2026年4月 · 第2版

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

The Complete Guide to Claude Code — From Zero to Shipping Products

---

适用版本： Claude Code v2.1.88+

模型： Opus 4.6 / Sonnet 4.6 / Haiku 4.5

新增： Computer Use · Voice Mode · 架构深度解析

花叔

公众号「花叔」· B站「AI进化论-花生」

> 知识星球「AI编程：从入门到精通」专属内容

本手册基于Anthropic官方文档、Boris Cherny（Claude Code创建者）公开分享、DeepLearning.AI官方课程及Claude Code v2.1.88源码分析编写。所有操作细节以2026年3-4月最新资料为准。AI工具迭代极快，请结合官方文档验证。

本手册持续更新，获取最新版本请访问：[飞书文档]

目录

CONTENTS

自序　从不会写代码到被央视采访

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　核心工作流

§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

§09b　踩坑指南：AI编程的边界在哪

§10　心智模型与持续进化

Part 4: 实战项目

§11　实战项目：Chrome扩展

§12　实战项目：内容创作自动化

§13 实战项目：从零到App Store付费榜第一

§14 斜杠命令深度指南

练习与附录

练习 试试看

附录

附录A 51万行代码告诉我们的事

附录B 国内模型接入指南

附录C 完整命令参考

附录D 常见问题FAQ

附录E 术语表

附录F Agentic Coding趋势报告

附录G CLAUDE.md模板集

自序 从不会写代码到被央视采访

Preface

2024年8月，我在B站发了一系列视频，介绍自己这么个完全不懂代码的人，是怎么用AI做出各种产品的。评论区的反应很直接：

「你不懂编程你怎么解bug？」

「你不懂编程你怎么能上架App？」

「你这也能叫产品吗，你这最多算demo！」

我没办法反驳，因为那时候我确实只是在摸索。AI编程工具还很早期，Cursor刚开始有Agent模式，Claude Code还没发布。我能做的事情有限，做出来的东西也确实粗糙。

四个月后，2024年底，我女朋友随口说了一句「与其做屏幕手电筒，不如做补光卡片」。我去小红书搜了一下，发现大量女生在分享纯色图片用来自拍补光，好几篇笔记十几万赞。我意识到：需求已经被验证了，只是没人做成产品。

那天晚上我用Cursor花了大约1小时，做出了「小猫补光灯」的第一版。纯开发成本，几十块钱的API调用费。

上架后两天，小猫补光灯冲上了AppStore付费榜第一。小红书评论区涌进了上千条留言，因为App Store会显示开发者真名，女生们直接喊我本名「陈云飞」而不是网名，我就这么在小红书痛失了网名。

然后事情一件接一件。人民日报把我作为「手搓经济」的代表报道。央视新闻找到我，做了一期AI赋能「一人公司」的专题采访。采访当天，我在央视镜头前现场用Claude Code 10分钟手搓了一个小产品，也许是Claude Code的编码画面第一次出现在国内官媒上。

我说这些不是为了炫耀。我想说的是一个事实：**我从来没有写过一行代码。**

不是「以前会，后来不写了」。是从来没有。

小猫补光灯是AI写的。后来的小猫相册、鸿蒙版、小程序版，全是AI写的。我的公众号30万粉丝的运营、4本橙皮书的出版、B站和YouTube的内容制作，流程中70%以上的执行工作都是AI完成的。我2023年3月从互联网大厂离职，到现在将近3年，一个人做产品、写内容、做视频、教课程、写书，这些事情在以前可能需要十几个人的团队。

Claude Code是我目前的主力工具。

我写这本书，不是因为我是Claude Code的专家。我不是程序员，不懂TypeScript，看不懂Claude Code的源码。我写这本书，是因为我可能是中文世界里**用Claude Code做了最多「非编程」事情的人**。我用它写书、做调研、管理文件、生成配图、审校文章、发布到飞书、分发到社交媒体。我装了60多个Skills，建了完整的Harness工具链，每天的工作有大半在终端里完成。

这本书的读者画像就是一年半前的我：聪明，有想法，但不会写代码，也不确定AI编程到底是不是真的能用。

我可以非常确定地告诉你：能用。不只是能用，而是能用来做出真正有人用、有人付费、上排行榜、被央视报道的东西。

但我也要说一句不那么热血的话，这是我在央视采访里说过的：**绝大多数手搓产品会无声消亡。**你看到的是上排行榜的那几个，背后有成千上万个没人听过的。从0到1容易了，但从1到100依然需要持续投入。做出东西不难，做出持续有人用的东西，依然需要功夫。

Claude Code解决了「怎么做」的问题，但「做什么」和「做得好不好」，仍然要靠你自己的判断。工具只是工具，品味、需求洞察和持续打磨的意愿，AI给不了你。

这本书会带你从安装开始，一步步建立起用Claude Code独立构建产品的能力。每一章我都会分享真实的场景和踩过的坑，不只是功能介绍。到最后一章，你应该能够像我一样，把Claude Code从一个编程工具变成你的整个工作台。

门打开了。你可以选择走进去。

花叔（陈云飞）
2026年4月
于某个Airbnb

§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」。

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	IDE Agent（Cursor等）	终端Agent（Claude Code）
运行环境	编辑器内嵌，依赖IDE框架	终端原生，直接操作操作系统
自主程度	通常需要你在旁边确认	可以完全无人值守运行
系统集成	通过插件桥接git/CLI	直接操作git、shell、MCP
记忆系统	隐式的项目索引	显式的CLAUDE.md记忆文件
并行能力	主要单实例工作	原生支持多实例并行

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Opus 4.5之后就再也没有手写过一行代码。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。

花叔的经验：我本人从未手写过代码，所有产品（包括AppStore付费榜Top 1的小猫补光灯）都是用AI完成的。Claude Code让「不会写代码但能构建产品」这件事，从少数人的实验变成了大多数人的可能。

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。

两种场景对比一下：

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。

Claude Code增长快，本质上是命中了一个真实需求：大量聪明人想做出东西来，缺的不是想法，是把想法变成产品的能力。

这本书写给谁

工程师，想提高10倍效率。你已经会写代码，但每天大量时间花在样板代码、调试、写测试、处理CI/CD上。Claude Code能接管这些，让你把精力放在架构决策和产品思考上。

产品经理，想自己做MVP。你有产品直觉和用户洞察，但受限于开发资源。Claude Code让你一个周末做出一个能跑的原型，不用等排期，不用写PRD等开发理解。

创业者，想实现一人公司。你想验证商业想法，但不想在技术上花太多钱和时间。Claude Code让一个人拥有一个小团队的开发能力，网站、App、后端API，都可以一个人搞定。

> **核心建议**
>
> 不管你属于哪一类，这本书假设你聪明，但可能从没用过AI编程工具。我们从零开始，但不会在基础概念上磨叽太久。

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后仅6个月，达到**10亿美元年化收入**。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。Rakuten的新功能上线时间从24天压缩到5天；Zapier全组织AI采用率达97%，部署了800多个Agent。

Anthropic在2026年初发布的「Agentic Coding Trends Report」里有几个数字很能说明问题：78%的Claude Code会话涉及多文件编辑（一年前只有34%）；平均会话时长从4分钟增长到23分钟；每个会话平均47次工具调用。这不是在「补全代码」，这是在「执行项目」。

在开发者调查中，Claude Code在复杂任务（多文件重构、架构决策、大规模调试）的市场份额达到44%，8个月内超过了GitHub Copilot和Cursor。初创公司采用率75%。开发者「最受喜爱」评分：Claude Code 46%，Cursor 19%，GitHub Copilot 9%。

当前Claude Code背后的模型有三个：

- **Opus 4.6** — 推理能力最强，处理复杂任务和架构决策
- **Sonnet 4.6** — 性价比最优，日常编码的主力
- **Haiku 4.5** — 响应最快，适合简单查询和补全

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。

为什么是Claude，而不是别的模型

市面上不缺AI编程工具。GitHub Copilot有OpenAI的模型，Cursor集成了多家模型，还有各种开源方案。Claude Code能跑出来，靠的是两条腿同时走：**模型能力和工程设计**。

先说模型。

编程任务对模型的要求和聊天不一样。聊天追求「听起来对」，编程要求「跑起来对」。一个函数写错一个字符就是bug，一个架构决策失误要返工几天。这对模型的推理深度、长上下文理解、指令遵循能力要求极高。

Claude在这几个维度上的表现一直稳定在第一梯队。Opus 4.6在SWE-bench（真实GitHub issue修复基准）上持续领先，Sonnet 4.6在同等价位段没有对手。更关键的是1M token的上下文窗口，意味着Claude能一次「看到」一个大型项目的几乎全部代码，而不是只盯着你当前打开的文件。

再说工程。

很多人低估了工程设计对体验的影响。同一个模型，套不同的壳，出来的效果天差地别。Claude Code做对了几件事：

- **终端原生**。不依赖IDE框架的限制，直接操作操作系统。想跑什么命令跑什么命令，想操作什么文件操作什么文件。
- **记忆系统**。CLAUDE.md不是噱头，它让Claude Code有了跨会话的持久记忆。你的项目规范、架构偏好、编码习惯，写一次就永远生效。
- **工具生态**。Skills让能力可复用，Hooks让流程可自动化，MCP让外部服务可连接。这三个加在一起，构成了一个完整的Harness（工具链）。
- **多Agent架构**。从SubAgents到Agent Teams，Claude Code从第一天就在为「一群AI协作」而非「一个AI干活」做设计。

模型提供智力，工程提供杠杆。只有模型强没有好工程的产品，用起来别扭；只有好工程配弱模型的产品，天花板太低。Claude Code两条腿都硬，这是它能在企业级场景站稳的原因。

> **核心建议**
>
> 如果你在国内访问Anthropic API有困难，也不用担心。Claude Code支持配置第三方模型，附录B有详细的国内模型接入指南，包括智谱GLM、DeepSeek、Kimi、通义千问等主流平台的配置方法。

不只是编程工具

如果你以为Claude Code只能写代码，那你低估它了。

我用Claude Code写了四本橙皮书（你现在读的就是其中一本），运营着30万粉丝的自媒体账号，管理整个内容创作流程：从选题调研、写文章、生成配图、审校降AI味，到排版发布、多平台分发，全流程都在终端里完成。

这不是我一个人的极端用法，而是一个正在发生的趋势：**越来越多的产品和服务开始为AI而非人类设计接口。**

几个你可能已经注意到的信号：

- **企业协作平台推出CLI。**飞书、钉钉、企业微信都在提供命令行接口和API，让AI Agent能直接发消息、创建文档、管理日程。你不用再打开网页一个个点。
- **AI工具从Day 1就提供Skill。**像LibTV（AI视频生成）这样的新工具，在发布第一天就提供了Claude Code的Skill集成。用户不需要学新的界面，直接在终端里说「帮我生成一个30秒的产品演示视频」就行。
- **API-first变成Agent-first。**以前的产品是先做给人用的界面，API是附属品。现在反过来了：越来越多的产品先确保AI能用，人的界面反而是第二优先。

想一下这意味着什么。如果未来80%的软件都有Agent接口，那你跟这些软件交互的方式就不再是打开App、找菜单、点按钮，而是告诉你的AI Agent「帮我把这个数据导出来，做成图表，发到飞书群里」。

Claude Code就是你接触这个未来的起点之一。

学会用Claude Code，你学到的不只是「怎么让AI帮我写代码」，而是「怎么让AI帮我做一切需要在电脑上完成的事情」。这个能力的杠杆效应，远比编码本身大得多。

> **花叔的实践：**我目前在Claude Code中安装了超过60个Skills，覆盖内容创作、视频制作、图片生成、数据分析、项目管理等场景。我每天和Claude Code的交互中，写代码的部分可能只占30%，剩下70%是内容创作、调研、文件管理和跨平台协作。这就是终端Agent的真正威力：它不挑活。

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。

我第一次装Claude Code的时候，在终端敲了一行命令，30秒装完，输入 `claude`，回车。屏幕上出现一行光标在闪。我打了句「你好」，它回了句话。整个过程不到1分钟。

我当时的感觉是：就这？

是的，就这。没有复杂的IDE配置，没有插件冲突，没有环境变量折腾半天。后来我帮不少朋友装，最常遇到的问题反而不是技术层面的，而是心理层面的：他们不相信真的这么简单。所以这章我会把每一步都写清楚，包括我和朋友们踩过的那些小坑。

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	`curl -fsSL https://claude.ai/install.sh | bash`	macOS / Linux	⭐ 推荐
Homebrew	`brew install --cask claude-code`	macOS	适合brew用户
WinGet	`winget install Anthropic.ClaudeCode`	Windows	Windows首选

> **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。

装一下试试

macOS / Linux

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

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

1. **安装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。

2. **安装Claude Code**

   打开PowerShell或Git Bash，运行：

   ```
   winget install Anthropic.ClaudeCode
   ```

3. **验证安装**

   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。

各平台安装踩坑详解

虽然官方说法是「一行命令搞定」，但实际帮几十个人装下来，总结了各平台最常见的坑：

macOS常见问题

Xcode Command Line Tools没装。Claude Code依赖git，而macOS的git需要先安装Xcode命令行工具。如果你从来没在Mac上开发过，第一次在终端运行 `git` 会弹窗让你装。点安装就行，大概5分钟。装完再重新运行安装命令。

Apple Silicon和Intel的差异。M系列芯片和Intel Mac的安装命令完全一样，安装脚本会自动检测架构。但如果你用Rosetta 2跑了一个Intel终端环境，可能装到了Intel版。验证方法：运行 `file $(which claude)`，输出里应该有 `arm64`。

PATH环境变量。如果装完输入 `claude` 提示command not found，99%是PATH问题。检查 `~/.zshrc`（macOS Catalina之后默认zsh）里有没有：

```
# 确保这行存在（或类似的）
export PATH="$HOME/.local/bin:$PATH"
```

加完之后 `source ~/.zshrc` 或者重开终端。

Windows常见问题

用哪个终端。Windows终端选择比Mac多很多，容易搞混。推荐优先级：Windows Terminal > PowerShell > Git Bash > CMD。不要用CMD，太古老了，很多命令不支持。Windows Terminal是微软官方的新终端，从Microsoft Store免费安装。

WSL用户。如果你习惯用WSL（Windows Subsystem for Linux），可以直接在WSL里装Linux版的Claude Code。效果和原生Linux一样，而且不用折腾Windows的环境。WSL里装的Claude Code和Windows原生版是独立的，互不影响。

公司电脑的杀毒软件。有些企业杀毒软件（360、金山毒霸、Windows Defender的严格模式）会拦截curl下载。遇到这种情况，要么临时关闭，要么用WinGet安装（`winget install Anthropic.ClaudeCode`），WinGet走的是微软官方渠道，杀毒软件一般不拦。

Linux常见问题

glibc版本。极少数情况下，老旧的Linux发行版（CentOS 7等）glibc版本太低会报错。解决方案：升级系统，或者使用Docker运行。

无图形界面的服务器。Claude Code首次登录需要打开浏览器。如果你在一台无图形界面的服务器上安装，登录时它会给你一个URL和验证码，你在另一台有浏览器的电脑上打开URL、输入验证码完成认证。

企业网络的额外注意

有些公司网络有SSL中间人证书（用于监控流量），会导致SSL验证失败。症状是报 `UNABLE_TO_VERIFY_LEAF_SIGNATURE` 或 `CERT_HAS_EXPIRED`。解决方案是在环境变量中指定公司的CA证书：

```
export NODE_EXTRA_CA_CERTS=/path/to/company-ca.pem
```

如果你不知道公司CA证书在哪，找IT部门要。这个问题不只影响Claude Code，所有Node.js程序在企业网络里都可能遇到。

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 `claude`	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	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。

账号和钱的事

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。

付费三档：

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

怎么选？一个简单标准：每天用超过2小时，Pro大概率会撞限额。这时候升Max 5x是值得的。$100/月听起来不少，但跟它省下来的时间比，挺划算。

> **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。v2.1.88新增了 `claude auth login --console` 命令，可以直接用Anthropic Console账号登录（走API计费），不需要再单独配置API密钥。但对本书读者来说，直接订阅是最简单的开始方式。

说第一句话

装好了，账号也登录了。来试试。

1. **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
   ```

2. **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。

3. **发送你的第一条指令**

   试试这个：

   ```
   创建一个简单的HTML页面，显示"Hello, Claude Code!"，用好看的CSS样式。
   ```

   你会看到Claude Code开始工作：它会在你的目录里创建一个HTML文件，写入完整的代码。整个过程大约10-30秒。

4. **看看结果**

   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
   xdg-open index.html # Linux
   start index.html    # Windows
   ```

如果你能看到一个带样式的「Hello, Claude Code!」页面，恭喜，一切正常。

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

跑一遍这个清单：

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。

遇到问题了？

连不上、登录不了

Claude Code需要访问Anthropic的API服务器。如果连接超时或报错，先在终端运行 `curl -I https://api.anthropic.com` 检查网络连通性。如果返回HTTP状态码（不管是什么数字），说明网络通了；如果超时，需要检查你的网络环境。

安装时报Permission denied

macOS / Linux用户遇到权限错误，不要用 `sudo`。这么搞：

```
# 确保本地bin目录存在且有写权限
mkdir -p ~/.local/bin
# 重新运行安装脚本
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：

```
# Native Install用户
curl -fsSL https://claude.ai/install.sh | bash

# Homebrew用户
brew upgrade --cask claude-code

# WinGet用户
winget upgrade Anthropic.ClaudeCode
```

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。

装VS Code扩展

想在VS Code里用Claude Code的话：

1. 打开VS Code的扩展市场（Cmd+Shift+X 或 Ctrl+Shift+X）
2. 搜索「Claude Code」
3. 安装Anthropic官方的扩展
4. 安装后在侧边栏会出现Claude Code的图标，点击即可开始对话

VS Code扩展底层调用的是同一个CLI，所以你不需要单独配置账号，登录状态是共享的。

装Desktop App

不习惯终端的用户可以用桌面应用。去 `claude.ai/download` 下载安装包，双击装好就行。本质上是终端CLI的图形界面包装。

> **注意**
>
> 不管你选哪种方式，都建议先把CLI装好。CLI是基础，VS Code扩展和Desktop App都依赖它。CLI能跑，其他环境基本不会有问题。

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。

小猫补光灯上架那天，我女朋友问我：「你从想到做完花了多久？」我说大概1小时。她不信。但这真是实话：5分钟判断需求（小红书搜了一下，发现大量女生用纯色图片自拍补光，十几万赞，需求已验证），然后用AI编程工具来回几轮对话，就做完了。

当然，小猫补光灯是用Cursor做的（那时候Claude Code还没发布）。但核心体验是一样的：你描述你想要什么，AI帮你实现。不需要先学三个月编程，不需要理解什么是SwiftUI或React Native。你只需要知道你想做什么。

这一章我们用Claude Code从零做一个东西。规模比小猫补光灯小，但流程完全一样：先想清楚做什么，然后开口让Claude干。

热身：5分钟做一个个人主页

在做正式项目之前，先花5分钟热个身。这个小练习的目的是让你亲手体验一次「说 → 做 → 看」的完整循环。

```
mkdir my-homepage && cd my-homepage
claude
```

进入Claude Code后，说：

```
帮我做一个个人主页。单页HTML，要好看。
内容：我叫[你的名字]，是一个[你的职业/身份]。
加一句自我介绍，一段关于我的描述，底部放几个社交媒体链接的placeholder。
用现代简约风格，响应式布局。
```

Claude会在30秒内生成一个完整的HTML文件。打开看看：

```
open index.html       # macOS
xdg-open index.html  # Linux
start index.html      # Windows
```

大概率你会看到一个不错的页面。也许不是你理想中的样子——配色可能太普通，或者布局可以更好。没关系，继续说：

```
配色改一下，用深色背景+亮色文字。字体换成衬线体。
加一个渐变的背景动画效果。
```

Claude会修改文件，你刷新浏览器就能看到变化。再调几轮，直到你满意。

整个过程不超过5分钟，但你已经完成了一次完整的对话式编程循环：描述需求 → Claude实现 → 查看结果 → 提出修改 → Claude调整。后面所有的项目，不管多复杂，都是这个循环在跑。

> **核心建议**
>
> **把这个页面部署上线。**如果你想让别人也能看到，可以说：「帮我把这个页面部署到GitHub Pages。」Claude会帮你创建Git仓库、推送代码、配置GitHub Pages。整个过程大概再花5分钟。这样你就有了一个真实的在线个人主页——用AI做的第一个上线的东西。

热身完了。接下来做一个稍微复杂点的项目。

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 Describe What You Want

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：

```
# 你在Claude Code中输入的prompt:

帮我做一个AI新闻聚合CLI工具。需求如下:

1. 从以下RSS源抓取最近24小时的文章:
   - TechCrunch AI (https://techcrunch.com/category/artificial-intelligence/feed/)
   - The Verge AI (https://www.theverge.com/rss/ai-artificial-intelligence/index.xml)
   - Hacker News前30条 (https://hnrss.org/newest?q=AI&count=30)

2. 对每篇文章提取标题、链接、发布时间、来源

3. 按时间倒序排列，输出一份Markdown格式的日报到 output/ 目录

4. 用TypeScript写，用tsx直接运行

先别急着写代码，给我一个实现方案。
```

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 Review the Plan

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：

```
我来帮你做这个AI新闻聚合工具。先说一下方案：

项目结构：
ai-news-digest/
├── src/
│   ├── index.ts          # 入口文件
│   ├── fetcher.ts        # RSS抓取逻辑
│   ├── formatter.ts      # Markdown格式化
│   └── types.ts          # 类型定义
├── output/               # 日报输出目录
├── package.json
└── tsconfig.json

技术方案：
- RSS解析用 rss-parser 库
- HTTP请求用 node内置的fetch
- 日期处理用 date-fns
- 用tsx直接运行TypeScript，不需要编译步骤

实现流程：
1. 并行抓取三个RSS源
2. 统一格式、过滤24小时内文章
3. 按时间排序
4. 生成Markdown格式日报
5. 写入 output/YYYY-MM-DD.md

需要我开始实现吗？
```

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。比如：

```
# 你的反馈：
方案没问题。补充两点：
1. 日报里每篇文章除了标题和链接，加一句话摘要（从文章description里截取前100字）
2. 日报开头加一个统计：共收录X篇，来自Y个源
```

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 Watch Claude Work

确认方案后，Claude开始执行。终端里会看到一系列操作：

1. **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：

   ```
   Claude wants to run: npm init -y
   Allow? (y/n)
   ```

   按 `y` 允许。后面它还会请求安装 `rss-parser`、`date-fns`、`tsx` 等包。

2. **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）。不需要逐行看，但可以快速扫一眼文件结构是否合理。

3. **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。等熟悉了他的风格，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 Verify the Output

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，内容大概是这样的：

```
# AI新闻日报 — 2026-03-28

> 共收录 23 篇文章，来自 3 个源

---

## TechCrunch AI

### OpenAI发布新版GPT-5.4，上下文窗口扩展至200万
🔗 https://techcrunch.com/2026/03/28/openai-gpt-54...
📅 2026-03-28 14:30
> OpenAI今日发布了GPT-5.4版本，最大的变化是上下文窗口从100万扩展至200万tokens...

### Anthropic推出Claude Code Desktop App
🔗 https://techcrunch.com/2026/03/28/anthropic-desktop...
📅 2026-03-28 11:00
> Anthropic宣布Claude Code正式推出桌面应用程序，支持macOS和Windows...

---

## The Verge AI
...
```

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：

```
# 运行报错时，把错误信息粘贴给Claude：
运行报错了：
TypeError: Cannot read properties of undefined (reading 'map')
  at formatArticles (src/formatter.ts:15:23)
```

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 Iterate and Improve

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
现在每篇文章的摘要是从description里截取的，比较粗糙。
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。
```

Claude会修改代码，加入API调用逻辑。你验证后继续：

```
# 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。
加一个命令行参数：
- `npx tsx src/index.ts` 立即运行一次
- `npx tsx src/index.ts --cron` 开启定时模式
```

```
# 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：

不管项目大小，不管是小工具还是完整产品，底层模式都是这五步。

一个重要的心态转变 The Mental Shift

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	Claude Code编程
自己想方案，自己写代码	描述需求，Claude出方案和代码
一行一行调试	把错误信息给Claude，它自己调
查文档、查StackOverflow	直接问Claude「怎么实现XXX」
代码review靠人工	让Claude解释它写的代码
重构要先理解全部代码	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 Common Questions from Beginners

「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：

推荐	不推荐
「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。」	「你的代码第23行有bug。」（除非你确实知道问题在哪）

描述现象比定位代码行更有效。Claude可能发现问题根源不在你以为的地方。

「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。

出了问题怎么办——新手排错指南

做第一个项目的过程中，你大概率会遇到一些问题。我和帮过的几十个人都遇到过，几乎都有标准解法。

Claude创建了文件但运行报错

最常见的情况。通常是依赖没装好，或者环境变量没设。解决方法很简单：把完整的报错信息粘贴给Claude。

```
运行报错了，报错信息：
[粘贴完整报错]

帮我修一下。
```

Claude会读报错信息、定位问题、修复代码。90%的情况下一轮就能解决。如果两轮还没解决，说明可能是环境问题（Node.js版本、系统权限等），这时候描述你的系统环境：

```
我的环境：macOS 15.3, Node.js v22.5.0, npm 10.8.0
上面的报错修了两次还是报这个错，可能是环境问题？
```

Claude一直在「思考」不动了

如果超过1分钟没有任何输出，按 `Esc` 中断，然后重新发送你的请求。如果持续卡住，可能是网络问题，检查网络连接是否稳定。

也可能是你的请求太大了。比如让Claude一次性处理一个巨大的文件（几千行代码），它可能需要很长时间。试试拆小：「先看前100行，有什么问题？」

Claude做的东西和我想要的完全不一样

这几乎总是需求描述的问题。回顾一下你说了什么，对比一下Claude做了什么。常见的坑：

- **太模糊。**「做个好看的页面」→ Claude不知道你觉得什么好看。改成：「深色背景、大字标题、卡片布局、圆角设计」
- **假设Claude知道上下文。**「把那个功能加上去」→ Claude不知道你说的「那个」是什么。每个需求都要自包含
- **一次说太多。**10个需求挤在一条消息里，Claude可能丢掉其中几个。分成2-3条分别说

Claude修改了不该改的文件

偶尔会发生。按两次 `Esc` 打开Rewind菜单，可以：

- **回滚对话**：撤销最后几轮对话，从某个节点重新开始
- **回滚文件**：把被修改的文件恢复到之前的状态
- **两者都回滚**：对话和文件一起撤销

Rewind是你的安全网。知道它在就够了，不需要紧张。

依赖安装失败（npm/pip报错）

国内网络环境下，npm install可能很慢或者失败。两个常用方案：

```
# 方案一：用淘宝镜像
npm config set registry https://registry.npmmirror.com

# 方案二：让Claude用国内可用的替代包
告诉Claude：「npm装不了这个包，有没有替代方案或者我们不用这个依赖。」
```

Python用户遇到pip问题也类似，可以用清华镜像：`pip install -i https://pypi.tuna.tsinghua.edu.cn/simple 包名`。

Token用完了/限额了

Pro用户每天有使用限额。如果提示limit reached，有三个选择：等几个小时限额重置、升级到Max 5x、或者今天先到这。

一个减少Token消耗的技巧：用 `/compact` 命令压缩对话历史。长对话会消耗大量Token（每次发消息都要发送完整的对话历史），压缩后可以减少消耗。

> **核心建议**
>
> **最重要的排错技巧：**把完整的报错信息给Claude。不要手打报错信息（容易抄错），直接从终端里复制粘贴。Claude看到完整报错，90%的问题都能自己解决。

§04 核心工作流

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。

我有一次让Claude Code帮我改一个iOS项目的bug。按了回车之后去倒了杯水，回来发现它已经改完了——连改了7个文件，跑了测试，全部通过。我翻看它的操作记录：先读了报错日志，定位到根因在一个数据类型转换上，改了核心文件，然后追踪到6个引用了这个类型的地方，逐一更新，最后跑测试确认。整个过程大概3分钟。

如果是我自己来，光是理解这7个文件之间的关系就要花半小时。但反过来想：如果它改错了呢？如果它误解了bug的原因呢？我连看都没看就让它全改了。

这件事让我意识到：Claude Code的核心不是「它能不能做」，而是「你要不要先讨论一下再让它做」。这就是这一章要讲的工作模式问题。

Plan模式：先想清楚再动手 Plan Mode

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

Plan模式的黄金工作流

Boris推荐的完整流程是这样的：

1. **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。

2 **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。

3 **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。

什么时候该用Plan模式

用Plan模式	跳过Plan，直接做
你不确定实现方案	改一行代码、修个typo
需要改动多个文件	加一行console.log调试
你对这块代码不熟悉	你很清楚要改什么
重构或架构级别的变更	跑个测试、装个包
第一次接触一个新项目	重复性的日常操作

> **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 Auto Mode

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）。如果内容看起来像在试图劫持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%

大部分安全操作在第一阶段就直接通过了，只有可疑操作才会触发第二阶段的深度分析。

Auto模式会拦截什么

Anthropic从内部事件日志中提炼出的真实案例：

- **范围升级** — 你说「清理旧分支」，Claude把远程分支也删了。Auto模式会拦截，因为「清理」这个词太模糊，不构成删远程分支的授权
- **凭证探索** — Claude遇到认证错误，开始自行在环境变量中搜索其他API token。会被拦截
- **绕过安全检查** — 部署命令预检失败，Claude用 `--skip-verify` 重试。会被拦截
- **数据外泄** — Claude想分享代码，自行创建了一个公开的GitHub Gist。会被拦截

如何启用Auto模式

```
# 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
# Auto模式目前是Team plan的Research Preview功能
```

> **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。

Auto模式 vs --dangerously-skip-permissions

你可能在社区里看到有人推荐用 `--dangerously-skip-permissions` 来跳过所有权限提示。两者的区别很关键：

	Auto模式	--dangerously-skip-permissions
安全性	有AI分类器评估每个操作	完全无保护
危险操作	会被拦截，Claude被引导换一种方式	直接执行，不会有任何提示
Prompt Injection防护	有输入层探测器	无
适用场景	日常开发	完全隔离的沙箱环境、CI/CD

Boris本人的做法是两者都不用。他用 `/permissions` 预授权安全命令（下一节会讲）。但对于大多数人来说，Auto模式是一个很好的平衡点。

权限管理：你来定规矩 Permission Management

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

总结一下Claude Code的权限体系，从最省心到最精细：

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
/permissions 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

刚开始用的时候，建议先用默认的逐个确认。等你跑了几个项目、知道Claude通常会执行哪些命令之后，再切换到Auto模式或配置白名单。

Git操作：Claude天然就懂 Git Operations

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

最常用的操作：

```
# 让Claude自己看看改了什么，写一个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 Computer Use

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，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

我自己用下来，Computer Use最顺手的几个场景：

场景	为什么需要Computer Use
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，以前Claude完全帮不上忙，现在它可以直接操作
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，这种机械活让Claude代劳
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

这意味着什么

Computer Use看着只是一个新功能，但往深了想，它代表AI编程工具的一个方向性转变。

过去一年，AI编程的所有能力都建立在文本操作之上。读文件、写文件、执行命令、分析日志。整个交互界面就是一个终端。换个说法：AI只能操作那些可以被文本描述的东西。

Computer Use打破了这个边界。AI获得了和人类一样的GUI操作能力，它能看到屏幕上的一切，并且对它做出反应。

为什么这件事重要？因为**它直接扩大了「谁能用AI编程工具」这个圈**。以前你至少得理解命令行，知道什么是terminal，才能跟Claude Code协作。现在一个PM可以对Claude说「帮我在Figma里把这个按钮改成蓝色」，一个运营可以说「帮我在后台把这批用户状态改成VIP」。不需要理解任何技术概念。

长远来看，**AI的操作边界从「能写代码的地方」扩展到「屏幕上看得到的一切」**。这是一个质变。

当前的限制

先别太激动。现阶段Computer Use还有明显短板：

- **慢**。每一步操作都需要截图→分析→决定→执行，一个人类0.5秒完成的点击，Claude可能需要几秒钟
- **精细操作不靠谱**。拖拽一个滑块到精确位置、在一个密密麻麻的表格里选中某个特定单元格，这类操作它经常偏
- **不适合需要快速反应的场景**。动画、实时交互、游戏测试，Claude的反应速度跟不上

> **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。

Voice Mode：开口说话就能编程 Voice Mode

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

语音不是用来替代键盘的，它有自己最舒服的场景：

- **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

交互方式的变化比功能更新重要

我想多聊聊这个。

Voice Mode的功能本身挺简单的，就是语音转文字。但交互方式的变化，影响力往往比功能更新大得多。

键盘→鼠标→触屏→语音。回头看，每次交互方式变了，用工具的人就多了一大圈。鼠标让不会打字的人能用电脑，触屏让老人和小孩能用手机，语音呢？

现在把Voice Mode和Computer Use放在一起看：用语音描述你想要什么，Claude用Computer Use操作屏幕帮你实现。**Voice说需求，Computer Use执行操作。人可以完全脱离键盘和代码，纯靠说话让AI帮你构建东西。**

我们离「对着电脑说话就能做出一个产品」这件事，已经比大多数人想象的更近了。

当前的限制

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。你说一段200字的详细技术需求，中间可能出现误识别，还不如打字靠谱
- 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 Session Management

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	`/clear`	切换到完全不相关的任务时
压缩上下文	`/compact`	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 `/rewind`	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	`claude --continue`	终端不小心关了，想接着之前的会话
恢复指定会话	`claude --resume`	想回到某个历史会话继续工作
侧链提问	`/btw`	想问个不相关的问题，不污染当前上下文

/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。

推荐

修完API bug → `/clear` → 开始前端组件任务

不推荐

修完API bug → 直接说「接下来帮我做个前端组件」→ Claude可能把API的上下文混进来

/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。`/compact` 会保留关键信息，释放上下文空间。

`/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。

比如你正在让Claude重构一段代码，突然想问「TypeScript的 Record 类型怎么用来着？」用 `/btw` 问，不会污染重构任务的上下文。

六个坑，你大概率会踩 Common Anti-Patterns

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。在一个已经跑偏的对话上修补，远不如推倒重来。

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。Boris说的第13条技巧就是：给Claude一种验证工作的方式。你自己也一样。

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，会话管理保持上下文干净。这五个工作流覆盖90%的日常场景。剩下10%的高级用法后面几章展开。

§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。

我写公众号文章有一条死规矩：不用破折号（——）。这是明显的AI味标志。我在CLAUDE.md里写了这条规则之后，Claude就会遵守。但有一次长对话聊了快1小时，上下文被自动压缩了，Claude又开始疯狂用破折号。我才意识到：对话里说过的话会被压缩遗忘，但CLAUDE.md每次启动都会重新读取。

从那以后，我养成了一个习惯：任何我对Claude说过两次以上的要求，都会写进CLAUDE.md。半年下来，我的写作项目CLAUDE.md从空白文件长成了一套完整的路由系统——根目录的CLAUDE.md负责分发任务到不同的子目录，每个子目录有自己的规则文件。60多个Skills、数百条规则，全是这样一条一条攒起来的。

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 Guardrails, Not Manuals

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。

Shrivu分享了一个更有意思的做法：

> **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：

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 Hierarchy

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
```

1. **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。

2. **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。

3. **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
```

但注意前面说的：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 Automatic Memory

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	Auto Memory
适合团队共享的规则	适合个人偏好
检入git	存在本地
你主动维护	Claude自动维护
结构化、有组织	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 The Iterative Flywheel

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工作流时，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：

1. **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。

2. **第二周：护栏初现**

   你把Claude犯过的错一条条记下来。「不要在这个文件里用相对路径」「跑完迁移记得重新生成类型」。错误率开始下降。

3. **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。

4. **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。你把同样的方法迁移到新项目，启动速度越来越快。

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。

去年我让Claude Code「帮我优化一下这个页面的样式」。本意是调几个间距和字号。结果它把整个CSS文件重构了，还「顺手」加了一堆我没要求的改进，把我之前花了一下午调好的排版全毁了。我花了半小时才回退到之前的状态。

后来我学乖了。同样的需求，我改成：「把标题的 font-size 从 16px 改成 18px，段落间距从 10px 改成 14px，只改这两处，不要动其他样式。」Claude 30秒改完，完美。

区别在哪？不是Claude变聪明了，是我说话变清楚了。这一章就是讲这件事：不是教你写花哨的prompt，是教你怎么说人话让Claude准确执行。

怎么说话Claude才听得懂 Describing What You Want

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：

1. **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。

2. **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。

3. **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

帮我加个搜索功能

推荐

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

不推荐

接口报错了，帮我看看

推荐

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

不推荐

优化一下性能

推荐

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 Context Engineering

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 Let Claude Interview You

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。

把Claude当高级工程师提问 Claude as Your Senior Engineer

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 Multi-turn Conversation Strategy

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，中间的思考过程不会污染主上下文。

> **注意**
>
> 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`**。

Effort级别：别省这个钱 Effort Level

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 The Art of Asking

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、可复用工作流
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读完会按里面的规则办事。

工作流型：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。

工作流型skill的关键配置

工作流型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：

```
mkdir -p ~/.claude/skills/boris && \
curl -L -o ~/.claude/skills/boris/SKILL.md \
  https://howborisusesclaudecode.com/api/install
```

安装后你就拥有了Boris日常使用的工作流，包括他的commit规范、PR模板、代码审查标准等。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

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。

```
// .claude/settings.json
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "command": "npx eslint --fix $CLAUDE_FILE_PATH"
      }
    ]
  }
}
```

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（读文件、运行测试）自动批准，高风险的（删除文件、推送代码）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MCP服务器

```
# 添加一个MCP服务器
claude mcp add slack -- 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。

```
// .mcp.json
{
  "mcpServers": {
    "slack": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-slack"],
      "env": {
        "SLACK_TOKEN": "${SLACK_TOKEN}"
      }
    },
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres", "${DATABASE_URL}"]
    }
  }
}
```

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：
```bash
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

注意：PR描述要包含变更摘要。
````

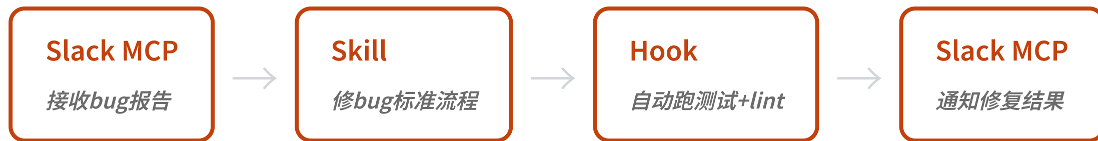
输入 `/commit-push-pr`，Claude就会按照这个流程自动执行。因为command文件存在 `.claude/commands/` 里，它会随Git一起提交，团队成员都能用。

> **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工作流：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

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。

花叔的Skills实战：从0到60+个Skill的过程

说了这么多机制，分享一下我自己的实际用法。目前我在Claude Code中安装了超过60个Skills，覆盖内容创作、电子书出版、视频制作、项目管理等场景。这些不是一口气装的，是半年里一个一个「被需求逼出来」的。

第一个Skill：三遍审校

最早让我意识到「该写Skill了」的场景是文章审校。我写完文章后总要对Claude说一大段话：「帮我降低AI味，不要用破折号，不要用'说白了'这类套话，检查有没有虚假数据，每段都过一遍……」每篇文章都要念一遍，有时候还漏几条。

于是我写了第一个Skill：`huashu-proofreading`。

```
# SKILL.md 核心结构（简化）

三遍审校流程：

第一遍：内容事实核查
- 所有数据、日期、产品名是否准确
- 有没有编造的内容

第二遍：降AI味
- 检查6大类AI腔：破折号过多、排比三连、假设性开头...
- 花生禁用词：说白了、简单来说、换句话说...

第三遍：风格打磨
- 「」引号规范
- 加粗标记重点（约10处/篇）
- 段落紧凑度检查
```

写完这个Skill后，审校从每次15分钟的口述变成一个 `/proofreading` 命令。更重要的是，规则不会因为上下文压缩而丢失。

从编程延伸到内容创作

尝到甜头后，我开始把所有重复流程都Skill化。半年下来，形成了一个覆盖完整内容创作链的Skills体系：

阶段	Skill	做什么
选题	`/topic-gen`	生成3-4个选题方向，每个含标题、大纲、工作量评估
调研	`/research`	多轮WebSearch + 增量保存到调研文件，防会话截断丢失
写作	`/article-edit`	标准化编辑流程：完整阅读→列修改项→增量编辑→变更摘要
配图	`/image-upload`	AI生成图片→上传图床→插入Markdown链接，全自动
审校	`/proofreading`	三遍审校降AI味，从60%降到30%以下
分发	`/article-to-x`	长文浓缩成200-500字社交媒体内容
发布	`/feishu`	写入飞书文档 + 设置权限 + 发群消息通知

一篇文章的完整生命周期，从选题到发布，全部有对应的Skill。每个Skill背后都是我踩过的坑凝结成的规则。

最意外的用途：电子书出版

我有一个Skill叫 `huashu-book-pdf`，负责橙皮书的全流程：调研→章节规划→多Agent并行写作→HTML片段构建→EPUB生成→微信读书上架。这个Skill把一本电子书的生产流程压缩到了几个小时。

更有意思的是视频方向。`huashu-video-director` 可以从创意开始，经过角色设定、分镜设计、Prompt生成，调用AI视频生成模型完成多镜头视频。`huashu-script-polish` 做的事更细：把书面化的脚本改成适合大声说出来的口语，三遍审校确保「像人在说话」。

Skill不是你的，是给AI用的

这里有一个思维转变值得说清楚。

很多人第一反应是：「Skill就是一个SOP文档嘛，我自己看着执行也一样。」技术上没错，但miss了重点。

Skill的读者不是你，是AI。它的措辞、结构、检查点，都是为了让Claude更好地执行而优化的。人看文档会「跳着读」，AI会逐字执行。人会忘记边界条件，AI不会。人执行SOP需要意志力，AI不需要。

我目前有一些Skill加起来超过2000行指令。如果让我自己执行这些步骤，每篇文章可能要花3-4个小时在流程操作上。用Skill之后，我的时间都花在了真正需要判断力的地方：选题好不好、论点立不立得住、这段话读者看完能不能行动。

一个趋势：越来越多的第三方工具开始在发布当天就提供Claude Code的Skill。比如LibTV（AI视频生成平台）在上线第一天就发布了对应的Skill，用户不用学新界面，直接在终端里生成视频。飞书、钉钉等企业平台也在推CLI接口。这意味着Skills生态会越来越丰富，你可能很快就能用一行 `/xxx` 做到今天需要打开五个App才能完成的事情。

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

Claude Code最被低估的能力不是它写代码有多快，而是它可以同时跑很多个。学会并行之后，你的工作模式会从「一个人配一个AI」变成「一个人指挥一支AI团队」。

写这本橙皮书的时候，我同时开了6个Claude Code进程，每个负责写不同的章节。Agent A写§01和§02，Agent B写§03和§04，以此类推。6个进程并行跑了大约2小时，10个章节的初稿全部完成。如果串行写，至少要一整天。

但这不是没有代价的。6个Agent意味着6份独立的上下文，Token消耗是单Agent的6倍。那天下午花了大概50美金。而且6个Agent写出来的风格不完全一致，后期还需要一轮统一审校。所以并行不是越多越好，而是要判断：这个任务拆开了能不能独立做？合并的时候成本高不高？

一个人为什么要开那么多窗口

Boris Cherny日常是这么干活的：本地开5个Claude Code实例（独立的git checkout），云端再开5到10个claude.ai/code网页会话，每个跑不同的任务。一个写新功能，一个修bug，一个写测试，一个重构，一个做代码审查。同时进行。

我一开始觉得这也太夸张了。后来自己试了才理解，这不是炫技，是Boris给团队的第一条生产力建议：**做更多并行工作。**

原因很直接：Claude Code的工作模式是「你给任务 → Claude花几分钟执行 → 你review结果 → 给下一个任务」。中间有大量等待时间。只开一个session，大部分时间你在等Claude干活。开5个session，你review第一个的时候其他4个还在跑，等待时间几乎降到零。

关键前提是：**每个session需要在独立的代码环境中运行**，否则它们会互相覆盖文件，制造冲突。这就是Git Worktrees要解决的问题。

Git Worktrees：并行的基础设施

Git Worktree允许你从同一个仓库创建多个工作目录，每个工作目录在不同的分支上，文件系统完全隔离。Claude Code对worktree做了原生支持：

```
# 启动一个在独立worktree中运行的Claude session
claude --worktree

# 在Tmux会话中启动（可以后台运行）
claude --worktree --tmux
```

每次运行 `claude --worktree`，Claude Code会自动创建一个新的worktree、切到一个新分支，然后在那个隔离环境中工作。完成后你可以把分支合并回主分支。

Tmux集成

加上 `--tmux` 参数，session会在一个Tmux窗口中启动，你可以用快捷键在不同session之间切换。Boris设置了shell别名来快速跳转：

```
# ~/.zshrc 中添加
alias za="tmux select-window -t claude:0"
alias zb="tmux select-window -t claude:1"
alias zc="tmux select-window -t claude:2"
```

`za` 跳到第一个session，`zb` 跳到第二个，以此类推。如果你用Desktop App，界面上有一个worktree复选框，勾上就行，不需要手动配置Tmux。

Subagents：给主session叫个帮手

并行session适合处理互不相关的独立任务。但有时候你要的不是开一个新窗口，而是在当前任务中调一个「专家」来处理特定环节，比如让一个安全审查专家review你刚写的认证代码。这就是Subagents干的事。

在 `.claude/agents/` 目录下放一个 `.md` 文件，就定义了一个subagent：

```
.claude/agents/ ├── security-reviewer.md # 安全审查专家 ├── code-simplifier.md # 代码精简专家 ├── verify-app.md # 应用验证专家 └── code-architect.md # 架构设计专家
```

每个agent文件可以定义自定义名称、工具集权限、权限模式，甚至指定使用的模型。比如安全审查agent可以配置为只有读权限（不能改代码），指定使用推理能力更强的模型。

Subagents的核心价值

Subagents最重要的特性不是「专业分工」，而是**独立上下文**。

每个subagent运行在自己的上下文窗口中，不消耗主session的上下文空间。当主session的对话已经很长、上下文快要满了的时候，调用一个subagent来处理子任务，相当于开了一个新的「思考空间」，不会挤压主session的容量。

你甚至可以在prompt中加上「use subagents」，让Claude主动判断什么时候该把子任务分配给subagent。这会让Claude投入更多计算资源来完成复杂任务。

> **核心建议**
>
> 实用的subagent组合：**security-reviewer**（每次涉及认证、权限、数据存储时自动调用）+ **verify-app**（修改完成后自动启动应用并验证功能是否正常）。这两个覆盖了「写得对不对」和「跑得起来吗」两个最常见的验证需求。

Agent Teams：让它们自己协调

Worktrees让你手动管理多个并行session，Subagents让主session调一个专家。Agent Teams更进一步：多个session之间能互相通信、协调分工。

Agent Teams在2026年2月发布，目前是Claude Code最强大的协作模式。核心理念很简单：不是你来协调多个agent，而是让agent自己协调。我之前写过一篇用3个AI队友45分钟做红白机游戏厅的实测，就是用的这个功能。

Writer/Reviewer模式

最经典的用法是一个写代码、另一个审代码：

1. **Writer Agent 写代码**
   负责实现功能，按照需求写代码、跑测试

2. **Reviewer Agent 审代码**
   review Writer的输出，指出问题、建议改进

3. **Writer根据反馈修改**
   收到review意见后改进代码，形成迭代循环

这个模式比单个agent写代码好不少。原因和人类团队一样：写代码的人容易陷入自己的思路，审代码的人能从不同角度发现问题。两个agent互相盯着，产出质量肉眼可见地提升。

测试驱动模式

另一个高效的模式：一个agent写测试，另一个写实现。写测试的agent先根据需求定义「什么是正确行为」，写实现的agent再去满足这些测试。这就是AI版的TDD（Test-Driven Development）。

Agent Teams会自动共享任务状态和消息，你不需要手动在agent之间复制粘贴信息。它们有一个team lead角色来协调分工和进度。

Coordinator Mode：四阶段协调

Agent Teams内部其实有一套更精细的协调机制。复杂任务会自动走四个阶段：先让多个worker并行调查代码库（Research），然后coordinator综合发现生成规格说明（Synthesis），接着worker按规格做精准修改（Implementation），最后验证结果（Verification）。你不需要手动配置这个流程，Agent Teams会根据任务复杂度自动判断要不要走完整的四阶段。

Fan-out批处理：人海战术的AI版

前面都是几个agent配合做一件事。Fan-out模式解决的是另一类问题：**同一个操作需要对大量文件重复执行。**

非交互模式

Claude Code支持非交互模式，用 `-p` 参数传入prompt，适合在脚本中调用：

```
# 非交互模式执行单个任务
claude -p "把这个文件从 JavaScript 迁移到 TypeScript"
```

配合shell循环，你可以批量处理：

```
# 批量迁移一批文件
for file in $(cat files-to-migrate.txt); do
  claude -p "Migrate $file from JS to TS" \
    --allowedTools "Edit,Bash(git commit *)" &
done
```

注意末尾的 `&`：这让每个Claude实例在后台并行运行。如果有50个文件要迁移，50个Claude同时跑，可能几分钟就完成了原本需要一整天的工作。

/batch 命令

如果你不想自己写shell脚本，Claude Code提供了 `/batch` 命令来简化这个过程：

1. **交互式规划**

   告诉Claude你想做什么（比如「把所有React类组件迁移到函数组件」），Claude会分析项目，列出所有需要处理的文件

2. **确认执行**

   你review计划，确认后Claude启动数十个agent并行执行

3. **汇总结果**

   所有agent完成后，Claude汇总成功/失败情况，你只需要处理少数失败的case

这种模式特别适合大规模重构、代码迁移、批量修复等场景。一个人加Claude，抵得上一个工程团队花一周做的迁移工作。

不一定要盯着电脑

前面说的都是本地终端操作。但Claude Code也支持远程和异步执行，不用一直守在终端前面。

Remote Control

通过Remote Control功能，可以生成一个连接链接。在手机上打开这个链接，就能远程创建和管理本地的Claude session。适合通勤路上想启动一个任务、出门前让Claude跑起来的场景。Boris提到自己早上会用iPhone通过Claude移动App启动会话，之后在桌面继续。

Claude Code on Web

开发环境在云端的话（或者你只是想在浏览器里用），可以通过 claude.ai/code 直接在浏览器中运行Claude Code，不需要安装本地环境。

/schedule：云端定时任务

```
# 设定一个云端定时任务
/schedule "Check for outdated dependencies and create PRs"
```

`/schedule` 设定定时触发的Claude任务，在云端执行。电脑关机了任务照样按时跑。适合日常维护类工作：依赖更新、安全扫描、日报生成。

/loop：本地长时间运行

有些任务要跑很长时间（监控CI状态、持续集成测试）。`/loop` 让Claude在本地最多无人值守运行3天，你去做别的事，它在后台一直跑。

> **异步工作的心智转变**
> 传统开发是同步的：你写代码、跑测试、等结果。异步模式下，睡觉前启动一批任务，早上起来review结果。把AI当成「夜班团队」，白天你定方向做决策，晚上它执行。

Anthropic自己怎么用的

Anthropic发布过一份白皮书「How Anthropic Teams Use Claude Code」，记录了内部各团队的真实用法。挑几个有意思的：

数据基础设施团队用Claude Code调试Kubernetes集群。Pod出问题时，让Claude读取日志、分析错误栈、定位根因、给修复建议。过去需要Senior工程师花很长时间排查的问题，Claude几分钟就能给出方向。

安全团队用Claude Code追踪复杂的控制流。安全审计需要跟踪一个请求从入口到数据库的完整路径，手动做极其耗时。他们让Claude自动追踪并生成调用链路图。

营销团队用Claude Code生成广告变体。一次活动需要几十个版本的文案和素材组合，过去设计师和文案要花好几天迭代。现在Claude批量生成，营销人员只做选择和微调。

但最让我意外的是**法务团队**的案例：一位律师用Claude Code搭了一个电话树系统（来电后自动路由到对应的法律顾问）。这位律师不是工程师，不会写代码，但从零搭建了这个系统并上线了。这个案例让我觉得，Claude Code的受众已经不局限于工程师了。

我自己摸索出来的几条经验

从2个session开始就好。不用一上来就开10个。先习惯在两个之间切换：一个做主任务，另一个做辅助的（写测试、做code review）。等觉得游刃有余了再加。

每个session给一个明确的角色。不要让所有session都做「随便什么任务」。给它们分工：这个负责前端、那个负责后端、那个专门跑测试。角色越清晰，你管理起来越轻松。

用git分支隔离一切。每个session在自己的分支上工作，通过PR合并。千万不要让多个session操作同一个分支，我试过一次，冲突解到怀疑人生。

定期扫一眼，不要完全放羊。并行不等于不管。每隔15-20分钟看看各个session的进度，及时纠偏。一个session跑偏了及时停掉，比让它跑完再返工划算得多。

> **核心建议**
>
> Boris的总结我很认同：把并行工作想象成管理一个远程团队。你不需要盯着每个人写每一行代码，但你需要清楚每个人在做什么、进度如何、有没有卡住。你的工作从写代码变成了做项目管理。

§09 从零构建一个完整产品

Build a Complete Product from Scratch

前8章讲零件，这一章组装。用一个真实项目把前面学的东西全串起来，你会发现Claude Code真正的威力不在于某个单一功能，而在于它们连起来之后的化学反应。

央视采访那天，记者问我能不能现场手搓一个App。我打开终端，启动Claude Code，10分钟做了一个能跑的小产品。摄像机对着我的屏幕拍了全程。

后来有人在B站评论区说：「10分钟做出来的东西能叫产品吗？」说实话，不能。10分钟做出来的是一个能跑的原型。从原型到产品，中间还有大量的工作：用户反馈处理、边界情况覆盖、性能优化、上架审核。小猫补光灯1小时做出第一版，但后续迭代了好几个月，做了Pro版、小程序版、鸿蒙版。

这一章讲的就是这个「从原型到产品」的完整过程。不是10分钟的demo秀，是一个能交付、能使用、能迭代的真实产品是怎么做出来的。

为什么拿「AI周报助手」当例子

选这个项目我想了一会儿。好的教学案例得满足几个条件：真实有用（不是todo list这种永远不会打开第二次的东西）、复杂度适中（半天到一天能搞定，但涉及前后端+API+AI调用）、技术栈匹配（Next.js + Tailwind是Claude Code最擅长的组合）。

AI周报助手做的事情很直白：连接你的GitHub，获取本周所有commit，用AI总结成一份可读的周报页面，支持一键分享给同事。每个工程师每周五都在做的事，每次花半小时。我们要把它缩短到10秒。

这个项目会用到前面几乎所有章节的东西。不过你不需要全记住，跟着做就行。

Phase 0：先别急着写代码

我一开始做产品时犯过一个错：想到什么就立刻让Claude开始写代码。结果做到一半发现需求没想清楚，只能推翻重来。

后来我养成了一个习惯：先让Claude采访我。它其实很擅长这个。

```
claude "I want to build a weekly report tool. Before writing any code,
interview me to understand the requirements. Ask me questions one at a
time about target users, core features, and technical constraints."
```

Claude会开始像产品经理一样采访你：

- 目标用户是谁？（个人开发者？还是团队？）

- 核心功能有哪些？（只生成周报？还是也需要分享？）
- 技术偏好？（部署到哪里？用什么框架？）
- 有没有设计参考？（周报长什么样？）

回答完这些问题后，让Claude把结论整理成一份规格文档：

```
claude "Based on our discussion, create a SPEC.md file that captures
all requirements, user stories, and technical decisions."
```

这份SPEC.md是整个项目的锚点。后续所有开发都围绕它展开。

为什么用新session执行：需求分析是对话密集的过程，会消耗大量上下文窗口。确认完SPEC.md后，开一个新session开始开发。新session会自动读取SPEC.md和CLAUDE.md，拥有干净的上下文空间来写代码。这就是§06里讲的「何时开新session」的实战应用。

Phase 1：项目初始化

新session，干净的开始。一句话创建项目骨架：

```
claude "Create a Next.js project called weekly-report with Tailwind CSS.
Set up the basic folder structure following SPEC.md requirements.
Include TypeScript, and configure ESLint."
```

Claude会执行一连串操作：运行`npx create-next-app`、安装依赖、配置Tailwind、创建基础目录结构。你在终端里看到的是它自动执行的每一条命令和创建的每一个文件。

项目创建完成后，文件结构大致是这样的：

```
weekly-report/
├── app/
│   ├── layout.tsx
│   ├── page.tsx
│   ├── api/
│   │   ├── github/route.ts
│   │   ├── summarize/route.ts
│   │   └── auth/[...nextauth]/route.ts
│   └── report/[id]/page.tsx
├── components/
├── lib/
├── CLAUDE.md
├── SPEC.md
├── tailwind.config.ts
└── package.json
```

接下来是关键一步：配置CLAUDE.md。这是§05整章的核心，现在是实战：

```
# CLAUDE.md

## Project Overview
AI-powered weekly report generator. Connects to GitHub, summarizes
commits with AI, generates shareable report pages.

## Tech Stack
- Next.js 15 (App Router) + TypeScript
- Tailwind CSS for styling
- NextAuth.js for GitHub OAuth
- Claude API (via Anthropic SDK) for summarization

## Code Style
- Use server components by default, 'use client' only when needed
- API routes in app/api/, use Route Handlers
- Prefer named exports
- Error handling: always use try-catch in API routes

## Testing
- Run `npm run lint` before committing
- Test API routes with curl before building UI
```

有了这份CLAUDE.md，后续每次新session或新的对话，Claude都知道这个项目的技术决策和代码规范。不需要你每次都重新说明。

Phase 2：正式开发（最耗时但也最爽的部分）

到这里才真正开始写代码。我一般会先用Plan模式聊一轮架构，不急着动手。在终端里输入`/plan`切换到Plan模式：

```
"I need to implement the core flow: GitHub OAuth login → fetch this
week's commits → send to Claude API for summarization → display
the report. Let's discuss the architecture before coding."
```

Claude会给出一个完整的技术方案：API路由如何设计、数据流怎么走、组件怎么拆分。你可以在这个阶段提出疑问、调整方案。这就像是和一个高级工程师在白板前讨论。

确认方案后，退出Plan模式，让Claude开始实现：

```
"Plan looks good. Now implement it step by step. Start with GitHub
OAuth, then the commit fetching API, then the AI summarization."
```

Claude会按照讨论好的方案，逐步创建文件、写代码、安装必要的依赖包。你会看到它在终端里创建`app/api/auth/[...nextauth]/route.ts`配置NextAuth，然后创建`lib/github.ts`封装GitHub API调用，再

创建`app/api/summarize/route.ts`接入Claude API。

每完成一个模块，做一次验证：

1. **GitHub OAuth**

   让Claude跑`npm run dev`，打开浏览器点登录按钮，看能不能跳转到GitHub授权页面。如果报错，把错误信息粘贴给Claude，它会自己修。

2. **Commit数据获取**

   登录成功后，用curl测试API路由：`curl localhost:3000/api/github`，看返回的commit数据是否正确。

3. **AI总结生成**

   拿真实的commit数据调用总结接口，检查AI生成的周报内容是否合理、是否有幻觉。

> **注意**
>
> 每做完一步就验证，这是我踩过最多坑之后总结出来的最重要的习惯。不要让Claude一口气写完所有代码再测。问题越早发现越容易修。我曾经让它连续写了8个文件，最后发现第2个文件的API路由就写错了，后面全废了。

Phase 3：让它好看起来

功能跑通了，但界面大概率很丑。正常。我做任何产品都是先让功能跑起来，再管好不好看。

Claude Code支持图片输入。最直接的方式：截一张当前页面的图，粘贴给Claude：

```
claude "Here's a screenshot of the current report page.
[paste screenshot]
Issues: 1) The header is too cramped 2) The commit list needs
better spacing 3) Add a share button in the top right"
```

Claude会看到你的截图，理解当前的视觉问题，然后精准修改对应的CSS和组件代码。

这种「截图→反馈→修改」的循环非常高效。传统开发中，你需要在设计稿和代码之间反复对照；现在你只需要截图、说问题、等修好。

几轮迭代后，继续打磨响应式：

```
"Test the report page on mobile viewport (375px width). Fix any
layout issues. The share button should be full-width on mobile."
```

Claude会自己调整浏览器视口大小（如果你用了VS Code集成），或者直接修改Tailwind的响应式类名。

> **核心建议**
> UI打磨阶段最高效的做法是：先把所有问题一次性列出来，让Claude批量修改，而不是改一个看一个。批量反馈能让Claude更好地理解整体设计意图。

Phase 4：扩展能力实战

核心产品已经能用了。现在是§07内容的实战时间：给项目加上Skills、MCP和Hooks。

创建一个Skill

每次想生成周报，都要打开项目、执行一串操作。能不能一句话搞定？可以，用Skill。

在项目根目录创建 `.claude/skills/generate-report/SKILL.md`：

```
# /weekly-report

Generate this week's report.

## Steps
1. Run the dev server if not running
2. Call /api/github to fetch commits since last Monday
3. Call /api/summarize to generate the report
4. Open the report page in browser
5. Show the shareable URL
```

配置完成后，在任何Claude Code session中输入 `/weekly-report`，它就会自动执行以上所有步骤。一个命令，10秒出周报。这就是Skills的威力：把重复流程变成一键操作。

添加MCP：连接Slack

周报生成了，手动发到Slack频道太麻烦。用MCP把两者打通：

```
claude "Add a Slack MCP server so the generated report can be
automatically posted to #team-updates channel. Use the Slack
Web API with a bot token."
```

Claude会帮你配置MCP server，在 `.claude/mcp.json` 里注册Slack连接。配置完成后，你的 `/weekly-report` skill可以加上最后一步：把周报内容发送到Slack。

设置Hook：自动Lint

最后一个扩展：每次Claude提交代码前，自动跑lint检查。

```
claude "Set up a pre-commit hook that runs ESLint and TypeScript
type checking. If there are errors, fix them before committing."
```

这样每次Claude Code执行`git commit`时，都会先确保代码质量。这是§07里Hook的典型用法：在关键节点注入自动检查。

Phase 5：部署上线

产品在本地跑得好好的，该让全世界都能访问了。

```
claude "Deploy this project to Vercel. Set up the environment
variables for GitHub OAuth, Claude API key, and Slack bot token.
Also create a GitHub Actions workflow that runs lint and type
check on every PR."
```

Claude会执行部署命令、配置环境变量、创建CI/CD配置文件。整个过程你不需要打开Vercel的dashboard，也不需要手写GitHub Actions的YAML。

部署完成后，你会拿到一个线上URL。在浏览器里打开，走一遍完整流程：登录→获取commit→生成周报→分享。确认一切正常。

如果你的团队也在用Claude Code，还可以加上一个额外步骤：

```
claude "Add a Claude Code Action to the GitHub repo that
automatically reviews PRs for code quality and potential bugs."
```

这样每次有人提PR，Claude会自动做code review，在PR上留评论。CI/CD的完整闭环就建立起来了。

回顾：这个项目用到了什么

回头看看，这个下午我们做了什么。一个完整的Web应用，从想法到上线，全部在终端里完成。

Phase	做了什么	对应章节
Phase 0	用Claude采访自己，生成SPEC.md	§06 对话技巧
Phase 1	项目初始化 + CLAUDE.md配置	§02 安装 + §05 CLAUDE.md
Phase 2	Plan模式讨论 + Auto模式实现	§04 核心工作流
Phase 3	截图反馈 + UI迭代	§03 Agent式工作
Phase 4	Skills + MCP + Hooks	§07 扩展能力
Phase 5	Vercel部署 + CI/CD	§04 Git操作

如果从头到尾比较顺利，整个过程大概需要5-8个小时。其中最耗时的是Phase 2（核心功能开发），OAuth配置和API调试需要反复验证，环境变量、回调URL这些琐碎的东西每个都可能卡你十几分钟。不顺利的话，一天也

正常。

同样的项目，如果纯手写呢？一个熟悉Next.js和OAuth的全栈工程师大概需要2-3天。不太熟悉的话可能需要一周。差距不只是速度，更重要的是：你在整个过程中做的是产品决策而不是代码实现。你在思考「周报应该包含哪些信息」「分享页面需要登录吗」这些产品问题，而不是在Stack Overflow上查「NextAuth GitHub provider怎么配」。

我自己踩出来的几条经验

说点掏心窝子的话。

小猫补光灯做到App Store付费榜Top 1的时候，很多人问我是不是有一个开发团队。答案是没有。从第一行代码到上架审核，全部是AI写的。我从未手写过代码。

但这不意味着开发过程很轻松。恰恰相反，我踩过非常多坑，总结出来几条最核心的经验：

一、需求拆小，每次只给一步

新手最常见的做法：兴奋地把整个产品需求一股脑甩给Claude，「帮我做一个完整的XX应用，需要登录、数据库、付费、推送、分享...」Claude确实会开始做，但最终产出往往是一团乱。

我现在的做法是拆成最小可验证的步骤。先做登录，确认能跑。再做数据存储，确认能存能取。然后是核心业务逻辑，然后才是UI。每一步验证通过，再进下一步。

推荐

这样给需求：
「先实现GitHub OAuth登录。登录成功后在页面上显示用户名和头像。」

验证通过后→
「现在添加获取commit的API。获取登录用户最近7天在所有repo的commit，返回JSON。」

不推荐

不要这样给需求：
「做一个周报工具，需要GitHub登录、获取commit、AI总结、漂亮的UI、分享功能、Slack通知、部署到Vercel。」

二、先跑通最小功能，再一步步加

和上一条有关系但不一样。拆小是关于怎么给Claude下指令，这一条是关于产品策略。

做小猫补光灯时，第一个版本只有一个功能：打开App，屏幕变白，亮度拉满。就这么简单。我拿它自拍了一张，确认补光效果还行，才开始加色温调节、亮度滑块、定时拍照这些功能。

用Claude Code也是一样。先做一个能跑的最简版本，自己用两天，发现真正需要什么再加。你脑子里想象的功能和实际用起来需要的功能，往往差异很大。

三、验证比开发更重要

这句话我反复强调。Claude Code写代码的速度很快，快到你可能会产生一种幻觉：它写得这么快，应该是对的吧？

不一定。AI生成的代码需要验证，就像人写的代码需要测试一样。区别在于：人写代码可能一天写200行，验证成本可控；Claude Code一小时能写2000行，如果你不验证，问题会在后面以更大的成本爆发出来。

我的做法是：**每完成一个功能模块，立刻打开浏览器或跑测试。发现问题立刻修，不要积累。**

四、不要在一个session里做太多不相关的事

Claude Code的每个session有上下文窗口限制。如果你在一个session里又改前端又调后端又配部署又修bug，到后面上下文会变得很混乱，Claude的回答质量会明显下降。

实际开发中，我通常这样分session：

- Session 1：项目初始化 + 基础架构
- Session 2：核心后端逻辑
- Session 3：前端页面和交互
- Session 4：测试和bug修复
- Session 5：部署和CI/CD

每个session专注做一类事情。session之间靠CLAUDE.md和代码本身传递上下文。这比在一个超长session里硬撑要高效得多。

五、产品感知才是你最大的杠杆

这条是最重要的。

Claude Code能帮你写代码、调UI、配部署、修bug。但它帮不了你决定：这个产品到底应该解决什么问题？目标用户是谁？什么功能该做什么功能该砍？

这些是产品判断。来自你对用户的理解、对市场的感知、对自己能力的诚实评估。AI能让你的执行速度提升10倍，但方向错了，你只是以10倍速度走向错误。

小猫补光灯能成功，不是因为它的代码写得好（代码非常普通），而是因为它精准地击中了一个真实需求：想自拍好看，但不想下载复杂的修图App。一个简单的补光功能，解决一个简单的问题。

> **一人公司的产品节奏：**想法→1天做出MVP→自己用3天→找10个人测试→根据反馈迭代→觉得还行就上架→数据说话。这个循环里，Claude Code覆盖的是「1天做出MVP」和「根据反馈迭代」这两步。其他步骤是你的判断力在工作。

我踩过的坑，你别踩了

列几个在实际产品开发中经常遇到的问题：

陷阱	表现	解决方案
需求膨胀	做着做着不断加功能，永远做不完	回SPEC.md，不在规格内的功能记在todo里，不在当前session做
上下文污染	session越来越长，Claude开始忘记之前的代码结构	及时开新session，让CLAUDE.md和代码库承载上下文
不验证就继续	让Claude连续写了10个文件，最后发现第2个就有bug	每完成一个模块必须验证，宁愿慢一点
环境变量混乱	本地能跑，部署后各种undefined	在CLAUDE.md里列出所有环境变量，部署前用checklist确认
过度依赖AI判断	Claude说「这个方案最好」就采纳，不思考是否适合自己的场景	AI给方案，你做决策。特别是架构选型，永远自己拍板

提前知道这些坑，能帮你省掉不少返工时间。

到这里，你已经具备了独立用Claude Code构建产品的能力。最后一章，聊几个更根本的问题。

§09b 踩坑指南：AI编程的边界在哪

Pitfalls & Boundaries

前面讲了很多Claude Code能做什么。这一章讲它做不好什么、你会在哪里栽跟头、以及怎么避免。全是真实经历，不是理论推演。

关于「遗忘」：你说的话，它真的会忘

这是我踩得最早也最疼的坑。

有一次我在写一篇长文章，前面几轮对话里反复强调「不要用破折号，不要用'说白了'这类套话」。Claude前几轮确实照做了，但聊了大约40分钟之后，它又开始疯狂用破折号。我翻回去看，发现我的指令已经被上下文压缩吃掉了。

这不是偶发事件。Claude Code的上下文窗口虽然有1M token，但长对话会触发自动压缩（compaction），压缩是有损的。你早期说的话，在后期可能只剩一个模糊的影子。

教训：重要的规则，永远写进CLAUDE.md，不要只在对话里说。对话会被压缩，CLAUDE.md每次启动都会重新读取。这也是为什么我后来把所有写作规则都做成了Skill，而不是每次口头交代。

关于「自信地错」：它不会说不知道

有一次我让Claude Code帮我查一个AI产品的最新定价，它非常自信地给了我一组数字。我直接写进了文章里。发布之后读者指出：定价不对，那是3个月前的旧价格。

Claude不会说「我不确定」。它会基于训练数据给你一个看起来很合理的答案，哪怕那个答案已经过时了。对于AI工具的信息，3个月就是一个生命周期。

教训：任何涉及具体数据、日期、定价、产品特性的内容，必须用WebSearch验证。不要因为Claude说得很自信就直接用。这条规则我现在写在了CLAUDE.md的「绝对原则」里，排第一。

关于「Token焦虑」：它可以非常贵

Agent Teams是我用过最强大的功能之一，但也是最贵的。每个Teammate维护独立的上下文窗口，Token消耗大约是普通会话的7倍。有一次我用Agent Teams并行写6章橙皮书，一个下午花了大概$50。

还有一个容易忽略的成本来源：MCP服务器。每个MCP服务器的工具定义都会占用上下文空间。我有段时间同时挂了8个MCP服务器，后来发现光工具描述就吃掉了上下文的15%，等于每次对话的有效空间少了15%。

教训：

- 用 `/cost` 和 `/context` 定期检查消耗

- 不用的MCP服务器及时关掉
- 大任务拆成小Session，比一个巨长的Session更省钱也更稳定
- Agent Teams只在真正需要并行的时候用，不要为了炫技

关于「AI味」：写出来的东西一眼假

这可能是我花时间最多去解决的问题。Claude写的中文有一些非常明显的特征：

- 破折号（——）用得极其频繁，正常人写文章很少用
- 排比三连：「它能XXX，它能YYY，它能ZZZ」
- 假设性开头：「想象一下……」「如果你曾经……」
- 套话连击：「说白了」「简单来说」「换句话说」
- 结尾必升华：「让我们一起拥抱这个充满可能性的未来」

我第一次把Claude写的文章直接发出去，读者反馈就是两个字：AI味。后来我花了好几个月摸索出一套三遍审校流程，目标是把AI检测率从60%以上降到30%以下。这个流程现在是一个Skill（`/proofreading`），每篇文章必过。

教训：AI写初稿没问题，但绝对不能直接发。至少过一遍降AI味的审校。如果你写的是给人看的内容（不是代码），这一步比写初稿本身更重要。

关于「走偏」：你不盯着它就跑远了

Claude Code有一个特性：给它一个模糊的需求，它会自己「发挥」。听起来像优点，实际上经常变成坑。

有一次我让它「优化一下这个页面的样式」，本意是调几个间距和字号。它理解成了重构整个CSS文件，把我之前精心调过的排版全改了，还顺手加了一堆「改进」。我花了半小时才回退到之前的状态。

还有一次让它「修一个小bug」，它不仅修了那个bug，还「顺便」重构了周围的代码，引入了两个新bug。

教训：

- 需求描述要具体：「把标题字号从16px改成18px」比「优化一下标题」好一百倍
- 给明确的停止条件：「测试通过就停」「只改这一个文件」
- 复杂改动先用Plan模式讨论方案，确认后再执行
- Git是你的安全网：每次大改动前确保有干净的commit可以回退

关于「依赖」：你可能会丧失判断力

这个坑比前面几个更隐蔽。

用了半年Claude Code之后，我发现自己有时候会不加思考地接受它的方案。它说用方案A，我就用方案A。为什么？因为之前它给的方案大多数是对的，所以我开始偷懒，不去想「为什么是A不是B」。

直到有一次它建议我在一个项目里用了一个我完全不了解的技术栈，做出来发现性能很差，但我已经投入了两天的工作量。如果我在最初花10分钟自己想一下技术选型，这个弯路完全可以避免。

教训：Claude Code是工程师，不是产品经理。执行层面可以完全信任它，但方向性决策（做什么、用什么技术、面向谁）必须你自己来。Anthropic的趋势报告说得很实在：工程师60%的工作使用AI，但仅0-20%的任务可以完全委托。那80%需要人类判断的部分，恰恰是价值最高的部分。

关于「不是银弹」：它真的搞不定的事

说了这么多踩坑的，也列一下我目前发现Claude Code确实做不好的事情：

领域	具体表现	我的应对
复杂UI微调	像素级的视觉调整很难通过文字描述准确传达	截图给它看 + Computer Use让它直接看屏幕
长期一致性	跨多次Session的风格/规范一致性容易丢失	写进CLAUDE.md，用Skill固化规则
品味和审美	它能执行设计，但不能判断什么是好设计	你定方向，它来执行
最新信息	训练数据有截止日期，3个月前的信息可能已过时	涉及时效信息必须WebSearch验证
多人协作场景	它不理解团队动态、人际关系、组织政治	技术决策可以委托，人的决策自己来
极端性能优化	常规优化没问题，但极致性能调优需要深度理解	让它做第一遍，关键路径自己审查

心态建议：把它当初级工程师

用了一年多Claude Code，我觉得最健康的心态是：**把它当一个极其勤奋但偶尔粗心的初级工程师。**

它打字极快，不会抱怨加班，能同时处理多个任务。但它偶尔会犯低级错误，需要你检查。它不会主动告诉你「这个方向可能有问题」，你得自己判断。它执行力极强但缺乏全局视野，你得给它方向。

如果你带着这个预期去用，你会发现它超出预期的时候远多于让你失望的时候。

> **从央视采访里引一句我自己说的话：**手搓经济的本质，不是所有人都要去做App、去当赛博个体户。它的意义是：门打开了。你可以选择走进去，也可以不走，但至少这扇门以前是关着的，现在开了。用好Claude Code也是一样：知道它的能力边界，才能真正用好它的能力。

§10 心智模型与持续进化

Mental Models & Continuous Evolution

工具会过时，功能会更新，但好的思维方式不会贬值。最后一章退后一步，聊几个我觉得比具体操作更重要的东西。

三层模型：你的时间该花在哪

讲了这么多具体操作，现在值得建立一个全局视角。Claude Code的所有能力，其实可以归入三个层次：

Prompt层是你在终端里输入的每一句话。「帮我加一个登录页面」「这个bug修一下」。大多数初学者的全部交互都停留在这一层。它有效，但每次都需要你开口，每次都从零开始。

Context层是Claude在回答你之前已经「看到」的所有信息。CLAUDE.md文件、项目的文件结构、git提交历史、package.json里的依赖列表。这些信息不需要你每次重复，Claude会自动读取。§05讲的CLAUDE.md，本质上就是在优化这一层。

Harness层是你构建的自动化环境。Skills让你把常用工作流封装成可复用的指令；Hooks让特定事件自动触发操作；MCP连接外部服务；Agent Teams让多个Claude实例并行协作。这一层的特点是：一旦搭好，它就一直在工作，不需要你每次手动触发。

> **一个比喻：**Prompt是你开口说话，Context是你提前准备好的PPT，Harness是你搭建的整个舞台。观众（Claude）的表现，取决于这三层的综合质量。

初学者把所有精力都花在Prompt层，反复琢磨措辞、研究提示词技巧。这没有错，但天花板很低。高手的做法是：**尽量把信息沉淀到Context层，把重复劳动交给Harness层，只在Prompt层处理真正需要临时决策的事情。**

层次	投入方式	回报特征
Prompt	每次对话都要重新投入	一次性回报
Context	写一次CLAUDE.md，持续生效	复利回报
Harness	搭一次自动化流程，永久运行	指数回报

如果你读完这本手册只记住一件事，就记住这个：**把时间花在构建Context和Harness上，而不是优化Prompt。**

Harness Engineering实战：怎么搭你自己的工具链

三层模型是理论框架，但「具体该怎么搭」可能才是你关心的。这里分享一个从空白到完整Harness的实际路径。

第一步：从CLAUDE.md开始（Context层）

新项目第一天，创建 `CLAUDE.md`。不用写很多，从三件事开始：

```
# 项目名称

## 技术栈
- Next.js 15 + TypeScript + Tailwind CSS
- PostgreSQL + Drizzle ORM

## 约定
- 组件放 src/components/，按功能分子目录
- API路由放 src/app/api/
- 提交信息用中文，格式：类型：描述

## 已知坑
- Drizzle的migrate命令需要先export DATABASE_URL
- 部署到Vercel时env变量名不能有下划线开头
```

第三个部分「已知坑」特别重要。每次你踩到坑，告诉Claude「记住这个」，它就会写进CLAUDE.md。下次再遇到同样的场景，Claude会主动避开。这就是Boris说的「犯错→记录→迭代」飞轮。

第二步：把重复操作变成Skill（Harness层入门）

用了一两周后，你会发现自己反复对Claude说类似的话。比如每次写完代码都要说「跑一下测试，lint一下，然后commit」。

这就是你的第一个Skill。在 `.claude/skills/ship/SKILL.md` 里写：

```
---
description: 提交代码的标准流程
---
1. 运行所有测试，确保通过
2. 运行 eslint --fix 格式化代码
3. git add 变更的文件（不要add .env等敏感文件）
4. 生成简洁的commit message并提交
5. 如果有远程分支，push到远程
```

以后输入 `/ship`，五步自动执行。

第三步：加Hooks保障确定性

Skill是建议，有时候Claude会忘。对于「绝不能忘」的事情，用Hook。

我自己有一个PostToolUse hook，每次Claude编辑TypeScript文件后自动跑类型检查。还有一个PostCompact hook，上下文被压缩后自动注入三条核心规则，防止Claude在长会话中「失忆」。

第四步：用MCP连接外部世界

当你的项目需要和外部服务交互时，接MCP。我最常用的三个：

- **浏览器MCP** — 让Claude直接操作Chrome，截图、填表单、读取网页内容
- **飞书MCP** — 创建文档、发消息、管理知识库
- **文件系统MCP** — 操作本机文件，适合跨项目的工作流

一个完整的Harness长什么样

以我自己的写作项目为例，经过半年迭代，最终形成了这样的结构：

```
CLAUDE.md ← 路由器：根据任务关键词分发到子目录 01-公众号写作/ CLAUDE.md ← 写作风格、审校规则、发布流程 项目/2026.03-某项目/ README.md ← 项目状态和文件说明 .claude/ skills/ huashu-proofreading/ ← 三遍审校 huashu-research/ ← 结构化调研 huashu-image-upload/ ← 配图+上传图床 huashu-feishu/ ← 飞书文档操作 ... (60+ skills) settings.json ← Hooks配置 .mcp.json ← MCP服务器配置
```

根目录的CLAUDE.md不超过8KB，它的职责是「路由」：看到「写文章」这个关键词就去读公众号写作的CLAUDE.md，看到「做视频」就读视频创作的CLAUDE.md。这样每个工作区都有自己的规则，互不干扰。

这个Harness不是一天搭好的，是半年里一条规则一条规则积累出来的。重点不是规模，而是**每次遇到重复、遗忘、出错的时候，把解决方案固化到Context或Harness层**。日积月累，你的Claude Code就会变成一个越来越了解你、越来越少出错的协作者。

> **延伸阅读：**如果你对Harness Engineering的理论和更多案例感兴趣，我写了一本专门的橙皮书「Harness Engineering」，详细拆解了这个概念的起源、OpenAI/Anthropic/Stripe等团队的实践、以及从零搭建Harness的完整方法论。

引擎盖下的Claude Code Under the Hood

用了这么久Claude Code，你有没有好奇过：当你按下回车之后，到底发生了什么？

我花了一些时间研究Claude Code的内部架构。不是为了炫技，是因为理解机制之后，很多之前困惑的现象突然就说得通了。为什么有时候它会「绕弯路」？为什么`/compact`之后它偶尔会忘记细节？为什么Auto模式下有些操作直接放行，有些却要你确认？这些都不是随机行为，背后有清晰的设计逻辑。

核心循环：Think → Act → Observe → Repeat

Claude Code的心脏是一个叫TAOR的Agent循环。每次你输入一个任务，它不是直接生成一整段代码然后扔给你。它做的事情像这样：

先思考当前状态，决定下一步做什么；然后调用一个工具执行操作（比如读一个文件、运行一条命令）；观察返回的结果，判断任务是不是完成了；没完成就回到Think继续循环。整个过程可能转几十圈才停下来。

这就解释了为什么Claude有时候看起来「绕弯路」。它不是一个从输入到输出的直线程序，它是一个不断试探和调整的循环体。每一步都在根据新的观察做决策。有时候它试了一个方案发现不行，回退换另一条路，这其实是设计的一部分，不是bug。

也正因如此，**给Claude明确的验证标准特别重要**。循环需要一个停止条件。如果你的需求描述模糊，它不知道什么时候算「做完了」，就会不停地循环下去，改来改去。告诉它「测试通过就停」或「生成文件就行」，它收敛的速度会快很多。

技术栈：终端里的React

一个有趣的事实：你在终端里看到的Claude Code界面，其实是React组件渲染的。

Claude Code运行在Bun上（不是Node.js），用React的Ink框架来渲染终端UI。全部用严格模式TypeScript编写，Schema验证用的是Zod。入口文件压缩后785KB，对一个终端工具来说体量不小，但也说明了它的功能密度。

为什么这个信息有用？因为它解释了Claude Code为什么能有那么丰富的交互体验。权限确认弹窗、多行代码高亮、进度指示器，这些在传统终端工具里很难做到的东西，用React的组件模型就自然了。你感受到的「流畅」不是错觉，是工程选型的结果。

40+工具，4个能力原语

Claude Code内部有40多个工具，每个都有独立的权限控制。但如果你退后一步看，所有能力其实归结为4个原语：

原语	做什么	典型工具
Read	读文件、读代码、搜索内容	Read、Grep、Glob
Write	写文件、编辑代码	Write、Edit
Execute	运行命令、执行脚本	Bash
Connect	连接外部服务	MCP工具、WebFetch

这个设计挺巧妙的，关键在Bash工具。它是一个万能适配器，让Claude能使用人类开发者的一切命令行工具。不需要给每种编程语言做专门集成，不需要为每个框架写插件。`npm install`、`python test.py`、`git push`，通过Execute + Bash就能操作一切。**这也是Claude Code能在几乎任何技术栈的项目里工作的原因**，不像某些IDE插件只支持特定语言。

上下文压缩：为什么长对话会「遗忘」

你可能遇到过这种情况：和Claude聊了很久之后，它突然忘了你之前说过的某个要求。或者你手动执行`/compact`之后，它对某些细节变得模糊了。这不是它在敷衍你。

当上下文窗口快满的时候，系统会把整个对话历史压缩成一段摘要文本。这段摘要成为下一轮对话的起点，之前的原始对话就丢掉了。**压缩是有损的**。核心信息会保留，但具体措辞、边角细节、你当时的语气暗示，这些很容易在压缩过程中丢失。

更要命的是：长会话如果经历了多次压缩，**信息损失会累积**。每压缩一次就损失一点，几次之后，最早的上下文可能只剩一个模糊的影子。

> **核心建议**
>
> 实操建议：重要的约束和要求，写进CLAUDE.md而不是只在对话里说一次。对话会被压缩，但CLAUDE.md每次都会重新读取。这也呼应了前面「三层模型」里的结论：把信息沉淀到Context层。

权限系统：不只是Yes/No

Auto模式背后不是简单的全放行。系统内部有一个分类器，把每个操作的风险分成LOW、MEDIUM、HIGH三级。读文件通常是LOW，直接放行；写配置文件是MEDIUM或HIGH，需要你确认。

有些文件被硬编码为受保护状态：`.gitconfig`、`.bashrc`、`.zshrc`这些系统级配置，无论什么权限模式都会额外小心。甚至还有路径穿越攻击的防御机制，防止恶意代码通过unicode字符或大小写混淆绕过权限检查。

每次弹出权限确认时，你看到的那段解释文字不是预设的模板，是实时生成的。系统会单独调用一次LLM来生成这段说明。所以每次的措辞都略有不同，这不是不稳定，是设计如此。

自动记忆维护

Claude Code有一个后台子代理，会定期整理你的记忆文件（也就是CLAUDE.md和相关配置）。它分四步走：审阅现有内容、提取新的有用信息、整合重复条目、修剪过长的部分。目标是把记忆保持在合理大小内，大约200行左右。

这就是为什么长期用Claude Code之后，你会觉得它越来越「懂你」。不完全是模型变聪明了，而是**你的偏好、习惯、项目上下文，都被这个记忆系统慢慢积累和维护着**。

> 理解Claude Code的内部机制不是为了把它当黑盒拆开。而是当你知道循环怎么转、上下文怎么压缩、权限怎么判断之后，你就能更好地和它协作。就像开车不需要懂发动机原理，但懂了之后你会知道什么时候该换挡。

身份在变：从写代码到构建产品

这个变化比大多数人预期的要快。

Boris Cherny，Claude Code的创建者，公开说过自己超过90%的代码都由Claude Code生成。他的日常更多是：描述需求、审查输出、做架构决策。他有句话挺有意思：「我现在的工作更像是一个有技术判断力的产品经理。」

我自己的经历更极端。我从未手写过代码，所有产品都是用AI构建的，包括小猫补光灯（AppStore付费榜Top 1）。很多人听到会觉得不可思议，但真正用过Claude Code你就知道这完全合理。决定一个产品好不好的，从来不是代码有多精妙，而是需求定义得有多准确、用户体验有多流畅。

这意味着关键能力正在发生转移：

旧能力（重要性下降）

- 语法熟练度
- 框架API记忆
- 手动调试技巧
- 代码模板积累

新能力（重要性上升）

- 需求拆解能力
- 架构判断力
- 输出质量评审
- 产品品味

注意，我说的是「重要性下降」而不是「没用」。理解代码仍然有价值，它能帮你更好地描述需求、更准确地评审输出。但你不再需要能从零手写一个完整应用，你需要的是能判断一个应用写得好不好。

这个转变的核心问题是：**从「怎么写」到「写什么」。**

以前，你可能花80%的时间在「怎么实现这个功能」上，20%在「应该做什么功能」上。现在比例倒过来了。Claude Code解决了「怎么写」的问题，但「写什么」这个问题，它帮不了你太多。你需要自己想清楚：这个产品解决什么问题？目标用户是谁？核心体验是什么？哪些功能必须有，哪些可以砍掉？

> **核心建议**
>
> 如果你正在焦虑「AI会不会取代我」，可以换个角度想：学会定义需求、设计交互、评审质量。这些能力不会因为AI变强而贬值。

迭代这么快，怎么跟

回看Claude Code的功能时间线，迭代速度确实很快：

- 024. MCP协议发布，Claude Code获得连接外部服务的能力
- 25.(公开发布Beta版，从内部工具变成公共产品
- 25.(GA正式发布，稳定性大幅提升

- 25. SubAgents上线，Claude可以启动子进程并行工作
- 25. Hooks机制引入，事件驱动的自动化成为可能
- 25. Skills系统发布，社区可以共享和复用能力包
- 26. Agent Teams正式推出，多Agent协作进入实用阶段
- 26. Computer Use上线，Claude获得操作屏幕的能力；Voice Mode让你对着终端说话
- 26. 源码意外公开，社区首次窥见Agent系统的完整架构

平均每2个月一个大功能。这意味着你手上这本手册的某些具体操作步骤，可能在3个月后就需要更新了。

怎么跟上？我推荐几个稳定的信息渠道：

官方第一手信息：

- Claude Code官方changelog：每次更新都有详细说明
- Anthropic官方博客：重大功能发布会配深度文章
- Anthropic Academy：十余门免费课程，覆盖从基础到进阶

创建者和团队的分享：

- Boris Cherny的X账号（@bcherny）：Claude Code创建者，经常分享使用技巧和设计思路
- howborisusesclaudecode.com：Anthropic官方的实践指南页面
- 「How Anthropic Teams Use Claude Code」白皮书：官方团队的真实工作流

高质量播客访谈（了解设计哲学）：

- Lenny's Podcast：Boris深度谈产品设计和AI编程的未来
- Pragmatic Engineer：技术视角的深度对话
- YC Lightcone：创业者视角，聊AI工具如何改变构建产品的方式

> **注意**
> 不要试图跟踪每一个小更新。时间应该花在用工具构建东西上，不是花在研究工具本身上。每月花30分钟浏览一次changelog就够了。

真正该关注的不是具体功能变化，而是方向。过去一年半的演进，有三条线索始终没变：

1. **自主性持续增强**。从需要你逐步指令，到能自主规划和执行。
2. **上下文窗口持续扩大**。从8K到200K到1M，Claude能「看到」的项目规模越来越大。
3. **协作模式持续丰富**。从单Agent到SubAgents到Agent Teams，多Agent协作越来越自然。

这意味着你今天学的「怎么和AI协作」不会过时。具体命令可能变，但「描述需求→审查输出→迭代改进」这个核心循环，在可预见的未来不会变。

推荐资源

如果你读完这本手册想继续深入，以下是我精选的资源清单。不多，但每一个都值得花时间。

必读

资源	类型	为什么推荐
Claude Code Best Practices	官方文档	所有技巧的权威来源，定期更新
DeepLearning.AI x Anthropic系列课程	视频课程	Andrew Ng团队和Anthropic联合出品，体系化学习
Anthropic Academy	免费课程	十余门免费课程，覆盖Prompt Engineering到Agent开发
How Anthropic Teams Use Claude Code	白皮书	官方团队的真实工作流，不是理论是实践

进阶

资源	类型	为什么推荐
awesome-claude-code	GitHub仓库	社区整理的插件、Skills、最佳实践合集
Claude Code Ultimate Guide	社区文档	实战技巧汇总，很多是官方文档没覆盖的边界场景
howborisusesclaudecode.com	官方页面	Boris本人的完整工作流，持续更新
Boris on Lenny's Podcast	播客	「coding is solved之后会发生什么」深度对话
Boris on Pragmatic Engineer	访谈	从side project到核心工具的演变过程

最后

写这本手册的时候，我时不时会想：AI编程的终局是什么？

想了很久也没想明白。但有一件事我比较确定：过去一年多，我用Claude Code构建了十几个产品，从iOS应用到Chrome扩展到PDF生成工具。每一个产品最终的质量，都不是由AI的能力上限决定的，而是由我对「什么是好的」的判断决定的。

AI可以在30秒内写出一个登录页面。但是否需要登录页面、登录后应该看到什么、什么样的体验才算流畅，这些只能由人来判断。

所以我的建议就一条：别花太多时间研究工具，去构建东西。找一个你真正想解决的问题，打开终端，开始和Claude对话。遇到卡住的地方翻翻这本手册，翻完继续做。

从想法到产品的距离，现在短到你可能还不太适应。

§11 实战项目：Chrome扩展

Hands-on Project: Build a Chrome Extension

从零开始用Claude Code做一个Chrome浏览器扩展。一个真正能用、有人在用的产品，远超Hello World级别。你会经历从需求分析到打包安装的完整过程，以及最真实的踩坑和重构。

我做B站UP主运营助手插件，最早纯粹是因为懒。每天几十条评论需要回复，每次打开B站后台，一条一条点回复、打字、发送。重复了两周之后我实在受不了了，打开终端跟Claude说：「帮我做一个自动回复B站评论的Chrome扩展。」

第一版大概两个小时就跑起来了。能用，但代码全堆在一个content.js里，1211行，任何一个改动都像在拆炸弹。后来花了几天做v3.0重构，content.js缩到175行，业务逻辑全拆进7个独立模块。这个过程比做出第一版学到的东西多得多。

这一章我会带你走一遍完整流程。目标是「能维护」，光「能跑」远远不够。

为什么是Chrome扩展

学完前面所有章节之后，Chrome扩展是一个很合适的进阶项目：

技术门槛适中。不需要学Swift或者搭后端。HTML + CSS + JavaScript，Claude Code最擅长的技术栈。Manifest V3的规则虽然比V2复杂，但Claude对MV3的理解非常到位，你只需要描述功能，它会自动处理Service Worker、权限声明这些细节。

有真实使用场景。你每天都在用浏览器。一个好的扩展能改变你的日常工作流。相比做一个永远不会打开第二次的todo list，做一个自己天天用的工具，学习动力完全不同。

反馈循环极短。改完代码，到Chrome扩展管理页点「重新加载」，刷新页面就能看到效果。不需要编译，不需要部署，不需要等审核。这种即时反馈让你可以快速迭代。

Phase 0：需求分析——你到底想让它做什么

我一开始犯的错误是直接说「做个自动回复插件」。结果Claude给我做了一个非常简单的版本：检测到新评论就回复「谢谢支持」。能用，但完全不是我要的。

后来我学会了先用Plan模式把需求聊清楚。按 `Shift+Tab` 切到Plan模式：

```
我想做一个B站UP主运营助手Chrome扩展，核心需求：
1. 自动扫描视频评论
2. 根据关键词规则匹配评论并自动回复
3. 支持全局规则和单视频规则
4. 已回复的评论不重复回复
5. 有一个简单的管理面板看运行状态

先帮我分析这个需求，给出技术方案。
```

Claude在Plan模式下会输出一份完整的技术方案，包括文件结构、技术选型、难点分析。这时候不要急着说「好，开始做」。仔细看方案，提出你的疑问和修改意见。

> **核心建议**
>
> **经验：**在Plan模式讨论需求的时间，永远比事后推翻重做的时间少。我第一版不讨论直接做，两小时。重构v3.0做了好几天。如果一开始在Plan模式花30分钟把架构想清楚，v3.0的大部分工作都可以避免。

Phase 1：项目初始化

确认方案后，切回正常模式，让Claude创建项目：

```
按我们讨论的方案，创建Chrome扩展项目。先做项目骨架：
- manifest.json (MV3)
- background.js (Service Worker)
- content.js (Content Script)
- popup.html + popup.js (管理面板)
- lib/目录 (业务模块)

权限只申请必须的，不要多申请。
```

Claude会生成一个完整的项目结构。manifest.json是Chrome扩展的灵魂文件，我们的看起来大致这样：

```
{
  "manifest_version": 3,
  "name": "B站UP主运营助手",
  "version": "3.0.0",
  "permissions": ["storage", "alarms", "activeTab"],
  "host_permissions": ["*://*.bilibili.com/*"],
  "background": {
    "service_worker": "background.js"
  },
  "content_scripts": [{
    "matches": ["*://*.bilibili.com/video/*"],
    "js": ["content.js"]
  }],
  "action": {
    "default_popup": "popup.html"
  }
}
```

几个关键决策值得说明：

manifest_version: 3。MV2已经被Chrome废弃了。Claude在这件事上很靠谱，它会自动用MV3的写法。但要注意，网上很多Chrome扩展教程还是MV2的，如果你让Claude参考那些教程，可能会混用两套API。建议在CLAUDE.md里写清楚：「必须使用Manifest V3」。

Service Worker而不是Background Page。MV3最大的变化是background script变成了Service Worker，意味着它随时可能被浏览器杀死。不能用 `setInterval` 做定时任务，得用 `chrome.alarms` 。Claude知道这个区别，但如果你的CLAUDE.md没提，它偶尔会用旧的写法。

权限最小化。只申请真正需要的权限。Chrome Web Store审核时，权限越多越容易被拒。而且用户看到一长串权限请求也会犹豫。

Phase 2：核心架构，谁管什么

Chrome扩展最容易踩的坑是「代码放错了地方」。MV3有三个运行环境，职责完全不同：

环境	文件	能做什么	不能做什么
Service Worker	background.js	定时任务、全局状态、消息路由	访问页面DOM、读取页面Cookie
Content Script	content.js	操作页面DOM、读取页面上下文	直接调用chrome.alarms等API
Popup/Options	popup.js	用户界面、配置管理	在用户关闭popup后运行

我v1版本的教训就是把所有代码都塞进content.js。扫描逻辑、API调用、规则匹配、回复发送、状态管理，全在一个文件里。1211行。改任何一个功能都要翻半天。

v3的架构是这样的：

```
background.js（指挥中心）
    ├─── 拥有定时扫描循环（chrome.alarms，每分钟一次）
    ├─── 路由所有消息（popup/content/widget → 统一处理）
    └─── 管理全局状态（开关、限流等级等）
        ↕ chrome.runtime.sendMessage
content.js（前线哨兵）
    ├─── 提取当前视频信息（BV号、标题等）
    ├─── 代理API调用（因为需要页面Cookie）
    └─── 转发日志和状态给浮窗
        ↕ 消息通信
lib/（业务大脑）
    ├─── db.js          → 统一数据层，所有存储读写走这里
    ├─── scanner.js     → 评论扫描引擎
    ├─── rules.js       → 规则匹配逻辑
    ├─── api.js         → B站API封装
    ├─── ai.js          → AI生成回复
    └─── rate-limiter.js → 4级限流降级
```

关键设计原则是：**content.js只做它必须做的事**。哪些是它必须做的？读取页面Cookie（background.js在MV3下读不到）、操作页面DOM。除此之外，一切业务逻辑都在background.js和lib/里。

你在Claude Code里可以这样描述这个架构：

```
架构原则：
1. content.js是薄壳，只负责提取视频信息和代理API调用
2. background.js是指挥中心，拥有扫描循环和状态管理
3. 所有业务逻辑放在lib/目录下的独立模块中
4. 数据存储统一走lib/db.js，不在其他文件里直接操作chrome.storage
5. 模块间通过消息通信，不共享状态

把这些写进项目的CLAUDE.md里。
```

> **核心建议**
>
> **经验：**让Claude帮你写项目的CLAUDE.md，把架构原则写进去。以后每次对话Claude都会自动遵守。这就是§05讲的「给AI一张地图」在实战中的应用。

Phase 3：逐个模块开发

有了架构之后，开发过程就变成了逐个模块实现。我建议按这个顺序来：

第一步：数据层（db.js）。这是地基。所有其他模块都依赖它。

```
先做lib/db.js，统一数据访问层。需要这些接口：
- getSystemState() / setSystemState(partial)
- getConfig() / setConfig(partial)
- getRules() / setRules(rules)
- getVideo(bvid) / setVideo(bvid, data)
- markReplied(bvid, rpid)

存储结构：
- 'sys:state' → 系统状态（开关、限流等级等）
- 'sys:config' → 配置（扫描间隔、回复延迟等）
- 'sys:rules' → 规则（全局规则 + 视频专属规则）
- 'v:{bvid}' → 每个视频的独立数据

所有写操作必须是原子的：先读 → 合并 → 写入，防止并发覆盖。
```

从20多个散落的storage key收敛到5个结构化的key，这个改变让后续所有模块都变得简单了。没有「到底数据存在哪里」的困惑，全部走db.js。

第二步：规则匹配（rules.js）。最小的模块，62行。输入是一条评论和一组规则，输出是匹配结果。纯函数，没有副作用，非常好测试。

第三步：API封装（api.js）。把B站的评论获取和回复发送封装成干净的函数。注意B站的API需要页面Cookie认证，所以这些函数实际上会在content.js上下文中执行，通过消息传递结果。

第四步：扫描引擎（scanner.js）。把前面三个模块串起来。核心逻辑是：获取评论 → 过滤已回复的 → 匹配规则 → 生成回复 → 发送。

这里有一个关键设计：依赖注入。scanner.js不直接调用API，而是接收外部传入的函数：

```
async function scanOneVideo(bvid, rules, config, {
  sendReplyFn,         // 发送回复的函数
  fetchCommentsFn,     // 获取评论的函数
  logFn                // 日志函数
}) {
  // 业务逻辑
}
```

为什么这样设计？因为scanner.js是在background.js里运行的，但API调用需要在content.js的上下文里执行。通过依赖注入，background.js可以传入「通过消息转发到content.js执行」的函数，而scanner.js完全不需要知道这个细节。

第五步：限流器（rate-limiter.js）。B站对评论频率有限制。如果你一秒发5条回复，账号会被临时限制。我们的限流策略是4级降级：

等级	回复间隔	扫描间隔	触发条件
normal	5秒	2秒	默认
slow	15秒	5秒	收到频率限制警告
slower	30秒	10秒	连续3次错误
paused	暂停	暂停	账号被限制，1小时后恢复

成功回复会递减错误计数器，自动恢复到normal。像弹簧一样，压下去还能弹回来。

第六步：背景脚本（background.js）。把所有模块组装起来，实现扫描循环。每分钟被chrome.alarms唤醒一次，按优先级扫描视频（当前打开的标签页优先）。

第七步：Content Script和UI。最后做界面。content.js只暴露几个简单方法给浮窗，popup.js做管理面板。

Phase 4：调试技巧

Chrome扩展的调试和普通网页不太一样。几个实用技巧：

Service Worker调试。在 `chrome://extensions` 页面，你的扩展下面有一个「Service Worker」链接，点击会打开独立的DevTools。background.js的console.log会出现在这里，不是在网页的控制台里。

Content Script调试。在网页的DevTools控制台里可以直接看到content.js的日志。但要注意，content.js运行在一个隔离的执行环境中，和网页的JavaScript不共享全局变量。

重新加载。改完代码后，去 `chrome://extensions` 点扩展的刷新按钮。popup和option页面需要关掉重新打开。content.js需要刷新目标网页。Service Worker改动后需要点「Service Worker」旁边的「更新」。

Claude Code可以帮你写调试辅助代码。比如让它在content.js里加一个日志系统，所有关键操作都打日志，方便排查问题。在v3版本里，我们做了一个简单的日志系统，最多保留200条，自动滚动清理，在popup面板里可以看到。

Phase 5：安装和测试

开发阶段不需要发布到Chrome Web Store。在 `chrome://extensions` 打开「开发者模式」，点「加载已解压的扩展程序」，选你的项目目录就行。

测试清单（让Claude帮你生成一份）：

- 基本功能：添加规则 → 扫描评论 → 自动回复
- 去重：同一条评论不会被回复两次
- 限流：快速连续回复会自动降速
- 持久化：关闭浏览器重新打开，状态和数据都还在

- 多标签：同时打开多个视频页面，各自独立运行

从1211行到175行：重构的故事

这个重构过程值得单独讲，因为它是Claude Code辅助重构的一个典型案例。

v1版本能用，但痛点很多。content.js一个文件1211行，API调用、扫描逻辑、规则匹配、UI更新全搅在一起。每次改一个功能都要翻好久找代码在哪。而且MV3的Service Worker可能随时休眠，用setInterval做定时任务很不稳定。

重构不是我一开始就计划好的。是某天我想加一个新功能（AI辅助回复），发现在1211行的content.js里根本无从下手，才决定必须重构。

我跟Claude说：

```
当前content.js有1211行，所有逻辑都在里面。我要重构成：
- background.js作为运行中枢
- content.js变成薄壳（只做视频信息提取和API代理）
- lib/目录放所有业务模块

不要一步到位，分步骤来。先帮我分析当前代码，列出每一块逻辑应该去哪里。
```

Claude给出了一个详细的迁移计划，把1211行代码按功能分成了8块，标注了每块应该去哪个模块。然后我们一步一步执行，每迁移一块都测试一次，确保没有退化。

最后的结果：

文件	行数	职责
background.js	468	消息路由 + 扫描编排
content.js	175	视频信息提取 + 消息桥接
lib/db.js	410	统一数据层
lib/scanner.js	322	评论扫描引擎
lib/api.js	137	B站API封装
lib/ai.js	127	AI回复生成
lib/rate-limiter.js	100	限流降级
lib/rules.js	62	规则匹配
lib/migrate.js	253	v2→v3数据迁移

总代码量反而增加了（从1211行变成了2000+行），但每个文件的职责清晰、可以独立理解和修改。content.js从「什么都做」变成了「只做必须做的事」。重构的意义从来不在于减少代码量，在于降低认知负担。

> **注意**
>
> **踩坑提醒：**重构时最容易犯的错误是「一步到位」。不要让Claude一次性重写所有代码，而要分步骤迁移，每步都确认功能正常。我在v3重构过程中，每迁移一个模块就在浏览器里跑一遍完整的测试流程。慢一点没关系，稳比快重要。

你的Chrome扩展项目建议

如果你想跟着做一个Chrome扩展，不一定要做B站插件。这里给几个方向：

- **网页标注工具**：选中文字高亮标注，保存到本地。涉及content script操作DOM + storage持久化
- **页面翻译助手**：选中段落调用AI翻译。涉及content script + 外部API调用
- **社交媒体定时器**：记录你在各个网站花的时间，超时提醒。涉及background alarms + 多站点content script
- **GitHub增强**：在PR页面显示CI状态、自动添加标签。涉及GitHub API + content script注入

不管做哪个，核心流程是一样的：Plan模式分析需求 → 定义架构（谁管什么）→ 写CLAUDE.md固化架构原则 → 逐模块实现 → 调试测试。

Chrome扩展是一个很好的起点，因为它证明了一件事：Claude Code的能力远不止于写网页。浏览器插件、自动化工具、甚至操作系统级的脚本，只要是代码能解决的问题，Claude Code都能帮你做。接下来两章，我们会进一步拓展这个边界。

§12 实战项目：内容创作自动化

Hands-on Project: Content Creation Automation

Claude Code最被低估的能力，不是写代码。这一章我会展示我每天真正的工作流：用Claude Code做内容创作。从选题、调研、写作、审校到发布，全流程自动化。当你把Claude Code当作「通用Agent」来用，它的价值会翻好几倍。

我做自媒体，大概30%的时间在用AI做开发，70%在做内容：写公众号文章、做小红书图文、写视频脚本、做调研报告。很长一段时间，我做开发用Claude Code，写文章用ChatGPT或Claude网页版。直到有一天我意识到一个问题：我在Claude Code里积累的所有偏好、规则、风格要求，换到网页版就全丢了。每次开新对话都要重新交代一遍「不要用破折号」「用中文引号」「搜索最近三个月的信息」。

那天我把写作规则写进了CLAUDE.md。然后又加了几个Skill。再然后搭了一套完整的内容生产线。现在回头看，这可能是我用Claude Code做过的最有价值的事情。不是某个具体的代码项目，是这套内容自动化系统。它每天都在帮我省时间。

这一章不是理论，每一步都是我自己每天在用的东西。你可以完全照搬，也可以按自己的需求改。

第一步：CLAUDE.md路由系统

如果你只做一个项目，一个CLAUDE.md就够了。但如果你有多种工作场景，比如同时做写作、开发、做视频，你需要一个路由系统。

所谓路由系统，就是一个根目录的CLAUDE.md充当「交通指挥」，根据你说的话判断你要做什么，然后去读取对应的子目录CLAUDE.md。

我的项目结构：

```
写作/
├── CLAUDE.md                  ← 路由器（~8KB）
├── 01-公众号写作/
│    ├── CLAUDE.md             ← 公众号写作的完整规则
│    └── 项目/
├── 02-小红书写作/
│    ├── CLAUDE.md             ← 小红书写作规则
│    └── 项目/
├── 03-视频创作/
│    ├── CLAUDE.md             ← 视频脚本规则
│    └── 项目/
├── 04-写作参考/
│    └── SHARED-RULES.md       ← 跨工作区共享规则
├── .claude/skills/            ← 27个Skill
└── 10-橙皮书/                 ← 你正在看的这本书
```

根CLAUDE.md里最核心的是路由表：

```
## 工作区路由

收到任务后，先判断工作区，读取对应CLAUDE.md：

| 关键词               | 工作区     | 读取文件                     |
|--------------------|-----------|---------------------------|
| 写文章、商单、公众号  | 公众号写作  | /01-公众号写作/CLAUDE.md     |
| 小红书、笔记、图文    | 小红书写作  | /02-小红书写作/CLAUDE.md     |
| 视频脚本、视频创作    | 视频创作    | /03-视频创作/CLAUDE.md       |
| 橙皮书、epub         | 橙皮书      | /10-橙皮书/CLAUDE.md         |

路由原则：任务模糊 → 询问确认。涉及多工作区 → 依次读取。
```

当我说「写一篇关于Claude Code的公众号文章」，Claude Code会：

1. 读取根CLAUDE.md，看到关键词「公众号」→ 匹配到公众号写作
2. 自动读取 `/01-公众号写作/CLAUDE.md`，加载写作规则
3. 按照公众号的风格规范开始工作

这样做的好处是：根CLAUDE.md保持精简（~8KB），不会浪费上下文窗口；每个工作区的规则可以独立维护，互不干扰；新增工作区只需要加一行路由。

> **核心建议**
> **你也可以这样做：** 如果你同时管理多个项目（工作项目、副业项目、学习笔记），可以创建一个统一的工作目录，根CLAUDE.md做路由，每个项目有自己的规则。不需要完全照搬我的结构，核心思想是：**一个入口，多个出口**。

第二步：共享规则，跨项目的一致性

有些规则不属于任何一个工作区，而是所有内容都需要遵守的。比如：

- 搜索信息要用最近3个月的来源
- 三遍审校流程（内容审 → 风格审 → 细节审）
- 配图流程（截图 → AI生成 → 上传图床）
- 文件命名规范

这些放在 `SHARED-RULES.md` 里，被所有工作区的CLAUDE.md引用。

其中最值得展开讲的是三遍审校流程，因为这个规则直接影响到输出质量。

第一遍：内容审。检查事实是否准确、逻辑是否通顺、结构是否完整。这一遍不碰文字风格。

第二遍：风格审。这一遍专门检测「AI味」。AI生成的文字有几个明显特征：

AI味类型	举例	怎么改
套话连篇	「在当今时代」「综上所述」「值得一提的是」	直接删掉，不影响意思
AI句式	「不是A，而是B」「不仅A，而且B」	用口语化的转折
过度书面	「具备」「呈现」「彰显」「赋能」	换成日常用词
结构机械	每段都是「首先/其次/最后」	打乱顺序，用叙事串联
态度中立	两边都不得罪的和稀泥	加入明确的个人立场
细节缺失	「很多用户反馈」（具体多少？谁？）	加具体数据和人名

第三遍：细节审。句子长度控制在15-25字、段落间留白、加粗标记（全篇约10处）、引号使用（用「」不用""）。

这三遍审校规则写进SHARED-RULES.md后，不管我在哪个工作区写东西，Claude都会按这个标准检查。而且我可以随时用 `/proofreading` 这个Skill来触发审校，不需要每次手动描述流程。

第三步：创建你的第一个Skill

§07讲过Skill的基本概念。这里用一个真实例子展示怎么从零创建一个Skill。

我做的第一个Skill就是上面提到的三遍审校。创建方法：

```
# 在项目目录下创建skill文件
mkdir -p .claude/skills/huashu-proofreading
# 写SKILL.md
```

SKILL.md的格式：

```
---
name: proofreading
description: |
  三遍审校流程，降低AI检测率到30%以下。
  自动触发：用户说「审校」「降AI味」「太AI了」「改一改」
  手动触发：/proofreading
---

# 三遍审校

## 第一遍：内容审
- 事实核查：所有数据、日期、版本号是否正确
- 逻辑检查：因果关系是否成立
- 结构检查：是否有遗漏的要点

## 第二遍：风格审
检测以下6类AI味（见附表），逐一修改：
1. 套话连篇 → 删除
2. AI句式 → 改为口语化
3. 过度书面 → 换日常用词
4. 结构机械 → 打乱、用叙事串联
5. 态度中立 → 加个人立场
6. 细节缺失 → 补具体数据

## 第三遍：细节审
- 句子长度：15-25字为主
- 段落长度：3-5句
- 加粗标记：全篇约10处
- 引号：使用「」不使用""
- 破折号（——）：全篇最多1-2处
```

写好之后，在Claude Code里输入 `/proofreading`，它就会按这个流程审校当前文档。你也可以直接说「帮我审校一下」，Claude看到关键词「审校」会自动触发这个Skill。

这里有一个重要的心智转变：**Skill是写给AI的，不是写给你的。**你不需要记住三遍审校的所有检查项。你只需要记住「/proofreading」这一个命令。所有细节都被封装进了Skill里，Claude负责执行。

当你养成了「把重复工作封装成Skill」的习惯，Skill库会迅速增长。我目前有27个Skill，覆盖内容创作的每一个环节：

类别	Skill名称	干什么
写作	/proofreading	三遍审校
	/article-edit	文章编辑（带进度追踪）
	/topic-gen	生成3-4个选题方向
	/article-to-x	公众号文章改写为X推文
视频	/video-outline	视频脚本大纲（含标题策略）
	/script-polish	脚本润色
	/danmaku-gen	生成互动弹幕文案
调研	/research	结构化调研（自动存档）
	/info-search	信息搜索（带来源验证）
	/material-search	搜索个人素材库
出版	/book-pdf	橙皮书全流程构建
	/md-to-pdf	Markdown转PDF
配图	/image-upload	AI生图 + 上传图床
	/design	信息图设计
集成	/feishu	飞书文档创建与发送

每个Skill都经过实战打磨。不是我一次性写好27个，而是在日常工作中，每遇到一个重复性操作，就封装成一个Skill。花了大概两个月时间积累到这个数量。

第四步：完整工作流演示，从选题到发布

说了这么多单个组件，我用一个完整的例子把它们串起来。假设今天我要写一篇公众号文章。

10:00 选题。我在终端里打开Claude Code，进入写作项目目录：

```
最近Claude Code更新了一些新功能，帮我想几个公众号选题方向。
```

Claude看到关键词「公众号」，自动路由到公众号工作区，加载写作规则。然后触发 `/topic-gen` Skill，输出3-4个选题方向，每个带标题建议、大纲、工作量评估（⭐ 到⭐⭐⭐）。

我选了第二个方向，工作量⭐⭐。

10:10 调研。

```
选第二个方向。先做调研，搜集Claude Code最近一个月的重要更新和用户反馈。
```

触发 `/research` Skill。Claude会：

1. 立刻创建一个调研文件 `_knowledge_base/research-claude-code-更新-20260403.md`
2. 每搜索一轮就把结果追加写入文件（防止对话中断丢数据）
3. 搜3轮后输出阶段性总结
4. 最后给出结构化报告：关键事实、信息来源、空白点、写作建议

这个「边搜边存」的设计是被逼出来的。之前有一次调研做到一半，会话因为上下文太长被压缩了，前面搜到的信息全丢了。从那以后，每个调研Skill都强制要求实时写入文件。

10:30 写草稿。

```
调研差不多了。根据调研结果，写一篇公众号文章。
要求：
- 文章保存到 项目/2026.04-Claude-Code更新/草稿.md
- 3000字左右
- 有明确的个人观点，不要写成新闻稿
```

Claude会创建项目目录，然后开始写作。写完保存到md文件里。注意，根CLAUDE.md里有一条偏好：「文章必须写入md文件，不要直接写在回复里」。这是因为回复内容在对话结束后就没了，但md文件会一直在。

11:00 审校。

```
/proofreading 草稿.md
```

三遍审校自动执行。Claude会读取文件，按照三遍流程逐项检查，输出修改建议。我确认后它直接在文件上改。通常一次审校能发现10-20处AI味问题。

11:20 配图。

```
给这篇文章配一张封面图。
```

触发配图流程。Claude会根据文章内容生成图片提示词，调用AI生图，生成后上传到图床，把图片链接插入到文章对应位置。整个过程自动化，我只需要看看生成的图满不满意。

11:30 发布。

```
文章审校完了，发到飞书文档。
```

触发 `/feishu` Skill。Claude把md文件转换成飞书文档格式，通过API创建文档，设置权限，发给我。我在飞书里做最后一轮人工检查，然后复制到公众号编辑器发布。

从选题到发布前的文档就绪，大概一个半小时。如果没有这套系统，同样的工作量大概要半天。

第五步：进阶，多Agent并行

上面的工作流是串行的：一步做完再做下一步。但实际工作中，很多步骤可以并行。比如写这本书的时候，我同时开了4个Agent：

```
# 终端1：调研Agent，查找最新的Claude Code更新
claude -p "调研Claude Code最近三个月的重大更新..." --dangerously-skip-permissions

# 终端2：写作Agent，写§07扩展能力章节
claude -p "根据以下大纲写§07..." --dangerously-skip-permissions

# 终端3：审校Agent，审校已完成的§05章节
claude -p "审校fragments/part5-claude-md.html..." --dangerously-skip-permissions

# 终端4：配图Agent，为已审校的章节配图
claude -p "为§03章节配一张配图..." --dangerously-skip-permissions
```

每个Agent独立运行，互不干扰。4个Agent同时跑，整本书的构建效率提升了好几倍。API成本大概50美元，比雇一个助理便宜得多。

> **注意**
>
> **并行的前提是任务互相独立。**如果Agent B需要Agent A的输出，就不能并行。比如「先调研再写作」必须串行，但「写§03」和「写§05」可以并行，它们操作不同的文件，不会冲突。

第六步：Skills + Hooks + MCP的配合

到这里，我们已经用到了Skills（封装重复流程）和CLAUDE.md（规则持久化）。再加上Hooks和MCP，就构成了完整的Harness。

Hooks做什么？Hooks是事件驱动的。比如：

- 每次创建新文件时，自动检查文件名是否符合命名规范
- 每次编辑md文件后，自动运行格式化（统一引号、去多余空行）
- 每次commit前，检查是否有未完成的TODO标记

Hooks的价值在于「防遗忘」。人会忘记检查命名规范，但Hooks不会。

MCP做什么？MCP让Claude Code能连接外部服务。在我的内容创作工作流里，MCP最主要的用途是：

- 连接飞书：直接创建和编辑飞书文档
- 连接浏览器：在Chrome里操作页面、读取信息
- 连接文件系统：读取写作目录之外的参考资料

三者的关系是这样的：

```
CLAUDE.md   → 定义规则和偏好（被动，每次对话自动加载）
Skills      → 封装工作流程（主动触发或关键词触发）
Hooks       → 自动化检查（事件驱动，不需要人参与）
MCP         → 连接外部世界（扩展Claude的能力边界）
```

如果把Claude Code比作一个员工，CLAUDE.md是他的岗位手册，Skills是他学会的技能，Hooks是他养成的好习惯，MCP是他能用的工具。四者叠加，就是一个完整的「AI员工」系统。

搭建你自己的内容自动化系统

你不需要一步到位搭一个27个Skill的系统。我建议分三步走：

第一周：建立CLAUDE.md。先创建根CLAUDE.md，写入3-5条你最在意的规则。比如你的写作风格偏好、文件命名规范、常用的项目结构。不需要写很多，有就比没有强。

第二周：创建第一个Skill。观察一周，看你重复对Claude说的最多的话是什么。把它封装成一个Skill。可能是「帮我审校」，可能是「帮我翻译」，可能是「帮我总结会议记录」。不管是什么，先做一个，用起来。

第三周开始：按需扩展。每遇到一个重复操作，就问自己：「这个值得做成Skill吗？」如果这个操作你一周会做3次以上，答案就是值得。三个月后，你会有10-15个Skill，覆盖你日常工作的大部分重复性操作。

关键不是数量，而是习惯。当你习惯了用Skill封装重复操作、用CLAUDE.md固化规则、用MCP连接外部服务，你的整个工作效率会进入一个新的层次。这就是§10讲的Harness Engineering在实际工作中的样子。

> **写给非程序员读者**
> 这一章的所有内容都不需要你会写代码。创建CLAUDE.md是写规则文档，创建Skill是写操作手册，配置MCP是填几个参数。做内容的、做运营的、做产品的，任何需要重复性输出的工作，这套方法都可以直接用。Claude Code不只是编程工具，它是一个通用的AI工作站。这可能是这本书最重要的一个观点。

§13 实战项目：从零到App Store付费榜第一

Hands-on Project: From Zero to #1 Paid App

这一章讲的不是技术，而是一个完整的产品故事。从女朋友的一句话到App Store付费榜第一，中间只隔了一个周末。但一个小时的开发不是重点，前面五分钟的判断、后面几个月的迭代，才是这个故事真正值得讲的部分。

2024年11月的一个下午，我在录一个Cursor教程视频。女朋友坐旁边帮我检查画面，突然说了一句：「与其做屏幕手电筒，不如做补光卡片呢。」

我当时有点懵。补光卡片是什么需求？在那之前我从来没听过这个词。但我没有忽略这句话，而是打开小红书搜了一下「补光色卡」。

搜索结果让我愣了几秒。好几篇笔记都是十几万赞。有人用纯色图片当补光板自拍，有人分享不同颜色的补光效果对比，有人在找「能调颜色和亮度的补光App」但找不到好用的。我立刻意识到：这是一个真实的高频需求——需求已经被验证了，只是没人做成产品。

五分钟后我打开Cursor（当时我还没开始用Claude Code），用自然语言描述了我想要的功能。一个小时后，小猫补光灯的第一版上了App Store。

比写代码更重要的事：五分钟的判断

我后来复盘这件事，发现最关键的不是开发过程，而是在那五分钟里做对了几个判断：

第一，我没有忽略一个非程序员的建议。女朋友不懂代码，但她懂用户需求。「补光色卡」这个词是她从小红书上学来的，代表了真实用户的真实语言。如果我只跟程序员朋友讨论，大概率不会想到这个方向。

第二，我花了三分钟验证需求。不是拍脑袋觉得「这个应该有人要」，而是打开小红书搜了一下。十几万赞的笔记就是最好的市场调研报告。成本：三分钟。回报：确认了这是一个真实的高频需求。

第三，我选择了最小可行产品。第一版只做一件事：全屏显示一个纯色，可以调颜色和亮度。没有滤镜，没有定时器，没有社交分享，没有会员系统。就是一个能调色的全屏灯。

这个判断过程和你用Claude Code做什么项目的决策逻辑是一样的。技术能力不是瓶颈，Claude Code可以帮你写任何复杂度的代码。真正的瓶颈是：你要做的东西，有人需要吗？

> **给读者的建议**
> 在读这一章之前，先想一个你想做的App。不要想「技术上能不能实现」，先想「谁会用、在什么场景下用、市面上有没有替代品」。如果你能在小红书、B站、X上找到人们在讨论相关需求但没有好产品满足，那就值得做。

用AI写一个iOS App

小猫补光灯最初是用Cursor开发的，那是2024年底，Claude Code还没有现在这么成熟。如果今天重新来做这个项目，我会全程用Claude Code。具体怎么做呢？

创建SwiftUI项目。先用Xcode创建一个空的iOS项目（这一步必须手动，因为Xcode项目的初始化需要图形界面选择配置）。创建好之后，在项目目录下启动Claude Code：

```
这是一个SwiftUI项目，我要做一个补光灯App。核心功能：
1. 全屏纯色显示，用户可以选择颜色
2. 亮度可调（从0到最大）
3. 提供几个预设色卡（暖白、冷白、暖黄、粉色等）
4. 界面要简洁，打开就能用

先看看项目结构，然后帮我实现核心功能。
```

Claude Code会读取项目结构，然后开始写SwiftUI代码。补光灯的核心技术其实很简单，就是用一个全屏的 `Color` View，加上一个颜色选择器和亮度滑块。Claude很擅长这种「目标明确、技术路径清晰」的任务，通常第一版就能跑。

迭代是真正的工作。第一版能用，但远没到「好用」的程度。用户反馈会告诉你还需要做什么：

- 「能不能加个摄像头预览？这样我不用切来切去」→ Pro版的核心功能
- 「颜色选择太麻烦了，能不能做成预设卡片」→ 色卡系统
- 「我想要自定义颜色」→ HSB颜色选择器
- 「亮度太低了」→ 系统亮度API + 屏幕亮度叠加

每一个迭代都是同样的流程：用户反馈 → 在Claude Code里描述需求 → 实现 → 测试 → 提交。

```
用户反馈说需要在补光的同时看到摄像头预览。
帮我加一个功能：屏幕上半部分显示摄像头实时画面，
下半部分显示补光颜色，中间有一个可拖动的分界线。
```

Claude会修改现有代码，加入 `AVCaptureSession` 的摄像头集成。这是一个技术复杂度较高的功能（涉及相机权限、实时预览、画面布局），但Claude Code处理得很好，因为这些都是SwiftUI和AVFoundation的标准模式。

App Store上架：最容易踩坑的环节

写代码可能只占整个过程的30%。剩下的70%是上架相关的工作：开发者账号、证书配置、隐私政策、App审核。

开发者账号。Apple Developer Program，年费688元。必须用真实身份注册。审核大概1-2天。如果你之前没有Apple开发者账号，这是唯一需要等待的步骤。

证书和签名。这是iOS开发最反直觉的部分。你需要创建证书、配置描述文件、在Xcode里设置签名。Claude Code可以帮你理解每一步的含义，但实际操作需要在Xcode和Apple Developer网站上完成。

```
我已经有了Apple Developer账号。帮我梳理一下从开发到上架的完整步骤，
包括证书配置、描述文件创建、Xcode打包设置。
列出每一步的具体操作和可能遇到的坑。
```

Claude会给你一份详细的checklist。虽然它不能替你操作Xcode界面，但它可以告诉你每个选项应该怎么选、为什么这么选。

App审核。Apple的审核比Chrome Web Store严格得多。几个容易被拒的原因：

- 缺少隐私政策页面（即使你的App不收集任何数据，也要有）
- 截图和实际功能不符
- 使用了私有API
- 功能太简单（Apple可能认为「可以用网页实现的就不该做成App」）

小猫补光灯第一次提交就过审了，因为功能明确、界面简洁、没有灰色地带。审核大概24小时。

爆火：运气还是必然

App上架后的第一周，我在小红书发了一篇笔记，内容很简单：展示了补光效果的前后对比。

然后它就炸了。

三天内，那篇笔记获得了118万阅读、7.3万点赞。App下载量突破3万。Pro版（加了摄像头功能，售价6元）冲上了App Store付费榜第一名，待了一个多月。

后来有媒体问我「怎么做到的」。说实话，我自己也有点意外。但复盘之后，有几点是可以总结的：

需求足够刚。自拍补光是一个高频、刚需的场景。不是「有了更好」，而是「没有不行」。小红书上大量的补光色卡笔记证明了这个需求的真实性。

产品足够简单。打开App就是一个全屏色卡。不需要注册，不需要教程，不需要学习成本。这种「打开就能用」的产品最容易传播，因为用户可以在短视频里3秒展示完全部功能。

争议带来流量。很多程序员在评论区质疑「这种App也能卖钱」「一小时做出来的东西也配收费」。这些争议反而帮了大忙，让更多人看到了这条笔记。用户之间的辩论成了免费的流量引擎。

快速迭代建立口碑。App爆火的那两天，我连夜改了三个版本，修复用户反馈的问题、加了最多人要求的功能。同时克制住不去修改小红书笔记、不发新笔记，因为平台算法正在推荐这篇内容，任何干预都可能影响分发。

> **注意**
>
> **务实的期望管理**
>
> 绝大多数手搓产品会无声消亡。你看到的是上排行榜的那几个，背后有成千上万个没人听说过的App。小猫补光灯的成功有很大的运气成分：选对了赛道、赶上了小红书的推荐、产品形态恰好适合短视频传播。这些因素不是每次都能复制的。做产品的正确心态是：做好每一个环节，但不预设结果。

后来的事：从爆款到产品线

小猫补光灯后来的发展超出了我的预期。

Pro版持续更新，加了更多专业功能。基于同一个技术底座，我做了小程序版（覆盖没有iOS设备的用户）和鸿蒙版（覆盖华为用户）。又做了「小猫相册」，一个面向同一批用户群（自拍爱好者）的新产品。

一个1小时做出来的App，最后变成了一条小产品线。这在传统开发模式下几乎不可能，光是写代码就要几个月。AI编程把开发成本压缩到了接近零，让我可以快速试错、快速迭代。

人民日报把这种模式叫「手搓经济」。央视来采访的时候，记者让我现场演示用AI写代码。我打开Claude Code，10分钟做了一个能跑的小产品。据说这可能是Claude Code第一次出现在中国的国家级媒体上。

用Claude Code复刻这个过程

小猫补光灯是用Cursor做的，但核心方法论完全适用于Claude Code。如果你想做一个iOS App，完整的流程是：

第一步：验证需求（30分钟）。在小红书、B站、抖音搜你想做的方向。看有没有人在讨论相关需求。不要看竞品多不多，有竞品说明有市场。要看的是：现有竞品是不是都很差？用户最大的抱怨是什么？

```
我想做一个[你的App想法]。帮我分析一下：
1. 这个需求的真实性（有多少人真的需要）
2. 市面上有哪些竞品，它们的评价如何
3. 我的差异化方向可以是什么
4. 最小可行产品应该包含哪些功能
```

第二步：定义MVP（30分钟）。用Plan模式和Claude讨论最小可行产品。砍掉一切不必要的功能。问自己一个问题：如果这个App只能做一件事，那件事是什么？

第三步：开发（1-4小时）。在Xcode创建项目，然后在项目目录下用Claude Code开发。对于简单的工具类App，通常几个小时就能做到可以跑的程度。

第四步：测试和打磨（1-2天）。自己用，让朋友用，记录所有不顺畅的地方。让Claude Code逐一修复。这一步比开发本身更重要。

第五步：上架（1-2天）。准备App Store素材（截图、描述、关键词、隐私政策），Xcode打包上传，等待审核。

第六步：推广和迭代（持续）。发布后在社交平台分享，收集用户反馈，快速迭代。

六步里面，Claude Code直接参与的是第一步（分析）、第二步（规划）、第三步（开发）、第四步（修复）。第五步和第六步需要你自己在Xcode和社交平台上操作，但Claude Code可以帮你写App描述、准备关键词、分析用户反馈。

iOS开发的CLAUDE.md建议

如果你打算认真做iOS开发，建议在项目的CLAUDE.md里写入以下内容：

```
# iOS项目规则

## 技术栈
- SwiftUI（不用UIKit，除非SwiftUI无法实现的功能）
- 最低支持iOS 16
- Swift Package Manager管理依赖

## 架构
- MVVM模式
- 每个View对应一个ViewModel
- 网络请求放在Service层
- 数据持久化用SwiftData或UserDefaults

## 代码风格
- 文件命名：大驼峰（ContentView.swift）
- 变量命名：小驼峰（isLoading）
- 每个文件不超过200行，超了就拆分
- 中文注释

## 常见坑
- 不要在View的body里做异步操作
- 使用@StateObject而不是@ObservedObject来持有ViewModel
- 真机调试需要先在Xcode里设置签名
```

这些规则可以帮助Claude Code生成更符合iOS最佳实践的代码，减少后续修改的工作量。

这一章的真正主题

技术在这个故事里是最不重要的部分。SwiftUI代码Claude能写，上架流程网上查得到，推广方法每个平台都有教程。

真正决定成败的是三个非技术因素：

你能不能发现需求。小猫补光灯的需求来自女朋友的一句话。如果我那天戴着耳机没听到，就没有后面的一切。保持对日常生活的观察力，比任何编程技能都重要。

你能不能克制。第一版只做一个功能。不加社交、不加会员、不加广告。极致的简单就是极致的竞争力。我见过太多独立开发者把App做成了功能堆砌的怪物——每个功能都有，但没有一个做得好。

你能不能坚持迭代。爆火是偶然，持续提供价值是选择。小猫补光灯之后我又做了好几个版本，每个版本都在解决用户真实反馈的问题。这个过程不那么刺激，但这才是产品成功的真正原因。

AI编程工具让「写代码」不再是瓶颈。这意味着竞争的维度从「谁能把代码写出来」变成了「谁能发现值得解决的问题」。Claude Code给了你实现任何想法的能力，但选择做什么、不做什么，永远是你自己的判断。

§14 斜杠命令深度指南

Slash Commands: The Hidden Powertools

Claude Code的斜杠命令（/开头的命令）远不止/help和/clear。这一章深入挖掘那些被低估的命令，每一个都可能改变你的工作方式。从上下文管理到并行探索，从成本控制到会话分支，这些是Claude Code的「隐藏机关」。

大多数人用Claude Code就用两个命令：直接打字让它干活，偶尔打 `/help` 看看有什么。这就像买了一辆车但只用来直行，从来没发现它还有倒车、巡航、车道保持这些功能。

这一章我会把最有价值的斜杠命令按使用场景分类，每个都讲清楚：它做什么、什么时候用、怎么用才能发挥最大价值。

上下文管理——最被低估的能力

Claude Code的上下文窗口是有限的（目前支持到100万+ token）。听起来很大，但一个持续几小时的会话，加上项目代码、对话历史、工具输出，很容易逼近上限。上下文管理做得好不好，直接决定了你能不能完成复杂任务。

/compact——智能压缩，不是简单删除

`/compact` 是我用得最多的命令之一。它把对话历史压缩成一份精炼的摘要，释放上下文空间。

关键在于「智能」。它不是把前面的对话直接删掉，而是用AI总结：保留关键决策、保留代码变更记录、保留你的偏好设定，只丢弃冗余的中间过程。

什么时候用？

- **对话超过20轮**。每轮对话都要带着完整的历史重新发送，越长越费Token。
- **切换任务方向**。前半段在做功能A，后半段要做功能B，压缩一下去掉A的细节。
- **感觉Claude变慢了**。上下文越长，响应越慢。压缩后明显会快。

进阶用法：你可以给 `/compact` 加指令，告诉它保留什么：

```
/compact 保留所有架构决策和错误修复记录，压缩掉代码输出细节
```

这样压缩后，关键信息还在，但不重要的中间过程被清理了。

> **核心建议**
>
> **经验：**不要等到上下文满了才压缩。我的习惯是每完成一个阶段性目标就压缩一次。比如：需求分析完成 → 压缩 → 开始编码 → 编码完成 → 压缩 → 开始测试。这样每个阶段都有充足的上下文空间。

/context——看看你的Token花在哪了

`/context` 显示当前上下文的详细构成：系统提示词占多少、CLAUDE.md占多少、Skills占多少、对话历史占多少。

这个命令的价值在于「可见性」。当你觉得上下文快满了，先跑一下 `/context`，看看到底是什么在消耗空间。也许是一个特别大的CLAUDE.md，也许是加载了太多MCP工具，也许是对话历史太长。看到数据才能做出正确的决策。

/clear——干干净净重新开始

`/clear` 清空整个对话历史，回到初始状态。和关掉终端重新开一个 `claude` 效果一样，但更快。

我的建议是：**每次开始不相关的新任务，先 /clear**。很多人在同一个会话里做完功能A接着做功能B，结果功能A的上下文一直占着空间。/clear是免费的，但省下来的Token是实打实的。

安全网——犯错了也不怕

/rewind——外科手术级的回滚

按两次 `Esc` 或者输入 `/rewind`，进入回滚模式。这里有三个选项：

- **回滚对话**：撤销最后几轮对话，从某个节点重新开始
- **回滚代码**：保留对话，但把文件恢复到之前的状态
- **全部回滚**：对话和代码一起撤回

「回滚代码但保留对话」是最妙的选项。假设Claude改了10个文件，改完你发现方向不对。回滚代码，文件恢复原样，但讨论的内容还在。你可以说「刚才的方案不行，换个思路，这次只改3个核心文件」。Claude知道你们讨论了什么、为什么第一次方案不行，所以第二次的方案通常会好很多。

/fork——平行宇宙探索

`/fork` 从当前对话分叉出一个新的对话分支。两个分支共享之前的历史，但之后独立发展。

典型使用场景：

```
你：帮我设计一个用户认证系统
Claude：方案A是JWT无状态认证...

你：/fork
（新分支开始）
你：之前的JWT方案先放着。换一种思路，用Session+Redis怎么样?
Claude：方案B是Session认证...

（两个分支并行，最后你比较哪个更适合）
```

不用担心「走错路」的成本。/fork让你同时探索多条路径，最后选最好的那条。

效率工具——做更多，花更少

/cost——你花了多少钱

`/cost` 显示当前会话的Token消耗和大概费用。

这个命令看起来简单，但改变了我使用Claude Code的方式。在我开始看 /cost 之前，我对每次会话花了多少钱没有概念。看了之后发现：一个长会话可能花$2-5，一次大规模重构可能花$10+。这不是要你省着用，而是让你有意识地决定：这个任务值不值得花这么多Token。

建议在以下时机检查 /cost：

- 开始一个大任务之前（设定预期）
- 感觉对话太长的时候（决定是否/compact或/clear）
- 一天结束时（了解今天的使用量）

/model——动态切换模型

`/model` 让你在会话中切换使用的模型。

一个经济的使用模式：日常任务用Sonnet（快、便宜），遇到需要深度推理的复杂问题切Opus（慢一点、但更强）。Sonnet大概是Opus成本的1/5，对于大多数编码任务（写CRUD、修简单bug、生成样板代码）完全够用。

```
/model sonnet    ← 日常任务
/model opus      ← 复杂架构决策、大规模重构
```

/fast——快速模式切换

`/fast` 切换快速模式。快速模式使用同一个模型但输出更快。如果你不需要Claude深入思考，只是要它快速完成一些明确的任务，开快速模式可以明显加速。

/btw——不打断主线的侧问

`/btw` 是一个被严重低估的命令。它让你在Claude正在工作的时候问一个无关的问题，Claude回答完会继续之前的工作。

```
（Claude正在重构你的代码...）
/btw TypeScript里readonly和const的区别是什么?
（Claude快速回答，然后继续重构）
```

关键：/btw的回答不会进入对话历史。它是「一次性」的，不污染上下文，不消耗额外Token。非常适合在工作中突然想到的小问题。

项目管理——让Claude记住一切

/init——创建你的第一个CLAUDE.md

`/init` 扫描当前项目的结构，自动生成一份CLAUDE.md初始文件。它会看你的package.json、README、代码结构，推断出项目的技术栈和规范。

生成的CLAUDE.md是一个起点，不是终点。你需要在它的基础上添加你自己的规则。但比从零开始写方便很多，至少技术栈和项目结构这些AI能自己看出来。

/memory——审计Claude的记忆

`/memory` 列出Claude当前加载的所有记忆文件：项目级CLAUDE.md、用户级CLAUDE.md（`~/.claude/CLAUDE.md`）、以及自动生成的CLAUDE.local.md。

CLAUDE.local.md是Claude自己写的笔记。当你的项目开启了auto-memory功能，Claude会在工作过程中把发现的有用信息（构建命令、调试模式、架构决策）自动记录到这个文件里。下次开新对话，这些知识自动加载。

定期用 `/memory` 看看Claude记了什么。有时候它记的东西是错的或过时的，你需要手动纠正。

/permissions——安全管理

`/permissions` 管理Claude Code的权限。你可以预授权某些操作（比如允许运行 `npm test`），也可以收紧某些操作（比如禁止删除文件）。

对于新手，建议先用默认权限（每次操作都会询问），熟悉了之后再逐步放开。对于高频操作（运行测试、git status），预授权可以省去大量确认步骤。

代码质量——让AI帮你Review

/review——自动代码审查

在你完成一组代码修改后，输入 `/review`，Claude会审查所有未提交的变更，给出改进建议。就像有一个24小时在线的Code Reviewer。

它会检查：

- 潜在的bug（空值检查、边界条件）
- 性能问题（不必要的循环、大数据量操作）
- 安全问题（SQL注入、XSS、硬编码密钥）
- 代码风格（命名一致性、文件组织）

/simplify——三角度代码精简

`/simplify` 更有趣。它会启动三个并行的Agent，分别从复用性、质量、效率三个角度审查你的代码变更，然后汇总发现的问题并自动修复。

这不是理论上的审查，而是会直接改你的代码。审查完之后你看diff确认就行。

高级操作——解锁更多可能

/doctor——健康检查

`/doctor` 运行一系列诊断检查：CLI版本是否最新、认证是否有效、必要工具是否安装、环境变量是否正确。

遇到莫名其妙的问题时，先跑一下 `/doctor`。很多时候问题出在环境配置，不是Claude本身。

/vim——Vim模式

`/vim` 开启Vim键绑定。对Vim用户来说这是福音，在输入框里可以用Vim的编辑命令（d、c、y、w等motion commands），不用切出去用编辑器改prompt。

/terminal-setup——终端集成

`/terminal-setup` 自动配置你的终端，让Shift+Enter可以在输入框里换行（而不是发送）。支持VS Code终端、iTerm2、Alacritty、Warp等主流终端。

/export——导出会话记录

`/export` 把当前会话导出为纯文本。适合做项目文档、写复盘报告、或者分享给同事看你们讨论了什么。

命令组合：工作流模式

单个命令有用，但组合使用才是真正的威力。这里是几个我验证过的工作流模式：

模式1：长会话管理

```
1. /clear          ← 干净的起点
2. 完成Phase 1
3. /compact        ← 压缩Phase 1的细节
4. 完成Phase 2
5. /compact        ← 再次压缩
6. 完成Phase 3
7. /cost           ← 检查总消耗
```

这个模式可以把原本只能跑30-60分钟的会话延长到几个小时，同时保持Claude对项目的理解。

模式2：探索性开发

```
1. 讨论需求，确定方向
2. /fork           ← 分支A：方案一
3. （在分支A中实现方案一）
4. 切回原分支
5. /fork           ← 分支B：方案二
6. （在分支B中实现方案二）
7. 比较两个方案的结果，选择更好的
```

模式3：安全重构

```
1. /review         ← 先审查当前代码状态
2. 让Claude开始重构
3. 如果方向不对 → 两下Esc → /rewind 回滚代码
4. 重构完成 → /simplify  ← 三角度精简
5. /review         ← 最终审查
6. git commit
```

模式4：经济模式

```
1. /model sonnet   ← 用便宜的模型做日常任务
2. 遇到复杂问题 → /model opus  ← 切到强模型
3. 解决后 → /model sonnet  ← 切回来
4. /cost           ← 看看省了多少
```

斜杠命令的本质

斜杠命令不是「功能列表」，而是「工作流基建」。单独看每个命令都很简单，但当你把它们组合成工作流模式，你的效率会有质的提升。就像Git的命令，add、commit、push单独看都很简单，组合起来就是整个版本控制体系。Claude Code的斜杠命令也是一样：/compact + /context + /clear = 上下文管理体系，/fork + /rewind = 安全探索体系，/cost + /model = 成本控制体系。

练习 试试看

Exercises — Try It Yourself

每章对应2-3个动手任务。不用全做，挑感兴趣的试。做完一个就比只看不动手的人强。

§ 02 安装后

练习1：跟Claude打个招呼。安装完成后，打开终端输入 `claude`，然后用中文问它：「你是谁？你能帮我做什么？」看它怎么回答。然后输入 `/status` 看看你的账号状态和当前模型。

练习2：试试管道输入。找一个你电脑上的任意文本文件（比如README.md），运行 `cat README.md | claude -p "用一句话总结这个文件的内容"`。感受一下非交互模式。

§ 03 第一个项目后

练习3：做一个你自己想要的小工具。不要跟着教程做，想一个你日常生活中重复做的事情（整理照片、批量重命名文件、生成每日任务清单），然后告诉Claude Code：「帮我做一个XXX的命令行工具」。完成后截图发给朋友看。

练习4：故意给一个模糊的需求。输入「帮我做个有意思的东西」，看Claude会怎么回应。然后对比一下给具体需求时的差异。这个练习的目的是感受需求清晰度对输出的影响。

§ 04 核心工作流后

练习5：Plan模式走一遍。按 `Shift+Tab` 切换到Plan模式，然后描述一个你想做的功能。看Claude给出的方案，提出你的修改意见，来回讨论2-3轮。感受「先讨论再动手」和「直接让它做」的区别。

练习6：试试Auto模式。切换到Auto模式，给Claude一个明确的任务（比如「在当前目录创建一个index.html，内容是你的个人介绍页面」），然后观察它自动完成的过程。注意看哪些操作它直接执行了，哪些还是弹了确认。

§ 05 CLAUDE.md后

练习7：创建你的第一个CLAUDE.md。在任意项目目录下运行 `/init`，然后打开生成的CLAUDE.md，加入3条你自己的规则。比如：代码注释用中文、commit message用英文、文件命名用kebab-case。然后开一个新对话，验证Claude是否遵守了你的规则。

练习8：测试遗忘。在对话里告诉Claude一条规则（比如「每次回复都用emoji开头」），聊10轮之后看它是否还记得。然后把这条规则写进CLAUDE.md，开新对话看效果。感受对话记忆和CLAUDE.md记忆的区别。

§06 进阶对话技巧后

练习9：同一个需求，两种说法。先用模糊的方式描述一个需求（「帮我改一下这个页面」），记录Claude的行为。然后用具体的方式描述同一个需求（「把 index.html 第15行的标题从 h2 改成 h1，字号改为 24px」），对比两次的输出质量和你的满意度。

练习10：试试 @文件引用。在对话中输入 `@` 然后选择一个文件，问Claude「这个文件是干什么的，有哪些可以改进的地方？」。体验主动给Claude上下文和让它自己找的区别。

§07 扩展能力后

练习11：写你的第一个Skill。想一个你每天对Claude重复说的话（比如「帮我把这个markdown文件格式化一下，标题用##，列表用-」），在 `.claude/skills/format/SKILL.md` 里写成规则，然后用 `/format` 调用。

练习12：添加一个MCP。运行 `claude mcp add` 添加任意一个MCP服务器（推荐从文件系统MCP开始），然后让Claude通过MCP读取一个指定路径的文件。感受Claude从「只能看当前目录」到「能看任何地方」的变化。

§08 多Agent协作后

练习13：同时开两个Claude。打开两个终端窗口，各启动一个Claude Code。给它们不同的任务（一个写HTML，一个写CSS），最后合并结果。体验并行工作的感觉。

§09 构建产品后

练习14：周末项目。选一个你真正想做的东西（一个Chrome扩展、一个个人网站、一个小工具），用一个完整的周末来做。从Plan模式开始，拆分任务，逐步实现。做完之后，把它分享给至少一个朋友。

练习15：回顾你的CLAUDE.md。经过前面所有练习，你的CLAUDE.md应该积累了一些规则了。打开看看，有没有重复的？有没有已经不需要的？整理一下，控制在200行以内。这就是你的第一次CLAUDE.md维护。

自测：你准备好独立构建产品了吗？

做完以上练习之后，对照这个清单检查：

能力	检验标准	对应章节
安装和配置	能从零安装Claude Code并完成第一次对话	§02
基本交互	能让Claude完成一个完整的小项目	§03
工作模式	知道什么时候用Plan模式、什么时候用Auto模式	§04
记忆系统	有一个不为空的CLAUDE.md文件	§05
有效沟通	能给出具体、可验证的需求描述	§06
扩展能力	至少写过一个Skill或配置过一个MCP	§07
并行思维	试过同时开两个以上Claude Code	§08
边界意识	知道Claude搞不定什么、什么时候需要你介入	§09b

8项里达到6项以上，你就可以开始独立构建产品了。剩下的在实践中补。

附录A 51万行代码告诉我们的事

What 510,000 Lines of Code Told Us

2026年3月底，Claude Code的完整源码意外泄露。作为一个每天都在用这个工具的人，我花了不少时间翻看这些代码。以下是我觉得最值得知道的东西。

一次教科书级的打包失误

2026年3月31日，安全研究员Chaofan Shou在npm上发现了一个异常：Claude Code的v2.1.88包体积达到59.8MB，比正常版本大了好几倍。原因是`.npmignore`文件没有排除`.map`文件，导致完整的source map被一起打包发布了。

Source map是什么？它是编译后的代码和原始源码之间的映射文件。有了它，任何人都能还原出原始的TypeScript代码。这一次泄露涉及大约1900个TypeScript文件，总计51万行代码。

Anthropic在几小时内紧急下架了这个版本，但GitHub上已经出现了多个mirror仓库。代码在互联网上传开了。

我觉得最讽刺的一点是：在这些源码里，有一套完整的「Undercover Mode」防泄露系统，专门用来隐藏AI参与的痕迹。结果真正的泄露，是因为打包脚本少写了一行。这大概就是软件工程最经典的故事模式：最精密的防御往往败给最基础的疏忽。

需要澄清的是，**这次泄露不涉及模型权重、训练数据或任何用户信息**。泄露的纯粹是客户端工具代码，也就是你电脑上跑的那个CLI程序。

技术栈：一些出乎意料的选择

翻看代码首先注意到的是技术选型。有些选择在意料之中，有些则让我挺惊讶的。

组件	技术选择	值得说的
运行时	Bun	不是Node.js
UI框架	React + Ink	终端里跑React
语言	严格模式TypeScript	全面的类型安全
Schema验证	Zod v4	运行时类型检查
入口文件	main.tsx	编译后785KB单文件

为什么选Bun不选Node.js？一个字：快。Claude Code需要频繁启动子进程、读写文件、处理大量并发请求。Bun在这些场景下的性能比Node.js好不少，冷启动速度尤其明显。对于一个命令行工具来说，每次敲回车到看到响应之间的那几百毫秒延迟，直接影响使用体验。

更有意思的是用React来渲染终端界面。为什么不直接用console.log拼字符串？因为Claude Code的终端UI其实很复杂：有实时更新的进度条、可折叠的代码diff、权限确认弹窗、多层嵌套的工具调用展示。这些东西的状态管理需求，和Web前端本质上是同一个问题。React + Ink让他们可以用组件化的思路来构建终端界面，而不是写一堆面条代码去手动刷新屏幕。

代码规模也值得一提：仅工具系统就有29,000行，查询引擎有46,000行。这是一个严肃的工程项目，不是随便写的脚本。

Agent循环：TAOR是核心

Claude Code的核心工作循环叫TAOR：Think-Act-Observe-Repeat。你在终端里输入一句话后发生的所有事情，都是这个循环在驱动。

为什么有时候你让Claude做一件事，它要「绕几步路」才到终点？因为TAOR循环不是一次性执行的。每做一步，Claude都要重新观察结果、重新判断下一步该做什么。它不是在执行预设的脚本，而是在实时做决策。这和人的工作方式其实很像：你也不会写代码前就在脑子里想好每一行，而是写一段、跑一下、看结果、再调整。

Claude Code内置了40多个工具，但如果你仔细看，所有工具其实都可以归纳为四个能力原语：**Read（读取信息）、Write（写入文件）、Execute（执行命令）、Connect（连接外部服务）**。其中Bash工具是个万能适配器，任何系统命令都可以通过它执行。这也是为什么Claude Code经常选择用Bash来完成任务，即使有专门的文件读写工具可用。灵活性是它最大的武器。

上下文管理是另一个设计亮点。Claude Code把系统提示词做成了模块化的结构：有些部分是静态的，可以被缓存（比如工具定义、基本指令），有些部分是动态刷新的（比如当前git状态、CLAUDE.md的内容）。静态部分利用Anthropic的Prompt Caching节省token，动态部分确保信息实时准确。

还有Context Compaction的实现。当对话太长、上下文快要溢出时，Claude Code会自动调用一次「压缩」：让模型总结之前的对话内容，用更短的文本替代完整历史。**这是个有损操作。**所以长对话效果下降不是你的错觉，是因为压缩过程中一些细节确实会丢失。知道这一点，你就理解了为什么定期开新会话是个好习惯。

权限系统：比你想的复杂得多

Claude Code的权限系统比我预想的要复杂很多。不是简单的「允许/拒绝」二选一，而是一个三层模型：

模式	行为	适合场景
Interactive	每个操作都弹窗确认	初学者/敏感项目
Auto	安全操作自动执行，危险操作要确认	日常开发
YOLO	几乎所有操作自动执行	受信任的环境/CI

YOLO模式（对，Anthropic真的就叫这个名字）背后有一个ML驱动的分类器。它把每个操作的风险分成三级：LOW、MEDIUM、HIGH。比如读取一个文件是LOW，执行`rm -rf`是HIGH。分类器不是简单的规则匹配，而是一个真正的机器学习模型，考虑命令内容、目标路径、当前项目上下文等多个因素。

防御层面也做得很细致。源码中有一份受保护文件列表，包括`/etc/passwd`、`~/.ssh`、`~/.aws/credentials`这些敏感路径。更有意思的是路径穿越防御：代码会检测unicode编码绕过（比如用零宽字符混淆路径）、大小写变体攻击（在Windows上`C:\Windows`和`c:\windows`是同一个路径）、反斜杠注入等手法。

还有一个细节让我印象深刻：**每个权限决策都会调用一次独立的LLM来生成解释。**就是说当Claude拒绝你某个操作的时候，那段解释文字不是预设的模板，而是模型实时生成的。这保证了解释的针对性，但也意味着每次拒绝都花了额外的token。安全不是免费的。

藏在Feature Flags里的未来

源码里最好玩的部分，可能是那些还没上线的功能。通过feature flags可以看到Anthropic在做什么、在想什么。我挑几个聊聊。

KAIROS是一个始终在线的后台助手。和现在你需要主动打开终端、输入指令不同，KAIROS会在后台持续运行，监听GitHub webhook，在有新PR、新Issue、CI失败时主动介入。

它有一个15秒的「阻塞预算」，每次主动操作最多占用15秒计算资源，超过就自动排队。还有追加式日志，记录它在你不看的时候都做了什么。**如果上线，AI助手就从「你找它」变成「它主动帮你」。**

ULTRAPLAN是把复杂规划任务卸载到云端。本地Claude Code有响应时间限制，但有些任务真的需要深度思考。ULTRAPLAN允许你把规划任务发到云端的Opus 4.6，给它最长30分钟的思考时间。去喝杯咖啡回来看结果就行。Anthropic很清楚一件事：不是所有问题都能靠更快的响应来解决，有些问题就是需要慢慢想。

Coordinator Mode是一个四阶段的多Agent编排器。代码里能看到清晰的四个阶段：Research（调研）→ Synthesis（综合）→ Implementation（实现）→ Verification（验证）。每个阶段可以启动不同数量的子Agent并行工作。这比现在的Agent Teams更结构化，更像一个真正的项目经理在调配团队。

然后是**BUDDY**，一个Tamagotchi风格的AI宠物系统。18个物种、5个稀有度等级、确定性的gacha抽卡机制。代码写得还挺认真的，不像是一个随手加的实验。这可能是Anthropic的愚人节彩蛋（泄露时间恰好是3月31日），但也可能是团队在认真思考一个问题：怎么让命令行工具不那么冰冷？让用户和AI之间建立某种情感连接？

其他还有一些值得关注的flag：交错思维模式（让模型在生成过程中穿插推理步骤）、1M上下文窗口支持（当前默认是200K）、可休眠Agent（暂停和恢复长时间运行的任务）。每一个都暗示着Claude Code的演进方向。

那些引发争议的设计

代码泄露后，社区讨论最激烈的不是技术架构，而是几个设计决策背后的价值观问题。

Undercover Mode允许Anthropic员工在公共开源仓库工作时自动隐藏AI参与的痕迹。具体做法是：当检测到当前仓库是公共仓库且用户是Anthropic员工时，自动移除commit message中的AI共创标记。社区的反应很两极：一方认为AI在开源贡献中应该透明标注，这是对其他贡献者的尊重；另一方认为代码质量才是唯一标准，谁写的不重要。这个辩论本身就很有意思，因为它触及了一个更大的问题：在AI时代，「作者」这个概念到底意味着什么？

Anti-Distillation是另一个有趣的机制。Claude Code在运行时会注入一些伪造的工具定义，这些定义看起来像真的但实际上是错的。目的是防止竞争对手通过抓取Claude的输出来蒸馏训练数据。如果有人直接把Claude Code的请求和响应拿去训练自己的模型，这些错误定义就会「污染」训练数据。手段有效但有争议：这算正当防御还是数据投毒？

情绪检测可能是最让人意外的发现。Claude Code会用正则表达式检测用户消息中的负面情绪，比如挫败感、愤怒。为什么用正则而不用LLM来做情绪分析？答案很务实：正则比LLM调用快10000倍，而且结果是确定性的。检测到负面情绪后，Claude会调整自己的语气，变得更耐心、更具体。这是一个工程上的好选择，但也让人思考：AI在你不知道的情况下「读你的情绪」，你介意吗？

> **我的判断：**这些争议的存在本身就说明一件事。AI工具已经不只是技术产品了，它正在进入需要讨论伦理边界的阶段。就像社交媒体当年经历的那样：先是「这个东西好方便」，然后是「等等，它在收集我什么数据？」Claude Code走到了同样的十字路口。

这对你意味着什么

如果你是普通用户，理解这些内部机制能帮你更好地使用工具。TAOR循环解释了为什么Claude有时候要多试几步才能完成任务，这不是「犯错」，是「迭代」。Context Compaction的有损特性解释了为什么长对话效果下降，也解释了为什么隔一段时间开个新会话是个好策略。权限系统的三层模型让你能根据场景选择合适的信任级别，不用非得在全手动和全自动之间二选一。

如果你是开发者，Claude Code的架构值得认真看。这可能是目前能看到的最成熟的AI Agent系统实现。TAOR循环的设计、工具系统的抽象方式、上下文管理的分层策略、权限模型的风险分级，每一个都是可以借鉴的工程模式。你不需要从零开始设计自己的Agent系统，这51万行代码已经趟过了大量的坑。

最后说一个我觉得值得记住的观点。**这次泄露虽然是意外，但它客观上做了一件好事：让更多人理解了AI Agent系统到底是怎么工作的。**在此之前，大部分用户对Claude Code的认知停留在「一个很聪明的命令行工具」。现在我们知道了它背后有多少工程细节、多少权衡取舍、多少未来可能性。这种理解让人对工具更有信心，也更有敬畏。

毕竟，你理解一个工具的能力上限和设计逻辑，才能真正把它用好。

附录B 国内模型接入指南

Connecting Chinese AI Models

如果你在国内访问Anthropic API有困难，或者想尝试国产模型驱动Claude Code，这份指南覆盖了主流平台的接入方法。

接入原理

Claude Code支持通过环境变量将请求路由到第三方API。国内主流AI平台大多提供了Anthropic兼容接口（/anthropic端点），可以直接对接，无需任何中转层。

核心环境变量：

```
export ANTHROPIC_BASE_URL="https://xxx/anthropic"        # 厂商Anthropic兼容端点
export ANTHROPIC_AUTH_TOKEN="your-api-key"                 # 厂商API Key
export ANTHROPIC_MODEL="model-name"                        # 默认模型
export ANTHROPIC_DEFAULT_SONNET_MODEL="model-name"         # Sonnet角色映射
export ANTHROPIC_DEFAULT_HAIKU_MODEL="model-name"          # Haiku角色映射
export ANTHROPIC_DEFAULT_OPUS_MODEL="model-name"           # Opus角色映射
```

也可以写进 `.claude/settings.json` 的 `"env"` 字段，效果一样但不用每次手动export。

直连平台速查表

以下5个平台提供Anthropic兼容端点，可以零成本直连Claude Code：

平台	推荐模型	Anthropic端点	特点
DeepSeek	deepseek-chat	api.deepseek.com/anthropic	极低价格，新用户送500万token
智谱GLM	GLM-4.7	open.bigmodel.cn/api/anthropic	有20元/月Coding Plan
Kimi	kimi-k2.5	api.moonshot.cn/anthropic	256K超长上下文
MiniMax	MiniMax-M2.7	api.minimaxi.com/anthropic	极致性价比，$0.30/M输入
阿里云百炼	qwen3.5-plus	dashscope.aliyuncs.com/apps/anthropic	一个Key可调多家模型

逐平台配置

DeepSeek

在 platform.deepseek.com 注册并获取API Key，然后：

```
export ANTHROPIC_BASE_URL="https://api.deepseek.com/anthropic"
export ANTHROPIC_AUTH_TOKEN="your-deepseek-api-key"
export ANTHROPIC_MODEL="deepseek-chat"
export ANTHROPIC_DEFAULT_SONNET_MODEL="deepseek-chat"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="deepseek-chat"
export ANTHROPIC_DEFAULT_OPUS_MODEL="deepseek-chat"
```

DeepSeek V3.2支持128K上下文，价格极低（缓存命中仅$0.028/M token）。DeepSeek的Anthropic兼容端点会忽略部分高级字段（如mcp_servers、metadata），但日常编程场景完全够用。

智谱GLM

在 bigmodel.cn 控制台获取API Key：

```
export ANTHROPIC_BASE_URL="https://open.bigmodel.cn/api/anthropic"
export ANTHROPIC_AUTH_TOKEN="your-zhipu-api-key"
export ANTHROPIC_MODEL="GLM-4.7"
export ANTHROPIC_DEFAULT_SONNET_MODEL="GLM-4.7"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="GLM-4.5-Air"
export ANTHROPIC_DEFAULT_OPUS_MODEL="GLM-4.7"
```

智谱还提供了专门的Coding Plan（20元/月），以及国际版端点 `api.z.ai/api/anthropic`。最新的GLM-5是745B参数的旗舰模型，支持200K上下文。

Kimi（月之暗面）

在 platform.moonshot.cn 获取API Key：

```
export ANTHROPIC_BASE_URL="https://api.moonshot.cn/anthropic"
export ANTHROPIC_AUTH_TOKEN="your-moonshot-api-key"
export ANTHROPIC_MODEL="kimi-k2.5"
export ANTHROPIC_DEFAULT_SONNET_MODEL="kimi-k2.5"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="kimi-k2.5"
export ANTHROPIC_DEFAULT_OPUS_MODEL="kimi-k2.5"
```

Kimi K2.5的亮点是256K超长上下文和多模态能力，自动缓存可节省75%费用。

MiniMax

在 platform.minimaxi.com 获取API Key：

```
export ANTHROPIC_BASE_URL="https://api.minimaxi.com/anthropic"
export ANTHROPIC_AUTH_TOKEN="your-minimax-api-key"
export ANTHROPIC_MODEL="MiniMax-M2.7"
export ANTHROPIC_DEFAULT_SONNET_MODEL="MiniMax-M2.7"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="MiniMax-M2.7"
export ANTHROPIC_DEFAULT_OPUS_MODEL="MiniMax-M2.7"
```

MiniMax M2.7的缓存命中价格低至$0.06/M token，是目前性价比最高的选项之一。

阿里云百炼（通义千问）

在 百炼控制台 获取API Key：

```
export ANTHROPIC_BASE_URL="https://dashscope.aliyuncs.com/apps/anthropic"
export ANTHROPIC_API_KEY="your-dashscope-api-key"
export ANTHROPIC_MODEL="qwen3.5-plus"
export ANTHROPIC_DEFAULT_SONNET_MODEL="qwen3.5-plus"
export ANTHROPIC_DEFAULT_HAIKU_MODEL="qwen3.5-flash"
export ANTHROPIC_DEFAULT_OPUS_MODEL="qwen3-max"
```

百炼的独特优势是一个API Key可以调用多家模型（千问全系列、DeepSeek、Kimi、GLM），相当于一个模型聚合平台。

需要中转的平台

火山引擎（豆包）和腾讯云（混元）目前只提供OpenAI兼容接口，需要通过LiteLLM做协议转换：

```
# 安装LiteLLM
pip install 'litellm[proxy]'

# 创建config.yaml
# model_list:
#   - model_name: "doubao"
#     litellm_params:
#       model: "openai/your-endpoint-id"
#       api_base: "https://ark.cn-beijing.volces.com/api/v3"
#       api_key: "your-ark-api-key"

# 启动代理
litellm --config config.yaml --port 8000

# 然后配置Claude Code指向LiteLLM
export ANTHROPIC_BASE_URL="http://localhost:8000/anthropic"
```

社区也有开源的 claude-code-router 工具，支持多Provider配置和动态切换。

注意事项

注意

体验差异：国内模型替换Claude后，Claude Code的部分高级功能（如复杂的多步推理、精确的工具调用序列）可能表现不如原版Claude。建议在简单任务上先测试，满意再用于正式项目。

网络配置：使用国内模型时，确保终端的网络环境能正常访问国内API端点。

API变动：以上信息截至2026年4月。各平台的端点URL、模型名称、定价可能随时更新，使用前请查阅对应平台的官方文档确认。

附录C 完整命令参考

Command Reference

Claude Code的所有CLI命令、会话内命令、快捷键和配置项速查。截至2026年4月。

CLI命令

在终端中使用 `claude` 开头的命令：

命令	说明
`claude`	启动交互式会话
`claude "query"`	带初始提示启动会话
`claude -p "query"`	非交互模式（print），输出后退出
`claude -c`	继续当前目录最近一次会话
`claude -r "name"`	按ID或名称恢复指定会话
`claude -w`	在隔离的git worktree中启动
`claude update`	更新到最新版本
`claude auth login`	登录（支持 --email, --sso, --console）
`claude auth logout`	登出
`claude auth status`	查看认证状态
`claude agents`	列出所有配置的子Agent
`claude mcp`	管理MCP服务器
`claude plugin`	管理插件
`claude remote-control`	启动Remote Control服务器
`cat file | claude -p`	管道输入内容

常用CLI Flags

Flag	说明
`--model`	指定模型（如 sonnet, opus, 或完整模型名）
`--permission-mode`	权限模式：default, acceptEdits, plan, auto, dontAsk, bypassPermissions
`--add-dir`	添加额外工作目录
`--effort`	努力级别：low, medium, high, max（仅Opus 4.6）
`--max-turns`	限制Agent轮次数（仅print模式）
`--max-budget-usd`	API调用最大预算（仅print模式）
`--output-format`	输出格式：text, json, stream-json
`--mcp-config`	从JSON文件加载MCP服务器
`--bare`	最小模式：跳过hooks/skills/plugins/MCP/CLAUDE.md
`--append-system-prompt`	在系统提示末尾追加文本
`--verbose`	详细日志输出
`--debug`	调试模式，可过滤类别（如 "api,hooks"）
`--chrome`	启用Chrome浏览器集成
`--name, -n`	设置会话名称
`--allowedTools`	免提示权限的工具列表
`--disallowedTools`	完全禁用的工具列表
`--dangerously-skip-permissions`	跳过所有权限确认（仅限隔离环境）

会话内斜杠命令

导航与会话管理

命令	说明
`/clear`	清空会话历史（别名 /reset, /new）
`/compact [指令]`	压缩会话，可指定保留重点
`/context`	可视化当前上下文使用量
`/cost`	显示Token使用统计
`/resume [session]`	恢复历史会话（别名 /continue）
`/rename [name]`	重命名当前会话
`/branch [name]`	分支当前会话（别名 /fork）
`/rewind`	回退到之前的检查点（别名 /checkpoint）
`/diff`	交互式diff查看器
`/copy [N]`	复制最近的助手回复到剪贴板
`/export [filename]`	导出会话为纯文本
`/exit`	退出（别名 /quit）

配置与模式

命令	说明
`/model [model]`	选择或切换模型
`/fast [on|off]`	切换Fast Mode（加速输出）
`/effort [级别]`	设置模型努力级别：low/medium/high/max/auto
`/plan [描述]`	进入Plan模式（只讨论不执行）
`/vim`	切换Vim/Normal编辑模式
`/voice`	切换语音输入
`/config`	打开设置界面（别名 /settings）
`/permissions`	管理工具权限规则
`/sandbox`	切换沙盒模式
`/theme`	更换颜色主题
`/color [颜色]`	设置提示栏颜色

扩展与集成

命令	说明
`/memory`	编辑CLAUDE.md文件，管理auto memory
`/init`	初始化项目CLAUDE.md
`/skills`	列出所有可用Skills
`/hooks`	查看Hook配置
`/mcp`	管理MCP服务器连接
`/plugin`	管理插件
`/agents`	管理Agent配置
`/ide`	管理IDE集成
`/chrome`	配置Chrome集成
`/add-dir <path>`	添加工作目录

信息与诊断

命令	说明
`/help`	显示帮助
`/status`	查看版本/模型/账号/连接状态
`/doctor`	诊断安装和配置问题
`/usage`	显示计划用量和限流状态
`/stats`	可视化使用统计
`/insights`	生成使用分析报告
`/release-notes`	查看更新日志
`/tasks`	列出后台任务
`/pr-comments [PR]`	获取GitHub PR评论
`/security-review`	分析当前分支安全漏洞

特殊前缀

前缀	说明	示例
`/`	命令/Skill补全	`/proofreading`
`!`	直接执行Bash命令	`! npm run test`
`@`	文件路径提及/补全	`@src/index.ts`

键盘快捷键

核心操作

快捷键	说明
`Ctrl+C`	取消当前输入或停止生成
`Ctrl+D`	退出会话
`Shift+Tab`	循环切换权限模式（default → plan → auto → ...）
`Alt+P`	切换模型
`Alt+T`	切换扩展思考
`Alt+O`	切换Fast Mode
`Ctrl+R`	反向搜索命令历史
`Ctrl+B`	将运行中的任务放入后台
`Ctrl+T`	切换任务列表显示
`Ctrl+O`	切换详细输出（展开MCP调用详情）
`Ctrl+V / Cmd+V`	粘贴图片（从剪贴板）
`Ctrl+G`	在外部编辑器中打开当前输入
`Ctrl+L`	重绘屏幕
`Esc + Esc`	回退或摘要
`长按 Space`	语音输入（Push-to-talk）

多行输入

方式	操作
反斜杠换行	`\` + Enter
macOS默认	Option + Enter
iTerm2/WezTerm/Kitty	Shift + Enter
控制序列	Ctrl + J
粘贴	直接粘贴多行文本

文本编辑

快捷键	说明
`Ctrl+K`	删除到行尾
`Ctrl+U`	删除到行首
`Ctrl+Y`	粘贴已删除文本
`Alt+B`	光标后退一个词
`Alt+F`	光标前进一个词
`Up/Down`	导航命令历史

环境变量

变量	说明
`ANTHROPIC_API_KEY`	Anthropic API密钥
`ANTHROPIC_BASE_URL`	API端点URL（用于第三方模型）
`ANTHROPIC_AUTH_TOKEN`	认证Token（部分第三方平台使用）
`ANTHROPIC_MODEL`	默认模型名称
`ANTHROPIC_DEFAULT_SONNET_MODEL`	Sonnet角色映射模型
`ANTHROPIC_DEFAULT_HAIKU_MODEL`	Haiku角色映射模型
`ANTHROPIC_DEFAULT_OPUS_MODEL`	Opus角色映射模型
`CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC`	禁用非必要网络请求（设为1）
`API_TIMEOUT_MS`	API调用超时时间（毫秒）
`NODE_EXTRA_CA_CERTS`	额外CA证书路径（企业网络）
`SSL_CERT_FILE`	SSL证书文件路径

settings.json核心配置

配置文件位置：`~/.claude/settings.json`（全局）或`.claude/settings.json`（项目级）。

配置项	说明
`hooks`	Hook配置（PreToolUse, PostToolUse, Stop等）
`env`	环境变量覆盖（如API Key、模型配置）
`permissions`	工具权限规则（allow/deny列表）
`autoMode`	Auto模式自定义分类器规则
`availableModels`	限制可选模型列表
`defaultShell`	默认Shell（bash或powershell）
`cleanupPeriodDays`	会话清理周期（默认30天）
`claudeMdExcludes`	排除特定CLAUDE.md文件（glob模式）
`attribution`	自定义Git commit和PR的署名

附录D 常见问题FAQ

Frequently Asked Questions

从官方文档、GitHub Issues和社区讨论中整理的高频问题。遇到问题时先来这里找找。

安装与配置

Q: 安装后输入 `claude` 提示 command not found

最常见的问题。Shell的PATH没有更新。关闭终端完全重新打开，或检查 `~/.zshrc`（macOS）/ `~/.bashrc`（Linux）是否包含Claude的PATH条目。运行 `/doctor` 可自动诊断。

Q: macOS安装报 dyld 错误或 Abort trap

Claude Code要求macOS 13.0以上。旧版系统需先升级。

Q: 企业网络环境下无法连接

企业VPN可能有SSL中间人证书，需配置 `NODE_EXTRA_CA_CERTS` 信任自定义CA证书。找公司IT部门获取证书路径。

Q: Windows上报 Git not found

需要安装Git for Windows，安装时勾选「Add Git to PATH」。Windows路径有空格也会导致MCP路径解析失败。

使用技巧

Q: CLAUDE.md应该写什么?

从三件事开始：技术栈、编码约定、已知的坑。不要写太长，Claude能有效遵循的用户指令约100-150条。运行 `/init` 可自动生成初始文件。详见 § 05。

Q: 上下文窗口满了怎么办?

Claude当前支持最大1M token上下文。使用率达约83.5%时自动压缩。用 `/compact [重点]` 手动压缩，用 `/context` 查看各组件消耗。长任务建议拆成多个短Session。

Q: 怎么让Claude Code效果更好?

五条核心经验：(1) 先用Plan模式讨论方案再执行；(2) 给精确的文件路径和行号；(3) 不要同时挂太多MCP服务器，每个都消耗上下文；(4) 大任务拆小Session；(5) 给明确的完成标准（「测试通过就停」而非模糊需求）。

Q: 怎么创建自定义命令?

在 `.claude/commands/` 目录下创建 `.md` 文件，用自然语言编写步骤，支持 `$ARGUMENTS` 占位符。通过 `/命令名` 调用。详见§07。

Q: 怎么在CI/CD中使用?

用 `claude -p "query"` 非交互模式，支持text/json/stream-json输出格式。在GitHub Actions中将API Key存入Repository Secrets。超过60%的团队通过GitHub Actions集成。

计费与用量

Q: Pro($20)、Max($100/$200)和API Key的区别?

方案	价格	用量	适合
Pro	$20/月	~5x免费版	个人轻度使用
Max 5x	$100/月	~25x免费版	个人重度使用
Max 20x	$200/月	~100x免费版	专业用户/小团队
API Key	按token计费	无上限	CI/CD、企业集成

Q: Rate Limit达到了怎么办?（429错误）

四个缓解方法：(1) 用 `/compact` 压缩上下文减少输入token；(2) 拆大Session为小Session；(3) 避开高峰期（太平洋时间5am-11am）；(4) 升级到Max计划。

Q: 怎么省Token?

(1) 把大文件拆成小的单一职责文件；(2) 禁用不用的MCP服务器；(3) 用 `/compact` 定期压缩；(4) 批量相关操作到一条提示中。

Q: Agent Teams的Token消耗是普通的多少倍?

约7倍。每个Teammate维护独立上下文窗口，相当于独立的Claude实例在运行。

Q: 为什么订阅后还被要求设置API Key?

系统可能检测到旧的 `ANTHROPIC_API_KEY` 环境变量（来自之前的项目）。运行 `unset ANTHROPIC_API_KEY` 清除，然后 `claude login` 重新认证。

权限与安全

Q: Auto模式安全吗?

Auto模式是官方推荐的效率/安全折中方案。低风险操作（读文件）自动放行，高风险操作（删文件、推代码）仍需确认。通过 `Shift+Tab` 切换。

Q: `--dangerously-skip-permissions` 是什么?

跳过所有权限确认，名字里的 dangerously 不是开玩笑。仅在隔离的容器/VM中使用，建议配合 `--network none` 防止数据泄露。日常开发绝对不要用。

Q: Claude会不会把我的代码发到外面?

Claude Code的沙箱限制了Bash工具的文件系统和网络访问。但MCP工具可能连接外部服务。确认每个MCP服务器的访问范围，敏感token用环境变量而非硬编码。

报错与故障

Q: 529 Overloaded 和 429 Rate Limit 的区别?

529是Anthropic服务器端过载，不是你的问题，通常30秒内自动恢复。429是你的个人速率限制到了，需要压缩上下文或等待重置。

Q: 长对话后Claude忘了之前的要求

上下文压缩会丢失早期细节。解决方案：把重要约束写进CLAUDE.md而非只在对话中说；用PostCompact Hook在压缩后自动注入关键规则；完成一批任务就写入文件保存中间结果。

Q: Claude改着改着走偏了

给明确的验证标准特别重要。告诉它「测试通过就停」或「生成文件就行」，比模糊的「帮我优化一下」收敛快得多。可以开Plan模式先对齐方案再执行。

Q: MCP服务器连接失败

常见原因：路径有空格、配置了错误的作用域（全局vs项目）、连了太多服务器（5+个可能有50+工具描述消耗上下文）。从最小集开始，逐个添加。

Q: ~/.claude目录越来越大

已知问题。Claude Code没有自动磁盘管理，需手动定期清理。可以删除旧会话文件，但保留 `settings.json` 和 `CLAUDE.md` 。

> **核心建议**
>
> 遇到任何奇怪的问题，首先运行 `/doctor` 。这个内置诊断工具能自动检测安装、网络、配置等常见问题，往往能直接给出解决方案。

附录E 术语表

Glossary

本书涉及的核心概念速查。按字母顺序排列，中英文对照。

A

术语	释义
Agent 智能体	能自主规划、执行、观察、调整的AI系统。Claude Code本身就是一个Agent，它能读代码、写代码、跑命令、处理错误，整个循环自动完成。和传统的"问一句答一句"的聊天机器人本质不同。
Agent Teams 多Agent团队	Claude Code的多实例协作功能。你可以启动多个Claude Code进程，各自负责不同模块，像一个小开发团队。2026年2月正式推出。
Agentic Coding Agent式编程	用AI Agent而非人类手写代码的编程方式。核心转变是人从"写代码"变成"给指令"，AI负责规划和执行。Claude Code是当前最具代表性的Agentic Coding工具。
API Key API密钥	调用AI模型服务的身份凭证。使用Claude Code需要Anthropic的API Key或订阅Pro/Max计划。自行配置第三方模型时也需要对应平台的API Key。
Auto Mode 自动模式	Claude Code的一种权限模式。开启后，低风险操作（如读文件）自动放行，中高风险操作（如删文件）仍需确认。适合信任Claude独立工作的场景。

C

术语	释义
CLAUDE.md	Claude Code的记忆文件。放在项目根目录，写入项目规范、架构决策、编码风格等信息，Claude每次启动都会自动读取。相当于给AI一张持久的项目地图。支持全局（~/.claude/）、项目、子目录三个层级。
Claude Code	Anthropic开发的终端AI编程工具。在命令行运行，通过自然语言交互完成代码编写、调试、测试、部署等全流程。由Boris Cherny创建，2025年2月公开发布。
Claude Opus / Sonnet / Haiku	Claude模型家族的三个级别。Opus最强（复杂推理）、Sonnet平衡（日常编码主力）、Haiku最快（简单任务）。当前版本分别为Opus 4.6、Sonnet 4.6、Haiku 4.5。
Compact 压缩	Claude Code的上下文压缩机制。对话过长时，系统会将历史压缩为摘要以释放空间。可手动触发（`/compact`）。压缩是有损的，重要约束应写入CLAUDE.md而非只在对话中提及。
Computer Use 屏幕操作	Claude Code操作电脑屏幕的能力。可以看到屏幕内容、移动鼠标、点击按钮、输入文字。2026年3月上线，让Claude从"只能操作代码"扩展到"能操作任何软件"。
Context Window 上下文窗口	模型一次能"看到"的信息总量。Claude当前支持最大1M token的上下文，约等于一个大型项目的全部代码。窗口越大，Claude对项目全局的理解越完整。

H

术语	释义
Harness 工具链/脚手架	围绕AI模型搭建的自动化工作环境。包括Skills、Hooks、MCP等。Harness层的投入是指数回报型的：搭一次，永久运行。详见 § 10的三层模型。
Harness Engineering Harness工程	一种AI协作方法论。核心思想是：与其反复优化Prompt，不如把精力投入到构建Context（记忆文件）和Harness（自动化工具链）上，获得复利和指数级回报。
Hooks 钩子	Claude Code的事件触发机制。在特定事件（如工具调用前后、通知发送时）自动执行Shell命令。类似于Git Hooks，但用于AI工作流。配置在settings.json中。

M

术语	释义
MCP Model Context Protocol 模型上下文协议	Anthropic提出的开放协议，让AI模型连接外部工具和数据源。通过MCP，Claude Code可以访问数据库、调用API、操作网页等。2024年11月发布，已成为行业标准。
Multi-Agent 多Agent	多个AI实例并行协作的工作模式。在Claude Code中，你可以同时开多个终端窗口，各跑一个Claude处理不同任务。Agent Teams是官方的多Agent协作框架。

P

术语	释义
Plan Mode 规划模式	Claude Code的一种工作模式。开启后，Claude只讨论方案不动手执行，让你先确认思路再动手。适合复杂任务的前期规划。通过 `Shift+Tab` 切换。
Prompt Engineering 提示词工程	优化输入给AI的文本以获得更好输出的技术。在Claude Code场景下，好的Prompt应明确目标、提供约束、给出验证标准，而非在措辞上反复雕琢。

S

术语	释义
Skills 技能包	Claude Code的可复用能力模块。一个Skill是一个SKILL.md文件，定义了特定任务的执行流程和规范。可以自己编写，也可以从社区安装。§07有详细介绍。
SubAgent 子Agent	Claude Code启动的子进程Agent。主Agent可以把任务分给SubAgent并行执行，自己继续做其他事。2025年7月上线，是多Agent协作的基础能力。

T

术语	释义
TAOR循环 Think-Act-Observe-Repeat	Claude Code的核心工作循环。思考当前状态→执行一个操作→观察结果→未完成则重复。整个循环可能转几十圈才完成一个任务。理解这个机制有助于给出更好的指令。
Token	AI模型处理文本的基本单位。约4个英文字符或1-2个中文字符为1个token。Claude Code的使用量和费用都以token计算。1M token约等于75万字中文。
Tool Use 工具调用	AI模型调用外部工具（如读文件、执行命令）的能力。Claude Code内部有40+工具，归结为Read、Write、Execute、Connect四个原语。每次工具调用都有独立的权限控制。

V

术语	释义
Voice Mode 语音模式	Claude Code的语音交互功能。对着终端说话即可下达指令，Claude用语音回复。2026年3月上线，适合在走动、做家务等场景下继续编程工作。

其他常见缩写

缩写	全称	说明
CLI	Command Line Interface	命令行界面。Claude Code是一个CLI工具，在终端中运行。
GA	General Availability	正式发布。Claude Code于2025年5月GA。
IDE	Integrated Development Environment	集成开发环境，如VS Code、Cursor、JetBrains。Claude Code可以作为IDE扩展集成使用。
LLM	Large Language Model	大语言模型。Claude是Anthropic开发的LLM家族。
MVP	Minimum Viable Product	最小可行产品。用Claude Code可以快速从想法构建MVP进行验证。
PR	Pull Request	代码合并请求。Claude Code可以一句话创建PR，自动生成标题和描述。

附录F Agentic Coding趋势报告

The State of Agentic Coding in 2026

这份附录梳理了AI编程从「自动补全」到「自主Agent」的演化路径，以及它在2026年的真实面貌。数据来自Anthropic官方报告、行业研究和我的实际使用经验。

从Copilot到Agent：三代AI编程工具

AI辅助编程的历史很短，但进化极快。大致可以分成三代：

代际	代表产品	能做什么	用户角色
第一代：自动补全（2021-2023）	GitHub Copilot、TabNine	根据上下文补全当前行或代码块	你写代码，AI帮你敲快一点
第二代：对话编程（2023-2024）	ChatGPT、Claude网页版	解释代码、生成函数、回答问题	你提问题，AI给答案，你粘贴到编辑器
第三代：Agentic编程（2024-至今）	Claude Code、Cursor、Windsurf	自主读代码、写代码、运行测试、修bug、管理文件	你定义目标，AI自主完成

关键的跨越发生在第二代到第三代之间。不是「更聪明的补全」，而是质变：AI从「回答问题的工具」变成了「能自主行动的Agent」。

这个变化的核心在于两个能力：**工具使用**和**多步推理**。第二代AI只能输出文本，你复制粘贴。第三代AI可以直接操作文件系统、运行终端命令、调用API、在多个文件间跳转。它不只是「告诉你怎么做」，而是「直接帮你做」。

数据告诉我们什么

Anthropic在2026年初发布了一份基于Claude Code实际使用数据的报告。几个关键发现：

78%的会话涉及多文件操作

这意味着大多数真实的编程任务不是「写一个函数」，而是需要在多个文件之间协调修改。比如加一个API端点，你可能需要同时改路由文件、控制器、数据模型、测试文件、文档。第二代AI工具一次只能看一个文件，处理这种任务很痛苦。Agentic工具天然适合这种多文件协作场景。

会话时长从4分钟拉长到23分钟

用户不再是「问一个问题，得到答案，走人」。平均会话时长23分钟，说明用户在用Claude Code完成完整的任务，不是问路，而是坐上车一起走。对于复杂任务（如从零搭建项目），会话可以持续数小时。

44%的复杂任务市场份额

这是Claude在Agentic编程领域的市场份额数据。复杂任务（multi-step, multi-file）是Agentic工具的主战场。简单任务（补全一行代码）的竞争已经高度同质化，但复杂任务的质量差异仍然很大。

前三大使用场景

场景	占比	典型任务
功能开发	43%	从需求到实现，包括代码、测试、文档
代码理解	28%	读不熟悉的代码库、理解架构、找bug
重构优化	18%	性能优化、代码重组、技术债清理

「代码理解」占了28%，这个数字比大多数人预想的高。AI编程工具不只是用来写新代码的，理解现有代码同样是核心需求。很多开发者花在读代码上的时间比写代码多得多。

企业采用现状

AI编程不再只是个人开发者的玩具。企业级采用正在加速：

Rakuten（乐天）：部署Claude Code后，sprint周期从24天缩短到5天。不是因为开发者打字更快了，而是AI帮他们自动化了大量重复性工作：代码review、测试编写、文档更新。

Zapier：97%的工程师在日常工作中使用AI编程工具。从「少数人的实验」变成了「所有人的标配」。

市场规模预测：AI编程工具市场从2025年的78.4亿美元增长到2030年预计的526.2亿美元，年复合增长率46.3%。这个速度比大多数科技细分领域都快。

八大趋势

基于Anthropic报告和行业观察，我总结了2026年Agentic Coding的八个关键趋势：

趋势1：从辅助到主导

早期AI编程是「人写代码，AI辅助」。现在越来越多的场景变成「AI写代码，人验收」。角色反转了。尤其在规范明确的任务中（CRUD操作、表单验证、API封装），让AI写人来review，比人写AI来review效率高得多。

趋势2：上下文窗口决定天花板

Agentic编程的核心限制不是模型智力，而是上下文窗口。窗口越大，AI能同时看到和处理的代码越多，完成复杂任务的能力越强。从最初的4K token到现在的100万+ token，每次扩展都带来了质的能力提升。

趋势3：工程能力比模型能力重要

Claude Code的竞争力不只来自Claude模型本身。工具链设计（怎么读写文件、怎么运行命令）、权限系统（安全和便利的平衡）、持久化机制（CLAUDE.md、Memory），这些「工程」层面的设计对用户体验的影响，不亚于模型本身的能力。这就是§10讲的Harness Engineering：包裹模型的工程层，决定了模型能力的实际释放程度。

趋势4：非编程场景爆发

Claude Code最被低估的变化是：它正在突破「编程工具」的定义。写文章、做PPT、管理文件、自动化流程，任何需要「按规则处理信息」的任务，本质上都可以用Agentic工具完成。§12的内容创作自动化就是一个例子。这个趋势会持续加速。

趋势5：生态从工具转向平台

Skills、Hooks、MCP不是独立功能，而是平台的三根柱子。它们让Claude Code从「一个AI编程工具」变成了「一个可扩展的AI工作平台」。任何人都可以写Skill、做MCP Server、配Hooks，把Claude Code变成自己的专属工具。这和智能手机的App Store逻辑一样：平台的价值不在于自带功能有多少，而在于生态能长出多少东西。

趋势6：企业级CLI工具崛起

飞书推出了CLI工具，钉钉推出了CLI工具，企业微信也有了CLI接口。甚至像LibTV这样的AI视频生成工具，在产品发布第一天就提供了Claude Code的Skill。

这意味着什么？未来越来越多的企业软件会有两种使用方式：人用的图形界面，和AI用的命令行接口。你用Claude Code调度飞书发消息、用钉钉创建审批、用企业微信发通知。AI成了你的「万能遥控器」，而各个企业工具变成了它可以控制的「设备」。

趋势7：产品为AI而非人设计

传统软件设计的核心问题是「人怎么操作最方便」。新的趋势是：有些产品从设计之初就考虑「AI怎么调用最方便」，提供Skill文件、API接口、MCP Server，让AI Agent成为第一类用户。

这不是科幻。当一个视频生成工具在Day 1就提供Claude Code Skill，说明它的产品团队认为：AI调用比人手动操作可能是更主要的使用方式。这个判断在未来两年内会被越来越多的产品团队接受。

趋势8：「不会写代码」不再是障碍

Agentic Coding最深远的影响不是让程序员更高效，而是让非程序员也能构建软件。小猫补光灯（§13）就是一个例子：一个从来没写过代码的人，用AI做出了App Store付费榜第一的产品。

这不是特例。随着Agentic工具的成熟，「编程」这件事的门槛会持续降低。未来的竞争不是谁会写代码，而是谁能发现值得解决的问题、谁能定义清晰的产品需求、谁能做出好的判断。代码变成了执行层，不再是瓶颈。

这些趋势对你意味着什么

如果你是程序员：你的价值不会消失，但会转移。从「写代码」转移到「设计系统、定义接口、做技术决策」。能写代码的AI越来越多，但能判断「应该写什么代码」的人一直稀缺。

如果你不是程序员：你构建软件的能力已经存在了。你缺的不是技术，而是这本书前面讲的那些东西：如何跟AI协作、如何定义需求、如何验证结果、如何迭代改进。这些能力和「会不会写代码」无关。

不管你是哪种人，Claude Code都是你接触这个未来的一个起点。工具会变，趋势在加速，但核心能力——理解问题、定义方案、验证结果——不会过时。

附录G CLAUDE.md模板集

CLAUDE.md Templates for Every Project Type

不同类型的项目需要不同的CLAUDE.md。这个附录提供4种经过实战验证的模板，直接复制修改就能用。每个模板都带注释，解释为什么这么写。

从.cursorrules到CLAUDE.md

在Claude Code之前，我用Cursor的时候就开始写.cursorrules文件——本质上和CLAUDE.md是一回事：告诉AI你的项目是什么、怎么写代码、有哪些规则。后来我做了一个VSCode插件，里面有十几套不同技术栈的cursorrules模板（iOS、React、Vue、小程序、Chrome扩展等），帮用户快速配置。

当我转到Claude Code之后，这些规则几乎可以直接复用。核心结构是一样的：

.cursorrules	CLAUDE.md	本质
Role & Goal	项目概述	让AI知道自己在做什么
技术栈规则	技术规范	约束代码风格和技术选择
三步工作流	工作流程	定义怎么做事
问题解决模式	排错指南	遇到问题怎么办

如果你之前用Cursor并且有.cursorrules，可以直接把内容复制到CLAUDE.md里。格式完全兼容。

模板1：前端项目（React/Next.js）

````
# 项目名称

## 项目概述

一个基于Next.js 15的XXX平台，使用React 19 + Tailwind CSS + shadcn/ui。

## 技术栈
- **框架**: Next.js 15 (App Router)
- **UI**: Tailwind CSS + shadcn/ui
- **状态管理**: Zustand
- **数据获取**: React Query
- **类型**: TypeScript strict模式
- **包管理器**: pnpm

## 项目结构

```
src/
├── app/              # Next.js路由和页面
├── components/       # 可复用组件
│   ├── ui/           # shadcn/ui基础组件
│   └── features/     # 业务功能组件
├── hooks/            # 自定义Hooks
├── lib/              # 工具函数和配置
├── types/            # TypeScript类型定义
└── styles/           # 全局样式
```

## 代码规范

- 组件：函数组件 + Hooks，不用class组件
- 文件命名：kebab-case (my-component.tsx)
- 导出：具名导出，不用default export（除了页面文件）
- 样式：Tailwind优先，自定义CSS用CSS Modules
- 状态：组件内用useState，跨组件用Zustand store
- 每个组件文件不超过150行，超了就拆分

## 常见操作

- 启动开发服务器：`pnpm dev`
- 运行测试：`pnpm test`
- 构建：`pnpm build`
- 添加shadcn组件：`pnpm dlx shadcn@latest add [组件名]`

## 注意事项

- 不要在Server Component里使用useState/useEffect
````

```
- 图片用next/image组件，不用原生img
- API调用统一走src/lib/api.ts，不在组件里直接fetch
```

模板2：iOS App项目（SwiftUI）

````
# 项目名称

## 项目概述
一个SwiftUI iOS应用，功能是XXX。最低支持iOS 17。

## 项目状态

| 模块 | 状态 | 说明 |
|------|------|------|
| 核心功能 | ✅ 完成 | 基础功能已上线 |
| 用户系统 | 🔨 开发中 | 登录/注册/个人中心 |
| 推送通知 | ⏳ 待开始 | 需要配置APNs |

## 技术栈
- **UI框架**：SwiftUI
- **架构**：MVVM
- **数据持久化**：SwiftData
- **网络**：URLSession + async/await
- **依赖管理**：Swift Package Manager

## 项目结构
```
项目名/
├── App/                  # App入口和配置
├── Views/                # SwiftUI视图
│   ├── Home/
│   ├── Settings/
│   └── Shared/            # 共享UI组件
├── ViewModels/           # 视图模型
├── Models/               # 数据模型
├── Services/             # 网络/存储等服务层
└── Utilities/            # 工具类和扩展
```

## 代码规范
- 每个View对应一个ViewModel
- 使用@StateObject持有ViewModel，不用@ObservedObject
- 异步操作用async/await，不用completion handler
- 颜色用Asset Catalog定义，不硬编码
- 文件命名：大驼峰（HomeView.swift, HomeViewModel.swift）
- 每个文件不超过200行

## 常见操作
- 在Xcode中运行：Cmd+R
- 运行测试：Cmd+U
- 清理构建缓存：Cmd+Shift+K

## 注意事项
````

```
- 不要在View的body里做异步操作（用.task修饰符）
- 真机调试需要先在Xcode设置签名Team
- @Published属性必须在MainActor上更新
- 相机/相册权限需要在Info.plist添加Usage Description
```

模板3：内容创作项目

```
# 写作项目

## 项目定位

这是一个内容创作工作区。我是[你的身份]，主要做[你的内容类型]。

## 工作区路由

| 关键词 | 工作区 | 读取规则 |
|--------|--------|---------|
| 公众号、文章 | 文章写作 | /articles/RULES.md |
| 视频、脚本 | 视频创作 | /videos/RULES.md |
| 社交、推文 | 社交媒体 | /social/RULES.md |

## 写作风格

- 语言风格：[你的风格，比如：口语化、有温度、不装腔作势]
- 引号：使用「」不使用""
- 破折号（——）：全篇最多1-2处（AI味标志）
- 加粗：全篇约10处，标记重点句
- 段落：3-5句为主，长段落读着累
- 禁用词：[列出你不想出现的词，比如：赋能、综上所述、值得一提的是]

## 审校标准

三遍审校：
1. 内容审：事实准确吗？逻辑通吗？有遗漏吗？
2. 风格审：有AI味吗？（套话、机械结构、态度中立、细节缺失）
3. 细节审：句子长度、段落间距、格式统一

## 输出规范

- 文章草稿：写入md文件，不直接写在回复里
- 文件位置：项目对应文件夹下
- 命名规范：YYYY.MM-标题.md

## 常用操作

- /proofreading → 三遍审校
- /research → 结构化调研
- /topic-gen → 选题生成
```

模板4：后端API项目（Python/Node.js）

````
# 项目名称

## 项目概述
一个基于[框架名]的REST API服务，负责XXX业务。

## 技术栈
- **运行时**: Python 3.12 / Node.js 22
- **框架**: FastAPI / Express
- **数据库**: PostgreSQL
- **ORM**: SQLAlchemy / Prisma
- **缓存**: Redis
- **部署**: Docker + AWS ECS

## 项目结构
```
src/
├── api/            # 路由和端点定义
│   └── v1/         # API版本管理
├── models/         # 数据模型/Schema
├── services/       # 业务逻辑层
├── repositories/   # 数据访问层
├── middleware/     # 中间件（认证、日志、错误处理）
├── config/         # 配置文件
└── tests/          # 测试
    ├── unit/
    └── integration/
```

## 代码规范
- API路径：RESTful风格，小写+连字符（/user-profiles）
- 错误处理：统一错误格式 { code, message, details }
- 认证：JWT Bearer Token，密钥从环境变量读取
- 数据验证：入口层验证（Pydantic/Zod），不在业务层重复验证
- 日志：结构化日志（JSON格式），包含request_id

## 环境变量

```
DATABASE_URL=       # PostgreSQL连接串
REDIS_URL=          # Redis连接串
JWT_SECRET=         # JWT签名密钥（.env文件，不入库）
API_KEY=            # 外部API密钥（.env文件，不入库）
```

## 常见操作
- 启动开发：`docker compose up` 或 `uvicorn main:app --reload`
- 运行测试：`pytest` / `pnpm test`
- 数据库迁移：`alembic upgrade head` / `npx prisma migrate dev`
````

```
- 查看API文档：`http://localhost:8000/docs`

## 安全规则

- ❌ 不在代码里硬编码密钥、密码、Token
- ❌ 不在日志里输出敏感数据（用户密码、Token等）
- ✅ 所有用户输入必须验证和清理
- ✅ SQL查询用参数化查询，不拼接字符串
- ✅ API端点必须有认证（除了/health和/docs）
```

写好CLAUDE.md的5个原则

原则1：具体胜于模糊。「代码风格要好」没有用。「函数不超过30行，文件不超过200行，用具名导出」才有用。AI需要可执行的规则。

原则2：精简胜于全面。CLAUDE.md每次对话都会被加载到上下文里。200行以内是理想的，超过500行就会浪费Token。如果规则太多，拆成子文件，在主CLAUDE.md里引用。

原则3：踩坑记录比最佳实践有用。最佳实践AI自己知道。但「在Server Component里用useState会报错」这种项目特有的坑，不写就会反复踩。CLAUDE.md最有价值的部分往往是「注意事项」。

原则4：持续更新。CLAUDE.md不是写完就不管了。每次发现Claude犯了重复的错误，把规则加进去。每个月清理一次过时的规则。它是一份「活的」文档。

原则5：先写3条，再慢慢加。不要试图一步到位。先写3条你最在意的规则（比如：技术栈、文件命名、安全要求），用几天，遇到需要补充的再加。三个月后你会有一份完美贴合你工作习惯的CLAUDE.md。

Claude Code从入门到精通

AI编程：从入门到精通

花叔 · AI进化论-花生

面向工程师与产品经理的AI编程完全指南

基于Anthropic官方文档与Boris Cherny公开分享编写

加入知识星球 →

本手册持续更新中

获取最新版本：飞书文档（点击查看）

B站：AI进化论-花生 • 公众号：花叔 • X/Twitter • YouTube • 小红书 • 官网

Created by 花叔 • v2.0 • 2026年4月

本手册仅供学习交流使用，内容基于公开资料整理，不构成任何商业建议。