

# 多智能体框架调研报告

Multi-agents frame research report

Press Space for next page →



# 为什么需要多智能体框架？

很多人认为 Agent 不就是 LLM 加上工具调用吗？用几个 Python 函数就能实现了。

这么想不能说错，但这只是简单的 agent，一旦 agent 有以下的需要，那你的 Python 函数估计会写到爆炸

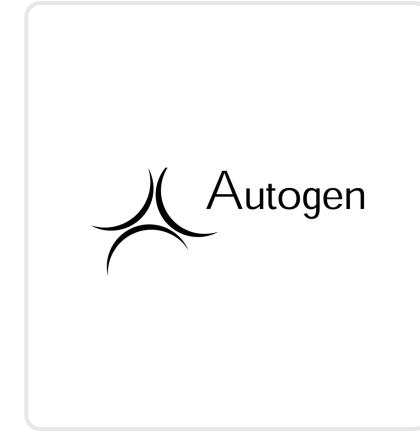
-  多轮对话、上下文管理，即发生用户多次交互的场景
-  多种工具组合调用，如外部的 API、数据库、搜索引擎
-  多个 agent 合作

以上需求都不是简单调个API就能满足的，而是真正的“系统”级问题。你需要考虑

- 如何组织各种模块？比如处理用户交互的对话模块、调用外部工具的工具集成模块、存储信息的记忆模块等。
- 如何保持状态？包括用户多轮对话的上下文（如用户之前提问内容、智能体已给出的回答）、任务执行进度、工具调用记录（如已调用的 API 接口及返回结果）等
- 如何让多个 agent 相互合作，agent 之间又如何传递信息？

# 多智能体框架的价值

- **提升代码复用与开发效率：**这是最直接的价值。可以基于框架提供的标准组件快速搭建 agent 系统，从而避免重复劳动。
- **实现核心组件的解耦与可扩展性：**一个健壮的智能体系统应该由多个松散耦合的模块组成。这种模块化的设计使得整个系统极具可扩展性，更换或升级任何一个组件都变得简单。模型层负责与大语言模型交互，可以轻松替换不同的模型。工具层提供标准化的工具定义、注册和执行接口，添加新工具不会影响其他代码。
- **简化可观测性与调试过程：**当智能体的行为变得复杂时，理解其决策过程变得至关重要。一个精心设计的框架可以内置强大的可观测性能力。



# AutoGen

微软研发，通过多智能体协作构建下一代大语言模型应用的框架

将复杂任务分解成多个子任务，并分配给不同的智能体。每个智能体都有独特的角色和功能，就像一个团队中不同成员各有专长，通过协作完成项目，智能体们通过对话来协调工作，共同完成复杂任务。

# 核心机制: RoundRobinGroupChat

## 轮询群聊

一种明确的、顺序化的对话协调机制。

智能体按照预定义的顺序依次发言，流程固定，易于管理。

开发者只需定义好每个团队成员的角色和发言顺序，剩下的协作流程便可由群聊机制自主驱动。

### 核心思想：

通过“对话”实现协作，将复杂的协作关系，简化为一个流程清晰、易于管理的自动化“圆桌会议”。



# 实例——构建一个模拟的软件开发团队

## 目标：实时显示比特币当前价格

这里是四个核心智能体的定义代码，每个智能体都有其明确的职责和专业领域。

### 产品经理 (ProductManager)

```
def create_product_manager(model_client):
    """创建产品经理智能体"""
    system_message = """你是一位经验丰富的产品经理 ...
你的核心职责包括：xxx
当接到开发任务时，请按以下结构进行分析：
xxx
请简洁明了地回应，并在分析完成后说"请工程师开始实现"。"""

    return AssistantAgent(
        name="ProductManager",
        model_client=model_client,
        system_message=system_message,
    )
```

### 软件工程师 (Engineer)

```
def create_engineer(model_client):
    """创建软件工程师智能体"""
    system_message = """你是一位资深的软件工程师 ...
你的技术专长包括：xxx
当收到开发任务时，请：
xxxx
请提供完整的可运行代码，并在完成后说"请代码审查员检查"。"""

    return AssistantAgent(
        name="Engineer",
        model_client=model_client,
        system_message=system_message,
    )
```

## 代码审查员 (CodeReviewer)

```
def create_code_reviewer(model_client):
    """创建代码审查员智能体"""
    system_message = """你是一位经验丰富的代码审查专家 ...
```

你的审查重点包括：

1. \*\*代码质量\*\*：检查代码的可读性、可维护性和性能
2. \*\*安全性\*\*：识别潜在的安全漏洞和风险点
3. \*\*最佳实践\*\*：确保代码遵循行业标准和最佳实践
4. \*\*错误处理\*\*：验证异常处理的完整性和合理性

审查流程：

1. 仔细阅读和理解代码逻辑
2. 检查代码规范和最佳实践
3. 识别潜在问题和改进点
4. 提供具体的修改建议
5. 评估代码的整体质量

请提供具体的审查意见，完成后说“代码审查完成，请用户代理测试”。”””

```
return AssistantAgent(
    name="CodeReviewer",
    model_client=model_client,
```

## 用户代理 (UserProxy)

```
def create_user_proxy():
    """创建用户代理智能体"""
    return UserProxyAgent(
        name="UserProxy",
        description="""用户代理，负责以下职责：
```

1. 代表用户提出开发需求
2. 执行最终的代码实现
3. 验证功能是否符合预期
4. 提供用户反馈和建议

完成测试后请回复 TERMINATE。”””，  
)

**最终效果**

# 优势与局限性



## 主要优势

### ✖ 贴近人类协作模式：

这种方式更贴近人类团队的协作模式，显著降低了为复杂任务建模的门槛。

### ◎ 聚焦核心问题：

开发者可以将更多精力聚焦于定义“谁（角色）”以及“做什么（职责）”，而非“如何做（流程控制）”。



## 面临挑战

### ✖ 对话的不确定性：

基于 LLM 的对话本质上具有不确定性。智能体可能会产生偏离预期的回复，导致对话走向意外的分支，甚至陷入循环。

### ✖ 对话式调试难题：

调试时没有清晰的错误日志，而是一长串对话历史，定位问题非常棘手。

# 应用场景



## 软件开发与代码生成

多个智能体协作完成编程任务，如代码编写、执行和调试。



## 数据分析与可视化

智能体可自动制定数据处理计划，编写代码，执行并生成可视化结果。



## 市场研究与报告生成

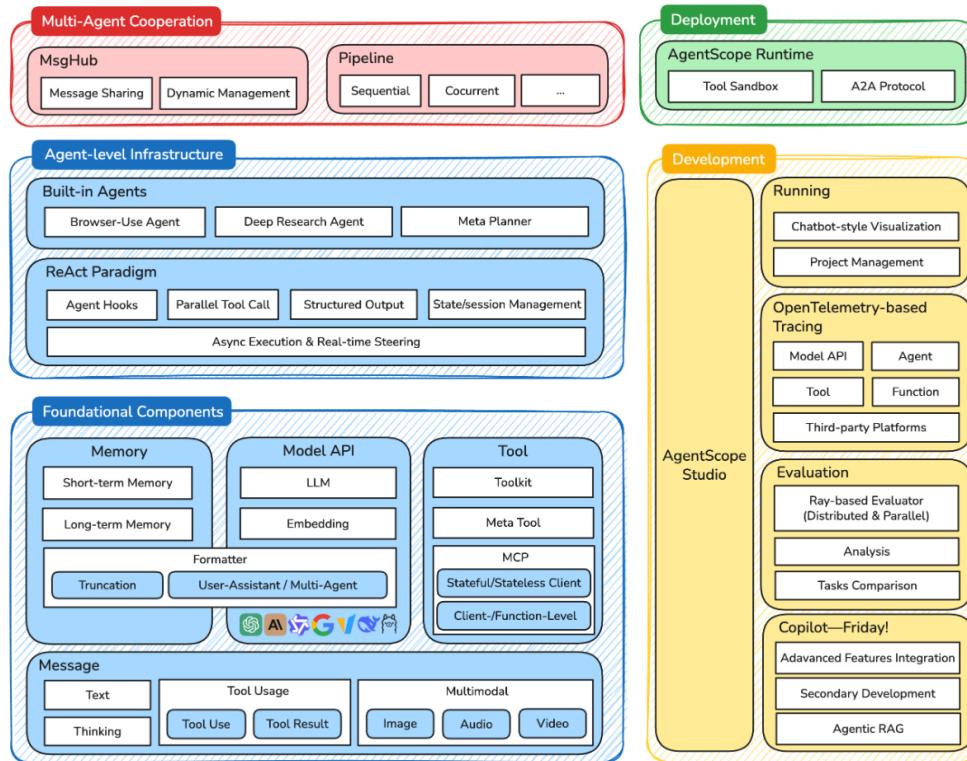
自动化市场分析、报告或宣传材料生成，从网页或数据库提取数据，生成结构化报告。

# Agentscope

阿里研发，工程化优先的框架

专门为构建大规模、高可靠性的多智能体应用而设计

# AgentScope：工程化优先的多智能体平台



分层架构体系：基础组件层 → 智能体基础设施层 → 多智能体协作层 → 开发与部署层

# 消息驱动：智能体交互的"语言"

## 消息结构 (Msg 类)

```
from agentscope.message import Msg

# 标准消息结构
message = Msg(
    name="Alice",          # 发送者名称
    content="Hello, Bob!",  # 消息内容 (支持文本/多模态)
    role="user",            # 角色类型 (user/assistant/system)
    metadata={              # 元数据 (扩展信息)
        "timestamp": "2024-01-15T10:30:00Z",
        "message_type": "text",
        "priority": "normal"
    }
)
```

## 消息驱动优势

- **异步解耦**: 发送方与接收方无需等待，支持高并发
- **位置透明**: 智能体可本地/远程部署，路由自动处理
- **可观测性**: 消息可记录、追踪，便于调试监控
- **可靠性**: 支持持久化与重试，保证最终一致性

消息是 AgentScope 中所有交互的基础单元，替代了传统的函数调用模式

# 生命周期管理与消息传递机制

## 智能体生命周期 (AgentBase)

```
from agentscope.agents import AgentBase

class CustomAgent(AgentBase):
    def __init__(self, name: str, **kwargs):
        super().__init__(name=name, **kwargs)
        # 初始化逻辑: 加载模型、初始化记忆等

    def reply(self, x: Msg) -> Msg:
        # 核心响应逻辑: 处理输入消息并返回结果
        response = self.model.generate(x.content)
        return Msg(name=self.name, content=response)

    def observe(self, x: Msg) -> None:
        # 可选: 观察消息 (如存入记忆, 不返回结果)
        self.memory.add(x)
```

## 消息传递机制 (MsgHub)

- **灵活路由:** 支持点对点、广播、组播等通信模式
- **消息持久化:** 自动保存到 SQLite/MongoDB，支持状态恢复
- **分布式支持:** 跨进程/服务器通信，对开发者透明
- **容错能力:** 内置重试机制，确保消息可靠传递

在 AgentScope 中，每个智能体都有明确的生命周期（初始化、运行、暂停、销毁等），并基于一个统一的基类 AgentBase 来实现。

# 实例：三国狼人杀

以刘备、诸葛亮等三国角色为蓝本，构建兼具**狼人杀核心玩法**与**三国人格特质**的多智能体游戏，验证AgentScope消息驱动架构、角色建模与并发处理能力。



## 多角色协作

功能角色+人格角色



## 消息驱动

替代传统状态机



## 规则约束

结构化输出保障

# 工程化思想：三层解耦设计



## 游戏控制层

- ThreeKingdomsWerewolfGame
- 维护全局状态（存活列表、游戏阶段）
- 推进流程（夜晚/白天阶段切换）
- 裁定游戏胜负



## 智能体交互层

- 核心驱动：MsgHub 消息中心
- 路由分发所有智能体通信
- 支持私密频道（狼人间协商）
- 支持全员广播（白天讨论）



## 角色建模层

- 基础实例：DialogAgent
- 双重身份注入：狼人杀+三国人格
- 通过提示词工程定义行为模式
- 体现角色性格（曹操狡猾、张飞冲动）

# 机制：消息驱动与规则约束

## ↔ 消息驱动游戏流程

### 狼人阶段

MsgHub 创建临时私密频道 → 狼人间协商击杀目标 →  
消息汇总至控制层

### 白天阶段

全员广播频道 → 公开辩论 → 并行收集投票 → 统计结果

### 预言家查验

## ☽ 结构化输出约束规则

- 为游戏行为定义严格输出格式（击杀、查验、投票）
- 确保智能体输出格式一致，便于自动化处理
- 通过字段定义与验证，自动执行游戏规则

# 亮点：角色建模与并发容错



## 双重角色建模

通过提示词工程让智能体扮演好两个层面的角色

既要考虑游戏本身的角色，还要考虑三国人物特性

### 示例：狼人角色差异

曹操（狼人）

狡猾伪装  
挑拨离间

张飞（狼人）

直接强硬  
情绪外露

VS



## 并发处理与容错机制

- 并发处理：**通过 fanout\_pipeline 并行收集多智能体决策（如投票），模拟真实同步场景
- 效率提升：**避免串行等待，大幅缩短多智能体交互耗时
- 容错机制：**关键环节添加超时重试、默认值兜底，单个智能体异常不影响全局流程

```
# 并行收集投票示例
vote_msgs = await fanout_pipeline(
    self.alive_players,
    await
    self.moderator.announce(""),
    structured_model=get_vote_model_cn(self.alive_players),
    enable_gather=False,
)
```

# 优势、劣势与应用场景

## + 核心优势

- **消息驱动架构**: 将复杂流程映射为并发、异步的消息传递，避免传统状态机的僵硬与复杂。
- **良好的并发性能**: 架构设计使其在处理并发任务时展现出天然的性能优势。
- **稳健的容错处理**: 单个智能体异常不会导致整体流程崩溃，保证系统稳健运行。

## - 潜在劣势

- **较高的技术门槛**: 要求开发者理解分布式通信等复杂概念。
- **过度工程化风险**: 对于简单的多智能体对话场景，架构可能显得过于复杂。
- **生态系统尚不成熟**: 作为较新的框架，其社区资源和第三方库还有待完善。

## ⚡ 应用场景

- **大规模生产级系统**: 需要构建高可靠性、高并发的多智能体应用。
- **复杂流程自动化**: 业务逻辑复杂、步骤繁多、需要灵活协作的场景。
- **分布式智能体系统**: 智能体需要在不同节点上运行并进行可靠通信。

# CAMEL

专注于“角色扮演”智能体协作的开发框架

探索在最少的人类干预下，以创新的协作模式简化多智能体系统构建，支持灵活的角色  
定义与任务分配

# 角色扮演(Roly-Playing)

## 核心逻辑

CAMEL 通过为智能体赋予**互补的明确定义角色**，将复杂任务转化为跨领域专家对话，解决单一智能体的能力边界问题。

## 标准协作模式

- **AI 用户 (AI User)**: 提出需求、下达指令、构思步骤
- **AI 助理 (AI Assistant)**: 执行操作、提供方案、落地实现

## 例子：股票交易策略分析工具开发

### AI 用户 → 资深股票交易员

懂市场、懂策略，但不懂编程，负责提出专业需求。

### AI 助理 → 优秀 Python 程序员

精通编程，但不懂股票交易，负责将需求转化为代码。

# 引导性提示：自主协作的“行动纲领”

通过结构化初始指令（System Prompt），确保智能体“不跑偏、高效协作”，实现无人类监督的自主推进。

## 1. 明确自身角色

定义专业背景与能力边界，知道“我是谁、我能做什么”。

## 2. 告知协作者角色

了解伙伴的优势与局限，知道“和我协作的是谁”。

## 3. 定义共同目标

明确协作的最终产出物，知道“我们要去哪里”。

## 4. 行为约束与沟通协议

设定对话规则（如“一次一个步骤”，用什么作为对话完成标志）。

# 优势与局限性

## + 优势

- **轻架构重提示**: 通过精心设计的初始提示实现高质量协作，比硬编码工作流更灵活高效。
- **多模态能力**: 支持文本、图像、音频等多种模态的智能体协作。
- **丰富工具集成**: 内置搜索、计算、代码执行等工具库。
- **多模型适配**: 支持 OpenAI、Anthropic、Google 及开源模型。

## - 主要局限性

- **提示工程依赖高**: 成功高度取决于初始提示质量，设计门槛高，调试复杂。
- **协作规模限制**: 在大规模多智能体场景下面临对话管理和服务同步挑战。
- **缺乏冲突解决**: 多个智能体意见分歧时，无有效的仲裁机制。
- **任务适用性边界**: 不适合严格流程控制、大规模并发和复杂决策树场景。

# 场景选择

## ✓ 理想应用场景

- **深度协作任务**: 需要跨领域专家紧密配合的复杂任务。
- **创造性思维任务**: 如产品构思、创意写作、方案设计等。
- **快速原型验证**: 轻量级架构，适合快速验证多智能体协作思路。
- **多模态交互**: 需要处理文本、图像、音频等多种模态的场景。



## ⚠ 非最优选择场景

- **严格流程控制**: 需要精确步骤控制的任务 (LangGraph 更合适)。
- **大规模并发**: 高并发场景 (AgentScope 更有优势)。
- **复杂决策树**: 多方决策场景 (AutoGen 群聊模式更灵活)。
- **长周期任务**: 需要稳定状态管理和容错机制的长期运行任务。

# LangGraph

构建状态化多智能体应用的图结构框架

通过图结构定义智能体的状态流转与协作关系，支持复杂的决策逻辑和动态任务编排，  
让多智能体系统像精密仪器一样运转。

# LangGraph 核心特性

LangGraph 将智能体执行流程建模为状态机 (State Machine) , 并以有向图 (Directed Graph) 形式呈现。

这种范式的革命性之处在于：**天然支持循环**，让构建具备迭代、反思和自我修正能力的复杂智能体工作流，变得直观且简单。



# 三大基本构成要素 (上)

LangGraph 的运作依赖三个基础要素，共同支撑起完整的工作流体系：



## 全局状态 (State)

- 围绕共享状态对象展开，通常定义为 Python 的 TypedDict
- 可追踪对话历史、中间结果、迭代次数等任意信息
- 所有节点均可读取和更新这个中心状态

```
class State(TypedDict):
    conversation: list
    intermediate_results: dict
    iteration_count: int
```



## 节点 (Nodes)

- 接收当前状态作为输入，返回更新后的状态作为输出
- 本质是独立的 Python 函数，是执行具体工作的单元
- 功能灵活：可调用 LLM、执行工具、处理数据等

# 三大基本构成要素 (下)

## ↔ 边 (Edges)

### 常规边 (Regular Edges)

指定节点输出固定流向另一个节点，适用于线性逻辑

示例：节点A → 节点B

### 条件边 (Conditional Edges)

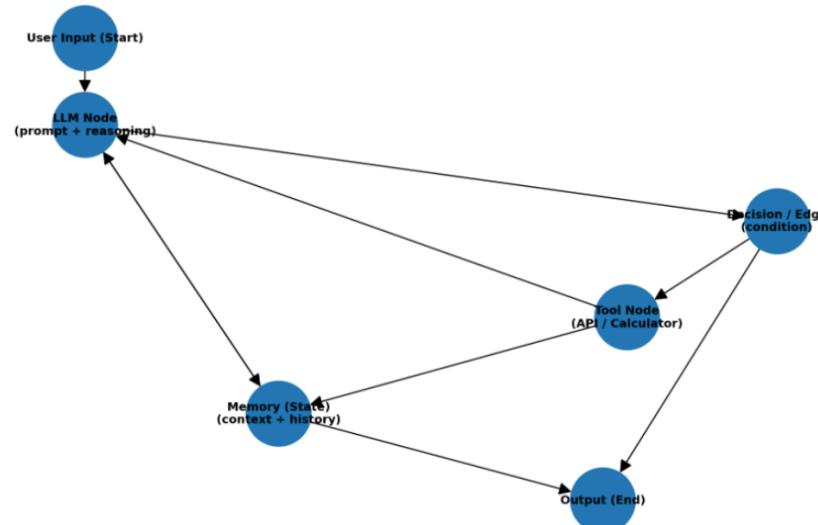
通过函数判断当前状态，动态决定跳转目标，是实现循环和分支的核心

示例：根据结果是否满意 → 节点C 或 节点A (重试)

```
def conditional_edge(state: State) → str:  
    if state["intermediate_results"]["satisfied"]:  
        return "end_node" # 满足条件，跳转到结束节点  
    else:  
        return "process_node" # 不满足，返回重试节点
```

# LangGraph 图结构

LangGraph: Nodes, Edges, and State (flow visualization)



# LangGraph 优劣分析

+

## 核心优势

- 高度可控与可预测：**将流程显式定义为图结构，每一步行为都清晰可见，便于调试和审计，是构建生产级应用的理想选择。
- 原生支持循环与迭代：**通过条件边轻松实现“反思-修正”或“重试”循环，为构建自我优化和容错的智能体提供了强大工具。
- 模块化与可扩展性：**每个节点都是独立的 Python 函数，易于复用、测试和替换，同时便于横向扩展复杂流程。

-

## 局限性

- 前期代码量较大：**需要定义状态、节点、边等，对于简单任务而言，开发流程相对繁琐。
- 需要关注流程控制细节：**开发者需要更多地思考“如何做”（how），而不仅仅是“做什么”（what），增加了认知负荷。
- 灵活性与动态性不足：**工作流预先定义，行为可控但缺少对话式智能体那种动态的、“涌现”式的交互能力。

# 多智能体框架对比总结

框架	核心概念	主要优势	典型应用场景
AutoGen	对话的智能体 (Conversable Agent)	自动化多智能体对话流，高度可定制的交互模式。	自动化代码生成与测试、模拟产品开发流程。
AgentScope	消息传递 (Message Passing)	易用性强，工程化程度高，支持分布式部署。	构建复杂、可运维的大规模多智能体应用。
CAMEL	角色扮演 (Role-Playing)	极简的协作设定，通过初始提示驱动自主对话。	探索性任务、创意生成、模拟特定领域专家协作。
LangGraph	状态图 (State Graph)	天然支持循环和条件分支，精准控制复杂工作流。	实现 Reflection、迭代式优化、需要人工介入的流程。

# More

## AI Agents Stack

NOVEMBER 2024

### VERTICAL AGENTS



### AGENT HOSTING & SERVING

### OBSERVABILITY

提示：图片可上下滑动查看完整内容

# 参考链接

- <https://zhuanlan.zhihu.com/p/1903527757977191290>
- <https://msdocs.cn/autogen/stable/index.html>
- <https://zhuanlan.zhihu.com/p/664937747>
- <https://blog.csdn.net/HJS123456780/article/details/150278747>
- [https://blog.csdn.net/l01011\\_/article/details/146248813](https://blog.csdn.net/l01011_/article/details/146248813)
- [https://blog.csdn.net/qq\\_35082030/article/details/149172011](https://blog.csdn.net/qq_35082030/article/details/149172011)
- <https://datawhalechina.github.io/hello-agents/#./chapter6/>
- <https://zhuanlan.zhihu.com/p/1926741887202592142>
- <https://langgraph.com.cn/index.html>