Claude Code
源码架构深度解析 V2.1

从 4756 个文件里读懂 Agent 系统工程

Xiao Tan

X (Twitter): x.com/tvytlx
公众号: Xiao Tan AI
小红书: tvytlx
抖音: tvytlx
邮箱: tvytlx@gmail.com

April 4, 2026

Contents

1 全局视角：CLI 工具 vs Agent Operating System

1.1 你在看什么

这份报告的素材，来自 Claude Code 的 npm 包。包里有一个 `cli.js.map` 文件，里面的 `sourcesContent` 字段存着完整的 TypeScript 源码。把它提取出来，你会得到将近 4756 个文件。

这个数字本身就说明了很多。市面上大部分开源 coding agent，你打开 `src/` 目录，会看到一个 `main` 文件、一个 `prompt` 文件、几个 `tool` 文件、一个 `utils`。Claude Code 的 `src/` 顶层有超过 50 个模块目录。

1.2 代码库规模

先看几个数字，有个直观感受：

模块	文件数
utils/	564
components/	389
commands/	207
tools/	184
services/	130
hooks/	104
ink/ (TUI 框架)	96
bridge/ (远程/IDE 集成)	31
skills/	20
tasks/	12

关键文件的行数也很有说服力：

文件	行数
main.tsx	4,683
toolExecution.ts	1,745
query.ts (主循环)	1,729
AgentTool.tsx	1,397
QueryEngine.ts	1,295
runAgent.ts	973
prompts.ts	914
Tool.ts (工具基类)	792

1.3 入口层暴露的设计意图

Claude Code 有多个入口：`cli.tsx`、`init.ts`、`mcp.ts`、`sdk/`。同一个 agent 运行时，能服务 CLI 终端、MCP 协议、SDK 调用、IDE 插件。这就是平台化设计。

`cli.tsx` 本身值得看一眼。它不是简单地加载 `main.tsx` 然后跑。它有一套 fast-path 分发逻辑：

- `--version`：零模块加载，直接打印版本号退出
- `remote-control`：走 bridge 模式，不进主 REPL
- `daemon`：走长驻进程模式
- `ps / logs / attach / kill`：走后台会话管理
- `--worktree --tmux`：直接 exec 进 tmux，不加载完整 CLI

只有没命中任何 fast-path 的时候，才会加载 `main.tsx`。而且所有 import 都是 dynamic import，按需加载。

为什么这很重要

这不是过度设计。当你的 CLI 工具每天被上百万次调用的时候，`--version` 的响应速度直接影响用户体验和 CI 流水线性能。把 “不需要加载整个运行时” 的场景全部用 fast-path 拦截，是产品化思维的体现。

1.4 命令系统不是装饰品

`src/commands/` 下有 101 个命令。从 `/mcp`、`/memory`、`/permissions`、`/hooks`、`/plugin`、`/skills`、`/tasks`、`/plan`、`/review`、`/agents`，到 `/compact`、`/doctor`、`/bridge`、`/teleport`。

这些命令不只是快捷操作。命令系统统一加载 plugin commands、skill commands、bundled skills、动态 skills。所以命令系统本身就是用户跟系统运行时交互的控制面，也是生态的入口。

2 引擎：主循环与 Prompt 编排

这一节是整份报告的地基。Claude Code 的一切能力，最终都要通过主循环跑出来，通过 prompt 告诉模型该做什么。

2.1 一个请求怎么跑起来的

从 `cli.tsx` 到最终模型响应，链路大概是这样：

1. `cli.tsx` 分发到 `main.tsx`

2. `main.tsx` 初始化状态、注册工具、构造 `ToolUseContext`
3. 用户输入进入 `query()` 函数
4. `query()` 进入 `queryLoop()`，这是一个 `while(true)` 主循环
5. 每次迭代：压缩上下文 → 组装 system prompt → 调用模型 API → 处理工具调用 → 继续或退出

2.2 query.ts：1729 行的状态机

`query.ts` 是整个系统的心脏。它是一个 async generator，内部是 `while(true)` 循环，通过 `state` 对象在迭代之间传递状态。

每次循环迭代做的事情，按顺序来：

1. **上下文预处理**：snip compact → micro compact → context collapse → auto compact。四道压缩机制依次执行，目的是在有限的上下文窗口里装最有用的信息。
2. **Token 预算检查**：如果 auto compact 关了，检查是否快撞到硬限制。
3. **调用模型 API**：把消息、system prompt、工具定义发给模型。
4. **流式处理响应**：模型的输出是流式的，边收边处理。如果有 tool_use block，可以立刻开始执行工具（streaming tool execution）。
5. **错误恢复**：prompt 太长？试 context collapse drain，再试 reactive compact。输出 token 超限？注入恢复消息让模型继续。模型降级？切 fallback model。
6. **Stop hooks**：模型停止输出后，运行 stop hooks，决定要不要让模型继续。
7. **Token budget**：如果有 token 预算，检查是否该继续。
8. **工具执行**：批量执行所有工具调用，通过 `runTools()` 或 `StreamingToolExecutor`。
9. **附件注入**：工具执行完后，注入 memory attachments、skill discovery、queued commands。
10. **下一轮**：把结果组装成新的消息列表，回到循环开头。

为什么 query.ts 用状态机取代了递归

早期版本的 query 是递归调用自身。但递归在长会话里会爆栈。现在改成了 `while(true)` + `state` 对象的设计，每次 continue 就是一个 state transition。代码里有 9 个不同的 continue 点，每个对应一种“为什么要再跑一轮”的原因：下一轮工具调用（next_turn）、reactive compact 重试、max output tokens 恢复、stop hook 阻断、token budget 继续，等等。

2.3 Streaming Tool Execution：边收边跑

传统做法是等模型完整输出，收齐所有 tool_use block，再一起执行。Claude Code 做了一个优化：`StreamingToolExecutor`。模型还在输出的时候，已经完成的 tool_use block 就开始执行了。

这对用户体验的影响很大。一次 turn 里如果有 5 个工具调用，传统做法要等模型输出完（可能 5-30 秒），然后串行或并行执行工具。streaming 模式下，第一个工具在模型还在生成第二个 tool_use 的时候就已经跑完了。

2.4 Prompt 组装：一台精密的拼装机器

`prompts.ts` 里的 `getSystemPrompt()` 返回的是一个字符串数组，每个元素对应一个 section。

整个 prompt 分成两大块：

静态部分（可缓存）：

- 身份定位（`getSimpleIntroSection`）
- 系统运行规范（`getSimpleSystemSection`）
- 做任务的行为规范（`getSimpleDoingTasksSection`）
- 风险动作规范（`getActionsSection`）
- 工具使用语法（`getUsingYourToolsSection`）
- 语气风格（`getSimpleToneAndStyleSection`）
- 输出效率（`getOutputEfficiencySection`）

动态部分（按会话状态注入）：

- Session guidance（当前启用了哪些工具，影响行为指引）
- Memory（CLAUDE.md 内容）
- 环境信息（OS、shell、cwd、模型名称）
- 语言偏好、输出风格
- MCP server instructions
- Function result clearing 说明
- Token budget 说明

中间用一个 `SYSTEM_PROMPT_DYNAMIC_BOUNDARY` 标记隔开。

2.5 Prompt Cache 经济学

`SYSTEM_PROMPT_DYNAMIC_BOUNDARY` 这个标记的存在，说明 Anthropic 在管理 system prompt 的时候已经在考虑成本了。

原理是这样的：API 层可以对 system prompt 做前缀缓存。如果两次请求的 system prompt 前缀完全一样（字节级一致），第二次就可以跳过对前缀部分的处理。所以把不变的内容放前面，会变的放后面，缓存命中率就上去了。

源码注释里写得很直白：不要随意修改 boundary 之前的内容，否则会破坏缓存。

Prompt Section Registry

动态部分的 section 不是每次都重新计算的。`systemPromptSections.ts` 里有一个 section registry，用 `systemPromptSection()` 创建的 section 会被缓存，直到 `/clear` 或 `/compact`。只有用 `DANGEROUS_uncachedSystemPromptSection()` 创建的才会每次重算。MCP instructions 就是用的 DANGEROUS 版本，因为 MCP server 可能在两个 turn 之间连接或断开。

2.6 行为规范：怎么让 AI 工程师不乱来

`getSimpleDoingTasksSection()` 这个函数可能是整个 prompt 里最有价值的部分。它做的事情就是告诉模型：什么该做，什么不该做。

看看它具体写了什么：

- 不要加用户没要求的功能
- 不要过度抽象，三行重复代码好过一个不成熟的抽象
- 不要给你没改的代码加注释和文档字符串
- 不要做不必要的错误处理和兜底逻辑
- 不要设计面向未来的抽象
- 先读代码再改代码
- 不要轻易建新文件
- 不要给时间估计
- 方法失败了先诊断，不要盲目重试，也不要一次失败就放弃
- 结果要如实汇报，没跑过的不要说跑过了

用过其他 coding agent 的人应该都遇到过这些问题：你让它改个 bug，它顺手重构了半个文件；你让它加一个功能，它加了三层抽象和五个你没要的错误处理。

这些问题的根源在于模型的行为没有被约束。Claude Code 的做法是把行为规范写成制度，

不依赖模型临场发挥。

3 工具系统：42 个工具和一条治理流水线

3.1 Tool 接口设计

`Tool.ts` 定义了一个工具接口。这不是一个简单的“函数名 + 输入 + 输出”的结构。一个完整的 Tool 有这些关键方法：

- `call()`：执行工具
- `inputSchema`：Zod schema，做输入校验
- `validateInput()`：更精细的输入校验
- `checkPermissions()`：工具级别的权限检查
- `preparePermissionMatcher()`：为 Hook 模式匹配做预处理
- `isReadOnly()`：是否只读
- `isDestructive()`：是否有破坏性
- `isConcurrencySafe()`：是否可以并发执行
- `prompt()`：动态生成这个工具的 prompt 描述
- `backfillObservableInput()`：在 SDK stream 和 transcript 里补充可观测字段
- `toAutoClassifierInput()`：给安全分类器的紧凑表示

还有 6 个以上的 render 方法，分别处理工具调用的展示、进度、结果、错误、拒绝、分组等 UI 场景。

`buildTool()` 是工厂函数，提供 fail-closed 的默认值。比如 `isConcurrencySafe` 默认 false（假设不安全），`isReadOnly` 默认 false（假设会写），`checkPermissions` 默认 allow（交给通用权限系统处理）。

fail-closed 默认值

这个选择很有意思。新写一个工具的时候，如果忘了声明并发安全性，系统会默认它不安全，串行执行。忘了声明只读，系统会默认它会写，走更严格的权限检查。这种“忘了就严格”的设计，避免了一个很常见的问题：某个工具忘了配置，结果在没有权限检查的情况下执行了危险操作。

3.2 42 个工具的分类

源码里一共有 42 个工具目录，36 个有独立的 `prompt.ts`。按功能大致可以分成这几类：

文件操作：FileRead、FileEdit、FileWrite、GlobTool、GrepTool、NotebookEdit

Shell 执行：BashTool、PowerShellTool

Agent 调度：AgentTool、TaskCreate/Get/List/Stop/Update/Output

MCP 集成：MCPTool、ListMcpResources、ReadMcpResource、McpAuth

Web 能力：WebSearch、WebFetch

用户交互：AskUserQuestion、SendMessage

模式切换：EnterPlanMode、ExitPlanMode、EnterWorktree、ExitWorktree

其他：SkillTool、SleepTool、TodoWrite、Config、ToolSearch、Brief、ScheduleCron、RemoteTrigger

每个工具都有自己的 `prompt.ts`，定义了这个工具在 system prompt 里的描述。这些描述会通过 `prompt()` 方法动态生成，根据当前可用的工具集和权限模式调整内容。

3.3 工具执行 Pipeline

当模型决定调用一个工具的时候，`toolExecution.ts` 里有一条完整的执行流水线。不是“模型说调就调”。

1. **找工具**：通过名字或别名（alias）找到对应的 Tool 对象
2. **解析 MCP 元数据**：如果是 MCP 工具，提取 server 信息
3. **Zod schema 校验**：用输入 schema 做第一道校验，挡住乱传参数
4. **validateInput()**：工具自己的细粒度校验
5. **Speculative classifier**：如果是 BashTool，启动一个预测性分类器，在权限决策之前就开始分析命令的风险等级
6. **PreToolUse hooks**：运行所有注册的 pre-hook
7. **解析 Hook 权限结果**：Hook 可能返回 allow、ask、deny，也可能修改输入
8. **走权限决策**：综合 Hook 结果、规则配置、用户交互，做出最终允许/拒绝决策
9. **修正输入**：如果权限决策或 Hook 修改了输入，用修改后的版本
10. **执行 tool.call()**：真正跑工具
11. **记录 analytics / tracing / OTel**：遥测和可观测性
12. **PostToolUse hooks**：成功后的 post-hook
13. **处理结果**：结构化输出、tool_result block 构建
14. **PostToolUseFailure hooks**：如果失败了，跑失败 hook

Speculative Classifier

BashTool 有一个独特的设计：在正式权限检查之前，系统会启动一个 speculative classifier 来预判命令的风险。这个分类器可以在 Hook 执行的同时并行运行，不阻塞主流程。等到需要做权限决策的时候，分类结果可能已经出来了。这种“提前开始异步计算”的做法，减少了用户等待权限弹窗的时间。

4 多 Agent 体系：分工和调度

4.1 为什么需要多个 Agent

Claude Code 至少有 6 个内建 Agent：

- **General Purpose Agent**：通用任务执行
- **Explore Agent**：纯只读的代码探索
- **Plan Agent**：纯规划，不执行
- **Verification Agent**：对抗性验证
- **Claude Code Guide Agent**：使用指导
- **Statusline Setup Agent**：状态栏配置

这个设计选择的出发点很朴素：让一个 Agent 同时研究、规划、实现、验证，每件事都做不扎实。

4.2 Explore Agent：裁剪出来的只读专家

Explore Agent 的 prompt 非常严格。它被明确禁止：

- 创建新文件（任何形式）
- 修改已有文件
- 删除文件
- 用重定向写文件
- 运行任何改变系统状态的命令

它能用的工具只有 Glob、Grep、FileRead，Bash 也只允许 `ls`、`git status`、`git log` 这些读操作。

为什么要这么极端？因为探索阶段如果不小心改了东西，后面实现阶段就会出问题。把权限彻底隔离，是一种朴素但有效的安全设计。

它还有一个性能优化：外部用户默认用 Haiku 模型（更快更便宜），内部用户继承主模型。探索不需要最强的推理能力，速度更重要。

4.3 Verification Agent：整个系统里最狠的 prompt

Verification Agent 的 prompt 写了 130 行，可能是整个源码里最精心设计的一段文本。

它的核心方向就一件事：“想办法搞坏它”（try to break it）。

prompt 开头就点出两种常见的验证失败模式：

1. **Verification avoidance**：只看代码，不跑检查，写个 PASS 就走了
2. **被前 80% 迷惑**：UI 看着不错，测试也过了，就忽略剩下 20% 的问题

然后它强制要求一系列验证动作。根据变更类型有不同策略：前端改动要启动 dev server 然后用浏览器自动化去点；后端改动要 curl 实测；CLI 改动要看 stdout/stderr/exit code；数据库迁移要测 up/down 还要测已有数据。

更狠的是，它列出了验证者常见的逃避借口，然后要求“识别你自己的合理化倾向”：

- “代码看起来是对的” → 看是不是验证。跑一下。
- “实现者的测试已经通过了” → 实现者也是 LLM。独立验证。
- “大概没问题” → 大概不是验证。跑一下。
- “这个太费时间了” → 不是你说了算的。

每个检查项必须包含实际执行的命令和观察到的输出。最后给出 VERDICT: PASS、FAIL 或 PARTIAL。

实现者和验证者分离

这种设计在传统软件工程里是常识：写代码的人不应该是验收代码的人。但在 AI Agent 系统里，大部分产品还没做到这一步。Claude Code 把它做成了一个独立角色，有自己的 system prompt、工具集（不能 edit/write）和评判标准。写代码的 Agent 可能会倾向于觉得自己写得没问题，但验证 Agent 没有这个偏见，它的工作就是找问题。

4.4 AgentTool.tsx：调度总控

`AgentTool.tsx` 有 1397 行，是整个多 Agent 系统的调度中心。它要处理的分支非常多：

- 判断是 fork、built-in agent、multi-agent teammate 还是 remote
- 解析输入参数：description、prompt、subagent_type、model、run_in_background、isolation、cwd
- 根据权限规则过滤可用的 agent

- 检查 MCP 依赖
- 构造 system prompt 和 prompt messages
- 处理 worktree 隔离
- 注册前台/后台任务
- 调用 `runAgent()`

4.5 Fork path 的 cache 优化

fork 和 normal path 的区分是一个很精妙的设计。

当你 fork 一个子任务时，它会继承主线程的 system prompt、完整对话上下文、工具集，尽量保持字节级一致。为什么？为了让 API 请求的前缀一样，从而复用主线程的 prompt cache。

源码注释里写得很直白：fork 不应该换模型，因为换模型会改变 system prompt 里的模型描述字段，破坏 cache 前缀匹配。

普通人做子 Agent 调度，想的是“子任务能跑起来就行”。Claude Code 想的是“子任务能跑起来，而且尽量复用主线程的缓存”。

4.6 runAgent.ts：子 Agent 的完整运行时

`runAgent.ts` 有 973 行，负责子 Agent 的完整生命周期。它做的事情包括：

- 初始化 agent 专属的 MCP servers（可以从 frontmatter 定义）
- 克隆 file state cache
- 获取 user/system context
- 对只读 agent 做内容瘦身
- 构造 agent 专属的权限模式
- 组装工具池
- 注册 frontmatter hooks
- 预加载 skills
- 执行 SubagentStart hooks
- 调用 `query()` 进入主循环
- 记录 transcript
- 清理 MCP 连接、hooks、perfetto tracing、todos、shell tasks 等

这里有一个细节：agent 可以自带 MCP servers。通过 frontmatter 配置，一个 agent 可以连

接自己专属的外部工具。这让插件 agent 有了非常强的扩展能力。

4.7 任务系统

`tasks/` 目录下有 5 种任务类型：

- `DreamTask`：自主后台任务
- `LocalAgentTask`：本地 agent 任务（前台/后台/异步）
- `RemoteAgentTask`：远程 agent 任务
- `InProcessTeammateTask`：进程内 teammate
- `LocalShellTask`：shell 任务

后台 agent 有独立的 abort controller，可以在后台持续运行，完成后通过 notification 回到主线程。前台 agent 可以在执行过程中被转成后台。这些都是产品化的生命周期管理。

5 安全层：权限、Hook 和三层防护网

5.1 权限系统概览

`utils/permissions/` 下有 27 个文件，管理一套完整的权限模型：

- **PermissionMode**：default、plan、auto 等模式
- **PermissionRule**：规则定义（allow/deny/ask）
- **PermissionResult**：决策结果
- **bashClassifier**：Bash 命令风险分类
- **yoloClassifier**：auto 模式下的分类器
- **dangerousPatterns**：危险命令模式匹配
- **shellRuleMatching**：Shell 命令的规则匹配
- **pathValidation**：文件路径校验

5.2 Hook 系统：不只是事件钩子

Hook 系统是整个安全层里最有表达力的部分。它支持三个时点：

- **PreToolUse**：工具执行前
- **PostToolUse**：工具执行成功后
- **PostToolUseFailure**：工具执行失败后

Pre-hook 能做的事情远不止“记日志”：

- 返回 `permissionBehavior`（allow / ask / deny）
- 返回 `updatedInput`（修改工具输入）
- 返回 `blockingError`（直接阻断）
- 返回 `preventContinuation`（阻止后续流程）
- 返回 `additionalContexts`（补充上下文信息）

Post-hook 也能修改 MCP 工具的输出、追加消息、注入上下文。

5.3 resolveHookPermissionDecision：安全的关键粘合层

`toolHooks.ts` 里的 `resolveHookPermissionDecision()` 函数定义了一个关键规则：

Hook 说 allow，不一定绕过 settings 里的 deny/ask 规则。

具体逻辑：

- Hook allow + 工具要求用户交互 + Hook 没提供 updatedInput → 仍然走 canUseTool
- Hook allow + settings 里有 deny 规则 → deny 规则生效
- Hook allow + settings 里有 ask 规则 → 仍要弹窗
- Hook deny → 直接生效
- Hook ask → 作为 forceDecision 传给权限弹窗

强大但受控

这个设计非常成熟。Hook 有足够的表达力来做运行时策略调整，但它不能绕开核心安全模型。这意味着即使某个 Hook 写了 bug 或者被恶意利用，它也不能让一个被 settings deny 掉的操作悄悄通过。“强大但受控”是工程成熟度的标志。

5.4 三层防护网

把权限系统、Hook 系统和工具执行 pipeline 放在一起看，Claude Code 有三层防护：

1. **Speculative Classifier**（提前预判）：BashTool 的风险分类器，在 Hook 执行的同时并行运行
2. **Hook Policy Layer**（策略层）：PreToolUse hooks 可以做权限决策、修改输入、阻断流程
3. **Permission Decision**（权限决策）：综合规则配置和用户交互的最终决策

这三层互相配合但互不绕过。Speculative classifier 的结果只是辅助信息，不能绕过 Hook；Hook 的 allow 不能绕过 settings deny。每层都有自己的职责边界。

6 生态：Skill、Plugin、MCP

6.1 Skill：带元数据的 workflow package

`skills/` 目录下有 17 个 bundled skills，包括 `verify`、`commit`、`loop`、`simplify`、`stuck`、`debug` 等。

Skill 的形态是带 frontmatter metadata 的 markdown 文件。它可以声明 allowed-tools、model、effort hints，可以按需注入当前上下文。

系统要求模型在任务匹配到某个 skill 时必须调用 Skill tool 执行，不能只是提到这个 skill 而不执行。

6.2 Plugin：模型行为层面的扩展

`utils/plugins/` 有 42 个文件，涵盖从加载到验证到市场管理的完整链条。Plugin 能提供的能力包括：

- markdown commands 和 SKILL.md 目录
- hooks（Pre/PostToolUse）
- output styles
- MCP server 配置
- 模型和 effort hints
- 运行时变量替换（`${CLAUDE_PLUGIN_ROOT}` 等）
- 自动更新、版本管理、blocklist

Plugin 不是普通的 CLI 插件。它能影响模型的行为：改变 prompt、增加工具、修改权限规则。

6.3 MCP：工具桥 + 行为说明注入

MCP 不只提供新工具。从 `prompts.ts` 可以看到，当 MCP server 连接上来的时候，如果 server 提供了 instructions，这些 instructions 会被拼进 system prompt。

也就是说，一个 MCP server 能同时给模型两样东西：

1. 新工具（通过 MCP 协议注册）
2. 怎么用这些工具的说明（通过 instructions 注入 prompt）

这让 MCP 的价值远高于一个简单的工具注册表。模型不只知道“有这个工具”，还知道“什么时候该用、怎么用”。

6.4 生态的关键：模型“感知到”自己的能力

很多平台也有插件系统、也有工具市场，但模型本身不知道这些东西存在。就像你给一个人配了一整套工具箱，但他不知道箱子里有什么。

Claude Code 通过 skills 列表、agent 列表、MCP instructions、session-specific guidance、command integration 这些通道，让模型感知到自己当前有哪些扩展能力。这才是生态真正发挥作用的前提。

7 上下文经济学：Token 就是预算

7.1 四道压缩机制

`query.ts` 里，每次调用模型之前，消息列表会经过四道处理：

1. **Snip Compact**：把历史消息中过长的部分裁剪掉
2. **Micro Compact**：更细粒度的压缩，基于 tool_use_id 做缓存编辑
3. **Context Collapse**：把不活跃的上下文区域折叠成摘要
4. **Auto Compact**：当总 token 数接近阈值时，触发全量压缩

这四道不是互斥的，它们可以叠加执行。优先级是：先做轻量的（snip、micro compact），再做重量的（context collapse、auto compact）。如果轻量压缩把 token 数压到阈值以下了，重量压缩就不需要跑。

7.2 Reactive Compact：API 413 的兜底

如果四道压缩都没能把 token 数压下来，API 返回了 413（prompt too long），还有一个 reactive compact 机制。它会在收到 413 之后立刻触发一次紧急压缩，然后重试。

但这个机制有防循环设计：`hasAttemptedReactiveCompact` 标记确保每个 turn 只尝试一次。

7.3 Token Budget

`query/tokenBudget.ts` 实现了 token 预算系统。当用户指定一个 token 目标（比如“+500k”），系统会追踪每个 turn 的输出 token，在接近目标时注入 nudge message 让模型继续工作。

这对长任务很有用。以前模型可能自己觉得“差不多了”就停了，现在有了明确的预算概念。

7.4 其他上下文优化

- **Skill 按需注入**：匹配到了才注入，启动时不会全部塞进去

- **MCP instructions 按连接状态注入**：没连上的 server 的说明不占空间
- **Memory prefetch**：在模型流式输出的同时，预取可能相关的 memory 内容
- **Skill discovery prefetch**：同上，预取可能相关的 skill
- **Tool result budget**：单个工具结果太大时，持久化到磁盘，只保留摘要在上下文里

8 记忆系统：跨会话的知识持久化

用户在设置里能看到两个开关：**Auto-memory**（记忆的读写）和 **Auto-dream**（记忆的定期整理）。它们背后是两条独立的后台流水线，一条负责写入，一条负责整理。

8.1 记忆该存什么

记忆被约束为 4 种类型：**user**（用户画像）、**feedback**（行为反馈）、**project**（项目上下文）、**reference**（外部系统指针）。每条记忆是一个带 frontmatter 的 `.md` 文件，存在 `~/.claude/projects/<project>/memory/` 下。

比类型更重要的是**什么不该存**：代码模式、架构、文件结构、git 历史。这些信息可以从当前代码库推导出来，存进记忆只会制造冗余和过时风险。用户要求存一份 PR 列表怎么办？prompt 会引导模型追问“里面有什么意外的或反直觉的”，只保留那部分。

这个设计选择背后的思路是：记忆应该只存**不可推导**的知识。能从代码 grep 到的、能从 git log 看到的，都不值得占记忆空间。

8.2 两条流水线：增量写入 + 周期整理

extractMemories（增量写入）在每次模型给出最终回复后触发，作为 forked sub-agent 后台运行。它分析当前对话，把值得持久化的信息写入记忆文件。写操作被限制在记忆目录内，硬上限 5 个 turn。如果主 agent 已经直接写了记忆（用户说了“记住这个”），后台提取自动跳过，互不干扰。

autoDream（周期整理）每 24 小时触发一次（需要累积至少 5 个新会话）。它做的事情更重：回顾多个会话的 transcript，把新信号合并到已有主题文件里，清理过时内容，维护 `MEMORY.md` 索引。

增量写入和全局整理为什么要分开

分开之后，增量写入可以很轻（2-4 turn，只看当前对话），全局整理可以做得更深（读 transcript、交叉验证、去重合并）。合成一个流程的话，要么每次对话后做太多事（慢），要么周期性整理时漏掉当前对话的信息（不完整）。

8.3 召回：用便宜模型做筛选

记忆多了之后，不可能全部塞进上下文。召回流程用了双模型架构：先扫描所有记忆文件的 frontmatter，生成一份 manifest 清单（文件名 + description + type + 时间戳），然后用一个更便宜的 Sonnet 模型挑最多 5 条相关记忆。主模型只看到被选中的那几条。

选择器有一个细节值得注意：如果最近在用某个工具，会跳过这个工具的使用文档类记忆（已经在用了，不需要教程），但保留关于这个工具的 warnings 和 known issues（正在用的时候才最需要知道坑在哪）。

8.4 新鲜度：把时间问题变成数据问题

模型在推理“这个日期距离现在多久”这件事上表现不好。但如果你直接说“47 days ago”，它能正确判断信息可能过时了。所以系统把时间戳转成自然语言格式，注入每条被召回的记忆旁边。

配合一套验证协议：记忆里提到某个函数存在，推荐给用户之前先 grep 确认。记忆里说某个文件在某个路径，先检查文件还在不在。大部分系统处理过时信息的方式是在 prompt 里加一句“注意可能过时”。但模型很擅长忽视这种泛泛的提醒。Claude Code 的做法是在数据层面给出具体时间，在 protocol 层面给出可执行的验证步骤，而不只是一个抽象的警告。

8.5 Session Memory 复用为 Compact 数据源

Session Memory 在后台持续维护一份会话总结。这份总结有一个很聪明的双重用途：它既是会话内的知识提取，也直接作为 compact 的数据源。

传统的 auto compact 需要额外调一次 API 让模型总结历史对话。Session Memory 换了思路：总结已经在后台增量维护好了，compact 触发的时候直接用，省掉了总结的 API 调用。

8.6 安全：记忆路径不可信

记忆目录的路径配置有一个安全细节：`projectSettings`（仓库里提交的 `.claude/settings.json`）被明确排除在配置来源之外。原因是一个恶意仓库可以设置 `autoMemoryDirectory` 指向敏感目录，利用文件系统的 write 白名单获得静默写入权限。只有用户本地设置和策略设置这些受信任来源可以配置记忆路径。

9 产品化：从 prototype 到 product

9.1 生命周期管理

`runAgent.ts` 里有很多不起眼但说明问题的函数调用：

- `recordSidechainTranscript()`：记录子 agent 对话

- `writeAgentMetadata()`：写元数据
- `registerPerfettoAgent()`：性能追踪
- `cleanupAgentTracking()`：清理跟踪状态
- `killShellTasksForAgent()`：清理 shell 进程
- 清理 session hooks、克隆的文件状态、todos entry

大部分 Agent 系统在第一天跑得挺好。问题出在第二天。任务中断了怎么续？脏状态怎么清？子 Agent 的 shell 进程没杀掉怎么办？MCP 连接泄漏了怎么办？

Claude Code 对这些都有明确的处理。

9.2 Bridge 系统

`bridge/` 有 31 个文件，实现了远程控制和 IDE 集成。让 Claude Code 可以在不同环境之间桥接：本地 CLI 控制远程容器，IDE 插件连接 Claude Code 运行时，等等。

9.3 State 管理

`state/AppState.tsx` 和 `store.ts` 管理全局应用状态。权限模式、MCP 连接、工具配置、effort 设置，所有运行时状态都在这里统一管理。

9.4 UI 层

`components/`（389 文件）和 `ink/`（96 文件）实现了终端 UI。Claude Code 用 React + Ink 构建了一个完整的 TUI 应用，有权限弹窗、进度条、diff 显示、agent 状态面板。

9.5 Telemetry

`services/analytics/`、`utils/telemetry/` 覆盖了 Datadog、Perfetto tracing、OTel 事件。cost tracker 追踪 API 调用成本，rate limit 管理做进了运行时。

10 从源码里提炼出的设计原则

把前面所有模块看完之后，可以归纳出这些设计原则。每一条都有对应的源码实现做支撑。

10.1 原则 1：不信任模型的自觉性

好行为要写成制度。你希望模型先读代码再改代码，就写进 prompt。你希望模型不要乱加功能，就写进 prompt。你希望模型遇到风险操作停下来，就在 runtime 层加权限检查。

不要指望一个 LLM 每次都 “想到” 该怎么做。制度化的行为比临场发挥稳定得多。

对应源码：`getSimpleDoingTasksSection()`、`getActionsSection()`

10.2 原则 2：把角色拆开

至少把“做事的人”和“验收的人”分开。哪怕用同一个模型，职责拆开也会有明显改善。因为同一个 Agent 既实现又验证，天然倾向于觉得自己做得没问题。

对应源码：Verification Agent、Explore Agent、Plan Agent

10.3 原则 3：工具调用要有治理

模型说要调工具，中间还要过输入校验、权限检查、风险预判。执行完了还有后处理和失败处理。这层治理决定了系统在异常情况下的表现。

对应源码：`toolExecution.ts` 的 14 步 pipeline

10.4 原则 4：上下文是预算

每个 token 都有成本，每条信息都占空间。能缓存的要缓存，能按需加载的不要一开始就塞进去，能压缩的要压缩。

对应源码：`SYSTEM_PROMPT_DYNAMIC_BOUNDARY`、fork cache optimization、四道压缩机制

10.5 原则 5：安全层要互不绕过

三层防护（classifier、hook、permission）可以互相配合，但任何一层不能绕过另一层。Hook 的 allow 不能绕过 settings 的 deny。这样即使某一层出了问题，整体安全性不会崩塌。

对应源码：`resolveHookPermissionDecision()`

10.6 原则 6：生态的关键是模型感知

你给系统接了十个插件，但模型不知道什么时候该用哪个，那这十个插件就等于不存在。扩展机制的最后一步，是让模型看到自己的能力清单。

对应源码：MCP instructions injection、skill discovery、session-specific guidance

10.7 原则 7：产品化在于处理第二天

第一天跑起来不难。难的是任务中断怎么续、脏状态怎么清、进程泄漏怎么办、session 怎么恢复。这些问题不解决，产品就只能是 Demo。

对应源码：`runAgent.ts` 的 cleanup chain、transcript recording、task lifecycle

11 附录：核心文件索引

11.1 入口与主循环

- src/entrypoints/cli.tsx —CLI 入口，fast-path 分发
- src/main.tsx —主应用，状态初始化，4683 行
- src/query.ts —主循环状态机，1729 行
- src/QueryEngine.ts —查询引擎，1295 行
- src/Tool.ts —工具基类定义，792 行

11.2 Prompt 系统

- src/constants/prompts.ts —System prompt 组装，914 行
- src/constants/systemPromptSections.ts —Section 缓存机制
- src/tools/*/prompt.ts —36 个工具的独立 prompt

11.3 工具执行

- src/services/tools/toolExecution.ts —执行 pipeline，1745 行
- src/services/tools/toolHooks.ts —Hook 系统，650 行
- src/services/tools/toolOrchestration.ts —工具编排
- src/services/tools/StreamingToolExecutor.ts —流式执行器

11.4 Agent 系统

- src/tools/AgentTool/AgentTool.tsx —调度总控，1397 行
- src/tools/AgentTool/runAgent.ts —子 agent 运行时，973 行
- src/tools/AgentTool/forkSubagent.ts —fork 机制
- src/tools/AgentTool/built-in/verificationAgent.ts
- src/tools/AgentTool/built-in/exploreAgent.ts
- src/tools/AgentTool/built-in/planAgent.ts

11.5 权限与安全

- src/utils/permissions/ —27 个文件
- src/utils/permissions/bashClassifier.ts
- src/utils/permissions/dangerousPatterns.ts

11.6 生态扩展

- `src/skills/` —Skill 系统，含 17 个 bundled skills
- `src/utils/plugins/` —Plugin 系统，42 个文件
- `src/services/mcp/` —MCP 集成，23 个文件

11.7 上下文管理

- `src/services/compact/` —压缩系统，11 个文件
- `src/services/compact/sessionMemoryCompact.ts` —Session Memory 驱动的 compact
- `src/query/tokenBudget.ts` —Token 预算

11.8 记忆系统

- `src/memdir/memdir.ts` —记忆系统主入口，prompt 构建
- `src/memdir/memoryTypes.ts` —四种记忆类型定义 + 行为规范
- `src/memdir/memoryScan.ts` —记忆文件扫描与 manifest 生成
- `src/memdir/memoryAge.ts` —新鲜度计算与 staleness 标注
- `src/memdir/findRelevantMemories.ts` —双模型记忆召回
- `src/memdir/paths.ts` —记忆路径解析与安全校验
- `src/services/SessionMemory/sessionMemory.ts` —Session Memory 后台提取
- `src/services/SessionMemory/sessionMemoryUtils.ts` —阈值管理与状态追踪
- `src/services/extractMemories/extractMemories.ts` —后台记忆提取 agent，616 行
- `src/services/autoDream/autoDream.ts` —后台记忆整理（dream）
- `src/services/autoDream/consolidationPrompt.ts` —四阶段整理 prompt
- `src/services/autoDream/consolidationLock.ts` —整理锁与时间戳管理
- `src/services/autoDream/config.ts` —autoDreamEnabled 开关逻辑
- `src/tools/AgentTool/agentMemory.ts` —Agent 专属记忆（三种 scope）
- `src/tools/AgentTool/agentMemorySnapshot.ts` —团队记忆快照同步
- `src/utils/memoryFileDetection.ts` —记忆路径安全检测，290 行