2026年4月 · 橙皮书系列

Claude Code 源码解析

从51万行泄露代码中，拆解Anthropic的AI工程决策

Inside Claude Code — Engineering Decisions Behind the World's Best AI Coding Tool

---

基于版本：Claude Code v2.1.88 源码（2026年3月泄露）

定位：给AI产品人看的架构课

前置阅读：建议先阅读「Claude Code从入门到精通」橙皮书

页数：~80页 · 12章

花叔

公众号「花叔」· B站「AI进化论-花生」

> 知识星球「AI编程：从入门到精通」专属内容

本手册基于2026年3月Claude Code v2.1.88源码泄露事件中公开的TypeScript源码分析编写。分析对象为客户端工具代码，不涉及模型权重、训练数据或用户信息。源码已被多个GitHub仓库镜像存档。AI工具迭代极快，源码分析反映的是泄露版本时的状态。

本手册持续更新，获取最新版本请访问：飞书文档

目录

Table of Contents

01 一个价值数十亿美元的设计蓝图

A Billion-Dollar Blueprint

2026年3月31日，Anthropic不小心把Claude Code的全部源码打包进了npm发布包。1902个TypeScript文件，51万行代码，所有设计决策一览无余。这本书就是对这份设计蓝图的深度解读。

这本书讲什么 What This Book Is About

如果你读过我的「Claude Code从入门到精通」橙皮书，那本书教你怎么开这辆车。这本书带你看发动机。

但我不会逐行读代码。那是程序员的事，而且源码随时会更新，逐行分析的保质期很短。

我要做的是拆解**设计决策**。Anthropic在构建Claude Code的过程中，做了大量非显而易见的工程选择。为什么搜索用grep不用向量数据库？为什么记忆系统只记偏好不记代码？为什么Auto模式背后跑着第二个AI做安全审查？为什么选Bun不选Node.js？

每一个选择背后都有理由。这些理由比代码本身更有价值，因为它们是可迁移的。不管你用什么语言、什么框架，这些设计思路都能直接借鉴。

Harness Engineering：一个正在成形的学科 The Emerging Discipline

2026年有一个概念越来越热：**harness engineering**。

什么意思？AI Agent好不好用，不只取决于底层模型多强，更取决于你围绕模型搭建的那套系统有多好。这套系统叫harness（笼具），包括工具调用、权限控制、记忆管理、上下文压缩、安全护栏、多Agent协调……所有让AI从「能力强但不可预测」变成「稳定可靠能交付」的工程组件。

Claude Code的源码，就是目前为止最完整的一份harness engineering教材。

市面上那么多AI编程工具，底层都在调Claude或GPT的API，使用体验却天差地别。差异不在模型，在harness。同一个Opus 4.6模型，套上Cursor的harness是一种产品，套上Claude Code的harness是另一种产品，套上一个业余开发者写的wrapper又是另一种。**模型决定了能力上限，harness决定了这个上限能兑现多少。**

我的判断是：Claude Code好用，60%靠模型能力，40%靠harness工程。这本书讲的就是那40%。

为什么这本书值得存在 Why This Book Matters

你可能会问：源码都公开了，为什么不直接去读？

因为51万行代码不是设计来被阅读的。没有架构文档，没有设计说明文档，没有README解释为什么这样做而不那样做。代码告诉你what和how，但不告诉你**why**。

而why才是最有价值的部分。为什么搜索不用向量数据库？因为LLM够聪明。为什么记忆系统不记代码？因为代码变化快，记了反而有害。为什么权限分类器用两阶段而不是一阶段？因为第一阶段用64 token就能放行大部分操作。

这些why不在代码里。它们散落在注释片段中、隐含在设计选择中、藏在PR号引用和模型版本标注里。这本书做的事情就是把这些散落的why收集起来，连成一套连贯的设计思维。

另一个原因是时效性。AI工具迭代极快，Claude Code的代码每天都在变。但设计决策的演化速度比代码慢得多。Anthropic不太可能明天就把grep换成向量数据库，不太可能明天就取消权限系统的四层审查。这些架构决策会持续很长时间，理解它们的价值远大于理解某一行代码。

泄露的故事 The Leak

简单回顾一下。详细版本可以看v2.0橙皮书的附录A。

2026年3月31日，安全研究员Chaofan Shou在npm上发现Claude Code的v2.1.88包体积异常，59.8MB，比正常版本大了好几倍。原因是构建流水线没有排除 `.map` 文件（source map），导致完整的TypeScript源码被一起打包发布了。

泄露规模：约1900个TypeScript文件，51万行代码，59个内置工具定义，44个feature flags。

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

最讽刺的是：源码里有一套完整的防泄露系统（Undercover Mode），结果真正的泄露是因为打包脚本少写了一行。而且这已经是第二次了，2025年2月Claude Code刚发布时就出过一模一样的事故。

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

泄露余波

故事没有到此结束。泄露后48小时内发生了几件值得记录的事：

Anthropic向GitHub发送了DMCA通知。由于fork网络的链式效应，GitHub执行过程中**误禁了8100多个仓库**，其中许多只是Anthropic自己公开仓库的fork（包含skills和示例代码），跟泄露完全无关。这个失误比泄露本身闹得更大。Anthropic后来把取消范围缩减到1个仓库和96个fork。

Claude Code的创建者Boris Cherny公开回应了这件事。他的原话值得收录：「This was human error. There was a manual deploy step that should have been better automated. No one was fired — this was an honest mistake. As a team, it's important to recognize that this is never an individual's fault — it's a process, culture, or infrastructure issue.」

「不怪个人怪流程」。这句话比任何技术分析都更能说明Anthropic的工程文化。

更讽刺的是泄露的可能根因。社区发现Bun有一个已知bug（oven-sh/bun#28001，3月11日提交），报告source map在生产模式下仍然被服务，尽管文档说应该被禁用。该issue至今仍然开放。如果这确实是原因，那Anthropic的工具链（Bun）发布了一个已知bug暴露了自己的产品。

还有一个安全事件：泄露同期，npm上出现了恶意版本的axios（1.14.1和0.30.4），包含远程访问木马。在特定时间窗口内安装Claude Code的用户可能拉入了恶意依赖。这提醒我们：**供应链安全不是理论威胁**。

这本书的读法 How to Read This Book

全书分四个部分：

Part I · 全景（第1-2章）：建立整体认知。了解Claude Code的技术栈、核心架构和运行循环。

Part II · 核心系统（第3-8章）：逐一拆解六个核心系统的设计决策。每章的结构是：这个系统做什么 → 源码里怎么实现的 → 为什么这么设计 → 你能学到什么。

Part III · 高级架构（第9-11章）：多Agent协作、未发布功能、内外部版本差异。这些内容更偏前沿，适合对AI产品架构有深入兴趣的读者。

Part IV · 方法论（第12章）：把前面所有章节的洞察提炼成可复用的harness engineering原则。

你可以从头到尾读，也可以挑感兴趣的章节跳着看。每章都是相对独立的。

> **关于源码引用**：本书中引用的代码片段和文件路径，基于2026年3月泄露的v2.1.88版本。Claude Code在持续迭代，实际代码可能已经变化。但设计决策和工程理念的变化通常比代码慢得多，这也是为什么我更关注「为什么」而不是「怎么写」。

一些数字 By the Numbers

在开始之前，先用几个数字建立直觉：

指标	数量	说明
TypeScript源文件	~1,900	全部严格模式
代码总行数	~510,000	不含依赖
内置工具	59	从Read到TeamCreate
Feature Flags	44	控制未发布功能
权限分类器	2阶段	64→4096 token预算
记忆类型	4种	user/feedback/project/reference
压缩结构	9段	结构化信息保留
向量数据库	0个	搜索全靠grep

最后一行可能是最能引起你好奇心的。51万行代码，零向量数据库。这个决策背后的逻辑，第8章会详细展开。

02 架构全景：Harness的骨架

Architecture Overview — The Skeleton of the Harness

在深入每个子系统之前，先建立对整体架构的认知。Claude Code不是一个简单的API wrapper，它是一个精心设计的运行时系统，包含多个协作的子系统。

技术栈：每个选择都有理由 The Tech Stack

翻开源码首先看到的是技术选型。有些选择在意料之中，有些让人挺惊讶。

组件	选择	为什么
运行时	**Bun**	比Node.js快，尤其是冷启动和子进程创建
UI框架	**React + Ink**	终端UI的状态管理需求和Web前端本质相同
语言	**严格模式TypeScript**	1900个文件的项目，没有类型系统会崩溃
Schema验证	**Zod v4**	运行时类型检查，防止AI返回格式错误的数据
入口文件	**main.tsx**（785KB）	单文件打包，减少模块解析开销
重模块加载	**懒加载**	OpenTelemetry、gRPC等按需加载，不拖慢启动

为什么选Bun不选Node.js

Claude Code需要频繁启动子进程（每次调用Bash工具就是一个子进程）、大量文件读写（Grep、Glob、Read工具）、处理并发请求（多Agent同时工作）。在这些场景下，Bun的性能比Node.js好不少。

但更关键的是**冷启动速度**。对于一个命令行工具来说，敲回车到看到响应之间的延迟直接影响体验。Bun的冷启动比Node.js快3-5倍。每天用几百次的工具，每次快200毫秒，累计起来感受明显。

为什么终端界面用React

这是最让人意外的选择。终端程序用React？

但如果你想想Claude Code的交互复杂度就理解了：实时更新的进度条、可折叠的代码diff、权限确认弹窗、多层嵌套的工具调用展示、多Agent并行时的分屏状态……这些东西的状态管理需求和Web前端本质上是同一个问题。用 `console.log` 拼字符串来处理这些，代码会变成无法维护的面条。

React的Ink框架让Anthropic可以用组件化的思路构建终端界面。每个UI元素是一个组件，状态变了自动重渲染。这是**复杂度管理**的选择，不是技术炫技。

社区在泄露后还发现了更底层的性能优化。 `ink/screen.ts` 和 `ink/optimizer.ts` 借鉴了**游戏引擎**的技术：用Int32Array支持的字符池代替字符串操作，用位掩码编码样式元数据，有专门的补丁优化器合并光标移动、取消冗余的hide/show操作。

还有自清除的行宽缓存，在token流式传输场景下减少了约50倍的 `stringWidth` 计算。Claude Code每秒可能重绘几十次，这些优化是流畅体验的必要条件。

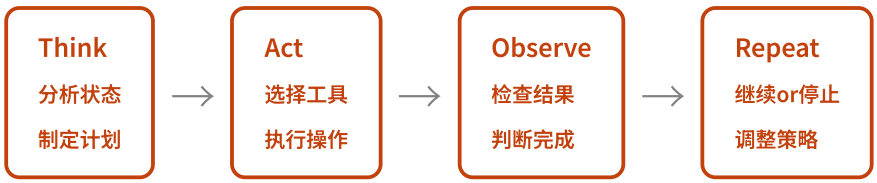
为什么Schema验证用Zod

TypeScript的类型只存在于编译期。代码运行时，AI返回的JSON可能是任何形状。Zod提供运行时的类型检查和转换，确保从API返回的数据符合预期格式。

这在AI应用中特别重要。模型的输出是不确定的，你不能假设它每次都返回正确的格式。Zod在「不确定的AI输出」和「确定性的程序逻辑」之间建立了一道防线。

核心循环：TAOR The TAOR Loop

Claude Code的心脏是一个叫**TAOR**的Agent循环：Think → Act → Observe → Repeat。



你在终端输入一句话后发生的所有事情，都是这个循环在驱动。模型先理解你要什么（Think），然后选一个工具执行操作（Act），观察工具返回的结果（Observe），判断任务是否完成。没完成就回到Think继续。一个任务可能要转几十圈才结束。

TAOR不是Claude Code独创的概念，但Claude Code的实现有几个有意思的设计细节：

工具调用是唯一的「行动」方式。模型不能直接操作文件系统或运行命令，必须通过工具间接执行。这意味着所有操作都经过一层中间件，可以在这层做权限检查、日志记录、安全审查。这是第5章权限系统的基础。

每次循环都是一次完整的API调用。每个Think-Act周期都需要把当前上下文发给API，等待模型返回工具调用指令，执行后再把结果加入上下文，再发一次API调用。这解释了为什么复杂任务耗费大量token。不是因为模型在「想太多」，而是循环的结构决定了每一步都要把完整上下文发过去。这也是第7章上下文管理如此重要的原因。

停止条件是模型自己决定的。循环什么时候停？当模型判断任务完成时，它不再调用工具，直接输出文本回答。这就是为什么给Claude明确的验证标准特别重要。如果需求描述模糊，模型不知道什么时候算「做完了」，循环会一直转下去。

五大子系统 The Five Subsystems

TAOR是骨架，但让Claude Code真正好用的是围绕这个骨架构建的五个子系统：

子系统	职责	代码量	章节
System Prompt	定义AI的身份、能力边界和行为准则	—	第3章
工具系统	59个工具的注册、调用和权限管理	~50,000行	第4章
查询引擎	所有LLM API调用、流式传输、缓存和编排	~54,000行	—
权限系统	操作的安全审查和访问控制	—	第5章
记忆系统	跨会话的持久化偏好和上下文	—	第6章

除了这五个核心子系统，还有几个重要的辅助模块：

IDE Bridge：VS Code和JetBrains与CLI之间的双向通信层。用JWT做认证，支持两个方向的消息传递。IDE可以向CLI发任务，CLI也可以向IDE推状态更新。这是Claude Code能同时作为独立CLI和IDE插件运行的基础。

上下文管理器：负责在对话变长时进行结构化压缩，以及管理prompt缓存策略。第7章详述。

多Agent协调器：管理多个Agent的创建、通信、隔离和结果合并。第9章详述。

一次请求的完整旅程 Anatomy of a Request

把上面的概念串起来，看看当你在Claude Code里输入「帮我写一个排序函数」时，完整的旅程是什么样的：

1. **System Prompt拼装**
   系统从缓存中加载静态prompt段（身份定义、安全准则、工具说明），再拼接动态段（你的CLAUDE.md、当前目录信息、记忆文件、git状态）。这可能是一个数千token的巨大prompt。

2. **API调用（Think）**
   你的消息连同system prompt和对话历史，一起发送给Claude API。查询引擎处理流式传输，你看到文字逐渐出现。

3. **工具调用（Act）**
   模型决定先读一下当前目录的文件结构（调用Glob工具），再看看有没有已有的排序相关代码（调用Grep工具）。每个工具调用都经过权限系统审查。

4. **结果注入（Observe）**
   工具的返回结果被添加到对话上下文中。模型现在知道了目录结构和现有代码。

5. **循环继续（Repeat）**
   模型决定用Write工具创建文件，然后用Bash工具运行测试。每一步都是一次新的API调用，带上所有之前的上下文。

6 **完成**

测试通过后，模型判断任务完成，输出总结文本。如果对话触发了记忆提取条件，后台还会fork一个子agent来提取值得记住的偏好信息。

看起来简单？实际上，即使是「写一个排序函数」这样的小任务，可能就涉及3-5次TAOR循环，每次循环都是一次完整的API调用。一个复杂任务可能有几十次循环，消耗几万token。

但这就是TAOR的魅力：**它不是在执行预设的脚本，而是在实时做决策。**每一步都在根据最新的观察调整策略。有时候试了一个方案发现不行，回退换另一条路。这不是bug，是设计的一部分。

代码规模透视 Code Scale in Perspective

最后看几个数字，帮你建立对这个项目规模的直觉：

指标	数量	说明
源文件总数	~1,900	全部TypeScript
代码总行数	~510,000	不含node_modules
工具系统	~50,000行	最复杂的模块之一
查询引擎	~54,000行	最大的单一模块（services/目录）
内置工具	59	每个独立权限控制
斜杠命令	~50	用户可调用的快捷命令
Feature Flags	44	控制未发布功能
src/子目录	41	高度模块化的项目结构

这是一个严肃的工程项目。不是三五个工程师周末写的hobby project，而是一个大团队持续投入、精心设计的产品级软件。

Hacker News上有人评论说Anthropic的代码像是vibe coded，「感觉对了就行」的写法。这个评价是否公允另说，但它说明一件事：**Claude Code的竞争力不在代码是否优雅，而在系统设计的决策是否正确。**做对了几个关键的架构选择，比把每一行代码写得漂亮重要得多。

接下来的十章，我们就逐一拆解这些关键决策。

03 System Prompt工程

System Prompt Engineering

当你在Claude Code里输入一句话，AI收到的远不止你写的那几个字。你的指令只是冰山一角，水面下是一个庞大的、精心组装的系统提示词。

你发一句话，AI收到一整本说明书 What the AI Actually Receives

源码的 `prompts.ts` 文件有911行，定义了系统提示词的完整构建过程。核心函数 `getSystemPrompt()` 返回的是一个字符串数组，每个元素是一个prompt段落。拼起来就是AI在你发消息前已经读到的所有指令。

这些指令分为几大类：

类别	源码函数	内容
身份定义	`getSimpleIntroSection()`	「你是Claude Code，Anthropic的官方CLI工具」
安全准则	`CYBER_RISK_INSTRUCTION`	安全边界：允许的和禁止的操作
系统行为	`getSimpleSystemSection()`	Markdown渲染、权限模式、hooks处理
做事准则	`getSimpleDoingTasksSection()`	不过度工程、不编造数据、不随意删文件
谨慎行动	`getActionsSection()`	高风险操作前确认、不用破坏性方式解决问题
工具使用	`getUsingYourToolsSection()`	优先用专用工具而非Bash
风格要求	`getSimpleToneAndStyleSection()`	不用emoji、简洁直接、引用代码格式
输出效率	`getOutputEfficiencySection()`	开门见山、先行动后解释

以上是**静态**部分，所有用户共享，可以跨请求缓存。在这些之后，还有**动态**部分：你的CLAUDE.md配置、当前目录信息、记忆文件、MCP服务器说明、git状态、语言偏好……

光静态部分估计就有3000-5000个token。加上动态部分，一个典型的system prompt可能在5000-10000 token之间。你还没说第一句话，AI已经读了一本小册子。

分界线：一刀切出缓存策略 The Dynamic Boundary

源码里有一个关键常量：

```
export const SYSTEM_PROMPT_DYNAMIC_BOUNDARY =
  '__SYSTEM_PROMPT_DYNAMIC_BOUNDARY__'
```

这是一条分界线，把整个system prompt一刀切成两段。分界线上面是静态内容，所有用户共享，可以用 `scope: 'global'` 跨组织缓存。分界线下面是动态内容，每个用户独立加载。

为什么这么做？**省钱。**

Claude API的prompt缓存机制是：如果两次请求的prompt前缀完全相同，第二次可以复用第一次的缓存，不需要重新处理。这意味着所有用户共享的静态段只需要处理一次。

源码注释很直白地解释了设计意图：

```
/**
 * Session-variant guidance that would fragment the cacheScope:'global'
 * prefix if placed before SYSTEM_PROMPT_DYNAMIC_BOUNDARY. Each conditional
 * here is a runtime bit that would otherwise multiply the Blake2b prefix
 * hash variants (2^N). See PR #24490, #24171 for the same bug class.
 */
```

翻译一下：如果在分界线上面放了任何「有条件」的内容（比如根据用户类型显示不同文字），缓存命中率会指数级下降。每多一个条件分支，缓存变体数量翻倍（2^N）。这两个PR（#24490和#24171）就是修复这类把动态内容误放到静态区域的bug。

这是一个非常务实的工程决策。**prompt的物理排布不仅影响语义，更影响缓存命中率，直接关联运营成本。**

三种Prompt段落类型 Three Types of Sections

源码定义了三种注册system prompt段落的方式：

类型	函数	缓存行为	使用场景
静态段	直接字符串	全局缓存	身份定义、安全准则、做事准则
动态段	`systemPromptSection()`	按内容hash缓存	环境信息、记忆、语言偏好
不缓存段	`DANGEROUS_uncachedSystemPromptSection()`	不缓存	MCP服务器指令（随时可能变化）

名字叫 `DANGEROUS_uncached` 不是随便取的。源码注释说明了原因：

```
DANGEROUS_uncachedSystemPromptSection(
  'mcp_instructions',
  () ⇒ isMcpInstructionsDeltaEnabled()
    ? null
    : getMcpInstructionsSection(mcpClients),
  'MCP servers connect/disconnect between turns',
)
```

MCP服务器可能在对话中途连接或断开，所以它的指令不能缓存，缓存了可能会给模型过时的信息。但不缓存意味着每次API调用都要重新发送这部分内容，增加成本。所以Anthropic给它加了 `DANGEROUS_` 前缀作为警告：**用这个函数之前三思，确定真的不能缓存。**

内部版 vs 外部版的prompt差异 Internal vs External Prompts

源码中大量 `process.env.USER_TYPE === 'ant'` 判断，让内部版和外部版的system prompt有显著差异：

代码风格指令

外部版没有特别的代码注释要求。内部版多了四条严格要求：

```
// 内部版独有
- Default to writing no comments. Only add one when the WHY
  is non-obvious.
- Don't explain WHAT the code does, since well-named
  identifiers already do that.
- Don't reference the current task, fix, or callers.
- Before reporting a task complete, verify it actually works.
```

最后一条特别重要。「报告完成前必须验证」，源码注释标注这是「capy v8 thoroughness counterweight」，意思是针对Capybara模型（内部代号）v8版本的「不够仔细」倾向的反制措施。**他们发现模型有时候会说「完成了」但其实没验证，所以在prompt里明确要求验证。**

诚实性指令

内部版还有一段很长的反虚假声明：

```
Report outcomes faithfully: if tests fail, say so with the
relevant output; if you did not run a verification step, say
that rather than implying it succeeded. Never claim "all tests
pass" when output shows failures...
```

这段指令同样带有模型版本标注：「False-claims mitigation for Capybara v8 (29-30% FC rate vs v4's 16.7%)」。翻译：Capybara v8的虚假声明率是29-30%，比v4的16.7%高了很多，所以需要更强的prompt来抑制。

这个数据很有价值。它告诉我们：**即使是最先进的模型，虚假声明率也不低**。Anthropic对自己模型的缺陷有清醒的认识，并用prompt工程来补偿。

输出风格

差异最大的部分。外部版只有4行：「开门见山、简洁、别废话」。内部版是一整段长文，核心要求是：

「你写的是给人读的文字，不是往控制台打日志。假设用户已经离开了，丢失了上下文，他们不知道你中间用了什么代号、什么缩写。写的时候要让他们能cold pick up。」

还有具体的写作要求：用连贯的散文，避免碎片式表达、过多的破折号、符号和记号。不要把解释性推理塞进表格格子里。避免语义回溯，让读者能线性阅读，不需要回头重新理解前面的内容。

外部版为什么没有这些？可能是因为外部版需要更简洁（大多数用户是开发者，习惯看代码式输出），也可能是还在A/B测试阶段。源码注释确实标注了「un-gate once validated on external via A/B」。

五层优先级覆盖 Five-Level Priority Override

源码的 `systemPrompt.ts` 揭示了一个更复杂的设计：system prompt不是一个固定模板，而是一个**五层优先级系统**，从高到低依次覆盖：

优先级	来源	使用场景
1（最高）	**overrideSystemPrompt**	测试/调试模式，完全替换
2	**Coordinator Mode**	多Agent协调模式（第9章详述）
3	**Agent System Prompt**	特定领域agent的指令
4	**Custom System Prompt**	用户通过 `--system-prompt` 指定
5（最低）	**Default System Prompt**	标准Claude Code指令

还有一个特殊的 `appendSystemPrompt` ，不论使用哪层，都会追加到末尾。

这个设计支持了Claude Code从单一CLI到多模式平台的演进：普通用户看到Default，子agent看到Agent prompt，Coordinator模式有自己完全不同的角色定义，而开发者测试时可以用override完全替换。

三种运行模式 Three Operating Modes

源码实际上实现了三套完全不同的system prompt，根据运行模式切换：

模式A：简化模式（CLAUDE_CODE_SIMPLE=1）

```
You are Claude Code, Anthropic's official CLI for Claude.
CWD: /path/to/project
Date: 2026-04-02
```

就这么多。三行。用于嵌入式场景（比如作为子进程被调用），不需要完整的行为指导。

模式B：标准交互模式

就是前面详述的完整prompt，8个静态段 + N个动态段，几千token。

模式C：Proactive模式（KAIROS/助手模式）

源码里有一段分叉逻辑：如果检测到处于Proactive模式（KAIROS/助手模式），整个system prompt会切换到一套完全不同的极简版本：

```
if (proactiveModule?.isProactiveActive()) {
  return [
    `You are an autonomous agent. Use the available tools
     to do useful work.`,
    getSystemRemindersSection(),
    await loadMemoryPrompt(),
    envInfo,
    // ...
  ]
}
```

从「你是Claude Code，帮用户做编程任务」变成「你是一个自主Agent，用工具做有用的事」。身份定义完全不同，行为约束也大幅简化。这暗示了KAIROS不是Claude Code的增强版，而是一个概念上不同的产品——一个**不等用户指令就主动行动的AI**。

缓存TTL的稳定性工程 Cache TTL Stability

一个容易被忽略但非常精巧的设计：缓存TTL的稳定性处理。

Claude API支持两种缓存TTL：5分钟（默认）和1小时（付费用户）。源码中用户是否有资格使用1小时缓存的判断，被**锁定在session启动时**：

```
let userEligible = getPromptCache1hEligible()
if (userEligible === null) {
  userEligible = process.env.USER_TYPE === 'ant' ||
    (isClaudeAISubscriber() && !currentLimits.isUsingOverage)
  setPromptCache1hEligible(userEligible)  // 锁定
}
```

为什么要锁定？因为如果用户在会话中途从免费变成付费（或反过来），TTL会从5分钟变成1小时。TTL变化会改变cache key，导致整个prompt cache失效，大约浪费20,000 token的重新处理。

所以Anthropic选择在session开始时就决定TTL，整个session内不变。**稳定性比精确性更重要。**偶尔用错TTL的成本远低于mid-session cache失效的成本。

这个系统能教我们什么 Lessons for AI Product Builders

> **核心建议**
>
> **System prompt是一个工程系统，不是一段文字。**它需要像代码一样管理版本、测试效果、优化性能。Claude Code的prompt有明确的缓存策略、模块化结构、AB测试标注、模型版本兼容处理。这不是「写一段好的指令」的问题，而是「构建一个可维护的指令系统」的问题。

几个关键takeaway：

缓存决定prompt结构。不要想好了内容再随便排。先划出缓存边界，把不变的放前面，变的放后面。每一个放错位置的条件分支都在浪费缓存预算。

用prompt补偿模型缺陷。Anthropic不会假装自己的模型完美。Capybara v8虚假声明率30%？那就在prompt里加反虚假声明指令。模型不够仔细？加「完成前必须验证」。**了解你的模型的弱点，然后用prompt targeting来修补。**

内外部版本暴露了理想状态。内部版的那些严格要求，就是Anthropic认为AI应该怎样工作的理想状态。如果你在设计自己的AI产品，内部版的prompt是个很好的参考基准。

不同模式需要不同的prompt。Proactive模式的prompt和普通模式完全不同。不要试图用一个通用prompt覆盖所有使用场景。场景不同，身份定义和行为约束都应该不同。

缓存系统比你想象的脆弱。社区发现了一个真实案例：Claude Code v2.1.76前后出现了prompt cache回归bug。两个技术原因叠加：每次请求中attestation数据的变化导致缓存哈希失效，以及anti-distillation注入的假工具定义在每次请求中不同。

正常情况下Turn 4的成本应该降到Turn 1的13%（缓存命中），但bug版本中成本反而从$0.04膨胀到$0.40。**缓存前缀中任何微小的变化都会导致整个缓存失效**，成本膨胀10-20倍。这不是Claude Code特有的问题，而是所有使用prompt缓存的AI产品都要警惕的陷阱。

04 工具系统：4个原语，59个工具

The Tool System — Four Primitives, Fifty-Nine Tools

Claude Code的工具系统可能是整个源码中规模最大的部分，约50,000行代码，59个独立的工具目录。但所有这些工具，归结到底只做四件事。

四个能力原语 Four Capability Primitives

59个工具听起来很多，但如果退后一步看，所有能力归结为4个原语：

原语	做什么	代表工具
Read	读取信息	FileRead、Grep、Glob、WebFetch、WebSearch
Write	创建或修改	FileWrite、FileEdit、NotebookEdit
Execute	运行命令	Bash、PowerShell、REPL
Connect	连接外部系统	MCP工具、Agent、SendMessage、TeamCreate

这个分类不是我总结的，是从源码的设计意图中提炼的。工具按照这四个维度划分权限等级：Read通常是低风险的（不改变任何东西），Write和Execute是中高风险的（会改变文件或系统状态），Connect涉及外部交互。

工具目录全景 The Complete Tool Directory

源码的 `src/tools/` 目录下有59个工具目录。按功能分类：

类别	工具数量	代表工具
文件操作	5	FileRead, FileWrite, FileEdit, Glob, Grep
命令执行	3	Bash, PowerShell, REPL
Web访问	3	WebFetch, WebSearch, WebBrowser
Agent协调	9	Agent, SendMessage, TeamCreate, TeamDelete, TaskCreate/Get/List/Update/Stop
计划与工作区	4	EnterPlanMode, ExitPlanMode, EnterWorktree, ExitWorktree
MCP相关	4	MCPTool, McpAuth, ListMcpResources, ReadMcpResource
技能与配置	4	Skill, DiscoverSkills, Config, ToolSearch
内部专用	7+	Tungsten, Monitor, ReviewArtifact, SuggestBackgroundPR...
KAIROS专用	3	PushNotification, SendUserFile, SubscribePR
其他	若干	Sleep, Brief, TodoWrite, NotebookEdit, RemoteTrigger...

Bash：万能适配器 Bash as the Universal Adapter

在59个工具中，Bash是最特殊的一个。它是一个**万能适配器**，让Claude能使用人类开发者的一切命令行工具。

不需要给每种编程语言做专门集成，不需要为每个框架写插件。`npm install`、`python test.py`、`git push`、`docker build`，通过Bash就能操作一切。这也是Claude Code能在几乎任何技术栈的项目里工作的原因。不像某些IDE插件只支持特定语言。

但源码对Bash的使用有严格的限制。system prompt中明确要求：

```
Do NOT use the Bash to run commands when a relevant
dedicated tool is provided.
- To read files use Read instead of cat, head, tail
- To edit files use Edit instead of sed or awk
- To search for files use Glob instead of find or ls
- To search content use Grep instead of grep or rg
```

为什么不让AI用Bash做一切？因为**专用工具比万能工具更可控**。Read工具只能读文件，权限系统对它的审查很简单。Bash命令可以做任何事，权限系统需要理解命令的语义才能判断是否安全，这要难得多。

所以Bash是最后手段，不是首选。能用专用工具的场景，永远用专用工具。

Bash的安全分析引擎

Bash工具的安全分析非常精细。源码包含了完整的命令语义分析模块（`commandSemantics.ts`），能理解管道中每个命令的性质：

```
// 搜索类：find, grep, rg, ag, ack, locate, which
// 读取类：cat, head, tail, less, jq, awk, cut, sort
// 列表类：ls, tree, du
// 语义中立：echo, printf, true, false
```

关键设计：**管道中所有部分都必须是搜索/读取命令**，整个管道才被认为是read-only。例如 `cat file | jq .field` 是安全的（都是read），但 `find . | xargs rm` 不安全（包含写操作）。

还有AST级别的安全解析（`bashSecurity.ts`），不只是看命令名，而是解析整个命令的语法树，检测输出重定向、命令替换等可能隐藏危险操作的模式。

Deferred Tools：延迟加载的智慧 Deferred Tools — Load on Demand

这是一个很聪明的优化。源码中有一个 `ToolSearchTool`，它的作用是让模型按需加载工具定义。

为什么需要延迟加载？因为每个工具的描述（名称、参数schema、使用说明）都需要占用system prompt的token。59个工具如果全部展开，可能要占用几万token。对于大多数任务，用户只需要其中的5-10个。

Deferred Tools的机制是：系统一开始只告诉模型「有这些工具可用」（名称列表），但不提供完整的参数定义。当模型判断需要使用某个工具时，先调用 `ToolSearch` 获取完整定义，然后再调用实际工具。

这个设计在MCP场景下尤其重要。如果你连接了5个MCP服务器，每个暴露10个工具，那就是50个额外的工具定义。全部加载可能消耗20,000-30,000 token的上下文。Deferred加载让大部分工具在不需要时不占用任何上下文空间。

源码注释提到，每个MCP服务器的工具定义大约固定消耗4,000-6,000 token。**工具不是越多越好，每一个都有认知成本。**

工具的插件式架构 Plugin Architecture

每个工具都是一个独立的目录，包含以下文件：

```
src/tools/BashTool/
├── BashTool.tsx      # 工具实现（含UI组件）
├── prompt.ts         # 工具描述和参数定义
├── toolName.ts       # 工具名称常量
├── bashSecurity.ts   # AST级安全解析
├── commandSemantics.ts  # 命令语义分析
└── utils.ts          # 工具函数
```

这种结构意味着添加新工具非常简单：创建一个新目录，实现标准接口，注册到工具列表。不需要修改核心代码。这就是为什么Claude Code能以很快的速度不断添加新工具，从最初的十几个到现在的59个。

每个工具都实现了统一的 `Tool` 接口，包括：

- **name**：工具名称
- **description**：给模型看的工具描述
- **parameters**：Zod schema定义的参数格式
- **execute**：实际执行逻辑
- **permissionContext**：权限检查需要的上下文

FileEdit的精妙设计 The Elegance of FileEdit

很多人第一反应是：让AI编辑文件，直接用 `sed` 或者重写整个文件不就行了？Claude Code的设计者显然不这么想。

FileEdit的核心机制是**唯一匹配**。当你调用Edit工具时，你需要提供 `old_string` （要替换的内容）和 `new_string` （替换后的内容）。系统会在文件中搜索 `old_string` ，如果找到多于一处匹配，直接报错：

```
if (matches > 1 && !replace_all) {
  return {
    result: false,
    message: `Found ${matches} matches of the string
to replace, but replace_all is false. To replace
all occurrences, set replace_all to true. To replace
only one occurrence, please provide more context to
uniquely identify the instance.`,
    errorCode: 9,
  }
}
```

这个设计解决了AI编辑文件时最常见的事故：**改错了地方**。当一个函数名在文件中出现5次，AI只想改其中一处时，唯一匹配约束会迫使AI提供更多上下文（比如包含前后几行代码），直到能精确定位到目标位置。这比sed的行号匹配可靠得多。行号在文件编辑过程中会变化，但文本内容的上下文不会。

如果确实想批量替换呢？把 `replace_all` 设为 `true` 。这个参数的典型场景是重命名变量——一个变量名在文件中出现20次，你确实要全部替换。prompt中也明确告诉模型：「Use `replace_all` for replacing and renaming strings across the file.」

还有一个容易被忽略的防护：**Edit之前必须先Read**。源码中检查了 `readFileState` ，如果这个文件还没被读取过，直接拒绝编辑。这防止了AI在不了解文件内容的情况下盲目修改，也确保了唯一匹配检查基于最新的文件状态。

```
const readTimestamp =
  toolUseContext.readFileState.get(fullFilePath)
if (!readTimestamp || readTimestamp.isPartialView) {
  return {
    result: false,
    message: 'File has not been read yet.
      Read it first before writing to it.',
    errorCode: 6,
  }
}
```

更进一步，源码还检查了文件在读取后是否被修改过（通过时间戳比对）。如果用户或linter在AI读取后修改了文件，Edit会要求AI重新读取，防止基于过时内容做替换。

为什么这比sed好？ `sed` 是行导向的文本处理工具，用正则表达式匹配。AI生成的正则表达式经常出错，转义字符、贪婪匹配、多行模式，每一个都是坑。而Edit工具用的是精确字符串匹配，不需要正则。AI只需要复制粘贴它刚刚读到的文本，然后写出修改后的版本。这把错误率从「经常出问题」降到了「几乎不出问题」。

工具结果的大小控制 Tool Result Size Control

这是一个很容易被忽略但非常关键的机制。每个工具都有一个 `maxResultSizeChars` 属性，定义了工具返回结果的最大字符数：

```
/**
 * Maximum size in characters for tool result
 * before it gets persisted to disk.
 * When exceeded, the result is saved to a file
 * and Claude receives a preview with the file
 * path instead of the full content.
 *
 * Set to Infinity for tools whose output must
 * never be persisted (e.g. Read, where persisting
 * creates a circular Read→file→Read loop and the
 * tool already self-bounds via its own limits).
 */
maxResultSizeChars: number
```

大多数工具的阈值是100,000字符。当一个工具的返回结果超过这个阈值时，系统不是直接截断，而是把完整结果保存到磁盘上的一个临时文件，然后在对话中只放一个摘要和文件路径。这样Claude可以在需要时通过Read工具去读取完整结果。

为什么不直接截断？因为截断可能丢失关键信息。而保存到磁盘的方案保留了完整数据，只是不把它塞进上下文窗口。这是一个**把上下文窗口当稀缺资源管理**的思路。

但有一个例外：FileReadTool的 `maxResultSizeChars` 被设为 `Infinity` 。源码注释解释了原因：如果Read工具的结果被保存到文件，Claude需要用Read工具去读那个文件，这又可能产生一个新的大结果，再被保存到文

件……形成无限循环。Read工具通过自己的行数限制（默认2000行）来控制输出大小，不需要外部的持久化兜底。

这个设计说明了一点：**上下文窗口是AI产品最宝贵的资源之一**。如果让一次Grep搜索返回10万字符的结果塞满上下文，后续的对话质量必然下降。大小控制机制把工具结果的「存储」和「可见性」分离了。数据不丢，但不占用宝贵的上下文空间。

ToolSearch的匹配算法 ToolSearch Matching Algorithm

前面介绍了Deferred Tools的概念，这里深入看ToolSearch的匹配逻辑。当模型调用ToolSearch时，有两种查询模式：

精确选择模式：查询以 `select:` 开头时，直接按名称匹配。可以用逗号分隔选择多个工具，例如 `select:Read,Edit,Grep` 。这是模型已经知道要用什么工具时的快速路径。

关键词搜索模式：不以 `select:` 开头时，进入关键词搜索。这时ToolSearch需要理解工具名称的语义。源码中有两个关键的名称解析逻辑：

对于普通工具名（CamelCase格式），按驼峰分词：

```
// Regular tool - split by CamelCase and underscores
const parts = name
  .replace(/([a-z])([A-Z])/g, '$1 $2')
  .replace(/_/g, ' ')
  .toLowerCase()
  .split(/\s+/)
  .filter(Boolean)
// FileEditTool → ['file', 'edit', 'tool']
```

对于MCP工具名（ `mcp__server__action` 格式），按双下划线和单下划线拆分：

```
if (name.startsWith('mcp__')) {
  const withoutPrefix =
    name.replace(/^mcp__/, '').toLowerCase()
  const parts =
    withoutPrefix.split('__').flatMap(p ⇒ p.split('_'))
  // mcp__slack__send_message
  //   → ['slack', 'send', 'message']
}
```

搜索还支持 `+` 前缀来标记必须匹配的关键词（如 `+slack send` 要求工具名中必须包含slack），以及按MCP前缀批量匹配（如查询 `mcp__slack` 返回所有Slack相关工具）。

搜索得分综合考虑了工具名称的分词匹配、工具描述中的关键词匹配，以及可选的 `searchHint` 字段匹配。这让模型能用自然语言描述需求（如查询「modify file」），ToolSearch就能返回FileEdit工具。

这个系统能教我们什么 Lessons for AI Product Builders

> **核心建议**
>
> **控制AI的最好方式不是写更长的prompt，而是设计更好的工具。**工具定义了AI能做什么、不能做什么。每个工具的参数schema就是一组约束。AI只能在这些约束范围内操作。

万能工具是后备，不是首选。Bash能做一切，但正因如此，它最难控制。优先使用功能明确、权限清晰的专用工具。万能工具只在专用工具覆盖不到的场景使用。

工具有token成本。每个工具描述都占用上下文空间。59个工具全加载，成本很高。Deferred加载是必要的优化，尤其在MCP扩展工具数量的场景下。

插件式架构支持快速迭代。添加新工具不需要动核心代码，这让团队可以并行开发不同的工具。对于一个快速迭代的AI产品，这种架构的灵活性至关重要。

05 权限系统：信任是设计出来的

The Permission System — Trust by Design

你用Auto模式的时候，以为什么操作都直接放行？其实背后跑着两个AI：一个执行你的任务，另一个专门审查第一个的操作是否安全。

三种权限模式 Three Permission Modes

Claude Code有三种权限模式，源码中的定义很清晰：

模式	行为	适用场景
default / interactive	每个危险操作都弹窗确认	日常使用，安全优先
auto	ML分类器自动判断是否安全	熟练用户，效率优先
yolo / bypassPermissions	跳过所有确认	非交互式脚本，完全信任

大多数人关心的是Auto模式。它怎么决定一个操作是否安全？

Auto模式的四层审查流水线 The Four-Layer Pipeline

Auto模式不是简单地让AI自己决定。它是一个**四层流水线**，每层像一道门禁，越往后越严格：

1. **规则匹配**

   检查用户预设的权限规则。如果某个操作已经被明确允许或禁止（比如在settings.json里配置了「允许所有git命令」），直接按规则执行，不需要走后面的步骤。

2. **低风险跳过**

   某些操作天然是低风险的（比如读取文件、搜索代码），直接放行。不需要浪费一次AI分类器调用来判断「读一个文件是否安全」。

3. **白名单放行**

   只读工具（Read、Grep、Glob等）在白名单上，直接放行。这些工具从设计上就不会修改任何东西。

4. **ML分类器判断**

   前三层都没拦截的操作，才会调用一个独立的Claude Sonnet做安全分类。这个分类器有自己专门的system prompt，和主AI完全不同，专门用来判断操作风险等级。

为什么这么设计？**性能和成本**。每次调用ML分类器都是一次额外的API调用，有延迟也有费用。四层流水线确保只有真正需要判断的操作才走到第四层。绝大多数日常操作在前三层就处理了。

YOLO分类器的实现 The YOLO Classifier

第四层的ML分类器在源码中叫 `yoloClassifier` 。它的实现在 `src/utils/permissions/yoloClassifier.ts` 。

分类器有自己的system prompt，存在一个独立的txt文件中（ `auto_mode_system_prompt.txt` ）。它接收主AI想要执行的操作信息，然后输出三种判断之一：

判断	含义	后续动作
Allow	操作安全，放行	直接执行
Soft Deny	有风险，需确认	弹窗让用户确认
Hard Deny	危险操作，拦截	直接拒绝执行

分类器实际上采用了**两阶段判断**，这是源码中最精巧的设计之一：

Stage 1：快速判断（64 token预算）

分类器先用极小的token预算（仅64 token）做第一轮判断。输出格式很简单： `<block>yes/no</block>` 。如果Stage 1判断为「安全」，直接放行，不需要进入Stage 2。

为什么只给64 token？因为大多数操作是明显安全的（读文件、搜索代码），不需要深度推理。64 token足够输出一个yes/no。

Stage 2：深度推理（4096 token预算）

只有Stage 1判断为「可能危险」的操作才进入Stage 2。这次给了4096 token的预算，要求分类器输出详细的推理过程和拒绝理由： `<block>yes/no</block><reason>explanation</reason>` 。

两阶段的好处是**成本优化**：安全操作在Stage 1就放行了（耗费极少token），只有可疑操作才需要Stage 2的深度分析。据源码统计，大多数操作不需要进入Stage 2。

几个其他技术细节也很关键：

温度设为0。两个阶段的temperature都设为0，确保相同输入产生相同输出。安全判断不需要创造性，需要确定性。

CLAUDE.md被注入分类器上下文。分类器不只看操作本身，还看你的CLAUDE.md配置。如果你在CLAUDE.md里写了「允许对/tmp的任何操作」，分类器会把这作为用户意图来参考。这意味着CLAUDE.md不仅影响主AI的行为，也影响安全判断。

内外部版本用不同的权限模板。源码中加载了两套权限模板： `permissions_external.txt` 和 `permissions_anthropic.txt` 。外部用户的自定义规则**替换**默认规则，内部员工的自定义规则**追加**到企业默认

规则。

转录格式压缩。分类器看到的对话历史不是完整的原始消息，而是一种紧凑格式。每个工具调用可以自定义自己的分类器输入格式（比如FileEdit只发送文件路径但省略文件内容），减少token消耗。

熔断机制 The Circuit Breaker

源码中有一个精巧的 `denialTracking.ts` 模块，实现了熔断机制：

```
export const DENIAL_LIMITS = {
  maxConsecutive: 3,
  maxTotal: 20,
} as const

export function shouldFallbackToPrompting(
  state: DenialTrackingState
): boolean {
  return (
    state.consecutiveDenials >= DENIAL_LIMITS.maxConsecutive ||
    state.totalDenials >= DENIAL_LIMITS.maxTotal
  )
}
```

规则很简单：**连续被拒3次，或者累计被拒20次，Auto模式自动降级为手动确认模式。**

为什么需要熔断？想象一个场景：主AI在执行一个合理的任务，但分类器反复误判为危险操作。如果没有熔断，AI会一直重试然后被拒，陷入死循环。熔断机制让系统在出问题时优雅降级，而不是卡死。

成功执行一次操作会重置连续拒绝计数器（但不重置总计数器）。这意味着偶尔被拒不会触发熔断，但频繁被拒一定会触发。

受保护文件和路径防御 Protected Files and Path Defense

源码中定义了一系列受保护的文件，任何操作都不允许修改：

- `.gitconfig` — Git全局配置
- `.bashrc` / `.zshrc` — Shell配置
- `.claude.json` — Claude Code自身的配置
- `.mcp.json` — MCP服务器配置

这些文件之所以受保护，是因为修改它们可能导致持久的、跨会话的危害。比如，如果AI修改了 `.bashrc` ，加入了一条恶意命令，那么用户每次打开终端都会执行这条命令。

路径穿越防御也很完善。源码检查了unicode编码、大小写变换、反斜杠注入等常见的路径穿越攻击手法。防止AI（或通过AI的prompt注入攻击者）用 `../../../.bashrc` 这样的路径绕过保护。

权限解释文本 Permission Explanation Text

一个容易被忽略的细节：当分类器拒绝一个操作时，它不只是简单地说「不允许」。它还会生成一段人类可读的解释文本，告诉用户为什么这个操作被认为是危险的。

这段解释文本是由另一个独立的LLM调用生成的。Anthropic认为，如果你要拒绝用户的操作，至少要告诉他们原因，而且原因要是人能理解的自然语言，不是技术错误码。

危险权限的自动剥离 Dangerous Permission Stripping

这是一个很微妙的安全问题：用户在日常使用中可能配置了一些宽泛的权限规则，比如 `Bash(*)`（允许所有bash命令）或 `Bash(python:*)`（允许所有python命令）。在手动确认模式下，这些规则没什么问题，用户每次都会看到操作内容再确认。但在Auto模式下，这些规则会**绕过分类器**。

为什么？因为Auto模式的四层流水线中，第一层就是规则匹配。如果用户的规则说「允许所有Bash命令」，那么任何Bash操作都会在第一层就被放行，根本走不到第四层的ML分类器。这意味着分类器的安全审查形同虚设。

源码的解决方案很优雅：进入Auto模式时，自动**剥离**这些危险的宽泛规则，并把它们暂存起来；退出Auto模式时再**恢复**。

```
export function stripDangerousPermissionsForAutoMode(
  context: ToolPermissionContext,
): ToolPermissionContext {
  // 收集所有 alwaysAllowRules
  const rules: PermissionRule[] = []
  for (const [source, ruleStrings] of
    Object.entries(context.alwaysAllowRules)) {
    // ...解析每条规则
  }
  // 找出其中危险的规则
  const dangerousPermissions =
    findDangerousClassifierPermissions(rules, [])
  if (dangerousPermissions.length === 0) {
    return context // 没有危险规则，直接返回
  }
  // 剥离危险规则，同时暂存到 strippedDangerousRules
  return {
    ...removeDangerousPermissions(
      context, dangerousPermissions),
    strippedDangerousRules: stripped,
  }
}
```

哪些规则被认为是「危险的」？源码中定义了详细的匹配逻辑。以Bash为例，以下模式都会被剥离：

- `Bash`（无参数，即允许所有命令）

- `Bash(*)`（通配符，同上）
- `Bash(python:*)`（允许所有python命令，python可以执行任意代码）
- `Bash(node:*)` 、 `Bash(ruby:*)` 等脚本解释器的宽泛规则

退出Auto模式时， `restoreDangerousPermissions` 函数把暂存的规则恢复回去。用户的配置没有被永久修改，只是在Auto模式期间临时屏蔽。

这个设计的精妙之处在于：**它不信任用户的权限配置在自动化环境中仍然安全**。用户手动使用时设置的宽泛规则，可能在无人值守的自动模式下变成安全漏洞。系统在模式切换时自动适配安全等级，而不是把这个责任交给用户。

风险评估器 The Permission Explainer

当分类器判断一个操作需要用户确认时，用户看到的不只是一个「允许/拒绝」的按钮。系统会同时展示一段人类可读的解释，帮助用户做出判断。这段解释由一个独立的LLM调用生成。

源码中这个模块叫 `permissionExplainer` ，它的实现很有意思。首先是工具定义，系统使用 `tool_choice` 强制模型以结构化格式输出，确保每次都能得到可解析的结果：

```
const EXPLAIN_COMMAND_TOOL = {
  name: 'explain_command',
  input_schema: {
    properties: {
      explanation: {
        description:
          'What this command does (1-2 sentences)',
      },
      reasoning: {
        description:
          'Why YOU are running this command.
           Start with "I"',
      },
      risk: {
        description:
          'What could go wrong, under 15 words',
      },
      riskLevel: {
        enum: ['LOW', 'MEDIUM', 'HIGH'],
        description:
          'LOW (safe dev workflows),
           MEDIUM (recoverable changes),
           HIGH (dangerous/irreversible)',
      },
    },
  },
}
```

注意 `reasoning` 字段的设计：要求以「I」开头，从AI自身的视角解释为什么要执行这个操作。这不是随意的格式要求，它迫使模型生成第一人称的解释（「I need to check the test results」），而不是干巴巴的技术描述。用户读起来更自然，也更容易判断操作是否符合预期。

`risk` 字段限制在15个词以内。这迫使模型把风险浓缩到最核心的部分，而不是长篇大论列出所有可能的风险。用户在做权限确认时需要快速判断，不是阅读论文。

风险等级的三级定义也很实用：

等级	含义	典型操作
LOW	安全的开发工作流	运行测试、格式化代码
MEDIUM	可恢复的变更	安装包、修改配置文件
HIGH	危险/不可逆操作	删除文件、force push

系统还会把最近的对话上下文注入到评估请求中（最后3条AI消息，最多1000字符）。这让风险评估器不只看操作本身，还能理解操作的上下文。同样是 `rm` 命令，在「清理临时文件」的上下文中是LOW风险，在没有任何上下文时可能被评为HIGH。

用户可以通过配置关闭这个功能（ `permissionExplainerEnabled` ），但默认是开启的。Anthropic的设计哲学是：**如果你要拒绝或质疑用户的操作，至少给一个人能理解的理由**。

Zig层的DRM：编译级安全 Zig-Layer DRM

社区在泄露后发现了一个出人意料的安全层：Claude Code的API请求中包含一个 `cch=00000` 占位符。在请求离开进程之前，Bun底层的**Zig编译代码**会把这5个零替换为一个计算出的加密哈希。服务端验证此哈希以确认请求来自真正的Claude Code二进制文件。

为什么用Zig而不是JavaScript？JavaScript可以在运行时被monkey-patch，你可以覆盖任何函数、拦截任何HTTP请求。但编译进Bun的Zig代码无法在运行时修改，除非重新编译整个运行时。这本质上是**API调用的DRM**，实现在HTTP传输层。

Bash安全的23项检查 23 Security Checks

`bashSecurity.ts` 包含23项编号的安全检查，覆盖了很多非显而易见的攻击面：Zsh等号展开防御（ `=curl` 在Zsh中会执行curl）、Unicode零宽空格注入、IFS空字节注入、18个被阻止的Zsh内建命令、以及HackerOne审查中发现的畸形token绕过。每一项背后都有一个真实的攻击场景。

这个系统能教我们什么 Lessons for AI Product Builders

> **核心建议**
>
> **安全系统的投入应该和核心功能一样多。**Claude Code的权限系统涉及多个文件、数千行代码、独立的ML分类器、Zig层DRM、23项Bash安全检查、熔断机制、路径防御。这不是一个周末写的feature，而是一个持续投入的核心系统。

多层防御，逐层过滤。不要把所有判断都压在一层上。简单的判断用简单的规则（白名单、规则匹配），复杂的判断才用复杂的方法（ML分类器）。这样既高效又可靠。

安全和效率不是对立的。四层流水线的设计让大多数操作在前三层就放行了，用户几乎感受不到安全检查的存在。只有真正可疑的操作才会触发第四层的AI判断，而这时候多等两秒是完全可接受的。好的安全系统是隐形的。

设计降级路径。熔断机制的存在说明Anthropic承认分类器不完美。与其追求完美的分类器（不可能），不如设计好出错时的降级行为。连续被拒就切回手动确认，简单、可靠、用户不会卡住。

保护持久化文件。AI修改临时文件影响有限。但修改Shell配置、系统配置这类持久化文件的影响是跨会话、甚至跨项目的。对这类文件的保护级别应该远高于普通文件。

权限策略要随模式切换而调整。用户在手动模式下设置的宽泛权限，在自动模式下可能是安全漏洞。好的系统会在模式切换时自动适配安全等级，而不是把这个判断留给用户。危险权限的自动剥离就是这个思路的体现。

给用户做判断的信息，而不只是按钮。一个「允许/拒绝」的弹窗，如果不告诉用户这个操作做什么、为什么要做、可能出什么问题，用户要么盲目允许，要么盲目拒绝。风险评估器生成的结构化解释，让权限确认从「打扰」变成了「informed decision」。

06 记忆系统：只记偏好，不记代码

The Memory System — Remember Preferences, Forget Code

Claude Code的auto memory可能是你用了之后觉得最惊艳的功能之一。但看了源码才知道，「记住什么」和「不记什么」之间的权衡，比表面看起来讲究得多。

记忆系统的物理实现 Physical Implementation

记忆系统的实现极其简单。没有数据库，没有向量存储，没有复杂的索引。就是文件系统上的一个目录：

```
~/.claude/projects/<project-slug>/memory/
├── MEMORY.md          # 索引文件，每次会话自动加载
├── user_role.md       # 用户身份信息
├── feedback_testing.md # 行为反馈
├── project_status.md  # 项目状态
└── reference_docs.md  # 外部资源指针
```

核心入口文件是 `MEMORY.md` ，源码中定义了它的硬性约束：

```
export const ENTRYPOINT_NAME = 'MEMORY.md'
export const MAX_ENTRYPOINT_LINES = 200
export const MAX_ENTRYPOINT_BYTES = 25_000
```

200行、25KB。这是MEMORY.md的物理上限。超过任一限制都会被截断，并追加一条警告信息。

为什么要这么严格？因为MEMORY.md每次对话都会被完整注入到system prompt中。如果不限制大小，记忆文件会不断膨胀，最终吃掉大量上下文窗口。200行 × ~125字符/行 ≈ 25KB，是Anthropic实测后认为「够用但不浪费」的平衡点。

四类记忆 Four Memory Types

源码的 `memoryTypes.ts` 定义了四种记忆类型，每种有不同的用途和存储策略：

类型	记什么	变化速度	例子
user	用户的身份、角色、偏好	很慢	「用户是高级工程师，喜欢TypeScript」
feedback	用户对AI行为的反馈	中等	「不要在测试中mock数据库」
project	项目的状态、目标、deadline	较快	「API重构目标是4月15日完成」
reference	外部系统的位置和用途	较慢	「bug追踪在Linear的INGEST项目」

这个分类不是随便定的。每种类型对应不同的使用策略。user和feedback类的记忆几乎在每次对话中都有用（它们影响AI怎么做事），project类只在相关上下文中有用，reference类只在用户提到外部系统时才需要。

核心设计决策：只记偏好，不记代码 The Core Design Decision

源码中有一段明确的「不记什么」的说明：

```
## What NOT to save in memory

- Code patterns, conventions, architecture, file paths,
  or project structure
- Git history, recent changes, or who-changed-what
- Debugging solutions or fix recipes
- Anything already documented in CLAUDE.md files
- Ephemeral task details
```

为什么不记代码相关的信息？一个例子就能说清楚：

假设AI记住了「验证函数 `validateUser` 在 `src/utils/auth.ts` 第42行」。下次对话时，你重构了代码，这个函数被移到了 `src/services/auth/validation.ts` 的第15行。AI的记忆现在是**错误的**，而且它不知道这个记忆已经过时了。

更糟的是，AI可能基于这条错误记忆直接开始修改错误的文件。记忆不像搜索。搜索永远给你最新结果，记忆给你的是「上次记住的东西」。

所以Claude Code的做法是：**代码相关的事实永远实时读取，不记忆。**记忆只存人的偏好和判断，这些东西变化慢，不会因为代码重构而失效。

> **设计原则**：持久化应该存「变化慢的东西」。模型的能力（搜索、读文件）可以实时获取「变化快的东西」。把变化快的东西放进记忆，是在制造定时炸弹。

记忆提取管线 The Extraction Pipeline

记忆提取不是每条消息都触发。源码中的限流逻辑确保提取不会过于频繁，只在AI完成一轮回答后才考虑触发。

提取工作由一个独立的**fork agent**完成。这个子AI继承了主对话的上下文（所以它知道刚才聊了什么），但权限被严格限制：只能读文件和写入记忆目录，连Bash命令都不能跑。

为什么用独立的agent而不是让主AI顺便做？两个原因：

上下文隔离。记忆提取的过程（分析对话、判断什么值得记、格式化记忆文件）会产生大量中间推理。如果在主对话中做，这些推理会占用主对话的上下文窗口。Fork agent有自己的上下文，不会影响主对话。

安全隔离。记忆文件是持久化的，跨会话生效。如果主AI可以自由写记忆，恶意的prompt注入攻击可能通过记忆系统实现持久化。在一次对话中注入恶意记忆，在未来所有对话中生效。限制记忆写入权限到一个专门的、权限最小化的agent，缩小了攻击面。

Cache共享设计。这是一个非常精巧的优化。Fork agent必须和主对话共享完全相同的system prompt、工具列表和模型，这样两者可以复用同一份prompt cache。如果fork改变了其中任何一个，cache key就不同了，fork需要重新处理整个prompt，速度会慢10倍以上。

所以fork agent的权限控制不是通过修改工具列表来实现的（那样会破坏cache），而是通过一个 `canUseTool` 钩子函数。工具列表一样，但钩子在运行时拦截不允许的操作。这个设计把「cache效率」和「安全隔离」两个看似矛盾的需求都满足了。

双向互斥

主agent和提取fork之间是互斥关系。源码会检测主agent是否已经在当前轮次中直接写入了记忆文件：

```
if (hasMemoryWritesSince(messages, lastMemoryMessageUuid)) {
  // 主agent已经写了，跳过后台提取
  return
}
```

如果主agent凭自己的judgment直接保存了记忆，后台提取就跳过这一轮。避免两个agent同时修改同一个记忆文件。

重叠请求的Stash模式

如果用户连续快速发多条消息，可能在上一轮提取还没完成时就触发了新一轮。源码的处理方式是**stash-and-continue**：不启动新的提取，而是把新请求暂存（stash）。当前一轮完成后，再处理暂存的请求。这避免了提取任务的堆积。

MEMORY.md的索引设计 Index Design

MEMORY.md不是记忆本身，而是一个**索引**。每条记忆的详细内容存在独立的文件中（如 `user_role.md`），MEMORY.md只包含一行指针：

```
- [用户身份](user_role.md) — 高级工程师，偏好TypeScript
```

这个设计解决了200行上限的约束问题。索引可以指向任意数量的记忆文件，每个文件可以有任意长度。MEMORY.md的200行限制只约束了「同时可见的记忆条目数量」，而不是「总记忆容量」。

每个记忆文件有标准的frontmatter格式：

```
---
name: 用户角色
description: 高级工程师，偏好TypeScript和简洁风格
type: user
---

记忆内容...
```

`description` 字段很重要——它不只是给人看的注释，而是被主动用于**记忆召回**。当新会话开始时，系统用一个独立的Sonnet侧查询（sideQuery），把所有记忆文件的description和当前对话上下文做匹配，选出最相关的几条加载到上下文中。

description写得太模糊（如「一些偏好」），匹配不到具体场景；写得太窄（如「关于React组件的测试」），其他场景下用不上。好的description应该具体到足够匹配，又抽象到足够泛化。

记忆的年龄感知

源码中还有一个精细的年龄感知机制。每条记忆在被召回时，系统会计算它的年龄，超过1天的记忆会附上一段陈旧警告：

```
export function memoryFreshnessText(mtimeMs: number): string {
  const d = memoryAgeDays(mtimeMs)
  if (d ≤ 1) return ''
  return `This memory is ${d} days old. Memories are
    point-in-time observations, not live state — claims
    about code behavior or file:line citations may be
    outdated. Verify against current code before
    asserting as fact.`
}
```

1天内的记忆不加警告，超过1天的会明确提示：「这是N天前的观察，不是当前状态。涉及代码行为或文件位置的声称可能已过时，请先验证再当真。」

这个设计防止了一个微妙的问题：AI可能基于3天前的记忆自信地说「这个函数在第42行」，但代码已经重构了。年龄警告让AI在使用旧记忆时保持怀疑，而不是盲目信任。

autoDream：睡眠中整理记忆 Memory Tidying in Sleep

还有一个叫autoDream的功能。触发需要通过**三重门**，全部条件满足才运行：时间门（距上次巩固≥24小时，通过文件mtime判断）、会话门（积累了≥5个新会话）、锁门（无其他进程正在巩固，通过文件锁实现）。另有10分钟的扫描节流防止频繁检查。如果运行失败，系统会回滚锁的mtime让下轮重试。

整理过程分四个阶段：

1. **Orient（定向）**
   审阅现有的记忆文件，了解当前记忆的全貌。

2. **Gather Signal（收集信号）**
   从最近的会话日志中提取值得记忆的信息。

3. **Consolidate（整合）**
   把新信息写入持久记忆文件。重要细节：相对日期（如「下周四」）必须转换为绝对日期，否则记忆在未来无法解读。

4. **Prune / Index（修剪/索引）**
   删除过时的记忆，确保MEMORY.md保持在200行/25KB以内。

名字叫Dream很有意味，像人在睡觉时大脑整理白天的记忆。autoDream在后台静默运行，用户完全不知道。它使用只读的Bash访问，是纯反思性的、非破坏性的整理。

记忆的信任问题 Trusting Recalled Memories

源码中有一段很重要的指令：

```
Memory records can become stale over time. Use memory as
context for what was true at a given point in time. Before
answering the user or building assumptions based solely on
information in memory records, verify that the memory is
still correct and up-to-date by reading the current state
of the files or resources.
```

翻译：**记忆是「某个时间点的真相」，不是「现在的真相」**。在用记忆做决策之前，先验证它还是不是对的。

这解决了一个微妙的问题。如果AI盲目信任记忆，旧记忆可能导致错误的行为。但如果AI完全不用记忆，记忆系统就没有意义。折中方案是：**把记忆当线索，不当真相**。记忆说「函数X在文件Y」，那就先去Y看看X是不是还在那儿。

这个系统能教我们什么 Lessons for AI Product Builders

> **核心建议**
>
> **最简单的存储方案往往是最好的。**纯文本文件、Markdown格式、文件系统目录结构。没有数据库、没有API、没有复杂的查询语言。这个方案的维护成本几乎为零，而且用户可以直接用文本编辑器查看和修改自己的记忆文件。

分类存储，分级信任。不同类型的记忆有不同的变化速率和使用频率。用户偏好几乎不变，每次都该考虑。项目状态变化快，只在相关时使用。一刀切的「全部记住」或「全不记」都不如分类管理。

给记忆系统设上限。200行、25KB。没有上限的记忆系统会膨胀到无法管理。上限迫使系统做取舍，只留真正有价值的记忆，丢弃琐碎的细节。

用独立agent做记忆提取。安全隔离 + 上下文隔离 + cache共享。这三个需求看似矛盾，但通过canUseTool钩子函数实现了优雅的统一。

记录成功，不只记录失败。Feedback类记忆的设计有一个反直觉的要求，不只记录用户的纠正（「不要这样做」），也要记录用户的验证（「是的，就这样」）。原因是，如果只记录失败，AI会变得过于保守，什么都不敢做。记录成功让AI知道哪些做法是被认可的，在类似场景下可以继续沿用。

时间规范化是必须的。用户说「下周四」，AI必须在记忆中存为「2026-04-10」。否则两周后回看这条记忆，「下周四」指的是哪个周四？这个规则看起来琐碎，但它解决了跨时间召回记忆时的歧义问题。

即使用户要求记，也要追问。源码有一条有趣的规则：当用户说「帮我记住这周的PR列表」时，AI不应该直接记PR列表（那是临时信息），而应该追问「其中什么是意外的或非显而易见的？」**那个**才值得记。这防止了记忆系统被当成笔记本用。

07 上下文管理：长对话的生存术

Context Management — Surviving Long Conversations

和Claude聊了很久之后，它偶尔会忘记你之前说过的某个要求。这不是它在敷衍你，而是上下文压缩的必然代价。源码告诉我们，Anthropic在「保留什么」上做了非常精细的设计。

压缩何时触发 When Compression Triggers

源码中有两种压缩路径：

自动压缩（autoCompact）：当对话的token数接近上下文窗口上限时，系统自动触发。源码中定义了精确的触发阈值：

```
AUTOCOMPACT_BUFFER_TOKENS = 13_000

autoCompactThreshold = effectiveContextWindow - 13_000
```

也就是说，当token用量达到「有效上下文窗口 - 13000」时触发自动压缩。对于一个200K的上下文窗口，大约在187K时开始压缩。13000 token的缓冲区给了系统足够的空间完成压缩操作本身。

手动压缩：用户执行 `/compact` 命令时触发。通常在感觉对话变慢、或者想清理上下文时使用。

还有一种**micro compact**，更轻量的压缩方式，在 `microCompact.ts` 中实现。它不压缩整个对话，只压缩工具调用的返回结果（比如一次grep搜索返回了几百行代码，压缩后只保留关键发现）。

源码还提到了两种额外的压缩模式：**反应式压缩**（reactive compaction，当API返回context too long错误时触发）和**上下文折叠**（context folding，一种实验性的更高效压缩方式）。这些模式和自动压缩互斥，同时只有一种生效。

9段式结构化压缩 The Nine-Section Structured Compression

压缩不是随便「总结一下」。源码的 `compact/prompt.ts` 中定义了一个严格的9段式结构化提取模板。模型必须按照以下格式输出压缩结果：

段落	内容	重要性
1. Primary Request and Intent	用户的所有显式请求和意图	最高
2. Key Technical Concepts	讨论过的技术概念、框架、技术	高
3. Files and Code Sections	具体的文件名、代码片段、修改记录	高
4. Errors and Fixes	遇到的错误和修复方法	中高
5. Problem Solving	解决问题的过程和正在进行的排查	中
6. All User Messages	所有非工具结果的用户消息	最高
7. Pending Tasks	明确被要求但尚未完成的任务	高
8. Current Work	压缩前正在做的工作的详细描述	最高
9. Optional Next Step	下一步计划（必须与用户最近的请求直接相关）	中

为什么所有用户消息必须完整保留 Why All User Messages Must Be Preserved

第6段是最关键的设计决策：**所有用户消息必须完整列出。**

源码prompt明确要求：「List ALL user messages that are not tool results. These are critical for understanding the users' feedback and changing intent.」

为什么？因为用户的每一句话都可能包含隐含的偏好和修正。

举个例子：在第3轮对话中，用户说了一句「别用var，用const」。如果压缩时丢掉了这句话，AI在后续的代码中可能又开始用var。用户以为AI已经学会了，但实际上那条修正被「遗忘」了。

用户的消息是压缩中损失最大的部分，因为它们不只是信息，还是约束。

analysis + summary的两阶段压缩 Two-Stage Compression

压缩过程分为两个阶段：

analysis阶段：模型先在 `<analysis>` 标签内做自由的分析思考，确保覆盖了所有要点。这个阶段的输出不会被保留到最终摘要中。

summary阶段：基于analysis的结果，按9段格式输出结构化摘要。这个摘要成为下一轮对话的初始上下文。

为什么需要两个阶段？因为直接要求模型输出结构化摘要，容易遗漏细节。analysis阶段让模型先「自由思考」，确保没有遗漏，然后再格式化输出。

源码注释解释了一个具体的工程问题：

```
// Aggressive no-tools preamble. On Sonnet 4.6+ adaptive-thinking
// models the model sometimes attempts a tool call despite the
// weaker trailer instruction. With maxTurns: 1, a denied tool
// call means no text output → falls through to the streaming
// fallback (2.79% on 4.6 vs 0.01% on 4.5).
```

翻译：压缩时模型只有1轮机会输出，如果它尝试调用工具（被拒绝），就不会产生任何文本输出，压缩就失败了。Sonnet 4.6的失败率是2.79%（4.5只有0.01%），所以他们加了一段非常激进的「禁止调用工具」前言来降低失败率。

这个细节暴露了一个真实的工程挑战：**不同模型版本的行为差异需要不同的prompt策略。**模型升级不总是免费的午餐，新版本可能引入新的问题。

第9段的防漂移设计 Anti-Drift Design

第9段（Next Step）有一段很长的警告：

```
IMPORTANT: ensure that this step is DIRECTLY in line with
the user's most recent explicit requests. Do not start on
tangential requests or really old requests that were already
completed without confirming with the user first.

Include direct quotes from the most recent conversation
showing exactly what task you were working on and where
you left off. This should be verbatim to ensure there's
no drift in task interpretation.
```

这段规则解决的是**任务漂移**问题。压缩之后，模型可能根据摘要中提到的某个旧任务开始工作，而不是继续用户最近要求的任务。要求引用原始对话的「verbatim quotes」，是为了让模型锚定在真实的用户请求上，而不是自己对请求的理解。

压缩Prompt的精妙细节 The Craftsmanship Behind the Compact Prompt

前面我们看了9段式结构的骨架，现在来看prompt原文中几个容易被忽略的精妙设计。

为什么第8段要求"direct quotes"

第8段（Current Work）和第9段（Next Step）的prompt原文中，有一段措辞值得逐字品味：

```
Include direct quotes from the most recent conversation
showing exactly what task you were working on and where
you left off. This should be verbatim to ensure there's
no drift in task interpretation.
```

注意这里的用词：不是"describe"，不是"summarize"，而是**"direct quotes"**和**"verbatim"**（逐字照抄）。

为什么非要引用原文？因为每次压缩，模型都在做一次「翻译」，把长对话翻译成短摘要。翻译就会丢失nuance。用户说的「把按钮改小一点」，如果被压缩成「调整按钮样式」，模型可能把颜色也改了。要求逐字引用，就是用原始文本来锚定模型的理解，防止每次压缩都偏移一点点，几次之后彻底跑偏。

这个问题叫**任务漂移**（task drift），是所有长对话AI系统的核心挑战。Anthropic的解法不是更复杂的算法，而是一条简单的prompt规则：让模型抄原文。有时候最有效的工程方案就是最笨的那个。

Partial Compact vs Full Compact

源码中有两种压缩模式：

维度	Full Compact（全量压缩）	Partial Compact（部分压缩）
压缩范围	整个对话历史	只压缩部分消息，保留其余原文
触发方式	自动压缩、手动/compact	智能选择分割点
Prompt缓存	完全失效	direction=from时可保留前半部分缓存
信息损失	较大	较小（保留了部分原始消息）

Partial Compact有两个方向：

direction = 'from'：保留前面的旧消息原文，只压缩后面的新消息。旧消息的Prompt缓存完全命中，不花钱重新处理。

direction = 'up_to'：压缩前面的旧消息，保留后面的新消息原文。Prompt缓存会失效（因为压缩摘要插到了最前面），但新消息保持了完整的上下文。

源码中Partial Compact的prompt也有微妙的差异。Full Compact的prompt说"create a detailed summary of **the conversation**"，而Partial Compact说"summary of **the RECENT portion** of the conversation — the messages that follow earlier retained context"。这不是随意措辞。它告诉模型：前面的消息还在，你只需要总结新的部分，不要重复已有的上下文。

Micro Compact：工具结果的精细压缩 Micro Compact — Surgical Compression of Tool Results

前面讨论的Full Compact和Partial Compact都是对整个对话的大手术。但Claude Code还有一种更精细的「微手术」：Micro Compact，定义在 `microCompact.ts` 中。

它解决什么问题

在实际使用中，工具返回的结果常常非常大。一次grep搜索可能返回几百行代码，一次文件读取可能返回上千行。这些工具结果在当时很有用，但5轮对话之后，它们就成了上下文中的「死肉」，占着位置却不再提供新信息。

Micro Compact不压缩整个对话，只针对单个工具的返回结果做清理。它把旧的工具结果替换成一条简短的占位信息：

```
const TIME_BASED_MC_CLEARED_MESSAGE = '[Old tool result content cleared]'
```

哪些工具会被压缩

不是所有工具结果都会被清理。源码中定义了一个白名单：

```
const COMPACTABLE_TOOLS = new Set([
  FILE_READ_TOOL_NAME,     // 文件读取
  ...SHELL_TOOL_NAMES,     // Shell命令
  GREP_TOOL_NAME,          // 搜索
  GLOB_TOOL_NAME,          // 文件查找
  WEB_SEARCH_TOOL_NAME,    // 网页搜索
  WEB_FETCH_TOOL_NAME,     // 网页抓取
  FILE_EDIT_TOOL_NAME,     // 文件编辑
  FILE_WRITE_TOOL_NAME,    // 文件写入
])
```

这些工具的共同特点是：返回结果可能很大，但核心信息已经被模型消化过了。模型读了一个1000行的文件，提取了关键信息后，那1000行原文就不需要再占上下文了。

两种触发方式

基于时间的触发：如果用户离开一段时间（比如上次对话过去了很久），服务器端的Prompt缓存已经失效，整个prefix都要重新处理。这时候不如趁机清理旧的工具结果，减少重新处理的token数量。源码的逻辑是：计算上次assistant消息到现在的时间差，超过阈值就清理。

基于缓存编辑的触发（Cached MC）：这是一种更高级的方式，利用API的cache_edits功能直接在服务器端删除缓存中的工具结果，不需要重新发送完整的消息。这样既节省了token，又不破坏已有的缓存。

> **核心建议**
>
> **Micro Compact的设计哲学值得学习。**与其等到上下文快满了再做一次大手术（Full Compact），不如在平时就持续做小清理。这类似于内存管理中的「增量GC」思想——频繁的小清理比偶尔的大清理对系统影响更小。如果你在构建长对话AI应用，考虑在全量压缩之前先做工具结果的增量清理。

压缩的成本：信息必然丢失 The Cost of Compression — Information Will Be Lost

理解了压缩机制之后，有一个事实必须正视：**无论压缩策略多精妙，信息丢失都是必然的。**

压缩的本质是用更短的文本去表达更长的对话内容。9段式结构再好，也只是让模型「有选择地遗忘」，而不是「不遗忘」。

哪些信息最容易丢失

信息类型	丢失风险	原因
隐含的偏好	高	用户说「别用var」，这种one-liner很容易被压缩掉
尝试过但失败的方案	高	Errors and Fixes段会保留错误，但试过的方案细节容易丢
中间讨论过程	高	讨论了A、B、C三种方案最终选了B，压缩后可能只剩「选了B」
代码的具体修改	中	第3段会保留文件和代码，但长会话中早期的修改可能被覆盖
用户的直接请求	低	第6段要求完整保留所有用户消息，这是保护最好的部分
当前正在做的工作	低	第8段要求逐字引用，保护力度强

如何减少压缩带来的信息损失

理解了压缩机制，你就能反过来利用它：

1. 把关键约束写进CLAUDE.md。CLAUDE.md在系统提示中，压缩根本不会碰它。如果你有一条规则需要整个项目期间都遵守（比如「使用TypeScript」「函数命名用camelCase」），写进CLAUDE.md比在对话中反复提醒有效得多。对话中说的会被压缩，CLAUDE.md里写的永远在。

2. 在长会话中定期重申关键要求。如果你发现AI开始「忘记」某个规则，不要生气。它可能刚刚经历了一次自动压缩，那条规则的原始上下文已经被替换成了摘要。重新说一遍就好。

3. 及时使用/compact而不是等到自动触发。手动压缩时你可以通过 `/compact` 命令附加自定义指令，告诉模型压缩时要特别保留哪些信息。自动压缩不会给你这个机会。

4. 大任务拆成多个会话。压缩是单个会话内的权宜之计。如果任务复杂到需要多次压缩，不如拆成多个会话，每个会话的CLAUDE.md里写清楚上下文。这比让模型在摘要的摘要上继续工作可靠得多。

5. 关注/compact后的第一轮回复。压缩之后的第一轮回复最容易出问题。如果发现AI理解偏了，立刻纠正。这一轮的纠正会成为压缩摘要后的新上下文，影响后续所有对话。

这个系统能教我们什么 Lessons for AI Product Builders

> **核心建议**
>
> **结构化压缩比自由总结可靠得多。**如果你让AI「总结一下前面的对话」，它可能遗漏关键细节。给它一个明确的模板，哪些信息必须保留、按什么格式组织，结果会好得多。9段式模板就是这个理念的具体实践。

用户消息是最不能丢的信息。工具返回的数据可以再查，代码可以再读，但用户说过的话代表了意图和约束。丢失用户消息等于丢失对齐。

防漂移需要主动设计。每次压缩都可能引入理解的微小偏差，几次压缩之后偏差会累积。要求verbatim quotes、限制下一步必须与最近请求相关，都是主动的防漂移措施。

模型升级需要prompt适配。4.5和4.6的失败率差异（0.01% vs 2.79%）说明，模型版本变化可能破坏已有的prompt策略。持续监控模型行为、及时调整prompt，是一个长期的工程任务。

08 搜索：为什么grep打败了RAG

Why grep Beats RAG

这可能是整个源码里最反直觉的发现。当整个行业都在推向量数据库和Embedding搜索时，Anthropic选择了最朴素的方案。

你可能以为的方案 What You'd Expect

如果你在2026年要做一个AI编程工具的代码搜索功能，业界的标准方案大概是这样的：

1. 把代码库切成小块（chunking）
2. 用Embedding模型把每个块变成向量
3. 存进向量数据库（Pinecone、Weaviate、Chroma之类）
4. 用户查询时，先把查询变成向量，再做相似度搜索
5. 把最相关的代码块喂给LLM

这就是RAG（Retrieval-Augmented Generation）。整个AI应用开发领域最热门的架构范式。无数创业公司在做RAG相关的工具和基础设施。

Claude Code实际的方案 What Claude Code Actually Does

grep。和ripgrep。

没有Embedding。没有向量数据库。没有语义搜索。就是纯文本的模式匹配。

源码里的搜索相关工具就两个：`Grep`（基于ripgrep的内容搜索）和`Glob`（基于文件名模式匹配）。此外还有`Read`工具直接读取文件内容。没了。

你觉得它搜代码搜得很准？背后就是grep加上一个足够聪明的大脑。

为什么这样做反而更好 Why This Works Better

这个决策背后的逻辑其实很清晰，一旦想通了就觉得理所当然：

1. LLM已经够聪明了

RAG解决的核心问题是：搜索引擎不够聪明，需要语义理解才能找到相关内容。但当消费搜索结果的是一个世界顶级的LLM时，情况变了。

Claude Opus可以从一堆grep结果里理解代码之间的关系、推断调用链、识别设计模式。它不需要一个很聪明的搜索引擎来预先筛选和排序，它自己就是最强的「理解引擎」。

与其让每个环节都变复杂，不如让一个环节足够强，其他环节保持简单。

2. grep的优势被低估了

grep做一件事，做得非常好：精确的文本匹配。它告诉你「这个字符串在哪些文件的哪些行出现过」，100%准确，没有误报，几乎没有延迟。

对比RAG的Embedding搜索：结果是概率性的，可能返回「语义相关但实际无关」的内容，也可能遗漏「写法不同但实际相关」的内容。还需要调整chunk大小、选择Embedding模型、维护向量索引。每个环节都可能出错。

用个比喻：grep就像在图书馆里用ISBN号查书，精确但要求你知道ISBN。RAG像让一个人帮你找「大概讲这个主题的书」，灵活但可能找错。当你有一个足够聪明的图书管理员（LLM）知道该查哪些ISBN时，前者的精确性反而更有价值。

3. 维护成本的差距是巨大的

grep零维护。代码变了？grep直接搜到最新内容。没有索引需要更新，没有向量需要重建，没有数据库需要管理。

RAG方案的维护成本不可小觑：代码每次变更都要重新chunk和embed，向量索引可能失效，Embedding模型升级需要重建所有向量，数据库本身也需要运维。对于一个每天在几十个不同项目之间切换的工具来说，这个维护成本是灾难性的。

Claude Code要做到「打开任何项目就能用」，零维护的grep是唯一可行的方案。如果每打开一个新项目都要先建索引，用户体验会大打折扣。

4. 上下文窗口变大了

RAG在上下文窗口很小（4K、8K）的时代是必需品，你只能往prompt里塞这么点内容，必须精挑细选。但Claude Code用的是100K甚至1M token的上下文窗口。

窗口大了，暴力搜索就变得可行了。多搜几个文件、多读几百行代码，反正放得下。不需要精密的检索来节省每一个token。

Grep工具的设计细节 Inside the Grep Tool

说完了「为什么不用RAG」，来看Anthropic到底怎么设计这个Grep工具的。源码里的GrepTool基于ripgrep（rg），不是GNU grep。这个选择本身就说明了问题。ripgrep比传统grep快一个数量级，原生支持.gitignore、Unicode、多线程，而且对大型代码库做了大量优化。

先看GrepTool的输入参数定义，信息量很大：

```
const inputSchema = lazySchema(() =>
  z.strictObject({
    pattern: z.string().describe(
      'The regular expression pattern to search for in file contents',
    ),
    path: z.string().optional(),
    glob: z.string().optional().describe(
      'Glob pattern to filter files (e.g. "*.js", "*.{ts,tsx}")'
    ),
    output_mode: z.enum([
      'content', 'files_with_matches', 'count'
    ]).optional(),
    '-B': semanticNumber(z.number().optional()),  // 前置上下文行
    '-A': semanticNumber(z.number().optional()),  // 后置上下文行
    '-C': semanticNumber(z.number().optional()),  // 对称上下文
    '-n': semanticBoolean(z.boolean().optional()), // 行号
    '-i': semanticBoolean(z.boolean().optional()), // 忽略大小写
    type: z.string().optional(),  // 文件类型过滤
    head_limit: semanticNumber(z.number().optional()),
    offset: semanticNumber(z.number().optional()),
    multiline: semanticBoolean(z.boolean().optional()),
  })
)
```

三种 `output_mode` 的设计很精妙：

`files_with_matches` （默认），只返回匹配的文件路径列表。这是最省token的模式，适合「先摸清哪些文件相关」的第一轮搜索。

`content`，返回匹配行及上下文。支持 `-A` / `-B` / `-C` 参数控制上下文行数，支持 `-n` 显示行号。适合「知道在哪个文件，想看具体代码」的精确搜索。

`count`，只返回每个文件的匹配次数。适合「这个函数被调用了多少次」之类的统计场景。

这三种模式本质上是一个**信息密度的梯度**。LLM可以先用 `files_with_matches` 低成本扫描，锁定目标后用 `content` 精确读取。这比一次性返回所有内容要高效得多。

还有一个不起眼但很重要的设计： `head_limit` 默认250行。

```
// Default cap on grep results when head_limit is unspecified.
// Unbounded content-mode greps can fill up to the 20KB persist
// threshold (~6-24K tokens/grep-heavy session).
// 250 is generous enough for exploratory searches while preventing
// context bloat. Pass head_limit=0 explicitly for unlimited.
const DEFAULT_HEAD_LIMIT = 250
```

这个注释本身就是一篇小论文。不限制结果数量的grep可能返回几万行，直接吃掉上下文窗口。250行是一个平衡点：足够做初步探索，又不至于浪费token。如果LLM真的需要更多结果，可以显式传 `head_limit=0` 或用

`offset` 翻页。

那为什么不直接让AI调用 `bash grep`？源码里的BashTool prompt明确要求**避免使用grep命令**，而是用Grep工具。原因有几个：GrepTool有权限校验（`checkReadPermissionForTool`），确保不会搜索不该搜的文件；它自动排除 `.git`、`.svn` 等版本控制目录的噪音；它限制每行最大500字符（`--max-columns 500`），防止base64或minified代码把结果撑爆；它还会把绝对路径转成相对路径来节省token。这些都是裸grep做不到的。

Glob工具：文件发现的第一步 GlobTool: File Discovery First

Grep负责「文件里搜什么」，Glob负责「先找到哪些文件」。这是搜索流程的第一步，经常被忽略但其实至关重要。

Glob工具的底层也是ripgrep，用的是 `rg --files --glob` 模式，让ripgrep列出匹配模式的文件而不是搜索文件内容。它继承了ripgrep的所有性能优势。

一个关键的设计细节：**结果按修改时间排序**。

```
const args = [
  '--files',
  '--glob',
  searchPattern,
  '--sort=modified',  // 按修改时间排序（oldest first）
  ...(noIgnore ? ['--no-ignore'] : []),
  ...(hidden ? ['--hidden'] : []),
]
```

Glob的结果按修改时间排序（ripgrep的 `--sort=modified` 是升序，最旧的在前），限制最多100个文件。LLM拿到完整的文件列表后，可以根据上下文自行判断哪些文件最相关。

而GrepTool的 `files_with_matches` 模式做了更进一步的优化，在应用层做了降序排序，**最近修改的文件排在最前面**：

```
// files_with_matches mode
const stats = await Promise.allSettled(
  results.map(_ => getFsImplementation().stat(_)),
)
const sortedMatches = results
  .map((_, i) => {
    const r = stats[i]!
    return [
      _,
      r.status === 'fulfilled' ? (r.value.mtimeMs ?? 0) : 0,
    ] as const
  })
  .sort((a, b) => b[1] - a[1])  // 最近修改的排最前
  .map(_ => _[0])
```

这不是一个技术上有多复杂的决策，但它体现了一种产品思维：**搜索结果的排序本身就是一种「智能」**。不需要向量数据库来判断相关性，修改时间就是一个极强的信号。Grep在 `files_with_matches` 模式下把最近修改的文件排在最前面，Glob按时间排序并限制最多100个文件，组合起来就能高效锁定目标。

Read工具：最被低估的搜索工具 The Read Tool: Underrated Search Weapon

Read工具表面上是「读文件」，但它其实是整个搜索系统的最后一环，也是能力最被低估的一环。

先看它的输入参数：

```
z.strictObject({
  file_path: z.string(),
  offset: semanticNumber(
    z.number().int().nonnegative().optional()
  ).describe('The line number to start reading from'),
  limit: semanticNumber(
    z.number().int().positive().optional()
  ).describe('The number of lines to read'),
  pages: z.string().optional().describe(
    'Page range for PDF files (e.g., "1-5", "3")'
  ),
})
```

`offset` 和 `limit` 参数意味着LLM不需要读整个文件。它可以先用Grep找到 `validateUser` 在第187行，然后用 `Read(file, offset=180, limit=30)` 只读周围的代码。对于几千行的大文件，这省掉了大量上下文。

但Read工具真正让人意外的是它的多模态能力。看prompt模板：

```
- This tool allows Claude Code to read images
  (eg PNG, JPG, etc). When reading an image file
  the contents are presented visually.
- This tool can read PDF files (.pdf). For large PDFs
  (more than 10 pages), you MUST provide the pages
  parameter to read specific page ranges.
- This tool can read Jupyter notebooks (.ipynb files)
  and returns all cells with their outputs, combining
  code, text, and visualizations.
```

它能读图片（利用LLM的多模态能力直接「看」截图和设计稿）、能分页读PDF、能解析Jupyter Notebook（包括输出结果和可视化）。这些能力组合起来，Read工具就不只是一个文件读取器，而是一个完整的**文档理解工具**。

还有一个藏在代码深处的优化： `FILE_UNCHANGED_STUB` 。如果LLM在同一个会话里重复读同一个文件，而文件没有变化，Read工具会返回一个简短的提示而不是完整内容：

```
export const FILE_UNCHANGED_STUB =
  'File unchanged since last read. The content from
   the earlier Read tool_result in this conversation
   is still current — refer to that instead of
   re-reading.'
```

这又是一个token节省策略。LLM经常在调试过程中反复读同一个文件，这个stub避免了重复加载。

把Grep + Glob + Read放在一起看：**Glob发现文件 → Grep定位位置 → Read精确读取**。三个工具各做一件事，组合起来就是一个完整的代码理解系统。没有索引，没有预处理，随开随用。

没有向量数据库的佐证 No Vector Database — The Evidence

前面说了Claude Code没用向量数据库。这不是猜测，是实证。

我在Claude Code的全部512,920行源码中搜索了以下关键词：

`embedding`：找到2处，一处是「embedding a file:// URL」（在URL中嵌入链接），一处是「string-embedding」（字符串嵌入bash命令）。都是英语单词的普通用法，和向量嵌入毫无关系。

`vector`：0处匹配。没有向量数据库，没有向量搜索，没有向量索引。

`semantic search`：0处匹配。

`pinecone / weaviate / chroma / faiss`：0处匹配。没有任何主流向量数据库的引用。

51万行代码。零向量搜索。

这不是「还没来得及做」或「下一版再加」。Claude Code已经是市场上最成功的AI编程工具之一，在这个状态下就已经证明了：对于代码搜索这个场景，你不需要向量数据库。

Anthropic作为一家AI公司，比谁都更了解Embedding和RAG的能力边界。他们选择不用，是一个**有意识的架构决策**，不是能力不足。整个搜索系统就三个工具、不到1500行代码，却支撑起了Claude Code对任意规模代码库的理解能力。

这给所有做AI应用的人一个提醒：**手里有锤子（RAG）的时候，不是所有问题都是钉子。**在LLM足够强的时代，最简单的工具链往往就是最有效的。

工作流解析 How the Search Workflow Works

Claude Code搜索代码时的典型流程是这样的：

1. **理解意图**
   LLM理解用户要找什么。比如「用户数据的验证逻辑」，它知道应该搜 `validate` 、 `user` 、 `schema` 等关键词。

2 **选择搜索策略**

先用Glob找相关文件（`**/*user*.ts`），再用Grep在候选文件里搜关键词（`validate`）。可能跑多轮不同的搜索来覆盖各种命名方式。

3 **读取和理解**

找到候选文件后，用Read工具读取完整内容。LLM理解代码的含义、调用关系、设计意图。

4 **迭代搜索**

如果第一轮没找到，LLM会根据已有发现调整搜索策略。比如发现验证逻辑在 `middleware` 目录而不是 `models` 目录，就转向那里继续搜。

搜索质量的瓶颈不在检索算法，而在搜索策略。知道该搜什么、在哪里搜、怎么从初步结果中提炼下一步方向，这些是LLM擅长的。grep给它精确的原始数据，LLM负责所有高层智能。分工明确。

这个决策能教我们什么 Lessons for AI Product Builders

> **核心建议**
>
> **「足够好」原则**：在设计AI应用时，不要默认每个环节都需要最先进的技术。先问一个问题：如果这个环节用最简单的方案，瓶颈在哪里？如果LLM自身的能力可以补偿简单方案的不足，那就用简单方案。复杂方案的维护成本可能比它节省的智能要贵得多。

几个可推广的判断标准：

如果LLM是消费者，搜索可以简单。RAG适合「搜索结果直接展示给人看」的场景。当搜索结果是给LLM看的中间步骤时，精确匹配 + LLM理解 > 模糊匹配 + 预排序。

零维护胜过高精度。如果你的工具需要在任意环境下即开即用（像Claude Code那样），任何需要预处理的方案都是障碍。

先做减法，再考虑加法。当你想加一个复杂组件时，先问：不加行不行？如果行，那就不加。Claude Code证明了，不用向量数据库也能做出最好用的AI编程工具。

当然，这个结论有边界条件。对于上百万个文档的企业知识库，纯grep可能不够。但对于代码搜索这个特定场景，项目通常在几万到几十万行之间，grep的速度足够，LLM的理解力足够，简单方案就是最优方案。

09 多Agent架构：像公司一样运转

Multi-Agent Architecture — Run Like a Company

Claude Code的多Agent系统可能是整个源码中最复杂的部分。它不只是fork几个进程并行跑，而是实现了一套完整的组织管理架构。

组织结构 Organization Structure

源码的 `src/utils/swarm/` 目录下实现了一个多Agent协作框架。核心组织模型是：

角色	职责	权限
Team	一组Agent的容器，负责资源和目标管理	最高
Leader	分配任务、审批权限请求、合并结果	高
Teammate	执行具体任务、报告进度	受限

这和一个真实公司的结构几乎一模一样。Leader就是tech lead，Teammate就是开发者，Team就是项目组。

三种执行方式 Three Execution Modes

Agent可以通过三种方式并行运行：

同进程隔离：Agent在同一个Node.js进程内运行，通过 `AsyncLocalStorage` 实现上下文隔离。每个Agent有独立的AbortController（Teammate不会因Leader被中断而中断）、独立的状态存储、共享的API客户端和MCP连接。最轻量，零spawn开销，适合简单的并行搜索任务。

tmux窗口：每个Agent在独立的tmux窗格中运行。可以看到各个Agent的实时输出。适合需要监控进度的场景。

iTerm2分割窗格：利用iTerm2的分屏能力，每个Agent在独立的窗格中运行。视觉效果最直观，但只在macOS + iTerm2环境下可用。

为什么提供三种方式？因为不同的使用场景对可视性的需求不同。做简单调研不需要看每个Agent的输出，但做复杂的多模块开发时，能同时看到所有Agent的进度很重要。

Git Worktree隔离 Isolation via Git Worktree

这是多Agent架构中最关键的设计决策。

当多个Agent同时修改代码时，最大的风险是**冲突**。Agent A在改文件X，Agent B也在改文件X，两个人互相覆盖。

Claude Code的解决方案是 `git worktree` 。每个Agent在自己的独立工作树中工作，有独立的文件系统副本。修改完成后，通过git的merge机制合并结果。

Git worktree不是完整的repo克隆。它共享同一个 `.git` 目录，只是创建了一个新的工作目录和分支。创建速度快，磁盘开销小，适合短生命周期的Agent任务。

如果Agent没有做任何更改，worktree会自动清理。如果做了更改，worktree的路径和分支名会返回给调用者，供后续合并。

邮箱通信 Mailbox Communication

Agent之间怎么通信？不是通过API调用，不是通过消息队列。是通过**邮箱文件**。

每个Agent有一个邮箱文件。其他Agent想给它发消息，就往这个文件里写。Agent定期检查自己的邮箱，读取新消息。

这又是一个「最简单方案」的例子。文件系统是最可靠的通信基础设施，不需要启动额外的服务，不会有连接超时，重启后消息还在。缺点是延迟高（定期轮询），但对于Agent级别的协作（分钟级的任务），几秒的延迟完全可以接受。

权限冒泡 Permission Bubbling

当Teammate遇到需要确认的操作（比如删除文件），权限请求不会直接弹给用户。它会**冒泡**给Leader。Leader决定是否批准。

为什么不直接问用户？因为如果5个Agent同时工作，每个都弹权限确认窗口，用户会被淹没。Leader作为中间层，可以根据任务上下文批量处理权限请求，减少对用户的打扰。

这和真实公司的审批流程一样。开发者不会每个操作都去找CEO确认，而是让tech lead代为判断。

Coordinator Mode The Coordinator Pattern

除了基本的Team模式，Claude Code还实现了一个更高级的**Coordinator Mode**。四个阶段：

1. **Research（调研）**
   多个worker并行调查代码库，收集信息。比如一个worker分析前端代码，另一个分析后端API。

2. **Synthesis（综合）**
   Coordinator收集所有worker的发现，综合分析，生成规格说明。这一步很重要，它把分散的信息变成统一的行动计划。

3 **Implementation（实现）**

Worker按照规格说明做精准修改。每个worker只负责自己的部分，按照统一的规格来。

4 **Verification（验证）**

Worker验证修改结果。可能包括运行测试、检查兼容性、确认没有引入新问题。

Coordinator的系统提示分析 Dissecting the Coordinator Prompt

Coordinator Mode的系统提示本身就是一份极有价值的文档。它不只是告诉模型「你是什么」，而是建立了一整套行为规范。源码中定义的角色描述开门见山：

```
You are a **coordinator**. Your job is to:
- Help the user achieve their goal
- Direct workers to research, implement and verify code changes
- Synthesize results and communicate with the user
- Answer questions directly when possible —
  don't delegate work that you can handle without tools
```

最后一条很关键：Coordinator不是一个纯粹的调度器，它应该**能自己回答的就自己回答**。这避免了过度委派的问题。如果用户问「这个函数是做什么的」，Coordinator不应该spawn一个worker去搜索，而是直接回答。

系统提示中还有一条容易被忽略但极其重要的规则：

```
Every message you send is to the user. Worker results
and system notifications are internal signals, not
conversation partners — never thank or acknowledge them.
Summarize new information for the user as it arrives.
```

Workers的返回结果以 `<task-notification>` 格式插入到对话中，看起来像用户消息，但它们不是。Coordinator必须区分两者：来自用户的真正消息需要回应，来自Worker的通知只是内部信号，应该被消化、综合、然后以自然语言向用户汇报。

还有一个更深层的设计约束：**Workers看不到Coordinator和用户之间的对话**。每个发给Worker的prompt必须是自包含的，包含所有Worker需要的上下文、文件路径、行号和具体要求。这和很多人想象中的多Agent系统不同。很多人以为Agent之间共享同一个对话上下文，但实际上每个Worker都是一个独立的世界，它只看到Coordinator给它的那一段指令。

这个设计选择经过深思熟虑。共享上下文听起来方便，但会带来两个问题：上下文膨胀（每个Worker都要加载完整对话历史）和信息干扰（前一个Worker的探索噪声会影响下一个Worker的判断）。自包含prompt更干净、更可控。

权限同步的文件锁设计 Permission Sync with File Locks

权限冒泡的概念前面已经说了，但具体实现值得再展开。源码中的 `permissionSync.ts` 实现了一套基于文件系统的权限请求队列。

目录结构如下：

```
~/.claude/teams/{teamName}/permissions/
├─── pending/           # 待处理的权限请求
│    ├─── .lock          # 文件级锁
│    ├─── perm-1712345-abc.json
│    └─── perm-1712346-def.json
└─── resolved/          # 已处理的权限请求
     └─── perm-1712345-abc.json
```

流程是这样的：Worker遇到需要权限的操作时，创建一个 `SwarmPermissionRequest` 对象（包含toolName、input、description等完整信息），写入 `pending/` 目录。Leader定期扫描 `pending/` 目录，发现新请求后展示给用户，用户做出决定，Leader将请求移到 `resolved/` 目录。Worker轮询 `resolved/` 目录，读取结果，继续执行。

源码中同时存在一套更新的**邮箱式权限通信**（ `sendPermissionRequestViaMailbox` ），通过前面提到的邮箱文件来传递权限请求和响应，正在逐步替代文件目录方式。新方式复用了已有的邮箱基础设施，避免了单独维护pending/resolved目录的开销。

一个技术细节：当多个Worker同时发送权限请求时，可能出现并发写入冲突。源码的解决方案是在 `pending/` 目录下维护一个 `.lock` 文件，通过 `lockfile.lock()` 实现互斥写入。获取锁、写入请求文件、释放锁，经典的文件级锁模式。

已处理的请求不会无限积累。 `cleanupOldResolutions()` 函数会清理 `resolved/` 目录中超过1小时的过期文件（ `maxAgeMs = 3600000` ）。这是一个典型的工程权衡：保留太久浪费磁盘，删除太快可能导致Worker还没读到结果。1小时是一个合理的中间值。对于Agent级别的任务，1小时内Worker一定已经读取了结果。

Team的生命周期管理 Team Lifecycle Management

Team不是创建了就行，它的销毁同样需要精心设计。源码中 `cleanupSessionTeams()` 定义了一个严格的清理顺序：

1. **Kill Panes**

   先杀死所有pane-backed的teammate进程。调用各backend（tmux/iTerm2）的 `killPane()` 方法，确保没有孤立进程继续运行。

2. **Destroy Worktrees**

   销毁所有成员的git worktree。先读取team文件获取worktree路径列表，然后逐一调用 `git worktree remove --force` 。如果git命令失败，fallback到直接 `rm -rf` 。

3 **清理Team目录**

删除 `~/.claude/teams/{team-name}/` 目录及其下所有内容（包括权限请求文件、团队配置等）。

4 **清理Tasks目录**

删除 `~/.claude/tasks/{taskListId}/` 目录，并触发 `notifyTasksUpdated()` 通知UI更新。

为什么顺序很重要？源码的注释说得很清楚：

```
Kill panes first — on SIGINT the teammate processes
are still running; deleting directories alone would
orphan them in open tmux/iTerm2 panes.
```

如果先删目录再杀进程，Agent进程还在运行，但它依赖的工作目录已经被删了。这些孤儿进程会在tmux窗格里不断报错，又没有人清理它们。先杀进程再删目录，确保不会出现这种状态。

源码还区分了两种清理场景：**正常删除**（通过TeamDeleteTool，此时teammate已经优雅退出）和**异常清理**（Leader进程被SIGINT/SIGTERM杀死）。正常删除不需要kill panes，进程已经自己退出了。异常清理才需要强制kill。这个区分避免了不必要的kill操作，也避免了kill一个已经不存在的进程时的错误。

Team的注册和注销通过 `registerTeamForSessionCleanup()` 和 `unregisterTeamForSessionCleanup()` 管理。创建Team时注册，正常删除时注销（防止重复清理），Session结束时批量清理所有注册但未注销的Team。这是一个防御性编程模式：即使TeamDelete没被调用（比如用户直接关闭了终端），Team资源也不会泄漏。

源码中还有几个值得深入的设计细节：

反偷懒规则

Coordinator Mode的prompt中包含了严格的反偷懒指令。源码明确禁止以下模式：

```
Never write "based on your findings" or "based on the
research." These phrases delegate understanding to the
worker instead of doing it yourself.
```

Coordinator在收到Worker的研究报告后，**必须自己理解内容**，然后写一个包含具体文件路径、行号和修改方案的指令。不能把理解的工作甩给下一个Worker。

Verification阶段同样有严格要求：「Verification means proving the code works, not confirming it exists」。验证者必须实际运行测试、检查类型错误、测试边界情况。不能只是看一眼代码说「看起来没问题」。

Continue vs Spawn的决策矩阵

Coordinator Mode的prompt中详细定义了什么时候继续用同一个Worker（SendMessage），什么时候spawn一个新的：

场景	选择	原因
Research探索了恰好要编辑的文件	Continue	Worker已有文件在context中
Research很广但实现很窄	Spawn fresh	避免探索噪声干扰精准实现
第一次实现完全错误	Spawn fresh	错误方法的context会锚定重试
验证另一个Worker的代码	Spawn fresh	验证者应以新鲜眼光看代码

.worktreeinclude支持

Worktree隔离有一个边界情况：某些文件被gitignore了（比如 `.env` 配置文件、本地密钥），但Agent工作时需要用到。源码实现了 `.worktreeinclude` 文件来解决这个问题，列出需要从主repo复制到worktree的gitignored文件。

这个系统能教我们什么 Lessons for AI Product Builders

> **核心建议**
>
> **多Agent系统的难点不在技术实现，而在组织设计。**任务怎么拆分、信息怎么流转、冲突怎么解决、结果怎么合并、质量怎么保证——这些问题和管理真人团队完全一样。

隔离优于共享。Git worktree让每个Agent在独立的文件系统中工作，从根本上消除了并发修改的冲突问题。代价是需要额外的合并步骤，但这比处理运行时冲突简单得多。

通信用最简单的方案。邮箱文件比消息队列简单100倍，对于Agent级别的协作完全够用。不要在通信基础设施上过度投入。

需要中间管理层。Leader角色不是多余的。它处理权限冒泡、监督任务进度、协调结果合并。没有这一层，5个Agent直接与用户交互会混乱不堪。

10 Feature Flags里的未来

The Future in Feature Flags

源码中有44个feature flags，控制着一批尚未发布的功能。这些flags就像一扇窗，让我们窥见Claude Code团队的产品路线图。

GrowthBook：Feature Flags的基础设施 GrowthBook — The Feature Flag Infrastructure

具体看flags之前，先看Anthropic用什么来管理它们。

答案是**GrowthBook**，一个开源的feature flag和A/B测试平台。Claude Code的源码中有一个完整的GrowthBook集成层（`src/services/analytics/growthbook.ts`），超过400行代码专门处理feature flag的获取、缓存和实验追踪。

几个关键设计：

- **远程评估模式**（remoteEval）：flag的值不是在客户端计算的，而是由GrowthBook服务端根据用户属性预先计算好、直接下发。客户端拿到的是结果，不是规则
- **磁盘缓存**：flag值会被写入本地的 `~/.claude.json` 配置文件。这意味着即使GrowthBook服务暂时不可用，Claude Code也能用上次缓存的值正常运行。代码里有一段注释：「Without this running on refresh, remoteEvalFeatureValues freezes at its init-time snapshot」
- **多层覆写**：环境变量覆写 > 本地配置覆写 > 远程值。内部员工可以通过 `CLAUDE_INTERNAL_FC_OVERRIDES` 环境变量强制指定任意flag的值，方便调试和测试
- **曝光追踪**（exposure logging）：每个feature flag被访问时都会记录一次「曝光事件」，用于后续的A/B测试分析。同一session内自动去重，避免热路径（如渲染循环中被反复调用的flag）产生重复数据

所有flag都以 `tengu_` 为前缀。Tengu（天狗）是Claude Code的内部项目代号。

用户属性的丰富程度值得一看：

```
type GrowthBookUserAttributes = {
  id: string
  sessionId: string
  deviceID: string
  platform: 'win32' | 'darwin' | 'linux'
  apiBaseUrlHost?: string
  organizationUUID?: string
  accountUUID?: string
  userType?: string           // 'ant' 或 undefined
  subscriptionType?: string
  rateLimitTier?: string
  firstTokenTime?: number
  email?: string
  appVersion?: string
  github?: GitHubActionsMetadata
}
```

这些属性让GrowthBook可以做非常精细的定向：按组织、按订阅类型、按平台、甚至按GitHub CI环境来控制功能的开放范围。一个功能可以先在Mac上的Pro用户中灰度，确认没问题再推给所有人。

为什么一个AI编程工具需要这么重的feature flag基础设施？因为AI产品的行为比传统软件复杂得多。传统软件的行为由代码决定，代码不变行为就不变。AI产品的行为还依赖模型版本、prompt措辞、上下文管理策略，任何一个因素变化都可能导致用户体验的显著波动。Feature flags让团队可以**在不部署代码的情况下调整这些因素**，出问题时秒级回滚。

Dead Code Elimination：编译时的Feature Gate Dead Code Elimination — Compile-Time Feature Gate

GrowthBook处理的是运行时的feature flags。还有另一层gate发生在更早的阶段：编译时。

`process.env.USER_TYPE === 'ant'` 是一个编译时常量。构建外部版本时，bundler会把它常量折叠为 `false` ，然后Dead Code Elimination（DCE）会把所有 `false` 分支的代码完全删除。

源码中遍布这种模式：

```
// 外部版本构建后，这行变成 false ? require(...) : null
// bundler直接消除整个require，连模块都不会被打包
const assistantModule = true
  ? require('./assistant/index.js')
  : null

// 同理，!true 在外部版本变成 !false 即 true
// 内部版本变成 !true 即 false，函数直接return
if (!true) return []
```

这个模式在整个代码库中出现了几十次。KAIROS的assistant模块、BUDDY的宠物系统、coordinator协调器、proactive主动模式……所有内部功能都通过这种方式在编译时被gate掉。

效果是：**外部用户拿到的包里，物理上不存在任何内部代码。**不是「代码在但不执行」，是「代码压根不在包里」。更小的包体积、更快的启动速度、更小的攻击面。外部用户即使反编译Claude Code的二进制文件，也看不到这些内部功能的任何痕迹。

这次源码泄露之所以有意思，恰恰是因为泄露的是编译前的源码，而不是编译后的产物。

KAIROS：始终在线的AI助手 KAIROS — The Always-On Assistant

KAIROS是源码中出现频率最高的feature flag，在代码中被引用了近150次。它代表Claude Code的终极愿景：一个**不等用户输入就主动观察、记录、行动**的AI助手。

核心特性：

- **追加式日志**（append-only daily logs）：记录观察和决策，形成AI自己的工作日志
- **15秒阻塞预算**：Bash命令在主Agent中运行超过15秒后自动转入后台，避免阻塞用户交互
- **专属工具**：PushNotification（给用户发通知）、SendUserFile（给用户发文件）、SubscribePR（订阅PR更新）
- **自动记忆整理**（autoDream）：24小时时间门 + 5次会话最低要求 + 整理锁三重gate，避免过于频繁的自动行为。注意autoDream是独立于KAIROS的子系统，KAIROS激活时autoDream反而会让位给KAIROS自己的记忆整理
- **GitHub webhook集成**：监听仓库事件，自动响应
- **Cron定时任务**：定时执行预设的工作

通过 `claude assistant [sessionId]` 启动。它有自己完全不同的system prompt，不再是「帮用户做编程」，而是「你是一个自主Agent，用工具做有用的事」。

KAIROS代表的方向很清晰：Claude Code不满足于「你问它答」的工具定位，正在走向一个**能主动思考、持续运行的AI伙伴**。

BUDDY：电子宠物系统 BUDDY — The Digital Pet

这大概是最出人意料的发现。

`src/buddy/` 目录下藏着一个完整的虚拟宠物系统。Tamagotchi风格。每个用户会拥有一只用确定性算法生成的伙伴精灵。

技术实现上很有趣：

- **18种物种**：duck, goose, blob, cat, dragon, octopus, owl, penguin, turtle, snail, ghost, axolotl, capybara, cactus, robot, rabbit, mushroom, chonk。稀有度从Common（60%权重）到Legendary（1%权重）分5级

- **6种眼睛样式**：· ✦ × ◉ @ ° ，细节到每只宠物的眼神都不一样
- **8种帽子**：crown, tophat, propeller, halo, wizard, beanie, tinyduck（头上顶个小鸭子）。注意只有uncommon以上的宠物才有帽子，common级别永远是 `none`
- **1%闪光变体概率**
- **确定性随机化**：使用Mulberry32 PRNG算法，种子 = 用户ID的hash + salt `'friend-2026-401'` 。每个用户永远分到同一只宠物，但不同用户分到不同的。宠物一旦生成就被缓存，不会重复计算
- **5个程序化属性**：DEBUGGING, PATIENCE, CHAOS, WISDOM, SNARK——每个属性值从同一个PRNG链中抽取，且有一个「峰值属性」和一个「垃圾属性」，制造出每只宠物的性格差异
- **稀有度影响属性下限**：Common的属性下限是5，Legendary的下限是50。越稀有的宠物基础属性越高
- **首次孵化时AI生成个性**：宠物有「骨骼」（Bones，确定性部分）和「灵魂」（Soul，LLM生成部分）。骨骼每次从hash重新计算，灵魂只在第一次孵化时生成并永久存储

每只宠物还有完整的ASCII艺术精灵图，3帧动画用于闲置时的微动效果。比如鸭子：

```
    __
  <(· )___
   (  ._>
    `--´
```

代码注释里有一句让人会心一笑的话：「Mulberry32, good enough for picking ducks」（用来挑鸭子够用了）。

一个防作弊设计值得一提：骨骼数据（包括稀有度和物种）永远不持久化存储，每次都从用户ID重新计算。存储在config里的只有灵魂数据（名字和性格描述）。用户编辑配置文件也无法把自己的Common鸭子改成Legendary龙。代码注释说得很直白：「users can't edit their way to a legendary」。

还有个有趣的细节：物种名在源码中用 `String.fromCharCode` 的十六进制编码来定义，而不是直接写字符串字面量。原因是某个物种名跟模型代号冲突，构建检查会扫描输出文件中的敏感字符串，用charCode构造可以绕过这个检查。

BUDDY说明了一件事：**即使是最严肃的技术公司，也知道产品需要趣味性。**一个让你每天都想打开的AI工具，光功能强大不够，还需要一点人情味。

ULTRAPLAN：远程规划会话 ULTRAPLAN — Remote Planning

将复杂的规划任务卸载到远程Cloud Container Runtime (CCR)，运行Opus 4.6。

核心参数：

- 最长**30分钟思考时间**
- 浏览器端审批UI，实时监控进度
- 每3秒轮询结果
- 特殊哨兵 `__ULTRAPLAN_TELEPORT_LOCAL__` 将结果传回终端

这解决了一个实际问题：有些规划任务太复杂，需要模型长时间思考。在本地终端中等30分钟体验很差，但在后台运行、用浏览器监控进度、完成后自动把结果传回终端，体验就完全不同了。

其他值得关注的Flags Other Notable Flags

Flag	功能	意义
`tengu_penguins_off`	Penguin Mode（快速执行模式）	在速度和质量之间的动态切换
`tengu_amber_flint`	Swarm/Team能力	多Agent协作的开关
`tengu_scratch`	共享暂存目录	Agent间的文件共享
`tengu_hive_evidence`	Verification Agent	完成后的对抗性验证

从Feature Flags看产品演进策略 Feature Flags as Product Strategy

44个flags本身就说明了Claude Code的产品演进方式：

渐进式发布。功能先在内部（ant用户）启用，验证后逐步开放给外部用户。出问题可以秒级关闭。这让团队可以大胆实验，因为失败的成本是关一个开关而不是回滚代码。

A/B测试驱动。源码中大量注释标注了「un-gate once validated on external via A/B」，在外部用户上A/B验证通过后才全量开放。每个功能的发布都是数据驱动的决策。

模型版本绑定。很多flags跟特定模型版本绑定（如「Capybara v8 thoroughness counterweight」）。模型升级时，相关的flags可能需要重新评估。

> **核心建议**
>
> **Feature flags是AI产品的必备基础设施。**AI产品的行为高度依赖模型版本、prompt策略和用户反馈。Feature flags让你可以在不部署代码的情况下调整产品行为，这对于快速迭代的AI产品至关重要。GrowthBook + DCE的组合给了Claude Code团队两层控制：运行时的灰度开关和编译时的硬隔离。前者灵活，后者安全。

11 两个Claude Code

Internal vs External — Two Products in One Codebase

源码中大量 `process.env.USER_TYPE === 'ant'` 判断。Anthropic内部员工用的Claude Code和你用的版本有不少区别。这些差异揭示了Anthropic认为AI应该怎样工作的理想状态。

编译时分化：同一套代码，两个产品 Compile-Time Divergence — One Codebase, Two Products

`USER_TYPE` 是一个构建时常量（build-time `--define`）。构建内部版时它是 `'ant'`，构建外部版时它是另一个值。关键在于，**它不是运行时的环境变量，而是编译时的常量。**

这意味着bundler在打包时会做常量折叠（constant folding）。所有 `process.env.USER_TYPE === 'ant'` 在外部版构建中被替换为 `false`，然后Dead Code Elimination把整个 `false` 分支连同它require的模块一起删除。

源码中的注释说得很清楚：

```
// All code paths are gated on process.env.USER_TYPE === 'ant'.
// Since USER_TYPE is a build-time --define, the bundler
// constant-folds these checks and dead-code-eliminates the
// ant-only branches from external builds. In external builds
// every function in this file reduces to a trivial return.
```

以undercover.ts为例，外部版构建后，`isUndercover()` 变成 `return false`，`getUndercoverInstructions()` 变成 `return ''`。整个文件的实质性代码全部消失。同样的事情发生在KAIROS的assistant模块、BUDDY的宠物系统、coordinator协调器等所有内部功能上。

这给了Claude Code团队一个独特的能力：**在同一个代码库里维护两个产品，但外部版从物理上不包含任何内部代码。**不是「代码在但被if跳过了」，是真的不存在。外部用户反编译二进制文件也看不到这些代码。

核心差异一览 Key Differences

维度	外部版	内部版
代码注释	无特殊要求	默认不写注释，只在WHY不明显时才加
诚实性	无特殊要求	不粉饰测试结果、不暗示未运行的验证通过了
输出风格	简洁直接（4行）	像给人写信（20+行详细指导）
完成验证	无特殊要求	报告完成前必须实际验证
输出长度	无硬性限制	工具调用间≤25词，最终回复≤100词
Bug反馈	指向GitHub Issues	指向/issue和/share命令，可发送到内部Slack
Verification Agent	未启用	A/B测试中，完成后自动对抗性验证

数字化的输出约束 Quantitative Output Constraints

差异表中「输出长度」那行值得展开。内部版的prompt里有这样一段：

```
// Numeric length anchors — research shows ~1.2% output token
// reduction vs qualitative "be concise". Ant-only to measure
// quality impact first.
...(process.env.USER_TYPE === 'ant'
  ? ['Length limits: keep text between tool calls to ≤25 words.
      Keep final responses to ≤100 words unless the task
      requires more detail.']
  : [])
```

注意注释：「research shows ~1.2% output token reduction vs qualitative 'be concise'」。

这说明Anthropic做过对照实验。对照组用的是定性指令（「be concise」「keep it short」），实验组用的是定量指令（≤25词、≤100词）。**结果是定量约束比定性约束多减少了约1.2%的输出token。**

1.2%听起来不多，但放在Anthropic的规模上就不一样了。Claude Code每天处理的API调用量极大，1.2%的token减少意味着可观的成本节省和更快的响应速度。而且这个优化是零成本的，不需要改模型，不需要改代码，只需要把「be concise」换成「≤25 words」。

外部版用的是定性指令：「Your responses should be short and concise.」内部版用的是定量指令加上完全不同的输出哲学。来看这段只在内部版出现的prompt：

```
When sending user-facing text, you're writing for a person,
not logging to a console. Assume users can't see most tool
calls or thinking - only your text output.

Write user-facing text in flowing prose while eschewing
fragments, excessive em dashes, symbols and notation...

What's most important is the reader understanding your output
without mental overhead or follow-ups, not how terse you are.
```

外部版说「简洁」，内部版说「像给人写信」。外部版追求「少说」，内部版追求「说清楚」。两种截然不同的产品理念，Anthropic显然认为后者更好。那为什么不给外部用户也用？大概是因为这个prompt更长、消耗更多token，需要先在内部验证效果再推广。

Undercover Mode Going Undercover

当 `USER_TYPE === 'ant'` 且在公共仓库工作时，自动激活Undercover Mode。这个模式在社区引发了不小争议。

仓库分类的逻辑在 `commitAttribution.ts` 里。代码维护了一个允许列表 `INTERNAL_MODEL_REPOS` ，列出了所有确认为私有的内部仓库（claude-cli-internal、anthropic、apps、casino等约20个仓库）。判断逻辑是：

```
// 'internal' = remote matches allowlist (private repo)
// 'external' = has a remote, not on allowlist (public repo)
// 'none'     = no remote URL (not a git repo)
let repoClassCache: 'internal' | 'external' | 'none' | null
```

分类结果被缓存，每个进程只检查一次。然后Undercover Mode的激活逻辑是：

```
export function isUndercover(): boolean {
  if (process.env.USER_TYPE === 'ant') {
    if (isEnvTruthy(process.env.CLAUDE_CODE_UNDERCOVER))
      return true
    // Auto: active unless we've positively confirmed we're
    // in an allowlisted internal repo. 'external', 'none',
    // and null all resolve to ON.
    return getRepoClassCached() !== 'internal'
  }
  return false
}
```

注意默认值的设计：只有当仓库被**确认**是内部仓库时才关闭Undercover Mode。其他所有情况，外部仓库、没有remote、还没检查完，都默认开启。源码注释说得很明确：「There is NO force-OFF. This guards against model codename leaks — if we're not confident we're in an internal repo, we stay undercover.」

激活后的行为：

- 从system prompt中去掉所有模型名称
- 禁止泄露内部模型代号（动物名：Capybara、Tengu等）
- 禁止泄露未发布模型版本（如opus-4-7）
- 禁止暴露内部工具和Slack引用
- 禁止显示任何表明commit来自AI的痕迹
- commit消息要写得「像人类开发者写的」——只描述代码改了什么，不提AI

争议点在于最后两条。社区认为这是明确指示AI「假装是人类贡献者」，由此引发了关于AI参与开源的透明度讨论。

首次在公共仓库中自动激活时，代码还会弹出一个一次性的解释对话框（`shouldShowUndercoverAutoNotice`），告诉员工当前处于Undercover Mode。看到一次后就不再提示。

Anti-Distillation：反蒸馏防御 Anti-Distillation Defense

这是另一个引发社区讨论的发现。

Anthropic在API请求中注入**伪造的工具定义**。目的是：如果竞争对手试图通过截获API请求来训练自己的模型（蒸馏），这些假数据会污染训练集。

这个策略的伦理争议很大。支持者认为这是合法的知识产权保护，反对者认为这跟注水差不多，而且可能影响正常用户的体验。

源码还暴露了另一种反蒸馏手段：**思维摘要隐藏**（redacted thinking）。通过 `redact-thinking-2026-02-12` beta header，模型的推理过程在API响应中被编辑或隐藏。即使竞争对手截获了API响应，也无法获取模型的思维链。而思维链恰恰是蒸馏最需要的数据。

还有一种更先进的「连接文本汇总」（Slate Prism POC）：由服务端对输出做汇总并签名，确保下游消费者无法获取原始输出。不过从源码看这个功能还在POC阶段。

情绪检测 Sentiment Detection

源码中有一个出人意料的模块：用**正则表达式**检测用户消息中的负面情绪。不是用LLM，是用正则。

社区的第一反应是嘲笑：一个LLM公司为什么用正则做情绪检测？但答案很务实：

- 正则比最快的LLM调用快约**10,000倍**
- 成本几乎为零
- 结果确定性强，适合做指标对比
- 不需要额外的API调用

检测到负面情绪后只是记录日志，不会改变AI的行为。这是一个**观测指标**，用于团队了解用户体验，不是一个功能。

这又回到了第8章的主题：**不是每个问题都需要最先进的技术。**正则做情绪检测精度不高，但对于「大致了解用户情绪趋势」这个目标够用了。

Verification Agent Adversarial Verification

内部版正在A/B测试的一个功能：完成复杂实现后，自动启动一个独立的agent做**对抗性验证**。

源码中的描述很严格：

```
Independent adversarial verification must happen before you
report completion. Your own checks, caveats, and a fork's
self-checks do NOT substitute — only the verifier assigns
a verdict.
```

关键设计点：

- 验证Agent和执行Agent完全独立
- 执行Agent不能给自己的工作打分
- 验证结果有三级：PASS、FAIL、PARTIAL
- FAIL时必须修复后重新验证，直到PASS
- 验证Agent的命令输出要和执行Agent的实际再跑一遍对比

这本质上是**代码审查**的AI版本。一个人写代码，另一个人审查。只不过两个都是AI。

这些差异告诉我们什么 What the Differences Reveal

> **核心建议**
>
> **内部版暴露了Anthropic的产品理想。**「默认不写注释」说明他们认为好代码不需要注释。「报告完成前必须验证」说明他们知道AI爱偷懒。「写给人读不是往console打日志」说明他们认为AI的输出应该有温度。「≤25词」而不是「be concise」说明他们用数据驱动而非直觉驱动来优化产品。

如果你在设计自己的AI产品，这些内部版的规则是很好的参考标准：

要求AI诚实。在指令中明确说「不确定的地方说不确定」「测试失败了就说失败」。不给AI模糊过关的空间。

要求AI验证。别让AI说「应该可以了」，要求它实际运行验证并展示验证结果。

把AI的输出当给人看的文字。不是日志，不是调试信息，而是有结构、有重点、能让忙碌的人快速理解的文字。

用定量约束替代定性约束。「≤100词」比「keep it concise」有效。具体的数字给模型一个明确的锚点，而模糊的形容词留下太多解释空间。

编译时隔离比运行时判断更安全。如果你的产品也分内部版和外部版，在构建阶段就把内部代码剥离，比在运行时用if判断可靠得多。前者保证外部版物理上不包含敏感代码，后者总有绕过的可能。

12 Harness Engineering方法论

The Harness Engineering Playbook

前11章拆解了Claude Code六个核心系统的设计决策。最后一章把散落的洞察提炼成可复用的原则。不是教条，是经验法则。

十条设计原则 Ten Design Principles

原则一：简单组件 + 智能大脑 > 全链路复杂

Claude Code的搜索用grep不用RAG，记忆用纯文本文件不用数据库，通信用邮箱文件不用消息队列。每个独立组件都很简单，但组合起来依靠LLM的理解力，整体表现反而超过每个环节都很复杂的方案。

当你有一个足够强的LLM作为系统的核心时，外围组件应该尽可能简单。复杂度集中在一个点（模型），比分散在十个点（每个组件都有自己的复杂度）更容易管理。

原则二：安全是信任的基础，不是功能的限制

权限系统不是限制AI能做什么，而是建立用户对AI的信任。因为你知道有四层安全审查、有熔断机制、有受保护文件列表，所以才敢打开Auto模式让它自由操作。

设计AI产品时，安全系统的投入应该和核心功能一样多。用户愿意给AI多大的权限，取决于他们对安全边界有多少信心。

原则三：记住偏好，忘记事实

记忆系统只记用户的偏好和判断，不记代码和事实。偏好稳定，但事实会变。记住「用户喜欢TypeScript」比记住「函数X在文件Y第30行」可靠得多。

推广到更一般的情况：**持久化应该存「变化慢的东西」**。模型的能力（搜索、理解）可以实时获取「变化快的东西」。

原则四：压缩是有损的，要设计损失什么

上下文压缩不是随便总结一下，而是精心设计的9段式结构化提取。哪些信息必须保留（用户的每一句话）、哪些可以浓缩（工具返回的详细数据）、按什么格式组织（9个固定段落），都有明确规范。

如果你的AI应用需要处理长对话，不要让模型自由发挥「帮我总结一下」。给它一个结构化的压缩模板，明确保留什么、丢弃什么。

原则五：工具是能力的边界

Claude Code的所有操作都通过工具间接执行。模型不能直接操作文件系统，必须调用Read、Write、Bash等工具。这层间接性不是性能开销，是架构优势。

因为所有操作都经过工具这一层，你可以在这层做权限检查、日志记录、安全审查、使用量统计。工具定义了AI能做什么，也定义了AI不能做什么。**控制AI的最好方式不是写更长的prompt，而是设计更好的工具。**

原则六：多Agent协作需要组织设计

Claude Code的多Agent不是简单地fork几个进程。它有Team/Leader/Teammate的层级、独立的git worktree隔离、异步的邮箱通信、权限冒泡机制。

多Agent系统的难点不在技术实现，而在组织设计。任务怎么拆分、信息怎么流转、冲突怎么解决、结果怎么合并，这些问题和管理真人团队完全一样。技术实现反而是简单的部分。

原则七：Feature Flags管理产品演进

44个feature flags，控制着KAIROS、BUDDY、ULTRAPLAN等未发布功能。每个功能都可以独立开关，可以按用户分群灰度发布，出问题可以秒级回退。

对于快速迭代的AI产品，feature flags是必备基础设施。它让团队可以大胆实验，因为失败的成本是关一个开关而不是回滚代码。

原则八：内部和外部应该是同一个产品

Anthropic自己用Claude Code，而且内部版本比外部版本更严格。这不是特权，是产品自信的来源。自己都不用自己的AI产品，怎么知道它好不好？

内部版本的那些严格要求（不粉饰测试结果、不写不必要的注释、完成前必须验证）是Anthropic认为AI应该怎样工作的理想状态。这些要求也可以成为你设计AI产品时的参考基准。

原则九：缓存策略直接影响成本

System prompt的静态/动态分界线、Deferred Tools的延迟加载、prompt缓存的跨用户复用。这些看起来是技术细节，实际上直接影响每次API调用的token消耗和成本。

设计AI产品时，从第一天就要考虑prompt缓存策略。把不变的部分放前面可以缓存复用，把变化的部分放后面按需加载。这可能是整个产品最大的成本优化杠杆。

原则十：60/40法则

Claude Code好用，60%靠模型能力，40%靠harness工程。

这个比例说明两件事。**模型仍然是最重要的**，选对底层模型比什么都重要。但**harness的40%决定了产品体验的上限**，同样的模型套上不同的harness就是完全不同的产品。

不要因为「反正模型最重要」就忽略harness工程，也不要因为「harness可以弥补」就用弱模型。两者缺一不可。

一张决策检查清单 A Decision Checklist

如果你正在设计一个AI产品或AI Agent，以下问题值得在每个设计节点问自己：

设计节点	Claude Code的选择	问自己
搜索/检索	grep，不用RAG	LLM的理解力能否补偿简单检索的不足？
持久化	纯文本文件	用最简单的存储方案，瓶颈在哪？
记忆	只记偏好	这个信息变化快还是慢？快的不记，慢的才记
安全	四层审查 + 熔断	用户需要多少信任才愿意给AI这个权限？
上下文管理	9段式结构化压缩	压缩时哪些信息绝对不能丢？
多Agent	Team + worktree隔离	Agent之间需要隔离还是共享？冲突怎么解决？
成本	prompt缓存 + 延迟加载	prompt的哪些部分是跨请求不变的？
演进	44个feature flags	新功能上线后出问题，能秒级回退吗？

案例研究：如果你要构建一个AI编程工具 Case Study: Building an AI Coding Tool from Scratch

假设你明天开始从零构建一个AI编程工具。不是Claude Code的克隆版，而是你自己的、针对你自己场景的AI编程助手。从Claude Code的源码中，你能学到什么？

你面对的第一个决策：代码检索用什么方案？

大多数人的第一反应是RAG，把代码库做embedding，存到向量数据库里，用语义搜索找相关代码。这是2024年最热门的方案，看起来最「AI原生」。

但Claude Code选择了grep。

原因在第8章讨论过：grep的结果是精确的、可解释的、零运维成本的。模型已经足够聪明，能理解grep返回的原始代码片段，不需要向量搜索帮它「预理解」一遍。这个决策的本质是：**把复杂度放在模型端（理解能力），而不是工具端（检索能力）**。底层模型足够强，工具层就应该尽可能简单。

你面对的第二个决策：要不要做权限系统？

很多开发者觉得权限系统是「以后再做」的功能，先让工具跑起来，权限慢慢加。但Claude Code从第一天就有四层权限审查、受保护路径列表、熔断机制。第5章分析过，这不是过度工程，而是**信任工程**。没有权限系统，用户不敢开Auto模式；不敢开Auto模式，工具的核心价值就打了折扣。权限系统不是限制AI的枷锁，而是释放AI能力的前提。

你面对的第三个决策：记忆系统记什么？

直觉是记越多越好，记住用户改过的每个函数、每次对话的完整上下文、所有文件的修改历史。但第6章告诉我们，Claude Code的CLAUDE.md只记偏好：用户喜欢什么代码风格、项目的技术栈是什么、哪些目录有特殊规则。事实和代码不记，因为它们变化太快，记了反而比不记更危险（模型会用过时的记忆覆盖真实的当前状态）。

三个决策，对应三个反直觉的选择。每个选择背后都有同一个思维模型：**简单组件 + 强大模型 > 复杂组件 + 任何模型**。这本书的11个章节拆解的就是这个思维模型在不同维度上的具体展开。

Harness Engineering vs Prompt Engineering The Bigger Picture

过去两年，整个行业都在讨论Prompt Engineering。怎么写更好的system prompt，怎么做few-shot，Chain-of-Thought还是Tree-of-Thought。这些都很重要，但Claude Code的源码让我们看到一个更大的图景。

Prompt Engineering是**Harness Engineering的一个子集**。

什么是Harness？Harness是围绕LLM构建的整个运行环境。它包含：

层级	对应本书章节	关注点
Prompt	Ch2 System Prompt	模型的角色、行为规范、输出格式
Tools	Ch3 工具系统	模型能做什么操作、工具的输入输出设计
Permissions	Ch5 安全模型	哪些操作需要审批、怎么分级、怎么熔断
Memory	Ch7 记忆系统	持久化什么、忘记什么、怎么召回
Context	Ch8 上下文管理	怎么压缩、保留什么、丢弃什么
Multi-Agent	Ch9 多Agent架构	组织结构、通信、隔离、冲突解决

大多数人只关注第一层Prompt，但Claude Code的源码证明，其他五层同样决定产品体验。同一个模型、同一段system prompt，套上不同的Tool设计就是完全不同的产品。加上不同的Permission策略，用户信任度天差地别。配上不同的Context管理方案，长对话的体验截然不同。

一个类比：**Prompt Engineering像写菜谱，Harness Engineering像开餐厅**。菜谱决定一道菜好不好吃，但餐厅的体验远不止菜谱。采购、厨房动线、服务流程、卫生管理、成本控制，每个环节都影响最终体验。一个只有好菜谱的餐厅开不下去，一个只有好prompt的AI产品也一样。

开放问题 Open Questions

这本书分析的是2026年4月的Claude Code源码。但harness engineering本身还在快速演化，有几个问题目前没有定论。

Harness会被模型能力升级淘汰吗？有人认为，当模型足够强，就不需要这些外围工程了。我倾向于认为不会。harness解决的很多问题和模型能力无关。权限控制是产品需求，不是模型缺陷；上下文窗口总是有限的，压缩策略总是需要的；多Agent的组织设计是人机协作的结构问题，不会因为模型更聪明就自动解决。模型变强会改变harness的实现方式（比如更简单的tool description就够用），但不会消除harness的必要性。

开源harness框架的未来？目前每个AI产品都在自己造轮子。Claude Code的harness、Cursor的harness、Windsurf的harness，底层思路相似但实现完全不同。会不会出现一个「harness框架」，像React之于前端、Rails之于后端那样，提供标准化的工具系统、权限系统、记忆系统？也许会，但harness的特殊性在于它和具体产品形态强耦合。编程工具的harness和客服机器人的harness差异太大，很难用同一个框架覆盖。

每个AI应用都需要自己的harness吗？如果你只是用模型做文本生成，可能不需要。但如果你的AI产品涉及工具调用、状态管理、用户交互、长期记忆中的任何一个，你就在做harness engineering——只是可能还没意识到。Claude Code的价值在于把这些隐性工程显性化了，让我们看到一个成熟的AI产品在每个层面上做了什么。

最后 Final Thoughts

Harness engineering还是一个非常早期的学科。没有教科书，没有公认的最佳实践，没有标准化的框架。Claude Code的源码是目前为止最好的一份参考实现，但它也只是一种可能的方案。

这本书尝试做的是：把一个价值数十亿美元的AI产品的设计蓝图翻译成可理解、可借鉴的工程原则。不是让你照抄Claude Code的实现（那没意义，技术栈和场景都不一样），而是理解每个设计决策背后的Why，然后把这些Why应用到你自己的AI产品中。

如果这本书让你在做下一个设计决策时，多问了一句「为什么Claude Code选择了另一种方式」，那它就完成了使命。

关注花叔

公众号 · B站 · 知识星球

公众号「花叔」：AI Native Coder的一线实战

B站「AI进化论-花生」：AI工具深度评测与教程

知识星球：AI编程：从入门到精通

B站 · X/Twitter · YouTube

Claude Code源码解析橙皮书 · 2026年4月

基于Claude Code v2.1.88源码分析