Claude Code工程实战

从工具到平台

树 懒 老 K

个人网站　个人微信　公众号

Claude Code工程实战

树懒老K

作者：树懒老K

出版：树懒老K

年份：2026

目 录

第3章 记忆系统：CLAUDE.md 工程实践

第1章 架构全景：Claude Code的四层引擎

1.1 不要把它当 CLI 工具

很多开发者第一次接触 Claude Code 时的反应是：「哦，又一个 AI 终端助手。」然后装好，跑个 `claude`，让它改个文件，感觉不错——就把它归类为「加强版 Copilot」。

这是严重的认知压缩。

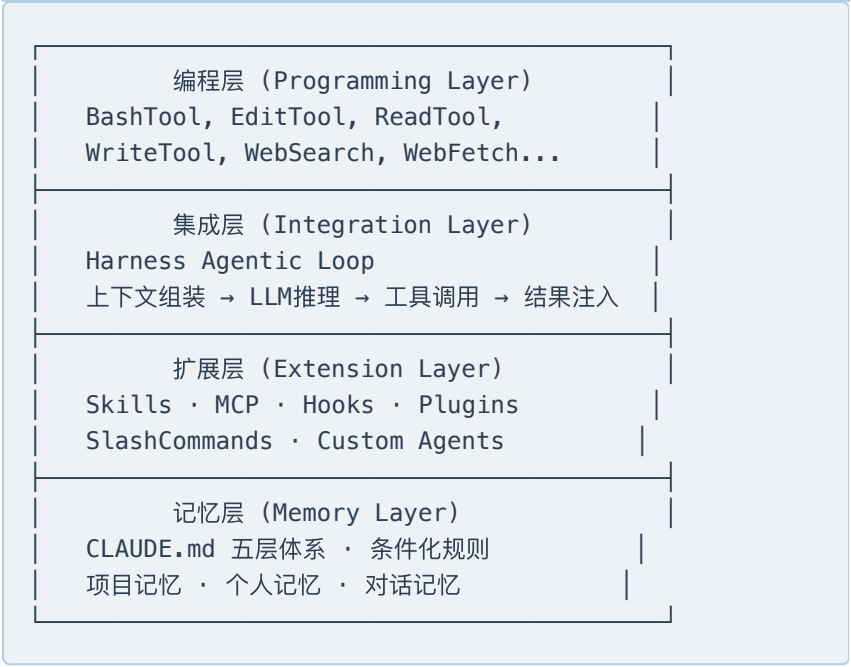
Claude Code 真正的定位是 **Agent 运行平台**。CLI 只是它最薄的一层外壳。如果你把眼光放到 2026 年的完整产品矩阵——CLI、Desktop 桌面应用、VS Code 插件、JetBrains 插件、Web 界面——你会发现所有这些端共享同一个核心引擎。写一次 CLAUDE.md，所有端生效；定义一个 Skill，所有端可用；配置一个 Hook，所有端触发。

这不是一个工具。这是一个平台。

理解这一点，是你后续深入学习记忆系统、Skills、子智能体、Hooks 的前提。没有全局架构观，你会把一个个特性当成孤立的 features 去学，永远拼不出完整的拼图。

1.2 四层架构总览

Claude Code 的引擎可以抽象为四个正交层。每一层有自己的职责边界，层与层之间通过明确的接口通信。



记忆层：Claude 的「长期记忆」

记忆层解决的是 AI Agent 的核心痛点：「你是谁？你在哪？你要遵守什么规则？」Claude Code 用 CLAUDE.md 文件体系回答了这三个问题。这不是一个简单的 system prompt——它是一个分层的、可继承的、带条件判断的规则引擎。

记忆层的工作时机非常明确：**每次对话开始前**，Harness 会从五层体系中收集规则、拼装上下文、注入到 system prompt。这意味着你写的每一条 CLAUDE.md 规则，都会在每一个对话轮次中参与决策。

扩展层：能力的「插件化」

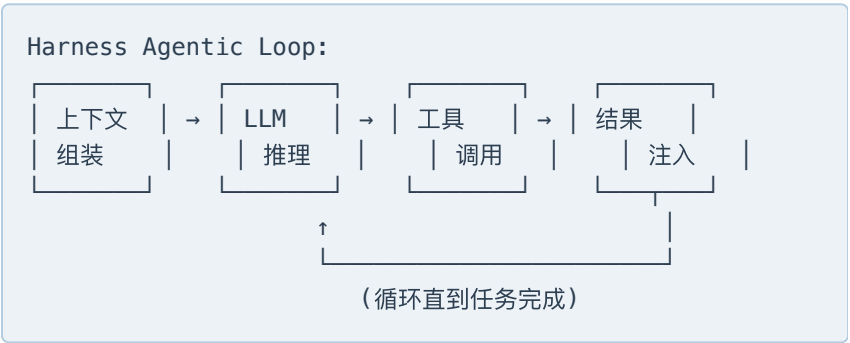
扩展层是 Claude Code 区别于其他 AI 编码工具的关键差异。这里装了三个重量级机制：

- **Skills**：可组合的知识模块，渐进式披露，按需加载
- **MCP (Model Context Protocol)**：外部工具和数据源的标准化接入协议
- **Hooks**：事件驱动的拦截器，在 Agent 生命周期的 17 个节点插入自定义逻辑

这三者配合起来，你能做到的事情远超「改代码」：你可以让 Claude 连接你的内部数据库（MCP），在每次编辑文件前自动运行 lint（Hooks），并在遇到特定技术栈时自动加载该栈的最佳实践（Skills）。

集成层：Agentic Loop 的心脏

集成层是 Harness 的驻地。Harness 不是 Claude 的别称——它是驱动 Agentic Loop 的调度引擎。



每一轮循环包含四个步骤：

1. **上下文组装**：收集 CLAUDE.md 体系规则、当前对话历史、可用工具清单、Skills 的 description 摘要
2. **LLM 推理**：模型根据完整上下文生成下一步行动计划（可能是回复用户，也可能是工具调用）
3. **工具调用**：如果模型决定调用工具，Harness 执行该工具（读写文件、执行命令、搜索 Web 等）
4. **结果注入**：工具执行结果被注入到对话历史，作为下一轮推理的输入

这个循环会持续运转，直到模型认为任务完成（不再产生工具调用）或达到 `max-turns` 上限。

关键设计洞察： Harness 不做「自动拆分任务」。你让它改一个文件，它就不会自作主张重构整个项目。Agentic Loop 的每一步都是有边界的——模型的工具调用必须显式声明，Harness 不会替模型做决定。这个设计哲学贯穿 Claude Code 的整个产品：**给予 Agent 强大的能力，但让人类保持控制权。**

编程层：Agent 的「手和脚」

编程层是 Agent 改变世界的能力集。Claude Code 内置了一套完整的工具链：

表 1–1

工具类别	代表工具	能力
文件读写	Read, Write, Edit (patch)	精准代码修改，基于 fuzzy matching 的 9 策略查找
命令执行	BashTool	运行任意 shell 命令，支持 PTY 交互模式
搜索	Grep, Glob	ripgrep 后端，亚秒级全仓库搜索
Web	WebSearch, WebFetch	联网搜索和页面抓取
子智能体	Task	派生子智能体执行独立子任务
用户交互	AskUserQuestion	向用户提问获取澄清
结构化输出	TodoWrite	任务跟踪和状态管理

注意：这些工具不是简单的命令行封装。比如 `EditTool` 的 patch 模式使用了 9 种 fuzzy matching 策略来处理缩进和空白差异——这是从数百万次代码编辑中沉淀下来的工程经验，不是你写个 `sed` 能比的。

1.3 多端统一引擎

前面说过，Claude Code 的核心引擎是跨端的。但「统一」不是「完全相同」——每个端有自己的交互特征，引擎在不同端上呈现不同的「profile」。

表 1–2

端	交互模型	适用场景	特有优势
CLI	终端会话，stdin/stdout	CI/CD、远程服务器、脚本集成	Headless 模式，零 UI 依赖
Desktop	GUI 应用，内嵌终端	日常开发，沉浸式工作	视觉反馈（图片、diff 预览）
VS Code	IDE 插件	编辑器内协作	与 VS Code 生态集成
JetBrains	IDE 插件	Java/Kotlin 等 JetBrains 技术栈	IntelliJ 平台能力
Web	浏览器	轻量使用、移动端	零安装

不管你在哪个端发起对话，背后都是同一个 Harness 引擎、同一套 CLAUDE.md 体系、同一个 Skills 库。这种架构带来了一个巨大的工程优势：**你在 CLI 上调试好的 Agent 行为，到了 VS Code 里完全一致。**

1.4 实战：从零搭建 Claude Code 工作环境

理论讲完，我们动手。一个工程化的 Claude Code 项目不是 `claude` 回车就开始，而应该有一个清晰的项目结构：

```
my-project/
├── CLAUDE.md              # 项目级记忆（提交到 Git）
├── .claude/
│   ├── skills/            # 项目专属 Skills
│   │   ├── react-patterns.md
│   │   └── api-design.md
│   ├── hooks/             # 项目 Hooks（可选）
│   └── settings.json      # 项目级设置
├── .gitignore             # 确保不提交敏感记忆
└── src/                   # 你的源代码
```

CLAUDE.md 最小可用模板：

```
# 项目记忆

<span style="font-size:0.5pt;">§TOC0-5§</span>
## 技术栈
- 前端：React 18 + TypeScript + Vite
- 后端：Node.js 20 + Express
- 数据库：PostgreSQL 15

<span style="font-size:0.5pt;">§TOC0-6§</span>
## 编码规范
- 使用函数组件 + Hooks，不使用 Class 组件
- 类型定义放在 `types/` 目录
- API 路由统一前缀 `/api/v1`

<span style="font-size:0.5pt;">§TOC0-7§</span>
## 项目约束
- Node.js 版本 >= 20.0.0
- 所有 API 必须返回 JSON 格式
- 单元测试覆盖率 > 80%
```

存好这个文件后，打开终端，跑一次 `claude`。你会发现 Claude 已经「知道」你用的是 React + Node.js + PostgreSQL，不再需要每次对话开头解释一遍技术栈。

这就是记忆层在起作用。而这只是第 2 章要深入的内容。

1.5 Harness Agentic Loop 深度剖析

理解了四层架构的静态结构，我们需要深入动态机制——Harness 到底如何驱动 Agent 的每一次决策？

5.1 一轮推理的微观过程

让我们跟踪一次典型的工具调用全过程：

```
[用户输入] "帮我在 src/utils/format.ts 里加一个 formatPhoneNu

Harness 上下文组装阶段 (Context Assembly):
├─ 从五层记忆体系收集 CLAUDE.md 规则
├─ 加载可用 Skills 的 description 列表
├─ 加载可用工具的函数签名清单
├─ 附加当前对话历史（含之前的工具调用结果）
└─ 拼装为完整的 system prompt + user message
    → 总 token: ~3500

LLM 推理阶段 (LLM Inference):
├─ 模型分析任务：需要读取文件 → 理解现有代码 → 编写新函数
├─ 生成第一个工具调用：Read(file_path="src/utils/format.ts")
└─ 注意：模型只声明工具调用，不执行

Harness 工具执行阶段 (Tool Execution):
├─ 解析模型输出的工具调用
├─ 匹配到 ReadTool
├─ 读取 src/utils/format.ts
├─ 将文件内容封装为 tool_result
└─ tool_result 注入到对话历史

[第二轮]
Harness 上下文组装:
├─ 所有 CLAUDE.md + Skills 再次注入（注意：这是每次必做的）
├─ 对话历史：用户消息 + Read 调用 + Read 结果
└─ 总 token: ~4200（增加了文件内容）

LLM 推理:
├─ 分析 format.ts 的代码风格和现有函数模式
├─ 生成 Edit 调用：在文件末尾插入 formatPhoneNumber 函数
└─ 使用 fuzzy matching 策略确定插入位置

Harness 工具执行:
```

```
├─ EditTool 使用 patch 模式
├─ 9 种 fuzzy matching 策略依次尝试，找到最佳匹配位置
├─ 执行编辑
└─ tool_result 注入对话历史

[第三轮]
LLM 推理:
├─ 检查编辑结果
├─ 认为任务完成
└─ 无工具调用，循环终止
```

关键洞察： Harness 的每一轮都**重新注入** CLAUDE.md 和 Skills description。这意味着你在对话中途修改 CLAUDE.md，下一轮推理就会生效。但也意味着这些固定上下文在每一轮都消耗 token——这再次印证了第 2 章要深入讲的「CLAUDE.md 精简要义」。

5.2 Harness 的边界控制

Harness 不会无限循环。它有三道防线：

表 1–3

防线	默认值	作用	配置方式
`max-turns`	25	单个对话的最大推理轮次	`--max-turns 50` 或 settings.json
Token 预算感知	模型上下文窗口	接近上下文上限时，Harness 主动压缩/截断	自动管理
无工具调用终止	N/A	模型连续两轮不调用工具，循环自动终止	内置行为

实战中曾遇到的问题： 一个复杂的重构任务，Claude 在第 23 轮时被 `max-turns` 截断，导致一些文件修改了一半。解决方案不是无脑调大 `max-turns`，而是： 1. 用子智能体（第 4 章）将大任务拆分 2. 在 CLAUDE.md 中要求 Claude 在任务过半时主动汇报进度 3. 对于确实复杂的大任务，分多次对话完成

5.3 Harness 的权限模型

Harness 对工具调用有一层权限控制。不是所有工具调用都会自动执行：

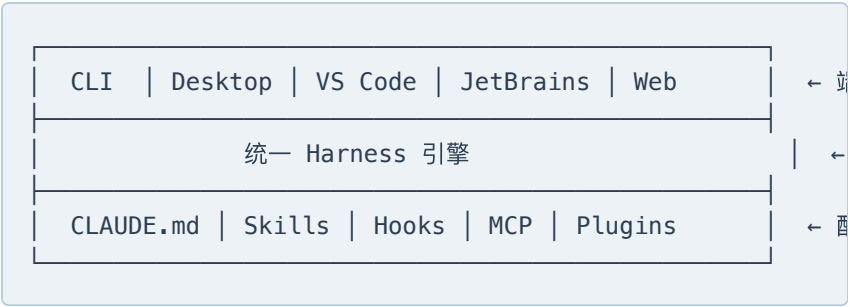
```
工具调用 → Harness 权限检查
├─ 白名单工具（Read, Grep, Glob） → 自动执行
├─ 灰名单工具（Edit, Write, Bash*） → 根据权限模式决定
│   ├─ default mode: 首次使用需用户确认，后续自动
│   ├─ acceptEdits mode: 编辑自动，命令需确认
│   ├─ bypassPermissions mode: 全部自动（CI/CD 等信任环境）
│   └─ plan mode: 全部阻止，只展示计划
└─ 黑名单（无） → Claude Code 没有硬编码的黑名单工具
```

老K的权限建议： – 本地开发：`default` 模式，在安全和效率间取平衡 – CI/CD：`bypassPermissions`，但必须配合 Hooks 安全防线（见第 5 章） – 代码审查：`plan` 模式，让 Claude 只分析不给方案 – 新手学习：`acceptEdits` 模式，减少打断但保留命令确认

1.6 多端架构的工程统一性

前面用表格对比了各端差异，但真正值得理解的是**统一性如何实现**。

6.1 引擎分层与端的关系



端层只负责三件事： 1. **输入捕获**：接收用户消息（CLI 的 stdin、Desktop 的聊天框、IDE 的内联对话框） 2. **输出渲染**：展示 Agent 响应（CLI 的 stdout、Desktop 的富文本、IDE 的代码 diff） 3. **文件访问**：提供对工作目录的读写能力

引擎层完全独立于端，不感知自己被哪个端调用。这意味着你在 CLI 上调试好的 CLAUDE.md、Skills、Hooks，拿到 VS Code 插件里行为完全一致——因为跑的是同一个引擎。

6.2 多端协作的实战场景

```
场景：一个全栈功能的开发日

早上 9:00  VS Code 端：
  └ 在编辑器中打开项目，用 Claude Code 插件生成初始代码框架

上午 10:30  CLI 端：
  └ 在终端中运行 claude，让 Agent 执行批量重构
  └ 利用 CLI 的 headless 模式自动处理 14 个文件的模式替换

下午 2:00  Desktop 端：
  └ 打开 Desktop 应用，与 Claude 进行交互式调试
  └ 利用视觉反馈能力查看图表和架构图

下午 4:00  JetBrains 端：
  └ 在 IntelliJ 中打开后端代码，利用 Claude Code 插件优化 Jav

全天   同一个引擎 + 同一套 CLAUDE.md + 同一组 Skills
```

关键收益： 你不需要为每个端重新配置 Claude——一次配置，处处生效。这是平台化思维的核心价值。

1.7 架构决策指南

表 1–4

问题	解决的层	方法
Claude 不了解代码规范	记忆层	写入 CLAUDE.md
Claude 缺少领域知识	扩展层（Skills）	编写 Skill 文件
需要在 Git commit 前检查	扩展层（Hooks）	配置 PreToolUse hook
需要连接外部 API	扩展层（MCP）	编写 MCP Server
工具调用顺序/流程控制	集成层	调整 max-turns, permission 规则
需要新工具能力	编程层	通过 MCP 扩展工具

老 K 的硬核经验： 大部分团队的问题是「在错误的层解决」。看到 Claude 写代码风格不对，不去改 CLAUDE.md，而是每次手动纠正——这是最浪费 token 的做法。记住：**有规律的行为，放到记忆层；有边界的知识，放到扩展层；一次性操作，保留在对话中。**

1.6 本章小结

Claude Code 的四层架构不是纸上谈兵——它是 Claude Code 团队从数百万次 Agent 交互中抽象出来的稳定模型。理解这四层，你就理解了 Claude Code 的「操作系统」。接下来每一章，我们都会深入一层：

- 第 2 章：深入记忆层，掌握 CLAUDE.md 工程化
- 第 3 章：深入扩展层的核心——Skills 知识工程
- 第 4 章：探索子智能体与 Agent Teams 的协作模式
- 第 5 章：用 Hooks 编织 Agent 生命周期的事件网络

现在，让我们打开记忆层的大门。

第2章 Harness与Agentic Loop

> *Claude Code不是一个问答机器*
>
> —— 它是一个在底层不断'思考–行动–观察'的Looper。"

【决策者摘要】

Harness不是Claude的别名，它是驱动Agentic Loop的调度引擎。本章是全书最底层的技术章节——深入剖析Harness的三阶段循环（上下文组装→LLM推理→工具执行）、边界控制的三道防线、权限模型的四种模式、Token经济学的微观运作，以及Loop Engineer在Harness层面能做的优化。读完本章，你对Claude Code的理解从”一个黑盒工具”升级为”一个可工程化的Loop引擎”。

2.1 Harness不是Claude

很多人的直觉：Claude Code里的Claude就是那个回答问题的AI。这个直觉对了一半——Claude（大模型）只负责推理和生成工具调用。真正驱动整个流程的，是**Harness**。

Harness是Claude Code的调度引擎。它的角色不是”回答问题”，而是**管理Agentic Loop**——不断循环执行”组装上下文→调用LLM→执行工具”这个过程，直到任务完成或达到边界条件。

用一个比喻来区分三者的角色：

表 2–1

组件	比喻	职责
Harness（调度引擎）	工头	管理Loop流程、权限检查、Token预算
LLM（Claude模型）	老师傅	推理、分析、生成工具调用决策
Tools（Bash/Edit/Write等）	工具箱	执行具体操作

Harness不做决策。 它只管理流程。决策永远是LLM做的。这个分工是Claude Code工程化设计的基石——流程控制和智能推理解耦。

2.2 Agentic Loop的三阶段微观剖析

让我们跟踪一次典型的工具调用全过程：

```
[用户输入] "帮我在 src/utils/format.ts 里加一个 formatPhoneNu

━━ 第一轮 ━━

【阶段1: 上下文组装 (Context Assembly)】
Harness 收集:
├─ 五层记忆体系中的 CLAUDE.md 规则
├─ 可用 Skills 的 description 列表
├─ 可用工具的函数签名清单
├─ 当前对话历史（含之前的工具调用结果）
└─ 拼装为完整的 system prompt + user message
    → 总 token: ~3500

【阶段2: LLM推理 (LLM Inference)】
模型分析任务:
├─ 需要读取文件 → 理解现有代码 → 编写新函数
├─ 生成工具调用: Read(file_path="src/utils/format.ts")
└─ 注意: 模型只声明工具调用，不执行

【阶段3: 工具执行 (Tool Execution)】
Harness 执行:
├─ 解析模型输出的工具调用
├─ 匹配到 ReadTool
├─ 读取 src/utils/format.ts
├─ 将文件内容封装为 tool_result
└─ tool_result 注入到对话历史 → 进入下一轮

━━ 第二轮 ━━

【阶段1: 上下文组装】
├─ 所有 CLAUDE.md 再次注入（每轮必做！）
├─ 对话历史: 用户消息 + Read调用 + Read结果
└─ 总 token: ~4200（增加了文件内容）
```

```
【阶段2: LLM推理】
├─ 分析 format.ts 代码风格和现有函数模式
├─ 生成 Edit 调用: 在文件末尾插入 formatPhoneNumber 函数
└─ 使用 fuzzy matching 策略确定插入位置

【阶段3: 工具执行】
├─ EditTool 使用9种fuzzy matching策略定位
├─ 执行编辑
└─ tool_result 注入对话历史 → 进入下一轮

─── 第三轮 ───

LLM推理:
├─ 检查编辑结果
├─ 认为任务完成
└─ 无工具调用 → 循环终止
```

核心洞察

1. 上下文是每轮重新注入的。 这不是bug，是设计。这意味着你在对话中途修改CLAUDE.md，下一轮推理就会生效。但代价是——这些固定上下文在每一轮都消耗token。

2. LLM不执行任何操作。 模型只输出工具调用的JSON声明。真正读写文件的是Harness。这个解耦让权限模型可以在Harness层面独立运作——模型永远不知道哪些操作被允许、哪些被阻止。

3. 每一轮都是独立决策。 Harness不替LLM拆分任务。你让它改一个文件，它不会自作主张重构整个项目。

2.3 边界控制：Loop不会无限循环

Harness有三道防线防止无限循环或失控：

表 2–2

防线	默认值	作用	配置方式
`max-turns`	25	单个对话最大推理轮次	`--max-turns 50`
Token预算感知	上下文窗口	接近上限时自动压缩/截断	自动管理
无工具调用终止	N/A	连续两轮不调用工具，自动终止	内置行为

一个翻车案例： 重构2000行遗留代码时，Claude在第23轮被`max-turns`截断，文件改了一半。解决方案不是调大`max-turns`——那样token消耗翻倍——而是：

1. 用子智能体拆分大任务（每个子任务在各自上下文窗口内完成）
2. 在CLAUDE.md中要求Claude主动汇报进度
3. 分多次对话完成复杂任务

Loop Engineer的铁律：别把max-turns当性能参数调。它是你的最后防线。

2.4 权限模型：在Harness层面守护安全

Harness对每次工具调用都有权限检查。不是所有工具调用都会自动执行：

```
工具调用 → Harness 权限检查
├─ 读操作（Read, Grep, Glob）  → 自动执行
├─ 写操作（Edit, Write）       → 根据权限模式决定
├─ 命令执行（Bash）            → 根据权限模式决定
└─ 网络操作（WebSearch等）     → 根据权限模式决定
```

四种权限模式：

表 2–3

模式	读操作	写操作	命令	网络	适用场景
`default`	自动	首次确认	首次确认	需确认	本地开发
`acceptEdits`	自动	自动	需确认	需确认	受控环境
`bypassPermissions`	自动	自动	自动	自动	CI/CD沙箱
`plan`	自动	阻止	阻止	阻止	代码审查

老K的建议： – 本地开发：`default` — 平衡安全和效率 – CI/CD：`bypassPermissions` — 但必须配合Hooks安全防线 – 代码审查：`plan` — 让Claude只分析不给方案 – 新手学习：`acceptEdits` — 减少打断但保留命令确认

2.5 Loop Engineer的优化空间

Loop Engineer的工作不是在Harness外面看着它跑——而是主动设计Loop的每个环节。

上下文预算管理

每一轮推理，以下内容都会消耗Token：

表 2–4

内容	每轮成本	优化方向
CLAUDE.md	~500–2000 tokens	精简到关键规则，用渐进式Skills代替
Skills descriptions	~200–800 tokens	控制description字数，降低触发频率
对话历史	逐轮增长	长任务拆分为子对话
工具调用结果	可变	引导LLM生成精准的工具调用

Loop性能调优清单

- [ ] CLAUDE.md 是否精简到每行都有明确决策价值?
- [ ] Skills 的 description 是否精确控制在50字以内?
- [ ] 复杂任务是否拆分成了子对话/子智能体?
- [ ] 是否有定期清理长对话历史的策略?
- [ ] 是否设定了合理的 max-turns 值?

2.6 本章小结

1. Harness是Claude Code的调度引擎，驱动Agentic Loop三阶段循环
2. 上下文每轮重新注入是设计选择——允许动态更新，但需管理token成本
3. 三道防线（max-turns、Token预算、无工具终止）防止Loop失控
4. 四种权限模式对应不同的安全场景
5. Loop Engineer的核心工作是上下文预算管理和Loop性能优化

📚 延伸阅读

- **Claude Code Architecture (官方文档)** — Agentic Loop的实现细节
- **第3章 记忆系统** — CLAUDE.md如何参与上下文组装
- **第5章 子智能体** — 用子对话拆分Loop，降低单轮复杂度

思考题

1. 如果max-turns设置为30，一个对话进行了28轮，此时Claude请求执行一个需要5轮的复杂操作——Harness应该允许还是拒绝？作为Loop Engineer你会怎么设计这个边界？
2. 设计一个场景，在default权限模式下，如何用Hooks（第6章）在不升级到bypassPermissions的情况下实现CI/CD自动化？

第3章 记忆系统：CLAUDE.md 工程实践

2.1 五层记忆体系

Claude Code 的记忆不是一个大杂烩文件。它是一个有明确层级和继承关系的五层体系：

```
优先级从低到高 ↓   （高优先级覆盖低优先级）

第1层  AUTO.md            — 自动生成，Claude 自我总结的对话记忆
第2层  CLAUDE.md          — 项目根目录，团队共享的公共规则（提交到
第3层  .claude/CLAUDE.md— 本地覆盖，个人加入但不提交的规则（.gi
第4层  项目级记忆          — 通过 /memory 命令写入的项目持久记忆
第5层  个人级记忆          — 跨项目的用户全局记忆（~/.claude/CLAU
```

各层详解

第1层：AUTO.md

AUTO.md 是 Claude Code 最容易被忽视但最智能的一层。它不是人写的——是 Claude 在对话中自动生成的「自总结」。当你完成一个较长的任务后，Claude 可能会在 `~/.claude/auto.md` 或项目目录下写入一个 AUTO.md，记录本次对话中的关键发现。

```
# AUTO.md（自动生成示例）
- 用户偏好使用 pnpm 而非 npm
- 项目使用 Vitest 做测试，配置文件在 vitest.config.ts
- API base URL 是 https://api.example.com/v2
```

关键特征： AUTO.md 的生命周期管理由 Claude 自行决定。这是一层「自适应记忆」——它会随着你的使用自动演化。

第2层：CLAUDE.md（项目根目录）

这是团队共享的项目记忆。提交到 Git，所有团队成员共用。

第3层：.claude/CLAUDE.md（本地覆盖）

这是你个人的「小抄」，不会被 Git 跟踪。比如你可能在这里写：

```
# 个人偏好
- 我的本地数据库端口是 5433（不同于团队默认的 5432）
- 我的 npm registry 使用公司内部镜像
```

第4层：项目级记忆

通过 `/memory` 斜杠命令写入，存储在 Claude Code 内部。适合记录需要持久但不想污染 Git 的项目上下文。

第5层：个人级记忆（~/.claude/CLAUDE.md）

这是你的「全局身份」。适合放跨项目的个人偏好：

```
# 老王的工作偏好
- 代码注释使用中文
- 变量命名偏好驼峰式（camelCase）
- 错误处理喜欢 early return 模式
- 测试框架偏好 Vitest（前端）和 Jest（后端）
```

老K的忠告： 个人级记忆要谨慎使用。写太多全局规则会让每个项目都受你个人偏好的约束，这可能导致在不是你主导的项目中出现意外的行为。原则是：**全局规则只放「你是谁」，项目规则放「项目是什么」。**

2.2 记忆合并机制

当一次对话开始时，Harness 会从五个层级中收集记忆并按优先级合并。规则很简单：

1. 从第1层开始收集，逐层向上
2. 高层级覆盖低层级相同 key 的规则
3. 非冲突的规则全部保留（additive merge）
4. 最终生成的 system prompt 是合并后的结果

```
合并示意：

第5层(个人)：  "使用 pnpm" + "注释用英文"
第2层(项目)：  "使用 yarn" + "缩进用2空格"
第3层(本地)：  "缩进用4空格"   ← 个人本地覆盖

最终结果：  "使用 yarn" + "注释用英文" + "缩进用4空格"
          （项目yarn覆盖了个人pnpm，本地缩进覆盖了项目缩进）
```

工程实践： 如果你发现自己经常在某些项目中需要大幅覆盖 CLAUDE.md 的内容，那说明你的项目级 CLAUDE.md 写得不够好，或者在错误的位置写了规则。优先把团队共识放在第2层，个

人偏好放在第3层。

2.3 条件化规则系统

Claude Code 的 CLAUDE.md 支持条件化规则——你不必把所有规则一股脑全注入。通过条件判断，你可以让不同情境加载不同的规则。

基本语法

条件化规则使用 `<!-- if: condition -->` 和 `<!-- endif -->` 标记（这是 Claude Code 2025 年底引入的特性）。Harness 在解析 CLAUDE.md 时会对条件求值，只注入满足条件的段落。

```
# 全局规则（总是生效）
- 所有 API 返回 JSON 格式

<!-- if: language == "typescript" -->
<span style="font-size:0.5pt;">§TOC2-4§</span>
## TypeScript 规则
- 使用 `as` 语法做类型断言，不用 `<Type>` 语法
- 启用 strict mode
- 避免使用 `any`
<!-- endif -->

<!-- if: framework == "react" -->
<span style="font-size:0.5pt;">§TOC2-5§</span>
## React 规则
- 使用函数组件 + Hooks
- 状态管理优先使用 useState/useReducer
- 避免过早引入状态管理库
<!-- endif -->

<!-- if: context contains("testing") -->
<span style="font-size:0.5pt;">§TOC2-6§</span>
## 测试规则
- 测试文件命名：`*.test.ts`
- 使用 describe/it 模式
- 每个 describe 不超过 5 个测试用例
<!-- endif -->
```

条件变量参考

表 3–1

条件变量	含义	示例
`language`	当前编辑文件的语言	`language == "python"`
`framework`	检测到的框架	`framework == "react"`
`context`	对话上下文关键词	`context contains("debug")`
`file_path`	当前文件路径匹配	`file_path matches "src/api/**"`
`task`	当前任务类型	`task == "refactor"`

老K的经验： 条件化规则最实用的场景有三个：

1. **多语言仓库**：一个仓库同时有 Python 和 TypeScript 代码，不同子目录适用不同规则
2. **任务模式切换**：开发、测试、重构三种模式下需要不同级别的检查
3. **渐进式约束**：新手开发者和资深开发者的规则严格程度不同（通过本地覆盖调整）

2.4 实战案例：三种典型项目配置

案例一：React 前端项目

项目结构：

```
react-dashboard/
├── CLAUDE.md
├── .claude/
│   └── CLAUDE.md   （本地覆盖，gitignore）
├── src/
│   ├── components/
│   ├── hooks/
│   ├── types/
│   └── utils/
└── package.json
```

CLAUDE.md（项目级，提交 Git）：

```
# React Dashboard — 项目记忆

<span style="font-size:0.5pt;">§TOC2-8§</span>
## 技术栈
- React 18 + TypeScript 5
- 构建工具：Vite 5
- 状态管理：Zustand
- 样式方案：TailwindCSS 3
- 测试：Vitest + React Testing Library
- 包管理：pnpm

<span style="font-size:0.5pt;">§TOC2-9§</span>
## 组件规范
- 每个组件一个文件，组件名即文件名（PascalCase）
- 组件文件内结构：imports → types → component → exports
- 使用 React.memo() 包裹纯展示组件
- Props 接口命名为 `{ComponentName}Props`

<!-- if: file_path matches "src/components/**" -->
<span style="font-size:0.5pt;">§TOC2-10§</span>
## 组件目录结构
- 复杂组件使用文件夹：`Button/Button.tsx + Button.test.tsx +
- 简单组件使用单文件：`Spinner.tsx`
- 导出统一使用 barrel export（index.ts）
<!-- endif -->

<!-- if: task == "testing" -->
<span style="font-size:0.5pt;">§TOC2-11§</span>
## 测试策略
- 组件测试优先测试用户交互，而非实现细节
- 使用 screen.getByRole 而非 getByTestId
- 异步操作使用 waitFor 和 findBy*
<!-- endif -->
```

```
<span style="font-size:0.5pt;">§TOC2-12§</span>
## 代码风格
- 函数声明：箭头函数（const 声明）
- 解构 Props：在函数参数中直接解构
- Hooks 调用顺序：useState → useEffect → 自定义 hooks
- 不要在 JSX 中写复杂逻辑，提取为函数或变量
```

.claude/CLAUDE.md（本地覆盖，不提交）：

```
# 个人本地覆盖

<span style="font-size:0.5pt;">§TOC2-13§</span>
## 开发环境
- 本地 API 代理端口：3000
- mock 数据服务器：`pnpm mock` 启动在 3001

<span style="font-size:0.5pt;">§TOC2-14§</span>
## 个人偏好
- 我喜欢在组件顶部加 JSDoc 注释描述用途
- 调试时优先使用 console.table 而非 console.log
```

案例二：Node.js 后端项目

CLAUDE.md（项目级）：

```
# API Server — 项目记忆

<span style="font-size:0.5pt;">§TOC2-15§</span>
## 技术栈
- Runtime: Node.js 20 LTS
- 框架: Express 4 + TypeScript
- 数据库: PostgreSQL 15 + Prisma ORM
- 缓存: Redis 7
- 消息队列: BullMQ
- 测试: Jest + Supertest

<span style="font-size:0.5pt;">§TOC2-16§</span>
## 架构约束
- 分层架构：Controller → Service → Repository
- Controller 只做参数校验和响应格式化，不写业务逻辑
- Service 层写所有业务逻辑，可跨 Repository 调用
- Repository 层只管数据访问，不包含业务判断

<!-- if: file_path matches "src/controllers/**" -->
<span style="font-size:0.5pt;">§TOC2-17§</span>
## Controller 规范
- 使用 express-async-errors 统一处理异步错误
- 响应格式统一为 `{ code: number, data: T, message: string
- 参数校验使用 Zod schema，在 controller 入口完成
<!-- endif -->

<!-- if: file_path matches "src/services/**" -->
<span style="font-size:0.5pt;">§TOC2-18§</span>
## Service 规范
- 每个 Service 方法必须显式返回类型
- 事务操作必须通过 Prisma 的 $transaction
- 外部 API 调用必须有超时设置（默认 5s）和重试（最多 2 次）
<!-- endif -->
```

```
<span style="font-size:0.5pt;">§TOC2-19§</span>
## 数据库规范
- 迁移文件只通过 Prisma Migrate 生成，不手动编辑
- 查询默认包含软删除过滤（deletedAt IS NULL）
- N+1 查询：关联数据使用 Prisma 的 include/select

<span style="font-size:0.5pt;">§TOC2-20§</span>
## 错误处理
- 所有 try-catch 的 catch 块必须做两件事：log + 抛出标准错误
- 自定义错误类继承 AppError（含 code + httpStatus）
- 禁止吞掉错误（bare catch）
```

案例三：Python 数据科学项目

CLAUDE.md（项目级）：

```
# ML Pipeline — 项目记忆

<span style="font-size:0.5pt;">§TOC2-21§</span>
## 技术栈
- Python 3.11
- 数据处理：Pandas 2 + Polars（新代码迁移中）
- ML 框架：scikit-learn 1.5 + XGBoost 2
- 实验追踪：MLflow
- 环境管理：Poetry（依赖）+ pyenv（Python 版本）
- 代码质量：Ruff（lint）+ MyPy（type check）

<span style="font-size:0.5pt;">§TOC2-22§</span>
## Python 规范
- 类型注解：所有函数参数和返回值必须有类型注解
- 文档字符串：公共函数使用 Google style docstring
- 导入顺序：stdlib → third-party → local，每组之间空一行

<!-- if: context contains("notebook") -->
<span style="font-size:0.5pt;">§TOC2-23§</span>
## Notebook 规范
- 每个 Notebook 的第一个 cell 是 markdown，描述目的和输入输出
- Notebook 不包含超过 50 行的 cell
- 完成后执行 Kernel → Restart & Run All 确保可复现
<!-- endif -->

<!-- if: file_path matches "src/features/**" -->
<span style="font-size:0.5pt;">§TOC2-24§</span>
## Feature Engineering 规范
- 特征工程函数必须是无状态的纯函数
- 输入输出都是 Pandas DataFrame，不依赖全局状态
- 函数名使用 `build_` 前缀（build_date_features, build_text_
<!-- endif -->

<span style="font-size:0.5pt;">§TOC2-25§</span>
```

```
## 数据处理最佳实践
- 链式调用优先于逐步赋值（df.pipe().assign().query()）
- 避免 inplace=True 操作
- 使用 .loc[] 和 .iloc[]，不用链式索引 df[col][row]
- 缺失值处理必须在 Pipeline 中显式声明策略

<span style="font-size:0.5pt;">§TOC2-26§</span>
## 实验管理
- 每次实验记录参数、指标、数据版本到 MLflow
- 实验命名：`{日期}_{简短描述}_{关键参数}`
- 废弃代码不删除，移动到 `archive/` 目录
```

2.5 CLAUDE.md 工程化清单

写好 CLAUDE.md 不难，写好一套**工程化的** CLAUDE.md 体系需要纪律。以下是老 K 从数十个项目里总结的清单：

表 3–2

检查项	要求	反模式
项目级是「公约」	只写团队共识，不写个人偏好	「小明喜欢用 tab 缩进」
本地覆盖是「例外」	只写项目规则的例外和本地环境差异	在本地覆盖里重写整个项目规则
条件化减少噪音	不同场景的规则用条件隔离	所有规则无差别注入，token 浪费
有具体版本号	技术栈声明含主版本号	「使用最新版本」——下周就变了
有反例	关键规则写明「不要做什么」	只有正面约束，缺少禁止项
能跑通	规则中引用的命令在当前环境下可执行	`npm test` 但项目用的是 `pnpm test`

一个残酷的事实： 90% 的项目 CLAUDE.md 的问题是「太泛了」。写「使用最佳实践」「注重代码质量」「保证性能」——这些话对 AI 来说等于没说。你需要的是具体的、可验证的、可执行的约束。

```
# 差
- 注重代码性能

# 好
- 避免在 render 中创建新对象/函数引用
- List 渲染必须使用 key prop（禁止使用 index 作为 key）
- 超过 100 项的数据使用虚拟滚动（react-window）
```

2.6 记忆系统的「暗面」——你需要知道的问题

讲了这么多好处，也要诚实地说说问题。

问题一：记忆膨胀。 随着项目发展，CLAUDE.md 往往会变成「垃圾桶」——什么都往里扔。一个 500 行的 CLAUDE.md 比没有更糟糕，因为关键信息被噪音淹没了。解决方法是定期 review，把过时的规则删除，把仍在使用的规则用条件隔离。

问题二：冲突不可见。 五层合并机制很强大，但也带来了「静的冲突」——你的本地覆盖和团队规则冲突时，没有人会提醒你。唯一的防御是保持本地覆盖的最小化。

问题三：Token 成本。 每一层记忆都消耗 token。一个 200 行的 CLAUDE.md 加上 100 行的 Skills description，再加上对话历史和工具描述——一轮对话的 system prompt 轻松超过 2000 token。如果不加节制，你会花大量 token 在「提醒 Claude 你是谁」而不是「让 Claude 帮你写代码」。

老K的建议： 把 CLAUDE.md 当作代码来维护。每个季度做一次「记忆清理」——检查哪些规则还在生效、哪些已经过时、哪些应该从项目级下沉到本地级。你的 CLAUDE.md 应该是一个活文档，不是一块石碑。

2.7 本章小结

CLAUDE.md 五层记忆体系是 Claude Code 的基石。它不是一次配置终身受益——它需要像代码一样迭代和维护。

三条最核心的原则：

1. **分层明确**：全局身份 → 项目公约 → 本地例外，每层只放该放的东西
2. **条件化隔离**：用 `<!-- if -->` 让规则按场景加载，减少无关上下文噪音
3. **具体可验证**：每条规则都能在 code review 中被客观验证，不存在「尽量」「最好」这种模糊词

下一章，我们将进入扩展层的核心——Skills 知识工程。你会发现 CLI 已更新到 `deepseek-v4-pro`，而 Skills 的「渐进式披露」设计才是让 Claude Code 真正理解你的业务的关键。

第4章 Skills 工程：知识的两个维度

3.1 CLAUDE.md vs Skills：知识的两个维度

很多开发者一开始会混淆 CLAUDE.md 和 Skills——「不都是告诉 Claude 一些信息吗？」这个混淆会让你写出错误的 Skill。

表 4–1

维度	CLAUDE.md	Skills
知识类型	规则（Rules）	知识（Knowledge）
回答的问题	「做什么 / 不做什么」	「怎么做」
加载时机	每次对话必定加载	按需激活（渐进式披露）
内容特征	约束、偏好、禁令	流程、模式、最佳实践
存放位置	项目根 / ~/.claude/	.claude/skills/
Token 成本	固定开销	按需开销

一个生动的类比：

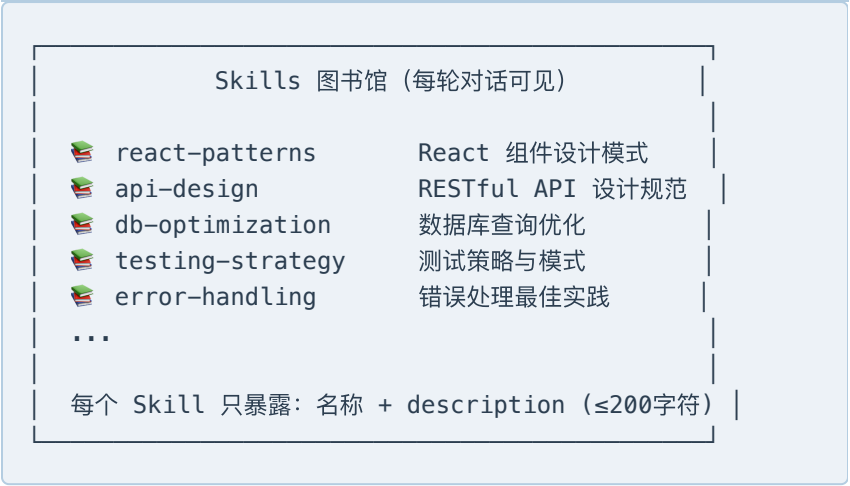
CLAUDE.md 是驾驶手册——「红灯停、绿灯行、不要超速」。Skills 是导航地图——「从这里到目的地，有 A/B/C 三条路线，A 最近但山路多，B 稍远但全程高速。」

你不会每次开车都翻一遍全国地图（Skills），但你每次开车都要遵守交通规则（CLAUDE.md）。这就是为什么 CLAUDE.md 每次必加载，而 Skills 是按需激活。

老K的判断标准： 如果一个知识块满足这三个条件，它适合做 Skill： 1. 只在特定场景（特定技术栈、特定任务类型）下有用 2. 内容超过 50 行，全量加载太浪费 token 3. 有独立的知识边界，可以作为一个完整的「知识单元」

3.2 渐进式披露：图书馆模型

Claude Code 的 Skills 系统采用了一个精妙的设计：**渐进式披露（Progressive Disclosure）**。这是一种「图书馆」模型——用户进入图书馆时，只能看到每本书的书名和简介（description）。只有当用户对某本书产生兴趣，才会翻开它（激活 Skill），看到完整内容。



每次对话开始，Harness 会收集所有可用 Skills 的名称和 description，注入到 system prompt 中。Claude 看到的是一个「目录」，而不是所有 Skill 的完整内容——这就是 token 节省的核心机制。

当 Claude 判断某个 Skill 与当前任务相关时，它会「打开这本书」——Skill 的完整内容（包括详细指令、代码示例、流程图）被注入到对话上下文中，Claude 获得该领域的深度知识。

实例对比

全量注入模式（没有渐进式披露）：

```
[System Prompt]
... CLAUDE.md (200行) ...
... react-patterns.md (150行) ...
... api-design.md (180行) ...
... db-optimization.md (200行) ...
... testing-strategy.md (160行) ...
... error-handling.md (120行) ...
总 token 消耗：~5000+ token，其中大部分与当前任务无关
```

渐进式披露模式：

```
[System Prompt]
... CLAUDE.md (200行) ...
可用 Skills：
- react-patterns: React 组件设计模式与最佳实践
- api-design: RESTful API 设计规范与命名约定
- db-optimization: 常见数据库查询性能优化方法
- testing-strategy: 前端与后端测试策略选择指南
- error-handling: 错误处理模式与异常传播策略
总 token 消耗（目录）：~3500 token

当 Claude 判断需要 react-patterns 时：
[注入 react-patterns.md 完整内容：150 行]
额外 token 消耗：~1500 token
但只在需要时发生！
```

Token 节省效果：

表 4–2

场景	全量注入	渐进式披露	节省比例
React 组件开发（1个Skill激活）	5000 token	5000 token	0%
API 设计（1个Skill激活）	5000 token	5000 token	0%
通用重构（0个Skill激活）	5000 token	3500 token	**30%**
全栈开发（3个Skill激活）	5000 token	8000 token	–60%（但提供更多知识）

注意：当多个 Skill 被激活时，渐进式披露的 token 消耗反而更多——但这意味着 Claude 获得了对应每个任务的专业知识，这些 token 花得值。

3.3 Description 预算机制

前面提到每个 Skill 的 description 控制在 200 字符以内。这不是一个建议——它是一个「预算」。你要在这个预算内完成三件事：

1. **告诉 Claude 这个 Skill 是关于什么的**
2. **告诉 Claude 什么时候应该用这个 Skill**

3. **暗示这个 Skill 的价值**——为什么值得打开它

Description 的工程写法

```
# 差：太泛，Claude 不知道什么时候激活
description: "React 开发相关的最佳实践"

# 差：太啰嗦，浪费了 token 却没有更多信息量
description: "本 Skill 包含了我们团队在多年 React 开发中积累的
包括组件设计模式、状态管理策略、性能优化技巧以及测试方法等等"

# 好：精准、有边界、含触发信号
description: "React 组件设计模式。当创建新组件或重构现有组件时激
含组合模式、Render Props、HOC、自定义 Hooks 的选择决策树。"
```

Description 设计原则：

表 4–3

原则	含义	反例
名字即分类	Skill 文件名清晰表达知识域	`stuff.md`，`utils.md`
描述含触发词	写入 Claude 判断时用的关键场景	「当需要时使用」——太泛
暗示收益	让 Claude 看到「打开的好处」	只描述内容，不说价值
不超过 200 字符	严格控制长度	250 字符——超预算了

老K的实战技巧： 写 description 的时候，把自己想象成一个在图书馆里挑书的读者。你站在书架前，每本书只有 5 秒时间吸引你的注意。你不会拿起一本标题模糊、简介空洞的书。Claude 也是一样——如果你的 description 不够精准，这个 Skill 就永远不会被激活。

3.4 双通道激活机制

Claude Code 的 Skills 有两条激活路径：

通道一：自动激活（Implicit Activation）

Claude 根据 description 和对话上下文自动判断是否需要激活某个 Skill。这是渐进式披露的默认工作方式。

```
用户: "帮我创建一个用户登录表单组件"
          ↓
Claude 判断: 创建 React 组件 → 需要 react-patterns Skill
          ↓
Claude: skill_view(name='react-patterns')
          ↓
系统注入 react-patterns.md 全部内容
          ↓
Claude 基于 Skill 知识开始编码
```

通道二：显式激活（Explicit Activation）

用户在对话中通过斜杠命令或自然语言直接要求加载某个 Skill。

```
用户: "用 api-design skill 帮我设计用户模块的 API"
或
用户: /api-design
```

两种通道的协同：

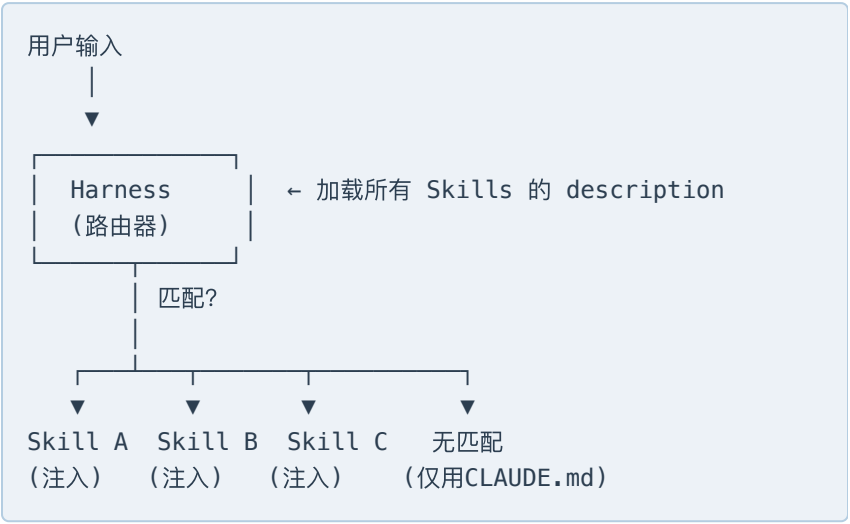
表 4–4

通道	触发方	适用场景	优点	缺点
自动激活	Claude	日常开发，Claude 自己判断	用户无感，流畅	可能遗漏或错误激活
显式激活	用户	特定需求，精确控制	准确，可控	用户需要知道 Skill 存在

工程建议： 设计 Skill 时假设它主要通过「自动激活」被使用。如果某个 Skill 频繁需要用户显式指定才能激活，说明你的 description 写得不够好，Claude 无法自动判断激活时机。

3.5 路由器思维

这是理解 Skills 系统的关键思维方式：**把 Skills 当作知识路由器，而不是知识仓库。**



好的 Skill 体系像一个设计精良的路由表：

- 每个 Skill 覆盖一个独立的路由域（不重叠）
- Description 是路由规则（匹配条件）
- Skill 正文是路由目标（处理逻辑）
- 同一个请求可以命中多个路由（多 Skill 同时激活）

路由器设计原则：

1. **互不重叠**：两个 Skill 的知识域不应该重叠。如果 `react-patterns` 和 `component-design` 都讲组件设计，你该合并它们。

2. **覆盖完整**：你频繁遇到的知识场景应该都有对应的 Skill 覆盖。如果经常手动纠正 Claude 的 API 设计，你需要一个 `api-design` Skill。
3. **粒度适中**：一个 Skill 应该是「一次对话中可能用到的完整知识单元」。太大（500+ 行）该拆分，太小（20 行）不值得独立成 Skill，放进 CLAUDE.md 即可。

3.6 500 行法则

这是老 K 从实践中总结的一个硬性约束：**单个 Skill 文件不能超过 500 行。**

这不是一个随意的数字，而是基于三个工程现实：

为什么是 500 行?

表 4–5

原因	解释
上下文窗口效率	500 行的 Skill 大约消耗 3000–5000 token。加上对话历史和其他上下文，仍在一个健康的范围内
认知负担	超过 500 行的 Skill，Claude 容易迷失在细节中，无法有效提取关键信息
维护可行性	人写的文档，超过 500 行就很难保持结构清晰和内容最新

超过 500 行怎么办?

答案是**拆分**，但要按正确的维度拆分：

```
# 差：随机拆分
api-design-part1.md  (350行)
api-design-part2.md  (320行)
→ Claude 不知道什么时候激活哪个，路由混乱

# 好：按子域拆分
api-rest-design.md       (400行) — RESTful API 设计
api-graphql-design.md    (350行) — GraphQL schema 设计
api-error-handling.md    (200行) — 错误码和异常处理
api-versioning.md        (180行) — API 版本管理策略
→ 每个子域独立路由，各自有精准的 description
```

拆分决策树

```
Skill 超过 500 行?
  ├─ 是 → 是否有自然子域分界?
  │         ├─ 是 → 按子域拆分
  │         └─ 否 → 内容是否包含大量代码示例?
  │                   ├─ 是 → 精简要义，代码示例精简到最小可用
  │                   └─ 否 → 考虑是否部分内容更适合 CLAUDE.md
  └─ 否 → 保持现状
```

3.7 allowed-tools 权限模型

Skills 可以声明自己需要哪些工具的权限。这是一个容易被忽视但影响深远的特性。

```
---
name: db-migration
description: 数据库迁移操作指南。当需要创建、运行或回滚数据库迁移时
allowed-tools: Bash(git:*), Bash(prisma:*), Read, Write
---
```

权限声明的三种策略

表 4-6

策略	allowed-tools 配置	适用场景
最小权限	只声明必需的几个工具	数据库迁移、部署脚本等危险操作 Skill
适度权限	声明常用工具，给一定自由度	大部分开发 Skill
开放权限	不声明（继承默认权限）	纯知识型 Skill（无副作用操作）

```
# 最小权限示例：数据库迁移 Skill
---
allowed-tools: Bash(prisma:migrate, prisma:generate), Rea
---

# 适度权限示例：测试 Skill
---
allowed-tools: Bash(npm:test, pnpm:test, jest:*), Read, G
---

# 开放权限示例：代码规范 Skill（纯知识，不需要工具）
---
# 不声明 allowed-tools，继承默认权限
---
```

老K的建议： 对于操作型 Skill（会产生副作用——修改数据库、执行部署、操作文件系统），一定要声明最小权限。这不仅是为了安全，更是给 Claude 一个明确的信号：「在这个 Skill 的上下文中，你只能做这些事。」

3.8 Skill 设计的 4 种范式

经过大量实践，我们归纳出 Skill 设计的 4 种核心范式。你的每个 Skill 都应该属于其中一种。

范式一：流程型 Skill（Process Skill）

定义： 描述完成某个任务的完整步骤流程。

特征： – 有明确的步骤序列（1→2→3→4） – 有判断分支（如果 X 则 A, 否则 B） – 有预期的输出产物

示例：API 设计评审 Skill

```
---
name: api-review
description: API 设计评审 checklist。当设计新 API 或修改现有 A
---

# API 设计评审流程

<span style="font-size:0.5pt;">§TOC3-9§</span>
## 步骤 1：路径设计
- 资源名使用复数名词：`/users` 而非 `/user`
- 层级关系体现为路径嵌套：`/users/{id}/orders`
- 非 CRUD 操作使用动词后缀：`/users/{id}/activate`

<span style="font-size:0.5pt;">§TOC3-10§</span>
## 步骤 2：请求设计
- GET 请求参数不超过 5 个，否则考虑 POST + body
- 分页参数统一为 `page` + `page_size`
- 排序参数格式：`sort=field:asc|desc`

<span style="font-size:0.5pt;">§TOC3-11§</span>
## 步骤 3：响应设计
- 成功响应统一包裹：`{ code: 0, data: T, message: "ok" }`
- 列表响应包含分页信息：`{ items: [], total: N, page: N, page
```

范式二：模式型 Skill（Pattern Skill）

定义： 提供可复用的代码模式或设计模式库，让 Claude 在面对特定场景时选择合适模式。

特征： – 模式名称 + 适用场景 – 代码示例（最小可用示例）– 选择决策树（何时用模式 A vs 模式 B）

示例：错误处理模式 Skill

```
---
name: error-patterns
description: 错误处理模式库。含 Result/Option 模式、异常链、降级
---

# 错误处理模式

<span style="font-size:0.5pt;">§TOC3-12§</span>
## 模式选择决策树
```

需要区分「可恢复」和「不可恢复」错误? ├── 是 → Result 模式（推荐用于业务逻辑） └── 否 → 异常模式（推荐用于基础设施层）

调用外部服务可能失败? ├── 是 → 必须加降级策略（fallback） └── 否 → 不需要降级

```
<span style="font-size:0.5pt;">§TOC3-13§</span>
## 模式 1：Result<T, E>（推荐）

```typescript
type Result<T, E = Error> =
  | { success: true; data: T }
  | { success: false; error: E };

// 使用
function divide(a: number, b: number): Result<number> {
  if (b === 0) return { success: false, error: new Error(
  return { success: true, data: a / b };
}
```

适用：业务逻辑层，需要调用方显式处理错误
```

```
### 范式三：决策型 Skill（Decision Skill）

**定义：** 帮助 Claude 在多个可选项之间做出正确的技术决策。

**特征：**
- 明确的选择题（A vs B vs C）
- 决策矩阵（条件→推荐方案）
- 不涉及具体代码实现

**示例：状态管理选型 Skill**

```markdown
---
name: state-management
description: 前端状态管理方案选型指南。当项目中需要选择状态管理方案
---

# 状态管理方案选型

<span style="font-size:0.5pt;">§TOC3-14§</span>
## 决策矩阵

| 条件 | 推荐方案 | 原因 |
|------|---------|------|
| 组件局部状态，不需共享 | useState | 最简单，零依赖 |
| 少量跨组件共享状态 | useContext + useReducer | 原生方案，轻量
| 中型应用，复杂状态逻辑 | Zustand | 轻量、TS 友好、无 boilerpl
| 大型应用，多人协作 | Redux Toolkit | 规范化结构、DevTools、中
| 服务端状态为主 | TanStack Query | 专注异步状态、缓存、同步 |

<span style="font-size:0.5pt;">§TOC3-15§</span>
```
```

```
## 硬性约束（不参与决策，直接执行）
- 新项目默认使用 Zustand（除非明确需要 Redux）
- 禁止同时使用两个状态管理库
```

范式四：参考型 Skill（Reference Skill）

定义： 提供 Claude 完成任务所需的参考数据——不是规则，不是流程，纯粹的参考资料。

特征： – 大量结构化数据（表格、清单） – 不需要「怎么做」，只需「是什么」 – 通常被其他三种范式 Skill 引用

示例：公司技术栈清单 Skill

```
---
name: tech-stack
description: 公司技术栈清单。含各项目使用的框架、版本和服务地址。当
---

# 公司技术栈

<span style="font-size:0.5pt;">§TOC3-16§</span>
## 项目清单

| 项目 | 前端 | 后端 | 数据库 | 部署 |
|------|------|------|--------|------|
| user-center | React 18 | Node.js 20 | PostgreSQL 15 | K
| payment-svc | - | Go 1.21 | MySQL 8.0 | K8s |
| admin-dashboard | Vue 3 | Java 17/Spring | PostgreSQL 1

<span style="font-size:0.5pt;">§TOC3-17§</span>
## 服务地址

| 服务 | 开发环境 | 测试环境 | 生产环境 |
|------|---------|---------|---------|
| API Gateway | localhost:8080 | api-test.example.com | a
| Auth Service | localhost:8081 | auth-test.example.com |
```

3.9 Skills 体系的实践检验

设计完 Skill 体系后，用以下问题做自检：

1. **覆盖度**：我频繁遇到的知识场景是否都有对应的 Skill？

2. **路由精度**：Claude 是否能根据 description 准确判断激活时机？（观察 10 次对话，看看激活正确率）
3. **去重**：有没有两个 Skill 讲同一件事？有没有 Skill 和 CLAUDE.md 的内容重叠？
4. **500 行红线**：所有 Skill 都在 500 行以内吗？
5. **权限正确**：操作型 Skill 是否声明了最小权限？

老K的血泪教训： 不要在项目第一天就试图设计完美的 Skill 体系。正确的做法是：先用一周 Claude Code 裸奔（只有 CLAUDE.md），记录下你每次手动纠正 Claude 的场景。一周后，统计 Top 5 高频纠正项，为它们各写一个 Skill。第二周再观察，再补充。**Skills 是被需求「逼」出来的，不是提前设计出来的。**

3.10 本章小结

Skills 和 CLAUDE.md 代表了知识的两个维度——规则和知识。理解这个区别，你就能正确地分配内容。

Skills 工程的四个核心原则：

1. **渐进式披露**：让 Claude 在需要时才看到完整内容，用 description 做路由
2. **500 行红线**：单个 Skill 不超过 500 行，超过就按子域拆分
3. **权限最小化**：操作型 Skill 声明必要的工具权限

4. **4 种范式选一**：每个 Skill 明确属于流程型、模式型、决策型或参考型

下一章，我们将进入 Agent 协作的世界——子智能体与 Agent Teams。你会发现，当单个 Claude 不够用时，如何用 Agent Teams 组建一个 AI 开发团队。

第5章 子智能体与 Agent Teams：多 Agent 协作实战

4.1 为什么需要子智能体?

先建立正确的认知：子智能体不是「辅助工具」，而是**独立的工作单元**。每一个子智能体拥有：

- 独立的对话上下文（不污染主对话）
- 独立的任务目标（单一职责）
- 独立的工具权限（可做权限收窄）
- 独立的 Token 计费（可精确核算）

```
主对话 (Main Agent)
│
├── 子智能体 A: "扫描所有遗留的 console.log"
│   └── 结果: 找到 23 处，已生成修复建议
│
├── 子智能体 B: "检查所有 API 路由的错误处理"
│   └── 结果: 3 个路由缺少 try-catch
│
└── 子智能体 C: "生成 CHANGELOG.md"
    └── 结果: 已生成
```

什么时候用子智能体?

表 5–1

场景	直接用主 Agent	用于子智能体	原因
改一个文件	✅	❌	简单任务，开子智能体 overhead 太大
搜索 + 报告	✅	✅	搜索可并行时用于子智能体
跨多个文件重构	❌	✅	主对话上下文会膨胀，子智能体隔离
全仓库代码审查	❌	✅	任务量大，子智能体并行执行
多步骤部署流程	❌	✅	每一步有独立上下文，失败隔离

老K的判断标准： 如果一个任务满足以下三个条件中的两个，开子智能体： 1. 任务耗时超过 30 秒 2. 会产生大量中间输出（污染主对话上下文） 3. 可以和其他任务并行执行

4.2 5 种子智能体使用模式

模式 1：只读模式（Read-only Agent）

定义： 子智能体只做搜索、分析、报告，不修改任何文件。

Token 特征： 低成本。只读操作不产生副作用，可以放心地给较大范围的任务。

实战场景： 代码审查、代码搜索、依赖分析、安全检查。

```
主 Agent: "在合并 PR 之前，做一次全面的代码审查"
    │
    ├── 子智能体 1（只读）：审查 src/components/ 目录
    │   搜索：未使用的导入、console.log、TODO 注释、any 类型
    │   返回：发现 5 个问题（3 个 console.log，2 个 any 类型）
    │
    ├── 子智能体 2（只读）：审查 src/utils/ 目录
    │   返回：发现 2 个问题（1 个未使用函数，1 个缺失类型）
    │
    └── 子智能体 3（只读）：审查 package.json + 依赖
        返回：发现 1 个问题（lodash 可替换为原生方法）
```

Claude Code 实现方式：

```
在对话中：
> 派生一个只读子智能体，扫描 src/ 下所有 .tsx 文件，
> 检查是否有未使用的 import 和 console.log，汇报结果
```

子智能体会使用 Read、Grep 工具搜索代码，但 **不写任何文件**。完成后将结果返回给主对话。

关键参数指定：

- `--permission-mode readonly`：禁止子智能体写入文件
- 或通过 CLAUDE.md 在子智能体上下文中声明只读规则

模式 2：执行模式（Execution Agent）

定义： 子智能体执行一个有明确输入输出的任务，完成后返回结果。

Token 特征： 中等成本。有明确的边界，不会无限扩展。

实战场景： 生成代码、修复 bug、重构一个模块、生成测试。

```
主 Agent: "重构 UserService，将业务逻辑从 Controller 移到 Ser
    │
    └── 子智能体（执行）：
        输入：src/controllers/userController.ts + src/serv
        任务：重构
        输出：修改后的两个文件 + 修改说明
        约束：不改变 API 行为，不修改测试文件
```

最佳实践：

```
# 好：任务边界清晰
"将 utils/formatDate.ts 中的所有函数改为 TypeScript，添加完整类

# 差：任务边界模糊
"优化 utils 目录下的代码质量"
# — 什么是「优化」？什么算「质量」？子智能体会迷失方向
```

执行模式的任务定义模板：

```
任务：[一句话描述]
输入文件：[明确的文件路径列表]
约束条件：
- [约束1：不修改什么]
- [约束2：遵循什么规范]
- [约束3：边界条件]
期望输出：
- [产物1]
- [产物2]
```

模式 3：并行模式（Parallel Agents）

定义： 同时派遣多个子智能体执行互不依赖的任务，合并结果。

Token 特征： 总成本 = Σ(各子智能体成本)，但总耗时 = max(各子智能体耗时)。投入多，但速度快。

实战场景： 多模块独立开发、分目录代码审查、批量测试生成。

```
主 Agent: "为 src/ 下每个一级子目录生成单元测试"

┌─ 子智能体 1: 为 src/components/ 生成测试 ──────── 2 min ─┐
├─ 子智能体 2: 为 src/hooks/ 生成测试 ──────────── 1 min ─┤
├─ 子智能体 3: 为 src/utils/ 生成测试 ──────────── 1 min ─┤
└─ 子智能体 4: 为 src/services/ 生成测试 ──────── 3 min ─┘
   并行执行，总耗时 3 min（而非串行的 7 min）

主 Agent 收集结果 → 汇总报告
```

并行模式的硬性约束：

1. 各子任务**绝对不能互相依赖**——如果子智能体 B 需要子智能体 A 的输出，不能并行
2. 各子任务**操作的文件集合必须不重叠**——两个子智能体不能同时编辑同一个文件
3. 子智能体的**权限应该一致**——如果一个需要写权限，另一个也需要

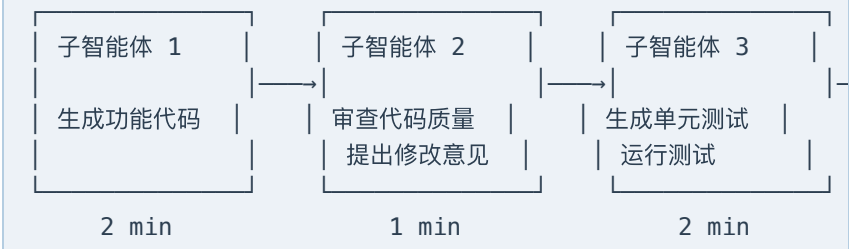
老K的血泪教训： 并行模式下最常见的失败是「文件冲突」——两个子智能体同时修改同一个文件的不同位置，后完成的覆盖了先完成的。避免方法：**按目录划分子任务边界，绝不交叉。**

模式 4：流水线模式（Pipeline Agents）

定义： 子智能体按顺序执行，前一个的输出是后一个的输入。形成处理管线。

Token 特征： 总成本 = Σ(各阶段成本)，但质量更高（每阶段专注一件事）。

实战场景： 代码生成→审查→测试→修复的工作流。



流水线模式示例：API 端点开发全流程

```
阶段 1: 子智能体 A（设计）
  输入: API 需求描述
  任务: 设计路由 + 请求/响应 Schema
  输出: API 设计文档 + OpenAPI spec

阶段 2: 子智能体 B（实现）
  输入: 阶段 1 的设计文档
  任务: 实现 Controller + Service + Repository
  输出: 实现代码 + 迁移文件

阶段 3: 子智能体 C（测试）
  输入: 阶段 2 的实现代码
  任务: 编写集成测试 + 运行测试
  输出: 测试文件 + 测试结果报告

阶段 4: 主 Agent（审查）
  输入: 阶段 1-3 的全部产物
  任务: 最终审查 + 合并
```

流水线模式的关键设计点：

- 每个阶段有**明确的产物格式**——纯文本、JSON、代码文件路径
- 阶段间传递的是**结构化数据**，不是「你去看看上一个做了什么」
- 任一阶段失败，**流水线中止**，不继续执行后续阶段

模式 5：团队模式（Agent Teams）

定义： 多个子智能体同时工作在不同领域，互相通信协调，共同完成一个复杂目标。

Token 特征： 高成本，用于高价值复杂任务。

实战场景： 全栈功能开发（前端 + 后端 + 数据库 + 测试协同）。

```
主 Agent（Team Lead）
  │
  ├── 前端 Agent: "实现用户管理页面"
  │   职责: React 组件 + 状态管理 + API 对接
  │
  ├── 后端 Agent: "实现用户管理 API"
  │   职责: 路由 + Service + Repository + 迁移
  │
  ├── 数据库 Agent: "设计用户相关表结构"
  │   职责: Schema 设计 + 索引优化 + 迁移脚本
  │
  └── 测试 Agent: "端到端测试用户管理流程"
      职责: 集成测试 + E2E 测试
```

团队模式的协调机制：

```
协调点 1: API 契约（前后端 Agent 之间）
  前端 Agent 需要: GET /api/users 返回 User[]
  后端 Agent 产出: 确认该接口的 Request/Response 格式
  协调方式: 主 Agent 传递「契约文档」，双方独立工作但遵守同一契约

协调点 2: 数据库 Schema（后端 ↔ 数据库 Agent 之间）
  数据库 Agent 产出: Prisma schema
  后端 Agent 消费: 基于 schema 生成 Prisma Client 查询
  协调方式: Schema 写好后，后端 Agent 再启动
```

团队模式的适用条件：

- ✅ 任务足够复杂，单一 Agent 无法在合理时间内完成
- ✅ 子任务之间有清晰的接口边界
- ✅ 有足够的 Token 预算（团队模式是成本最高的模式）
- ❌ 任务边界模糊，需要大量跨领域协商

4.3 Agent Teams 深度实战

Teams 配置

Claude Code 支持通过配置文件定义 Agent Teams：

```
// .claude/teams.json
{
  "teams": {
    "fullstack-dev": {
      "description": "全栈开发团队：前端+后端+测试协同",
      "agents": [
        {
          "name": "frontend",
          "role": "React 前端开发",
          "skills": ["react-patterns", "api-design"],
          "permission_mode": "default",
          "workdir": "src/frontend"
        },
        {
          "name": "backend",
          "role": "Node.js 后端开发",
          "skills": ["api-design", "db-optimization"],
          "permission_mode": "default",
          "workdir": "src/backend"
        },
        {
          "name": "qa",
          "role": "测试工程师",
          "skills": ["testing-strategy"],
          "permission_mode": "readonly",
          "workdir": "."
        }
      ]
    }
  }
}
```

实战：用 Agent Teams 完成一个全栈功能

任务： 为博客系统添加「评论」功能（前端 UI + 后端 API + 数据库迁移 + 测试）

主 Agent 的编排逻辑：

```
第一步：数据库 Agent（独立执行）
  → 设计 comments 表，生成 Prisma 迁移
  → 产物：prisma/schema.prisma + 迁移文件

第二步：后端 Agent + 前端 Agent（并行执行）
  后端 Agent：
    → 基于数据库 Schema 实现 CRUD API
    → 产物：controller + service + repository
  前端 Agent：
    → 实现评论列表 + 评论输入组件
    → 产物：React 组件文件

第三步：测试 Agent（依赖前两步）
  → 验证 API 行为 + 组件渲染
  → 产物：测试文件 + 测试报告

第四步：主 Agent 整合审查
  → 检查所有产物
  → 运行 lint + 测试
  → 提供最终报告
```

Token 消耗分析：

表 5-2

阶段	Agent	Token 估算	累计
数据库	db-agent	~3000	3000
后端	backend-agent	~5000	8000
前端	frontend-agent	~5000	13000
测试	qa-agent	~4000	17000
审查	main-agent	~3000	**20000**

总消耗约 20000 token（按 Claude Sonnet 定价约 $0.06）。一个人工开发者完成同样工作大约需要 2-3 小时。这是 Agent Teams 的核心价值：**用极低的金钱成本换取极高的时间效率。**

4.4 Worktrees：会话隔离的工程方案

问题场景

并行 Agent 同时编辑同一个 Git 仓库的不同文件——没有冲突时一切正常。但一旦有交叉依赖（Agent A 改了类型定义，Agent B 在使用旧类型定义），就会出现静默错误：Agent B 的输出在它的上下文中看起来正确，但合并后完全不可用。

解决方案：Git Worktrees

Claude Code 支持为每个子智能体创建独立的 Git worktree：

```
主仓库: /project/main
  │
  ├── Worktree: /project/worktrees/agent-001 (前端 Agent)
  ├── Worktree: /project/worktrees/agent-002 (后端 Agent)
  └── Worktree: /project/worktrees/agent-003 (测试 Agent)
```

每个 worktree 是一个完整的独立工作目录，互不干扰。子智能体在各自的 worktree 中工作，完成后由主 Agent 协调合并。

Worktree 工作流：

```
1. 主 Agent 为每个子 Agent 创建 worktree
   git worktree add /tmp/cc-worktrees/frontend main

2. 子 Agent 在 worktree 中独立工作
   cd /tmp/cc-worktrees/frontend && [修改文件]

3. 子 Agent 完成后，主 Agent 审查变更
   git -C /tmp/cc-worktrees/frontend diff

4. 主 Agent 合并所有 worktree 的变更
   → 逐文件 cherry-pick 或 merge
   → 处理冲突（如有）

5. 清理 worktree
   git worktree remove /tmp/cc-worktrees/frontend
```

Worktree 的使用场景

表 5–3

场景	是否需要 Worktree	原因
只读 Agent	❌ 不需要	不修改文件
单个执行 Agent	❌ 不需要	只有一个 writer
并行 Agent 操作不同目录	⚠️ 建议	安全第一，避免意外冲突
并行 Agent 操作相同目录	✅ 必须	否则必有冲突
流水线 Agent	❌ 不需要	串行执行，天然隔离

老K的实战建议： 刚开始用子智能体时，不需要立即引入 Worktrees。当你第一次遇到两个 Agent 冲突造成的 bug 时，你就会理解 Worktrees 的价值。在此之前，保持简单。

4.5 Token 经济学

使用子智能体的一个重要动机是成本优化。但如果不理解 Token 经济学，你可能花更多钱。

成本模型

表 5–4

模式	串行成本	并行加价	总上下文效率	适用场景
主 Agent 单干	基准 100%	0%	低（上下文膨胀）	简单任务
只读 Agent	80–90%	0%	高（隔离上下文）	搜索/分析
执行 Agent	90–110%	0%	高（专注任务）	明确边界任务
并行 Agent	110–130%	**并行加速**	最高（完全隔离）	多模块独立任务
流水线 Agent	120–150%	0%	高（每阶段专注）	多步骤质量门控
团队模式	150–200%	团队加速	中（需协调通信）	复杂全栈任务

成本优化策略

策略 1：任务合并

```
# 差：三个小任务各开一个子智能体
子智能体 A：改文件1（200 token）
子智能体 B：改文件2（200 token）
子智能体 C：改文件3（300 token）
总：700 + 3×overhead（~600）= 1300 token

# 好：合并为一个任务
子智能体 A：改文件1、2、3（600 token）
总：600 + 1×overhead（~200）= 800 token  ← 节省 38%
```

策略 2：结果摘要而非全文返回

子智能体完成任务后，如果只返回摘要而非完整输出，可以大幅减少主 Agent 上下文的 token 消耗。

```
# 在子智能体任务描述中加入：
"完成后只需返回：修改了哪些文件、关键决策、遇到的问题（不超过 200 字
```

策略 3：权限收窄降低误操作成本

给子智能体更窄的权限，不仅是为了安全，也是为了减少错误操作的重试成本。

成本监控清单

- [ ] 每个子智能体任务完成后，检查 token 消耗（在对话中查看 usage 信息）
- [ ] 对于重复性任务，对比「主 Agent 单干」vs「子智能体」的成本差异
- [ ] 并行模式下，确认**每个子任务确实做到了文件操作不交叉**

- [ ] 流水线模式下，检查中途是否有不必要的上下文传递（每一步不应看到上一步的全部输出）

4.6 本章小结

子智能体和 Agent Teams 将 Claude Code 从「一个 AI 助手」升级为「一个 AI 开发团队」。5 种使用模式覆盖了从简单搜索到全栈协作的整个光谱。

三条核心原则：

1. **模式匹配任务**：不要为了用子智能体而用子智能体。简单任务开子智能体是杀鸡用牛刀——overhead 大于收益。
2. **边界就是一切**：子智能体的任务边界越清晰，成功率越高。模糊的任务描述是子智能体失败的第一大原因。
3. **Token 意识**：子智能体不是免费的午餐。每次使用前问自己：这个任务用主 Agent 直接做，成本会差多少？

下一章，我们将进入 Hooks 的世界——用事件驱动机制在 Agent 生命周期的关键节点织入你的自定义逻辑。你会发现，Hooks 才是让 Claude Code 真正融入你的工程体系的秘密武器。

第6章 Hooks 事件驱动：编织 Agent 生命周期

> ⚠️ *Agent 处理器是**递归的**

—— 一个 Agent 里的 Hook 可能触发另一个 Agent。如果不加限制，可能形成无限循环。务必为 Agent 处理器设置合适的 `permission_mode` 和作用域边界。

5.1 Hooks 的哲学：从「相信 AI」到「验证 AI」

大多数开发者使用 AI 编码工具的模式是这样的：

```
提出需求 → AI 生成代码 → 人工审查 → 修改 → 再审查 → 通过
```

这个流程的问题在于：**责任后置。** 你在 AI 产出之后才开始检查，但此时错误已经进入了文件系统。如果 AI 在生成过程中就触发了 lint 报错，你可能要花几分钟来修复——而如果 lint 在 AI 写入文件之前就阻止了它，这些问题根本不会产生。

Hooks 改变的就是这个时序。

```
有 Hooks:
提出需求 → AI 规划修改 → [Hook: 检查] → ✅ 通过 → AI 写入文件
                                        → ❌ 拒绝 → AI 重新规划
```

Hook 不是「亡羊补牢」——Hook 是「防患于未然」。

5.2 17 个事件类型全景

Claude Code 的 Hooks 系统在 Agent 生命周期的以下 17 个节点提供了拦截点：

事件分类

表 6–1

类别	事件名	触发时机	典型用途
会话	`SessionStart`	对话开始	环境检查、注入项目状态
会话	`SessionEnd`	对话结束	清理临时文件、记录日志
用户输入	`UserPromptSubmit`	用户提交消息	输入审查、敏感信息过滤
用户输入	`PreToolUse`	工具调用**前**	**最常用**——安全检查、lint、验证
用户输入	`PostToolUse`	工具调用**后**	结果验证、自动提交、通知
用户输入	`PostToolUseFailure`	工具调用失败	错误处理、自动重试、告警
权限	`PermissionRequest`	权限申请时	自定义权限规则、审批流
文件	`PreWrite`	写入文件前	代码格式化、安全检查
文件	`PostWrite`	写入文件后	自动 lint、自动格式化
文件	`PreRead`	读取文件前	文件存在性检查

类别	事件名	触发时机	典型用途
命令	`PreCommand`	执行 shell 命令前	命令审查、危险命令拦截
命令	`PostCommand`	执行 shell 命令后	输出解析、结果验证
通知	`Notification`	系统通知事件	集成外部通知系统
停止	`Stop`	Agent 主动停止	清理、状态保存
子智能体	`SubagentStart`	子智能体启动	注入子智能体配置
子智能体	`SubagentStop`	子智能体结束	收集子智能体结果
错误	`Error`	未捕获错误	全局异常处理、告警

事件频率热力图

```
高频（每次对话触发多次）
  ████████ PreToolUse        ← 最核心的事件
  ████████ PostToolUse
  ██████   PreWrite
  █████    PostWrite

中频（每次对话触发 1–2 次）
  ████     SessionStart
  ████     UserPromptSubmit
  ███      PreCommand
  ███      PostCommand

低频（按需触发）
  ██       PermissionRequest
  █        SubagentStart / SubagentStop
  █        PostToolUseFailure
  █        Error / Stop / Notification / SessionEnd
```

5.3 3 种处理器类型

Claude Code 的 Hooks 支持 3 种处理器，各有不同的能力边界和适用场景。

类型对比

表 6–2

处理器类型	说明	能做什么	不能做什么	适用场景
command	执行一个 shell 命令	运行任何 CLI 工具、脚本	不能访问 Claude 内部状态	lint、格式化、安全检查
prompt	向 Claude 注入提示	修改 Claude 的行为/策略	不能执行外部命令	动态指令、上下文补充
agent	启动一个 Agent 子任务	完整的 Agent 能力	开销最大	复杂验证、多步骤检查

Command 处理器

最常用、最高效的处理器类型。Hook 触发时执行一个 shell 命令，根据退出码判断通过/失败。

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "command": "pnpm lint-staged --diff",
        "description": "在编辑文件前运行 lint 检查"
      }
    ]
  }
}
```

Command 处理器的协议：

```
Hook 触发 → 执行 command
  ├─ 退出码 0: 通过，允许操作继续
  ├─ 退出码 1: 警告，显示 stderr 但允许操作继续
  └─ 退出码 2+: 阻止，操作被取消，stderr 反馈给 Claude
```

Command 环境变量：

Hook 执行时可以访问以下环境变量获取上下文：

表 6-3

环境变量	含义	示例值
`CLAUDE_EVENT`	触发的事件名	`PreToolUse`
`CLAUDE_TOOL_NAME`	被调用的工具名	`Edit`，`Write`，`Bash`
`CLAUDE_FILE_PATH`	操作的文件路径（如适用）	`/path/to/file.ts`
`CLAUDE_TOOL_INPUT`	工具的输入参数（JSON）	`{"file_path": "...", ...}`

Prompt 处理器

不执行命令，而是向 Claude 的系统提示中注入文本。适合需要「提醒」而非「阻止」的场景。

```
{
  "hooks": {
    "UserPromptSubmit": [
      {
        "matcher": "*",
        "prompt": "注意：用户最近多次提到性能问题，请在回答中优先
        "description": "上下文感知提示"
      }
    ]
  }
}
```

Prompt vs Command 选择指南：

表 6–4

场景	选 Prompt	选 Command
需要在操作前「提醒」Claude	✅	❌
需要「阻止」操作	❌	✅（退出码 2）
需要运行外部工具	❌	✅
需要根据操作结果动态决策	❌	✅
纯文本补充信息	✅	❌

Agent 处理器

最强大的处理器——启动一个完整的 Agent 子任务来处理 Hook 触发的检查。

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit",
        "agent": {
          "prompt": "检查即将编辑的文件。如果修改涉及安全敏感代码
          "permission_mode": "readonly"
        },
        "description": "安全 Agent 审查"
      }
    ]
  }
}
```

Agent 处理器的使用警告：

5.4 实战：安全防护 5 道防线

这是 Hooks 最核心的实战场景——用事件驱动机制构建 Agent 行为的安全防护网。

防线 1：危险命令拦截 (PreCommand)

```
{
  "hooks": {
    "PreCommand": [
      {
        "matcher": "*",
        "command": "danger-check.sh",
        "description": "拦截危险命令"
      }
    ]
  }
}
```

danger-check.sh：

```
#!/bin/bash
# 读取 Claude 要执行的命令
COMMAND="$CLAUDE_TOOL_INPUT"

# 黑名单检查
DANGEROUS_PATTERNS=(
  "rm -rf /"
  "git push --force origin main"
  "DROP TABLE"
  "DROP DATABASE"
  "> /dev/sda"
  "chmod 777"
  "curl.*|.*sh"
)

for pattern in "${DANGEROUS_PATTERNS[@]}"; do
  if echo "$COMMAND" | grep -qi "$pattern"; then
    echo "❌ 拦截危险命令: $pattern" >&2
    echo "命令已被安全策略阻止。如需执行，请手动操作。" >&2
    exit 2  # 阻止执行
  fi
done

exit 0  # 安全，允许执行
```

防线 2：敏感文件保护 (PreWrite / PreRead)

```
{
  "hooks": {
    "PreWrite": [
      {
        "matcher": "*.env|*.pem|*.key|credentials.*|secre
        "command": "sensitive-file-guard.sh",
        "description": "防止 AI 修改敏感文件"
      }
    ],
    "PreRead": [
      {
        "matcher": "*.env|*.pem|*.key",
        "command": "secret-read-guard.sh",
        "description": "防止 AI 读取密钥文件"
      }
    ]
  }
}
```

sensitive-file-guard.sh：

```
#!/bin/bash
FILE="$CLAUDE_FILE_PATH"

echo "⚠️ 文件 $FILE 被标记为敏感文件。" >&2
echo "AI 不能修改密钥、证书和凭证文件。请手动操作。" >&2
exit 2
```

防线 3：操作日志审计 (PostToolUse)

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write|Bash",
        "command": "audit-log.sh",
        "description": "记录所有写操作和命令执行"
      }
    ]
  }
}
```

audit-log.sh：

```
#!/bin/bash
LOG_FILE=".claude/audit.log"

echo "[$(date -Iseconds)] EVENT=$CLAUDE_EVENT TOOL=$CLAUD
```

防线 4：权限申请审批 (PermissionRequest)

```
{
  "hooks": {
    "PermissionRequest": [
      {
        "matcher": "Bash*",
        "command": "permission-approval.sh",
        "description": "命令执行需要额外审批"
      }
    ]
  }
}
```

防线 5：全局异常告警 (Error)

```
{
  "hooks": {
    "Error": [
      {
        "matcher": "*",
        "command": "error-alert.sh",
        "description": "异常发生时发送告警"
      }
    ]
  }
}
```

error-alert.sh：

```
#!/bin/bash
# 发送 Slack/飞书/钉钉 通知
curl -X POST "$WEBHOOK_URL" \
  -H "Content-Type: application/json" \
  -d "{\"text\": \"⚠️ Claude Code 异常: $CLAUDE_EVENT\"}"
```

5.5 实战：代码质量 4 道门

Hooks 在代码质量方面同样可以建立严密的防线。

门 1：写入前 lint (PreWrite)

```
{
  "hooks": {
    "PreWrite": [
      {
        "matcher": "*.ts|*.tsx",
        "command": "npx eslint --stdin --stdin-filename \
        "description": "TypeScript 文件写入前 ESLint 检查"
      },
      {
        "matcher": "*.py",
        "command": "ruff check --stdin-filename \"$CLAUDE
        "description": "Python 文件写入前 Ruff 检查"
      }
    ]
  }
}
```

协议说明： lint 工具从 stdin 读取 Claude 要写入的内容，根据 lint 结果： – `exit 0`: 通过，写入文件 – `exit 1+`: 失败，阻止写入，将 lint 错误反馈给 Claude 让其修复

门 2：写入后格式化 (PostWrite)

```
{
  "hooks": {
    "PostWrite": [
      {
        "matcher": "*.ts|*.tsx|*.js|*.json|*.md",
        "command": "npx prettier --write \"$CLAUDE_FILE_P
        "description": "文件写入后自动格式化"
      }
    ]
  }
}
```

注意：PostWrite 的格式化结果如果与 Claude 写入的内容不同，**Claude 不会感知到这个差异**（因为 PostWrite 发生在工具调用完成后）。因此，格式化 Hook 应该是幂等的格式化操作，而非可能引入语义变化的操作。

门 3：类型检查 (PreToolUse)

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit",
        "command": "typecheck-guard.sh",
        "description": "编辑前运行类型检查作为基线"
      }
    ]
  }
}
```

typecheck-guard.sh：

```
#!/bin/bash
# 在编辑前运行类型检查，确保当前状态是干净的
npx tsc --noEmit 2>&1 | head -20
exit_code=$?

if [ $exit_code -ne 0 ]; then
  echo "⚠️ 当前代码存在类型错误，请先修复再继续编辑。" >&2
  exit 1  # 警告而非阻止
fi
exit 0
```

门 4：测试守护 (PostWrite)

```
{
  "hooks": {
    "PostWrite": [
      {
        "matcher": "src/**/*.ts",
        "command": "related-test-check.sh",
        "description": "检查修改的文件是否有对应测试"
      }
    ]
  }
}
```

related-test-check.sh：

```
#!/bin/bash
FILE="$CLAUDE_FILE_PATH"
TEST_FILE="${FILE/src\//src/__tests__/}"
TEST_FILE="${TEST_FILE/.ts/.test.ts}"

if [ ! -f "$TEST_FILE" ]; then
  echo "⚠️ 文件 $FILE 没有对应的测试文件 $TEST_FILE" >&2
  echo "请考虑添加测试。" >&2
  exit 1  # 警告，不阻止
fi
exit 0
```

5.6 Hooks 配置的最佳实践

配置文件位置

```
项目级 Hooks:
  .claude/settings.json    → 提交到 Git，团队共享

个人级 Hooks:
  ~/.claude/settings.json → 个人偏好，不提交
```

层级合并规则： 与 CLAUDE.md 类似，个人级和项目级 Hooks 会合并。相同 matcher 的 Hook，个人级优先。

完整配置示例

`.claude/settings.json`（推荐的项目级配置）：

```
{
  "hooks": {
    "SessionStart": [
      {
        "matcher": "*",
        "command": "node .claude/hooks/session-start.js",
        "description": "检查开发环境是否就绪"
      }
    ],
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "command": ".claude/hooks/pre-write-check.sh",
        "description": "写入前安全检查 + lint"
      }
    ],
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "command": ".claude/hooks/auto-format.sh",
        "description": "自动格式化"
      },
      {
        "matcher": "Edit|Write|Bash",
        "command": ".claude/hooks/audit-log.sh",
        "description": "审计日志"
      }
    ],
    "PreCommand": [
      {
        "matcher": "*",
        "command": ".claude/hooks/danger-check.sh",
        "description": "危险命令拦截"
      }
```

```
    ],
    "Error": [
      {
        "matcher": "*",
        "command": ".claude/hooks/error-notify.sh",
        "description": "异常告警"
      }
    ]
  }
}
```

配置性能注意事项

Hooks 是**同步阻塞**的——Hook 执行期间，Agent 完全等待。这意味着：

表 6-5

注意事项	说明	建议
执行时间	Hook 耗时直接增加用户等待时间	单个 Hook < 5 秒
非必要不 Hook	每个 Hook 都会降低 Agent 响应速度	只为关键节点配置 Hook
避免重操作	不要在 Hook 中做 npm install、Docker build	重操作用 PostToolUse + 异步
退出码协议	Command 处理器必须遵守退出码协议	0=通过, 1=警告, 2=阻止

老K的配置哲学： Hooks 是「少而精」的艺术。一个精心设计的 PreWrite lint Hook 胜过 10 个不痛不痒的 PostWrite 通知 Hook。优先把预算花在「阻止问题」而不是「报告问题」上。

5.7 调试 Hooks

Hooks 的执行是「幕后的」，用户通常看不到 Hook 的输出。当 Hook 行为不符合预期时，调试可能会比较困难。

调试策略

策略 1：查看 Hook 日志

```
# Claude Code 将 Hook 执行日志写入
cat ~/.claude/logs/hooks.log
```

策略 2：在 Hook 脚本中输出调试信息

```
#!/bin/bash
# 所有 stderr 输出会显示给用户（当 Hook 失败时）
echo "[DEBUG] Hook triggered: $CLAUDE_EVENT" >> /tmp/clau
echo "[DEBUG] Tool: $CLAUDE_TOOL_NAME" >> /tmp/claude-hoo
echo "[DEBUG] File: $CLAUDE_FILE_PATH" >> /tmp/claude-hoo

# 实际检查逻辑...

# Hook 通过
echo "[DEBUG] Hook passed" >> /tmp/claude-hook-debug.log
exit 0
```

策略 3：dry-run 模式

在 Hook 脚本中加一个 dry-run 模式，用于手动测试：

```
#!/bin/bash
FILE="$CLAUDE_FILE_PATH"

# dry-run 模式：手动测试时使用
if [ "$1" = "--dry-run" ]; then
  echo "Dry-run mode: would check $FILE"
  exit 0
fi

# 实际检查逻辑...
```

5.8 Hooks 反模式

知道不该做什么，和知道该做什么一样重要。

反模式 1：Hook 地狱

```
// ❌ 不要这样
{
  "hooks": {
    "PreToolUse": [
      { "command": "check1.sh" },
      { "command": "check2.sh" },
      { "command": "check3.sh" },
      { "command": "check4.sh" },
      { "command": "check5.sh" }
    ]
  }
}
// 每次工具调用前执行 5 个 Hook → 响应延迟 10+ 秒
```

正确做法： 合并到一个脚本中：

```
#!/bin/bash
# all-checks.sh — 一次执行所有检查
check_lint
check_types
check_security
check_format
```

反模式 2：Hook 中修改 Agent 的输出

```
// ❌ PostWrite Hook 中执行 sed 替换
{
  "PostWrite": [
    {
      "command": "sed -i 's/var/let/g' \"$CLAUDE_FILE_PAT
    }
  ]
}
// 问题：Claude 不知道文件被修改了，产生不一致
```

正确做法： PostWrite 只做格式化（Prettier、autopep8），不做语义修改。语义修改应在 PreWrite 阶段通过 lint 报错的方式阻止。

反模式 3：Hook 无限循环

```
// ❌ PostWrite 中触发文件修改，又触发 PostWrite...
{
  "PostWrite": [
    {
      "command": "npx prettier --write \"$CLAUDE_FILE_PAT
    }
  ]
}
```

正确做法： 使用 PreWrite（阻止写入）或确保 PostWrite 的操作是幂等的。Claude Code 有限循环检测机制（同一事件的同一 Hook 不重复触发），但不应依赖于此。

5.9 本章小结

Hooks 事件驱动系统是让 Claude Code 从「AI 工具」进化到「工程体系」的关键一步。它不是锦上添花——在严肃的生产环境中，Hooks 是**必需品**，不是**可选项**。

核心要点：

表 6–6

要点	具体内容
最常用的 3 个事件	PreToolUse（质量门控）、PreCommand（安全拦截）、PostWrite（自动格式化）
处理器选择	Command 优先（快速、可控）、Prompt 辅助（提示注入）、Agent 兜底（复杂检查）
安全 5 道防线	危险命令拦截、敏感文件保护、操作审计、权限审批、异常告警
质量 4 道门	写入前 lint、写入后格式化、类型检查基线、测试覆盖守护
配置原则	少而精，阻塞优于报告，单脚本优于多 Hook

老K的最后一句话： 如果你只用 Hooks 做一件事，那就是 PreToolUse + lint。在一个 50 人的团队里，这一个 Hook 每天可以阻止上百次「AI 写了不符合团队规范的代码」——回报率远

超任何其他 Hooks 投入。

第7章 MCP协议：连接AI与外部世界的桥梁

决策者摘要

MCP（Model Context Protocol）是Anthropic定义的一套开放协议，它解决了AI Agent最根本的瓶颈：**模型只能推理，不能行动**。通过MCP，Claude Code可以查询数据库、调用API、操作文件系统、控制外部设备——把大模型从”聊天机器人”升级为”数字员工”。

核心决策点：

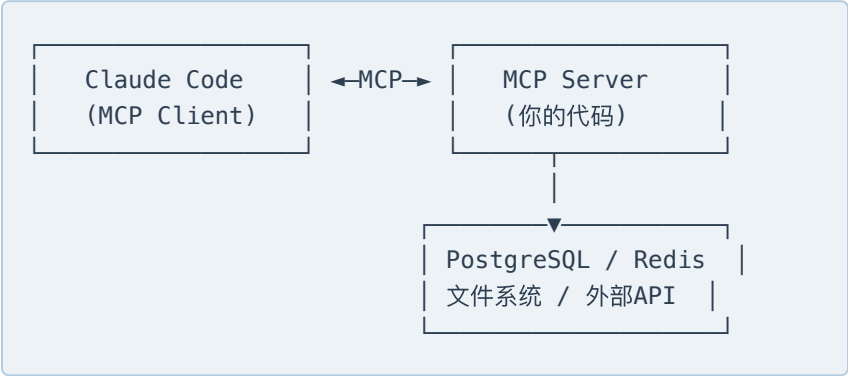
表 7–1

决策项	建议	理由
传输方式选择	开发用stdio，生产用HTTP+SSE	stdio零配置，HTTP支持远程和负载均衡
自建还是用社区MCP	先搜索社区MCP市场，找不到再自建	避免重复造轮子
一个MCP服务器管多少资源	按领域拆分，一个服务器一个职责	跟微服务同理，减少耦合
安全策略	三层纵深防御，缺一不可	单一防线被突破后全盘沦陷

读完本章你会掌握：三种传输方式的选型标准和实战配置、从零构建数据库查询MCP服务器、三层纵深安全机制的设计与落地、MCP与Skills协同作战的工程模式。

6.1 MCP协议的核心架构

MCP的架构极其简洁：**客户端–服务器模型**。Claude Code作为MCP客户端，通过标准化协议与MCP服务器通信。每个MCP服务器暴露一组能力——可以是资源（Resources）、工具（Tools）、提示模板（Prompts）。



协议本身基于JSON-RPC 2.0，所有消息都是结构化的JSON。客户端发起请求（Request），服务器返回响应（Response），服务器也可以主动发通知（Notification）。

老K的经验之谈：**MCP的本质是"能力外包"**。不要把业务逻辑写进MCP服务器——它只应该是一个薄薄的适配层，把外部系统的接口翻译成MCP协议。真正的业务逻辑放在Skills或Agent的prompt里。

6.2 三种传输方式：选型与实战

MCP支持三种传输方式，各有适用场景。老K用一个表格给你讲清楚：

表 7–2

维度	stdio	HTTP + SSE	WebSocket
配置复杂度	★☆☆☆☆ 零配置	★★★☆☆ 需要端口和路由	★★★★☆ 需要WebSocket框架
适用场景	本地开发、CLI工具、单机部署	远程访问、多客户端、微服务	双向实时推送、状态流
性能	最高（进程内通信）	中等（HTTP开销）	最高（长连接复用）
安全	依赖OS进程隔离	可加TLS + 认证	可加WSS + Token
Claude Code支持	✅ 原生	✅ 原生	⚠️ 需额外配置
调试难度	低（stdout/stderr）	中（需抓包工具）	高（帧级调试）

6.2.1 stdio模式

这是最简单的模式。Claude Code启动MCP服务器进程，通过标准输入输出通信。

claude_desktop_config.json 配置：

```
{
  "mcpServers": {
    "my-db-server": {
      "command": "python",
      "args": ["-m", "my_mcp_db_server"],
      "env": {
        "DATABASE_URL": "postgresql://localhost:5432/mydb
      }
    }
  }
}
```

Python MCP服务器骨架：

```
# server.py — stdio模式MCP服务器骨架
import asyncio
from mcp.server import Server
from mcp.server.stdio import stdio_server

server = Server("my-db-server")

@server.list_tools()
async def list_tools():
    return [
        {"name": "query_db", "description": "执行SQL查询"}
        {"name": "list_tables", "description": "列出所有表"
    ]

@server.call_tool()
async def call_tool(name: str, arguments: dict):
    if name == "query_db":
        return await execute_query(arguments["sql"])
    # ...

async def main():
    async with stdio_server() as (read, write):
        await server.run(read, write, server.create_initi

asyncio.run(main())
```

老K的坑：stdio模式下千万别在服务器里写 `print()` 调试——stdout 被协议占用了，你的 print 会破坏 JSON-RPC 消息结构。调试用 `logging` 模块输出到 stderr。

6.2.2 HTTP + SSE模式

stdio只能本地用。如果你要把MCP服务器部署到远程机器，或者你需要多个Claude Code实例共享一个MCP服务器，HTTP模式是正解。

```
# http_server.py — HTTP+SSE模式
from mcp.server.sse import SseServerTransport
from starlette.applications import Starlette
from starlette.routing import Route

transport = SseServerTransport("/messages/")

async def handle_sse(request):
    async with transport.connect_sse(request.scope, reque
        await server.run(streams[0], streams[1], server.c

app = Starlette(routes=[
    Route("/sse", endpoint=handle_sse),
    Route("/messages/", endpoint=transport.handle_post_me
])
```

配置方式也不同：

```
{
  "mcpServers": {
    "remote-db": {
      "url": "https://mcp.example.com/sse",
      "headers": {
        "Authorization": "Bearer sk-xxx"
      }
    }
  }
}
```

6.2.3 WebSocket模式

当你需要服务器主动推送消息时（比如数据库变更通知、长时间任务进度），WebSocket是唯一选择。Claude Code对WebSocket的支持目前通过社区插件实现，但核心API已预留。

6.3 实战：数据库连接MCP服务器

让我们做一个真正有用的MCP服务器：连接PostgreSQL，让Claude Code能读表结构、查数据、甚至帮你写SQL。

6.3.1 完整实现

```
# pg_mcp_server.py — PostgreSQL MCP服务器
import asyncpg
from mcp.server import Server
from mcp.server.stdio import stdio_server
from mcp.types import Tool, TextContent

server = Server("pg-explorer")

@server.list_tools()
async def list_tools() -> list[Tool]:
    return [
        Tool(name="list_tables", description="列出数据库所有
        Tool(name="describe_table", description="查看表结构
             inputSchema={"type": "object", "properties":
        Tool(name="run_query", description="执行只读SQL查询
             inputSchema={"type": "object", "properties":
    ]

# 全局连接池
pool: asyncpg.Pool = None

async def get_pool():
    global pool
    if pool is None:
        pool = await asyncpg.create_pool(
            dsn="postgresql://user:pass@localhost:5432/my
            min_size=1, max_size=5
        )
    return pool

@server.call_tool()
async def call_tool(name: str, arguments: dict):
```

```
    p = await get_pool()
    if name == "list_tables":
        async with p.acquire() as conn:
            rows = await conn.fetch(
                "SELECT table_name FROM information_schem
            )
            return [TextContent(type="text", text=str([r[
    elif name == "describe_table":
        async with p.acquire() as conn:
            rows = await conn.fetch(
                "SELECT column_name, data_type, is_nullab
                arguments["table"]
            )
            return [TextContent(type="text", text=str([di
    elif name == "run_query":
        sql = arguments["sql"].strip().upper()
        # 安全：只允许SELECT
        if not sql.startswith("SELECT"):
            return [TextContent(type="text", text="错误：只
        async with p.acquire() as conn:
            rows = await conn.fetch(arguments["sql"])
            return [TextContent(type="text", text=str([di

async def main():
    async with stdio_server() as (read, write):
        await server.run(read, write, server.create_initi

import asyncio
asyncio.run(main())
```

6.3.2 实战对话效果

配置好这个MCP服务器后，你在Claude Code里的体验是这样的：

```
你：帮我分析一下users表里有多少用户是上周注册的

Claude Code：[调用 list_tables] → [调用 describe_table tabl
         → [调用 run_query sql="SELECT COUNT(*) FROM users

回复：上周新注册了284位用户，详细分布如下...
```

不需要手动粘贴数据库结构，不需要在几个窗口之间切换。**MCP让Claude Code的上下文自动包含了你的数据库全貌。**

6.4 三层纵深安全机制

MCP最致命的风险是：**AI获得了执行能力**。一个prompt注入攻击可能变成一句 `DROP TABLE users` 。老K的纵深防御策略分为三层：

第一层：Agent侧（Claude Code）

在CLAUDE.md或Skills的prompt里预置安全护栏：

```
<span style="font-size:0.5pt;">§TOC6-6§</span>
## 数据库操作安全规则

- 永远不要执行 DROP、TRUNCATE、ALTER、CREATE、INSERT、UPDATE、
- 查询时必须带 LIMIT，默认 LIMIT 100
- 涉及用户隐私数据（email, phone, password_hash）的查询必须跳过
- 查询超过1000行的，先 COUNT 确认规模，再分页
```

第二层：MCP服务器侧

这是最关键的一层。**永远不要信任客户端的SQL**——在服务器端做白名单校验：

```
# 安全中间件：SQL白名单
FORBIDDEN_KEYWORDS = ["DROP", "DELETE", "UPDATE", "INSERT
MINIMUM_REQUIRED_KEYWORDS = ["SELECT"]

async def safe_query(conn, sql: str):
    upper_sql = sql.upper().strip()
    for kw in FORBIDDEN_KEYWORDS:
        if kw in upper_sql:
            raise PermissionError(f"FORBIDDEN_KEYWORDS: {
    if not any(kw in upper_sql for kw in MINIMUM_REQUIRED
        raise PermissionError("Only SELECT is allowed")
    if "LIMIT" not in upper_sql:
        sql = sql.rstrip(";") + " LIMIT 100"
    return await conn.fetch(sql)
```

老K的实战教训：**正则过滤不靠谱**——攻击者可以用注释、编码、大小写绕过。用`sqlparse`做AST级别的解析是唯一可靠的方案。

第三层：数据库侧

给MCP服务器使用的数据库账号只授予只读权限：

```
-- 创建MCP专用只读角色
CREATE ROLE mcp_readonly WITH LOGIN PASSWORD 'xxx';
GRANT CONNECT ON DATABASE mydb TO mcp_readonly;
GRANT USAGE ON SCHEMA public TO mcp_readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO mcp_readon
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON
```

三层纵深防御的失效模式分析：

表 7–3

防线	被突破后	后果	补救
第1层（Agent prompt）被绕过	AI发出了危险SQL	第2层拦截	修复prompt
第2层（服务器校验）被绕过	危险SQL到达数据库	第3层拦截（权限不足）	紧急下线修复
第3层（DB权限）被绕过	数据被破坏	**灾难**	从备份恢复

老K的原则：**宁可让Agent的功能受限，也不要给它不必要的权限。只读就是只读。**

6.5 MCP + Skills协同实战

MCP和Skills是Claude Code的两大扩展机制，它们的定位完全不同：

表 7–4

维度	MCP	Skills
本质	**能力扩展** （让AI能做新的事）	**知识扩展** （让AI知道该怎么做）
运行位置	MCP服务器进程	Claude Code内部
主要内容	Tools, Resources, Prompts	方法论、规则、上下文
典型用途	查数据库、调API、读文件	代码规范、架构知识、流程指南
更新频率	很少变化（接口稳定）	频繁迭代（知识更新快）

真正的威力来自**两者协同**。

实战案例：智能数据库助手

假设你要让Claude Code成为一个数据库管理员助手。

Skill文件（`dba-assistant.md`）：

```
# 数据库管理员助手

<span style="font-size:0.5pt;">§TOC6-8§</span>
## 分析流程
当用户询问数据库相关问题时，遵循以下流程：

1. **了解全貌**：先用 list_tables 了解数据库有哪些表
2. **定位问题表**：根据用户描述，用 describe_table 查看相关表结构
3. **编写查询**：基于表结构，编写精确的SQL查询
4. **验证结果**：确保返回数据符合预期

<span style="font-size:0.5pt;">§TOC6-9§</span>
## 性能优化规则
- 查询大表时务必使用索引列作为WHERE条件
- 如果 users 表超过 100 万行，按 created_at 分区查询
- 关联查询超过 3 张表时，建议先查主表再逐步关联

<span style="font-size:0.5pt;">§TOC6-10§</span>
## 数据解释规范
- 金额字段以人民币显示，格式：¥x,xxx.xx
- 时间字段显示为北京时间（UTC+8）
- 百分比保留一位小数
```

MCP配置（`claude_desktop_config.json`）：

```
{
  "mcpServers": {
    "pg-explorer": {
      "command": "python",
      "args": ["-m", "pg_mcp_server"]
    }
  },
  "skills": ["dba-assistant.md"]
}
```

这样配置后，当你问”帮我看看本周新用户注册趋势”，Claude Code会：

1. Skill 告诉它”先 list_tables 了解全貌”
2. MCP 执行 list_tables，返回 [“users”, “orders”, “products”]
3. Skill 告诉它”用 describe_table 查看 users 表结构”
4. MCP 返回 users 表的列信息
5. Skill 告诉它”按 created_at 分区查询”
6. MCP 执行按日分组的查询
7. Skill 告诉它”时间显示为北京时间”

整个过程自动化、规范化、可复制。

6.6 自定义MCP服务器实战：文件系统浏览器

作为一个综合性实战，我们构建一个文件系统MCP服务器，让Claude Code能安全地浏览和搜索文件。

```
# filesystem_mcp.py
import os, glob
from pathlib import Path
from mcp.server import Server
from mcp.server.stdio import stdio_server
from mcp.types import Tool, TextContent

server = Server("fs-explorer")

# 白名单：只允许访问这些目录
ALLOWED_ROOTS = ["/Users/wangguobao/projects", "/Users/wa

def is_allowed(path: str) -> bool:
    real = os.path.realpath(path)
    return any(real.startswith(os.path.realpath(root)) fo

@server.list_tools()
async def list_tools():
    return [
        Tool(name="list_dir", description="列出目录内容",
             inputSchema={"type": "object", "properties":
        Tool(name="read_file", description="读取文件内容（限
             inputSchema={"type": "object", "properties":
        Tool(name="search_files", description="按glob模式搜
             inputSchema={"type": "object", "properties":
        Tool(name="grep", description="在文件中搜索内容",
             inputSchema={"type": "object", "properties":
    ]

@server.call_tool()
async def call_tool(name: str, arguments: dict):
    if name == "list_dir":
        path = arguments["path"]
        if not is_allowed(path):
```

```
            return [TextContent(type="text", text="拒绝访问
        try:
            items = os.listdir(path)
            return [TextContent(type="text", text=str(ite
        except Exception as e:
            return [TextContent(type="text", text=str(e))

    elif name == "read_file":
        path = arguments["path"]
        if not is_allowed(path):
            return [TextContent(type="text", text="拒绝访问
        size = os.path.getsize(path)
        if size > 10 * 1024:
            return [TextContent(type="text", text=f"文件过
        with open(path) as f:
            return [TextContent(type="text", text=f.read(

    elif name == "search_files":
        pattern = arguments["pattern"]
        results = []
        for root in ALLOWED_ROOTS:
            for p in glob.glob(os.path.join(root, pattern
                if is_allowed(p):
                    results.append(p)
        return [TextContent(type="text", text=str(results

    elif name == "grep":
        import subprocess
        path, pattern = arguments["path"], arguments["pat
        if not is_allowed(path):
            return [TextContent(type="text", text="拒绝访问
        result = subprocess.run(["grep", "-rn", "-m", "20
                                capture_output=True, text
        return [TextContent(type="text", text=result.stdo
```

```
async def main():
    async with stdio_server() as (read, write):
        await server.run(read, write, server.create_initi

import asyncio
asyncio.run(main())
```

这个实战包含了本章所有的核心概念：三种Tool类型、安全白名单、大小限制、超时控制。

6.7 常见陷阱与最佳实践

老K在多个生产项目中积累的MCP实战经验：

表 7–5

陷阱	症状	解决方案
stdout被print污染	协议解析失败	只用logging输出到stderr
连接池耗尽	MCP响应变慢甚至超时	设置合理的pool size和超时
大结果集	Claude上下文溢出	服务器端做分页，默认LIMIT
环境变量泄露	敏感信息出现在日志	.env文件 + 最小权限原则
工具数量过多	Claude选择困难	一个MCP 5–10个工具为宜
没有输入校验	prompt注入	永远不信任客户端参数

老K的MCP设计原则：

1. **单一职责**：一个MCP服务器只做一件事，做好
2. **最小权限**：给MCP的数据库账号/API Key只给必需的权限
3. **故障隔离**：一个MCP崩溃不应影响其他MCP
4. **显式优于隐式**：每个工具的功能和限制在description里说清楚
5. **可观测**：每个工具调用都打日志，方便排查问题

本章小结

MCP是Claude Code从”代码助手”升级为”工程平台”的关键基础设施。选对传输方式、做好安全、和Skills协同作战，这三个要点做到位，你的MCP实践就不会翻车。

下一章我们将进入Headless模式——把Claude Code送进CI/CD流水线，让AI成为你的自动化基础设施的一部分。

第8章 Headless与CI/CD：把Claude Code送进流水线

决策者摘要

Headless模式的本质是把Claude Code从一个**交互式对话工具**变成一个**可编程的自动化引擎**。你不再对着终端打字，而是让GitHub Actions、GitLab CI或者Jenkins来调用它。这让代码审查、文档生成、重构任务、发布检查这些重复劳动从”人做”变成”机器做”。

核心决策点：

表 8–1

决策项	建议	理由
CI/CD集成方式	GitHub Actions 优先	开箱即用，社区actions丰富
API Key管理	使用OIDC + GitHub Secrets	零硬编码，自动轮换
执行频率	审慎，每次PR跑一次即可	Claude Code调用有成本
失败处理	降级而非阻断	AI输出不100%可靠，设fallback
Routine vs CI	定时任务用Routine，事件驱动用CI	各司其职，不要混用

读完本章你会掌握：4个维度的完整参数控制体系、三大CI平台的实战集成方案、Routines定时任务的配置与监控、通过Channels把AI输出推送到IM和Webhooks。

7.1 Headless的四个维度参数控制

Headless模式下，你失去了交互界面，但获得了完整的编程控制。Claude Code暴露了四个维度的参数让你精确控制AI的行为。

维度一：任务定义（What）

这是最核心的维度——告诉Claude Code做什么。

```
# 直接传入prompt
claude -p "帮我审查这个PR中的安全漏洞" --print

# 从--system-prompt注入系统级指令
claude -p "review pr" \
  --system-prompt "你是资深安全审计专家。只关注OWASP Top 10相关

# 引用文件作为上下文
claude -p "审查 src/auth.py 的安全问题" \
  --add-dir src/ \
  --print
```

维度二：权限控制（Can）

在自动化环境中，权限管理是生死攸关的问题。你不可能让CI流水线有 `sudo rm -rf /` 的能力。

```
# 权限模式三档
claude -p "fix this" --permission-mode acceptEdits  # 只接
claude -p "check this" --permission-mode bypassPermission
claude -p "review" --permission-mode plan  # 只读模式，不修
```

表 8-2

权限模式	适用场景	风险等级
`acceptEdits`	自动化重构、格式化	🟡 中等——会修改文件
`bypassPermissions`	只有完全可信任务	🔴 高——可能执行任意命令
`plan`	代码审查、分析	🟢 低——只读不写

老K的铁律：**CI/CD中永远不用 `bypassPermissions`，除非你想在凌晨3点被on-call电话叫醒。**

维度三：输出控制（How）

```
# --print 把最终回复打印到stdout（CI必备）
claude -p "分析代码质量" --print

# --output-format json 输出结构化JSON
claude -p "分析循环复杂度" --output-format json --print

# --max-turns 限制交互轮次，防止失控
claude -p "重构这个函数" --max-turns 5

# --verbose 显示详细日志
claude -p "debug" --verbose
```

维度四：会话管理（State）

```
# --resume 从上次中断的会话继续
claude --resume <session-id> -p "继续上次的任务"

# --continue 自动恢复最近的会话
claude --continue

# 列出所有会话
claude --resume
```

完整参数矩阵

表 8–3

参数	类型	默认值	说明
`-p / --print`	flag	false	非交互模式，输出到stdout
`--output-format`	enum	text	text / json / stream-json
`--max-turns`	int	无限制	最大工具调用轮数
`--model`	string	默认模型	指定模型
`--permission-mode`	enum	default	acceptEdits / bypassPermissions / plan
`--allowed-tools`	list	全部	白名单工具列表
`--disallowed-tools`	list	空	黑名单工具列表
`--add-dir`	path	cwd	添加工作目录
`--system-prompt`	string	空	系统级提示
`--verbose`	flag	false	详细日志

7.2 GitHub Actions 集成实战

7.2.1 最小可用集成

```
# .github/workflows/claude-review.yml
name: Claude Code Review

on:
  pull_request:
    types: [opened, synchronize]

jobs:
  review:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
    steps:
      - uses: actions/checkout@v4

      - name: Claude Code PR Review
        env:
          ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KE
        run: |
          claude -p "审查这个PR的所有变更。关注安全问题、代码质量
          输出简洁的审查意见，每个问题标注文件和行号。" \
            --print \
            --permission-mode plan \
            --output-format json > review.json

      - name: Post Review Comment
        uses: actions/github-script@v7
        with:
          script: |
```

```
            const fs = require('fs');
            const review = JSON.parse(fs.readFileSync('re
            await github.rest.issues.createComment({
              owner: context.repo.owner,
              repo: context.repo.repo,
              issue_number: context.issue.number,
              body: `## 🤖 Claude Code 审查意见\n\n${revie
            });
```

7.2.2 生产级集成：带条件判断和降级

```
name: Production Claude Review

on:
  pull_request:
    paths:
      - 'src/**'
      - 'lib/**'
      - '*.py'
      - '*.js'
      - '*.ts'

jobs:
  claude-review:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Get Changed Files
        id: changed-files
        uses: tj-actions/changed-files@v41
        with:
          files: |
            src/**
            lib/**

      - name: Skip if no relevant changes
        if: steps.changed-files.outputs.any_changed != 't
```

```
        run: echo "No relevant files changed, skipping"

      - name: Claude Code Review
        if: steps.changed-files.outputs.any_changed == 't
        id: claude
        continue-on-error: true  # 关键：不因为AI不稳定而阻断
        env:
          ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KE
        run: |
          # 只审查变更的文件
          FILES="${{ steps.changed-files.outputs.all_chan
          claude -p "只审查这些文件：$FILES。评估代码质量、安全
            --print \
            --permission-mode plan \
            --add-dir src/ \
            --max-turns 10 \
            | tee review.md

      - name: Post Review or Fallback
        if: always()
        uses: actions/github-script@v7
        with:
          script: |
            const fs = require('fs');
            let body;
            if (fs.existsSync('review.md')) {
              body = fs.readFileSync('review.md', 'utf8')
            } else {
              body = '⚠️ Claude Code 审查跳过（无相关文件变更
            }
            await github.rest.issues.createComment({
              issue_number: context.issue.number,
              body: body
            });
```

老K的实战经验：`continue-on-error: true` 是救命稻草。API 偶尔 429、偶尔超时、偶尔返回非预期格式——你的 CI 不能因为 AI 不稳定就挂了。

7.2.3 OIDC认证（不用存API Key）

存 API Key 在 GitHub Secrets 里已经比硬编码好很多，但还有更安全的方案：OIDC。

```
# 使用 OIDC + AWS Secrets Manager 获取 API Key
- name: Configure AWS Credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::123456789:role/github-ac
    aws-region: us-east-1

- name: Get API Key from Secrets Manager
  run: |
    export ANTHROPIC_API_KEY=$(aws secretsmanager get-sec
      --secret-id anthropic/api-key \
      --query SecretString --output text | jq -r '.ANTHRO
    claude -p "review" --print --permission-mode plan
```

7.3 GitLab CI 集成

GitLab CI 的集成方式和 GitHub Actions 类似，但有几个关键差异。

```
# .gitlab-ci.yml
claude-review:
  image: node:20
  stage: review
  only:
    - merge_requests
  before_script:
    - npm install -g @anthropic-ai/claude-code
  script:
    - |
      claude -p "审查这个MR的所有变更。输出格式:
      ## 严重问题
      - ...
      ## 建议改进
      - ...
      ## 总结
      " \
        --print \
        --permission-mode plan \
        --output-format json > review.json
    - |
      # 通过GitLab API发评论
      curl --request POST \
        --header "PRIVATE-TOKEN: $GITLAB_API_TOKEN" \
        --header "Content-Type: application/json" \
        --data "{\"body\": $(jq -Rs . review.json)}" \
        "$CI_API_V4_URL/projects/$CI_PROJECT_ID/merge_req
  allow_failure: true  # GitLab 的等价于 continue-on-error
```

7.4 Jenkins 集成

Jenkins 需要多一步：确保 agent 节点上安装了 claude CLI。

```
// Jenkinsfile
pipeline {
    agent {
        docker {
            image 'node:20'
            args '-v $HOME/.claude:/root/.claude'
        }
    }
    environment {
        ANTHROPIC_API_KEY = credentials('anthropic-api-ke
    }
    stages {
        stage('Claude Review') {
            steps {
                script {
                    sh '''
                        npm install -g @anthropic-ai/clau
                        claude -p "审查代码变更" --print --
                    '''
                }
            }
        }
    }
    post {
        failure {
            echo 'Claude review failed but pipeline conti
        }
    }
}
```

7.5 Routines：定时任务

Routines 是 Claude Code 的定时任务功能——在指定时间自动执行 AI 任务。

配置方式

在你的项目根目录下创建 `.claude/routines/` 目录：

```
# .claude/routines/daily-standup.md
# 每天早上9:30自动生成团队站会摘要

cron: 30 9 * * 1-5
channel: slack-standup
permission-mode: plan
max-turns: 5
---

分析昨天今天的工作进展：

1. 查看 git log --since="24 hours ago" 获取所有提交
2. 总结主要变更：按功能分类（新功能/修复/重构）
3. 标注正在进行的任务
4. 输出格式为 Slack Markdown，适合直接发送
```

```
# .claude/routines/weekly-report.md
# 每周五下午5点生成周报

cron: 0 17 * * 5
channel: email-report
permission-mode: plan
max-turns: 10
---

生成本周工作总结：

1. 汇总本周所有 git 提交
2. 统计代码变更量（增删行数）
3. 列出本周关闭的 issues
4. 输出为中文周报格式，包含：
   - 本周完成
   - 下周计划
   - 风险/阻塞
```

老K的提醒：**Routines 的 `max-turns` 一定要设**。一个失控的定时任务可能跑几个小时，烧掉几百刀 token。

7.6 Channels：事件推送

Channels 让 Claude Code 能把执行结果推送到外部平台。目前支持 Telegram、Discord、iMessage 和 Webhooks。

7.6.1 Telegram 集成

```
# .claude/channels/telegram.md
channel: telegram
config:
  bot_token: ${TELEGRAM_BOT_TOKEN}
  chat_id: "-1001234567890"
---

# 当Claude完成重要任务时，推送通知到这个频道
```

然后在 Routine 里引用：

```
# .claude/routines/error-monitor.md
cron: */30 * * * *
channel: telegram  # 结果推送到Telegram
permission-mode: plan
---

分析最近30分钟的日志，发现ERROR/FATAL级别的异常立即报警。
```

7.6.2 Discord 集成

```
# .claude/channels/discord.md
channel: discord
config:
  webhook_url: ${DISCORD_WEBHOOK_URL}
```

7.6.3 自定义 Webhook

```
# .claude/channels/custom-webhook.md
channel: webhook
config:
  url: "https://api.your-company.com/claude-results"
  headers:
    Authorization: "Bearer ${WEBHOOK_TOKEN}"
    Content-Type: "application/json"
```

7.6.4 iMessage 集成（macOS 专属）

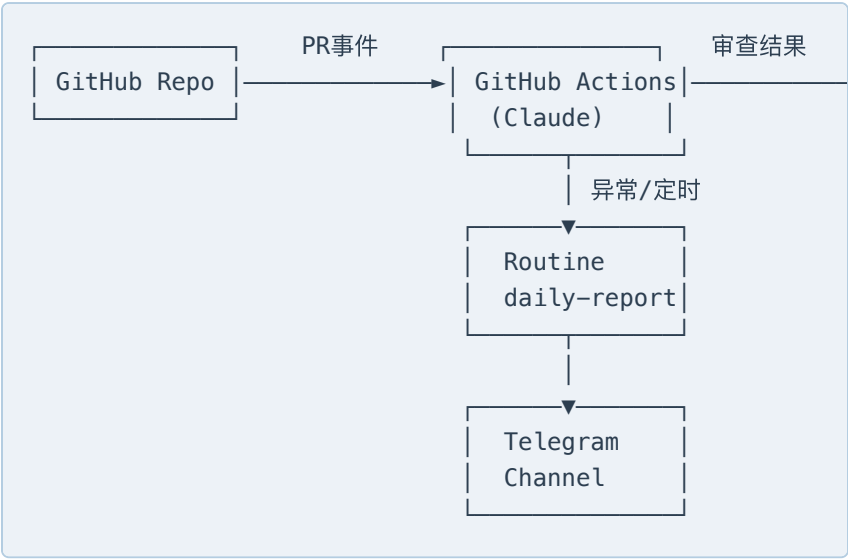
```
# .claude/channels/imessage.md
channel: imessage
config:
  recipients:
    - "+8613800138000"
    - "lao.k@icloud.com"
```

老K的实战坑：iMessage Channel 只能在 macOS 上用，依赖 AppleScript 桥接。出问题时先检查”信息”app 是否正常登录。

7.7 实战：全自动代码审查流水线

让我们把本章所有内容整合成一个生产级流水线。

架构



完整配置

GitHub Actions 工作流：

```
# .github/workflows/ai-pipeline.yml
name: AI Pipeline

on:
  pull_request:
    types: [opened, synchronize, reopened]
  issue_comment:
    types: [created]

jobs:
  # Job 1: 代码审查
  code-review:
    if: github.event_name == 'pull_request'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Claude Review
        run: |
          claude -p "$(cat .claude/prompts/pr-review.md)"
            --print \
            --permission-mode plan \
            --max-turns 10 \
            --output-format json > review.json || echo '{

  # Job 2: 注释触发的检查
  comment-trigger:
    if: github.event_name == 'issue_comment' && contains(
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Process Command
```

```
        run: |
          COMMAND=$(echo "${{ github.event.comment.body }
          claude -p "$COMMAND" --print --permission-mode
```

Routine 配置：

```
# .claude/routines/daily-code-health.md
cron: 0 9 * * 1-5
channel: telegram
permission-mode: plan
max-turns: 8
---

分析昨日所有仓库的提交，生成代码健康日报：

1. 检查是否有 TODO/FIXME/HACK 注释增加
2. 检查是否有大文件提交（>500行变更）
3. 检查测试覆盖率变化
4. 输出简洁的Markdown报告
```

Channel 配置：

```
# .claude/channels/telegram.md
channel: telegram
config:
  bot_token: ${TELEGRAM_BOT_TOKEN}
  chat_id: ${TELEGRAM_CHAT_ID}
  parse_mode: Markdown
```

7.8 常见陷阱

表 8–4

陷阱	症状	解决
CI超时	Claude跑了10分钟没停	设 `--max-turns 5`
成本失控	月底账单吓人	控制PR触发频率 + max-turns
输出格式错误	后续脚本解析失败	用 `--output-format json` + jq
并发冲突	多个PR同时触发	用 concurrency group
没有降级	AI挂了CI就挂	`continue-on-error: true`
API Key泄露	日志里出现Key	永远用 secrets，打印前清理

本章小结

Headless模式是把Claude Code从”个人工具”变成”团队基础设施”的关键一步。四维度参数让你精确控制行为，CI/CD集成让AI进入自动化流水线，Routines和Channels让AI主动推送信息。记住老K的三条铁律：**1) 设max-turns 2) 用continue-on-error降级 3) 永远只在CI里用plan模式。**

下一章我们深入Agent SDK——用编程的方式调用Claude Code，构建你自己的AI应用。

第9章 Agent SDK：编程式AI工程

决策者摘要

Agent SDK是把Claude Code从”黑盒工具”变成”可编程平台”的核心。通过SDK，你可以把Claude Code嵌入到你自己的Python/Node.js应用里，构建自定义的AI Agent——不再需要通过CLI交互，而是用代码完全控制：发消息、收回复、注册工具、管理会话、处理流式输出。

核心决策点：

表 9–1

决策项	建议	理由
SDK vs CLI	需要编程控制用SDK，简单任务用CLI	SDK更灵活但需要写代码
同步 vs 流式	内部工具用同步，用户面用流式	流式体验好但处理复杂
工具数量	5–15个为宜	太多模型会选错，太少功能不足
会话管理	短任务免会话，长任务必须持久化	避免上下文膨胀和成本浪费
结构化输出	能用就用	后处理代码从100行变10行

读完本章你会掌握：query() 核心API的全部参数和用法、消息类型的选型策略、完整会话生命周期管理、@tool装饰器的限制隔离与资源管理、结构化输出的4种模式、4道安全防线的实战编码、构建一个完整的代码分析Web服务。

8.1 核心API：query() 函数的正确打开方式

`query()` 是整个SDK的入口。理解它等于理解了SDK的80%。

最小调用

```
from anthropic import Anthropic

client = Anthropic()
response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=4096,
    messages=[{"role": "user", "content": "帮我审查这段代码的
)
```

但这只是普通的API调用。Agent SDK的 `query()` 在这个基础上增加了**工具调用循环**——Claude会自动决定需要调用哪些工具，执行后把结果传回去，继续推理，直到完成任务。

```
from claude_agent_sdk import ClaudeAgent, tool

agent = ClaudeAgent(
    model="claude-sonnet-4-20250514",
    max_turns=10,        # 最多10轮工具调用
    permission_mode="plan",
)

response = agent.query("分析src/auth.py的安全问题")
print(response.result)
```

完整参数说明

```
response = agent.query(
    prompt="审查所有Python文件的类型注解完整性",

    # === 上下文注入 ===
    system_prompt="你是Python类型系统专家。用中文回复。",  # 系
    allowed_tools=["Bash", "Read", "Grep"],
    disallowed_tools=["Write", "Edit"],
    add_dir=["src/", "lib/"],

    # === 行为控制 ===
    max_turns=15,                      # 最大推理轮数
    permission_mode="acceptEdits",     # 权限模式
    model="claude-sonnet-4-20250514", # 模型选择

    # === 输出控制 ===
    output_format="json",              # text | json | stre
    verbose=True,                      # 详细日志

    # === 高级 ===
    resume="session_abc123",           # 恢复会话
    mcp_servers=["pg-explorer"],       # 加载MCP服务器
    skills=["code-review.md"],         # 加载Skills
)
```

老K的建议：**`max_turns` 是最容易被忽略但最重要的参数。**不设这个，你的程序可能在某个循环里跑几百个回合，烧掉几十刀。

8.2 消息类型：不止是文本

Agent SDK 的消息比普通聊天API丰富得多。理解每种类型的适用场景是写出好Agent的关键。

消息类型全景

表 9–2

类型	用途	示例
`user`	用户输入	“审查这段代码”
`assistant`	Claude的文本回复	“我发现了3个问题…”
`tool_use`	Claude决定调用工具	`{"name": "Read", "input": {"file_path": "app.py"}}`
`tool_result`	工具执行结果	`{"content": "import flask\n...", "is_error": false}`
`system`	系统级指令	角色设定、规则约束

多轮对话中的消息拼接

```
from claude_agent_sdk import types as t

messages = [
    t.user("帮我分析这个函数的性能"),
    t.assistant("我来分析。首先读取文件内容。"),
    t.tool_use(id="call_1", name="Read", input={"file_pat
    t.tool_result(
        tool_use_id="call_1",
        content="def heavy(): ...",
        is_error=False
    ),
    # Claude继续推理...
]
```

流式响应处理

生产环境中，用户不想等10秒看一堵墙的文本——流式输出是必需品。

```
async for event in agent.stream_query("分析代码质量"):
    match event:
        case t.TextDelta(text=chunk):
            print(chunk, end="", flush=True)  # 逐字输出
        case t.ToolUse(name=name, input=params):
            print(f"\n🔧 调用工具: {name}({params})")
        case t.ToolResult(output=content):
            print(f"✅ 工具完成: {len(content)} chars")
        case t.AssistantMessage(content=final):
            print(f"\n\n=== 最终回复 ===\n{final}")
        case t.Error(message=msg):
            print(f"\n❌ 错误: {msg}")
```

老K的实战模式：**用 `match-case` 处理流式事件**。Python 3.10+ 的模式匹配让事件处理代码可读性提高一个数量级。

8.3 会话管理：上下文的生命周期

会话的三种模式

```
# 模式1：无状态（每次query独立）
agent.query("fix bug in auth.py")

# 模式2：手动管理会话
session = agent.create_session()
session.query("read auth.py")
session.query("find the security issue")
session.query("fix it")
session.close()  # 释放资源
stats = session.usage  # 获取token用量

# 模式3：持久化会话（保存到磁盘）
session = agent.create_session(
    session_id="my-project-session",
    persist=True,  # 自动保存到 ~/.claude/sessions/
)
session.query("analyze codebase")
# 程序退出后...
session2 = agent.resume_session("my-project-session")
session2.query("continue the analysis")
```

会话状态查询

```
# 查看当前会话状态
info = session.info()
print(f"消息数: {info.message_count}")
print(f"Token用量: {info.total_tokens}")
print(f"工具调用次数: {info.tool_calls}")

# 获取会话完整历史
history = session.get_messages()
for msg in history:
    print(f"{msg.role}: {msg.content[:80]}...")
```

会话策略对比

表 9–3

策略	适用场景	优点	缺点
无状态	API式单次调用	简洁，无状态泄漏	不能连续对话
手动管理	Web应用请求级	灵活控制生命周期	需要自行清理
持久化	长时间分析任务	跨进程/重启恢复	磁盘占用，陈旧上下文

老K的建议：**Web服务用请求级会话（手动管理），批处理任务用持久化会话。** 不要在请求之间共享一个全局会话——并发会乱。

8.4 自定义工具：@tool 装饰器实战

`@tool` 装饰器是Agent SDK最强大的功能——你可以用纯Python函数扩展Claude的能力。

基础工具

```
from claude_agent_sdk import tool
from typing import Annotated

@tool(
    name="get_weather",
    description="获取指定城市的天气信息"
)
async def get_weather(
    city: Annotated[str, "城市名称，如'北京'或'Beijing'"],
    unit: Annotated[str, "温度单位：celsius 或 fahrenheit"]
) -> str:
    """实际调用天气API"""
    import httpx
    async with httpx.AsyncClient() as client:
        resp = await client.get(
            f"https://api.weather.example.com/current",
            params={"city": city, "unit": unit}
        )
        data = resp.json()
        return f"{city}: {data['temp']}°{unit}, {data['co
```

带错误处理的工具

```
@tool(
    name="query_database",
    description="执行只读SQL查询。数据库包含用户、订单、产品表。'
)
async def query_database(
    sql: Annotated[str, "SQL查询语句，只允许SELECT"]
) -> str:
    # 第1道防线：输入校验
    if not sql.strip().upper().startswith("SELECT"):
        return "错误：只允许SELECT查询"

    if ";" in sql.rstrip(";"):
        return "错误：不允许批量查询"

    try:
        # 第2道防线：资源限制
        async with pool.acquire() as conn:
            result = await conn.fetch(sql + " LIMIT 100")

        # 第3道防线：数据脱敏
        sanitized = []
        for row in result:
            d = dict(row)
            for sensitive in ['email', 'phone', 'password
                if sensitive in d:
                    d[sensitive] = '***REDACTED***'
            sanitized.append(d)

        return str(sanitized)
```

```
    except Exception as e:
        return f"查询失败: {str(e)}"
```

工具隔离实战

你的工具可能有副作用（发邮件、写数据库、调支付接口）。老K的隔离策略：

```
from contextlib import asynccontextmanager
import asyncio

class ToolSandbox:
    """工具执行沙箱"""

    def __init__(self, timeout: int = 30, max_memory: int
        self.timeout = timeout
        self.max_memory = max_memory

    @asynccontextmanager
    async def run(self, tool_name: str):
        try:
            # 超时控制
            yield await asyncio.wait_for(
                self._execute(tool_name),
                timeout=self.timeout
            )
        except asyncio.TimeoutError:
            return f"工具 '{tool_name}' 执行超时 (>{self.ti
        except Exception as e:
            return f"工具 '{tool_name}' 执行异常: {e}"

# 使用
@tool(name="risky_operation")
async def risky_operation(data: str) -> str:
    sandbox = ToolSandbox(timeout=10)
    async with sandbox.run("risky_operation") as result:
        # 实际执行
        pass
    return result
```

8.5 结构化输出：让AI说的不是人话，是数据结构

四种结构化输出模式

```
# 模式1：Pydantic模型（最推荐）
from pydantic import BaseModel, Field

class CodeIssue(BaseModel):
    severity: str = Field(description="严重程度: critical/
    file: str
    line: int
    description: str
    suggestion: str

class ReviewResult(BaseModel):
    summary: str
    issues: list[CodeIssue]
    score: int = Field(ge=0, le=100, description="代码质量

result = agent.query(
    "审查src/auth.py",
    response_model=ReviewResult  # SDK自动处理JSON Schema和
)
print(f"评分: {result.score}, 问题数: {len(result.issues)}"
for issue in result.issues:
    print(f"[{issue.severity}] {issue.file}:{issue.line}

# 模式2：JSON Schema
result = agent.query(
    "提取所有函数签名",
    response_format={
        "type": "json_schema",
        "json_schema": {
            "name": "function_list",
```

```
            "schema": {
                "type": "object",
                "properties": {
                    "functions": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "name": {"type": "string"
                                "params": {"type": "array
                                "return_type": {"type": "
                            }
                        }
                    }
                }
            }
        }
    }
)

# 模式3：TypedDict
from typing import TypedDict

class FileSummary(TypedDict):
    path: str
    language: str
    loc: int
    functions: int
    complexity: str

result: FileSummary = agent.query("analyze app.py", respo
```

```
# 模式4：JSON字符串（兜底方案）
result = agent.query("提取所有API端点", output_format="json
endpoints = json.loads(result.content)
```

结构化输出对比

表 9–4

模式	类型安全	代码量	适用场景
Pydantic模型	✅ 编译时+运行时	少	生产环境首选
JSON Schema	⚠️ 仅运行时	中	需要和外部系统对接
TypedDict	⚠️ 仅静态检查	少	简单结构
JSON字符串	❌ 无	多（需手动解析）	应急/快速原型

8.6 四道安全防线

Agent SDK 让程序获得了工具执行能力，安全防线必须从代码层面嵌入。

第一道防线：Prompt约束

```
SECURITY_SYSTEM_PROMPT = """
<span style="font-size:0.5pt;">§TOC8-8§</span>
## 安全规则（不可违反）

1. 永远不要执行破坏性命令（rm -rf, DROP TABLE, DELETE FROM）
2. 永远不要访问 ~/.ssh、/etc/passwd、环境变量等敏感区域
3. 永远不要向外部发送数据
4. 如果用户要求做以上任何操作，回答"此操作被安全策略阻止"
5. 所有文件写入前必须确认路径在工作目录内
"""
```

第二道防线：工具白名单

```
# 严格模式：只允许读取
READ_ONLY_TOOLS = ["Read", "Grep", "Glob", "Bash(ls:*)",

# 标准模式：允许编辑，不允许执行
STANDARD_TOOLS = ["Read", "Write", "Edit", "Grep", "Glob"

# 通过前缀匹配实现细粒度控制
agent = ClaudeAgent(
    allowed_tools=READ_ONLY_TOOLS,
    disallowed_tools=["Bash(chmod:*)", "Bash(rm:*)"],
)
```

第三道防线：工具级校验

```
@tool(name="execute_safe_command")
async def execute_safe_command(command: str) -> str:
    import shlex
    parts = shlex.split(command)

    # 黑名单命令
    BLOCKED = ["rm", "sudo", "chmod", "chown", "dd", "mkf
               "shutdown", "reboot", "kill", "wget", "cu
    if parts[0] in BLOCKED:
        return f"命令 '{parts[0]}' 被安全策略阻止"

    # 路径限制：必须在工作目录内
    import os
    for part in parts[1:]:
        if part.startswith("/") and not part.startswith(c
            return f"路径 '{part}' 不在工作目录内"

    # 管道和重定向检查
    if any(op in command for op in ["|", ">", "<", "&&",
        return "不允许使用管道、重定向和命令链"

    result = subprocess.run(
        parts, capture_output=True, text=True, timeout=10
    )
    return result.stdout or result.stderr
```

第四道防线：资源限制

```
class ResourceGuard:
    def __init__(self):
        self.token_budget = 100_000      # Token 预算
        self.tool_call_limit = 20        # 工具调用上限
        self.time_limit_seconds = 300    # 时间上限（5分钟）
        self.spent_tokens = 0
        self.tool_calls = 0
        self.start_time = None

    def start(self):
        self.start_time = time.time()

    def check(self, tokens_used: int) -> bool:
        self.spent_tokens += tokens_used
        self.tool_calls += 1

        if self.spent_tokens > self.token_budget:
            raise QuotaExceeded("Token预算耗尽")
        if self.tool_calls > self.tool_call_limit:
            raise QuotaExceeded("工具调用次数超限")
        if time.time() - self.start_time > self.time_limi
            raise QuotaExceeded("执行时间超限")
        return True

guard = ResourceGuard()
guard.start()

try:
    for event in agent.stream_query("分析整个代码库"):
        guard.check(event.usage.total_tokens)
```

```
        # 处理事件...
except QuotaExceeded as e:
    print(f"任务中止: {e}")
```

四道防线失效分析

表 9–5

防线	被绕过的方式	后果	如何加强
Prompt约束	Jailbreak/越狱攻击	AI可能尝试危险操作	多层prompt防御，定期审计
工具白名单	利用白名单内的工具做坏事	受限但仍然危险	细化到命令参数级别
工具级校验	编码绕过（Base64/Unicode）	执行危险命令	在shell层面用seccomp/AppArmor
资源限制	多个会话累计消耗	账单爆炸	加全局预算和告警

8.7 实战：代码分析Web服务

把上面的所有知识整合成一个可运行的Web服务。

```
# app.py — Claude Code Analysis Web Service
from fastapi import FastAPI, HTTPException, BackgroundTas
from fastapi.responses import StreamingResponse
from pydantic import BaseModel, Field
from claude_agent_sdk import ClaudeAgent, tool
from typing import Annotated
import asyncio, json, uuid
from datetime import datetime

app = FastAPI(title="Claude Code Analysis API")

# === 请求/响应模型 ===
class AnalyzeRequest(BaseModel):
    repo_url: str = Field(description="Git仓库URL")
    task: str = Field(description="分析任务描述，如'审查代码安
    files_filter: str = Field(default="*.py,*.js", descri
    max_turns: int = Field(default=10, ge=1, le=50)

class AnalyzeResponse(BaseModel):
    task_id: str
    status: str
    created_at: str

class TaskResult(BaseModel):
    task_id: str
    status: str
    result: str | None
    tokens_used: int
    tool_calls: int
    errors: list[str]

# === 任务存储（生产环境换成Redis/DB） ===
tasks: dict[str, TaskResult] = {}
```

```
# === Agent配置 ===
async def create_agent():
    return ClaudeAgent(
        model="claude-sonnet-4-20250514",
        permission_mode="plan",
        allowed_tools=["Read", "Grep", "Glob", "Bash(git:
    )

# === 工具：克隆仓库 ===
@tool(name="clone_repo", description="克隆Git仓库到本地临时目
async def clone_repo(
    url: Annotated[str, "Git仓库URL"],
    branch: Annotated[str, "分支名"] = "main"
) -> str:
    import tempfile, subprocess
    tmpdir = tempfile.mkdtemp(prefix="claude-analysis-")
    result = subprocess.run(
        ["git", "clone", "--depth", "1", "--branch", bran
        capture_output=True, text=True, timeout=30
    )
    if result.returncode != 0:
        return f"克隆失败: {result.stderr}"
    return f"已克隆到: {tmpdir}"

# === API端点 ===
@app.post("/analyze", response_model=AnalyzeResponse)
async def start_analysis(req: AnalyzeRequest, bg: Backgro
    task_id = str(uuid.uuid4())[:8]
    tasks[task_id] = TaskResult(
        task_id=task_id,
        status="queued",
        result=None,
        tokens_used=0,
        tool_calls=0,
```

```
        errors=[]
    )
    bg.add_task(run_analysis, task_id, req)
    return AnalyzeResponse(
        task_id=task_id,
        status="queued",
        created_at=datetime.now().isoformat()
    )

@app.get("/analyze/{task_id}", response_model=TaskResult)
async def get_result(task_id: str):
    if task_id not in tasks:
        raise HTTPException(404, "任务不存在")
    return tasks[task_id]

@app.get("/analyze/{task_id}/stream")
async def stream_result(task_id: str):
    if task_id not in tasks:
        raise HTTPException(404, "任务不存在")

    async def generate():
        task = tasks[task_id]
        while task.status in ("queued", "running"):
            yield f"data: {json.dumps({'status': task.sta
            await asyncio.sleep(1)
        yield f"data: {json.dumps({'status': task.status,

    return StreamingResponse(generate(), media_type="text

async def run_analysis(task_id: str, req: AnalyzeRequest)
    task = tasks[task_id]
    task.status = "running"

    try:
```

```
        agent = await create_agent()
        agent.register_tool(clone_repo)

        response = agent.query(
            prompt=f"""
            任务：{req.task}

            步骤：
            1. clone_repo("{req.repo_url}")
            2. 用Grep/Glob找到相关文件（过滤：{req.files_fil
            3. 按任务要求分析
            4. 输出完整的分析报告（Markdown格式）
            """,
            max_turns=req.max_turns,
            permission_mode="plan",
        )

        task.status = "completed"
        task.result = response.result
        task.tokens_used = response.usage.total_tokens
        task.tool_calls = response.usage.tool_calls

    except Exception as e:
        task.status = "failed"
        task.errors.append(str(e))

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

调用示例

```
# 提交分析任务
curl -X POST http://localhost:8000/analyze \
  -H "Content-Type: application/json" \
  -d '{
    "repo_url": "https://github.com/user/project.git",
    "task": "审查所有认证相关代码的安全漏洞",
    "max_turns": 15
  }'

# 查询结果
curl http://localhost:8000/analyze/abc12345

# 流式监听进度
curl http://localhost:8000/analyze/abc12345/stream
```

本章小结

Agent SDK 把 Claude Code 的能力从 CLI 里解放出来，让你可以构建真正的 AI 原生应用。`query()` 是入口，`@tool` 是扩展点，结构化输出是工程润滑剂，四道安全防线是生产环境的基本要求。

记住老K的SDK四原则：**1) 永远设max_turns 2) 工具要做输入校验 3) 结构化输出优于正则解析 4) 安全防线宁可多不可少。**

下一章进入Plugins生态——如何把前面的所有能力打包、分发、复用。

第10章 Plugins生态：能力的打包与分发

决策者摘要

Plugin是Claude Code能力的**最小可复用单元**。如果说Skills封装的是”知识”、MCP封装的是”能力”、Agent SDK封装的是”编程接口”，那么Plugin就是把这三者打包成一个独立、可分发的模块。Plugin生态决定了你的团队是”每个项目从零开始”还是”复用已有的能力积木”。

核心决策点：

表 10–1

决策项	建议	理由
开发生态还是消费生态	先消费社区Plugin，再自建	80%的需求已经被社区解决
公有Plugin市场 vs 私有市场	企业必须有私有市场	安全+内网API隔离
一个Plugin还是多个小Plugin	按领域拆分，一个Plugin一个职责	降低耦合，提高复用率
Plugin命名规范	`scope/name` 格式	避免命名冲突，方便组织管理
安装策略	Lock版本号	防止上游更新破坏你的流水线

读完本章你会掌握：plugin.json的完整结构定义和最佳实践、三种安装来源的配置与安全考量、Plugin生命周期的钩子编程、搭建私有Plugin市场的完整方案、多Plugin共存时的命名空间与冲突解决。

9.1 plugin.json：Plugin的身份证

`plugin.json` 是一个Plugin的入口文件，定义了元数据、依赖、能力声明和配置项。它是Plugin生态的基石。

完整结构

```
{
  "name": "code-reviewer",
  "version": "1.3.0",
  "description": "自动化代码审查Plugin，支持安全、性能、风格三个
  "author": {
    "name": "树懒老K",
    "email": "lao.k@example.com"
  },
  "license": "MIT",
  "repository": "https://github.com/sloth/cc-code-reviewe

  "capabilities": {
    "skills": ["code-review.md", "security-audit.md"],
    "mcpServers": [],
    "hooks": ["PostToolUse"],
    "tools": [
      {
        "name": "analyze_complexity",
        "description": "分析代码圈复杂度",
        "parameters": {
          "file": {"type": "string"},
          "threshold": {"type": "integer", "default": 10}
        }
      }
    ],
    "commands": [
      {
        "name": "/review",
        "description": "触发代码审查",
        "handler": "review_handler.py"
      }
    ],
```

```
    "channels": []
  },

  "dependencies": {
    "plugins": {
      "community/python-toolkit": ">=2.0.0 <3.0.0"
    }
  },

  "config": {
    "severity_threshold": {
      "type": "string",
      "default": "medium",
      "enum": ["critical", "high", "medium", "low"],
      "description": "最低报警级别"
    },
    "max_file_size_kb": {
      "type": "integer",
      "default": 500,
      "description": "跳过超过此大小的文件"
    }
  },

  "permissions": {
    "filesystem": ["read", "write:./reports/"],
    "network": ["api.github.com"],
    "commands": ["git", "grep", "find"]
  },

  "hooks": {
    "onInstall": "scripts/setup.py",
    "onUninstall": "scripts/cleanup.py"
  }
}
```

字段详解

表 10–2

字段	必须	说明	老K的坑
`name`	✅	全局唯一标识	别用 `my-plugin` 这种通用名
`version`	✅	SemVer版本号	Patch更新不要引入breaking change
`capabilities`	✅	能力声明	不要过度声明——声明的能力必须可工作
`dependencies`	❌	依赖的其他Plugin	版本范围宁严勿松
`config`	❌	用户可配置项	每个配置项必须有默认值
`permissions`	✅	需要的权限	申请最小权限
`hooks`	❌	生命周期钩子	安装脚本别写交互式逻辑

老K被坑过的教训：`version` 用 `0.x` 的时候要特别注意——SemVer 规定 `0.x` 版本可以有任何breaking change，但你升级别人的Plugin的时候才发现API全变了就晚了。**1.0.0 之前的Plugin不要作为生产依赖。**

9.2 安装来源与生命周期

三种安装来源

```
# 来源1：官方/社区市场（最推荐）
claude plugin install code-reviewer

# 来源2：Git仓库（私有Plugin / 开发中版本）
claude plugin install github.com/myteam/internal-plugin
claude plugin install github.com/myteam/internal-plugin@v
claude plugin install github.com/myteam/internal-plugin@m

# 来源3：本地路径（开发调试）
claude plugin install ./my-local-plugin
claude plugin install /Users/laok/projects/experimental-p
```

安装来源对比

表 10–3

来源	版本管理	安全审计	离线可用	适用场景
官方市场	✅ 自动	✅ 官方审核	❌	通用Plugin
Git仓库	⚠️ 手动指定Tag	⚠️ 自行审计	❌	私有Plugin、开发中
本地路径	❌ 无	✅ 完全控制	✅	开发调试、内网隔离

Plugin 生命周期

```
install → configure → enable → (update →) disable → unins
```

每个阶段可以注册钩子：

```
# scripts/setup.py — Plugin安装钩子
def on_install(context):
    """Plugin安装时执行"""
    # 检查依赖
    context.require_command("git")
    context.require_command("python3")

    # 初始化配置
    config_dir = context.config_dir
    os.makedirs(config_dir, exist_ok=True)

    # 写入默认配置
    if not os.path.exists(f"{config_dir}/settings.json"):
        context.write_default_config()

    print(f"✅ code-reviewer v{context.version} 安装成功")

def on_uninstall(context):
    """Plugin卸载时执行"""
    # 清理配置文件（保留用户数据）
    if context.confirm("删除所有配置和数据?"):
        context.cleanup()
```

升级策略

```
# 查看可升级的Plugin
claude plugin outdated

# 结果示例:
# code-reviewer     1.2.0 → 1.3.0
# python-toolkit    2.1.0 → 3.0.0  ⚠️ Major upgrade

# 升级单个
claude plugin update code-reviewer

# 升级所有（先dry-run检查）
claude plugin update --dry-run
claude plugin update --all
```

老K的建议：**Major版本升级前一定要看Changelog。**`2.x → 3.x` 可能改掉了你依赖的API。

9.3 私有Plugin市场

企业不能用公共市场——内网隔离、安全审计、自定义API这些东西都在防火墙里。私有市场是必然选择。

搭建私有市场

私有Plugin市场就是一个托管 plugin.json 索引文件的HTTP服务器。最简单的方案：

```
# market_server.py — 最简单的私有市场
from fastapi import FastAPI, HTTPException
from fastapi.responses import FileResponse
import json, os

app = FastAPI()
PLUGIN_DIR = "/data/plugins"

# 索引文件：列出所有Plugin
@app.get("/index.json")
async def index():
    plugins = []
    for name in os.listdir(PLUGIN_DIR):
        pj_path = os.path.join(PLUGIN_DIR, name, "plugin.
        if os.path.exists(pj_path):
            with open(pj_path) as f:
                pj = json.load(f)
                plugins.append({
                    "name": pj["name"],
                    "version": pj["version"],
                    "description": pj.get("description",
                    "author": pj.get("author", {}),
                    "download_url": f"/plugins/{name}/v{p
                })
    return {"plugins": plugins}

# 下载Plugin包
@app.get("/plugins/{name}/v{version}.tar.gz")
async def download(name: str, version: str):
    path = os.path.join(PLUGIN_DIR, name, f"v{version}.ta
    if not os.path.exists(path):
        raise HTTPException(404)
    return FileResponse(path)
```

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=9000)
```

使用私有市场

```
# 注册私有市场
claude plugin registry add mycompany https://plugins.myco

# 从私有市场安装
claude plugin install mycompany/internal-toolkit

# 查看已注册的市场
claude plugin registry list
# 结果:
# official      https://plugins.claude.ai
# mycompany     https://plugins.mycompany.com
```

私有市场安全策略

```
# 私有市场的安全措施清单
security:
  # 1. TLS加密传输
  tls: true

  # 2. Plugin包签名验证
  signature_verification:
    enabled: true
    public_key: "-----BEGIN PUBLIC KEY-----..."

  # 3. 元数据审计
  metadata_scan:
    check_licenses: ["MIT", "Apache-2.0", "Proprietary"]
    block_licenses: ["GPL-3.0"]  # 按公司政策阻止
    scan_permissions: true       # 审计权限申请

  # 4. 代码扫描
  code_scan:
    enabled: true
    tools: ["semgrep", "bandit"]
    on_publish: true
    schedule: "0 2 * * *"
```

9.4 命名空间与多Plugin共存

当你的项目里装了10个Plugin，冲突就来了。两个Plugin都注册了 `/review` 命令怎么办？两个Plugin都提供了 `code-review.md` Skill怎么办？

命名空间机制

```
// Plugin A: "sloth-code-reviewer" 的 plugin.json
{
  "name": "sloth-code-reviewer",
  "namespace": "code-review",
  "capabilities": {
    "commands": [
      {"name": "/review", "handler": "review.py"}  // 实际
    ],
    "skills": ["code-review.md"]  // 实际注册为 code-review
  }
}

// Plugin B: "team-security-audit" 的 plugin.json
{
  "name": "team-security-audit",
  "namespace": "security",
  "capabilities": {
    "commands": [
      {"name": "/review", "handler": "sec_review.py"}  //
    ]
  }
}

// 使用时
// /code-review:review  → 触发 Sloth 的代码审查
// /security:review     → 触发 Team 的安全审查
// /review              → 如果有歧义，提示用户选择
```

多Plugin共存策略

```
// claude.json — 项目级Plugin配置
{
  "plugins": {
    // 按命名空间组织
    "code-review": {
      "plugin": "sloth-code-reviewer",
      "version": "1.3.0",
      "enabled": true,
      "config": {
        "severity_threshold": "high"
      },
      "scope": ["src/backend/"]  // 只在backend目录生效
    },
    "testing": {
      "plugin": "team-test-runner",
      "version": "2.0.1",
      "enabled": true,
      "config": {
        "framework": "pytest"
      },
      "scope": ["tests/"]
    },
    "docs": {
      "plugin": "auto-doc-generator",
      "version": "1.0.0",
      "enabled": true,
      "scope": ["src/", "docs/"]
    }
  }
}
```

冲突解决规则

表 10–4

冲突类型	默认行为	自定义方式
命令重名	带命名空间前缀 `/ns:cmd`	在claude.json里设 `alias`
Skill重名	带命名空间前缀	设置优先级 `priority`
Tool重名	先安装的优先	设置 `override: true`
配置项重名	各自独立	在各自namespace下配置
Hook冲突	按优先级+安装顺序执行	设置 `hook_order`

```
// 自定义冲突解决
{
  "plugins": {
    "my-reviewer": {
      "plugin": "sloth-code-reviewer",
      "alias": {"commands": {"/review": "/cr"}},  // 重命
      "priority": 10  // 数字越大优先级越高
    }
  }
}
```

9.5 实战：构建一个完整的Plugin

让我们构建一个完整的Plugin，把前面几章学到的技能打包。

目录结构

```
cc-code-reviewer/
├── plugin.json              # Plugin元数据
├── README.md                # 用户文档
├── CHANGELOG.md             # 版本变更
├── skills/
│   ├── code-review.md       # 审查Skill
│   └── security-audit.md    # 安全审计Skill
├── tools/
│   ├── complexity.py        # 复杂度分析工具
│   └── diff_analyzer.py     # Diff分析工具
├── commands/
│   ├── review.py            # /review 命令处理
│   └── security.py          # /security 命令处理
├── hooks/
│   └── post_tool_use.py     # PostToolUse钩子
├── scripts/
│   ├── setup.py             # 安装脚本
│   └── cleanup.py           # 卸载脚本
├── tests/
│   ├── test_complexity.py
│   └── test_diff.py
└── .github/
    └── workflows/
        └── publish.yml       # 自动发布到私有市场
```

plugin.json

```
{
  "name": "cc-code-reviewer",
  "version": "1.3.0",
  "description": "企业级代码审查工具包",
  "author": {"name": "树懒老K", "email": "lao.k@example.co

  "capabilities": {
    "skills": [
      "skills/code-review.md",
      "skills/security-audit.md"
    ],
    "tools": [
      {
        "name": "analyze_complexity",
        "description": "分析代码圈复杂度",
        "handler": "tools/complexity.py"
      }
    ],
    "commands": [
      {"name": "/review", "handler": "commands/review.py"
      {"name": "/security", "handler": "commands/security
    ],
    "hooks": ["PostToolUse"]
  },

  "permissions": {
    "filesystem": ["read"],
    "commands": ["git", "grep", "wc"]
  },

  "config": {
    "severity_threshold": {"type": "string", "default": "
```

```
    "max_complexity": {"type": "integer", "default": 15},
    "ignore_patterns": {"type": "array", "default": ["**/
  }
}
```

自动发布CI

```
# .github/workflows/publish.yml
name: Publish Plugin

on:
  push:
    tags: ['v*']

jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Validate plugin.json
        run: |
          python -c "
          import json
          with open('plugin.json') as f:
              data = json.load(f)
          assert 'name' in data
          assert 'version' in data
          print(f'Valid: {data[\"name\"]}@{data[\"version
          "

      - name: Run Plugin Tests
        run: python -m pytest tests/

      - name: Scan for secrets
        uses: zricethezav/gitleaks-action@v2

      - name: Package Plugin
        run: tar -czf cc-code-reviewer.tar.gz .
```

```
      - name: Upload to Private Market
        run: |
          curl -X POST https://plugins.mycompany.com/publ
            -H "Authorization: Bearer ${{ secrets.MARKET_
            -F "package=@cc-code-reviewer.tar.gz" \
            -F "version=$(git describe --tags)"
```

9.6 常见陷阱

表 10–5

陷阱	症状	解决方案
版本不兼容	Plugin安装后无法工作	声明清晰的依赖版本范围
权限过度申请	安全审计不通过	只申请最小必需权限
配置项无默认值	Plugin装完不能直接用	每个配置项必须有默认值
更新破坏CI	自动升级后流水线失败	CI里锁死版本号
命名冲突	两个Plugin的 `/review` 冲突	使用命名空间
安装脚本卡死	Plugin安装无响应	安装脚本不要读stdin

本章小结

Plugin生态是Claude Code从”个人工具”走向”团队平台”的关键基础设施。plugin.json定义身份，安装来源决定分发渠道，私有市场保障企业安全，命名空间解决多Plugin共存。

老K的Plugin三原则：**1) 一个Plugin一个职责 2) 只申请最小权限 3) CI中永远锁版本**。

下一章进入新前沿——Claude Code最新推出的/goal自主Agent、Dynamic Workflows、Remote Control等新兴能力。

第11章 新兴能力：Claude Code的进化前沿

【决策者摘要】

Claude Code的进化速度惊人——从最初的命令行助手到今天的Agent平台。本章聚焦六大新兴能力：/goal自主目标Agent（让Claude自己判断"做完没有"）、Dynamic Workflows流程编排（声明式多步骤任务链）、Routines定时调度（把Claude当作定时任务引擎）、Remote Control远程会话接管、Artifacts输出物分享、以及Deep Links深度链接启动。这些能力共同构建了"零人值守"的Agent化开发体验。

10.1 /goal：目标驱动的自主Agent

概念

`/goal`是Claude Code最具革命性的新能力之一——你的角色从"指挥每一步怎么做"变成了"设定完成标准，让Claude自己判断什么时候做完"。

传统模式下，你跟Claude的交互是逐轮推进的：你提出一个问题，Claude给出一个回答，你检查结果，再提下一个问题。在/goal模式下，你一次性设定完成条件，Claude在多个轮次中持续检查是否已达到条件，直到满足为止——不再需要你每次都说"继续"。

工作机制

```
用户输入: /goal "所有测试通过，lint没有error"
          ↓
Claude 开始工作:
  Round 1: 读取当前代码状态 → 修改 → 运行测试 → 3个测试失败
  Round 2: 修复失败的测试 → 运行lint → 发现1个warning
  Round 3: 修复lint warning → 运行完整测试套件 → 42/42 passe
  Round 4: 检查lint → 0 warnings ✓
          ↓
/goal 完成条件已满足 → 停止
```

适用场景

表 11–1

场景	示例 /goal
CI修复	`/goal "CI流水线全部通过"`
代码迁移	`/goal "所有import路径迁移到新结构，构建无错误"`
重构	`/goal "提取公共逻辑到utils模块，单元测试90%覆盖率"`
依赖更新	`/goal "升级所有依赖到最新稳定版，测试全部通过"`
Bug修复	`/goal "修复issue #42中描述的3个bug，添加回归测试"`

与Headless模式的对比

表 11–2

	/goal	Headless (—print)
目标定义	自然语言完成条件	单次prompt
迭代次数	自动多轮	单轮
用户参与	零（设置后无需干预）	需调用循环
适用	目标明确但实现路径不确定	单次确定性任务

实战

```
# 修复CI中所有失败的测试
claude goal "Fix all failing tests in the CI pipeline"

# 自动升级项目依赖
claude goal "Update all npm dependencies to latest compat
```

10.2 Dynamic Workflows：声明式流程编排

概念

Dynamic Workflows是Claude Code的流程编排引擎——你可以声明一个多步骤的自动化流程，Claude按步骤执行，处理条件分支和错误重试。

多步骤任务链

```
<span style="font-size:0.5pt;">§TOC10-4§</span>
## Workflow: PR Review Pipeline

### Step 1: Checkout
- 检查PR分支是否存在
- 合并最新main到PR分支

### Step 2: Review
- 运行`claude review-pr`
- 检查安全问题和性能隐患

### Step 3: Test
- 运行`npm test`
- If tests fail → Go to Step 4 (Fix)
- If tests pass → Go to Step 5 (Merge)

### Step 4: Fix
- 分析测试失败原因
- 自动修复后返回Step 3

### Step 5: Merge
- 合并PR到main
- 通知Slack频道
```

条件分支与错误重试

Workflows支持三种控制流： – **条件分支** — `If X then Y else Z` – **循环重试** — 失败后自动回退到前序步骤 – **并行执行** — 独立步骤同时运行

10.3 Scheduled Tasks & Routines：让Claude替你值守

Routines

Routines是Claude Code的定时任务系统——你可以设定”每天早上9点运行代码质量分析”或”PR合并后触发自动化测试”。

```
# .claude/routines/release-check.yml
name: Release Health Check
schedule: "0 10 * * 1"  # 每周一上午10点
tasks:
  - goal: "检查所有依赖的CVE漏洞，生成安全报告"
  - goal: "对比上周的生产日志，发现异常模式"
```

常见定时场景

- 📅 **每日站会前** — 自动汇总昨天的代码变更和测试结果
- 🔒 **每周五** — 安全漏洞扫描并生成报告
- 📦 **版本发布前** — 全面回归测试 + changelog自动生成

10.4 Channels：外部事件驱动Claude

Channels让其他平台的事件能推送到Claude Code会话中——来自Telegram的消息、GitHub的issue事件、Slack的@mention，都能触发Claude执行任务。

```
外部事件 → Channel → Claude Code会话 → 执行任务 → 返回结果
```

- **Telegram/Discord** — 让Claude回答群聊中的问题
- **GitHub** — 新的issue自动触发代码分析
- **iMessage** — 从手机上让Claude跑一个任务
- **Webhooks** — 自建系统的任意事件

10.5 Remote Control：远程接管会话

Remote Control让你可以在任何设备上接管正在运行的Claude Code会话——在桌面启动一个/goal任务，回家的地铁上在手机上检查进度，根据结果调整方向。

10.6 Artifacts 与 Deep Links

Artifacts — 把Claude Code会话中的输出物（代码片段、架构图、测试报告）分享为独立链接，团队成员可以查看、评论甚至fork。

Deep Links — 从URL直接启动Claude Code会话并传入初始上下文，适合CI系统、Slack bot、项目管理工具的集成。

10.7 新兴能力选型指南

表 11–3

你的需求	推荐方案
一个明确目标，Claude自己判断”做完”	/goal
多步骤带条件分支的流程	Dynamic Workflows
定期自动执行的任务	Routines / Scheduled Tasks
外部平台的@mention触发任务	Channels
在不同设备间切换任务	Remote Control
分享Claude的输出给团队	Artifacts
从链接直接启动Claude会话	Deep Links

10.8 本章小结

1. /goal让Claude从”对话助手”变为”目标驱动Agent”——你设条件，它自己迭代到你满意
2. Dynamic Workflows提供了声明式的多步骤任务编排
3. Routines/Channels实现了”零人值守”的自动化开发体验
4. Remote Control/Artifacts/Deep Links完善了协作和分发

思考题

1. 你的团队中最适合用/goal自动化的工作流是什么?
2. Channels接入Slack后，如何防止误触发的安全问题?
3. 你觉得Workflows和CI/CD Jenkins Pipeline有什么区别和互补之处?

第12章 工程化落地：从个人到团队

【决策者摘要】

Claude Code从个人提效工具变成团队基础设施，需要在四个维度上完成工程化落地：成本控制（理解Token定价、模型选择策略、Prompt缓存）、调试（打开黑盒的能力）、安全（最小权限是底线）、团队文化（共享配置、治理规范、渐进式能力建设）。本章提供每一维度的实操指南，尾声给出SDD四层生态的全局视图。

11.1 Token成本控制

理解定价结构

Claude Code的消耗不是按月订阅，而是按实际使用的Token计费。一次完整对话的成本由三部分组成：

表 12–1

组成部分	占比	影响因素
输入Token	~70%	上下文大小（代码库、对话历史）
输出Token	~20%	Claude生成的回答长度
工具调用	~10%	搜索、读写文件、命令执行

模型选择策略

表 12-2

场景	推荐模型	原因
简单修改（改名、重构小函数）	Claude 4 Sonnet	快、便宜、够用
复杂架构设计、多文件改动的分析	Claude 4 Opus	推理能力强
CI/CD自动发现	Haiku	速度优先，输出可控

策略：**热点路径用Opus，常规操作用Sonnet，自动化扫描用Haiku。** 不要让Opus去重命名变量。

Prompt缓存

Claude Code默认会缓存频繁重复的上下文——CLAUDE.md、Skills定义、项目结构。利用这一点，把大块的上下文结构化：

```
# CLAUDE.md
<span style="font-size:0.5pt;">§TOC11-3§</span>
## 项目结构（缓存命中）
/src/app/      - 主应用
/src/lib/      - 工具库
/src/tests/    - 测试
```

相同的环境描述每次对话都命中缓存，大幅减少输入Token。

成本监控

```
# 查看当前会话Token消耗
claude status

# 查看历史消耗
claude cost --last 7d
```

11.2 调试：打开黑盒

Claude不是完美的——它会犯错、会产生幻觉、会走弯路。但它的优势是：每一步都是可追溯的。

三板斧

1. `--debug` 模式

```
claude --debug "修复测试失败"
```

调试模式会输出每次工具调用的完整输入输出——你能看到Claude读了哪个文件、执行了什么命令、返回了什么结果。这是排查Claude行为异常的第一步。

2. stream-json：实时透视

```
claude --output-format stream-json "分析代码质量"
```

流式JSON输出让你能在Claude运行时实时监控它的思考过程——每个工具调用、每个推理步骤、每个中间结果都能被外部系统解析和处理。

3. PostToolUse Hook：构建持久化审计日志

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "",
        "command": "echo \"[$(date -Iseconds)] $CLAUDE_TO
      }
    ]
  }
}
```

每次工具调用后自动写入审计日志——事后可以回溯Claude的每一步操作。

异常行为诊断

表 12-3

现象	可能原因	诊断方法
Claude反复读同一个文件	记忆中没有代码位置信息	检查CLAUDE.md是否声明了文件用途
工具调用频繁失败	环境变量缺失或命令不存在	用 `--debug` 看错误输出
Claude给出不一致的建议	Skills描述冲突	检查多个Skills是否有互斥的allowed-tools

11.3 安全准则：最小权限是底线

权限模式

Claude Code提供三种内置权限模式：

表 12–4

模式	描述	适用
`default`	可读写项目内文件，需确认危险命令	日常开发
`acceptEdits`	自动接受编辑，手动确认命令	受控CI环境
`bypassPermissions`	完全信任	隔离沙箱

铁律：除非在完全隔离的沙箱中，不要使用bypassPermissions。

工具权限精细化控制

```
{
  "permissions": {
    "allow": [
      "Bash(npm test:*)",
      "Bash(git diff:*)",
      "Bash(git status:*)"
    ],
    "deny": [
      "Bash(rm -rf:*)",
      "Bash(curl:*)",
      "Bash(ssh:*)",
      "Bash(sudo:*)"
    ]
  }
}
```

精确到命令参数的权限控制——Claude可以跑测试但不能删文件，可以看git状态但不能curl外部URL。

API密钥生命周期管理

1. **不要硬编码** — 所有密钥通过环境变量注入
2. **定期轮换** — API密钥每90天更换
3. **权限最小化** — 每个密钥只授予必要的最小权限
4. **审计日志** — 所有密钥使用记录可追溯

11.4 大型代码库策略

层次化CLAUDE.md

对于百万行代码的单体仓库，一个CLAUDE.md不可能说清楚全部。采用层次化结构：

```
project/
├── CLAUDE.md                # 项目级：全局背景、约定、工具
├── src/
│   ├── CLAUDE.md            # 模块级：API边界、依赖
│   ├── auth/
│   │   └── CLAUDE.md        # 子模块：认证逻辑的具体实现说明
│   └── payment/
│       └── CLAUDE.md        # 子模块
```

Claude Code会自动合并上下文范围内所有CLAUDE.md文件，但只加载当前任务相关的部分——这是降低上下文成本的核心策略。

引导搜索，而非全量阅读

好的CLAUDE.md不是列出所有文件的用途，而是告诉Claude"去哪找"：

```
<span style="font-size:0.5pt;">§TOC11-7§</span>
## 关键文件
- 认证中间件: [/src/auth/middleware.ts](src/auth/middleware
- 路由定义: [/src/routes/](src/routes/)
- 测试放在对应源码的`sibling/__tests__`目录
```

让Claude自己去读文件，而不是把所有代码都塞进上下文。

11.5 团队落地

共享配置的治理

```
company-claude-config/
├── CLAUDE.md           # 公司级编码规范（所有项目继承）
├── skills/             # 共享Skills库
│   ├── code-review/
│   ├── deploy/
│   └── setup-jira/
├── hooks/              # 安全Hooks
│   └── block-dangerous-commands/
└── .claude/
    └── settings.json   # 团队统一设置
```

渐进式能力建设路线图

表 12–5

阶段	时间	目标
入门	第1–2周	每个人会用Claude Code做代码审查和简单修改
熟练	第3–4周	掌握Skills和自定义Hooks
团队化	第2–3月	建立共享配置库，使用MCP和Agent Teams
平台化	第4–6月	CI/CD集成、Channels、Routines自动化

新人培训计划

1. **Day 1：安装 + 第一个对话** — 让新人用Claude Code修一个真实的bug
2. **Day 2–3：CLAUDE.md写作** — 为自己的项目写CLAUDE.md
3. **Week 2：Skills和Hooks** — 建立3–5个常用Skill
4. **Week 3–4：Agent模式和CI/CD** — 把Claude Code接入CI流水线

11.6 SDD四层生态

SDD（Spec-Driven Development）生态是Claude Code工程化的全景视图：

每一层都有明确的责任边界、扩展机制和最佳实践。从第1层到第4层，是你从”用Claude写代码”到”让Claude替你写代码”再到”让Claude替你管代码”的演进路径。

11.7 本章小结

1. Token成本控制的核心是模型选择+Prompt缓存+上下文管理
2. 调试三板斧：—debug打开操作日志，stream-json实时监控，PostToolUse持久化审计

3. 最小权限是安全的底线——精确到命令参数的权限控制
4. 团队落地的关键是共享配置治理+渐进式能力建设

思考题

1. 你的团队每月Claude Code的Token消耗是多少？如何优化？
2. 如果要设计一个”零人值守”的CI/CD流程，哪些步骤可以移交给Claude Code？哪些不能？
3. SDD四层生态中，你的团队现在在哪一层？下一步应该怎么走？

树懒老K（拙一）

30年企业服务经验 · 专注AI智能体与组织变革

个人网站　　个人微信　　公众号

慢一点，深一度