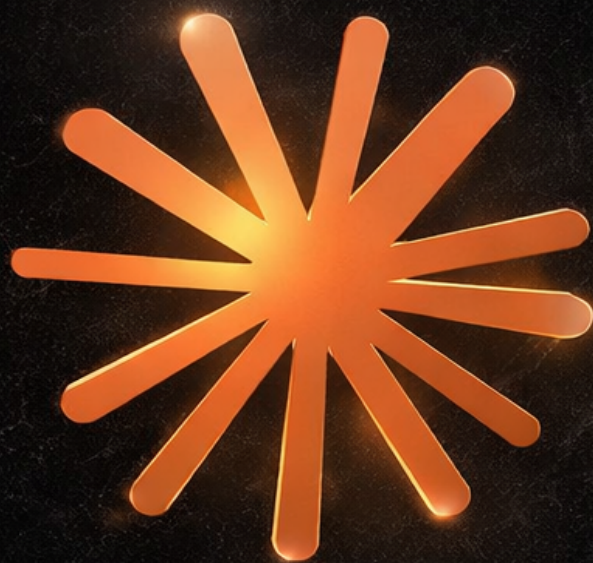


— 计算机科学经典著作 —

深入浅出 Claude Code



— SCUTBrothers —

深入浅出 Claude Code

从源码理解 *AI Agent* 编程系统

版本基线：Claude Code 2.1.88

2026 年 4 月 1 日

推荐语

以下推荐语为角色化样稿，不代表真实人物或机构背书。

“这本书真正难得的地方，不在于介绍 Claude Code 能做什么，而在于把它为什么能这样工作讲清楚了。它把会话、工具、权限、上下文和运行时连接成了一个完整系统。”

——角色化样稿，某大厂 AI 基础设施工程师

“很多人会用 Agent 工具，但很少有人从工程实现层面理解它的边界和代价。这本书适合已经在一线写代码、正在思考工具系统化问题的工程师。”

——角色化样稿，资深终端工具工程师

“如果你关心的不只是产品体验，而是一个 Agent 编程系统如何被构建、约束、扩展和治理，这本书提供了一条相对扎实的源码阅读路径。”

——角色化样稿，LLM Agent 平台负责人

“从 feature flag 到权限门禁，从上下文压缩到多代理协作，这本书把一个 22MB 的 CLI 工具拆解成了可理解的工程模块。对于想构建类似系统的团队，这是一份有价值的参考。”

——角色化样稿，编程工具产品工程师

序言

当我们谈论 AI 编程助手时，多数讨论停留在“它能做什么”的层面——能写代码、能调试、能重构。但很少有人追问一个更基本的问题：它是怎么做到的？

Claude Code 是 Anthropic 推出的命令行 AI 编程工具。它不是一个简单的 API 包装器，而是一个完整的 Agent 编程系统——包含工具调用、权限控制、上下文管理、多代理协作、插件扩展等多个子系统，总计超过 1900 个 TypeScript 源文件。

本书的目标是从源码层面解析这个系统。

我们关心的不是“Claude Code 有哪些功能”，而是“这些功能背后的工程实现是什么”。一个查询从用户输入到 Claude 响应，经过了哪些模块？工具调用的权限检查如何实现？上下文窗口快满时，压缩策略是什么？子代理如何派生和通信？这些问题的答案都在源码中。

本书基于 Claude Code 2.1.88 版本的源码进行分析。所有技术结论都区分三个层次：源码直接可证实的事实、基于源码结构的合理推断、以及需要进一步核实的问题。我们不会把推测写成定论，也不会用空泛的赞美替代具体的分析。

目标读者是有 TypeScript 或 Node.js 经验的工程师，尤其是正在构建或评估 AI 工具系统的从业者。你不需要事先了解 Claude Code 的使用方法——本书从构建系统讲起，逐步深入到每个子系统的设计与实现。

如果你关心的是一个 Agent 编程系统如何被构建、约束、扩展和治理，这本书提供了一条基于源码的阅读路径。

导读：如何阅读本书

全书结构

本书共 10 章，按照从底层到上层的顺序组织：

- 第 1 章讲构建系统，建立工程基线认知
- 第 2-3 章讲核心执行路径：入口、查询、工具
- 第 4 章讲终端界面的 React 化实现
- 第 5 章讲上下文工程——AI 应用中最关键的工程问题之一
- 第 6 章讲多任务与多代理运行时
- 第 7 章讲扩展系统的多层架构
- 第 8 章讲安全与权限模型
- 第 9 章讲性能与可靠性
- 第 10 章做跨章节的设计哲学总结

阅读路径建议

顺序阅读是最完整的路径，每章都为后续章节建立概念基础。
如果你对特定主题更感兴趣：

- 想理解 AI 工具调用机制：第 2 章 → 第 3 章
- 想理解上下文管理与压缩：第 5 章（需要第 2 章基础）
- 想理解安全模型：第 8 章（需要第 3 章基础）
- 想理解扩展架构：第 7 章（需要第 3 章基础）
- 想理解多代理协作：第 6 章（需要第 2、3 章基础）

术语约定

本书的核心术语在术语表（附录 A）中统一定义。正文中术语首次出现时会标注英文原名，后续使用统一叫法。

源码中的标识符（函数名、类型名、文件路径）保留英文原名，使用等宽字体排版。

方法论边界

本书的分析方法以静态源码阅读为主。所有技术结论都尽可能追溯到具体的文件、函数和类型定义。

对于静态阅读难以确定的机制（如运行时状态转换、异步调度顺序），我们会通过动态验证补充——但会明确标注哪些结论来自运行观测，哪些来自纯静态分析。

基于源码结构的合理推断会与确定性事实区分。我们不会把”可能是这样”写成”就是这样”。

版本基线

本书基于 Claude Code 2.1.88 版本。Claude Code 是一个快速迭代的产品，后续版本的实现细节可能发生变化。本书描述的是这个版本的工程快照，而非永恒不变的设计。

目录

第一章 构建系统与工程基线	1
1.1 项目概览：规模、依赖与技术栈	1
1.2 Bun 构建流程：从源码到单文件产物	2
1.3 特性开关：编译期产品形态控制	3
1.4 MACRO 常量：编译期值注入	4
1.5 依赖管理：私有包存根与第三方补丁	5
1.6 Native 模块体系：vendor 目录	5
第二章 入口、REPL 与 Query 生命周期	7
2.1 CLI 入口：多路快速分发	7
2.2 初始化：init() 的职责链	8
2.3 main() 函数：Commander 解析与 REPL 启动	9
2.4 QueryEngine：查询生命周期的核心	9
2.4.1 submitMessage 执行流	10
2.5 query()：单次查询的执行流	11
2.6 Task 系统：后台任务管理	11
第三章 Tool、Command 与 Hook：执行系统的主干	12
3.1 工具类型系统：Tool.ts 的设计	12
3.2 工具注册：tools.ts 的组装逻辑	12
3.2.1 BashTool：命令执行与安全检查	13
3.2.2 AgentTool：子代理派生	14
3.3 命令系统：斜杠命令的注册与分发	14
3.4 Hook 系统：生命周期事件拦截	15
3.5 执行流串联：从 API 响应到工具结果	15
第四章 终端界面架构：Ink、组件树与交互模型	18
4.1 自研 Ink 引擎：终端中的 React	18
4.2 组件层次：从 App 到 REPL	19
4.3 核心 UI 组件	20

4.4	键绑定系统	21
4.5	交互模型：事件流与状态管理	21
第五章	Context Engineering：上下文、记忆与压缩	23
5.1	上下文组装：从系统提示词到完整请求	23
5.2	记忆系统：MEMORY.md 与自动记忆	23
5.3	自动压缩：compact 机制	24
5.3.1	触发条件	24
5.3.2	压缩策略	24
5.3.3	压缩变体	25
5.4	会话记忆：SessionMemory	26
5.5	记忆提取：extractMemories	26
5.6	Token 估算与预算	26
第六章	Agent Runtime：任务、子代理、协作与远程执行	28
6.1	任务系统回顾：从类型到实例	28
6.2	LocalShellTask：后台命令执行	28
6.3	LocalAgentTask：子代理执行	29
6.4	Companion 系统：虚拟伙伴	30
6.5	团队协作：InProcessTeammate	30
6.6	远程执行：Remote 模块	31
6.7	任务生命周期管理	31
第七章	扩展系统：Commands、Skills、MCP、Plugins、Bridge	33
7.1	扩展层次概览	33
7.2	技能系统：可复用的提示词模板	33
7.2.1	技能加载	33
7.3	MCP 协议集成	34
7.3.1	客户端协议	35
7.3.2	连接管理	35
7.3.3	安全与权限	35
7.4	插件系统	36
7.5	Bridge 桥接层	36
7.6	扩展系统的设计权衡	37
第八章	权限、安全、策略与治理	38
8.1	权限模式：PermissionMode 的层次	38
8.2	权限规则：来源与优先级	39
8.3	工具级权限检查：CanUseToolFn	39

8.4	文件系统安全边界	40
8.5	策略限制: PolicyLimits	41
8.6	OAuth 认证	41
8.7	SSRF 防护	42
8.8	MoreRight: 权限提升	42
8.9	安全模型的设计哲学	42
第九章	性能、可靠性与产品化细节	43
9.1	启动性能优化	43
9.2	API 调用与流式响应	44
9.3	错误处理与重试策略	44
9.3.1	withRetry 重试机制	44
9.3.2	API 服务模块全景	45
9.4	费用追踪	45
9.5	Token 估算	46
9.6	分析与可观测性	46
9.7	诊断与调试	47
9.8	产品化细节	47
第十章	设计哲学与工程取舍	48
10.1	编译期 vs 运行时: 双层门控	48
10.2	单文件产物的利弊	48
10.3	React 终端 UI 的得失	49
10.4	上下文工程的实用主义	49
10.5	权限模型的演化方向	49
10.6	扩展性与安全性的张力	50
10.7	工程复杂度管理	50
10.8	类型系统作为工程约束工具	51
10.9	从源码看产品演化	51
10.10	系统架构总览	52
10.11	结语	53
附录 A	术语表	54
附录 B	源码目录索引	56

第一章 构建系统与工程基线

1.1 项目概览：规模、依赖与技术栈

Claude Code 是一个大型 TypeScript 项目。src/ 目录包含 1906 个核心源文件，覆盖终端 UI、工具执行、上下文管理、权限控制、插件系统等多个子系统。

项目的技术栈选型围绕三个核心决策展开。

第一，构建工具选择 Bun。package.json 中 "build": "bun run build.ts" 表明构建流程由 Bun 驱动。选择 Bun 而非 webpack 或 esbuild 的直接原因是源码大量使用了 bun:bundle 模块提供的 feature() API——这是 Bun bundler 的专有特性，用于编译期特性门控。

第二，模块体系采用 ESM。package.json 声明 "type": "module"，tsconfig.json 配置 "module": "ESNext" 和 "moduleResolution": "bundler"。构建产出也是 ESM 格式。

第三，终端 UI 基于 React。tsconfig.json 中 "jsx": "react-jsx" 和 "jsx-ImportSource": "react" 表明项目使用 React JSX 语法。具体的终端渲染引擎是 Ink（一个基于 React 的终端 UI 框架），这将在第 4 章详细讨论。

依赖方面，package.json 声明了大量依赖，其中几个值得关注：

- @anthropic-ai/sdk (0.80.0): Anthropic 官方 SDK，负责与 Claude API 通信
- @modelcontextprotocol/sdk (1.29.0): MCP 协议 SDK，支撑工具扩展体系
- @commander-js/extra-typings (14.0.0): CLI 参数解析
- 多个 @opentelemetry/* 包：可观测性基础设施
- 多个 @aws-sdk/* 和 @azure/* 包：多云 API 支持

构建入口点是 src/entrypoints/cli.tsx，最终打包为单文件 dist/cli.js，体积约 22MB。

1.2 Bun 构建流程：从源码到单文件产物

`build.ts` 是整个构建流程的核心，总共不到 200 行代码，但承载了四个关键职责：定义特性开关、注入编译期常量、配置构建插件、执行后处理。

构建调用的核心是 `Bun.build()`：

```
1 const result = await Bun.build({
2   entrypoints: ['./src/entrypoints/cli.tsx'],
3   outdir: './dist',
4   target: 'node',
5   format: 'esm',
6   sourcemap: 'linked',
7   minify: false,
8   define: { /* MACRO 常量 */ },
9   external: ['*.node', 'sharp', '@img/*'],
10  plugins: [ /* 两个构建插件 */ ],
11 })
```

几个配置值得注意。`target: 'node'` 表明产物运行在 Node.js 环境而非浏览器。`minify: false` 说明外部版本不做代码压缩。`external` 数组排除了 `.node` 原生模块和 `sharp` 图像库，这些在运行时动态加载。

构建完成后，脚本执行两步后处理：在产物头部添加 shebang (`#!/usr/bin/env node`)，然后设置 755 可执行权限。这使得 `dist/cli.js` 可以直接作为命令行工具运行。

图 1.1 展示了完整的构建流程。

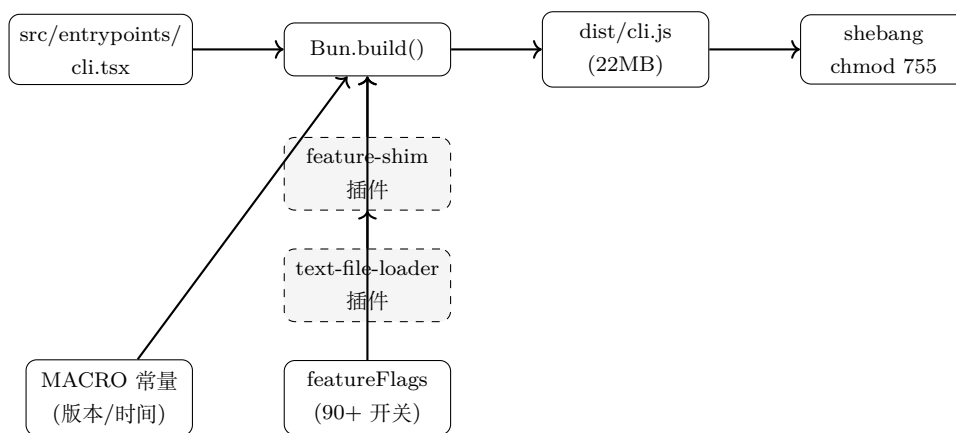


图 1.1: Claude Code 构建流程

构建流程中最关键的设计是两个自定义插件，它们分别处理特性开关和文本文件内联。

第一个插件 `bun-bundle-feature-shim` 拦截所有对 `bun:bundle` 的导入，将

`feature()` 函数替换为一个查表实现。表中的键值对来自 `build.ts` 顶部定义的 `featureFlags` 对象。由于表中的值都是布尔字面量，Bun 的 tree-shaking 会消除所有 `if (feature('SOME_FLAG'))` 中条件为 `false` 的分支，实现编译期死代码消除。

第二个插件 `text-file-loader` 将 `.md` 和 `.txt` 文件转换为 JavaScript 模块（导出文件内容字符串），并将 `.d.ts` 类型声明文件替换为空模块。这意味着源码中可以直接导入 markdown 文件作为字符串使用——系统提示词等文本资源就是通过这种方式内联到构建产物中的。

1.3 特性开关：编译期产品形态控制

`build.ts` 的前 100 行定义了一个 `featureFlags` 对象，包含 90 余个布尔开关。这些开关不是运行时配置，而是编译期常量——它们在构建时被替换为 `true` 或 `false` 字面量，由 bundler 执行死代码消除。

源码中使用特性开关的典型模式如下：

```
1 import { feature } from 'bun:bundle'
2
3 const coordinatorModule = feature('COORDINATOR_MODE')
4   ? require('./coordinator/coordinatorMode.js')
5   : null
```

当 `COORDINATOR_MODE` 为 `false` 时，整个 `require` 分支在构建时被消除，相关模块不会进入最终产物。

从 `build.ts` 中可以将这些开关分为三类。

外部版本启用的开关

- `BUILTIN_EXPLORE_PLAN_AGENTS`：内置的探索和计划代理
- `TOKEN_BUDGET`：Token 预算显示
- `MCP_SKILLS`：MCP 技能系统
- `COMPACTION_REMINDERS`：上下文压缩提醒

内部功能开关（外部版本关闭）

- `BRIDGE_MODE`：IDE 桥接模式
- `COORDINATOR_MODE`：多代理协调
- `KAIROS`：助手模式（含 `KAIROS_DREAM`、`KAIROS_CHANNELS` 等子开关）

- DAEMON: 后台守护进程
- BUDDY: 子代理系统

实验性功能开关

- VOICE_MODE: 语音输入
- WEB_BROWSER_TOOL: 浏览器工具
- ULTRAPLAN / ULTRATHINK: 增强规划与思考

这套机制的工程意义在于: 同一份源码库通过不同的开关组合, 可以产出面向不同场景的产品变体。外部用户拿到的 npm 包是一个特定开关组合的构建结果, 而 Anthropic 内部可能运行着启用了更多功能的版本。

cli.tsx 入口文件中可以直接观察到这种模式的实际效果。例如, `--daemon-worker` 参数处理被 `feature('DAEMON')` 门控, `remote-control` 子命令被 `feature('BRIDGE_MODE')` 门控。在外部构建中, 这些代码路径被完全消除, 用户甚至无法通过命令行参数触发它们。

1.4 MACRO 常量: 编译期值注入

除了布尔开关, build.ts 还通过 `define` 字段注入了一组编译期常量:

```
1 define: {
2   'MACRO.VERSION': JSON.stringify('2.1.88'),
3   'MACRO.BUILD_TIME': JSON.stringify(BUILD_TIME),
4   'MACRO.ISSUES_EXPLAINER': JSON.stringify('report ...'),
5   'MACRO.FEEDBACK_CHANNEL': JSON.stringify('https://...'),
6   'MACRO.PACKAGE_URL': JSON.stringify('https://...'),
7 }
```

这些 `MACRO.*` 标识符在源码中以全局变量形式使用, 构建时被替换为对应的字符串字面量。例如 cli.tsx 中的版本输出:

```
1 console.log(`${MACRO.VERSION} (Claude Code)`)
2 // -> "2.1.88 (Claude Code)"
```

MACRO 与 feature flag 的区别在于: feature flag 是布尔门控, 控制代码路径的存废; MACRO 是值替换, 将构建时信息注入运行时。两者都在编译期完成替换, 运行时不存在任何查询开销。

另外, `Bun.env.NODE_ENV` 被定义为 'production', 这会影响 React 和其他库的行为模式。

1.5 依赖管理：私有包存根与第三方补丁

Claude Code 的依赖中有若干不在公开 npm registry 中的内部包。`package.json` 通过 `file:./stubs/...` 语法将它们指向本地存根目录：

存根包	用途
<code>color-diff-napi</code>	语法高亮 native 模块，存根禁用高亮功能
<code>modifiers-napi</code>	macOS 按键修饰符检测，存根返回空
<code>@ant/claude-for-chrome-mcp</code>	Chrome 扩展 MCP 服务器
<code>@anthropic-ai/mcpb</code>	MCP bundle 处理器
<code>@anthropic-ai/sandbox-runtime</code>	沙盒运行时

表 1.1: 私有包存根清单

这些存根提供了最小的类型兼容接口，使得构建可以通过，但相关功能在外部版本中不可用。

第三方补丁方面，`pnpm-workspace.yaml` 声明了一个补丁：

```
1 patchedDependencies:
2   commander@14.0.3: patches/commander@14.0.3.patch
```

这个补丁修改了 `commander` 库的 `splitOptionFlags` 函数，将短选项的正则表达式从只允许单字符改为允许多字符。原因是 Claude Code 使用了 `-d2e` 作为调试标志的短选项，而 `commander v14` 的标准实现只支持单字符短选项如 `-v`。

这是一个典型的最小侵入式补丁：只改一行正则，不改变 `commander` 的其他行为，且通过 `pnpm` 的 `patch` 机制管理，升级 `commander` 时补丁会自动提示冲突。

1.6 Native 模块体系：vendor 目录

`vendor/` 目录包含 4 个 native 模块的 TypeScript 包装器：

模块	平台	用途
<code>audio-capture-src</code>	macOS, Linux, Windows	音频采集（录音与播放）
<code>image-processor-src</code>	跨平台	图像处理（sharp 兼容 API）
<code>modifiers-napi-src</code>	macOS	按键修饰符检测
<code>url-handler-src</code>	macOS	URL 事件处理（Apple Event）

表 1.2: vendor 目录 native 模块清单

这四个模块共享相同的加载策略：懒加载、平台检测、多路径回退。以 `audio-capture-src` 为例：

1. 首次调用时才尝试加载 native 模块 (`loadAttempted` 标志位)
2. 先检查平台是否支持 (`process.platform`)
3. 优先尝试 bundled 路径 (通过环境变量 `AUDIO_CAPTURE_NODE_PATH`)
4. 回退到 `npm-install` 路径和 `dev` 路径
5. 所有路径都失败则返回 `null`, 调用方通过返回值判断功能是否可用

这种设计的工程考量是: native 模块的可用性取决于平台和安装方式, 不能假设它一定存在。通过懒加载避免了启动时的 `dlopen` 开销, 通过多路径回退兼容了不同的部署场景 (开发、npm 安装、Bun 编译打包)。

`image-processor-src` 比较特殊——它提供了一个与 `sharp` 库 API 兼容的接口 (`metadata()`、`resize()`、`jpeg()`、`png()`、`toBuffer()` 等), 但底层使用自研的 native 模块而非 `sharp` 本身。`build.ts` 中 `external: ['sharp', '@img/*']` 将 `sharp` 排除在构建之外, 运行时由这个 `vendor` 模块替代。

第二章 入口、REPL 与 Query 生命周期

2.1 CLI 入口：多路快速分发

Claude Code 的执行从 `src/entrypoints/cli.tsx` 开始。这个文件的核心设计是一个多路分发器：根据命令行参数，将请求路由到不同的快速路径 (fast path)，只有在没有匹配到任何快速路径时，才加载完整的 CLI 主模块。

这种设计的工程目的是最小化启动开销。`cli.tsx` 中所有模块导入都是动态的 (`await import(...)`)，源码注释写道：“All imports are dynamic to minimize module evaluation for fast paths. Fast-path for `--version` has zero imports beyond this file.”

快速路径按优先级排列：

1. `--version` / `-v`：直接输出 `MACRO.VERSION`，零模块加载
2. `--dump-system-prompt`：输出系统提示词（内部功能，feature flag 门控）
3. `--claude-in-chrome-mcp`：启动 Chrome MCP 服务器
4. `--chrome-native-host`：启动 Chrome Native Host
5. `--daemon-worker`：启动守护进程 worker（feature flag 门控）
6. `remote-control` / `bridge`：启动 Bridge 模式（feature flag 门控）
7. `daemon`：启动守护进程主进程（feature flag 门控）
8. `ps` / `logs` / `attach` / `kill` / `--bg`：后台会话管理（feature flag 门控）
9. `new` / `list` / `reply`：模板任务命令（feature flag 门控）
10. `--worktree` `--tmux`：tmux worktree 快速路径

如果没有匹配到任何快速路径，`cli.tsx` 执行三步操作：启动早期输入捕获 (`startCapturingEarlyInput`)、动态导入 `main.tsx`、调用 `cliMain()`。

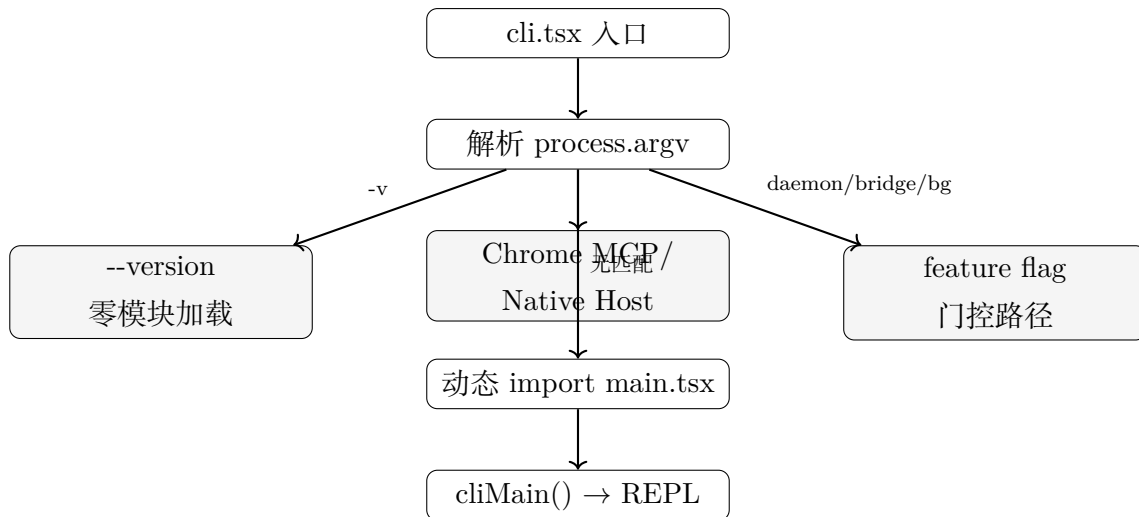


图 2.1: CLI 入口多路分发

图 2.1 展示了入口分发的整体结构。

多数快速路径被 feature flag 门控。在外部构建中，DAEMON、BRIDGE_MODE、BG_SESSIONS、TEMPLATES 等开关均为 false，这些代码路径在编译时被完全消除。外部用户实际可用的快速路径只有 --version、Chrome 相关路径和 worktree 路径。

2.2 初始化：init() 的职责链

当进入完整 CLI 路径后，main.tsx 被加载。在 REPL 启动之前，系统需要完成一系列初始化工作，这些工作集中在 src/entrypoints/init.ts 的 init() 函数中。

init() 使用 memoize 包装，确保只执行一次。它的职责链按顺序包括：

1. 启用配置系统 (enableConfigs())
2. 应用安全环境变量 (applySafeConfigEnvironmentVariables())
3. 配置 CA 证书——必须在首次 TLS 握手前完成
4. 设置优雅退出处理 (setupGracefulShutdown())
5. 异步初始化分析事件日志
6. 异步填充 OAuth 账户信息
7. 异步检测 JetBrains IDE 环境
8. 异步检测 Git 仓库
9. 初始化远程管理设置和策略限制的加载

10. 配置 mTLS 和全局 HTTP 代理

11. 预连接 Anthropic API (TCP+TLS 握手预热)

这个初始化序列的设计体现了两个原则：关键路径同步执行（CA 证书、代理配置必须在网络请求前完成），非关键路径异步并行（OAuth、IDE 检测、仓库检测用 `void` 触发后不等待）。

`main.tsx` 本身在模块顶层也执行了三个副作用操作，且必须在其他 `import` 之前运行：

1. `profileCheckpoint('main_tsx_entry')`：标记启动时间点
2. `startMdmRawRead()`：启动 MDM 子进程读取，与后续 `import` 并行
3. `startKeychainPrefetch()`：预取 macOS 钥匙串中的 OAuth 和 API key

源码注释解释了原因：钥匙串预取将两次串行的 macOS keychain 读取变为并行，节省约 65ms 的启动时间。这是一个典型的启动性能优化：把串行 I/O 变成并行。

2.3 main() 函数：Commander 解析与 REPL 启动

`main.tsx` 导出的 `main()` 函数是完整 CLI 的入口。它使用 `Commander.js` 定义命令行参数，处理认证、配置加载、工具注册等准备工作，最终启动 REPL 或执行单次查询。

`main.tsx` 的 `import` 列表揭示了系统的核心依赖关系：`Commander`（CLI 解析）、`React`（UI 渲染）、`chalk`（终端着色）、以及大量内部模块——从 `context.js`（上下文管理）到 `tools.js`（工具注册）到 `replLauncher.js`（REPL 启动）。

`Commander` 解析完成后，系统进入 REPL 循环或单次查询模式。REPL 通过 `launchRepl()` 启动，这是一个基于 `Ink` 的 `React` 应用。

2.4 QueryEngine：查询生命周期的核心

`QueryEngine` 是管理查询生命周期和会话状态的核心类。源码注释明确定义了它的职责：“`QueryEngine owns the query lifecycle and session state for a conversation.`”

`QueryEngineConfig` 类型定义了创建 `QueryEngine` 所需的完整配置：

`QueryEngine` 的内部状态包括可变消息历史（`mutableMessages`）、中断控制器（`abortController`）、权限拒绝记录（`permissionDenials`）、累计 token 使用量（`totalUsage`）和文件状态缓存（`readFileState`）。

设计上，一个 `QueryEngine` 实例对应一个会话。每次 `submitMessage()` 调用开始一个新的对话轮次，状态跨轮次持久化。

配置项	用途
<code>cwd</code>	工作目录
<code>tools</code>	可用工具集合
<code>commands</code>	斜杠命令列表
<code>mcpClients</code>	MCP 服务器连接
<code>agents</code>	代理定义
<code>canUseTool</code>	权限检查函数
<code>getAppState / setAppState</code>	全局状态访问
<code>readFileCache</code>	文件状态缓存
<code>maxTurns</code>	最大轮次限制
<code>maxBudgetUsd</code>	预算限制
<code>thinkingConfig</code>	思考模式配置

表 2.1: QueryEngineConfig 核心配置项

2.4.1 submitMessage 执行流

`submitMessage()` 是一个 `AsyncGenerator`，通过 `yield` 逐步返回 SDK 消息事件。它的执行流程包括：

1. 解构配置——从 `QueryEngineConfig` 中提取工具、命令、MCP 客户端等
2. 包装权限检查——将 `canUseTool` 包装为 `wrappedCanUseTool`，在原始检查基础上追踪权限拒绝记录 (`permissionDenials`)
3. 确定模型——优先使用用户指定模型 (`userSpecifiedModel`)，否则使用默认模型
4. 配置思考模式——默认启用 `adaptive thinking`，除非显式禁用
5. 组装系统提示词——通过 `fetchSystemPromptParts()` 获取默认提示词、用户上下文和系统上下文，然后按优先级合并：自定义提示词 > 默认提示词，再追加记忆机制提示词和附加提示词
6. 处理用户输入——通过 `processUserInput()` 解析斜杠命令、处理附件
7. 调用 `query()` 执行实际的 API 交互循环

系统提示词的组装逻辑值得注意：当 SDK 调用方提供了 `customSystemPrompt` 时，它完全替换默认提示词（而非追加）。但如果同时设置了 `CLAUDE_COWORK_MEMORY_PATH_OVERRIDE` 环境变量，记忆机制提示词仍会被注入——这确保了自定义提示词的调用方仍然可以使用记忆系统。

2.5 query(): 单次查询的执行流

src/query.ts 中的 query() 函数实现了单次查询的完整执行流。它的核心依赖包括工具查找与执行 (Tool.ts)、自动压缩 (services/compact/)、API 调用与错误处理 (services/api/)、消息创建与规范化 (utils/messages.ts)、以及记忆附件处理 (utils/attachments.ts)。

query.ts 开头的条件导入展示了 feature flag 在运行时的实际效果:

```
1 const reactiveCompact = feature('REACTIVE_COMPACT')
2   ? require('./services/compact/reactiveCompact.js')
3   : null
4 const contextCollapse = feature('CONTEXT_COLLAPSE')
5   ? require('./services/contextCollapse/index.js')
6   : null
```

这些模块只在对应 feature flag 启用时才被加载，否则为 null。调用方在使用前需要检查是否为空，这种模式贯穿整个代码库。

2.6 Task 系统：后台任务管理

src/Task.ts 定义了 Claude Code 的任务类型系统。任务是后台运行的工作单元，支持多种类型：

```
1 type TaskType =
2   | 'local_bash'           // 本地 shell 命令
3   | 'local_agent'         // 本地子代理
4   | 'remote_agent'        // 远程代理
5   | 'in_process_teammate' // 进程内团队成员
6   | 'local_workflow'      // 本地工作流
7   | 'monitor_mcp'         // MCP 监控
8   | 'dream'               // 梦境模式
```

任务状态机定义了五个状态：pending、running、completed、failed、killed。isTerminalTaskStatus() 函数判断任务是否处于终态 (completed/failed/killed)，用于防止向已结束的任务注入消息、清理孤儿任务等场景。

TaskContext 提供了任务执行所需的环境：abortController (中断控制)、getAppState (状态读取)、setAppState (状态写入)。这个设计使得每个任务都能访问和修改全局应用状态，同时通过 AbortController 支持外部取消。

任务 ID 使用类型前缀区分：b 代表 local_bash，其他类型有各自的前缀。这种命名约定使得从 ID 就能判断任务类型，便于日志分析和调试。

第三章 Tool、Command 与 Hook： 执行系统的主干

3.1 工具类型系统：Tool.ts 的设计

Claude Code 的工具系统定义在 `src/Tool.ts` 中。这个文件不包含具体工具的实现，而是定义了工具的类型系统——所有工具必须遵循的接口契约。

`ToolUseContext` 是工具执行时的环境对象，包含工具运行所需的一切：可用命令列表、调试标志、模型名称、MCP 客户端连接、中断控制器、文件状态缓存、全局应用状态的读写接口。它是工具与系统其余部分之间的桥梁。

`ToolPermissionContext` 定义了权限检查的完整上下文，采用 `DeepImmutable` 包装确保不可变性：

```
1 type ToolPermissionContext = DeepImmutable<{
2   mode: PermissionMode
3   alwaysAllowRules: ToolPermissionRulesBySource
4   alwaysDenyRules: ToolPermissionRulesBySource
5   alwaysAskRules: ToolPermissionRulesBySource
6   isBypassPermissionsModeAvailable: boolean
7   shouldAvoidPermissionPrompts?: boolean
8 }>
```

权限模式 (`PermissionMode`) 决定了工具执行前的检查严格程度。`alwaysAllowRules`、`alwaysDenyRules`、`alwaysAskRules` 三组规则按来源分组，支持用户在不同层级（全局、项目、会话）配置权限策略。这个设计将在第 8 章详细讨论。

3.2 工具注册：tools.ts 的组装逻辑

`src/tools.ts` 负责将所有工具实例组装为一个可用工具集合。它的 `import` 列表直接反映了 Claude Code 提供的内置工具：

工具注册同样使用 `feature flag` 进行条件加载。例如 `SleepTool` 只在 `PROACTIVE` 或 `KAIROS` 启用时加载，`CronCreateTool` 等定时任务工具只在 `AGENT_TRIGGERS` 启

工具	用途
BashTool	执行 shell 命令
FileReadTool	读取文件内容
FileEditTool	编辑文件（精确替换）
FileWriteTool	创建或覆写文件
GlobTool	文件模式匹配搜索
GrepTool	文件内容正则搜索
AgentTool	派生子代理执行复杂任务
SkillTool	调用技能模板
WebFetchTool	获取网页内容
WebSearchTool	网页搜索
NotebookEditTool	编辑 Jupyter notebook
LSPTTool	语言服务器协议交互
TodoWriteTool	任务列表管理
TaskOutputTool	读取后台任务输出
TaskStopTool	停止后台任务

表 3.1: Claude Code 核心内置工具

用时加载。部分工具还通过 `process.env.USER_TYPE === 'ant'` 限制为内部用户专用。

每个工具目录（如 `tools/BashTool/`）通常包含工具实现、输入 schema 定义和相关的辅助函数。工具通过 `findToolByName()` 函数按名称查找，这个函数在 `query()` 处理 Claude API 返回的 `tool_use` block 时被调用。

3.2.1 BashTool: 命令执行与安全检查

`tools/BashTool/` 是最复杂的工具实现之一，包含 18 个文件。除了核心的 `BashTool.tsx`，它还包含一套完整的安全检查体系：

- `bashSecurity.ts`: 检测危险的 shell 模式——命令替换 (`$()`、`${}`)、进程替换 (`<()`、`>()`)、Zsh 特有的扩展语法 (`=cmd`、`glob` 限定符)、甚至 PowerShell 注释语法（防御性深度）
- `bashPermissions.ts`: 工具级权限检查
- `destructiveCommandWarning.ts`: 检测破坏性命令（如 `rm -rf`）并发出警告
- `pathValidation.ts`: 验证命令中的文件路径是否在允许范围内

- `readOnlyValidation.ts`：在只读模式下验证命令是否安全
- `sedValidation.ts` / `sedEditParser.ts`：专门针对 `sed` 命令的编辑操作解析与验证
- `commandSemantics.ts`：命令语义分析
- `shouldUseSandbox.ts`：判断是否应在沙盒中执行

`bashSecurity.ts` 中的安全模式检测覆盖了多种 shell 注入向量。例如，Zsh 的 `=cmd` 语法会将 `=curl` 展开为 `/usr/bin/curl`，绕过基于命令名的 `deny` 规则。源码注释详细解释了每种模式的攻击场景。

3.2.2 AgentTool：子代理派生

`tools/AgentTool/` 包含 15 个文件，实现了子代理的完整生命周期。`runAgent.ts` 是核心——它直接调用 `query()` 函数创建一个独立的查询循环，子代理拥有自己的系统提示词、消息历史和工具集合。

`loadAgentsDir.ts` 从文件系统加载代理定义（`.claude/agents/` 目录），`builtinAgents.ts` 注册内置代理（如 Explore、Plan 代理）。`agentMemory.ts` 管理子代理的记忆隔离，`forkSubagent.ts` 处理子代理的派生逻辑。

`prompt.ts` 中的 `DEFAULT_AGENT_PROMPT` 定义了子代理的默认系统提示词，`enhanceSystemPromptWithEnvDetails()` 将环境信息（工作目录、git 状态等）注入到子代理的提示词中。

3.3 命令系统：斜杠命令的注册与分发

`src/commands.ts` 是斜杠命令的注册中心。它导入了 40 余个命令模块，每个模块对应一个 `/command` 命令。

从 `import` 列表可以看到命令的完整清单，涵盖多个类别：

- 会话管理：`/clear`、`/compact`、`/resume`、`/session`
- 代码操作：`/commit`、`/diff`、`/review`
- 配置与状态：`/config`、`/cost`、`/status`、`/usage`
- 扩展系统：`/mcp`、`/skills`、`/tasks`
- 用户界面：`/theme`、`/vim`、`/keybindings`
- 认证与账户：`/login`、`/logout`

- 辅助功能：/help、/doctor、/memory、/context

命令同样使用 feature flag 进行条件注册。/proactive 命令需要 PROACTIVE 或 KAIROS 开关，/bridge 命令需要 BRIDGE_MODE 开关，/voice 命令需要 VOICE_MODE 开关。在外部构建中，这些命令不存在。

命令与工具的区别在于：工具由 Claude 模型在响应中主动调用（通过 tool_use block），命令由用户通过 / 前缀手动触发。两者共享相同的执行上下文（ToolUseContext），但触发路径不同。

3.4 Hook 系统：生命周期事件拦截

src/hooks/ 目录包含 80 余个 React hook，构成了 Claude Code 的生命周期事件系统。这些 hook 不是传统意义上的“钩子函数”，而是 React 自定义 hook——它们在 Ink 组件树中被调用，管理各种副作用和状态。

从目录结构可以看到几类关键 hook：

权限与工具控制 useCanUseTool.tsx 是工具权限检查的核心 hook。它实现了 CanUseToolFn 类型，在每次工具调用前判断是否允许执行。这个 hook 综合考虑权限模式、规则配置、用户交互历史等因素做出决策。

用户交互 useExitOnCtrlCD.ts 处理 Ctrl+C/Ctrl+D 退出逻辑，useArrowKeyHistory.tsx 管理方向键历史导航，useCopyOnSelect.ts 实现选中复制功能，useCommandKeybindings.tsx 处理命令快捷键绑定。

会话与状态 useCommandQueue.ts 管理命令队列，useDynamicConfig.ts 处理动态配置变更，useAssistantHistory.ts 管理助手对话历史。

用户自定义 Hook 除了内置的 React hook，Claude Code 还支持用户通过 settings.json 配置的 shell hook。这些 hook 在特定事件（如 PreToolUse、PostToolUse、Notification）触发时执行用户指定的 shell 命令。用户自定义 hook 的执行结果可以影响工具调用的行为——例如 PreToolUse hook 可以阻止某个工具的执行。

这两层 hook 机制的区别在于：React hook 是系统内部的状态管理机制，用户自定义 hook 是面向用户的扩展点。前者在组件渲染周期中运行，后者通过子进程执行 shell 命令。

3.5 执行流串联：从 API 响应到工具结果

将上述三个子系统串联起来，图 3.1 展示了一次完整的工具调用流程。

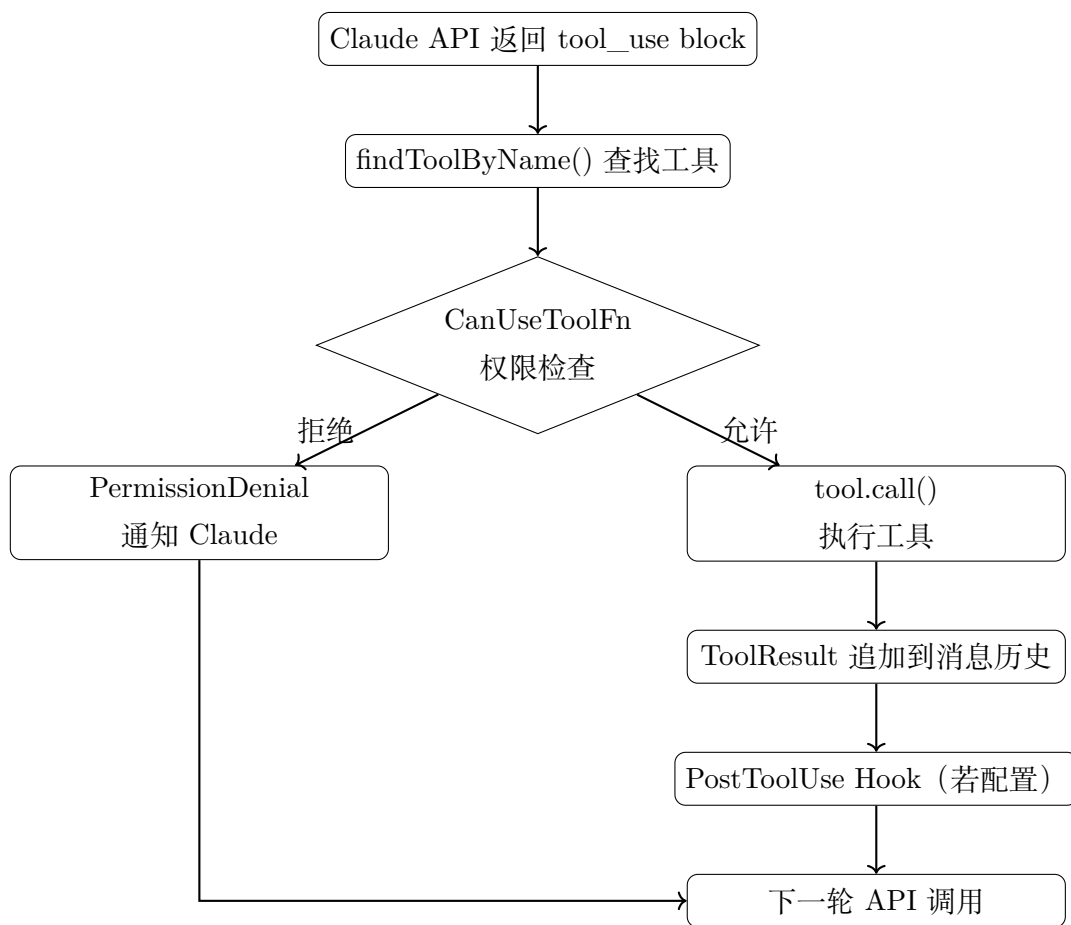


图 3.1: 工具调用执行流程

具体步骤如下：

1. Claude API 返回包含 `tool_use` block 的响应
2. `query()` 解析 `tool_use` block, 通过 `findToolByName()` 查找对应工具
3. `CanUseToolFn` (来自 `useCanUseTool` hook) 执行权限检查
4. 若权限通过, 调用工具的执行函数, 传入 `ToolUseContext`
5. 工具执行完成, 返回 `ToolResult`
6. 结果被追加到消息历史, 作为下一轮 API 调用的上下文
7. 若用户配置了 `PostToolUse` hook, 执行对应的 shell 命令

如果用户在步骤 3 拒绝了权限请求, 系统生成一个 `PermissionDenial`, Claude 会收到拒绝信息并调整后续行为。

这个流程的关键设计决策是：权限检查发生在工具执行之前, 而不是之后。这意味着 Claude 提出的工具调用请求可以被用户拦截, 系统不会执行任何未经授权的操作。这是 Claude Code 安全模型的基础, 将在第 8 章深入分析。

第四章 终端界面架构：Ink、组件树与交互模型

4.1 自研 Ink 引擎：终端中的 React

Claude Code 的终端界面不是传统的字符串拼接输出，而是一个完整的 React 应用。`src/ink/` 目录包含约 50 个文件，实现了一个自研的终端渲染引擎——基于 Ink 框架的深度定制版本。

`ink.tsx` 是渲染引擎的核心入口。从其 `import` 列表可以看到引擎的技术栈：React reconciler (`react-reconciler`)、Yoga 布局引擎（用于 flexbox 布局计算）、以及大量终端控制序列处理模块。

引擎的工作原理是：React 组件树通过自定义 reconciler 转换为终端 DOM 节点，Yoga 计算布局位置，然后渲染为终端输出。每一帧的渲染流程包括：

1. React reconciler 处理组件更新，生成 DOM 变更
2. Yoga 布局引擎计算每个节点的位置和尺寸
3. `render-node-to-output` 将节点树渲染为字符网格
4. `render-to-screen` 将字符网格转换为终端控制序列
5. 增量输出到终端 (`writeDiffToTerminal`)

引擎支持多种高级终端特性：`alt-screen` 模式、鼠标追踪(`ENABLE_MOUSE_TRACKING`)、Kitty 键盘协议 (`ENABLE_KITTY_KEYBOARD`)、文本选择与复制、超链接、搜索高亮。这些特性通过终端控制序列 (CSI、DEC、OSC) 实现。

`ink/` 目录的内部结构反映了渲染管线的各个阶段：

`screen.ts` 中的对象池设计值得注意：`CharPool`、`StylePool`、`HyperlinkPool` 通过对象复用减少 GC 压力。`migrateScreenPools()` 在屏幕尺寸变化时迁移池状态，避免重新分配。

增量渲染是性能的关键——`writeDiffToTerminal()` 只输出两帧之间变化的部分，而不是每帧重绘整个屏幕。`optimizer.ts` 进一步减少输出中冗余的终端控制序列。

模块	职责
<code>reconciler.ts</code>	自定义 React reconciler，桥接 React 与终端 DOM
<code>dom.ts</code>	终端 DOM 节点定义与操作
<code>layout/</code>	Yoga 布局引擎集成 (flexbox 计算)
<code>render-node-to-output.ts</code>	将 DOM 树渲染为字符网格
<code>render-to-screen.ts</code>	将字符网格转换为终端输出
<code>screen.ts</code>	屏幕缓冲区管理 (CellWidth、CharPool、StylePool)
<code>output.ts</code>	输出管理
<code>optimizer.ts</code>	输出优化——减少不必要的终端控制序列
<code>selection.ts</code>	文本选择系统 (选中、复制、URL 检测)
<code>searchHighlight.ts</code>	搜索高亮渲染
<code>terminal.ts</code>	终端能力检测与差量写入
<code>parse-keypress.ts</code>	按键事件解析
<code>focus.ts</code>	焦点管理 (FocusManager)
<code>hit-test.ts</code>	鼠标点击与悬停的命中测试

表 4.1: Ink 渲染引擎模块结构

帧率控制通过 `FRAME_INTERVAL_MS` 常量和 `throttle` 实现，避免过于频繁的终端刷新。

4.2 组件层次：从 App 到 REPL

组件树的顶层是 `src/components/App.tsx`，它是所有交互式会话的根包装器。App 组件提供三层 Context Provider：

1. `FpsMetricsProvider`：帧率性能指标
2. `StatsProvider`：统计数据上下文
3. `AppStateProvider`：全局应用状态（消息、任务、配置等）

App 组件使用了 React Compiler 的优化输出 (`react/compiler-runtime`)，通过缓存数组 `$` 实现细粒度的 memoization，避免不必要的子树重渲染。

图 4.1 展示了组件树的核心层次。

App 之下是屏幕组件。`src/screens/` 目录包含三个屏幕：

- `REPL.tsx`：主交互界面，是最核心的屏幕组件
- `Doctor.tsx`：诊断界面

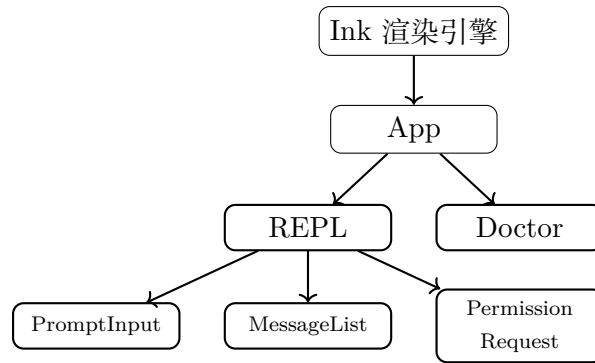


图 4.1: 终端 UI 组件树核心层次

- `ResumeConversation.tsx`: 会话恢复界面

`REPL.tsx` 是整个终端界面中最复杂的组件，从其 60 余行的 `import` 列表可以看出它整合了几乎所有子系统：消息渲染、权限请求对话框、提示输入、命令队列、后台任务导航、搜索高亮、会话管理、团队协作等。

4.3 核心 UI 组件

`src/components/` 目录包含 140 余个组件文件，构成了 Claude Code 的完整 UI 组件库。按功能可以分为几类：

输入与交互 `PromptInput/` 是用户输入组件，`BaseTextInput.tsx` 提供基础文本输入能力。`CustomSelect/` 实现自定义选择菜单。

权限与对话框 `permissions/` 子目录包含权限请求相关组件，如 `PermissionRequest.tsx`（权限确认对话框）和 `WorkerPendingPermission.tsx`（worker 等待权限状态）。`AutoModeOptInDialog.tsx`、`BypassPermissionsModeDialog.tsx` 等处理不同权限模式的切换。

消息与内容展示 `CompactSummary.tsx` 展示压缩摘要，`diff/` 子目录处理代码差异展示，`ContextVisualization.tsx` 可视化上下文状态。

状态与反馈 `AgentProgressLine.tsx` 展示代理执行进度，`CostThresholdDialog.tsx` 处理费用阈值提醒，`DiagnosticsDisplay.tsx` 展示诊断信息。

设计系统 `design-system/` 子目录包含基础 UI 原语，为上层组件提供统一的视觉风格。

4.4 键绑定系统

`src/keybindings/` 目录实现了一套完整的键绑定系统，包含 14 个文件。

`defaultBindings.ts` 定义默认快捷键映射，`loadUserBindings.ts` 加载用户自定义绑定（从 `~/.claude/keybindings.json`），`resolver.ts` 负责将按键事件解析为对应的操作。

键绑定系统支持：

- 单键快捷键和组合键（chord）
- 用户自定义覆盖默认绑定
- 保留快捷键保护（`reservedShortcuts.ts`）
- 上下文感知的快捷键激活（不同模式下同一按键触发不同操作）
- 快捷键提示显示（`useShortcutDisplay.ts`）

`useKeybinding.ts` 是组件中使用键绑定的 hook，它将键盘事件与注册的绑定进行匹配，触发对应的回调。`schema.ts` 和 `validate.ts` 确保用户自定义绑定的格式正确。

4.5 交互模型：事件流与状态管理

Claude Code 的交互模型建立在 React 的单向数据流之上。`AppState` 是全局状态对象，通过 `AppStateProvider` 注入组件树。状态更新通过 `setAppState` 函数式更新，触发组件重渲染。

用户输入事件的处理流程：

1. 终端原始按键事件被 Ink 引擎捕获
2. `parse-keypress.ts` 解析为结构化的 `ParsedKey`
3. 键绑定系统匹配快捷键
4. 若匹配到命令，执行对应操作
5. 若为文本输入，传递给 `PromptInput` 组件
6. 用户提交输入后，进入 `QueryEngine` 的查询流程

Ink 引擎还支持鼠标事件（点击、悬停）和终端焦点事件（`useTerminalFocus`），使得终端界面具备了接近 GUI 的交互能力。

这套架构的工程权衡在于：React 的声明式编程模型大幅降低了复杂 UI 的维护成本，但引入了 reconciler、Yoga 布局等运行时开销。对于一个 CLI 工具来说，这是一个不寻常的选择——它意味着 Claude Code 的终端界面本质上是一个运行在终端中的 React 应用。

第五章 Context Engineering: 上下文、记忆与压缩

5.1 上下文组装：从系统提示词到完整请求

Claude Code 发送给 API 的每次请求都包含三个层次的上下文：系统提示词、记忆附件和消息历史。`src/context.ts` 负责组装前两个层次。

`getSystemContext()` 和 `getUserContext()` 是两个 memoize 缓存的函数，分别收集系统级和用户级上下文信息。系统上下文包括 git 状态（分支、最近提交、工作区变更），用户上下文包括 CLAUDE.md 文件内容和记忆文件。

git 状态的获取展示了并行化的设计思路：`getGitStatus()` 通过 `Promise.all` 同时执行五个 git 命令（获取分支名、默认分支、status、log、用户名），而不是串行执行。git status 输出被截断到 `MAX_STATUS_CHARS`（2000 字符），避免大型仓库的状态信息占用过多上下文窗口。

上下文组装还支持注入机制：`systemPromptInjection` 变量允许在运行时向系统提示词注入额外内容，修改注入值时会立即清除 memoize 缓存。这是一个内部调试功能。

5.2 记忆系统：MEMORY.md 与自动记忆

`src/memdir/` 目录实现了 Claude Code 的记忆系统。记忆是跨会话持久化的结构化知识，存储在用户的 `~/.claude/` 目录下。

记忆系统的入口点是 `MEMORY.md` 文件。`memdir.ts` 定义了关键常量：

- `ENTRYPOINT_NAME`: 固定为 `'MEMORY.md'`
- `MAX_ENTRYPOINT_LINES`: 200 行上限
- `MAX_ENTRYPOINT_BYTES`: 25,000 字节上限

`truncateEntrypointContent()` 函数实现了双重截断策略：先按行数截断（自然边界），再按字节数截断（在最后一个换行符处切断，避免截断到行中间）。截断时会附加警告信息，告知用户哪个限制被触发。

记忆类型在 `memoryTypes.ts` 中定义, 包括用户记忆 (user)、反馈记忆 (feedback)、项目记忆 (project) 和引用记忆 (reference)。每种类型有不同的保存时机和使用场景。

`findRelevantMemories.ts` 实现了记忆检索——根据当前对话上下文, 从已存储的记忆中找出相关条目, 作为附件注入到 API 请求中。

记忆系统还支持团队记忆 (`teamMemPaths.ts`、`teamMemPrompts.ts`), 通过 TEAM-MEM feature flag 门控。团队记忆允许多个代理共享记忆状态, 这是多代理协作的基础设施之一。

5.3 自动压缩: compact 机制

当对话历史接近上下文窗口限制时, Claude Code 会自动触发压缩 (compact)。`src/services/compact/` 目录包含 13 个文件, 实现了完整的压缩子系统。

5.3.1 触发条件

`autoCompact.ts` 定义了自动压缩的触发逻辑。核心函数 `getEffectiveContextWindowSize()` 计算有效上下文窗口大小:

```
1 function getEffectiveContextWindowSize(model: string): number {
2   const reservedTokensForSummary = Math.min(
3     getMaxOutputTokensForModel(model),
4     MAX_OUTPUT_TOKENS_FOR_SUMMARY, // 20,000
5   )
6   return contextWindow - reservedTokensForSummary
7 }
```

有效窗口 = 模型上下文窗口 - 预留的摘要输出空间 (最多 20,000 token)。这个预留值基于 p99.99 的压缩摘要输出为 17,387 token 的统计数据。

`AutoCompactTrackingState` 跟踪压缩状态, 包括是否已压缩、轮次计数器、连续失败次数。连续失败次数作为熔断器, 当上下文不可恢复地超限时停止重试。

环境变量 `CLAUDE_CODE_AUTO_COMPACT_WINDOW` 允许用户自定义压缩触发的窗口大小。

5.3.2 压缩策略

`compact.ts` 实现了核心压缩逻辑。压缩的基本策略是: 将历史消息发送给 Claude API, 请求生成一个摘要, 然后用摘要替换原始消息。

`prompt.ts` 中的压缩提示词设计揭示了多个工程细节。首先, 提示词以一段强制性的“禁止工具调用”前言开头:

```
1 CRITICAL: Respond with TEXT ONLY. Do NOT call any tools.
2 Tool calls will be REJECTED and will waste your only turn.
```

这段前言的存在有具体的工程原因：压缩请求使用 `maxTurns: 1`，如果模型尝试调用工具，工具调用会被拒绝，导致没有文本输出。源码注释记录了这个问题在 Sonnet 4.6 上的发生率（2.79%）远高于 4.5（0.01%），因此需要更强的前言约束。

压缩输出采用两段式结构：`<analysis>` 块用于模型的思考草稿（按时间顺序分析每条消息的意图、方法、关键决策），`<summary>` 块是最终摘要。`<analysis>` 块在摘要进入上下文前会被 `formatCompactSummary()` 剥离——它只是一个提升摘要质量的中间步骤，不占用后续的上下文空间。

压缩执行流程还包含 hook 集成：`executePreCompactHooks()` 在压缩前执行，`executePostCompactHooks()` 在压缩后执行。用户可以通过 hook 在压缩前后执行自定义逻辑。

`compact.ts` 还处理了压缩后的上下文重建：通过 `generateFileAttachment()`、`getAgentListingDeltaAttachment()`、`getMcpInstructionsDeltaAttachment()` 等函数，将压缩后丢失的关键附件（文件状态、代理列表、MCP 指令）重新注入到压缩后的上下文中。

压缩后的清理工作由 `postCompactCleanup.ts` 处理，`sessionMemoryCompact.ts` 处理会话记忆的压缩。

5.3.3 压缩变体

压缩系统提供了多种变体以适应不同场景：

模块	用途
<code>compact.ts</code>	标准压缩——完整的对话摘要
<code>microCompact.ts</code>	微压缩——更轻量的压缩策略
<code>apiMicrocompact.ts</code>	API 级微压缩
<code>cachedMicrocompact.ts</code>	带缓存的微压缩（feature flag 门控）
<code>snipCompact.ts</code>	片段压缩——按边界截断（feature flag 门控）
<code>sessionMemoryCompact.ts</code>	会话记忆专用压缩

表 5.1: 压缩策略变体

`prompt.ts` 包含压缩时使用的提示词模板，指导 Claude 如何生成有效的对话摘要。`grouping.ts` 处理消息分组逻辑，决定哪些消息应该一起被压缩。

5.4 会话记忆：SessionMemory

`src/services/SessionMemory/` 实现了会话级别的记忆管理。与 `memdir/` 中的持久化记忆不同，`SessionMemory` 管理的是单次会话内的临时状态。

`sessionMemoryUtils.ts` 提供了会话记忆的工具函数，包括 `setLastSummarizedMessageId()` 等，用于跟踪压缩边界——记录最后一条被摘要的消息 ID，确保后续压缩不会重复处理已摘要的内容。

5.5 记忆提取：extractMemories

`src/services/extractMemories/` 实现了自动记忆提取功能。当对话中出现值得记住的信息时，系统可以自动将其提取为持久化记忆。

`extractMemories.ts` 是提取逻辑的核心，`prompts.ts` 包含指导 Claude 识别和提取记忆的提示词。这个功能通过 `EXTRACT_MEMORIES` feature flag 控制，在外部版本中默认关闭。

5.6 Token 估算与预算

上下文管理的基础是准确的 token 计数。`autoCompact.ts` 中引用的 `tokenCountWithEstimation()` 函数（来自 `utils/tokens.ts`）提供 token 估算能力。

`getContextWindowForModel()` 根据模型名称返回对应的上下文窗口大小。不同模型有不同的窗口限制，压缩策略需要据此调整触发阈值。

`QueryEngineConfig` 中的 `maxBudgetUsd` 和 `taskBudget` 字段提供了费用级别的预算控制。当累计费用接近预算时，系统会提醒用户或自动停止。

图 5.1 展示了上下文管理的完整生命周期。

整个上下文工程子系统的设计目标是：在有限的上下文窗口内，最大化信息密度。记忆系统提供跨会话的知识持久化，压缩系统在窗口接近满载时回收空间，token 估算确保系统始终知道还剩多少空间可用。这三者协同工作，使得 Claude Code 能够处理远超单次上下文窗口容量的长对话。

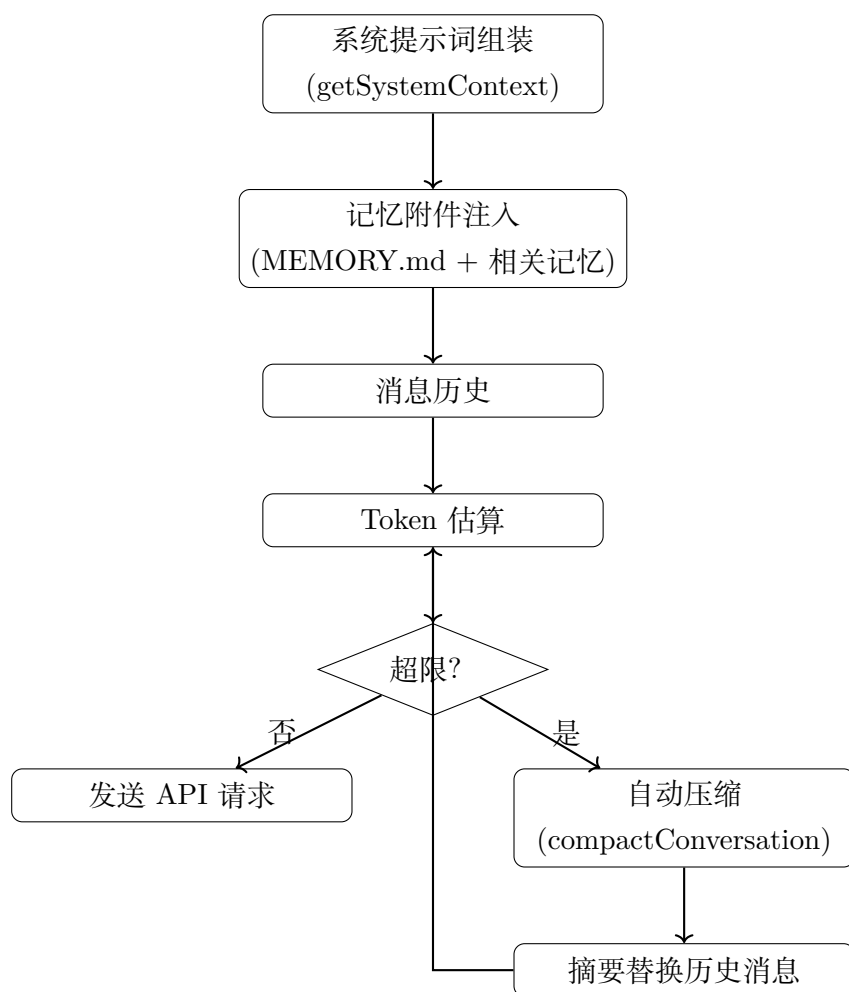


图 5.1: 上下文管理生命周期

第六章 Agent Runtime：任务、子代理、协作与远程执行

6.1 任务系统回顾：从类型到实例

第 2 章介绍了 `Task.ts` 中的任务类型定义。本章深入到任务的实际实现。`src/-tasks/` 目录包含多个任务类型的具体实现，每种类型对应一个子目录：

目录	用途
<code>LocalShellTask/</code>	本地 shell 命令的后台执行
<code>LocalAgentTask/</code>	本地子代理——在独立上下文中运行的 Claude 实例
<code>InProcessTeammateTask/</code>	进程内团队成员——与主代理共享进程的协作代理
<code>RemoteAgentTask/</code>	远程代理——通过网络连接的 Claude 实例
<code>DreamTask/</code>	梦境任务——后台异步处理（内部功能）
<code>LocalMainSessionTask.ts</code>	本地主会话任务管理

表 6.1: 任务类型实现目录

`stopTask.ts` 提供统一的任务停止接口，`pillLabel.ts` 处理任务在 UI 中的标签显示，`types.ts` 定义任务间共享的类型。

6.2 LocalShellTask：后台命令执行

`LocalShellTask/` 实现了后台 shell 命令的执行与监控。它是最基础的任务类型，由 `BashTool` 在需要后台执行时创建。

`LocalShellTask.tsx` 的实现包含一个值得注意的机制：stall 检测。当后台命令的输出停止增长超过 45 秒（`STALL_THRESHOLD_MS`），且输出末尾看起来像交互式提示（如 `(y/n)`、`Press Enter`、`Continue?`），系统会发送通知提醒用户命令可能在等待输入。

```
1 const STALL_CHECK_INTERVAL_MS = 5_000;
2 const STALL_THRESHOLD_MS = 45_000;
```

```
3 const STALL_TAIL_BYTES = 1024;
```

looksLikePrompt() 函数通过正则表达式匹配输出末尾的模式来判断命令是否在等待输入。这个设计区分了“命令执行慢”（如长时间编译）和“命令在等待交互”两种情况，只对后者发出通知。

guards.ts 定义了 LocalShellTaskState 类型和类型守卫函数，killShellTasks.ts 实现了 shell 任务的终止逻辑——需要正确处理子进程树的清理。

6.3 LocalAgentTask: 子代理执行

LocalAgentTask/ 实现了本地子代理——由 AgentTool 派生的独立 Claude 实例。每个子代理有自己的上下文、消息历史和工具集合，但共享主代理的权限配置。

LocalAgentTask.tsx 中定义了完整的进度追踪系统：

```
1 type AgentProgress = {
2   toolUseCount: number;
3   tokenCount: number;
4   lastActivity?: ToolActivity;
5   recentActivities?: ToolActivity[];
6   summary?: string;
7 };
```

ProgressTracker 分别追踪输入和输出 token，避免重复计数——Claude API 的 input_tokens 是每轮累计值（包含所有历史上下文），而 output_tokens 是每轮增量值，两者的计数方式不同。

ToolActivity 记录子代理最近的工具调用活动，包括工具名称、输入参数和预计算的活动描述（如“Reading src/foo.ts”）。MAX_RECENT_ACTIVITIES（5 条）限制了活动历史的长度，避免内存膨胀。

ActivityDescriptionResolver 类型允许在记录时预计算活动描述，而不是在渲染时计算——这是一个性能优化，避免在 UI 渲染路径上执行工具描述解析。

子代理的输出通过磁盘文件中转（getTaskOutputPath()），而不是内存缓冲。initTaskOutputAsSymlink() 将输出文件链接到会话存储目录，使得子代理的完整对话记录可以在会话结束后保留。

子代理与主代理之间的通信通过 AppState 中的任务状态实现。主代理通过 registerTask() 注册子代理任务，通过 updateTaskState() 更新进度信息。子代理完成后，通过 enqueuePendingNotification() 向主代理的消息队列注入完成通知——这个通知会在主代理的下一轮 query 循环中被处理。

killShellTasksForAgent() 在子代理终止时清理其创建的所有后台 shell 任务，防止孤儿进程。cleanupAgentTracking() 清理 API 调用追踪状态。createChild-

`AbortController()` 确保父代理中断时子代理也被中断。

子代理还支持 `worktree` 隔离——`LocalAgentTask.tsx` 中引用了 `WORKTREE_PATH_TAG` 和 `WORKTREE_BRANCH_TAG`，表明子代理可以在独立的 `git worktree` 中运行，避免与主代理的文件操作冲突。

6.4 Companion 系统：虚拟伙伴

`src/buddy/` 目录实现的并非传统意义上的子代理系统，而是一个虚拟伙伴 (Companion) 机制。`companion.ts` 使用确定性伪随机数生成器 (Mulberry32 PRNG) 基于用户身份生成一个虚拟角色，具有物种、稀有度、属性值等特征。

`prompt.ts` 中的 `companionIntroText()` 函数揭示了 Companion 的实际定位：它是一个“坐在用户输入框旁边的小动物”，偶尔在气泡中发表评论。当用户直接称呼 Companion 的名字时，气泡会回应。

Companion 系统通过 `BUDDY feature flag` 门控，在外部版本中关闭。它是一个趣味性功能，不参与实际的代码执行或任务处理。

真正的子代理能力由 `AgentTool` (`src/tools/AgentTool/`) 提供，它在外部版本中可用，是用户通过工具调用派生子代理的标准方式。`AgentTool` 创建的子代理对应 `LocalAgentTask` 类型。

6.5 团队协作：InProcessTeammate

`InProcessTeammateTask/` 实现了进程内团队协作。源码注释明确了它与 `LocalAgentTask` 的区别：

1. 在同一个 Node.js 进程中运行，使用 `AsyncLocalStorage` 实现隔离
2. 具有团队感知的身份标识 (`agentName@teamName`)
3. 支持计划模式审批流程
4. 可以处于空闲状态（等待工作）或活跃状态（处理中）

消息注入机制是团队协作的核心。`injectUserMessageToTeammate()` 函数将消息放入团队成员的 `pendingUserMessages` 队列，同时追加到 `messages` 列表以便立即在 UI 中显示。消息注入允许在 `running` 和空闲状态下进行，只有终态 (`completed/failed/killed`) 才会拒绝。

`requestTeammateShutdown()` 通过设置 `shutdownRequested` 标志请求团队成员优雅退出，而不是直接终止。`appendTeammateMessage()` 将消息追加到团队成员的对话历史，使用 `appendCappedMessage()` 限制历史长度。

团队记忆同步由 `src/services/teamMemorySync/` 处理：

- `watcher.ts`: 文件系统监听，检测记忆文件变更
- `secretScanner.ts`: 扫描记忆内容中的敏感信息
- `teamMemSecretGuard.ts`: 防止 API key、密码等通过团队记忆泄露

安全扫描是团队记忆同步的关键环节——多代理共享记忆时，必须防止敏感信息跨代理传播。

6.6 远程执行：Remote 模块

`src/remote/` 目录实现了远程代理执行能力，包含 4 个文件：

`RemoteSessionManager.ts` 管理远程会话的生命周期——创建、连接、断开、重连。`SessionsWebSocket.ts` 实现了与远程 Claude 实例的 WebSocket 通信。

`sdkMessageAdapter.ts` 负责消息格式转换，将本地消息格式适配为远程 SDK 期望的格式。`remotePermissionBridge.ts` 处理远程执行中的权限同步——当远程代理需要执行敏感操作时，权限请求需要传回本地由用户确认。

远程执行的典型场景是 Claude Code Remote (CCR)：用户在本地终端操作，但实际的代码执行发生在远程容器中。这需要解决权限同步、文件系统映射、网络代理等一系列工程问题。

6.7 任务生命周期管理

所有任务类型共享相同的生命周期。图 6.1 展示了任务状态机。

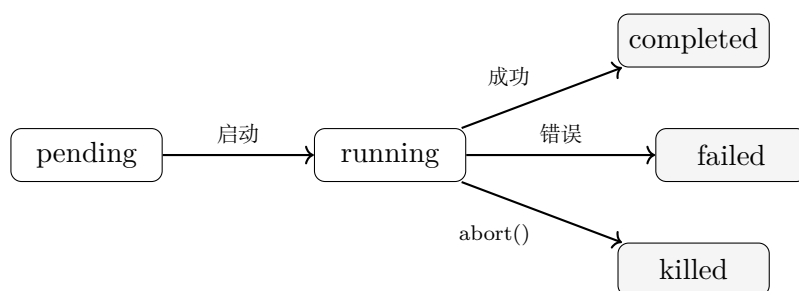


图 6.1: 任务状态机：pending → running → completed/failed/killed

`AppState` 中维护了所有活跃任务的列表，UI 组件通过订阅状态变更来更新任务显示。`isTerminalTaskStatus()` 判断 `completed`、`failed`、`killed` 三个终态，用于防止向已结束的任务注入消息和触发资源清理。

任务的取消通过 `AbortController` 实现。每个任务在创建时获得一个 `abortController`，外部可以通过调用 `abort()` 来请求取消。任务实现需要检查 `abortController.signal` 来响应取消请求。子代理任务还支持 `createChildAbortController()`，创建与父级联动的中断控制器——父级中断时子级自动中断。

`stopTask.ts` 提供了统一的停止接口，它根据任务类型分发到对应的 `kill()` 实现。不同类型的任务有不同的清理逻辑——`shell` 任务需要终止子进程树，代理任务需要中断 API 调用并清理输出文件，团队成员任务通过 `killInProcessTeammate()` 处理。

任务状态更新通过 `updateTaskState()` 函数统一管理，它接受一个类型安全的更新函数，确保状态转换的原子性。`registerTask()` 在 `AppState` 中注册新任务，`registerCleanup()` 确保进程退出时所有任务资源被正确清理。

第七章 扩展系统：Commands、Skills、MCP、Plugins、Bridge

7.1 扩展层次概览

Claude Code 的扩展能力不是单一机制，而是一个多层架构。从内到外，扩展层次依次为：

1. 内置工具 (`src/tools/`) —— 编译时确定，不可外部扩展
2. 斜杠命令 (`src/commands/`) —— 编译时确定，部分通过 feature flag 门控
3. 技能系统 (`src/skills/`) —— 运行时加载的提示词模板
4. MCP 协议 (`src/services/mcp/`) —— 标准化的外部工具协议
5. 插件系统 (`src/plugins/`) —— 打包了 MCP 服务器和技能的分发单元
6. Bridge 桥接 (`src/bridge/`) —— IDE 与 Claude Code 的通信层（内部功能）

每一层都建立在前一层之上，提供更高层次的抽象和更灵活的扩展能力。

7.2 技能系统：可复用的提示词模板

`src/skills/` 目录实现了技能 (Skill) 系统。技能本质上是结构化的 markdown 文件，包含 frontmatter 元数据和提示词正文，用户通过 `/skill-name` 语法触发。

7.2.1 技能加载

`loadSkillsDir.ts` 负责从文件系统加载技能定义。它的 import 列表揭示了技能系统的丰富能力：frontmatter 解析 (`parseFrontmatter`)、参数替换 (`substituteArguments`)、shell 命令执行 (`executeShellCommandsInPrompt`)、gitignore 过滤 (`isPathGitignored`)、token 估算 (`roughTokenCountEstimation`)。

技能可以来自多个位置，按优先级排列：

1. 内置技能：src/skills/bundled/——随 Claude Code 分发
2. 用户全局技能： ~/.claude/skills/——用户自定义
3. 项目级技能： .claude/skills/——项目特定
4. 额外目录：通过 getAdditionalDirectoriesForClaudeMd() 获取的附加路径

技能文件的 frontmatter 支持多种配置：描述（用于触发匹配）、参数定义（parseArgumentNames）、shell 命令（parseShellFrontmatter）、effort 级别（parseEffortValue）、模型指定（parseUserSpecifiedModel）、工具限制（parseSlashCommandToolsFromFrontmatter）。

一个典型的技能文件结构如下：

```
1 ---
2 description: 简短描述，用于触发匹配
3 model: claude-sonnet-4-6
4 tools: [Read, Grep, Glob]
5 ---
6
7 技能提示词正文...
8 可以包含 $ARGUMENTS 参数占位符
```

技能加载时，roughTokenCountEstimation() 估算提示词的 token 开销，isPathGitignored() 过滤被 gitignore 排除的技能文件。技能文件中还可以嵌入 shell 命令（通过 parseShellFrontmatter），这些命令在技能触发时执行，其输出被注入到提示词中——这使得技能可以动态收集上下文信息。

bundledSkills.ts 注册内置技能，mcpSkillBuilders.ts 将 MCP 服务器的能力包装为技能接口——这意味着 MCP 工具可以通过技能系统暴露给用户，提供更友好的 /skill-name 调用方式。

SkillTool（位于 src/tools/SkillTool/）是技能系统与工具系统的桥梁——当 Claude 决定调用一个技能时，它通过 SkillTool 执行。技能系统通过 MCP_SKILLS feature flag 控制，在外部版本中默认启用。

7.3 MCP 协议集成

src/services/mcp/ 是 Claude Code 中最复杂的扩展子系统之一，包含 20 余个文件。MCP（Model Context Protocol）是一个标准化协议，允许外部服务向 Claude 提供工具、资源和提示词。

7.3.1 客户端协议

`client.ts` 实现了完整的 MCP 客户端。从 `import` 列表可以看到它使用了 `@modelcontextprotocol/sdk` 的多个模块：

- `Client`：MCP 客户端核心类
- `StdioClientTransport`：标准 I/O 传输——通过子进程的 `stdin/stdout` 通信
- `SSEClientTransport`：Server-Sent Events 传输——通过 HTTP 长连接通信
- `StreamableHTTPClientTransport`：可流式 HTTP 传输

客户端支持 MCP 协议的三种核心能力：工具 (`ListToolsResult`、`CallToolResultSchema`)、资源 (`ListResourcesResultSchema`) 和提示词 (`ListPromptsResult`)。

`ElicitRequestSchema` 处理 MCP 的 elicitation 机制——当 MCP 工具需要用户提供额外输入时（如 OAuth 授权），它通过 elicitation 请求触发交互式对话框。

MCP 工具在 Claude Code 中被包装为 `MCPTool` (`src/tools/MCPTool/`)，资源通过 `ListMcpResourcesTool` 和 `ReadMcpResourceTool` 访问，认证通过 `McpAuthTool` 处理。

图 7.1 展示了 MCP 工具从外部服务器到 Claude 可调用工具的包装层次。

7.3.2 连接管理

`MCPConnectionManager.tsx` 是 MCP 连接的核心管理器，它是一个 React 组件 (`.tsx` 扩展名)，负责 MCP 服务器的发现、连接、断开和重连。作为 React 组件意味着连接状态变更会自动触发 UI 更新。

`config.ts` 处理 MCP 服务器配置的解析和验证，`normalization.ts` 统一不同来源的配置格式，`envExpansion.ts` 处理配置中的环境变量展开。

7.3.3 安全与权限

`auth.ts` 处理 MCP 服务器的认证，`channelPermissions.ts` 和 `channelAllowlist.ts` 控制哪些 MCP 通道被允许。`channelNotification.ts` 在新 MCP 通道连接时通知用户。

`officialRegistry.ts` 管理官方 MCP 服务器注册表——`prefetchOfficialMcpUrls()`（在 `main.tsx` 中调用）预取官方注册表，加速 MCP 服务器的发现。

`xaa.ts` 和 `xaaIdpLogin.ts` 处理企业级身份认证集成，支持组织内部的 MCP 服务器认证。

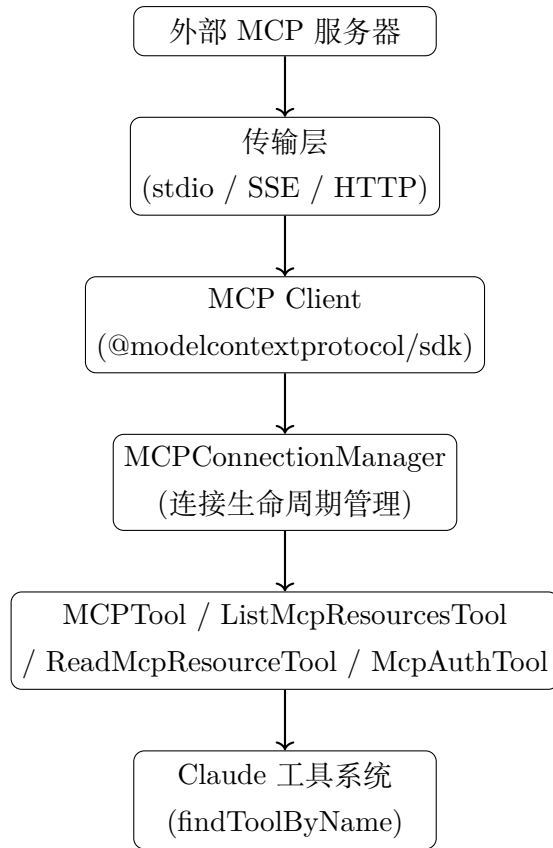


图 7.1: MCP 工具包装层次

7.4 插件系统

`src/plugins/` 目录实现了插件系统。插件是 MCP 服务器和技能的打包分发单元。一个插件可以包含：MCP 服务器定义、技能模板、配置文件。

`builtinPlugins.ts` 注册内置插件，`bundled/` 目录包含打包的插件资源。用户通过 `/mcp` 命令或配置文件安装和管理插件。

插件系统的加载由 `loadAllPluginsCacheOnly()`（在 `QueryEngine.ts` 中引用）处理，它从缓存中加载已安装的插件，避免每次启动都重新解析。`skillChangeDetector.ts`（在 `main.tsx` 中引用）监控技能文件变更，支持热重载。

7.5 Bridge 桥接层

`src/bridge/` 目录包含 20 余个文件，实现了 Claude Code 与 IDE 之间的桥接通信。这是一个内部功能，通过 `BRIDGE_MODE` feature flag 门控。

Bridge 的核心架构分为三层：

连接层 `bridgeMain.ts` 是桥接模式的主入口，`bridgeConfig.ts` 和 `bridgeEnabled.ts` 处理配置与启用检查。启动前需要通过 OAuth 认证检查和 GrowthBook

运行时门控。

通信层 `bridgeMessaging.ts` 定义消息传递协议，`bridgeApi.ts` 定义桥接 API。`inboundMessages.ts` 和 `inboundAttachments.ts` 处理从 IDE 传入的消息和附件。`createSession.ts` 和 `codeSessionApi.ts` 管理桥接会话。

安全层 `bridgePermissionCallbacks.ts` 处理权限回调——IDE 端的权限确认需要通过桥接传回 Claude Code。`jwtUtils.ts` 处理 JWT token 认证。`capacityWake.ts` 和 `flushGate.ts` 处理连接容量控制。

Bridge 模式允许 IDE（如 VS Code、JetBrains）作为 Claude Code 的前端，终端界面被 IDE 的 UI 替代。`bridgeUI.ts` 负责 UI 适配，`initReplBridge.ts` 在 REPL 启动时初始化桥接连接。

7.6 扩展系统的设计权衡

多层扩展架构的设计反映了不同的扩展需求：

层次	复杂度	适用场景
技能	低（markdown 文件）	提示词模板、工作流快捷方式
MCP	中（需实现服务器）	外部工具集成、数据源接入
插件	中（打包分发）	完整的功能扩展包
Bridge	高（协议对接）	IDE 深度集成（仅内部）

表 7.1: 扩展层次的复杂度与适用场景

这种分层设计的好处是：简单需求用简单机制（技能），复杂需求用复杂机制（MCP/插件），极端需求用极端机制（Bridge）。用户不需要为了添加一个提示词模板而去实现一个 MCP 服务器。

从实现角度看，技能系统的成本最低——只需要文件系统读取和 frontmatter 解析。MCP 的成本最高——需要管理子进程生命周期、处理多种传输协议、实现认证和权限控制。插件系统介于两者之间，它复用了 MCP 和技能的基础设施，只增加了打包和分发的逻辑。

第八章 权限、安全、策略与治理

8.1 权限模式：PermissionMode 的层次

Claude Code 的权限系统核心是 PermissionMode，定义在 src/types/permissions.ts 中。源码将权限模式分为外部可见和内部两个层次：

```
1  const EXTERNAL_PERMISSION_MODES = [
2    'acceptEdits',
3    'bypassPermissions',
4    'default',
5    'dontAsk',
6    'plan',
7  ] as const
8
9  type InternalPermissionMode =
10    ExternalPermissionMode | 'auto' | 'bubble'
```

各模式的行为差异：

模式	行为
default	每次工具调用都需要用户确认
acceptEdits	文件编辑自动允许，其他操作仍需确认
plan	只允许只读操作，写操作需要确认
dontAsk	不弹出确认对话框，不允许的操作直接拒绝
bypassPermissions	跳过所有权限检查（需要显式启用）
auto	自动模式，通过分类器判断是否安全（feature flag 门控）
bubble	内部模式，用于子代理权限冒泡

表 8.1: PermissionMode 各级别行为

auto 模式通过 TRANSCRIPT_CLASSIFIER feature flag 门控，只有在该开关启用时才会出现在运行时验证集合中。bubble 模式是纯内部模式，不对用户暴露。

8.2 权限规则：来源与优先级

权限规则不是简单的布尔值，而是带有来源标记的结构化数据：

```

1 type PermissionRuleSource =
2   | 'userSettings'      // 用户全局设置
3   | 'projectSettings'   // 项目级设置
4   | 'localSettings'     // 本地设置
5   | 'flagSettings'      // 命令行标志
6   | 'policySettings'    // 组织策略
7   | 'cliArg'            // CLI 参数
8   | 'command'           // 命令触发
9   | 'session'           // 会话内设置
10
11 type PermissionRule = {
12   source: PermissionRuleSource
13   ruleBehavior: PermissionBehavior // 'allow' | 'deny' | 'ask'
14   ruleValue: PermissionRuleValue   // toolName + ruleContent
15 }
```

每条规则指定了工具名称（toolName）和可选的规则内容（ruleContent），以及该规则的行为（允许、拒绝或询问）。来源标记使得系统可以追溯每条规则的设置位置，便于调试和审计。

ToolPermissionContext 将规则按行为分为三组（alwaysAllowRules、always-DenyRules、alwaysAskRules），每组内按来源分组。规则的优先级是：deny > ask > allow，确保安全约束不会被低优先级的允许规则覆盖。

整个权限上下文使用 DeepImmutable 包装，防止运行时意外修改权限配置。

8.3 工具级权限检查：CanUseToolFn

useCanUseTool.tsx（位于 src/hooks/）实现了 CanUseToolFn 类型的权限检查函数。每次 Claude 请求调用工具时，这个函数被调用来决定是否允许。

检查流程综合考虑多个因素：

1. 当前权限模式——决定默认行为
2. 三组规则匹配——deny 规则优先于 allow 规则
3. 用户的历史授权决策——”始终允许”选项会写入 session 来源的规则
4. 组织策略限制——PolicyLimits 可以覆盖用户设置

- 5. 后台代理特殊处理——`shouldAvoidPermissionPrompts` 为 `true` 时，无法弹出 UI 的代理自动拒绝需要确认的操作
- 6. 协调器模式——`awaitAutomatedChecksBeforeDialog` 为 `true` 时，先等待自动化检查（分类器、hook）再显示对话框

图 8.1 展示了权限检查的决策流程。

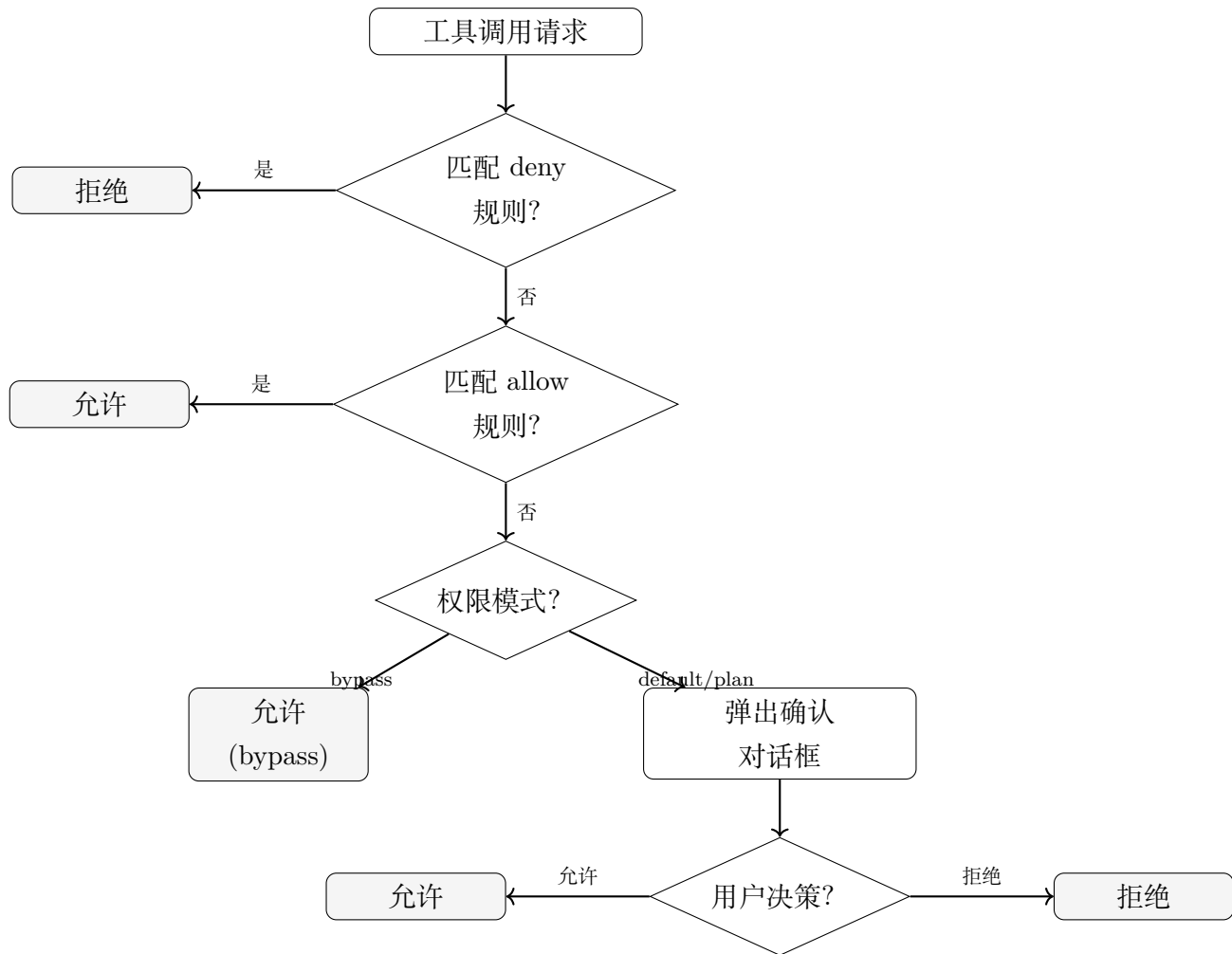


图 8.1: 权限检查决策流程

当权限检查需要用户确认时，`PermissionRequest` 组件渲染交互式对话框。`ToolPermissionContext` 中的 `prePlanMode` 字段记录了进入 `plan` 模式前的权限模式，以便退出时恢复。

8.4 文件系统安全边界

Claude Code 对文件系统操作实施了边界控制。`src/utils/permissions/-filesystem.ts` 中的 `isScratchpadEnabled()` 和 `ensureScratchpadDir()` 函数管理 `scratchpad` 目录——一个受限的临时工作区。

文件操作工具 (FileReadTool、FileEditTool、FileWriteTool) 在执行前会检查目标路径是否在允许的范围内。additionalWorkingDirectories (在 ToolPermission-Context 中) 允许用户显式添加额外的可访问目录，每个额外目录都是一个 AdditionalWorkingDirectory 对象，包含路径和权限范围。

8.5 策略限制：PolicyLimits

src/services/policyLimits/ 实现了组织级别的策略限制系统。源码注释详细说明了其设计原则：

- Console 用户 (API key)：全部有资格
- OAuth 用户 (Claude.ai)：仅 Team 和 Enterprise 订阅者有资格
- API 调用采用 fail-open 策略——获取失败时继续运行，不施加限制
- 无策略限制的用户收到空的限制列表

fail-open 是一个关键的设计决策：策略限制服务不可用时，系统选择继续运行而不是阻塞用户。这意味着策略限制是“尽力而为”的治理，而不是硬性的安全边界。

实现细节方面，策略通过 HTTP API 获取，使用 ETag 缓存避免重复下载，本地缓存在 policy-limits.json 文件中。获取超时为 10 秒 (FETCH_TIMEOUT_MS)，最多重试 5 次 (DEFAULT_MAX_RETRIES)，后台每小时轮询一次 (POLLING_INTERVAL_MS)。

isPolicyAllowed() 是策略检查的核心函数，在各个功能入口处被调用。例如，Bridge 模式启动前检查 isPolicyAllowed('allow_remote_control')。

8.6 OAuth 认证

src/services/oauth/ 实现了 Claude Code 的 OAuth 认证流程：

- client.ts：OAuth 客户端实现
- auth-code-listener.ts：授权码监听器——启动本地 HTTP 服务器接收 OAuth 回调
- crypto.ts：PKCE (Proof Key for Code Exchange) 加密支持
- getOAuthProfile.ts：获取用户 OAuth 配置文件

OAuth 流程使用 PKCE 增强安全性，这是 CLI 工具的标准做法——CLI 无法安全存储 client secret，PKCE 通过动态生成的 code verifier 替代。认证令牌存储在 macOS 钥匙串中（第 2 章提到的 startKeychainPrefetch() 就是预取这些令牌）。

8.7 SSRF 防护

`src/utils/hooks/ssrfGuard.ts` 实现了 SSRF (Server-Side Request Forgery) 防护。当 Claude 请求访问 URL 时 (通过 `WebFetchTool`)，SSRF Guard 检查目标地址是否安全，阻止对内网地址、localhost、云元数据服务 (如 `169.254.169.254`) 等敏感端点的访问。

这是一个必要的防护层——Claude 可能被提示词注入攻击诱导去访问内部服务，SSRF Guard 在网络层阻断这类请求。

8.8 MoreRight: 权限提升

`src/moreright/useMoreRight.tsx` 实现了权限提升的 UI 交互。当用户需要从受限模式切换到更高权限模式时 (例如从 `default` 切换到 `bypassPermissions`)，这个组件处理确认对话框和模式切换逻辑。

权限提升是单向的——用户可以在会话中提升权限，但不能降低 (降低需要重新启动会话)。`isBypassPermissionsModeAvailable` 标志控制 `bypass` 模式是否可用，`isAutoModeAvailable` 控制 `auto` 模式是否可用。

8.9 安全模型的设计哲学

Claude Code 的安全模型遵循几个设计原则：

第一，默认安全。未经明确授权的操作不会执行。权限检查发生在工具执行之前，而不是之后。

第二，分层控制。从用户级 (`PermissionMode` + 规则) 到组织级 (`PolicyLimits`) 到协议级 (SSRF Guard)，每一层都可以独立施加约束。层与层之间的关系是“与”——所有层都通过才允许执行。

第三，可审计。权限拒绝被记录在 `permissionDenials` 中，每条规则都带有来源标记 (`PermissionRuleSource`)，用户可以追溯哪些操作被拒绝、被哪条规则拒绝、规则来自哪个配置层级。

第四，渐进式信任。用户可以从 `default` 模式开始，逐步放宽到 `acceptEdits` 或 `bypassPermissions`。“始终允许”选项将决策持久化为 `session` 来源的规则，避免重复确认。

第五，fail-open vs fail-closed 的分层选择。本地权限检查是 fail-closed (检查失败则拒绝)，远程策略限制是 fail-open (获取失败则不限制)。这个分层确保了：本地安全不依赖网络可用性，组织治理不阻塞个人使用。

第九章 性能、可靠性与产品化细节

9.1 启动性能优化

Claude Code 对启动性能投入了大量工程努力。从第 2 章的分析中已经看到多个优化手段，本节将它们系统化。

`src/utils/startupProfiler.js` 提供了启动性能度量基础设施。`profileCheckpoint()` 函数在关键节点打点，记录从进程启动到各个里程碑的耗时。这些检查点贯穿整个启动路径：`cli_entry`、`main_tsx_entry`、`init_function_start`、`init_configs_enabled`、`init_network_configured` 等。

启动优化的核心策略包括：

1. 动态 import: `cli.tsx` 中所有模块导入都是动态的，快速路径零模块加载
2. 并行预取: `startMdmRawRead()` 和 `startKeychainPrefetch()` 在模块加载期间并行执行 I/O
3. 懒加载: native 模块、MCP 连接、插件等在首次使用时才加载
4. 预连接: `preconnectAnthropicApi()` 在初始化期间预热 TCP+TLS 连接（约 100-200ms），与后续 100ms 的准备工作重叠
5. 条件加载: feature flag 消除未使用的代码路径，减少模块评估开销
6. 延迟初始化: OpenTelemetry（约 400KB）和 gRPC 导出器（约 700KB）在实际需要时才加载

图 9.1 展示了启动过程中关键操作的时序关系。

预连接的实现有明显的前置条件：必须在 CA 证书和代理配置完成之后执行，确保预热的连接使用正确的传输配置。对于使用代理、mTLS 或 Unix socket 的场景，预连接被跳过——SDK 的连接池不会复用全局池中的连接。

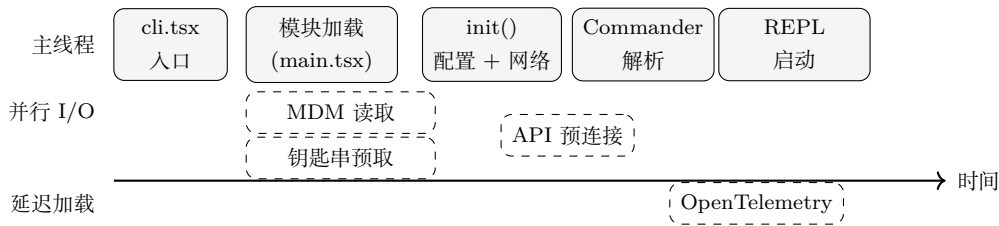


图 9.1: 启动性能时序：并行 I/O 与延迟加载

9.2 API 调用与流式响应

`src/services/api/claude.ts` 是与 Claude API 通信的核心模块。从其 `import` 列表可以看到它使用 Anthropic SDK 的 Beta API (`BetaMessage`、`BetaMessageStreamParams`)，支持流式响应 (`Stream<BetaRawMessageStreamEvent>`)。

API 调用的构建过程涉及多个步骤：将内部消息格式转换为 API 参数 (`MessageParam`)、将工具定义转换为 API schema (`toolToAPISchema`)、处理系统提示词前缀 (`splitSysPromptPrefix`)、注入归因头 (`getAttributionHeader`)。

流式响应处理是 Claude Code 交互体验的关键——用户看到的逐字输出效果依赖于对 `BetaRawMessageStreamEvent` 的实时处理。每个流事件可能包含文本增量 (`TextDelta`)、工具调用 (`ToolUse`) 或思考内容 (`Thinking`)。

9.3 错误处理与重试策略

`src/services/api/errors.ts` 定义了 API 错误的分类体系。`categorizeRetryableAPIError()` 函数将 API 错误分为可重试和不可重试两类。可重试错误包括速率限制 (429)、服务器错误 (500/502/503) 等；不可重试错误包括认证失败 (401)、请求格式错误 (400) 等。

`PROMPT_TOO_LONG_ERROR_MESSAGE` 和 `isPromptTooLongMessage()` 处理上下文超限错误——这是触发自动压缩的信号之一。

9.3.1 withRetry 重试机制

`withRetry.ts` 实现了完整的重试策略，其核心参数揭示了设计考量：

```
1 const DEFAULT_MAX_RETRIES = 10
2 const FLOOR_OUTPUT_TOKENS = 3000
3 const MAX_529_RETRIES = 3
4 const BASE_DELAY_MS = 500
```

默认最多重试 10 次，基础延迟 500ms（指数退避）。529 错误（容量过载）单独限制为最多 3 次重试——源码注释解释了原因：在容量级联故障期间，每次重试会

产生 3-10 倍的网关放大效应。

重试策略还区分前台和后台查询：用户正在等待结果的前台查询（如主对话）会重试 529 错误，而后台任务（摘要生成、标题生成、分类器）遇到 529 时立即放弃——用户不会看到这些失败，重试只会加剧容量压力。

`withRetry.ts` 还处理了多种特殊场景：

- OAuth 401 错误：调用 `handleOAuth401Error()` 尝试刷新令牌后重试
- AWS 凭证错误：清除凭证缓存（`clearAwsCredentialsCache()`）后重试
- Fast Mode 过载：触发冷却期（`triggerFastModeCooldown()`），回退到标准模式
- 连接错误：禁用 keep-alive（`disableKeepAlive()`）后重试，处理代理连接问题

`FallbackTriggeredError` 表示主模型失败后触发了备用模型——`QueryEngineConfig` 中的 `fallbackModel` 字段定义了备用模型。

9.3.2 API 服务模块全景

`src/services/api/` 目录包含 20 个文件，覆盖了 API 交互的完整生命周期：

模块	职责
<code>claude.ts</code>	核心 API 调用、流式响应处理
<code>client.ts</code>	API 客户端配置与实例管理
<code>withRetry.ts</code>	重试策略与错误恢复
<code>errors.ts</code>	错误分类与用户提示
<code>errorUtils.ts</code>	错误格式化工具
<code>bootstrap.ts</code>	启动引导数据获取
<code>logging.ts</code>	API 调用日志
<code>usage.ts</code>	Token 使用量追踪
<code>filesApi.ts</code>	文件上传/下载 API
<code>promptCacheBreakDetection.ts</code>	提示缓存失效检测

表 9.1: API 服务模块

9.4 费用追踪

`src/cost-tracker.ts` 实现了实时费用追踪系统。它从 `bootstrap/state.js` 导入了大量状态访问函数，追踪的维度包括：

- Token 使用量：输入 token、输出 token、缓存创建 token、缓存读取 token
- 费用：通过 `calculateUSDCost()` 将 token 使用量转换为美元费用
- 时间：总耗时、API 调用耗时、API 调用耗时（不含重试）、工具执行耗时
- 代码变更：新增行数、删除行数
- 按模型分组的使用量（`getUsageForModel()`）

`hasUnknownModelCost` 标志处理未知模型的费用计算——当使用自定义模型时，系统可能无法准确计算费用，此时会提示用户。

费用追踪与 `QueryEngineConfig` 中的 `maxBudgetUsd` 配合使用：当累计费用接近预算时，`CostThresholdDialog` 组件提醒用户。

9.5 Token 估算

`src/services/tokenEstimation.ts` 提供 token 计数能力。准确的 token 估算对上下文管理至关重要——它决定了何时触发压缩、还能容纳多少消息、是否接近预算限制。

`tokenCountWithEstimation()`（在 `utils/tokens.ts` 中）采用混合策略：在关键决策点（如压缩触发判断）使用精确计数，在非关键路径使用基于字符数的快速估算。`getContextWindowForModel()` 根据模型名称返回对应的上下文窗口大小，`getModelMaxOutputTokens()` 返回模型的最大输出 token 数。

9.6 分析与可观测性

`src/services/analytics/` 实现了分析埋点系统。`logEvent()` 是所有分析事件的统一入口，其类型签名中包含 `AnalyticsMetadata_I_VERIFIED_THIS_IS_NOT_CODE_OR_FILEPATH`——这个冗长的类型名是一种工程约束，强制开发者确认元数据不包含代码内容或文件路径。

`GrowthBook` 提供运行时特性门控，与编译期 `feature flag` 互补。`getFeatureValue_CACHED_MAY_BE_STALE()` 函数名中的后缀提醒调用者返回值可能是缓存的旧值。`onGrowthBookRefresh()` 支持配置变更时的回调，例如重新初始化事件日志配置。

`sinkKillswitch.ts` 实现了分析管道的紧急关闭开关——当分析系统出现问题时，可以快速禁用所有事件输出，避免影响主功能。

`datadog.ts` 集成了 `Datadog` 监控，`firstPartyEventLogger.ts` 实现了第一方事件日志，使用 `OpenTelemetry` 的日志导出器。事件日志的初始化被延迟到 `init()` 中异步执行，避免阻塞启动。

9.7 诊断与调试

`src/services/diagnosticTracking.ts` 实现诊断追踪。`logForDiagnosticsNoPII()` 函数名中的 `NoPII` 后缀提醒开发者这个日志通道不应包含个人身份信息。

`src/utils/debug.js` 中的 `logForDebugging()` 提供开发调试日志，`logAntError()` 记录内部错误。这些日志通道与分析事件分离，确保调试信息不会进入生产分析管道。

9.8 产品化细节

除了上述核心系统，Claude Code 还包含多个产品化细节：

模块	用途
<code>services/preventSleep.ts</code>	防止系统休眠，确保长任务不被中断
<code>services/notifier.ts</code>	系统通知服务（任务完成、错误提醒）
<code>services/tips/</code>	使用提示系统
<code>migrations/</code>	数据迁移——处理配置格式升级（11 个迁移文件）
<code>utils/gracefulShutdown.js</code>	优雅退出——确保资源正确清理
<code>utils/cleanupRegistry.js</code>	清理注册表——统一管理退出时的清理回调
<code>utils/asciicast.js</code>	终端录制——将会话录制为 <code>asciicast</code> 格式
<code>utils/fpsTracker.js</code>	帧率追踪——监控终端渲染性能
<code>services/voice.ts</code>	语音输入支持（feature flag 门控）

表 9.2: 产品化细节模块

这些模块单独看不起眼，但它们共同构成了一个生产级 CLI 工具的完整体验。`migrations/` 目录中的 11 个迁移文件表明 Claude Code 经历了多次配置格式变更，每次都需要向后兼容地升级用户数据。`cleanupRegistry.js` 确保即使在异常退出时，临时文件、子进程、网络连接等资源也能被正确清理。

第十章 设计哲学与工程取舍

10.1 编译期 vs 运行时：双层门控

Claude Code 的特性控制采用了双层门控架构：编译期 feature flag 和运行时 GrowthBook。

编译期 feature flag 通过 `bun:bundle` 的 `feature()` API 实现，在构建时将布尔值内联并消除死代码。这意味着外部版本的二进制文件中物理上不包含内部功能的代码——不是隐藏，而是不存在。90 余个开关中，外部版本只启用了 4 个（`BUILTIN_EXPLORE_PLAN_AGENTS`、`TOKEN_BUDGET`、`MCP_SKILLS`、`COMPACTION_REMINDERS`）。

运行时 GrowthBook 提供了更灵活的控制：同一份二进制可以根据用户身份、组织策略、A/B 测试分组等条件动态启用或禁用功能。`getFeatureValue_CACHED_MAY_BE_STALE()` 的命名暗示了运行时门控的一个固有限制——缓存值可能过时，不适合做安全关键决策。

两层门控的分工是：编译期决定代码是否存在（安全边界），运行时决定功能是否激活（产品策略）。这种分离确保了即使运行时门控被绕过，受保护的代码也不会被执行。

从源码中可以观察到一个有趣的模式：某些功能同时使用两层门控。例如 Bridge 模式在 `cli.tsx` 中先检查 `feature('BRIDGE_MODE')`（编译期），通过后再调用 `getBridgeDisabledReason()`（运行时 GrowthBook）。这种双重检查确保了即使编译期开关被意外启用，运行时门控仍然可以阻止功能激活。

10.2 单文件产物的利弊

Claude Code 将 1906 个源文件打包为一个 22MB 的 `cli.js`。这个决策带来了明确的利弊。

优势：分发简单（一个文件 + `node_modules`），启动时无需解析模块依赖图，source map 可以完整保留，版本管理清晰（一个文件对应一个版本）。

代价：文件体积大，任何修改都需要重新构建整个产物，调试时需要 source map 支持，无法做增量更新。

从 `build.ts` 的 `minify: false` 配置可以推断, Anthropic 选择了可调试性优先于体积优化。`sourcemap: 'linked'` 保留了完整的源码映射, 使得生产环境的错误堆栈可以映射回原始 TypeScript 源码。对于一个 CLI 工具来说, 22MB 的体积在现代环境中是可接受的——`npm` 安装时的网络传输是一次性成本。

10.3 React 终端 UI 的得失

用 React 构建终端 UI 是一个不寻常的选择。`src/ink/` 中约 50 个文件的自研渲染引擎表明, Claude Code 团队不仅使用了 Ink 框架, 还对其进行了深度定制——包括自定义 `reconciler`、选择系统、搜索高亮、鼠标支持等。

这个选择的收益是: 声明式 UI 编程模型、组件化架构 (140+ 组件)、React 生态的工具链 (如 React Compiler 优化——`App.tsx` 中可以看到编译器生成的缓存数组)、状态管理的成熟模式 (Context + Provider)。

代价是: 运行时开销 (`reconciler`、Yoga 布局引擎)、内存占用 (虚拟 DOM 树)、学习曲线 (终端开发者需要理解 React)、调试复杂度 (终端渲染问题叠加 React 状态问题)。

从组件数量和交互复杂度来看, 这个选择是合理的——`REPL.tsx` 单个文件就有 60+ 行 `import`, 整合了权限对话框、消息渲染、命令队列、后台任务导航、搜索高亮、团队协作等功能。用传统的字符串拼接方式维护这样的界面几乎不可行。

10.4 上下文工程的实用主义

第 5 章分析的上下文管理系统体现了实用主义的工程哲学。

token 估算采用混合精度策略——关键路径精确计数, 非关键路径粗略估算。压缩预留空间基于 p99.99 统计数据 (17,387 token) 而非理论最大值。自动压缩的熔断器 (`consecutiveFailures`) 在连续失败时停止重试, 而不是无限循环。

记忆系统的 `MEMORY.md` 采用双重截断 (200 行 + 25KB), 而不是单一限制。这种设计处理了两种边缘情况: 行数少但每行很长 (字节截断生效), 行数多但每行很短 (行数截断生效)。

这些设计选择的共同特点是: 基于实际数据而非理论假设, 处理边缘情况而非忽略它们, 在精度和性能之间做明确的权衡。

10.5 权限模型的演化方向

Claude Code 的权限模型从简单的“允许/拒绝”发展到了多层次的规则系统 (`alwaysAllow/alwaysDeny/alwaysAsk`), 再到组织级策略 (`PolicyLimits`)。

这个演化路径反映了 Agent 工具的核心张力：自主性与安全性。用户希望 Claude 能自主完成复杂任务（减少确认对话框），但又不希望它执行危险操作（删除文件、推送代码）。

当前的解决方案是渐进式信任：默认严格，用户可以逐步放宽。Permission-Mode 的多个级别（default → acceptEdits → bypassPermissions）提供了从保守到激进的完整光谱。权限规则的 8 种来源（userSettings、projectSettings、localSettings、flagSettings、policySettings、cliArg、command、session）确保了每条规则都可追溯。

PolicyLimits 的 fail-open 设计是一个值得注意的决策：组织策略获取失败时不阻塞用户。这意味着策略限制是治理工具而非安全边界——真正的安全由本地权限检查（fail-closed）保证。

10.6 扩展性与安全性的张力

MCP 协议为 Claude Code 带来了强大的扩展能力，但也引入了安全风险。外部 MCP 服务器可以提供任意工具，这些工具的行为不受 Claude Code 直接控制。

Claude Code 的应对策略包括：MCP 通道白名单（channelAllowlist.ts）、MCP 权限检查（channelPermissions.ts）、SSRF 防护（ssrfGuard.ts）、团队记忆的敏感信息扫描（secretScanner.ts）。但这些都是防御性措施，无法完全消除风险。

这是所有可扩展系统面临的根本问题：扩展能力越强，攻击面越大。Claude Code 选择了“默认不信任，显式授权”的策略，将最终决策权交给用户。

10.7 工程复杂度管理

一个 1906 文件的 TypeScript 项目如何保持可维护性？从源码中可以观察到几个策略：

第一，模块化隔离。每个子系统有独立的目录，通过类型接口交互。Tool.ts 定义接口，tools/ 目录实现接口，tools.ts 组装实例。循环依赖通过 types/permissions.ts 等纯类型文件打破——源码注释明确写道“Pure permission type definitions extracted to break import cycles”。

第二，feature flag 隔离。内部功能通过编译期开关完全隔离。条件 require 的模式（feature('X') ? require('...') : null）确保未启用的功能不会增加模块加载开销。

第三，类型系统约束。DeepImmutable 防止意外修改权限配置。AnalyticsMetadata_I_VERIFIED_THIS_IS_NOT_CODE_OR_FILEPATHS 通过冗长的类型名强制开发者审视数据安全。NoPII 后缀标记不应包含个人信息的日志通道。

第四，懒加载策略。动态 import 贯穿整个代码库——从 cli.tsx 的快速路径到 init.ts 的 OpenTelemetry 延迟加载，再到 native 模块的首次使用加载。这不仅优

化了启动性能，也减少了模块间的耦合。

第五，防御性编程。`looksLikePrompt()` 区分“命令慢”和“命令等待输入”。`isTerminalTaskStatus()` 防止向已结束的任务注入消息。`truncateEntrypointContent()` 的双重截断处理两种边缘情况。这些细节体现了对生产环境异常场景的充分考虑。

10.8 类型系统作为工程约束工具

Claude Code 对 TypeScript 类型系统的使用超越了常规的类型安全，将其作为工程流程约束的载体。

最典型的例子是 `AnalyticsMetadata_I_VERIFIED_THIS_IS_NOT_CODE_OR_FILEPATHS`。这个类型名出现在 `logEvent()` 的参数签名中，强制每个调用点的开发者在编写代码时“阅读”这个类型名——它本身就是一条审查清单。这种做法在 `migrations/` 目录中也有体现：`migrateLegacyOpusToCurrent.ts`、`migrateSonnet45ToSonnet46.ts` 等文件名直接编码了迁移的语义。

`DeepImmutable` 类型包装器用于 `ToolPermissionContext`、消息映射器（`utils/messages/mappers.ts`）和应用状态存储（`state/AppStateStore.ts`）。它不是运行时检查——TypeScript 的 `readonly` 在运行时不存在——而是编译期约束，防止开发者在不经意间修改应该是只读的数据结构。

`logForDiagnosticsNoPII()` 中的 `NoPII` 后缀出现在多个模块中（`context.ts`、`bridgeMain.ts`、`jwtUtils.ts` 等）。它不是类型约束，而是命名约定约束——通过函数名提醒调用者这个日志通道的数据安全要求。

这三种模式（类型名约束、不可变包装、命名约定）共同构成了一套“代码即文档、类型即审查”的工程实践。它们的共同特点是：约束发生在编写代码时而非运行时，成本几乎为零，但能有效防止常见的安全和数据泄露问题。

10.9 从源码看产品演化

Claude Code 的源码中保留了大量产品演化的痕迹，这些痕迹为理解系统的设计决策提供了额外的上下文。

`migrations/` 目录中的 11 个迁移文件记录了配置格式的演化历史。`migrateLegacyOpusToCurrent.ts` 和 `migrateSonnet45ToSonnet46.ts` 表明模型默认值经历了多次变更，每次都需要向后兼容地升级用户配置。

feature flag 的命名也透露了产品方向：`KAIROS`（含 `KAIROS_DREAM`、`KAIROS_CHANNELS`、`KAIROS_GITHUB_WEBHOOKS`、`KAIROS_PUSH_NOTIFICATION`）暗示了一个完整的助手模式子系统。`COORDINATOR_MODE` 暗示了多代理协调能力。`DAEMON` 和 `BG_SESSIONS` 暗

示了后台持续运行的能力。这些功能在外部版本中关闭，但源码结构表明它们已经有了相当完整的实现。

`bridge/` 目录的 20+ 文件表明 IDE 集成不是简单的 API 包装，而是一个完整的通信协议层——包含会话管理、权限同步、附件传输、容量控制等。这解释了为什么 Claude Code 的 VS Code 和 JetBrains 扩展能够提供接近原生终端的体验。

这些演化痕迹的价值在于：它们展示了一个 Agent 编程系统从 CLI 工具向平台化方向发展的路径。当前的外部版本是这个演化过程中的一个快照，而源码中的 feature flag 和内部模块则预示了未来可能的产品形态。

10.10 系统架构总览

回顾全书十章的分析，图 10.1 将 Claude Code 的核心子系统及其关系整合为一张架构图。

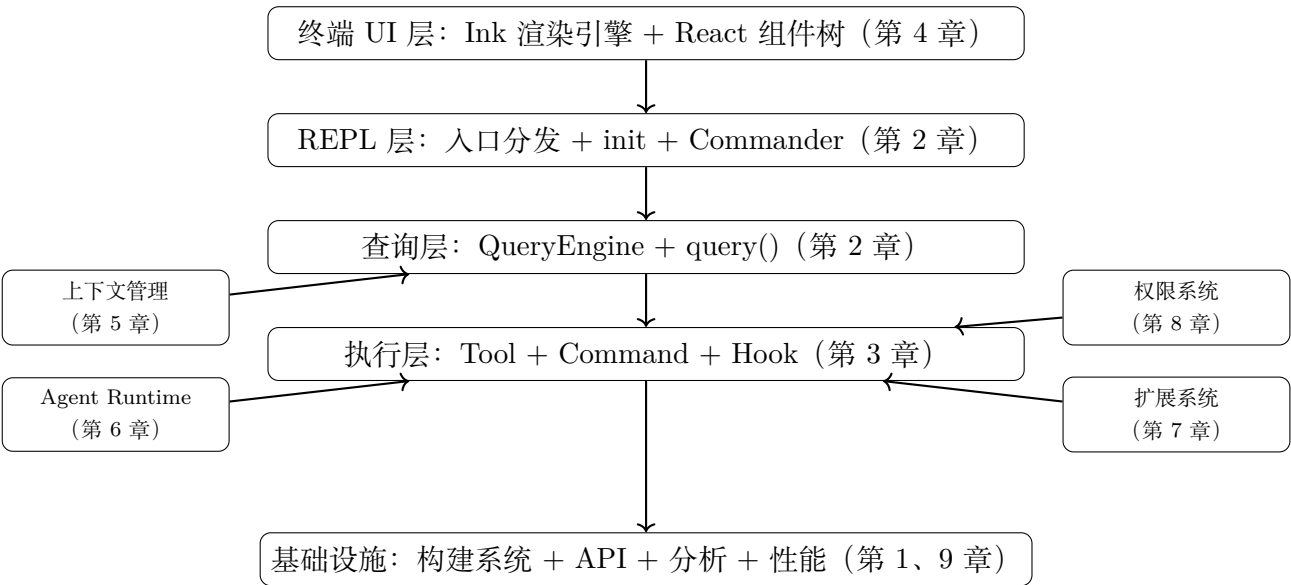


图 10.1: Claude Code 系统架构总览

从数据流的视角看，一次完整的用户交互经过以下路径：

1. 用户在终端输入文本，Ink 引擎捕获按键事件
2. REPL 层将输入传递给 QueryEngine
3. QueryEngine 组装系统提示词、记忆附件和消息历史（上下文管理）
4. query() 将组装好的上下文发送给 Claude API
5. Claude 返回文本和/或工具调用请求
6. 工具调用经过权限检查后执行，结果追加到消息历史

7. 若上下文接近窗口限制，触发自动压缩
8. 循环继续直到 Claude 不再请求工具调用
9. 最终响应通过 Ink 渲染到终端

这个数据流贯穿了本书的所有章节。每一章深入分析了流程中某个环节的实现细节，而第 10 章的目标是把这些环节重新连接起来，展示它们如何作为一个整体运作。

10.11 结语

Claude Code 的源码展示了一个真实的、正在演化的 Agent 编程系统。它不是教科书式的完美架构，而是在产品需求、工程约束和安全要求之间持续权衡的结果。

从构建系统的 feature flag 到终端 UI 的 React 化，从上下文压缩的统计驱动策略到权限模型的渐进式信任，每个设计决策都有其具体的工程背景和约束条件。理解这些决策背后的“为什么”，比记住具体的实现细节更有价值。

对于正在构建类似系统的工程师，Claude Code 的源码提供了一个参考坐标：不是唯一正确的做法，而是一种经过生产验证的做法。

附录 A 术语表

英文名	中文译名	定义	章节
Feature Flag	特性开关	编译期布尔常量，通过 <code>feature()</code> API 控制代码路径的存废	第 1 章
MACRO	编译期常量	通过 <code>define</code> 字段在构建时替换为字符串字面量的标识符	第 1 章
Bun	—	JavaScript/TypeScript 运行时与构建工具	第 1 章
Stub	存根	为不可用的私有包提供最小类型兼容接口的替代实现	第 1 章
Entry Point	入口点	构建和运行的起始文件 (<code>cli.tsx</code>)	第 2 章
REPL	交互式循环	Read-Eval-Print Loop，主交互模式	第 2 章
Query	查询	一次用户输入到 Claude 响应完成的完整交互周期	第 2 章
QueryEngine	查询引擎	管理查询生命周期、消息历史、系统提示词组装的核心组件	第 2 章
AppState	应用状态	全局应用状态对象，包含消息、任务、配置等	第 2 章
Tool	工具	Claude 可调用的能力单元，如文件读写、命令执行等	第 3 章
ToolUseContext	工具使用上下文	工具执行时的环境信息，包含权限、状态、配置等	第 3 章
Command	命令	用户通过 / 前缀触发的斜杠命令	第 3 章
Hook	钩子	在工具调用等事件前后执行的用户自定义 shell 命令	第 3 章

英文名	中文译名	定义	章节
CanUseToolFn	权限检查函数	每次工具调用前判断是否允许执行的函数	第 3 章
Ink	—	基于 React 的终端 UI 渲染框架	第 4 章
Keybinding	键绑定	键盘快捷键到操作的映射	第 4 章
Context	上下文	发送给 Claude API 的消息序列与系统提示词的组合	第 5 章
Compact	压缩	当上下文接近 token 限制时，对历史消息进行摘要压缩	第 5 章
Memory	记忆	跨会话持久化的用户偏好、项目信息等结构化知识	第 5 章
Task	任务	后台运行的工作单元（bash/agent/teammate 等类型）	第 6 章
TaskType	任务类型	任务分类：local_bash, local_agent, remote_agent 等	第 6 章
Companion	虚拟伙伴	基于用户身份生成的虚拟角色（内部功能）	第 6 章
Skill	技能	可复用的提示词模板，通过 /skill-name 触发	第 7 章
MCP	模型上下文协议	Model Context Protocol, 标准化的工具扩展协议	第 7 章
Plugin	插件	打包了 MCP 服务器和技能的分发单元	第 7 章
Bridge	桥接	IDE 与 Claude Code 之间的通信层（内部功能）	第 7 章
PermissionMode	权限模式	权限检查的严格程度级别（default/plan/auto 等）	第 8 章
PolicyLimits	策略限制	组织级别的功能限制策略	第 8 章
SSRF Guard	SSRF 防护	阻止对内网地址等敏感端点的访问	第 8 章
GrowthBook	—	运行时特性门控与 A/B 测试框架	第 9 章

附录 B 源码目录索引

以下索引基于 Claude Code 2.1.88 版本的 `src/` 目录结构。

顶层文件

文件	用途
<code>main.tsx</code>	主 REPL 逻辑, Commander 参数解析与会话启动
<code>Tool.ts</code>	工具类型系统定义 (接口, 不含实现)
<code>tools.ts</code>	工具实例注册与组装
<code>Task.ts</code>	任务类型与状态机定义
<code>tasks.ts</code>	任务系统入口
<code>QueryEngine.ts</code>	查询引擎——会话生命周期管理
<code>query.ts</code>	单次查询执行流
<code>commands.ts</code>	斜杠命令注册中心
<code>context.ts</code>	系统/用户上下文组装
<code>cost-tracker.ts</code>	费用追踪
<code>history.ts</code>	会话历史管理
<code>ink.ts</code>	Ink 渲染引擎导出

核心目录

目录	章节	用途
<code>entrypoints/</code>	第 2 章	CLI 入口点与初始化
<code>tools/</code>	第 3 章	40+ 工具实现
<code>commands/</code>	第 3 章	40+ 斜杠命令实现
<code>hooks/</code>	第 3 章	80+ React hook
<code>ink/</code>	第 4 章	自研终端渲染引擎
<code>components/</code>	第 4 章	140+ 终端 UI 组件
<code>screens/</code>	第 4 章	屏幕组件

目录	章节	用途
keybindings/	第 4 章	键绑定系统
context/	第 5 章	React Context 定义
memdir/	第 5 章	记忆目录系统
buddy/	第 6 章	Companion 虚拟伙伴
tasks/	第 6 章	任务类型实现
remote/	第 6 章	远程代理执行
skills/	第 7 章	技能系统
plugins/	第 7 章	插件系统
bridge/	第 7 章	IDE 桥接层（内部）
moreright/	第 8 章	权限提升 UI
state/	第 2 章	全局状态管理
types/	全书	共享类型定义
utils/	全书	工具函数（564 文件）
constants/	全书	全局常量
migrations/	第 9 章	数据迁移
vim/	第 4 章	Vim 模式支持
voice/	第 9 章	语音输入

services/ 子目录

目录	章节	用途
services/api/	第 9 章	API 调用、错误处理、重试
services/analytics/	第 9 章	分析埋点、GrowthBook
services/compact/	第 5 章	上下文压缩
services/mcp/	第 7 章	MCP 协议集成
services/oauth/	第 8 章	OAuth 认证流程
services/policyLimits/	第 8 章	组织策略限制
services/SessionMemory/	第 5 章	会话记忆管理
services/extractMemories/	第 5 章	自动记忆提取
services/teamMemorySync/	第 6 章	团队记忆同步
services/tools/	第 3 章	工具服务层
services/lsp/	第 3 章	LSP 服务器管理
services/plugins/	第 7 章	插件服务
services/remoteManagedSettings/	第 8 章	远程管理设置
services/tokenEstimation.ts	第 9 章	Token 估算
services/diagnosticTracking.ts	第 9 章	诊断追踪

目录	章节	用途
<code>services/preventSleep.ts</code>	第 9 章	防止系统休眠
<code>services/notifier.ts</code>	第 9 章	系统通知服务