01


WARWOLF TEAM · OPEN SOURCE READING PROJECT
Claude Code 源码解析红宝书
基于 v2.1.88 双生逆向源码的设计思想深度解析
重点宣传
全书封面现已切换为 Claude Code

重点宣传

Claude Code 源码解析合成封面

首页主视觉、分享图与电子书封面同步升级

2026 SOURCE CODE DROP

源码解析合成海报，首页开屏直接突出“源码拆解”主题。

开源、免费，只为培养下一代中

国开源战狼

来自中国开源战狼团队

42 章正文

5 个附录

54 个工具

88 个命令

89 个 Feature Flag

开始阅读

下载 PDF

下载 EPUB

源码

PDF / EPUB 现在直接由站点提供下载，源码入口直达仓库主页；GitHub Releases 页面也会同步提供版本文件。

左侧目录负责全书导航，右侧内容区负责阅读与深入，尽量保持像一本书一样顺着读。

第二页 / 一起聊下一代技术

一群热爱技术的小伙伴一起探讨下一个时代最新技术进展

从 Claude Code、AI Agent、MCP 与插件生态，到源码逆向、自动化工作流和工程实

践，我们把“看懂”变成“能一起做出来”。

讨论方向

源码拆解、AI 工程化、开源协作、下一代工具链

交流方式

问题互答、资料共享、项目实战、路线讨论

适合谁

想认真理解 AI 编程助手的人，和愿意一起做开源的人

群聊: Claude Code源码交流群

该二维码7天内(4月12日前)有效，重新进入将更新

扫码加入交流群。若二维码过期，我们会在仓库中更新最新版本。

快速入口

01

欢迎来到源码的世界

先建立地图，再进入正文章节

GUIDE

三种读法与推荐路径

按你的背景选择最顺手的阅读路线

APPX

附录与速查资料

工具、命令、Feature Flag 和证据分级

四条设计思想主线

不是聊天壳

Claude Code 不是套了个壳的聊天应用，而是一个会调用工具的任务执行器。

核心铁三角

理解这套系统，关键是上下文装配、Agent Loop 和工具编排这三根主梁。

最难的是约束

真正的工程挑战不只是生成代码，而是权限、安全、压缩、恢复和一致性。

必须分清两套代码库

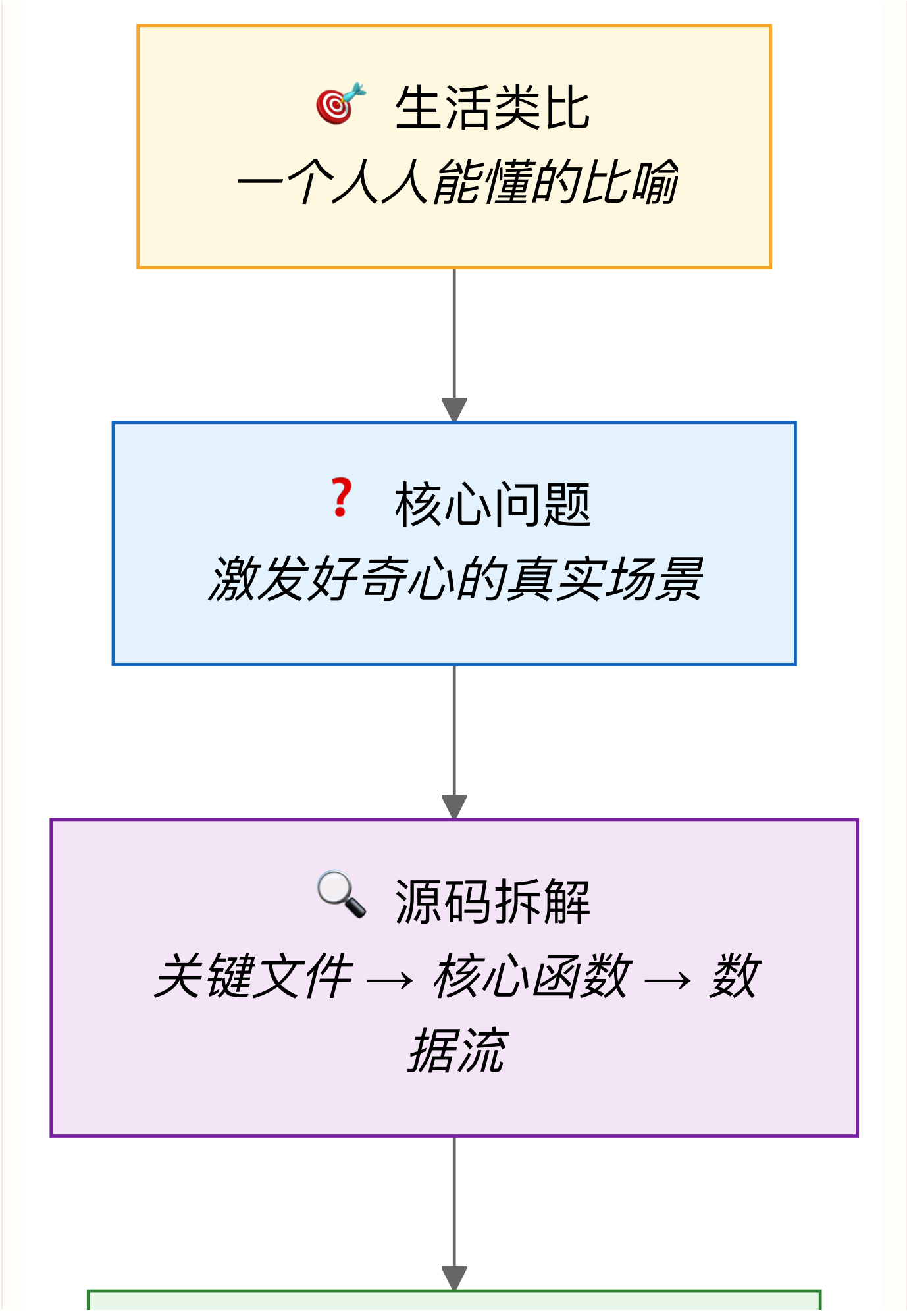
能跑起来不等于官方原始设计。阅读时要持续区分还原层和补全层。

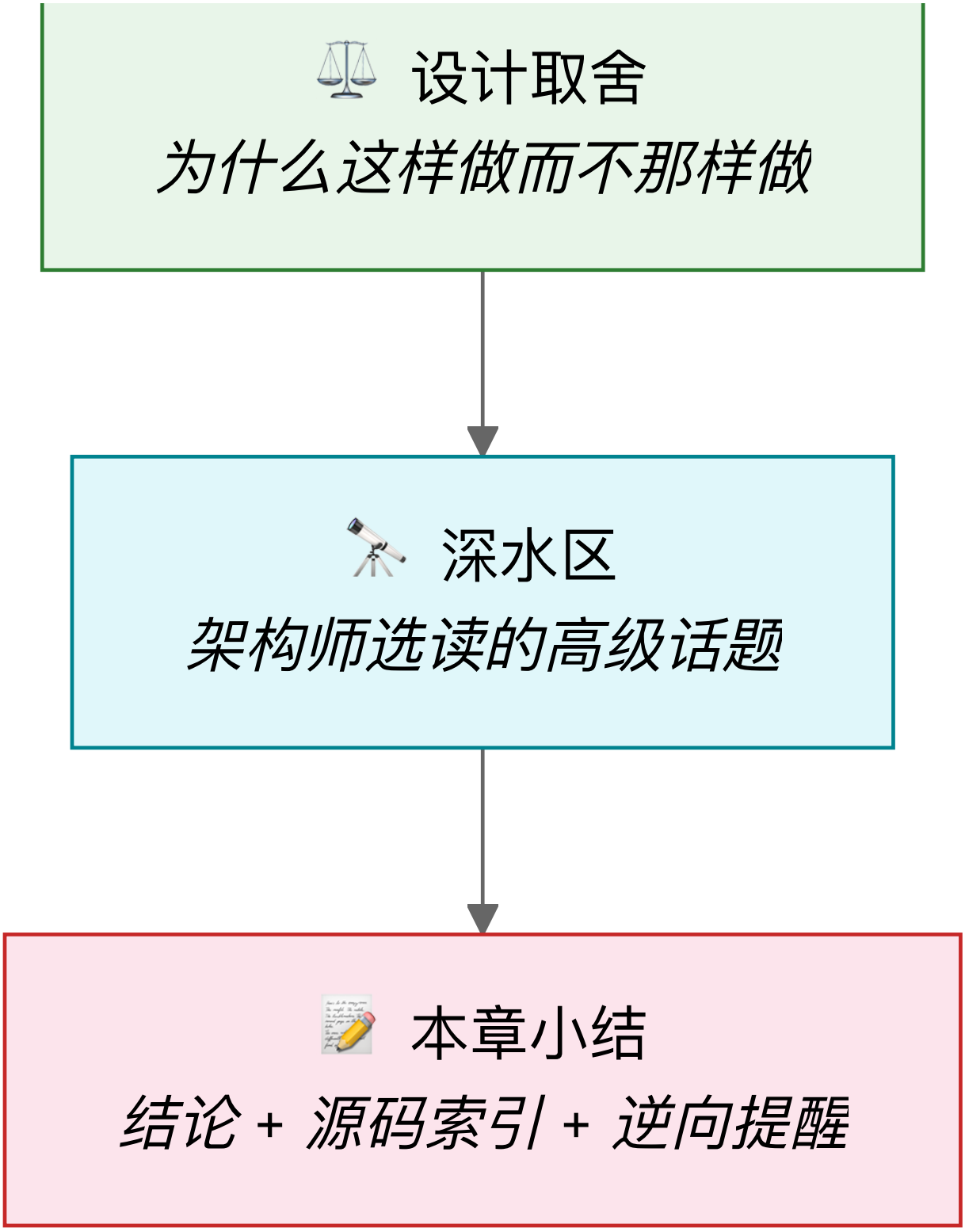
02

阅读指南

每章的阅读地图

本书每一章都遵循相同的结构，方便不同水平的读者各取所需：


🎯 生活类比
一个人人能懂的比喻
❓ 核心问题
激发好奇心的真实场景
🔍 源码拆解
关键文件 → 核心函数 → 数据流



所有人都读

生活类比 + 核心问题 —— 建立直觉，激发好奇心

有基础的读者继续

源码拆解 + 设计取舍 —— 理解"怎么做"和"为什么"

架构师选读

深水区 —— 高级话题、边界情况、竞品对比

特殊标记说明

本书使用以下标记帮助你快速定位内容：

可信度等级

每个源码引用都标注可信度：

等级	标记	含义
A级	A	确认原始 — Source Map 直接还原
B级	B	高度可信 — 主体原始，少量补全
C级	C	补全推测 — shim/stub/fallback

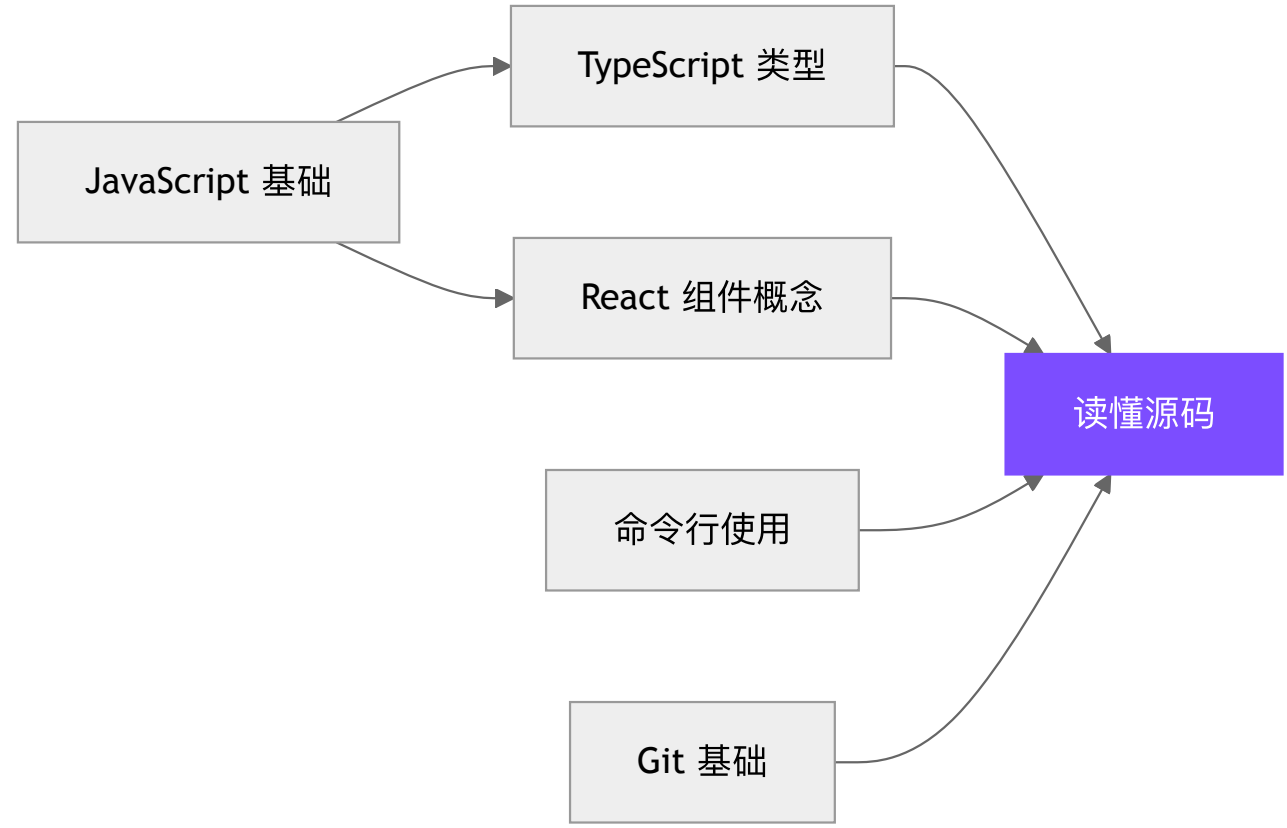
逆向提醒

每章末尾的逆向提醒用三个图标区分：

- ✅ RELIABLE：可以放心引用的分析
- ⚠️ CAUTION：需要注意版本差异或可能的变化
- ❌ SHIM/STUB：来自补全层，不代表官方实现

技术准备

阅读本书不需要成为 TypeScript 专家，但以下基础知识会帮助你更好地理解：



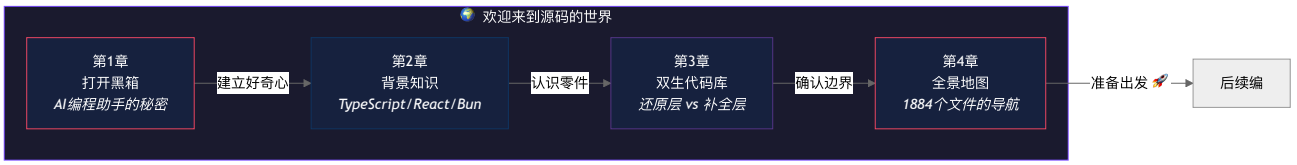
第2章会为你补充必要的背景知识，即使你目前只熟悉其中一两项也没关系。

03

第一编：欢迎来到源码的世界

> *打开黑箱之前，先建立好奇心和方向感。*
>
> 本编不写一行源码分析——我们先回答三个前置问题：**我们在看什么、怎么确认它可信、从哪里开始读**。

本编总览



本编四章速览

章	标题	核心问题	生活类比
1	打开黑箱	你在终端输入一句话，AI 帮你改了代码——中间到底发生了什么？	拆开收音机
2	背景知识	TypeScript、React、CLI 是什么？为什么选它们？	学开车前先认识仪表盘
3	双生代码库	59.8MB 的文件泄露了源码——但看到的都可信吗？	考古现场的两份拼图
4	全景地图	面对 1884 个文件，从哪里开始？怎么不迷路？	第一次走进大城市

设计思想主线

本编建立的认知基础

1. Claude Code **不是聊天壳**——它是一个能读写文件、执行命令的任务执行器
2. 源码来自 **两套逆向代码库**——必须区分"原始还原"和"社区补全"
3. **可信度分级**（A/B/C 三级）是后续所有分析的基础
4. 掌握 **六大架构层次**，后面怎么深入都不会迷路

推荐路径

○ ○ ○

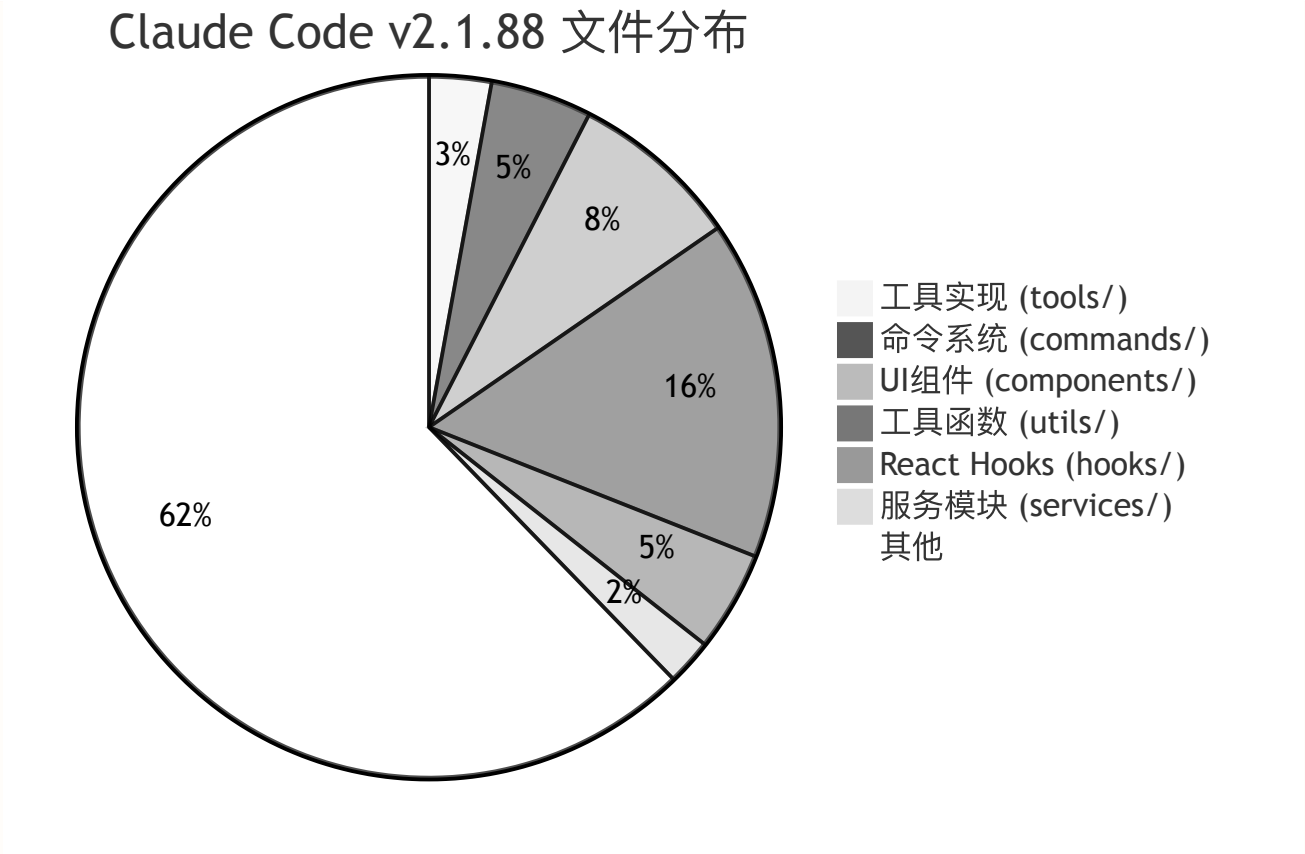
🌱 初学者 🔧 开发者 🏗️ 架构师

四章全读，**重点看生活类比和核心问题**。不需要理解所有技术细节，建立直觉即可。

第1章快速浏览，**第2章按需跳过已熟悉的技术**，第3–4章仔细读。

第1–2章跳过，**直接从第3章开始**——证据边界是一切分析的前提。第4章的深水区值得细读。

代码库数字画像



两套代码库对比

维度	sourcemap 还原层	OpenClaudeCode 补全层
文件数	1,884	1,989 (+105)
可运行	否	是
Shim	无	7 个
可信度	A 级为主	A/B/C 混合

04

入门 架构总览

第1章：打开黑箱——AI 编程助手的秘密

生活类比

你用过收音机吗？拧旋钮就能换台，但你知道里面发生了什么吗？拆开收音机看到电路板的那一刻，"换台"从魔法变成了物理。**读源码就是"拆开收音机"。**

这一章要回答的问题

你在终端输入一句话，AI 帮你改了代码——中间到底发生了什么？

你每天都在用 AI 编程助手，但它不是魔法。从你敲下一行自然语言，到代码被修改、文件被保存，中间有一条完整的执行链条。理解这条链条，是理解一切的起点。

1.1 Claude Code 是什么

一个运行在终端里的 AI 编程助手

你可能用过网页版的 Claude 对话——在浏览器里输入问题，AI 回复文字。Claude Code 不一样，它运行在你的**终端（Terminal）**里：

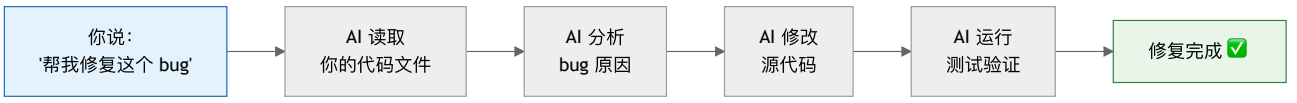
```
$ claude
```

```
╭──────────────────────────────────╮
│ ✻ Welcome to Claude Code!        │
│   /help for available commands   │
╰──────────────────────────────────╯
```

```
> 帮我修复这个 bug
```

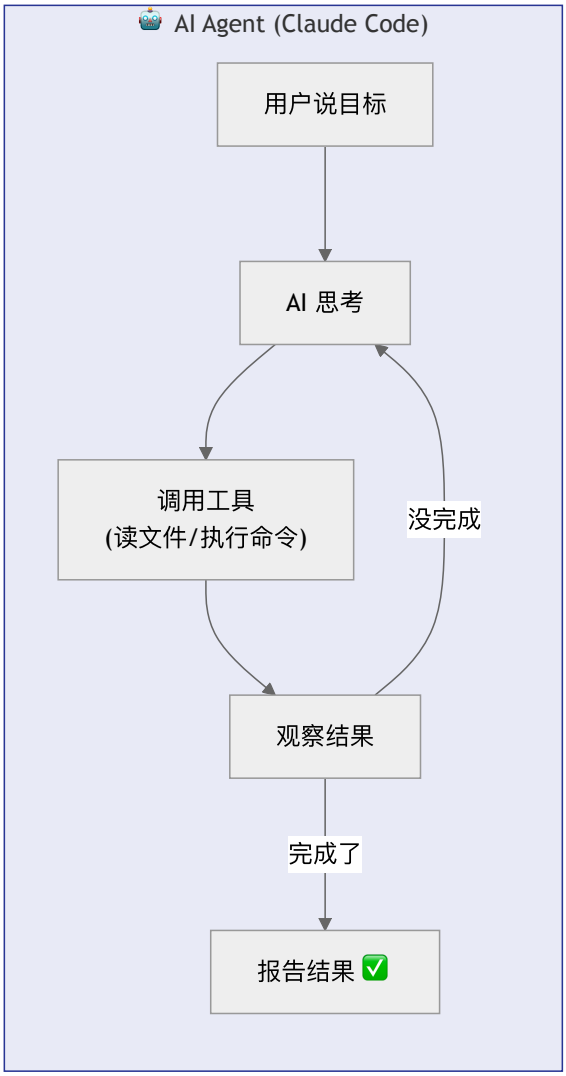
不只是"说话"——能"做事"

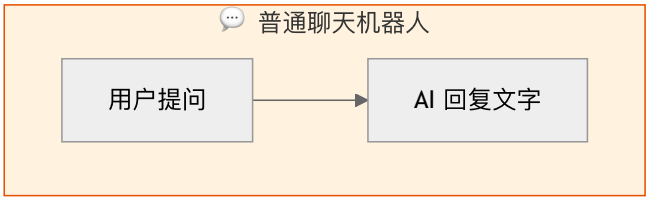
网页版 Claude 只能给你建议。Claude Code 可以**真正动手**：



这背后是 **54 个内置工具**在支撑——文件读写、Shell 执行、代码搜索、网页获取……AI 不只是"说"，它可以"做"。

Agent 与 Chatbot 的根本区别





这个**"思考 → 行动 → 观察 → 再思考"**的循环，就是所谓的 **Agent Loop**——它是整个 Claude Code 的心脏，我们将在第11章深入剖析。

1.2 为什么要读源码

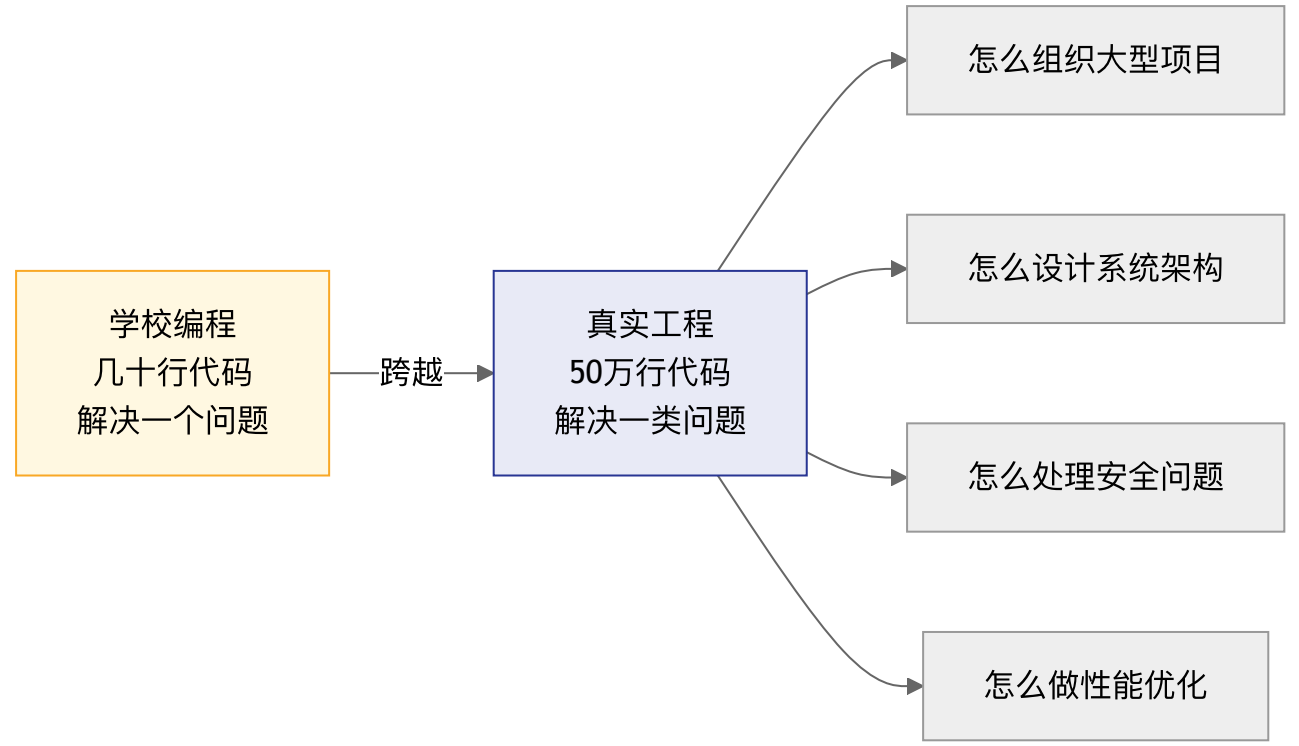
打开黑箱：从魔法到工程

当你知道了 Claude Code 的内部机制，很多"魔法"就变成了可以理解的工程：

你的疑问	源码给出的答案	对应章节
"它怎么知道要改哪个文件？"	工具系统告诉 AI 可以做什么	第14章
"它为什么有时候出错？"	Agent Loop 的停止条件和错误处理	第11章
"它怎么保证不搞坏我的代码？"	七层安全防御体系	第20章
"它怎么记住我之前说的？"	四层记忆架构	第29章
"多个 AI 能一起工作？"	三种多智能体路径	第33章

学习真实世界的工程

学校里的编程作业通常几十行代码解决一个小问题。Claude Code 有 **512,664 行代码**，分布在 **1,884 个文件**里。读它就像从练习本走进建筑工地：



站在巨人肩膀上

Claude Code 是 Anthropic 顶级工程师们的作品。他们的代码体现了多年的工程经验。理解他们的设计决策——**为什么这样做、为什么不那样做**——比学任何课程都管用。

1.3 这本书怎么读

三种读法

本书为三种读者设计了三条路径：

○ ○ ○

🌱 探索路径 🔧 实战路径 🏗️ 架构路径

适合：高中生、编程初学者

每章只读 **生活类比** 和 **核心问题**，跳过代码细节和深水区。你会建立对大型软件设计的直觉——这比任何教科书都生动。

推荐路线：1 → 4 → 5 → 8 → 11 → 14 → 20 → 29 → 39 → 42（10章精选）

适合：有经验的开发者

按编顺序通读，**重点看源码拆解和设计取舍**，深水区选读。你会获得可复用的架构模式。

推荐路线：按编顺序

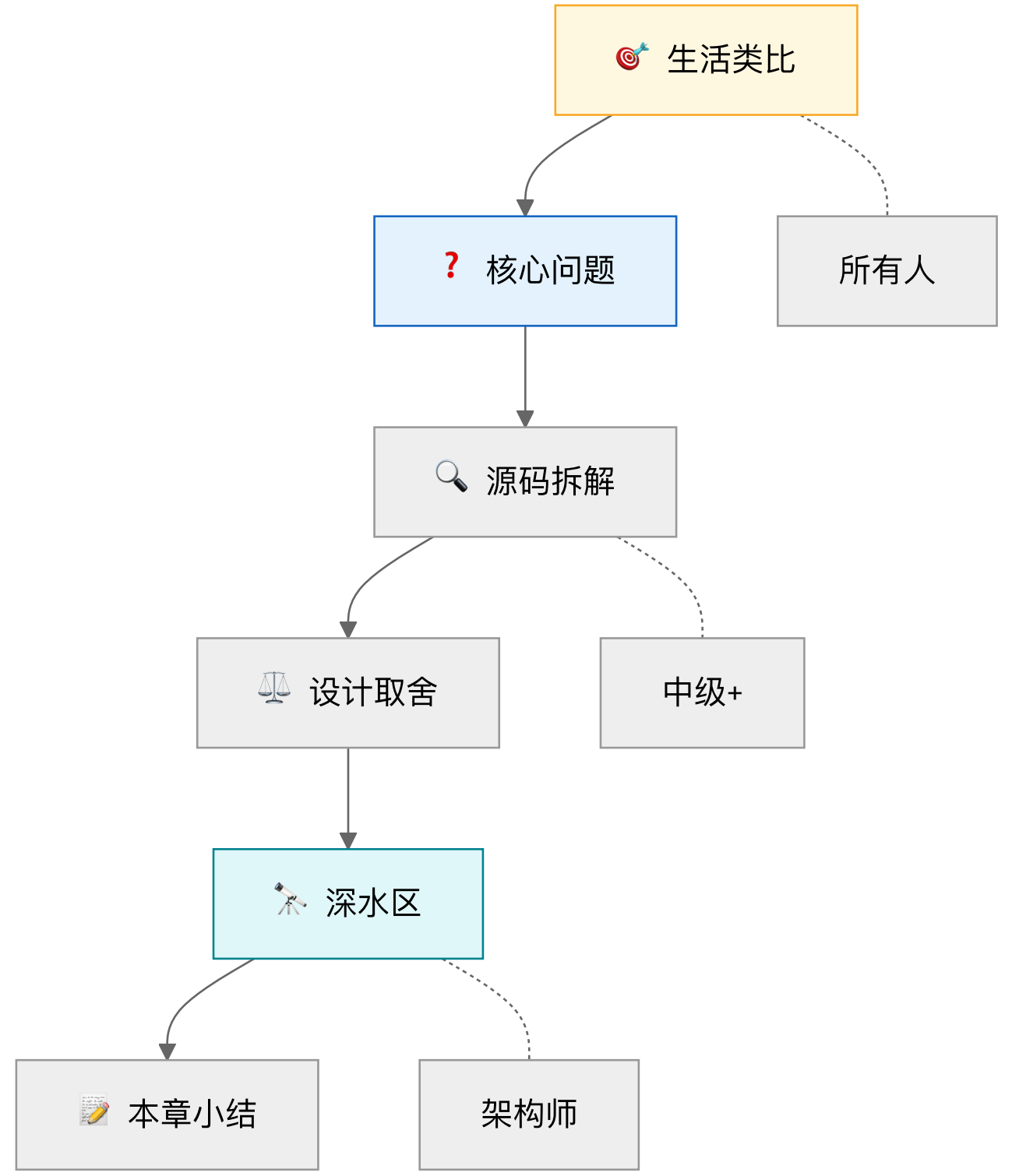
适合：架构师、AI Agent 开发者

先读**第3章**和**第41章**确立证据边界，再按兴趣深入每章的**深水区**。

推荐路线：3 → 41 → 11 → 14 → 20 → 25 → 33 → 39

每章的结构

每一章都遵循相同的阅读地图：

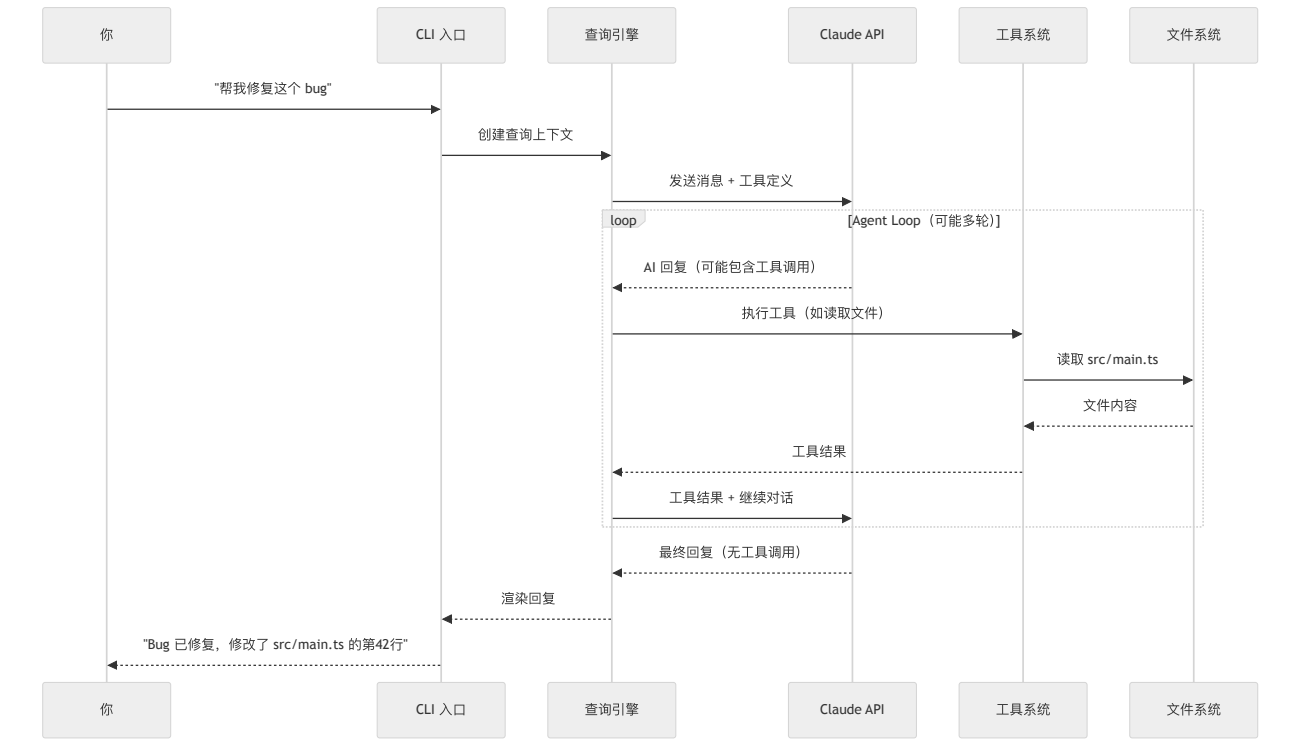


不需要读懂每一行代码

本书的目标是理解**设计思想**，不是逐行翻译代码。关键代码会附上解释，跳过细节不影响理解主线。

一条命令的完整旅程——预览

在深入各章之前，先预览一下全景：当你输入 `帮我修复这个 bug` 时，Claude Code 内部发生了什么：



这条旅程涉及本书几乎所有核心概念：

- **CLI 入口**（第5章）→ **查询引擎**（第10章）→ **Agent Loop**（第11章）
- **工具系统**（第14章）→ **文件工具**（第17章）→ **权限系统**（第21章）
- **流式响应**（第12章）→ **Token 管理**（第13章）

🔭 深水区（架构师选读）

AI Agent 框架在技术光谱中的位置

Claude Code 采用的是"**单模型、多工具、循环驱动**"的 Agent 架构——不依赖多模型编排（如 AutoGPT 的 GPT-4 + GPT-3.5 分工），也不依赖框架抽象（如 LangChain 的链式调用），而是让一个 Claude 模型在 `while(true)` 循环中自主决策工具调用。

这种架构的优势是**简洁和可控**——所有决策出自同一个模型，不存在多模型协调的一致性问题。代价是**单模型的能力上限即整个系统的上限**——但 Claude 模型本身的能力足够强，使这个取舍在当前阶段是合理的。

源码中的 `query.ts`（1,729行）就是这个循环的完整实现。我们将在第11章逐行拆解它。

本章小结

一句话：Claude Code 是一个能读写文件、执行命令的命令行 AI 助手，读它的源码就是拆开黑箱理解 AI 编程工具的真实运作方式。

关键源码索引

文件	职责	可信度
`src/bootstrap-entry.ts`	5行启动入口	A
`src/entrypoints/cli.tsx`	CLI 命令路由（302行）	A
`src/main.tsx`	主协调器（4,683行）	A
`src/query.ts`	Agent Loop 核心（1,729行）	A
`src/Tool.ts`	工具统一接口（792行）	A

逆向提醒

- ✅ **RELIABLE**：CLI 启动链条和 Agent Loop 的存在——完整的 Source Map 还原
- ⚠️ **CAUTION**：部分内部配置项（如模型名称、API 端点）可能随版本变化
- ❌ **SHIM/STUB**：无——本章为概述章节，不涉及特定实现细节

05

入门 技术栈

第2章：你需要知道的背景知识

生活类比

学开车之前，你至少要知道方向盘、油门、刹车在哪里。你不需要会修发动机，但要认识这几个零件。读 Claude Code 源码也一样——先认识四个核心"零件"就够了：**TypeScript 是发动机的语言，React+Ink 是仪表盘，Bun 是涡轮增压器，Zod 是安全气囊。**

这一章要回答的问题

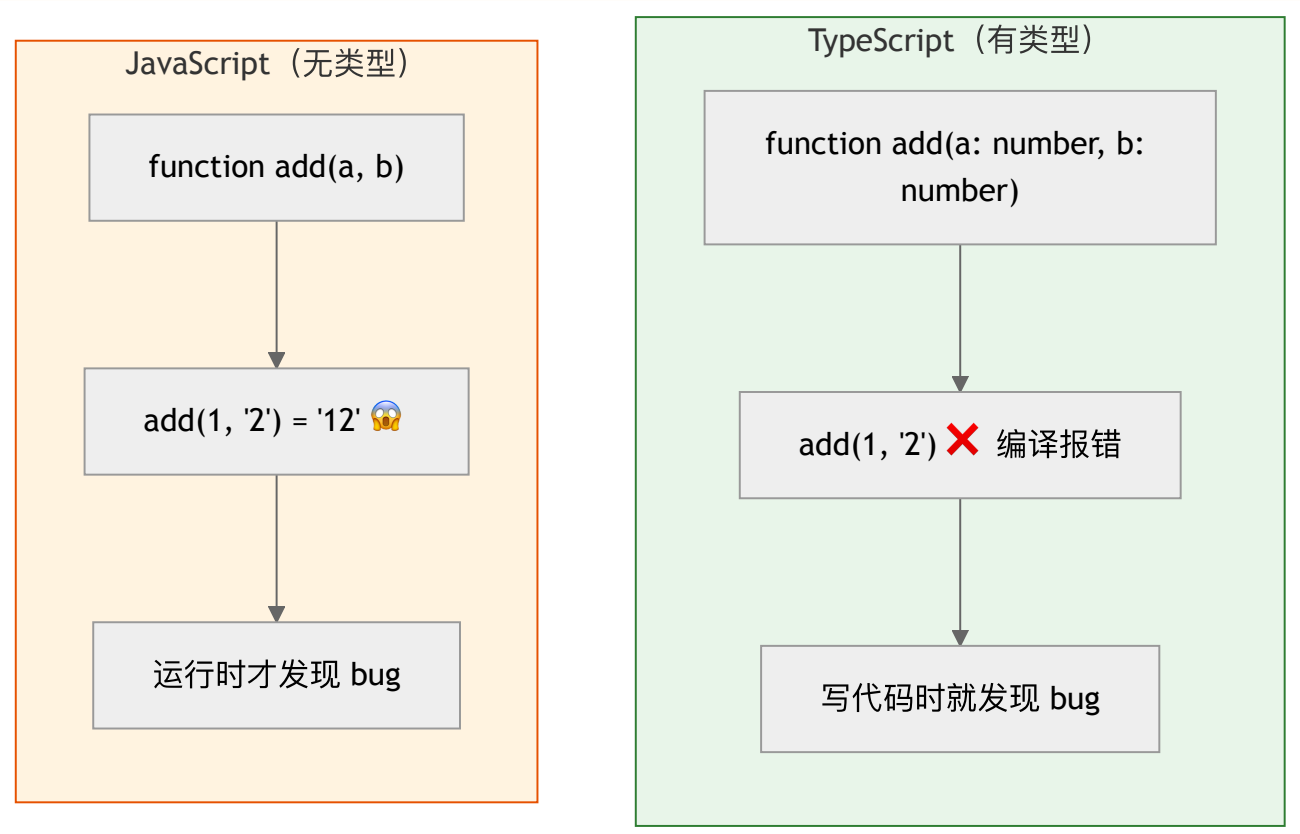
TypeScript、React、Bun、Zod 是什么？Claude Code 为什么选这些技术？

技术选型不是拍脑袋。Claude Code 选择 TypeScript 而非 Python，选择 React 而非直接打印字符，选择 Bun 而非 Node.js——每个选择都有工程上的理由。了解这些背景，你才能看懂后面的源码。

2.1 TypeScript：给 JavaScript 穿上铠甲

为什么 50 万行代码需要类型

JavaScript 是动态类型语言——一个变量可以上一秒是数字、下一秒变成字符串。写 50 行脚本时这很灵活，但在 **512,664 行**的大项目中，没有类型就像在黑暗中走钢丝：



TypeScript 在编译阶段就能发现类型错误，不用等到用户面前崩溃。当 AI 返回的 JSON 缺少字段时，TypeScript 能在你按下保存键时就告诉你，而不是在线上出事故。

Claude Code 的 TypeScript 配置

打开 `tsconfig.json`，最关键的几行：

```
{
  "compilerOptions": {
    "target": "ESNext",
    "module": "ESNext",
    "moduleResolution": "bundler",
    "jsx": "react-jsx",
```

```
    "strict": false,
    "types": ["bun"]
  }
}
```

配置项	含义	为什么这样选
`target: "ESNext"`	编译到最新 JS 标准	Bun 运行时支持最新语法，不需要降级
`jsx: "react-jsx"`	支持 JSX 语法	因为用了 React+Ink 做终端 UI
`strict: false`	没有开启最严格模式	50 万行代码全部严格化成本太高
`types: ["bun"]`	加载 Bun 的类型定义	让 TypeScript 认识 Bun 的 API

你需要记住的 TypeScript 基础

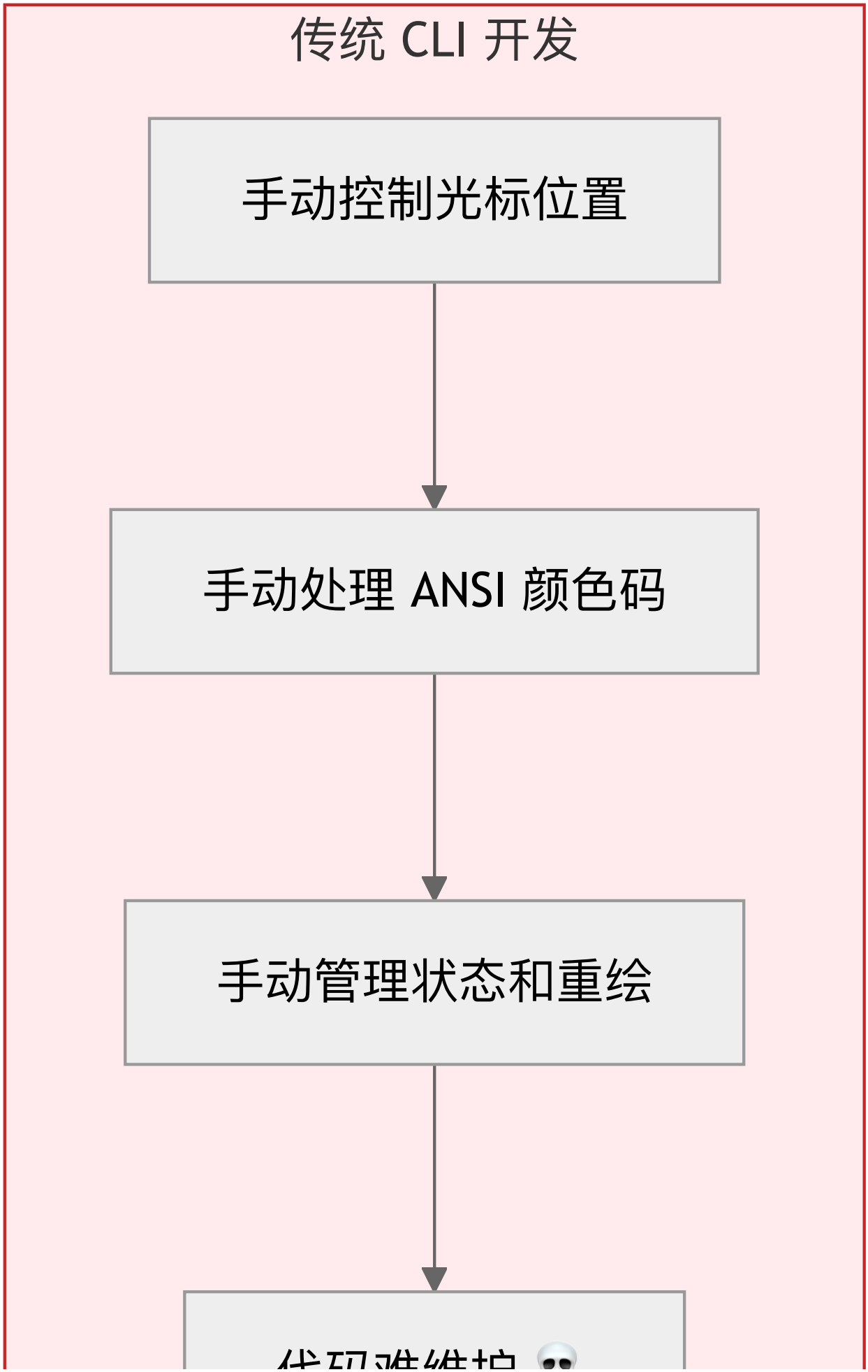
读后面的源码时，你会反复遇到这些语法：

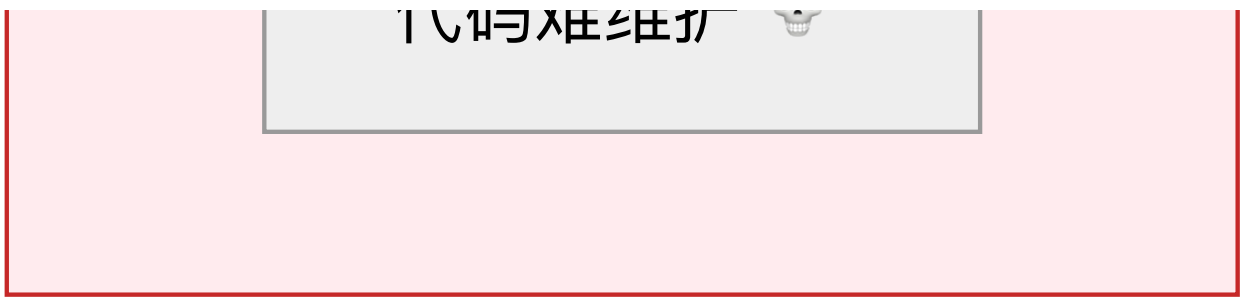
语法	含义	示例
`x: Type`	类型标注	`name: string`
`Type[]`	数组类型	`tools: Tool[]`
`T extends U`	泛型约束	`function parse<T extends Schema>()`
`interface`	接口定义	`interface ToolInput { ... }`
`as`	类型断言	`result as ToolOutput`
`?.`	可选链	`config?.model?.name`

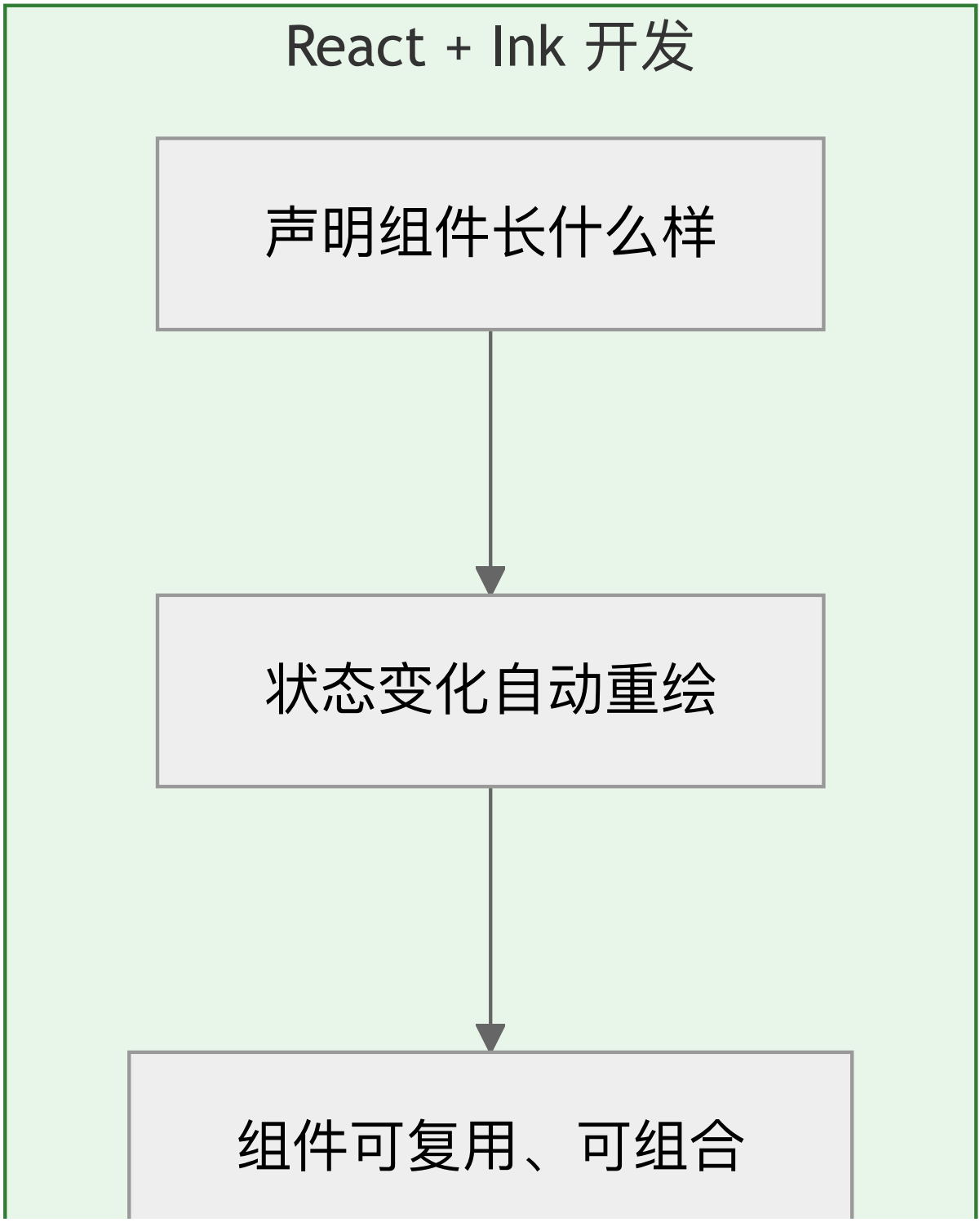
2.2 React 与 Ink：终端里的"前端框架"

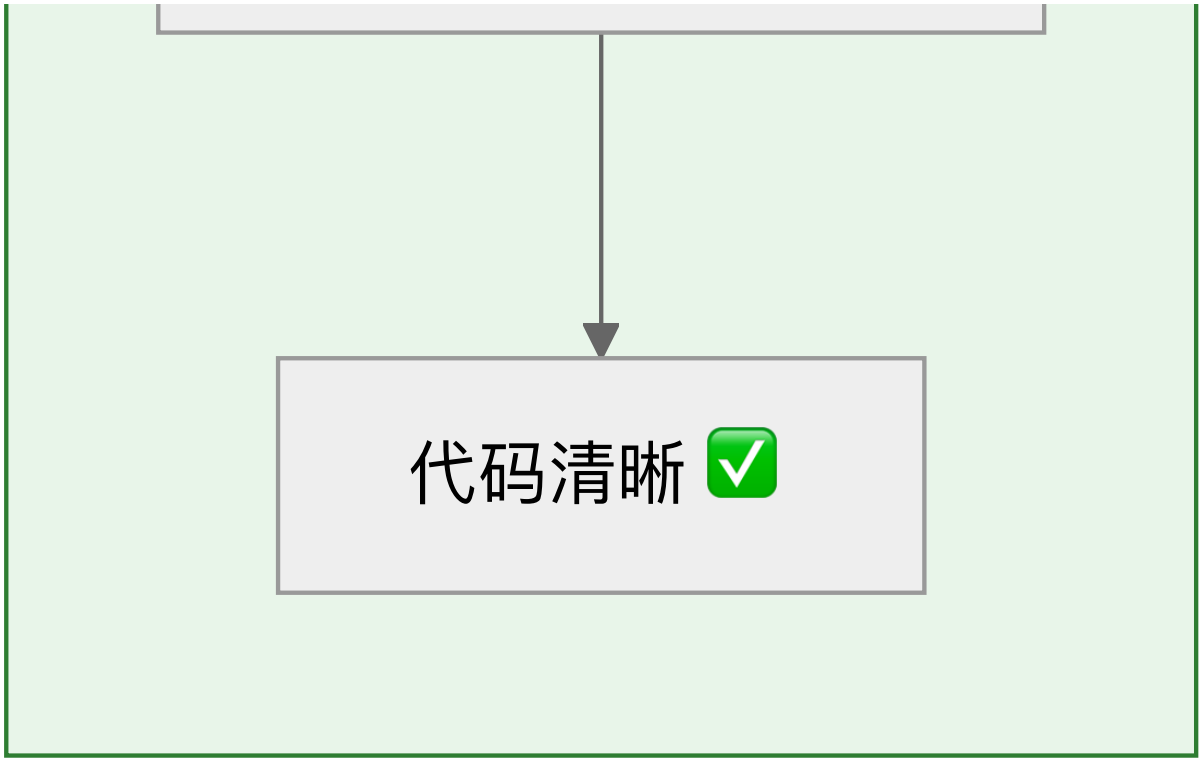
命令行也需要 UI 框架吗？

你可能觉得终端程序就是 `console.log`——打印一行文字而已。但看看 Claude Code 的界面：有彩色文字、有实时更新的进度条、有可交互的确认对话框、有 Markdown 渲染……这些用 `console.log` 拼出来要写多少 `if-else`？


传统 CLI 开发
手动控制光标位置
手动处理 ANSI 颜色码
手动管理状态和重绘

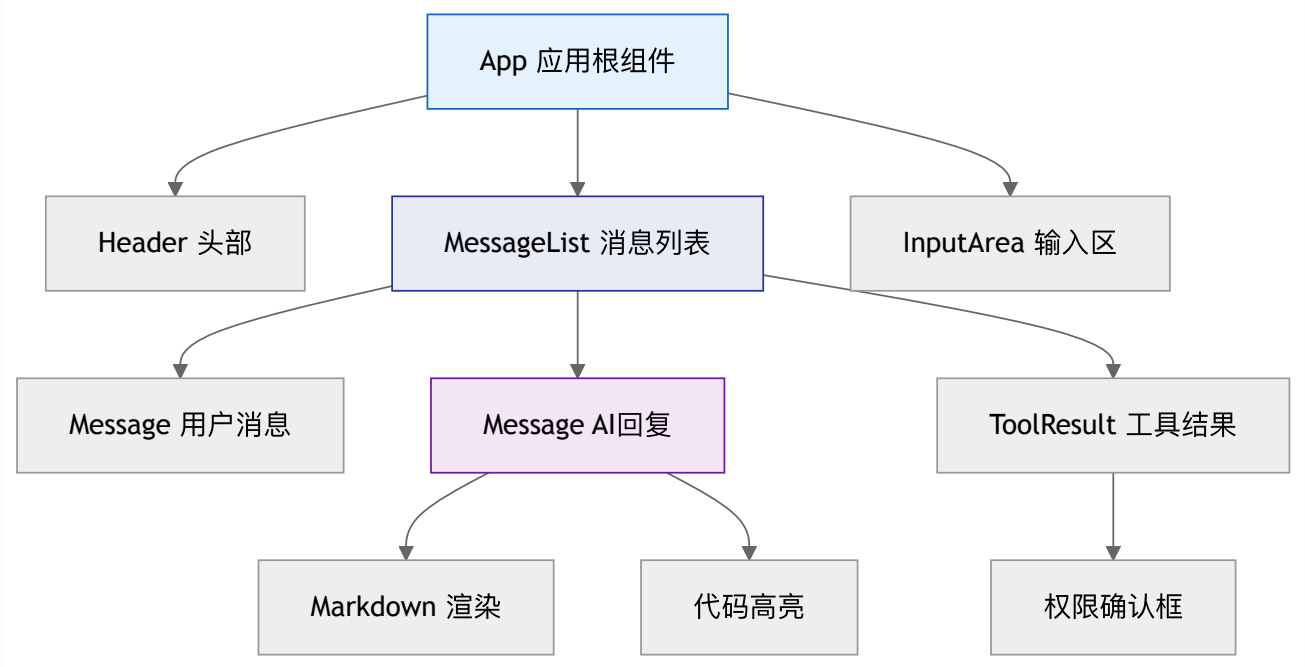

代码难维护


React + Ink 开发
声明组件长什么样
状态变化自动重绘
组件可复用、可组合



组件思维：把 UI 拆成积木

React 的核心思想是**组件化**——把界面拆成独立的、可复用的积木块：



在 Claude Code 中，Ink 框架让 React 组件渲染到终端而非浏览器。`<Box>` 替代 `<div>`，`<Text>` 替代 `<span>`——概念完全一样，只是渲染目标从像素变成了字符。

声明式 vs 命令式

传统 CLI：你要告诉计算机"先清屏、在第 3 行第 5 列打印蓝色文字、光标移到下一行……"——这叫**命令式**。

React 方式：你只说"我要一个蓝色文字组件，内容是这个"——React 自动算出怎么画、状态变了自动重绘——这叫**声明式**。

`package.json` 中的关键依赖：

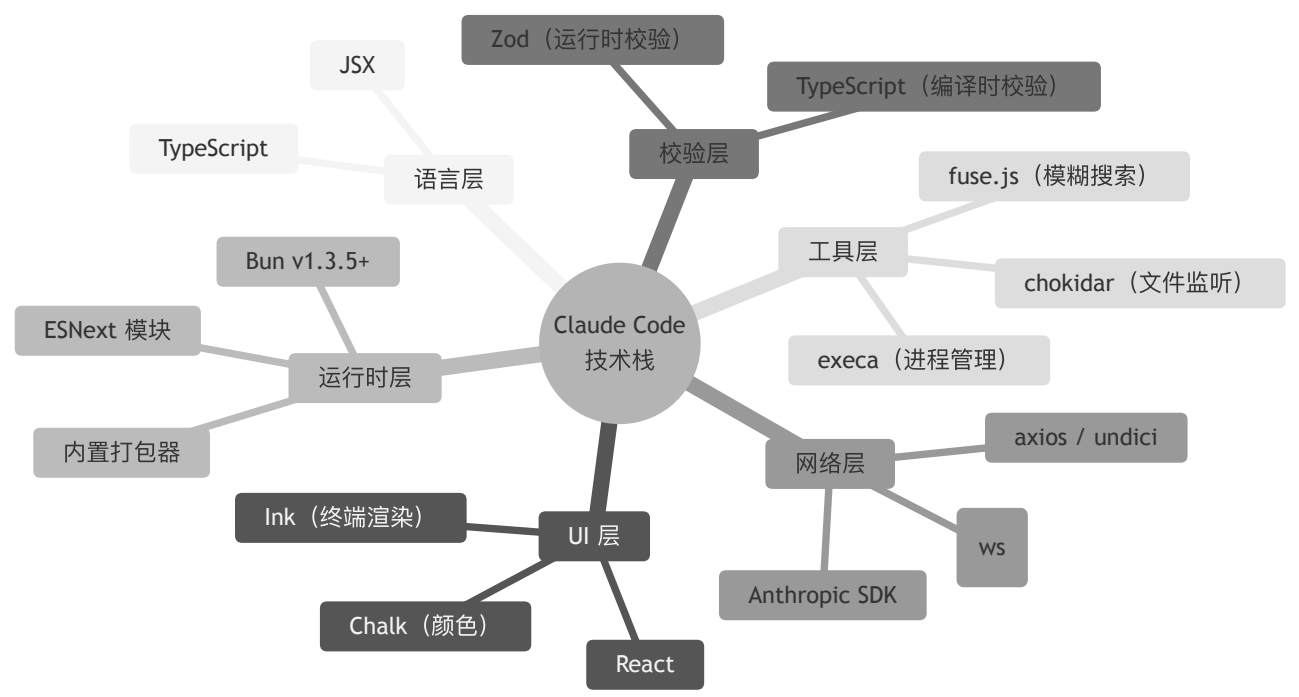
依赖	版本要求	角色
`react`	*	UI 组件模型

依赖	版本要求	角色
`react-reconciler`	*	React 自定义渲染器核心
`ink`	*	终端渲染适配层
`chalk`	*	终端颜色与样式
`cli-boxes`	*	终端边框与布局
`wrap-ansi`	*	文本自动换行

Claude Code 甚至自研了一套 Ink 实现（`src/ink/` 目录），对官方 Ink 做了深度定制——这是为了解决终端环境下的特殊渲染需求。

2.3 Bun 运行时与 Zod 校验

Bun：更快的 JavaScript 引擎



Bun 是一个用 Zig 语言编写的 JavaScript 运行时。和 Node.js 做同样的事，但有三个关键优势：

对比维度	Node.js	Bun
冷启动速度	~200ms	~50ms
包管理器	npm/yarn/pnpm（需额外安装）	内置
打包能力	需要 webpack/esbuild	内置
TypeScript	需要 ts-node 或编译	原生支持

`package.json` 中明确声明了 `"packageManager": "bun@1.3.5"`，并且启动脚本直接使用 `bun run`：

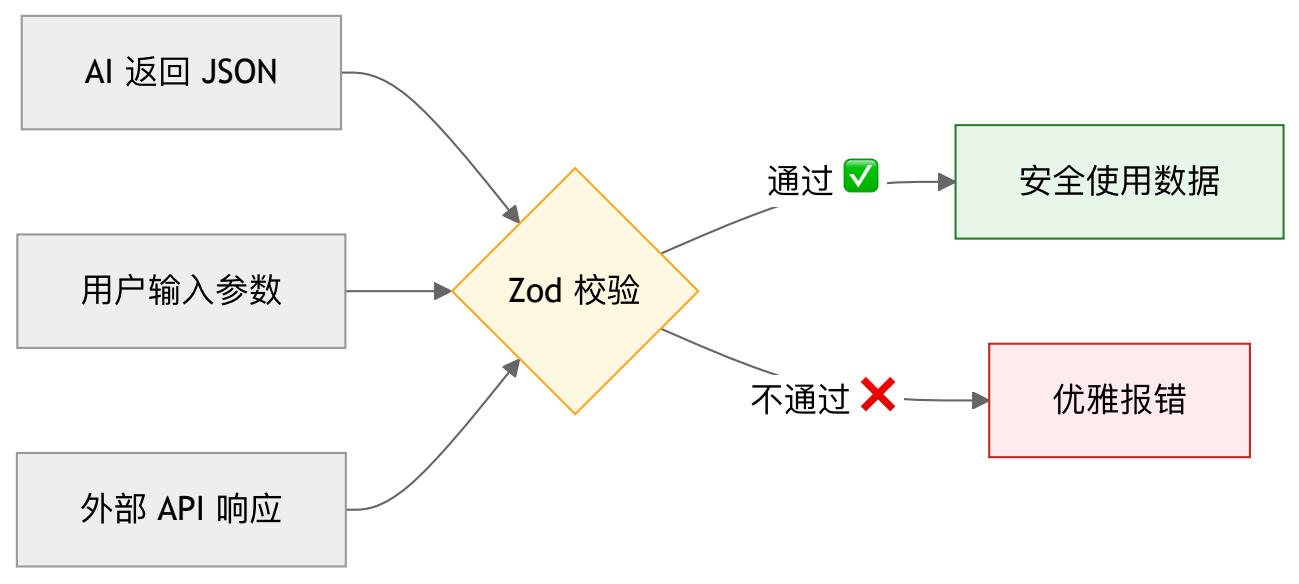
```
"scripts": {
    "dev": "bun run ./src/bootstrap-entry.ts",
    "start": "bun run ./src/bootstrap-entry.ts"
}
```

对 CLI 工具来说，**冷启动速度是关键体验**。用户输入 `claude` 后等待 200ms 还是 50ms，感受完全不同。Bun 的内置打包器还能把整个项目打成单个 `cli.js` 文件，简化分发。

Zod：运行时的最后一道防线

TypeScript 的类型只在编译时有效——代码运行起来后，类型信息就消失了。但 AI 返回的 JSON、用户输入的参数、外部 API 的响应——这些都是**运行时**才到达的数据。谁来保证它们符合预期？

Zod 就是运行时的类型守卫：



在 Claude Code 中，每个工具（Tool）的输入参数都用 Zod 定义 Schema：

- 编译时：TypeScript 确保**你写的代码**类型正确
- 运行时：Zod 确保**外部传入的数据**格式正确
- 给 AI：Zod Schema 还能自动转成 JSON Schema，告诉 Claude 模型每个工具需要什么参数

这就是**双层类型安全**——编译时和运行时各守一层，不留死角。

其他关键依赖一览

`package.json` 中的 97 个依赖各有分工，这里列出你在后续章节会频繁遇到的：

依赖	用途	出现章节
`@anthropic-ai/sdk`	Claude API 客户端	第9章
`@modelcontextprotocol/sdk`	MCP 协议客户端	第25章
`commander` (extra-typings)	命令行参数解析	第5章
`execa`	子进程管理	第15章
`chokidar`	文件系统监听	第17章
`marked`	Markdown 解析	第12章
`highlight.js`	代码语法高亮	第12章
`fuse.js`	模糊搜索	第16章
`@opentelemetry/*`	可观测性追踪	第36章
`@growthbook/growthbook`	功能开关（Feature Flag）	第35章
`ws`	WebSocket 通信	第26章

◯ ◯ ◯

🌱 探索路径 🔧 实战路径 🏗️ 架构路径

记住四个零件的**角色**就够了：TypeScript = 安全网，React+Ink = 终端 UI 框架，Bun = 快速引擎，Zod = 数据守卫。不需要深入语法。

建议打开 `package.json` 和 `tsconfig.json` 实际看一遍。特别注意 `"strict": false` 和 `"types": ["bun"]`——这两个配置影响你读源码时 IDE 的提示。

关注**技术选型的取舍**：为什么 `strict: false`？为什么自研 Ink 而非用官方版？为什么 97 个依赖全部用 `*` 通配版本？这些决策反映了工程优先级的排序。

🔭 深水区（架构师选读）

TypeScript 的类型体操在 Claude Code 中的应用

Claude Code 中有不少高级类型应用。`Tool<Input, Output, Progress>` 泛型让 54 个工具在定义阶段就能获得完整的类型推导——输入参数的 Zod Schema 自动推导出 TypeScript 类型，工具的返回值类型自动传递给调用方。`DeepImmutable<T>` 递归地将对象所有层级设为只读，防止状态被意外修改。`type-fest` 库提供了额外的工具类型如 `PartialDeep`、`SetRequired` 等。

另一个值得注意的设计：`package.json` 中所有依赖版本都使用 `"*"` 通配符。这不是偷懒——因为这是还原项目，实际发布时 Bun 的打包器会将所有依赖打入单文件 `cli.js`，版本在打包时已经锁定。消费者安装的是打包产物，不会触发依赖解析。

本章小结

一句话：TypeScript 提供编译时类型安全，React+Ink 实现声明式终端 UI，Bun 加速启动与打包，Zod 守卫运行时数据——这四个"零件"是理解 Claude Code 源码的前置知识。

关键源码索引

文件	职责	可信度
`package.json`	97 个依赖声明与启动脚本	A
`tsconfig.json`	TypeScript 编译配置（29行）	A
`src/ink/`	自研 Ink 终端渲染框架	A
`src/types/`	全局类型定义	A
`src/Tool.ts`	工具泛型接口（792行）	A

逆向提醒

- ✅ RELIABLE：`package.json` 中的依赖列表和脚本定义——与 npm 发布包一致
- ⚠️ CAUTION：`tsconfig.json` 的部分编译选项可能与 Anthropic 内部发布配置不同
- ❌ SHIM/STUB：无——本章为背景知识章节，不涉及特定实现细节

06

入门 可信度 逆向工程

第3章：双生代码库：我们到底在看什么

生活类比

考古学家挖出一座古城遗址，发现了两份拼图。第一份是从废墟中原样挖出的碎片（Source Map 还原）——虽然有些地方模糊了，但每一块都是真迹。第二份是有人参照碎片、补上缺失部分后重建的完整拼图（OpenClaudeCode）——更完整，但你必须分清**哪些是原始的、哪些是后来补的**。读源码的第一课不是"怎么读"，而是**"你看到的可信吗"**。

这一章要回答的问题

一份 59.8MB 的文件泄露了整个源码——但"看到的"都可信吗？

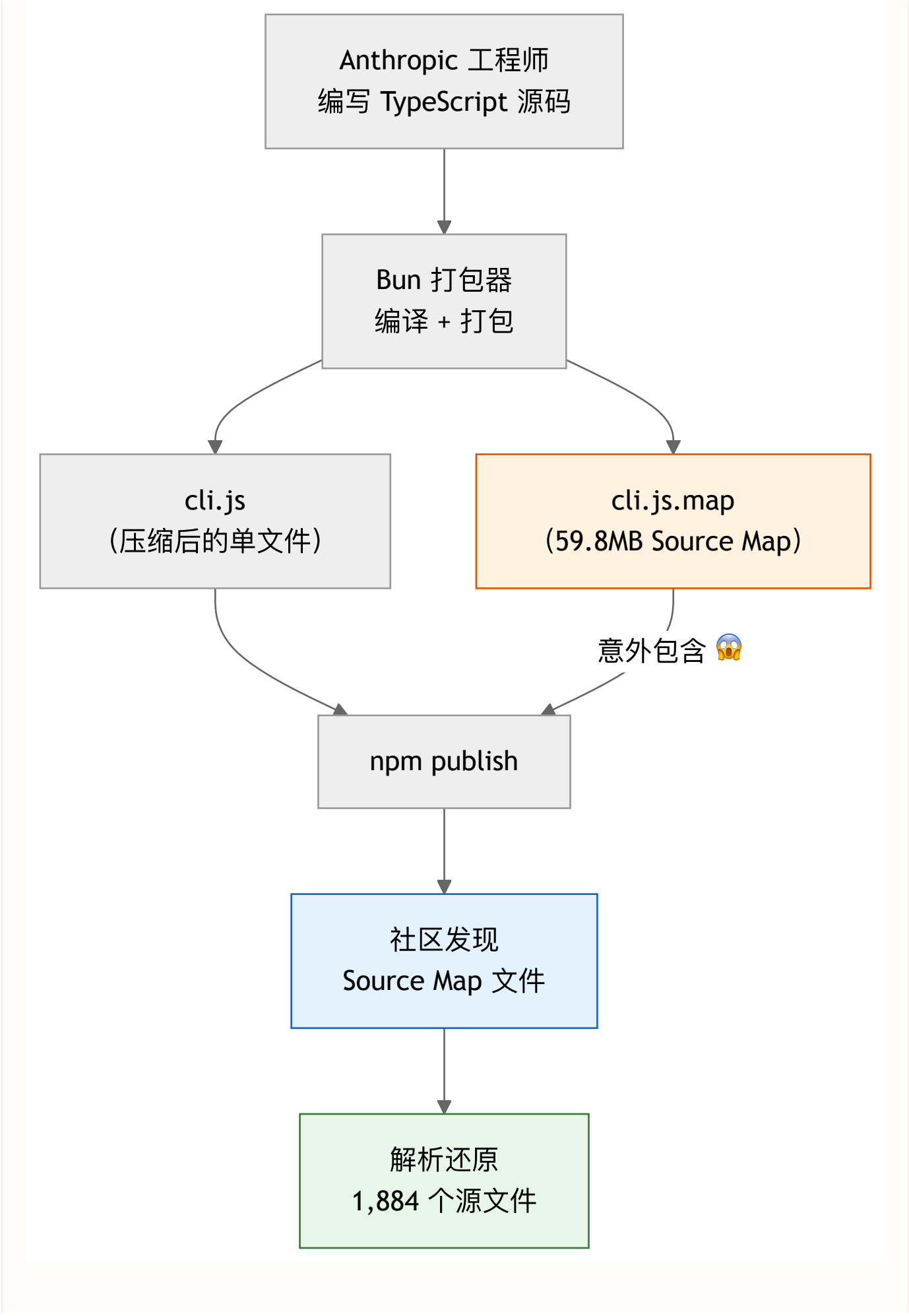
源码逆向不像正常阅读开源项目——你看到的每一行代码都需要评估其可信度。理解两套代码库的来源和差异，是后续所有分析的基础。没有这个判断力，你可能会把社区补全代码当成 Anthropic 的原始设计来学习。

3.1 Source Map 泄露事件

npm 包中的意外

Claude Code 作为 npm 包 `@anthropic-ai/claude-code` 发布。正常情况下，发布的只有打包后的 `cli.js`——一个压缩混淆过的单文件。但在某个版本中，`cli.js.map` 被意外包含在了 npm 包里。

这个 **59.8MB** 的 Source Map 文件，包含了几乎完整的源码信息。


Anthropic 工程师
编写 TypeScript 源码
Bun 打包器
编译 + 打包
cli.js
(压缩后的单文件)
cli.js.map
(59.8MB Source Map)
意外包含 😱
npm publish
社区发现
Source Map 文件
解析还原
1,884 个源文件

什么是 Source Map

Source Map 是一种标准技术（V3 版本），最初用于浏览器调试——让你在浏览器中调试压缩代码时，能看到原始的源码位置。它的核心结构：

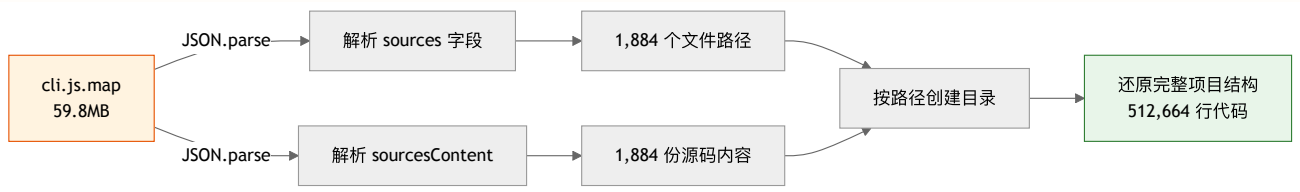
```
{
  "version": 3,
  "sources": ["src/main.tsx", "src/query.ts", "...共1884个"],
  "sourcesContent": ["文件1的完整内容", "文件2的完整内容", "..."],
  "mappings": "AAAA,SAAS,IAAI,MAAM;..."
}
```

- `sources`：所有原始文件的路径列表
- `sourcesContent`：每个文件的**完整源码内容**
- `mappings`：压缩代码与原始代码的位置映射（VLQ 编码）

正是 `sourcesContent` 字段，让我们能完整还原出每个源文件的内容——包括注释、空行、原始格式。

1,884 个文件的还原

通过解析 Source Map 的 `sources` 和 `sourcesContent` 字段，社区还原出了 **1,884 个原始文件**，总计约 **512,664 行代码**。文件路径、目录结构、甚至开发者的注释都被完整保留。



这就是本书的**主要分析对象**——从 Source Map 中还原的原始代码。

3.2 两套代码库的关系

第一层：Source Map 还原（1,884 个文件）

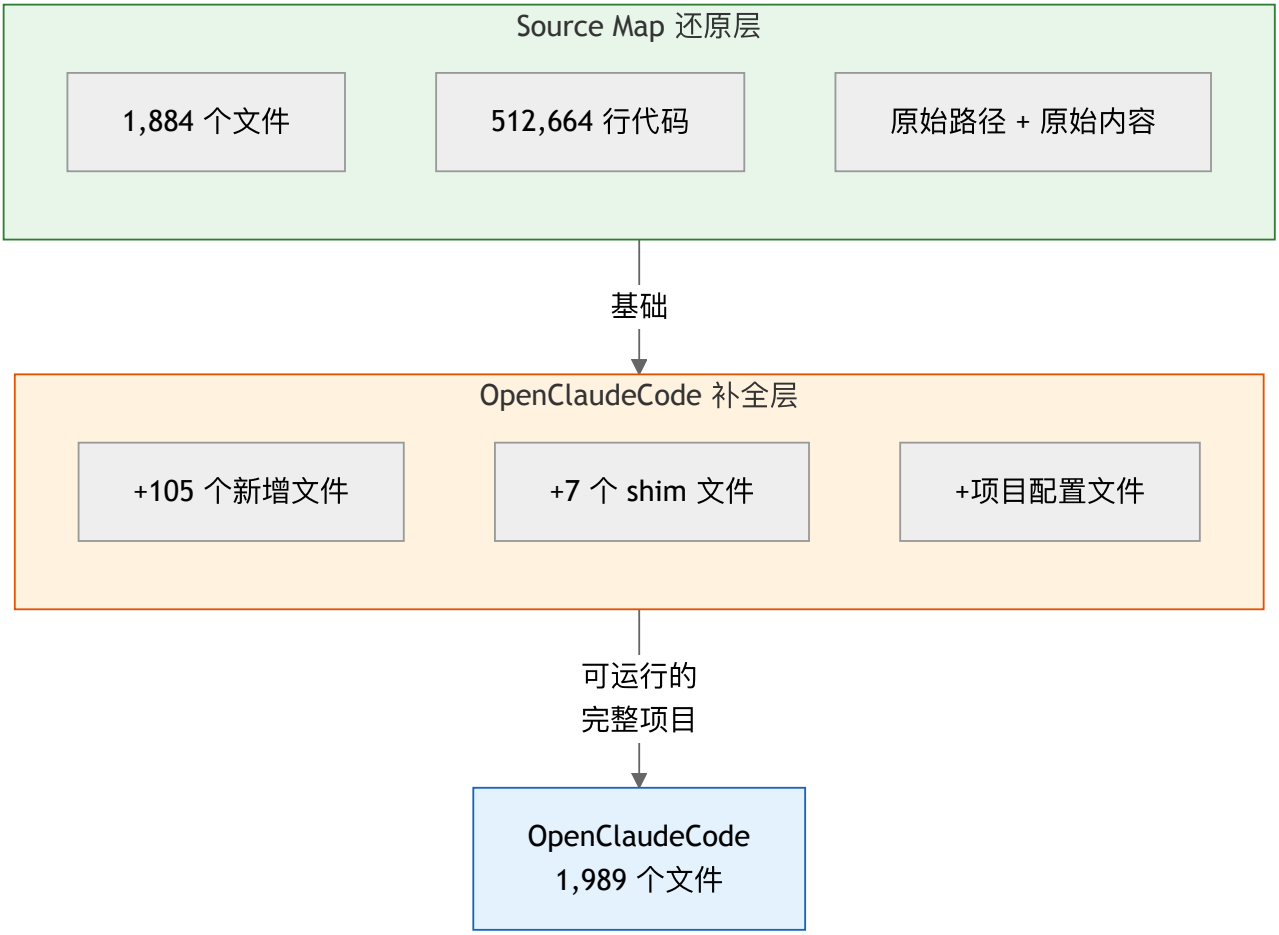
直接从 `cli.js.map` 中提取的代码，拥有最高可信度：

- 每个文件都有明确的原始路径（如 `src/main.tsx`、`src/query.ts`）
- 代码内容来自 `sourcesContent` 字段，未经第三方修改
- 与 `cli.js` 中的实际执行代码有映射关系可以交叉验证

第二层：OpenClaudeCode 补全（1,989 个文件）

社区项目 OpenClaudeCode 在 Source Map 还原的基础上做了增强：

- 新增了约 **105 个文件**（1,989 − 1,884 = 105）
- 补全了缺失的类型声明文件（`.d.ts`）
- 添加了 `package.json`、`tsconfig.json` 等项目配置
- 引入了 7 个 **shim** 文件来填补运行时缺失



7 个 Shim 文件——社区的"桥梁"

OpenClaudeCode 引入了 7 个 shim（垫片）来让还原代码能实际运行。这些 shim 替代了 Anthropic 内部的私有模块：

Shim 名称	替代目标	功能
`ant-claude-for-chrome-mcp`	`@ant/claude-for-chrome-mcp`	Chrome 浏览器 MCP 集成
`ant-computer-use-input`	`@ant/computer-use-input`	计算机操作输入处理
`ant-computer-use-mcp`	`@ant/computer-use-mcp`	计算机操作 MCP 服务
`ant-computer-use-swift`	`@ant/computer-use-swift`	macOS 原生交互（Swift）
`color-diff-napi`	`color-diff-napi`	颜色差异对比（原生模块）
`modifiers-napi`	`modifiers-napi`	键盘修饰键检测（原生模块）
`url-handler-napi`	`url-handler-napi`	URL 协议处理（原生模块）

在 `package.json` 中，这些 shim 通过 `file:` 协议引用本地目录：

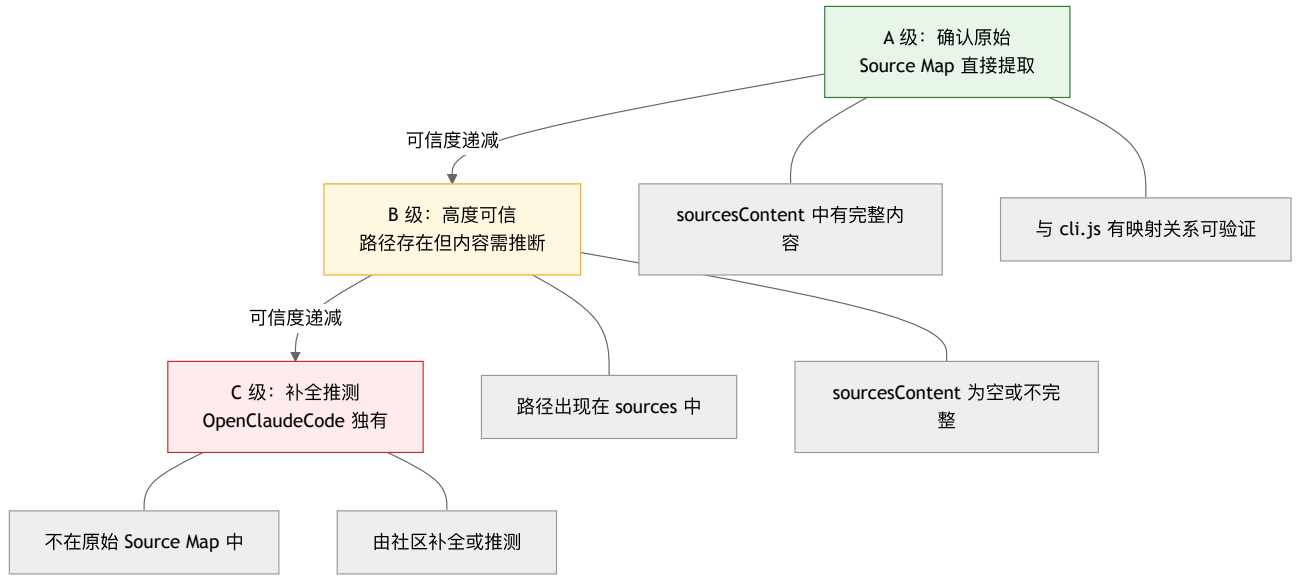
```
{
  "@ant/claude-for-chrome-mcp": "file:./shims/ant-claude-for-chrome-mcp",
  "@ant/computer-use-input": "file:./shims/ant-computer-use-input",
  "color-diff-napi": "file:./shims/color-diff-napi"
}
```

每个 shim 目录都包含一个 `index.ts` 和 `package.json`——它们是社区根据代码上下文**推测**的实现，不是 Anthropic 的原始代码。

3.3 可信度分级：A / B / C 三级标准

为什么需要分级

这不是一个开源项目——我们看到的代码是逆向还原的。不同来源的代码可信度不同，如果不加区分地引用，就会把推测当成事实。本书为每个源码引用标注可信度等级：



A 级：确认原始

来源：直接从 Source Map 的 `sourcesContent` 中提取的文件。

特征：

- 文件内容与 `cli.js` 中的打包代码有精确的行列映射关系
- 包含原始的注释、空行、代码格式
- 经过 `mappings` 字段交叉验证

代表文件：`src/main.tsx`（4,690行）、`src/query.ts`（1,729行）、`src/Tool.ts`（792行）

本书 **90% 以上的分析**基于 A 级文件。

B 级：高度可信

来源：路径存在于 Source Map 的 `sources` 字段中，但 `sourcesContent` 对应位置为空或不完整。

特征：

- 我们知道这个文件**存在**，但不确定其完整内容
- 可以通过上下文（import 关系、类型引用）推断部分内容
- 需要结合 OpenClaudeCode 的补全来理解

处理方式：本书引用 B 级文件时，会标注"内容基于上下文推断"。

C 级：补全推测

来源：OpenClaudeCode 独有的文件，不在原始 Source Map 中。

特征：

- 7 个 shim 文件全部属于 C 级
- 部分类型声明文件属于 C 级
- 可能正确，也可能与 Anthropic 真实实现有偏差

处理方式：本书引用 C 级文件时，会明确标注"社区补全，仅供参考"，避免误导。

可信度在表格中的标记

在后续章节的"关键源码索引"表格中，你会看到这样的标记：

可信度标记	含义	使用建议
A	Source Map 直接还原	可放心引用作为分析依据
B	路径可信，内容需验证	结合上下文谨慎引用
C	社区补全 / Shim	仅作参考，不作为结论依据

○ ○ ○

🌱 探索路径 🔧 实战路径 🏗️ 架构路径

记住一个核心结论：**我们有两套代码，一套是"原版拼图碎片"（1,884 文件），一套是"补全后的完整拼图"（1,989 文件）。看到 A 级标记的代码可以信赖，看到 C 级的要打个问号。**

建议自己动手用 `JSON.parse` 解析一下 Source Map 文件，亲眼看看 `sources` 和 `sourcesContent` 的结构。理解逆向还原的过程，能帮你判断后续遇到的任何可疑代码。

重点关注 7 个 shim 的设计——它们代表了 Anthropic 私有模块的**接口边界**。即使 shim 的实现是推测的，但它的接口签名（函数名、参数类型）大概率是准确的，因为调用方的代码是 A 级可信的。从接口反推实现，是逆向工程的核心方法论。

🔭 深水区（架构师选读）

Source Map V3 的编码机制

Source Map V3 的核心是 `mappings` 字段——一串用分号和逗号分隔的 VLQ（Variable-Length Quantity）编码字符串。每个分号代表生成文件（`cli.js`）的一行，每个逗号分隔的段（segment）包含 4-5 个 VLQ 编码数字：生成列号、源文件索引、源文件行号、源文件列号、以及可选的名称索引。

VLQ 编码使用 6-bit 分组和 Base64 字符集，将整数压缩为可变长度的字符串。例如，数字 0 编码为 `A`，数字 1 编码为 `C`，较大的数字需要多个字符。所有数值都是**相对于上一个段的增量**，这种差分编码大幅减小了文件体积。

理解这个格式后，你甚至可以写一个 Source Map 解析器：读取 `mappings`，逐字符解码 VLQ，重建原始文件和打包文件之间的完整映射关系。这也是验证还原质量的终极手段。

本章小结

一句话：Claude Code 的源码来自 Source Map 泄露（1,884 个原始文件）和社区补全（OpenClaudeCode，共 1,989 个文件），读源码时必须区分 A / B / C 三级可信度——这是贯穿全书的方法论基础。

关键源码索引

文件	职责	可信度
`cli.js.map`	59.8MB Source Map 文件	A
Source Map 还原（1,884 文件）	原始还原代码	A
OpenClaudeCode 根目录（1,989 文件）	社区补全代码	B/C
`shims/ant-claude-for-chrome-mcp/`	Chrome MCP 集成垫片	C
`shims/ant-computer-use-input/`	计算机操作输入垫片	C
`shims/ant-computer-use-mcp/`	计算机操作 MCP 垫片	C
`shims/ant-computer-use-swift/`	macOS 原生交互垫片	C
`shims/color-diff-napi/`	颜色差异对比垫片	C
`shims/modifiers-napi/`	键盘修饰键垫片	C
`shims/url-handler-napi/`	URL 协议处理垫片	C

逆向提醒

- ✅ **RELIABLE**：`sourcesContent` 中直接提取的 1,884 个文件内容——未经第三方修改
- ⚠️ **CAUTION**：OpenClaudeCode 新增的 105 个文件需要与 A 级代码交叉验证后使用
- ❌ **SHIM/STUB**：7 个 shim 文件为社区推测实现，接口签名可信但内部实现可能与真实代码不同

07

入门 导航 架构总览

第4章：源码全景地图：给 1,884 个文件画导航图

生活类比

第一次走进一座大城市，你需要一张地图。不需要记住每条街道的名字，只要知道"**商业区在东边、住宅区在西边、市政府在中心**"就能找到方向。1,884 个文件就是一座城市——本章就是那张地图。

这一章要回答的问题

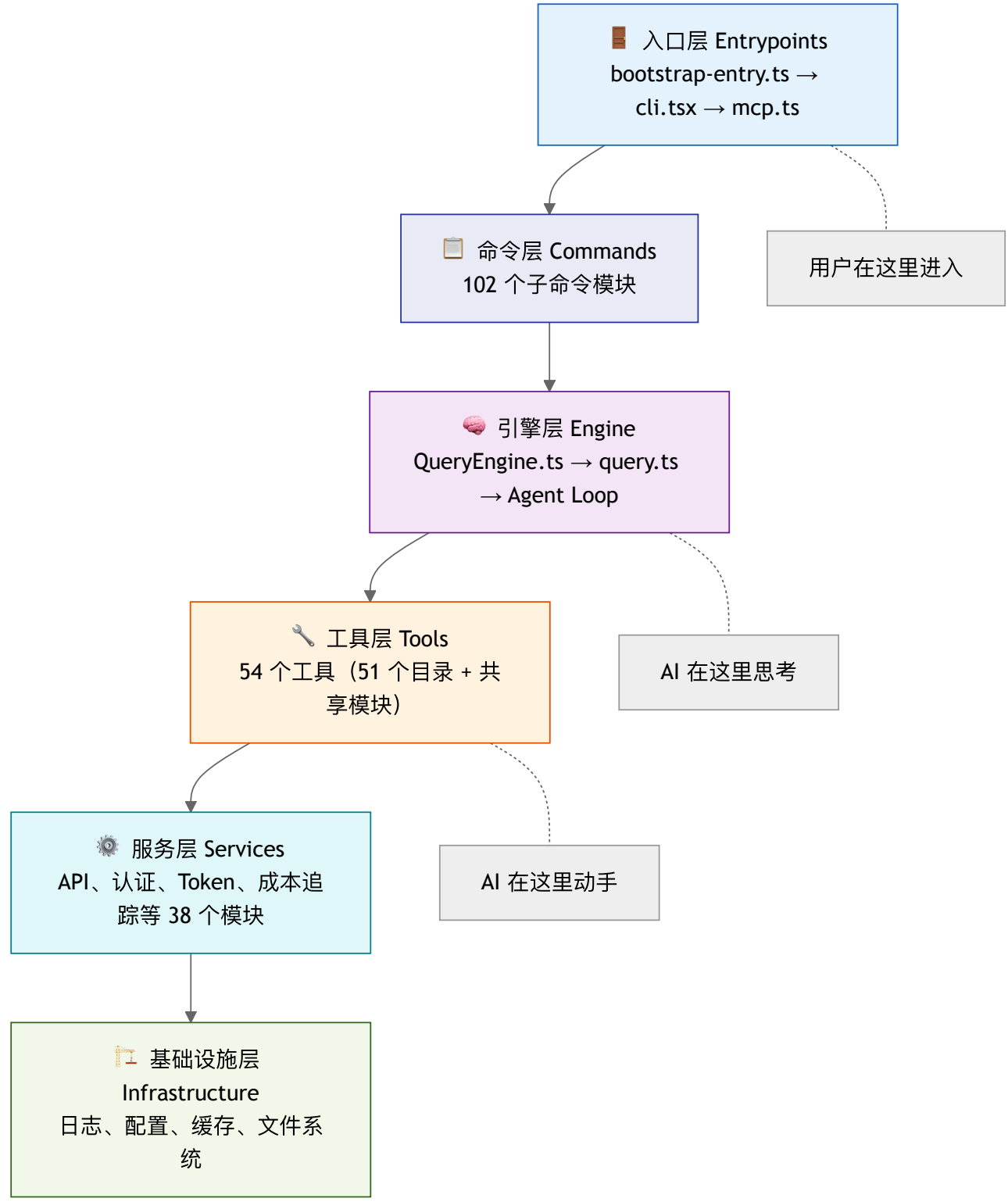
面对 1,884 个文件、512,664 行代码，应该从哪里开始读？怎么不迷路？

直接打开一个 50 万行的项目，很容易陷入细节迷宫——你可能花一小时读完一个文件，却发现它只是个工具函数。你需要先俯瞰全局——哪些是核心文件、哪些是辅助设施、各模块之间是什么关系。有了这张地图，再深入任何一个角落都不会迷失方向。

4.1 六大架构层次

从用户输入到 AI 响应的分层

打开 `src/` 目录，你会看到 38 个子目录和 22 个顶层文件。它们不是随意堆放的，而是遵循清晰的**六层架构**：



入口层（Entrypoints）——一切从这里开始

用户输入 `claude` 命令后，执行链从这里启动：

文件	行数	职责
`src/bootstrap-entry.ts`	5	最小启动入口，加载 bootstrap 模块
`src/entrypoints/cli.tsx`	302	CLI 命令路由与模式判断
`src/entrypoints/mcp.ts`	—	MCP 服务器模式入口
`src/entrypoints/init.ts`	—	项目初始化入口

入口层的文件数量极少，但地位最高——它们决定了用户的请求走哪条路径。

命令层（Commands）——102 个子命令

`src/commands/` 目录下有 **102 个模块**，每个处理一个具体的子命令：

- **核心交互**：chat、compact、resume
- **项目管理**：init、config、doctor
- **开发工具**：commit、review、ship
- **高级功能**：agents、mcp、bridge、tasks

所有子命令通过 `src/commands.ts`（754行）统一注册和分发。

引擎层（Engine）——AI 的"大脑"

文件	行数	职责
`src/QueryEngine.ts`	—	会话管理，维护对话上下文
`src/query.ts`	1,729	Agent Loop 核心——AI 的"思考循环"
`src/query/`	—	查询相关的辅助模块
`src/context.ts`	—	上下文注入，决定 AI"知道什么"

引擎层是整个系统最核心的部分——它决定了 AI **怎么思考**和**什么时候停下来**。

工具层（Tools）——54 个工具

`src/tools/` 目录下有 **51 个工具子目录** + 共享模块，加上 `src/Tool.ts`（792行）定义的统一接口，构成完整的工具系统：

```
src/tools/
├── BashTool/          # Shell 命令执行
├── FileReadTool/      # 文件读取
├── FileWriteTool/     # 文件写入
├── FileEditTool/      # 文件编辑
├── GrepTool/          # 内容搜索
├── GlobTool/          # 文件名搜索
├── WebFetchTool/      # 网页获取
├── AgentTool/         # 子 Agent 生成
├── MCPTool/           # MCP 工具调用
├── ...                # 还有 42 个工具
├── shared/            # 工具间共享代码
└── utils.ts           # 工具通用函数
```

服务层（Services）——基础通信

`src/services/` 目录下有 **38 个模块**，提供 API 通信、认证、遥测等基础服务：

- **API 客户端**：与 Claude API 通信
- **OAuth 认证**：用户身份验证
- **Token 估算**：计算消耗的 Token 数量
- **成本追踪**：`cost-tracker.ts` 记录每次请求的花费
- **MCP 服务**：Model Context Protocol 客户端
- **分析与遥测**：OpenTelemetry 集成

基础设施层（Infrastructure）——支撑一切的地基

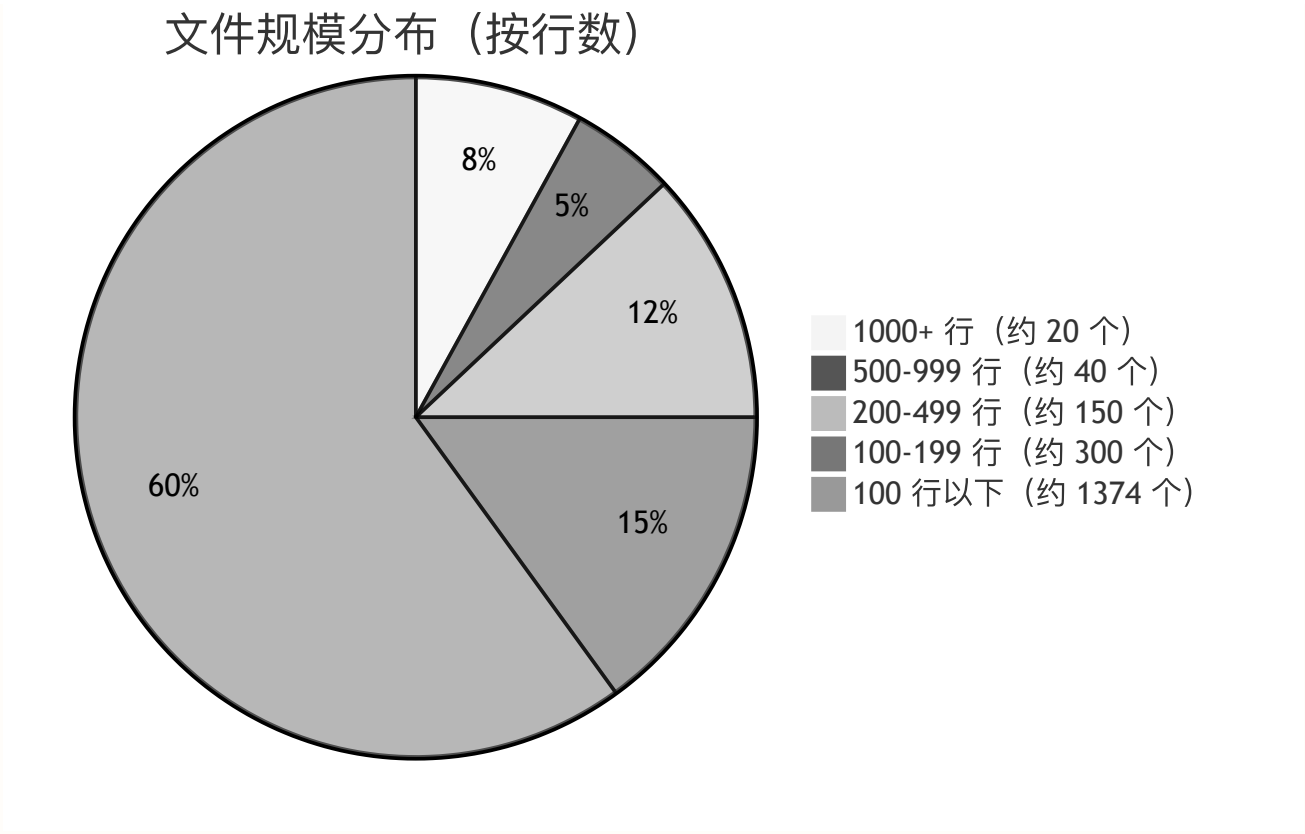
分散在多个目录中，不直接参与 AI 逻辑，但支撑整个系统运转：

目录	职责
`src/utils/`	通用工具函数
`src/constants/`	全局常量
`src/types/`	类型定义
`src/state/`	状态管理
`src/ink/`	自研终端渲染
`src/hooks/`	React Hooks
`src/schemas/`	数据校验 Schema
`src/migrations/`	数据迁移

4.2 关键文件 Top 20

文件规模分布

1,884 个文件的规模分布极不均匀——少数"巨型"文件承载了核心逻辑，大量"小"文件各司其职：



超过 **70%** 的文件不到 100 行——它们是螺丝钉，重要但不需要逐个研究。真正需要深入的是那 **Top 20** 文件。

超大文件（1000+ 行）——系统的支柱

排名	文件	行数	职责	对应章节
1	`src/main.tsx`	4,690	主协调器——启动、配置、认证、UI	第8章
2	`src/query.ts`	1,729	Agent Loop 核心	第11章
3	`src/Tool.ts`	792	工具统一接口	第14章
4	`src/commands.ts`	754	命令注册中心	第6章
5	`src/entrypoints/cli.tsx`	302	CLI 入口路由	第5章

不要忽略的"小"文件

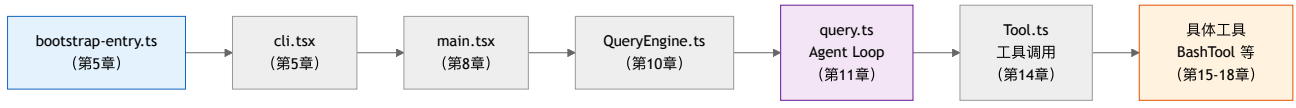
有些文件虽然行数少，但地位关键：

文件	行数	为什么重要
`src/bootstrap-entry.ts`	5	**一切的起点**——整条启动链从这里开始
`src/context.ts`	—	决定 AI"看到"什么信息
`src/cost-tracker.ts`	—	每次请求的 Token 和费用记录
`src/tasks.ts`	—	多 Agent 任务管理
`src/tools.ts`	—	工具注册清单

4.3 三条阅读路线

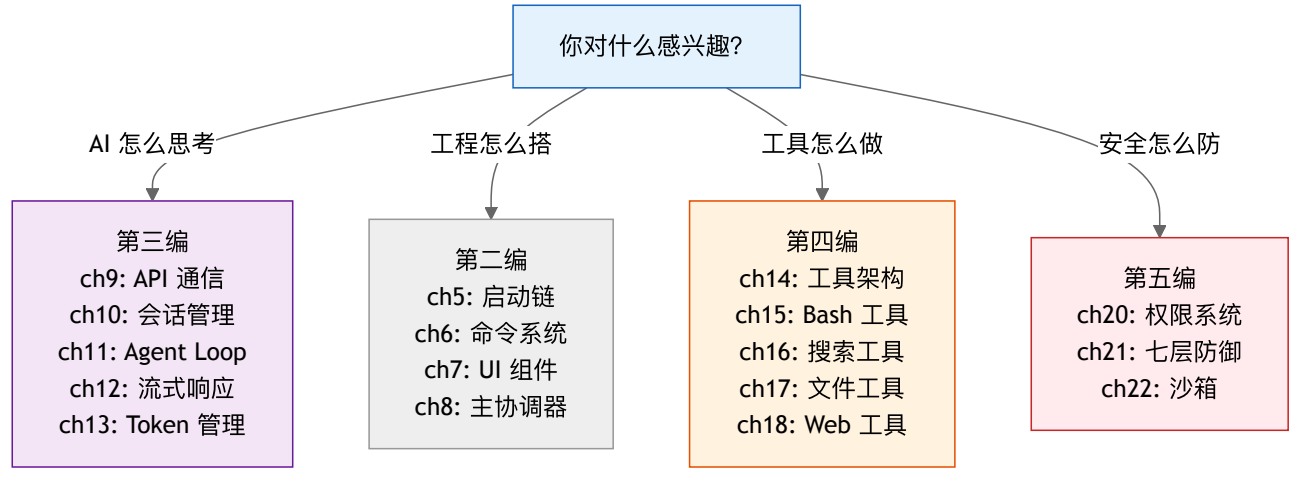
路线一：按执行链读（推荐新手）

跟着一条命令从输入到输出的完整旅程：



这条路线让你理解**一次请求的完整生命周期**——从用户按下回车到 AI 修改代码。

路线二：按兴趣跳读



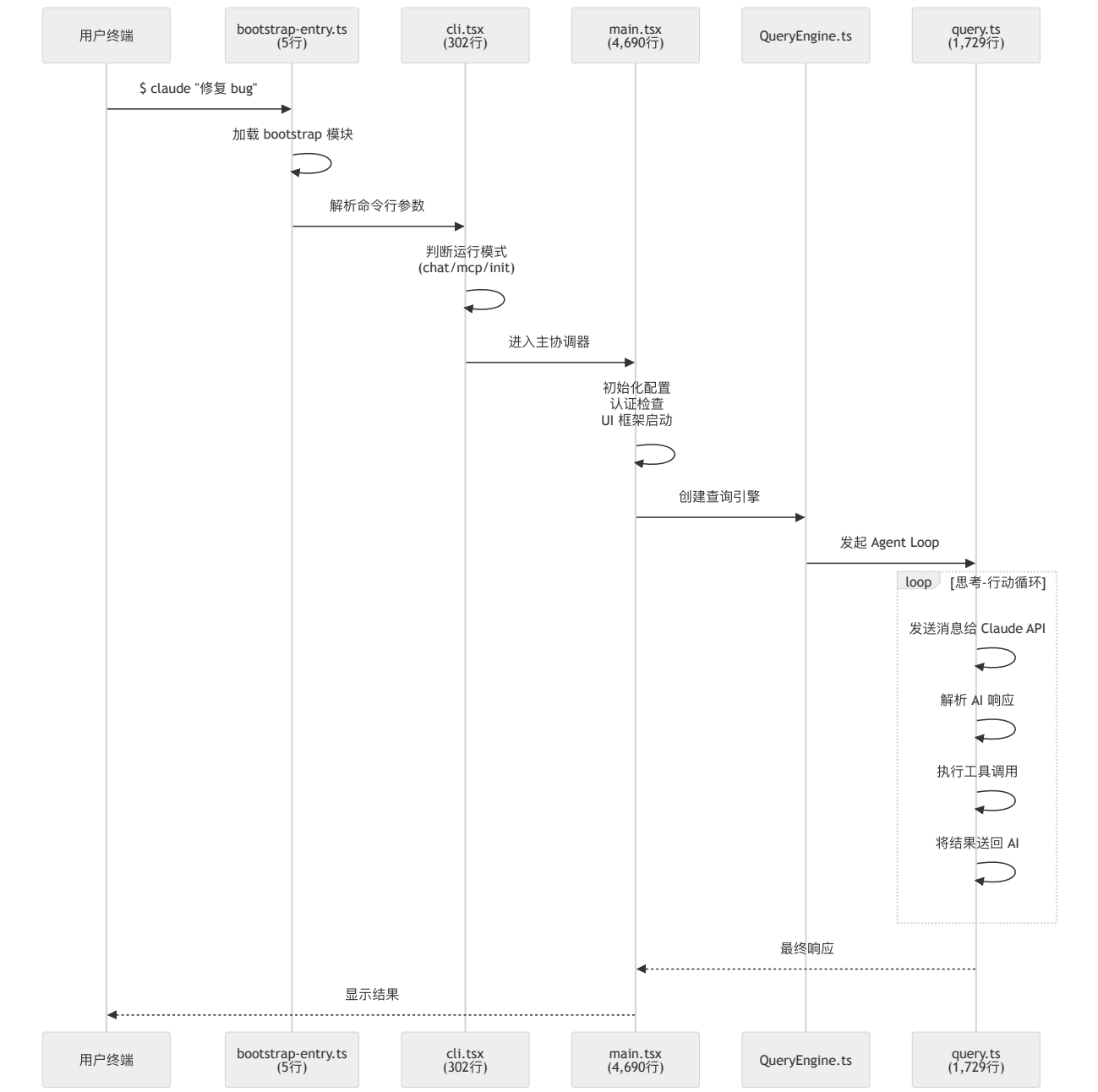
路线三：按编顺序读（最完整）

从第一编到最后一编，渐进式深入。每一编都在前一编的基础上展开：

编	主题	章节范围	核心关注
第一编	欢迎来到源码的世界	ch1–4	建立全局认知
第二编	启动与入口	ch5–8	代码怎么组织
第三编	AI 引擎	ch9–13	AI 怎么工作
第四编	工具系统	ch14–18	AI 怎么动手
第五编	安全与权限	ch19–22	怎么保证安全

启动链预览：从 5 行到 4,690 行

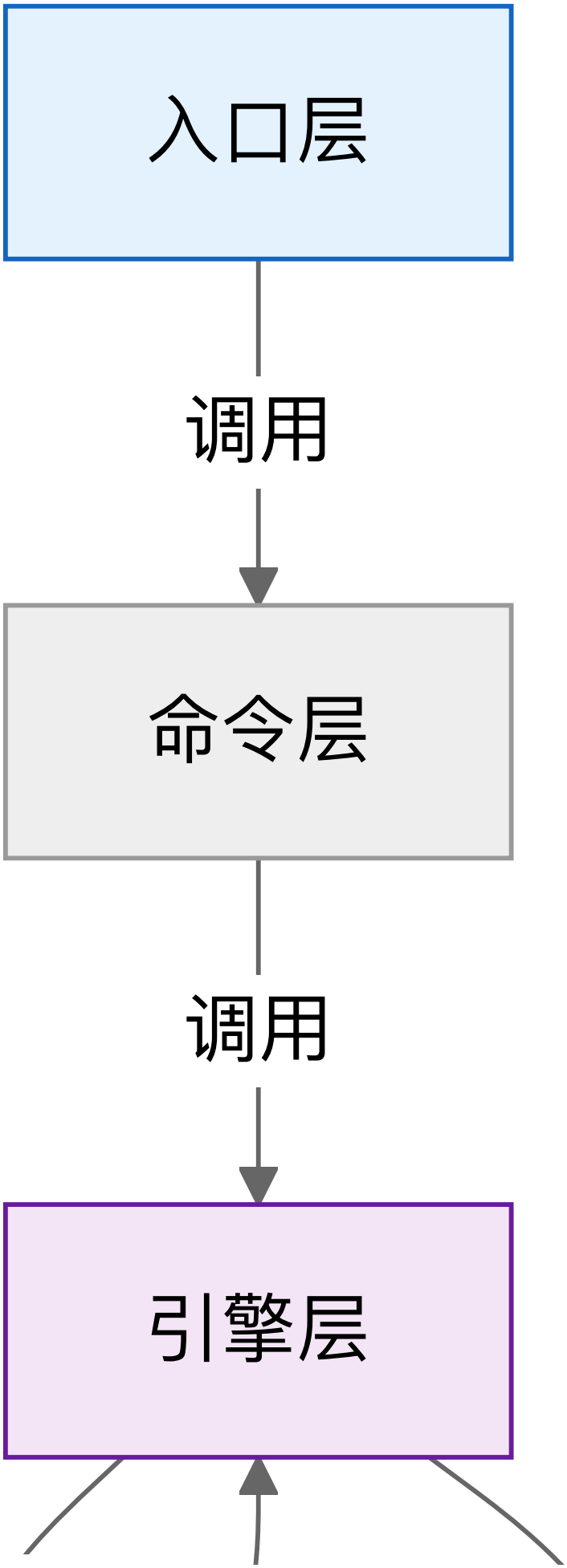
在深入后续章节之前，先看看 Claude Code 启动时的完整调用链：

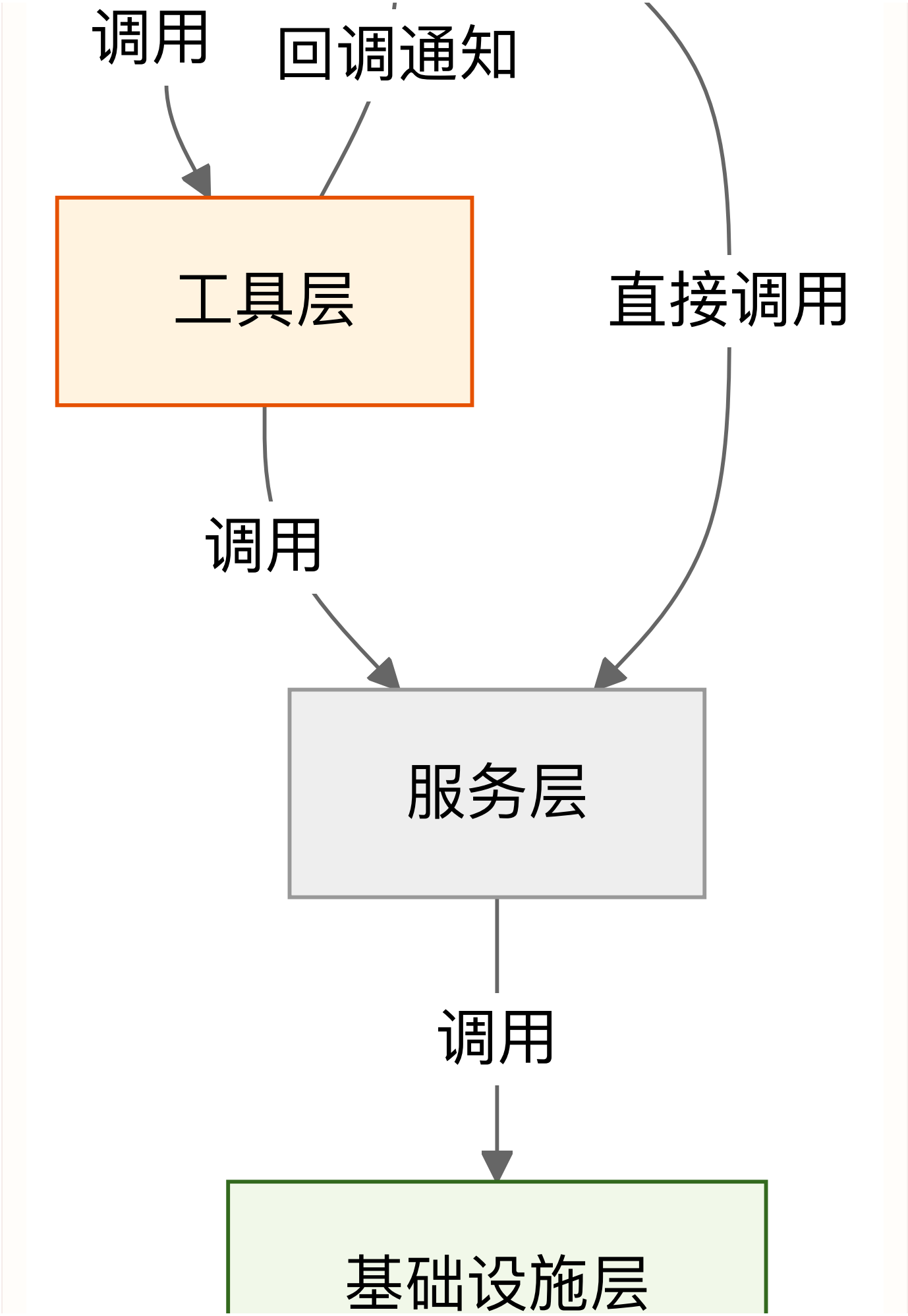


从 5 行的 `bootstrap-entry.ts` 开始，到 4,690 行的 `main.tsx`，再到 1,729 行的 `query.ts`——这条链路的细节将在第5章到第11章逐一展开。

模块依赖关系概览

六大层次之间的依赖关系遵循"**上层依赖下层，下层不依赖上层**"的原则：


入口层
调用
命令层
调用
引擎层


调用
回调通知
工具层
直接调用
调用
服务层
调用
基础设施层

注意 Tools → Engine 之间存在一条**反向箭头**（回调通知）——工具执行完毕后需要通知引擎。这不是"下层依赖上层"的违反，而是通过回调函数和事件机制实现的**控制反转**。这种设计在第14章会详细讨论。

◯ ◯ ◯

🌱 探索路径 🔧 实战路径 🏗️ 架构路径

记住六层架构的名字和职责就够了：**入口 → 命令 → 引擎 → 工具 → 服务 → 基础设施**。当后面章节提到某个文件时，你能迅速定位它在哪一层。

建议在 IDE 中打开 `src/` 目录，对照本章的 Top 20 文件列表，分别打开看看文件头部的 import 语句——这能帮你快速建立模块间的依赖关系感。

关注**层次边界的设计**：每层之间通过什么接口通信？`Tool` 接口的泛型参数如何保证类型安全？`QueryContext` 抽象如何解耦引擎和工具？这些接口设计是大型项目架构的精髓。

🔭 深水区（架构师选读）

循环依赖与接口隔离

大型 TypeScript 项目不可避免地会遇到模块间的循环依赖问题。Claude Code 通过严格的层次隔离来缓解——下层不依赖上层，同层通过**接口而非实现**通信。但在 tools 和 services 之间仍然存在一些"回调式"的依赖关系。

具体来说，`Tool` 接口定义了 `execute(input, context)` 方法，其中 `context` 参数来自引擎层——但 `Tool.ts` 本身不 import 引擎层的具体实现，而是依赖一个抽象的 `ToolContext` 类型。这就是**依赖倒置原则**的应用：高层模块和低层模块都依赖抽象，而非具体实现。

如果你想可视化整个项目的依赖图，可以使用 `madge` 工具：`npx madge --ts-config tsconfig.json src/ --image deps.svg`。但要做好心理准备——1,989 个文件的依赖图会非常壮观。

本章小结

一句话：1,884 个文件分布在六个架构层次中——入口、命令、引擎、工具、服务、基础设施。抓住 Top 20 关键文件和三条阅读路线（执行链 / 兴趣跳读 / 编顺序），就能不迷路地探索整个代码库。

关键源码索引

文件	职责	可信度
`src/`（38 子目录 + 22 顶层文件）	源码根目录	A
`src/main.tsx`	主协调器（4,690行）	A
`src/query.ts`	Agent Loop 核心（1,729行）	A
`src/Tool.ts`	工具统一接口（792行）	A
`src/commands.ts`	命令注册中心（754行）	A
`src/bootstrap-entry.ts`	5行启动入口	A
`src/entrypoints/cli.tsx`	CLI 命令路由（302行）	A
`src/tools/`（54 个工具）	工具实现目录	A
`src/services/`（38 个模块）	基础服务目录	A
`src/commands/`（102 个模块）	子命令目录	A

逆向提醒

- ✅ RELIABLE：目录结构和文件路径——Source Map 直接提供，完全可信
- ⚠️ CAUTION：文件行数统计基于还原版本，可能与 Anthropic 最新内部版本有差异
- ❌ SHIM/STUB：部分子目录下的 `index.ts` 文件可能是 OpenClaudeCode 补全，引用时需交叉验证

08

第二编：程序是怎么启动的

> *汽车点火不只是拧钥匙——从电路检测到发动机运转，每一步都不能少。*
>
> 本编解剖 Claude Code 从 `claude` 命令输入到 REPL 就绪的完整启动链条：**入口引导、多模式分发、状态初始化、终端渲染**。

本编总览

本编四章速览

章	标题	核心问题	生活类比
5	启动链条	从敲下命令到 REPL 就绪，中间经过几层?	汽车点火链条
6	多面手	同一个 npm 包为什么支持 10+ 种启动模式?	瑞士军刀
7	状态管理	Redux 太重、全局变量太乱——用什么管状态?	大脑的工作记忆
8	终端渲染	终端只有字符和颜色，为什么还用 React?	乐高搭控制面板

设计思想主线

本编建立的认知基础

1. 启动不是"打开就完了"——**配置加载、认证验证、UI 初始化、插件注册**都在启动链条中
2. 单一代码库管理多种模式（CLI / SDK / MCP Server）——**模式隔离**是关键
3. 自研轻量状态库代替 Redux——**够用就好**的工程哲学
4. 终端里的 React 是**重写后的 Ink 框架**——声明式 UI 不只属于浏览器

推荐路径

○ ○ ○

🌱 初学者🔧 开发者🏗️ 架构师

从第5章的生活类比开始，理解"启动链条"的概念。**第8章的终端渲染最有趣**——原来终端也能用 React！

重点看第6章的多模式架构和第7章的状态管理。**自研 Store 的设计值得学习**。

关注第6章的模式隔离策略和第8章的 Ink 重写决策——**什么时候该用社区方案、什么时候该自研**。

本编阅读目标

读完这一编，你会把 Claude Code 的启动过程看成一条清晰的工程链：入口分流、模式选择、状态初始化、终端渲染，各自解决不同问题，又在启动阶段紧密咬合。

09

启动链 第二编

第5章：启动链条：从敲下命令到屏幕亮起

生活类比

你按下汽车启动键时，真正发生的不是“发动机突然活了”，而是一串精密动作：电瓶供电、自检、喷油、点火、仪表盘亮起。`claude` 启动也一样，**看起来只是一条命令，背后却是一条分层启动链。**

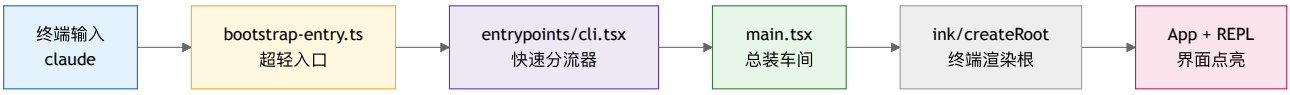
这一章要回答的问题

从你在终端输入 `claude` 到 REPL 提示符出现，中间到底经过了几层？为什么不能把所有逻辑都塞进一个文件里？

如果你只看表面，会以为 Claude Code 的启动过程就是“读参数，然后显示界面”。但源码告诉我们，真实情况是：**超轻入口层负责抢速度，命令分流层负责挑路线，主协调器负责装配上下文，渲染层最后把界面点亮。**

5.1 一条命令，四层启动链

先别急着看细节，先记住全景地图：



层级	代表文件	干什么	为什么单独拆出来
第1层	`src/bootstrap-entry.ts`	最小入口，先做宏初始化	尽量让冷启动更快
第2层	`src/entrypoints/cli.tsx`	检查快速路径、避免重载整个 CLI	`--version` 这类命令不该拖着 4000 多行主文件一起启动
第3层	`src/main.tsx`	解析参数、加载配置、搭建上下文、决定进入哪种运行模式	这里是总协调器
第4层	`src/ink.ts` + `src/ink/root.ts` + `src/components/App.tsx`	创建终端渲染根，把 REPL 真正挂载上去	把“业务逻辑”和“怎么画到终端”隔开

这套拆分有一个很重要的设计思想：

设计思想

启动链不是按“代码模块”拆，而是按“时间敏感度”拆。

越早执行的部分越轻，越重的部分越往后推。这和网页里的“首屏优化”是同一个思路。

5.2 第一跳：`bootstrap-entry.ts` 只有 5 行

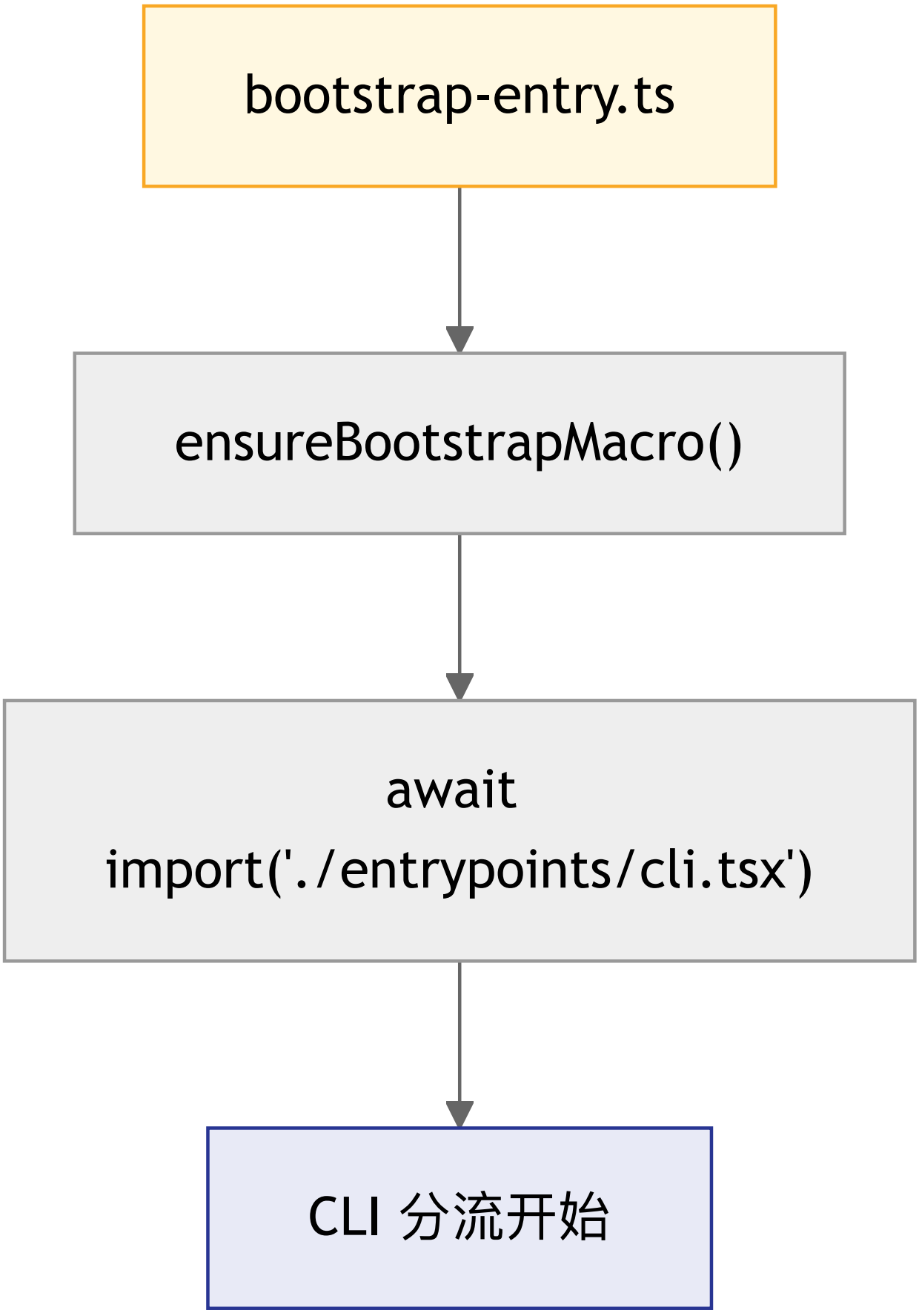
在 `OpenClaudeCode/src/bootstrap-entry.ts` 里，真正的入口短得惊人：

```
import { ensureBootstrapMacro } from './bootstrapMacro'

ensureBootstrapMacro()

await import('./entrypoints/cli.tsx')
```

这 5 行说明了两件事：

1. **先做宏初始化，再进入 CLI**
2. **CLI 用动态导入，而不是顶层静态导入**

动态导入的意义是什么？很像开商场时，先打开总电源，不会一上来就把所有店铺的灯都打开。只有当你真的走到那一层，才按需通电。


bootstrap-entry.ts
ensureBootstrapMacro()
await import('./entrypoints/cli.tsx')
CLI 分流开始

为什么入口越短越好

命令行工具和网页首页有一个共同点：**第一次响应非常重要**。用户打出 `claude --version`，如果也要等一大堆 React、Ink、插件、认证逻辑初始化完，体验会非常差。

所以 Claude Code 的第一跳几乎不做业务，只做“把舞台搭好”。

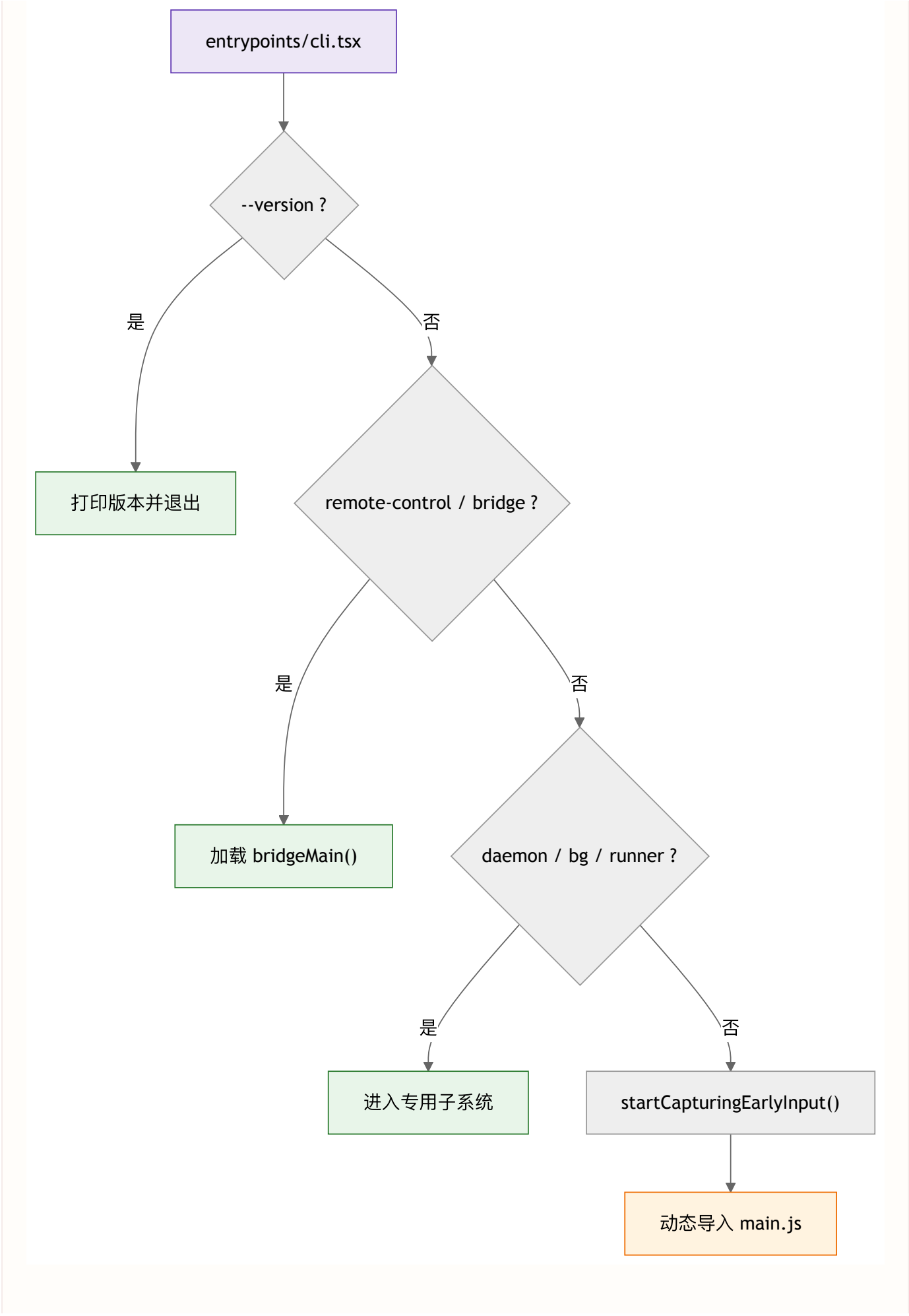
5.3 第二跳：`entrypoints/cli.tsx` 是“高速收费站”

真正聪明的地方出现在 `claude-code-sourcemap/restored-src/src/entrypoints/cli.tsx`。这个文件不是完整主程序，而是一个**快速分流器**。

先拦下那些根本不需要完整启动的命令

源码里最典型的几个快速路径：

- `--version` / `-v`：直接打印版本，立即退出
- `remote-control`：直接走桥接模式
- `daemon` / `--daemon-worker`：直接进入守护进程路径
- `ps` / `logs` / `attach` / `kill` / `--bg`：直接走后台会话管理
- `environment-runner` / `self-hosted-runner`：直接进入无头运行器
- `--worktree --tmux`：优先处理 tmux/worktree 特殊分支


entrypoints/cli.tsx
--version ?
是
否
打印版本并退出
remote-control / bridge ?
是
否
加载 bridgeMain()
daemon / bg / runner ?
是
否
进入专用子系统
startCapturingEarlyInput()
动态导入 main.js

一个很重要的性能技巧：按需动态导入

在 `entrypoints/cli.tsx` 中，几乎每条快速路径都写成这样：

```
const { bridgeMain } = await import('../bridge/bridgeMain.js')
await bridgeMain(args.slice(1))
return
```

意思是：

- 你没走桥接模式，就**完全不用加载**桥接代码
- 你没走后台模式，就**完全不用加载**后台会话管理代码
- 你只是查版本，就连 `main.tsx` 都不用读

这就是“把重模块推迟到真正需要的那一刻”。

真正的完整启动只在最后发生

分流器的尾声非常关键：

```
const { startCapturingEarlyInput } = await import('../utils/earlyInput.js')
startCapturingEarlyInput()

const { main: cliMain } = await import('../main.js')
await cliMain()
```

在这一刻，Claude Code 才真正决定：“好，现在确实需要完整主程序了。”

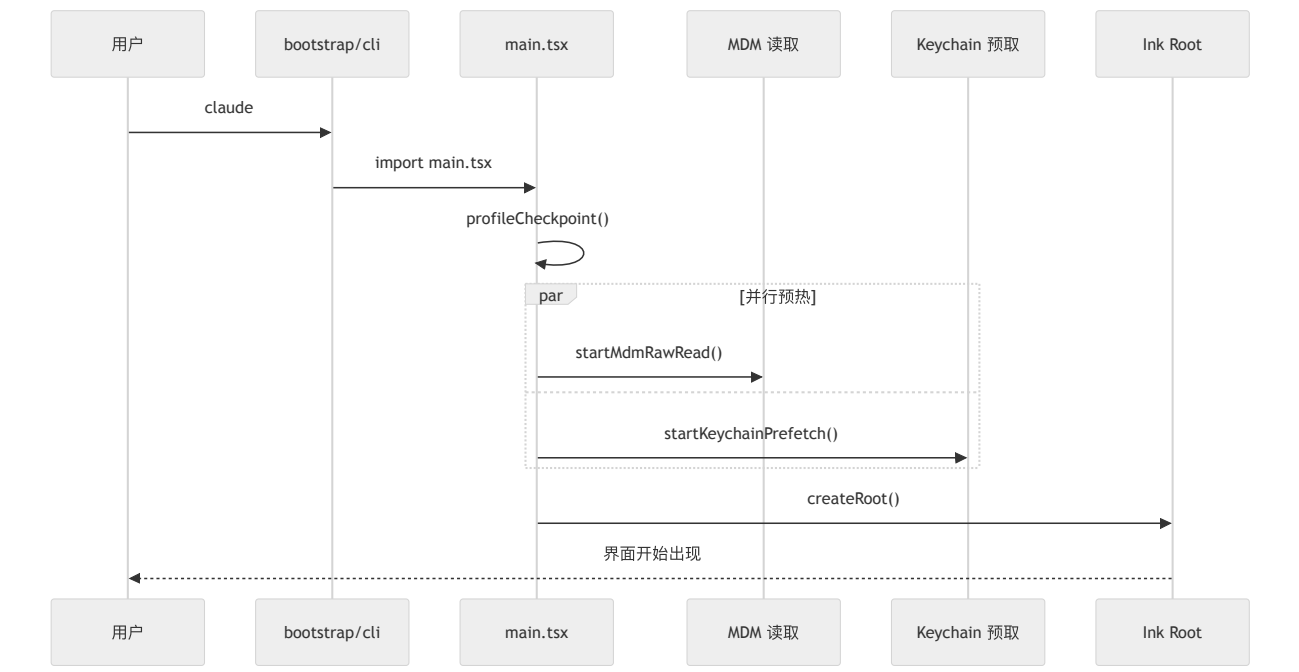
5.4 第三跳：`main.tsx` 是总装车间

如果说 `entrypoints/cli.tsx` 是收费站，`main.tsx` 就是整车总装厂。它不再只负责“分流”，而是开始做真正昂贵但必要的准备工作。

一上来先做并行预热

`OpenClaudeCode/src/main.tsx` 的开头注释写得非常直白：某些副作用必须在最早时刻并行发起。比如：

- `profileCheckpoint('main_tsx_entry')`：启动性能打点
- `startMdmRawRead()`：发起 MDM 配置读取
- `startKeychainPrefetch()`：预取钥匙串里的认证信息



你可以把它理解成饭店后厨：客人还没点单，厨师就先把高汤热上、刀具摆好、常用食材拿出来。这样真正开始做菜时就不会手忙脚乱。

为什么这些副作用要“顶层启动”

源码注释甚至给出了性能解释：

- MDM 读取与后续导入并行，可以节省大约 100ms 量级等待

- Keychain 预取能避免后面同步读取认证信息拖慢启动

这说明 Claude Code 团队并不是“感觉慢”，而是在**用数据驱动启动优化**。

5.5 只有交互模式，才真的创建 Ink 根节点

`main.tsx` 里有一个非常值得初学者记住的判断：

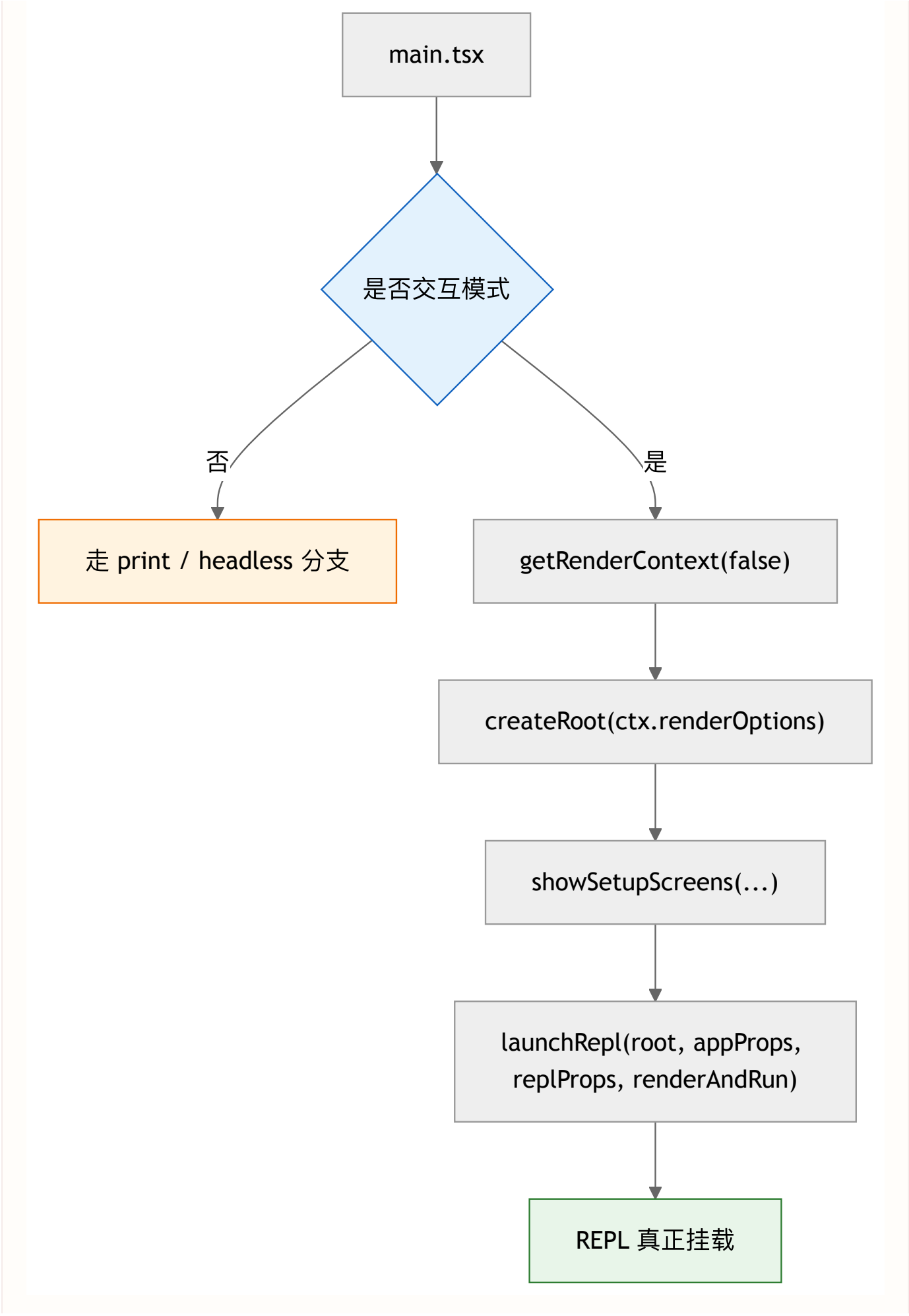
> Ink root is only needed for interactive sessions.

翻成人话就是：**只有你真的要进入交互 REPL，才创建终端 UI 根节点。**

交互模式的关键动作

在 `main.tsx` 的交互路径里，重要步骤大致是：

1. 获取渲染上下文 `getRenderContext(false)`
2. 动态导入 `createRoot`
3. `root = await createRoot(ctx.renderOptions)`
4. 显示 trust / onboarding / setup screens
5. 最后 `launchRepl(...)`


main.tsx
是否交互模式
否
是
走 print / headless 分支
getRenderContext(false)
createRoot(ctx.renderOptions)
showSetupScreens(...)
launchRepl(root, appProps, replProps, renderAndRun)
REPL 真正挂载

为什么这么设计？

- `--print` 模式不需要复杂 UI
- 无头模式如果也 patch console、挂 Ink，会干扰标准输出
- 交互模式则必须要完整 UI、快捷键、标题栏、状态栏等能力

也就是说，Claude Code 启动的目标不是“永远把所有系统都启动起来”，而是“**根据当前模式只启动需要的那部分**”。

5.6 第四跳：`launchRepl()` 把 App 和 REPL 组装起来

在 `OpenClaudeCode/src/replLauncher.tsx` 里，`launchRepl()` 的职责非常清楚：

```
const { App } = await import('./components/App.js')
const { REPL } = await import('./screens/REPL.js')

await renderAndRun(
  root,
  <App {...appProps}>
    <REPL {...replProps} />
  </App>,
)
```

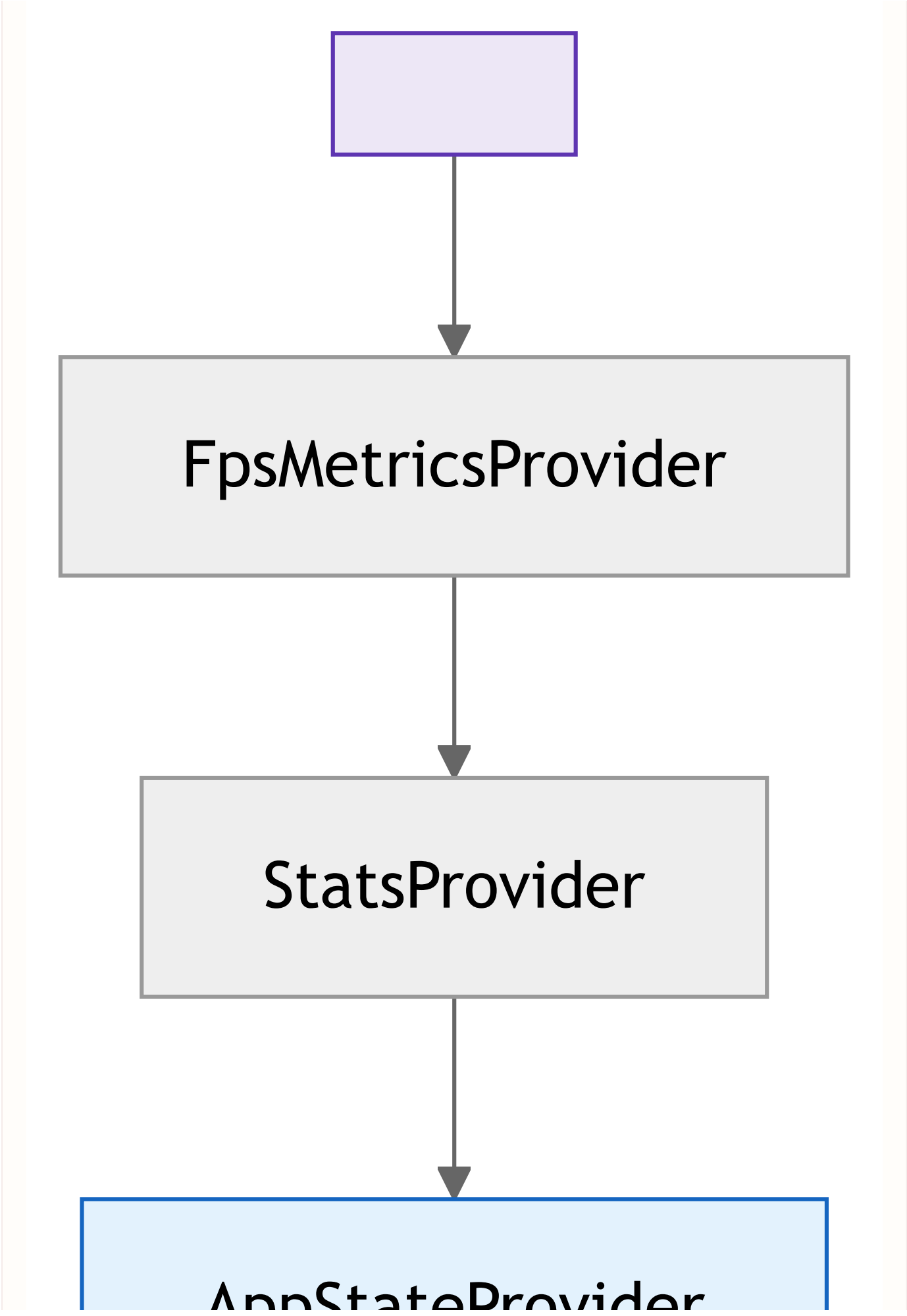
这段代码像在搭舞台：

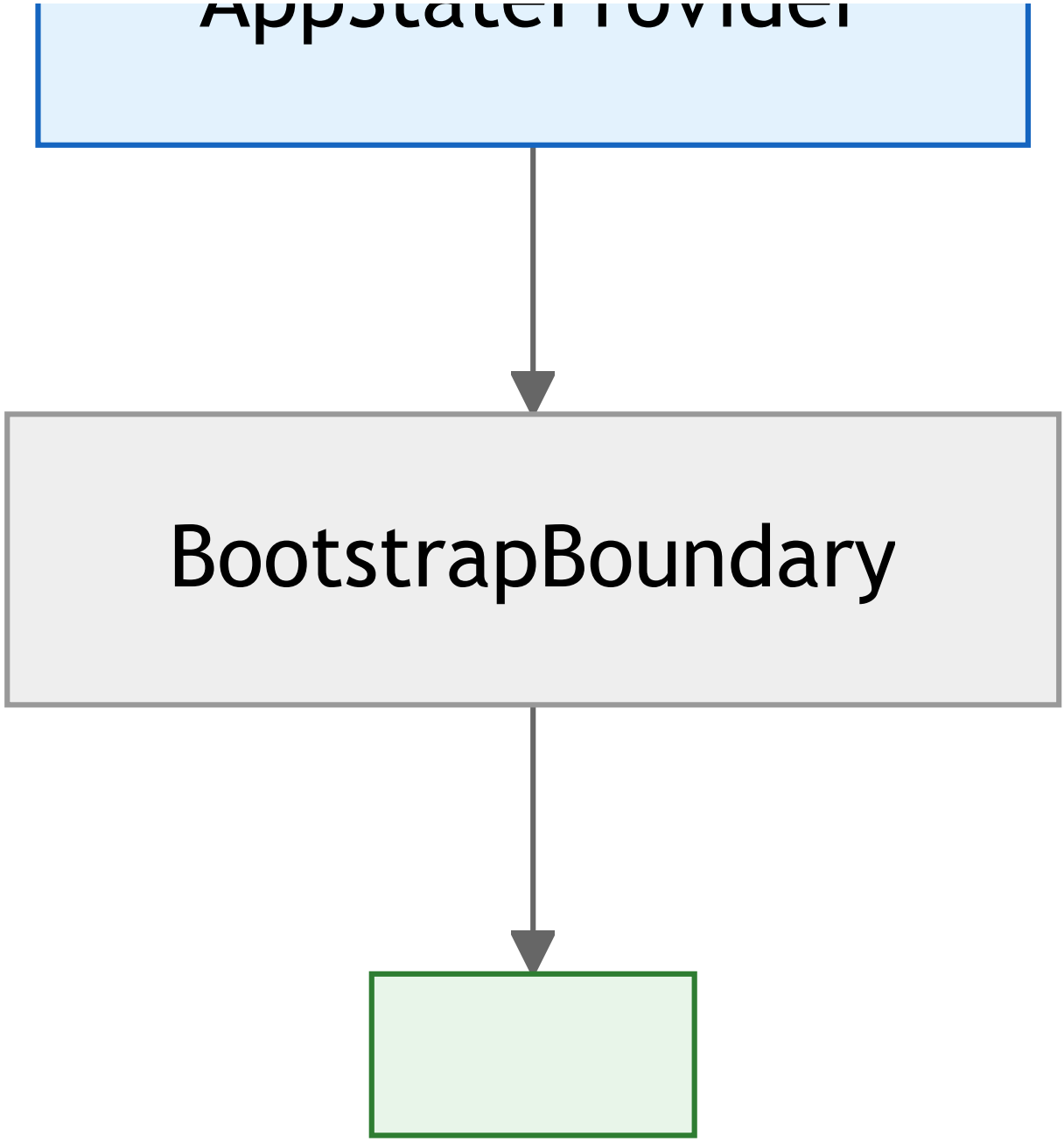
- App 是总外壳，提供上下文
- REPL 是真正的交互主界面
- renderAndRun 负责把整棵组件树挂到终端

App 不是摆设，它负责注入上下文

`components/App.tsx` 里，App 会套上这些提供器：

- FpsMetricsProvider
- StatsProvider
- AppStateProvider
- BootstrapBoundary


FpsMetricsProvider
StatsProvider



这很像玩游戏时加载场景：

- 先把引擎挂上
- 再把全局状态挂上
- 再放一个“出错保护罩”
- 最后才把真正的场景内容放进去

5.7 为什么不直接一个 `main()` 写到底

这是本章最值得记住的工程思想。

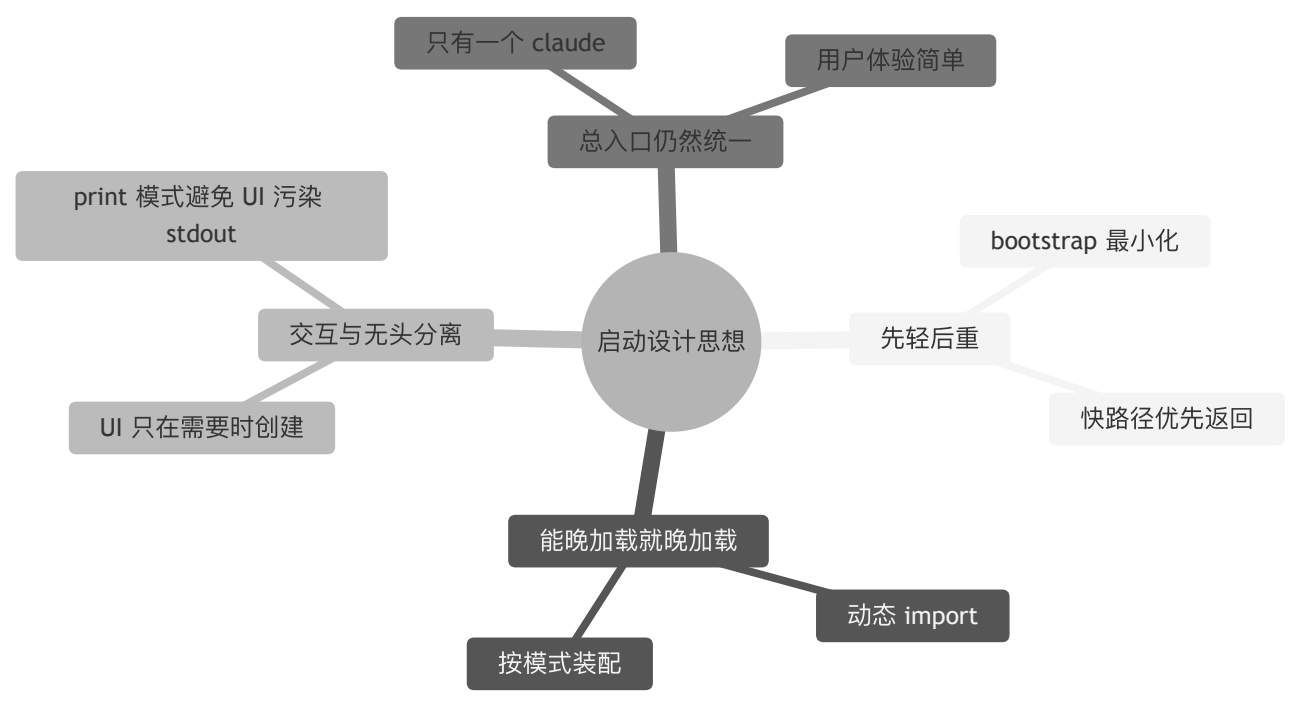
如果你是第一次写 CLI，很可能会这样做：

```
main()
  解析参数
  初始化认证
  初始化配置
  初始化 UI
  初始化插件
  创建 REPL
```

这样的问题是：

- `--version` 也会被迫跑一大堆没用逻辑
- 任何一个功能变重，所有模式一起变慢
- 交互模式和无头模式会互相拖累
- 每加一种模式，`main()` 都会继续膨胀

Claude Code 的做法更像高速公路立交桥：



这套架构兼顾了两件看似冲突的事：

- 对用户来说，入口只有一个：`claude`
- 对内部实现来说，路径可以非常分层、非常精细

深水区（架构师选读）

Claude Code 的启动链本质上是一套“冷启动预算管理”方案。

`bootstrap-entry.ts` 极小化、`entrypoints/cli.tsx` 大量动态导入、`main.tsx` 早期并行预热、Ink 根节点只在交互模式创建，这四个决策都在服务同一件事：**让用户尽快看到第一个有效反馈，同时不把所有模式绑死在同一条最慢路径上。**

很多团队做 CLI 会先把功能做对，再慢慢优化性能；Claude Code 的源码明显表现出另一种成熟度：**它从一开始就在把“启动速度”视为架构问题，而不是后期小修小补。**

本章小结

一句话：Claude Code 的启动不是“一口气跑到头”，而是经过 `bootstrap-entry`、`cli` 分流、`main` 总装、`Ink/REPL` 挂载四层链条，层层只做当下最该做的事。

关键源码索引

证据层	文件	本章关注点
还原层	`claude-code-sourcemap/restored-src/src/entrypoints/cli.tsx:33-42`	`--version` 快速路径
还原层	`claude-code-sourcemap/restored-src/src/entrypoints/cli.tsx:108-160`	bridge 快速路径
还原层	`claude-code-sourcemap/restored-src/src/entrypoints/cli.tsx:287-298`	最终导入 `main.js`
补全层	`OpenClaudeCode/src/bootstrap-entry.ts:1-5`	极简启动入口
补全层	`OpenClaudeCode/src/main.tsx:1-20`	顶层并行预热
补全层	`OpenClaudeCode/src/main.tsx:2217-2248`	交互模式创建 Ink 根节点
补全层	`OpenClaudeCode/src/replLauncher.tsx:12-21`	挂载 App 与 REPL
补全层	`OpenClaudeCode/src/components/App.tsx:46-95`	注入 AppState / Stats / FPS 上下文

逆向提醒

- ✅ **可信度高**：启动分层、动态导入、快路径分流，这些在还原层和补全层都能互相印证
- ⚠️ **需留意差异**：`OpenClaudeCode` 为了可运行性补齐了部分外围依赖，但启动骨架本身与还原层高度一致
- ❌ **不要误读**：某些 ant-only 或 feature flag 分支不等于所有发行版本都会启用

10

模式矩阵 第二编

第6章：多面手：同一个程序的10种模式

生活类比

瑞士军刀只有一个手柄，却能翻出刀、剪刀、开瓶器、锉刀。Claude Code 也是这样：用户看到的只是一个 `claude`，但它背后能翻出很多不同“刀片”。

这一章要回答的问题

为什么同一个程序要支持这么多模式？这些模式为什么不会互相绊脚？

交互 REPL、`--print` 无头模式、远程桥接、守护进程、后台会话、插件管理、认证管理……如果这些能力都做成独立二进制，用户会记不住；如果全部塞进一个大函数，工程师会崩溃。Claude Code 的做法，是把“**一个入口**”和“**多条执行路线**”同时保住。

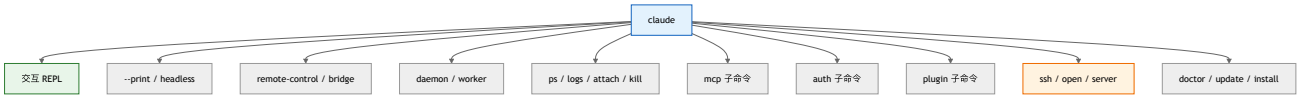
6.1 先建立一个认知：`claude` 不是一种模式，而是一个总入口

很多初学者会误以为：

> `claude` = 终端聊天界面

其实不是。更准确的说法是：

> `claude` = 一套命令分发平台，REPL 只是其中最常见的一种运行方式



这就是本章的主角：**模式矩阵**。

6.2 第一层分流：入口文件先拦截“高频快路径”

`entrypoints/cli.tsx` 不是完整 CLI，而是一个**轻量总闸门**。它先判断你是不是走某些特定路径，如果是，就直接导入专用模块；如果不是，再进入 `main.tsx`。

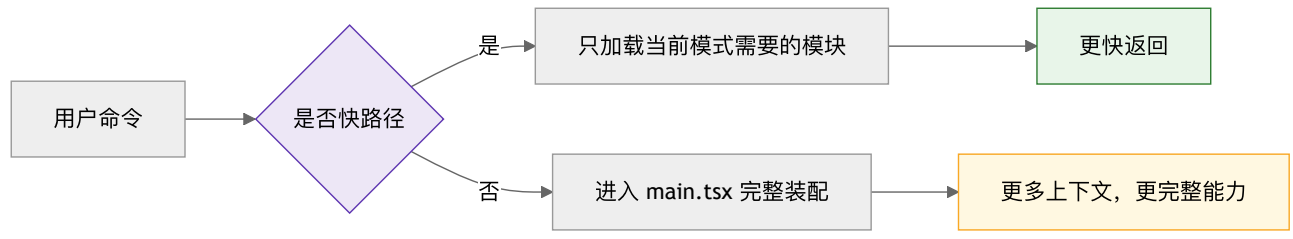
从源码里能数出至少 10 类代表性模式

下面这张表不是“所有命令的完整清单”，而是最值得理解的 10 类模式：

模式	触发方式	主要入口证据	设计意图
1. 交互 REPL	`claude`	`main.tsx` action handler + `launchRepl()`	默认主路径
2. 无头输出	`claude -p` / `--print`	`main.tsx` headless 分支	给脚本、管道、SDK 用
3. 版本查询	`--version`	`entrypoints/cli.tsx:36-42`	零重载快速返回
4. 远程桥接	`remote-control` / `bridge`	`entrypoints/cli.tsx:108-160`	本地 CLI 变成桥接器
5. 守护进程	`daemon`	`entrypoints/cli.tsx:164-179`	后台长驻服务
6. 后台会话	`ps / logs / attach / kill / --bg`	`entrypoints/cli.tsx:182-208`	管理分离会话
7. 模板任务	`new / list / reply`	`entrypoints/cli.tsx:211-221`	模板工作流专门路径
8. 环境运行器	`environment-runner`	`entrypoints/cli.tsx:224-232`	无头执行环境
9. 自托管运行器	`self-hosted-runner`	`entrypoints/cli.tsx:235-244`	对接自托管服务
10. 远程会话族	`ssh` / `open` / `server`	`main.tsx:3968+`, `4052+`, `4065+`	本地 UI 连接远程执行

为什么先在入口层分流，而不是都等进 `main.tsx`

原因很简单：不是每条命令都值得付出完整启动成本。



这和大型网站的路由网关很像：健康检查、静态资源、API 网关、完整页面渲染，不会走同一条最重路径。

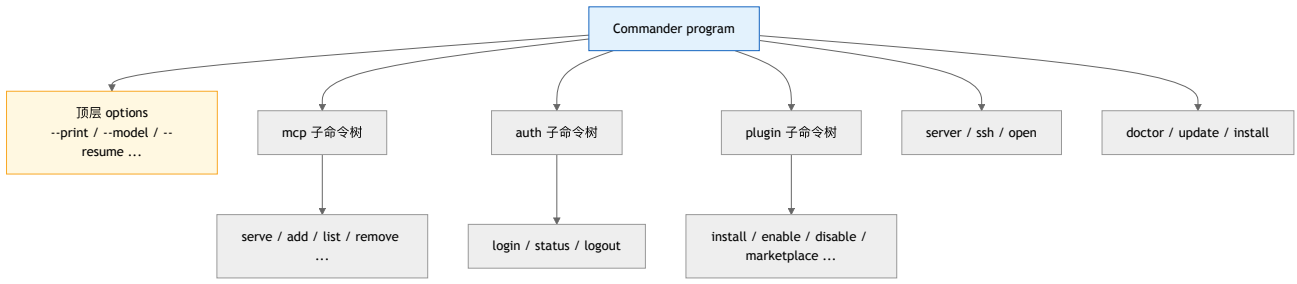
6.3 第二层分流：`main.tsx` 里的 Commander 像“总控面板”

进入 `main.tsx` 后，Claude Code 用 `CommanderCommand` 建立完整命令系统：

```
const program = new CommanderCommand()
  .configureHelp(createSortedHelpConfig())
  .enablePositionalOptions()
```

然后围绕这个 `program` 挂上：

- 顶层选项：`--print`、`--model`、`--permission-mode`、`--settings` 等
- 子命令树：`mcp`、`auth`、`plugin`、`server`、`ssh`、`open`、`doctor`、`update`……



这意味着什么

进入 `main.tsx` 后，Claude Code 的模式切换不再只是“读 argv 然后 if-else”，而是进入一套正式的命令框架。

好处有三个：

1. **帮助文档自动生成**
2. **参数校验集中管理**
3. **模式的边界更清楚**

这对大型 CLI 非常重要。命令一多，如果没有 Commander 这类结构化框架，很快就会变成一团乱麻。

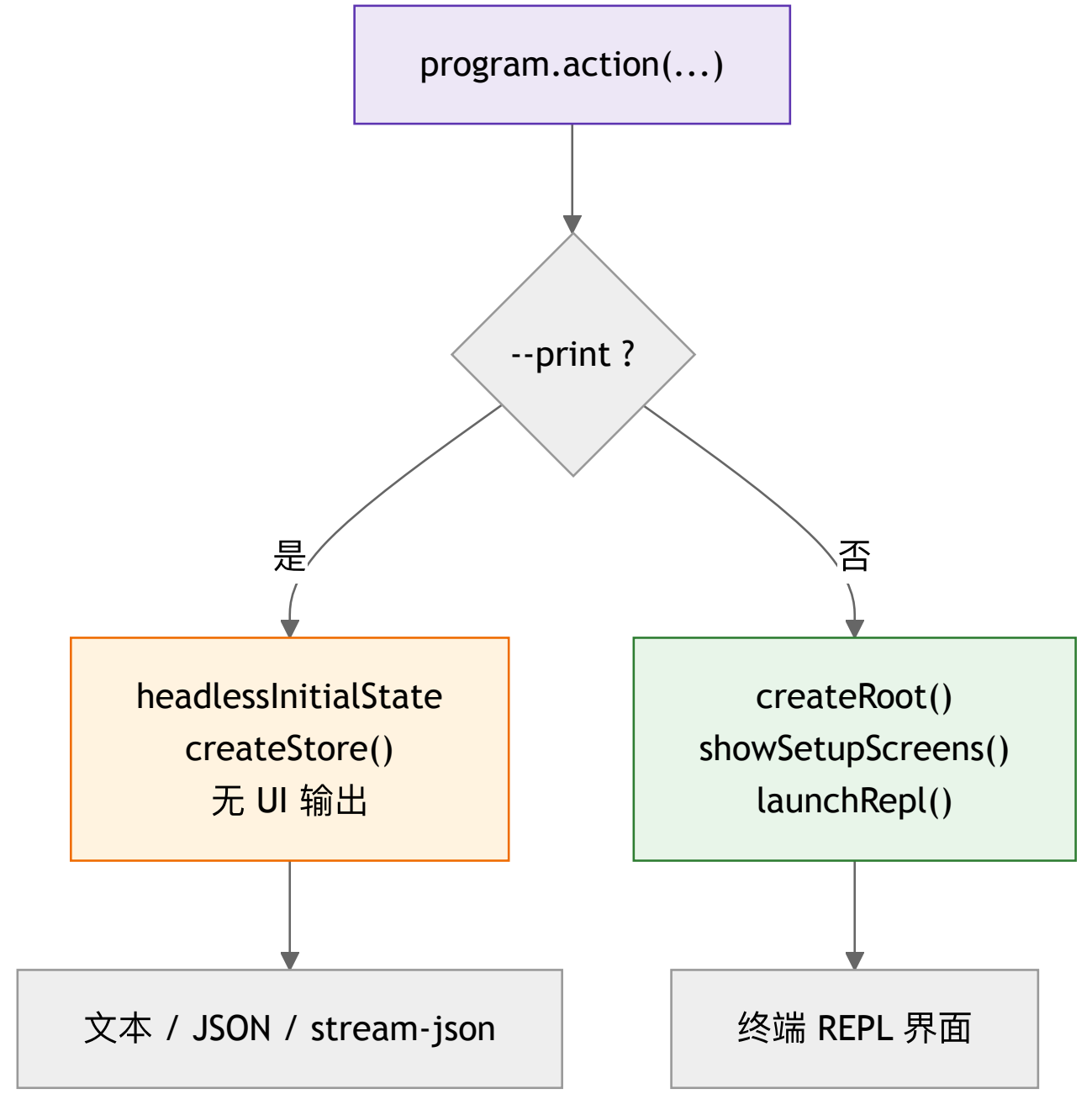
6.4 第三层分流：同一个 action handler，继续拆成交互与无头

即使到了顶层 `program.action(async (prompt, options) => { ... })`，Claude Code 仍然不会“一条路跑到底”。

同一套参数，不同的运行引擎

最核心的一刀是：

- **交互模式**：创建 Ink root，最后 `launchRepl()`
- **无头模式**：创建 `headlessStore`，直接跑 headless/print 流程



为什么 `--print` 不复用 REPL

因为二者的目标完全不同：

模式	最优先目标
交互 REPL	给人看，支持输入、重绘、快捷键、状态栏
`--print`	给程序看，输出稳定、可解析、不要污染 stdout

如果强行让 `--print` 走 REPL，那会遇到很多问题：

- Ink patch console 会影响标准输出
- ANSI 样式会污染 JSON
- 状态栏、提示框、标题栏都成了噪声

所以 Claude Code 明确分成两套运行体验，但又尽量让它们共享同一套核心状态和查询逻辑。

6.5 远程模式并不是“另一个产品”，而是同一个入口的分支

很多工具一旦加远程能力，就会分裂成另一套客户端。Claude Code 这里做得更优雅：远程能力仍然是 `claude` 这一个入口上的不同分支。

远程相关的几条主要路径

路径	作用	用户感受
`remote-control`	本地机器作为 bridge 环境	像把 CLI 变成桥
`server`	启动 Claude Code 会话服务器	像把 CLI 变成服务端
`open <cc-url>`	连接已有会话服务器	像远程登录
`ssh <host> [dir]`	通过 SSH 在远端运行 Claude Code	UI 在本地，执行在远端

这一点特别能体现 Claude Code 的产品哲学：

设计思想

它没有把“本地模式”和“远程模式”做成两个产品，而是把它们视为**同一任务执行框架在不同运行环境中的展开形式**。

这比“再做一个远程版客户端”更难，但也更统一。

6.6 多模式如何避免互相打架

这是很多初学者最关心的问题：一个程序支持这么多模式，不会越改越乱吗？

Claude Code 的答案主要有三点：

1. 先按层分流，再按模式装配

- 入口层先拦快路径
- 主程序层再挂子命令
- action handler 再做交互/无头拆分
- 真正执行前再按模式拼装上下文

2. 重模块尽量动态导入

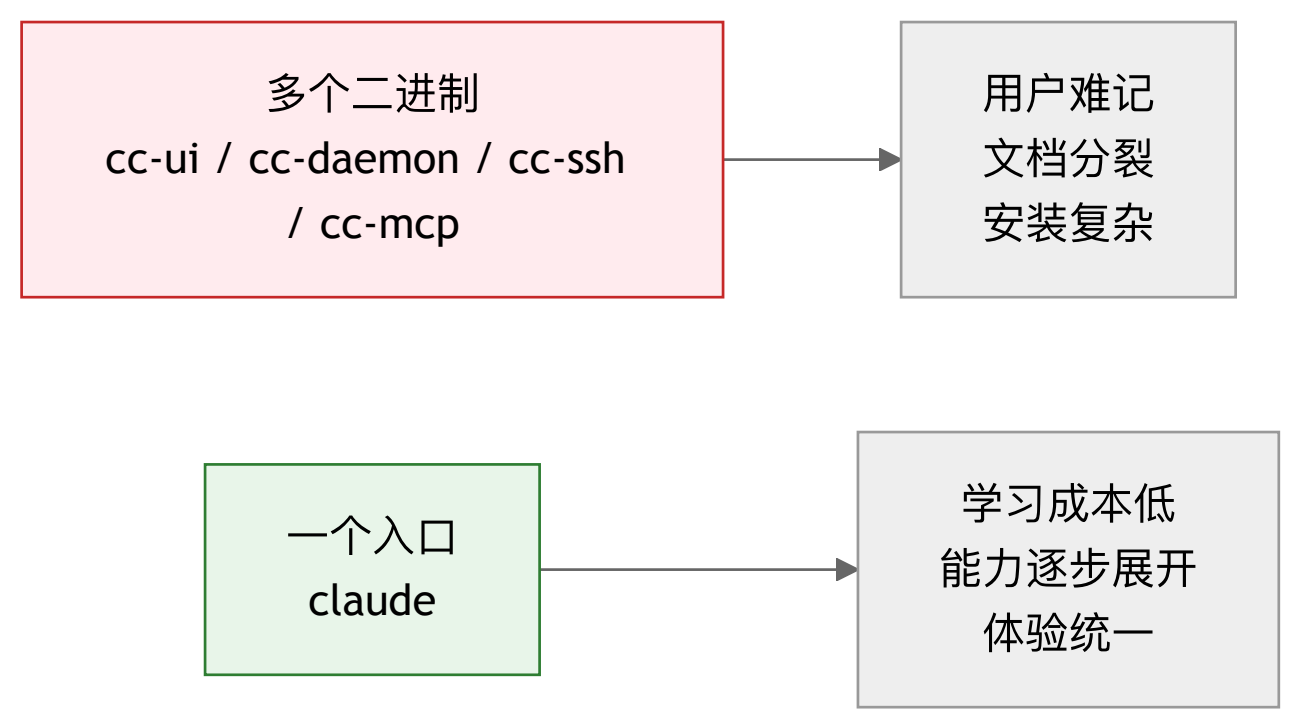
你没用 bridge，就不导 bridge。

你没用 daemon，就不导 daemon。

你没开交互，就不创建 Ink root。

3. 用户只记一个命令

从产品角度看，这一点非常值钱：



这就是“**对用户收敛，对内部分层**”的经典工程取舍。

深水区（架构师选读）

Claude Code 的“多模式”本质上不是命令行参数技巧，而是**单入口、多运行时人格**。

入口文件用动态导入和 feature flag 保住冷启动性能；`main.tsx` 用 Commander 保住命令结构化；action handler 再根据交互/无头/远程实际场景创建不同的运行容器。这样既避免了“一套代码复制成多个产品”，又避免了“所有模式挤在同一条最重路径上”。

代价也很明显：`main.tsx` 会变得非常大，模式之间的边界要靠严格分层和持续重构来维持。这也是为什么读源码时要始终分清：**入口分流、命令注册、运行时装配，是三件不同的事。**

本章小结

一句话：`claude` 不是“一个终端聊天程序”，而是一个统一入口下的多模式平台，靠入口快路径、Commander 命令树和运行时分支三层分流来保持既统一又不混乱。

关键源码索引

证据层	文件	本章关注点
还原层	`claude-code-sourcemap/restored-src/src/entrypoints/cli.tsx:33-179`	版本、bridge、daemon 等快路径
还原层	`claude-code-sourcemap/restored-src/src/entrypoints/cli.tsx:182-298`	后台会话、runner、worktree、导入 `main.js`
补全层	`OpenClaudeCode/src/main.tsx:908-1013`	Commander 顶层 program 与 options
补全层	`OpenClaudeCode/src/main.tsx:2618-2659`	`--print` / headless 状态装配
补全层	`OpenClaudeCode/src/main.tsx:3140-3208`	交互 REPL、direct connect、SSH 远程
补全层	`OpenClaudeCode/src/main.tsx:3900-4498`	`mcp`、`auth`、`plugin`、`server` 等子命令树

逆向提醒

- ✅ **可信度高**：模式矩阵与命令树在还原层和补全层都很清晰
- ⚠️ **不要把 feature flag 当默认功能**：有些模式受构建目标或权限门控控制
- ⚠️ **“10种模式”是阅读框架，不是全部命令总数**：真实源码里的分支比这更多

11

状态管理 第二编

第7章：程序的"记忆"：状态管理的艺术

生活类比

你一边做数学题，一边还记得老师刚刚说的提示、桌上放着哪张草稿纸、上一步算到了哪里——这叫**工作记忆**。程序也需要这种“当前脑内状态”，否则每次都要从头回忆。

这一章要回答的问题

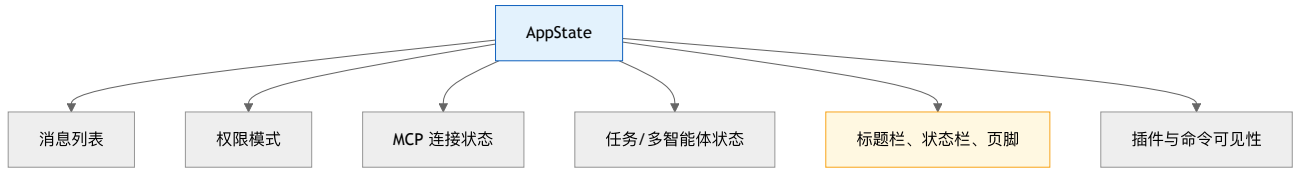
Claude Code 这种大型交互程序，用什么保存“当前状态”？为什么不用 Redux，也不直接用一堆全局变量？

终端里显示哪条消息、当前权限模式是什么、哪些 MCP 工具已连接、是否在远程桥接、有没有后台任务、终端标题显示什么……这些都不是“数据库里的永久数据”，而是**会不断变化的运行时状态**。本章就是要拆开 Claude Code 的“短期记忆系统”。

7.1 为什么大型 CLI 也逃不过状态管理

很多初学者觉得只有前端网页才需要状态管理。其实 Claude Code 这种交互式 CLI 更需要。

你看到的每一块界面都依赖状态



如果没有一套稳定的状态管理方案，会很快掉进两个坑：

坑	会发生什么
全局变量到处飞	谁改了状态、什么时候改的，没人说得清
所有组件全量重渲染	稍微改一点状态，整个终端 UI 都抖一遍

Claude Code 的选择很有意思：**不用 Redux 那种重型框架，而是自己写了一个极小 store，再用 React 的 `useSyncExternalStore` 订阅切片。**

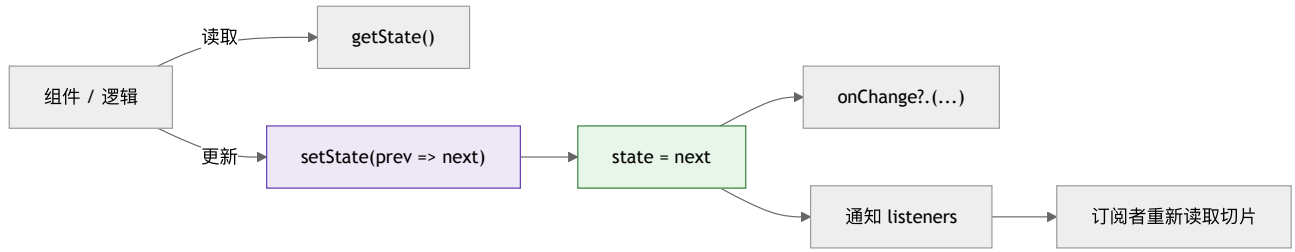
7.2 `createStore()`：状态底座只有 34 行

先看 `OpenClaudeCode/src/state/store.ts`，整个 store 核心只有几十行：

```
export function createStore<T>(initialState: T, onChange?: OnChange<T>): Store<T> {
  let state = initialState
  const listeners = new Set<Listener>()

  return {
    getState: () => state,
    setState: updater => { ... },
    subscribe: listener => { ... },
  }
}
```

它做的事其实非常朴素

- `getState()`：读当前状态
- `setState(updater)`：基于旧状态算出新状态
- `subscribe(listener)`：当状态变化时通知订阅者



为什么说它“像 Zustand，但更小”

它的风格很像很多轻量状态库：

- 没有 reducer 模板代码
- 没有 action type 字符串
- 没有复杂 middleware 管道
- 更新方式就是一个函数：`prev => next`

这对 Claude Code 来说特别合适，因为它的状态变化经常来自：

- 工具执行完成
- 权限状态变更
- 会话恢复完成
- 插件/MCP 动态重载
- REPL 输入与快捷键事件

这些场景要的是**灵活、低摩擦**，不是 ceremony（繁文缛节）。

7.3 `AppStateProvider`：把 store 送进整棵组件树

只有 store 还不够，还得把它安全地送到 React 组件树里。这就是 `AppStateProvider` 的工作。

Provider 做了三件关键事

1. 创建 store（只创建一次）
2. 用 React Context 把 store 向下传
3. 监听外部设置变化，同步回 AppState

`AppStateProvider` 里有几个特别值得记住的点。

1. 不允许嵌套 Provider

源码里直接写了：

```
if (hasAppStateContext) {
  throw new Error("AppStateProvider can not be nested within another AppStateProvider")
}
```

这很好理解。状态容器一旦套娃，开发者就很容易分不清自己读到的是哪一层状态。

2. store 用 `useState(() => createStore(...))` 固定住

这意味着 Provider 自己不会因为父组件重渲染就重新创建 store。对大型交互程序来说，这一点非常关键。

3. 组件只订阅自己需要的那一小块状态

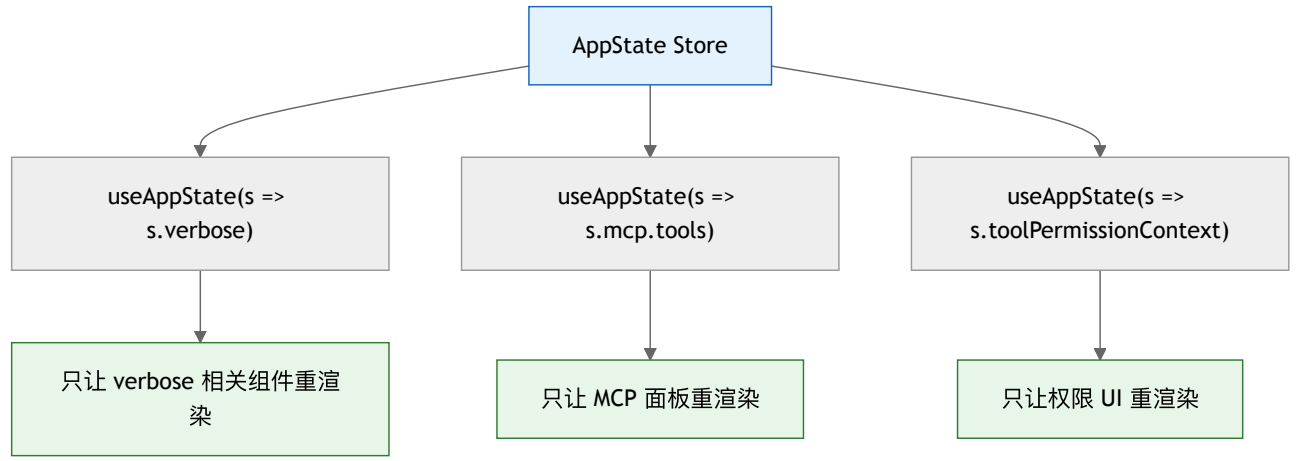
真正优雅的地方在 `useAppState(selector)`：

```
return useSyncExternalStore(store.subscribe, get, get)
```

它要求你传一个 selector，例如：

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

也就是说，**组件不需要知道整个世界，只拿它关心的那一片。**



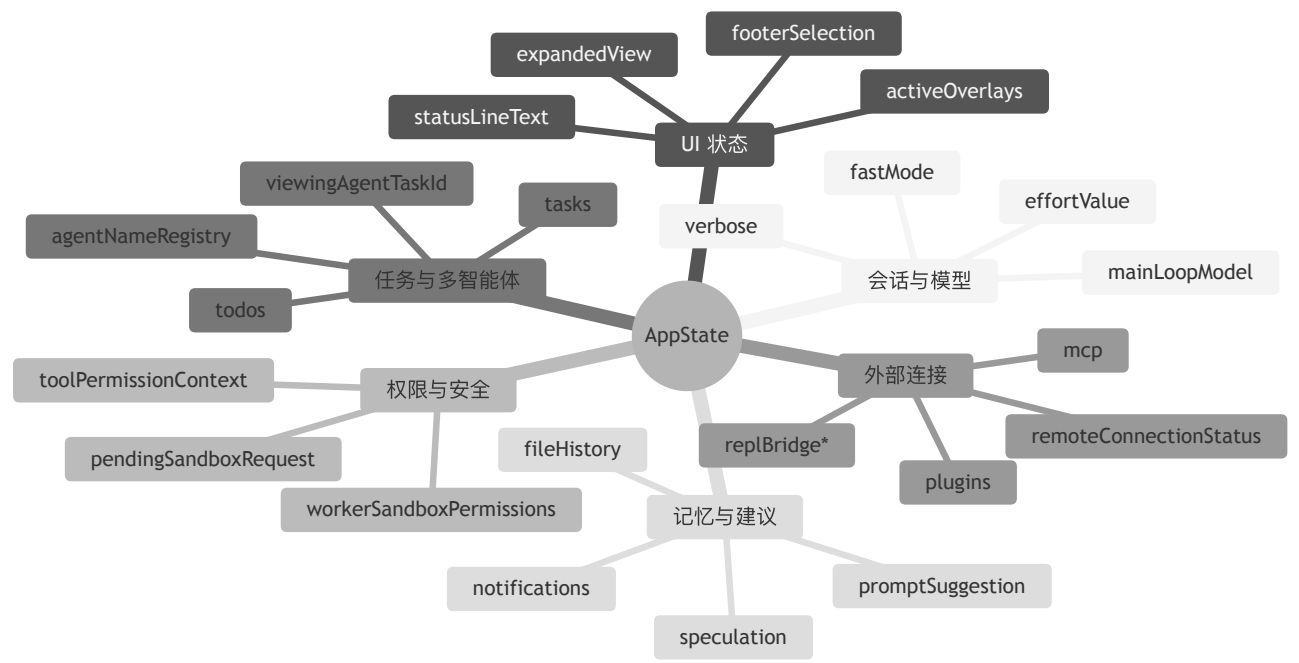
这就是为什么源码里还专门提醒你：

- 不要在 selector 里返回新对象
- 最好多次调用 hook，分别拿独立字段

因为这样才能让渲染足够精准。

7.4 AppState 到底装了什么

`AppStateStore.ts` 里的 `AppState` 非常大，但你不必被吓到。把它按“功能分区”看，就容易多了。



初学者最该先记住的 6 类状态

类别	代表字段	作用
会话基础	`verbose`、`mainLoopModel`	当前对话用什么风格、什么模型
视图状态	`expandedView`、`footerSelection`	当前 UI 展开什么、焦点在哪
权限状态	`toolPermissionContext`	当前允许什么工具、处于哪种权限模式
任务状态	`tasks`、`viewingAgentTaskId`	多智能体和后台任务的运行情况
外部连接	`mcp`、`plugins`、`replBridge*`	外部能力接入和远程桥接
辅助能力	`promptSuggestion`、`fileHistory`	建议输入、文件历史、推测状态

默认状态不是“空白”，而是“可运行起点”

`getDefaultAppState()` 返回的并不只是空对象，它已经给出了一套能启动会话的默认形状：

- `verbose: false`
- `expandedView: 'none'`
- `toolPermissionContext.mode` 根据环境初始化为 `default` 或 `plan`
- `mcp.clients/tools/commands/resources` 先从空开始
- `plugins.enabled/disabled` 先留空壳
- `thinkingEnabled`、`promptSuggestionEnabled` 会走默认策略

这就像电脑开机后的 BIOS 默认设置。它并不包含用户任务，但已经给系统一个稳定起跑姿势。

7.5 交互模式和无头模式，居然共用同一套状态底座

这一点非常漂亮，也非常值得学习。

在交互模式里：状态通过 Provider 进 UI 树

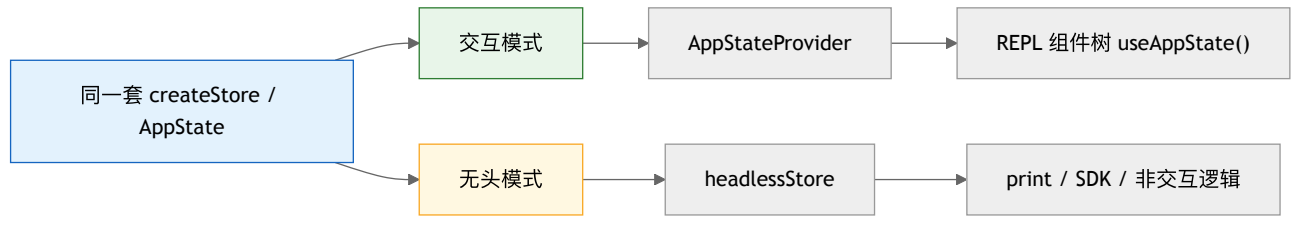
路径大致是：

- `App.tsx` 创建 `<AppStateProvider initialState=...>`
- REPL 里的组件通过 `useAppState()` 读取切片

在无头模式里：状态通过 `createStore()` 直接给逻辑层

`main.tsx` 的 headless 分支会这样做：

```
const defaultState = getDefaultAppState()
const headlessInitialState = { ...defaultState, ... }
const headlessStore = createStore(headlessInitialState, onChangeAppState)
```



这说明 Claude Code 团队没有把“交互程序状态”和“无头模式状态”各写一套，而是抽出了一个更底层、更中性的状态底座。

这非常像一辆车：

- 交互模式像“你坐在驾驶位自己开”
- 无头模式像“把车放到测试台上跑程序”

方向不同，但底盘是同一个。

7.6 为什么这套方案比 Redux 更适合 Claude Code

这不是说 Redux 不好，而是**工程要看场景**。

对比项	Redux 式方案	Claude Code 方案
更新方式	action + reducer	`setState(prev => next)`
模板代码	较多	很少
对 CLI 适配	需要额外组织	天然直接
精准订阅	可以，但配置更多	`useSyncExternalStore(selector)` 直接做
学习成本	中等	较低

Claude Code 的状态问题不是“多人协作修改浏览器页面”，而是“**终端里的大量即时状态 + 工具执行 + 会话切换 + 权限变化**”。这种场景下，轻量和可嵌入性更重要。

不过它也有代价：

- `AppState` 会越来越大
- 状态更新缺少强约束时，容易出现“谁都能改”
- 需要团队自己维护好命名、分层和 selector 习惯

所以这套方案的关键前提不是框架神奇，而是团队 discipline（自律）够强。

🔭 深水区（架构师选读）

Claude Code 的状态层可以概括成一句话：**用最小 Store 内核，换最大的运行时适配性。**

`createStore()` 本身极小，既能嵌进 React，也能直接用于 headless；`AppStateProvider` 负责 UI 世界的桥接；`useSyncExternalStore` 负责精准订阅。这样它既不像 Redux 那样偏“前端框架化”，也不像全局变量那样失控，更接近一种“工程化的内存总线”。

真正的难点不在 API，而在 AppState 规模膨胀后的治理：哪些字段应当持久化，哪些只活一帧，哪些属于权限域，哪些属于 UI 域。读这类源码时，你会发现：**状态管理从来不只是技术问题，更是建模能力问题。**

本章小结

一句话：Claude Code 用一个极小的自研 store 做状态底座，再用 `AppStateProvider + useSyncExternalStore` 把它安全、高效地接进 React 和无头逻辑两边。

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/state/store.ts:1-34`	极简 store 核心
补全层	`OpenClaudeCode/src/state/AppState.tsx:37-109`	Provider 创建与注入 store
补全层	`OpenClaudeCode/src/state/AppState.tsx:117-198`	`useAppState` / `useSetAppState` / `useSyncExternalStore`
补全层	`OpenClaudeCode/src/state/AppStateStore.ts:89-240`	AppState 类型分区
补全层	`OpenClaudeCode/src/state/AppStateStore.ts:456-569`	默认状态初始化
补全层	`OpenClaudeCode/src/main.tsx:2629-2659`	headless 分支复用同一底座
还原层	`claude-code-sourcemap/restored-src/src/state/store.ts`	还原层中同样存在轻量 store 骨架

逆向提醒

- ✅ **可信度高**：状态层的总体架构和 API 非常清楚
- ⚠️ **字段规模大**：`AppState` 很多分支受 feature flag 和构建目标影响，阅读时要分“核心字段”和“条件字段”
- ⚠️ **不要误解为 React 专属**：这套状态底座故意设计成既能服务 UI，也能服务无头逻辑

12

第二编 终端UI

第8章：终端里的"前端"：React如何画出CLI

生活类比

用乐高搭控制面板时，你不会先想“我要打印第 17 个字符”，而是先想“这里需要一个屏幕、这里需要一排按钮、这里需要一个状态灯”。Claude Code 的终端 UI 也是这样搭出来的：**先想组件，再想渲染**。

这一章要回答的问题

终端明明只有字符和颜色，为什么 Claude Code 还要上 React、上自研 Ink、上布局引擎？直接 `console.log` 不就行了吗？

这是很多人第一次读 Claude Code 源码时最惊讶的地方：它不像传统 CLI 那样大量 `stdout.write(...)`，而是搭了一整套“终端版前端框架”。本章就来拆这件事。

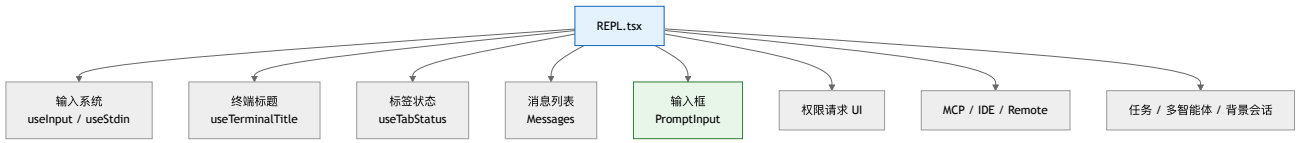
8.1 先别看代码，先回答一个问题：终端界面复杂到什么程度

如果 Claude Code 只是输出一段文字，那确实不需要 React。

但真实情况是，它的 REPL 要同时处理：

- 消息列表
- Markdown 渲染
- 工具调用进度
- 权限确认框
- 状态栏/标题栏/页脚
- 快捷键输入
- 搜索高亮
- 全屏 transcript 视图
- 远程桥接状态
- 多智能体任务树

`OpenClaudeCode/src/screens/REPL.tsx` 本身就有 **5000+ 行**，开头导入的 hook 和组件已经说明一切。



如果用 `console.log` 硬拼，会立刻遇到这些问题：

问题	直接打印会怎样
状态变化后局部刷新	很难，只能粗暴重画
输入与输出并存	光标位置管理会非常痛苦
高亮、选择、搜索	要自己维护字符缓冲区
多组件协作	会变成庞大的 if-else

所以 Claude Code 选择的是：**把终端当成一种“特殊的渲染目标”，而不是低级输出流。**

8.2 `ink.ts`：对外暴露“像前端一样”的终端 API

`OpenClaudeCode/src/ink.ts` 是一个很重要的门面层。

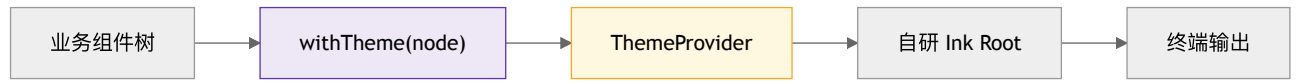
它做的第一件事：自动包上主题

源码里最关键的 3 行是：

```ts
function withTheme(node: ReactNode): ReactNode {
  return createElement(ThemeProvider, null, node)
}
```

然后无论你是 `render(node)` 还是 `createRoot().render(node)`，都会自动先过 `withTheme(node)`。

这说明 Claude Code 并不是把 Ink 当成裸渲染器，而是强行在最外层加了一套设计系统。



它做的第二件事：重新导出终端版“前端原语”

在这个文件里，你会看到一堆熟悉又陌生的东西：

- `Box`
- `Text`
- `Button`
- `Link`
- `useInput`
- `useTerminalTitle`
- `useTabStatus`
- `useTerminalViewport`

这非常像一个自定义 UI 框架的公共入口。对调用方来说，不需要知道底下是怎么写 ANSI、怎么 diff 屏幕、怎么处理 alt-screen，只需要像写 React 组件一样使用。

8.3 `root.ts`：真正把 React 世界挂到终端上的地方

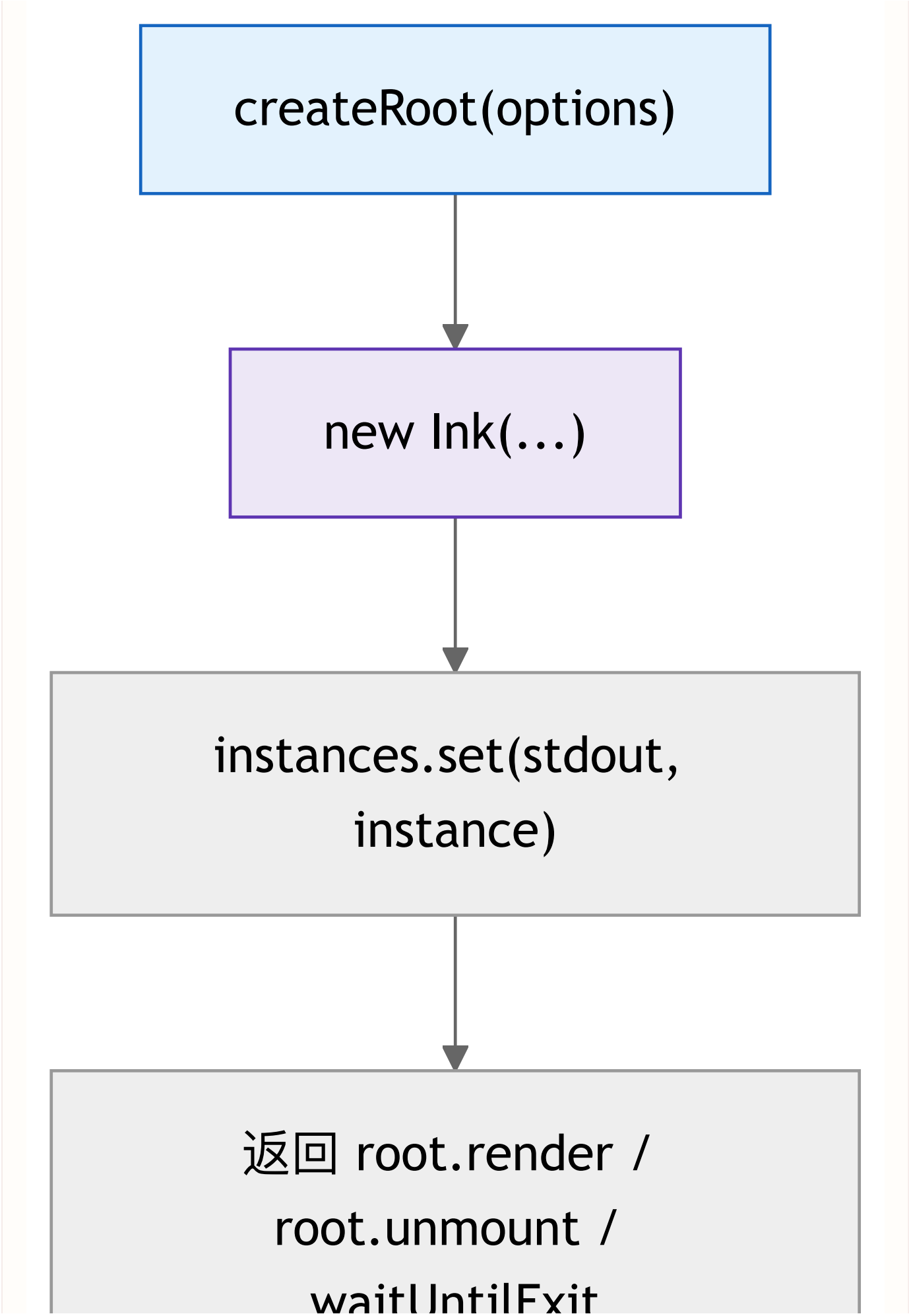
如果说 `ink.ts` 是门面，`ink/root.ts` 就是真正的渲染根。

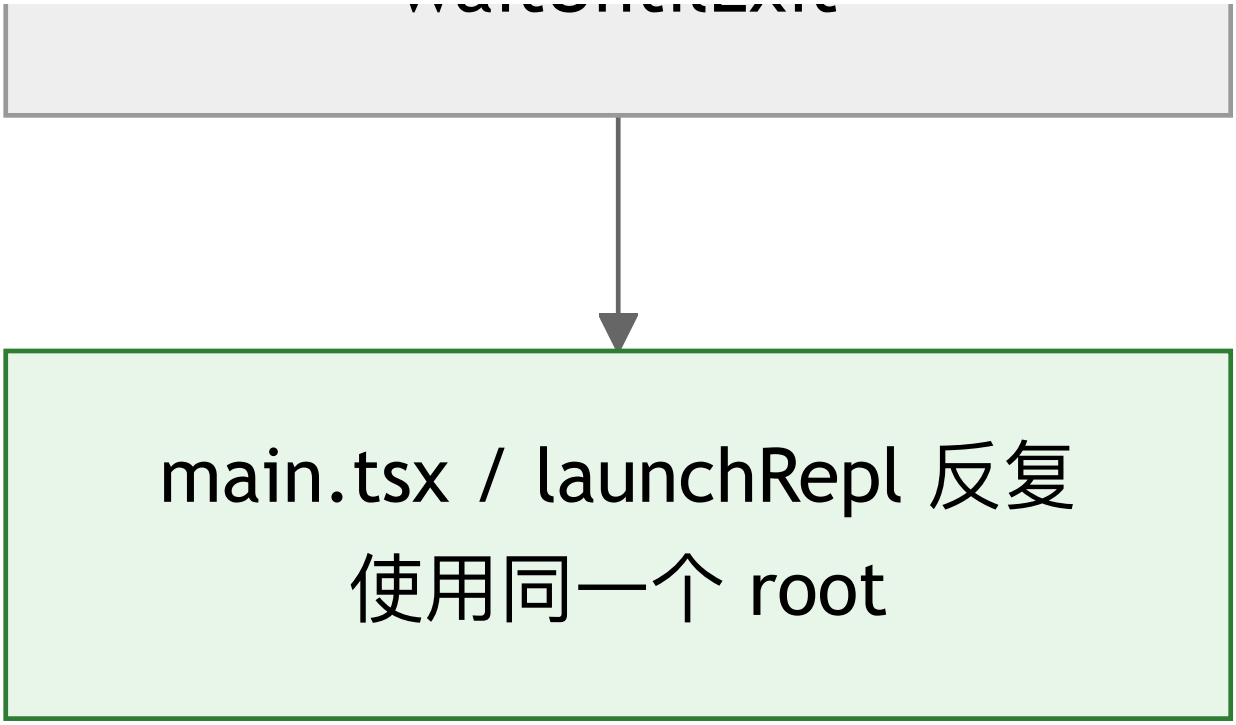
它提供了两个层级的 API

API	含义	适合什么场景
`renderSync` / 默认 `render()`	一次性挂载输出	简单渲染
`createRoot()`	先创建 root，再多次 `render(node)`	交互式会话、顺序多屏切换

对 Claude Code 这种 REPL 来说，`createRoot()` 更重要，因为会话不是“一次渲染完就结束”，而是要长时间活着、不断重绘。

`createRoot()` 背后的思路非常像 `react-dom`


createRoot(options)
new Ink(...)
instances.set(stdout, instance)
返回 root.render / root.unmount / waitUntilExit



源码注释甚至直接说了：

> A managed Ink root, similar to react-dom's createRoot API.

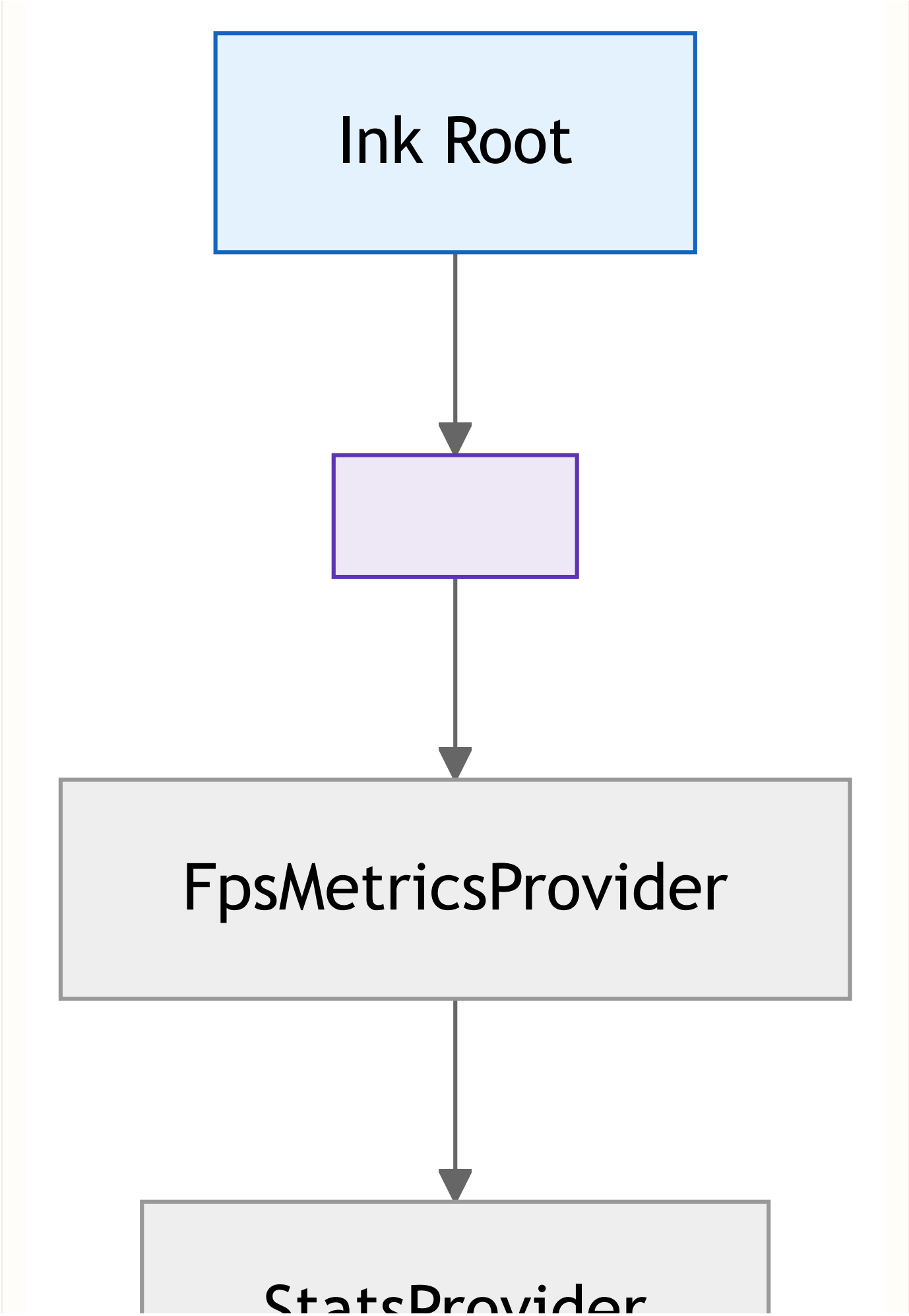
也就是说，Claude Code 在终端里复刻的不是简单打印，而是**一整套“挂载根节点 -> 渲染 -> 更新 -> 卸载”的前端心智模型**。

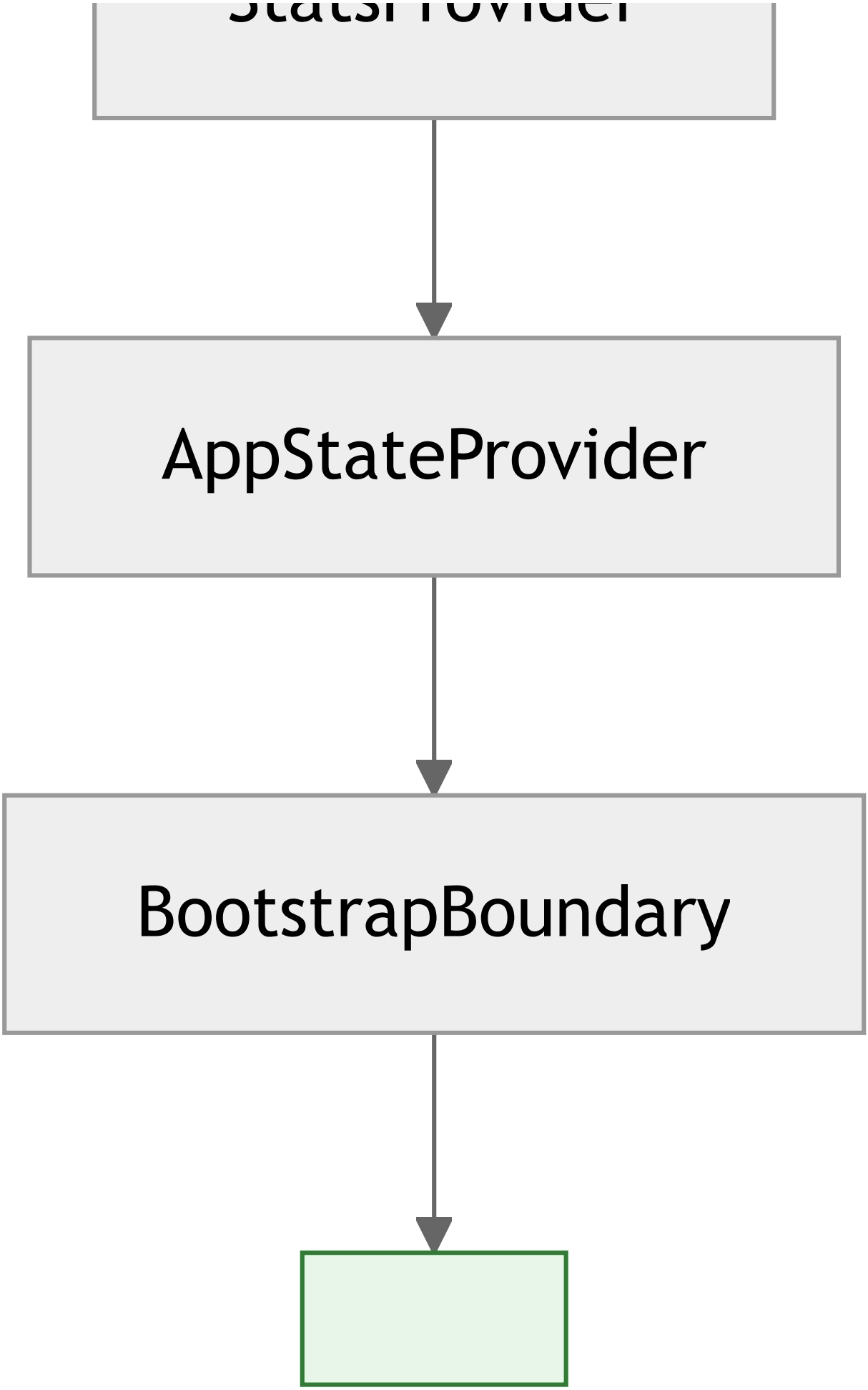
8.4 App 外壳：先注入上下文，再让 REPL 登场

`launchRepl()` 并不会直接渲染 `<REPL />`，而是先包一层 `<App>`。

在 `components/App.tsx` 里，App 会把这些上下文一层层包进去：

- `FpsMetricsProvider`
- `StatsProvider`
- `AppStateProvider`
- `BootstrapBoundary`


Ink Root
FpsMetricsProvider
StatsProvider


AppStateProvider
BootstrapBoundary

为什么要有 `BootstrapBoundary`

这就像浏览器世界的 Error Boundary。终端程序没有浏览器 DevTools 帮你兜底，一旦初始化时组件树炸掉，至少得给用户一个清楚的错误信息，而不是屏幕直接坏掉。

这说明 Claude Code 团队不是只关心“渲染成功的路径”，也在关心“渲染失败时怎么优雅地失败”。

8.5 真正难的地方：不是画出来，而是高效地“改出来”

很多人第一次听说终端 UI，会以为难点在“怎么画彩色文字”。其实真正难的是：

> **上一帧和这一帧只有一点点变化时，如何只改必要部分，而不是整屏重画到闪烁？**

这就是 `ink/renderer.ts` 和 `ink/screen.ts` 的价值。

`renderer.ts`：根据布局结果生成新的一帧

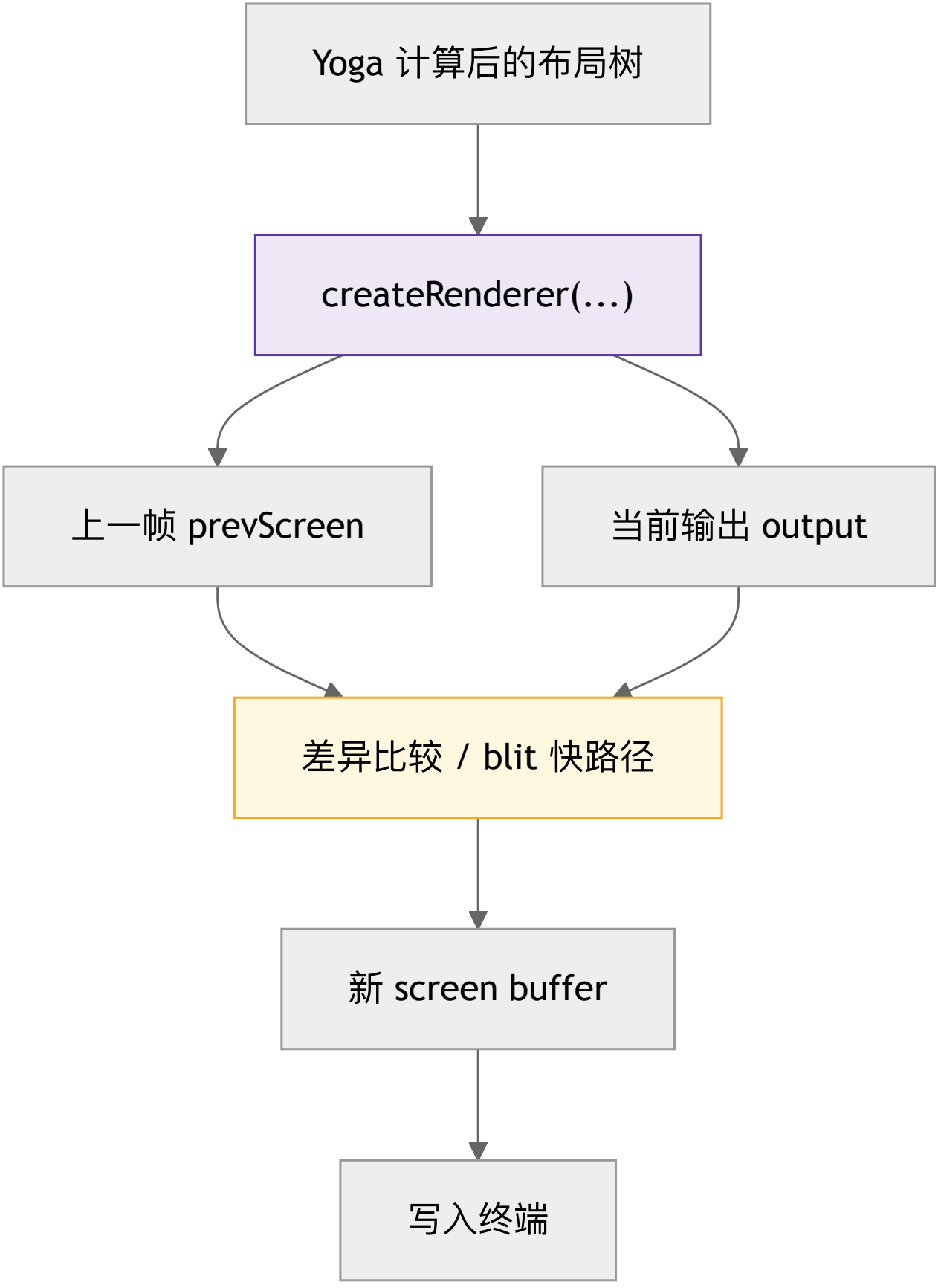
`createRenderer(node, stylePool)` 会返回一个 renderer，它会：

1. 读取前一帧和后一帧缓冲
2. 检查 Yoga 布局尺寸是否合法
3. 创建或复用 `screen`
4. 调用 `renderNodeToOutput(...)`
5. 只在必要时放弃 blit 快路径

源码里有一句特别能说明思路：

> When clean, blit restores the O(unchanged) fast path for steady-state frames.

翻成人话：**如果前一帧没被污染，就尽量走“复制没变区域”的快路径。**



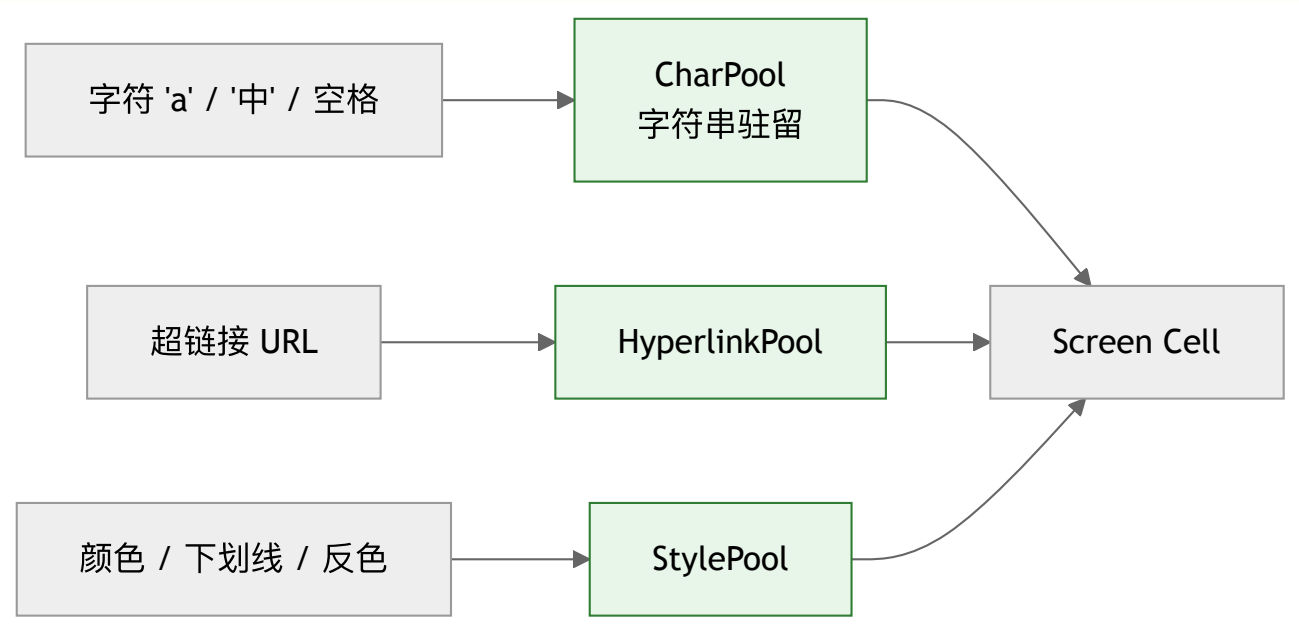
`screen.ts`：把字符池、样式池、链接池都池化起来

`screen.ts` 里最让架构师眼前一亮的，是这些对象：

- `CharPool`
- `HyperlinkPool`

- `StylePool`

它们的核心想法都一样：**不要重复存同样的东西，用整数 ID 代替重复字符串和样式数组。**



这背后其实是一个非常“系统程序员”的思路：

- 用整数比较代替字符串比较
- 用池化减少内存分配
- 用缓存减少重复序列化 ANSI

所以你可以说，Claude Code 的终端渲染栈并不是“前端味很重”，而是**前端抽象 + 系统级优化**同时存在。

8.6 `REPL.tsx`：像前端应用一样接键盘、标题栏、搜索栏

当你看到 `REPL.tsx` 里的这些调用时，就会彻底意识到它不是传统 CLI：

- `useTerminalTitle(...)`
- `useTabStatus(...)`
- `useInput(...)`
- `useSearchHighlight()`

终端标题和标签状态都是 Hook 驱动的

例如：

- `useTerminalTitle(...)` 控制终端标题栏
- `useTabStatus(...)` 控制终端标签栏状态

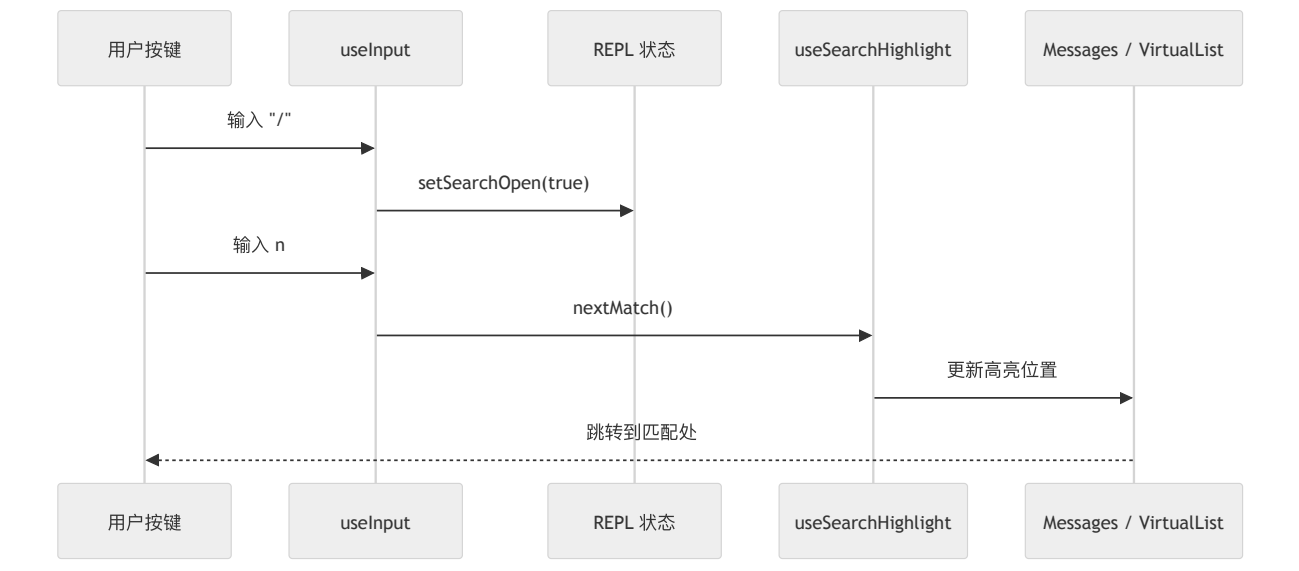
这和浏览器里 `useEffect(() => document.title = ...)` 的心智模型很像，只不过目标从浏览器 tab 变成了终端窗口。

搜索不是“命令”，而是 REPL 内部状态机

在 transcript 模式下，`useInput(...)` 会接管 `/`、`n`、`N` 等输入：

- `/`：打开搜索
- `n`：跳下一个匹配
- `N`：跳上一个匹配

这说明 Claude Code 的输入系统已经不是“读一行文本再提交”，而更像一个真正的 TUI 应用。



这就是为什么第 8 章的标题是“终端里的前端”。从交互模型上看，它和单页应用已经很接近了。

8.7 为什么 Claude Code 不满足于官方 Ink

这本书一个很重要的发现是：Claude Code 并不是“简单用了 Ink”，而是围绕它做了大量自研。

你能从目录结构直接看出它的野心

`src/ink/` 下面不是几个小工具，而是一整套子系统：

- root
- renderer
- screen
- terminal
- events
- hooks
- components
- termio

这说明官方抽象不够时，团队选择的是**把终端渲染栈往下钻透**，而不是凑合。

为什么要这么做

因为 Claude Code 面临的不是普通 CLI 的需求，而是：

- 大量持续流式更新
- 长消息与虚拟列表
- alt-screen 全屏体验
- 搜索与选择覆盖层
- 多智能体任务树
- 复杂权限弹窗与状态面板

官方 Ink 提供了起点，但不一定能直接覆盖这些边界场景。

✨ 深水区（架构师选读）

Claude Code 的终端 UI 架构最有意思的地方，在于它把三个世界缝在了一起：

1. **React 的声明式组件模型**
2. **Yoga/缓冲区/ANSI diff 的底层渲染优化**
3. **CLI 独有的输入、终端标题、alt-screen、scrollback 语义**

这不是“把网页搬到终端”，而是“把前端的抽象能力移植到终端，再用系统级手法把性能拉回来”。所以读这一层源码时，最值得学的不是某个 Hook 名字，而是一个方法论：**高层抽象负责组织复杂度，底层优化负责守住性能。**

本章小结

一句话：Claude Code 之所以能在终端里做出接近前端应用的体验，是因为它把 React 组件模型、自研 Ink 根节点、屏幕缓冲区和终端事件系统整合成了一套完整的渲染栈。

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/ink.ts:12-31`	`withTheme`、`render()`、`createRoot()` 门面层
补全层	`OpenClaudeCode/src/ink/root.ts:62-157`	终端版 `createRoot` 与实例管理
补全层	`OpenClaudeCode/src/components/App.tsx:46-95`	App 外壳与上下文注入
补全层	`OpenClaudeCode/src/ink/renderer.ts:31-177`	帧渲染、alt-screen、blit 快路径
补全层	`OpenClaudeCode/src/ink/screen.ts:21-220`	`CharPool` / `StylePool` 等池化设计
补全层	`OpenClaudeCode/src/screens/REPL.tsx:12-30`	REPL 顶层依赖规模
补全层	`OpenClaudeCode/src/screens/REPL.tsx:517`	终端标题 Hook
补全层	`OpenClaudeCode/src/screens/REPL.tsx:1205-1207`	终端标签状态 Hook
补全层	`OpenClaudeCode/src/screens/REPL.tsx:4253-4294`	transcript 搜索输入逻辑
还原层	`claude-code-sourcemap/restored-src/src/screens/REPL.tsx`	交互层的还原证据

逆向提醒

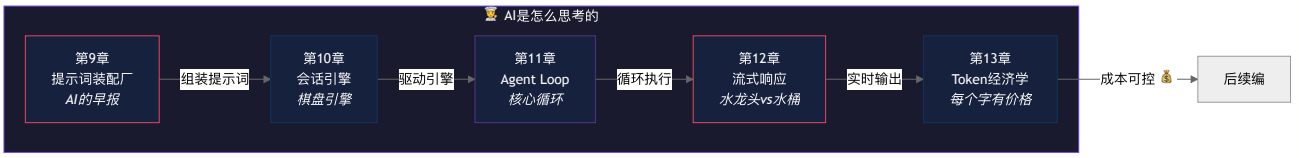
- ✅ **可信度高**：自研 Ink 渲染栈、REPL 交互方式、屏幕缓冲优化都能直接从源码看出
- ⚠️ **细节复杂**：`screen.ts` 和 `renderer.ts` 很底层，第一次阅读应先抓住“池化 + 差分 + alt-screen”三条主线
- ⚠️ **不要把它当网页 UI**：它借用了 React 的思维，但约束来自终端，不是浏览器

13

第三编：AI是怎么思考的

> *厨师做菜的循环：看菜单、备食材、烹饪、尝味道、调整、上菜。AI 的工作循环也是如此。*
>
> 本编深入 Claude Code 的"大脑"：**提示词组装**、**会话引擎**、**Agent Loop**、**流式响应**、**Token 经济学**。

本编总览



本编五章速览

章	标题	核心问题	生活类比
9	提示词装配厂	在你说第一个字之前，AI 已经收到了多少"背景知识"?	新员工的培训手册
10	会话引擎	直接调 API 不行吗？为什么还要抽出"引擎"?	下棋的棋盘引擎
11	Agent Loop	一个 `while(true)` 怎么支撑整个 AI 助手?	厨师做菜的循环
12	流式响应	回复为什么一个字一个字蹦出来?	水龙头 vs 水桶
13	Token经济学	怎么在有限"流量"里完成更多任务?	手机流量套餐

设计思想主线

本编建立的认知基础

1. System prompt 在用户说第一句话之前就已组装完成——**几千 token 的"早报"决定 AI 行为质量**
2. QueryEngine 封装了多轮对话、工具分发、中断恢复等复杂性——**上层只需"发问题、收答案"**
3. Agent Loop 的核心是一个 `while(true)`——**简单但不容易**
4. 流式响应把用户感知延迟从 30 秒压到 2 秒——**体验工程的典范**
5. Token 是真金白银——**prompt caching 可省 70–90% 成本**

推荐路径

◯ ◯ ◯

🌱 初学者🔧 开发者🏗️ 架构师

第11章的 Agent Loop 是最核心的概念——**理解"想→做→看→再来"的循环就理解了 AI Agent 的本质**。

第9章和第10章揭示了生产级 AI 应用的工程细节。**prompt 组装和会话管理是自建 AI 应用的必修课**。

第13章的 Token 经济学直接影响产品的商业可行性。**成本控制是 AI 产品从 demo 到产品的关键一步**。

阅读建议

如果你第一次读 Claude Code 源码，建议按 **第9章 → 第10章 → 第11章 → 第12章 → 第13章** 的顺序通读。这样会先建立“提示词底座”，再理解“会话引擎”，最后再把 Agent Loop、流式响应和 Token 成本串成一条完整主线。

14

提示词工程 第三编

第9章：提示词装配厂：AI上班前的"早报"

生活类比

一个新员工第一天来公司，不会被直接扔到工位上就开始干活。通常会先拿到一套资料：公司规则、团队分工、项目背景、常见流程、不能踩的坑。**Claude Code 每次开工前，也会先给模型发一份“入职资料包”。**

这一章要回答的问题

在你说第一个字之前，Claude Code 已经给 AI 塞了多少背景知识？这些背景知识又是按什么顺序装进去的？

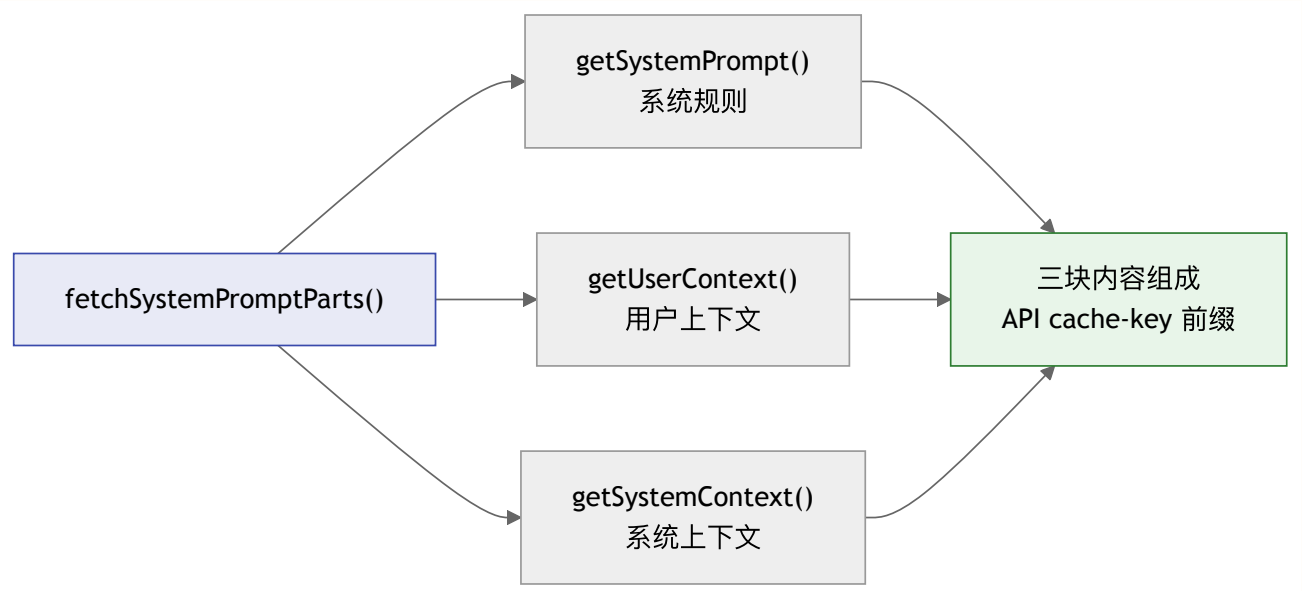
很多人把 system prompt 想成一大段神秘咒语。但源码里真正发生的事更像一条装配线：**先收集通用规则，再拼装运行时上下文，再把项目记忆和额外要求叠上去，最后还要考虑缓存命中率和成本。**

9.1 先别想成“一段大 prompt”，它其实是三块前缀

在 `utils/queryContext.ts` 里，Claude Code 把每次请求真正依赖的“上下文前缀”拆成三块：

- `defaultSystemPrompt`
- `userContext`
- `systemContext`

而且是并行取回的，不是串行慢慢拼：

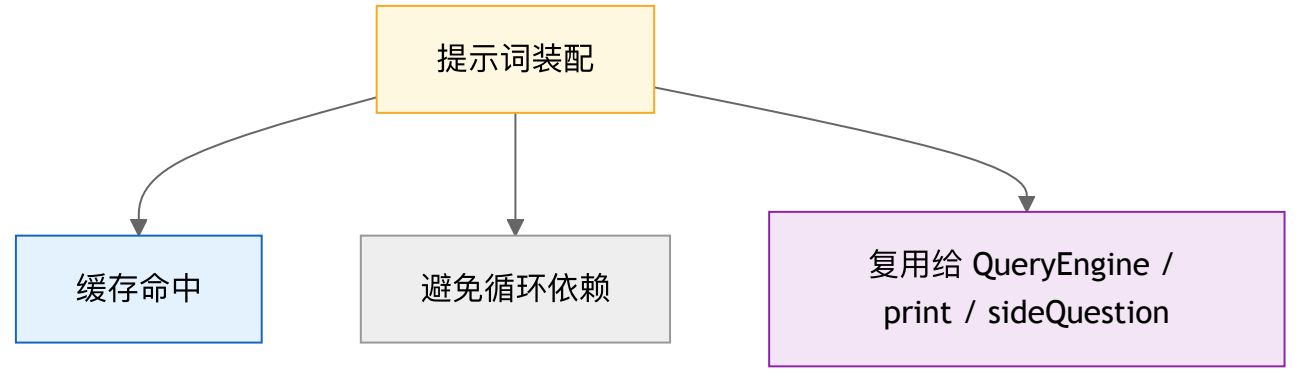


这件事非常重要，因为它告诉我们：

1. Claude Code 不把所有信息都塞进 system prompt
2. **有些内容属于“系统规则”，有些属于“用户现场”，还有些是“运行时环境”**
3. **这三块共同影响缓存命中和后续循环**

为什么这里要专门拆成一个 `queryContext.ts`

文件开头的注释已经点破了设计动机：这部分逻辑之所以被抽到独立文件里，是为了避免依赖环，同时把“缓存安全前缀”定义得足够稳定。



自定义 system prompt 为什么会跳过默认构建

`fetchSystemPromptParts()` 里有一个很值得初学者记住的判断：

- 如果传入了 `customSystemPrompt`
- 就不再构建默认 `getSystemPrompt()`
- 同时也跳过 `getSystemContext()`

这意味着“自定义提示词”不是在默认提示词后面再追加一段，而是**直接替换默认底座**。这和很多人以为的“多加一句”完全不同。

源码证据

`OpenClaudeCode/src/utils/queryContext.ts:29-73` 明确写出了三块上下文的并行获取与 custom prompt 的替换规则。

9.2 真正的装配顺序：谁覆盖谁，谁追加谁

如果说上一节是在“备料”，那这一节就是“总装”。真正决定最后发给模型的 system prompt 长什么样的，是 `buildEffectiveSystemPrompt()`。

它的优先级在注释里写得非常清楚：

换成大白话就是：

场景	最终 system prompt 怎么来
有 `overrideSystemPrompt`	直接整包替换
协调者模式开启	用 coordinator prompt 当主干
有主线程 agent	默认情况下 agent prompt 替换默认 prompt
开启 proactive / KAIROS	agent prompt 不替换，而是追加到默认 prompt 后
有 `customSystemPrompt`	替换默认 prompt
有 `appendSystemPrompt`	无论前面选了谁，最后都可以再加一层

这套规则很像公司内部的“层层发文”：

- 总部通知可以覆盖部门通知
- 特殊项目组会有自己的工作守则
- 最后领导还可以再补一句“今天注意这个”

为什么 `appendSystemPrompt` 永远留在最后

因为它常常扮演的是“最后一条临时要求”：

- 这次回答请更简洁
- 这次重点关注某个目录
- 这次不要调用某类工具

如果不放在最后，它很容易被前面的规则吞掉注意力。

一个很容易忽略的细节：Agent Prompt 不总是替换默认 Prompt

在 PROACTIVE 或 KAIROS 模式下，源码不是“把默认 prompt 丢掉再上 agent prompt”，而是：

- 保留默认 prompt
- 再追加一段 `# Custom Agent Instructions`

这个设计说明 Claude Code 已经不把 agent 看成“另一个聊天机器人”，而是看成**跑在统一底座上的一类角色**。

源码证据

`OpenClaudeCode/src/utils/systemPrompt.ts:28-120` 给出了完整优先级；其中 `:99-113` 明确写出 proactive / KAIROS 模式下的追加策略。

9.3 项目记忆怎么进来：不是神秘感知，而是文件装配

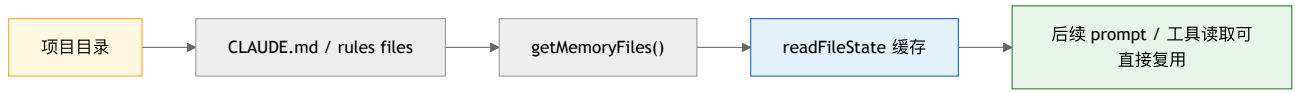
很多初学者会问一个很自然的问题：

> Claude Code 怎么知道这个项目的特殊约定?
> 它又怎么记住我之前说过的话?

答案不是“它自己突然懂了”，而是：**Claude Code 把文件系统里的记忆和规则，主动装进了模型能读到的位置。**

REPL 启动时，会先把 CLAUDE.md 和规则文件读进缓存

在 `screens/REPL.tsx` 里，初始化阶段就会调用 `getMemoryFiles()`，把找到的 `CLAUDE.md / rules` 文件塞进 `readFileState`。



这说明 Claude Code 不是“每到需要时临时扫盘”，而是提前把这类高价值规则文件放进一个统一缓存里。

Memory prompt 不是聊天记录，而是一套“怎么记”的规则

`memdir/memdir.ts` 里最有意思的一点是：`loadMemoryPrompt()` 注入的不是具体记忆内容本身，而是**如何使用 MEMORY.md 的行为规则**。

源码里能看到这些事实：

- `ENTRYPOINT_NAME` 固定为 `MEMORY.md`
- 有 `MAX_ENTRYPOINT_LINES = 200`
- 有 `MAX_ENTRYPOINT_BYTES = 25_000`
- 超长时会截断，并附上 warning
- `loadMemoryPrompt()` 会在需要时确保 memory 目录存在

这就像老师发给你一本“学习档案使用手册”，不是把你过去所有考试卷都钉进去，而是先规定：

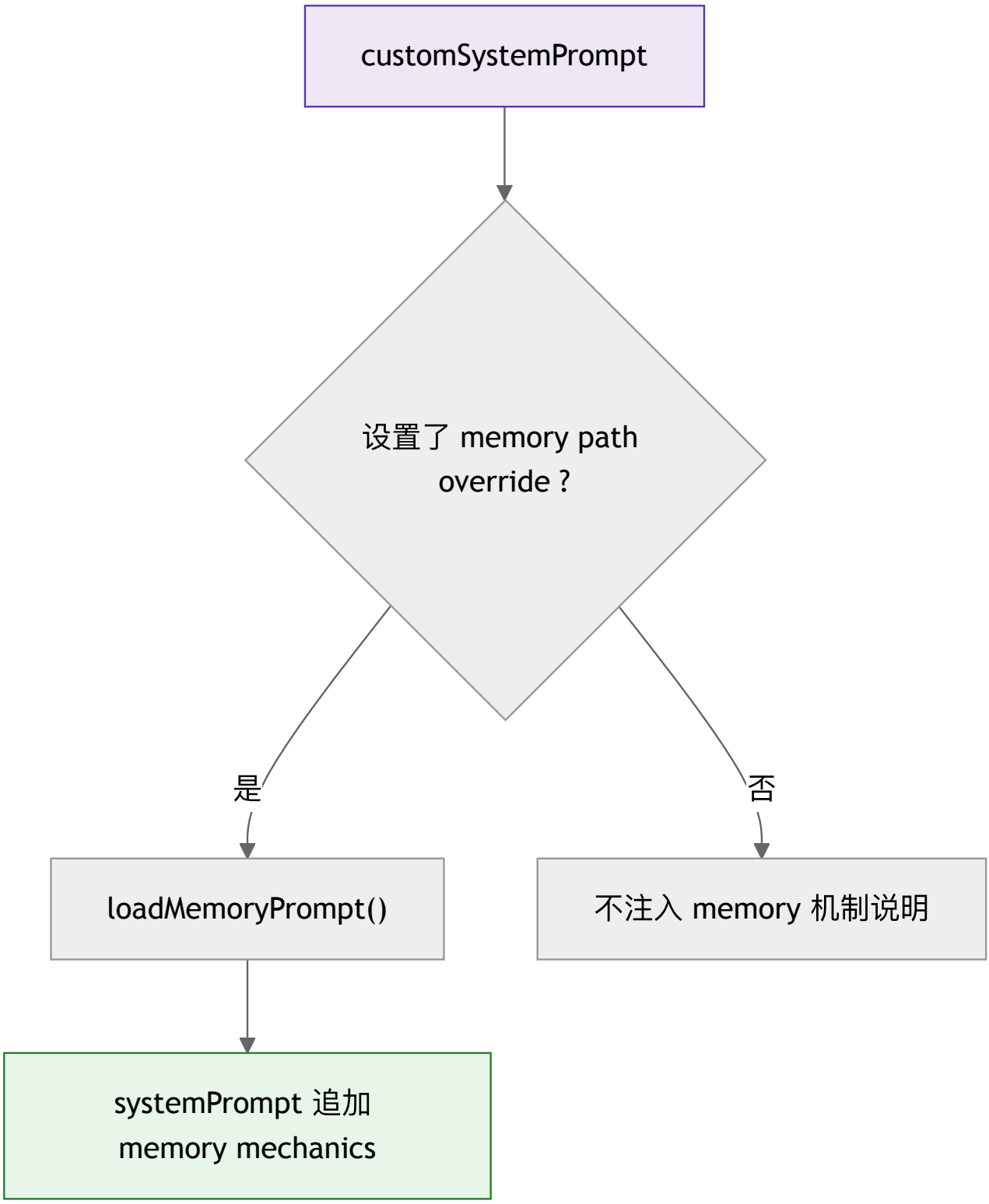
- 哪些内容该记
- 哪些不该记
- 文件放哪里
- 格式怎么写

QueryEngine 还能在特定条件下注入“记忆机制说明”

`QueryEngine.ts` 里还有个很妙的判断：

- 如果 SDK 调用者提供了 `customSystemPrompt`
- 同时设置了 `CLAUDE_COWORK_MEMORY_PATH_OVERRIDE`
- 那就额外注入 `memoryMechanicsPrompt`

也就是说，Claude Code 连“自定义 system prompt 场景下，模型还知不知道怎么写 memory”都考虑到了。



这类设计特别值得架构师注意，因为它反映出一个成熟系统会主动处理“定制化之后会不会把默认能力弄丢”的问题。

源码证据

- `OpenClaudeCode/src/screens/REPL.tsx:3830-3849`：启动时把 `CLAUDE.md / rules` 放进 `readFileState`
- `OpenClaudeCode/src/memdir/memdir.ts:34-38`：`MEMORY.md` 的入口定义与尺寸上限
- `OpenClaudeCode/src/memdir/memdir.ts:419-507`：`loadMemoryPrompt()` 的自动记忆注入逻辑
- `OpenClaudeCode/src/QueryEngine.ts:310-325`：特定条件下注入 `memoryMechanicsPrompt`

9.4 为什么提示词工程已经变成“提示词基础设施”

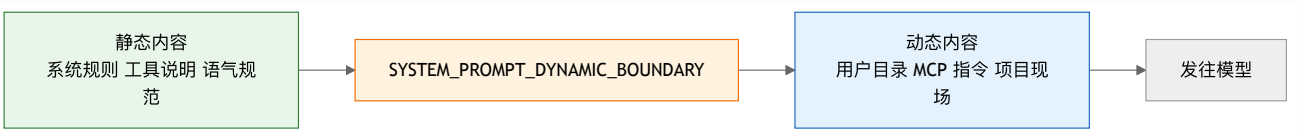
如果你只把 prompt 当成一段文案，会错过 Claude Code 最有价值的设计思想。

真正重要的是：**它已经把 prompt 当成基础设施来管理了。**

最典型的证据，就是 `SYSTEM_PROMPT_DYNAMIC_BOUNDARY`。

源码注释直接说：

- 这个边界前面的内容属于静态、可全局缓存内容
- 后面的内容属于用户和会话相关内容，不应共享缓存
- 不能随便改位置，否则要同步改缓存逻辑



在 `constants/prompts.ts` 的尾部，系统提示词最终是这么返回的：

1. 先放静态部分
2. 中间打一个边界标记
3. 再拼动态 section

这就不是“写 prompt”，而是在做：

- 分层
- 标记
- 缓存友好化
- 成本优化
- 可演化的组装策略

再看 `query.ts` 就更清楚了

模型真正被调用时，并不是只收到 `systemPrompt`，而是：

- `messages: prependUserContext(messagesForQuery, userContext)`
- `systemPrompt: fullSystemPrompt`

也就是说，Claude Code 不是把“所有上下文混成一坨”，而是让不同类型的信息走各自更合适的通道。

设计思想

从 Claude Code 的源码看，Prompt Engineering 已经升级成 Harness Engineering。

重点不再是“我写一句多厉害的话”，而是“我怎么把规则、上下文、角色、缓存边界、成本控制装成一套稳定可维护的系统”。

深水区（架构师选读）

第 9 章最值得记住的不是某一句 system prompt 文案，而是它背后的**装配哲学**：

1. 先定义哪些信息属于稳定底座，哪些属于现场变量
2. 再决定哪些该走 system prompt，哪些该走 userContext
3. 再把缓存边界明确到字符串常量
4. 最后才谈 wording

这和搭数据库、消息队列、配置中心很像。真正成熟的 AI 产品，不会把 prompt 当“灵感”，而会把它当“基础设施”。

本章小结

一句话：Claude Code 的提示词不是一段孤零零的大字符串，而是由 `systemPrompt + userContext + systemContext` 组成的装配体系，还要同时兼顾角色优先级、项目记忆、缓存边界和成本控制。

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/utils/queryContext.ts:29-73`	三块上下文的并行获取
补全层	`OpenClaudeCode/src/utils/systemPrompt.ts:28-120`	system prompt 的优先级与覆盖规则
补全层	`OpenClaudeCode/src/constants/prompts.ts:105-115`	动静态 prompt 边界常量
补全层	`OpenClaudeCode/src/constants/prompts.ts:560-576`	静态部分、边界标记、动态部分的最终拼装
补全层	`OpenClaudeCode/src/QueryEngine.ts:284-325`	QueryEngine 中的 system prompt 总装
补全层	`OpenClaudeCode/src/screens/REPL.tsx:3830-3849`	`CLAUDE.md / rules` 的启动期加载
补全层	`OpenClaudeCode/src/memdir/memdir.ts:419-507`	memory prompt 的注入与自动目录保障

逆向提醒

- ✅ **可信度高**：prompt 的三段式结构、优先级策略、动静态边界，在源码中都有直接注释和实现
- ⚠️ **要分清楚**：`MEMORY.md` 的“行为规则注入”和“具体记忆内容注入”不是一回事
- ❌ **不要误读**：`customSystemPrompt` 在很多路径里是“替换默认底座”，不是“在默认底座后面多加一句”

15

会话引擎 第三编

第10章：会话引擎：不只是聊天那么简单

生活类比

你下棋时，真正负责“这盘棋现在是什么局面、谁走到哪一步、还能不能悔棋”的，不是棋子本身，而是棋盘背后的规则系统。**QueryEngine 就是 Claude Code 的“棋盘引擎”。**

这一章要回答的问题

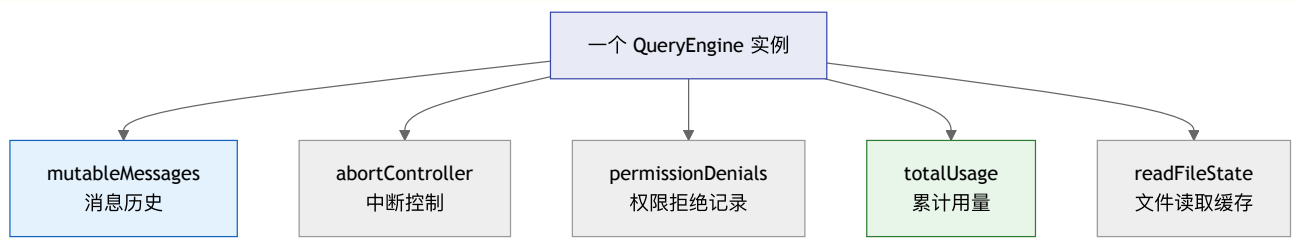
直接调 API 不行吗？为什么还要专门抽出一个 `QueryEngine`？

对一个 demo 来说，也许“拿到用户输入 -> 调模型 -> 打印回复”就够了。但对一个真正能长期对话、能调用工具、能记录成本、能被 SDK 复用的 AI 产品来说，这远远不够。Claude Code 需要一个专门管理“整场对话”的引擎。

10.1 `QueryEngine` 不是 API 封装，而是“整段会话的总管”

`QueryEngine.ts` 的类注释写得很直白：

- 它拥有一段会话的生命周期
- 一次会话里可以多次 `submitMessage()`
- 状态会跨 turn 持续存在



这五块字段已经说明它管的不是“某次请求”，而是“整段会话”：

字段	它保存什么	为什么要跨轮次保留
`mutableMessages`	所有消息历史	下一轮要继续看得见上一轮
`abortController`	中断信号	用户随时可能打断当前任务
`permissionDenials`	被拒绝的工具调用	SDK 需要知道哪些工具没被放行
`totalUsage`	累计 token/usage	会话成本不能每轮归零
`readFileState`	读过的文件缓存	减少重复读盘，提高一致性

这和“普通聊天页面”的根本区别

普通聊天页面往往只关心“把上一轮消息显示出来”。
而 Claude Code 的会话引擎还要关心：

- 是否调用过工具
- 哪些权限被拒绝
- 文件缓存是否还是最新
- 成本是不是超预算
- 会话是否需要落盘持久化

这已经不是“聊天框”了，而是一个**任务执行会话容器**。

源码证据

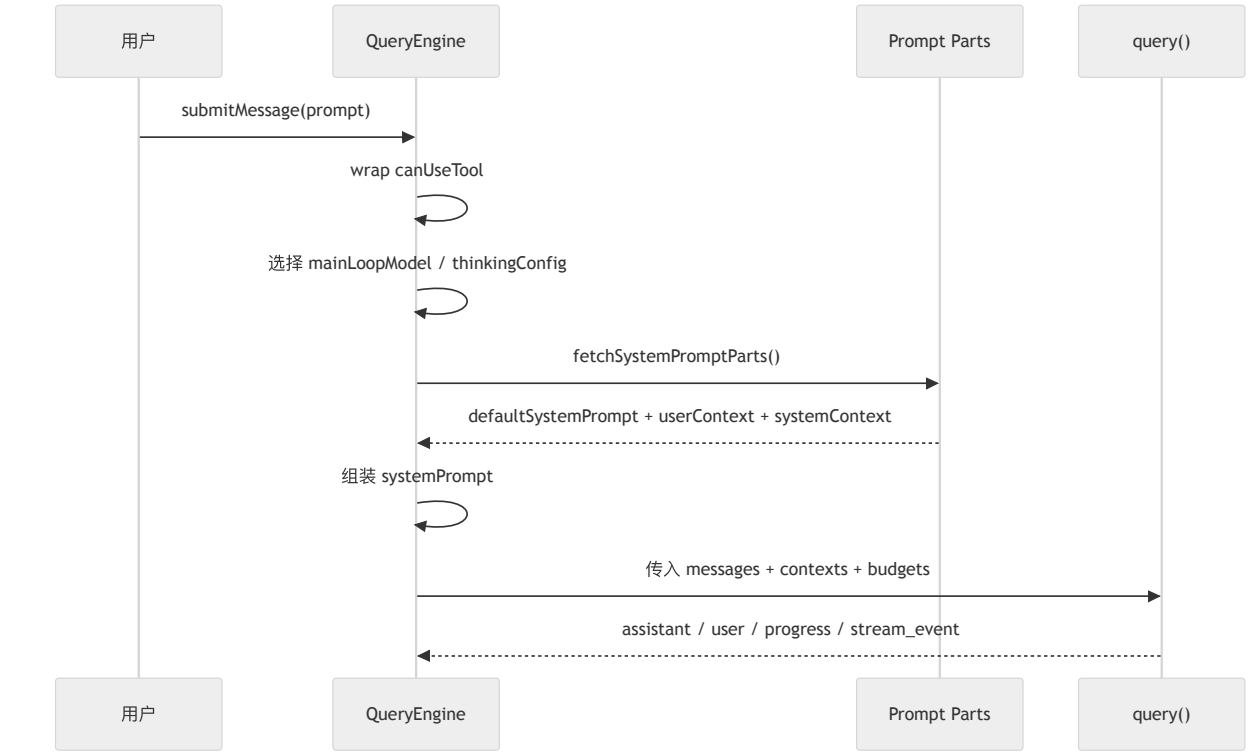
`OpenClaudeCode/src/QueryEngine.ts:176-206` 直接定义了 QueryEngine 的职责与核心状态。

10.2 `submitMessage()` 并不是“发一句话”，而是启动一整轮编排

`submitMessage()` 看起来像一个简单名字，但它干的事非常多：

1. 解构当前配置

2. 包装 `canUseTool`，顺手记录 permission denial
3. 选择模型与 thinking 配置
4. 取回 system prompt / userContext / systemContext
5. 组装 `processUserInputContext`
6. 最后才把这些东西交给 `query()`



一个很容易忽略的细节：拒绝权限也会被会话引擎记住

`submitMessage()` 里会把原始 `canUseTool` 包一层，如果结果不是 `allow`，就把：

- `tool_name`
- `tool_use_id`
- `tool_input`

记录到 `permissionDenials` 里。

这很像公司的门禁系统，不只是负责“让不让进”，还会留下记录，供后续汇总和审计。

模型和 thinking 配置也是在这里定下来的

引擎不会把这些决定完全丢给下层：

- 如果用户指定模型，就解析用户指定值
- 否则走默认主模型
- 如果没有显式 thinking 配置，就根据默认策略决定是 `adaptive` 还是 `disabled`

这说明 QueryEngine 不只是转发器，它还承担了**会话级运行策略决策**。

System prompt 不是 REPL 专属能力

`QueryEngine` 在这里自己调用 `fetchSystemPromptParts()` 和 `asSystemPrompt(...)`，说明这套 prompt 装配逻辑已经从 UI 中抽离出来了。SDK/headless 路径不需要 React，也能完整跑起来。

源码证据

`OpenClaudeCode/src/QueryEngine.ts:209-325` 展示了 `submitMessage()` 前半段的核心编排过程。

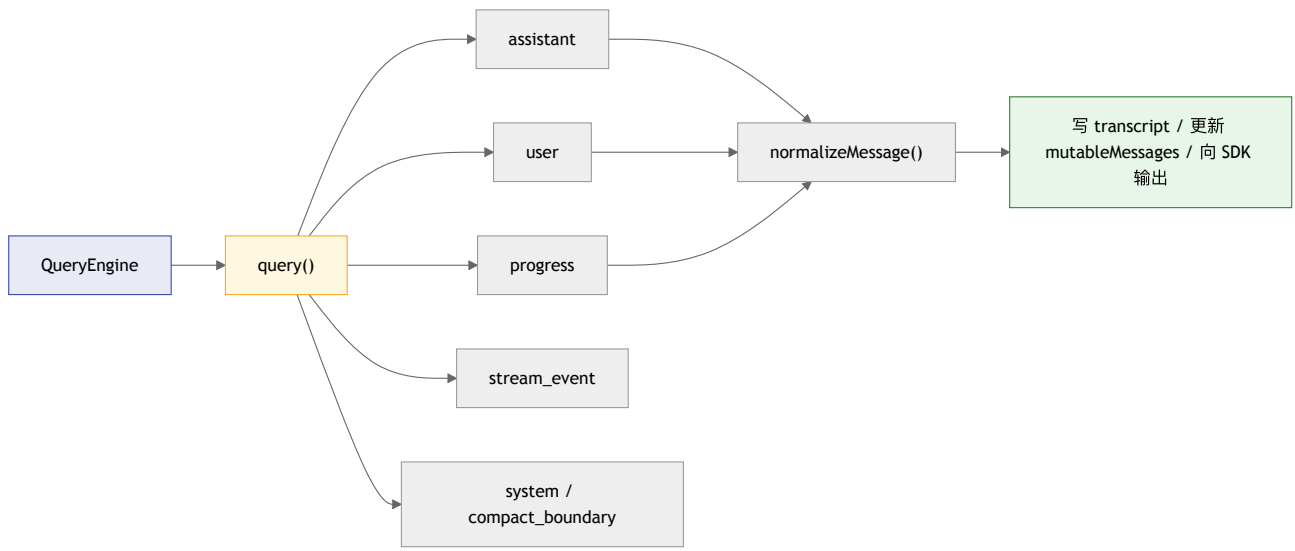
10.3 真正的核心动作：把上下文交给 `query()`，再把结果重新组织回来

`QueryEngine` 最关键的一步发生在 `:675-686`：

它把下面这些东西统一交给 `query()`：

- `messages`
- `systemPrompt`
- `userContext`
- `systemContext`
- `wrappedCanUseTool`
- `toolUseContext`
- `fallbackModel`
- `maxTurns`
- `taskBudget`

然后它自己不直接“计算答案”，而是**消费 `query()` 这个异步生成器吐出来的各种消息**。



为什么这样拆层很聪明

因为 `query()` 专注于：

- Agent Loop
- 模型流式调用
- 工具结果回填
- 停止条件判断

而 `QueryEngine` 专注于：

- 会话级状态保存
- transcript 记录
- SDK 输出格式
- 总成本和总 usage 累计

这就像厨房里把“炒菜的人”和“传菜结账的人”分开。一个负责把菜做好，另一个负责保证整顿饭的秩序。

`QueryEngine` 处理的消息类型比你想的多

它不仅处理：

- `assistant`
- `user`

还处理：

- `progress`
- `stream_event`
- `attachment`
- `system`
- `tool_use_summary`
- `tombstone`

这说明 AI 会话在工程上根本不只是“用户消息”和“模型回复”两个对象，而是一整套事件流。

源码证据

- `OpenClaudeCode/src/QueryEngine.ts:675-686`：把完整上下文交给 `query()`
- `OpenClaudeCode/src/QueryEngine.ts:687-970`：对各类消息做 transcript、归一化和 SDK 输出处理

10.4 为什么 headless / SDK 能复用同一颗核心

`QueryEngine` 的类注释里有一句很关键的话：

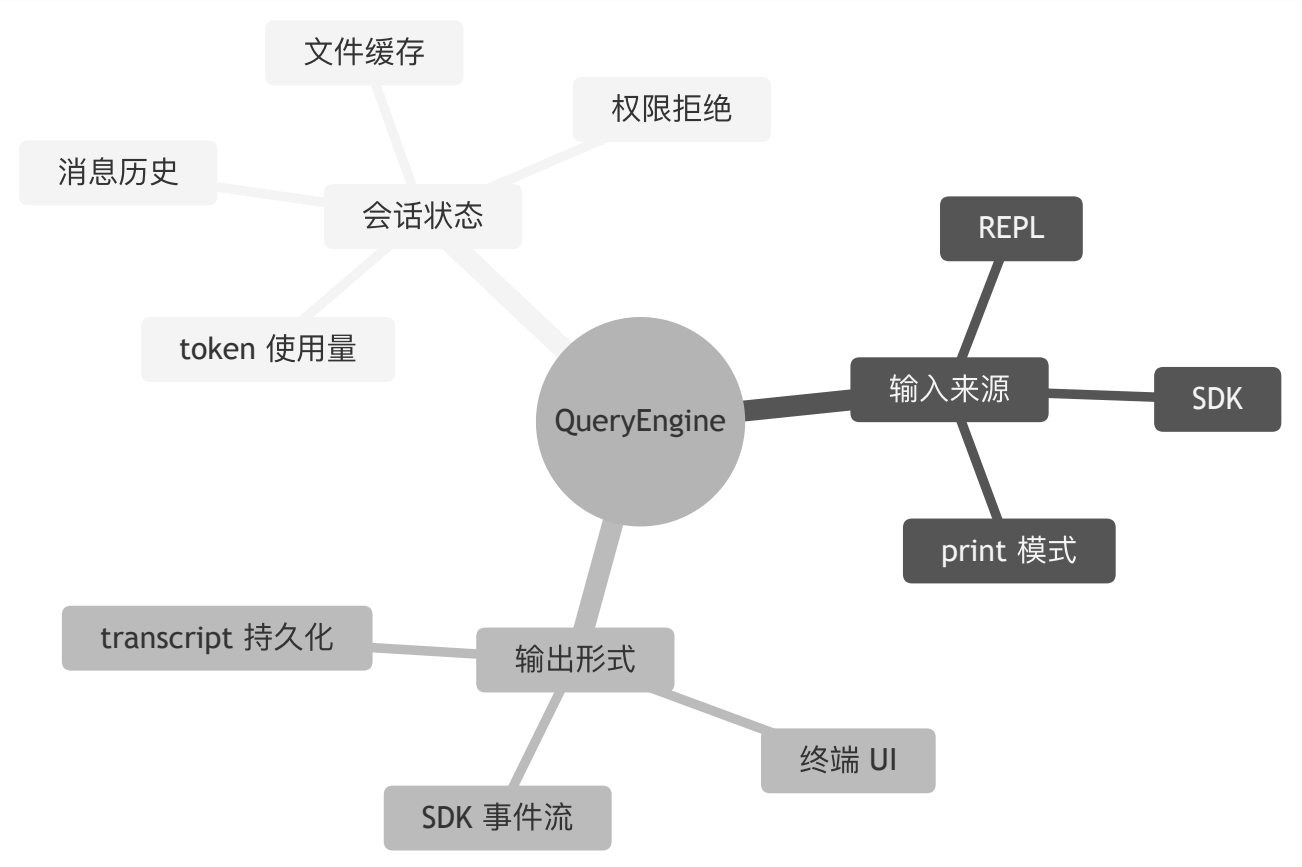
> 它被抽出来，是为了同时服务 headless / SDK 路径，并在未来逐步服务 REPL。

换句话说，Claude Code 的设计方向不是：

- REPL 一套逻辑
- SDK 再写一套类似逻辑

而是：

- 把真正通用的“会话核心”抽出来
- UI 只是壳
- SDK 只是另一种输出方式



这对初学者意味着什么

当你以后自己做 AI 应用时，不要一上来把所有逻辑糊在页面组件里。更好的方式是先问自己：

- 哪部分是“会话核心”？
- 哪部分只是“展示层”？
- 哪部分是“对外接口层”？

Claude Code 的 QueryEngine 正是在回答这个问题。

这对架构师意味着什么

一旦会话核心和 UI 解耦，很多能力就自然出来了：

- CLI 可以复用
- SDK 可以复用
- 远程桥接可以复用
- 历史恢复可以复用

换句话说，**它让 Claude Code 从“一个终端程序”变成了“一套会话内核”。**

深水区（架构师选读）

QueryEngine 最妙的地方，不是它做了很多事，而是它**管住了边界**：

- 它不自己写 UI
- 不自己实现工具
- 不自己直连具体输入控件
- 也不把 transcript 逻辑散落到每个组件

它只做一件大事：**把一次次用户输入，组织成可持续、多轮、可恢复、可统计的会话。**

这就是很多 AI demo 和 AI 产品之间的分水岭。demo 只有“一次回答”，产品必须有“会话内核”。

本章小结

一句话：`QueryEngine` 不是一个简单的模型调用器，而是 Claude Code 的会话中枢，负责组织上下文、驱动 `query()`、积累历史与成本，并把同一套核心能力同时服务给 REPL 和 SDK。

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/QueryEngine.ts:176-206`	QueryEngine 的职责与持久状态
补全层	`OpenClaudeCode/src/QueryEngine.ts:209-325`	`submitMessage()` 的前置编排
补全层	`OpenClaudeCode/src/QueryEngine.ts:675-686`	把完整上下文交给 `query()`
补全层	`OpenClaudeCode/src/QueryEngine.ts:687-970`	对流式结果、进度、系统消息的统一处理
补全层	`OpenClaudeCode/src/utils/queryContext.ts:29-73`	会话前缀上下文的构建来源

逆向提醒

- ✅ **可信度高**：QueryEngine 的定位、状态字段和 `submitMessage()` 主路径都能直接从源码读出来
- ⚠️ **要注意边界**：它不是单独完成所有工作，而是把具体推理循环委托给 `query()`
- ❌ **不要误解**：`QueryEngine` 不是“给 REPL 专门做的类”，它的设计目标从一开始就是 headless / SDK 复用

16

Agent Loop 第三编

第11章：Agent Loop：AI工作的核心循环

生活类比

厨师做菜不是“看一眼菜单，然后瞬间端菜上桌”。真实过程是：看菜单、拿食材、下锅、尝味道、不对就调整，再继续。**Claude Code 的 Agent 也是这样工作的：想一步、做一步、看结果、再决定下一步。**

这一章要回答的问题

一个 Agent 到底是靠什么“自己持续干活”的？真的是一个 `while(true)` 吗？

如果你把 Claude Code 想成“模型回一次话就结束”，你会完全看不懂它为什么能读文件、跑命令、修完 bug 再回来汇报。真正让它动起来的，是 `query.ts` 里那套持续推进任务的循环。

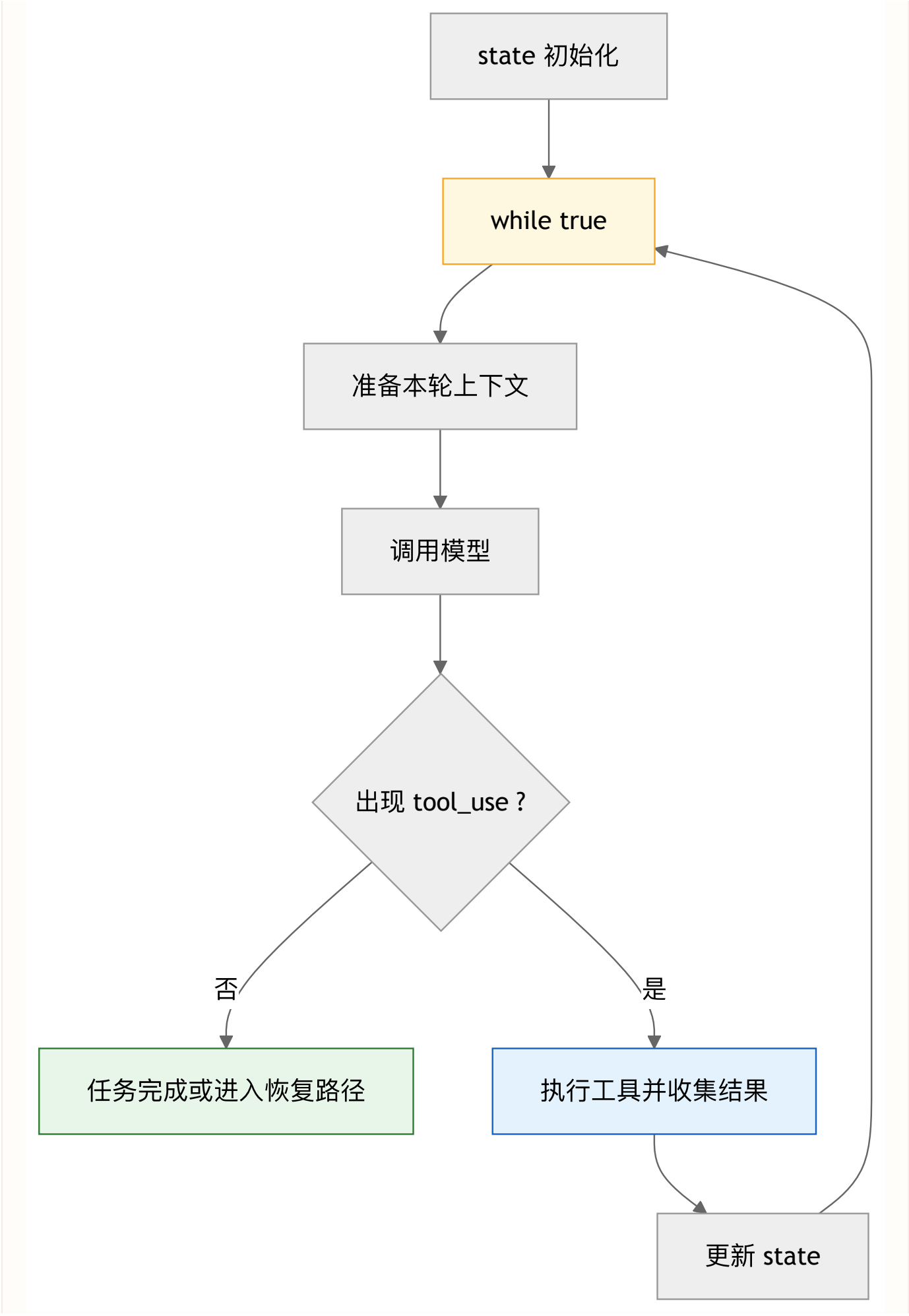
11.1 概念上像 ReAct，源码里真的有一个 `while (true)`

很多资料会把 Agent Loop 讲成一句抽象口号：

> 思考 -> 行动 -> 观察 -> 再思考

Claude Code 没停留在概念层。`query.ts` 里真的定义了一个 `State`，然后在 `queryLoop()` 中进入：

```
// eslint-disable-next-line no-constant-condition
while (true) {
  ...
}
```


state 初始化
while true
准备本轮上下文
调用模型
出现 tool_use ?
否
是
任务完成或进入恢复路径
执行工具并收集结果
更新 state

这不是“简单循环”，而是“状态机循环”

State 里带着这些跨轮次信息：

- messages
- toolUseContext
- autoCompactTracking
- maxOutputTokensRecoveryCount
- hasAttemptedReactiveCompact
- pendingToolUseSummary
- turnCount
- transition

也就是说，每次循环不是“重新开始”，而是带着上一轮留下的状态继续前进。

taskBudgetRemaining 为什么不放进 State

源码注释还专门写了：taskBudgetRemaining 不放在 State 里，是为了不去污染多个 continue 路径。这个细节很像资深工程师会做的事：

- 需要跨迭代保留
- 但不值得塞进每次状态转移对象里

这说明 Claude Code 的循环实现，已经在认真处理**状态边界和维护成本**。

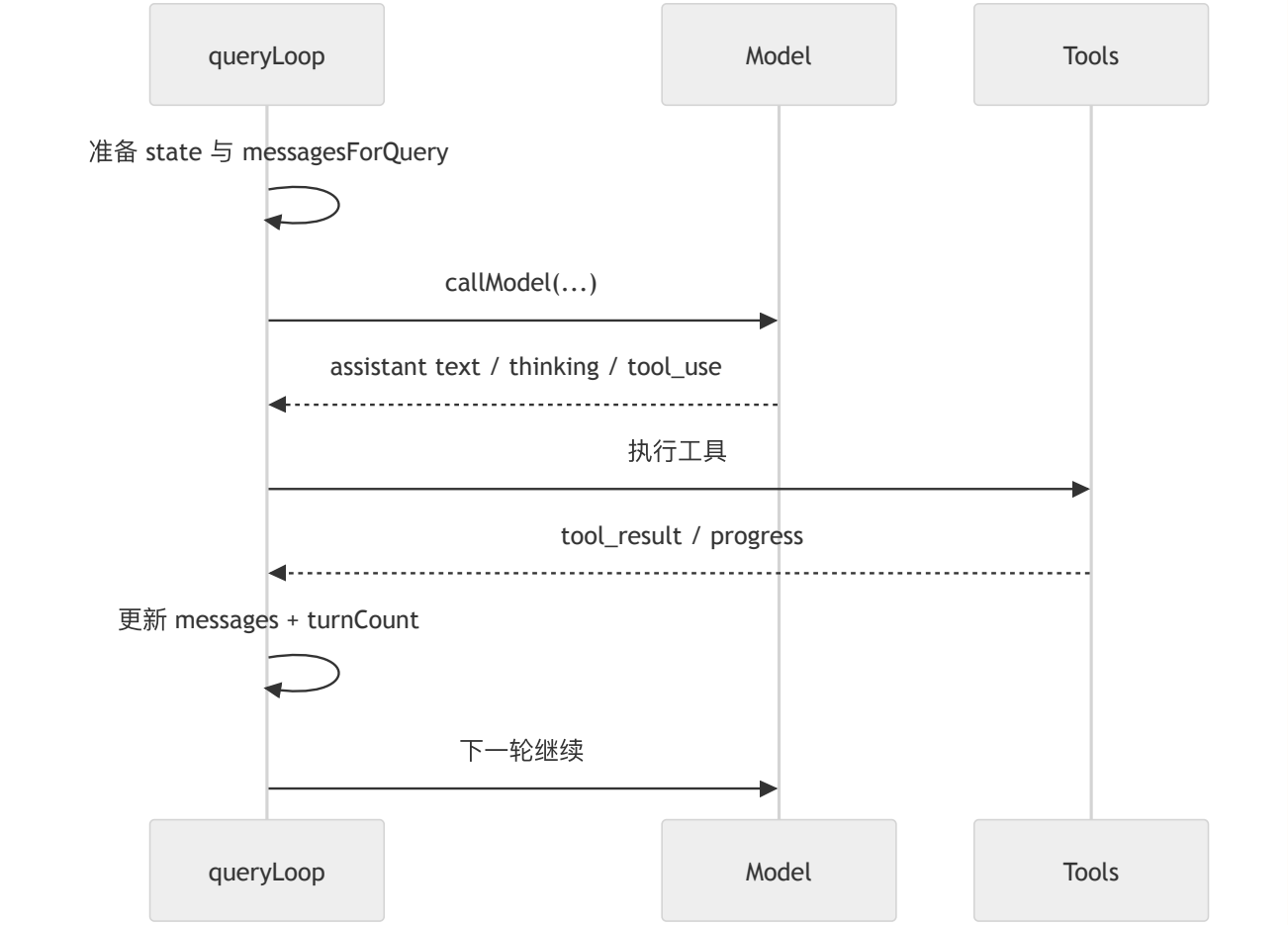
源码证据

OpenClaudeCode/src/query.ts:181-307 定义了 QueryParams、State 与 queryLoop() 的 while (true) 主循环。

11.2 一轮循环里真正发生了什么

进入每一轮以后，Claude Code 大致按这个顺序做事：

1. 准备当前 messagesForQuery
2. 初始化 assistantMessages、toolResults、toolUseBlocks
3. 如果开启流式工具执行，就创建 StreamingToolExecutor
4. 先检查是否已经触到上下文 blocking limit
5. 调用模型开始流式返回
6. 收集文本、thinking、tool_use
7. 若有工具结果，拼回消息历史，进入下一轮
8. 若没有工具调用，任务可能完成



一个极其关键的判断：不能只信 `stop_reason === 'tool_use'`

源码里有一句非常值钱的注释：

```
stop_reason === 'tool_use' is unreliable
```

所以 Claude Code 真正依赖的不是 stop reason，而是：

- 在流式过程中有没有收到 `tool_use` block
- 如果收到了，就把它们放进 `toolUseBlocks`
- 用 `needsFollowUp = true` 作为是否继续循环的信号

这非常像做后端工程时的经验法则：

- 不要盲信一个看起来方便但不稳定的字段
- 要自己根据更可靠的底层证据建判断条件

流式工具执行器在这里登场

如果开了 `streamingToolExecution`，`query.ts` 会在拿到 `tool_use` block 的同时，把它交给 `StreamingToolExecutor`。这意味着：

- 模型还在流式吐内容
- 工具已经可以提前排队，甚至开始跑

这就是 Claude Code“看起来特别流畅”的关键之一。

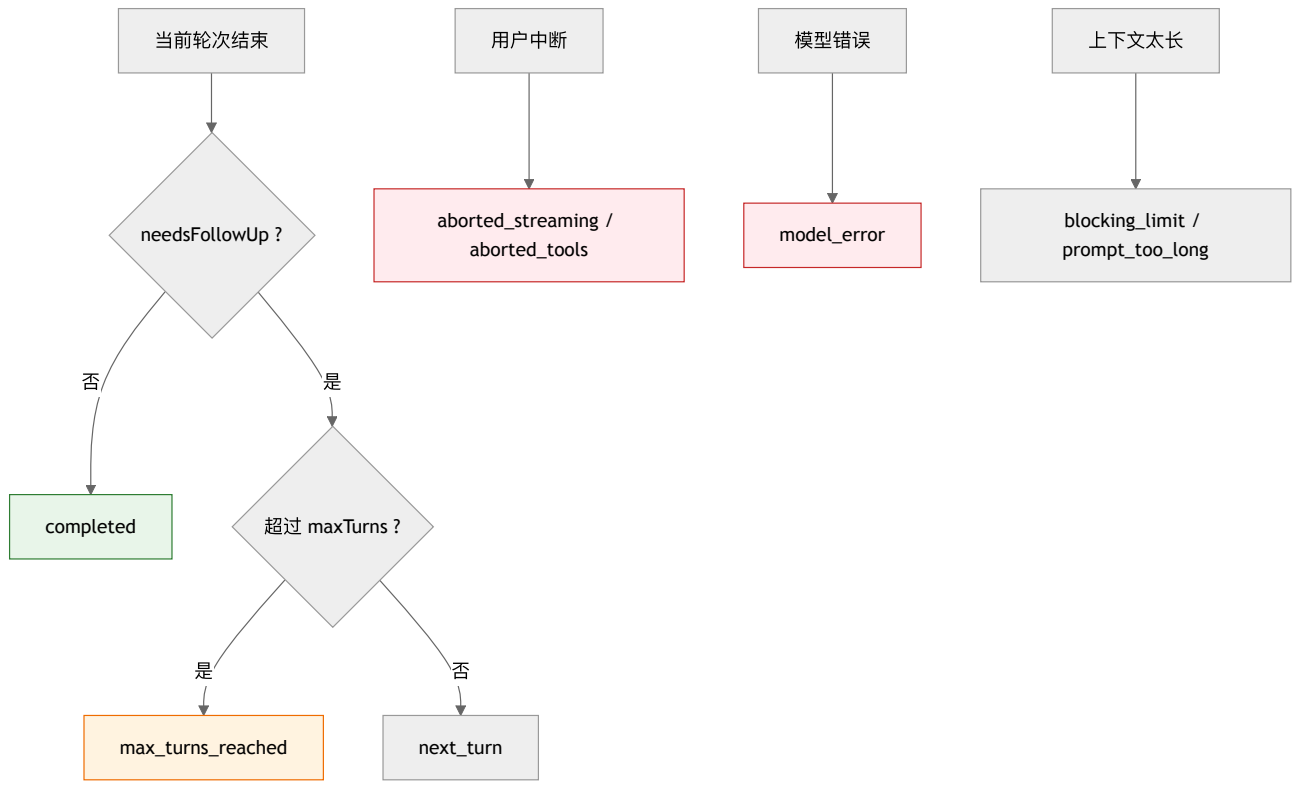
源码证据

`OpenClaudeCode/src/query.ts:545-568` 初始化每轮核心容器；`OpenClaudeCode/src/query.ts:653-708` 调模型；`OpenClaudeCode/src/query.ts:821-852` 在流式过程中收集工具块并与执行器联动。

11.3 循环什么时候停：五种典型出口

一个真正可靠的 Agent Loop，不难在“会继续”，难在“知道什么时候该停”。

Claude Code 主要有几类出口：



1. 没有后续工具调用，正常完成

如果本轮流式结束后 `needsFollowUp` 仍然是 `false`，就说明模型没有再请求工具，循环可以走向完成或进入恢复判断。

2. 达到 `maxTurns`

当有工具结果、准备递归进入下一轮时，代码会先算：

- `nextTurnCount = turnCount + 1`
- 如果超过 `maxTurns`
- 直接产出 `max_turns_reached` 附件并返回

甚至连“用户在工具执行阶段中断了”的路径，也会顺手检查 maxTurns，避免系统在中断情况下丢失这个限制信号。

3. 用户中断

当 `abortController.signal.aborted` 触发时：

- 如果用了 `StreamingToolExecutor`，要先消费剩余结果，补齐必要的 synthetic tool_result
- 然后再发中断消息
- 最后返回 `aborted_streaming` 或 `aborted_tools`

这说明中断不是“粗暴停机”，而是**先把会话补到可恢复的一致状态**。

4. 模型错误

如果 `callModel` 真抛出异常，`query.ts` 会：

- `yieldMissingToolResultBlocks(...)`
- 再给用户一个真实 API error
- 返回 `model_error`

也就是说，即使模型在半路坏掉，Claude Code 也会尽量补上那些已经发出 `tool_use` 但还没等到 `tool_result` 的空洞。

5. 上下文过长与恢复失败

如果 token 早已到了 blocking limit，Claude Code 会直接生成 `PROMPT_TOO_LONG_ERROR_MESSAGE` 返回，而不会傻傻继续打 API。

源码证据

- `OpenClaudeCode/src/query.ts:637-646`：blocking limit 直接阻断
- `OpenClaudeCode/src/query.ts:1498-1515`：中断工具后仍然检查 `maxTurns`
- `OpenClaudeCode/src/query.ts:1678-1711`：递归进入下一轮前检查 `maxTurns`

- `OpenClaudeCode/src/query.ts:977-996`：模型错误时补齐缺失的 tool result

11.4 为什么这个循环不会轻易失控

“AI 会不会陷入死循环？”这是最自然的担忧。Claude Code 的回答不是一句“不会”，而是一层层保险。

保险一：阻断过长上下文

在每轮正式打模型之前，它会先用 `calculateTokenWarningState(...)` 检查是否已经到 blocking limit。

这相当于厨师上菜前先看煤气还够不够，别做到一半锅都要熄火了。

保险二：Fallback 不是简单重试，而是会清理“半截消息”

如果流式过程中触发 fallback model：

- 之前已经出来的半截 assistant message 会先 tombstone 掉
- `StreamingToolExecutor` 会 discard
- `assistantMessages / toolResults / toolUseBlocks` 都会被清空

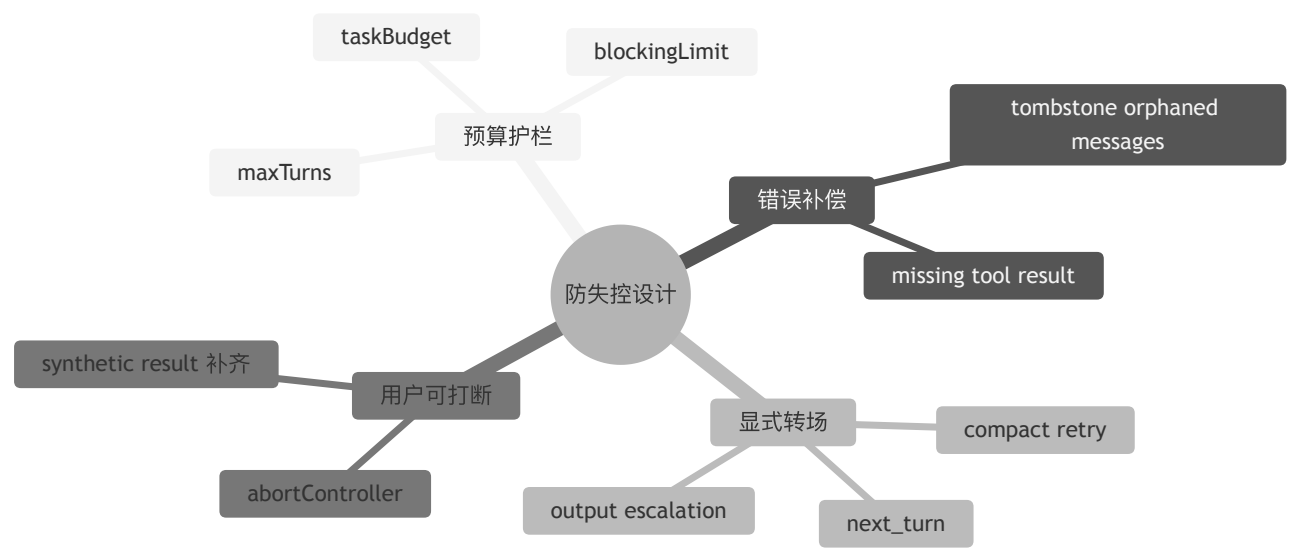
这一步特别讲究，因为“半截 thinking block”或“旧 tool_use_id 对应的新 tool_result”都会让后续对话历史变脏。

保险三：转场理由是显式记录的

`State.transition` 不是装饰字段，它会清楚记录这次继续是因为：

- `next_turn`
- `reactive_compact_retry`
- `max_output_tokens_escalate`
- `token_budget_continuation`

这让测试和调试都能判断：系统究竟是为什么继续跑下去的。



最值得记住的工程思想

Claude Code 的 Agent Loop 之所以稳，不是因为“模型很聪明”，而是因为循环外面和里面都围了很多护栏：

- 预算护栏
- 上下文护栏
- 一致性护栏
- 中断护栏

这也是你自己做 Agent 时最该学的地方。

🔭 深水区（架构师选读）

第 11 章最有价值的一点，是它把“Agent Loop”从一个概念词，变成了一套**可调试、可约束、可恢复的循环控制器**。

很多人讲 ReAct，会把重点放在“模型会想，会做”。Claude Code 源码告诉我们，真正难的是：

- 什么时候继续
- 什么时候终止
- 错一半时怎么补齐
- 中断时怎么不把历史搞脏
- fallback 时怎么不留下孤儿消息

这就是为什么 AI Agent 的核心难题，不是“会不会生成”，而是“能不能稳定运行”。

本章小结

一句话：Claude Code 的 Agent Loop 真的是一个 `while (true)` 驱动的多轮状态机，它通过 `tool_use` 检测、显式转场、预算限制、错误补偿和中断补齐，把“想 -> 做 -> 看 -> 再来”这件事做成了可靠工程。

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/query.ts:181-307`	`QueryParams`、`State` 与 `while (true)` 主循环
补全层	`OpenClaudeCode/src/query.ts:545-568`	每轮循环的核心容器初始化
补全层	`OpenClaudeCode/src/query.ts:637-708`	blocking limit 检查与模型调用
补全层	`OpenClaudeCode/src/query.ts:821-852`	识别 `tool_use` 并联动流式工具执行
补全层	`OpenClaudeCode/src/query.ts:977-1051`	模型错误和中断时的补偿逻辑
补全层	`OpenClaudeCode/src/query.ts:1498-1515`	中断路径上的 `maxTurns` 处理
补全层	`OpenClaudeCode/src/query.ts:1678-1725`	递归进入下一轮前的转场与终止判断

逆向提醒

- ✅ **可信度高**：`while (true)`、`State`、`needsFollowUp`、`maxTurns` 和各类 return reason 都是源码直出
- ⚠️ **需要纠正旧印象**：它不只是“一个 while 循环”，更是“状态机 + 恢复路径 + 护栏系统”
- ❌ **不要误读**：继续下一轮的判断不是单纯靠 `stop_reason === tool_use`，源码明确说这个字段并不可靠

17

流式响应 第三编

第12章：流式响应：为什么回复一个字一个字蹦出来

生活类比

你口渴的时候，希望得到的是“马上拧开水龙头就能喝到第一口水”，而不是“等水桶装满 30 秒再一起倒给你”。**流式响应就是 AI 世界的水龙头。**

这一章要回答的问题

如果等 AI 全部生成完再显示，会发生什么？Claude Code 又是怎么把“模型内部正在生成的半成品”安全地变成终端上的实时输出的？

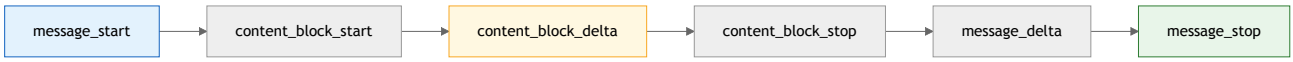
这一章最关键的一点是：流式响应不是“界面好看一点”的小优化，而是 Claude Code 整个体验和工具编排速度的核心基础设施。

12.1 SSE 事件在 `claude.ts` 里被拆成了可持续生长的内容块

真正处理流式事件的关键逻辑在 `services/api/claude.ts`。

它收到的不是“一整条最终消息”，而是一连串事件：

- `message_start`
- `content_block_start`
- `content_block_delta`
- `content_block_stop`
- `message_delta`
- `message_stop`



`message_start` 先定框架

`message_start` 到来时，代码会：

- 记录 `partialMessage`
- 计算 TTFT（首字节时间）
- 累加初始 usage

这就像直播开始前，先把摄像机、时间戳、计数器都架好。

`content_block_start` 按块开槽位

Claude Code 不会假设“接下来全是普通文本”。它会根据块类型分别开不同槽位：

- `tool_use`
- `server_tool_use`
- `text`
- `thinking`

而且初始化方式不同：

- `tool_use.input` 先设为空字符串
- `text.text` 先设为空
- `thinking.thinking` 和 `signature` 分开攒

这很像仓库管理员先把不同货架分好：生鲜、文具、易碎品不能全堆一块。

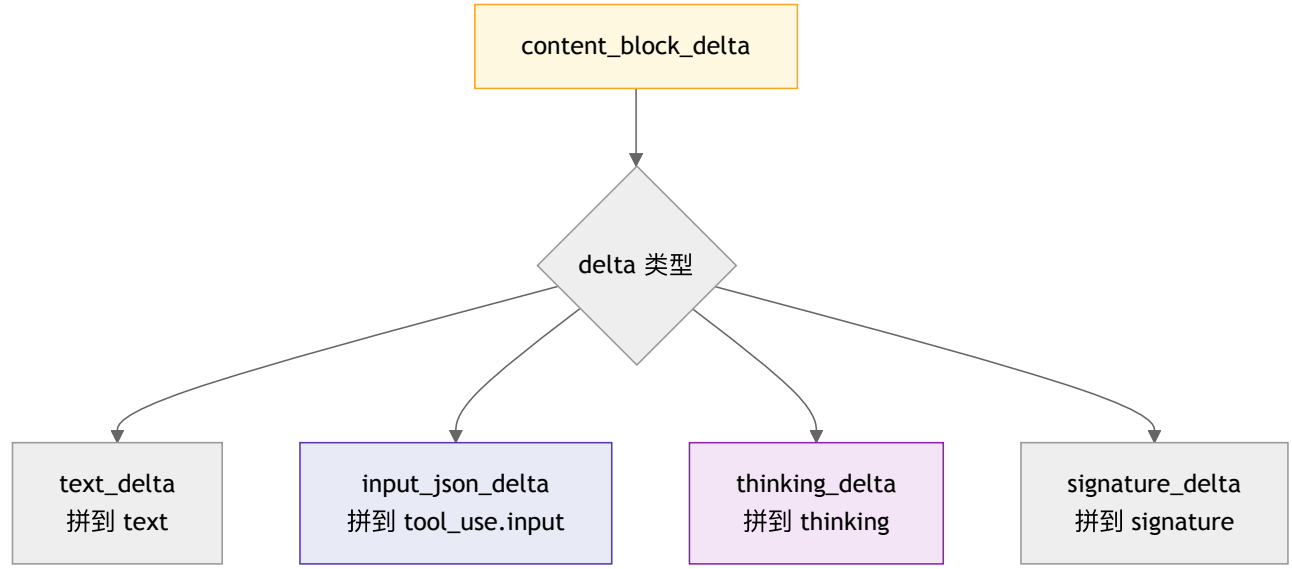
`content_block_delta` 才是真正一点点长出来的部分

源码里分别处理了：

- `input_json_delta`
- `text_delta`
- `thinking_delta`
- `signature_delta`

也就是说，Claude Code 不只是“边收到边显示文字”，它还在流式拼：

- 工具输入 JSON
- thinking 内容
- thinking 签名



源码证据

`OpenClaudeCode/src/services/api/claude.ts:1980-2169` 详细展示了各种流式事件和 block delta 的拼接逻辑。

12.2 一条 assistant message 其实是“分块结束时”才真正成形的

很多人以为模型每吐一个字，Claude Code 就立刻创建一条完整 assistant message。其实不是。

真正的 assistant message 出现在 `content_block_stop`：

- 取出当前 block
- `normalizeContentFromAPI(...)`
- 生成标准化后的 `AssistantMessage`
- `yield m`

也就是说，**Claude Code 不是按“字符”建消息，而是按“内容块”建消息。**

为什么这样做更稳

因为不同块的语义不同：

- 文本块可以直接显示
- thinking 块要特殊处理
- tool_use 块要等 JSON 足够完整

如果不按块，而是按字符生硬推给上层，整个系统会非常混乱。

`message_delta` 还要回头补 usage 和 stop_reason

更微妙的是：assistant message 在 `content_block_stop` 就 yield 了，但最终：

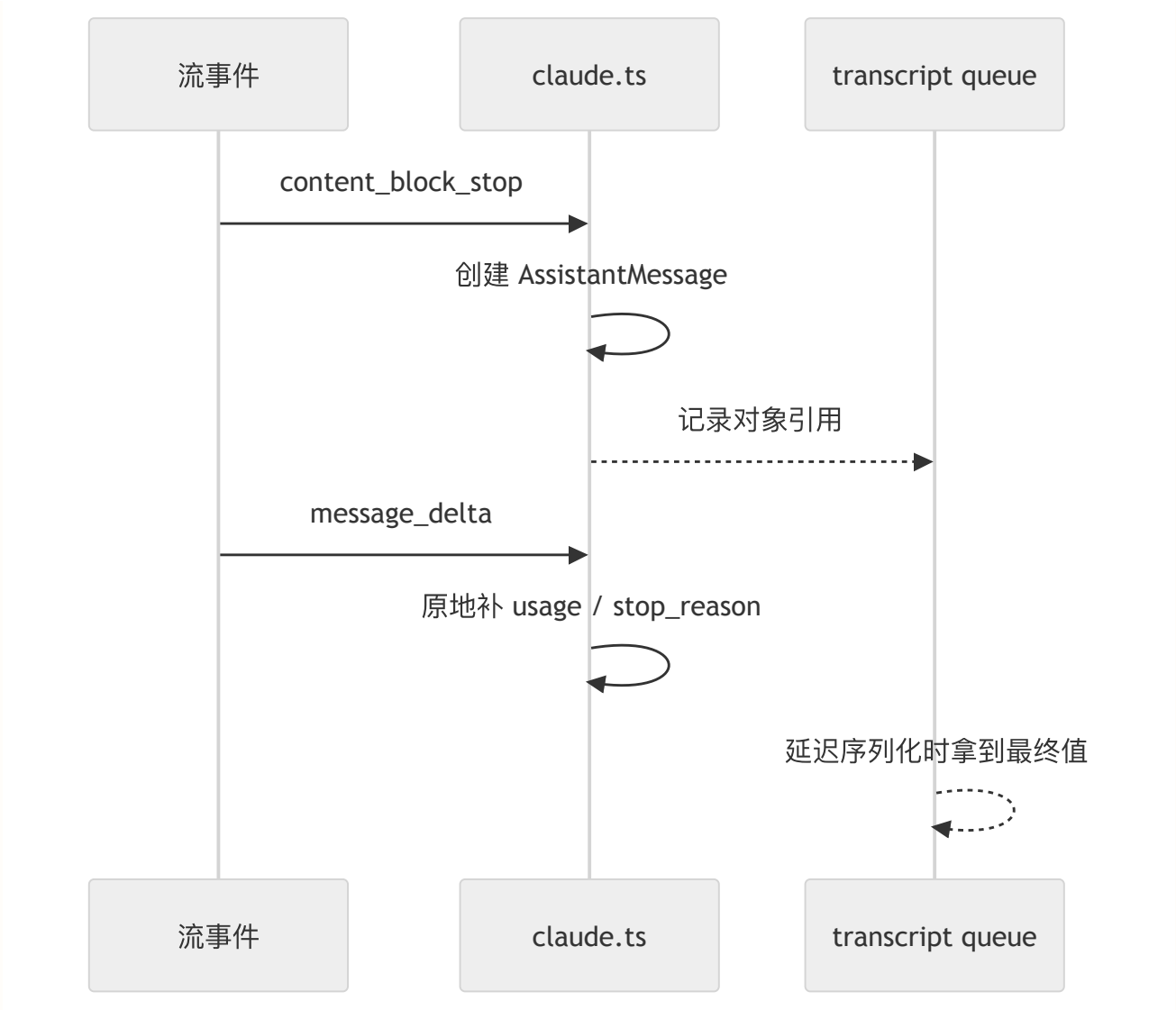
- `usage`
- `stop_reason`

这两个关键信息却是在后面的 `message_delta` 里才到。

所以源码做了一件很“工程脑”的事：

- 直接**原地修改**最后那条已创建消息的字段
- 而不是重新创建一个新对象

原因注释写得特别清楚：transcript 写队列拿的是对象引用，如果这里替换对象，延迟落盘时会拿到旧对象。



这类细节特别适合学习：真正稳定的流式系统，很多时候胜负就在这些“对象什么时候生成、什么时候补全”的边界处理上。

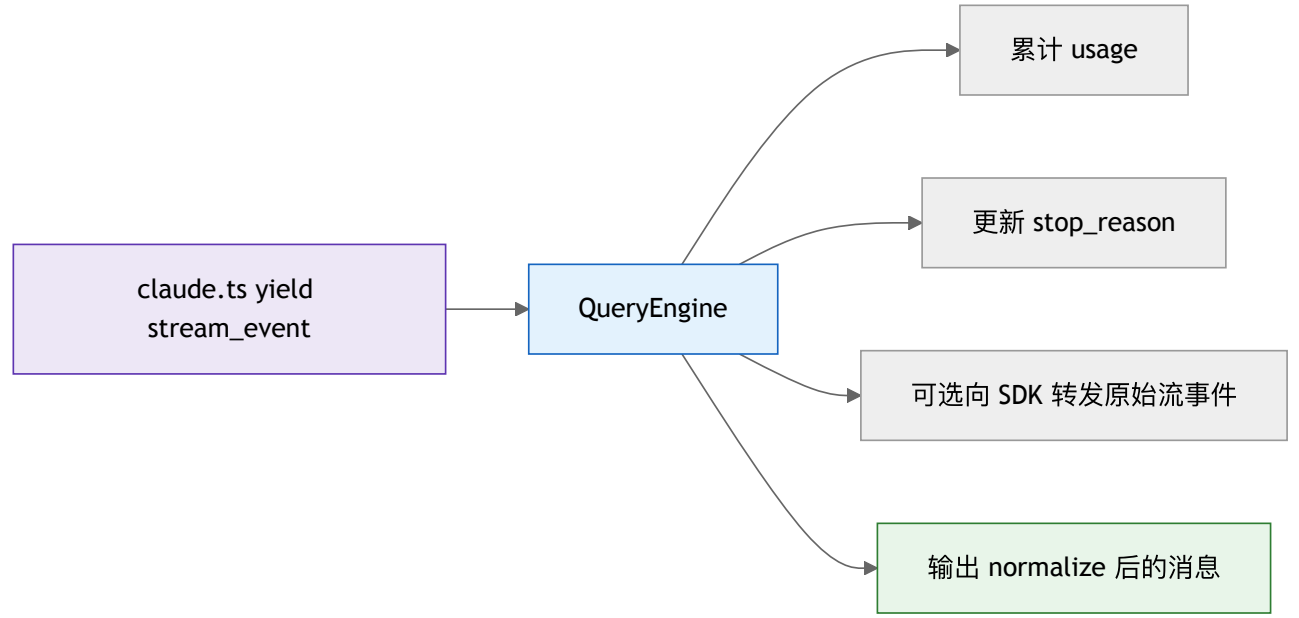
源码证据

`OpenClaudeCode/src/services/api/claude.ts:2171-2248` 展示了 `content_block_stop` 产出消息，以及 `message_delta` 回填 `usage / stop_reason` 的过程。

12.3 QueryEngine 一边吃事件，一边把它们变成用户看得懂的东西

流式事件不是直接“裸奔到终端”的。`QueryEngine` 会再接一层：

- 在 `message_start` 时重置当前消息 usage
- 在 `message_delta` 时累积 usage 并记录 stop_reason
- 在 `message_stop` 时把当前 usage 累加到总 usage
- 如果 `includePartialMessages` 开启，还能把原始 `stream_event` 再往外吐给 SDK



这说明了一个重要架构原则

`claude.ts` 关心的是“怎么从 API 流里拼出正确内容”。
`QueryEngine` 关心的是“怎么把这些内容纳入整场会话的账本和 transcript”。

职责非常清楚：

- 下层负责还原真实流
- 上层负责管理会话语义

为什么 `stream_event` 也值得保留

因为对 SDK 场景来说，有时候调用者并不只想拿“最终文字”，而是想自己做：

- 实时进度 UI
- 调试面板
- streaming analytics
- 自定义转译层

如果底层不保留这些原始事件，上层扩展空间就会大大变小。

源码证据

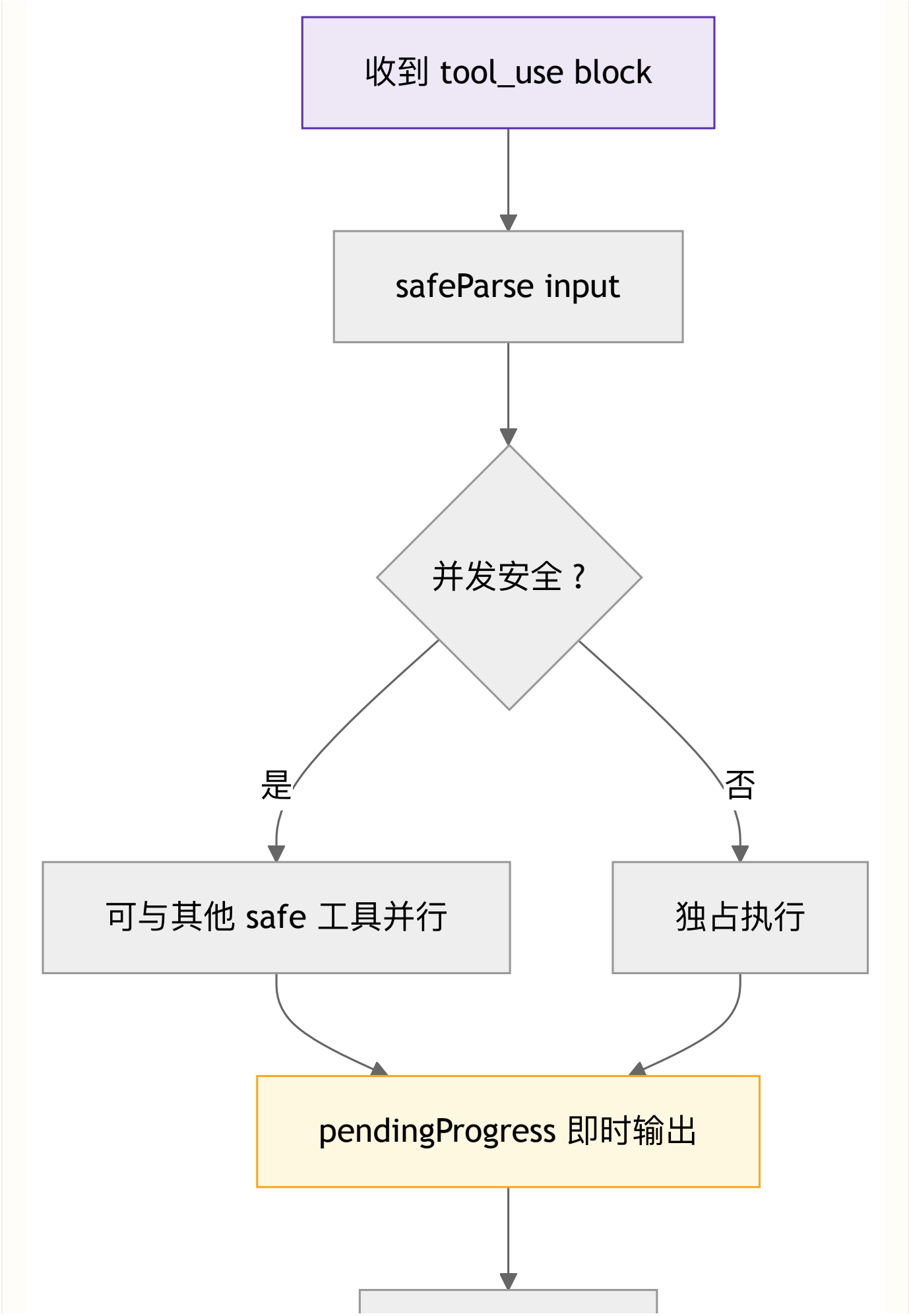
`OpenClaudeCode/src/QueryEngine.ts:788-826` 展示了 QueryEngine 如何吸收并转发 `stream_event`。

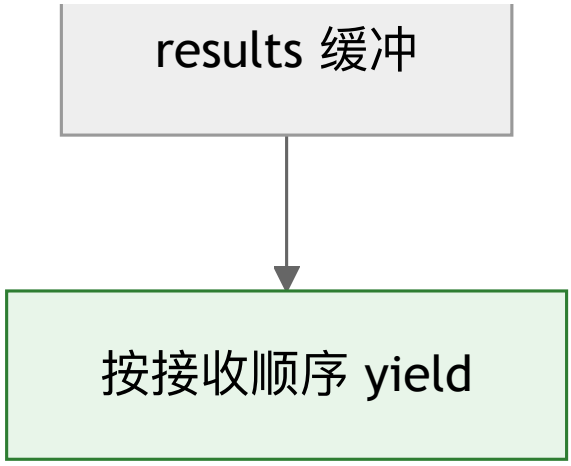
12.4 工具为什么也能“边流边准备”：`StreamingToolExecutor`

Claude Code 的流式体验之所以高级，不是因为它只会一边吐文字，而是因为工具调用也被接到了这条流式链上。

`StreamingToolExecutor` 的职责是：

- 工具块一出现就排队
- 能并发的工具并发执行
- 不能并发的工具独占执行
- 结果按收到顺序吐回
- 进度消息提前显示


收到 tool_use block
safeParse input
并发安全？
是
否
可与其他 safe 工具并行
独占执行
pendingProgress 即时输出



并发不是乱跑，而是带约束的

`StreamingToolExecutor` 不会粗暴地把所有工具都扔进 `Promise.all`。

它先问：

- 输入能不能通过 schema
- 这个工具是不是 `isConcurrencySafe`
- 当前有没有别的非并发安全工具在执行

这很像高速公路收费站：

- 小轿车可以多车道并行
- 超宽货车要单独走

中断和 fallback 也有自己的“善后逻辑”

如果遇到：

- `user_interrupted`
- `streaming_fallback`
- `sibling_error`

`StreamingToolExecutor` 不会静默丢掉，而是会构造 synthetic error message 补齐 `tool_result`。

这意味着即使在流式过程中出事故，消息历史仍然是闭环的。

为什么 Bash 出错会中断兄弟工具

源码里有个非常真实的经验判断：

- 只有 Bash 工具出错时，才会主动取消兄弟工具
- 因为 Bash 命令常常有隐含依赖链

比如：

- `mkdir` 失败了
- 后面的 `cd`、`ls`、`npm test` 多半也没意义

但像 WebFetch、ReadFile 这类工具，彼此失败往往没有因果关系，就没必要全盘中止。

源码证据

- `OpenClaudeCode/src/services/tools/StreamingToolExecutor.ts:34-151`：工具排队与并发规则
- `OpenClaudeCode/src/services/tools/StreamingToolExecutor.ts:153-259`：中断原因与 synthetic error
- `OpenClaudeCode/src/services/tools/StreamingToolExecutor.ts:261-347`：执行、进度输出和剩余结果消费

🔭 深水区（架构师选读）

流式响应最容易被低估的地方，是大家往往只把它看成“更快显示首字”的 UI 优化。Claude Code 的源码告诉我们，真正的流式系统至少有三层：

1. **协议层流式**：SSE 事件如何被拆分和累积
2. **语义层流式**：什么时候才能说“一条消息已经成立”
3. **工具层流式**：工具能不能在模型还没说完前就提前进入执行状态

只有三层一起做好，用户才会觉得它“聪明、快、顺”。否则要么是假流式，要么只是闪烁着输出几个字，但系统本体仍然很迟钝。

本章小结

一句话：Claude Code 的流式响应不是简单地“把文字一点点打印出来”，而是从 SSE 事件、消息块拼装、usage 回填，到工具并发和中断补偿，全链路都做成了可持续流动的系统。

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/services/api/claude.ts:1980-2169`	`message_start / content_block_delta` 等流式事件处理
补全层	`OpenClaudeCode/src/services/api/claude.ts:2171-2248`	在块结束时创建消息，并在 `message_delta` 回填 usage
补全层	`OpenClaudeCode/src/services/api/claude.ts:2299-2303`	向上层统一吐出 `stream_event`
补全层	`OpenClaudeCode/src/QueryEngine.ts:788-826`	QueryEngine 对流式事件的会话级处理
补全层	`OpenClaudeCode/src/services/tools/StreamingToolExecutor.ts:34-151`	工具的流式排队与并发
补全层	`OpenClaudeCode/src/services/tools/StreamingToolExecutor.ts:153-259`	中断、fallback、兄弟工具取消策略

逆向提醒

- ✅ **可信度高**：SSE 事件形态、消息分块、tool stream 执行器都能在源码中直接看到
- ⚠️ **要分清层次**：assistant message 的创建时机和 `usage / stop_reason` 的最终到达时机不是同一个瞬间
- ❌ **不要误解**：流式不只是前端表现层；工具调度和 transcript 一致性也深度依赖流式设计

18

Token经济学 第三编

第13章：Token经济学：每个字都有价格

生活类比

手机流量套餐很好理解：你不是“爱怎么用就怎么用”，而是在固定额度里分配给视频、聊天、下载、导航。**Claude Code 里的 token 也是预算，不只是技术指标，更是产品成本。**

这一章要回答的问题

Claude Code 怎么知道自己会不会“超套餐”？它又是怎么把静态规则、动态上下文、thinking、tool search 这些都装进有限预算里的？

如果不理解 token 经济学，你会把很多现象看成随机：为什么有时自动压缩？为什么 prompt 里要放边界标记？为什么任务预算要跨 compact 继续算？其实这些都和“钱”和“窗口”有关。

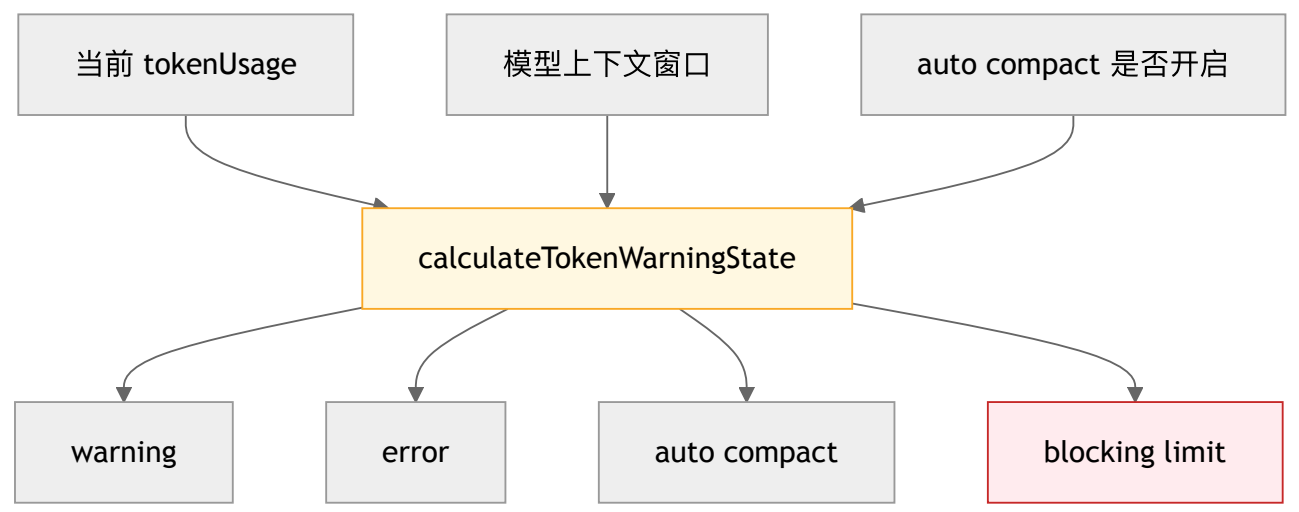
13.1 先算账：Claude Code 并不等到 API 报错才知道自己快满了

在 `services/compact/autoCompact.ts` 里，`calculateTokenWarningState(...)` 会根据：

- 当前 tokenUsage
- 模型上下文窗口
- 是否开启 auto compact

算出一组关键状态：

- `percentLeft`
- `isAboveWarningThreshold`
- `isAboveErrorThreshold`
- `isAboveAutoCompactThreshold`
- `isAtBlockingLimit`



这说明 Claude Code 的策略不是：

- “先猛跑”
- “撞墙以后再说”

而是：

- 提前估算
- 提前预警
- 真到危险线时再阻断

为什么 `blocking limit` 和 `auto compact threshold` 要分开

这两个阈值不是一回事：

- `auto compact threshold`：系统还想自己救一下

- `blocking limit`：再往前走就危险，必须停

换成生活类比：

- 油表见底灯亮了，不代表车马上趴窝
- 但真到完全没油，系统就该阻止你继续赌运气

`query.ts` 会在打 API 前先做 blocking 检查

如果当前上下文已经到了 blocking limit，`query.ts` 会直接返回 `PROMPT_TOO_LONG_ERROR_MESSAGE`，而不是发一笔大概率失败的请求出去。

这不是“小优化”，而是**真钱优化**。

源码证据

- `OpenClaudeCode/src/services/compact/autoCompact.ts:93-145`：warning / error / auto compact / blocking 四档判断
- `OpenClaudeCode/src/query.ts:628-646`：真正打模型前的 blocking limit 预检

13.2 为什么 prompt caching 能省钱：因为 prompt 被分成了静态和动态两半

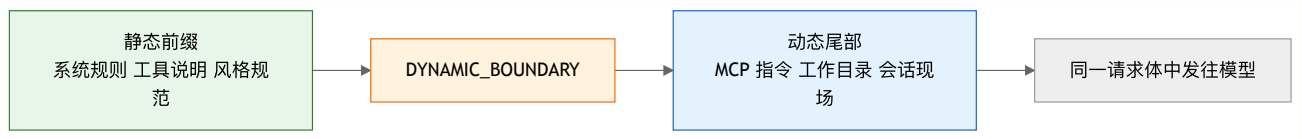
第 9 章讲过 `SYSTEM_PROMPT_DYNAMIC_BOUNDARY`，这里我们从“钱”的角度再看一次。

`constants/prompts.ts` 里专门定义了一个边界标记：

- 前面是静态、跨组织可缓存内容
- 后面是用户和会话相关动态内容

而在最终返回 system prompt 数组时，源码明确做了：

1. 静态 section
2. 边界 marker
3. 动态 section



这背后真正节省的是哪部分钱

多轮任务里最稳定的，往往不是用户消息，而是这些大块内容：

- 系统规则
- 工具说明
- 输出风格
- 安全要求

如果每一轮都把这些内容当成“全新输入”重新计算，会很浪费。
所以 Claude Code 不只是“写 prompt”，而是**把 prompt 做成缓存友好结构**。

为啥注释反复强调“不要随便移动边界”

因为一旦边界位置改了，相关缓存逻辑和 API 端的分割也得跟着改。
这就像数据库 schema 的分区键，不是你想换位置就换位置。

换句话说，这个字符串常量不是文案细节，而是**成本架构的一部分**。

源码证据

- `OpenClaudeCode/src/constants/prompts.ts:105-115`：动静态边界常量定义
- `OpenClaudeCode/src/constants/prompts.ts:560-576`：边界在最终 system prompt 里的插入位置

13.3 Task Budget：不只是“最多几轮”，而是“这次任务允许花多少 token”

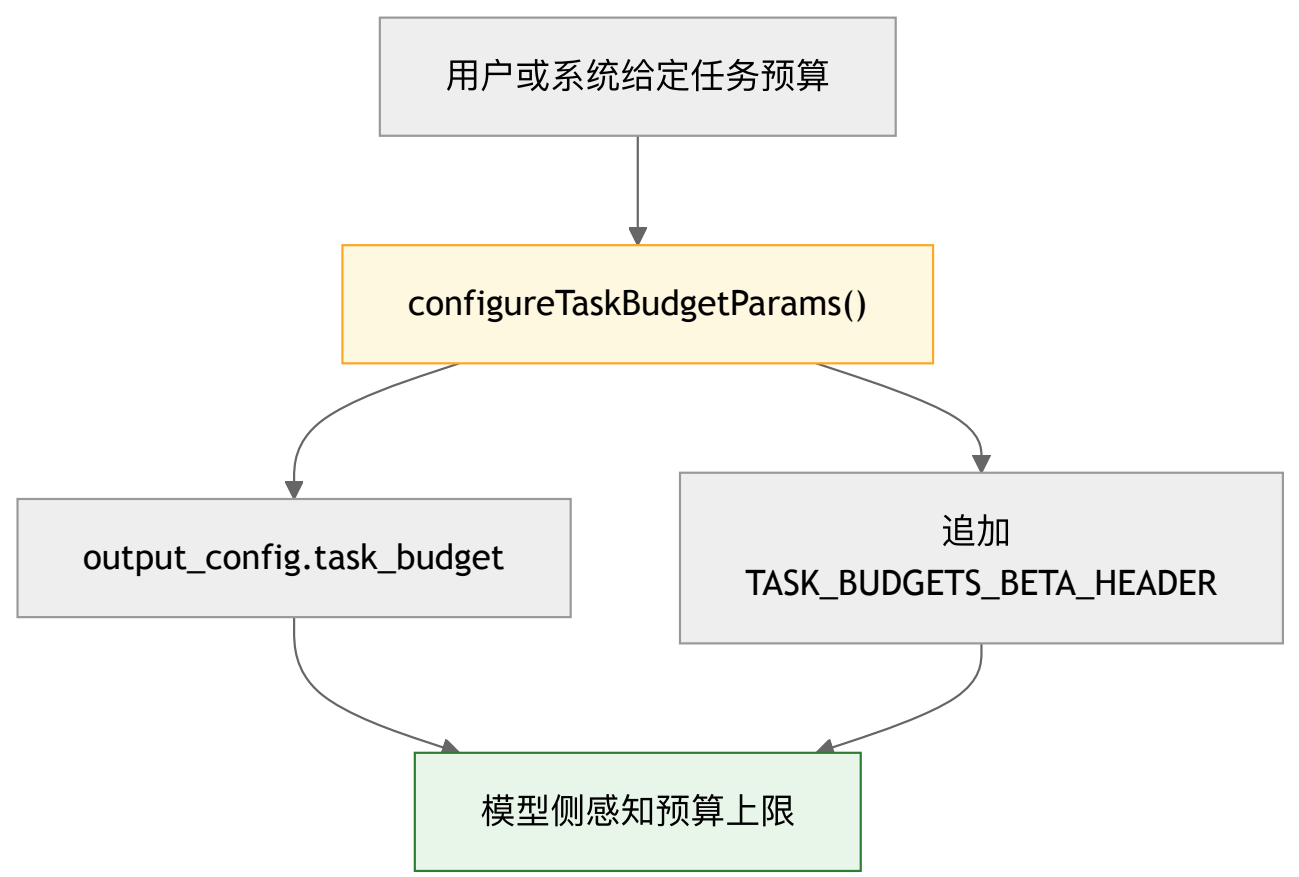
很多人会把 `maxTurns` 和预算混为一谈，其实不是。

- `maxTurns` 管的是循环次数
- `taskBudget` 管的是 token 预算

在 `services/api/claude.ts` 里，Claude Code 会把任务预算编码成：

- `type: 'tokens'`
- `total`
- `remaining`

并在需要时把 `TASK_BUDGETS_BETA_HEADER` 加进请求。



更高级的一点：预算会跨 compact 继续扣

`query.ts` 里有一段特别漂亮的注释：

- 如果 compact 发生了
- 服务器只看得到压缩后的摘要
- 它就会低估之前已经花掉的上下文

所以 Claude Code 自己维护了一个 `taskBudgetRemaining`：

- compact 前，服务器自己能看全量历史，先不用额外处理
- compact 后，客户端会把“被摘要掉的那部分上下文成本”继续从 remaining 里扣掉

这就像公司报销系统：你把明细装订成摘要，不代表财务可以假装前面那几页没花钱。

为什么这很重要

如果没有这层补偿，compact 会带来一个错觉：

- 上下文变短了
- 看起来预算又富裕了

但实际上，那只是“账单被折叠了”，不是“钱回来了”。

源码证据

- `OpenClaudeCode/src/services/api/claude.ts:473-500`：`task_budget` 的 API 参数编码
- `OpenClaudeCode/src/query.ts:282-291`：`taskBudgetRemaining` 的设计目的
- `OpenClaudeCode/src/query.ts:504-515`：compact 前上下文成本的扣减
- `OpenClaudeCode/src/query.ts:699-705`：把 total / remaining 带回模型调用
- `OpenClaudeCode/src/query.ts:1135-1145`：恢复路径下继续扣减 budget

13.4 成本不只是 token 数量，还受请求形态影响

到这里你可能会问：

> 既然都按 token 算，为什么还要讲 thinking、advisor、tool search、betas?

因为这些开关会改变**一次请求长什么样**，而请求形态又会影响 token 的花法。

Thinking 会占输出预算

在 `claude.ts` 里，如果 thinking 开启，系统会根据模型能力决定：

- 支持 adaptive 的模型：走 `type: 'adaptive'`
- 不支持 adaptive 的模型：走 `budget_tokens`

而且 thinking budget 还会被限制在 `maxOutputTokens - 1` 之内。

这说明“thinking 不是白送的”，它是输出预算的一部分。

Tool search 会带 beta header，还会动态筛工具

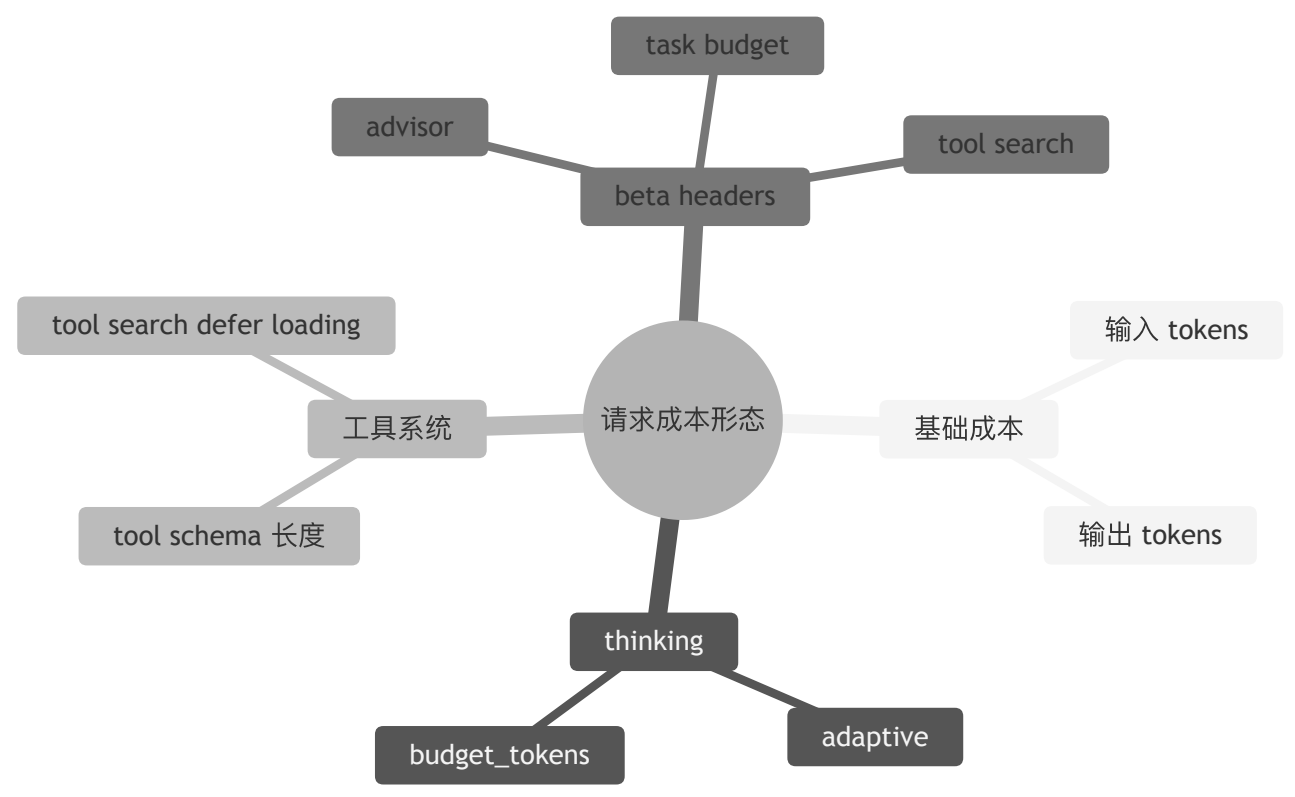
如果工具搜索开启，系统会：

- 判断哪些工具是 deferred
- 根据历史里有没有 `tool_reference` 决定是否把工具声明真正带上
- 再追加对应的 tool-search beta header

这既影响提示词长度，也影响请求元数据。

Advisor 和其他 server-side tool 也会改变请求形态

当 advisor 启用时，会追加 advisor beta header，还可能选择一个专门的 advisor model。
这意味着“同样一句用户输入”，实际请求体并不总是同一形状。



这就是“Token 经济学”比“Token 计数”更重要的原因

`Token 计数` 是问：

- 这次用了多少 token?

`Token 经济学` 是问：

- 这些 token 为什么会花在这里?
- 哪部分是结构性开销?
- 哪部分可以缓存?
- 哪部分是为了质量提升而主动付出的成本?

Claude Code 的源码显然站在第二种视角上。

源码证据

- `OpenClaudeCode/src/services/api/claude.ts:1064-1181`：betas、advisor、tool search 的请求形态调整
- `OpenClaudeCode/src/services/api/claude.ts:1591-1628`：thinking 的 adaptive / budget 选择

🔭 深水区（架构师选读）

第 13 章最值得带走的，不是“token 很贵”这句废话，而是 Claude Code 对成本的态度：

- 它在客户端提前估算
- 在 prompt 层做缓存友好切分
- 在循环层做 blocking / compact / budget 护栏
- 在 API 层显式编码 task budget 和 betas
- 在请求形态层承认 thinking、advisor、tool search 都会改变成本结构

这说明它把成本当成架构问题，而不是财务报表上的事后统计。

本章小结

一句话：Claude Code 的 Token 经济学不是“事后看看花了多少”，而是从 warning 阈值、prompt caching、task budget、thinking 配置到 beta header，整条链路都在主动管理成本和上下文窗口。

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/services/compact/autoCompact.ts:93-145`	token warning / error / blocking 计算
补全层	`OpenClaudeCode/src/query.ts:628-646`	正式请求前的 blocking limit 检查
补全层	`OpenClaudeCode/src/constants/prompts.ts:105-115`	prompt 动静态边界
补全层	`OpenClaudeCode/src/constants/prompts.ts:560-576`	静态前缀与动态尾部拼接
补全层	`OpenClaudeCode/src/services/api/claude.ts:473-500`	`task_budget` API 参数
补全层	`OpenClaudeCode/src/query.ts:282-291`	compact 后继续追踪 remaining budget
补全层	`OpenClaudeCode/src/query.ts:504-515`	compact 时预算扣减
补全层	`OpenClaudeCode/src/services/api/claude.ts:1064-1181`	advisor / tool search / beta headers
补全层	`OpenClaudeCode/src/services/api/claude.ts:1591-1628`	thinking 的预算选择

逆向提醒

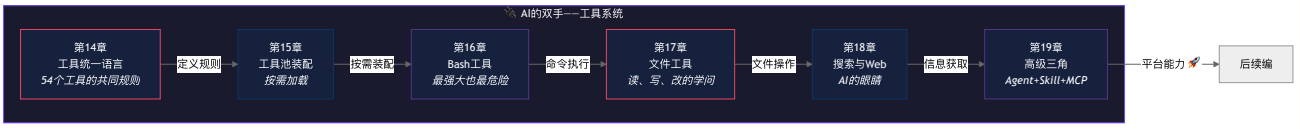
- ✅ **可信度高**：blocking、task budget、dynamic boundary、thinking 配置都能直接在源码里定位
- ⚠️ **要避免简化**：`maxTurns` 不是 token budget，compact 也不等于“成本归零”
- ❌ **不要误读**：prompt caching 节省的不是“所有输入”，而是边界前那一大段稳定前缀

19

第四编：AI的双手——工具系统

> *所有电器用同一种插头标准——冰箱、电视、手机都不需要定制插座。*
>
> 本编解析 Claude Code 的 54 个工具如何用**统一接口**组织起来：**工具类型**、**工具池装配**、**Bash/文件/搜索三大核心工具**、**高级三角（Agent+Skill+MCP）**。

本编总览



本编六章速览

章	标题	核心问题	生活类比
14	工具统一语言	功能天差地别的工具怎么用同一个 Tool 类型?	电器的统一插头标准
15	工具池装配	54 个工具全开会怎样?	出门只带需要的工具
16	Bash工具	能执行任意命令有多危险?	万能钥匙
17	文件工具	AI 说"改第3行"为什么不能直接 sed?	图书馆三个柜台
18	搜索与Web	AI 怎么在百万行代码中找 bug?	侦探的放大镜和情报网
19	高级三角	Claude Code 只是 CLI 还是可编程 AI 平台?	从独行侠到项目经理

设计思想主线

本编建立的认知基础

1. 统一的 Tool 接口让新增工具只需"填空"——**接口设计的力量**
2. 工具池按模式和上下文动态装配——**不是所有工具都随时可用**
3. BashTool 是最强大也最危险的能力——**力量越大，约束越必要**
4. 文件工具分读/写/改三个——**分工本身就是安全防线**
5. Agent+Skill+MCP 三角让 Claude Code 从 CLI 升级为**开放 AI 平台**

推荐路径

○ ○ ○

🌱 初学者 🔧 开发者 🏗️ 架构师

从第16章 Bash 工具开始——**最直观也最震撼**。然后看第14章理解统一接口的设计。

第14–15章的工具抽象和装配模式是**可复用的架构智慧**。第17章的文件工具设计值得细读。

第19章的 Agent+Skill+MCP 三角展示了**平台化思维**——从工具到平台的演进路径。

阅读建议

如果你想快速理解 Claude Code 为什么不只是“会聊天”，优先读 **第14章 → 第15章 → 第16章**。如果你更关心平台化能力，再接着读 **第19章**，会一下子看清这套系统的野心。

20

工具接口 第四编

第14章：工具的统一语言：54个工具的共同规则

生活类比

家里有冰箱、电视、台灯、手机充电器，但墙上的插座并不会为每件电器单独定制。它们能共存，是因为背后有一套统一标准。**Claude Code 的 `Tool` 接口，就是 54 个工具共用的“插座标准”。**

这一章要回答的问题

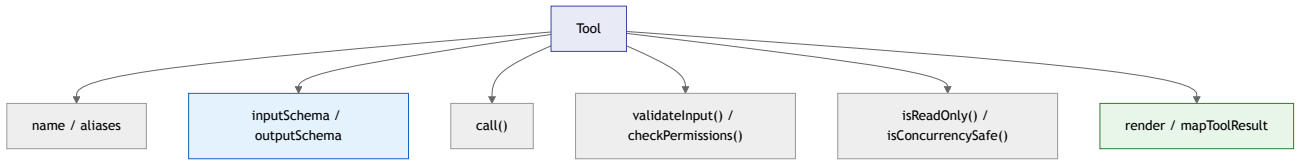
文件读取、Shell 执行、网页搜索、子智能体派遣，这些看起来完全不同的能力，为什么都能放进同一条执行管线里？

如果每种工具都有自己的参数格式、权限逻辑和展示方式，Claude Code 会很快失控。源码真正高明的地方在于：它先定义了一门统一语言，再让每个工具用这门语言“填空”。

14.1 `Tool` 不是一个函数，而是一整份能力合同

`Tool.ts` 里的 `Tool<Input, Output, P>` 泛型类型，很像一份标准合同。它要求每个工具都说清楚：

- 我叫什么
- 我接受什么输入
- 我怎么执行
- 我怎么检查权限
- 我是不是只读
- 我能不能并发
- 用户打断我时怎么办
- 我的结果该怎么映射成 tool_result



为什么这里要用泛型

因为每个工具都长得不一样：

- Bash 的输入是 `command`
- FileRead 的输入是 `file_path + offset + limit`
- WebSearch 的输入是 `query + domains`

但统一接口让系统仍然能在编译期知道：

- 这个工具的输入长什么样
- 这个工具的输出长什么样
- 它的 progress 数据长什么样

对初学者来说，这相当于：

- “所有工具都要交作业”
- 但每门课交的作业内容不同
- 老师依然可以用一套总规则管理全班

一个特别值得注意的字段：`searchHint`

`Tool` 不只定义运行行为，还定义“怎么被模型找到”。`searchHint` 的注释写得很清楚：这是给 ToolSearch 用的能力短语。

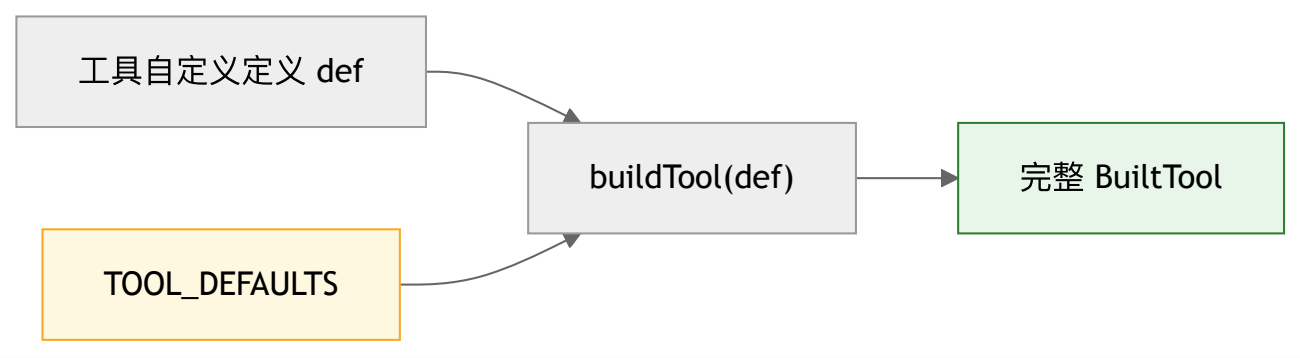
也就是说，Claude Code 的工具系统已经不是“把工具挂上去”那么简单，而是连“模型如何发现工具”都纳入了统一类型。

源码证据

`OpenClaudeCode/src/Tool.ts:362-520` 定义了 `Tool` 的核心合同字段。

14.2 `buildTool()` 的意义：不是少写代码，而是统一默认行为

如果只靠接口约束，每个工具还是得自己实现一堆重复样板。
Claude Code 的做法更进一步：用 `buildTool()` 和 `TOOL_DEFAULTS` 把默认行为集中起来。



默认值里最关键的几条：

- `isEnabled: () => true`
- `isConcurrencySafe: () => false`
- `isReadOnly: () => false`
- `isDestructive: () => false`
- `checkPermissions: allow`
- `userFacingName: () => def.name`

这套默认值背后的设计哲学很实用

它不是“给工具开绿灯”，而是：

- 默认假设**不并发安全**
- 默认假设**不是只读**
- 默认假设**不是破坏性操作**
- 默认把权限交给总权限系统处理

也就是说，Claude Code 对工具采取的是一种“保守默认”。

`buildTool()` 解决的是一致性问题

新增一个工具时，开发者只要提供差异部分：

- 名字
- schema
- call
- 特殊权限逻辑

剩下那些通用行为自动继承。
这让 54 个工具的“长相”非常统一。

`toolMatchesName()` 和 `findToolByName()` 是整个工具生态的路标

别小看这两个小函数。它们决定了：

- 工具怎么被查找到
- alias 如何兼容重命名
- StreamingToolExecutor、ToolSearch、MCP 适配层都怎么定位具体工具

很多系统最后会因为“名字匹配规则混乱”而产生幽灵 bug。Claude Code 这里做得很克制。

源码证据

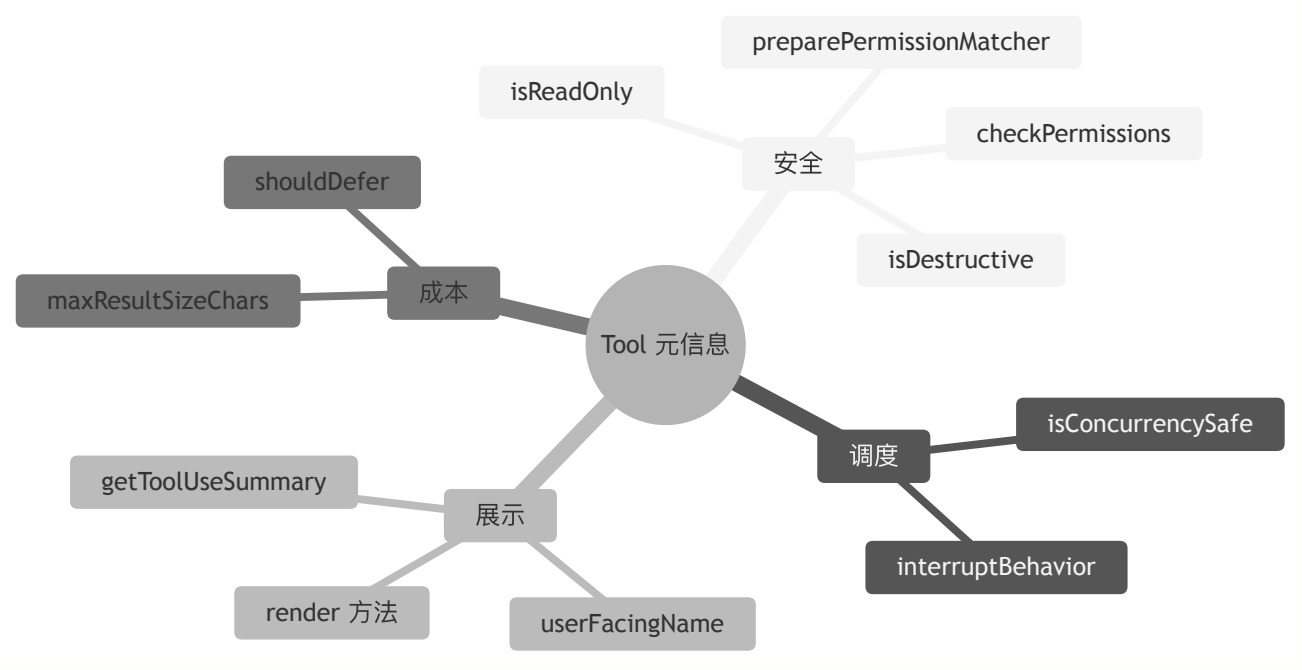
- `OpenClaudeCode/src/Tool.ts:348-359`：工具查找与别名匹配
- `OpenClaudeCode/src/Tool.ts:757-792`：`TOOL_DEFAULTS` 与 `buildTool()`

14.3 统一接口最厉害的地方，是把“风险特征”也标准化了

很多人第一次看工具接口，只关注 `call()`。其实更值钱的是这些“风险描述字段”：

- `isReadOnly`
- `isConcurrencySafe`
- `interruptBehavior`
- `preparePermissionMatcher`

- `maxResultSizeChars`
- `isSearchOrReadCommand`



为什么这很重要

因为 Claude Code 不是“调用一个函数”就结束了。
它还要知道：

- 这个工具能不能和别的工具并发
- 用户输入新消息时要不要取消它
- 这个结果要不要持久化到磁盘
- 这个调用是不是应该在 UI 里折叠显示成“读/搜”操作

换句话说，工具不仅是功能单元，还是**调度单元、权限单元、显示单元、成本单元**。

`maxResultSizeChars` 是一个特别容易被忽略的系统设计点

这个字段说明工具结果不是想返回多少就返回多少。
如果结果太大，系统会考虑把内容持久化到磁盘，再给模型一个预览或路径。

这和第 13 章的 token 经济学直接连起来了：

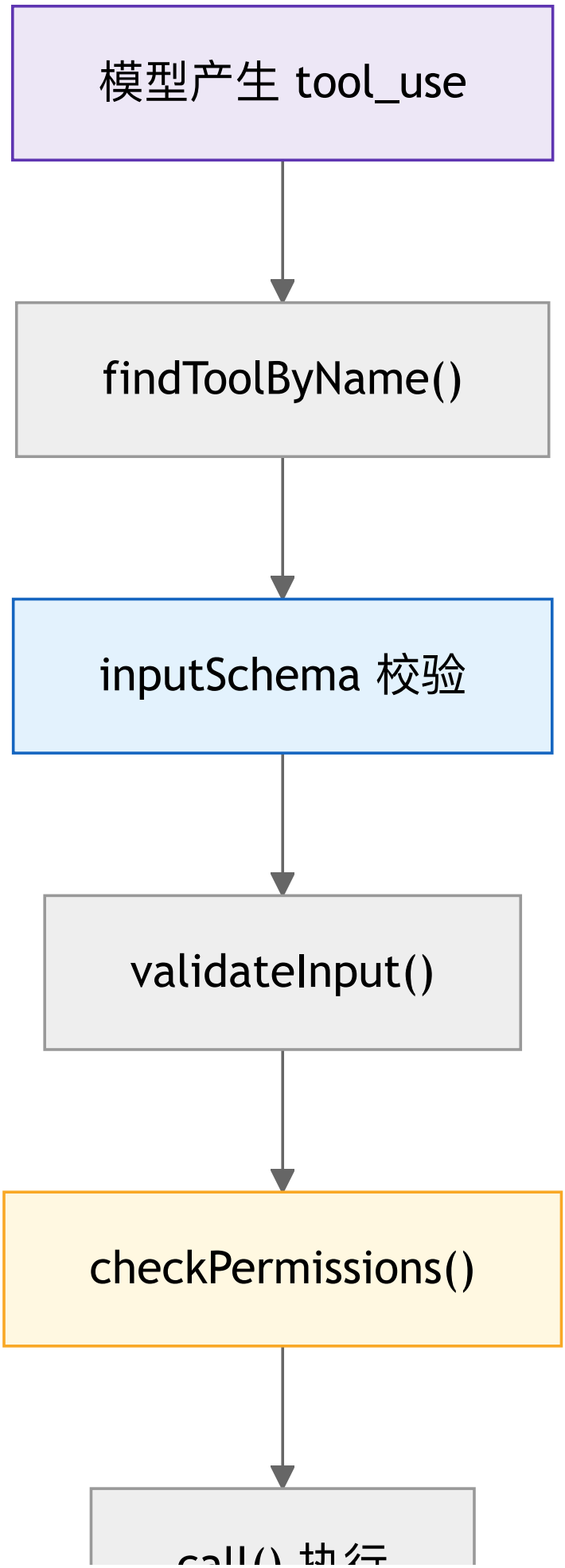
- 工具系统不是“只负责产出”
- 它还要为上下文窗口负责

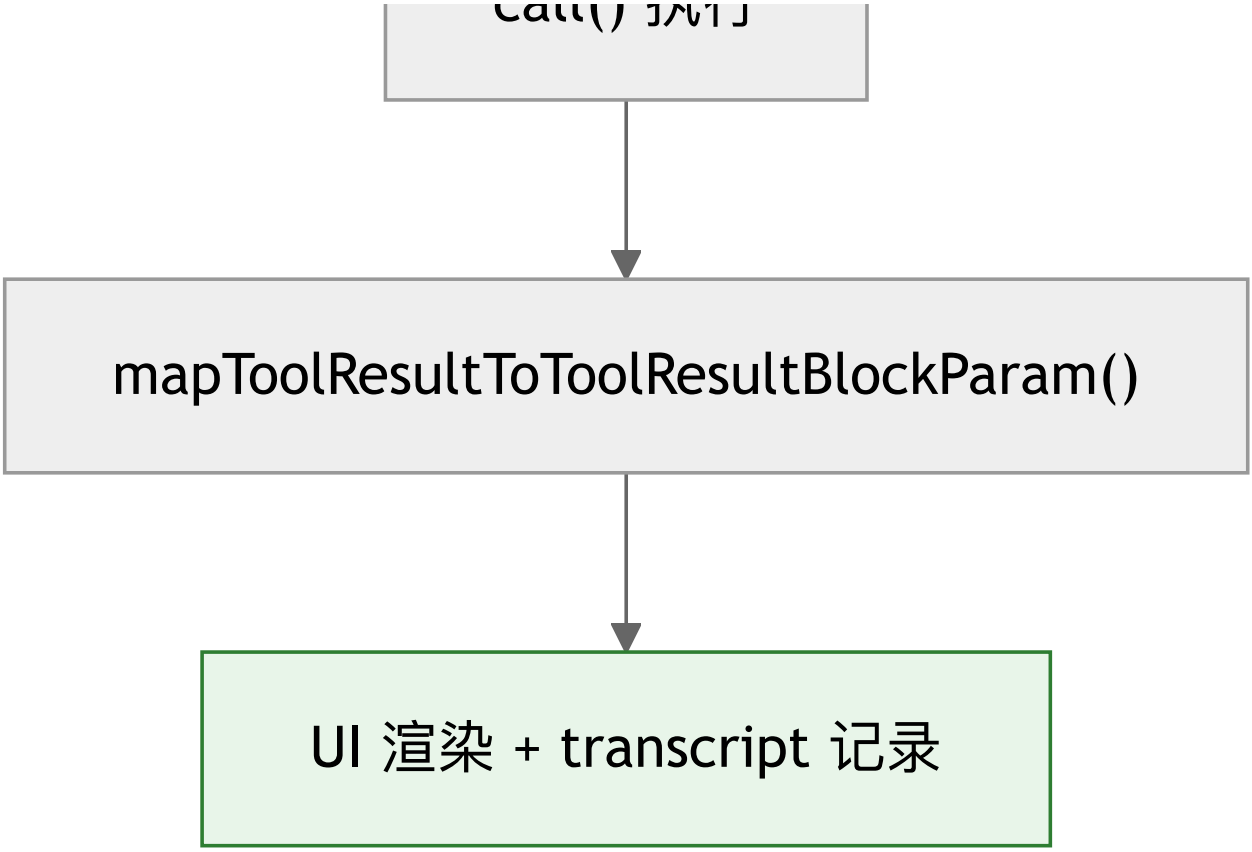
设计思想

Claude Code 把工具当成“可被编排的资源”，而不是“随手调的函数”。

14.4 一次工具调用，到底会经历哪些步骤

统一接口最直观的价值，是所有工具都能走同一条主流水线：


模型产生 tool_use
findToolByName()
inputSchema 校验
validateInput()
checkPermissions()
call() 执行



这条流水线带来的三个好处

1. **新增工具更容易**
   只要遵守合同，自动接上既有系统。
2. **调试路径统一**
   出问题时知道该看 schema、权限、执行还是结果映射。
3. **系统功能可以横向扩展**
   比如流式执行、权限规则、ToolSearch、MCP 注入，都能复用同一条总线。

所以第 14 章真正讲的不是“54 个工具”

而是 Claude Code 先发明了一种“工具语言”，然后所有工具都用这门语言说话。
只要语言没坏，工具数量再多，系统也不会立刻崩。

深水区（架构师选读）

很多团队做工具系统时，第一反应是“先做几把工具出来再说”。Claude Code 的做法恰恰相反：**先把抽象做稳，再把工具往里放**。

这让工具系统从一开始就具备了四种扩展能力：

- 类型扩展：泛型和 schema 让新工具长得一致
- 执行扩展：统一执行流水线
- 权限扩展：权限系统不必为每个工具单独重写
- 平台扩展：MCP、Skill、Agent 都能接入同一套合同

第 14 章是整本书很关键的一章，因为后面所有工具章节，本质上都在解释这份合同的不同填写方式。

本章小结

一句话：Claude Code 不是先堆出 54 个工具，再想办法管理它们；它是先定义了一套 `Tool` 合同和 `buildTool()` 默认行为，再让所有工具用同一门语言接入系统。**

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/Tool.ts:348-359`	工具查找与 alias 匹配
补全层	`OpenClaudeCode/src/Tool.ts:362-520`	`Tool<Input, Output, Progress>` 主合同
补全层	`OpenClaudeCode/src/Tool.ts:757-792`	`TOOL_DEFAULTS` 与 `buildTool()`

逆向提醒

- ✅ **可信度高**：接口字段、默认行为、buildTool 组装过程都在源码里有直接定义
- ⚠️ **要看整体**：`Tool` 不只是 `call()`，更重要的是围绕它的并发、权限、展示、持久化元信息
- ❌ **不要误解**：默认 `checkPermissions=allow` 不等于所有工具天然安全，很多工具会显式覆写这条默认值

21

工具池装配 第四编

第15章：工具池装配：不是所有工具都随时可用

生活类比

水管工上门修水龙头，不会背着整家五金店来。他会先看任务，再决定带扳手、胶带还是测压表。**Claude Code 也一样：不是所有工具都应该一直摊在模型面前。**

这一章要回答的问题

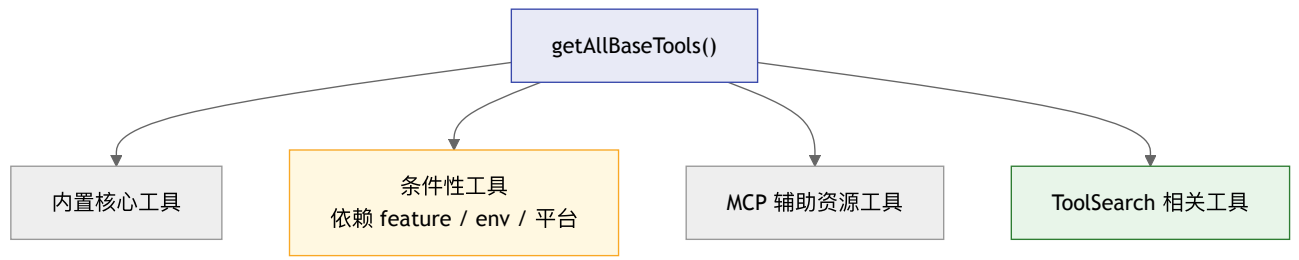
如果 54 个工具全开，会发生什么？Claude Code 又是怎么决定“这次会话里，模型手边到底有哪些工具”的？

这一章最重要的认知转变是：`tools.ts` 不是一个“工具清单文件”，而是一个**工具装配车间**。

15.1 第一层：先定义“当前环境下可能存在的全部基础工具”

`getAllBaseTools()` 是工具系统的第一道大门。
它定义的不是“这轮一定能用哪些工具”，而是：

> 在当前环境、当前构建、当前 feature gate 条件下，**理论上可用的基础工具全集**。



在这份列表里，你能看到很典型的分层：

- 永远重要的：`AgentTool`、`BashTool`、`FileReadTool`、`FileEditTool`、`FileWriteTool`
- 条件出现的：`GlobTool`、`GrepTool`、`LSPTool`、`REPLTool`
- 依赖环境变量或 feature 的：worktree、workflow、monitor、cron、powershell
- 与 MCP 资源相关的辅助工具：`ListMcpResourcesTool`、`ReadMcpResourceTool`

为什么源码特别强调“这必须和缓存策略保持同步”

`tools.ts` 的注释明确写了：

> 这个列表必须和 system caching 配置保持同步。

原因很简单：工具清单本身会进入 prompt。
如果基础工具的排序和内容一直乱变，prompt cache 就很难稳定命中。

这再次说明，工具装配不是单纯的功能问题，也是**上下文成本问题**。

源码证据

`OpenClaudeCode/src/tools.ts:193-250` 定义了当前环境下的基础工具全集。

15.2 第二层：按模式和权限先把工具池裁一遍

真正给模型用之前，还要再过几道筛子。
`getTools(permissionContext)` 负责做这件事。

它至少做了三层过滤：

1. **模式过滤**
   比如 `CLAUDE_CODE_SIMPLE` 只保留少量原始工具。
2. **deny rule 过滤**
   如果权限上下文里有 blanket deny，工具在模型看见之前就被拿掉。

3. **REPL 特殊处理**
   如果启用了 REPL，会隐藏某些 primitive tools，避免直接暴露。

`SIMPLE` 模式很有代表性

源码里写得很直白：简单模式下，只保留：

- `BashTool`
- `FileReadTool`
- `FileEditTool`

这说明“工具池装配”不是装饰行为，而是会直接改变 Claude Code 的能力边界。

为什么 deny rule 要在“模型看到之前”就生效

因为和其事后拦截，不如事前不暴露。
如果一个工具本来就肯定不能用，还把 schema 塞进 prompt，只会带来两种坏处：

- 浪费 token
- 让模型做无意义尝试

这和前端里“禁用按钮”比“点了再弹错”更好，是同一个道理。

REPL 模式为什么还要隐藏 primitive tools

注释点得很清楚：当 REPL 开启时，某些原始工具依然能在 VM 内部用，但不应该直接暴露给模型。
这是典型的“内部实现能力”和“外部公开接口”分离。

源码证据

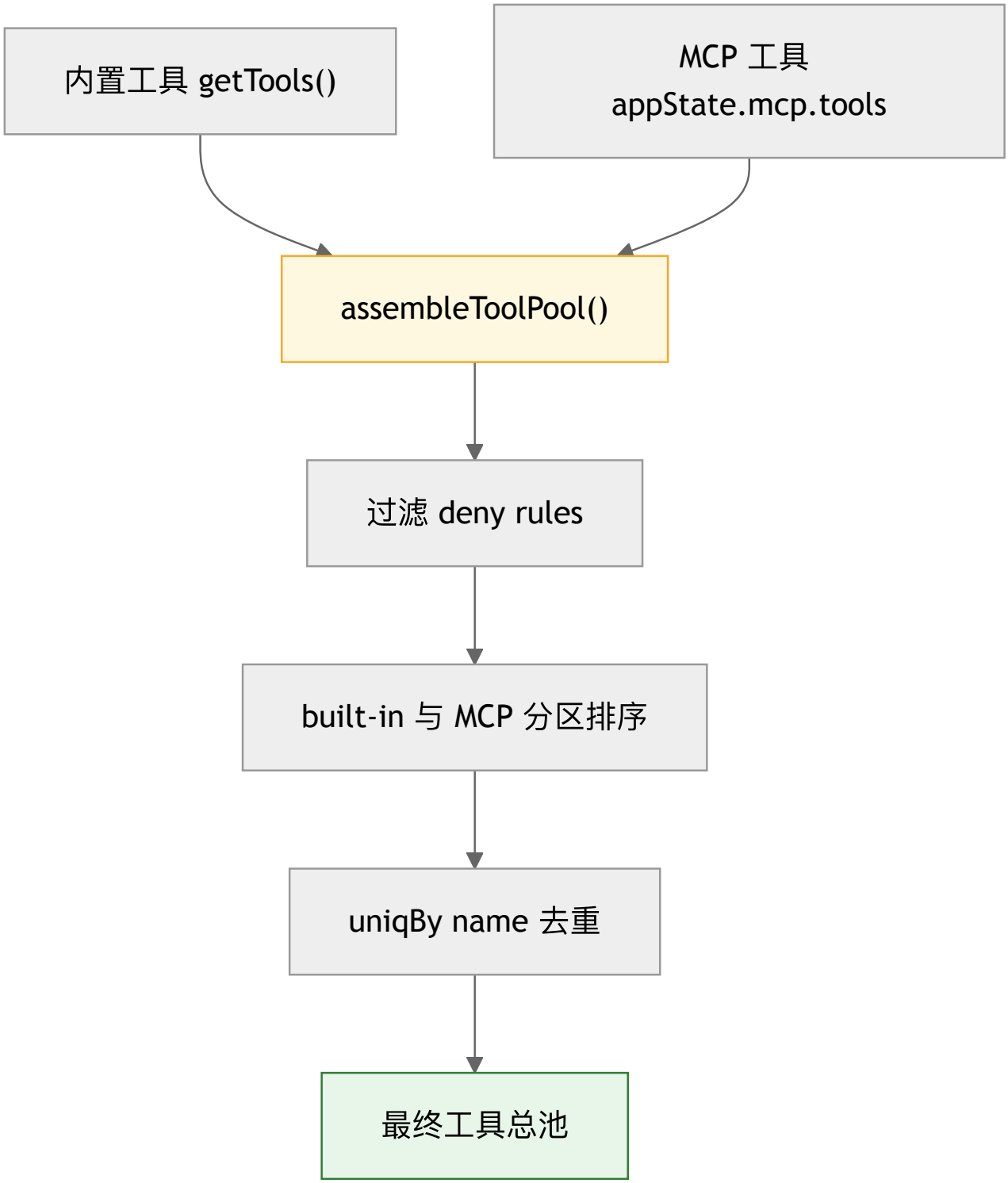
`OpenClaudeCode/src/tools.ts:271-327` 展示了模式、deny rule、REPL 和 `isEnabled()` 的多层过滤。

15.3 第三层：把内置工具和 MCP 工具拼成一个稳定的总池

有了“当前允许的内置工具”以后，还不够。Claude Code 还要把运行时连进来的 MCP 工具一起拼上去。

真正负责这件事的是 `assembleToolPool()`：

1. 先拿到 built-in tools
2. 再过滤 MCP tools 的 deny rules
3. 最后按名称排序并去重



这里最值钱的细节：为什么要“分区排序”

源码注释写得非常到位：

- built-in tools 作为连续前缀
- MCP tools 排在后面
- 这样做是为了 **prompt-cache stability**

如果用“全量平铺排序”，只要某个 MCP 工具名字刚好插进内置工具中间，后面的序列化顺序就全变了，缓存键也跟着大面积失效。

这说明 `assembleToolPool()` 看起来只是数组操作，实际上是在做：

- 平台扩展
- 名称冲突处理
- prompt 稳定性维护

为什么 built-in 工具在同名冲突时优先

`uniqBy` 保留插入顺序，而 built-in 工具先排在前面。
这意味着如果出现同名冲突：

- 内置工具赢
- MCP 工具被去掉

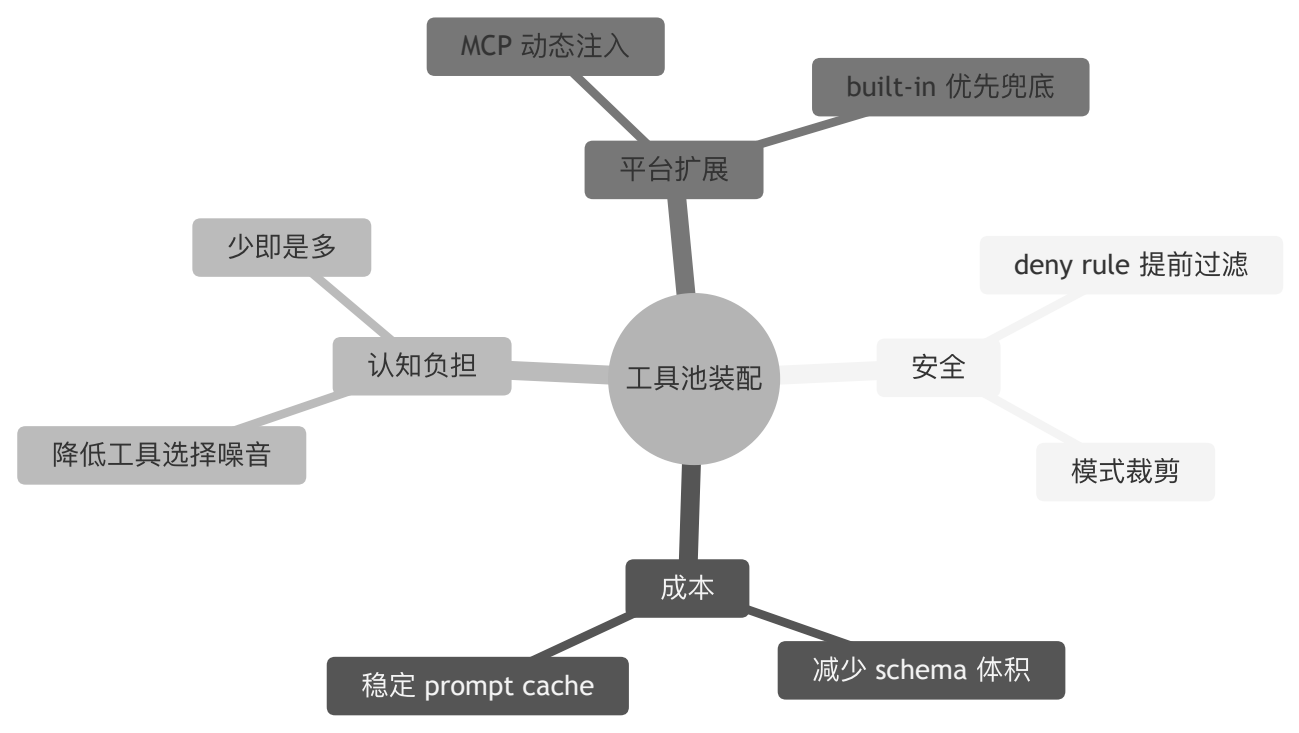
这是一种很保守、很合理的默认策略，因为内置工具的行为更可控、更被系统理解。

源码证据

`OpenClaudeCode/src/tools.ts:329-389` 展示了内置工具与 MCP 工具的合并、排序和去重策略。

15.4 所以“工具池装配”到底在解决什么问题

到这里，我们可以把第 15 章真正讲的东西总结成四个关键词：



为什么“少即是多”在工具系统里尤其成立

工具不是越多越好。
对模型来说，每多一个工具，就多一份：

- schema 负担
- 选择负担
- 误用风险

所以优秀的工具系统，不是“把所有能力都摊平”，而是“按场景给出最合适的一把工具箱”。

这也是为什么 Claude Code 能从 CLI 长成平台

如果工具池不能动态装配：

- MCP 很难接进来
- 子智能体拿不到自己的独立工具集
- 简化模式和 REPL 模式也会互相干扰

所以第 15 章其实是在解释 Claude Code 的一个底层平台能力：
它可以根据上下文，重组自己的“手”。

深水区（架构师选读）

工具池装配的本质，是把三类看似无关的问题压到同一个函数里解决：

- 哪些工具应该被看见
- 这些工具怎么排顺序
- 这个顺序会不会打坏缓存

这很像数据库里的查询规划器，不只是“把东西列出来”，而是把“安全、性能、扩展、稳定性”同时纳入考虑。Claude Code 在 `tools.ts` 里表现出的成熟度，远远超过一般 demo 级 AI 工具项目。

本章小结

一句话：`tools.ts` 不是一份静态清单，而是一条装配流水线：先确定当前环境下可能存在的基础工具，再按模式和权限裁剪，最后和 MCP 工具合并成一份对缓存友好、对模型友好的最终工具池。**

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/tools.ts:193-250`	基础工具全集
补全层	`OpenClaudeCode/src/tools.ts:271-327`	模式过滤、deny rules、REPL 特殊处理
补全层	`OpenClaudeCode/src/tools.ts:329-389`	`assembleToolPool()` 合并 built-in 与 MCP 工具

逆向提醒

- ✅ **可信度高**：基础工具列表、simple 模式、deny rule 过滤和 MCP 合并策略都能直接定位
- ⚠️ **要注意动态性**：最终工具池不是常量，会随模式、权限、MCP 连接状态变化
- ❌ **不要误读**：`getAllBaseTools()` 不等于“本轮模型一定可见的工具”；真正给模型看的还要经过后续多轮过滤

22

Bash工具 第四编

第16章：Bash工具：最强大也最危险的能力

生活类比

给 AI 一把万能钥匙，效率会暴涨；但万能钥匙的问题也很明显：它开的不只是该开的门。**BashTool 就是 Claude Code 手里那把最强的钥匙。**

这一章要回答的问题

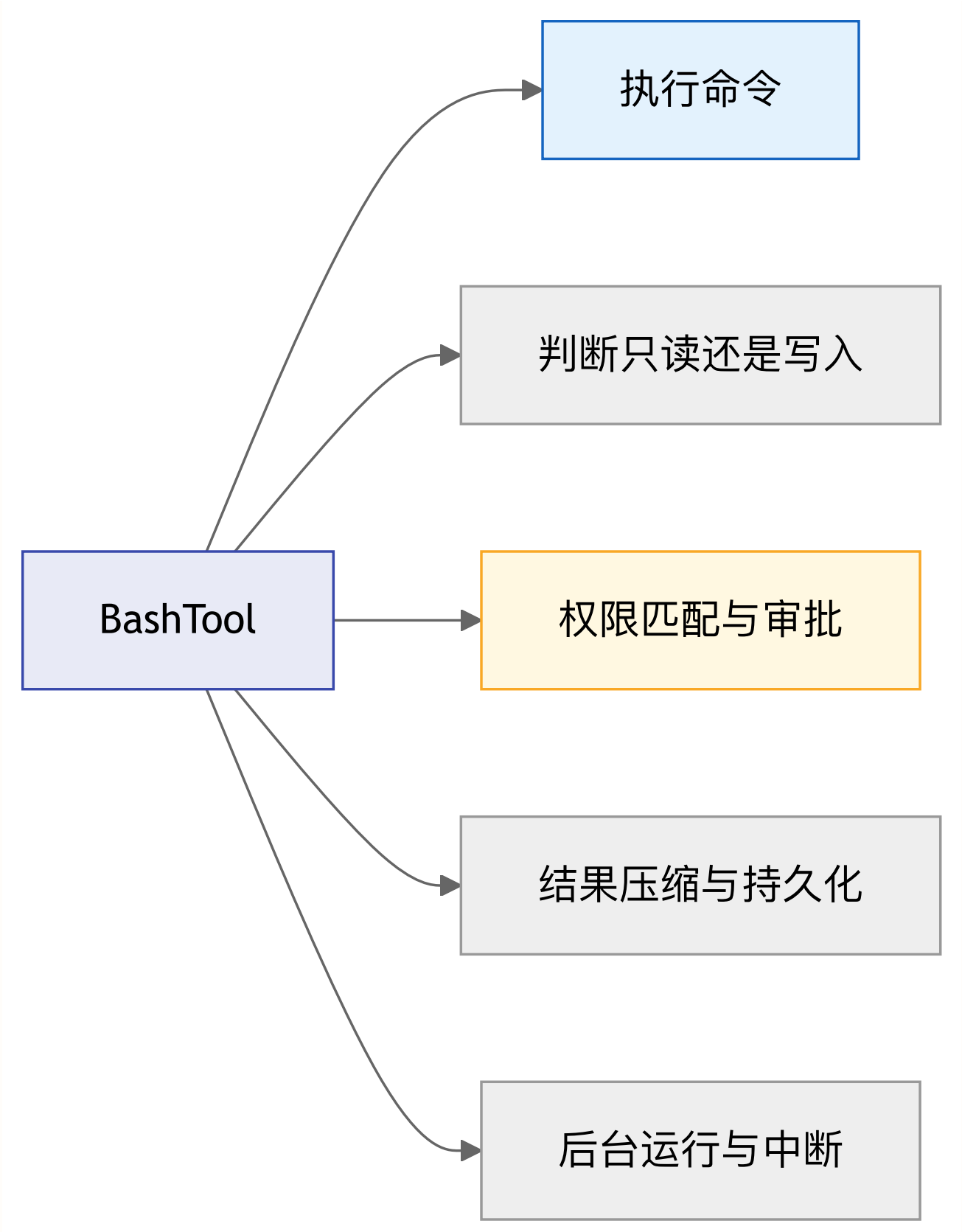
Claude Code 为什么还要保留一个能跑 Shell 的 BashTool？既然它这么危险，为什么不干脆全用专用工具替代？

答案很现实：专用工具再多，也覆盖不了真实开发环境的全部需求。BashTool 是“最后的通用能力”，但也因此必须被包上最多的约束。

16.1 BashTool 表面上是一个普通 Tool，实际上是“能力兜底”

在 `BashTool.tsx` 里，它照样通过 `buildTool()` 注册：

- `name = Bash`
- `searchHint = execute shell commands`
- `strict = true`
- `maxResultSizeChars = 30_000`



这说明一个很重要的事实：

> Bash 不是系统外的“特权旁路”，而是工具体系内的正式成员。

也正因为它是正式成员，Claude Code 才能把：

- 权限系统
- 并发系统
- transcript 系统

- 结果持久化系统

全部套在它身上。

为什么专用工具越多，Bash 仍然不会消失

因为真实开发任务总会出现这些情况：

- 需要调用项目自己的脚本
- 需要运行测试、构建、安装依赖
- 需要使用仓库里独有的命令组合

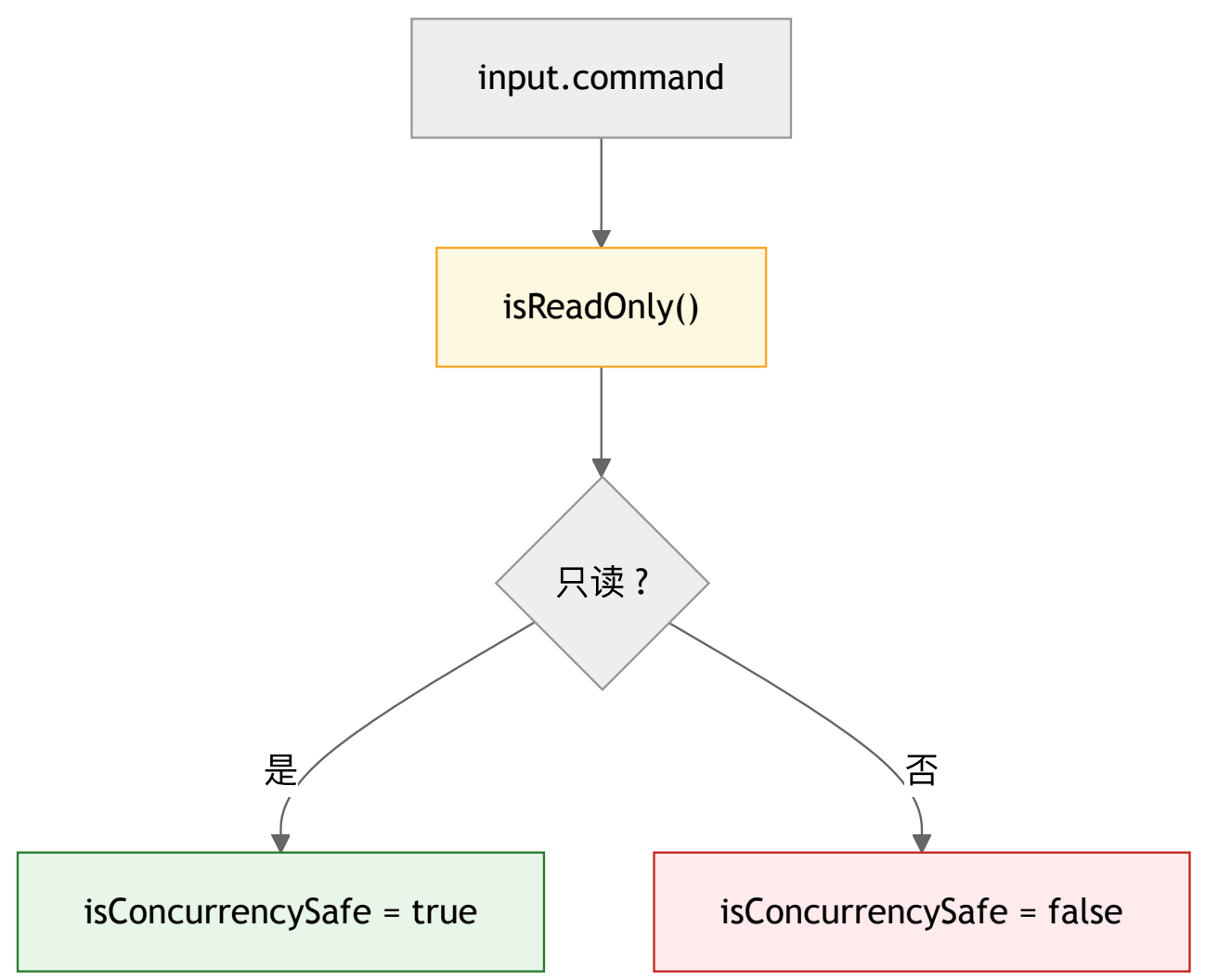
这类事情很难靠若干固定专用工具完全覆盖。
所以 BashTool 的价值不是“优雅”，而是“兜底”。

源码证据

`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:420-443` 展示了 BashTool 作为正式 Tool 的注册形态。

16.2 Claude Code 不会把所有 Bash 命令都当成一样危险

BashTool 最有意思的一点，是它并不是“命令一律高危”。
源码里显式区分了只读和非只读，并把并发安全绑定到这个判断上。



这背后的逻辑非常符合直觉

- 只读命令
  比如 `ls`、`cat`、`git status`，通常可以更大胆地并发。
- 会改状态的命令
  比如安装依赖、改文件、跑有副作用的脚本，就应该更保守。

源码里 `isConcurrencySafe(input)` 直接写成：

```
取决于 isReadOnly(input)
```

这说明 Claude Code 并不是通过“工具名字”判断安全，而是通过**输入语义**判断。

`userFacingName()` 还有一个很妙的细节

如果命令里是一个就地修改文件的 `sed`，BashTool 会把这个调用在 UI 里渲染得更像一次文件编辑，而不是冷冰冰的 Shell 命令。

这说明 BashTool 并不想把自己永远呈现成“黑箱命令”，而是在尽量把用户体验对齐到更可理解的动作语义。

源码证据

`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:434-503` 展示了只读判断、并发安全判断和 user-facing name 逻辑。

16.3 真正危险的不是“执行命令”四个字，而是命令结构本身

Claude Code 没有靠简单字符串匹配来做 Bash 安全。
`preparePermissionMatcher()` 里会先调用 `parseForSecurity(command)`，把命令拆成结构，再做匹配。

为什么要做到这一步

因为类似这样的命令：

- `ls && git push`
- `FOO=bar git push`

不能靠简单的“字符串里有没有 git”来判断。
源码注释说得很清楚：如果不拆子命令，像 `Bash(git *)` 这种权限规则就可能被复合命令绕过去。

这就是“从模式匹配到语义匹配”的升级

这类设计非常像安全工程里的成熟路线：

- 低级做法：搜关键词
- 高级做法：分析结构

Claude Code 在 BashTool 上明显已经走到了第二层。

`validateInput()` 还会拦截某些“表面合法、实际很糟”的命令

比如源码里会检测一些阻塞式 `sleep` 模式，并建议：

- 真要长期跑，请后台运行
- 持续监控流式事件时，请用 Monitor tool

这说明 BashTool 不只是怕“危险命令”，它也在防“糟糕的使用方式”。

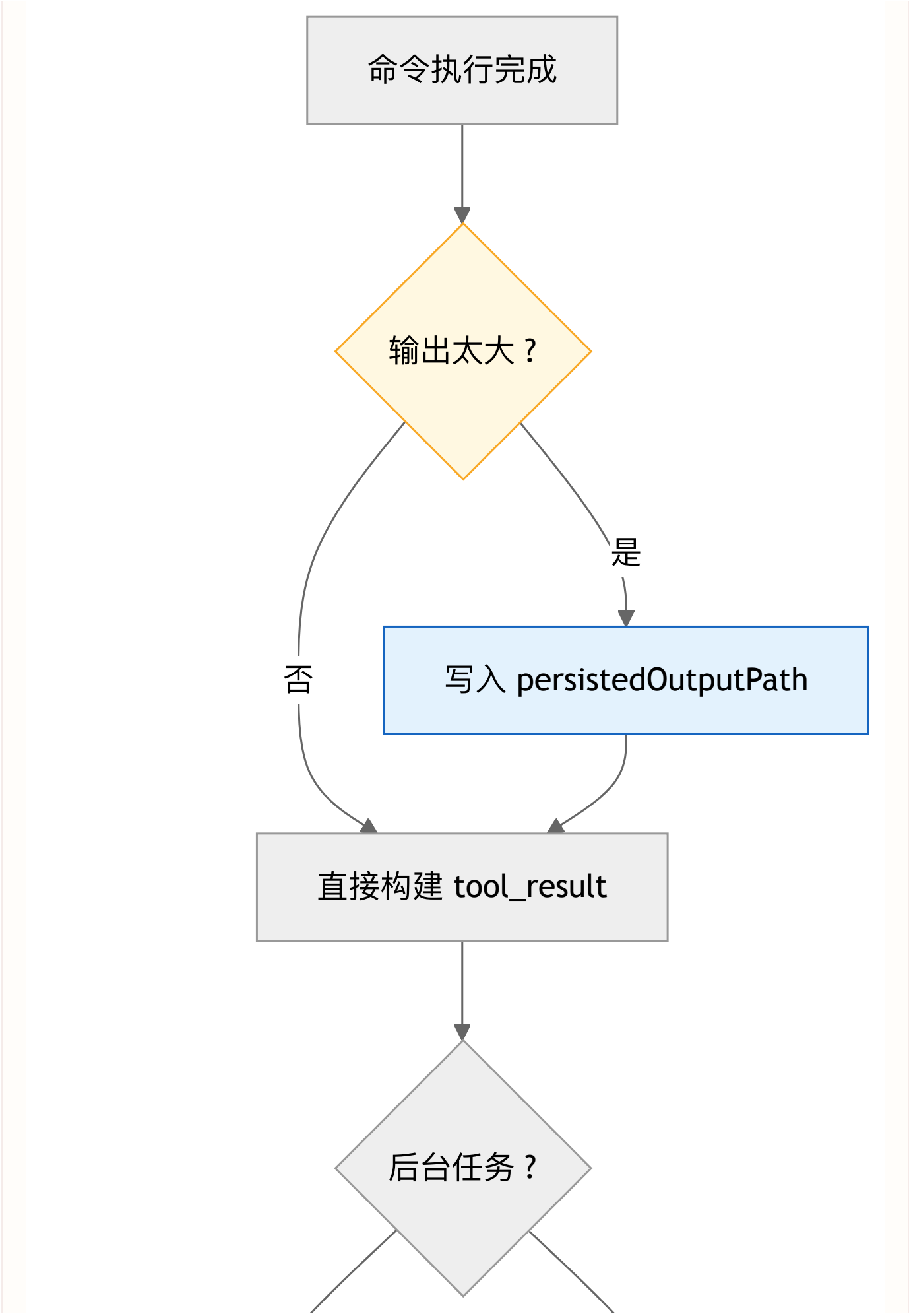
源码证据

`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:445-467` 展示了结构化 permission matcher；
`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:524-540` 展示了输入验证与权限委托。

16.4 BashTool 的结果不是“随便打一坨 stdout”就完事

BashTool 还有三件特别工程化的事情：

1. **大输出会被持久化到磁盘**
2. **长任务可以转后台**
3. **中断信息会被显式补进 tool_result**


命令执行完成
输出太大？
是
否
写入 persistedOutputPath
直接构建 tool_result
后台任务？

为什么大输出要落盘

因为像测试日志、构建产物、长 diff 这种内容：

- UI 未必适合全展示
- 模型也未必适合全吞下去

所以 Claude Code 会生成预览，再把完整结果放进持久化路径。这和第 13 章讲的 token 预算是一体两面。

为什么后台信息也要写进结果

因为“已经转后台”本身就是用户需要知道的系统事实。
源码里会根据不同情况拼出不同的 backgroundInfo：

- 自动后台化
- 用户手动后台化
- 普通后台任务

这意味着 BashTool 的返回值不只是“命令输出”，还包含**任务状态语义**。

中断不会变成沉默失败

如果命令被中止，tool result 里会明确补上：

```
Command was aborted before completion
```

这对排查非常重要。最糟糕的系统不是报错，而是“突然没声了”。

🔭 深水区（架构师选读）

BashTool 是 Claude Code 工具系统的压力测试器。只要 BashTool 能被放进统一接口并且还能保持：

- 权限可控
- 并发可控
- 输出可控
- 中断可控
- UI 语义可控

那说明整套 Tool 架构是真的有韧性。

从源码看，Claude Code 对 Bash 的态度不是“能不用就别用”，而是“必须保留，但要把它关在最多护栏里”。这是非常成熟的工程选择。

本章小结

一句话：BashTool 是 Claude Code 最强大的通用兜底能力，但它并不是系统外的特权后门，而是被纳入统一工具合同、权限分析、输出持久化和后台任务体系中的“高风险正式成员”。**

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:420-443`	BashTool 的基础注册与行为定位
补全层	`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:434-503`	只读判断、并发安全、UI 命名
补全层	`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:445-467`	结构化权限匹配
补全层	`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:524-540`	输入验证与权限委托
补全层	`OpenClaudeCode/src/tools/BashTool/BashTool.tsx:555-620`	结果映射、持久化输出与后台信息

逆向提醒

- ✅ **可信度高**：只读判断、并发策略、permission matcher、后台输出信息都在源码里直接可见

- ⚠️ **要分清楚**：BashTool 危险，不等于每条 Bash 命令都同样危险；源码明确区分了只读与写入语义
- ❌ **不要误解**：专用工具越多，不代表 BashTool 会消失；它承担的是“能力兜底”，不是“首选路径”

23

文件工具 第四编

第17章：文件工具：读、写、改的学问

生活类比

图书馆里至少有三个不同的柜台：借书、修书、归档新书。虽然都和“书”有关，但流程完全不同。**Claude Code 的文件工具也是这样：Read、Edit、Write 明明都碰文件，却故意拆成三把不同的工具。**

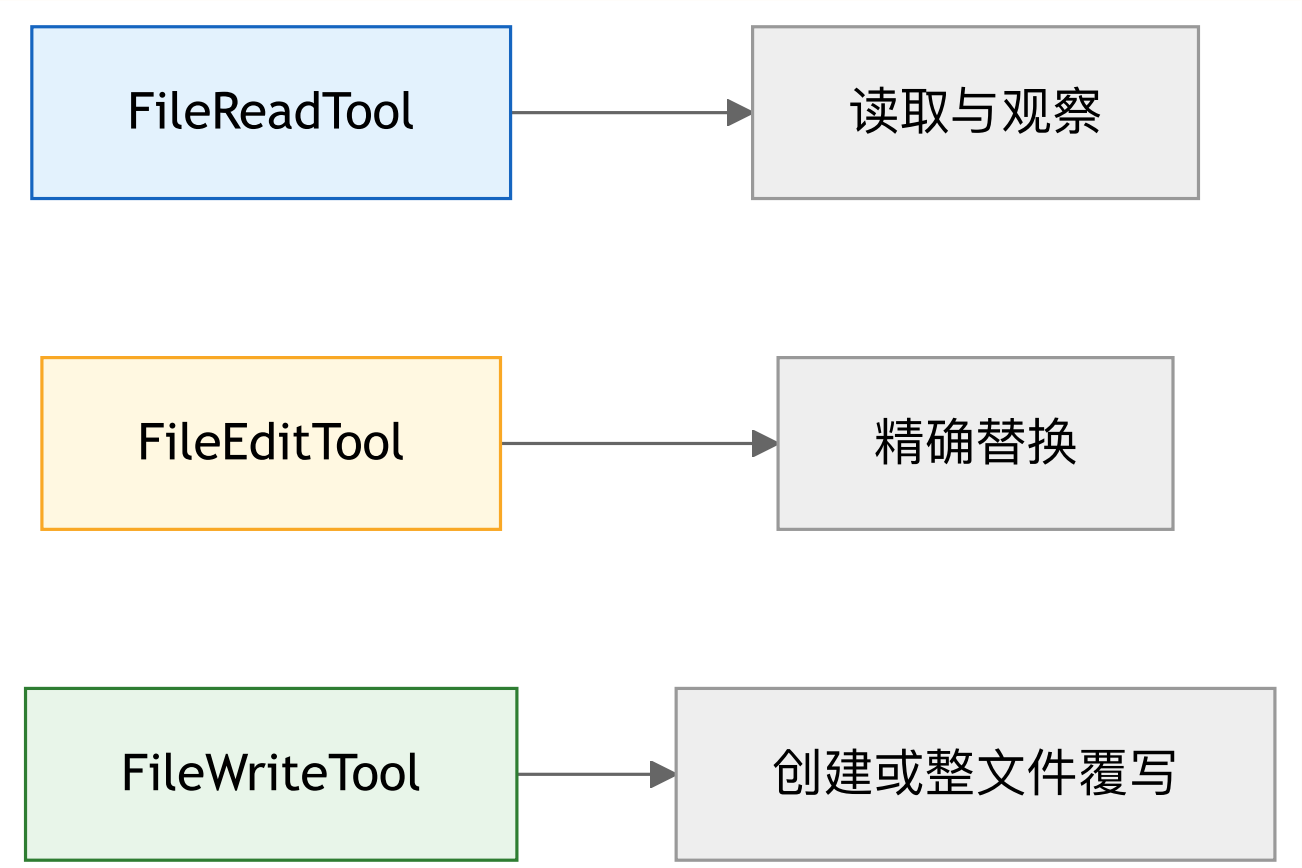
这一章要回答的问题

AI 说“把这里改一下”，为什么 Claude Code 不让它直接靠 sed 或一把万能文件工具搞定?

因为文件操作最怕的不是“不会改”，而是“改错地方”“覆盖掉别人刚改的内容”“读到半截就去写”。文件工具的分工，本身就是安全设计。

17.1 三把工具，三种责任，不要混成一把万能刀

第四编里最能体现“分工就是防线”的，就是 Read / Edit / Write 这三者。



工具	主要职责	风险类型	为什么不能合并
Read	看清当前事实	低	不该背负写入风险
Edit	在已知文件上做精确替换	中	需要唯一匹配和上下文约束
Write	创建或整体覆写文件	高	需要更强的 freshness 和原子性保护

Claude Code 的思想很明确

不是“功能越全越方便”，而是：

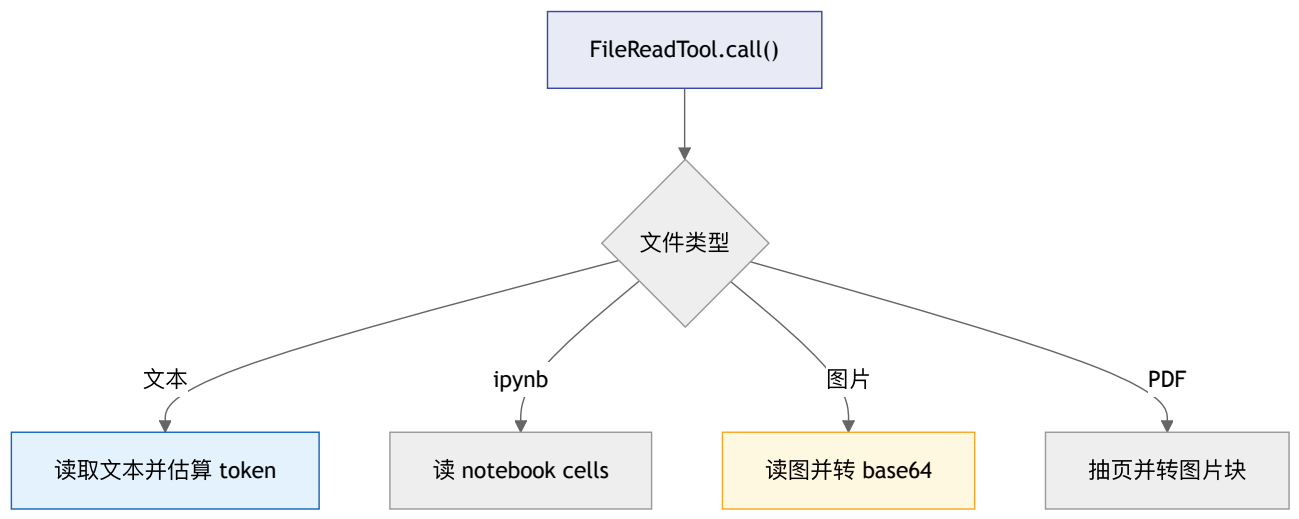
- 让每把工具只负责一种动作
- 让每种动作匹配一套最合适的约束

这比一把 `FileTool.doAnything()` 更啰嗦，但也更可靠。

17.2 Read 工具不是“读文本”，而是“统一读各种文件对象”

`FileReadTool` 很容易被误解成“读个文本文件”。
其实从源码看，它处理的东西远比这多：

- 普通文本
- notebook
- image
- PDF 指定页范围



它为什么默认是并发安全、只读安全

源码里这两条写得很直接：

- `isConcurrencySafe() => true`
- `isReadOnly() => true`

这说明对系统调度器来说，Read 是最容易并发跑的一类工具。

它还负责控制“读太多”

`FileReadTool` 会做：

- `pages` 参数校验
- token 估算
- 必要时调用 API 计 token
- 超过阈值就报 `MaxFileReadTokenExceededError`

这意味着 Read 并不是“你想读多少都行”，而是主动帮整个系统守住上下文预算。

为何 `maxResultSizeChars = Infinity` 反而合理

源码注释解释得很漂亮：

> Read 工具结果本来就由 maxTokens 等机制控制，如果再把它持久化到文件，再用 Read 读回来，会形成循环。

这是一种典型的系统级约束思维：
不是“这个字段所有工具都统一处理”，而是“这个工具的语义决定它不适合走那条路径”。

源码证据

- `OpenClaudeCode/src/tools/FileReadTool/FileReadTool.ts:337-418`：Read 工具的主合同与权限判断
- `OpenClaudeCode/src/tools/FileReadTool/FileReadTool.ts:760-771`：token 校验
- `OpenClaudeCode/src/tools/FileReadTool/FileReadTool.ts:821-920`：notebook、image、PDF 等分支读取

17.3 Edit 工具最重要的不是“能改”，而是“必须精确改”

`FileEditTool` 的整个设计，都在告诉模型一件事：

> 你不能模模糊糊地改文件，你必须清楚指出“把哪一段替换成哪一段”。

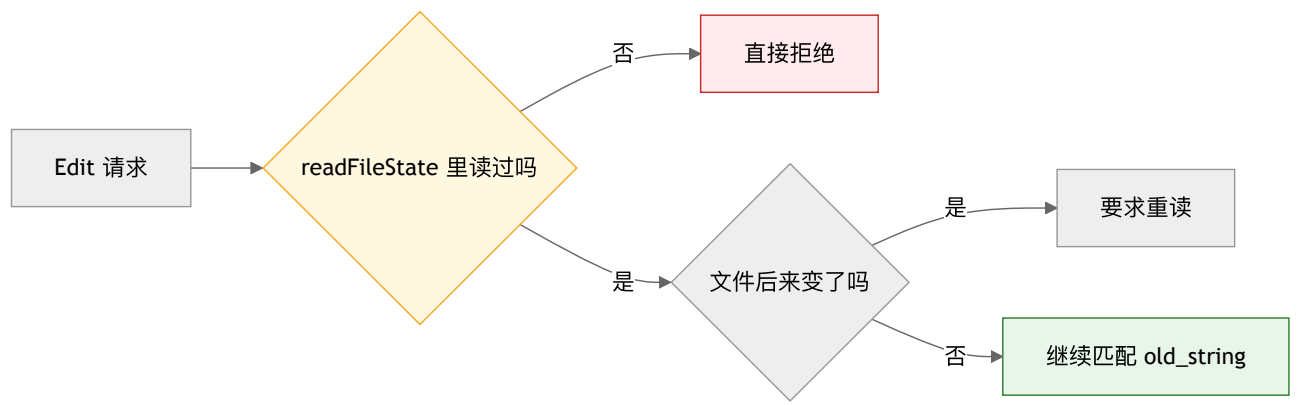
它会强制要求先读后改

无论在提示词说明里，还是在 `validateInput()` 里，Claude Code 都非常坚持：

- 先读
- 再改

如果 `readFileState` 里找不到这个文件，或者是 partial view，Edit 直接拒绝：

> File has not been read yet. Read it first before writing to it.



“唯一匹配”是核心中的核心

源码里真正决定 Edit 安全性的，是这段逻辑：

- 先找 `actualOldString`
- 如果找不到，直接报错
- 统计匹配次数
- 如果匹配次数大于 1 且 `replace_all = false`
- 直接拒绝，并提示“请提供更多上下文，或者显式 replace_all”

这就是为什么本书一直强调：

- Edit 不是“按第 3 行改一下”
- 而是“对一段唯一可识别的文本做替换”

这比 sed 慢一点，但稳很多

如果只靠行号：

- 文件前面多插一行，定位就偏了

如果只靠很短的字符串：

- 可能一模一样出现很多次

所以 Claude Code 明确要求：

- 要么给足够长的上下文，保证唯一
- 要么诚实地说“我要全量 replace_all”

它还会防“你刚读完，别人就改了”

`FileEditTool` 不只检查有没有读过，还检查：

- 上次读的时间戳
- 文件当前写入时间
- 如果 Windows 时间戳可疑，还会用内容比对做兜底

如果发现内容已经变了，就抛：

`FILE_UNEXPECTEDLY_MODIFIED_ERROR`

这相当于协作文档里的“有人已经改了这段，请先刷新再编辑”。

源码证据

- `OpenClaudeCode/src/tools/FileEditTool/prompt.ts:1-25`：提示词层面的“先读后改”“唯一 old_string”规则

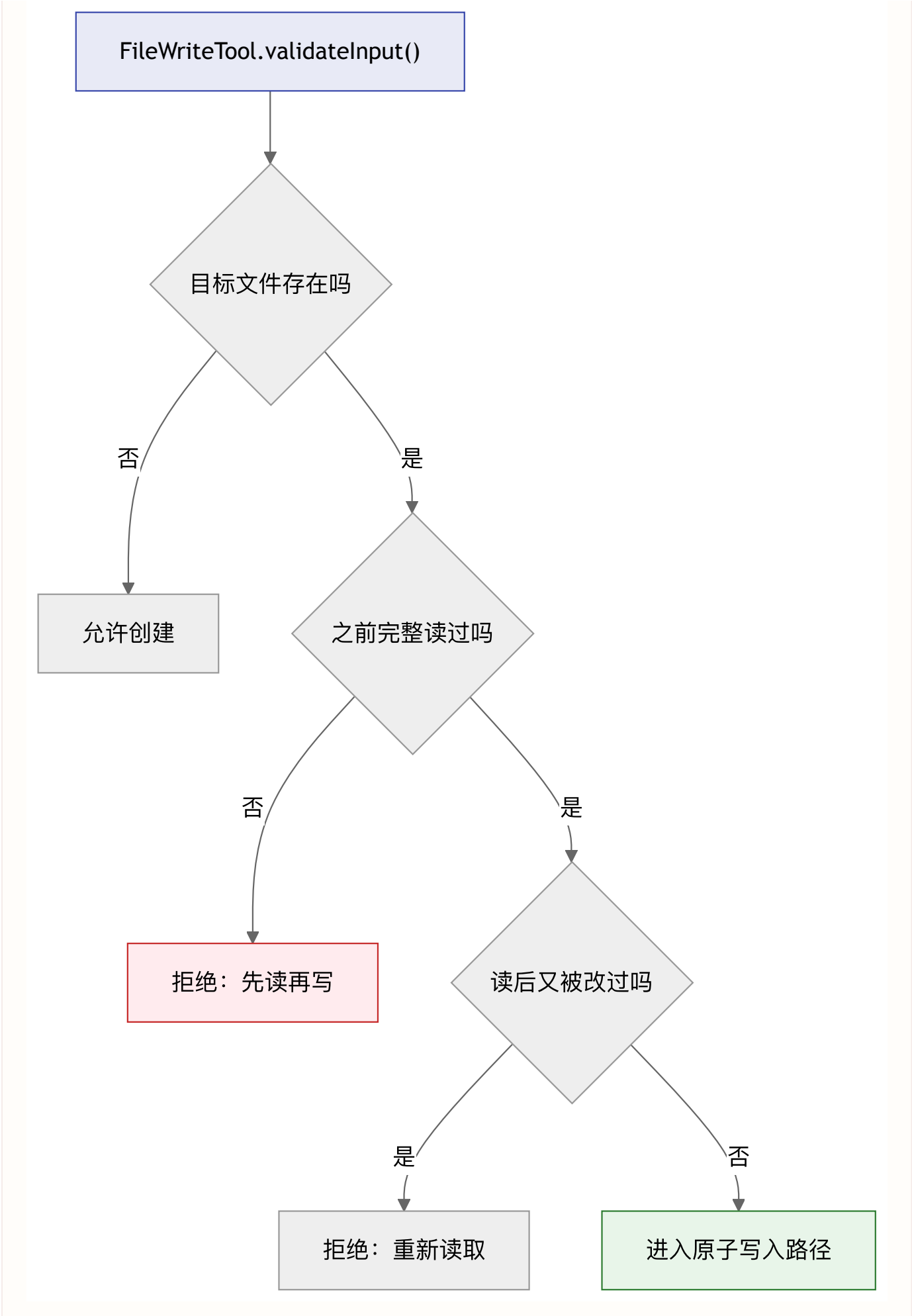
- `OpenClaudeCode/src/tools/FileEditTool/FileEditTool.ts:137-180`：输入验证、deny rule、UNC 风险处理
- `OpenClaudeCode/src/tools/FileEditTool/FileEditTool.ts:275-337`：未读先改、文件变更检测、唯一匹配与 `replace_all`
- `OpenClaudeCode/src/tools/FileEditTool/FileEditTool.ts:442-466`：真正写入前再次确认文件未被意外修改

17.4 Write 工具最怕的，是“整文件覆盖时把别人的改动抹掉”

和 Edit 不同，`FileWriteTool` 负责的是：

- 创建新文件
- 或对已有文件进行整体覆写

风险也因此更大。


FileWriteTool.validateInput()
目标文件存在吗
否
是
允许创建
之前完整读过吗
否
是
拒绝：先读再写
读后又被改过吗
是
否
拒绝：重新读取
进入原子写入路径

它为什么也要求“先读后写”

如果文件本来就存在，Write 会检查：

- `readFileState` 里有没有记录
- 这个记录是不是完整读视图

如果没有，就直接拒绝：

> File has not been read yet. Read it first before writing to it.

原因很好理解：整文件覆写的风险，比精确 edit 更高。

它还会在真正写盘前再做一次 freshness 校验

`call()` 里先：

- `mkdir(dir)`
- 处理 file history
- 再读一次当前文件状态
- 再确认当前内容没偏离上次 read

源码注释特别强调：

> 在 freshness 检查和写入之间尽量不要插入新的异步操作，以保持原子性。

这非常像数据库事务思维：
你不是“最终写成功就行”，而是要尽量缩小“检查完到落盘前”这段竞争窗口。

Write 还会主动区分“创建”和“更新”

如果原文件存在：

- 生成 patch
- 记录为 `update`

如果原文件不存在：

- 记录为 `create`

这说明 Write 并不是“只会把内容写上去”，而是会尽量把结果转换成更可理解的语义。

源码证据

- `OpenClaudeCode/src/tools/FileWriteTool/FileWriteTool.ts:153-221`：先读后写与 freshness 校验
- `OpenClaudeCode/src/tools/FileWriteTool/FileWriteTool.ts:223-280`：call 中的目录准备与原子写前检查
- `OpenClaudeCode/src/tools/FileWriteTool/FileWriteTool.ts:359-430`：update / create 两种结果分支

 深水区（架构师选读）

文件工具这章真正厉害的地方，是它把“看、改、写”三种动作拆成了三种不同的风险模型：

- Read 重点防爆上下文
- Edit 重点防误改位置
- Write 重点防覆盖新鲜内容

这就是成熟工程系统和“万能工具函数”的区别：不是把所有事情都塞进一个入口，而是让每种危险对应一套最适合的约束。

本章小结

一句话：Claude Code 把文件操作拆成 Read / Edit / Write 三把专用工具，不是为了显得复杂，而是为了分别守住“看清事实”“精确替换”“避免覆盖”的三条底线。**

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/tools/FileReadTool/FileReadTool.ts:337-418`	Read 的只读/并发/权限合同
补全层	`OpenClaudeCode/src/tools/FileReadTool/FileReadTool.ts:760-771`	Read 的 token 预算控制
补全层	`OpenClaudeCode/src/tools/FileEditTool/prompt.ts:1-25`	Edit 的提示词约束
补全层	`OpenClaudeCode/src/tools/FileEditTool/FileEditTool.ts:275-337`	未读先改、唯一匹配、`replace_all`
补全层	`OpenClaudeCode/src/tools/FileEditTool/FileEditTool.ts:442-466`	写前 freshness 再确认
补全层	`OpenClaudeCode/src/tools/FileWriteTool/FileWriteTool.ts:153-221`	Write 的先读后写约束
补全层	`OpenClaudeCode/src/tools/FileWriteTool/FileWriteTool.ts:223-280`	原子写前准备与竞争窗口控制
补全层	`OpenClaudeCode/src/tools/FileWriteTool/FileWriteTool.ts:359-430`	create / update 结果语义

逆向提醒

- ✅ **可信度高**：先读后改、唯一匹配、freshness 检查、create/update 区分都在源码里有直接实现
- ⚠️ **要注意语义差别**：Edit 和 Write 都会写文件，但防护目标完全不同
- ❌ **不要误读**：文件工具的“啰嗦”不是低效，而是在主动防止 AI 改错、覆盖错、读半截就乱写

24

搜索工具 第四编

第18章：搜索与上网：AI的眼睛

生活类比

侦探办案至少要有两样东西：一只放大镜，用来在现场找线索；一个情报网，用来从外部获取最新消息。**Claude Code 里的 Grep / Glob 和 WebSearch / WebFetch，正好就是这两类“眼睛”。**

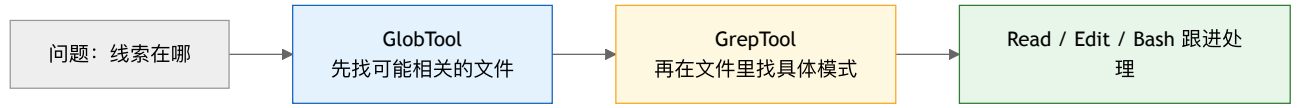
这一章要回答的问题

AI 怎么在你的仓库里快速找线索，又怎么在需要时把视野伸到互联网？为什么这两类能力都被设计成只读、并发安全，但权限模型却不完全相同？

18.1 本地搜索：Grep 和 Glob 负责“在现场找证据”

本地搜索这对搭档的分工非常清楚：

- `GlobTool` 找文件
- `GrepTool` 找内容



GrepTool：内容级搜索

从源码看，GrepTool 的核心特点是：

- `searchHint = search file contents with regex`
- `isConcurrencySafe() = true`
- `isReadOnly() = true`
- 支持 `path` 校验和分页结果输出

它甚至会把搜索结果模式分成：

- `content`
- `count`
- `files_with_matches`

这说明它不是一个简单“执行 rg 的壳”，而是有自己稳定的输出语义。

GlobTool：文件级搜索

GlobTool 更像目录层面的侦察兵：

- 检查 `path` 是否存在且是目录
- 用 `glob(...)` 拉取文件列表
- 限制结果数
- 把路径相对化，节省 token

源码里还会在结果过多时附上：

> Results are truncated. Consider using a more specific path or pattern.

这很像一个训练有素的助手，不只是返回一大坨结果，还会顺手提示你怎么缩小范围。

为什么它们都被标成 `isReadOnly + isConcurrencySafe`

因为搜索最适合并发：

- 搜这个目录
- 再搜那个模式
- 不会彼此踩状态

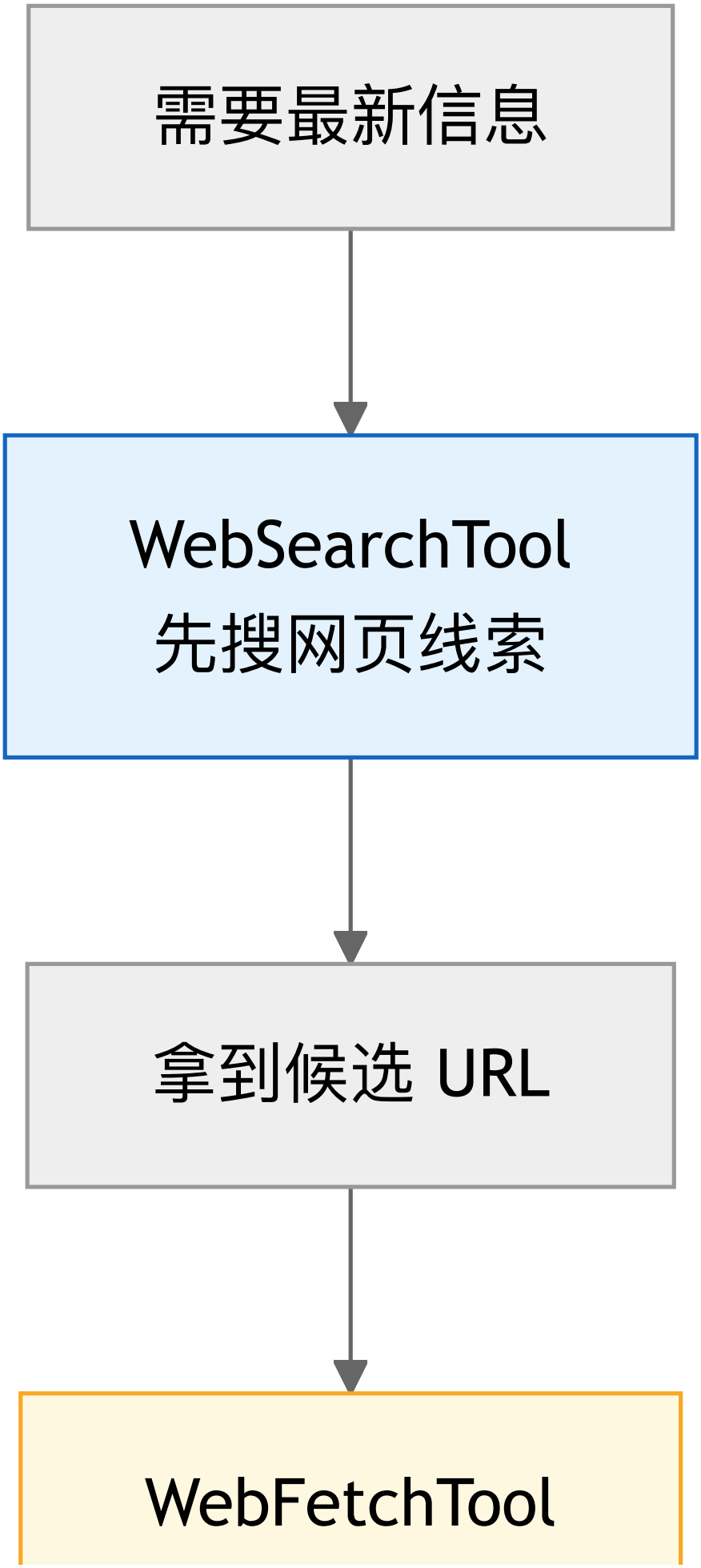
这也是为什么 Claude Code 鼓励“多用专用搜索工具，少用 Bash 自己敲 grep”。

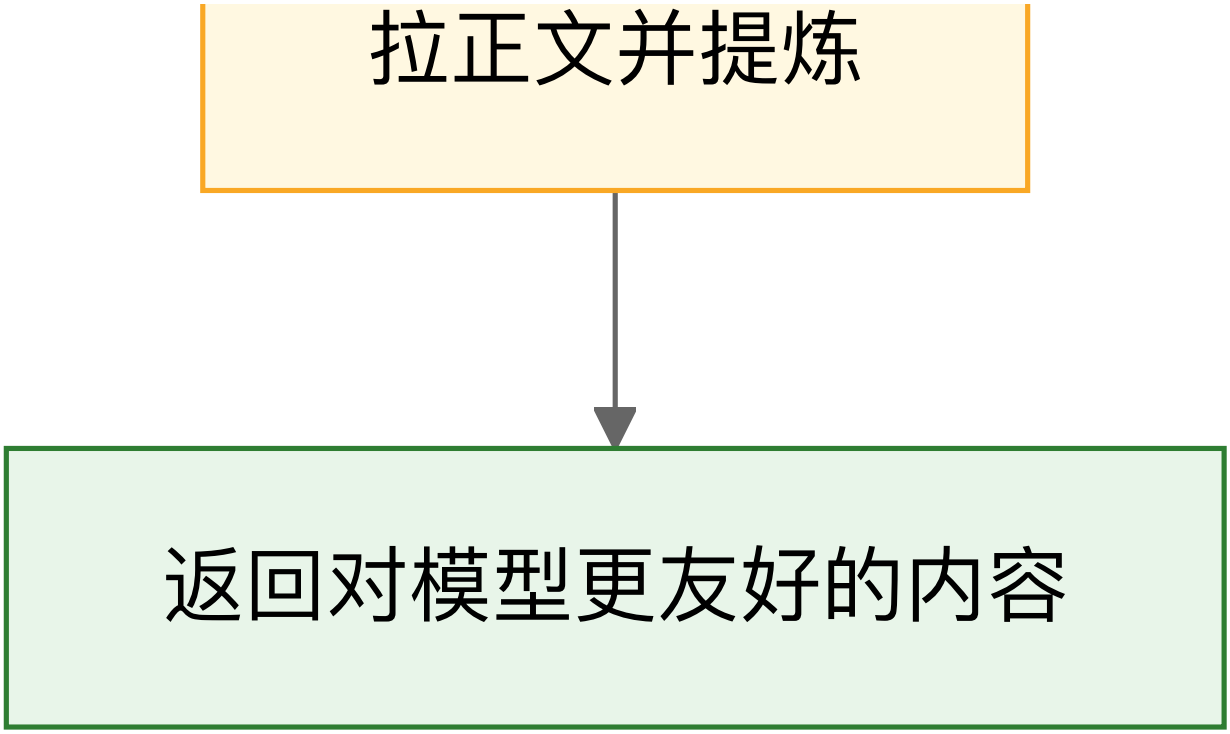
源码证据

- `OpenClaudeCode/src/tools/GrepTool/GrepTool.ts:160-240`：Grep 的只读/并发/权限和路径校验
- `OpenClaudeCode/src/tools/GrepTool/GrepTool.ts:254-280`：分页与结果模式映射
- `OpenClaudeCode/src/tools/GlobTool/GlobTool.ts:94-176`：Glob 的目录校验、结果限制与相对路径输出

18.2 外部搜索：WebSearch 和 WebFetch 负责“把视野伸到仓库外”

如果 Grep / Glob 是放大镜，那么 WebSearch / WebFetch 就是情报网。


需要最新信息
WebSearchTool
先搜网页线索
拿到候选 URL
WebFetchTool



WebSearchTool：先找，再选

它的设计很有意思：

- `shouldDefer = true`
- `max_uses = 8`
- 只有某些 provider / model 组合会启用
- 权限返回的是 `passthrough`，并附带本地允许规则建议

这说明 WebSearch 并不是一个“总在初始工具池里大张旗鼓出现”的工具，而是更偏向：

- 需要时发现
- 需要时开启
- 需要时授权

它甚至知道不同云厂商能力不同

源码里会根据 provider 决定是否启用：

- firstParty：可用
- vertex：只对某些 Claude 4 系列模型可用
- foundry：可用

这再次说明 Claude Code 的工具系统不是“抽象一写就完”，而是会认真处理底层提供方差异。

WebFetchTool：不是只下载 HTML，而是把网页变成模型可消费内容

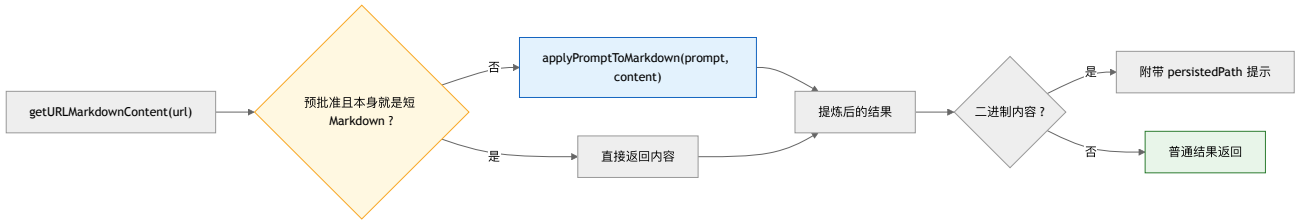
WebFetch 的输入有两个字段：

- `url`
- `prompt`

这已经暴露出它的设计思想：
它不是“原封不动把网页搬回来”，而是“边取内容，边按目标问题提炼”。

在 `call()` 里可以看到两种路径：

- 如果是预批准 URL、内容本来就是合适长度的 Markdown：直接用
- 否则：`applyPromptToMarkdown(...)`



这意味着 WebFetch 并不是一个通用爬虫，而更像一个“网页阅读代理”。

源码证据

- `OpenClaudeCode/src/tools/WebSearchTool/WebSearchTool.ts:76-152`：搜索结果结构化
- `OpenClaudeCode/src/tools/WebSearchTool/WebSearchTool.ts:152-220`：启用条件、defer 和权限 passthrough
- `OpenClaudeCode/src/tools/WebFetchTool/WebFetchTool.ts:66-180`：URL、域名规则与权限模型
- `OpenClaudeCode/src/tools/WebFetchTool/WebFetchTool.ts:208-307`：抓取、转 Markdown、提炼和二进制落盘提示

18.3 为什么 Web 权限和本地搜索权限长得不一样

本地搜索工具通常只需要：

- 路径存在
- 读权限允许

而 WebFetch 还要额外关心：

- 域名是否预批准
- 是否命中 allow / ask / deny 的规则内容
- 是否跳转到别的 host

这说明“读操作”这个词虽然一样，但它的风险边界完全不同：

- 读本地代码：主要是项目权限问题
- 读外部网页：还涉及域名信任、认证资源、跳转风险

WebFetch 对“认证网页”尤其谨慎

它的 prompt 里直接警告：

> 对需要认证或私有 URL，WebFetch 很可能失败，应该优先找专门的 MCP 工具。

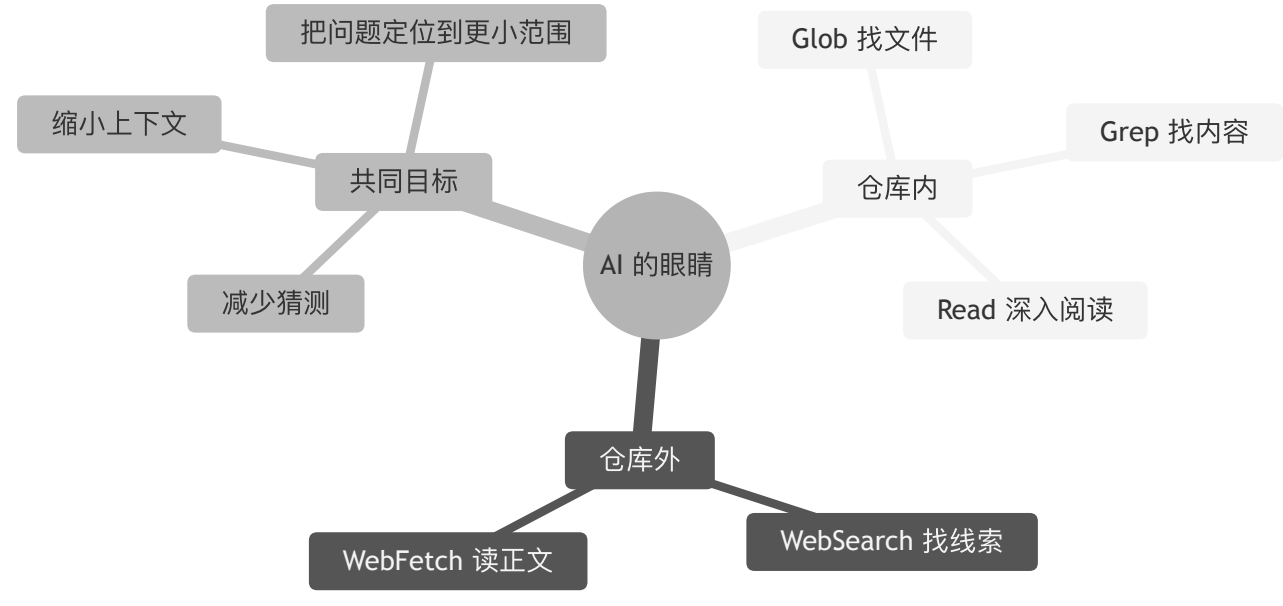
这句话非常值钱，因为它体现了 Claude Code 对“正确入口”的偏好：

- 公开网页：WebFetch 可以
- 私有系统：更应该走受控连接器 / MCP

这不是技术限制而已，更是一种产品边界感。

18.4 搜索工具的真正价值：让模型别再“瞎猜”

这一章的主线并不只是列工具，而是解释 Claude Code 为什么要给模型装上“眼睛”。



一个没有搜索能力的模型，常常只能：

- 猜文件名
- 猜 API
- 猜文档版本

而 Claude Code 的设计目标很明确：
能查就不要猜，能找就不要拍脑袋。

深水区（架构师选读）

第 18 章把 Claude Code 的一个根本设计哲学暴露得很清楚：它不是把模型当“全知者”，而是把模型当“会主动检索证据的执行者”。

这也是现代 Agent 系统的分水岭：

- 弱系统：让模型靠记忆硬答
- 强系统：让模型先找证据，再组织答案

Grep / Glob 负责项目内证据，WebSearch / WebFetch 负责项目外证据，两者合起来，才构成了“可工作的 AI 视野”。

本章小结

一句话：Grep 和 Glob 让 Claude Code 在仓库内部精准找线索，WebSearch 和 WebFetch 让它在仓库外获取最新证据；这四个工具共同构成了 AI 的“眼睛”。**

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/tools/GrepTool/GrepTool.ts:160-240`	Grep 的合同、路径校验与权限
补全层	`OpenClaudeCode/src/tools/GrepTool/GrepTool.ts:254-280`	Grep 的结果模式与分页
补全层	`OpenClaudeCode/src/tools/GlobTool/GlobTool.ts:94-176`	Glob 的目录校验与受限输出
补全层	`OpenClaudeCode/src/tools/WebSearchTool/WebSearchTool.ts:76-220`	WebSearch 的启用条件、defer 和权限
补全层	`OpenClaudeCode/src/tools/WebFetchTool/WebFetchTool.ts:66-180`	WebFetch 的域名权限与输入验证
补全层	`OpenClaudeCode/src/tools/WebFetchTool/WebFetchTool.ts:208-307`	WebFetch 的抓取、提炼和二进制内容处理

逆向提醒

- ✅ **可信度高**：本地搜索、Web 搜索和抓取的分工都能在源码里直接看到
- ⚠️ **要看权限差异**：本地读权限和外部域名权限是两套不同问题
- ❌ **不要误读**：WebFetch 不是万能浏览器；对认证网页和私有系统，Claude Code 更偏向 MCP 或专用连接器

25

AgentTool 第四编

第19章：高级三角：Agent+Skill+MCP

生活类比

一个人会干活，是能力；会带人、会学新方法、会借外部资源，才叫体系。**AgentTool、SkillTool、MCPTool 这三者，正是 Claude Code 从“单个助手”走向“平台能力”的三根支柱。**

这一章要回答的问题

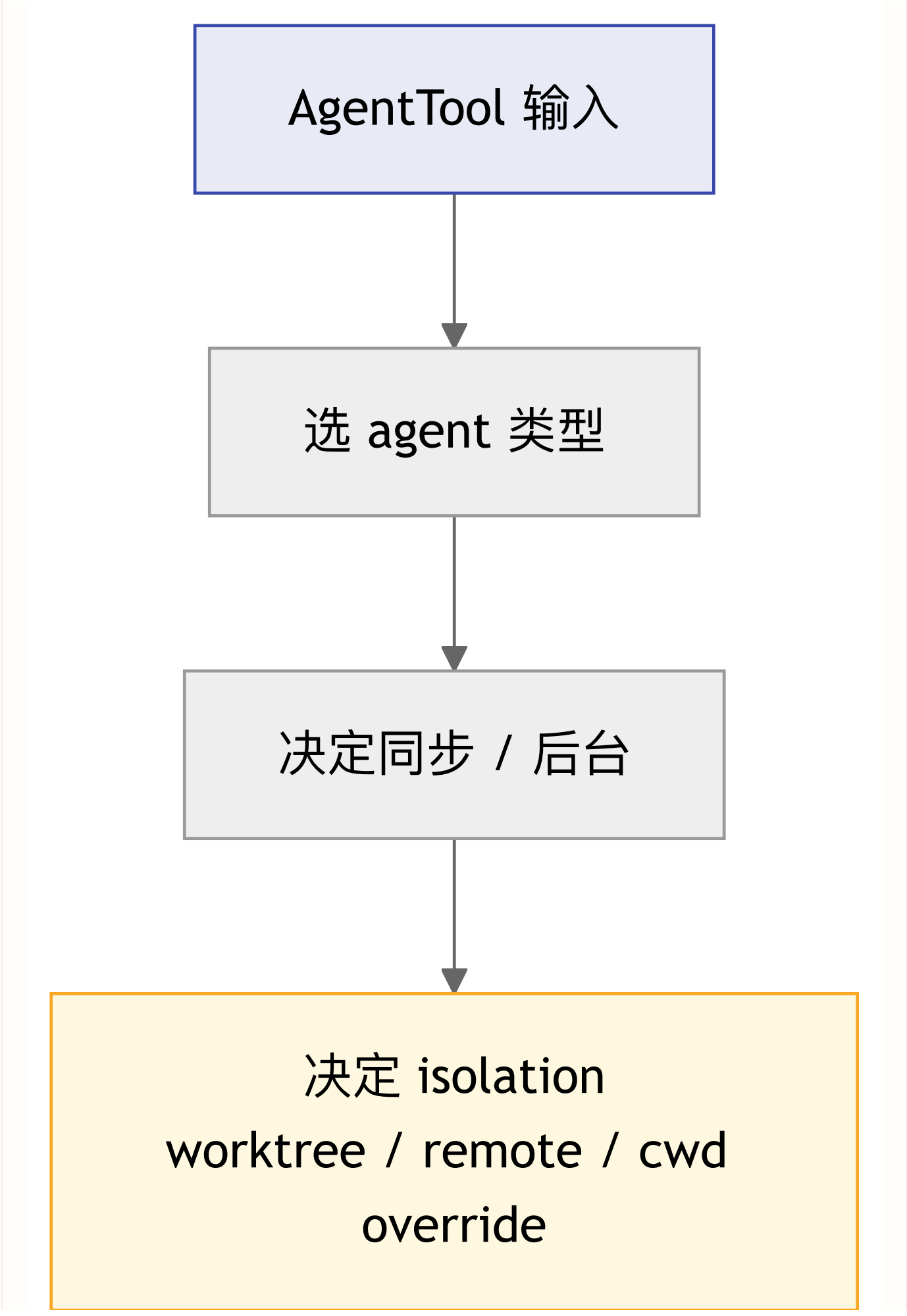
Claude Code 到底只是一个会改代码的 CLI，还是一个可编排、可扩展、可接外部能力的 AI 平台?

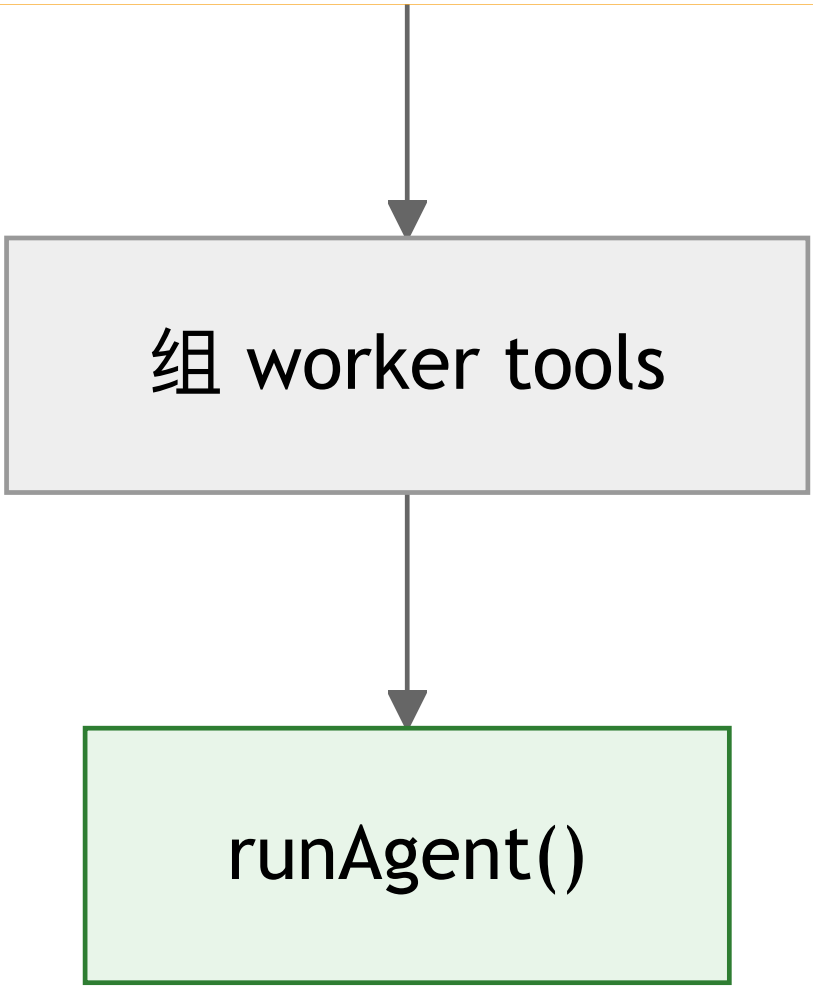
第 19 章的答案是：从源码看，它明显在往后者走。

19.1 AgentTool：让 Claude Code 从“自己干”变成“会派活”

`AgentTool.tsx` 的输入 schema 已经说明了它不是普通工具：

- `description`
- `prompt`
- `subagent_type`
- `model`
- `run_in_background`
- `team_name`
- `mode`
- `isolation`
- `cwd`


AgentTool 输入
选 agent 类型
决定同步 / 后台
决定 isolation
worktree / remote / cwd
override



它不是只会“开一个子进程”

AgentTool 实际上能走很多条路：

- 普通 subagent
- teammate spawn
- background async agent
- worktree isolated agent
- remote launched agent

这说明它本质上是一个**任务派发器**。

worker 不会无脑继承父工具池

源码里有一段特别值得注意：

- worker 的工具池通过 `assembleToolPool(workerPermissionContext, appState.mcp.tools)` 单独组装
- 它使用的是 worker 自己的 permission mode
- 不直接继承父工具限制

这意味着子智能体不是父智能体的“影子”，而是一个有自己工作权限边界的执行单元。

worktree isolation 是平台级能力，不是噱头

如果设置了 `isolation = worktree`：

- 会为 agent 创建独立 worktree
- fork path 下还会额外注入 worktree notice
- 任务结束后，没改动就自动回收；有改动就保留 worktree

这和“开一个线程”完全不是一个量级的设计。
它已经在认真处理多智能体协作里的文件隔离问题了。

源码证据

- `OpenClaudeCode/src/tools/AgentTool/AgentTool.tsx:81-155`：AgentTool 输入 / 输出 schema
- `OpenClaudeCode/src/tools/AgentTool/AgentTool.tsx:282-316`：teammate spawn 分支

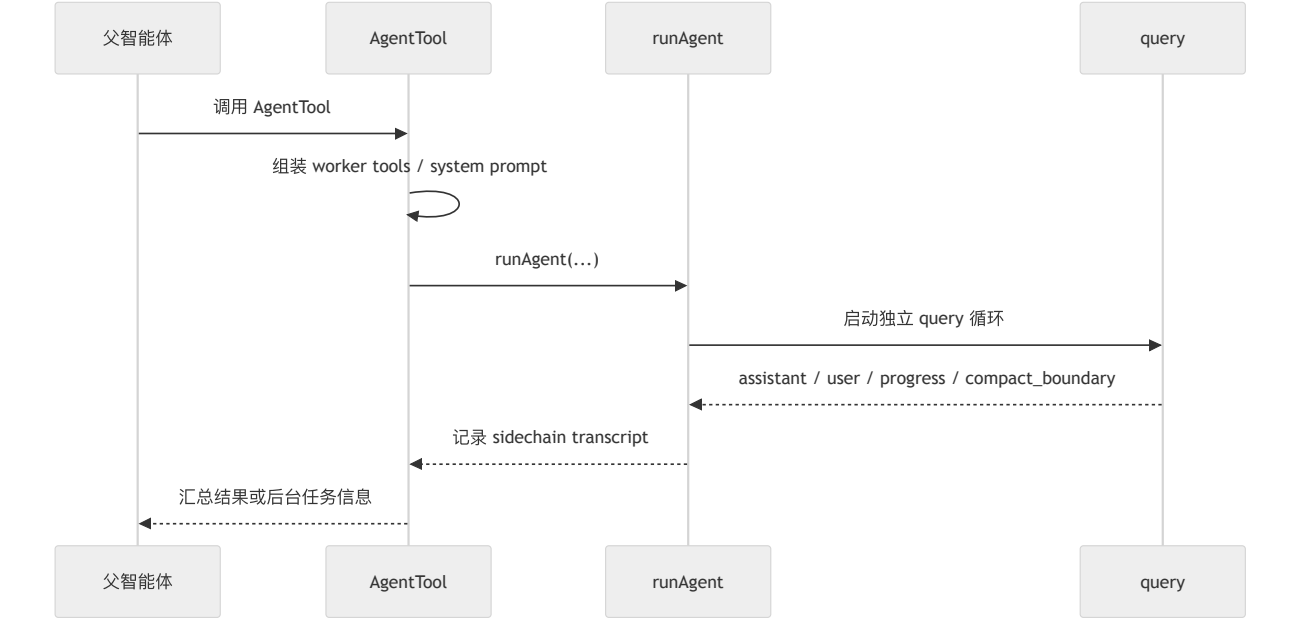
- `OpenClaudeCode/src/tools/AgentTool/AgentTool.tsx:567-641`：worker tool pool、async 判断、worktree / cwd 装配
- `OpenClaudeCode/src/tools/AgentTool/AgentTool.tsx:643-685`：worktree 清理与保留逻辑

19.2 `runAgent()` 说明：子智能体不是“函数调用”，而是一段独立会话

如果说 AgentTool 负责决定“派不派”，那 `runAgent()` 负责真正把一名子智能体跑起来。

从 `runAgent.ts` 的参数看，它至少关心：

- `agentDefinition`
- `promptMessages`
- `toolUseContext`
- `isAsync`
- `forkContextMessages`
- `availableTools`
- `allowedTools`
- `worktreePath`
- `description`



`forkContextMessages` 是一个很关键的开关

源码注释写得很明确：

- 普通 agent：按自己的 system prompt、上下文和工具集运行
- fork path：为了 prompt cache 命中，要尽量保持和父请求前缀一致

这说明 Claude Code 连“父子 agent 之间的缓存连续性”都考虑到了。

子智能体也有自己的 transcript 和清理逻辑

`runAgent()` 不只是跑起来，还负责：

- 记录 sidechain transcript
- 处理 `max_turns_reached`
- 清理 agent 专属 MCP server
- 清理 readFileState
- 清理 todos
- 杀掉 agent 生成的后台 bash / monitor 任务

这说明子智能体不是“用完就丢的 Promise”，而是被当成一等公民去管理生命周期。

源码证据

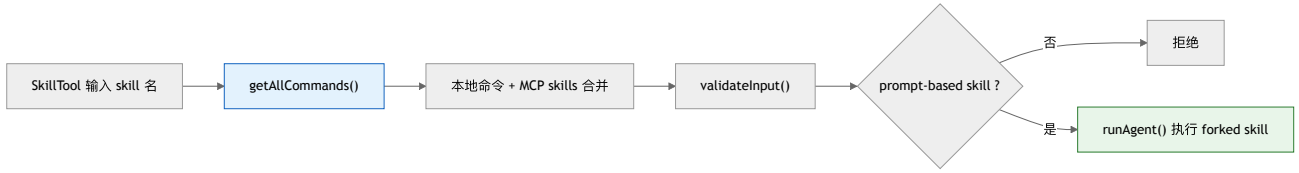
`OpenClaudeCode/src/tools/AgentTool/runAgent.ts:248-329` 展示了子智能体运行参数；
`OpenClaudeCode/src/tools/AgentTool/runAgent.ts:792-859` 展示了 transcript 与清理逻辑。

19.3 SkillTool：把“技能”变成可调用的能力包

SkillTool 是这一组三角里最像“给 AI 装插件大脑”的部分。

它不是直接执行一个 shell 命令，而是：

- 校验 skill 名称
- 从本地命令和 MCP skill 中找技能
- 判断它是不是 prompt-based command
- 再决定 inline 处理还是 forked skill 执行



这说明 skill 不是“神秘 prompt 片段”

它是有完整生命周期的：

- 可以被发现
- 可以被校验
- 可以被权限规则匹配
- 可以被 fork 到子 agent 中执行

SkillTool 甚至把 MCP prompt 型技能也纳进来了

`getAllCommands()` 会把：

- 本地 / bundled commands
- `loadedFrom === 'mcp'` 的 MCP skills

合并起来。

这意味着 skill 系统已经不是“本地 markdown 文件的小花活”，而是在朝统一技能总线发展。

`executeForkedSkill()` 的平台味道很浓

在 forked skill 路径里，它会：

- 创建新 `agentId`
- 准备 forked command context
- 调 `runAgent()`
- 收集 agent messages
- 把其中的 tool_use / tool_result 转成 progress

也就是说，SkillTool 本质上是：

> 用一个更窄、更预先设计好的 prompt，把 AgentTool 的派遣能力包装成“技能调用”。

源码证据

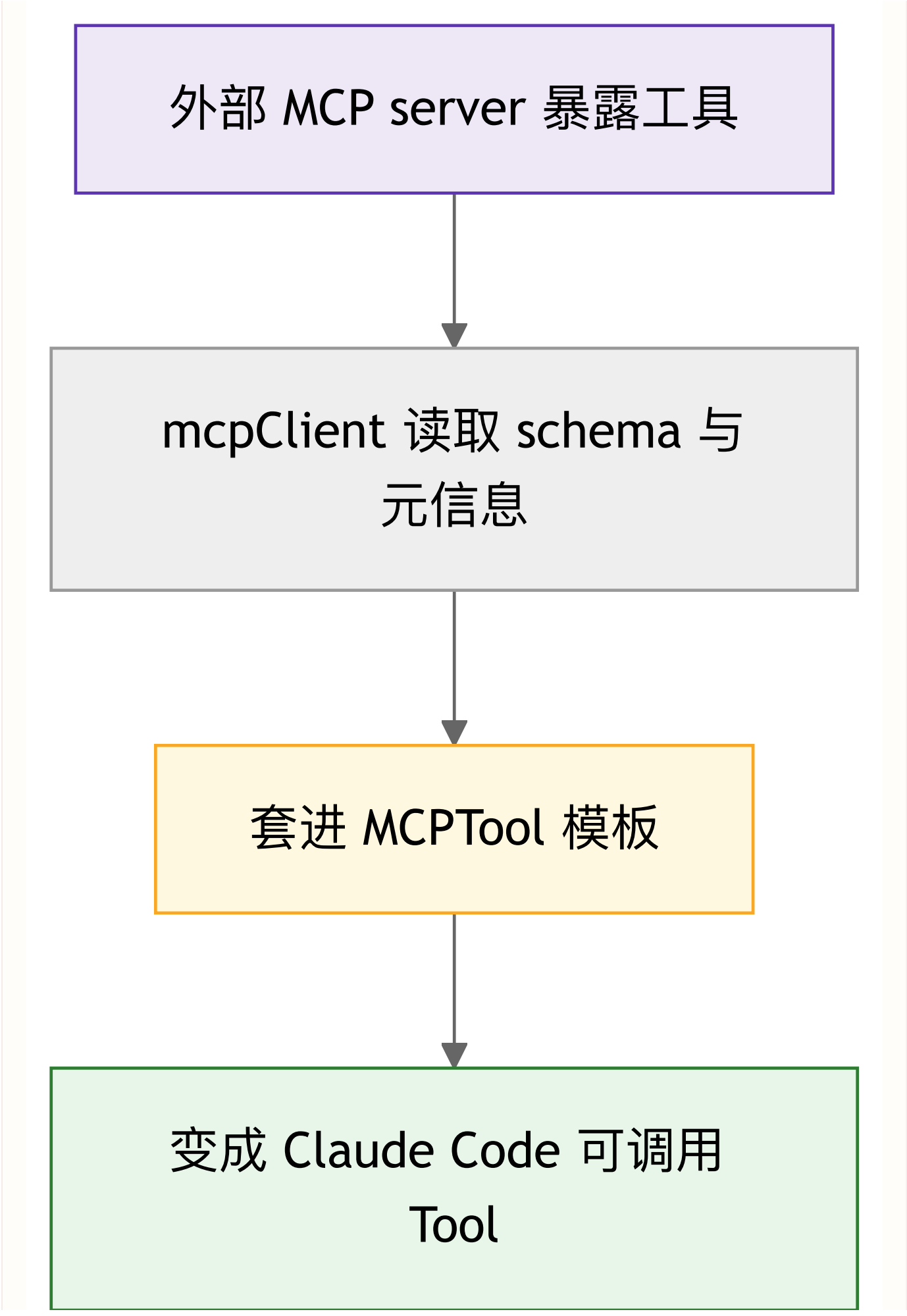
- `OpenClaudeCode/src/tools/SkillTool/SkillTool.ts:81-94`：本地命令与 MCP skills 合并
- `OpenClaudeCode/src/tools/SkillTool/SkillTool.ts:118-260`：forked skill 的执行路径
- `OpenClaudeCode/src/tools/SkillTool/SkillTool.ts:331-429`：SkillTool 的 schema 与输入校验

19.4 MCPTool：不是一个具体工具，而是“外部工具适配模板”

MCPTool 很容易被误解成“又一个工具”。
其实它更像一个**适配器模板**。

源码里能看到：

- `isMcp = true`
- `inputSchema = passthrough object`
- `name = 'mcp'`
- `description / prompt / call` 都标注为会在 `mcpClient.ts` 中被覆写


外部 MCP server 暴露工具
mcpClient 读取 schema 与元信息
套进 MCPTool 模板
变成 Claude Code 可调用 Tool

这说明 Claude Code 的野心很清楚

它不想把“工具”只定义成自己仓库里写死的东西。
它想定义一套标准，然后让外部能力也能变成自己的工具。

为什么 `inputSchema` 要用 passthrough

因为 MCP 工具的 schema 来自外部服务器，不可能在编译期全知道。
所以 MCPTool 本体只提供一个可接任何对象的底座，再由客户端层注入真实 schema。

权限这里仍然要经过 Claude Code

即使是外部能力，`checkPermissions()` 也不是绕过去，而是返回 `passthrough`，继续进入 Claude Code 自己的权限体系。

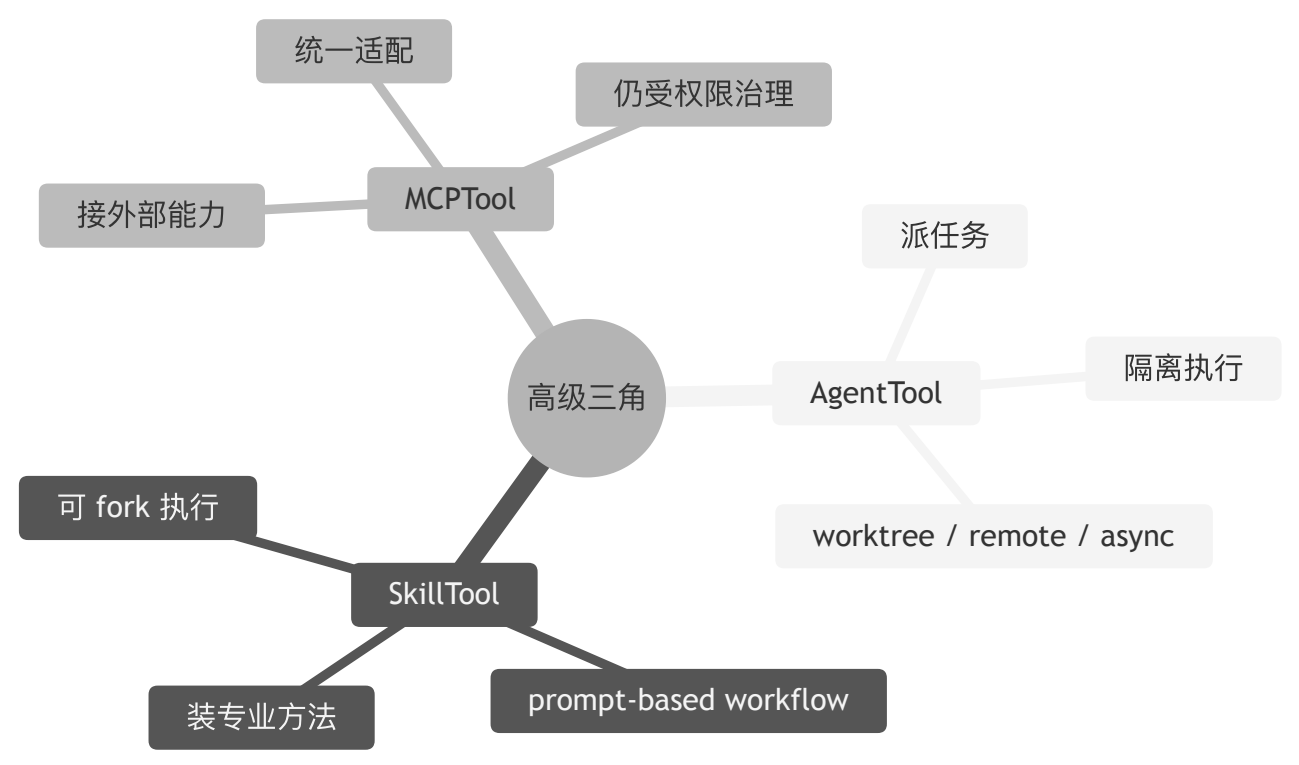
这意味着 MCP 并没有把系统变成“随便接个外部服务就裸奔”，而是接进了同一套工具治理框架。

源码证据

`OpenClaudeCode/src/tools/MCPTool/MCPTool.ts:13-77` 展示了 MCPTool 作为适配模板而非具体业务工具的定位。

19.5 为什么说 Agent + Skill + MCP 构成了“平台三角”

到这里，三者的分工可以非常清楚地画出来：



它们分别解决三种不同的“扩展”：

维度	对应能力	回答的问题
纵向扩展	AgentTool	一个人做不完，能不能拆任务给别人
认知扩展	SkillTool	不会的事，能不能临时装上方法论
外部扩展	MCPTool	本地没有的能力，能不能接进来

这就是为什么本章叫“高级三角”。
它们共同把 Claude Code 从“一个会改代码的终端助手”推进成“一个可编排、可扩展、可联外部资源的平台”。

深水区（架构师选读）

第 19 章的真正价值，不是告诉你有三个高级工具，而是让你看清 Claude Code 的平台化方向：

- AgentTool 负责把单线程任务执行扩展成多执行体协作

- SkillTool 负责把经验和工作流模块化
- MCPTool 负责把外部世界接入同一条工具总线

这三者加在一起，就不再是“工具多一点”，而是系统架构级别的跃迁。它意味着 Claude Code 的边界不再等于自己的仓库。

本章小结

一句话：AgentTool 让 Claude Code 会派活，SkillTool 让它会按方法做事，MCPTool 让它能接入外部能力；三者合在一起，构成了 Claude Code 迈向平台化的高级三角。**

关键源码索引

证据层	文件	本章关注点
补全层	`OpenClaudeCode/src/tools/AgentTool/AgentTool.tsx:81-155`	AgentTool 输入 / 输出 schema
补全层	`OpenClaudeCode/src/tools/AgentTool/AgentTool.tsx:282-316`	teammate spawn 路径
补全层	`OpenClaudeCode/src/tools/AgentTool/AgentTool.tsx:567-641`	worker tool pool、async、worktree / cwd 组装
补全层	`OpenClaudeCode/src/tools/AgentTool/runAgent.ts:248-329`	子智能体运行参数与上下文
补全层	`OpenClaudeCode/src/tools/AgentTool/runAgent.ts:792-859`	transcript 与生命周期清理
补全层	`OpenClaudeCode/src/tools/SkillTool/SkillTool.ts:81-94`	本地与 MCP skills 合并
补全层	`OpenClaudeCode/src/tools/SkillTool/SkillTool.ts:118-260`	forked skill 执行
补全层	`OpenClaudeCode/src/tools/SkillTool/SkillTool.ts:331-429`	SkillTool schema 与输入校验
补全层	`OpenClaudeCode/src/tools/MCPTool/MCPTool.ts:13-77`	MCPTool 适配模板

逆向提醒

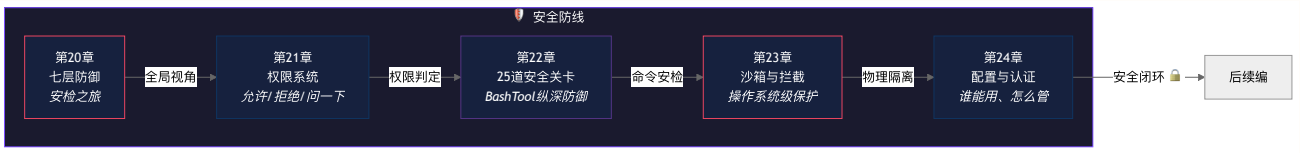
- ✅ **可信度高**：AgentTool 的多路径派发、SkillTool 的 fork 执行、MCPTool 的适配模板性质，都能从源码直接印证
- ⚠️ **要理解角色差别**：MCPTool 更像模板和桥接层，不是一个单独业务工具
- ❌ **不要误读**：这三个能力不是互相替代，而是分别扩展“执行体、方法论、外部资源”三条维度

26

第五编：安全防线

> *机场安检：行李过 X 光、人过安检门、可疑物开箱检查——没有单一防线是完美的，但层层叠加后威胁穿透的概率趋近于零。*
>
> 本编解剖 Claude Code 的安全架构：**七层防御体系**、**权限判定系统**、**BashTool 25 道安全关卡**、**操作系统级沙箱**、**配置与认证**。

本编总览



本编五章速览

章	标题	核心问题	生活类比
20	七层防御	一个能执行 `rm -rf` 的 AI，你敢用吗？	机场七层安检
21	权限系统	为什么有的操作直接执行、有的要问、有的直接拒绝？	手机权限弹窗
22	25道安全关卡	25 道关卡具体每道检查什么？能全部绕过吗？	银行金库25道门禁
23	沙箱与拦截	应用层检查不够，操作系统还能做什么？	化学实验室安全柜
24	配置与认证	配置散在5个地方、认证散在3个协议——怎么管？	公司门禁系统

设计思想主线

本编建立的认知基础

1. 信任不靠承诺建立，靠**架构**建立——假设 AI 随时可能犯错
2. 七层独立防线确保**即使 AI 犯错，错误也会被拦截**
3. 权限判定不是随机的——**三级分类（直接执行/需确认/直接拒绝）**有精确的判定逻辑
4. BashTool 的 25 道安全关卡是**纵深防御的教科书级实现**
5. 操作系统级沙箱是**最后一道物理防线**——即使应用层被攻破

推荐路径

○ ○ ○

🌱 初学者 🔧 开发者 🏗️ 架构师

第20章的七层防御提供了全局视角——**先理解"为什么需要这么多层"**。

第21章的权限系统和第22章的安全关卡是**构建安全 AI 应用的实战参考**。

第23章的沙箱设计和第24章的配置架构展示了**企业级安全的完整方案**。

阅读建议

如果你想先抓住全局，按 **第20章 → 第21章 → 第23章** 读；如果你最关心“Bash 到底危险在哪”，直接跳到 **第22章**。第24章则适合放在最后看，它会把前面所有安全能力重新接回“组织治理”和“长期会话”这两个更大的问题上。

27

安全 第五编

第20章：七层防御：一条命令的安检之旅

生活类比：机场安检

机场不会只靠一道门保护你。行李要过 X 光，人要过安检门，可疑物还要开箱复查，最后还有隔离区和登机口复核。Claude Code 的安全设计也不是“靠 AI 自己别犯错”，而是假设它**一定会犯错**，于是把风险拆成多层来挡。

这一章先回答一个问题

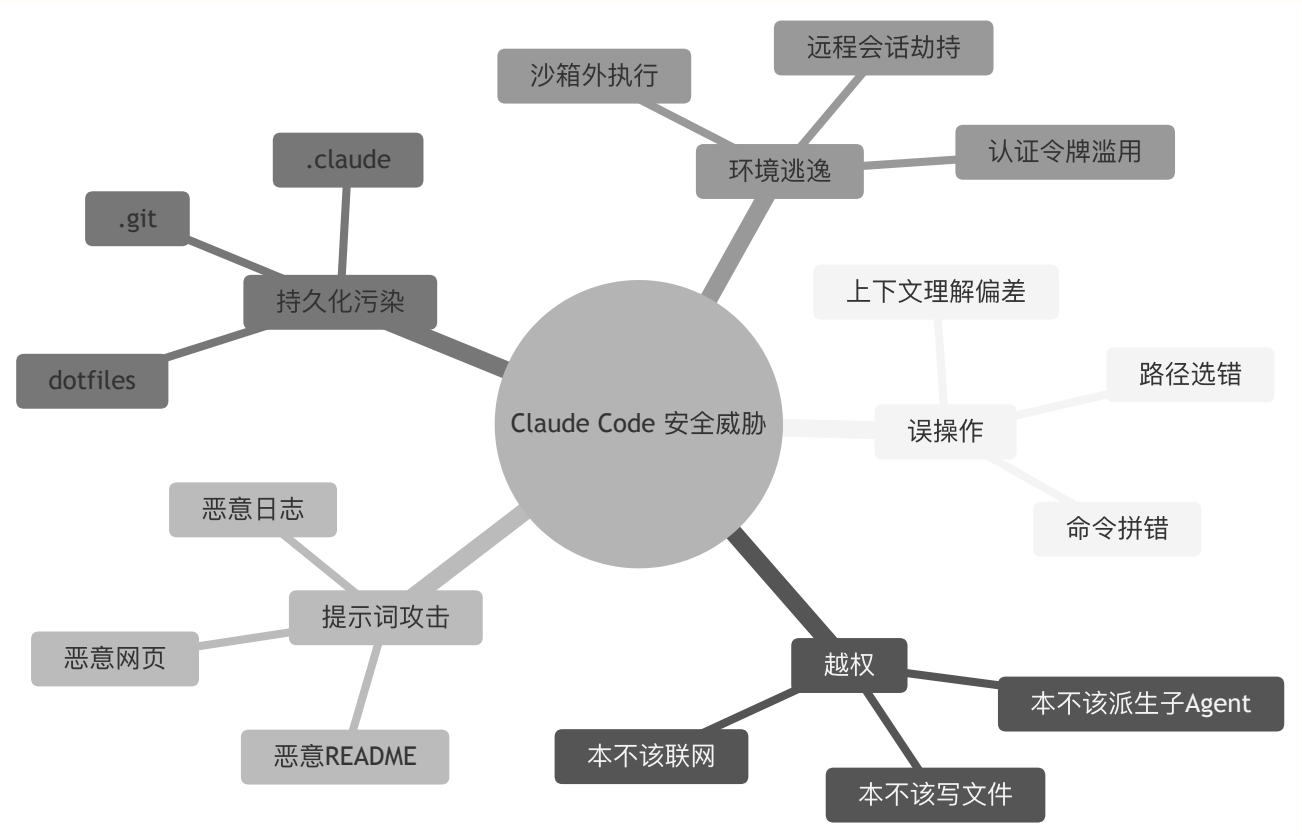
一个能执行 `rm -rf`、改文件、联网、开子 Agent 的 AI 助手，为什么还能被人放心地放进真实工程里？

Claude Code 的答案不是一句“我们模型很聪明”，而是一整条**源码可追踪的安全链**。从权限模式、规则匹配、工具自检，到 Hook、分类器、沙箱、策略与认证，任何一层出错，后面还有别的层补上。

20.1 先别急着看代码，先看它在防什么

源码里反复出现的不是“能力”，而是“约束”。这通常意味着作者面对的是几类真实威胁：

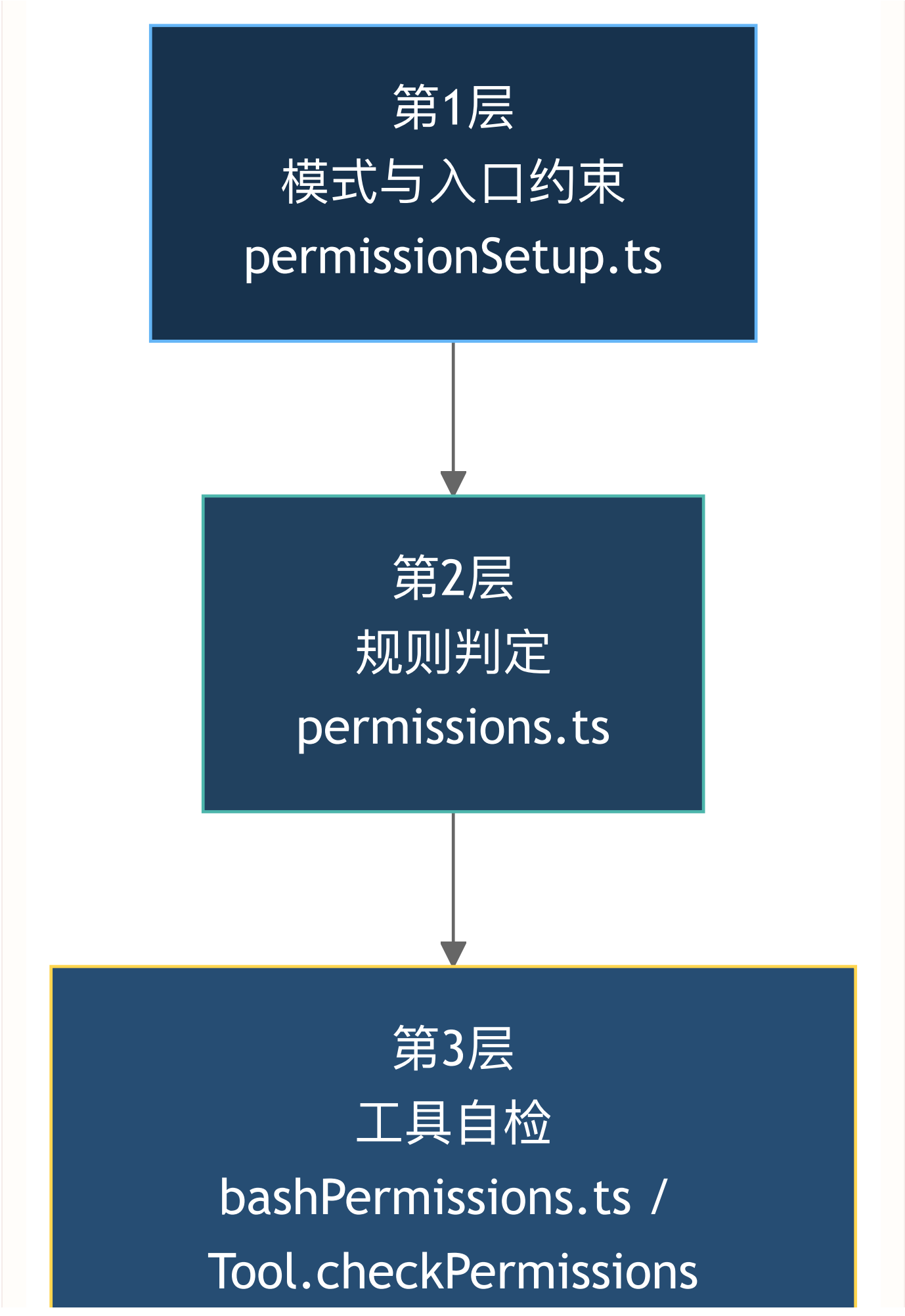
- **误操作**：模型本意是好的，但把路径、参数、上下文理解错了。
- **越权操作**：模型能做的事太多，顺手越过了当前任务边界。
- **提示词攻击**：恶意文件、日志、网页内容诱导模型执行危险动作。
- **持久化污染**：把风险写进 `.git`、`.claude`、shell 配置或凭据目录。
- **环境逃逸**：应用层判断失手后，命令直接碰到系统资源。

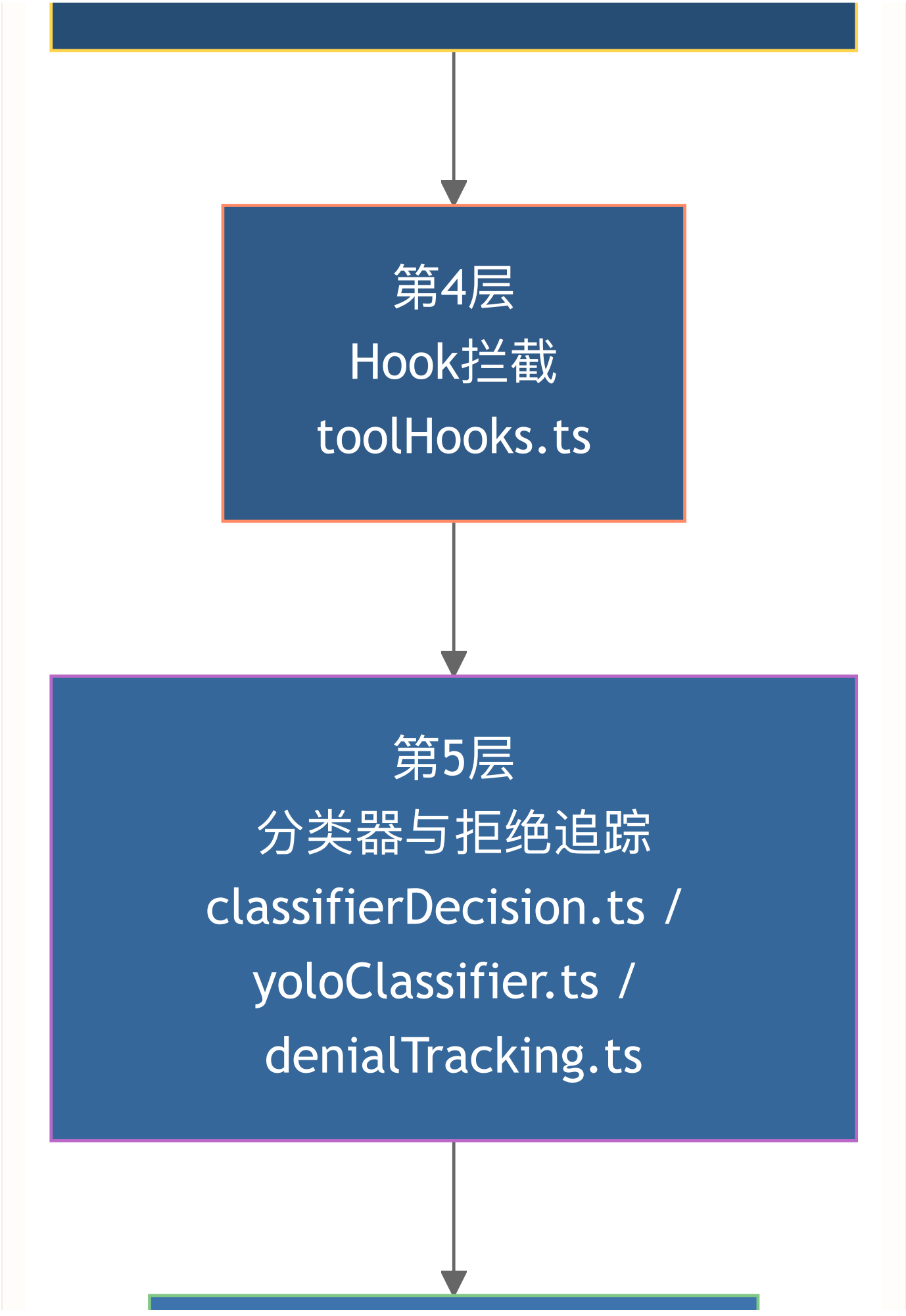


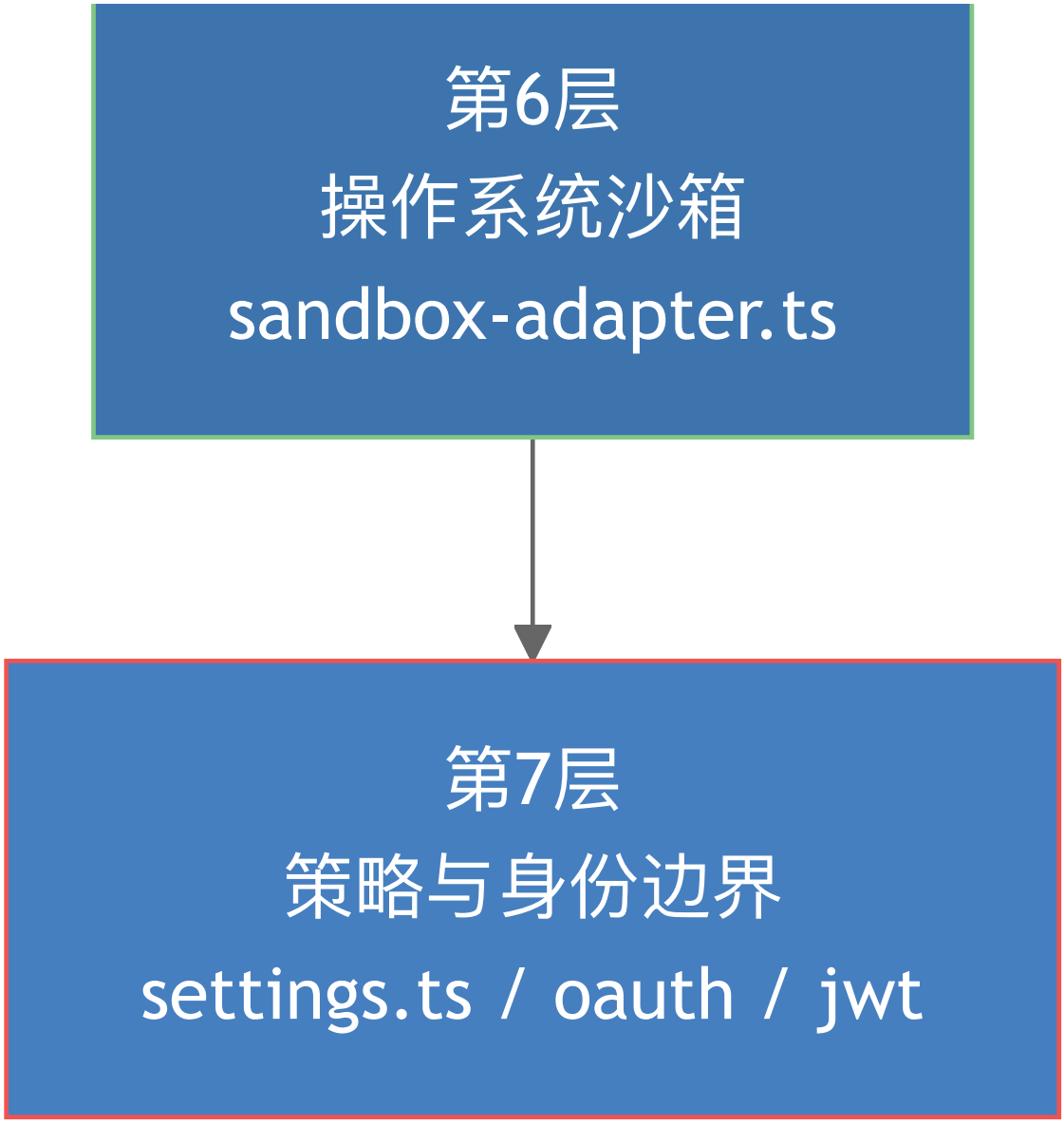
如果只靠“执行前弹个确认框”，这些风险根本挡不住。因为很多危险操作看起来像普通操作，很多普通操作连着做也会变危险。

20.2 从源码角度，可以把它归纳成七层防御

这里的“七层”不是某一个枚举常量的名字，而是我们根据调用链整理出的**分析框架**。你会在不同文件里看到这些层一起工作。


第1层
模式与入口约束
permissionSetup.ts
第2层
规则判定
permissions.ts
第3层
工具自检
bashPermissions.ts /
Tool.checkPermissions


第4层
Hook拦截
toolHooks.ts
第5层
分类器与拒绝追踪
classifierDecision.ts /
yoloClassifier.ts /
denialTracking.ts



第 1 层：模式与入口约束

权限模式不是 UI 装饰，而是第一层开关。`PermissionMode.ts` 定义了 `default`、`plan`、`acceptEdits`、`bypassPermissions`，以及在特性开启时才出现的 `auto` 模式；`permissionSetup.ts` 再把 CLI 参数、settings、GrowthBook 门控合并成实际模式。这样一来，很多风险在“命令还没跑”之前就被模式收窄了。

第 2 层：规则判定

`permissions.ts` 先查整工具 deny，再查 ask，再交给工具自己的 `checkPermissions`。这是 Claude Code 最核心的一层：它先问“这类行为该不该做”，而不是先跑完再回头解释。

第 3 层：工具自检

最典型的是 Bash。`bashPermissions.ts` 不是一句正则，而是把路径、sed、只读命令、安全模式、沙箱自动放行、最终提示拆成多个阶段。也就是说，即使用户已经给过某些大类权限，工具自己还会继续做二次筛查。

第 4 层：Hook 拦截

`toolHooks.ts` 允许在工具前后插入外部治理逻辑。但它又特别小心：Hook 即使返回 `allow`，也**不会绕过** settings 里的 deny/ask 规则。这是很成熟的安全思路。

第 5 层：分类器与拒绝追踪

自动模式不是“全部自动通过”。`classifierDecision.ts` 先区分一批天然安全工具，剩下的交给 `yoloClassifier.ts` 做转录压缩、系统提示拼装和风险分类；`denialTracking.ts` 则负责在连续拒绝太多时回退到人工确认，避免自动模式越跑越偏。

第 6 层：操作系统沙箱

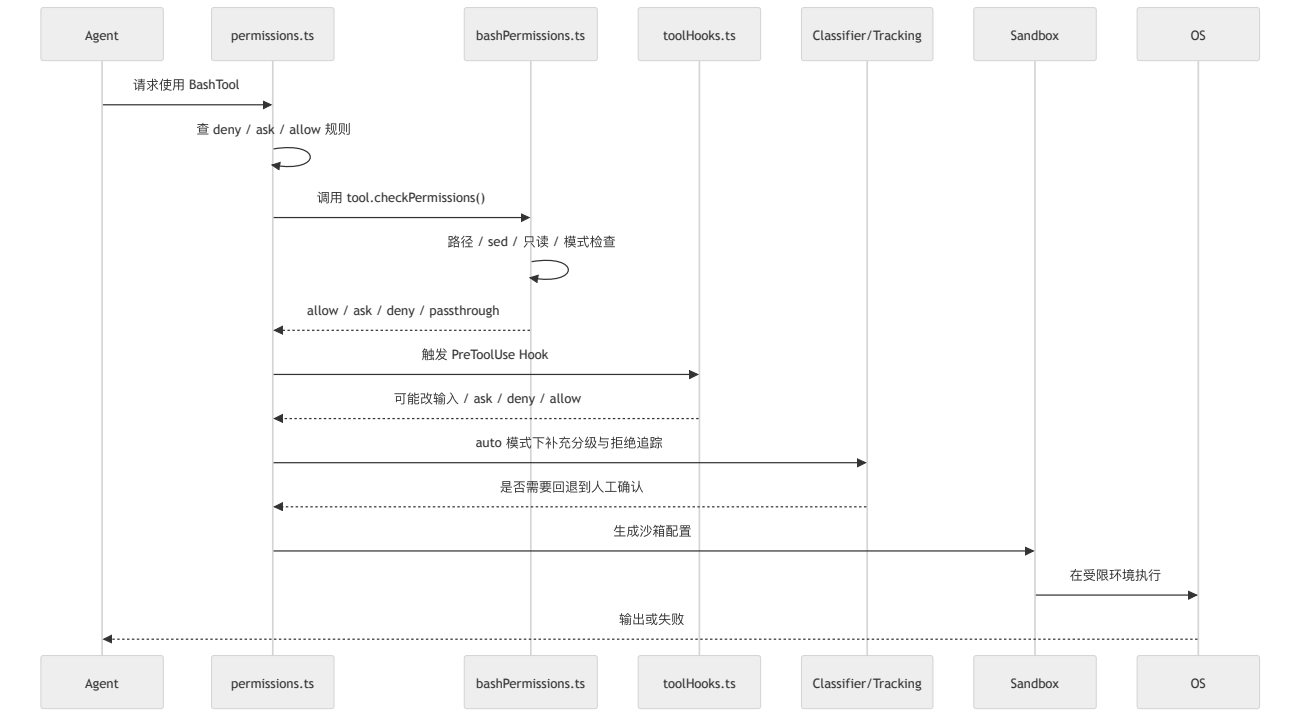
`sandbox-adapter.ts` 把 Claude Code 自己的权限规则翻译成底层沙箱配置，包括可写目录、禁止写入的设置文件、网络域名等。它不是“更漂亮的提示框”，而是系统级隔离。

第 7 层：策略与身份边界

最后还有 MDM、managed settings、OAuth、Bridge JWT 刷新这些“外围护城河”。它们不直接判断 `rm -rf`，但决定了谁能进入某种模式、谁能拿到某种令牌、哪些组织策略不可绕过。

20.3 一条 Bash 命令要过哪些检查

把源码摊开看，一条命令的安全旅程差不多是这样的：



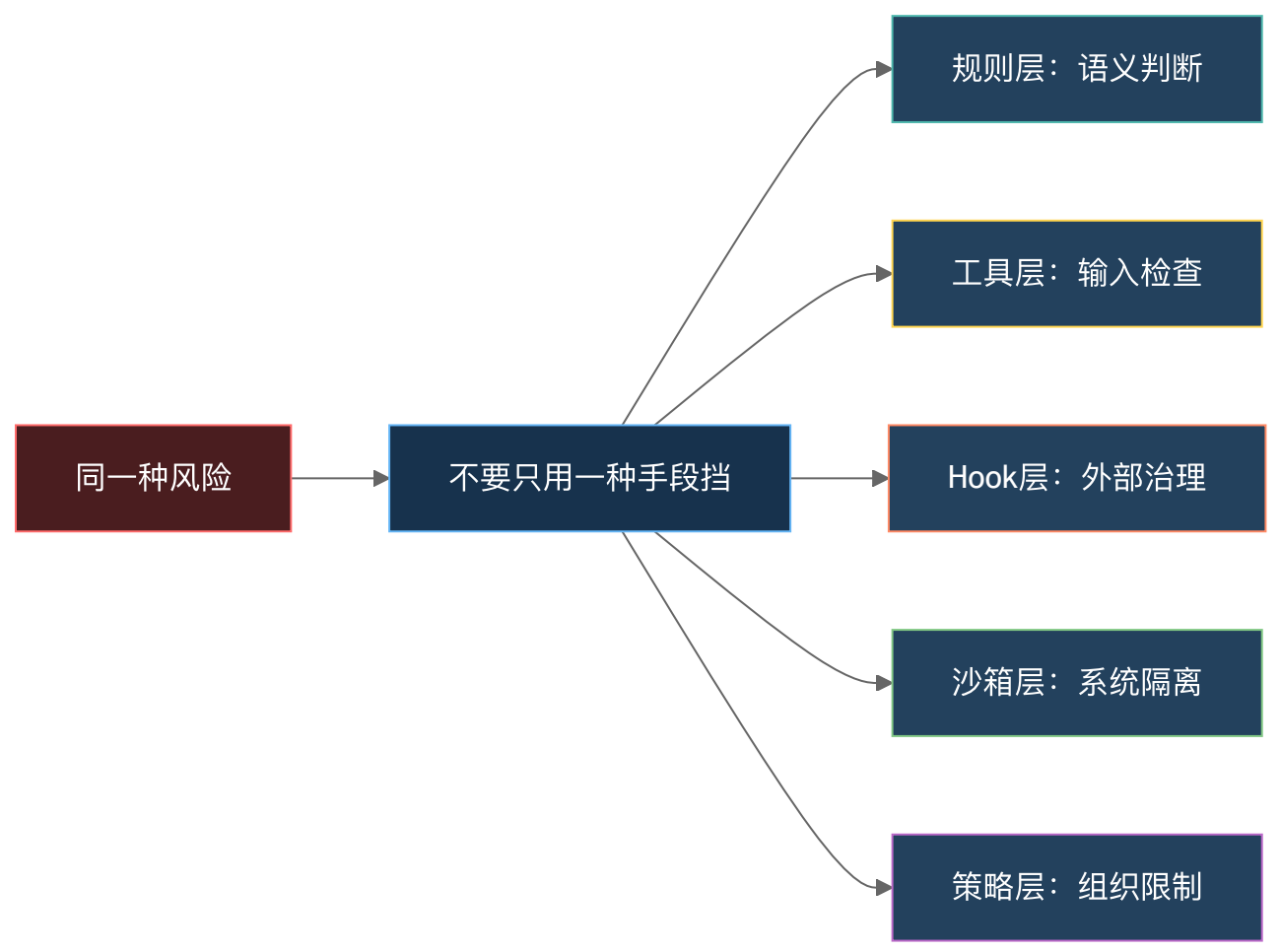
如果我们把关键代码片段压缩成一句话，大概就是：

- `checkRuleBasedPermissions()` 先看“规则层面是否反对”。
- `tool.checkPermissions()` 再看“这个具体工具、这次具体输入有没有问题”。
- `resolveHookPermissionDecision()` 再确保“外部 Hook 的结论不能冲掉内建规则”。
- 真要跑 Bash 时，再决定是否进入沙箱。

20.4 纵深防御的关键，不是层数多，而是层与层独立

很多系统也喜欢说“我们有很多安全措施”，但其实几层都在做同一件事，失效时会一起失效。Claude Code 这套设计更有价值的地方，是它让不同层解决不同问题：

层	主要回答的问题	典型文件
模式层	这次会话总体有多激进?	`PermissionMode.ts`
规则层	这个工具/模式在当前上下文下该不该做?	`permissions.ts`
工具层	这个具体命令或路径安全吗?	`bashPermissions.ts`
Hook 层	组织或扩展逻辑要不要追加治理?	`toolHooks.ts`
分类层	自动模式下是否需要再判断一次?	`yoloClassifier.ts`
沙箱层	就算判断错了，系统还能拦什么?	`sandbox-adapter.ts`
策略层	哪些配置和身份边界根本不让你碰?	`settings.ts` / `oauth`

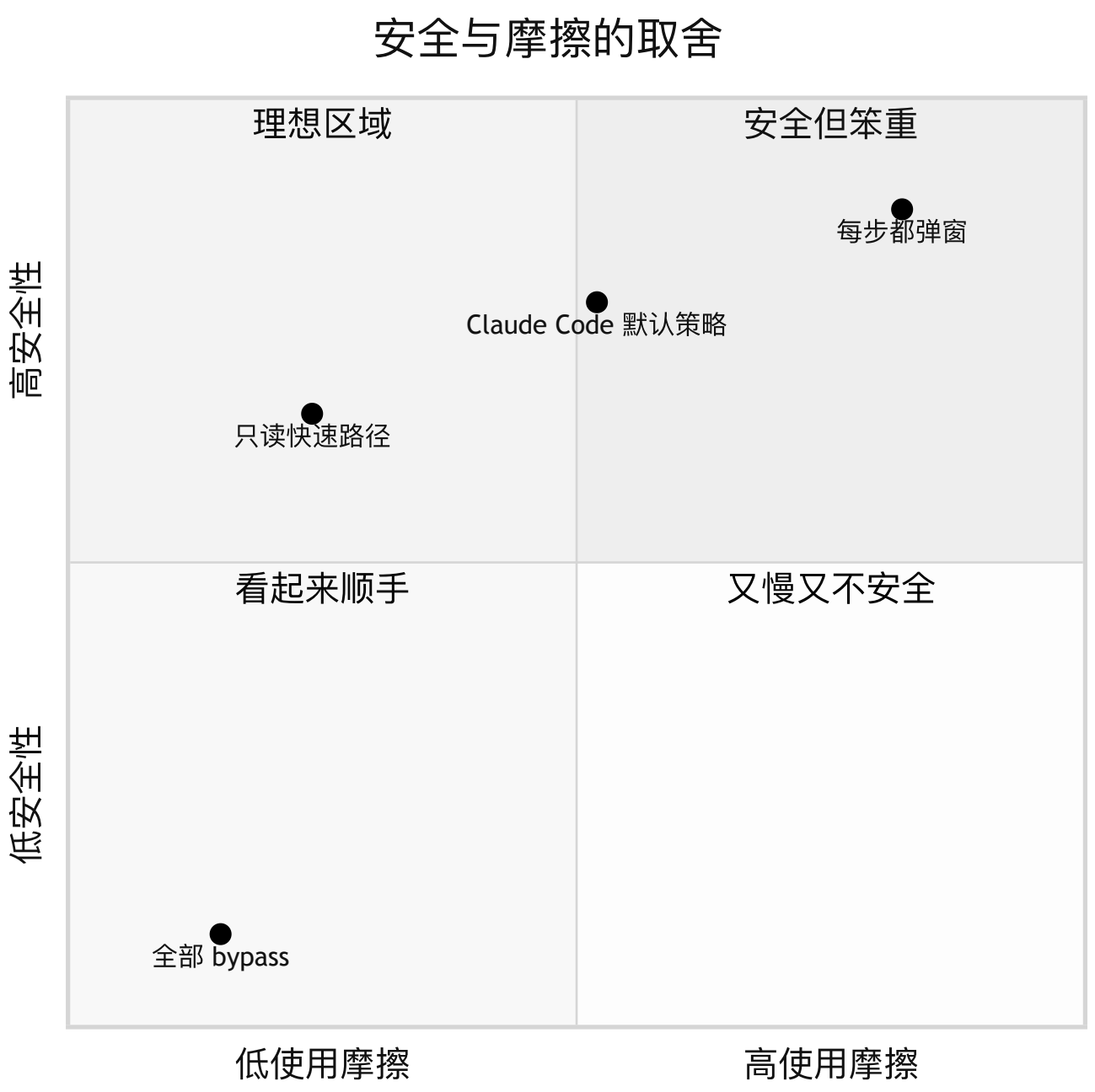


这也是为什么 Claude Code 的安全代码看起来“不优雅”地分散在很多目录里。它不是偷懒，而是在避免“一个总开关失效，全盘失守”。

20.5 为什么它不追求零弹窗，也不追求零风险

安全产品最难的不是“更严”，而是“既严又能用”。源码里能看到两个平衡动作：

- 一方面，它给只读命令、安全工具、沙箱内 Bash 留了快速路径。
- 另一方面，它对危险规则、危险路径、危险 sed、危险 shell 特性采取了绕不过去的强约束。



深入一点看，你会发现它并不是把“快”和“安全”当成对立面，而是努力把**低风险操作快速化**，把**高风险操作精细化**。这是比“永远先问用户”更成熟的产品观。

深水区（架构师选读）

这一章最值得带走的思想是：安全系统最好不要只有一个“聪明的大脑”。Claude Code 把治理拆成模式、规则、工具、Hook、分类器、沙箱、策略这几类相对独立的部件。这样就算其中一块判断失误，其他块还有机会补救。对任何带执行能力的 AI 系统来说，这种“可失败但不易失控”的设计，比追求某个单点模型判断 100% 正确更现实。

本章小结

Claude Code 的安全性不是来自“模型不会犯错”，而是来自一条多层、独立、互相补位的防线。你后面读权限系统、BashTool、沙箱、认证时，都可以把它们放回这七层框架里理解。

关键源码索引

- PermissionMode 模式定义：PermissionMode.ts
- 规则判定主流程：permissions.ts
- 最终权限决策：permissions.ts
- 自动模式危险权限剥离：permissionSetup.ts
- 危险权限扫描：permissionSetup.ts
- Bash 二次检查顺序：bashPermissions.ts
- Hook 权限合流：toolHooks.ts
- 拒绝追踪回退：denialTracking.ts

- 沙箱配置翻译：`sandbox-adapter.ts`

逆向提醒

“七层防御”是本书根据多处源码整理出的分析框架，不是源码里某个现成常量名。真实实现是分散的：有的在权限引擎里，有的在工具内部，有的在沙箱和认证层。

28

权限系统 第五编

第21章：权限系统：允许、拒绝，还是先问一句

生活类比：手机权限弹窗

手机上的相机、麦克风、定位，不是都按同一规则处理。手电筒不需要定位，导航又必须拿位置，转账 App 还会对录屏做额外限制。Claude Code 的权限系统也是这样：它不是简单地问“要不要执行”，而是问“**这件事在当前上下文下该怎么执行**”。

这一章先回答一个问题

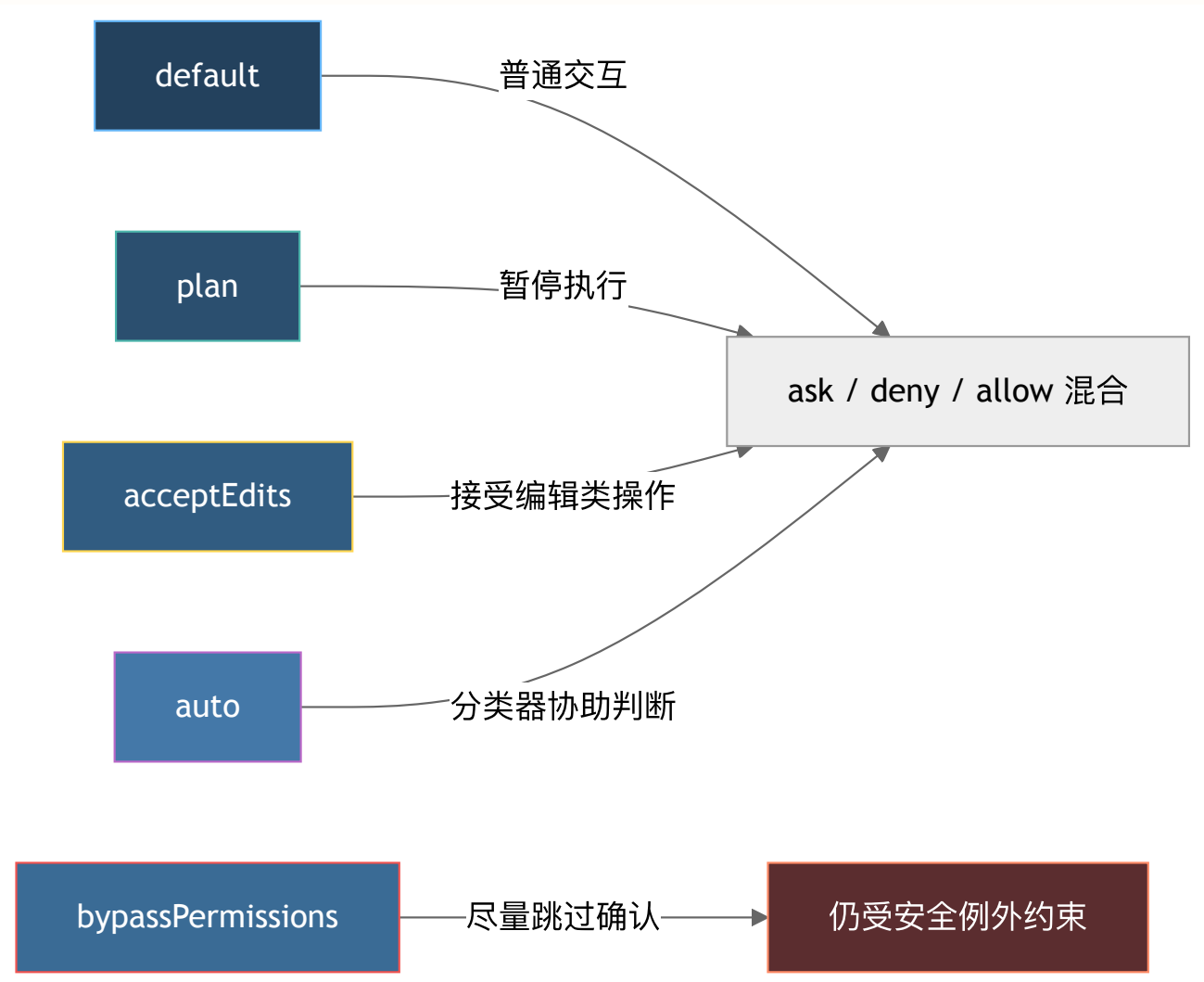
为什么有些操作会直接执行，有些会弹确认框，有些则会直接被拦下？

答案藏在三组代码里：

- `PermissionMode.ts` 负责定义“会话的大气候”；
- `permissionSetup.ts` 负责把 CLI、settings、组织策略合并成当前模式；
- `permissions.ts` 负责把某次具体工具调用判定成 `allow`、`ask`、`deny`。

21.1 权限不是一个开关，而是一套模式

`PermissionMode.ts` 把模式做成了明确的配置对象，每种模式都有 `title`、`symbol`、`color` 和对外映射。也就是说，模式不是隐藏在 if/else 里的字符串，而是一个完整的“产品状态”。



更关键的是，`permissionSetup.ts` 不会直接相信某一个来源。它会按优先级合并：

- 命令行参数

- `--dangerously-skip-permissions`
- `settings.permissions.defaultMode`
- GrowthBook/Statsig 门控
- 远程环境限制

这意味着用户“想进入某模式”，不等于系统“允许进入某模式”。

为什么 `bypassPermissions` 也不是绝对的

`permissionSetup.ts` 明确检查：

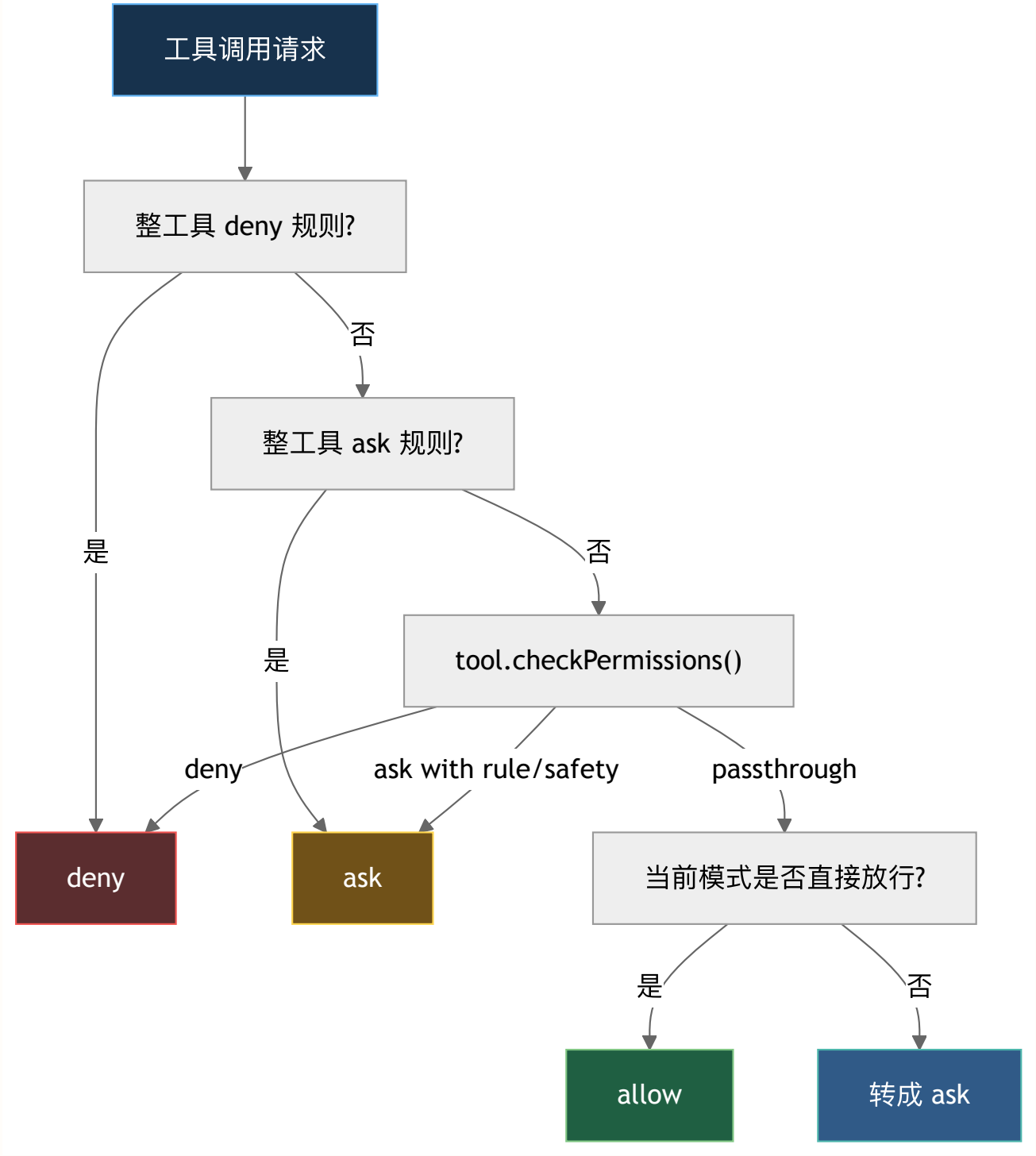
- 组织门控是否禁用 bypass；
- settings 是否禁用 bypass；
- 远程环境是否支持当前默认模式；
- auto 模式是否被 circuit breaker 关停。

这就是成熟系统和 demo 工具的差别：它允许“有力模式”存在，但不允许它无限制扩散。

21.2 真正的判定发生在 `permissions.ts`

权限引擎的核心逻辑可以压成一句话：

1. 先看整工具规则；
2. 再看工具自己的 `checkPermissions()`；
3. 再看当前模式能不能直接放行；
4. 最后把没有明确结论的操作转成 `ask`。



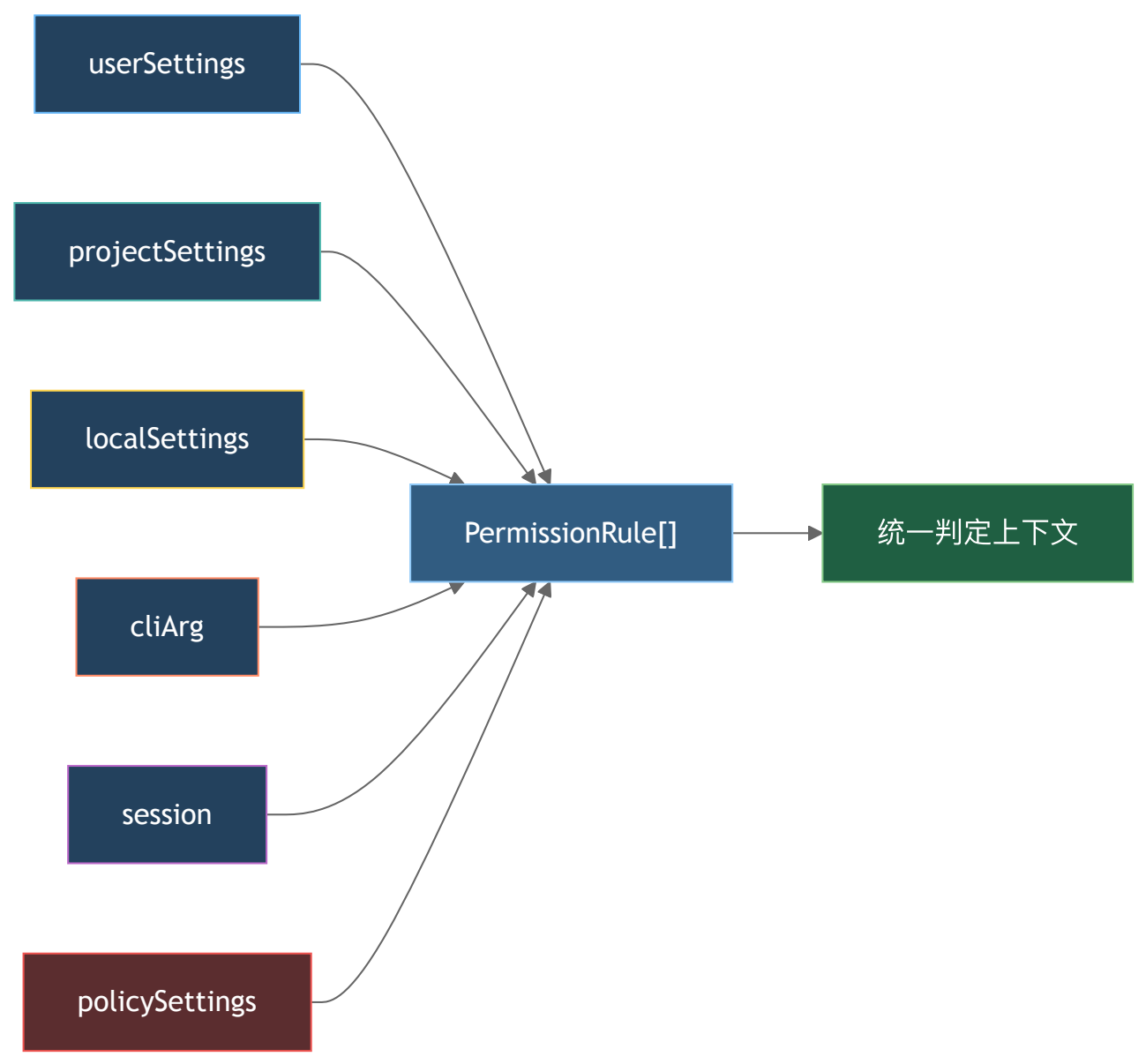
`checkRuleBasedPermissions()` 是早期过滤器，`hasPermissionsToUseToolInner()` 则是完整决策器。它们有几个非常重要的细节：

- **deny 优先于一切**；
- **content-specific ask** 不能被 bypass 模式冲掉；
- **safetyCheck** 这类路径安全检查也是 bypass-immune；
- 没被明确允许的 `passthrough` 最终会被收束成 `ask`。

也就是说，Claude Code 不是“默认都放行，特殊情况才拦”；它更像“默认先怀疑，再找理由自动放行”。

21.3 规则从哪里来：不是一层，是多源合并

权限规则的来源不止一个。`permissionsLoader.ts` 会从多个 settings source 里加载规则，再把它们转成统一的 `PermissionRule` 结构。



其中最值得注意的是 `allowManagedPermissionRulesOnly`：

- 一旦它在 `policySettings` 中开启；
- 非托管来源的权限规则会被忽略；
- “永远允许”这类选项也会在界面上隐藏。

这就是企业治理味道很重的一段代码：管理员不是“多给一个建议”，而是能**改变规则系统本身的边界**。

为什么要有 `localSettings`

从写书角度最容易忽略的一点是：`localSettings` 和 `projectSettings` 不是一回事。

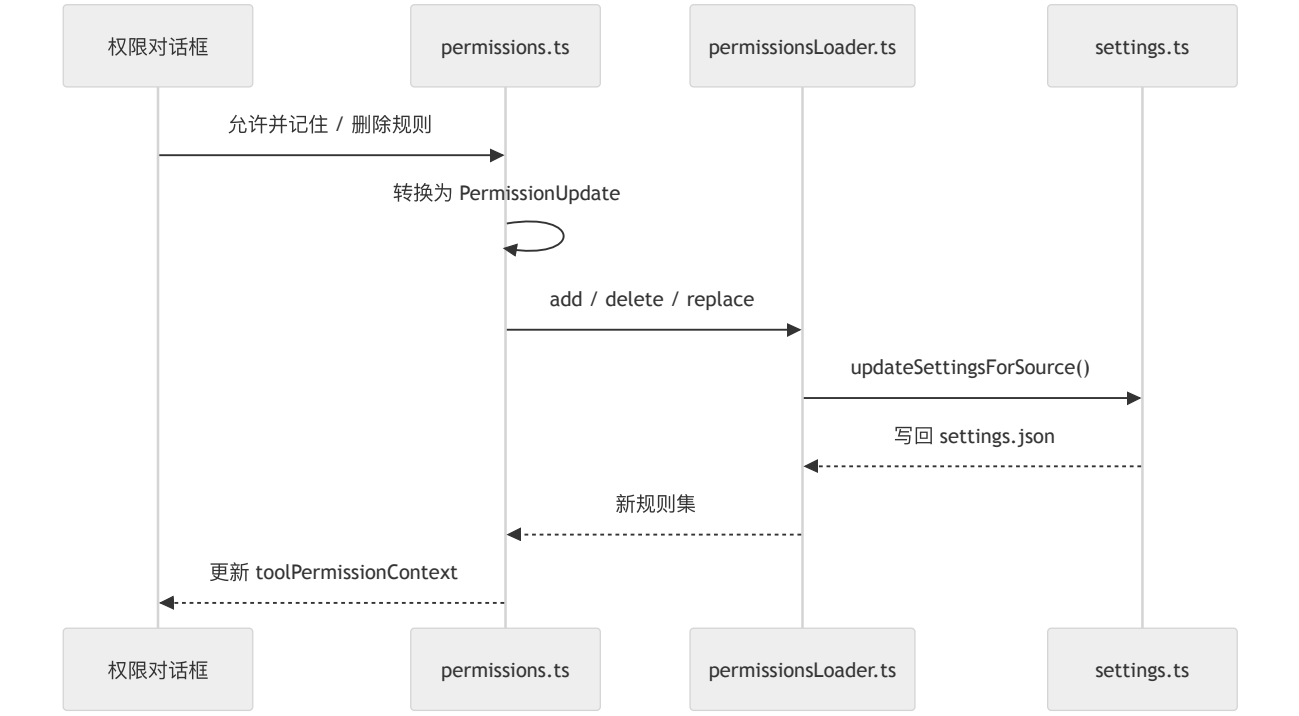
- `projectSettings` 更像项目共享规则；
- `localSettings` 更像当前开发者的本地偏好；
- `userSettings` 则跨项目生效。

这种分层不是形式主义，而是在回答一个现实问题：什么规则是“我个人现在想这么做”，什么规则是“这个仓库里的所有人都该这么做”。

21.4 权限系统还要解决“改规则”这件事

如果规则只能读不能写，这套系统只适合管理员，不适合真实开发者工作流。所以 `permissions.ts` 和 `permissionsLoader.ts` 还处理了：

- 删除规则；
- 把规则按 source + behavior 分组；
- 同步磁盘变化；
- 在受托管限制时清空非托管来源。



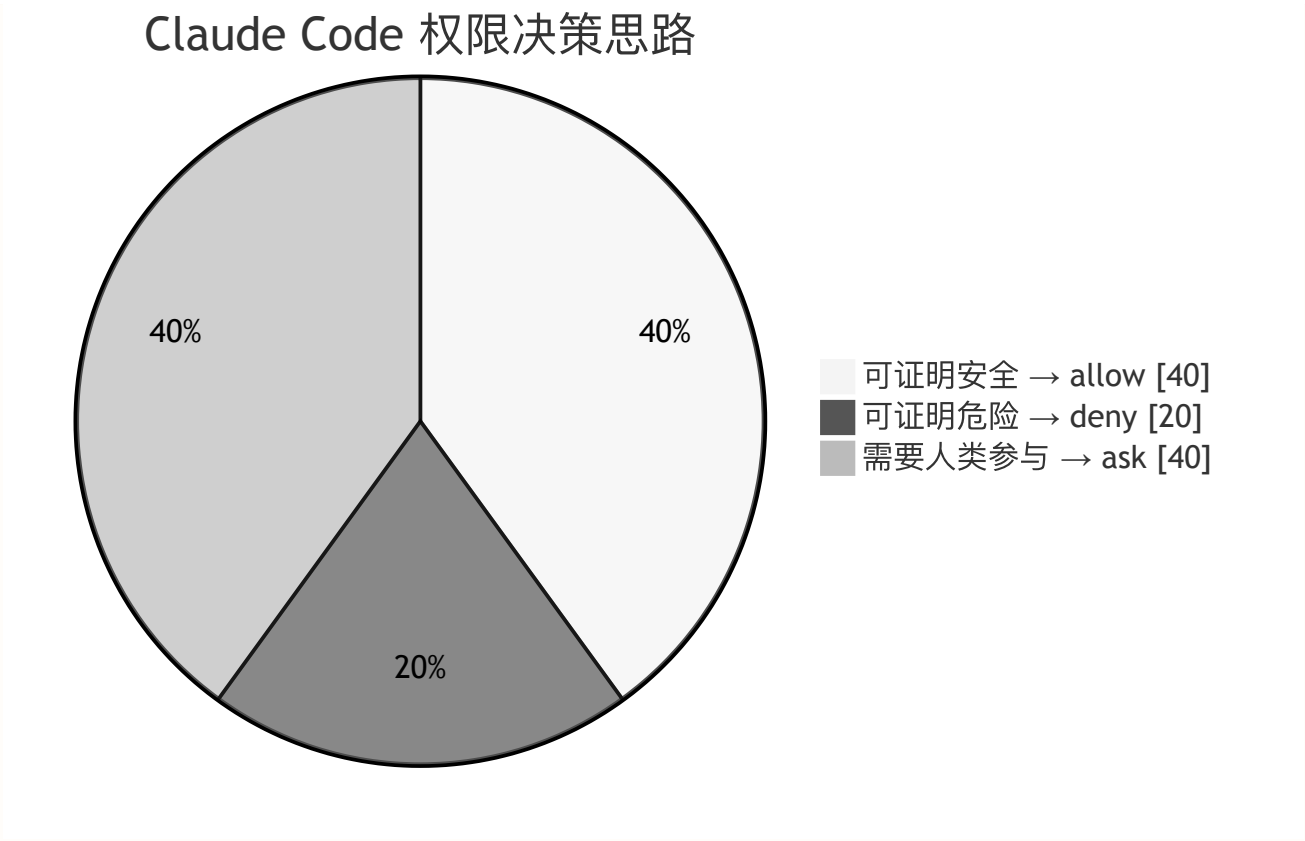
这里有一个很容易被忽视但非常工程化的设计：`syncPermissionRulesFromDisk()` 在重载时会**先清空磁盘来源的 source:behavior 组合，再应用新规则**。这样删除掉的旧规则不会因为“这次没出现”而偷偷残留在内存里。

这类细节恰恰说明这不是一个“能跑就行”的权限系统，而是考虑过长期使用的状态一致性。

21.5 为什么它能同时做到“少打扰”和“少失控”

权限系统不是为了制造弹窗，而是为了降低误判成本。你可以把 Claude Code 的思路理解成三句话：

- 能明确证明安全的，自动过；
- 能明确证明危险的，直接拦；
- 其余不猜，问你。



这也是为什么它会同时保留：

- 显式规则；
- 工具内部检查；
- 模式控制；
- 组织级托管策略。

没有哪一种机制能独立解决全部问题，但合起来就形成了“可理解、可追踪、可治理”的权限体系。

🔭 深水区（架构师选读）

这套权限系统最值得学习的地方，不是 `allow / ask / deny` 这三个词本身，而是它把“会话状态”“规则来源”“工具自检”“组织治理”这几件事拆开了。很多产品把权限只做成一个弹窗组件，结果所有复杂性都挤进 UI；Claude Code 则把 UI 变成最后呈现层，真正的语义判断发生在底下的上下文和规则引擎里。

本章小结

Claude Code 的权限系统不是一套“是否弹窗”的小功能，而是一台把模式、规则、来源、状态同步到一起的判定引擎。你看到的每一个“直接执行 / 先问一句 / 直接拒绝”，背后都能追到具体源码。

关键源码索引

- 模式定义：`PermissionMode.ts`
- 初始模式选择：`permissionSetup.ts`
- auto/bypass 门控：`permissionSetup.ts`
- 规则读取与 managed-only 策略：`permissionsLoader.ts`
- 规则判定入口：`permissions.ts`
- 最终权限决策：`permissions.ts`
- 规则删除与磁盘同步：`permissions.ts`
- 从磁盘同步规则：`permissions.ts`

逆向提醒

你在界面里看到的权限弹窗，只是这一章的“表层”。真正重要的是 `toolPermissionContext` 的构造、规则来源的优先级，以及 bypass/auto 这类模式被哪些安全例外重新收紧。

29

BashTool 第五编

第22章：25道安全关卡：BashTool 的层层安检

生活类比：银行金库的多道门禁

银行金库不会只靠一把锁。门口有人脸识别，里层有密码门，再里面有时间锁、录像、值班员复核。BashTool 也一样，因为它几乎是 Claude Code 最强、也最危险的工具。

这一章先回答一个问题

Claude Code 在执行一条 shell 命令前，到底检查了什么，为什么要检查这么多次？

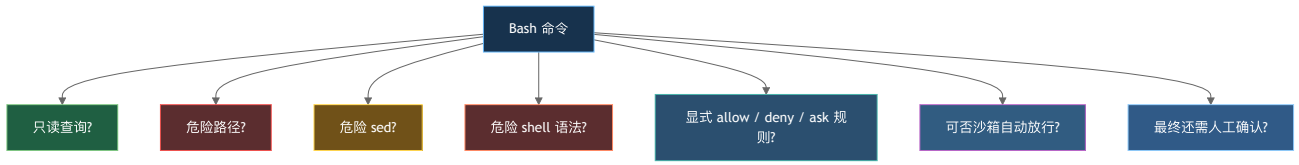
如果说前一章讲的是“谁有资格过门”，这一章讲的就是“**进门之后还要过哪些闸机**”。尤其是 BashTool，它不仅能读文件、搜代码、跑测试，还能删目录、改配置、联网、开子进程。所以源码里围着它长出了一大片独立的安全代码。

22.1 为什么 BashTool 是最难做安全的工具

因为它太通用了。对模型来说，Bash 像一把万能钥匙：

- 读文件可以靠 `cat`、`sed`、`awk`
- 搜索可以靠 `grep`、`find`、`rg`
- 构建测试可以靠 `npm test`、`pytest`
- 破坏系统也可以靠 `rm -rf`、`curl | bash`、危险 `git`

所以 Claude Code 对 Bash 的态度不是“默认危险，全部拦下”，也不是“既然强大就相信模型”。它选择第三条路：把 Bash 命令拆成不同风险层级，逐层检查。



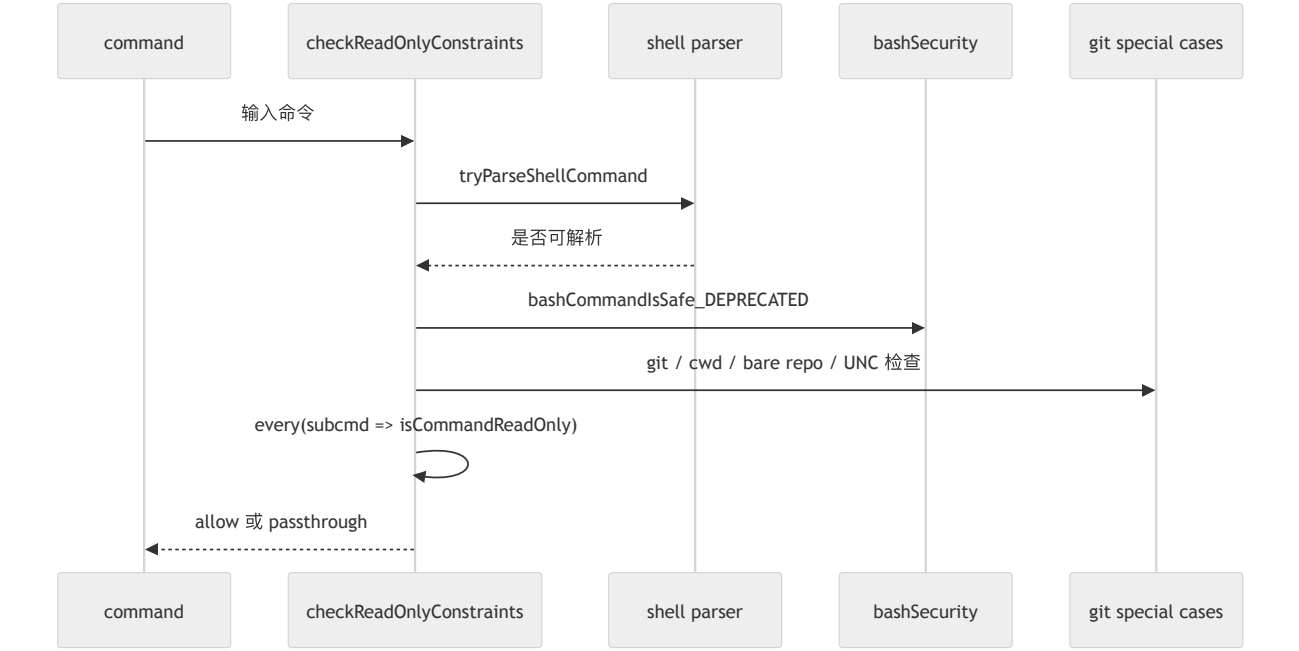
22.2 第一批闸机：先把显然安全的命令快速放行

`readOnlyValidation.ts` 的设计思路很有代表性：它不试图证明“所有命令都安全吗”，而是只证明“这批命令足够像只读操作”。

只读快速路径

`checkReadOnlyConstraints()` 会做这些事：

- 命令能不能可靠解析；
- 原始命令有没有明显危险模式；
- 是否含有可能触发 WebDAV 的 UNC 路径；
- 是否出现 `cd + git` 这类可组合成逃逸的模式；
- 是否在非原始工作目录里跑 git；
- 所有子命令是否都能判成只读。



特别值得注意的是几段“很像漏洞复盘”的注释：

- `cd /malicious/dir && git status`
- 在当前目录伪造 bare repo 结构
- 先写 git internal files 再执行 git

这些不是学术化威胁模型，而是把历史上踩过的坑直接写进代码。

为什么 `xargs` 也要单独看

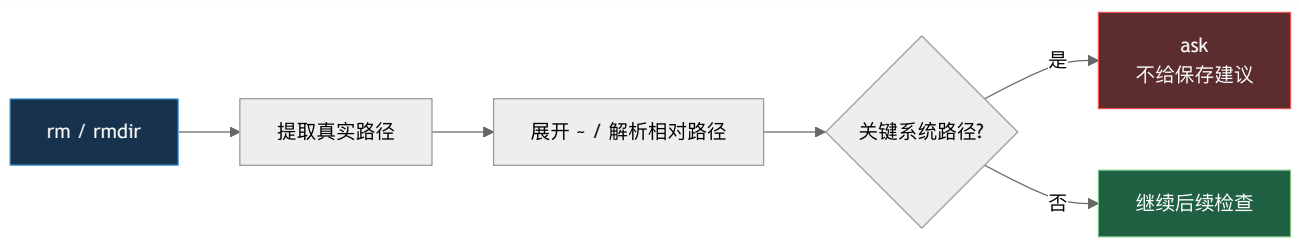
`readOnlyValidation.ts` 甚至连 `xargs` 后面允许跟什么命令都做了白名单，只放 `echo`、`printf`、`wc`、`grep`、`head`、`tail` 这种纯读或纯输出工具。意思很明确：它不是按“命令名大致安全”来判，而是按“组合后还能不能继续保持只读语义”来判。

22.3 第二批闸机：危险路径和危险编辑要单独拦

不是所有危险都长得像 `rm -rf /`。有些危险是“命令长得普通，但目标路径很可怕”。

危险移除路径

`pathValidation.ts` 明确规定：`rm` / `rmdir` 命中关键系统目录时，必须要求显式批准，而且**不能被普通 allow 规则自动放行**。



这段代码的书写方式很有意思：它特意说明**不用解析符号链接后的真实路径**，因为像 `/tmp` 这种在 macOS 上虽然会跳到 `/private/tmp`，但用户输入层面仍应被视为危险目标。

危险 sed

`sedValidation.ts` 也不是一刀切禁止 `sed`。它区分：

- 普通替换；
- `-i` 原地编辑；
- 带 `w/W/e/E` 这类可能写文件或执行命令的危险表达式。

而且在 `acceptEdits` 模式下，它允许普通文件写入，但仍然会对危险 `sed` 操作保留 `ask`。

这套设计非常像一句产品原则：

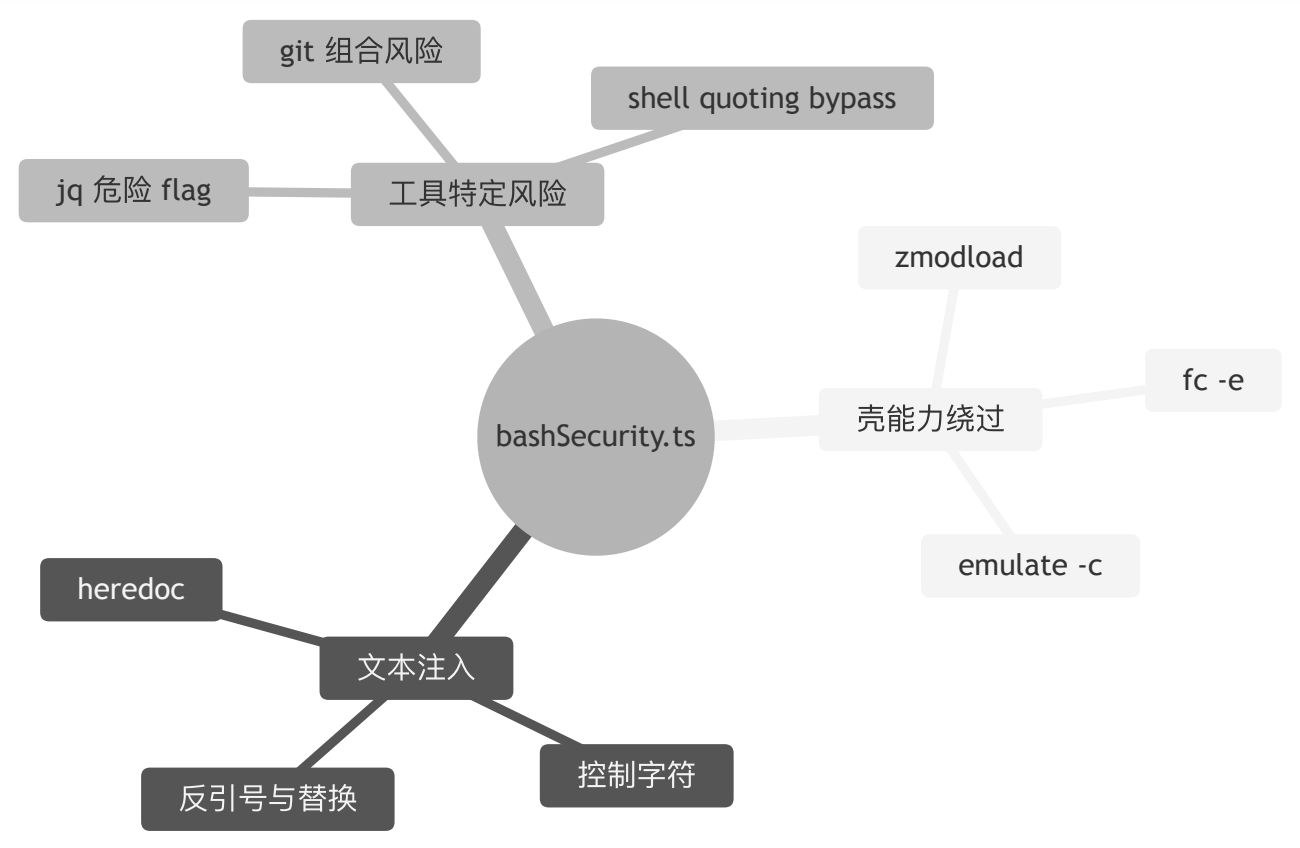
> 允许“编辑”，不等于允许“顺便执行别的东西”。

22.4 第三批闸机：壳层语法本身也会变成攻击面

再往下，你会看到 `bashSecurity.ts` 这种更偏“语法安全”的检查。

它关心的不是业务语义，而是 shell 自身的绕过方式，比如：

- Zsh 的 `zmodload`
- `fc -e` 这种借编辑器执行命令的形式
- 某些危险变量上下文
- 某些 heredoc、转义、控制字符、jq 危险 flag



这一层很像“语法取证”。

它提醒我们：同样是 `grep`、`sed`、`git` 这些熟悉命令，只要被放进不同的 shell 语境里，安全含义就会立刻变化。所以 BashTool 的安全不是“列黑名单”那么简单，而是要同时懂：

- 命令名
- 参数
- 组合方式
- shell 解释规则

22.5 最后的合流：所有检查不是并列堆着，而是有顺序的

`bashPermissions.ts` 最值得细读的地方，不只是它做了很多检查，而是它把检查排出了顺序。

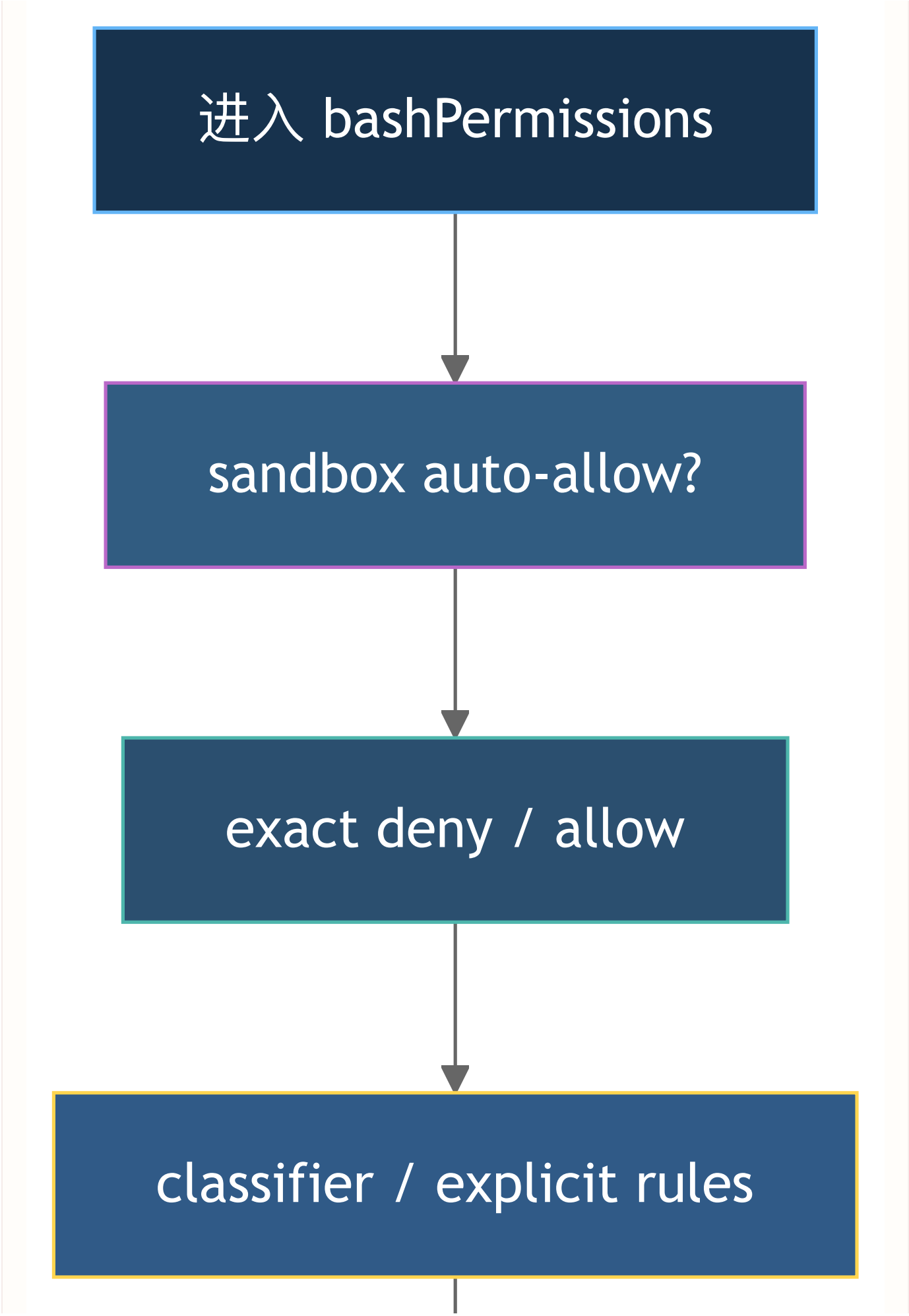
从代码上看，靠前的通常是：

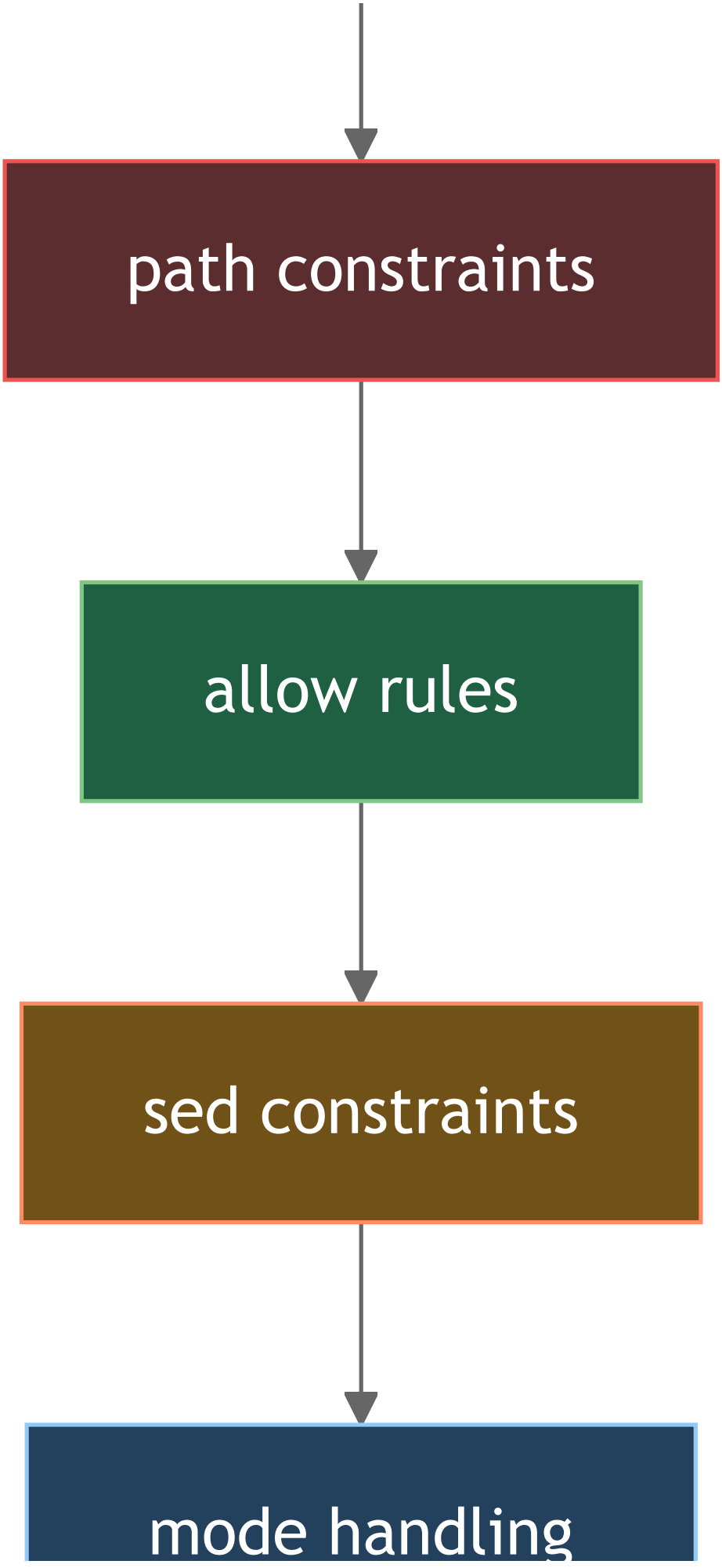
- 明确 deny/ask
- 路径限制
- allow 规则
- sed 约束
- 模式判断
- 只读快速路径

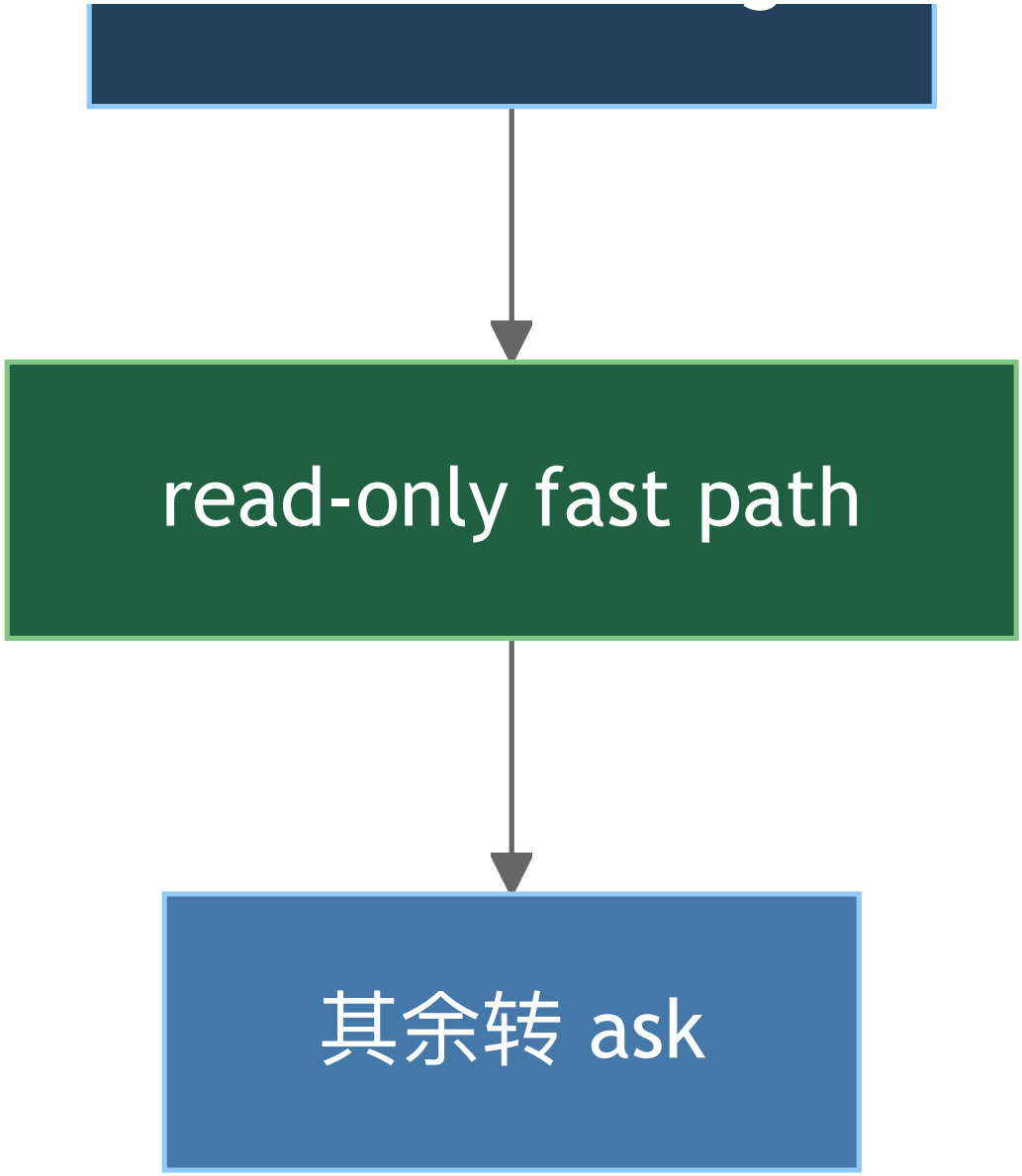
- 最终转成确认请求

而在更上面的调用入口里，还会插入：

- 沙箱自动放行
- 分类器权限
- exact match


进入 bashPermissions
sandbox auto-allow?
exact deny / allow
classifier / explicit rules


path constraints
allow rules
sed constraints
mode handling



顺序为什么重要?

- 因为早一点发现明确 deny，就不用继续浪费算力；
- 因为危险路径要早于普通 allow，不然规则会把它吞掉；
- 因为只读快速路径要晚于语法安全检查，不然会误判；
- 因为模式放行也不能覆盖某些特殊安全例外。

这就是“25 道安全关卡”的真正含义：不只是数量多，而是每一道都在守一个不同的漏洞口。

深水区（架构师选读）

BashTool 这一章最像“安全工程现场”。你会发现 Claude Code 并没有追求一种完美的 shell 安全证明，而是用大量保守、可解释、可叠加的约束来缩小风险面。这和现实里的安全系统很像：与其幻想一次判断永远正确，不如让不同阶段都能抓住不同类型的坏输入。

本章小结

BashTool 的安全不是一条黑名单，而是一串按顺序运行的关卡：只读识别、路径校验、危险 sed、shell 语法安全、规则匹配、模式判断、沙箱配合。正因为它最强，所以它的安检也最重。

关键源码索引

- 只读约束主入口：`readOnlyValidation.ts`
- `xargs` 安全目标白名单：`readOnlyValidation.ts`
- 危险移除路径检查：`pathValidation.ts`
- 危险路径插入顺序：`pathValidation.ts`
- `sed` 约束入口：`sedValidation.ts`
- Zsh 危险命令检查：`bashSecurity.ts`
- Bash 权限后半段顺序：`bashPermissions.ts`
- Bash 权限前半段入口：`bashPermissions.ts`

逆向提醒

“25 道安全关卡”是对 BashTool 安全检查簇的概括，不是源码里排成 1 到 25 的官方清单。真正实现横跨 `bashPermissions.ts`、`readOnlyValidation.ts`、`pathValidation.ts`、`sedValidation.ts`、`bashSecurity.ts` 多个文件。

30

Hook 沙箱 第五编

第23章：沙箱与拦截：应用层之外的第二道墙

生活类比：化学实验室的安全柜

实验员再小心，也可能手抖；流程再完善，也可能有人判断失误。所以危险实验要放在安全柜里做。就算操作出错，泄漏也被控制在柜体内部。Claude Code 的沙箱就是这只“柜子”，Hook 则像柜门口的附加检查员。

这一章先回答一个问题

如果上层权限判断失手了，Claude Code 还有什么办法把危险限制在小范围内？

这一章要讲两个互补系统：

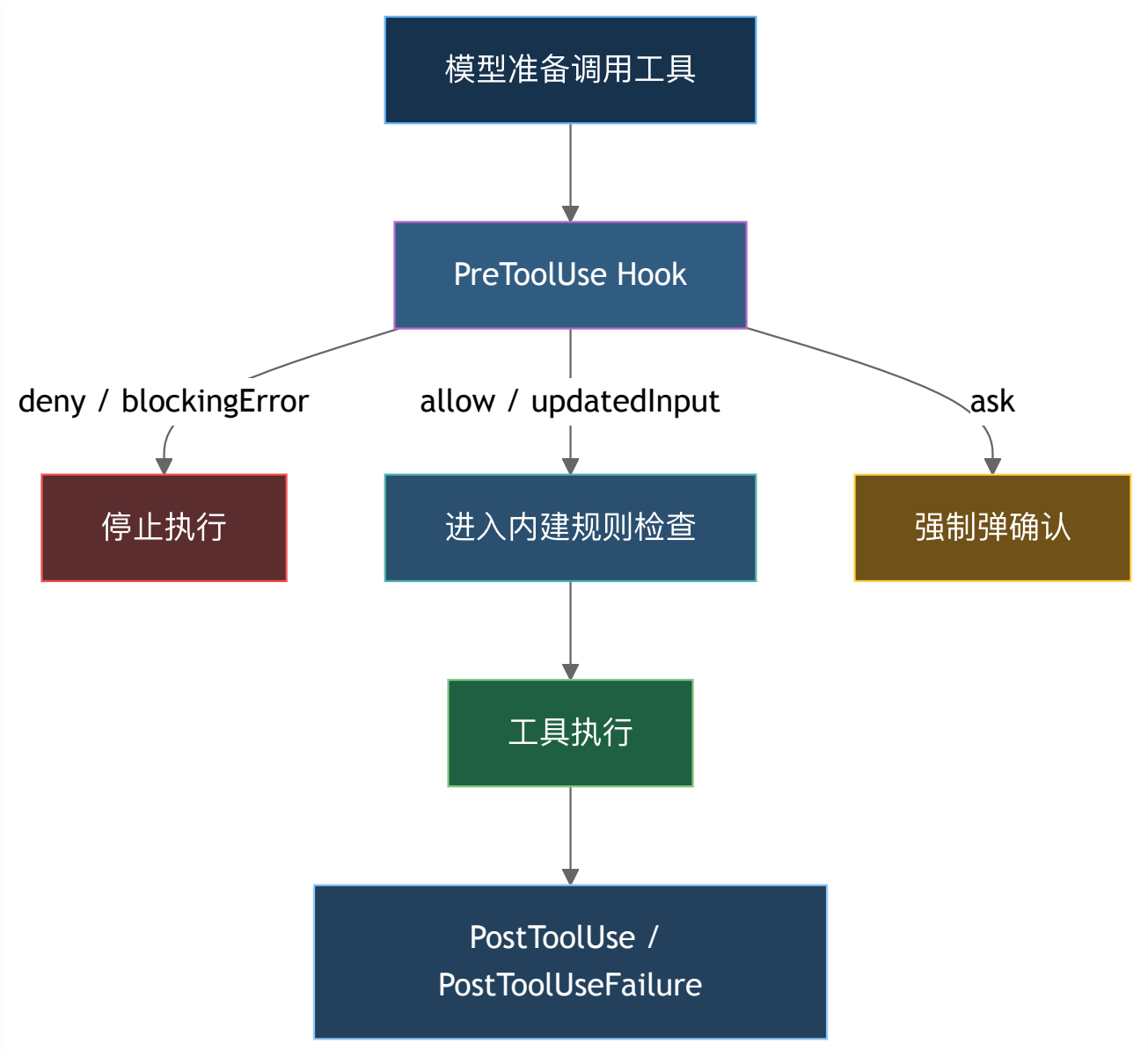
- **Hook**：在应用层补充组织策略与自定义治理；
- **Sandbox**：在操作系统层提供最后的隔离边界。

23.1 Hook 不是“插件彩蛋”，而是安全闸门

`toolHooks.ts` 里最值得注意的一句注释是：**Hook 的 allow 不能绕过 settings.json 的 deny/ask 规则**。这几乎把它的定位讲透了。

Hook 在这里不是“想怎么改就怎么改”的扩展机制，而是：

- 能补充策略；
- 能修改输入；
- 能阻止继续；
- 但不能推翻更基础的安全边界。



PreToolUse 到底能做什么

从 `runPreToolUseHooks()` 和 `resolveHookPermissionDecision()` 的逻辑看，PreToolUse Hook 主要有四种影响方式：

- 返回阻断错误，直接拒绝工具执行；
- 返回 `allow` 并可附带 `updatedInput`；
- 返回 `ask`，强制把这次操作抬升为人工确认；
- 不给权限结论，只修改输入，让后续正常权限流继续。

这种设计很聪明：Hook 有治理力，但不会把主系统变成一个“谁先写 Hook 谁说了算”的黑盒。

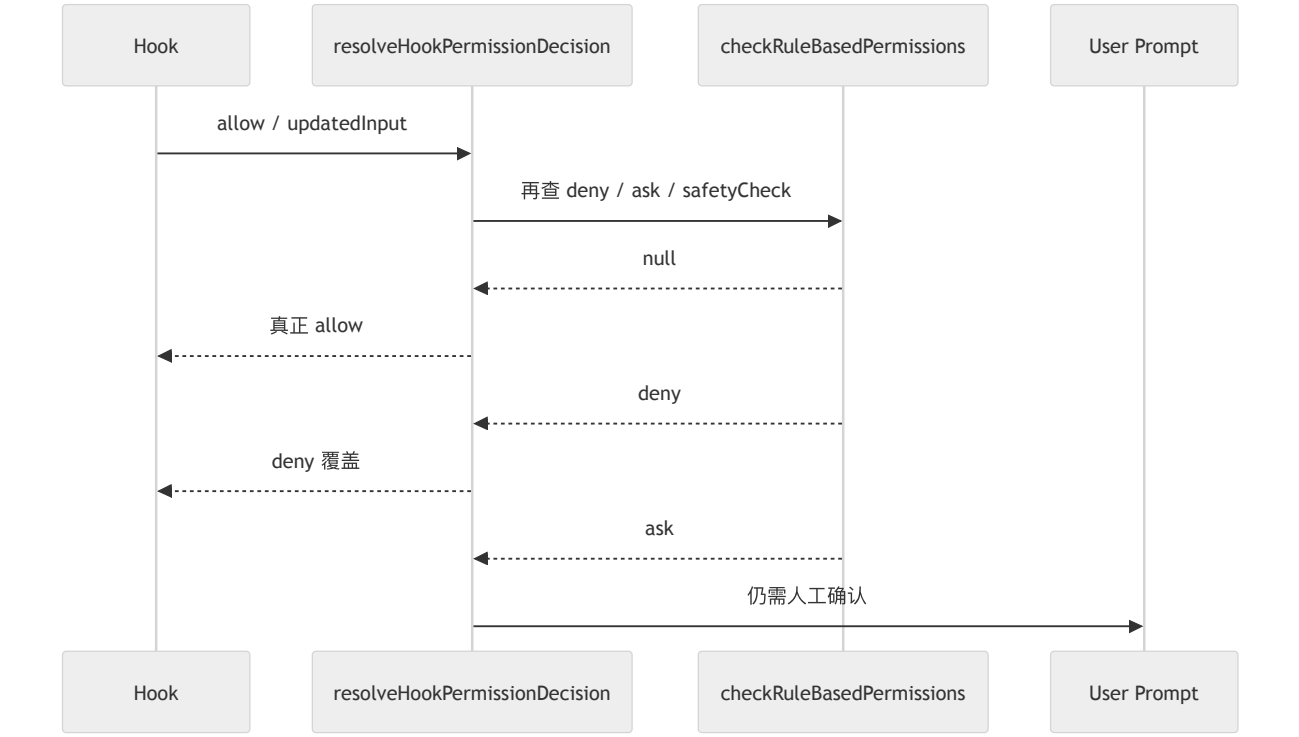
23.2 Hook 为什么要服从内建规则

`resolveHookPermissionDecision()` 的实现很有代表性。它专门处理一种很危险的情况：

> Hook 说 allow，但系统规则其实说 deny 或 ask。

源码的选择是：

- 如果 Hook allow，但规则层没有异议，才真的 allow；
- 如果规则层 deny，Hook allow 会被覆盖；
- 如果规则层 ask，仍然必须弹确认框。



这段逻辑非常值得读者记住，因为它体现了 Claude Code 的治理哲学：

- Hook 是“外加策略”；
- 内建规则是“底座边界”；
- 外加策略可以收紧，也可以补充，但不能把底座掀掉。

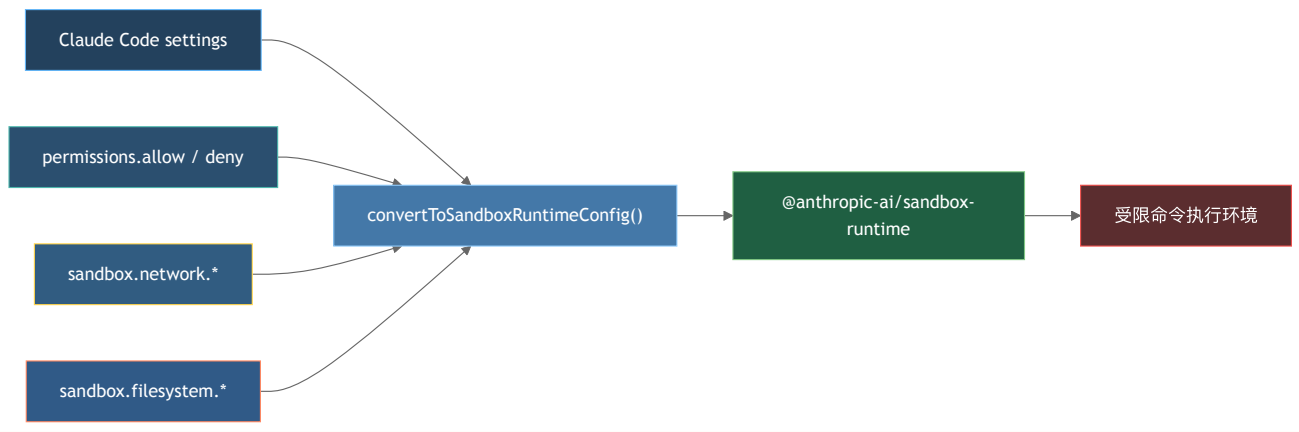
这和很多企业系统里的“审批流不能覆盖安全基线”是同一思路。

23.3 沙箱做的是另一种事：不是判断，而是隔离

Hook 负责判断，沙箱负责隔离。这两者不要混在一起。

`sandbox-adapter.ts` 做的核心工作，是把 Claude Code 内部的 settings / permission 语义，翻译成底层 sandbox runtime 能理解的配置：

- 网络允许哪些域名；
- 哪些目录可读、可写；
- 哪些 settings 文件必须拒写；
- 哪些 `.claude/skills` 路径要保护；
- 当前工作目录与原始工作目录有什么差异。



沙箱里最有味道的几条规则

从 `convertToSandboxRuntimeConfig()` 可以直接看到一些非常“实战派”的防护：

- 默认把当前目录和 Claude temp 目录加入可写；
- 始终拒写 settings 文件；
- 如果 cwd 变化了，还要补保护新的 `.claude/settings*.json`；
- 额外拒写 `.claude/skills`，因为技能和 commands/agents 一样会带来高权限能力；
- 对 git 内部结构进行专门保护，防止借 bare repo 结构逃逸。

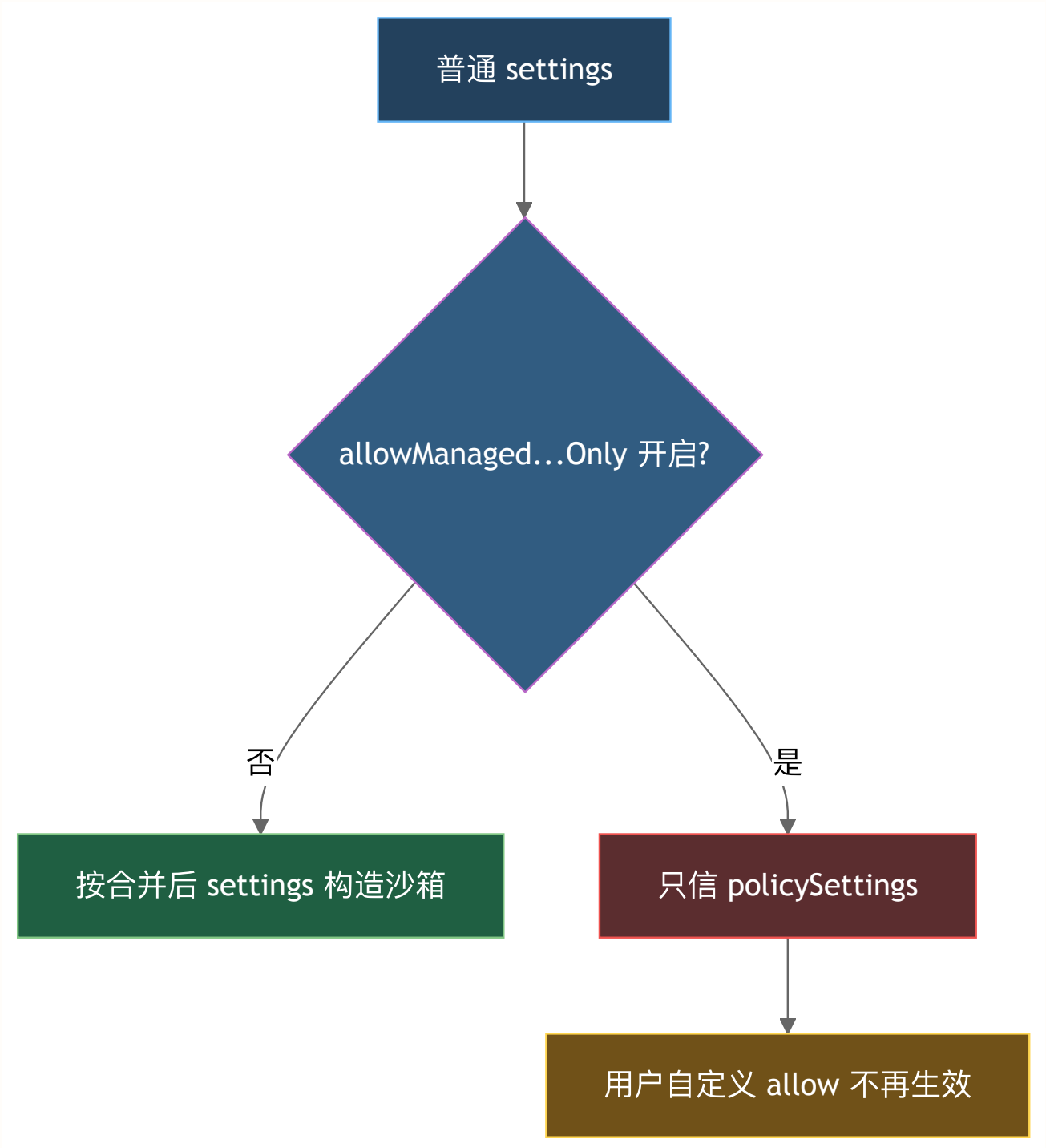
这说明 Claude Code 的沙箱不是“给所有进程套个笼头”这么粗糙，而是很明确地围绕**自己系统里最危险的持久化点**来加固。

23.4 托管策略还能进一步收紧沙箱边界

`sandbox-adapter.ts` 里还有两条很重要的“组织收紧”逻辑：

- `allowManagedDomainsOnly`
- `allowManagedReadPathsOnly`

这类名字的含义很直白：当组织级 policy 打开后，允许访问哪些域名、哪些路径，不再由普通用户 settings 决定，而只看 managed settings。



这就是“应用层权限”与“组织级安全基线”真正接上的地方。不是只在 UI 里写一句“你所在组织禁用了此功能”，而是直接改变底层沙箱的构造方式。

23.5 两堵墙一起工作，才是可靠系统

如果只用 Hook，会有两个问题：

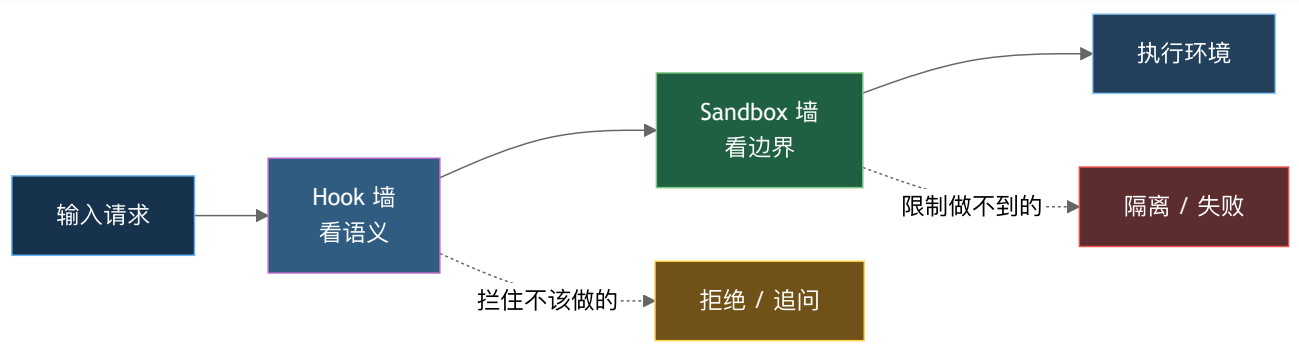
- Hook 可能写错；
- Hook 只在应用层，进程一旦真跑起来，还是靠环境约束。

如果只用沙箱，也有两个问题：

- 它不知道业务语义；
- 很多“该不该问用户”这种体验级判断不适合下沉到 OS。

所以最稳的结构，是两堵墙配合：

机制	擅长什么	不擅长什么
Hook	业务语义、组织策略、额外上下文	物理隔离
Sandbox	系统级隔离、目录/网络限制	高层语义判断



对初学者来说，可以把它记成一句最简单的话：

> Hook 负责“这件事该不该做”，沙箱负责“就算做了，也别跑出圈”。

深水区（架构师选读）

这一章真正高级的地方，在于“Hook 不是万能管理员，Sandbox 也不是万能警察”。Claude Code 没有让两者互相替代，而是让它们互相补位：Hook 以语义为中心，Sandbox 以边界为中心。前者更像策略引擎，后者更像内核护栏。对任何带执行能力的 AI 平台，这都是值得直接借鉴的分层。

本章小结

Claude Code 的第二道墙不是某一个文件，而是一种组合：Hook 负责应用层拦截，Sandbox 负责系统层隔离。两者共同确保“判断失误”不容易升级成“环境失控”。

关键源码索引

- PreToolUse / PostToolUse 总线：`toolHooks.ts`
- Hook 与规则合流：`toolHooks.ts`
- PreToolUse 结果处理：`toolHooks.ts`
- 内部 post-sampling hooks：`postSamplingHooks.ts`
- 沙箱适配层入口：`sandbox-adapter.ts`
- 路径规则翻译：`sandbox-adapter.ts`
- 托管域名/路径限制：`sandbox-adapter.ts`
- 沙箱配置转换：`sandbox-adapter.ts`

逆向提醒

OpenClaudeCode 这里调用的是外部包 `@anthropic-ai/sandbox-runtime`。我们能清楚看到 Claude Code 如何构造和使用沙箱配置，但更底层的运行时细节在这个仓库之外。

31

第五编 认证 配置治理

第24章：配置与认证：谁能用，怎么管

生活类比：公司的门禁系统

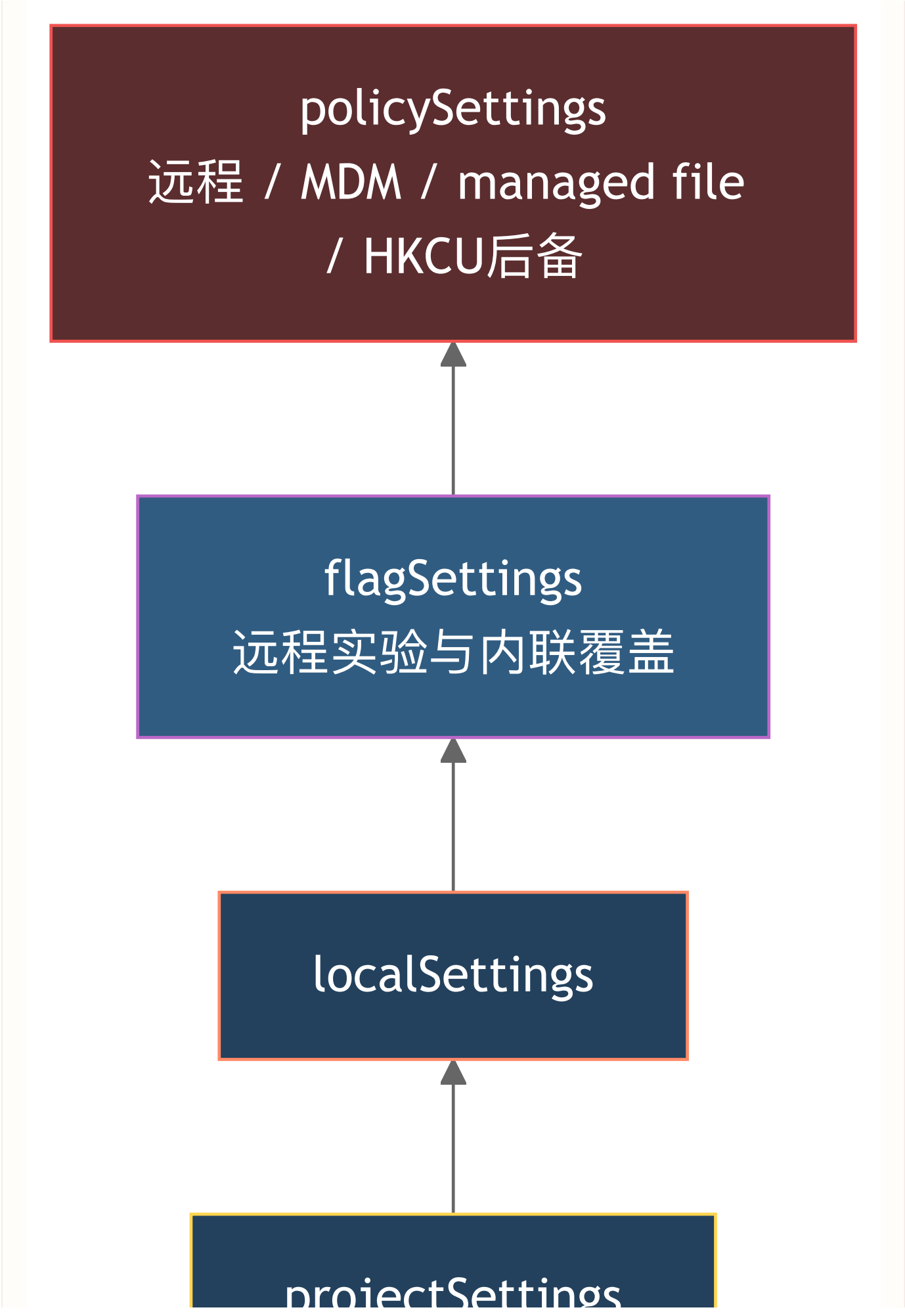
一家公司不会只靠一把钥匙运转。员工工牌、访客证、楼层权限、总部策略、远程办公 VPN、门禁日志，全都要配合。Claude Code 的配置与认证系统也是这样：它解决的不只是“能不能登录”，而是“**谁的规则生效、谁的身份可信、令牌如何续命**”。

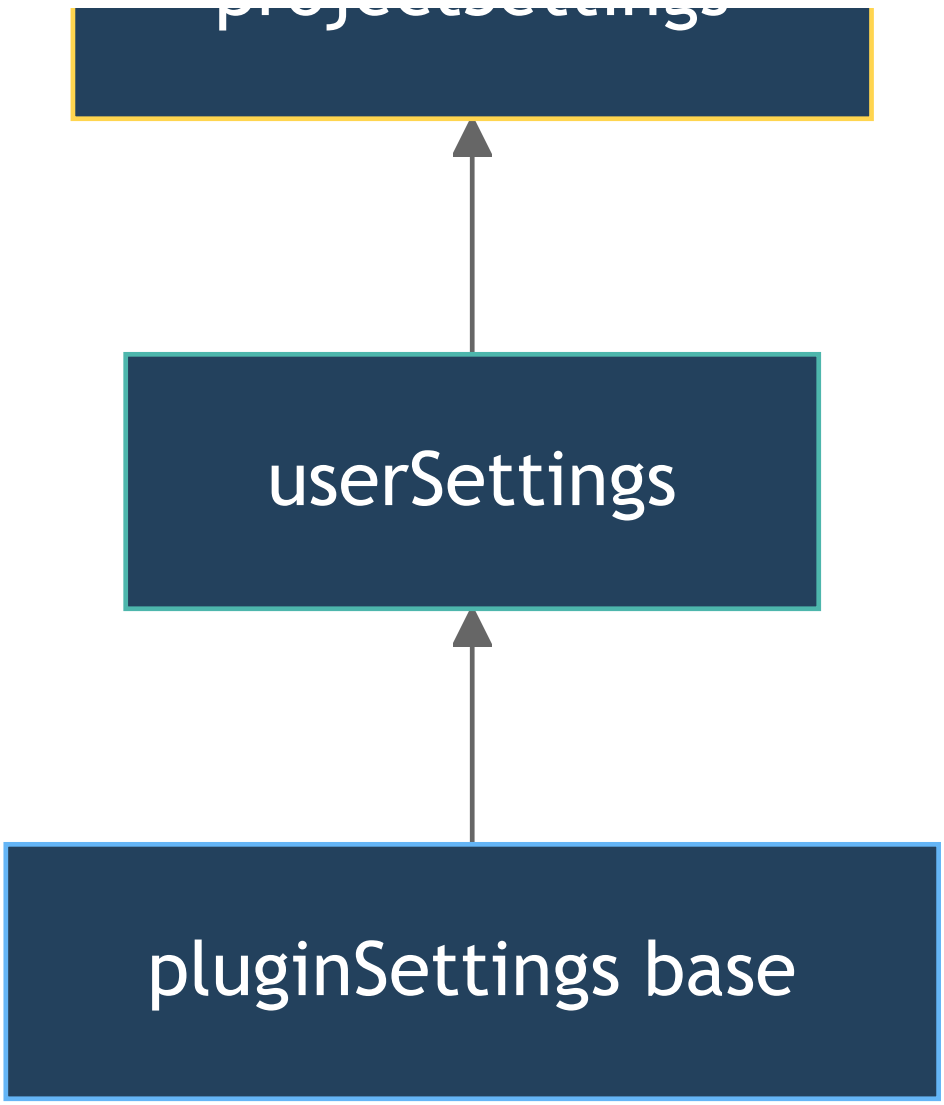
这一章先回答一个问题

当配置散在多个来源、认证又跨 OAuth、secure storage、Bridge JWT 时，Claude Code 是怎么把它们收拢成一个体系的？

24.1 设置不是一个文件，而是一条优先级链

`settings.ts` 里最关键的不是某个 schema，而是**加载顺序**。Claude Code 会按优先级依次把不同来源叠起来，形成最终的 effective settings。


policySettings
远程 / MDM / managed file / HKCU后备
flagSettings
远程实验与内联覆盖
localSettings
projectSettings



`policySettings` 为什么最特殊

它不是简单的“再多一个配置文件”，而是“先选胜者，再整体生效”：

- 远程托管设置；
- macOS plist 或 Windows HKLM；
- `managed-settings.json` 与 `managed-settings.d/*.json`；
- 最后才是 Windows HKCU 作为低优先级兜底。

这段逻辑明显在服务企业部署场景。对普通用户来说，settings 更像偏好设置；对企业来说，settings 其实是**安全策略分发机制**。

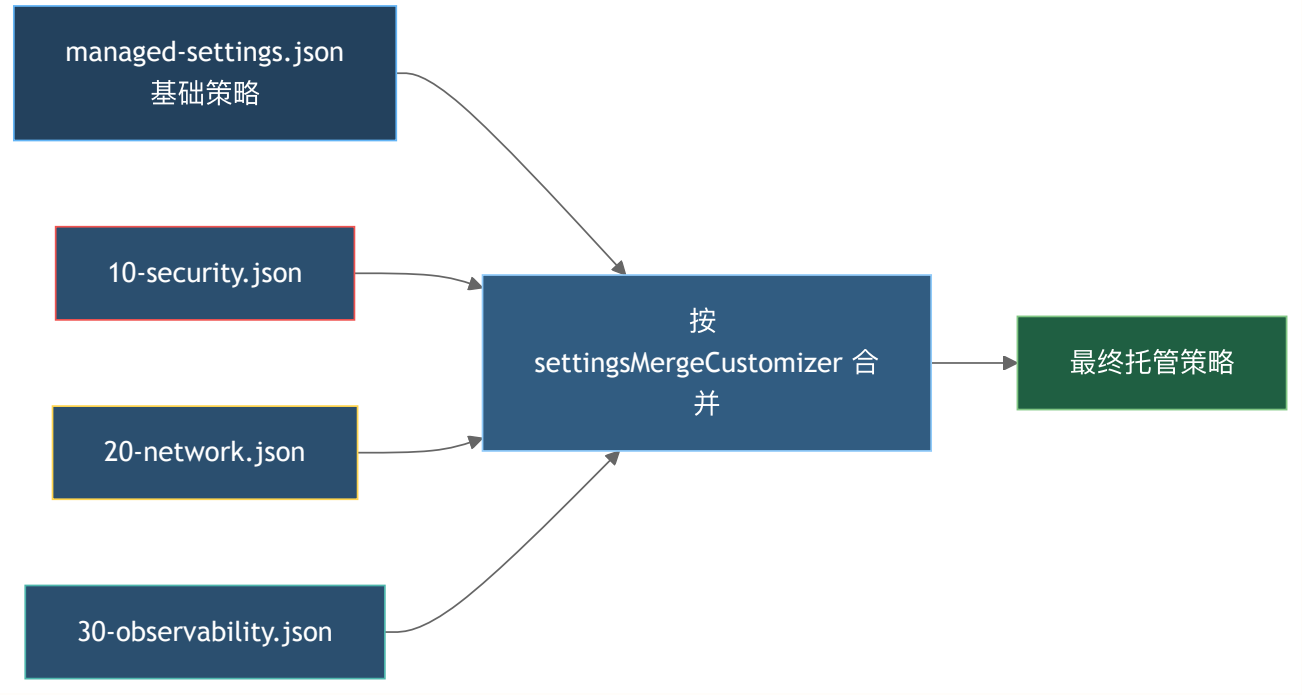
24.2 `managed-settings.d` 这种细节，透露了很强的运维味道

`loadManagedFileSettings()` 不只读取一个 `managed-settings.json`，还会读取 `managed-settings.d/` 目录下的多个 drop-in 文件，并按文件名字母序合并。

这背后的思想非常像 Linux 生态里常见的：

- `systemd` drop-in
- `sudoers.d`
- Nginx `conf.d`

也就是说，Claude Code 的 managed settings 不是按“一个大 JSON 文件集中改”设计的，而是按“不同团队可以分别下发策略片段”设计的。



更进一步，`settingsMergeCustomizer()` 对数组用的是“拼接并去重”，而不是无脑替换。这样权限、Hooks、沙箱列表之类的数组配置就能自然叠加。

为什么写入又变成“数组直接替换”

`updateSettingsForSource()` 在写回某个 settings source 时，又故意对数组采用“由调用方给出最终状态，整体替换”的策略。这是为了避免局部 merge 带来的状态不一致。

简单说：

- **读配置**：尽量合并；
- **写配置**：尽量精确。

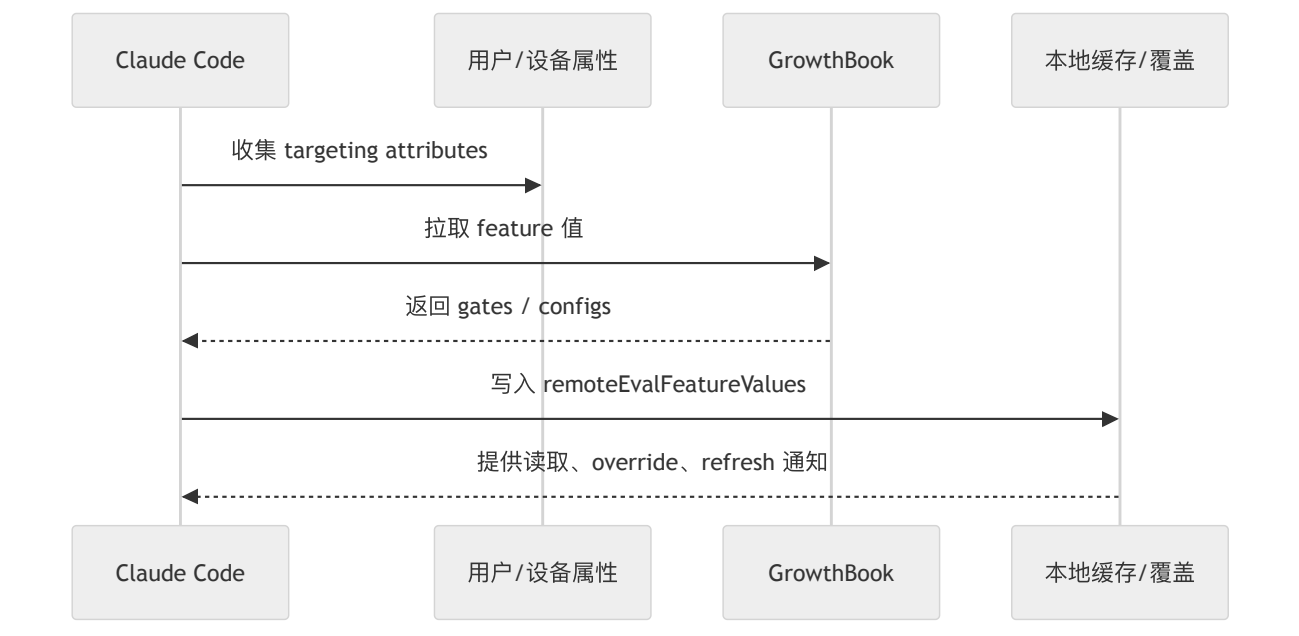
这是一种很有经验的工程取舍。

24.3 GrowthBook 不是花哨开关，而是运行时治理入口

前面很多章已经见过 feature gate，这一章要把它放回配置治理里看。

`services/analytics/growthbook.ts` 说明了几件事：

- GrowthBook 客户端带着用户与设备属性做远程评估；
- feature 值会被缓存；
- 支持环境变量覆盖；
- 支持本地 config override；
- 刷新后还能通知长期驻留对象重建。



这解释了为什么像 `auto mode`、`bypassPermissions`、某些默认模型之类的行为，不全是硬编码在本地配置里的。Claude Code 把一部分运行时治理交给了远程配置系统。

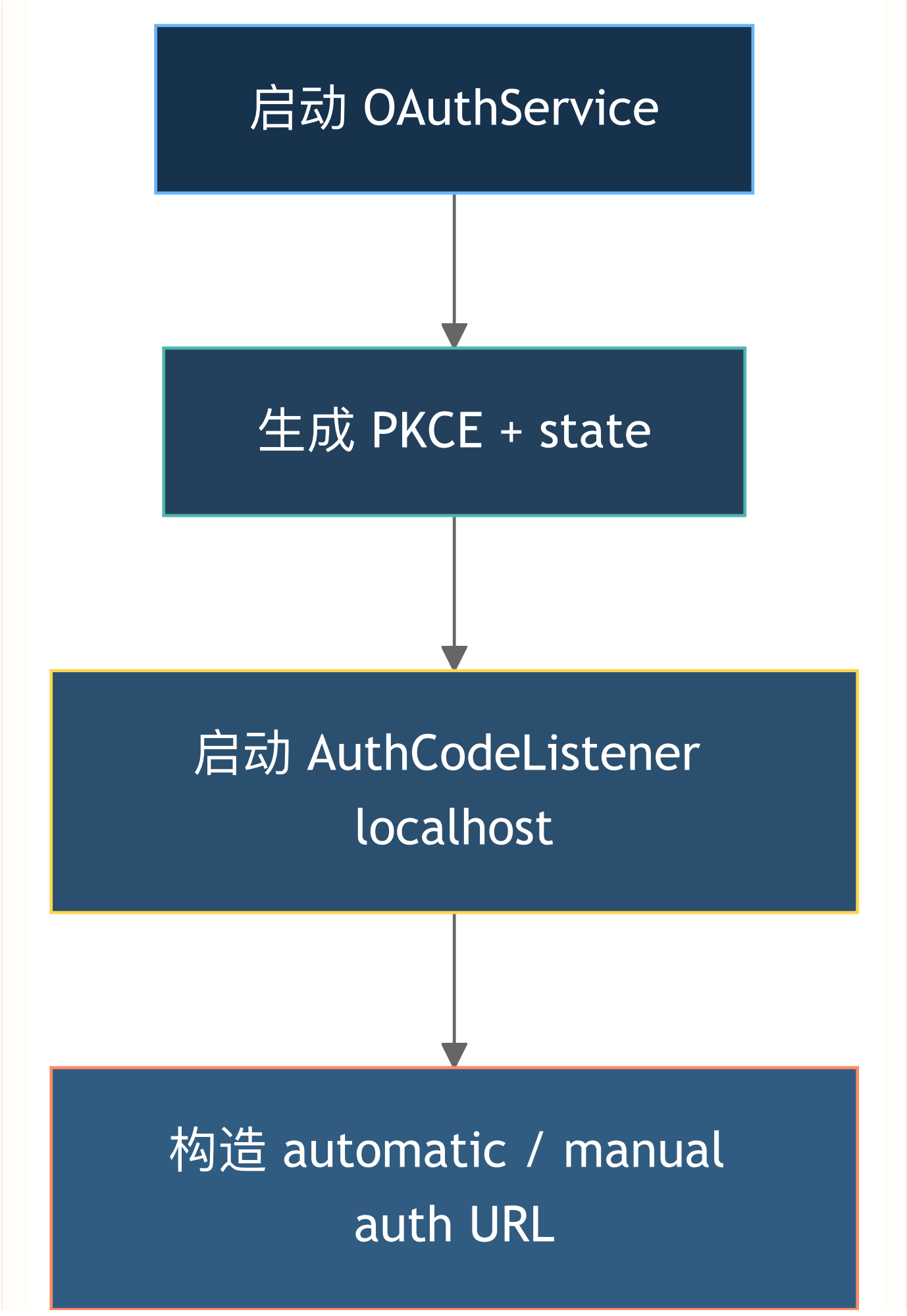
从产品角度看，这非常重要：

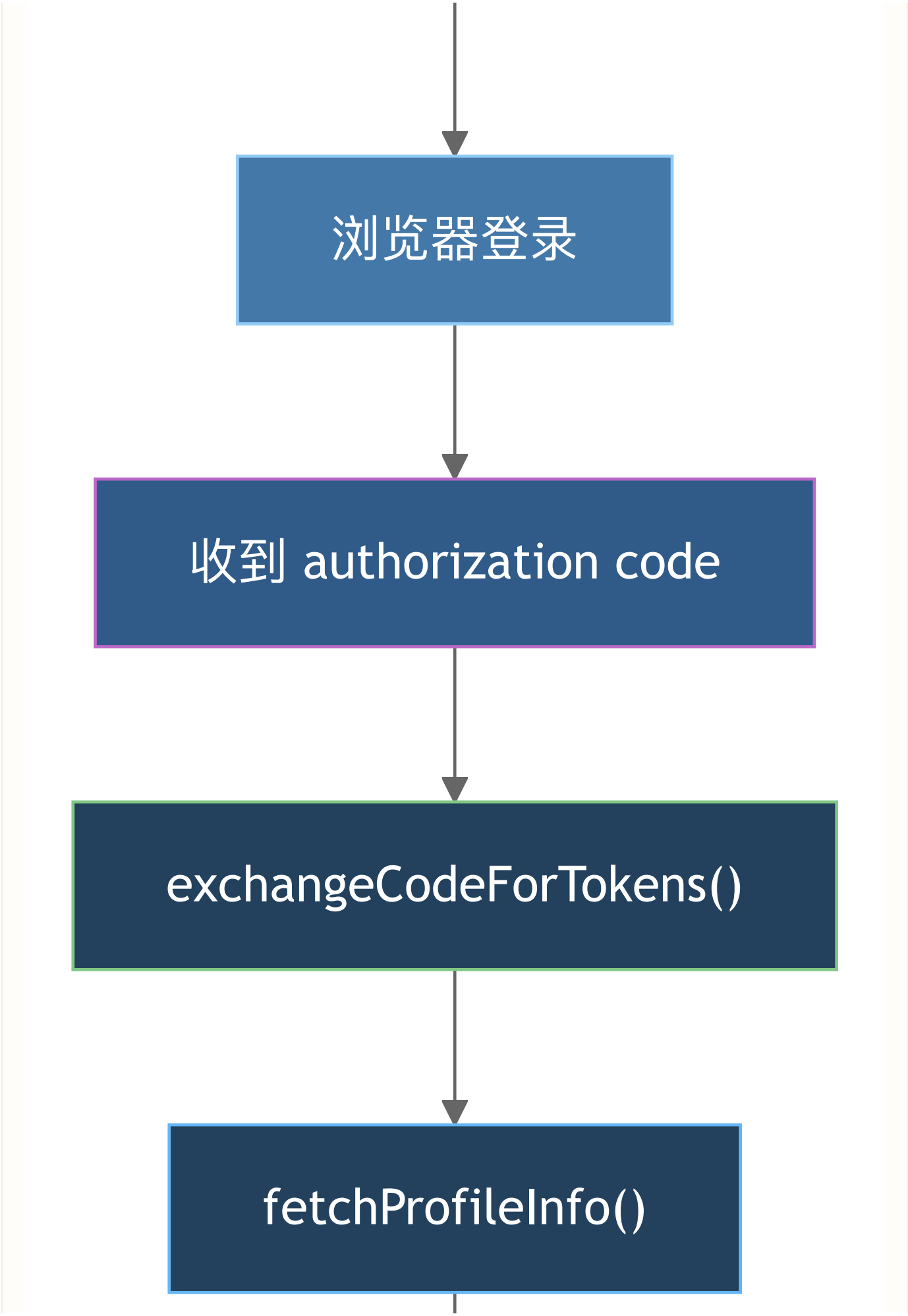
- 出问题时能快速熔断；
- 新特性能灰度发布；
- 企业或内部用户能有不同策略。

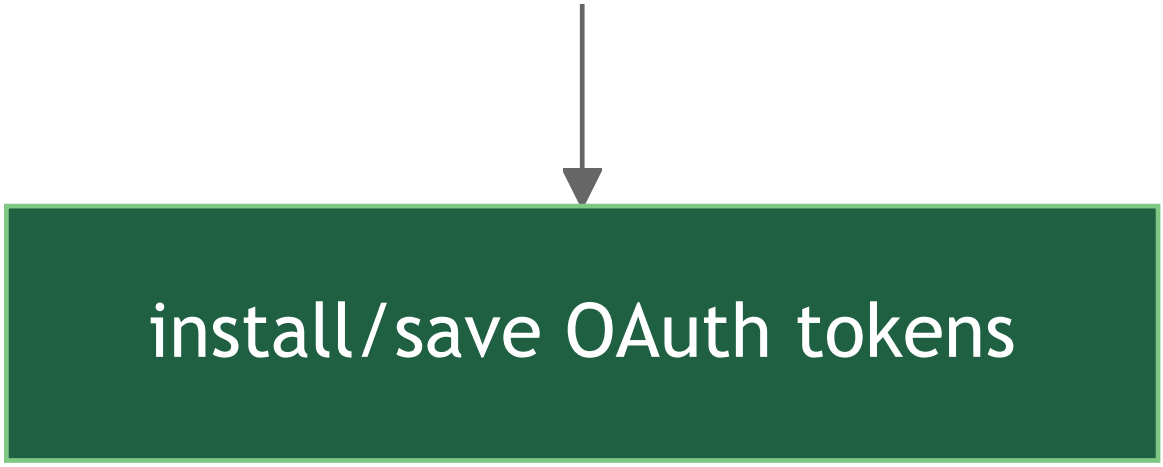
24.4 OAuth 登录链：从浏览器到 secure storage

`OAuthService` 做的是标准但完整的 OAuth 2.0 Authorization Code + PKCE 流程：

- 生成 `code_verifier` 与 `code_challenge`
- 启动本地回调监听
- 同时准备自动跳转 URL 和手动复制 URL
- 拿到 authorization code 后交换 token
- 再取 profile 信息并格式化成本地 token 结构


启动 OAuthService
生成 PKCE + state
启动 AuthCodeListener
localhost
构造 automatic / manual
auth URL


浏览器登录
收到 authorization code
exchangeCodeForTokens()
fetchProfileInfo()



这里的“体验优化”也很有意思：

- 同时支持自动回调和手动贴码；
- `keychainPrefetch.ts` 在 macOS 上会并行预取 keychain 里的两类凭据，尽量把几十毫秒启动成本藏在主模块加载期间；
- `saveOAuthTokensIfNeeded()` 又会把 token 写进 secure storage，而不是依赖纯内存。

也就是说，Claude Code 对认证的目标不是“能登进去”，而是：

- 首次登录要稳；
- 下次启动要快；
- 多终端刷新要尽量不互相打架。

24.5 401、刷新锁与 JWT：真正难的是长期会话

一旦会话变长，真正的麻烦就不是登录，而是**持续保持有效身份**。

OAuth token 刷新

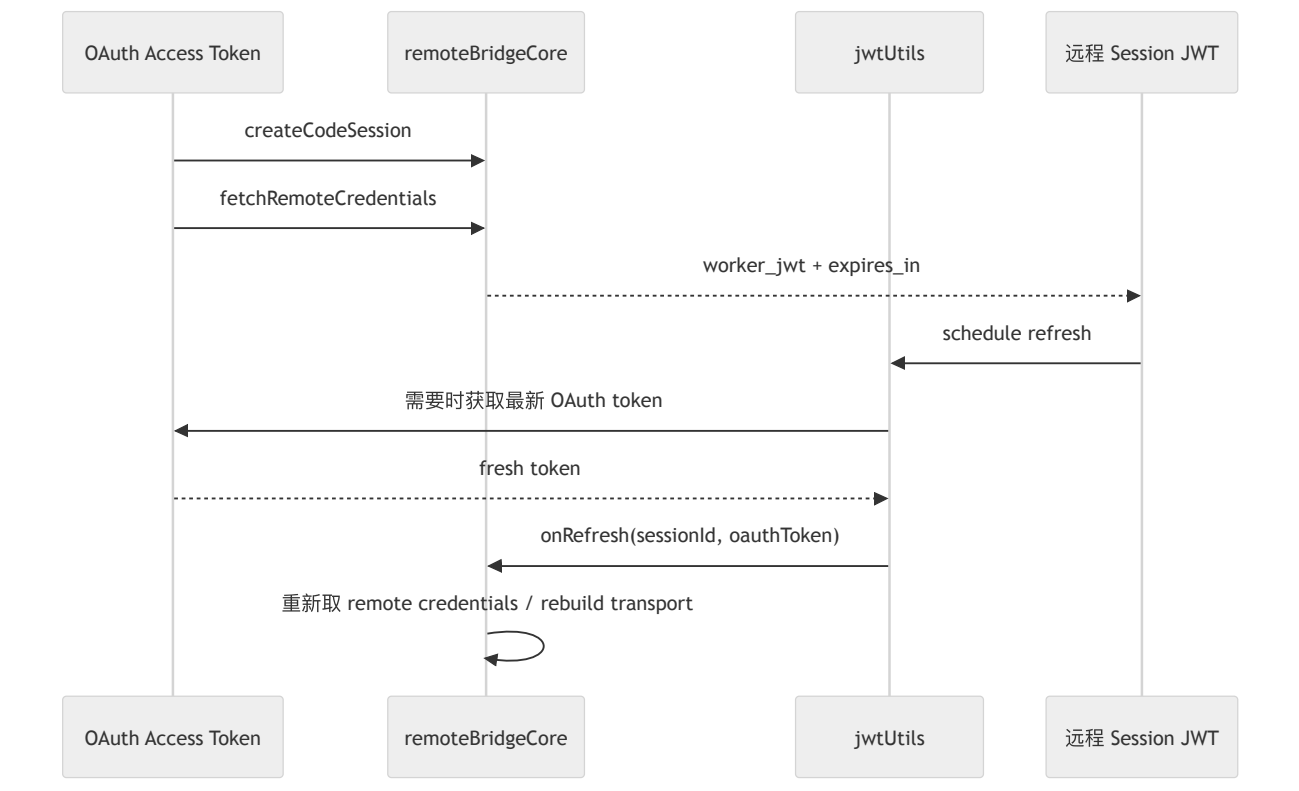
`auth.ts` 里能看到一整套“多进程别互相抢刷”的机制：

- 401 时先清缓存，重新从 secure storage 读；
- 如果发现别的进程已经刷新过，就直接复用；
- 真的要刷新时，会加锁，避免多个进程同时刷新；
- 刷新完成后再回写 secure storage 并清掉 memoize/cache。

Bridge JWT 刷新

远程桥接还有一层 session JWT。`jwtUtils.ts` 负责：

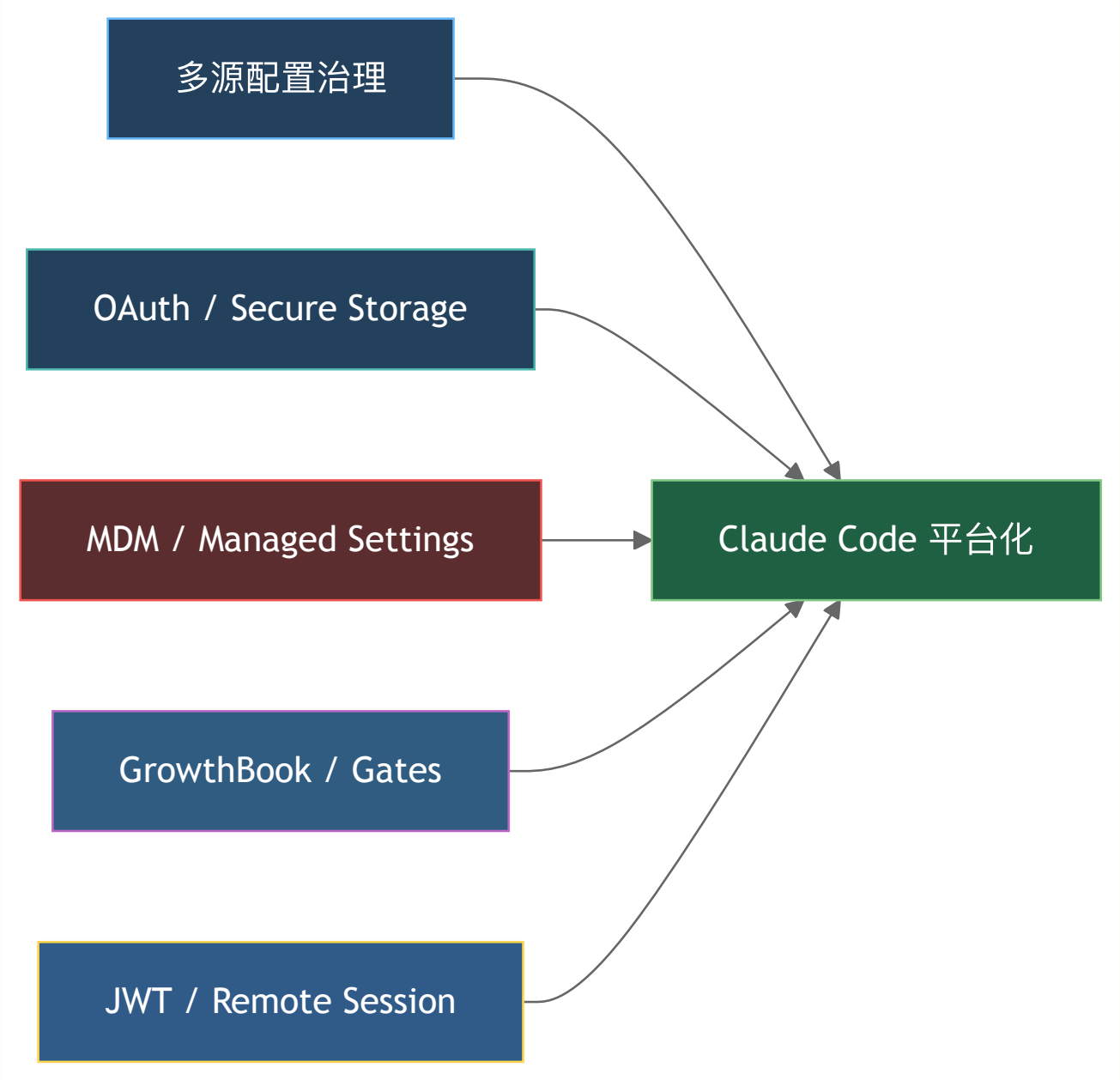
- 解码 `exp`
- 提前一定 buffer 安排刷新
- 在无法解码时用 fallback interval
- 失败后重试
- cancel / cancelAll 以防孤儿定时器



这也是为什么 `remoteBridgeCore.ts` 里会有专门的 `recoverFromAuthFailure()`：远程会话不是浏览器里一次请求结束就算了，它是长连接，是要和过期、掉线、唤醒、401 一直周旋的。

24.6 把这些放在一起看，就会发现它其实在做“平台治理”

这一章最容易被误读成“杂项收尾”。其实恰好相反，它是 Claude Code 从个人 CLI 走向平台级产品的关键基础设施。



如果没有这些能力，Claude Code 充其量是：

- 一个本地好用的 AI CLI；
- 一个单人开发者工具；
- 一个短会话的命令行助手。

而有了这些能力，它才可能变成：

- 可治理的组织工具；
- 可远程控制的会话平台；
- 可灰度、可熔断、可统一策略下发的产品。

🔭 深水区（架构师选读）

这一章最值得带走的设计思想是“配置、认证、远程会话不是边角料，而是平台骨架”。很多工程团队前期把这些内容当杂务，最后产品越做越大时被迫重构。Claude Code 源码里反而很早就把 settings source、managed policy、OAuth refresh、Bridge JWT refresh 这些通路理顺了，这也是它能承载企业级场景的重要原因。

本章小结

Claude Code 的配置与认证系统回答了三个问题：规则从哪里来、谁说了算、身份怎么长期有效。把这三件事理顺之后，前面几章讲的权限、沙箱、自动模式才真正有了落点。

关键源码索引

- managed file 与 drop-ins：`settings.ts`

- 设置写回与替换策略：`settings.ts`
- settings 合并与优先级：`settings.ts`
- 从磁盘加载全部 settings：`settings.ts`
- MDM 首源优先逻辑：`mdm/settings.ts`
- 托管路径约定：`managedPath.ts`
- GrowthBook 客户端与缓存：`growthbook.ts`
- OAuthService 与 PKCE：`oauth/index.ts`
- 授权 URL 与 token 交换：`oauth/client.ts`
- macOS keychain 预取：`keychainPrefetch.ts`
- OAuth token 保存与 401 刷新：`auth.ts`
- Bridge JWT 刷新调度：`jwtUtils.ts`
- 远程凭据与 401 恢复：`remoteBridgeCore.ts`

逆向提醒

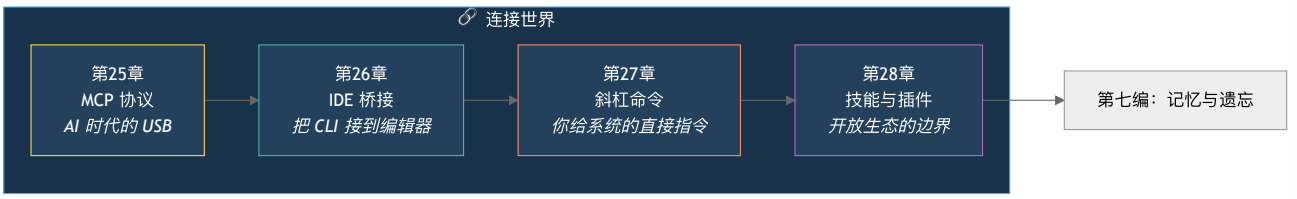
这一章覆盖了很多“跨目录协作”的基础设施。它们单个文件看起来像杂项，但组合起来才构成完整治理体系。阅读时不要只盯着某一个 `settings.ts` 或 `auth.ts`，要沿着调用链一起看。

32

第六编：连接世界

> *USB 统一了键盘、鼠标、U 盘和相机的连接方式，电脑才真正变成“万物接口”。*
>
> Claude Code 的外部连接层也在做同样的事：把 **外部工具**、**IDE 编辑器**、**用户命令**、**第三方技能/插件** 接到同一套执行引擎上。

本编总览



本编四章速览

章	标题	核心问题	生活类比
25	MCP 协议：AI 时代的 USB	内置工具已经很多了，为什么还要接外部能力？	USB 标准接口
26	IDE 桥接：一座把 CLI 接到编辑器的桥	一个命令行工具，为什么要维护一整套桥接层？	电话的三代演进
27	斜杠命令：你给系统的直接指令	工具是 AI 调的，命令是人调的，这两套入口怎么配合？	餐厅菜单系统
28	技能与插件：安全地扩展能力边界	开放生态以后，怎么避免系统被扩展反噬？	手机 App Store

这一编建议怎么读

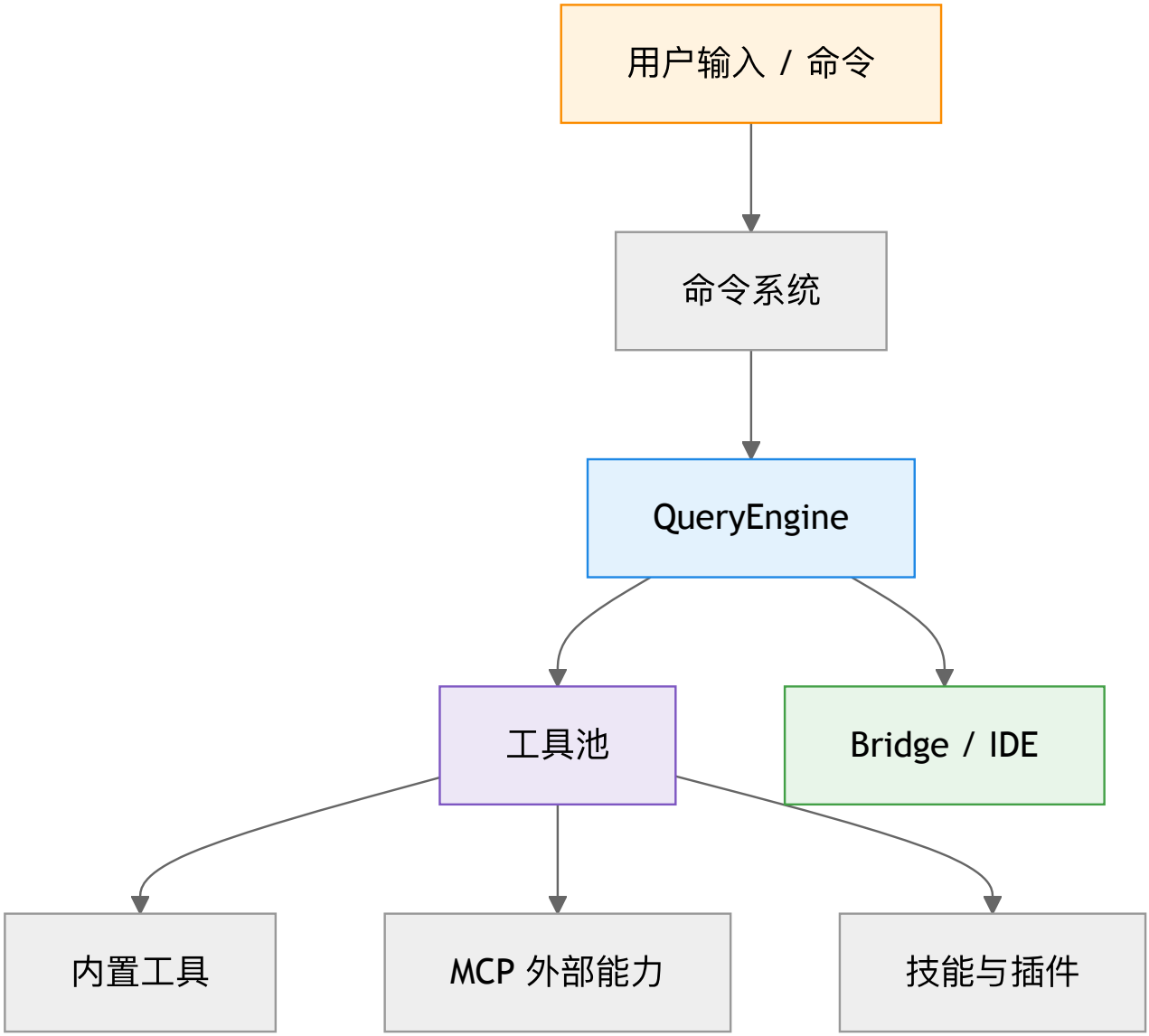
◉ ○ ○

🌱 初学者 🔧 开发者 🏗️ 架构师

先读第 27 章。它最接近日常体验，能帮你理解 `/help`、`/config`、`/batch` 这类命令背后的结构。

第 25 章和第 28 章最关键。一个讲协议，一个讲生态，是把 Claude Code 从产品看成平台的起点。

第 26 章最值得细读。CLI、桥接、远程会话、认证续期、会话恢复，这些组合在一起才是真正的系统难点。



本编阅读目标

读完这一编，你应该能回答三个关键问题：Claude Code 怎样接外部世界、怎样把能力送进 IDE、怎样在开放生态里保持秩序。

33

MCP 第六编

第25章：MCP 协议：AI 时代的 USB

生活类比：USB 接口

在 USB 出现之前，键盘、鼠标、打印机、相机各有各的接口。统一标准之后，电脑不用提前认识每一种设备，也能把它们接进来。MCP 对 AI 工具生态做的，就是这件事。

这一章先回答一个问题

Claude Code 已经有很多内置工具了，为什么还要再搞一层 MCP 协议，把外部服务接进来？

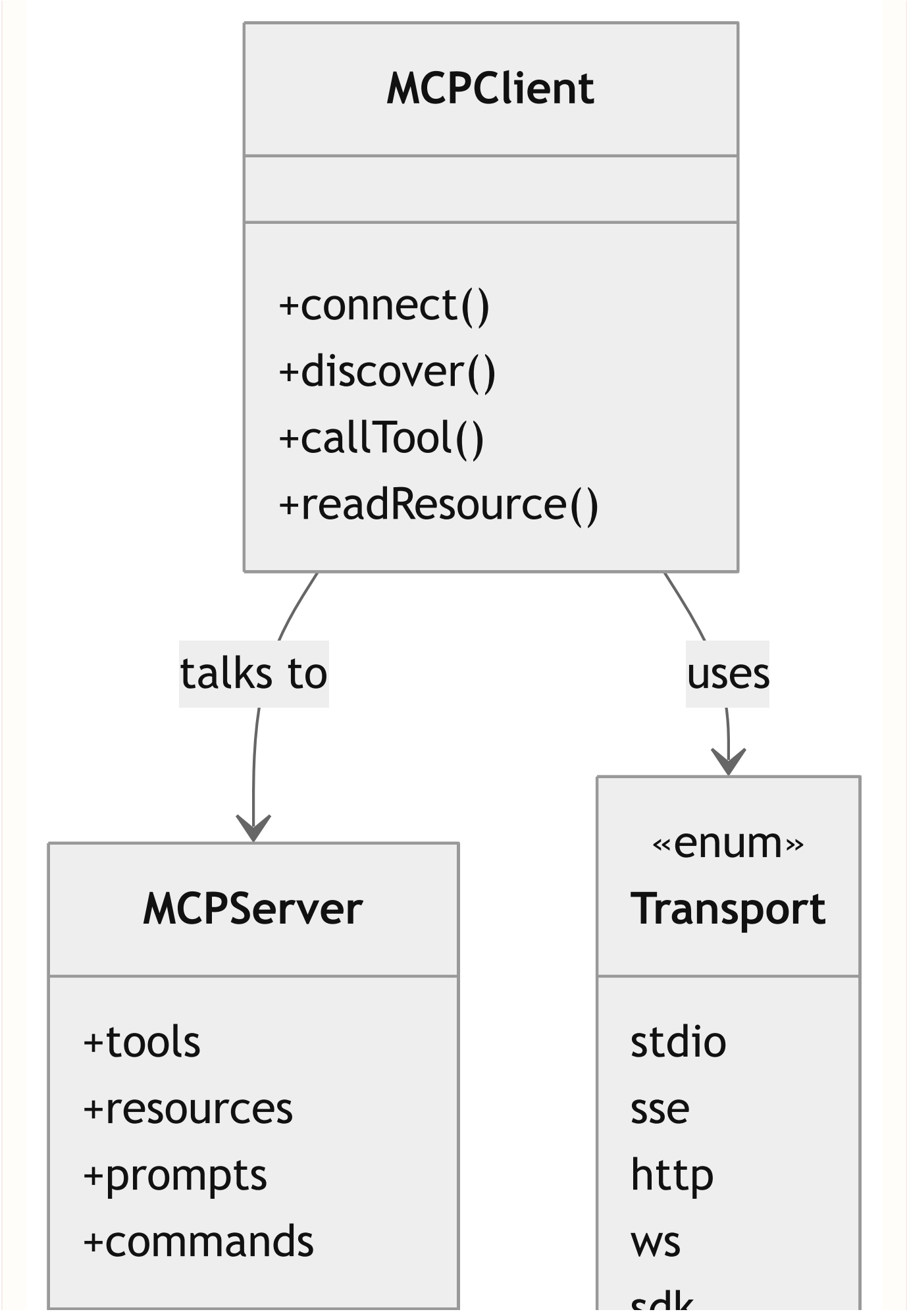
Claude Code 的答案是：**内置工具解决“常见能力”，MCP 解决“无限扩展”**。你不可能把企业内网、数据库、工单系统、设计平台、测试平台全都内建进一个 CLI，但你可以给它们一个统一接口。

25.1 MCP 不是“又一个插件系统”，它先解决的是语言统一

MCP 在源码里被拆成几类核心对象：

- Server 配置
- Transport 传输方式
- Tools / Resources / Prompts / Commands 四类能力
- Client 侧发现、缓存、调用和错误恢复

在 `types.ts` 里，Claude Code 先把这些概念用类型固定下来，再在 `client.ts` 里把它们串成真正的运行流程。


MCPClient
+connect()
+discover()
+callTool()
+readResource()
talks to
uses
MCPServer
+tools
+resources
+prompts
+commands
«enum»
Transport
stdio
sse
http
ws
sdk

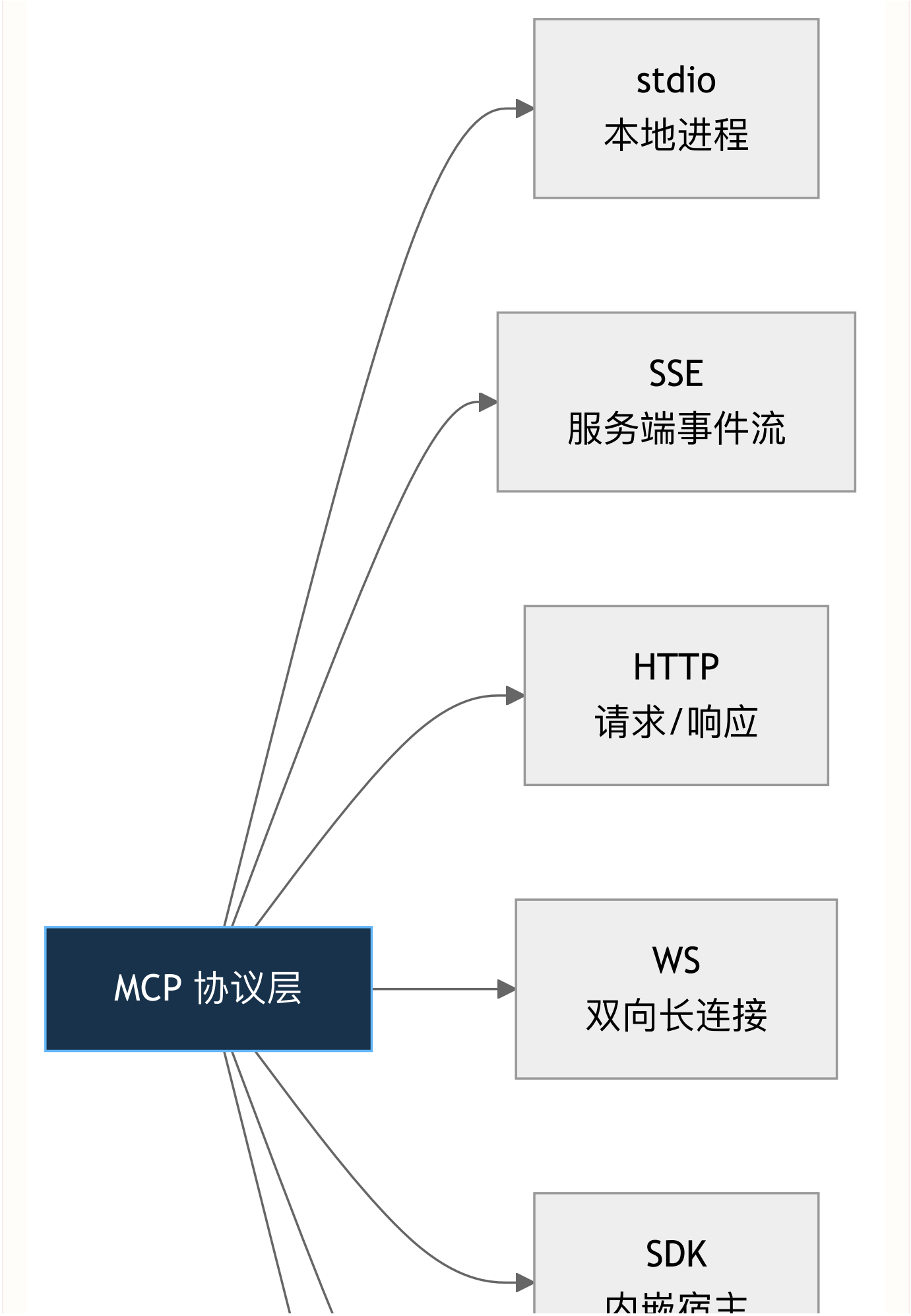
从读者角度看，最关键的是别把 MCP 只理解成“远程工具”。它实际上想统一四种东西：

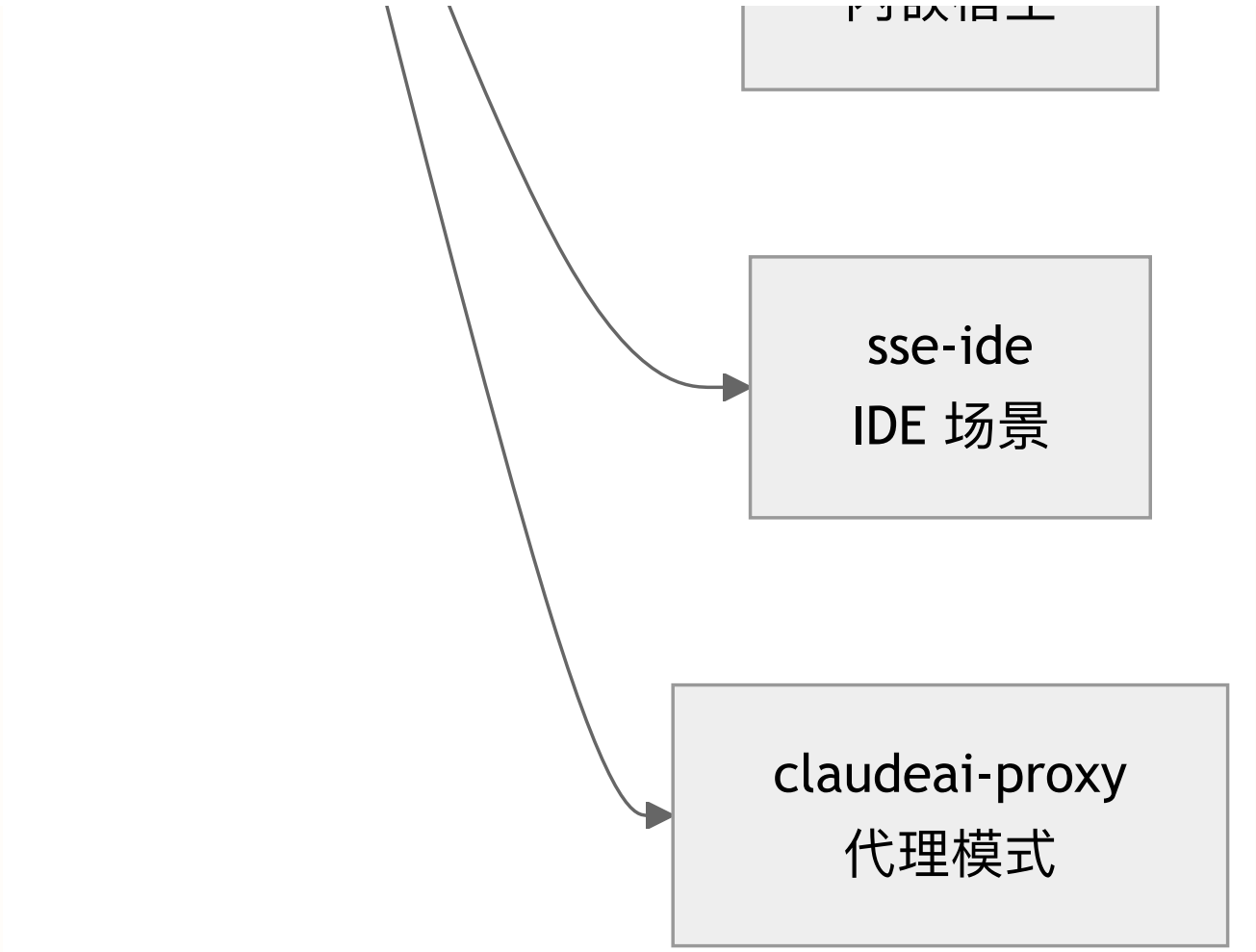
能力	你可以把它理解成什么
Tool	一个能被调用的动作
Resource	一份可读取的数据
Prompt	一段预制好的上下文模板
Command	一条用户可直接触发的入口

这意味着 MCP 接进来的不只是“能做事的手”，还有“能读的资料”和“能启动的入口”。

25.2 七种传输方式，背后是“同一协议，不同水管”

源码里最醒目的一个细节，是 `Transport` 并不只有一种。`types.ts` 明确列出了 `stdio`、`sse`、`sse-ide`、`http`、`ws`、`sdk`，还配套了不同的 server config 变体。


stdio
本地进程
SSE
服务端事件流
HTTP
请求/响应
MCP 协议层
WS
双向长连接
SDK



为什么要这么多传输?

- 本地命令型服务，最适合 `stdio`
- 长时间推送事件的服务，适合 `SSE`
- 已有 Web 服务的系统，适合 `HTTP`
- 需要实时双向通信的系统，适合 `WS`
- 嵌入式宿主或内部集成，适合 `SDK`

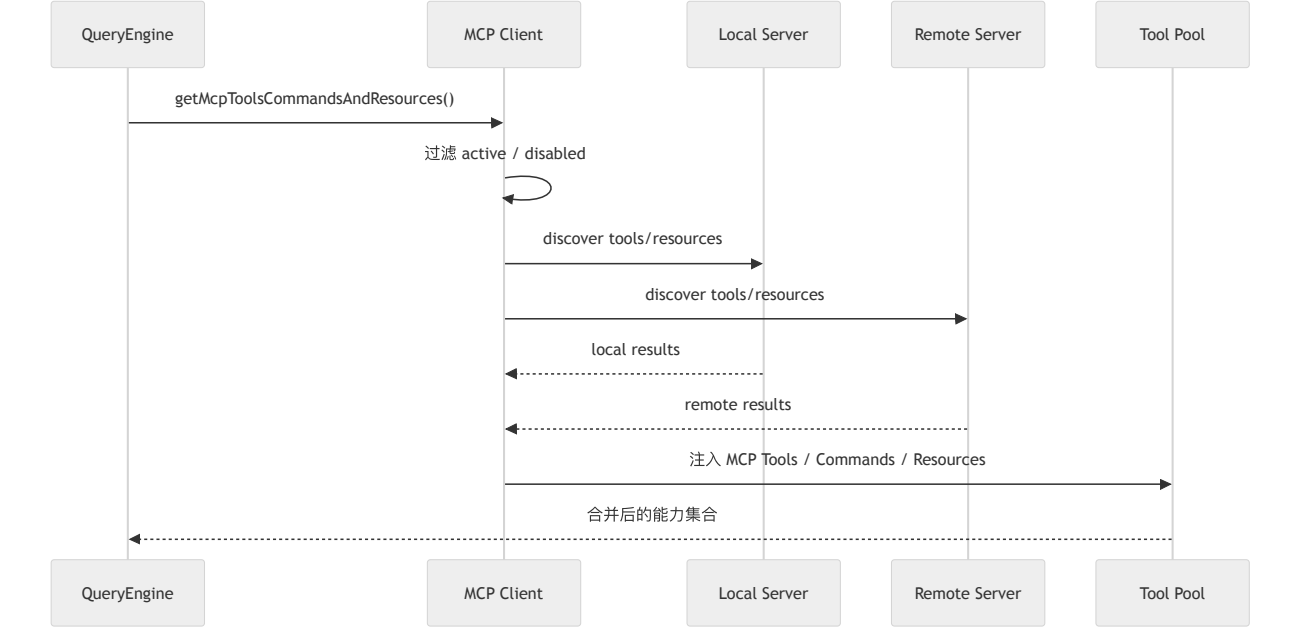
所以这里不是“协议设计不统一”，恰恰相反，是**上层统一，下层适配现实世界**。

25.3 真正复杂的地方不在调用，而在发现与装配

MCP 的难点不是“发个 JSON-RPC 请求”，而是：系统如何知道现在有哪些服务连着、每个服务暴露了什么、是否可用、哪些要变成工具、哪些要变成命令。

`client.ts` 里的 `getMcpToolsCommandsAndResources()` 就在做这件事。它会：

1. 区分 active 和 disabled servers
2. 按本地 / 远程特征分组
3. 并发拉取每个 server 的 tools、commands、skills、resources
4. 在资源能力存在时，自动补进 `ListMcpResourcesTool` 和 `ReadMcpResourceTool`
5. 把这些结果折叠进 Claude Code 自己的工具池和命令系统

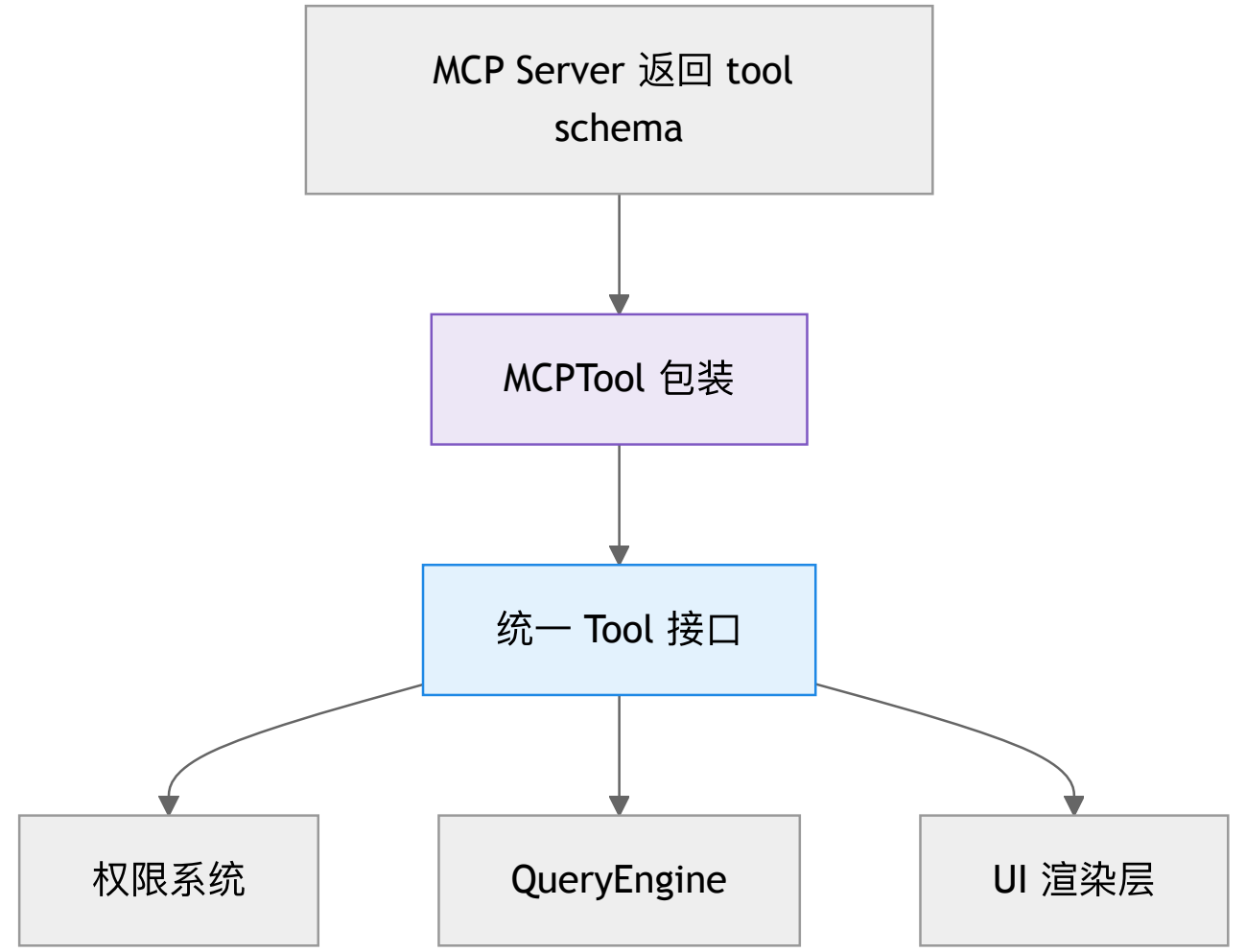


这里特别能体现 Claude Code 的工程取向：它没有把 MCP 当成“外面那一套”，而是把它当成**原生能力的延伸层**。

25.4 MCPTool 的价值，是把“外部工具”伪装成“系统内工具”

`MCPTool.ts` 很短，但意义很大。它不是为某个具体工具写业务，而是做一层**协议转译**：

- 外部 server 说“我有个名叫 x 的工具”
- Claude Code 把它包装成内部 `Tool` 接口
- 后面的权限、UI、进度、结果渲染继续按统一路径走



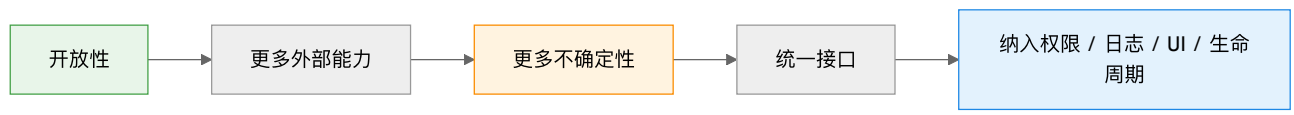
这就是为什么从模型角度看，内置工具和 MCP 工具“长得几乎一样”。统一接口把生态复杂度压在了系统内部，而不是扔给模型。

25.5 设计取舍：开放生态最怕的不是慢，而是失控

MCP 把能力接进来以后，马上会遇到三个问题：

1. 一个外部工具是否应该出现在当前会话里？
2. 一个外部资源是否能被当前用户读到？
3. 一个外部命令是否应该直接出现在 / 菜单里？

Claude Code 的处理方式很成熟：**先发现，再包装，再纳入原有治理系统**。它没有单独再造一套“外部工具特权通道”。



这也是读源码时最值得学的一点：平台化不是“什么都能接”，而是“接进来以后仍然受系统秩序约束”。

深水区（架构师选读）

MCP 在 Claude Code 里最有价值的，不是“支持外部协议”这件事本身，而是它证明了一个 CLI 型 Agent 也能长成平台。协议、类型、发现、统一工具接口、资源读写、命令透出，这些组合在一起，构成的是一个可扩展 Agent 运行时，而不只是一个会说话的命令行程序。

本章小结

MCP 让 Claude Code 从“自带一组工具的产品”升级成“可接入外部能力的平台”。它的重点不是炫技，而是把多种传输和多类能力统一进既有的工具与命令体系。

关键源码索引

- MCP 类型与传输枚举：`types.ts`
- Server 配置与作用域定义：`types.ts`
- 发现并合并工具/命令/资源：`client.ts`
- 资源工具自动补齐：`client.ts`
- MCPTool 统一包装层：`MCPTool.ts`

逆向提醒

MCP 相关类型和客户端代码在还原层里相对完整，但具体 server 端行为不在本仓库中。本章分析的是 Claude Code 作为 MCP Client 的实现，而不是所有外部 MCP Server 的真实能力边界。

34

Bridge 第六编

第26章：IDE 桥接：一座把 CLI 接到编辑器的桥

生活类比：电话的三代演进

固定电话解决了“远距离通话”，移动电话解决了“人不能被线拴住”，智能手机又把通信、应用和持续连接合成到了一起。Claude Code 的桥接系统，也是在一次次迭代里把 CLI 的能力送进 IDE。

这一章先回答一个问题

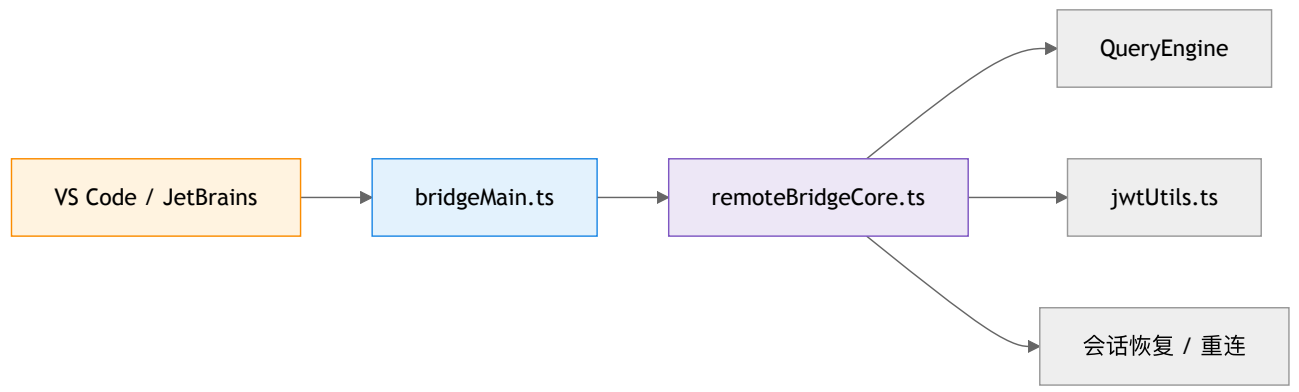
一个本质上跑在终端里的工具，为什么要维护一大块桥接代码，还要处理会话、认证、重连、恢复和远程模式？

因为真正的开发现场不只在终端里。程序员写代码、看 diff、点文件、读报错，大多发生在编辑器里。Claude Code 如果不能把能力稳定地“伸进 IDE”，它就永远只是一把好用但割裂的外部工具。

26.1 Bridge 不是装饰层，而是“第二个入口”

从源码结构看，`bridgeMain.ts` 并不是轻飘飘的 adapter。它更像一个独立入口：

- 维护会话
- 解析命令行参数
- 管理 ingress token
- 接收 IDE 侧消息
- 与主执行引擎交换事件



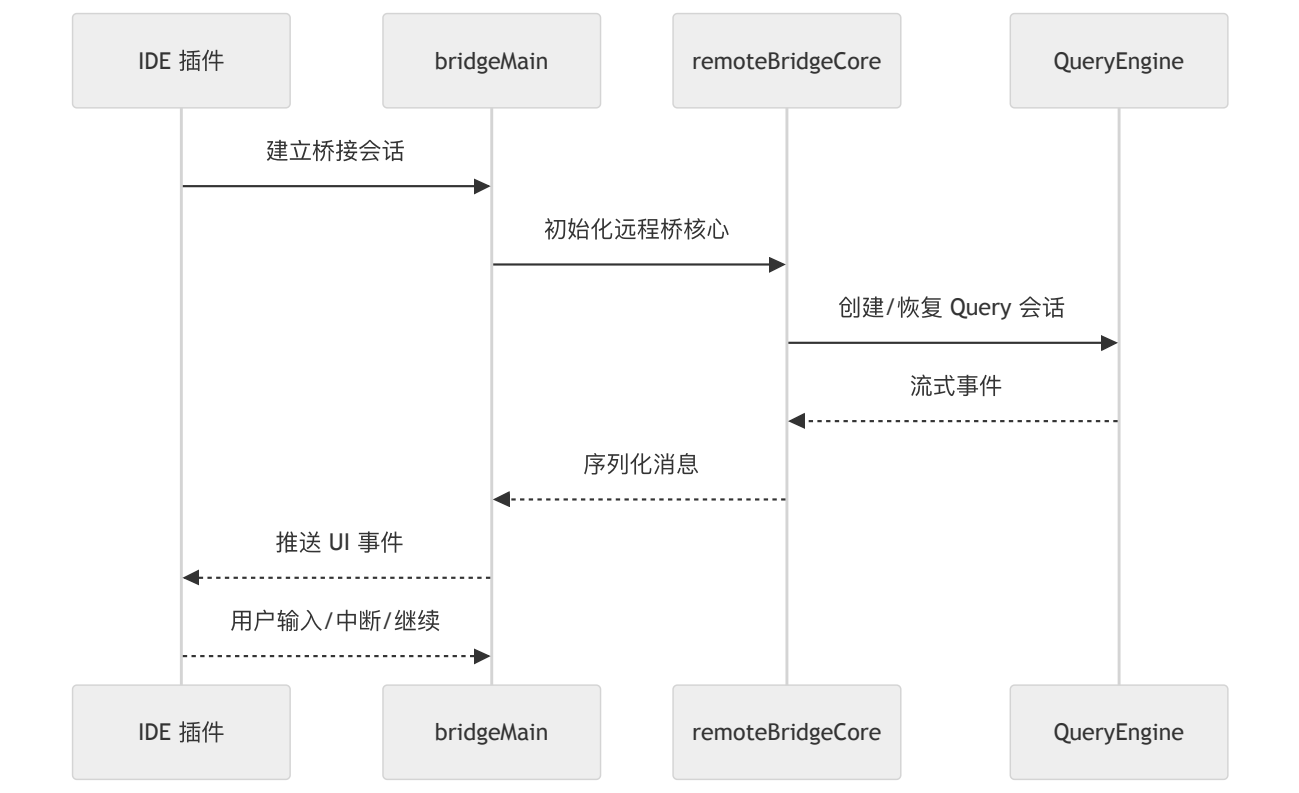
这说明桥接层不是“把终端输出贴到侧边栏里”，而是另一种会话宿主。

26.2 `runBridgeLoop()` 解决的是“长连接世界”的问题

在 `bridgeMain.ts` 里，`runBridgeLoop()` 会持续处理活跃会话、token、兼容 ID、心跳和断线恢复。

它面对的是编辑器场景下典型的几类不稳定因素：

- IDE 进程重启
- 插件更新
- 网络波动
- 登录状态变化
- 一个项目里存在多个并发会话



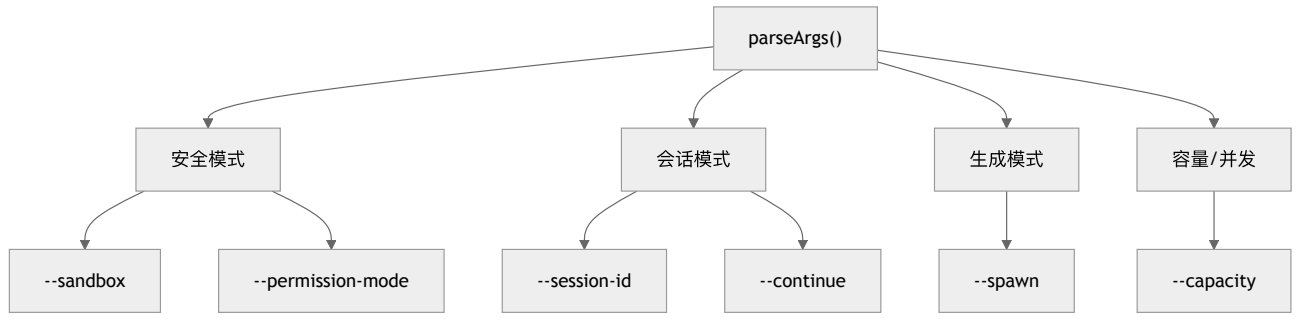
这里最值得注意的是“恢复”二字。终端里一次断开，用户大不了重开；IDE 里如果每次焦点切换、插件刷新都丢上下文，体验就彻底崩了。

26.3 参数解析说明：桥接层其实支撑了很多模式

`parseArgs()` 并不只认一个简单的 `--bridge`。它还要处理：

- `--sandbox`
- `--permission-mode`
- `--session-id`
- `--continue`
- `--spawn`
- `--capacity`
- `--create-session-in-dir`

这些参数背后的意思很直白：桥接层必须能启动新会话、接管旧会话、按不同安全模式运行，还得为多实例场景留余地。



所以你可以把 Bridge 理解成：**为 IDE 场景重新包了一层更可控的 CLI 外壳。**

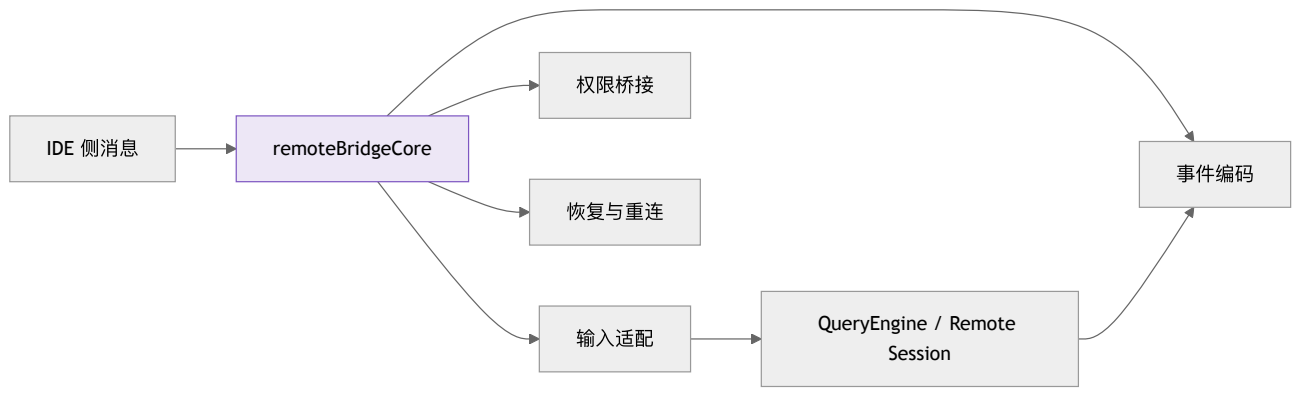
26.4 `remoteBridgeCore` 才是桥中间那块真正承重的钢梁

`bridgeMain.ts` 偏入口和宿主管理，`remoteBridgeCore.ts` 更像“协议和状态中心”。

它要做三件重活：

1. 把 IDE 消息翻译成 Claude Code 能吃的事件

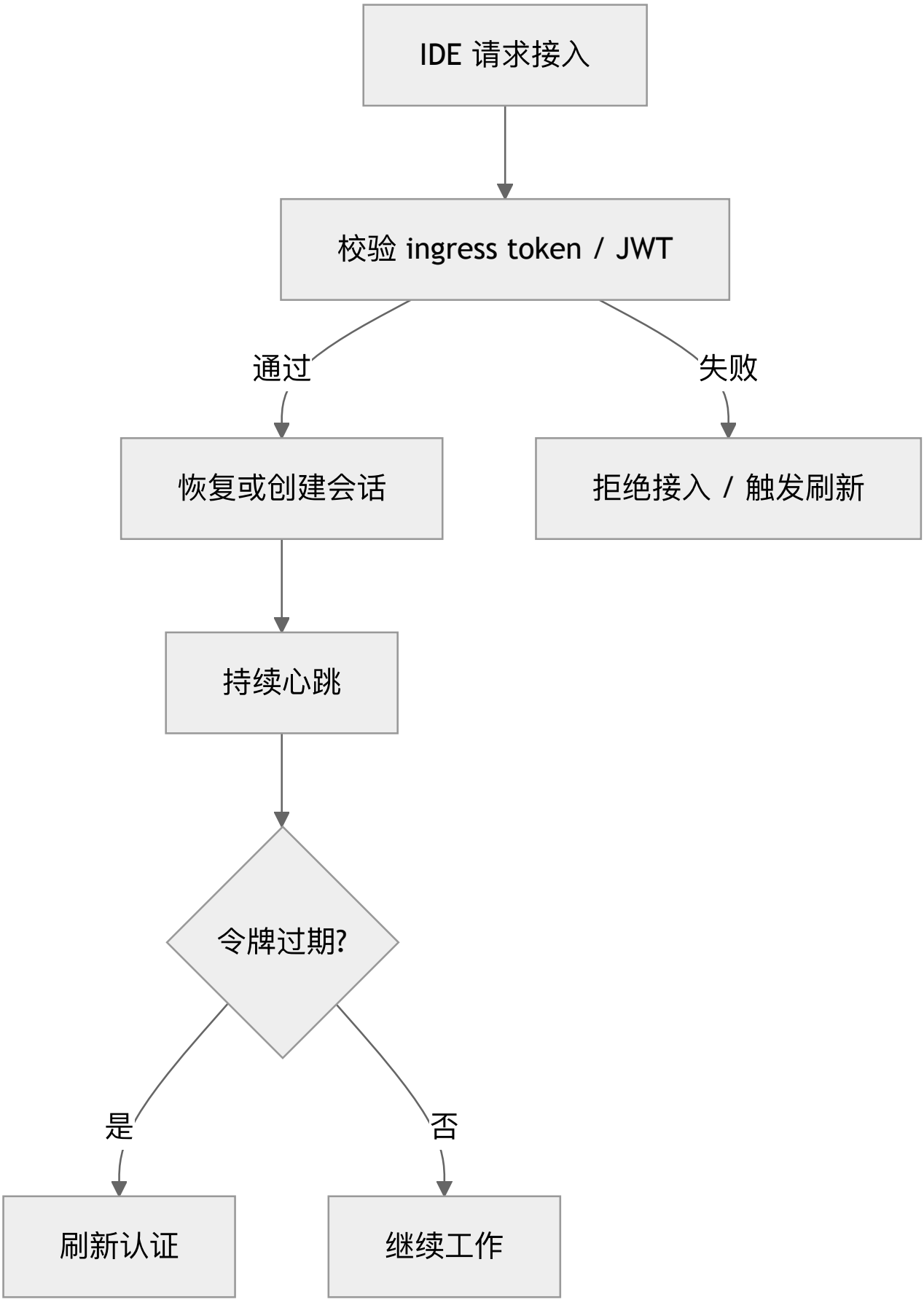
2. 把 Claude Code 产生的流式事件，重新编码成 IDE 可消费的结构
3. 处理远程权限、认证刷新、会话恢复等复杂情形



这就是为什么桥接代码会长。它不是业务重复，而是**交互边界一旦变成“跨进程 + 跨宿主 + 可能跨网络”，复杂度就会指数上升**。

26.5 JWT 和 ingress token：桥不是谁都能走的

如果桥接系统只是“谁都能连进来”，那它就会变成一条安全后门。所以桥接层还要配合 `jwtUtils.ts` 处理身份和有效期。



从设计思想上说，这很像“内部系统也要零信任”。即便是本地 IDE 发来的请求，也不能默认安全。

26.6 设计取舍：为什么不用“更简单”的办法

你也许会想，为什么不直接让 IDE 插件自己集成模型调用，而一定要桥接到 CLI?

Claude Code 的答案很工程化：

- CLI 已经拥有完整的工具系统、权限系统、记忆系统
- 桥接复用这套核心，比在 IDE 里重写一遍更稳
- 统一宿主有助于让终端、SDK、IDE 三种入口共享同一个 QueryEngine

它承担了更高的桥接复杂度，换来的是核心能力只维护一份。这对长期演化是更划算的。

🔭 深水区（架构师选读）

桥接层最值得借鉴的，不是具体的协议细节，而是“把 CLI 当成能力内核，把 IDE 当成宿主界面”的分层思想。真正昂贵的是执行引擎、一致性和治理能力，UI 宿主反而应该尽量薄。Claude Code 把这个分层做得很彻底。

本章小结

Bridge 让 Claude Code 不再只是一个终端工具，而是一套可被 IDE 托管的会话系统。入口、协议、认证、恢复、重连看起来繁琐，但正是这些细节决定了编辑器内体验是否稳定。

关键源码索引

- Bridge 主循环：`bridgeMain.ts`
- 桥接参数解析：`bridgeMain.ts`

- 远程桥核心初始化：`remoteBridgeCore.ts`
- 桥接消息与会话处理：`remoteBridgeCore.ts`
- 远程会话恢复末段逻辑：`remoteBridgeCore.ts`
- JWT 工具：`jwtUtils.ts`

逆向提醒

Bridge 代码很完整，但 IDE 插件本体不在这个仓库里。本章能讲清 Claude Code 这一端“如何被接入”，却无法完全覆盖每个编辑器宿主的 UI 细节与扩展实现。

35

Commands 第六编

第27章：斜杠命令：你给系统的直接指令

生活类比：餐厅菜单

厨房会自己决定火候，但不会自己决定你今天吃宫保鸡丁还是番茄炒蛋。菜单给顾客直接表达意图的入口。Claude Code 的斜杠命令，也是在给用户保留“直接下指令”的通道。

这一章先回答一个问题

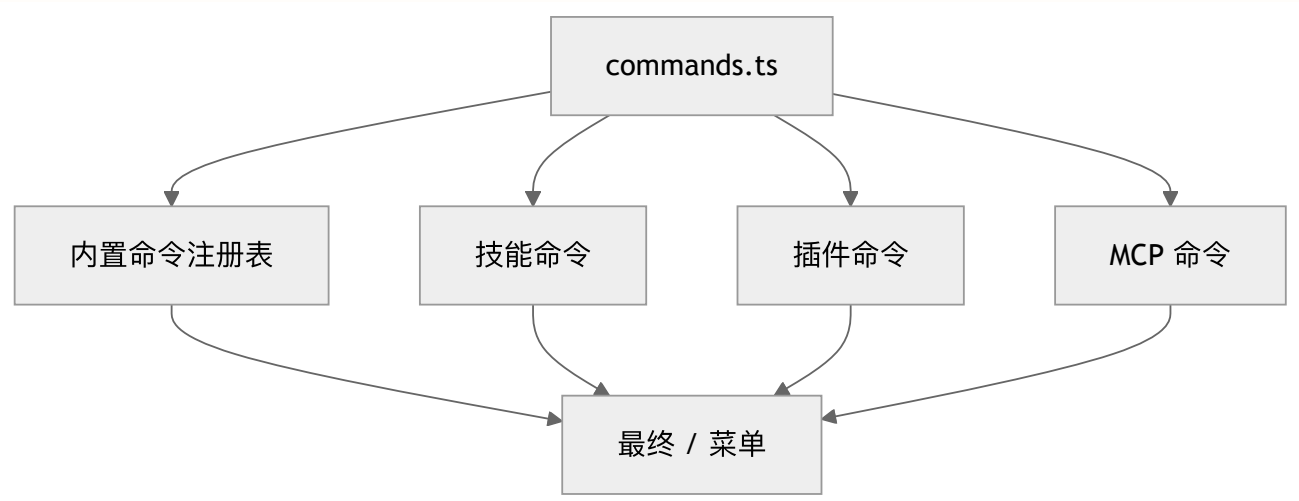
工具是 AI 在循环里自主调用的，命令是用户显式敲出来的。这两套入口为什么要并存？边界到底在哪里？

答案很关键：**工具负责执行，命令负责发令**。工具偏向“Agent 的手”，命令偏向“用户的嘴”。看懂这点，你就不会再把 `/compact`、`/batch`、`/config` 这种能力和 Bash、Read、Edit 混为一谈。

27.1 `commands.ts` 是菜单，不是厨房

`commands.ts` 集中定义了内置命令注册表。源码里最直观的感受，就是它很“显式”：

- 每条命令都有名称和描述
- 命令是否可见、是否可用可以被过滤
- 之后还会和技能命令、插件命令、MCP 命令一起合并



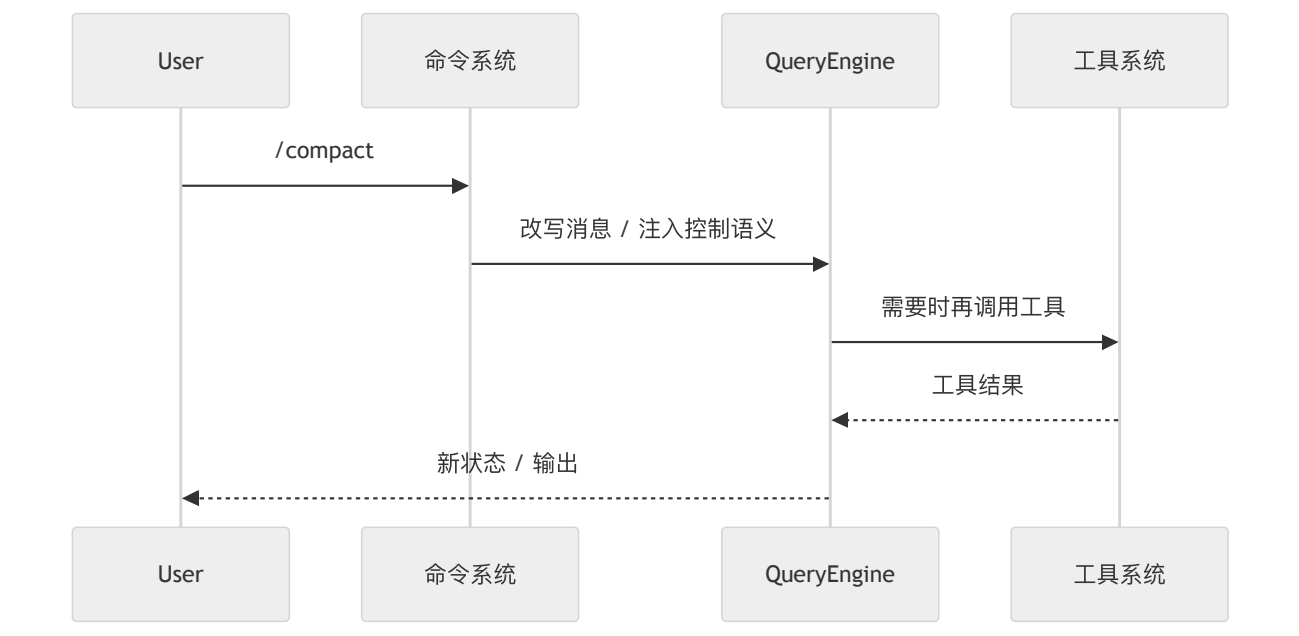
这说明 `/` 不是“写死的一串命令”，而是一个会动态扩展的命令空间。

27.2 为什么命令不能直接等于工具

命令和工具最大的差别在于触发者：

维度	命令	工具
触发者	用户	模型
目的	直接表达意图	执行中间步骤
生命周期	通常是入口动作	常嵌在 Agent Loop 中
展示位置	/ 菜单、帮助、补全	系统提示和工具池

所以 `/config` 不等于某个工具，`/memory` 也不只是一个读文件动作。命令常常是**高层语义入口**，会触发一段更复杂的流程。



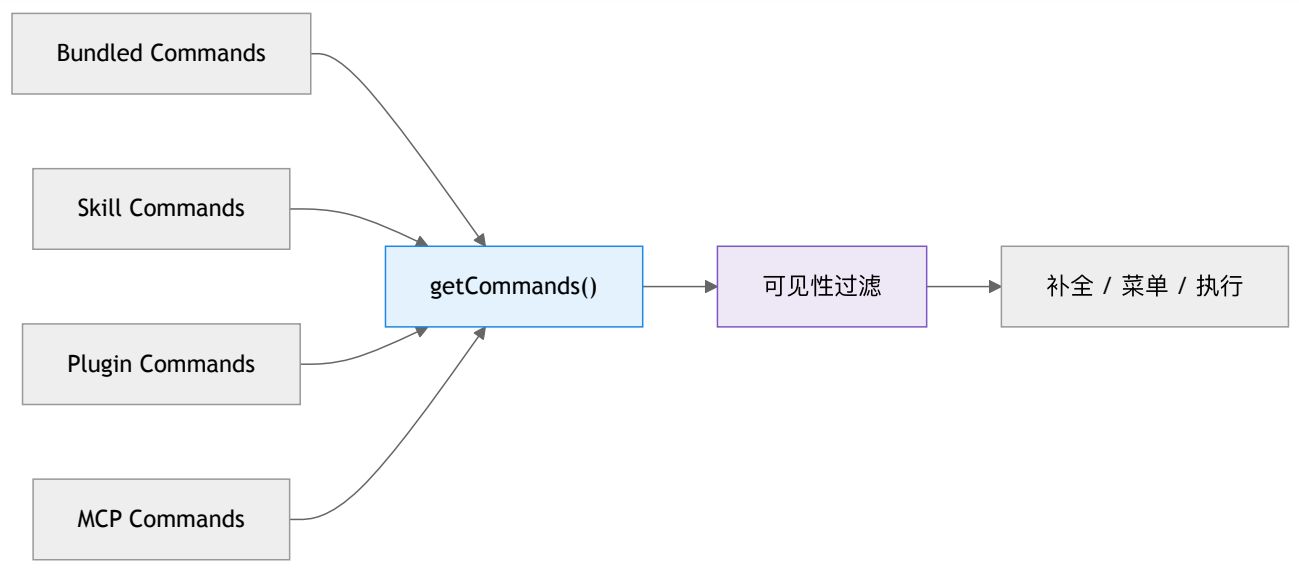
这就是为什么 Claude Code 同时保留两层抽象，而不是把一切都糊成“都是调用”。

27.3 真正有意思的是“命令注册表会继续长”

在 `commands.ts` 之后，系统还会把：

- 技能目录里生成的命令
- 插件暴露的命令
- MCP server 提供的命令

继续合并进来。`loadAllCommands()` 和 `getCommands()` 做的，就是从“内置命令世界”走向“动态命令世界”。



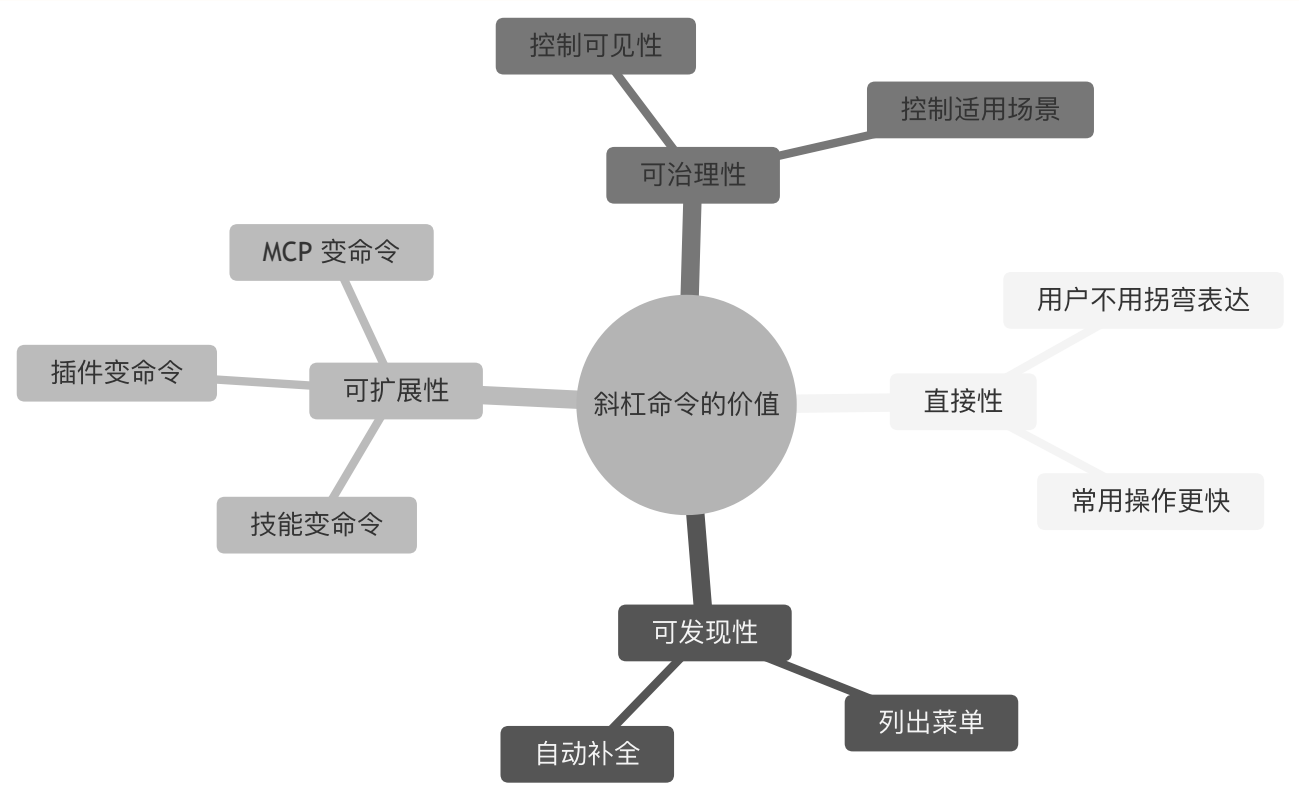
这很像一个现代编辑器的 Command Palette：你看到的是统一入口，背后则是多来源拼装。

27.4 命令系统为什么比看起来更重要

如果没有命令系统，Claude Code 就会变成纯聊天产品。你要做的每件事都得“绕一圈说出来”。有了命令系统以后，它变成了：

- 聊天式 Agent
- 命令式工具
- 可编排的 CLI

三者合体的产品。



这里的产品哲学很清楚：Claude Code 不想把所有交互都推给“自然语言理解”。它承认有些动作就应该有明确入口。

27.5 `loadSkillsDir` 告诉我们：命令也可以是“文档驱动生成”的

这一章最漂亮的一点，藏在技能系统和命令系统的连接处。`loadSkillsDir.ts` 会把带 frontmatter 的技能定义，变成真正能执行的命令。

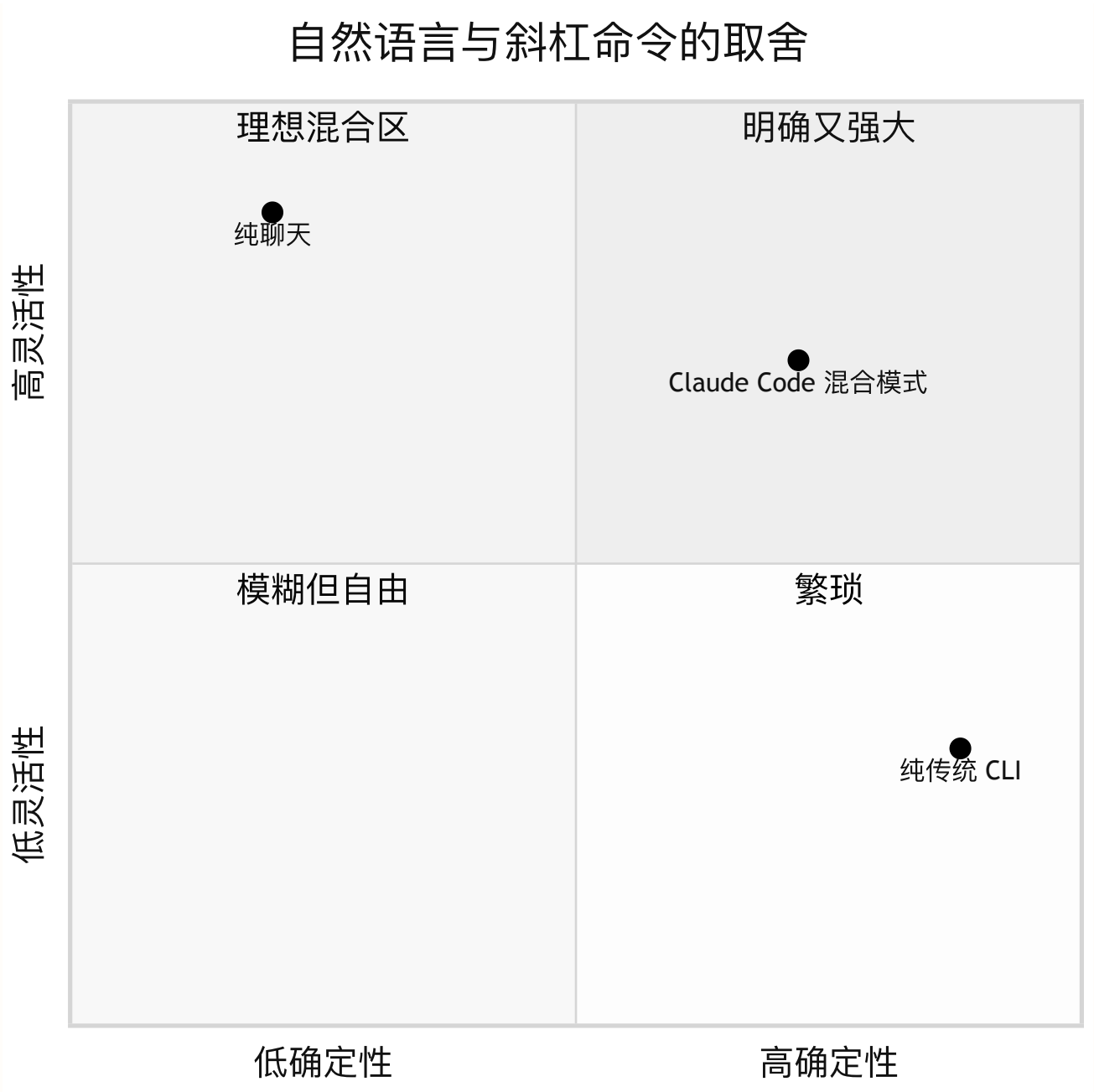
这意味着：

- 命令不一定都要手写一大段 TypeScript
- 一部分命令可以由结构化文档生成
- 命令系统因此具备很强的生态扩展性

所以 Claude Code 的命令系统不是“纯代码注册”，而是“代码注册 + 文档注册”的混合体。

27.6 设计取舍：为什么菜单要显式，而不是全部自然语言

自然语言很灵活，但也很模糊。命令恰好相反：不够自由，却很明确。



Claude Code 选择的是中间路线：

- 想探索，就自然语言
- 想直达，就 / 命令
- 想自动执行，就让 Agent 去调用工具

三种方式各有位置，不互相取代。

深水区（架构师选读）

命令系统的真正价值，不是“提供一堆快捷方式”，而是给产品增加第二种交互语法。用户、模型、插件、MCP 都能围绕这套语法继续扩展。长期看，这比完全押注自然语言更稳，也更适合工程产品。

本章小结

斜杠命令是 Claude Code 的用户控制面。它把高频动作变成明确入口，又通过命令合并机制让技能、插件和 MCP 继续长进来，最终形成统一的命令空间。

关键源码索引

- 内置命令注册表：`commands.ts`
- 技能与插件命令合并：`commands.ts`
- 可用性过滤：`commands.ts`
- 统一加载入口：`commands.ts`

- MCP Skill 命令拼接：`commands.ts`

逆向提醒

目录统计里常见“87 个命令目录”和“88 个命令”的说法，差异通常来自隐藏命令、动态命令和门控命令。读源码时要区分“目录里存在”与“运行时会出现”这两个概念。

36

Plugins Skills 第六编

第28章：技能与插件：安全地扩展能力边界

生活类比：App Store

开放生态让手机变得强大，但如果任何 App 都能随便读通讯录、偷偷驻留后台、改系统设置，手机也会很快失控。Claude Code 的技能与插件系统面对的是同一个难题。

这一章先回答一个问题

当第三方也能往 Claude Code 里加能力时，系统怎么做到既开放，又不把安全边界打穿？

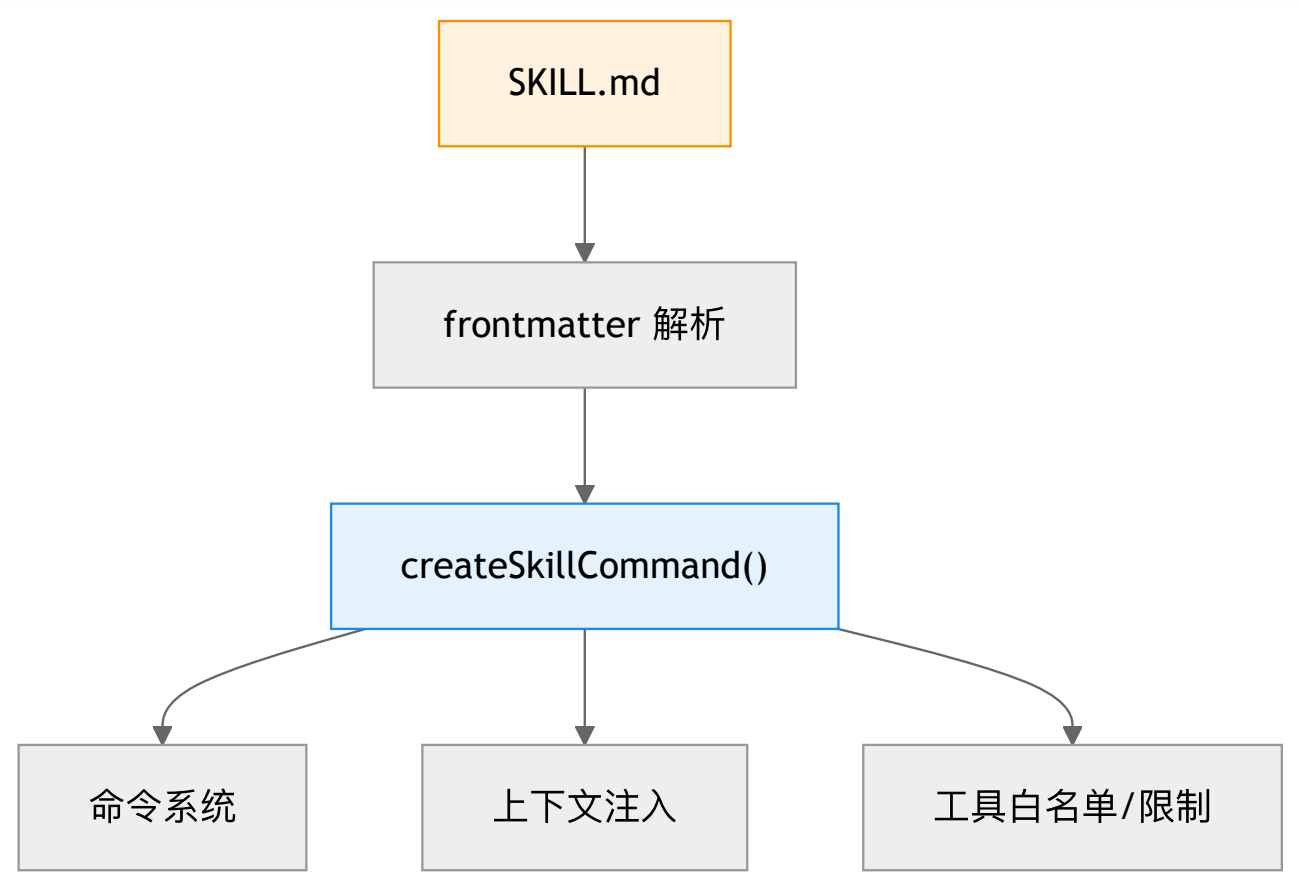
答案不在单一审核环节，而在整套设计里：**技能是轻量提示封装，插件是带生命周期的扩展单元，最终都要回到权限、命令、工具这三条治理链里。**

28.1 技能系统的本质，是把“经验”变成可调用单元

`loadSkillsDir.ts` 里最有意思的，不是扫描文件夹，而是 frontmatter 字段非常丰富：

- `description`
- `allowed-tools`
- `argument-hint`
- `arguments`
- `when_to_use`
- `model`
- `hooks`
- `context=fork`
- `agent`
- `effort`
- `shell`

这说明技能不是一段随手写的 prompt，而是有元数据、有触发条件、有工具约束的结构化资产。

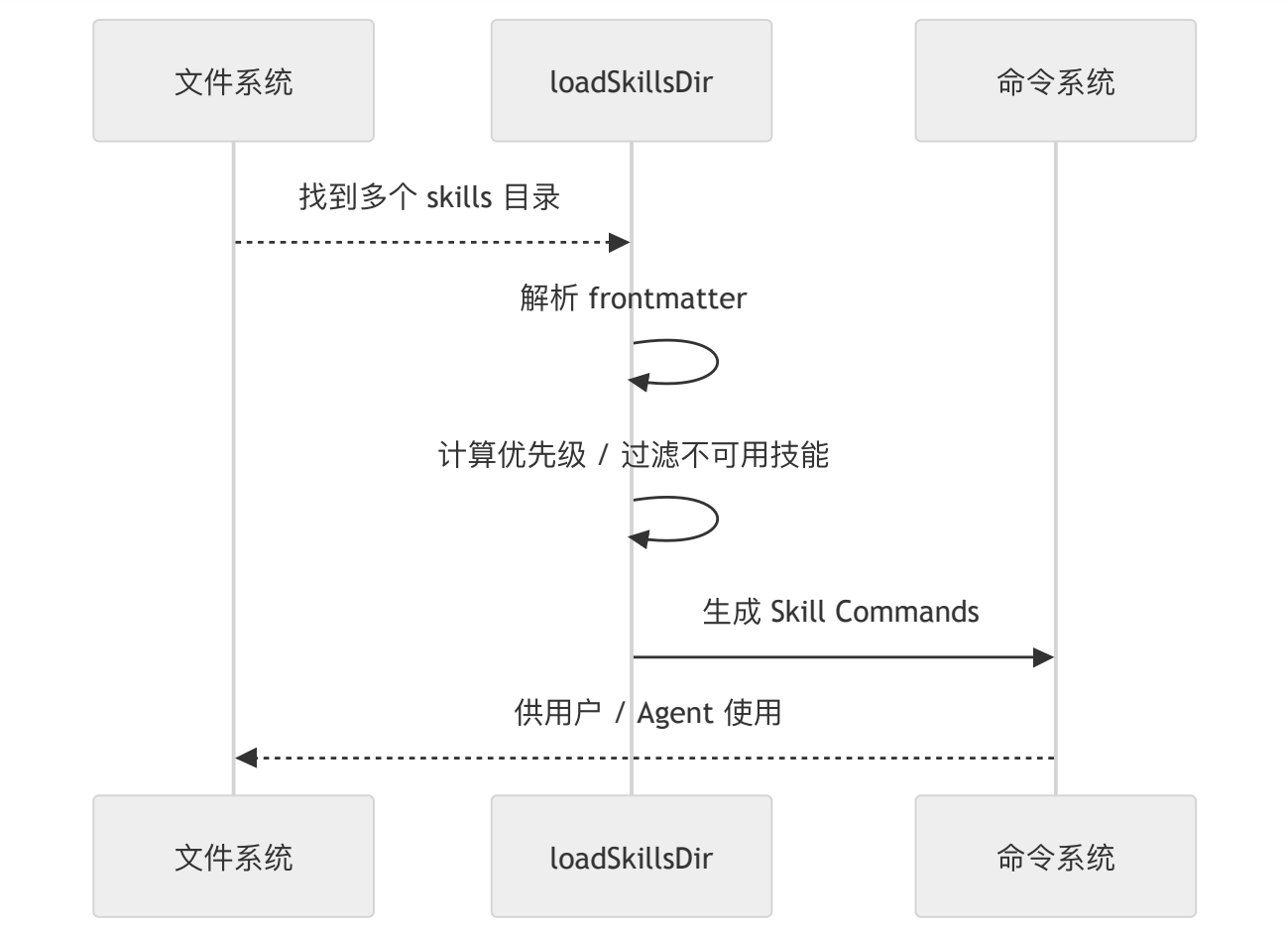


从写书角度，这一层特别值得强调：Claude Code 把“经验与流程”也当成可编排对象，而不仅仅把“代码和命令”当成对象。

28.2 技能发现机制透露了很强的项目感知能力

`loadSkillsDir.ts` 不只是读一个固定目录。它还会：

- 递归寻找 `.claude/skills`
- 按路径深度处理优先级
- 跳过 gitignore 目录
- 根据 project settings 和 plugin-only 规则决定是否启用



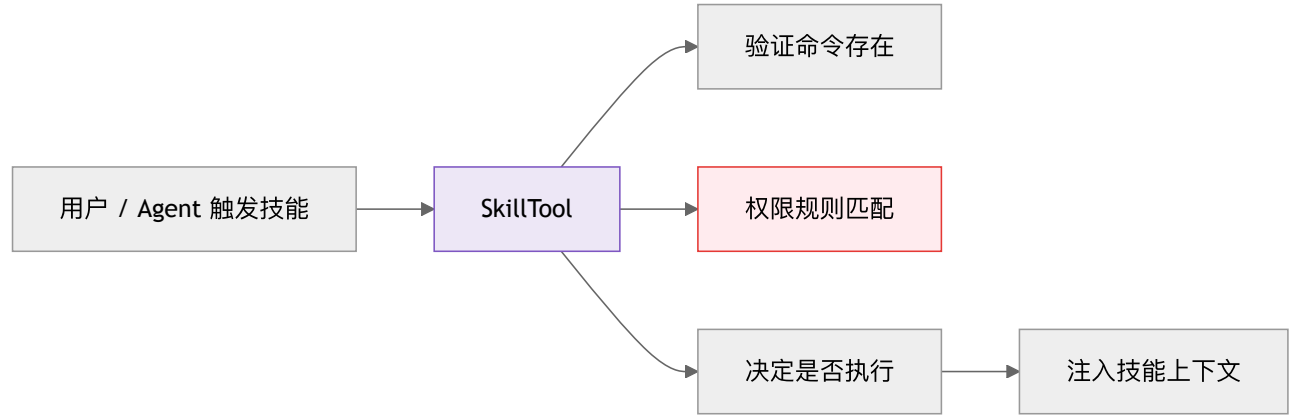
这意味着 Claude Code 的技能不是“全局一锅端”，而是可以跟着项目、目录和上下文变化的。

28.3 `SkillTool` 说明：技能不是纯文本，它也受权限治理

`SkillTool.ts` 做的事情很关键：

- 校验技能命令是否存在
- 检查 deny / allow 规则
- 对远程 canonical skill 走自动放行等特定逻辑

也就是说，技能虽然长得像“提示词模板”，但执行时仍然是系统能力的一部分，必须受治理。

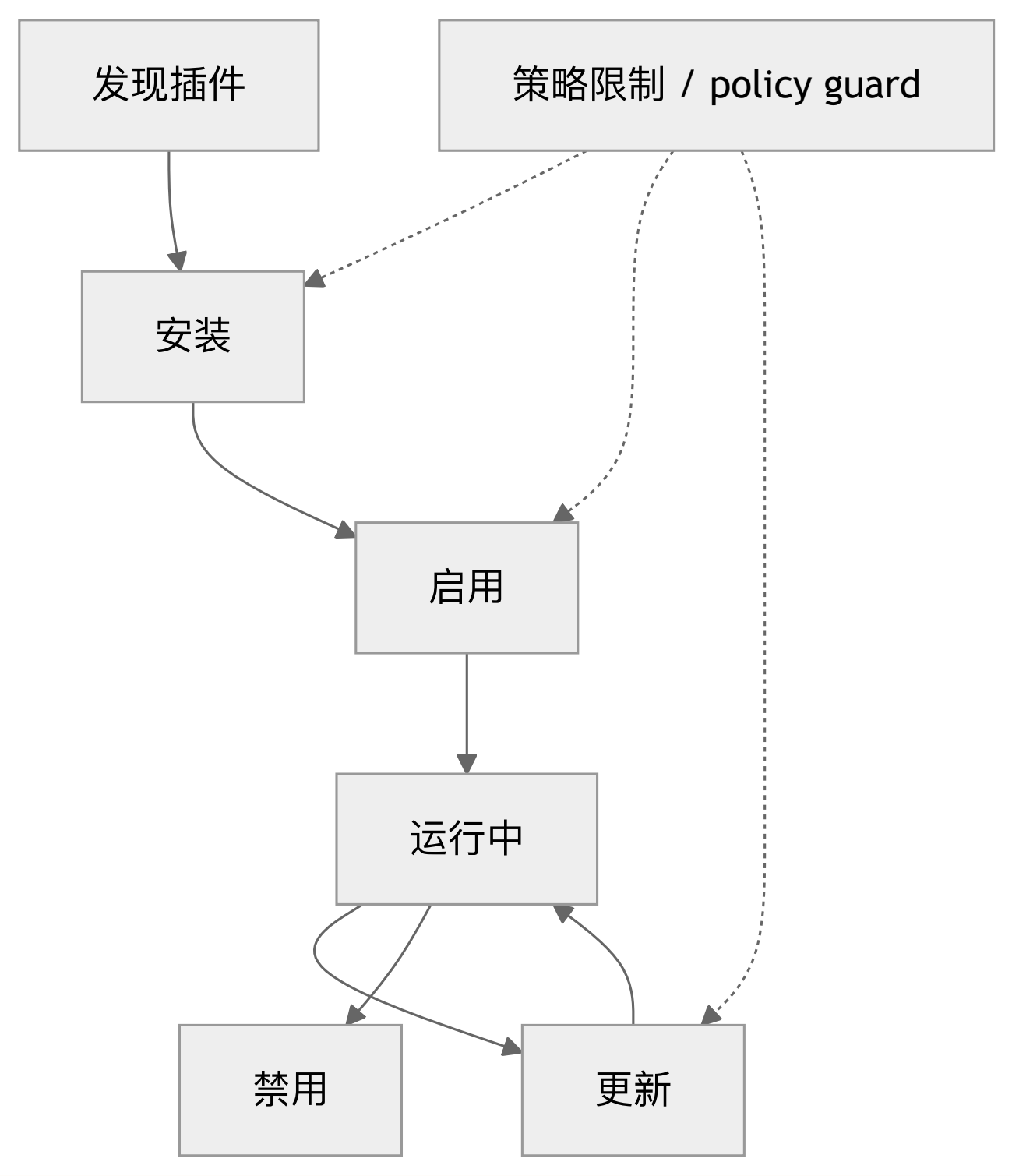


这和很多“提示词市场”最大的不同，就是它不是野生文本拼装，而是纳入了宿主系统的安全模型。

28.4 插件系统关心的，是安装、启用、升级、禁用这些“生命周期”

在 `pluginOperations.ts` 里，插件不是“拖进来就完了”。它有完整的操作面：

- install
- enable
- disable
- update
- 依赖关系处理
- built-in plugin 特殊处理
- policy guard



这说明插件在 Claude Code 里更像“被系统托管的软件包”，而不是“一段直接执行的脚本”。

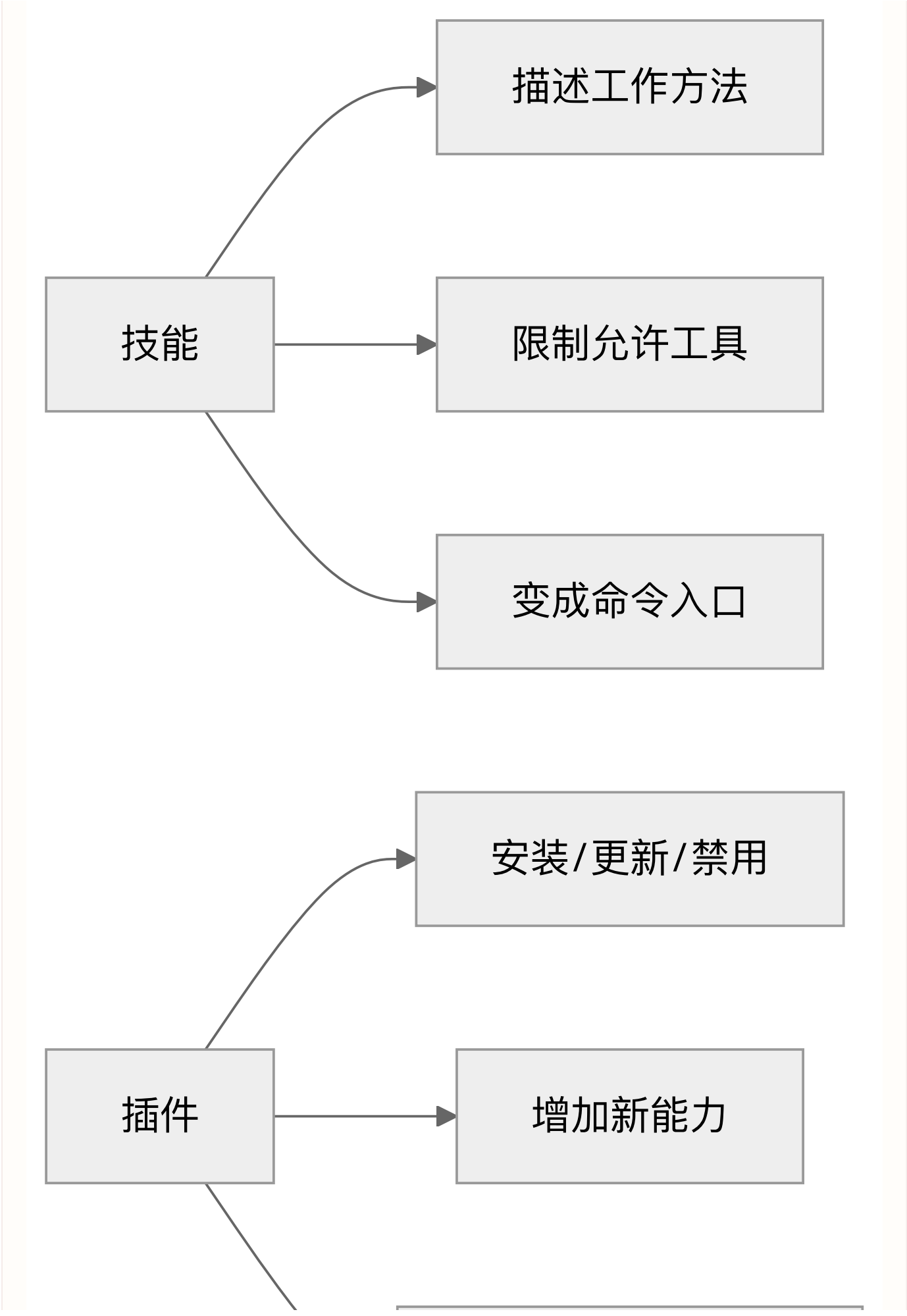
28.5 为什么技能和插件要分两层

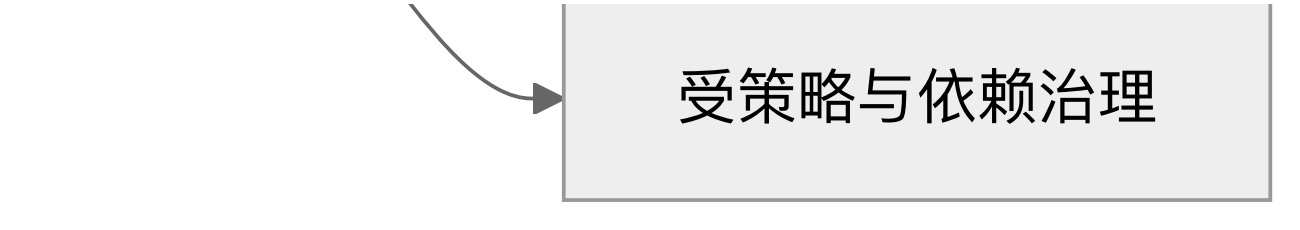
很多读者第一次看会疑惑：既然都有扩展性，为什么不把技能和插件合并？

因为两者解决的问题不同：

层	更像什么	主要解决什么
技能	一张流程卡片	把经验、提示、工作流封装起来
插件	一个功能包	把命令、工具、资源、生命周期接进系统

技能偏轻，适合快速装配工作方法；插件偏重，适合真正扩展能力。


描述工作方法
技能
限制允许工具
变成命令入口
安装/更新/禁用
插件
增加新能力

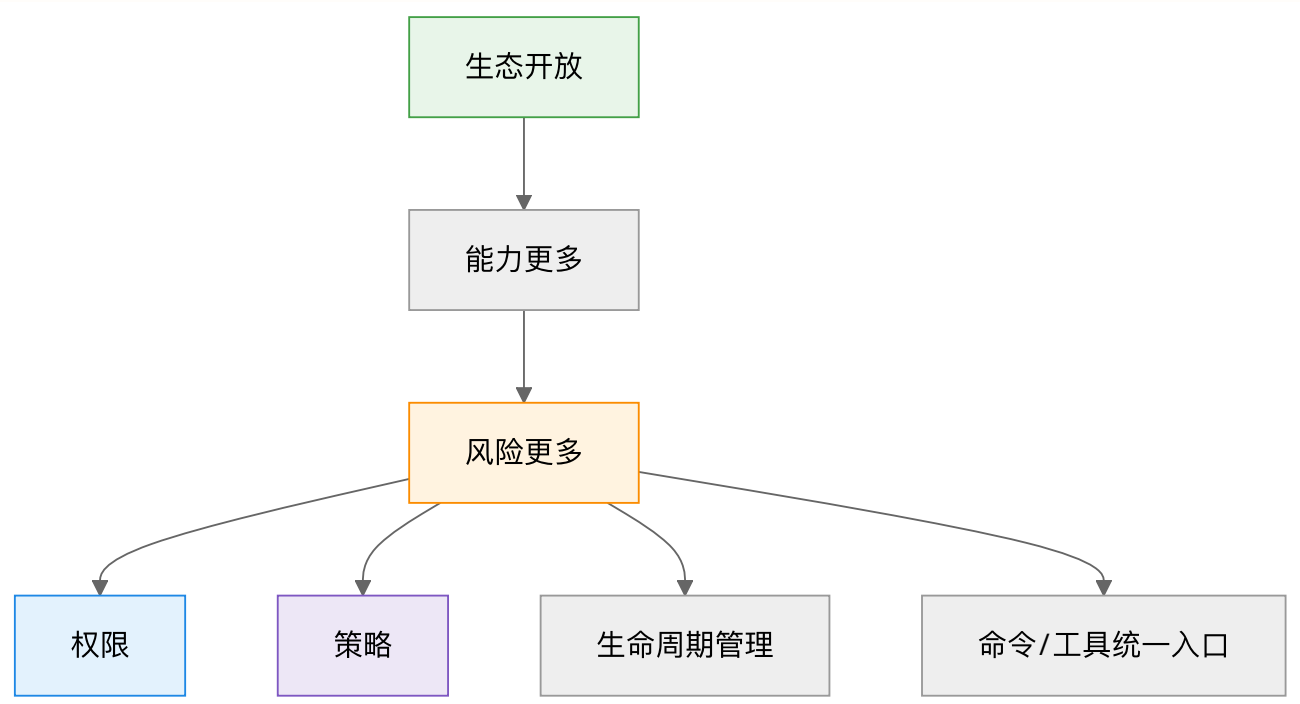


这是一种很清醒的产品拆分：别让所有扩展都变成重型插件，也别让所有扩展都停留在提示词文本。

28.6 设计取舍：生态越开放，越要先想“怎么收口”

Claude Code 在生态层最成熟的一点，是它几乎总在问“扩展之后如何收回控制”：

- 技能能否限制工具
- 插件是否受组织策略约束
- 动态命令是否可隐藏
- 远程技能是否需要特殊放行策略



这让它的开放生态更像“可控扩张”，而不是“把系统大门全拆了”。

🔭 深水区（架构师选读）

技能与插件这一章真正传达的，是 Claude Code 对“知识扩展”和“能力扩展”做了分层：前者用结构化文档，后者用受托管的生命周期单元。很多系统把这两种扩展混在一起，结果要么生态太重，要么治理太弱。Claude Code 的拆分非常值得借鉴。

本章小结

技能负责把工作方法结构化，插件负责把外部能力正规化。两者最终都必须回到命令、工具和权限体系里，才能既开放又不失控。

关键源码索引

- 技能 frontmatter 解析：`loadSkillsDir.ts`
- 技能命令生成：`loadSkillsDir.ts`
- 递归发现 `.claude/skills`：`loadSkillsDir.ts`
- 动态技能启用：`loadSkillsDir.ts`
- SkillTool 校验与权限：`SkillTool.ts`
- 插件安装流程：`pluginOperations.ts`
- 插件启停与更新：`pluginOperations.ts`

逆向提醒

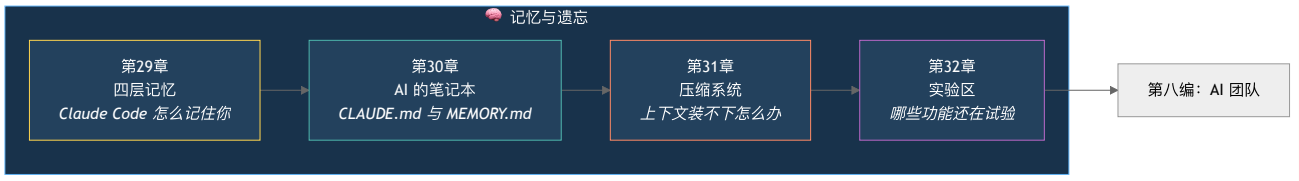
技能目录、插件操作和命令系统在 OpenClaudeCode 中可运行度较高，但第三方插件生态本身并不包含在仓库内。本章讨论的是 Claude Code 如何承载生态，而不是某个具体社区插件的真实质量。

37

第七编：记忆与遗忘

> *一个好助手，不只是“现在听得懂你”，还要“过几天还能接上你的上下文”。*
>
> Claude Code 的记忆系统不是一个大缓存，而是多层结构：**会话内上下文**、**项目级指令**、**持久化记忆目录**、**团队共享记忆**，再加上一整套压缩和实验机制来控制体积与新鲜度。

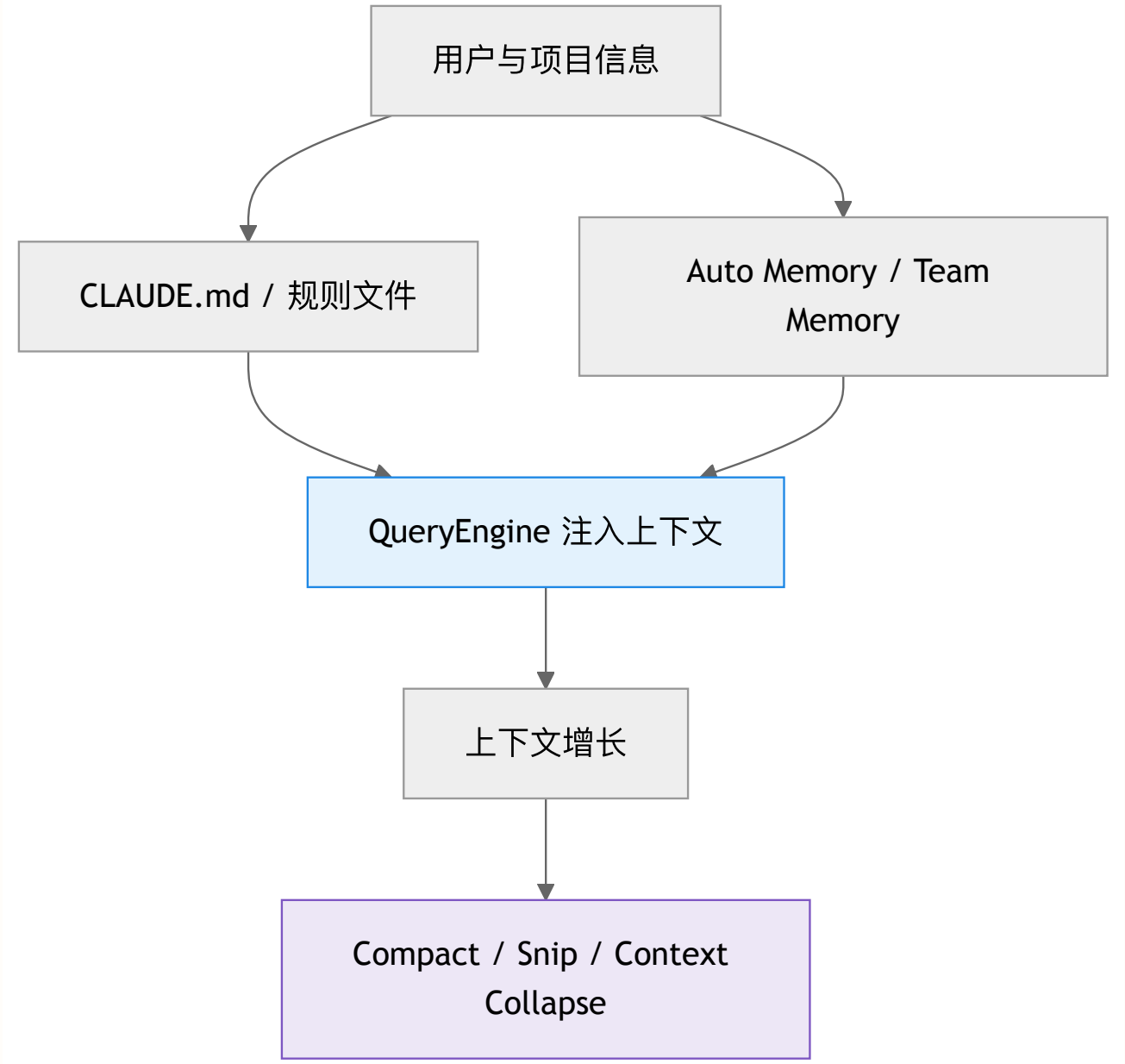
本编总览



本编四章速览

章	标题	核心问题	生活类比
29	四层记忆	关掉终端之后，Claude Code 还记得什么？	人的记忆分层
30	AI 的笔记本	CLAUDE.md、MEMORY.md、DreamTask 各自负责什么？	教科书、笔记本、错题集
31	压缩系统	上下文快满了，怎么瘦身又不丢重点？	塞满的行李箱
32	实验区	哪些代码已经上线，哪些还只是概念车？	车展上的概念车

你会在这一编看到什么



本编阅读目标

读完这一编，你会明白 Claude Code 的“记忆”不是魔法，而是一套文件化、可注入、可压缩、可同步、也会过期的工程系统。

38

Memory 第七编

第29章：四层记忆：Claude Code 怎么“记住”上下文

生活类比：人的四层记忆

我们脑子里同时有几层记忆：眼前正在处理的、今天刚学到的、长期记住的、团队共享的常识。Claude Code 也不是只有“一个上下文窗口”，而是有多层记忆一起工作。

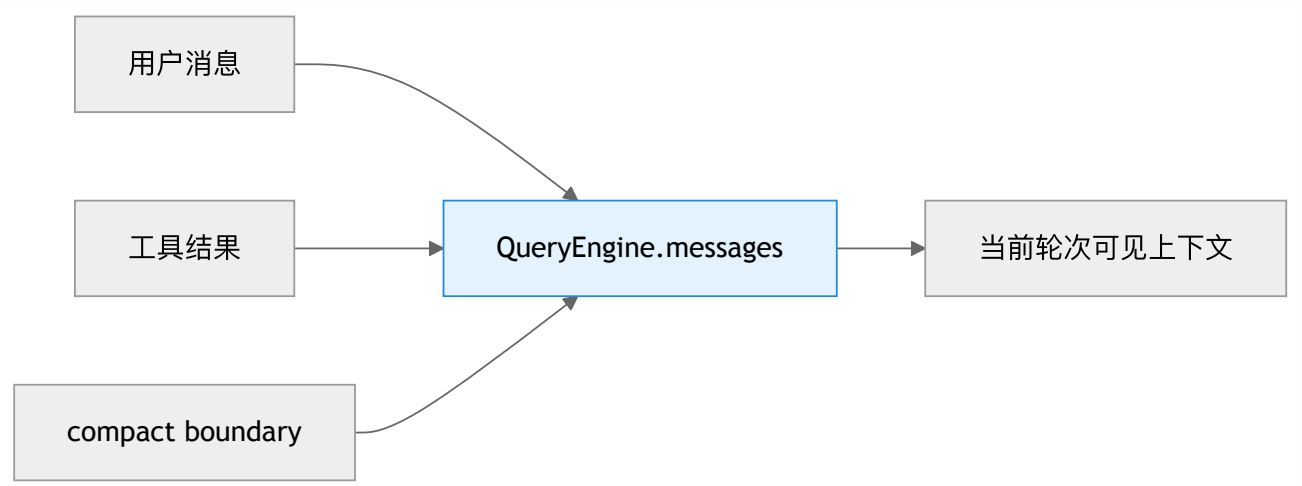
这一章先回答一个问题

你和 Claude Code 聊了一下午，它知道你的项目、习惯和团队规则。关掉终端之后，这些东西到底哪些会留下，哪些会消失？

最短的答案是：**不是所有信息都该存成同一种记忆**。Claude Code 把“现在要用的”“项目长期规则”“个人长期偏好”“团队共享共识”分到了不同层里。

29.1 第一层：会话内记忆，活得最短却最鲜活

QueryEngine 自己就维护着一大块会话状态：消息数组、工具结果、压缩边界、恢复信息、继续执行条件。这一层最像人的“工作记忆”。



这层的特点是：

- 更新最快
- 对当前任务影响最大
- 但生命周期最短

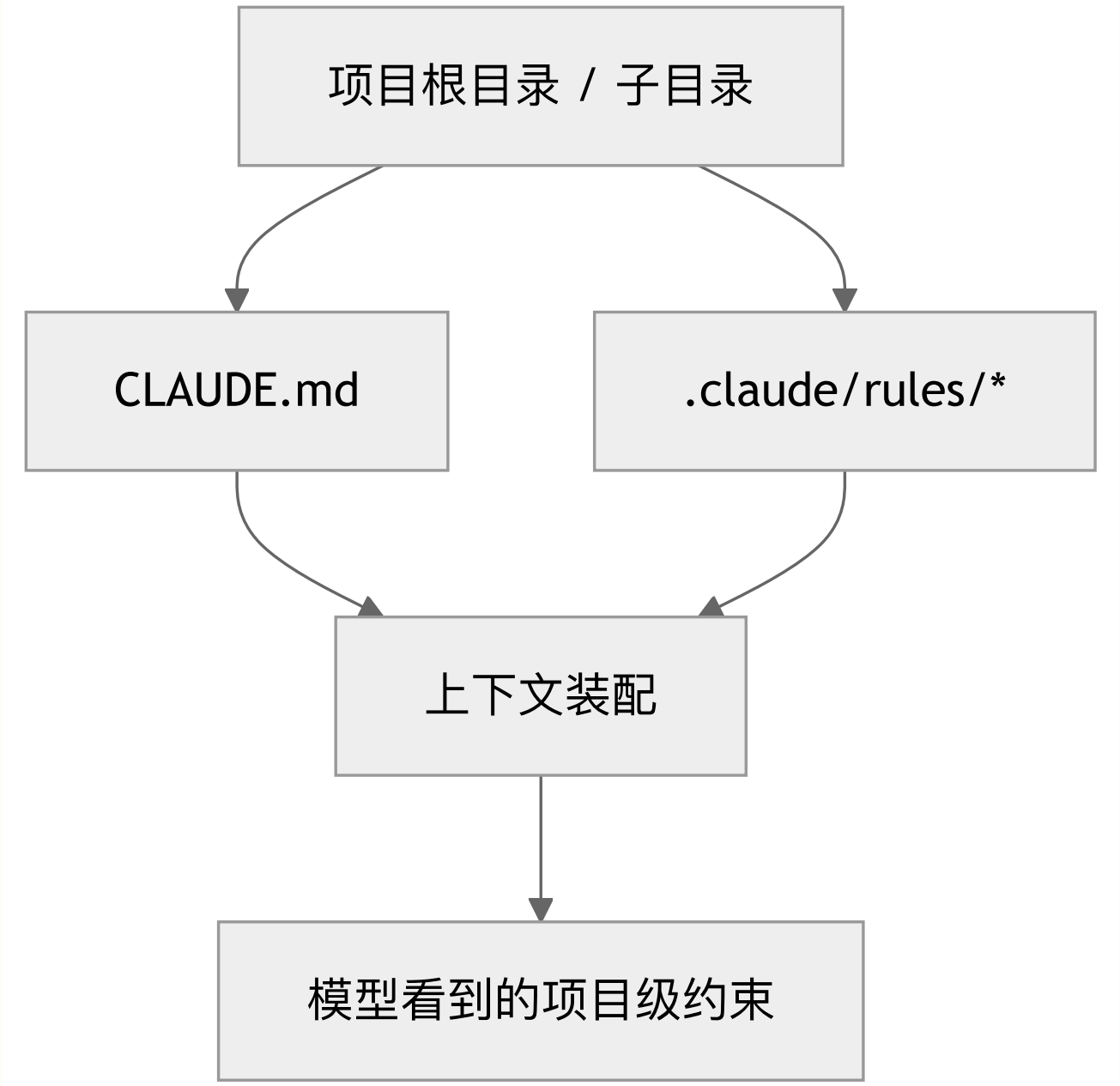
一旦会话结束，它就不再是“天然存在”的记忆了，除非被写进别的层。

29.2 第二层：项目级记忆，最像“你们项目的教科书”

项目里的 `CLAUDE.md` 和规则文件，承担的是“这个仓库长期有效的工作方式”。

它和普通聊天历史不同：

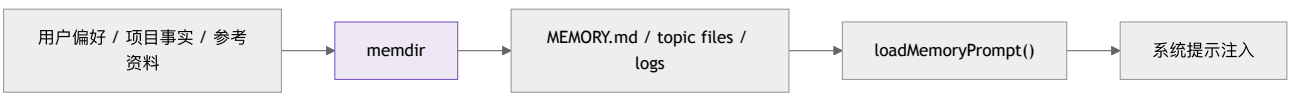
- 不是这轮对话里临时冒出来的
- 通常由项目维护者或使用者刻意维护
- 每次进入项目时会被重新装配进上下文



从设计思想上说，这层是在解决“**别让 AI 每次重新学项目文化**”的问题。

29.3 第三层：Auto Memory，最像“用户自己的长期笔记”

真正持久化的个人记忆，主要落在 `memdir` 这一层。`paths.ts` 和 `memdir.ts` 会决定自动记忆目录在哪里、是否启用、怎么生成统一的 memory prompt。



这层和 `CLAUDE.md` 的区别在于：

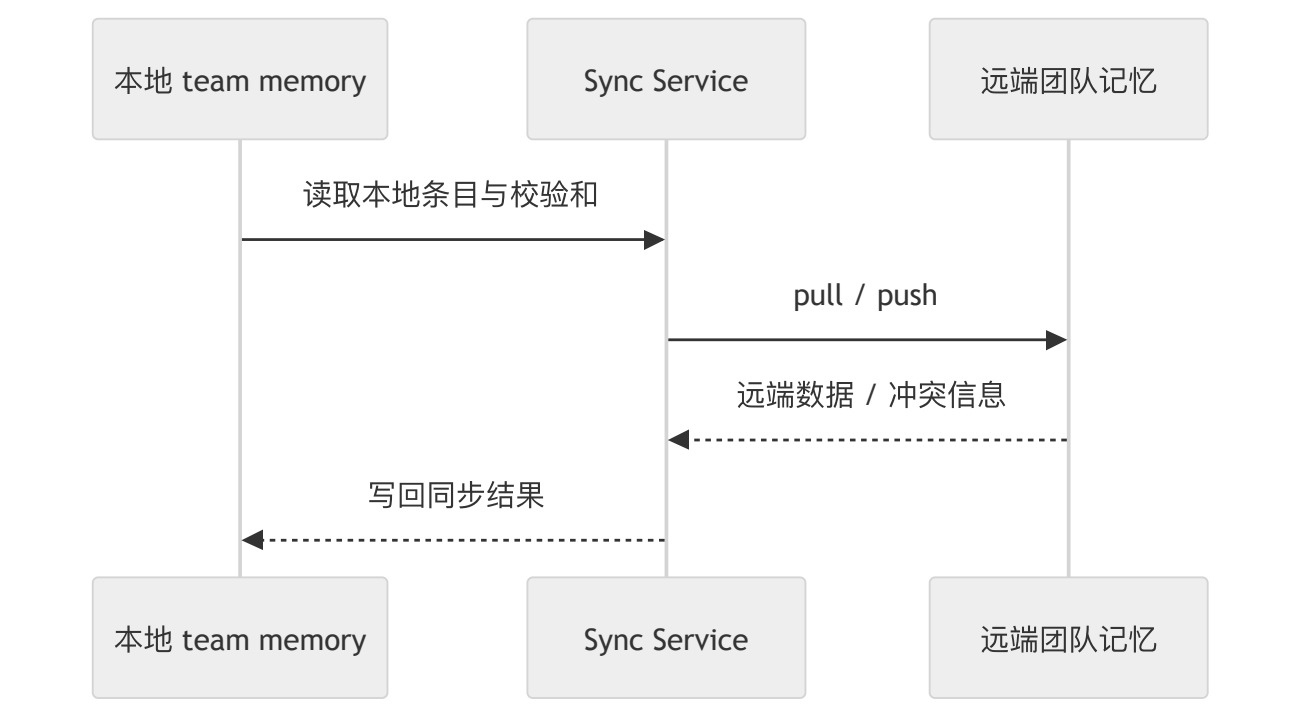
- `CLAUDE.md` 更像项目明规则
- auto memory 更像个人与项目互动中沉淀出来的长期经验

所以它们不是替代关系，而是互补关系。

29.4 第四层：Team Memory，让“团队共识”不再只活在人脑里

再往上一层，就是 team memory。源码里能看到：

- `teamMemPaths.ts` 约束团队记忆目录
- `teamMemorySync/index.ts` 负责与服务端同步
- secret guard 负责阻止把敏感信息写进共享记忆

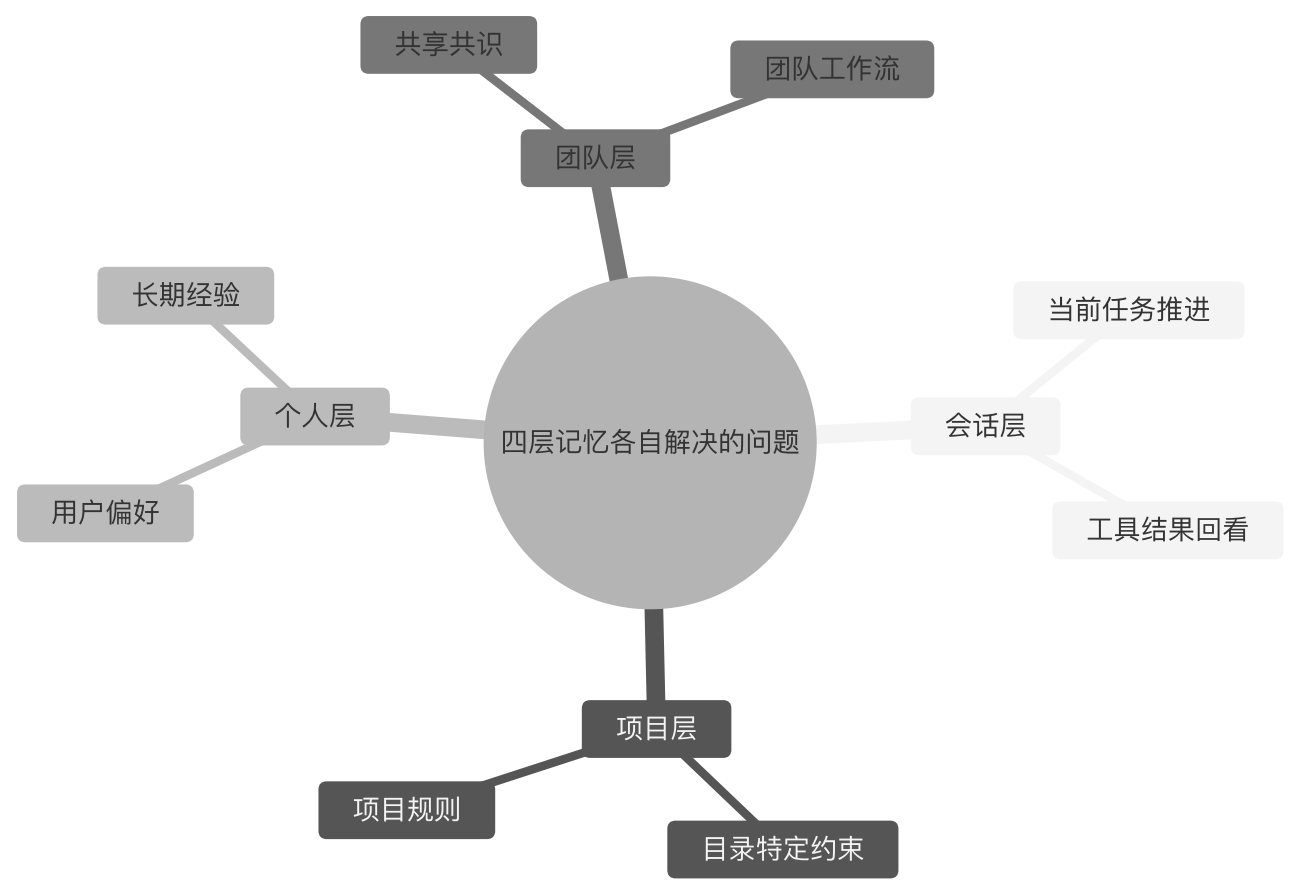


这一层解决的是单人记忆无法解决的问题：**团队规则、项目共识、共享背景知识要能被多人共同维护和读取。**

29.5 四层为什么不能合成一层

如果把所有东西都扔进一个大文件里，会立刻遇到四个问题：

1. 短期消息会污染长期记忆
2. 团队共识会被个人偏好淹没
3. 项目规则和聊天废话混在一起
4. 压缩时很难知道该删什么、不该删什么



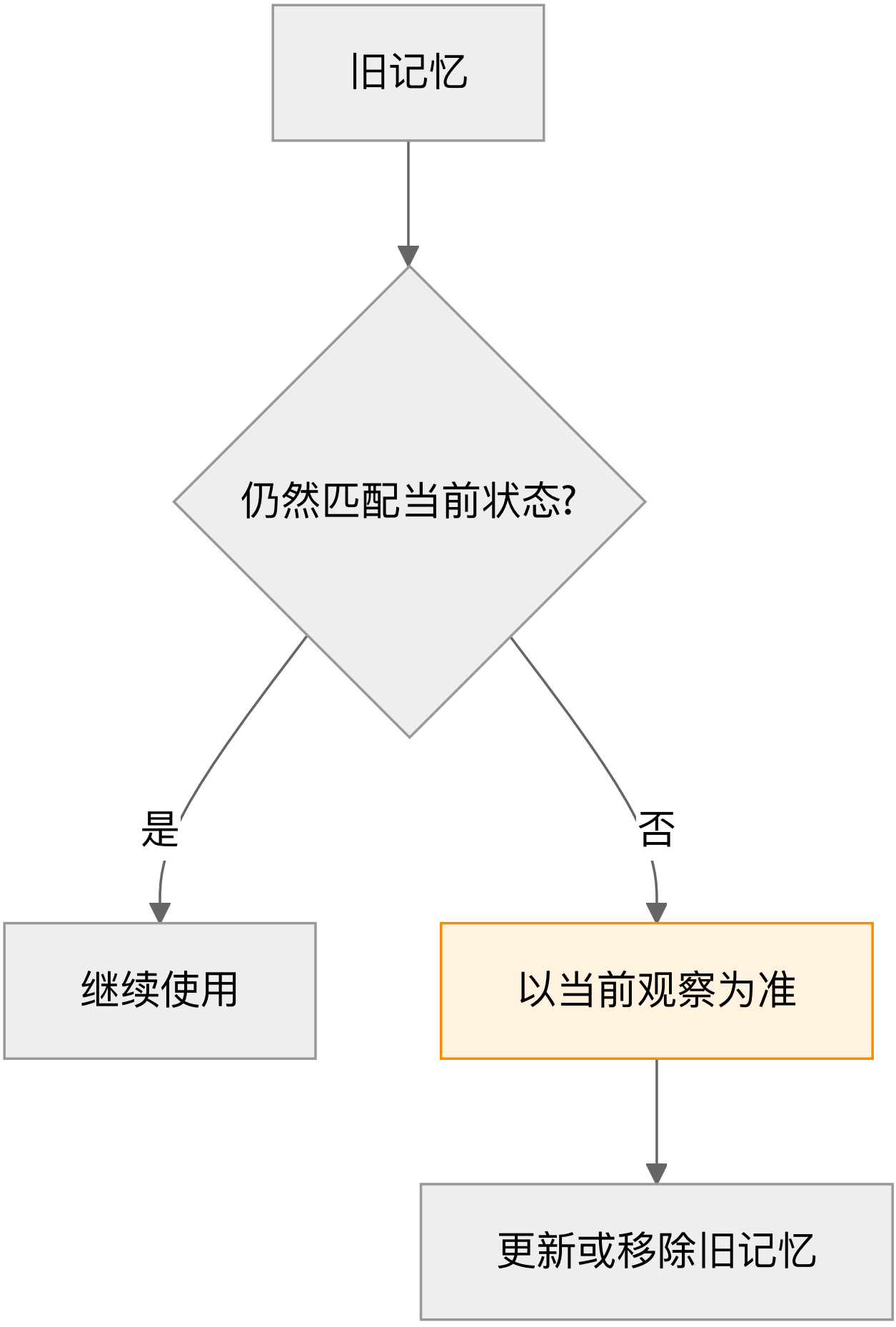
分层的核心价值，不是“更复杂”，而是“每层都能独立演化”。

29.6 设计取舍：记忆最难的不是保存，而是避免陈旧

`memoryTypes.ts` 和 `memoryAge.ts` 里有很鲜明的思路：记忆不是“写进去就永远可信”。它会老，会过期，会和当前文件状态冲突。

所以 Claude Code 的记忆设计强调两点：

- 记忆是辅助上下文，不是真理
- 真要依赖它做判断，最好再读一次当前状态



这让它更像一个“会提醒你去复核的记忆系统”，而不是“把过去写死的数据库”。

深水区（架构师选读）

这章最值得带走的思想是：Agent 的记忆不该只有一个层次。会话、项目、个人、团队这四层各自有不同的更新频率、可信度和作用域。把它们混在一起，只会让系统越来越难维护，也更难压缩与治理。

本章小结

Claude Code 的记忆不是一个大缓存，而是四层协作系统：会话层负责当前任务，项目层负责规则，个人层负责长期经验，团队层负责共享共识。

关键源码索引

- QueryEngine 注入 memory prompt：`QueryEngine.ts`
- QueryEngine 读取 memory prompt：`QueryEngine.ts`
- Auto memory 开关与路径：`paths.ts`
- auto memory 目录解析：`paths.ts`
- 统一 memory prompt 构建：`memdir.ts`
- team memory 路径与边界：`teamMemPaths.ts`
- team memory 同步服务：`index.ts`

逆向提醒

记忆系统的文件化部分在仓库里很清晰，但“哪些记忆最终会被模型采纳”仍然取决于提示装配和模型行为。本章分析的是系统如何提供记忆，不是模型是否一定会完美使用这些记忆。

39

CLAUDE.md Memory 第七编

第30章：AI 的笔记本：CLAUDE.md 与记忆目录

生活类比：教科书、笔记本、错题集

学习不是把所有内容写在一个本子上。教科书负责稳定知识，笔记本负责自己的理解，错题集负责反复提醒。Claude Code 的 `CLAUDE.md`、`MEMORY.md`、typed memory、DreamTask，也在做类似分工。

这一章先回答一个问题

为什么 Claude Code 不把所有“记忆”都塞进 `CLAUDE.md`？为什么还要有 memdir、typed memory、DreamTask 这些看起来更复杂的东西？

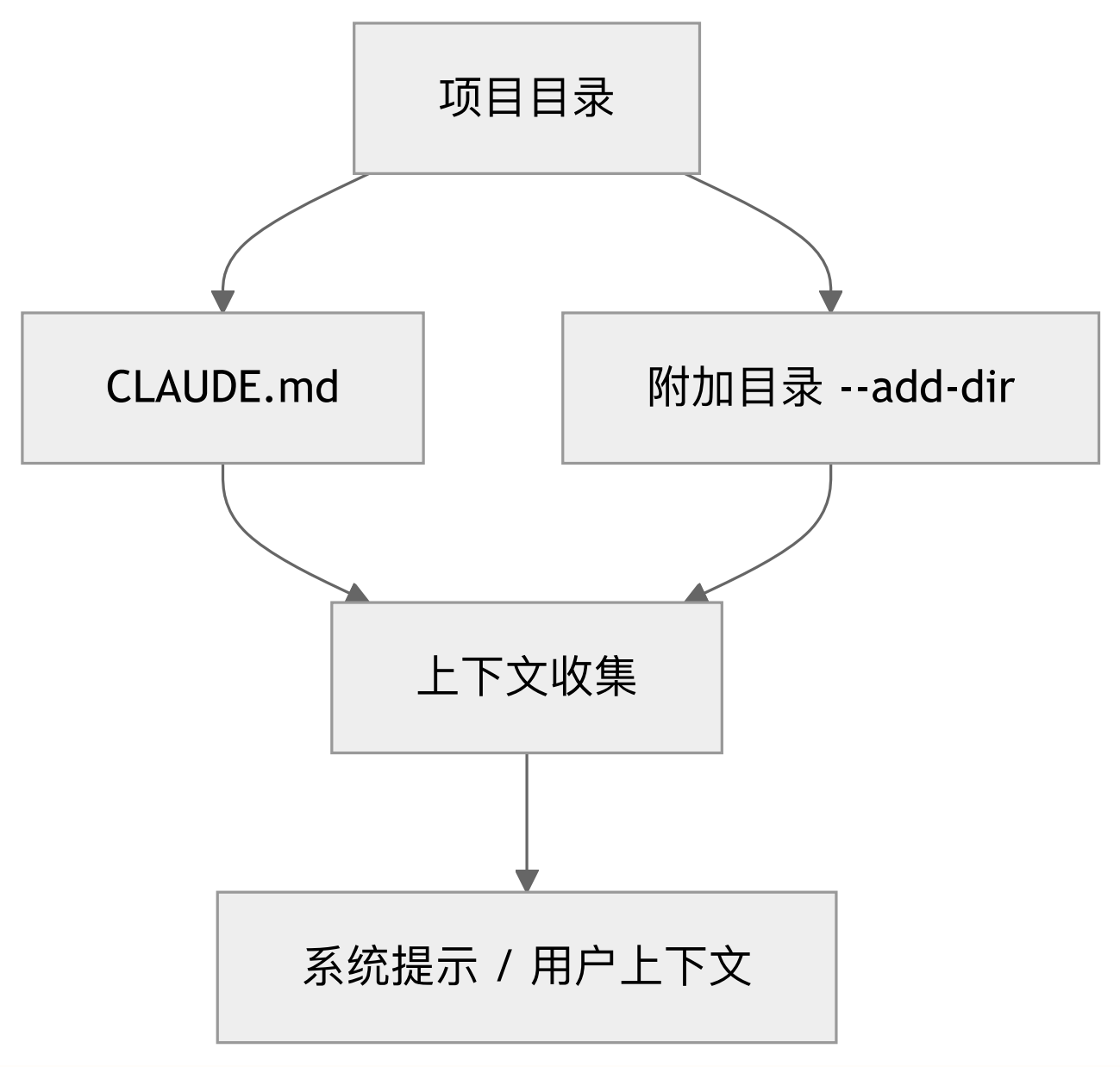
因为它要同时解决四件事：**稳定规则、长期经验、可检索结构、自动整理**。一个文件很难同时把这四件事都做好。

30.1 `CLAUDE.md` 解决的是“规则稳定传达”

`CLAUDE.md` 最像项目教科书。它适合放：

- 长期有效的开发规则
- 项目结构说明
- 高价值的固定约束

源码里能看到，`main.tsx` 和 `bootstrap/state.ts` 都把它当成正式上下文输入的一部分，还支持 `--add-dir` 扩展额外目录。



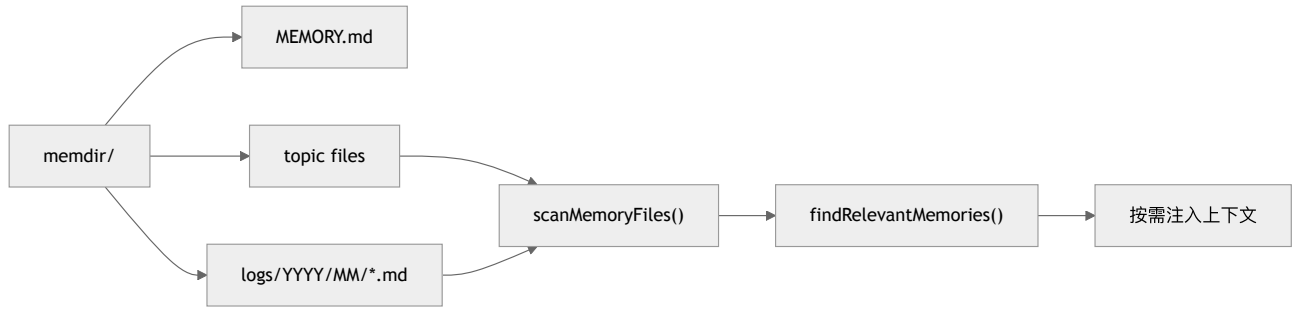
它的优点很明确：

- 人类可直接编辑
- 项目成员容易达成共识
- 适合作为“默认规范”

但它不适合承载太多高频变化信息。

30.2 `memdir` 解决的是“把记忆组织成目录，而不是一锅粥”

`memdir.ts`、`memoryScan.ts`、`findRelevantMemories.ts` 这一组文件，体现的设计思想是：记忆不该只是一个大文本块，而应该是**一组可扫描、可分类、可择优召回的文件**。

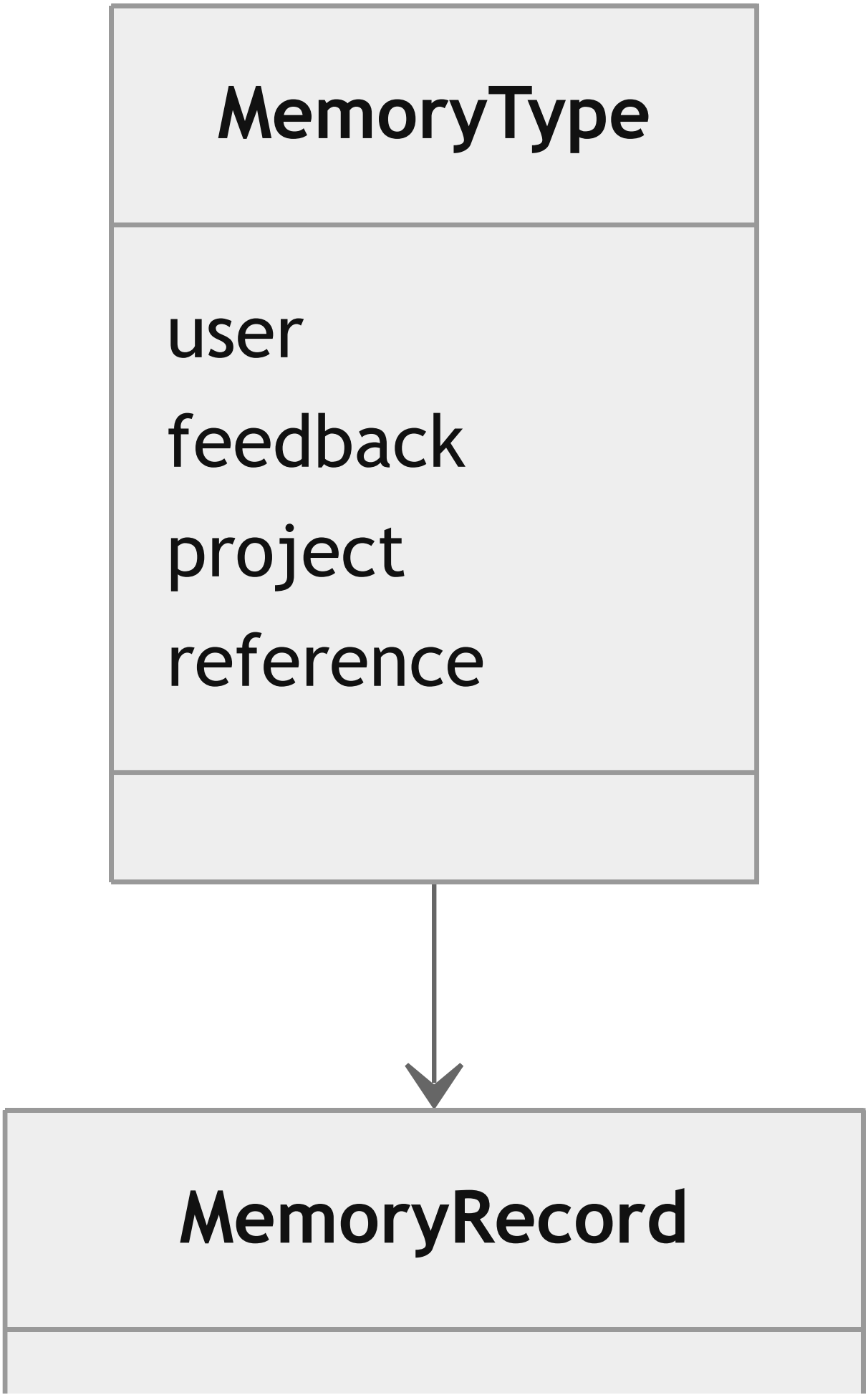


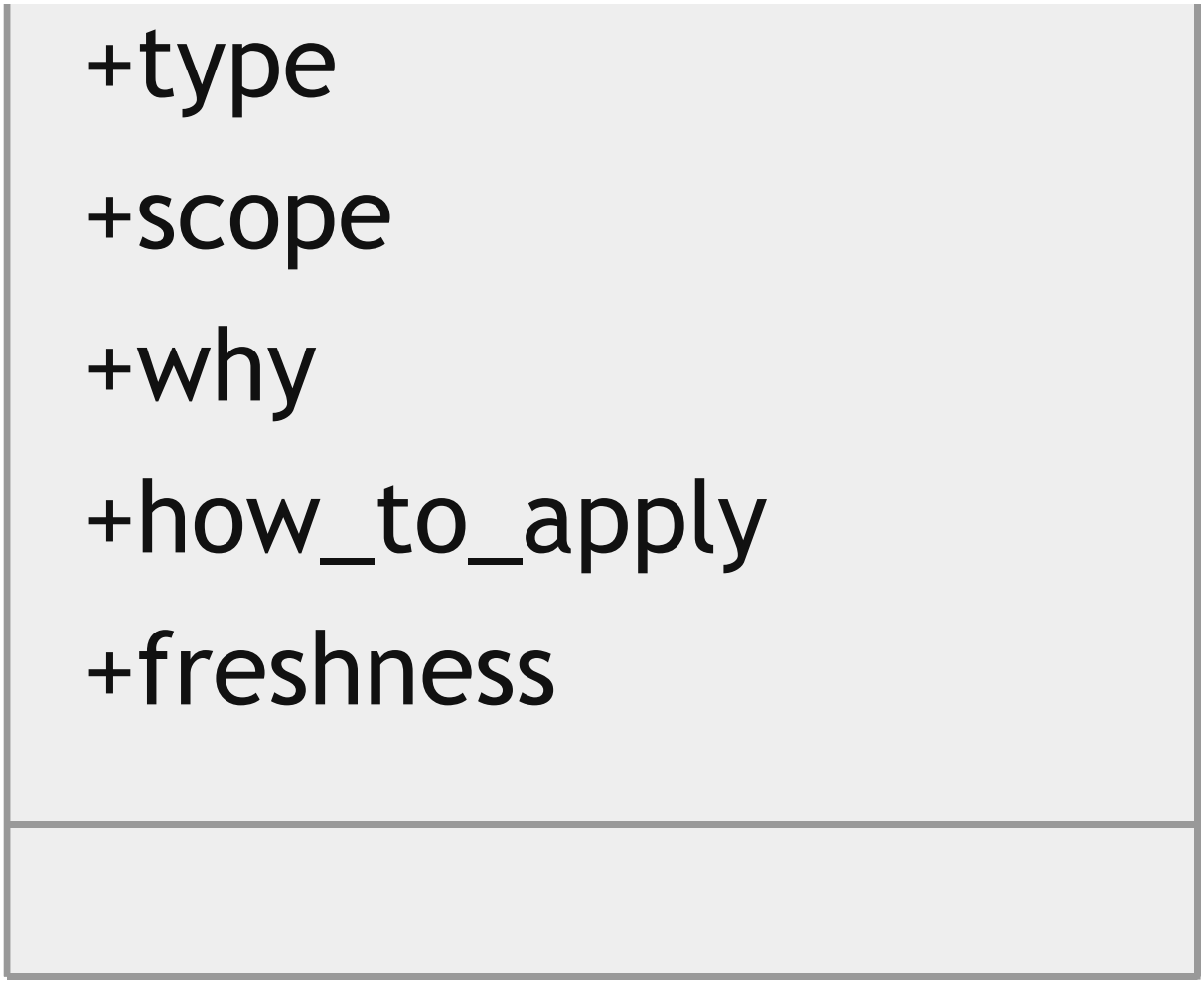
这比“全贴进 prompt”更高级，因为系统终于能对记忆做三件事：

- 扫描
- 过滤
- 选择性回忆

30.3 typed memory 的价值，是让系统知道“这条记忆属于哪一类”

`memoryTypes.ts` 里明确枚举了不同 memory type，并给出保存建议、示例和不该保存的内容。这一点非常像给 AI 写了一份“记忆写作规范”。


MemoryType
user
feedback
project
reference
MemoryRecord



一旦有了类型，系统才能更聪明地处理：

- 哪些适合长期保留
- 哪些更容易过时
- 哪些适合 team scope
- 哪些只是个人偏好

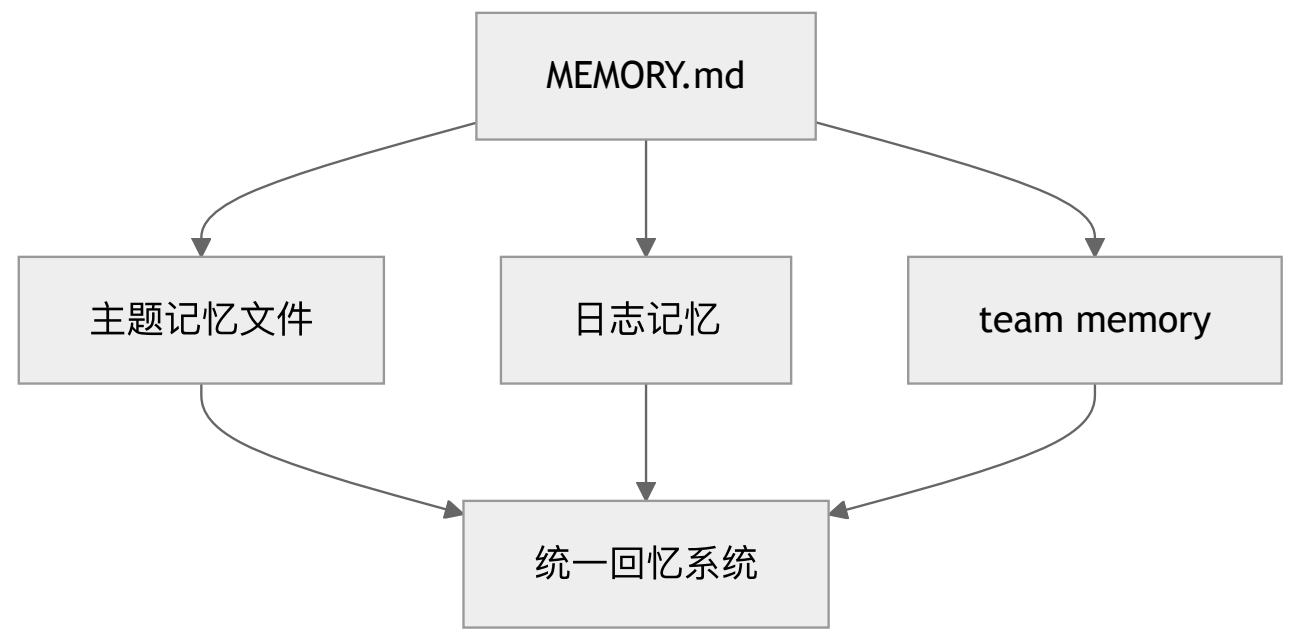
这比“全凭一句自然语言描述”稳得多。

30.4 MEMORY.md 不是唯一记忆文件，而像目录入口

很多初学者会误以为 MEMORY.md 就是全部记忆。其实从实现上看，它更像：

- 入口页
- 索引页
- 汇总页

真正的记忆可以散落在多个主题文件、日志文件和 team memory 文件中。



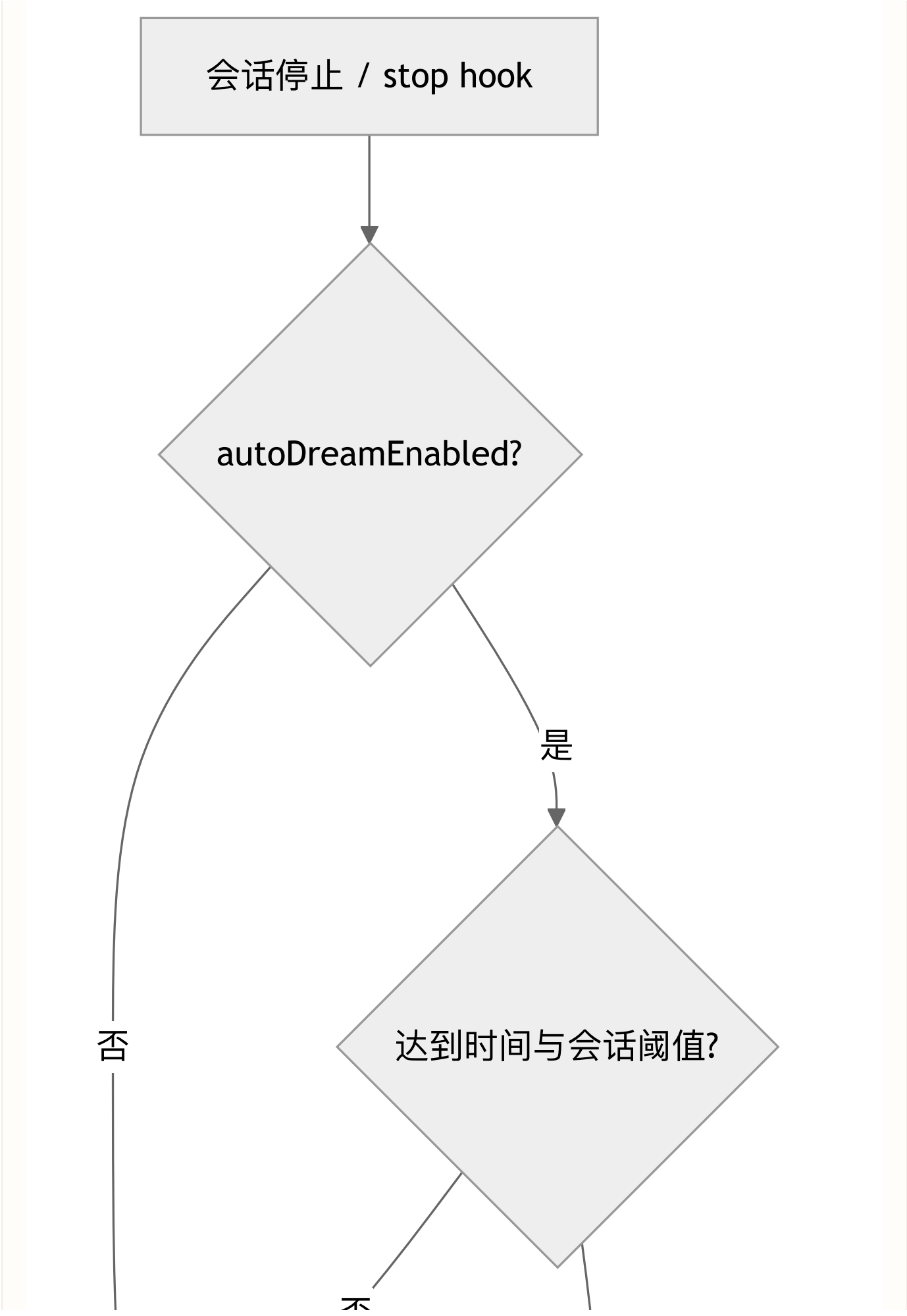
这样做的好处是，记忆可以随着时间不断增长，但不会立刻把一个文件撑爆。

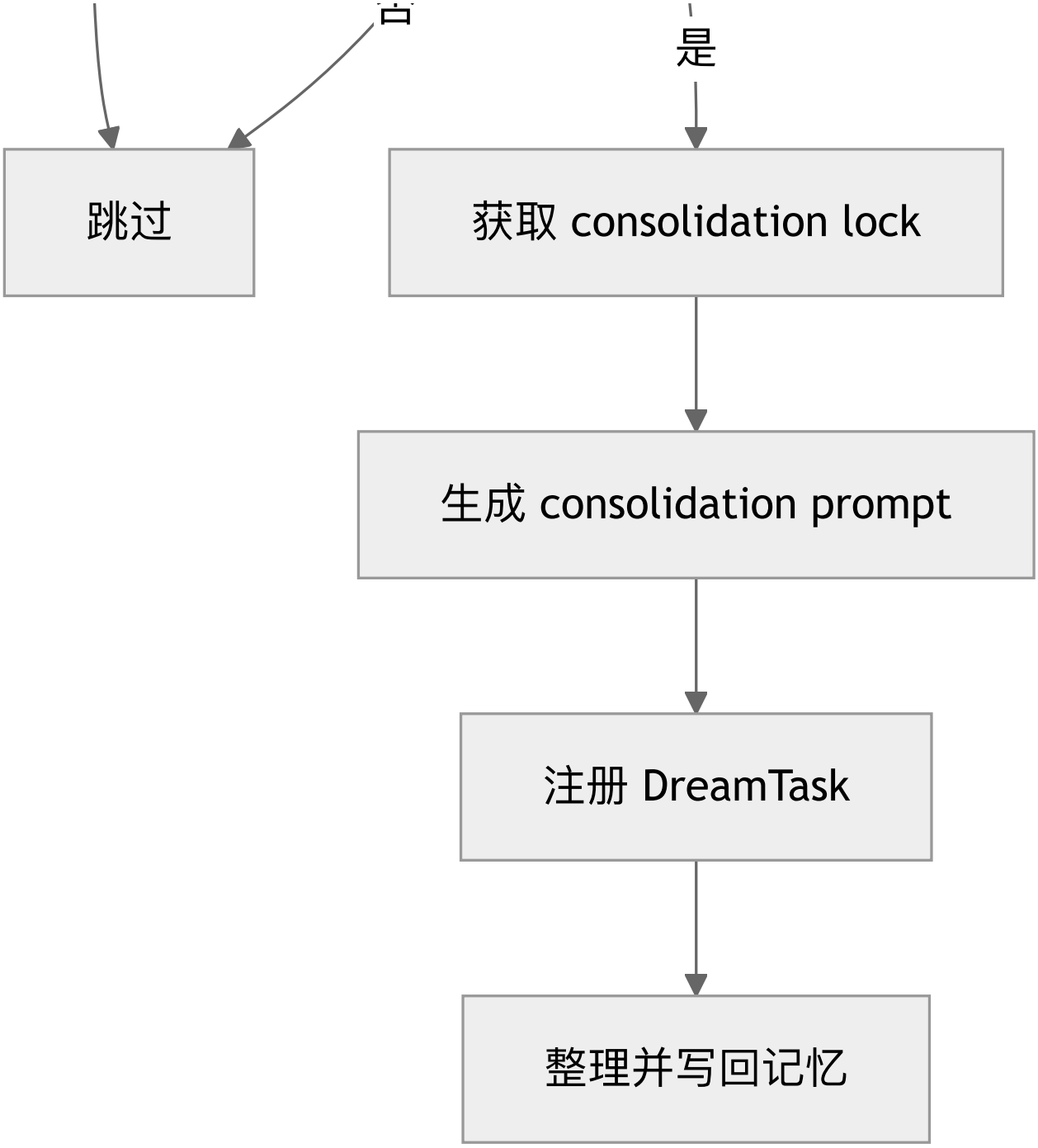
30.5 DreamTask 与 autoDream：系统开始自己整理笔记

`autoDream.ts` 的命名非常形象。它不是让模型“做梦”，而是让系统在合适时机自动做记忆整合。

从源码看，它会考虑：

- 是否启用 auto dream
- 是否处在 KAIROS 模式
- 距离上次 consolidation 过去多久
- 最近有没有足够多的会话值得总结
- 是否需要加锁防止并发整理

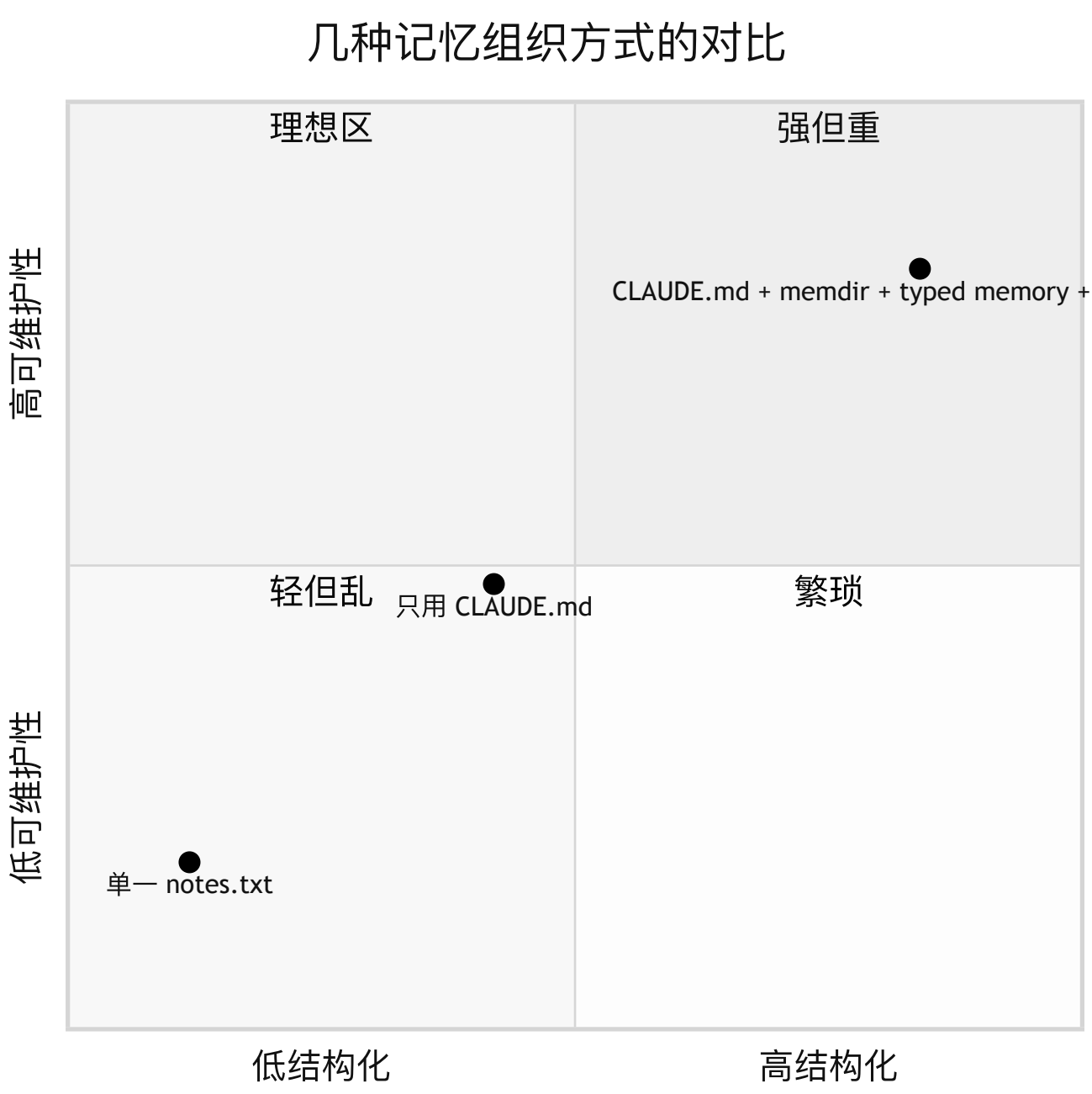

会话停止 / stop hook
autoDreamEnabled?
是
否
达到时间与会话阈值?



这代表一个很重要的转变：Claude Code 的记忆系统已经不是纯手工维护，而开始具备“后台整理”的能力。

30.6 为什么这套系统比一个 `notes.txt` 强太多

如果只用一个文本文件，当然也能“记东西”。但 Claude Code 需要的不是“能记”，而是“能长期用”。



Claude Code 之所以复杂，是因为它同时追求：

- 人能读
- 程序能扫
- 模型能用
- 长期不乱

这四个目标同时成立，本来就需要分层设计。

深水区（架构师选读）

`CLAUDE.md` 与 memdir 的组合，体现的是“规则系统”和“经验系统”分离。前者适合稳定显式指令，后者适合持续积累与选择性召回。DreamTask 则开始把“记忆维护”从人工劳动变成后台任务，这是 Agent 产品向长期陪伴型系统演进的重要信号。

本章小结

`CLAUDE.md` 负责稳定规则，memdir 负责结构化长期记忆，typed memory 负责分类与召回，autoDream 负责自动整理。它们分工明确，合起来才像真正可持续的记忆系统。

关键源码索引

- `CLAUDE.md` 额外目录状态：`bootstrap/state.ts`
- 设置额外 `CLAUDE.md` 目录：`bootstrap/state.ts`
- 命令行 `--add-dir` 说明：`main.tsx`

- typed memory 类型定义：`memoryTypes.ts`
- typed-memory prompt 构建：`memdir.ts`
- memory scan：`memoryScan.ts`
- relevant memory 查找：`findRelevantMemories.ts`
- autoDream 开关：`config.ts`
- autoDream 主流程：`autoDream.ts`

逆向提醒

`CLAUDE.md` 的加载、memory prompt 的构建、autoDream 的触发条件都能在源码里看到，但“哪些记忆最终最有用”仍然高度依赖真实使用场景。本章分析的是系统结构，不是某个团队应当如何写记忆的唯一答案。

40

Compact 第七编

第31章：当记忆装不下：压缩系统

生活类比：塞满的行李箱

行李箱快爆了，你通常有三种办法：自己整理、提前清理、快爆时紧急腾空间。Claude Code 的压缩系统，也差不多就是这三套思路。

这一章先回答一个问题

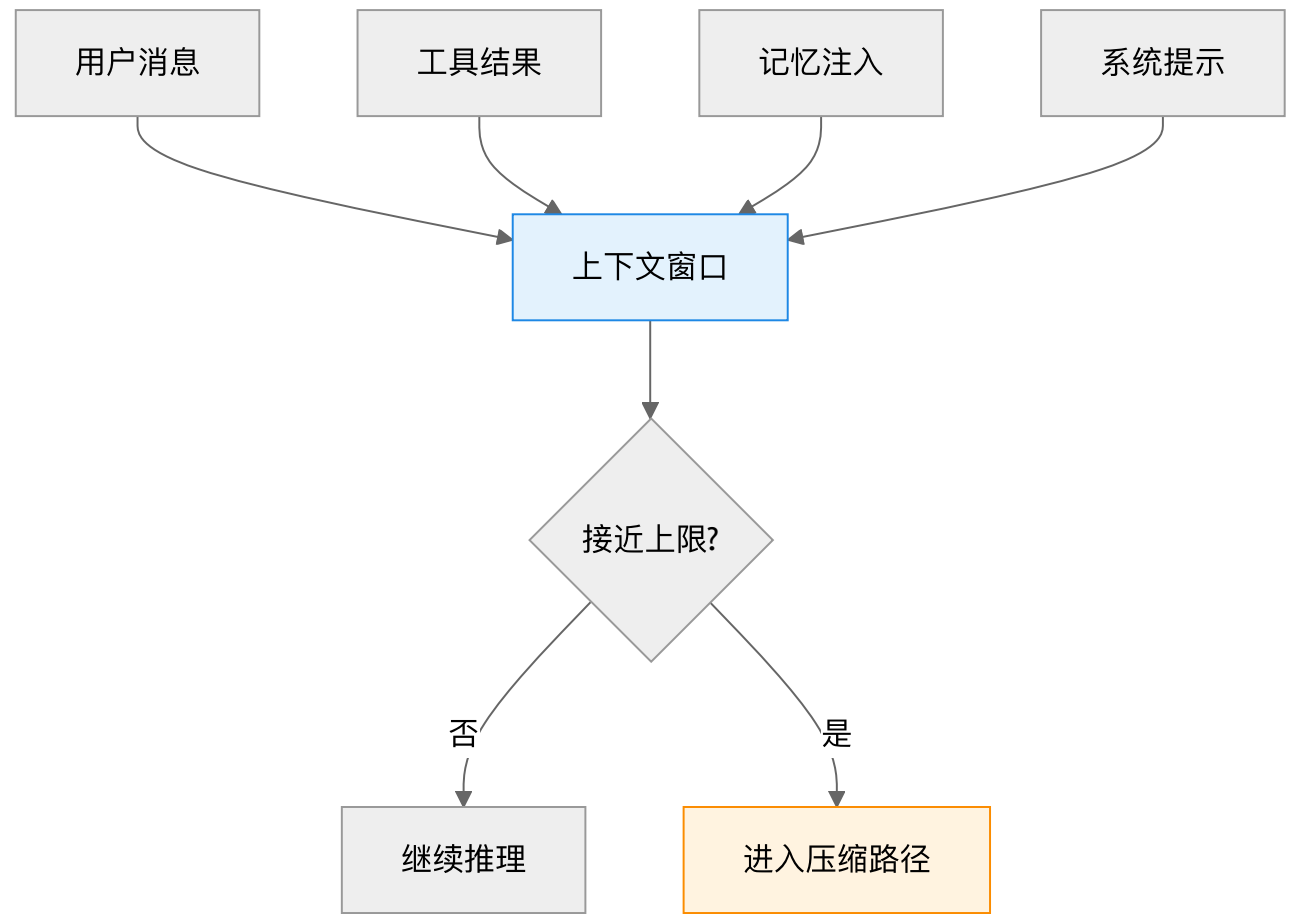
对话越来越长、工具结果越来越多、上下文快超限时，Claude Code 是怎么“瘦身”而不把关键任务线索一起删掉的?

它没有只靠一种压缩，而是做了一个组合拳：**manual compact、auto compact、microcompact、reactive compact、context collapse、snip、tool summary**。

31.1 为什么“上下文压缩”是 Agent 系统的生死线

普通聊天机器人上下文满了，最多就是聊不下去。Claude Code 不一样，它在任务执行中还要保留：

- 用户目标
- 工具结果
- 中间决策
- 文件上下文
- 记忆注入

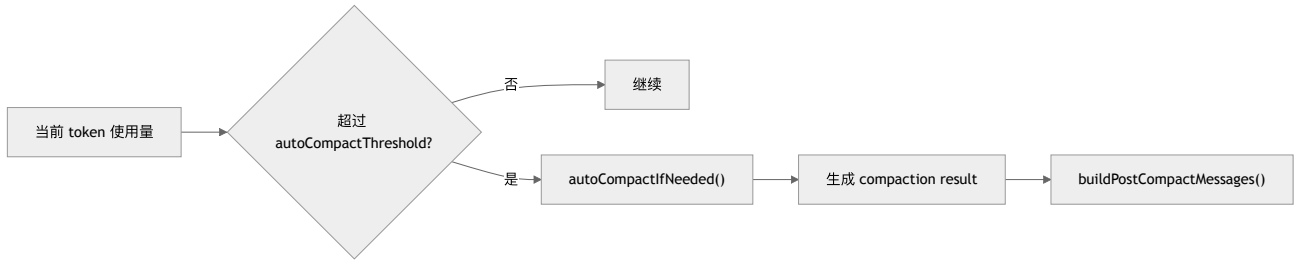


所以压缩不是优化项，而是“能不能继续工作的前提”。

31.2 auto compact：提前减肥，而不是等到完全爆掉

`autoCompact.ts` 会根据模型上下文大小和阈值判断是否需要自动压缩。它的思路很像“提前做容量管理”：

- 先看是否启用
- 计算阈值
- 到了阈值就提前压缩
- 把 tracking 信息带回主循环

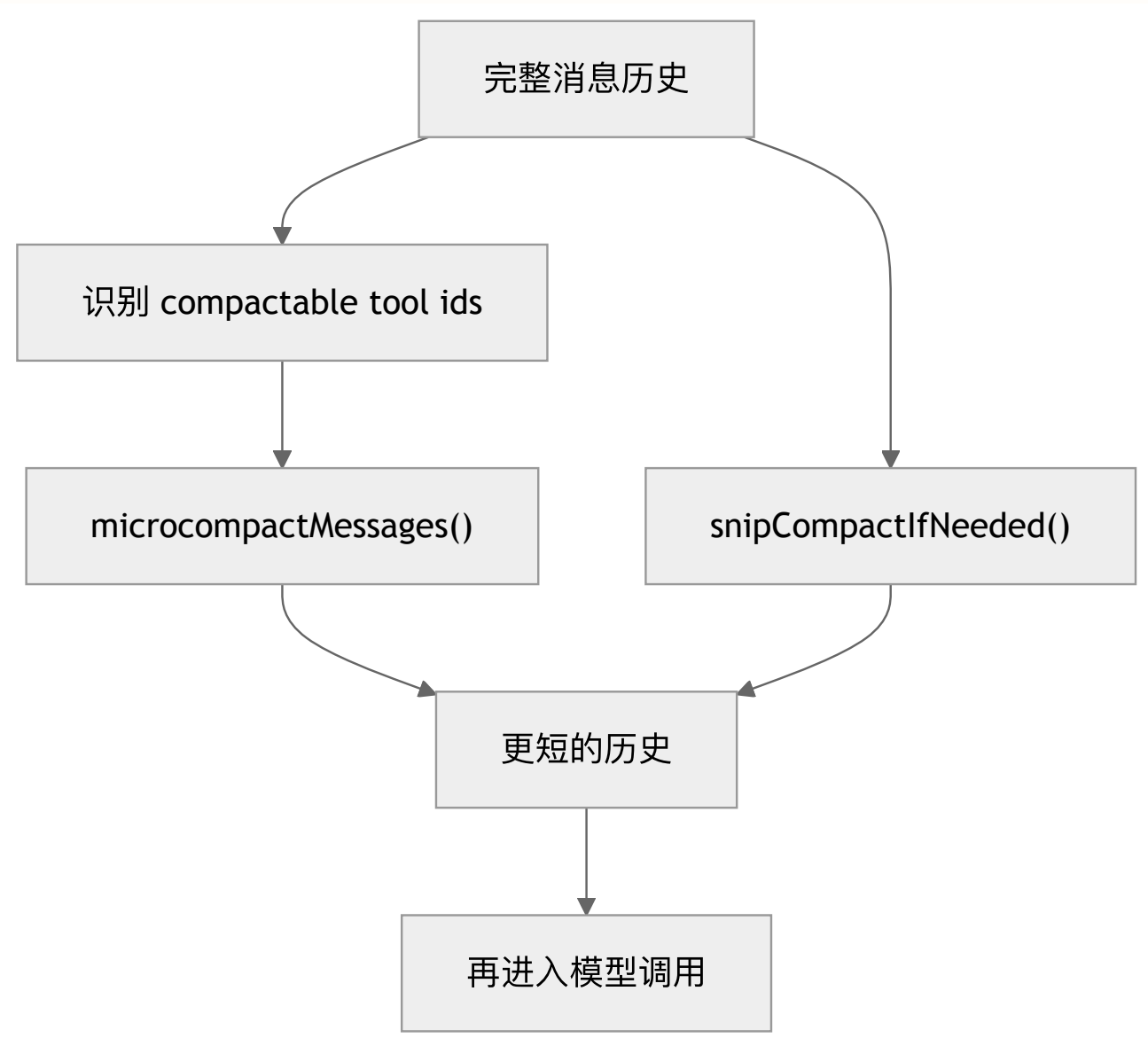


这类压缩的优点，是它通常发生在系统还“比较从容”的时候，所以保真度更高。

31.3 microcompact 与 snip：优先删“最该删的部分”

Claude Code 并不是一上来就对整段历史做大摘要。`microCompact.ts` 和 `snipCompact.ts` 更像精细手术：

- 优先处理旧工具结果
- 尽量保留最近和高价值片段
- 在 API 调用前先做更细粒度瘦身

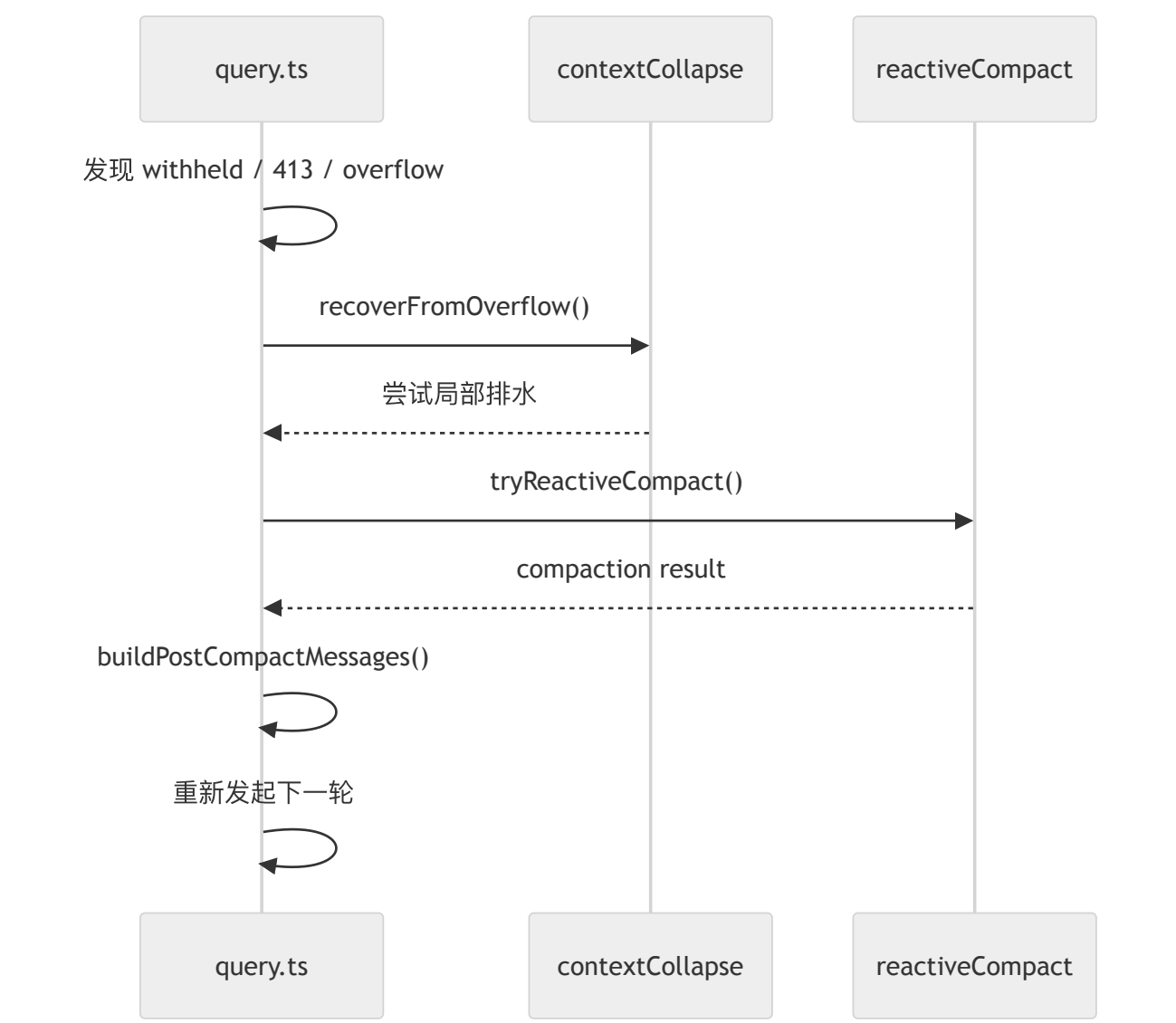


这很重要，因为“粗暴摘要整段对话”太容易把任务骨架也丢掉。微压缩和 snip 的价值，就是先拿掉体积大但信息密度低的部分。

31.4 reactive compact：真的快爆了，才动用紧急策略

在 `query.ts` 里，`reactiveCompact` 和 `contextCollapse.recoverFromOverflow()` 出现在错误恢复链上。也就是说，这些手段更像**应急逃生通道**：

- 上下文已被 withheld
- 或媒体/提示过大
- 普通路径无法继续
- 这时才触发更激进的紧急瘦身

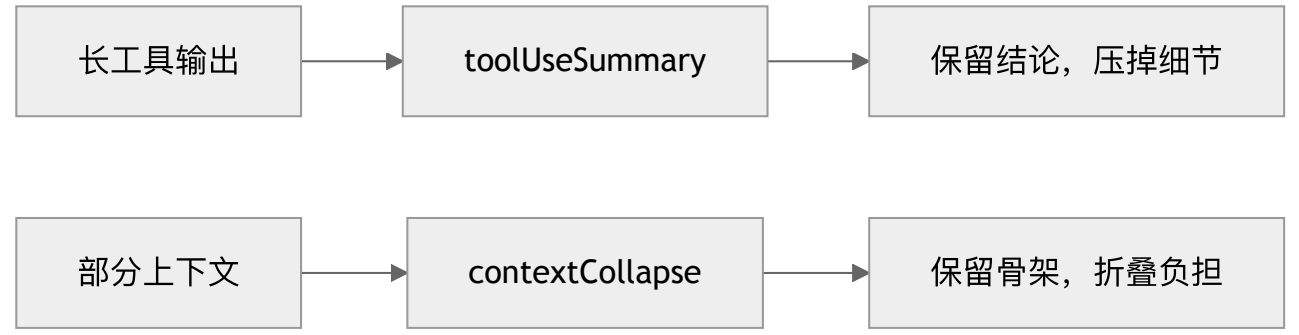


这类压缩保真度通常不如提前压缩，但胜在能救回一次本该失败的对话。

31.5 context collapse 与 tool summary：让“骨架”继续露在外面

有意思的是，Claude Code 并不总是选择“全部压成摘要”。`contextCollapse` 和 `generateToolUseSummary()` 更像在做结构化保留：

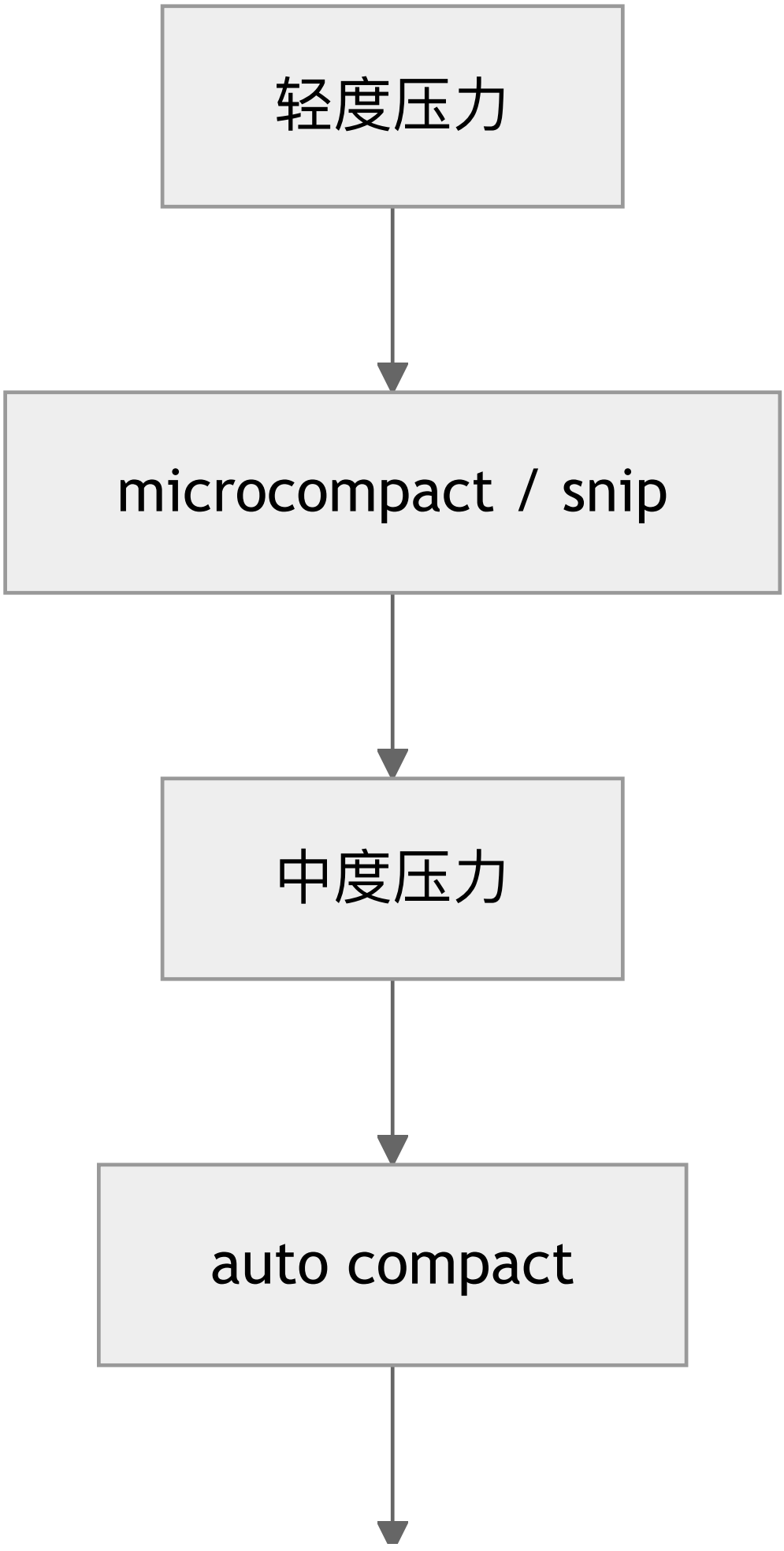
- 让一些片段折叠，而不是彻底消失
- 把冗长工具结果改写成摘要
- 保留边界消息和任务骨架

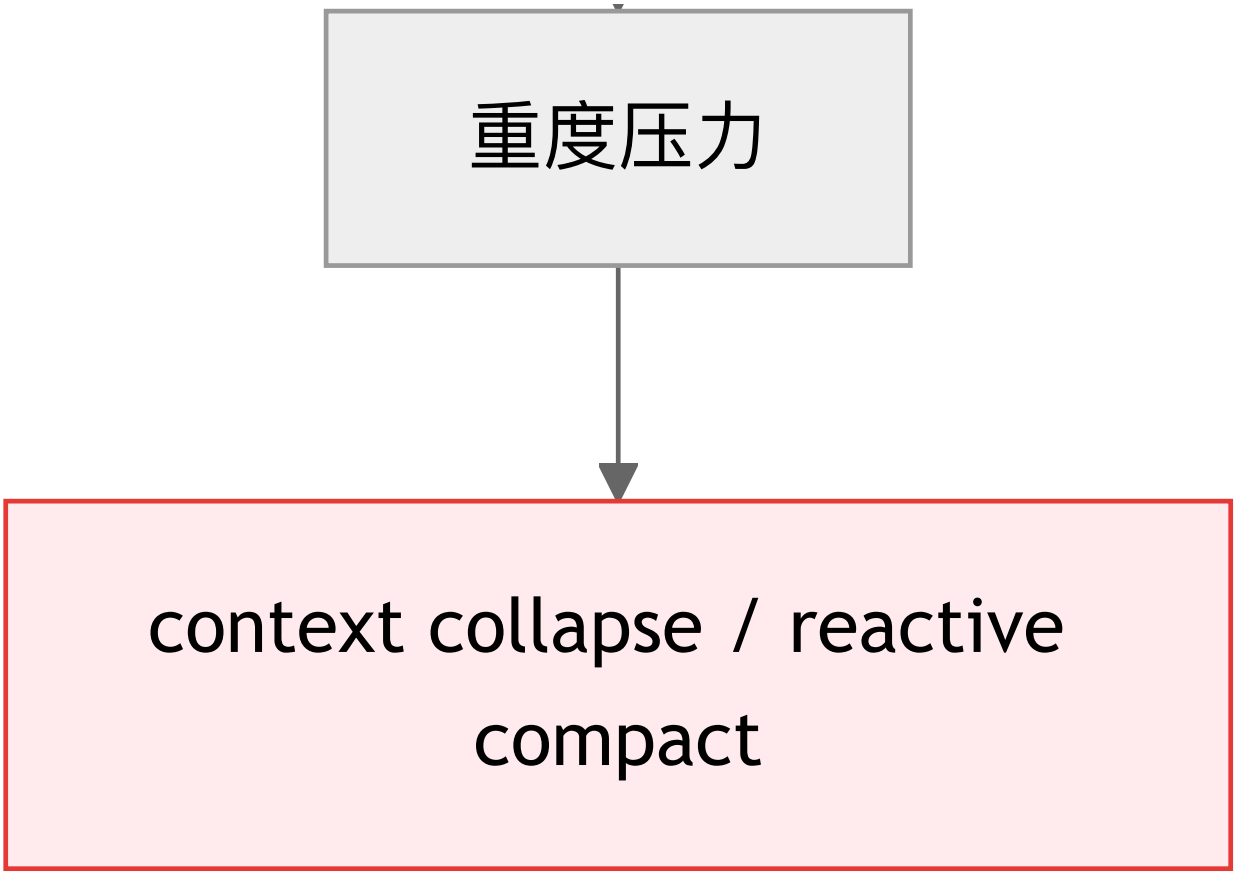


这是一种很高级的“信息密度管理”思路：不只问“删掉什么”，还问“怎样删得更聪明”。

31.6 设计取舍：压缩系统为什么一定要多层

如果只有一种压缩策略，它迟早会在某种场景下失效。Claude Code 选择多层压缩，本质上是在处理不同级别的上下文压力：


轻度压力
microcompact / snip
中度压力
auto compact



这背后的产品观也很清楚：

- 平时少动大刀
- 先删低价值负担
- 真危险了再走激进路径

这样既能保住体验，也能保住任务连续性。

深水区（架构师选读）

上下文压缩最难的不是“压缩率”，而是“压完还能不能继续工作”。Claude Code 把压缩设计成一条渐进式流水线：先微调，再自动，再应急。这个层级结构，比单一摘要器要稳得多，也更适合 Agent 场景。

本章小结

Claude Code 的压缩系统不是单点功能，而是一套多层策略：microcompact 和 snip 负责精细减重，auto compact 负责提前管理，context collapse 和 reactive compact 负责应急救火。

关键源码索引

- `query.ts` 引入 compact 体系：`query.ts`
- Snip 预处理：`query.ts`
- context collapse 应用：`query.ts`
- overflow 恢复：`query.ts`
- reactive compact 重试：`query.ts`
- auto compact 阈值与入口：`autoCompact.ts`
- autoCompactIfNeeded：`autoCompact.ts`
- buildPostCompactMessages：`compact.ts`
- microcompact 主入口：`microCompact.ts`
- snipCompactIfNeeded：`snipCompact.ts`
- recoverFromOverflow：`index.ts`

逆向提醒

压缩效果和模型行为强相关。源码能解释“什么时候压、怎么压、压后消息如何重建”，但不能保证任何一次压缩都保留了最优语义，这依然是运行时策略和模型质量共同决定的结果。

41

Experimental 第七编

第32章：实验区：哪些功能还在试验

生活类比：车展上的概念车

车展上有些车下个月就量产，有些只是展示未来方向。源码里的实验功能也一样：存在，不代表已经是成熟产品。

这一章先回答一个问题

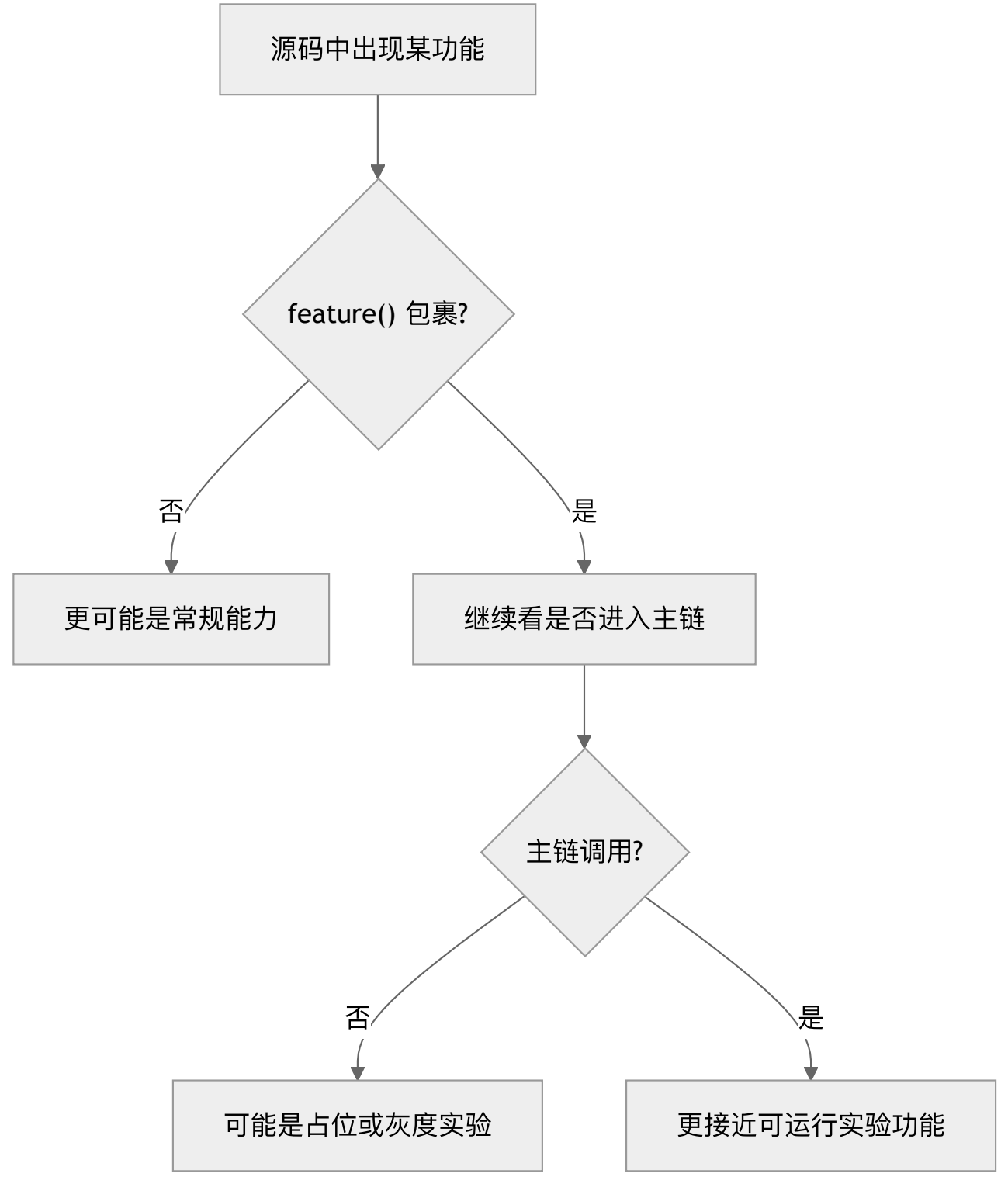
当你在 Claude Code 源码里看到 `contextCollapse`、`snip`、`KAIROS`、隐藏命令、feature gate 时，怎么判断它们是正式功能、灰度功能，还是仅仅为将来留的位置？

真正的答案不是“看目录里有没有”，而是看三件事：**是否被 feature gate 包裹、是否接进主调用链、是否有完整治理路径。**

32.1 第一个判断标准：有没有被 `feature()` 包起来

在 Claude Code 里，很多实验代码不会直接静态常驻，而是通过 `feature('FLAG')` 有条件加载。最典型的例子就在 `QueryEngine.ts`：

- `HISTORY_SNIP`
- `COORDINATOR_MODE`
- `KAIROS`



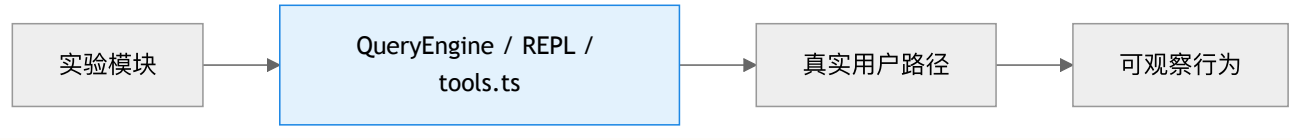
这也是为什么“看到了代码”不等于“用户一定能用到”。很多路径会在构建时被 DCE 直接裁掉。

32.2 第二个判断标准：是否真的接进主执行链

一个实验功能如果只是孤零零躺在目录里，价值不大。真正值得关注的，是它有没有进入：

- QueryEngine
- REPL
- tools.ts
- commands.ts
- settings / config

例如 `contextCollapse`、`snipCompactIfNeeded()`、`getCoordinatorUserContext()` 这些，都已经在主链上占到了位置。



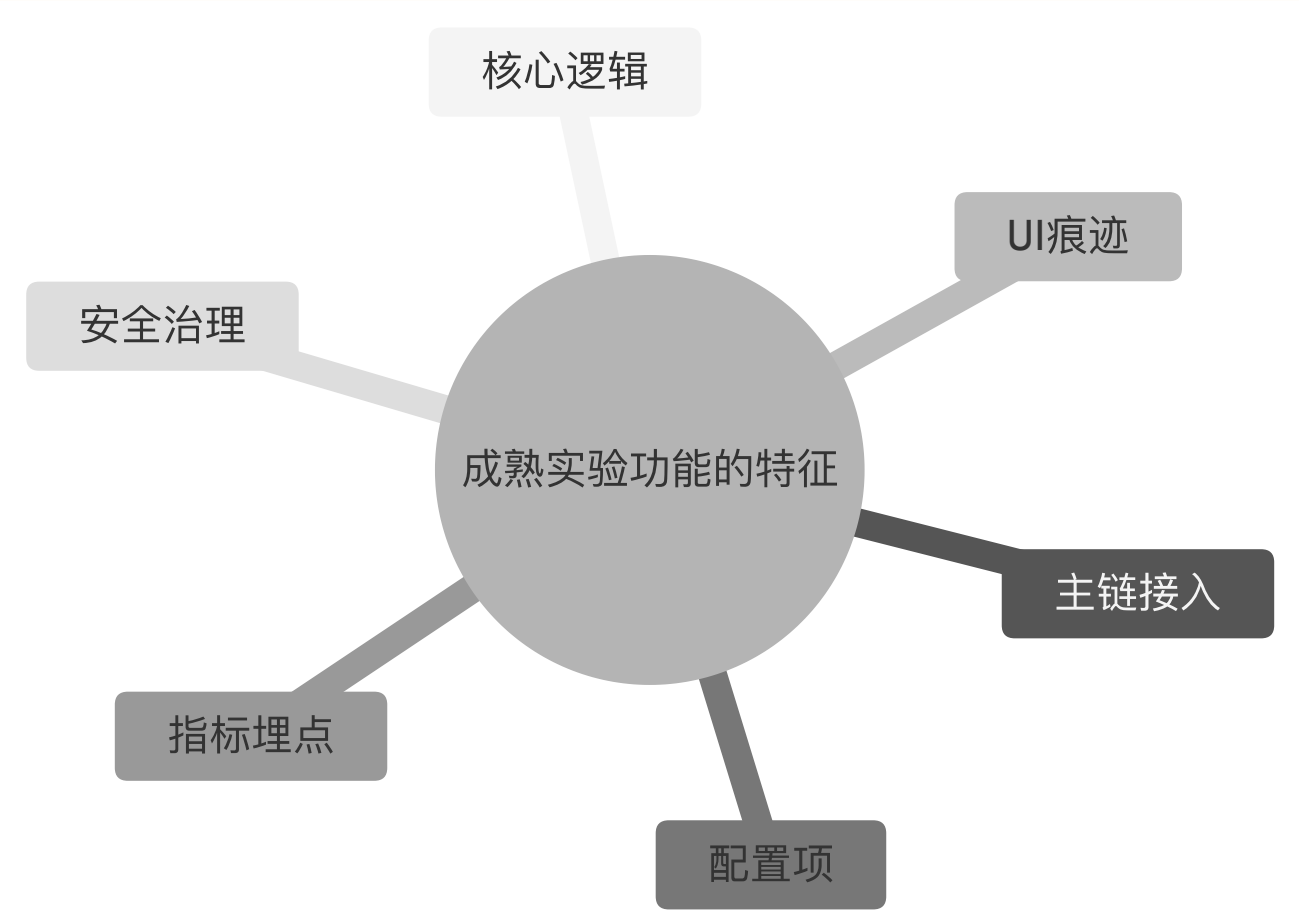
如果一段代码既有 gate，又有主链接入，那它通常不是“随手放着玩的”。

32.3 第三个判断标准：有没有完整的配套治理

很多真正要上线的实验功能，不会只有一段核心逻辑，还会伴随：

- 配套设置项
- 提示词或系统上下文
- 权限规则
- UI 入口
- 日志与指标

例如 KAIROS 不只出现在一个目录里，还能在 `tools.ts`、`main.tsx`、`REPL.tsx`、analytics metadata、BriefTool 等多个位置看到痕迹。



如果只看到一段核心逻辑，没有配套治理，那它更像概念验证。

32.4 `contextCollapse` 和 `snip` 为什么最值得放进“实验区”讲

这两类能力特别适合作为“如何识别实验层”的教材，因为它们恰好卡在正式能力和探索能力之间：

- 已经和主循环挂上了
- 但仍受门控控制
- 功能目标明确
- 语义稳定性仍在打磨

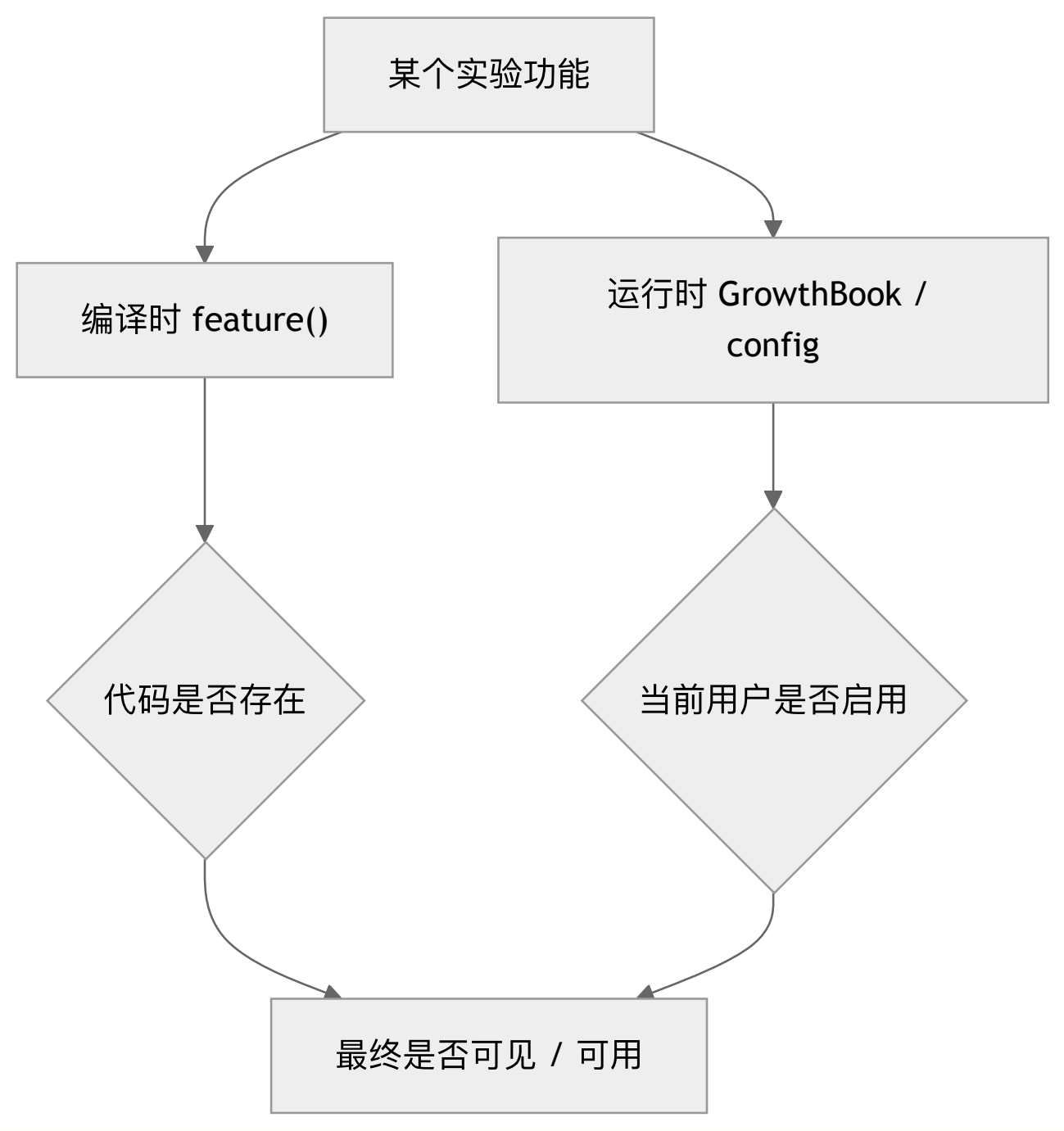
这也说明读源码时不能只问“有没有”，还要问“成熟到什么程度”。

32.5 GrowthBook 和 feature gate 一起，形成了实验层的双门控

Claude Code 的实验能力，不完全只靠编译时 gate。还有一部分会进入 `GrowthBook` 这类远程配置体系。

所以实验功能的真实状态，可能要同时看：

- 构建时是否被编进来
- 运行时是否被组织、账号、环境、实验分组打开

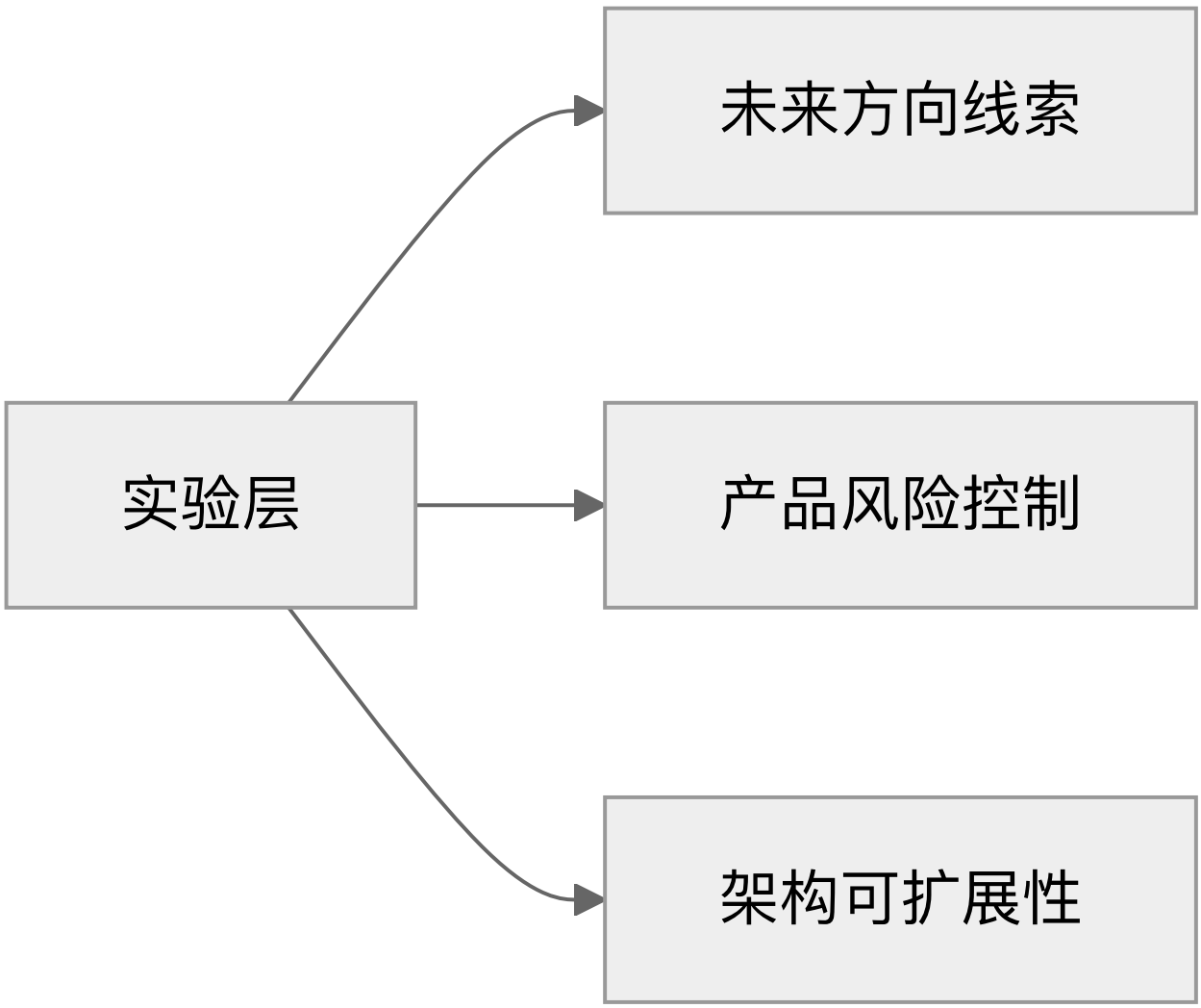


这也是为什么“我在仓库里看到了，但机器上没出现”经常是正常现象。

32.6 设计取舍：实验层不是脏东西，而是产品演化现场

很多人读源码会下意识嫌弃实验代码，觉得“这不纯”。其实对大型产品来说，实验层恰好是最有信息密度的地方，因为它暴露了：

- 团队正在试什么
- 哪些方向还没定型
- 哪些功能被谨慎推进
- 哪些只适合特定用户群



对这本书来说，实验区不是边角料，反而是理解 Claude Code 演化方向的重要窗口。

🔭 深水区（架构师选读）

判断实验层最实用的方法可以记成一句话：看 gate，看主链，看治理。三者都在，说明它是认真推进中的实验；只有一两个角落提到，多半还只是方向探索；如果再叠加 shim/stub，就要格外小心，不要把补全层误当成正式设计。

本章小结

`feature()`、主链接入程度、治理配套程度，是判断一段代码成熟度的三把尺子。实验功能不是杂质，而是产品未来方向最直接的证据。

关键源码索引

- QueryEngine 条件导入 snip / coordinator：`QueryEngine.ts`
- QueryEngine 条件导入 memory/snip：`QueryEngine.ts`
- Query 主循环中的 snip / collapse：`query.ts`
- 工具池中的 feature gate 痕迹：`tools.ts`
- GrowthBook 功能读取：`growthbook.ts`
- cached + refresh 路径：`growthbook.ts`

逆向提醒

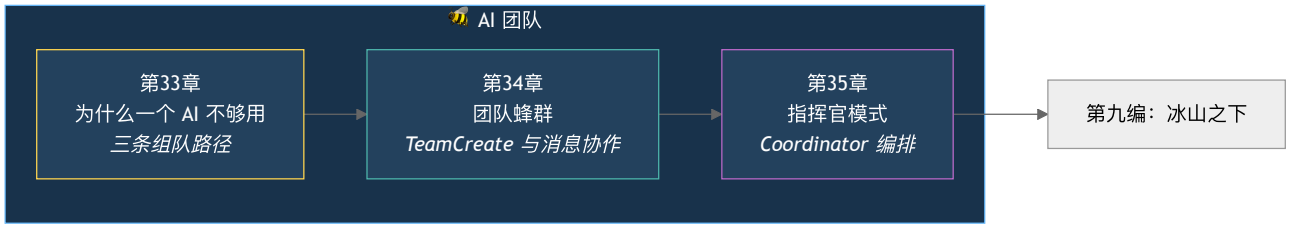
实验功能是最容易被误读的区域。因为你同时会在还原层、OpenClaudeCode 补全层、feature gate、GrowthBook、隐藏入口里看到它的碎片。结论一定要基于多处证据交叉，而不是只看一个目录或一个函数名。

42

第八编：AI 团队

> *一个人能做很多事，但真正复杂的工程，往往靠分工协作。*
>
> Claude Code 的多智能体体系，不只是“多开几个窗口”，而是把 **子 Agent、worktree 隔离、团队消息、Coordinator 编排** 组合成了多种协作模式。

本编总览



本编三章速览

章	标题	核心问题	生活类比
33	为什么一个 AI 不够用	子 Agent、worktree、远程隔离到底差在哪?	一个人搬家 vs 搬家公司
34	团队蜂群	多个 Agent 怎样像团队一样协作?	蜜蜂分工
35	指挥官模式	什么时候需要 Coordinator，而不是自由协作?	乐队指挥

本编阅读目标

读完这一编，你会明白 Claude Code 的多智能体不是噱头，而是一套围绕隔离、协作、编排和恢复设计出来的工作体系。

43

Multi-Agent 第八编

第33章：为什么一个 AI 不够用

生活类比：搬家公司

一个人搬家可以慢慢来，但遇到打包、拆装、运输、清点同时发生时，分工协作会快得多。Claude Code 的多智能体体系，也是为了把复杂任务拆成可以并行推进的工作单元。

这一章先回答一个问题

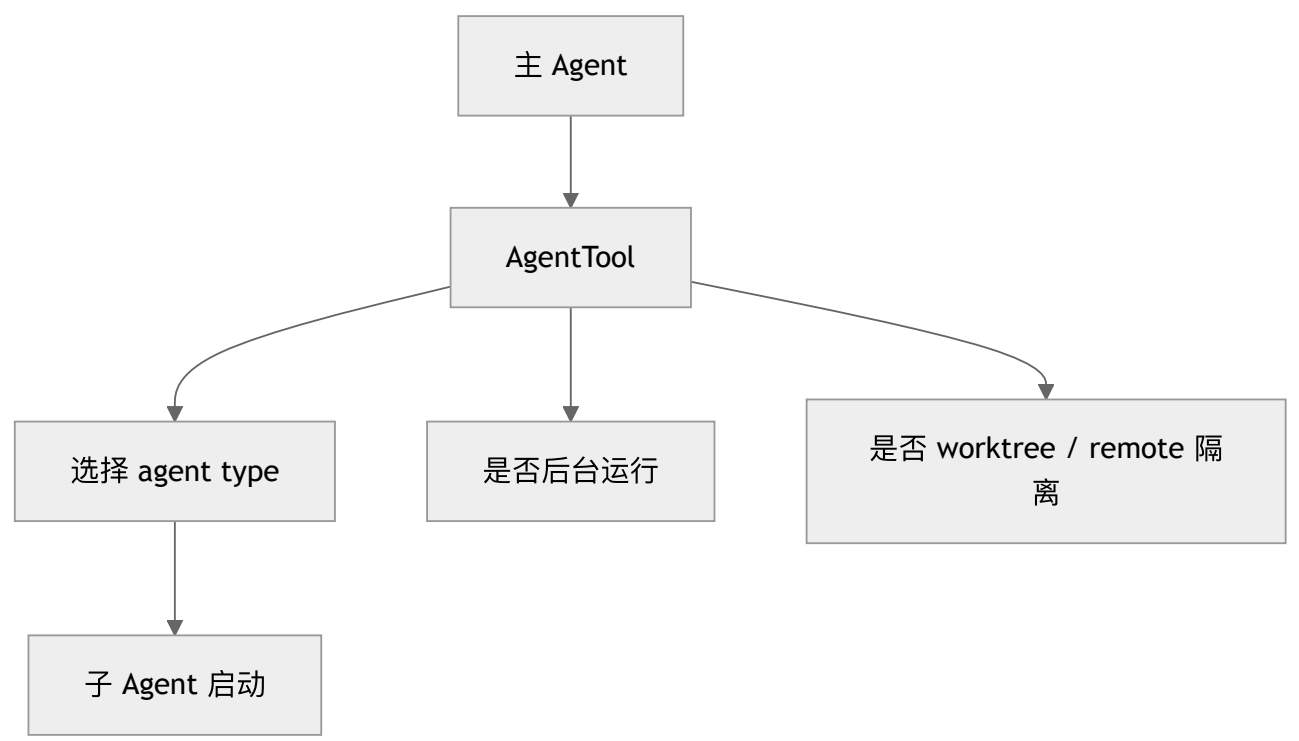
Claude Code 为什么不坚持“一个 Agent 干到底”，而要引入子 Agent、worktree 隔离、远程隔离这些更复杂的路径？

因为复杂任务有三个痛点：**上下文会被塞爆、不同子任务会互相干扰、串行执行太慢**。多智能体就是对这三个问题的系统性回答。

33.1 AgentTool：多智能体的大门

`AgentTool.tsx` 不是一个小插件，而是整套多智能体体系的入口。它的 schema 已经直接暴露出几个关键概念：

- `subagent_type`
- `run_in_background`
- `isolation`

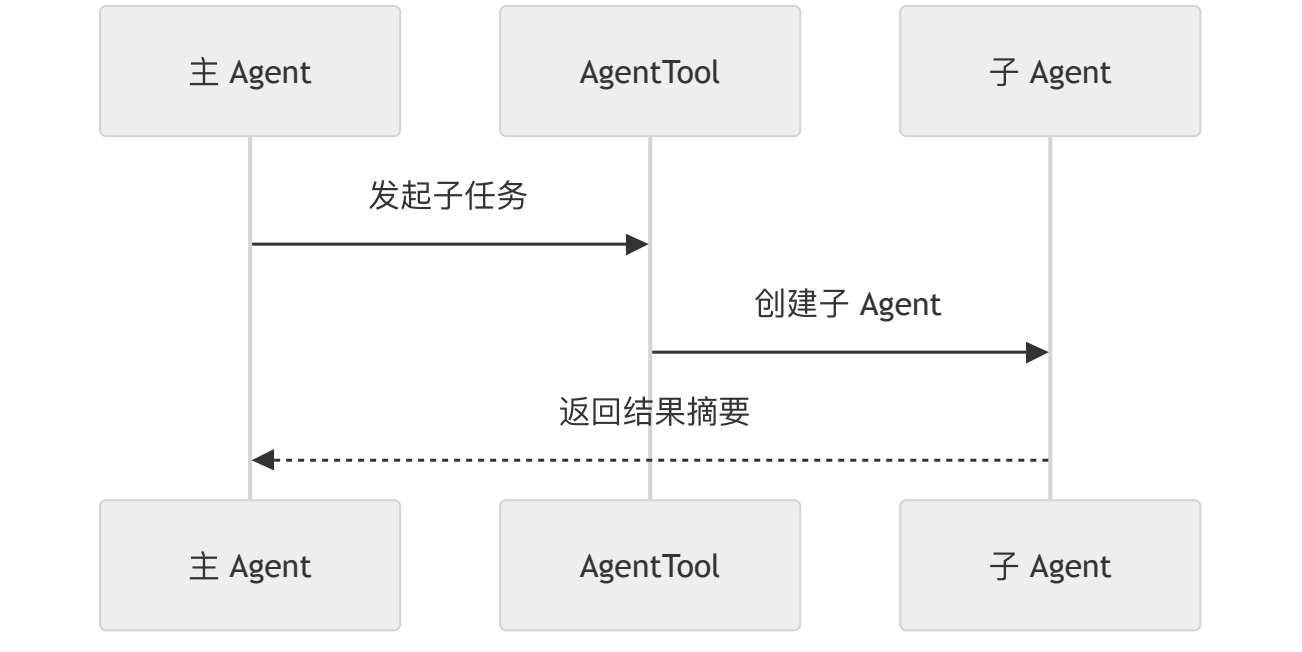


从这个 schema 就能看出来：Claude Code 把“派一个子 Agent 去做事”当成了一等公民能力。

33.2 第一条路径：普通子 Agent，最快但共享风险也最多

最基础的情况是直接派生子 Agent，让它继承一部分任务描述和工具能力去执行。它适合：

- 搜索与调研
- 独立分析
- 一次性验证



这条路径的优点是启动快，缺点是如果任务涉及文件改动，互相踩到上下文和工作区的风险会更高。

33.3 第二条路径：worktree 隔离，让子 Agent 真正拥有独立工位

Claude Code 很认真地对待“同时改代码”这件事，所以 `AgentTool.tsx` 会和 `utils/worktree.ts` 配合，支持 `isolation: "worktree"`。

这条路径做的事情包括：

- 创建或恢复独立 git worktree
- 复用仓库结构
- 允许子 Agent 在隔离副本里工作
- 完成后根据变更决定保留还是清理

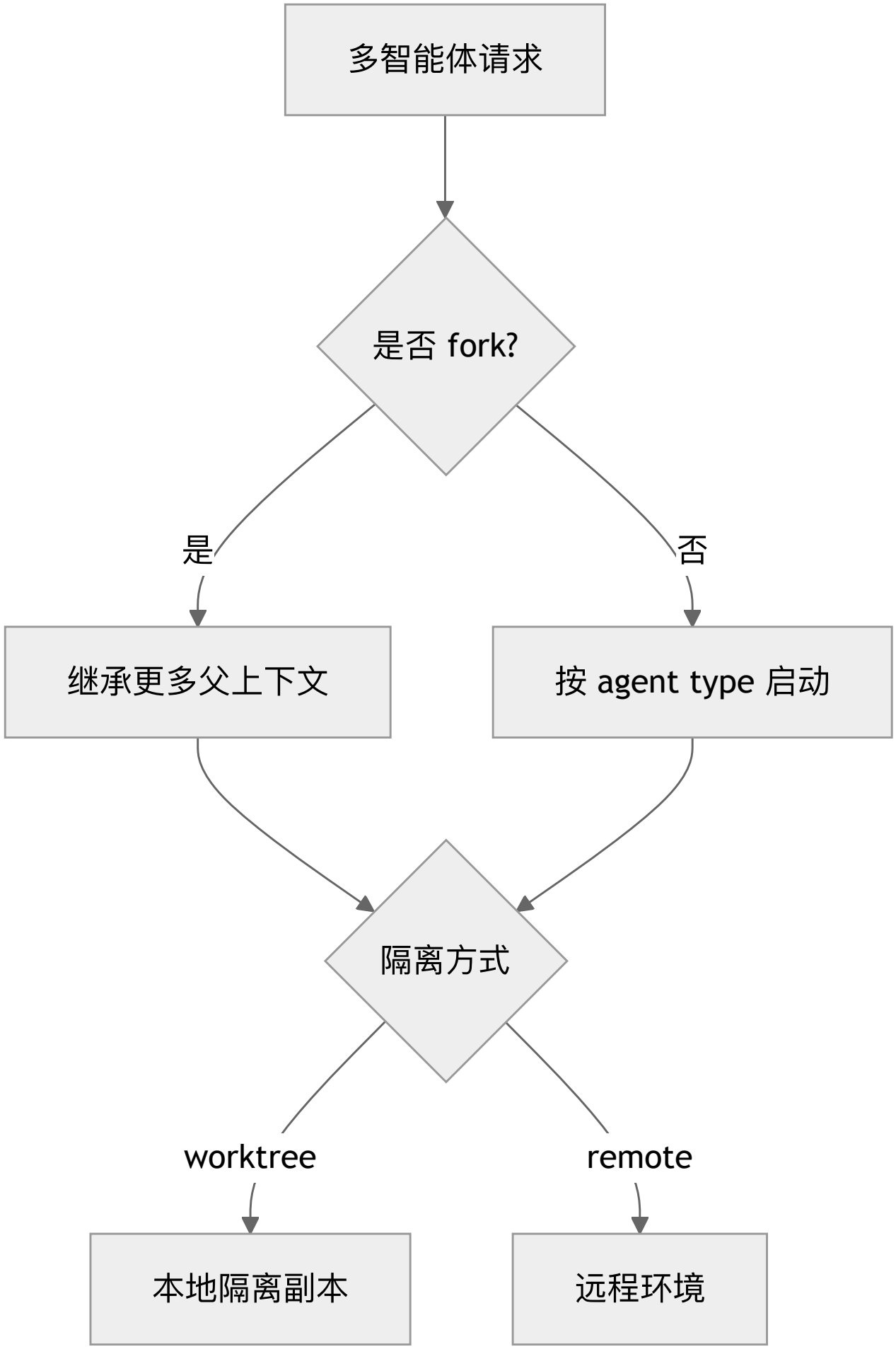
这很像给每个子 Agent 发了一张独立工位和独立草稿纸。

33.4 第三条路径：fork / remote，把上下文继承和环境隔离都做成可选

`forkSubagent.ts` 展示了另一条路线：当实验门控打开时，省略 `subagent_type` 可以走隐式 fork，让子 Agent 继承更完整的上下文。

再加上 `loadAgentsDir.ts` 对 `isolation` 的定义，Claude Code 实际上允许两种隔离思路：

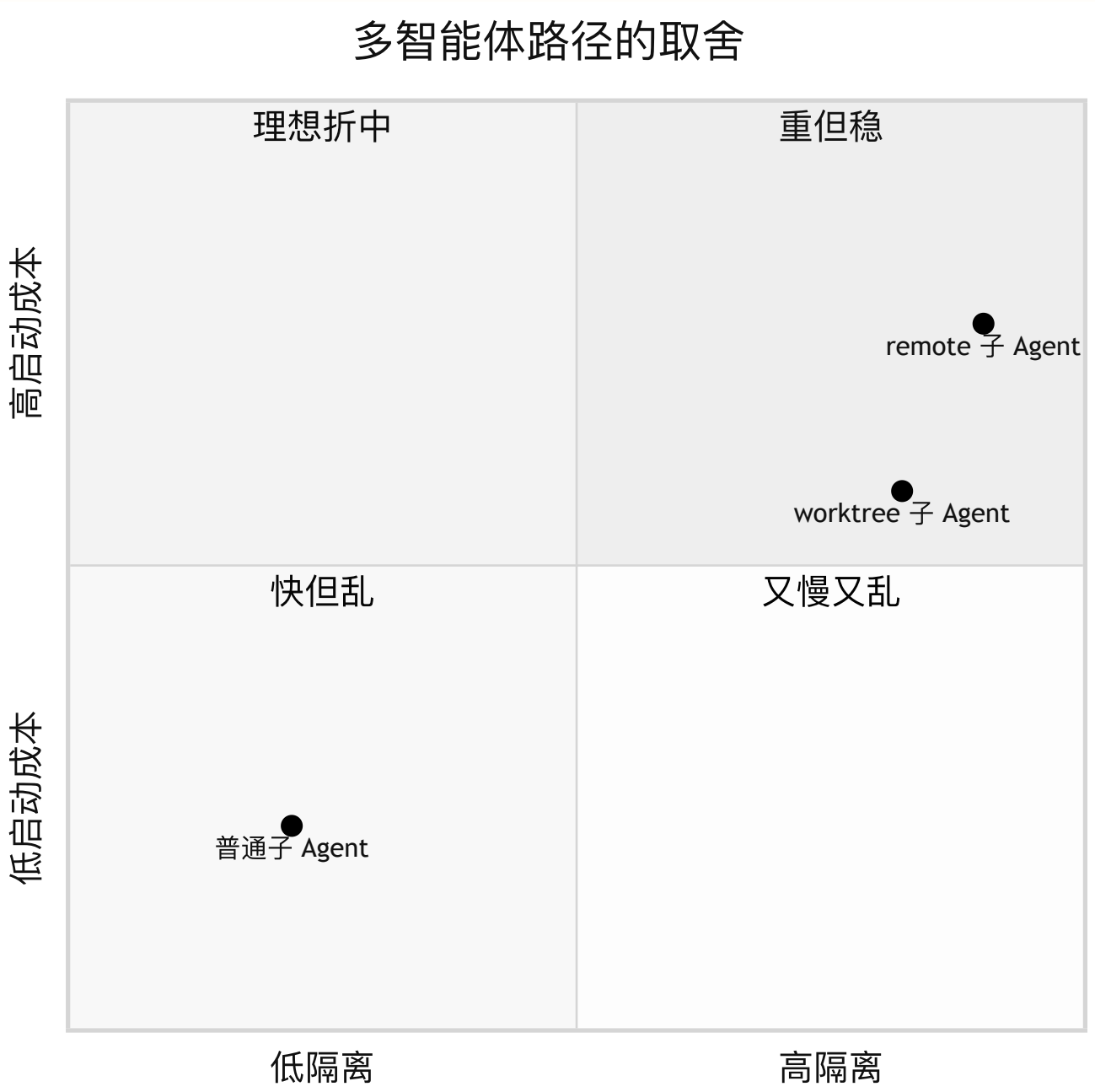
- `worktree`
- `remote`


多智能体请求
是否 fork?
是
否
继承更多父上下文
按 agent type 启动
隔离方式
worktree
remote
本地隔离副本
远程环境

这说明 Claude Code 并没有把多智能体固化成一种模式，而是在做“上下文继承程度”和“执行环境隔离程度”的组合设计。

33.5 设计取舍：为什么多智能体一定要配隔离

如果只是“多开几个 Agent”，却让它们都在同一个工作区里随便写，那收益会很快被混乱吞掉。



Claude Code 的多智能体设计非常现实：它承认协作越强，越要先解决隔离。

深水区（架构师选读）

这一章最重要的不是记住三个名字，而是理解两个设计轴：上下文继承程度、执行环境隔离程度。Claude Code 的多智能体不是“多模型并发”这么简单，而是在这两条轴上组合出多种协作形态。

本章小结

多智能体存在的理由，是为了让复杂任务可以拆、可以并行、可以隔离。AgentTool、fork、worktree 和 remote 共同构成了 Claude Code 的三条组队路径。

关键源码索引

- AgentTool 输入 schema：`AgentTool.tsx`
- 隔离模式定义：`AgentTool.tsx`

- worktree 创建与清理：`AgentTool.tsx`
- fork subagent 实验说明：`forkSubagent.ts`
- fork + worktree notice：`forkSubagent.ts`
- agent isolation frontmatter：`loadAgentsDir.ts`
- worktree 实现：`worktree.ts`

逆向提醒

多智能体代码里既有正式能力，也混入了 feature gate 与 ant-only 路径。尤其 `remote` 和部分 fork 行为，要区分“接口已存在”与“默认环境里一定可用”。

44

Swarm 第八编

第34章：团队蜂群：AI 组队的架构

生活类比：蜜蜂分工

蜜蜂不是一窝蜂乱飞。侦察蜂找方向，工蜂搬运，守卫蜂维持秩序。Claude Code 的团队模式也一样，重点不是“多几个 Agent”，而是“多几个 Agent 之后如何协作”。

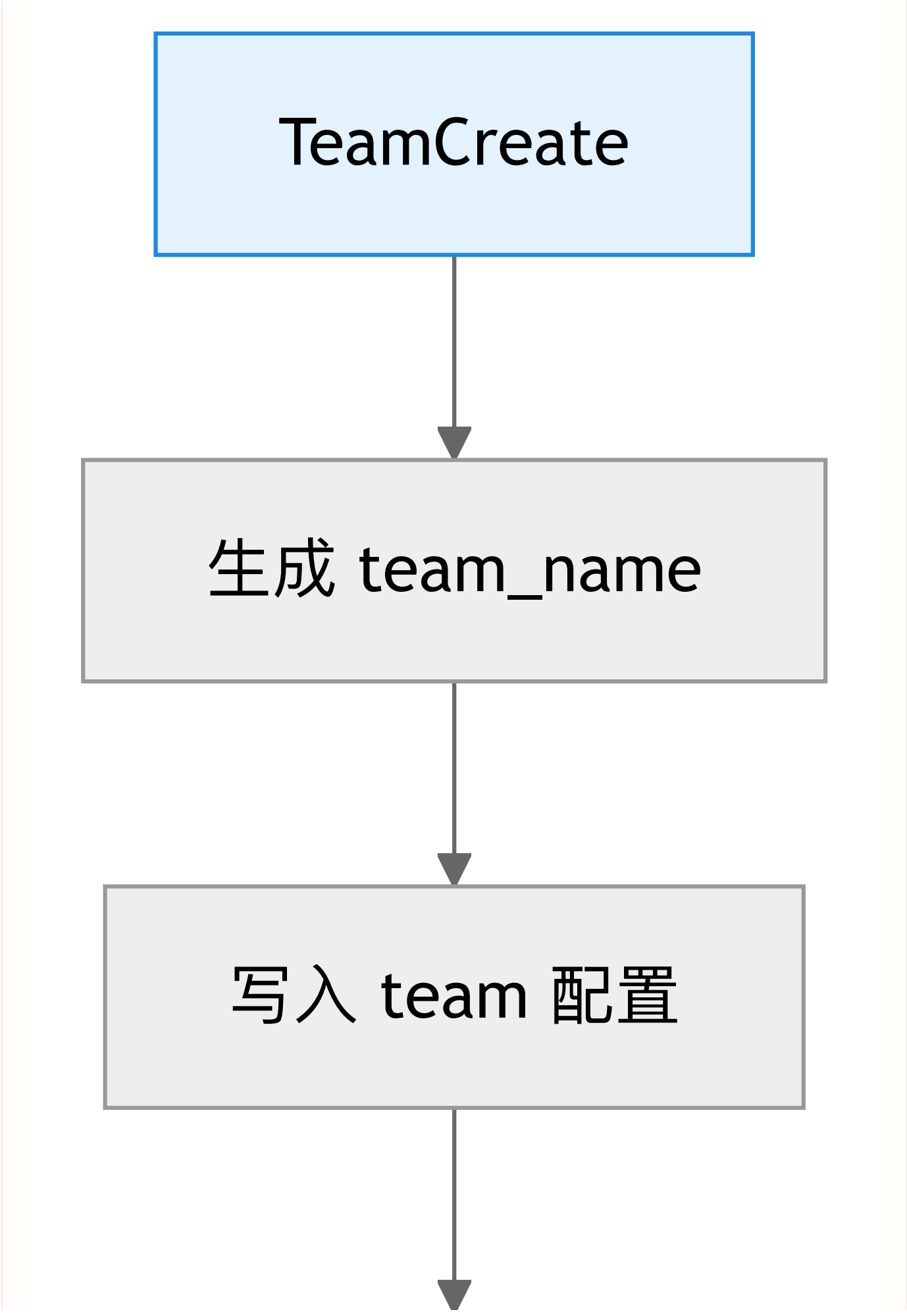
这一章先回答一个问题

多个 Agent 怎样像真正团队一样创建、沟通、同步状态、优雅结束，而不是各干各的、最后一地鸡毛？

Claude Code 的回答是：**TeamCreate 负责建队，SendMessage 负责沟通，team helpers 与 mailbox 负责维持秩序。**

34.1 TeamCreate：先把“团队”变成正式对象

`TeamCreateTool.ts` 会显式要求 `team_name`，并在内部创建团队上下文、成员信息和任务目录。这说明“团队”在 Claude Code 里不是隐喻，而是一个真正存在的数据结构。


TeamCreate
生成 team_name
写入 team 配置

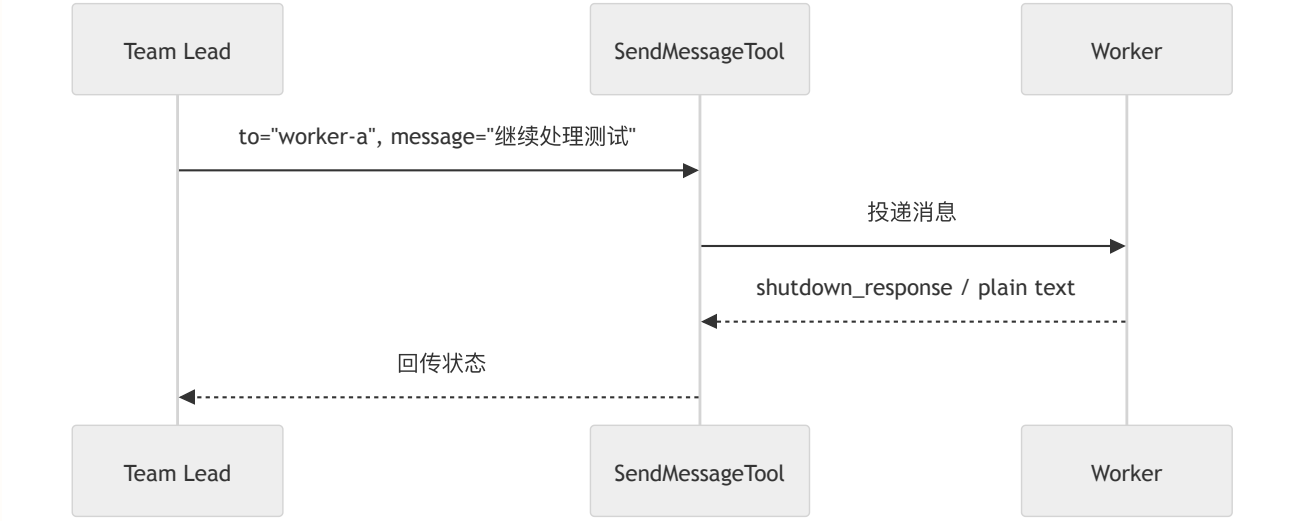
这一步很重要，因为只有先有“团队对象”，后面的权限同步、消息投递、空闲状态更新才有依托。

34.2 SendMessage：团队协作靠的不是猜，而是显式通信

`SendMessageTool` 的 prompt 直接强调：普通文本不会自动被别的 Agent 看见，要沟通就必须调用这个工具。

它支持：

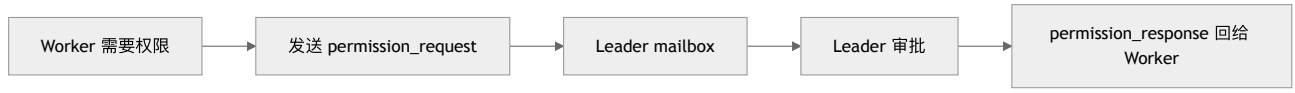
- 定向发给某个 teammate
- `*` 广播
- 结构化消息，如 `shutdown_request`



这让团队协作变成显式协议，而不是“大家共享心电感应”。

34.3 mailbox 与 permission sync：团队协作最难的是“谁替谁拿权限”

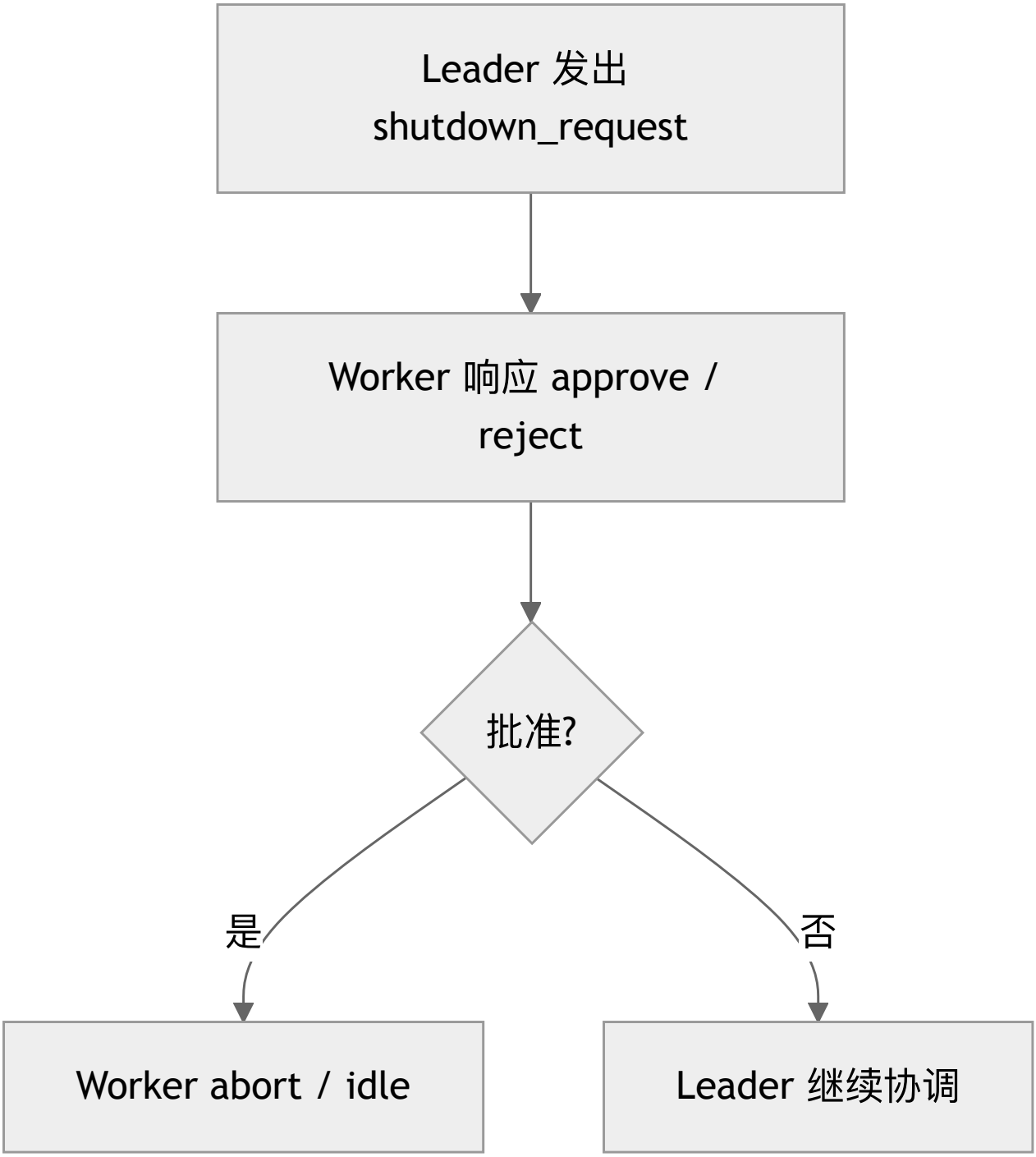
多 Agent 协作时，很容易遇到一种现实问题：工作 Agent 自己没权限，但它的负责人有权限。源码里的 `permissionSync.ts` 就是在解决这类问题。



这一步很妙，因为它把团队中的“审批链”也做进了系统，而不是要求所有 Agent 都拥有同样权限。

34.4 团队不只要能开始，还要能优雅结束

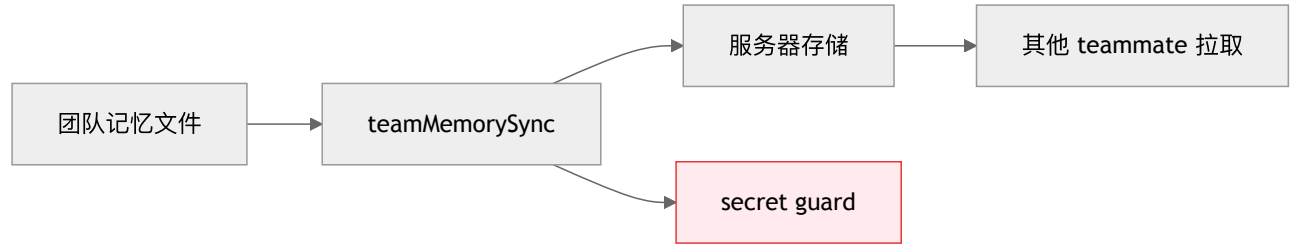
`SendMessageTool` 里能看到 `shutdown_request` / `shutdown_response`，`teammateInit.ts` 还会注册 Stop hook 通知 leader。说明团队结束不是“进程直接杀掉”，而是有协议化关机流程。



这很像真实团队里的“收尾确认”：确保每个人知道任务是否已完成、会话是否可以结束。

34.5 team memory sync：团队共识也要跟着团队一起流动

多智能体团队如果没有共享记忆，很快就会出现“每个人各记各的版本”。`teamMemorySync/index.ts` 负责把 repo-scoped 的 team memory 同步起来，并且还加了 secret guard。



这说明 Claude Code 眼里的“团队”，不只是会互发消息，还包括共享记忆和共享规则。

34.6 设计取舍：为什么团队协作一定要协议化

如果多人协作没有清晰协议，问题会迅速冒出来：

- 谁该响应谁
- 谁能广播
- 谁负责审批
- 谁决定关机
- 谁维护团队状态

Claude Code 选择把这些都写成结构化工具和消息协议，而不是留给 prompt 自由发挥。

深水区（架构师选读）

TeamCreate + SendMessage + mailbox + permission sync 这一套设计说明，Claude Code 的多智能体不是“多线程聊天”，而是在认真构建一个带协议的团队运行时。真正复杂的不是派出 worker，而是让 worker 和 lead 在权限、状态和消息上维持一致。

本章小结

团队模式的关键，不是多几个 Agent，而是团队对象、消息协议、权限同步、优雅关机和共享记忆这几件事一起成立。Claude Code 在这些方面都做了明确设计。

关键源码索引

- TeamCreate schema 与入口：`TeamCreateTool.ts`
- TeamCreate 工具定义：`TeamCreateTool.ts`
- TeamCreate 写入 teammates：`TeamCreateTool.ts`
- SendMessage 广播与结构化消息提示：`prompt.ts`
- SendMessage 结构化消息类型：`SendMessageTool.ts`
- 广播与 shutdown 处理：`SendMessageTool.ts`
- permission sync 协议说明：`permissionSync.ts`
- team memory sync：`index.ts`

逆向提醒

团队协作涉及 tmux、mailbox、hook、远程模式等多条路径。不同环境里实际走的后端不一定完全一样，但消息协议和状态管理思路在源码里是清楚的。

45

Coordinator 第八编

第35章：指挥官模式：编排引擎

生活类比：乐队指挥

指挥不拉小提琴、不吹长笛，但他决定谁先入场、谁做主旋律、什么时候收尾。Coordinator 在多智能体系统里的角色，也很像这样。

这一章先回答一个问题

当任务之间存在依赖关系时，为什么自由协作不够，还要引入 Coordinator 这种更重的编排模式？

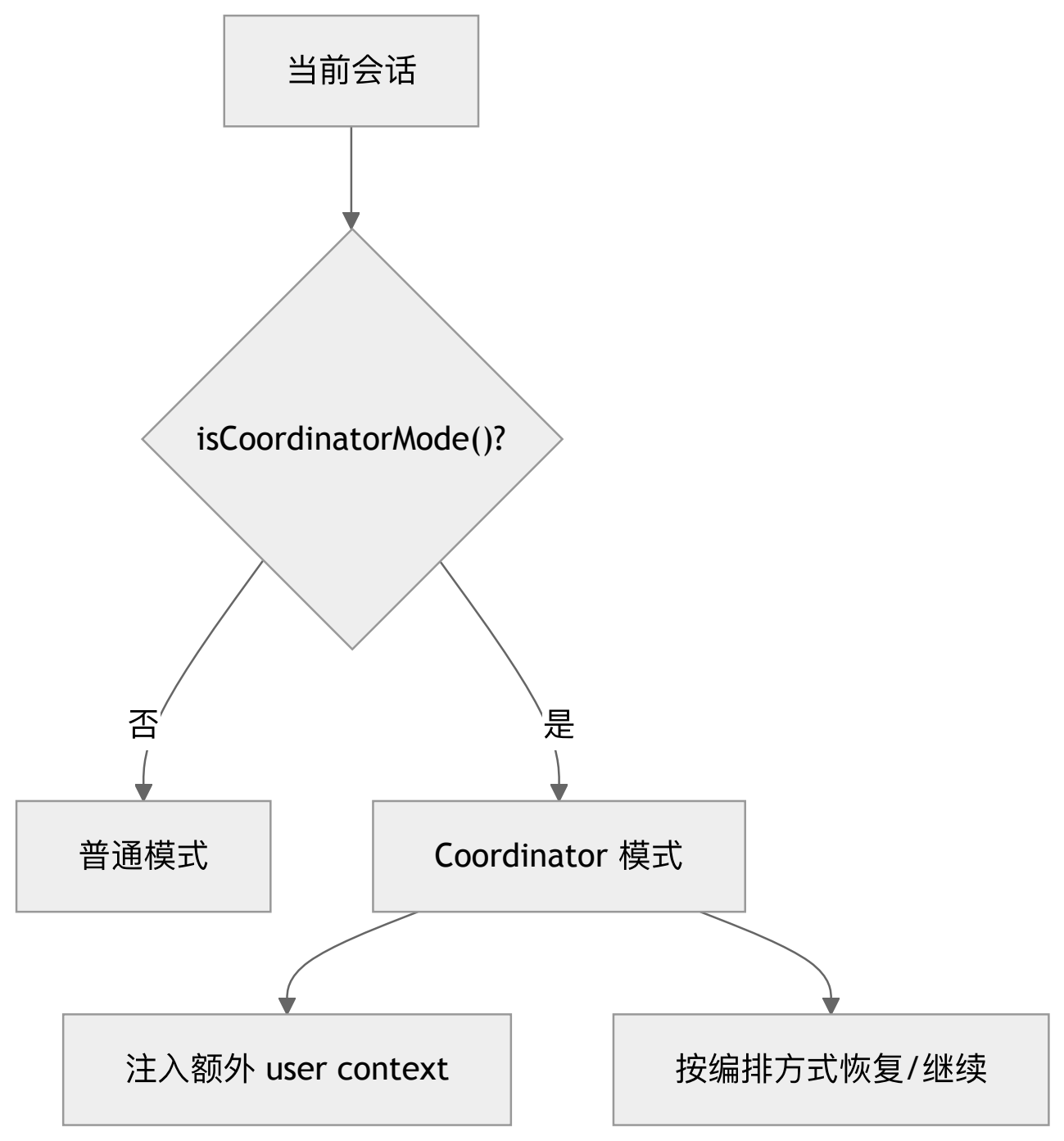
因为一旦任务之间有前后依赖、资源冲突、顺序要求，单纯“大家各干各的”就会让全局最优变成局部最忙。Coordinator 解决的是**顺序与全局视角**。

35.1 Coordinator 模式首先是一种“会话模式”

`coordinatorMode.ts` 里有非常清晰的模式判断：

- `isCoordinatorMode()`
- `matchSessionMode()`
- `getCoordinatorUserContext()`

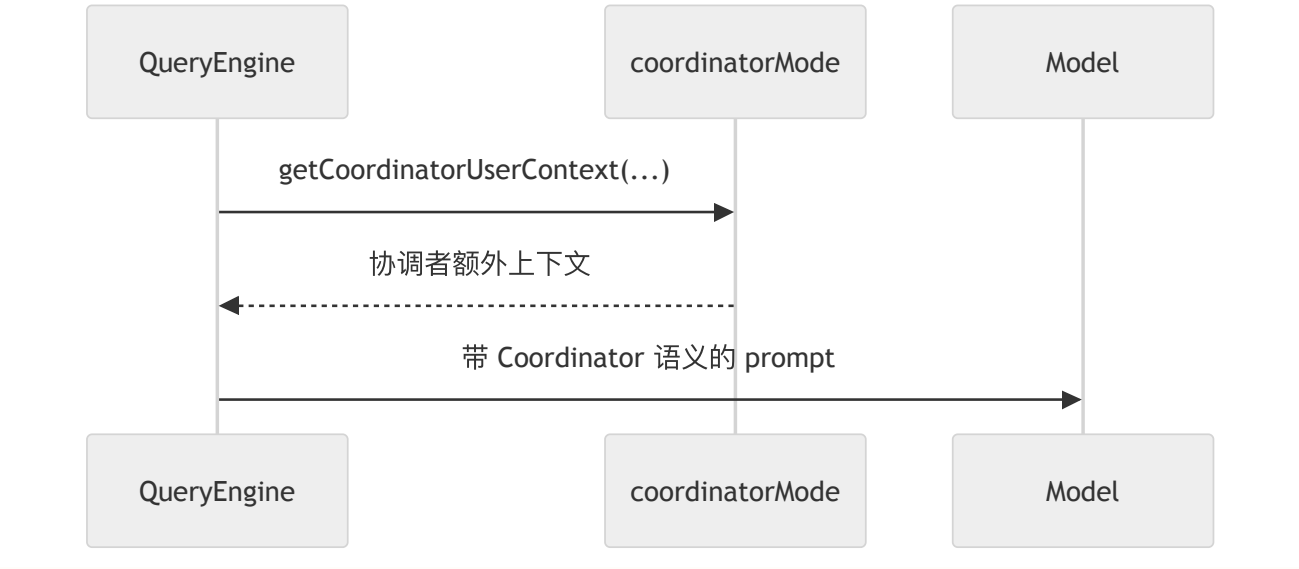
也就是说，Coordinator 不是某个散落工具，而是整个会话运行方式的一种切换。



这是非常重要的信号：编排不是临时附加动作，而是主循环本身要知道的上下文。

35.2 QueryEngine 也要知道“我现在是不是指挥官”

`QueryEngine.ts` 通过 `getCoordinatorUserContext()` 把协调者视角的上下文拼进来。换句话说，Coordinator 并不是只在 UI 层显示个标签，它会影响真正送进模型的上下文。

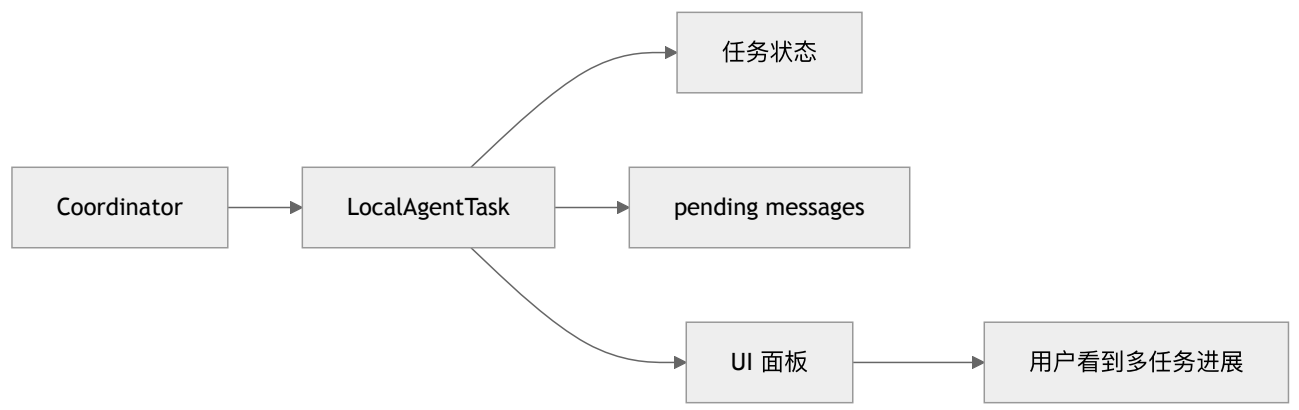


这说明 Coordinator 的本质，是让模型以“调度者”而不是“单个执行者”的身份思考。

35.3 LocalAgentTask：编排模式还需要一个可视化任务面板

`LocalAgentTask.tsx` 负责的，正是 Coordinator 模式下本地 agent task 的状态管理。源码里可以看到：

- 后台 agent 执行
- pending messages
- task state 更新
- panel 选择逻辑



这让 Coordinator 不只是“模型脑中的调度者”，还变成了用户可观察的任务编排界面。

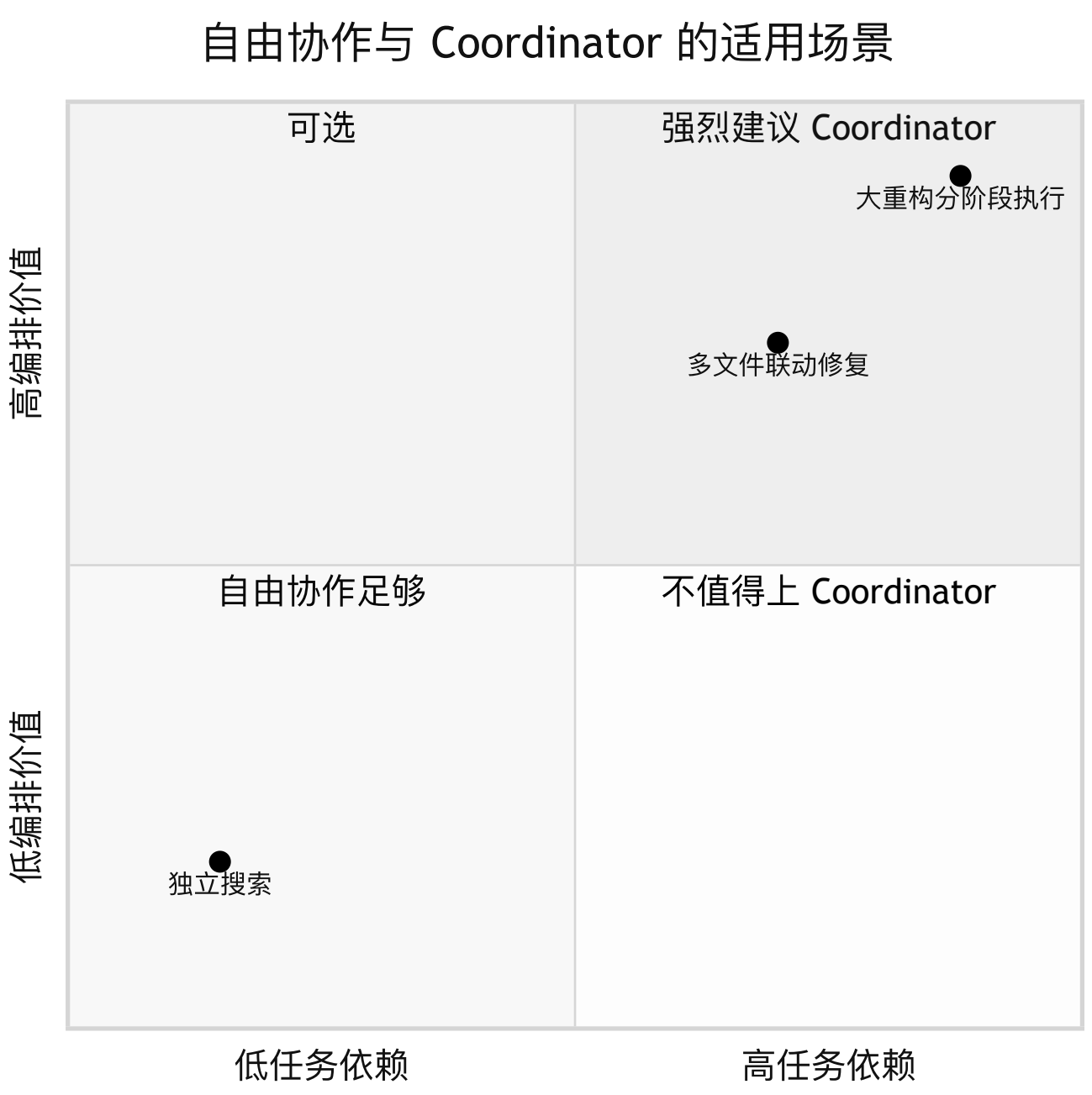
35.4 指挥官模式擅长什么，不擅长什么

Coordinator 特别适合：

- 有依赖关系的多任务
- 需要统一收口的执行流
- 需要全局进度视角的长任务

不那么适合：

- 超短小的独立搜索
- 完全无依赖的小任务



这就是“不是所有多人协作都需要指挥官”的意思。

35.5 设计取舍：Coordinator 的真正成本，是更强的系统自我意识

一旦系统进入编排模式，它就要额外维护：

- 会话模式一致性
- 恢复时模式匹配
- 子任务状态面板
- 多 agent 的消息与任务同步

所以它更强，但也更重。

深水区（架构师选读）

Coordinator 最值得学的地方，是把“系统角色”显式化。普通 Agent 是执行者，Coordinator 是编排者。只要角色不同，系统提示、状态管理、恢复逻辑和 UI 面板都要一起变。很多多智能体系统只改 prompt，不改运行时，结果就会半吊子。

本章小结

Coordinator 模式不是多开几个 Agent，而是让整场会话切换成“指挥官视角”。它适合高依赖、高复杂度任务，也因此需要更完整的状态、恢复和 UI 支撑。

关键源码索引

- QueryEngine 条件导入 Coordinator：`QueryEngine.ts`
- QueryEngine 注入 Coordinator user context：`QueryEngine.ts`
- 判断 Coordinator 模式：`coordinatorMode.ts`
- 匹配恢复会话模式：`coordinatorMode.ts`
- Coordinator user context：`coordinatorMode.ts`
- LocalAgentTask 状态类型：`LocalAgentTask.tsx`
- LocalAgentTask 主体：`LocalAgentTask.tsx`

逆向提醒

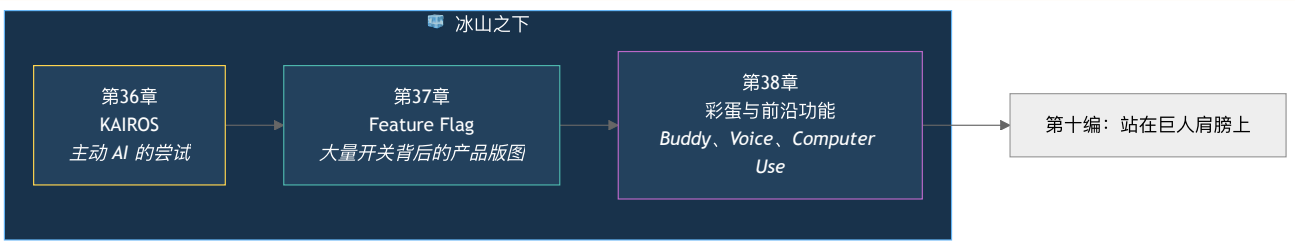
Coordinator 代码在 OpenClaudeCode 中能看到完整骨架，但任务依赖图和更高层策略细节并不都显式暴露。读到这里时，要区分“运行时支撑已在”与“最终产品策略已完全公开”。

46

第九编：冰山之下

> *很多产品真正透露方向感的，不是首页上写了什么，而是那些藏在 feature gate、实验模块和边缘目录里的代码。*
>
> 这一编专门看 Claude Code 的隐藏层：**KAIROS 主动模式、Feature Flag 系统、彩蛋与前沿能力**。

本编总览



本编三章速览

章	标题	核心问题	生活类比
36	KAIROS	AI 能不能从“等你下指令”变成“主动找事做”?	从实习生到资深同事
37	Feature Flag	为什么代码里到处都是开关?	一栋很多房间的大楼
38	彩蛋与前沿功能	Buddy、Voice、Computer Use 说明了什么?	游戏隐藏关卡

本编阅读目标

读完这一编，你会知道哪些代码只是彩蛋，哪些代码其实在提前泄露 Claude Code 的未来方向。

47

KAIROS 第九编

第36章：KAIROS：AI 学会主动思考

生活类比：从被动实习生到主动同事

被动型助手会一直等你发话，主动型同事会在合适的时候自己发现问题、整理线索、推进事情。KAIROS 代表的，就是这种从“响应式 AI”走向“主动式 AI”的尝试。

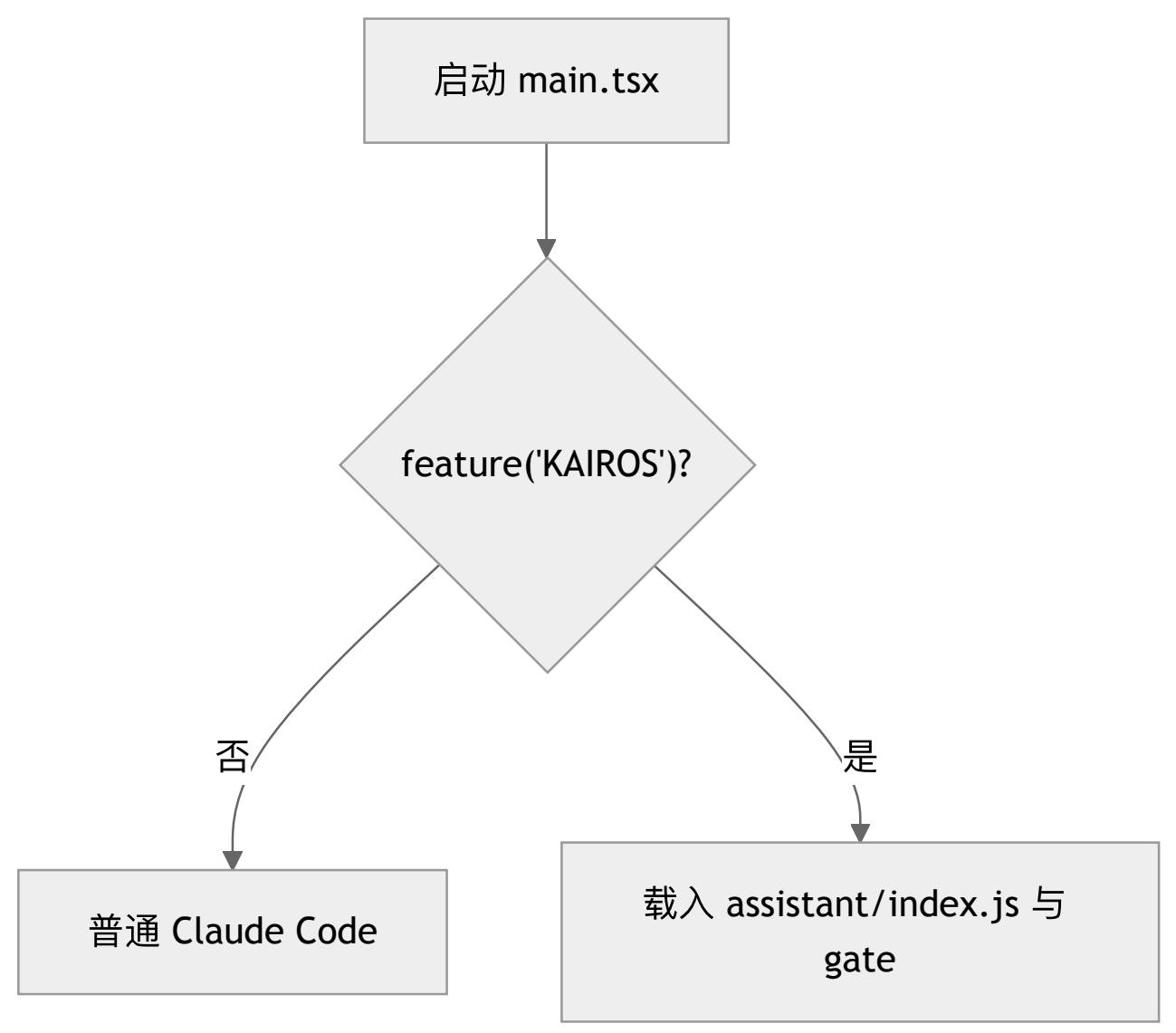
这一章先回答一个问题

如果 AI 不只是等你说一句做一步，而是能自己安排后台动作、主动整理记忆、主动给简报，你会更高效，还是更没安全感？

Claude Code 显然已经在探索这个方向，因为 KAIROS 的痕迹不是一两个变量，而是横跨 `main.tsx`、`tools.ts`、Brief、autoDream、BashTool、REPL 的系统级实验。

36.1 KAIROS 不是单点功能，而是一种模式切换

在 `main.tsx` 里，`assistantModule` 和 `kairosGate` 是以条件导入的方式出现的。这说明 KAIROS 不是普通小工具，而是一个足以影响启动路径和会话模式的能力。

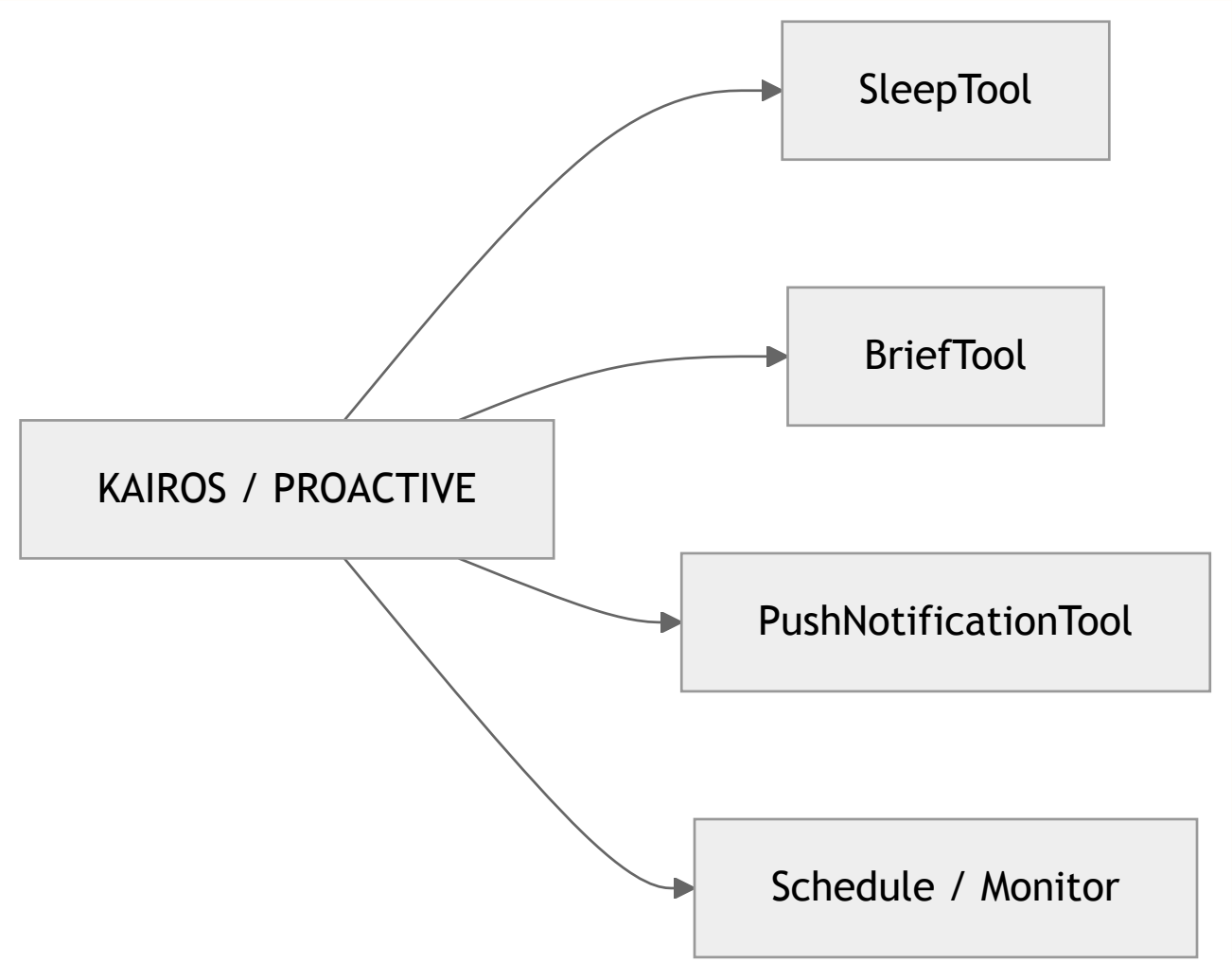


这件事非常关键，因为它说明 KAIROS 不是“聊天时偶尔多说一句”，而是会改变运行时结构的能力。

36.2 KAIROS 最直观的特征：它开始把后台任务纳入核心体验

从 `tools.ts` 可以看到，只要 `feature('PROACTIVE') || feature('KAIROS')` 成立，一系列工具就会被带进来：

- SleepTool
- BriefTool 相关
- Push Notification 相关
- Trigger / Monitor 相关能力

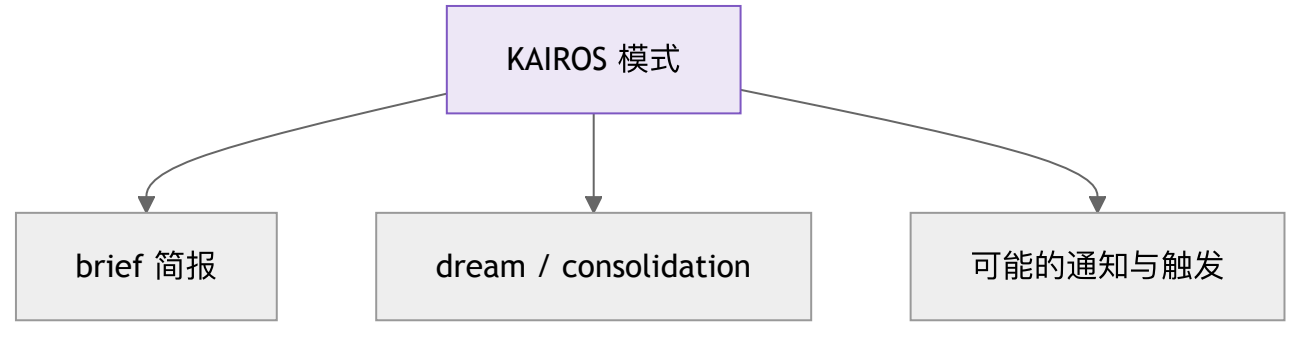


这说明 KAIROS 的目标不是“换个名字的聊天模式”，而是让 Claude Code 具备更持续、更异步、更后台化的行为模式。

36.3 autoDream 和 Brief，都是“主动整理”的侧面证据

`autoDream.ts` 里有一句很有意思的话：`if (getKairosActive()) return false // KAIROS mode uses disk-skill dream`。这说明 KAIROS 会影响记忆整理路径。

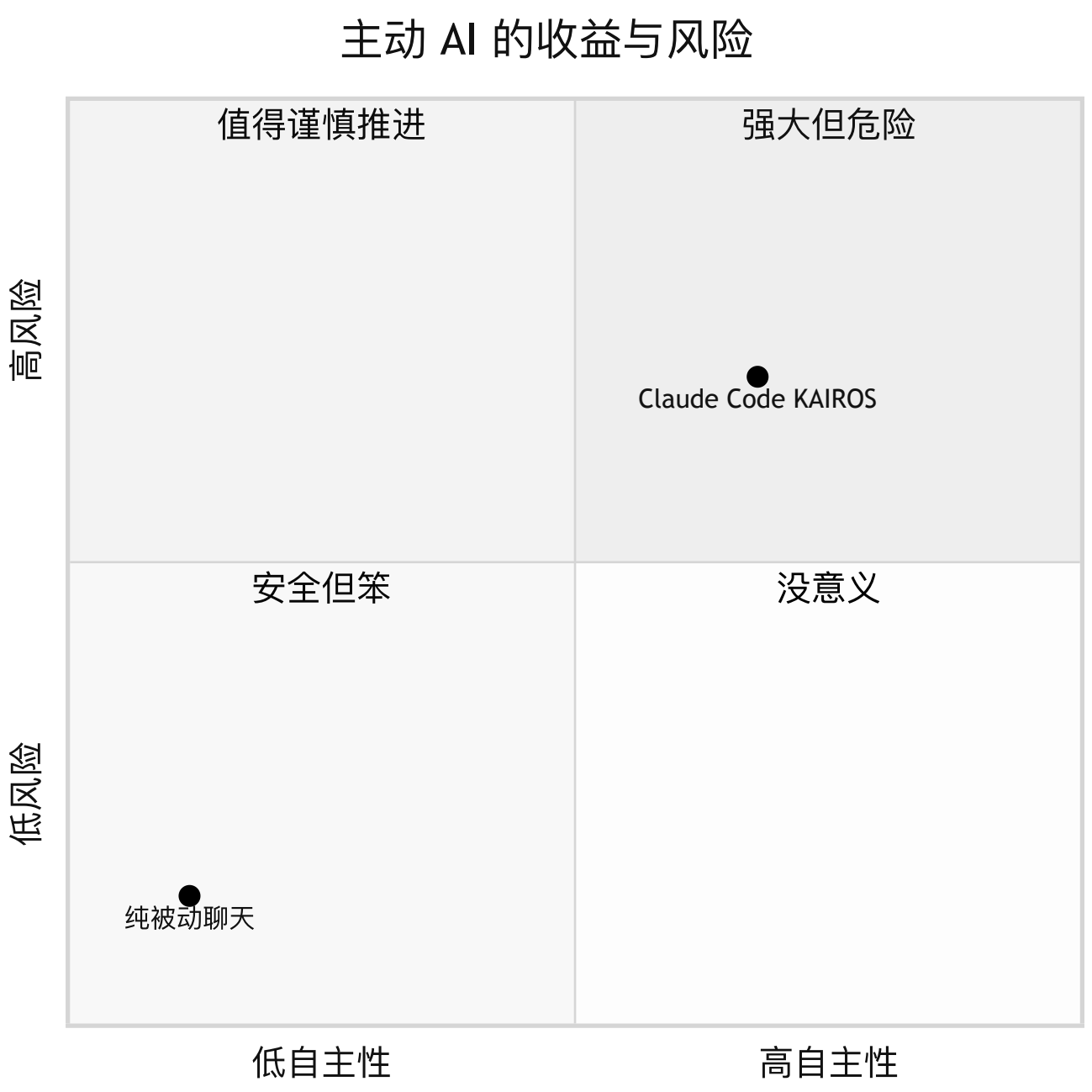
同时 `/brief` 命令和 `BriefTool` 也受 `KAIROS` 或 `KAIROS_BRIEF` 控制。



从产品视角看，这些都指向同一件事：**AI 不再只在前台回答，而开始在后台维持连续性。**

36.4 KAIROS 为什么危险，也为什么迷人

主动 AI 最大的好处，是帮你减少显式指挥成本；最大的风险，是它开始拥有“时机判断权”。



这也是为什么 KAIROS 一直被 gate、brief entitlement、autoDream 分流等多种机制包着。它显然很有潜力，但也显然不能轻率全量放开。

36.5 设计取舍：从“命令式助手”到“主动代理”的分水岭

KAIROS 最值得我们注意的，不是它现在成熟了多少，而是它定义了 Claude Code 想去的方向：

- 更持续的后台运行
- 更主动的任务整理
- 更少“你每一步都得明说”

深水区（架构师选读）

KAIROS 代表的是交互范式升级：从“prompt 驱动”走向“时机驱动”。一旦系统开始判断何时行动，它就不再只是工具，而更像一个长期在线的代理。这会把权限、安全、记忆、通知、后台执行全都重新拉到一个更高难度层级上。

本章小结

KAIROS 不是单个隐藏功能，而是 Claude Code 朝“主动式 Agent”演进的总代号。它的痕迹贯穿启动、工具、简报、记忆整理和后台行为。

关键源码索引

- `main.tsx` 中 KAIROS 条件导入：`main.tsx`
- `tools.ts` 中 KAIROS / PROACTIVE 工具门控：`tools.ts`
- autoDream 对 KAIROS 的分流：`autoDream.ts`
- `/brief` 命令门控：`brief.ts`
- analytics 中的 kairosActive 痕迹：`metadata.ts`

逆向提醒

KAIROS 是最容易被过度解读的区域之一。你能确认它是强烈存在的方向，但不能把所有相关门控分支都视为已稳定发布的最终产品能力。

48

Feature Flag 第九编

第37章：Feature Flag：89 个开关背后的秘密

生活类比：一栋很多房间的大楼

有些房间一直开放，有些房间只对特定人开放，有些房间正在装修，还有些房间甚至只在特定时间启用。Feature Flag 就像这栋楼里的开关与门禁系统。

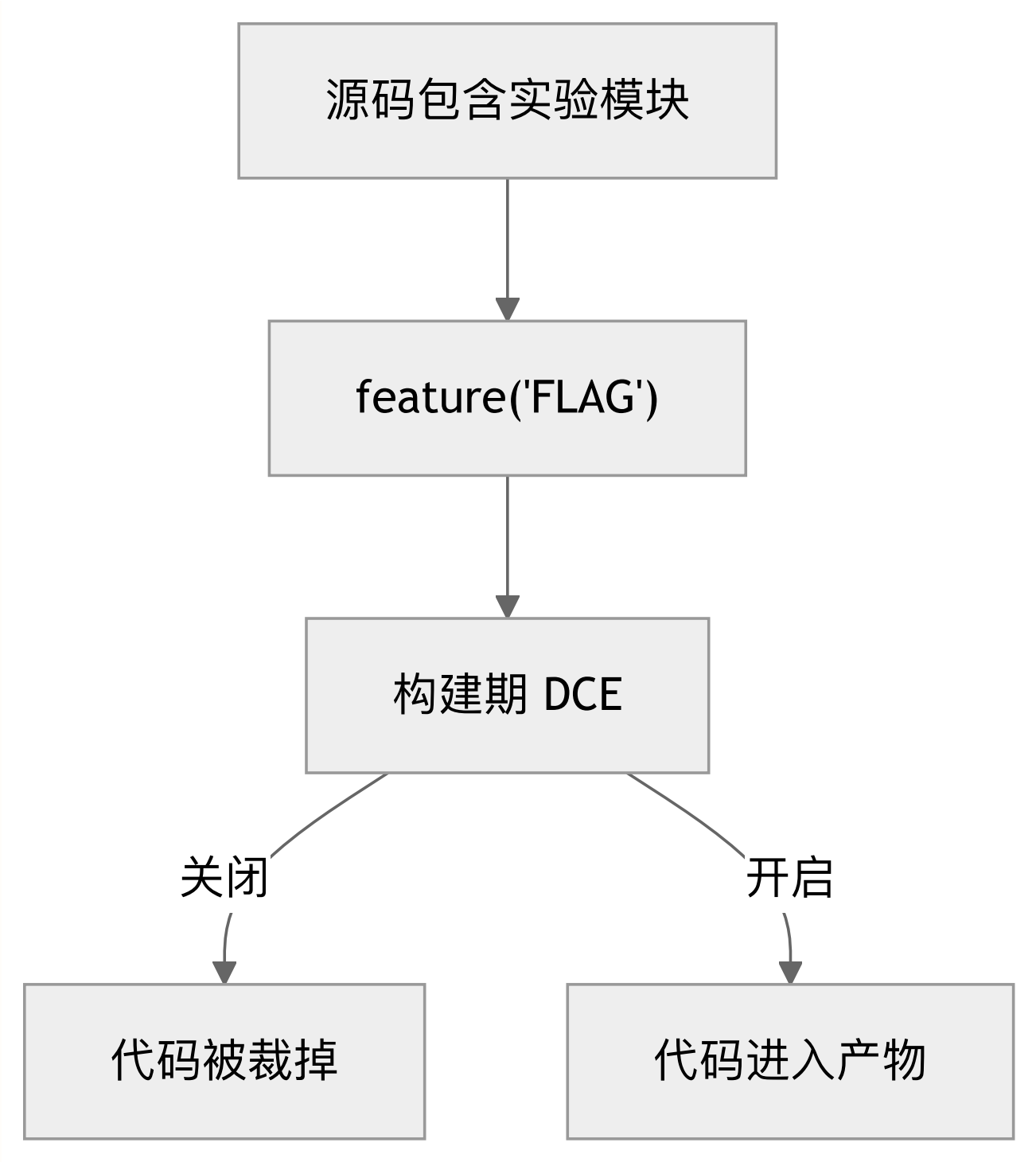
这一章先回答一个问题

为什么 Claude Code 的源码里到处都是 `feature('XXX')`、GrowthBook 配置和 cached gate？这到底是在增加复杂度，还是在控制复杂度？

答案是后者。对一个快速演化的产品来说，Feature Flag 不是“临时补丁”，而是让试验、灰度、组织策略和构建裁剪可以并存的基础设施。

37.1 编译时 gate：让某些代码根本不进产物

`feature()` 的一个重要用途，是配合 dead code elimination。像 `tools.ts`、`cli.tsx`、`QueryEngine.ts` 这类入口文件里，很多能力都是条件导入的。



这类 gate 特别适合：

- 构建体积敏感的功能
- 平台差异明显的功能
- 暂时不希望外部构建拿到的功能

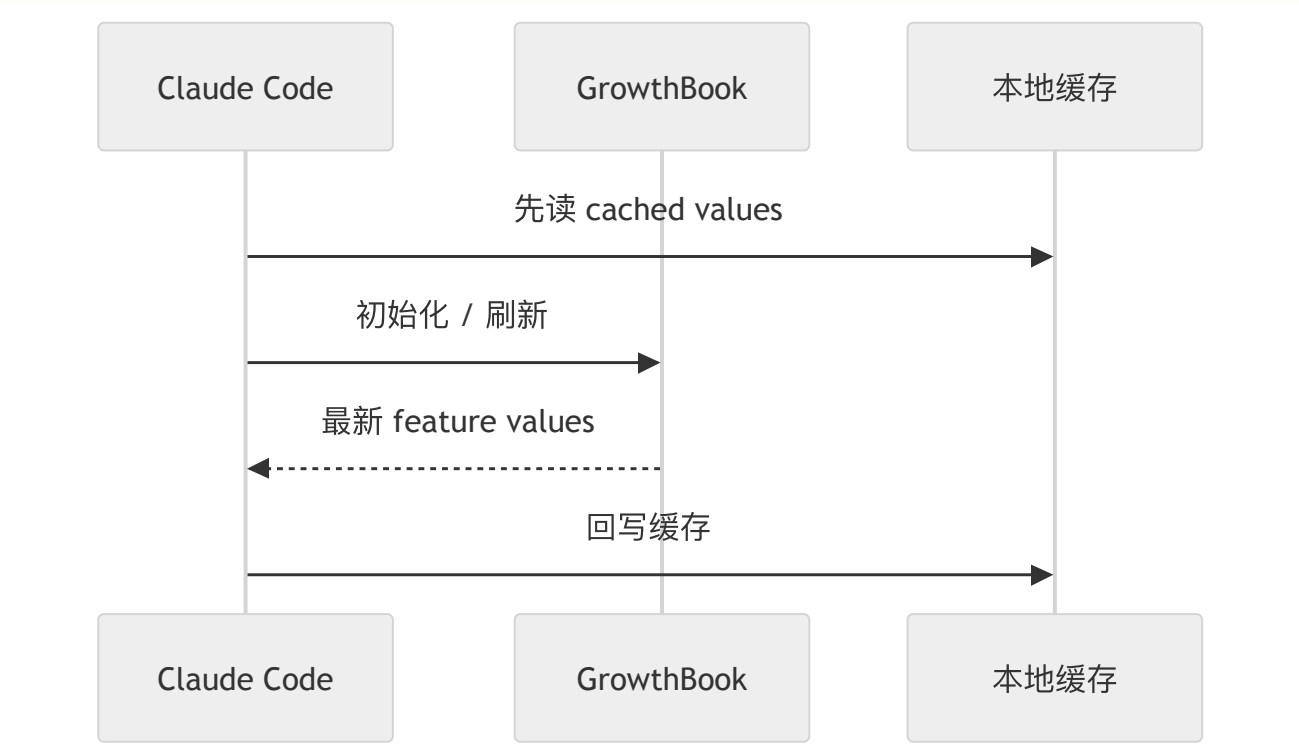
37.2 运行时 gate：GrowthBook 让“同一份代码”服务不同用户

`growthbook.ts` 展示的则是另一层：运行时特性读取。

它支持：

- 读取全部特性值
- 初始化客户端
- cached may be stale
- cached with refresh

- entitlement / security gate



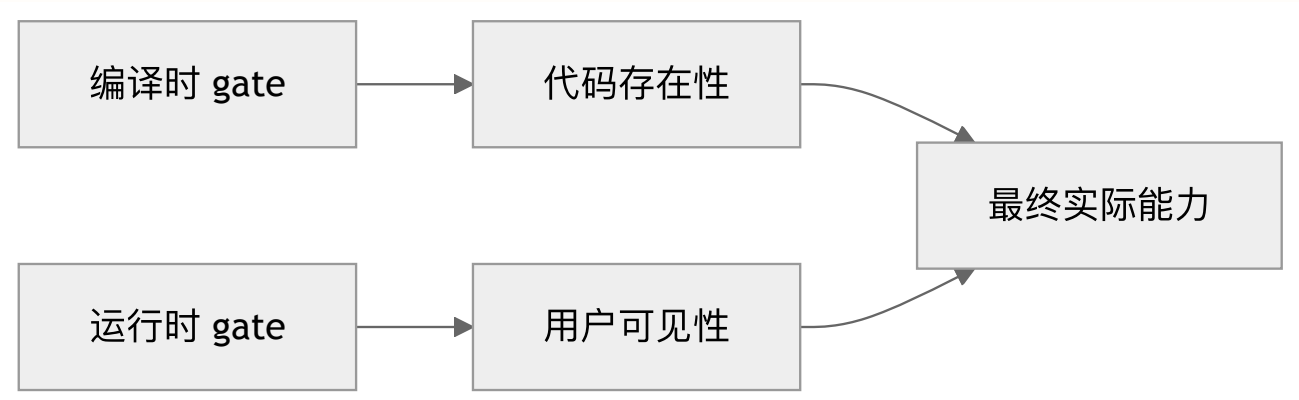
这让 Claude Code 能做到：同一个版本，不同用户、不同组织、不同时间看到的功能并不完全一样。

37.3 编译时 gate 和运行时 gate 为什么要并存

很多人会问：既然有 GrowthBook，为什么还要 `feature()`？

因为它们解决的问题不同：

类型	作用
编译时 gate	决定代码是否存在于产物中
运行时 gate	决定存在的代码是否对当前用户生效



只有两层都做，产品才既能瘦身，又能灰度。

37.4 Flag 的真正价值，是把产品版图写进代码里

你只要扫一遍 `tools.ts`、`ConfigTool`、`REPL`、`main.tsx`，就能感受到 Claude Code 的能力版图远远超过默认界面里能看到的那一点点。

从源码能明显看见几类方向：

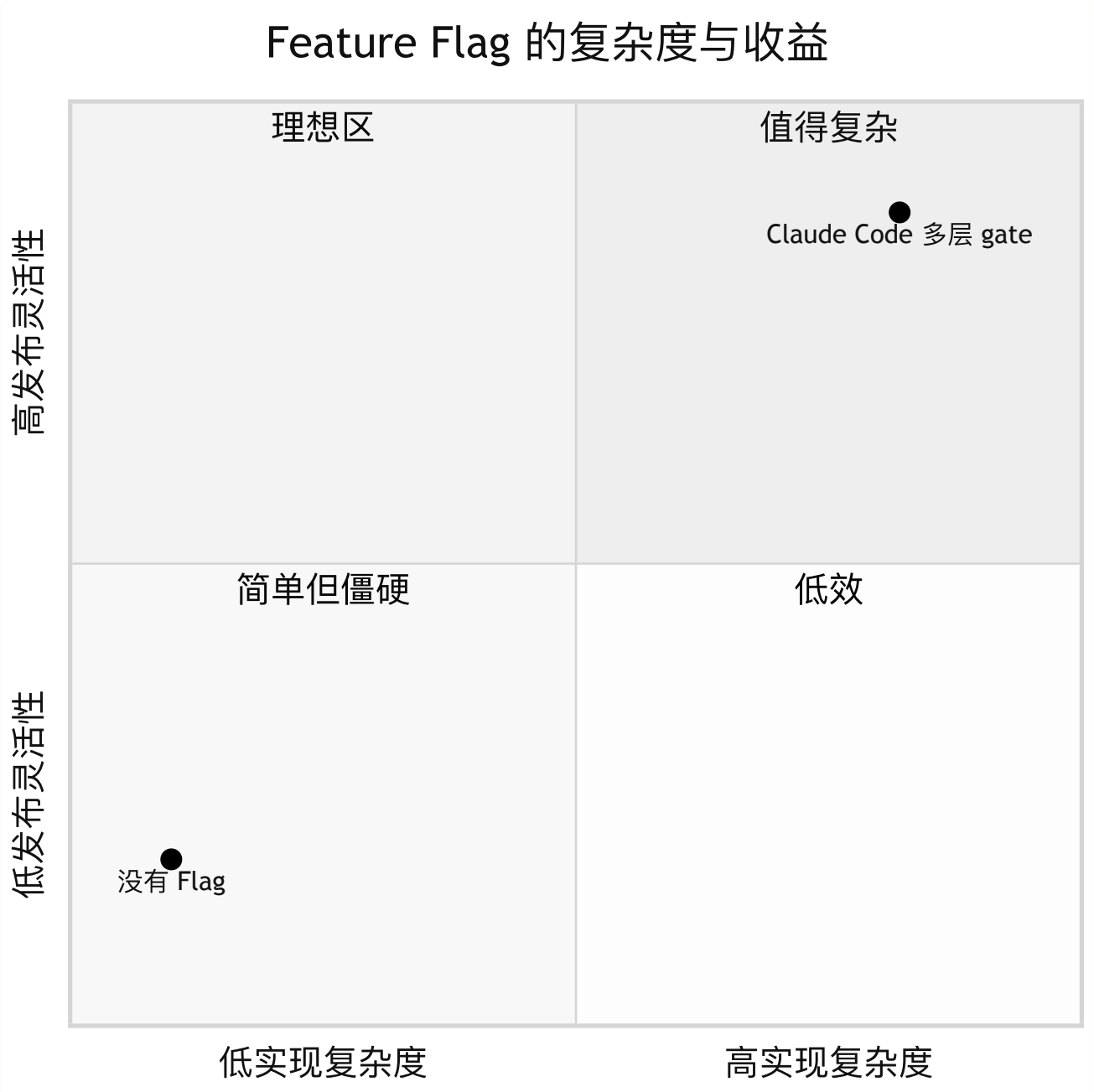
- 主动模式与触发系统
- 语音与多模态
- 监控与后台任务
- 多智能体与编排
- 实验性上下文管理

所以 Feature Flag 不是“技术债列表”，反而是产品路线图的侧写。

37.5 设计取舍：开关很多，会不会让系统更乱

会增加复杂度，但换来的收益也非常大：

- 降低一次性发布风险
- 允许平台差异
- 支持组织级策略
- 让实验不必污染默认体验



对这样一款快速迭代、能力跨度又极大的产品来说，这笔复杂度支出是划算的。

深水区（架构师选读）

Feature Flag 最成熟的用法，不是“留个 if 将来删”，而是把构建裁剪、灰度分发、组织门禁和 kill switch 统一成一套策略层。Claude Code 的 gate 体系正是在承担这个角色，所以它看起来复杂，但并不随意。

本章小结

Feature Flag 是 Claude Code 的演化基础设施。编译时 gate 决定代码有没有，运行时 gate 决定谁能用到它们，两者合起来才让试验和发布变得可控。

关键源码索引

- `tools.ts` 中的多处 gate：`tools.ts`
- `cli.tsx` 的 fast path 与条件分流：`cli.tsx`
- 全部 GrowthBook features 读取：`growthbook.ts`
- GrowthBook 初始化：`growthbook.ts`
- `CACHED_MAY_BE_STALE` 读取：`growthbook.ts`
- `CACHED_WITH_REFRESH` 读取：`growthbook.ts`

逆向提醒

“89 个开关”是研究过程中得到的能力版图印象，但具体数字会随版本和还原范围变化。比数字更重要的是读懂 gate 的结构和它们暗示的产品方向。

49

Hidden Features 第九编

第38章：彩蛋与前沿功能：Buddy、Voice 与 Computer Use

生活类比：游戏隐藏关卡

好游戏里总会藏一些只有细心玩家才会发现的机关。Claude Code 的源码里，也藏着一些不属于“默认主线”，但很能透露团队思路的功能。

这一章先回答一个问题

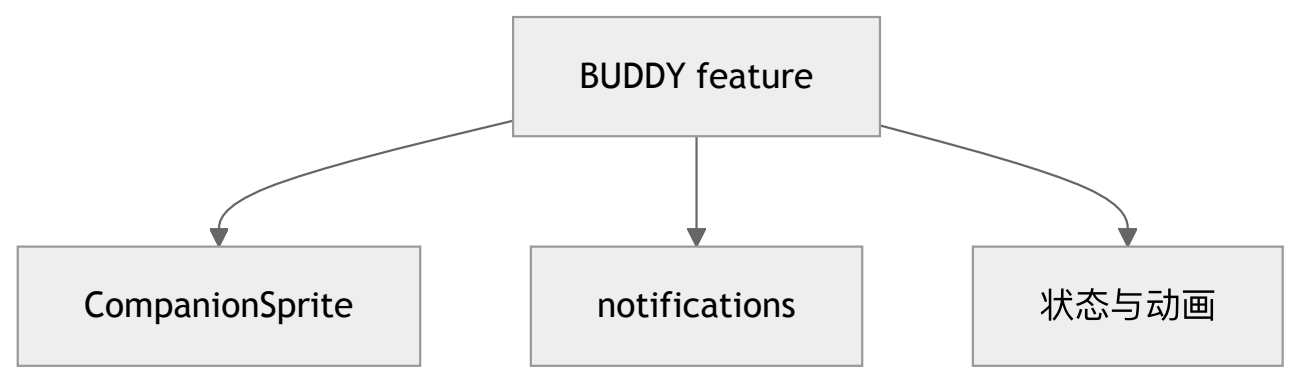
Buddy 小宠物、Voice 语音输入、Computer Use、Brief、Monitor 这些看起来五花八门的能力，究竟只是彩蛋，还是产品未来的试验田？

答案是两者都有。有些更偏趣味与陪伴，有些则是在提前验证未来的人机交互形态。

38.1 Buddy：一个看似轻松的功能，暴露了终端 UI 的野心

`src/buddy/` 目录里能看到完整的小伙伴体系，包括：

- `CompanionSprite.tsx`
- `companion.ts`
- `sprites.ts`
- `useBuddyNotification.tsx`

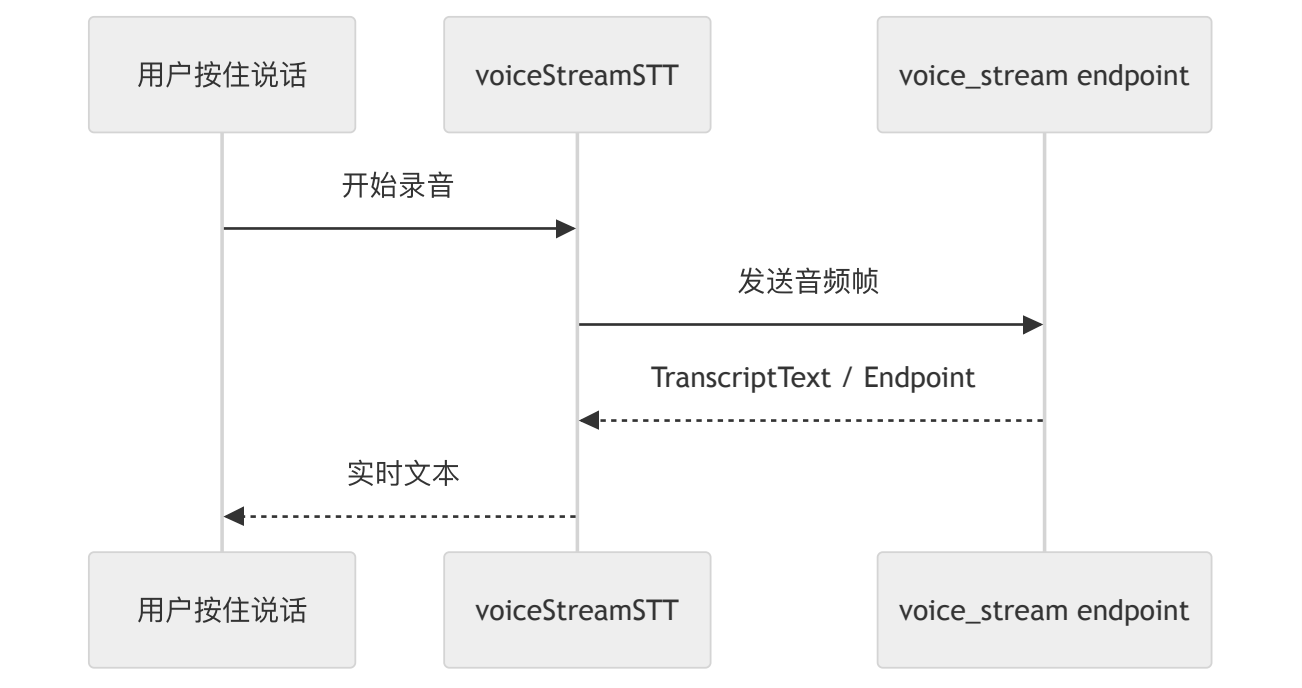


这件事看似只是彩蛋，但它说明了 Claude Code 的终端 UI 并不满足于“纯文本壳”，而是愿意探索更具陪伴感和反馈感的界面表达。

38.2 Voice：说明输入方式正在从键盘扩展出去

`voiceStreamSTT.ts` 非常直白地展示了一个语音流式转写客户端：

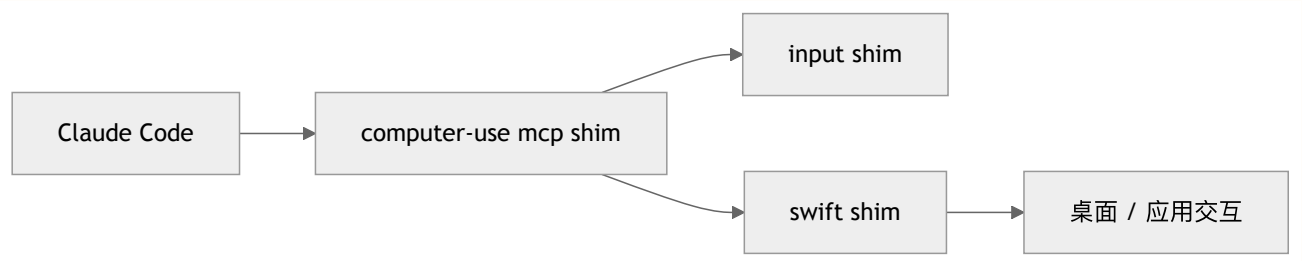
- 使用 WebSocket
- 走 OAuth 凭证
- 有 KeepAlive / CloseStream 协议
- 处理 transcript chunk 和 finalize



从设计角度看，这代表 Claude Code 不只在扩展“能做什么”，也在扩展“你怎么和它说话”。

38.3 Computer Use：从代码世界伸向桌面世界

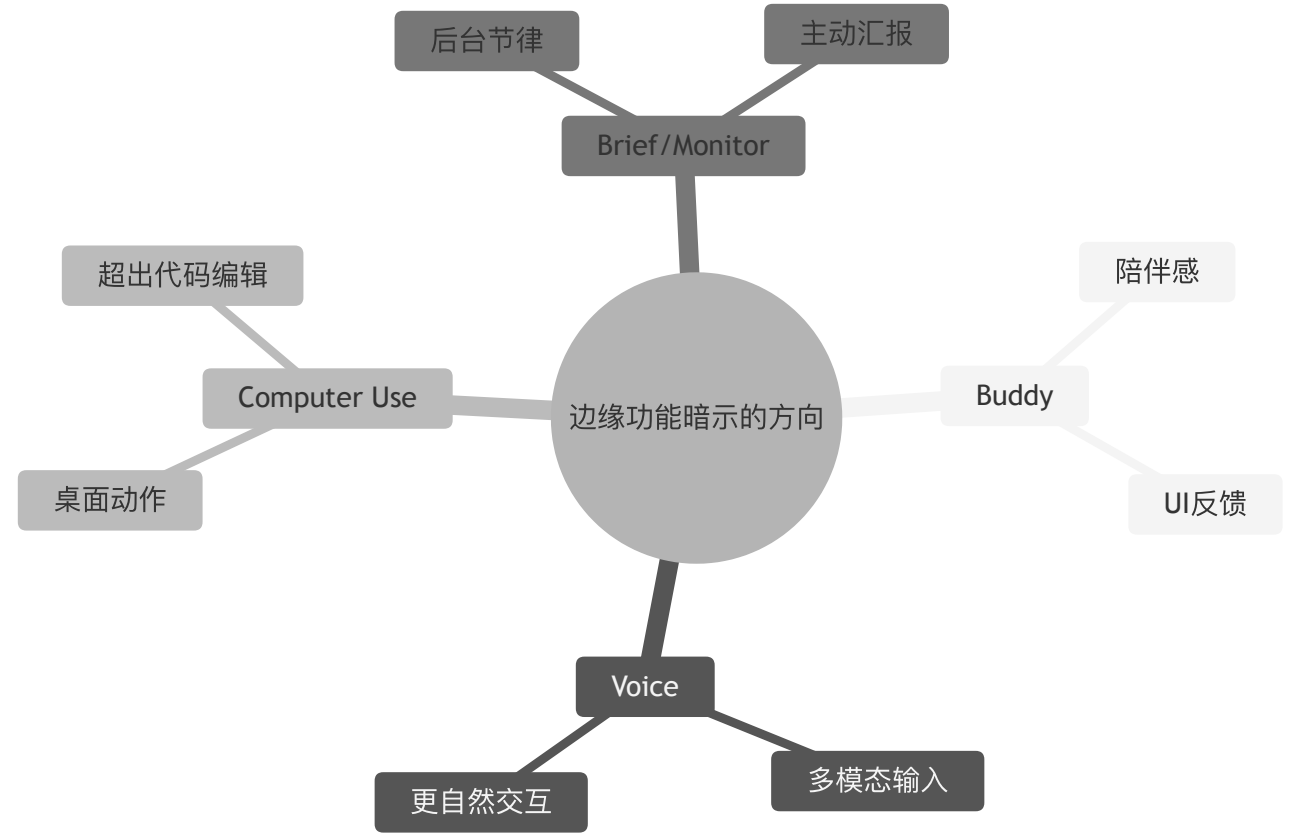
`shims/ant-computer-use-mcp`、`ant-computer-use-input`、`ant-computer-use-swift` 这些 shim，很明显不是普通代码编辑能力，而是在探索桌面与输入层控制。



这类功能一旦成熟，Claude Code 就不再只是“会改代码的 CLI”，而会开始触碰真实操作环境。

38.4 Brief / Monitor / Trigger：说明系统正在长出“后台节律”

`BriefTool`、`ScheduleCronTool`、`MonitorTool` 这些 gate 组合起来，很像一个更长期在线的代理系统骨架。



所以这章里真正重要的，不是“发现了多少彩蛋”，而是发现这些彩蛋在往同一个方向汇聚。

38.5 设计取舍：隐藏功能为什么值得认真看

它们往往更能说明团队的真实探索方向，因为这些能力尚未被市场话术打磨过，反而更原始、更直接。

深水区（架构师选读）

Buddy、Voice、Computer Use、Brief、Monitor 放在一起看，会得到一个非常清晰的方向图：Claude Code 正在从“文本式编程助手”向“多模态、后台化、主动式代理”试探。这些能力未必都成熟，但方向已经非常明确。

本章小结

彩蛋不是边角料。Buddy 透露 UI 野心，Voice 透露输入演化，Computer Use 透露能力边界正在扩张，Brief/Monitor 则透露系统正走向更持续的后台代理。

关键源码索引

- Buddy 组件：`CompanionSprite.tsx`
- Buddy 目录：`buddy/`
- Voice 流式 STT：`voiceStreamSTT.ts`
- Voice 模式设置痕迹：`supportedSettings.ts`
- Computer Use shim：`index.ts`
- Computer Use 输入 shim：`index.ts`
- REPL 中 Buddy / Voice 痕迹：`REPL.tsx`
- `BUDDY` 在 REPL 中的分支：`REPL.tsx`

逆向提醒

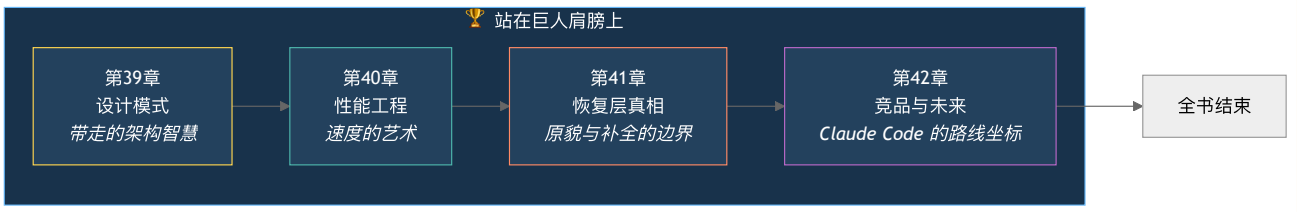
这一章覆盖的很多能力都带强烈门控或 shim 色彩。它们是“方向证据”非常有价值，但不能简单等同于“今天默认可用的稳定功能清单”。

50

第十编：站在巨人肩膀上

> *前面九编是在拆机器，这一编开始总结“这台机器为什么这样设计”。*
>
> 我们会从全局视角回看 Claude Code：提炼跨系统的设计模式，理解它为什么能在 CLI 里跑得又快又稳，分清还原层与补全层的边界，并最终把它放回更大的 AI 编程路线图里。

本编总览



本编四章速览

章	标题	核心问题	生活类比
39	设计模式	读完大规模源码，哪些通用工程心法最值得带走?	武术心法
40	性能工程	这么大的系统为什么还能做出秒启 CLI?	F1 赛车调校
41	恢复层真相	哪些是官方原貌，哪些是逆向补全?	修复古画
42	竞品与未来	Claude Code 代表了哪种路线，下一步会走向哪里?	路线图与地图坐标

本编阅读目标

读完这一编，你不只会“看懂 Claude Code”，还会更清楚哪些设计值得复用、哪些结论需要谨慎，以及这套系统在更大 AI 编程版图中的位置。

51

Patterns 第十编

第39章：设计模式：带走的架构智慧

生活类比：武术心法

学会一套拳，不如看懂它背后的发力原理。读源码也一样，看完几万行实现之后，真正能带走的，是那些能迁移到别的项目里的通用模式。

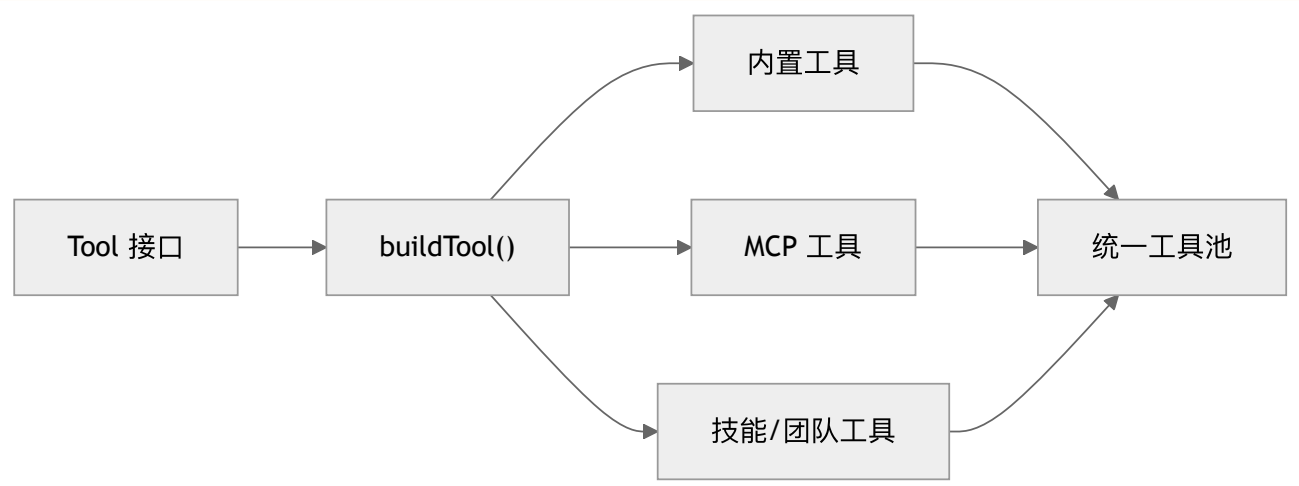
这一章先回答一个问题

如果把 Claude Code 所有具体业务都拿掉，只保留“值得抄走的工程思想”，最重要的会是哪几条？

这章不再追某个单文件，而是提炼横跨全书的几种模式：统一协议、分层治理、显式入口、条件装配、后台化运行与可恢复执行。

39.1 策略 + 工厂：先统一接口，再让能力自由增长

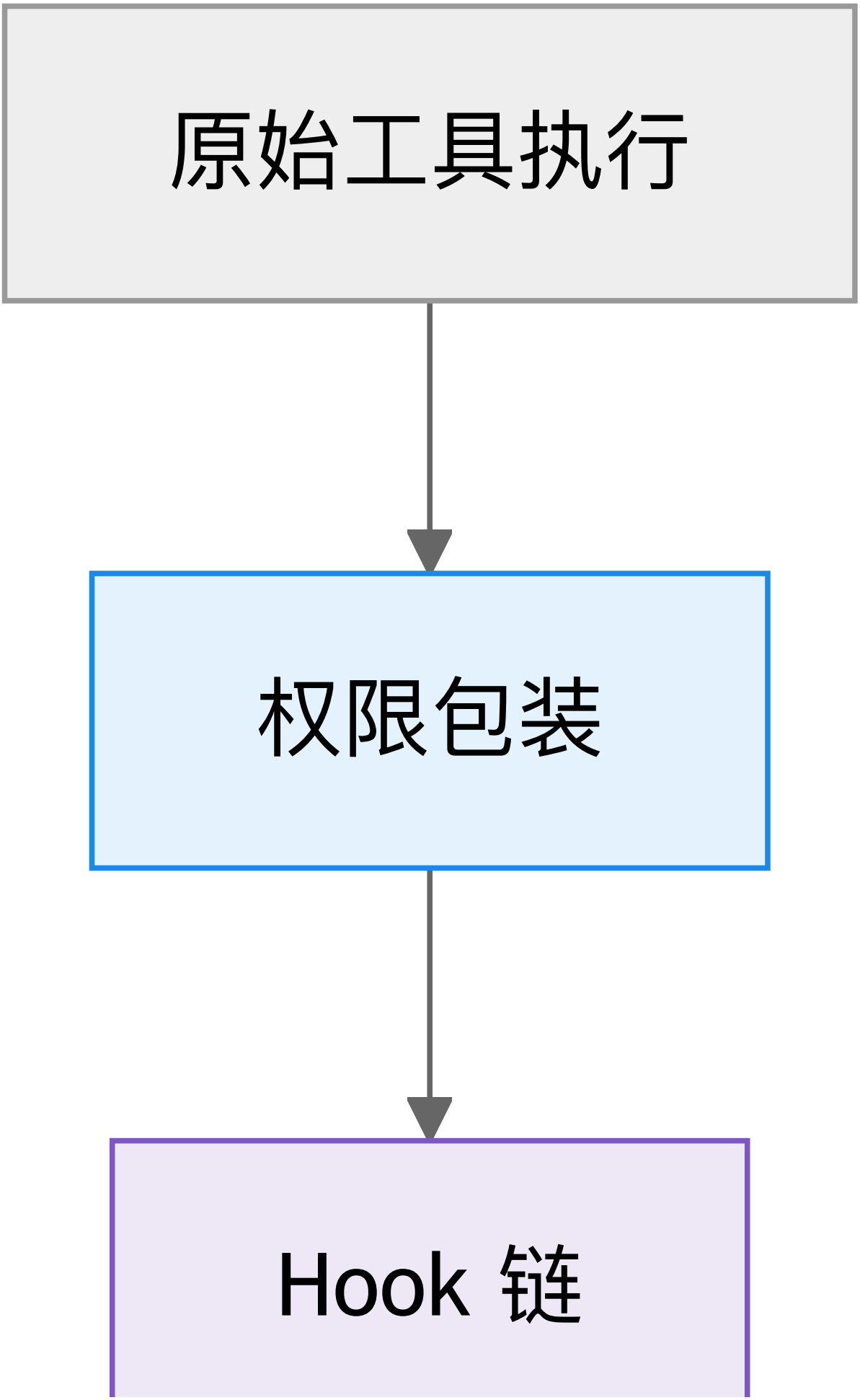
Claude Code 最鲜明的一条主线，就是工具系统。`Tool.ts` 先定义共同协议，`buildTool()` 再用工厂式方式统一装配，`tools.ts` 则把不同来源的工具拼成同一池子。

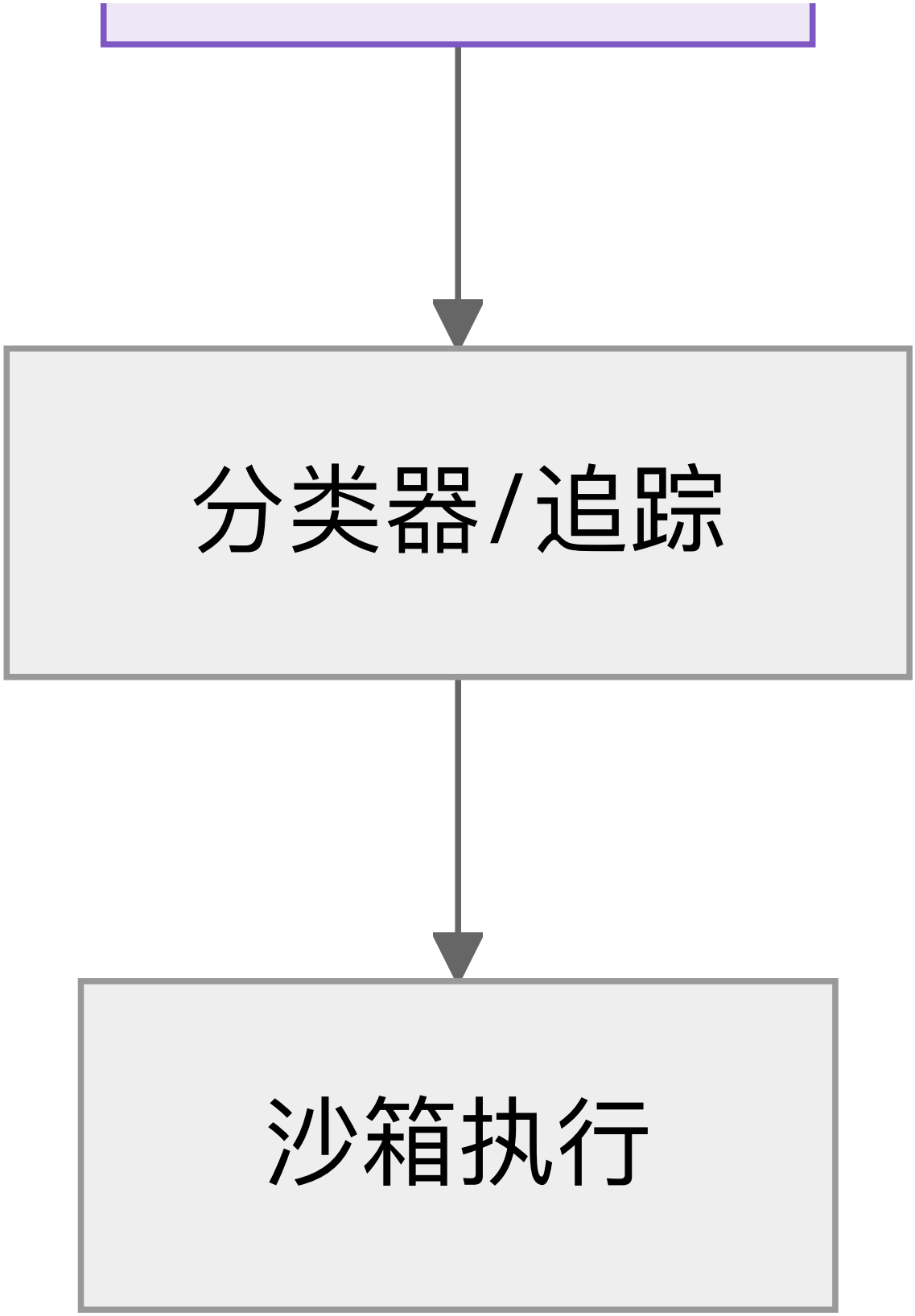


这套设计最值得学的地方是：先把入口统一，再允许行为多样。这样新增能力时，不必反复发明一套新框架。

39.2 装饰器 + 中间件：横切关注点不要长进业务里

权限检查、Hook、日志、分类器、沙箱这些能力，都不是某个工具的“核心业务”，却又必须发生在执行路径上。Claude Code 的处理方式非常成熟：让这些横切关注点作为包装层和中间件链存在。


原始工具执行
权限包装
Hook 链



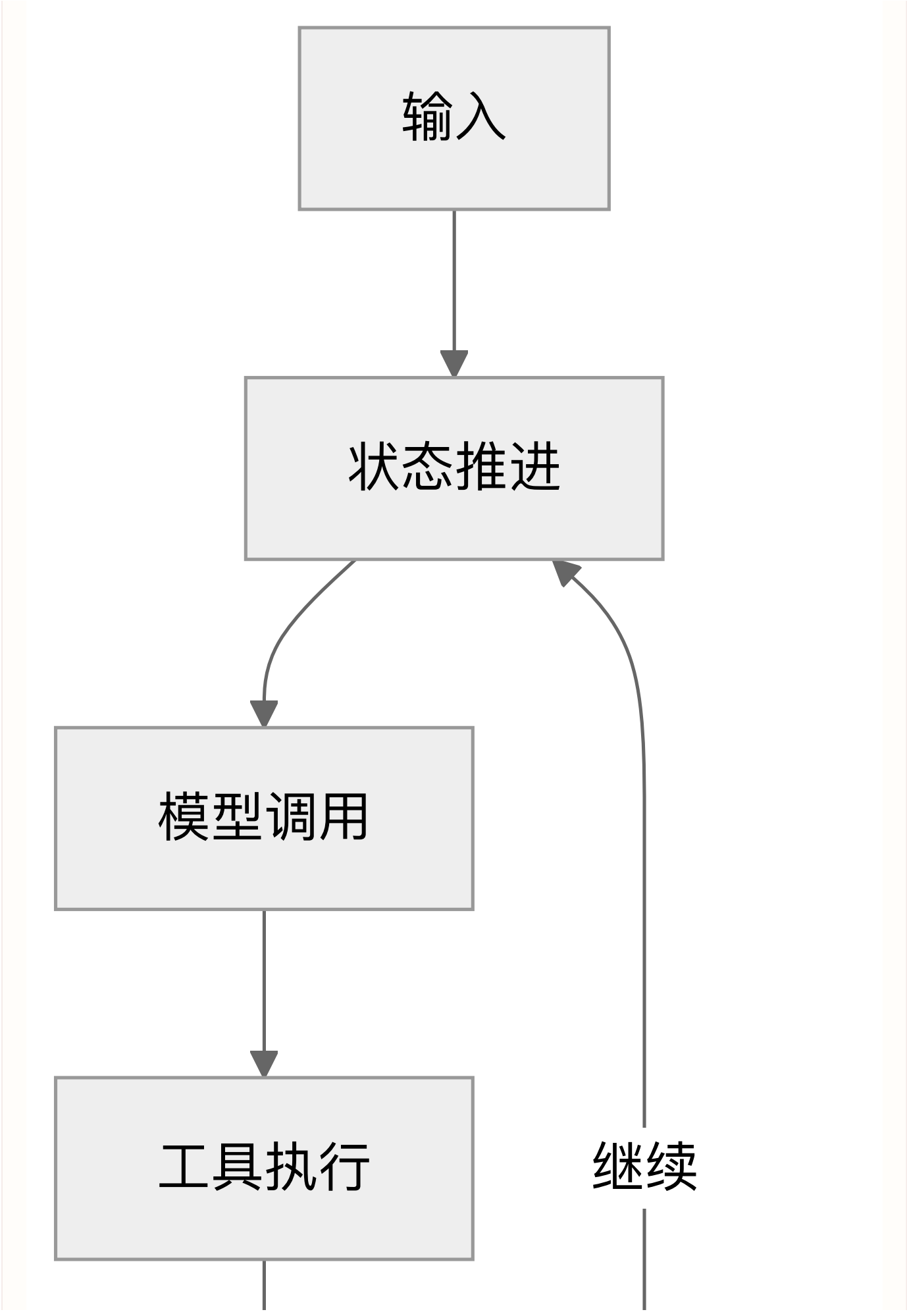
这让工具本身可以聚焦“做什么”，而不必背着一堆与业务无关的治理逻辑。

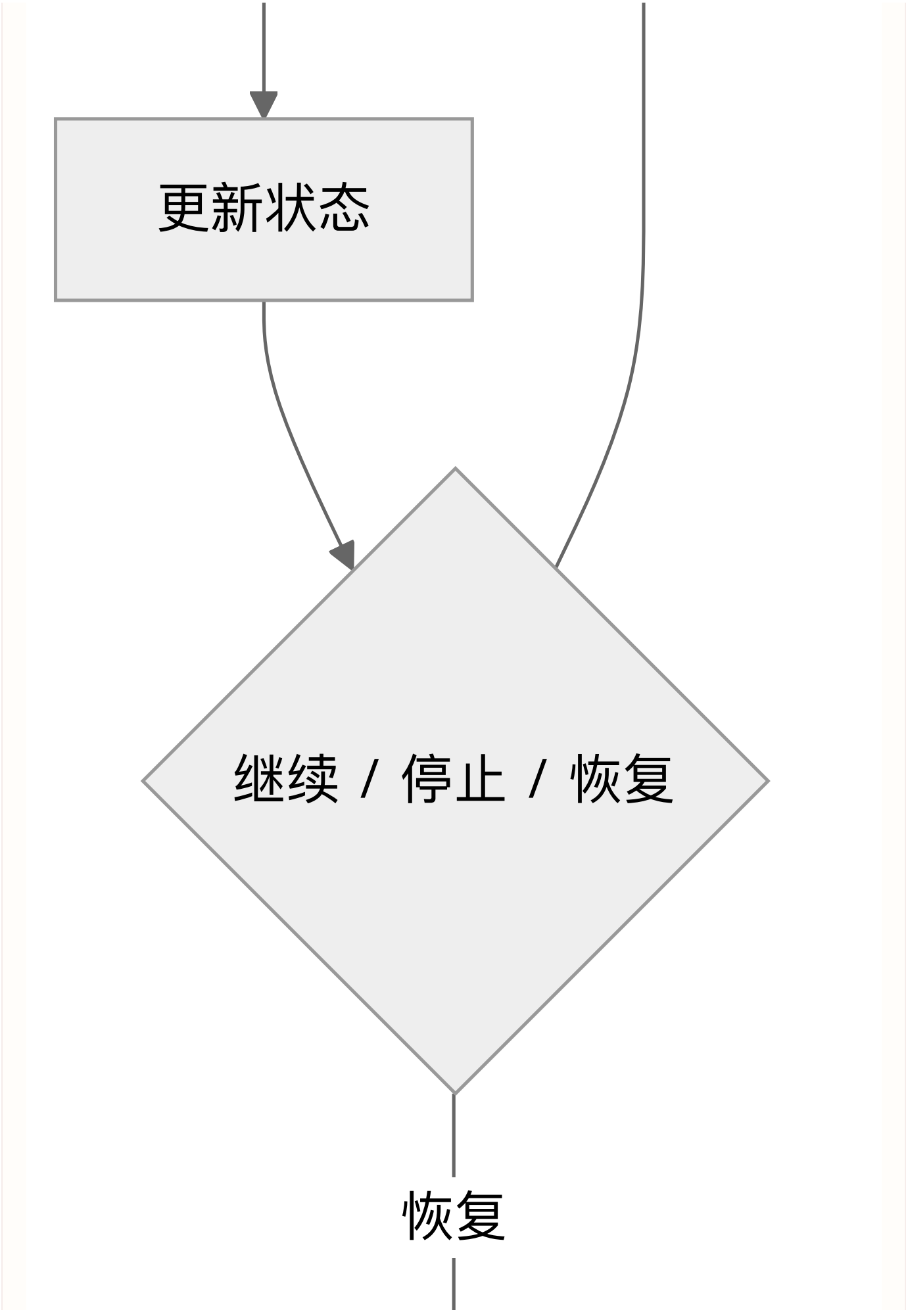
39.3 状态机 + 可恢复执行：让 Agent 不只是“会循环”

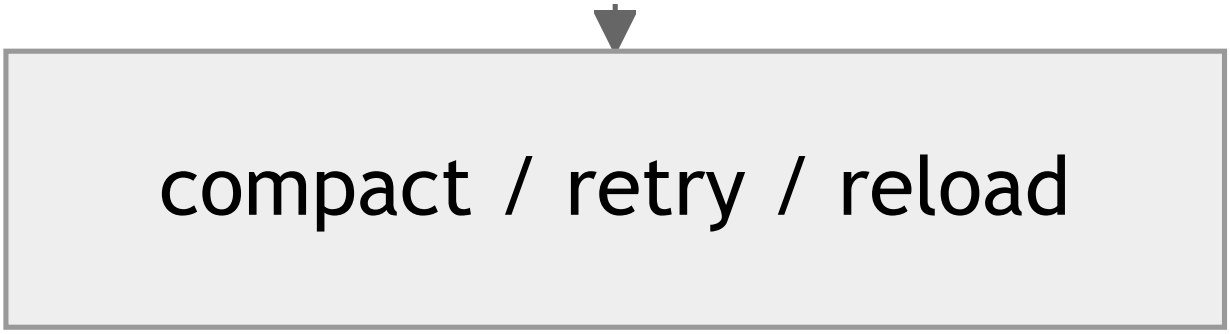
`query.ts` 和 `QueryEngine.ts` 里最珍贵的，不是有个 `while (true)`，而是：

- 有明确状态
- 有退出条件
- 有错误恢复

- 有压缩边界
- 有继续执行的入口


输入
状态推进
模型调用
工具执行
继续


更新状态
继续 / 停止 / 恢复
恢复

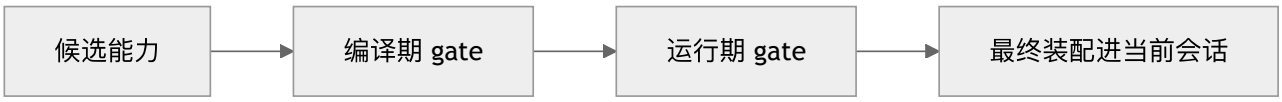


真正好的 Agent 系统，不是“能跑起来”，而是“中断之后还能接着跑”。

39.4 条件装配：功能很多，但默认体验不能失控

Claude Code 的大量 `feature()`、GrowthBook 和动态加载，说明它大量使用了条件装配模式：

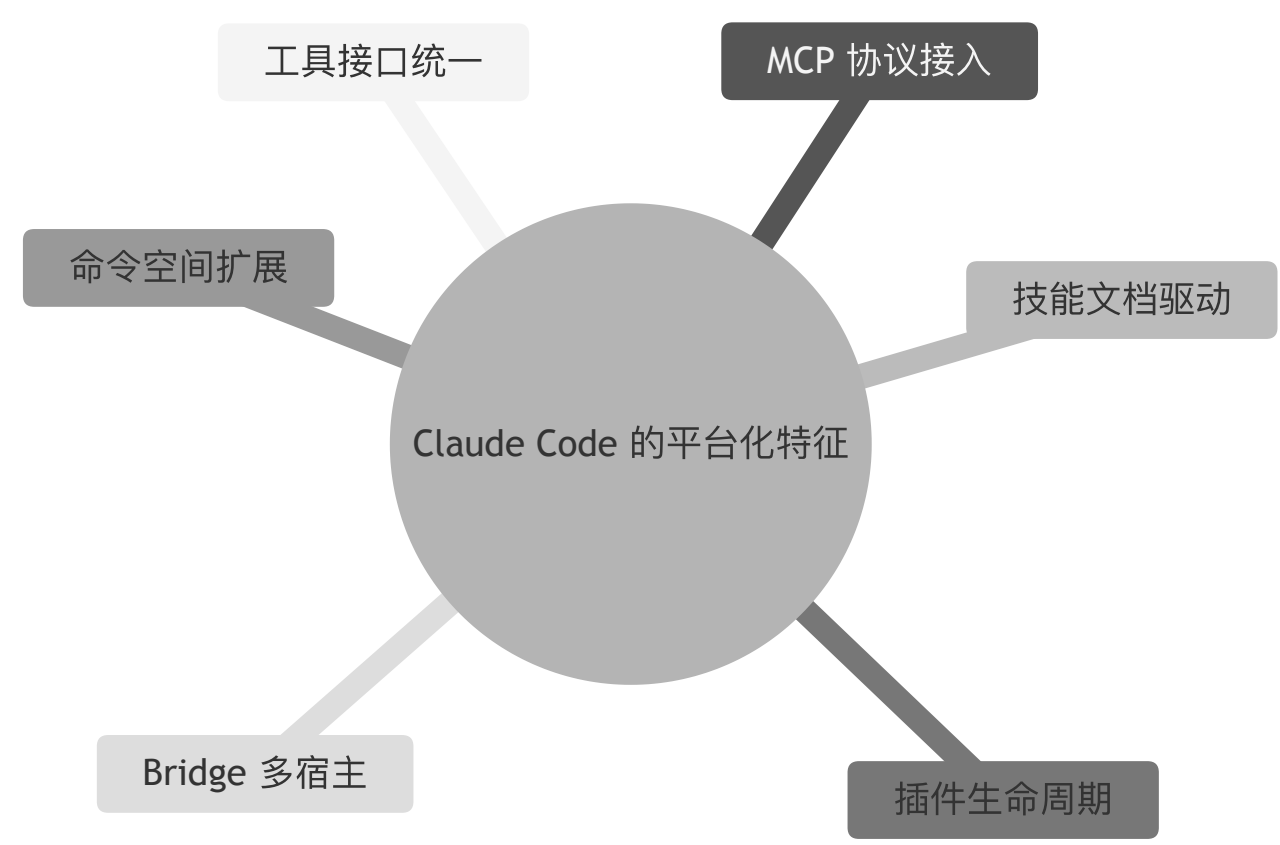
- 某些能力只在特定构建进入
- 某些能力只在特定用户/组织启用
- 某些命令和工具只在当前上下文暴露



这比“所有能力永远全部打开”更克制，也更适合长期演化。

39.5 平台模式：Claude Code 不是只想做产品

MCP、技能、插件、命令系统这些组合在一起，说明 Claude Code 不是只想做一个“内置功能足够多的 CLI”，而是正在长成一个平台。



这类架构最难得的不是某个功能多强，而是它开始具备“承载别人能力”的能力。

39.6 本章心法：六条值得带走的工程原则

1. 先统一接口，再扩展能力。
2. 让横切关注点离开业务核心。
3. Agent 必须是状态机，而不是一次性函数。
4. 可恢复性和可继续性比“首次执行很聪明”更重要。
5. 平台能力必须回到同一治理框架中。
6. 大系统不要害怕门控，要害怕无门控扩张。

深水区（架构师选读）

Claude Code 真正迷人的地方，不是它用了某个时髦 AI 技术，而是它把大量经典软件工程思想重新放进 Agent 场景里：统一协议、横切治理、状态机、平台化、条件装配、可恢复执行。这些思想不会随模型迭代过时。

本章小结

如果只带走一句话，那就是：Claude Code 的优秀并不神秘，它把传统软件工程里的好模式，用一种非常现代的方式重新组合进了 Agent 产品。

关键源码索引

- 工具协议：`Tool.ts`
- 工具池装配：`tools.ts`
- Agent 主循环：`query.ts`
- 会话引擎：`QueryEngine.ts`
- 命令系统：`commands.ts`
- Bridge 分层：`bridgeMain.ts`

逆向提醒

设计模式是本书的分析视角，不是官方术语。它们的价值在于帮助你迁移工程思想，而不是要求你逐字对应每个 GoF 名称。

52

Performance 第十编

第40章：性能工程：为什么它能在终端里跑得这么顺

生活类比：F1 赛车调校

一辆快车不是只靠发动机大，而是靠每个细节都不浪费。Claude Code 的启动性能也是这样做出来的。

这一章先回答一个问题

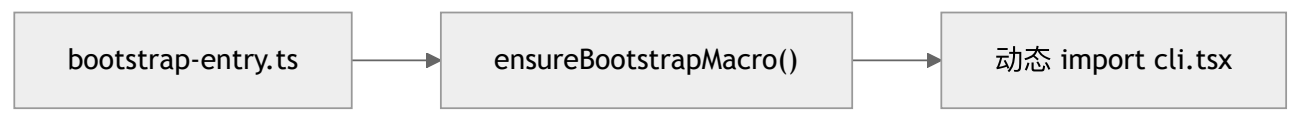
一个接近两千个 TypeScript 文件、带 UI、带工具、带认证、带桥接、带插件的系统，为什么还能做成一个日常使用不拖泥带水的 CLI?

因为它在启动链、导入方式、预取、缓存和构建策略上都做了非常激进的优化。

40.1 第一招：入口链尽可能薄

`bootstrap-entry.ts` 只有三件事：

- 确保 bootstrap macro
- 立刻进入 CLI 入口
- 不提前加载重模块

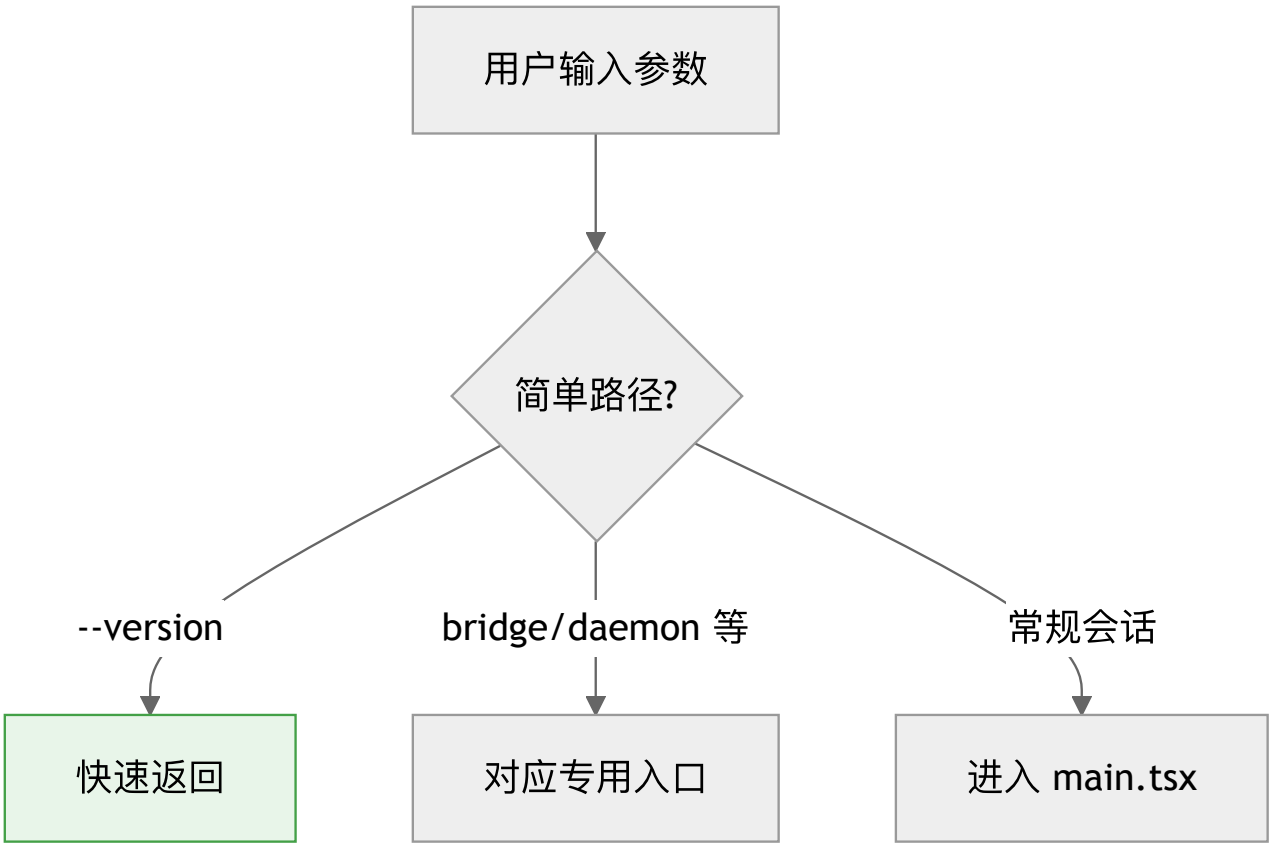


这么做的价值很直接：越早的入口越薄，越容易把后面的复杂度延后。

40.2 第二招：`cli.tsx` 里塞满了 fast path

`cli.tsx` 明确写着：所有 import 都尽量动态化，`--version` 这种路径要做到几乎零额外加载。源码里还能看到很多分流点：

- `--version`
- bridge 路径
- daemon 路径
- templates 路径
- `--worktree --tmux` fast path

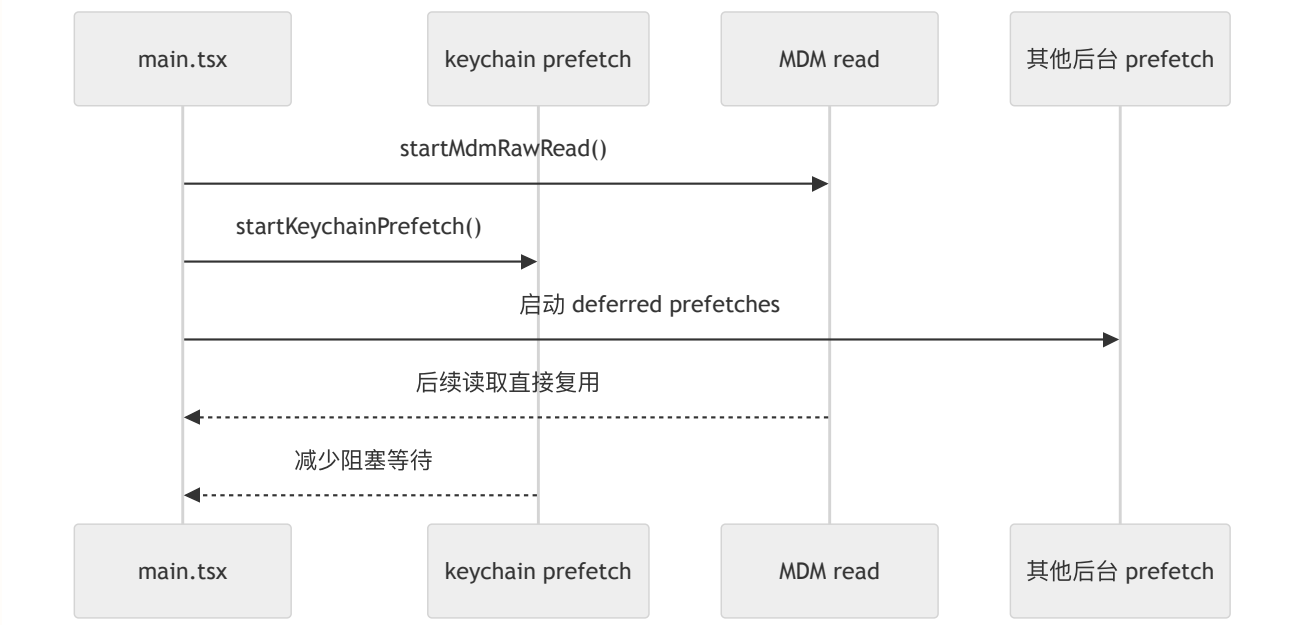


这是一种非常典型的 CLI 优化思路：别让最简单的请求为最复杂的系统买单。

40.3 第三招：重模块导入时并行做预取

`main.tsx` 开头的注释几乎可以当成性能工程教科书：

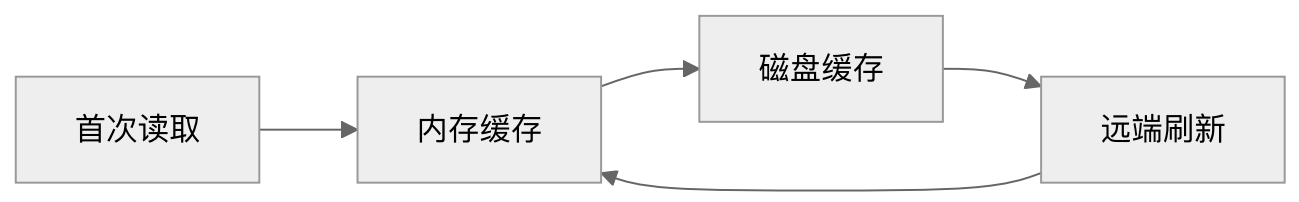
- MDM 子进程提早启动
- keychain 预取提早启动
- 一批认证与系统上下文预取在后台展开
- MCP 资源预热被延后到安全时机



这里最值得学的地方是：**用户还在想要输入什么时，系统就已经开始把大概率要用到的东西准备起来了。**

40.4 第四招：把“缓存”做成多层，而不是单点

从 GrowthBook、metrics、Grove、MCP resource prefetch 这些模块都能看到，Claude Code 非常喜欢“内存缓存 + 磁盘缓存 + 后台刷新”的组合。

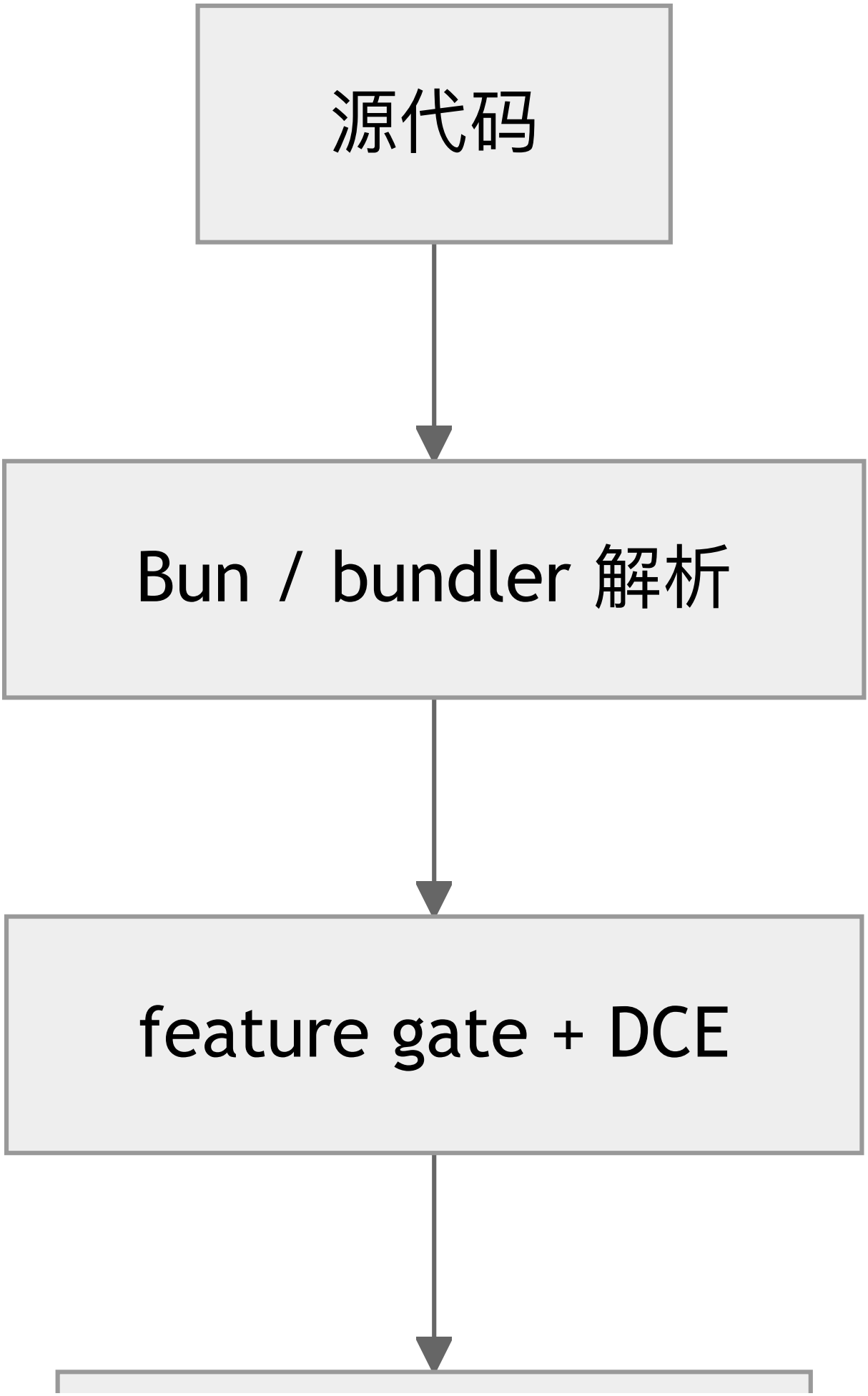


这类设计对 CLI 特别重要，因为 CLI 的一个核心痛点就是“每次重开进程都像重新冷启动一次”。

40.5 第五招：构建系统主动为裁剪服务

`package.json`、`tsconfig.json` 和源码里的 `feature()` 一起构成了一个很强的构建层策略：

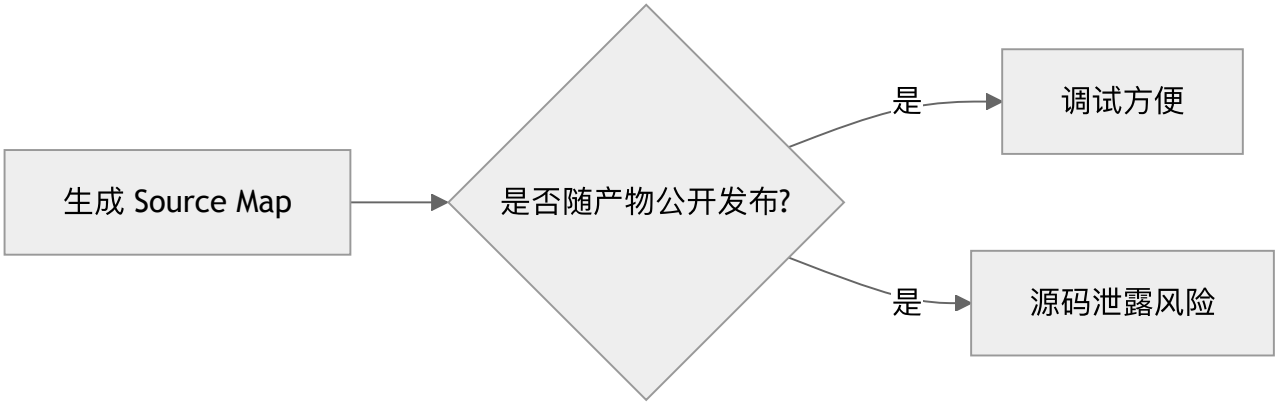
- `moduleResolution: bundler`
- 大量 feature gate
- shims/vendor 显式纳入编译边界


源代码
Bun / bundler 解析
feature gate + DCE

这说明性能工程并不只是运行时调优，也包括“从构建时就别把无关代码带进来”。

40.6 Source Map 事件给的教训：性能和安全必须一起看

`OpenClaudeCode/package.json` 里直白写着：这是“reconstructed from source maps”的 restored tree。对我们读者来说，这是礼物；对任何产品团队来说，这是严肃的发布与构建教训。



构建安全和性能工程看似是两回事，实际上都属于“发布产物控制”的一部分。

🔭 深水区（架构师选读）

Claude Code 的性能工程告诉我们一个很现实的道理：大型 CLI 想顺滑，不靠一招鲜，而靠入口瘦身、参数分流、动态导入、并行预取、多层缓存和构建裁剪一起发生。任何单点优化都救不了系统级迟钝。

本章小结

Claude Code 能快，不是因为它“小”，而是因为它在每一层都避免了不必要的等待：薄入口、快路径、后台预取、多层缓存和构建裁剪缺一不可。

关键源码索引

- 最薄入口：`bootstrap-entry.ts`
- `cli.tsx` fast path 说明：`cli.tsx`
- `--version` fast path：`cli.tsx`
- bridge / daemon / worktree 分流：`cli.tsx`
- `main.tsx` 顶层预取注释：`main.tsx`
- deferred prefetches：`main.tsx`
- `package.json` restored 标记：`package.json`
- `moduleResolution: bundler`：`tsconfig.json`

逆向提醒

我们能明确看到大量性能策略和构建边界，但打包脚本与发布流水线的全部细节并不都在仓库里。因此，本章可以高度可信地分析“代码如何优化启动”，但无法完整还原 Anthropic 内部 CI/CD 全貌。

53

Recovery 第十编

第41章：恢复层真相：哪些是原貌，哪些是补全

生活类比：修复古画

修复师越高明，越会让补笔不刺眼；但做研究的人，恰恰必须知道哪里是原画、哪里是修复。读 OpenClaudeCode 也一样。

这一章先回答一个问题

当我们说“基于源码分析 Claude Code”时，哪些代码可以当成 Anthropic 原貌，哪些只能当成为了让仓库可运行而做的补全层？

这一章是全书最重要的边界提醒之一。因为如果不先分清证据层级，你会很容易把 shim 的取舍误判成官方设计。

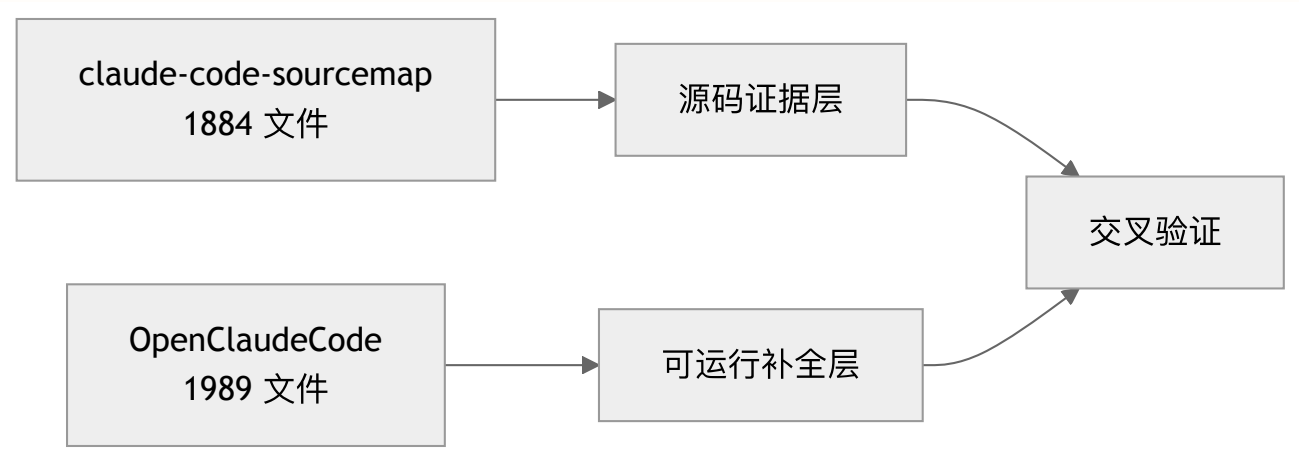
41.1 两套代码库不是竞争关系，而是证据互补关系

本书始终围绕两套材料展开：

- `claude-code-sourcemap/restored-src/src`：更接近还原层
- `OpenClaudeCode/src`：更接近可运行重建层

我们刚刚重新核对过文件数：

- 还原层 `src` 下 `ts/tsx` 文件：1884
- OpenClaudeCode `src` 下 `ts/tsx` 文件：1989
- `shims/` 文件：16
- `vendor/` 文件：4



这组数字本身就说明：OpenClaudeCode 不是单纯复制，而是为了可运行多补了一层东西。

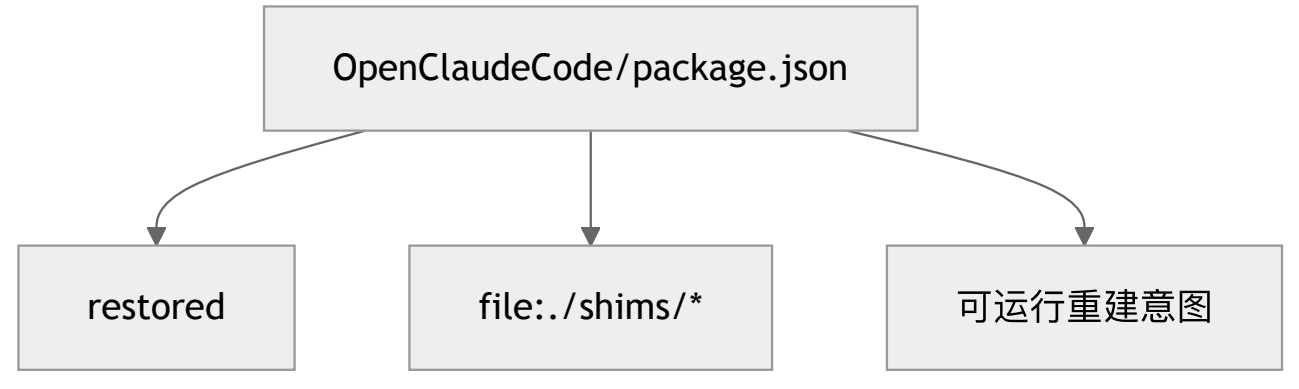
41.2 `package.json` 已经把“补全层身份”写明了

`OpenClaudeCode/package.json` 直接写着：

- 版本：`999.0.0-restored`
- 描述：`Restored Claude Code source tree reconstructed from source maps.`

再加上它把多个依赖指向本地 `file:./shims/...`，这几乎是明示：

- 这里有大量还原代码
- 也有明确的兼容和补全代码

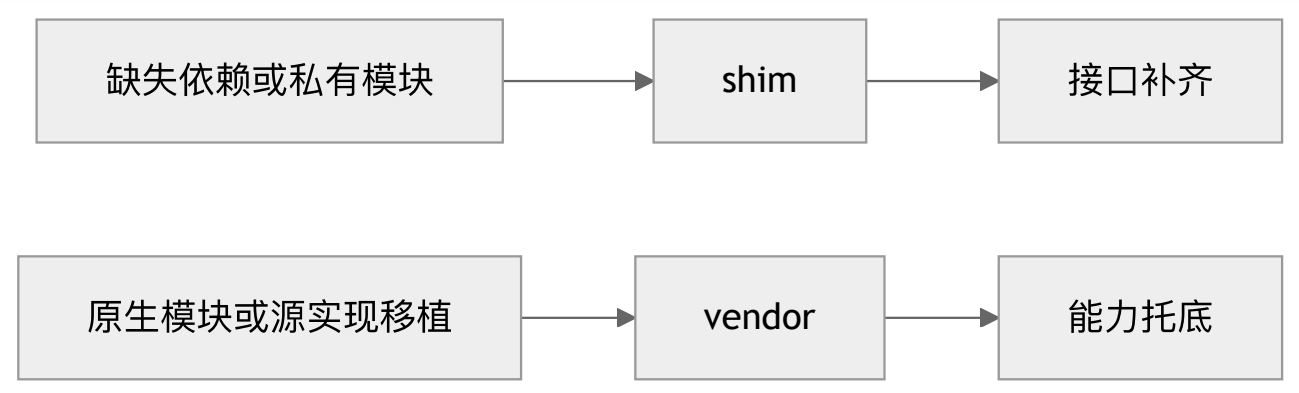


从写书方法论上说，这比任何猜测都更直接。

41.3 `shims/` 与 `vendor/` 的角色不同，别混为一谈

这两个目录都不属于“核心还原源码”，但性质不同：

目录	更像什么	典型作用
`shims/`	兼容补片	补齐缺失模块接口，让仓库可跑
`vendor/`	附带源实现	把原生/外部依赖的源代码搬进来或替代掉



所以不能简单地说“不是 src 就都不可信”。更准确的说法是：**可信度取决于它在证据链里的位置**。

41.4 哪些模块通常可以高信任，哪些必须降级看待

高信任区域通常有这些特征：

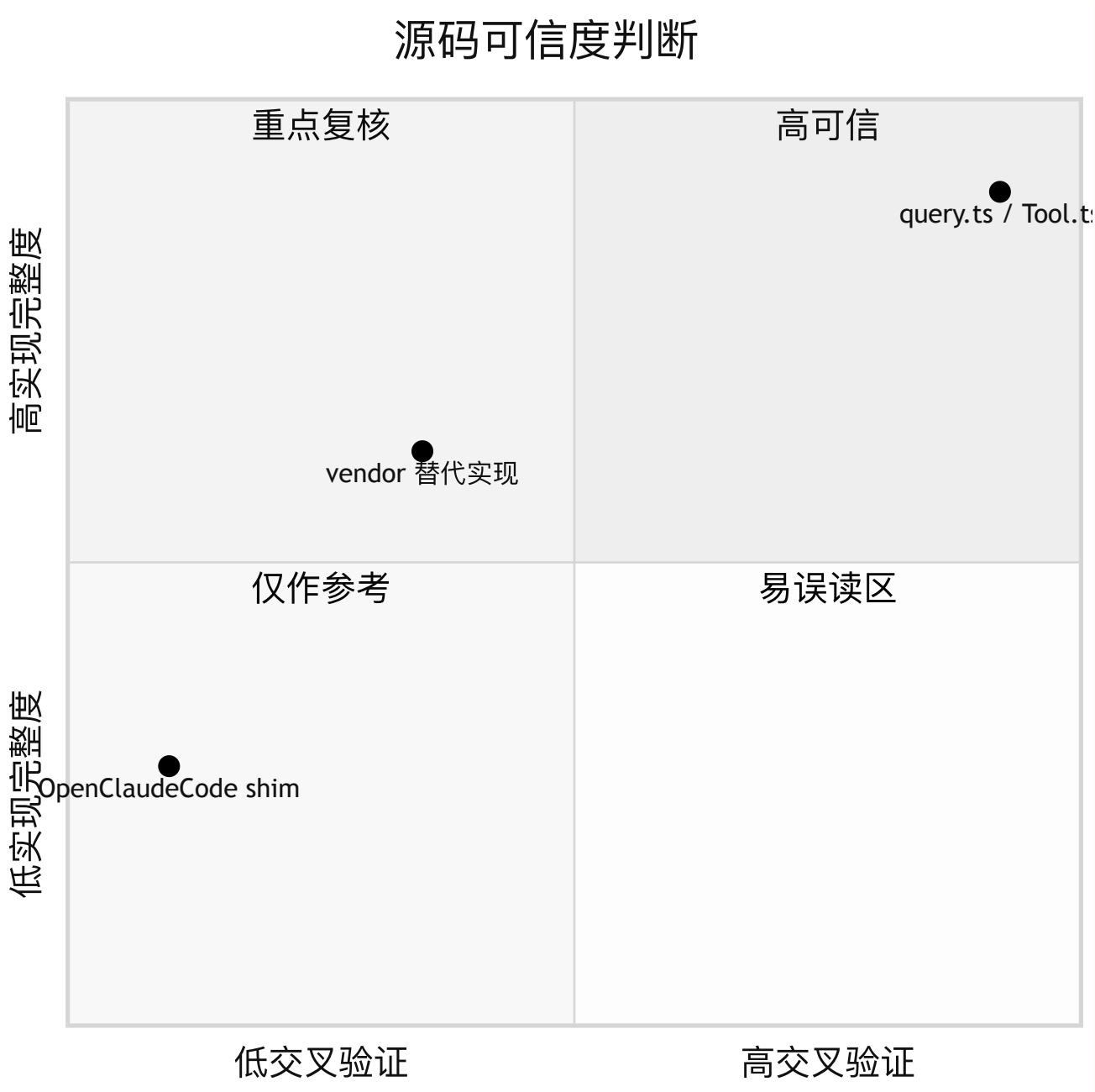
- 同时存在于 sourcemap restored 与 OpenClaudeCode
- 逻辑复杂、调用链完整、上下文自洽
- 没有明显的占位注释或空实现

典型如：

- `query.ts`
- `QueryEngine.ts`
- `Tool.ts`
- `tools.ts`
- 大量 UI 与状态管理代码

需要谨慎的区域通常是：

- `shims/`
- 与私有服务、原生能力强绑定的模块
- 只在 OpenClaudeCode 出现、在 restored 层找不到交叉验证的补片



41.5 这章最重要的方法论：别把“能跑”误当成“原作”

OpenClaudeCode 的价值非常大，因为它让我们：

- 看到更完整的运行形态
- 理解缺失模块怎么被补
- 验证某些调用链是否能通

但它不能自动回答“这是不是 Anthropic 当初就这么写的”。

🔭 深水区（架构师选读）

对逆向源码分析来说，最危险的不是信息不足，而是证据层混淆。还原层、补全层、shim、vendor、stub、fallback 这些概念如果不分开，所有结论都会被稀释。你越想写一本严肃的源码书，就越要先写清楚自己的证据边界。

本章小结

`claude-code-sourcemap` 和 `OpenClaudeCode` 必须双线阅读：前者更像原貌证据，后者更像运行补全。`shims/` 与 `vendor/` 的存在不是问题，问题是把它们误当作官方原始设计。

关键源码索引

- OpenClaudeCode restored 标记：`package.json`

- 本地 shim 依赖入口：`package.json`
- `vendor/` 与 `shims/` 编译边界：`tsconfig.json`
- 仓库说明中的 compatibility 提示：`AGENTS.md`
- shim 目录示例：`shims/`
- vendor 目录示例：`vendor/`

逆向提醒

这一章本身就是全书最大的“逆向提醒”。如果你引用的是 `shims/` 或只存在于 OpenClaudeCode 的兼容代码，必须明确标注它的证据等级，不能伪装成官方实现细节。

54

Future 第十编

第42章：竞品与未来：Claude Code 站在什么位置

生活类比：地图坐标

了解一座城市，不只要看它自己，还要看它位于哪条交通线上、和周边城市是什么关系。Claude Code 也是如此。

这一章先回答一个问题

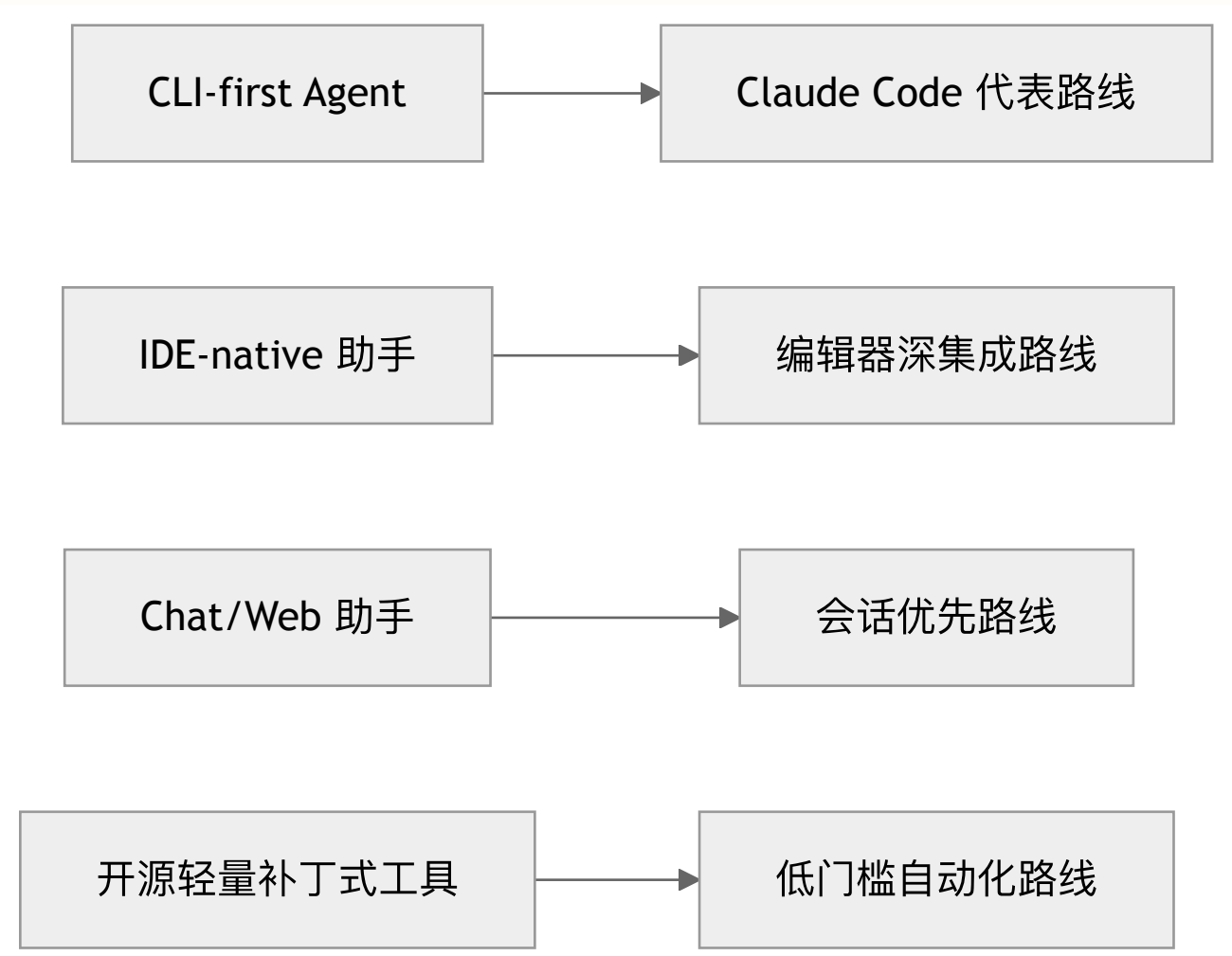
如果把 Claude Code 放到更大的 AI 编程路线图里，它究竟代表了哪种思路？它的源码又透露了哪些未来方向？

这一章不做“排行榜”，而做“路线比较”。重点不是谁赢谁输，而是不同产品在架构层面押了什么注。

42.1 四条常见路线里，Claude Code 更像“CLI-first 平台型 Agent”

从源码看，Claude Code 的路线非常鲜明：

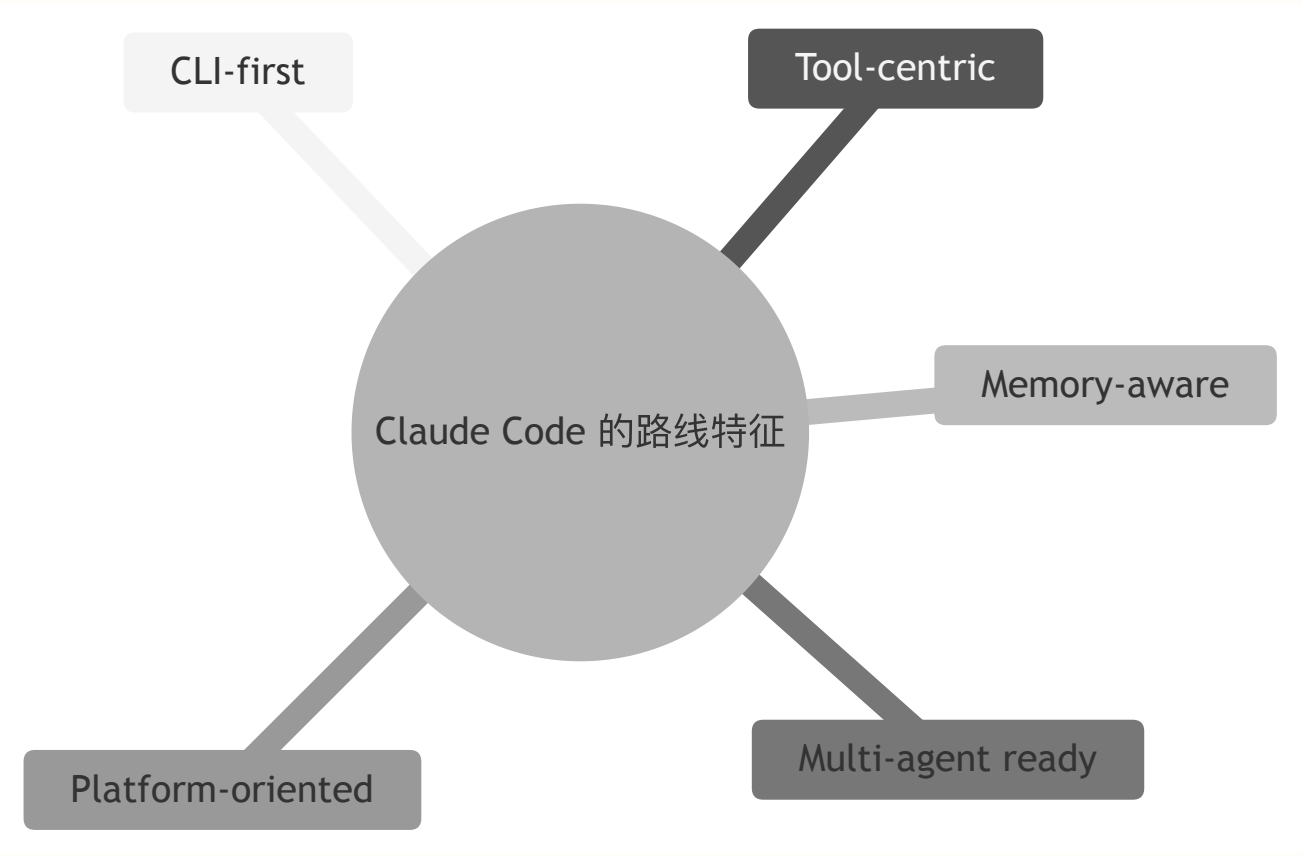
- CLI-first
- QueryEngine 为核心
- 工具池驱动
- Bridge 把能力送进 IDE
- MCP 让它长成平台



这让 Claude Code 在整个版图里的位置很独特：它不是最“像聊天”，也不是最“像 IDE 插件”，而是最强调**统一执行内核 + 外部宿主扩展**。

42.2 如果按架构维度看，它最突出的有五点

1. Agent Loop 做得深，而不是只做单步调用。
2. 工具与权限治理被放在核心位置。
3. 记忆与压缩是系统级设计，不是附属功能。
4. 多智能体被认真建模，而不是只做并行调用。
5. MCP、技能、插件说明它想做平台，而不只做产品。

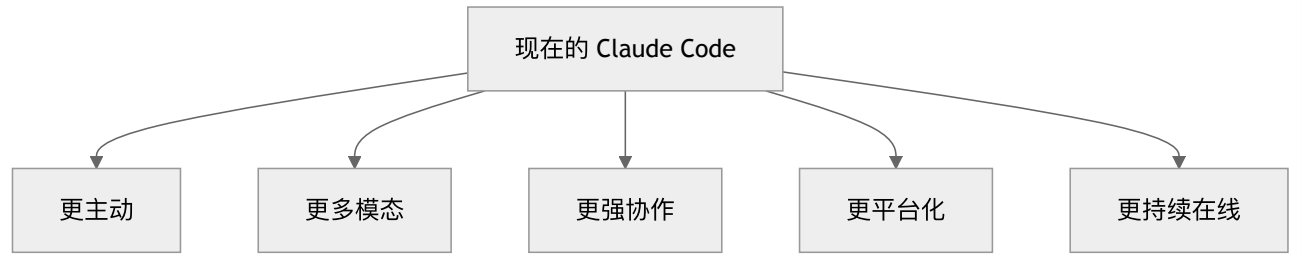


这些特征组合起来，决定了 Claude Code 更适合被理解为“工程代理运行时”，而不是“会写代码的聊天框”。

42.3 源码最明确泄露出来的未来方向

如果只看功能门面，你可能会低估它。但把全书证据串起来，未来方向其实很清楚：

- KAIROS：更主动
- VOICE_MODE：更多模态
- COORDINATOR_MODE：更强编排
- AGENT_TRIGGERS / MONITOR_TOOL：更持续的后台运行
- Computer Use 相关 shim：更接近真实操作环境

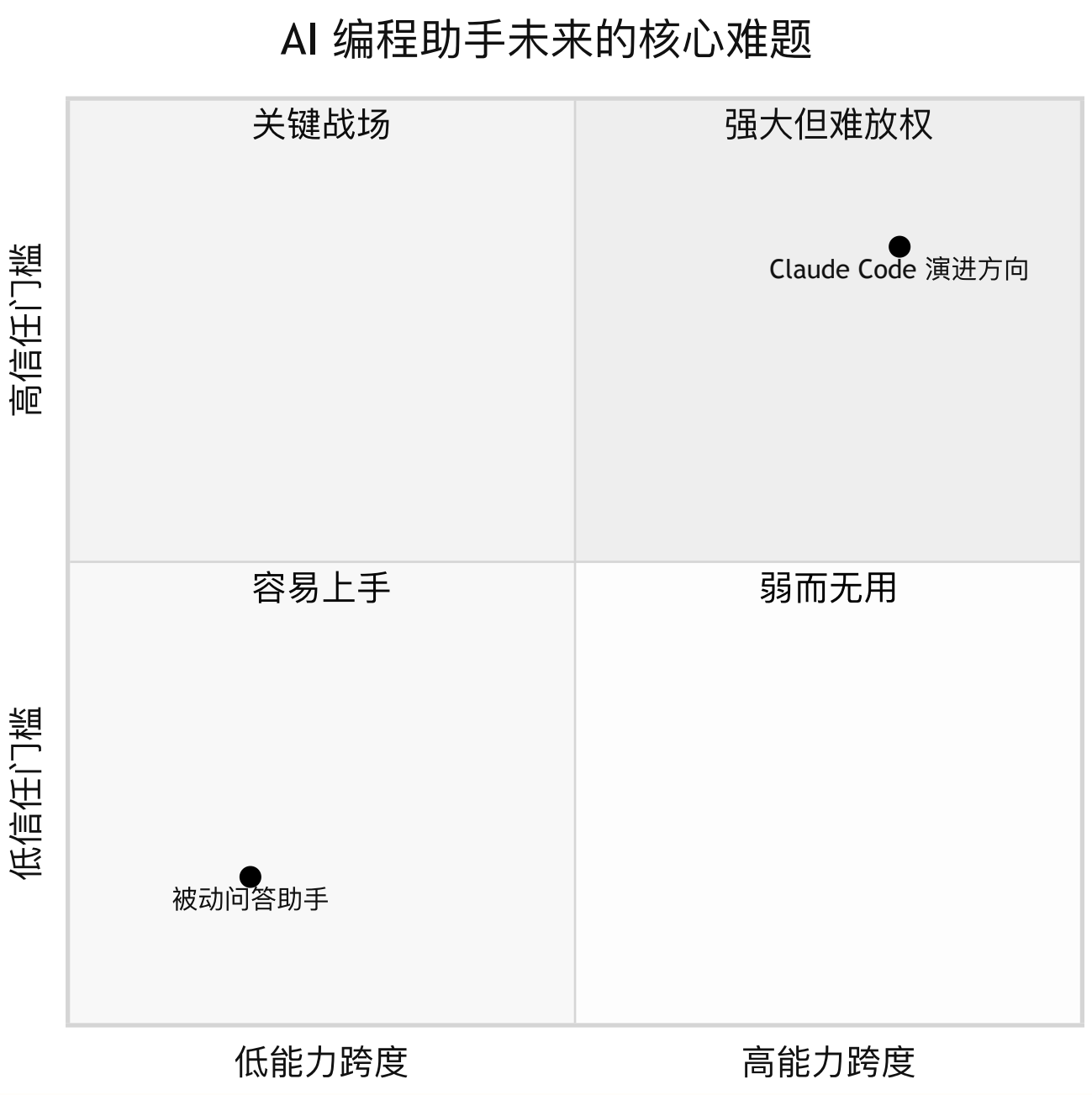


这五条线并不是互相独立，而是会慢慢汇合成一种更完整的“长期在线工程代理”形态。

42.4 未来真正困难的不是能力，而是信任

Claude Code 如果继续往前走，最难的问题不会是“能不能多做点事”，而会是：

- 用户愿不愿意让它更主动
- 团队愿不愿意把更多权限交给它
- 组织能不能接受它持续在线
- 多智能体是否仍可控、可审计、可恢复



所以真正的竞赛终点，不会只是模型更强，而是**谁能把能力、安全、记忆、编排和信任同时做平衡**。

42.5 全书落点：为什么值得读 Claude Code

因为 Claude Code 的源码不是只在回答“怎么接模型 API”，而是在回答一组更难的问题：

- AI 如何拥有工具
- AI 如何被约束
- AI 如何记住长期上下文
- AI 如何与其他 AI 协作
- AI 如何从单点产品演化成平台

深水区（架构师选读）

Claude Code 最值得研究的，不是它代表了“唯一正确路线”，而是它把当代 AI 编程助手最重要的几个难题都正面碰了一遍：工具、安全、记忆、协作、平台、主动性。哪怕你最后走另一条路线，这套源码也能帮你少踩很多坑。

本章小结

Claude Code 代表的是一种 CLI-first、平台型、强治理、强记忆、强协作的 Agent 路线。它未来最可能继续向主动化、多模态、后台化和平台化推进，而真正决定上限的，将是信任与治理能力。

关键源码索引

- 工具平台基准：`tools.ts`
- 记忆系统基准：`memdir/`
- 多智能体基准：`AgentTool/`
- Coordinator 方向：`coordinatorMode.ts`
- Feature Flag 方向图：`growthbook.ts`
- KAIROS 与主动能力入口：`tools.ts`

逆向提醒

这一章谈的是架构路线与趋势，不是最新市场情报。所有判断都基于本书分析到的源码与设计线索，而不是对外部产品的实时功能比拼。

55

附录

> *正文适合系统阅读，附录适合高频翻阅。*
>
> 这 5 个附录分别解决：**怎么判断证据可信度、怎么快速定位工具、怎么快速定位命令、如何读懂 Feature Flag 版图、如何分辨还原层与补全层。**

附录导航

使用建议

如果你想查……	先看哪里
一段代码到底多可信	附录A
某类工具在哪个目录	附录B
某个斜杠命令的大概位置	附录C
某个实验方向有哪些 gate	附录D
`sourcemap` 和 `OpenClaudeCode` 差异	附录E

附录定位

这部分不追求叙事性，而追求“翻得到、查得快、能回正文”。

56

方法论 附录

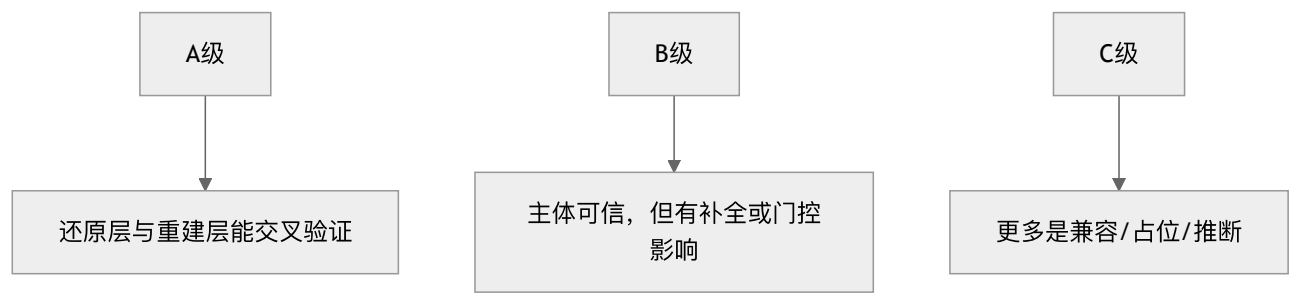
附录A：逆向方法与证据分级

这一附录给出本书统一使用的证据等级。它的目标很简单：**避免把“能跑”误读成“原貌”，避免把“有代码”误读成“已上线”。**

A.1 四类材料

材料	含义	典型位置
还原层源码	直接从 Source Map 恢复出的主要源码	`claude-code-sourcemap/restored-src/src/`
可运行补全层	以可运行、可研究为目标的重建树	`OpenClaudeCode/src/`
Shim	为缺失依赖补接口或兼容层	`OpenClaudeCode/shims/`
Vendor	为缺失原生/外部实现提供源替代	`OpenClaudeCode/vendor/`

A.2 证据等级



等级	说明	在本书中的写法
A	多处交叉验证，逻辑完整	可直接作为主要论据
B	主体可信，但局部需谨慎	会明确标注“可能受补全影响”
C	只作参考，不宜下结论	会明显加“推测/补全层”提醒

A.3 三步判断法

1. 先看它是不是存在于 `claude-code-sourcemap/restored-src/src/`。
2. 再看 `OpenClaudeCode/src/` 是否存在对应实现或调用链。
3. 最后看它是否被 `feature()`、GrowthBook 或 shim 包裹。

A.4 写作时的引用规则

- 引用 `src/` 的核心主链代码，默认优先用 A 级。
- 涉及 `shims/`、`vendor/`、隐藏 gate、实验入口时，必须加提醒。
- 任何“未来方向”判断都要基于多处证据，而不是单一 flag。

A.5 本书最常见的误区提醒

误区	正确做法
看到目录就当功能已上线	先看 gate 与主链接入
看到 OpenClaudeCode 代码就当 Anthropic 原貌	先问它是否是补全层
看到 shim 就完全忽略	它仍能揭示接口边界
只看单文件不看调用链	交叉验证 QueryEngine / tools / commands / settings

附录A结论

逆向阅读最重要的不是“看得多快”，而是“证据层分得清”。只有先把还原层、补全层、门控层和兼容层分开，后面的分析才站得住。

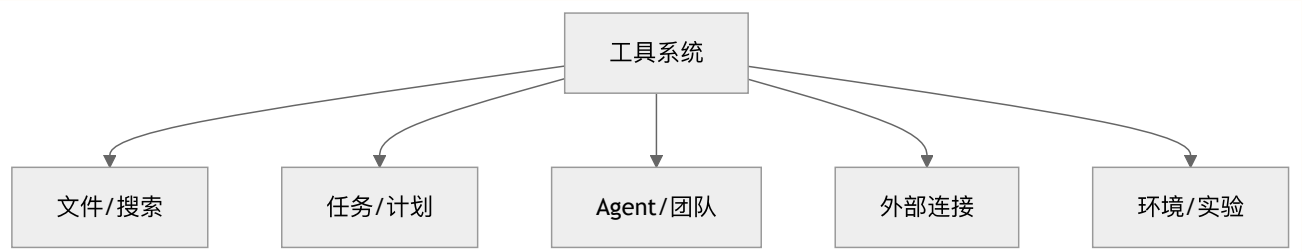
57

工具 附录

附录B：54 个工具速查手册

本书把 Claude Code 的工具体系理解为：**53 个顶层工具目录 + 1 个运行时 MCP 包装入口**。这就是“54 类工具”的由来。

B.1 工具总览图



B.2 按类别速查

类别	代表工具
文件与搜索	`FileReadTool` `FileEditTool` `FileWriteTool` `GlobTool` `GrepTool` `NotebookEditTool`
任务与计划	`TaskCreateTool` `TaskGetTool` `TaskListTool` `TaskUpdateTool` `TaskStopTool` `TaskOutputTool` `TodoWriteTool`
Agent 与团队	`AgentTool` `TeamCreateTool` `TeamDeleteTool` `SendMessageTool` `SkillTool`
外部连接	`MCPTool` `ListMcpResourcesTool` `ReadMcpResourceTool` `WebFetchTool` `WebSearchTool` `WebBrowserTool`
模式与环境	`EnterPlanModeTool` `ExitPlanModeTool` `EnterWorktreeTool` `ExitWorktreeTool` `ConfigTool`
平台/实验	`BriefTool` `SleepTool` `RemoteTriggerTool` `MonitorTool` `SnipTool` `WorkflowTool`

B.3 顶层工具目录清单

```
AgentTool
AskUserQuestionTool
BashTool
BriefTool
ConfigTool
DiscoverSkillsTool
EnterPlanModeTool
EnterWorktreeTool
ExitPlanModeTool
ExitWorktreeTool
FileEditTool
FileReadTool
FileWriteTool
GlobTool
GrepTool
LSPTool
ListMcpResourcesTool
MCPTool
McpAuthTool
MonitorTool
NotebookEditTool
OverflowTestTool
PowerShellTool
REPLTool
ReadMcpResourceTool
RemoteTriggerTool
ReviewArtifactTool
ScheduleCronTool
SendMessageTool
SendUserFileTool
SkillTool
SleepTool
SnipTool
SyntheticOutputTool
TaskCreateTool
TaskGetTool
TaskListTool
TaskOutputTool
TaskStopTool
```

```
TaskUpdateTool
TeamCreateTool
TeamDeleteTool
TerminalCaptureTool
TodoWriteTool
ToolSearchTool
TungstenTool
VerifyPlanExecutionTool
WebBrowserTool
WebFetchTool
WebSearchTool
WorkflowTool
shared
testing
```

补充说明：

- `MCPTool` 是运行时包装入口，会把外部 MCP server 暴露出来的工具也接成统一 Tool。
- `shared/` 与 `testing/` 更像支撑目录，但它们也属于工具层的一部分。

B.4 快速定位建议

想找什么	优先看哪里
工具统一协议	`src/Tool.ts`
工具池与启停逻辑	`src/tools.ts`
某个具体工具	`src/tools/<ToolName>/`
运行时外接工具	`src/services/mcp/` + `src/tools/MCPTool/`

附录B结论

工具系统最重要的不是工具数量，而是它们全部会回到同一协议、同一权限和同一主循环里。这正是 Claude Code 工具层真正强大的地方。

58

命令 附录

附录C：88 个命令速查手册

本地可见的命令目录统计为 87 个，再加上运行时合并进来的动态命令、技能命令或隐藏入口，构成了书中所说的“88 个命令级入口”。

C.1 命令分组思路

类别	示例
环境与登录	`login` `logout` `oauth-refresh` `privacy-settings`
项目与上下文	`add-dir` `context` `memory` `model` `config`
任务与协作	`plan` `tasks` `review` `share` `plugin` `skills`
系统与诊断	`doctor` `heapdump` `stats` `usage` `debug-tool-call`
隐藏与实验	`ctx_viz` `break-cache` `voice` `desktop` `mobile`

C.2 目录级命令清单

```
add-dir
agents
agents-platform
ant-trace
autofix-pr
backfill-sessions
branch
break-cache
bridge
btw
bughunter
chrome
clear
color
compact
config
context
copy
cost
ctx_viz
debug-tool-call
desktop
diff
doctor
effort
env
exit
export
extra-usage
fast
feedback
files
good-claude
heapdump
help
hooks
ide
install-github-app
install-slack-app
issue
keybindings
login
logout
mcp
memory
mobile
mock-limits
model
oauth-refresh
onboarding
output-style
passes
perf-issue
permissions
plan
plugin
pr_comments
privacy-settings
rate-limit-options
release-notes
```

```
reload-plugins
remote-env
remote-setup
rename
reset-limits
resume
review
rewind
sandbox-toggle
session
share
skills
stats
status
stickers
summary
tag
tasks
teleport
terminalSetup
theme
thinkback
thinkback-play
upgrade
usage
vim
voice
```

C.3 命令系统阅读入口

想看什么	入口文件
命令合并总入口	`src/commands.ts`
某个命令实现	`src/commands/<name>/`
技能命令如何进来	`src/skills/loadSkillsDir.ts`
MCP 命令如何进来	`src/commands.ts` 中 `getMcpSkillCommands`

C.4 使用这一附录的方法

- 想找某个 /命令 背后的目录，就先在上面按名字定位。
- 若目录有但界面没出现，优先检查 gate、可见性过滤和当前模式。
- 若命令来自技能或 MCP，可能不会在静态目录列表里占一席之地。

附录C结论

Claude Code 的命令系统不是固定菜单，而是“内置命令 + 技能命令 + 插件命令 + MCP 命令”的合并空间。目录列表只是静态入口，真正运行时的命令空间会更动态。

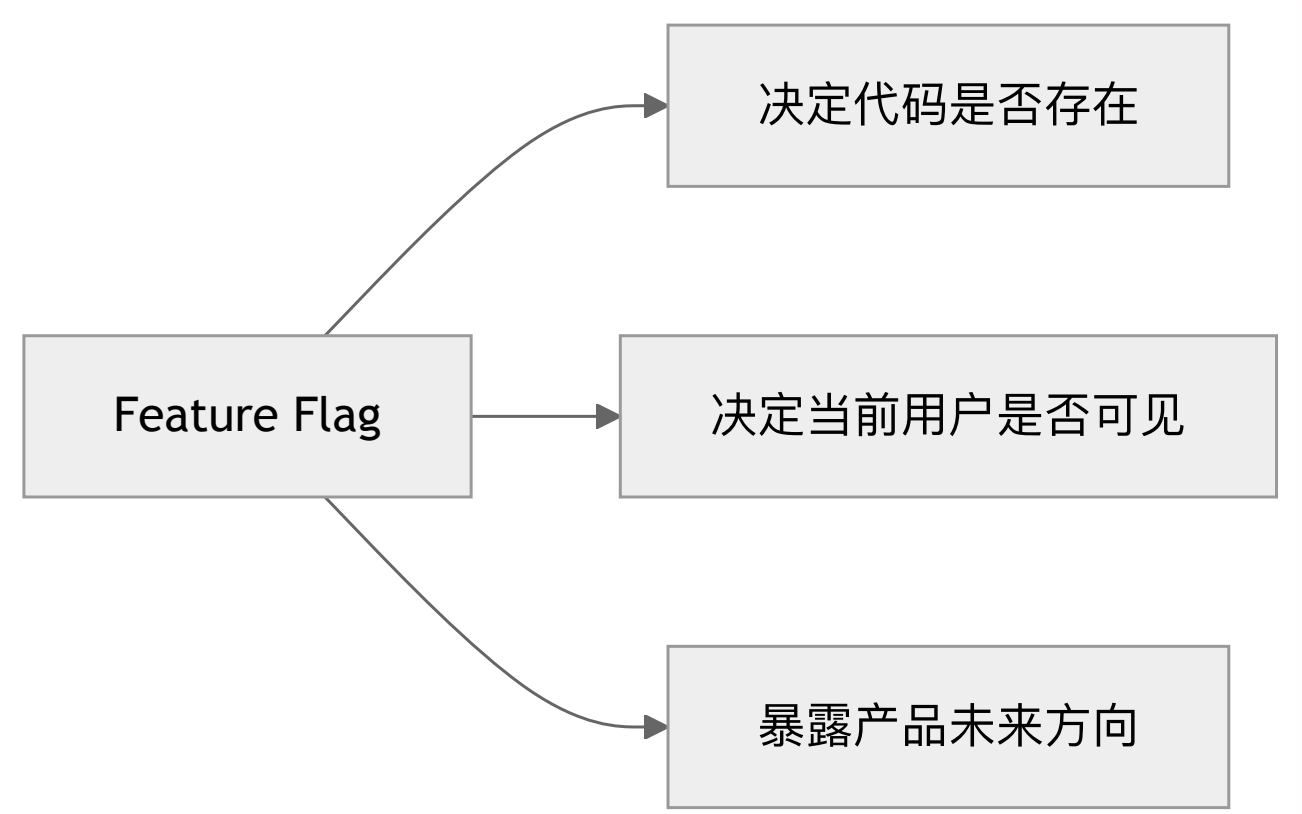
59

Feature Flag 附录

附录D：89 个 Feature Flag

这一附录基于本地源码重新统计，确认当前可识别的唯一 `feature('...')` 标记共 89 个。

D.1 它们为什么重要



D.2 按主题分组

主题	代表 flag
主动与后台	KAIROS PROACTIVE AGENT_TRIGGERS MONITOR_TOOL
上下文与压缩	CONTEXT_COLLAPSE HISTORY_SNIP REACTIVE_COMPACT CACHED_MICROCOMPACT
协作与远程	COORDINATOR_MODE UDS_INBOX BRIDGE_MODE CCR_REMOTE_SETUP
多模态与界面	VOICE_MODE BUDDY TERMINAL_PANEL MESSAGE_ACTIONS
扩展与生态	MCP_SKILLS MCP_RICH_OUTPUT WORKFLOW_SCRIPTS WEB_BROWSER_TOOL

D.3 全量清单（按字母序）

```
ABLATION_BASELINE
AGENT_MEMORY_SNAPSHOT
AGENT_TRIGGERS
AGENT_TRIGGERS_REMOTE
ALLOW_TEST_VERSIONS
ANTI_DISTILLATION_CC
AUTO_THEME
AWAY_SUMMARY
BASH_CLASSIFIER
BG_SESSIONS
BREAK_CACHE_COMMAND
BRIDGE_MODE
BUDDY
BUILDING_CLAUDE_APPS
```

```
BUILTIN_EXPLORE_PLAN_AGENTS
BYOC_ENVIRONMENT_RUNNER
CACHED_MICROCOMPACT
CCR_AUTO_CONNECT
CCR_MIRROR
CCR_REMOTE_SETUP
CHICAGO_MCP
COMMIT_ATTRIBUTION
COMPACTION_REMINDERS
CONNECTOR_TEXT
CONTEXT_COLLAPSE
COORDINATOR_MODE
COWORKER_TYPE_TELEMETRY
DAEMON
DIRECT_CONNECT
DOWNLOAD_USER_SETTINGS
DUMP_SYSTEM_PROMPT
ENHANCED_TELEMETRY_BETA
EXPERIMENTAL_SKILL_SEARCH
EXTRACT_MEMORIES
FILE_PERSISTENCE
FORK_SUBAGENT
HARD_FAIL
HISTORY_PICKER
HISTORY_SNIP
HOOK_PROMPTS
IS_LIBC_GLIBC
IS_LIBC_MUSL
KAIROS
KAIROS_BRIEF
KAIROS_CHANNELS
KAIROS_DREAM
KAIROS_GITHUB_WEBHOOKS
KAIROS_PUSH_NOTIFICATION
LODESTONE
MCP_RICH_OUTPUT
MCP_SKILLS
MEMORY_SHAPE_TELEMETRY
MESSAGE_ACTIONS
MONITOR_TOOL
NATIVE_CLIENT_ATTESTATION
NATIVE_CLIPBOARD_IMAGE
NEW_INIT
OVERFLOW_TEST_TOOL
PERFETTO_TRACING
POWERSHELL_AUTO_MODE
PROACTIVE
PROMPT_CACHE_BREAK_DETECTION
QUICK_SEARCH
REACTIVE_COMPACT
REVIEW_ARTIFACT
RUN_SKILL_GENERATOR
SELF_HOSTED_RUNNER
SHOT_STATS
SKILL_IMPROVEMENT
SLOW_OPERATION_LOGGING
SSH_REMOTE
STREAMLINED_OUTPUT
TEAMMEM
TEMPLATES
TERMINAL_PANEL
TOKEN_BUDGET
TORCH
TRANSCRIPT_CLASSIFIER
TREE_SITTER_BASH
TREE_SITTER_BASH_SHADOW
UDS_INBOX
ULTRAPLAN
ULTRATHINK
UNATTENDED_RETRY
UPLOAD_USER_SETTINGS
VERIFICATION_AGENT
VOICE_MODE
WEB_BROWSER_TOOL
WORKFLOW_SCRIPTS
```

D.4 阅读提醒

- Flag 存在，不代表功能默认开启。
- 同一个能力可能同时受编译期和运行期 gate 控制。
- 想判断某 flag 是否重要，要看它是否进入主链、是否有配套治理、是否跨多目录出现。

附录D结论

89 个 flag 组成的，不只是开关表，而是一张产品路线图。越是跨越工具、UI、主循环、设置与 analytics 多处出现的 flag，越值得重点关注。

60

差异矩阵 附录

附录E：还原层与补全层差异矩阵

这一附录把 `claude-code-sourcemap` 与 `OpenClaudeCode` 的关系做成一张表，方便你在阅读时随时回来看“这个结论到底站在什么证据上”。

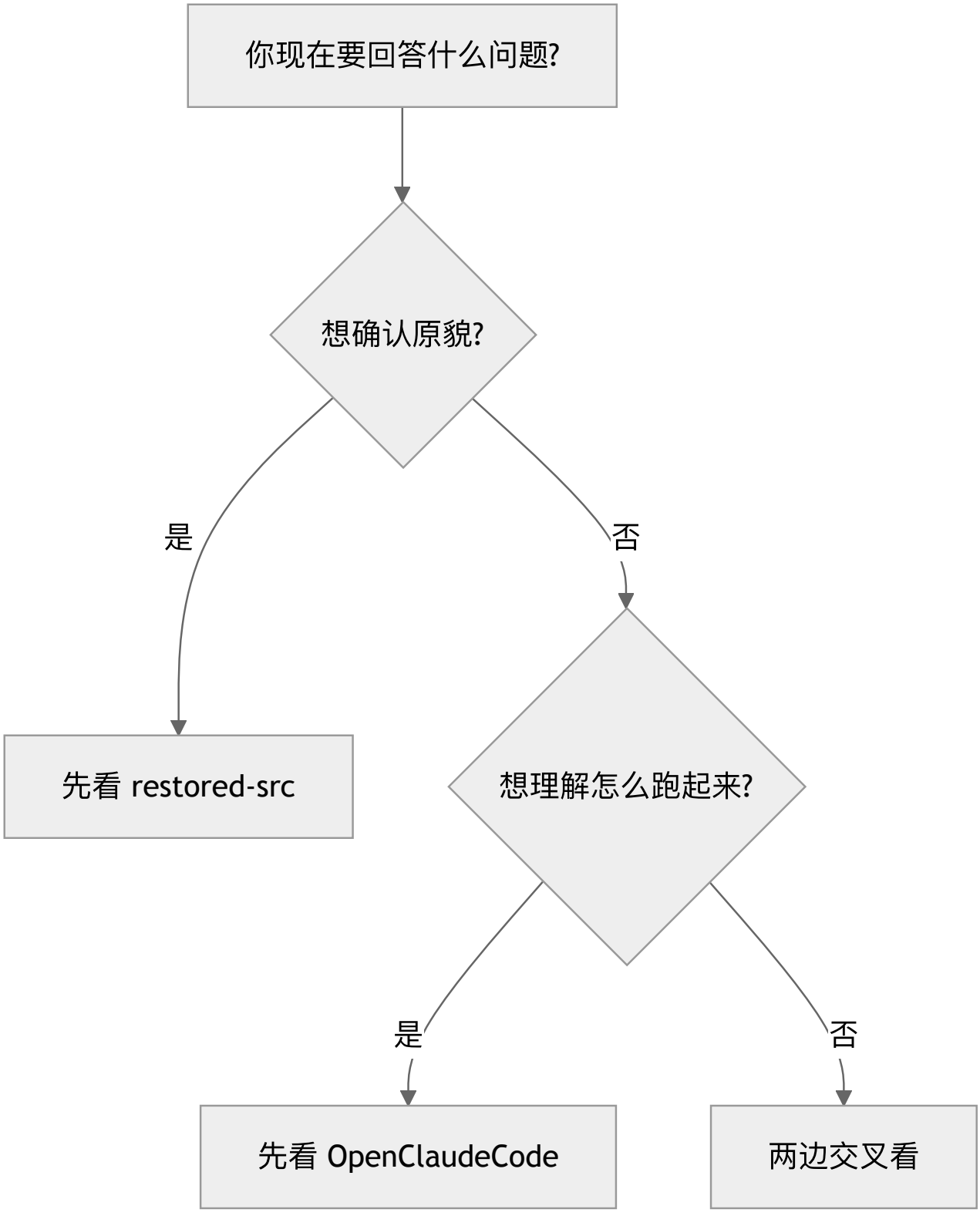
E.1 快速对照

维度	`claude-code-sourcemap`	`OpenClaudeCode`
主要目标	尽量还原已发布包的源码形态	尽量补全到可运行、可研究
`src` 文件数	1884	1989
辅助目录	以还原层为主	额外含 `shims/` 与 `vendor/`
最适合做什么	证据基底、调用链考证	运行验证、结构补全、目录导航

E.2 模块级差异矩阵

模块	还原层可信度	OpenClaudeCode 价值	阅读建议
`query.ts` / `QueryEngine.ts`	高	高	可作为核心主证据
`Tool.ts` / `tools.ts`	高	高	适合双线交叉看
`commands.ts` / `commands/`	高	高	适合做功能全景导航
`bridge/`	中高	高	OpenClaudeCode 更利于研究整体结构
`memdir/`	高	高	两边互证效果好
`shims/`	无	中	只看接口与补全策略，别当原作
`vendor/`	无	中	用于理解托底能力，不宜直接推官方设计

E.3 什么时候优先看哪一套



E.4 本书的默认原则

1. 核心执行链优先引用还原层。
2. 涉及运行时补全、shim、vendor、兼容逻辑时优先说明 OpenClaudeCode 角色。
3. 只在 OpenClaudeCode 出现、无法与还原层互证的结论，自动降一档可信度。

E.5 最终结论

`claude-code-sourcemap` 和 `OpenClaudeCode` 的关系，不是“谁真谁假”，而是“谁更像证据基底，谁更像运行补全”。两者一起读，才最接近严肃源码研究应有的姿势。

附录E结论

还原层帮你守住“别过度想象”，补全层帮你守住“别只停留在碎片理解”。真正可靠的结论，几乎都来自两边的交叉验证。