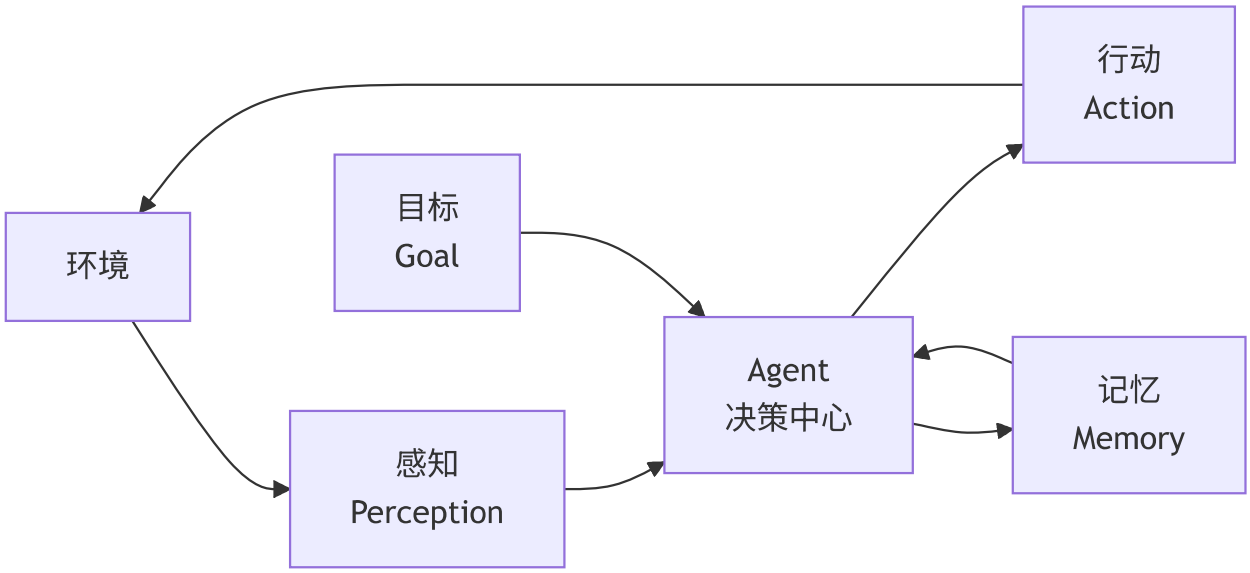
第9章 Agent系统

> Agent让大模型从"对话工具"进化为"能解决问题的智能体"

9.1 什么是Agent?

9.1.1 Agent的定义

Agent（智能体）： 能够感知环境、自主决策并采取行动以达成目标的系统。



LLM Agent的特点：

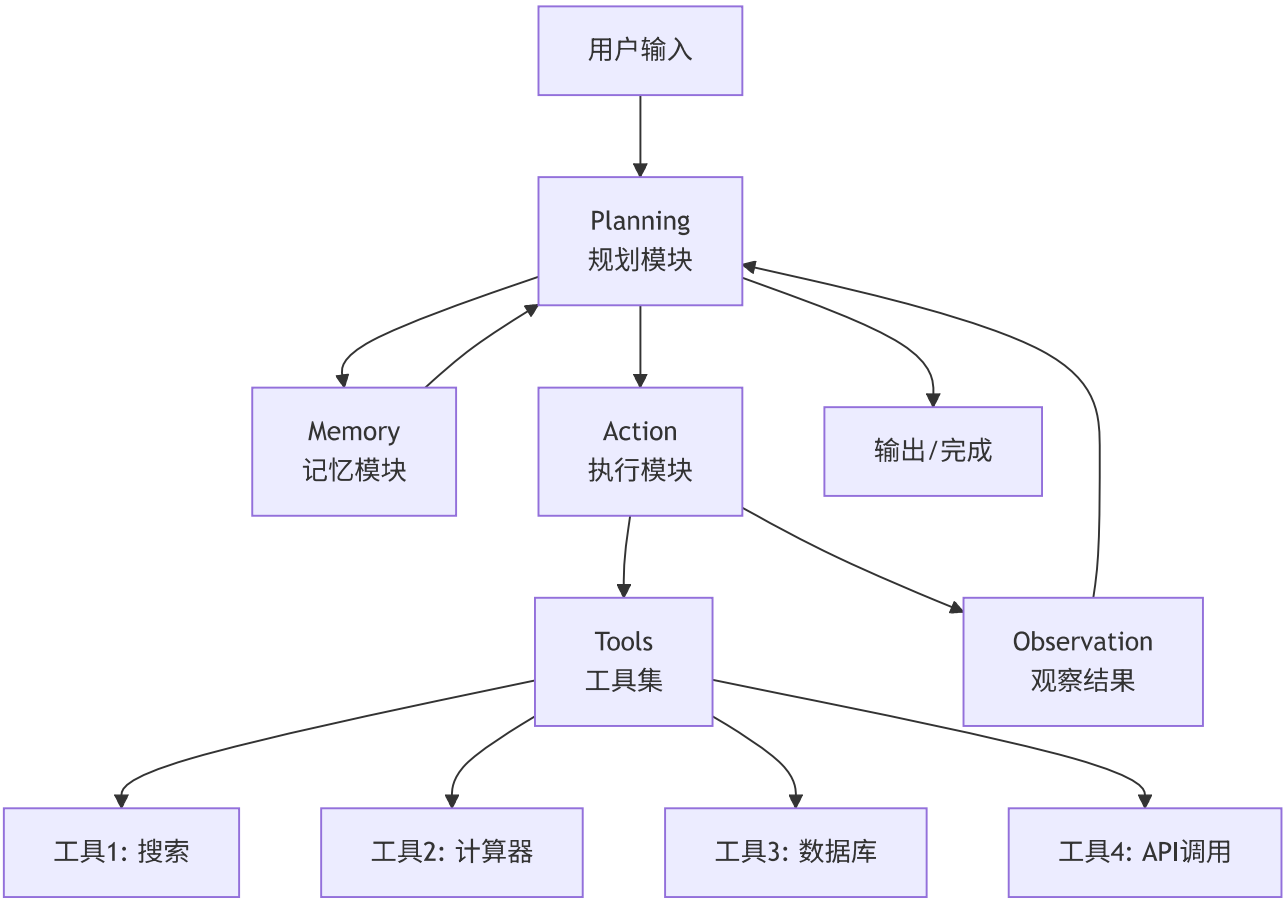
- 🧠 **推理**：使用LLM进行思考和决策
- 🔧 **工具使用**：调用外部工具和API
- 💾 **记忆**：维护历史信息
- 🎯 **目标导向**：自主规划达成目标

9.1.2 Agent vs 传统对话系统

维度	传统对话系统	Agent系统
交互模式	问答式	**任务导向**
能力范围	文本生成	**工具调用、执行操作**
规划能力	无	**多步骤规划**
环境感知	无	**获取实时信息**
自主性	被动响应	**主动行动**

9.2 Agent核心组件

9.2.1 组件架构



9.2.2 规划模块（Planning）

作用： 将复杂任务分解为可执行的步骤。

```
class PlanningModule:
    """
    Agent的规划模块
    """
    def __init__(self, llm):
        self.llm = llm

    def create_plan(self, task, available_tools):
        """
        为任务创建执行计划

        Args:
            task: 用户任务描述
            available_tools: 可用工具列表

        Returns:
            plan: 步骤列表
        """
        prompt = f"""
你是一个任务规划助手。给定任务和可用工具，创建详细的执行计划。
```

```
任务：{task}

可用工具：
{self._format_tools(available_tools)}

请创建执行计划，输出JSON格式：
{{
    "steps": [
        {{"step": 1, "action": "使用工具X", "reason": "原因"}},
        {{"step": 2, "action": "...", "reason": "..."}}
    ]
}}

计划：
"""
        plan_json = self.llm.generate(prompt)
        plan = json.loads(plan_json)

        return plan['steps']

    def _format_tools(self, tools):
        """格式化工具列表"""
        return "\n".join([f"- {tool.name}: {tool.description}"
                          for tool in tools])


# 使用示例
planning = PlanningModule(llm)
task = "查询明天北京的天气，如果下雨则预定室内活动"
tools = [weather_tool, booking_tool, search_tool]

plan = planning.create_plan(task, tools)
# 输出:
# [
#   {"step": 1, "action": "使用weather_tool查询明天北京天气", "reason": "..."},
#   {"step": 2, "action": "根据天气结果，使用booking_tool预定", "reason": "..."}
# ]
```

9.2.3 记忆模块（Memory）

短期记忆 vs 长期记忆：

```
class MemoryModule:
    """
    Agent的记忆系统
    """
```

```
def __init__(self):
    self.short_term = []  # 短期记忆：当前对话
    self.long_term = []   # 长期记忆：历史经验
    self.working = {}     # 工作记忆：当前任务状态

def add_to_short_term(self, interaction):
    """
    添加到短期记忆

    Args:
        interaction: {"role": "user"/"assistant", "content": "..."}
    """
    self.short_term.append(interaction)

    # 限制短期记忆长度
    if len(self.short_term) > 20:
        self.short_term = self.short_term[-20:]

def add_to_long_term(self, experience):
    """
    添加到长期记忆（重要经验）

    Args:
        experience: {
            "task": "...",
            "solution": "...",
            "outcome": "success/failure",
            "timestamp": "..."
        }
    """
    self.long_term.append(experience)

def retrieve_relevant(self, query, k=3):
    """
    从长期记忆中检索相关经验

    Args:
        query: 当前任务/问题
        k: 返回top-k相关记忆

    Returns:
        relevant_memories: 相关记忆列表
    """
    # 使用embedding计算相似度
    query_emb = get_embedding(query)
```

```
        scored_memories = []
        for memory in self.long_term:
            memory_emb = get_embedding(memory['task'])
            similarity = cosine_similarity(query_emb, memory_emb)
            scored_memories.append((memory, similarity))

        # 排序并返回top-k
        scored_memories.sort(key=lambda x: x[1], reverse=True)
        return [m[0] for m in scored_memories[:k]]

    def get_context(self):
        """
        获取当前上下文（用于LLM输入）
        """
        context = {
            "conversation": self.short_term[-10:],  # 最近10轮对话
            "task_state": self.working,
            "relevant_history": []  # 可选：相关历史
        }
        return context
```

9.2.4 工具调用（Tool Calling）

工具定义：

```
from typing import Callable, Dict, Any

class Tool:
    """
    工具的抽象基类
    """
    def __init__(
        self,
        name: str,
        description: str,
        func: Callable,
        parameters: Dict[str, Any]
    ):
        self.name = name
        self.description = description
        self.func = func
        self.parameters = parameters

    def run(self, **kwargs):
        """执行工具"""
        return self.func(**kwargs)
```

```
    def get_schema(self):
        """
        返回工具的JSON Schema（用于LLM理解）
        """
        return {
            "name": self.name,
            "description": self.description,
            "parameters": self.parameters
        }


# 定义具体工具
def search_web(query: str) -> str:
    """在网上搜索信息"""
    # 实际实现：调用搜索API
    results = google_search(query)
    return f"搜索结果：{results}"

search_tool = Tool(
    name="search",
    description="在互联网上搜索信息",
    func=search_web,
    parameters={
        "type": "object",
        "properties": {
            "query": {
                "type": "string",
                "description": "搜索关键词"
            }
        },
        "required": ["query"]
    }
)


def calculate(expression: str) -> float:
    """计算数学表达式"""
    try:
        result = eval(expression)  # 注意：生产环境需要安全的eval
        return result
    except Exception as e:
        return f"计算错误：{e}"

calculator_tool = Tool(
    name="calculator",
```

```
    description="计算数学表达式",
    func=calculate,
    parameters={
        "type": "object",
        "properties": {
            "expression": {
                "type": "string",
                "description": "数学表达式，如 '2 + 3 * 4'"
            }
        },
        "required": ["expression"]
    }
)
```

9.3 ReAct框架

9.3.1 ReAct原理

ReAct = Reasoning + Acting

核心思想：交替进行**推理**和**行动**。

```
Thought（思考）-> Action（行动）-> Observation（观察）-> Thought -> ...
```

示例流程：

```
问题：2023年图灵奖得主是谁？他们在哪个大学工作？

Thought 1: 我需要先搜索2023年图灵奖得主的信息
Action 1: search("2023年图灵奖得主")
Observation 1: Avi Wigderson获得2023年图灵奖

Thought 2: 现在我需要查询Avi Wigderson在哪个大学工作
Action 2: search("Avi Wigderson university")
Observation 2: Avi Wigderson是普林斯顿高等研究院的教授

Thought 3: 我已经获得了所需的所有信息
Answer: 2023年图灵奖得主是Avi Wigderson，他在普林斯顿高等研究院工作。
```

9.3.2 ReAct实现

```
class ReActAgent:
    """
    ReAct框架的Agent实现
```

```
    """
    def __init__(self, llm, tools, max_iterations=5):
        self.llm = llm
        self.tools = {tool.name: tool for tool in tools}
        self.max_iterations = max_iterations

    def run(self, task):
        """
        执行任务

        Args:
            task: 用户任务

        Returns:
            final_answer: 最终答案
        """
        conversation_history = []

        for i in range(self.max_iterations):
            # 构造prompt
            prompt = self._build_prompt(task, conversation_history)

            # LLM生成响应
            response = self.llm.generate(prompt)

            # 解析响应
            action_type, content = self._parse_response(response)

            if action_type == "Thought":
                conversation_history.append({"type": "Thought", "content": content}

            elif action_type == "Action":
                # 执行工具
                tool_name, tool_args = self._parse_action(content)

                if tool_name not in self.tools:
                    observation = f"错误：工具 {tool_name} 不存在"
                else:
                    try:
                        observation = self.tools[tool_name].run(**tool_args)
                    except Exception as e:
                        observation = f"执行错误：{e}"

                conversation_history.append({
                    "type": "Action",
                    "content": content
```

```
                })
                conversation_history.append({
                    "type": "Observation",
                    "content": observation
                })

            elif action_type == "Answer":
                # 得到最终答案
                return content

        return "未能在规定步骤内完成任务"

    def _build_prompt(self, task, history):
        """
        构造ReAct的prompt
        """
        tools_desc = "\n".join([
            f"{name}: {tool.description}"
            for name, tool in self.tools.items()
        ])

        history_text = "\n".join([
            f"{item['type']}: {item['content']}"
            for item in history
        ])

        prompt = f"""
你是一个问题解决助手，使用ReAct框架完成任务。

可用工具：
{tools_desc}

使用以下格式：
Thought: 你的推理过程
Action: tool_name(arg1="value1", arg2="value2")
Observation: 工具返回的结果
... (重复Thought/Action/Observation)
Answer: 最终答案

任务：{task}

{history_text}

下一步：
"""
        return prompt
```

```python
    def _parse_response(self, response):
        """
        解析LLM响应，提取类型和内容
        """
        lines = response.strip().split('\n')
        first_line = lines[0]

        if first_line.startswith("Thought:"):
            return "Thought", first_line[8:].strip()
        elif first_line.startswith("Action:"):
            return "Action", first_line[7:].strip()
        elif first_line.startswith("Answer:"):
            return "Answer", first_line[7:].strip()
        else:
            return "Thought", response

    def _parse_action(self, action_str):
        """
        解析Action字符串

        例如: "search(query='2023 图灵奖')"
        """
        import re

        # 提取工具名和参数
        match = re.match(r'(\w+)\((.*)\)', action_str)
        if not match:
            return None, {}

        tool_name = match.group(1)
        args_str = match.group(2)

        # 简单的参数解析（生产环境需要更robust的解析）
        args = {}
        for arg in args_str.split(','):
            if '=' in arg:
                key, value = arg.split('=', 1)
                key = key.strip()
                value = value.strip().strip('"').strip("'")
                args[key] = value

        return tool_name, args


# 使用示例
```

```
agent = ReActAgent(
    llm=your_llm,
    tools=[search_tool, calculator_tool],
    max_iterations=5
)

answer = agent.run("2023年图灵奖得主是谁？他们在哪个大学工作？")
print(answer)
```

9.4 Function Calling

9.4.1 OpenAI Function Calling

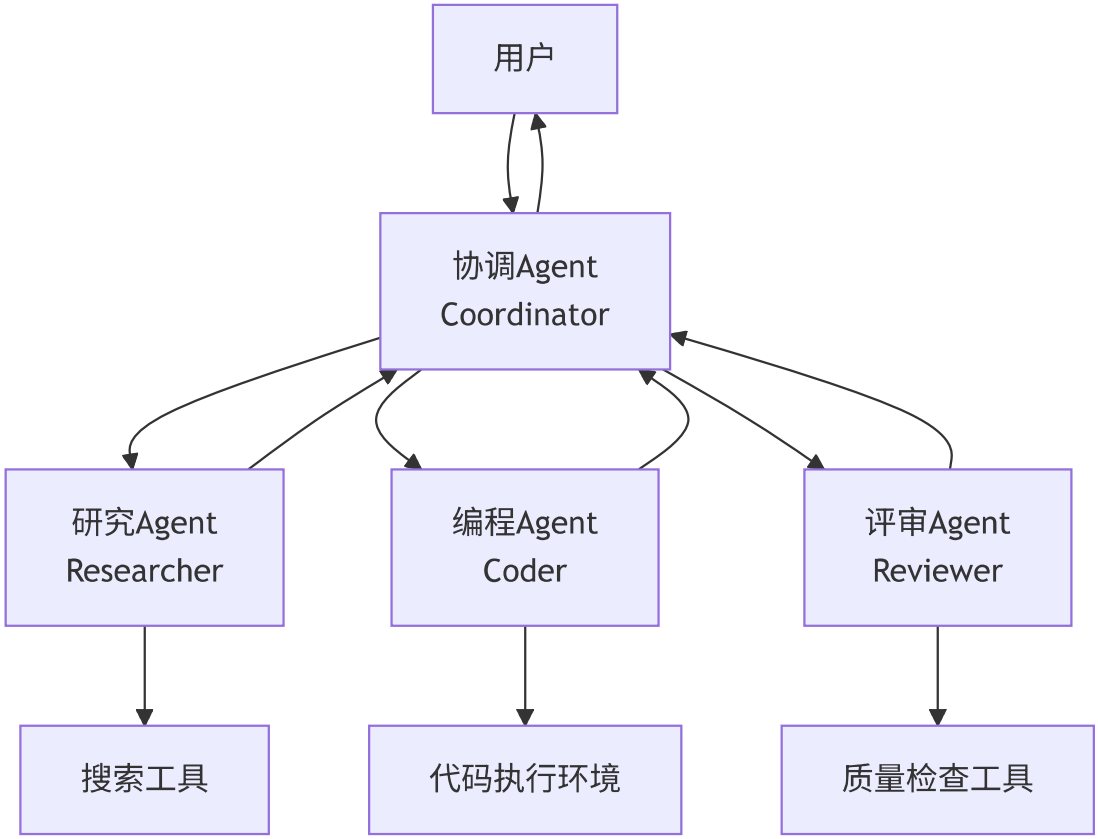
```
from openai import OpenAI

client = OpenAI()

# 定义函数
functions = [
    {
        "name": "get_weather",
        "description": "获取指定城市的天气",
        "parameters": {
            "type": "object",
            "properties": {
                "city": {
                    "type": "string",
                    "description": "城市名称，如'北京'、'上海'"
                },
                "unit": {
                    "type": "string",
                    "enum": ["celsius", "fahrenheit"],
                    "description": "温度单位"
                }
            },
            "required": ["city"]
        }
    },
    {
        "name": "search_knowledge_base",
        "description": "在知识库中搜索信息",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
```

```
                    "type": "string",
                    "description": "搜索查询"
                }
            },
            "required": ["query"]
        }
    }
]

# 实际的函数实现
def get_weather(city, unit="celsius"):
    """获取天气（模拟）"""
    return f"{city}的天气：晴，温度25{unit}"

def search_knowledge_base(query):
    """搜索知识库（模拟）"""
    return f"关于'{query}'的信息：..."


# Agent主循环
def run_agent_with_function_calling(user_message):
    """
    使用Function Calling的Agent
    """
    messages = [{"role": "user", "content": user_message}]

    while True:
        # 调用LLM
        response = client.chat.completions.create(
            model="gpt-4",
            messages=messages,
            functions=functions,
            function_call="auto"
        )

        message = response.choices[0].message

        # 检查是否需要调用函数
        if message.function_call:
            # 提取函数名和参数
            function_name = message.function_call.name
            function_args = json.loads(message.function_call.arguments)

            print(f"调用函数: {function_name}({function_args})")

            # 执行函数
```

```
            if function_name == "get_weather":
                function_response = get_weather(**function_args)
            elif function_name == "search_knowledge_base":
                function_response = search_knowledge_base(**function_args)
            else:
                function_response = "未知函数"

            # 将函数结果添加到对话
            messages.append({
                "role": "assistant",
                "content": None,
                "function_call": {
                    "name": function_name,
                    "arguments": json.dumps(function_args)
                }
            })
            messages.append({
                "role": "function",
                "name": function_name,
                "content": function_response
            })

        else:
            # 没有函数调用，返回最终答案
            return message.content


# 使用
answer = run_agent_with_function_calling("北京明天天气怎么样？")
print(answer)
```

9.5 多Agent协作

9.5.1 多Agent架构



9.5.2 AutoGen风格的多Agent

```
class Agent:
    """
    基础Agent类
    """
    def __init__(self, name, role, llm, tools=None):
        self.name = name
        self.role = role
        self.llm = llm
        self.tools = tools or []
        self.message_history = []

    def receive_message(self, message, sender):
        """
        接收消息
        """
        self.message_history.append({
            "from": sender,
            "content": message
        })

    def generate_response(self):
        """
        生成响应
```

```
        """
        context = self._build_context()
        response = self.llm.generate(context)
        return response

    def _build_context(self):
        """
        构建上下文
        """
        history_text = "\n".join([
            f"{msg['from']}: {msg['content']}"
            for msg in self.message_history[-10:]
        ])

        prompt = f"""
你是{self.name}，角色是{self.role}。

对话历史：
{history_text}

请根据你的角色，给出回应：
"""
        return prompt


class MultiAgentSystem:
    """
    多Agent协作系统
    """
    def __init__(self, agents):
        self.agents = {agent.name: agent for agent in agents}

    def run_conversation(self, initial_message, max_turns=10):
        """
        运行多Agent对话

        Args:
            initial_message: 初始任务
            max_turns: 最大对话轮数
        """
        conversation_log = []

        # 从coordinator开始
        current_speaker = "Coordinator"
        message = initial_message
```

```
        for turn in range(max_turns):
            agent = self.agents[current_speaker]

            # Agent接收消息
            agent.receive_message(message, sender="System" if turn == 0 else previc

            # Agent生成响应
            response = agent.generate_response()

            conversation_log.append({
                "turn": turn,
                "speaker": current_speaker,
                "message": response
            })

            # 决定下一个发言者
            next_speaker = self._decide_next_speaker(response, current_speaker)

            if next_speaker == "FINISH":
                break

            previous_speaker = current_speaker
            current_speaker = next_speaker
            message = response

        return conversation_log

    def _decide_next_speaker(self, message, current_speaker):
        """
        决定下一个发言者
        """
        # 简化版本：从消息中提取
        if "FINISH" in message:
            return "FINISH"
        elif "@Researcher" in message:
            return "Researcher"
        elif "@Coder" in message:
            return "Coder"
        elif "@Reviewer" in message:
            return "Reviewer"
        else:
            return "Coordinator"


# 创建多Agent系统
coordinator = Agent(
```

```
    name="Coordinator",
    role="协调各个Agent，分配任务",
    llm=llm
)

researcher = Agent(
    name="Researcher",
    role="负责研究和信息收集",
    llm=llm,
    tools=[search_tool]
)

coder = Agent(
    name="Coder",
    role="负责编写代码",
    llm=llm,
    tools=[code_executor]
)

reviewer = Agent(
    name="Reviewer",
    role="负责代码审查和质量检查",
    llm=llm
)

system = MultiAgentSystem([coordinator, researcher, coder, reviewer])

# 运行任务
task = "开发一个爬虫程序，爬取新闻网站的最新文章"
conversation = system.run_conversation(task)

for entry in conversation:
    print(f"Turn {entry['turn']} - {entry['speaker']}:")
    print(entry['message'])
    print("-" * 50)
```

9.6 实战案例

9.6.1 数据分析Agent

```
class DataAnalysisAgent:
    """
    数据分析Agent
    """
    def __init__(self, llm):
```

```
        self.llm = llm
        self.tools = {
            "read_csv": self._read_csv,
            "plot": self._plot,
            "calculate_stats": self._calculate_stats,
            "run_sql": self._run_sql
        }
        self.data = None

    def analyze(self, dataset_path, question):
        """
        分析数据集并回答问题
        """
        # 1. 加载数据
        self.data = pd.read_csv(dataset_path)
        data_info = self._get_data_info()

        # 2. 生成分析计划
        plan = self._create_analysis_plan(question, data_info)

        # 3. 执行计划
        results = []
        for step in plan:
            result = self._execute_step(step)
            results.append(result)

        # 4. 生成最终报告
        report = self._generate_report(question, results)

        return report

    def _create_analysis_plan(self, question, data_info):
        """
        创建分析计划
        """
        prompt = f"""
你是数据分析专家。给定数据集信息和问题，创建分析计划。

数据集信息:
{data_info}

问题: {question}

请创建步骤化的分析计划（JSON格式）:
{{
    "steps": [
```

```
        {{"action": "calculate_stats", "params": {{"column": "..."}}]}}
    ]
}}

计划：
"""
        plan_json = self.llm.generate(prompt)
        return json.loads(plan_json)['steps']

    def _execute_step(self, step):
        """
        执行单个分析步骤
        """
        action = step['action']
        params = step.get('params', {})

        if action in self.tools:
            return self.tools[action](**params)
        else:
            return f"未知操作：{action}"

    def _read_csv(self, path):
        """读取CSV"""
        self.data = pd.read_csv(path)
        return f"已加载数据，共{len(self.data)}行"

    def _calculate_stats(self, column):
        """计算统计信息"""
        stats = self.data[column].describe().to_dict()
        return stats

    def _plot(self, x, y, plot_type="scatter"):
        """绘图"""
        plt.figure(figsize=(10, 6))
        if plot_type == "scatter":
            plt.scatter(self.data[x], self.data[y])
        elif plot_type == "line":
            plt.plot(self.data[x], self.data[y])
        plt.xlabel(x)
        plt.ylabel(y)
        plt.savefig('plot.png')
        return "图表已保存到plot.png"

    def _run_sql(self, query):
        """执行SQL查询"""
        # 使用pandasql或DuckDB
```

```
        import pandasql as ps
        result = ps.sqldf(query, {"data": self.data})
        return result.to_dict()
```

9.6.2 个人助理Agent

```
class PersonalAssistantAgent:
    """
    个人助理Agent
    """
    def __init__(self, llm):
        self.llm = llm
        self.calendar = []  # 日程表
        self.reminders = []  # 提醒事项
        self.preferences = {}  # 用户偏好

        self.tools = {
            "check_calendar": self._check_calendar,
            "add_event": self._add_event,
            "set_reminder": self._set_reminder,
            "search_web": self._search_web,
            "send_email": self._send_email
        }

    def handle_request(self, user_input):
        """
        处理用户请求
        """
        # 1. 理解用户意图
        intent = self._understand_intent(user_input)

        # 2. 执行相应操作
        if intent['type'] == 'schedule':
            return self._handle_scheduling(intent)
        elif intent['type'] == 'query':
            return self._handle_query(intent)
        elif intent['type'] == 'reminder':
            return self._handle_reminder(intent)
        else:
            return self._general_response(user_input)

    def _understand_intent(self, user_input):
        """
        理解用户意图
        """
```

```
        prompt = f"""
分析以下用户输入的意图：

用户输入：{user_input}

输出JSON格式：
{{
    "type": "schedule/query/reminder/general",
    "entities": {{
        "time": "...",
        "event": "...",
        ...
    }}
}}

分析结果：
"""
        intent_json = self.llm.generate(prompt)
        return json.loads(intent_json)

    def _handle_scheduling(self, intent):
        """
        处理日程安排
        """
        # 检查时间冲突
        conflicts = self._check_conflicts(intent['entities']['time'])

        if conflicts:
            return f"时间冲突：{conflicts}。是否要重新安排？"
        else:
            self._add_event(
                time=intent['entities']['time'],
                event=intent['entities']['event']
            )
            return f"已添加：{intent['entities']['event']}"

    def _check_calendar(self, date=None):
        """检查日程"""
        if date is None:
            date = datetime.now().date()

        events = [e for e in self.calendar if e['date'] == date]
        return events

    def _add_event(self, time, event):
        """添加事件"""
```

```
        self.calendar.append({
            "time": time,
            "event": event,
            "created_at": datetime.now()
        })
```

9.7 面试高频问题

Q1: Agent和传统对话系统的本质区别?

答案要点:

维度	对话系统	Agent
主动性	被动响应	**主动规划和执行**
工具使用	无	**调用外部工具**
状态管理	简单上下文	**复杂状态和记忆**
目标导向	单轮问答	**多步骤达成目标**

关键: Agent具有**自主性**和**工具使用能力**。

Q2: ReAct框架为什么有效?

答案:

1. **交替推理和行动**: 避免盲目行动
2. **可解释性**: 推理过程可见
3. **错误纠正**: 观察结果后可调整策略
4. **类似人类思维**: Think -> Act -> Observe

对比:

- 纯推理: 无法获取外部信息
- 纯行动: 缺乏规划，效率低
- ReAct: 结合两者优势

Q3: 如何设计Agent的工具集?

设计原则:

1. **原子性**: 每个工具做一件事
2. **可组合**: 工具可以组合使用
3. **清晰描述**: 让LLM理解工具用途
4. **错误处理**: 工具应返回清晰的错误信息

示例:

```
# ✅ 好的设计
Tool(
    name="search_weather",
    description="查询指定城市的天气预报。返回温度、湿度、天气状况。",
    parameters={"city": "城市名称"}
)

# ❌ 不好的设计（功能不清晰）
Tool(
    name="get_info",
    description="获取信息",
    parameters={"query": "查询"}
)
```

Q4: 多Agent系统的通信机制?

常见模式:

1. **中心化**: Coordinator分配任务

```
User -> Coordinator -> Agent1/Agent2/... -> Coordinator -> User
```

2. **去中心化**: Agent直接通信

```
Agent1 <-> Agent2 <-> Agent3
```

3. **混合模式**: 结合两者

选择依据:

- 任务复杂度
- Agent数量
- 协作紧密程度

Q5: Agent的记忆管理策略?

三层记忆:

```
记忆结构 = {
    "短期记忆": {
        "内容": "当前对话",
        "容量": "有限（10-20轮）",
        "作用": "维持上下文连贯性"
    },
    "工作记忆": {
```

```
            "内容": "当前任务状态",
            "容量": "任务相关",
            "作用": "跟踪任务进度"
        },
        "长期记忆": {
            "内容": "重要经验、知识",
            "容量": "无限（向量数据库）",
            "作用": "学习和改进"
        }
    }
```

检索策略：

- 短期：FIFO队列
- 工作：任务相关
- 长期：向量相似度检索

9.8 本章小结

本章全面介绍了Agent系统：

✅ **核心概念**：Agent vs 对话系统 ✅ **关键组件**：规划、记忆、工具调用 ✅ **ReAct框架**：推理和行动结合 ✅ **Function Calling**：OpenAI的实现方式 ✅ **多Agent协作**：分工合作完成复杂任务 ✅ **实战案例**：数据分析、个人助理

关键要点：

- Agent的核心是**自主性**和**工具使用**
- ReAct是最经典的Agent框架
- 记忆管理对长期任务至关重要
- 多Agent适合复杂、多步骤任务

下一章预告： 第10章将讲解模型部署与服务化。