Claude Code

从 Source Map 看一个 AI Agent Runtime 的内部构造

基于 npm 包 @anthropic-ai/claude-code v2.1.88
通过 cli.js.map 还原的 4,756 个源文件

核心论点：这不是“带工具的聊天程序”，而是一个完整的 Terminal Agent Runtime Platform

Query Loop
递归执行内核

Tool Runtime
40+ 内置工具

Permission System
多层治理架构

Multi-Agent
Swarm 协作模式



phodal 公众号

背景 · 01

这是从 npm 包 Source Map 还原的高保真快照，而不是官方 monorepo

理解来源，是正确解读架构的前提。

还原过程

- npm 包内包含完整 cli.js.map（Source Map）
- extract-sources.js 提取 sourcesContent 字段
- 逐文件写回 restored-src/，做 webpack 路径清洗
- 还原版本：v2.1.88，共 4,756 个文件
- 包含 1,884 个 .ts/.tsx 源文件，总计 512,664 行

分析可信度

- 运行时主链、类型签名、注释：高度可信
- 目录结构、模块边界：与发布版一致
- 构建脚本、测试框架：可能不完整
- 内部工具链、CI 配置：不在 source map 范围
- 结论：足够支撑架构与设计决策分析

规模数据一眼就能判断：这是平台级产品，不是小工具

从文件数量、目录分布和功能模块密度，可以还原出系统的真实复杂度。

4,756	1,884	512K	40+	101
还原文件总数	.ts / .tsx 文件	代码行数	内置工具	内置命令
含各类资源与配置	纯 TypeScript 源码	TypeScript 总计	tools/ 目录	commands/ 目录

关键目录分布（按功能层划分）

核心运行时

QueryEngine.ts
query.ts
main.tsx
entrypoints/

工具层 (40+)

BashTool
FileEditTool
AgentTool
WebSearchTool + 37 more

服务层

services/mcp/
services/compact/
services/api/
services/lsp/

扩展与能力

plugins/
skills/
commands/ (101)
coordinator/

非典型功能

voice/
vim/
buddy/
memdir/

SECTION

架构全景

从入口分流到 Agent 协作，梳理这套系统的五层结构。

01

先建立大图，再逐层拆解。
不要在细节里迷路。

五层架构：从「用户输入」到「模型递归执行」的完整通路

这五层不是简单的功能列表，而是被 Query Loop 串起来的有序执行体系。

01

入口层

entrypoints/cli.tsx

轻量分发器：版本快路径 / 模式分流 / 懒加载

02

启动编排层

main.tsx （4,683 行）

初始化 Config · Auth · MCP · Agents · UI

03

会话核心层

QueryEngine.ts + query.ts

会话状态 · 递归 Turn Loop · Budget 管理

04

执行治理层

toolExecution.ts + toolHooks.ts

Pre/Post Hook · 权限多层决策 · 追踪

05

扩展与能力层

Agents · MCP · Skills · Commands · Plugins

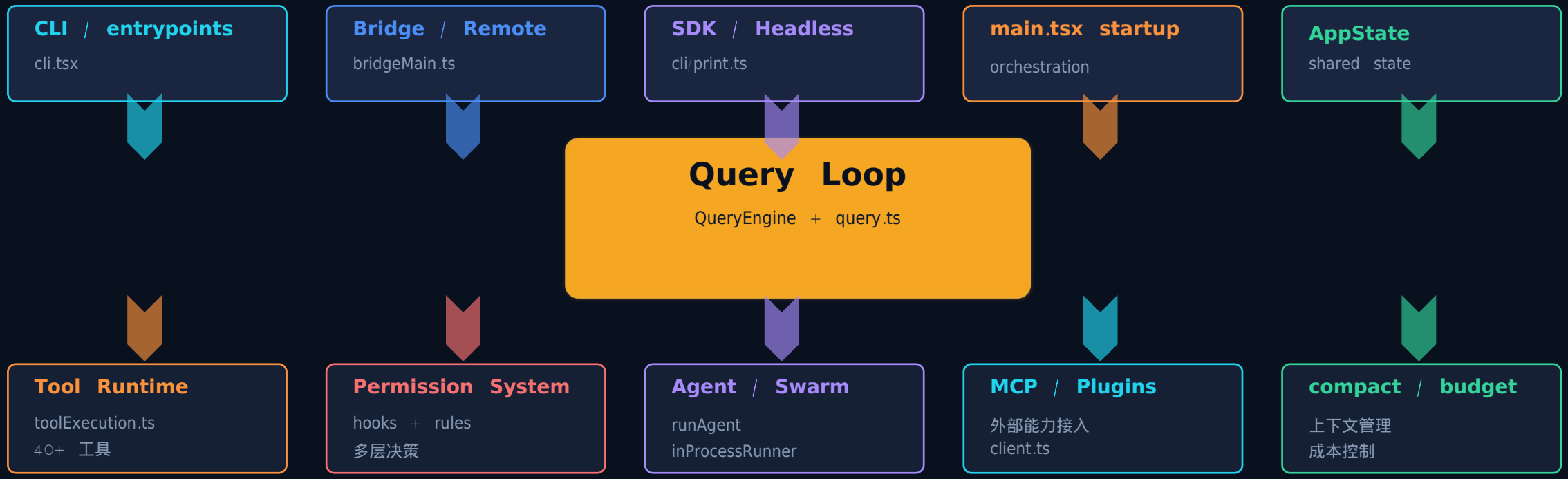
多 Agent Swarm · 外部能力接入

为什么这五层很重要

- 入口层做到「极速分流」：--version 等命令不需要加载 main.js，大幅减少启动延迟
- main.tsx 是平台级 orchestration，4,683 行的 startup 涵盖了 Auth / Feature Gates / MCP / LSP / Session Restore
- QueryEngine 与 query.ts 职责分离：前者管会话生命周期，后者管单轮递归状态机
- 执行治理层不是可选 middleware，而是工具调用的必经主路径
- 扩展层可以从 Plugin / Skill / MCP 三条通道接入，但最终都折叠进共享 Tool Surface
- 不同 Surface（Terminal / Bridge / SDK）只改变入口和输出，不改变中间三层

模块关系图：被 Query Loop 串起来的多层系统

真正的架构边界不是文件夹，而是这些模块之间的依赖方向和数据流。



五层架构全景——从 CLI 入口到扩展生态的层次关系

每一层的边界和方向性是理解系统的关键。

入口层	**cli.tsx** 版本快路径 / 模式分流	**--version** 不加载 main.js	**--help** 极速输出	**full mode** lazy require main.tsx
启动编排	**main.tsx** 4,683 行 orchestration	**Auth + Config** OAuth / API key	**MCP + LSP init** 外部服务连接	**Session restore** 历史会话恢复
会话核心	**QueryEngine** 会话生命周期管理	**query.ts** 递归 Turn 状态机	**buildQueryConfig** 运行时配置快照	**Stream + Collect** 流式消费 tool_use
执行治理	**toolExecution** 工具执行主路径	**Pre/Post Hooks** 策略注入点	**Permissions** 多层决策树	**Bash Classifier** 自动风险分类
扩展能力	**AgentTool** 子代理 → query()	**MCP Client** 外部 server 能力	**Skills + Plugins** 工作流封装	**101 Commands** 用户显式入口

SECTION

Query Loop
执行内核

从用户按下回车，到 agent 递归执行，系统究竟怎么串起来的？

02

这里才是系统真正的大脑，
而不是 UI 或 CLI 入口。

QueryEngine 与 query.ts：会话生命周期 vs 单轮状态机

这是理解整套系统最关键的一对概念。两者职责完全不同，却又紧密配合。

QueryEngine（会话对象）

- one QueryEngine per conversation（注释明确说明）
- 维护 mutableMessages：整个会话累积的全部消息
- 管理 abortController：支持终止单轮或整个会话
- 记录 permissionDenials / totalUsage / readFileState
- 封装 SDK/headless 接口，对外提供稳定 API
- 处理 orphaned permission / skill discovery / session persist
- submitMessage() 启动新一轮 turn，把实际递归交给 query()

query.ts（单轮状态机）

- buildQueryConfig()：将当次 turn 所需配置快照下来
- 调用模型 API，流式消费 stream_event
- 并行维护三个集合：assistantMessages / toolResults / toolUseBlocks
- 检测到 tool_use → 不结束，而是切入执行阶段
- runTools() / StreamingToolExecutor 执行所有工具
- tool results 回注为新的 user messages → 进入下一轮
- needsFollowUp / maxTurns / compact 共同控制边界

一次 Turn 的完整执行时序

从 submitMessage() 到「继续还是停止」——每一步都在 query.ts 中有对应代码。



关键设计信号

stop_reason 不可靠

stop_reason=tool_use 在流式场景下有缺陷，因此 query.ts 在流式过程中显式收集 toolUseBlocks 作为判断依据

工具结果不是副作用

tool results 被编码成新的 user/attachment messages 回注对话，成为下一轮推理的输入——这是 agent loop 的核心机制

StreamingToolExecutor

feature gate 打开时，启用 StreamingToolExecutor 允许工具在模型仍在输出时并行开始执行，降低端到端延迟

Budget & Compact

taskBudget / compactWarning / microCompact 共同保证长时任务不会因 context 膨胀而崩溃

长任务稳定性：Fallback 恢复、Context 压缩与预算管理

这部分代码量不小，但它们是 Claude Code 能真正跑完大型任务的基础设施。

流式 Fallback 恢复

- prompt-too-long / media-size / max-output-tokens 错误会先被 withheld
- recovery 逻辑判断是否能自救
- 如需 fallback：给旧 assistant message 发 tombstone
- 清空旧的 toolUse / toolResult 暂存
- 重建 StreamingToolExecutor（避免旧 tool_use_id 泄漏进 retry）

Context 压缩策略

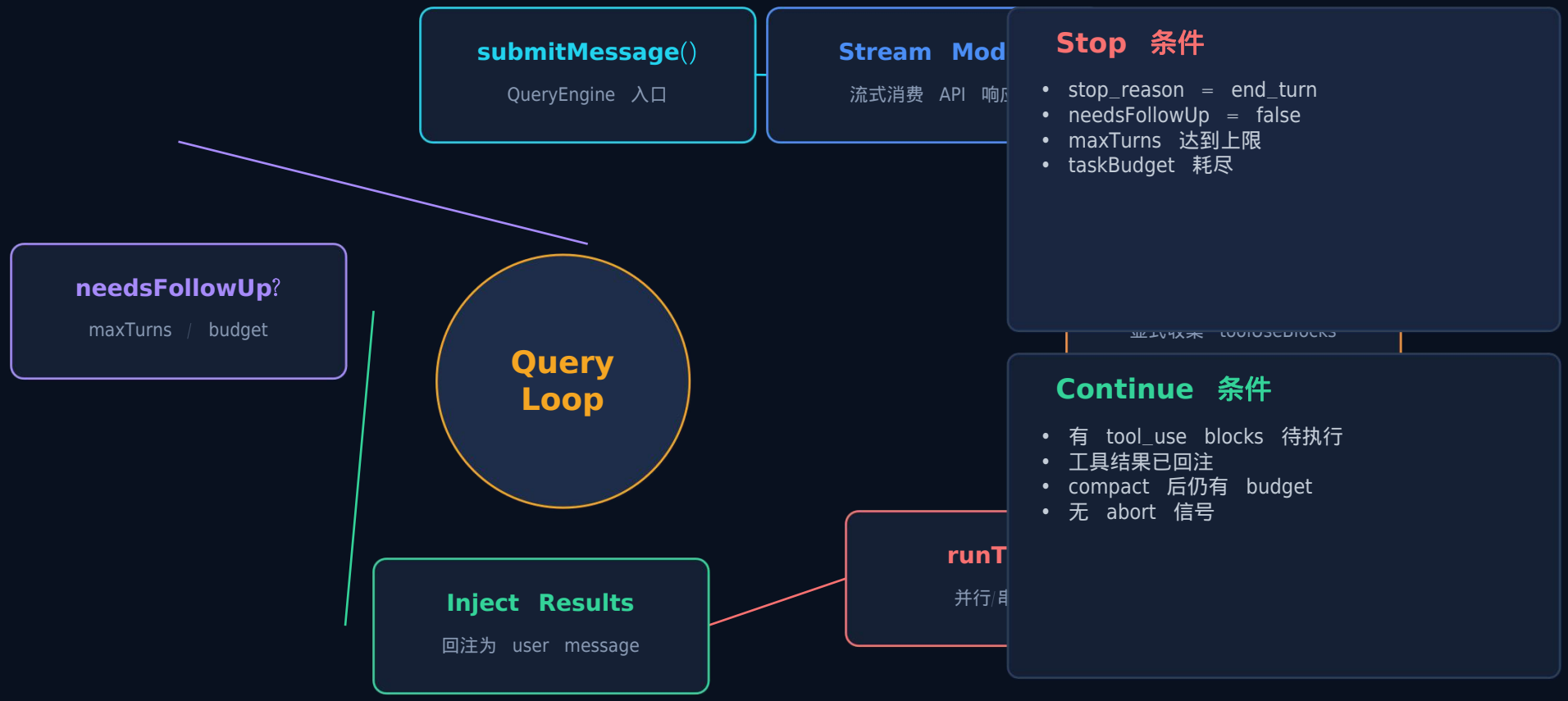
- autoCompact：自动触发上下文压缩
- microCompact：轻量级压缩，保留最近 turns
- compact：完整会话摘要替换
- sessionMemoryCompact：跨会话记忆压缩
- compactWarning：提前预警，给用户调整机会

预算与续跑控制

- taskBudget：整个 agentic turn 的 API 调用预算
- 动态计算 token budget / output tokens
- continuation 计数防止无限递归
- fallback model 切换（降级到更便宜的模型）
- maxTurns 硬限制作为最后防线

Turn 循环图——Agent Loop 的核心执行路径

每轮 turn 都是完整的 Stream → Collect → Execute → Inject → Decide 循环。



SECTION

Tool Runtime & Permission

模型提出动作，runtime 决定能不能做、怎么做——这才是真正的「受控执行」。

03

权限不是外围 UI，
而是工具执行主路径的组成部分。

40+ 内置工具：覆盖从代码编辑到系统管理的完整工作流

工具不是简单的函数，而是带有 schema、权限声明、hook 交互能力的可治理执行单元。

文件操作

FileReadTool

FileWriteTool

FileEditTool · GlobTool · GrepTool

代码执行

BashTool · PowerShellTool · REPLTool

(JS) · NotebookEditTool

Agent 协作

AgentTool

TaskCreateTool

TaskGetTool

外部集成

MCPTool

WebSearchTool

WebFetchTool

McpAuthTool · ListMcpResourcesTool

开发辅助

LSPTool

AskUserQuestionTool

SkillTool

BriefTool · TodoWriteTool · ToolSearchTool

流程控制

EnterPlanModeTool

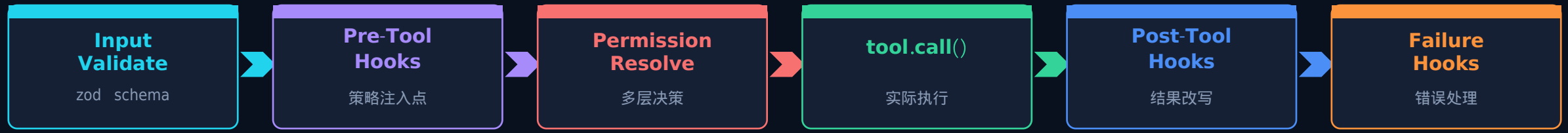
ExitPlanModeTool

EnterWorktreeTool

SleepTool · ScheduleCronTool

Hook 不是扩展点，而是执行主路径的组成部分

Pre Hook / 权限决策 / Post Hook / Failure Hook 贯穿工具执行的全生命周期。



Pre Hook 能返回什么

- message：直接注入 UI / transcript
- hookPermissionResult：allow / ask / deny
- hookUpdatedInput：只改输入，不改权限
- preventContinuation：阻断后续控制流
- additionalContext：附加上下文给模型

权限决策优先级

- ① Hook deny → 直接拒绝
- ② Hook allow → 再跑 checkRuleBasedPermissions
- ③ Rule deny/ask → 可覆盖 hook allow
- ④ 无 hook 决策 → 进入 canUseTool() 流程
- 结论：权限是 hook + rule + UI 三层叠加

Post / Failure Hook

- 成功：可产出额外 attachment / message
- 成功：可 preventContinuation 阻止继续
- MCP tool：可 updatedMCPToolOutput 改写输出
- 失败：runPostToolUseFailureHooks() 策略化处理
- 因此成功与失败都被统一纳入 hook 协议

权限系统：不是一个 bool，而是 Mode + Rules + Classifier + Shell Analysis

这套系统让 Claude Code 可以被企业策略、本地规则和用户偏好共同治理。

Permission Modes

- default：默认交互模式
- plan：只规划，不执行
- acceptEdits：自动接受文件修改
- bypassPermissions：受信任模式
- dontAsk：静默模式
- auto：自动（feature gated）
- 每个 mode 都有 title / symbol / color / external 映射，同时服务策略、UI 和对外接口

规则系统

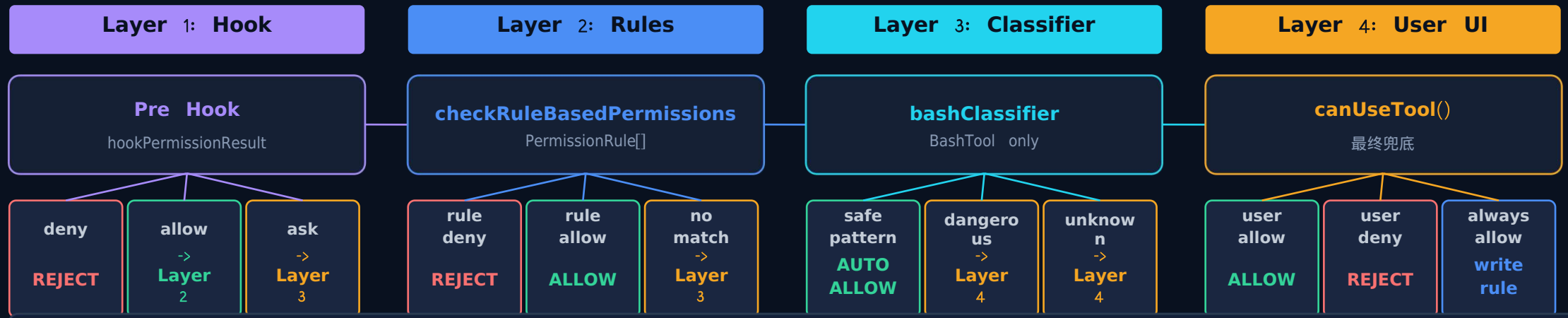
- permissionRuleParser.ts 解析规则字符串
- 例：Bash(npm install) / Edit(src/**)
- shellRuleMatching.ts 匹配 shell 命令
- filesystem.ts 处理读写范围与路径分析
- shadowedRuleDetection.ts 检测冗余规则
- 规则可来自：CLAUDE.md / 企业策略 / 用户设置 / 会话临时授权

Bash Classifier（自动审批）

- bashClassifier.ts 对 shell 命令做风险分类
- yoloClassifier.ts 在 bypass 模式下处理
- dangerousPatterns.ts 维护危险命令模式库
- classifierDecision.ts 给出审批建议
- 高置信度安全命令可跳过用户确认
- 这使 agent 在 auto 模式下能高效运行
- 同时保留对高风险命令的拦截能力

权限决策树——一次工具调用经历的四层判断

从 Hook 到 Rule 到 Classifier 到 UI，每层都可能中断或放行。



权限系统的工程意义

不可绕过

权限检查在 toolExecution.ts 主路径上，不是可选 middleware。跳过执行 = 跳过权限检查 = 工具不会被调用。

层层叠加

Hook deny 最优先，但 Hook allow 不代表通过——Rule 层仍可覆盖。这使企业策略 > 用户偏好成为可能。

自动化友好

bashClassifier + yoloClassifier 让 auto 模式下安全命令跳过确认，同时保留对 rm -rf / sudo 等的拦截。

可追溯

permissionDenials 在 QueryEngine 中累积，denialTracking.ts 记录完整决策链路，可用于事后审计。

SECTION

Multi-Agent Swarm

多 Agent 不是演示特性，而是带状态、可追踪、可治理的 Actor 子系统。

04

主 agent 只是第一个 agent，
系统从一开始就允许递归生成更多。

Multi-Agent · 01

AgentTool → runAgent → inProcessRunner：从工具调用到子 Agent 运行时

这条链路证明：多 Agent 是 Query Loop 的内生能力，不是外部 orchestration shell。



runAgent() 关键逻辑

- 计算子代理使用的 model（可与父代理不同）
- 创建 agentId，设置 transcript 子目录与 trace 层级
- 构造 initialMessages，过滤 incomplete tool calls
- 根据 agent type 裁剪 claudeMd / gitStatus 等上下文
- 子代理不继承父代理所有上下文——这是关键设计决策
- 最终调用 query()，形成子代理递归 loop

inProcessRunner 额外职责

- AsyncLocalStorage / teammate context 做上下文隔离
- 把进度更新和 AppState 写回 leader 可见状态
- 完成后给 leader 发 idle notification
- 支持 plan mode approval flow / abort cleanup
- 权限申请优先通过 leader UI bridge，不可用时降级为 mailbox sync

Teammate 是持续存活的 Actor，而不是一次性函数调用

waitForNextPromptOrShutdown() 实现的 idle 循环，让 subagent 能响应多轮任务。



每 500ms 轮询

Mailbox 通信

- leader / teammate 通过 mailbox 交换消息
- 优先处理 shutdown request
- 再处理 team lead 消息
- teammateMailbox.ts 管理消息队列

Task Claim 系统

- tasks.ts 维护团队任务列表
- 空闲时主动尝试 claim 可用任务
- 任务有 owner 概念，防止重复执行
- 完成后写回 task output，持久化

Permission Bridge

- worker ask 权限 → 先尝试 leader UI queue
- leader UI 不可用 → 降级为 mailbox sync
- 成功时写回 permission updates 到 leader context
- worker 不会绕开权限系统

Built-in Agents

- builtInAgents.ts：feature-gated agent registry
- Explore / Plan / Verification / Code Guide
- SDK 场景可通过环境变量禁用
- coordinator mode 下使用 workerAgent.js

SECTION

扩展层与多 Surface

MCP / Skills / Plugins / Commands 如何汇合；
Terminal / Bridge / SDK 如何共用一套引擎。

05

不同入口改变的只是 ingress/egress，
推理内核始终是同一套。

扩展层 · 01

Plugin / Skill / Command / MCP：四条通道，最终折叠进同一个 Tool Surface

这四层经常被混淆，但在源码里边界非常清晰，职责完全不同。



Plugin

管分发、安装、manifest，最终转化为可用工具或命令

Skill

封装工作流模板，很多情况下最终下沉为 command

Command

用户显式调用入口，101 个命令覆盖开发全流程

MCP

把外部 server 的 tools / resources 映射进本地 runtime，仍受本地治理

扩展层 · 02

Terminal / Bridge / SDK：不同 Surface 共用同一套推理引擎

这是「same engine across surfaces」在源码层面的真实落点。

Terminal REPL

ink/components/App.tsx
AppStateProvider
cli/print.ts

- Ink（React 终端 UI）渲染交互界面
- AppState 共享运行时状态
- 支持 Vim 模式（vim/ 目录）
- 权限弹窗直接在终端 UI 显示
- 支持 voice/ 语音交互（实验性）

Bridge / Remote

bridge/bridgeMain.ts
initReplBridge.ts
bridgeUI.ts

- 把远端消息重新注入本地消息队列
- replBridge.ts 建立双向通道
- bridgeUI.ts 在 bridge 模式显示 UI
- sessionRunner.ts 管理远程会话生命周期
- trustedDevice.ts 处理远端认证

SDK / Headless

QueryEngine.ts SDK 接口
cli/print.ts 结构化输出
remote/ 目录

- QueryEngine 对外提供稳定 SDK API
- cli/print.ts 格式化结构化输出
- 可通过环境变量禁用 built-in agents
- sdkMessageAdapter.ts 适配消息格式
- SessionsWebSocket.ts 管理 SDK 会话

特色功能 · 01

这些你可能没想到它会有：Voice、Vim、Buddy、KAIROS

这些不是边角功能——它们暗示了产品对「AI 工作方式」更深远的设想。

Voice 模式

services/voice.ts
services/voiceStreamSTT.ts

- 流式语音转文字（STT）
- voiceModeEnabled.ts 控制开关
- voiceKeyterms.ts 关键词识别优化
- 集成进主 UI 交互流程

Vim 模式

vim/motions.ts
vim/operators.ts
vim/textObjects.ts

- 完整的 Vim 动作系统
- motions / operators / textObjects
- transitions.ts 管理模式切换
- 在终端编辑器内提供 Vim 体验

AI Buddy（伴侣 UI）

buddy/companion.ts
buddy/CompanionSprite.tsx
buddy/sprites.ts

- 像素风格 AI 伴侣动画精灵
- useBuddyNotification 推送通知
- companion.ts 管理伴侣状态
- 通过精灵动画表达工作状态

Memory / memdir

memdir/memdir.ts
memdir/findRelevantMemories.ts
services/SessionMemory/

- 结构化记忆文件目录系统
- findRelevantMemories 语义搜索
- teamMemPrompts 团队共享记忆
- 跨会话持久化 agent 经验

特色功能 · 02

Voice 模式：Push-to-Talk，接入 Anthropic 语音流服务

不是通用 TTS/STT 库封装——而是直连 claude.ai voice_stream WebSocket，要求 OAuth 登录。

实现架构

- services/voice.ts — 音频录制主服务（525 行）
- 原生 audio-capture-napi（CoreAudio / ALSA）
- fallback：SoX rec 或 arecord（Linux）
- dlopen 懒加载——首次按键触发，避免启动卡顿
-
- services/voiceStreamSTT.ts — 流式 STT（544 行）
- WebSocket 连接 /api/ws/speech_to_text/voice_stream
- 协议：JSON 控制帧 + 二进制音频帧
- 服务端响应 TranscriptText / TranscriptEndpoint
- 8s KeepAlive 保活，5s safety timeout
-
- services/voiceKeyterms.ts — 关键词识别优化（106 行）
- voice/voiceModeEnabled.ts — GrowthBook 开关

Push-to-Talk 工作方式

- 按住快捷键 → 开始录音（16kHz 单声道）
- 松开 → 发送 CloseStream，等待 finalize()
- 沉默检测：SoX 2s / 3% 阈值（自动停止模式）
- 连接音频帧流式推送，边录边传，降低延迟

权限与限制（源码直接说明）

- 必须 Anthropic OAuth 登录——API key 不支持
- Bedrock / Vertex / Foundry 均不可用
- tengu_amber_quartz_disabled GrowthBook 紧急关闭开关
- WSL1 / headless Linux：arecord 需 ALSA/PulseAudio
- WSL2+WSLg（Win11）通过 RDP pipes 支持
- 冷启动 dlopen 可能阻塞事件循环 1–8s（已有注释说明）

特色功能 · 03

Vim 模式：1,513 行的完整终端编辑器状态机

不是「支持几个快捷键」——而是严格的 INSERT / NORMAL 双模式状态机，含 dot-repeat 和 text objects。

状态机设计（types.ts 注释原文）

```
VimState
    INSERT    (tracks insertedText)
    NORMAL    — CommandState 子状态机

    idle ─┬─[d/c/y]──► operator
          ├─[1-9]────► count
          ├─[fFtT]───► find
          ├─[g]──────► g
          ├─[r]──────► replace
          └─[><]─────► indent

    operator ─┬─[motion]──► execute
              ├─[0-9]─────► operatorCount
              ├─[ia]──────► operatorTextObj
              └─[fFtT]────► operatorFind
```

实现文件（1,513 行）

- types.ts（199 行）——状态机类型即文档
- transitions.ts（490 行）——状态转移核心
- operators.ts（556 行）——delete / change / yank
- textObjects.ts（186 行）——w W ([{ < " ' ` …
- motions.ts（82 行）——h j k l w b e ^ $ …

支持特性（源码验证）

- 完整 operators：delete(d) / change(c) / yank(y)
- Text objects：word / WORD / 括号 / 引号 / 花括号
- Find motions：f F t T（含反向）
- Count prefix：3dw / 2dd / 5j 等
- Dot-repeat（.）：RecordedChange 记录完整重放
- Register：yank/paste 寄存器 + linewise 标记
- MAX_VIM_COUNT = 10000（防止卡死）

特色功能 · 04

Buddy：你的 AI 伴侣——18 个物种、5 个稀有度、确定性孵化算法

源码里的隐藏彩蛋：2026 年 4 月 1 日 Teaser 窗口，基于 userId hash 生成唯一伴侣。

角色生成系统（1,298 行）

- 18 种物种：duck / goose / cat / dragon / octopus /
- owl / penguin / turtle / snail / ghost / axolotl /
- capybara / cactus / robot / rabbit / mushroom / chonk /
blob
-
- 5 个稀有度（按权重）：
- common 60% / uncommon 25% / rare 10%
- epic 4% / legendary 1%
-
- 6 种眼睛：· ✦ × ◉ @ °
- 8 种帽子：crown / tophat / propeller / halo /
- wizard / beanie / tinyduck（common 无帽子）
- Shiny 概率：1%
-
- 5 维属性：DEBUGGING / PATIENCE / CHAOS / WISDOM / SNARK

确定性孵化算法

- mulberry32（seeded PRNG）+ hashString(userId + SALT)
- SALT = 'friend-2026-401'（固定种子）
- 同一 userId 永远孵化同一只伴侣
- Bones 每次从 hash 重新生成，不持久化
- — 防止用户编辑 config 作弊稀有度
- Soul（name + personality）由模型生成，持久化
- Bun.hash 加速（Bun 环境下用原生 hash）

发布窗口（源码硬编码）

- Teaser 窗口：2026-04-01 ~ 04-07（本地时区）
- — 跨时区滚动，避免 UTC 午夜流量峰值
- 功能永久上线：2026 年 4 月之后
- feature('BUDDY') gate 控制全局可见性
- companion.ts 注释：'good enough for picking ducks'

特色功能 · 05

KAIROS：隐藏在 feature gate 背后的 Assistant 模式

代码里出现 20+ 次 feature('KAIROS')——这是一套与 coding agent 并列的 AI 助手运行模式。

KAIROS 是什么

- assistant/ 目录下的独立运行模块
- assistant/index.js：isAssistantMode() / initializeAssistantTeam()
- assistant/gate.js：isKairosEnabled()——GrowthBook 动态开关
- assistant/sessionHistory.ts：会话历史翻页 API
- 通过 --assistant 命令行参数强制启用
- defaultView: 'chat' 时触发 brief-only 模式

与 Coding Agent 模式的差异

- 独立 assistantTeamContext——不共用 coding agent team
- 有 pendingAssistantChat（session discover 逻辑）
- assistantActivationPath 影响工具可见性
- getUserMsgOptIn() 控制 brief-only 初始状态
- KAIROS_BRIEF / KAIROS_CHANNELS 是子特性开关

关联子特性

- KAIROS_BRIEF — BriefTool 可见性（检查点汇报）
- KAIROS_CHANNELS — 允许 --channels / --dangerously-load-development-channels
- PROACTIVE — 主动触发，与 KAIROS 部分逻辑共享

源码直接可见的信号

- main.tsx 78 行：Dead code elimination 注释
- — 说明 KAIROS 模块在外部构建中被剔除
- tengu_kairos GrowthBook 标志控制实际启用
- session history 使用 ccr-byoc-2025-07-29 beta 头

可以合理推断

- KAIROS 是 Claude.ai 对话能力接入 Claude Code 的通道
- 与 Bridge/Remote 共同构成多 surface 战略
- 目前仅对内部或白名单用户开放

总结 · 01

如何继续阅读源码：建议按这条顺序推进

先建立全局模型，再逐层下钻——避免在细节里迷失。

第一轮

建立系统大图（理解 70% 行为）

entrypoints/cli.tsx

入口分流逻辑与启动策略

main.tsx

startup 编排全景（4,683 行，建议用 outline view）

QueryEngine.ts

会话对象设计与 SDK 接口

query.ts

agent loop 的真正核心，递归状态机

Tool.ts + tools.ts

tool 抽象模型与 registry

第二轮

攻克执行与扩展层（理解剩余 30%）

services/tools/toolExecution.ts

工具执行主路径与 hook 编排

services/tools/toolHooks.ts

Pre/Post hook 协议细节

tools/AgentTool/ + utils/swarm/

多 Agent 运行时

services/mcp/client.ts

外部能力接入协议

utils/permissions/

权限系统全景

三个可以直接带走的结论

这些结论由源码事实支撑，不是主观评价。

01

它是 Terminal Agent Runtime，不是「聊天 CLI」

系统中心是 QueryEngine / query.ts 构成的递归 Agent Loop，而不是终端 UI 本身。UI、Bridge、SDK 只是不同的 surface，共享同一套推理内核。4,683 行的 main.tsx 启动编排直接说明了这是平台级产品。

02

权限系统是架构必需品，不是后期添加的防护

Hook + Rules + Classifier + Mode 四层构成的权限决策树，位于工具执行的主路径上，不是可绕过的 middleware。这使它能同时服务本地开发者、企业策略和 headless 自动化场景。

03

多 Agent 是可治理的 Actor 系统，不是并发调用模型

inProcessRunner 的 Actor 模型（mailbox + task claim + idle loop）+ 权限 bridge 机制，让子代理在不绕开任何治理的情况下真正协作。这才是「multi-agent」从概念变成工程的关键。

Thanks

Claude Code Sourcemap 源码解读

基于 @anthropic-ai/claude-code v2.1.88 source map 还原分析

仅供研究与学习目的，版权归 Anthropic 所有

关键文件索引

QueryEngine.ts + query.ts → Agent Loop

main.tsx → startup orchestration

services/tools/toolExecution.ts → Tool Runtime

tools/AgentTool/ + utils/swarm/ → Multi-Agent

utils/permissions/ → Permission System



关注公众号 · 持续更新

phodal