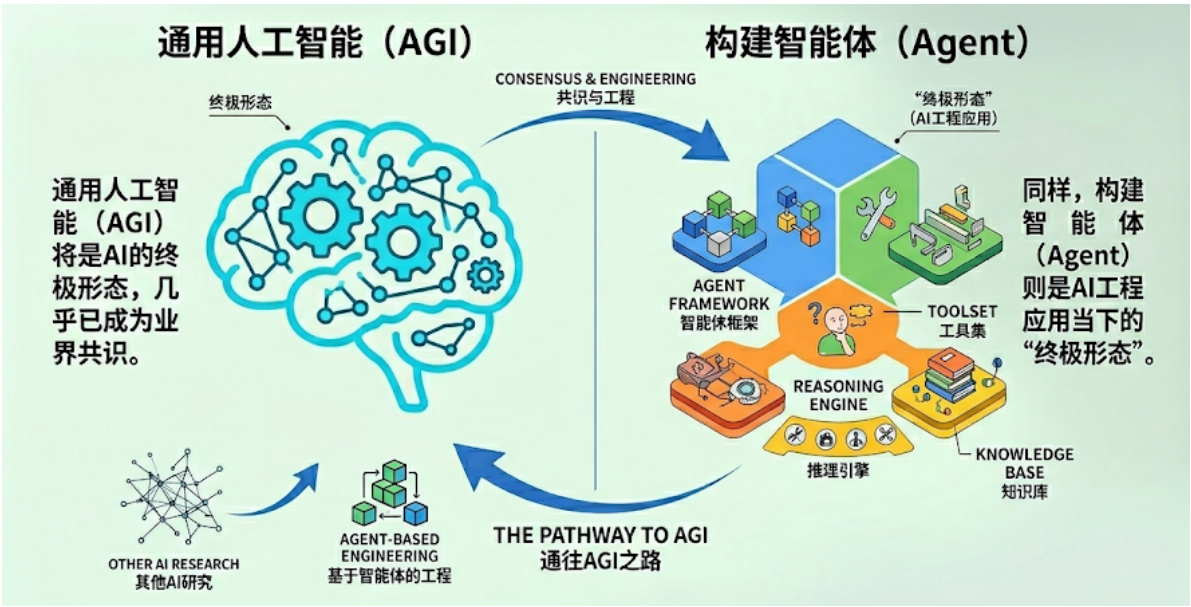
第07章：智能体

讲师：尚硅谷-宋红康

官网：尚硅谷

1、理解Agents

通用人工智能（AGI）将是 AI的终极形态 ，几乎已成为业界共识。同样，构建智能体（Agent）则是AI工程应用 当下的“终极形态” ，即Agent是大模型应用开发的核心。



1.1 什么是Agent?

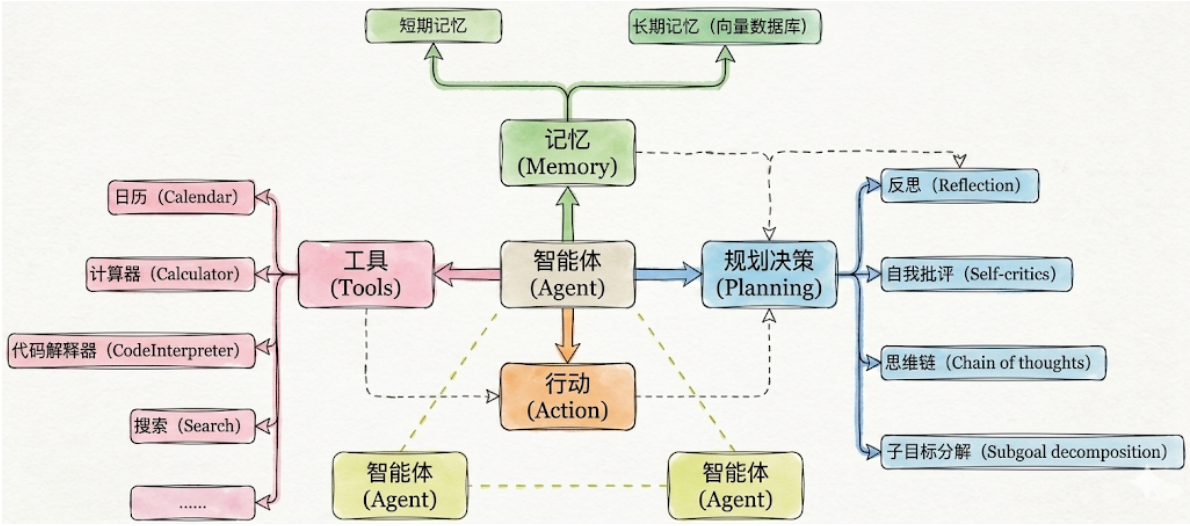
在大模型应用开发中，智能体通常指一种以 大语言模型为推理与决策核心 ，结合 记忆 、 工具调用 与环境交互能力，能够进行 规划决策 并执行 复杂任务 以达成目标的软件系统。

Agent的关键能力

- 理解用户问题
- 如何 拆解任务
- 判断 是否需要工具
- 需要 调用哪些工具
- 如何利用好 工具结果 生成回答&推进任务

1.3 Agent的核心组件

前面讲过现在AI Agent的架构：



实际开发中几个要素并不需要同时出现，一句话总结

- 必须的：**行动**（Action）
- 几乎总是存在的：**工具**（Tool）
- 有条件存在的：**规划决策**（Planning）
- 最容易被省略的：**记忆**（Memory）

1.4 Agent创建与调用

1.4.1 历史上的调用

在 LangChain 0.x 时代，框架内的 Agent 系统经历了"碎片化"阶段。当时的设计理念是 **"针对场景设计特定 Agent"**：

- 如果你要实现思维链推理（ReAct），就用 `create_react_agent`；
- 如果需要结构化输出，就用 `create_structured_chat_agent`；
- 要工具调用，则用 `create_tool_calling_agent`。

举例：❌ v0.x 的复杂方式

```python
# 需要多个步骤
from langchain_openai import ChatOpenAI
from langchain.agents import AgentExecutor, create_react_agent
from langchain_core.prompts import PromptTemplate

# 1. 模型初始化
model = ChatOpenAI(model="gpt-4o-mini")

# 2. 创建提示词模板
prompt = PromptTemplate.from_template("""
You are a helpful assistant.

Tools: {tools}
Tool Names: {tool_names}

{agent_scratchpad}
""")

# 3. 创建 agent
agent = create_react_agent(
    llm=model,
    tools=tools,
```

```
    prompt=prompt
)

# 4. 创建 executor
executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=True
)

# 5. 调用
result = executor.invoke({"input": "问题"})

```

这种方式灵活，但也带来了三个明显问题：

1. **心智负担高**——每种 Agent 都要单独记忆 API 与参数；
2. **可组合性差**——多个 Agent 之间无法统一调度；
3. **生态碎片化**——不同模块难以复用或协同演化。

1.4.2 全新的调用

LangChain 在 1.0 版本后，团队做出了彻底重构：将所有 Agent 的创建方式**统一为一个入口：create_agent()**。它取代了旧版本中的 `create_react_agent`、`create_json_agent`、`create_tool_calling_agent` 等多种分支函数，真正让开发者用一行代码即可创建任何类型的智能体。

同时在底层通过“中间件机制（Middleware）”和“标准模型接口（invoke / stream）”实现全局统一。这让框架更轻、更稳，也更易于被集成到其他 Agent 平台中。

举例：☑ v1.x 的简洁方式：

```
from langchain.chat_models import init_chat_model
from langchain.agents import create_agent

# 1. 初始化模型
model = init_chat_model("gpt-4o-mini", model_provider="openai")

# 2. 创建 agent（一步完成）
agent = create_agent(
    model=model,
    tools=[tool1, tool2],
    system_prompt="Agent 的行为指令"  # 可选
)

# 3. 调用
result = agent.invoke({
    "messages": [{"role": "user", "content": "问题"}]
})
```

2、Agent的基本用法1：模型的传入方式

在 LangChain 1.2 中，`create_agent` 是构建智能体的核心方式，底层基于 `LangGraph` 实现。

create_agent 完整参数：

```python
from langchain.agents import create_agent

agent = create_agent(
    model: str | BaseChatModel,             # 必需：聊天模型
    tools: List[BaseTool],                  # 必需：工具列表
    *,
    system_prompt: str = "",                # 系统提示词
    middleware: Sequence[AgentMiddleware[StateT_co, ContextT]] = () # 中间件
    interrupt_before: List[str] = None,     # 在某些工具前暂停（人机协作）
    interrupt_after: List[str] = None,      # 在某些工具后暂停
    debug: bool = False                     # 调试模式
    name: str | None = None,                # 设置模型名称
)
```

Agent在创建时，涉及到 模型(Agent使用的模型) 、 可调用工具 、 系统提示词 等参数的设置。

更多参数参考：https://reference.langchain.com/python/langchain/agents/factory/create_agent

Agent中模型的传入方式：

本节我们只关注模型的传入。

模型是 Agent 的“大脑”，负责决策和推理。根据模型传入agent方式的不同，分为两种方式。

2.1 传入模型字符串

Agent根据传入的模型字符串，自主创建模型对象

```python
from langchain.agents import create_agent

from dotenv import load_dotenv

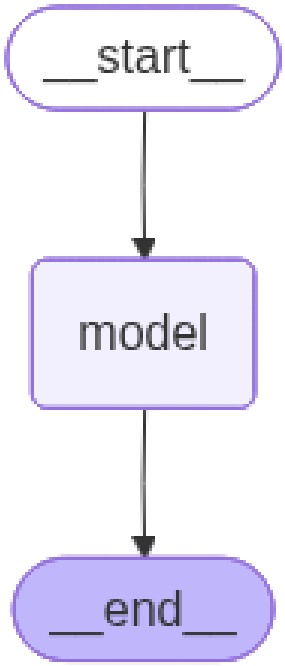
load_dotenv(override=True)

agent = create_agent("deepseek-v4-flash")
print(type(agent))

from IPython.display import Image, display
display(Image(agent.get_graph().draw_mermaid_png()))
```

输出

```
<class 'langgraph.graph.state.CompiledStateGraph'>
```



由上可知，agent本质上是LangGraph的**CompiledStateGraph**实例，底层实现是一个图结构。

通过上述代码最后一行可以看到agent的图结构。

2.2 传入模型对象

```python
from langchain.chat_models import init_chat_model
from langchain.agents import create_agent
from langchain_deepseek import ChatDeepSeek
from dotenv import load_dotenv
import os
load_dotenv(override=True)

# 以ChatDeepSeek为例
# model = ChatDeepSeek(model="deepseek-v4-flash")

# 以init_chat_model为例
model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

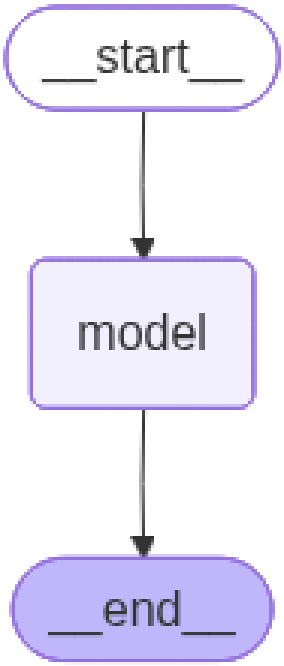

agent = create_agent(model)

print(type(agent))

from IPython.display import Image, display
display(Image(agent.get_graph().draw_mermaid_png()))
```

输出同上

```
<class 'langgraph.graph.state.CompiledStateGraph'>
```



3、Agent的基本用法2：如何调用Agent

`agent.invoke()` 是Agent 最基本的同步调用方法，它会阻塞程序执行直到返回最终结果。具体的：

- **输入：**传入的参数为字典类型，字典内通过 `messages字段传递消息列表` 。即：“ `{"messages": [{"role": "...", "content": "..."}]}` ”
- **输出：**通过invoke调用Agent，底层可能会经历多轮交互，返回的是完整的 `消息列表` ，被封装在字典中，是 `messages字段的值` 。

```
response = agent.invoke({"messages": [...]})

# response 是字典类型
{
    "messages": [
        HumanMessage(...),        # 用户问题
        AIMessage(...),           # AI 工具调用
        ToolMessage(...),         # 工具返回结果
        AIMessage(...)            # 最终回答 ← 通常取这个
    ]
}

# 获取最终回答
final_answer = response['messages'][-1].content
```

举例1：

```
from langchain.agents import create_agent
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os
load_dotenv(override=True)
from rich import print as rprint

# 以init_chat_model为例
model = init_chat_model(
    model="gpt-5.4-mini",
```

```
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

agent = create_agent(model=model)

response = agent.invoke({"messages": ["你好"]}) # 默认是HumanMessage
print(type(response))
rprint(response)
```

输出如下

```
{
 'messages': [
     HumanMessage(
         content='你好',
         additional_kwargs={},
         response_metadata={},
         id='93ffcb22-179a-4cab-81a7-a3bd3a1795d4'
     ),
     AIMessage(
         content='你好！有什么我可以帮你的吗？',
         additional_kwargs={'refusal': None},
         response_metadata={
             'token_usage': {
                 'completion_tokens': 13,
                 'prompt_tokens': 7,
                 'total_tokens': 20,
                 'completion_tokens_details': {
                     'accepted_prediction_tokens': 0,
                     'audio_tokens': 0,
                     'reasoning_tokens': 0,
                     'rejected_prediction_tokens': 0
                 },
                 'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                 'latency_checkpoint': {
                     'engine_tbt_ms': 3,
                     'engine_ttft_ms': 48,
                     'engine_ttlt_ms': 90,
                     'pre_inference_ms': 95,
                     'service_tbt_ms': 4,
                     'service_ttft_ms': 673,
                     'service_ttlt_ms': 712,
                     'total_duration_ms': 626,
                     'user_visible_ttft_ms': 578
                 }
             },
             'model_provider': 'openai',
             'model_name': 'gpt-5.4-mini-2026-03-17',
             'system_fingerprint': None,
             'id': 'chatcmpl-DlUJdEGrlJtKPHQ9wgdp1KsrEjIDc',
             'service_tier': 'default',
             'finish_reason': 'stop',
             'logprobs': None
         },
```

```
        id='lc_run--019e7ccd-c08b-7f60-8b07-32324cccf2c2-0',
        tool_calls=[],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 7,
            'output_tokens': 13,
            'total_tokens': 20,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    )
 ]
}
```

举例2：

invoke调用的核心就是输入一系列消息（messages），每条消息通常包含 role（如 "user", "assistant", "system", "tool"）和 "content"。

我们也可以在message列表的开头加入"system"角色的消息来定义Agent的行为。

```python
from langchain.agents import create_agent
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os
load_dotenv(override=True)
from rich import print as rprint

# 以init_chat_model为例
model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

agent = create_agent(model)

resp = agent.invoke( {
    "messages": [
        {"role": "system", "content": "你是一个小学数学老师，耐心，幽默，讲解深入浅出"},
        {"role": "user", "content": "100加上50等于多少？"}
    ]
})

rprint(resp)
```

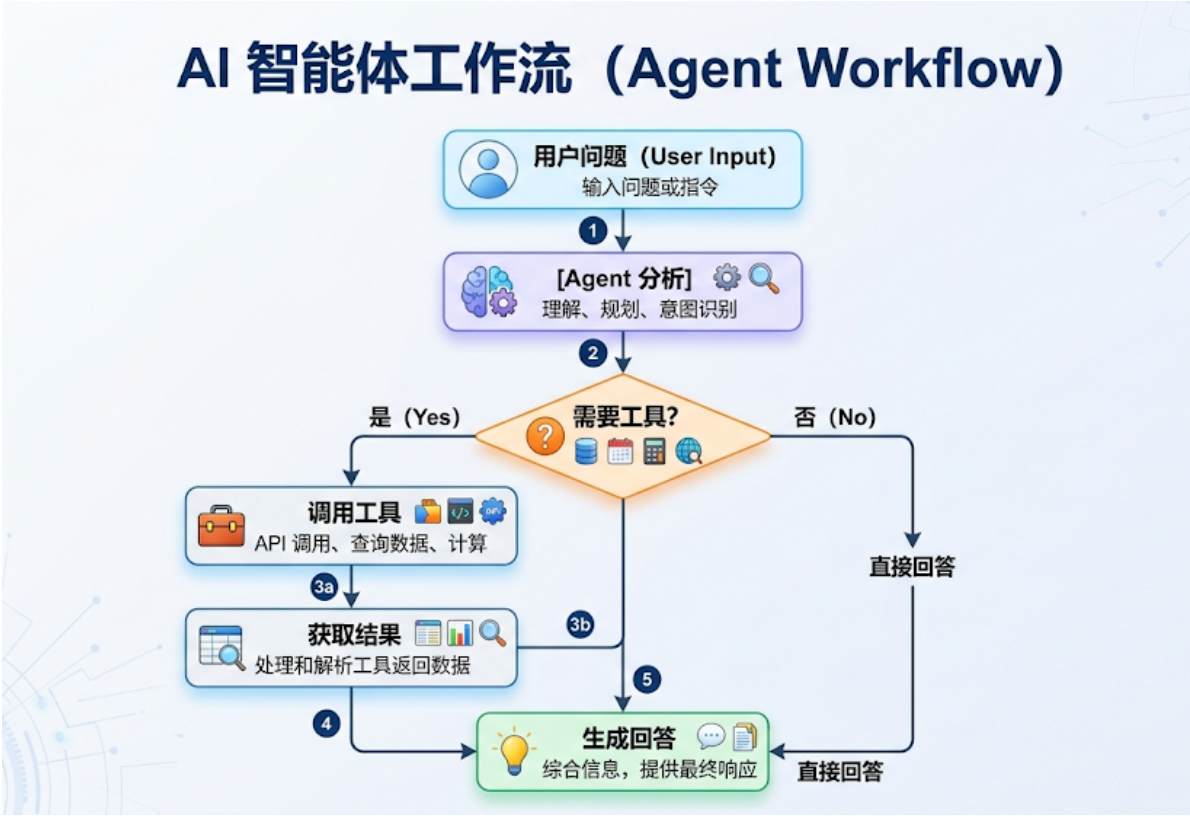
```
{
 'messages': [
     SystemMessage(
         content='你是一个小学数学老师，耐心，幽默，讲解深入浅出',
         additional_kwargs={},
         response_metadata={},
         id='6e02c29f-0360-499a-9d31-f6fcf5921310'
     ),
     HumanMessage(
```

```
        content='100加上50等于多少？',
        additional_kwargs={},
        response_metadata={},
        id='7a70d206-28d9-4081-9bb3-2f387aba2fd3'
    ),
    AIMessage(
        content='100 加上 50 等于 **150**。  \n\n可以这样想：  \n- 100 是 1 个百  \n- 再加 50，也就是 5 个十
\n- 合起来就是 **150**\n\n如果你愿意，我还可以教你怎么用“数数法”快速算这种题。',
        additional_kwargs={'refusal': None},
        response_metadata={
            'token_usage': {
                'completion_tokens': 71,
                'prompt_tokens': 36,
                'total_tokens': 107,
                'completion_tokens_details': {
                    'accepted_prediction_tokens': 0,
                    'audio_tokens': 0,
                    'reasoning_tokens': 0,
                    'rejected_prediction_tokens': 0
                },
                'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0},
                'latency_checkpoint': {
                    'engine_tbt_ms': 6,
                    'engine_ttft_ms': 33,
                    'engine_ttlt_ms': 426,
                    'pre_inference_ms': 75,
                    'service_tbt_ms': 6,
                    'service_ttft_ms': 253,
                    'service_ttlt_ms': 638,
                    'total_duration_ms': 572,
                    'user_visible_ttft_ms': 178
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DlUOHPXmUQEGFyu46rJf2kmbbjWrG',
            'service_tier': 'default',
            'finish_reason': 'stop',
            'logprobs': None
        },
        id='lc_run--019e7cd2-260e-7ae0-a8c0-e247fbdf767e-0',
        tool_calls=[],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 36,
            'output_tokens': 71,
            'total_tokens': 107,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    )
 ]
}
```

4、Agent的基本用法3：绑定工具

只有接入了一些工具，create_agent完成Agent创建才算完整。

Agent支持 静态 和 动态 绑定工具，后者需要用到**中间件**，后面会讲。在执行时：



这里的工具可以是LangChain内置的，也可以是自定义的。LangChain生态中已经内置集成了非常多的实用工具，开发者可以快速调用这些工具完成更加复杂工作流的开发。

LangChain内置工具列表：https://docs.langchain.com/oss/python/integrations/tools

其中典型的工具如下：

领域	工具名称	功能说明
搜索引擎与信息检索	🔹 DuckDuckGo Search	免费搜索引擎，返回网页标题、摘要与链接，常用于联网问答。
	🔹 Brave Search API	提供隐私友好的网页搜索结果，可配置输出深度与过滤。
	🔹 Bing Search Tool	微软 Bing 搜索接口，支持精确检索与多语言查询。
网页浏览与内容提取	🔹 Playwright Browser Toolkit	提供浏览器自动化访问、网页截图与文本抓取功能。
	🔹 Selenium Toolkit	类似于 Playwright，但偏向传统网页 DOM 操控。
生产力工具集成	🔹 Slack Toolkit	允许 Agent 在 Slack 中发送、读取或总结消息。
	🔹 Jira Toolkit	用于查询、创建或更新 Jira 项目工单。
	🔹 Google Drive / Docs Toolkit	操作 Google 文档、表格与文件搜索。
数据库与数据操作	🔹 SQLDatabase Toolkit	连接 SQL 数据库，自动生成并执行 SQL 查询。
	🔹 Faiss / Chroma / Milvus VectorStore Tools	进行向量相似度搜索，用于 RAG 检索。
代码与终端执行	🔹 Python REPL Tool	执行 Python 代码片段并返回结果（沙盒环境）。
	🔹 Shell Tool (Bash)	允许执行 Shell 命令（需安全限制）。
	🔹 Requests Wrapper Tool	发起 HTTP 请求、API 调用，用于外部数据交互。
多模态工具	🔹 DALL-E Tool	生成图像，可结合文本描述创作内容。
	🔹 SerpAPI Image Search	图片搜索引擎，可返回图片 URL 与元信息。
	🔹 OpenAI Whisper Tool	音频转录工具，将语音文件转换为文本。
知识问答与嵌入检索	🔹 VectorStoreRetrieverTool	基于嵌入模型检索文档片段，用于 RAG 问答。
	🔹 MultiQueryRetrieverTool	生成多种查询变体以提升检索覆盖率。
云与AI服务	🔹 OpenAI Functions / Azure Functions Tool	让 Agent 调用 OpenAI 或 Azure 定义的函数。
	🔹 Anthropic ToolKit	集成 Claude 模型工具调用能力。
文件系统与数据处理	🔹 CSV Toolkit	读取、过滤与分析 CSV 文件数据。
	🔹 Pandas DataFrame Agent Tool	直接在 DataFrame 上执行查询与统计任务。

4.1 基本用法

Agents支持绑定一或多个工具。

举例1：绑定一个工具

调用查询天气工具进行天气查询

```python
from langchain.agents import create_agent
from langchain.tools import tool
from rich import print as rprint


@tool(parse_docstring=True)
def get_weather(city: str) -> str:
    """
    天气查询工具

    Args:
        city: 城市名称
    """
```

```python
    return f"{city}的天气为晴朗，25°C。"


agent = create_agent(
    model = model,
    tools=[get_weather]
)

resp = agent.invoke( {
    "messages": [
        {"role": "system", "content": "你是一个天气查询助手，只回答天气相关的问题，其他问题请直接回答：我不清楚这问题答案。"},
        {"role": "user", "content": "北京的天气怎么样？"}
        # {"role": "user", "content": "100加上50等于多少？"}
    ]
})

rprint(resp)
```

以上代码运行结果如下：

```
{
 'messages': [
     SystemMessage(
         content='你是一个天气查询助手，只回答天气相关的问题，其他问题请直接回答：我不清楚这问题答案。',
         additional_kwargs={},
         response_metadata={},
         id='6f754fe0-7cd2-4e6b-a3a0-7592a71a7d20'
     ),
     HumanMessage(
         content='北京的天气怎么样？',
         additional_kwargs={},
         response_metadata={},
         id='ecf89457-2901-41d3-b960-ac8f7484d48e'
     ),
     AIMessage(
         content='',
         additional_kwargs={'refusal': None},
         response_metadata={
             'token_usage': {
                 'completion_tokens': 18,
                 'prompt_tokens': 162,
                 'total_tokens': 180,
                 'completion_tokens_details': {
                     'accepted_prediction_tokens': 0,
                     'audio_tokens': 0,
                     'reasoning_tokens': 0,
                     'rejected_prediction_tokens': 0
                 },
                 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0},
                 'latency_checkpoint': {
                     'engine_tbt_ms': 3,
                     'engine_ttft_ms': 48,
                     'engine_ttlt_ms': 110,
                     'pre_inference_ms': 102,
```

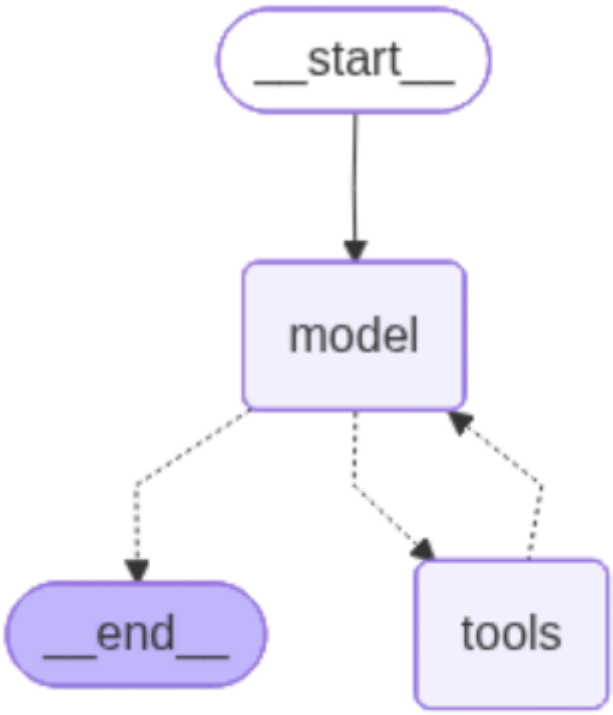
```
                    'service_tbt_ms': 4,
                    'service_ttft_ms': 312,
                    'service_ttlt_ms': 368,
                    'total_duration_ms': 274,
                    'user_visible_ttft_ms': 210
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DlavpIfl1rVj1nwpU8F39s8IAkexg',
            'service_tier': 'default',
            'finish_reason': 'tool_calls',
            'logprobs': None
        },
        id='lc_run--019e7e51-d485-7331-aa8b-cf03d9e23d7f-0',
        tool_calls=[
            {
                'name': 'get_weather',
                'args': {'city': '北京'},
                'id': 'call_4dp4PGy6Ghaj5m6WcNkrwKFe',
                'type': 'tool_call'
            }
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 162,
            'output_tokens': 18,
            'total_tokens': 180,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    ),
    ToolMessage(
        content='北京的天气为晴朗，25°C。',
        name='get_weather',
        id='b5b82403-973c-463b-b9f2-2aa76e0bff77',
        tool_call_id='call_4dp4PGy6Ghaj5m6WcNkrwKFe'
    ),
    AIMessage(
        content='北京天气晴朗，25°C。',
        additional_kwargs={'refusal': None},
        response_metadata={
            'token_usage': {
                'completion_tokens': 12,
                'prompt_tokens': 200,
                'total_tokens': 212,
                'completion_tokens_details': {
                    'accepted_prediction_tokens': 0,
                    'audio_tokens': 0,
                    'reasoning_tokens': 0,
                    'rejected_prediction_tokens': 0
                },
                'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                'latency_checkpoint': {
                    'engine_tbt_ms': 3,
                    'engine_ttft_ms': 34,
```

```
                    'engine_ttlt_ms': 71,
                    'pre_inference_ms': 94,
                    'service_tbt_ms': 3,
                    'service_ttft_ms': 272,
                    'service_ttlt_ms': 303,
                    'total_duration_ms': 219,
                    'user_visible_ttft_ms': 178
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DlavqO2DEdjb4JOBx3MTblngJKbTQ',
            'service_tier': 'default',
            'finish_reason': 'stop',
            'logprobs': None
        },
        id='lc_run--019e7e51-d9d7-7e81-bed0-cc990abfa293-0',
        tool_calls=[],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 200,
            'output_tokens': 12,
            'total_tokens': 212,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    )
 ]
}
```

进一步：

```
from IPython.display import Image, display

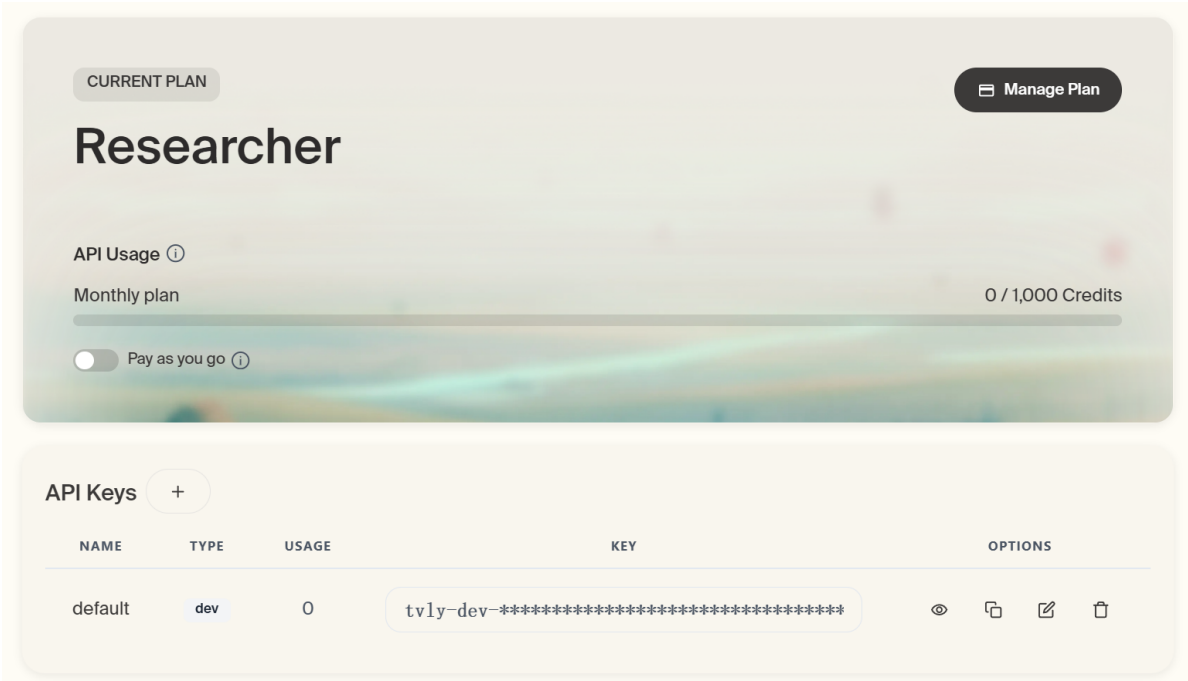
display(Image(agent.get_graph().draw_mermaid_png()))
```



举例2：接入内置工具

绑定内置的TavilySearch搜索工具，可以借助Tavily进行网络搜索和信息爬取。

这里我们需要先在tavily官网注册并获得API-KEY（每月有免费额度）：https://www.tavily.com/。



然后将API-KEY写到本地.env中的`TAVILY_API_KEY`变量中，即可进行调用了。

```python
from langchain_tavily import TavilySearch
from dotenv import load_dotenv
import os

load_dotenv(override=True)

web_search = TavilySearch(
    tavily_api_key=os.getenv("TAVILY_API_KEY"),
    max_results=2
)

#这是一个高度封装的网络搜索工具，可以直接调用：
web_search.invoke("请问2026年足球世界杯有哪些参赛队？")
```

```
{'query': '请问2026年足球世界杯有哪些参赛队？',
'follow_up_questions': None,
'answer': None,
'images': [],
'results': [{'url': 'https://www.instagram.com/p/DWk8mU-geNg',
'title': '你準備好了嗎？ 2026 FIFA世界盃™ 48支隊伍全數到齊 ... - Instagram',
'content': '本屆賽事由美國、加拿大、墨西哥三國聯合主辦，總共48 支球隊、104 場比賽、39 天賽程，這次的世界盃賽程本身就是前所未見的規模。',
'score': 0.9974885,
'raw_content': None},
{'url': 'https://zh.wikipedia.org/zh-hans/2026%E5%B9%B4%E5%9C%8B%E9%9A%9B%E8%B6%B3%E5%8D%94%E4%B8%96%E7%95%8C%E7%9B%83%E5%A4%96%E5%9C%8D%E8%B3%BD',
'title': '2026年国际足联世界杯预选赛 - 维基百科',
'content': '2026年国际足联世界杯预选赛是一项国家队足球预选赛赛事，以决定出哪些球队能够参与由加拿大、墨西哥和美国联合举办的2026年国际足联世界杯。本届世界杯决赛周名额增',
'score': 0.9974291,
'raw_content': None}],
'response_time': 0.79,
'request_id': 'dc00f93f-1b75-4222-86cc-4b6af567edf1'}
```

可以直接带入create_agent中作为外部工具。

```python
from langchain.chat_models import init_chat_model
from langchain.agents import create_agent
from langchain_tavily import TavilySearch
from dotenv import load_dotenv
import os
load_dotenv(override=True)
from rich import print as rprint

# 1.模型初始化
model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)
# 2.工具实例化
web_search = TavilySearch(max_results=2)

```

```python
# 3.创建Agent
agent = create_agent(
    model=model,
    tools=[web_search],
    #system_prompt="你是一名多才多艺的智能助手，可以调用工具帮助用户解决问题。"
)

# 4.运行Agent获得结果
result = agent.invoke(
    {"messages": [{"role": "user", "content": "请帮我查询2024年诺贝尔物理学奖得主是谁？"}]}
)

# rprint(result)
print(result['messages'][-1].content)
```

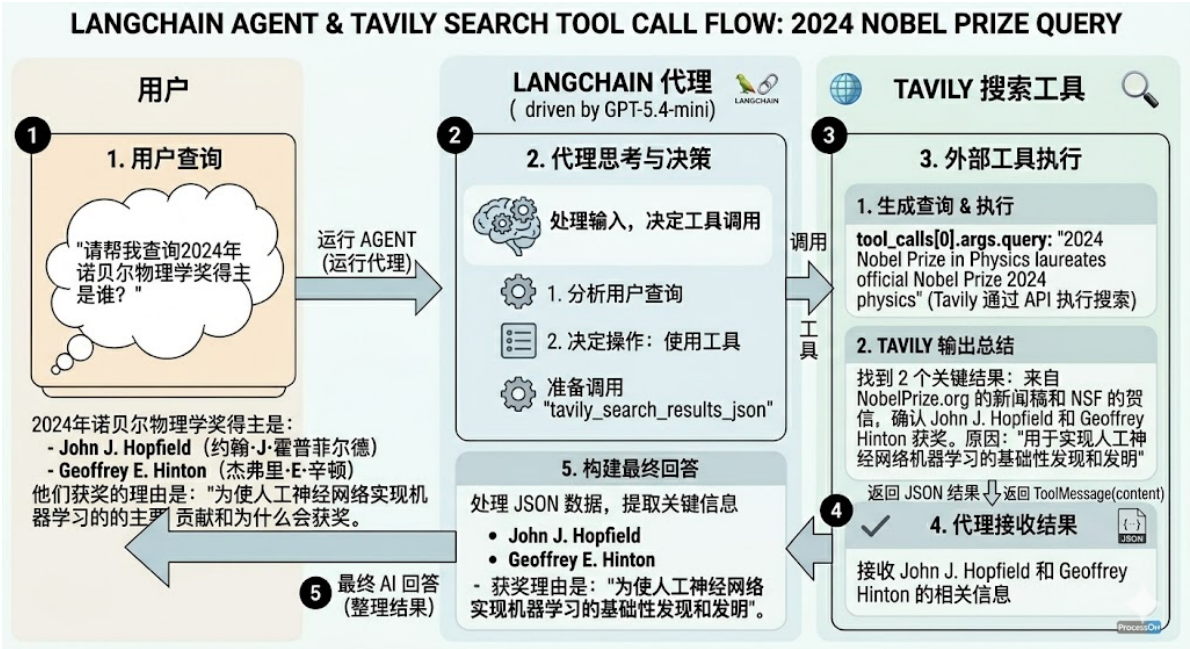
```
2024年诺贝尔物理学奖得主是：

- **John J. Hopfield**
- **Geoffrey E. Hinton**

授奖理由是：**“为实现机器学习的人工神经网络奠定基础性的发现和发明”**。
```

如果仔细观察本次运行过程，本次工具调用仍然是一次典型的Function calling执行流程，包括用户首次发起消息在内，总共创建了4条消息，分别是human message、ai message (涉及function call message）、tool message（涉及function response message）以及ai message（涉及final responses）。

一次完整的 Function calling 执行流程如下：



举例3：绑定多个工具

```python
from langchain.agents import create_agent
from langchain.tools import tool
from dotenv import load_dotenv
from rich import print as rprint
```

```python

load_dotenv()

@tool(parse_docstring=True)
def get_weather(city: str):
    """
    天气查询工具

    Args:
        city: 城市名称
    """
    return f"{city}今天天气挺好"

@tool(parse_docstring=True)
def get_news():
    """
    新闻查询工具
    """
    return "近期，受全球储蓄芯片短缺等多重因素影响，多地回收商称废旧手机回收市场迎来“火热潮”，回收价格普遍上涨，旧手机成“香饽饽”。"

agent = create_agent(
    model,
    tools=[get_weather, get_news]
)
response = agent.invoke({
    "messages": ["你好，杭州今天的天气如何？今天有哪些新闻？"]
})

rprint(response)

```

输出

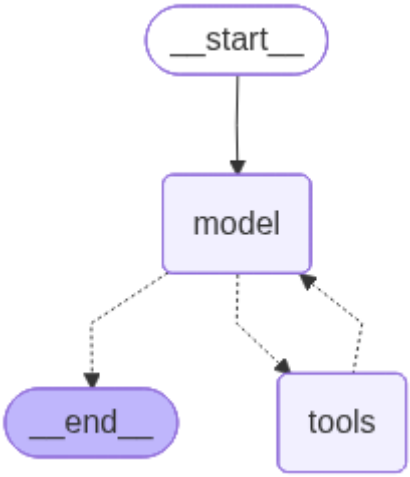
```
{
 'messages': [
     HumanMessage(
         content='你好，杭州今天的天气如何？今天有哪些新闻？',
         additional_kwargs={},
         response_metadata={},
         id='6f69ed6d-76fe-4d1d-9e31-c00a4bedf623'
     ),
     AIMessage(
         content='',
         additional_kwargs={'refusal': None},
         response_metadata={
             'token_usage': {
                 'completion_tokens': 46,
                 'prompt_tokens': 152,
                 'total_tokens': 198,
                 'completion_tokens_details': {
                     'accepted_prediction_tokens': 0,
                     'audio_tokens': 0,
                     'reasoning_tokens': 0,
                     'rejected_prediction_tokens': 0
                 },
```

```
                'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                'latency_checkpoint': {
                    'engine_tbt_ms': 4,
                    'engine_ttft_ms': 32,
                    'engine_ttlt_ms': 197,
                    'pre_inference_ms': 90,
                    'service_tbt_ms': 4,
                    'service_ttft_ms': 272,
                    'service_ttlt_ms': 429,
                    'total_duration_ms': 348,
                    'user_visible_ttft_ms': 182
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DlcRS3bbpm4iVX6laeXtWj2vYnYMm',
            'service_tier': 'default',
            'finish_reason': 'tool_calls',
            'logprobs': None
        },
        id='lc_run--019e7eaa-6712-7323-91e6-8af52084b894-0',
        tool_calls=[
            {
                'name': 'get_weather',
                'args': {'city': '杭州'},
                'id': 'call_w9ARuAgN8iqfG2GR7gvOOiBq',
                'type': 'tool_call'
            },
            {'name': 'get_news', 'args': {}, 'id':
'call_vOLekRymIme8bWuVQdIew4vu', 'type': 'tool_call'}
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 152,
            'output_tokens': 46,
            'total_tokens': 198,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    ),
    ToolMessage(
        content='杭州今天天气挺好',
        name='get_weather',
        id='21da1cc4-1d26-4d5b-a04a-a836644274fc',
        tool_call_id='call_w9ARuAgN8iqfG2GR7gvOOiBq'
    ),
    ToolMessage(
        content='近期，受全球储蓄芯片短缺等多重因素影响，多地回收商称废旧手机回
收市场迎来“火热潮”，回收价格普遍
上涨，旧手机成“香饽饽”。',
        name='get_news',
        id='d1fd947b-3d0f-47f5-9942-c4330f4d7fc6',
        tool_call_id='call_vOLekRymIme8bWuVQdIew4vu'
    ),
    AIMessage(
        content='杭州今天天气挺好。\n\n今天的新闻：\n1.
```

```
近期受全球储蓄芯片短缺等多重因素影响，多地回收商称废旧手机回收市场迎来“火热潮”，回收价格普遍上涨，旧手机成了“香饽饽
”。\n\n如果你愿意，我也可以帮你继续整理成“天气 + 新闻摘要”的简版。',
          additional_kwargs={'refusal': None},
          response_metadata={
              'token_usage': {
                  'completion_tokens': 91,
                  'prompt_tokens': 272,
                  'total_tokens': 363,
                  'completion_tokens_details': {
                      'accepted_prediction_tokens': 0,
                      'audio_tokens': 0,
                      'reasoning_tokens': 0,
                      'rejected_prediction_tokens': 0
                  },
                  'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                  'latency_checkpoint': {
                      'engine_tbt_ms': 5,
                      'engine_ttft_ms': 38,
                      'engine_ttlt_ms': 466,
                      'pre_inference_ms': 83,
                      'service_tbt_ms': 5,
                      'service_ttft_ms': 265,
                      'service_ttlt_ms': 686,
                      'total_duration_ms': 610,
                      'user_visible_ttft_ms': 183
                  }
              },
              'model_provider': 'openai',
              'model_name': 'gpt-5.4-mini-2026-03-17',
              'system_fingerprint': None,
              'id': 'chatcmpl-DlcRTgUZZHsugIYnroPNDk3PXXQwW',
              'service_tier': 'default',
              'finish_reason': 'stop',
              'logprobs': None
          },
          id='lc_run--019e7eaa-6cac-7901-b58c-d677bf6135be-0',
          tool_calls=[],
          invalid_tool_calls=[],
          usage_metadata={
              'input_tokens': 272,
              'output_tokens': 91,
              'total_tokens': 363,
              'input_token_details': {'audio': 0, 'cache_read': 0},
              'output_token_details': {'audio': 0, 'reasoning': 0}
          }
      )
  ]
}
```

```python
from IPython.display import Image, display

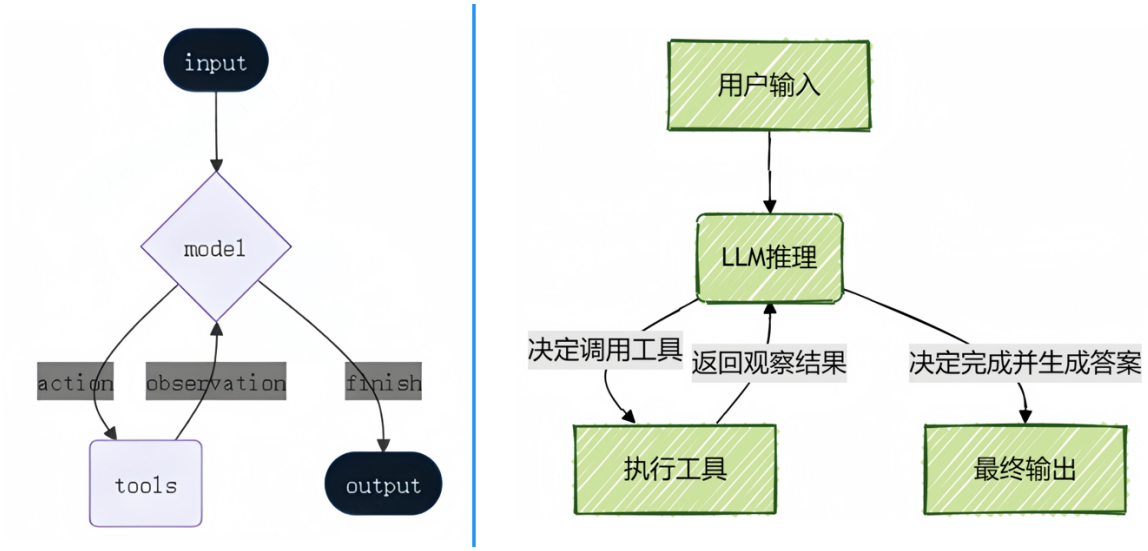
display(Image(agent.get_graph().draw_mermaid_png()))
```



> 注意：只给 Agent 需要的工具，工具太多会混淆。一般2-5 个工具最佳

4.2 工具调用流程分析

LangChain 的 Agent 会将 模型 与 工具 结合起来，在实现上由一个基于 LangGraph 的图结构来编排执行流程，如下所示。



这与前文得到的 Agent 图结构是一致的，本质上就是经典的 **ReAct 结构**：一个具备“ 思考-行动-观察 ”不断循环的自主工作者。

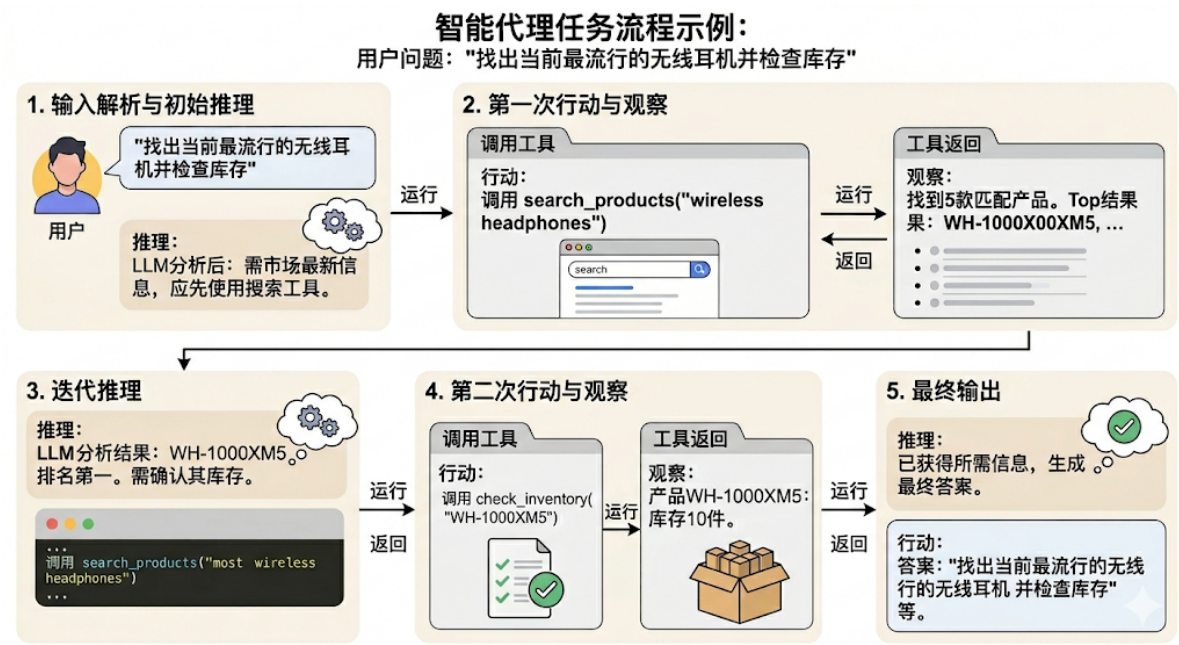
```
用户问题 → AI 思考 → 调用工具 → 观察结果 → 继续思考 → ... → 最终答案
```

当用户提出一个复杂需求时，Agent会像人类一样，先理解任务、规划步骤、使用合适的工具（如搜索网络、查询数据库、执行计算）获取信息，Agent 会在一个循环中 反复调用模型和工具 ，直到某次模型输出中 不再包含工具调用 则结束，最后综合所有信息给出最终答案。

完整流程：



举例：用户问题：“找出当前最流行的无线耳机并检查库存”的任务。



1、输入解析与初始推理

- 输入：用户查询：“找出当前最流行的无线耳机并检查库存”。
- 推理：LLM分析任务后认为：“要找出‘最流行’的产品，需要最新的市场信息，我应该先使用搜索工具。”

2、第一次行动(action)与观察(observation)

- 行动：Agent调用`search_products工具`，参数为“wireless headphones”。
- 观察：工具返回结果：“找到5款匹配产品。Top结果：WH-1000XM5, ...”

3、迭代推理

- 推理：LLM根据搜索结果分析：“WH-1000XM5是排名第一的型号。现在需要确认其库存状态才能回答用户问题。”

4、第二次行动(action)与观察(observation)

- 行动：Agent调用`check_inventory工具`，参数为“WH-1000XM5”。

- 观察：工具返回：“产品WH-1000XM5：库存10件。”

5、最终输出

- 推理：LLM综合所有信息后认为：“已获得所需信息，可以生成最终答案。”
- 行动：模型生成最终答案，不再调用工具。

4.3 重试机制

Agent可以在工具调用结果不满足要求时，自主重试。

```python
from langchain.agents import create_agent
from langchain.tools import tool
from langchain.messages import SystemMessage, HumanMessage
from dotenv import load_dotenv
from rich import print as rprint

load_dotenv(override=True)

flag = 0

@tool
def get_weather(city: str):
    """
    天气查询工具

    Args:
        city: 城市名称
    """
    global flag
    flag += 1

    if flag < 3:
        # raise Exception("暂时无法访问")
        return "TEMP_UNAVAILABLE: 天气服务暂时不可用，请稍后重试"

    return f"{city}今天天气挺好"


messages = [
    SystemMessage("""
    你是一个天气助手。
    当工具返回以 'TEMP_UNAVAILABLE:' 开头的结果时，
    说明是临时故障，不要立即放弃；
    你应再次调用同一个工具，最多重试 3 次。
    如果 3 次后仍失败，再向用户说明服务暂时不可用。
    """),
    HumanMessage("你好，杭州今天的天气如何？")
]
agent = create_agent(model, tools=[get_weather])
response = agent.invoke({"messages": messages})

# rprint(response)

for msg in response["messages"]:
    msg.pretty_print()
```

输出

```
================================ System Message ================================


 你是一个天气助手。
 当工具返回以 'TEMP_UNAVAILABLE:' 开头的结果时，
 说明是临时故障，不要立即放弃；
 你应再次调用同一个工具，最多重试 3 次。
 如果 3 次后仍失败，再向用户说明服务暂时不可用。

================================ Human Message ================================

你好，杭州今天的天气如何？
================================== Ai Message ==================================
Tool Calls:
  get_weather (call_cvICuCPSk4U7aWk6YYkAzI5o)
Call ID: call_cvICuCPSk4U7aWk6YYkAzI5o
Args:
 city: 杭州
================================= Tool Message =================================
Name: get_weather

TEMP_UNAVAILABLE: 天气服务暂时不可用，请稍后重试
================================== Ai Message ==================================
Tool Calls:
get_weather (call_UQL2QTDxKaTCOXbY5Crwbae0)
Call ID: call_UQL2QTDxKaTCOXbY5Crwbae0
  Args:
 city: 杭州
================================= Tool Message =================================
Name: get_weather

TEMP_UNAVAILABLE: 天气服务暂时不可用，请稍后重试
================================== Ai Message ==================================
Tool Calls:
get_weather (call_gZKuGbhp36wEyhjYgeupfu3b)
Call ID: call_gZKuGbhp36wEyhjYgeupfu3b
Args:
   city: 杭州
================================= Tool Message =================================
Name: get_weather

杭州今天天气挺好
================================== Ai Message ==================================

杭州今天天气挺好。
```

模型三次调用 `get_weather`，最终获得了满意的结果。

4.4 常见问题

问题1：Agent 如何选择工具?

依据：工具的 docstring

```python
@tool
def get_weather(city: str) -> str:
    """获取指定城市的天气信息"""  # ← AI Agent读这个!
    ...

@tool
def calculator(operation: str, a: float, b: float) -> str:
    """执行基本的数学计算"""  # ← AI Agent也读这个!
    ...
```

AI 会根据：

1. 问题内容
2. 每个工具的描述
3. 自动选择最匹配的工具

问题2：Agent 为什么没有调用工具?

原因：

- 工具的 docstring 不清晰
- 问题表述不明确
- 模型认为不需要工具

解决：

```python
# ❌ 不好
@tool
def tool1(x: str) -> str:
    """做一些事情"""  # 太模糊


# ☑ 好
@tool
def get_weather(city: str) -> str:
    """
    获取指定城市的实时天气信息

    Args:
        city: 城市名称，如"北京"、"上海"
    """
```

问题3：Agent 选错工具?

原因：

- 多个工具的功能描述相似

- 工具太多导致混淆

解决：

- 只给必要的工具
- 工具描述要有明确区分
- 在 system_prompt 中说明工具使用场景

```python
# ☑ 好：只给需要的工具
agent = create_agent(
    model=model,
    tools=[get_weather, calculator]  # 2-5 个工具最佳
)

# ❌ 不好：工具太多
agent = create_agent(
    model=model,
    tools=[tool1, tool2, ..., tool20]  # 会混淆
)
```

问题4：如何知道 Agent 何时完成？

当 AIMessage 不包含 tool_calls 时：

```python
for msg in response['messages']:
    if isinstance(msg, AIMessage):
        if hasattr(msg, 'tool_calls') and msg.tool_calls:
            print("还在调用工具...")
        else:
            print("完成！最终答案：", msg.content)
```

问题5：Agent 可以调用多少次工具？

默认没有限制，直到得到最终答案。但可能会：

- 超时
- 达到 token 限制
- 模型决定停止

问题6：如何限制工具调用次数？

LangChain 1.0 的 `create_agent` 默认使用 LangGraph，可以通过配置限制：

```python
# 注意：这是高级用法，后续会详细学习
config = {
    "recursion_limit": 5  # 最多 5 步
}

response = agent.invoke(input, config=config)
```

5、Agent的高级用法1：设置Agent名称

创建Agent时，LangChain允许用户指定其名称

5.1 用法

```python
from langchain.agents import create_agent
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os
from rich import print as rprint

# 从.env文件中加载环境变量
load_dotenv(override=True)

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)


agent = create_agent(
    model=model,
    name = "chat_assistant"
)

response = agent.invoke({"messages": ["你好"]})

# rprint(response)

for msg in response["messages"]:
    msg.pretty_print()
```

输出

```
================================ Human Message =================================

你好
================================== Ai Message ==================================
Name: chat_assistant

你好！有什么我可以帮你的吗？
```

输出的AI Message带有**Name**信息。

5.2 经典使用场景

`name` 在 Multi-Agent 场景中最常被提及，用于区分不同的 Agent。但它的作用并不局限于多 Agent 编排。在实际工程中，出现如下场景，通常都建议为 Agent `设置一个清晰且稳定的 name` 。

1. 流式输出归因

在启用流式输出时，name 可用于 `标识当前输出内容来自哪个 Agent` 。

这在多 Agent 协作、Agent 嵌套调用，或前端需要实时展示不同执行主体输出时尤其有用，便于准确区分 token 或事件的来源。

2. 消息身份标记

设置 name 后，Agent 产生的 AIMessage 会携带对应的name信息 。

这使得系统在 保存会话记录 、 回放执行过程 、 构建审计日志 或 前端展示消息角色 时，能够明确识别消息的生成者。

3. 调试与trace可读性

在调试、日志分析和链路追踪过程中，name 可以作为 Agent 的稳定标识，帮助开发者 快速判断当前执行的是哪个 Agent 。

当系统中存在多个能力相近的 Agent，或一个 Agent 被嵌套在更复杂的工作流中时，名称能够显著提升 trace 的可读性 和 问题定位 效率。

4. 组件化封装

在工程实践中，Agent 常被封装为可复用的能力模块，例如检索助手、SQL 助手、报告生成助手等。

为 Agent 设置 name ，有助于在模块注册、运行监控、日志归档和能力复用时保持一致的身份标识。

如果后续需要将该 Agent 进一步作为子图节点、工具能力或子模块接入更复杂系统，也能降低维护和迁移成本。

5. 前端展示与运行态可观测性

在带有可视化界面的应用中，name 还可以直接作为运行时 展示标识 使用。

例如，在执行面板中显示“当前活跃 Agent”“本轮输出来源”或“调用链路中的执行节点”时，name 能帮助开发者和用户更直观地理解系统当前的执行状态。

6. 作为稳定的运行时身份标识

从更通用的角度看，name 可以理解为 Agent 在系统中的“ 运行时身份 ID ”。

相比临时性的展示名称，一个稳定、规范的 name 更适合用于 日志检索 、 监控统计 、 链路分析 和 跨模块协作 ，因此在生产环境中通常建议显式设置，而不是依赖默认行为。

6、Agent的高级用法2：系统提示词

使用 create_agent 创建 Agent 时，需传入 模型 和 工具 、可选地传入 系统提示词 。**提示词为Agent提供了任务背景、行为准则和操作指南。**

系统指令，即SystemMessage，通过 system_prompt 设置，定义 Agent 行为。这个参数可以是 str 或者 SystemMessage类型 。

使用建议：

- 明确说明 Agent 的角色
- 定义输出格式
- 说明何时使用工具

比如：

```
agent = create_agent(
    model=model,
    tools=[get_weather],
    system_prompt="""你是天气助手。

```

```
工作流程：
1. 理解用户的城市查询
2. 使用 get_weather 工具获取数据
3. 简洁清晰地回答

输出格式：
- 天气状况
- 温度
- 注意事项（如有）
"""
)
```

提示词设置有两种方式：静态设置 和 动态设置 。动态设置需要借助**中间件**，后续讲解。

举例1：

```python
from langchain_tavily import TavilySearch
from langchain.agents import create_agent
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os
load_dotenv(override=True)


# 1.导入模型
model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

# 2.导入工具
web_search = TavilySearch(max_results=2)

# 3.创建Agent
agent = create_agent(
    model=model,
    tools=[web_search],
    system_prompt="你是一名多才多艺的智能助手，可以调用工具帮助用户解决问题。"
)

# 4.运行Agent获得结果
result = agent.invoke(
    {"messages": [
        {"role": "user", "content": "请帮我查询2026年足球世界杯是哪个国家举办的？"}
    ]}
)

print(result['messages'][-1].content)
```

```
2026年足球世界杯由**三个国家联合举办**：**加拿大、墨西哥和美国**。
```

举例2：

```python
from langchain.agents import create_agent
```

```python
from langchain_core.messages import SystemMessage
from langchain_core.tools import tool
from rich import print as rprint

# 工具：实现两数相加
@tool
def add_numbers(a: int, b: int) -> str:
    """计算并返回两个数的和。"""
    return f"和为：{a + b}"


# 创建客服助手Agent
agent = create_agent(
    model=model,
    tools=[add_numbers],  # 工具列表
    # system_prompt="你是一个数学助手，解决日常的算术问题"
    system_prompt=SystemMessage(content="你是一个数学助手，解决日常的算术问题")
)

response = agent.invoke(
    {"messages": [
        {"role": "user", "content": "10加上20再加上30是多少？"}
    ]},
)

rprint(response)
# print(response["messages"][-1].content)
```

结果如下：

```
{
 'messages': [
     HumanMessage(
         content='10加上20再加上30是多少？',
         additional_kwargs={},
         response_metadata={},
         id='bd84f000-da12-450c-aeef-0719afcfecfd'
     ),
     AIMessage(
         content='',
         additional_kwargs={'refusal': None},
         response_metadata={
             'token_usage': {
                 'completion_tokens': 22,
                 'prompt_tokens': 158,
                 'total_tokens': 180,
                 'completion_tokens_details': {
                     'accepted_prediction_tokens': 0,
                     'audio_tokens': 0,
                     'reasoning_tokens': 0,
                     'rejected_prediction_tokens': 0
                 },
                 'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                 'latency_checkpoint': {
                     'engine_tbt_ms': 3,
                     'engine_ttft_ms': 31,
```

```
                    'engine_ttlt_ms': 106,
                    'pre_inference_ms': 94,
                    'service_tbt_ms': 4,
                    'service_ttft_ms': 282,
                    'service_ttlt_ms': 352,
                    'total_duration_ms': 267,
                    'user_visible_ttft_ms': 188
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DmKguOQquKeWrXBncvxCak9zkIkEf',
            'service_tier': 'default',
            'finish_reason': 'tool_calls',
            'logprobs': None
        },
        id='lc_run--019e88cd-d248-76b0-836c-71c63856787a-0',
        tool_calls=[
            {
                'name': 'add_numbers',
                'args': {'a': 10, 'b': 20},
                'id': 'call_PWuscHI7NFdVbAV9wsMzOWKy',
                'type': 'tool_call'
            }
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 158,
            'output_tokens': 22,
            'total_tokens': 180,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    ),
    ToolMessage(
        content='和为：30',
        name='add_numbers',
        id='12635dff-6bd8-4325-b64b-bf203d4cacbb',
        tool_call_id='call_PWuscHI7NFdVbAV9wsMzOWKy'
    ),
    AIMessage(
        content='',
        additional_kwargs={'refusal': None},
        response_metadata={
            'token_usage': {
                'completion_tokens': 22,
                'prompt_tokens': 194,
                'total_tokens': 216,
                'completion_tokens_details': {
                    'accepted_prediction_tokens': 0,
                    'audio_tokens': 0,
                    'reasoning_tokens': 0,
                    'rejected_prediction_tokens': 0
                },
                'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                'latency_checkpoint': {
```

```
                    'engine_tbt_ms': 3,
                    'engine_ttft_ms': 31,
                    'engine_ttlt_ms': 105,
                    'pre_inference_ms': 82,
                    'service_tbt_ms': 4,
                    'service_ttft_ms': 257,
                    'service_ttlt_ms': 336,
                    'total_duration_ms': 269,
                    'user_visible_ttft_ms': 176
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DmKgvO5dZRkveTKDtLdPF4BObFmdZ',
            'service_tier': 'default',
            'finish_reason': 'tool_calls',
            'logprobs': None
        },
        id='lc_run--019e88cd-d880-7601-a250-d4ddea91669d-0',
        tool_calls=[
            {
                'name': 'add_numbers',
                'args': {'a': 30, 'b': 30},
                'id': 'call_dZDyMZS1BYRJHdW32uj4z8R8',
                'type': 'tool_call'
            }
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 194,
            'output_tokens': 22,
            'total_tokens': 216,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    ),
    ToolMessage(
        content='和为: 60',
        name='add_numbers',
        id='ffa8b21a-d3fe-4970-bddf-57d3e121f3bf',
        tool_call_id='call_dZDyMZS1BYRJHdW32uj4z8R8'
    ),
    AIMessage(
        content='10加上20再加上30等于 **60**。',
        additional_kwargs={'refusal': None},
        response_metadata={
            'token_usage': {
                'completion_tokens': 18,
                'prompt_tokens': 230,
                'total_tokens': 248,
                'completion_tokens_details': {
                    'accepted_prediction_tokens': 0,
                    'audio_tokens': 0,
                    'reasoning_tokens': 0,
                    'rejected_prediction_tokens': 0
                },
```

```
                'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                'latency_checkpoint': {
                    'engine_tbt_ms': 4,
                    'engine_ttft_ms': 41,
                    'engine_ttlt_ms': 115,
                    'pre_inference_ms': 91,
                    'service_tbt_ms': 4,
                    'service_ttft_ms': 294,
                    'service_ttlt_ms': 360,
                    'total_duration_ms': 278,
                    'user_visible_ttft_ms': 203
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DmKgwNFV9EzDelG0YCsReEfggh4Gg',
            'service_tier': 'default',
            'finish_reason': 'stop',
            'logprobs': None
        },
        id='lc_run--019e88cd-dd3b-7271-80ec-76a8b980a858-0',
        tool_calls=[],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 230,
            'output_tokens': 18,
            'total_tokens': 248,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    )
 ]
}
```

举例3:

```python
from langchain.agents import create_agent
from langchain.tools import tool
from langchain.messages import SystemMessage, HumanMessage

flag = 0

@tool
def get_weather(city: str):
    """
    天气查询工具

    Args:
        city: 城市名称
    """
    global flag
    flag += 1

    if flag < 3:
        # raise Exception("暂时无法访问")
```

```
        return "TEMP_UNAVAILABLE: 天气服务暂时不可用，请稍后重试"

    return f"{city}今天天气挺好"


messages = [
    HumanMessage("你好，杭州今天的天气如何？")
]

agent = create_agent(
    model=model,
    tools=[get_weather],
    system_prompt=SystemMessage(
        "你是一个天气助手。"
        "当工具返回以 'TEMP_UNAVAILABLE:' 开头的结果时，"
        "说明是临时故障，不要立即放弃；"
        "你应再次调用同一个工具，最多重试 3 次。"
        "如果 3 次后仍失败，再向用户说明服务暂时不可用。"
    )
)
response = agent.invoke({"messages": messages})
# print(response)
for msg in response["messages"]:
    msg.pretty_print()
```

输出如下

```
================================ Human Message =================================

你好，杭州今天的天气如何？
================================== Ai Message ==================================
Tool Calls:
get_weather (call_1NZMHHj1xByTOZx7WhiK6AO1)
Call ID: call_1NZMHHj1xByTOZx7WhiK6AO1
Args:
   city: 杭州
================================= Tool Message =================================
Name: get_weather

TEMP_UNAVAILABLE: 天气服务暂时不可用，请稍后重试
================================== Ai Message ==================================
Tool Calls:
get_weather (call_Lgfn3Ll8WTJqQRVVwLIzRQnX)
Call ID: call_Lgfn3Ll8WTJqQRVVwLIzRQnX
Args:
 city: 杭州
================================= Tool Message =================================
Name: get_weather

TEMP_UNAVAILABLE: 天气服务暂时不可用，请稍后重试
================================== Ai Message ==================================
```

```
Tool Calls:
get_weather (call_PghYww2kjkigDZK8h6P5LthO)
Call ID: call_PghYww2kjkigDZK8h6P5LthO
Args:
 city: 杭州
================================= Tool Message
=================================
Name: get_weather

杭州今天天气挺好
================================= Ai Message
=================================

杭州今天天气挺好。
```

7、Agent的高级用法3：结构化输出

结构化输出是Agent的核心功能之一，它允许Agent以特定、可预测的格式返回数据，而不是传统的自然语言响应。通过结构化输出，开发者可以直接获得 Pydantic模型 、 JSON对象 或 数据类 等结构化数据，这些数据能够被应用程序直接使用，无需复杂的解析过程。

7.1 模型 vs Agent的结构化输出对比

第06章已经介绍过结构化输出，当时的重点是与 模型对象 的绑定，这里与Agent的结构化输出做对比：

维度	模型的结构化输出	Agent 结构化输出
操作对象	作用于 大模型对象	作用于 Agent
解析时机	每次模型调用 生成 AIMessage 时，进行解析	仅在 Agent 决定 "任务结束" 并输出最终答案时解析
数据流转	模型 → 结构化对象	模型 → 工具 → 反思 → ... → 结构化对象
绑定方式	使用 with_structured_output	使用 response_format 参数
适用场景	单次、确定性 的任务（如提取字段、翻译、分类）	多步、复杂推理 的任务（如查文档后汇总报表）

7.2 结构化输出的4种策略

LangChain的create_agent()函数自动处理结构化输出的全过程。用户只需通过 “response_format”参数 设置期望的输出模式（Schema）。

当模型生成结构化数据时，系统会自动捕获、验证并将结果存储在Agent状态的 structured_response 键中。

```python
def create_agent(
    ...
    response_format: Union[
        ToolStrategy[StructuredResponseT],
        ProviderStrategy[StructuredResponseT],
        type[StructuredResponseT],
        None,
    ]
```

create_agent函数中的response_format参数支持四种不同的策略设置方式：

① ProviderStrategy

使用模型提供商的 原生结构化输出功能 实现结构化输出。

- 这里所说的“原生结构化输出”指的是大语言模型（LLM）提供商通过其API直接提供的、在模型响应阶段就强制保证 输出格式符合预定规范 的能力，这种能力能够在模型生成内容的源头确保结构化准确性。
- 适用于支持原生结构化输出的模型，比如OpenAI、Anthropic Claude或xAI Grok等。

举例：

```python
from pydantic import BaseModel, Field
from langchain.agents import create_agent
from langchain.agents.structured_output import ProviderStrategy
from langchain.messages import HumanMessage
from rich import print as rprint
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os

load_dotenv(override=True)

# 1.模型初始化
model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)
# 2.Pydantic结构化方式定义
class ContactInfo(BaseModel):
    """用户的联系方式"""
    name: str = Field(description="用户姓名")
    email: str = Field(description="用户邮箱地址")
    phone: str = Field(description="用户的手机号")

# 3.agent初始化
agent = create_agent(
    model=model,
    response_format=ProviderStrategy(ContactInfo)
)

# 4.调用
response = agent.invoke({
        "messages": [
```

```
            HumanMessage("从这段话中抽取结构化信息：小明的邮箱地址为：
shkstart@atguigu.com，手机号：12345678912")
        ]
    }
)

# rprint(response)

for msg in response["messages"]:
    msg.pretty_print()

```

输出

```
================================ Human Message
=================================

从这段话中抽取结构化信息：小明的邮箱地址为：shkstart@atguigu.com，手机号：
12345678912
================================== Ai Message
==================================

{"name":"小明","email":"shkstart@atguigu.com","phone":"12345678912"}
```

② ToolStrategy

对于不支持原生结构化输出的模型，LangChain采用“ToolStrategy”工具调用的方式实现结构化输出。

此策略兼容绝大多数 支持工具调用 的现代模型，其核心原理是动态创建一个" 虚拟工具 "，该工具的输入参数对应着期望的数据结构。

当模型需要生成最终答案时，系统会引导模型 "调用"这个虚拟工具 ，从而间接产生符合要求的结构化数据。

举例：

```python
from pydantic import BaseModel
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os

load_dotenv(override=True)

# 1.模型初始化
model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=os.getenv("CLOSEAI_API_KEY"),
    base_url=os.getenv("CLOSEAI_BASE_URL")
)

# 2.Pydantic结构化方式定义
class ContactInfo(BaseModel):
    name: str = Field(description="姓名")
    email: str = Field(description="邮箱")
```

```python
    phone: str = Field(description="电话")

# 3.工具的定义（根据需要定义）
@tool
def search_tool(query: str) -> str:
    """这是一个搜索引擎。当大模型发现给定的上下文里缺少必要的联系人信息，
    需要去互联网上查询时，才会调用这个工具。
    """
    return f"搜索结果: 未找到关于 '{query}' 的更多额外信息。"

# 3.agent初始化
agent = create_agent(
    model=model,
    tools=[search_tool],
    response_format=ToolStrategy(ContactInfo)
)

result = agent.invoke({
    "messages": [{"role": "user", "content": "联系人信息: John Doe, john@atguigu.com, (010) 56253825"}]
})

# print(result)
print(result["structured_response"])
```

```
name='John Doe' email='john@atguigu.com' phone='(010) 56253825'
```

③ type / AutoStrategy

官方没有在参数列表或官方文档列出这种策略，但阅读源码可以看到。

```
ResponseFormat = ToolStrategy[SchemaT] | ProviderStrategy[SchemaT] | AutoStrategy[SchemaT]
"""Union type for all supported response format strategies."""
```

当我们**直接传入一个定义类型**时，LangChain会**自动包装**为**AutoStrategy**，触发自动选择策略：如果模型支持原生结构化输出（如OpenAI、Anthropic Claude或xAI Grok），则优先使用ProviderStrategy；否则使用ToolStrategy。

举例1：

```python
from langchain.agents.structured_output import AutoStrategy
from pydantic import BaseModel, Field
from langchain.agents import create_agent


class ContactInfo(BaseModel):
    """联系人信息"""
    name: str = Field(description="姓名")
    email: str = Field(description="邮箱")
    phone: str = Field(description="电话")


agent = create_agent(
    model=model,
    response_format=AutoStrategy(ContactInfo)
```

```
)

result = agent.invoke({
    "messages": [{"role": "user", "content": "联系人信息: John Doe, john@atguigu.com, (010) 56253825"}]
})

# print(result)
print(result["structured_response"])
```

输出

```
name='John Doe' email='john@atguigu.com' phone='(010) 56253825'
```

特别注意：在LangChain 1.0及以上版本中，直接传递模式（如response_format=ContactInfo）不再支持，必须显式使用ToolStrategy或ProviderStrategy。（经过测试，目前LangChain 1.2版本还可使用）。

举例2：

```
from pydantic import BaseModel, Field
from langchain.agents import create_agent


class ContactInfo(BaseModel):
    """联系人信息"""
    name: str = Field(description="姓名")
    email: str = Field(description="邮箱")
    phone: str = Field(description="电话")

agent = create_agent(
    model=model,
    response_format=ContactInfo  # Auto-selects ProviderStrategy
)

result = agent.invoke({
    "messages": [{"role": "user", "content": "联系人信息: John Doe, john@atguigu.com, (010) 56253825"}]
})

# print(result)
print(result["structured_response"])
```

```
name='John Doe' email='john@atguigu.com' phone='(010) 56253825'
```

④ None

默认配置，表示不以结构化输出，以`自然语言`响应用户问题。

总结：

在实际大模型Agent开发场景中，如果使用到了结构化输出，推荐使用`“ToolStrategy”策略`，所以后续重点介绍这种策略方式结构化输出。

7.3 ToolStrategy使用详解

ToolStrategy通过 工具调用 （Tool Calling/Function Calling）实现结构化输出，所以LangChain会在消息列表末尾 追加一条ToolMessage ，让整个链路完整。但实际上没有实际的工具执行，这是一条伪消息。

ToolStrategy适用于任何支持工具调用的现代模型。

ToolStrategy的配置包含三个主要参数：

```
class ToolStrategy(Generic[SchemaT]):
    schema: type[SchemaT]
    tool_message_content: str | None
    handle_errors: Union[bool, str, type[Exception], tuple[type[Exception],
...], Callable[[Exception], str]]
```

- **schema（必需参数）**：与提供商策略的schema参数功能一致，支持 Pydantic模型 、 TypedDict 、 JSON Schema 、 数据类(@dataclass) ，同时还支持联合类型 Union[类型1, 类型2] （允许模型根据输入内容选择最匹配的数据结构）。
- **tool_message_content（可选参数）**：用于自定义生成结构化输出时，会话历史中记录的提示信息。默认使用展示输出数据的标准响应语句。
- **handle_errors（可选参数）**：用于指定数据校验失败时的重试策略， 默认值为True 。

7.3.1 结构化输出：schema参数

下面演示四种Schema进行结构化输出代码实现。

提前说明：因为涉及到不同的Schema在不同模型供应商下表现的支持力度不同（上一章有说明，这里提供了两个模型供应商，大家自己选择）

使用CloseAI平台的模型：

```
from langchain.chat_models import init_chat_model
from dotenv import load_dotenv
import os

# 从.env文件中加载环境变量
load_dotenv(override=True)

CLOSEAI_API_KEY = os.getenv("CLOSEAI_API_KEY")
CLOSEAI_BASE_URL = os.getenv("CLOSEAI_BASE_URL")

model = init_chat_model(
    model="gpt-5.4-mini",
    model_provider="openai",
    api_key=CLOSEAI_API_KEY,
    base_url=CLOSEAI_BASE_URL
)
```

使用OpenRouter平台的模型（使用梯子）：

```
from langchain_openrouter import ChatOpenRouter
from dotenv import load_dotenv
import os
```

```
load_dotenv(override=True)

OPENROUTER_API_KEY = os.getenv("OPENROUTER_API_KEY")
OPENROUTER_API_BASE = os.getenv("OPENROUTER_API_BASE")

model = ChatOpenRouter(
    model="openai/gpt-5.4-mini",
    api_key=OPENROUTER_API_KEY,
    base_url=OPENROUTER_API_BASE,
)
```

输出模式1：Pydantic类型

Pydantic类型的Schema支持数据验证，**是优先推荐使用的方式。**

举例1：

```
from pydantic import BaseModel, Field
from langchain.agents.structured_output import ToolStrategy
from langchain.agents import create_agent
from langchain.messages import HumanMessage

class ContactInfo(BaseModel):
    """用户的联系方式"""
    name: str = Field(description="用户姓名")
    email: str = Field(description="用户邮箱地址")
    phone: str = Field(description="用户的手机号")


agent = create_agent(
    model=model,
    response_format=ToolStrategy(ContactInfo)
)

response = agent.invoke({
        "messages": [
            HumanMessage("从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912")
        ]
    }
)

for msg in response["messages"]:
    msg.pretty_print()

# print(response["structured_response"])
```

输出

```
================================ Human Message =================================

从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912
================================== Ai Message ==================================
```

```
Tool Calls:
 ContactInfo (call_OULr5y5MWny1wAgv2JXLDDyO)
Call ID: call_OULr5y5MWny1wAgv2JXLDDyO
 Args:
    name: 小明
    email: songhk@atguigu.com
    phone: 12345678912
================================= Tool Message
=================================
Name: ContactInfo

Returning structured response: name='小明' email='songhk@atguigu.com'
phone='12345678912'
```

观察日志可知，这种方式将结构化信息作为 伪工具传递 ，显然使用了Function Calling方法。

举例2：

```python
from langchain_core.messages import SystemMessage
from pydantic import BaseModel, Field
from typing import Literal
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from langchain.tools import tool
from rich import print as rprint

# 定义工具
@tool(parse_docstring=True)
def search_customer_database(query: str) -> str:
    """
    在客户数据库中搜索信息

    Args:
        query (str): 客户查询字符串，例如 "张三" 或 "李四"

    Returns:
        str: 客户记录字符串，包含客户姓名、等级、最近购买日期和累计消费
    """
    # 模拟数据库查询结果
    if "张三" in query.lower():
        return "客户记录：张三，VIP客户，最近购买日期：2026-01-15，累计消费：
$15,000"
    elif "李四" in query.lower():
        return "客户记录：李四，普通客户，最近购买日期：2025-12-20，累计消费：$3,200"
    else:
        return f"关于客户{query}，无记录"


@tool(parse_docstring=True)
def send_email(customer: str) -> str:
    """
    发送感谢邮件

    Args:
        customer (str): 客户名称，例如 "张三" 或 "李四"

    Returns:
        str: 确认消息，包含已发送的客户名称
```

```
    """
    return f"已向 {customer} 发送感谢邮件"


# 定义Pydantic Schema
class CustomerAnalysis(BaseModel):
    """客户分析报告"""
    customer_name: str = Field(None, description="客户姓名")
    customer_tier: Literal["潜在客户", "普通客户", "VIP客户", "流失风险"] = Field("潜在客户",description="客户等级,只能是潜在客户、普通客户、VIP客户或流失风险")
    recent_activity: str = Field(None, description="最近活动")
    spending_level: Literal["低", "中", "高"] = Field(None, description="消费水平")
    send_email: bool = Field(False, description="是否已发送感谢邮件")


# 创建智能体
agent = create_agent(
    model=model,
    system_prompt=SystemMessage(content=""
                                        "请分析指定客户的情况："
                                        "1. 先搜索客户数据库了解最新情况 "
                                        "2. 如果是VIP客户，则发送感谢邮件 "
                                        "3. 基于搜索结果生成结构化分析报告 "
                                        "4. 如果用户提问与客户记录无关或找不到客户信息，则返回空对象，不发送感谢邮件"
                                ),
    tools=[search_customer_database, send_email],
    response_format=ToolStrategy(CustomerAnalysis)
)

# 执行分析
result = agent.invoke({
    "messages": [{"role": "user", "content": "请分析客户张三"}]
    # "messages": [{"role": "user","content": "请分析客户李四"}]
    # "messages": [{"role": "user","content": "请分析客户王五"}]
    # "messages": [{"role": "user","content": "今天天气如何"}]
})

# 处理结果
# rprint(result)

if "structured_response" in result:
    analysis = result["structured_response"]
    print(analysis)
```

```
customer_name='张三' customer_tier='VIP客户' recent_activity='最近购买日期: 2026-01-15' spending_level='高' send_email=True
```

注意：

1. 如果是结构化输出，在系统提示词中**最后提示结构化输出结果**，如果提示词中先结构化输出结果（Agent已经执行完成），可能会导致一些工具不会再被调用。
2. 系统提示词中最后加入“未找到用户”时的处理提示，避免程序一直调用工具尝试查找对应用户信息。

输出模式2：TypedDict类型

TypedDict 允许为字典对象定义固定的键名和对应的 值类型 ，是 带有类型提示的字典结构 。具体的：

1. TypedDict字段定义采用 Annotated[类型, 默认值, "描述"]格式
2. 可选字段使用 Optional包装，默认值在 Annotated 中指定
3. TypedDict 不支持运行时验证。

举例1：

```python
from typing import TypedDict, Annotated
from langchain.agents.structured_output import ToolStrategy
from langchain.agents import create_agent
from langchain.messages import HumanMessage


class ContactInfo(TypedDict):
    """用户的联系方式"""
    name: Annotated[str, ..., "用户姓名"]
    email: Annotated[str, ..., "用户邮箱地址"]
    phone: Annotated[str, ..., "用户的手机号"]

agent = create_agent(
    model=model,
    response_format=ToolStrategy(ContactInfo)
)

response = agent.invoke({
        "messages": [
            HumanMessage("从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912")
        ]
    }
)

for msg in response["messages"]:
    msg.pretty_print()

# print(response["structured_response"])
```

```
================================ Human Message =================================

从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912
================================== Ai Message ==================================
Tool Calls:
ContactInfo (call_cyteO1flJ8pJnPcsq7Gkb5V7)
Call ID: call_cyteO1flJ8pJnPcsq7Gkb5V7
Args:
 name: 小明
 email: songhk@atguigu.com
 phone: 12345678912
================================= Tool Message =================================
Name: ContactInfo
```

```
Returning structured response: {'name': '小明', 'email':
'songhk@atguigu.com', 'phone': '12345678912'}
```

举例2：

```python
from langchain_core.messages import SystemMessage
from typing import Literal, Optional
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from langchain.tools import tool


# 定义工具
@tool(parse_docstring=True)
def search_customer_database(query: str) -> str:
    """
    在客户数据库中搜索信息

    Args:
        query (str): 客户查询字符串，例如 "张三" 或 "李四"

    Returns:
        str: 客户记录字符串，包含客户姓名、等级、最近购买日期和累计消费
    """
    # 模拟数据库查询结果
    if "张三" in query.lower():
        return "客户记录：张三，VIP客户，最近购买日期：2026-01-15，累计消费：$15,000"
    elif "李四" in query.lower():
        return "客户记录：李四，普通客户，最近购买日期：2025-12-20，累计消费：$3,200"
    else:
        return f"关于客户{query}，无记录"


@tool(parse_docstring=True)
def send_email(customer: str) -> str:
    """
    发送感谢邮件

    Args:
        customer (str): 客户名称，例如 "张三" 或 "李四"

    Returns:
        str: 确认消息，包含已发送的客户名称
    """
    return f"已向 {customer} 发送感谢邮件"


# 使用 TypedDict 定义客户分析报告 Schema
class CustomerAnalysis(TypedDict):
    """客户分析报告"""
    customer_name: Annotated[Optional[str], None, "客户姓名"]
    customer_tier: Annotated[Literal["潜在客户", "普通客户", "VIP客户", "流失风险"], "潜在客户", "客户等级"]
    recent_activity: Annotated[Optional[str], None, "最近活动"]
```

```
    spending_level: Annotated[Optional[Literal["低", "中", "高"]], None, "消费水平"]
    send_email: Annotated[bool, False, "是否已发送感谢邮件"]


# 创建智能体
agent = create_agent(
    model=model,
    system_prompt=SystemMessage(content=""
                                        "请分析指定客户的情况："
                                        "1. 先搜索客户数据库了解最新情况 "
                                        "2. 如果是VIP客户，则发送感谢邮件 "
                                        "3. 基于搜索结果生成结构化分析报告 "
                                        "4. 如果用户提问与客户记录无关或找不到客户信息，则返回空对象，不发送感谢邮件"
                                    ),
    tools=[search_customer_database, send_email],
    response_format=ToolStrategy(CustomerAnalysis)
)

# 执行分析
result = agent.invoke({
    "messages": [{"role": "user", "content": "请分析客户张三"}]
    # "messages": [{"role": "user","content": "请分析客户李四"}]
    # "messages": [{"role": "user","content": "请分析客户王五"}]
    # "messages": [{"role": "user","content": "今天天气如何"}]
})

# 处理结果
# print("result:", result)
if "structured_response" in result:
    analysis = result["structured_response"]
    print(analysis)
```

```
{'customer_name': '张三', 'customer_tier': 'VIP客户', 'recent_activity': '最近购买日期：2026-01-15，累计消费：$15,000', 'spending_level': '高', 'send_email': True}
```

输出模式3：JsonSchema类型

JSON Schema是提供一个标准的 JSON Schema 字典来定义结构。适合需要与多种编程语言交互或进行复杂数据约束定义的场景。

举例1：

```
from langchain.agents import create_agent
from langchain.messages import HumanMessage


json_schema = {
    "title": "ContactInfo",
    "description": "用户的联系方式",
    "type": "object",
    "properties": {
        "name": {
```

```
            "description": "用户姓名",
            "type": "string"
        },
        "email": {
            "description": "用户邮箱地址",
            "type": "string"
        },
        "phone": {
            "description": "用户的手机号",
            "type": "string"
        }
    },
    "required": [
        "name",
        "email",
        "phone"
    ]
}

agent = create_agent(
    model=model,
    response_format=ToolStrategy(json_schema)
)

response = agent.invoke(
    {
        "messages": [
            HumanMessage("从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912")
        ]
    }
)

for msg in response["messages"]:
    msg.pretty_print()

# print(response["structured_response"])
```

输出

```
================================ Human Message =================================

从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912
================================== Ai Message ==================================
Tool Calls:
ContactInfo (call_4Q8JPgCttdnV9AHZEpNgSyuM)
Call ID: call_4Q8JPgCttdnV9AHZEpNgSyuM
Args:
 name: 小明
 email: songhk@atguigu.com
 phone: 12345678912
================================= Tool Message =================================
Name: ContactInfo
```

```
14
15  Returning structured response: {'name': '小明', 'email':
    'songhk@atguigu.com', 'phone': '12345678912'}
```

举例2：

```python
from langchain_core.messages import SystemMessage
from typing import Literal, Optional
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from langchain.tools import tool


# 定义工具
@tool(parse_docstring=True)
def search_customer_database(query: str) -> str:
    """
    在客户数据库中搜索信息

    Args:
        query (str): 客户查询字符串，例如 "张三" 或 "李四"

    Returns:
        str: 客户记录字符串，包含客户姓名、等级、最近购买日期和累计消费
    """
    # 模拟数据库查询结果
    if "张三" in query.lower():
        return "客户记录：张三，VIP客户，最近购买日期：2026-01-15，累计消费：
$15,000"
    elif "李四" in query.lower():
        return "客户记录：李四，普通客户，最近购买日期：2025-12-20，累计消费：
$3,200"
    else:
        return f"关于客户{query}，无记录"


@tool(parse_docstring=True)
def send_email(customer: str) -> str:
    """
    发送感谢邮件

    Args:
        customer (str): 客户名称，例如 "张三" 或 "李四"

    Returns:
        str: 确认消息，包含已发送的客户名称
    """
    return f"已向 {customer} 发送感谢邮件"


# 定义 JSON Schema 替代 Pydantic 模型
customer_analysis_schema = {
    "title": "CustomerAnalysis",
    "type": "object",
    "description": "客户分析报告",
    "properties": {
        "customer_name": {
```

```
            "type": "string",
            "default": "",
            "description": "客户姓名"
        },
        "customer_tier": {
            "type": "string",
            "enum": ["潜在客户", "普通客户", "VIP客户", "流失风险"],
            "default": "潜在客户",
            "description": "客户等级"
        },
        "recent_activity": {
            "type": "string",
            "default": "",
            "description": "最近活动"
        },
        "spending_level": {
            "type": "string",
            "enum": ["低", "中", "高"],
            "default": "低",
            "description": "消费水平"
        },
        "send_email": {
            "type": "boolean",
            "default": False,
            "description": "是否已发送感谢邮件"
        }
    },
    # 所有字段都是必须输出的
    "required": ["customer_name", "customer_tier", "recent_activity",
"spending_level"]
}

# 创建智能体
agent = create_agent(
    model=model,
    system_prompt=SystemMessage(content=""
                                        "请分析指定客户的情况："
                                        "1. 先搜索客户数据库了解最新情况 "
                                        "2. 如果是VIP客户，则发送感谢邮件 "
                                        "3. 基于搜索结果生成结构化分析报告 "
                                        "4. 如果用户提问与客户记录无关或找不到客
户信息，则返回空对象，不发送感谢邮件"
                                ),
    tools=[search_customer_database, send_email],
    response_format=ToolStrategy(customer_analysis_schema)
)

# 执行分析
result = agent.invoke({
    "messages": [{"role": "user", "content": "请分析客户张三"}]
    # "messages": [{"role": "user","content": "请分析客户李四"}]
    # "messages": [{"role": "user","content": "请分析客户王五"}]
    # "messages": [{"role": "user","content": "今天天气如何"}]
})

# 处理结果
# print("result:", result)
if "structured_response" in result:
```

```
    analysis = result["structured_response"]
    print(analysis)
```

```
{'customer_name': '张三', 'customer_tier': 'VIP客户', 'recent_activity':
'最近购买日期：2026-01-15，累计消费：$15,000', 'spending_level': '高',
'send_email': True}
```

注意以上代码中定义json_schema的时候指定的title, description, type, properties, required是遵循JSON Schema 规范的标准关键字，是固定写法。具体细节在第06章2.3节已经介绍过了。

输出模式4：@dataclass类型

@dataclass是Python 3.7引入的一个装饰器，用于简化**数据存储类的定义**。

举例1：

```
from dataclasses import dataclass
from langchain.agents import create_agent
from langchain.messages import HumanMessage

@dataclass
class ContactInfo:
    """用户的联系方式"""
    name: str = Field(description="用户姓名")
    email: str = Field(description="用户邮箱地址")
    phone: str = Field(description="用户的手机号")


agent = create_agent(
    model=model,
    response_format=ToolStrategy(ContactInfo)
)

response = agent.invoke(
    {
        "messages": [
            HumanMessage("从这段话中抽取结构化信息：小明的邮箱地址为：
songhk@atguigu.com，手机号：12345678912")
        ]
    }
)

for msg in response["messages"]:
    msg.pretty_print()

# print(response["structured_response"])
```

```
================================ Human Message
=================================

从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：
12345678912
================================== Ai Message
==================================
Tool Calls:
ContactInfo (call_3MRoBpJHDaoYB6jK7plgW1YF)
```

```
Call ID: call_3MRoBpJHDaoYB6jK7plgW1YF
Args:
 name: 小明
 email: songhk@atguigu.com
 phone: 12345678912
================================= Tool Message
=================================
Name: ContactInfo

Returning structured response: ContactInfo(name='小明',
email='songhk@atguigu.com', phone='12345678912')
```

举例2:

```python
from langchain_core.messages import SystemMessage
from pydantic import BaseModel, Field
from typing import Literal
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from langchain.tools import tool


# 定义工具
@tool(parse_docstring=True)
def search_customer_database(query: str) -> str:
    """
    在客户数据库中搜索信息

    Args:
        query (str): 客户查询字符串，例如 "张三" 或 "李四"

    Returns:
        str: 客户记录字符串，包含客户姓名、等级、最近购买日期和累计消费
    """
    # 模拟数据库查询结果
    if "张三" in query.lower():
        return "客户记录：张三，VIP客户，最近购买日期：2026-01-15，累计消费：$15,000"
    elif "李四" in query.lower():
        return "客户记录：李四，普通客户，最近购买日期：2025-12-20，累计消费：$3,200"
    else:
        return f"关于客户{query}，无记录"


@tool(parse_docstring=True)
def send_email(customer: str) -> str:
    """
    发送感谢邮件

    Args:
        customer (str): 客户名称，例如 "张三" 或 "李四"

    Returns:
        str: 确认消息，包含已发送的客户名称
    """
    return f"已向 {customer} 发送感谢邮件"

```

```python

# 使用Dataclass定义Schema
@dataclass
class CustomerAnalysis:
    """客户分析报告"""
    customer_name: str = Field(None, description="客户姓名")
    customer_tier: Literal["潜在客户", "普通客户", "VIP客户", "流失风险"] = Field("潜在客户",

     description="客户等级,只能是潜在客户、普通客户、VIP客户或流失风险")
    recent_activity: str = Field(None, description="最近活动")
    spending_level: Literal["低", "中", "高"] = Field(None, description="消费水平")
    send_email: bool = Field(False, description="是否已发送感谢邮件")


# 创建智能体
agent = create_agent(
    model=model,
    system_prompt=SystemMessage(content=""
                                        "请分析指定客户的情况："
                                        "1. 先搜索客户数据库了解最新情况 "
                                        "2. 如果是VIP客户，则发送感谢邮件 "
                                        "3. 基于搜索结果生成结构化分析报告 "
                                        "4. 如果用户提问与客户记录无关或找不到客户信息，则返回空对象，不发送感谢邮件"
                                    ),
    tools=[search_customer_database, send_email],
    response_format=ToolStrategy(CustomerAnalysis)
)

# 执行分析
result = agent.invoke({
    "messages": [{"role": "user", "content": "请分析客户张三"}]
    # "messages": [{"role": "user","content": "请分析客户李四"}]
    # "messages": [{"role": "user","content": "请分析客户王五"}]
    # "messages": [{"role": "user","content": "今天天气如何"}]
})

# 处理结果
# print("result:", result)
if "structured_response" in result:
    analysis = result["structured_response"]
    print(analysis)
```

```
CustomerAnalysis(customer_name='张三', customer_tier='VIP客户',
recent_activity='最近购买日期：2026-01-15', spending_level='高',
send_email=True)
```

多schema联合模式

ToolStrategy允许指定多个类型“`Union[类型1, 类型2]`”这种写法，LLM能够根据输入文本的内容，智能地选择`最合适的一个`数据模型（Schema）来生成结构化输出，但是最终会只有一种类型输出。

适用于根据不同输入内容，生成不同的结构化输出的场景，但是底层工具转换结构化输出只会转换成一种结构化类型输出。

```python
from pydantic import BaseModel, Field
from typing import Union
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from langchain.messages import HumanMessage


class ContactInfo(BaseModel):
    """用户的联系方式"""
    name: str = Field(description="用户姓名")
    email: str = Field(description="用户邮箱地址")
    phone: str = Field(description="用户的手机号")

class EventInfo(BaseModel):
    """事件详情"""
    event_name: str = Field(description="事件名称")
    date: str = Field(description="事件发生日期")

agent = create_agent(
    model=model,
    response_format=ToolStrategy(
        Union[ContactInfo, EventInfo]
    )
)

response = agent.invoke(
    {
        "messages": [
            HumanMessage("从这段话中抽取结构化信息：小明的邮箱地址为：shkstart@atguigu.com，手机号：12345678912")
        ]
    }
)

for msg in response["messages"]:
    msg.pretty_print()

print(response["structured_response"])
```

输出

```
================================ Human Message =================================

从这段话中抽取结构化信息：小明的邮箱地址为：shkstart@atguigu.com，手机号：12345678912
================================== Ai Message ==================================
Tool Calls:
ContactInfo (call_teJAkORKANUHugePT4Flnrut)
Call ID: call_teJAkORKANUHugePT4Flnrut
Args:
 name: 小明
 email: shkstart@atguigu.com
 phone: 12345678912
================================= Tool Message =================================
```

```
Name: ContactInfo

Returning structured response: name='小明' email='shkstart@atguigu.com'
phone='12345678912'
name='小明' email='shkstart@atguigu.com' phone='12345678912'
```

继续：

```python
response = agent.invoke(
    {
        "messages": [
            HumanMessage("从这段话中抽取结构化信息：2026年高考报名人数突破1200万")
        ]
    }
)

for msg in response["messages"]:
    msg.pretty_print()

print(response["structured_response"])
```

输出

```
================================ Human Message
=================================

从这段话中抽取结构化信息：2026年高考报名人数突破1200万
================================== Ai Message
==================================
Tool Calls:
EventInfo (call_BrkVCSmkMZqWthLaW3sDaOzr)
Call ID: call_BrkVCSmkMZqWthLaW3sDaOzr
Args:
 event_name: 高考报名人数突破1200万
 date: 2026年
================================= Tool Message
=================================
Name: EventInfo

Returning structured response: event_name='高考报名人数突破1200万'
date='2026年'
event_name='高考报名人数突破1200万' date='2026年'
```

7.3.2 自定义工具消息：tool_message_content参数

> 如果采用ToolStrategy策略处理结构化输出时，LangChain会在消息列表末尾追加一条Tool_message，让整个链路完整。但实际上没有实际的工具执行，这是一条伪消息。

我们可以通过ToolStrategy的 `tool_message_content` 参数定制其消息内容，将指定的内容写入对话历史的提示信息，这样做的好处如下：

1. 在最终用户可见的对话流中，使用 `更自然的消息` 替代原始数据。
2. 用简短的确认信息替代可能很长的数据块， `减少token消耗` 。

举例：

当不设置 tool_message_content时，模型收到的 ToolMessage里就包含了像 `{'name': '张三', 'email': 'zhangsan@email.com'... ...}` 这样的具体数据。当设置了tool_message_content时，模型收到的ToolMessage只是一个预定义的确认信息，如“ `格式化输出成功！` ”。这种方式节省了上下文窗口的令牌消耗，并且让对话流对最终用户更友好。

说明： 无论 tool_message_content如何设置，成功提取的结构化数据最终都会正确存入 `result["structured_response"]` 返回，自定义消息仅影响对话历史中的一条记录。

举例：

默认情况

```python
from pydantic import BaseModel, Field
from langchain.agents.structured_output import ToolStrategy
from langchain.agents import create_agent
from langchain.messages import HumanMessage
from rich import print as rprint

class ContactInfo(BaseModel):
    """用户的联系方式"""
    name: str = Field(description="用户姓名")
    email: str = Field(description="用户邮箱地址")
    phone: str = Field(description="用户的手机号")


agent = create_agent(
    model=model,
    response_format=ToolStrategy(ContactInfo)
)

response = agent.invoke({
        "messages": [
            HumanMessage("从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912")
        ]
    }
)

rprint(response)
```

输出

```
{
 'messages': [
     HumanMessage(
         content='从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912',
         additional_kwargs={},
         response_metadata={},
         id='cee8f1fc-fb73-43b0-827f-3c7db875b21f'
     ),
     AIMessage(
         content='',
         additional_kwargs={},
         response_metadata={
             'model_name': 'openai/gpt-5.4-mini-20260317',
             'id': 'gen-1780468963-yh9c7lZAjqiNoxJJwr5f',
```

```
            'created': 1780468963,
            'object': 'chat.completion',
            'finish_reason': 'tool_calls',
            'logprobs': None,
            'model_provider': 'openrouter'
        },
        id='lc_run--019e8c38-53c3-7801-9c2d-073587f3aea5-0',
        tool_calls=[
            {
                'name': 'ContactInfo',
                'args': {'name': '小明', 'email': 'songhk@atguigu.com',
'phone': '12345678912'},
                'id': 'call_qPF7AApZBYkhS8Lv5DuXWYtr',
                'type': 'tool_call'
            }
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 91,
            'output_tokens': 35,
            'total_tokens': 126,
            'input_token_details': {'cache_read': 0, 'cache_creation':
0},
            'output_token_details': {'reasoning': 0}
        }
    ),
    ToolMessage(
        content="Returning structured response: name='小明'
email='songhk@atguigu.com' phone='12345678912'",
        name='ContactInfo',
        id='3d644146-3292-4817-b3e0-fffced5b55f1',
        tool_call_id='call_qPF7AApZBYkhS8Lv5DuXWYtr'
    )
 ],
 'structured_response': ContactInfo(name='小明',
email='songhk@atguigu.com', phone='12345678912')
}
```

自定义tool_message_content

```python
from pydantic import BaseModel, Field
from langchain.agents.structured_output import ToolStrategy
from langchain.agents import create_agent
from langchain.messages import HumanMessage
from rich import print as rprint

class ContactInfo(BaseModel):
    """用户的联系方式"""
    name: str = Field(description="用户姓名")
    email: str = Field(description="用户邮箱地址")
    phone: str = Field(description="用户的手机号")


agent = create_agent(
    model=model,
    response_format=ToolStrategy(ContactInfo,tool_message_content="已成功抽取
信息")
```

```
)

response = agent.invoke({
        "messages": [
            HumanMessage("从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912")
        ]
    }
)

rprint(response)
```

输出

```
{
 'messages': [
     HumanMessage(
         content='从这段话中抽取结构化信息：小明的邮箱地址为：songhk@atguigu.com，手机号：12345678912',
         additional_kwargs={},
         response_metadata={},
         id='189ac7ea-a632-4391-ab01-dba014fe8e89'
     ),
     AIMessage(
         content='',
         additional_kwargs={},
         response_metadata={
             'model_name': 'openai/gpt-5.4-mini-20260317',
             'id': 'gen-1780469017-pHgxPoxhnVjrXf3OSKgy',
             'created': 1780469017,
             'object': 'chat.completion',
             'finish_reason': 'tool_calls',
             'logprobs': None,
             'model_provider': 'openrouter'
         },
         id='lc_run--019e8c39-26df-7bb3-afb1-19cda137f52a-0',
         tool_calls=[
             {
                 'name': 'ContactInfo',
                 'args': {'name': '小明', 'email': 'songhk@atguigu.com', 'phone': '12345678912'},
                 'id': 'call_3WLGnyP5fjHRcBe5JjJX3eIa',
                 'type': 'tool_call'
             }
         ],
         invalid_tool_calls=[],
         usage_metadata={
             'input_tokens': 91,
             'output_tokens': 35,
             'total_tokens': 126,
             'input_token_details': {'cache_read': 0, 'cache_creation': 0},
             'output_token_details': {'reasoning': 0}
         }
     ),
     ToolMessage(
         content='已成功抽取信息',
```

```
            name='ContactInfo',
            id='0db7198c-edc2-4401-a09f-13aca5a7d3ae',
            tool_call_id='call_3WLGnyP5fjHRcBe5JjJX3eIa'
        )
    ],
    'structured_response': ContactInfo(name='小明',
email='songhk@atguigu.com', phone='12345678912')
}
```

7.3.3 错误处理：handle_errors参数

受限于模型能力，大模型输出的内容可能并不符合格式要求，ToolStrategy通过其handle_errors参数提供了结构化过程错误处理策略，以下是主要的几种方式及其用途：

- **handle_errors=True**：LangChain默认方式，捕获所有异常，并使用LangChain 内置的、信息明确的错误消息模板提示模型重试，确保最终能得到符合预定格式的有效数据。适用于大多数希望自动处理错误的通用场景。
- **handle_errors=False**：关闭自动重试机制，任何异常都会直接抛出，会中断程序运行。
- **handle_errors="自定义字符串"**：捕获所有异常，但使用开发者预设的固定字符串作为错误消息。适用于需要统一、友好的用户提示，或进行特定业务引导的场景。
- **handle_errors=ExceptionType**：仅捕获指定类型(如ValueError)或元组中的异常类型并进行重试，其他异常直接抛出。适用于需要精准控制，只对特定错误进行重试的场景。
- **handle_errors=callable**：灵活性最高的方式，使用开发者自定义的函数来处理异常，可根据不同的异常类型返回差异化的提示信息。适用于需要复杂、精细化错误处理的场景。

情况1：设置为True/False/固定字符串

设计思路：模型对于单条信息的格式化输出请求，输出了多个工具调用请求。也称为多结构化输出错误。

```python
from pydantic import BaseModel, Field
from typing import Union
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from rich import print as rprint

class ContactInfo(BaseModel):
    """个人联系信息"""
    name: str = Field(description="姓名")
    email: str = Field(description="电子邮箱")


class EventDetails(BaseModel):
    """活动详情"""
    event_name: str = Field(description="活动名称")
    date: str = Field(description="活动日期")


agent = create_agent(
    model=model,
    response_format=ToolStrategy(
        Union[ContactInfo, EventDetails],
        tool_message_content="提取完成！",
        handle_errors=True
        #handle_errors="请检查输入数据"
```

```
    )
)

result = agent.invoke({
    "messages": [{
        "role": "user",
        "content": f"请提取以下文本中内容：姓名：张三，电子邮箱：
zhang3@atguigu.com，活动名称：公司年会，活动日期：2026-07-15"
    }]
})

rprint(result)

# for msg in result["messages"]:
#     msg.pretty_print()
#
# report_data = result["structured_response"]
# print(report_data)
```

1）handle_errors设置为True

运行后结果如下：

```
            }
        ),
        ToolMessage(
            content='Error: Model incorrectly returned multiple structured responses (ContactIr
when only one is expected.\n Please fix your mistakes.',
            name='ContactInfo',
            id='97994f2f-d7b7-475e-b22a-beb0c09e74cb',
            tool_call_id='call_0TffUOpyA2dmMUkgNio1Ghf5'
        ),
        ToolMessage(
            content='Error: Model incorrectly returned multiple structured responses (ContactIr
when only one is expected.\n Please fix your mistakes.',
            name='EventDetails',
            id='a1be2e66-400c-461b-b59e-18a16d68812d',
            tool_call_id='call_zOJds6NO4lamrHypRvfwh3Xy'
        ),
        AIMessage(
            content='',
            additional_kwargs={'refusal': None},
            response_metadata={
```

模型接收到错误反馈后，再次生成新的调用请求，直至成功或达到内部的最大重试次数。

ToolStrategy中自定义的tool_message_content控制的是成功后的消息内容，不影响错误消息内容。

2）handle_errors设置为False

运行后结果如下：

```
MultipleStructuredOutputsError: Model incorrectly returned multiple
structured responses (ContactInfo, EventDetails) when only one is
expected.
During task with name 'model' and id 'db9a512a-d2a9-f6e4-0051-
fc5703dcb9c0'
```

3）handle_errors设置为“请检查输入数据”

运行后结果如下：

```
}
{
 'messages': [
     HumanMessage(
```

```
        content='请提取以下文本中内容：姓名：张三，电子邮箱：zhang3@atguigu.com，活动名称：公司年会，活动日期：2026-07-15',
        additional_kwargs={},
        response_metadata={},
        id='fc016142-2bba-4d7d-ae9d-a049d89339bb'
    ),
    AIMessage(
        content='',
        additional_kwargs={'refusal': None},
        response_metadata={
            'token_usage': {
                'completion_tokens': 71,
                'prompt_tokens': 208,
                'total_tokens': 279,
                'completion_tokens_details': {
                    'accepted_prediction_tokens': 0,
                    'audio_tokens': 0,
                    'reasoning_tokens': 0,
                    'rejected_prediction_tokens': 0
                },
                'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0},
                'latency_checkpoint': {
                    'engine_tbt_ms': 3,
                    'engine_ttft_ms': 33,
                    'engine_ttlt_ms': 276,
                    'pre_inference_ms': 88,
                    'service_tbt_ms': 3,
                    'service_ttft_ms': 375,
                    'service_ttlt_ms': 609,
                    'total_duration_ms': 531,
                    'user_visible_ttft_ms': 288
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DmaeBotSr05l6BGYZkf2AfZiXCW4j',
            'service_tier': 'default',
            'finish_reason': 'tool_calls',
            'logprobs': None
        },
        id='lc_run--019e8c75-b72e-7121-8492-c4850c549ba7-0',
        tool_calls=[
            {
                'name': 'ContactInfo',
                'args': {'name': '张三', 'email': 'zhang3@atguigu.com'},
                'id': 'call_NBLcCvMstWj1CK8PiOJ3yH4Z',
                'type': 'tool_call'
            },
            {
                'name': 'EventDetails',
                'args': {'event_name': '公司年会', 'date': '2026-07-15'},
                'id': 'call_74mfheDESm2LsUcO2SpC9rRD',
                'type': 'tool_call'
```

```
            }
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 208,
            'output_tokens': 71,
            'total_tokens': 279,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    ),
    ToolMessage(
        content='请检查输入数据',
        name='ContactInfo',
        id='e4c86dc2-c481-489b-b47a-a3a4a1c6fc17',
        tool_call_id='call_NBLcCvMstWj1CK8PiOJ3yH4Z'
    ),
    ToolMessage(
        content='请检查输入数据',
        name='EventDetails',
        id='d7675b23-eb5e-4bc4-adcc-7a1d82399edf',
        tool_call_id='call_74mfheDESm2LsUcO2SpC9rRD'
    ),
    AIMessage(
        content='',
        additional_kwargs={'refusal': None},
        response_metadata={
            'token_usage': {
                'completion_tokens': 71,
                'prompt_tokens': 303,
                'total_tokens': 374,
                'completion_tokens_details': {
                    'accepted_prediction_tokens': 0,
                    'audio_tokens': 0,
                    'reasoning_tokens': 0,
                    'rejected_prediction_tokens': 0
                },
                'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                'latency_checkpoint': {
                    'engine_tbt_ms': 3,
                    'engine_ttft_ms': 31,
                    'engine_ttlt_ms': 254,
                    'pre_inference_ms': 79,
                    'service_tbt_ms': 4,
                    'service_ttft_ms': 414,
                    'service_ttlt_ms': 623,
                    'total_duration_ms': 636,
                    'user_visible_ttft_ms': 335
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DmaeCOCB5Uxpn7FAtTGpNOPsBgSZm',
            'service_tier': 'default',
            'finish_reason': 'tool_calls',
            'logprobs': None
```

```
        },
        id='lc_run--019e8c75-c0c0-76f1-95cb-a350fc846794-0',
        tool_calls=[
            {
                'name': 'ContactInfo',
                'args': {'name': '张三', 'email':
'zhang3@atguigu.com'},
                'id': 'call_jlVDo6MSOODjW3S6qQ2Rks47',
                'type': 'tool_call'
            },
            {
                'name': 'EventDetails',
                'args': {'event_name': '公司年会', 'date': '2026-07-
15'},
                'id': 'call_OyyMPD4ff2P5BlB7tZP1g3lX',
                'type': 'tool_call'
            }
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 303,
            'output_tokens': 71,
            'total_tokens': 374,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    ),
    ToolMessage(
        content='请检查输入数据',
        name='ContactInfo',
        id='8fc7e28a-c20a-459b-a1cf-4975561d6037',
        tool_call_id='call_jlVDo6MSOODjW3S6qQ2Rks47'
    ),
    ToolMessage(
        content='请检查输入数据',
        name='EventDetails',
        id='620c3d86-ff48-4af4-aa81-07370d56c548',
        tool_call_id='call_OyyMPD4ff2P5BlB7tZP1g3lX'
    ),
    AIMessage(
        content='',
        additional_kwargs={'refusal': None},
        response_metadata={
            'token_usage': {
                'completion_tokens': 71,
                'prompt_tokens': 398,
                'total_tokens': 469,
                'completion_tokens_details': {
                    'accepted_prediction_tokens': 0,
                    'audio_tokens': 0,
                    'reasoning_tokens': 0,
                    'rejected_prediction_tokens': 0
                },
                'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                'latency_checkpoint': {
                    'engine_tbt_ms': 4,
                    'engine_ttft_ms': 46,
```

```
                    'engine_ttlt_ms': 310,
                    'pre_inference_ms': 128,
                    'service_tbt_ms': 3,
                    'service_ttft_ms': 487,
                    'service_ttlt_ms': 687,
                    'total_duration_ms': 571,
                    'user_visible_ttft_ms': 359
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DmaeDvP6HV3Zn4eEv5fRqEXV92Nhy',
            'service_tier': 'default',
            'finish_reason': 'tool_calls',
            'logprobs': None
        },
        id='lc_run--019e8c75-c65d-7d42-881f-3557838a3f03-0',
        tool_calls=[
            {
                'name': 'ContactInfo',
                'args': {'name': '张三', 'email':
'zhang3@atguigu.com'},
                'id': 'call_o935XfYmmwgt5ZimBJVg8fS7',
                'type': 'tool_call'
            },
            {
                'name': 'EventDetails',
                'args': {'event_name': '公司年会', 'date': '2026-07-
15'},
                'id': 'call_loG6Ou2yeX3cSToqvEwjG4DI',
                'type': 'tool_call'
            }
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 398,
            'output_tokens': 71,
            'total_tokens': 469,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    ),
    ToolMessage(
        content='请检查输入数据',
        name='ContactInfo',
        id='6a5dc8d1-1a9f-4934-83a2-6dc037296b69',
        tool_call_id='call_o935XfYmmwgt5ZimBJVg8fS7'
    ),
    ToolMessage(
        content='请检查输入数据',
        name='EventDetails',
        id='fc8b46e9-cfb9-42cb-b92f-97c650f91937',
        tool_call_id='call_loG6Ou2yeX3cSToqvEwjG4DI'
    ),
    AIMessage(
        content='',
        additional_kwargs={'refusal': None},
```

```
        response_metadata={
            'token_usage': {
                'completion_tokens': 29,
                'prompt_tokens': 493,
                'total_tokens': 522,
                'completion_tokens_details': {
                    'accepted_prediction_tokens': 0,
                    'audio_tokens': 0,
                    'reasoning_tokens': 0,
                    'rejected_prediction_tokens': 0
                },
                'prompt_tokens_details': {'audio_tokens': 0,
'cached_tokens': 0},
                'latency_checkpoint': {
                    'engine_tbt_ms': 4,
                    'engine_ttft_ms': 39,
                    'engine_ttlt_ms': 166,
                    'pre_inference_ms': 126,
                    'service_tbt_ms': 5,
                    'service_ttft_ms': 405,
                    'service_ttlt_ms': 525,
                    'total_duration_ms': 414,
                    'user_visible_ttft_ms': 279
                }
            },
            'model_provider': 'openai',
            'model_name': 'gpt-5.4-mini-2026-03-17',
            'system_fingerprint': None,
            'id': 'chatcmpl-DmaeF5dnz4cKcsowkMwqaysAyy4s2',
            'service_tier': 'default',
            'finish_reason': 'tool_calls',
            'logprobs': None
        },
        id='lc_run--019e8c75-cce4-7403-81f6-dbd0cd24ce63-0',
        tool_calls=[
            {
                'name': 'ContactInfo',
                'args': {'name': '张三', 'email':
'zhang3@atguigu.com'},
                'id': 'call_FXgGHA6Q2eYrXIqtyp7rLb37',
                'type': 'tool_call'
            }
        ],
        invalid_tool_calls=[],
        usage_metadata={
            'input_tokens': 493,
            'output_tokens': 29,
            'total_tokens': 522,
            'input_token_details': {'audio': 0, 'cache_read': 0},
            'output_token_details': {'audio': 0, 'reasoning': 0}
        }
    ),
    ToolMessage(
        content='提取完成！',
        name='ContactInfo',
        id='ff4ca4b0-eec0-485a-972b-4fc07fb0f799',
        tool_call_id='call_FXgGHA6Q2eYrXIqtyp7rLb37'
    )
```

```
 ],
 'structured_response': ContactInfo(name='张三',
email='zhang3@atguigu.com')
}
```

以上代码中注意如下几点：

1. ToolStrategy允许指定多个类型“Union[ContactInfo, EventDetails]”这种写法，但是最终只会转换成一种结构化类型输出。
2. 当ToolStrategy通过“Union[ContactInfo, EventDetails]”指定多个类型时，在内部调用生成结构化类型工具会报错。此时：
    - handle_errors=True（默认值）开始发挥作用，系统会生成一个ToolMessage，明确告诉LLM“Error: Model incorrectly returned multiple structured responses (ContactInfo, EventDetails) when only one is expected.”，大模型收到这个精准的反馈后，会重新进行推理，最终选择并输出一个最符合要求的Schema。
    - 如果handle_errors 设置为False，执行代码过程直接报错。
3. 当格式化输出有错误时，Agent内部会进行工具调用重试，直到符合要求格式化输出前，`可能会进行多次重试`。

情况2：设置为指定异常类型

默认情况下，LangChain会处理结构化输出处理时抛出的两类异常：

1. MultipleStructuredOutputsError

多结构化输出错误，当返回的工具调用**请求数量大于1**时，抛出该异常，默认情况下LangChain会拦截异常并提醒模型重试。

2. StructuredOutputValidationError

输出结构化验证错误，当输出格式不符合结构化要求时，抛出上述异常。

默认情况下LangChain会拦截该异常并自动重试。

```python
from pydantic import BaseModel, Field
from typing import Union
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy,
StructuredOutputValidationError,    MultipleStructuredOutputsError
from rich import print as rprint

class ContactInfo(BaseModel):
    """个人联系信息"""
    name: str = Field(description="姓名")
    email: str = Field(description="电子邮箱")


class EventDetails(BaseModel):
    """活动详情"""
    event_name: str = Field(description="活动名称")
    date: str = Field(description="活动日期")


agent = create_agent(
    model=model,
    response_format=ToolStrategy(
        Union[ContactInfo, EventDetails],
        tool_message_content="提取完成！",
```

```
        handle_errors=(MultipleStructuredOutputsError,StructuredOutputValidationError)
        # handle_errors=(StructuredOutputValidationError)
    )
)

result = agent.invoke({
    "messages": [{
        "role": "user",
        "content": f"请提取以下文本中内容：姓名：张三，电子邮箱：zhang3@atguigu.com，活动名称：公司年会，活动日期：2026-07-15"
    }]
})

rprint(result)

```

输出如下：

```
            }
        ),
        ToolMessage(
            content='Error: Model incorrectly returned multiple structured responses (ContactInfo, EventDetails) when only one is expected.\n Please fix your mistakes.',
            name='ContactInfo',
            id='122a95a0-5b6f-408a-8dc9-9055225cf400',
            tool_call_id='call_ai43DUpLAFRDJme0FiURTgZ4'
        ),
        ToolMessage(
            content='Error: Model incorrectly returned multiple structured responses (ContactInfo, EventDetails) when only one is expected.\n Please fix your mistakes.',
            name='EventDetails',
            id='6182ff86-4e98-42a8-a9b9-b04b4e3b7b0c',
            tool_call_id='call_OXL7D04hPz58SalhX0zPBfai'
        ),
        AIMessage(
            content='',
            additional_kwargs={'refusal': None},
            response_metadata={
                'token_usage': {
                    'completion_tokens': 29,
                }
        ),
        ToolMessage(
            content='提取完成！',
            name='ContactInfo',
            id='db6b889d-15e1-4f7b-a8ad-c033d4a25182',
            tool_call_id='call_Yl2C3tcC0zy2XvIHnJdfExya'
        )
    ],
    'structured_response': ContactInfo(name='张三', email='zhang3@atguigu.
}
```

情况3：设置为自定义错误处理函数

指定异常处理函数并返回字符串时，LangChain会在遇到异常时自动重试并将异常处理函数的返回值作为ToolMessage的内容。

我们也可以选择在异常处理函数中抛出异常。

```
from pydantic import BaseModel, Field
from typing import Union
from langchain.agents import create_agent

```

```python
from langchain.agents.structured_output import ToolStrategy, StructuredOutputValidationError,    MultipleStructuredOutputsError
from rich import print as rprint

# 自定义错误处理函数
def custom_error_handler(error: Exception) -> str:
    """自定义错误处理器"""
    error_str = str(error)

    print(f"捕获到错误类型：{type(error).__name__}")
    print(f"错误详情：{error_str}")

    if isinstance(error, StructuredOutputValidationError):
        return "数据格式有误，请检查字段是否符合要求。"
    elif isinstance(error, MultipleStructuredOutputsError):
        return "检测到多个响应，请选择最相关的一个进行返回。"
    else:
        return f"Error: {error_str}"


class ContactInfo(BaseModel):
    """个人联系信息"""
    name: str = Field(description="姓名")
    email: str = Field(description="电子邮箱")


class EventDetails(BaseModel):
    """活动详情"""
    event_name: str = Field(description="活动名称")
    date: str = Field(description="活动日期")


agent = create_agent(
    model=model,
    response_format=ToolStrategy(
        Union[ContactInfo, EventDetails],
        tool_message_content="提取完成！",
        handle_errors=custom_error_handler
    )
)

result = agent.invoke({
    "messages": [{
        "role": "user",
        "content": f"请提取以下文本中内容：姓名：张三，电子邮箱：zhang3@atguigu.com，活动名称：公司年会，活动日期：2026-07-15"
    }]
})

rprint(result)

```

handle_errors指定自定义错误处理，运行后结果如下：

```
捕获到错误类型：MultipleStructuredOutputsError
错误详情：Model incorrectly returned multiple structured responses (ContactInfo, EventDetails) when only one is expected.
```

```
        ),
        ToolMessage(
            content='检测到多个响应，请选择最相关的一个进行返回。',
            name='ContactInfo',
            id='c18debf4-a225-4a23-b9eb-7b464ac19063',
            tool_call_id='call_VExXSt7g5MhkW4v97SqCiYEN'
        ),
        ToolMessage(
            content='检测到多个响应，请选择最相关的一个进行返回。',
            name='EventDetails',
            id='87e29234-f21b-4f06-9a98-bb246cef985c',
            tool_call_id='call_e3OeHRc82GT6ZVMbyGQWdVLW'
        ),
        AIMessage(
            content='',
            additional_kwargs={'refusal': None},
            response_metadata={
                'token_usage': {
```

8、Agent的高级用法4：流式输出及模式

8.1 流式输出的说明

通过 invoke 调用Agent时，内部可能经历多次调用，长时间看不到调用情况，用户体验不好，可以通过 流式调用 （渐进式显示输出）优化用户体验， 实时显示 Agent 运行过程中的更新。特别是在处理 LLM 延迟时尤其有效。

流式输出好处：

- 大型语言模型生成完整响应通常需要几秒钟时间，对于长输出可能达到10-20 秒，用户期望即时反馈， 流式传输让等待过程更加可控 。
- 相比非流式传输需要用户长时间等待完整响应，流式传输可以立即显示文字逐渐出现的效果， 大幅降低用户的等待焦虑 。

设置方式：

通过“ agent.stream(stream_mode=指定模式) ”来指定。具体模式有：values、updates（默认）、messages、custom、checkpoints、tasks、debug。

8.2 具体的输出模式

8.2.1 values输出模式

当 stream_mode 设置为values模式时，每个步骤执行后，都会输出完整的状态信息，适用于每一步都要获取完整状态、状态持久化场景。

举例：

```python
from langchain.agents import create_agent
from langchain.tools import tool
from typing import Dict, Any
from rich import print as rprint

@tool
def query_customer_data(customer_id: str) -> Dict[str, Any]:
    """
    查询客户基本信息

    Args:
        customer_id: 客户ID，用于唯一标识客户

```

```
    Returns:
        包含客户基本信息的字典，如姓名、等级、加入日期等
    """
    # 模拟数据库查询
    return {"name": "张三","level": "VIP","join_date": "2023-01-15"}


@tool
def check_order_history(customer_id: str) -> Dict[str, Any]:
    """
    查询客户订单历史

    Args:
        customer_id: 客户ID，用于唯一标识客户

    Returns:
        包含客户订单历史的字典，如总订单数、总花费等
    """
    return {"total_orders": 15,"total_spent": 25800.00}


@tool
def get_current_promotions() -> Dict[str, Any]:
    """
    获取当前可用促销活动

    Returns:
        包含当前可用促销活动的字典，如活动名称、有效日期等
    """
    return {
        "promotions": ["老用户优惠", "会员专属折扣"],
        "valid_until": "2027-01-31"
    }


# 创建客户服务Agent
customer_service_agent = create_agent(
    model=model,
    tools=[query_customer_data, check_order_history, get_current_promotions]
)


for chunk in customer_service_agent.stream(

        {"messages": [{"role": "user","content": "查询客户ID为 CUST123456 的个人信息、历史订单和可用优惠"}]},
        stream_mode="values"
):
    rprint(chunk)
    print("-" * 50)
```

代码运行后示意结果如下：



8.2.2 updates输出模式

这种模式就是默认模式。该模式中，每个步骤执行后，只增量更新状态中发生变化的内容，用于监控 Agent 执行进度，例如观察Agent决定调用工具、工具执行结果等步骤。

代码如下：

```
# 其他工具代码同上，保持不变
# ... ...
for chunk in customer_service_agent.stream(
        {"messages": [{"role": "user","content": "查询客户ID为 CUST123456 的完整信息和可用优惠"}]},
        stream_mode="updates"
):
    rprint(chunk)
    print("-" * 50)
```

运行结果如下：



8.2.3 messages输出模式

该模式中会输出流式返回的Token以及相关的元数据（如：来自哪个节点），可以用在实现类似 ChatGPT 的打字机效果场景，为聊天机器人等交互式应用提供最佳的实时体验。

代码如下：

```
# 其他工具代码同上，保持不变
# ... ...
for chunk in customer_service_agent.stream(
        {"messages": [{"role": "user","content": "查询客户ID为 CUST123456 的完整信息和可用优惠"}]},
        stream_mode="messages"
):
    print(chunk)
    print("-" * 50)

    # print(chunk[0].content,end="",flush=True)
```

运行结果如下：



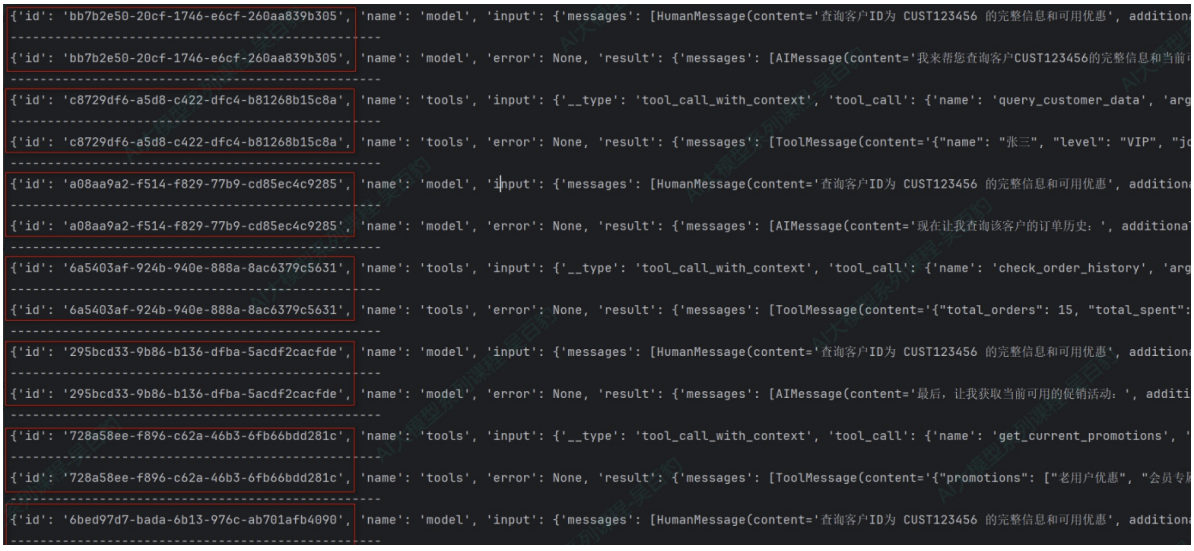
8.2.4 tasks输出模式

该模式会输出当前task任务开始和结束的时间，包含任务的结果和错误信息，该模式用于监控任务的生命周期。

代码如下：

```python
# 其他工具代码同上，保持不变
# ... ...
for chunk in customer_service_agent.stream(
        {"messages": [{"role": "user","content": "查询客户ID为 CUST123456 的完整信息和可用优惠"}]},
        stream_mode="tasks"
):
    print(chunk)
    print("-" * 50)
```

运行结果如下：



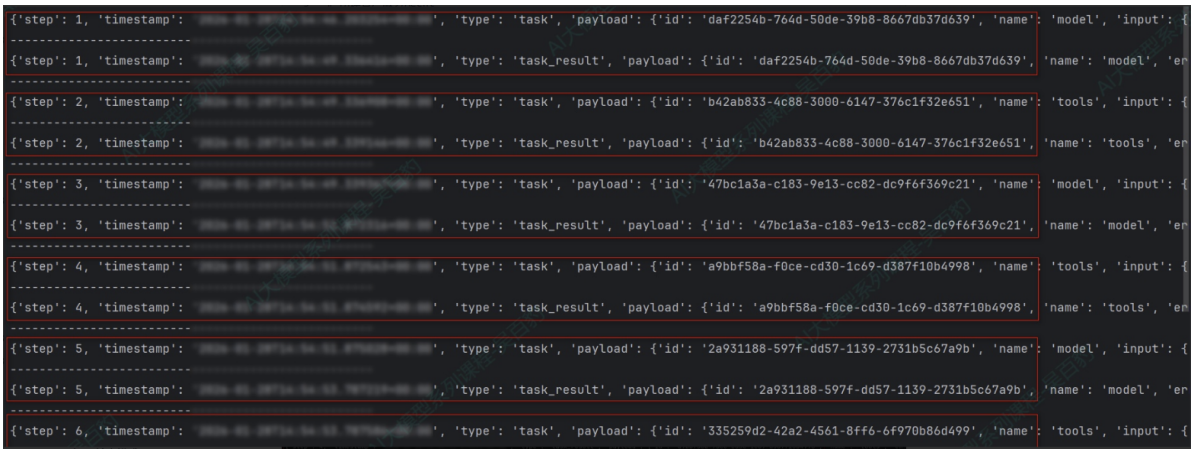
8.2.5 debug输出模式

该模式与tasks模式类似，比task模式多输出任务步骤、时间戳、task类型（task/task_result），该模式用于调试、监控task任务的生命周期。

代码如下：

```
# 其他工具代码同上，保持不变
# ... ...
for chunk in customer_service_agent.stream(
        {"messages": [{"role": "user","content": "查询客户ID为 CUST123456 的完整信息和可用优惠"}]},
        stream_mode="debug"
):
    print(chunk)
    print("-" * 50)
```

运行结果如下：



8.2.6 checkpoints输出模式

该模式中，每当检查点（checkpoint）被创建时会触发输出，输出包含检查点中的状态，用于需要状态持久化、工作流恢复或分布式执行跟踪的高级场景。

代码案例如下：

```
from langgraph.checkpoint.memory import InMemorySaver

# 其他工具代码同上，保持不变
# ... ...
# 1. 创建内存检查点存储
checkpointer = InMemorySaver()

# 2. 创建Agent
customer_service_agent = create_agent(
    model=model,
    tools=[query_customer_data, check_order_history, get_current_promotions],
    checkpointer=checkpointer  # 启用检查点
)

# 3. 创建唯一的会话ID
config = {"configurable": {"thread_id": "session01"}}

# 4. 调用Agent
checkpoint_count = 0

# 使用checkpoints模式进行流式监控
for chunk in customer_service_agent.stream(
        {"messages": [{"role": "user","content": "查询客户ID为 CUST123456 的完整信息和可用优惠"}]},
```

```
        config=config,
        stream_mode="checkpoints"
):
    checkpoint_count += 1
    print(f"检查点 #{checkpoint_count}")
    print(chunk)
    print("-" * 50)
```

运行结果如下，每次输出都会将相关MESSAGE追加到values.messages中。



8.2.7 custom输出模式

开发者通过 get_stream_writer 在工具或节点内部 自定义发送的数据 ，用于 输出 业务逻辑相关的进度信息（如“已处理10/100条记录”）、自定义日志或指标。

举例：生成销售报告和库存报告Agent

```
from langchain.agents import create_agent
from langgraph.config import get_stream_writer
from langchain.tools import tool
import time

@tool
def generate_sales_report() -> str:
    """生成销售报告"""
    writer = get_stream_writer()

    writer({"type": "生成销售报告", "message": "开始生成销售报告"})

    # 模拟数据处理
    for i in range(1, 4):
        time.sleep(0.5)
        writer({"type": "生成销售报告","message": f"生成销售报告进度百分比：{i * 25}%"})

    writer({"type": "生成销售报告", "message": "报告生成完成"})

    return f"销售报告：总收入150万元，同比增长12%"


```

```python
@tool
def generate_inventory_report() -> str:
    """生成库存报告"""
    writer = get_stream_writer()
    writer("开始库存分析...")
    time.sleep(0.5)
    writer("检查当前库存量...")
    time.sleep(0.5)
    writer("生成库存报告...")

    return "当前库存量为10000件，库存充足，无异常"


# 创建报告生成agent
reporting_agent = create_agent(
    model=model,
    tools=[generate_sales_report, generate_inventory_report]
)

for chunk in reporting_agent.stream(
        {"messages": [{"role": "user","content": "生成销售报告和库存报告"}]},
        stream_mode="custom"
):
    print(chunk)
    print("-" * 50)
```

运行结果如下：

```
{'type': '生成销售报告', 'message': '开始生成销售报告'}
--------------------------------------------------
开始库存分析...
--------------------------------------------------
{'type': '生成销售报告', 'message': '生成销售报告进度百分比：25%'}
--------------------------------------------------
检查当前库存量...
--------------------------------------------------
{'type': '生成销售报告', 'message': '生成销售报告进度百分比：50%'}
--------------------------------------------------
生成库存报告...
--------------------------------------------------
{'type': '生成销售报告', 'message': '生成销售报告进度百分比：75%'}
--------------------------------------------------
{'type': '生成销售报告', 'message': '报告生成完成'}
--------------------------------------------------
```

8.3 流式输出模式总结

如下是LangChain Agent输出模式对比：

模式	输出内容	使用场景
values	每个步骤执行后，都会输出完整的状态信息	适用于每一步都要获取完整状态、状态持久化场景
updates（默认）	每个步骤执行后，只增量更新状态中发生变化的内容	用于监控Agent 执行进度，例如观察Agent决定调用工具、工具执行结果等步骤
messages	输出流式返回的Token以及相关的元数据（如：来自哪个节点model/tool）	实现类似ChatGPT 的打字机效果，为聊天机器人等交互式应用提供最佳的实时体验
tasks	输出当前task任务开始和结束的时间，包含任务的结果和错误信息	该模式用于监控任务的生命周期
debug	与tasks模式类似，比task模式多输出任务步骤、时间戳、task类型（task/task_result）	该模式用于调试、监控task任务的生命周期
checkpoints	当检查点（checkpoint）被创建时会触发输出，输出包含检查点中的状态	用于需要状态持久化、工作流恢复或分布式执行跟踪的高级场景
custom	通过get_stream_writer在工具或节点内部自定义发送的数据	用于输出业务逻辑相关的进度信息（如“已处理10/100条记录”）、自定义日志或指标

我们可以根据不同的目标来选择不同的输出模式。例如：

- 实现 实时对话交互 ，优先选择messages模式；
- 观察Agent的 思考与执行步骤 ，优先选择updates模式；
- 需要查看 每一步状态 优先选择values/tasks/debug模式；
- 在工具执行时 输出自定义业务 日志优先选择custom模式。

此外，以上这些模式还可以组合使用，例如，可以同时指定 stream_mode=[“tasks”, “updates”] ，这样在同一个循环里 既能 查看Agent task任务执行内容， 又能 显示Agent每步的更新。

举例：

```
# 其他工具代码同上，保持不变
# ... ...
# 创建客户服务Agent
customer_service_agent = create_agent(
    model=model,
    tools=[query_customer_data, check_order_history, get_current_promotions]
)


for stream_mode, chunk in customer_service_agent.stream(
        {"messages": [{"role": "user","content": "查询客户ID为 CUST123456 的完整信息和可用优惠"}]},
        stream_mode=["tasks", "updates"]
):
    print(f"当前流模式: {stream_mode}, 当前数据: {chunk}")
    print("-" * 50)
```

当指定多模式后，可以通过“ for stream_mode, chunk in customer_service_agent.stream... ”来遍历dict，dict的key(stream_mode)是执行模式，value(chunk)是该模式输出的结果。代码运行输出结果如下：



9、实战：多功能智能助手

项目需求：开发一个多功能智能助手，支持：

1. **天气查询**：查询城市天气
2. **数学计算**：复杂数学运算
3. **时间查询**：获取当前时间、日期计算
4. **货币转换**：多种货币之间转换
5. **信息搜索**：搜索产品、新闻等信息

9.1 模型的初始化

```
1
2  # 1、模型的初始化
3  import os
4  from dotenv import load_dotenv
5  from langchain.chat_models import init_chat_model
6
7  # 从.env文件中加载环境变量
8  load_dotenv(override=True)
9
10 # 模型的初始化
11 model = init_chat_model(
12     model="gpt-5.4-mini",
13     model_provider="openai",
14     api_key=os.getenv("CLOSEAI_API_KEY"),
15     base_url=os.getenv("CLOSEAI_BASE_URL")
16 )
```

9.2 工具的定义

```
1  # ==================== 工具定义 ====================
2  from langchain_core.tools import tool
3  import math
4  from datetime import datetime, timedelta
5
6  @tool
```

```python
def get_weather(city: str) -> str:
    """获取指定城市的实时天气信息

    支持中国主要城市的天气查询

    Args:
        city: 城市名称，如"北京"、"上海"、"深圳"等

    Returns:
        包含温度、天气状况、空气质量的详细信息

    Examples:
        get_weather("北京") 返回 "多云，15-22℃，空气质量良"
    """
    weather_db = {
        "北京": "多云，15-22℃，空气质量良，湿度 45%",
        "上海": "晴天，18-25℃，空气质量优，湿度 60%",
        "深圳": "小雨，22-28℃，空气质量优，湿度 75%",
        "成都": "阴天，16-23℃，空气质量良，湿度 70%",
        "杭州": "晴天，17-24℃，空气质量优，湿度 55%",
        "广州": "多云，21-29℃，空气质量良，湿度 72%"
    }

    result = weather_db.get(city)
    if result:
        return f"{city}：{result}"
    else:
        return f"抱歉，暂不支持查询{city}的天气信息。当前支持：北京、上海、深圳、成都、杭州、广州"

@tool
def calculator(expression: str) -> str:
    """执行数学计算

    支持基本运算符（+、-、*、/、**）和常用数学函数

    Args:
        expression: 数学表达式，可以包含：
            - 基本运算：2 + 3, 10 * 5, 100 / 4
            - 幂运算：2 ** 10
            - 函数：sqrt(16), abs(-5), pow(2, 3)

    Returns:
        计算结果或错误信息

    Examples:
        calculator("2 + 3 * 4") 返回 "14"
        calculator("sqrt(16)") 返回 "4.0"
    """
    try:
        # 安全的数学运算环境
        safe_functions = {
            "sqrt": math.sqrt,
            "pow": pow,
            "abs": abs,
            "round": round,
            "sin": math.sin,
            "cos": math.cos,
```

```
            "tan": math.tan,
            "log": math.log,
            "pi": math.pi,
            "e": math.e
        }

        result = eval(expression, {"__builtins__": {}}, safe_functions)
        return f"{expression} = {result}"
    except Exception as e:
        return f"计算出错：{str(e)}\n提示：请检查表达式格式，支持的函数有 sqrt, abs, pow, sin, cos, tan, log"

@tool
def get_time_info(query_type: str = "current") -> str:
    """获取时间相关信息

    Args:
        query_type: 查询类型
            - "current": 当前时间
            - "date": 今天日期
            - "tomorrow": 明天日期
            - "yesterday": 昨天日期
            - "weekday": 星期几

    Returns:
        时间信息字符串

    Examples:
        get_time_info("current") 返回 "2025年1月25日 14:30:25"
        get_time_info("weekday") 返回 "星期六"
    """
    now = datetime.now()

    if query_type == "current":
        return now.strftime("当前时间：%Y年%m月%d日 %H:%M:%S")
    elif query_type == "date":
        return now.strftime("今天是：%Y年%m月%d日")
    elif query_type == "tomorrow":
        tomorrow = now + timedelta(days=1)
        return tomorrow.strftime("明天是：%Y年%m月%d日")
    elif query_type == "yesterday":
        yesterday = now - timedelta(days=1)
        return yesterday.strftime("昨天是：%Y年%m月%d日")
    elif query_type == "weekday":
        weekdays = ["星期一", "星期二", "星期三", "星期四", "星期五", "星期六", "星期日"]
        return f"今天是{weekdays[now.weekday()]}"
    else:
        return f"不支持的查询类型：{query_type}。支持：current, date, tomorrow, yesterday, weekday"

@tool
def convert_currency(amount: float, from_curr: str, to_curr: str) -> str:
    """货币转换工具
    支持主要货币之间的实时汇率转换

    Args:
        amount: 金额数值
```

```
        from_curr: 源货币代码（CNY/USD/EUR/GBP/JPY/HKD）
        to_curr: 目标货币代码（CNY/USD/EUR/GBP/JPY/HKD）

    Returns:
        转换结果

    Examples:
        convert_currency(100, "CNY", "USD") 返回 "100 CNY = 14.00 USD"
    """
    # 汇率表（相对于 CNY）
    exchange_rates = {
        "CNY": 1.0,      # 人民币
        "USD": 0.14,     # 美元
        "EUR": 0.13,     # 欧元
        "GBP": 0.11,     # 英镑
        "JPY": 20.8,     # 日元
        "HKD": 1.09      # 港币
    }

    # 货币名称
    currency_names = {
        "CNY": "人民币", "USD": "美元", "EUR": "欧元",
        "GBP": "英镑", "JPY": "日元", "HKD": "港币"
    }

    from_curr = from_curr.upper()
    to_curr = to_curr.upper()

    if from_curr not in exchange_rates:
        return f"不支持的源货币：{from_curr}。支持的货币：CNY, USD, EUR, GBP, JPY, HKD"
    if to_curr not in exchange_rates:
        return f"不支持的目标货币：{to_curr}。支持的货币：CNY, USD, EUR, GBP, JPY, HKD"

    # 转换逻辑：先转为 CNY，再转为目标货币
    cny_amount = amount / exchange_rates[from_curr]
    result_amount = cny_amount * exchange_rates[to_curr]

    from_name = currency_names[from_curr]
    to_name = currency_names[to_curr]

    return f"{amount} {from_name}（{from_curr}）= {result_amount:.2f} {to_name}（{to_curr}）"

@tool
def search_info(keyword: str, category: str = "all") -> str:
    """搜索各类信息

    Args:
        keyword: 搜索关键词
        category: 搜索分类
            - "product": 搜索产品
            - "news": 搜索新闻
            - "all": 搜索所有

    Returns:
        搜索结果
```

```
    """
    # 模拟数据库
    products = {
        "手机": "iPhone 15 (¥5999), 小米14 (¥3999), 华为Mate60 (¥6999)",
        "笔记本": "MacBook Pro (¥12999), ThinkPad X1 (¥9999), 华为MateBook (¥7999)",
        "耳机": "AirPods Pro (¥1999), Sony WH-1000XM5 (¥2499)"
    }

    news = {
        "AI": "1. GPT-5 即将发布  2. AI 芯片市场增长 30%  3. 新AI法规出台",
        "科技": "1. 量子计算新突破  2. 6G 技术测试  3. 新能源汽车销量创新高"
    }

    results = []

    if category in ["product", "all"]:
        for key, value in products.items():
            if keyword in key:
                results.append(f"【产品】{key}: {value}")

    if category in ["news", "all"]:
        for key, value in news.items():
            if keyword in key or keyword in value:
                results.append(f"【新闻】{key} 相关: {value}")

    if results:
        return "\n".join(results)
    else:
        return f"未找到关于 '{keyword}' 的{category}信息"

```

9.3 agent的创建

```
from langchain.agents import create_agent


class SmartAssistant:
    """多功能智能助手"""
    def __init__(self):
        # 初始化模型
        self.model = model

        # 工具列表
        self.tools = [
            get_weather,
            calculator,
            get_time_info,
            convert_currency,
            search_info
        ]

        # 系统提示词
        system_prompt = """你是一个多功能智能助手，可以帮助用户：

          🌤 查询天气：使用 get_weather 工具
          🔢 数学计算：使用 calculator 工具
```

```
            ⏰ 时间查询：使用 get_time_info 工具
            💱 货币转换：使用 convert_currency 工具
            🔍 信息搜索：使用 search_info 工具

            重要提示：
            1. 仔细阅读用户问题，确定需要使用哪个工具
            2. 如果需要多个工具，按顺序调用
            3. 总是用友好、专业的语气回答
            4. 如果工具返回了数据，要用通俗易懂的语言解释给用户
            5. 如果无法完成任务，诚实地告诉用户原因

            请始终使用中文回答。"""

        # ☑️ 创建 agent
        self.agent = create_agent(
            model=self.model,
            tools=self.tools,
            system_prompt=system_prompt
        )

        # 对话历史
        self.messages = []

    def chat(self, user_input: str) -> str:
        """对话接口"""
        # 添加用户消息
        self.messages.append({"role": "user", "content": user_input})

        # 调用 agent
        result = self.agent.invoke({"messages": self.messages})

        # 更新消息历史
        self.messages = result["messages"]

        # 返回最后一条 AI 消息
        for msg in reversed(self.messages):
            if msg.type == "ai" and msg.content:
                return msg.content

        return "抱歉，我无法处理这个请求。"

    def reset(self):
        """重置对话历史"""
        self.messages = []

```

9.4 主程序

```
# ==================== 主程序 ====================

def main():
    assistant = SmartAssistant()

    print("=" * 40)
    print("🤖 多功能智能助手（LangChain 1.2）")
    print("=" * 40)
    print("\n我可以帮你：")
```

```
    print("  🌤️  查询天气")
    print("  🔢 数学计算")
    print("  ⏰ 时间查询")
    print("  💱 货币转换")
    print("  🔍 信息搜索")
    print("\n输入 'quit' 退出，输入 'reset' 重置对话\n")

    demos = [
        "北京今天天气怎么样？",
        "帮我算一下 (25 + 17) * 3",
        "现在几点了？",
        "100 美元等于多少人民币？"
    ]

    for demo in demos:
        print(f"👤 {demo}")
        response = assistant.chat(demo)
        print(f"🤖 {response}\n")

    # 重置对话
    assistant.reset()

    # 交互模式
    print("=" * 40)
    print("💬 进入交互模式")
    print("=" * 40)

    while True:
        user_input = input("\n👤 你: ")

        if user_input.lower() == 'quit':
            print("再见！👋")
            break

        if user_input.lower() == 'reset':
            assistant.reset()
            print("☑️ 对话已重置")
            continue

        if not user_input.strip():
            continue

        # 调用助手
        response = assistant.chat(user_input)
        print(f"🤖 助手: {response}")

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

输出：

```
========================================
🤖 多功能智能助手（LangChain 1.2）
========================================

我可以帮你：
🌤️  查询天气
```

```
🔢 数学计算
⏰ 时间查询
💱 货币转换
🔍 信息搜索

输入 'quit' 退出，输入 'reset' 重置对话

👤 北京今天天气怎么样？
🤖 北京今天天气是：**多云，15-22℃**，**空气质量良**，湿度 **45%**。
整体来说比较适合外出，建议穿**轻薄外套**会更舒服。

👤 帮我算一下 (25 + 17) * 3
🤖 计算结果是：**126**。

👤 现在几点了？
🤖 现在是：**2026年06月03日 19:45:40**。

👤 100 美元等于多少人民币？
🤖 **100 美元（USD）约等于 714.29 人民币（CNY）**。

========================================
💬 进入交互模式
========================================
再见！👋
```