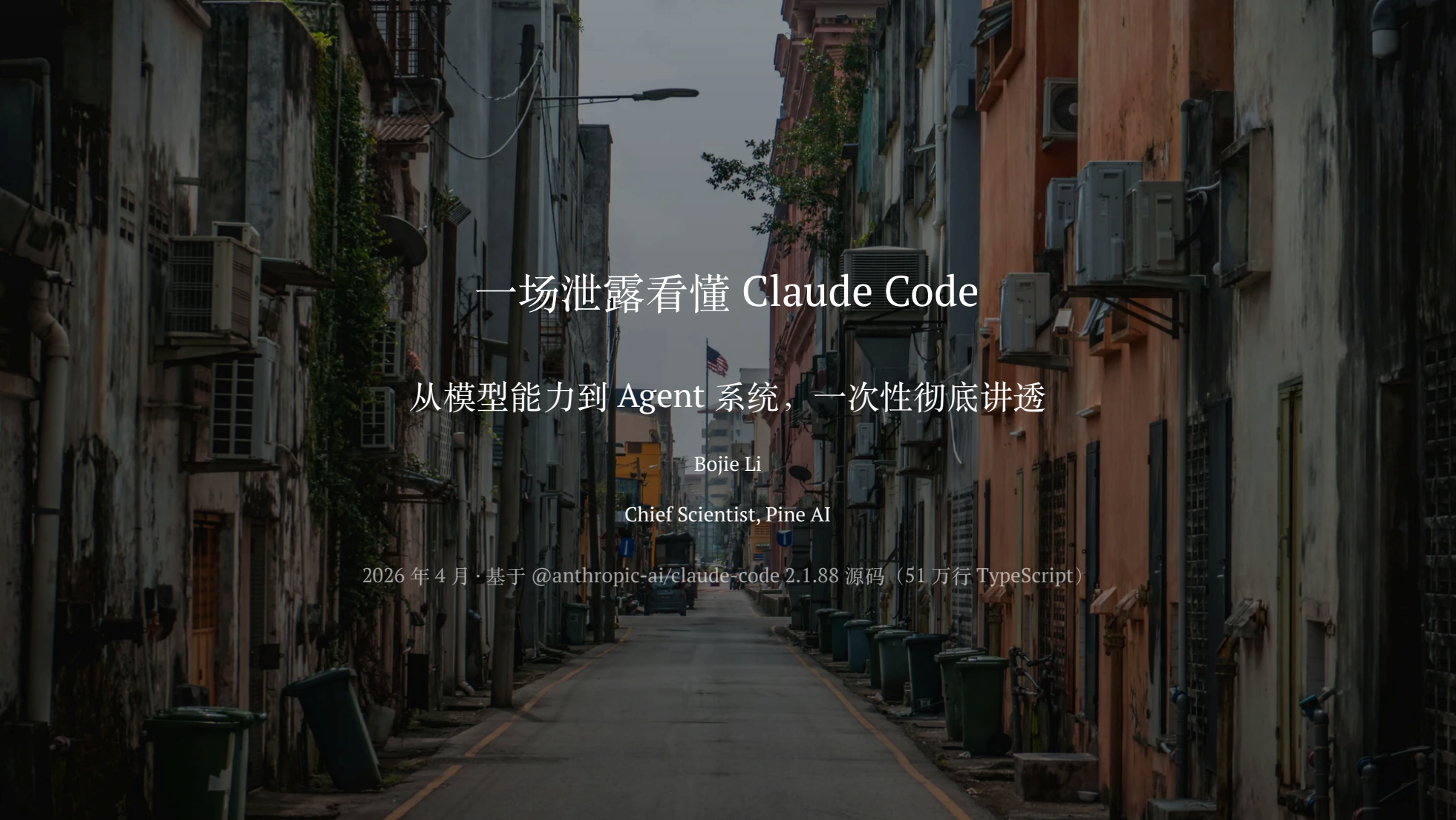

一场泄露看懂 Claude Code

从模型能力到 Agent 系统，一次性彻底讲透

Bojie Li

Chief Scientist, Pine AI

2026 年 4 月 · 基于 @anthropic-ai/claude-code 2.1.88 源码（51 万行 TypeScript）

先聊个有趣的：源码里藏着一个完整的宠物扭蛋机

2026/4/1，Claude Code 源码通过 npm 包泄露。大家在里面发现了一个完整的宠物系统。

Buddy 系统：你的 CLI 宠物

输入 `/buddy` 就能"孵化"一只专属宠物——完整的扭蛋系统。

```
const SALT = 'friend-2026-401'

export function roll(userId: string): Roll {
  const key = userId + SALT
  // hash + 种子 PRNG → 确定性生成
  return rollFrom(mulberry32(hashString(key)))
}
```

每个用户的宠物由 `userId + SALT` 确定性生成——每次打开都是"你的"宠物。

扭蛋属性系统

维度	设定
物种	18 种（duck, goose, cat, dragon, capybara...）
稀有度	common(60%) → uncommon(25%) → rare(10%) → epic(4%) → legendary(1%)
属性	DEBUGGING · PATIENCE · CHAOS · WISDOM · SNARK（1-100）
外观	6 种眼型、8 种帽子、1% 闪光概率、3 帧 ASCII 动画

一个 51 万行的生产级 AI Agent 里，藏着一个如此用心的宠物系统。但仔细看代码，有几个地方很耐人寻味——

代码里的几个"巧合"

这个宠物系统的代码里，有两处让人忍不住多想的细节：

物种名全部用 hex 编码——为什么？

```
// types.ts — 18 种全部用 hex 编码
const c = String.fromCharCode
export const duck = c(0x64,0x75,0x63,0x6b)
export const cat  = c(0x63,0x61,0x74)
export const capybara = c(
  0x63,0x61,0x70,0x79,0x62,0x61,0x72,0x61
)  // → 解码出来就是 'capybara'
```

注释说：**capybara 碰撞了某个模型的内部代号**。为了不让它特别突出，所有 18 种都统一 hex 编码——"so one doesn't stand out"。

但 capybara 正好是此前被泄露的 Anthropic 新模型名字。

时间窗口写在了代码里

```
// Sustained Twitter buzz instead of a
// single UTC-midnight spike.
// Teaser window: April 1-7, 2026 only.
export function isBuddyTeaserWindow() {
  const d = new Date()
  return d.getFullYear() === 2026
    && d.getMonth() === 3 && d.getDate() <= 7
}
```

"**Sustained Twitter buzz**"——这不像工程师对内部功能的描述，更像营销策划的用语。

而且这个时间窗口精确到 **April 1-7, 2026**——源码泄露的日期正好是 4 月 1 日。

加上 SALT = `'friend-2026-401'`（friend + 2026年4月1日）——这些真的都是巧合吗？

所以——这场泄露真的是巧合吗？

这个宠物系统里有四个耐人寻味的细节：

四个证据

证据一：SALT = `'friend-2026-401'` ——friend + 2026 年 4 月 1 日。泄露日期精确到天。

证据二：Teaser Window 精确到 **April 1-7, 2026**。注释写的是 `"Sustained Twitter buzz"` ——这不像是工程师对内部功能的描述，更像是营销策划的用语。

证据三：18 个物种名全部 hex 编码，原因是 **capybara 碰撞了下一代模型的内部代号**——而 capybara 正好是此前被泄露的新模型名字。

证据四：统一 hex 编码，"so one doesn't stand out"——反而让每个逆向工程者都去解码了。

三种可能的解读

A. 纯巧合 (10%)：Buddy 是计划中的愚人节彩蛋，source map 是配置失误，碰巧同一天。需要相当大的巧合。

B. 技术团队"不小心" (55%)：有人在那次构建中"不小心"开启了 source map。法务发 DMCA 是真实的应激反应，但十几个小时的窗口期已经足够代码传遍全球。Buddy 彩蛋是**提前埋好的引爆物**。

C. 其他可能：完全意外但事后默许（20%），或公司策划（15%）。

不管答案是什么，结果是一样的：全球开发者免费做了一次深度代码审查和口碑传播。这可能是 2026 年最成功的技术营销，无论是否有意为之。

今天要聊什么

第一部分：Harness Engineering 与 Claude Code

从 Demo 到生产的真正距离
Harness Engineering：行业最火的 Agent 工程范式
Claude Code 全局架构与 React CLI

第二部分：Environment — 让 Agent 能干事

Prompt Cache 是架构约束，不是优化
五层上下文压缩管线
记忆架构与 Dream 系统

第三部分：约束与验证 — 让 Agent 不出错

Fail-closed 安全哲学与多层权限系统
工具编排与投机执行
多 Agent 的能力分区

第四 & 五部分：纠正 + 工程化实践

错误恢复、熔断器与死亡螺旋防护
消融实验、隐私工程与防泄漏
宠物彩蛋的秘密

第一部分：Harness Engineering 与 Claude Code

Agent 工程的核心不在"让模型调用工具"，而在工具调用之后的一切

从 Demo 到生产的距离，比大多数人想象的远得多

绝大多数公开讨论的两个极端

入门级："让模型调工具"——ReAct 循环的入门教程，五分钟搭个 Agent

宏大叙事级："AGI 即将到来"——对 Agent 未来的畅想

中间的空白：一个真正服务百万用户的 Agent 产品，在工程层面到底在解决什么问题？——几乎没有人讲清楚过

Claude Code 源码泄露的价值

不是实验室原型，而是日活用户庞大的商业产品。51 万行 TypeScript，完整工程实现。

真正的工程量在"工具调用之后"

问题	工程量
权限怎么判？	LLM 分类器 + 规则 + 熔断器
工具出错了？	消息扣留 + 静默升级 + 多轮接续
上下文太长？	五层管线按保质期分别处理
多工具同时跑？	并发安全标记 + 错误级联
缓存怎么共享？	CacheSafeParams + 字节级一致
用户还没输入？	投机执行 + 覆盖层文件系统
构建产物泄密？	字符串黑名单 + 编译时 DCE
系统休眠了？	防休眠 + 自愈子进程

核心观点：这些"之后"的工程，才是从 Demo 到生产的真正门槛。

Harness Engineering：2026 年最火的 Agent 工程范式

演进路线

阶段	关注点	说明
Prompt Engineering	问什么	优化输入给模型的指令
Context Engineering	看什么	系统性管理模型能看到的信息
Harness Engineering	整个系统	模型运行的全部基础设施

三者是层层包含关系：Prompt ⊂ Context ⊂ Harness。

核心观点

模型能力趋于商品化，**竞争优势正在转移到模型之外的工程实践**。Harness 是模型运行的全部基础设施——上下文怎么给、工具怎么调、出错怎么恢复、安全怎么保障。

关键案例

LangChain Terminal Bench 2.0：不换模型，只改 Harness，准确率 52.8% → 66.5%（排行榜 30 名外 → 前 5）。

OpenAI 内部：3 名工程师用 Agent + 合理 Harness，5 个月完成约百万行代码、~1500 个 PR。

Claude Code：51 万行 TypeScript，其中绝大部分代码不是在做"让模型调工具"，而是在做工具调用之后的一切——上下文管理、安全约束、错误恢复、缓存经济学。**这就是 Harness Engineering 的实战样本。**

Agent 的核心公式

📖 书第一章：我们在《AI Agent 实战营》里建立的框架，在 Claude Code 里得到了完整验证

一个 Agent 的三件事

层面	取决于	类比
智力——能不能想明白	**Model**	大脑
能力——能不能干成事	**Environment**	手脚
靠谱——会不会干错事	约束/验证/纠正	缰绳

```
Agent = Model + Harness
```

Harness 包含 Environment + 约束/验证/纠正——**模型之外的一切**。

Environment（手脚）：上下文、工具、记忆——让 Agent 能干事

约束/验证/纠正（缰绳）：安全、权限、错误恢复——让 Agent 靠谱地干事

Claude Code 中的 Harness 全貌

Environment：40 个内置工具、五层上下文压缩、Prompt Cache、CLAUDE.md 记忆、Dream 巩固、Side Query 并行、投机执行

约束：Fail-closed 默认值、工具白名单、Shell 语义解析、Undercover Mode

验证：LLM 权限分类、Hook 系统、五层权限判断

纠正：熔断器、消息扣留、模型降级、死亡螺旋防护

后面的内容按这个结构组织：Part 2 讲 Environment，Part 3 讲约束与验证，Part 4 讲纠正。

ReAct 循环：Agent 的基本运行模式

📖 书第一章：`<think>` → `<tool_call>` → `<tool_response>` 的迭代循环

最简形式

```
while True:
    response = LLM(messages, tools)

    if response.has_tool_calls:
        for tool in response.tool_calls:
            result = execute(tool)
            messages.append(result)
        continue
    else:
        return response.content
```

看起来很简单——但 **Claude Code** 的 **query.ts** 把这个循环写了 **1700** 行。

从简单到复杂的距离

简单版	Claude Code 版
一个 while 循环	7 个命名 continue 分支的状态机
出错就报错	静默升级 → 多轮接续 → 消息扣留
工具顺序执行	流式并行 + 并发安全标记
无上下文管理	五层压缩管线
无安全检查	五层权限判断 + 熔断器
无缓存考虑	Cache 经济学贯穿全局

这就是今天要讲的全部内容——Claude Code 是如何把这个 10 行的 ReAct 循环，变成一个 51 万行的生产级系统的。

Claude Code 的全局架构

一个完整的生产级 **Coding Agent** 是什么样的?

核心组件

组件	作用	对应代码
主循环	1700 行 while(true) 状态机	`query.ts`
Side Query	并行辅助 LLM 调用	`sideQuery.ts`
工具系统	40 个内置工具 + buildTool 工厂	`Tool.ts` + `tools/`
上下文压缩	五层管线	`services/compact/`
安全系统	权限分类 + Hook	`hooks.ts`
投机执行	覆盖层文件系统	`speculation.ts`
记忆系统	Dream 离线巩固	`autoDream/`
Feature Flag	编译时 + 运行时	`cli.tsx`
防泄漏	Undercover + Canary	`undercover.ts`
防休眠	caffeinate 自愈	`preventSleep.ts`

七个基础工具 = 完备能力

📖 书第五章：只需七个工具，Coding Agent 就能完成几乎任何任务

工具	功能
Read / Write / Edit	文件操作
Glob / Grep	文件发现与内容搜索
Bash	Shell 命令执行
Agent	子 Agent 调度

文件系统是 Agent 的交互总线

📖 书第五章：Anthropic 的核心观点——所有信息都通过文件来持久化、迭代和版本控制。所有通用 Agent 归根结底都是 Coding Agent。

Claude Code 源码完美印证了这一点：记忆用 Markdown 文件、配置用 CLAUDE.md、工具结果存磁盘、投机执行用覆盖层文件系统。

架构设计模式：用 React 写 CLI

Claude Code 不是传统 CLI——它用 React（Ink）渲染整个终端界面

关键事实

- 源码中包含 **552 个** `.tsx` 文件
- 使用 **vendored Ink**（Facebook 的 React CLI 框架）
- 入口文件是 `cli.tsx`——注意后缀
- 整个 UI 层都是 React 组件树

为什么选 React 来做 CLI？

传统 CLI	React CLI (Ink)
手动管理输出缓冲区	声明式 UI，状态驱动渲染
状态和渲染耦合	状态/渲染天然分离
交互逻辑难复用	组件化复用
难以做复杂布局	Flexbox 布局模型
测试困难	可用 React Testing Library

对 Agent 的意义

流式输出 + 多工具并行执行 + 权限弹窗 + 进度展示——这些都是高度动态的 UI 需求。

传统 CLI 写一个 spinner 就够折腾。Claude Code 需要同时显示：LLM 流式输出、多个工具执行状态、权限确认对话框、文件 diff 预览。只有声明式框架能优雅处理。

冷启动优化

```
// cli.tsx 入口：--version 零导入路径
if (process.argv.includes('--version')) {
  process.stdout.write(version + '\n')
  process.exit(0)
}
// 其他模块在此之后才 import
```

React 框架的启动成本不低。Claude Code 用"零导入快速路径"让 `--version` 毫秒级返回，不加载任何 React 代码。

第二部分：Environment — 让 Agent 能干事

Prompt Cache、上下文压缩、记忆系统——给 Agent 足够的感知、行动和学习能力

Prompt Cache：不是优化，是第一天就要考虑的架构约束

如果只能从 **Claude Code** 源码中学一条原则，我选这条

缓存边界写进了系统提示的物理结构

系统提示里有一个显式的 `SYSTEM_PROMPT_DYNAMIC_BOUNDARY` 标记：

```
[全局稳定内容 - 跨用户可缓存]
--- DYNAMIC BOUNDARY ---
⚠️ WARNING: Do not remove or reorder
   without updating cache logic
[会话特定内容 - 不缓存]
```

系统提示词的组织结构首先由缓存边界决定，其次才是语义逻辑。

工具结果也在为缓存让路

超出阈值的工具输出存磁盘，替换决策被**冻结**——同一条消息在不同时刻必须产生完全相同的字符串，否则 cache 就废了。

Fork Agent 的缓存共享

```
CacheSafeParams = {
  systemPrompt,    // 字节级一致
  userContext,      // 字节级一致
  tools,            // 字节级一致
  messagePrefix,    // 字节级一致
  thinkingConfig    // 也会影响 cache key!
}
```

11 个 fork 调用者（压缩、记忆提取、投机执行、摘要生成……）都走这套缓存共享。

为什么这很重要？

📖 书第二章：Agent 输入输出比 100:1，核心原则是最大化 KV 缓存命中率。稳定的提示前缀、Append-only 上下文、确定性序列化。

Cache 命中 vs 未命中 = **成本和延迟的量级差别**。Prompt caching 不是微优化，它决定了消息怎么序列化、子 agent 怎么分叉、工具结果怎么存储。

缓存经济学的全局影响

每一个架构决策都在围绕"不要破坏 **cache**"做设计让步

受缓存约束的设计决策

设计点	缓存考量
系统提示分段	全局/动态边界分离
工具定义位置	放在系统提示中稳定缓存
Fork agent 参数	CacheSafeParams 字节级一致
工具结果截断	冻结替换字符串
Thinking config	maxOutputTokens 也影响 cache key
消息序列化	确定性 JSON key 顺序
Turn 边界保存	全局 slot 供 post-turn fork 复用

📖 书第二章：KV Cache 的系统性分析

- Chat Template → KV Cache 的物理映射
- Attention mask 与 prefix caching 的关系
- 工具描述放在 user 侧 vs system 侧的缓存影响

📖 书第八章：工具定义与 KV 缓存

> 工具 schema 放在用户消息侧（system prompt 外），以保持 system KV cache 的稳定性。工具数量变化不会破坏全局缓存。

实践启示

第一天就画缓存边界图：哪些内容全局稳定（工具定义、基础规则），哪些会话级稳定（对话前缀），哪些每轮变化（最新消息）。

所有子 **agent fork** 都要走缓存共享：不能各自独立构造 system prompt，否则每个 fork 都会创建新的 cache entry。

消息内容必须确定性序列化：同一条消息在任何时刻、任何路径上序列化的结果必须完全一致。否则 cache 失效。

上下文管理：五层压缩管线

不同类型的信息有完全不同的"保质期"，需要完全不同的处理方式

Layer 1 `Tool Result Budget`	巨量输出存磁盘，模型只看预览 替换决策冻结以保护缓存	轻
`HISTORY_SNIP`	最精细裁剪，纯噪声直接删掉 搜索返回 500 行但只用了 3 行，不值得摘要	轻
Layer 3 `Microcompact`	在 API 缓存层面做编辑（ `cache_edits` ） 本地消息不变，压缩完全在 API 层完成	中
Layer 4 `CONTEXT_COLLAPSE`	旧对话轮次归档为摘要 保留结构——哪一轮做了什么、结论是什么	重
Layer 5 `Autocompact`	最后兜底，先 session memory 再全量 有熔断器：连续 3 次失败后放弃	最重

核心原则：前面能搞定就不触发后面。大部分时候最重的 Autocompact 根本不需要跑。

五层压缩的设计哲学

信息的"保质期"

信息类型	保质期	处理方式
工具原始输出	一轮	Tool Result Budget 截断
搜索中间结果	几轮	HISTORY_SNIP 直接删
历史 tool result	几轮	Microcompact 从 cache 删
对话结构信息	中期	CONTEXT_COLLAPSE 归档
任务目标与约束	全程	Autocompact 保留

📖 书第二章：上下文压缩策略

- 8 段式压缩模板：背景 → 关键决策 → 工具记录 → 意图演进 → 执行结果 → 错误解决 → 未解决 → 后续计划
- System Hint 技术：在关键时刻注入提醒
- Context 是 Agent 的"操作系统"

Autocompact 的熔断器

```
const MAX_CONSECUTIVE_AUTOCOMPACT_FAILURES = 3

// BQ 2026-03-10: 1,279 sessions had 50+
// consecutive failures (up to 3,272) in a
// single session, wasting ~250K API calls/day
```

内部数据驱动的决策：曾经有 1279 个会话出现 50+ 次连续失败，每天全球浪费约 25 万次 API 调用。

📖 书第二章：Context Window 的管理

- Token 计数与估算
- 对话窗口滑动策略
- 压缩触发阈值设计

设计启示：不同保质期的信息需要不同的策略。一种压缩搞不定所有场景，需要一条管线。

Side Query：主循环不是唯一的 LLM 调用者

把"调用 LLM"当成可以到处撒的轻量操作

sideQuery 的定义

```ts
export type SideQueryOptions = {
  model: string
  system?: string | TextBlockParam[]
  messages: MessageParam[]
  tools?: Tool[] | BetaToolUnion[]
  max_tokens?: number  // default: 1024
  maxRetries?: number  // default: 2
  querySource: QuerySource
  thinking?: number | false
  // ...
}

export async function sideQuery(
  opts: SideQueryOptions
): Promise<BetaMessage> {
  // 轻量 API 封装，非流式
  // 自动处理 OAuth、fingerprint、betas
}
```

5+ 类并行辅助调用

调用场景	模型	说明
权限分类	小模型	判断工具调用是否安全
记忆检索	小模型	哪些 CLAUDE.md 与当前任务相关
Tool Use Summary	Haiku	异步总结工具操作（见下）
Agent 摘要	小模型	子 Agent 进度报告
提示建议	小模型	预测用户下一步

Tool Use Summary：藏在推理时间里的摘要

主模型推理时，Haiku 同时总结上一轮的工具操作：

```
主模型推理 (5-30s)
  ├── Haiku 异步总结工具 → Promise<ToolUseSummary>
  └── compact 时用摘要替代原始输出 → 省 token
```

📖 书第四章：异步 Agent 架构——sideQuery 是并行化思想的工程实现。主模型推理期间，权限分类、记忆检索、摘要生成全部并行。

记忆架构：从 Markdown 到 Dream

📖 书第三章：用户记忆与知识库——记忆层级、检索策略、长期记忆

Markdown 记忆：反直觉但极其有效

文件	用途
`CLAUDE.md`	项目级记忆与约定
`AGENTS.md`	Agent 能力与角色描述
`memory/YYYY-MM-DD.md`	按日期归档的交互日志
`MEMORY.md`	核心事实与用户偏好

为什么 Markdown 比向量数据库更好?

透明可编辑：直接打开看 AI 记了什么，记错了直接改——向量数据库做不到

Git 版本控制：每次记忆修改可追溯可回滚

文件系统即交互总线：与 Agent 的其他组件天然集成

Dream：Agent 的"睡眠学习"

```
触发条件：
  ① 距上次巩固 ≥ 24 小时
  ② 期间有 ≥ 5 个会话
  ③ 获取文件锁成功

门控顺序（成本从低到高）：
  时间检查 → 会话扫描 → 文件锁

执行：
  fork agent → 读取近期会话
  → 整理成长期记忆文件
  → 更新 MEMORY.md 索引
```

巩固提示（4 个阶段）

阶段	动作
Orient	ls 记忆目录，读索引
Gather	收集新信号，grep 会话
Consolidate	合并/更新/删除记忆
Prune	修剪索引，保持精简

Dream 系统的工程细节

人类睡觉时大脑巩固记忆，Agent 空闲时巩固会话记忆

巩固提示的核心指令

```
# Dream: Memory Consolidation

You are performing a dream — a reflective
pass over your memory files. Synthesize
what you've learned recently into durable,
well-organized memories so that future
sessions can orient quickly.
```

关键规则：

- 合并新信号到已有主题文件，避免创建近似重复
- 将相对日期转换为绝对日期（"yesterday" → "2026-04-01"）
- 删除被推翻的旧事实
- 索引保持 <25KB、每条 <150 字符

工具约束

Bash 限制为只读命令（ls, find, grep, cat 等），不能写文件、不能修改状态。

源码中的实现

```
const DEFAULTS: AutoDreamConfig = {
  minHours: 24,
  minSessions: 5,
}

// 门控逻辑
function isGateOpen(): boolean {
  if (getKairosActive()) return false
  if (getIsRemoteMode()) return false
  if (!isAutoMemoryEnabled()) return false
  return isAutoDreamEnabled()
}
```

可远程配置：阈值通过 GrowthBook 远程可调（flag: `tengu_onyx_plover`），可以针对不同用户群调整巩固频率。

第三部分：约束与验证 — 让 Agent 不出错

Fail-closed 安全、多层权限、工具编排、投机执行的安全边界——Harness 的核心竞争力

安全：不是功能，而是默认姿态 ← Harness 的核心

Harness 的约束能力：每一个默认值都选择最保守的选项。

Fail-closed 的工具默认值

```
// buildTool() 工厂函数的默认值
TOOL_DEFAULTS = {
  isConcurrencySafe: false,  // 假定不安全
  isReadOnly: false,         // 假定有写操作
  toAutoClassifierInput: "", // 不参与分类
}
```

每一个默认值都选择了更保守的选项：

- 忘了声明只读？当写操作处理，需更严格权限
- 忘了声明并发安全？独占运行
- 忘了声明比错误声明安全得多

📖 书第四章：工具执行的六层安全——输入校验 → 权限控制 → 沙箱隔离 → 执行监控 → 错误恢复 → 审计记录

Shell 安全：语义解析 > 关键词黑名单

Claude Code 有完整的命令语义解析器：

- 每条 git 子命令的每个 flag 做类型化标注
- 处理 `--` 终止符、UNC 路径、复合命令拆解

真实安全案例：

```
git diff -S -- --output=/tmp/pwned
# -S 原标记为 "none"(不带参数)
# 实际 git 行为：-S 强制消费下一个 argv
# 攻击路径：-S → 推进 1 token → -- → 停止检查
#           → --output 未检查 → 任意文件写入
# 修复：-S 改标记为 "string" 类型
```

📖 书第五章：本地 Agent 的"四大致命要素"——访问私有数据 + 暴露于不受信任内容 + 外部通信能力 + 持久性记忆。四者同时满足时，安全风险指数级增长。

LLM 作为权限分类器

用 LLM 判断另一个 LLM 的工具调用是否安全

输入隔离：防御 Prompt Injection

分类器从对话历史中只提取 `tool_use` **block**——不包含助手的自由文本。

```
主对话: [user, assistant_text, tool_use, ...]
          ↓ 过滤
分类器输入: [tool_use blocks only]
```

为什么？ 如果分类器能看到助手的自然语言输出，攻击者可以通过让主模型输出特定文本来误导分类器。只看结构化的工具调用，攻击面小得多。

每个工具控制自己的分类器输入

```
toAutoClassifierInput(toolInput) {
  // 安全无关的工具返回空字符串
  // → 直接跳过分类
  return ""
}
```

只读工具（Read、Grep、Glob）被白名单跳过。

拒绝熔断器：防止 Agent 被卡死

```
连续拒绝 ≥ 3 次   → 回退交互式提问
或 累计拒绝 ≥ 20 → 回退交互式提问
允许一次操作      → 连续计数器归零
Headless 模式     → 直接抛异常中止
```

解决的问题：

- 太松 → 放过危险操作
- 太紧 → Agent 被卡住做不了事
- **LLM 分类 + 熔断器 = 动态折中**

📖 书第四章：Sidecar 安全机制

> 安全检查作为"边车进程"与主 Agent 并行运行，不阻塞主循环但可以否决危险操作。

📖 书第六章：评估系统中的"一票否决"——某些安全红线无论整体得分多高都直接判定为失败。熔断器就是运行时的"一票否决"机制。

权限系统全流程：从工具调用到最终决策

五层判断，每层都有明确的否决权

```
模型请求执行工具
    │
    ▼
① Settings 静态规则  ← alwaysDeny / alwaysAllow / alwaysAsk
    │ deny? → ❌ 直接拒绝
    │ allow? → ✅ 跳过后续
    ▼
② PreToolUse Hook    ← 用户自定义脚本（退出码 2 = 阻止）
    │ deny? → ❌ 拒绝（stderr 反馈给模型）
    │ allow? → 不能绕过 Settings deny
    ▼
③ Tool 自身属性      ← isReadOnly? → 白名单直接放行
    │
    ▼
④ LLM Auto-Classifier ← sideQuery 调用（只看 tool_use block，不看自然语言）
    │ allow? → ✅
    │ deny?  → 进入拒绝计数
    ▼
⑤ 拒绝熔断器         ← 连续 ≥3 次 或 累计 ≥20 次
    │ 触发? → 回退到交互式提问（Headless 模式直接抛异常）
    │ 未触发? → 用最新的 LLM 分类结果
```

Hook 系统：可编程的控制面

不是"跑个脚本"，而是一个结构化的自动化协议

退出码语义

退出码	含义
0	成功，stdout/stderr 不显示
2	阻止工具执行，stderr 展示给模型
其他	给用户显示警告，继续执行

结构化 JSON 协议

Hook 收到 `{tool_name, tool_input, permission_mode}`，可返回 `{permissionDecision, updatedInput, additionalContext}` ——不只能拦截，还能改写工具输入。

层次化权限

Hook `allow` 不会绕过 Settings `deny` ——全局安全策略有最终否决权。

六个生命周期事件

`PreToolUse` · `PostToolUse` · `Stop` · `PreCompact` · `PostCompact` · `SessionStart`

覆盖工具执行、上下文压缩、会话管理三大阶段。

企业级用途示例

- **CI/CD 集成**：PreToolUse 检查是否在受保护分支上操作
- **审计合规**：PostToolUse 记录所有文件修改
- **自定义安全策略**：阻止特定命令或文件路径

📖 书第四章：Hook 是工具层面的"事件驱动"——PreToolUse 对应审批流，PostToolUse 对应结果后处理。

工具编排：StreamingToolExecutor 全貌

不等模型说完，前面的工具就已经在跑了

流式执行 + 并发安全标记

```
模型流式输出：[Read(a)，Read(b)，Write(c)，Read(d)]
                ↓ 按 isConcurrencySafe 分批
Batch 1 (并行)：Read(a) + Read(b)  ← 只读，并行
Batch 2 (串行)：Write(c)           ← 写操作，独占
Batch 3 (并行)：Read(d)            ← 只读，并行
```

工具状态机

每个工具经历四个状态：`queued → executing → completed → yielded`

- `queued`：已解析出 JSON 参数，等待执行
- `executing`：正在运行
- `completed`：执行完毕，结果已缓冲
- `yielded`：结果已交给主循环

输出顺序严格按接收顺序（不是完成顺序），防止上下文错乱。

三层 Abort Controller 层级

```
toolUseContext.abortController  ← 顶层（整个 turn）
    │
    └── siblingAbortController  ← 中层（同级错误传播）
            │
            └── per-tool AC     ← 底层（单个工具）
```

Bash 错误级联：Bash 出错 → `siblingAbortController.abort()` → 所有同级工具立即终止。但不会 abort 顶层——主循环继续运行，因为模型需要看到错误并决定下一步。

Read/Grep 错误不级联——它们是独立查询，失败不影响其他工具。

工具延迟加载

```
核心工具 (alwaysLoad)：Agent，Read，Write，Bash...
MCP 工具 (deferred)：  通过 ToolSearchTool 按需加载
```

📖 书第八章：少量基础工具 + discover 元工具；schema 放用户侧以保 KV cache。

投机执行：在用户打字时就开始干活

不只是"预测下一步"——真的在覆盖层文件系统上执行

完整流程

```
1. 用户还在打字...
   ↓
2. Prompt Suggestion 预测下一步
   ↓
3. 启动 runForkedAgent（受限 fork）
   ↓
4. Agent 在覆盖层文件系统上执行
   - 读操作：先检查覆盖层 → 回退真实磁盘
   - 写操作：只写入覆盖目录
   ↓
5a. 用户接受 → 覆盖层"提升"到真实磁盘
5b. 用户拒绝 → 覆盖层直接删掉
```

限制与安全

约束	值
最大轮数	20
最大消息数	100
Transcript	不写入主会话
写操作	只在权限允许自动接受时

覆盖层文件系统

每个投机执行在 `<tmp>/speculation/<pid>/<id>/` 下创建独立覆盖目录。

工具白名单：写操作只允许 `Edit/Write/NotebookEdit`，只读允许 `Read/Glob/Grep/LSP` 等。白名单之外的工具直接拒绝。

CompletionBoundary：暂停条件

遇到 Bash / 遇到需确认的编辑 / 工具被安全约束拒绝 → 投机执行暂停，等用户决策。

核心不是"预测准不准"，而是"预测错了零代价"。覆盖层 + 工具白名单 + 主会话隔离 = 做对了赚时间，做错了零成本。

多 Agent 的本质：能力分区，而非角色扮演

📖 书第十章：多 Agent 协作——共享上下文 vs 不共享上下文的架构选择

工具面决定 Agent 身份

Agent 类型	工具面
主 **Agent**	完整工具集
子 **Agent**	禁止递归创建 Agent、退出计划模式等
Coordinator	只有 3 个工具：创建/发消息/停止 worker
Async Agent	只有文件读写、搜索、shell
In-process Teammate	Async + 任务管理 + cron

Agent "是谁"不是由 system prompt 决定的，而是由它能做什么决定的。 Capability-based 的身份定义比角色提示词更硬性、更可审计、更难绕过。

信任边界的显式编码

子 Agent 加载 MCP 工具时检查 `isRestrictedToPluginOnly('mcp')` ——只有 `isSourceAdminTrusted(agentDefinition.source)` 的 Agent 才能使用 MCP。

📖 书第十章：分类框架

维度	类型
上下文共享	共享（阶段化）/ 不共享
不共享时	对等 / 管理者 / 去中心化

Claude Code 的实现：

- **共享上下文**：fork agent（Dream、压缩、投机执行）共享 CacheSafeParams
- **管理者模式**：Coordinator 模式，中心调度 worker
- **去中心化**：Async Agent 通过文件系统传递状态

第四部分：纠正 — 出错时怎么办

错误恢复、熔断器、死亡螺旋防护——在确认无法恢复之前，不暴露中间态

错误恢复（上）：输出触顶与消息扣留

输出触顶的两步恢复

第一步：静默升级上限

模型输出撞 `max_output_tokens` 时，如果当前是默认 8k，**直接用 64k 重发——不加 meta 消息，对用户完全透明。只火一次。**

第二步：多轮接续（最多 3 次）

```
"Output token limit hit.
Resume directly — no apology,
no recap of what you were doing.
Pick up mid-thought if that is
where the cut happened."
```

明确告诉模型：**不要道歉、不要复述、直接接着说。**

类似策略处理其他错误

- prompt-too-long (413)
- 媒体文件过大
- 流式中断

消息扣留（Message Withholding）

```
正常流程：
  模型响应 → yield 给消费者 → 显示给用户

恢复期间：
  模型报错 → 标记 withheld → 不 yield
           → 推入 assistantMessages 供恢复检查

恢复成功：→ 消费者永远不知道出过错
全部失败：→ 释放(yield)扣留的错误
```

为什么不直接吐错误再重试？

因为 SDK 消费者（桌面端、IDE 插件、远程会话）**看到 error 字段就终止会话——恢复循环还在跑但已经没人在听了。**

> **设计哲学**：错误处理的边界不是"单次 API 请求"，而是整个恢复循环。在循环结束前，消费者不应看到任何中间状态。

错误恢复（下）：模型降级与上下文崩塌恢复

Model Fallback（模型降级）

主模型不可用时（过载、服务中断），自动切换到备用模型：

```
主模型超时/报错
    |
    ① 丢弃旧 StreamingToolExecutor 中间结果
    |
    ② 剥离旧模型特有的签名块 (signature blocks)
    |
    ③ 用备用模型重新发起请求
    |
    用户无感知，对话继续
```

关键难点：不同模型可能有不同的 tool_use 格式和签名机制，降级时需要清洗消息历史中的模型特定内容。

Context Collapse + Reactive Compact

当上下文超过模型限制（prompt-too-long 413）时的恢复链：

```
prompt-too-long 错误
    |
    ① Context Collapse (轻量)
    |   归档旧对话轮次为摘要
    |   成功? → 重试请求
    |
    ② Reactive Compact (重量)
    |   全量压缩 session memory
    |   成功? → 重试请求
    |
    ③ 都失败 → 暴露错误给用户
```

📖 书第五章：Harness 的"回退机制"

消息扣留、模型降级、自动压缩——三者构成 Harness "回退机制"在不同层面的实现：

- **消息扣留**：请求级回退
- **模型降级**：服务级回退
- **上下文压缩**：资源级回退

熔断器无处不在

Agent 最怕的不是某个操作失败，而是在失败上无限重试、烧掉整个 **token** 预算

上下文压缩

连续 **3** 次 Autocompact 失败后停止重试

内部数据：1,279 个会话出现 50+ 次连续失败，最极端的 3,272 次，每天浪费 ~250K API 调用

修复成本：一个 `MAX_CONSECUTIVE_AUTOCOMPACT_FAILURES = 3` 常量

权限分类

连续 **3** 次或累计 **20** 次拒绝后回退交互式提问

防止：

- Agent 被安全策略卡死
- 无限循环消耗 token
- Headless 模式：直接中止

输出触顶

最多 **3** 次多轮接续

先静默升级 8k→64k（1次），再注入 meta 消息续写（最多3次）

全部失败后才释放错误给消费者

📖 书第六章：评估驱动的产品改进——从 Benchmark 报告到系统改进。熔断器的阈值不是拍脑袋定的，而是基于真实产线数据。消融实验基础设施让 Anthropic 可以量化每个熔断器的价值。

Stop Hooks：每轮结束后的收尾编排

模型说完不代表这一轮结束——还有一串关键的后处理

收尾流水线（按顺序执行）

```
模型输出完毕（stop_reason = end_turn）
    |
    ① 保存 CacheSafeParams（供下轮缓存命中）
    |
    ② Job Classification（判断本轮任务类型）
    |
    ③ Prompt Suggestion（预测用户下一步）
    |
    ④ Memory Extraction（提取长期记忆）
    |
    ⑤ Auto Dream（触发条件满足时启动）
    |
    ⑥ 执行用户定义的 Stop Hooks
    |
    done
```

Tool Use Summary（异步）

在主模型推理的同时，用更快的小模型（Haiku）异步生成工具操作摘要，供后续上下文压缩使用。

死亡螺旋防护

核心问题：如果 API 调用出错 → 触发 stop hooks → stop hooks 也调 API → 也出错 → 又触发 stop hooks...

```
// query.ts 源码中的显式注释：
// "Skip stop hooks on API errors to prevent
//  death spirals where error recovery triggers
//  more errors"
```

规则：API 错误时跳过所有 stop hooks。宁可丢失一次记忆提取和 prompt suggestion，也不能让系统陷入无限循环。

Auto-Compact 熔断器

```
const MAX_CONSECUTIVE_AUTOCOMPACT_FAILURES = 3
// BQ 2026-03-10: 1,279 sessions had 50+
// consecutive failures (up to 3,272),
// wasting ~250K API calls/day globally.
```

真实教训：发现每天浪费 25 万次 API 调用后加上的。连续失败 3 次就停止尝试。

容错的深水区：消息修复与系统适配

工具调用配对修复

对话历史本质是结构化协议：每个 `tool_use` 需要配对的 `tool_result`，否则 API 拒绝。

破坏场景：会话恢复、远程同步、流式中断

```
ensureToolResultPairing():
  前向：孤立 tool_use → 注入合成错误结果
  后向：孤立 tool_result → 剥离
  检测：跨消息重复 tool_use_id → 去重
```

strict 模式：为训练数据（HFI trajectory）时，配对不匹配直接抛异常。因为合成的 tool_result 是假的——修复 API 协议没问题，但混入训练数据会"毒化"模型。

凭证轮换修复

切换 API key 后，旧 thinking/connector block 的签名失效 → `stripSignatureBlocks` 立即剥离所有签名 block。

不让电脑睡着：caffeinate 自愈

```
// 5 分钟超时，每 4 分钟重启
const CAFFEINATE_TIMEOUT_SECONDS = 300
const RESTART_INTERVAL_MS = 4 * 60 * 1000

// 为什么设超时?
// 如果 Node 进程被 SIGKILL 杀死
// (不触发 cleanup handler)
// 孤儿 caffeinate 会在超时后自动退出
// → 自愈 (self-healing) 而非永远挂着

// 引用计数：多个并发请求共享一个实例
let refCount = 0
export function startPreventSleep() {
  refCount++
  if (refCount === 1) spawnCaffeinate()
}
export function stopPreventSleep() {
  if (--refCount === 0) killCaffeinate()
}
```

生产级 **Agent** 运行在真实 **OS** 上——"系统休眠导致 API 请求超时"在 demo 里不存在，在用户那里天天发生。设计要假设自己会被粗暴杀死

第五部分：Harness 的工程化实践

消融实验、隐私工程、防泄漏——用科学方法持续改进 Harness

用科学方法做产品：消融实验与 Feature Flag

消融实验基础设施

`ABLATION_BASELINE` flag 启用后，一次性关掉 **7 个功能**：

功能	作用
Thinking mode	扩展推理
上下文压缩	Autocompact
自动记忆	Memory extraction
后台任务	Background tasks
简单模式	CLAUDE_CODE_SIMPLE
交织思考	Interleaved thinking
后台任务(2)	第二个相关 flag

注意：这段代码放在 `cli.tsx`（入口文件）——因为 BashTool、AgentTool 在模块加载时就捕获环境变量到模块级常量，init() 运行时已经来不及了。

📖 **书第六章**：消融实验——不是"感觉有用就上"，是"数据证明有用才上"。

双层 Feature Flag

层级	机制	用途
编译时	Bun `feature()` 宏	未发布功能门控
运行时	GrowthBook	灰度发布 / kill switch

编译时 flag：构建时替换为 true/false，false 分支的代码被物理删除——不是运行时不执行，是从二进制文件里消失。安全研究员反编译也找不到。

运行时 flag：所有 gate 以 `tengu_` 开头（项目代号），从磁盘缓存读取，接受脏读，不阻塞启动。

📖 书第六章：从外部评估到内部评估

方法	说明
消融实验	关掉某个功能，量化价值
AB 测试	机制 vs 目标 + 护栏
特性开关	编译时 + 运行时
提示 CI	提示变更的自动化测试
隐私感知分析	类型系统强制审计

隐私与可观测性的工程化

类型系统作为审计工具

```
// 这个类型实际上是 never
type AnalyticsMetadata_I_VERIFIED_THIS_IS_NOT_CODE_OR_FILEPATH

// 每次使用必须显式 as 转换
const queryChainIdForAnalytics =
  queryTracking.chainId as
  AnalyticsMetadata_I_VERIFIED_THIS_IS_NOT_CODE_OR_FILEPATHS
```

每一处使用都是对代码审查者的声明："我确认过这个字符串不包含敏感信息。"

MCP 工具名脱敏

```
if (toolName.startsWith('mcp__')) {
  return 'mcp_tool'   // 用户自定义服务器名
                      // 可能包含隐私
}
return toolName  // 内置工具名直接放行
```

NDJSON stdout 守卫

SDK 客户端以 NDJSON 消费流式输出。一行非 JSON 内容混入 → 客户端 parser 崩溃，无恢复路径。

```
stdout.write 守卫:
  每行输出 → JSON.parse 验证
  合法 → 转发 stdout
  非法 → 转发 stderr + [stdout-guard] 标记
```

生产级 Agent 不能假设所有依赖都是安静的。

📖 书第六章：Agent 的可观测性

- LangSmith 等追踪工具
- 每轮调用的 token 消耗追踪
- 错误率与恢复率的监控

设计启示：隐私保护不是事后加的"合规层"——编码进类型系统，让开发者在写代码时就被迫思考"这个字符串会不会泄露用户信息"。

防泄漏工程：Undercover Mode 与 Canary 检测

Undercover Mode：没有 force-OFF 的安全开关

```
export function isUndercover(): boolean {
  if (process.env.USER_TYPE === 'ant') {
    // 环境变量强制开启
    if (isEnvTruthy(CLAUDE_CODE_UNDERCOVER))
      return true
    // 自动：除非确认在内部白名单
    return getRepoClassCached() !== 'internal'
  }
  return false
}
```

关键设计：`There is NO force-OFF. This guards against model codename leaks — if we're not confident we're in an internal repo, we stay undercover.`

不对称安全

- 可以 force-ON（`CLAUDE_CODE_UNDERCOVER=1`）
- 不能 **force-OFF**
- 默认开启，确认安全才关

Canary 检测：构建产物里的字符串审计

`excluded-strings.txt` 黑名单 + CI grep 最终二进制

最有趣的案例：**Buddy** 宠物系统

```
// capybara 是 Anthropic 某模型的内部代号
// 直接写 'capybara' 会触发 canary 检测

export const capybara = c(
  0x63, 0x61, 0x70, 0x79,
  0x62, 0x61, 0x72, 0x61
) as 'capybara'
```

为了不让 capybara 特别突出，所有 **18** 个物种名都统一用 **hex** 编码。

API Key 前缀同理

```
// 不是 'sk-ant-api'
['sk', 'ant', 'api'].join('-')
```

启示：构建产物本身就是攻击面。内部代号、API key 前缀、内部 URL 都不应以明文出现在最终产物中。

彩蛋：藏在 CLI 里的宠物扭蛋机

Buddy 宠物系统

每个用户的 userId 通过 hash + 种子 PRNG 确定性地生成一个宠物：

属性	设定
物种	18 种（duck, goose, cat, dragon…）
稀有度	common(60%), uncommon(25%), rare(10%), epic(4%), legendary(1%)
属性面板	DEBUGGING, PATIENCE, CHAOS, WISDOM, SNARK
外观	6 种眼型、8 种帽子、1% 闪光概率
动画	3 帧待机动画的 ASCII art

种子 salt：`'friend-2026-401'` ——2026 年 4 月 1 日，愚人节。

还记得开头的四个证据吗？

证据一：SALT = `'friend-2026-401'` ——这里就是源码出处。friend + 4/1，精确到天。

证据二：`isBuddyTeaserWindow()` 精确检查 April 1-7, 2026。注释写着 "Sustained Twitter buzz"——这不像内部彩蛋的运营用语，更像**有预期传播量的营销策划**。

证据三：capybara 碰撞模型代号 → 倒逼 hex 编码方案。但 capybara 正好是此前被泄露的新模型名字——巧合？

证据四：所有 18 种统一 hex 编码，"so one doesn't stand out"——反而让每个逆向工程者都去解码了。

不管泄露是否有意，Anthropic 至少做到了通过一份源码，让全球开发者免费做了一次深度代码审查和口碑传播。**这可能是 2026 年最成功的技术营销案例。**

总结：从 Claude Code 源码中学到的 Agent 设计原则

一条主线、六条原则

主线：Harness 的两面

Agent = Model + Harness，而 Harness 的工程挑战在两个层面

Environment（能干事）

能力	Claude Code 实现
感知	五层上下文压缩、Prompt Cache
行动	40 个内置工具、流式并行执行
记忆	CLAUDE.md + Dream 巩固
加速	投机执行、sideQuery 并行

大家都会做——给模型提供上下文和工具。

约束 + 验证 + 纠正（靠谱地干事）

能力	Claude Code 实现
约束	Fail-closed 默认值、工具白名单
验证	LLM 权限分类、Hook 审批
纠正	熔断器、消息扣留、模型降级

真正的竞争力在这里——Claude Code 的绝大部分代码都在做这件事。

📖 书第一章：行业正从"能做事"向"可靠地做事"转变。Environment 让 Agent 强大，约束/验证/纠正让 Agent 可靠。两者缺一不可，但后者才是 Harness Engineering 的核心竞争力。

六条核心设计原则

Environment 原则一：Cache 经济学是架构约束

缓存命中率不是优化——它决定了消息怎么序列化、子 agent 怎么分叉、工具结果怎么存储。📖 书第二章

Environment 原则二：分层处理不同"保质期"的信息

工具中间输出几轮后没价值（SNIP），对话结构需压缩（COLLAPSE），全局背景需持久保留。📖 书第二章

Environment 原则三：并行化一切可以并行的 LLM 调用

sideQuery 把"调 LLM"变成到处撒的轻量操作。主模型推理期间，五六个辅助任务同时在跑。📖 书第四章

约束/纠正 原则四：熔断器要到处放

Agent 最怕的不是失败，而是在失败上无限重试。每个恢复路径都有上限——压缩 3 次、权限 3 次、输出 3 次。📖 书第六章

约束/纠正 原则五：错误不要过早暴露

在确认真的无法恢复之前，中间错误不应泄漏给消费者。静默升级、扣留错误、模型降级——"恢复循环"替代"单次重试"。📖 书第五章

约束/纠正 原则六：安全默认值必须保守

工具默认不安全、默认有写操作。Undercover 默认开启、没有 force-OFF。**不确定时，永远选更安全的选项。** 📖 书第一章

前三条是 Environment 层面——让 Agent 又快又聪明；后三条是约束/纠正层面——确保 Agent 又稳又安全。两者共同构成完整的 **Harness**。

全景回顾：理论与实践的闭环

📖 AI Agent 实战营		Claude Code 工程实践	Harness 层面
第一章 Agent 公式 = Harness	◀——▶	主循环 + 权限 + Hook	Harness 整体
第二章 Context Engineering	◀——▶	五层压缩 + CacheSafeParams	Environment（感知）
第三章 记忆与 RAG	◀——▶	CLAUDE.md + Dream + 混合检索	Environment（记忆）
第四章 工具与异步架构	◀——▶	buildTool + sideQuery	Environment（行动）
第五章 Coding Agent	◀——▶	7 工具 + 覆盖层 FS + old→new	Environment（能力）
第六章 评估	◀——▶	消融 + AB + Feature Flag	约束/纠正（验证）
第七章 后训练	◀——▶	先形后神 ≈ prompt 约束	约束/纠正（引导）
第八章 自我进化	◀——▶	Dream + Skills + 工具发现	Environment（学习）
第九章 多模态	◀——▶	sideQuery 快慢分离	Environment（感知）
第十章 多 Agent	◀——▶	工具面分区 + 信任边界	Environment + 约束/纠正

核心结论：AI Agent 实战营课程虽然是去年 8-10 月开的，但核心设计理念——`Agent = Model + Harness`、Context Engineering、ReAct 循环、外部化学习——在 Claude Code 51 万行源码中得到了完整验证。每一章的内容都能在源码中找到 Environment 或约束/纠正的工程对应物。

从 Claude Code 看 Agent 工程的未来

已经解决的问题

上下文管理：五层管线 + 缓存经济学，成本可控、延迟可预测

安全机制：LLM 分类器 + fail-closed + 熔断器 + Hook，多层防御

错误恢复：消息扣留 + 静默升级 + 配对修复，对消费者透明

离线进化：Dream 系统实现了"越用越好"的自我进化闭环

值得借鉴的方法论

- **用做研究的方法做工程**：消融实验、数据驱动决策
- **安全即默认**：fail-closed、不对称开关
- **缓存即架构**：第一天就画缓存边界图

仍然开放的问题

多 Agent 协作：Agent 之间如何协作、分工，与人类组织有什么异同？

实时交互：语音 + Computer Use + 物理世界的带宽和延迟差距

自主进化：Dream 只做记忆巩固，还没有真正的策略学习

评估标准化：如何评估 Harness 而非只评估模型？

📖 书的展望

> 模型算力有望变得像水和电一样——Infra 的边界正在从 OS 转向 LLM Context 管理，这才是下一个 UNIX 可能诞生的地方。

Claude Code 的 51 万行源码，正是这个"下一个 UNIX"的早期工程实践。

谢谢

只有 Environment 是失控的天才，只有 Harness 是安全的废物

Bojie Li

Chief Scientist, Pine AI

博客：01.me