多智能体系统：AI时代的软件模块化

Original 熊节 极限编程合作社 2025年11月5日 11:03 上海

> 从长提示词的困境到智能体的模块化设计

1. 长提示词的困境

当我们开始深度使用大语言模型（LLM）来处理复杂任务时，很快就会遇到一个看似矛盾的现象：提示词工程的技巧越来越多，提示词本身也越来越长，但效果却并不总是随之提升。

1.1 提示词工程的演进

提示词工程作为一门新兴的实践，在过去几年快速发展。我们学会了使用 Problem Orientation （问题导向）先说明要解决什么问题再给具体指令，使用 Epistemology Frame （认识论框架）明确指定分析视角和理论框架，使用 Exhaustive Input （详尽输入）提供充分的背景信息和上下文，使用 Prescribed Process （规定流程）分步骤说明如何完成任务，使用 Structured Output （结构化输出）精确定义输出格式，使用 Style Mimicry （风格模仿）给出范例让LLM学习特定风格。这些技巧确实有效，但它们同时也意味着提示词越来越长。

从最初的几句话，到几百字的详细指令,再到几千字的复杂提示词。当我们处理真正复杂的知识工作时——比如撰写研究报告、进行多维度分析——提示词很容易达到数万字的规模。更进一步，我们甚至发展出了 Attachment （附件）模式：上传大量文档作为提示词的一部分。这些文档可能包括方法论说明（如辩证分析方法，上万字）、理论框架（如马克思主义认识论，上万字）、风格指南（如学术写作规范，数千字）、以及各种参考资料。本质上，这些"附件"都是一个巨长提示词的组成部分。有些时候，完整的提示词可以达到3-5万words的规模。

1.2 长提示词的三大问题

作为程序员，我很快意识到长提示词带来的问题是结构性的。

第一个问题是注意力分散。即使没有超出上下文窗口的限制，当提示词包含大量且详细的指示时，LLM也很难完美执行所有要求。就像人类阅读一份冗长的文档一样，LLM会出现注意力失焦（attention dilution）而忽略某些细节，会自主简化指令按自己的理解来执行，会在多个目标之间失去优先级判断。结果就是输出质量不稳定，很难达到长提示词想要的那么完美的效果。

第二个问题是缺乏模块隔离。你可能会觉得自己在提示词里做了"模块划分"——比如"第一步做什么，第二步做什么"。但实际上，所有内容都被一次性load到LLM里。这意味着：

- • "牵一发动全身"：修改矛盾分析的方法可能意外影响到阶级分析的执行；
- • 没有真正的边界，所有模块都在同一个上下文中互相影响；
- • 无法实现隔离，不同部分之间的耦合是不可控的。

这就像把所有代码写在一个巨大的函数里，虽然你用注释分了段落，但本质上还是一个整体。

第三个问题是维护困难。长提示词几乎不可能实现灵活的参数传递和流程控制。

- • 参数传递不可能：你只能在最外层设置一些"全局常量"（如"分析XXX国家"），几乎不可能从一个模块中产生一些变量然后用这些变量来调用别的模块。
- • 动态流程控制困难：想让"信息收集"循环三轮，每轮收集后评估效果、查漏补缺？在长提示词中几乎无法实现——你无法让LLM"暂停"、检查中间结果、再"继续"。
- • 修改成本高：如果想调整风格指南，需要重新load整个3万字的提示词；如果想增加一个新的分析维度，整个提示词会变得更长，复杂度指数上升。

1.3 问题的本质

仔细思考就会发现，长提示词本质上就是静态的代码include。你把多个部分（角色定义、流程说明、方法论文档、风格指南）拼接在一起，最终执行的是一整个完整的"源文件"。这和早期编程语言中的 #include 或文本拼接没有本质区别。这是最原始的软件模块化和复用方式——它不可避免地带来前面提到的所有问题。

2. 软件工程的启示

作为软件从业者，我们对这个问题并不陌生。软件工程在过去几十年里，解决的正是类似的模块化问题。

2.1 模块化的演进历程

回顾软件模块化的演进，可以看到清晰的发展路径：

- • 阶段0是所有代码在一个文件，这是最原始的方式，所有逻辑写在一起，问题是难以理解、难以维护、无法复用。
- • 阶段1是源代码include/import（编译时合并），可以把代码分散到多个文件，编译或解释时合并成一个整体，但问题是所有内容最终还是在一个命名空间里，边界不清、耦合过紧。
- • 阶段2是静态链接库（独立编译，链接时合并），模块可以独立编译再链接时组装成可执行文件，进步是有了编译单元的概念，但运行时还是一个整体。
- • 阶段3是动态链接库（运行时加载，独立生命周期），模块在运行时动态加载，可以按需加载、用完释放，关键突破是模块有了独立的生命周期。
- • 阶段4是RPC/Web Services（独立进程，网络通信），模块可以运行在不同进程、不同机器，通过网络调用彼此的服务，进一步实现了物理隔离解耦。

- • 阶段5是微服务（独立部署，松耦合），每个服务独立开发、部署、扩展，通过定义良好的接口通信，实现了最大化的模块自治。

每一次演进都是为了解决上一阶段的问题：边界不清、耦合过紧、难以复用、难以维护。

2.2 模块的本质特征

从这个演进历程中，我们可以总结出一个真正的模块必须具备四个特征。

- • **边界（Boundary）** 意味着明确的职责范围，不与其他模块的职责混淆，对外暴露接口而隐藏内部实现。
- • **生命周期（Lifecycle）** 意味着独立的创建、执行、销毁过程，可以按需启动和终止，资源可以回收和复用。
- • **接口（Interface）** 意味着清晰的输入输出定义，调用者不需要了解内部实现，通过契约而非实现来协作。
- • **隔离（Isolation）** 意味着内部实现对外不可见，修改内部实现不影响其他模块，资源（内存、上下文）不会泄漏或冲突。

2.3 对照表：三种范式的对比

让我们对比一下传统编程的不同阶段、当前的提示词工程、以及我们将要讨论的多智能体系统：

维度	传统编程（原始）	提示词工程（当前）	多智能体系统（未来）
代码组织	所有代码在一个.c文件	长提示词拼接多段指令	独立的智能体模块
命名空间	全局变量	全局"常量"（外层参数）	模块私有+参数传递
生命周期	程序启动到结束	单次LLM调用	按需创建/销毁智能体实例
上下文/内存	共享全局内存空间	共享一个上下文窗口	每个智能体独立上下文窗口
数据传递	全局变量+函数参数	拼接到提示词中	参数+文件管道
并发执行	顺序执行	不支持	支持并行调度
复用方式	复制粘贴代码	复制粘贴文本片段	调用智能体模块
影响范围	修改一处可能影响全局	修改一段影响整个提示词	模块内部修改不影响外部

这个对比清楚地表明：提示词工程目前停留在"阶段1"（源代码include）。而软件工程的经验告诉我们，我们需要进化到"阶段3+"——真正具有独立生命周期和清晰边界的模块。

3. 多智能体系统（MAS）

基于软件工程的启示，我们需要为AI应用引入真正的模块化概念。这就是多智能体系统（Multi-Agent System, MAS）。

3.1 什么是MAS

多智能体系统（MAS）是用大语言模型（LLM）实现的模块化系统。在MAS中，每个模块称为一个"智能体"（Agent），多个智能体协作完成复杂任务，每个智能体具备真正的模块特征：明确的边界、各自独立的生命周期、清晰的输入输出接口、彼此隔离的运行时上下文。

与传统软件模块相比，智能体的特殊之处在于它使用自然语言定义（而非编程语言），可以理解复杂的意图和上下文，可以自主决策执行策略（而非机械执行指令），可以使用工具完成任务（如搜索、读写文件）。

3.2 核心概念体系

为了清晰地讨论MAS，我们需要建立一套精确的概念体系。

模块（Module） 是软件工程中的通用概念，有明确边界、生命周期、输入输出的独立单元，可以用多种技术实现（函数、类、库、服务、容器等），是一个抽象概念，不特定于某种技术。

智能体（Agent） 是用LLM实现的模块。除了传统模块的特征，它还具备理解自然语言指令（不需要精确的编程语法）、自主决策执行策略（根据情况灵活调整方法）、使用工具完成任务（如文件操作、网络搜索、数据分析）的能力。需要明确的是，单次LLM调用不是智能体（虽然使用了LLM，但没有明确的边界和生命周期管理），长提示词的一个段落不是智能体（虽然在逻辑上是"一部分"，但没有真正的隔离），传统API或函数不是智能体（虽然是模块，但不使用LLM，无法理解自然语言意图）。

智能体蓝图（Agent Blueprint） 是智能体的"源代码"，通常是一个Markdown文档，包含角色定义（你是谁负责什么）、输入规格（接收什么参数和数据）、工作流程（如何完成任务，可以是声明式的而非命令式的详细步骤）、输出规格（产出什么结果写到哪里）、质量标准（如何自检什么是好的结果）。类比面向对象编程，蓝图就是类定义（class definition）。

智能体实例（Agent Instance） 是蓝图在运行时的具象化，拥有独立的上下文窗口（不与其他实例共享），执行特定的任务（带着具体的参数），执行完毕后释放资源（上下文窗口被回收）。类比面向对象编程，实例就是对象（object instance）。你可以从同一个蓝图创建多个实例，比如同时创建3个"信息收集智能体"实例分别收集经济、政治、社会信息，或循环创建"分析智能体"实例对多个国家进行分析。

参数（Parameters） 是调用智能体时传入的数据，可以很短（如国家代码"ARG"、日期"2024-10-15"），也可以比较长（如一段问题描述、一组指导原则），但不建议特别长（长数据应该通过文件传递）。需要注意的是，在现有的运行时环境（如Claude Code）中，通常只支持入参（input parameters），不支持显式的出参或返回值。一个智能体执行

完就是执行完，调用者无法直接获得"返回值"。那么智能体之间如何传递数据呢？答案是文件。

文件管道（File-based Pipeline） 是智能体间的主要通信方式：一个智能体的输出文件就是另一个智能体的输入文件，通过约定的文件路径来协调数据流。这种方式有三大优势：完全透明（人类可以直接读取、审查任何中间产物），简单直接（不需要复杂的消息队列或RPC配置），持久化（可追溯、可恢复，出问题可以从中间阶段重新开始）。

参考库（Reference Library） 是领域知识的静态存储。它相对静态，不随每次任务变化，是相对稳定的知识（方法论、理论框架、标准等）。它跨智能体共享，多个智能体可以读取同一个参考库，避免了知识的重复定义。它可配置，修改参考库可以改变系统行为而无需修改智能体蓝图本身。它的内容类型包括方法论文档（如 dialectical-analysis-method.md ）、风格指南（如 academic-writing-style.md ）、理论框架（如 marxist-epistemology.md ）、领域本体（如 class-analysis-framework.md ）。需要区分的是，输入文件是任务特定的数据（如"某国家的原始材料"），而参考库是通用的知识；智能体蓝图描述"做什么、怎么做"，而参考库提供"用什么知识来做"。

3.3 MAS的关键优势

理解了这些概念后，我们可以看到MAS相比长提示词的根本优势。

第一是突破上下文限制。每个智能体使用独立的上下文窗口：智能体A可能用了15万tokens执行完后释放，智能体B重新开始又有完整的上下文预算，理论上可以处理无限大的任务（只要模块划分合理）。这就像动态链接库可以按需加载和卸载，而不是把所有库都常驻内存。

第二是真正的模块隔离。修改一个智能体不影响其他智能体：改进矛盾分析方法只需更新 dialectical-analysis-method.md 参考库，调整报告格式只需修改报告生成智能体的蓝图，增加新分析维度只需增加一个新智能体而现有智能体无需修改。每个智能体有清晰的边界和职责，易于测试和优化。

第三是支持复杂协作。MAS支持多种协作模式：顺序执行（流水线模式，一个阶段的输出是下一个阶段的输入），并行执行（多个智能体同时工作如同时收集经济、政治、社会信息），循环迭代（orchestrator可以调用同一个智能体多轮每轮检查质量），动态决策（编排者根据中间结果决定下一步执行什么）。这是长提示词几乎不可能实现的。

第四是可维护性。MAS系统有显著的可维护性优势：自然语言定义使非程序员也可以理解和维护智能体蓝图，配置驱动使修改参考库即可调整行为无需改"代码"，完整审计追踪是文件管道的副产品所有中间步骤都可见，渐进式优化可以单独优化某个智能体立即看到效果。

4. 运行时环境

有了MAS的概念，下一个问题是：在哪里运行？

4.1 MAS对运行时的核心要求

一个完备的MAS运行时环境需要支持四项核心能力：

1. 独立上下文窗口管理：能够创建新的上下文窗口（启动新的智能体实例），运行完毕后释放资源（回收上下文窗口），多个实例之间不共享上下文。
2. 文件系统操作：读取输入文件和参考库，写入输出文件，创建和管理目录结构，进行文件路径操作（相对路径、绝对路径）。
3. 并行任务调度：同时运行多个智能体实例，自动管理并发和资源，等待所有并行任务完成后再继续。
4. 参数传递机制：向智能体传递运行时参数，支持变量替换（如 {TOPIC} 、{COUNTRY_CODE} ），参数可以是简单值或较长文本。

并且这四项能力都需要运行时环境背后的大模型（例如Claude Sonnet）能根据具体、明确（但并非一板一眼）的要求灵活调度。

4.2 不同环境的能力层次

让我们对比几种常见的运行时环境：

环境类型	上下文隔离	文件操作	并行调度	参数传递	综合评价
Web界面 (ChatGPT/Gemini)	需手动新建会话	需手动复制粘贴	无法并行	可在提示词中	低效但可行
Claude Code	Subagent原生支持	Bash工具	自动并行Task	参数替换	最便捷
编程框架 (AutoGen/LangGraph)	代码控制	代码控制	代码控制	代码控制	功能完备，需编程

Web界面的局限性在于你需要手动在ChatGPT/Gemini中打开新的会话（模拟"新上下文窗口"），需要手动复制粘贴文件内容在会话之间传递数据，无法自动化、无法并行，但理论上可行只是效率极低。

Claude Code的便捷性在于Subagent能力可以启动子智能体自动管理独立上下文，Bash工具可以读写文件操作目录，Task并行可以同时运行多个Task自动等待完成，参数传递在提示词中使用变量替换，这使得Claude Code成为快速原型和实验的理想环境。

编程框架的功能性在于AutoGen、LangGraph、CrewAI等框架提供完整的MAS能力，但需要编写Python代码来定义和编排智能体，适合需要复杂逻辑控制、版本管理、团队协作的场景，学习曲线较陡但功能完备。

4.3 为什么是现在？技术成熟度的临界点

多智能体的概念并不新鲜（学术界早就有JADE等框架），但为什么MAS现在才变得实用？因为多项技术同时成熟。

- • **LLM能力突破**使Claude 3.5 Sonnet、GPT-4等模型能够理解复杂的智能体蓝图，能够理解协作关系和数据依赖，能够自主决策执行策略而不是机械执行指令。
- • **长上下文窗口**让单个智能体可以处理相当复杂的任务，可以一次性读取大量参考资料和输入数据，降低了模块划分的压力。
- • **工具使用能力**让LLM可以操作文件系统，可以调用搜索引擎、API、数据库，从"对话伙伴"进化为"行动者"。
- • **成本下降**使Token价格降到可以大规模运行复杂系统的程度，一个包含数十次智能体调用的MAS任务成本可以控制在几美元，使得MAS从实验室概念变为实用工具。

这些因素共同作用，使MAS从理论概念变为可以日常使用的实用技术。

5. 实例：区域研究的多智能体系统

概念再清晰，也需要具体的例子来帮助理解。让我们看一个实际的MAS应用：一个用于区域国家研究的多智能体系统。

5.1 任务背景

研究目标是全面分析某个国家在特定时期的经济、政治、社会状况，生成一份结构化的研究报告。复杂度分析显示这个任务需要收集大量信息（新闻报道、学术文献、统计数据、历史背景等），需要应用多种分析框架（阶级分析、矛盾分析、经济评估等），需要生成结构化的研究报告符合特定风格和格式，信息量和任务复杂度远超单次LLM调用的处理能力。

如果用长提示词，你需要准备一个约3-5万words的超长提示词，包含所有步骤的指令和所有参考材料。即使这样，你也面临我们前面讨论的所有问题：注意力分散、缺乏隔离、无法并行、流程僵化、维护困难。

用MAS实现，我们将这个复杂任务拆解为一个三阶段流水线，由7个专门化的智能体协作完成。

5.2 系统架构

整个系统遵循一个清晰的三阶段流水线架构：

- • **Phase 1: 数据收集** → 输出到 data/01.materials/
- • **Phase 2: 分析** → 输出到 data/02.analysis/
- • **Phase 3: 报告生成** → 输出到 data/03.reports/

每个阶段的输出成为下一阶段的输入，通过文件系统传递数据。

系统包含7个智能体，按职能分工如下：

0. orchestrator（编排者）

协调整个研究流程的特殊智能体，理解用户需求、制定执行计划、依次调用各个智能体、监控进度。

- • 输入：国家代码（如"ARG"）、研究维度、分析框架
- • 工作流程：读取用户需求和参考库 → 创建目录结构 → 依次调用各阶段智能体 → 检查输出质量 → 必要时循环调用
- • 输出：无（协调者不生产内容）

Phase 1: 数据收集智能体

- • **1.1 broad-reconnaissance（广泛侦察）**
  - • 职责：快速收集该国基本信息和重要事件，建立信息地图
  - • 输入：国家代码参数
  - • 输出： data/01.materials/overview.md （概览文档，包含主题列表）
- • **1.2 focused-collection（聚焦收集）**
  - • 职责：针对关键主题深度收集信息
  - • 输入：国家代码 + overview.md
  - • 输出： topic-economy.md 、 topic-politics.md 、 topic-society.md 等
  - • 注：可并行启动3-5个实例分别收集不同主题域

Phase 2: 分析智能体（三个智能体可并行执行）

- • **2.1 contradiction-analyzer（矛盾分析器）**
  - • 职责：应用辩证法识别社会主要矛盾和次要矛盾
  - • 输入：所有材料文件 + references/dialectical-analysis-method.md
  - • 输出： data/02.analysis/contradictions.md
- • **2.2 class-analyzer（阶级分析器）**
  - • 职责：分析阶级构成和阶级斗争形势
  - • 输入：所有材料文件 + references/class-analysis-framework.md
  - • 输出： data/02.analysis/class-structure.md
- • **2.3 economic-evaluator（经济评估器）**
  - • 职责：评估经济状况和发展趋势
  - • 输入：所有材料文件（特别关注经济主题）
  - • 输出： data/02.analysis/economic-assessment.md

Phase 3: 报告生成智能体

- • **3.1 report-synthesizer（报告综合器）**
  - • 职责：将所有分析综合成结构化、风格统一的最终报告

- 输入：所有分析结果 + 报告模板 + 风格指南
- 输出： data/03.reports/final-report.md

5.3 模块边界的体现

这个系统完美地体现了模块化的四个关键特征。

边界隔离，体现在 focused-collection 只负责收集材料不需要知道后续会有什么分析，contradiction-analyzer 只专注矛盾分析不知道其他分析器在做什么，各模块的修改不会互相影响：改进阶级分析的方法不会影响矛盾分析。

独立生命周期，体现在 broad-reconnaissance 执行完毕后上下文窗口立即释放，即使它处理了10万tokens的信息也不影响后续智能体的上下文预算，每个智能体都是按需创建用完销毁。

清晰接口，体现在每个智能体都明确知道输入从哪里来（参数如国家代码、文件路径如 overview.md 、参考库如 class-analysis-framework.md ），输出写到哪里（具体的文件路径如 data/02.analysis/contradictions.md ），职责是什么（角色定义明确不越界），调用者（orchestrator）只需要知道接口不需要知道内部实现。

透明可追溯，体现在所有中间产物都保存为文件人类可以随时检查，可以看到 overview.md 识别了哪些主题，可以看到 topic-economy.md 收集了什么信息，可以看到 contradictions.md 的分析质量如何，出问题可以从任何中间阶段重新开始而不需要从头运行。

5.4 与长提示词的对比

让我们明确对比一下，同样的任务用长提示词和MAS实现的差异。

如果用长提示词实现相同任务，你需要把所有指令和参考材料拼接成一个完整的提示词（总长度约3万多字），包括：

- 角色定义与总体要求（约500字）
- 第一步：广泛侦察（约800字）
- 第二步：聚焦收集（约1000字）
- 第三步：矛盾分析（约1200字）
- 第四步：阶级分析（约1000字）
- 第五步：经济评估（约800字）
- 第六步：报告综合（约1000字）
- 附件1：马克思主义认识论框架（约1万字）
- 附件2：辩证分析方法（约5000字）
- 附件3：阶级分析范式（约8000字）
- 附件4：报告模板（约2000字）
- 附件5：写作风格指南（约1500字）

这种做法的问题在于：

- • 注意力分散：虽然没超上下文窗口但LLM需要同时"记住"6个步骤的要求，在执行第五步时第一步的细节可能已经模糊了
- • 缺乏隔离：修改辩证分析的方法即附件2可能意外影响到阶级分析的执行，因为所有内容都在一个上下文中LLM可能产生意外的"串扰"
- • 无法并行：必须顺序执行全部6步即使矛盾分析、阶级分析、经济评估三个步骤其实可以同时进行
- • 流程僵化（想让"聚焦收集"循环三轮每轮收集后评估效果查漏补缺在长提示词中几乎无法实现，你无法让LLM"暂停"检查中间结果再"继续"）
- • 维护困难（想调整风格指南需要重新装载整个3万字的提示词，想增加"国际关系分析"整个提示词更长复杂度指数上升）

用MAS实现的时候，系统被拆分为7个独立智能体。每个智能体蓝图500–1200字，每个智能体运行时只装载自己需要的参考库（1–2个文档合计1万字左右）。更加简洁清晰的同时，还带来一些好处：

- • **并行执行**。可以同时启动3–5个智能体收集不同问题域的信息（如经济、政治、社会）彼此无冲突，Phase 2的三个分析器也可并行运行节省大量时间。
- • **循环迭代**。orchestrator可以调用focused–collection三轮，每轮检查收集质量动态决定是否继续。
- • **独立维护**。修改辩证分析方法只需更新 dialectical–analysis–method.md 参考库不影响其他智能体，增加"国际关系分析"只需增加一个新智能体。
- • **透明可控**。每个阶段的输出都可见可以人工审查可以从任何阶段重新开始。

6. 从模块到设计原则

到这里，我们已经建立了MAS的完整概念体系，也看到了一个具体的实例。但这只是起点。

6.1 提升讨论层次

有了MAS的概念体系后，我们对AI应用的讨论就不再是"提示词怎么写更有效"、"怎么让LLM理解我的意图"、"怎么避免LLM偏离主题"，而是升级为"这个智能体的职责是否单一"、"模块之间的耦合度是否合理"、"接口设计是否清晰"、"系统是否易于扩展和维护"、"如何权衡模块粒度和通信成本"。这是设计层面的讨论，而不是技巧层面的调试。就像软件工程从"怎么写代码"提升到"怎么设计系统"一样，MAS让我们从"怎么写提示词"提升到"怎么设计智能体系统"。

6.2 软件工程设计原则的适用性

既然MAS是模块化系统，那么经典的软件设计原则就可以应用，例如：

- **单一职责原则（SRP – Single Responsibility Principle）**。要求一个智能体应该只有一个明确的职责，只有一个改变的理由。
- **开放–封闭原则（OCP – Open–Closed Principle）**。要求智能体应该对扩展开放对修改封闭。
- **依赖倒置原则（DIP – Dependency Inversion Principle）**。要求智能体应该依赖抽象接口而不是具体实现。
- **迪米特法则（LoD – Law of Demeter）**。要求智能体应该只与直接协作者通信，不应该了解系统的全局结构。
- **合成复用原则（CRP – Composite Reuse Principle）**。要求优先通过组合智能体实现复杂功能，而不是试图编写一个"全能"智能体。

6.3 下一步：设计原则的深入探讨

这些原则的简要介绍只是开胃菜。在后续文章中，我们将结合过去软件设计的经验，详细讨论更多智能体设计的相关问题，例如（但不限于）：

- 如何将这些原则应用到智能体设计中（每个原则的详细解释和实例、如何识别违反原则的设计、如何重构以遵循原则）
- 智能体设计的常见模式（Orchestrator模式、Pipeline模式、Parallel Execution模式、Iterative Refinement模式、Specialist Collaboration模式等）
- 如何评估一个MAS设计的质量（模块内聚度Cohesion指标、模块耦合度Coupling指标、可测试性Testability评估、可维护性Maintainability评估等）
- MAS设计的权衡和挑战（模块粒度的选择太粗还是太细、通信成本vs隔离收益、调试和监控的策略、错误处理和恢复机制等）。

附录：关键术语对照表

中文	英文	说明
多智能体系统	Multi–Agent System (MAS)	多个智能体协作的系统
智能体	Agent	用LLM实现的模块
智能体蓝图	Agent Blueprint	智能体的定义文档（源代码）
智能体实例	Agent Instance	运行中的智能体（对象实例）
编排者	Orchestrator	协调其他智能体的特殊智能体
文件管道	File–based Pipeline	通过文件系统传递数据的方式
参考库	Reference Library	共享的领域知识文档
上下文窗口	Context Window	LLM一次可处理的token数量
模块	Module	软件工程中的通用概念
边界	Boundary	模块的职责范围

中文	英文	说明
生命周期	Lifecycle	创建、执行、销毁的过程
接口	Interface	输入输出的定义
隔离	Isolation	模块间互不干扰的特性

结语

从长提示词的困境出发，我们看到了软件工程模块化的启示，建立了多智能体系统的概念体系，探讨了运行时环境的要求，通过一个具体实例理解了MAS的运作方式，最后指出了下一步设计原则的方向。

多智能体系统可能昭示着AI时代软件的一种新形态。不是传统的"编写代码"而是"设计智能体"，不是"编译执行"而是"编排协作"，不是"函数调用"而是"模块对话"，但本质上还是软件工程，只是用自然语言而非编程语言来表达。

对于软件从业者来说，这是一个激动人心的转折点。我们积累的软件工程知识——模块化、设计原则、架构模式——并没有过时，反而找到了新的应用场景。MAS不是对软件工程的颠覆，而是继承和演进。

熊节

Like the Author