

Claude Code

从入门到精通

面向工程师与产品经理的AI编程完全指南

适用版本：Claude Code v2.1.30+ | 模型：Opus 4.6 / Sonnet 4.6 / Sonnet 4.5

新增：Computer Use · Voice Mode · Agent Teams · /loop · Claude in Chrome · 1M Context

QingTeng·AI

<https://qingtengstudio.com/>

本手册基于Anthropic官方文档、Claude Code Changelog及社区最佳实践编写。所有操作细节以2026年4月最新资料为准。AI工具迭代极快，请结合官方文档验证。

目录

Part 1: 起步

§ 01 为什么是Claude Code

§ 02 10分钟完成安装

§ 03 你的第一个项目

Part 2: 核心能力

§ 04 核心 workflow

§ 05 CLAUDE.md: 给AI一张地图

§ 06 进阶对话技巧

Part 3: 进阶实战

§ 07 扩展能力: Skills、Hooks与MCP

§ 08 多Agent协作

§ 09 从零构建一个完整产品

§ 10 心智模型与持续进化

附录

附录A Claude Code演进时间线

附录B 新版本功能速查

附录C 推荐资源

§ 01 为什么是Claude Code

Why Claude Code

AI编程工具在三年里变了三次。搞清楚这个演变路径，你就能理解Claude Code到底在做一件什么不一样的事。

三年变了三次

2022年，GitHub Copilot出来了。你写上半句，它帮你猜下半句。像一个坐在旁边的实习生，打字确实快了，但本质没变：你还是那个写代码的人。

2023到2024年，Cursor火了。你可以用自然语言让编辑器帮你改函数、重构模块，不用精确描述「怎么写」，说「我想要什么效果」就行。后来Cursor也加了Agent模式，能跨文件操作、自动跑命令。但它始终长在IDE里，是编辑器的延伸。

2025年，Claude Code出来了。它不住在任何编辑器里，直接在终端运行。你描述一个需求，它自己规划步骤、读代码、写代码、跑测试、操作git，整个循环自动完成。你的角色从「写代码的人」变成了「给指令的人」。

Copilot 补全 → Cursor 对话 → Claude Code 终端Agent

三步走下来，变的不是技术有多先进，而是你和AI之间的关系。Copilot是你的输入法，Cursor是你的结对伙伴，Claude Code是你的独立工程师团队。

它跟Cursor到底有什么不一样

这是被问得最多的问题：「Cursor也有Agent模式了，不都是AI帮我写代码吗？」确实，2024年之后的Cursor已经很强了，能跨文件操作、理解整个项目、自动执行命令。两者的差异不在「能不能做」，而在做到什么程度。

维度	差异
运行环境	Cursor：编辑器内嵌，依赖IDE框架；Claude Code：终端原生，直接操作操作系统
自主程度	Cursor：通常需要你在旁边确认；Claude Code：可以完全无人值守运行
系统集成	Cursor：通过插件桥接git/CLI；Claude Code：直接操作git、shell、MCP
记忆系统	Cursor：隐式的项目索引；Claude Code：显式的CLAUDE.md记忆文件
并行能力	Cursor：主要单实例工作；Claude Code：原生支持多实例并行、Agent Teams
最新能力	Cursor：IDE内深度编辑；Claude Code：Computer Use操作桌面、Voice Mode语音编程

重点看最后两行。CLAUDE.md让你把项目知识、编码规范、架构决策写成文件，Claude Code每次启动都会读，相当于给AI一个持久的项目记忆。多实例并行和Agent Teams意味着你可以同时让几个Claude Code各自处理不同模块，像一个小团队。

打个比方：Cursor像坐在你IDE里的结对伙伴，你们看着同一个屏幕协作；Claude Code更像一个独立干活的工程师，你告诉他需求，他自己拉代码、写代码、跑测试、提交，你去喝杯咖啡回来看结果就好。

Boris Cherny，Claude Code的创建者，说自己用Claude 4系列模型后基本不再手写代码。47天里有46天都在用，最长单次session跑了1天18小时50分钟。这不是营销话术，是一个正在大规模发生的现实。

它其实不是在帮你写代码

这句话听起来有点奇怪，但确实是我用了半年之后最大的感受：Claude Code不是在帮你写代码，它在帮你构建产品。

传统的AI编程工具解决的是代码生产效率：怎么更快地写出这个函数、这个组件。Claude Code解决的是产品构建效率：怎么更快地从一个想法变成一个能跑的东西。

推荐用Claude Code：「帮我做一个支持Markdown的博客系统，用Next.js，部署到Vercel，支持暗色模式和RSS。」它会：分析需求 → 选技术方案 → 创建项目 → 逐步实现 → 跑测试 → 修bug → 完成。全程你只需要确认和调整方向。

不推荐用IDE内的Agent：体验也不差，但你大概率需要：盯着IDE看它改了什么 → 出了问题手动切回来 → 它不太敢自己跑命令需要你确认 → 你始终是坐在旁边的人。全程你是监工。

前者你在做产品决策，后者你在做过程监督。随着AI能力持续提升，「盯着AI干活」会越来越不值钱，产品决策能力会越来越值钱。

增长有多快

说几个数字。Claude Code在2025年2月公开发布（研究预览版），5月正式GA。GA后收入增速极快，weekly active users在2026年初翻倍增长。这个速度在SaaS历史上极其罕见。

企业端采用也很快。Netflix、Spotify、DoorDash、Notion、Vercel都在内部大规模使用。Anthropic的数据显示，使用Claude Code的团队平均提效2-5倍。

截至2026年4月，Claude Code背后的主力模型有三个：

- Opus 4.6 — 推理能力最强，处理复杂任务和架构决策，支持1M token上下文
- Sonnet 4.6 — 性价比最优，日常编码的主力，也是最佳的Computer Use模型
- Sonnet 4.5 — 响应较快，适合一般性编码任务和中等复杂度项目

这些数字背后的信号其实就一个：Agent式编程不再是极客的玩具，正在变成软件开发的标准方式。

全书路线图

全书按一条主线走：一个聪明人怎么在一周内从零用AI构建产品。

阶段	章节	你会学到
Day 1: 上手	§ 01- § 03	理解AI编程 → 安装配置 → 做出第一个项目
Day 2-3: 核心	§ 04- § 06	掌握工作流 → 配置记忆系统 → 学会有效沟通
Day 4-5: 进阶	§ 07- § 08	扩展能力（Skills/MCP） → 多Agent协作
Day 6-7: 实战	§ 09- § 10	独立构建完整产品 → 建立长期心智模型

每章都有实操部分，可以跟着做。不用一口气读完，读一章、做一章、再回来读下一章，完全没问题。

下一章，我们花10分钟把Claude Code装好。

§ 02 10分钟完成安装

Get Started in 10 Minutes

安装过程比你想象的简单。这一章覆盖安装、使用环境选择、付费方案，以及你和Claude Code的第一次对话。

三种安装方式

macOS、Linux、Windows都支持。选哪个取决于你的系统：

安装方式	命令	适用平台	推荐度
Native Install	<code>curl -fsSL https://claude.ai/install.sh bash</code>	macOS / Linux	★ 推荐
Homebrew	<code>brew install --cask claude-code</code>	macOS	适合brew用户
WinGet	<code>winget install Anthropic.ClaudeCode</code>	Windows	Windows首选
NPM	<code>npm install -g @anthropic-ai/claude-code</code>	跨平台	备用方案

[核心建议] 不确定选哪个？用Native Install。一行命令搞定，不需要额外依赖。

macOS / Linux 安装

打开终端（macOS用Terminal或iTerm2，Linux用你习惯的终端），运行：

```
curl -fsSL https://claude.ai/install.sh | bash
```

脚本会自动检测系统、下载二进制文件、把 `claude` 命令加到你的PATH。装完输入 `claude` 就能启动。Homebrew用户也可以用 `brew install --cask claude-code`，效果一样。

Windows 安装

Windows的安装多一个前置步骤：你需要先装Git for Windows。

- 从 git-scm.com 下载安装，或者用WinGet: `winget install Git.Git`。安装时默认选项即可，它会带一个Git Bash终端。
- 打开PowerShell或Git Bash，运行: `winget install Anthropic.ClaudeCode`
- 重新打开终端，输入 `claude --version`，能看到版本号就说明安装成功。

[注意]Windows用户请注意：Claude Code需要Git Bash提供的Unix工具链。如果你直接在CMD中运行可能会遇到问题，建议使用PowerShell或Git Bash。

五种用法，选哪个

装完之后，其实有五种方式使用Claude Code。体验各有不同：

环境	特点	适合谁
终端CLI	最原生的体验，功能最完整，直接在终端输入 <code>claude</code>	日常开发的主力方式
VS Code扩展	在VS Code侧边栏运行，可以直接看到文件变更，支持 inline edits	习惯VS Code的开发者
Desktop App	独立桌面应用，不需要打开终端	不熟悉终端的用户
Web版	浏览器直接访问 claude.ai/code ，无需安装	临时使用、体验试用
JetBrains插件	在IntelliJ IDEA、WebStorm等JetBrains IDE中使用	JetBrains用户

建议：这本书后续所有演示都基于终端CLI。就算你平时用VS Code或JetBrains，也建议先在终端把Claude Code的完整能力摸熟，之后再切到IDE集成。终端才是它的完全体。

账号和付费方案

Claude Code需要Anthropic账号。首次启动时会自动弹浏览器让你登录或注册。

截至2026年4月，付费方案如下：

方案	月费	用量	适合谁
Pro	\$20/月	基础用量，日常开发够用	个人开发者、学习者
Max 5x	\$100/月	5倍于Pro的用量	重度用户、全职AI编程
Max 20x	\$200/月	20倍于Pro的用量	团队用户、商业项目
Team	按席位	团队协作+管理后台	中小团队
Enterprise	定制	SSO、审计日志、合规	大型企业

[核心建议]先从Pro开始就好。用几天你自然知道够不够。很多人的路径是：第一周觉得Pro够了，第二周开始上瘾然后升Max。

企业用户还可以通过Anthropic API按token计费，适合有自定义集成或合规要求的团队。Claude Code支持通过API Console账号登录（走API计费），不需要再单独配置API密钥。

说第一句话

装好了，账号也登录了。来试试。

- 打开终端，进入一个项目目录。Claude Code会以你当前所在的目录作为工作目录。建议新建一个测试文件夹：`mkdir ~/my-first-project && cd ~/my-first-project`
- 启动Claude Code：`claude`。首次启动会打开浏览器让你登录Anthropic账号。登录成功后，终端会显示Claude Code的交互界面。

- 发送你的第一条指令：「创建一个简单的HTML页面，显示Hello, Claude Code!，用好看的CSS样式。」
- Claude Code创建完文件后，你可以直接在浏览器打开看效果：open index.html (macOS) / start index.html (Windows)

看起来简单，但注意它的意义：一句自然语言，AI完成了「理解需求 → 创建文件 → 编写代码」的完整循环。后面所有进阶操作都是在这个基础上展开的。

确认一切正常

检查项	命令/操作	预期结果
CLI可用	claude --version	显示版本号
账号已登录	claude 启动后不再要求登录	直接进入对话界面
能创建文件	让Claude Code创建一个测试文件	文件出现在当前目录
能读取项目	在已有项目目录启动，问「这个项目是做什么的」	Claude Code能正确描述项目
能运行命令	让它运行 ls 或 git status	返回命令输出结果

全部通过？可以开干了。

遇到问题？

连不上、登录不了：Claude Code需要访问Anthropic的API服务器。国内网络环境下可能连不上，需要配代理：

```
export HTTPS_PROXY=http://127.0.0.1:7890
export HTTP_PROXY=http://127.0.0.1:7890
```

建议把这两行加到你的shell配置文件（ ~/.zshrc 或 ~/.bashrc ）中，这样每次打开终端都会自动生效。

安装时报Permission denied: macOS / Linux用户遇到权限错误, 不要用 sudo 。这么搞:

```
mkdir -p ~/.local/bin  
curl -fsSL https://claude.ai/install.sh | bash
```

如果问题仍然存在, 检查你的PATH中是否包含 ~/.local/bin 。

怎么升级

Claude Code会提醒你有新版本。手动更新就重新跑一遍安装命令。保持更新。Claude Code迭代极快, 几乎每周都有新功能。新版本不只是修bug, 经常带来能力上的明显提升。建议至少每两周更新一次。

新功能: Claude in Chrome (Beta)

2025年12月推出的Claude in Chrome是一个浏览器扩展, 让Claude Code能直接控制你的Chrome浏览器。你可以让Claude打开网页、填写表单、提取数据、甚至走完整的多步骤流程。安装扩展后, 在Claude Code中直接使用即可, 无需额外配置。

适用场景: 前端开发者测试网页交互、爬虫场景自动化、后台管理系统操作。目前Beta版已向所有付费用户开放。

装完了, 账号登了, 第一次对话也跑通了。下一章, 正式开始做一个真实项目。

§ 03 你的第一个项目

Your First Project — Learning by Doing

理论讲完了，直接上手。这一章从零做一个真实的CLI工具。做完之后，你就真正理解对话式编程是怎么回事了。

做个什么

一个每日AI新闻聚合器，CLI工具。功能很简单：

- 从几个RSS源（TechCrunch AI、The Verge AI、Hacker News等）抓取最新文章
- 用AI总结每篇文章的要点
- 输出一份格式整齐的Markdown日报

为什么选这个？够小，一个下午能做完。又够完整，涉及网络请求、数据处理、AI调用、文件输出。一次项目就能把Claude Code的各种能力体验个遍。

先转换一下心态：从现在开始，你是产品经理，Claude是你的工程师。你的活是说清楚要什么，不是写代码。哪怕你是资深程序员，也请先按这个方式来一遍。

第一步：告诉Claude你要什么

打开终端，进一个空文件夹，启动Claude Code：

```
mkdir ai-news-digest && cd ai-news-digest
claude
```

然后，用自然语言告诉Claude你想要什么。注意最后那句「先别急着写代码，给我一个实现方案」。这不是客气，是一个重要技巧：让Claude先想清楚再动手。

[核心建议]需求描述的质量直接决定输出质量。几个原则：说清楚最终产物长什么样（Markdown日报）、指定技术选型（TypeScript + tsx）、给具体数据源URL而不是说「几个RSS源」。越具体越好。

第二步：看它怎么想的

Claude收到需求后，不会直接开写，会先给你一个方案。大概长这样：项目结构建议、技术方案（rss-parser库、node内置fetch、date-fns）、实现流程。

这时候你就是在看工程方案的产品经理。觉得行就说OK，想调整就直接说。这个来回就是对话式编程的核心：Claude出方案，你补充细节，Claude修正。不用画流程图、不用写技术文档，直接用人话说。

第三步：看它干活

确认方案后，Claude开始执行。终端里会看到一系列操作：

- 初始化项目：Claude会运行 `npm init -y`，然后安装依赖包。你会看到它请求运行命令的权限提示。按 `y` 允许。
- 创建源代码文件：Claude会逐个创建TypeScript文件。你会看到它写入代码的过程，每个文件都会展示差异（diff）。
- 运行测试：代码写完后，Claude通常会自己试着运行一次看看有没有报错。如果报错了，它会自己读错误信息、找问题、修代码、再运行，形成一个自动修复循环。

整个过程大约2-5分钟。你干嘛？看着就行。就像把任务交给新同事，前几次你会盯着看他怎么做事。等熟悉了，以后放心让他自己干就好。

[注意]执行过程中Claude可能多次请求权限。初期建议每次都看一眼它要跑什么命令。熟悉后可以用 `/permissions` 预授权常见命令（§ 04会细讲），或者直接开Auto模式。

第四步：看看结果对不对

Claude干完了，跑一下看看：

```
npx tsx src/index.ts
```

如果一切顺利，你会在 output/ 目录下看到一个Markdown文件。打开文件看看格式和内容。大多数情况下，第一次就能跑通。

如果报错了？直接把错误信息丢给Claude。Claude会读错误信息、定位问题、改代码、再跑。这个报错到修复的循环，一般1-2轮就搞定。

第五步：加功能

能跑了，但你想加点东西。继续用自然语言说就行：

- 第一个改进：加AI摘要。改成用AI来总结：对每篇文章的标题+description，用Claude API生成一句话总结。API key从环境变量 ANTHROPIC_API_KEY 读取。
- 第二个改进：加定时运行。加一个cron模式，每天早上8点自动运行一次，用node-cron实现。
- 第三个改进：加去重逻辑。有些文章在多个源里重复出现了。加一个基于URL的去重。

每一轮，Claude改代码、跑测试、确认结果。你始终只做两件事：说清楚要什么，验证结果。

到这里，第一个项目完成了。回头看看整个过程：描述需求 → 审查方案 → 确认执行 → 验证结果 → 迭代改进。不管项目大小，底层模式都是这五步。

一个重要的心态转变

做完这个项目，你应该有个直觉上的感受：你的价值不在于写代码，而在于定义要做什么、判断做得对不对。

很多工程师第一次用Claude Code时，本能地想看每一行代码、理解每个实现细节。正常反应，但它会让你变慢。

更高效的方式是把它当团队成员来管理。传统编程是自己想方案、自己写代码、一行一行调试、查文档查StackOverflow。Claude Code编程是描述需求Claude出方案和代码、把错误信息给Claude它自己调、直接问Claude「怎么实现XXX」、让Claude解释它写的代码、告诉Claude「把这块重构成XXX模式」。

不是说完全不管代码。而是你的注意力应该在更高层面：需求准不准确？方案合不合理？结果符不符合预期？这些才是你该花时间的地方。

新手常见困惑

「代码看不懂怎么办？」直接问它。「解释一下 `fetcher.ts` 的实现逻辑」，Claude会用人话讲清楚。还可以追问：「为什么用 `Promise.allSettled` 而不是 `Promise.all`？」它会解释背后的技术选择。你不需要能写出这段代码，但需要理解它在做什么。就像你不用会修发动机，但得知道车在正常运转。

「写错了怎么办？」直接说哪里不对。你不需要知道怎么改，描述现象就够了：「运行后只输出了TechCrunch的文章，另外两个源的文章没有。检查一下抓取逻辑。」描述现象比定位代码行更有效。

「该管多少？」四个字：信任但验证。让Claude去做，但每一步检查结果。方案阶段仔细看，确保方向对。编码阶段扫一眼文件结构就行。运行阶段看输出符不符合预期。改进阶段多测试边界情况：空数据、网络超时、格式异常。这个分寸感做几个项目自然就有了。

「跑偏了怎么办？」偏得不远，直接纠正：「停，不要用XXX库，换YYY。」偏得太远，按Esc停止，重新描述需求。按两次Esc会打开Rewind菜单，可以回滚对话、回滚代码改动，或者两者都回滚。一个经验：纠正两次还不行，就果断停下来重来。在错误基础上打补丁只会越补越乱。

这一章的核心：描述需求、审查方案、确认执行、验证结果、迭代改进。你管要什么和好不好，Claude管怎么实现。这个分工形成默契之后，生产力会有质的变化。

§ 04 核心 workflow

Core Workflows — The Patterns That Matter

跑通第一个项目之后，你可能觉得 Claude Code 也就这样——写代码、确认权限、看结果。但日常用下来，真正拉开效率差距的是几个核心工作模式。这一章把它们拆开聊。

Plan 模式：先想清楚再动手

Boris（Claude Code 的创建者）说过一句话：「一个好的计划真的很重要。」他自己大多数会话都从 Plan 模式开始。

Plan 模式的作用很直接：让 Claude 只规划、不执行。它会告诉你它打算怎么做，但不会动你的代码、不会装包、不会运行命令。你们来回讨论方案，确认了再放手让它去做。

如何进入 Plan 模式：在 Claude Code 的输入框中，按两次 Shift+Tab。你会看到界面切换到 Plan 模式。此时 Claude 的行为会改变：它可以读取文件来理解代码，但不会修改任何文件；它会给出详细的实现方案，包括要改哪些文件、怎么改；你可以反复讨论、修改方案。

Boris 推荐的完整流程是这样的：

- Plan 模式下描述需求，来回讨论。用 Shift+Tab × 2 进入 Plan 模式，描述你的需求。Claude 给出方案后，你可以说「第三步换个方式」「这里的库用 XXX 替换」，反复调整。
- 用编辑器写一份详细的执行指令。方案大致满意后，按 Ctrl+G，会在你的默认编辑器（由 \$EDITOR 环境变量决定）中打开输入框。你可以在编辑器里写一份完整的执行指令，把讨论中确认的方案细节、约束条件都写进去，保存后内容回到 Claude Code 的输入框，作为下一步的 prompt 发出。
- 切换到执行模式，开启 Auto-accept。计划确认后，按 Shift+Tab 切回正常模式。因为计划已经讨论充分，你可以放心地让 Claude 一次性执行完成，不需要逐步确认。

这个流程的精髓在于：把纠结放在 Plan 阶段解决完，执行阶段一气呵成。边做边改、反复返工是最浪费 tokens 的用法。

[核心建议] 一条实用判断标准：如果你需要跟同事解释才能让他做的任务，就值得用 Plan 模式。如果你能用一句话说清的任务，直接做就行。

Auto模式：更安全的自动驾驶

用了一阵Claude Code之后，你大概已经体会到那种「烦躁感」了。每次写文件要确认、运行命令要确认、装个依赖也要确认。按到第50次 y 的时候，你根本不看它要执行什么了。

Anthropic内部的数据证实了这一点：93%的权限请求被用户直接批准了。审批疲劳让安全机制形同虚设。

Auto模式就是为了解决这个问题。核心思路：用一个AI分类器替你做权限判断。安全操作自动放行，危险操作才拦截。

Auto模式的工作原理：Auto模式有两层防御。输入层：一个Prompt Injection探测器会扫描Claude读取的所有内容（文件、网页、命令输出）。如果内容看起来像要劫持Claude的行为（比如某个文件里写着「忽略之前的指令」），探测器会在内容传给Claude之前附加一条警告。输出层：一个Transcript分类器会在每个操作执行前评估它的风险。分类器分两个阶段运行：第一阶段快速判断（一个token的 yes/no），倾向于谨慎，有8.5%的误报率；第二阶段只对第一阶段标记的操作做深度推理，把误报率降到0.4%。

Auto模式会拦截什么？范围升级（你说「清理旧分支」，Claude把远程分支也删了）、凭证探索（Claude遇到认证错误，开始在环境变量中搜索其他API token）、绕过安全检查（部署命令预检失败，Claude用 `--skip-verify` 重试）、数据外泄（Claude想分享代码，自行创建了一个公开的GitHub Gist）。

```
# 启动时指定
claude --permission-mode auto
# 或者在运行中用 Shift+Tab 循环切换模式
# 默认 → Auto-accept edits → Plan → Auto → 默认
```

[注意]Auto模式不是万能的。Anthropic公开的数据显示，对于「过度主动行为」，分类器仍有17%的漏检率。操作生产数据库、管理云基础设施这类事情，还是手动确认比较踏实。Auto模式最适合日常开发：写代码、跑测试、Git操作。

权限管理：你来定规矩

Auto模式之外，Claude Code还有更精细的权限控制。

`/permissions`：预授权安全命令。输入 `/permissions` 打开权限管理界面。你可以预先允许 Claude 执行某些操作，这样它就不会每次都问你了。支持通配符匹配：

```
# 允许运行所有npm脚本
Bash(npm run *)
# 允许编辑docs目录下的所有文件
Edit(/docs/**)
# 允许运行测试
Bash(npx vitest *)
Bash(npx jest *)
# 允许git操作
Bash(git add *)
Bash(git commit *)
Bash(git push)
```

这些规则可以保存到 `.claude/settings.json` 并提交到Git，让整个团队共享同一套权限配置。

[核心建议] Boris的做法：不用Auto模式也不跳过权限，而是用 `/permissions` 仔细配置一套白名单。白名单会check进git，和团队共享。这是最精细也最安全的方案，只是初始配置需要花点时间。

三层权限选择：

方式	省心程度	安全程度	适合谁
Auto模式	高	中（AI分类器保护）	大多数日常开发者
<code>/permissions</code> 白名单	中	高（精确控制每条命令）	团队使用、需要精细控制
逐个确认（默认）	低	最高	高风险操作、初学阶段

Git操作：Claude天然就懂

Claude Code对Git的理解不只是帮你跑 git 命令。它真的知道你项目当前的版本控制状态，知道你改了哪些文件、在哪个分支上。

一句话commit和PR：

```
# 让Claude自己看看改了些什么，写一个有意义的commit message
提交当前的改动，写一个有意义的commit message
# 或者更具体一点
commit这些变更，描述清楚我们添加了RSS抓取功能
# 直接创建PR
创建一个PR，标题和描述写清楚这个功能的作用
```

Claude会分析你的代码变更，生成一个描述性的commit message，然后执行 git add + git commit。创建PR时它还会自动生成PR描述，包括改了些什么、为什么改。

Git Worktrees：并行工作利器

这是Boris的第一推荐技巧。Git worktree允许你在同一个仓库中同时checkout多个分支，每个分支有自己的工作目录。

```
# 创建一个worktree，在新目录中开始工作
claude --worktree
```

--worktree 标志会让Claude Code自动创建一个新的git worktree，在一个隔离的目录中工作。好处是：不影响你当前分支的代码；可以同时开多个worktree，每个处理不同的任务；每个worktree有独立的工作目录，Claude不会互相干扰。

这在多任务并行时特别有用。比如你在修一个bug的同时，想让Claude在另一个分支做一个新功能。用worktree，两件事互不干扰。

Boris的并行工作方式：他在终端里同时运行5个Claude Code实例，每个在不同的worktree中工作。加上claude.ai/code网页端的5-10个会话，他一个人就能同时推进十几个任务。这就是Agent式工作的威力：你不需要自己做所有事，你管理一群Agent帮你做事。

Computer Use: AI长了眼睛和手

2026年3月推出的Computer Use是Claude Code最突破性的功能之一。前面讲的所有工作流，Claude都是在文本世界里操作的：读代码、写代码、跑命令行。文字处理这块它确实强，但你桌面上那个Figma窗口、那个Photoshop、那个只有GUI没有API的老旧管理后台，它碰不到。

现在可以了。Claude Code的Computer Use功能让Claude直接看到你的屏幕截图，然后操控鼠标和键盘。不是模拟，不是调API，是真的在看你的屏幕、移动你的光标、点击你的按钮。

怎么用：零配置。Pro和Max订阅用户自动可用（目前为Research Preview），你不需要开任何开关。Claude在工作过程中如果判断需要操作GUI，它会自己截一张屏幕截图来「看」当前画面，然后决定下一步该点哪里、该输入什么。

你也可以主动让它看屏幕：

```
# 让Claude看看当前屏幕上的东西
看一下我屏幕上的这个页面，告诉我布局有什么问题
# 让它操作一个GUI应用
打开系统偏好设置，把暗色模式关掉
# 测试你正在开发的Web应用
在浏览器里打开 localhost:3000，走一遍注册流程，看看有没有bug
```

实际场景：测试Web应用的UI（Claude不只是跑测试脚本，它能像用户一样点击页面、填表单、看到渲染结果，发现视觉上的问题）；操作没有API的桌面软件（老旧的管理后台、只有GUI的工具）；自动化重复的GUI操作（批量处理文件、在多个窗口之间来回复制数据）；调试Chrome扩展（扩展的popup、content script效果只能在浏览器里看到，Claude可以直接截图查看并定位问题）。

[核心建议]Computer Use现阶段最好的定位是：把它当成一个耐心但手速慢的测试员。给它那些「按照固定流程重复操作」的任务，它做得很好。需要灵活判断和快速反应的，还是自己来。当前限制：macOS only；每一步操作都需要截图→分析→决定→执行，比人类慢；精细操作（拖拽滑块到精确位置、选中特定单元格）经常偏。

Voice Mode: 开口说话就能编程

2026年3月推出的Voice Mode让Claude Code支持语音输入。按住空格说话，松开发送。就这么简单。

在Claude Code里输入 /voice，就进入了语音模式。支持20种语言，中文当然包括在内。你按住空格键说出你的需求，松手后Claude会把语音转成文字，然后像正常输入一样处理。

什么时候用语音比打字好：

- 手不方便的时候。走路想到一个bug的修复思路、做饭时突然想起一个需求。掏出手机（如果你用SSH连了服务器的话）或者对着电脑说一嘴，比找键盘快。
- 脑暴的时候。想法哗哗地冒，打字速度跟不上大脑。语音可以一口气把一段混乱的思路倒给Claude，让它帮你理成结构化的需求。
- 描述空间和视觉概念的时候。「我想要一个左边是侧边栏、右边分上下两栏、上面是图表下面是表格」，这种话说出来比画ASCII图快多了。

语音模式目前还有几个不够顺滑的地方：需要相对安静的环境，嘈杂背景下识别率会下降；长指令还是打字更精确。目前最适合的是启动任务和快速交互：「帮我跑一下测试」「把这个函数重命名成XXX」「看看这个文件有什么问题」。

[核心建议]一个实用的组合：语音快速启动任务（「帮我做个XXX」），然后切回键盘输入精确的细节和约束条件。两种交互方式混着用，比纯用一种效率高。

会话管理：别让上下文变成垃圾场

Claude Code有上下文限制。对话越长，Claude对当前任务的注意力越分散。用好Claude Code，会话管理这件事比你想象的重要得多。

核心命令速查：

操作	命令/快捷键	什么时候用
清空当前会话	<code>/clear</code>	切换到完全不相关的任务时
压缩上下文	<code>/compact</code>	会话太长、Claude开始变慢或遗忘
停止当前操作	Esc	Claude在做你不想要的事
Rewind（回滚）	Esc × 2 或 <code>/rewind</code>	Claude改坏了代码，打开回滚菜单选择恢复对话/代码/两者
恢复上次会话	<code>claude --continue</code>	终端不小心关了，想接着之前的会话
恢复指定会话	<code>claude --resume</code>	想回到某个历史会话继续工作
侧链提问	<code>/btw</code>	想问个不相关的问题，不污染当前上下文
查看上下文	<code>/context</code>	查看当前上下文窗口的使用情况

`/clear` 的使用时机：这个命令比你想象的更重要。`/clear` 会清空当前会话的所有对话历史，回到一个干净的起点。Claude Code启动时读取的CLAUDE.md和项目文件不受影响。

什么时候该用？当你要开始一个和之前对话完全不同的任务时。比如你刚才在修一个API的bug，现在想让Claude帮你写一个新的前端组件。如果不clear，Claude的上下文里还残留着大量关于那个API bug的信息，会干扰它对新任务的理解。

`/compact` 和 `/btw` 的妙用：`/compact` 不是清空对话，而是让Claude把当前对话压缩成一个摘要。适合在一个长会话中途使用：你和Claude已经讨论了很多，上下文太长影响了性能，但你不想丢掉讨论的结论。`/btw` 是一个容易被忽略但非常实用的命令。它开启一个「侧链」对话：你可以问Claude一个和当前任务不相关的问题，问完之后侧链结束，主对话的上下文不受影响。

Shrivu特别提醒：不要依赖 `/compact`（自动压缩）。它是不透明的、容易出错的。需要重启时用 `/clear`，不要用 `/compact`。

六个坑，你大概率会踩

聊聊使用 Claude Code 时最常见的错误。这些坑我自己踩过，官方 Best Practices 也反复提，社区里更是老生常谈。

- 坑1：一个会话什么都塞。修bug、加功能、重构代码、写文档，全在一个会话里做。上下文被塞满，Claude对每个任务的理解都很浅。一个会话聚焦一个任务。做完就 `/clear`，或者开新终端窗口。
- 坑2：反复纠正，越改越偏。Claude做错了一步，你纠正；改了又错另一个地方，再纠正；第三次还是不对。你花在纠正上的时间比自己动手还多。纠正两次不行，果断 `/clear` 重来。重新描述需求，这次说得更具体。
- 坑3：看着像对的就接受了。Claude写了一大堆代码，输出看着挺合理，你就接受了，没实际跑一下。过几天发现边界情况的bug。每一轮改动都实际运行一次。「代码看起来对」和「代码是对的」差距可能很大。
- 坑4：过度微操。Claude每写一个文件你都要看、每改一行代码你都要评论。结果你和Claude都很慢，而且你其实在用Claude Code做传统编程。关注结果。让Claude把一个完整任务做完，看最终输出是否符合预期。中间过程除非明显跑偏，不用管。
- 坑5：需求模糊，然后怪Claude不懂你。「帮我优化一下这个代码」「让这个页面好看点」。Claude只能猜，而它猜的方向很可能不是你想要的。给具体的、可验证的需求：「把这个API的响应时间从2秒优化到500ms以内，瓶颈在数据库查询，考虑加缓存或优化SQL」。越具体，输出越接近预期。
- 坑6：不写CLAUDE.md。项目根目录没有CLAUDE.md，或者有但从不更新。每次新会话都要重新解释项目背景、代码规范、技术选型。这个太重要了，整整一章来讲。翻到 § 05。

这一章的核心：Plan模式想清楚再动手，Auto模式减少审批疲劳，`/permissions`精细控权限，Git集成管版本，Computer Use和Voice Mode扩展交互边界，会话管理保持上下文干净。这六个 workflow 覆盖95%的日常场景。

§ 05 CLAUDE.md: 给AI一张地图

CLAUDE.md — The Map You Draw for Your AI

Claude Code每次对话开始时会自动读取CLAUDE.md。这个文件不是说明书，它更像一份契约。你和AI之间关于怎么干活的约定，就写在这一个文件里。

为什么它是最重要的文件

用Claude Code写代码，你会接触到很多配置文件。package.json、tsconfig.json、.eslintrc ...但有一个文件的重要性超过它们加起来。

CLAUDE.md。原因很简单：Claude Code每次启动新会话，第一件事就是读这个文件。项目结构、代码风格、测试命令、常见陷阱，Claude都从这里了解。没有它，Claude就像空降到陌生代码库的新同事，什么都得从头摸索。有了它，它一进来就知道规矩。

Shrivu Shankar（Abnormal AI的AI战略VP，团队每月消耗数十亿tokens做代码生成）说得很直白：在有效使用Claude Code时，代码库中最重要的文件就是根目录的CLAUDE.md。这个文件是agent的「宪法」，是它了解你的特定代码库如何工作的主要真相来源。

他用了「宪法」这个词。宪法的特点是短、原则性强、不处理细节。这个类比很精确。

从护栏开始，别写手册

新手写CLAUDE.md最常犯的错：试图写一本百科全书。把每个函数的用法、每个文件的作用、每个API的参数都塞进去。写了几千行，Claude光读这个文件就吃掉大量上下文，真正干活的空间反而被挤小了。

Boris（Claude Code的创建者）团队的CLAUDE.md只有大约2500 tokens，大概100行。管理Claude Code这个产品本身的核心规则文件，就这么短。

[核心建议]你的CLAUDE.md应该从小开始，基于Claude做错的事情来记录。不要试图预先写一本完整手册，而是每次Claude犯错，就加一条规则。这就是「从护栏开始」的意思。这个方法好在哪？规则文件永远精准，因为每条都对应一个真实踩过的坑。文件也天然保持精简，因为你只记录真正出过问题的地方。

Boris在他的使用技巧中还提到一个飞轮效应：Claude犯错 → 记录到CLAUDE.md → 下次不再犯 → 错误率持续降低。他在代码审查时甚至会在同事的PR上@claude，让它自动把某条规则加到CLAUDE.md里。团队共享一个CLAUDE.md文件，check进git，每周都有人贡献。这个文件是活的，不是写完就放那不管的。

Boris的原话：「Claude非常擅长为自己编写规则。」你告诉它犯了什么错，它自己就能写出精确的规则防止下次再犯。

CLAUDE.md到底该写什么

这可能是最实用的部分。判断标准就一条：Claude自己能从代码里读出来的，不要写；Claude猜不到的，必须写。

该写	不该写
Claude猜不到的Bash命令（如自定义构建脚本）	Claude读代码就能知道的事（如「这是一个React项目」）
与默认不同的代码风格偏好	标准语言规范（Claude已经知道）
测试命令和偏好的测试框架	详细API文档（给链接，不要全文粘贴）
项目架构决策和背景	频繁变化的信息（每次都要改的东西不适合放这里）
开发环境的坑（如特殊的环境变量）	文件逐一描述（Claude会自己看文件树）
常见陷阱和修复方式	「写整洁代码」「遵循最佳实践」这种废话

Shrivu补充了几个常见反模式：

- 不要用 @ 引用大文档。在CLAUDE.md里 @ 一个长文件，它会在每次会话开始时被完整嵌入，白白吃掉上下文。正确做法是提到路径，告诉Claude什么情况下去读。比如：「遇到 FooBarError时，参阅 docs/troubleshooting.md 了解故障排除步骤。」
- 不要只写「永远不要做X」。当Claude觉得必须做X时它会卡住。永远提供替代方案：「不要用 --foo-bar 标志，改用 --baz 。」

- 把CLAUDE.md当作简化代码库的强制函数。如果某个CLI命令复杂到需要在CLAUDE.md里写几段话来解释，那说明这个命令本身需要简化。写一个bash包装器，用清晰的API，然后在CLAUDE.md里只记录那个包装器。

层级结构

CLAUDE.md不只是一个文件，而是一套层级系统。Claude Code会自动按顺序读取多个位置的CLAUDE.md：

```
~/ .claude/CLAUDE.md      ← 全局级：所有项目共用的偏好
./CLAUDE.md               ← 项目级：检入git，与团队共享
./src/CLAUDE.md           ← 子目录级：monorepo中特定模块的规则
./src/api/CLAUDE.md       ← 更深层子目录
```

全局级 `~/ .claude/CLAUDE.md`：放你个人的通用偏好。比如：优先用TypeScript、测试框架偏好Jest、commit message用英文。这些规则在所有项目中生效，不需要每个项目都写一遍。

项目级 `./CLAUDE.md`：放项目特有的规则。这个文件应该检入git，团队成员共享。Boris团队就是这么做的。代码风格、架构约束、测试命令、常见陷阱，全在这里。

子目录级：monorepo场景下特别有用。前端目录放前端的规则，后端目录放后端的规则，互不干扰。Claude进入某个目录时会自动加载对应的CLAUDE.md。

还有一个 `@` 引用语法，可以在CLAUDE.md中导入其他文件：

```
# CLAUDE.md
@docs/coding-standards.md
@docs/api-conventions.md
```

[注意]被 `@` 引用的文件会完整嵌入上下文。只引用那些真正每次都需要的短文件。

一个真实的好CLAUDE.md长什么样

下面是一个简洁精炼的项目级CLAUDE.md示例。注意它有多短：

```
# MyApp

## 架构
- Next.js 15 + TypeScript + Tailwind CSS
- 数据库: PostgreSQL + Drizzle ORM
- 认证: Better Auth
- 状态管理: Zustand (不要用Redux)

## 开发命令
- 启动开发服务器: pnpm dev
- 跑测试: pnpm test (Jest + React Testing Library)
- 类型检查: pnpm typecheck
- Lint: pnpm lint

## 代码风格
- 组件用函数式, 不用class
- 样式用Tailwind, 不要写CSS文件
- 数据获取用server component, 不用useEffect
- 错误处理用error.tsx边界, 不用try-catch包裹组件

## 常见陷阱
- Drizzle迁移后必须跑 pnpm db:generate, 否则类型不同步
- 环境变量改了之后要重启dev server
- better-auth的session检查在middleware中, 不要在页面组件里重复检查

## 不要做
- 不要安装新依赖除非我明确同意
- 不要修改 drizzle.config.ts
- 不要在client component中直接调数据库
```

整个文件不到300字。但每一行都有价值：要么是Claude猜不到的命令，要么是踩过坑的经验。没有一句废话。

[注意]不要把这个示例直接复制过去。好的CLAUDE.md是从你自己的项目中「长出来」的。空文件开始，Claude犯一次错就加一条，三个月后那个文件就是你的定制护栏。

Auto Memory: Claude自己记住的东西

除了你手写的CLAUDE.md，Claude Code还有一个自动记忆系统。当你在对话中纠正Claude的行为，比如「以后commit message都用英文」「测试文件放在 __tests__ 目录」，

Claude会自动把这些偏好保存下来。下次对话它就记住了，不需要你再说一遍，也不需要你手动写进CLAUDE.md。

这些记忆存储在 `~/claude/projects/<项目>/memory/` 目录下，以MEMORY.md为入口文件，和CLAUDE.md并行工作。区别是：手写CLAUDE.md适合团队共享的规则、检入git、你主动维护、结构化有组织；Auto Memory适合个人偏好、存在本地、Claude自动维护、零散按时间累积。两者配合使用效果最好。团队规则写在项目CLAUDE.md里，个人习惯让Auto Memory自动处理。

迭代飞轮：越用越好的系统

回到开头说的飞轮。这不只是个比喻，它就是Claude Code用户的真实体验曲线。

Mitchell Hashimoto（HashiCorp联合创始人，Terraform的创造者）描述过一模一样的过程。他给Ghostty搭建AI workflow时，配置文件里的每一行都对应agent过去犯过的一次错。文件是活的，一直在长。

这个过程是这样的：第一周空文件，你只写了基本的项目架构和开发命令，Claude犯很多错。第二周护栏初现，你把Claude犯过的错一条条记下来，错误率开始下降。第一个月飞轮启动，CLAUDE.md有了20-30条规则，都是真实的坑，Claude的输出质量明显提升，你需要纠正的次数越来越少。之后持续迭代，偶尔加新规则，偶尔删掉过时的，文件保持精简但高度定制。

这就是为什么说CLAUDE.md是最重要的文件。每一条规则背后都是一次真实踩过的坑，每次迭代都让Claude更懂你的项目。

[一句话总结]CLAUDE.md从空文件开始，每次犯错加一条，保持精简（Boris团队只用了约2500 tokens），检入git与团队共享。用三个月养出来的那个文件，是你最有价值的AI资产。

§ 06 进阶对话技巧

Advanced Prompting & Context Engineering

Claude Code不是搜索引擎，你不需要精心雕琢关键词。但怎么跟它说话，确实会影响输出质量。这一章聊的都是实战中真正管用的对话策略，不讲理论。

怎么说话Claude才听得懂

很多人第一次用Claude Code，会写「帮我做一个用户管理系统」。Claude会做，但做出来的东西大概率不是你想要的。信息太少，它只能猜。

官方Best Practices总结了三条原则，我觉得确实管用：

- 具体化：指定文件、场景、偏好。不要说「做个登录功能」，要说「在 `src/auth/` 目录下新增 Google OAuth登录，用Better Auth库，参考现有的GitHub登录实现方式」。文件路径、技术选型、参考模式，越具体Claude越知道往哪走。
- 指向已有模式：「照着这个做」。你项目里已经有一个UserWidget写得很好？直接告诉 Claude：「看 `src/components/UserWidget.tsx` 的实现方式，照着做一个CalendarWidget」。Claude读代码的能力极强，给它一个范本比写十行描述有效。
- 描述症状，不要猜原因。遇到bug别说「token刷新逻辑有问题」（除非你确认了），说「用户在session超时后登录失败，请检查 `src/auth/` 下的token刷新流程」。Claude能看到全部代码，让它自己定位原因比你猜更靠谱。

看几个Before/After对比就明白了：

不推荐	推荐
帮我加个搜索功能	在 src/components/Header.tsx 的导航栏中添加搜索框，用 Fuse.js 做模糊搜索，搜索范围是 posts 数组，参考现有的 FilterDropdown 组件的样式
接口报错了，帮我看	POST /api/orders 在 quantity > 100 时返回500，检查 src/api/orders.ts 的输入验证和数据库写入逻辑
优化一下性能	首页加载需要4秒，主要瓶颈在Dashboard组件，它一次获取了所有用户数据。改成分页加载，每页20条

Context Engineering：信息不是越多越好

后面第十章会详细聊Harness Engineering的三层架构：Prompt、Context、Harness。这一节先聊Context。

Context不只是你打的那句话。CLAUDE.md的内容、Claude读过的文件、你粘贴的截图、对话历史，全部加起来都是Context。

直觉上你可能觉得：给Claude的信息越多越好吧？恰恰相反。Anthropic工程团队发现，上下文太多，模型表现反而变差。它会在海量信息中迷失，做出混乱的决策。Shrivu建议定期用 /context 命令看看上下文窗口的使用情况。他在monorepo里测过，一个新会话光加载基础配置就吃掉约20k tokens，剩下180k才是干活的空间。

[核心建议]上下文管理的核心原则：不是给所有信息，而是给对的信息。让Claude看到它解决当前问题需要的上下文，而不是整个项目的百科全书。

你可以通过几种方式主动管理上下文：

- @ 引用文件：用 @src/utils/auth.ts 告诉Claude去读某个特定文件
- 粘贴截图：UI问题直接截图粘贴，比文字描述准确10倍
- Pipe数据： cat error.log | claude 直接把日志喂给Claude
- 给URL：Claude可以读取网页内容，给它API文档的链接比复制粘贴更好

让Claude采访你

当你要做一个比较大的功能（比如从零搭建一个支付系统），不要一上来就写需求文档。先对Claude说：「我想做一个支付功能，在动手之前，先采访我，问清楚所有你需要知道的事情。」

Claude会开始问你一系列问题：支持哪些支付方式？需要处理退款吗？并发量预估多少？需要支持webhook回调吗？用什么货币？

这些问题中，至少有一半是你自己没考虑过的。Claude帮你做了需求分析师的工作。

采访结束后，让Claude把答案整理成一份Spec（规格文档）。然后关键来了：开一个全新的会话，把Spec喂给新的Claude，让它执行。

为什么要开新会话？因为采访过程中积累的对话历史已经很长了，占了大量上下文。新会话从一份干净的Spec开始，Claude能更专注地执行，不会被中间讨论过程干扰。采访阶段 → 生成Spec → 新会话执行。

把Claude当高级工程师提问

很多人只把Claude Code当写代码的工具。其实它同样是一个极好的代码库导航员。你可以直接问它：「项目里的logging怎么工作的？」「怎么新建一个API endpoint？」「这个 useAuth hook的调用链是什么？」「src/lib/db.ts 和 src/utils/database.ts 有什么区别？为什么有两个？」

Claude会读相关代码，然后给你一个结构化的解释。比读文档快，比问同事方便，尤其是刚接手一个新项目的时候。

Boris团队的人就是这么用的。新成员入职不是先读一堆wiki，而是直接问Claude Code。它对代码库的理解往往比过时的文档更准确。

[Onboarding加速器]加入一个新项目后，先花10分钟问Claude Code：「这个项目的架构是什么？核心模块有哪些？数据流是怎么走的？」你会省下至少半天翻文档的时间。

多轮对话策略

和Claude Code的对话不是一次性的。你经常需要在多轮对话中逐步推进一个任务。这里有几个经过实战验证的策略：

紧密反馈循环：别等Claude写完500行代码再看结果。发现方向偏了，立刻纠正。越早纠正成本越低。Claude写了10行时你说「不对，换个方式」，成本几乎为零。写完整个功能再推倒重来，浪费的是tokens和时间。

两次纠正不行，换条路：纠正了两次Claude还是不按你的意思来？别继续纠正了。/clear 清掉上下文，用一个更好的初始prompt重新开始。在一个已经跑偏的对话里纠缠，往往越绕越远。

换任务就清上下文：写完一个组件后要去改数据库schema？/clear。不同任务有不同的上下文需求，把前一个任务的对话历史带进新任务只会增加噪音。Shrivu推荐的做法：/clear 之后跑一个自定义的 /catchup 命令，让Claude读取当前git分支中的变更来恢复上下文。

用subagent做调研：有时你需要Claude先调研再动手：「看看这个库怎么用」「分析一下竞品的实现方式」。这些调研任务可以用subagent来做，调研结果返回主会话，中间思考过程不会污染主上下文。

Effort级别：别省这个钱

Claude Code有四个effort级别：Low、Medium、High、Max。控制Claude执行任务时投入多少推理资源。

级别	适合场景	特点
Low	简单的格式化、重命名	快，但容易犯低级错误
Medium	日常开发任务	比默认更轻量
High	复杂功能开发、调试	默认级别，Boris也用这个
Max	极端复杂的架构决策	无限推理token，最慢最深

High本身就是默认值，Boris的做法是从不把它调低。理由和他坚持用Opus一样：Claude想得更深，需要返工的次数更少，总体效率反而更高。很多人觉得「这个任务简单，调到Low省点时间」。但Low做错了，你纠正它花的时间可能比直接用High做对还长。

[核心建议]如果你用的是Max计划，High已经是默认值，不需要额外调整。别为了省几秒钟把effort调低，修低级错误花的时间远不止几秒。

三个提问原则

和Claude Code对话的核心其实就三个字。

具体。文件名、行号、函数名、期望行为，能给就给。越具体的指令，越精确的输出。

指向。你代码库里一定有写得好的部分。把它们当参考范本指给Claude看。「像那个一样做」比「做一个漂亮的」有效100倍。

克制。一次只做一件事。任务大就分步来，每步确认结果后再下一步。一条消息里塞三个不相关的需求，Claude大概率只做好其中一个。

我觉得一个好的心态是：把Claude当成一个非常聪明但刚入职的同事。能力很强，但不了解你项目的历史和惯例。你给的上下文越精准，它的产出越接近预期。

这一章的核心：好的对话靠的不是花哨的prompt，而是精准的上下文。具体化需求，让Claude采访你来补盲点，用 /clear 保持干净，别把effort调低。说到底，最有效的进阶是在实践中积累和Claude协作的直觉，不是学更多prompt技巧。

§ 07 扩展能力：Skills、Hooks与MCP

Extensions: Skills, Hooks & MCP

用到后面你会发现，Claude Code真正的价值不是它本身有多强，而是你能在它身上接多少东西。Skills、Hooks、MCP三种扩展机制，让它从一个终端工具变成一个可以无限生长的工作台。

为什么需要扩展

我一开始以为Claude Code装好就完事了。后来发现自己总在重复同样的话：每次提交代码前念叨一遍「先跑个lint」、每次新建组件要交代一遍项目规范、每次查数据要手动复制SQL结果贴给Claude。

这种重复一旦超过三次，就该想办法自动化了。Claude Code提供了三种扩展机制，各自解决不同层面的问题：

机制	本质	确定性	适用场景
Skills	Markdown指令包	高但非100%（advisory）	领域知识、可复用 workflow
Hooks	Shell脚本钩子	100%确定执行	格式化、lint、安全检查
MCP	外部工具连接器	100%	数据库、API、第三方服务

三者的关系：Skills教Claude怎么做事，Hooks在关键节点自动执行检查，MCP把外面的世界接进来。我个人用得最多的是Skills，几乎每天都在加新的。

Skills：我觉得最值得先学的

Skills是最容易上手的扩展方式。原理简单到有点不像话：在 `.claude/skills/` 目录下创建一个文件夹，放一个 `SKILL.md` 文件，Claude就会根据上下文自动加载里面的指令。

```
.claude/skills/
├─ react-component/
│   └─ SKILL.md      # 创建React组件的规范和步骤
├─ fix-issue/
│   └─ SKILL.md      # 修bug的标准流程
└─ deploy-preview/
    └─ SKILL.md      # 部署预览环境的步骤
```

用 `/skill-name` 可以手动调用某个skill，Claude也会根据对话内容自动判断要不要加载。比如你说「帮我创建一个新的React组件」，Claude会自动加载 `react-component` skill里的规范。

两种类型的Skills：知识型告诉Claude「这个项目里的事情应该怎么做」。比如API规范、编码风格、项目约定。这类skill更像文档，Claude读完后会按里面的规则办事。 workflow型告诉Claude「遇到这种任务按什么步骤执行」。比如 `/fix-issue`（修bug的标准流程）、`/review-pr`（代码审查流程）。这类skill更像SOP，有明确的步骤和检查点。

Boris有一个挺实用的判断标准：如果一件事你每天做超过一次，就应该把它变成skill或command。我自己更激进一点，超过两次我就写了。

实际案例：创建 `/techdebt` 命令。把「发现技术债 → 评估影响 → 创建issue → 关联到sprint」这个流程写成skill。以后发现技术债时，直接输入 `/techdebt`，Claude会自动走完整个流程，包括评估优先级、创建GitHub issue、加上正确的标签。

workflow型skill的关键配置：workflow型skill通常会执行有副作用的操作（比如创建issue、发消息、部署）。为了防止Claude在不合适的时候自动触发，可以在 `SKILL.md` 的front matter中加一行 `disable-model-invocation: true`。加了这个配置后，skill只能通过 `/skill-name` 手动调用，Claude不会自作主张地触发它。

安装别人的skill：Skills可以共享。Boris自己整理了一套高频使用的skills，你可以一行命令安装。社区也在不断贡献新的skills，你可以在Claude Code里用 `/plugin` 浏览市场。

[核心建议]写skill的最佳实践：从你最常对Claude说的那句话开始。如果你总是在提交前说「先跑一下测试，格式化一下代码，然后commit」，那这就是一个skill的雏形。把这些步骤写进SKILL.md，下次一个斜杠命令就搞定。

Hooks：不是建议，是强制

Skills有一个天然的局限：它本质上是对Claude的「建议」。Claude会尽量遵守，但遵从率不是100%，尤其在长对话后期，它可能就忘了。大多数场景下够用，但有些事情你需要100%的确定性。

比如我自己踩过一个坑：在CLAUDE.md里写了「每次编辑文件后跑eslint格式化」，前几轮对话都好好的，聊到后面上下文一压缩，这条规则就被吃掉了。

Hooks就是为了解决这个问题。Hooks vs CLAUDE.md：本质区别。很多人会把规则写在CLAUDE.md里：「每次修改文件后请运行 `npx eslint --fix`」。这在大多数时候有效，但Claude偶尔会忘记，尤其在长对话、上下文被压缩之后。

CLAUDE.md是建议，Hooks是强制执行。CLAUDE.md通过自然语言影响Claude的行为；Hooks是Claude Code平台层面的机制，在特定生命周期节点触发Shell脚本，Claude无法跳过或忽略。

生命周期钩子：Hooks支持多个触发时机：

钩子	触发时机	典型用途
PreToolUse	Claude调用工具之前	拦截危险操作
PostToolUse	Claude调用工具之后	自动格式化、自动测试
PermissionRequest	需要用户授权时	自动批准低风险操作
Stop	Claude完成回合时	推动继续执行
PostCompact	上下文压缩后	注入关键指令防遗忘
PermissionDenied	Auto模式分类器拒绝操作后	记录被拒操作、通知用户、触发替代方案
SessionStart	新会话启动时	初始化环境变量、加载配置

实用案例：

- 案例1: 自动格式化。每次Claude编辑文件后自动跑eslint, 不依赖Claude「记住」要格式化。
- 案例2: 自动批准低风险操作。用PermissionRequest hook把权限请求路由到一个脚本, 脚本判断操作类型, 低风险的自动批准, 高风险的仍然弹出确认。
- 案例3: 上下文压缩后注入关键指令。长对话中Claude会压缩上下文来节省token。压缩后, 一些早期的重要指令可能丢失。PostCompact hook可以在压缩发生后自动把关键规则重新注入, 确保Claude不会「失忆」。
- 案例4: 推动Claude继续。有时候Claude会在一个复杂任务中途停下来问「要继续吗?」。Stop hook可以检测这种情况, 自动让Claude继续执行, 适合无人值守的批处理场景。

[核心建议]你不需要自己从零写hooks。直接告诉Claude: 「Write a hook that runs eslint after every file edit」, 它会帮你生成配置并写入 .claude/settings.json。

MCP: 让Claude看到外面的世界

Skills教Claude知识, Hooks保证执行确定性, 但它们都在Claude Code的内部世界运作。如果你需要Claude直接查数据库、调API、读取设计稿, 就需要MCP。

MCP (Model Context Protocol) 是Anthropic推出的开放标准, 让AI工具能连接外部数据源和服务。把它想象成Claude Code的USB接口: 插上不同的MCP服务器, Claude就获得了对应的能力。

```
# 添加一个MCP服务器
claude mcp add slack -n npx -y @modelcontextprotocol/server-slack
# 查看已安装的MCP
claude mcp list
```

添加后, MCP服务器的能力会以「工具」的形式暴露给Claude。比如安装了Slack MCP后, Claude就可以搜索Slack消息、发送消息、创建频道。

实用MCP推荐:

MCP	能力	适用场景
Slack MCP	搜索/发送消息	让Claude自动同步进度、回复问题
数据库MCP	直接查询数据库	不用手动复制SQL结果
Figma MCP	读取设计稿	把设计直接转成代码
Sentry MCP	获取错误日志	Claude自动定位线上bug
GitHub MCP	操作仓库/Issue/PR	自动化项目管理

Boris有一个经典用法：他给Claude Code接上Slack MCP，当有人在Slack里报告bug时，Claude会自动读取bug描述、找到相关代码、尝试修复、提交PR，然后在Slack里回复「已修复，PR链接在这里」。整个过程不需要人工介入。

MCP配置文件：MCP的配置存在项目根目录的 `.mcp.json` 中，可以跟代码一起提交到Git仓库，这样团队成员clone项目后就自动获得相同的MCP配置。

[注意] MCP服务器会获得对外部服务的访问权限。添加新的MCP前，确认你理解它会访问哪些数据。敏感token不要硬编码在 `.mcp.json` 里，用环境变量引用。

Plugins：打包好的扩展包

Skills、Hooks、MCP可以各自独立使用，但组合起来才真正厉害。Plugins就是这种组合的打包形式。在Claude Code里输入 `/plugin`，你可以浏览一个不断增长的插件市场。每个Plugin可能包含skills、hooks、agents、MCP配置中的一种或多种，一键安装就全部配置好。

比如一个「代码智能」Plugin，可能同时包含：一个skill（告诉Claude如何利用符号导航理解代码结构）、一个hook（编辑后自动运行类型检查）、一个MCP（连接语言服务器获取精确的符号信息）。这三者配合，让Claude在理解和修改代码时更准确，而你只需要一次安装。

Slash Commands: 带预计算的快捷入口

除了 /skill-name 调用skill，还有一种更灵活的方式：Slash Commands。Commands存在 .claude/commands/ 目录中。和skills不同的是，commands可以包含内联的Bash脚本来预计算信息。在Claude读到prompt之前，command先跑一些shell命令，把结果嵌入进去。

```
# .claude/commands/commit-push-pr.md
帮我完成以下操作：
1. 查看当前的git diff:
   git diff --stat
2. 生成commit message并提交
3. 推送到远程分支
4. 创建Pull Request, 标题基于commit内容
```

输入 /commit-push-pr，Claude就会按照这个流程自动执行。因为command文件存在 .claude/commands/ 里，它会随Git一起提交，团队成员都能用。

Skills vs Commands选择指南：两者有重叠，但定位不同。Skills更像「知识和能力」，Claude根据上下文自动应用或手动调用；Commands更像「宏」，包含预计算步骤，强调执行流程。经验法则：如果需要Claude「知道什么」，用skill；如果需要Claude「做一串事」，用command。

三种扩展机制的协作

实际项目中，三种机制经常协同工作。一个完整的例子：

假设你的团队有这样一个 workflow：收到bug报告 → 定位问题 → 修复 → 跑测试 → 提交PR → 通知相关人。

MCP (Slack) 让Claude收到bug报告并能回复修复结果 → Skill (fix-issue) 指导Claude按标准流程定位和修复问题 → Hook (PostToolUse) 确保每次修改后都自动跑测试和格式化 → Slack MCP通知修复结果。

单独用任何一个都有价值，组合起来就是一个完整的自动化bug修复流水线。

[核心建议]不要想着一上来就搭一套完整的扩展体系。从你最痛的那一个重复操作开始：总在口述规则？写个skill。总忘记跑lint？加个hook。总在手动倒腾数据？接个MCP。一个一个加，慢慢你的Claude Code就变成了一个为你量身定制的工作台。

§ 08 多Agent协作

Multi-Agent Collaboration

前面七章，你学到的是如何用好「一个」Claude。但真正的效率质变发生在你开始用「多个」Claude同时工作的时候。这一章聊多Agent协作的几种模式，以及如何在Claude Code里实现它们。

为什么一个不够

单个Claude Code会话的上下文窗口是200k tokens（1M token beta已推出，约750k英文词），听起来很多，但一个中等规模项目的基础代码就能把上下文吃掉大半。剩下可用的tokens，还要扣除对话历史、CLAUDE.md、MCP工具调用的开销。

Monorepo里问题更严重。前后端混在一个仓库里，Claude每次都要同时处理React组件和数据库schema，注意力被分散，输出质量下降。

多Agent的核心思路：把任务拆成多个并行的、独立的子任务，每个Agent只处理一小部分。就像把一个大项目拆成几个小团队，每个团队专注自己的模块。

三种模式：并行（同时开多个Claude Code实例，各自做不同任务）、串联（一个Agent的成果传给下一个Agent继续处理）、Subagent（在一个Claude Code会话中临时创建短命的子任务）。

模式一：并行多终端

最原始也最灵活。在多个终端窗口各自运行 `claude`，每个处理不同模块。

```
# 终端1: 修bug
claude

# 终端2: 新功能
claude --worktree

# 终端3: 写文档
claude --worktree
```

Git Worktree是并行工作的关键。--worktree 让 Claude Code 自动创建一个新的 git worktree，在隔离的目录中工作。每个worktree有自己的分支和工作目录，互不干扰。

并行模式的具体操作：先创建worktree，然后启动Claude Code：

```
# 创建worktree
git worktree add ../feature-auth auth-branch
cd ../feature-auth
claude
```

规划大任务时，先告诉Claude：「这个任务比较大，先把它拆成几个子任务，用subagent分别处理。」Claude会自动创建subagent，每个处理一个独立子任务，最后合并结果。

模式二：Subagent会话

如果你不想开多个终端窗口，Claude Code也支持在一个会话中创建子任务（subagent）。在主对话中，Claude可以自己决定什么时候创建subagent，给它一个独立的上下文去执行特定任务。

启动方式有两种：隐式subagent（Claude自动决定何时拆分任务）和显式subagent（用户通过 /agents 命令创建子任务，再跟每个子Agent协作）。

Subagent的使用场景：需要调研再动手（先用subagent调研技术方案，主agent等待结果后再执行）；需要并行处理（同时让多个subagent处理不同文件）；上下文隔离（主agent需要保持当前任务上下文不变，把不相关的子任务派给subagent）。

[核心建议]多Agent协作的关键不在于「有多少个Claude在跑」，而在于「任务拆分是否合理」。拆得太碎，Agent之间通信开销大；拆得太粗，又失去了并行的好处。经验法则：一个Agent处理一个独立的、自包含的模块或任务。

模式三：Team Plan与Agent Teams

2026年2月推出的Agent Teams（Research Preview，Team/Enterprise Plan可用）是Claude Code原生支持的多Agent协作方案。你不再需要开多个终端窗口或手动协调子任务，Claude Code自己就能创建和管理一个Agent团队。

Agent Teams的核心特点：自动创建专门的Agent子角色（Code Agent负责实现、Fix Agent负责修复、Write Agent负责文档、Analyze Agent负责分析）；任务拆分和进度追踪自动化；Claude自动分配任务、追踪进度、合并结果。

一句话启动：「创建一个团队来完成：重构认证模块。拆分任务，追踪进度。」Claude会自动创建需要的Agent角色，分配任务，并在过程中汇报进度。

[注意] Agent Teams目前还是Research Preview，需要通过feature flag启用（`claude --agent-teams`）。功能可能在实际使用中有所变化，建议关注官方更新。

模式四：云端Agent队列

除了Team Plan的Agent Teams，Claude Code还提供更轻量的云端多Agent方案：

- Cowork (Jan 2026)：独立的云端环境，每个任务分配一个Agent处理。适合需要云端环境的大型任务。Cowork环境有独立的文件系统，Claude可以在里面做任何事，不影响你本地代码。适合需要完整环境的测试、原型验证、破坏性操作。
- UltraPlan（云端规划）：类似Plan模式，但规划过程在云端运行，可以处理更大的上下文。先让云端Agent分析整个项目，然后本地Agent执行具体任务。UltraPlan可以访问1M token上下文，适合分析大型代码库。启动方式：`claude --ultra-plan`。
- Remote Control (Research Preview)：从一台机器启动Claude Code，在另一台机器上实时查看和操作。适合跨设备工作：在办公室的台式机上跑任务，回家用笔记本继续查看进度。或者在不同架构的机器间协作。启动方式：`claude --remote`。

多Agent的最佳实践

任务拆分的艺术：一个好的拆分应该满足三个条件。独立（每个任务可以独立完成，不依赖其他任务的中间结果）、自包含（一个任务有明确的输入和输出）、粒度适中（太小导致通信开销大；太大失去并行的意义）。

上下文管理：每个Agent的上下文是独立的。如果一个Agent需要知道另一个Agent做了什么，你需要显式传递信息：用CLAUDE.md共享项目规范、用 `/compact` 压缩对话后传递关键信息、用文件系统共享结果（一个Agent写入文件，另一个读取）。

错误隔离：一个Agent出错不会影响其他Agent。但要注意：如果多个Agent同时修改同一个文件，Git会报冲突。用worktree可以完全避免这个问题。

[核心建议]从简单的并行开始。先在两个终端里各跑一个Claude Code，一个修bug一个写新功能。感受一下「多线程编程」的快感。熟悉了再尝试更复杂的多Agent模式。

§ 09 从零构建一个完整产品

Building a Full Product from Scratch

前面八章讲的是工具和方法。这一章我们把它们串起来，从零构建一个真实的SaaS产品。

项目概述

做一个「个人财务管理」的Web应用。功能包括：用户注册/登录（Better Auth）、收入/支出记录（CRUD）、数据可视化（图表）、导出报表（CSV/PDF）。技术栈：Next.js 15 + TypeScript + Tailwind CSS + shadcn/ui + PostgreSQL + Drizzle ORM。

Day 1: 项目骨架

打开终端，创建项目目录，启动Claude Code。直接说需求：

帮我创建一个个人财务管理Web应用。技术栈：

- Next.js 15 + TypeScript + Tailwind CSS
- 数据库：PostgreSQL + Drizzle ORM
- 认证：Better Auth
- UI组件：shadcn/ui
- 图表：recharts

功能需求：

1. 用户注册/登录（邮箱+密码）
2. 收入/支出记录（CRUD）
3. 数据可视化（月度收支饼图、趋势折线图）
4. 导出报表（CSV格式）

先给我架构方案，再逐步实现。

Claude会给出架构方案，包括目录结构、技术选型理由、开发顺序。审查方案，确认后开始执行。

第一天的目标：跑通开发环境、数据库连接、认证系统。

Day 2: 核心功能

继续推进，让Claude实现CRUD功能。不需要你写一行代码，只需要确认方案和验证结果。

关键提示：「按我们已经确定的架构，实现收支记录的CRUD。参考现有的auth模块的代码风格。」

Claude会自动创建API路由、数据库schema、前端页面。它会自己跑测试，验证CRUD功能正常。

第二天的目标：收支记录的增删改查、基础列表页面。

Day 3: 图表和导出

继续推进数据可视化和导出功能。

提示：「在Dashboard页面添加两个图表：月度收支饼图、最近6个月的趋势折线图。用recharts实现。同时添加导出CSV功能，可以按日期范围筛选后导出。」

Claude会读取现有的Dashboard代码，理解数据流，然后添加图表组件。它可能会自动安装recharts依赖，创建图表组件文件，更新页面引用。

第三天的目标：Dashboard有图表、能导出CSV。

Day 4: 优化和部署

最后一步：让Claude优化性能、修复发现的bug、准备部署。

提示：「优化一下性能：首页加载有点慢，看看能不能减少数据库查询次数。同时检查所有页面在移动端的表现。最后，准备部署到Vercel的配置。」

Claude会分析性能瓶颈，可能建议添加数据库索引、优化查询、添加加载状态。然后生成vercel.json配置文件。

第四天的目标：性能优化完成、移动端适配、可以一键部署。

实战经验

这个项目按每天4-6小时计算，实际上4天就能完成一个可用的MVP。以下是几个实战中的经验：

- 每完成一个功能就 /clear 。不要让前一天的对话历史干扰新任务。
- 遇到Claude犹豫不决（「你想用A还是B？」），直接替它决定。省下的时间比纠结选项更有价值。
- 复杂UI不要口述，画个草图拍照上传。Claude读图能力极强，一个草图顶十行描述。
- 部署前让Claude自己跑一遍完整流程：注册 → 登录 → 添加记录 → 看图表 → 导出。它会自己发现流程中的断点。

[核心建议]构建完整产品的关键不是「让Claude写多少代码」，而是「你给了多清晰的指令」。花5分钟写一个好的prompt，省下的可能是几个小时的重做。

§ 10 心智模型与持续进化

Mental Models & Continuous Evolution

工具会过时，但心智模型不会。这一章聊的不是具体功能，而是怎么用Claude Code才能持续获得竞争优势。

Harness Engineering的三层架构

Claude Code的使用深度可以分成三层。大多数人停在第一层，少数人摸到第二层，真正释放潜力的是第三层。

层级	你在做什么	效果
第一层: Prompt	每次对话精心措辞	单次对话质量高，但不可持续
第二层: Context	CLAUDE.md + 文件引用 + 截图	可复用的项目记忆，减少重复沟通
第三层: Harness	Skills + Hooks + MCP + Agent Teams	系统化扩展，形成自动化流水线

第一层靠你的prompt技巧，第二层靠你的项目配置，第三层靠你的系统搭建。

进阶路径：刚开始用的时候，重点在Prompt和Context。怎么跟Claude说话、怎么管理上下文、怎么写CLAUDE.md。用熟了之后，重点应该转向Harness。

Harness Engineering：从零开始搭建系统

第三部分的核心是搭建Harness（控制系统）：从一个空项目开始，先配置好扩展系统，再开始写功能代码。

Harness搭建的四步：

- Day 1: 搭骨架。写好CLAUDE.md，装上必要的MCP（数据库、API文档），配置好Hooks（格式化、lint）。

- Day 2: 写Skills。把项目中反复出现的任务写成skills。比如 /create-component、/add-api、/write-test。
- Day 3: 自动化。用Hooks把检查流程自动化。每次提交前跑测试、每次部署前验证、每次PR创建时自动生成描述。
- Day 4: 连接外部。把Slack、GitHub、CI/CD接到MCP，形成一个完整的自动化工作流。

到Day 4结束时，你拥有了一个完整的Harness系统：Claude Code不只是帮你写代码，它还帮你管理项目沟通、自动化检查、追踪进度。这才是一个真正的AI-native工作流。

[核心建议]不要急着写功能。先花1-2天把Harness搭好。初期的投入会在之后的每一天产生复利效应。

持续学习的方法

Claude Code的迭代速度极快，几乎每周都有新功能。保持跟进的方法：

- 定期查看官方Changelog。claude.ai/code/docs/changelog 有详细的版本更新记录。
- 加入社区。Reddit的r/ClaudeCode、Discord的Claude Code频道、GitHub Discussions，都是获取最新用法的好地方。
- 每周试一个新功能。不需要每个功能都深度使用，但至少了解它解决什么问题。Computer Use、Voice Mode、Agent Teams，每周选一个试试看。
- 更新你的Skills库。新功能发布后，思考「这个新能力能不能融入我的某个skill里？」比如Computer Use推出后，你的test skill可以更新为「跑测试 + 截图检查UI」。

[核心建议]最好的学习方式是教别人。把你学到的东西写成blog post、录成视频、在团队内部分享。教的过程中你会发现自己理解得不够深的地方。

长期竞争力

Agent式编程的普及速度远超大多数人预期。当「用AI写代码」变成标配，你的竞争力不再是「会不会用AI」，而是「用AI能做出什么别人做不出来的东西」。

几个判断标准：

- 你的Harness系统有多完善？别人还在逐句跟Claude对话时，你一套斜杠命令就搞定了。
- 你的CLAUDE.md有多精准？别人每次都要解释项目背景，你启动Claude它就已经知道了。
- 你的多Agent协作有多流畅？别人单线程工作时，你同时在推进三四个任务。
- 你的产品思维有多强？别人让Claude写函数，你让Claude做产品。

最后想说的是：Claude Code是一个放大器。它放大的不是你的编程能力，而是你的产品构建能力。一个有好想法但不会写代码的人，加上Claude Code，可以做出一个完整的SaaS产品。一个有好想法而且会写代码的人，加上Claude Code，可以做出十个。

区别在于：前者在跟Claude学习怎么做，后者在跟Claude合作怎么做更快。两者都能成功，但速度不一样。

这就是Agent式编程的核心：不是AI取代你，是你用AI取代那些重复、低效、不应该由人来做的事情。把时间花在真正创造价值的地方：定义问题、判断方向、验证假设。

这本书到这里就结束了。但你的旅程刚刚开始。去搭你的Harness，去建你的产品，去验证那些你一直觉得「等我学会编程再做」的想法。

现在就可以开始。

附录

Appendices

附录A Claude Code演进时间线

以下是Claude Code从发布到2026年4月的关键功能演进时间线，帮助你理解这个工具的迭代节奏和能力边界。

时间	版本/事件	关键变化
2025.02	研究预览版发布	Claude Code首次对外发布，支持终端CLI
2025.05	GA正式发布	推出Pro/Max付费方案，VS Code扩展上线
2025.06	v1.0.11	CLAUDE.md支持、auto-accept安全机制、1M context beta
2025.07	Summer Release	GitHub集成、复合Agent、代码执行工具
2025.08	Claude 4系列	Claude 4 Opus和Sonnet发布，Hooks早期支持
2025.09	Hooks GA	7种生命周期钩子正式发布
2025.10	Skills系统	.claude/skills/ 目录支持，Skills市场上线
2025.11	v2.0	大幅扩展功能集，UI改进，Claude 4.1模型
2025.12	Claude in Chrome	浏览器扩展Beta，1M context正式GA
2026.01	Cowork	云端独立环境，支持完整项目测试
2026.02	Plugins + Remote Control	插件市场、跨设备远程控制（Research Preview）
2026.02	Agent Teams	原生多Agent协作（Research Preview，Team Plan）
2026.03	Computer Use	AI操作桌面GUI（Research Preview，macOS only）
2026.03	Voice Mode	语音输入支持20种语言
2026.03	/loop	定时任务循环执行
2026.03	Claude 4.5	Sonnet 4.5发布，Opus 4.1被弃用
2026.04	Claude 4.6	Opus 4.6和Sonnet 4.6发布，Anthropic Web发布

几个值得注意的趋势：

- 从「文本Agent」到「多模态Agent」：Computer Use让Claude突破了纯文本边界，获得了GUI操作能力。
- 从「单机工具」到「云端协作」：Cowork、Remote Control、Agent Teams标志着Claude Code正在从个人工具向团队协作平台演进。
- 从「对话式」到「系统化」：Skills、Hooks、MCP、Plugins的推出，让用户可以把重复的对话模式固化为可复用的系统组件。

附录B 新版本功能速查

以下是截至2026年4月的Claude Code重要功能速查表，方便快速定位和查找。

功能	启动方式	适用计划	状态
Computer Use	零配置，Pro/Max自动可用	Pro/Max	Research Preview
Voice Mode	/voice	所有付费计划	已发布
Agent Teams	claude --agent-teams	Team/Enterprise	Research Preview
/loop 定时任务	/loop	所有计划	已发布
Remote Control	claude --remote	所有计划	Research Preview
Cowork	claude --cowork	所有计划	已发布
UltraPlan	claude --ultra-plan	所有计划	已发布
Claude in Chrome	安装Chrome扩展	所有付费计划	Beta
1M Token Context	自动可用	所有计划	已发布
Security Scanning	自动运行	Team/Enterprise	已发布
Code Execution Tool	自动可用	所有计划	已发布

附录C 推荐资源

官方资源：

- Anthropic官方文档：docs.anthropic.com — 最权威的功能说明和API参考
- Claude Code Changelog：claude.ai/code/docs/changelog — 每周更新，跟踪最新功能
- Anthropic博客：anthropic.com/news — 重大发布和深度技术文章

社区资源：

- Reddit r/ClaudeCode — 社区讨论、技巧分享、问题解答
- GitHub Anthropic Discussions — 官方支持的开发者讨论区
- Discord Claude Code频道 — 实时交流、快速问答

进阶学习：

- Boris Cherny的使用技巧 — Claude Code创建者的官方博客（boris.io）
- Mitchell Hashimoto的AI工作流 — HashiCorp创始人在Ghostty项目中使用Claude Code的深度实践
- Shrivu Shankar的Best Practices — Abnormal AI的VP分享的团队级使用经验

工具和扩展：

- MCP服务器列表：github.com/modelcontextprotocol/servers — 官方维护的MCP服务器集合
- Claude Code Plugins市场：在Claude Code中输入 /plugin 浏览
- Skills社区分享：reddit.com/r/ClaudeCode 和 Discord 频道的 #skills-sharing

联系作者：

- 主页：https://qingtengstudio.com/
- 署名：QingTeng·AI

本手册持续更新，获取最新版本请访问主页获取。