12 智能体

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 理解典型智能体的软件结构和基本原理
- ⋆⋆⋆ 能够阅读智能体代码逻辑并定制智能体

12.1 智能体的软件结构

智能体（Agent）是近年来大语言模型应用中的一个重要概念。传统聊天机器人通常采用“用户输入—模型回答”的模式，只能根据已有知识生成文本。而智能体不仅能够进行对话，还能够主动规划任务、调用外部工具、访问记忆，并根据环境反馈持续调整自己的行为。图12.1展示了一个典型智能体的软件结构。

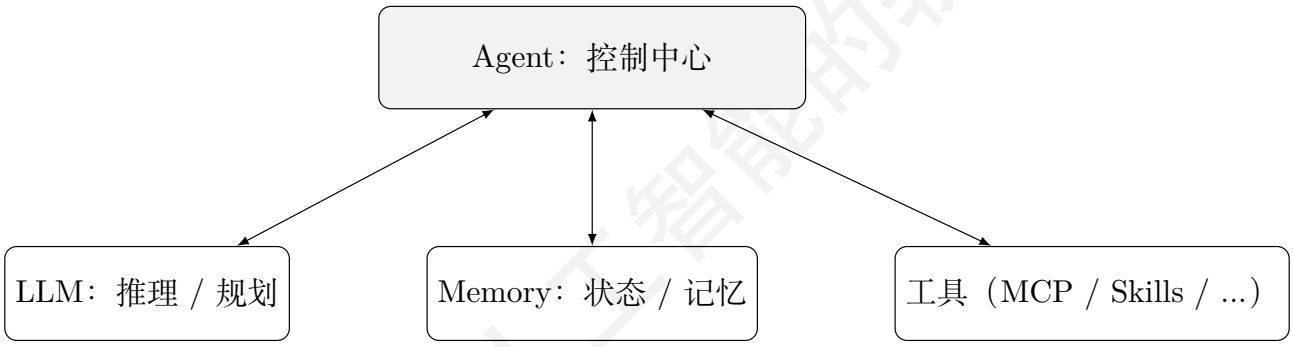

图 12.1: 典型智能体软件栈

- **Agent（控制中心）**：Agent 负责整个系统的调度与控制。它接收用户输入，组织上下文信息，调用大语言模型进行推理，并根据模型输出决定是否需要调用工具。在工具执行结束后，Agent 还需要收集执行结果，并组织最终回复返回给用户。
- **LLM（推理 / 规划）**：LLM 是智能体的核心决策模块，负责理解用户意图、分析当前状态并生成行动计划。例如，当用户要求查询天气、搜索网页或者执行代码时，LLM 需要判断应当调用哪些工具，并决定工具调用顺序。在现代 Agent 系统中，任务规划、任务分解以及推理通常都由 LLM 完成。
- **Memory（状态 / 记忆）**：Memory 用于保存智能体运行过程中的状态信息。最简单的记忆形式是对话历史，即保存用户和系统之前的交互内容。记忆模块使智能体能够在多轮交互中保持上下文一致性。
- **Tools（工具）**：工具模块赋予智能体与外部环境交互的能力。常见工具包括搜索引擎、各种 Web API 以及代码执行器等。通过工具调用，智能体能够执行查询、计算、文件操作等实际任务，而不仅仅是生成文本。近年来出现了多种工具扩展机制。例如，MCP（Model Context Protocol）提供了标准化的工具接入协议；Skill 则为智能体提供额外知识和操作规范，例如软件操作手册或知识库等。二者共同扩展了智能体的能力边界。

12.2 案例：CLI 智能体

本节通过一个最小可运行的 CLI 智能体示例，展示智能体系统的基本工作方式。命令行环境（Linux Shell）的一个典型问题是用户需要记忆大量命令及其参数，这在实际使用中具有较高的学习成本。因此，该智能体的目标是：当用户在命令行中输入自然语言指令时，系统能够自动判断是否需要调用系统命令；在需要时生成并执行对应的 Shell 命令；并将执行结果返回给用户。同时，系统还会对执行结果进行进一步分析与解释，从而降低用户理解命令输出的难度，提升命令行使用的易用性与智能化水平。

智能体的实现如代码12.1所示。

```python
from openai import OpenAI
import os
import subprocess
import json

API_KEY = "sk-YOUKEY" #替换为你自己的API_KEY。
client = OpenAI(
    api_key=API_KEY,
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1"
)
messages = [
    {
        "role": "system",
        "content": """
你是一个CLI智能体。
如果用户请求涉及系统操作（如ls, dir, cat, pwd等），请返回JSON格式:
{
  "action": "run_shell",
  "command": "实际要执行的命令"
}
否则正常回答。
"""
    }
]

def run_shell(cmd: str) -> str:
    """执行shell命令"""
    try:
        result = subprocess.run(
            cmd,
            shell=True,
            capture_output=True,
            text=True
        )
        return result.stdout + result.stderr
    except Exception as e:
        return str(e)

def chat(user_input: str):
    messages.append({"role": "user", "content": user_input})
    response = client.chat.completions.create(
        model="qwen-plus",
```

```
        messages=messages,
        temperature=0.2,        max_tokens=512,
    )
    content = response.choices[0].message.content.strip()
    # 尝试解析是否是工具调用
    try:
        data = json.loads(content)
        if isinstance(data, dict) and data.get("action") == "run_shell":
            cmd = data["command"]
            print(f"\n[EXECUTING] {cmd}\n")
            output = run_shell(cmd)
            # 把执行结果回传给模型
            messages.append({"role": "assistant", "content": content})
            messages.append({
                "role": "user",
                "content": f"命令执行结果: \n{output}\n请总结结果。"
            })

            final = client.chat.completions.create(
                model="qwen-plus",
                messages=messages,
                temperature=0.2,
                max_tokens=512,
            )
            reply = final.choices[0].message.content
        else:
            reply = content

    except Exception:
        # 不是JSON就当普通回答
        reply = content

    messages.append({"role": "assistant", "content": reply})
    return reply

def main():
    print("=== CLI Agent with Execution ===")
    while True:
        user_input = input("\nYou: ")
        if user_input in ["exit", "quit"]:
            break
        reply = chat(user_input)
        print("\nAgent:", reply)

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

代码 12.1: Linux shell 智能体

练习

1) 运行代码 12.1所示的 CLI 智能体程序，观察其在不同指令下的执行效果，并分析其优点与局限性。如果你的开发环境不支持 Linux 系统，可以在大模型辅助下将其改写为 Windows Shell 智能体，并测试其在 Windows 环境下的可用性。