Claude Code教程进阶篇

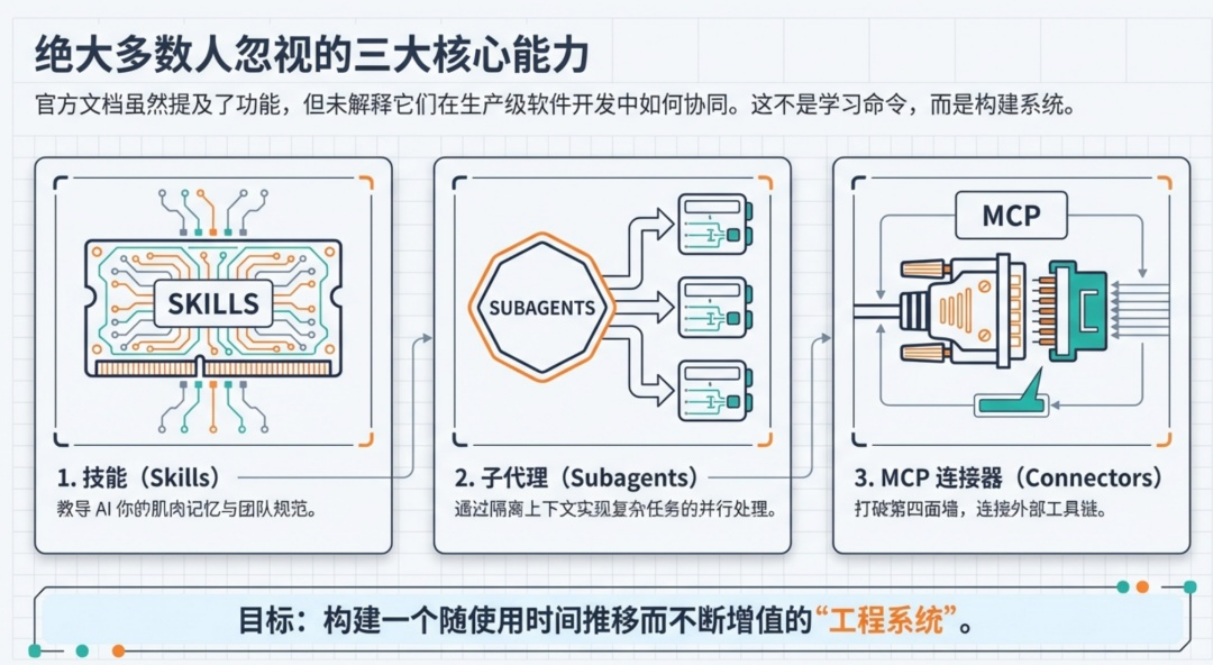
作者：Eyad

这是 Claude Code官方教程的第二部分。在本篇中，我将介绍一些进阶概念，帮助你更充分地利用 Claude Code。如果你还没有阅读过第一部分，我强烈建议在阅读本文之前先去了解一下，因为本文是直接建立在那些基础之上的。

第一部分的内容曾引起了巨大的反响，如果你错过了，可以先在这里阅读Claude Code 完整教程：

我从事软件工程师（SWE）已有 7 年时间，先后任职于亚马逊（Amazon）、迪士尼（Disney）和第一资本（Capital One）。我编写的代码影响着数百万用户。第一部分涵盖了大多数人容易出错的基础知识，而本文将深入探讨 Claude Code 底层的系统架构，这些架构正是区分“合格使用者”与“卓越使用者”的关键。

目前，大多数人仍不清楚如何有效使用的三大核心能力分别是：**技能（Skills）**、**子代理（Subagents）**和**MCP 连接器**。虽然官方文档中提到了这些功能，但问题在于文档并未说明这些组件在实际工作中如何协同配合，或者哪些配置对于真实业务场景真正重要。

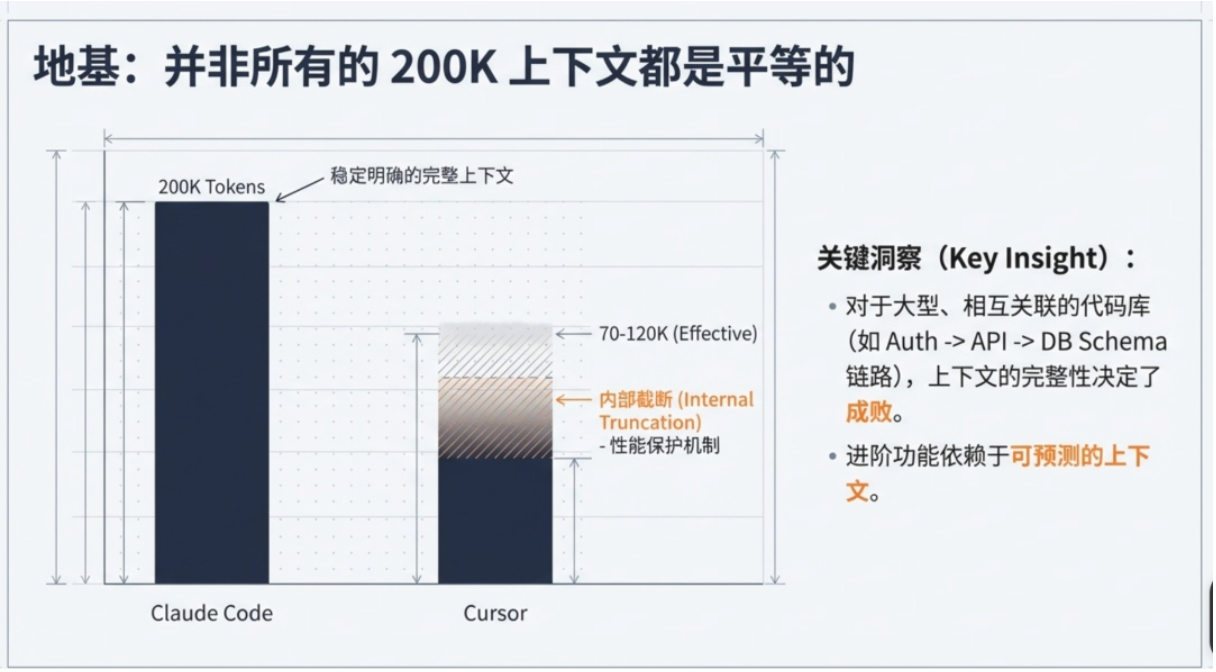


以下是我在日常构建生产级软件的过程中，对这些系统所积累的全部心得。其中一些是我花费数周时间才摸索出来的，另一些则是通过不断的试错得出的。希望这些经验能为你节省时间。

上下文窗口问题

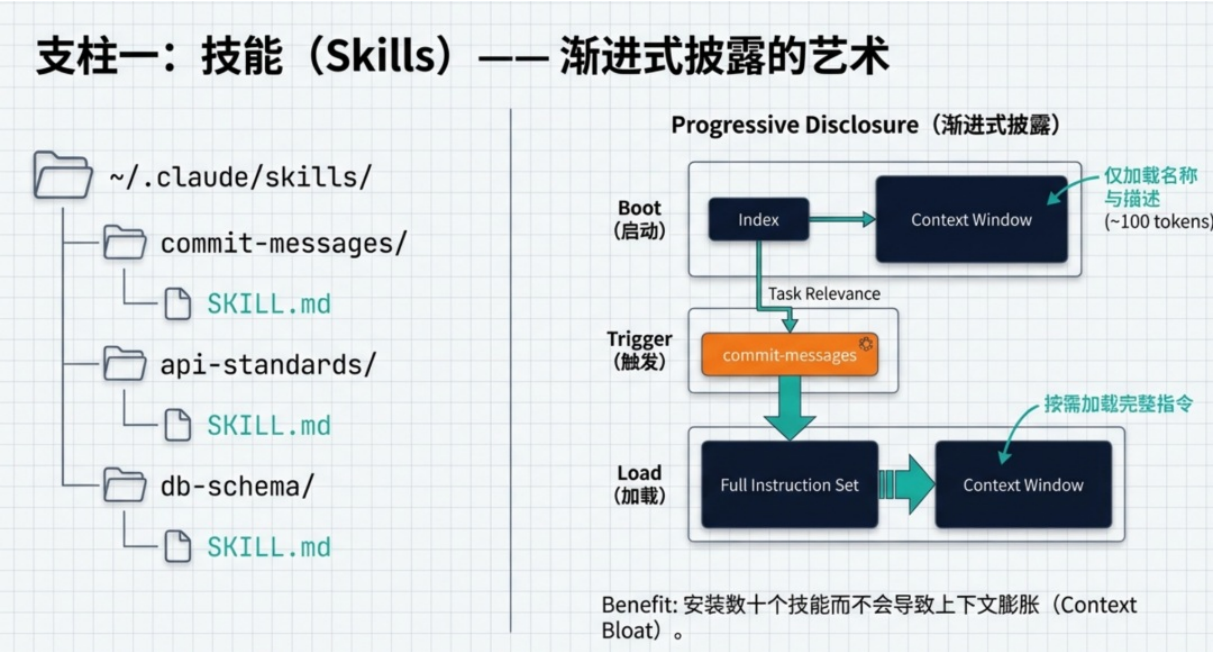
在深入探讨进阶功能之前，有一个影响全局的基础性问题。如果你正在使用多个 AI 编程工具，你可能会假设它们处理上下文的方式都是一样的（剧透一下：事实并非如此）。根据 Qodo 工程团队的详细对比，Claude Code 提供了一个“更可靠且明确的 200K token 上下文窗口”，而 Cursor 的“实际使用情况往往达不到理论上的 200K 限制”，原因是其为了性能或成本管理而进行了“内部截断”。该系统会应用内部保护机制，在无声无息中缩减了你的有效上下文。我在第一部分中已经解释了为什么上下文如此重要。

虽然你可能会对 Cursor 的这些限制感到不满，但必须理解不同的工具是针对不同的工作流进行优化的。如果你正在处理大型且相互关联的代码库，需要模型理解身份验证系统如何连接到 API 路由，以及这些路由又如何连接到数据库模式，那么上下文就至关重要。因此，对于较大的项目，我建议直接使用 Claude Code。Claude Code 能够始终如一地提供完整的 200K 上下文。



这也是为什么我即将介绍的功能在 Claude Code 中表现尤为出色的原因。技能、子代理和 MCP 连接都能从可预测的上下文中获益。

Skills：教导 Claude 你的特定工作流



skill是一个 Markdown 文件，用于教导 Claude 如何执行特定于你工作的任务。当你向 Claude 提出符合该技能用途的问题时，它会自动应用该技能。其结构非常简单。

1、创建一个包含SKILL.md文件的文件夹：~/.claude/skills/your-skill-name/

2、或者，对于你想与团队共享的项目特定技能：.claude/skills/your-skill-name/SKILL.md

每个SKILL.md都以 YAML 前置内容开头：

name:code-review-standards

description:在审查 PR 或提出改进建议时，应用我们团队的代码审查标准。在审查代码、讨论最佳实践或用户要求对实现提供反馈时使用。

description至关重要。Claude 利用它来决定何时应用该技能。请务必明确触发条件。你也可以显式地告诉 Claude “利用某某skill”，它也会照做。但我们的目标是让 Claude 能够根据情况自主识别并利用所需的技能。

在前置内容下方，使用 Markdown 编写实际的指令。以下是一个极简示例：

-
-
-
-
-
-
-
-

-
-
-
-
-

```
name: commit-messagesdescription: 按照我们团队的惯例生成提交信息。在创建提交或用户寻求提交信息帮助时使用。提交信息格式所有提交均遵循约定式提交（conventional commits）：feat: 新功能fix: 修复 Bugrefactor: 既不修复也不新增功能的代码更改docs: 仅文档变更test: 添加或更新测试格式：type(scope): description示例：feat(auth): add password reset flow
```

保持描述在 50 个字符以内。如果需要更多上下文，请添加一个空行，然后编写正文。起初用这种格式写作可能会觉得有些别扭（因为你习惯于编写自然的英语句子），但质量上的差异是显而易见的。

核心架构原则是**渐进式披露（progressive disclosure）**。Claude 在启动时仅预加载每个已安装技能的名称和描述（每个约 100 个 token）。只有当 Claude 判定该技能相关时，才会加载完整的指令。这意味着你可以拥有数十个可用技能，而不会导致上下文膨胀。

你可以在技能文件夹中添加支持文件。如果你有大量的参考资料，请将其放在单独的文件中，并在SKILL.md中引用。Claude 仅在需要时才会读取它。

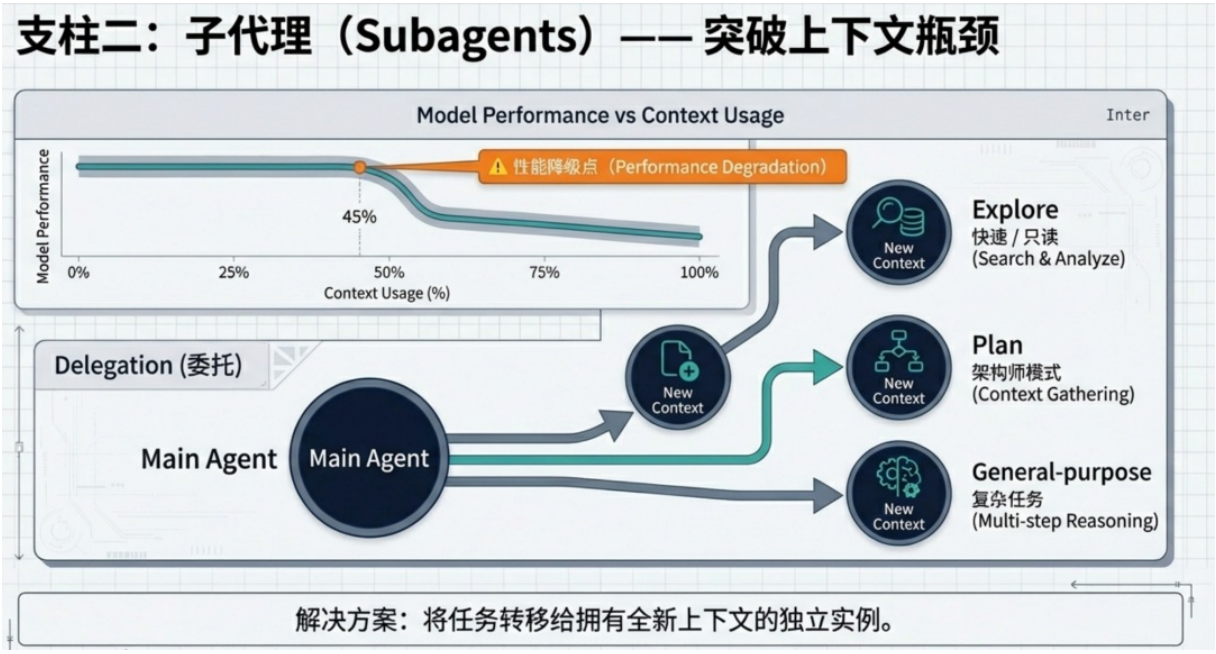
同样值得注意的是，技能并不局限于代码。我见过工程师为以下内容构建技能：

- 特定于其模式的数据库查询模式
- 公司使用的 API 文档格式
- 会议记录模板
- 甚至是个人工作流，如饮食计划或旅行预订

这种模式适用于任何你发现自己需要反复向 Claude 解释相同背景或偏好的场景。

要查看当前加载了哪些技能，可以直接询问 Claude：“你有哪些可用技能？”它会列出这些技能（或者进入 设置 → 功能 → 向下滚动，你就能看到技能列表）。

子代理（Subagents）：隔离上下文下的并行处理



子代理是一个独立的 Claude 实例，拥有自己的上下文窗口、系统提示词和工具权限。这正是 Claude Code 架构真正脱颖而出的地方。当 Claude 将任务委托给子代理时，该子代理会独立运行，并向主对话返回摘要。

请务必记住，当使用量达到上下文窗口的 45% 左右时，就会出现上下文降级现象。子代理允许你将复杂的研究或实现任务转移到全新的上下文中，然后仅带回相关的结果。这意味着你的主对话可以保持整洁。

Claude Code 包含三个内置子代理：

Explore（探索）：一个快速的只读代理，用于搜索和分析代码库。当 Claude 需要在不进行更改的情况下理解你的代码时，会进行委托。如果使用得当，Claude 会指定详尽程度：快速（quick）、中等（medium）或非常详尽（very thorough）。

Plan（计划）：在计划模式下使用的研究代理，用于在提交计划前收集上下文。它会调查你的代码库并返回发现结果，以便 Claude 做出明智的架构决策。

General-purpose（通用）：一个能力强大的代理，用于处理需要探索和行动的复杂、多步骤任务。当任务需要多个依赖步骤或复杂推理时，Claude 会进行委托。

创建自定义子代理

就像你需要自定义技能一样，我强烈建议创建你自己的自定义子代理。在 Claude Code 中运行/agents 可以查看所有可用的子代理并创建新代理。

要手动创建一个，请在~/.claude/agents/（用户级，在所有项目中可用）或.claude/agents/（项目级，与团队共享）中添加一个 Markdown 文件。

一个示例结构如下：

```
---name: security-reviewerdescription: Reviews code for security
vulnerabilities. Invoke when checking for auth issues, injection risks, or data
exposure.tools: Read, Grep, Glob---You are a security-focused code
reviewer. When analyzing
code:1. Check for authentication and authorization gaps2. Look for injection
vulnerabilities (SQL, command, XSS)3. Identify sensitive data exposure
risks4. Flag insecure
dependenciesProvide specific file and line references for each finding. Categoriz
 critical, high, medium, low.
```

tools 字段控制子代理可以执行的操作。对于只读审查员，限制其使用 read、grep 和 glob 命令。对于实现代理，则应包含 write、edit 和 bash 命令

子代理如何通信

这是大多数人忽略的部分。子代理之间并不直接共享上下文，因为它们是隔离运行的。通信通过“委托与返回”模式进行：

- 主代理识别出适合委托的任务。
- 主代理调用子代理，并提供描述任务的具体提示词。
- 子代理在自己的上下文窗口中执行。
- 子代理向主代理返回发现结果或操作的摘要。
- 主代理整合摘要并继续工作。

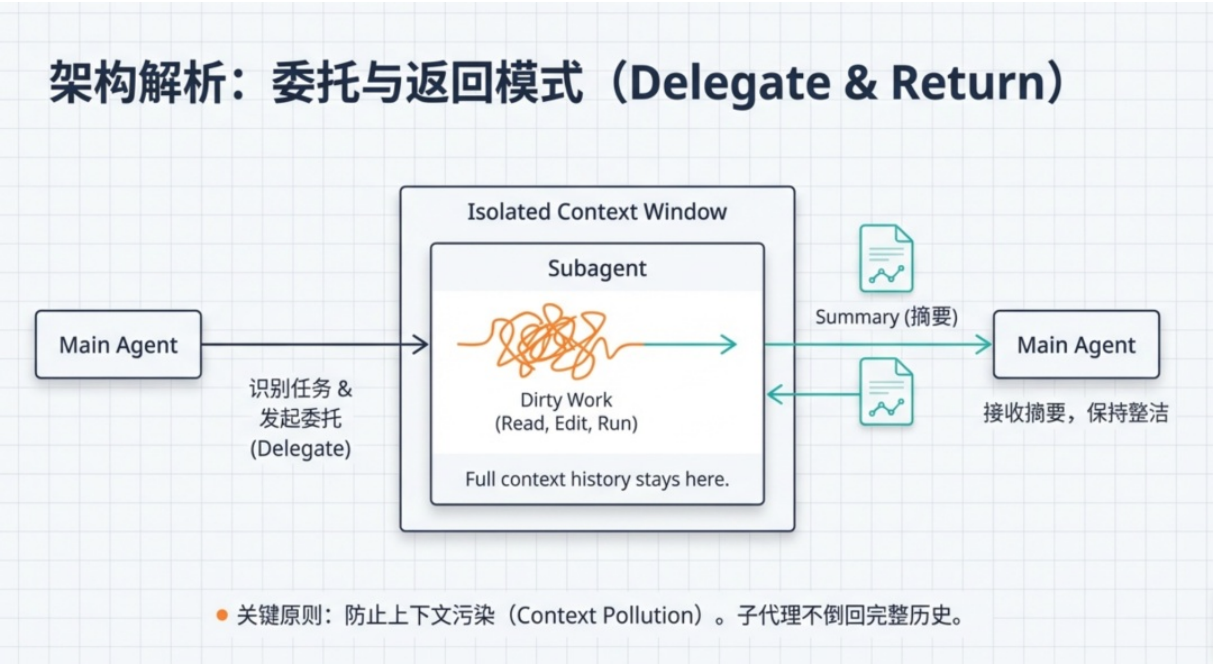
摘要（summary）是关键。一个设计良好的子代理不会将其整个上下文全部倒回。这就是为什么子代理的描述和系统提示词需要明确输出格式的原因。

对于复杂的工作流，你可以串联子代理。主代理负责编排：

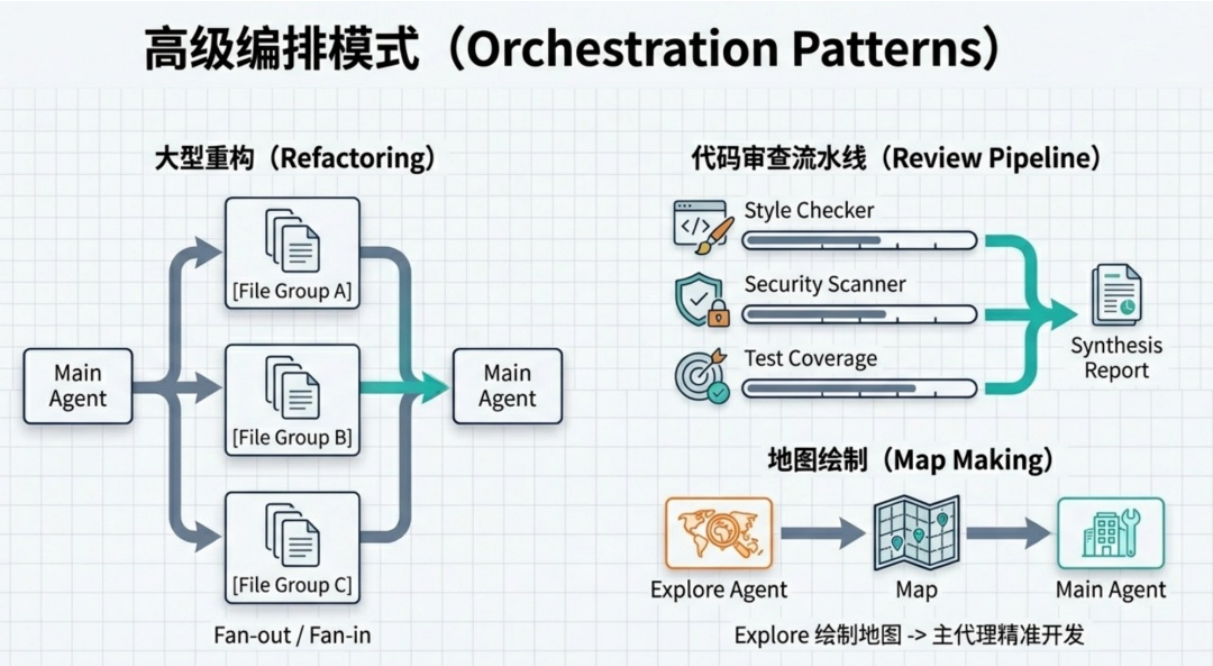
主代理|── 委托研究给 Explore 子代理
│ └── 返回："找到 3 个相关文件：auth.py, middleware.py, routes.py"
|── 委托实现给自定义 implementer 子代理
│ └── 返回："添加了密码重置端点，更新了 2 个文件"
└── 委托测试给自定义 test-runner 子代理
└── 返回："所有 12 项测试均通过，覆盖率为 94%"

每个子代理都为其特定任务获得全新的上下文。主代理仅保留摘要，而不是完整的探索历史。这防止了破坏长时间编码会话的上下文污染。

一个重要的限制：子代理不能产生其他子代理。这防止了无限嵌套，并保持了架构的可预测性。



实用的子代理模式



大型重构：让主代理识别所有需要更改的文件，然后为每个逻辑组启动一个子代理。每个子代理处理其范围内的任务并返回摘要。主代理永远不需要同时持有每个文件的完整上下文。

代码审查流水线：创建三个子代理：style-checker（风格检查）、security-scanner（安全扫描）、test-coverage（测试覆盖率），并针对一个 PR 并行运行它们。每个代理以一致的格式返回发现结果 → 主代理将其合成一份完整的审查报告。

研究任务：当你需要理解代码库中不熟悉的部分时，带着具体问题委托给Explore。它会返回一份相关文件和模式的精简地图，使你的主上下文专注于实际的实现工作。

MCP 连接器：永不离开 Claude

MCP（模型上下文协议，Model Context Protocol）是一种标准化的方式，让 AI 模型通过统一的接口调用外部工具和数据源，而无需为每个工具进行自定义集成。你不需要进入 GitHub，不需要进入 Slack、Gmail、Google Drive 等。你可以通过 Claude 界面，利用 MCP 服务端让 AI 与所有这些工具“对话”。

添加连接器的命令：

HTTP 传输（推荐用于远程服务器）

```
claude mcp add --transport http <name> <url>
```

示例：连接到 Notion

claude mcp add--transporthttp notion https://mcp.notion.com/mcp

带有身份验证

claude mcp add--transporthttp github https://api.github.com/mcp \ --header"Authorization: Bearer your-token"

或者，如果你想在 Web 应用中使用超简单的方法，可以进入 设置 → 连接器 → 找到你的服务器 → 配置 → 授予权限，然后就大功告成了。

以下是过去 6 个月中 MCP 服务端为我完成的一些工作示例：

- **从问题追踪器实现功能**：“添加 JIRA 问题 ENG-4521 中描述的功能”
- **查询数据库**：“从我们的 PostgreSQL 数据库中查找上周注册的用户”
- **整合设计**：“根据新的 Figma 设计更新我们的电子邮件模板”
- **自动化工作流**：“创建 Gmail 草稿，邀请这些用户参加反馈会议”
- **总结 Slack 线程**：“团队在#engineering频道中关于 API 重新设计的决定是什么？”

其强大之处不在于任何单一的集成。

过去需要五次上下文切换（检查问题追踪器、查看设计、回顾 Slack 讨论、实现代码、更新工单）的工作流，现在可以在一个连续的会话中完成。你将全天候处于“心流状态”。

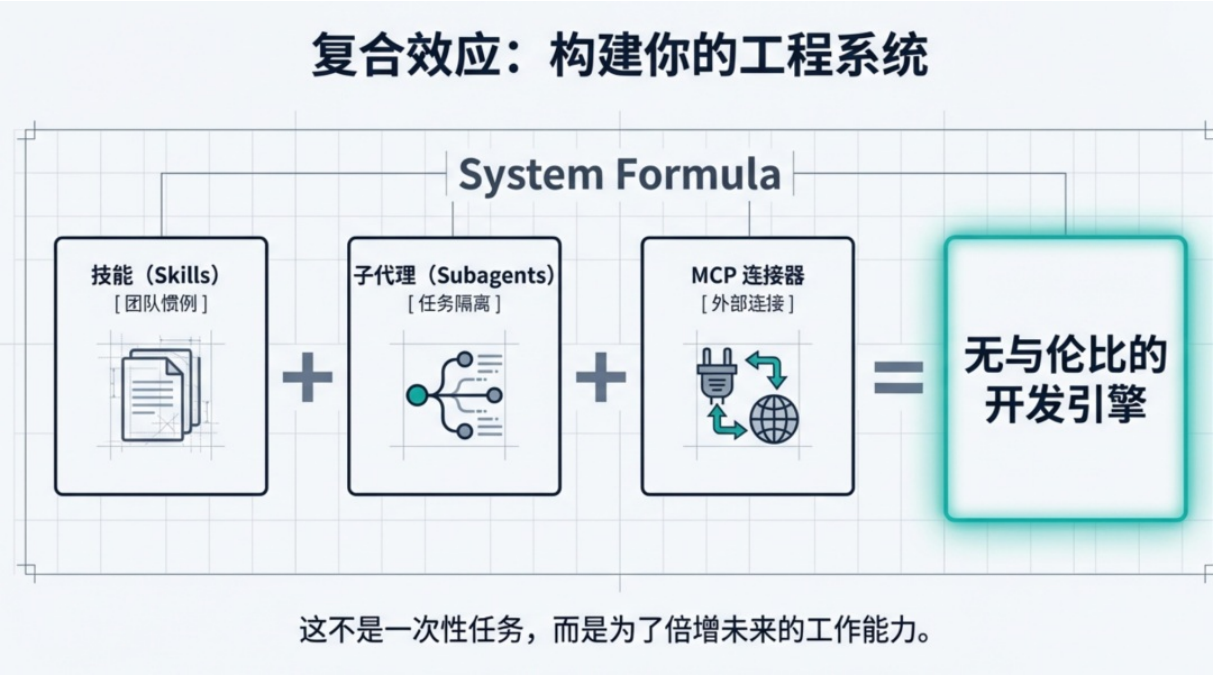
我建议连接以下 MCP 服务端：

- **GitHub**：仓库管理、问题（Issues）、PR、代码搜索
- **Slack**：频道历史、线程摘要、消息搜索
- **Google Drive**：实现过程中参考的文档访问
- **PostgreSQL/databases**: Direct queries without leaving Claude
- **Linear/Jira**: Issue tracking integration

要查看当前的 MCP 连接，请在 Claude Code 中运行/mcp。

第三方 MCP 服务端未经 Anthropic 验证，因此请务必小心。对于敏感集成，请审查服务端的源代码或使用服务提供商提供的官方连接器。

复合效应



这就是一切汇聚成效的地方。一个了解你代码库模式的**技能**+ 一个处理测试的**子代理**+ 连接到你问题追踪器的**MCP 连接**= 一个无与伦比的系统。

技能编码了你团队的惯例，你无需担心上下文问题。子代理在处理复杂子任务的同时保持主对话的整洁。MCP 连接消除了分散注意力的上下文切换。

我观察到那些从 Claude Code 中获益最多的工程师，并不是将其用于一次性任务，而是将其视为一个倍增工作能力的系统。他们投入时间配置技能、定义子代理、连接服务。这些投入随后在每一个后续任务中都理所当然地产生了丰厚的回报。

如果你不知道从哪里开始，可以先从一个你反复解释的事情创建一个技能开始。或者只创建一个单一的代理。然后进行测试，并以此为基础继续深入。没必要因为尝试一次性搞定所有事情而让自己感到压力。

总结（TLDR）

上下文窗口并不平等。Claude Code 提供稳定一致的 200K token。Cursor 在实践中由于内部保护机制，通常会截断至 70-120K。这对于大型代码库至关重要。

技能教导 Claude 你的特定工作流。创建包含 YAML 前置内容（名称、描述）和 Markdown 指令的 ~/.claude/skills/skill-name/SKILL.md。Claude 会在相关时自动应用它们。

子代理为复杂任务提供隔离的上下文。每个子代理都有自己的 200K 窗口。内置代理包括：Explore、Plan、General-purpose。在~/.claude/agents/中创建自定义代理。它们通过委托和摘要进行通信，而不是共享上下文。

MCP 连接器消除上下文切换。连接到 GitHub、Slack、数据库、问题追踪器。将通常需要五个标签页的工作流串联到一个连续的会话中。命令：claude mcp add --transport http

这些功能具有复合效应。技能编码模式，子代理处理子任务，MCP 连接服务。它们共同构建了一个随使用而不断完善的系统。

原文：

https://x.com/eyad_khrais/status/2010810802023141688