Agent全面爆发！万字长文详解上下文工程

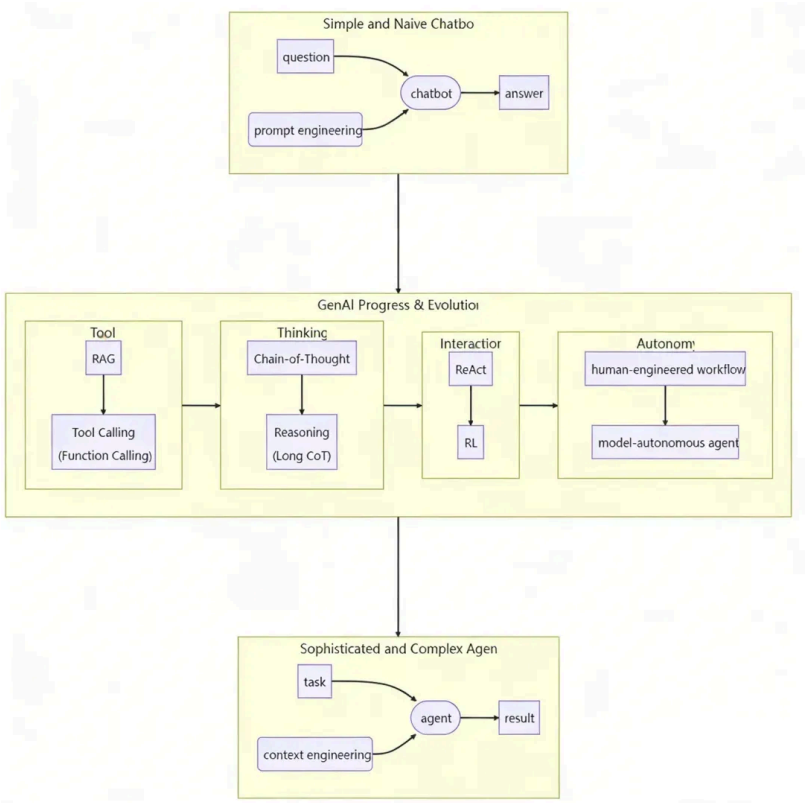
原创 腾讯云开发者 腾讯云开发者 2026年1月7日 08:46 北京

01

Agent 全面爆发的前夜：上下文正在成为核心变量

1.1 从 Chatbot 到 Agent：能力形态的变化

在大语言模型开始被应用到真实产品之前，Chatbot 是最常见的一种形态。它的工作流程相对简单，模型接收用户输入，在当前上下文中生成一次回应，交互随之结束。



在这种模式下，模型关注的重点集中在单轮或有限多轮对话内。工程实践更多围绕提示词展开，包括措辞选择、角色描述以及少量上下文拼接技巧。只要输入清晰，模型通常就能给出可用的输出。

随着应用场景逐渐复杂，这种交互方式开始暴露出问题。许多任务并不能在一次对话中完成，而是需要经过多个步骤，期间还会不断产生新的中间结果。模型如果只能看到当前输入，就无法判断这些信息在整体流程中的位置，也难以维持行为上的连贯性。

Agent 的概念正是在这种背景下被提出的。它要求模型围绕某个目标持续工作，能够根据已有信息判断当前进展，并决定接下来应该采取的行动。为了支撑这种能力，模型需要看到的内容明显增多，上下文开始承担起更重要的角色。

1.2 Agent 出现的技术动因

从工程角度来看，Agent 的出现是系统需求发生了深切的变化。在 Agent 场景中，模型需要处理的不再只是问题本身，还包括问题所处的阶段、已经完成的步骤以及尚未解决的子任务。与此同时，模型往往需要借助外部工具来获取信息或执行操作，这些行为都会对后续决策产生影响。

这些变化使得模型必须具备对历史信息的感知能力。它需要理解哪些行为已经发生，哪些结果已经产生，以及当前状态距离目标还有多远。如果这些信息无法被有效传递给模型，Agent 的行为就会变得不稳定，甚至出现反复试错却无法推进的问题。

因此，Agent 的本质需求可以归结为一点：模型需要在更大的时间尺度上理解任务上下文。

1.3 上下文角色的重新定位

在早期应用中，上下文更多被视为对输入的补充，用来帮助模型理解当前问题的背景。这种用法下，上下文的存在与否，并不会对系统结构产生决定性影响。

进入 Agent 阶段后，上下文开始承担系统状态的表达职责。模型的每一次决策，都会受到上下文内容的直接影响。上下文中包含的信息越完整，模型越容易判断当前所处的位置以及下一步的行动方向。

这也意味着，上下文不再只是被动拼接的文本，变成了一种需要被精心管理的资源。哪些信息应该长期保留，哪些只在短时间内有效，哪些需要反复强调，这些问题都会对 Agent 的行为产生实际影响。

在这一阶段，模型的能力表现，很大程度上取决于上下文的组织方式。

1.4 提示工程向上下文工程的自然演进

在 Agent 出现之前，提示工程是提升模型表现的主要手段。通过设计合适的提示词，可以在一定范围内引导模型输出符合预期的结果。

当系统开始引入多轮决策与复杂状态时，仅依赖提示词已经难以满足控制需求。提示词更适合描述规则与格式，却不擅长表达持续变化的任务状态。随着上下文内容不断增长，单一提示往往会被淹没在大量信息之中。

正是在这种情况下，上下文工程逐渐被单独拎出来讨论。它关注的问题不再是某一句提示写得是否精巧，而是整个上下文如何被构建、更新和裁剪，以确保模型始终获取到对当前决策最有价值的信息。

这一转变，为后续 Agent 系统的工程化奠定了基础。

02

提示词工程的能力边界

在大语言模型刚被引入实际应用时，提示词工程发挥了极其关键的作用。通过对输入内容进行精心设计，可以在不改动模型本身的前提下，显著改善输出质量。这种方式成本低、见效快，很快成为开发者最常用的调优手段。

在这一阶段，提示词主要承担两个任务：一是明确模型的角色和行为边界，二是约束输出的格式和风格。只要任务本身相对简单，提示词往往就足以支撑一个可用的应用。

这也是为什么，在相当长的一段时间内，人们会把模型能力的提升，与提示词写得好不好直接画上等号。

当任务开始涉及多步骤推理和持续决策时，提示词工程的局限逐渐显现出来。

提示词本质上是一种静态描述，它擅长表达规则，却不擅长反映变化。随着任务流程拉长，模型需要参考的信息越来越多，单一提示往往会被新的上下文内容不断稀释，最终失去控制力。

在这种情况下，开发者往往只能通过不断叠加提示内容来补救。结果是提示词本身变得越来越长，结构也越来越复杂，维护成本随之上升。

另一个不可忽视的问题，是提示词工程的可维护性。

提示词效果往往高度依赖具体模型版本和参数设置。当模型升级或环境发生变化时，原本表现良好的提示词，可能会突然失效。这种不确定性，使得提示词难以作为长期稳定的工程方案。

对于需要持续运行的系统而言，这种不稳定性会被不断放大。一旦提示词出现偏移，整个 Agent 的行为就可能发生变化，排查和修复成本也会迅速增加。

随着 Agent 系统逐渐成熟，人们开始意识到，单靠提示词已经无法承担全部控制职责。

模型的行为不只受到提示内容的影响，还与上下文中呈现的信息结构密切相关。即使提示词本身保持不变，只要上下文发生变化，模型的决策结果也可能出现明显差异。

这促使工程实践开始向更系统化的方向发展，提示词逐渐退居为其中的一个组成部分，而不再是唯一的调控工具。

03

上下文工程：一个被低估的系统问题

在实际工程中，最常见的一种误解，是把上下文工程简单理解为“尽可能多地把信息塞给模型”。

在上下文窗口不断扩大的背景下，这种做法看起来顺理成章，也一度被视为提升模型能力的直接手段。

但随着实践深入，问题很快暴露出来。模型虽然能够看到更多内容，却并不一定能从中提取出真正有用的信息。大量无关或低价值的上下文，反而会干扰模型的判断，使输出变得不稳定。

从工程视角来看，上下文是一种受限资源。它不仅受到长度限制，还受到模型注意力分布和推理能力的影响。上下文中每一段内容，都会与其他内容竞争模型的关注度。

如果上下文缺乏清晰的结构，模型就难以判断哪些信息是当前决策所必需的，哪些只是背景补充。最终表现为模型在不同轮次中给出风格和质量差异较大的输出。

因此，上下文工程需要解决的，是如何在有限空间内，持续为模型提供高价值信息，并避免无效干扰。

在 Agent 系统中，上下文往往需要承担状态表达的职责。模型需要通过上下文了解当前任务的进展情况，包括已经完成的步骤、产生的中间结果以及尚未解决的问题。

如果这些状态信息没有被明确表达，而是隐含在零散的历史对话中，模型就很难形成稳定的任务认知。即使模型本身具备较强的推理能力，也容易出现重复执行或偏离目标的情况。

将状态信息显性化，是上下文工程中一个非常重要的实践原则。

随着 Agent 持续运行，上下文内容不可避免地会不断增长。如果缺乏有效的更新与裁剪机制，上下文很快就会失控，既增加成本，也降低效果。

工程实践中，通常需要明确哪些信息是短期有效的，哪些需要被长期保留。通过定期整理和压缩上下文，可以让模型始终关注当前阶段最相关的内容。

这种动态管理方式，是上下文工程区别于提示工程的关键特征之一。

上下文工程所包含的模块

4.1工具：让模型具备行动能力

在 Agent 系统中，工具是模型能力向外扩展的主要方式。通过工具调用，模型可以访问外部数据源、执行计算任务，甚至对现实系统产生直接影响。

以当前智能体（或者LLM）的视角来看，环境中所有它可以交互的对象都是它可以利用的工具，包括各种应用程序、各种网站服务、其他AI应用、其他智能体以及甚至人类（包括但不限于当前用户）。

正如Factor 7: Contact humans with tool calls和HuggingFace 博客Tiny Agents所展示的那样，每个智能体都可以主动地请求人类的介入，遇到歧义的地方可以主动寻求用户来澄清然后再进行下一步操作（见下图和下面的代码示例）。此时的用户就是当前智能体可以调用的工具。

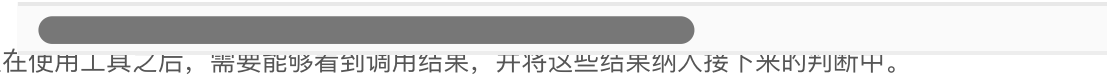
不仅如此，生成最终答案（见Factor 7: Contact humans with tool calls）和指定任务状态（比如任务完成，见HuggingFace 博客Tiny Agents）也都能作为工具来调用，真的是万物工具化（everything is tools）。

```
const taskCompletionTool: ChatCompletionInputTool = {
    type: "function",
    function: {
        name: "task_complete",
        description: "Call this tool when the task given by the user i
        parameters: {
            type: "object",
```

```
            properties: {},
        },
    },
};
const askQuestionTool: ChatCompletionInputTool = {
    type: "function",
    function: {
        name: "ask_question",
        description: "Ask a question to the user to get more info requ
        parameters: {
            type: "object",
            properties: {},
        },
    },
};
const exitLoopTools = [taskCompletionTool, askQuestionTool];
```

4.2 工具调用结果的上下文反馈

工具调用并不是一次性行为，它会对后续决策产生持续影响。



模型在使用工具之后，需要能够看到调用结果，并将这些结果纳入接下来的判断中。

如果工具返回的信息只是简单拼接进上下文，而缺乏明确结构，模型往往难以判断哪些内容是关键结论，哪些只是过程性信息。长时间运行后，这种混乱会不断累积，最终影响 Agent 的整体稳定性。

因此，工具调用结果需要被结构化地写回上下文，使模型能够快速识别其语义角色。

4.3 思考过程的上下文表达

在复杂任务中，模型往往需要经过多步推理才能得出结论。

这些中间思考过程，对模型自身而言是有价值的，但并不意味着它们都应该被完整保留下来。

上下文工程需要在可解释性与效率之间做出权衡。某些阶段，显式保留中间推理有助于模型稳定决策；而在其他阶段，过多的推理痕迹反而会干扰后续判断。

工程实践中，常见的做法是对思考过程进行分层管理，仅在必要时将其纳入上下文。

4.4 交互：Agent 与环境的持续协同

Agent 并不是孤立运行的，它需要不断与外部环境交互。这种交互既包括成功执行后的正向反馈，也包括失败、异常以及不符合预期的结果。

上下文需要完整记录这些交互结果，让模型能够基于真实反馈调整行为策略。如果交互信息被忽略或丢失，模型很容易重复同样的错误。

因此，交互信息的管理，是上下文工程中不可或缺的一部分。

4.5 自主性与约束的平衡

随着 Agent 能力增强，模型的自主决策空间也在不断扩大。如果缺乏必要约束，模型可能会偏离预期目标，甚至产生不可控行为。

上下文工程在这里承担的是边界设定的角色。通过明确目标、约束条件以及可接受的行动范围，可以在不干预具体决策的情况下，引导模型行为朝着预期方向发展。

这种设计方式，使 Agent 能够在保持一定自由度的同时，避免行为失控。

05

MCP：上下文工程的结构化落地方式

当 Agent 系统逐渐复杂，上下文内容开始呈现出明显的工程问题。

不同模块生成的信息被不断拼接进上下文，状态、工具描述、历史记录混在一起，结构逐渐变得模糊。

在这种情况下，即便模型能力足够强，也很难稳定地理解当前上下文中各类信息的角色。工程上的问题并不在模型本身，而在于上下文缺乏统一的组织方式。

MCP 正是在这样的背景下被提出的。它试图解决的，是上下文在系统层面的可读性与可维护性问题。

MCP 的核心思路，是将上下文视为一种可以被规范化的接口，而不是自由拼接的文本。

在 MCP 体系中，不同类型的信息被明确区分，例如任务状态、可用能力、行为约束以及外部反馈。每一类信息都有相对固定的表达方式，模型可以据此判断其在决策中的作用。

这种做法并不会限制模型的生成能力，而是降低理解成本，使模型能够更快地抓住当前最重要的信息。

当上下文被结构化之后，系统的整体行为会变得更加可预测。模型在不同轮次中面对的上下文形式保持一致，即使具体内容发生变化，其语义角色依然清晰。

Manus上下文工程实践：将不用的工具掩盖掉，而不是移除掉

对于持续运行的 Agent 系统而言，这种一致性尤为重要。它可以显著降低行为漂移的风险，也让问题排查变得更加直接。

从工程角度看，结构化上下文也更利于模块化设计，不同组件可以围绕统一协议协同工作。

MCP 并非独立存在的组件，而是服务于整个 Agent 系统。通过 MCP，Agent 在运行过程中产生的状态变化、工具调用结果以及环境反馈，都可以被有序地写回上下文。模型在后续决策时，看到的是一个经过整理的状态快照，而不是零散的信息碎片。

这种协同方式，使得上下文真正成为系统运转的一部分。

06

智能体系统可靠性

作为一个智能系统，其稳定性需要额外关注。而上下文工程中的其中一环就是需要从工程的角度加强系统稳定性。

LangChain 博客How and when to build multi-agent systems中讲到了如何利用LangChain全家桶来构建和增强系统的稳定性或者**可靠性（reliability）**，主要包括以下维度：

- 可靠的执行和错误处理，这需要一个智能体编排框架，比如LangGraph。
- 智能体调试和可观测性，这需要一个追踪、调试、交互系统，比如LangSmith和LangGraph Studio。
  - LangSmith 是一个统一的用于观测和评估AI应用的平台。具体地，LangSmith 可以调试、评估和监控AI应用及其性能。而且虽然LangSmith对LangChain的原生支持更好，但是它是独立于LangChain的，所以也可以很方便地应用于其他AI应用框架。
  - LangGraph Studio 是一个智能体集成开发环境（agent IDE），专门用于可视化、交互和调试基于LangGraph构建的智能体系统（agentic system）。而且LangGraph Studio集成了LangSmith，所以也支持观测和评估智能体系统。
- 评估：无论是人工评估还是LLM评估（即LLM-as-a-Judge），在原型制作阶段可以先人工标注一些测试样本，并且可以借助于LLM来不断充实和完善。
  - LangSmith 也可以用于自动化或更高效地评估AI应用。

监控日志一方面供算法追踪和调试，另一方便还可以提供或者展示给用户。这样不仅可以增加AI应用的透明度和可解释性，还可以提升用户体验和便于建立AI应用与用户之间的信任。而且可以将LangGraph Studio提供给用户体验，那么在Studio集成的LangSmith监控系统的基础上，用户可以

可视化地无代码地与智能体系统进行交互。这样不仅便于AI应用的开发测试人员收据用户反馈，而且用户使用体验和反馈更加真实有效。

07

总结与展望

从Chatbot到Agent，LLMs 的上下文也变得更加复杂多样。之前的提示工程主要围绕系统和用户提示词，而如今的上下文工程还要处理状态、记忆、工具和结构化输出等内容及其之间的协同工作。

在长期运行、多轮决策和工具协同的场景下，模型能否稳定发挥，往往取决于它所“看到”的上下文是否清晰、是否连贯、是否始终围绕当前目标展开。上下文不再只是输入的一部分，而是系统状态的集中体现。

这也解释了为什么，在 Agent 体系中，单纯依赖提示词已经难以支撑复杂需求。提示词仍然有价值，但更多承担的是局部约束和行为引导的角色。真正决定系统上限的，是上下文如何被构建、更新和管理。

从这个角度来看，上下文工程背后设计到的是一类系统性问题。它涉及信息筛选、状态表达、结构设计以及运行过程中的动态调整。这些问题一旦处理不当，模型能力再强，也难以转化为稳定输出。

随着 Agent 应用不断扩展，上下文工程的重要性只会进一步凸显。在 Agent 时代，这种可控性，正逐渐成为一项基础能力。

AI 正在从人类数据时代进入经验时代，即从收集和利用人类数据来训练模型到让智能体在与外界环境的自主交互中经历和成长。根据使用工具的经历来更新使用工具的经验只是其中一个典型案例。未来AI 的经验学习（experiential learning，俗称“干中学”）将更加贴近甚至超越人类。

-End-

感谢你读到这里，不如关注一下？👇

腾讯云开发者

腾讯云官方社区公众号，汇聚技术开发者群体，分享技术干货，打造技术影响力交流社...
1064篇原创内容

公众号

📣📣来抢开发者限席名额！点击下方图片直达👇

你对本文内容有哪些看法？同意、反对、困惑的地方是？欢迎留言，我们将邀请作者针对性回复你的评论，欢迎评论留言补充。我们将选取1则优质的评论，送出腾讯云定制文件袋套装1个（见下图）。1月14日中午12点开奖。