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

页数： ~75页 · 10章 + 附录

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

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

§10 心智模型与持续进化

附录

附录A 51万行代码告诉我们的事

§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。

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。

全书路线图

全书按一条主线走：**一个聪明人怎么在一周内从零用AI构建产品。**

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

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

下一章，我们花10分钟把Claude Code装好。

§02 10分钟完成安装

Get Started in 10 Minutes

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

三种安装方式

macOS、Linux、Windows都支持。选哪个取决于你的系统：

安装方式	命令	适用平台	推荐度
Native Install	`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。

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。国内网络环境下可能连不上，需要配代理：

```
# 设置HTTP代理（替换为你的代理地址）
export HTTPS_PROXY=http://127.0.0.1:7890
export HTTP_PROXY=http://127.0.0.1:7890
```

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

安装时报Permission denied

macOS / Linux用户遇到权限错误，不要用 `sudo`。这么搞：

```
# 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。

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。

§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。

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。

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。

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。
````

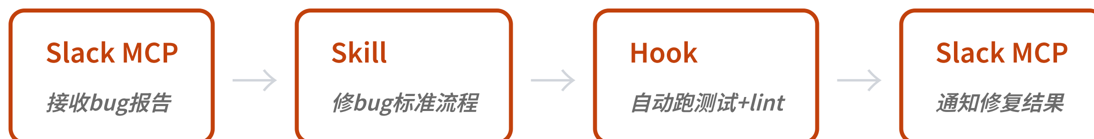
输入 `/commit-push-pr`，Claude就会按照这个流程自动执行。因为command文件存在 `.claude/commands/` 里，它会随Git一起提交，团队成员都能用。

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。

> **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」。

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。

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。

§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。**

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
- 025. 公开发布Beta版，从内部工具变成公共产品
- 025. GA正式发布，稳定性大幅提升
- 025. SubAgents上线，Claude可以启动子进程并行工作
- 025. Hooks机制引入，事件驱动的自动化成为可能
- 025. Skills系统发布，社区可以共享和复用能力包
- 026. Agent Teams正式推出，多Agent协作进入实用阶段
- 026. Computer Use上线，Claude获得操作屏幕的能力；Voice Mode让你对着终端说话
- 026. 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。

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。

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。