

Claude Code 源码深度研究报告

Xiao Tan
X: x.com/tvytlx
公众号: Xiao Tan AI

March 31, 2026

Contents

1	Claude Code 源码深度研究报告 (增强完整版)	4
1.1	目录	4
2	1. 研究范围与结论总览	4
2.1	1.1 这次到底研究了什么	4
2.2	1.2 关键确认事实	4
2.3	1.3 先给最重要的总判断	5
3	2. 源码结构全景: 它为什么更像 Agent Operating System	5
3.1	2.1 顶层结构暴露出的系统复杂度	5
3.2	2.2 入口层说明它是平台, 而不是单一界面	6
3.3	2.3 命令系统是产品的操作面板	6
3.4	2.4 Tools 层才是模型真正“能做事”的根	7
4	3. 系统提示词总装: prompts.ts 的真实地位	7
4.1	3.1 真正的主入口: src/constants/prompts.ts	7
4.2	3.2 getSystemPrompt() 不是文本, 而是编排器	8
4.2.1	静态前缀 (更适合 cache)	8
4.2.2	动态后缀 (按会话条件注入)	8
4.3	3.3 Prompt cache boundary: 基础设施级思维	8
5	4. Prompt 全量提取与模块级拆解	9
5.1	4.1 身份与基础定位: getSimpleIntroSection()	9
5.2	4.2 基础系统规范: getSimpleSystemSection()	9
5.3	4.3 做任务哲学: getSimpleDoingTasksSection()	10
5.4	4.4 风险动作规范: getActionsSection()	10
5.5	4.5 工具使用规范: getUsingYourToolsSection()	10
5.6	4.6 Session-specific guidance: 运行时可变的局部指令	11
5.7	4.7 Output efficiency: 高价值但常被忽视的 section	11
5.8	4.8 Tone and style: 统一交互感受	11
5.9	4.9 DEFAULT_AGENT_PROMPT: 子 Agent 的基础人格	12

6	5. Agent Prompt 与 built-in agents 深挖	12
6.1	5.1 AgentTool/prompt.ts: Agent 协议说明书	12
6.2	5.2 fork 语义为什么很强	12
6.3	5.3 “How to write the prompt” 一节非常值钱	13
6.4	5.4 built-in agents: 职责分工而不是万能 worker	13
6.5	5.5 Explore Agent: 纯读模式的代码探索专家	13
	6.5.1 绝对只读	14
	6.5.2 核心能力	14
6.6	5.6 Plan Agent: 纯规划, 不做编辑	14
6.7	5.7 Verification Agent: 为什么它非常值钱	14
7	6. Agent 调度链深挖: 从 AgentTool 到 runAgent 再到 query	15
7.1	6.1 总体调用链长什么样	15
7.2	6.2 AgentTool.call(): 真正的调度总控	15
7.3	6.3 fork path vs normal path	16
	7.3.1 fork path	16
	7.3.2 normal path	16
7.4	6.4 为什么 fork 会强调 cache-identical prefix	16
7.5	6.5 background agent 与 foreground agent 是两套生命周期	17
	7.5.1 background path 特点	17
	7.5.2 foreground path 特点	17
7.6	6.6 runAgent(): 真正的子 Agent runtime	17
7.7	6.7 agent-specific MCP servers: 真正的 additive 能力注入	18
7.8	6.8 frontmatter hooks 与 frontmatter skills	18
7.9	6.9 query() 才是最终主循环执行器	18
7.10	6.10 transcript / metadata / cleanup: 产品化 runtime 的证据	18
8	7. Skills / Plugins / Hooks / MCP 生态深挖	19
8.1	7.1 Skill: 不是文档, 而是 workflow package	19
	8.1.1 Skill 的本质	19
8.2	7.2 Plugin: Prompt + Metadata + Runtime Constraints	19
8.3	7.3 Hook: 运行时治理层	20
8.4	7.4 Hook 与权限的耦合方式非常成熟	20
8.5	7.5 MCP: 不只是工具桥, 还是行为说明注入通道	20
9	8. 权限、Hook、工具执行链深挖	21
9.1	8.1 toolExecution.ts: 真正的工具 runtime 主线	21
9.2	8.2 输入校验: 先挡低级错误	21
9.3	8.3 PreToolUse hooks: 最关键的拦截点	21
	9.3.1 updatedInput	21
	9.3.2 permissionBehavior	22
	9.3.3 preventContinuation	22
9.4	8.4 resolveHookPermissionDecision(): 权限语义的关键粘合层	22
9.5	8.5 工具执行后也不是结束	22

10 9. 为什么 Claude Code 这么强：从源码看它真正的护城河	22
10.1 9.1 它不是一个 prompt, 而是一个 operating model	22
10.2 9.2 它把“好行为”制度化了	23
10.3 9.3 它特别懂“上下文是稀缺资源”	23
10.4 9.4 Agent specialization 是很深的设计优势	23
10.5 9.5 它的生态不是“可安装”，而是“模型可感知”	24
11 10. 关键文件索引与后续可继续深挖方向	24
11.1 10.1 核心 Prompt 文件	24
11.1.1 主系统提示词	24
11.1.2 Agent Tool Prompt	24
11.1.3 Skill Tool Prompt	24
11.1.4 其他可继续挖的 tool-specific prompt	24
11.2 10.2 核心 Agent 文件	24
11.2.1 built-in agents	24
11.3 10.3 核心 Skill / Plugin / Hook / MCP 文件	25
11.3.1 Skill	25
11.3.2 Plugin	25
11.3.3 Hook	25
11.3.4 Tool execution	25
11.3.5 MCP	25
11.4 10.4 如果继续做下一轮，还能怎么挖	25
11.5 10.5 最终结论	26

1 Claude Code 源码深度研究报告（增强完整版）

基于 @anthropic-ai/claude-code npm 包的 `cli.js.map` 还原源码后完成的系统性研究。本文是单文件、长篇、研究报告版，重点覆盖：整体架构、系统提示词、Agent 提示词、Skills、Plugins、Hooks、MCP、权限与工具调用机制，以及新增的全量 Prompt 提取框架分析与 Agent 调度链深挖。

1.1 目录

1. 研究范围与结论总览
 2. 源码结构全景：它为什么更像 Agent Operating System
 3. 系统提示词总装：`prompts.ts` 的真实地位
 4. Prompt 全量提取与模块级拆解
 5. Agent Prompt 与 built-in agents 深挖
 6. Agent 调度链深挖：从 AgentTool 到 runAgent 再到 query
 7. Skills / Plugins / Hooks / MCP 生态深挖
 8. 权限、Hook、工具执行链深挖
 9. 为什么 Claude Code 这么强：从源码看它真正的护城河
 10. 关键文件索引与后续可继续深挖方向
-

2 1. 研究范围与结论总览

2.1 1.1 这次到底研究了什么

这次不是只看某一个 prompt 文件，也不是只做“目录级扫一眼”。这次研究的核心，是把 `cli.js.map` 中的 `sourcesContent` 还原成可读源码后，沿着以下主线做系统性拆解：

- Claude Code 的整体源码结构
- 主系统提示词如何动态拼装
- AgentTool / SkillTool 的模型侧协议
- built-in agents 的角色分工
- Agent 调度链路如何跑通
- Plugin / Skill / Hook / MCP 如何接入并影响运行时
- Permission / Tool execution / Hook decision 如何协同
- 它为什么在体验上比“普通 LLM + 工具调用器”强很多

2.2 1.2 关键确认事实

本次已经确认：

1. npm 包里的 `cli.js.map` 包含完整 `sourcesContent`
2. 已从 map 中提取出 4756 个源码文件
3. 主系统提示词核心文件为：

- `src/constants/prompts.ts`
- 4. Agent Tool Prompt 核心文件为:
 - `src/tools/AgentTool/prompt.ts`
- 5. Skill Tool Prompt 核心文件为:
 - `src/tools/SkillTool/prompt.ts`
- 6. Agent 调度核心文件至少包括:
 - `src/tools/AgentTool/AgentTool.tsx`
 - `src/tools/AgentTool/runAgent.ts`
- 7. 工具执行链核心文件至少包括:
 - `src/services/tools/toolExecution.ts`
 - `src/services/tools/toolHooks.ts`

2.3 1.3 先给最重要的总判断

Claude Code 的强，不是来自某个“神秘 system prompt”，而是来自一个完整的软件工程系统：

- Prompt 不是静态文本，而是模块化 runtime assembly
- Tool 不是直接裸调，而是走 permission / hook / analytics / MCP-aware execution pipeline
- Agent 不是一个万能 worker，而是多种 built-in / fork / subagent 的分工系统
- Skill 不是说明文档，而是 prompt-native workflow package
- Plugin 不是外挂，而是 prompt + metadata + runtime constraint 的扩展机制
- MCP 不是单纯工具桥，而是同时能注入工具与行为说明的 integration plane

一句话总结：

Claude Code 的价值，不是一段 prompt，而是一整套把 prompt、tool、permission、agent、skill、plugin、hook、MCP、cache 和产品体验统一起来的 Agent Operating System。

3 2. 源码结构全景：它为什么更像 Agent Operating System

3.1 2.1 顶层结构暴露出的系统复杂度

从提取出来的 `src/` 顶层看，Claude Code 至少有这些重要模块：

- `src/entrypoints/`：入口层
- `src/constants/`：prompt、系统常量、风险提示、输出规范
- `src/tools/`：工具定义与具体实现
- `src/services/`：运行时服务，例如 tools、mcp、analytics

- `src/utils/`: 底层共用能力
- `src/commands/`: slash command 与命令系统
- `src/components/`: TUI / UI 组件
- `src/coordinator/`: 协调器模式
- `src/memdir/`: 记忆 / memory prompt
- `src/plugins/` 与 `src/utils/plugins/`: 插件生态
- `src/hooks/` 与 `src/utils/hooks.js`: hook 系统
- `src/bootstrap/`: 状态初始化
- `src/tasks/`: 本地任务、远程任务、异步 agent 任务

这已经说明它不是简单 CLI 包装器，而是一个完整运行平台。

3.2 2.2 入口层说明它是平台，而不是单一界面

可见入口包括：

- `src/entrypoints/cli.tsx`
- `src/entrypoints/init.ts`
- `src/entrypoints/mcp.ts`
- `src/entrypoints/sdk/`

也就是说它从设计上就考虑了：

- 本地 CLI
- 初始化流程
- MCP 模式
- SDK 消费者

这是一种平台化思维：同一个 agent runtime，可以服务多个入口和多个交互表面。

3.3 2.3 命令系统是产品的操作面板

`src/commands.ts` 暴露出非常多系统级命令，例如：

- `/mcp`
- `/memory`
- `/permissions`
- `/hooks`
- `/plugin`
- `/reload-plugins`
- `/skills`
- `/tasks`
- `/plan`
- `/review`
- `/status`
- `/model`
- `/output-style`
- `/agents`

- `/sandbox-toggle`

这说明命令系统不是“锦上添花”，而是用户与系统运行时交互的重要控制面。
更关键的是，它不仅注册 builtin commands，还统一加载：

- plugin commands
- skill commands
- bundled skills
- 动态 skills
- 可用性过滤后的 commands

所以 command system 本身就是生态入口。

3.4 2.4 Tools 层才是模型真正“能做事”的根

从 prompt 和工具名能确认的重要工具包括：

- FileRead
- FileEdit
- FileWrite
- Bash
- Glob
- Grep
- TodoWrite
- TaskCreate
- AskUserQuestion
- Skill
- Agent
- MCPTool
- Sleep

工具层的本质，是把模型从“回答器”变成“执行体”。Claude Code 的强，很大程度来自这层做得正式、清晰、可治理。

4 3. 系统提示词总装：prompts.ts 的真实地位

4.1 3.1 真正的主入口：src/constants/prompts.ts

这份文件是整个系统最关键的源码之一。不是因为它写了一大段神奇文案，而是因为它承担了：

- 主系统提示词的总装配
- 环境信息注入
- 工具使用规范注入
- 安全与风险动作规范
- Session-specific guidance 注入
- language / output style 注入

- MCP instructions 注入
- memory prompt 注入
- scratchpad 说明注入
- function result clearing 提示注入
- brief / proactive / token budget 等 feature-gated section 注入

Claude Code 的 prompt 不是静态字符串，而是一个 **system prompt assembly architecture**。

4.2 3.2 `getSystemPrompt()` 不是文本，而是编排器

`getSystemPrompt()` 里最核心的结构，是先构造静态部分，再加上动态部分。你可以把它理解成：

4.2.1 静态前缀（更适合 cache）

- `getSimpleIntroSection()`
- `getSimpleSystemSection()`
- `getSimpleDoingTasksSection()`
- `getActionsSection()`
- `getUsingYourToolsSection()`
- `getSimpleToneAndStyleSection()`
- `getOutputEfficiencySection()`

4.2.2 动态后缀（按会话条件注入）

- session guidance
- memory
- ant model override
- env info
- language
- output style
- mcp instructions
- scratchpad
- function result clearing
- summarize tool results
- numeric length anchors
- token budget
- brief

这个设计非常值钱，因为它不是“把能想到的都写进 system prompt”，而是把 prompt 当作可编排运行时资源来管理。

4.3 3.3 Prompt cache boundary: 基础设施级思维

源码中明确存在：

- `SYSTEM_PROMPT_DYNAMIC_BOUNDARY`

并且注释说明：- 边界前尽量可 cache - 边界后是用户 / 会话特定内容 - 不能乱改，否则会破坏 cache 逻辑

这点非常重要。

因为这说明 Claude Code 已经不是“会写 prompt”，而是在做：

Prompt assembly with cache economics

也就是说，它连 system prompt 的 token 成本与缓存命中都做了工程化优化。

5 4. Prompt 全量提取与模块级拆解

这一节是本次新增重点：不是只说 prompt 在哪，而是把主 prompt 的主要 section 和行为价值拆出来。

5.1 4.1 身份与基础定位：getSimpleIntroSection()

这一段做的事包括：

- 定义自己是 interactive agent
- 说明是帮助用户完成软件工程任务
- 指出输出风格会受 Output Style 约束
- 注入 CYBER_RISK_INSTRUCTION
- 明确禁止随意生成或猜 URL

这段意义不在于“介绍自己”，而在于为后续行为定基调：

1. 它不是普通聊天机器人
2. 它是工具驱动的工程协作者
3. 风险防护从第一屏开始就被注入

5.2 4.2 基础系统规范：getSimpleSystemSection()

这段本质上在定义 runtime reality：

- 所有非工具输出都直接给用户看
- 工具运行在 permission mode 下
- 用户拒绝后不能原样重试
- tool result / user message 里可能有 <system-reminder> 等标签
- 外部工具结果可能包含 prompt injection
- 有 hooks
- 上下文会被自动压缩，不是硬性上下文窗口

这段极其重要，因为它把模型从“语言模型幻觉世界”拉回了“受控 runtime 世界”。

5.3 4.3 做任务哲学: `getSimpleDoingTasksSection()`

这部分是 Claude Code 行为稳定性的核心之一。它非常明确地约束模型:

- 不要加用户没要求的功能
- 不要过度抽象
- 不要瞎重构
- 不要乱加 comments / docstrings / type annotations
- 不要做不必要的 error handling / fallback / validation
- 不要设计一堆 future-proof abstraction
- 先读代码再改代码
- 不要轻易创建新文件
- 不要给时间估计
- 方法失败时要先诊断再换策略
- 注意安全漏洞
- 删除确认没用的东西, 不搞 compatibility 垃圾
- 结果要如实汇报, 不能假装测试过

这块本质上不是 prompt 细节, 而是:

Anthropic 对 AI 工程师行为规范的制度化表达

很多 coding agent 不稳定, 不是不会写代码, 而是行为发散。这一段就是为了解决行为漂移。

5.4 4.4 风险动作规范: `getActionsSection()`

这部分定义了什么叫“需要确认”的风险动作:

- destructive operations
- hard-to-reverse operations
- 修改共享状态
- 对外可见动作
- 上传到第三方工具

并且还强调: - 不要用 destructive actions 当捷径 - 发现陌生状态先调查 - merge conflict / lock file 不要粗暴删

这一段值钱的点在于: 它把 blast radius 思维编码进了系统提示词。

5.5 4.5 工具使用规范: `getUsingYourToolsSection()`

这里面有非常清楚的工具策略:

- 读文件优先 FileRead, 不要 cat/head/tail/sed
- 改文件优先 FileEdit, 不要 sed/awk
- 新建文件优先 FileWrite, 不要 echo 重定向
- 搜文件优先 Glob
- 搜内容优先 Grep

- Bash 只保留给真正需要 shell 的场景
- 有任务管理工具时要用 TodoWrite / TaskCreate
- 没有依赖关系的工具调用要并行

这非常关键，因为它不只是说“你有工具”，而是说：

你要以正确的操作语法使用这些工具

Claude Code 的稳，和这套 tool usage grammar 有很大关系。

5.6 4.6 Session-specific guidance: 运行时可变的局部指令

`getSessionSpecificGuidanceSection()` 是个非常关键的动态 section。里面会根据当前工具集和 feature gate 拼出一些当下约束，例如：

- 如果有 AskUserQuestion，被拒绝后可以问用户
- 非交互模式下的行为差异
- 是否启用 AgentTool
- Explore / Plan agents 是否可用
- slash skill 的使用规则
- DiscoverSkills 工具的调用 guidance
- Verification agent 的强制验证合同

这一段说明 Claude Code 的 system prompt 不是“总规则”，而是“总规则 + 当前会话的局部规则”。

5.7 4.7 Output efficiency: 高价值但常被忽视的 section

这一段在 ant 与外部用户上有分支，但核心目标一致：

- 用户看的是自然语言，不是日志
- 先说动作或结论，不要铺垫
- 该更新时更新，但不要废话
- 不要过度解释
- 不要塞无谓表格
- 短句直给

这说明 Claude Code 不只管“能不能完成任务”，还管“完成任务时用户体验像不像正式产品”。

5.8 4.8 Tone and style: 统一交互感受

这部分规定：

- 不要乱用 emoji
- 响应要简洁
- 引用代码位置时用 `file_path:line_number`
- GitHub issue / PR 用 `owner/repo#123`
- tool call 前不要加冒号

这类细则看起来小，但它们会显著塑造产品质感。

5.9 4.9 DEFAULT_AGENT_PROMPT: 子 Agent 的基础人格

在 `prompts.ts` 里还定义了:

- `DEFAULT_AGENT_PROMPT`

核心意思是:

- 你是 Claude Code 的 agent
- 用工具完成任务
- 任务要完整, 不要半成品
- 完成后给简洁报告

这说明主线程与子 agent 在 prompt 结构上是有分层的。

6 5. Agent Prompt 与 built-in agents 深挖

6.1 5.1 AgentTool/prompt.ts: Agent 协议说明书

这份文件非常值钱。它不是普通说明文, 而是 AgentTool 的模型侧协议文档。

它主要说明:

- 如何展示 agent list
- 每个 agent 的描述格式
- 何时 fork 自己
- 何时显式指定 `subagent_type`
- fork 与 fresh agent 的区别
- 什么情况下不要用 AgentTool
- 如何写给子 agent 的 prompt
- foreground / background 的行为差异
- isolation: worktree / remote 的语义

这说明多 agent 不是暗箱, 而是明确写给模型看的使用协议。

6.2 5.2 fork 语义为什么很强

当 fork 开启时, prompt 明确告诉模型:

- omit `subagent_type` 就是 fork 自己
- fork 继承完整 conversation context
- 研究任务很适合 fork
- 实现任务如果会产生大量中间输出, 也适合 fork
- fork 很便宜, 因为共享 prompt cache
- 不要给 fork 单独设 model, 否则 cache 命中会变差
- 不要偷窥 fork 输出文件
- 不要预言 fork 结果

这个设计本质上是在解决一个大问题：

怎么让复杂子任务并行运行，但不污染主上下文

这是多 agent 系统里非常核心、也非常难做对的一件事。

6.3 5.3 “How to write the prompt” 一节非常值钱

Agent prompt 里很明确地教育模型：

- fresh agent 没有上下文，要像对新同事 briefing 一样写 prompt
- 说明目标和原因
- 说明你已经排除了什么
- 提供足够背景，让它能做判断
- 如果要短答，明确说
- 不要把理解任务的工作外包给 agent
- 不要写“基于你的发现再去修 bug”这种偷懒 prompt
- 应该给到 file path、line、具体改动要求

这其实是在限制“懒 delegation”。也是为什么 Claude Code 的 subagent 效果会更稳：主 agent 被 prompt 强制要求承担 synthesis 责任。

6.4 5.4 built-in agents：职责分工而不是万能 worker

从源码能确认至少有这些内建 agents：

- General Purpose Agent
- Explore Agent
- Plan Agent
- Verification Agent
- Claude Code Guide Agent
- Statusline Setup Agent

这说明 Anthropic 的方向不是让一个 agent 什么都做，而是：

- 探索给 Explore
- 规划给 Plan
- 验证给 Verification
- 通用任务给 General Purpose

这是典型的 specialization 思路。

6.5 5.5 Explore Agent：纯读模式的代码探索专家

exploreAgent.ts 的 system prompt 很有代表性。它明确规定：

6.5.1 绝对只读

- 不能创建文件
- 不能修改文件
- 不能删除文件
- 不能移动文件
- 不能写临时文件
- 不能用重定向 / heredoc 写文件
- 不能运行任何改变系统状态的命令

6.5.2 核心能力

- 用 Glob / Grep / FileRead 快速探索代码库
- Bash 只允许读操作: `ls, git status, git log, git diff, find, grep, cat, head, tail`
- 尽量并行用工具
- 要快, 尽快给结果

这说明 Explore 不是“会搜索的普通 agent”，而是被故意裁成 read-only specialist。

6.6 5.6 Plan Agent: 纯规划, 不做编辑

`planAgent.ts` 的 system prompt 也非常清晰:

- 只读
- 不准改文件
- 需要先理解需求
- 需要探索代码库、模式、架构
- 需要输出 step-by-step implementation plan
- 最后必须列出 Critical Files for Implementation

这里最关键的是: Plan Agent 被定义成 architect / planner, 而不是 executor。这样做的价值是降低角色混杂。

6.7 5.7 Verification Agent: 为什么它非常值钱

`verificationAgent.ts` 是本轮挖掘里非常重要的新增部分。

它的 prompt 非常强, 核心方向不是“确认实现看起来没问题”, 而是:

你的工作是 try to break it

它甚至在 prompt 开头就点出两类失败模式:

1. verification avoidance: 只看代码、不跑检查、写 PASS 就走
2. 被前 80% 迷惑: UI 看起来还行、测试也过了, 就忽略最后 20% 的问题

然后 prompt 强制要求:

- build

- test suite
- linter / type-check
- 根据变更类型做专项验证
- frontend 要跑浏览器自动化 / 页面子资源验证
- backend 要 curl/fetch 实测响应
- CLI 要看 stdout/stderr/exit code
- migration 要测 up/down 和已有数据
- refactor 也要测 public API surface
- 必须做 adversarial probes
- 每个 check 必须带 command 和 output observed
- 最后必须输出 VERDICT: PASS / FAIL / PARTIAL

这说明 Verification Agent 不是“再跑一次测试”，而是一个 adversarial validator。这非常强，因为它把很多 LLM 常见的“差不多就算了”直接用 prompt 反制掉了。

7 6. Agent 调度链深挖：从 AgentTool 到 runAgent 再到 query

这是本次新增的第二个重点：Agent 调度链深挖。

7.1 6.1 总体调用链长什么样

从 AgentTool.tsx 与 runAgent.ts 看，主链路可以抽象为：

1. 主模型决定调用 Agent 工具
2. AgentTool.call() 解析输入
3. 解析是否 teammate / fork / built-in / background / worktree / remote
4. 选择 agent definition
5. 构造 prompt messages
6. 构造 / 继承 system prompt
7. 组装工具池
8. 创建 agent-specific ToolUseContext
9. 注册 hooks / skills / MCP servers
10. 调用 runAgent()
11. runAgent() 内部再调用 query()
12. query 产出消息流
13. runAgent 记录 transcript、处理 lifecycle、清理资源
14. AgentTool 汇总结果或走异步任务通知

这已经是一条非常完整的 subagent runtime pipeline。

7.2 6.2 AgentTool.call(): 真正的调度总控

AgentTool.call() 的职责远比“转发到子 agent”复杂。它要处理：

- 解析输入参数: description、prompt、subagent_type、model、run_in_background、name、team_name、mode、isolation、cwd

- 判断是否 multi-agent teammate spawn
- 解析 team context
- 判断是否允许 background
- 区分 fork path 与 normal path
- 根据 permission rules 过滤 agent
- 检查 MCP requirements
- 计算 selectedAgent
- 处理 remote isolation
- 构造 system prompt / prompt messages
- 注册 foreground / async agent task
- 启动 worktree isolation
- 调用 runAgent()

也就是说，AgentTool 本质上是 agent orchestration controller。

7.3 6.3 fork path vs normal path

源码里有非常明显的分叉：

7.3.1 fork path

- subagent_type 省略且 fork feature 开启
- 继承主线程 system prompt
- 用 buildForkedMessages() 构造 prompt messages
- 用父线程完整 context
- 工具集尽量与父线程一致，保证 prompt cache 命中
- useExactTools = true

7.3.2 normal path

- 明确指定 built-in / custom agent type
- 基于 agentDefinition 生成新的 agent system prompt
- 只给该 agent 所需上下文
- 走该 agent 的 tool restrictions

这里最值钱的地方是：fork 不是“再开一个普通 agent”，而是为了 **cache 和 context 继承专门优化过的执行路径**。

7.4 6.4 为什么 fork 会强调 cache-identical prefix

在注释里可以看出，fork path 会尽量继承父线程的 system prompt 和 tool defs，以保持 API request prefix byte-identical，从而提高 prompt cache 命中。

这是非常高级的设计：

- 普通人只想“子任务能跑”
- Claude Code 想的是“子任务能跑，而且尽量复用主线程 cache，不白烧 token”

这就是产品级系统思维。

7.5 6.5 background agent 与 foreground agent 是两套生命周期

`AgentTool.call()` 会根据条件决定：

- foreground sync path
- async background path
- remote launched path
- teammate spawned path

7.5.1 background path 特点

- 注册 async agent task
- 独立 abort controller
- 可以在后台运行
- 完成后通过 notification 回到主线程
- 可选自动 summarization
- 可查看 outputFile 但 prompt 里明确不鼓励偷看

7.5.2 foreground path 特点

- 主线程等待结果
- 可以在执行中被 background 化
- 有 foreground task 注册与 progress tracking

这说明 Claude Code 对 agent lifecycle 的处理是产品化的，而不是“一次函数调用”。

7.6 6.6 runAgent(): 真正的子 Agent runtime

`runAgent.ts` 负责的事情很多：

- 初始化 agent-specific MCP servers
- 过滤 / 克隆 context messages
- 处理 file state cache
- 获取 system/user context
- 对 read-only agent 做 claudeMd / gitStatus slimming
- 构造 agent-specific permission mode
- 组装 resolved tools
- 获取 agent system prompt
- 创建 abortController
- 执行 SubagentStart hooks
- 注册 frontmatter hooks
- 预加载 frontmatter skills
- 合并 agent MCP tools
- 构造 subagent ToolUseContext
- 调用 `query()` 进入主循环
- 记录 transcript
- 清理 MCP、hooks、perpetto、todo、bash tasks 等资源

这说明 `runAgent` 不是简单 wrapper，而是子 agent 的完整 runtime constructor。

7.7 6.7 agent-specific MCP servers: 真正的 additive 能力注入

`initializeAgentMcpServers()` 很有意思。

它支持 `agentDefinition` 自带 `mcpServers`, 并且可以:

- 从现有配置按名字引用服务器
- 在 `frontmatter` 里内联定义 agent-specific MCP server
- 连接 server
- 拉取 tools
- 把 agent-specific MCP tools 合并进当前 agent 的 tools
- 在 agent 结束时做 cleanup

这说明 agent 不只是消费全局 MCP, 它还可以带自己的外接能力。这对插件 agent / 专职 agent 很强。

7.8 6.8 frontmatter hooks 与 frontmatter skills

`runAgent()` 里还会:

- `registerFrontmatterHooks(...)`
- 读取 `agentDefinition.skills`
- 通过 `getSkillToolCommands()` 加载技能
- 把 skill prompt 内容预加载成 meta user messages 注入初始消息

这很关键: 说明 agent 本身也是可配置的 prompt container, 而不是固定硬编码角色。

7.9 6.9 query() 才是最终主循环执行器

虽然这次没有把 `query.ts` 全文展开, 但从 `runAgent()` 能明确看到:

- 真正的模型对话循环发生在 `query({ ... })`
- `runAgent()` 只是子 agent 的上下文准备与生命周期控制器

这就让整个分层很清楚:

- `AgentTool`: 调度与模式分流
- `runAgent`: 子 agent 上下文构造与生命周期管理
- `query`: 真正的模型消息流与 tool-calling 主循环

7.10 6.10 transcript / metadata / cleanup: 产品化 runtime 的证据

`runAgent()` 里非常多产品级细节:

- `recordSidechainTranscript()`
- `writeAgentMetadata()`
- `registerPerfettoAgent()`
- `cleanupAgentTracking()`
- `killShellTasksForAgent()`
- 清理 session hooks

- 清理 cloned file state
- 清理 todos entry

这说明 Anthropic 并不是只让 subagent “跑起来”，而是把 transcript、telemetry、cleanup、resume 都纳入正式生命周期。

8 7. Skills / Plugins / Hooks / MCP 生态深挖

8.1 7.1 Skill: 不是文档，而是 workflow package

源码里: - SKILL_TOOL_NAME = 'Skill'

在 SkillTool/prompt.ts 以及命令系统相关代码中，它明确要求：

- task 匹配 skill 时必须调用 Skill tool
- 不能只提 skill 不执行
- slash command 可以视为 skill 入口
- 如果 skill 已经通过 tag 注入，则不要重复调用

这说明 Skill 是一个 first-class primitive。

8.1.1 Skill 的本质

可以把它理解成：

- markdown prompt bundle
- 带 frontmatter metadata
- 可声明 allowed-tools
- 可按需注入当前上下文
- 可把重复工作流压缩成可复用能力包

这比“在 system prompt 里塞一堆固定流程”高级很多。

8.2 7.2 Plugin: Prompt + Metadata + Runtime Constraints

关键文件: - src/utils/plugins/loadPluginCommands.ts

插件能提供的能力至少包括：

- markdown commands
- SKILL.md skill 目录
- commandsMetadata
- userConfig
- shell frontmatter
- allowed-tools
- model / effort hints
- user-invocable
- disable-model-invocation

- runtime 变量替换

例如支持:- `${CLAUDE_PLUGIN_ROOT}` - `${CLAUDE_PLUGIN_DATA}` - `${CLAUDE_SKILL_DIR}`
- `${CLAUDE_SESSION_ID}` - `${user_config.X}`

所以 plugin 不是普通 CLI 插件，而是模型行为层面的扩展单元。

8.3 7.3 Hook: 运行时治理层

关键文件: - `src/services/tools/toolHooks.ts`

Hook 支持: - `PreToolUse` - `PostToolUse` - `PostToolUseFailure`

而且 hook 结果不仅仅能“记日志”，还能：

- 返回 message
- `blockingError`
- `updatedInput`
- `permissionBehavior`
- `preventContinuation`
- `stopReason`
- `additionalContexts`
- `updatedMCPToolOutput`

这意味着 Hook 是 runtime policy layer。

8.4 7.4 Hook 与权限的耦合方式非常成熟

`resolveHookPermissionDecision()` 说明：

- hook 可以给出 `allow` / `ask` / `deny`
- 但 hook 的 `allow` 也不自动突破 `settings deny/ask rules`
- 如果需要 `user interaction` 或 `requireCanUseTool`，仍然要走统一 `permission flow`
- hook 还能通过 `updatedInput` 满足交互输入

这说明 Hook 强，但没有绕开核心安全模型。

8.5 7.5 MCP: 不只是工具桥，还是行为说明注入通道

从 `prompts.ts` 可以明确看到：

- `getMcpInstructionsSection()`
- `getMcpInstructions(mcpClients)`

逻辑是：- 如果 `connected MCP server` 提供 `instructions` - 就把这些 `instructions` 拼进 `system prompt`

也就是说 MCP 能同时注入：

1. 新工具
2. 如何使用这些工具的说明

这让 MCP 的价值远高于简单 `tool registry`。

9 8. 权限、Hook、工具执行链深挖

9.1 8.1 toolExecution.ts: 真正的工具 runtime 主线

Claude Code 的工具执行并不是“模型决定 → 直接跑函数”。实际链路大致是：

1. 找 tool
2. 解析 MCP metadata
3. 做 input schema 校验
4. 做 validateInput
5. 为 Bash 启动 speculative classifier check
6. 运行 PreToolUse hooks
7. 解析 hook permission result
8. 走 permission 决策
9. 再次根据 permission updatedInput 修正输入
10. 真正执行 tool.call()
11. 记录 analytics / tracing / OTEL
12. 运行 PostToolUse hooks
13. 处理 structured output / tool_result block
14. 失败则走 PostToolUseFailure hooks

这是一条标准的 runtime pipeline，而不是直连函数调用。

9.2 8.2 输入校验：先挡低级错误

工具执行前会先做：

- Zod schema parse
- tool-specific validateInput

如果失败：- 直接生成 tool_result 错误消息 - 记录 tengu_tool_use_error
这保证模型随便乱生成参数时不会直接污染执行层。

9.3 8.3 PreToolUse hooks: 最关键的拦截点

在 runPreToolUseHooks() 中，hook 可以产出：

- 普通 message
- hookPermissionResult
- hookUpdatedInput
- preventContinuation
- stopReason
- additionalContext
- stop

最关键的几个能力是：

9.3.1 updatedInput

hook 可以改写输入，但不一定做权限决策。

9.3.2 permissionBehavior

hook 可以直接说: - allow - ask - deny

9.3.3 preventContinuation

即使没 deny, 也能阻止流程继续。

这使得 Hook 能真正参与控制流。

9.4 8.4 resolveHookPermissionDecision(): 权限语义的关键粘合层

这段逻辑非常值钱。它定义了:

- hook allow 不一定绕过 settings 规则
- 如果 tool 要求 user interaction, 而 hook 没提供 updatedInput, 则仍要走 canUseTool
- ask 类型 hook 会作为 forceDecision 传递下去
- deny 类型直接生效

也就是说, Hook 的权限语义是被严格嵌进总权限模型里的, 不是外挂旁路。

9.5 8.5 工具执行后也不是结束

runPostToolUseHooks() 与 runPostToolUseFailureHooks() 说明, Claude Code 不把 “tool 成功返回” 当终点。

成功后 hook 还能: - 追加 message - 注入 additional context - 阻断 continuation - 对 MCP tool output 进行更新

失败后 hook 还能: - 补充失败上下文 - 发阻断说明 - 给用户更多恢复线索

这就是为什么整个系统比 “tool call 一把梭” 可治理得多。

10 9. 为什么 Claude Code 这么强: 从源码看它真正的护城河

10.1 9.1 它不是一个 prompt, 而是一个 operating model

很多人复刻 coding agent 时只会拿走:

- 一个 system prompt
- 一个文件编辑工具
- 一个 bash 工具
- 一个 CLI 壳

但 Claude Code 真实的护城河是:

- Prompt architecture
- Tool runtime governance
- Permission model
- Hook policy layer
- Agent specialization

- Skill workflow packaging
- Plugin integration
- MCP instruction injection
- Prompt cache optimization
- Async/background/remote lifecycle
- Transcript / telemetry / cleanup / task system

少一个都行，但会显著掉“手感”。

10.2 9.2 它把“好行为”制度化了

Claude Code 最大的优势之一，不是模型更聪明，而是：

它不把“好习惯”交给模型即兴发挥，而是写进 prompt 和 runtime 规则里。

例如：- 不要乱加功能 - 不要过度抽象 - 不要瞎重试被拒绝的工具 - 不要未验证就说成功 - 不要随便做风险操作 - 不要让 fork 输出污染主上下文 - 匹配 skill 时必须执行 skill - verification
不能只看代码，必须跑命令

这种制度化，会极大提高系统一致性。

10.3 9.3 它特别懂“上下文是稀缺资源”

源码中大量设计都在围绕上下文做优化：

- system prompt 动静边界
- prompt cache boundary
- fork path 共享 cache
- skill 按需注入
- MCP instructions 按连接状态注入
- function result clearing
- summarize tool results
- compact / transcript / resume

这说明他们不是把 token 当免费空气，而是当 runtime 预算来管理。

10.4 9.4 Agent specialization 是很深的设计优势

Explore / Plan / Verification 这套 built-in agents 的价值，不在于“多了三个 agent”，而在于：

- 研究和探索不用污染主线程
- 规划和实现分离，降低混乱
- 验证独立出来，对抗“实现者 bias”

很多系统的问题，就是一个 agent 既研究、又规划、又实现、又验收，最终哪件事都不够稳定。

Claude Code 则是明确分工。

10.5 9.5 它的生态不是“可安装”，而是“模型可感知”

这是 Claude Code 另一个很强的点。

很多系统也有插件，也有工具，也有外部协议，但模型本身不知道：- 有哪些扩展 - 什么时候该用 - 怎么用

Claude Code 不一样。它通过：- skills 列表 - agent 列表 - MCP instructions - session-specific guidance - command integration

让模型“知道自己的扩展能力是什么”。这才是生态真正能发挥作用的关键。

11 10. 关键文件索引与后续可继续深挖方向

11.1 10.1 核心 Prompt 文件

11.1.1 主系统提示词

- `src/constants/prompts.ts`

11.1.2 Agent Tool Prompt

- `src/tools/AgentTool/prompt.ts`

11.1.3 Skill Tool Prompt

- `src/tools/SkillTool/prompt.ts`

11.1.4 其他可继续挖的 tool-specific prompt

- `src/tools/FileReadTool/prompt.ts`
- `src/tools/GlobTool/prompt.ts`
- `src/tools/GrepTool/prompt.ts`
- `src/tools/BriefTool/prompt.ts`
- 以及更多 `prompt.ts`

11.2 10.2 核心 Agent 文件

- `src/tools/AgentTool/AgentTool.tsx`
- `src/tools/AgentTool/runAgent.ts`
- `src/tools/AgentTool/resumeAgent.ts`
- `src/tools/AgentTool/forkSubagent.ts`
- `src/tools/AgentTool/agentMemory.ts`
- `src/tools/AgentTool/agentMemorySnapshot.ts`
- `src/tools/AgentTool/builtInAgents.ts`

11.2.1 built-in agents

- `src/tools/AgentTool/built-in/exploreAgent.ts`
- `src/tools/AgentTool/built-in/planAgent.ts`

- `src/tools/AgentTool/built-in/verificationAgent.ts`
- `src/tools/AgentTool/built-in/generalPurposeAgent.ts`
- `src/tools/AgentTool/built-in/claudeCodeGuideAgent.ts`
- `src/tools/AgentTool/built-in/statuslineSetup.ts`

11.3 10.3 核心 Skill / Plugin / Hook / MCP 文件

11.3.1 Skill

- `src/tools/SkillTool/constants.ts`
- `src/tools/SkillTool/prompt.ts`
- `src/commands.ts`

11.3.2 Plugin

- `src/utils/plugins/loadPluginCommands.ts`

11.3.3 Hook

- `src/services/tools/toolHooks.ts`
- `src/utils/hooks.js`

11.3.4 Tool execution

- `src/services/tools/toolExecution.ts`

11.3.5 MCP

- `src/services/mcp/types.ts`
- `src/services/mcp/normalization.ts`
- `src/services/mcp/mcpStringUtils.ts`
- `src/services/mcp/utils.ts`
- `src/entrypoints/mcp.ts`

11.4 10.4 如果继续做下一轮，还能怎么挖

如果要再往下继续深挖，下一轮我建议重点看：

1. `query.ts`: 主会话循环与模型交互流
2. `resumeAgent.ts`: agent 恢复机制
3. `loadSkillsDir`: skills 完整加载链
4. `pluginLoader`: 插件加载与内建插件生态
5. `systemPromptSections.ts`: prompt section registry 细节
6. `coordinator/*`: 多 agent 协调器模式
7. `attachments.ts`: skill / agent listing / MCP delta 的消息注入方式
8. `AgentSummary`: 后台 agent 进度总结机制

11.5 10.5 最终结论

如果只给一句话总结这份增强版研究报告：

Claude Code 的真正秘密，不是一段 system prompt，而是一个把 prompt architecture、tool runtime、permission model、agent orchestration、skill packaging、plugin system、hooks governance、MCP integration、context hygiene 和 product engineering 全部统一起来的系统。

这就是为什么它不像一个“会调工具的聊天机器人”，而更像一个真正可扩展、可治理、可产品化的 Agent Operating System。