传入自定义数据集：

```

from hello_agents.tools import RLTrainingTool
rl_tool = RLTrainingTool()

# SFT训练
result = rl_tool.run({
    "action": "train",
    "algorithm": "sft",
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/custom_sft",
}

```

2/9/26, 1:56 PM

```
"num_epochs": 3,  
"batch_size": 4,  
"use_lora": True,  
"custom_dataset": sft_dataset # 直接传入自定义数据集  
})  
  
# GRPO训练  
result = rl_tool.run({  
    "action": "train",  
    "algorithm": "grpo",  
    "model_name": "Qwen/Qwen3-0.6B",  
    "output_dir": "./models/custom_grpo",  
    "num_epochs": 2,  
    "batch_size": 2,  
    "use_lora": True,  
    "custom_dataset": rl_dataset # 直接传入自定义数据集  
})
```

(3) 注册自定义数据集(推荐)

对于需要多次使用的数据集，推荐使用注册方式：

```
# 1. 注册数据集  
rl_tool.register_dataset("my_math_dataset", rl_dataset)  
  
# 2. 使用注册的数据集  
result = rl_tool.run({  
    "action": "train",  
    "algorithm": "grpo",  
    "dataset": "my_math_dataset", # 使用注册的数据集名称  
    "output_dir": "./models/custom_grpo",  
    "num_epochs": 2,  
    "use_lora": True  
})
```

奖励函数用于评估模型生成的答案质量。自定义奖励函数需要遵循以下签名：

```
from typing import List  
import re  
  
def custom_reward_function(  
    completions: List[str],  
    **kwargs  
) -> List[float]:  
    ....  
    自定义奖励函数  
  
    Args:  
        completions: 模型生成的完成文本列表  
        **kwargs: 其他参数,通常包含：  
            - ground_truth: 正确答案列表  
            - 其他数据集字段  
  
    Returns:  
        奖励值列表(每个值在0.0-1.0之间)  
    ....  
    ground_truths = kwargs.get("ground_truth", [])  
    rewards = []  
  
    for completion, truth in zip(completions, ground_truths):  
        reward = 0.0  
  
        # 提取答案  
        numbers = re.findall(r'-?\d+\.\?\d*', completion)  
        if numbers:  
            try:  
                pred = float(numbers[-1])  
                truth_num = float(truth)  
                error = abs(pred - truth_num)  
  
                # 根据误差给予不同奖励  
                if error < 0.01:  
                    reward = 1.0 # 完全正确  
                elif error < 1.0:  
                    reward = 0.8 # 非常接近  
                elif error < 5.0:  
                    reward = 0.5 # 接近  
  
                # 额外奖励:鼓励展示推理步骤  
                if "step" in completion.lower() or "=" in completion:  
                    reward += 0.1  
  
            except ValueError:  
                reward = 0.0  
  
        rewards.append(min(reward, 1.0)) # 限制最大值为1.0  
  
    return rewards
```

有两种方式使用自定义奖励函数：

(1) 直接传入

```
result = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/custom_grpo",
    "custom_dataset": rl_dataset,
    "custom_reward": custom_reward_function # 直接传入奖励函数
})
```

(2) 注册使用(推荐)

```
# 1. 注册奖励函数
rl_tool.register_reward_function("my_reward", custom_reward_function)

# 2. 使用注册的奖励函数
result = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",
    "dataset": "my_math_dataset",
    "output_dir": "./models/custom_grpo"
    # 奖励函数会自动使用与dataset同名的注册函数
})
```

以下是一个完整的自定义数据集和奖励函数示例:

```
from datasets import Dataset
from hello_agents.tools import RLTrainingTool
from hello_agents.rl import format_math_dataset
import re
from typing import List

# 1. 准备自定义数据
custom_data = [
    {"question": "What is 2+2?", "answer": "2+2=4. #### 4"},
    {"question": "What is 5+3?", "answer": "5+3=8. #### 8"},
    {"question": "What is 10+7?", "answer": "10+7=17. #### 17"}
]

# 2. 转换为训练格式
raw_dataset = Dataset.from_list(custom_data)
rl_dataset = format_math_dataset(raw_dataset, format_type="rl")

# 3. 定义自定义奖励函数
def tolerant_reward(completions: List[str], **kwargs) -> List[float]:
    """带容差的奖励函数"""
    ground_truths = kwargs.get("ground_truth", [])
    rewards = []

    for completion, truth in zip(completions, ground_truths):
        numbers = re.findall(r'-?\d+\.\?\d*', completion)
        if numbers:
            try:
                pred = float(numbers[-1])
                truth_num = float(truth)
                error = abs(pred - truth_num)

                if error < 0.01:
                    reward = 1.0
                elif error < 5.0:
                    reward = 0.5
                else:
                    reward = 0.0
            except ValueError:
                reward = 0.0
            else:
                reward = 0.0
            rewards.append(reward)
        else:
            reward = 0.0

    return rewards

# 4. 创建工具并注册
rl_tool = RLTrainingTool()
rl_tool.register_dataset("my_dataset", rl_dataset)
rl_tool.register_reward_function("my_dataset", tolerant_reward)

# 5. 训练
result = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",
    "model_name": "Qwen/Qwen3-0.6B",
    "dataset": "my_dataset",
    "output_dir": "./models/custom_grpo",
    "num_epochs": 2,
    "batch_size": 2,
    "use_lora": True
})
```

11.3 SFT 训练

监督微调(Supervised Fine-Tuning, SFT)是强化学习训练的第一步，也是最重要的基础。SFT 让模型学习任务的基本格式、对话模式和初步的推理能力。没有 SFT 的基础，直接进行强化学习往往会失败，因为模型连基本的输出格式都不会。

11.3.1 为什么需要 SFT

在开始强化学习之前，我们需要先进行 SFT 训练。这是因为预训练模型虽然具备强大的语言能力，但它并不知道如何完成特定任务。预训练模型的训练目标是预测下一个词，而不是解决数学问题或使用工具。预训练模型的输出格式是自由文本，而我们需要结构化的输出(如“Step 1: ...”，“Step 2: ...”，“Final Answer: ...”)。预训练模型没有见过任务相关的数据，不知道什么是“好的”推理过程。

SFT 的作用是教会模型任务的基本规则。首先，学习输出格式，让模型知道如何组织答案(如使用“Step 1”，“Final Answer”等标记)。其次，学习推理模式，通过示例学习如何分解问题、逐步推导。再次，建立基线能力，为后续的强化学习提供一个合理的起点。最后，减少探索空间，强化学习不需要从零开始，可以在 SFT 的基础上优化。

让我们通过一个对比实验来理解 SFT 的重要性。假设我们直接用预训练模型解决 GSM8K 问题：

```
from transformers import AutoTokenizer, AutoModelForCausalLM

# 加载预训练模型
model_name = "Qwen/Qwen3-0.6B"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# 测试问题
question = """Natalia sold clips to 48 of her friends in April, and then she sold half as many clips in May. How many clips did Natalia sell altogether?"""

# 构造输入
prompt = f"<|im_start|>user\n{question}<|im_end|>\n<|im_start|>assistant\n"
inputs = tokenizer(prompt, return_tensors="pt")

# 生成回答
outputs = model.generate(**inputs, max_new_tokens=200)
response = tokenizer.decode(outputs[0], skip_special_tokens=False)

print("预训练模型的回答:")
print(response)
```

预训练模型的输出可能是：

```
<|im_start|>user
Natalia sold clips to 48 of her friends in April, and then she sold half as many clips in May. How many clips did Natalia sell altogether in .
<|im_start|>assistant
I can help you with that. Let me think about this problem. Natalia sold clips to her friends. In April she sold 48 clips. In May she sold hal
```

可以看到，预训练模型虽然理解了问题，但输出非常冗长、缺乏结构、没有明确的答案、推理过程混乱。这样的输出无法用于强化学习，因为我们无法提取答案、无法评估质量、无法提供有效的奖励信号。

现在让我们看看 SFT 后的模型：

```
# 加载SFT模型(假设已经训练好)
sft_model_path = "./models/sft_model"
sft_model = AutoModelForCausalLM.from_pretrained(sft_model_path)

# 使用相同的问题
outputs = sft_model.generate(**inputs, max_new_tokens=200)
response = tokenizer.decode(outputs[0], skip_special_tokens=False)

print("SFT模型的回答:")
print(response)
```

SFT 模型的输出：

```
<|im_start|>user
Natalia sold clips to 48 of her friends in April, and then she sold half as many clips in May. How many clips did Natalia sell altogether in .
<|im_start|>assistant
Let me solve this step by step.
```

```
Step 1: Calculate clips sold in May
Natalia sold half as many clips in May as in April.
Clips in May = 48 / 2 = 24
```

```
Step 2: Calculate total clips
Total = April + May = 48 + 24 = 72
```

```
Final Answer: 72<|im_end|>
```

可以看到，SFT 模型的输出结构清晰(使用“Step 1”，“Step 2”，“Final Answer”标记)、推理正确、答案明确、格式统一。这样的输出可以用于强化学习，因为我们可以提取答案、计算奖励、优化策略。

如图 11.6 所示，SFT 是从预训练模型到强化学习的桥梁。



图 11.6 SFT 在训练流程中的作用

11.3.2 LoRA:参数高效微调

直接微调整个模型需要大量的计算资源和显存。对于 Qwen3-0.6B(0.6B 参数), 全量微调需要约 12GB 显存(FP16)或 24GB 显存(FP32)。对于更大的模型(如 7B、13B), 全量微调几乎不可能在消费级 GPU 上进行。

LoRA(Low-Rank Adaptation)^[3]是一种参数高效微调方法, 它只训练少量的额外参数, 而保持原模型参数冻结。LoRA 的核心思想是:模型微调时的参数变化可以用低秩矩阵表示。

假设原模型的权重矩阵为 $W \in \mathbb{R}^{d \times k}$, 微调后的权重为 $W' = W + \Delta W$ 。LoRA 假设 ΔW 可以分解为两个低秩矩阵的乘积:

$$\Delta W = BA$$

其中 $B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}, r \ll \min(d, k)$ 是秩(rank)。

前向传播时, 输出为:

$$h = Wx + \Delta Wx = Wx + BAx$$

原模型参数 W 保持冻结, 只训练 B 和 A 。

参数量对比:原模型参数量为 $d \times k$, LoRA 参数量为 $d \times r + r \times k = r(d + k)$ 。当 $r \ll \min(d, k)$ 时, LoRA 参数量远小于原模型。例如, 对于 $d = 4096, k = 4096, r = 8$ 的情况, 原模型参数量为 $4096 \times 4096 = 16,777,216$, LoRA 参数量为 $8 \times (4096 + 4096) = 65,536$, 参数量减少了 256 倍!

因此可以总结 LoRA 的优势:显存占用大幅降低、训练速度更快、易于部署、防止过拟合。不过训练的效果通常会比全量调参更差一些。

如表 11.5 所示, LoRA 在不同模型规模下的效果对比。

表 11.5 LoRA vs 全量微调对比



LoRA 的关键超参数包括:秩(rank, r), 控制 LoRA 矩阵的秩, 越大表达能力越强, 但参数量也越多, 典型值为 4-64, 默认 8;Alpha(α), LoRA 的缩放因子, 实际更新为 $\Delta W = \frac{\alpha}{r} BA$, 控制 LoRA 的影响强度, 典型值等于 rank;目标模块(target_modules), 指定哪些层应用 LoRA, 通常选择注意力层(q_proj, k_proj, v_proj, o_proj), 也可以包括 MLP 层(gate_proj, up_proj, down_proj)。

11.3.3 SFT 训练实战

现在让我们使用 HelloAgents 进行 SFT 训练。完整的训练流程包括:准备数据集、配置 LoRA、设置训练参数、开始训练、保存模型。

基础训练示例:

```
from hello_agents.tools import RLTrainingTool

# 创建训练工具
rl_tool = RLTrainingTool()

# SFT训练
result = rl_tool.run({
    # 训练配置
    "action": "train",
    "algorithm": "sft",

    # 模型配置
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/sft_model",

    # 数据配置
    "max_samples": 100,      # 使用100个样本快速测试

    # 训练参数
    "num_epochs": 3,         # 训练3轮
    "batch_size": 4,          # 批次大小
    "learning_rate": 5e-5,    # 学习率

    # LoRA配置
    "use_lora": True,        # 使用LoRA
    "lora_rank": 8,           # LoRA秩
    "lora_alpha": 16,          # LoRA alpha
})

print(f"\n✓ 训练完成!")
print(f" - 模型保存路径: {result['model_path']}")
print(f" - 训练样本数: {result['num_samples']}")
print(f" - 训练轮数: {result['num_epochs']}")
print(f" - 最终损失: {result['final_loss']:.4f}")
```

如果训练过程中损失逐渐下降, 说明模型正在学习。

(1) 训练参数详解

让我们详细了解各个训练参数的含义和调优建议。

数据参数:

- `max_samples`: 使用的训练样本数量。快速测试时可以用 100-1000 个样本，完整训练建议使用全部数据(7473 个样本)。更多数据通常带来更好的效果，但训练时间也更长。
- `split`: 数据集划分，默认“train”。可以设置为“train[:1000]”只使用前 1000 个样本。

训练参数:

- `num_epochs`: 训练轮数。1 轮表示遍历整个数据集一次。太少(1-2 轮)可能欠拟合，太多(>10 轮)可能过拟合。建议从 3 轮开始，观察损失曲线调整。
- `batch_size`: 每次更新使用的样本数。越大训练越稳定，但显存占用越高。建议根据显存调整:4GB 显存用 `batch_size=1-2`，8GB 显存用 `batch_size=4-8`，16GB 显存用 `batch_size=8-16`。
- `learning_rate`: 学习率，控制参数更新的步长。太小($1e-6$)收敛慢，太大($1e-3$)可能不收敛。SFT 推荐 $5e-5$ ，LoRA 可以稍大($1e-4$)。

LoRA 参数:

- `use_lora`: 是否使用 LoRA。建议始终开启，除非有充足的显存。
- `lora_rank`: LoRA 秩，控制表达能力。4-8 适合小任务，16-32 适合复杂任务，64 适合大规模微调。
- `lora_alpha`: LoRA 缩放因子，通常设置为 rank 的 2 倍。rank=8 时，`alpha=16`; rank=16 时，`alpha=32`。

优化器参数:

- `optimizer`: 优化器类型，默认“adamw”。AdamW 是最常用的选择，也可以尝试“sgd”或“adafactor”等。
- `weight_decay`: 权重衰减，防止过拟合。默认 0.01，可以尝试 0.001-0.1。
- `warmup_ratio`: 学习率预热比例。前 `warmup_ratio` 的步数学习率线性增加，然后线性衰减。默认 0.1(前 10% 步数预热)。

(2) 完整训练示例

让我们进行一次完整的 SFT 训练，使用全部数据和最佳实践:

```
from hello_agents.tools import RLTrainingTool
rl_tool = RLTrainingTool()

# 完整SFT训练
result = rl_tool.run({
    "action": "train",
    "algorithm": "sft",

    # 模型配置
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/sft_full",

    # 数据配置
    "max_samples": None,      # 使用全部数据(7473个样本)

    # 训练参数
    "num_epochs": 3,
    "batch_size": 8,
    "learning_rate": 5e-5,
    "warmup_ratio": 0.1,
    "weight_decay": 0.01,

    # LoRA配置
    "use_lora": True,
    "lora_rank": 16,          # 使用更大的rank
    "lora_alpha": 32,
    "lora_target_modules": ["q_proj", "k_proj", "v_proj", "o_proj"],

    # 其他配置
    "save_steps": 500,        # 每500步保存一次
    "logging_steps": 100,      # 每100步记录一次
    "eval_steps": 500,         # 每500步评估一次
})

print(f"训练完成！模型保存在: {result['model_path']}")
```

这个配置适合在 8GB 显存的 GPU 上训练，预计耗时 30-60 分钟。

(3) 训练监控和调试

在训练过程中，我们需要监控三个关键指标。损失(Loss)应该逐渐下降，如果不下降可能是学习率太小或数据有问题，如果下降后又上升则可能是学习率太大或出现过拟合。梯度范数(Gradient Norm)应该在 0.1-10 的合理范围内，过大(>100)说明出现梯度爆炸需要降低学习率，过小(<0.01)说明梯度消失需要检查模型配置。学习率(Learning Rate)应该按照 warmup 策略变化，前 10% 步数线性增加，然后线性衰减到 0。

训练中常见的问题及解决方案:显存不足时可以减小 `batch_size` 或 `max_length`，使用梯度累积或更小的模型;训练速度慢时可以增大 `batch_size`，减少 `logging` 频率，或使用混合精度训练;损失不下降时可以增大学习率，检查数据格式，或增加训练轮数;过拟合时可以增大 `weight_decay`，减少训练轮数，或使用更多数据。

11.3.4 模型评估

训练完成后，我们需要评估模型的效果。评估指标包括：

- **准确率(Accuracy)**: 答案完全正确的比例，最直接的指标，范围 0-1，越高越好。
- **平均奖励(Average Reward)**: 所有样本的平均奖励，综合考虑准确率、长度、步骤等因素，范围取决于奖励函数设计。
- **推理质量(Reasoning Quality)**: 推理过程的清晰度和逻辑性，需要人工评估或使用专门的评估模型。

使用 HelloAgents 评估模型：

```
from hello_agents.tools import RLTrainingTool
rl_tool = RLTrainingTool()

# 评估SFT模型
eval_result = rl_tool.run({
    "action": "evaluate",
    "model_path": "./models/sft_full",
    "max_samples": 100,      # 在100个测试样本上评估
    "use_lora": True,
})

eval_data = json.loads(eval_result)
print(f"\n评估结果:")
print(f" - 准确率: {eval_data['accuracy']}")
```

对于 Qwen3-0.6B 这样的小模型，SFT 后在 GSM8K 上达到 40-50% 的准确率是正常的。通过强化学习，我们可以进一步提升到 60-70%。

为了更好地理解 SFT 的效果，我们可以对比不同阶段的模型：

```
# 评估预训练模型(未经SFT)
base_result = rl_tool.run({
    "action": "evaluate",
    "model_path": "Qwen/Qwen3-0.6B",
    "max_samples": 100,
    "use_lora": False,
})
base_data = json.loads(base_result)

# 评估SFT模型
sft_result = rl_tool.run({
    "action": "evaluate",
    "model_path": "./models/sft_full",
    "max_samples": 100,
    "use_lora": True,
})
sft_data = json.loads(sft_result)

# 对比结果
print("模型对比:")
print(f"预训练模型准确率: {base_data['accuracy']}")
print(f"SFT模型准确率: {sft_data['accuracy']}
```

在本节中，我们学习了 SFT 的重要性(学习格式、建立基线)、LoRA 原理(低秩分解、参数高效)、SFT 训练实战(参数配置、训练监控)、模型评估(准确率、对比分析)。

11.4 GRPO 训练

在完成 SFT 训练后，我们已经得到了一个能够生成结构化答案的模型。但是，SFT 模型只是学会了“模仿”训练数据中的推理过程，并没有真正学会“思考”。强化学习可以让模型通过试错来优化推理策略，从而超越训练数据的质量。

11.4.1 从 PPO 到 GRPO

在强化学习领域，PPO(Proximal Policy Optimization)^[1]是最经典的算法之一。PPO 通过限制策略更新的幅度，保证训练的稳定性。但是，PPO 在 LLM 训练中存在一些问题：需要训练 Value Model(价值模型)，增加了训练复杂度和显存占用；需要同时维护四个模型(Policy Model、Reference Model、Value Model、Reward Model)，工程实现复杂；训练不稳定，容易出现奖励崩塌或策略退化。

GRPO(Group Relative Policy Optimization)^[2]是一种简化的 PPO 变体，专门为 LLM 设计。GRPO 的核心思想是：不需要 Value Model，使用组内相对奖励代替绝对奖励；简化训练流程，只需要 Policy Model 和 Reference Model；提高训练稳定性，减少奖励崩塌的风险。

让我们通过数学公式来理解 GRPO 的原理。PPO 的目标函数为：

$$\text{J}_{\text{PPO}}(\theta) = \mathbb{E}_{s,a}[\pi_{\theta}(a|s) \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta'}(a|s)} \right)^{\alpha} A(s,a)]$$

其中 $A(s,a)$ 是优势函数(Advantage)，需要 Value Model 来估计：

$$A(s, a) = Q(s, a) - V(s) = r(s, a) + \gamma V(s') - V(s)$$

GRPO 的目标函数简化为:

$$\text{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{s,a}[\pi_\theta] \left[\frac{\pi_\theta(a|s)}{\pi_{\text{ref}}(a|s)} \cdot (r(s,a) - \bar{r}_{\text{group}}) \right] - \beta \cdot \text{KL}(\pi_\theta \| \pi_{\text{ref}})$$

其中 \bar{r}_{group} 是组内平均奖励, β 是 KL 散度惩罚系数。关键区别在于:GRPO 使用 $r(s, a) - \bar{r}_{\text{group}}$ 代替优势函数 $A(s, a)$, 不需要 Value Model;GRPO 使用组内相对奖励, 减少奖励方差;GRPO 添加 KL 散度惩罚, 防止策略偏离太远。

如图 11.7 所示, PPO 和 GRPO 的训练流程对比。



图 11.7 PPO vs GRPO 训练流程

可以看到, GRPO 省去了 Value Model 的训练, 大大简化了流程。

如表 11.6 所示, PPO 和 GRPO 的详细对比。

表 11.6 PPO vs GRPO 对比



对于 LLM 训练, GRPO 是更好的选择, 因为它更简单、更稳定、显存占用更低。

11.4.2 GRPO 训练实战

现在让我们使用 HelloAgents 进行 GRPO 训练。GRPO 训练的前提是已经完成 SFT 训练, 因为 GRPO 需要一个合理的初始策略。

基础 GRPO 训练示例:

```
from hello_agents.tools import RLTrainingTool

# 创建训练工具
rl_tool = RLTrainingTool()

# GRPO训练
result = rl_tool.run({
    # 训练配置
    "action": "train",
    "algorithm": "grpo",

    # 模型配置
    "model_name": "./models/sft_full", # 从SFT模型开始
    "output_dir": "./models/grpo_model",

    # 数据配置
    "max_samples": 100,      # 使用100个样本快速测试

    # 训练参数
    "num_epochs": 3,
    "batch_size": 4,
    "learning_rate": 1e-5,   # GRPO学习率通常比SFT小

    # GRPO特定参数
    "num_generations": 4,   # 每个问题生成4个答案
    "kl_coef": 0.05,        # KL散度惩罚系数

    # LoRA配置
    "use_lora": True,
    "lora_rank": 16,
    "lora_alpha": 32,

    # 奖励函数配置
    "reward_type": "accuracy", # 使用准确率奖励
})

print(f"\n✓ 训练完成!")
print(f" - 模型保存路径: {result['model_path']}")
print(f" - 训练样本数: {result['num_samples']}")
print(f" - 训练轮数: {result['num_epochs']}")
print(f" - 平均奖励: {result['average_reward']:.4f}")
```

如果 GRPO 训练过程中平均奖励逐渐提升, KL 散度保持在合理范围内, 说明训练正常进行。

GRPO 有一些特定的参数需要理解和调优。

生成参数:

- `num_generations`: 每个问题生成多少个答案。越多越好, 但计算成本也越高。典型值为 4-8。生成多个答案的目的是计算组内相对奖励, 增加训练信号的多样性。

- `max_new_tokens`: 每个答案最多生成多少个 token。太少可能截断答案，太多浪费计算。建议 256-512。
- `temperature`: 生成温度，控制随机性。0 表示贪婪解码，1 表示标准采样。GRPO 建议 0.7-1.0，保持一定的探索性。

优化参数:

- `learning_rate`: GRPO 的学习率通常比 SFT 小，因为我们不想偏离 SFT 模型太远。建议 $1e-5$ 到 $5e-5$ 。
- `kl_coef`: KL 散度惩罚系数，控制策略更新的幅度。太小(0.01)可能导致策略偏离太远，太大(0.5)可能限制学习。建议 0.05-0.1。
- `clip_range`: 策略比率裁剪范围，类似 PPO 的 `epsilon`。建议 0.2。

奖励参数:

- `reward_type`: 奖励函数类型，可以是“accuracy”、“length_penalty”、“step”或“combined”。
- `reward_config`: 奖励函数的额外配置，如长度惩罚的目标长度、步骤奖励的系数等。

让我们进行一次完整的 GRPO 训练，使用全部数据和最佳实践:

```
from hello_agents.tools import RLTrainingTool
rl_tool = RLTrainingTool()

# 完整GRPO训练
result = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",

    # 模型配置
    "model_name": "./models/sft_full",
    "output_dir": "./models/grpo_full",

    # 数据配置
    "max_samples": None,      # 使用全部数据

    # 训练参数
    "num_epochs": 3,
    "batch_size": 4,
    "learning_rate": 1e-5,
    "warmup_ratio": 0.1,

    # GRPO特定参数
    "num_generations": 4,
    "max_new_tokens": 512,
    "temperature": 0.8,
    "kl_coef": 0.05,
    "clip_range": 0.2,

    # LoRA配置
    "use_lora": True,
    "lora_rank": 16,
    "lora_alpha": 32,

    # 奖励函数配置
    "reward_type": "combined",
    "reward_config": {
        "components": [
            {"type": "accuracy", "weight": 1.0},
            {"type": "length_penalty", "weight": 0.5, "target_length": 200},
            {"type": "step", "weight": 0.3, "step_bonus": 0.1}
        ]
    },
    # 其他配置
    "save_steps": 500,
    "logging_steps": 100,
})

print(f"训练完成！模型保存在: {result['model_path']}")
```

11.4.3 GRPO 训练过程解析

让我们深入理解 GRPO 的训练过程，看看每一步都发生了什么。

(1) 训练循环

GRPO 的训练循环包括以下步骤:

1. 采样阶段:对于每个问题，使用当前策略生成多个答案(`num_generations`个)。这些答案构成一个“组”，用于计算相对奖励。
2. 奖励计算:对每个生成的答案计算奖励 r_i 。奖励可以是准确率、长度惩罚、步骤奖励或它们的组合。
3. 相对奖励:计算组内平均奖励 $\bar{r} = \frac{1}{N} \sum_{i=1}^N r_i$ ，然后计算相对奖励 $\hat{r}_i = r_i - \bar{r}$ 。这样做的好处是减少奖励方差，使训练更稳定。
4. 策略更新:使用相对奖励更新策略，同时添加 KL 散度惩罚，防止策略偏离参考模型太远。
5. 重复:重复上述步骤，直到完成所有训练轮次。

让我们通过一个具体例子来理解:

```
# 假设我们有一个问题
question = "What is 48 + 24?"

# 生成4个答案
answers = [
    "48 + 24 = 72. Final Answer: 72",      # 正确
    "48 + 24 = 72. Final Answer: 72",      # 正确
    "48 + 24 = 70. Final Answer: 70",      # 错误
    "Let me think... 72. Final Answer: 72" # 正确但冗长
]

# 计算奖励(假设使用准确率 + 长度惩罚)
rewards = [1.0, 1.0, 0.0, 0.8] # 第4个答案因为冗长被惩罚

# 计算组内平均奖励
avg_reward = (1.0 + 1.0 + 0.0 + 0.8) / 4 = 0.7

# 计算相对奖励
relative_rewards = [
    1.0 - 0.7 = 0.3,      # 正确且简洁, 相对奖励为正
    1.0 - 0.7 = 0.3,      # 正确且简洁, 相对奖励为正
    0.0 - 0.7 = -0.7,     # 错误, 相对奖励为负
    0.8 - 0.7 = 0.1      # 正确但冗长, 相对奖励较小
]
```

策略更新: 增加前两个答案的概率, 减少第三个答案的概率

可以看到, 相对奖励机制鼓励模型生成“比平均水平更好”的答案, 而不是简单地追求高奖励。这样可以减少奖励方差, 提高训练稳定性。

(2) KL 散度惩罚

KL 散度惩罚是 GRPO 的关键组成部分, 它防止策略偏离参考模型太远。KL 散度定义为:

$$\text{D}_{\text{KL}}(\pi_{\theta} \parallel \pi_{\text{ref}}) = \mathbb{E}_{s,a \sim \pi_{\theta}} [\log \frac{\pi_{\theta}(a|s)}{\pi_{\text{ref}}(a|s)}]$$

在实践中, 我们计算每个 token 的 KL 散度, 然后求和:

$$\text{D}_{\text{KL}} = \sum_{t=1}^T \log \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\text{ref}}(a_t|s_t)}$$

KL 散度越大, 说明当前策略与参考模型差异越大。通过添加 KL 散度惩罚项 $-\beta \cdot D_{\text{KL}}$, 我们限制策略更新的幅度, 避免“遗忘”SFT 阶段学到的知识。

`kl_coef` (β) 的选择很重要:

- 太小(0.01): 策略可能偏离太远, 导致输出格式混乱或质量下降
- 太大(0.5): 策略更新受限, 学习缓慢, 难以超越 SFT 模型
- 建议(0.05-0.1): 平衡探索和稳定性

(3) 训练监控

在 GRPO 训练过程中, 我们需要监控以下指标:

- 平均奖励(Average Reward): 应该逐渐上升。如果奖励不上升, 可能是学习率太小、KL 惩罚太大、奖励函数设计不合理。如果奖励先升后降, 可能是过拟合或奖励崩塌。
- KL 散度(KL Divergence): 应该保持在合理范围内(0.01-0.1)。如果 KL 散度过大(>0.5), 说明策略偏离太远, 需要增大 `kl_coef` 或降低学习率。如果 KL 散度过小(<0.001), 说明策略几乎没有更新, 需要减小 `kl_coef` 或增大学习率。
- 准确率(Accuracy): 应该逐渐提升。这是最直观的指标, 反映模型的实际能力。
- 生成质量(Generation Quality): 需要人工检查生成的答案, 确保格式正确、推理清晰。

HelloAgents 集成了两种主流的训练监控工具:Weights & Biases(wandb)和TensorBoard。

方式 1: 使用 Weights & Biases(推荐)

Weights & Biases 是目前最流行的机器学习实验跟踪平台, 提供了强大的可视化和实验管理功能。

```
import os

# 1. 设置wandb(需要先注册账号: https://wandb.ai)
os.environ["WANDB_PROJECT"] = "hello-agents-grpo" # 项目名称
os.environ["WANDB_LOG_MODEL"] = "false"           # 不上传模型文件

# 2. 在训练配置中启用wandb
result = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/grpo_monitored",
    "num_epochs": 2,
```

```

    "batch_size": 2,
    "use_lora": True,
    # wandb会自动记录所有训练指标
})

# 训练完成后,访问 https://wandb.ai 查看训练曲线

```

wandb 会自动记录以下指标: - train/reward: 平均奖励 - train/kl: KL 散度 - train/loss: 训练损失 - train/learning_rate: 学习率 - train/epoch: 训练轮数

方式 2: 使用 TensorBoard

TensorBoard 是 TensorFlow 提供的可视化工具, 也支持 PyTorch 训练。

```

# 1. 训练时会自动在output_dir下创建tensorboard日志
result = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/grpo_tb",
    "num_epochs": 2,
    "batch_size": 2,
    "use_lora": True,
})

# 2. 启动TensorBoard查看训练曲线
# 在命令行运行:
# tensorboard --logdir=./models/grpo_tb
# 然后访问 http://localhost:6006

```

方式 3: 离线监控(无需外部工具)

如果不使用 wandb 或 TensorBoard, 也可以通过训练日志进行监控:

```

# 训练过程会打印详细日志
result = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/grpo_simple",
    "num_epochs": 2,
    "batch_size": 2,
    "use_lora": True,
})

# 日志示例:
# Epoch 1/2 | Step 100/500 | Reward: 0.45 | KL: 0.023 | Loss: 1.234
# Epoch 1/2 | Step 200/500 | Reward: 0.52 | KL: 0.031 | Loss: 1.156
# ...

```

在 GRPO 训练中, 可能会遇到一些问题。当奖励不上升时, 可能是学习率太小或 KL 惩罚太大限制了策略更新, 也可能是奖励函数设计不合理或 SFT 模型质量太差, 此时可以增大学习率(从 1e-5 到 5e-5)、减小 kl_coef(从 0.1 到 0.05)、检查奖励函数或重新训练 SFT 模型。

当 KL 散度爆炸(超过 0.5 甚至 1.0)导致生成答案格式混乱时, 通常是学习率太大或 KL 惩罚太小, 或者奖励函数过于激进, 可以降低学习率(从 5e-5 到 1e-5)、增大 kl_coef(从 0.05 到 0.1)、调整奖励函数或使用梯度裁剪。

当生成质量下降(准确率提升但格式混乱、推理不清晰)时, 可能是奖励函数只关注准确率忽略了其他质量指标, 或 KL 惩罚太小导致模型偏离 SFT 太远, 或出现过拟合, 此时应使用组合奖励函数同时优化多个指标、增大 kl_coef 保持一致性、减少训练轮数或增加训练数据。

GRPO 训练的显存占用比 SFT 高, 因为需要同时生成多个答案并存储参考模型输出, 容易出现 OOM。可以通过减小 num_generations(从 8 到 4)、batch_size(从 4 到 2)或 max_new_tokens(从 512 到 256), 或使用梯度检查点和混合精度训练来缓解。

11.5 模型评估与分析

训练完成后, 我们需要全面评估模型的性能, 不仅要看准确率这个指标, 还要深入分析模型的推理质量、错误模式、泛化能力等。本节将介绍如何系统地评估和分析 Agentic RL 模型。

11.5.1 评估指标体系

一个好的评估体系应该是多维度的, 从不同角度衡量模型的能力。我们将评估指标分为三类:准确性指标、效率指标、质量指标。

(1) 准确性指标

准确性指标衡量模型是否能够得出正确答案。

准确率(Accuracy):最基本的指标, 答案完全正确的比例。计算公式为: $\text{Accuracy} = \frac{\text{正确答案数}}{\text{总问题数}}$

优点是简单直观, 易于理解和比较。缺点是无法区分“接近正确”和“完全错误”, 对于复杂任务可能过于粗糙。

Top-K 准确率:生成 K 个答案, 只要有一个正确就算对。计算公式为: $\text{Accuracy}_{@K} = \frac{\text{至少有一个正确答案的问题数}}{\text{总问题数}}$

这个指标反映了模型的“潜力”, 即通过多次采样能否找到正确答案。

数值误差(Numerical Error):对于数学问题，可以计算预测值与真实值的误差。计算公式为:

$$\text{Error} = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$$

这个指标可以区分“接近正确”(如预测 72.5, 真实 72)和“完全错误”(如预测 100, 真实 72)。

(2) 效率指标

效率指标衡量模型生成答案的成本。

平均长度(Average Length):生成答案的平均 token 数。计算公式为:

$$\text{Avg Length} = \frac{1}{N} \sum_{i=1}^N |s_i|$$

更短的答案意味着更低的推理成本和更快的响应速度。

推理步骤数(Reasoning Steps):答案中包含的推理步骤数量。计算公式为:

$$\text{Avg Steps} = \frac{1}{N} \sum_{i=1}^N s_i$$

适当的步骤数(2-5 步)说明模型能够系统地分解问题，过多的步骤可能说明推理冗余。

推理时间(Inference Time):生成一个答案所需的时间。这个指标在实际部署中很重要，影响用户体验。

(3) 质量指标

质量指标衡量答案的可读性和可解释性。

格式正确率(Format Correctness):答案是否符合预期格式(如包含“Step 1”，“Final Answer”等标记)。计算公式为: $\frac{\text{格式正确的答案数}}{\text{总答案数}}$

格式正确是基本要求，格式混乱的答案即使结果正确也难以使用。

推理连贯性(Reasoning Coherence):推理步骤之间是否逻辑连贯。这个指标通常需要人工评估或使用专门的评估模型。

可解释性(Explainability):答案是否容易理解和验证。包含清晰步骤的答案比直接给出结果的答案更具可解释性。

如表 11.7 所示，不同指标的对比。

表 11.7 评估指标对比



11.5.2 评估实战

HelloAgents 提供了全面的评估功能，可以一次性计算多个指标。

```
from hello_agents.tools import RLTrainingTool
rl_tool = RLTrainingTool()

# 全面评估
print("=" * 50)
print("全面评估GRPO模型")
print("=" * 50)

result = rl_tool.run({
    "action": "evaluate",
    "model_path": "./models/grpo_full",
    "max_samples": 200,
    "use_lora": True,
}

# 评估配置
"metrics": [
    "accuracy",          # 准确率
    "accuracy_at_k",     # Top-K准确率
    "average_length",    # 平均长度
    "average_steps",     # 平均步骤数
    "format_correctness", # 格式正确率
],
"k": 3,  # Top-3准确率
})

# 解析结果
eval_data = json.loads(result)

# 打印结果
print(f"\n评估结果:")
print(f"  准确率: {eval_data['accuracy']}")
print(f"  平均奖励: {eval_data['average_reward']}")
print(f"  测试样本数: {eval_data['num_samples']}
```

我们可以对比预训练模型、SFT 模型、GRPO 模型的性能:

```
# 评估三个模型
models = [
    ("预训练模型", "Qwen/Qwen3-0.6B", False),
    ("SFT模型", "./models/sft_full", True),
    ("GRPO模型", "./models/grpo_full", True),
]

results = []
for name, path, use_lora in models:
    print(f"\n评估{name}...")
    result = rl_tool.run({
        "action": "evaluate",
        "model_path": path,
        "max_samples": 200,
        "use_lora": use_lora,
        "metrics": ["accuracy", "average_length", "format_correctness"],
    })
    results.append((name, result))

# 打印对比表格
print("\n" + "=" * 70)
print(f'{{"模型":<15} {"准确率":<12} {"平均长度":<15} {"格式正确率":<12}}')
print("=" * 70)
for name, result in results:
    print(f'{{"name":<15} {"result[\'accuracy\']:<12.2%} {"result[\'average_length\']:<15.1f} {"result[\'format_correctness\']:<12.2%}}')
print("=" * 70)
```

11.5.3 错误分析

仅仅知道准确率是不够的，我们需要深入分析模型在哪些类型的问题上容易出错，从而指导后续改进。模型的错误可以分为四类：计算错误（推理步骤正确但计算出错，如“ $48/2=25$ ”，说明数值计算能力不足）、推理错误（推理逻辑错误导致解题思路不对，如先加后除而非先除后加，说明逻辑推理能力不足）、理解错误（没有正确理解问题，如问题问“总共”但只计算了一部分，说明语言理解能力不足）、格式错误（答案正确但格式不符合要求，如缺少“Final Answer”标记，说明格式学习不足）。

错误分析示例：

```
from hello_agents.tools import RLTrainingTool
rl_tool = RLTrainingTool()

# 评估并收集错误样本
result = rl_tool.run({
    "action": "evaluate",
    "model_path": "./models/grpo_full",
    "max_samples": 200,
    "use_lora": True,
    "return_details": True, # 返回详细结果
})

# 分析错误样本
errors = result['errors'] # 错误样本列表
print(f"总错误数: {len(errors)}")

# 按错误类型分类
error_types = {
    "计算错误": 0,
    "推理错误": 0,
    "理解错误": 0,
    "格式错误": 0,
}

for error in errors:
    question = error['question']
    prediction = error['prediction']
    ground_truth = error['ground_truth']

    # 简单的错误分类逻辑（实际应用中可能需要更复杂的分析）
    if "Final Answer:" not in prediction:
        error_types["格式错误"] += 1
    elif "Step" in prediction:
        # 有推理步骤，可能是计算或推理错误
        # 这里需要更细致的分析
        error_types["计算错误"] += 1
    else:
        error_types["理解错误"] += 1

# 打印错误分布
print("\n错误类型分布:")
for error_type, count in error_types.items():
    percentage = count / len(errors) * 100
    print(f' {error_type}: {count} ({percentage:.1f}%)')
```

输出示例：

总错误数: 76

错误类型分布:

- 计算错误: 32 (42.1%)
- 推理错误: 18 (23.7%)
- 理解错误: 22 (28.9%)
- 格式错误: 4 (5.3%)

可以看到, 计算错误是最主要的错误类型(42.1%), 说明模型的数值计算能力需要加强。格式错误很少(5.3%), 说明 SFT 训练效果良好。我们还可以分析模型在不同难度的问题上的表现:

```
# 按推理步骤数分组
step_groups = {
    "简单(1-2步)": [],
    "中等(3-4步)": [],
    "困难(5+步)": []
}

for sample in result['details']:
    steps = sample['ground_truth_steps'] # 真实答案的步骤数
    correct = sample['correct']

    if steps <= 2:
        step_groups["简单(1-2步)"].append(correct)
    elif steps <= 4:
        step_groups["中等(3-4步)"].append(correct)
    else:
        step_groups["困难(5+步)"].append(correct)

# 计算每组的准确率
print("\n不同难度的准确率:")
for group_name, results in step_groups.items():
    if len(results) > 0:
        accuracy = sum(results) / len(results)
        print(f" {group_name}: {accuracy:.2%} ({len(results)}个样本)")


输出示例:
```

不同难度的准确率:

- 简单(1-2步): 78.50% (85个样本)
- 中等(3-4步): 58.30% (96个样本)
- 困难(5+步): 31.60% (19个样本)

可以看到, 模型在简单问题上表现良好(78.5%), 但在困难问题上表现较差(31.6%)。这说明模型的多步推理能力还有待提升

11.5.4 改进方向

基于评估和分析结果, 我们可以确定模型的改进方向, 如图 11.8 所示。



图 11.8 模型改进迭代流程

这是一个持续迭代的过程:训练模型 → 评估性能 → 分析错误 → 确定问题 → 选择改进方向 → 重新训练。通过多次迭代, 模型性能会不断提升。

11.6 完整训练流程实战

在前面的章节中, 我们分别学习了数据准备、SFT 训练、GRPO 训练和模型评估。现在, 让我们把这些知识整合起来, 完成一个端到端的 Agentic RL 训练流程。

11.6.1 端到端训练流程

一个完整的 Agentic RL 训练流程包括以下阶段:数据准备、SFT 训练、SFT 评估、GRPO 训练、GRPO 评估、模型部署。如图 11.9 所示。



图 11.9 端到端训练流程

让我们通过一个完整的脚本来实现这个流程:

```
"""
完整的Agentic RL训练流程
从数据准备到模型部署的端到端示例
"""

from hello_agents.tools import RLTrainingTool
import json
from datetime import datetime

class AgenticRLPipeline:
    """Agentic RL训练流水线"""

    def __init__(self, config_path="config.json"):

        self.config = json.load(open(config_path))
        self.rl_tool = RLTrainingTool(self.config)
```

初始化训练流水线

```

Args:
    config_path: 配置文件路径
    ...
self.rl_tool = RLTrainingTool()
self.config = self.load_config(config_path)
self.results = {}

def load_config(self, config_path):
    """加载配置文件"""
    with open(config_path, 'r') as f:
        return json.load(f)

def log(self, message):
    """记录日志"""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    print(f"[{timestamp}] {message}")

def stage1_prepare_data(self):
    """阶段1: 数据准备"""
    self.log("=" * 50)
    self.log("阶段1: 数据准备")
    self.log("=" * 50)

    # 加载并检查数据集
    result = self.rl_tool.run({
        "action": "load_dataset",
        "format": "sft",
        "max_samples": self.config["data"]["max_samples"],
    })

    # 解析JSON结果
    dataset_info = json.loads(result)

    self.log(f"✓ 数据集加载完成")
    self.log(f" - 样本数: {dataset_info['dataset_size']}")
    self.log(f" - 格式: {dataset_info['format']}")
    self.log(f" - 数据列: {', '.join(dataset_info['sample_keys'])}")

    self.results["data"] = dataset_info

    return dataset_info

def stage2_sft_training(self):
    """阶段2: SFT训练"""
    self.log("\n" + "=" * 50)
    self.log("阶段2: SFT训练")
    self.log("=" * 50)

    sft_config = self.config["sft"]

    result = self.rl_tool.run({
        "action": "train",
        "algorithm": "sft",
        "model_name": self.config["model"]["base_model"],
        "output_dir": sft_config["output_dir"],
        "max_samples": self.config["data"]["max_samples"],
        "num_epochs": sft_config["num_epochs"],
        "batch_size": sft_config["batch_size"],
        "use_lora": True,
        # 训练监控配置
        "use_wandb": self.config.get("monitoring", {}).get("use_wandb", False),
        "use_tensorboard": self.config.get("monitoring", {}).get("use_tensorboard", True),
        "wandb_project": self.config.get("monitoring", {}).get("wandb_project", None),
    })

    # 解析JSON结果
    result_data = json.loads(result)

    self.log(f"✓ SFT训练完成")
    self.log(f" - 模型路径: {result_data['output_dir']}")
    self.log(f" - 状态: {result_data['status']}")

    self.results["sft_training"] = result_data

    return result_data["output_dir"]

def stage3_sft_evaluation(self, model_path):
    """阶段3: SFT评估"""
    self.log("\n" + "=" * 50)
    self.log("阶段3: SFT评估")
    self.log("=" * 50)

    result = self.rl_tool.run({
        "action": "evaluate",
        "model_path": model_path,
        "max_samples": self.config["eval"]["max_samples"],
        "use_lora": True,
    })
    eval_data = json.loads(result)

    self.log(f"✓ SFT评估完成")

```

```

        self.log(f" - 准确率: {eval_data['accuracy']}")
        self.log(f" - 平均奖励: {eval_data['average_reward']}")

    self.results["sft_evaluation"] = eval_data
    return eval_data

def stage4_grpo_training(self, sft_model_path):
    """阶段4: GRPO训练"""
    self.log("\n" + "=" * 50)
    self.log("阶段4: GRPO训练")
    self.log("=" * 50)

    grpo_config = self.config["grpo"]

    result = self.rl_tool.run({
        "action": "train",
        "algorithm": "grpo",
        "model_name": sft_model_path,
        "output_dir": grpo_config["output_dir"],
        "max_samples": self.config["data"]["max_samples"],
        "num_epochs": grpo_config["num_epochs"],
        "batch_size": grpo_config["batch_size"],
        "use_lora": True,
        # 训练监控配置
        "use_wandb": self.config.get("monitoring", {}).get("use_wandb", False),
        "use_tensorboard": self.config.get("monitoring", {}).get("use_tensorboard", True),
        "wandb_project": self.config.get("monitoring", {}).get("wandb_project", None),
    })

    # 解析JSON结果
    result_data = json.loads(result)

    self.log(f"✓ GRPO训练完成")
    self.log(f" - 模型路径: {result_data['output_dir']}")
    self.log(f" - 状态: {result_data['status']}")

    self.results["grpo_training"] = result_data
    return result_data["output_dir"]

def stage5_grpo_evaluation(self, model_path):
    """阶段5: GRPO评估"""
    self.log("\n" + "=" * 50)
    self.log("阶段5: GRPO评估")
    self.log("=" * 50)

    result = self.rl_tool.run({
        "action": "evaluate",
        "model_path": model_path,
        "max_samples": self.config["eval"]["max_samples"],
        "use_lora": True,
    })
    eval_data = json.loads(result)

    self.log(f"✓ GRPO评估完成")
    self.log(f" - 准确率: {eval_data['accuracy']}")
    self.log(f" - 平均奖励: {eval_data['average_reward']}")

    self.results["grpo_evaluation"] = eval_data
    return eval_data

def stage6_save_results(self):
    """阶段6: 保存结果"""
    self.log("\n" + "=" * 50)
    self.log("阶段6: 保存结果")
    self.log("=" * 50)

    # 保存训练结果
    results_path = "training_results.json"
    with open(results_path, 'w') as f:
        json.dump(self.results, f, indent=2)

    self.log(f"✓ 结果已保存到: {results_path}")

def run(self):
    """运行完整流程"""
    try:
        # 阶段1: 数据准备
        self.stage1_prepare_data()

        # 阶段2: SFT训练
        sft_model_path = self.stage2_sft_training()

        # 阶段3: SFT评估
        self.stage3_sft_evaluation(sft_model_path)

        # 阶段4: GRPO训练
        grpo_model_path = self.stage4_grpo_training(sft_model_path)

        # 阶段5: GRPO评估
        self.stage5_grpo_evaluation(grpo_model_path)
    
```

```

# 阶段6: 保存结果
self.stage6_save_results()

self.log("\n" + "=" * 50)
self.log("✓ 训练流程完成!")
self.log("=" * 50)

except Exception as e:
    self.log(f"\nx 训练失败: {str(e)}")
    raise

# 使用示例
if __name__ == "__main__":
    # 创建配置文件
    config = {
        "model": {
            "base_model": "Owen/Qwen3-0.6B"
        },
        "data": {
            "max_samples": 1000 # 使用1000个样本
        },
        "sft": {
            "output_dir": "./models/sft_model",
            "num_epochs": 3,
            "batch_size": 8,
        },
        "grpo": {
            "output_dir": "./models/grpo_model",
            "num_epochs": 3,
            "batch_size": 4,
        },
        "eval": {
            "max_samples": 200,
            "sft_accuracy_threshold": 0.40 # SFT准确率阈值
        },
        "monitoring": {
            "use_wandb": False, # 是否使用Wandb
            "use_tensorboard": True, # 是否使用TensorBoard
            "wandb_project": "agentic-rl-pipeline" # Wandb项目名
        }
    }

# 保存配置
with open("config.json", 'w') as f:
    json.dump(config, f, indent=2)

# 运行训练流程
pipeline = AgenticRLPipeline("config.json")
pipeline.run()

```

运行这个脚本，你将看到完整的训练过程。

运行小建议：

从小规模开始:不要一开始就用全部数据训练。先用 100-1000 个样本快速迭代，验证流程和参数，确认效果后再扩大规模。这样可以节省大量时间和计算资源。

数据质量检查:在训练前检查数据质量，确保格式正确、答案准确、没有重复样本。可以使用以下代码:

```

def check_data_quality(dataset):
    """检查数据质量"""
    issues = []

    # 检查必需字段
    required_fields = ["prompt", "completion"]
    for field in required_fields:
        if field not in dataset.column_names:
            issues.append(f"缺少字段: {field}")

    # 检查空值
    for i, sample in enumerate(dataset):
        if not sample["prompt"] or not sample["completion"]:
            issues.append(f"样本{i}包含空值")

    # 检查重复
    prompts = [s["prompt"] for s in dataset]
    duplicates = len(prompts) - len(set(prompts))
    if duplicates > 0:
        issues.append(f"发现{duplicates}个重复样本")

    return issues

# 使用
issues = check_data_quality(dataset)
if issues:
    print("数据质量问题:")
    for issue in issues:
        print(f" - {issue}")
else:
    print("✓ 数据质量检查通过")

```

数据增强:如果数据量不足,可以考虑数据增强,如改写问题(保持答案不变)、生成相似问题、反向翻译(translate back)。但要注意保持数据质量,避免引入噪声。

11.6.2 超参数调优

超参数调优是提升模型性能的关键。下面是一些常用的调优策略。

(1) 网格搜索

网格搜索(Grid Search)是最简单的调优方法,遍历所有参数组合,选择最佳的一组。

```
# 定义参数网格
param_grid = {
    "learning_rate": [1e-5, 5e-5, 1e-4],
    "lora_rank": [8, 16, 32],
    "kl_coef": [0.05, 0.1, 0.2],
}

best_accuracy = 0
best_params = None

# 遍历所有组合
for lr in param_grid["learning_rate"]:
    for rank in param_grid["lora_rank"]:
        for kl in param_grid["kl_coef"]:
            print(f"测试参数: lr={lr}, rank={rank}, kl={kl}")

            # 训练模型
            result = rl_tool.run({
                "action": "train",
                "algorithm": "grpo",
                "learning_rate": lr,
                "lora_rank": rank,
                "kl_coef": kl,
                # 其他参数...
            })

            # 评估模型
            eval_result = rl_tool.run({
                "action": "evaluate",
                "model_path": result["model_path"],
            })

            # 更新最佳参数
            if eval_result["accuracy"] > best_accuracy:
                best_accuracy = eval_result["accuracy"]
                best_params = {"lr": lr, "rank": rank, "kl": kl}

print(f"最佳参数: {best_params}")
print(f"最佳准确率: {best_accuracy:.2%}")
```

网格搜索的优点是简单直接,能找到全局最优。缺点是计算成本高,参数多时不可行。

(2) 随机搜索

随机搜索(Random Search)随机采样参数组合,比网格搜索更高效。

```
import random

# 定义参数范围
param_ranges = {
    "learning_rate": (1e-6, 1e-4), # 对数均匀分布
    "lora_rank": [4, 8, 16, 32, 64],
    "kl_coef": (0.01, 0.5),
}

best_accuracy = 0
best_params = None

# 随机采样N次
N = 10
for i in range(N):
    # 随机采样参数
    lr = 10 ** random.uniform(-6, -4) # 对数均匀
    rank = random.choice(param_ranges["lora_rank"])
    kl = random.uniform(0.01, 0.5)

    print(f"[{i+1}/{N}] 测试参数: lr={lr:.2e}, rank={rank}, kl={kl:.3f}")

    # 训练和评估(同上)
    # ...

print(f"最佳参数: {best_params}")
print(f"最佳准确率: {best_accuracy:.2%}")
```

随机搜索的优点是效率高,适合参数空间大的情况。缺点是可能错过最优解。

(3) 贝叶斯优化

贝叶斯优化(Bayesian Optimization)使用概率模型指导搜索，更加智能。可以使用 Optuna 等库：

```
import optuna

def objective(trial):
    """优化目标函数"""
    # 采样参数
    lr = trial.suggest_loguniform("learning_rate", 1e-6, 1e-4)
    rank = trial.suggest_categorical("lora_rank", [8, 16, 32])
    kl = trial.suggest_uniform("kl_coef", 0.01, 0.5)

    # 训练模型
    result = rl_tool.run({
        "action": "train",
        "algorithm": "grpo",
        "learning_rate": lr,
        "lora_rank": rank,
        "kl_coef": kl,
        # 其他参数...
    })

    # 评估模型
    eval_result = rl_tool.run({
        "action": "evaluate",
        "model_path": result["model_path"],
    })

    return eval_result["accuracy"]

# 创建研究
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=20)

# 打印最佳参数
print(f"最佳参数: {study.best_params}")
print(f"最佳准确率: {study.best_value:.2%}")
```

贝叶斯优化的优点是样本效率高，能快速找到好的参数。缺点是实现复杂，需要额外的库。

如表 11.8 所示，不同调优方法的对比。

表 11.8 超参数调优方法对比



11.6.3 分布式训练

当数据量和模型规模增大时，单 GPU 训练会变得非常缓慢。这时我们需要使用分布式训练来加速训练过程。HelloAgents 基于 TRL 和 Hugging Face Accelerate，天然支持多 GPU 和多节点分布式训练

方案选择建议：

- 单机多卡(2-8 卡): 使用 DDP，简单高效
- 大模型(>7B): 使用 DeepSpeed ZeRO-2 或 ZeRO-3
- 多节点集群: 使用 DeepSpeed ZeRO-3 + Offload

(1) 配置 Accelerate

首先需要创建 Accelerate 配置文件。运行以下命令：

```
accelerate config
```

根据提示选择配置：

```
In which compute environment are you running?
> This machine
```

```
Which type of machine are you using?
> multi-GPU
```

```
How many different machines will you use?
> 1
```

```
Do you wish to optimize your script with torch dynamo?
> NO
```

```
Do you want to use DeepSpeed?
> YES
```

```
Which DeepSpeed config file do you want to use?
> ZeRO-2
```

```
How many GPU(s) should be used for distributed training?
> 4
```

这会在`~/.cache/huggingface/accelerate/default_config.yaml`生成配置文件。

(2) 使用 DDP 训练

数据并行(DDP)是最简单的分布式方案，每个 GPU 持有完整模型副本，数据被分割到各个 GPU 上。

Accelerate 配置文件 (multi_gpu_ddp.yaml):

```
compute_environment: LOCAL_MACHINE
distributed_type: MULTI_GPU
num_processes: 4 # GPU数量
machine_rank: 0
num_machines: 1
gpu_ids: all
mixed_precision: fp16
```

训练脚本(无需修改):

```
from hello_agents.tools import RLTrainingTool
rl_tool = RLTrainingTool()

# 训练代码完全不变
result = rl_tool.run({
    "action": "train",
    "algorithm": "grpo",
    "model_name": "Qwen/Qwen3-0.6B",
    "output_dir": "./models/grpo_ddp",
    "num_epochs": 3,
    "batch_size": 4, # 每个GPU的batch size
    "use_lora": True,
})
```

启动训练:

```
# 使用配置文件
accelerate launch --config_file multi_gpu_ddp.yaml train_script.py

# 或者直接指定参数
accelerate launch --num_processes 4 --mixed_precision fp16 train_script.py
```

(3) 使用 DeepSpeed ZeRO 训练

DeepSpeed ZeRO通过分片优化器状态、梯度和模型参数，大幅降低显存占用，支持更大的模型和 batch size。

ZeRO-2 配置文件 (deepspeed_zero2.yaml):

```
compute_environment: LOCAL_MACHINE
distributed_type: DEEPSPEED
num_processes: 4
machine_rank: 0
num_machines: 1
gpu_ids: all
mixed_precision: fp16
deepspeed_config:
    gradient_accumulation_steps: 4
    gradient_clipping: 1.0
    offload_optimizer_device: none
    offload_param_device: none
    zero3_init_flag: false
    zero_stage: 2 # ZeRO-2
```

ZeRO-3 配置文件 (deepspeed_zero3.yaml):

```
compute_environment: LOCAL_MACHINE
distributed_type: DEEPSPEED
num_processes: 4
machine_rank: 0
num_machines: 1
gpu_ids: all
mixed_precision: fp16
deepspeed_config:
    gradient_accumulation_steps: 4
    gradient_clipping: 1.0
    offload_optimizer_device: cpu # 优化器状态卸载到CPU
    offload_param_device: cpu # 参数卸载到CPU
    zero3_init_flag: true
    zero_stage: 3 # ZeRO-3
```

启动训练:

```
# ZeRO-2
accelerate launch --config_file deepspeed_zero2.yaml train_script.py

# ZeRO-3
accelerate launch --config_file deepspeed_zero3.yaml train_script.py
```

如表 11.9 所示，这是 Qwen3-0.6B 模型用不同方式训练的显存对比:

表 11.9 显存对比 (Qwen3-0.6B 模型)



(4) 多节点训练

对于超大规模训练，可以使用多个节点(机器)。

主节点配置 (multi_node_main.yaml):

```
compute_environment: LOCAL_MACHINE
distributed_type: DEEPSPEED
num_processes: 16 # 4节点 x 4GPU
machine_rank: 0 # 主节点
num_machines: 4
main_process_ip: 192.168.1.100 # 主节点IP
main_process_port: 29500
gpu_ids: all
mixed_precision: fp16
deepspeed_config:
  zero_stage: 3
  offload_optimizer_device: cpu
  offload_param_device: cpu
```

工作节点配置 (修改machine_rank为 1, 2, 3):

```
machine_rank: 1 # 工作节点1
# 其他配置相同
```

启动训练:

```
# 在主节点上
accelerate launch --config_file multi_node_main.yaml train_script.py

# 在工作节点1上
accelerate launch --config_file multi_node_worker1.yaml train_script.py

# 在工作节点2上
accelerate launch --config_file multi_node_worker2.yaml train_script.py

# 在工作节点3上
accelerate launch --config_file multi_node_worker3.yaml train_script.py
```

(5) 分布式训练最佳实践

1. Batch Size 调整

分布式训练时，总 batch size = per_device_batch_size × num_gpus × gradient_accumulation_steps

```
# 单GPU: batch_size=4, gradient_accumulation=4, 总batch=16
# 4GPU DDP: batch_size=4, gradient_accumulation=1, 总batch=16 (保持一致)
```

2. 学习率缩放

使用线性缩放规则: lr_new = lr_base × sqrt(total_batch_size_new / total_batch_size_base)

```
# 基准: 单GPU, batch=16, lr=5e-5
# 4GPU: batch=64, lr=5e-5 × sqrt(64/16) = 1e-4
```

3. 监控和调试

```
# 启用详细日志
export ACCELERATE_LOG_LEVEL=INFO

# 启用NCCL调试(多节点)
export NCCL_DEBUG=INFO

# 检查GPU利用率
watch -n 1 nvidia-smi
```

11.6.4 生产部署

训练完成后，我们需要将模型部署到生产环境。下面是一些部署建议。

(1) 模型导出

将 LoRA 权重合并到基础模型，方便部署:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import PeftModel

# 加载基础模型
base_model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen3-0.6B")

# 加载LoRA权重
model = PeftModel.from_pretrained(base_model, "./models/grpo_model")
```

2/9/26, 1:56 PM

```
# 合并权重
merged_model = model.merge_and_unload()

# 保存合并后的模型
merged_model.save_pretrained("./models/merged_model")

# 保存tokenizer
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen3-0.6B")
tokenizer.save_pretrained("./models/merged_model")

print("\v 模型已导出到: ./models/merged_model")
```

(2) 推理优化

使用量化和优化技术加速推理:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

# 加载模型(使用8-bit量化)
model = AutoModelForCausalLM.from_pretrained(
    "./models/merged_model",
    load_in_8bit=True, # 8-bit量化
    device_map="auto", # 自动分配设备
)

tokenizer = AutoTokenizer.from_pretrained("./models/merged_model")

# 推理
def generate_answer(question):
    prompt = f"<|im_start|>user\n{question}<|im_end|>\n<|im_start|>assistant\n"
    inputs = tokenizer(prompt, return_tensors="pt").to(model.device)

    outputs = model.generate(
        **inputs,
        max_new_tokens=512,
        temperature=0.7,
        do_sample=True,
    )

    response = tokenizer.decode(outputs[0], skip_special_tokens=False)
    return response

# 测试
question = "What is 48 + 24?"
answer = generate_answer(question)
print(answer)
```

(3) API 服务

使用 FastAPI 创建推理服务:

```
from fastapi import FastAPI
from pydantic import BaseModel
from transformers import AutoModelForCausalLM, AutoTokenizer

app = FastAPI()

# 加载模型
model = AutoModelForCausalLM.from_pretrained("./models/merged_model")
tokenizer = AutoTokenizer.from_pretrained("./models/merged_model")

class Question(BaseModel):
    text: str
    max_tokens: int = 512

class Answer(BaseModel):
    text: str
    confidence: float

@app.post("/generate", response_model=Answer)
def generate(question: Question):
    """生成答案"""
    prompt = f"<|im_start|>user\n{question.text}<|im_end|>\n<|im_start|>assistant\n"
    inputs = tokenizer(prompt, return_tensors="pt")

    outputs = model.generate(
        **inputs,
        max_new_tokens=question.max_tokens,
        temperature=0.7,
        return_dict_in_generate=True,
        output_scores=True,
    )

    response = tokenizer.decode(outputs.sequences[0], skip_special_tokens=False)

    # 计算置信度(简化版)
    confidence = 0.8 # 实际应该基于输出概率计算

    return Answer(text=response, confidence=confidence)

# 运行: uvicorn api:app --host 0.0.0.0 --port 8000
```

11.8 本章小结

在本章中，我们系统地学习了 Agentic RL 的理论和实践，从基础概念到完整的训练流程，从数据准备到模型部署。让我们回顾一下本章的主要内容。

(1) Agentic RL 的本质

Agentic RL 是将 LLM 作为可学习策略，嵌入到智能体的感知-决策-执行循环中，通过强化学习优化智能体在多步任务中的表现。它与传统的 PBRFT(Preference-Based Reinforcement Fine-Tuning)的核心区别在于：

- 任务性质:从单轮对话优化扩展到多步序贯决策
- 状态空间:从静态提示扩展到动态演化的环境状态
- 行动空间:从纯文本生成扩展到文本+工具+环境操作
- 奖励设计:从单步质量评估扩展到长期累积回报
- 优化目标:从短期响应质量扩展到长期任务成功

(2) 六大核心能力

Agentic RL 旨在提升智能体的六大核心能力：

1. 推理(**Reasoning**):多步逻辑推导，学习推理策略
2. 工具使用(**Tool Use**):API/工具调用，学会何时用、如何用
3. 记忆(**Memory**):长期信息保持，学习记忆管理
4. 规划(**Planning**):行动序列规划，学会动态规划
5. 自我改进(**Self-Improvement**):自我反思优化，从错误中学习
6. 感知(**Perception**):多模态理解，视觉推理和工具使用

(3) 训练流程

完整的 Agentic RL 训练流程包括：

1. 预训练(**Pretraining**):在大规模文本上学习语言知识(通常使用现成的预训练模型)
2. 监督微调(**SFT**):学习任务格式和基础推理能力
3. 强化学习(**RL**):通过试错优化推理策略，超越训练数据质量

其中，SFT 是基础，RL 是提升。没有 SFT 的基础，RL 很难成功;没有 RL 的优化，模型只能模仿训练数据。

如果你想深入学习 Agentic RL，建议按照以下路径：

基础阶段

1. 强化学习基础:学习 MDP、策略梯度、PPO 等基本概念
2. LLM 基础:了解 Transformer、预训练、微调等技术
3. 实践 **HelloAgents**:运行本章的示例代码，理解完整流程

进阶阶段

1. 深入 TRL:学习 TRL 库的实现，理解 SFT 和 GRPO 等算法的细节
2. 自定义数据集:使用自己的数据集训练模型
3. 自定义奖励函数:设计适合自己任务的奖励函数
4. 参数调优:系统地调优超参数，提升模型性能

高级阶段

1. 多步推理:研究长序列推理任务
2. 工具学习:让智能体学会使用工具
3. 多智能体:研究多智能体协作
4. 前沿论文:阅读最新的研究论文，跟进前沿进展

希望本章能够帮助你理解和掌握 Agentic RL 技术，在自己的项目中应用这些知识，构建更智能的 Agent 系统！

参考文献

- [1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.
- [2] Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., ... & Guo, D. (2024). DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*.
- [3] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., ... & Chen, W. (2021). LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*.
- [4] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., ... & Schulman, J. (2021). Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*.

- [5] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., ... & Lowe, R. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35, 27730-27744.
- [6] Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D., & Finn, C. (2023). Direct Preference Optimization: Your Language Model is Secretly a Reward Model. *arXiv preprint arXiv:2305.18290*.
- [7] Lee, H., Phatale, S., Mansoor, H., Lu, K., Mesnard, T., Bishop, C., ... & Rastogi, A. (2023). RLAIF: Scaling Reinforcement Learning from Human Feedback with AI Feedback. *arXiv preprint arXiv:2309.00267*.
- [8] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., ... & Zhou, D. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems*, 35, 24824-24837.
- [9] von Werra, L., Belkada, Y., Tunstall, L., Beeching, E., Thrush, T., Lambert, N., & Huang, S. (2020). TRL: Transformer Reinforcement Learning. *GitHub repository*. <https://github.com/huggingface/trl>
- [10] Qwen Team. (2025). Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388*.
- [11] Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., DasSarma, N., ... & Kaplan, J. (2022). Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. *arXiv preprint arXiv:2204.05862*.
- [12] Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., ... & Zhou, D. (2022). Self-Consistency Improves Chain of Thought Reasoning in Language Models. *arXiv preprint arXiv:2203.11171*.
- [13] Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). Deep Reinforcement Learning from Human Preferences. *Advances in Neural Information Processing Systems*, 30.
- [14] Stiennon, N., Ouyang, L., Wu, J., Ziegler, D., Lowe, R., Voss, C., ... & Christiano, P. F. (2020). Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33, 3008-3021.
- [15] Ziegler, D. M., Stiennon, N., Wu, J., Brown, T. B., Radford, A., Amodei, D., ... & Irving, G. (2019). Fine-Tuning Language Models from Human Preferences. *arXiv preprint arXiv:1909.08593*.

习题

提示：部分习题没有标准答案，重点在于培养学习者对 Agentic RL 和智能体训练的综合理解和实践能力。

1. 本章介绍了从 LLM 训练到 Agentic RL 的演进过程。请分析：

- 在 11.1.3 节的表 11.1 中，对比了 PBRFT（基于偏好的强化微调）和 Agentic RL 在 MDP 框架下的差异。请深入解释：为什么 Agentic RL 的状态空间 $s_t = (\text{prompt}, o_1, o_2, \dots, o_t)$ 包含历史观察，而 PBRFT 的状态 $s_0 = \text{prompt}$ 只包含初始提示？这种差异对训练过程和最终效果有什么影响？
- 假设你要训练一个“智能代码调试助手”，它需要：(1) 分析代码找出 bug；(2) 查阅文档了解 API 用法；(3) 修改代码；(4) 运行测试验证修复效果。请将这个任务映射到强化学习框架，明确定义状态空间、行动空间、奖励函数和状态转移函数。
- 在 11.1.1 节中提到，传统监督学习存在“难以优化长期目标”的局限。请设计一个具体的多步推理任务（如数学证明、复杂问题求解），展示为什么监督学习难以优化中间步骤，而强化学习可以通过延迟奖励来解决这个问题。

2. SFT（监督微调）和 GRPO（群组相对策略优化）是本章的两个核心训练方法。基于 11.2 节和 11.3 节的内容，请深入思考：

提示：这是一道动手实践题，建议实际操作

- 在 11.2.4 节的 SFT 训练代码中，我们使用了 LoRA（低秩适配）技术来减少训练参数。请分析：LoRA 的核心思想是什么？为什么它能够用少量参数（如 0.16%）实现接近全参数微调的效果？在什么情况下应该选择 LoRA 而不是全参数微调？
- GRPO 算法（11.3 节）相比传统的 PPO 算法有什么优势？请对比两者的训练流程，分析 GRPO 如何通过“群组相对奖励”来简化训练过程并提升稳定性。如果要将 GRPO 应用到其他任务（如代码生成、对话优化），需要做哪些调整？
- 请基于 11.2.5 节的代码，扩展 SFT 训练流程，添加以下功能：(1) 支持多轮对话数据的训练；(2) 添加数据增强策略（如同义改写、难度调整）；(3) 实现训练过程的可视化监控（如 loss 曲线、样本质量评估）。

3. 奖励函数设计是 Agentic RL 的核心挑战。基于 11.3.3 节的内容，请完成以下扩展实践：

提示：这是一道动手实践题，建议实际操作

- 在 11.3.3 节中，我们为 GSM8K 数学问题设计了简单的二元奖励（正确+1，错误 0）。请设计一个更精细的奖励函数，能够：(1) 对部分正确的答案给予部分奖励；(2) 对推理过程的合理性进行评分；(3) 惩罚过于冗长或低效的解题路径。这个奖励函数应该如何实现？
- 奖励函数的设计往往需要领域知识。请为以下三个不同的智能体任务设计奖励函数：(1) 代码生成助手（需要考虑代码正确性、可读性、效率）；(2) 客服对话智能体（需要考虑问题解决率、用户满意度、响应时间）；(3) 游戏 AI（需要考虑胜率、策略多样性、对抗鲁棒性）。
- 在实际应用中，奖励函数可能存在“奖励黑客”（reward hacking）问题：智能体找到了获得高奖励的捷径，但并没有真正完成任务。请举例说明这种现象，并设计防御机制来避免奖励黑客。

4. 在 11.4 节的“数学推理智能体训练”案例中，我们看到了完整的训练流程。请深入分析：

- 案例中使用了 GSM8K 数据集进行训练和评估。请分析：这个数据集的特点是什么？它适合训练什么类型的推理能力？如果要训练一个能够处理更复杂数学问题（如高等数学、数学证明）的智能体，应该如何扩展数据集和训练方法？
- 在 11.4.3 节的训练结果中，我们观察到模型在训练集上的准确率提升，但可能存在过拟合风险。请设计一个“泛化能力评估”方案：如何测试模型是否真正学会了数学推理，而不是记住了训练数据？如何通过正则化、数据增强等技术提升泛化能力？
- 案例中的训练是离线的（使用预先收集的数据集）。请设计一个“在线学习”方案：智能体在实际使用过程中持续收集用户反馈，并自动更新模型。这个方案需要考虑哪些技术挑战（如数据质量控制、灾难性遗忘、安全性保障）？

5. Agentic RL 的一个重要应用是让智能体学会使用工具。请思考：

- 在 11.1.3 节中提到，Agentic RL 适合优化”需要多步推理、工具使用、长期规划”的任务。请设计一个”工具学习”训练方案：给定一组工具（如搜索引擎、计算器、代码执行器），如何训练智能体学会在合适的时机选择合适的工具？奖励函数应该如何设计？
- 工具使用往往涉及复杂的依赖关系（如”必须先调用工具 A 获取信息，才能调用工具 B”）。请设计一个”分层强化学习”方案：高层策略负责任务规划，低层策略负责工具调用。这种分层结构如何训练？如何协调高层和低层的优化目标？
- 在实际应用中，工具的数量可能非常多（如 50+ 个 API），直接训练可能面临”探索效率低”的问题。请设计一个”课程学习”（curriculum learning）方案：从简单任务（使用少量工具）开始训练，逐步增加任务难度和工具数量。这个方案应该如何设计课程顺序？如何评估智能体是否准备好进入下一阶段？

第十二章 智能体性能评估

在前面的章节中，我们构建了 HelloAgents 框架的核心功能，实现了多种智能体范式、工具系统、记忆机制和强化学习训练等。在构建智能体系统时，我们还需要解决一个核心问题：如何客观地评估智能体的性能？具体来说，我们需要回答以下问题：

1. 智能体是否具备预期的能力？
2. 在不同任务上的表现如何？
3. 与其他智能体相比处于什么水平？

本章将为 HelloAgents 增加性能评估系统（Evaluation System）。我们将深入理解智能体评估的理论基础，并实现评估的工具。

12.1 智能体评估基础

12.1.1 为何需要智能体评估

我们现在的 SimpleAgent，它已经具备了强大的推理和工具调用能力。让我们看一个典型的使用场景：

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import SearchTool
```

```
# 创建LLM和智能体
llm = HelloAgentsLLM()
```

```
# 创建一个强调工具使用的系统提示词
system_prompt = """你是一个AI助手，可以使用搜索工具来获取最新信息。
```

当需要搜索信息时，请使用以下格式：

```
[TOOL_CALL:search:搜索关键词]
```

例如：

- [TOOL_CALL:search:最新AI新闻]
- [TOOL_CALL:search:Python编程教程]

请在回答问题前先使用搜索工具获取最新信息。""""

```
agent = SimpleAgent(name="AI助手", llm=llm, system_prompt=system_prompt)
```

```
# 添加搜索工具
agent.add_tool(SearchTool())
```

```
# 示例：使用搜索工具回答问题
response = agent.run("最新的AI技术发展趋势是什么？")
print(f"\n回答: {response}")
```

这个智能体能正常工作，但我们面临一个核心问题：如何客观地评估它的性能？当我们优化提示词或更换 LLM 模型后，如何知道是否真的有改进？在部署到生产环境前，如何保证智能体的可靠性？这些问题都需要通过系统化的评估来解决。

智能体评估的核心价值在于提供标准化的方法来衡量智能体的能力。通过评估，我们可以用具体的数字指标量化智能体的表现，客观比较不同设计方案的优劣，及时发现智能体在特定场景下的弱点，并向用户证明智能体的可靠性。

与传统软件测试不同，智能体评估面临着独特的挑战。首先是输出的不确定性，同一问题可能有多个正确答案，很难用简单的对错来判断。其次是评估标准的多样性，不同任务需要不同的评估方法，工具调用需要检查函数签名，问答任务需要评估语义相似度。最后是评估成本的高昂，每次评估都需要大量的 API 调用，成本可能达到数百元甚至更多。

为了应对这些挑战，学术界和工业界提出了多个标准化的评估基准（Benchmark）。这些基准提供了统一的数据集、评估指标和评分方法，使我们能够在相同的标准下评估和对比不同的智能体系统。

12.1.2 主流评估基准概览

智能体评估领域已经涌现出多个具有影响力的基准测试。下面介绍一些主流的评估基准和指标：

(1) 工具调用能力评估

工具调用是智能体的核心能力之一。智能体需要理解用户意图，选择合适的工具，并正确构造函数调用。相关的评估基准包括：

- **BFCL (Berkeley Function Calling Leaderboard)**^[1]: UC Berkeley 推出，包含 1120+ 测试样本，涵盖 simple、multiple、parallel、irrelevance 四个类别，使用 AST 匹配算法评估，数据集规模适中，社区活跃。
- **ToolBench**^[2]: 清华大学推出，包含 16000+ 真实 API 调用场景，覆盖真实世界的复杂工具使用场景。
- **API-Bank**^[3]: Microsoft Research 推出，包含 53 个常用 API 工具，专注于评估智能体对 API 文档的理解和调用能力。

(2) 通用能力评估

评估智能体在真实世界任务中的综合表现，包括多步推理、知识运用、多模态理解等能力：

- **GAIA (General AI Assistants)**^[4]: Meta AI 和 Hugging Face 联合推出, 包含 466 个真实世界问题, 分为 Level 1/2/3 三个难度级别, 评估多步推理、工具使用、文件处理、网页浏览等能力, 使用准精确匹配 (Quasi Exact Match) 算法, 任务真实且综合性强。
- **AgentBench**^[5]: 清华大学推出, 包含 8 个不同领域的任务, 全面评估智能体的通用能力。
- **WebArena**^[6]: CMU 推出, 评估智能体在真实网页环境中的任务完成能力和网页交互能力。

(3) 多智能体协作评估

评估多个智能体协同工作的能力:

- **ChatEval**^[7]: 评估多智能体对话系统的质量。
- **SOTONIA**^[8]: 评估智能体在社交场景中的互动能力。
- 自定义协作场景: 根据具体应用场景设计的评估任务。

(4) 常用评估指标

不同基准使用不同的评估指标, 常见的包括:

- 准确性指标: Accuracy (准确率)、Exact Match (精确匹配)、F1 Score (F1 分数), 用于衡量答案的正确性。
- 效率指标: Response Time (响应时间)、Token Usage (Token 使用量), 用于衡量执行效率。
- 鲁棒性指标: Error Rate (错误率)、Failure Recovery (故障恢复), 用于衡量容错能力。
- 协作指标: Communication Efficiency (通信效率)、Task Completion (任务完成度), 用于衡量协作效果。

12.1.3 HelloAgents 评估体系设计

考虑到学习曲线和实用性, 本章将重点介绍以下评估场景:

1. **BFCL**: 评估工具调用能力
 - 选择理由: 数据集规模适中, 评估指标清晰, 社区活跃
 - 适用场景: 评估智能体的函数调用准确性
2. **GAIA**: 评估通用 AI 助手能力
 - 选择理由: 任务真实, 难度分级, 综合性强
 - 适用场景: 评估智能体的综合问题解决能力
3. 数据生成质量评估: 评估 LLM 生成数据质量
 - 选择理由: 通过这个案例可以完整体验如何使用 Agent 创造数据, 评估数据的完整演示。
 - 适用场景: 评估生成的训练数据、测试数据的质量
 - 评估方法: LLM Judge、Win Rate、人工验证

通过这三个评估场景, 我们将构建一个完整的评估体系, 如图 12.1 展示了我们的评估系统构建思路。



图 12.1 HelloAgents 评估体系架构图

12.1.4 本章学习目标与快速体验

让我们先看看第十二章的学习内容:

```
hello_agents/
└── evaluation/
    └── benchmarks/
        ├── bfcl/
        │   ├── dataset.py
        │   ├── evaluator.py
        │   ├── metrics.py
        │   └── ast_matcher.py
        └── gaia/
            ├── dataset.py
            ├── evaluator.py
            ├── metrics.py
            └── quasi_exact_match.py
    └── data_generation/
        ├── dataset.py
        ├── llm_judge.py
        └── win_rate.py
    └── tools/builtin/
        ├── bfcl_evaluation_tool.py
        ├── gaia_evaluation_tool.py
        ├── llm_judge_tool.py
        └── win_rate_tool.py
    └── # 评估模块
    └── # 评估基准实现
    └── # BFCL 评估实现
    └── # BFCL 数据集加载器
    └── # BFCL 评估器 (AST 匹配)
    └── # BFCL 专用指标
    └── # AST 匹配算法
    └── # GAIA 评估实现
    └── # GAIA 数据集加载器
    └── # GAIA 评估器 (准精确匹配)
    └── # GAIA 专用指标
    └── # 准精确匹配算法
    └── # 数据生成评估实现
    └── # AIME 数据集加载器
    └── # LLM Judge 评估器
    └── # Win Rate 评估器
    └── # 内置工具模块
    └── # BFCL 评估工具
    └── # GAIA 评估工具
    └── # LLM Judge 工具
    └── # Win Rate 工具
```

对于这一章的内容, 学习目标是掌握应用评估工具的能力。让我们先准备好开发环境:

```
# 安装HelloAgents框架 (第12章版本)
pip install "hello-agents[evaluation]==0.2.7"
```

```
# 设置环境变量
export HF_TOKEN="your_huggingface_token"      # 用于GAIA数据集(后续也会有设置步骤)

# 由于 `bfcl-eval` 官方包强制要求 numpy<=2.0.0, 和HelloAgents 主依赖版本存在冲突,因此需要单独安装
pip install "numpy==1.26.4" bfcl-eval
```

在接下来的章节中，我们将深入学习每种评估方法的详细用法和介绍。

12.2 BFCL：工具调用能力评估

12.2.1 BFCL 基准介绍

BFCL (Berkeley Function Calling Leaderboard) 是由加州大学伯克利分校推出的函数调用能力评估基准^[1]。在智能体系统中，工具调用（Tool Calling）是核心能力之一。智能体需要完成以下任务：

1. 理解任务需求：从用户的自然语言描述中提取关键信息
2. 选择合适工具：从可用工具集中选择最适合的工具
3. 构造函数调用：正确填写函数名和参数
4. 处理复杂场景：支持多函数调用、并行调用等高级场景

BFCL 基准包含四个评估类别，难度递增。从最基础的单函数调用（Simple）开始，逐步增加到需要调用多个函数的场景（Multiple），再到需要并行调用多个函数的复杂场景（Parallel），最后是需要判断是否需要调用函数的场景（Irrelevance）。这四个类别覆盖了智能体在实际应用中可能遇到的各种工具调用场景，如表 12.1 所示：

表 12.1 BFCL 基准中的四个评估类别



BFCL 的评估流程遵循标准的基准测试流程：首先加载数据集并选择评估类别，然后运行智能体获取预测结果，接着将预测结果解析为抽象语法树（AST），最后通过 AST 匹配算法判断预测是否正确。整个流程会遍历所有测试样本，最终计算出准确率等评估指标并生成评估报告。完整的评估流程如图 12.2 所示：



图 12.2 BFCL 评估流程图

(1) BFCL 数据集结构

BFCL 数据集采用 JSON 格式，每个测试样本包含以下字段：

```
{
  "id": "simple_001",
  "question": "What's the weather like in Beijing today?",
  "function": [
    {
      "name": "get_weather",
      "description": "Get the current weather for a location",
      "parameters": {
        "type": "object",
        "properties": {
          "location": {
            "type": "string",
            "description": "The city name"
          }
        },
        "required": ["location"]
      }
    }
  ],
  "ground_truth": [
    {
      "name": "get_weather",
      "arguments": {
        "location": "Beijing"
      }
    }
  ]
}
```

关键字段说明：

- **question**: 用户的自然语言请求
- **function**: 可用的函数列表（包含函数签名和描述）
- **ground_truth**: 标准答案（期望的函数调用）

(2) AST 匹配说明

BFCL 使用 AST 匹配（Abstract Syntax Tree Matching）作为核心评估算法，因此下文可以了解一下评估的策略。

BFCL 使用抽象语法树（AST）进行智能匹配，而不是简单的字符串匹配。AST 匹配的核心思想是：将函数调用解析为语法树，然后比较树的结构和节点值。

给定预测的函数调用 P 和标准答案 G , AST 匹配函数定义为:

```
 $$ \text{AST\_Match}(P, G) = \begin{cases} 1 & \text{if } \text{AST}(P) \equiv \text{AST}(G) \\ 0 & \text{otherwise} \end{cases} $$
```

其中 $\text{AST}(x)$ 表示将函数调用解析为抽象语法树, \equiv 表示语法树等价。

两个语法树等价需要满足三个核心条件: 函数名必须完全一致 (精确匹配), 参数键值对集合相等 (忽略顺序), 以及每个参数的值在语义上等价 (例如 $2+3$ 等价于 5)。在具体的匹配过程中, 函数名匹配要求字符串精确匹配, 例如 `get_weather` 和 `get_temperature` 被视为不同的函数。参数匹配则使用 AST 进行智能比较, 允许参数顺序不同 ($f(a=1, b=2)$ 等价于 $f(b=2, a=1)$), 允许等价表达式 ($f(x=2+3)$ 等价于 $f(x=5)$), 也允许不同的字符串表示 ($f(s="hello")$ 等价于 $f(s='hello')$)。对于多函数调用的场景, 匹配算法要求调用相同数量的函数, 每个函数调用都必须匹配, 但调用顺序可以不同 (使用集合匹配)。

AST 匹配示例:

示例1: 参数顺序不同 (匹配成功)

预测: `get_weather(city="Beijing", unit="celsius")`
标准: `get_weather(unit="celsius", city="Beijing")`

结果: 匹配成功

示例2: 等价表达式 (匹配成功)

预测: `calculate(x=2+3)`
标准: `calculate(x=5)`

结果: 匹配成功

示例3: 函数名错误 (匹配失败)

预测: `get_temperature(city="Beijing")`
标准: `get_weather(city="Beijing")`

结果: 匹配失败

示例4: 参数值错误 (匹配失败)

预测: `get_weather(city="Shanghai")`
标准: `get_weather(city="Beijing")`

结果: 匹配失败

(3) BFCL 评估指标

BFCL 使用以下指标评估智能体性能:

1. 准确率 (Accuracy)

准确率是最核心的指标, 定义为 AST 匹配成功的样本比例:

```
 $$ \text{Accuracy} = \frac{1}{N} \sum_{i=1}^N \text{AST\_Match}(P_i, G_i) $$
```

其中: - N 是总样本数 - P_i 是第 i 个样本的预测结果 - G_i 是第 i 个样本的标准答案 - $\text{AST_Match}(P_i, G_i) \in \{0, 1\}$ 是 AST 匹配函数

2. AST 匹配率 (AST Match Rate)

与准确率相同, 强调使用 AST 匹配算法:

$\text{AST Match Rate} = \text{Accuracy}$

3. 分类准确率 (Category-wise Accuracy)

对于每个类别 $c \in \{\text{simple}, \text{multiple}, \text{parallel}, \dots\}$, 计算该类别的准确率:

```
 $$ \text{Accuracy}_c = \frac{1}{|D_c|} \sum_{i \in D_c} \text{AST\_Match}(P_i, G_i) $$
```

其中 D_c 是类别 c 的样本集合, $|D_c|$ 是该类别的样本数。

4. 加权准确率 (Weighted Accuracy)

考虑不同类别的难度权重:

$\text{Weighted Accuracy} = \sum_c w_c \cdot \text{Accuracy}_c$

其中 w_c 是类别 c 的权重, 满足 $\sum_c w_c = 1$ 。

5. 错误率 (Error Rate)

未能正确调用函数的样本比例:

```
 $$ \text{Error Rate} = 1 - \text{Accuracy} = \frac{1}{N} \sum_{i=1}^N (1 - \text{AST\_Match}(P_i, G_i)) $$
```

指标解释:

- **Accuracy = 1.0:** 所有样本都完全正确
- **Accuracy = 0.8:** 80%的样本正确, 20%的样本错误

- **Accuracy = 0.0:** 所有样本都错误

分类准确率示例：

```
# 假设评估结果
simple_accuracy = 0.95      # Simple类别: 95%正确
multiple_accuracy = 0.82    # Multiple类别: 82%正确
parallel_accuracy = 0.68     # Parallel类别: 68%正确

# 加权准确率 (假设权重相等)
weighted_accuracy = (0.95 + 0.82 + 0.68) / 3 = 0.817
```

(4) BFCL 官方评估工具

BFCL 提供官方 CLI 工具进行评估：

```
# 安装BFCL评估工具
pip install bfcl

# 运行官方评估
bfcl evaluate \
--model-result-path ./results.json \
--test-category simple_python
```

使用官方评估工具的优势在于：它使用官方的 AST 匹配算法，评估结果与排行榜完全一致，支持所有 BFCL v4 类别，并且能够自动生成详细的评估报告。

12.2.2 获取 BFCL 数据集

BFCL 数据集可以通过以下方式获取：

方法 1：从官方 GitHub 仓库克隆（推荐）

这是最可靠的方式，可以获取完整的数据集和 ground truth：

```
# 克隆BFCL仓库
git clone https://github.com/ShishirPatil/gorilla.git temp_gorilla
cd temp_gorilla/berkeley-function-call-leaderboard

# 查看BFCL v4数据集
ls bfcl_eval/data/
# 输出: BFCL_v4_simple_python.json  BFCL_v4_multiple.json  BFCL_v4_parallel.json  ...

# 查看ground truth
ls bfcl_eval/data/possible_answer/
# 输出: BFCL_v4_simple_python.json  BFCL_v4_multiple.json  ...
```

推荐这种方式的原因是：它包含完整的 ground truth（标准答案），数据格式与官方评估工具完全一致，可以直接使用官方评估脚本，并且支持 BFCL v4 最新版本。

方法 2：使用 HelloAgents 加载官方数据

克隆仓库后，使用 HelloAgents 加载数据：

```
from hello_agents.evaluation import BFCLDataset

# 加载BFCL官方数据
dataset = BFCLDataset(
    bfcl_data_dir="./temp_gorilla/berkeley-function-call-leaderboard/bfcl_eval/data",
    category="simple_python" # BFCL v4类别
)

# 加载数据（包括测试数据和ground truth）
data = dataset.load()

print(f"✅ 加载了 {len(data)} 个测试样本")
print(f"✅ 加载了 {len(dataset.ground_truth)} 个ground truth")
# 输出:
# ✅ 加载了 400 个测试样本
# ✅ 加载了 400 个ground truth
```

这个加载器的工作原理是：首先从 bfcl_eval/data/ 加载测试数据，然后从 bfcl_eval/data/possible_answer/ 加载 ground truth，接着自动合并测试数据和 ground truth，最后保留原始 BFCL 数据格式。其中 BFCL v4 数据集类别可以在表 12.2 查看。

表 12.2 BFCL 基准中的四个评估类别



当然也可以通过代码查看可用类别：

```
# 获取所有支持的类别
categories = dataset.get_available_categories()
```

```
print(f"支持的类别: {categories}")
# 输出: ['simple_python', 'simple_java', 'simple_javascript', 'multiple', ...]
```

12.2.3 在 HelloAgents 中实现 BFCL 评估

现在让我们看看如何在 HelloAgents 框架中实现 BFCL 评估。我们提供了三种使用方式：

方式 1：使用 BFCEvaluationTool（推荐）

这是最简单的方式，一行代码完成评估、报告生成和官方评估：

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import BFCEvaluationTool

# 1. 创建要评估的智能体
llm = HelloAgentsLLM()
agent = SimpleAgent(name="TestAgent", llm=llm)

# 2. 创建BFCL评估工具
bfcl_tool = BFCEvaluationTool()

# 3. 运行评估（自动完成所有步骤）
results = bfcl_tool.run(
    agent=agent,
    category="simple_python", # 评估类别
    max_samples=5 # 评估样本数（0表示全部）
)

# 4. 查看结果
print(f"准确率: {results['overall_accuracy']:.2%}")
print(f"正确数: {results['correct_samples']}/{results['total_samples']}")
```

运行输出：

```
=====
BFCL一键评估
=====

配置：
评估类别: simple_python
样本数量: 5
智能体: TestAgent

=====
步骤1: 运行HelloAgents评估
=====

✓ BFCL数据集加载完成
数据目录: ./temp_gorilla/berkeley-function-call-leaderboard/bfcl_eval/data
类别: simple_python
样本数: 400
Ground truth数: 400

\ 开始 BFCL 评估...
进度: 1/5
进度: 5/5

✓ BFCL 评估完成
总体准确率: 100.00%
simple_python: 100.00% (5/5)

\ 评估结果:
准确率: 100.00%
正确数: 5/5

=====
步骤2: 导出BFCL格式结果
=====

✓ BFCL格式结果已导出
输出文件: ./evaluation_results/bfcl_official/BFCL_v4_simple_python_result.json

=====
步骤3: 运行BFCL官方评估
=====

✓ 结果文件已复制到: ./result/Qwen_Qwen3-8B/BFCL_v4_simple_python_result.json

\ 运行命令: bfcl evaluate --model Qwen/Qwen3-8B --test-category simple_python --partial-eval

=====
BFCL官方评估结果
=====

\ 评估结果汇总:
Model,Overall Acc,simple_python
Qwen/Qwen3-8B,100.00,100.00

\ 最终结果:
```

2/9/26, 1:56 PM

准确率: 100.00%
正确数: 5/5

```
=====
步骤4: 生成评估报告
=====
报告已生成: ./evaluation_reports/bfcl_report_20251011_005938.md
```

准确率: 100.00%
正确数: 5/5

自动生成的 Markdown 报告:

评估完成后，会自动生成一份详细的 Markdown 报告，包含：

```
# BFCL评估报告
**生成时间**: 2025-10-11 00:59:38

## 📈 评估概览
- **智能体**: TestAgent
- **评估类别**: simple_python
- **总体准确率**: 100.00%
- **正确样本数**: 5/5

## 📈 详细指标
### 分类准确率
- **simple_python**: 100.00% (5/5)

## 📈 样本详情
样本ID	问题	预测结果	正确答案	是否正确
simple_python_0	Find the area of a triangle...	[{"name': 'calculate_triangle_area'}]	[{"function_name': {'base': [10]}}]	✓
simple_python_1	Calculate the factorial of 5...	[{"name': 'calculate_factorial'}]	[{"function_name': {'number': [5]}}]	✓
...
## 💡 准确率可视化
准确率: 

100.00%


```

💡 建议
- 表现优秀！智能体在工具调用方面表现出色。

方式 2：使用一键评估脚本

适合命令行快速评估，在这一章配套的代码案例里，我们提供了 `04_run_bfcl_evaluation.py`，支持直接命令行调用测评：

```
# 运行评估脚本
python chapter12/04_run_bfcl_evaluation.py --category simple_python --samples 10

# 指定模型名称 (用于BFCL官方评估)
python examples/04_run_bfcl_evaluation.py \
--category simple_python \
--samples 10 \
--model-name "Qwen/Qwen3-8B"
```

脚本支持三个参数：`--category`指定评估类别（默认 `simple_python`），`--samples`指定评估样本数（默认 5，0 表示全部），`--model-name`指定模型名称用于 BFCL 官方评估（默认 `Qwen/Qwen3-8B`）。

方式 3：直接使用 Dataset 和 Evaluator

适合需要自定义评估流程的场景：

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.evaluation import BFCLDataset, BFCLEvaluator

# 1. 创建智能体
llm = HelloAgentsLLM()
agent = SimpleAgent(name="TestAgent", llm=llm)

# 2. 加载数据集
dataset = BFCLDataset(
    bfcl_data_dir="./temp_gorilla/berkeley-function-call-leaderboard/bfcl_eval/data",
    category="simple_python"
)
data = dataset.load()

# 3. 创建评估器
evaluator = BFCLEvaluator(
    dataset=dataset,
    category="simple_python",
    evaluation_mode="ast" # 使用AST匹配模式
)
```

```
# 4. 运行评估
results = evaluator.evaluate(agent, max_samples=10)

# 5. 查看结果
print(f"准确率: {results['overall_accuracy']:.2%}")
print(f"正确数: {results['correct_samples']}/{results['total_samples']}")

# 6. 导出BFCL格式结果 (可选)
evaluator.export_to_bfcl_format(
    results,
    output_path=".//evaluation_results/my_results.json"
)
```

通过以上三种方式，我们可以根据不同的需求选择合适的评估方法。如果只是想快速了解智能体的表现，使用 BFCEvaluationTool 的一键评估最为便捷；如果需要批量评估或集成到 CI/CD 流程，使用命令行脚本更加合适；如果需要深度定制评估流程或集成到自己的系统中，直接使用 Dataset 和 Evaluator 提供了最大的灵活性。

12.2.4 BFCL 官方评估工具集成

前面我们学习了如何使用 HelloAgents 内置的评估功能。实际上，BFCEvaluationTool 已经自动集成了 BFCL 官方评估工具，让你能够获得权威的、可对比的评估结果。

整个评估流程包括四个步骤：首先从 BFCL v4 数据集加载测试数据，然后使用 HelloAgents 运行评估获取智能体的预测结果，接着将结果导出为 BFCL 官方格式（JSONL），最后使用官方评估脚本计算最终分数。这个流程确保了评估结果与 BFCL 排行榜完全一致，如图 12.3 所示：



图 12.3 Helloagents 载入 BFCL 评估过程

使用 BFCEvaluationTool 时，官方评估会自动运行（默认启用）：

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import BFCEvaluationTool

# 创建智能体
llm = HelloAgentsLLM()
agent = SimpleAgent(name="TestAgent", llm=llm)

# 创建评估工具
bfcl_tool = BFCEvaluationTool()

# 运行评估 (自动运行官方评估)
results = bfcl_tool.run(
    agent=agent,
    category="simple_python",
    max_samples=5,
    # run_official_eval=True # 默认为True, 可以省略
    model_name="Qwen/Qwen3-8B" # 可选, 指定模型名称
)
```

工具会自动执行完整的评估流程：首先运行 HelloAgents 评估获取预测结果，然后将结果导出为 BFCL 格式并保存到 evaluation_results/bfcl_official/ 目录，接着复制结果文件到 result/{model_name}/ 目录以符合官方评估工具的要求，随后运行 BFCL 官方评估命令计算分数，最后显示官方评估结果并生成 Markdown 格式的评估报告。

官方评估输出示例：

```
=====
步骤3: 运行BFCL官方评估
=====

✓ 结果文件已复制到:
./result/Qwen_Qwen3-8B/BFCL_v4_simple_python_result.json

⌚ 运行命令: bfcl evaluate --model Qwen/Qwen3-8B --test-category simple_python --partial-eval
=====

BFCL官方评估结果
=====

📊 评估结果汇总:
Model,Overall Acc,simple_python
Qwen/Qwen3-8B,100.00,100.00

⌚* 最终结果:
准确率: 100.00%
正确数: 5/5
```

如果你想手动控制评估流程，可以禁用自动官方评估：

```
# 禁用官方评估
results = bfcl_tool.run(
    agent=agent,
    category="simple_python",
    max_samples=5,
```

```

        run_official_eval=False # 禁用官方评估
    )

# 然后手动运行官方评估
import subprocess
subprocess.run([
    "bfcl", "evaluate",
    "--model", "Owen/Owen3-8B",
    "--test-category", "simple_python",
    "--partial-eval"
])

```

你也可以手动生成报告：

```

# 运行评估
results = bfcl_tool.run(agent, category="simple_python", max_samples=5)

# 手动生成报告
report = bfcl_tool.generate_report(
    results,
    output_file="./my_reports/custom_report.md"
)

# 打印报告内容
print(report)

```

12.2.5 核心组件实现细节

在前面的小节中，我们学习了如何使用 BFCL 评估工具。现在让我们深入了解 HelloAgents 评估系统的核心组件是如何实现的。理解这些实现细节不仅能帮助你更好地使用评估系统，还能让你根据自己的需求进行定制和扩展。

(1) BFCLDataset: 数据集加载器

BFCLDataset 负责加载和管理 BFCL 数据集：

```

class BFCLDataset:
    """BFCL数据集加载器"""

    def __init__(self, category: str = "simple", local_data_path: Optional[str] = None):
        self.category = category
        self.local_data_path = local_data_path
        self.data = []

    def load(self) -> List[Dict[str, Any]]:
        """加载数据集"""
        # 优先从本地加载
        if self.local_data_path:
            return self._load_from_local()
        # 否则从Hugging Face加载
        return self._load_from_huggingface()

```

因为 BFCL 的数据集就在官方的仓库内，所以这里建议的方式是直接在本地 clone 一份进行测评。当找不到时才到 huggingface 进行加载。

(2) BFCLEvaluator: 评估执行器

BFCLEvaluator 负责执行评估流程。它的核心是 evaluate() 方法，该方法协调整个评估过程：

```

class BFCLEvaluator:
    """BFCL评估器"""

    def evaluate(self, agent: Any, max_samples: Optional[int] = None) -> Dict[str, Any]:
        """执行评估"""
        results = []

        for item in self.dataset[:max_samples]:
            # 1. 构造提示词
            prompt = self._build_prompt(item)

            # 2. 调用智能体
            response = agent.run(prompt)

            # 3. 提取函数调用
            predicted_calls = self._extract_function_calls(response)

            # 4. 与标准答案对比
            is_correct = self._compare_calls(predicted_calls, item["ground_truth"])

            results.append({
                "id": item["id"],
                "prediction": predicted_calls,
                "ground_truth": item["ground_truth"],
                "is_correct": is_correct
            })

        return {"results": results, "total_samples": len(results)}

```

这个评估器的设计包含三个核心要点：首先是提示词构造，需要将数据集中的问题和函数定义转换为智能体可理解的提示词；其次是函数调用提取，需要从智能体的响应中提取函数调用，并支持多种格式（JSON、代码块等）；最后是 AST 匹配，使用抽象语法树进行函数调用对比，这比简单的字符串匹配更准

确。

让我们看看函数调用提取的实现:

```
def _extract_function_calls(self, response: str) -> List[Dict[str, Any]]:
    """从响应中提取函数调用

    支持多种格式:
    1. JSON格式: {"name": "func", "arguments": {...}}
    2. 代码块格式: ```python\nfunc(arg1=val1)\n```
    3. 纯文本格式: func(arg1=val1)
    """
    calls = []

    # 尝试JSON解析
    try:
        json_match = re.search(r'\{.*\}', response, re.DOTALL)
        if json_match:
            data = json.loads(json_match.group())
            if isinstance(data, dict) and "name" in data:
                calls.append(data)
            elif isinstance(data, list):
                calls.extend(data)
    except json.JSONDecodeError:
        pass

    # 尝试代码块提取
    code_blocks = re.findall(r'```(?:python)?\n(.*)\n```', response, re.DOTALL)
    for code in code_blocks:
        # 解析Python函数调用
        parsed_calls = self._parse_python_calls(code)
        calls.extend(parsed_calls)

    return calls
```

(3) BFCLMetrics: 指标计算器

BFCLMetrics 负责计算各种评估指标:

```
class BFCLMetrics:
    """BFCL指标计算器"""

    def compute_metrics(self, results: List[Dict[str, Any]]) -> Dict[str, Any]:
        """计算所有指标"""
        return {
            "accuracy": self._compute_accuracy(results),
            "ast_match_rate": self._compute_ast_match_rate(results),
            "parameter_accuracy": self._compute_parameter_accuracy(results),
            "f1_score": self._compute_f1_score(results),
            "category_statistics": self._compute_category_stats(results)
        }
```

AST 匹配的实现:

AST 匹配是 BFCL 评估的核心技术。它比简单的字符串匹配更智能，能够识别语义等价的函数调用:

```
def _ast_match(self, pred_call: Dict, true_call: Dict) -> bool:
    """使用AST匹配函数调用

    AST匹配的优势:
    1. 忽略参数顺序: func(a=1, b=2) 等价于 func(b=2, a=1)
    2. 识别等价表达式: 2+3 等价于 5
    3. 忽略空格和格式差异
    """
    # 1. 函数名必须完全匹配
    if pred_call.get("name") != true_call.get("name"):
        return False

    # 2. 将参数转换为AST节点
    pred_args = self._args_to_ast(pred_call.get("arguments", {}))
    true_args = self._args_to_ast(true_call.get("arguments", {}))

    # 3. 比较AST节点
    return ast.dump(pred_args) == ast.dump(true_args)

def _args_to_ast(self, args: Dict[str, Any]) -> ast.AST:
    """将参数字典转换为AST节点"""
    # 构造一个虚拟的函数调用
    code = f"func({', '.join(f'{k}={repr(v)}' for k, v in args.items())})"
    tree = ast.parse(code)
    return tree.body[0].value # 返回Call节点
```

(4) 工具化封装: BFCLEvaluationTool

最后，我们将这些组件封装成一个 Tool，让它可以被智能体直接调用:

```
class BFCLEvaluationTool(Tool):
    """BFCL评估工具"""

    file:///Users/knight/workspace/hello-agents/docs/Hello-Agents-Full.html
```

```

def __init__(self, local_data_path: Optional[str] = None):
    super().__init__(
        name="bfcl_evaluation",
        description="评估智能体的工具调用能力"
    )
    self.dataset = None
    self.evaluator = None
    self.metrics_calculator = BFCLMetrics()

def run(self, parameters: Dict[str, Any]) -> str:
    """执行评估"""
    # 1. 加载数据集
    self.dataset = BFCLDataset(...)

    # 2. 创建评估器
    self.evaluator = BFCLEvaluator(...)

    # 3. 运行评估
    results = self.evaluator.evaluate(...)

    # 4. 计算指标
    metrics = self.metrics_calculator.compute_metrics(...)

    # 5. 返回JSON结果
    return json.dumps(results, ensure_ascii=False)

```

这个工具的设计遵循三个核心原则：首先继承 Tool 基类以遵循 HelloAgents 的工具规范，确保与框架的无缝集成；其次进行严格的参数验证，检查必需参数并提供友好的错误提示，提升用户体验；最后对结果进行格式化，返回 JSON 字符串以便于解析和展示。通过这种模块化的设计，我们实现了一个既易用又灵活的评估系统，用户可以直接使用高层的 Tool 接口快速完成评估，也可以深入到底层组件进行定制以满足特殊需求。

12.2.6 扩展与优化建议

通过前面的学习，我们已经掌握了如何使用 HelloAgents 进行 BFCL 评估。需要注意的是，我们目前的实现是基于 SimpleAgent 的简单复现，主要完成了 BFCL 评估的基础功能。在实际应用中，BFCL 基准包含多个难度级别和场景，要在排行榜上获得更高的分数，还需要进一步的优化和扩展。

(1) 当前实现的局限性

我们当前的 SimpleAgent 实现主要聚焦于评估流程的搭建，在工具调用能力上还有提升空间。SimpleAgent 使用自定义的工具调用格式 [TOOL_CALL:tool_name:parameters]，这种格式需要 LLM 主动学习和使用，在复杂场景下的表现可能不如使用原生函数调用（Function Calling）的智能体。此外，我们目前只测试了 simple_python 等基础类别，对于 multiple、parallel、irrelevance 等更复杂的场景，还需要针对性的优化。

(2) 提升 BFCL 分数的方向

要进一步提升 BFCL 评估分数，可以从以下几个方向入手。首先是优化智能体的工具调用能力，可以考虑使用支持原生函数调用的 LLM（如 GPT-4、Claude 等），或者改进提示词让 LLM 更好地理解工具调用格式。其次是扩展工具库，BFCL 测试中涉及各种类型的函数，可以根据测试数据集的特点，预先实现常用的工具类型，提高智能体的工具覆盖率。第三是针对不同难度级别设计不同的策略，例如在 multiple 场景下需要智能体能够规划多步骤的工具调用序列，在 parallel 场景下需要识别可以并行执行的工具调用，在 irrelevance 场景下需要判断是否真的需要调用工具。

(3) 实践建议

对于想要在 BFCL 上取得更好成绩的开发者，建议采用以下实践策略。首先，从 simple 类别开始，确保基础的单函数调用能够稳定工作，这是后续优化的基础。然后，逐步测试 multiple、parallel 等更复杂的类别，分析失败案例，找出智能体的薄弱环节。在优化过程中，可以参考 BFCL 排行榜上的高分模型，学习它们的设计思路和优化技巧。同时，建议使用官方评估工具进行验证，确保优化后的结果与排行榜标准一致。

这里总结一些评估时可以进一步处理的建议：

1. 渐进式评估

从小样本开始，逐步增加样本数：

```

# 第一步：快速测试（5个样本）
results_quick = bfcl_tool.run(agent, category="simple_python", max_samples=5)

# 第二步：中等规模测试（50个样本）
if results_quick['overall_accuracy'] > 0.8:
    results_medium = bfcl_tool.run(agent, category="simple_python", max_samples=50)

# 第三步：完整评估（全部样本）
if results_medium['overall_accuracy'] > 0.8:
    results_full = bfcl_tool.run(agent, category="simple_python", max_samples=0)

```

2. 多类别评估

评估不同难度的任务：

```

categories = ["simple_python", "multiple", "parallel", "irrelevance"]

for category in categories:
    print(f"\n评估类别: {category}")
    results = bfcl_tool.run(agent, category=category, max_samples=10)
    print(f"准确率: {results['overall_accuracy']:.2%}")

```

3. 对比评估

对比不同配置的智能体：

```
# 配置1：默认提示词
agent1 = SimpleAgent(name="Agent-Default", llm=llm)
results1 = bfcl_tool.run(agent1, category="simple_python", max_samples=10)

# 配置2：优化提示词
agent2 = SimpleAgent(name="Agent-Optimized", llm=llm)
# ... 设置优化的系统提示词 ...
results2 = bfcl_tool.run(agent2, category="simple_python", max_samples=10)

# 对比结果
print(f"默认配置准确率: {results1['overall_accuracy']:.2%}")
print(f"优化配置准确率: {results2['overall_accuracy']:.2%}")

如果你的评估结果很好，可以考虑提交到 BFCL 官方排行榜！
```

步骤 1：准备提交材料

1. 模型描述文档
2. 评估结果文件（所有类别）
3. 模型访问方式（API 或开源链接）

步骤 2：提交到 GitHub

访问 BFCL 官方仓库，按照说明提交 Pull Request：

- 仓库地址：<https://github.com/ShishirPatil/gorilla>
- 提交指南：参考 [CONTRIBUTING.md](#)

步骤 3：等待审核

BFCL 团队会审核你的提交，验证结果的准确性。审核通过后，你的模型将出现在官方排行榜上！

12.3 GAIA：通用 AI 助手能力评估

12.3.1 GAIA 基准介绍

GAIA (General AI Assistants) 是由 Meta AI 和 Hugging Face 联合推出的评估基准，专注于评估 AI 助手的通用能力^[2]。与 BFCL 专注于工具调用不同，GAIA 评估的是智能体在真实世界任务中的综合表现。

GAIA 的设计理念是：真实世界的问题往往需要多种能力的综合运用。一个优秀的 AI 助手不仅需要调用工具，还需要：

- 多步推理：将复杂问题分解为多个子问题
- 知识运用：利用内置知识和外部知识库
- 多模态理解：处理文本、图片、文件等多种输入
- 网页浏览：从互联网获取最新信息
- 文件操作：读取和处理各种格式的文件

(1) GAIA 数据集结构

了解 GAIA 的评估理念后，让我们深入了解 GAIA 数据集的具体结构。GAIA 包含 466 个精心设计的真实世界问题，这些问题按照复杂度和所需推理步骤分为三个难度级别，从简单的零步推理任务到需要多步复杂推理的困难任务，全面覆盖了智能体在实际应用中可能遇到的各种场景，如表 12.3 所示：

表 12.3 GAIA 数据集难度级别分布

| 难度级别 | 任务数量 |
|------|------|
| 零步 | 466 |
| 一步 | 100 |
| 多步 | 100 |

关于 GAIA 数据集的样本示例可以参考下面的代码片段：

```
{
  "task_id": "gaia_001",
  "Question": "What is the total population of the top 3 most populous cities in California?",
  "Level": 2,
  "Final answer": "12847521",
  "file_name": "",
  "file_path": "",
  "Annotator Metadata": {
    "Steps": [
      "Search for most populous cities in California",
      "Get population data for top 3 cities",
      "Sum the populations"
    ],
    "Number of steps": 3,
    "How long did this take?": "5 minutes",
    "Tools": ["web_search", "calculator"]
  }
}
```

关键字段说明: - Question: 问题描述 - Level: 难度级别 (1-3) - Final answer: 标准答案 (可能是数字、文本或文件) - file_name/file_path: 附件文件 (如果有) - Annotator Metadata: 标注者提供的元数据 (推理步骤、所需工具等)

(2) 准精确匹配介绍

GAIA 使用准精确匹配 (**Quasi Exact Match**) 评估算法, 这是 GAIA 官方定义的评估标准。该算法的核心思想是: 先对答案进行归一化处理, 然后进行精确匹配。

给定预测答案 A_{pred} 和标准答案 A_{true} , 准精确匹配函数定义为:

```
$$ \text{Quasi\_Exact\_Match}(A_{\text{pred}}, A_{\text{true}}) = \begin{cases} 1 & \text{if } \mathcal{N}(A_{\text{pred}}) = \mathcal{N}(A_{\text{true}}) \\ 0 & \text{otherwise} \end{cases}
```

其中 $\mathcal{N}(\cdot)$ 是归一化函数, 根据答案类型应用不同的规则。

归一化函数根据答案类型应用不同的规则。对于数字类型, 需要移除逗号分隔符 ($1,000 \rightarrow 1000$) 和单位符号 ($\$100 \rightarrow 100$, $50\% \rightarrow 50$) , 例如 " $\$1,234.56$ " 归一化为 " 1234.56 "。对于字符串类型, 需要转换为小写 (" Apple " → " apple ") 、移除冠词 (" the apple " → " apple ") 、移除多余空格 (" hello world " → " hello world ") 和移除末尾标点 (" hello. " → " hello ") , 例如 " The United States " 归一化为 " united states "。对于列表类型, 需要按逗号分隔元素, 对每个元素应用字符串归一化, 按字母顺序排序后重新连接, 例如 " $\text{Paris, London, Berlin}$ " 归一化为 " $\text{berlin, london, paris}$ "。

归一化示例:

```
# 数字答案
原始答案: "$1,234.56"
归一化后: "1234.56"

# 字符串答案
原始答案: "The United States of America"
归一化后: "united states of america"

# 列表答案
原始答案: "Paris, London, Berlin"
归一化后: "berlin, london, paris"
```

(3) GAIA 评估指标

GAIA 使用以下指标评估智能体性能:

1. 精确匹配率 (Exact Match Rate)

精确匹配率是 GAIA 的核心指标, 定义为准精确匹配成功的样本比例:

```
$$ \text{Exact Match Rate} = \frac{1}{N} \sum_{i=1}^N \text{Quasi\_Exact\_Match}(A_{\text{pred},i}, A_{\text{true},i}) $$
```

其中: - N 是总样本数 - $A_{\text{pred},i}$ 是第 i 个样本的预测答案 - $A_{\text{true},i}$ 是第 i 个样本的标准答案 - $\text{Quasi_Exact_Match}(\cdot, \cdot) \in \{0, 1\}$ 是准精确匹配函数

2. 分级准确率 (Level-wise Accuracy)

对于每个难度级别 $\ell \in \{1, 2, 3\}$, 计算该级别的准确率:

```
$$ \text{Accuracy}_{\ell} = \frac{1}{|D_{\ell}|} \sum_{i \in D_{\ell}} \text{Quasi\_Exact\_Match}(A_{\text{pred},i}, A_{\text{true},i}) $$
```

其中 D_{ℓ} 是难度级别 ℓ 的样本集合, $|D_{\ell}|$ 是该级别的样本数。

3. 难度递进下降率 (Difficulty Progression Drop Rate)

衡量智能体在难度增加时的性能衰减:

```
$$ \text{Drop Rate}_{\ell_1 \rightarrow \ell_2} = \frac{\text{Accuracy}_{\ell_1} - \text{Accuracy}_{\ell_2}}{\text{Accuracy}_{\ell_1}} 
```

- $\text{Drop Rate}_{1 \rightarrow 2}$: 从 Level 1 到 Level 2 的下降率
- $\text{Drop Rate}_{2 \rightarrow 3}$: 从 Level 2 到 Level 3 的下降率

4. 平均推理步骤数 (Average Reasoning Steps)

评估智能体完成任务所需的平均步骤数:

```
$$ \text{Avg Steps} = \frac{1}{N_{\text{correct}}} \sum_{i \in \text{Correct}} \text{steps}_i 
```

其中 N_{correct} 是正确回答的样本数, steps_i 是第 i 个样本的推理步骤数。

指标解释:

- **Exact Match Rate = 1.0**: 所有样本都完全正确
- **Exact Match Rate = 0.5**: 50% 的样本正确, 50% 的样本错误
- **Drop Rate = 0.3**: 难度增加导致准确率下降 30%
- **Drop Rate = 0.0**: 难度增加不影响准确率 (理想情况)

评估示例：

假设我们评估了 10 个样本，结果可以参考表 12.4 所示：

表 12.4 GAIA 数据集难度级别分布



如果要计算这个案例的指标的话，可以参考下面的 Python 脚本。

```
# 1. 精确匹配率
total_samples = 10
correct_samples = 7 # 样本1,2,3,5,6,8,9
exact_match_rate = correct_samples / total_samples = 0.70 # 70%

# 2. 分级准确率
level_1_correct = 3 # 样本1,2,3
level_1_total = 3
level_1_accuracy = 3 / 3 = 1.00 # 100%

level_2_correct = 2 # 样本5,6
level_2_total = 3
level_2_accuracy = 2 / 3 = 0.67 # 67%

level_3_correct = 2 # 样本8,9
level_3_total = 4
level_3_accuracy = 2 / 4 = 0.50 # 50%

# 3. 难度递进下降率
drop_rate_1_to_2 = (1.00 - 0.67) / 1.00 = 0.33 # 33%
drop_rate_2_to_3 = (0.67 - 0.50) / 0.67 = 0.25 # 25%

print(f"精确匹配率: {exact_match_rate:.2%}") # 70.00%
print(f"Level 1准确率: {level_1_accuracy:.2%}") # 100.00%
print(f"Level 2准确率: {level_2_accuracy:.2%}") # 66.67%
print(f"Level 3准确率: {level_3_accuracy:.2%}") # 50.00%
print(f"Level 1→2 下降率: {drop_rate_1_to_2:.2%}") # 33.00%
print(f"Level 2→3 下降率: {drop_rate_2_to_3:.2%}") # 25.00%
```

结果分析：

- 整体表现：70%的精确匹配率，表现良好
- 难度敏感性：从 Level 1 到 Level 2 下降 33%，说明智能体在中等难度任务上有明显衰减
- 能力边界：Level 3 准确率为 50%，说明智能体在复杂任务上仍有提升空间

下降率越大，说明智能体在处理复杂任务时的能力衰减越明显。

(4) GAIA 官方系统提示词

GAIA 要求使用特定的系统提示词，确保模型输出符合评估格式：

```
GAIA_SYSTEM_PROMPT = """You are a general AI assistant. I will ask you a question. Report your thoughts, and finish your answer with the foll
YOUR FINAL ANSWER should be a number OR as few words as possible OR a comma separated list of numbers and/or strings.

If you are asked for a number, don't use comma to write your number neither use units such as $ or percent sign unless specified otherwise.

If you are asked for a string, don't use articles, neither abbreviations (e.g. for cities), and write the digits in plain text unless specifi
If you are asked for a comma separated list, apply the above rules depending of whether the element to be put in the list is a number or a st

GAIA 对答案格式有严格的要求：答案必须以FINAL ANSWER: [答案]的格式给出；对于数字类型的答案，不使用逗号分隔符和单位符号；对于字符串类型的答
案、不使用冠词和缩写；对于列表类型的答案，使用逗号分隔并按字母顺序排列。
```

12.3.2 获取 GAIA 数据集

重要提示：GAIA 是受限数据集（Gated Dataset），需要先在 HuggingFace 上申请访问权限。

步骤 1：申请访问权限

1. 访问 <https://huggingface.co/datasets/gaia-benchmark/GAIA>
2. 点击“Request access”按钮
3. 填写申请表单（通常会在几秒内批准）
4. 获取你的 HuggingFace Token：<https://huggingface.co/settings/tokens>

步骤 2：配置环境变量

在 .env 文件中添加你的 HuggingFace Token：

```
# HuggingFace API 配置
HF_TOKEN=hf_your_token_here
```

方法 1：使用 HelloAgents 自动下载（推荐）

HelloAgents 会自动处理 GAIA 数据集的下载和缓存：

```
from hello_agents.evaluation import GAIADataset
import os

# 确保设置了HF_TOKEN，如果设置了.env无需这一行
os.environ["HF_TOKEN"] = "hf_your_token_here"

# 自动下载到 ./data/gaia/
dataset = GAIADataset(
    dataset_name="gaia-benchmark/GAIA",
    split="validation", # 或 "test"
    level=1 # 可选: 1, 2, 3, None(全部)
)
items = dataset.load()

print(f"加载了 {len(items)} 个测试样本")
# 输出: 加载了 53 个测试样本 (Level 1)
```

工作原理：

- 首次运行时，使用snapshot_download下载整个数据集到./data/gaia/
- 数据集包含 114 个文件（问题、图片、PDF 等材料）
- 后续使用直接从本地加载，速度很快

数据集目录结构：

```
./data/gaia/
├── 2023/
│   ├── validation/
│   │   ├── metadata.jsonl (165个问题)
│   │   └── *.png, *.pdf, *.csv, *.xlsx (附件文件)
│   └── test/
│       ├── metadata.jsonl (301个问题)
│       └── ... (附件文件)
└── GAIA.py
└── README.md
```

方法 2：手动下载

如果你想手动下载数据集：

```
from huggingface_hub import snapshot_download
import os

# 设置Token
os.environ["HF_TOKEN"] = "hf_your_token_here"

# 下载数据集
snapshot_download(
    repo_id="gaia-benchmark/GAIA",
    repo_type="dataset",
    local_dir="../data/gaia",
    token=os.getenv("HF_TOKEN")
)
```

查看数据集统计：

```
# 查看数据集统计
stats = dataset.get_statistics()
print(f"总样本数: {stats['total_samples']}")
print(f"级别分布: {stats['level_distribution']}")
# 输出:
# 总样本数: 165
# 级别分布: {1: 53, 2: 62, 3: 50}
```

12.3.3 在 HelloAgents 中实现 GAIA 评估

与 BFCL 类似，我们提供两种评估方式，推荐使用方式 1。

方式 1：使用 GAIAEvaluationTool 一键评估

这是最简单的方式，自动完成数据集下载、评估执行、结果导出和报告生成：

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import GAIAEvaluationTool

# GAIA官方系统提示词（来自论文）
GAIA_SYSTEM_PROMPT = """You are a general AI assistant. I will ask you a question. Report your thoughts, and finish your answer with the foll
YOUR FINAL ANSWER should be a number OR as few words as possible OR a comma separated list of numbers and/or strings.

If you are asked for a number, don't use comma to write your number neither use units such as $ or percent sign unless specified otherwise.
```

2/9/26, 1:56 PM

```
If you are asked for a string, don't use articles, neither abbreviations (e.g. for cities), and write the digits in plain text unless specific
If you are asked for a comma separated list, apply the above rules depending of whether the element to be put in the list is a number or a string

# 1. 创建智能体（使用GAIA官方系统提示词）
llm = HelloAgentsLLM()
agent = SimpleAgent(
    name="TestAgent",
    llm=llm,
    system_prompt=GAIA_SYSTEM_PROMPT # 关键：使用GAIA官方提示词
)

# 2. 创建GAIA评估工具
gaia_tool = GAIAEvaluationTool()

# 3. 一键运行评估
results = gaia_tool.run(
    agent=agent,
    level=1, # Level 1: 简单任务
    max_samples=5, # 评估5个样本
    export_results=True, # 导出GAIA格式结果
    generate_report=True # 生成评估报告
)

# 4. 查看结果
print(f"精确匹配率: {results['exact_match_rate']:.2%}")
print(f"部分匹配率: {results['partial_match_rate']:.2%}")
print(f"正确数: {results['exact_matches']}/{results['total_samples']}")
```

运行结果:

```
=====
GAIA一键评估
=====

配置:
智能体: TestAgent
难度级别: 1
样本数量: 5

=====
步骤1: 运行HelloAgents评估
=====
正在从HuggingFace下载: gaia-benchmark/GAIA
    📥 下载GAIA数据集...
    ✓ 数据集下载完成
    ✓ 加载了 165 个样本
    ✅ GAIA数据集加载完成
    数据源: gaia-benchmark/GAIA
    分割: validation
    级别: 1
    样本数: 53

    🌟 开始 GAIA 评估...
    样本数量: 5
    进度: 5/5
    ✅ GAIA 评估完成
    精确匹配率: 80.00%
    部分匹配率: 80.00%

=====
步骤2: 导出GAIA格式结果
=====
    ✅ GAIA格式结果已导出
    输出文件: evaluation_results\gaia_official\gaia_level1_result_20251011_012648.jsonl
    样本数: 5
    包含推理轨迹: True
    📄 提交说明已生成: evaluation_results\gaia_official\SUBMISSION_GUIDE_20251011_012648.md

=====
步骤3: 生成评估报告
=====
    📄 报告已生成: evaluation_reports\gaia_report_20251011_012648.md

=====
    🎯 最终结果
=====
    精确匹配率: 80.00%
    部分匹配率: 80.00%
    正确数: 4/5
```

评估完成后会自动生成三类文件：首先是 GAIA 格式结果文件（evaluation_results/gaia_official/gaia_level1_result_*.jsonl），采用 JSONL 格式（每行一个 JSON 对象），可直接用于提交到 GAIA 排行榜；其次是提交说明文件（evaluation_results/gaia_official/SUBMISSION_GUIDE_*.md），包含详细的提交步骤、结果文件格式说明和注意事项；最后是评估报告（evaluation_reports/gaia_report_*.md），包含评估结果摘要、详细指标、样本详情和可视化图表。

注意：如果你发现生成的评估结果不理想（例如准确率较低），这是正常现象。虽然 Level 1 是一步推理任务，但仍然需要智能体具备工具调用能力（如搜索引擎、计算器等）才能正确回答问题。我们当前使用的 SimpleAgent 主要用于演示评估流程，在工具调用能力上还有提升空间。

方式 2：使用 Dataset + Evaluator（灵活定制）

如果需要更细粒度的控制，可以直接使用底层组件：

```
from hello_agents.evaluation import GAIADataset, GAIAEvaluator

# 1. 加载数据集
dataset = GAIADataset(level=1)
items = dataset.load()
print(f"加载了 {len(items)} 个样本")

# 2. 创建评估器
evaluator = GAIAEvaluator(dataset=dataset, level=1)

# 3. 运行评估
results = evaluator.evaluate(agent, max_samples=5)

# 4. 导出GAIA格式结果
evaluator.export_to_gaia_format(
    results,
    "gaia_results.jsonl",
    include_reasoning=True
)
```

生成的评估报告 (gaia_report_*.md) 可参考下面的文件：

```
# GAIA评估报告

**生成时间**: 2025-10-11 01:26:48

## 📈 评估概览

- **智能体**: TestAgent
- **难度级别**: 1
- **总样本数**: 2
- **精确匹配数**: 1
- **部分匹配数**: 1
- **精确匹配率**: 50.00%
- **部分匹配率**: 50.00%
```

📈 详细指标

分级准确率

- **Level 1**: 50.00% 精确 / 50.00% 部分 (1/2)

📈 样本详情 (前10个)

| 任务ID | 级别 | 预测答案 | 正确答案 | 精确匹配 | 部分匹配 |
|--------------------------------------|----|-------|------|------|------|
| e1fc63a2-da7a-432f-be78-7c4a95598703 | 1 | 24000 | 17 | ✗ | ✗ |
| 8e867cd7-cff9-4e6c-867a-ff5ddc2550be | 1 | 3 | 3 | ✓ | ✓ |

📈 准确率可视化



💡 建议

- 🌟 表现一般，需要改进。
- 💡 建议检查工具使用和多步推理能力。

**生成的 GAIA 格式结果 (gaia_level1_result_*.jsonl) :

```
{"task_id": "e1fc63a2-da7a-432f-be78-7c4a95598703", "model_answer": "24000", "reasoning_trace": "24000"}  
{"task_id": "8e867cd7-cff9-4e6c-867a-ff5ddc2550be", "model_answer": "3", "reasoning_trace": "3"}
```

12.3.4 提交结果到 GAIA 官方排行榜

使用 GAIAEvaluationTool 运行评估后，会在 evaluation_results/gaia_official/ 目录下生成提交所需的文件和详细的提交说明。

1. GAIA 格式结果文件**: gaia_level1_result_*.jsonl

```
{"task_id": "xxx", "model_answer": "答案", "reasoning_trace": "推理过程"}  
{"task_id": "yyy", "model_answer": "答案", "reasoning_trace": "推理过程"}
```

2. 提交说明文件: SUBMISSION_GUIDE_*.md

打开自动生成的SUBMISSION_GUIDE_*.md文件，里面包含完整的提交指南：

具体来说，打开浏览器，访问：

<https://huggingface.co/spaces/gaia-benchmark/leaderboard>

如图 12.4 所示，提交表单中填写信息即可：



图 12.4 BFCL 评估流程图

提交前，可以手动检查生成的 JSONL 文件：

```
import json

# 读取结果文件
with open("evaluation_results/gaia_official/gaia_level1_result_*.jsonl", "r") as f:
    for line in f:
        result = json.loads(line)
        print(f"Task ID: {result['task_id']}")
        print(f"Answer: {result['model_answer']}")
        print(f"Reasoning: {result['reasoning_trace']}")
        print("-" * 50)
```

12.3.5 核心组件实现细节

GAIA 评估系统的实现与 BFCL 类似，但针对通用能力评估有一些特殊的设计。

(1) GAIADataset：支持多模态的数据加载器

GAIA 数据集的特殊之处在于它包含多模态数据（文本、文件、图片等）：

```
class GAIADataset:
    """GAIA数据集加载器

    支持从HuggingFace加载GAIA数据集（受限数据集）
    """

    def __init__(
        self,
        level: Optional[int] = None,
        split: str = "validation",
        local_data_dir: Optional[str] = None
    ):
        self.level = level
        self.split = split
        self.local_data_dir = local_data_dir or "./data/gaia"
        self.data = []

    def load(self) -> List[Dict[str, Any]]:
        """加载数据集"""
        # 从HuggingFace下载
        items = self._load_from_huggingface()

        # 按级别过滤
        if self.level:
            items = [item for item in items if item.get("level") == self.level]

        self.data = items
        return items

    def _load_from_huggingface(self) -> List[Dict[str, Any]]:
        """从HuggingFace下载GAIA数据集"""
        from huggingface_hub import snapshot_download
        import json

        # 下载数据集
        repo_id = "gaia-benchmark/GAIA"
        local_dir = snapshot_download(
            repo_id=repo_id,
            repo_type="dataset",
            local_dir=self.local_data_dir,
            local_dir_use_symlinks=False
        )

        # 加载JSONL文件
        data_file = Path(local_dir) / "2023" / self.split / "metadata.jsonl"
        items = []
        with open(data_file, 'r', encoding='utf-8') as f:
            for line in f:
                item = json.loads(line)
                items.append(self._standardize_item(item))

        return items
```

(2) GAIAEvaluator：实现 GAIA 官方评估算法

GAIA 的评估使用准精确匹配（Quasi Exact Match）算法，需要特殊的答案归一化和匹配逻辑：

2/9/26, 1:56 PM

```
class GAIAEvaluator:  
    """GAIA评估器  
  
    实现GAIA官方的准精确匹配 (Quasi Exact Match) 评估算法  
    """  
  
    def evaluate(self, agent: Any, max_samples: Optional[int] = None) -> Dict[str, Any]:  
        """执行评估"""  
        dataset_items = self.dataset.load()  
  
        if max_samples:  
            dataset_items = dataset_items[:max_samples]  
  
        results = []  
        for i, item in enumerate(dataset_items, 1):  
            # 1. 构造提示词  
            prompt = self._build_prompt(item["question"], item)  
  
            # 2. 调用智能体  
            response = agent.run(prompt)  
  
            # 3. 提取答案 (GAIA格式: FINAL ANSWER: [答案])  
            predicted_answer = self._extract_answer(response)  
  
            # 4. 归一化答案 (GAIA官方规则)  
            normalized_pred = self._normalize_answer(predicted_answer)  
            normalized_truth = self._normalize_answer(item["final_answer"])  
  
            # 5. 准精确匹配  
            exact_match = (normalized_pred == normalized_truth)  
  
            results.append({  
                "task_id": item["task_id"],  
                "predicted": predicted_answer,  
                "expected": item["final_answer"],  
                "exact_match": exact_match,  
                "level": item.get("level", 0)  
            })  
  
        return self._format_results(results)
```

GAIA 使用特定的归一化规则来处理不同类型的答案:

```
def _normalize_answer(self, answer: str) -> str:  
    """标准化答案字符串 (GAIA官方标准化规则)  
  
    规则:  
    1. 数字: 移除逗号分隔符和单位符号  
    2. 字符串: 移除冠词、转小写、移除多余空格  
    3. 列表: 逗号分隔, 按字母顺序排序  
    """  
  
    if not answer:  
        return ""  
  
    answer = answer.strip()  
  
    # 检查是否是逗号分隔的列表  
    if ',' in answer:  
        parts = [self._normalize_single_answer(p.strip()) for p in answer.split(',')]  
        parts.sort() # GAIA要求按字母顺序排序  
        return ','.join(parts)  
    else:  
        return self._normalize_single_answer(answer)  
  
def _normalize_single_answer(self, answer: str) -> str:  
    """标准化单个答案 (不包含逗号的答案) """  
    answer = answer.strip().lower()  
  
    # 移除常见的冠词  
    articles = ['the', 'a', 'an']  
    words = answer.split()  
    if words and words[0] in articles:  
        words = words[1:]  
        answer = ' '.join(words)  
  
    # 移除货币符号和百分号  
    answer = answer.replace('$', '').replace('%', '').replace('€', '').replace('£', '')  
  
    # 移除数字中的逗号分隔符  
    answer = re.sub(r'(\d),(\d)', r'\1\2', answer)  
  
    # 移除多余空格  
    answer = ' '.join(answer.split())  
  
    # 移除末尾的标点符号  
    answer = answer.rstrip('.,:!?)')  
  
    return answer
```

GAIA 要求模型输出格式为FINAL ANSWER: [答案]:

2/9/26, 1:56 PM

```
def _extract_answer(self, response: str) -> str:  
    """从响应中提取答案 (GAIA格式)  
  
    GAIA要求答案格式为: FINAL ANSWER: [答案]  
    """  
    # 首先尝试提取GAIA官方格式的答案  
    final_answer_pattern = r'FINAL ANSWER:\s*(.+?)(?:\n|$)'  
    match = re.search(final_answer_pattern, response, re.IGNORECASE | re.MULTILINE)  
    if match:  
        answer = match.group(1).strip()  
        # 移除可能的方括号  
        answer = answer.strip('[]')  
        return answer  
  
    # 备用方案: 查找其他答案标记  
    answer_patterns = [  
        r'答案[: :]\s*(.+)',  
        r'最终答案[: :]\s*(.+)',  
        r'Final answer[: :]\s*(.+)',  
        r'Answer[: :]\s*(.+)',  
    ]  
  
    for pattern in answer_patterns:  
        match = re.search(pattern, response, re.IGNORECASE)  
        if match:  
            return match.group(1).strip()  
  
    # 如果没有找到标记, 返回最后一个非空行  
    lines = response.strip().split('\n')  
    for line in reversed(lines):  
        line = line.strip()  
        if line and not line.startswith('#'):  
            return line  
  
    return response.strip()
```

评估完成后, 可以导出为 GAIA 官方要求的 JSONL 格式:

```
def export_to_gaia_format(  
    self,  
    results: Dict[str, Any],  
    output_path: Union[str, Path],  
    include_reasoning: bool = True  
) -> None:  
    """导出为GAIA官方格式 (JSONL)  
  
    GAIA要求的格式:  
    {"task_id": "xxx", "model_answer": "答案", "reasoning_trace": "推理过程"}  
    """  
    output_path = Path(output_path)  
    output_path.parent.mkdir(parents=True, exist_ok=True)  
  
    with open(output_path, 'w', encoding='utf-8') as f:  
        for result in results.get("detailed_results", []):  
            entry = {  
                "task_id": result["task_id"],  
                "model_answer": result["predicted"]  
            }  
  
            if include_reasoning:  
                entry["reasoning_trace"] = result.get("response", result["predicted"])  
            f.write(json.dumps(entry, ensure_ascii=False) + '\n')
```

(3) GAIAEvaluationTool: 一键评估工具

GAIAEvaluationTool 封装了完整的评估流程, 提供一键评估功能:

```
class GAIAEvaluationTool(Tool):  
    """GAIA评估工具  
  
    提供一键评估功能:  
    1. 运行HelloAgents评估  
    2. 导出GAIA格式结果  
    3. 生成评估报告  
    4. 生成提交说明  
    """  
  
    def run(  
        self,  
        agent: Any,  
        level: Optional[int] = None,  
        max_samples: Optional[int] = None,  
        local_data_dir: Optional[str] = None,  
        export_results: bool = True,  
        generate_report: bool = True  
    ) -> Dict[str, Any]:  
        """执行GAIA一键评估"""  
        # 步骤1: 运行HelloAgents评估  
        results = self._run_evaluation(agent, level, max_samples, local_data_dir)  
  
        # 步骤2: 导出GAIA格式结果
```

```

if export_results:
    self._export_results(results)

# 步骤3：生成评估报告
if generate_report:
    self.generate_report(results)

return results

```

GAIAEvaluationTool 会自动生成评估报告：

```

def generate_report(
    self,
    results: Dict[str, Any],
    output_file: Optional[Union[str, Path]] = None
) -> str:
    """生成评估报告"""
    report = f"""# GAIA评估报告

**生成时间**: {datetime.now().strftime("%Y-%m-%d %H:%M:%S")}

## 📈 评估概览

- **智能体**: {results.get("agent_name", "Unknown")}
- **难度级别**: {results.get("level_filter") or '全部'}
- **总样本数**: {results.get("total_samples", 0)}
- **精确匹配数**: {results.get("exact_matches", 0)}
- **精确匹配率**: {results.get("exact_match_rate", 0):.2%}

## 📈 详细指标

### 分级准确率

{self._format_level_metrics(results.get("level_metrics", {}))}

## 📈 样本详情 (前10个)

{self._format_sample_details(results.get("detailed_results", [])[:10])}

## 📈 准确率可视化

{self._format_visualization(results.get("exact_match_rate", 0))}

## 💡 建议

{self._format_suggestions(results.get("exact_match_rate", 0))}

# 保存报告
if output_file is None:
    output_dir = Path("./evaluation_reports")
    output_dir.mkdir(parents=True, exist_ok=True)
    output_file = output_dir / f"gaia_report_{datetime.now().strftime('%Y%m%d_%H%M%S')}.md"
with open(output_file, 'w', encoding='utf-8') as f:
    f.write(report)

return report

```

12.4 数据生成质量评估

在 AI 系统开发中，高质量的训练数据是系统性能的基础。本节介绍如何使用 HelloAgents 框架评估生成数据的质量，以 AIME（美国数学邀请赛）^[9]风格的数学题目生成为例。

AIME 是美国数学协会（MAA）主办的中等难度数学竞赛，介于 AMC 10/12 和美国数学奥林匹克（USAMO）之间。AIME 题目具有鲜明的特点：每道题的答案都是 0 到 999 之间的整数，题目涵盖代数、几何、数论、组合、概率等多个数学领域，需要多步推理但不涉及高深理论，难度适中（相当于 AIME 第 6-9 题的水平）。这些特点使得 AIME 题目成为评估数学题目生成质量的理想基准：答案格式统一便于自动化评估，题目难度适中适合大规模生成。我们使用 HuggingFace 上的 TianHongZXY/aime-1983-2025 数据集作为参考，该数据集包含从 1983 年到 2025 年的 900 多道 AIME 真题，为我们的生成和评估提供了丰富的参考样本。

12.4.1 评估方法概述

在数据生成质量评估中，我们采用三种互补的评估方法：LLM Judge、Win Rate 和人工打分。选择这三种方法有两个重要原因。首先，从方法论角度来看，这些都是当前智能体领域常用的自动化测评方案，也是许多学术论文中的主流做法，具有广泛的认可度和实践基础。其次，从适用性角度来看，这三种方法天然适合我们的评估场景：LLM Judge 和 Win Rate 用于评估题目生成质量（从正确性、清晰度、难度匹配等维度进行多维度评估），而人工打分用于评估答案生成质量（通过人类专家验证答案的准确性），这种分工非常合理且易于理解。

下面我们详细介绍这三种评估方法的具体实现。整个案例的实现流程如图 12.5 所示：



图 12.5 数据生成质量评估流程图

(1) LLM Judge 评估

设计动机：在数据生成质量评估中，我们需要对大量生成的题目进行快速、一致的质量评估。传统的人工评估虽然准确，但成本高、效率低，难以应对大规模数据生成的需求。LLM Judge 通过使用大语言模型作为评委，可以自动化地从多个维度评估生成数据的质量，不仅大幅提升评估效率，还能保持评估标准的一致性。更重要的是，LLM Judge 可以提供详细的评分理由和改进建议，帮助我们理解生成数据的优缺点，为后续优化提供方向。

在我们的实现中，LLM Judge 从四个关键维度评估 AIME 题目的质量：

表 12.5 LLM Judge 评估 AIME 题目的维度



有了四个维度的评分后，我们需要将这些评分汇总成整体的评估指标。我们定义了三个关键指标来衡量生成题目的质量水平：

评估指标：

1. 平均分 (Average Score)：计算所有题目在四个维度上的平均得分，反映生成题目的整体质量水平。 $\text{Average Score} = \frac{1}{N} \sum_{i=1}^N S_{i,d}$

2. 及格率 (Pass Rate)：统计平均分达到 3.5 分及以上的题目比例，反映生成题目的基本质量保障。

$$\text{Pass Rate} = \frac{\{i : \text{Score}_i \geq 3.5\}}{N}$$

3. 优秀率 (Excellent Rate)：统计平均分达到 4.5 分及以上的题目比例，反映生成题目的高质量占比。

$$\text{Excellent Rate} = \frac{\{i : \text{Score}_i \geq 4.5\}}{N}$$

其中： - N 是评估的题目总数 - $S_{i,d}$ 是第 i 个题目在第 d 个维度的得分（1-5 分） - Score_i 是第 i 个题目的平均分（四个维度得分的平均值）

这三个指标从不同角度反映生成质量：平均分给出整体水平，及格率保证基本质量，优秀率衡量高质量产出能力。

(2) Win Rate 评估

设计动机：虽然 LLM Judge 可以提供多维度的绝对评分，但我们还需要一个相对评估指标来衡量生成题目与真题的质量差距。Win Rate 评估通过成对对比的方式，让 LLM 直接判断生成题目和真题哪个更好，这种相对比较比绝对评分更符合人类的判断习惯，也更容易发现生成题目的相对优势和劣势。理想情况下，如果生成题目的质量接近真题，Win Rate 应该在 50% 左右（即生成题目和真题各有 50% 的胜率）。这个指标简单直观，可以快速判断生成系统的整体质量水平。

在我们的实现中，Win Rate 评估通过以下图 12.6 所示流程进行评估：



图 12.6 数据生成质量评估流程图

在成对对比评估中，每次比较会产生三种可能的结果：生成题目获胜 (Win)、真题获胜 (Loss) 或平局 (Tie)。我们通过统计这三种结果的比例来评估生成题目的质量：

评估指标：

1. 胜率 (Win Rate)：生成题目被判定为更好的比例，反映生成题目相对于真题的优势。

$$\text{Win Rate} = \frac{\text{Wins}}{\text{Total Comparisons}}$$

2. 败率 (Loss Rate)：真题被判定为更好的比例，反映生成题目相对于真题的劣势。

$$\text{Loss Rate} = \frac{\text{Losses}}{\text{Total Comparisons}}$$

3. 平局率 (Tie Rate)：两者被判定为质量相当的比例，反映生成题目与真题的相似程度。

$$\text{Tie Rate} = \frac{\text{Ties}}{\text{Total Comparisons}}$$

其中，Total Comparisons 是总的对比次数，Wins、Losses 和 Ties 分别是生成题目获胜、失败和平局的次数。这三个指标满足： $\text{Win Rate} + \text{Loss Rate} + \text{Tie Rate} = 100\%$ 。

理想结果：Win Rate $\approx 50\%$ （说明生成质量接近真题）。如果 Win Rate 显著低于 50%，说明生成题目质量不如真题，需要优化生成策略；如果 Win Rate 显著高于 50%，可能说明生成题目在某些方面超越了真题，或者评估标准存在偏差。

(3) 人工验证

设计动机：尽管 LLM Judge 和 Win Rate 可以自动化评估题目质量，但对于数学题目这种需要严格逻辑推理的内容，人工验证仍然是不可或缺的。特别是在评估答案生成质量时，需要人类专家验证答案的准确性、解答步骤的完整性和数学推论的严密性。此外，人工验证还可以发现自动化评估可能遗漏的问题，如题目的创新性、趣味性等主观因素。为了提高人工验证的效率和体验，我们开发了基于 Gradio 的 Web 界面，让验证者可以方便地浏览题目、评分、标注状态和添加评论，大大降低了人工验证的门槛。

在我们的实现中，人工验证通过以下步骤进行：

1. 阅读题目、答案、解答
2. 评分 (1-5 分)：正确性、清晰度、难度匹配、完整性
3. 标注状态：
 - approved (通过)
 - rejected (拒绝)
 - needs_revision (需修改)
4. 添加评论

12.4.2 系统架构

数据生成与评估系统采用模块化设计：

```
data_generation/
└── aime_generator.py          # AIME题目生成器
└── human_verification_ui.py   # 人工验证界面
└── run_complete_evaluation.py # 完整评估流程

└── generated_data/           # 生成的数据
    └── aime_generated_XXXXXX.json
    └── generation_report_XXXXXX.md

└── evaluation_results/       # 评估结果
    └── XXXXXX/
        └── llm_judge/
        └── win_rate/
        └── comprehensive_report.md
```

系统包含四个核心组件：首先是 AIMEGenerator（题目生成器），使用 HelloAgents 框架生成 AIME 风格题目，支持批量生成和进度保存，并能自动处理 API 速率限制；其次是 LLMDJUDGETool（LLM Judge 评估工具），提供 4 维度质量评估，自动生成 JSON 结果和 Markdown 报告；第三是 WinRateTool（Win Rate 评估工具），通过成对对比评估计算胜率、败率和平局率；最后是 HumanVerificationUI（人工验证界面），基于 Gradio Web 界面，支持评分和状态标注。

12.4.3 AIME 题目生成器实现

```
class AIMEGenerator:
    """AIME Problem Generator"""

    def __init__(self,
                 llm: HelloAgentsLLM = None,
                 delay_seconds: float = 1.0,
                 use_reference_examples: bool = True,
                 reference_dataset: str = "TianHongZXY/aime-1983-2025"):
        self.llm = llm or HelloAgentsLLM()
        self.agent = SimpleAgent(
            name="AIME Generator",
            llm=self.llm,
            system_prompt="You are a professional mathematics competition problem designer."
        )
        self.delay_seconds = delay_seconds
        self.use_reference_examples = use_reference_examples

    # Load reference examples from 900+ AIME problems (1983–2025)
    if use_reference_examples:
        dataset = load_dataset(reference_dataset, split="test")
        self.reference_examples = list(dataset)
```

我们的目标是生成类似风格的数据集，所以从 900+ 道 AIME 真题（1983-2025）中随机选择参考样例

生成提示词设计（英文）：

```
GENERATION_PROMPT = """You are a professional mathematics competition problem designer, skilled in creating AIME (American Invitational Mathe
```

```
[Reference Example] (For style reference only, please generate a completely different problem)
Problem: {example_problem}
Answer: {example_answer}
```

```
AIME Problem Characteristics:
1. Answer: An integer between 0 and 999
2. Topics: Algebra, Geometry, Number Theory, Combinatorics, Probability, etc.
3. Style: Requires multi-step reasoning, but no advanced theory
4. Difficulty: Medium to hard (similar to AIME problems 6–9)
```

```
Please generate a **completely different** AIME-style mathematics problem, including:
1. Problem statement (clear and complete, different from the reference)
2. Answer (an integer between 0 and 999, different from the reference)
3. Detailed solution (including all reasoning steps)
4. Topic classification (Algebra/Geometry/Number Theory/Combinatorics/Probability)
```

```
Please output in the following JSON format:
{
  "problem": "Problem statement in English",
  "answer": 123,
  "solution": "Detailed solution steps in English",
  "topic": "Algebra"
}"""

```

我们选择使用英文生成题目有四个重要原因：首先是与 AIME 真题保持一致（AIME 是英文竞赛，生成英文题目更合理），其次是确保评估的公平性（LLM Judge 评估时英文 vs 英文更公平），第三是便于国际化（英文题目可以被更广泛使用），最后是避免翻译问题（不需要担心中文翻译的准确性）。

批量生成实现：

```
def generate_and_save(self, num_problems: int = 30, output_dir: str = "data_generation/generated_data"):
    """Generate and save problems with intelligent delay"""
    # Clean old checkpoints
    for file in os.listdir(output_dir):
        if file.startswith("checkpoint_") and file.endswith(".json"):
            os.remove(os.path.join(output_dir, file))

    # Generate with tqdm progress bar
    with tqdm(total=num_problems, desc="Generating AIME problems", unit="problem") as pbar:
        last_call_time = 0

        for i in range(num_problems):
            # Ensure minimum delay between API calls
            if last_call_time > 0:
                elapsed = time.time() - last_call_time
                if elapsed < self.delay_seconds:
                    wait_time = self.delay_seconds - elapsed
                    time.sleep(wait_time)

            # Generate problem (randomly select reference example)
            start_time = time.time()
            problem = self.generate_single()
            last_call_time = time.time()
            generation_time = last_call_time - start_time

            # Update progress bar
            pbar.set_postfix({
                "topic": problem.get('topic', 'N/A'),
                "answer": problem.get('answer', 'N/A'),
                "time": f'{generation_time:.1f}s'
            })
            pbar.update(1)

    return generated_data_path
```

LaTeX 数学公式支持：

生成的 AIME 题目包含 LaTeX 数学公式（如 $\frac{a}{b}$ 、 \sqrt{x} ），需要特殊处理 JSON 解析：

```
def _parse_response(self, response: str) -> Dict[str, Any]:
    """解析LLM响应（支持LaTeX数学公式）"""
    import re

    # 提取JSON部分
    if "```json" in response:
        json_str = response.split("```json")[1].split("```")[0].strip()
    else:
        json_str = response.strip()

    try:
        problem_data = json.loads(json_str)
    except json.JSONDecodeError:
        # 修复LaTeX转义问题：将 \frac 转为 \\frac
        # 正则表达式：找到未转义的反斜杠
        fixed_json_str = re.sub(r'(?![!\\])\\(?![!"\\bf\\fr\\ntu])', r'\\\\', json_str)
        problem_data = json.loads(fixed_json_str)

    return problem_data
```

LaTeX 公式中的反斜杠（如 \frac 、 \sqrt ）在 JSON 中是非法的转义字符，会导致解析失败：

Invalid \escape: line 4 column 185 (char 375)

通过正则表达式将未转义的反斜杠替换为双反斜杠，使其在 JSON 中合法。

12.4.4 LLM Judge 评估工具

LLM Judge 工具使用 LLM 作为评委，对生成的题目进行多维度评估。

```
class LLMDJTool(Tool):
    """LLM Judge评估工具"""

    def run(self, params: Dict[str, Any]) -> str:
        """运行LLM Judge评估"""
        # 1. 加载生成数据
        gen_dataset = AIIDataset(dataset_type="generated", data_path=params["generated_data_path"])
        gen_problems = gen_dataset.load()

        # 2. 加载参考数据 (AIME 2025)
        ref_dataset = AIIDataset(dataset_type="real", year=2025)
        ref_problems = ref_dataset.load()

        # 3. 创建评估器
        evaluator = LLMDJTool(llm=self.llm, judge_model=params.get("judge_model", "gpt-4o"))
```

```

# 4. 运行评估
results = evaluator.evaluate_batch(gen_problems, max_samples=params.get("max_samples"))

# 5. 保存结果
evaluator.export_results(results, result_file)

# 6. 生成报告
self._generate_report(results, report_file)

return json.dumps({"status": "success", "metrics": results["metrics"]})

```

评估提示词:

EVALUATION_PROMPT = """请评估以下AIME数学题目的质量。

题目:

{problem}

答案: {answer}

解答:

{solution}

请从以下4个维度评分 (1-5分):

1. 正确性 (Correctness): 数学逻辑是否正确, 答案是否准确
2. 清晰度 (Clarity): 问题表述是否清晰, 解答是否易懂
3. 难度匹配 (Difficulty Match): 难度是否符合AIME标准 (中等偏难)
4. 完整性 (Completeness): 解答步骤是否完整, 是否包含必要的推理

请按以下JSON格式输出:

```
{
  "correctness": 5,
  "clarity": 4,
  "difficulty_match": 4,
  "completeness": 5,
  "comments": "评价理由"
}
....
```

评估报告示例:

```
# LLM Judge评估报告

## 总体评分

- <strong>平均总分</strong>: 4.2/5.0
- <strong>通过率</strong>: 85.0% (≥3.5分)
- <strong>优秀率</strong>: 40.0% (≥4.5分)

## 各维度评分

| 维度 | 平均分 | 评级 |
```

| 维度 | 平均分 | 评级 |
|------|---------|----------|
| 正确性 | 4.3/5.0 | 良好 ★★★★★ |
| 清晰度 | 4.1/5.0 | 良好 ★★★★★ |
| 难度匹配 | 4.0/5.0 | 良好 ★★★★★ |
| 完整性 | 4.4/5.0 | 良好 ★★★★★ |

12.4.5 Win Rate 评估工具

Win Rate 工具通过成对对比评估生成数据相对于真题的质量。

```

class WinRateTool(Tool):
    """Win Rate评估工具"""

    def run(self, params: Dict[str, Any]) -> str:
        """运行Win Rate评估"""
        # 1. 加载生成数据
        gen_dataset = AIIDataset(dataset_type="generated", data_path=params["generated_data_path"])
        gen_problems = gen_dataset.load()

        # 2. 加载参考数据 (AIME 2025)
        ref_dataset = AIIDataset(dataset_type="real", year=2025)
        ref_problems = ref_dataset.load()

        # 3. 创建评估器
        evaluator = WinRateEvaluator(llm=self.llm, judge_model=params.get("judge_model", "gpt-4o"))

        # 4. 运行评估
        results = evaluator.evaluate_win_rate(gen_problems, ref_problems, num_comparisons=params.get("num_comparisons"))

        # 5. 保存结果和报告
        evaluator.export_results(results, result_file)
        self._generate_report(results, report_file)

        return json.dumps({"status": "success", "metrics": results["metrics"]})

```

AIDataset 负责加载生成数据和 AIME 真题数据，支持两种数据类型：

```
class AIDataset:
    """AI数据集加载器

    支持两种数据类型：
    1. generated: 生成的数据 (JSON格式)
    2. real: AIME真题 (从HuggingFace加载)
    """

    def __init__(
        self,
        dataset_type: str = "generated",
        data_path: Optional[str] = None,
        year: Optional[int] = None
    ):
        self.dataset_type = dataset_type
        self.data_path = data_path
        self.year = year # 仅用于real类型, 默认2025

    def load(self) -> List[Dict[str, Any]]:
        """加载数据集"""
        if self.dataset_type == "generated":
            return self._load_generated_data()
        elif self.dataset_type == "real":
            return self._load_real_data()

    def _load_real_data(self) -> List[Dict[str, Any]]:
        """从HuggingFace加载AIME 2025真题"""
        from huggingface_hub import snapshot_download

        # 使用AIME 2025数据集
        repo_id = "math-ai/aime25"

        # 下载数据集
        local_dir = snapshot_download(
            repo_id=repo_id,
            repo_type="dataset"
        )

        # 读取JSONL文件
        data_file = list(Path(local_dir).glob("*.jsonl"))[0]
        data = []
        with open(data_file, 'r', encoding='utf-8') as f:
            for line in f:
                if line.strip():
                    data.append(json.loads(line))

        # 统一数据格式 (AIME 2025使用小写字段名)
        problems = []
        for idx, item in enumerate(data):
            problem = {
                "problem_id": item.get("id", f"aime_2025_{idx}"),
                "problem": item.get("problem", ""),
                "answer": item.get("answer", ""),
                "solution": item.get("solution", ""), # AIME 2025没有solution字段
            }
            problems.append(problem)

        return problems
```

我们选择只使用 AIME 2025 数据集有四个原因：首先是数据的时效性（2025 年是最新的 AIME 竞赛数据），其次是简化维护（只维护一个数据集，代码更简洁），第三是格式统一（JSONL 格式，字段名统一为小写），最后是代表性充分（30 道题目足以评估生成质量）。

对比提示词：

COMPARISON_PROMPT = "请比较以下两个AIME数学题目的质量，判断哪个更好。

【题目A - 生成题目】

问题: {problem_a}
答案: {answer_a}
解答: {solution_a}

【题目B - AIME真题】

问题: {problem_b}
答案: {answer_b}
解答: {solution_b}

请从以下方面比较：

1. 数学逻辑的严谨性
2. 问题表述的清晰度
3. 难度的合理性
4. 解答的完整性

请按以下JSON格式输出：

```
{
    "winner": "A" 或 "B" 或 "Tie",
    "reason": "判断理由"
}
"""
```

评估报告示例：

```
# Win Rate评估报告
## 胜率统计

指标	数值	百分比
生成数据胜出	9次	45.0%
AIME真题胜出	8次	40.0%
平局	3次	15.0%

<strong>Win Rate</strong>: 45.0%
```

良好：生成数据质量接近参考数据（差距<10%）。

12.4.6 人工验证界面

使用 Gradio 创建 Web 界面，支持人工验证生成的题目。

```
class HumanVerificationUI:
    """人工验证界面"""

    def launch(self, share: bool = False):
        """启动Gradio界面"""
        with gr.Blocks(title="AIME题目人工验证") as demo:
            gr.Markdown("# 🚀 AIME题目人工验证系统")

            with gr.Row():
                with gr.Column(scale=2):
                    # 题目显示区域
                    problem_text = gr.Textbox(label="问题描述", lines=5, interactive=False)
                    answer_text = gr.Textbox(label="答案", interactive=False)
                    solution_text = gr.Textbox(label="解答过程", lines=10, interactive=False)

                with gr.Column(scale=1):
                    # 评分区域
                    correctness_slider = gr.Slider(1, 5, value=3, step=1, label="正确性")
                    clarity_slider = gr.Slider(1, 5, value=3, step=1, label="清晰度")
                    difficulty_slider = gr.Slider(1, 5, value=3, step=1, label="难度匹配")
                    completeness_slider = gr.Slider(1, 5, value=3, step=1, label="完整性")

                    # 状态选择
                    status_radio = gr.Radio(
                        choices=["approved", "rejected", "needs_revision"],
                        value="approved",
                        label="状态"
                    )

            # 验证按钮
            verify_btn = gr.Button("✅ 提交验证", variant="primary")

        demo.launch(share=share, server_name="127.0.0.1", server_port=7860)
```

使用方法：

```
# 启动人工验证界面
python data_generation/human_verification_ui.py data_generation/generated_data/aime_generated_XXXXXX.json

# 打开浏览器访问
http://127.0.0.1:7860
```

最终效果可以参考图 12.7 所示，对于题目的正确性，最好人工打标 Review:



图 12.7 AIME 试题人工验证页面

验证流程：

1. 浏览器打开验证界面
2. 阅读题目、答案、解答
3. 从 4 个维度评分（1-5 分）
4. 选择验证状态 (approved/rejected/needs_revision)
5. 添加评论（可选）
6. 点击“提交验证”
7. 查看下一题

验证结果保存：

验证结果自动保存为 <data_path>_verifications.json:

```
{
    "gen_aime_1": {
```

```

    "problem_id": "gen_aime_1",
    "scores": {
        "correctness": 5,
        "clarity": 4,
        "difficulty_match": 4,
        "completeness": 5
    },
    "total_score": 4.5,
    "status": "approved",
    "comments": "题目质量很好，逻辑严谨",
    "verified_at": "2025-01-10T12:00:00"
}
}

```

12.4.7 完整评估流程

将所有评估方法整合到一个完整的流程中。

```

def run_complete_evaluation(
    num_problems: int = 30,
    delay_seconds: float = 3.0
):
    .....
    运行完整评估流程

Args:
    num_problems: 生成题目数量
    delay_seconds: 每次生成之间的延迟（秒），避免API速率限制
    .....

# 步骤1: 生成AIME题目
generator = AIMEGenerator(delay_seconds=delay_seconds)
generated_data_path = generator.generate_and_save(
    num_problems=num_problems,
    output_dir="data_generation/generated_data"
)

# 步骤2: 评估
# 创建评估结果目录
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
evaluation_dir = f"data_generation/evaluation_results/{timestamp}"
os.makedirs(evaluation_dir, exist_ok=True)
os.makedirs(os.path.join(evaluation_dir, "llm_judge"), exist_ok=True)
os.makedirs(os.path.join(evaluation_dir, "win_rate"), exist_ok=True)

# 创建LLM
llm = HelloAgentsLLM()

# 步骤2.1: LLM Judge评估
llm_judge_result = None
try:
    llm_judge_tool = LLMJudgeTool(llm=llm)
    llm_judge_result_json = llm_judge_tool.run({
        "generated_data_path": generated_data_path,
        "reference_year": 2025,
        "max_samples": num_problems,
        "output_dir": os.path.join(evaluation_dir, "llm_judge"),
        "judge_model": "gpt-4o"
    })
    llm_judge_result = json.loads(llm_judge_result_json)
except Exception as e:
    print(f"✗ LLM Judge评估失败: {e}")

# 步骤2.2: Win Rate评估
win_rate_result = None
try:
    win_rate_tool = WinRateTool(llm=llm)
    win_rate_result_json = win_rate_tool.run({
        "generated_data_path": generated_data_path,
        "reference_year": 2025,
        "num_comparisons": min(num_problems, 20),
        "output_dir": os.path.join(evaluation_dir, "win_rate"),
        "judge_model": "gpt-4o"
    })
    win_rate_result = json.loads(win_rate_result_json)
except Exception as e:
    print(f"✗ Win Rate评估失败: {e}")

# 步骤3: 生成综合报告
comprehensive_report_path = None
if llm_judge_result or win_rate_result:
    comprehensive_report_path = os.path.join(evaluation_dir, "comprehensive_report.md")
    report = generate_comprehensive_report(
        generated_data_path,
        llm_judge_result,
        win_rate_result
    )
    with open(comprehensive_report_path, 'w', encoding='utf-8') as f:
        f.write(report)

return {
    "generated_data_path": generated_data_path,
    "llm_judge_result": llm_judge_result,
    "win_rate_result": win_rate_result,
}

```

2/9/26, 1:56 PM

```
        "comprehensive_report_path": comprehensive_report_path  
    }
```

运行方法:

```
# 基本用法 (默认3秒延迟)  
python data_generation/run_complete_evaluation.py 30  
  
# 自定义延迟 (推荐3-5秒, 避免API速率限制)  
python data_generation/run_complete_evaluation.py 30 3.0  
  
# 参数说明:  
# - 30: 生成题目数量  
# - 3.0: 每次生成之间的延迟 (秒)  
  
# 说明:  
# - 生成阶段: 从900+道AIME真题 (1983-2025) 中随机选择参考样例  
# - 评估阶段: 与AIME 2025年真题进行质量对比  
# - 数据集来源: math-ai/aime25 (JSONL格式)
```

输出示例:

```
=====  
🚀 AIME数据生成与评估完整流程  
=====  
  
配置信息:  
- 生成题目数量: 30  
- API延迟: 3.0秒/题  
- 生成参考数据: TianHongZXY/aime-1983-2025 (900+道题)  
- 评估参考: AIME 2025真题  
=====  
📝 步骤1: 生成AIME题目  
=====  
📚 加载AIME真题数据集: TianHongZXY/aime-1983-2025  
✓ 已加载 963 道参考题目  
  
🌐 开始生成AIME题目  
目标数量: 30  
生成模型: gpt-4o  
延迟设置: 3.0秒/题  
  
生成AIME题目: 100%|██████████| 30/30 [01:30<00:00, 3.00s/题, 主题=Algebra, 答案=123, 耗时=3.0s]  
✅ 步骤1完成! 生成数据保存在: data_generation/generated_data/aime_generated_20250110_120000.json  
  
🌐 步骤2.1: LLM Judge评估 (vs AIME 2025)  
✅ LLM Judge评估完成!  
平均总分: 4.2/5.0  
通过率: 85.0%  
  
🏆 步骤2.2: Win Rate评估 (vs AIME 2025)  
✅ Win Rate评估完成!  
Win Rate: 45.0%  
=====  
📊 步骤3: 生成综合报告  
=====  
✅ 综合报告已保存: data_generation/evaluation_results/20250110_120000/comprehensive_report.md  
=====  
🎉 完整评估流程完成!  
=====  
📁 输出文件:  
- 生成数据: data_generation/generated_data/aime_generated_20250110_120000.json  
- 评估结果目录: data_generation/evaluation_results/20250110_120000  
- LLM Judge报告: data_generation/evaluation_results/20250110_120000/llm_judge/llm_judge_report_20250110_120000.md  
- Win Rate报告: data_generation/evaluation_results/20250110_120000/win_rate/win_rate_report_20250110_120000.md  
- 综合报告: data_generation/evaluation_results/20250110_120000/comprehensive_report.md  
  
💡 下一步:  
1. 查看综合报告: data_generation/evaluation_results/20250110_120000/comprehensive_report.md  
2. 运行人工验证: python data_generation/human_verification_ui.py data_generation/generated_data/aime_generated_20250110_120000.json
```

12.4.8 综合评估报告

系统自动生成综合评估报告，汇总所有评估结果。以下是示例报告：

AIME数据生成与评估综合报告

1. 基本信息

- 生成时间: 2025-01-10 12:00:00
- 生成题目数量: 30
- 参考AIME年份: 2025

2. 数据生成统计

主题分布

| 主题 | 数量 | 占比 |
|----|----|-------|
| 代数 | 10 | 33.3% |
| 几何 | 8 | 26.7% |
| 数论 | 7 | 23.3% |
| 组合 | 3 | 10.0% |
| 概率 | 2 | 6.7% |

3. LLM Judge评估结果

总体评分

- 平均总分: 4.2/5.0
- 通过率: 85.0% (≥ 3.5 分)
- 优秀率: 40.0% (≥ 4.5 分)

各维度评分

| 维度 | 平均分 | 评级 |
|------|---------|----------|
| 正确性 | 4.3/5.0 | 良好 ★★★★★ |
| 清晰度 | 4.1/5.0 | 良好 ★★★★★ |
| 难度匹配 | 4.0/5.0 | 良好 ★★★★★ |
| 完整性 | 4.4/5.0 | 良好 ★★★★★ |

4. Win Rate评估结果

胜率统计

| 指标 | 数值 | 百分比 |
|----------|----|-------|
| 生成数据胜出 | 9次 | 45.0% |
| AIME真题胜出 | 8次 | 40.0% |
| 平局 | 3次 | 15.0% |

Win Rate: 45.0%

良好: 生成数据质量接近参考数据 (差距<10%)。

5. 综合结论

基于LLM Judge和Win Rate两种评估方法的结果:

1. LLM Judge评估: 生成数据的平均质量为 4.2/5.0
2. Win Rate评估: 生成数据相对于AIME 2025真题的胜率为 45.0%

结论: 生成数据质量优秀, 达到或超过AIME真题水平。可以用于实际应用。

6. 改进建议

- 继续保持当前的生成策略
- 可以考虑增加生成数量
- 建议进行人工验证以确保质量

7. 下一步行动

1. 人工验证: 运行 `python data_generation/human_verification_ui.py <data_path>` 进行人工验证
2. 查看详细结果:
 - LLM Judge详细报告
 - Win Rate详细报告
3. 数据使用: 如果质量满意, 可以将生成的数据用于训练或测试

基于实际使用经验, 总结以下内容:

在数据生成方面, 应该使用合适的延迟时间 (2-3 秒) 避免 API 速率限制, 启用检查点保存以避免中断损失, 先小批量测试 (10 个) 确认无问题后再大批量生成, 并定期检查生成质量及时调整提示词。在评估策略上, 建议结合 LLM Judge 和 Win Rate 两种方法, 其中 LLM Judge 用于绝对质量评估, Win Rate 用于相对质量对比, 人工验证用于最终质量把关。质量标准方面, 建议 LLM Judge 平均分达到 4.0/5.0 以上, Win Rate 达到 45%以上 (接近 50%), 通过率达到 80%以上, 人工验证通过率达到 90%以上。在迭代优化过程中, 应根据评估结果调整生成提示词, 分析低分题目的共同问题, 参考高分题目的优点, 持续改进生成策略。

通过本节的学习，我们掌握了如何使用 HelloAgents 框架进行数据生成质量评估，包括 LLM Judge 评估、Win Rate 评估和人工验证三种方法。这套完整的评估体系可以确保生成数据的高质量，为 AI 系统的训练和测试提供可靠的数据支持。

对于 LLM Judge 和 Win Rate 评估，HelloAgents 也进行了工具集成，并提供了完整的示例代码。如果你对这两种评估方法的具体实现细节感兴趣，同样可以参考示例代码。

12.5 本章小结

在本章中，我们为 HelloAgents 框架构建了一个完整的性能评估系统。让我们回顾一下学到的核心内容：

(1) 评估体系概览

我们建立了一个三层评估体系，全面覆盖智能体的不同能力维度。首先是工具调用能力评估（BFCL），专注于评估智能体的函数调用准确性，包含 simple、multiple、parallel、irrelevance 四个类别，使用 AST 匹配技术进行精确评估。其次是通用能力评估（GAIA），评估智能体的综合问题解决能力，包含三个难度级别共 466 个真实世界问题，关注多步推理、工具使用、文件处理等能力。第三是数据生成质量评估（AIME），评估 LLM 生成数据的质量，使用 LLM Judge 和 Win Rate 两种方法，支持人工验证和综合报告生成，确保生成数据达到参考数据的质量标准。

(2) 核心技术要点

在技术实现上，我们采用了六个核心技术要点。首先是模块化设计，评估系统采用三层架构：数据层（Dataset 负责数据加载和管理）、评估层（Evaluator 负责执行评估流程）和指标层（Metrics 负责计算各种评估指标）。其次是工具化封装，所有评估功能都封装成 Tool，可以被智能体直接调用、集成到工作流中或通过统一接口使用。第三是 AST 匹配技术，使用抽象语法树匹配函数调用，比简单字符串匹配更智能，能够忽略参数顺序、识别等价表达式和忽略格式差异。第四是多模态支持，GAIA 评估支持文本问题、附件文件和图片输入等多模态数据。第五是 LLM Judge 评估，使用 LLM 作为评委评估生成数据质量，提供多维度评分（正确性、清晰度、难度匹配、完整性）、自动化评估流程、详细评估报告，并支持自定义评估维度和标准。第六是 Win Rate 对比评估，通过成对对比评估生成质量（生成数据 vs 参考数据），由 LLM 判断哪个更好并计算胜率统计，接近 50% 表示质量相当。

(3) 扩展方向

基于本章的评估系统，你可以在四个方向上进行扩展。首先是添加新的评估基准，可以参考 BFCL 和 GAIA 的实现模式，实现 Dataset、Evaluator、Metrics 三个组件，并封装成 Tool 供使用。其次是自定义评估指标，在 Metrics 类中添加新的指标计算方法，根据具体应用场景设计指标。第三是集成到 CI/CD 流程，在代码提交时自动运行评估，设置性能阈值防止性能退化，生成评估报告并归档。第四是扩展数据生成评估，支持更多数据类型（代码、对话、文档等），添加更多评估维度（创新性、多样性等），集成更多参考数据集，支持多模型对比评估。

恭喜你完成了第十二章的学习！

评估是智能体开发的重要环节，它让我们能够：

- 客观衡量智能体的能力
- 发现和修复问题
- 持续改进系统

在下一章中，我们将探讨如何将 HelloAgents 框架应用于实际项目中。

继续加油！

习题

提示：部分习题没有标准答案，重点在于培养学习者对智能体性能评估的综合理解和实践能力。

1. 本章介绍了多个智能体评估基准。请分析：

- 在 12.1.2 节中介绍了 BFCL、GAIA、AgentBench 等评估基准。请对比 BFCL 和 GAIA：它们分别评估智能体的哪些核心能力？为什么 BFCL 使用 AST 匹配算法，而 GAIA 使用准精确匹配（Quasi Exact Match）？这两种评估方法各有什么优缺点？
- 假设你要构建一个“智能客服系统”，需要评估以下能力：（1）理解用户意图的准确性；（2）调用后台 API 的正确性；（3）回答的友好性和专业性；（4）处理异常情况的鲁棒性。请为每个能力选择或设计合适的评估指标和方法。
- 在 12.1.1 节中提到，智能体评估面临“输出不确定性”、“评估标准多样性”、“评估成本高昂”三大挑战。请针对每个挑战提出具体的解决方案，并分析方案的可行性和局限性。

2. BFCL（Berkeley Function Calling Leaderboard）是评估工具调用能力的重要基准。基于 12.2 节的内容，请深入思考：

提示：这是一道动手实践题，建议实际操作

- 在 12.2.3 节的 AST 匹配算法中，我们通过比较抽象语法树来判断函数调用是否正确。请分析：为什么 AST 匹配比简单的字符串匹配更合适？在什么情况下 AST 匹配可能会产生误判（假阳性或假阴性）？如何改进 AST 匹配算法来提高准确性？
- BFCL 数据集包含 simple、multiple、parallel、irrelevance 四个类别。请为每个类别设计 2-3 个新的测试样本，要求能够测试智能体在该类别下的边界情况或容易出错的场景。
- 请基于 12.2.4 节的代码，扩展 BFCL 评估器，添加以下功能：（1）支持评估工具调用的执行顺序（对于有依赖关系的多个工具调用）；（2）评估工具调用的效率（如是否使用了最少的调用次数）；（3）生成详细的错误分析报告（如哪些类型的错误最常见）。

3. GAIA（General AI Assistants）评估智能体的综合能力。基于 12.3 节的内容，请完成以下扩展实践：

提示：这是一道动手实践题，建议实际操作

- 在 12.3.2 节中介绍了 GAIA 的三个难度级别 (Level 1/2/3)。请分析：这三个级别在任务复杂度、所需能力、评估标准等方面有什么差异？如果要设计 Level 4（超高难度），应该包含什么类型的任务？
- GAIA 使用“准精确匹配”算法来评估答案的正确性。请分析：这种方法如何处理答案的多样性（如“42”、“四十二”、“42.0”都应该被认为是正确的）？在什么情况下准精确匹配可能不够用？请设计一个更智能的答案匹配算法，能够处理语义等价的答案。
- 请基于 12.3.4 节的代码，实现一个“自定义 GAIA 评估集”：选择一个特定领域（如医疗、法律、金融），设计 10 个真实世界问题，并实现完整的评估流程。要求问题涵盖不同难度级别，并提供标准答案和评分标准。

4. LLM Judge 是使用大语言模型进行评估的新兴方法。基于 12.4 节的内容，请深入分析：

- 在 12.4.2 节中，我们使用 GPT-4 作为评判者来评估智能体的回答质量。请分析：LLM Judge 相比传统的规则匹配或指标计算有什么优势？它存在哪些潜在的偏见或局限性（如对某些回答风格的偏好、对长度的敏感性）？
- LLM Judge 的评分标准设计至关重要。请为以下三个不同的评估场景设计详细的评分标准（包括评分维度、权重、示例）：(1) 代码生成质量评估；(2) 创意写作质量评估；(3) 技术文档质量评估。
- 在 12.4.3 节中提到，可以使用多个 LLM Judge 进行“评审团”式评估。请设计一个“多评委评估系统”：使用 3-5 个不同的 LLM（如 GPT-4、Claude、Qwen）作为评委，如何聚合它们的评分？如何处理评委之间的分歧？如何检测和过滤异常评分？

5. 智能体评估的实践应用需要考虑多个方面。请思考：

- 在实际项目中，评估往往需要在“评估成本”和“评估质量”之间权衡。请设计一个“分层评估策略”：(1) 快速评估（低成本，用于日常开发迭代）；(2) 标准评估（中等成本，用于版本发布前）；(3) 全面评估（高成本，用于重大更新或对外发布）。每层应该包含哪些评估项目？如何设计评估流程？
- 智能体的性能可能随时间变化（如依赖的外部 API 变化、用户需求变化）。请设计一个“持续评估系统”：能够定期自动运行评估，监控智能体性能的变化趋势，并在性能下降时及时告警。这个系统应该包含哪些组件？如何设计告警规则？
- 评估结果需要以清晰的方式呈现给不同的受众（如开发者、产品经理、用户）。请设计一个“评估报告生成系统”：能够根据受众类型自动生成不同详细程度的报告。开发者报告应该包含哪些技术细节？产品经理报告应该突出哪些业务指标？用户报告应该如何简化和可视化？

参考文献

- [1] Patil, S. G., Zhang, T., Wang, X., & Gonzalez, J. E. (2023). Gorilla: Large Language Model Connected with Massive APIs. arXiv preprint arXiv:2305.15334.
- [2] Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., ... & Sun, M. (2023). ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv preprint arXiv:2307.16789.
- [3] Li, M., Zhao, Y., Yu, B., Song, F., Li, H., Yu, H., ... & Li, Y. (2023). Api-bank: A comprehensive benchmark for tool-augmented llms. arXiv preprint arXiv:2304.08244.
- [4] Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., ... & Scialom, T. (2023). GAIA: a benchmark for General AI Assistants. arXiv preprint arXiv:2311.12983.
- [5] Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., ... & Zhang, D. (2023). AgentBench: Evaluating LLMs as Agents. arXiv preprint arXiv:2308.03688.
- [6] Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., ... & Neubig, G. (2023). WebArena: A Realistic Web Environment for Building Autonomous Agents. arXiv preprint arXiv:2307.13854.
- [7] Chan, C. M., Chen, W., Su, Y., Yu, J., Xue, W., Zhang, S., ... & Liu, Z. (2023). ChatEval: Towards Better LLM-based Evaluators through Multi-Agent Debate. arXiv preprint arXiv:2308.07201.
- [8] Zhou, X., Zhu, H., Mathur, L., Zhang, R., Yu, H., Qi, Z., ... & Neubig, G. (2023). SOTPIA: Interactive Evaluation for Social Intelligence in Language Agents. arXiv preprint arXiv:2310.11667.
- [9] Mathematical Association of America. (2024). American Invitational Mathematics Examination (AIME). Retrieved from <https://www.maa.org/math-competitions/invitational-competitions/aime>

第十三章 智能旅行助手

在前面的章节中，我们从零开始构建了 HelloAgents 框架，实现了多种智能体范式、工具系统、记忆机制、协议通信和性能评估等核心功能。从本章开始，我们将进入一个全新的阶段：将所学知识融会贯通，构建完整的实用应用。

还记得在第一章中，我们构建的第一个智能体吗？那是一个简单的智能旅行助手，展示了 Thought–Action–Observation 循环的基本原理。本章的智能旅行助手将是一个完整的项目，包含以下核心功能：

- (1) 智能行程规划：用户输入目的地、日期、偏好等信息，系统自动生成包含景点、餐饮、酒店的完整行程计划。
- (2) 地图可视化：在地图上标注景点位置、绘制游览路线，让行程一目了然。
- (3) 预算计算：自动计算门票、酒店、餐饮、交通费用，显示预算明细。
- (4) 行程编辑：支持添加、删除、调整景点，实时更新地图。
- (5) 导出功能：支持导出为 PDF 或图片，方便保存和分享。

13.1 项目概述与架构设计

13.1.1 为什么需要智能旅行助手

规划一次旅行是一件既令人兴奋又令人头疼的事情。你需要在网上搜索景点信息，对比不同的攻略，查看天气预报，预订酒店，计算预算，规划路线。这个过程可能需要花费几个小时甚至几天的时间。而且即使花了这么多时间，你也不确定规划的行程是否合理，是否遗漏了什么重要的景点，预算是否准确。

传统的旅行规划方式有几个痛点。首先是信息分散。景点信息在旅游网站上，天气信息在天气网站上，酒店信息在预订网站上，你需要在多个网站之间切换，手动整合这些信息。其次是缺少个性化。大部分攻略都是通用的，不考虑你的个人偏好、预算限制、出行时间等因素。最后是难以调整。当你想修改行程时，可能需要重新规划整个行程，因为景点的顺序、时间安排、预算都是相互关联的。

AI 技术为解决这些问题提供了新的可能。想象一下，你只需要告诉系统“我想去北京玩 3 天，喜欢历史文化，预算中等”，系统就能自动为你生成一个完整的行程计划，包括每天去哪些景点、在哪里吃饭、住哪个酒店、需要多少预算。而且这个计划是可以调整的，你可以删除不喜欢的景点，调整游览顺序，系统会自动更新地图和预算。

这就是我们要构建的智能旅行助手。它不仅仅是一个技术演示，而是一个真正有用的应用。通过这个项目，你会学到如何将 AI 技术应用到实际问题中，如何设计多智能体系统，如何构建完整的 Web 应用。

13.1.2 技术架构概览

系统采用经典的前后端分离架构，分为四个层次，如图 13.1 所示：



图 13.1 智能旅行助手技术架构

- (1) 前端层 (**Vue3+TypeScript**)：负责用户交互和数据展示，包括表单输入、结果展示、地图可视化。
- (2) 后端层 (**FastAPI**)：负责 API 路由、数据验证、业务逻辑。
- (3) 智能体层 (**HelloAgents**)：负责任务分解、工具调用、结果整合。包含 4 个专门的 Agent。
- (4) 外部服务层：提供数据和服务，包括高德地图 API、Unsplash API、LLM API。

数据流转过程如下：用户在前端填写表单 → 后端验证数据 → 调用智能体系统 → 智能体依次调用景点搜索、天气查询、酒店推荐、行程规划 Agent → 每个 Agent 通过 MCP 协议调用外部 API → 整合结果返回前端 → 前端渲染展示。

项目的结构参考如下，提供便于定位源码：

```
helloagents-trip-planner/
  -- backend/          # 后端代码
    -- app/
      -- agents/       # 智能体实现
      -- api/          # API 路由
      -- models/        # 数据模型
      -- services/      # 服务层
      -- config.py      # 配置文件
      requirements.txt  # Python 依赖
  -- frontend/         # 前端代码
    -- src/
      -- views/         # 页面组件
      -- services/      # API 服务
      -- types/         # 类型定义
```

```

  └─ router/
    └─ package.json      # 路由配置
                                # npm依赖

```

详细的架构设计和数据流转将在后续章节中介绍。

13.1.3 快速体验：5分钟运行项目

在深入学习实现细节之前，让我们先把项目跑起来，看看最终的效果。这样你会对整个系统有一个直观的认识。

环境要求：

- Python 3.10 或更高版本
- Node.js 16.0 或更高版本
- npm 8.0 或更高版本

获取 API 密钥：

你需要准备以下 API 密钥：

- LLM 的 API(OpenAI、DeepSeek 等)
- 高德地图 Web 服务 Key：访问 <https://console.amap.com/> 注册并创建应用
- Unsplash Access Key：访问 <https://unsplash.com/developers> 注册并创建应用

将所有 API 密钥放入 .env 文件。

启动后端：

```

# 1. 进入后端目录
cd helloagents-trip-planner/backend

# 2. 安装依赖
pip install -r requirements.txt

# 3. 配置环境变量
cp .env.example .env
# 编辑.env文件，填入你的API密钥

# 4. 启动后端服务
uvicorn app.api.main:app --reload
# 或者
python run.py

```

成功启动后，访问 <http://localhost:8000/docs> 可以看到 API 文档。

打开新的终端窗口：

```

# 1. 进入前端目录
cd helloagents-trip-planner/frontend

# 2. 安装依赖
npm install

# 3. 启动前端服务
npm run dev

```

成功启动后，访问 <http://localhost:5173> 即可使用应用。

体验核心功能：

首先需在首页表单中填写目的地城市、旅行日期、偏好、预算、交通及住宿类型等信息。点击“开始规划”按钮后，系统会显示加载进度条，并很快生成结果页面，如图 13.2 所示。



图 13.2 旅行助手规划进行页面

随后加载成功，该页面会清晰展示行程概览、预算明细、景点地图、每日行程详情和天气信息，如图 13.3、13.4 所示。



图 13.3 旅行助手规划完成页面



图 13.4 旅行助手规划完成页面

如果用户需要个性化调整，可以点击“编辑行程”按钮，自由调整景点顺序或删除某个景点，如图 13.5 所示。规划完成后，通过“导出行程”下拉菜单，即可将最终方案轻松保存为图片或 PDF 文件，方便随时查阅。



图 13.5 旅行助手规划完成页面

13.2 数据模型设计

13.2.1 Web 应用中的数据流转

在构建智能旅行助手时，我们需要解决一个核心问题：如何表示和传递旅行计划数据？

我们需要理解一个完整的 Web 应用中数据是如何流转的。想象一下，当用户在浏览器中点击“开始规划”按钮时，会发生什么？

用户在前端填写的表单数据(目的地、日期、预算等)需要通过 HTTP 请求发送到后端服务器。后端接收到数据后，会调用智能体系统进行处理。智能体又会调用高德地图 API、Unsplash API 等外部服务获取数据。这些外部 API 返回的数据格式各不相同，有的用`lng`，有的用`lon`，有的用`longitude`。最后，后端需要将处理好的数据返回给前端，前端再渲染成用户看到的页面。

在这个过程中，数据经历了多次转换：前端表单 → HTTP 请求 → 后端 Python 对象 → 外部 API 响应 → 后端 Python 对象 → HTTP 响应 → 前端 TypeScript 对象 → 页面展示。如果没有统一的数据格式，每一步转换都可能出错。这就是为什么我们需要数据模型。

13.2.2 从字典到 Pydantic 模型

让我们从第一章的简单原型开始。在那个原型中，我们使用 Python 字典来表示景点数据：

```
# 第一章的做法：使用字典
attraction = {
    "name": "故宫",
    "location": {"lng": 116.397128, "lat": 39.916527},
    "price": 60
}

# 访问数据
lng = attraction["location"]["lng"]
```

这种方式在原型阶段很方便，但在实际项目中会遇到很多问题。首先是字段名不统一的问题。高德地图 API 返回的位置数据是“116.397128, 39.916527”这样的字符串，需要手动分割成经纬度。而 Unsplash API 可能使用`longitude`和`latitude`。如果我们在代码中到处都用字典，就需要在每个地方都处理这些差异。

其次是类型安全的问题。假设我们不小心把`price`写成了字符串“60”，在 Python 中这不会立即报错，但在计算总预算时就会出问题。更糟糕的是，这种错误只能在运行时才能发现，而且错误信息可能很难定位。

最后是维护性的问题。当我们需要给景点添加新字段(比如`rating`评分)时，需要在代码的多个地方修改。如果遗漏了某个地方，就会导致数据不一致。

Pydantic 提供了一个解决方案。它是 Python 的数据验证库，可以让我们用类来定义数据结构，并自动处理验证、转换和序列化。让我们看一个简单的例子：

```
from pydantic import BaseModel, Field

class Location(BaseModel):
    longitude: float = Field(..., description="经度")
    latitude: float = Field(..., description="纬度")

class Attraction(BaseModel):
    name: str
    location: Location
    ticket_price: int = 0

# 创建对象
attraction = Attraction(
    name="故宫",
    location=Location(longitude=116.397128, latitude=39.916527),
    ticket_price=60
)

# 类型安全的访问
lng = attraction.location.longitude # IDE会提供代码补全
```

这样做有几个好处。首先，如果我们传入了错误的类型(比如把`ticket_price`设为字符串)，Pydantic 会立即抛出异常，告诉我们哪里出错了。其次，IDE 可以根据类型定义提供代码补全和类型检查，大大减少了拼写错误。最后，当我们需要修改数据结构时，只需要修改类定义，所有使用这个类的地方都会自动更新。

13.2.3 Pydantic 的核心概念

在深入设计我们的数据模型之前，让我们先了解 Pydantic 的几个核心概念。Pydantic 的基础是`BaseModel`类，所有的数据模型都需要继承这个类。每个字段都可以指定类型，Pydantic 会自动进行类型检查和转换。

字段定义使用`Field`函数，它可以指定默认值、描述、验证规则等。`... 表示这个字段是必填的，如果创建对象时没有提供这个字段，Pydantic 会抛出异常。我们也可以使用Optional来表示可选字段，或者直接提供默认值。`

```
from pydantic import BaseModel, Field
from typing import Optional, List

class Attraction(BaseModel):
    name: str = Field(..., description="景点名称") # 必填
```

```
rating: float = Field(default=0.0,ge=0,le=5) # 默认值, 范围验证
visit_duration: int = Field(default=60,gt=0) # 大于0
description: Optional[str] = None # 可选字段
```

Pydantic 还支持嵌套模型和列表。我们可以在一个模型中使用另一个模型作为字段类型,这样就可以构建复杂的数据结构。比如,一个景点包含位置信息,一个行程包含多个景点。

```
class DayPlan(BaseModel):
    date: str
    attractions: List[Attraction] # 景点列表
    hotel: Optional[Hotel] = None # 可选的酒店信息
```

最强大的功能之一是自定义验证器。有时候外部 API 返回的数据格式不符合我们的要求,我们可以使用`field_validator`装饰器来自定义验证和转换逻辑。比如,高德地图返回的温度是"16°C"这样的字符串,我们需要把它转换成数字:

```
from pydantic import field_validator

class WeatherInfo(BaseModel):
    temperature: int

    @field_validator('temperature', mode='before')
    def parse_temperature(cls,v):
        """解析温度字符串: "16°C" -> 16"""
        if isinstance(v,str):
            v = v.replace('°C','').replace('C','').strip()
            return int(v)
        return v
```

这个验证器会在创建对象之前自动执行,将字符串转换成整数。这样我们就不需要在代码的每个地方都手动处理温度格式了。

13.2.4 自底向上的模型设计

现在让我们开始设计智能旅行助手的数据模型。一个好的设计原则是自底向上:先定义最基础的模型,然后逐步组合成复杂的结构。这样做好处是每个模型都很简单,容易理解和维护。

最基础的模型是位置信息。无论是景点、酒店还是餐厅,都需要位置信息。我们定义一个`Location`类来表示经纬度坐标:

```
class Location(BaseModel):
    """位置信息(经纬度坐标)"""
    longitude: float = Field(...,description="经度",ge=-180,le=180)
    latitude: float = Field(...,description="纬度",ge=-90,le=90)
```

这里我们使用了范围验证(`ge`表示大于等于, `le`表示小于等于),确保经纬度的值在合理范围内。

接下来是景点信息。一个景点包含名称、地址、位置、游览时间、描述、评分、图片和门票价格等信息。注意我们使用了`Location`作为字段类型,这就是嵌套模型:

```
class Attraction(BaseModel):
    """景点信息"""
    name: str = Field(...,description="景点名称")
    address: str = Field(...,description="地址")
    location: Location = Field(...,description="经纬度坐标")
    visit_duration: int = Field(...,description="建议游览时间(分钟)",gt=0)
    description: str = Field(...,description="景点描述")
    category: Optional[str] = Field(default="景点",description="景点类别")
    rating: Optional[float] = Field(default=None,ge=0,le=5,description="评分")
    image_url: Optional[str] = Field(default=None,description="图片URL")
    ticket_price: int = Field(default=0,ge=0,description="门票价格(元)")
```

类似地,我们定义餐饮信息和酒店信息。这些模型的结构都很相似,都包含名称、地址、位置和费用等基本信息:

```
class Meal(BaseModel):
    """餐饮信息"""
    type: str = Field(...,description="餐饮类型: breakfast/lunch/dinner/snack")
    name: str = Field(...,description="餐饮名称")
    address: Optional[str] = Field(default=None,description="地址")
    location: Optional[Location] = Field(default=None,description="经纬度坐标")
    description: Optional[str] = Field(default=None,description="描述")
    estimated_cost: int = Field(default=0,description="预估费用(元)")

class Hotel(BaseModel):
    """酒店信息"""
    name: str = Field(...,description="酒店名称")
    address: str = Field(default="",description="酒店地址")
    location: Optional[Location] = Field(default=None,description="酒店位置")
    price_range: str = Field(default="",description="价格范围")
    rating: str = Field(default="",description="评分")
    distance: str = Field(default="",description="距离景点距离")
    type: str = Field(default="",description="酒店类型")
    estimated_cost: int = Field(default=0,description="预估费用(元/晚)")
```

预算信息是一个特殊的模型,它不包含位置信息,而是包含各项费用的汇总:

```
class Budget(BaseModel):
    """预算信息"""
    total_attractions: int = Field(default=0,description="景点门票总费用")
    total_hotels: int = Field(default=0,description="酒店总费用")
    total_meals: int = Field(default=0,description="餐饮总费用")
    total_transportation: int = Field(default=0,description="交通总费用")
    total: int = Field(default=0,description="总费用")
```

现在我们可以组合这些基础模型，构建单日行程。一个单日行程包含日期、描述、交通方式、住宿安排、酒店、景点列表和餐饮列表：

```
class DayPlan(BaseModel):
    """单日行程"""
    date: str = Field(...,description="日期")
    day_index: int = Field(...,description="第几天(从0开始)")
    description: str = Field(...,description="当日行程描述")
    transportation: str = Field(...,description="交通方式")
    accommodation: str = Field(...,description="住宿安排")
    hotel: Optional[Hotel] = Field(default=None,description="酒店信息")
    attractions: List[Attraction] = Field(default_factory=list,description="景点列表")
    meals: List[Meal] = Field(default_factory=list,description="餐饮安排")
```

注意这里使用了List[Attraction]来表示景点列表，default_factory=list表示默认值是一个空列表。

天气信息需要特殊处理，因为高德地图返回的温度格式不规范。我们使用自定义验证器来处理：

```
class WeatherInfo(BaseModel):
    """天气信息"""
    date: str = Field(...,description="日期")
    day_weather: str = Field(...,description="白天天气")
    night_weather: str = Field(...,description="夜间天气")
    day_temp: int = Field(...,description="白天温度(摄氏度)")
    night_temp: int = Field(...,description="夜间温度(摄氏度)")
    wind_direction: str = Field(...,description="风向")
    wind_power: str = Field(...,description="风力")

    @field_validator('day_temp','night_temp',mode='before')
    def parse_temperature(cls,v):
        """解析温度字符串: "16°C" -> 16"""
        if isinstance(v,str):
            v = v.replace('°C','').replace('°','').replace(' ','').strip()
            try:
                return int(v)
            except ValueError:
                return 0 # 容错处理
        return v
```

最后，我们定义完整的旅行计划。这是最顶层的模型，包含了所有的信息：

```
class TripPlan(BaseModel):
    """旅行计划"""
    city: str = Field(...,description="目的地城市")
    start_date: str = Field(...,description="开始日期")
    end_date: str = Field(...,description="结束日期")
    days: List[DayPlan] = Field(default_factory=list,description="每日行程")
    weather_info: List[WeatherInfo] = Field(default_factory=list,description="天气信息")
    overallSuggestions: str = Field(...,description="总体建议")
    budget: Optional[Budget] = Field(default=None,description="预算信息")
```

这样，我们就完成了整个数据模型的设计。从最基础的Location，到Attraction、Meal、Hotel，再到DayPlan，最后到TripPlan，形成了一个清晰的层次结构。

13.2.5 数据模型在 Web 应用中的应用

现在让我们看看这些数据模型如何在实际的 Web 应用中使用。在 FastAPI 中，Pydantic 模型可以直接用作请求和响应的类型定义。FastAPI 会自动进行数据验证、序列化和文档生成。

```
from fastapi import FastAPI
from app.models.schemas import TripPlanRequest,TripPlan

app = FastAPI()

@app.post("/api/trip/plan",response_model=TripPlan)
async def create_trip_plan(request: TripPlanRequest) -> TripPlan:
    """
    创建旅行计划

    FastAPI自动:
    1. 验证请求数据(TripPlanRequest)
    2. 验证响应数据(TripPlan)
    3. 生成OpenAPI文档
    """
    trip_plan = await generate_trip_plan(request)
    return trip_plan
```

当用户发送 POST 请求到 /api/trip/plan 时，FastAPI 会自动将 JSON 数据转换成 TripPlanRequest 对象。如果数据格式不正确（比如缺少必填字段，或者类型不匹配），FastAPI 会自动返回 400 错误，并告诉用户哪里出错了。

在前端，我们也需要定义对应的 TypeScript 类型。虽然 TypeScript 和 Python 是不同的语言，但数据结构是一样的：

```
interface Location {
    longitude: number;
    latitude: number;
}

interface Attraction {
    name: string;
    address: string;
    location: Location;
    visit_duration: number;
    ticket_price: number;
}

interface TripPlan {
    city: string;
    start_date: string;
    end_date: string;
    days: DayPlan[];
}
```

这样，前后端就使用了统一的数据格式。当后端返回 TripPlan 对象时，前端可以直接使用，不需要任何转换。TypeScript 的类型检查也能帮助我们避免很多错误。

13.3 多智能体协作设计

13.3.1 为何需要多智能体

在第七章中，我们学习了如何使用 SimpleAgent 来构建智能体。SimpleAgent 的设计理念是简单直接：每次调用 run() 方法时，Agent 会分析用户的问题，决定是否需要调用工具，然后返回结果。这种设计在处理简单任务时非常有效，但当面对旅行规划这样的任务时，就会遇到一些问题。

如果用单个 Agent 来完成旅行规划。这个 Agent 需要做什么呢？首先，它要搜索景点信息，这需要调用高德地图的 POI 搜索工具。然后，它要查询天气信息，这需要调用天气查询工具。接着，它要搜索酒店信息，这又需要调用 POI 搜索工具。最后，它要把所有这些信息整合起来，生成一个完整的旅行计划。

这听起来很简单，但实际操作时会遇到第一个问题：工具调用的限制。SimpleAgent 每次 run() 调用只能执行一个工具。这意味着我们需要多次调用 run() 方法，每次调用处理一个任务。但这样做会带来一个新问题：如何在多次调用之间传递信息？第一次调用得到的景点信息，如何传递给第二次调用？我们需要手动管理这些中间结果，代码会变得很复杂。

当然，我们可以使用 ReactAgent 来解决这个问题。ReactAgent 可以在一次调用中执行多个工具，它会自动进行多轮思考和行动。但这又带来了新的问题：时间成本。ReactAgent 的每一轮思考都需要调用 LLM，如果需要调用三个工具，就需要至少三轮思考，这意味着至少三次 LLM 调用。而且这些调用是串行的，必须等前一个完成才能开始下一个，总时间会很长。

第二个问题是提示词的复杂度。如果我们要让一个 Agent 完成所有任务，就需要在提示词中详细描述每个任务的执行逻辑。比如：

```
COMPLEX_PROMPT = """你是旅行规划助手。你需要：
1. 使用maps_text_search搜索景点，关键词根据用户偏好确定
2. 使用maps_weather查询天气，获取未来几天的天气预报
3. 使用maps_text_search搜索酒店，类型根据用户需求确定
4. 整合所有信息生成旅行计划，包括每天的景点、餐饮、住宿安排
注意：必须按顺序执行，每个工具只能调用一次，输出必须是JSON格式...
"""

这样
```

这样的提示词有几个问题。首先是难以维护。如果我们想修改景点搜索的逻辑（比如增加评分筛选），就需要修改整个提示词，很容易影响到其他部分。其次是容易出错。LLM 需要同时理解多个任务的要求，很容易搞混不同任务的格式和参数。最后是难以调试。当生成的计划不符合预期时，我们很难知道是哪个环节出了问题，是景点搜索不准确，还是天气查询失败，还是整合逻辑有问题？

面对这些问题，一个自然的想法是：能不能把复杂的任务分解成多个简单的任务，让不同的 Agent 各司其职？这就是多 Agent 协作的核心思想。

想象一下现实世界中的旅行社。当你去旅行社咨询旅行计划时，不会只有一个人为你服务。通常会有专门的景点顾问，负责推荐景点；有酒店顾问，负责预订酒店；还有行程规划师，负责把所有信息整合成完整的行程。每个人都专注于自己擅长的领域，最后由行程规划师把所有信息汇总。这种分工协作的方式，比让一个人做所有事情要高效得多。

13.3.2 Agent 角色设计

基于任务分解原则，我们设计了四个专门的 Agent，如图 13.6 所示：



图 13.6 多智能体协作流程

- **AttractionSearchAgent**（景点搜索专家）专注于搜索景点信息。它只需要理解用户的偏好（比如“历史文化”、“自然风光”），然后调用高德地图的 POI 搜索工具，返回相关的景点列表。它的提示词很简单，只需要说明如何根据偏好选择关键词，如何调用工具。

- **WeatherQueryAgent(天气查询专家)**专注于查询天气信息。它只需要知道城市名称，然后调用天气查询工具，返回未来几天的天气预报。它的任务非常明确，几乎不会出错。
- **HotelAgent(酒店推荐专家)**专注于搜索酒店信息。它需要理解用户的住宿需求(比如“经济型”、“豪华型”)，然后调用 POI 搜索工具，返回符合要求的酒店列表。
- **PlannerAgent(行程规划专家)**负责整合所有信息。它接收前三个 Agent 的输出，加上用户的原始需求(日期、预算等)，然后生成完整的旅行计划。它不需要调用任何外部工具，只需要专注于信息的整合和行程的安排。

现在让我们详细设计每个 Agent 的角色和提示词。设计提示词时，我们需要考虑几个关键问题：这个 Agent 需要什么输入？它应该产生什么输出？它需要调用什么工具？它可能遇到什么问题？

AttractionSearchAgent的任务是根据用户偏好搜索景点。它的输入是城市名称和用户偏好(比如“历史文化”、“自然风光”)。它需要调用 `amap_maps_text_search`工具，参数是关键词和城市。它的输出是景点列表，包含名称、地址、评分等信息。

```
ATTRACTION_AGENT_PROMPT = """你是景点搜索专家。
```

```
**工具调用格式:**  
`[TOOL_CALL:amap_maps_text_search:keywords=景点,city=城市名]`  
  
**示例:**  
- `[TOOL_CALL:amap_maps_text_search:keywords=景点,city=北京]`  
- `[TOOL_CALL:amap_maps_text_search:keywords=博物馆,city=上海]`  
  
**重要:**  
- 必须使用工具搜索，不要编造信息  
- 根据用户偏好({preferences})搜索{city}的景点  
....
```

这个提示词很简洁，但包含了所有必要的信息。它明确说明了工具调用的格式，提供了具体的示例，还强调了两个重要原则：必须使用工具(不能编造)，要根据用户偏好搜索。

WeatherQueryAgent的任务更简单，只需要查询天气。它的输入是城市名称，输出是天气信息。

```
WEATHER_AGENT_PROMPT = """你是天气查询专家。
```

```
**工具调用格式:**  
`[TOOL_CALL:amap_maps_weather:city=城市名]`  
  
请查询{city}的天气信息。  
....
```

HotelAgent的任务是搜索酒店。它的输入是城市名称和住宿类型，输出是酒店列表。

```
HOTEL_AGENT_PROMPT = """你是酒店推荐专家。
```

```
**工具调用格式:**  
`[TOOL_CALL:amap_maps_text_search:keywords=酒店,city=城市名]`  
  
请搜索{city}的{accommodation}酒店。  
....
```

PlannerAgent是最复杂的，因为它需要整合所有信息。它的输入是用户需求和前三个 Agent 的输出，输出是完整的旅行计划(JSON 格式)。

```
PLANNER_AGENT_PROMPT = """你是行程规划专家。
```

```
**输出格式:**  
严格按照以下JSON格式返回：  
{  
    "city": "城市名称",  
    "start_date": "YYYY-MM-DD",  
    "end_date": "YYYY-MM-DD",  
    "days": [...],  
    "weather_info": [...],  
    "overall_suggestions": "总体建议",  
    "budget": {...}  
}
```

```
**规划要求:**  
1. weather_info必须包含每天的天气  
2. 温度为纯数字(不带°C)  
3. 每天安排2-3个景点  
4. 考虑景点距离和游览时间  
5. 包含早中晚三餐  
6. 提供实用建议  
7. 包含预算信息  
....
```

13.3.3 Agent 协作流程

现在让我们看看这四个 Agent 如何协作完成旅行规划任务。整个流程可以分为五个步骤：

```

class TripplannerAgent:
    def __init__(self):
        self.attraction_agent = SimpleAgent(name="景点搜索", prompt=ATTRACTION_PROMPT)
        self.weather_agent = SimpleAgent(name="天气查询", prompt=WEATHER_PROMPT)
        self.hotel_agent = SimpleAgent(name="酒店推荐", prompt=HOTEL_PROMPT)
        self.planner_agent = SimpleAgent(name="行程规划", prompt=PLANNER_PROMPT)

    def plan_trip(self, request: TripPlanRequest) -> TripPlan:
        # 步骤1: 景点搜索
        attraction_response = self.attraction_agent.run(
            f"请搜索{request.city}的{request.preferences}景点"
        )

        # 步骤2: 天气查询
        weather_response = self.weather_agent.run(
            f"请查询{request.city}的天气"
        )

        # 步骤3: 酒店推荐
        hotel_response = self.hotel_agent.run(
            f"请搜索{request.city}的{request.accommodation}酒店"
        )

        # 步骤4: 整合生成计划
        planner_query = self._build_planner_query(
            request, attraction_response, weather_response, hotel_response
        )
        planner_response = self.planner_agent.run(planner_query)

        # 步骤5: 解析JSON
        trip_plan = self._parse_trip_plan(planner_response)
        return trip_plan

```

这个流程顺序执行四个步骤，每个步骤的输出作为下一个步骤的输入。注意我们使用了TripPlanRequest和TripPlan这两个 Pydantic 模型，这是在 13.2 节中定义的。

13.3.4 查询构建

PlannerAgent 需要整合所有信息，这个查询需要包含所有必要的信息，而且要组织得清晰有序，让 LLM 能够准确理解。

```

def _build_planner_query(
    self,
    request: TripPlanRequest,
    attraction_response: str,
    weather_response: str,
    hotel_response: str
) -> str:
    """构建规划Agent的查询"""
    return f"""
请根据以下信息生成{request.city}的{request.days}日旅行计划:

```

```

**用户需求:**
- 目的地: {request.city}
- 日期: {request.start_date} 至 {request.end_date}
- 天数: {request.days}天
- 偏好: {request.preferences}
- 预算: {request.budget}
- 交通方式: {request.transportation}
- 住宿类型: {request.accommodation}

**景点信息:**
{attraction_response}

**天气信息:**
{weather_response}

**酒店信息:**
{hotel_response}

```

请生成详细的旅行计划，包括每天的景点安排、餐饮推荐、住宿信息和预算明细。
.....

通过这种多 Agent 协作的设计，我们把一个复杂的旅行规划任务分解成了四个简单的子任务。每个 Agent 都专注于自己擅长的领域，也为未来功能扩展（比如添加餐厅推荐 Agent、交通规划 Agent）打下了良好的基础。

13.4 MCP 工具集成详解

13.4.1 为什么不直接调用 API

在 13.3 节中，我们设计了四个 Agent 来协作完成旅行规划任务。其中 AttractionSearchAgent、WeatherQueryAgent 和 HotelAgent 都需要调用高德地图的 API 来获取数据。一个自然的问题是：为什么不直接在 Agent 中调用高德地图的 HTTP API？

让我们先看看直接调用 API 会是什么样子。高德地图提供了 POI 搜索 API，我们需要构造 HTTP 请求，传递参数，解析响应：

```

import requests

def search_poi(keywords: str, city: str, api_key: str):
    """直接调用高德地图POI搜索API"""
    url = "https://restapi.amap.com/v3/place/text"
    params = {
        "keywords": keywords,
        "city": city,
        "key": api_key,
        "output": "json"
    }
    response = requests.get(url, params=params)
    data = response.json()
    return data

```

这种方式看起来很简单，但在实际使用中会遇到几个问题。首先是Agent无法自主调用。在我们的HelloAgents框架中，Agent通过识别提示词中的工具调用标记(比如[T0OL_CALL:tool_name:arg1=value1])来调用工具。如果我们直接在代码中调用API，Agent就失去了自主决策的能力，变成了一个简单的函数调用。

其次是参数传递复杂。高德地图的API有很多参数，比如POI搜索有keywords、city、types、offset、page等十几个参数。如果我们要让Agent能够灵活使用这些参数，就需要在提示词中详细说明每个参数的含义和格式，这会让提示词变得非常复杂。

第三是响应解析困难。高德地图API返回的是JSON格式的数据，结构比较复杂。我们需要编写代码来解析这些数据，提取我们需要的字段。如果API的响应格式发生变化，我们就需要修改解析代码。

最后是工具管理混乱。高德地图提供了十几个不同的API(POI搜索、天气查询、路线规划等)，如果我们为每个API都编写一个函数，然后手动注册到Agent的工具列表中，代码会变得很冗长。而且当我们想添加新的API时，需要修改多个地方。

13.4.2 高德地图 MCP 集成

MCP(Model Context Protocol)是Anthropic提出的标准协议，用于连接LLM和外部工具。本节将介绍如何在项目中集成高德地图MCP服务器。我们的项目用的是amap-mcp-server，这是一个用Node.js实现的MCP服务器：



图 13.7 amap-mcp-server 工具

高德地图MCP服务器提供了多种工具，主要分为以下类别，如表13.1所示：

表 13.1 高德地图 MCP 工具分类



通过MCP协议，我们可以很方便地在HelloAgents中集成：

```

from hello_agents.tools import MCPTool
from app.config import get_settings

settings = get_settings()

# 创建MCP工具
mcp_tool = MCPTool(
    name="amap_mcp",
    command="npx",
    args=["-y", "@sugarforever/amap-mcp-server"],
    env={"AMAP_API_KEY": settings.amap_api_key},
    auto_expand=True
)

```

这段代码做了什么呢？首先，command和args指定了如何启动MCP服务器。npx -y @sugarforever/amap-mcp-server会从npm仓库下载并运行amap-mcp-server这个包。env参数传递了环境变量，这里我们传递了高德地图的API密钥。

注意：本文档中部分示例使用npx启动MCP(Model Context Protocol)服务。而在本节代码仓中，我们实际采用的是uvx方式。需要说明的是，npx和uvx在设计理念上高度一致，区别仅在于所处的生态系统，npx面向JavaScript/Node.js(包来自npm)，而uvx面向Python(包来自PyPI)。两种方式并无优劣之分，请大家在使用时按需进行选择。

当我们创建MCPTool对象时，它会在后台启动MCP服务器进程，并通过标准输入输出(stdin/stdout)与服务器通信。这是MCP协议的一个特点：使用进程间通信而不是HTTP，这样更高效，也更容易管理。

最关键的是auto_expand=True这个参数。当设置为True时，MCPTool会自动查询MCP服务器提供了哪些工具，然后为每个工具创建一个独立的Tool对象。这就是为什么我们只创建了一个MCPTool，但Agent却获得了16个工具。让我们看看这个过程：

```

# 创建一个MCPTool
mcp_tool = MCPTool(..., auto_expand=True)
agent.add_tool(mcp_tool)

# Agent实际上获得了16个工具！
print(list(agent.tools.keys()))
# ['amap_maps_text_search', 'amap_maps_weather', ...]

```

如图13.8所示，假设用户想搜索北京的景点，AttractionSearchAgent接收到查询“请搜索北京的历史文化景点”。Agent分析这个查询，决定调用amap_maps_text_search工具，参数是keywords=景点，city=北京。

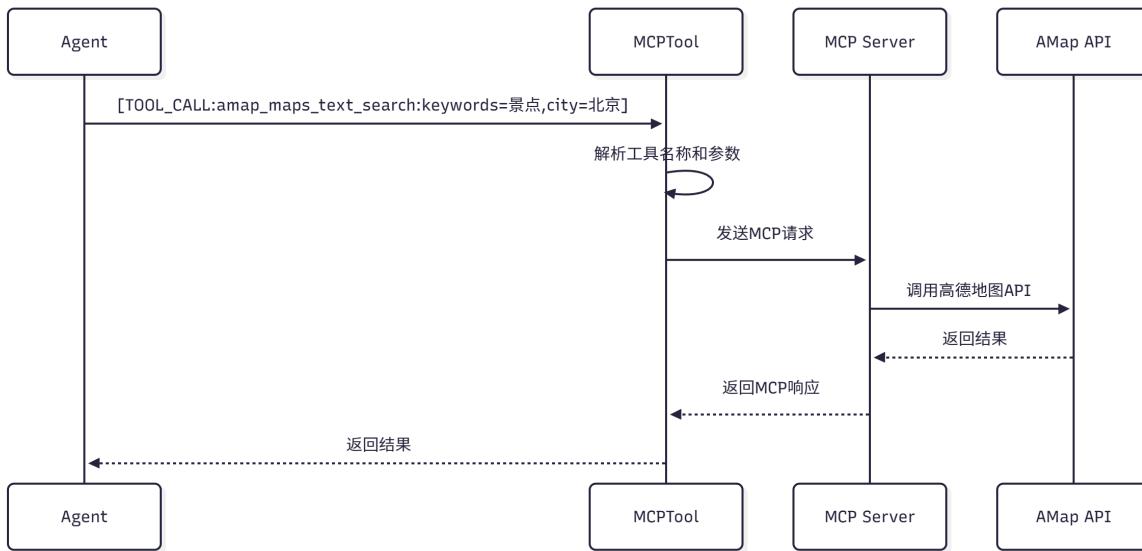


图 13.8 MCP 工具调用流程

Agent 生成工具调用标记: [TOOL_CALL:amap_maps_text_search:keywords=景点, city=北京]。HelloAgents 框架解析这个标记, 提取工具名称和参数, 然后调用对应的 Tool 对象。

Tool 对象是MCPTool自动创建的, 它会把调用请求发送给 MCP 服务器。具体来说, 它会构造一个 JSON-RPC 格式的消息, 通过 stdin 发送给服务器进程:

```
{
  "jsonrpc": "2.0",
  "method": "tools/call",
  "params": {
    "name": "amap_maps_text_search",
    "arguments": {
      "keywords": "景点",
      "city": "北京"
    }
  }
}
```

MCP 服务器接收到这个消息, 解析参数, 然后调用高德地图的 HTTP API。它会构造 HTTP 请求, 添加 API 密钥, 发送请求, 接收响应。

高德地图 API 返回 JSON 格式的数据, 包含景点列表、地址、坐标等信息。MCP 服务器解析这些数据, 提取关键字段, 然后构造响应消息, 通过 stdout 返回给MCPTool:

```
{
  "jsonrpc": "2.0",
  "result": {
    "content": [
      {
        "type": "text",
        "text": "找到以下景点:\n1. 故宫博物院 - 地址: 东城区景山前街4号\n2. 天坛公园 - 地址: 东城区天坛路\n..."
      }
    ]
  }
}
```

MCPTool接收到响应, 提取文本内容, 返回给 Agent。Agent 把这个结果作为工具调用的输出, 继续生成最终的回复。

这个流程看起来很复杂, 但对于 Agent 来说, 它只需要知道有一个叫amap_maps_text_search的工具, 可以搜索景点。所有的底层细节都被 MCP 协议和 MCPTool封装起来了。

13.4.3 共享 MCP 实例

在我们的多 Agent 系统中, 有三个 Agent 都需要使用高德地图的工具。那么每个 Agent 应该创建自己的MCPTool实例, 还是共享同一个实例?

如果每个 Agent 都创建一个MCPTool实例, 这意味着会有三个服务器进程同时运行。每个进程都会独立地调用高德地图 API, 这可能会超过 API 的速率限制。而且多个进程会占用更多的内存和 CPU 资源。

更好的做法是让所有 Agent 共享同一个MCPTool实例。这样只需要启动一个 MCP 服务器进程, 所有的 API 调用都通过这个进程进行。这不仅节省资源, 还可以更好地控制 API 调用频率。

在代码中, 我们在TripPlannerAgent的构造函数中创建一个MCPTool实例, 然后把它添加到每个子 Agent 的工具列表中:

```
class TripPlannerAgent:
    def __init__(self):
        settings = get_settings()
        self.llm = HelloAgentsLLM()
```

```

# 创建共享的MCP工具实例(只创建一次)
self.mcp_tool = MCPTool(
    name="amap_mcp",
    command="npx",
    args=["-y", "@sugarforever/amap-mcp-server"],
    env={"AMAP_API_KEY": settings.amap_api_key},
    auto_expand=True
)

# 创建多个Agent,共享同一个MCP工具
self.attraction_agent = SimpleAgent(
    name="AttractionSearchAgent",
    llm=self.llm,
    system_prompt=ATTRACTION_AGENT_PROMPT
)
self.attraction_agent.add_tool(self.mcp_tool) # 共享

self.weather_agent = SimpleAgent(
    name="WeatherQueryAgent",
    llm=self.llm,
    system_prompt=WEATHER_AGENT_PROMPT
)
self.weather_agent.add_tool(self.mcp_tool) # 共享

self.hotel_agent = SimpleAgent(
    name="HotelAgent",
    llm=self.llm,
    system_prompt=HOTEL_AGENT_PROMPT
)
self.hotel_agent.add_tool(self.mcp_tool) # 共享

```

这样，三个 Agent 都可以使用高德地图的 16 个工具，但底层只有一个 MCP 服务器进程在运行。当我们调用TripPlannerAgent的plan_trip方法时，三个 Agent 会依次调用工具，所有的请求都通过同一个 MCP 服务器发送到高德地图 API。

13.4.4 Unsplash 图片 API 集成

除了高德地图，我们还需要为景点获取图片，让旅行计划更加生动直观。我们使用 Unsplash API 来搜索景点图片。需要注意的是，Unsplash 是国外的服务，而且是为数不多可以免费使用的图片 API，所以搜索结果可能不够准确。在实际项目中，可以考虑使用必应、百度或高德的 POI 图片 API，但这些服务通常需要付费。

Unsplash API 的集成比较简单，我们创建一个UnsplashService类来封装 API 调用：

```

# backend/app/services/unsplash_service.py
import requests
from typing import Optional, List, Dict
import logging

logger = logging.getLogger(__name__)

class UnsplashService:
    """Unsplash图片服务"""

    def __init__(self, access_key: str):
        self.access_key = access_key
        self.base_url = "https://api.unsplash.com"

    def search_photos(self, query: str, per_page: int = 10) -> List[Dict]:
        """搜索图片"""
        try:
            url = f"{self.base_url}/search/photos"
            params = {
                "query": query,
                "per_page": per_page,
                "client_id": self.access_key
            }

            response = requests.get(url, params=params, timeout=10)
            response.raise_for_status()

            data = response.json()
            results = data.get("results", [])

            # 提取图片URL
            photos = []
            for result in results:
                photos.append({
                    "url": result["urls"]["regular"],
                    "description": result.get("description", ""),
                    "photographer": result["user"]["name"]
                })

            return photos
        except Exception as e:
            logger.error(f"搜索图片失败: {e}")
            return []

    def get_photo_url(self, query: str) -> Optional[str]:
        """获取单张图片URL"""

```

```
photos = self.search_photos(query, per_page=1)
return photos[0].get("url") if photos else None
```

这个服务类提供了两个方法：search_photos搜索多张图片，get_photo_url获取单张图片的 URL。我们在 API 路由中使用这个服务，为每个景点获取图片：

```
# backend/app/api/routes/trip.py
from app.services.unsplash_service import UnsplashService
unsplash_service = UnsplashService(settings.unsplash_access_key)

@router.post("/plan", response_model=TripPlan)
async def create_trip_plan(request: TripPlanRequest) -> TripPlan:
    # 生成旅行计划
    trip_plan = trip_planner_agent.plan_trip(request)

    # 为每个景点获取图片
    for day in trip_plan.days:
        for attraction in day.attractions:
            if not attraction.image_url:
                image_url = unsplash_service.get_photo_url(
                    f"{attraction.name} {trip_plan.city}"
                )
                attraction.image_url = image_url

    return trip_plan
```

注意我们没有把 Unsplash 封装成 Tool 或 MCP 工具，而是直接在 API 路由中调用。这是因为图片搜索不需要 Agent 的智能决策，只是一个简单的数据增强步骤。如果你想让 Agent 能够自主决定是否需要图片，或者选择不同的图片来源，可以考虑把它封装成 Tool。

13.5 前端开发详解

13.5.1 前后端分离的 Web 架构

在开始前端开发之前，我们需要理解现代 Web 应用的架构模式。在早期的 Web 开发中，前端和后端是混在一起的，比如 PHP、JSP 这样的技术，HTML 模板和业务逻辑代码写在同一个文件里。这种方式在小项目中很方便，但在大型项目中会遇到很多问题：前端和后端开发者需要频繁协调，代码难以复用，测试困难。

现代 Web 应用普遍采用前后端分离的架构。后端只负责提供 API 接口，返回 JSON 格式的数据。前端是一个独立的应用，通过 HTTP 请求调用后端 API，获取数据后渲染页面。这种架构有几个明显的优势：前端和后端可以独立开发、独立部署、独立测试；前端可以是 Web 应用、移动应用或桌面应用，都使用同一套后端 API；前端可以使用现代的框架和工具链，提供更好的用户体验。

在我们的智能旅行助手项目中，后端是用 Python 和 FastAPI 实现的，提供了一个核心 API 接口 POST /api/trip/plan，接收旅行需求，返回旅行计划。前端是用 Vue 3 和 TypeScript 实现的，是一个单页应用(SPA)，用户在浏览器中填写表单，点击“开始规划”按钮，前端发送 HTTP 请求到后端，等待响应，然后渲染结果页面。整个过程中，页面不会刷新，用户体验很流畅。

前端技术栈的选择需要考虑几个因素：开发效率、性能、生态系统、学习曲线。如表 13.2 所示，该项目选择了以下技术栈：

表 13.2 前端技术栈

| 技术 | 用途 |
|-------------------------|--------------|
| Vue 3 | 前端框架(组合式API) |
| TypeScript | 类型安全 |
| Vite | 构建工具 |
| Ant Design Vue | UI组件库 |
| Axios | HTTP客户端 |
| Vue Router | 路由管理 |
| @amap/amap-jsapi-loader | 高德地图加载器 |
| html2canvas | 截图导出 |
| jspdf | PDF导出 |

项目的目录结构是这样的：

```

frontend/
├── src/
│   ├── views/          # 页面组件
│   │   ├── Home.vue    # 首页(表单)
│   │   └── Result.vue  # 结果页
│   ├── services/       # API服务
│   │   └── api.ts
│   ├── types/          # 类型定义
│   │   └── index.ts
│   ├── router/         # 路由配置
│   │   └── index.ts
│   ├── App.vue
│   └── main.ts
├── package.json
└── vite.config.ts
tsconfig.json

```

其中views目录存放页面组件，services目录存放API调用逻辑，types目录存放TypeScript类型定义，router目录存放路由配置。

13.5.2 类型定义

在13.2节中，我们在后端使用Pydantic定义了数据模型，比如Location、Attraction、DayPlan、TripPlan等。在前端，我们需要定义对应的TypeScript类型。

让我们看看如何定义这些类型。首先是最基础的Location类型，表示经纬度坐标：

```
// frontend/src/types/index.ts
export interface Location {
  longitude: number
  latitude: number
}
```

这个类型定义和后端的Pydantic模型完全对应。注意TypeScript使用interface关键字定义类型，字段类型用冒号分隔，不需要默认值。

接下来是Attraction类型，表示景点信息：

```
export interface Attraction {
  name: string
  address: string
  location: Location
  visit_duration: number
  description: string
  category?: string
  rating?: number
  image_url?: string
  ticket_price?: number
}
```

注意这里使用了Location类型作为字段类型，这就是嵌套类型。问号?表示可选字段，对应后端Pydantic模型中的Optional。

类似地，我们定义Meal、Hotel、Budget、WeatherInfo等类型。最后是顶层的TripPlan类型：

```
export interface TripPlan {
  city: string
  start_date: string
  end_date: string
  days: DayPlan[]
  weather_info: WeatherInfo[]
  overallSuggestions: string
  budget?: Budget
}
```

还有请求类型TripPlanRequest，对应后端的请求模型：

```
export interface TripPlanRequest {
  city: string
  start_date: string
  end_date: string
  days: number
  preferences: string
  budget: string
  transportation: string
  accommodation: string
}
```

这些类型定义有什么用呢？首先，当我们调用API时，TypeScript会检查我们传递的数据是否符合TripPlanRequest类型。如果我们不小心把days写成了字符串，TypeScript会立即报错。其次，当我们接收API响应时，TypeScript会检查响应数据是否符合TripPlan类型。如果后端返回的数据结构发生变化，前端会立即发现。最后，IDE可以根据类型定义提供代码补全，我们输入tripPlan时，IDE会自动列出所有可用的字段。

13.5.3 API服务封装

有了类型定义，我们就可以封装API调用了。我们创建一个api.ts文件，使用Axios来发送HTTP请求：

```
import axios from 'axios'
import type { TripPlanRequest, TripPlan } from '../types'

const api = axios.create({
```

```
baseURL: 'http://localhost:8000/api',
timeout: 120000, // 2分钟超时
headers: {
  'Content-Type': 'application/json'
}
})
```

这里我们创建了一个 Axios 实例，配置了基础 URL、超时时间和请求头。为什么超时时间设置为 2 分钟？因为生成旅行计划需要调用多个 Agent，每个 Agent 都要调用 LLM 和外部 API，整个过程可能需要 10-30 秒。如果超时时间太短，请求会被中断。

接下来我们添加拦截器。拦截器可以在请求发送前和响应接收后执行一些通用逻辑，比如日志记录、错误处理、认证等：

```
// 请求拦截器
api.interceptors.request.use(
  config => {
    console.log('发送请求：', config)
    return config
  },
  error => Promise.reject(error)
)

// 响应拦截器
api.interceptors.response.use(
  response => {
    console.log('收到响应：', response)
    return response
  },
  error => {
    console.error('请求失败：', error)
    return Promise.reject(error)
}
)
```

最后我们定义 API 函数，这是前端调用后端的唯一入口：

```
// 生成旅行计划
export const generateTripPlan = async (request: TripPlanRequest): Promise<TripPlan> => {
  const response = await api.post<TripPlan>('/trip/plan', request)
  return response.data
}
```

注意这个函数的类型签名：参数是TripPlanRequest类型，返回值是Promise<TripPlan>类型。这意味着 TypeScript 会检查调用者传递的参数是否符合要求，也会检查返回值的使用是否正确。

13.5.4 Home 表单设计

Home 页面是用户的入口，包含一个表单，让用户填写旅行需求。我们使用 Vue 3 的 Composition API 来组织代码：

```
<script setup lang="ts">
import { ref } from 'vue'
import { useRouter } from 'vue-router'
import { message } from 'ant-design-vue'
import { generateTripPlan } from '@/services/api'
import type { TripPlanRequest } from '@/types'

const router = useRouter()
const loading = ref(false)
const loadingProgress = ref(0)
const loadingStatus = ref('')

const formData = ref<TripPlanRequest>({
  city: '',
  start_date: '',
  end_date: '',
  days: 3,
  preferences: '历史文化',
  budget: '中等',
  transportation: '公共交通',
  accommodation: '经济型酒店'
})
</script>
```

这里我们使用ref来创建响应式变量。formData是表单数据，类型是TripPlanRequest。loading表示是否正在加载，loadingProgress表示加载进度，loadingStatus表示加载状态文本。

表单提交的逻辑是这样的：

```
const handleSubmit = async () => {
  loading.value = true
  loadingProgress.value = 0

  // 模拟进度更新
  const progressInterval = setInterval(() => {
    if (loadingProgress.value < 90) {
      loadingProgress.value += 10
      if (loadingProgress.value <= 30) loadingStatus.value = '🔍 正在搜索景点...'
      else if (loadingProgress.value <= 50) loadingStatus.value = '☀️ 正在查询天气...'
    }
  }, 100)
}
```

```

        else if (loadingProgress.value <= 70) loadingStatus.value = '正在推荐酒店...'
        else loadingStatus.value = '正在生成行程计划...'
    },
    500
}

try {
    const response = await generateTripPlan(formData.value)
    clearInterval(progressInterval)
    loadingProgress.value = 100
    router.push({ name: 'result', state: { tripPlan: response } })
} catch (error) {
    clearInterval(progressInterval)
    message.error('生成计划失败,请重试')
} finally {
    loading.value = false
}
}

```

这段代码做了几件事。首先，设置loading为 true，显示加载状态。然后，启动一个定时器，每 500 毫秒更新一次进度条和状态文本。这是一个模拟的进度，因为我们无法准确知道后端的处理进度。但这样可以让用户知道系统正在工作，而不是卡住了。

接着，调用generateTripPlan函数发送 API 请求。这是一个异步操作，我们使用await等待响应。如果请求成功，清除定时器，设置进度为 100%，然后跳转到结果页面，并把旅行计划数据传递过去。如果请求失败，显示错误消息。最后，无论成功还是失败，都设置loading为 false，隐藏加载状态。

模板部分使用 Ant Design Vue 的组件：

```

<template>
    <div class="home-container">
        <div class="page-header">
            <h1 class="page-title">智能旅行助手</h1>
            <p class="page-subtitle">基于AI的个性化旅行规划</p>
        </div>

        <a-card class="form-card">
            <a-form :model="formData" @finish="handleSubmit">
                <a-form-item label="目的地城市" name="city" :rules="[{ required: true }]">
                    <a-input v-model:value="formData.city" placeholder="如：北京" />
                </a-form-item>

                <!-- 更多表单项... -->

                <a-form-item>
                    <a-button type="primary" html-type="submit" size="large" :loading="loading">
                        开始规划
                    </a-button>
                </a-form-item>

                <!-- 加载进度条 -->
                <a-form-item v-if="loading">
                    <a-progress :percent="loadingProgress" status="active" />
                    <p>{{ loadingStatus }}</p>
                </a-form-item>
            </a-form>
        </a-card>
    </div>
</template>

```

注意v-model:value指令，它实现了双向数据绑定。当用户在输入框中输入内容时，`formData.city`会自动更新。当`formData.city`的值改变时，输入框的内容也会自动更新。

13.5.5 Result 页面展示

Result 页面是整个应用的核心，展示生成的旅行计划。这个页面包含几个部分：行程概览、预算明细、地图可视化、每日行程详情、天气信息。

首先是地图可视化。我们使用高德地图 JS API 在地图上标注景点位置：

```

import AMapLoader from '@amap/amap-jsapi-loader'

const initMap = async () => {
    const AMap = await AMapLoader.load({
        key: 'your_amap_web_key',
        version: '2.0'
    })

    map = new AMap.Map('amap-container', {
        zoom: 12,
        center: [116.397128, 39.916527]
    })

    // 添加景点标记
    tripPlan.value.days.forEach((day) => {
        day.attractions.forEach((attraction, index) => {
            const marker = new AMap.Marker({
                position: [attraction.location.longitude, attraction.location.latitude],
                title: attraction.name,
                label: { content: `${index + 1}`, direction: 'top' }
            })
            map.add(marker)
        })
    })
}

```

```

    })
}

```

这段代码首先加载高德地图 SDK，然后创建地图实例，最后遍历所有景点，为每个景点创建一个标记(Marker)。标记的位置是景点的经纬度坐标，这些坐标是从后端的Attraction对象中获取的。

导出功能使用html2canvas和jsPDF库。html2canvas可以把DOM元素转换成Canvas，然后我们可以把Canvas导出为图片或PDF：

```

import html2canvas from 'html2canvas'
import jsPDF from 'jspdf'

// 导出为图片
const exportAsImage = async () => {
  const element = document.getElementById('trip-plan-content')
  const canvas = await html2canvas(element, { scale: 2 })
  const link = document.createElement('a')
  link.download = `${tripPlan.value.city}旅行计划.png`
  link.href = canvas.toDataURL()
  link.click()
}

// 导出为PDF
const exportAsPDF = async () => {
  const element = document.getElementById('trip-plan-content')
  const canvas = await html2canvas(element, { scale: 2 })
  const imgData = canvas.toDataURL('image/png')
  const pdf = new jsPDF('p', 'mm', 'a4')
  const imgWidth = 210
  const imgHeight = (canvas.height * imgWidth) / canvas.width
  pdf.addImage(imgData, 'PNG', 0, 0, imgWidth, imgHeight)
  pdf.save(`${tripPlan.value.city}旅行计划.pdf`)
}

```

通过这些前端技术，我们实现了一个完整的Web应用。用户可以在浏览器中填写表单，提交请求，等待AI生成旅行计划，然后查看详细的行程安排，在地图上看到景点位置，还可以导出为图片或PDF。整个过程流畅自然，这就是现代Web应用的魅力。

13.6 功能实现详解

本节介绍智能旅行助手的核心功能实现，包括预算计算、加载进度条、行程编辑、导出功能和侧边导航。

13.6.1 预算计算功能

在规划旅行时，预算是一个非常重要的考虑因素。用户需要知道这次旅行大概要花多少钱，钱都花在哪里。我们的智能旅行助手提供了自动预算计算功能，将费用分为四大类：景点门票、酒店住宿、餐饮和交通。

预算计算的逻辑在哪里实现呢？我们选择在后端的PlannerAgent中实现。为什么不在前端计算？因为预算的估算需要基于景点的门票价格、酒店的价格范围、餐饮的标准等信息，这些信息都是PlannerAgent在生成行程时已经获取的。如果在前端计算，就需要重复这些逻辑，而且可能不准确。

在PlannerAgent的提示词中，我们明确要求LLM生成预算信息：

```

PLANNER_AGENT_PROMPT = """
你是行程规划专家。

**输出格式：**
严格按照以下JSON格式返回：
{
  ...
  "budget": {
    "total_attractions": 180,
    "total_hotels": 1200,
    "total_meals": 480,
    "total_transportation": 200,
    "total": 2060
  }
}

**规划要求：**
...
7. 包含预算信息，根据景点门票、酒店价格、餐饮标准和交通方式估算
"""

```

LLM会根据行程中的景点、酒店、餐饮安排，估算每一项的费用。比如，如果行程中包含故宫(门票60元)、天坛(门票15元)、颐和园(门票30元)，那么景点门票总费用就是105元。如果是3天2晚的行程，酒店是经济型(每晚300元)，那么酒店总费用就是600元。

在前端，我们使用Ant Design Vue的Statistic组件来展示预算信息。这个组件专门用于展示统计数据，支持数字动画、前缀后缀、自定义样式等：

```

<a-card v-if="tripPlan.budget" title="💰 预算明细">
  <a-row :gutter="16">
    <a-col :span="6">
      <a-statistic title="景点门票" :value="tripPlan.budget.total_attractions" suffix="元" />
    </a-col>
    <a-col :span="6">
      <a-statistic title="酒店住宿" :value="tripPlan.budget.total_hotels" suffix="元" />
    </a-col>
    <a-col :span="6">
      <a-statistic title="餐饮" :value="tripPlan.budget.total_meals" suffix="元" />
    </a-col>
    <a-col :span="6">
      <a-statistic title="交通" :value="tripPlan.budget.total_transportation" suffix="元" />
    </a-col>
  </a-row>
</a-card>

```

```

<a-statistic title="餐饮费用" :value="tripPlan.budget.total_meals" suffix="元" />
</a-col>
<a-col :span="6">
  <a-statistic title="交通费用" :value="tripPlan.budget.total_transportation" suffix="元" />
</a-col>
</a-row>
<a-divider />
<a-row>
  <a-col :span="24" style="text-align: center;">
    <a-statistic
      title="预估总费用"
      :value="tripPlan.budget.total"
      suffix="元"
      :value-style="{ color: '#cf1322', fontSize: '32px', fontWeight: 'bold' }"
    />
  </a-col>
</a-row>
</a-card>

```

这段代码使用了栅格布局(a-row和a-col)，将四项费用并排显示。每项费用使用一个a-statistic组件，显示标题和数值。最后用一个分隔线(a-divider)隔开，下面显示总费用，使用红色大字体突出显示。

注意v-if="tripPlan.budget"这个条件渲染。因为预算信息是可选的(在 Pydantic 模型中定义为Optional[Budget])，如果 LLM 没有生成预算信息，这个卡片就不会显示。这体现了前端对数据的容错处理。

13.6.2 加载进度条

生成旅行计划是一个耗时的操作。后端需要依次调用 AttractionSearchAgent、WeatherQueryAgent、HotelAgent 和 PlannerAgent，每个 Agent 都要调用 LLM 和外部 API。整个过程可能需要 10-30 秒。如果用户点击“开始规划”按钮后，页面没有任何反馈，用户会以为系统卡住了，可能会刷新页面或重复点击。

为了提升用户体验，我们添加了加载进度条和状态提示。现在只是模拟进度，可以让用户知道系统正在工作。

```

const loading = ref(false)
const loadingProgress = ref(0)
const loadingStatus = ref('')

const handleSubmit = async () => {
  loading.value = true
  loadingProgress.value = 0

  // 模拟进度更新
  const progressInterval = setInterval(() => {
    if (loadingProgress.value < 90) {
      loadingProgress.value += 10
    }
    if (loadingProgress.value <= 30) loadingStatus.value = '🔍 正在搜索景点...'
    else if (loadingProgress.value <= 50) loadingStatus.value = '☀️ 正在查询天气...'
    else if (loadingProgress.value <= 70) loadingStatus.value = '🏨 正在推荐酒店...'
    else loadingStatus.value = '🕒 正在生成行程计划...'
  }, 500)

  try {
    const response = await generateTripPlan(formData.value)
    clearInterval(progressInterval)
    loadingProgress.value = 100
    loadingStatus.value = '✅ 完成!'
    router.push({ name: 'result', state: { tripPlan: response } })
  } catch (error) {
    clearInterval(progressInterval)
    message.error('生成计划失败')
  } finally {
    loading.value = false
  }
}

```

13.6.3 行程编辑功能

AI 生成的旅行计划虽然很智能，但可能不完全符合用户的个人需求。比如，用户可能不喜欢某个景点，想删除它；或者想调整景点的游览顺序。我们提供了行程编辑功能，让用户可以自定义行程。

编辑功能的核心是状态管理。我们需要维护两个状态：当前的行程计划和原始的行程计划。当用户进入编辑模式时，我们保存原始计划的副本。如果用户取消编辑，就恢复原始计划。如果用户保存修改，就更新当前计划：

```

const editMode = ref(false)
const originalPlan = ref<TripPlan | null>(null)

// 进入编辑模式
const toggleEditMode = () => {
  editMode.value = true
  originalPlan.value = JSON.parse(JSON.stringify(tripPlan.value))
}

```

注意这里使用了JSON.parse(JSON.stringify(...))来深拷贝对象。为什么不直接赋值？因为 JavaScript 中对象是引用类型，如果直接赋值，originalPlan 和 tripPlan 会指向同一个对象，修改一个会影响另一个。深拷贝可以创建一个完全独立的副本。

移动景点的逻辑是交换数组中两个元素的位置：

```
// 移动景点
const moveAttraction = (dayIndex: number, attractionIndex: number, direction: 'up' | 'down') => {
  const attractions = tripPlan.value.days[dayIndex].attractions
  const newIndex = direction === 'up' ? attractionIndex - 1 : attractionIndex + 1

  if (newIndex >= 0 && newIndex < attractions.length) {
    [attractions[attractionIndex], attractions[newIndex]] =
    [attractions[newIndex], attractions[attractionIndex]]
  }
}
```

这里使用了 ES6 的解构赋值语法来交换两个元素。`[a,b] = [b,a]` 是一个很优雅的交换方式，不需要临时变量。

删除景点使用数组的 `splice` 方法：

```
// 删除景点
const deleteAttraction = (dayIndex: number, attractionIndex: number) => {
  tripPlan.value.days[dayIndex].attractions.splice(attractionIndex, 1)
}
```

保存修改时，我们需要重新初始化地图，因为景点的位置可能发生了变化：

```
// 保存修改
const saveChanges = () => {
  editMode.value = false
  message.success('修改已保存')
  initMap() // 重新初始化地图
}

// 取消编辑
const cancelEdit = () => {
  if (originalPlan.value) {
    tripPlan.value = originalPlan.value
  }
  editMode.value = false
}
```

在模板中，我们根据 `editMode` 的值显示不同的 UI。编辑模式下，每个景点旁边会显示上移、下移、删除按钮：

```
<div v-if="editMode" class="edit-buttons">
  <a-button size="small" @click="moveAttraction(dayIndex, index, 'up')">上移</a-button>
  <a-button size="small" @click="moveAttraction(dayIndex, index, 'down')">下移</a-button>
  <a-button size="small" danger @click="deleteAttraction(dayIndex, index)">删除</a-button>
</div>
```

13.6.4 导出功能

用户生成了满意的旅行计划后，可能想保存下来或分享给朋友。我们提供了两种导出方式：导出为图片和导出为 PDF。

导出功能的核心是 `html2canvas` 库。这个库可以把 DOM 元素转换成 Canvas，然后我们可以把 Canvas 导出为图片。但这里有一个技术难点：地图是用 Canvas 渲染的，而 `html2canvas` 在处理嵌套 Canvas 时存在兼容性问题。

我们尝试了多种解决方案，包括将地图 Canvas 转换成图片后再导出，但由于高德地图的 Canvas 渲染机制和跨域限制，这个方案并没有完全解决问题。在实际项目中，可能需要考虑以下替代方案：

1. 使用高德地图的静态地图 API：调用 `maps_staticmap` 工具生成静态地图图片，替代动态地图
2. 分开导出：地图和行程内容分开导出，最后在后端合并
3. 使用截图服务：使用 `Puppeteer` 等无头浏览器在服务端截图
4. 简化导出内容：导出时隐藏地图，只导出行程的文字内容

目前的实现中，我们采用了简化方案，在导出时暂时隐藏地图部分，只导出行程的文字内容和景点信息。虽然这不是最理想的方案，但可以保证导出功能的可用性。

导出为图片的逻辑很简单：

```
import html2canvas from 'html2canvas'

const exportAsImage = async () => {
  const element = document.getElementById('trip-plan-content')
  if (!element) return

  const canvas = await html2canvas(element, {
    backgroundColor: '#ffffff',
    scale: 2,
    useCORS: true
  })

  const link = document.createElement('a')
  link.download = `${tripPlan.value.city}旅行计划.png`
  link.href = canvas.toDataURL('image/png')
  link.click()
  message.success('导出成功!')
}
```

scale: 2表示使用 2 倍分辨率，这样导出的图片更清晰。useCORS: true允许跨域加载图片，这对于景点图片(来自 Unsplash)很重要。

导出为 PDF 需要额外的步骤：先转换成 Canvas，再转换成图片，最后添加到 PDF 中：

```
import jsPDF from 'jspdf'

const exportAsPDF = async () => {
  // 先截取地图
  await captureMapImage()

  const element = document.getElementById('trip-plan-content')
  if (!element) return

  const canvas = await html2canvas(element, {
    backgroundColor: '#ffffff',
    scale: 2,
    useCORS: true,
    allowTaint: true
  })

  // 恢复地图
  restoreMap()

  const pdf = new jsPDF('p', 'mm', 'a4')
  const imgData = canvas.toDataURL('image/png')
  const imgWidth = 210 // A4宽度
  const imgHeight = (canvas.height * imgWidth) / canvas.width

  pdf.addImage(imgData, 'PNG', 0, 0, imgWidth, imgHeight)
  pdf.save(` ${tripPlan.value.city}旅行计划.pdf`)
  message.success('导出成功!')
}

这里需要计算图片的高度，保持宽高比。A4 纸的宽度是 210mm，我们根据 Canvas 的宽高比计算出对应的高度。
```

13.6.5 侧边导航与锚点跳转

Result 页面的内容很多，包括行程概览、预算明细、地图、每日行程、天气信息等。如果用户想快速跳转到某个部分，需要滚动很长的距离。我们提供了侧边导航和锚点跳转功能，让用户可以快速定位。

侧边导航使用 Ant Design Vue 的 Menu 组件：

```
<a-menu
  v-model:selectedKeys=" [activeSection]"
  mode="inline"
  @click="scrollToSection"
>
  <a-menu-item key="overview">行程概览</a-menu-item>
  <a-menu-item key="budget">预算明细</a-menu-item>
  <a-menu-item key="map">地图</a-menu-item>
  <a-menu-item key="days">每日行程</a-menu-item>
  <a-menu-item key="weather">天气</a-menu-item>
</a-menu>
```

点击菜单项时，调用scrollToSection函数：

```
const activeSection = ref('overview')

// 滚动到指定区域
const scrollToSection = ({ key }: { key: string }) => {
  activeSection.value = key
  const element = document.getElementById(key)
  if (element) {
    element.scrollIntoView({ behavior: 'smooth', block: 'start' })
  }
}
```

scrollIntoView是浏览器原生的 API，可以让元素滚动到可视区域。behavior: 'smooth'表示平滑滚动，而不是瞬间跳转。block: 'start'表示元素的顶部对齐到可视区域的顶部。

在页面的各个部分，我们需要添加对应的 id：

```
<div id="overview">
  <!-- 行程概览内容 -->
</div>

<div id="budget">
  <!-- 预算明细内容 -->
</div>

<div id="map">
  <!-- 地图内容 -->
</div>
```

这样，当用户点击侧边导航的某个菜单项时，页面会平滑滚动到对应的部分。

通过这些功能的实现，我们的智能旅行助手不仅能够生成旅行计划，还提供了丰富的交互功能：预算计算让用户了解费用，加载进度条让等待不再焦虑，行程编辑让计划更符合个人需求，导出功能让计划可以分享和保存，侧边导航让长页面易于浏览。这些功能的组合，构成了一个完整、易用、实用的 Web 应用。

13.7 结语

恭喜你完成了第十三章的学习！

通过本章，你不仅学会了如何构建一个完整的智能旅行助手应用，更重要的是掌握了：

1. 系统设计思维：如何将复杂问题分解为多个简单任务
2. 工程实践能力：如何将理论知识转化为可运行的代码
3. 全栈开发能力：如何整合前后端技术栈
4. AI 应用开发：如何利用 LLM 构建实用的应用

这个项目是一个起点，而不是终点。你可以基于这个项目：

- 添加更多功能
- 优化用户体验
- 扩展到其他领域(如智能购物助手、智能学习助手等)
- 部署到生产环境，服务真实用户

最好的学习方式是实践。不要只是阅读代码，而是要动手修改、扩展、优化。每一次实践都会让你对多 Agent 系统有更深的理解。

祝你在 AI 应用开发的道路上越走越远！

第十四章 自动化深度研究智能体

在第十三章的旅行助手项目中，我们体验了如何将 HelloAgents 应用于一个多智能体产品。本章我们继续向前，聚焦「知识密集型应用」：构建一个能够自动化执行深度研究任务的智能体助手。

相比旅行规划，深度研究的难点在于信息的不断发散、事实的快速更新以及用户对引用来源的高要求。为了交付可信的研究报告，我们需要让智能体具备三个核心能力：

- (1) 问题剖析：将用户的开放主题拆解为可检索的查询语句。
- (2) 多轮信息采集：结合不同搜索 API 持续挖掘资料，并去重整合。
- (3) 反思与总结：依据阶段结果识别知识空白，决定是否继续检索，并生成结构化总结。

14.1 项目概述与架构设计

14.1.1 为什么需要深度研究助手

在信息爆炸的时代，我们每天都需要快速了解新的技术、概念或事件。传统的研究方式有几个痛点。首先是信息过载。搜索引擎返回成千上万的结果，你需要逐个点开链接，阅读大量内容，才能找到有用的信息。其次是缺少结构。即使找到了相关信息，这些信息往往是碎片化的，缺少系统性的组织。最后是重复劳动。每次研究新主题时，都需要重复“搜索→阅读→总结→整理”的过程。

这就是深度研究助手需要解决的问题。它不仅仅是一个搜索工具，而是一个能够自主规划、执行和总结的研究助手。

深度研究助手的核心价值：

1. 节省时间：将 1-2 小时的研究工作压缩到 5-10 分钟
2. 提高质量：系统化的研究流程，避免遗漏重要信息
3. 可追溯：记录所有搜索结果和来源，方便验证和引用
4. 可扩展：可以轻松添加新的搜索引擎、数据源和分析工具

14.1.2 技术架构概览

此次系统仍然采用经典的前后端分离架构，如图 14.1 所示。

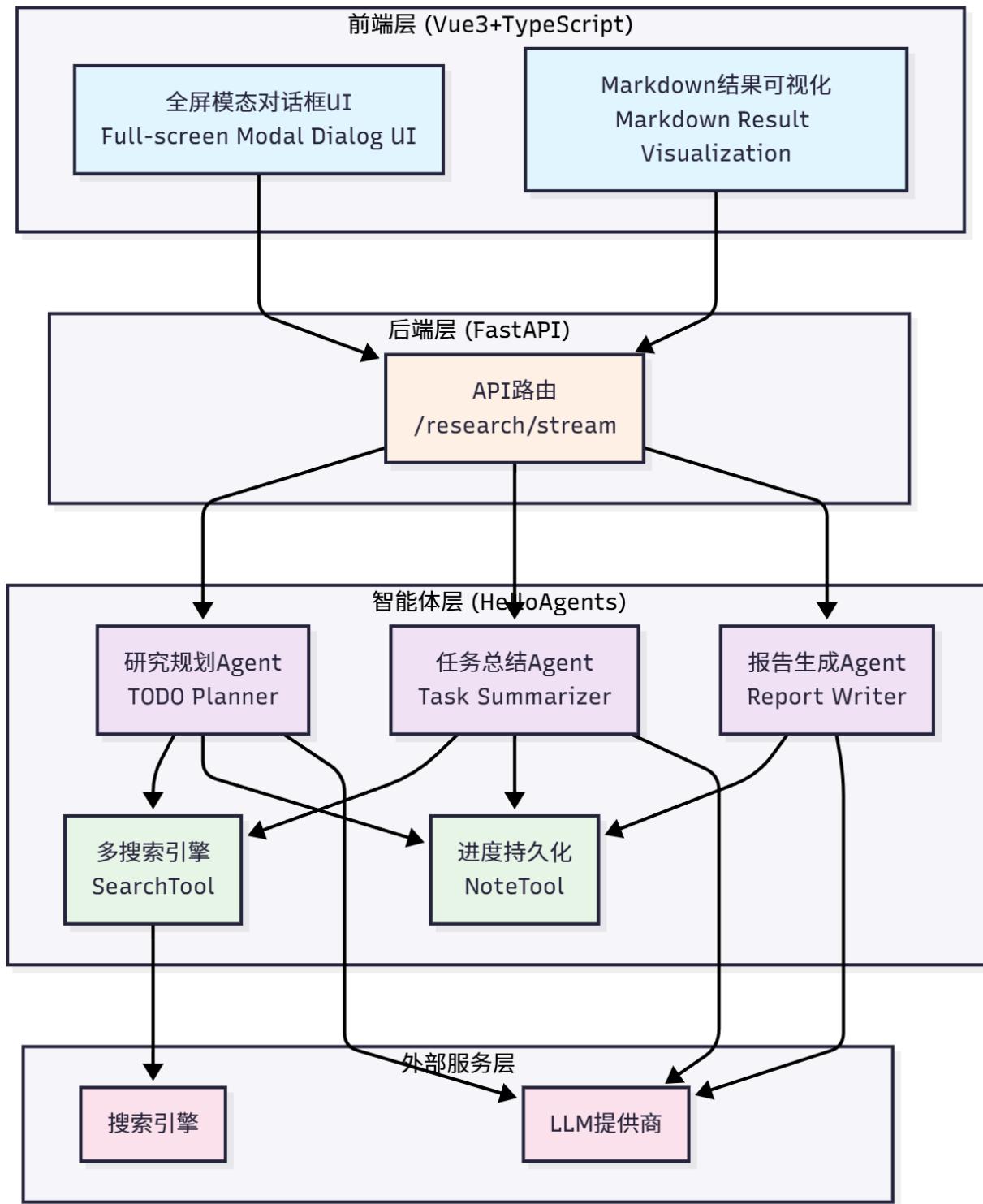


图 14.1 深度研究助手技术架构

系统分为四层架构设计：

前端层 (Vue3+TypeScript): 全屏模态对话框 UI、Markdown 结果可视化

后端层 (FastAPI): API 路由 (/research/stream)

智能体层 (HelloAgents): 三个专门 Agent (TODO Planner、Task Summarizer、Report Writer) + 两个核心工具 (SearchTool、NoteTool)

外部服务层: 搜索引擎+ LLM 提供商

让我们看看一个完整的研究请求是如何在系统中流转的，如图 14.2 所示：

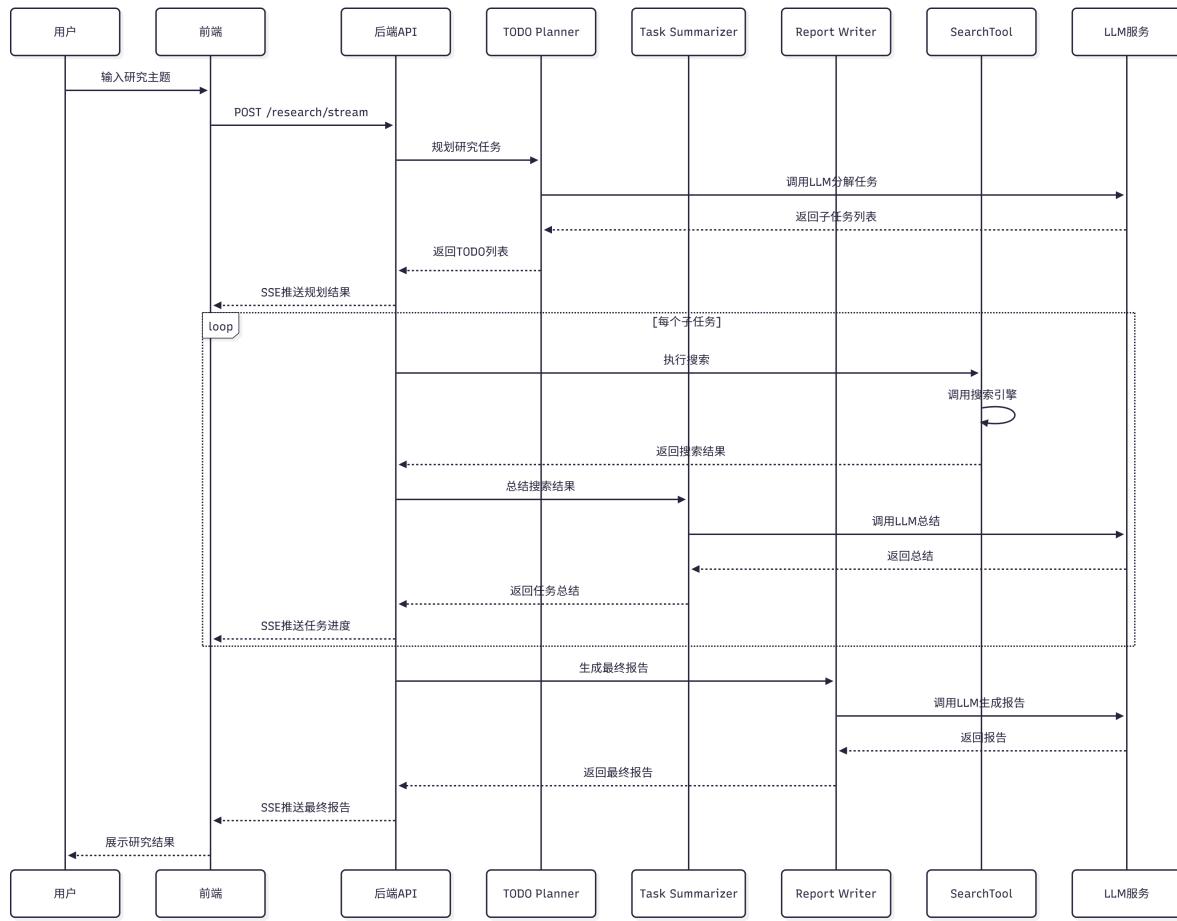


图 14.2 深度研究助手数据流转过程

1. 用户输入：用户在前端输入研究主题
2. 前端发送：前端通过 SSE 连接到/research/stream
3. 后端接收：FastAPI 接收请求，创建研究状态
4. 规划阶段：调用研究规划 Agent，分解为 3 个子任务
5. 执行阶段：逐个执行每个子任务
 - 使用 SearchTool 搜索
 - 调用任务总结 Agent 总结
 - 使用 NoteTool 记录结果
6. 报告阶段：调用报告生成 Agent，整合所有总结
7. 流式返回：通过 SSE 推送进度和结果到前端
8. 前端展示：前端实时更新任务状态、进度条、日志、报告

项目的目录结构如下：

```

helloagents-deepresearch/
  +-- backend/
  |   +-- src/
  |       +-- agent.py          # 核心协调器
  |       +-- main.py           # FastAPI入口
  |       +-- models.py         # 数据模型
  |       +-- prompts.py        # Prompt模板
  |       +-- config.py         # 配置管理
  |       +-- services/
  |           +-- planner.py    # 规划服务
  |           +-- summarizer.py  # 总结服务
  |           +-- reporter.py   # 报告服务
  |           +-- search.py     # 搜索服务
  |       +-- .env
  |       +-- pyproject.toml
  |       +-- workspace/        # 研究笔记
  +-- frontend/
      +-- src/
          +-- App.vue            # 主组件
          +-- components/
              +-- ResearchModal.vue # UI组件
          +-- composables/        # 组合式函数

```

2/9/26, 1:56 PM

```
└── useResearch.ts  
├── package.json      # npm依赖  
└── vite.config.ts    # 构建配置
```

14.1.3 快速体验：5分钟运行项目

在深入学习实现细节之前，让我们先把项目跑起来，看看最终的效果。这样你会对整个系统有一个直观的认识。

你可以通过以下命令检查版本：

```
python --version  # 应该显示 Python 3.10.x 或更高  
node --version   # 应该显示 v16.x.x 或更高  
npm --version    # 应该显示 8.x.x 或更高
```

(1) 启动后端

```
# 1. 进入后端目录  
cd helloagents-deepresearch/backend  
  
# 2. 安装依赖  
# 方式1：使用uv（推荐，更快的Python包管理器）  
uv sync  
  
# 方式2：使用pip  
pip install -e .  
  
# 3. 配置环境变量  
cp .env.example .env  
  
# 4. 编辑.env文件，填入你的API密钥  
# 使用你喜欢的编辑器打开.env文件  
# 至少需要配置：  
# - LLM_PROVIDER (如 openai、deepseek、qwen)  
# - LLM_API_KEY (你的LLM API密钥)  
# - SEARCH_API (如 duckduckgo、tavily)  
  
# 5. 启动后端  
python src/main.py
```

如果一切正常，你会看到类似的输出：

```
INFO:     Started server process [12345]  
INFO:     Waiting for application startup.  
INFO:     Application startup complete.  
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

(2) 启动前端

打开一个新的终端窗口：

```
# 1. 进入前端目录  
cd helloagents-deepresearch/frontend  
  
# 2. 安装依赖  
npm install  
  
# 3. 启动前端  
npm run dev
```

如果一切正常，你会看到类似的输出：

```
VITE v5.0.0  ready in 500 ms  
→ Local:  http://localhost:5174/  
→ Network: use --host to expose  
→ press h + enter to show help
```

(3) 开始研究

打开浏览器访问 <http://localhost:5174>，你会看到一个居中的输入卡片，如图 14.3 所示。输入研究主题，例如 Datawhale 是一个什么样的组织？，选择搜索引擎（如果配置了多个），点击“开始研究”按钮。



图 14.3 深度研究助手搜索页面

如图 14.4 所示，系统会自动展开为全屏，左侧显示研究信息，右侧实时显示研究进度和结果。整个研究过程大约需要 1-3 分钟，取决于主题的复杂度和搜索引擎的响应速度。

图 14.4 深度研究助手展开研究

研究完成后，你会看到：

- 任务列表：显示所有子任务及其状态
- 进度日志：显示研究过程中的所有操作
- 最终报告：结构化的 Markdown 报告，包含所有子任务的总结和来源引用

现在你已经成功运行了深度研究助手，对系统有了直观的认识。

14.2 TODO 驱动的研究范式

14.2.1 什么是 TODO 驱动的研究

传统的搜索引擎只能回答单个问题，而深度研究需要回答一系列相关的问题。TODO 驱动的研究范式将复杂的研究主题分解为多个子任务（TODO），逐个执行并整合结果。

这种范式的核心思想是：将“研究”这个复杂任务转化为“规划→执行→整合”的流程。

让我们通过一个例子来理解这个转变。假设你想研究“Datawhale 是一个什么样的组织？”传统的搜索方式是：

用户输入: Datawhale是一个什么样的组织?

搜索引擎: 返回10~20个链接

用户: 逐个点开链接, 阅读内容, 记录笔记

结果: 碎片化的信息, 缺少系统性

这种方式的问题在于每个链接只涵盖主题的一个方面、缺少系统性结构, 需要手动整理和总结。

TODO 驱动方式: 系统化研究

用户输入: Datawhale是一个什么样的组织?

系统规划:

- └─ TODO 1: Datawhale的基本信息 (组织定位)
- └─ TODO 2: Datawhale的主要项目 (核心内容)
- └─ TODO 3: Datawhale的社区文化 (价值观)
- └─ TODO 4: Datawhale的影响力 (社会贡献)

系统执行:

对每个TODO:

1. 搜索相关资料
2. 总结关键信息
3. 记录来源引用

系统整合:

生成结构化报告:

- └─ 第一部分: 组织定位 (来自TODO 1)
- └─ 第二部分: 核心内容 (来自TODO 2)
- └─ 第三部分: 价值观 (来自TODO 3)
- └─ 第四部分: 社会贡献 (来自TODO 4)
- └─ 参考文献: 所有来源引用

这种方式的优势在于将复杂主题分解为清晰的子问题, 每个子任务的搜索结果和总结都被记录下来, 方便追溯。同时, 系统化的研究流程避免了遗漏重要信息, 可以轻松添加新的子任务或调整执行顺序。

一个完整的 TODO 驱动研究系统包含三个核心要素:

(1) 智能规划器 (TODO Planner) : 负责将研究主题分解为子任务。一个好的规划器需要理解主题的关键方面和研究目标, 将主题分解为 3-5 个子任务 (太少覆盖不全, 太多会冗余), 并为每个子任务设计合适的搜索查询。

(2) 任务执行器 (Task Executor) : 负责执行每个子任务。执行器需要使用搜索引擎获取相关资料, 提取关键信息并去除冗余内容, 同时保存所有来源引用以方便验证。

(3) 报告生成器 (Report Writer) : 负责整合所有子任务的结果。生成器需要按照逻辑顺序组织内容, 合并重复的信息, 并为每个观点添加来源引用。

在我们的案例里, TODO 驱动的研究流程如图 14.5 所示:

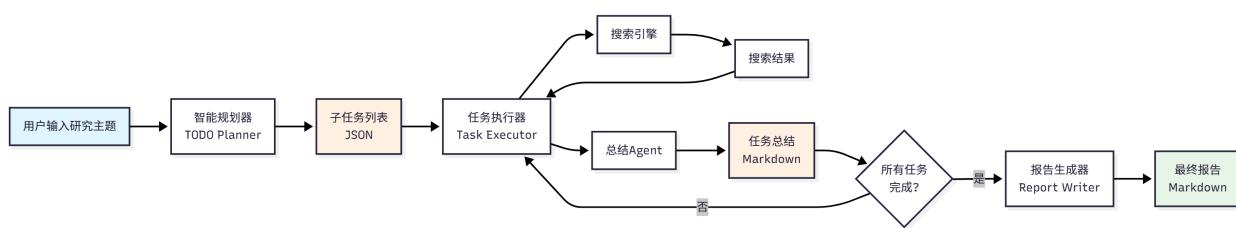


图 14.5 TODO 驱动的研究流程

整个流程是线性的, 但每个阶段都有明确的输入和输出。这种设计使得系统易于理解和调试。

14.2.2 三阶段研究流程

TODO 驱动的研究流程分为三个阶段: 规划 (Planning) 、执行 (Execution) 、报告 (Reporting) 。每个阶段都有专门的 Agent 负责。

(1) 阶段 1: 规划

规划阶段的目标是将研究主题分解为 3-5 个子任务。系统接收研究主题和当前日期作为输入, 输出 JSON 格式的子任务列表。每个子任务包含三个字段: title (任务标题) 、 intent (研究意图) 和 query (搜索查询) 。

研究规划 Agent 会根据主题特点采用不同的分解策略, 通常从基础概念入手, 然后了解技术现状、实际应用和发展趋势, 必要时还会进行对比分析。例如, 对于“Datawhale 是一个什么样的组织?”, 规划 Agent 可能生成以下子任务:

```
[
  {
    "title": "Datawhale的基本信息",
    "intent": "了解Datawhale的组织定位、成立时间、发展历程",
    "query": "Datawhale organization introduction history 2024"
  },
  {
    ...
  }
]
```

```

        "title": "Datawhale的主要项目",
        "intent": "了解Datawhale的核心开源项目和教程",
        "query": "Datawhale projects tutorials open source 2024"
    },
]
.....

```

一个好的规划应该覆盖全面、逻辑清晰、查询精准、条目数量适中。

(2) 阶段 2: 执行

执行阶段逐个执行每个子任务，搜索并总结相关资料。系统接收子任务列表和搜索引擎配置作为输入，输出每个子任务的总结（Markdown 格式）和来源引用列表。执行流程如下：

对于每个子任务，执行器会：

1. 搜索资料：使用配置的搜索引擎执行搜索

```

search_results = search_tool.run({
    "input": task.query,
    "backend": "tavily",
    "mode": "structured",
    "max_results": 5
})

```

2. 获取搜索结果：提取标题、URL、摘要

```

{
  "results": [
    {
      "title": "What is a Multimodal Model?",
      "url": "https://example.com/multimodal-model",
      "snippet": "A multimodal model is an AI model that can process multiple types of data..."
    },
    ...
  ]
}

```

3. 调用总结 Agent：总结搜索结果

```

summary = summarizer_agent.run(
    task=task,
    search_results=search_results
)

```

4. 记录总结和来源：保存到 NoteTool

```

note_tool.run({
    "action": "create",
    "title": task.title,
    "content": f"## {task.title}\n\n{summary}\n\n## 来源\n{sources}",
    "tags": ["research", "summary"]
})

```

任务总结 Agent 会从每个搜索结果中提取核心观点，合并相似信息，保留重要的数字、日期、名称等关键数据，并为每个观点添加来源引用。例如，对于“Datawhale 的基本信息”的搜索结果，总结 Agent 可能生成：

Datawhale的基本信息

Datawhale是一个专注于数据科学与AI领域的开源组织，成立于2018年[1]。组织的核心使命是"for the learner, 和学习者一起成长"，致力于构建一个纯粹的学习社区[2]。

核心定位：

1. **开源教育平台**：提供高质量的AI和数据科学学习资源[1]
2. **学习者社区**：汇聚了数万名AI学习者和实践者[3]
3. **知识共享**：倡导开源精神，所有内容完全免费开放[2]

发展历程：

- **2018年**：Datawhale成立，发布首个开源教程[1]
- **2020年**：成为国内领先的AI学习社区之一[3]
- **2024年**：累计发布50+开源项目，影响10万+学习者[4]

来源

- [1] <https://github.com/datawhalechina>
- [2] <https://datawhale.club/about>
- [3] <https://www.zhihu.com/org/datawhale>
- [4] <https://datawhale.cn>

在执行过程中，系统会实时推送进度信息到前端：

```

{
  "type": "status",
  "message": "正在搜索: Datawhale的基本信息"
}

```

```
{
  "type": "status",
  "message": "正在总结搜索结果..."
}

{
  "type": "task",
  "task": {
    "id": 1,
    "title": "Datawhale的基本信息",
    "status": "completed"
  }
}
```

(3) 阶段 3: 报告

报告阶段的目标是整合所有子任务的总结，生成最终报告。系统接收所有子任务的总结和研究主题作为输入，输出 Markdown 格式的最终报告。报告包含标题、概述、各个子任务的详细分析、总结和参考文献五个部分。例如，对于“Datawhale 是一个什么样的组织？”，最终报告可能是：

```
# Datawhale是一个什么样的组织?

## 概述

本报告系统地研究了Datawhale这个开源组织，涵盖基本信息、主要项目、社区文化和影响力四个方面。

## 1. Datawhale的基本信息

Datawhale是一个专注于数据科学与AI领域的开源组织，成立于2018年...

(此处插入子任务1的总结)

## 2. Datawhale的主要项目

Datawhale发布了多个高质量的开源教程，包括Hello-Agents、Joyful-Pandas等...

(此处插入子任务2的总结)
.....
## 总结

通过本次研究，我们了解了Datawhale的组织定位、核心项目、社区文化和社会贡献。Datawhale是一个纯粹的学习社区，为AI教育做出了重要贡献。

## 参考文献

[1] https://github.com/datawhalechina
[2] https://datawhale.club/about
...
```

报告生成 Agent 会按照子任务的逻辑顺序组织内容，在开头添加简要概述，合并重复的信息，统一 Markdown 格式，并将所有来源引用整理到参考文献部分。

14.3 智能体系统设计

14.3.1 Agent 职责划分

在深度研究助手中，我们设计了三个专门的 Agent，每个 Agent 负责一个特定的任务。这使得每个 Agent 都很简单，易于理解和维护。

在第七章中，我们学习了如何使用SimpleAgent来构建智能体。SimpleAgent的设计理念是简单直接：每次调用run()方法时，Agent 会分析用户的问题，决定是否需要调用工具，然后返回结果。这种设计在处理简单任务时非常有效，但当面对深度研究这样的复杂任务时，就需要我们继续采用多智能体协作的方案进行。

如表 14.1 所示，三个 Agent 分别负责规划、总结和报告生成。

表 14.1 三个 Agent 的职责划分

Agent	职责	输入	输出	工具
TODO Planner	任务规划	研究主题	子任务列表 (JSON)	无 (纯推理)
Task Summarizer	任务总结	搜索结果	总结 (Markdown)	NoteTool (读取/更新笔记)
Report Writer	报告生成	所有总结	最终报告 (Markdown)	NoteTool (读取/创建笔记)

让我们详细介绍每个 Agent 的设计。

Agent 1: 研究规划专家 (TODO Planner)

职责：将研究主题分解为 3-5 个子任务

设计理念：研究规划专家的核心任务是理解用户的研究主题，分析主题的关键方面，然后生成一系列子任务。这个过程类似于人类研究者在开始研究前的“头脑风暴”阶段。

Prompt 设计:

```
todo_planner_instructions = """
```

你是一个研究规划专家。你的任务是将用户的研究主题分解为3-5个子任务。

当前日期: {current_date}

研究主题: {research_topic}

请分析这个研究主题，将其分解为3-5个子任务。每个子任务应该：

1. 涵盖主题的一个重要方面
2. 有明确的研究目标
3. 可以通过搜索引擎找到相关资料

请以JSON格式返回子任务列表，每个子任务包含：

- title: 任务标题（简洁明了）
- intent: 任务意图（为什么要研究这个）
- query: 搜索查询（用于搜索引擎的查询字符串，可以使用英文以获得更好的搜索结果）

示例输出：

```
[  
  {},  
  {"title": "什么是多模态模型",  
   "intent": "了解多模态模型的基础概念，为后续研究打下基础",  
   "query": "multimodal model definition concept 2024"},  
  ...  
]
```

请确保：

1. 子任务数量在3-5个之间
2. 子任务之间有逻辑关系（如从基础到应用，从现状到趋势）
3. 搜索查询能够准确找到相关资料
4. 只返回JSON，不要包含其他文本

"""

关键设计点： 提示词包含当前日期以获取最新信息，明确要求JSON格式输出便于解析，通过示例帮助Agent理解期望输出，并强调子任务数量、逻辑关系等约束。

实现代码：

这里的ToolAwareSimpleAgent是根据SimpleAgent拓展实现，可以在14.3.2了解，这里不用深究。

```
class PlanningService:  
    def __init__(self, llm: HelloAgentsLLM):  
        self._agent = ToolAwareSimpleAgent(  
            name="TODO Planner",  
            system_prompt="你是一个研究规划专家",  
            llm=llm,  
            tool_call_listener=self._on_tool_call  
        )  
  
        def plan_todo_list(self, state: SummaryState) -> List[TodoItem]:  
            prompt = todo_planner_instructions.format(  
                current_date=get_current_date(),  
                research_topic=state.research_topic,  
            )  
  
            response = self._agent.run(prompt)  
            tasks_payload = self._extract_tasks(response)  
  
            todo_items = []  
            for idx, item in enumerate(tasks_payload, start=1):  
                task = TodoItem(  
                    id=idx,  
                    title=item["title"],  
                    intent=item["intent"],  
                    query=item["query"],  
                )  
                todo_items.append(task)  
  
            return todo_items  
  
        def _extract_tasks(self, response: str) -> List[dict]:  
            """从Agent响应中提取JSON"""  
            # 使用正则表达式提取JSON部分  
            json_match = re.search(r'\[.*\]', response, re.DOTALL)  
            if json_match:  
                json_str = json_match.group(0)  
                return json.loads(json_str)  
            else:  
                raise ValueError("无法从响应中提取JSON")
```

Agent 2：任务总结专家 (Task Summarizer)

职责： 总结搜索结果，提取关键信息

设计理念： 任务总结专家的核心任务是阅读搜索结果，提取关键信息，并以结构化的方式呈现。这个过程类似于人类研究者在阅读文献后做笔记的过程。

Prompt 设计:

```
task_summarizer_instructions = """
你是一个任务总结专家。你的任务是总结搜索结果，提取关键信息。
```

```
任务标题: {task_title}
任务意图: {task_intent}
搜索查询: {task_query}
```

```
搜索结果:
{search_results}
```

请仔细阅读以上搜索结果，提取关键信息，并以Markdown格式返回总结。

总结应该包含：

1. **核心观点**：搜索结果中的核心观点和结论
2. **关键数据**：重要的数字、日期、名称等
3. **来源引用**：为每个观点添加来源引用（使用[1]、[2]等标记）

请确保：

1. 总结简洁明了，避免冗余
2. 保留重要的细节和数据
3. 为每个观点添加来源引用
4. 使用Markdown格式（标题、列表、加粗等）

示例输出：

```
## 核心观点
```

多模态模型是一种能够处理多种类型数据的AI模型[1]。与传统的单模态模型不同，多模态模型可以同时理解文本、图像、音频等[2]。

关键特点：

- 跨模态理解[1]
- 统一表示[3]
- 端到端训练[2]

```
## 来源
```

```
[1] https://example.com/source1
[2] https://example.com/source2
[3] https://example.com/source3
....
```

关键设计点：提示词包含任务标题、意图、查询等上下文帮助 Agent 理解任务，明确要求输出包含核心观点、关键数据、来源引用，强调为每个观点添加来源引用，并通过示例帮助 Agent 理解期望的输出格式。

实现代码：

```
class SummarizationService:
    def __init__(self, llm: HelloAgentsLLM):
        self._agent = ToolAwareSimpleAgent(
            name="Task Summarizer",
            system_prompt="你是一个任务总结专家",
            llm=llm,
            tool_call_listener=self._on_tool_call
        )

    def summarize_task(
        self,
        task: TodoItem,
        search_results: List[dict]
    ) -> str:
        # 格式化搜索结果
        formatted_sources = self._format_sources(search_results)

        prompt = task_summarizer_instructions.format(
            task_title=task.title,
            task_intent=task.intent,
            task_query=task.query,
            search_results=formatted_sources,
        )

        summary = self._agent.run(prompt)
        return summary

    def _format_sources(self, search_results: List[dict]) -> str:
        """格式化搜索结果"""
        formatted = []
        for idx, result in enumerate(search_results, start=1):
            formatted.append(
                f"[{idx}] {result['title']}\n"
                f"URL: {result['url']}\n"
                f"摘要: {result['snippet']}\n"
            )
        return "\n".join(formatted)
```

Agent 3：报告撰写专家 (Report Writer)

职责：整合所有子任务的总结，生成最终报告

设计理念：报告撰写专家的核心任务是将所有子任务的总结整合成一份结构化的报告。这个过程类似于人类研究者在完成所有调研后撰写研究报告的过程。

Prompt 设计：

```
report_writer_instructions = """
```

你是一个报告撰写专家。你的任务是整合所有子任务的总结，生成一份结构化的研究报告。

研究主题: {research_topic}

子任务总结:

{task_summaries}

请整合以上所有子任务的总结，生成一份结构化的研究报告。

报告应该包含：

1. **标题**: 研究主题
2. **概述**: 简要介绍研究主题和报告结构 (2-3段)
3. **各个子任务的详细分析**: 按照逻辑顺序组织 (使用二级标题)
4. **总结**: 总结研究的主要发现 (1-2段)
5. **参考文献**: 所有来源引用 (按照子任务分组)

请确保：

1. 报告结构清晰，逻辑连贯
2. 消除重复的信息
3. 保留所有来源引用
4. 使用Markdown格式

示例输出：

```
# 多模态大模型的最新进展
```

```
## 概述
```

本报告系统地研究了多模态大模型的最新进展...

```
## 1. 什么是多模态模型
```

(此处插入子任务1的总结)

```
## 2. 最新的多模态模型有哪些
```

(此处插入子任务2的总结)

...

```
## 总结
```

通过本次研究，我们了解了...

```
## 参考文献
```

任务1: 什么是多模态模型

```
[1] https://example.com/source1
```

....

关键设计点：提示词明确要求报告包含标题、概述、详细分析、总结、参考文献等结构，强调按逻辑顺序组织内容，要求合并重复信息消除冗余，并保留所有来源引用。

实现代码：

```
class ReportingService:
    def __init__(self, llm: HelloAgentsLLM):
        self._agent = ToolAwareSimpleAgent(
            name="Report Writer",
            system_prompt="你是一个报告撰写专家",
            llm=llm,
            tool_call_listener=self._on_tool_call
        )

    def generate_report(
        self,
        research_topic: str,
        task_summaries: List[Tuple[TodoItem, str]]
    ) -> str:
        # 格式化子任务总结
        formatted_summaries = self._format_summaries(task_summaries)

        prompt = report_writer_instructions.format(
            research_topic=research_topic,
            task_summaries=formatted_summaries,
        )

        report = self._agent.run(prompt)
        return report

    def _format_summaries(
        self,
        task_summaries: List[Tuple[TodoItem, str]]
    ) -> str:
```

```
"""格式化子任务总结"""
formatted = []
for idx, (task, summary) in enumerate(task_summaries, start=1):
    formatted.append(
        f"## 任务{idx}: {task.title}\n"
        f"意图: {task.intent}\n\n"
        f"{summary}\n"
    )
return "\n".join(formatted)
```

14.3.2 ToolAwareSimpleAgent 的设计

在第七章中，我们实现了SimpleAgent，它是HelloAgents框架的基础Agent。但在深度研究助手中，我们需要一个能够记录工具调用的Agent。这就是ToolAwareSimpleAgent的由来。

在深度研究助手中，我们需要记录每个Agent的工具调用情况，用于：

1. 调试：查看Agent调用了哪些工具，传入了什么参数
2. 日志：记录研究过程中的所有操作
3. 分析：分析Agent的行为模式
4. 进度展示：实时显示Agent正在做什么

SimpleAgent本身不支持工具调用监听，因此我们需要扩展它。

ToolAwareSimpleAgent在SimpleAgent的基础上增加了一个tool_call_listener参数，这是一个回调函数，每次工具调用时都会被调用。

使用示例：

```
from hello_agents import ToolAwareSimpleAgent

def tool_listener(call_info):
    print(f"Agent: {call_info['agent_name']}")
    print(f"工具: {call_info['tool_name']}")
    print(f"参数: {call_info['parsed_parameters']}")
    print(f"结果: {call_info['result']}")

agent = ToolAwareSimpleAgent(
    name="研究助手",
    system_prompt="你是一个研究助手",
    llm=llm,
    tool_call_listener=tool_listener
)
```

ToolAwareSimpleAgent继承自SimpleAgent，重写了_execute_tool_call方法：

```
class ToolAwareSimpleAgent(SimpleAgent):
    def __init__(self, name: str, system_prompt: str, llm: HelloAgentsLLM, tool_registry: Optional[ToolRegistry] = None, tool_call_listener: Optional[Callable] = None):
        super().__init__(name=name, system_prompt=system_prompt, llm=llm, tool_registry=tool_registry)
        self._tool_call_listener = tool_call_listener

    def _execute_tool_call(self, tool_name: str, parameters: str) -> str:
        """执行工具调用，并通知监听器"""
        # 解析参数
        parsed_parameters = self._parse_parameters(parameters)

        # 调用工具
        result = super()._execute_tool_call(tool_name, parameters)

        # 通知监听器
        if self._tool_call_listener:
            self._tool_call_listener({
                "agent_name": self.name,
                "tool_name": tool_name,
                "parsed_parameters": parsed_parameters,
                "result": result,
            })
        return result
```

在深度研究助手中，我们使用ToolAwareSimpleAgent来记录所有Agent的工具调用：

```
class DeepResearchAgent:
    def __init__(self, config: Configuration):
        self.config = config
        self.llm = HelloAgentsLLM(...)
```

```

# 创建工具调用监听器
def tool_listener(call_info):
    self._emit_event({
        "type": "tool_call",
        "agent": call_info["agent_name"],
        "tool": call_info["tool_name"],
        "parameters": call_info["parsed_parameters"],
    })

# 创建三个Agent, 都使用相同的监听器
self.planner = PlanningService(self.llm, tool_listener)
self.summarizer = SummarizationService(self.llm, tool_listener)
self.reporter = ReportingService(self.llm, tool_listener)

```

这样，所有 Agent 的工具调用都会被记录，并通过 SSE 推送到前端，实时显示给用户。

14.3.3 Agent 协作模式

三个 Agent 之间是顺序协作的关系，如图 14.6 所示。

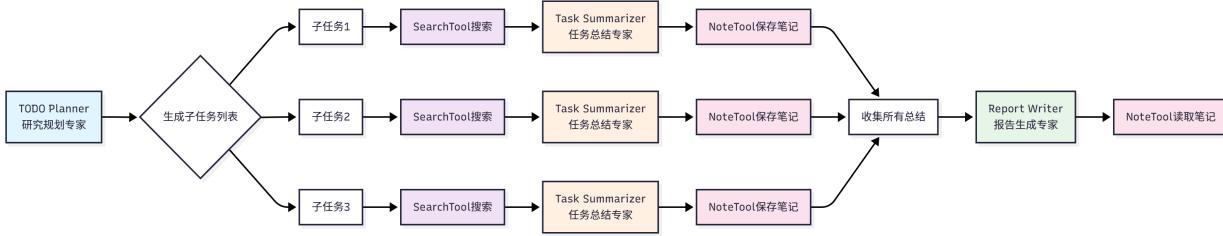


图 14.6 Agent 协作流程

顺序协作模式的特点是：

1. 线性流程：Agent 按照固定的顺序执行
2. 明确的输入输出：每个 Agent 的输入来自上一个 Agent 的输出
3. 无并发：同一时间只有一个 Agent 在工作

DeepResearchAgent 是整个系统的核心协调器，负责调度三个 Agent：

```

class DeepResearchAgent:
    def run(self, research_topic: str) -> str:
        # 1. 规划阶段
        self._emit_event({"type": "status", "message": "正在规划研究任务..."})
        todo_list = self.planner.plan_todo_list(research_topic)
        self._emit_event({"type": "tasks", "tasks": todo_list})

        # 2. 执行阶段
        task_summaries = []
        for task in todo_list:
            self._emit_event({
                "type": "status",
                "message": f"正在研究: {task.title}"
            })

            # 搜索
            search_results = self.search_service.search(task.query)

            # 总结
            summary = self.summarizer.summarize_task(task, search_results)
            task_summaries.append((task, summary))

            self._emit_event({
                "type": "task_completed",
                "task_id": task.id
            })

        # 3. 报告阶段
        self._emit_event({"type": "status", "message": "正在生成报告..."})
        report = self.reporter.generate_report(research_topic, task_summaries)
        self._emit_event({"type": "report", "content": report})

    return report

```

14.4 工具系统集成

14.4.1 SearchTool 扩展

在第七章中，我们实现了 SearchTool 的基础版本，集成了 Tavily 和 SerpApi 两个搜索引擎，展示了多源搜索的设计思想。在本章的深度研究助手中，我们进一步扩展了 SearchTool 的能力，新增了 DuckDuckGo、Perplexity、SearXNG 等搜索引擎，并实现了 Advanced 模式（组合多个搜索引擎）。搜索是深度研究助手最核心的功能，这些扩展使得系统能够适应不同的使用场景和需求。

如表 14.2 所示，这次增加的搜索引擎有不同的特点和适用场景。

表 14.2 多搜索引擎对比

搜索引擎	需要API密钥	返回格式	特点	适用场景
Tavily	是	结构化JSON	专为AI设计，结果质量高	生产环境
DuckDuckGo	否	结构化JSON	免费，无需注册	快速体验
Perplexity	是	AI总结+来源	AI驱动，返回总结	需要总结的场景
SearXNG	否（自建）	结构化JSON	开源，可自建	私有部署
Advanced	混合	结构化JSON	组合多个引擎	需要全面结果

我们不再单独讨论如何扩展，可以参考源码以及第七章的拓展案例实现。SearchTool提供了统一的搜索接口，无论使用哪个搜索引擎，调用方式都是一样的。

在深度研究助手中，我们通过配置文件选择搜索引擎：

```
# config.py
class SearchAPI(str, Enum):
    TAVILY = "tavily"
    DUCKDUCKGO = "duckduckgo"
    PERPLEXITY = "perplexity"
    SEARXNG = "searxng"
    ADVANCED = "advanced"

class Configuration(BaseModel):
    search_api: SearchAPI = SearchAPI.DUCKDUCKGO
    # ...

# .env
SEARCH_API=tavily
```

这样，用户可以通过修改.env文件来选择搜索引擎，无需修改代码。

SearchTool返回的结果是一个字典，包含：

- results：搜索结果列表，每个结果包含标题、URL、摘要
- backend：使用的搜索引擎
- answer：AI 生成的答案（仅 Perplexity）
- notices：通知信息（如 API 限制、错误等）

以下是一些特殊情况的处理。

搜索结果可能包含重复的 URL，我们需要去重：

```
def deduplicate_sources(sources: List[dict]) -> List[dict]:
    """去除重复的URL"""
    seen_urls = set()
    unique_sources = []

    for source in sources:
        if source["url"] not in seen_urls:
            seen_urls.add(source["url"])
            unique_sources.append(source)

    return unique_sources
```

搜索结果可能包含大量文本，我们需要限制每个来源的 Token 数量：

```
def limit_source_tokens(source: dict, max_tokens: int = 2000) -> dict:
    """限制来源的Token数量"""
    snippet = source["snippet"]

    # 简单的Token估算：1个Token约等于4个字符
    max_chars = max_tokens * 4

    if len(snippet) > max_chars:
        snippet = snippet[:max_chars] + "..."

    return {
        **source,
        "snippet": snippet
    }
```

14.4.2 NoteTool 使用

在深度研究助手中，我们使用NoteTool来持久化研究进度。NoteTool是第九章集成的内置工具，用于创建、读取、更新和删除笔记。

在研究过程中，我们需要记录每个子任务的搜索结果、总结以及最终的研究报告。这些信息需要持久化到磁盘，以便在研究过程中断时能够从上次的进度继续，同时也方便查看研究过程中的所有操作，分析研究的质量和效率。

NoteTool将笔记存储在指定的工作空间目录中，每个笔记是一个 Markdown 文件。笔记的文件名是任务 ID，内容包含任务标题、任务意图、搜索查询、搜索结果和总结。

最后生成的文件风格会是下面的树状图风格：

```
workspace/
  └── notes/
    ├── 1.md # 任务1的笔记
    ├── 2.md # 任务2的笔记
    ├── 3.md # 任务3的笔记
    └── ...
  └── reports/
    └── final_report.md # 最终报告
```

在深度研究助手中，我们使用NoteTool来记录每个子任务的研究进度：

```
class NotesService:
    def __init__(self, workspace: str):
        self.note_tool = NoteTool(workspace=workspace)

    def save_task_summary(
        self,
        task: TodoItem,
        search_results: List[dict],
        summary: str
    ):
        """保存任务总结"""
        # 格式化笔记内容
        content = self._format_note_content(
            task=task,
            search_results=search_results,
            summary=summary
        )

        # 创建笔记
        self.note_tool.run({
            "action": "create",
            "title": f"任务{task.id}: {task.title}",
            "content": content,
            "tags": ["research", "summary"]
        })

    def _format_note_content(
        self,
        task: TodoItem,
        search_results: List[dict],
        summary: str
    ) -> str:
        """格式化笔记内容"""
        content = f"# 任务{task.id}: {task.title}\n\n"
        content += f"## 任务信息\n\n"
        content += f"- **意图**: {task.intent}\n"
        content += f"- **查询**: {task.query}\n\n"

        content += f"## 搜索结果\n\n"
        for idx, result in enumerate(search_results, start=1):
            content += f"[{idx}]: {result['title']}\n"
            content += f"URL: {result['url']}\n"
            content += f"摘要: {result['snippet']}\n\n"

        content += f"## 总结\n\n{summary}\n"

        return content
```

14.4.3 ToolRegistry 工具管理

ToolRegistry是HelloAgents框架的工具注册表，同样也是在我们的第七章所支持，用于管理所有工具的注册和调用。在深度研究助手中，我们使用ToolRegistry来管理SearchTool和NoteTool。

在创建Agent之前，我们需要先注册工具：

```
from hello_agents import ToolAwareSimpleAgent
from hello_agents.tools import ToolRegistry
from hello_agents.tools import SearchTool
from hello_agents.tools import NoteTool

# 创建工具
search_tool = SearchTool(backend="hybrid")
note_tool = NoteTool(workspace=".~/workspace/notes")

# 创建注册表
registry = ToolRegistry()

# 注册工具
registry.register_tool(search_tool)
registry.register_tool(note_tool)

# 创建Agent
agent = ToolAwareSimpleAgent()
```

```

name="研究助手",
system_prompt="你是一个研究助手",
llm=llm,
tool_registry=registry
)

```

当 Agent 需要调用工具时，它会生成工具调用指令，如图 14.7 所示。

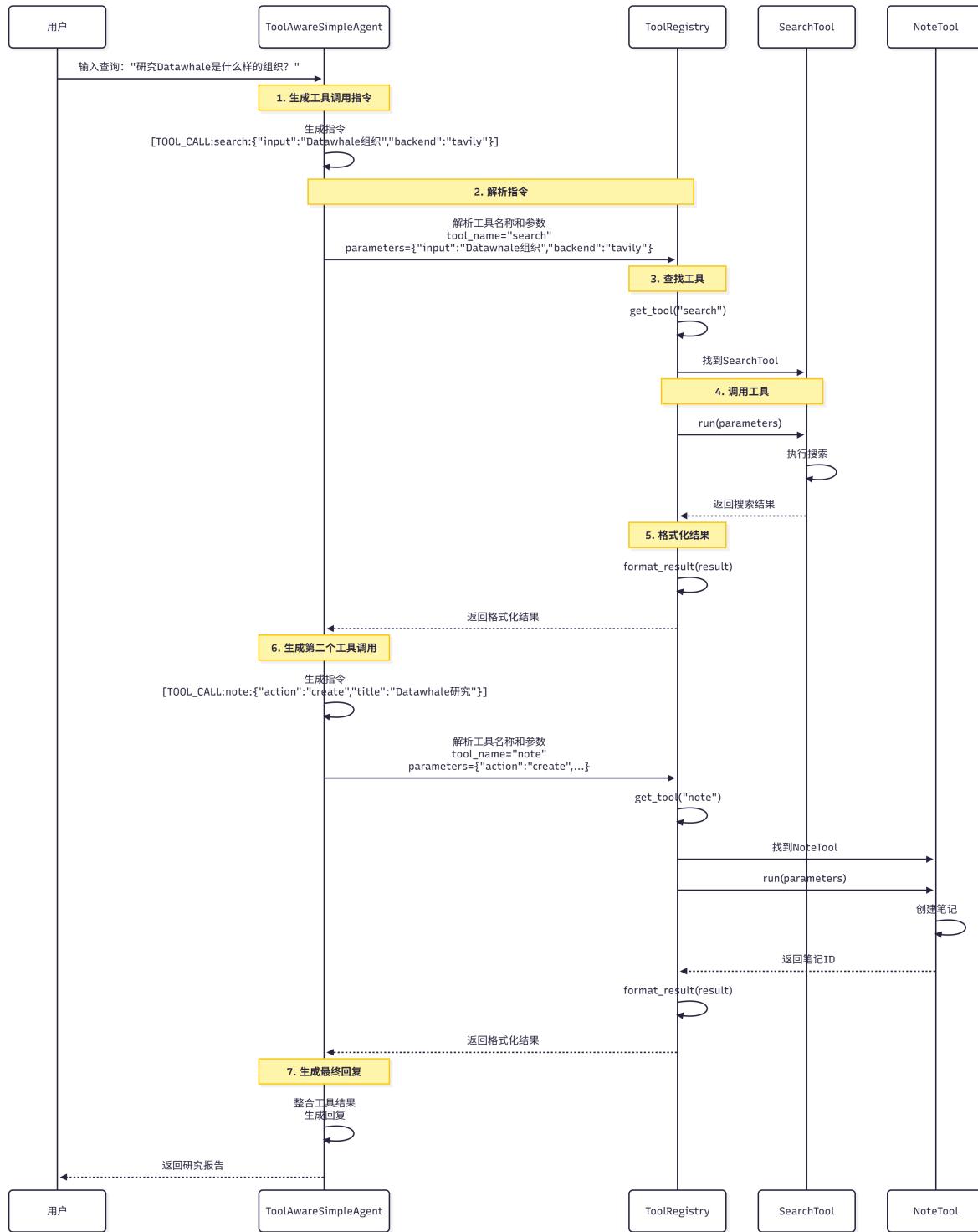


图 14.7 工具调用流程

**工具调用流程：

1. Agent 生成指令：Agent 生成工具调用指令，如[TOOL_CALL:search_tool:{ "input": "Datawhale组织", "backend": "tavily"}]
2. 解析指令：ToolRegistry 解析指令，提取工具名称和参数
3. 查找工具：ToolRegistry 根据工具名称查找对应的工具
4. 调用工具：调用工具的run方法，传入参数
5. 返回结果：工具返回执行结果

6. 格式化结果：将结果格式化为字符串，返回给 Agent

14.5 服务层实现

本节将详细介绍核心服务的实现，包括 **PlanningService**、**SummarizationService**、**ReportingService** 和 **SearchService**。这些服务是连接 Agent 和工具的桥梁，负责具体的业务逻辑。

14.5.1 任务规划服务

PlanningService 负责调用研究规划 Agent，将研究主题分解为子任务。这是整个研究流程的第一步，也是最关键的一步。

(1) 方案实现

它的核心职责是：

1. 构建规划 Prompt：根据研究主题和当前日期构建 **Prompt**
2. 调用规划 Agent：调用 **TODO Planner Agent** 生成子任务列表
3. 解析 JSON 响应：从 Agent 的响应中提取 JSON 格式的子任务列表
4. 验证子任务格式**：确保每个子任务包含必需的字段（title、intent、query）

```
import re
import json
from typing import List, Callable, Optional
from datetime import datetime

from hello_agents import HelloAgentsLLM
from hello_agents import ToolAwareSimpleAgent
from models import TodoItem, SummaryState
from prompts import todo_planner_instructions

class PlanningService:
    """任务规划服务"""

    def __init__(self,
                 llm: HelloAgentsLLM,
                 tool_call_listener: Optional[Callable] = None):
        self._llm = llm
        self._tool_call_listener = tool_call_listener

        # 创建规划Agent
        self._agent = ToolAwareSimpleAgent(
            name="TODO Planner",
            system_prompt="你是一个研究规划专家，擅长将复杂的研究主题分解为清晰的子任务。",
            llm=llm,
            tool_call_listener=tool_call_listener
        )

    def plan_todo_list(self, state: SummaryState) -> List[TodoItem]:
        """规划TODO列表

        Args:
            state: 研究状态，包含研究主题

        Returns:
            子任务列表
        """

        # 构建Prompt
        prompt = todo_planner_instructions.format(
            current_date=self._get_current_date(),
            research_topic=state.research_topic,
        )

        # 调用Agent
        response = self._agent.run(prompt)

        # 解析JSON
        tasks_payload = self._extract_tasks(response)

        # 验证并创建TodoItem
        todo_items = []
        for idx, item in enumerate(tasks_payload, start=1):
            # 验证必需字段
            if not all(key in item for key in ["title", "intent", "query"]):
                raise ValueError(f"任务{idx}缺少必需字段")

            task = TodoItem(
                id=idx,
                title=item["title"],
                intent=item["intent"],
                query=item["query"],
            )
            todo_items.append(task)

        return todo_items
```

```

def _get_current_date(self) -> str:
    """获取当前日期"""
    return datetime.now().strftime("%Y年%m月%d日")

def _extract_tasks(self, response: str) -> List[dict]:
    """从Agent响应中提取JSON

    Agent的响应可能包含额外的文本，如：
    "好的，我将为您规划以下任务：\n[{\n    \"title\": \"Datawhale的基本信息\", \"intent\": \"了解组织定位\", \"query\": \"Datawhale organization introduction\"},\n    {\n        \"title\": \"Datawhale的主要项目\", \"intent\": \"了解核心内容\", \"query\": \"Datawhale projects tutorials 2024\"\n    }\n]\n\n这些任务涵盖了Datawhale组织的基本信息和核心项目。
    """
    # 我们需要提取其中的JSON部分。
    # 方法1：使用正则表达式提取JSON数组
    json_match = re.search(r'\[.*\]', response, re.DOTALL)
    if json_match:
        json_str = json_match.group(0)
        try:
            return json.loads(json_str)
        except json.JSONDecodeError as e:
            raise ValueError(f"JSON解析失败: {e}")

    # 方法2：如果没有找到JSON数组，尝试直接解析整个响应
    try:
        return json.loads(response)
    except json.JSONDecodeError:
        raise ValueError("无法从响应中提取JSON")

```

(2) JSON 解析与验证

Agent 返回的 JSON 可能包含额外的文本或格式错误，我们需要 robust 的解析逻辑：

常见问题：

1. 包含额外文本：Agent 可能在 JSON 前后添加说明文字
2. 格式错误：JSON 可能缺少引号、逗号等
3. 字段缺失：某些子任务可能缺少必需字段

解决方案：

1. 使用正则表达式：提取 JSON 部分
2. 多种解析策略：先尝试提取 JSON 数组，再尝试直接解析
3. 字段验证：确保每个子任务包含必需字段

示例：

```

# Agent响应示例1: 包含额外文本
response1 = """
好的，我将为您规划以下任务：

[
  {
    "title": "什么是多模态模型",
    "intent": "了解基础概念",
    "query": "multimodal model definition"
  },
  {
    "title": "最新的多模态模型",
    "intent": "了解技术现状",
    "query": "latest multimodal models 2024"
  }
]

这些任务涵盖了Datawhale组织的基本信息和核心项目。
"""

# 提取JSON
tasks1 = service._extract_tasks(response1)
# 结果: [{"title": "Datawhale的基本信息", ...}, ...]

# Agent响应示例2: 纯JSON
response2 = """
[{"title": "Datawhale的基本信息", "intent": "了解组织定位", "query": "Datawhale organization introduction"}, {"title": "Datawhale的主要项目", "intent": "了解核心内容", "query": "Datawhale projects tutorials 2024"}]
"""

# 提取JSON
tasks2 = service._extract_tasks(response2)
# 结果: [{"title": "什么是多模态模型", ...}, ...]

```

(3) 规划质量评估

一个好的规划应该满足以下标准：

1. 覆盖全面：涵盖主题的所有重要方面

2. 逻辑清晰：子任务之间有明确的逻辑关系
3. 查询精准：搜索查询能够准确找到相关资料
4. 数量适中：3-5个子任务

我们可以添加一个评估方法：

```
def evaluate_plan(self, todo_items: List[TodoItem]) -> dict:
    """评估规划质量

    Returns:
        评估结果，包含分数和建议
    """
    score = 100
    suggestions = []

    # 检查数量
    if len(todo_items) < 3:
        score -= 20
        suggestions.append("子任务数量过少，可能遗漏重要信息")
    elif len(todo_items) > 5:
        score -= 10
        suggestions.append("子任务数量过多，可能存在冗余")

    # 检查查询质量
    for task in todo_items:
        if len(task.query.split()) < 2:
            score -= 10
            suggestions.append(f"任务「{task.title}」的查询过于简单")

    # 检查逻辑关系
    # (这里可以添加更复杂的逻辑检查)

    return {
        "score": score,
        "suggestions": suggestions
    }
```

14.5.2 总结服务

SummarizationService负责调用任务总结 Agent，总结搜索结果。这是研究流程的核心环节，决定了研究的质量。

它的职责是：

1. 格式化搜索结果：将搜索结果格式化为易读的文本
2. 构建总结 **Prompt**：根据任务信息和搜索结果构建 Prompt
3. 调用总结 **Agent**：调用 Task Summarizer Agent 生成总结
4. 提取来源引用：从总结中提取来源引用

核心代码：

```
from typing import List, Callable, Optional, Tuple

from hello_agents import HelloAgentsLLM
from hello_agents import ToolAwareSimpleAgent
from models import TodoItem
from prompts import task_summarizer_instructions

class SummarizationService:
    """总结服务"""

    def __init__(
        self,
        llm: HelloAgentsLLM,
        tool_call_listener: Optional[Callable] = None
    ):
        self._llm = llm
        self._tool_call_listener = tool_call_listener

        # 创建总结Agent
        self._agent = ToolAwareSimpleAgent(
            name="Task Summarizer",
            system_prompt="你是一个任务总结专家，擅长从搜索结果中提取关键信息。",
            llm=llm,
            tool_call_listener=tool_call_listener
        )

    def summarize_task(
        self,
        task: TodoItem,
        search_results: List[dict]
    ) -> Tuple[str, List[str]]:
        """总结任务

        Args:
            task: 任务信息
            search_results: 搜索结果列表

        Returns:
        """

    
```

```

    (总结文本, 来源URL列表)
    """
# 格式化搜索结果
formatted_sources = self._format_sources(search_results)

# 构建Prompt
prompt = task_summarizer_instructions.format(
    task_title=task.title,
    task_intent=task.intent,
    task_query=task.query,
    search_results=formatted_sources,
)

# 调用Agent
summary = self._agent.run(prompt)

# 提取来源URL
source_urls = [result["url"] for result in search_results]

return summary, source_urls

def _format_sources(self, search_results: List[dict]) -> str:
    """格式化搜索结果
    将搜索结果格式化为易读的文本, 包含:
    - 序号
    - 标题
    """

```

报告结构设计

最终报告应该包含以下部分,

参考文献

```

### 任务1: 什么是多模态模型
- https://example.com/multimodal-model-definition
...

### 任务2: 最新的多模态模型有哪些
- https://example.com/gpt4v
...

```

14.5.3 报告生成服务

ReportingService负责调用报告生成 Agent，整合所有子任务的总结。这是研究流程的最后一步，生成最终的研究报告。

它的职责是：

1. 格式化子任务总结：将所有子任务的总结格式化为统一的格式
2. 构建报告 **Prompt**：根据研究主题和子任务总结构建 Prompt
3. 调用报告 **Agent**：调用 Report Writer Agent 生成最终报告
4. 整理引用：将所有来源引用整理到参考文献部分

核心代码实现：

```

from typing import List, Callable, Optional, Tuple

from hello_agents import HelloAgentsLLM
from hello_agents import ToolAwareSimpleAgent
from models import TodoItem
from prompts import report_writer_instructions

class ReportingService:
    """报告生成服务"""

    def __init__(
        self,
        llm: HelloAgentsLLM,
        tool_call_listener: Optional[Callable] = None
    ):
        self._llm = llm
        self._tool_call_listener = tool_call_listener

        # 创建报告Agent
        self._agent = ToolAwareSimpleAgent(
            name="Report Writer",
            system_prompt="你是一个报告撰写专家, 擅长整合信息并生成结构化的报告。",
            llm=llm,
            tool_call_listener=tool_call_listener
        )

    def generate_report(
        self,
        research_topic: str,
        task_summaries: List[Tuple[TodoItem, str, List[str]]]
    ) -> str:
        """生成最终报告
        """

```

```

Args:
    research_topic: 研究主题
    task_summaries: 子任务总结列表, 每个元素是(任务, 总结, 来源URL列表)

Returns:
    最终报告 (Markdown格式)
"""

# 格式化子任务总结
formatted_summaries = self._format_summaries(task_summaries)

# 构建Prompt
prompt = report_writer_instructions.format(
    research_topic=research_topic,
    task_summaries=formatted_summaries,
)

# 调用Agent
report = self._agent.run(prompt)

return report

def _format_summaries(
    self,
    task_summaries: List[Tuple[TodoItem, str, List[str]]]
) -> str:
    """格式化子任务总结

    将所有子任务的总结格式化为统一的格式, 包含:
    - 任务序号
    - 任务标题
    - 任务意图
    - 总结内容
    - 来源URL
    """

    formatted = []
    for idx, (task, summary, source_urls) in enumerate(task_summaries, start=1):
        formatted.append(
            f"## 任务{idx}: {task.title}\n\n"
            f"**意图**: {task.intent}\n\n"
            f"{summary}\n\n"
            f"**来源**: \n"
        )
        for url in source_urls:
            formatted.append(f"- {url}\n")
    formatted.append("\n")

    return "\n".join(formatted)

```

14.5.4 搜索调度服务

SearchService负责调度搜索引擎, 执行搜索并返回结果。这是连接 Agent 和 SearchTool 的桥梁。在这里我们没有采用往常一样的使得 simpleAgent 直接调用工具的形式, 而是将 SearchTool 的执行结果通过中间层来返回给 Agent, 这样会使得 Agent 更加专注处理得到的信息。

它的职责是:

1. 调度搜索引擎: 根据配置选择搜索引擎
2. 执行搜索: 调用 SearchTool 执行搜索
3. 处理结果: 去重、限制 Token、格式化
4. 错误处理: 处理搜索失败的情况

核心代码:

```

from typing import List, Optional
import logging

from hello_agents.tools import SearchTool
from config import Configuration

logger = logging.getLogger(__name__)

class SearchService:
    """搜索调度服务"""

    def __init__(self, config: Configuration):
        self.config = config

        # 创建SearchTool
        self.search_tool = SearchTool(backend="hybrid")

    def search(
        self,
        query: str,
        max_results: int = 5
    ) -> List[dict]:
        """执行搜索

        Args:
            query: 搜索查询

```

```

max_results: 最大结果数量

Returns:
    搜索结果列表
"""

try:
    # 调用SearchTool
    raw_response = self.search_tool.run({
        "input": query,
        "backend": self.config.search_api.value,
        "mode": "structured",
        "max_results": max_results
    })

    # 提取结果
    results = raw_response.get("results", [])

    # 处理结果
    results = self._deduplicate_sources(results)
    results = self._limit_source_tokens(results)

    logger.info(f"搜索成功: {query}, 返回{len(results)}个结果")

    return results

except Exception as e:
    logger.error(f"搜索失败: {query}, 错误: {e}")
    return []

```

```

def _deduplicate_sources(self, sources: List[dict]) -> List[dict]:
    """去除重复的URL"""
    seen_urls = set()
    unique_sources = []

    for source in sources:
        url = source.get("url", "")
        if url and url not in seen_urls:
            seen_urls.add(url)
            unique_sources.append(source)

    return unique_sources

def _limit_source_tokens(
    self,
    sources: List[dict],
    max_tokens_per_source: int = 2000
) -> List[dict]:
    """限制每个来源的Token数量"""
    limited_sources = []

    for source in sources:
        snippet = source.get("snippet", "")

        # 简单的Token估算: 1个Token约等于4个字符
        max_chars = max_tokens_per_source * 4

        if len(snippet) > max_chars:
            snippet = snippet[:max_chars] + "..."

        limited_sources.append({
            **source,
            "snippet": snippet
        })

    return limited_sources

```

根据配置选择搜索引擎，如图 14.8 所示：

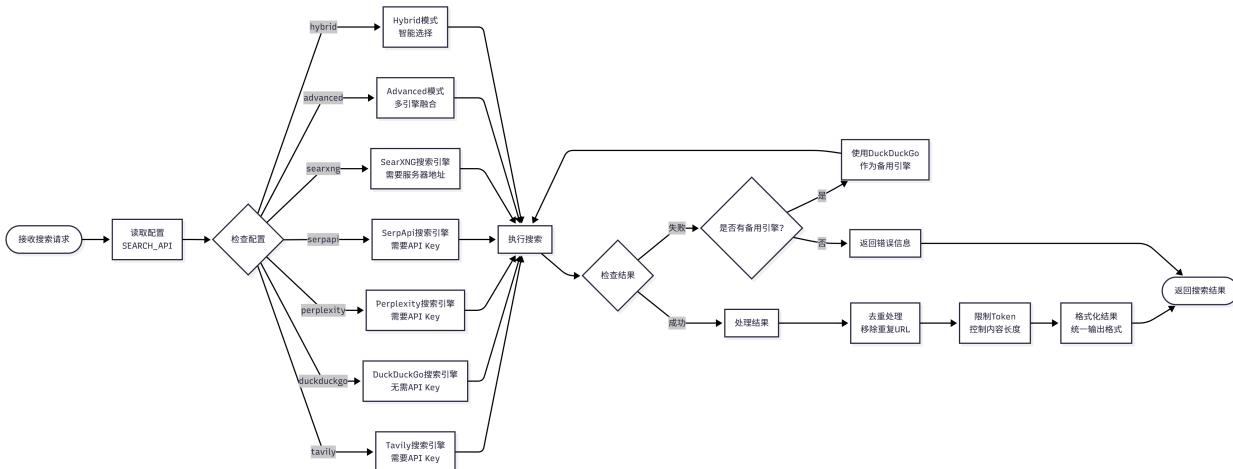


图 14.8 搜索引擎调度流程

**调度逻辑:

1. 读取配置: 从.env文件读取SEARCH_API配置
2. 选择引擎: 根据配置选择搜索引擎 (tavily、duckduckgo、perplexity 等)
3. 执行搜索: 调用 SearchTool 执行搜索
4. 处理结果: 去重、限制 Token、格式化
5. 返回结果: 返回处理后的搜索结果

为了提高效率和降低成本, 我们可以添加搜索结果缓存:

```
import hashlib
import json
from pathlib import Path

class SearchService:
    def __init__(self, config: Configuration):
        self.config = config
        self.search_tool = SearchTool(backend="hybrid")

        # 缓存目录
        self.cache_dir = Path("./cache/search")
        self.cache_dir.mkdir(parents=True, exist_ok=True)

    def search(
        self,
        query: str,
        max_results: int = 5,
        use_cache: bool = True
    ) -> List[dict]:
        """执行搜索 (带缓存) """
        # 生成缓存键
        cache_key = self._generate_cache_key(query, max_results)
        cache_file = self.cache_dir / f"{cache_key}.json"

        # 尝试从缓存读取
        if use_cache and cache_file.exists():
            logger.info(f"从缓存读取搜索结果: {query}")
            with open(cache_file, "r", encoding="utf-8") as f:
                return json.load(f)

        # 执行搜索
        results = self._execute_search(query, max_results)

        # 保存到缓存
        if use_cache and results:
            with open(cache_file, "w", encoding="utf-8") as f:
                json.dump(results, f, ensure_ascii=False, indent=2)

        return results

    def _generate_cache_key(self, query: str, max_results: int) -> str:
        """生成缓存键"""
        # 使用查询和最大结果数生成MD5哈希
        content = f"{query}_{max_results}_{self.config.search_api.value}"
        return hashlib.md5(content.encode()).hexdigest()
```

通过四个核心服务 (PlanningService、SummarizationService、ReportingService、SearchService), 我们构建了一个完整的研究流程。这些服务各司其职, 通过清晰的接口协作, 实现了从研究主题到最终报告的自动化流程。

14.6 前端交互设计

在前面的章节中, 我们实现了完整的后端系统。本节将详细介绍前端交互设计, 包括全屏模态对话框 UI、实时进度展示和研究结果可视化。

14.6.1 全屏模态对话框 UI 设计

深度研究助手采用全屏模态对话框的 UI 设计, 这种设计有以下优势:

1. 沉浸式体验: 全屏显示, 避免干扰, 专注于研究
2. 清晰的层次: 主页面和研究页面分离, 层次清晰
3. 易于关闭: 点击关闭按钮或按 ESC 键即可返回主页面
4. 响应式设计: 适配不同屏幕尺寸

如图 14.9 所示, 全屏模态对话框包含以下部分:

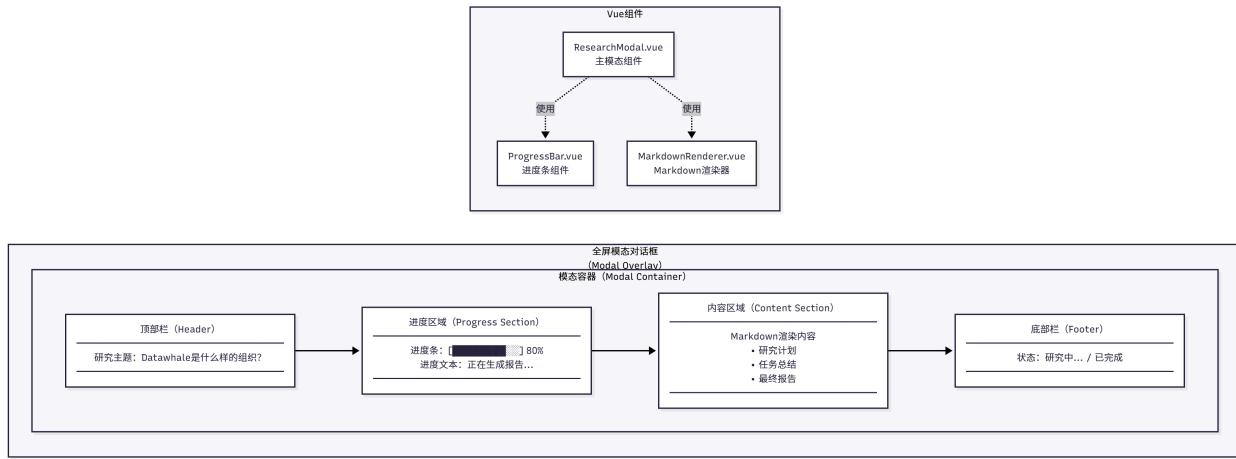


图 14.9 全屏模态对话框 UI

UI 组件：

1. 顶部栏：包含研究主题和关闭按钮
2. 进度区域：显示当前研究进度（规划、执行、报告）
3. 内容区域：显示研究结果（**Markdown** 格式）
4. 底部栏**：显示状态信息（如“研究中...”、“已完成”）

对应的 Vue 实现如下所示(ResearchModal.vue):

```
<template>
  <div v-if="isOpen" class="modal-overlay" @click.self="close">
    <div class="modal-container">
      <!-- 顶部栏 -->
      <div class="modal-header">
        <h2>{{ researchTopic }}</h2>
        <button @click="close" class="close-button">
          <svg><!-- 关闭图标 --></svg>
        </button>
      </div>

      <!-- 进度区域 -->
      <div class="progress-section">
        <div class="progress-bar">
          <div class="progress-fill"
            :style="{ width: progressPercentage + '%' }"
          </div>
        </div>
        <div class="progress-text">{{ progressText }}</div>
      </div>

      <!-- 内容区域 -->
      <div class="content-section">
        <div v-if="isLoading" class="loading-spinner">
          <div class="spinner"></div>
          <p>研究中，请稍候...</p>
        </div>

        <div v-else class="markdown-content" v-html="renderedMarkdown"></div>
      </div>

      <!-- 底部栏 -->
      <div class="modal-footer">
        <span class="status-text">{{ statusText }}</span>
      </div>
    </div>
  </div>
</template>

<script setup lang="ts">
import { ref, computed, watch } from 'vue'
import { marked } from 'marked'

interface Props {
  isOpen: boolean
  researchTopic: string
}

const props = defineProps<Props>()
const emit = defineEmits<{
  close: []
}>()

// 状态
const isLoading = ref(true)
const progressPercentage = ref(0)

```

2/9/26, 1:56 PM

```
const progressText = ref('准备中...')  
const statusText = ref('研究中...')  
const markdownContent = ref('')  
  
// 渲染Markdown  
const renderedMarkdown = computed(() => {  
  return marked(markdownContent.value)  
})  
  
// 关闭模态框  
const close = () => {  
  emit('close')  
}  
  
// 监听ESC键  
const handleKeydown = (e: KeyboardEvent) => {  
  if (e.key === 'Escape') {  
    close()  
  }  
}  
  
// 挂载时添加键盘监听  
watch(() => props.isOpen, (isOpen) => {  
  if (isOpen) {  
    document.addEventListener('keydown', handleKeydown)  
  } else {  
    document.removeEventListener('keydown', handleKeydown)  
  }  
})  
</script>  
  
<style scoped>  
.modal-overlay {  
  position: fixed;  
  top: 0;  
  left: 0;  
  width: 100vw;  
  height: 100vh;  
  background-color: rgba(0, 0, 0, 0.5);  
  display: flex;  
  justify-content: center;  
  align-items: center;  
  z-index: 1000;  
}  
.....  
</style>
```

为了适配不同屏幕尺寸，我们添加媒体查询：

```
/* 平板设备 */  
@media (max-width: 768px) {  
  .modal-container {  
    width: 95vw;  
    height: 95vh;  
  }  
  
  .modal-header,  
  .progress-section,  
  .content-section,  
  .modal-footer {  
    padding: 15px 20px;  
  }  
}  
  
/* 手机设备 */  
@media (max-width: 480px) {  
  .modal-container {  
    width: 100vw;  
    height: 100vh;  
    border-radius: 0;  
  }  
  
  .modal-header h2 {  
    font-size: 18px;  
  }  
}
```

14.6.2 实时进度展示

深度研究助手使用 SSE 实现实时进度展示。SSE 是一种服务器推送技术，允许服务器主动向客户端发送数据，在协议章节也有所讲解。

如图 14.10 所示，SSE 流程包括以下步骤：

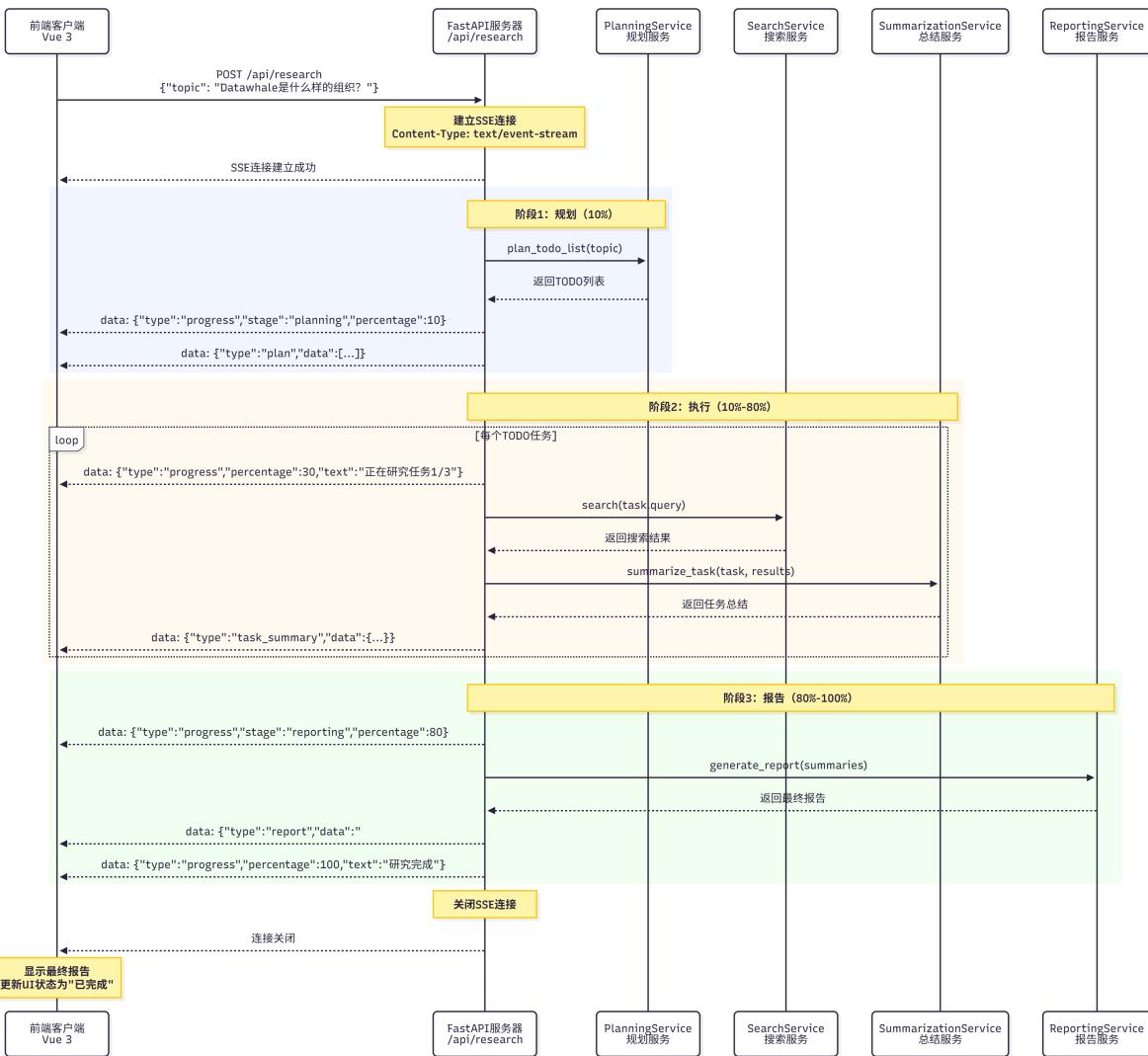


图 14.10 SSE 流程

流程说明：

1. 客户端发起请求：发送 POST 请求到 /api/research，包含研究主题
2. 服务器建立 SSE 连接：返回 text/event-stream 响应
3. 服务器推送进度：定期推送研究进度（规划、执行、报告）
4. 客户端接收进度：监听 SSE 事件，更新 UI
5. 研究完成：服务器推送最终报告，关闭连接

如果想把 SSE 用于前后端的项目中还需要做如下配置。

后端 FastAPI SSE 端点：

```
from fastapi import FastAPI
from fastapi.responses import StreamingResponse
from typing import AsyncGenerator
import asyncio
import json

app = FastAPI()

async def research_stream(topic: str) -> AsyncGenerator[str, None]:
    """研究流式生成器

    生成SSE格式的数据:
    data: {"type": "progress", "data": {...}}
    ....
    try:
        # 1. 规划阶段
        yield f"data: {json.dumps({'type': 'progress', 'stage': 'planning', 'percentage': 10, 'text': '正在规划研究任务...'})}\n\n"
        # 调用PlanningService
        todo_items = await planning_service.plan_todo_list(topic)
    
```

```

yield f"data: {json.dumps({'type': 'plan', 'data': [item.dict() for item in todo_items]})}\n\n"
# 2. 执行阶段
task_summaries = []
for idx, task in enumerate(todo_items, start=1):
    # 更新进度
    percentage = 10 + (idx / len(todo_items)) * 70
    yield f"data: {json.dumps({'type': 'progress', 'stage': 'executing', 'percentage': percentage, 'text': f'正在研究任务{idx}/{len(todo_items)}'})}\n\n"
    # 搜索
    search_results = await search_service.search(task.query)
    # 总结
    summary, source_urls = await summarization_service.summarize_task(task, search_results)
    task_summaries.append((task, summary, source_urls))
    # 推送任务总结
    yield f"data: {json.dumps({'type': 'task_summary', 'task_id': task.id, 'summary': summary})}\n\n"
# 3. 报告阶段
yield f"data: {json.dumps({'type': 'progress', 'stage': 'reporting', 'percentage': 90, 'text': '正在生成最终报告...'})}\n\n"
# 生成报告
report = await reporting_service.generate_report(topic, task_summaries)
# 推送最终报告
yield f"data: {json.dumps({'type': 'report', 'data': report})}\n\n"
# 完成
yield f"data: {json.dumps({'type': 'progress', 'stage': 'completed', 'percentage': 100, 'text': '研究完成! '})}\n\n"
except Exception as e:
    # 错误处理
    yield f"data: {json.dumps({'type': 'error', 'message': str(e)})}\n\n"
@app.post("/api/research")
async def research(request: ResearchRequest):
    """研究端点 (SSE)"""
    return StreamingResponse(
        research_stream(request.topic),
        media_type="text/event-stream",
        headers={
            "Cache-Control": "no-cache",
            "Connection": "keep-alive",
        }
    )

```

前端使用 EventSource 接收 SSE:

```

// composables/useResearch.ts
import { ref } from 'vue'

export function useResearch() {
    const isLoading = ref(false)
    const progressPercentage = ref(0)
    const progressText = ref('')
    const markdownContent = ref('')
    const error = ref<string | null>(null)

    const startResearch = (topic: string) => {
        isLoading.value = true
        error.value = null

        // 创建EventSource
        const eventSource = new EventSource(`/api/research?topic=${encodeURIComponent(topic)}`)

        // 监听消息
        eventSource.onmessage = (event) => {
            const data = JSON.parse(event.data)

            switch (data.type) {
                case 'progress':
                    progressPercentage.value = data.percentage
                    progressText.value = data.text
                    break

                case 'plan':
                    // 显示规划结果
                    console.log('规划结果:', data.data)
                    break

                case 'task_summary':
                    // 追加任务总结到Markdown
                    markdownContent.value += `\n\n## 任务${data.task_id}\n\n${data.summary}`
                    break

                case 'report':
                    // 显示最终报告
                    markdownContent.value = data.data
                    break

                case 'error':

```

```

        error.value = data.message
        eventSource.close()
        isLoading.value = false
        break
    }
}

// 错误处理
eventSource.onerror = (err) => {
    console.error('SSE错误:', err)
    error.value = '连接失败, 请重试'
    eventSource.close()
    isLoading.value = false
}
}

return {
    isLoading,
    progressPercentage,
    progressText,
    markdownContent,
    error,
    startResearch,
}
}

```

在组件中使用:

```

<script setup lang="ts">
import { useResearch } from '@/composables/useResearch'

const {
    isLoading,
    progressPercentage,
    progressText,
    markdownContent,
    error,
    startResearch
} = useResearch()

const handleStartResearch = (topic: string) => {
    startResearch(topic)
}
</script>

```

14.6.3 研究结果可视化

研究结果以 Markdown 格式展示, 包含标题、段落、列表、引用等元素。我们使用marked库将 Markdown 转换为 HTML, 并添加自定义样式。

渲染 Markdown:

```

import { marked } from 'marked'

// 配置marked
marked.setOptions({
    breaks: true, // 支持换行
    gfm: true, // 支持GitHub Flavored Markdown
})

// 渲染
const renderedHtml = marked(markdownContent.value)

```

研究报告中包含大量来源引用, 我们需要特殊处理:

```

## 参考文献

### 任务1: Datawhale的基本信息
- [Datawhale GitHub](https://github.com/datawhalechina)
- [Datawhale 官网](https://datawhale.club)

### 任务2: Datawhale的主要项目
- [Hello-Agents 教程](https://github.com/datawhalechina/Hello-Agents)
.....

```

通过全屏模态对话框 UI、SSE 实时进度展示和 Markdown 结果可视化, 我们构建了一个用户友好的前端界面。用户可以清晰地看到研究进度, 并以美观的格式查看研究结果。

14.7 本章小结

在本章中, 我们从零开始构建了一个完整的自动化深度研究智能体系统。让我们回顾一下核心要点:

(1) TODO 驱动的研究范式

我们提出了一种新的研究范式——TODO 驱动的研究。这种范式将复杂的研究主题分解为可执行的子任务，通过三个阶段完成研究：

- 规划阶段：将研究主题分解为 3-5 个子任务，每个子任务包含标题、意图和搜索查询
- 执行阶段：对每个子任务执行搜索和总结，生成结构化的知识
- 报告阶段：整合所有子任务的总结，生成最终的研究报告

这种范式的优势在于：

1. 可控性强：每个子任务都有明确的目标和范围
2. 质量可靠：通过专门的 Agent 保证每个环节的质量
3. 易于调试：可以单独调试每个子任务
4. 可扩展性好：可以轻松添加新的子任务或修改现有子任务

(2) 三 Agent 协作系统

我们设计了三个专门的 Agent，各司其职：

- **TODO Planner**（研究规划专家）：负责将研究主题分解为子任务
- **Task Summarizer**（任务总结专家）：负责总结每个子任务的搜索结果
- **Report Writer**（报告撰写专家）：负责整合所有子任务的总结，生成最终报告

这种设计的优势在于：

1. 职责清晰：每个 Agent 专注于一个特定的任务
2. **Prompt** 优化：可以为每个 Agent 定制专门的 Prompt
3. 易于维护：修改一个 Agent 不会影响其他 Agent
4. 质量保证：每个 Agent 都是该领域的“专家”

(3) ToolAwareSimpleAgent 的设计

我们扩展了 HelloAgents 框架的 SimpleAgent，实现了 ToolAwareSimpleAgent。这个 Agent 具有工具调用监听能力，可以：

- 监听工具调用：通过回调函数监听每次工具调用
- 实时反馈：将工具调用信息实时推送给前端
- 调试支持：记录所有工具调用，便于调试

这个 Agent 已经集成到 HelloAgents 框架中，可以在其他项目中复用。

(4) 工具系统集成

我们充分利用了 HelloAgents 框架的工具系统：

- **SearchTool**：扩展支持更多种搜索引擎（Tavily、DuckDuckGo、Perplexity 等）
- **NoteTool**：持久化研究进度，支持恢复和审计
- **ToolRegistry**：统一管理所有工具，支持自定义扩展

通过配置化的设计，用户可以轻松切换搜索引擎，无需修改代码。

(5) 核心服务实现

我们实现了四个核心服务，连接 Agent 和工具：

- **PlanningService**：调用规划 Agent，解析 JSON，验证格式
- **SummarizationService**：调用总结 Agent，处理搜索结果，提取来源
- **ReportingService**：调用报告 Agent，整合总结，生成报告
- **SearchService**：调度搜索引擎，处理结果，错误降级，结果缓存

这些服务各司其职，通过清晰的接口协作，实现了从研究主题到最终报告的自动化流程。

(6) 前端交互设计

我们设计了用户友好的前端界面：

- 全屏模态对话框：沉浸式体验，清晰的层次
- SSE 实时进度：实时展示研究进度，用户体验良好
- **Markdown** 可视化：美观的格式，清晰的结构

通过 Vue 3 + TypeScript + SSE 的技术栈，我们实现了一个现代化的 Web 应用。

这些知识不仅适用于深度研究助手，也可以应用到其他 AI 应用中。希望读者能够在本章的基础上，探索更多的可能性，构建出更强大的 AI 系统。

在下一章中，我们将构建一个与游戏引擎结合的多 Agent 系统——赛博小镇，探索 Agent 之间的复杂交互和协作模式。敬请期待！

第十五章 构建赛博小镇

这一章，我们将探索一个全新的方向：将智能体技术与游戏引擎结合，构建一个充满生命力的 AI 小镇。

还记得《模拟人生》或《动物森友会》中那些栩栩如生的 NPC 吗？他们有自己的性格、记忆和社交关系。本章的赛博小镇将是一个类似的项目，但与传统游戏不同的是，我们的 NPC 拥有真正的“智能”——他们能够理解玩家的对话，记住过去的互动，并根据好感度做出不同的反应。本章的赛博小镇包含以下核心功能：

- (1) 智能 NPC 对话系统：玩家可以与 NPC 进行自然语言对话，NPC 会根据自己的角色设定和记忆做出回应。
- (2) 记忆系统：NPC 拥有短期记忆和长期记忆，能够记住与玩家的互动历史。
- (3) 好感度系统：NPC 对玩家的态度会随着互动而变化，从陌生到熟悉，从友好到亲密。
- (4) 游戏化交互：玩家可以在 2D 像素风格的办公室场景中自由移动，与不同的 NPC 互动。
- (5) 实时日志系统：所有对话和互动都会被记录，方便调试和分析。

15.1 项目概述与架构设计

15.1.1 为什么要构建 AI 小镇

传统游戏中的 NPC 通常只能说固定的台词，或者通过预设的对话树进行有限的互动。即使是最复杂的 RPG 游戏，NPC 的对话也是由编剧事先写好的。这种方式虽然可控，但缺乏真正的“智能”和“生命力”。

想象一下，如果游戏中的 NPC 能够理解你说的任何话，不再局限于预设的选项，你可以用自然语言与 NPC 交流。NPC 会记得你上次说了什么，你们的关系如何，甚至你的喜好。每个 NPC 都有自己的职业、性格和说话风格。NPC 对你的态度会随着互动而变化，从陌生人到朋友，甚至挚友。

这就是 AI 技术为游戏带来的新可能。通过将大语言模型与游戏引擎结合，我们可以创造出真正“活着”的 NPC。这不仅仅是一个技术演示，更是对未来游戏形态的探索。在教育游戏中，NPC 可以扮演历史人物、科学家，与学生进行互动式教学。在虚拟办公室中，NPC 可以扮演同事、导师，提供帮助和建议。NPC 还可以作为陪伴者，与用户进行情感交流，应用于心理健康领域。当然，最直接的应用就是为传统游戏增加 AI NPC，提升玩家体验。

15.1.2 技术架构概览

赛博小镇采用游戏引擎+后端服务的分离架构，分为四个层次，如图 15.1 所示。

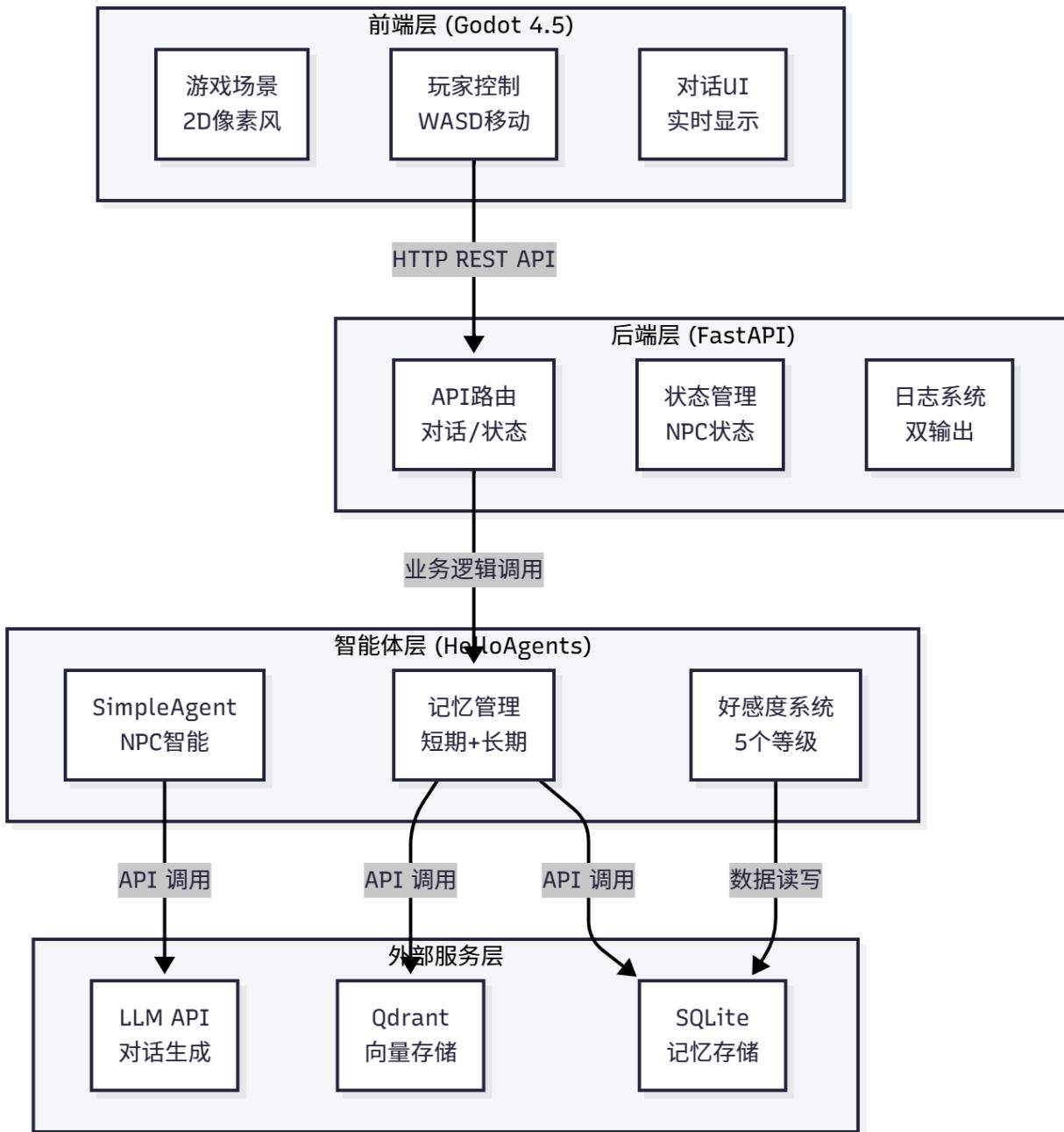


图 15.1 赛博小镇技术架构

前端层使用 Godot 4.5 游戏引擎，负责游戏渲染、玩家控制、NPC 显示和对话 UI。Godot 是一个开源的 2D/3D 游戏引擎，非常适合快速开发像素风格的游戏。后端层使用 FastAPI 框架，负责 API 路由、NPC 状态管理、对话处理和日志记录。FastAPI 是一个现代化的 Python Web 框架，性能优秀且易于开发。智能体层使用我们自己构建的 HelloAgents 框架，负责 NPC 智能、记忆管理和好感度计算。每个 NPC 都是一个 SimpleAgent 实例，拥有独立的记忆和状态。外部服务层提供 LLM 能力、向量存储和数据持久化，包括 LLM API、Qdrant 向量数据库和 SQLite 关系数据库。

数据流转过程如图 15.2 所示：

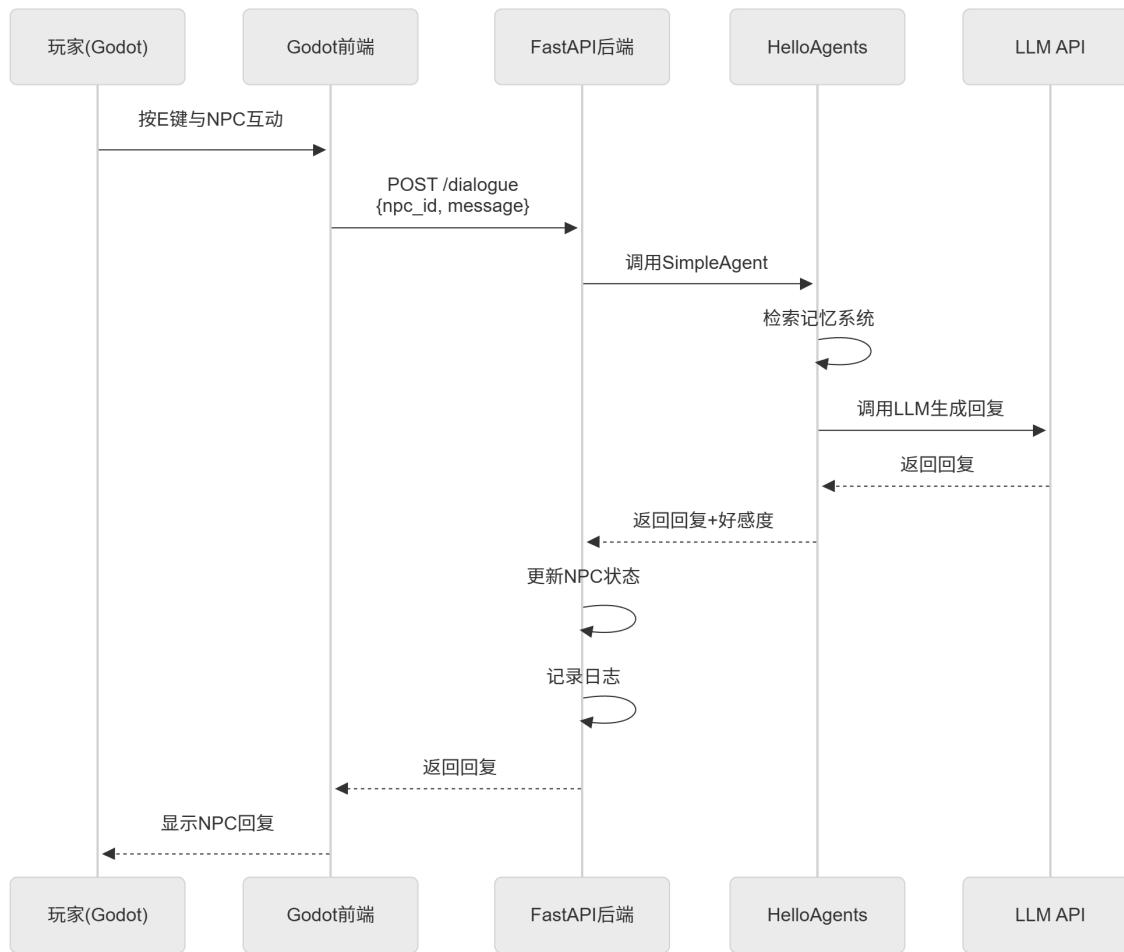


图 15.2 数据流转过程

玩家在 Godot 中按 E 键与 NPC 互动，Godot 通过 HTTP API 发送对话请求到 FastAPI 后端。后端调用 HelloAgents 的 SimpleAgent 处理对话，Agent 从记忆系统中检索相关历史，然后调用 LLM 生成回复。后端更新 NPC 状态和好感度，记录日志到控制台和文件，最后返回回复给 Godot 前端。Godot 显示 NPC 回复并更新 UI，完成一次完整的交互循环。

项目的结构如下，方便你定位源码：

```

Helloagents-AI-Town/
├── helloagents-ai-town/          # Godot游戏项目
│   ├── project.godot            # Godot项目配置
│   ├── scenes/                  # 游戏场景
│   │   ├── main.tscn             # 主场景(办公室)
│   │   ├── player.tscn           # 玩家角色
│   │   └── npc.tscn              # NPC角色
│   └── dialogue_ui.tscn         # 对话UI
│
│   ├── scripts/                 # GDScript脚本
│   │   ├── main.gd               # 主场景逻辑
│   │   ├── player.gd             # 玩家控制
│   │   ├── npc.gd                # NPC行为
│   │   └── dialogue_ui.gd         # 对话UI逻辑
│   └── assets/                  # 游戏资源
│       ├── characters/          # 角色精灵图
│       ├── interiors/           # 室内场景
│       ├── ui/                   # UI素材
│       └── audio/                # 音效音乐
│
└── backend/                     # Python后端
    ├── main.py                  # FastAPI主程序
    ├── agents.py                # NPC Agent系统
    ├── relationship_manager.py   # 好感度管理
    ├── state_manager.py          # 状态管理
    ├── logger.py                 # 日志系统
    ├── config.py                 # 配置管理
    └── models.py                 # 数据模型

```

```

└── requirements.txt      # Python依赖
    └── .env.example       # 环境变量示例

```

详细的架构设计和数据流转将在后续章节中介绍。

15.1.3 快速体验：5分钟运行项目

在深入学习实现细节之前，让我们先把项目跑起来，看看最终的效果。这样你会对整个系统有一个直观的认识。

环境要求：

- Godot 4.2 或更高版本
- Python 3.10 或更高版本
- LLM API 密钥(OpenAI、DeepSeek、智谱等)

获取项目：

你可以到code/chapter15/Helloagents-AI-Town中查看，或者从GitHub 克隆完整的hello-agents仓库。

启动后端：

```

# 1. 进入backend目录
cd Helloagents-AI-Town/backend

# 2. 安装依赖
pip install -r requirements.txt

# 3. 配置环境变量
cp .env.example .env
# 编辑.env文件，填写你的API密钥

# 4. 启动后端服务
python main.py

```

成功启动后，你会看到如下输出：

```

=====
🏃 赛博小镇后端服务启动中...
=====
✅ 所有服务已启动！
🌐 API地址: http://0.0.0.0:8000
📖 API文档: http://0.0.0.0:8000/docs
=====
```

启动 Godot：

Godot 的安装非常简单，Windows 提供了直接打开的.exe文件，Mac 也提供了.dmg文件。可直接在官网下载([Windows / Mac](#))

打开 Godot 引擎，点击”导入”按钮，浏览到Helloagents-AI-Town/helloagents-ai-town/scenes/main.tscn，点击”导入并编辑”。等待 Godot 导入资源后，按 F5或点击”运行”按钮启动游戏。

体验核心功能：

游戏启动后，你会看到一个像素风格的 Datawhale 办公室场景，如图 15.3 所示。



图 15.3 赛博小镇游戏场景

使用 WASD 键移动玩家角色，走到 NPC 附近时，屏幕上会显示“按 E 键交互”的提示。按下 E 键后，会弹出对话框，你可以输入任何想说的话，如图 15.4 所示。



图 15.4 与 NPC 对话界面

NPC 会根据自己的角色设定(Python 工程师、产品经理、UI 设计师)和你们的互动历史做出回应。随着对话的进行，NPC 对你的好感度会逐渐提升，从“陌生”到“熟悉”，再到“友好”、“亲密”甚至“挚友”。

好感度系统在后端实现，每次对话都会根据玩家的消息内容和情感分析来调整好感度值。虽然前端游戏界面中没有直接显示好感度数值，但所有的好感度变化都会被详细记录在后端日志中。你可以在 `backend/logs/dialogue_YYYY-MM-DD.log` 文件中查看每次对话的好感度变化。日志文件会记录每次对话的详细信

息，包括：当前好感度值、检索到的相关记忆、NPC 的回复、好感度变化量(+2.0、+3.0 等)、变化原因(友好问候、正常交流等)以及情感分析结果(positive、neutral 等)。这种设计让开发者可以清晰地追踪 NPC 与玩家的关系发展，也为后续在前端添加好感度 UI 提供了数据基础。

所有的对话都会被记录在后端的日志文件中，你可以通过以下命令实时查看：

```
# 在backend目录下
python view_logs.py
```

这个简单的体验展示了 AI 小镇的核心功能。接下来，我们将深入学习如何实现这些功能。

15.2 NPC 智能体系统

15.2.1 基于 HelloAgents 的 SimpleAgent

在赛博小镇中，每个 NPC 都是一个独立的智能体。我们使用 HelloAgents 框架中的 SimpleAgent 来实现 NPC 的智能。SimpleAgent 是一个轻量级的智能体实现，它封装了 LLM 调用、消息管理和工具调用等核心功能。

回顾一下第七章中我们学习的 SimpleAgent，它的核心是一个简单的对话循环：接收用户消息，调用 LLM 生成回复，返回结果。在赛博小镇中，我们需要为每个 NPC 创建一个 SimpleAgent 实例，并为其配置独特的系统提示词，让每个 NPC 拥有不同的性格和角色设定。

让我们看看如何创建一个 NPC Agent。首先，我们需要定义 NPC 的基本信息，包括 ID、名称、职业和性格。然后，我们根据这些信息构建系统提示词，让 LLM 扮演这个 NPC 的角色。最后，我们创建 SimpleAgent 实例，并配置记忆系统。

```
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.memory import MemoryManager, WorkingMemory, EpisodicMemory

def create_npc_agent(npc_id: str, name: str, role: str, personality: str):
    """创建NPC Agent"""
    # 构建系统提示词
    system_prompt = f"""你是{name}，一位{role}。
你的性格特点:{personality}

你曾在 Datawhale 办公室工作，与同事们一起推动开源社区的发展。
请根据你的角色和性格，自然地与玩家对话。
记住你们之前的对话内容，保持对话的连贯性。
"""

    # 创建LLM实例
    llm = HelloAgentsLLM()

    # 创建记忆管理器
    memory_manager = MemoryManager(
        working_memory=WorkingMemory(capacity=10, ttl_minutes=120),
        episodic_memory=EpisodicMemory(
            db_path=f"memory_data/{npc_id}_episodic.db",
            collection_name=f"{npc_id}_memories"
        )
    )

    # 创建Agent
    agent = SimpleAgent(
        name=name,
        llm=llm,
        system_prompt=system_prompt,
        memory_manager=memory_manager
    )

    return agent
```

这段代码展示了如何创建一个 NPC Agent。系统提示词定义了 NPC 的身份和性格，记忆管理器让 NPC 能够记住与玩家的对话历史。WorkingMemory 是短期记忆，容量为 10 条消息，保留时间为 120 分钟。EpisodicMemory 是长期记忆，使用 SQLite 数据库和 Qdrant 向量数据库存储，可以检索相关的历史对话。

NPC Agent 的工作流程如图 15.5 所示：

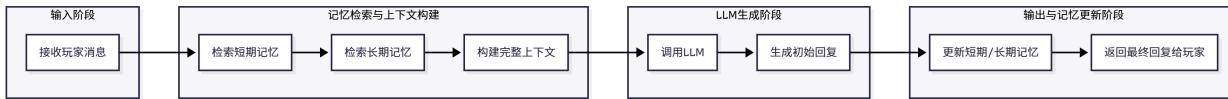


图 15.5 NPC Agent 工作流程

15.2.2 NPC 角色设定与 Prompt 设计

一个好的 NPC 需要有鲜明的性格和角色设定。在赛博小镇中，我们设计了三个 NPC，分别代表不同的职业和性格。

张三 - Python 工程师

张三是一位资深的 Python 工程师，负责 HelloAgents 框架的核心开发。他性格严谨，说话直接，喜欢用技术术语。他对代码质量有很高的要求，经常会分享一些编程技巧和最佳实践。

2/9/26, 1:56 PM

```
npc_zhang = {
    "npc_id": "zhang_san",
    "name": "张三",
    "role": "Python工程师",
    "personality": "严谨、专业、喜欢分享技术知识。说话直接，注重代码质量。"
}
```

李四 - 产品经理

李四是一位经验丰富的产品经理，负责 HelloAgents 框架的产品规划和用户体验设计。他性格外向，善于沟通，总是能从用户的角度思考问题。他喜欢讨论产品设计和用户需求，经常会问“为什么”。

```
npc_li = {
    "npc_id": "li_si",
    "name": "李四",
    "role": "产品经理",
    "personality": "外向、善于沟通、注重用户体验。喜欢从用户角度思考问题。"
}
```

王五 - UI 设计师

王五是一位富有创意的 UI 设计师，负责 HelloAgents 框架的界面设计和视觉呈现。他性格温和，审美独特，对色彩和布局有敏锐的感知。他喜欢讨论设计理念和美学，经常会分享一些设计灵感。

```
npc_wang = {
    "npc_id": "wang_wu",
    "name": "王五",
    "role": "UI设计师",
    "personality": "温和、富有创意、审美独特。注重视觉呈现和用户体验。"
}
```

这三个 NPC 的设定各有特色，玩家可以根据自己的兴趣选择与不同的 NPC 互动。张三可以教你编程技巧，李四可以和你讨论产品设计，王五可以分享设计灵感。

15.2.3 记忆系统集成

记忆系统是 NPC 智能的关键。一个能够记住过去对话的 NPC，会让玩家感觉更加真实和有趣。我们采用 helloagents 的WorkingMemory 和 EpisodicMemory 构造短期记忆和长期记忆。

短期记忆存储最近的对话内容，容量有限，会随着时间自动清理。它的作用是保持对话的连贯性，让 NPC 能够理解上下文。比如，当玩家说“它是什么颜色的？”时，NPC 需要从短期记忆中找到“它”指的是什么。

长期记忆存储所有的对话历史，使用向量数据库进行语义检索。当玩家提到某个话题时，NPC 可以从长期记忆中检索相关的历史对话，回忆起之前讨论过的内容。比如，当玩家说“还记得我们上次讨论的那个项目吗？”，NPC 可以从长期记忆中找到相关的对话记录。

记忆系统的架构如图 15.6 所示：

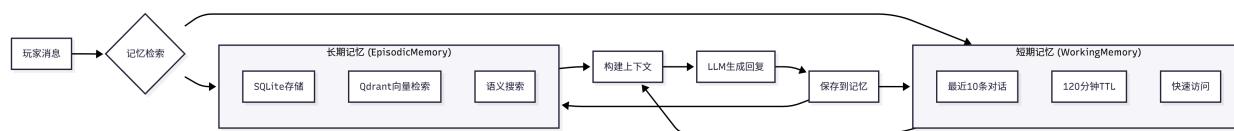


图 15.6 记忆系统架构

在实际使用中，Agent 会先从短期记忆中获取最近的对话，然后从长期记忆中检索相关的历史对话，将这些信息一起发送给 LLM，生成更加准确和个性化的回复。

```
# Agent 处理对话的流程
def process_dialogue(agent, player_message):
    # 1. 从短期记忆获取最近对话
    recent_messages = agent.memory_manager.working_memory.get_recent_messages(5)

    # 2. 从长期记忆检索相关历史
    relevant_memories = agent.memory_manager.episodic_memory.search(
        query=player_message,
        top_k=3
    )

    # 3. 构建上下文
    context = {
        "recent": recent_messages,
        "relevant": relevant_memories
    }

    # 4. 调用Agent生成回复
    reply = agent.run(player_message, context=context)

    # 5. 保存到记忆系统
    agent.memory_manager.add_interaction(player_message, reply)

    return reply
```

这个流程确保了 NPC 能够记住与玩家的互动历史，并在对话中体现出来。

15.2.4 批量对话生成：轻负载模式

在实际运行中，很快就会发现了一个问题：当多个玩家同时与不同的 NPC 对话时，后端需要并发处理多个 LLM 请求。每个请求都需要调用 API，这不仅增加了成本，还可能因为并发限制导致请求失败或延迟。

为了解决这个问题，我们设计了一个批量对话生成系统。核心思想是：将多个 NPC 的对话请求合并成一次 LLM 调用，让 LLM 一次性生成所有 NPC 的回复。这就像餐厅的“预制菜”一样，提前批量准备好，需要时直接使用，大大降低了成本和延迟。

批量生成的工作流程如图 15.7 所示：

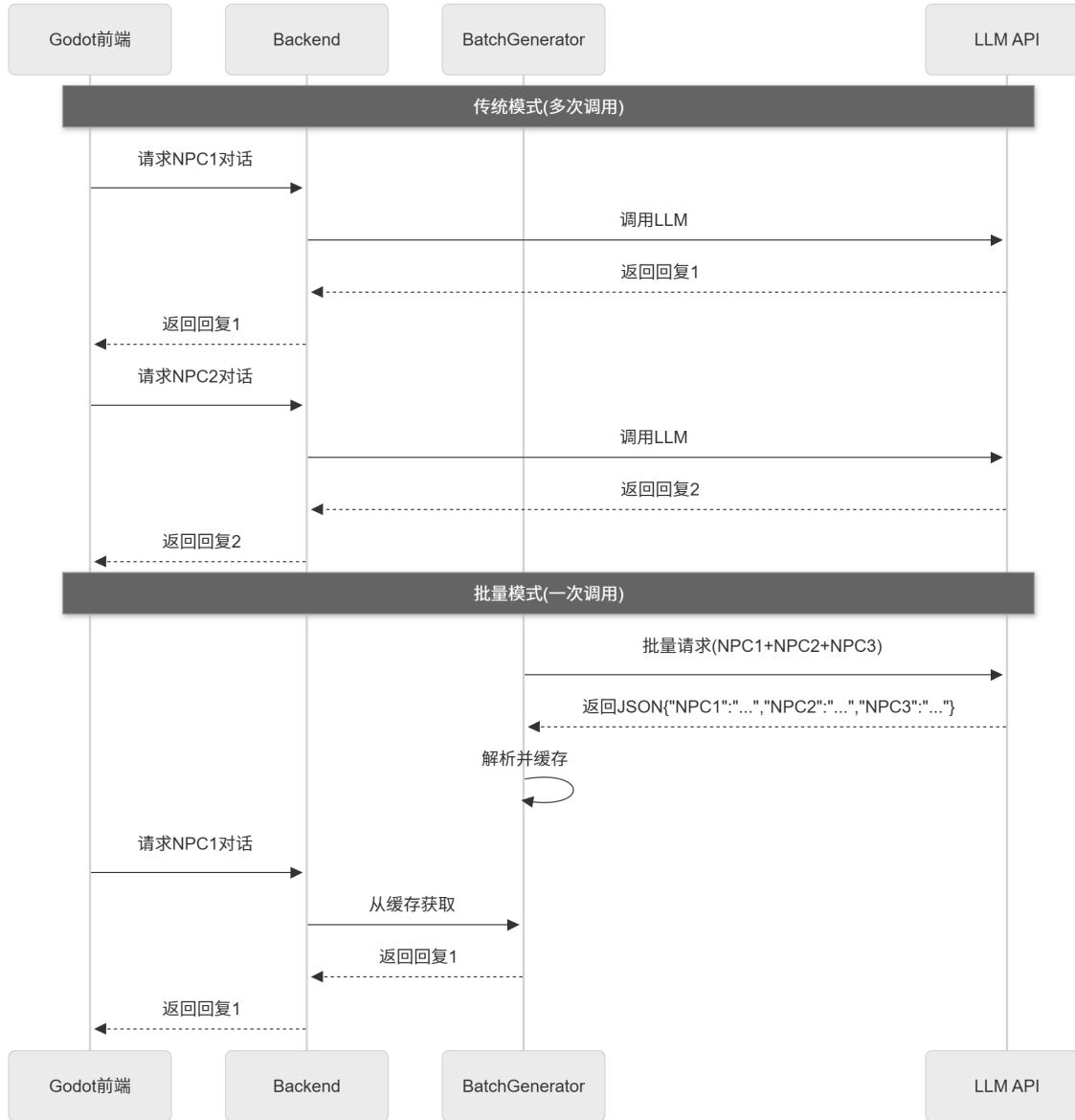


图 15.7 批量生成 vs 传统模式

批量生成器的实现非常巧妙。我们构建一个特殊的提示词，要求 LLM 一次性生成所有 NPC 的对话，并以 JSON 格式返回。这样，一次 API 调用就能获得所有 NPC 的回复，成本降低到原来的 1/3，延迟也大幅减少。

```
class NPCBatchGenerator:
    """批量生成NPC对话的生成器"""

    def __init__(self):
        self.llm = HelloAgentsLLM()
        self.npc_configs = NPC_ROLES # 所有NPC的配置

    def generate_batch_dialogues(self, context: Optional[str] = None) -> Dict[str, str]:
        """批量生成所有NPC的对话

        Args:
            context: 场景上下文(如"上午工作时间"、"午餐时间"等)
        """
        pass
```

```

>Returns:
    Dict[str, str]: NPC名称到对话内容的映射
"""
# 构建批量生成提示词
prompt = self._build_batch_prompt(context)

# 一次LLM调用生成所有对话
response = self.llm.invoke([
    {"role": "system", "content": "你是一个游戏NPC对话生成器，擅长创作自然真实的办公室对话。"},
    {"role": "user", "content": prompt}
])

# 解析JSON响应
dialogues = json.loads(response)
# 返回格式: {"张三": "...", "李四": "...", "王五": "..."}

return dialogues

def _build_batch_prompt(self, context: Optional[str] = None) -> str:
    """构建批量生成提示词"""
    # 根据时间自动推断场景
    if context is None:
        context = self._get_current_context()

    # 构建NPC描述
    npc_descriptions = []
    for name, cfg in self.npc_configs.items():
        desc = f"- {name}({cfg['title']}): 在{cfg['location']}{cfg['activity']}, 性格{cfg['personality']}"
        npc_descriptions.append(desc)

    npc_desc_text = "\n".join(npc_descriptions)

    prompt = f"""请为Datawhale办公室的3个NPC生成当前的对话或行为描述。
【场景】{context}

【NPC信息】
{npc_desc_text}

【生成要求】
1. 每个NPC生成1句话(20-40字)
2. 内容要符合角色设定、当前活动和场景氛围
3. 可以是自言自语、工作状态描述、或简单的思考
4. 要自然真实,像真实的办公室同事
5. **必须严格按照JSON格式返回**"""

    return prompt

```

【场景】{context}

【NPC信息】
{npc_desc_text}

【生成要求】

1. 每个NPC生成1句话(20-40字)
2. 内容要符合角色设定、当前活动和场景氛围
3. 可以是自言自语、工作状态描述、或简单的思考
4. 要自然真实,像真实的办公室同事
5. **必须严格按照JSON格式返回**

【输出格式】(严格遵守)

{{"张三": "...", "李四": "...", "王五": "..."}}

【示例输出】

{{"张三": "这个bug真是见鬼了,已经调试两小时了...", "李四": "嗯,这个功能的优先级需要重新评估一下。", "王五": "这杯咖啡的拉花真不错,灵感来了!"}}

请生成(只返回JSON,不要其他内容):

"""
 return prompt

这个设计的关键在于提示词的构建。我们明确要求 LLM 返回 JSON 格式，并提供了示例输出。LLM 会严格按照这个格式生成回复，我们只需要解析 JSON 就能获得所有 NPC 的对话。

批量生成还有一个额外的好处：所有 NPC 的对话是在同一个上下文中生成的，因此它们之间会有一定的关联性。比如，如果张三在调试 bug，李四可能会提到要帮忙看看；如果王五在设计界面，张三可能会说等会儿去看看设计稿。这让整个办公室的氛围更加真实和连贯。

当然，批量生成也有一些限制。它更适合生成 NPC 的“背景对话”或“自言自语”，而不是与玩家的直接互动。对于玩家发起的对话，我们仍然使用单独的 Agent 来处理，以保证回复的个性化和准确性。批量生成主要用于以下场景：

1. **NPC 背景对话**: 玩家进入场景时，NPC 正在做什么、说什么
2. 定时更新: 每隔一段时间更新 NPC 的状态和对话
3. 场景氛围: 根据时间(早上、中午、晚上)生成不同的对话
4. 降低成本: 在高并发场景下，使用批量生成降低 API 调用次数

混合模式: 批量生成+即时响应

在实际实现中，我们采用了一种混合模式，将批量生成和即时响应结合起来。这个设计非常巧妙，既保证了效率，又保证了交互的质量。

具体来说，系统会在后台定期运行批量生成，为所有 NPC 生成当前场景下的“背景对话”。这些对话会被缓存起来，当玩家靠近 NPC 但还没有发起交互时，NPC 会显示这些背景对话，比如“正在调试代码...”、“在看产品文档...”等。这让 NPC 看起来是“活着的”，而不是静止的模型。

但是，当玩家按下 E 键发起交互时，系统会立即切换到即时响应模式。此时，后端会调用该 NPC 的专属 Agent，根据玩家的具体消息、历史记忆和好感度，生成个性化的回复。这个过程是实时的，确保 NPC 的回复与玩家的输入高度相关。

```

# 在main.py中的混合模式实现
@app.post("/dialogue")
async def dialogue(request: DialogueRequest):

```

```

"""
处理玩家与NPC的对话(即时响应模式)
"""

npc_id = request.npc_id
player_message = request.player_message
player_name = request.player_name

# 获取NPC Agent(每个NPC有独立的Agent)
agent = npc_agents.get(npc_id)
if not agent:
    raise HTTPException(status_code=404, detail="NPC not found")

# 即时生成个性化回复
# 这里不使用批量生成,而是调用Agent的run方法
reply = agent.run(player_message)

# 更新好感度
affinity_change = relationship_manager.update_affinity(
    npc_id, player_name, player_message, reply
)

return {
    "npc_reply": reply,
    "affinity_score": affinity_change["score"],
    "affinity_level": affinity_change["level"]
}

# 后台任务:定期批量生成背景对话
async def background_dialogue_update():
    """
    后台任务:每5分钟更新一次NPC背景对话
    """
    while True:
        try:
            # 使用批量生成器生成所有NPC的背景对话
            batch_generator = get_batch_generator()
            dialogues = batch_generator.generate_batch_dialogues()

            # 更新到状态管理器
            for npc_name, dialogue in dialogues.items():
                state_manager.update_npc_background_dialogue(npc_name, dialogue)

            print(f"✅ 背景对话更新完成: {len(dialogues)}个NPC")
        except Exception as e:
            print(f"🔴 背景对话更新失败: {e}")

        # 等待5分钟
        await asyncio.sleep(300)

```

这种混合模式的优势非常明显:

- 降低成本: 背景对话使用批量生成, 一次调用生成所有 NPC 的对话, 成本低
- 保证质量: 玩家交互使用即时响应, 每个回复都是个性化的, 质量高
- 提升体验: NPC 始终有“背景对话”, 看起来很生动; 玩家交互时回复准确, 体验好
- 灵活调整: 可以根据服务器负载动态调整批量生成的频率

通过批量生成和即时响应的结合, 我们实现了一个既高效又智能的 NPC 系统。在正常情况下, 玩家感受不到任何差异, 但后端的成本和性能得到了显著优化。这个设计思路也可以应用到其他需要大量 AI 调用的场景中。

15.3 好感度系统设计

15.3.1 好感度等级划分

在赛博小镇中, NPC 对玩家的态度会随着互动而变化。我们设计了一个五级好感度系统, 从陌生到挚友, 每个等级都有不同的分数范围和对应的行为表现。

好感度系统的核心思想是: 通过量化 NPC 与玩家的关系, 让 NPC 的回复更加真实和有层次感。当玩家刚进入游戏时, 所有 NPC 对玩家都是陌生的态度, 回复比较礼貌但疏远。随着对话的进行, 如果玩家表现友好, NPC 的好感度会逐渐提升, 回复也会变得更加亲切和详细。

我们将好感度分为五个等级, 每个等级对应一个分数范围, 如图 15.8 所示:



图 15.8 好感度等级划分

- 陌生(0-20 分):** NPC 刚认识玩家, 态度礼貌但保持距离。回复简短, 不会主动分享个人信息。
- 熟悉(21-40 分):** NPC 开始记住玩家, 愿意进行简单的交流。回复变得更加自然, 偶尔会分享一些工作相关的信息。
- 友好(41-60 分):** NPC 把玩家当作朋友, 愿意分享更多信息。回复更加详细, 会主动询问玩家的情况。
- 亲密(61-80 分):** NPC 非常信任玩家, 愿意分享私人话题。回复充满热情, 会给玩家提供帮助和建议。

- 挚友(81-100 分): NPC 把玩家当作最好的朋友, 无话不谈。回复非常亲切, 会分享内心的想法和感受。

这个设计让玩家能够清晰地感受到与 NPC 关系的变化, 也为后续的游戏玩法提供了基础。比如, 只有达到一定好感度, NPC 才会分享某些特殊信息或提供特殊任务。

15.3.2 好感度计算逻辑

好感度的计算需要考虑多个因素。我们不能简单地让每次对话都增加固定的分数, 这样会让系统显得机械和不真实。一个好的好感度系统应该能够识别玩家的态度, 并根据对话内容动态调整分数。

在赛博小镇中, 我们使用 LLM 来分析对话内容, 判断玩家的态度是友好、中立还是不友好。然后根据判断结果调整好感度分数。这个过程是自动的, 不需要玩家刻意选择选项, 让互动更加自然。

好感度计算流程如图 15.9 所示:

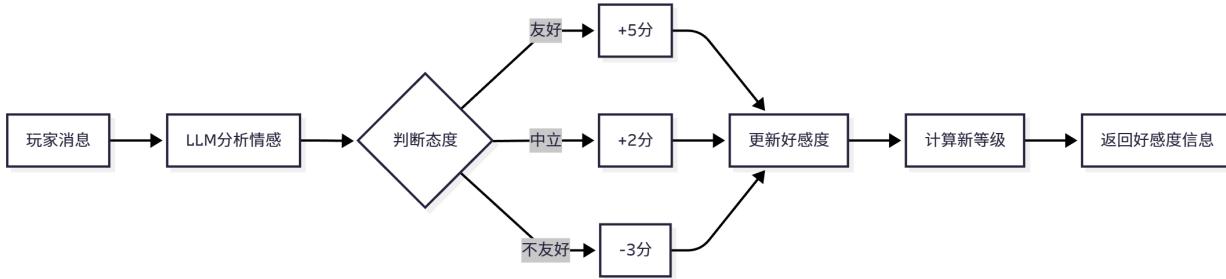


图 15.9 好感度计算流程

```

class RelationshipManager:
    """好感度管理器"""

    def __init__(self):
        self.affinity_data = {} # 存储好感度数据
        self.llm = HelloAgentsLLM() # 用于分析对话

    def analyze_sentiment(self, player_message: str, npc_reply: str) -> int:
        """分析对话情感, 返回好感度变化值"""
        prompt = f"""分析以下对话中玩家的态度:
玩家: {player_message}
NPC: {npc_reply}

请判断玩家的态度是:
1. 友好(+5分): 礼貌、热情、表示感谢或赞同
2. 中立(+2分): 普通的询问或陈述
3. 不友好(-3分): 粗鲁、冷漠、批评或否定

只返回数字, 不要其他内容。"""
  
```

```

        response = self.llm.think([{"role": "user", "content": prompt}])
        try:
            score_change = int(response.strip())
            return max(-3, min(5, score_change)) # 限制在-3到5之间
        except:
            return 2 # 默认中立
  
```

```

    def update_affinity(self, npc_id: str, player_name: str,
                        player_message: str, npc_reply: str) -> dict:
        """更新好感度"""
        key = f"{npc_id}_{player_name}"

        # 获取当前好感度
        if key not in self.affinity_data:
            self.affinity_data[key] = {
                "score": 0,
                "level": "陌生",
                "interaction_count": 0
            }

        # 分析对话情感
        score_change = self.analyze_sentiment(player_message, npc_reply)

        # 更新分数
        current_score = self.affinity_data[key]["score"]
        new_score = max(0, min(100, current_score + score_change))

        # 更新等级
        level = self.get_affinity_level(new_score)

        # 更新数据
        self.affinity_data[key].update({
            "score": new_score,
            "level": level,
            "interaction_count": self.affinity_data[key]["interaction_count"] + 1
  
```

2/9/26, 1:56 PM

```
    })
    return self.affinity_data[key]

def get_affinity_level(self, score: int) -> str:
    """根据分数获取好感度等级"""
    if score <= 20:
        return "陌生"
    elif score <= 40:
        return "熟悉"
    elif score <= 60:
        return "友好"
    elif score <= 80:
        return "亲密"
    else:
        return "挚友"
```

这个实现使用 LLM 来分析对话内容，自动判断玩家的态度并调整好感度。这样的设计让好感度系统更加智能和自然，玩家不需要刻意讨好 NPC，只需要正常交流即可。

15.3.3 好感度影响对话

好感度不仅仅是一个数字，它应该真正影响 NPC 的行为。在赛博小镇中，我们通过修改 NPC 的系统提示词，让 NPC 根据当前的好感度等级调整回复风格。

当好感度较低时，NPC 会保持礼貌但疏远的态度。当好感度提升后，NPC 会变得更加热情和健谈。这种变化是通过动态调整系统提示词实现的。

```
def create_npc_agent_with_affinity(npc_id: str, name: str, role: str,
                                    personality: str, affinity_level: str):
    """创建带好感度的NPC Agent"""

    # 根据好感度等级调整提示词
    affinity_prompts = {
        "陌生": "你刚认识这位玩家，保持礼貌但不要过于热情。回复简短专业。",
        "熟悉": "你已经认识这位玩家，可以进行正常的交流。回复自然友好。",
        "友好": "你把这位玩家当作朋友，愿意分享更多信息。回复详细热情。",
        "亲密": "你非常信任这位玩家，可以分享私人话题。回复充满关心。",
        "挚友": "你把这位玩家当作最好的朋友，无话不谈。回复亲切真诚。"
    }

    system_prompt = f"""你是{name}，一位{role}。
你的性格特点:{personality}

当前与玩家的关系:{affinity_level}
{affinity_prompts.get(affinity_level, affinity_prompts["陌生"])}"""

    你在Datawhale办公室工作，与同事们一起推动开源社区的发展。
    请根据你的角色、性格和与玩家的关系，自然地回复。
    """

    # 创建Agent
    llm = HelloAgentsLLM()
    agent = SimpleAgent(
        name=name,
        llm=llm,
        system_prompt=system_prompt
    )

    return agent
```

这个设计让 NPC 的行为随着好感度动态变化。玩家可以明显感受到，随着互动的增加，NPC 对自己的态度在逐渐改变，这大大增强了游戏的沉浸感和趣味性。

15.4 后端服务实现

15.4.1 FastAPI 应用结构

赛博小镇的后端使用 FastAPI 框架构建，负责处理 Godot 前端的请求，调用 HelloAgents 的 NPC Agent，管理 NPC 状态和好感度，以及记录日志。一个清晰的应用结构能够让代码更易于维护和扩展。

我们的 FastAPI 应用采用模块化设计，将不同的功能分离到不同的文件中，如图 15.10 所示：

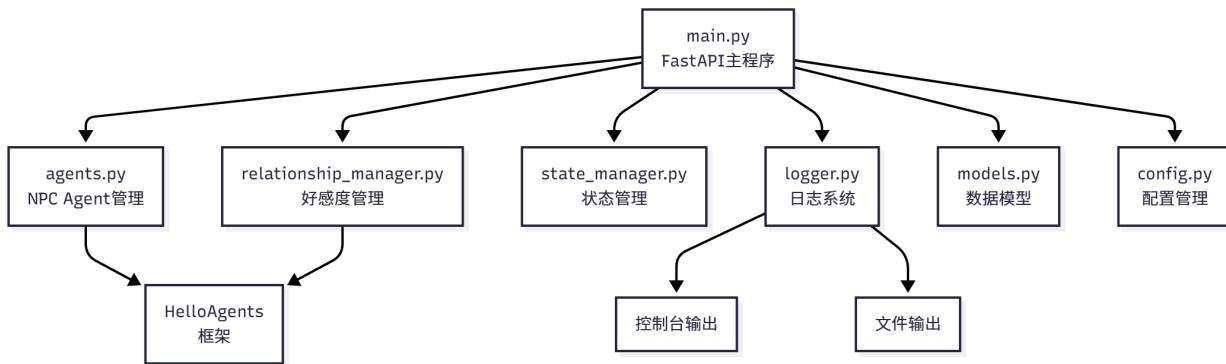


图 15.10 后端应用结构

让我们从 `main.py` 开始，这是 FastAPI 应用的入口文件：

```

from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel, Field
from typing import Optional
import uvicorn

from agents import NPCAgentManager
from relationship_manager import RelationshipManager
from state_manager import StateManager
from logger import DialogueLogger
from config import settings

# 创建FastAPI应用
app = FastAPI(
    title="赛博小镇后端服务",
    description="基于HelloAgents的AI NPC对话系统",
    version="1.0.0"
)

# 配置CORS,允许Godot前端访问
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # 生产环境应该限制具体域名
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# 初始化各个管理器
agent_manager = NPCAgentManager()
relationship_manager = RelationshipManager()
state_manager = StateManager()
dialogue_logger = DialogueLogger()

@app.on_event("startup")
async def startup_event():
    """应用启动时的初始化"""
    print("=" * 60)
    print("🏃 赛博小镇后端服务启动中....")
    print("=" * 60)

    # 初始化NPC Agents
    agent_manager.initialize_npcs()
    print("✅ NPC Agents已初始化")

    # 初始化状态管理器
    state_manager.initialize_npcs()
    print("✅ 状态管理器已初始化")

@app.get("/")
async def root():
    """健康检查"""
    return {
        "status": "running",
        "message": "赛博小镇后端服务正在运行",
        "version": "1.0.0",
        "npcs": state_manager.get_npc_count()
    }

if __name__ == "__main__":
    uvicorn.run(
        app,
        host=settings.HOST,
        port=settings.PORT,
        log_level="info"
    )
  
```

这个主程序文件定义了 FastAPI 应用的基本结构，配置了 CORS 中间件以允许跨域请求，并在启动时初始化各个管理器。接下来我们将实现具体的 API 路由。

15.4.2 API 路由设计

赛博小镇的后端需要提供几个核心 API 端点，用于处理 Godot 前端的请求。我们将这些路由添加到 `main.py` 中。

获取 NPC 状态

这个 API 返回所有 NPC 的当前状态，包括位置、是否忙碌等信息：

```
from models import NPCStatusResponse

@app.get("/npcs/status", response_model=NPCStatusResponse)
async def get_npc_status():
    """获取所有NPC的状态"""
    npcs = state_manager.get_all_npc_states()
    return {"npcs": npcs}

@app.get("/npcs/{npc_id}/status")
async def get_single_npc_status(npc_id: str):
    """获取单个NPC的状态"""
    npc = state_manager.get_npc_state(npc_id)
    if not npc:
        raise HTTPException(status_code=404, detail=f"NPC {npc_id} 不存在")
    return npc
```

对话接口

这是最核心的 API，处理玩家与 NPC 的对话：

```
from models import DialogueRequest, DialogueResponse

@app.post("/dialogue", response_model=DialogueResponse)
async def dialogue(request: DialogueRequest):
    """处理玩家与NPC的对话"""
    # 1. 验证NPC是否存在
    if not agent_manager.has_npc(request.npc_id):
        raise HTTPException(status_code=404, detail=f"NPC {request.npc_id} 不存在")

    # 2. 检查NPC是否忙碌
    if state_manager.is_npc_busy(request.npc_id):
        raise HTTPException(status_code=409, detail=f"NPC {request.npc_id} 正在与其他玩家对话")

    # 3. 标记NPC为忙碌状态
    state_manager.set_npc_busy(request.npc_id, True)

    try:
        # 4. 获取当前好感度
        affinity_info = relationship_manager.get_affinity(
            request.npc_id,
            request.player_name
        )

        # 5. 调用Agent生成回复
        agent = agent_manager.get_agent(request.npc_id, affinity_info["level"])
        reply = agent.run(request.player_message)

        # 6. 更新好感度
        new_affinity = relationship_manager.update_affinity(
            request.npc_id,
            request.player_name,
            request.player_message,
            reply
        )

        # 7. 记录日志
        dialogue_logger.log_dialogue(
            npc_id=request.npc_id,
            player_name=request.player_name,
            player_message=request.player_message,
            npc_reply=reply,
            affinity_info=new_affinity
        )

        # 8. 返回回复
        return DialogueResponse(
            npc_reply=reply,
            affinity_level=new_affinity["level"],
            affinity_score=new_affinity["score"]
        )

    except Exception as e:
        dialogue_logger.log_error(f"对话处理失败: {str(e)}")
        raise HTTPException(status_code=500, detail=f"对话处理失败: {str(e)}")

    finally:
        # 9. 释放NPC状态
        state_manager.set_npc_busy(request.npc_id, False)
```

好感度查询

这个 API 允许查询玩家与 NPC 的好感度：

```
from models import AffinityInfo

@app.get("/affinity/{npc_id}/{player_name}", response_model=AffinityInfo)
async def get_affinity(npc_id: str, player_name: str):
    """获取玩家与NPC的好感度"""
    if not agent_manager.has_npc(npc_id):
        raise HTTPException(status_code=404, detail=f"NPC {npc_id} 不存在")

    affinity = relationship_manager.get_affinity(npc_id, player_name)
    return affinity
```

API 路由的调用流程如图 15.11 所示：

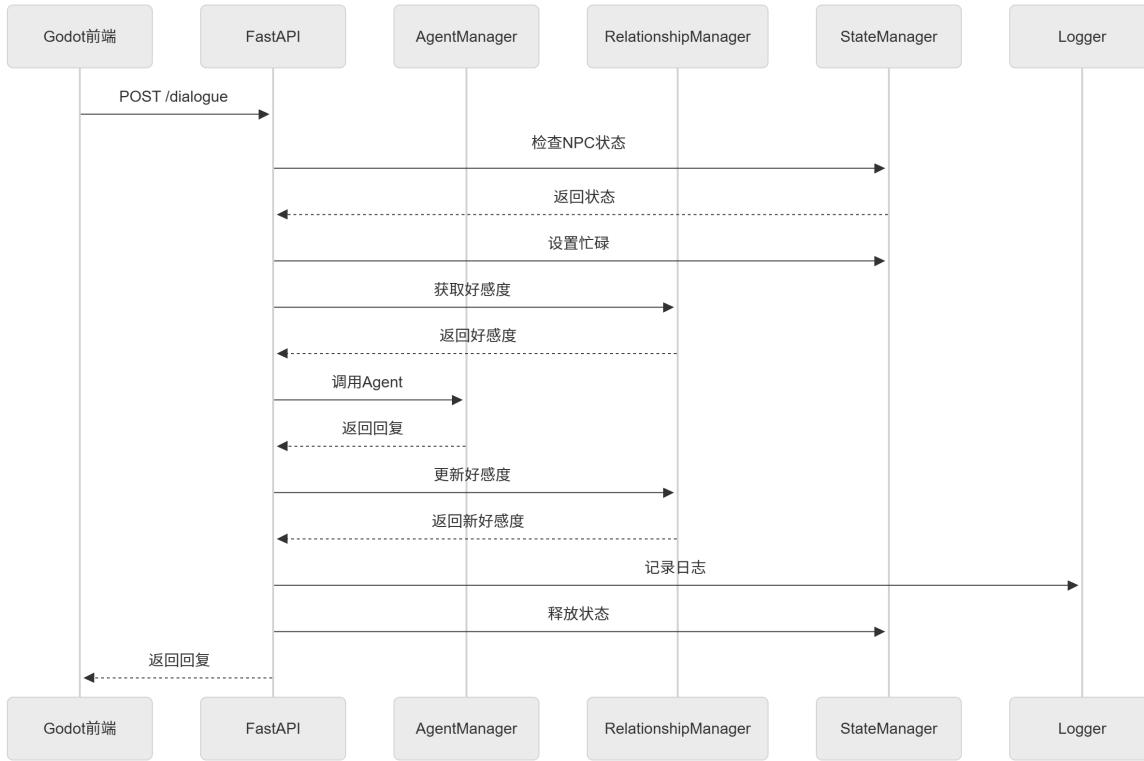


图 15.11 API 调用流程

15.4.3 状态管理与日志系统

状态管理器

状态管理器负责跟踪每个 NPC 的当前状态，包括位置、是否忙碌、当前动作等。这对于防止并发问题很重要，比如避免一个 NPC 同时与多个玩家对话。

```
# state_manager.py
from typing import Dict, List, Optional
from datetime import datetime

class StateManager:
    """NPC状态管理器"""

    def __init__(self):
        self.npc_states: Dict[str, dict] = {}

    def initialize_npcs(self):
        """初始化NPC状态"""
        npcs = [
            {
                "npc_id": "zhang_san",
                "name": "张三",
                "role": "Python工程师",
                "position": {"x": 300, "y": 200}
            },
            {
                "npc_id": "li_si",
                "name": "李四",
                "role": "产品经理",
                "position": {"x": 500, "y": 200}
            },
            {
                "npc_id": "wang_wu",
                "name": "王五",
                "role": "市场专员",
                "position": {"x": 300, "y": 400}
            }
        ]
```

```

        "role": "UI设计师",
        "position": {"x": 700, "y": 200}
    }
]

for npc in npcs:
    self.npc_states[npc["npc_id"]] = {
        **npc,
        "is_busy": False,
        "current_action": "idle",
        "last_interaction": None
    }

def get_npc_state(self, npc_id: str) -> Optional[dict]:
    """获取NPC状态"""
    return self.npc_states.get(npc_id)

def get_all_npc_states(self) -> List[dict]:
    """获取所有NPC状态"""
    return list(self.npc_states.values())

def is_npc_busy(self, npc_id: str) -> bool:
    """检查NPC是否忙碌"""
    npc = self.npc_states.get(npc_id)
    return npc["is_busy"] if npc else False

def set_npc_busy(self, npc_id: str, busy: bool):
    """设置NPC忙碌状态"""
    if npc_id in self.npc_states:
        self.npc_states[npc_id]["is_busy"] = busy
        if busy:
            self.npc_states[npc_id]["last_interaction"] = datetime.now().isoformat()

def get_npc_count(self) -> int:
    """获取NPC数量"""
    return len(self.npc_states)

```

日志系统

日志系统实现了双输出：控制台和文件。这样既方便实时查看，又能保存历史记录。

```

# logger.py
import logging
from datetime import datetime
from pathlib import Path

class DialogueLogger:
    """对话日志记录器"""

    def __init__(self, log_dir: str = "logs"):
        self.log_dir = Path(log_dir)
        self.log_dir.mkdir(exist_ok=True)

        # 创建日志文件名(按日期)
        today = datetime.now().strftime("%Y-%m-%d")
        log_file = self.log_dir / f"dialogue_{today}.log"

        # 配置日志
        self.logger = logging.getLogger("DialogueLogger")
        self.logger.setLevel(logging.INFO)

        # 控制台处理器
        console_handler = logging.StreamHandler()
        console_handler.setLevel(logging.INFO)
        console_formatter = logging.Formatter(
            '%(asctime)s - %(levelname)s - %(message)s',
            datefmt='%H:%M:%S'
        )
        console_handler.setFormatter(console_formatter)

        # 文件处理器
        file_handler = logging.FileHandler(log_file, encoding='utf-8')
        file_handler.setLevel(logging.INFO)
        file_formatter = logging.Formatter(
            '%(asctime)s - %(levelname)s - %(message)s',
            datefmt='%Y-%m-%d %H:%M:%S'
        )
        file_handler.setFormatter(file_formatter)

        # 添加处理器
        self.logger.addHandler(console_handler)
        self.logger.addHandler(file_handler)

    def log_dialogue(self, npc_id: str, player_name: str,
                    player_message: str, npc_reply: str,
                    affinity_info: dict):
        """记录对话"""
        log_message = f"""
{'='*60}
NPC: {npc_id}
玩家: {player_name}
玩家消息: {player_message}
NPC回复: {npc_reply}
"""

```

```

好感度: {affinity_info['level']} ({affinity_info['score']}/100)
互动次数: {affinity_info['interaction_count']}
{='*60'}
"""
    self.logger.info(log_message)

def log_error(self, error_message: str):
    """记录错误"""
    self.logger.error(error_message)

```

这个日志系统会在控制台实时显示对话内容，同时保存到文件中。每天的日志会保存在单独的文件中，方便后续分析。

15.4.4 理解 Godot 的场景系统

在开始构建游戏场景之前，我们需要先理解 Godot 的核心概念——场景(Scene)和节点(Node)。这是 Godot 与其他游戏引擎最大的不同之处，也是它最强大的特性之一。

什么是节点？

节点是 Godot 中最基本的构建块。你可以把节点想象成乐高积木，每个节点都有特定的功能。比如，Sprite2D 节点用于显示图片，AudioStreamPlayer 节点用于播放音频，CharacterBody2D 节点用于处理角色的物理移动。Godot 提供了上百种不同类型的节点，每种节点都专注于做好一件事。

节点之间可以形成父子关系，构成一个树状结构。父节点可以影响子节点，比如移动父节点会同时移动所有子节点，隐藏父节点会同时隐藏所有子节点。这种层级关系让我们可以轻松地组织和管理复杂的游戏对象。

什么是场景？

场景是一组节点的集合，保存在一个.tscn 文件中。你可以把场景理解为一个“预制件”。比如，我们可以创建一个“玩家”场景，包含角色的精灵、碰撞体、音效等所有相关节点。然后在游戏中多次使用这个场景，每次使用都会创建一个独立的实例。

场景的强大之处在于它的可复用性和模块化。我们可以在一个场景中实例化另一个场景，形成嵌套结构。比如，主场景可以包含玩家场景、多个 NPC 场景和 UI 场景。修改 NPC 场景会自动影响所有 NPC 实例，这大大简化了开发和维护。

一个简单的例子

让我们用一个简单的例子来理解场景和节点。假设我们要创建一个“玩家”场景：

```

Player (CharacterBody2D) ← 根节点, 负责物理移动
├─ AnimatedSprite2D      ← 子节点, 显示角色动画
├─ CollisionShape2D     ← 子节点, 定义碰撞形状
└─ Camera2D              ← 子节点, 摄像机跟随玩家

```

这个场景包含 4 个节点，形成树状结构。CharacterBody2D 是根节点，其他三个是它的子节点。我们可以给每个节点添加脚本来控制它的行为，也可以给根节点添加脚本来协调所有子节点。

当我们在主场景中实例化这个 Player 场景时，Godot 会创建这整个节点树的一个副本。我们可以创建多个玩家实例，每个实例都是独立的，有自己的位置、状态和行为。

场景实例化的优势

在赛博小镇中，我们有三个 NPC：张三、李四和王五。如果不使用场景系统，我们需要为每个 NPC 分别创建节点、设置属性、编写脚本，这会导致大量重复工作。而使用场景系统，我们只需要创建一个通用的 NPC 场景，然后实例化三次，通过脚本参数设置不同的名称和角色信息即可。

这种设计的好处是：如果我们想给所有 NPC 添加一个新功能(比如头顶显示对话气泡)，只需要修改 NPC 场景，所有实例都会自动获得这个功能。

15.5 Godot 游戏场景构建

为什么选择 Godot 作为游戏引擎？

在众多游戏引擎中，我们选择 Godot 4.5 作为前端引擎，主要基于以下几个考虑：

(1) Godot 在 2D 游戏开发上有着天然的优势。赛博小镇是一个俯视角的 2D 像素风格游戏，Godot 的 2D 引擎非常成熟，提供了 TileMap、AnimatedSprite2D、CharacterBody2D 等专门为 2D 游戏设计的节点类型，开发效率远高于 Unity 等引擎。Godot 的场景系统(Scene System)让我们可以将玩家、NPC、UI 等元素封装成独立的场景，然后在主场景中实例化，这种组件化的设计非常适合我们的需求。

(2) Godot 是完全开源且免费的。Godot 使用 MIT 许可证，没有任何版权费用或收入分成，这对于教学项目和开源项目非常友好。你可以自由地修改引擎源码，也可以将游戏商业化而不用担心授权问题。相比之下，Unity 虽然功能强大，但在 2024 年引入了运行时费用政策，引发了开发者社区的广泛争议。

(3) Godot 的学习成本极低。Godot 使用 GDScript 作为主要脚本语言，这是一种类似 Python 的动态类型语言，语法简洁易懂，学习曲线非常平缓。对于已经熟悉 Python 的读者来说，学习 GDScript 几乎没有门槛——变量声明、函数定义、控制流程等语法都与 Python 高度相似，你甚至可以在几小时内就上手编写游戏脚本。Godot 的节点树结构也非常直观，你可以在编辑器中直观地看到场景的层级关系，这对于初学者非常友好。

(4) Godot 与 Python 后端的集成非常简单。Godot 内置了 HTTPRequest 节点，可以轻松地与 FastAPI 后端进行 HTTP 通信。我们只需要创建一个 API 客户端脚本，封装所有的 API 调用，就可以在游戏中调用后端的 AI 能力。这种前后端分离的架构让我们可以独立开发和测试游戏逻辑和 AI 逻辑，大大提高了开发效率。

当然, Godot 也有一些局限性。比如, Godot 的 3D 能力相比 Unreal Engine 和 Unity 还有差距, 如果你要开发大型 3D 游戏, 可能需要考虑其他引擎。但对于 2D 游戏、独立游戏和教学项目, Godot 是一个非常优秀的选择。

15.5.1 场景设计与资源组织

理解了 Godot 的场景系统后, 我们来看看赛博小镇的场景设计。整个游戏由四个核心场景组成: Main(主场景)、Player(玩家)、NPC(非玩家角色)和 DialogueUI(对话界面)。每个场景都是一个独立的模块, 可以单独编辑和测试, 然后组合在一起形成完整的游戏。

赛博小镇的场景组织采用了模块化设计。我们首先创建三个基础场景: Player(玩家)、NPC(非玩家角色)和 DialogueUI(对话界面)。然后在 Main(主场景)中将这些场景实例化并组合起来。特别值得注意的是, 三个 NPC(张三、李四、王五)都是同一个 NPC 场景的实例, 只是通过脚本参数设置了不同的角色信息。

让我们先看看四个核心场景的结构, 如图 15.12 所示:

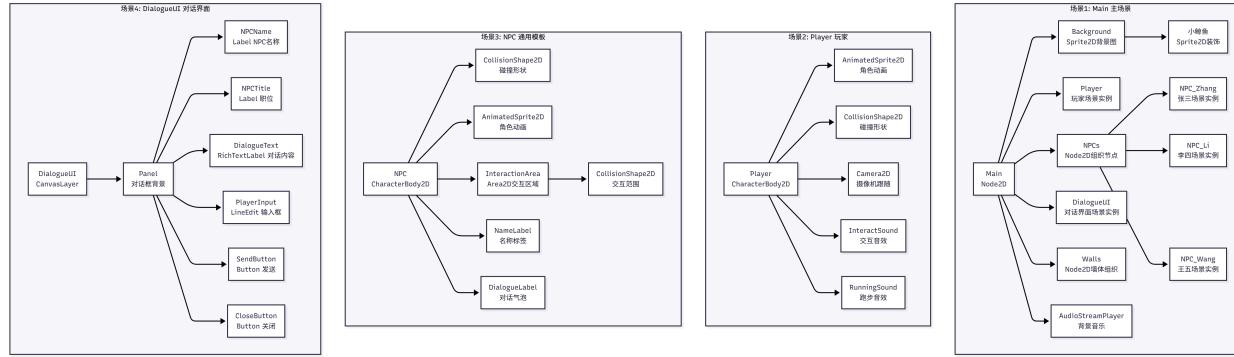


图 15.12 赛博小镇的四个核心场景

这个图展示了四个独立的场景及其内部结构。场景 1(Main)是主场景, 它包含了背景图片(Sprite2D)、玩家实例、NPCs 组织节点(下面有三个 NPC 实例)、对话界面实例、墙体组织节点和背景音乐。注意, 这里的 Player、NPC_Zhang、NPC_Li、NPC_Wang 和 DialogueUI 都是场景实例, 不是普通节点。场景 2(Player)定义了玩家角色的结构, 包含动画、碰撞、摄像机和两个音效节点。场景 3(NPC)是一个通用模板, 张三、李四、王五都是这个场景的实例, 包含碰撞、动画、交互区域和两个标签。场景 4(DialogueUI)是一个 CanvasLayer 节点, 包含 Panel 和各种 UI 元素。

场景实例化的过程可以这样理解: 我们在 Godot 编辑器中创建了 NPC.tscn 这个场景文件, 定义了 NPC 的节点结构。然后在 Main 场景中, 我们三次“实例化”这个 NPC 场景, 创建了三个独立的副本, 分别命名为 NPC_Zhang、NPC_Li 和 NPC_Wang。每个副本都有自己的位置和状态, 但它们共享相同的节点结构。如果我们修改 NPC.tscn, 比如给 NPC 添加一个新的音效节点, 那么所有三个实例都会自动获得这个音效。

在 Godot 中创建这些场景的步骤如下:

1. 创建 Player 场景: 新建场景, 选择 CharacterBody2D 作为根节点, 添加 AnimatedSprite2D、CollisionShape2D、Camera2D、InteractSound 和 RunningSound 子节点, 保存为 Player.tscn。
2. 创建 NPC 场景: 新建场景, 选择 CharacterBody2D 作为根节点, 添加 CollisionShape2D、AnimatedSprite2D、InteractionArea(Area2D, 下面有 CollisionShape2D)、NameLabel 和 DialogueLabel 子节点, 保存为 NPC.tscn。
3. 创建 DialogueUI 场景: 新建场景, 选择 CanvasLayer 作为根节点, 添加 Panel 子节点, 在 Panel 下添加 NPCName、NPCTitle、DialogueText(RichTextLabel)、PlayerInput(LineEdit)、SendButton 和 CloseButton, 保存为 DialogueUI.tscn。
4. 创建 Main 场景: 新建场景, 选择 Node2D 作为根节点, 添加 Background(Sprite2D)作为背景图, 在 Background 下添加小鲸鱼装饰, 然后实例化 Player 场景, 创建 NPCs 节点并在其下三次实例化 NPC 场景, 实例化 DialogueUI 场景, 创建 Walls 节点用于组织墙体碰撞, 最后添加 AudioStreamPlayer 播放背景音乐。

这种场景组织方式的优势在于: 每个场景都是独立的, 可以单独测试; NPC 使用同一个场景的实例, 修改一次就能影响所有 NPC; 场景之间通过信号通信, 耦合度低, 易于维护和扩展。

15.5.2 玩家控制实现

玩家角色是游戏中最重要的元素之一。我们需要实现 WASD 移动控制、动画切换、碰撞检测、与 NPC 的交互, 以及音效系统。

玩家场景的结构包括: 一个 CharacterBody2D 作为根节点, 负责物理移动和碰撞; 一个 AnimatedSprite2D 显示角色动画; 一个 CollisionShape2D 定义碰撞形状; 一个 Camera2D 跟随玩家; 两个 AudioStreamPlayer 分别播放交互音效和走路音效。

玩家控制脚本 player.gd 实现了移动、交互和音效逻辑:

```
extends CharacterBody2D

# 移动速度
@export var speed: float = 200.0

# 当前可交互的NPC
var nearby_npc: Node = null

# 交互状态(交互时禁用移动)
```

2/9/26, 1:56 PM

```
var is_interacting: bool = false

# 节点引用
@onready var animated_sprite: AnimatedSprite2D = $AnimatedSprite2D
@onready var camera: Camera2D = $Camera2D

# 音效引用
@onready var interact_sound: AudioStreamPlayer = null
@onready var running_sound: AudioStreamPlayer = null

# 走路音效状态
var is_playing_running_sound: bool = false

func _ready():
    # 添加到player组(重要!NPC需要通过这个组来识别玩家)
    add_to_group("player")

    # 获取音效节点(可选,如果不存在也不会报错)
    interact_sound = get_node_or_null("InteractSound")
    running_sound = get_node_or_null("RunningSound")

    # 启用相机
    camera.enabled = true

    # 播放默认动画
    if animated_sprite.sprite_frames != null and animated_sprite.sprite_frames.has_animation("idle"):
        animated_sprite.play("idle")

func _physics_process(_delta: float):
    # 如果正在交互,禁用移动
    if is_interacting:
        velocity = Vector2.ZERO
        move_and_slide()
        # 播放idle动画
        if animated_sprite.sprite_frames != null and animated_sprite.sprite_frames.has_animation("idle"):
            animated_sprite.play("idle")
        # 停止走路音效
        stop_running_sound()
    return

    # 获取输入方向
    var input_direction = Input.get_vector("ui_left", "ui_right", "ui_up", "ui_down")

    # 设置速度
    velocity = input_direction * speed

    # 移动
    move_and_slide()

    # 更新动画和朝向
    update_animation(input_direction)

    # 更新走路音效
    update_running_sound(input_direction)

func update_animation(direction: Vector2):
    """更新角色动画(支持4方向)"""
    if animated_sprite.sprite_frames == null:
        return

    # 根据移动方向播放动画
    if direction.length() > 0:
        # 移动中 - 判断主要方向
        if abs(direction.x) > abs(direction.y):
            # 左右移动
            if direction.x > 0:
                # 向右
                if animated_sprite.sprite_frames.has_animation("walk_right"):
                    animated_sprite.play("walk_right")
                    animated_sprite.flip_h = false
                elif animated_sprite.sprite_frames.has_animation("walk"):
                    animated_sprite.play("walk")
                    animated_sprite.flip_h = false
            else:
                # 向左
                if animated_sprite.sprite_frames.has_animation("walk_left"):
                    animated_sprite.play("walk_left")
                    animated_sprite.flip_h = false
                elif animated_sprite.sprite_frames.has_animation("walk"):
                    animated_sprite.play("walk")
                    animated_sprite.flip_h = true
        else:
            # 上下移动
            if direction.y > 0:
                # 向下
                if animated_sprite.sprite_frames.has_animation("walk_down"):
                    animated_sprite.play("walk_down")
                elif animated_sprite.sprite_frames.has_animation("walk"):
                    animated_sprite.play("walk")
            else:
                # 向上
                if animated_sprite.sprite_frames.has_animation("walk_up"):
                    animated_sprite.play("walk_up")
```

```

        elif animated_sprite.sprite_frames.has_animation("walk"):
            animated_sprite.play("walk")
    else:
        # 静止
        if animated_sprite.sprite_frames.has_animation("idle"):
            animated_sprite.play("idle")

func _input(event: InputEvent):
    # 按E键与NPC交互
    if event is InputEventKey:
        if event.pressed and not event.echo:
            if event.keycode == KEY_E or event.keycode == KEY_ENTER:
                if nearby_npc != null:
                    interact_with_npc()

func interact_with_npc():
    """与附近的NPC交互"""
    if nearby_npc != null:
        # 播放交互音效
        if interact_sound:
            interact_sound.play()

    # 发送信号给对话系统
    get_tree().call_group("dialogue_system", "start_dialogue", nearby_npc.npc_name)

func set_nearby_npc(npc: Node):
    """设置附近的NPC"""
    nearby_npc = npc

func set_interacting(interacting: bool):
    """设置交互状态"""
    is_interacting = interacting
    if interacting:
        # 停止走路音效
        stop_running_sound()

func update_running_sound(direction: Vector2):
    """更新走路音效"""
    if running_sound == null:
        return

    # 如果正在移动
    if direction.length() > 0:
        # 如果音效还没播放,开始播放
        if not is_playing_running_sound:
            running_sound.play()
            is_playing_running_sound = true
    else:
        # 如果停止移动,停止音效
        stop_running_sound()

func stop_running_sound():
    """停止走路音效"""
    if running_sound and is_playing_running_sound:
        running_sound.stop()
        is_playing_running_sound = false

```

这个脚本实现了完整的玩家控制。玩家使用 WASD 键(或方向键)移动，角色会根据移动方向播放相应的 4 方向动画(walk_up/down/left/right)。当玩家靠近 NPC 时，NPC 会调用 set_nearby_npc() 设置自己为可交互对象，玩家按 E 键即可触发交互。交互时会播放音效，并通过 call_group() 通知对话系统开始对话。对话期间，set_interacting(true) 会禁用玩家移动，对话结束后恢复移动。走路音效会在玩家移动时自动播放，停止时自动停止。

15.5.3 NPC 行为与交互

NPC 需要实现三个核心功能：在场景中随机巡逻游走、响应玩家的交互、显示对话气泡。我们使用 Area2D 来检测玩家是否靠近 NPC，当玩家进入交互范围时通知玩家，玩家按 E 键即可开始对话。

NPC 场景的结构包括：CharacterBody2D 作为根节点;CollisionShape2D 定义 NPC 的碰撞形状;AnimatedSprite2D 显示 NPC 动画;InteractionArea/Area2D 检测玩家进入交互范围，下面有 CollisionShape2D 定义交互范围;NameLabel 显示 NPC 名字;DialogueLabel 显示对话气泡。

NPC 脚本 npc.gd 实现了巡逻、交互和对话气泡逻辑：

```

extends CharacterBody2D

# NPC信息
@export var npc_name: String = "张三"
@export var npc_title: String = "Python工程师"

# NPC外观配置
@export var sprite_frames: SpriteFrames = null # 自定义精灵帧资源

# NPC移动配置
@export var move_speed: float = 50.0 # 移动速度
@export var wander_enabled: bool = true # 是否启用巡逻
@export var wander_range: float = 200.0 # 巡逻范围
@export var wander_interval_min: float = 3.0 # 最小巡逻间隔(秒)
@export var wander_interval_max: float = 8.0 # 最大巡逻间隔(秒)

```

```

# 当前对话内容(从后端获取)
var current_dialogue: String = ""

# 节点引用
@onready var animated_sprite: AnimatedSprite2D = $AnimatedSprite2D
@onready var interaction_area: Area2D = $InteractionArea
@onready var name_label: Label = $NameLabel
@onready var dialogue_label: Label = $DialogueLabel

# 玩家引用
var player: Node = null

# 巡逻相关变量
var wander_target: Vector2 = Vector2.ZERO # 巡逻目标位置
var wander_timer: float = 0.0 # 巡逻计时器
var is_wandering: bool = false # 是否正在巡逻
var is_interacting: bool = false # 是否正在与玩家交互
var spawn_position: Vector2 = Vector2.ZERO # 出生位置

func _ready():
    # 添加到npcs组
    add_to_group("npcs")

    # 设置NPC名字
    name_label.text = npc_name

    # 连接交互区域信号
    interaction_area.body_entered.connect(_on_body_entered)
    interaction_area.body_exited.connect(_on_body_exited)

    # 初始化对话标签
    dialogue_label.text = ""
    dialogue_label.visible = false

    # 设置自定义精灵帧(如果有)
    if sprite_frames != null:
        animated_sprite.sprite_frames = sprite_frames

    # 播放默认动画
    if animated_sprite.sprite_frames != null and animated_sprite.sprite_frames.has_animation("idle"):
        animated_sprite.play("idle")

    # 记录出生位置
    spawn_position = global_position

    # 初始化巡逻计时器
    if wander_enabled:
        wander_timer = randf_range(wander_interval_min, wander_interval_max)
        choose_new_wander_target()

func _on_body_entered(body: Node2D):
    """玩家进入交互范围"""
    if body.is_in_group("player"):
        player = body

        if player.has_method("set_nearby_npc"):
            player.set_nearby_npc(self)

func _on_body_exited(body: Node2D):
    """玩家离开交互范围"""
    if body.is_in_group("player"):
        if player != null and player.has_method("set_nearby_npc"):
            player.set_nearby_npc(null)
        player = null

func update_dialogue(dialogue: String):
    """更新NPC对话内容"""
    current_dialogue = dialogue
    dialogue_label.text = dialogue
    dialogue_label.visible = true

    # 10秒后隐藏对话
    await get_tree().create_timer(10.0).timeout
    dialogue_label.visible = false

func _physics_process(delta: float):
    """物理更新 - 处理移动"""
    # 如果正在与玩家交互, 停止移动
    if is_interacting:
        velocity = Vector2.ZERO
        move_and_slide()
        # 播放idle动画
        if animated_sprite.sprite_frames != null and animated_sprite.sprite_frames.has_animation("idle"):
            animated_sprite.play("idle")
        return

    # 如果未启用巡逻, 不移动
    if not wander_enabled:
        return

    # 更新巡逻计时器
    wander_timer -= delta

```

```

# 如果计时器结束,选择新目标并开始移动
if wander_timer <= 0:
    choose_new_wander_target()
    wander_timer = randf_range(wander_interval_min, wander_interval_max)

# 如果正在巡逻,移动到目标
if is_wandering:
    # 检查是否到达目标
    if global_position.distance_to(wander_target) < 10:
        # 到达目标,停止移动
        is_wandering = false
        velocity = Vector2.ZERO
        move_and_slide()
        # 播放idle动画
        if animated_sprite.sprite_frames != null and animated_sprite.sprite_frames.has_animation("idle"):
            animated_sprite.play("idle")
    else:
        # 继续移动到目标
        var direction = (wander_target - global_position).normalized()
        velocity = direction * move_speed
        move_and_slide()
        # 更新动画
        update_animation(direction)
else:
    # 停止移动
    velocity = Vector2.ZERO
    move_and_slide()
    # 播放idle动画
    if animated_sprite.sprite_frames != null and animated_sprite.sprite_frames.has_animation("idle"):
        animated_sprite.play("idle")

func choose_new_wander_target():
    """选择新的巡逻目标"""
    # 在出生位置附近随机选择一个点
    var offset = Vector2(
        randf_range(-wander_range, wander_range),
        randf_range(-wander_range, wander_range)
    )
    wander_target = spawn_position + offset
    is_wandering = true

func update_animation(direction: Vector2):
    """更新动画"""
    if animated_sprite.sprite_frames == null:
        return

    if direction.length() > 0:
        # 移动动画
        if abs(direction.x) > abs(direction.y):
            # 左右移动
            if direction.x > 0:
                if animated_sprite.sprite_frames.has_animation("walk_right"):
                    animated_sprite.play("walk_right")
                elif animated_sprite.sprite_frames.has_animation("walk"):
                    animated_sprite.play("walk")
                    animated_sprite.flip_h = false
            else:
                if animated_sprite.sprite_frames.has_animation("walk_left"):
                    animated_sprite.play("walk_left")
                elif animated_sprite.sprite_frames.has_animation("walk"):
                    animated_sprite.play("walk")
                    animated_sprite.flip_h = true
        else:
            # 上下移动
            if direction.y > 0:
                if animated_sprite.sprite_frames.has_animation("walk_down"):
                    animated_sprite.play("walk_down")
                elif animated_sprite.sprite_frames.has_animation("walk"):
                    animated_sprite.play("walk")
            else:
                if animated_sprite.sprite_frames.has_animation("walk_up"):
                    animated_sprite.play("walk_up")
                elif animated_sprite.sprite_frames.has_animation("walk"):
                    animated_sprite.play("walk")
    else:
        # 静止动画
        if animated_sprite.sprite_frames.has_animation("idle"):
            animated_sprite.play("idle")

func set_interacting(interacting: bool):
    """设置交互状态"""
    is_interacting = interacting

```

这个脚本实现了 NPC 的完整行为。NPC 会在出生位置附近的wander_range范围内随机巡逻，每隔wander_interval_min到wander_interval_max秒选择一个新的目标点并移动过去。移动时会播放 4 方向动画(walk_up/down/left/right)，到达目标后停止并播放 idle 动画。当玩家进入 InteractionArea 时，NPC 会调用玩家的set_nearby_npc(self)方法，将自己设置为可交互对象。玩家按 E 键后，对话系统会调用 NPC 的set_interacting(true)方法，NPC 停止移动。对话结束后调用set_interacting(false)，NPC 恢复巡逻。主场景会定时调用update_dialogue()方法更新 NPC 的对话气泡，显示 NPC 之间的自主对话内容。

15.6 前后端通信实现

15.6.1 API 客户端封装

Godot 前端需要与 FastAPI 后端进行 HTTP 通信。我们创建一个 API 客户端脚本 `api_client.gd`, 封装所有的 API 调用, 并将其设置为 AutoLoad(自动加载)单例, 让其他脚本可以方便地使用。

API 客户端使用 Godot 的 `HTTPRequest` 节点来发送 HTTP 请求。`HTTPRequest` 是一个异步节点, 发送请求后不会阻塞游戏, 而是通过信号通知请求完成。这样可以保证游戏的流畅性, 即使网络延迟较高也不会卡顿。我们使用信号机制来通知其他脚本 API 响应, 而不是使用 `await`, 这样可以让多个脚本同时监听同一个 API 响应。

```
# api_client.gd
extends Node

# 信号定义
signal chat_response_received(npc_name: String, message: String)
signal chat_error(error_message: String)
signal npc_status_received(dialogues: Dictionary)
signal npc_list_received(npcs: Array)

# HTTP请求节点
var http_chat: HTTPRequest
var http_status: HTTPRequest
var http_npcs: HTTPRequest

func _ready():
    # 创建HTTP请求节点
    http_chat = HTTPRequest.new()
    http_status = HTTPRequest.new()
    http_npcs = HTTPRequest.new()

    add_child(http_chat)
    add_child(http_status)
    add_child(http_npcs)

    # 连接信号
    http_chat.request_completed.connect(_on_chat_request_completed)
    http_status.request_completed.connect(_on_status_request_completed)
    http_npcs.request_completed.connect(_on_npcs_request_completed)

# ===== 对话API =====
func send_chat(npc_name: String, message: String) -> void:
    """发送对话请求"""
    var data = {
        "npc_name": npc_name,
        "message": message
    }

    var json_string = JSON.stringify(data)
    var headers = ["Content-Type: application/json"]

    var error = http_chat.request(
        Config.API_CHAT,
        headers,
        HTTPClient.METHOD_POST,
        json_string
    )

    if error != OK:
        print("[ERROR] 发送对话请求失败: ", error)
        chat_error.emit("网络请求失败")

func _on_chat_request_completed(_result: int, response_code: int, _headers: PackedStringArray, body: PackedByteArray) -> void:
    """处理对话响应"""
    if response_code != 200:
        print("[ERROR] 对话请求失败: HTTP ", response_code)
        chat_error.emit("服务器错误: " + str(response_code))
        return

    var json = JSON.new()
    var parse_result = json.parse(body.get_string_from_utf8())

    if parse_result != OK:
        print("[ERROR] 解析响应失败")
        chat_error.emit("响应解析失败")
        return

    var response = json.data

    if response.has("success") and response["success"]:
        var npc_name = response["npc_name"]
        var msg = response["message"]
        print("[INFO] 收到NPC回复: ", npc_name, " -> ", msg)
        chat_response_received.emit(npc_name, msg)
    else:
        chat_error.emit("对话失败")

# ===== NPC状态API =====
func get_npc_status() -> void:
    """获取NPC状态"""
    # 检查是否正在处理请求
    if http_status.get_http_client_status() != HTTPClient.STATUS_DISCONNECTED:
```

```

print("[WARN] NPC状态请求正在处理中,跳过本次请求")
return

var error = http_status.request(Config.API_NPC_STATUS)

if error != OK:
    print("[ERROR] 获取NPC状态失败: ", error)

func _on_status_request_completed(_result: int, response_code: int, _headers: PackedStringArray, body: PackedByteArray) -> void:
    """处理NPC状态响应"""
    if response_code != 200:
        print("[ERROR] NPC状态请求失败: HTTP ", response_code)
        return

    var json = JSON.new()
    var parse_result = json.parse(body.get_string_from_utf8())

    if parse_result != OK:
        print("[ERROR] 解析NPC状态失败")
        return

    var response = json.data

    if response.has("dialogues"):
        var dialogues = response["dialogues"]
        print("[INFO] 收到NPC状态更新: ", dialogues.size(), "个NPC")
        npc_status_received.emit(dialogues)

# ===== NPC列表API =====
func get_npc_list() -> void:
    """获取NPC列表"""
    var error = http_npcs.request(Config.API_NPCS)

    if error != OK:
        print("[ERROR] 获取NPC列表失败: ", error)

func _on_npcs_request_completed(_result: int, response_code: int, _headers: PackedStringArray, body: PackedByteArray) -> void:
    """处理NPC列表响应"""
    if response_code != 200:
        print("[ERROR] NPC列表请求失败: HTTP ", response_code)
        return

    var json = JSON.new()
    var parse_result = json.parse(body.get_string_from_utf8())

    if parse_result != OK:
        print("[ERROR] 解析NPC列表失败")
        return

    var response = json.data

    if response.has("npcs"):
        var npcs = response["npcs"]
        print("[INFO] 收到NPC列表: ", npcs.size(), "个NPC")
        npc_list_received.emit(npcs)

```

这个 API 客户端封装了三个核心功能：发送对话请求(send_chat)、获取 NPC 状态(get_npc_status)和获取 NPC 列表(get_npc_list)。所有的 HTTP 请求都是异步的，通过信号通知响应结果。我们为每个 API 创建了独立的 HTTPRequest 节点，这样可以同时发送多个请求而不会互相干扰。API 的 URL 从 Config 单例中获取，方便统一管理。对话系统监听chat_response_received信号来接收 NPC 回复，主场景监听npc_status_received信号来更新 NPC 对话气泡。

15.6.2 对话 UI 实现

对话 UI 是玩家与 NPC 交互的界面。我们需要设计一个简洁美观的对话框，包含 NPC 名称、职位、对话内容显示、输入框和按钮。

对话 UI 的结构如图 15.13 所示：

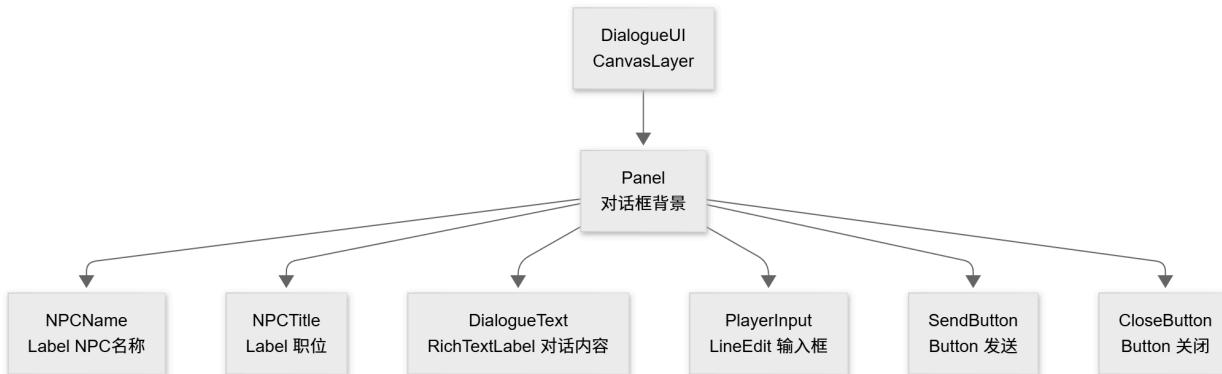


图 15.13 对话 UI 结构

对话 UI 的设计非常简洁。DialogueUI 是一个 CanvasLayer 节点，这意味着它会始终显示在游戏画面的最上层，不会被其他游戏对象遮挡。Panel 是对话框的背景，锚定在屏幕底部。Panel 下直接放置了 6 个 UI 元素：NPCName 显示 NPC 的名字，NPCTitle 显示职位，DialogueText 使用 RichTextLabel 显示对话内容（支持富文本格式），PlayerInput 是一个 LineEdit 用于玩家输入，SendButton 和 CloseButton 分别用于发送消息和关闭对话框。

对话 UI 脚本 dialogue_ui.gd 实现了对话界面的逻辑：

```
# dialogue_ui.gd
extends CanvasLayer

# UI节点引用
@onready var panel = $Panel
@onready var npc_name_label = $Panel/NPCName
@onready var npc_title_label = $Panel/NPCTitle
@onready var dialogue_text = $Panel/DialogueText
@onready var input_field = $Panel/PlayerInput
@onready var send_button = $Panel/SendButton
@onready var close_button = $Panel/CloseButton

# API客户端
var api_client: Node = null

# 当前对话的NPC
var current_npc_name: String = ""

func _ready():
    # 初始化时隐藏对话框
    visible = false

    # 连接按钮信号
    send_button.pressed.connect(_on_send_button_pressed)
    close_button.pressed.connect(_on_close_button_pressed)
    input_field.text_submitted.connect(_on_text_submitted)

    # 获取API客户端
    api_client = get_node_or_null("/root/APIClient")

func start_dialogue(npc_name: String):
    """开始与NPC对话"""
    current_npc_name = npc_name

    # 设置NPC信息
    npc_name_label.text = npc_name
    npc_title_label.text = get_npc_title(npc_name)

    # 清空对话内容
    dialogue_text.clear()
    dialogue_text.append_text("[color=gray]与 " + npc_name + " 的对话开始...[/color]\n")

    # 清空输入框
    input_field.text = ""

    # 显示对话框
    show_dialogue()

    # 聚焦输入框
    input_field.grab_focus()

func show_dialogue():
    """显示对话框"""
    visible = true

    # 通知玩家进入交互状态(禁用移动)
    var player = get_tree().get_first_node_in_group("player")
    if player and player.has_method("set_interacting"):
        player.set_interacting(true)

func hide_dialogue():
    """隐藏对话框"""
    visible = false
    current_npc_name = ""

    # 通知玩家退出交互状态(启用移动)
    var player = get_tree().get_first_node_in_group("player")
    if player and player.has_method("set_interacting"):
        player.set_interacting(false)

func _on_send_button_pressed():
    """发送按钮点击"""
    send_message()

func _on_close_button_pressed():
    """关闭按钮点击"""
    hide_dialogue()

func _on_text_submitted(_text: String):
    """输入框回车"""
    send_message()

func send_message():
    """发送消息"""
    var message = input_field.text.strip_edges()
```

```

if message.is_empty():
    return

if current_npc_name.is_empty():
    return

# 显示玩家消息
dialogue_text.append_text("\n[color=cyan]玩家:[/color] " + message + "\n")

# 清空输入框
input_field.text = ""

# 禁用输入
input_field.editable = false
send_button.disabled = true

# 发送API请求
if api_client:
    api_client.send_chat_request(current_npc_name, message)

func on_chat_response_received(npc_name: String, response: String):
    """收到NPC回复"""
    if npc_name == current_npc_name:
        # 显示NPC回复
        dialogue_text.append_text("[color=yellow]" + npc_name + "[/color] " + response + "\n")

        # 启用输入
        input_field.editable = true
        send_button.disabled = false
        input_field.grab_focus()

func get_npc_title(npc_name: String) -> String:
    """获取NPC职位"""
    var titles = {
        "张三": "Python工程师",
        "李四": "产品经理",
        "王五": "UI设计师"
    }
    return titles.get(npc_name, "")

```

这个对话 UI 实现了完整的对话功能。玩家可以输入消息并发送，UI 使用 RichTextLabel 的 append_text 方法显示对话内容，支持富文本格式(颜色、粗体等)。所有的 API 调用都是异步的，在等待响应时会禁用输入框，防止重复发送。对话框显示时会通知玩家进入交互状态，禁用移动，关闭时恢复移动。

15.6.3 主场景整合

最后，我们需要在主场景中整合所有的功能：玩家控制、NPC 交互、对话 UI 和 NPC 状态更新。主场景脚本 main.gd 负责协调这些组件，并定时从后端获取 NPC 状态，更新 NPC 的对话气泡。

```

# main.gd
extends Node2D

# NPC节点引用
@onready var npc_zhang: Node2D = $NPCs/NPC_Zhang
@onready var npc_li: Node2D = $NPCs/NPC_Li
@onready var npc_wang: Node2D = $NPCs/NPC_Wang

# API客户端
var api_client: Node = null

# NPC状态更新计时器
var status_update_timer: float = 0.0

func _ready():
    print("[INFO] 主场景初始化")

    # 获取API客户端
    api_client = get_node_or_null("/root/APIClient")
    if api_client:
        api_client.npc_status_received.connect(_on_npc_status_received)

        # 立即获取一次NPC状态
        api_client.get_npc_status()
    else:
        print("[ERROR] API客户端未找到")

func _process(delta: float):
    # 定时更新NPC状态
    status_update_timer += delta
    if status_update_timer >= Config.NPC_STATUS_UPDATE_INTERVAL:
        status_update_timer = 0.0
        if api_client:
            api_client.get_npc_status()

func _on_npc_status_received(dialogues: Dictionary):
    """收到NPC状态更新"""
    print("[INFO] 更新NPC状态: ", dialogues)

    # 更新各个NPC的对话
    for npc_name in dialogues:

```

```

var dialogue = dialogues[npc_name]
update_npc_dialogue(npc_name, dialogue)

func update_npc_dialogue(npc_name: String, dialogue: String):
    """更新指定NPC的对话"""
    var npc_node = get_npc_node(npc_name)
    if npc_node and npc_node.has_method("update_dialogue"):
        npc_node.update_dialogue(dialogue)

func get_npc_node(npc_name: String) -> Node2D:
    """根据名字获取NPC节点"""
    match npc_name:
        "张三":
            return npc_zhang
        "李四":
            return npc_li
        "王五":
            return npc_wang
        _:
            return null

```

主场景脚本的核心功能是定时从后端获取 NPC 状态。在 `_ready()` 中，我们获取 `APIClient` 单例的引用，并连接 `npc_status_received` 信号。然后立即调用 `get_npc_status()` 获取一次 NPC 状态。在 `_process()` 中，我们使用计时器每隔 `Config.NPC_STATUS_UPDATE_INTERVAL` 秒（默认 30 秒）调用一次 `get_npc_status()`。当收到 NPC 状态更新时，`_on_npc_status_received()` 回调函数会遍历所有 NPC，调用它们的 `update_dialogue()` 方法更新对话气泡。这样，即使玩家不与 NPC 交互，也能看到 NPC 之间的自主对话。

整个前后端通信流程如图 15.14 所示：

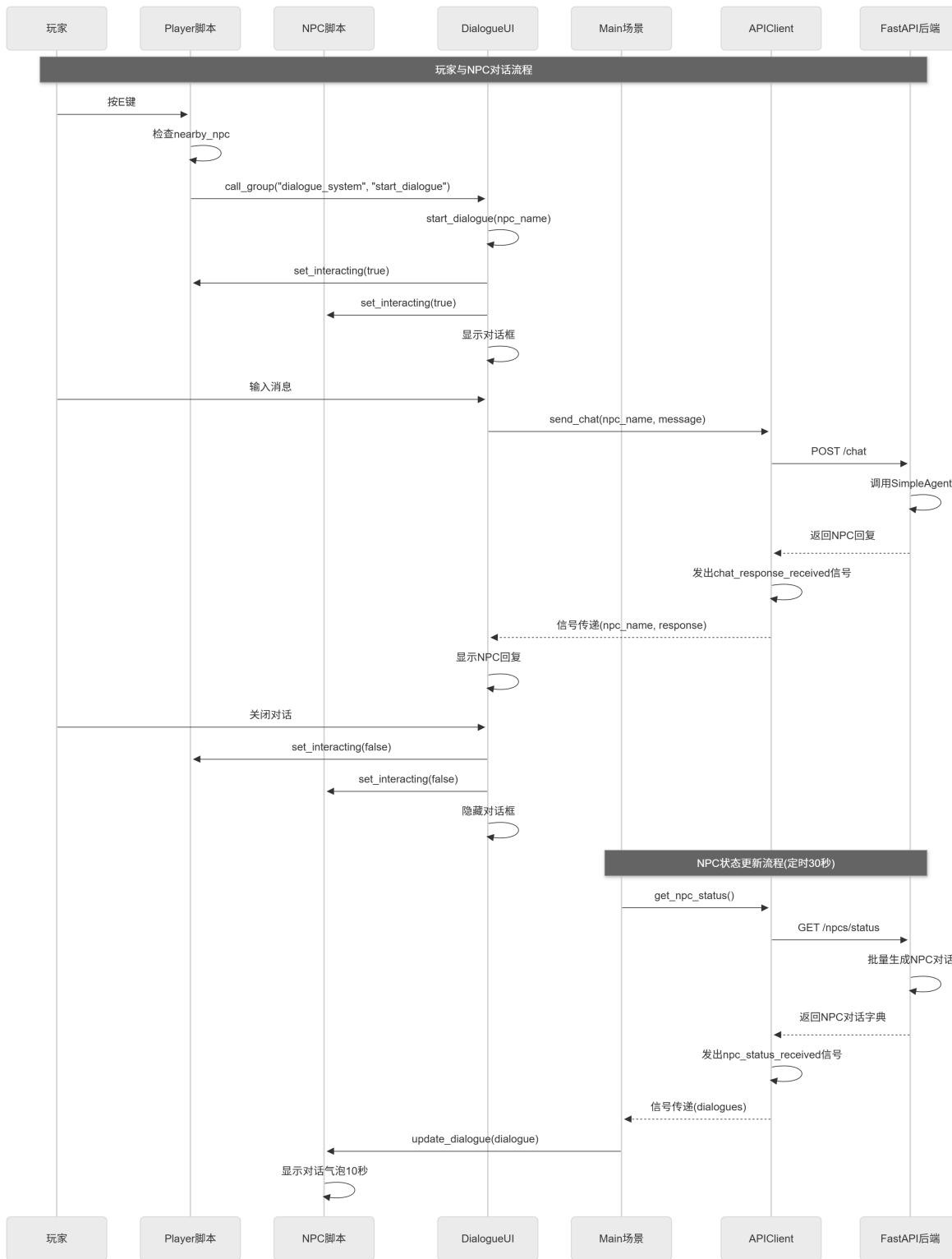


图 15.14 前后端通信完整流程

至此，前后端通信的所有功能都已实现。玩家可以在游戏中自由移动，与 NPC 互动，进行自然语言对话。同时，主场景会定时从后端获取 NPC 状态，更新 NPC 的对话气泡，展示 NPC 之间的自主对话。整个系统使用信号机制进行通信，各个组件之间松耦合，易于维护和扩展。

15.7 总结与展望

15.7.1 本章回顾

在本章中，我们完成了一个完整的 AI 小镇项目——赛博小镇。这个项目将 HelloAgents 框架与 Godot 游戏引擎结合，创造出了一个充满生命力的虚拟世界。让我们回顾一下我们学到的核心内容。

技术架构设计

我们采用了游戏引擎+后端服务的分离架构，将前端渲染、后端逻辑和 AI 智能分离到不同的层次。Godot 负责游戏画面和玩家交互，FastAPI 负责 API 服务和状态管理，HelloAgents 负责 NPC 智能和记忆系统。这种分层设计让每个部分都可以独立开发和测试，也为后续的扩展提供了良好的基础。

NPC 智能体系统

我们使用 HelloAgents 的 SimpleAgent 为每个 NPC 创建了独立的智能体。每个 NPC 都有自己的角色设定、性格特点和记忆系统。通过精心设计的系统提示词，我们让张三成为了一位严谨的 Python 工程师，李四成为了一位善于沟通的产品经理，王五成为了一位富有创意的 UI 设计师。这些 NPC 不仅能够理解玩家的对话，还能根据自己的角色特点做出相应的回复。

记忆与好感度系统

我们实现了两层记忆系统：短期记忆保持对话的连贯性，长期记忆存储所有的互动历史。通过向量数据库的语义检索，NPC 可以回忆起之前讨论过的话题。好感度系统让 NPC 对玩家的态度随着互动而变化，从陌生到挚友，每个等级都有不同的行为表现。这些设计让 NPC 显得更加真实和有趣。

游戏场景构建

我们使用 Godot 创建了一个像素风格的办公室场景，实现了玩家控制、NPC 游走、交互检测和对话 UI。通过场景系统的模块化设计，我们可以轻松地添加新的 NPC、新的场景和新的功能。GDScript 的简洁语法让游戏逻辑的实现变得直观和高效。

前后端通信

我们使用 HTTP REST API 实现了 Godot 前端与 FastAPI 后端的通信。通过异步请求和信号系统，我们保证了游戏的流畅性，即使网络延迟较高也不会影响玩家体验。API 客户端的封装让其他脚本可以方便地调用后端服务，对话 UI 的实现让玩家可以自然地与 NPC 交流。

整个项目的技术栈如图 15.15 所示：

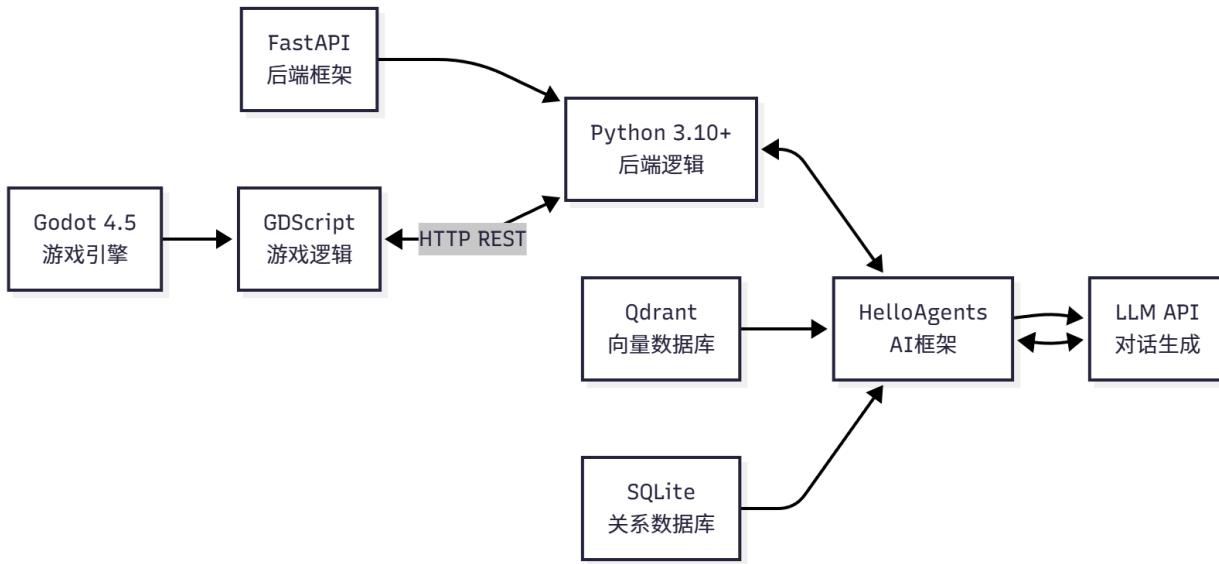


图 15.15 赛博小镇技术栈

15.7.2 扩展方向

赛博小镇只是一个起点，还有很多可以扩展的方向。这些扩展不仅能够增强游戏的趣味性，也能探索 AI 技术在游戏中的更多可能性。

(1) 多人在线支持

目前的赛博小镇是单人游戏，但我们可以将其扩展为多人在线游戏。多个玩家可以同时进入同一个办公室，与 NPC 和其他玩家互动。这需要引入 WebSocket 进行实时通信，以及数据库来持久化玩家数据和 NPC 状态。NPC 可以记住与不同玩家的互动，对每个玩家保持独立的好感度。

(2) 任务系统

我们可以为 NPC 设计任务系统。当玩家与 NPC 的好感度达到一定程度时，NPC 会提供特殊任务。比如张三可能会请玩家帮忙调试一段代码，李四可能会请玩家收集用户反馈，王五可能会请玩家评价设计方案。完成任务可以获得奖励，也能进一步提升好感度。

(3) NPC 之间的互动

目前 NPC 只与玩家互动，但我们可以让 NPC 之间也能互动。张三可以和李四讨论产品需求，李四可以和王五讨论界面设计，王五可以和张三讨论技术实现。这些互动可以在后台自动进行，玩家可以观察到 NPC 之间的对话，让整个世界显得更加生动。

(4) 情感系统

除了好感度，我们还可以为 NPC 添加更复杂的情感系统。NPC 可以有开心、难过、生气、兴奋等不同的情绪状态，这些情绪会影响 NPC 的回复风格和行为。比如当 NPC 心情好的时候，会更愿意分享信息；当 NPC 心情不好的时候，可能会比较冷淡。

(5) 动态事件系统

我们可以设计一些动态事件，让游戏世界更加丰富。比如定期举办团队会议，所有 NPC 和玩家聚在一起讨论项目进展；或者举办生日派对，庆祝某个 NPC 的生日；或者突发紧急任务，需要大家协作完成。这些事件可以增加游戏的变化性和趣味性。

(6) 更大的世界

目前的赛博小镇只有一个办公室场景，但我们可以扩展到更大的世界。可以添加咖啡厅、图书馆、公园等不同的场景，每个场景有不同的 NPC 和互动方式。玩家可以在不同场景之间移动，探索更广阔的虚拟世界。

(7) 个性化学习

NPC 可以学习每个玩家的偏好和习惯。比如如果玩家经常和张三讨论 Python，NPC 会记住玩家对编程感兴趣，以后会主动分享相关的内容。如果玩家喜欢在晚上玩游戏，NPC 会记住这个时间习惯，在晚上更加活跃。

15.7.3 思考与展望

赛博小镇展示了 AI 技术在游戏中的巨大潜力。传统游戏中的 NPC 受限于预设的对话树和脚本，而 AI NPC 可以理解和生成自然语言，与玩家进行真正的对话。这不仅提升了游戏的沉浸感，也为游戏设计带来了新的可能性。

但 AI NPC 也面临一些挑战。首先是成本问题，每次对话都需要调用 LLM API，这会产生一定的费用。对于大型多人在线游戏，这个成本可能会很高。其次是延迟问题，LLM 的推理需要时间，如果网络延迟较高，玩家可能需要等待几秒才能看到 NPC 的回复。最后是内容控制问题，LLM 生成的内容可能不完全可控，需要设计好的提示词和内容过滤机制。

尽管有这些挑战，AI NPC 的未来仍然充满希望。随着 LLM 技术的发展，推理速度会越来越快，成本会越来越低。本地化的小型 LLM 也在快速发展，未来可能可以在玩家的设备上直接运行，完全不需要网络请求。AI 技术与游戏的结合，将为玩家带来前所未有的体验。

在第五部分的毕业设计章节，我们将会学习如何用单智能体和多智能体构造通用智能体，这将是你的创作时间，敬请期待！

第十六章 毕业设计：构建属于你的多智能体应用

恭喜你来到 Hello-Agents 教程的最后一章！在前面的 15 章中，我们从零开始构建了 HelloAgents 框架，学习了智能体的核心概念、多种范式、工具系统、记忆机制、通信协议、强化学习训练和性能评估等知识。在第 13-15 章中，我们还通过三个完整的实战项目（智能旅行助手、自动化深度研究智能体、赛博小镇）展示了如何将所学知识融会贯通。

现在，是时候让你成为真正的智能体系统构建者了！本章将指导你构建属于你自己的多智能体应用，并通过开源协作的方式与社区分享你的成果。

16.1 毕业设计的意义

16.1.1 为什么要做毕业设计

学习技术最好的方式不是看教程，而是动手实践。通过前面章节的学习，你已经掌握了构建智能体系统的理论知识和技术工具。但是，真正的挑战在于：如何将这些知识应用到实际问题中？如何设计一个完整的系统？如何处理各种边界情况和异常？

毕业设计的核心价值在于培养你的综合应用能力，将前面学到的所有知识（智能体范式、工具系统、记忆机制、通信协议等）选择性的整合到一个完整的项目中。

通过本章的学习和实践，希望你能够独立设计并实现一个完整的智能体应用，熟练使用 HelloAgents 框架的各种功能，掌握 Git 和 GitHub 的基本操作，学会编写清晰的项目文档，参与开源社区的协作开发，最终获得一个可以展示的技术作品。

16.1.2 毕业设计的形式

你的毕业设计将以开源项目的形式提交到 Hello-Agents 的共创项目仓库（Co-creation-projects 目录）。具体要求如下：

1. 项目命名：使用{你的GitHub用户名}-{项目名称}的格式，例如jjyaoao-CodeReviewAgent
2. 项目内容：
 - 一个可运行的 Jupyter Notebook (.ipynb文件) 或 Python 脚本
 - 完整的依赖列表 (requirements.txt)
 - 清晰的 README 文档 (README.md)
 - 可选：演示视频、截图、数据集等
3. 提交方式：通过 GitHub 的 Pull Request (PR) 提交
4. 评审流程：社区成员会 review 你的代码，提出改进建议，通过后合并到主仓库

16.2 项目选题指南

16.2.1 选题原则

一个好的毕业设计项目应该具有实用性，解决真实的问题而不是为了技术而技术，我们需要追求在有限的时间和资源内可以完成，并且能够清晰地展示你的技术能力。

16.2.2 推荐选题方向

以下是一些推荐的项目方向，你可以选择其中一个，也可以自己提出新的想法：

(1) 生产力工具类

- 智能代码审查助手：自动分析代码质量、发现潜在 bug、提出优化建议
- 智能文档生成器：根据代码自动生成 API 文档、用户手册
- 智能会议助手：记录会议内容、生成会议纪要、提取行动项
- 智能邮件助手：自动分类邮件、生成回复草稿、提醒重要事项

(2) 学习辅助类

- 智能学习伙伴：根据学习进度推荐学习资源、生成练习题、答疑解惑
- 智能论文助手：帮助查找文献、总结论文、生成引用
- 智能编程导师：提供编程练习、代码 review、学习路径规划
- 智能语言学习助手：提供对话练习、语法纠错、词汇扩展

(3) 创意娱乐类

- 智能故事生成器：根据用户输入生成小说、剧本、诗歌
- 智能游戏 NPC：创建有个性的游戏角色，能够与玩家自然对话
- 智能音乐推荐：根据心情、场景推荐音乐，生成播放列表

- 智能菜谱助手：根据食材、口味推荐菜谱，生成购物清单

(4) 数据分析类

- 智能数据分析师：自动分析数据、生成可视化图表、撰写分析报告
- 智能股票分析：分析股票数据、新闻舆情，提供投资建议
- 智能舆情监控：监控社交媒体、新闻网站，分析舆情趋势
- 智能竞品分析：收集竞品信息、对比分析、生成报告

(5) 生活服务类

- 智能健康助手：记录健康数据、提供健康建议、制定运动计划
- 智能理财助手：记录收支、分析消费习惯、提供理财建议
- 智能购物助手：比价、推荐商品、生成购物清单
- 智能家居控制：通过自然语言控制智能家居设备

16.2.3 选题示例

让我们通过一个具体的例子来说明如何选题和设计项目。

项目名称：智能代码审查助手（CodeReviewAgent）

问题分析：代码审查是软件开发中的重要环节，但人工审查耗时且容易遗漏问题。现有的静态分析工具只能发现语法错误，无法理解代码逻辑，因此需要一个能够理解代码语义、提供深度分析的智能助手。

核心功能：该项目将实现代码质量分析（检查代码风格、命名规范、注释完整性）、潜在 bug 检测（发现逻辑错误、边界条件问题、资源泄漏）、性能优化建议（识别性能瓶颈、提出优化方案）、安全漏洞扫描（检测 SQL 注入、XSS 等安全问题）以及最佳实践推荐（根据语言特性和设计模式提出改进建议）。

预期成果：最终将交付一个可运行的 Jupyter Notebook 展示完整的审查流程，支持 Python、JavaScript 等主流语言，能够生成结构化的 Markdown 格式审查报告，并提供具体的代码示例和改进建议。

16.3 开发环境准备

16.3.1 安装必要工具

在开始开发之前，请确保你的开发环境已经安装了以下工具：

(1) Python 环境

```
# 安装HelloAgents
pip install "hello-agents[all]"
```

(2) Git 和 GitHub

```
# 检查Git版本
git --version

# 配置Git用户信息
git config --global user.name "你的名字"
git config --global user.email "你的邮箱"

# 配置GitHub SSH密钥（推荐）
# 1. 生成SSH密钥
ssh-keygen -t ed25519 -C "你的邮箱"

# 2. 将公钥添加到GitHub
# 复制 ~/.ssh/id_ed25519.pub 的内容
# 在GitHub Settings > SSH and GPG keys 中添加

# 3. 测试连接
ssh -T git@github.com
```

(3) Jupyter Notebook

```
# 安装Jupyter
pip install jupyter notebook

# 或者使用JupyterLab（推荐）
pip install jupyterlab

# 启动Jupyter
jupyter lab
```

16.3.2 Fork 项目仓库

步骤 1：Fork 仓库

1. 访问 Hello-Agents 仓库: <https://github.com/datawhalechina/hello-agents>
2. 点击右上角的“Fork”按钮, 如图 16.1 红色方框位置
3. 选择你的 GitHub 账号, 创建 Fork

The screenshot shows the GitHub repository page for 'hello-agents'. At the top right, there is a 'Fork' button with a red box around it. Below the header, there's a list of commits by 'jjyaoao' with dates and descriptions. On the right side, there's an 'About' section with repository statistics like '354 Commits', '2.1k stars', and '208 forks'. There are also links to 'Readme', 'View license', 'Activity', 'Custom properties', and 'Report repository'.

图 16.1 Fork 仓库步骤

步骤 2: 克隆到本地

```
# 如图16.2所示, 克隆你Fork的仓库
git clone git@github.com:你的用户名/hello-agents.git

# 进入项目目录
cd Hello-Agents

# 添加上游仓库 (用于同步更新)
git remote add upstream https://github.com/datawhalechina/hello-agents.git

# 查看远程仓库
git remote -v
```

● (base) PS D:\Example> git clone git@github.com:jjyaoao/Hello-Agents.git
 Cloning into 'Hello-Agents'...
 remote: Enumerating objects: 2428, done.
 remote: Counting objects: 100% (728/728), done.
 remote: Compressing objects: 100% (106/106), done.
 remote: Total 2428 (delta 676), reused 635 (delta 622), pack-reused 1700 (from 2)
 Receiving objects: 100% (2428/2428), 72.85 MiB | 8.23 MiB/s, done.
 Resolving deltas: 100% (1102/1102), done.
● (base) PS D:\Example> cd Hello-Agents
● (base) PS D:\Example\Hello-Agents> git remote add upstream https://github.com/datawhalechina/Hello-Agents.git
● (base) PS D:\Example\Hello-Agents> git remote -v
origin git@github.com:jjyaoao/Hello-Agents.git (fetch)
origin git@github.com:jjyaoao/Hello-Agents.git (push)
upstream https://github.com/datawhalechina/Hello-Agents.git (fetch)
upstream https://github.com/datawhalechina/Hello-Agents.git (push)

图 16.2 克隆仓库到本地

步骤 3: 创建开发分支

```
# 创建并切换到新分支
git checkout -b feature/你的项目名称

# 例如:
git checkout -b feature/code-review-agent
```

16.3.3 项目目录结构

在Co-creation-projects目录下创建你的项目文件夹:

```
# 进入共创项目目录
cd Co-creation-projects

# 创建项目文件夹 (格式:GitHub用户名-项目名称)
mkdir 你的用户名-项目名称

# 例如:
mkdir jjyaoao-CodeReviewAgent

# 进入项目目录
cd jjyaoao-CodeReviewAgent
```

推荐的项目结构:

```
jjyaoao-CodeReviewAgent/
├── README.md          # 项目说明文档
├── requirements.txt    # Python依赖列表
├── main.ipynb          # 主要的Jupyter Notebook
└── data/
    ├── sample_code.py   # 数据文件 (可选)
    └── test_cases.json
└── outputs/
    ├── review_report.md # 输出结果 (可选)
    └── screenshots/
└── src/
    ├── agents/          # 源代码 (可选, 如果代码较多)
    ├── tools/
    └── utils/
```

16.4 项目开发指南

16.4.1 编写 README 文档

README 是项目的门面，一个好的 README 应该包含以下内容:

```
# 项目名称
```

```
> 一句话描述你的项目
```

```
## 📝 项目简介
```

详细介绍你的项目:

- 解决什么问题?
- 有什么特色功能?
- 适用于什么场景?

```
## ✨ 核心功能
```

- [] 功能1: 描述
- [] 功能2: 描述
- [] 功能3: 描述

```
## 🛠 技术栈
```

- HelloAgents框架
- 使用的智能体范式 (如ReAct、Plan-and-Solve等)
- 使用的工具和API
- 其他依赖库

```
## 🚀 快速开始
```

```
### 环境要求
```

- Python 3.10+
- 其他要求

```
### 安装依赖
```

```
pip install -r requirements.txt
```

```
### 配置API密钥
```

```
# 创建.env文件
cp .env.example .env
```

```
# 编辑.env文件, 填入你的API密钥
```

```
### 运行项目
```

```
# 启动Jupyter Notebook
jupyter lab
```

```
# 打开main.ipynb并运行
```

```
## 📱 使用示例
```

展示如何使用你的项目，最好包含代码示例和运行结果。

```
## 🌟 项目亮点
```

- 亮点1：说明
- 亮点2：说明
- 亮点3：说明

📈 性能评估

如果有评估结果，展示在这里：

- 准确率：XX%
- 响应时间：XX秒
- 其他指标

🎨 未来计划

- [] 待实现的功能1
- [] 待实现的功能2
- [] 待优化的部分

🌟 贡献指南

欢迎提出Issue和Pull Request！

📄 许可证

MIT License

🚀 作者

- GitHub: [@你的用户名] (<https://github.com/你的用户名>)
- Email: 你的邮箱（可选）

🙏 致谢

感谢Datawhale社区和Hello-Agents项目！

16.4.2 编写 requirements.txt

列出项目所需的所有 Python 依赖：

```
# 核心依赖
hello-agents[all]>=0.2.7

# 可视化（如果需要）
matplotlib>=3.7.0
plotly>=5.14.0

# Web框架（如果需要）
fastapi>=0.109.0
uvicorn>=0.27.0
```

16.4.3 开发 Jupyter Notebook

(1) Notebook 结构建议

一个好的 Jupyter Notebook 应该包含以下部分：

```
# =====
# 第1部分：项目介绍
# =====

.....
# 项目名称

## 项目简介
简要介绍项目的目标和功能

## 作者信息
- 姓名:XXX
- GitHub:@XXX
- 日期:2025-XX-XX
.....

# =====
# 第2部分：环境配置
# =====

# 安装依赖
!pip install -q hello-agents[all]

# 导入必要的库
from hello_agents import SimpleAgent, HelloAgentsLLM
from hello_agents.tools import BaseTool
import os
from dotenv import load_dotenv

# 加载环境变量
load_dotenv()
```

```

# =====
# 第3部分:工具定义
# =====

class CustomTool(BaseTool):
    """自定义工具类"""

    name = "tool_name"
    description = "工具描述"

    def run(self, query: str) -> str:
        """工具执行逻辑"""
        # 实现你的工具逻辑
        return "结果"

# =====
# 第4部分:智能体构建
# =====

# 创建LLM
llm = HelloAgentsLLM()

# 创建智能体
agent = SimpleAgent(
    name="智能体名称",
    llm=llm,
    system_prompt="系统提示词"
)

# 添加工具
agent.add_tool(CustomTool())

# =====
# 第5部分:功能演示
# =====

# 示例1:基础功能
print("==>示例1:基础功能 ==>")
result = agent.run("用户输入")
print(result)

# 示例2:复杂场景
print("\n==>示例2:复杂场景 ==>")
result = agent.run("复杂的用户输入")
print(result)

# =====
# 第6部分:性能评估 (可选)
# =====

# 评估代码
# ...

# =====
# 第7部分:总结与展望
# =====

.....
## 项目总结

#### 实现的功能
- 功能1
- 功能2

#### 遇到的挑战
- 挑战1及解决方案
- 挑战2及解决方案

#### 未来改进方向
- 改进1
- 改进2
.....

```

16.4.4 测试你的项目

在提交之前，可以使用测试清单来判断自己的项目是否满足提交要求：

- [] 代码能够正常运行，没有报错
- [] README文档完整，说明清晰
- [] requirements.txt包含所有依赖
- [] 有清晰的使用示例
- [] 代码有适当的注释
- [] 输出结果符合预期
- [] 处理了常见的异常情况
- [] 项目结构清晰，文件命名规范
- [] 大文件已妥善处理（见下节）

16.4.5 大文件处理指南

⚠ 重要: 避免主仓库过大

为了保持 Hello-Agents 主仓库的轻量化, 请遵循以下大文件处理规范:

(1) 文件大小限制

- 项目总大小: 不超过 5MB
- 禁止直接提交: 视频文件、大型数据集、模型文件

(2) 大文件处理方案

如果你的项目包含大文件 (数据集、视频、模型等), 请使用以下方案:

方案 1: 使用外部链接 (推荐)

将大文件上传到外部平台, 在 README 中提供下载链接:

```
## 数据集
```

本项目使用的数据集较大, 请从以下链接下载:

- 数据集1: [百度网盘](链接) 提取码: xxxx
- 数据集2: [Google Drive](链接)
- 演示视频: [B站](链接) / [YouTube](链接)

推荐的外部平台: - 数据集: 百度网盘、Google Drive、Kaggle、HuggingFace Datasets - 视频: B 站、YouTube、腾讯视频 - 模型: HuggingFace Models、ModelScope - 图片: GitHub Issues、图床服务

方案 2: 创建独立仓库

如果项目资源较多, 建议创建独立的数据仓库:

```
## 项目资源
```

由于项目包含大量数据和演示资源, 已单独创建资源仓库:

- 资源仓库: <https://github.com/你的用户名/项目名称-resources>
- 包含内容: 数据集、演示视频、模型文件、测试数据等

```
### 使用方法
```

```
\```\`bash
# 克隆资源仓库
git clone https://github.com/你的用户名/项目名称-resources.git

# 将数据放到项目目录
cp -r 项目名称-resources/data ./data
\````
```

方案 3: 使用示例数据

在主仓库中只提供小规模的示例数据:

```
# 在README中说明
## 数据说明

- `data/sample.csv`: 示例数据 (100条记录)
- 完整数据集 (10万条记录) 请从[这里](链接)下载
```

(3) 最佳实践示例

```
你的用户名-项目名称/
├── README.md          # 包含外部资源链接
├── requirements.txt
├── main.ipynb
├── .gitignore          # 忽略大文件
└── data/
    └── sample.csv      # 仅示例数据 (<1MB)
└── outputs/
    └── demo_result.png # 仅演示结果 (<1MB)
```

README 中的说明:

```
## 数据和资源

#### 示例数据
项目包含小规模示例数据用于快速测试 (位于`data/sample.csv`)

#### 完整数据集
完整数据集 (500MB) 请从以下链接下载:
file:///Users/knight/workspace/hello-agents/docs/Hello-Agents-Full.html
```

2/9/26, 1:56 PM

- 百度网盘: [链接] 提取码: xxxx
- 下载后解压到`data/`目录

- ### 演示视频
- 站: [项目演示视频](链接)
 - YouTube: [Demo Video](链接)

16.5 提交 Pull Request

16.5.1 提交代码到 GitHub

步骤 1: 检查修改

```
# 查看修改的文件  
git status
```

步骤 2: 添加文件

```
# 添加所有修改的文件  
git add .  
  
# 或者添加特定文件  
git add Co-creation-projects/你的用户名-项目名称/
```

步骤 3: 提交修改

提交信息应遵循以下格式:

```
# 格式: 类型: 简短描述  
git commit -m "feat: 添加XXX毕业设计项目"
```

提交类型规范:

- feat: 新增功能或项目 (毕业设计项目使用此类型)
- fix: 修复 bug
- docs: 文档更新
- style: 代码格式调整 (不影响功能)
- refactor: 代码重构
- test: 测试相关
- chore: 其他修改 (如依赖更新)

步骤 4: 推送到 GitHub

```
# 推送到你的Fork仓库  
git push origin feature/你的项目名称
```

16.5.2 创建 Pull Request

步骤 1: 访问 GitHub

1. 访问你 Fork 的仓库: <https://github.com/你的用户名/hello-agents>
2. 点击“Pull requests”标签, 如图 16.3 所示
3. 点击“New pull request”按钮

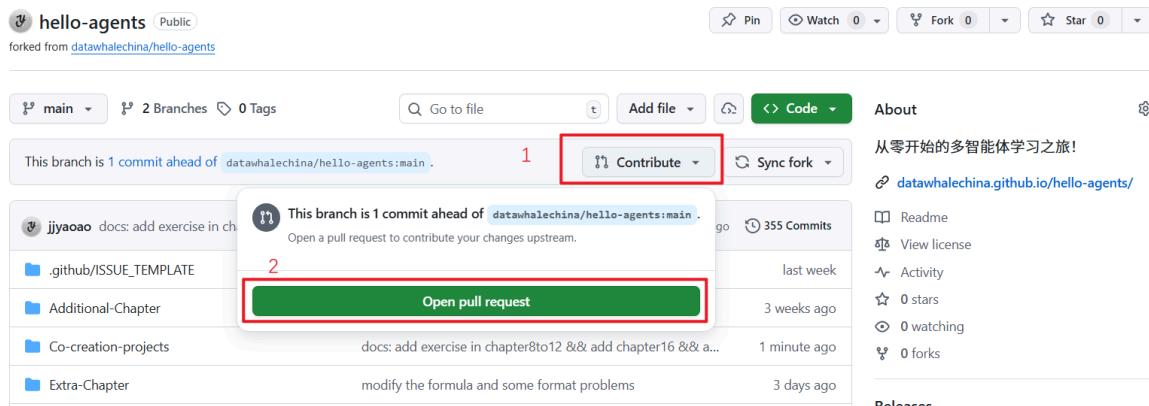


图 16.3 创建 Pull Request

步骤 2: 选择分支

- Base repository: [datawhalechina/hello-agents](https://github.com/datawhalechina/hello-agents)

- Base branch: main
- Head repository: 你的用户名/hello-agents
- Compare branch: feature/你的项目名称

步骤 3: 填写 PR 信息

⚠ 重要: PR 标题统一格式

为了便于管理和检索，所有毕业设计项目的 PR 标题必须遵循以下格式：

[毕业设计] 项目名称 - 简短描述

示例： - [毕业设计] CodeReviewAgent - 智能代码审查助手 - [毕业设计] StudyBuddy - AI学习伙伴 - [毕业设计] DataAnalyst - 智能数据分析师

PR 描述模板：

项目信息

- **项目名称**:XXX
- **作者**:@你的用户名
- **项目类型**:生产力工具/学习辅助/创意娱乐/数据分析/生活服务

项目简介

简要描述你的项目（2-3句话）

核心功能

- [] 功能1
- [] 功能2
- [] 功能3

技术亮点

- 使用了XXX范式
- 实现了XXX功能
- 优化了XXX性能

演示效果

(可选) 添加截图或GIF展示项目效果

自检清单

- [] 代码能够正常运行
- [] README文档完整
- [] requirements.txt完整
- [] 有清晰的使用示例
- [] 代码有适当的注释

其他说明

(可选) 其他需要说明的内容

步骤 4: 提交 PR

如图 16.4 所示，点击“Create pull request”按钮提交。

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#) or [learn more about diff comparisons](#).

The screenshot shows the GitHub interface for creating a pull request. At the top, it displays the base repository as `base repository: datawhalechina/hello-agents`, the base branch as `base: main`, the head repository as `head repository: jjyaoao/hello-agents`, and the compare branch as `compare: main`. A green checkmark indicates that the branches are **Able to merge**. Below this, there are sections for adding a title, description, and project information. The project information section includes a summary of the project: name (`CodeReviewAgent - 智能代码审查助手`), author (`@jjyaoao`), and type (`生产力工具`). There are tabs for `Write` and `Preview`. To the right, there are various settings for the pull request, such as **Reviewers**, **Suggestions**, **Copilot**, **Request**, **Assignees** (set to "No one—assign yourself"), **Labels** (set to "None yet"), **Projects** (set to "None yet"), **Milestone** (set to "No milestone"), **Development** (with a note about closing keywords), and **Helpful resources** (link to GitHub Community Guidelines). At the bottom, there is a red box highlighting the **Create pull request** button.

图 16.4 提交 Pull Request

16.5.3 响应 Review 意见

提交 PR 后，社区成员会 review 你的代码并提出建议。请及时响应：

1. 查看评论：在 PR 页面查看 reviewer 的评论

2. 修改代码：根据建议修改代码

3. 提交更新：

```
git add .
git commit -m "fix: 根据review意见修改XXX"
git push origin feature/你的项目名称
```

4. 回复评论：在 GitHub 上回复 reviewer，说明你的修改

16.6 示例项目展示

为了帮助你更好地理解毕业设计的要求，这里展示一个完整的示例项目，请别担心，小的创意同样可以被收录，只要是自己动手的作品都是值得珍惜的。

项目信息

- 项目名称：CodeReviewAgent
- 作者：@jjyaoao
- 项目路径：`Co-creation-projects/jjyaoao-CodeReviewAgent/`

项目结构

```
jjyaoao-CodeReviewAgent/
├── README.md          # 项目文档
├── requirements.txt    # 依赖列表
├── main.ipynb          # 主程序(含快速演示和完整功能)
├── .env.example         # 环境变量示例
├── .gitignore           # Git忽略规则
└── data/
    └── sample_code.py   # 示例代码
    └── outputs/
        └── review_report.md # 示例报告
```

核心代码片段 (main.ipynb)

```

# =====
# 智能代码审查助手
# =====

from hello_agents import SimpleAgent, HelloAgentsLLM, ToolRegistry
from hello_agents.tools import Tool, ToolParameter
from typing import Dict, Any, List
import ast
import os

# =====
# 0. 配置LLM参数
# =====

os.environ["LLM_MODEL_ID"] = "Qwen/Qwen2.5-72B-Instruct"
os.environ["LLM_API_KEY"] = "your_api_key_here"
os.environ["LLM_BASE_URL"] = "https://api-inference.modelscope.cn/v1/"
os.environ["LLM_TIMEOUT"] = "60"

# =====
# 1. 定义代码分析工具
# =====

class CodeAnalysisTool(Tool):
    """代码静态分析工具"""

    def __init__(self):
        super().__init__(
            name="code_analysis",
            description="分析Python代码的结构、复杂度和潜在问题"
        )

    def run(self, parameters: Dict[str, Any]) -> str:
        """分析代码并返回结果"""
        code = parameters.get("code", "")
        if not code:
            return "错误:代码不能为空"

        try:
            tree = ast.parse(code)
            functions = [node for node in ast.walk(tree)
                         if isinstance(node, ast.FunctionDef)]
            classes = [node for node in ast.walk(tree)
                       if isinstance(node, ast.ClassDef)]

            result = {
                "函数数量": len(functions),
                "类数量": len(classes),
                "代码行数": len(code.split('\n')),
                "函数列表": [f.name for f in functions],
                "类列表": [c.name for c in classes]
            }
            return str(result)
        except SyntaxError as e:
            return f"语法错误:{str(e)}"

    def get_parameters(self) -> List[ToolParameter]:
        return [
            ToolParameter(
                name="code",
                type="string",
                description="要分析的Python代码",
                required=True
            )
        ]

class StyleCheckTool(Tool):
    """代码风格检查工具"""

    def __init__(self):
        super().__init__(
            name="style_check",
            description="检查代码是否符合PEP 8规范"
        )

    def run(self, parameters: Dict[str, Any]) -> str:
        """检查代码风格"""
        code = parameters.get("code", "")
        if not code:
            return "错误:代码不能为空"

        issues = []
        lines = code.split('\n')
        for i, line in enumerate(lines, 1):
            if len(line) > 79:
                issues.append(f"第{i}行:超过79个字符")
            if line.startswith(' ') and not line.startswith('    '):
                if len(line) - len(line.lstrip()) not in [0, 4, 8, 12]:
                    issues.append(f"第{i}行:缩进不规范")

        if not issues:
            return "没有发现风格问题"
        else:
            return "\n".join(issues)

```

```

        return "代码风格良好, 符合PEP 8规范"
        return "发现以下问题:\n" + "\n".join(issues)

    def get_parameters(self) -> List[ToolParameter]:
        return [
            ToolParameter(
                name="code",
                type="string",
                description="要检查的Python代码",
                required=True
            )
        ]

# =====
# 2. 创建工具注册表和智能体
# =====

# 创建工具注册表
tool_registry = ToolRegistry()
tool_registry.register_tool(CodeAnalysisTool())
tool_registry.register_tool(StyleCheckTool())

# 初始化LLM
llm = HelloAgentsLLM()

# 定义系统提示词
system_prompt = """你是一位经验丰富的代码审查专家。你的任务是:

1. 使用code_analysis工具分析代码结构
2. 使用style_check工具检查代码风格
3. 基于分析结果, 提供详细的审查报告

审查报告应包括:
- 代码结构分析
- 风格问题
- 潜在bug
- 性能优化建议
- 最佳实践建议

请以Markdown格式输出报告。"""

# 创建智能体
agent = SimpleAgent(
    name="代码审查助手",
    llm=llm,
    system_prompt=system_prompt,
    tool_registry=tool_registry
)

# =====
# 3. 运行示例
# =====

# 读取示例代码
with open("data/sample_code.py", "r", encoding="utf-8") as f:
    sample_code = f.read()

print("==待审查的代码==")
print(sample_code)
print("\n" + "="*50 + "\n")

# 执行代码审查
print("==开始代码审查==")
review_result = agent.run(f"请审查以下Python代码:\n\n```python\n{sample_code}\n```")

print(review_result)

# 保存审查报告
with open("outputs/review_report.md", "w", encoding="utf-8") as f:
    f.write(review_result)

print("\n审查报告已保存到 outputs/review_report.md")

README.md示例

# CodeReviewAgent - 智能代码审查助手

> 基于HelloAgents框架的智能代码审查工具

## 项目简介

CodeReviewAgent是一个智能代码审查助手, 能够自动分析Python代码的质量、发现潜在问题并提供优化建议。

### 核心功能

-  代码结构分析:统计函数、类、代码行数等
-  风格检查:检查是否符合PEP 8规范
-  智能建议:基于LLM提供深度分析和优化建议
-  报告生成:生成Markdown格式的审查报告

```

🛡 技术栈

- HelloAgents 框架 (SimpleAgent + ToolRegistry)
- Python AST 模块 (代码解析)
- ModelScope API (Qwen2.5-72B 模型)

🚀 快速开始

安装依赖

```
\`\'\`bash
pip install -r requirements.txt
\`\'\`
```

配置LLM参数

方式1：使用.env文件

```
\`\'\`bash
cp .env.example .env
# 编辑.env文件，填入你的API密钥
\`\'\`
```

方式2：直接在Notebook中设置

项目已预配置ModelScope API，可直接运行。如需修改，编辑main.ipynb第1部分的配置代码。

运行项目

```
\`\'\`bash
jupyter lab
# 打开main.ipynb并运行所有单元格
\`\'\`
```

📊 使用示例

1. 将待审查的代码放入`data/sample_code.py`
2. 运行`main.ipynb`
3. 查看生成的审查报告`outputs/review_report.md`

🌟 项目亮点

- **自动化**：无需人工逐行检查，自动发现问题
- **智能化**：利用LLM理解代码语义，提供深度建议
- **可扩展**：易于添加新的检查规则和工具

🧑 作者

- GitHub: [JJYaoao] (<https://github.com/jjyaoao>)
- 项目链接: [CodeReviewAgent] (<https://github.com/datawhalechina/hello-agents/tree/main/Co-creation-projects/jjyaoao-CodeReviewAgent>)

🙏 致谢

感谢Datawhale社区和Hello-Agents项目！

16.7 总结与展望

通过完成毕业设计，你应该已经掌握了智能体系统设计的完整流程。从需求出发设计系统架构，熟练使用 HelloAgents 框架的各种功能和组件，开发自定义工具扩展智能体能力，完成从需求分析到代码实现的完整项目开发，学会使用 Git 和 GitHub 进行开源协作，以及编写清晰的技术文档。

在本项目中，我们从零开始构建了 HelloAgents 框架，并用它实现了多个实用的应用。完成毕业设计只是开始，你可以继续深入学习更多智能体范式和算法、提示工程和上下文工程、多智能体协作机制等理论知识；也可以扩展技术栈，学习 Web 开发构建完整的应用、学习数据库实现数据持久化、学习部署将应用上线；还可以持续优化你的项目，添加更多功能、优化性能和用户体验、完善测试和文档；更重要的是，积极参与社区贡献，帮助其他学习者、参与 Hello-Agents 框架开发、分享你的经验和心得。

从第一章的简单智能体，到现在能够独立构建完整的多智能体应用，你已经走过了一段精彩的学习旅程。但这不是终点，而是新的起点。

AI 技术日新月异，智能体领域更是充满无限可能。希望你能够保持好奇心持续学习新技术，勇于用 AI 技术解决实际问题创造价值，乐于将你的经验和成果分享给社区，不断打磨你的作品追求卓越。

最后，感谢你完整阅读了本项目。希望你在学习的过程中有所收获，也希望你能够将所学应用到实际项目中，创造出令人惊叹的智能体应用。AI 的未来充满无限可能，让我们一起探索和创造！

记住：最好的学习方式就是动手实践！

现在，开始构建属于你的智能体应用吧！我们期待在 Co-creation-projects 目录中看到你的精彩作品！

如果你觉得 Hello-Agents 项目对你有帮助，请给我们一个⭐ Star！

🎓 恭喜你完成了 Hello-Agents 教程的学习！🎉