递归语言模型

一份针对开创性论文《递归语言模型》的完整翻译与深度解读报告

第一部分：引言与核心概念——为新范式奠定基础

1.1. 论文标题、作者与发表信息

【原文翻译】

递归语言模型

我们提出了递归语言模型（Recursive Language Models, RLMs），这是一种推理策略，语言模型可以通过 REPL 环境对无限长度的输入上下文进行分解和递归交互。

作者 Alex Zhang Omar Khattab

发表日期 2025年10月15日

隶属机构 MIT CSAIL MIT CSAIL

【深度解读】

在深入探讨这篇论文的技术细节之前，我们首先需要理解其“出身”的重要性。这篇论文的作者来自**麻省理工学院计算机科学与人工智能实验室（MIT CSAIL）**，这是全球最顶尖的人工智能研究机构之一。这个背景告诉我们，这项研究处于世界科技的前沿，极有可能对人工智能领域产生深远影响 。

论文的第二作者 Omar Khattab 并非无名之辈，而是在**信息检索（Information Retrieval, IR）领域享有盛誉的学者。信息检索研究的核心就是如何让计算机高效地从海量数据中找到用户需要的信息，这正是搜索引擎等应用背后的关键技术。Khattab 教授的专业背景暗示了这篇论文的一个核心思想：它不仅仅是关于语言模型本身，更是关于如何设计一个更聪明的信息处理系统** 。

此外，这篇文章以博客文章的形式于2025年10月15日发表，这是一种在正式的同行评审论文发表前，与学术界快速分享前沿成果的方式。你可以将其理解为科学研究的“现场直播”，展示了最尖端、最新鲜的科学思想。

1.2. 摘要与“太长不看版”——论文精髓概览

【原文翻译】

太长不看版 (tl;dr) 我们探索了一种在给出最终答案前，会递归调用自身或其他大语言模型（LLMs）的语言模型。我们的目标是使其能够处理基本上无限长度的输入上下文和输出，并减轻性能下降的“上下文退化”（context rot）问题。

我们提出了递归语言模型（Recursive Language Models, 或 RLMs），这是一种通用的推理策略，语言模型可以将其输入上下文作为一个变量进行分解和递归交互。我们设计了一个具体的实例，其中 GPT-5 或 GPT-5-mini 在一个 Python REPL 环境中被查询，该环境将用户的提示存储在一个变量里。

我们证明了，在一个我们能找到的最困难的长上下文基准测试（OOLONG）的部分数据集上，使用 GPT-5-mini 的 RLM 在正确答案数量上比 GPT-5 的表现好出一倍以上，并且平均每次查询的成本更低！我们还基于 BrowseComp-Plus 121 构建了一个新的长上下文深度研究任务。在该任务上，我们观察到 RLM 的性能优于其他方法，如 ReAct 结合测试时索引和在提示上进行检索。令人惊讶的是，我们发现即使在推理时给定超过1000万个词元（tokens）的上下文，RLM 的性能也不会下降。

我们很高兴能分享这些非常早期的结果，并论证 RLM 将很快成为一个强大的范式。我们认为，经过专门训练以进行递归推理的 RLM，很可能代表继“思维链”（CoT-style）推理模型和“代理”（ReAct-style）模型之后，在通用推理能力扩展方面的下一个里程碑。

我们在原始推文中有个压缩版的摘要：
https://x.com/alzhang/status/1978469116542337259

【深度解读】

这段摘要信息量巨大，让我们用一个比喻来解构它。

想象一下，你让一个非常聪明的学生（一个标准的大语言模型，如 GPT-5）去读一本厚厚的历史书，然后回答一个复杂的问题。如果这本书太长，读到后面时，他可能会忘记开头的细节。这种现象就是论文中提到的**“上下文退化”（context rot）**。这就像我们在准备一门期末考试时，学到最后一章，第一章的内容已经变得模糊不清了。

这篇论文提出的解决方案，并不是让这个学生变得更聪明（即改变模型本身），而是给了他一套全新的、更高效的**学习和解决问题的方法**（即“推理策略”）。这个方法就是**递归语言模型（RLM）**。

具体来说，这个新方法包含两个关键点：

1. **递归调用**：这个学生在解决主问题时，可以把大问题拆解成许多小问题，然后把这些小问题“外包”给自己或别的同学去解决，最后再把所有答案汇总起来。这就是“递归调用自身或其他 LLMs”。

2. **REPL 环境**：我们给了这个学生一个“可编程的智能草稿本”（即 Python REPL 环境）。这本历史书（长上下文）不再需要他一口气读完并记在脑子里，而是被存放在这个草稿本里。他可以通过编写简单的代码指令来查阅、筛选、总结书中的任意部分，而不会让自己的大脑“内存”过载。

这篇论文最令人振奋的成果是：一个稍弱一些的模型（GPT-5-mini，可以看作一个普通班的学生），在使用 RLM 这套新方法后，解决复杂问题的能力竟然**超过了**顶尖模型（GPT-5，可以看作尖子班的学生），而且完成任务的**成本更低**。这证明了“方法比天赋更重要”。甚至当信息量扩大到1000万词元（相当于一个小型图书馆）时，这套方法的表现依然稳定，这正是它被称为能处理“无限上下文”的原因 。

一个更深层次的思考是，为什么这种方法如此巧妙。在人工智能领域，有一个被称为**“递归的诅咒”（The Curse of Recursion）的问题，指的是如果一个模型不断学习由自己或其他AI生成的数据，它会陷入一种退化的恶性循环，最终“忘记”真实世界的知识，导致“模型崩溃”（model collapse）。而 RLM 巧妙地避开了这个陷阱。它把“递归”严格限制在解决问题（推理）**的环节，而不是**学习（训练）**的环节。这意味着模型的知识库始终是纯净的、源于人类数据的，只是在解决具体问题时，它会使用递归这种聪明的策略来组织和处理信息。这个设计上的区分，是 RLM 强大且安全的关键。

1.3. RLM 架构示意图

【原文翻译】

（图表描述） 下图展示了一个递归语言模型（RLM）的调用流程。

- 流程始于用户的“查询”（query）和“上下文”（context）。
- 这两者被输入到顶层的“RLM（根/深度=0）”中。这个根 RLM 内部包含一个“语言模型”。
- 根 RLM 并不直接处理整个上下文，而是与一个“环境 E（例如 REPL）”进行交互。
- 通过这个环境，根 RLM 可以发起两个并行的“RLM（深度=1）”的子调用。
- 每个深度为1的 RLM 也包含一个“语言模型”，它们分别生成“子响应1”和“子响应2”。
- 这两个子响应被返回到环境中，并最终由根 RLM 汇总，生成“最终响应”（final response）并返回给用户。

【深度解读】

这张图直观地展示了 RLM 的工作模式，我们可以用一个“项目经理”的比喻来理解它。

- **根 RLM (root/depth=0)**：这就是项目经理。他接到了一个复杂的任务（用户的“查询”和庞大的项目资料“上下文”）。
- **环境 E (Environment E)**：这是项目经理的办公室，里面有电脑、文件柜等工具。项目资料（上下文）都存放在这里，而不是堆在经理的办公桌上。
- **RLM (depth=1)**：这些是项目经理手下的两名团队成员。
- **语言模型 (Language Model)**：这是每个成员（包括经理自己）的大脑，负责思考和处理信息。

整个流程是这样的：项目经理（根 RLM）接到任务后，他不会亲自阅读所有的资料。他会利用办公室的工具（环境 E），把任务分解成两个子任务，然后分别派给两名团队成员（深度=1的 RLM）。这两名成员各自独立工作，完成自己的部分后，把结果（子响应1和2）交回。最后，项目经理将这些结果汇总，形成一份完整的报告（最终响应）交给客户。

这个模式的优点显而易见：

1. **分而治之**：复杂任务被分解，处理起来更高效。
2. **减轻负担**：项目经理不需要记住所有细节，他的主要工作是分解任务和整合结果，大脑不会过载。
3. **可扩展性**：如果任务更复杂，经理可以雇佣更多的团队成员（发起更多的递归调用），理论上可以无限扩展。

这就是 RLM 架构的核心思想：通过一个管理和分解任务的“经理”层，让语言模型能够有条不紊地处理远超其自身记忆能力的海量信息 。

第二部分：问题的根源——深入理解“上下文退化”

2.1. 图1描述与“长上下文”研究的困境

【原文翻译】

图1. 一个递归语言模型（RLM）调用的例子，它扮演着一个从文本到文本（text→text）的映射角色，但比标准的语言模型调用更灵活，并且可以扩展到近乎无限的上下文长度。RLM 允许语言模型与一个存储着（可能非常巨大的）上下文的环境（在此例中是一个 REPL 环境）进行交互，在其中它可以递归地对“自身”、其他语言模型调用或其他的 RLM 调用进行子查询，以高效地解析这些上下文并提供最终响应。

前奏：为什么“长上下文”研究如此不尽如人意？

在语言模型（LMs）中，存在一个众所周知但难以描述的现象，叫做“上下文退化”（context rot）。Anthropic 公司将其定义为：“[当]上下文窗口中的词元数量增加时，模型从该上下文中准确回忆信息的能力会下降”，但社区中的许多研究人员都知道这个定义并未完全切中要害。例如，如果我们看看像 RULER 这样流行的“大海捞针”式基准测试，大多数前沿模型实际上表现得非常好（一年前的模型就能达到90%以上的准确率）。

（图片描述：一个南瓜灯的卡通漫画） *我让我的语言模型完成它昨天开始讲的那个关于雕刻南瓜的笑话。它说：“南瓜？什么南瓜？”——上下文完全退化了。*

但是人们已经注意到，上下文退化是一种奇怪的现象，当你的 Claude Code 历史记录变得臃肿，或者你和 ChatGPT 聊了很长时间后，它就会发生——就好像，随着对话的进行，模型变得……更笨了？这是一种众所周知但难以描述的失败模式，我们通常不会在论文中讨论它，因为我们无法对其进行基准测试。

一个自然而然的解决方案是这样的：“好吧，也许如果我把上下文分成两次模型调用，然后在第三次模型调用中将它们的结果结合起来，我就能避免这种性能下降的问题”。我们把这个直觉作为递归语言模型的基础。

【深度解读】

这一部分深刻地揭示了驱动这项研究的核心痛点：当前 AI 在处理长篇信息时的“智力衰退”问题。

首先，我们需要区分两种不同的“记忆”任务，这有助于我们理解为什么“上下文退化”如此棘手。

1. **“大海捞针”式任务**：这就像老师让你在课本第57页找到某句特定的名人名言。这是一个简单的信息定位任务。AI 在这方面非常出色，就像一个高效的文本搜索引擎，因此在 RULER 这类测试中得分很高 。

2. **“上下文退化”所描述的任务**：这更像是老师让你在读完整本历史书后，写一篇论文，需要你综合运用第1章、第5章和第12章的观点来论证一个复杂的主题。当你专注于第12章时，你对第1章的细节和微妙之处的记忆已经模糊了。AI 也面临同样的问题，在长时间的、连续的对话或分析中，它似乎会“忘记”早期的语境和指令，导致后续的回答质量下降，显得“变笨了”。

那个关于南瓜的笑话（“南瓜？什么南瓜？”）是一个绝佳的例子，它生动地展示了这种失败模式。模型不仅忘记了笑话的内容，甚至忘记了“南瓜”这个核心话题本身，这是一种彻底的语境丢失。

作者指出的一个关键问题是，这种“变笨”的现象很难用传统的**基准测试（benchmark）来量化。现有的测试大多是“大海捞针”式的，无法捕捉到在真实、动态的交互中模型推理能力的逐步衰退。这揭示了学术评估与用户实际体验之间的巨大鸿沟。用户能感觉**到模型在变笨，但现有的“考卷”却考不出这个问题。这正是这项研究的价值所在：它试图解决一个真实存在但难以衡量的重要问题。

而 RLM 的基本构想，正是源于一个非常符合直觉的想法：既然一口气处理不完，那就把它分开处理，最后再汇总。这就像我们写一篇长篇论文时，会先为每个章节打草稿，最后再把它们整合成一篇完整的文章，而不是试图一次性从头写到尾。这个简单的直觉，构成了 RLM 框架的基石。

2.2. 递归语言模型（RLM）的定义

【原文翻译】

递归语言模型 (RLMs).

一个递归语言模型是一个围绕着语言模型的“薄封装层”（thin wrapper），它可以为中间计算生成（递归的）语言模型调用——从用户或程序员的角度来看，它与一次模型调用是相同的。换句话说，你像查询一个语言模型一样通过“API”来查询一个 RLM，即 `rlm.completion(messages)` 是 `gpt5.completion(messages)` 的直接替代品。我们采用一

种**以上下文为中心**（context-centric）而非**以问题为中心**（problem-centric）的输入分解视角。这种框架保留了我们想要一个能够针对某个相关上下文回答特定查询的系统的功能性观点：

从用户 / API 的视角来看：

（图2描述） 图2展示了两种调用方式的对比。

- 上方是标准的语言模型调用：用户的“上下文”和“查询”直接输入给“语言模型”，然后输出“响应”。
- 下方是递归语言模型调用：用户的“上下文”和“查询”输入给“RLM”，然后输出“响应”。从外部看，两者的接口完全一样。

图2. 一次递归语言模型调用替代了一次语言模型调用。它为用户提供了近乎无限上下文的“幻觉”，而在底层，一个语言模型会相应地对上下文进行管理、分区，并递归地调用自身或另一个语言模型来避免上下文退化。

【深度解读】

这一节定义了 RLM 的核心身份。理解 RLM 的关键在于“**薄封装层**”（thin wrapper）和“**以上下文为中心**”这两个概念。

1. “薄封装层”：一个更聪明的任务管理器

想象一下，一个标准的语言模型调用（`gpt5.completion`）就像直接给一位才华横溢但没有经过系统训练的专家下达指令。你把所有资料（上下文）和问题（查询）都丢给他，然后等待他凭直觉给出一个答案。

而 RLM (`rlm.completion`) 则像是在这位专家身边配备了一位专业的项目助理。你仍然是和专家沟通（API 接口不变），但现在这位助理会帮助专家：

- 接收和整理所有资料，而不是让专家的大脑被信息淹没。
- 将复杂问题分解成专家可以一步步处理的小任务。
- 在专家需要时，帮他查找特定资料或进行一些辅助计算。

这个“助理”就是 RLM 这个“薄封装层”。它本身不产生最终答案，但它创建了一个高效的工作流程，让核心的“专家”（语言模型）能够发挥出远超其单打独钟时的水平。从用户的角度看，一切都没有变化，你只是感觉这个“专家”变得前所未有的强大和可靠，仿佛拥有了无限的记忆力，但这其实是工作流程优化带来的“幻觉”。

2. “以上下文为中心” vs. “以问题为中心”：思维模式的转变

这是一个非常深刻的转变，我们可以用解决数学应用题来类比：

- **以问题为中心（旧方法）**：拿到题目后，直接开始思考“我应该用哪个公式来解这道题？”。这种方法在问题简单、信息量少时很有效。但如果题目描述非常长，条件错综复杂，你可能会因为没有理清所有条件而出错。

- **以上下文为中心（RLM 的方法）**：拿到题目后，第一步不是解题，而是**理解和整理题目给出的所有信息（上下文）**。你会先把所有已知条件列出来，画出关系图，把冗余信息剔除。当你把整个“上下文”都梳理清楚之后，解题方法自然就浮现出来了。RLM 的哲学是，只要能高效地理解和驾驭信息本身，任何基于这些信息的问题都会变得容易解决。

这种思维模式的转变，使得 RLM 不仅仅是一个解决特定问题的工具，而是一个通用的、可扩展的**信息理解和处理框架**。

第三部分：解决方案——递归语言模型的内部架构

3.1. RLM 的底层工作机制：根模型与 REPL 环境

【原文翻译】

在底层，一个 RLM 只把**查询**提供给语言模型（我们称之为**根语言模型**，或深度为0的语言模型），并允许这个语言模型与一个**环境**进行交互，这个环境存储着（可能非常巨大的）**上下文**。

我们选择的环境是一个循环，在这个循环中，语言模型可以向一个 Python REPL 笔记本（类似于 Jupyter Notebook 环境）的单元格写入代码并读取其输出。这个笔记本预先将上下文加载为一个内存中的变量。**根语言模型**有能力在 REPL 环境中像调用代码中的一个函数一样，调用一个递归的语言模型（或深度为1的语言模型），这使得它能够自然地**窥视（peek at）**、**分区（partition）**、**文本搜索（grep through）以及对上下文发起递归的子查询（sub-queries）**。图3展示了带有 REPL 环境的 RLM 是如何产生最终答案的一个例子。

（图3描述） 图3标题为“带有 REPL 环境的 RLM”。

- 左侧是 RLM 的外部视图，接收“上下文”和“查询”，输出“响应”。

- 右侧展示了其内部工作流程，核心是一个“REPL Python 笔记本”。

- “根语言模型（深度=0）”在这个笔记本环境中工作。它接收到用户的初始查询（例如，“我正在尝试与包含上下文的 REPL 环境交互”）。

- 根模型不直接看到整个上下文，而是输出代码块（例如，`execute code`）来操作它。

- 代码在一个输入单元格（In）中执行，REPL 环境返回输出（Out），例如“最佳单卡匹配：正义盟军 灾亡兽”。

- 根模型可以根据需要发起一个“递归语言模型调用（深度=1）”，就像调用一个函数 `Illim.query()`。

- 经过多次交互后，当根模型找到答案时，它会输出一个带有 `FINAL` 标签的最终响应（例如，`FINAL(The an...)`）。

当根语言模型确信它有了答案时，它可以直接以 `FINAL(answer)` 的形式输出答案，或者它可以利用其 REPL 环境中的变量来构建一个答案，并以 `FINAL_VAR(final_ans_var)` 的形式返回该答案所在的字符串。

【深度解读】

为了彻底理解 RLM 的工作原理，让我们引入一个生动的比喻：**一位不进入案发现场的总探长**。

- **用户查询 (query)**：这是委托给总探长的案件目标，例如“找出凶手是谁？”
- **庞大的上下文 (context)**：这是案件的所有证据材料，堆满了一个巨大的证物仓库。它被数字化后，以一个名为 `context` 的变量形式存在。
- **根语言模型 (root LM)**：这就是我们的总探长。他非常聪明，但他的原则是**绝不亲自进入**那个杂乱的证物仓库，以免被海量无关信息淹没。
- **REPL 环境**：这是总探长的指挥室。指挥室里有一台功能强大的电脑，可以访问整个证物仓库的数字化档案。
- **递归语言模型调用 (recursive LM call)**：这是总探长手下的助理探员。他们可以被派去执行具体的调查任务。

现在，我们来看看总探长是如何破案的：

1. **接收任务**：总探长（根 LM）在他的指挥室（REPL 环境）里，只看到了案件目标（查询），他知道所有证据都在名为 `context` 的数字档案里。
2. **初步侦查（窥视）**：他不会去读完所有档案。他会在电脑上输入一条指令，比如 `print(context[:2000])`。这相当于说：“把证物档案的前5页给我看看，我需要了解一下档案的格式和大致内容。”
3. **缩小范围（文本搜索）**：根据初步的线索，他怀疑凶器是一把刀。于是他下达指令 `grep("knife", context)`，这相当于说：“在所有档案中搜索关键词‘刀’，把相关的段落都列出来。”
4. **分派任务（分区与子查询）**：他发现有10份关于“刀”的目击者证词非常可疑。他不会自己一份一份去看，而是把这10份证词分派给10个助理探员（发起10个并行的递归 LM 调用），并给他们下达一个明确的指令：“阅读你手上的这份证词，判断证人是否有不在场证明，并总结其核心内容。”
5. **汇总分析**：助理探员们完成任务后，将他们的报告（子响应）提交给总探长。总探长在指挥室里阅读这些高度精炼的报告，而不是原始的、冗长的证词。
6. **得出结论**：基于所有汇总的信息，总探长最终锁定了凶手，并输出最终的破案报告 `FINAL("凶手是张三")`。

这个过程完美地诠释了 RLM 的强大之处。总探长（根 LM）的“思维空间”（上下文窗口）始终保持清爽，只处理最关键的信息和指令，从而避免了“上下文退化”。

3.2. RLM 架构带来的核心优势

【原文翻译】

这个设置在实践中带来了几个显而易见的好处：

1. 根语言模型的上下文窗口很少被堵塞——因为它从不直接看到整个上下文，其输入上下文是缓慢增长的。

2. 根语言模型具有灵活性，可以查看上下文的子集，或者简单地对上下文的块进行递归处理。例如，如果查询是要求寻找一个“大海捞针”式的事实或一个多跳（multi-hop）事实，根语言模型可以使用正则表达式查询来大致缩小上下文范围，然后对这个范围内的上下文发起递归的语言模型调用。这对于任意长的上下文输入尤其有用，因为在这种情况下，即时为一个检索器建立索引的代价是昂贵的！

3. 理论上，上下文可以是任何能够被加载到内存中的模态（modality）。根语言模型拥有完全的控制权来查看和转换这些数据，以及向一个递归的语言模型提出子查询。

【深度解读】

基于我们“总探长”的比喻，这三个好处就变得非常容易理解了。

1. **总探长的大脑永远清晰**：总探长的办公桌上永远不会堆满成千上万份原始文件。他只看他自己发出的指令和下属提交的简洁报告。他的工作记忆（上下文窗口）负担很小，因此可以始终保持敏锐的判断力和宏观的洞察力。

2. **调查手段灵活高效**：总探长拥有多种调查工具。对于简单的任务，比如在文件中找一个名字，他可以用快速的“文本搜索”（正则表达式）自己搞定。对于复杂的任务，比如分析一份长篇的法医报告，他可以把这个任务“外包”给专业的助理探员（递归调用）。这种灵活性意味着他总能用最高效的方式来处理问题。特别是对于那些信息之间关联复杂、需要将多个线索串联起来的“多跳”问题，这种“先缩小范围，再深度分析”的策略远比漫无目的地阅读所有文件要高效得多。

3. **能处理任何形式的证据**：只要证据能被数字化并存入电脑（加载到内存中），总探长就能处理它。无论是文本文档、图片、音频还是视频（不同的“模态”），他都可以通过编写相应的代码来查看、处理，或者让专门的助理探员（能够处理特定模态的递归 LM）去分析。这使得 RLM 框架具有极强的通用性，未来可以应用于远超文本处理的领域。

总而言之，RLM 架构通过引入一个充当“指挥官”的根模型和一个作为“战场”的 REPL 环境，成功地将语言模型的“原始智力”转化为一种**有组织的、可扩展的、高效的解决问题的能力**。

第四部分：理论升华与设计哲学

4.1. 与测试时推理能力扩展的关系

【原文翻译】

与测试时推理能力扩展的关系。 我们对这种看待语言模型的视角感到特别兴奋，因为它为扩展测试时计算（test-time compute）提供了另一个维度。语言模型选择如何与其上下文进行交互和递归的轨迹是完全可以学习的，并且可以像目前为前沿模型训练推理能力那样，通过强化学习（RL-ified）来进行优化。有趣的是，它不直接要求训练能够处理巨大上下文长度的模型，因为**没有任何一次单独的语言模型调用需要处理巨大的上下文**。

【深度解读】

这段话揭示了 RLM 更深远的意义，它为提升 AI 能力提供了一条与众不同的新路径。在 AI 领域，提升模型性能通常有两条路：

1. **训练时提升**：用更多的数据、更大的模型、更好的算法来训练 AI，让它变得更“聪明”。这就像让一个学生在考试前花更多时间学习，掌握更多知识。这是目前的主流方法，但成本极高。

2. **测试时提升（推理时提升）**：不改变 AI 本身，而是在它解决问题（即“测试”或“推理”）时，给它更多的时间和资源去“思考”。这就像一个学生知识水平不变，但在考试时，允许他打草稿、多花时间检查、甚至使用计算器。

RLM 正是第二条路的杰出代表。它通过让模型在解决问题时执行一系列的计算步骤（交互、分解、递归），极大地增强了它的表现，而这一切都**不需要重新训练模型**。

更具革命性的一点是，模型“如何分解问题”、“何时进行递归”这些**策略本身是可以学习的**。我们可以通过强化学习（一种通过奖励和惩罚来学习最优行为的方法）来训练根模型，让它成为一个越来越出色的“总探长”。比如，如果它采取的一个分解策略很快地解决了问题，就给它一个“奖励”，反之则给一个“惩罚”。久而久之，它会自动学会最高效、最聪明的任务分解和递归策略。

这意味着，我们未来可能不需要费尽心机去造一个能一次性处理1亿词元上下文的“超级大脑”，而只需要训练一个普通的“大脑”如何高效地利用工具去管理和分析这1亿词元的信息。这是一种“四两拨千斤”的智慧，可能从根本上改变 AI 能力的扩展方式。

4.2. REPL 环境的强大之处与设计选择

【原文翻译】

带有 REPL 环境的 RLM 是强大的。 我们强调，环境的选择是灵活的，不一定非得是 REPL 或代码环境，但我们认为这是一个很好的选择。递归语言模型的两个关键设计选择是：1) 将提示视为一个 Python 变量，它可以在任意的 REPL 流程中以编程方式进行处理。这使得大语言模型能够在测试时自主决定要从长上下文中窥视哪些内容，并扩展它想采取的任何决策（例如，自适应地设计自己的分块和递归方案）；2) 允许该 REPL 环境回调大语言模型（或一个更小的 LLM），这得益于选择（1）所带来的分解能力和多功能性。

我们对 CodeAct 的设计感到兴奋，并推断向该系统添加递归模型调用可能会带来显著更强的能力——毕竟，语言模型函数调用功能非常强大。然而，我们认为 RLM 从根本上与先前工作的视角不同：在这里，**上下文是一个需要被模型理解的对象**，而代码执行和递归语言模型调用是高效理解这个上下文的**手段**。

最后，在我们的实验中，我们只考虑了1的递归深度——即根语言模型只能调用语言模型，而不能调用其他的 RLM。将 REPL 环境设置为调用 RLM 而非 LM 是一个相对容易的改动，但我们觉得对于大多数现代的“长上下文”基准测试来说，1的递归深度足以处理大部分问题。然而，对于未来的工作和对 RLM 的深入研究，启用更大的递归深度自然会带来更强大和更有趣的系统。

【深度解读】

为什么选择一个编程环境（Python REPL）作为 AI 的“草稿本”？这是一个至关重要的设计决策，其背后有深刻的考量。

首先，**代码是精确且无歧义的**。语言模型本身的“思考”可能是模糊的、带有创造性的，有时甚至是不可靠的（比如产生“幻觉”）。但是，当它必须将自己的问题分解策略表达为一系列 Python 代码时，这种模糊性就被消除了。`print(context[0:1000])` 这条指令的含义是唯一的、可执行的、可验证的。通过强迫模型用代码来“思考”，RLM 框架为模型的推理过程套上了一个**逻辑的“缰绳”**，确保了其行为的结构性和可靠性。

其次，**编程赋予了模型前所未有的自主性**。在传统的“代理”（Agent）框架中，通常是人类开发者为 AI 设计好一套固定的行为模式（例如，ReAct 框架中的“思考-行动-观察”循环）。而 RLM 的哲学则更为激进：它不给模型预设任何行为模式。它只是提供了一个功能强大的工具箱（REPL 环境）和使用工具的能力（执行代码和递归调用），然后让模型**自己决定**如何使用这些工具。模型可以自己发明全新的、人类开发者可能从未想过的策略来分解和分析上下文。这是一种从“授人以鱼”到“授人以渔”的飞跃，是通往更通用、更自适应智能的关键一步。

最后，作者点出了 RLM 与其他工作的根本区别。对于很多“代码智能体”来说，代码是**目标**（例如，写一个程序）。而对于 RLM 来说，代码和递归调用都只是**手段**，其真正的**目标**是**深刻理解给定的上下文**。这种“以理解为核心”的定位，使得 RLM 的应用场景远比代码生成更为广阔。

4.3. 形式化定义

【原文翻译】

形式化定义 (点击展开) 考虑一个通用设置，其中一个语言模型 M 接收一个查询 q 和一些相关的、可能很长的上下文 C=[c1,c2,…,cm]。标准方法是将 M(q,C) 视为一个黑盒函数调用，它接收一个查询和上下文，并返回某个字符串 `str` 输出。我们保留这种视角，但在模型之上定义了一个薄封装层，以提供一个具有相同输入和输出空间但更具**表达力**和**可解释性**的函数调用 RLMM(q,C)。

形式上，一个在环境 E 上运行的递归语言模型 RLMM(q,C) 同样接收一个查询 q 和一些相关的、可能很长的上下文 C=[c1,c2,…,cm]，并返回某个字符串 `str` 输出。主要区别在于，我们为模型提供了一个工具调用 RLMM(q^,C^)，它使用一个新的查询 q^ 和一个转换后的上下文版本 C^ 来生成一个隔离的子 RLM 实例，该实例拥有自己隔离的环境 E^；最终，这个递归被调用者的最终输出会被反馈回原始调用者的环境中。

环境 E 抽象地决定了如何提示、查询和处理语言模型 M 以提供最终输出的控制流。在本文中，我们特别探讨了使用一个 Python REPL 环境，该环境将输入上下文 C 存储为内存中的一个变量。这种特定的环境选择使得语言模型能够**窥视**、**分区**、**转换**和**映射**输入上下文，并使用递归的语言模型来回答关于这个上下文的子查询。与先前那些僵硬地定义这些工作流模式的代理方法不同，RLM 将这些决策完全交给了语言模型。

最后，我们注意到，环境 E 的特定选择是灵活的，并且是基础模型调用的一个泛化：最简单的可能环境 E0 会用输入的查询和上下文 q,C 来查询模型 M，并将模型的输出作为最终答案返回。

【深度解读】

对于高三学生来说，这段形式化的定义可能看起来有些抽象，但我们可以把它拆解成熟悉的数学函数概念。

- **标准模型调用 M(q,C)**：你可以把它看作一个简单的函数，输入是问题 q 和资料 C，输出是一个答案。例如 f(x,y)=x+y。你给它两个数，它直接给你一个和。
- **递归语言模型调用 RLMM(q,C)**：这是一个更复杂的、“封装”过的函数。它的输入输出和 M(q,C) 一样，但内部机制完全不同。它内部包含了一个环境 E（我们的指挥室），并且最重要的，它拥有调用**自身**的能力。

这个核心能力就是 RLMM(q^,C^)。这里的 q^ 和 C^ 分别代表“新的、更小的子问题”和“经过处理的、更小的部分资料”。这就像我们在数学中定义阶乘函数： n!=n×(n−1)!

要计算 5!，函数会调用自己去计算 4!；要计算 4!，它又会调用自己去计算 3!，以此类推，直到达到最简单的基础情况（1!=1）。这就是**递归**。

RLM 正是把这种强大的递归思想应用到了语言模型的任务处理中。根模型（总探长）在解决一个大问题时，可以生成新的、更小的子问题，并调用一个全新的、隔离的 RLM 实例（助理探员）去解决它。这个助理探员在自己的工作空间（隔离的环境 E^）里完成任务，然后把结果返回。

这种形式化定义的美妙之处在于它的**通用性**和**优雅性**。它没有规定模型**必须**如何分解问题，只赋予了它**可以**分解问题的能力。如何最好地利用这种能力，则完全由模型 M 的智能来决定。这为 AI 的自主性和创造力留下了广阔的空间。

第五部分：实验验证——在实战中检验 RLM 的威力

5.1. 实验一：OOLONG 基准测试（挑战上下文退化）

【原文翻译】

一些早期（且非常激动人心的）结果！

我们一直在寻找能够反映自然长上下文任务的基准，例如，长时间、多轮次的 Claude Code 会话。我们主要希望凸显限制现代前沿模型的两个特性：1) 上下文退化现象，即模型性能随上下文长度增加而下降；2) 处理巨大上下文的系统级限制。

在实践中我们发现，许多长上下文基准提供的上下文其实并不那么长，并且最新一代（或两代）的模型已经能够解决它们。事实上，我们发现有些基准测试中，**模型甚至可以在没有上下文的情况下回答问题**！幸运的是，我们很快找到了两个现代前沿大语言模型都难以表现出色的基准。

激动人心的结果 #1 — 应对上下文退化。 **OOLONG** 基准是一个具有挑战性的新基准，它评估在上下文中对细粒度信息进行的长上下文推理任务。我们有幸得到该数据集的作者（匿名，但与我们无关联）应允，分享了数据集的一个子集供我们进行实验。

设置。 `trec_coarse` 这个数据子集包含6种不同类型的查询，要求对一个巨大的“问题”条目列表进行分布式的查询。例如，其中一个问题是这样的： `对于以下问题，仅考虑与用户ID 67144, 53321,... , 59594 相关联的实例子集。在与这些用户相关联的实例中，有多少数据点应被分类为'entity'标签？请以'Answer: number'的形式给出你的最终答案。` 这个查询后面跟着大约3000到6000行的条目，每个条目都关联了用户ID（不一定是唯一的）和实例，但这些实例**没有被明确地打上标签**（即模型必须自己推断标签才能回答问题）。

分数是根据模型正确回答的查询数量来计算的，但对于数值/计数问题，他们使用了一个连续的评分标准。这个基准对前沿模型和代理都极其困难，因为它们必须在单次查询中对数千条信息进行语义映射和关联，并且不能预先计算任何东西。我们评估了以下模型/代理：

- **GPT-5**：给定全部上下文和查询，让 GPT-5 提供答案。
- **GPT-5-mini**：给定全部上下文和查询，让 GPT-5-mini 提供答案。
- **RLM(GPT-5-mini)**：给定全部上下文和查询，让 RLM(GPT-5-mini) 提供答案。其中，GPT-5-mini（根语言模型）可以在其 REPL 环境中递归调用 GPT-5-mini。
- **RLM(GPT-5) 无子调用**：给定全部上下文和查询，让 RLM(GPT) 提供答案。其中，GPT-5（根语言模型）不能在其 REPL 环境中递归调用 GPT-5。这是一个为了验证 REPL 环境在没有递归情况下的效果的消融实验。
- **ReAct w/ GPT-5 + BM25**：我们将每一行分块成一个独立的“文档”，并让一个 ReAct 循环访问一个 BM25 检索器，每次搜索请求返回10行。

结果。 我们专注于上下文超过128k词元的查询（约100个查询），并同时追踪了基准测试的性能和每次查询的总 API 成本。在以下所有结果中（图4a, b），整个输入都完全在 GPT-5/GPT-5-mini 的上下文窗口内——也就是说，错误的预测绝不是由于截断或上下文窗口大小限制造成的。

【深度解读】

这个实验的设计非常巧妙，它旨在模拟一个非常真实且困难的场景：**处理大量未经整理的原始数据并进行分类统计**。

想象一下，你拿到了一份几千行的用户反馈表格，任务是统计其中提到“实体”（entity）这个概念的反馈数量，但表格里并没有“分类”这一列。你需要先阅读每一行反馈，理解其语义，判断它是否属于“实体”类别，然后再进行计数。这个任务的难点在于，它不仅仅是信息检索，更要求模型具备**大规模的在线推理和分类能力**。

实验设置中的几个对比组非常有说服力：

- **GPT-5 vs. GPT-5-mini**：这是“尖子生”和“普通生”的直接对决，用来建立一个性能基线。
- **RLM(GPT-5-mini) vs. GPT-5**：这是实验的核心看点，即一个装备了先进方法（RLM）的“普通生”能否挑战甚至超越一个赤手空拳的“尖子生”。
- **RLM(GPT-5) 无子调用**：这是一个重要的**消融实验**（ablation study）。在科学实验中，消融实验指的是去掉系统中的某个组件，来观察其对整体性能的影响。这里，他们去掉了“递归调用”这个关键组件，只保留了 REPL 环境。这可以帮助我们判断，性能的提升究竟是来自 REPL 环境本身，还是来自递归这个核心思想。
- **ReAct w/ GPT-5 + BM25**：这是一个经典的“检索增强”方案，代表了另一种解决长上下文问题的主流思路。通过与它对比，可以说明 RLM 在这类任务上是否比传统检索方法更优越。

这个实验的背景设置，为我们接下来要看到的惊人结果铺平了道路。

5.1.1. 在 13.2万 和 26.3万 词元上下文长度下的表现

【原文翻译】

（图4a描述） 图4a包含两个柱状图。 左图标题为“OOLONG trec_coarse w/ 132k Token Context - Score (%)”，展示了在13.2万词元上下文下的得分率。

- ReAct+GPT-5+BM25: 1.5%
- GPT-5-mini: 22.2%
- RLM(GPT-5) w/o sub-calls: 45.6%
- GPT-5: 26.1%
- RLM(GPT-5-mini): 56.0%

右图标题为“OOLONG trec_coarse w/ 132k Token Context - Avg. Cost ($)”，展示了平均每次查询的成本。

- ReAct+GPT-5+BM25: $0.195
- GPT-5-mini: $0.033

- RLM(GPT-5) w/o sub-calls: $0.234
- GPT-5: $0.221
- RLM(GPT-5-mini): $0.276

图4a. 我们报告了在 OOLONG 基准的 `trec_coarse` 数据集上，针对上下文长度为13.2万词元的查询，各种方法的总得分。我们与 GPT-5 的性能进行了比较。RLM(GPT-5-mini) 的表现比 GPT-5 高出超过34个百分点（约114%的提升），并且每次查询的成本几乎同样低廉（我们发现由于一些异常昂贵的查询，导致平均成本较高，但中位数成本更低）。

事实证明，RLM(GPT-5-mini) 的原始得分比 GPT-5 和 GPT-5-mini 高出超过33%，性能提升超过一倍，而每次查询的总模型 API 成本与 GPT-5 大致相当！在消融掉递归功能后，我们发现 RLM 的性能下降了约10%，这可能是因为许多问题需要模型回答关于数据的语义问题（例如，为每个问题打标签）。

【数据表格】

表1：在 OOLONG 基准测试上的性能与成本（13.2万词元上下文）

方法	得分 (%)	平均每次查询成本 ($)
ReAct w/ GPT-5 + BM25	1.5	0.195
GPT-5-mini	22.2	0.033
GPT-5	26.1	0.221
RLM(GPT-5) 无子调用	45.6	0.234
RLM(GPT-5-mini)	**56.0**	**0.276**

【原文翻译】

我们在图4b中看到，当我们将上下文大小加倍到约26.3万词元时，这些优势大致得以保持，尽管性能有所下降！

（图4b描述） 图4b包含两个柱状图。 左图标题为“OOLONG trec_coarse w/ 263k Token Context - Score (%)”，展示了在26.3万词元上下文下的得分率。

- ReAct+GPT-5+BM25: 1.5%
- GPT-5-mini: 10.4%
- RLM(GPT-5) w/o sub-calls: 34.1%
- GPT-5: 27.2%
- RLM(GPT-5-mini): 40.9%

右图标题为“OOLONG trec_coarse w/ 263k Token Context - Avg. Cost ($)”，展示了平均每次查询的成本。

- ReAct+GPT-5+BM25: $0.237
- GPT-5-mini: $0.061
- RLM(GPT-5) w/o sub-calls: $0.312
- GPT-5: $0.425
- RLM(GPT-5-mini): $0.525

图4b. 我们报告了在 OOLONG 基准的 `trec_coarse` 数据集上，针对上下文长度为26.3万词元的查询，各种方法的总得分，这几乎是 GPT-5/GPT-5-mini 的极限。RLM(GPT-5-mini) 的表现比 GPT-5 高出超过15个百分点（约49%的提升），并且平均每次查询的成本更低。

值得注意的是，GPT-5-mini 的性能下降了，而 GPT-5 则没有，这表明上下文退化对 GPT-5-mini 更为严重。我们还注意到，RLM 方法的性能下降主要发生在**计数**问题上，当上下文长度增加时，它会犯更多错误——而对于 GPT-5，它在13.2万词元的情况下就已经答错了大部分这类问题，这解释了为什么其性能大致保持不变。

【数据表格】

表2：在 OOLONG 基准测试上的性能与成本（26.3万词元上下文）

方法	得分 (%)	平均每次查询成本 ($)
ReAct w/ GPT-5 + BM25	1.5	0.237
GPT-5-mini	10.4	0.061
GPT-5	27.2	0.425
RLM(GPT-5) 无子调用	34.1	0.312
RLM(GPT-5-mini)	**40.9**	**0.525**

【深度解读】

这两组数据讲述了一个激动人心的“以弱胜强”的故事。

核心发现 1：方法 > 智力 在13.2万词元的测试中（表1），最强的 GPT-5 模型得分仅为 26.1%。然而，较弱的 GPT-5-mini 模型在配备了 RLM 框架后，得分飙升至 56.0%，**性能提升了114%**。这就像一个普通学生，通过掌握了一套先进的学习方法（如费曼学习法），在考试中战胜了裸考的天才。这雄辩地证明了，一个优秀的推理**策略**，其价值可能超过模型本身的**原始智能**。

核心发现 2：递归是关键 再看“RLM(GPT-5) 无子调用”这一组，它的得分是 45.6%。这个分数远高于标准的 GPT-5，说明仅仅提供一个 REPL 环境让模型可以“打草稿”，就已经能带来巨大提升。但是，它又明显低于拥有递归能力的 RLM(GPT-5-mini) 的 56.0%。这清晰地表明，**递归调用**——即“总探长”将任务分解给“助理探员”的能力——是实现最佳性能的**关键所在**。

核心发现 3：RLM 的抗压性 当上下文长度加倍到26.3万词元时（表2），压力急剧增大。标准的 GPT-5-mini 性能从 22.2% 暴跌至 10.4%，几乎崩溃，这正是“上下文退化”的典型表现。然而，RLM(GPT-5-mini) 虽然性能也有所下降，但依然保持在 40.9% 的高位，仍然**远超**更强大的 GPT-5（27.2%）。这说明 RLM 框架不仅能提升性能，更能显著增强模型在处理超长上下文时的**鲁棒性**和**稳定性**。

成本考量 虽然 RLM(GPT-5-mini) 的绝对成本略高于 GPT-5，但考虑到其性能实现了翻倍以上的提升，其**性价比**（性能/成本比）是无与伦比的。这在实际应用中具有巨大的商业价值。

5.2. 实验二：BrowseComp-Plus（挑战海量上下文）

【原文翻译】

很好！所以我们在解决目标（1）方面取得了巨大进展，即在 GPT-5 刚好能容纳26.3万词元上下文的情况下。但目标（2）呢？当我们可能有100万、1000万甚至1亿个词元在上下文中时该怎么办？我们还能像处理单次模型调用一样处理它吗？

激动人心的结果 #2 — 极度巨大的上下文 我的导师 Omar 是信息检索（IR）领域的世界级专家，所以我们自然也想探索当给定数千（甚至更多！）文档时，RLM 是否能正确地扩展。OOLONG 提供的是一个难以索引的巨大文本块，因此很难与检索方法进行比较，所以我们研究了像 DeepResearch 这样的基准，它们评估在文档集合上回答查询的能力。

在巨大的离线语料库上进行检索。 我们最初对 BrowseComp 感兴趣，它评估代理在多跳、网络搜索查询上的表现，代理必须在网上找到相关文档。后来我们找到了 BrowseComp-Plus 基准，它为原始基准中的所有查询预先下载了所有可能相关的文档，并提供了一个约10万份文档的列表（平均约5000词），查询的答案散布在这个列表中。对于测试 RLM 来说，这个基准非常完美，可以看看我们是否能把数量惊人的上下文直接扔进一个 `chat.completion(...)` RLM 调用中，而不是去构建一个代理！

设置。 我们探索了扩展上下文中#文档数量如何影响各种处理文本语料库的常用方法以及 RLM 的性能。BrowseComp-Plus 基准上的查询是多跳的，因为它们需要关联来自几个不同文档的信息才能回答查询。这意味着，即使你检索到了包含正确答案的文档，你也无法知道它是正确的，直到你找出其他的关联信息。例如，基准中的第984号查询是这样的：
我正在寻找一款集换式卡牌游戏中的一张特定卡牌。这张卡牌在2005年到2015年之间发行，并且在发行年份有多种稀有度。这张卡牌曾被一位日本玩家在赢得该集换式卡牌游戏世界冠军时使用的牌组中使用过。在背景故事中，这张卡牌被用作另一张在2013年到2018年间发行的卡牌的盔甲。这张卡牌也曾一度在不同赛事中被禁止使用，并且等级低于8级。这张卡牌是什么？

在我们的实验中，我们探索了每种模型/代理/RLM 在给定不同大小的抽样文档语料库时的性能——唯一的保证是答案可以在这个语料库中找到。在实践中，我们发现 GPT-5 在超过输入上下文窗口（272k词元）之前，大约可以容纳40个文档，我们在选择基线的常数时考虑了这一点。我们探索了以下模型/代理，与之前的实验类似：

- **GPT-5**：给定所有文档作为上下文和查询，让 GPT-5 提供答案。如果超过上下文限制，则不返回任何内容。
- **GPT-5 (截断)**：给定所有文档作为上下文和查询，让 GPT-5 提供答案。如果超过上下文限制，则按最近的词元截断（即随机的文档）。
- **GPT-5 + 查询前 BM25**：首先使用原始查询通过 BM25 检索出排名前40的文档。给定这前40个文档和查询，让 GPT-5 提供答案。
- **RLM(GPT-5)**：给定所有文档作为上下文和查询，让 RLM(GPT-5) 提供答案。其中，GPT-5（根语言模型）可以在其 REPL 环境中“递归地”调用 GPT-5-mini。
- **RLM(GPT-5) 无子调用**：给定全部上下文和查询，让 RLM(GPT-5) 提供答案。其中，GPT-5（根语言模型）不能在其 REPL 环境中递归调用 GPT-5。这是一个为了验证 REPL 环境在没有递归情况下的效果的消融实验。
- **ReAct w/ GPT-5 + BM25**：给定所有文档，从一个使用 GPT-5 的 ReAct 循环中查询答案，该循环可以访问一个 BM25 检索器，每次请求可以返回5个文档。

结果。 我们想强调的是，这些初步结果并非基于整个 BrowseComp-Plus 数据集，而只是一个小小的子集。我们在图5中报告了在给定10、50、100和1000个文档作为上下文时，在20个随机抽样的 BrowseComp-Plus 查询上的性能。

【深度解读】

这个实验将挑战推向了极致。如果说上一个实验是让 AI 精读一本厚书，那么这个实验就是把它扔进一个小型图书馆，并要求它根据散落在成百上千本书籍中的线索，去破解一个复杂的谜题。

那个关于集换式卡牌的查询就是一个完美的例子。它包含了时间、稀有度、玩家国籍、比赛成绩、背景故事、游戏规则等多个维度的限制条件，这些信息极不可能出现在同一份文档里。要回答这个问题，模型必须：

1. 在海量文档中找到所有可能相关的片段。
2. 将这些来自不同文档的片段信息进行拼接和交叉验证。
3. 基于所有整合后的信息，进行逻辑推理，最终得出唯一的答案。

这是一个典型的**“多跳推理”（multi-hop reasoning）**任务，它对 AI 的信息整合和逻辑推理能力提出了极高的要求。

实验中设置的几个基线也很有代表性：

- **GPT-5 (截断)**：代表了最简单粗暴的方法——如果信息太多，就随机扔掉一部分。这显然很容易丢失关键线索。
- **GPT-5 + 查询前 BM25**：代表了传统的“先检索，后阅读”方法。它的成败完全取决于第一步的检索是否精准。但对于这种复杂的多跳查询，仅凭原始查询很难一次性检索出所有相关的文档。
- **ReAct w/ GPT-5 + BM25**：代表了更先进的“边想边查”的代理方法。它可以通过多次检索来逐步收集信息，理论上更强大。

RLM 将在这里与这些主流方法进行正面交锋，以证明它在处理真正海量、分散的信息时，是否具备无可替代的优势。

5.2.1. 在不同文档数量下的性能与成本表现

【原文翻译】

（图5描述） 图5包含两个折线图。 左图标题为“Score on BrowseComp-Plus vs # Context Docs”，展示了正确回答率随上下文文档数量的变化。

- GPT-5 和 GPT-5 (Truncated) 的线在10个文档时约为100%，但在50个文档时骤降至0%。
- GPT-5 + Pre-query BM25 的线在10个文档时为100%，然后逐渐下降，在1000个文档时约为20%。
- ReAct w/ GPT-5 + BM25 的线在10个文档时为100%，然后逐渐下降，在1000个文档时约为75%。
- RLM(GPT-5) w/o sub-calls 的线在10个文档时为100%，然后略微下降，在1000个文档时保持在90%。
- RLM(GPT-5) 的线在所有文档数量下都保持在100%。

右图标题为“Average API Cost ($) per Query vs # Context Docs”，展示了平均API成本随上下文文档数量的变化。

- 所有方法的成本都随文档数量增加而上升。
- RLM(GPT-5) 和 RLM(GPT-5) w/o sub-calls 的成本增长斜率较为平缓，在1000个文档时，RLM(GPT-5) 的成本约为$125。
- ReAct w/ GPT-5 + BM25 的成本增长最快，在1000个文档时成本超过$200。

图5. 我们绘制了在给定不断增加的上下文文档数量时，各种方法在 BrowseComp-Plus 的20个随机查询上的性能和单次回答的 API 成本。只有迭代式的方法（RLM, ReAct）在100个文档以上时还能保持合理的性能。

这里有几点值得注意——值得注意的是，RLM(GPT-5) 是唯一能够在1000个文档规模下达到并保持完美性能的模型/代理，而其消融版本（无递归）也能类似地达到90%。基础的 GPT-5 模型方法，无论如何调整，都显示出随着文档数量增加而性能急剧下降的明显迹象。与 OOLONG 不同，当给定足够小的上下文窗口（10个文档）时，所有方法都能解决这个任务，这使得这个问题变成了一个寻找正确信息的问题，而不是处理复杂查询的问题。此外，RLM(GPT-5) 的每次查询成本随上下文长度的增长也相当合理！

这些实验尤其令人兴奋，因为在没有任何额外的微调或模型架构改变的情况下，我们可以在现实的基准上，合理地处理巨大（超过1000万词元）的上下文语料库，而无需使用检索器。

【数据表格】

表3：在 BrowseComp-Plus 上的性能表现 vs. 上下文文档数量

方法	10个文档	50个文档	100个文档	1000个文档
GPT-5	~100%	0%	0%	0%
GPT-5 (截断)	~100%	0%	0%	0%
GPT-5 + Pre-query BM25	100%	~80%	~60%	~20%
ReAct w/ GPT-5 + BM25	100%	~95%	~90%	~75%
RLM(GPT-5) 无子调用	100%	100%	~95%	90%
RLM(GPT-5)	**100%**	**100%**	**100%**	**100%**

表4：在 BrowseComp-Plus 上的平均 API 成本 vs. 上下文文档数量

方法	10个文档 ($)	50个文档 ($)	100个文档 ($)	1000个文档 ($)
GPT-5 (截断)	~10	~20	~30	~40
GPT-5 + Pre-query BM25	~20	~20	~20	~20
ReAct w/ GPT-5 + BM25	~20	~50	~100	>200
RLM(GPT-5) 无子调用	~15	~30	~50	~100
RLM(GPT-5)	**~20**	**~40**	**~60**	**~125**

【深度解读】

如果说上一个实验的结果是“惊人”，那么这个实验的结果可以说是**“颠覆性”**的。

压倒性的性能优势 左边的性能图（表3）是这篇论文的“高光时刻”。当文档数量增加到50个时，标准 GPT-5 方法由于超出上下文长度限制，性能直接崩溃为0。传统的“先检索后阅读”（Pre-query BM25）方法也随着信息量的增加而性能迅速下降，在1000个文档时只剩下20%的准确率，说明它无法有效应对信息分散的多跳问题。即使是更强大的 ReAct 代理，性能也下降到了75%。

然而，**RLM(GPT-5) 的表现如同一条完美的直线，在所有情况下都保持了100%的正确率**。这直观地证明了 RLM 具备处理海量、分散信息源的超凡能力。它不像其他方法那样需要“猜测”哪些信息是重要的，而是有能力系统性地、有条不紊地探索整个信息空间，确保不会遗漏任何关键线索。

合理的成本扩展 右边的成本图（表4）同样重要。它回答了一个关键的现实问题：“这种强大的能力是否代价高昂到无法使用？” 答案是否定的。虽然 RLM 的成本随着文档数量的增加而上升，但其增长曲线是相对平缓的。相比之下，表现次优的 ReAct 代理的成本增长得更快。这说明 RLM 的策略是高效的，它能够智能地管理其计算资源，使其成本增长与问题复杂度的增长保持在一个合理的比例。

RLM 成功的深层原因 这些实验结果揭示了 RLM 成功的根本原因。它不是一个简单的“黑盒”，而是一个**透明且可解释的推理框架**。在接下来的部分，我们将深入 RLM 的“内心”，看看它到底是如何自主地发明出那些高效的问题解决策略的。这些策略，是它能够在海量信息中保持冷静和高效的秘密武器。

第六部分：深入 RLM 的“思维”——涌现出的问题解决策略

6.1. RLM 的行为轨迹分析

【原文翻译】

RLM 在做什么？一些有趣的案例...

RLM 框架的一个强大优势是能够大致解释它在做什么以及它是如何得出最终答案的。我们编写了一个简单的可视化工具来窥探 RLM 的轨迹，这给了我们几个有趣的例子来分享 RLM 正在做什么！

RLM 已经涌现出的策略。 在 RLM 层面，我们可以完全解释语言模型是如何选择与上下文进行交互的。请注意，在每种情况下，根语言模型开始时只拥有查询和一个指示，即上下文存在于一个它可以与之交互的 REPL 环境的变量中。

【深度解读】

这部分是论文中最引人入胜的内容之一。它不再仅仅展示“RLM 很厉害”这个结果，而是试图回答“RLM **为什么**这么厉害？”。作者通过一个可视化工具，让我们能够像观察一位侦探破案一样，观察 RLM 的“思考”过程。

最关键的概念是**“涌现”（emerge）。**这意味着 RLM 所使用的那些聪明的策略——比如先粗略看一下文件，再搜索关键词——并**不是人类程序员预先编写好的指令**。相反，这些策略是语言模型在面对问题和可用的工具（REPL 环境）时，**自主“发现”或“发明”出来的最**

高效的解题方法。

这背后蕴含着一个深刻的观点：与其由人类为 AI 设计复杂的推理流程，不如为 AI 创建一个足够丰富和强大的环境，让 AI 的智能本身去探索和学习最优的策略。这标志着我们与 AI 协作方式的根本性转变，从一个“指令下达者”变成一个“环境创造者”和“能力赋予者”。接下来，我们将具体看看 RLM 都“发明”了哪些像人类专家一样的策略。

6.2. 具体的涌现策略

【原文翻译】

窥视（Peeking）。 在 RLM 循环开始时，根语言模型完全看不到上下文——它只知道它的大小。就像一个程序员在分析一个数据集时会先看几条数据一样，语言模型可以窥视其上下文以观察其结构。在下面 OOLONG 的例子中，外部语言模型获取了上下文的前2000个字符。 （图片展示了模型执行 `context[:2000]` 代码，输出了数据集的开头部分，显示了数据的格式，如日期、用户ID和问题实例。）

文本搜索（Grepping）。 为了减少其上下文的搜索空间，RLM with REPL 不使用语义检索工具，而是可以寻找关键词或正则表达式模式来缩小感兴趣的行的范围。在下面的例子中，RLM 寻找包含问题和 ID 的行。 （图片展示了模型编写了一个 Python 函数 `extract_question`，该函数使用正则表达式来从复杂的行格式中稳健地提取出问题文本。）

分区 + 映射（Partition + Map）。 在许多情况下，由于模型正在寻找的内容存在语义上的等价性，模型无法直接进行文本搜索或检索信息。RLM 会执行的一种常见模式是将上下文分成更小的块，并运行几个递归的语言模型调用来提取答案或执行这种语义映射。在下面 OOLONG 的例子中，根语言模型要求递归的语言模型为每个问题打上标签，并使用这些标签来回答原始的查询。 （图片展示了模型定义了一个 `classify_questions_batch` 辅助函数，该函数将问题分批（每批50个），然后调用递归的语言模型 `llm.query()` 对每批问题进行分类，最后解析返回的标签。）

摘要（Summarization）。 RLM 是对常用于管理语言模型上下文窗口的基于摘要的策略的自然泛化。RLM 通常会对上下文的子集进行信息摘要，供外部语言模型做出决策。 （图片展示了模型为了进行最终的交叉核对，发起了一个子查询，要求对开篇章节进行摘要，以确定叙述者的身份和搬到东部的原因。）

长输入，长输出（Long-input, long-output）。 语言模型失败的一个特别有趣且昂贵的案例是需要长输出生成的任务。例如，你可能会给 ChatGPT 你的一系列论文，并要求它为所有论文生成 BibTeX。与巨大的乘法问题类似，有些人可能会认为不应该期望模型完美无瑕地解决这些程序化任务——在这些情况下，带有 REPL 环境的 RLM 应该能够一次性解决这些任务！一个例子是 **LoCoDiff** 基准，其中语言模型被要求从头到尾跟踪一个长的 `git diff` 历史，并根据初始文件输出这个历史的结果。对于超过75k词元的历史，GPT-5 甚至无法解决10%的历史记录！ （图片展示了模型没有逐行应用 `diff`，而是选择编写一个复杂的 Python 程序，该程序可以解析 `commit` 块，提取特定文件的 `diff`，并以编程方式将这些 `diff` 应用到原始文件行上，从而一次性完成任务。）

【深度解读】

这些涌现出的策略，完美地印证了我们之前使用的“总探长”比喻。让我们逐一对应：

1. **窥视 (Peeking)**：总探长拿到一箱新证据，他不会一头扎进去，而是会先打开箱子，看一眼最上面的几份文件，了解这箱证据大致是什么类型（是信件、照片还是报告？），这就是“窥视”。这是一种建立初步认知的高效方式。

2. **文本搜索 (Grepping)**：总探长根据线索，需要找到所有提到“红色轿车”的证词。他不会去重读所有证词，而是让助手（或使用电脑的搜索功能）在所有文档中进行关键词搜索。RLM 通过编写正则表达式来实现这种精确、高效的信息定位。

3. **分区 + 映射 (Partition + Map)**：这是 RLM 最核心、最强大的策略，也是“总探长”模式的精髓。面对1000份需要进行语义判断（例如，判断证词的可信度）的文件，总探长不会自己一份份去看。他会将文件分成10组，每组100份，分派给10个助理探员（**分区**），并给他们一个统一的任务：“评估你手中这100份文件的可信度，并给出‘高/中/低’的评级”。最后，他将所有助理探员的评级结果收集起来，形成一个全局的视图（**映射**）。RLM 正是通过将上下文分块，并对每个分块发起并行的递归调用来实现这一过程。

4. **摘要 (Summarization)**：助理探员在分析完一份长达50页的法医报告后，不会把报告原文直接交给总探长，而是会写一份一页纸的摘要，提炼出最重要的结论。RLM 也同样会利用递归调用来生成中间摘要，以减轻根模型的认知负担。

5. **程序化解决 (Long-input, long-output)**：这是一个非常高级的智能体现。在 LoCoDiff 任务中，模型面对的是一个冗长、重复且有严格逻辑规则的任务（应用一系列 git diff）。它没有选择“硬着头皮”一步步去模拟这个过程，因为它知道自己可能会出错。相反，它选择了一个更聪明、更根本的解决方法：**编写一个能自动完成这个任务的计算机程序**。这就像一个物理学家，在需要进行大量重复计算时，他不会用笔去算，而是会写一个程序来解决它。这表明 RLM 不仅能执行任务，还能**理解任务的内在逻辑，并选择最优的工具（编程）来一劳永逸地解决它**。

这些策略的涌现，强有力地证明了 RLM 框架的巨大潜力。它不仅仅是一个处理长上下文的工具，更是一个能够激发和承载语言模型**自主规划和复杂问题解决能力**的强大平台。

第七部分：讨论、局限与未来展望

7.1. 局限性与相关工作

【原文翻译】

更多模式...？ 我们预计，随着 1) 模型变得更好和 2) 模型被训练/微调以这种方式工作，未来会涌现出更多的模式。这项工作一个尚未充分探索的领域是，语言模型在选择如何与 REPL 环境交互方面能达到多高的**效率**，我们相信所有这些目标（例如，速度、效率、性能等）都可以作为标量奖励进行优化。

局限性。 我们没有为 RLM 的实现进行速度优化，这意味着每个递归的语言模型调用都是阻塞式的，并且没有利用任何类型的“前缀缓存”！根据 RLM 的根语言模型所采用的分区策略，**缺乏异步性**可能导致每次查询的时间从几秒钟到几分钟不等。此外，虽然我们可以通

过增加最大迭代次数来控制 RLM 的长度/“思考时间”，但我们目前对于控制每次调用的总 API 成本或总运行时间没有强有力的保证。对于那些在系统社区的人来说（*咳咳*，特别是 GPU MODE 社区），这是个好消息！这里有太多唾手可得的优化成果，而要让 RLM 规模化工作，需要我们重新思考推理引擎的设计。

相关工作 用于长输入上下文管理的支架。 RLM 将上下文管理的选择权交给了语言模型/REPL 环境，但大多数先前的工作并非如此。MemGPT 类似地将选择权交给了模型，但它建立在一个单一的上下文中，语言模型最终会调用这个上下文来返回响应。MemWalker 强加了一种树状结构来规定语言模型总结上下文的顺序。LADDER 从问题分解的角度来分解上下文，但这无法泛化到巨大的上下文。

其他（相当不同的）递归提议。 有大量工作在深度学习的背景下调用分叉线程或进行递归，但没有一个具备通用分解所需的结构。THREAD 修改了模型调用的输出生成过程，以生成写入输出的子线程。微型递归模型（Tiny Recursive Model, TRM） 是一个很酷的想法，用于在其潜空间中迭代地改进一个（不一定是语言）模型的答案。递归 LLM 提示是一个早期的实验，它将提示视为一个在你查询模型时会演变的状态。递归自聚合（Recursive Self-Aggregation, RSA） 是最近的一项工作，它结合了对一组候选响应进行测试时推理采样的方法。

【深度解读】

任何一项开创性的研究都会有其局限性，坦诚地讨论这些局限性不仅是科学严谨性的体现，也为未来的研究指明了方向。

局限性：当前的 RLM 是“原型车”，而非“量产车” 作者指出的核心局限在于**效率**。当前的实现是“阻塞式”的，这意味着“总探长”在派给一个助理探员任务后，必须原地等待他完成，才能派下一个任务。这显然效率低下。一个更优化的系统应该是“异步”的，即总探长可以同时给10个助理探员派发任务，让他们并行工作，这会大大缩短破案时间。此外，成本和时间的不可控性也是工程应用上需要解决的问题。

然而，作者巧妙地将此视为一个**机遇**。他们指出，这辆“原型车”虽然速度不快，但已经证明了其设计的优越性（强大的性能）。接下来，需要系统工程师们（特别是那些专注于 GPU 优化的专家）来优化它的“引擎”（推理系统），将其打造成一辆既强大又快速的“量产跑车”。

相关工作：在巨人的肩膀上，但看得更远 通过与相关工作的对比，作者清晰地标定了 RLM 的独特定位。

- 与 **MemGPT** 等上下文管理工具相比：大多数工具都试图为 AI 设计一套固定的信息管理规则（比如 MemWalker 的树状结构）。而 RLM 的哲学是，**不应该由人来教 AI 如何管理信息，而应该让 AI 自己学会管理信息**。

- 与其他的“递归”思想相比：这个领域的“递归”一词含义广泛。
    - **RSA（递归自聚合）**：它的递归是关于**答案**的。它会生成多个不同的答案，然后让模型“回头看”这些答案，并尝试将它们融合成一个更好的答案。这是一个**答案改进**的过程。
    - **TRM（微型递归模型）**：它的递归是关于**自身状态**的。一个很小的模型会反复迭代自己的“思考”，逐步修正自己的答案。这是一个**自我完善**的过程。
    - **RLM**：它的递归是关于**问题和上下文**的。它将一个大问题分解成多个小问题，并独立解决它们。这是一个**问题分解**的过程。

通过这些对比，我们可以清晰地看到，虽然大家都用了“递归”这个词，但 RLM 的思想——**通过环境交互和递归调用来实现由模型自主驱动的上下文分解**——是独一无二且极具开创性的。

7.2. 未来思考：RLM 作为一种新的扩展范式

【原文翻译】

我们现在及未来的思考。

语言模型中的长上下文能力曾经是一个模型架构问题（想想 ALiBi, YaRN 等）。然后社区声称这是一个系统问题，因为“注意力是二次方的”，但后来发现实际上我们的 MoE 层才是瓶颈。现在它或多或少成了两者的结合，再加上越来越长的上下文不太符合我们语言模型训练分布的事实。

我们必须解决上下文退化问题吗？ 对于“上下文退化”有几种合理的解释；对我来说，最可信的是，由于长序列在自然语言中出现频率较低且熵更高，导致它们超出了模型训练分布的范围。RLM 的目标一直是提出一个框架，可以在**无需直接解决这个问题**的情况下发出语言模型调用——虽然这个想法最初只是一个框架，但我们对它在现代语言模型上取得的强劲结果感到非常惊讶，并乐观地认为它们将继续良好地扩展。

RLM 不是代理，也不仅仅是摘要。 在一个系统中进行多次语言模型调用的想法并不新鲜——从广义上讲，这是大多数代理支架所做的事情。我们在野外看到的最接近的想法是 ROMA 代理，它分解一个问题并运行多个子代理来解决每个问题。另一个常见的例子是像 Cursor 和 Claude Code 这样的代码助手，它们会在上下文历史变长时对其进行摘要或修剪。这些方法通常将多次语言模型调用视为**从任务或问题的角度**进行的分解。我们保留这样的观点，即语言模型调用可以**按上下文进行分解**，并且分解的选择应该**纯粹是语言模型的选择**。

固定格式对于扩展定律的价值。 我们作为一个领域，从 CoT、ReAct、指令微调、推理模型等思想中学到，以可预测或固定的格式向模型呈现数据对于提高性能非常重要。基本思想是，如果我们可以将训练数据的结构简化为模型期望的格式，我们就可以用合理数量的数据极大地提高模型的性能。我们很兴奋地期待如何将这些思想应用于提高 RLM 的性能，作为另一个扩展的维度。

RLM 随着语言模型的改进而改进。 最后，RLM 调用的性能、速度和成本与基础模型能力的改进直接相关。如果明天，最好的前沿语言模型可以合理地处理1000万词元的上下文，那么一个 RLM 就可以合理地处理1亿词元的上下文（也许成本还能减半）。

作为结束语，**RLM 与现代代理是一个根本上不同的赌注。** 代理是基于人类/专家的直觉来设计的，关于如何将一个问题分解成语言模型可以消化的形式。RLM 的设计原则是，从根本上说，**语言模型应该自己决定如何将一个问题分解成语言模型可以消化的形式。** 我个人不知道最终哪种方法会成功，但我很兴奋地期待这个想法会走向何方！

-az

【深度解读】

这最后一部分是整篇论文思想的升华，它提出了一个关于人工智能未来的深刻哲学观点。

回避问题，而非解决问题 作者提出了一个非常聪明的观点：也许我们根本不需要从正面攻克“上下文退化”这个难题。上下文退化可能是因为超长、高复杂度的文本在模型的训练数据中极为罕见，导致模型天生就不擅长处理它们。与其耗费巨资去训练一个能处理这种“超纲”题目的模型，不如设计一个框架（RLM），让模型可以**自动将任何“超纲”的难题分解成一系列它能够处理的“大纲内”的简单题目**。这是一种工程上的智慧，即通过改变工作流程来绕过底层能力的限制。

RLM 的核心赌注：相信 AI，而不是人类的直觉 这是全文最核心的论点。目前主流的“代理”（Agent）系统，其背后都有一套由人类专家设计的“行为逻辑”，比如 ReAct 的“思考-行动-观察”循环。这些框架本质上是**人类在教 AI 如何思考**。

而 RLM 提出了一个截然相反的、更大胆的设想。它认为，随着语言模型本身变得越来越智能，**我们应该停止为它设计固定的思维框架，而是应该相信它能够自己找到最优的思考和解决问题的路径**。RLM 的角色，就是为 AI 提供一个足够强大的“舞台”（REPL 环境）和足够灵活的“能力”（递归调用），然后让 AI 在这个舞台上自由“表演”。

这是一个根本性的赌注：未来的进步，究竟是来自于更精巧的、由人类设计的“代理框架”，还是来自于更强大的、能让 AI 自主学习和规划的“赋能环境”？

一个良性循环的未来 最后，RLM 的发展与基础模型的发展形成了一个完美的正反馈循环。基础模型越强大，RLM 的根模型（总探长）就会变得越聪明，其分解和规划任务的能力就越强。而 RLM 框架的存在，又使得基础模型的能力能够被应用到前所未有的、更大规模和更复杂的任务上。

总而言之，递归语言模型不仅仅是一项技术创新，它更像是一种新的**人工智能哲学**。它倡导一种更信任、更赋能的与 AI 的协作关系，为我们指明了一条通往更通用、更自主的人工智能的激动人心的道路。

7.3. 致谢与引用

【原文翻译】

致谢 我们感谢我们出色的 MIT OASYS 实验室同仁 Noah Ziems、Jacob Li 和 Diane Tchuindjo，感谢他们在项目方向和解决难题方面的长时间讨论。我们感谢来自 MIT DSG 组的 Tim Kraska 教授、James Moore、Jason Mohoney、Amadou Ngom 和 Ziniu Wu，感谢他们在为长上下文问题构建此方法框架时的讨论和帮助。这项研究部分由 Laude 研究所支持。我们也感谢 OOLONG 基准的作者们（他们将保持匿名），允许我们在他们的长上下文基准上进行实验。他们在周一上午10:30告诉我们这个基准，到下午1点就与我们分享了，两天前，我们能够告诉你们这些酷炫的结果，多亏了他们。最后，我们感谢 Jack Cook 和其他 MIT EECS 一年级的学生们在我博士第一年期间的支持！

引用 你可以在完整论文发布前这样引用这篇博客：

```
@article{zhang2025rlm, title = "Recursive Language Models", author = "Zhang, Alex and Khattab, Omar", year = "2025", month = "October", url = "https://alexzhang13.github.io/blog/2025/rlm/" }
```

参考文献 [1-10]...

【深度解读】

致谢部分不仅是对贡献者的感谢，也让我们得以一窥顶级科研工作的真实面貌。一项开创性的工作，离不开实验室内部的激烈思想碰撞、跨学科团队的合作，以及整个学术社区开放、协作的精神。特别是 OOLONG 基准作者的快速响应，生动地体现了现代科研的高速度和高协同性。

引用部分则为这篇博客文章赋予了学术上的严肃性，表明它是一项正式的、可被引用的科研成果，尽管它尚未经过传统的同行评审流程。这反映了 AI 领域快速迭代、拥抱开放交流的文化。