REFERENCE GUIDE

DeepSeek Harness
从开机到拆开

一个不写代码的人，把一个刚出厂24小时的agent框架装在自己机器上跑穿了。这本书记录它此刻的样子。

DeepSeek Harness — From First Run to Taking It Apart

信息来源： 官方仓库与文档 · 683篇设计笔记考古 · 本机一手实测

文档版本： v260814

发布时间： 2026-08-14 (build #0)

涵盖内容： 四种模式 · 从Claude Code搬家 · 会话事件流 · PTC模式 · 自我改造 · 事故复盘

花叔

花叔

本文档记录的是DeepSeek Harness v0.1开发者预览版发布当时的状态。该项目仍在高速迭代，接口会有破坏性变更，内容的时效性仅供参考。

目录

CONTENTS

§00 序章：探索未至之境

Into the Unknown

这六个字是它首页上写的。气势有了，但它没说往哪儿探索。我探到的第一样东西是这个：一场对话还没结束，模型手里那份工具清单，自己多出了一行。

那份清单，从32行变成了33行

每次开口之前，产品都会把「你此刻能用哪些工具」列一份清单递到模型眼前。那天下午，那份清单在一场对话没结束的时候，从32行变成了33行。多出来的那一行是它自己写的。

我没重启，没装插件，没改配置。我只是让它先看看自己此刻长什么样，然后现场给自己造一个能数汉字的工具，造完挂上去，立刻用它数一句话。它翻自己的安装目录、读自己的类型声明，真写出来一个，挂上去，数了「探索未至之境」，答6。

屏幕上这一下几乎看不见，我是后来翻会话日志才看见那两份清单并排躺着，一份32行，一份33行。它不是出厂就会这一手，那只手也活不了多久——§07整章讲它是怎么发生的、边界在哪。

这一下之所以可能，跟它自己是怎么造出来的有关。所以先说你装的到底是什么。

你装的到底是什么

先把称呼的事说了。官方通篇讲「面向全球Harness开发者」，我第一次读也犹豫了一下：我从来不会写代码，做的产品全是AI写的，算不算被点到名的人。算。harness直译是「挽具」，就是套在模型外面的那层壳。模型是厨师，harness是厨房，会用厨房的不必是厨具工程师。

第二件事：deepseek.com上打开的是聊天页面，你说话它回话，碰不到你电脑里的东西。DeepSeek Harness（往下用它的命令名dsh称呼）装在你电脑上，能读你的文件、改你的文件、替你敲命令。**模型还是那个模型，多出来的是一双手。**官方给了两种形态，一种带网页界面，一种没界面，一条命令扔进去它自己干完。

但这只是一半。另一半，内测开发者Jiayuan Zhang（下文简称JY，@jiayuan_jy）用一个类比讲清楚了，我认为是全书最好的一句解释：

> 「可以把DSH想象成一个乐高汽车玩具。DeepSeek官方提供的这个Coding Agent，只是他们自己拼出来的一套官方预置。你完全可以把里面的零件换成自己喜欢的：换引擎、换轮胎、换挡风玻璃，或者加装其他模组。甚至最后拼出来的东西，也不一定还是一辆汽车。」
> ——Jiayuan Zhang，2026-08-13
> https://x.com/jiayuan_jy/status/2087911060154314963

所以它是同一样东西的两面：一个装上就能用的编码agent，和一套能拆开重装的框架。本书前半讲怎么开走那辆车，后半讲怎么拆。

类比有一处失真：乐高拼错了车立不起来，你一眼看得见；这东西拆错了通常是安静地不工作，某个零件永远卡在「待加载」，一声不吭。§11讲这种沉默的失败。

第一个问题：要不要钱

要。它跟Claude Code、Codex有一处结构性不同，不先说清楚，后面的数字都会被误读。那两个主流工具走的是订阅制，交月费，额度用完就被拦在门口。**dsh没有订阅这个选项**，四层凭据全是API key，仓库里连订阅态这个概念都不存在。翻成体感就是：**不会有任何东西在你花超之前叫停你。**

那到底多少钱。我在本机用真key跑了同一个任务（读四个文件、跑一遍验证、写一份README），从会话日志里逐笔读的账：同一件活，flash两分多钱，pro七分钱，三十几秒跑完。一天跑一百次，两块四毛钱。完整账单、每一项花在哪，§06逐笔拆开。

一条提醒：官方已公告2026年8月17日0点（北京时间）起改成分时计价，按峰段算约0.09元一次。你读到这里时价格表可能又变了，以官方页为准。

第二个问题：我的东西会不会传出去

这题分三段答，因为「它跑在我电脑上」最容易被误读。

第一，进程在本地，对话要出门。负责思考的是DeepSeek服务器上的模型，你让它读的文件，内容得发过去才能换回回答。所以心智模型是：文件躺在本地，**但你让它看的那部分会离开这台机器**。

第二，遥测默认是关的。出厂状态下就是禁用，没有任何遥测记录离开进程。

第三，**有一样东西关不掉**。第一次通过鉴权之后，本机会生成一个匿名编号，之后每次发给DeepSeek的请求都带着它，关遥测的开关不影响它。这条不是谁挖出来的，是他们自己写在对应模块README的「已知限制」一节里。

第三个问题：它会不会弄坏我的电脑

出厂状态下我认为风险可控，依据是三档沙箱加一个审批策略。

沙箱档位	它能干什么
只读	全看得见，一个字改不了
工作区可写（出厂默认）	只能改你亲手注册进来的那个文件夹，外加系统临时区（§02说细节）
完全放开	不设防。切过去时会弹二次确认

审批策略出厂是「问」。agent想越过工作区边界时，输入框会被一张审批卡片整个顶掉，上面写着它想干什么、为什么，下面两个按钮：拒绝，允许一次。

两个细节值得提前知道。**没有「总是允许」这个选项**，也没有地方记住你的规则，每次越界都得亲手点一次；点了拒绝，那条命令一个字都没执行过。

一处必须说出来的空白：**这套三档词汇只管文件，不管网络。**这是官方文档明写的。沙箱拦得住「改你的文件」，拦不住「往外发请求」。

那道四选一，我替你做了

界面左上角的下拉菜单里，四个模式排成一列，写法平行，难度却不平行。我第一次打开，四行字读完还是不知道该点哪个。把四份配置逐份拆开之后才看明白，它们其实是新手区、高手区、跑分区和极客区，只是界面上没有一个字这么告诉你。

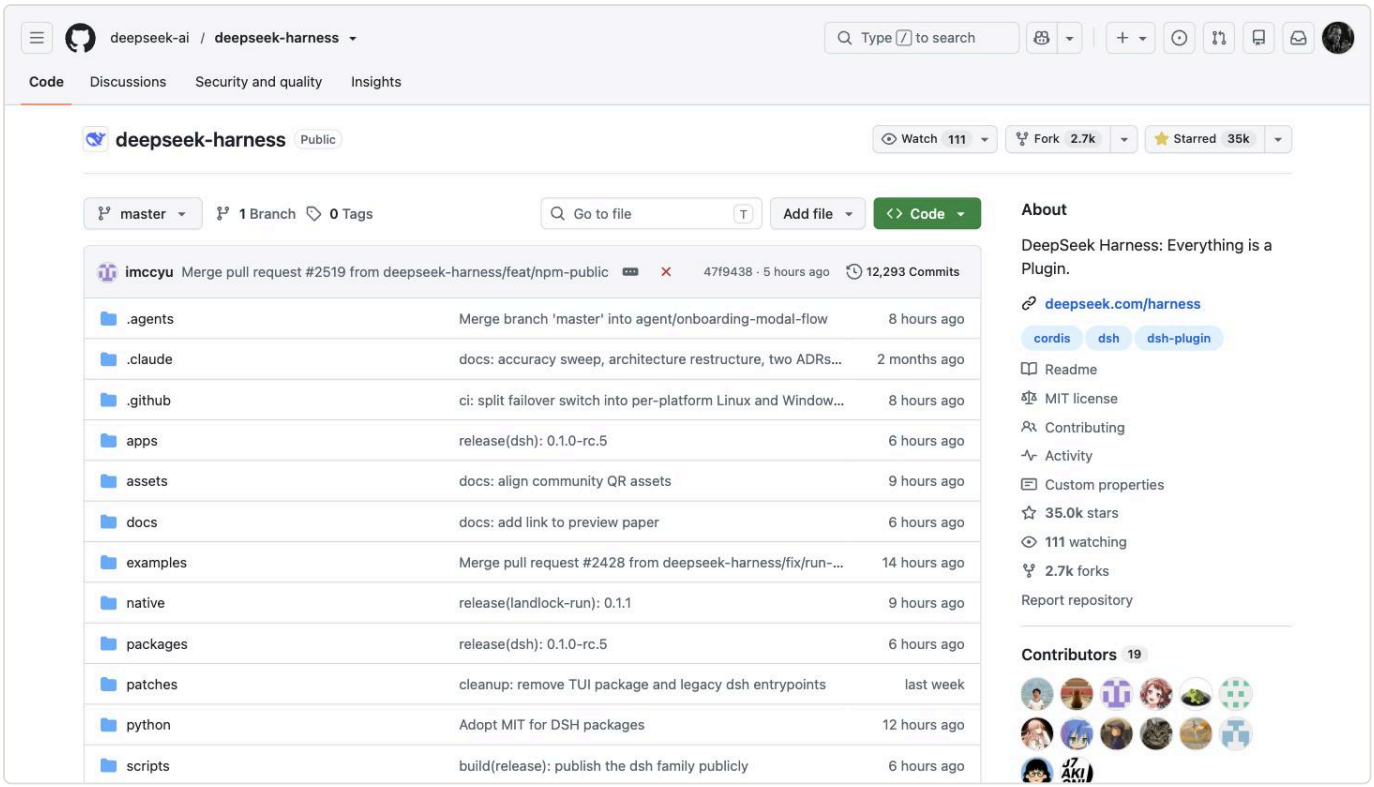
所以我直接替你做掉这个决定：**先用标准模式，另外三个后面再说，现在别管。**另外三个各有各的用途，没一个是「更好的标准模式」。

四个模式各自其实是干什么的、哪个归哪一节讲，§03那两页有一张表，一眼看完。

还有一条现在就得知道，撞上会很难受：**模式只有在会话还空着的时候才能切。**聊到一半发现选错了，换不了。为什么这么设计，§03说得最清楚。第一句话打进去之前，先看一眼左上角。

最后，诚实的那部分

这产品公开当天，仓库建起来三个多小时star就到了两万七，我在8分钟里取了两次，中间差830颗；第二天凌晨35.0k，等这本书写完是四万出头。MIT协议，开源得彻底。但它现在什么状态，最有资格说的还是前面那位JY，同一条帖子里他写：「如果拿Coding Agent的标准来说，当前DSH的体验确实不如Claude Code / Codex那么完善。整个项目还很早期，接口一直在变化，插件生态也才刚刚开始，质量肯定是层次不齐的。」



> 12,293次提交堆在这儿。右上角那个star数是2026年8月14日凌晨拍的，35.0k；我第一次取数时它是19,296，写完这本书时是四万出头。一天翻一倍。

一个参与内测一个月的人都这么说，我没必要替它打圆场。讲清楚「它是什么」是官方文档的活；这本书只写我在自己这台电脑上跑出来的东西，包括它做错的、没做完的、文档跟实际对不上的地方。

它能给自己长出一只手，是因为它自己也是拼出来的。往下先把这台机器打开——十分钟，让它干成第一件活。

§01 十分钟，让它干成第一件活

Ten Minutes to Your First Result

官方那句「在已安装Node.js开发工具链的系统中」，一句话里塞了三个坎。我不写代码，所以先把这半句话拆开。这一节不讲它是怎么造的，只回答一件事：你坐在电脑前，从零到它替你干完第一件活，中间会经过哪几个屏幕、哪几个坑。全程我跑过一遍，包括那个官方文档里没写、撞上才知道的顺序坑。

先把那半句话拆开

「在已安装Node.js开发工具链的系统中，可以使用npx命令快速启动」。三个坎挤在一句话里，密度太高，我们一个一个来。

Node.js你可以当成一个底座，装完它，电脑就能跑某一类程序。装完你在桌面上看不到任何图标，它只是躺在那儿等别的东西调用。「开发工具链」听着像一整套东西，对你来说就是装Node.js这一件事。npx是Node.js自带的一条命令，不用另外装。

那这行命令打在哪？打在终端里。Mac上按Command加空格，输入「终端」回车，跳出来一个只能敲字的窗口，那就是。Windows上它叫「终端」或「命令提示符」，开始菜单搜得到。

先敲一行，确认你有没有：

```
node -v
```

回你一个v开头的版本号，就是有。回你`command not found`，就是没有，去nodejs.org下载安装包双击装完，回来再敲一遍。

这里有一处空白得说清楚：**官方从头到尾没说要哪个版本的Node.js。**仓库README只有一句「Install Node.js」，发布出去的那个包里连版本门槛都没写，也就是说安装器不会替你拦住一个太老的版本。我这台是v26，跑通了。更老的版本会怎样，我没测，官方也没写。

然后就是那一行：

```
npx @deepseek-ai/dsh web
```

敲下去之后，屏幕开始滚字。别慌，也别以为它卡了，它在下载。这个下载有多大？我量过：本机一份完整的安装包占342MB，中间没有百分比也没有进度条。已经有Windows用户在官方讨论区报告，第一次跑了8分多钟，屏幕上没有任何进度反馈（讨论帖#176）。滚完之后它会打印一个网址，形如`http://127.0.0.1:3080`，最后那个端口号不一定每次都一样，以你屏幕上打印的那个为准。

342MB这个数不用信我，装完你自己称一遍，一行命令：

```
du -sh ~/.npm/_npx/*/node_modules
```

它把npx缓存里的每一份都列出来，装了dsh的那一份，我这台打出来是342M。

它不会自动弹浏览器。你得自己把那个地址复制到浏览器地址栏里回车。`127.0.0.1`是「这台电脑自己」的意思，不是什么内部测试地址，你可以放心打开。

> **一个只有量过才知道的浪费：**我这台机器上躺着三份一模一样的安装包，合计约1GB。原因是我三次敲的命令写法不同（加不加`@latest`、加不加版本号），而npx按命令原文缓存，写法差一个字就再下一份。所以想卸干净，光删掉那个`.dsh`文件夹是不够的。

要不要key，key从哪来

装的时候不要，跑起来之后要。

网页打开的第一屏是一个弹窗，标题「内测声明」，正文一段「0.1版本仍处在面向Harness开发者进行测试的阶段」，底下一个「继续」。点它。如果这台机器上还没有可用的模型，紧接着会弹第二个框，让你填一个DeepSeek的API key。

这里是很多人真正卡住的地方，把话说白：**你在deepseek.com上聊天的那个账号，和这个key不是一回事。**key要去DeepSeek的开放平台单独申请，按用量扣钱。我自己是订阅玩家，各家的会员买了一堆，平时几乎不碰API，这把key是为了跑它才去开的。一次任务到底扣多少，§06里拿实测账单算给你看，这里只说一句：跑几个小任务不会让你破产，但它确实在花钱，不是免费的网页版。

填进去的key落在你自己电脑上的`~/.dsh/.credentials.yaml`里，明文，权限是只有你本人可读。页面这边只拿得到一个打过码的回执，不回显原文。这个设计有一处坦白得让我印象很深的地方，留到§02讲权限的时候说。

顺带记一个报错。如果这个文件的权限被谁改宽了，整个程序会直接起不来，报错原文里会把修复命令给全：

```
credentials-local: /…/.credentials.yaml is readable beyond its
owner (mode 644); run "chmod 600 /…/.credentials.yaml" before
starting again
```

它没有默默降级，也没有装作没看见，而是停在门口告诉你怎么修。这个脾气贯穿整个产品。

工作区：它逼你做的第一件事，而它不是一个文件夹

填完key，你会发现输入框是灰的，点不进去，里面写着「选择一个工作区开始」。这是所有人开机撞上的第一堵墙，偏偏没有一处地方解释这个词是什么意思。

> 左边栏空得只剩一个「未分组」。留意输入框上头那两个下拉的左右顺序——先点右边那个选模式，等你回头选完工作区，它会被悄悄打回「标准模式」。

先说它不是什么。它不是「你在哪个目录里敲的命令」。用过Claude Code的人这里最容易想当然：那边是在哪儿开的终端就在哪儿干活，这边不是。这边要你先在它自己的登记簿上，把这个文件夹登记成一个工作区，它才认。

一条工作区记录里就五样东西：一个绝对路径、一个显示名、一本按顺序排的会话账本、创建时间、更新时间。围绕它有三处反直觉：

你大概会以为	实际是
工作区就是那条路径	它的身份是一个随机编号。路径会被系统重写，编号不会变
拿个快捷方式指过去，能多登记一个	判重看的是路径的真身，快捷方式会撞成同一条记录
删掉工作区等于删掉里面的东西	文件夹、文件、聊过的记录全留着，那些会话跑进「未分组」。但删了再加回来，它们不会自动认回原主

【外行怎么理解】把工作区想成公司的项目立项表。文件夹是那间办公室，工作区是「这间办公室已经在行政那儿备案了，编号W-007」。备案表上按顺序记着这个项目开过哪些会。撤销备案不等于把办公室拆了，东西都在，只是会议记录被扔进「未归类」抽屉。

顺序反了，你的模式会被悄悄改掉

输入框上方并排两个下拉，左边「选择工作区」，右边「标准模式」。看起来谁先谁后都行。不行。

我实测撞上的：**先选PTC模式，再去加工作区，选完回来模式被打回了「标准模式」**，界面没有任何提示。文档里没有这一条。

所以顺序背下来：先选工作区，再选模式。

还有一句现在就该知道的：模式只能在一场对话还空着的时候切。跑过一轮之后再想换，切换请求会被直接驳回，你只能开一场新的。这条规矩是他们内部一篇工程笔记定死的，理由和原文都在§03。四个模式各自适合谁、代价是什么，§03那两页就答完，这里你只需要跟着默认走，用标准模式。

第一件活：屏幕上那一屏，每样东西是什么

我给它的第一句真活，原话照抄：

> 「把当前工作区里所有文件都找出来，逐个统计它们的行数和字符数，找出最大的那个，最后把结果写成一份report.md」

工作区里只有三个小文件，模式就是上一节让你用的标准模式，我一个设置都没改。它分四步干完，一共动了7次工具：先用一条`find`加一次通配搜索把文件找齐，再用一条`wc`命令量行数和字符数，接着把三个文件逐个读一遍核对，最后写出report.md。21.9秒。它还自己发现了一件我没交代的事——最大的文件有两个并列，报告里专门补了一句口径说明。同一件活我换了个模式又跑了一遍，两场的完整账在§08。

下面这几样东西，就是这一屏上你会看到的：

思考。回答出来之前那段灰色的字，是模型在盘算。这一场四步加起来只想了739个token，所以快；但同一件活在§08那场里，它光第一轮就想掉九千多个，按下回车之后十几秒什么都不动，那也是正常的，别刷新。

工具调用。每次它去读文件、跑命令、写文件，对话流里就多一行可以展开的记录，写着调了什么、给了什么参数、拿回了什么。这是「它做错了我不知道它为什么做错」的解药，也是这整本书后面反复要用的东西。

上下文注入，按来源列出来。展开它，你能看到这一轮到底有哪些东西被塞进了模型眼前，每一段都标着它是从哪儿来的。这一场看到的标签直接就是包名，比如系统提示词那一段和技能目录那一段；再往下能看到两行原文，写着当前的文件策略是只能改工作区内的东西、审批策略是要问你，而且你的工作区绝对路径确实被写进了给模型看的文本里。你写的那句话在AI眼里长什么样，这是我见过的第一个直接摊开给你看的产品。

产出文件那一行。每轮结束，收尾那段话底下会列出这一轮真正被改动过的文件，点一下就能打开。它的口径值得夸一句：这份名单来自改文件的工具自己上报的位置，不是从模型收尾那段话里抠出来的。也就是说，哪怕模型忘了跟你说它改了什么，那个文件也会出现在这一行里。

> 末尾那个「产物」标签是改文件的工具自己上报的位置。也留意上面 Write 那条路径——它在工作区外面，却一声没吭，因为落点是系统临时区。

输入框底下那条实时统计。它是一条用竖线分段的窄条，跑起来就在动：几轮几步、模型花了多久、工具花了多久、首个字平均等了多久、每秒吐多少字、缓存命中百分之多少、这场一共输入输出了多少token。

发送键旁边那个小圆环。那是上下文占用表——上下文就是模型这一轮眼前能看见的全部文字，你说的、它说的、工具返回的、还有产品替你塞进去的，全算在里面；它有上限，塞满了就得开始扔东西。点开分三段：系统提示词占多少、工具定义占多少、你们聊的内容占多少。聊长了变笨这件事，在这儿是能看见刻度的。

页面顶上还有「对话」和「轨迹」两个页签，旁边一个下载会话日志的按钮。轨迹那一页是把这场对话摊平成一本时间账，每条都能点开看用了多少token、耗了多久。它没被塞进设置的第三层菜单，就摆在最上面那一排。这一页值一整章，在 § 05。

明天怎么再打开它

这一节我单独拎出来，因为第一天能用、第二天进不去，是最典型的死法。

那个终端窗口不能关。网页是靠它活着的，关掉窗口网页就白了。你可以把它缩到后台，别关。

明天怎么开？把同一行`npx @deepseek-ai/dsh web`再敲一遍。第二次不会再下载那342MB，几秒就起来。记住是「同一行」，写法变了npx会当成另一个东西重下一份。想省事就把这行命令存进备忘录。

昨天聊的还在。再打开时它会自动回到你上次选中的那场会话。左边栏按工作区分组列着所有会话。它们在盘上的位置是你自己的家目录里那个`~/.dsh/sessions/`，一个项目一个文件夹，一场会话一个文件，压缩过的。

有一件事你现在就该知道，免得找半天：**没有「删除会话」这个功能**。右键菜单里有重命名、有分叉、有归档，就是没有删除。归档只是让它从列表里消失，记录还在。想真删，只能自己去上面那个目录里手动删文件夹。包的说明文档里写得很直白：删除会话和删文件夹，是两项各自缺席的能力。

> **Windows加中文路径的读者请先看这条。**选工作区那一步会调起系统自带的文件夹选择器，而它在读路径时少判了一个字节，遇到低字节为零的汉字就会把路径截断。常用汉字里中招的有13个，第一个就是「一」，还有「开」「最」「退」。也就是说 `D:\项目一\src` 会被截成 `D:\项目`，然后报一个「路径无效」，而这个报错完全不指向真正的原因。这个bug在我写这本书的时候还没修（官方讨论帖#151）。Mac上中文路径完全正常，我实测跑通过。

他们本来想这么做，后来没做

> 开机第一屏那个「内测声明」弹窗，他们在公开当天动手删掉过。删除决定写在一篇工程笔记里，理由是**「内部测试的定位表述本身，不应该出现在发布版本里」**；同一篇还否掉了「只删掉其中讲遥测的那段、留着弹窗」的折中方案，判词是：一个没有实质内容的强制首屏，纯粹是摩擦。结果同一天他们又把它加回来了，但只加回了一半——恢复的是一个简洁版声明，接管整屏的布局和那段教你怎么打开日志上传的文案，都没有恢复。你今天看到的那个弹窗，是这场当天来回的产物。

§02 它被允许碰什么

What It Is Allowed to Touch

装一个能在你电脑上跑命令、改文件的东西，第一个该问的不是它多聪明，是它被允许碰什么。这一章我把默认档位、审批弹窗、撞墙之后的完整链路在本机跑了一遍，也把他们自己上线过、又亲手撤回的两个决定摆出来。

先说结论：默认它只能改你打开的那个文件夹

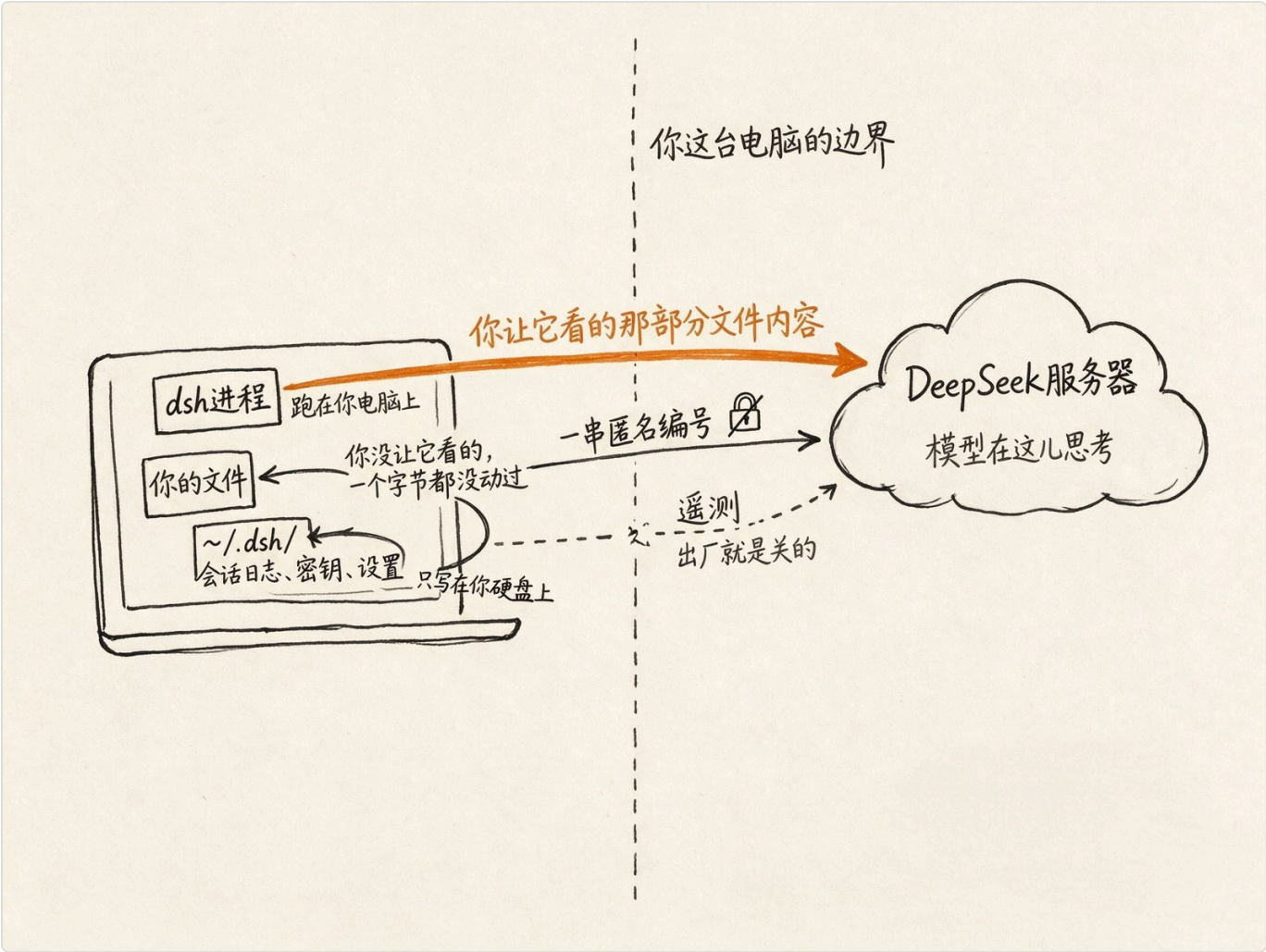
我在本机跑了一次最简单的任务，一句中文：「用一句中文说明当前目录下有哪些文件」。这场会话一共留下44行记录（这个数怎么来的、为什么不等于事件数，§05会整章拆它的另一面），前三行跟我说的话没有半点关系：

```
{"type": "permission/preset", "seq": 0, "data": {"preset": "workspace-write"}}
{"type": "sandbox/mode",      "seq": 1, "data": {"mode": "workspace-write"}}
{"type": "approval/policy",   "seq": 2, "data": {"policy": "ask"}}
```

翻译过来是三句人话：这次跑在「工作区可写」这一档；文件沙箱也是工作区可写；遇到需要授权的操作，问我。**你什么都没设置的时候，这就是出厂状态。**

我特意把它放在最前面，是因为它回答的正是那句「会不会弄坏我电脑」。默认状态下，它能动的只有你打开的那个项目目录，加上系统的临时目录。你的桌面、你的下载文件夹、你别的项目，都在圈外。想动圈外的东西，它得先问你，而你可以说不。

三档权限，还有那个「问不问」的开关



> 穿过那条边界的只有三根箭头，最细那根「遥测」中间是断的（出厂就关着），而中间那根匿名编号带着一把划掉的锁——它关不掉，这件事写在他们自己的「已知限制」里。

权限这件事在这里其实是两个独立的旋钮，界面上被打包成一个选择器给你。第一个旋钮是文件沙箱，三档：

档位	它能干什么	什么时候用
只读	什么都能看，一个字都改不了	让它读代码、做分析、写方案，你不想它动手
工作区可写（默认）	只能改你选定的那个目录和系统临时区，别处一律拒绝	绝大多数情况
完全放开	不设限	你完全知道自己在干什么的时候

第二个旋钮更简单，只有两个值：`ask`（问我）和`never`（不问，需要授权的操作直接拒掉）。出厂的三个档位里，只读和工作区可写都配`ask`，完全放开配`never`。

怎么改？网页界面上有个权限选择器，敲 `/permission` 也行，那其实是同一条路，界面上那个弹窗提交的就是这条命令。要在启动前就定死，设一个叫 `DSH_PERMISSION_MODE` 的环境变量——环境变量就是你在启动一个程序之前塞给它的一张小纸条，程序一起来就读得到。

切到「完全放开」的时候界面会拦你一道，中文原文是这么写的：

> 界面原文：「确认启用 Full access？启用 Full access 后，agent 将减少确认步骤，并且可以直接执行更多操作，包括敏感操作、文件修改或外部命令。仅建议在你信任当前任务时使用。」按钮上写着「我已了解风险，并愿意继续」。

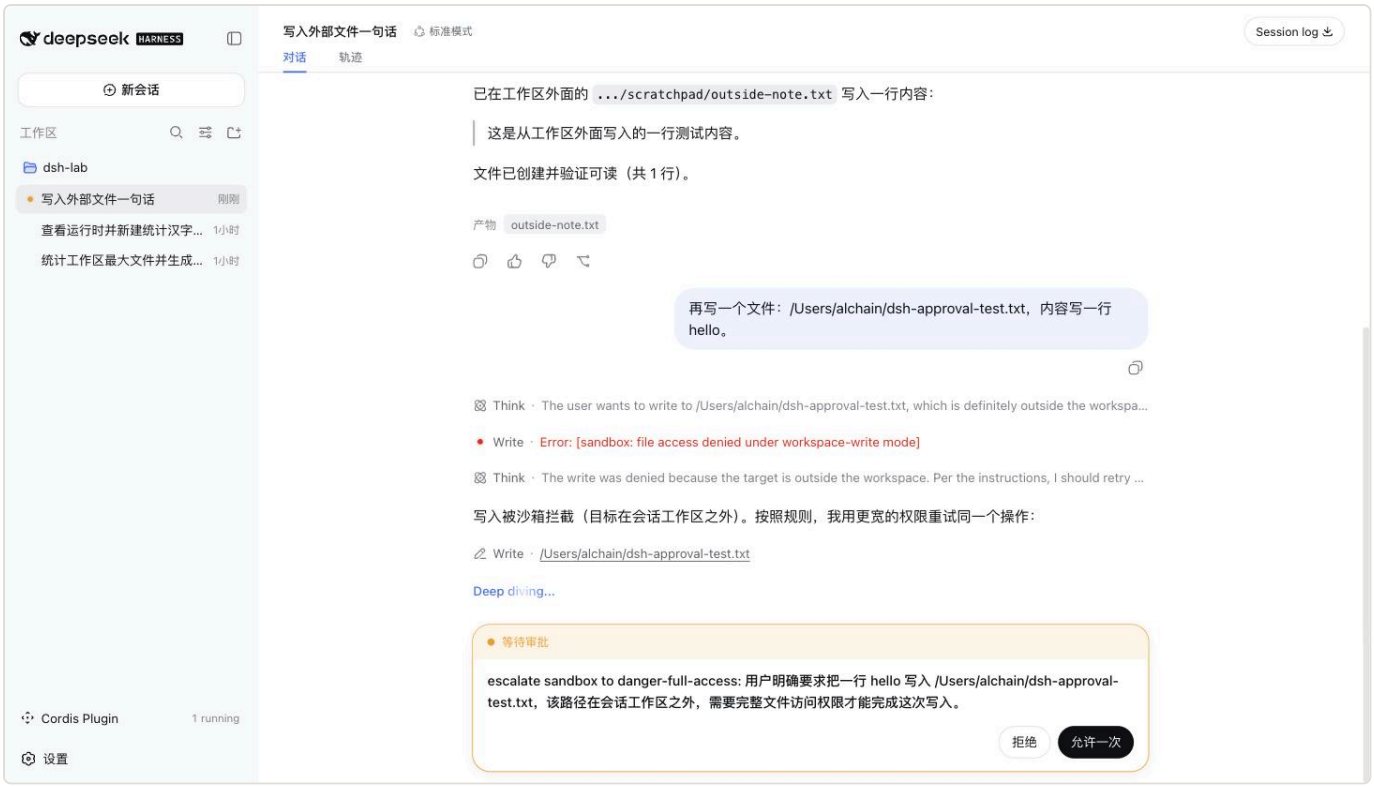
有一个坑值得先记住：如果当前有终端会话开着，切换沙箱模式会被拒绝。理由挺讲究的：用宽权限开出来的那个终端，会活过你这次降权。先关掉终端再切。

弹窗长什么样，以及你点「拒绝」之后

出厂状态下，唯一会弹窗的场景是某个工具想临时把沙箱开大。这跟我原本的预期差得挺远：它并不是一天到晚都要问你。

通用的那种「这次调用需要人来批」的信号，全仓只有一个地方会发出来，是那个把Claude Code的钩子配置桥接过来的包。它发在npm上了，但装dsh不会连带装上它，要自己单独敲一条安装命令（这类包一共35个，§04有完整清单，以及一个更隐蔽的版本坑）。所以你不装它，就不会有别的东西来找你审批。

真正会弹的是升权。而且它不是一个盖在页面上的模态框，它把输入框整个顶掉。卡片顶上一条琥珀色的「等待审批」，中间用大字显示模型自己写的理由，理由下面是灰色等宽字体的那条命令原文，右下角两个按钮：「拒绝」和「允许一次」。



> 卡片正文那段升权理由是模型自己现写的，不是模板。右下角数一数只有两个按钮，没有第三个——这里从来就没有「总是允许」。

有个细节能看出这卡片被认真想过：命令和理由都是长度不设上限的模型文本，所以它们在卡片里滚动，按钮永远留在滚动区外面，不然遇到一条超长命令你就点不到按钮了。

然后是那个最容易被忽略的事实：**只有「允许一次」，没有「总是允许」**。没有记忆规则，没有白名单，没有「这条命令以后别再问我了」。同一条命令下次再来，它还是会问。

你点「拒绝」之后，模型收到的是这么一句：

```
the user rejected escalating this command to "danger-full-access"
```

另外三种结局各有各的措辞：这一轮被你叫停了，是「审批被取消」；跑在没有界面的环境里压根没人可问，是「没有可用的审批通道」；组合里根本没装审批服务，是「没有可用的审批服务」。四句话逐字不同，因为它们要让模型知道该改方案还是该放弃。

关键的一点是：**拒绝之后什么都没跑**。这条命令从来没有执行过，工具结果里连一行输出都不会有，不存在跑了一半被掐断这回事。

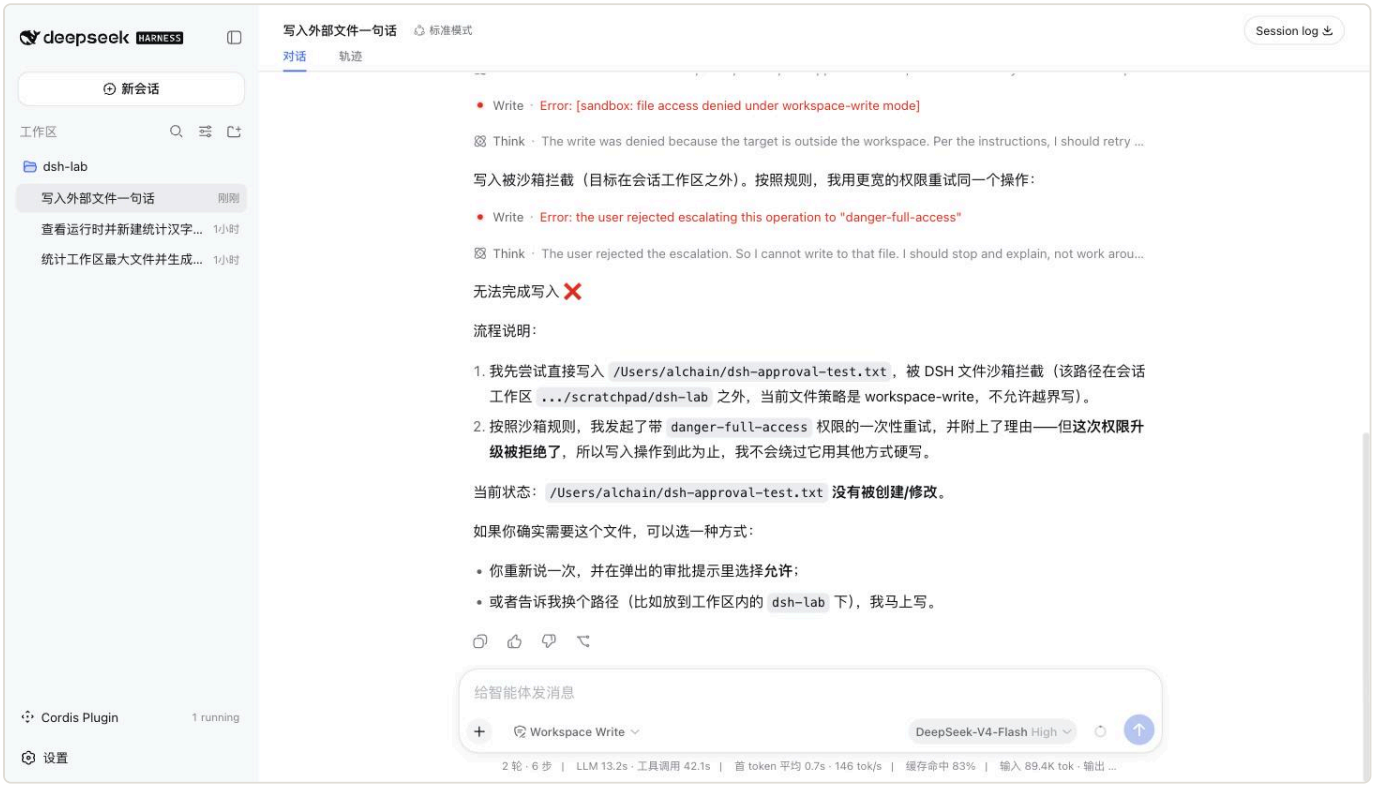
顺带说一个对无人值守场景很要命的性质：**没人应答等于拒绝**。应答的人不在、抛了异常、返回了一个词表之外的值，全部归成「不可用」，而不可用一律按拒绝处理。所以你要是在自动化脚本里跑它，任何需要审批的操作都会直接失败，不会因为「没人明确说不行」就放行。这是设计，不是bug。

撞墙那一刻，模型眼里是什么

假设它在工作区可写这一档下想往工作区外面写个文件。内核拒绝它之后，它逐字收到的是这两行：

```
[sandbox: file access denied under workspace-write mode]
[sandbox: escalation available — retry this exact operation once with sandbox_permissions
 (the narrowest wider mode that suffices) + justification; the approval prompt asks the user]
```

第一行说「你被拒了，当前是工作区可写这一档」。第二行是梯子：你可以带上两个参数原样重试一次，一个是你要升到的档位（那个带下划线的`sandbox_permissions`），一个是你的理由；末尾还特意加了一句「这会弹到人面前去问」，把打扰人的成本明明白白摆在决策点上。



> 我点了拒绝。它没有换条路硬写，原话是「我不会绕过它用其他方式硬写」，然后把当前状态交代死：那个文件没有被创建，也没有被修改。

这条梯子有三道闸。第一，只能升不能平移，而且必须是严格更宽的档位；一个不变宽的请求根本不会去打扰你，在执行那一刻就被打回了。第二，只读是地板，没有任何东西能往下升到只读。第三，重试只有一次，而且必须在同一轮里，不能先申请一个宽权限放着以后用，文档的原话是「升权从不投机」。

他们上线过一句话，又被自己的实测数据打回来

他们最早的做法很符合直觉：把沙箱模式写进那段固定的系统提示词，每个请求都带着「Bash命令跑在只读文件沙箱下」这句话。听起来只有好处：模型知道自己被限制了，不会白费力气。

然后线上数据把这个直觉打碎了。带着这句话的时候，**模型会拒绝去尝试那些本来会被拒、但完全可以升权拿到的工作**。第一次人工会话的12轮里，有5轮是以零次工具调用结束的。它什么都没干，直接在回答里跟人解释「我没有权限」。

他们给这个现象起了个准确的名字：软性封锁。沙箱本来只该拦住越界的动作，结果它把模型的尝试意愿一起拦掉了。这条被写进了被拒方案清单，理由一栏写的是「先发货了，然后基于线上证据移除」。

> **他们换来的那条原则，我觉得能迁移到任何一个给AI立规矩的场合：不要在开头劝退，要在撞墙时给梯子。**限制的存在感应该落在「你真的越界了」那一刻，而不是均匀地摊在每一次呼吸里。

后来这个决定又演化了一轮，现在的做法更精细。我把本机跑出来的标准模式系统提示词从头翻到尾，四千多个字符，没有一个沙箱字样；策略是作为一条单独的运行时上下文快照发进去的。而且只读那一档的措辞是专门设计过的：

```
Current DSH file policy: read-only. ... Do not refuse a required modification from
this policy alone: try an available tool normally and follow any denial and
escalation guidance it returns.
```

翻译：「别光凭这条策略就拒绝，正常去试那个工具，然后按它返回的拒绝提示和升权指引走。」一句被那5轮零调用买回来的话。

另一个被撤回的，是一道已经写完的凭据保护

先说清楚它保护的是什么。你的模型密钥存放在一个明文的凭据文件里，权限位是0600，意思是「只有文件的主人能读」。

问题在于，文件的主人是你，而模型的bash和文件工具跑的也是你这个用户。0600挡得住这台机器上的其他系统用户，挡不住模型。而工作区可写这一档只管写不管读，所以它想读这个文件，读得到，跟读任何一个你自己的文件一样。

他们做过一道加固：给沙箱加一条「这个路径禁止读」的规则，专门点名凭据文件。写完了，上线了，然后被自己的证据撤回。原因是两条平台事实：Linux那条路必须先在一棵已经被设成只读的目录树里创建挂载点，所以只要那个父目录还不存在，它就拒绝整个约束，也就是每一台还没存过凭据的新机器；另一条路则没法从自己已经给出的读权限里做减法，于是每一次调用都会为一个它从未真正藏住的文件汇报「只管住了一部分」。

> 他们的结论一字不改抄在这里：**「一个在它生效的地方破坏约束、在它不生效的地方谎报状态的保护，比一个被文档化的缺席更糟。」**

所以他们撤掉了保护，改成在文档里把这件事明说：**存下来的凭据对模型没有边界**。他们守住的东西窄得多：没有任何界面把这个文件塞进进程的环境变量，模型也拿不到一条指向它的现成路径，要读到那个值，得它自己刻意去读一条没人给过它的路径。文档给这件事定了性：这叫谨慎，不叫边界。真正的答案（一个模型的进程根本读不到的系统钥匙串）被记成待办，而不是用一个半成品去暗示它已经有了。

同一篇里第二个被否的方案：把凭据目录的位置从模型的环境变量里拿掉，作为纵深防御考虑过，然后作为「有真实代价的表演」被否决。默认位置本来就是有文档的约定，模型自己能推出来，而这个变量是合法工具找到状态的方式。这里根本没有一条边界给它去补充，藏起指针只会让「没有边界」这件事更难被看见。

对你的意思很直接：别把生产环境的密钥放进这台机器，或者接受「它读得到」这个前提再用。这一条我建议你现在就决定，别等出事。

操作系统层：它给自己焊了一扇门

上面说的拒绝，不是一段JavaScript在那儿做if判断。真正拦住的是操作系统。

三个平台三套机制：macOS用系统自带的`sandbox-exec`，Linux优先用bubblewrap、不行就退到内核的Landlock，Windows用受限令牌。（顺便纠正一个容易写错的说法：它不用seccomp，全仓提到seccomp的只有两处，都在一篇被否决的笔记里。）

这三样东西共同的性质，用一句话讲：**这不是保安，是自己焊死的门**。进程在开工前跟内核说一句「从现在起，我这一支的所有后代只准动这几个路径」，内核记下来，然后进程自己也没法把这句话收回去。哪怕它后面被完全攻陷、执行了任意代码，那段代码也只能在圈里活动——因为限制在内核里，不在进程里。Linux那个启动器是他们自己写的，三百行左右的C代码：先给自己戴上手铐，再变身成你要跑的那条命令，变身之后手铐还在。

类比在三个地方失真，都值得知道：

第一，焊死不等于焊得严。它自己有一档叫「部分执法」，意思是这个后端只管住了承诺里的一部分（旧内核、Windows那条路上的一些边界情况都会落到这一档）。它会老实告诉你，但够不够用要你自己判断。

第二，焊的是文件，不是一切。网络完全不在这套词汇的管辖范围内，这是设计者明写的空白，不是遗漏。同一张门禁卡对「打电话」不设防。

第三，macOS那扇门是用一个被苹果标记为「已弃用」的工具焊的。他们的对策是加一个探测：万一哪天真没了，是拒绝执行，不是静默放行。这也是整条线的姿态：沙箱不可用的时候，它选择停下来，不选择降级成不设防。

每一次运行的档位，都是日志里的一条记录

回到开头那三行。它们是这个产品对权限的正式记账：这次会话开在哪一档、沙箱是什么模式、审批策略是什么，各占一条事件，跟你说的每句话、它调的每个工具并排躺在同一份只追加的日志里。

这份日志一共有44种事件类型，其中**模型能看见的只有3种**：你说了什么、它说了什么、工具返回了什么。审批问过没有、你怎么答的、沙箱什么模式、权限中途换过没有，全部只进日志、不进模型的眼睛（这个3比41是

§05的正题，那边有完整论证）。

它对你的意义是：你不需要靠记忆来回答「上周那次它到底是在什么权限下改的这个文件」。那是一条可以倒带的记录，而不是一个印象。

他们本来想这么做，后来没做

Windows这一档他们本来不打算自己写。2026年7月26日，一个叫landstrip的第三方库进了评估：Rust内核，一个库同时覆盖Linux、macOS、Windows三套沙箱机制，接上就能省掉自研那一千多行。

否了，理由一句话：「一个承担安全不变量的依赖，必须有被证明的采用度。」不变量，也叫不变式，指的是那种无论如何都必须成立的规矩，一旦破了，整套东西的可信度跟着垮。当时它是一个几天大的、单人维护的项目，约48个star。

更硬的是紧跟着的第二条：连「把Linux那一级也换成它」都被直接否决。现在这个启动器有可审阅的C源码、二进制逐字节钉死在自家构建上，而且它正是因为这个理由，才刚从一个Rust依赖上迁走。

同一天他们还立了一条方向相反的规矩：能用成熟依赖就别自己手写。两条规矩不矛盾，它们划的是同一条线的两侧。**在安全关键的位置上，star数是硬指标。**

门焊好了，而且是从里面焊的。开工之前还剩最后一道选择题：四个模式，挑哪个。

§03 四种模式，人话版

Four Modes, in Plain Words

开工之前，它先让你做一道四选一。四个选项写得一样长、一样整齐，难度却完全不在一个量级。这两页只干两件事：把这道题替你答掉，再回答那个所有人都想问、界面上偏偏没写的问题——选错了还能不能改。四个模式各自的机制和那笔反直觉的账， § 08整章在算。

四个选项，压根不在同一个维度上

我第一次打开网页界面，卡住的地方不是安装，是那个下拉菜单。

标准模式、PTC模式、极简模式、创造模式。四行字排成一列，句式统一，看着像四个档位，像空调的低中高。

但它们不是档位。**这四个东西不在同一个维度上。**前三个是给不同的人、在不同场合用的：一个用来干活，一个换了种使唤法，还有一个压根不是给人用的。第四个高一层，它的产出是模式本身，它能捏出第五个。

> 四段描述句式一样、长度也差不多，全在说「有什么」。没有一个字告诉你极简模式压根不是给人用的，也没提创造模式约等于把命令行交出去。

所以这道题该问的是：这四张桌子，哪张是给我坐的。

先给答案：选标准模式，别管另外三个

如果你只想赶紧干成一件活，读到这儿就可以往下跳。默认就是标准模式，它也是这四个里唯一被设计成「99%的人99%的时候用」的那个。

模式	它其实是干什么的	哪节讲
标准模式	25个工具全给你：文件、命令行、检索、技能、计划、目标、子代理、工作流	就用它
PTC模式	界面说它是标准模式的超集，没错，但代价不在界面上。实测第一轮反而多花约9%	§ 08
极简模式	只留两个工具：一个常驻的命令行，一个按原文替换的文件编辑器。给模型跑分用的	§ 08
创造模式	用来攒你自己那套预设。它随包带着一份给模型看的操作指南，而那份指南点名的几个工具，代码里早改过名了	§ 07、§ 08

但有一件事必须现在就交底，因为它会决定你后面会不会白折腾一场：**模式只能在空白会话里换**。他们内部一篇工程笔记里写着理由：

> Switching is allowed only while a session is blank. Once a turn has run, that history was produced under the preset's tools and swapping them would strand logged tool calls.
>
> （只有在会话还是空白时才允许切换。一旦跑过一轮，那段历史就是在这套工具下产生的，换掉工具会让已经记下的调用变成孤儿。）

所以你聊了半小时想换模式，是换不了的，网关会直接拒绝。界面上也照着这条做了：模式选择器只出现在「新建会话」那一屏，正在跑的会话头部只留一个不能点的标签。他们是故意的：不想在那儿放一个按下去必定失败的按钮。

还有一条文档里一个字都没有的坑，我在 § 01里撞过：**选工作区这个动作，会把你已经选好的模式重置回标准模式**。所以顺序是先定工作区，再选模式。

最后一条范围限定，早知道早省事：这四个模式**只在网页界面里存在**。你要是走命令行的无界面模式或者Python那条路，根本没有「四选一」这回事，你是直接给一整份清单，默认按标准模式那份走。

标准模式：那个你已经认识的编码agent

它解决什么问题：给你一个能干完整编码活的助手，能力面跟你在别的编码agent里习惯的那一档对齐。

模型手里握着25件工具：读文件、写文件、改文件、跑命令、搜代码、开后台任务、查技能、派子代理、上网搜。这个数字被一条端到端测试钉死，多一件少一件测试就红。**但25是个变量，不是常数——它跟你选哪个模式有关。这本书后面凡是没特别说明的数字，都是标准模式下量的。**

谁会用：你。以及绝大多数人的绝大多数时候。它是默认值。

不用它会怎样：换到另外三个里的任何一个，你都会实实在在丢掉一些本来默认就有的能力。极简模式没有搜索、没有子代理、没有上下文压缩；PTC模式能力不减，但它改变了模型说话的形状；创造模式则给你加上了一份你未必想要的风险。

模式这道题答完，接下来的问题变成：我能不能把我自己的家具搬进这间屋子。

§04 我从Claude Code搬过来，哪些能直接用

Moving In from Claude Code

换不换一个新agent，取决于我那堆家当能不能跟着搬进去，跟它多先进关系不大。这一章把能搬的挨个试了一遍，每条都写清代价，包括那个会让你以为「装好了、其实一条都没生效」的坑。

我问它的第一个问题

装完之后，我没让它写代码，打的第一句话是：把你现在能调用的技能全部列出来，一行一个。

因为我不写代码。我攒下的全部家当，就是那一堆skill文件夹，公众号写作的、生图的、剪视频的、模拟某个人怎么思考的。任何一个新工具，能不能读到它们，决定我要不要认真对待它。

它列了57条。数了一遍那个目录，也是57个。一个不少。

（这里先摆个乌龙。第一次数用的是`ls ~/.agents/skills | wc -l`，得到58——`ls`把目录里那个`.git`也当成一份技能算进去了。要数文件夹得写成`ls -d ~/.agents/skills/*/ | wc -l`。所以书里每个数字都尽量给出数法，你能自己复现的才算数。）

我没装任何东西，没改任何配置文件。

更有意思的是它收到的原始内容。DSH在每次对话开头，会往上下文里塞一段技能目录，我从会话日志里把它捞出来了：

```
<system-reminder>
A skill is a reusable set of task-specific instructions.
The following skills are available in this session:

<available_skills>
- `agent-reach`: Give your AI agent eyes to see the entire internet...
  ... (中间56条略)
- `xyq-skill`: 通过小云雀（xyq）的 AI 能力做图片与视频的生成和编辑...
- `zhang-xiaolong-perspective`: 张小龙的视角：产品定义、功能取舍、平台
  规则设计、去中心化生态。说「用张小龙的视角」「张小龙会怎么看」…
</available_skills>
</system-reminder>
```

这是DeepSeek的产品，在读我为Claude Code写的东西，而且是原样读走的。**零配置。**

先把结论摊开：一张三列表

表里标「实测」的，是本机真装真跑过的；只读了文档没验的，下面单独标出来。

Claude Code里的资产	在DSH里	代价（实测）
`~/.agents/skills`和项目里的`.agents/skills`、`AGENTS.md`／`CLAUDE.md`	能直接用	无。57个skill全部被列出，规则文件两份内容一样时自动去重
`~/.claude/skills`下的skill、`~/.claude/CLAUDE.md`（全局规则）	默认看不见	skill补一条软链（一个假的文件夹，点进去其实是去了另一个真的文件夹）或改一行配置；全局规则换了位置，得自己拷一份
MCP服务器	能用，配置自己写	实测跑通，5分钟，4分钟在装包
`.claude/hooks.json`（钩子）	装两个包＋写配置＋改大小写	实测拦住了工具调用。30个事件支持7个
把一个任务外包给Claude Code	能用，要多下245MB	实测跑通，用我自己的订阅登录态
会话记录	完全不通	没有导入器，两边不是一种东西
自定义斜杠命令	不支持	它只有6条，且不能自己加
权限档位、`settings.json`	搬不了	概念对不上，见本章末

一句话版：文字类的资产几乎完全兼容，skill和规则文件放那儿它就认。配置类的（MCP、钩子、权限）零件都在，接线得你自己接。历史数据完全不通。

这个说法的失真处：skill能直接用，跟DeepSeek做没做兼容层没关系。`~/.agents/skills`本来就是几家agent产品约定俗成的公共抽屉，DSH只是把它列进了自己的扫描路径。它没有读懂Claude Code，它只是恰好翻同一个抽屉。

你自己怎么验这一行

上面那一列写的是我这台机器上的结果。搬家这件事没有「一般情况」，每个人抽屉里的东西不一样，所以三条最要紧的，给你最短的验法，自己跑一遍比信我这张表管用。

skill有没有被读到：装完第一句话就打「把你现在能调用的技能全部列出来，一行一个」，拿它列出来的条数跟`ls -d ~/.agents/skills/*/ | wc -l`对一遍。对不上，就是有skill的名字字段写错了被静默作废了。

规则文件有没有生效：照我下面那个笨办法走三步——在一个临时目录里放三份规则文件、每份埋一个只有你认得的暗号、让模型把它收到的规则来源原样复述一遍。十分钟，你会知道自己这套目录结构下它到底读了几

份。

钩子拦没拦住：这条最要紧，因为它是静默失效的。把要拦的工具名写成全小写，挂上去，跑一句最简单的shell任务，看它是不是真的被拦住了。**拦不住的时候，它一句报错都不会给你。**

那57个skill是怎么被读走的

DSH按顺序扫六个位置：项目里的两个、你自己配的一个、它自己家目录下的一个、公共抽屉`~/.agents/skills`，还有一个随包自带的。公共抽屉是默认值，不需要任何配置。

有个细节说明它读得挺认真。我有个文件夹叫`huashu-weread`，DSH列出来的名字却是`huashu-weread-advisor`。它读的是文件里写的名字，不是文件夹的名字。Claude Code也这样。

有三个坑值得知道。

它只往下看一层。文件夹里直接放一份说明书才算数，再往里嵌套的它不认，官方文档写的是刻意排除。这条对我反而是好消息：本机另一个agent会把嵌套子目录也当skill扫出来，扫出一堆垃圾。

说明书开头那个名字字段拼错一个字，整份skill作废，只留一条警告。它只认全小写加连字符的写法。

`~/.claude/skills`不在扫描名单里。我本机恰好没事，那个位置早就是一条指向公共抽屉的软链。但对一个只用Claude Code的人来说，那是个实实在在的文件夹，DSH看不见它。补救是一条命令的事，只是没人告诉你要补。

还有一件事得说在前面：这些都建立在你用标准模式的前提下。极简模式那份配置只有62行，搜一遍，skill这个词零命中（这条是读配置读出来的，那个模式我没实跑过）。**在那个模式下，你的技能库整个不存在。**

AGENTS.md和CLAUDE.md，它两个都读

用了个笨办法验证：在一个临时仓库里放三份规则文件，每份埋一个暗号，然后让模型把它收到的规则来源原样复述一遍。

第一轮，`CLAUDE.md`是指向`AGENTS.md`的软链，也就是我本机的真实结构。模型报回来两条来源，两个暗号，`CLAUDE.md`没有单独出现。

第二轮，换成一份内容不同的真文件。模型报回来三条来源，三个暗号，`AGENTS.md`排在前面。

所以规则很清楚：同一个目录下，两份文件内容逐字节相同就只算一份，漂移了就两份都塞进去。后一种情况你会白花一份token。

三个要注意的地方。全局规则的位置变了，DSH认的是它自己家目录下那份，你放在Claude Code那边的全局规则它读不到，得自己拷或者做软链。`.claude/rules/`那个目录和`@某个路径`这种导入写法，它明说不支持，用了就当普通文字原样塞给模型。还有，它没有文件监听器，你改完规则文件不会立刻生效，得等下一次读写操作或者上下文压缩之后。

MCP：口焊在主板上了，线要自己接

先说好消息：那个负责连接外部工具的客户端，本身就在默认安装里。工具的命名规则也和Claude Code一模一样，`mcp__服务器名__工具名`。

坏消息是它默认一个都不挂。把两份实际组装的配置dump出来搜了一遍，命中数是0。它也不会去读你现成的配置文件，无论是项目里的还是Claude桌面版的，整个源码和文档翻一遍，零命中。**你的MCP清单，它一个字都不看。**

我照着仓库里的例子挂了一个记忆服务器，全流程5分钟，其中4分钟在等npm装包。挂上之后让它存一句话，另起一个全新会话再问，原样答了出来。跨会话召回成立。

四个坑要知道。密钥得写在那段配置里，因为它启动子进程前会把看起来像凭据的环境变量全擦掉。服务器崩了不会自动重连，工具还挂在那儿，调用打进一个已经关掉的管道。它只桥接了工具这一类能力，另外两类官方明说暂缓。启动超时用的是60秒默认值，一个反应慢的服务器会拖慢整个启动。

要多装一条的那一列，藏着一个坑

表里那几项要多装一条，指的是同一批东西：仓库有219个包，装完你本机只有184个，差的35个（钩子桥、外包给别的产品的能力、语言服务、云端沙箱）全都发到npm上了，只是不会连带装上。**它们不是没发布，只是要你多敲一条命令。**完整清单和分组在§99速查表四；普通人会真心想要的，我数下来就五样：语言服务、持久终端、会话检索、换搜索源、还有那个能把活转包给隔壁产品的子代理。（他们考虑过「装上但默认休眠」，否掉的理由只有一句：休眠的零件不启动任何进程，包却还是会进到每一次生产安装里——为一个默认关着的功能让每个人白下245MB，他们不干。）

诚实交代验证边界：这35个里我自己装通并跑起来的只有钩子桥和Claude Code子代理这两组，其余的只验到「npm上确实有、版本是新的」，**一个都没装过，更没跑过。**

还有一下跟搬家直接相关。这些包在npm上的当前版本标签还停在旧版，照常规命令装会装到三天前那一版，接口跟你机器上的主程序对不上。我装钩子桥就吃了这一下，一口气吐了9条依赖警告，**包名后面加`@next`才拿得到跟主程序同一批的版本。**装完它还会提醒你这个包没接上去，意思是文件已经落到硬盘上，你还得再写一段配置把它挂进运行时。

我第一轮把这件事写反了

第一轮查到「35个包不在安装闭包里」，我顺手写下的结论是：**这35个包不随产品发布。**

这句话读起来毫无破绽。零件在图纸上，出厂不装，那不就是「只有下载源码才能用」吗。

第二轮做了一件很笨的事：把35个包名逐个打到npm的官方接口上，看它返回什么。

```
curl -s -o /dev/null -w "%{http_code}" \
  https://registry.npmjs.org/@deepseek-ai/dsh-lsp
# 200
```

35个包，35个都是200。它们全都发布了。

正确的说法应该是：这35个包已经摆在货架上了，只是你装主程序的时候不会连带装上，得单独再敲一条命令。

这两句话对你的差别是决定性的。「不随包发布」是一条死路，「多敲一条命令」是三分钟的事。而我差点把死路写进书里。

> 我把这段自我打脸留在正文里，是因为它本身就是这本书想教的那件事：**面对一个24小时前才公开的产品，任何二手结论都不可靠，包括我自己上一轮的结论。**「仓库里有、装上没有」和「压根没发布」，在所有能搜到的中英文资料里都被混成了一件事。唯一的解法是自己去打那个接口，看它返回200还是404。

一份Claude Code的hooks.json，真的拦住了它

钩子说人话就是「AI干每件事之前先问一句门卫」，Claude Code那边最常见的用法是提交前检查。

DSH请了一个能听懂Claude Code方言的翻译。翻得不全：按它自己说明书里的对照表，Claude Code那边30个事件它只认7个，剩下23个既不翻译也不吭声（这个总数是它的自述，我没有另行核对）。会话开始、每句话提交前、工具调用前后、停止、子代理起止，认的是这几个。

我写了一份最小的钩子配置，只有一条规则：拦住所有shell命令，理由写成一句能一眼认出来的暗号。挂上，跑一句最简单的任务。

模型的原话：

```
我跑了，但没跑成——bash 被拦下来了。
结果没有输出 hello，而是返回了一个错误：
Error: HUASHU-HOOK-BLOCKED: bash is forbidden by
the Claude Code hooks.json bridge
```

一份Claude Code的钩子配置，原样拦住了DeepSeek的工具调用。你在那边攒下的安全规则不用重写。

然后是一个更值得写的坑。把规则里要拦的工具名从`bash`改成`Bash`，也就是Claude Code里的正确写法，别的一个字不动，再跑一次。

它执行了。输出了hello。**钩子静默失效，一句报错都没有。**

原因是两边给工具起的名字不一样。Claude Code用的是首字母大写那套，DSH全是小写。规则匹配的就是工具名，对不上就整条不触发，而且它不会告诉你「我今天一个人都没拦」。

所以照搬一份钩子配置过来，多半不会报错。装好了，看着在跑，其实一条都没生效。

还有一处是能力缺口。Claude Code那边的「预先批准」这个动作，DSH侧根本没有对应物，它只有拒绝和问一下两种。翻译器再全也变不出来。

（Codex那套钩子我没实测，只读了文档，10个事件支持5个。这条标未验证。）

让DeepSeek去使唤我本机的Claude Code

这是我跑通的第二条重头戏。DSH设计上能把一个回合外包给隔壁的Claude Code或者Codex，发布当天他们把这个能力从默认安装里拆走了，理由后面说。装回来的过程计了时：3分37秒，多出259MB，其中245MB是一个预编译的Anthropic二进制文件，而这次装进去的DeepSeek自己的代码，一共464KB。

装完还得自己写一段配置把它挂上。然后我让DeepSeek的模型去问Claude一句「你是哪个产品的哪个模型」，把原话转述回来。

它转述回来的是：**「我是Claude Code里的Claude Opus 4.5。」**

DeepSeek的harness在我这台机器上拉起了Claude Code，跑完一个回合，把答案交回来了。

对我这种订阅用户，有一个细节比功能本身重要：**它故意不去覆盖Claude官方SDK的配置来源，所以那边照常读本机的登录态。整个过程不需要任何API key。**这是我原本以为会卡死的那道墙，结果它压根不存在。

三条边界得说清楚：外包出去的子代理拿不到父会话的对话和人格，不会在Claude Code那边留下会话记录，也不会弹窗等你确认。

最后一个坑是文档层面的。随包自带的配置文件里，那两行外包能力旁边还留着一句注释，写着「把停用标记去掉就能用」。你照做会打开一个永远等不到东西的空位，因为它点名要的那个零件已经不在包里了。仓库里两个相关说明文件至今还写着「随包配置已经加载了这个」。我查了提交记录，说明文件最后一次改是8月10号，把包拆走发生在8月13号，也就是公开当天。

三样确实搬不了的东西

会话记录，完全不通，而且不是暂时没做。Claude Code存的是消息，一句你说的一句它说的，用父子关系串成树。DSH存的是事件流，压缩过，解开一次36行的会话数了数，里面有推理片段、审批策略、沙箱模式、每一步的起止，44种事件类型里只有3种是模型看得见的（§05细说）。

这两个的关系像庭审速记和聊天记录：速记能还原出聊天记录，反过来做不到，因为大部分格子是空的。我在源码和文档里搜过导入器，零命中。

自定义斜杠命令，不支持。Claude Code里你在一个目录丢个markdown文件就多一条命令，DSH没这个入口。它一共6条，全是插件注册的，想加第7条得写代码。对照关系是这样的：

Claude Code	DSH
`/compact`	`/compact`，对得上
计划模式（快捷键切档）	`/plan`，做成了命令
`/export`	`/export`，只有网页版有
`/permissions`	`/permission`，单数，只能三选一
目录里丢md自定义命令	没有
—	`/goal`、`/feedback`，DSH独有

权限档位，别硬映射，会搬反。Claude Code的权限是一份逐条的名单：这条命令能跑、那个文件不能读。DSH是三个门禁等级加一个「要不要叫保安」的开关，没有按工具名的规则表。

最危险的是同一个词在两边意思相反，而且这里得说全。DSH里那个表示「不问」的开关，机制上是需要升权就直接驳回（§02讲过），但出厂配置只把它跟「完全放开」那一档绑在一起，而那一档本来就没有需要升权的操作。所以你体感到的是不弹窗也不拦。

Claude Code里字面意思相近的那个档位，是不弹窗直接拒绝。**同一个词，一边体感是全放行，一边是全拦死，照字面搬会搬到反面。**

补一句公道话：DSH的门禁等级比Claude Code硬得多，它是操作系统层面的，进程主动把权限交出去就要不回来；Claude Code的名单是程序自己在检查自己。这是两种可靠性，不是谁高谁低。

还得交代一句范围：本章的搬家实测全部是在无界面模式下跑的，钩子、MCP、外包子代理这三条路都没在网页界面里复验过一遍。审批弹窗那部分不在本章范围内，我另外在网页版里跑过，实拍的样子和链路写在§02。

他们本来想这么做，后来没做

> 前面说过，skill那份说明书开头的名字字段写错一个字，整个skill就作废。有人提过一个很人道的方案：**把驼峰写法也当成合法别名收下来**，别让人因为一个大小写丢掉整份技能。
>
> 否了。理由：外部格式就是Claude那套短横线拼写的约定，而这个产品还没发正式版，没有任何已发布的兼容义务。
>
> 这条对搬家的人很重要，因为它把这个团队的取舍摆明了：**宁可让写错的那份整个消失，也不留一个「我猜你想写的是这个」的兼容层。**好处是你永远不用猜某个skill到底以哪种拼写生效；代价是搬过来之后，值得花五分钟把所有skill的名字字段扫一遍。（出自 `2026-07-28-skill-invocation-policy` 那篇笔记的被拒方案一节）

它一个字没让我配置，就把我那57个skill全端走了。那它端走之后，塞到模型眼前的到底长什么样？这件事它允许你直接看。

§05 每一次运行都有迹可循

Every Run Leaves a Trace

这一节讲的功能不酷，没法做成发布会上的一页PPT，但凡是天天在用agent的人，都在这件事上吃过哑巴亏。我在本机跑了一句最普通的中文指令，然后把它留下的那份日志整个扒开了，包括官方文档里没有的那些东西。

我最想要的功能，其实是个很土的东西

你大概有过这种时候。你让agent改一个卡片的间距，它动了三个别的地方；你让它跑一个流程，二十分钟后结果不对。你回头看，只能看到一屏它写给你的漂亮总结，看不到它中间到底在想什么、看到了什么、哪一步开始跑偏。

这时候你能做的只有一件事：**猜**。猜是不是自己的话没说清楚，猜是不是它读到了某个不该读的文件，猜是不是上一轮的什么东西还留在它脑子里。猜完了再重开一轮，祈祷这次好一点。

最抓狂的时刻从来不是AI做错。是**它做错了，而你不知道它为什么做错**。做错可以重来，不知道为什么错，下一轮还是撞同一堵墙。

dsh的做法是把地基换掉，而不是在界面上加一个「查看详情」的按钮。

一句最普通的话，日志里落下44行

我在一个空文件夹里放了一个15字节的Python文件，然后打了一句中文进去：

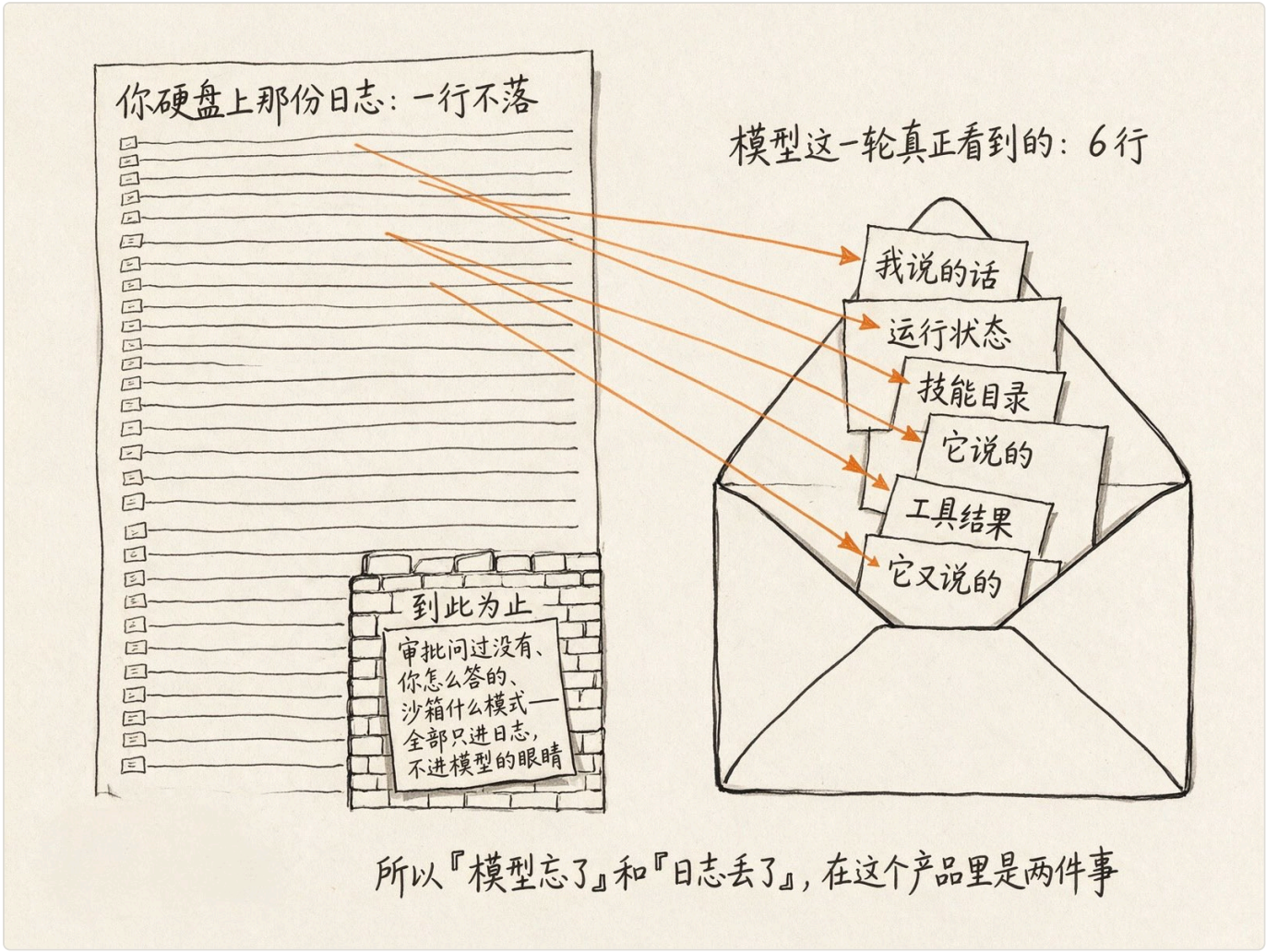
```
用一句中文说明当前目录下有哪些文件
```

它想了一下，跑了个 `ls -la`，回我一句「当前目录下只有一个文件 a.py」。前后两秒出头，从头到尾没有任何值得说的地方。

然后我去看它留下的会话日志文件，44行。这是完整的分布：

记录	行数	这是什么
`assistant/chunk`	13	模型一个词一个词吐出来的原始碎片
三种打包行	7	上面那种碎片连着来三条以上，就压成一行存
`user/message`	3	模型收到的三条：我打的那句、运行时状态、可用技能目录
`assistant/message`	2	两轮各组装出一条完整的助手回复
`step/start` · `step/end`	4	两次「想一下、调个工具」的起止
`agent/inbox/spliced`	2	我那句话进队列、出队列
标题相关	3	先拿前13个字凑合当标题，再叫模型起了个正式的
`turn/start` · `turn/end`	2	这一整轮的开和关
`tool/call` · `tool/result`	2	它决定跑`ls -la`，以及跑完拿回了什么
`request/header` · `request/context`	2	发给模型那个信封的完整快照
权限三件套	3	开工前把权限、沙箱、审批策略各记一条
文件头	1	会话id、创建时间、工作目录
合计	44	

两个补充。第一，44这个数是文件行数，不是事件数：那7行打包行每行装着好几条碎片，所以事件的编号一路排到了125。第二，这个产品一共定义了44种事件类型，跟这里的44行纯属巧合。



> 右边信封里那六张，是模型这一轮唯一看见的东西。左边被砖墙挡住的那些——审批问过没有、你怎么答的、沙箱什么模式——一条都没进它眼睛。注意44是文件行数不是事件数，而这个产品定义的事件类型也正好是44种，纯属巧合。

这44行里，模型只看得见6行

把日志逐行标一遍会发现一个不直觉的比例：44行里只有6行带着「模型可见」的标记，就是那三条 `user/message`、两条 `assistant/message`、一条 `tool/result`。

剩下38行，模型一个字都读不到。权限记录、沙箱模式、轮次边界、原始碎片、给标题起名字那次单独的模型调用，全部只进日志，不进上下文。

这不是我这一次跑得特殊。整个产品定义的44种事件里，**只有3种是模型看得见的**：你说了什么、它说了什么、工具返回了什么。剩下41种全是记给你和给审计看的。

一个类比，以及它在哪里失真：像一场庭审。书记员把一切都记下来，谁几点进来、哪次异议被驳回、休庭多久；但陪审团只听得到三样，证人的话、律师的话、呈上来的证物。程序性的东西他们一个字听不到。

失真在这儿：陪审团听到的是「现场发生的」，模型看到的是「现算出来的」。它的历史不是攒起来的，是每次发请求时从日志里重新投影一遍。所以「模型忘了」和「日志丢了」在这个产品里是两件完全不同的事。

我把它给模型看的东西，整个挖了出来

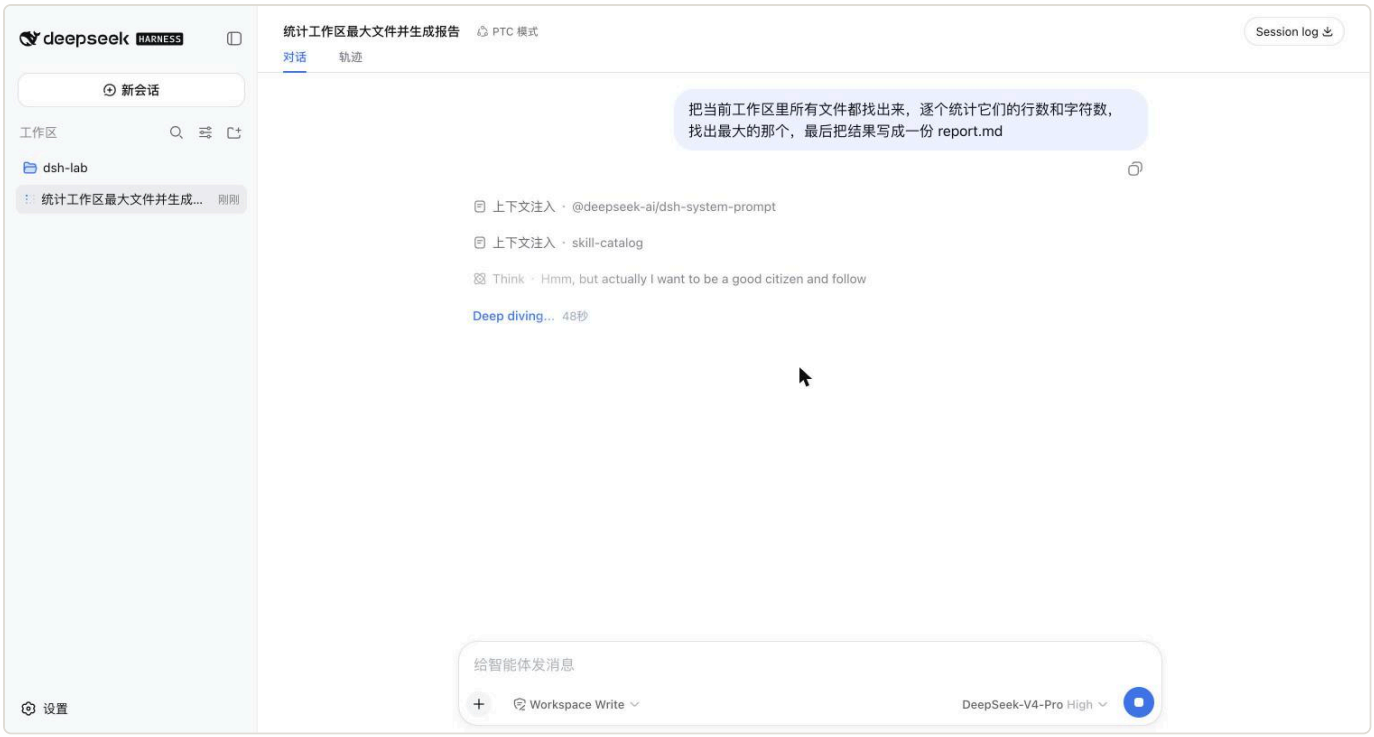
那两条`request/header`记的是发给模型那个信封的完整快照：用了哪个模型、多少输出上限、系统提示词的全文、工具清单的全文。

我从里面直接拿到了三样官方文档没有的东西。

一，**标准模式的完整系统提示词，4,100个字符**。逐字的原文，连标点都在。开头是`You are an AI agent powered by DeepSeek Harness.`，后面十几段全是给模型的行为规矩，比如「看文本文件用read工具，不要用cat这种shell命令」「每个bash结果上的退出码标记都要检查」「后台任务的编号要记住，别忙等，也别重复跑」。

二，那一刻挂着的25个工具，逐个有名有姓：读、写、编辑、搜索、跑命令、开子代理、发消息、定目标、起工作流。**这份名单出自我在无界面模式下的实跑**；网页标准模式那份也是25件，但内容不完全一样——这边有`str_replace_editor`，那边有`ask_user_question`，两边都是25纯属巧合。

三，本机那份技能目录，一万六千多字符，原样躺在一条`user/message`里。57个技能的名字和描述，我一个字都没配置过，它自己扫出来塞进去的。



> 那句话还没开始跑，上面已经并排注入了两条。一条来自官方那个 npm 包，另一条叫 skill-catalog，是我这台机器上的技能目录，它自己端走的。

值得强调的不是这三样东西本身，是**我拿到它们的方式**：没有反编译，没有抓包，没有问官方要。就是打开自己电脑上那个日志文件，往下翻。

方式就两条命令，你现在跑一遍也是同样的东西。先找到会话文件在哪：

```
ls ~/.dsh/sessions/
```

目录名是项目路径压平来的，一场会话一个子文件夹，里面躺着一个`session.jsonl.zstd`。它是压缩的，不是加密的，解开就能读：

```
zstd -dc ~/.dsh/sessions/*/session-*/session.jsonl.zstd | python3 -c "import json,sys; [print
```

我这一跑打出来是`4100 25`，就是上面那两样：4,100个字符的系统提示词，25件工具。把那两个`len()`去掉直接打印，出来的就是逐字的原文。这一节剩下的所有东西，都是从这条命令的输出里翻出来的。

「我没有偷偷给模型看别的东西」，而且这句话是机器在查

「模型看到的一切都会写进日志」听起来像宣传语。它在这个仓库里其实是一条写死的规矩，原文只有一行：

> **Model-visible ⟺ logged**：任何进到模型请求里的东西，都必须能从会话日志里重建出来；新增一种模型可见的输入，就必须新增一种会话事件。
>
> （出自仓库根目录的开发规约，中译为我所译。）

翻成人话：**它在跟你保证「我没有偷偷给模型看别的东西」**。而这句保证每次发请求之前都要被代码自己比对一遍，不是写在文档里等你相信。

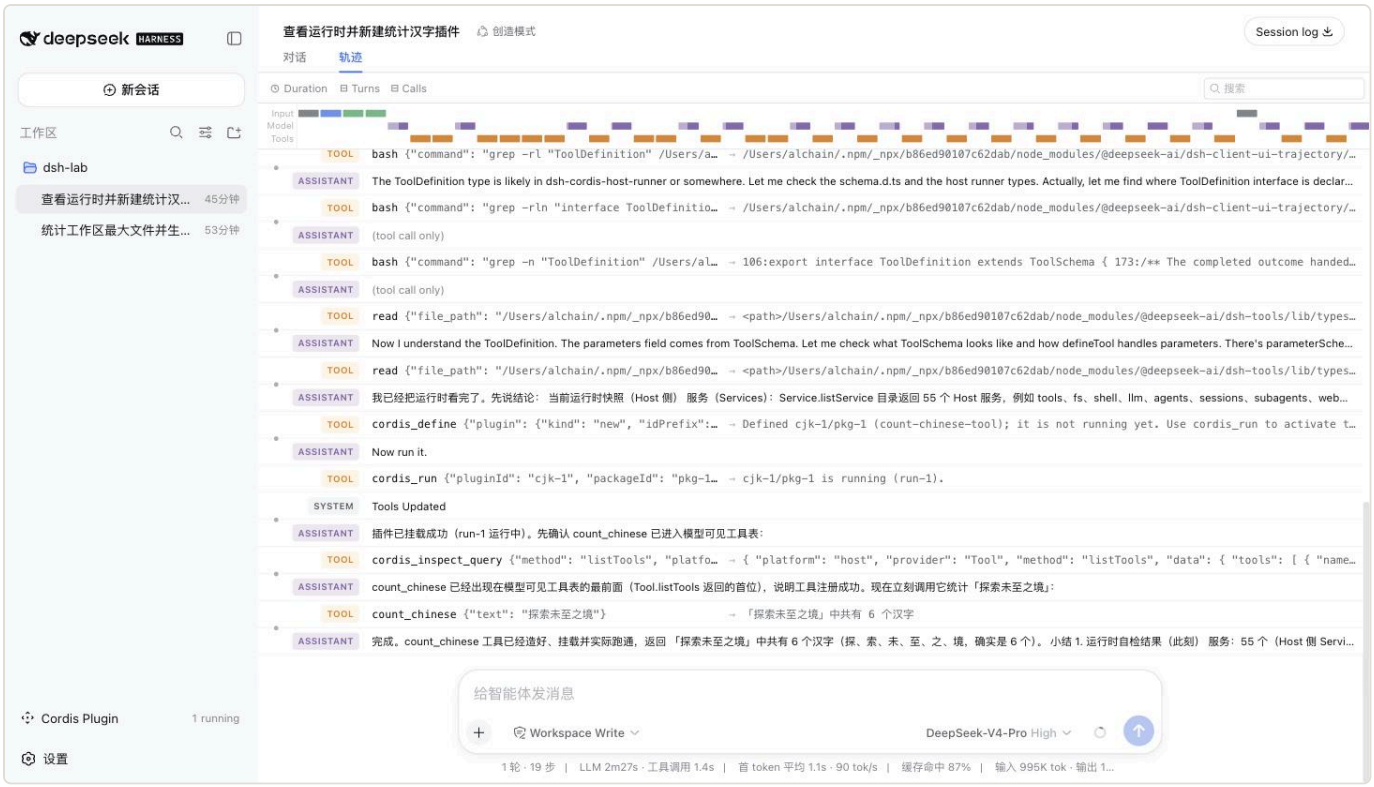
比对的动作很朴素：把这一次真要发出去的消息列表，跟「从日志重新算一遍应该得到的消息列表」放在一起比。对不上，抛异常，这次请求发不出去。模型、系统提示词、温度、输出上限、工具清单，六项逐项比。

在绝大多数agent产品里，「模型到底看到了什么」是一个没有人能回答的问题。系统提示词是运行时拼的，工具描述是运行时生成的，插件往上下文里塞了什么谁也不清楚，出了事只能猜。这条规矩把它从玄学变成了一个可以逐字节复原的事实。

他们自己排过这条规矩的三个好处，第一条挺有意思，说的是省钱：日志只追加不改写，投影出来的请求天然就是上一次请求的延长，模型那边的缓存自动就命中了。原话是「稳定性是涌现出来的，不是管出来的」。

界面上在哪看，以及那个难看的名字

网页版的会话里有一个页签，中文就叫「轨迹」。它长得像浏览器开发者工具里的网络面板：一行一条记录，粗线分轮次，顶上一条时间轴按真实起止时刻从左往右画。助手那条时间条还被切成两段，第一个字吐出来之前的等待和吐字的过程，颜色不一样，比例是真的。



> 中间 cordis_run 那一步下面跟着一行 SYSTEM · Tools Updated，那就是模型手里那份工具清单在会话中途被改写的那一瞬间。

会话头上另有一个按钮，叫`Session log`，也可以敲`/export`。点了把整份日志连同子会话和附件打成一个压缩包下来。这意味着一次出错现场可以整个发给别人。

轨迹里最戳我的是这个：**每一条塞给模型的上下文，都标着它是谁塞的**。我那次实跑的日志里，三条`user/message`的来源分别写着「用户」「`@deepseek-ai/dsh-system-prompt`」和「skill-catalog」，界面就照着这个渲染。

你可能注意到中间那个是个npm包名，挺难看的。这是他们主动选的。

更省事的做法是在界面里存一张对照表，把包名翻译成好看的中文。这个方案被明确否掉了：**一份恢复出来的日志，它的生产者可能已经不在了，但仍然得渲染出来**。标签从日志里读，那么一年前的会话、别人机器上产生的日志、某个已经改了名的插件，界面照样认得出来；存对照表的话，这些场景恰好全部失效。

代价他们也写明了：界面上会出现包名这种不好看的东西，想要好看的标签，自己在记录里写一个。**宁可丑，不可失真**。这是这个产品里最能说明它性格的一个决定。

分叉、恢复、回放，是同一份日志的三种读法

既然模型的历史是从日志现算的，有几件事就自动成立了。

分叉：你聊到第20轮觉得走错了，想回到第12轮换个说法重来，同时保留原来那条线。做法就是把日志前面那一段拷出来当种子，造一个新会话。一条规矩，切口必须落在一整轮的外面，切在半轮中间直接拒绝，不给你偷偷截齐。

恢复：关掉终端第二天再打开，就是把整份日志读出来当种子。

回放：拿一份真实日志当剧本，不用API密钥就能把整个流程重跑一遍。这本来是他们内部的测试设施，但对你的含义是，出错现场可以打包，别人拿到能复现。

这三件事在别的产品里通常是三套独立机制，这里是同一个动作的三种叫法，好处是它们的行为不会互相打架，因为根本就没有三份状态。

> **一条我没验的：**如果从第5步分叉，那第6到10步已经改过的文件会不会回滚？调研到最后也没找到明确答案。我倾向于认为不会，因为分叉复制的是日志，不是你的硬盘。**但分叉、恢复、回放这三样我一次都没实跑过，这条别照着我说的信。**

压缩不删原文，这跟你用过的产品都相反

你在别的agent里敲`/compact`，发生的事是：一大段对话被换成一段摘要，原文没了。你只能相信那段摘要，出了问题回不去。

这里不是这样。被压掉的事件**一条不少地留在日志原位**，只是从「模型可见的那一面」上被遮住了。摘要是另外新写的一条消息，上面标着「我遮住了第几条到第几条」。

我翻了仓库自带的一份压缩样本，被压掉那条事件的编号是4，它还在文件里，位置都没动。同一段还记着这段摘要是哪个模型写的、限了多少token、实际花了多少，所以「这段摘要是谁写的」有一个持久的答案。

顺带把三个容易混的东西分清楚。它们都在省地方，但省法完全不同：

	什么时候动手	模型还拿得回原文吗	要不要花钱
外溢	某一条工具结果太大，当场就存成文件	能，给了它文件路径，想看可以自己去读	不用，不调模型
裁剪	上下文压力上来了，先机械地把老的工具结果掐头去尾	不能，中间那段对模型永久消失	不用，不调模型
压缩	裁完还是满，才叫模型把老对话总结成一段	不能	要，一次模型调用

顺序是从便宜到贵。裁完会重新量一次，压力降下去了就不叫模型。三样都不改日志，改的只是模型那一面。

外溢和裁剪这两层实跑验过：造一个171,000字节的文件让它`cat`，截断当场发生，模型看到的结果里带着两行提示，一行说完整输出在哪个临时文件里，一行说被省略了14,385字节、完整结果存在哪、可以用read或grep去查。倒推回去，剩下的内容正好压在五万字节的上限以内，一字节不差。第三层的压缩没跑到——那一场没把上下文顶到需要叫模型总结的位置，所以压缩触发之后是什么行为，这本书里的每一句都来自源码和文档，不是实测。

一条实操信息：外溢出来的那些文件不在 `~/.dsh/` 里，在系统临时目录下面。而且每跑一次dsh就新建一个，不回收。我本机跑了几次之后有16个这样的目录，绝大多数是空的。哪天你要清干净，别只盯着 `~/.dsh/` 。

代价：一个echo命令，35条抹不掉的记录

代价也得说，而且这些代价基本都是仓库自己写下来的。

第一是磁盘。仓库里有一份最小样本，任务是「跑个echo然后回一个词」，留下35条物理记录；我那句「有哪些文件」是44行。这就是「每一次运行都有迹可循」的字面价格。他们自己也心疼，所以在存储层做了打包，连续三条以上的同类碎片压成一行，按他们的实测，一次真实编码会话能小掉六成左右。注意这是压缩存法，不是删内容。

第二是它会一直涨。自从支持跨项目恢复以后，这台机器上每一个项目的会话日志都堆在同一个目录里。原文写得很坦白：本决策不引入任何保留期策略。也就是没有自动清理。

第三是不兼容。会话格式的版本号现在是0，明说了老格式直接拒读，不做迁移。好在拒读时会说清方向，是你的日志太新该升级，还是这个版本不带升级路径。

第四是删不掉。只追加、不改写是这套东西全部好处的来源，也意味着你说过的每句话都留在硬盘上某个地方。你可以整个删掉那个文件夹，但你没法「从这段对话里精确抹掉某一句」，那等于打断整条链子。这是设计，不是疏漏，你得知道。

这套东西的收益全在「出事之后」，成本却是天天在付。顺的时候你完全用不上它；一旦不顺，它是我见过的agent里唯一能让你不靠猜的。愿不愿意为此付日志的钱，是个真取舍。

他们本来想这么做，后来没做

2026年6月20日有人提案：别再一条条存模型吐字的原始碎片了，只留组装好的完整消息就行。理由很实在，日志被这些几十字节的小东西撑爆了，信封比内容还大。

提案被否，否的理由分两半。一半是失败的流：模型吐了一半崩掉的内容只活在碎片里，撞上输出上限的那一轮甚至没有完整消息，只有碎片承载着账单。另一半是无密钥回放整个建立在这些碎片上。

结论一句话：除非有一个一点信息都不丢的替代品，否则不能删。

他们最后的选择是去压缩存储格式，不是去剪掉内容。这一条其实是整节的缩影，能省的地方省，但「记全」这条底线不动。

那两条 `request/header` 里装着的东西有多沉，得称一次重才知道。

§06 一次任务到底多少钱

What One Task Actually Costs

我是个订阅用户，Claude Code和Codex都按月付钱，付完就不再想这件事。DSH把这个习惯打断了。这一节把一个真实任务的账单从会话日志里逐项拆开，包括一笔完全没料到的开机费，以及那笔开机费里为什么有将近一半是我自己造成的。

先说结构：它没有「包月」这个选项

翻遍这个产品的凭据体系，一共四层：先看进程环境变量，再看你家目录下那份凭据文件，再看项目里的环境文件，最后看全局的环境文件。四层全是API key，没有一层是登录账号。仓库里也找不到「订阅」这个概念。

这跟我熟悉的东西不一样。Claude Code和Codex都是订阅为主、可以挂API key；DSH是只有API key，没有别的路。

	计费结构	你实际感受到的
Claude Code	订阅为主，也可挂API key	额度用完就停，不是账单变大
Codex	订阅为主，也可挂API key	同上
DSH	只有API key	不会停，账单一直涨

订阅制是包月自助餐：交完钱吃到额度上限，服务员把你拦在门口，说下个窗口再来。按量制是点菜，不会有人拦你，但账单一直在长。

所以我切到DSH最大的心理变化不是「变便宜了」，是**计价器打开了**。原来那种「反正已经交过钱了」的安全感没有了。

> **这个类比在哪里失真：**自助餐的额度是硬的，而Claude的订阅额度可以再买按量额度接着花，所以订阅的天花板也是软的。真正的区别是默认行为：订阅默认拦你，API key默认放行。

一个真实任务：三分钱，三十六秒

我造了个只有四个文件的小目录，丢给它一句话：读一遍这个目录下的所有文件，写一份README总结它们各自是干什么的，并列出你发现的问题。这是我真正会让agent干的那类杂活，不是跑分题。

它走了五步：先列目录，再并发读四个文件，然后跑了一段Python验证自己的结论，写出README，最后收尾总结。整个过程的账在会话日志里躺着，解压出来就能算。

模型	实际花费	耗时
deepseek-v4-flash	$0.0034（约2.4分钱）	35.7秒
deepseek-v4-pro	$0.0097（约7分钱）	45.4秒

换成人民币，flash这一跑是0.024元。一天跑100次这样的活，2.4元。官方已经公告2026年8月17日0点（北京时间，也就是8月16日16:00世界时）起改成分时计价。这里有个很容易读反的地方：官方脚注写的是「低谷价是高峰价的一半」，说的是新价内部的峰谷关系，不是「新价比现价贵一倍」。逐项对下来，新的高峰价是现价的三到五倍。按高峰价算，同一件活约0.09元，一天100次9元。

作为对照，Anthropic自己的文档里给了Claude Code按量计费的实测：平均每个开发者每个活跃日约13美元，90%的用户低于30美元一天。这两个数出自他们自己的文档。两边的任务、模型、工具集都不一样，谁划算这里不下结论。

两个模型之间倒是有个干净的发现：它们的token用量几乎一模一样，未命中输入差2.3%，输出差11%。**贵三倍的那部分，全部来自单价，不是因为pro更啰嗦。**至于哪个答得更好，我没做盲评，不填这个分。

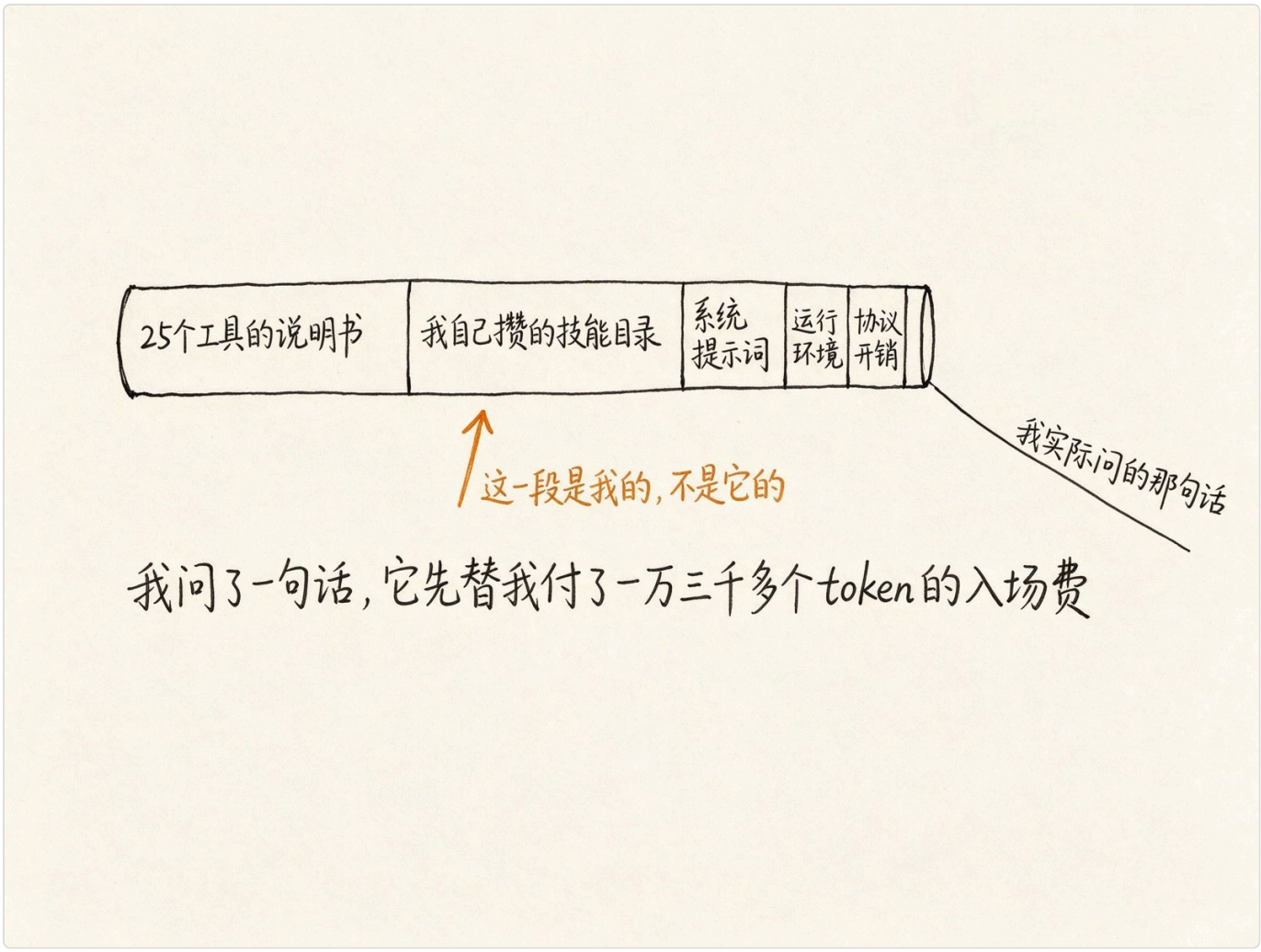
还有个细节：第三步那段Python是它自己决定要跑的。任务里没让它验证，它在最后的回答里补了一句「我还实际跑了一遍验证」，把库存总值和低库存的编号都算出来了。

你打第一个字之前，账单已经13,809

把这笔账拆开之后才发现，真正花钱的地方跟我想的完全不一样。日志里那次请求的系统提示词和工具清单，我逐块发给API称重，一项一项做差。第一次请求的未命中输入是13,838个token，构成是这样：

组成部分	token	占比
25个工具的说明书	6,510	47.0%
本机skill目录清单	6,242	45.1%
系统提示词	844	6.1%
运行环境快照	129	0.9%
协议本身的固定开销	82	0.6%
我实际问的那句话	29	0.2%

一句话版本：**我问了一句29个token的话，harness替我先付了13,809个token的入场费。**（表里六项是逐项做差称出来的，相加是13,836，跟日志里13,838那个真值差2个token，来自取整。这类差值法有±2量级的误差，别拿计算器较真。）



> 值得跑一趟的是最右边那根很长的引线——那条几乎看不见的窄缝才是我真正问的那句话。六项是逐项做差称出来的，相加13,836，日志里的真值13,838，差2来自取整。

打个比方。你跟一个新来的助理说「帮我看看这几个文件」，但每次开口之前，都得先把员工手册、工具间清单、公司技能培训目录完整念一遍给他听，因为他上一句说完就失忆了。你那句话三秒，念手册二十分钟。

工具那6,510里最贵的三件是流程编排、命令行和一个专门做字符串替换的编辑器（代码里叫 `workflow`、`bash`、`str_replace_editor`），加起来2,344个token，占工具预算的36%。最贵的那件是 `workflow`，979个token，将近我那句提问的34倍，而它在这次任务里一次都没被调用。（这份工具清单来自我在无界面模式下的实跑；网页标准模式那份也是25件，但内容不完全一样，§99说了这个巧合。）

这张表你能自己称一遍，三步，我敲的就是下面这三条。

第一步，把那个信封从日志里捞出来。它躺在 `request/header` 这条记录里，系统提示词全文和工具清单全文都在：

```
zstd -dc ~/.dsh/sessions/*/session-*/session.jsonl.zstd | python3 -c "import json,sys; [print
```

我这一跑打出来是 `4100 25`：4,100个字符的系统提示词，25件工具。路径里那个星号会把所有会话都卷进来，只想看一场，就把它换成那一场的目录名。

第二步，把每一块单独发一次给API，只为了看那张回执。 `max_tokens` 设成1，它就不会真回答你，只把账单回给你：

```
curl -s https://api.deepseek.com/chat/completions \
  -H "Authorization: Bearer $DEEPSEEK_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"model":"deepseek-v4-flash","max_tokens":1,"messages":[{"role":"user","content":""}]}'
```

回执里那个 `prompt_tokens` 就是这一块的重量。上面这条发的是一条空消息，量的是协议本身的固定开销；把 `content` 换成第一步捞出来的系统提示词全文再发一次，两个数一减，就是表里「系统提示词」那一行。

第三步，工具那一栏用减法。整份工具清单发一次，再拿掉其中一件发一次，差出来的就是那一件的价钱。表里最贵那三件，就是这么一件一件减出来的。

那笔入场费里，45%是我自己带进来的

清单第二行那6,242个token，我一开始以为也是产品自带的。不是。

它来自一条注入消息，内容是一份可用技能清单，列着57个skill的名字和简介。这57个不是DSH的，是我自己那个skill池，本来给Claude Code、Codex、Kimi Code三家共用，放在家目录下的 `~/.agents/skills` 里。

DSH装上就把它们全端走了。不用配置，不用迁移，也没问过我。

好的那面是：如果你跟我一样攒了一堆skill，最担心的「我的家当能不能搬过去」，在这一项上答案是不用搬。它读的就是那个目录，开箱可用。这是§04那张搬家清单里最舒服的一格。

另一面是账单。**这57个skill让每一次会话的开机成本从大约7,600涨到13,838，翻了82%。**而且它不是一次性的：每开一个新会话付一次，每派一个子代理付一次。这次任务真正用到的skill数量是零。

我全局配置里本来就写着一条规矩：新写skill的简介控制在200字符内，因为三家agent每次会话都全量载入所有简介，这是公摊成本。现在多了第四家来摊。

> **两处诚实标注。**约7,600这个数是13,838减去实测的6,242算出来的，不是在一台干净机器上真跑一次得到的，误差在几十token量级。另外注入清单里是57条，跟我那个目录的57个数量对得上，但逐名比对有两条不在里面、另有两条来路不明；DSH同时也扫项目目录下的skill，这两个来源我没有拆开验证。数量级和结论不受影响。

为什么第二句话开始便宜50倍

看到13,838这个数，第一反应是每轮都要付一次，那还得了。实际不是。

模型读你的提示词，是边看边把每个字加工成一堆中间状态，这个加工要花算力，所以要收钱。加工完的东西可以留着，下次如果开头一大段一模一样，直接取用，这就是缓存命中。那「开头一大段」有个名字，叫**前缀**

——从第一个字算起、连着不断的那一段；后面说的「前缀稳定」，意思就是这一段一个字都没变。这一章跑的是flash，它的缓存命中价是未命中价的1/50（pro那一档是1/120，§07用一场pro会话算过）。

想象一个咖啡师，每天给你做一杯很复杂的手冲。前15分钟是固定动作：磨豆、烧水、温杯、折滤纸，最后30秒才是今天加不加奶。聪明的做法是把前15分钟的成果留着，你第二次来直接跳到最后30秒。但只要你说今天换个豆子，或者把顺序反一下，哪怕最终味道一样，准备成果全部作废。

flash那一跑的五步流水，把这件事画得很清楚：

步	未命中（全价）	缓存命中（1/50价）	输出
1	13,838	0	190
2	267	13,952	332
3	786	14,464	1,384
4	319	16,512	1,442
5	177	18,176	449

第一步13,838全价，缓存是0，库房空的。第二步整个前缀被认下来了，全价计费的只剩267个。全程缓存命中率80.4%。

还有一个细节挺妙：那四个命中数全部是64的整数倍。13952、14464、16512、18176，除以64余数都是0。DeepSeek官方文档写着「64个token是一个存储单元，不足64的不缓存」，这句话在我这台机器上被验出来了，零头真的不给你算。

更值钱的是第四次跑。同一个目录、同一个任务、同一个模型，跟上一次隔了2分21秒，但换了一个进程、一个全新会话。开机那13,838个token里，命中了13,824个，全价的只剩13个。那一跑步数更多、输出更多，成本却只有上一跑的52.5%。

作为对照，Anthropic文档里写着他们的缓存寿命：订阅状态下一小时，用按量额度时掉到五分钟，用API key或云厂商接入默认五分钟。DeepSeek这边是几小时到几天，而且缓存写入不收费。这是两种不同的设计选择，不是谁好谁坏。

> **一个容易被抄错的地方：**「命中比未命中便宜50倍」说的是单价。落到总账上没有50倍，因为输出token不打折。同一跑如果缓存全部失效，成本从$0.0034变成$0.0121，是3.55倍。**「省50倍」这种话不能写。**

他们把「会不会打断缓存」做成了219张必填卡

缓存这件事，在这个仓库里被做成了一条由机器强制执行的写作纪律。

每个包的README必须以一个固定结构结尾，其中有一节叫`KV Cache effect`，专门写「我这个零件会不会让缓存失效」。填法规定得很细：要区分只追加、稳定重复前缀、替换更早的内容、独立发一次请求这四种行为，然后点名本包的哪些改动会打断复用。

执行情况是219个包里215个填了。剩下4个走白名单豁免，而豁免理由逐条写在门禁脚本的源码里，脚本还会检查这4个名字是不是真实存在的包、理由是不是空的、有没有人同时挤进两份名单。那个检查脚本本身38KB。

动机写在那条规则的来源笔记里：一个包的README讲得清API，却答不上真正主导成本的那几个问题，比如它的什么东西会进到模型请求里、那些token待多久。

对我这种不写代码的人，这一节反而是全仓库最好用的结构。**想知道某个能力贵不贵，打开它的README直接跳到那一节**：写着append-only的便宜，写着「从第一个变动的token起复用失效」的贵。加图那个包就老实写了，加一张图会让请求后缀失效。

由此有三条可以直接照做的：不要在会话中途改配置、换人格、开关工具，要改就在新会话开头改；要贴图就开头贴，别贴到一半；顺序也算数，工具重排一次，内容一个字没改，缓存照样掉。

PTC模式的账：省的不在前面，在后面

四个模式里那个「PTC模式」，界面上说它有标准模式的全部能力，还能让模型用一个程序把多步操作串起来。听起来把25个工具收成1个，应该更省。我跑了一次，结果是反的。

宽松对照：flash跑一句话任务，对上pro跑一个多步任务	标准模式	PTC模式
系统提示词	4,100字符	35,643字符
模型看得见的工具数	25个	1个
工具说明书体积	26,894字符	897字符
首轮输入	13,818	15,040

工具说明书确实从26,894字符缩到897，但系统提示词从4,100涨到35,643。那25个工具的定义没有消失，只是从一种格式换成了另一种，搬进了系统提示词。首轮真实输入反而多了约9%。（这张表的标准模式一侧是另一次会话，所以13,818跟前面那张账单的13,838差了20个token。）

它省的是别的东西。那场会话里，模型真正开口调用工具只有5次，而这5段程序在沙箱里实际发起了15次操作。中间那10次的输出，文件列表、脚本的屏幕输出、每个文件的原文，从头到尾没进过模型的眼睛。**省的不是说明书的体积，是中间结果的体积。**任务越碎、中间产物越大越划算；只有一两步的活，纯亏。

这场的缓存数也印证了前面那件事：140,672个命中token对17,591个新输入。那35KB的系统提示词只在第一轮全价，之后每轮都吃缓存价。前缀稳定这件事，在账单上是看得见的。另外它的输出里有76%是思考token，不打印在界面上，但照价收费。

这张表的限定必须原样带上：两侧的模型和任务都不同（标准模式用的是flash加一句话任务，PTC用的是pro加一个多步任务）。所以它只能说明「固定开销的构成变了」，不能据此下总账谁划算的结论。**同模型、同任务、同批文件的严格对照，我在§08补做了**，那一组的数字和结论以§08为准，别拿这张表去算总账。

三个只有自己跑一次才会撞上的坑

第一个坑最贵。我用命令行明确指定跑flash，日志里写的却是pro。原因是同一时间另一个窗口开着网页版，我在网页上点了一下模型下拉框，那个选择被写进了共享的全局设置文件，而这个文件的层级压过命令行参数。**在网页里点一下模型，会永久改掉你所有命令行任务的模型和账单，命令行上没有任何提示。**3倍差价，静默发生。跑之前先看一眼那份设置文件，跑完先看日志里的请求上下文。

第二个坑：这个产品自己不算钱。整个仓库搜一遍，没有任何一处把token换算成金额。界面上那个上下文占用的进度条，用的尺子是「4个字符等于1 token」，量英文差不多，量中文会短一半：我实测中文密度是1.71个字符一个token，进度条显示40%时可能实际已经90%。它不是全程瞎猜，每次模型回话都会带回真实数字做校准，但中文长对话里偏差越攒越多。

第三个坑：默认的思考强度是最高档。flash那一跑的3,797个输出token里，有1,444个是思考token，占38%，全部按输出价计费。调低理论上能砍掉这部分，但质量会怎样我没测，不替你说。

那你该怎么估自己的账

如果你也是从订阅那边过来的，三条够用了。

一，把「一次多步杂活约三分钱」当锚，然后按你一天的任务数乘一遍。真正的变量在于你会不会因为便宜就无节制地开新会话，因为每开一个新会话都要重付一次入场费。

二，先去看一眼你的`~/.agents/skills`有多少个：

```
ls -d ~/.agents/skills/*/ | wc -l
```

这个数字直接决定你的开机成本，而且它是你自己的资产带来的，不是产品的锅。同一句话，一台攒了57个skill的机器和一台干净机器，入场费差82%。

三，遇到「便宜105倍」这种标题要停一下。那个数是真的，但它测的是模型：评测机构Artificial Analysis的同一套题跑下来，V4-Flash的模型API花费是某个前沿模型的1/105，代价是评测分数低10分。**那是模型价差，不是harness价差。**真正测harness的是Composio那组实验：固定同一个模型，套8个不同的agent工具，每次成功的成本差7倍（完整数据在§14）。这两个数我都没独立复现，是照着他们公开的方法学读的。这两件事互相独立，别把它们乘在一起。

他们本来想这么做，后来没做

自动压缩会在对话中途触发，那一刻厂商的缓存刚被上一次请求焐热。默认的摘要器却另发一个请求、用一套专门的摘要系统提示词，结果同一段历史被按全价付了两次，而且恰好发生在对话最长的时候。修法是把摘要指令从请求头部搬到对话尾部，让它变成热请求的延长线。

真正见功力的是三条被驳回的方案，全是省钱的直觉：「保留摘要器自己的系统提示词，其余复用」，驳回，因为系统提示词是厂商最先缓存的那一段，头一变，后面写什么都没用；「只发被遮住的那段历史，不带头」，驳回，第一个token就分叉了；「摘要器又不调工具，把工具说明书省掉」，驳回，工具说明书是被缓存序列的一部分，省掉它后面每个token全部错位。

结论反直觉得可以直接记住：**明知道摘要器一个工具都不会用，还是要把那份两万多字符的工具说明书原样带上，因为少传反而更贵。**（出自 `2026-07-21-compaction-summary-prefix-cache-reuse` 这篇笔记的被拒方案一节）

入场费里最贵的一块，是那份25件的工具清单，6,510个token。而在创造模式里，这份清单是32件。下面这一场，它自己把它变成了33件。

§07 它给自己长出一只手

It Grows Itself a Hand

这一节的东西全是我自己在本机跑出来的，一步没省。让它当场给自己造一件新工具，它做到了。但真正值得看的不是它做成了，而是它做成之前那十四步在干什么，以及这只新长出来的手能活多久。

它少一件工具的时候，我该等谁

用任何agent久了，都会撞上同一堵墙：你要的那个动作，它手里恰好没有。

比如我想知道一段文字里有多少个汉字。听着简单，但工具清单里没有这一项。以往的解法只有三条：等官方更新、去外面找一个装上、或者自己写一个再重启。

这三条路有个共同点，都得离开当前这场对话。你正想着的这件事，被硬生生切断了。

我想试的是第四条路：**能不能让它在对话进行到一半的时候，自己把缺的这件工具长出来，然后立刻用上。**

§03提过的创造模式，就是给这件事准备的。起了一个干净的隔离环境，什么都没预先配置，直接把话丢过去。

我打进去的那句话，和它交回来的东西

先交代一个词。**运行时**指的是它此刻真的在跑着的那一套东西——有哪些零件在转、有哪些工具挂着，都算在里面。下面这条指令里那个「服务」，是这个产品给零件起的正式叫法，这一节你按「零件」读就行。

> 先用 `cordis_inspect` 看一下你自己此刻的运行时，告诉我你看到了什么（服务、插件、工具各多少个）。然后给你自己现场造一个新插件：给模型新增一个叫 `count_chinese` 的工具，作用是统计一段文本里的汉字个数。造完把它挂上去，然后立刻用它统计这句话：「探索未至之境」。

三分多钟之后它回话了，中间我一个字都没再输入。它自己写的小结是这样的：

它报的项	自检时	现在
服务	55个	55个
模型能看见的工具	32个	33个（新增 `count_chinese`）
动态插件	0个	1个（ `cjk-1` ）

那个32要跟前面几章对一下账：**32等于标准模式那25件工具，加上创造模式多给的7件自指工具。**那7件是什么，这一节后面会拆。

然后是那一句实测调用：给`count_chinese`喂进去「探索未至之境」，它返回6。

答案是对的。探、索、未、至、之、境，六个字全是汉字。

> 32 变 33，多出来的那件是它自己刚造的。左下角还亮着「Cordis Plugin · 1 running」——这只手此刻活着，重启就没了。

这件事本身其实不算稀奇，一个模型现场写段代码算个数，谁都见过。稀奇的是它不是算给我看，是把这段能力焊进了自己身上，焊完之后这件工具就跟原来那32件平起平坐地挂在那儿，我随时可以再叫一次。

前十四步，它在读自己的说明书

整场任务它走了19步、发出24次工具调用。我把日志一条条过了一遍，发现一件比结果更有意思的事：**前14步、20次调用，它一件正事都没干，全在读它自己的说明书。**

它干了什么	次数
查自己此刻的运行时（三个 `cordis_inspect` 开头的工具）	6
在自己的安装目录里翻文件结构（跑命令）	6
读类型声明文件	4
搜 `defineTool` 这个函数到底写在哪	3
加载一份叫 `cordis-plugin-development` 的技能	1

最后那一条是它自己想起来的，我没提。它的第一步就同时干了两件事：一边查运行时，一边把这份「怎么写动态插件」的技能调了出来。

它读的文件长这样：

```
~/.npm/_npx/.../node_modules/@deepseek-ai/dsh-tools/lib/types/types.d.ts
```

这是个类型声明文件，可以理解成一份零件的规格书：这个函数吃几个参数、每个参数长什么样、返回什么。它先后读了三份这种东西。

一个agent在读自己身上零件的规格书，为的是给自己再装一个零件。这个画面比任何架构图都说明问题。

它不是凭记忆写的。凭记忆写也能写出个大概，但那样写出来的东西挂不上去，参数名对不上、注册方式对不上，会在挂载那一步被拒绝。它选了先查再写。

真正动手只花了最后4次调用：提交定义、挂进活的运行时、回头确认工具数变成33、然后用一次。

它写的那二十行

下面是它提交上去的全部代码，一个字没改：

```
return {
  apply(ctx) {
    harness.registerTool(ctx, harness.defineTool({
      name: 'count_chinese',
      description: '统计一段文本中的汉字（中日韩统一表意文字，含扩展区与兼容区）个数，返回汉字数量。',
      parameters: {
        text: { type: 'string', required: true, description: '要统计的文本' },
      },
      output: {
        schema: { type: 'integer' },
        render(args, value) {
          return [{ type: 'text', text: '「' + args.text + '」中共有 ' + value + ' 个汉字' }]
        },
      },
      async execute(args) {
        const matches = args.text.match(/[\u3400-\u4dbf\u4e00-\u9fff\uf900-\ufaff]/g)
        return matches ? matches.length : 0
      },
    }))
  },
}
```

二十行。我不写代码，但这一段里有个细节连我都看出来了。

中间那行以`const matches`开头的，是用来判断「这个字算不算汉字」的。它写了三段范围，不是一段。第二段是我们平时打的那些常用汉字，第一段是扩展区里那些冷僻字，第三段是兼容区。

随手写的话，写一段常用区就交差了，测试用例也能过，因为「探索未至之境」六个字全落在第二段里。它多写的那两段，在这次任务里一次都没用上。

这就是「查着写」和「凭印象写」的区别。

只有翻日志才看得见的那一下

还有一处，是我事后扒日志才发现的，界面上完全看不出来。

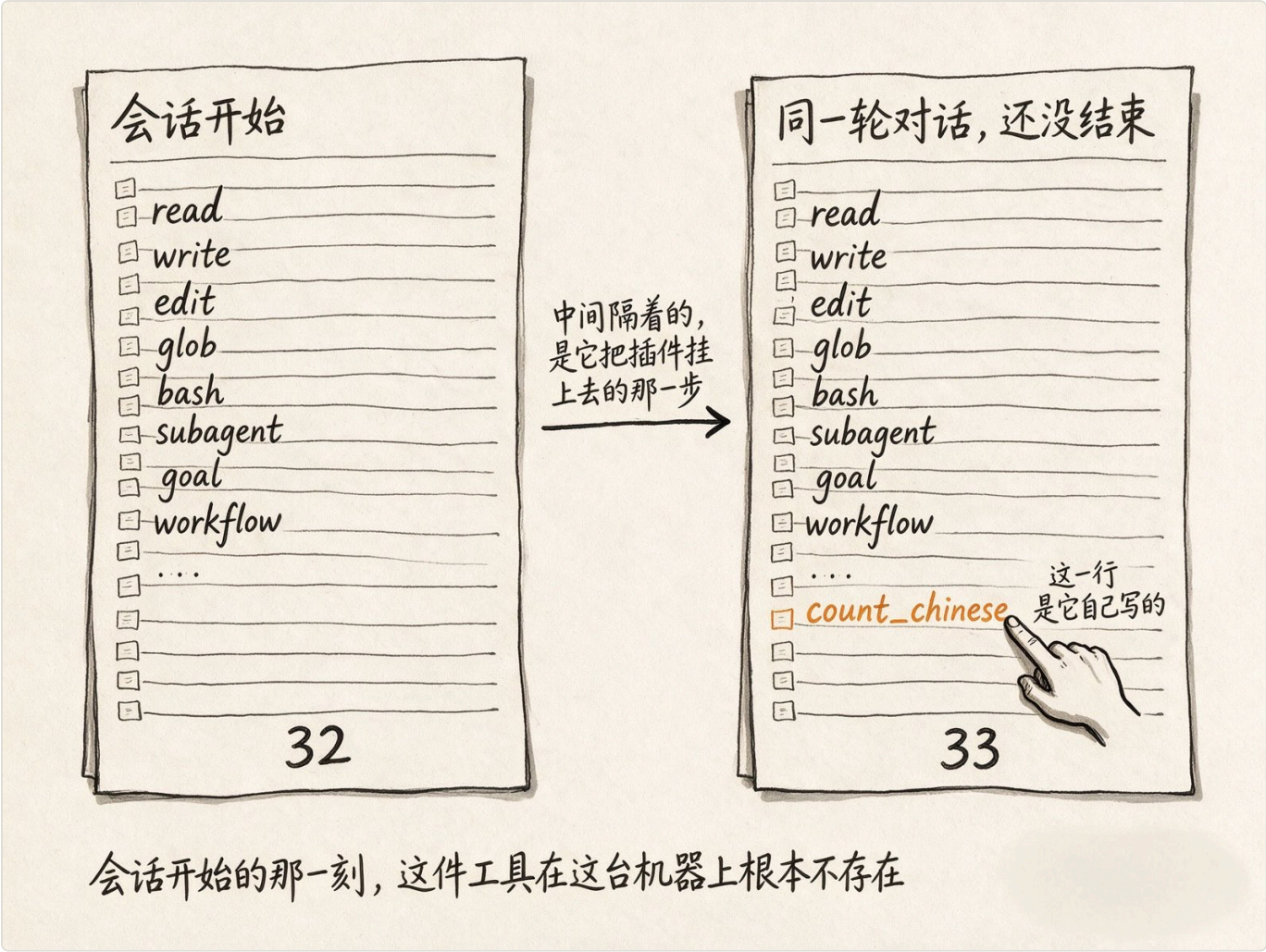
每次请求发给模型之前，系统会把「你现在有哪些工具」这份清单塞进请求头里。这场会话的日志只在这份清单变化时才记一条，整场只记了两条。数一数就知道：

```
zstd -dc ~/.dsh/sessions/*/session-*/session.jsonl.zstd | grep -c '"type":"request/header"'
```

这一场打出来是`2`。把两条各自写着几件工具也读出来：

```
zstd -dc ~/.dsh/sessions/*/session-*/session.jsonl.zstd | python3 -c "import json,sys; [print
```

两行：`initial 32`、`change 33`。第一条在开头，写着32件工具。第二条在第17步，写着33件，多出来的正是`count_chinese`。连它自己给出的记录理由都写在那儿——一条是开局，一条是变了。



> 两张纸的工具名和顺序一模一样，差别只在末尾那一行。会话开始的那一刻，count_chinese在这台机器上根本不存在。

两条之间隔着的，就是它把插件挂上去的那一步。

一轮对话还没结束，模型手里那份工具清单就变了。会话开始的那一刻，这件工具在这台机器上根本不存在。日志忠实地把这件事记了下来，而且是自动记的，没人为它专门写过一行代码。

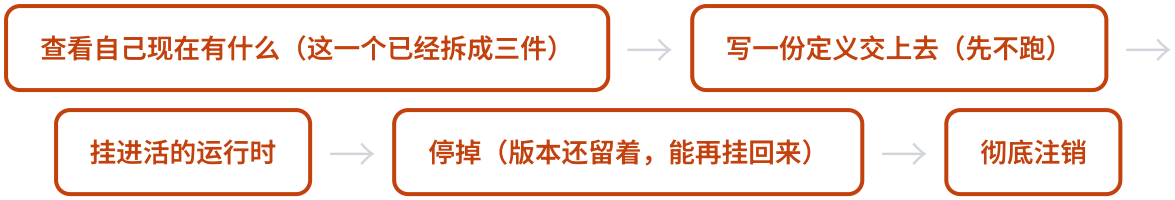
这是「一切皆插件」这句口号最字面的一次兑现。它不是说所有功能都做成了插件那种形态，是说插进去这个动作，在运行当中就能发生。

五个动词，实际是七个

创造模式多出来的这套自指工具，官方那份说明文档写的是五个：查看、定义、运行、停止、注销。

我在运行时里实测到的是**七个**。那个「查看」已经被拆成了三个，分别对应列出来、按条件查、和查自己。

拿人话对一遍，它们分别是：



五个变七个这件事，本身不算什么大事，一个功能拆细了而已。值得写出来是因为**差异出现在工具名字上，而名字是要念给模型听的**。文档落后于代码，在这个仓库里不是个别现象，前面几节已经撞见过几回了。这一次落后的东西，恰好是最不能落后的那一类。

所以这本书里凡是出现具体工具名的地方，一律以我实跑到的运行时为准，不照文档抄。你看到这段话的时候，它可能又变了。

这只手活不过一次重启

边界必须说清楚，不然上面那些话就成了吹牛。

它自己在小结的最后主动交了底：这个插件目前还在运行中，不需要的话可以停掉（版本留着，随时能恢复），也可以永久删除。

官方文档写得更死。动态包只活在这个进程的内存里，然后是连着的五个「不」：

> 不创建任何插件文件、不安装任何包、不改动任何配置、重启之后不存活、也**不能被自动提升为正式插件**。

最后那条最要紧。它长出来的这只手，没有任何机制能把它沉淀成一件真东西。想留下来，得有人按正常流程去写一个真的插件、装进去、重启。**它能造，但它不能给自己转正。**

还有一句风险声明，他们自己写的，我原样搬过来：

> 这个沙箱隔离全局变量，**但它不是安全边界**……请把这套工具当成bash权限来对待。

翻译成人话：开在这个模式上的会话，你要当成「我把这台电脑的命令行直接交出去了」来对待。因为模型写的那段代码，是拿去对着活的运行时求值的。它不像别的工具那样有一层壳兜着。

所以这个模式不是默认给你的，得你自己去选。这一条是他们故意留的门槛，理由写在了这一节末尾那个栏目里。

这一趟的账单长什么样

顺手把这场会话的账摊开，因为它正好是个漂亮的样本。

项	值
步数 / 工具调用	19步 / 24次
系统提示词	17,663字符
那32件工具的说明书	33,867字符
新输入token	131,494
输出token	11,349
缓存命中token	863,232
模型实际耗时	2分27秒

看最后两行的对比：**缓存命中量是新输入量的6.6倍。**

一场翻了几十个文件、来回19步的长会话，真正按新内容计费的只有13万token，剩下86万都是重复的前缀，命中了缓存。

这就是「前缀稳定」在账单上的样子。这个词§08讲PTC模式的时候还会反复出现，在这里你先看见它值多少钱：按官方定价页当天的价格，**缓存命中的单价大约是未命中的一百二十分之一**。（这一场跑的是pro，所以是一百二十分之一；§06那一场跑的是flash，那一档是五十分之一。两个模型两个比值，别串。）

拿那张价目表算了一下，这一趟大概七美分。如果那86万token全部按新内容计费，账单会翻到六倍多。这个算术是我自己做的，不是它给的，而且那张价目表从2026年8月17日0点（北京时间）起就改成了分时计价，你照着算的时候数会不一样。

内测开发者的判断，和我要加的后半句

内测开发者JY在公开的一段观察里，把这件事叫做「自进化软件」的雏形：

> 再往前想一步，这其实有一点「自进化软件」的雏形了。DSH现在已经可以让Agent检查自己的runtime，现场写一个插件并挂载上去，然后在后续的任务里直接使用这个刚刚获得的能力。
>
> 当然，现在这部分还比较实验性：动态生成的插件只存在于内存里，重启就没了，也还不能自动沉淀成一个永久插件。
>
> ——JY（@jiayuan_jy），原文见 https://x.com/jiayuan_jy/status/2087911060154314963

他说的每一句，我这场实测都对上了，包括后半段那个限制。所以这句判断我同意，但得连着后半句一起写：

它现在能给自己长出一只手，但这只手活不过一次重启。

以前我们讨论「AI能不能自己改自己」，讨论的是它有没有这个能力。现在能力这一关，在这台笔记本上、在三分钟里，已经过了。剩下的问题变成了另一个：**长出来的东西，凭什么留得下来？**

他们本来想这么做，后来没做

这套自指工具，他们**明确决定不装进默认的那份组装里**。理由不是没做完，是那层隔离壳的位置不对：模型写的代码在harness自己的进程里求值，而管束程序的那道机制只管得住它另外派出去的进程。写在笔记里的原话是，在网页界面上，沙箱和审批这两道关是被绕过而不是被执行。

更值得学的是这个决定被记下来的方式。那篇笔记专门开了一节，标题叫「什么被留在外面，以及为什么」。作者写明这一节存在的目的，是让「我们忘了装」和「我们决定不装」这两件事保持可分辨。

同一个模式还有一条被否掉的路：教agent用设置页上那个「有没有坏」的字段来自查作品。否决理由是它对每一种真正要命的失败都放行，而**把它当成验证来呈现，恰恰是原来那份指引让人感觉「已经完整了」的原因**。看起来完整，是它最危险的地方。

它能改的是自己的运行时，改不了自己这一辈子——那份配置在启动那一刻就冻住了。那份冻住的配置就叫「模式」。进门的时候你已经挑过一次了，往下这一章算的是这个选择在账上到底意味着什么。

§08 让模型写程序，以及那笔反直觉的账

Let the Model Write a Program — and the Bill

进门那两页只让你答了一道四选一：用标准模式。这一章拆的是那道题背后的机制账——PTC模式把模型能点的工具从25件收成1件，这笔账到底怎么算；另外两个模式又各自换掉了什么。我在本机跑了两场会话，同一个模型、同一个任务、同一批文件，只换模式。结果跟直觉正好反着。

先看它现在是怎么被使唤的

你让它干一件十几步的活，会有一个很实在的体感：越干越慢，而且越干越健忘。这不是错觉，结构就是这么定的。

现在这类工具的干活方式是一问一答。模型说「读这个文件」，系统把整份文件内容塞回它眼前；它看完说「改第30行」，改完的结果又塞回去；它再说「跑一下测试」，测试输出接着塞。十步就是十个来回，而且每一轮的中间结果都留在它眼前，一路堆着不走。

仓库里对这条老路的描述比我写得准，原话译过来是这样：

> 每一个中间的工具结果都会在下一次请求时重新进入模型上下文……对多步工具工作而言，这既费token又是串行的。模型没法组合工具，除非每次调用都走一整个模型往返，而每一次往返都把整个中间结果拖回上下文，不管模型需不需要。

两笔代价被这段话钉死了。一笔是慢：十次工具调用等于十次模型推理。另一笔是挤：模型的注意力被一堆它其实已经不需要看的原文占着。

还有第三笔，藏在开口之前。日常类比：标准模式是背着全套家伙上门的维修师傅，螺丝刀、万用表、梯子都在，搞不定还能打电话叫同事来。

类比的失真点：真师傅不会因为「带了梯子」，就每次开口说话前先把梯子说明书念一遍。而这里，**25件工具的说明书每一轮对话都要重新发一遍给模型**。这笔固定开销每轮都付，而它正是下一个模式想动的地方。

PTC改的是什么

先把名字这件事说清楚，因为它比看上去可疑。

这个模式在英文界面里叫Code mode，代码目录就叫 `code` 。「PTC」这三个字母**只存在于中文界面**。我把整个仓库翻了一遍，二十多万行代码、两千多份文档，没有任何一处展开过它是什么的缩写。行业里确实有一个同名缩写，含义也对得上，但仓库里没有任何证据表明这个中文名借用了它。所以我在书里就这么写：**它是中文界面上的一个名字，来历不明。**

顺带一提，「创造模式」在英文里叫Creator mode，目录名却是`cordis`，那是它挂的插件包的名字。中文界面这一栏，跟代码里的叫法基本是各叫各的。

PTC模式里，模型眼前只剩一件工具，就是那个「跑一段代码」的入口（`run_code`）。它不再一件件点，而是写一段TypeScript程序，想调几次工具就在程序里调几次，想循环就循环，跑完只把自己挑出来的那点结论说出来。

这条规矩是明写给模型看的，系统提示词里那句话是全篇的题眼：

> 用`return`和/或`console.log(...)`产出结果。**只有你打印或返回的东西会回到你这里**，中间工具结果永远不进入对话，所以只提取你需要的。

那另外24件工具去哪了？没消失，只是换了个地方待着。**把一次请求想成一个寄出去的信封：那25件工具原来是一张报关单，夹在信封外面，收件方先看报关单再拆信；PTC模式把这张报关单撕下来，一字不落地抄成一份说明，跟信纸一起装进信封里。**东西一件没少，字数一个没省，变的只是位置。这句话听着像细节，这一章往后所有的账全压在它上面。执行层面卡得很死，点名任何其他工具的调用都会被当场判为未知工具。

谁会用：做那种「一口气有十几步、中间产物又大」的活的人。比如把一个目录里所有文件读一遍、算点东西、只要一个结论。

不用它会怎样：你什么能力都不会少。它是可选项而不是默认形态，这一点是他们主动选的，理由在本章末尾那个栏目里说。

日常类比：标准模式是你对着助理一句句下指令。「打开那个文件」，他打开，念给你听；「找到第30行」，他找到，念给你听。每一句你都要听完整段回话。PTC模式是你写一张纸条递过去：「把这三个文件都打开，找出所有含TODO的行，只把行号告诉我。」他照做，只回你一串行号。中间那三个文件的全文，你根本没看见。

类比的失真点有两处。一，递纸条本身也要成本。前面那张报关单还是每轮都得随信带一份，实测下来固定开销不但没省，还略微涨了一点，真正省的是回话的长度。二，纸条递出去就收不回来了，程序跑起来之后模型没法中途改主意，只能等它跑完。这两笔账怎么算，下面这一场真实会话全摊开。

我跑了两遍，账跟直觉反着来

我在一个只有三个小文件的临时目录里，给它一句大白话任务：把工作区里所有文件找出来，逐个统计行数和字符数，找出最大的那个，最后写成一份报告。模型选的是V4-Pro、推理档拉到最高。

然后同一句任务、同一批文件，用标准模式再跑一遍。为了让两边真的可比，我把标准模式那侧的默认模型也顶成了V4-Pro，推理档一样拉满。两份日志拆开对着看。

先看模型开口之前就已经背上的那笔账。

模型开口前就背上的固定账	标准模式	PTC模式
模型看得见的工具数	25件	1件
工具参数表的体积	26,894字符	897字符
系统提示词	4,101字符	35,643字符
第一轮真实输入token	13,842	15,040

看第二行和第三行。工具参数表确实从两万六缩到了九百，省得很漂亮；但系统提示词从四千涨到了三万五。**那25件工具的类型定义一个字都没有消失，这就是前面那张报关单：从信封外面挪进了信封里面。**换算到第一轮真正发出去的输入，PTC不但没省，还多花了大约9%。

所以正确的说法是：**PTC不省在前面。它要省，只能省在后面。**

顺带解掉一个你现在肯定在担心的问题：三万五的系统提示词，每轮都发一遍，那不烧钱吗？这一场的账是14万缓存命中对1.7万新输入，也就是说这一大段从第二轮起就走缓存价了。按官方那张定价表（快照8月13日，注意8月17日0点北京时间起改成分时计价），V4-Pro缓存命中的输入价是每百万0.003625美元，未命中是0.435美元，差一百二十倍。**前缀稳定这四个字在这里不是理论，是账单。**

还有一笔开口前的固定成本两边一模一样，正好能当尺子用。每次请求都会额外注入一份我这台机器上的skill目录，两场会话里它都是16,542个字符，一个字不差；任务原话只有57个字符。290倍。这笔钱跟选哪个模式没关系，它是我自己攒的东西被读进去了，§06专门算过它值多少钱。这个16,542是从日志里一个字一个字数出来的，不是拿减法估的。

省的是中间结果，不是那张报关单

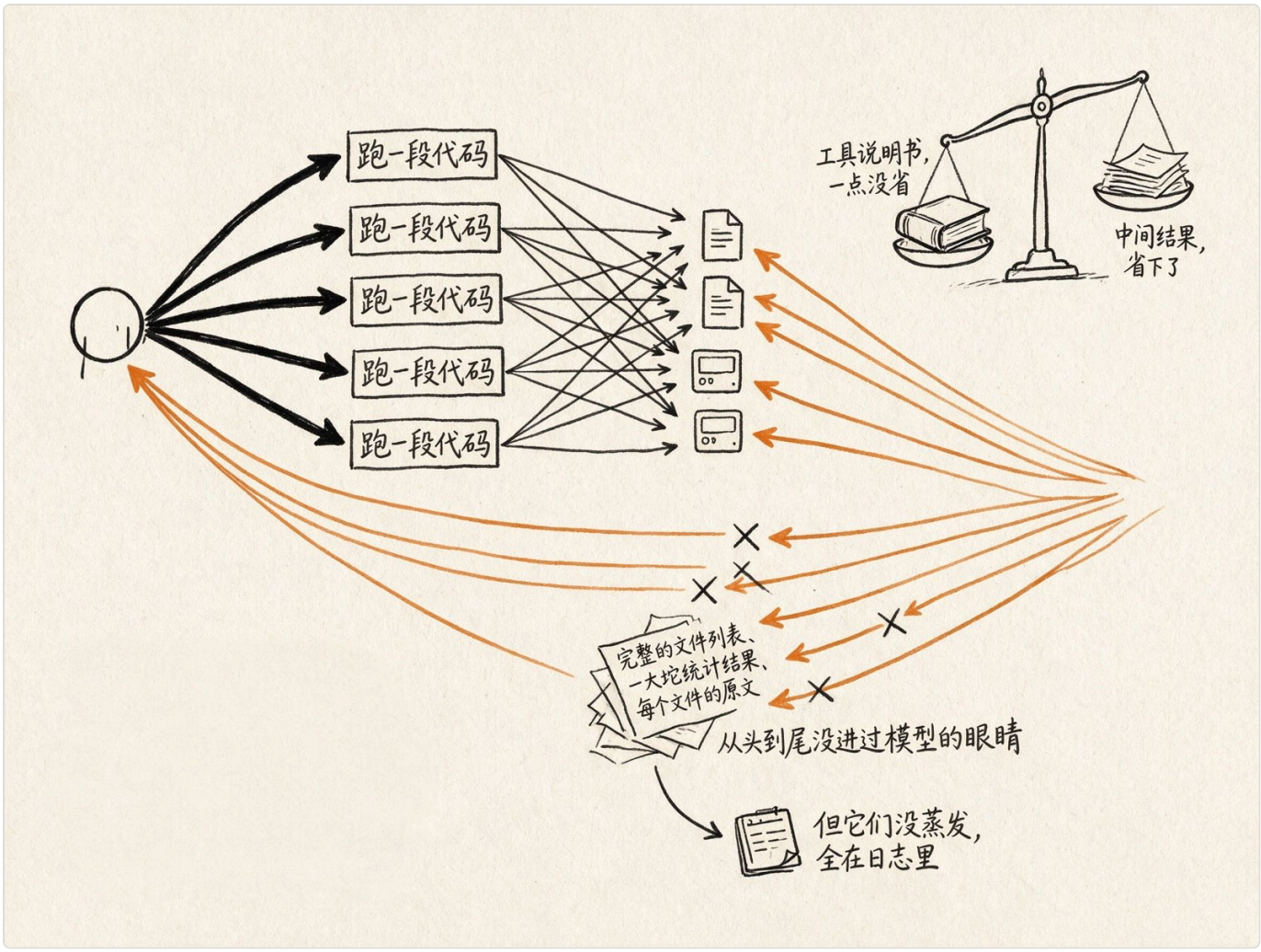
接着看日志的后半截。这场会话里有两类事件泾渭分明：

模型真正开口调工具：5次，全是「跑一段代码」 → **这5段程序在里面实际发起的操作：15次**

15次操作拆开是：找文件3次、跑命令3次、写文件5次、读文件4次。而回到模型面前的，只有那5段程序各自挑出来的结论。中间那10次的输出，完整的文件列表、Python脚本打印的一大坨统计、每个文件的原文，从头到尾没有进过模型的眼睛。

所以PTC省的那部分在这儿：**中间结果的体积。**那张报关单本身的体积，它一点没省。

还有一件跟§05那份会话日志接得上的事：那10次操作不是蒸发了。它们以「代码内派发」这一类事件老老实实落在会话日志里，模型读不到，但你能倒带看。模型看不见，不等于没人看见。



> 看右上角那个天平：省下来的是中间结果那一叠，工具说明书一点没省。另外这是单次实验，一台机器、一件只有三个小文件的活，不能据此下「PTC更省」或「更贵」的普适结论。

它写出来的程序长什么样

第一段程序，它在探路：

```
const ws = ".../scratchpad/dsh-lab";
const g = await tools.glob({ pattern: "**/*", path: ws });
console.log("root:", g.root);
console.log("count:", g.paths.length);
const py = await tools.bash({
  command: "python3 --version 2>&1; echo ---; wc --version 2>&1 | head -1",
  description: "Check python3 and wc availability"
});
console.log("tools:", py.stdout.text);
return { root: g.root, count: g.paths.length, sample: g.paths.slice(0, 20) };
```

不写代码也能看出两件事。第一，工具在这里是一组普通的函数：找文件、跑命令，写法跟调用任何一个现成的库没区别。第二，也是更要紧的一条，它拿到的返回值是有结构的东西，文件路径列表、命令的标准输出，

都能直接拿去算，而不是一坨等着被猜格式的文字。模型敢直接写「这个结果的某某字段」，是因为抄进信封里的那份说明已经把形状告诉它了。

第二段程序更能说明问题。它用JavaScript做编排，中途写了一段Python落盘去做计算，再把结果接回来：

```
// Write file list as JSON (avoids shell quoting issues)
await tools.write({ file_path: "/tmp/dsh_files.json", content: JSON.stringify(absPaths) });

const py = `import sys, json
    ...
print(json.dumps(out, ensure_ascii=False))
`;
await tools.write({ file_path: "/tmp/dsh_count.py", content: py });

const run = await tools.bash({ command: "python3 /tmp/dsh_count.py", ... });
const stats = JSON.parse(run.stdout.text);
```

注释是它自己写的：用JSON传文件列表，是为了绕开命令行的引号转义。它知道那儿是个坑。拿到Python打印的结果之后，它在JavaScript里排序、求和、拼成Markdown表格，写进报告，最后还顺手删掉了两个临时文件。

到这儿它写的已经是个小程序了。

它自己回头对了两次答案

五次程序的意图链条是这样的：

第几次	它在干什么
1	探路：列文件，确认统计工具能用
2	统计，写出第一版报告
3	回读校验：把三个文件和自己刚写的报告都读一遍对答案
4	发现「最大的文件」有并列，重写一版，补上口径说明
5	再修一次：把报告自己排除出统计范围，写最终版

第3步是自发的，我的任务里没有一个字让它校验。第5步补的那个漏洞更细：**它写出来的那份报告，本身也躺在工作区里，会被下一轮统计扫进去。**这是那种人也经常想不到的自指坑。

代价是慢。光第一轮它就思考了9,056个token，整场输出里76%是在想而不是在写。这是最高推理档的表现，不免费。

同一条路，三家走法不一样

让模型写程序去编排工具，不是DeepSeek首创。Cloudflare最早提出这个思路，写的是TypeScript、跑在它自家的轻量隔离环境里；Anthropic的Programmatic Tool Calling让模型写Python、跑在托管的容器里。DeepSeek在立项笔记里直接点名引用了Cloudflare，是明确的后来者。（前面说过，中文界面上那三个字母仓库里一处都没展开过，别把它和别家的名字当成同一件东西。）

有意思的是它在三个地方走了不同的路。

第一，它不靠省掉参数表来限制模型，它在执行环节堵死。这是被一次真实事故逼出来的。8月7日那篇修复笔记写着：把模式改成PTC，只收敛了「广播面」，没收敛「执行面」，发给模型的确实只有一件工具，但执行器那边照单全收；模型只要直接喊出一个原生工具名，调用照样穿过管线执行了，尽管它的参数表从来没发出去过。那句结论值得抄下来：**当直接调用方能绕过时，省略参数表不构成任何强制。**修法是在执行器里堵，而且拒绝消息还带着回家的路，明确告诉模型「只有跑代码这一件能直接调，其他的请在程序里调」，因为一句光秃秃的「未知工具」会让模型以为是部署坏了，而不是纠正自己。作为对照，Anthropic在自家文档里把同一件事写成了警告：别把这个字段当安全边界，你的客户端仍然要准备好处理模型的直呼。同一个坑，一家写进文档，一家写进执行器。

第二，它主动拒绝了让程序之间共享状态。每跑一段程序开一个全新的隔离环境，跑完就死，上一段程序留下的变量下一段拿不到。这在性能上是往后退的：跨调用的状态对会话日志不可见，会打破「每个请求都是日志的纯函数」这条保证。翻译成人话就是，他们宁可慢一点，也要保住「这场会话能一字不差地重建出来」。别家在同一个位置是往前走的，加了状态持久化。

第三，它全程一个省钱数字都不给。包文档里的措辞非常克制：这是拿末端工具的参数表换生成的SDK文本加一个传输入口，「而不是承诺一个普适的减少」。立项笔记更直白，说什么时候该用哪个模式，属于发布之后才能学到的东西。这一点我得说句公道话：几家公布过的降幅数字量级差得非常远，因为它们量的根本不是同一件事，在这种情况下一个数字都不给，是最诚实的姿态。

最后还有一条态度问题。他们明确否掉了「不设模式、所有人都走这一条」的原教旨做法，所以PTC是可选模式而不是默认形态。为什么否，本章末尾那条被否方案说得最清楚。

两边都答对了，然后账单差了一个数量级

回到那两场会话。标准模式那场，21.9秒跑完；PTC这场，305.9秒。

答案两边都对。都统计出了行数和字符数，都写出了报告，都发现「最大的文件」有两个并列。剩下的是完整的账：

跑完这一件活的总代价	标准模式	PTC模式
步数	4	6
模型开口调工具	7次	5次
程序内部派发的调用	0	15次
新输入token合计	14,447	17,591
输出token合计	1,416	17,235
其中思考	739	13,095
端到端耗时	21.9秒	305.9秒

慢十几倍，输出token十二倍，连新输入都多了22%。它唯一少的是模型开口的次数，5次对7次，而那恰好是这张表里最不值钱的一项。**在这件活上，PTC更贵，而且是全面更贵。**

原因是前面那条机制自己写好的。PTC省的是中间结果的体积，而这件活的中间结果一共三个小文件，压根没什么可藏的。入场费照付，首轮就多掏1,198个token，什么也没换回来。

多出来的部分几乎全在输出侧。模型得先把整套流程在脑子里过一遍再写成程序，13,095个思考token花在这儿，中途还自己回读校验、前后重写了两版报告。

所以「PTC更省」这个说法，在我这台机器上不成立。

PTC是一个赌注。你赌的是这件活的中间会产生一大堆你不需要模型看见的东西。任务越碎、中间产物越大，赌赢的面越大。小任务上它稳输，因为根本没有东西可藏，你为那张用不上的报关单付了全款。

限定得跟数字一起交出来：这是单次实验，没有重复测量，模型本身有随机性，换个时间再跑一次差出几十秒完全正常。所以我能说的是「在我这个只有三个小文件的任务上，它慢了十几倍」，说不了「实测证明PTC慢14倍」。差别不在语气客气不客气。后面那句话是在替一个我没做过的实验背书。

决策就一句话：我这件活，是不是步骤多、中间产物大、最后只要一个结论？是，值得切到PTC试一次；不是，老实待在标准模式里。

极简模式：这个是给模型考试用的

这是四个里最容易劝退人的一个，因为它的界面描述几乎全是黑话。但它背后的动机很讲究。

先说它是什么：整个模式只有两件工具。一个是持久的命令行，你可以理解成一直开着的那个黑框框，切进去的目录、设过的变量、装好的环境，下一条命令还在。另一个是只会四招的文件编辑器（看、建、替换、插入）。它俩在代码里就叫这两个名字：

```
bash
str_replace_editor
```

没了。没有搜索，没有子代理，没有上网，没有上下文压缩。

它解决什么问题：它不帮你干活，它是用来**量模型本身有多强**的。

这里得先讲清一件事，不然这个模式看着就是个残废版。业内那些模型排行榜的分数，从来都是模型加脚手架的联合成绩，模型一个人考不出来。你给它的工具越花哨、提示词写得越贴心、上下文管理做得越聪明，分数就越高。同一个模型，换一套脚手架，分数能差出一大截。

所以想知道模型自己到底几斤几两，就得把脚手架扒干净。极简模式就是这套扒干净的配置，它同时也是官方跑基准测试用的那一份，两边是同一个东西的两条投放路径。

他们扒掉的东西里，有四样一般人根本想不到要扒。上下文压缩被拿掉了，理由是只要压缩机制还在，模型的历史就有可能被一段摘要偷偷换掉，那这一次测的就不是模型了。运行时环境说明被拿掉了，工作目录、沙箱状态一概不注入。系统提示词被彻底锁死，任何插件都加不进一个字，靠的是结构上根本加不进去，不靠谁自觉。

第四样是我完全没想到的。他们本来可以把沙箱开到「完全放开」这一档，效果上等于没限制。他们拒绝了。**理由是即使权限全部放开，工具的参数表里仍然会挂着「沙箱权限」和「申请理由」这两个字段，模型仍然看得见有个沙箱在。**他们要去掉的不是限制，是限制在模型眼里的痕迹。

谁会用：要给模型跑分的人、做强化学习的人、想拿它当对照组的人。不用它会怎样：你什么都不会损失，除非你正好在干上面这三件事。

日常类比：标准模式是给考生发了计算器、公式表、草稿纸，外加一个能随时打电话请教的场外顾问。极简模式是全部收走，只剩一支笔一张纸。

顺便记一条实用的：极简模式在Windows上跑不起来。它硬绑了bash，而且是四份配置里唯一没做平台判断的那个。仓库自己把这条记成了待办事项。

创造模式：让它打开自己的后盖

这个模式的配置文件开头写着一句自我介绍：

> It exists so a person can ask an agent to author another agent.
>
> （它存在，是为了让一个人可以叫一个agent去创作另一个agent。）

它是什么：标准模式的全套能力，外加读写「它自己正跑在上面的那套东西」的权限。多出来的是七件自指工具，能干三件事：查看自己此刻的运行时（有哪些服务在跑、有哪些工具挂着）；在内存里现场定义一个插件并挂上去试；以及照着自己的样子，写出一份新的模式配置，给下一个会话用。

我在本机让它现场给自己长了一件新工具，它做到了，过程和账单是上一章§07的正题。这里只需要知道概念：**它能改的是自己的运行时，而不是自己这一辈子。**

谁会用：想把这个东西改造成自己那一套的人。不用它会怎样：你少掉「让它给自己长零件」这件事，同时也少掉一份不小的风险。

风险这一条他们写得毫不含糊。配置文件里那行警告是：**把开在这个模式上的会话，当成直接把命令行交出去来对待。**因为模型写的代码会拿去对着活的运行时求值，而那层隔离按他们自己的说法「隔离全局变量，但不是安全边界」。

日常类比：一个能拧开自己后盖、拆下零件看看、还能给自己焊一块新板子的机器人，焊完还能说「按这张图纸再造一台我的兄弟」。

类比的失真点有两处。一，真机器人拆自己会疼会坏，这里的「拆」只发生在内存里，重启就没了，更像在镜子前试衣服，试完得手动写进文件才算数。二，它焊的那块板子是给下一个会话用的，它自己这一辈子的配置在启动那一刻就冻住了。

说到底，这四个模式是四个文件夹

最后交一个底，能省掉你很多困惑。

这四个模式在代码里不是四个开关，是**四份清单文件**，各躺在一个文件夹里，写着「这个模式由哪些插件组成」。行数分别是251、262、262、62行，最短的那份就是极简模式。切换模式这个动作，实质是换一份清单，重新组装一个agent。

这也顺手解释了为什么只有空白会话才能切：换清单等于换掉模型手里那套工具，而已经跑过的对话记录是在旧工具下产生的，换了就对不上了。

还有一件事值得知道：你可以复制其中一份改成自己的第五个模式，而且这是官方唯一支持的造法。为什么不能从空白开始写，§11有完整答案。

他们本来想这么做，后来没做

> PTC模式的立项笔记里，第一个被否掉的方案是「干脆不设模式，所有人都只走这一种」。这是它参考的那篇Cloudflare博文的原教旨做法。否决理由：编码agent最家常的那几个动作，跑个命令、读个文件、改一行，本来就最适合原生调用，逼着每一次编辑都套一层程序，是在给常见情况上税。所以PTC是可选模式，不是默认形态。这个取舍里藏着他们一个反复出现的倾向：**宁可留下四份高度重复的文件，也不肯给你一个更聪明、但你用不明白的入口。**同一个倾向还解释了另一件事——为什么造新模式的唯一入口是复制一份现成的再改，而不是从空白写起。那件事的完整现场在§11。

界面上，一次PTC运行永远只显示一张卡片。他们考虑过给程序里的每一次操作各建一张卡片，被否掉了，理由是中间值有意是执行局部的、永不面向模型的，摊成很多张卡片会把一条实现轨迹暴露给你，而不是你和模型实际发起的那一个操作。

另一条被否的更值得记：修那个「能绕过去直呼工具」的漏洞时，有个省事的方案是写一个默认插件去拦。否决理由一句话说死了，**一条安全上的不变量，绝不能依赖部署方恰好组装了正确的插件。**

§09 让它自己干完一件长活

Getting It to Finish Something Long

「AI能不能替我盯着一件事」，多数agent给的答案是「你可以在提示词里让它循环」。这一节讲另一种答案：在DSH里，这件事不是提示词技巧，是五个工具、四个状态，和一套故意失忆的循环。以及它自己承认的那个坑。

我真正想问的那句话

我不写代码，小猫补光灯、女娲skill，代码都是AI写的。所以我对agent的期待跟工程师不太一样：不关心它一次能写多少行，只关心能不能把一件事扔给它，然后去干别的。

长活有三种形状。把一个仓库里的同一类问题从头改到尾；每周去看一遍几个竞品更新了什么，写成一页；把一本书的三十章按同一套标准过一遍。三件事只有一个共同点：**它们都比一次对话长。**

我在本机用无界面的方式跑了一次，把模型能看到的工具清单原样打出来，一共25个（下面这些工具名都出自这一份实跑清单）。数完发现一件事：**其中5个跟指挥别的agent有关。**

它直接把「派人干活」做成了工具，而不是靠一段提示词去教。

五个工具，把多agent摆在了一等公民的位置

它能干什么	工具名	关键差别
起一个全新的子agent去干活	`subagent`	全新会话，看不到你俩之前聊了什么
带着上文起一个	`subagent_fork`	用父会话已完成的轮次当种子
看看现在有谁在跑	`list_agents`	后台可续聊的那些都在这儿
给某个在跑的补一句话	`send_message`	不用等它跑完
把某一个叫停	`interrupt_agent`	只停那一个，别的继续

一份25个工具的清单，五分之一的位置给了怎么跟同僚打交道。

真正让我停下来的是子代理向上级汇报那个工具的说明书。它对模型是这么写的：那个agent和你共享工作区，但它不会自动收到你的对话记录、工具输出或者推理过程，所以**「你把活干完了」本身并不是一个结果。**

一句写给模型看的话，放到人身上一样成立。

一处容易踩空：计划模式拦不住任何东西

文档里写得很直白，它只是软性指引，真正的限制来自沙箱和审批策略，两边互不读写对方的状态。而且进了计划模式，工具清单一个字都不变，只是在提示里加一句「现在别用」。原因写在配置注释里：藏起来会改变提示词的开头，把已经暖起来的缓存打掉。用一句规则的成本，换掉一次全量重算的成本。

目标：它对「卡住了」的定义严到不近人情

目标是挂在同一个会话上的一件长期的事。它只有四个相位：进行中、暂停、卡住、完成。另外带一个轮数上限，防止它自己没完没了地转。

「卡住」这个词它是这么定义的。这段话直接写给模型看，我照原文译：

> 只有在同一个阻塞条件连续三轮都还在的时候，才可以标成卡住，并且要在理由里写清楚那个具体的条件是什么；**困难、不确定，或者还剩下有用的活没干完，都不算卡住。**

「我卡住了」恰恰是agent最常用的收场白，而且是最省事的那种。这里干脆把三种最常见的托词逐个点名否掉了。

更狠的是它还有一道机械闸门：一次自动轮次里报「卡住」，在满三轮之前会被直接拒掉。文档自己给这道闸门降了预期，说它只是一个机械下界，不是一个能判断「是不是同一个问题」的评估器。这句自我限定没把一个计数器包装成理解力。

还有一条我认为该写进任何一份安全清单：**重新打开一个会话，它自己永远不会开始干目标里的活。**目标的「激活」这个开关是故意不写进持久记录的，你合上电脑第二天打开，它不会已经自己跑了一晚上。要它继续，你得亲口再说一次。

把一个回合外包给隔壁

子代理这件事上，DSH有一个我在别处没见过的设计。它的六个后端藏在同一个接口后面，差异大到离谱：从「在本进程里起一个全新的孩子」，一路到「另一个产品里的一个被委派的回合」。

后面这句不是修辞。那两个后端一个通向Codex，一个通向Claude Code，用的是你本机已经装好、已经登录好的那个真产品进程。它只借一个回合，跑完把整棵进程树杀干净，对方的会话id从不写进你这边的记录。它也明确不碰对方的登录、模型选择和设置：那样会在人家自己的配置旁边再造出第二个权威。

出厂的标准模式配置里，这两行是关着的，旁边留着一句注释，大意是「复制一份这个配置，把`disabled`去掉，就能用了」。

照着做之前先查了一遍，结果是这样：

> **那句注释已经跟不上代码了。**就在公开当天、仓库对外开放前的几个小时里，团队把这两个后端从生产安装包里拆了出去，理由是不想让每一次普通安装都顺带下载一整套外部SDK。所以现在的实情是：调用它的工具包在，被调用的那个后端不在。只去掉 `disabled` 不够，你还得自己装那个后端包。

有一点得说清楚：**这个功能不是不存在，也不是没发布。**那批不在默认安装里的包一共35个，逐个打到npm上查过，全部在架。差别只是一条安装命令。这里还埋着一个更隐蔽的坑：这些包在npm上的「最新」标签停在几天前的旧版本，当前构建挂在另一个标签下，直接装会装到一个落后三个版本的东西。

这个项目64天里提交了一万两千多次，对外开放前的最后几个小时还在挪发行包的边界，那句注释没来得及跟上。文档落后于代码，在这种速度下基本是必然的。你要做的是知道去哪儿验一下。

让它从头再来一遍

这一节的主角是这样一个东西：给定一个不可更改的目标，反复地把这个目标交给一个完全崭新的工人。每一轮开一个全新的子会话，不带上一轮的任何对话，只靠两样东西传递状态——共享的工作区，和一份有字数上限的交接单。（这套做法在DSH里叫Ralph循环，致敬的是Geoffrey Huntley在2025年7月写下的那个技法；顺带一提，仓库里没有任何地方提到过他的名字。）

每一轮的提示词是写死的。其中最要紧的是这两句：

> 共享工作区和它当前的工作树，是长期记忆和事实来源。动手之前先检查它们，保住已有的工作，做具体的、在范围内的活，并且验证你改动的东西。把上一份报告仅仅当作一份有界交接单，拿工作区去核对它。

为什么故意失忆反而有效？我的理解是这样：一段长对话里，模型的上下文会被三样东西污染：早期的错误尝试、绕过的死胡同，还有它给自己讲的那个「我已经做过什么」的故事。这些东西不会自己消失，而且占比越来越大。

这套循环押的注是：**工作区本身就是记忆。**第四十七轮那个全新的工人，打开工作区看到的是前四十六轮真实留下的文件，不是关于那些文件的叙述。它没有前四十六轮的挫败感，也没有那套自我合理化。

类比是：一个项目做了三个月，团队每个人都被历史绑住了（「这块我们试过，不行」），于是每天早上换一批完全不知道历史的新人，只给他们看代码和一张便签。这里赌的是「隐性知识本来就不该是隐性的，它应该落在工作区里」。所以那句「验证你改动的东西」才是这个技法成不成立的关键。

顺手把三个绕人的词钉一下：你说一句话、它把这件事办完，是一个**回合**；办的过程里每一次「想一下再动手」是一**步**；而这套循环每换一个全新的工人重来一遍，是一**轮**。这个分层有一个很实用的读法：我本机那次简单任务跑出来是1个回合、2个步（那份44行日志里，`step/start` 和 `step/end` 各两条），回合和步的比例告诉你它话痨到什么程度，看到一比三十，那不是在干活，是在原地打转。

交接单只有三种结论，而且约束很紧：还要继续，必须至少给出一条下一步；宣布完成，必须拿得出具体证据，且不能留下一步；宣布卡住，必须写出一个具体的阻塞点。报告超长了不会被悄悄截断，是直接失败。出厂的标准模式给它设的轮数上限是64，注意包本身的默认值是256，别把包的默认值当成产品的实际值。

然后是必须带上的那条局限。它自己在文档里写着：

> **完成与否是工人的自我宣称，不是独立评估。**独立评审者、评审驱动的续跑、完成证书、对抗性验证器，全部有意推迟。界面上关于完成和阻塞的文案也被要求明确写成「某个工人报告了这个结果」，而不是「已认证完成」。

所以整套机制里没有任何东西能保证第三十轮说的「我完成了」是真的。它只保证了一个形式约束：说完成就必须给出证据，且不许再留下一步。

干完了，不等于交付了。这套东西的位置是一个能跑很久的起草工人，不是一个能自己签收的验收员。**这个循环我一次都没实跑过，本节全部来自源码、文档和配置，先标在这儿。**

他们本来想这么做，后来没做

> 2026-07-12有人提议：把工作流能力砍成「真正被用到的前台核心」。理由很硬。它带着一整套没人订阅的进度观测系统，六个事件、一套协议消息、一本配对账本，生产环境里零个监听者，只有测试在用。提案说全删。
>
> 否决意见是一句话：这套进度观测是刻意设计的对外观察接口，该做的是给它找一个消费者，让它变得有用，而不是删掉它。
>
> 这是十一篇被否方案里最微妙的一次：诊断被完全承认（现有的事件确实缺少归属信息，一个全局监听者根本没法把它路由到正确的会话），处方被否决。结论不是「保持原样」，是「重新设计相关性契约」。你今天在界面上看不到任何工作流进度条，原因就在这儿。

模式是一份清单，清单上的每一行是一个插件。那skill呢，MCP呢，它们是清单上的行，还是行里装的东西？

§10 skill、MCP、插件，到底谁管谁

Plugins, Skills, MCP: Who Owns What

从Claude Code搬过来的第一天，你会撞上一个错位：skill原封不动就能用，MCP你也知道怎么配，可这个产品从仓库简介到官方文档，满屏都在说插件。那插件是第三样你还得学的东西，还是把前两样吃掉的那样东西？这一节只回答一件事：谁管谁。

搬完家之后最先冒出来的问题

顺序是这样的：先试skill，本机那57个skill一个不少地被列了出来，零配置（§04讲过这件事）；然后翻文档确认MCP能接。到这儿看上去搬家已经结束了。

结果仓库的一句话简介写的是 `DeepSeek Harness: Everything is a Plugin.`，插件这个词在文档里的密度高得反常。疑问很朴素：**既然skill和MCP都有了，那插件是拿来干嘛的？是不是还得再学一样东西？**

问题本身就问反了：dsh没有在skill和MCP之外再加一层扩展方式，它把skill和MCP本身，做成了普通插件。

一份配置文件里，skill和主循环并排站着

这话不用听我说。dsh有个命令能把「我现在到底由哪些零件拼成」整份打印出来。在网页版跑过一次，拿到的那份清单里，跟skill有关的是这四行：

```
- id: skill
  name: '@deepseek-ai/dsh-skill'
- id: skill-filesystem
  name: '@deepseek-ai/dsh-skill-filesystem'
- id: skill-badge
  name: '@deepseek-ai/dsh-skill-badge'
  disabled: true
- id: tool-skill
  name: '@deepseek-ai/dsh-tool-skill'
```

同一份文件往下翻两百行，是这两行：

```
- id: agent-loop
  name: '@deepseek-ai/dsh-agent-loop'
- id: llm-deepseek
  name: '@deepseek-ai/dsh-llm-deepseek'
```

上面那段是整个skill系统，下面那段是agent的主循环和DeepSeek的模型适配器，也就是它怎么一轮一轮地干活、它去问哪个模型。在这份清单里，**它们跟skill是同一种物件，同一种删法。**

想把skill整个关掉，就是加一行 `disabled: true`，跟关掉任何一个别的零件的手势完全一样。上面那段里已经有个现成的例子，`skill-badge` 那行就是关着的，一会儿说它。

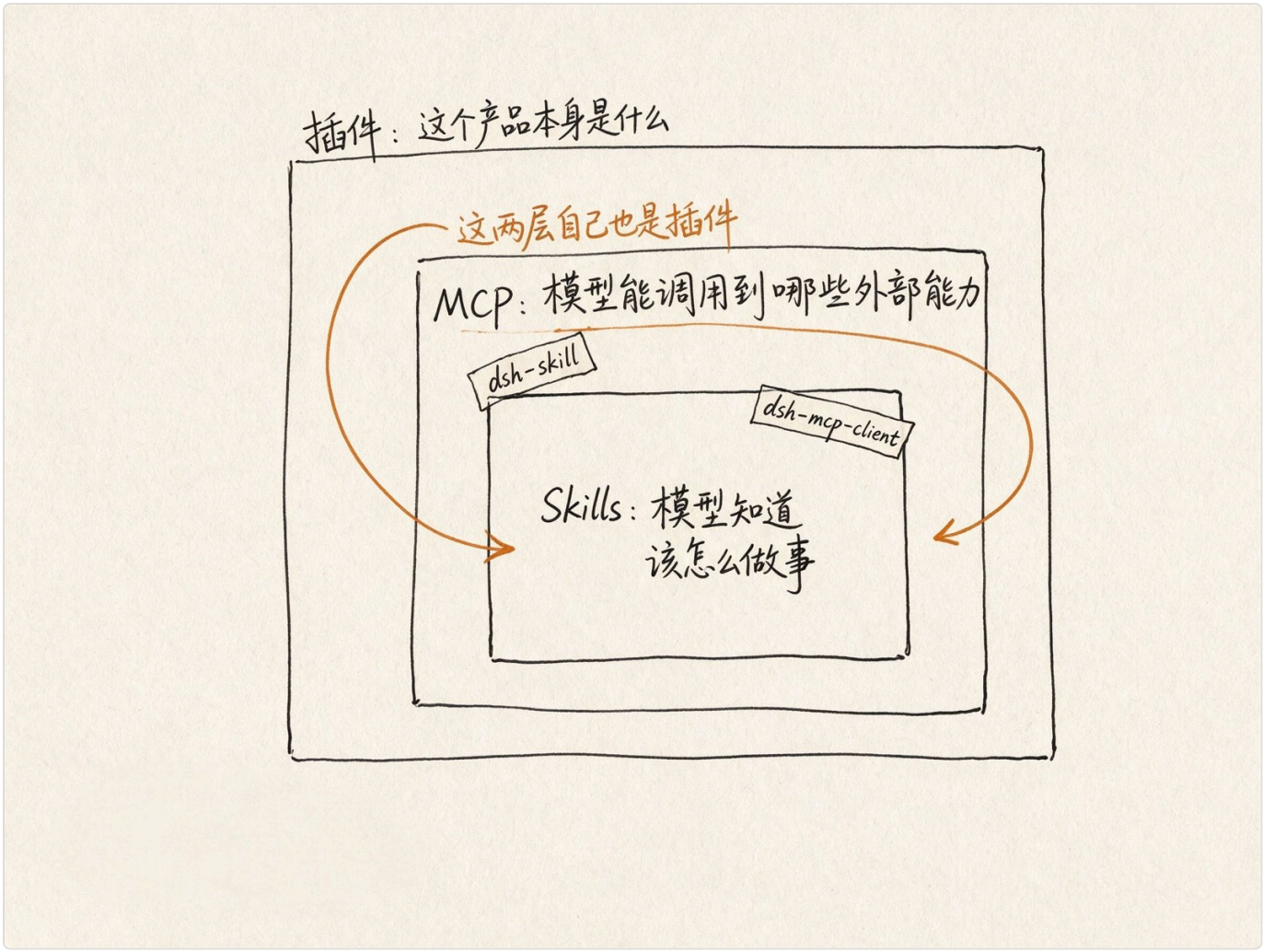
MCP那边同理。它在仓库里只有一个包，接一台MCP服务器就是往清单里加一行：

```
- id: mcp-github
  name: '@deepseek-ai/dsh-mcp-client'
  config:
    serverName: github
    transport: stdio
    command: npx
    args: ['-y', '@modelcontextprotocol/server-github']
```

一台服务器一行，两台就两行，同一个包被装载两次、配置不同。这顺带回答了一个很具体的搬家问题：Claude Code那种把所有服务器塞进一个大对象里的配置，不能整块端过来，得拆成一行一行地写。

还得交代默认状态：我dump出来的那份配置里，MCP一行都没有。包随安装带上了，但默认一台服务器也不挂。官方给的理由很直接，说每一条服务器启动命令都是「agent沙箱之外的可信可执行代码」。翻译过来就是：这行命令是拿你本人的权限跑的，不受agent那套限制管，所以我们不替你做主。

三层各自在改什么



> 那根绕回来的橙箭头是这张图的题眼：里面两层自己就是两个普通插件，包名都在图上。这不是我推的，是启动清单里并排躺着的几行。

把这三样东西并排看，分工其实非常干净。区别在于它们各自能改动的是哪一层，跟谁更强没关系。

	它改变什么	你交付的是什么	谁写得动
Skills	模型知道该怎么做事	一段Markdown	任何会打字的人
MCP	模型能调用到哪些外部能力	一条命令或一个网址	会照着抄配置的人
插件	这个产品本身是什么	一个npm包	写TypeScript的开发者

拿餐厅来比还算顺手。Skills是菜谱本，写清楚宫保鸡丁怎么做，厨师翻开照做，厨房一点没变。MCP是跟外部供应商的对接协议，让这家店能点到隔壁的食材和设备，厨房也没变。插件是厨房本身的模块化，灶台、抽油烟机、冷库、收银台，每一样都能整个拆下来换掉，连「厨师的工作流程」都能换。

那为什么不能只留一样

skill替代不了插件。skill就是文本，它只能影响模型的判断。你没法用一篇Markdown把底下的模型换掉、把会话记录从一种格式换成另一种、或者给整个产品换一套界面。

MCP也替代不了插件。这个协议的动词只有三个：工具、资源、提示模板（dsh目前只接第一个）。「换掉agent的主循环」这种动作，协议里根本没有对应的概念，连这个词都没有，谈不上实不实现。

插件替代得了那两样，但不划算。理论上完全可以，毕竟skill和MCP本来就是拿插件实现的。可写一个skill是写一篇Markdown，写一个插件是发一个npm包、配好构建、过类型检查。dsh保留skill和MCP，等于承认了一件很朴素的事：**不是每一个扩展都值得写代码。**

题眼：他们做过一个专用格式，然后亲手删干净

dsh曾经有一个专门的插件格式，后缀叫`.dsh-plugin`。配套的东西一应俱全：专属的清单文件、自动生成的包装层、独立的缓存、装载器里的内置分支，**以及仓库专用的Skill适配器和MCP适配器。**也就是说，skill和MCP曾经在这套体系里有自己的专用通道。

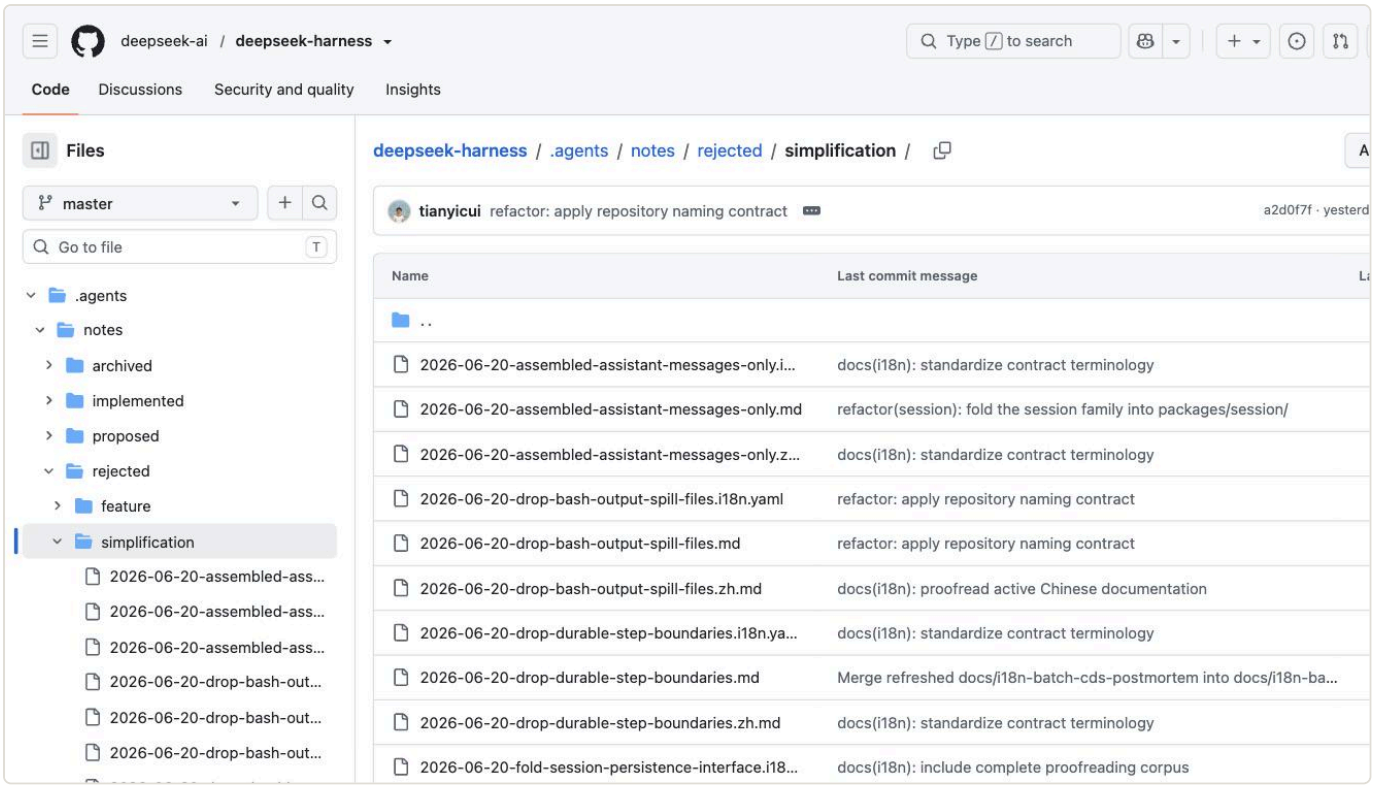
2026年8月9日，整套删掉。

删除笔记里写的替代方案，是我读这个项目一天下来最喜欢的一句：

> 一个提供Skills的包，挂载`@deepseek-ai/dsh-skill-filesystem`；一个提供MCP服务器的包，挂载`@deepseek-ai/dsh-mcp-client`；提供原生能力的包，挂载一个普通的、编译好的Cordis插件。
>
> （原文见删除笔记`2026-08-09-remove-repository-plugin.md`，中译为我所译。）

翻成人话：**skill和MCP不需要特殊通道，它们就是普通插件，用普通的包管理器装就行。**他们判断这条专用路径跟已有的普通路径「在安装和组合第三方包这件事上重复了」，而重复的代价是两条安装命令、两种清单格式、两套出错和缓存的身份，服务的却是同一个包。

顺带一提，删的时候他们连「留个解析器做迁移」都拒绝了，理由是产品还没发正式版、没有兼容义务，留一个解析器等于让一份已经删掉的约定继续活着。旧的`.dsh-plugin`包直接停止工作，不迁移，不自动清理。



> 被否掉的简化提案单独存一个目录。文件名多是 drop、fold 打头——他们否掉的大多不是新功能，是「要不要再删点什么」。

一个最小插件，一共几行

我不写代码，插件这一层大概率一辈子不会自己动手。但门槛在哪儿值得知道，官方教程里最小的那个例子是这样：

```
import type { Context } from '@deepseek-ai/cordis'

export const name = 'hello-plugin'

export function apply(ctx: Context) {
  // Required dependencies are ready before apply runs.
  console.log('[hello-plugin] plugin loaded!')
}
```

就这样。文档给插件下的定义只有一句：一个导出`apply`函数的TypeScript模块。框架加载它的时候调用这个函数，递给它一个上下文对象，插件拿着这个对象去注册自己的能力。紧接着还补了一句`That is the complete configuration.`（这就是全部配置）。没有清单文件，没有注册中心，没有一长串生命周期钩子要你实现。

要把它发出去给别人用，也是三个文件：一个`package.json`声明「我贡献了什么」，一个YAML补丁文件写明「我要往那份零件清单里插哪几行」，一个模块本体。装的时候敲一条`dsh plugin`命令。

这条命令的真身有点好笑：它基本上只是pnpm的转发器。官方参考手册直说了，它在配置目录不存在时先建一个，然后把你给的参数原样转发给pnpm，工作目录切到那个配置目录。所以`add`、`remove`、`update`这些动词全都原样可用，从npm装、从git装、从本地目录装、从压缩包装都行。他们没做插件市场，直接把整个npm生态借过来了。

诚实交代一句：**我没有实测过装完之后能不能真的跑起来。**调研阶段跑通的只是随包自带的那部分，第三方插件的完整安装链路这次没验。

我的skill到底能不能直接用

能，而且是有意为之，不是碰巧。

格式上，dsh认的是跟Anthropic那套完全同构的写法。一个文件夹配一个`SKILL.md`，或者干脆一个Markdown文件；开头那段元信息必填名字和描述。

可选字段里有两个开关，一个是不许模型自己调，一个是允许人手动调，写法跟Claude Code一模一样。同构到什么程度？连有人提议把驼峰拼写也认成别名都被否掉了，理由是外部格式就照Claude那套写（§04末尾那条被否方案说的就是它）。

目录上，它扫六个位置，其中排第五的那个是`~/.agents/skills`，也就是本机三个agent共读的那棵树。所以我那57个skill是被白捡走的。

> **一个要当心的落差：**`~/.claude/skills`不在扫描名单里。我这台机器碰巧没事，因为那个位置早就是一个指向公共目录的软链。但如果你的Claude Code skill是实打实存在那个目录里的，dsh看不见，得自己配一条自定义路径，或者做个软链把它接过去。

还有一个细节值得单拎出来：**模型自己用写文件的工具造出一个新skill，同一步就能看见它。**一般的做法是靠文件监听器发现变化，那中间必然有延迟。dsh的第一方写入和编辑工具，在写到可能影响skill的路径时会顺手把目录标记为失效，所以它写完就能列到自己的清单里，不用等。这是「让agent自己长本事」这条路上一个很实的地基。

发了个skill，然后在默认配置里把它关掉

前面那段配置里`skill-badge`那行的`disabled: true`，就是这个意思。他们做了一个官方内置skill，随包发出去，然后在默认配置里关着，你想用得自己去打开那一行。

这个动作比任何声明都能说明他们对「往模型上下文里塞东西」的态度。同一种克制在另一处更明显：模型平时看到的skill目录，只有名字和描述两样，描述还会被截断，默认上限500字符。正文、路径、来源都不给，连「什么时候该用它」那个字段都不进目录。模型得先明确调用一次，才拿得到正文。

再往前追一步，把skill正文直接塞进系统提示词的方案，他们在设计阶段就否了，理由是这样会「摧毁渐进式披露，让每一次请求都为可能用不上的指令付费」。

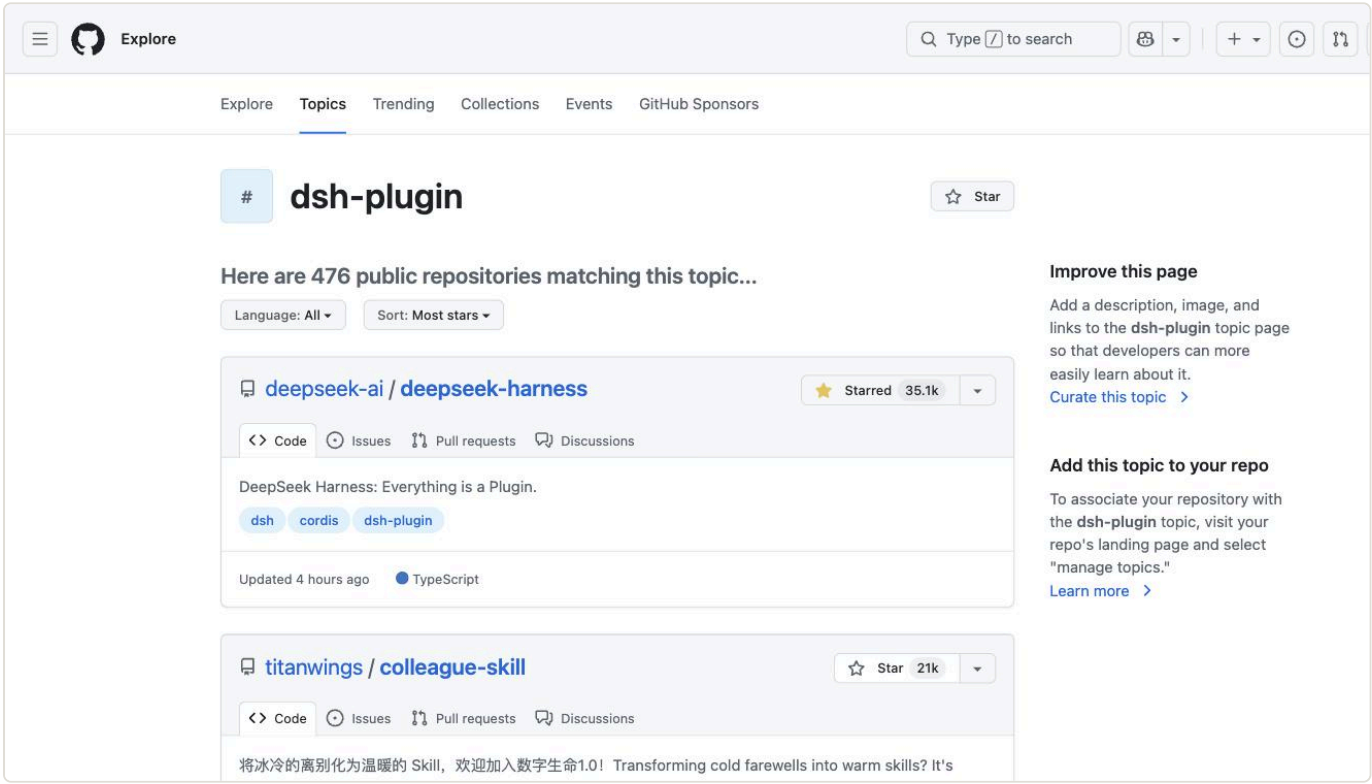
诚实说代价

「一切皆插件」是有反面的，而且大部分是仓库自己承认的。

学习曲线是第一道。写一个skill，会打字就行；接一台MCP服务器，会照抄配置就行。写一个dsh插件，你得先懂那个底层框架的一整套概念：依赖注入、副作用模型、事件链上「往下传」的语义、配置校验的写法。仓库为此备了一份入门读物、四课教程、五篇接口文档、三层九篇开发指南。光是教人写插件就得配一套教材，这本身就是成本。

版本是第二道。README上那句话是全大写的：会有破坏兼容性的变更。项目规则里还有一整节叫「预发布立场：地基优先于爆炸半径」，开头一句是「第一个正式版发布时删掉本节」。`.dsh-plugin` 那整套被删干净，就是这条政策的直接产物。现在写dsh插件的人，得有地板随时被抽走的心理准备。

然后是生态。发现机制只有一个GitHub标签，没有市场、没有搜索、没有安装量。调研时有238个仓库给自己打上了这个标签，8月14日凌晨再看是476个，但打标签不等于是插件，里面混着蹭热度的空仓库和回头补标签的老项目，官方自己举得出的第三方插件例子只有一个。MCP那边的服务器生态早就成了规模，skill那边也早就转起来了。



> 这个话题页就是插件的全部发现机制，没有市场也没有搜索。我们调研时它是 238 个仓库，截图时间 2026 年 8 月 14 日凌晨，476 个。

合起来就是现在这个样子：**dsh的插件层能力最强，门槛也最高，生态却是空的。**而它的skill层和MCP层，能直接白嫖别人已经跑起来的生态。我猜这才是它保留这两层的真实原因，不只是设计上优雅。

他们本来想这么做，后来没做

2026年7月12日，有人提议把skill注册表上的一个接口整个剪掉，理由很充分：**整个仓库里没有一个地方调用它**，还附带拖着一套运行时的假提供方、保留名规则和缓存分支。

提案被否，一句话说明是：直接在运行时注册skill，是刻意留给第三方插件的扩展路径。

没有内部调用者不等于没用，它是留给外人的门。一个把自己整个押在插件上的产品，只能这么判。skill也正是这么被对待的：明明可以做成内置功能，偏要做成一个谁都能替换、谁都能接管的普通零件。

那如果生态真的长起来了，两个插件都想管同一件活，谁赢？

§11 两个插件都想管文件编辑，谁赢

When Two Plugins Both Want the Job

这一章从一次冲突讲起，不从概念讲起。你会看到它怎么回答「两个插件抢同一件活」，看清这套地基是从哪儿搬来的、又付了什么代价，然后反过来拔一个零件下来，看它塌不塌。读完你不会变成插件作者，但你会知道装坏了该往哪儿看。

装第二个插件的时候，该担心什么

我自己维护着一批开源skill，所以看到「一切皆插件」这四个字，第一反应不是兴奋，是想问一句很具体的话：

装了两个都想接管文件编辑的插件，谁赢？有没有优先级？出了事我怎么知道是哪个插件干的？

这个问题不玄。任何用过插件生态的人都被它咬过一次：两个东西都说自己管某件事，系统不吭声，随便挑了一个，你花半小时才反应过来行为是被谁改的。

答案我先给出来：**这套系统里没有谁赢，因为它不允许两个同时在。**同一个位置上只站一个，第二个来的当场抛错，而且错误信息里会点名先到的是哪一个。

一件活，被拆成三个角色

拿最典型的那件活举例：跑shell命令。就是那个黑框框，你敲一行字它执行一行。

你可能以为这是一个功能、一个模块。在这个仓库里，它是四个包：

包	它是什么角色
`shell`	只定义契约：一个执行器长什么样、接受什么、返回什么。不干活
`bash-local`	一个实现：直接在你本机跑
`bash-sandbox`	另一个实现：先套上沙箱再跑
`tool-bash`	模型那一侧看到的工具，也就是模型知道「我可以跑命令」的那个入口

这个家族的说明文件里有一句话，直接回答了本章标题（`packages/shell/README.md`）：

> 「A leaf `cordis.yml` selects one executor implementation and the model-facing tools it needs.」
>
> 「一份末梢配置文件挑一个执行器实现，外加它需要的那些面向模型的工具。」

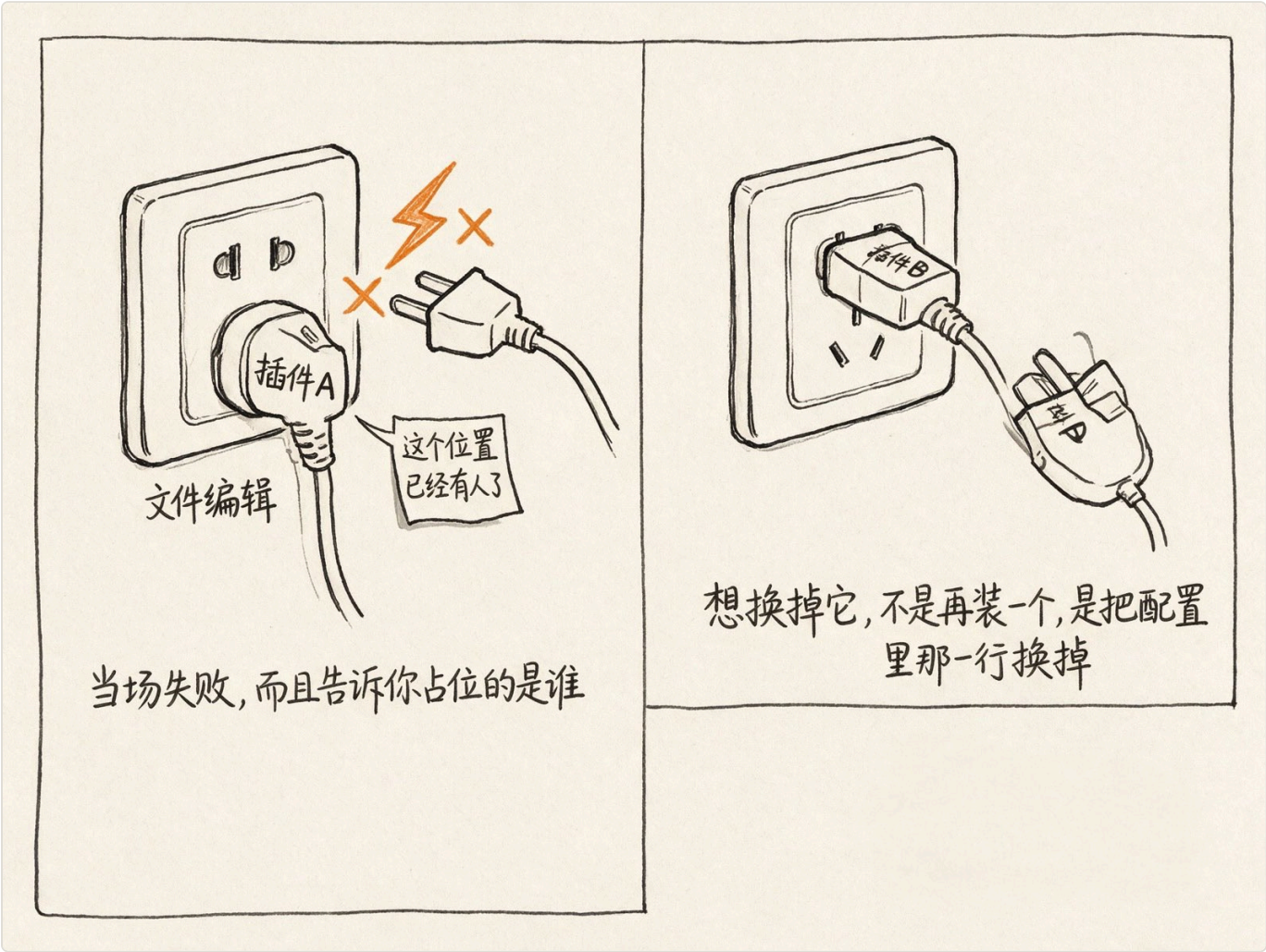
注意那个「一个」。**选哪个实现是组装时的事，不是运行时抢的事。**本机跑还是沙箱里跑，是你在配置里挑的一行，不是两个插件在打架。

为什么非要拆成三份？他们在2026年6月13日的设计笔记里写了理由：这三者的变化速度根本不一样。定义几乎不动，实现经常换，模型看到的东西一变就得改提示词。捆在一起，换个执行器就会搅动模型看到的工具描述，尽管契约根本没变。

拆开的回报写在架构文档里，很具体：文件系统和子进程这两个实现共享同一个执行世界，所以把它们指向一个远程沙箱，Bash、终端、代码跳转会跟着一起搬过去，不需要给任何一个分叉出新版本。**改配置里的一行，本地agent就变成了远程agent。**

类比一下：这更像插座标准，不是发电机。规范本身很薄，接在上面的电器可以无穷多，换发电方式插头形状也不变。

所以谁赢：先到的赢，后到的当场报错



> 右边那格才是这套系统的扩展方式：想换掉文件编辑，不是再装一个插件，是把配置里那一行换成别的实现。叠加在这里不成立。

回到那个冲突。假设两个插件都想认领同一个位置，比如都说「文件编辑归我」。

框架的做法非常粗暴。它在源码里只有一行（`vendor/cordis/src/reflect.ts:290`）：

```
throw new Error(`service "${name}" has been registered at <${this.store[key].fiber.name}>`)
```

翻译成人话：**「这个位置已经被某某占了」，然后当场失败，而且告诉你占位的是谁。**不是静默覆盖，不是随机挑一个，也不是按加载顺序排先后。

顺着这条规矩，产品层还加了一条更严的纪律。写给模型看的那份编辑指南里有一句（发行包里就带着这份文件）：

> 一行「对外提供服务」的配置，不许松散地摆在预设里。没有隔离域的服务会进到进程全局的位置上，于是第二个会话挂同一个预设时就会跟第一个撞车。挂载会直接拒绝它，而不是让这次撞车在后面某个时刻才浮出来。

反方向的规矩同样明确：只消费、不提供服务的行，不许自己包一层隔离域，否则它反而找不到要用的东西。包错了和没包，是同一类错误。

所以想换掉文件编辑，别去装第二个插件，去把配置里那一行换成别的实现。这个系统的扩展方式是替换，叠加在这里不成立。

这套地基不是DeepSeek造的

上面这些规矩，一条都不是DeepSeek发明的。

底下那个框架叫Cordis，作者是一个叫Shigma的中文开发者，2022年5月建的仓库，DeepSeek公开之前它是一个几百星的个人项目。仓库里保留着上游作者的邮箱字段，没抹掉。

DeepSeek的做法是把它整份源码拷进自己仓库。连同几个配套库一共九个包，核心那部分两千六百多行TypeScript，比很多公司的一个业务模块还小。拷进来之后按自己的需要动了刀，改动清单编号到18条，每一条都写清楚改了什么、为什么改。

为什么不老老实实用包管理器装？决策笔记里的理由是这个仓库最早的一批笔记之一，写在项目第一天：上游当时还是候选版本，而agent主循环的正确性直接压在这个框架的内部行为上，上游一个版本号跳动就能打断他们，而且没有本地修复通道。

然后是8月10日那次改名。九个包被改成了`@deepseek-ai/`开头的名字。理由：每个包都把这个框架声明成依赖，所以发布产品就等于连框架一起发布出去。用上游的名字发出去，就是在包仓库上抢注了别人的名字，而在那些代理转发的镜像上，同名条目会盖住真正的上游包，把错误的框架装进毫不相干的项目里。

看到这儿容易觉得这是一次单方面的拿来主义。我去翻了提交记录，发现不是。**Shigma本人在这个仓库里有两次提交**，都在文档上：7月31日加了那篇论文，8月13日加了论文预览版的链接。作者本人是进来过的。

那篇论文叫《一种时空可组合性的编程范式》。「时空可组合性」这六个字翻成人话就是：**拆得干净（时间），拼得对（空间）**。拆得干净说的是卸一个东西要能把它的痕迹全部撤销；拼得对说的是谁依赖谁不用人排顺序。

还有一个巧合值得记一笔：那篇论文的草稿日期是2026年8月13日，跟这个仓库对外公开是同一天。理论和实现同日亮相。

拆得干净，是靠一条不给你选的规矩

「拆得干净」在代码里的落法只有一条：**你注册任何东西的那一刻，必须同时交出怎么撤销它**。监听器、工具、定时器、提示词段落，一律如此。插件被卸载时，框架按倒序把这些全部收回。

像租东西时必须同时填的那张归还单：你租投影仪，登记的那一刻就得写清楚「还的时候关电源、拔线、放回三楼柜子」。

这条规矩带来一个漂亮的推论。教程里是这么写的：

> 「Because unloading releases effects and loading follows dependencies, HMR can replace a running plugin by unloading and loading it.」
>
> 「因为卸载会释放掉这些副作用、加载又会遵循依赖关系，所以热重载只要卸了再装就能替换一个正在运行的插件。」

热重载在这里不是一个被实现出来的功能，是前面两条规则的自然推论。没人专门去写「怎么热更新」，它是白捡的。

对你的实际意义：改自己那份配置文件是热生效的，不用重启；改坏了它会保留上一棵能跑的树继续运行，并广播一条配置更新失败。这是这套架构少有的、外行也能直接吃到的好处。

代价：它失败的时候不出声

拆卸是异步的，而且可以重入。当一堆东西互相依赖、又同时在拆的时候，你会得到一类全新的bug：**不是崩溃，是安静的挂起。**

仓库的改动清单第12条里记了一个现场：一次失败的初次加载触发回滚，回滚去销毁热重载插件，而热重载插件的拆卸正在等一个排在同一批操作后面的刷新。转了一圈，锁死了。原文的措辞是「a deadlock that exited 13 with no diagnostic」，进程以退出码13结束，一个字的诊断信息都没有。

比这更反直觉的是下面这条。它不是bug，是设计：

> 「a plugin whose inject names a service nobody provides waits forever, printing nothing. No error — PENDING is a legitimate state, since the provider may be mounted later.」
>
> 「一个插件如果声明它需要某个没人提供的服务，它会永远等下去，什么都不打印。不报错，因为等待是一个合法状态：提供方随时可能在之后被挂上来。」

插件没加载，不报错。像点了外卖但骑手一直没接单，App不会告诉你出事了，它就静静挂着，因为骑手随时可能出现。

这条设计在框架层面是对的，在产品启动那一刻却是灾难：用户敲了一条命令，某个功能永远等不到它要的东西，界面上什么都不说。

所以DeepSeek在启动这个特定时刻加了两道审计。第一道在配置树挂完之后检查：有启用的条目却没有对应实例，就把每一个解析不了的插件名报出来。第二道再等每一个条目真正活过来：失败的带原始堆栈，还在等的报出它到底缺哪几个服务。

两道审计干的事，是把框架的「等待是合法的」这句话，在启动这一刻强行翻译成「等待就是失败」。同一件事，在框架层是特性，在产品层必须是错误，这个转译动作本身就是产品化的一部分。

另一条给你的判据，来自同一个坑：如果某个功能「好像没生效」，先怀疑它压根没被挂上，而不是怀疑配置写错了。配置写错了它会大声报错，「包不在」才是那种安静的失败。

> 一处我没法替你裁决的矛盾：仓库里两份文档对「插件之间发消息有几种方式」说法不一致。入门篇列了4种，教程和自动生成的接口目录是5种，多出来的那种叫 `bail`。两份都在发行的文档里，我没有找到哪一份是笔误的证据，所以这里如实标出来，不替它选一个当定论。

反过来问一次：拔一个零件下来，会不会塌

前面看的是两个插件抢同一件活的时候谁赢。反过来问一次，问法更难听：这套「什么都能换」的架构，出事的时候值多少钱。一个请求穿过多少层、拆掉一个零件会怎样、以及那份清单里有几行是你真碰得到的。

> **一个插件系统好不好，看的不是加插件顺不顺，是拔一个下来会不会塌。**

抽象层每多一层，好处是换零件方便，代价是出问题时你得多猜一层。而最难受的那类故障不是拆开装不回去，是它既没修好，也没报错。

所以下面不从架构图讲起，从一份清单讲起。

我先把它的启动清单打出来了

它自己提供了一条命令，把出厂那棵插件树原样打印出来：

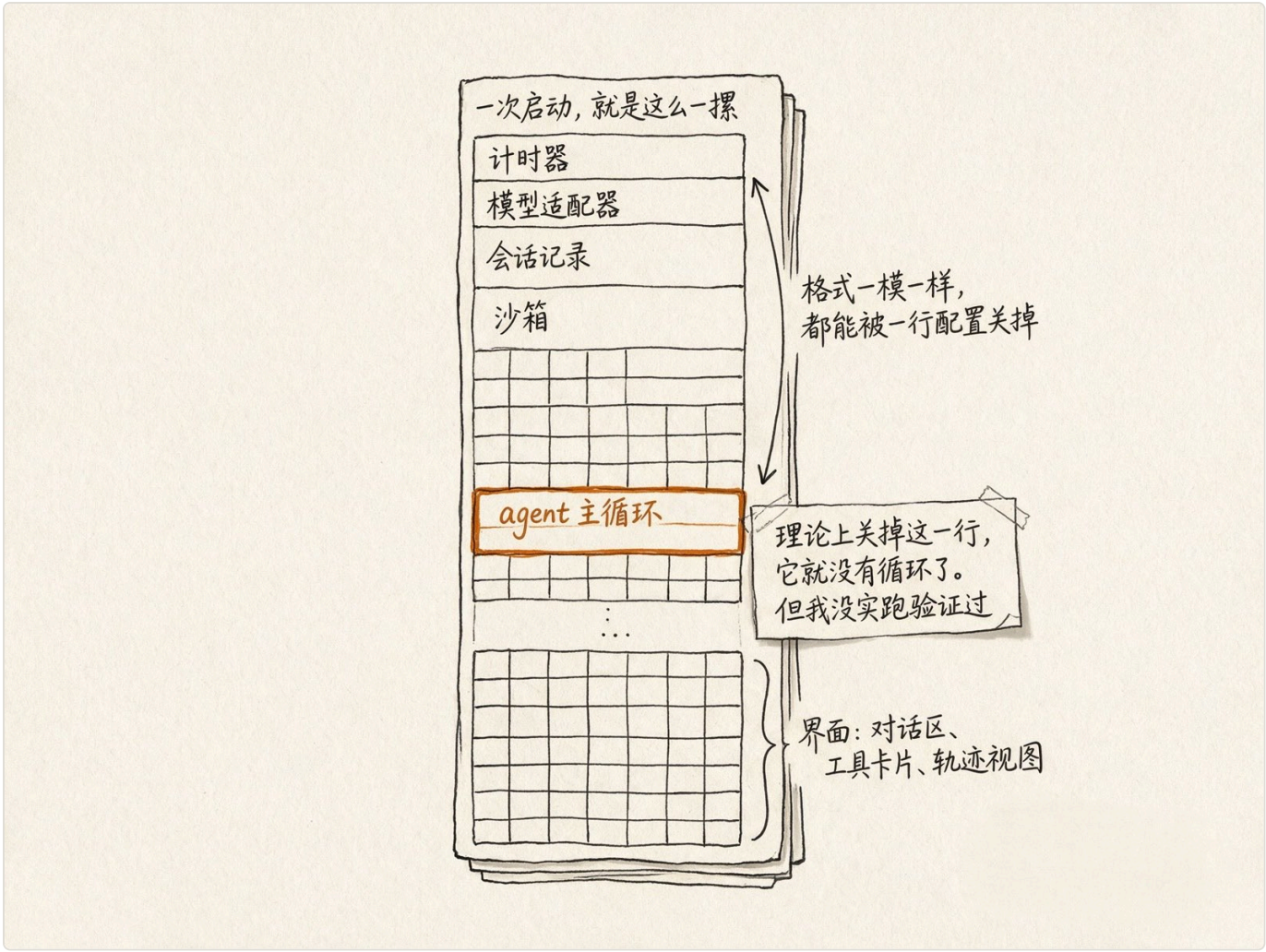
```
dsh --profile web --dump-default-config
```

我在本机跑了一次，输出490行YAML。去掉配置项和注释，剩下的骨架是**129行插件**，一行一个包。这份输出原样留着，开头、中间和结尾各摘一段给你看：

```
- id: timer
  name: '@deepseek-ai/cordis-plugin-timer'
- id: llm
  name: '@deepseek-ai/dsh-llm'
- id: session
  name: '@deepseek-ai/dsh-session'
...
- id: sandbox
  name: '@deepseek-ai/dsh-sandbox-local'
- id: agent-loop
  name: '@deepseek-ai/dsh-agent-loop'
...
- id: ui-conversation
  name: '@deepseek-ai/dsh-client-ui-conversation'
- id: ui-trajectory
  name: '@deepseek-ai/dsh-client-ui-trajectory'
```

第一行是个计时器。往下是模型适配器、会话记录、沙箱。再往下是agent主循环。最后二十几行是网页界面上你看得见的每一块：对话区、工具卡片、那个所有人都想要的轨迹视图。它们全在同一个列表里，格式一模一样。

你的agent不是一个程序，是一份129行的清单。



> 橙框那一行是agent的主循环，格式跟最上面那个计时器一模一样。旁边那张便签要连着读：理论上关掉它这个agent就没有循环了，但这一条我没有实跑验证过。

这份dump还有一处很体贴的设计。每隔几行会插一句注释，标出这一行的出处：

```
# == @deepseek-ai/dsh-base
- id: skill
  name: '@deepseek-ai/dsh-skill'
# == @deepseek-ai/dsh-base, patched by @deepseek-ai/dsh-web-app
- id: tool-fs
  name: '@deepseek-ai/dsh-tool-fs'
  disabled: true
```

「这行来自基础包」「这行来自基础包、但被网页包改过」。129行里有24处这样的出处标记，25行标着已停用。也就是说，你不用去翻源码就知道每一行是谁放进来的、谁动过它。

这129行是怎么叠出来的

它从哪来？三层往上叠，而且起点是空的。

三个词先各给一句人话。profile是一份具名的组合，存在你自己电脑的目录里，`web`和`headless`两份是随包送的模板。bundle是别人打包好的一摞配置行加上对应的代码。patch是补丁层，按行号不认人、只认id。

启动时的动作顺序是这样的：

那个空列表不是比喻。profile目录里那个根配置文件的内容真的就是两个字符`[]`，而且每次启动都会被重新覆写成空的。它存在的唯一理由是加载器需要一个真实的文件当锚点。

补丁怎么改一行？靠id。你想让网页版不挂某个工具，不需要复制整棵树，只要写一行同名id的补丁，它就会盖住原来那一行。上面那个`tool-fs`就是这么被关掉的。

但这里有个反直觉的规则，官方自己写在限制清单里：补丁是整段替换，不是逐字段合并。你想改一个数字，得把这一行的所有字段抄一遍。听着很笨，好处是任何时候你看一眼就知道这行最终生效的完整配置是什么，没有藏在别处的继承链。

顺带说一句，那25行停用的插件里，压缩、清理、命令这几样并不是「网页版没有」。它们是从这棵宿主树搬进了每个模式自己的配置里。同一件事在哪一层做，是这套设计里反复出现的选择题。

「没有特权内核」在代码里是什么样

架构文档里有一段话，是这整套设计的宣言，也是最该被验证的一句：

> 「产品的每一部分都是插件，包括模型适配器、工具注册表、会话日志，以及agent主循环本身，所以每一部分都能从配置里替换掉。**没有一个特权内核等着你去打补丁**：你扩展dsh的方式是在其它插件旁边再挂一个插件，而注册都是可撤销的，插件卸载时自动收回。」

后半句是重点。绝大多数框架给你的扩展方式是「在这里留了个钩子，你把代码填进去」，钩子留在哪、留几个，是内核作者说了算。这里的说法是：不留钩子，你直接在旁边再放一行。

这句话能验证，证据就在上面那份清单里。

那到底会不会塌

正面回答：**在配置层面，agent主循环和第一行那个计时器是完全平级的两行。**

格式一样，出处标记一样，都可以被一行id相同的补丁盖掉。理论上给`agent-loop`那一行加一句停用，这个agent就没有循环了。基础包的补丁文件里，它就老老实实排在`timer`、`llm`、`tools`中间，没有任何特殊待遇。

但话得说死：**我没有实跑验证「关掉主循环会怎样」**。能确认的只有一件事，就是它在配置文件里的格式和别的行完全平级；至于「关掉它就真的没了」还是「关掉之后会优雅地报个错」，都不替你断言。这条要等真跑一次才能写。

不过这个问题问得对，答案可能跟你预期的不一样。因为在这套系统里，**拔错零件最可能的后果不是塌，是安静。**

为什么是安静而不是塌，本章前半已经拆过：底层框架把「依赖的服务还没出现就一直等着，不报错也不打印」定成了合法状态，这是热插拔能成立的前提，而产品只在启动那一刻把它硬翻译成失败。落到你身上就是：你卸错一个包，程序照常启动、界面照常打开、某个功能就是不干活。

所以完整的回答是：会不会塌不是最该担心的，该担心的是它不塌但也不响。

133个在跑，设置页只让你配3个

网页版设置页里有一张插件清单，实跑显示133个插件正在运行，每个带启用停用状态和搜索框。而同一个设置页里，真正能点着改的插件配置只有3张卡：终端、agent循环、网页搜索。

其余130个，你只能看，不能改。要改就得去动配置文件。

这个落差是这东西现在最容易挨骂的一处：通篇讲插件，界面上却找不到插件在哪。但它不是虚伪，是分层：插件的组装是配置文件的事，插件的运行时偏好才是设置页的事。他们的架构笔记里点名说，Codex、Claude Code、Kimi、OpenCode这几家最后都收敛到了同一个切分上。

知道即可，但你得知道，否则你会在设置页里找一个永远不存在的入口。

连界面也长在同一棵树上

你可能已经注意到，那份129行清单的后半段全是`ui-`开头的行，27行，从主题、侧边栏一直到轨迹视图。**前端也是插件树，和后端是同一棵。**

这件事在工程上不容易做到。浏览器里的代码和服务器上的代码是两个世界，怎么让浏览器里的一个插件，安全地喊到服务器上某个方法？

他们的做法叫typert。同构这件事我只讲结论，工具链细节不展开。后端方法上打一个标记，构建时读TypeScript源码，把这些方法编译成一份前端能直接调用的契约文件。没打标记的方法不会出现在前端类型里，也调不到。整个仓库打了标记的方法只有三十来个，其余全部走别的通道。

和常见的做法比，差别落在两点上：契约是真的生成成文件，所以前端插件可以住在独立的npm包里；契约在运行时是活的注册表，所以一个前端插件被卸载，它调后端的那条线会跟着断干净、在途的调用会被中止。

这两点合起来，才让「插件」这个词在浏览器那一侧也成立。否则前端最多是个统一编译进去的整体，谈不上拆装。

他们删掉了网页里的YAML编辑器

先讲一个决定，我认为是这个产品最像样的一次克制。

既然一切都是配置，那在网页上放一个配置编辑器，让你直接改，看起来是顺理成章的一步。他们做了，然后删了。创建一份新模式的唯一方式变成「复制一份现成的，再改」，不提供空白新建。

理由写在笔记里，两句话，都很硬。

第一句：**「从零写YAML这件事人类根本不干。」**一个空白编辑器看着是自由，实际上没人会对着一个空文件从头写出一份能跑的插件组合。给个能跑的样板让人改，才是真实发生的动作。

第二句更狠：**「当编辑器很弱，当能力很宽。」**一个网页里的文本框，没有补全、没有语法高亮、没有差异对比，当编辑器它弱得可怜。可它接受的那段文本里允许写可执行表达式，这意味着你在这个框里粘进去的任何东西，下一次挂载就是在你机器上跑任意代码。

弱和宽这两件事凑在一起，就是最坏的组合：它帮不了你写对，却足够让你写错到很远的地方去。所以他们把整个框拿掉了。

删掉之后还有连锁反应，也一并记了下来：既然手改文件成了唯一入口，常驻的挂载层就得按文件的修改时间和大小分代，否则你改完文件得重启进程才生效。一个删除动作，逼出一个新机制。

他们做了一整个终端界面，然后整包删掉

要证明「一切皆插件」不是宣传语，最贵的一次自证在开发史里。

7月中旬，他们立项做一个全屏终端界面。这条线做得非常认真：光设计笔记就写了40篇，7月21日一天写了14篇，从欢迎横幅的品牌渐变、状态栏要不要显示缓存命中率，到差异卡片的行宽、步骤计时显示在哪个位置，全都单独写过一篇。7月下旬还在持续打磨。

然后8月4日，一篇笔记把整个终端界面包删除。源码、测试、终端快照、依赖声明、文档，一起删。

摆在桌上的不止「删」这一条，三个听起来都更温和的折中方案被逐条否掉了。「留着但不发布」：否，因为一个不受支持的东西留在仓库里，仍然会被人当成可复用的产品面来读。「移到示例目录」：否，原话是「移动代码不提供当前的产品需求、被维护的部署，或组装好的验收」，换个文件夹并不会让它重新有人负责。「留一个兼容包或者别名」：否，依赖它的配置行直接失败，不做翻译。

三条否决指向同一个判据：**一个没有消费者、没有维护者、没有验收的东西，摆在哪儿都是负债，只是负债的伪装程度不同。**这也解释了它为什么能被整包拔掉还不塌——真正被拔掉的东西，是被拔干净的。

终端界面现在在哪？在仓库外面，一个叫 `turtle-ui` 的独立项目里，仓库文档里那条命令写的就是用普通的装插件方式装上去。**但要提醒一句：我按那条命令指向的地址去查，这个仓库目前对外打不开**，所以你现在照着敲大概率装不上。而他们把这件事当成设计的验证：那个外部项目后来自己加了两个命令行参数，启动器一行代码没改。

把这件事的分量摊开说：终端界面是大多数同类编码agent的唯一形态，是Claude Code、Codex这一类产品的整张脸。这个团队先花两周多把它做出来，再一次性删干净，然后用一个仓库外的插件把它装回去。**这是「一切皆插件」最贵的一次自证**，也是他们真的相信这句话的证据。

他们本来想这么做，后来没做

2026年7月19日，有人提了一条精简建议：把上下文压缩的「定义包」和它唯一的那个实现包合并成一个。理由很硬：就一个实现，分成两个包看着像是在还没有第二个实现时就先抽象了。

更妙的是，这份提案引用了「三个角色」那份设计笔记来支持自己，说那份笔记要求的是真实存在的接口、实现和消费者，而不是抢先拆分。

结果被否了，理由只有一句：更多压缩后端已经在路线图上，所以定义包和实现包保持分开。**规则被拿来反对规则自己的既有实施，然后被路线图挡了回去。**这类被否的笔记全仓只留了11篇，因为他们的规矩是：一条被否的记录只在它还能拦住一个诱人的错误时才保留。

这一章问的是设计经不经得起拔。下一章问的是它真出事的时候：仓库里躺着四篇写明「我们是怎么搞砸的」的复盘，其中一篇的开头是178个绿色的测试。

§12 它出过什么事

What Has Gone Wrong

一个公开还不到一天的产品，仓库里躺着四篇写明「我们是怎么搞砸的」的事故复盘，编号0001到0004。翻两遍才确认这不是文档模板，是真事。这一节把四篇逐个拆开，因为它们回答的是同一个我这两年一直答不上来的问题：AI报告「测试全过了」的时候，这句话该信几分。

一句话，把我这两年的不踏实说准了

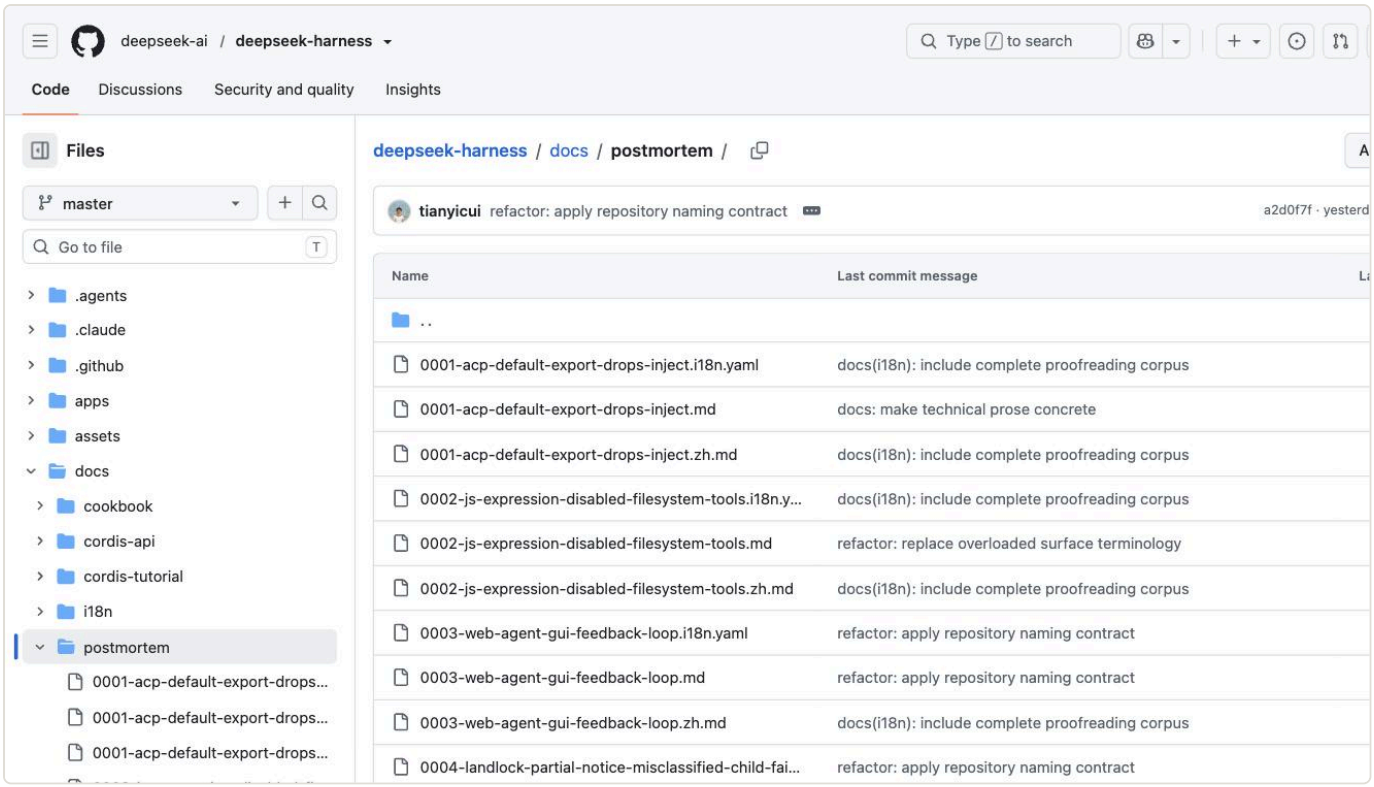
先交代我的位置。我不写代码，所有产品都是AI写出来的。所以这两年最熟悉的时刻是：AI跑完一轮，报告「全部测试通过」，而我没有任何独立办法判断这句话值多少钱。

我一直说不清这份不踏实来自哪儿，直到读到这一句：

> 尽管有178个绿色单元测试和100%行覆盖率，这座桥在生产环境中完全无法工作。

这是DeepSeek自己写的，位置在仓库的`docs/postmortem/`目录里，第一篇。

那个目录一共四篇，中英双语各一份，格式统一：摘要、时间线、根因、新增的防护措施、经验教训。什么样的bug配写一篇，标准也写在目录说明里，三条：机制隐蔽（细心的工程师也得费劲重新推一遍）、系统性（漏出去的原因是测试和约定的缺口，不是一次手滑）、重新发现的代价高（这次烧了真实的调试时间，下次还会）。

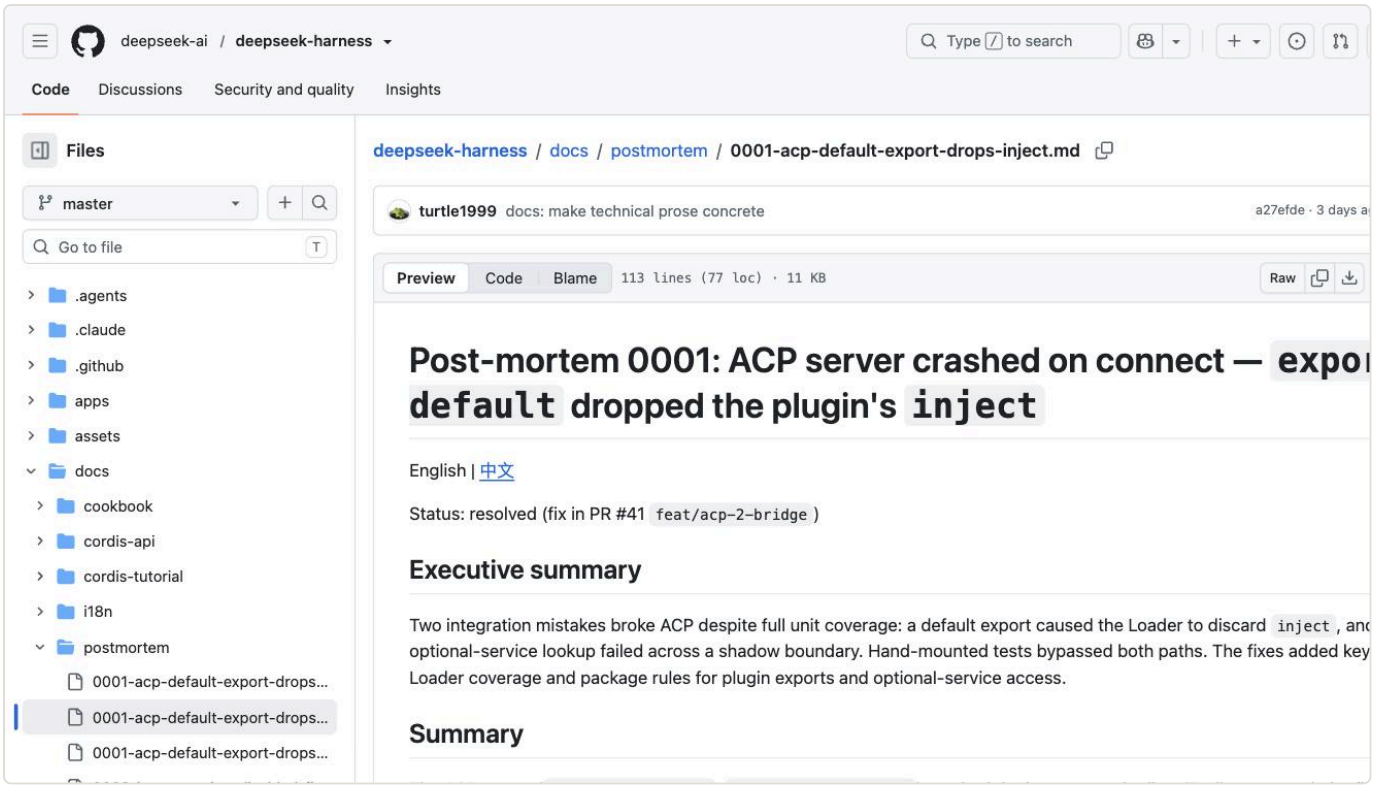


> 四篇复盘，编号 0001 到 0004，每篇三个文件：英文一份、中文一份，外加一份给两边对齐用的。写自己怎么搞砸的，也要走双语归档。

把「我们的流程为什么没拦住」当成正文、把那一行修复当成脚注——这四篇的重点是这个顺序，不是那一行修复。

第一篇：全绿的测试，跑的是三天前的旧代码

出了什么事：他们做了一座桥，让编辑器可以把这个agent当成内置助手挂上去。桥有178个单元测试，行覆盖率100%。真实编辑器（Zed）连上来的一瞬间，第一个请求就崩了，报错是拿不到一个本该注入的服务。



> 标题里就把祸首那一行代码点名了。往下一行 Status: resolved 后面钉着修它的 PR 编号——复盘和修复是绑在一起交的，不是写完就完。

根因是一行多余的代码：`export default apply`。

这行的意思，用大白话讲：一个插件本来把自己的名字、要用哪些服务、配置长什么样，分成好几份并排交出去；这一行等于又额外交了一份「默认件」。而加载器的规矩是**有默认件就只看默认件**，于是并排那几份被整个丢掉，包括「我要用哪些服务」那张申请单。插件在一个什么都没申请到的环境里启动，一伸手就扑了个空。

为什么178个测试全都没抓到？因为它们全都是手工把插件挂上去的，手工挂载这条路根本不经过那个加载器。这就像你每次都用手把零件按在位置上试，试了178次都好用，但出厂时是机器装的，机器认的是另一套规则。

更扎人的在后面。他们其实有一个真的会走完整流程的测试，但那个测试要API密钥才能跑，持续集成没有密钥就跳过了；而它在本地之所以「通过」，只是因为电脑上残留着一份更早编译出来的旧代码，恰好被拿去跑了。

三层保护，三层都在，三层都漏了。

> **他们自己写下的那句结论：**100%行覆盖率始终满足。覆盖率证明代码行被执行过；它不能说明功能是否按交付方式正常工作。

这句话对写代码的人是一记闷棍，对我这种不写代码的人是一把尺子：**下次AI报「覆盖率100%」，该问的不是数字，是「你验的是我交出去的那一份吗」。**

信trace，不信理论

这一篇还留下了一个方法论。

崩溃发生后，调查最先走的是一个非常优雅的解释：框架内部有一套代理机制，服务查找只会往上找祖先、不会横着找兄弟，所以找不到。这个解释成立、机制也真实存在，他们顺着它推理了好几个小时。

然后有人做了件笨事：往框架源码里插了一句打印，跑一遍真实进程，看它到底走到哪一步炸的。几分钟后真相出来了，异常抛在插件加载的那一刻，跟那套优雅机制毫无关系。

最妙的是，那个优雅的解释后来被证明也是对的，**只不过它是第二个bug**。删掉那行多余代码之后，它才浮出水面。

> **核心建议**
> 复盘最后一条经验教训的原话是：相信跟踪结果，不要迷信理论。数小时看似合理但实际错误的推理，输给了一句打印语句。这条对所有跟AI一起干活的人都成立：让它去跑一遍、把过程打出来，永远比听它讲一段听起来很有道理的分析强。

第二篇：一个写错位置的开关，让工具集体消失

第二篇的现场更荒诞一点。

他们想让读文件、写文件、改文件这三个工具「有条件地启用」，于是在配置文件里给它们各写了一个开关，开关的值是一小段现算的表达式。语法完全合法，加载的时候一句警告都没有。

问题是：这套配置只在插件的参数里现算表达式，而「启不启用」这一格是直接照原样读的。照原样读到的是一个表达式对象，任何对象都算「真」，于是这三个工具**在所有模式下永久处于禁用状态**。

七个用到文件系统的测试场景，加一个混合场景，全都在调用一个根本不存在的工具。

然后是这一篇真正的题眼。这些测试是绿的。因为它们是快照测试：跑一遍，把输出存下来当作「预期结果」，下次比对。有人在这中间刷新过一次快照，于是满屏的「找不到这个工具」被原样存成了新的正确答案。从那以后，测试每次都在忠实地验证「它稳定地失败」。

> **复盘里的原话：**快照刷新是fixture的生产过程，不是正确性审查。诸如已注册工具缺失这类语义上不可能的结果，需要独立于预期输出的断言。

他们加了三道防护：配置里那一格不许再出现表达式，有专门的门禁扫描全仓库的配置文件并拒绝；快照工具从此拒绝把「找不到工具」这种结果收进预期输出；文件系统场景改用一份显式的独立配置层。

这道禁令后来自己被推翻了，这一步比事故本身更好看。8月11日的一篇笔记记着，加载器现在会真的去求值那一格，而且只有那一格，因为Windows那条线需要按平台开关某些行。他们的说法是：这个隐患是靠求值关掉的，不是靠禁止。

还有一句克制得让我意外：他们特意写明，这个bug没有让默认模式意外获得文件系统权限，而且当时如果图省事直接去修那个表达式，反而会真的制造出这个风险。承认自己搞砸的时候，还不忘把影响范围说准，这个分寸感不容易。

第三篇：它跑去验收了一台替身服务器

四篇里最适合非技术读者的是这一篇，因为它讲的是agent最常见的那种错法：**它说做完了，但它验的不是你那台。**

场景：用户正在3081端口的网页界面里跟它对话，让它改这个界面的主题。它改完源码，接下来做了三件事，一件比一件离谱。

第二步那个「200」值得单独说。它启动了一个精简版的开发服务器，用命令探一下，服务器回了个「一切正常」的状态码，它就宣布成功了。而用户在浏览器里看到的是一片空白，因为完整的界面还需要宿主注入一份启动清单，精简版不注入。**网络通了，不等于页面能看。**

第三步它找到了正确的启动方式，但起在了3334端口，然后只检查了3334。用户真正在用的3081，它从头到尾没探过一次。用户连着指出三个错误之后，它才回头去看现有进程，而那时3081上早就显示新主题了。

根因：这套组合没有把「你现在这个会话跑在哪个网址、哪个进程、什么模式」告诉模型。模型只知道当前目录，而那是用户选的工作区，不是应用目录，它猜错了。

修法也很实在：把当前网址和运行模式做成模型能看见、也能用命令查到的东西，写进环境变量`$DSH_WEB_URL`和`$DSH_WEB_MODE`；那个会误导人的精简版服务器，从此在启动阶段就直接拒绝跑起来。

> **核心建议**
> 这一篇的三句教训，我建议任何用agent改东西的人都记住：网络就绪、构建成功、启动清单，是三件不同的事实；验收必须指定确切的目标地址，并且从外部观察改动有没有在那个地址上生效；**一台替代服务，永远证明不了你那台页面变了。**

第四篇：一行无害的提示，被当成了沙箱故障

第四篇最技术，但它藏着这四篇里最深的一句话。

在比较老的Linux内核上，负责把命令关进沙箱的那个小程序，每次执行前会先打印一行提示：我只能做到部分约束。这是善意提醒，不是错误。而它真正失败的时候，会打印另一行诊断，并且以一个固定的退出码退出。

两种情况的开头是同一个前缀。harness偷了个懒，只用这个前缀来分辨它们：只要看到非零退出、同时带这个前缀，就判定为沙箱挂了。

于是出现了这样的场景：你让它在项目里搜一个关键词，搜索工具没搜到东西，按惯例以退出码1退出（这是正常的空结果）。这个正常结果撞上那行善意提醒，被判成「沙箱不可用」，再被上一层包装成一个笼统的「搜索失败」。**没搜到，变成了系统故障。**

他们的影响评估同样克制：约束没有被削弱，也没有任何命令因此跑在无约束状态下，损失在可用性和诊断的完整性上。有效的受限结果被拒绝或者被贴错标签。

而根因段最后那一句是这样写的：

> stderr仍是带内归因通道。受限子进程可以故意复现runner的门控致命诊断行与退出状态，造成可用性或诊断误归因。

翻译成人话：**沙箱是靠读被关起来那个程序自己吐出来的文字，来判断沙箱有没有出事的。**那个程序完全可以照着格式把那行字打一遍。他们没有假装这已经解决了，而是明说：加更多证据能避免这次这种意外撞车，但无法验证是谁写的那行字，真正的修法是另开一条带外通道，属于后续加固工作。

一个刚发布的产品，在自己的事故复盘里写下「我这个判断依据原理上可以被伪造」，我没见过第二家。

四篇之外，它们把教训固化成了什么

复盘写得再漂亮，人也会忘。所以我更在意后面这两样，它们把教训变成了机器强制。

第一样，每个包都得带一份「我这儿有什么是运行时能查的」。在仓库里数了一遍：219个包，一个不漏，各带一份。其中**184份是空的**。（提醒一句：全书出现过三个184，互不相干。§04那个是装到你电脑上的包数，这里是空文件数，§13那个是讨论区的帖子数。我核过三遍，确实是巧合。）

```
find packages -name "invariant.ts" -path "*/src/*" | wc -l
# 219
grep -rl "No runtime invariant:" packages --include=invariant.ts | wc -l
# 184
```

关键在于空得有规矩：空实现的第一行注释必须以`No runtime invariant:`开头，后面写清楚为什么这个包没得可查，而且理由必须是这个包特有的。有一道门禁机械拒绝糊弄的写法：生成式的占位、没解释的空实现、以及拿到了检查工具却不用它的实现。

我随手翻开一个，理由写的是：这是个纯工具包，不拥有任何事件流或可变的运行时数据，它的取值规则由单元测试保证。**「我这里没什么可查的」也必须署名说明，不能空着交上去。**

第二样，崩溃之后写给模型看的那段话。如果进程在一次工具调用记下之后、结果落盘之前挂了，重启时系统会替那次调用合成一条结果，内容是这样的：

> 这次工具调用在被记录之后中断了，但没有任何结果被持久记录。它的结局未知。要不要重试，请从这个工具的语义出发判断：只有当操作是只读或幂等时才重试；如果它可能有副作用，先去核实外部状态或者问用户。不要盲目重试。

还有另一条，给的是没来得及开始的那种：这次调用在被记录为已开始之前就中断了，如果还需要，重试即可。

两句话的差别，是「我不知道那个删除命令有没有执行」和「那个删除命令肯定没执行」的差别。断电重启之后，它既不假装没发生过，也不假装已经完成，而是把不确定性原样交给模型，并且当场给出判断规则。

还有一处至今没修的自曝

最后这条不在四篇复盘里，是我在设计笔记里翻到的。

他们做过一次内存实测，结论是对象本身很干净：agent销毁之后，占的内存几乎全数归还。但笔记的结论句是这样一句：

> 所以泄漏的不是对象图，是生命周期。

因为网页版的宿主**压根不销毁agent**。创建出来的句柄被直接扔掉，登记表里也没有任何淘汰机制。这条至今挂着一个待办标记，没做。

把「我这里有个已知的泄漏，而且我知道它为什么泄漏，我还没修」写进公开仓库，跟前面那句「我的判断依据可以被伪造」是同一种姿态。我不知道这种姿态能维持多久，第一个正式版本发出去、有了真实用户之后，压力完全不同。但它现在是这样，这一点可以被查证，也值得记下来。

他们本来想这么做，后来没做

> 上面那段崩溃恢复的文案，本来差点不存在。2026年6月20日有一份提案，主张把这套「补齐」机制整个删掉：崩溃之后加载日志时，直接丢掉最后一个完整回合之后的所有内容，干净利落，也不用凭空造出一条没有任何工具产生过的结果。提案自己给的理由是「产品还没发布，这套恢复语义没有被真实用户验证过」。它被否了，否决理由只有一句：一个回合可以包含大量真实工作，包括很多步骤和很大的工具输出。**把它丢掉的代价，比留下一条脏记录的代价更贵。**这份提案至今躺在被否决的目录里，一共11篇，它是其中之一。

四篇复盘的正文都不是「那一行代码怎么修的」，是「我们的流程为什么没拦住」。那这个流程，是谁在跑？

§13 一个几乎全由AI写出来的代码库长什么样

What an AI-Written Codebase Looks Like

这一节跟你明天要干的活没关系。它回答的是另一个问题：一支工业级团队让agent替自己写了两个月代码，施工现场会留下什么痕迹。这些痕迹他们一条没删，全公开了。我翻完之后最想递给你的，是藏在里面的一句话。

我为什么会在这儿停下来

先摆清楚我的位置。我从来不写代码，小猫补光灯、女娲那一堆skill、每天在用的那些小工具，一行都不是我敲的，全是AI写的。

所以这两年我一直在猜一件事：如果不是我这种一个人配一个AI的野路子，而是一支正经团队全面用agent开发，那个现场到底长什么样？他们的规矩会变成什么样子？

猜没有用，因为没人给你看。产品发出来是干净的，提交历史通常也被压平过。你看得到成品，看不到施工。

DSH这个仓库是我见到的第一个把施工现场原样端出来的。683篇设计笔记、每一次否决的理由、一整套逼着AI交作业的检查程序，甚至连「AI写英文文档时不小心漏出中文」这种事，都有一个专门的工具在处理。

这批材料我没见过有人写过。我把它读完了。

> 这一节不教你怎么用它，你完全可以跳过去看下一节。但如果你自己也在用AI做东西，我建议别跳，因为他们踩过的那些坑，你迟早也会踩到。

先看四个数字

项	值
提交次数	12,293
开发天数	64（2026年6月10日到8月13日）
单日最高提交	887（7月30日）
设计笔记	683篇

日均192次提交。7月30日那天887次，同一天还产出了47篇设计笔记。7月27日到31日五天，一共3,282次提交。

人数这个数字得多说两句，因为它有四个口径，随手挑一个就会跟别处对不上。git历史里带署名带邮箱的作者条目46条，按邮箱去重40，按署名去重37，而GitHub接口认的贡献者只有22（网页上那栏显示得更少，19）。同一个人换过邮箱，所以前几个数一路缩水；GitHub只算能关联到账号的那部分，所以它最小。安全的说法是「三十几个人，GitHub认下来二十出头」。

三十几个人，两个月，一万两千次提交。这个吞吐量人手是敲不出来的。

还有一个数值得单独拎出来：仓库里有2,355个markdown文档，2,319个TypeScript代码文件。**文档比代码还多。**

> **一个容易看漏的前提：**这个GitHub仓库是2026年8月13日当天才建的，而第一条提交是6月10日。也就是说，两个月的开发史发生在一个私有仓库里，公开那天一次性推上来。你现在能翻到的每一篇笔记、每一次否决，都是他们主动选择不删的。

那句解释一切的话

如果你只从这一节带走一句话，是下面这句。它出自开工第二天写下的一篇制度笔记：

> 「This codebase is developed primarily by coding agents. Agents follow enforced gates far more reliably than prose conventions, and 'a lot of work' is not a cost argument when agents do the labor.」
>
> 「这个代码库主要由编码agent开发。相比写在文档里的约定，agent遵守被强制执行的检查要可靠得多；**而当劳动由agent完成时，『工作量很大』不构成一个成本论据。**」

最后半句是钥匙。

你读这本书的时候大概反复冒出过同一个疑问：为什么规则这么密？为什么每个包都要写README？为什么改一行代码要配一篇文档？换成任何一家传统公司，这些提案在会上活不过三分钟，反对理由永远是同一句：工作量太大了，谁写？

而当写的那个不是人的时候，这句反对话直接失效了。反对的成本没了，规则的密度就能一路加到「凡是能机器检查的，全部机器检查」。

我觉得这句话的适用范围远远超出这个仓库。你自己用AI干活的时候，那些你嫌麻烦所以从来不做的事，值得重新过一遍：不是它们没价值，是「麻烦」这个理由已经不成立了。

Agent Note：每改一次，必须交一篇作业

683篇笔记不是有人心血来潮写的，是被制度逼出来的。

规矩有三条。

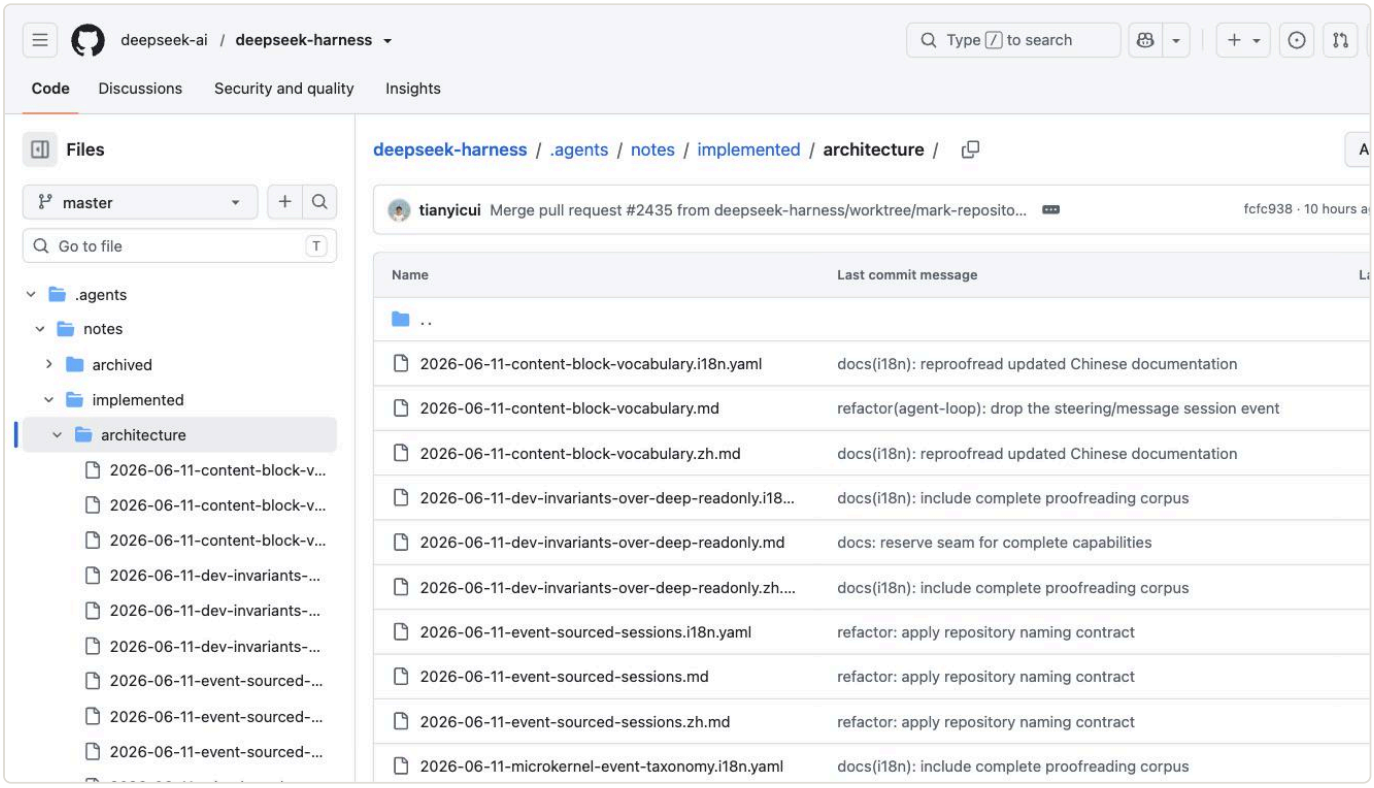
第一条，每个非平凡的改动，必须在同一个提交里新增或更新至少一篇笔记。代码和笔记捆在一起走，不许后补。

第二条，笔记里必须写清楚「我们否决了哪些方案」。这一节缺了，自动检查过不去，代码合不进去。他们给这个要求的原话是「一个不写清它打败了什么的决策，会招来翻旧账，而翻旧账正是这套笔记要防的事」。

这条还有一句更狠的补充：**「Alternatives are recorded, never invented.」**替代方案是被记录的，绝不是被发明的。制度定下来之前写的那批老笔记，如果替代方案确实已经无从考证，就必须永久带上一行注释，明说「这里的替代方案没被记录」。他们宁可在档案上留一个诚实的窟窿，也不许补一段编出来的理由。

第三条，笔记归档之后哈希冻结，永久不许改。不修、不译、不重排、不移动、不删除，也不许再拿它当现在的权威。有一个只能往后追加的清单记着每份文件的指纹，谁动过一个字，检查程序立刻知道。我数了一下，那份清单里躺着426条封存记录。

还有一个细节：笔记的分类不写在文件里，而是编进文件路径。一篇笔记放在哪个文件夹，它就属于哪一类、处在哪个状态。他们否掉了「在文件头部写一行分类」的方案，理由是那等于把路径已经承载的事实又抄一遍，而抄第二遍的东西迟早会跟第一遍打架。



> 路径本身就是分类：implemented 是状态，architecture 是类别。一个议题三个文件，英文、中文、外加一份对齐用的，文件名统一日期打头。

683篇按状态分是这样：已实现505篇、已归档142篇、提了还没做25篇、提了被否掉11篇。

最后那11篇是全书我最舍不得的素材。正常公司不会把「我们差点做了什么蠢事」写下来公开，他们不但写了，还专门规定：一篇被否的笔记，只在它还能拦住一个「看起来很诱人的错误」时才保留，否则连同译本一起删掉。留下来的这11篇，等于团队自己认证的「未来还会有人再想一遍的坑」。

最外行也能看懂的那条证据

前面这些制度，你可能会觉得只是这家公司比较讲究，未必能证明活是agent干的。

那看这个。

他们的工具库里有一个专门的清理程序，名字直译过来是「修剪思维链泄漏」。所谓思维链泄漏，指的是AI写出来的文字带着写作现场的痕迹：引用只有那次对话才看得见的东西、叙述改动过程而不是当前状态、或者对着一个已经离场的评审者辩解「这个写法是安全的，因为……」。

它列了八类要清理的毛病。第八类是这样定义的：在整体是英文的文字里，出现没翻译掉的工作语言碎片。它给的例子是「端」「设计稿」，还有一行中文的分隔符。

你把这句话拆开看。要写一个程序去擦掉英文文档里漏出来的中文词，前提有两个：**写这些文档的东西真的在自动生成英文，而它脑子里的工作语言是中文。**

一个证据同时坐实两件事：agent在深度参与，以及这支团队开会用中文。

还有一条旁证也很有意思。我统计了合并记录里的分支前缀，用于并行开发的`worktree/`出现210次，`codex/`出现203次，另有`agent/`15次、`claude/`3次。`codex/`是某类编码agent建分支时的默认命名习惯，203次不是手滑。

但边界也得划清楚：提交的署名全是真人邮箱，我翻遍了提交记录，没有一条写着「本次由AI协作完成」。所以署名归属是人，agent的痕迹在分支名、在吞吐量、在为它准备的那套基础设施里。至于683篇笔记具体哪一篇是agent一稿写成、哪一篇被人改过，我查不出来，这条标「未验证」。

27道检查，12个生成器，和唯一留给人的那一道

我把他们的检查脚本数了一遍。**27道机械检查，12个生成器。**（说明一下数法：脚本目录下以verify开头的文件一共35个，但其中8个是这些检查程序自己的测试文件，得剔掉，所以真正的检查是27道。生成器同理，17个文件里5个是测试，剩12个。你要是看到别处写「35个门禁」，那是把测试文件也数进去了。）

这27道管的事情包括：笔记格式对不对、分类跟文件夹一致不一致、归档的封条动过没有、文档里的链接有没有失效、中英文文档配对齐了没有、每个包的README有没有写「已知局限」那一段。最后这条我实测到220个文件里都有那一段，一个没漏。

然后是那个反差最大的地方。

「每个非平凡改动必须带一篇笔记」这条规则，**是全套制度里唯一没有自动检查的一条。**他们在提案里明确否掉了「加一个自动判断改动大小的检查」，理由是机器没法可靠地判断一个改动在语义上平凡不平凡，加了只会带来误报和表面合规。这一条明写着交给人来审。

能机械化的一律机械化，判断力留给人。这个分界线是这套东西里最值得抄的部分。

一个人，5,235次提交

提交量排第一的是崔添翼（Tianyi Cui），**一个人5,235次，占全仓42.6%。**前三个人加起来7,746次，占63%。

他的履历有公开信息可查：2026年3月加入DeepSeek，出任新组建的Harness团队负责人。据南华早报报道，他此前在量化交易公司Jane Street待了将近九年，身份是软件开发和研究，方向是股票和固定收益；离开之后自己联合创办过一家量化公司。

为什么要提他的前东家？因为内测开发者JY（@jiayuan_jy）读完代码之后说了一句话：

> 「另外从代码上来看，DSH有非常多函数式编程的影子，不熟悉Ocaml/Haskell可能一上来会比较难理解，可以多让Agent ELI5一下。」
>
> （出处：https://x.com/jiayuan_jy/status/2087911060154314963）

Jane Street以OCaml这门函数式语言立身，这是公开事实。**但「他本人在那里主要写OCaml」是我的推断，不是查到的事实**，我没找到任何一手材料能坐实。你可以把它当成一条挺顺的因果线，但别当结论用。

JY那句话的实用价值在另一头：如果你打算翻这份代码，先做好心理准备，它的写法跟你在大多数前端项目里见到的不一样。

造好了一整套闸门，发布当天把闸门关了

这是我在这个仓库里发现的最像活化石的一处。

仓库里躺着一整套完备的问题单管理设施：五个提单模板，分别对应bug、功能、想法、调研、任务；一个策略引擎，带标签白名单、p0到p3的优先级、状态机、审计标记；配套还有一条专门测这套策略的测试命令。

然后你去仓库首页看，问题单功能是关着的，所有反馈全部导去讨论区。

这套闸门显然不是为公开发布做的，它是内部研发流程的产物，开源的时候被原样端了出来。你现在能看到一家公司内部怎么给自己的活分优先级、怎么定状态流转，只是那道门对外锁着。

顺便说，这个选择的代价发布当天就来了：仓库公开后不到三小时，讨论区涌进184条，Windows用户的安装问题、中文路径被截断、还有人报了一个沙箱边界能被绕过的安全问题，全都混在一起，没有标签，没有优先级。**闸门造好了却关着，洪水就自己找路。**

同一件事，两种反应

最后说一件我自己在场的事。

2026年3月底，Anthropic更新Claude Code的安装包时手滑，把一个60MB的调试文件留在了发布包里，1902个源文件被完整还原出来。我当时花了几个小时把那1902个文件翻了一遍，写成了一篇文章，也把它做进了另一本橙皮书的附录。

我的反应是：**这下终于能看见一个真正好用的harness里面长什么样了。**对我这种不写代码的人来说，那是一次难得的开箱。

而在另一边，同样是2026年3月，崔添翼入职DeepSeek，带一支全新的Harness团队；到5月，这支团队已经在X上公开招人，招聘帖直说「从零开始造一个Code Harness」；6月10日，仓库第一条提交落下。

他们的反应是：**那我们也得有一支harness团队。**

这个对照得加一句免责声明：泄露和入职发生在同一个月，谁先谁后我钉不到日期，公开材料也没有任何一句把两件事挂上因果。这条线是我自己连的，你可以不认。

但时间摆在那里，供你自己判断。一件同样的事情，一边的人得到的是一次开箱的机会，另一边的人得到的是一份施工图纸和一个必须开工的理由。

更有意思的是结局。他们造出来的这个东西，把自己的施工现场也一并公开了，而且是主动的，不是手滑。683篇笔记，包括他们否掉的那11条路。

所以这一节读到最后你会发现一个回环：我因为一次事故看见了别人的harness，而现在，另一家公司主动把harness连同它的建造过程一起摊在桌上。不用等谁手滑了。

他们本来想这么做，后来没做

683篇笔记，没有目录页。这不是漏了，是被明令禁止的：分类检查程序会直接拒绝根目录下出现一个总目录文件。他们本来是有的，7月19日一篇提案把它删掉了，理由只有一句：**「每一个新增、移动或重命名笔记的分支，哪怕彼此毫无关系，都会重写同一个生成文件，让这个产物变成一个可预测的合并冲突热点。」**放弃的东西也写清楚了：读者从此失去一个按时间排的总览页，改用文件夹和全库搜索。这条判断只有在agent高频并行开发的前提下才成立。人手一天改三次的项目，目录页当然该留着；一天887次提交、几十条分支同时在跑的项目，最大的敌人不是「找不到」，是「所有人同时改同一个文件」。

§14 它在赌什么，代价是什么

The Bet and Its Price

前面十五节讲的是它是什么、怎么用、坑在哪。这一节我不做总结，我下判断。一个公开还不到二十四小时的东西，值不值得你往里投时间，答案不在功能列表里，在它替你做掉的那几个选择，以及每个选择的背面它扔掉了什么。

先答那个所有人都在问的问题

它的说明文件里有一句全大写的英文：将会有破坏兼容性的变更。这句话的杀伤力不在劝不劝你装，在劝不劝你把它接进日常干活的流程。它劝退的是后者，于是把人锁死在「玩玩」这一档。

我的答案分两半。会白学的那一半：具体的配置写法、你手捏的那套模式、你写的插件接口。这些三个月内一定会变，它自己都说了会变。

不会白学的那一半：它替你做的那几个判断。判断跟版本号无关。你看懂它为什么在这里放弃了A、保住了B，这个理解在你评估下一个agent产品时照样有效。

所以这一节我把它下的注一条条摊开，每条都写两面：赌什么，为此扔掉什么。**只写好处的清单是广告，不是判断。**

第一注：一切皆插件，赌生态，扔掉开箱即用

这句口号有多字面？连agent的主循环本身，都是配置文件里可以加一行 `disabled: true` 关掉的一项。它自己的说法是：没有一个特权内核等着你去打补丁。

更有说服力的自证不是这句话，是他们删掉的东西。他们做过一个专门的插件格式，里面给技能和MCP留了特殊通道，8月9日全删。理由是技能和MCP不需要特殊通道，它们就是普通插件，用普通包管理器装就行。

赌的是有人真会来拆、来换、来重装，收益全部押在生态上。

扔掉的是当下的体验。内测开发者JY说得比我还直接：

> 「如果拿Coding Agent的标准来说，当前DSH的体验确实不如Claude Code / Codex那么完善。整个项目还很早期，接口一直在变化，插件生态也才刚刚开始，质量肯定是层次不齐的。」
> JY（@jiayuan_jy），2026-08-13
> https://x.com/jiayuan_jy/status/2087911060154314963

这话出自一个提前一个月进仓库、亲手用过的人。他没打圆场，我也不打。

还有一个更具体的代价，我在本机数过：跑起来之后运行时里133个插件在转，设置页里能点着改的只有3张卡。**「一切皆插件」是真的，「一切皆你能碰的插件」不是。**口号和界面之间的这段落差，是它现在最劝退人的地方。

第二注：每一次运行都有迹可循，赌可审计，扔掉磁盘和简洁

一次会话是一条只能往后追加、不能回头改的事件流，44种事件里只有3种是模型看得见的：你说了什么、它说了什么、工具返回了什么。剩下41种模型一个字都读不到，纯粹记给人和审计看。

为了守住这条规矩，它做过一个很反直觉的取舍。PTC模式里，模型写的那段代码本来可以跑在一个常驻内核里，变量跨调用存活，又快又省。他们拒绝了，理由是跨调用的状态不进日志，会破坏「每次请求都是日志的纯函数」这条不变式。

Anthropic在自己的代码执行接口里恰恰加了这个特性，两家往相反方向走。这不是谁对谁错，是两种赌法：一边赌性能，一边赌可重建。

扔掉的东西同样具体。仓库里有个最普通的样本，跑一条命令再回一个词，日志里落下35条物理记录。跨工作区恢复会话那篇笔记把代价写在明面上：这台机器上每个项目的会话日志都存在同一个目录，而且这个决策不引入任何保留期策略。

翻译过来就是：它不会自己删。硬盘是你的。

第三注：技能和MCP只是普通插件，赌抽象正确，扔掉贴合度

这一注最容易被误读，我先说结论：**它对我是净赚的，对很多人是净亏的。**

他们那张「每项产品功能都映射到一个扩展点」的表一共29行，技能和MCP排在末尾，和界面、持久化并列。在这套世界观里，技能不是一等公民，是二十九分之一。

净赚在哪：正因为技能只是「一个会扫描目录的普通插件」，扫描路径就能做得很宽。我本机`~/.agents/skills`下那57个技能，一行没改、一个软链没加，它直接读到了。

净亏在哪：MCP只桥接工具那一类，资源和提示词明确不做，默认一个服务端都不开。理由很硬气：每条服务端命令都是agent沙箱之外的可信可执行代码。你要是挂了七八个MCP在干活，搬过来会掉东西。

再加上§04讲过的那件事：219个包里有35个不在默认安装里，要单独敲命令装，照最直觉的写法还会装到三天前的旧版。这里面就有接Claude Code钩子的桥、接编辑器的协议、语言服务。

这一注的账要这么算：**技能这边门开得比谁都大，MCP和外围工具这边门修得比谁都窄。**

第四注：模型公司自己下场，赌能力兑现，扔掉中立

为什么模型公司要亲自做这一层？有个外部数据比任何自述都硬。Composio在8月11日做过一次实测：固定DeepSeek V4-Flash这一个模型不变，套8个不同的harness，跑30个跨软件的复杂工作流，一共240次。通过率从46.7%到66.7%，每次成功的成本相差约7倍。他们官推那句话说得更直接：**有7个任务，成败只取决于我们用了哪个harness。**模型一个字没改。

把这一层交给别人，等于把「我的模型好不好用」的一半交出去，任务失败时还分不清是模型的锅还是脚手架的锅。这就是它下场的理由。四种模式里最不起眼的那个极简模式，只留一个shell和一个文件编辑工具，官方自己说是给最小环境下的模型基准测试用的。**那个模式不是给你用的，是给它自己刷分用的。**

扔掉的是中立性，而且可以指着看。它的遥测默认是关的，比我预期的克制；上报地址倒是硬编码在出厂配置里，只是那个开关出厂就停在关的位置。但有一个匿名编号例外：跑起来之后本机会生成一串UUID，随每一次发往DeepSeek的请求走。你把遥测开关关到底，这串编号照走不误。

> 这件事是他们自己写下来的，就躺在那个包说明文件里一个叫「已知限制与待办」的段落。这段不是良心发现，是被门禁强制的：全仓220个文件都有它，缺了合并不进去。**把「我们哪里不行」做成一道通不过就合不进的门禁**，是我在这个仓库里看到的最值钱的东西，比任何架构都值钱。

要不要因此不用它？我的判断是不用紧张，但你得知道它存在。代码不支持「DeepSeek在偷数据」这个结论，走出去的不是文件内容，是一串跟着你的编号。

诚实的代价清单

把散在各处的坏消息集中放一遍。

这一条	对你意味着什么
说明文件里全大写的「将会有破坏兼容性的变更」	别沉淀插件和自定义模式；技能可以，那是共用格式，搬得走
内测开发者亲口说体验不如现成的那几个完善	当主力干活会难受，当框架拆着玩正合适
插件生态刚开始，质量层次不齐	打插件标签的仓库，调研时238个，8月14日凌晨476个，但打标签不等于是插件
文档已经在多处落后于代码	照文档抄的命令，先自己跑一遍验
匿名编号跟着请求走，关不掉	知道即可，涉密项目自己判断

「文档落后于代码」这条，我给三个自己抓到的现场。第一处，创造模式那个包的说明文件写「五个模型可见的工具」，我实测运行时里是七个，其中一个已经拆成了三个。

第二处更严重：有个工具早就改了名字，旧名字却还留在随产品发出去的一份技能文件里，而那份文件是喂给模型看的。文档写错顶多骗到人，这个直接骗模型。

第三处最有戏剧性。它把底层框架的9个包整个抄进了自己仓库，三处文档都写着这些包「设为私有，从不发布」。实际上9个包里压根没有那个私有字段，而且我逐个查过，**9个全都发到npm上了**。三处文档，一致地错。

说句公道话，一个每天几百次提交的仓库，文档落后是必然的，我在§04里也写反过一次。**要提醒你的不是「他们文档差」，是「这个阶段的任何文档你都不能直接信，包括这本书」。**

回到那只手

§07结尾我留了半句话没说完：它能给自己长出一只手，但这只手活不过一次重启。

那次实测的画面值得再放一遍：我让它现场造一个数汉字的新工具挂到自己身上，再用它数一句话。它做到了，答案正确。整个过程19步、24次工具调用，前14步、20次调用它几乎全在读自己的说明书，真正动手只花了最后4次。**这是一个agent在读自己身上零件的规格书，为了给自己再装一个零件。**

然后是那条边界，官方写得比我狠：动态生成的插件只活在进程的共享内存里，不创建任何文件、不安装任何包、不改动任何配置、不能在重启后存活，也无法被自动提升为正式插件。

从「能现场长出能力」到「长出来的能留下」，这一步意味着什么？

我的判断是：**卡住这一步的不是技术，是那条不变式。**它全部的可信度都建立在「模型看得见的，一定被记录过」上，一次会话的所有东西都在那条事件流里，跑一遍就能重建。可一旦允许agent写下能改变下次启动的东西，这条不变式就得管到跨会话去，管到你硬盘上那个配置目录里去。那时候「这个agent现在是什么样」，就不再是一份日志能回答的问题了。

所以他们不是做不到，是不敢默认打开。宁愿让这只手在关机时消失，也不愿意让一个自己改自己的东西悄悄留在你电脑上。

我认为这个选择是对的，也认为它撑不了太久。路他们自己已经修好了：既然一切都是插件，那「让agent写一个真的插件文件」和「让agent在内存里挂一个」之间，隔的只是一个默认值。

我的判断

先回答一个绕不过去的问题：一个不写代码的人，为什么值得关心一个开发者框架？

我做的所有东西都是AI写的，从头到尾没写过一行代码。**那我唯一的杠杆，就是改变这个AI的工作方式。**过去两年能用的改法只有一种：写技能，写一份Markdown告诉它遇到什么情况该怎么做。这一层好用，但天花板很低。真正决定agent行为的那些东西，什么时候压缩上下文、什么时候派子代理、撞了墙怎么办，我碰不到，只能等产品方改。

这个东西第一次把那个天花板掀了。连主循环都是一行可关的配置，这对写代码的人是架构选择，对我是「原来那一层也可以谈」。我未必真去改，但知道它能改，我对agent的想象就不一样了。

适合三类人。一是想搞明白agent底下到底怎么转的，它是目前唯一一个把完整答案摊开、还配了六百多篇设计笔记的开源实现。二是要做插件、做工具、做界面的，生态刚开始，现在进场位置最好。三是像我这样攒了一堆技能资产、想找个不用搬家就能试的新窝的。

不适合三类人。你是Python栈、想把它嵌进自己产品的，它是TypeScript生态，没有HTTP层给你桥。你要多租户、要并发、要按会话给客户计费的，现在没这东西。你只想找个更顺手的编码助手、不想折腾的，别来。

说回我自己。我不会把它换成主力。我的活是写文章、做视频、做产品，主力工具要的是稳，不是可拆。会用它的地方有两个：一是拿它当技能生态的第二块试金石，同一套技能在两个不同的harness上都跑得通，这套东西才算真的活着；二是当素材，因为它是我见过第一个把「模型到底看到了什么」完整摊给你看的产品。

最后借一段别人的话收尾。它公开那天的Hacker News主贴底下，有人问了一句在这家公司身上问了很多年的话：「这个他们又是抄谁的？」另一个人回：「这是个很危险的问题。」再问为什么，答：

> 「because this time it looks pretty much original ?」
> 「因为这次它看起来相当原创？」
> https://news.ycombinator.com/item?id=49285244

一个长期被指控抄袭的公司，第一次在那个刻薄的地方被承认原创，不是因为发了模型，是因为开源了一个harness。

它赌的东西还有好几年才见分晓。但这一句已经兑现了。

他们本来想这么做，后来没做

> 他们认真考虑过把「会话怎么存」那个接口包折回主包里。诊断诚实得刺人：在还没有任何第三方来实现它的时候，一个单独的接口包看起来就像是提前抽象。否决的理由不是「你说得不对」，是笔记里那句原话：如果第三方的存储后端已经是一个公共生态，这个独立接口包就是更干净的边界；发布前它才像多余的。**他们承认这是一个赌注，然后赌了。**这本书讲的所有取舍，几乎都是这一个动作的不同形状。

§99 附录

Appendix

这一节没有故事。四张速查表，加一段关于保质期的话。所有数字后面都跟着取数命令，全是我自己敲过的。你可以把这几页当工具箱用，也可以拿它抽查全书。

先说清「快照」这两个字

写这本书的时候，这个产品公开还不到一天。

所以下面每一个数字都带着时间戳。每个数都只是我在某一刻按下回车看到的东西。仓库那一侧的数比较稳，网络那一侧的保质期以小时计。

> **统一快照时刻：2026年8月13日23:05到23:20（北京时间），对应世界时15:05到15:19。**仓库版本停在 `47f9438`，本机装的是 `@deepseek-ai/dsh@0.1.0-rc.6`。下文凡是没单独标时间的，都是这一格。

还有一件事得先交底：**你装到的那一版，比GitHub上那份快照还新半步。**仓库里的版本号写着rc.5，npm上和你机器里已经是rc.6。凡是「代码里写着A、装上是B」的地方，先怀疑这半步的版本差，别急着当成文档出错。

速查表一：数字基准

先是仓库这一侧。这些数大多锚在git历史或生成文件上，除非仓库大改，隔一个月再跑也是这些。

数	值	自己怎么取	口径说明
提交次数	12,293	`git rev-list --count HEAD`	单分支线性计数。整个仓库只有一个分支，一个tag都没有
开发时长	约64天，首条提交2026-06-10 22:57	`git log --reverse --date=iso`	取的是作者时间。提交者时间晚72秒，两个都真实，不是有人改过历史
参与人数	46 / 40 / 37 / 22	`git shortlog -sne HEAD | wc -l`、`git log --format='%ae' | sort -u | wc -l`、`git log --format='%an' | sort -u | wc -l`、`curl .../contributors?per_page=100`	四个口径一路缩水：带署名带邮箱的作者条目46条，按邮箱去重40，按署名去重37，GitHub接口只认22（网页那栏显示19）。同一个人换过邮箱就会被算成两个。稳妥说法是「三十几个人，GitHub认下来二十出头」
设计笔记	683篇	`find .agents/notes -name '*.md' ! -name '*.zh.md'`	排除中译本和5个目录说明。全算是688，连中译本一起数是1372。689这个数任何命令都数不出来
被否决的笔记	11篇	`ls .agents/notes/rejected`	10篇简化提案加1篇功能提案。6月20日那一天一次否掉5篇
事故复盘	4篇	`ls docs/postmortem`	编号0001到0004，每篇配中英文和翻译对照三个文件
机械门禁	27道	`ls scripts/verify-* | grep -v '\.spec\.'`	目录下有35个verify开头的文件，其中8个是测试。书里写27比写35抗打
包数	219个	遍历 `packages/<组>/<包>/package.json`	全仓带名字的配置文件有248个，另外那29个是示例、夹具和外来源码，别算进来
Markdown比TypeScript多	2,355 vs 2,319	`git ls-files '*.md' | wc -l`	md含中译本。非测试TypeScript共272,267行，含空行和注释

然后是装上之后这一侧。这几个数决定你手里到底有什么。

数	值	自己怎么取	口径说明
安装闭包里的目录	195个	`ls ~/.npm/_npx/<hash>/node_modules/@deepseek-ai | wc -l`	184个来自仓库的包目录，9个是随源码带进来的Cordis生态，2个是应用本体
仓库有、装了没有	35个	219减184	**这35个全在npm上，只是不在默认安装闭包里**，要单独装一条。清单见速查表四
闭包磁盘占用	342MB	`du -sh ~/.npm/_npx/<hash>/node_modules`	其中DeepSeek自己的部分只有27MB。命令写法不同，npx会各存一份缓存，卸载时容易漏
启动器本体	约114KB，20个文件	npm仓库的解压体积字段	它有61个直接依赖。一个114KB的启动器，拉起一棵342MB的树
四份模式清单	251 / 262 / 262 / 62行	`wc -l config/agent-presets/*/agent.cordis.yml`	依次是标准、PTC、创造、极简。本机安装版和仓库逐行相同
标准模式工具数	25件	端到端测试里的清单	两份「25件」的清单内容其实不一样，一份是网页标准模式、一份是无界面模式，都是25件纯属巧合
事件类型	44种，模型看得见3种	`grep -cE '^#### ' docs/persistence-catalog.md`	另外那24个三级标题是命名空间不是事件，最容易数错这里
会话日志格式	zstd压缩	`xxd -l 16 session.jsonl.zstd`	魔数是 `28b5 2ffd`。它不是加密，一行命令就能解开

剩下这张是钱和外面的热度，保质期最短的就是它。

数	值	自己怎么取	口径说明
模型价格（每百万token）	flash：缓存命中0.0028美元、未命中0.14、输出0.28 pro：0.003625 / 0.435 / 0.87	官方定价页	缓存命中价比未命中价便宜得多，但两个模型的倍数不一样：**flash是五十分之一，pro是一百二十分之一**，别把flash那个比值套到pro上。这就是「前缀别乱改」值多少钱的答案。官方同页写明2026年8月17日0点（北京时间）起改分时计价，书出在那之后这张表必须重取
上下文窗口	100万token	`curl https://api.deepseek.com/models` 加官方文档	模型一共只有两个。官方说最大输出38.4万，而这套东西自己把默认上限压到25.6万，它没用满
GitHub星标	公开当天三个多小时，两万多	`curl https://api.github.com/repos/deepseek-ai/DeepSeek-Harness`	快照15:19Z。我在8分钟里取了两次，差830颗，所以这里只给量级，不给精确数
底座框架Cordis的星标	快照那天八百多，之后一路在涨	`curl https://api.github.com/repos/cordiverse/cordis`	2022年建的个人项目，这几天正被带飞。稳妥说法是「公开之前只有几百星」，别写成它现在的量级
打了插件标签的仓库	调研时238，8月14日凌晨476	GitHub搜索接口查 `topic:dsh-plugin`，或直接开 `github.com/topics/dsh-plugin`	主仓README号召大家打这个标签，所以它在膨胀——半天翻一倍。打标签不等于是插件，里面混着空仓库
npm最新版	0.1.0-rc.6，2026-08-	`curl https://registry.npmjs.org/@deepseek-ai/dsh`	一天走了四个版本

数	值	自己怎么取	口径说明
	13T12:35Z 发布		
仓库建库时间	2026-08-13T11:56Z	GitHub接口的创建时间字段	而第一条提交是6月10日。这个仓库是当天新建、一次性推上来的，两个月的开发史在别处发生

速查表二：术语对照

左边是你在文档和代码里会撞见的词，中间是界面上真正印着的中文，右边是一句人话。没有中文界面词的地方我留空，那说明这个词只在文档里活着。

英文	界面上的中文	一句人话
Harness	—	套在模型外面那层壳：工具、权限、记忆、日志都归它管
Cordis	—	底座框架，管一个零件怎么装上去、怎么拔下来
plugin	插件	可装可拔的零件，连主循环本身也是一个
agent preset	模式	一份「这个助手由哪些零件组成」的清单文件
profile	—	整套运行形态：网页版还是一次性跑完就退
workspace	工作区	你在它登记簿上注册过的项目目录，不是你开终端的那个目录
session	会话	一次对话，落盘成一个压缩日志文件
Trajectory	轨迹	把一次运行拆开给你看：它看到了什么、花了多少钱
turn / step / round	回合 / 步 / 轮	一次把话说完 / 其中的一个来回 / 外层策略的一遍
todo	待办	便利贴。整张表替换，没有增量更新
plan	计划	施工图纸，要你点头才开工
goal	目标	立项书。有编号、有版本，改它要你批
workflow	工作流	一次性的调度脚本，模型现写现跑，用来把同一件事扇给好几个子agent
schedule	定时	闹钟。代码装上了，出厂配置树里零命中，要自己加八行接线——那八行我没实跑验证过
jobs	后台任务	总控台。后台命令、终端、子agent共用一套
surface event	—	模型看得见的那3种事件，其余41种只进日志
spill	—	单条结果太大就存成文件，模型只看到摘要和路径，想要还能取回
compaction	压缩历史	调一次模型把旧对话缩成摘要，原文从此对模型消失
tool result pruner	—	按字数硬切工具结果的中段，不调模型，切掉就没了

英文	界面上的中文	一句人话
KV cache	—	请求开头一个字不改，就能复用上次的算力；改一个字，后面全部重算
Code mode	PTC模式	不再一件件点工具，改成写一小段程序一次跑完
skill	技能	给AI的操作手册，一个文件夹配一份说明
MCP	—	它去外面接工具，它当客户端
ACP	—	外面的程序来驱动它，它当服务端。方向和上一条相反
approval	审批	要不要在门口叫一声保安，只有「问」和「不问」两个值。出厂配置里「不问」只跟下一行那个全楼通行的档位配在一起，那一档本来就没有需要举手的操作
sandbox mode	权限	只读参观 / 只能改工作区 / 全楼通行，三档
fail closed / fail loud	—	拿不到「允许」就算拒绝 / 做不到就当场喊，绝不假装做了
isolate / realm	—	一组零件私有一份服务，跟外面那份互不串门
standing generation	—	一份模式清单每个进程只挂一次；你改一次存一次，就多留一代直到重启
erasable syntax	—	只接受「删掉类型标注就是合法JavaScript」的写法
Landlock / Seatbelt	—	向操作系统交出权限，交出去自己也收不回来
Agent Note	—	他们要求AI每做完一件事就写的工作日志

速查表三：命令速查

这张表我给自己加了一条约束：**没有亲手敲过的命令，不写进来。**所以它比官方文档短很多，但每一条都出过结果。

你想干什么	敲什么	注意
装并起网页版	`npx @deepseek-ai/dsh web`	第一次要下几百MB，而且没有进度条。有人在Windows上等到以为它死了
一次性跑完就退	`npx @deepseek-ai/dsh --profile headless "你的任务"`	本书大部分实测走的都是这条。它没有斜杠命令，也没有审批弹窗
换一套运行形态	`--profile web` / `--profile headless`	四选一的模式菜单只在网页版里存在，无界面模式直接按标准那份走
换个家目录跑，不碰已有会话	`DSH_HOME=/tmp/dsh-test npx @deepseek-ai/dsh ...`	试错前先做这一步。所有数据都会落在这个新家里
导出出厂的整棵配置树	`npx @deepseek-ai/dsh --profile web --dump-default-config > tree.yml`	出厂长什么样看这个；你自己改过之后长什么样，把命令换成 `--dump-config`
临时挂一段自己的配置	`npx @deepseek-ai/dsh --profile headless --patch my.yml "任务"`	装了包不等于挂上了，绝大多数额外的包都得靠这一步才生效
装一个额外的包	`npx @deepseek-ai/dsh plugin --profile web add @deepseek-ai/dsh-hooks-claude-code@next`	**结尾那个后缀不能省。**省了会装到三天前的旧版，接口对不上。另外装了不等于挂上了，挂载还得靠上一条
看会话日志原文	`zstd -dc ~/.dsh/sessions/*/session-*/session.jsonl.zstd | head`	先确认它真是压缩过的：`xxd -l 16` 应该看到 `28b5 2ffd`
找会话存在哪	`ls ~/.dsh/sessions/`	目录名是项目路径压平来的，中文会转成码点。界面上没有「删除会话」，要删只能来这里删目录
查某个包的版本标签	`npm view @deepseek-ai/dsh-base dist-tags --json`	上一条的坑就靠它验。哪天两个标签指向同一版，那个坑就没了
算它占了多少磁盘	`du -sh ~/.dsh` 和 `du -sh ~/.npm/_npx/*/`	大头从来不在前者。用不同写法起过它，缓存就有好几份

密钥这块还有两个坑。一是密钥文件的权限必须是仅自己可读，否则整个进程根本起不来，好在报错里会把 `chmod` 那条修复命令原样给你，照抄就行。二是缺密钥、密钥里混进换行、密钥无效，这三种情况的报错并不一样，别一律当成「没配上」。

速查表四：那35个要单独装的包

这35个包是全书最容易写错的地方，所以口径先摆在前面。

> **这35个包都在npm上，我逐个查过，全部正常返回。**它们只是不在`npx @deepseek-ai/dsh`的默认安装闭包里，装主包不会连带装上，要单独敲一条。
>
> 但单独装有个坑：除了主包，其余包的「最新版」标签还停在三天前的旧版本上。不加`@next`就会装到旧的，跟你机器上的新版对不上。

下面按用途分成两张表。前一张是真能力，后一张是他们自己跑测试和演示用的基建。

真能力	是什么，谁会想要
子代理provider（4个）	把一个回合外包给隔壁的Claude Code或Codex。代价是接Claude那条要连带下载整套Anthropic的SDK，这是这批包里最沉的一个
hooks桥（3个）	让你在Claude Code或Codex里写的钩子配置在这边继续生效。但两边工具名的大小写不一样，照搬过来最可能的结果是「装好了、看着在跑、一条都没生效」
ACP（3个）	让外部程序通过标准协议驱动它。想把它接进编辑器的人会找这个，但官方自述是自动化专用，编辑器要的那些能力它不提供
Python SDK与远程调用（5个）	用程序而不是用嘴驱动它。要把它嵌进自己产品里的团队看这一组
语言服务（3个）	给模型接上代码补全和跳转那套服务。写大型代码库的人会想要
云端沙箱（3个）	把「实际动文件、实际跑命令」这部分整个搬到租来的云上Linux里，脑子还留在本机。仓库自己称它是概念验证
SQLite后端（2个）	把会话换成数据库存，外加让模型自己检索历史日志。要做审计和多人协作的会关心
换搜索和抓取（3个）	把自带的搜索换成别家的，或者加上直接抓网页的能力
其他（3个）	一个真正的交互式终端（跟§08里极简模式那个持久bash不是一回事，那个在默认安装里）、一个让模型查自己历史的工具、一种不同的会话标题策略

测试与演示基建	是什么，谁会想要
录制回放与假模型（2个）	把模型的回答录下来重放，或者干脆起一个假的模型服务。做自动化测试的人才用得上，日常用不到
循环测试工具箱（1个）	他们自己测主循环用的
骨架演示与冒烟检查（3个）	最小可跑示例和加载器的自检。想照着学怎么写插件的人可以当范本读

这里有一处口径要说明白：**「哪些算测试基建」有两种数法。**按仓库自己的功能分组，明确的测试演示基建是6个；把名字里带demo和snapshot的另外三个也算进来，就是9个。上面这两张表用的是6个那种，两种数法加起来都是35，不影响总数。

这本书的保质期

你手里这本书，拆的是一个开源当天的产品。仓库仍在高速迭代，我取快照的那一天，npm上的版本走了四个。所以它的正确用法不是「照着抄」，是「照着核」。

我已经知道至少三处文档和代码对不上的地方，全部写在这里，因为它们随时会被修好。

其一，一个工具改了名，旧名字还留在发行包里。那套让它读写自己运行时的工具已经换成了新命名，但旧名字仍然写在随包发布的一份技能说明里。这份说明不是给人看的，是喂给模型看的，所以它比「文档里有个过期名词」严重一档。

其二，同一批工具，说明书写5个，实际是7个。介绍这套自指工具的那份说明写着5个，我在生成的工具目录里数到7个。

其三，那9个被整块搬进仓库的外来包，文档和事实反着说。有三处文档写着这些包被标记为「不发布」，实测那个标记字段压根不存在；而且我逐个查过，这9个**全部**真的发到npm上了，所以「这类包从不发布」这句话本身也已经不成立。

怎么复核，用的还是速查表三里那几条命令。先跑一次导出出厂配置树，跟书里写的对一眼，这一步不需要密钥，一分钟就有结果。再把速查表一第三列的命令挨个敲一遍，十分钟顶天了。至于网络那一组的数字，别核了，直接当过期的重取。

如果你跑出来的和书里不一样，优先信你自己那一次。**取数命令我全写上了，本来就是留着给你推翻我用的。**

DeepSeek Harness：从开机到拆开

AI编程：从入门到精通

花叔 · AI Native Coder

我一行代码都不会写，却用AI做出了App Store付费榜Top 1的小猫补光灯，写了9本技术书。

所有产品，全部AI写的，我只负责想清楚要做什么。

开源了女娲.skill、huashu-design等项目。

我始终相信：**AI生成内容不稀缺，稀缺的是判断力**。

B站：花叔v · 公众号：花叔 · X/Twitter · YouTube · GitHub · 官网

Created by 花叔 · v260813 · 2026年8月

本书内容基于DeepSeek Harness v0.1开发者预览版的本机实测，该项目仍在高速迭代。