废才俱乐部 · FEICAI CLUB

DeepSeek Harness 白皮书

一切皆插件：从装上它，到看懂它，到自己扩展它

版本
第一版 · 2026-08

研究基线
DEEPSEEK-HARNESS
0.1.0-RC.5

篇幅
131 页

阅读方式
分层可读

00

阅读地图

怎么排的

按读者依次会产生的问题排，不按代码仓库的目录排。

怎么读

每页标题就是那页的判断；正文页末尾那条带线的横条，是这一页可以直接带走的东西（地图页、清单页和附录没有）。

这本书按你依次会产生的问题排，不按代码目录排。

七个部分，三十八章。每一部只回答一个问题，答完就把你交给下一部。你不必从头读到尾，但顺序是有理由的。

部	你在这一部要回答的问题	章
一 · 认识它	这东西到底是什么，值不值得我花时间	01-05
二 · 装上它，用起来	怎么在我自己的机器上跑起来、配上模型	06-10
三 · 它由什么组成	我看到的这两百多个包，是按什么分的	11-14
四 · 走完它的全程	我说一句话之后，里面究竟发生了什么	15-20
五 · 它的五个特点	它跟别的 harness 的差别落在哪几处	21-25
六 · 插件	我要自己扩展它，从哪一行开始写	26-35
七 · 判断	这一轮我到底该不该用它	36-38

你是哪一种	只读这些就够	篇幅
只想用它	01 → 04 → 06-10 → 38	约 32 页
要写插件	03 → 04 → 11 → 13 → 26-35	约 36 页
要做选型	一页摘要 → 05 → 第五部代价页 → 36-38	约 23 页

时间只够读两部，就读第一部和第七部：一头告诉你它是什么，一头告诉你要不要碰。

00

一页摘要

基线

仓库源码 0.1.0-rc.5，npm 上的 latest 是 0.1.0-rc.6。

数字口径

前三个数字来自仓库本身；第四个是 2026-08-14 的 GitHub API 快照。都不是宣传材料。

只读一页的话：它卖的不是「更好用」，是「可替换」。

harness 是让模型能读文件、跑命令、连着干活的那层外围程序；模型本身只会文字进文字出。DeepSeek 在 2026-08-13 用 MIT 协议开源了自己的这一层，叫 dsh。

219
个包，49 个目录组

26
条能力接缝，可整块替换

52
个内置工具，24 个工具包

0
公开仓库 issue 与 PR

优点	任何一层都能换：模型、文件系统、沙箱、压缩策略；连主循环、工具表、会话日志也只是配置里的一行——能摘能换，但没有契约接住你：它是「一行配置」，不是「一个接缝」
优点	模型看见的每个字都写进只追加的日志，完整、可重放、不加密
优点	质量门禁密度罕见：37 条 verify 脚本、逐文件覆盖率、文档陈旧会让 CI 变红。代价是它只服务内部：拦不住加载路径上的错误（33.1），对 fork 是长亮红灯（25.3）
代价	概念门槛高一个量级；改配置的值不用重启，但增删插件本身要重启，浏览器侧尤其如此
代价	插件与主程序同进程、无隔离：装一个插件等于交出全部权限
代价	官方明说会有破坏兼容的变更；不接受外部 PR，也没有 Issues 入口

一句话判断

要一个能干活的终端助手，它不是最优解；要一个能被你改造成自己形状的底座，目前没有更彻底的。

I

这东西到底是什么，值不值得我花时间

PART ONE

认识它

先不装、先不改。把它是什么、跟别人差在哪、现在处在什么阶段，一次说清楚。

1.1

任何 harness 都要干的五件事

任何一个 harness，都在干同样的五件事。

模型只会文字进、文字出：读不了文件，按不下回车，也记不住上一轮。把这三样接上、让循环转起来的那层程序，就是 harness。

官方口径

DeepSeek 官网把这件事写成一个等式：Agent = Model + Harness。

译名

harness 直译是「挽具」，套在马身上让它能拉车的那套东西。

对应关系

第四部会带着一句话，走完这五件事的全程。

01 **组装这一次该说的话**

把身份设定、工具说明、历史对话拼成一次请求，顺序直接决定表现。

02 **把模型的话变成动作**

模型说「调用 bash，命令是 ls」，得有人真去执行，再把输出交回去。

03 **把发生过的事记下来**

说了什么、调了什么、返回了什么全落进日志，下一轮和复盘都靠它。

04 **决定什么时候停**

再问模型一次，还是收工？判断错了不是半途而废，就是原地打转烧钱。

05 **危险动作前问一句**

删文件、装依赖之前先让人点头。没有这一层，就不敢让它碰真实项目。

记忆方法

组装 → 执行 → 记录 → 决定停不停 → 该问就问。五件事，少一件就不是 harness，只是一个聊天框。

2.1

判定标准只有一条

它不跟谁比更好用，它把 harness 本身做成了零件。

判断一个东西是不是真插件化，只有一条标准：产品自带的功能，本身是不是也是一行普通配置。

SOURCE

docs/architecture.md:13

SOURCE

apps/cli/src/profile-boot.ts:60

注意

「插件化」几乎每个 agent 工具都写在首页，值不值钱要看这一条。

能被一个 id 定位、能被一行 `disabled: true` 关掉、能被别的包顶替——三条都满足才叫零件；否则「插件」只是厂商在固定内核旁边留给你的几个挂钩。

dsh 满足。每个组合的根配置文件，字面上就是一个空数组，整个产品是打在这个空数组上的若干层补丁：

```
PROFILE 的根配置文件，每次启动都被重写成这样

# dsh profile root — an empty entry list. The tree is composed as
patches:
# each bundle in package.json's dsh.profile.bundles, then
cordis.patch.yml, then any
# --patch overlays. Edit cordis.patch.yml, not this file.
[]
```

> 不存在需要打补丁的特权内核：扩展 dsh 的方式是把插件挂载到其他插件旁边，而各项注册都是副作用，会在其插件卸载时撤销。
>
> DOCS/ARCHITECTURE.ZH.MD

一句话判断

看一个 harness 能不能删掉它自带的功能。删不掉的，你能扩展的永远只是厂商划给你的那几个位置。

2.2

官方自己关掉自带功能的证据

SOURCE

packages/bundle/web-app/cordis.patch.yml

数字

这一个文件里出现了 24 次 `disabled: true`，全部指向 base 层的行。

官方自己在走「关掉自带功能」这条通路，一次关了 24 行。

这不是理论上可行。你跑 `dsh web` 时，官方的 web 组合第一件事就是把基础层里二十多个功能整块摘掉。

dsh 的组合是分层叠加的：先铺一层通用的基础配置（base），再叠一层针对 Web 界面的配置（web-app）。而 web-app 这一层干的事，主要是关行——

被关掉的行	本来是什么
`tool-bash / tool-fs / tool-fs-search`	执行命令、读写文件、搜索代码这三组工具
`plan-mode / tool-todo / tool-goal`	计划模式与待办、目标状态的维护
`compaction-basic / tool-result-pruner`	上下文压缩与工具结果裁剪
`tool-subagent / tool-workflow / tool-ralph`	派子代理、跑工作流、循环重试

关掉之后不是没了，而是交给下一层：每开一个会话，再按这个会话选定的组合把需要的行装回来。同一个进程因此可以同时跑一个功能齐全的 agent 和一个只有两把工具的极简 agent。

常见误解

「关掉」不等于「删掉」。这里用 `disabled` 而不是删除是刻意的——base 是所有组合共享的层，删掉的行会在某人重排组合的那天悄悄复活。

3.1

六个词，读完全书够用

六个词就够撑住全书。

前四个回答同一件事：能力是谁提供的；后两个讲事情记在哪、谁去干。各花十秒钟对上号。

为什么只有 6 个

官方术语表条目远不止这些，但读完全书只需要这 6 个。

两个操作词

「配置组合」在 10.2 讲，「会话预设」在 8.2 讲，用到时再学。

完整版

附录 A 是全书术语表，需要时再查。

harness 外壳 / 挽具	让模型能读文件、跑命令、连着干活的那层外围程序。`dsh` 就是一个 harness。
plugin 插件	一个导出 `apply` 函数的模块，外加配置清单里的一行。产品的每一部分都是插件，包括主循环。
capability seam 能力接缝	一处可整块替换的能力，由声明接口、提供实现、使用它的三方构成。全仓 26 条。
tool 工具	模型能直接点名调用的函数，有名字和参数表，例如 `bash`、`read`。内置 52 个。
session log 会话日志	只能往后追加、不能改写的事件流，共 44 种事件类型。模型看见的每个字都必须在里面。
subagent 子代理	主 agent 派出去干一件独立子任务的另一个 agent，有自己的会话，权限只会更窄。

记忆方法

harness 是外壳，插件是零件，接缝是插座，工具是模型能按的按钮；按钮按下去发生的一切写进会话日志，忙不过来就派子代理。

4.1

三个词讲完整个系统

整个系统只有三样东西：一个框架、一堆插件、一份清单。

再复杂的细节都能塞进这三个词里。读不下去的时候回到这一页。

这是骨架页

后面所有章节都挂在这三个词的某一个上。

顺序有意义

框架最稳定，插件最多，清单最常改。

去哪读细节

框架看第 11 章，插件看第六部，清单看第 13 章。

一个框架

负责让插件挂上去、找到彼此、卸载时干净地撤回来。它不含任何产品语义——没有 agent，没有会话，没有工具。

一堆插件

产品的全部功能都在这里，219 个包。模型适配器、工具表、会话日志、主循环，一个不例外。

一份清单

一份 YAML，逐行写明这次启动装哪些插件、每个插件怎么配。改产品形态就是改这份清单。

三者的关系是：**框架不知道产品是什么，插件不知道自己会被谁组合，清单才是那个做决定的人**。这句话解释了本书后面几乎所有的设计选择。

三个词各自的来历留给后面：框架叫 Cordis，不是 DeepSeek 为这次发明的（附录 A），它只干三件事（第 11 章）；219 个包按什么分组见第 12 章；那份清单不是写出来的、是若干层补丁叠出来的（第 13 章），`--dump-config` 能把叠完的结果连同来源注释一起打印出来（10.5）。

一句话判断

以后遇到任何一个陌生名词，先问它属于哪一类：是框架的能力，是某一个插件，还是清单里的一行。三选一，很少落空。

5.1

时间、版本、社区、组织

star 很多，issue 和 PR 是 0——开发现场不在这个仓库里。

这几个数字放在一起，比任何一段评价都更能说明它现在是什么。

快照时间

下表 star、issue、PR 三项取自 2026-08-14 的 GitHub API，此后必然变化。

版本口径

本书研究基线是仓库源码 0.1.0-rc.5；npm 上的 latest 当时是 0.1.0-rc.6。

信号	值	怎么读
开源时间	2026-08-13	与 V4-Pro 正式上线、API 调价同一天
许可	MIT	作者在 HN 澄清过：永久，不会改
版本	0.1.0-rc	全部是预发布，没有一个正式 release
Star	87,843	开源约 25 小时后的快照
Issue / PR	0 / 0	README 与 CONTRIBUTING 把反馈引向 Discussions 和 Discord，没给 Issues 入口
发布物	无	没有 tag、没有 release、没有 CHANGELOG

还有一条更有信息量：公开仓库最近一次合并来自另一个组织的 2519 号 PR，而这里的 PR 计数是 0。**公开仓库更像一份单向发布的镜像**。推下去有三层意思：你读到的是内部主干在某个时点的快照；这两条线此后可能已经分叉，你看不到分在哪；社区拿到的任何东西都永远滞后一截。

常见误解

把八万多个 star 读成社区活跃。它是一个厂商单向发布的项目，不是一个已经在运转的社区——你提的问题，短期内不会有人回。

5.2

风险预警，前置到这里

现在就把它写进生产，这四件事要你自己扛。

这不是免责声明，是决策依据。四条里踩中任意两条，这一轮就先别上。

为什么放这么前

这四条会直接改变你要不要往下读、以什么姿势读。

推测

有中文媒体实测同款模型接别的框架，token 消耗只有它的三成多。单一来源，我们没有复现，不作为判断依据，但值得你自己测一遍。

后续

第七部第 37 章会把每一条展开成具体的应对。

01 **接口会断，官方明说**

README 原话是「THERE WILL BE COMPATIBILITY-BREAKING CHANGES」，开发约定里写得更直白：正式版之前宁可重命名重组，也不做兼容层。

02 **磁盘格式不迁移**

会话格式版本号停在 0，不作兼容承诺，后端遇到旧格式直接拒绝而不是升级。

03 **你改不动它**

贡献指南写明目前不接受外部 pull request，反馈只引向 Discussions 和 Discord。发现缺陷只能自己在本地打补丁，或者等。

04 **插件没有隔离**

插件跑在主进程、拥有你的全部权限，安装就是一次无审核的 `pnpm add`。

先别做

这一轮不要把它接进有真实用户的系统，也不要写依赖它磁盘格式的外部工具。先在一个可以随时丢掉的项目里用它。

II

五章都是操作，每页结尾告
诉你怎么算做对了

PART TWO

装上它，用起来

这一部里的每条命令都能直接敲。装、配、用、接脚本、找配置文件放在哪——按顺序走完，你会有一个能干活的 agent 和一份自己能改的配置。

6.1

动手之前

四样东西你多半已经有了，key 可以等界面跑起来再补

五项清单：前四项决定装不装得上，key 那项可以留到最后。

来源

README.md · package.json engines

补充

没有数据库，没有守护进程，不用注册账号。它就是一个跑在你机器上的 Node 程序。

WINDOWS

base 直接禁掉 tool-bash 换 tool-pwsh，沙箱只有部分强度，Python SDK 不支持。

Node.js 运行时	版本必须落在 `^22.19.0 || >=24.0.0` 里，这是仓库写死的引擎范围。
pnpm 包管理器	两种情况才需要：从源码构建，或者用 `dsh plugin` 装插件。它必须在 PATH 上。
git 只有克隆才用	从源码跑、或者装 git 托管的插件时需要。走 npx 那条路用不到。
一个项目目录 就是工作区	你在哪个目录敲 `dsh`，那个目录就是默认工作区根。
一个模型 key 装完再配	不用提前准备。先把界面开起来，再到设置页里填，保存后下一次请求就生效。

一句话判断

只是想用，走 npx；想读代码或写插件，才值得把仓库克隆下来。

6.2

最快的那条路

来源

README.md#run

补充

这条路不落地任何仓库代码，只在你的 npm 缓存里留一份包。

最快的一条路是一行命令，不用克隆仓库

装好 Node 之后，从敲下命令到看见界面，中间没有别的步骤。

01 **确认 Node 版本**

`node -v` 打印的版本必须落在 22.19.0 ~ 22.x 之间，或者 24.0.0 及以上；23.x 不在支持范围内。不满足就先升级，后面每一步都白搭。

02 **在你的项目目录里启动 Web 界面**

成功的标志是终端打印出一行访问地址，并且进程停在那里不退出。

03 **用浏览器打开它**

默认是 `http://127.0.0.1:3080`。页面能加载出来，就说明这一步成了。

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

在代理后面的话

进程只继承启动环境。要让 Node 认 `HTTP_PROXY` / `HTTPS_PROXY`，启动时得带上 `NODE_USE_ENV_PROXY=1`。npx 拉包走的是 npm 自己的代理设置，两回事。

成功标准

三条同时成立才算装好：终端打印出访问地址、浏览器能打开页面、按一次 `Ctrl+C` 进程能干净退出。

6.3

dsh 从哪来

本书后面每条 `dsh` 命令都假设它在 PATH 上，而 npx 不会把它装进去

`dsh` 是 `@deepseek-ai/dsh` 这个包声明的可执行文件。它怎么进到你的终端，决定了后面几章你要不要每次都加前缀。

来源

`apps/cli/package.json` 的 `bin` 字段

版本

npm 上 latest 标签是 0.1.0-rc.6，仓库源码是 0.1.0-rc.5，本书正文按后者写。

怎么调用它	代价
`npx @deepseek-ai/dsh …`	不落地任何东西，随用随取。每条命令都得带这个前缀
`npm i -g @deepseek-ai/dsh`	把 `dsh` 装进 PATH，之后裸命令直接能敲。升级要你自己管
`pnpm dsh …`	只在源码检出的仓库根目录里可用，不进 PATH

本书从第 9 章起写的都是裸 `dsh`。没有全局装的话，把每条命令前面补上 `npx @deepseek-ai/dsh` 就是等价写法——`dsh web` 即 `npx @deepseek-ai/dsh web`。

长期用别走 NPX

组合目录旁边那份软链每次启动都按「当次跑的这个安装」重建。npx 拉到新版本，软链跟着换指向，插件解析到的就变成另一份代码。要么全局装，要么把版本钉死：`npx @deepseek-ai/dsh@0.1.0-rc.6 web`。

记忆方法

三种调用方式：npx 不落地、全局装进 PATH、`pnpm dsh` 只在仓库里。敲一次 `dsh --help` 能出启动器的帮助，就是第二种。

6.4

从源码跑

来源

apps/cli/reference/README.md 的 Source execution 一节

适合谁

要读代码、改代码、或者开发插件的人。只想用它干活的人不必走这条。

从源码跑要多两步，`pnpm run build` 是不能省的那步

`pnpm dsh` 直接跑 TypeScript 入口，它不构建，也不检查产物新不新。

```
从仓库检出跑起来

git clone https://github.com/deepseek-ai/deepseek-harness.git
cd deepseek-harness

# 装依赖，然后构建产物（第一次必须做）
pnpm install
pnpm run build

# 启动，参数原样转发给 dsh
pnpm dsh web
```

漏了构建，boot 阶段直接报模块解析错误，不会提示你去补。

会静默出错

启动器不检查产物新鲜度。改完前端忘了重新构建，浏览器照样打开，跑的却是旧代码，终端一切正常。

成功标准

浏览器里看到的界面，和你刚改过的源码对得上。对不上就是漏了 `pnpm run build`。

6.5

第一次启动

术语

PROFILE

（组合）：一份有序的插件层清单，决定这次启动装出一棵什么样的树。

来源

packages/boot/app-boot/README.md

第一次启动，它替你建好了一个组合，你什么都不用配

自动发生的只有三件事：前两件在你的家目录里，第三件在你敲命令的那个目录里。

01 建组合目录

web 和 headless 两个组合在首次使用时从随包模板自动初始化。别的名字不会自动建，会明确报错并提示你用 `dsh plugin` 创建。

02 修软链

每次启动都重建 `$DSH_HOME/profiles/node_modules` 里的扁平软链，让组合里写裸包名也能解析到安装自带的那份。

03 认工作区并读说明

调用目录成为默认工作区根，该目录下适用的 `AGENTS.md` 或 `CLAUDE.md` 会被读进上下文，渲染预算 65,536 字节。

写错插件名不会被忽略

组合里有插件解析不到，启动直接失败退出，并把解析不到的插件名列出来——它不会半装出一棵树来骗你。真正安静的是启动之后再动态挂载的那些插件。

先别做

别去手改 `$DSH_HOME/profiles/<名字>/cordis.yml`——它每次启动都被覆写成空列表。你要改的是同目录下的 `cordis.patch.yml`。

7.1

配 DeepSeek

来源

docs/user/guide/providers.md

补充

整个配置过程不碰命令行，也不碰配置文件。

配 key 不用重启也不用改文件，保存后下一次请求就生效

不用重启服务，不用重开会话，不用改任何文件。

01 **打开 Settings → Models**

页面上第一张就是 DeepSeek 卡片，它只暴露一个 API key 字段。

02 **填进去，保存**

模型路线立刻可用。改动在下一次请求时生效，服务器不需要重启。

03 **在模型选择器里选一个模型**

选中它，它同时成为以后新建会话的默认模型。已经发过请求的会话保留自己日志里记下的那一个，不受影响。

会卡在这里

如果保存的默认模型指向一个已被删除的厂商，输入框会显示 `Select model` 并一直锁着，直到你另选一个。

成功标准

模型选择器里能选到 DeepSeek 的模型，并且发出一条消息不报 `MISSING_CREDENTIAL`。

7.2

key 存在哪

来源

packages/credentials/credentials-local/README.md

脱敏描述符

一段只够界面显示"这里有个 key"的元信息，不含明文。

key 存进去就拿不回来了，这是设计，不是限制

写入是单向的：页面能提交，不能读回。

保存之后，浏览器收到的只是一个脱敏描述符，永远拿不到明文。密钥落在 `$DSH_HOME/.credentials.yaml`，是 `0700` 目录下的 `0600` 文件；设置文件里只留一个指向它的凭据引用。

凭据按四层顺序解析

规则是**先命中的赢**——和第 13 章讲配置补丁时「越晚的层赢」正好相反。

1	继承的进程环境	启动时环境里已有的变量。永远最优先，而且只读
2	`$DSH_HOME/.credentials.yaml`	托管文档，Models 页面写的就是它
3	调用目录的 `.env`	项目级环境层
4	`$DSH_HOME/.env`	用户级环境层

第 1 层最高，正是 CI 和容器该走的那条：`DEEPSEEK_API_KEY=… dsh …`、CI secret、容器的 `-e` 都落在这里，不需要任何文件。它也因此是只读的。

常见误解

「key 放在模型读不到的地方」是错的。`0600` 挡的是别的系统用户，不是模型。真正守住的是两条：从不把这个文件的路径交给模型，从不把它装进 `process.env`。

7.3

接别的厂商

接别的厂商分两种：目录里已经有的，和你自己的网关

两个按钮，对应两条完全不同的路径。

来源

`docs/user/guide/providers.md`

目录

随安装带的一份厂商清单，包含端点、协议和模型列表，查它不发网络请求。

Add provider

目录内的厂商，比如 Anthropic 或 OpenAI。选中、填 key、保存，端点、协议、模型列表全由已安装的目录提供。

Add a custom provider

公司网关、自建服务、目录里没有的厂商。要填五项：小写 Provider ID、base URL、API 协议、凭据、至少一个模型。

只填 KEY 配不上

用原生认证的厂商需要各自的凭据：Bedrock 要 AWS 凭据加区域，Vertex 要 ADC 项目，Azure 要 `api-version`，Codex 走 OAuth。

先别做

先别随手起 Provider ID。它是永久的——请求、已存会话、模型默认、凭据引用都用它做键。改名只能新建一个再删旧的，其余字段倒是随时能改。

7.4

图片输入

你手工填的模型，默认按纯文本处理

给它附一张图，请求在发出去之前就被拒绝，并且点名是哪个模型。

为什么

没有任何办法去问一个 HTTP 端点它接受哪些模态，所以只能等你声明。

范围

只影响你手工填进去的模型；目录里的模型按目录记载走。

表单里没有这个字段。要让一个自定义厂商的视觉模型收图，得去 `$DSH_HOME/settings.yaml` 给那个模型补一行 `input`。本片段假设你已经在表单里建好了 `my-gateway` 这个厂商，这里只是补表单填不了的那一项。

```
$DSH_HOME/SETTINGS.YAML

llm-pi-ai:
  providers:
    my-gateway:
      apiKeyEnv: GATEWAY_API_KEY
      api: openai-completions
      baseURL: https://gateway.example/v1
      models:
        - id: legacy-chat
        - id: vision-preview
          input: [text, image]
```

`apiKeyEnv` 是按请求解析的凭据引用，机密本身不进这个文件。写在路线上的 `defaultInput` 是回退而不是覆盖，默认 `[text]`。目录内的厂商没有 `models` 列表可写，要收窄某个模型的模态得写在 `modelOverrides` 下面、按模型 id 作键。DeepSeek 自己的 chat-completions 路线本身就是纯文本，配不出图片输入。

常见误解

`input` 是一个声明，不是一次校验。你写了端点其实不支持的模态，这里不拦；请求发出去，由厂商替你拒绝。

8.1

选工作区

不选工作区，会话输入框就是灰的

全新的 Web 界面没有已选工作区。这一步没人替你做。

来源

docs/user/guide/index.md

补充

进程知道自己从哪个目录启动，但那只是默认文件系统位置，不等于你授权它动那个目录。

01 **点 Choose workspace**

入口在界面里始终可见，因为在选定之前你什么也开始不了。

02 **把你启动 dsh 的那个项目目录加进去**

加完之后还要选中它——添加和选中是两个动作。

`dsh` 进程当然知道自己从哪个目录启动，那个目录也确实是默认的文件系统位置。但它只是一个默认值，不等于你授权 agent 去动它。所以界面把「选定」做成一个你必须亲手完成的动作。

为什么单列一步

工作区是后面所有权限判断的原点：沙箱把 bash 和文件系统的写入限制在它（加上平台临时目录）里面；读取、网络访问、进程可见性都不受这一层约束。没有它，权限规则无从谈起。

成功标准

会话输入区能打字了，就说明工作区选好了。在此之前它一直不可用，不是界面坏了。

8.2

开一个会话

开一个会话，你面对的是一个会动手的 agent，不是聊天框

官方指南给的第一句话就是让它读代码库：Summarize this repository and identify its main packages。

来源

docs/user/guide/index.md · apps/cli/reference/README.md

子代理

由主 agent 派出去、带独立上下文跑一段活的第二个 agent。

它在会话里能做四件事

读写工作区文件	范围就是你选定的那个目录
执行命令	走沙箱化的 bash
委派子代理	把能并行的活拆出去
维护一份计划	计划本身是被记进日志的状态

建会话时还可以挑一个**会话预设**：一份决定这个会话带哪些面向模型的插件的配方，随包的有四个。其中「极简模式」把系统提示固定成一句话，只带持久 bash 和 `str_replace_editor`，其余面向模型的插件在那个 agent 里全部不在场；而浏览器、工作区、持久化、沙箱这套宿主设施照旧。预设是写进会话日志的，重开旧会话不会换成另一份。

一句话判断

想看清它到底做了什么，先用极简模式跑一遍：变量少，日志短，因果关系一眼能对上。

8.3

审批与权限档位

弹窗问你，是因为这个会话的档位是「问」

界面上是一个选择器，实际上一次选中两件事：沙箱管到哪，以及需要审批时怎么办。

来源

packages/interaction/permission-presets/README.md

只讲看得见的

这三个档位背后是两个独立旋钮，机制见第 24 章。

档位	沙箱	审批
read-only	完全不许写	问你
workspace-write	写入限制在工作区内	问你（新会话默认）
danger-full-access	不再限制	不问，直接拒

三档里 `read-only` 适合「只让它读和分析、不许改任何文件」的场景。一次批准也只对被问的那一次操作生效。进程级的回退档位可以用环境变量 `DSH_PERMISSION_MODE` 改——写成 `danger-full-access` 会顺手把审批策略也关成 never。

改设置不影响开着的会话

权限在会话创建时就被钉进这个会话。之后你在设置里改默认，只对以后新建的会话有效。

常见误解

`danger-full-access` 不是「什么都允许」。它把审批策略关成 never——仍然需要审批的动作会被自动拒绝，既不弹窗，也不放行。

8.4

第一个真实任务

第一个任务要挑一件你能当场验证的改动，不是一句「帮我看看」

四步，十分钟以内。做完你手上会有一份 `git diff`，那才是「它真的在干活」的证据。

为什么是这个

「帮我看看这个项目」只能证明它会说话，证明不了它会动手。

别拿主仓库试

第一次跑，选一个你敢让它改、而且改坏了也不心疼的目录。

01 选一个有测试、且已经 commit 干净的目录

干净的工作树是这次验收的基线：跑完之后 `git status` 里出现的每一行都是它写的。

02 把任务说成一件可验证的事

例：「读一遍 `src/`，把 README 里过时的安装步骤改对，然后跑一次测试，把结果告诉我。」有对象、有动作、有验收方式。

03 只有它要越界时才会弹窗

工作区内的读写和跑命令都在档位允许范围内，不打断你。bash 要写到工作区外面，才会带着一句理由申请更宽的沙箱模式——你点同意，只对那一次调用生效。

04 自己核对，别信它的总结

`git diff` 看改了什么，再自己跑一遍测试。

成功标准

`git diff` 里能看到那处改动，你自己跑测试的结果和它说的一致。对不上，问题多半不在模型，在你的任务没有验收方式。

9.1

四种入口模式

来源

apps/cli/README.md

LAUNCHER

指 dsh 命令本身。它只管把层叠好、把树点起来，不认识任何应用功能。

命令行只有四种入口，做的其实是同一件事

选一个组合，把那棵插件树点起来。区别只在点起来之后谁接管。

命令	做什么
`dsh --profile <名字>`	启动指定组合
`dsh --profile headless "任务"`	跑完一次任务、打印结果、退出
`dsh web`	等价于 `--profile web` 的硬编码别名
`dsh plugin --profile <名字>`	管理该组合的插件，参数转发给 pnpm

这一章往后写的都是裸 `dsh`。没全局装的话，按 6.3 的写法在前面补 `npx @deepseek-ai/dsh`。

`dsh` 自己只认四个 flag——`--profile`、可重复的 `--patch`、`--dump-config`、`--dump-default-config`——从它不认识的第一个 token 起，后面全部原样交给启动的那个应用。所以 `dsh --profile web --port 8080` 里的 `--port` 属于 web 应用；`dsh --help` 打的是启动器的帮助，`dsh --profile web --help` 打的是 web 应用的帮助，而且什么都不会启动。

记忆方法

前面是启动器的，后面是应用的，分界线就是第一个启动器不认识的词。所以启动器的 flag 必须写在最前面。

9.2

headless 一次性执行

来源

packages/bundle/headless/README.md

安静

指没有待处理的工作了。

headless 跑完就退出，退出码告诉你成没成

最轻的一条脚本入口：只看退出码和 stdout。

```
$ dsh --profile headless "run the tests"
```

01 **建一个全新的持久化会话，不复用旧的**

02 **把任务当成一条普通用户消息提交**

没有特殊通道，走的就是界面里那条路。

03 **等到安静**

等完 agent 自己发起的每一轮，不是等第一次回复。

04 **把会话刷盘**

先落盘再读结果，打印的和存下的是同一份。

05 **取最后一条非空助手文本写 stdout**

一句话判断

退出码只有两种：`turn/end` 是 completed 就是 0，其他都是 1。成功的运行 stderr 全空，也不开任何端口。

9.3

headless 怎么拿到 key

headless 里没有设置页，key 只能从环境变量进去

这个组合不挂 Host、不挂 Web、不开端口。第 7 章那条填表路径在这里不存在。

来源

packages/bundle/base/cordis.patch.yml · credentials-local/README.md

桥在这儿

第 7 章配 key 的唯一路径是设置页，而 headless 根本没有页面。

```
$ DEEPSEEK_API_KEY=sk-… dsh --profile headless "把 README 的安装步骤核对一遍"
```

凭据四层里第 1 层是继承的进程环境，优先级最高（7.2）。CI 的 secret、容器的 `-e`、上面这种一次性前缀，全都落在这一层，不需要往磁盘上写任何文件。

模型路线用的是 base 层的出厂默认：厂商 `deepseek-official`、模型 `deepseek-v4-flash`。要换成别的，去 `$DSH_HOME/cordis.patch.yml`（对这台机器上所有组合生效）或者组合自己的 `cordis.patch.yml` 里，patch `agent-default-model` 那一行的 `provider` 与 `model`。

同一台机器上两条路是通的

你在 Web 界面里存过的 key 落在 `$DSH_HOME/.credentials.yaml`，headless 同样读得到（第 2 层）。只有跨机器——CI、容器、别人的构建机——才必须走环境变量。

成功标准

命令跑完 `echo $?` 是 `0`、stdout 有一段助手文本、stderr 全空。报 `MISSING_CREDENTIAL` 就是这四层一层都没命中。

9.4

web 的启停

web 只有三个自己的 flag，而且它故意不让你监听 0.0.0.0

默认服务在 `http://127.0.0.1:3080`，这是一个本机工具的默认值。

来源

`apps/cli/reference/README.md` 的 Web alias 一节

版本

以仓库 0.1.0-rc.5 为准。这是预发布版，行为可能变。

`--host`	换绑定地址，但填 `0.0.0.0` 会以用法错误退出
`--port`	换端口
`--trusted-host`	可重复，给 `/api` 的浏览器信任围栏加一个可接受的名字

「浏览器信任围栏」是 `/api` 那一侧的一道检查：只接受来自被认可名字的请求。`--trusted-host` 就是给这一次调用临时添一个名字。

关停有确定的语义

第一次 `SIGINT` 或 `SIGTERM` 开始一段最多五秒的优雅拆卸。`SIGTERM` 是监管进程的普通停止请求，退出码 0；`SIGINT` 退 130。第二次信号立刻强制退出。

先别做

别想用 `--host` 把它开给别的机器——填 `0.0.0.0` 会直接以用法错误退出，官方给的理由是「这会把远程代码执行暴露到网络上」。注意命令行拦得住、配置层拦不住：webserver 那一行的 host 只认两个值，改配置仍然绑得上。为什么最好别这么干，见 37.3。

9.5

headless 之外的三条

headless 不是唯一能接进程序的入口，只是最省事的那个

要在任务中途拿到事件、答审批、或者取消，就得换一条。

来源

docs/user/guide/python-sdk.md · examples/

为什么要知道

headless 一次只吃一个任务，也没有回合中途插话的余地。

入口	什么时候换它
`Python SDK`	官方唯一正式 SDK。自带同版本 runtime，机器上不需要装 Node；官方跑 benchmark 就走它
`examples/jsonrpc-agent`	用 JSON-RPC 驱动一个无人值守的编码 agent，逐条拿事件
`examples/acp-agent`	Agent Client Protocol 自动化服务，带会话、审批应答和取消

Python SDK 支持 Linux x64、Linux arm64 与 macOS 14 以上的 arm64，Windows 不在其中。它跑的是仓库里那个 `examples/jsonrpc-agent` 例子：给一个工作区、一个会话目录、一句任务，打印最终回复，同时在会话目录里留下一份 JSONL 日志。

一句话判断

「跑完给我结果」用 headless；「跑的过程中我要插手」——审批、取消、看事件——就得上 SDK 或那两个协议例子。

10.1

$DSH_HOME 里有什么

你自己的东西全在一个目录里

默认是 `~/.dsh`。知道里面有什么，就知道该去哪儿改、该备份什么。

路径

优先取显式配置，其次 `$DSH_HOME`，最后回落到 `~/.dsh`。

版本

会话日志格式版本停在 `0`。读到别的版本号它拒绝打开整个日志（第 23 章）。

迁移

整个目录拷过去就行，但凭据文档靠文件权限保护、不靠加密，别塞进仓库或同步盘。

路径	是什么
`.credentials.yaml`	托管的凭据文档。Models 页面写的就是它
`settings.yaml`	用户设置。只留凭据引用，不留明文
`cordis.patch.yml`	机器本地的组合覆盖，所有组合共享
`.env`	普通启动环境层，会进 `process.env`
`sessions/`	会话日志。对话、工具调用、改动记录全在这儿，体积会一直涨
`skills/`	用户级技能，怎么加见下面那一小节
`attachments/v1/objects/`	按内容寻址的图片附件，被会话日志引用
`profiles/<名字>/`	一个组合目录
`profiles/node_modules/`	每次启动自愈的扁平软链，别手动动

往 skills/ 加一个技能

新建 `$DSH_HOME/skills/<名字>/SKILL.md`，名字用 kebab-case，frontmatter 里 `name` 和 `description` 缺一项就被跳过。目录是被监听的，不用重启：回输入框敲 `/`，菜单里就有它。

10.2

profile 是什么

BUNDLE

（层）：一个 npm 包，在自己的 `package.json` 里声明它导出一份 patch 清单。

来源

`apps/cli/README.md` 的 Profiles 一节

profile 不是一份配置档案，是一个真正的 npm 项目目录

刚建好的时候里面有三个文件，外加一个每次启动都被重写的根锚点。

`package.json`	树外插件的依赖，加一份 `dsh.profile` 清单，里面的 `bundles` 是有序的层列表
`cordis.patch.yml`	你自己那一层。要改配置就改这个
`pnpm-workspace.yaml`	给 pnpm 的设置，让树外插件共用同一份 cordis
`cordis.yml`	每次启动被重写成空列表的根锚点。不用管，也别改
`node_modules/`	跑过一次 `dsh plugin` 之后才有，连同一份 lockfile

启动时这四层按顺序叠

1	清单里每个 bundle 的 patch，按清单顺序
2	这个组合自己的 `cordis.patch.yml`
3	`$DSH_HOME/cordis.patch.yml`
4	命令行上每个 `--patch` 覆盖，按 argv 顺序

常见误解

家目录那层排在组合那层之后，也就是优先级更高——和「越具体越优先」的直觉相反。它被定位成跨组合的机器本地偏好。

10.3

用 dsh plugin 装插件

装插件就是在这个目录里跑一次 pnpm

`dsh plugin` 把参数原样转发给 pnpm，工作目录设成那个组合的目录。`add`、`remove`、`why`、`update` 全都照常工作。

前提

pnpm 必须在 PATH 上。这条路径本身就是一层 pnpm 包装。

来源

`apps/cli/reference/README.md` 的 Plugin management 一节

```
$ dsh plugin --profile demo add ./hello-plugin
```

```
$ dsh --profile demo
```

相对路径按你敲命令的那个目录解析，不是按组合目录，所以在插件检出里 `add .` 装的就是这份检出。`demo` 这个名字没有随包模板，`dsh plugin` 会当场替你建出来，只叠 `@deepseek-ai/dsh-base` 一层。

每次成功之后，它会拿安装结果去核对那份 `bundles` 清单：凡是依赖包在自己的 `package.json` 里声明了 bundle patch 的，自动加进层列表；没声明的留成普通依赖，并给一次性的提醒；被移除的自动退出列表。

第一次 ADD 可能会失败

从 git 装、且需要构建的插件会跑 `prepare` 脚本，pnpm 10 以上默认拦下来。照它打印的提示把那个键写进组合目录的 `pnpm-workspace.yaml`，再跑一次。装现成的 tarball 或本地检出不需要这一步。

成功标准

装完之后 `dsh --profile <名字>` 能起来，并且你能在下一页说的 dump 输出里看到新插件贡献的那些行。

10.4

自查组合

来源

apps/cli/reference/README.md

补充

打印用的是和真实启动同一套折叠算法，所以「打印的树」不会和「跑的树」漂移。

改任何一行之前，先 `--dump-config` 看一眼它现在长什么样

两条命令，差别在于叠不叠你自己那几层。

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

```
$ dsh web --dump-config
```

前者只打 bundle 层，后者再叠上组合自己的 patch、家目录的 patch 和每个 `--patch` 覆盖。打出来的是一份仍然合法可加载的 YAML；没命中目标的 patch 报到 stderr。

DUMP 和真实启动不等价

`!!js` 表达式原样打印，不求值；dump 也不运行应用的命令行解析，所以带应用参数的 dump 会被直接拒绝，`--port` 的效果在这里看不见。

一句话判断

想知道现在到底跑的是什么，别去翻那几个配置文件，dump 一次就够——它用的是和真实启动同一套折叠算法。

10.5

读懂 dump 的来源注释

dump 的注释直接告诉你每一段出自谁、又被谁改过

这是排查「我改的那行到底生没生效」的全部依据。

来源

`packages/boot/app-boot` 的 `renderConfigDump`

怎么定位

行的身份是 `id`，不是包名——同一个包可以出现在多行里。

```
--DUMP-CONFIG 输出片段

# == @deepseek-ai/dsh-base
- id: agent
  name: '@deepseek-ai/dsh-agent'
# == @deepseek-ai/dsh-base, patched by ~/.dsh/cordis.patch.yml
- id: agent-default-model
  config:
    provider: deepseek-official
```

连续来自同一来源的行归成一段，段前那行注释先写它们出自哪个包或哪个文件，后半句 `patched by` 列出改过它们的每一层，按叠加顺序。上面这段的意思是：这两行都来自 base，其中第二行被你家目录那层动过。

反过来就是排查方法：你刚改过的那一行，如果段注释里没有 `patched by`，说明没有任何一层动过它——多半是 `id` 写错了（这种情况 dump 还会往 stderr 报一条没命中），或者那份 patch 文件这次启动根本没被叠上。

成功标准

你想改的那一行，能在 dump 输出里按 `id` 找到。找不到，就说明它藏在嵌套的 include 后面，patch 够不着。

10.6

先跑官方的示例覆盖层

来源

examples/README.md

覆盖层

一个 --patch 文件，只在这一次启动里叠上去，不改你的任何配置。

从「装好了」到「看懂 patch 是怎么回事」，最短的路是跑一个官方示例

仓库 `examples/` 下有六个能直接跑的例子，每个解决一个不同的问题。

```
$ dsh web --patch examples/web-schedule/cordis.yml
```

这一条跑完，界面上立刻多出三个工具（建定时提醒、列出、删除）。关掉进程再普通启动，它们就没了——这就是覆盖层：一次性的、不落地的一层 patch。

示例	它演示什么
web-schedule	给 Web 加一组会话内的定时提醒工具
web-cordis	让 agent 检查、修改自己那棵内存里的插件树
mcp-memory	三份接第三方记忆 MCP server 的现成配置
headless-agent	一次性任务，输出可换成机器可读格式
jsonrpc-agent	用 Python SDK 加 JSON-RPC 驱动无人值守 agent
acp-agent	Agent Client Protocol 自动化服务

记忆方法

覆盖层就是一次性的一层 patch：带 `--patch` 的那次启动多出新工具，去掉就没了，你自己的配置文件全程没被动过。

10.7

接你已有的 MCP server

接一个 MCP server 是加一行插件，不是装一个集成

出厂状态下一个 MCP server 都不装——每个 server 的命令都是沙箱之外的可信可执行代码，得由你自己点头。

来源

`examples/mcp-memory/README.md` · `packages/mcp/mcp-client`

好消息

模型看到的工具名是 `mcp__<server>__<tool>`，和 Claude Code、Codex 一致。

```
$DSH_HOME/CORDIS.PATCH.YML

- insert:
    - id: my-mcp
      name: '@deepseek-ai/dsh-mcp-client'
      config:
        serverName: my_server
        transport: stdio
        command: my-mcp-server
        args: [--stdio]
        env:
          MY_SERVER_TOKEN: …
```

三条前提：可执行文件要你自己先装好，dsh 只负责启动和停止它；stdio 的桥在启动子进程前会摘掉名字看起来像凭据的环境变量和所有 `DSH_*`，要传的密钥得显式写进这一行的 `env`；HTTP 传输则要求上游服务已经在跑。

成功标准

启动之后，模型的工具列表里出现 `mcp__my_server__*` 开头的名字。没出现就去看启动日志，多半是那个可执行文件不在 PATH 上。

III

屏幕上那一整套东西，到底
是由什么拼出来的？

PART THREE · 第三部

它是由什么组成的

你已经把它装上、跑起来了。现在拆开看零件。屏幕上那一整套东西，是三样东西拼出来的：一个只干三件事的框架、219 个各司其职的包、一份启动时现算出来的清单。这一部不讲它怎么转，只讲它由什么拼成、拼法是什么。

11.1

第一件事：按名字要能力

SOURCE

vendor/cordis/src/context.ts、reflect.ts

补充

ctx.fs 这种写法能成立，是因为 ctx 是个代理对象，读属性时才现去找实现。

名字只有一层

服务名是全局扁平的，产品自己已经占掉 56 个。自定义服务要加前缀。

要用一个能力，你只需要知道它叫什么名字，不需要知道谁在实现它。

一个插件被装上时，框架只递给它一个对象：`ctx`。这个对象同时是三样东西——服务台、清洁工、广播站。这一节只讲服务台。

服务台的用法只有一种：喊一个名字，拿到一个能干活的东西。一个插件要读文件，它写 `ctx.fs`。至于名字背后是本机磁盘、是沙箱、还是一台远端容器，它不知道，也不需要知道。

这像酒店前台：你不认识洗衣房的师傅，你只跟前台说「洗衣服」。哪天酒店换了一家洗衣公司，你要说的那句话一个字都不用改。

你喊的名字	它代表的能力	现在谁在应
ctx.fs	读写文件	dsh-fs-sandbox
ctx.shell	跑命令	dsh-bash-sandbox
ctx.llm	调模型	dsh-llm-deepseek

第三列是出厂组合里真正挂着的包。Windows 上第二行换成 `dsh-pwsh-sandbox`，靠一个平台判断二选一。

这样的名字一共 56 个，其中 26 个明确留给人换实现，本书叫它们**能力接缝**。

记忆方法

名字是契约，实现是配置。你在代码里写下的是名字，在 YAML 里选定的是实现。这两件事从来不在同一个文件里。

11.2

第二件事：注册即可撤销

SOURCE

vendor/cordis/src/fiber.ts（effect 与逆序回滚）

补充

撤销动作还能自动归属：谁读的服务，登记就挂在谁名下。

插件做的每一件事都自带撤销键，撤销按什么顺序按，框架自己知道。

`ctx` 的第二重身份是清洁工。这一件事决定了这套东西能不能真的热插拔。

一个插件装上之后会做很多事：登记一个工具、加一段系统提示词、开一个定时器、连一个数据库。换在别处，这些都要作者自己记住，并在卸载时反着来一遍。漏一个，就是一条幽灵监听器。

这里反过来。每一次登记，框架当场把「怎么撤销」收走，记在这个**插件实例**名下——一个插件在运行时的那一份实例，仓库里叫 `fiber`；它记着这个插件登记过什么，卸载时逆序撤回。作者从头到尾没写过一行清理代码。

```
框架不认识的资源，用这个逃生舱交给它

ctx.effect(() => {
  const timer = setInterval(poll, 1000)
  return () => clearInterval(timer)   // 这一行就是撤销键
})
```

同一条回滚路径服务三个场景：你改了 YAML 触发重装、依赖的服务下线了、进程要退出了。三种情况下插件的卸载过程一模一样。

一句话判断

能登记就必须能注销，而注销不是你的活。你唯一要自己写的，是那类框架不认识的资源上的那一行 `return`。

11.3

第三件事：站到中间去

SOURCE

packages/guard/timeout-policy/src/index.ts

补充

框架一共五种事件分发方式，只有拦截点这一种能站到中间。

事件不是发出去就不管的通知，是一条你可以站在中间的链。

`ctx` 的第三重身份是广播站——审批、沙箱、超时，全靠站到中间实现。

常见的事件系统是「我喊一声，谁想听谁听，我不等你」。这里最要紧的那种不是：监听器按登记顺序一层层套在真正的动作外面，像洋葱——每一层先做点事再往下交，最里面那层才是本来要发生的事。

于是链上每一层能做三件事：**放行**、**改写**、**拦下**。本书把这种能站在中间的事件叫**拦截点**（源码里的 `waterfall`），只能旁观的叫**广播点**，在 `ctx` 上添东西的那些方法叫**注册入口**。

一个完整插件：给所有工具加执行超时

```
export const inject = ['tools']
export function apply(ctx) {
  ctx.on('tools/execute', async (exec, next) => {
    if (这个工具没声明超时预算) return next()    // 放行
    const result = await next()
    return 计时器响了 ? 超时结果() : result    // 改写
  })
}
```

常见误解

以为只想旁观就可以不往下交。恰恰相反：只记录的监听器也必须交，忘了就会把下游连同默认行为一起静默吞掉。

12.1

三类包，49 个目录

SOURCE

packages/README.md、pnpm-workspace.yaml

补充

目录只决定谁跟谁是一伙的，不决定启动时装哪些。

219 个包分成三类：能力家族、产品脊柱、支撑设施。

一个包一件事，一个目录一类事。这不是文档整理出来的分类，是仓库结构本身。

219	49	39	56
包	目录组	其中界面包	服务名

这一类	目录举例	它们是什么
能力家族	shell fs llm sandbox compaction	一个能力的定义、若干实现、外加把它做成工具的那个包，全装在同一个目录里
产品脊柱	core session api host client	产品自身的功能：会话、提示词、工具表、界面。它们也全是插件
支撑设施	util boot bundle test-support	不面向模型，给上面两类用

上表没画进去的还有一格：`packages/README.md` 的总表给每个组标了「发布预期」，绝大多数写的是 stable API，只有远端沙箱那一组 `e2b/` 标着 POC。要拿哪个组当依赖，先翻那一格。

到这儿就够了

目录结构只是一张阅读地图。你的组合里装什么、不装什么，跟包放在哪个目录一点关系都没有——那是第 13 章那份清单的事。

12.2

包名就是它的角色

SOURCE

packages/shell/README.md、各组 README

补充

49 组里有 25 组带一个跟目录同名的包，也就是 25 个明摆着的插槽。

一个能力家族里，跟目录同名的那个包定义能力，旁边的那些实现它。

认得这条命名规则，你不看文档也能判断出一个包是干什么的。

PACKAGES/SHELL/ 里的包	角色	它干什么
dsh-shell	能力定义	只写清「跑命令」这件事长什么样，一行怎么跑都没有
dsh-bash-sandbox	实现	开子进程跑，但每次都关进沙箱。出厂组合挂的是这一个
dsh-bash-local	另一个实现	同样的机制，不进沙箱。仓库里有，出厂清单里没有
dsh-tool-bash	工具	把这个能力包装成模型看得见、能调的工具

换实现只需要在清单里关掉一行、装上另一行。上面那个工具包一个字都不用改——它认的是名字 `ctx.shell`，不是某个具体实现。

可迁移的做法

碰到一个陌生的包，先看它在哪个目录、叫什么名字，八成就猜对了它的角色。这套命名在 219 个包上是统一的。

记忆方法

同名的是插槽，`tool-` 开头的是模型看得见的，`ui-` 开头的是人看得见的，剩下的是插上去的实现和策略。

13.1

起点是空的，四层叠上去

SOURCE

apps/cli/src/profile-boot.ts 的 composeProfile()

术语

组合（profile）= 家目录下的一个文件夹，代表一套启动配置。

补充

四层之后启动器还可能

追加两条：组合里有 agent-presets 行时补上预置 agent 的路径；设了 DSH_TELEMETRY_DISABLED 时补上遥测硬关开关。两条都不满足就一条都不追加。

没有一份默认清单躺在那里等你改——它每次启动现算。

组合的根配置是一个空数组。这不是还没写完，是设计。

组合目录下那份 `cordis.yml`，全文是三行注释加一对空方括号。它每次启动都会被原样覆写回去，所以在里面写什么都留不住。整棵树由补丁叠出来，包括 agent 主循环那一行。

补丁按固定顺序叠四层：先是组合 `package.json` 里列的各个 bundle——`dsh-base` 永远第一，一次插进几十行，产品的绝大部分就是这一下来的；然后是组合目录里的 `cordis.patch.yml`；接着是家目录的 `$DSH_HOME/cordis.patch.yml`；最后是命令行上每个 `--patch`，按你敲进去的先后。四层各自装什么，10.2 有一张完整的表。

反直觉的一条

家目录层排在组合层之后，所以优先级反而更高。它被定位成「这台机器的本地偏好」，而不是「更笼统所以更弱」。越晚的层赢——想让一处改动在所有组合里生效，就写家目录那一层。

先别做

不要编辑组合目录下的 `cordis.yml`。你要改的是同一个目录里的 `cordis.patch.yml`。

13.2

补丁只有两种语义

SOURCE

vendor/include/src/index.ts 的 applyEntryPatches

没有删除

补丁里根本没有「删掉一行」这个操作，只有关掉。

后一层认 id 不认位置：整块换掉那行的配置，或者干脆把它关掉。

补丁的语义只有两种，一分钟能讲完。

第一种是**按 id 覆盖**：写上目标行的 id 和新配置，这一行的配置被整块替换。第二种是**追加**：往清单尾巴上插一批新行；插进来的行会立刻登记 id，所以再后面的层能接着改它。

PACKAGES/BUNDLE/WEB-APP/CORDIS.PATCH.YML 的两行真实内容

```
# 把 base 里的这一行关掉
- id: hmr
  disabled: true

# 整块替换这一行的配置
- id: session-query-sqlite
  config:
    path: ':memory:'
    openAt: never
```

光是 web 这一层，就用「关掉」摘掉了 24 行。为什么只有关掉、没有删除，2.2 讲过成因。

常见误解

以为补丁会跟原来的配置合并。不会。写了 `config` 就是整块替换，漏抄的字段会退回插件自己的默认值。想改一个字段，得把这一行的全部字段重抄一遍。

14.1

内核的最小集

SOURCE

docs/architecture.md、vendor/README.md、packages/boot/app-boot/

不是插件的东西可以一次数完，而且里面没有一行产品语义。

「一切皆插件」这句话值多少，全看剩下那一小撮有多小、装了什么。

内核成分	为什么它不能是插件
框架本体	服务查找、生命周期、事件分发本身。插件靠它才存在
Loader 与它的两个内建插件	Loader、`cordis:include`、`cordis:group`：它们是插件，但被静态导入、硬编码挂载，因此无法被配置替换
补丁层的顺序	四层的先后写死在启动器里
启动器注入的三个槽位	启动环境快照、命令行原始参数、退出函数
浏览器侧的模块加载器	「插件代码怎么到达浏览器」这一层
几条全局契约	模型看得见的必须能从日志重建（运行期不变量断言）；跨边界的 id 必须带类型标记（编译期）；少数服务只允许一个提供者
遥测退出开关	刻意抬到配置层之上：配置里怎么开，环境变量都能一票关掉

这张表里没有 agent、没有会话、没有提示词、没有工具，一个都没有。产品的全部语义都在补丁层——那正是第 21 章那句「没有特权内核」的实际含义。

一句话判断

要判断一个「插件化」系统是不是真的，就看它内核里有没有产品语义。这里没有。

14.2

上表最后一行：遥测

SOURCE

packages/bundle/base/cordis.patch.yml、packages/session/session-telemetry-otel/README.md

合规评审用得上

「默认关」和「代码里根本没有这个端点」是两回事。这里是后者不成立的那种。

另一处

37.5 从合规风险角度再讲一次，机制以这一页为准。

遥测出厂是关的，但默认端点已经写死在代码里。

这一行被刻意抬到配置层之上，值得单独说清楚。

三档	这一档会传出什么
DISABLED	**出厂默认。**整条管道根本不构建，一个字节都不发
FEEDBACK_ONLY	只有你在界面里提交一次反馈，才回放并上传那一段会话日志
FULL	会话事件持续上传

端点不是空的：`dsh-base` 里写死了 `https://harness-telemetry.deepseeksvc.com/v1/logs`，只有 `DSH_TELEMETRY_OTLP_URL` 能改。代码里确实带着一条指向 DeepSeek 的默认出口，只是出厂时那条管道没打开。

打开之后没有出厂脱敏

基础组合没挂任何脱敏规则，两个上传档位导出的是原样会话事件：消息正文、工具参数与结果（命令输出、文件内容）、完整系统提示词与工具表、压缩摘要、工作区路径。要脱敏得自己在那个拦截点上挂规则。厂商 API key 不在其中——它从来不进会话日志。

到这儿就够了

评审时说两句就够：出厂默认不上传；但默认端点是硬编码的，开启后没有出厂脱敏。要彻底断掉，设 `DSH_TELEMETRY_DISABLED`——只要非空就生效，填 `0` 或 `false` 同样是关。

IV

你按下回车之后，这句话到底经过了什么？

PART FOUR · 第四部

跟着一句话，走完它的全程

你在输入框里敲下一句话，按下回车，然后一路跟着这句话走。它会依次经过五个站点，每一站决定一件事，每一站都留了给你改的口子。

15.1

五站总图

SOURCE

packages/core/agent-loop/src/agent.ts

补充

第四站不是走到最后才做，它贯穿全程：前三站每个动作都是先写日志再往下走。

你敲下的那句话，要走完五站，才轮到下一句。

每一站的产物就是下一站的输入。知道自己在第几站，比记住每个模块叫什么名字有用得多。

01 **说什么 —— 第 16 章**

现拼出这次要发给模型的全部内容：提示词、工具清单、对话历史。

02 **怎么说 —— 第 17 章**

交给某一家模型厂商，接住它一帧一帧吐回来的回答，边接边落盘。

03 **动手 —— 第 18 章**

模型要求调用工具时，这一站决定哪些能跑、按什么顺序跑、要不要先问你一句。

04 **记下来 —— 第 19 章**

每件事都在发生的当时被追加进一份只增不删的日志。界面与导出都是它的投影。

05 **停不停 —— 第 20 章**

是回到第一站再问一次模型，还是这一轮到此为止、把控制权还给你。

一句话判断

五站里只有第三站（动手）会停下来等你，其余四站你插不上手也不必插手——所以调试时先看第三站。

15.2

两个基本单位

SOURCE

packages/core/agent-loop/src/agent.ts、packages/core/session/src/types.ts

补充

回合的开始与结束一定成对写进日志，哪怕这个回合一个步都没跑成。

一个「回合」是问到不必再问为止，一个「步」是问模型一次。

这两个词在后面五章里反复出现。先把它们的嵌套关系定死，后面就不会乱。

一个**步**（`step`）就是走一遍上一页的前四站：拼好内容、问一次模型、把模型要求的工具跑完，全程落盘。一个**回合**（`turn`）是一串步——你发一句话开一个回合，回合内不断开新的步，直到没有新东西需要再问模型为止。全书此后一律只用中文的「回合」和「步」。

单位	由什么开启	什么时候结束
回合	你发出的一句话，或一个子代理收到的委派	六种结束原因中的一种：正常完成、被打断、被策略挡下、出错、撞上输出上限、崩溃后被持久化后端补上句号
步	回合内每一次准备向模型提问	模型这一次的回答处理完，包括它要求的工具全部跑完

「撞上输出上限」这一条是黏的：一个回合里只要有一个步碰到过输出上限，哪怕后面的步都正常完成，这个回合的结束原因仍然记成撞上上限。这样你在日志里一眼能看出这一回合的输出被截断过。

记忆方法

看到「步」，想成「一次模型请求，外加它引发的那批工具执行」；看到「回合」，想成「你这一句话说完之前的全部步」。

16.1

提示词是一张注册表

SOURCE

packages/core/system-prompt/src/index.ts

就近覆盖

全局层加上作用域链，同名的那一段由最近的作用域赢。一个 agent 能只给自己换人设。

别共用 ORDER

同一个 order 值上的两段按注册顺序排，而注册顺序是插件加载的副产品。

系统提示词不是写死的一段话，是每一步现拼出来的。

没有模板文件，也没有一个「主提示词」等着插件往里填空。它是一张注册表：谁拥有某件事实，谁就注册这一段。

section 段落	一段带排序号的文本。绝大多数贡献都是这一类，比如某个工具包写自己的使用说明。
context 动态上下文	每一步现求值的运行期事实。它不会进入提示词，去向见下一页。
tools 工具清单	一个提供者，报出自己这一批工具的名字与参数结构。
variable 变量	供段落里 `{{name}}` 插值。没注册过的名字、求值为空的值，当场报错，不静默留白。

排序号有三条约定的号段：−100 是固定的 harness 身份，0 是部署方写的那一段人设，100–199 留给各个工具包的使用说明。号段是全局唯一的协调手段——除此之外，各段互不知道对方存在。

常见误解

「它有一个提示词模板，我改那个文件就行。」不成立。没有那个文件。要改，是往这张注册表里加一段、或者用同名段落把已有的那段盖掉。

16.2

会变的东西去哪了

SOURCE

packages/core/agent-loop/src/runtime-context.ts

会变的东西不写进提示词，它被做成会话里的一条消息。

理由很实在：省钱。系统提示词在请求的最前面，它改一个字，前缀缓存就从第一个 token 起全部作废。

什么是前缀缓存

厂商按请求开头的公共前缀复用算好的中间结果，前缀一变，后面全部重算。

当前目录的状态、待办清单、计划模式开没开——这类事实每一步都可能不同。如果它们住在系统提示词里，那就等于每一步都在改请求的开头，每一步都按全价重算。

实际做法是：提示词装配完之后，把这些动态段落单独渲染成一段文本，跟上一次留下的那段比。**没变就什么都不做**；变了，才往日志里追加一条 user 角色的消息。历史因此只增不改，请求的开头始终稳定，缓存一路命中到最新那条消息为止。

> Current runtime context: none. Earlier runtime-context snapshots no longer apply.
>
> PACKAGES/CORE/AGENT-LOOP/SRC/RUNTIME-CONTEXT.TS

大意：当前没有任何运行期上下文，之前那几张快照都已作废。所有上下文都消失时它会写这一句，而不是悄悄什么都不写——否则模型会拿着一张过期快照继续干活。

一句话判断

你写插件时，凡是「每一步都可能不同」的事实，都别注册成段落，注册成动态上下文。判据只有一条：它会不会在下一步变。

16.3

窗口装不下的时候

SOURCE

`packages/compaction/compaction-basic/`

默认值

阈值是路由到的上下文窗口的 0.8；保留最近约 0.16 个窗口的内容原样不动。

剪枝是可选件

靠可选服务 `ctx.toolResultPruner`；不挂也能压缩。base 层挂着，headless 沿用；web 那份把它连同 compaction-basic 一起关掉，交给会话自己的组合装回来。

上下文装不下时它不删历史，只是给旧的那一段盖一层遮罩。

日志只增不删是硬约定。压缩因此是追加一条新消息，声明「我遮住前面第 a 到第 b 段」。

01 两个时机会触发

一是每步开始前按压力估算，超过阈值就动手；二是厂商直接回「上下文超了」。

02 先做不花钱的那一步

两条路径都先跑一遍不需要模型的工具结果剪枝再重新计量；按压力触发的那条如果因此降回阈值以下就到此为止，一次模型调用都不发。厂商报「上下文超了」的那条不看阈值，剪枝之后直接进摘要。

03 才轮到摘要请求

选一段旧内容交给模型摘要，结果作为遮罩消息落进日志。原事件一条没删。

可迁移的做法

摘要请求复用会话自己的提示词与工具，把压缩指令追加成最后一条用户消息——于是上一次请求正好是它的前缀，缓存一路命中到那条指令为止。

常见误解

「压缩过就看不到原文了」不成立。被压的只是模型那份。

17.1

七种帧，两次折叠

SOURCE

packages/llm/llm/src/types.ts

体积代价

逐帧落盘让 JSON 外壳远大于内容，源码注释里实测约 56 倍。存储层把连续同块的帧打包成一行，解码时精确还原。

模型的回答被拆成七种帧，每一帧都先落盘再进屏幕。

厂商吐回来的字节流，在适配器里被翻译成一条统一的帧流。模型的一次回答由若干「块」组成：一段正文是一块，一段思考是一块，一次工具调用也是一块。

帧	含义
block-start	开一个新块，并说明它是文本、推理还是工具调用
text-delta	可见正文又多了一小段
reasoning-delta	思考过程又多了一小段
tool-call-delta	某个工具调用的参数又多了一小段
block-end	关掉某个块，并直接附上组装好的完整块
usage	这次调用花了多少 token
finish	结束，并说明为什么停：正常停、要调工具、撞上输出上限、出错、被取消

两次折叠：同一批帧，一次被逐条追加进会话日志各拿一个序号，一次在界面里折叠成正在生长的段落。渲染和存档不是两条通路，是同一批事实的两次纯计算——屏幕上看到的和日志里存下来的，不可能不一致。

到这儿就够了

记住一句：日志里存的是帧，不是句子；句子是折出来的。后面所有关于重放、导出、没有 API key 也能重跑测试的说法，都从这一句推得出来。

17.2

接一家新厂商

SOURCE

`packages/llm/llm-deepseek/src/`

为什么有两个实现

官方刻意用两套写法做同一件事：一个直连 fetch，自己拥有请求序列化与响应翻译（SSE 分帧交给 eventsource-parser 库）；一个整体包住 `@earendil-works/pi-ai` 这套第三方 SDK。

换一家模型厂商，要改的只有三个地方。

差异没有散在各处。它被按在三处——两个文件加适配器类上的一个方法，其余代码对厂商一无所知。

serialize

把 harness 的消息翻成这家厂商的请求格式。出方向。

translate

把这家厂商的响应帧翻成上一页那七种统一帧。回方向。

resolveModel

报出这个模型的能力：上下文窗口多大、默认输出上限多少、有哪几档思考强度。

第三项最容易被做错。常见做法是造一个「低 / 中 / 高」的通用枚举，各家再往上映射——结果必然出现悄悄降档和语义漂移。这里的选择是：**档位归适配器所有，核心不枚举取值**。你选了一个这个模型不支持的档位，请求在发出任何网络包之前就报错，绝不自动帮你换成相近的一档。

一句话判断

要接一个新厂商，先在纸上把这三件事分别写清楚。哪一件写不出来，哪一件就是这次接入真正的难点——而不是代码量。

17.3

出错之后

SOURCE

packages/llm/llm-retry/README.md

默认策略

重试两次，对空响应、限流、服务端错误、超时、传输失败生效；500 毫秒起指数退避，封顶 10 秒，带 10% 抖动。

出错时它不重试那条流，而是退回到上一个能重建请求的位置。

一条已经吐出半截字的流，没有干净的重来点。所以重试被整个搬到了回合边界上。

如果按常规做法在流外面包一层重试：前半截帧已经落进日志了，再试一次又落一批，日志里就留下两段互相矛盾的半截回答。这条流没有持久的「一次尝试」边界，包住它就是在污染事实。

换成的做法是：**一次适配器调用就是一次尝试**，绝不在里面偷偷重试。失败被统一包成一个失败对象——一个厂商无关的错误码，外加可选的状态码、厂商建议的等待时长、请求 id。这个对象被抛到回合边界上的一个拦截点，重试插件在那里决定退避多久，然后开一个**全新编号的回合**，请求从日志重新拼一遍。

重试次数也不放在内存里，而是往回翻日志数出来的。所以进程重启不会把计数清零，而路由被换成另一套策略之后，计数会重新开始。

配 ALWAYS 之前先读这条

重试模式配成 `always` 就没有次数上限：鉴权失败、配额用尽这类永远不会自己好的错误也会被无限重试。只有成功、取消或卸载插件能让它停。

记忆方法

重试不在网络层，在会话层——每次重试都往会话日志里写两条事件（排期一条、开始一条），次数也是翻日志数出来的。所以查重试历史去翻会话日志，不要去翻网络日志。

18.1

一个工具由哪几部分构成

工具不是一段返回文本的函数，它必须交出一份结构化的值。

工具交出的是一个**规范 JSON 值**——只含字符串、数字、布尔、数组、对象的普通数据，不能有函数、Map 或循环引用。给模型看的那段话是另外渲染出来的。注册一个工具，要一次声明齐下面四件事。

什么是工具

模型能点名调用的一个动作。读文件、跑命令、搜网页都是工具。

规模

仓库内置 52 个工具，分装在 24 个工具包里。

SOURCE

`packages/core/tools/src/index.ts`、`docs/tool-catalog.md`

01 模型能看见的三样

名字、一句说明、参数表。别的字段永远不上线——投影时走的是白名单。

02 干活的那个函数

拿到已冻结的参数，返回那个值。抛出的错误由注册表归一化成带 `isError` 的结果，异常不会冒进循环。

03 输出声明（强制）

值的 JSON Schema，加一个把值渲染成模型可读内容的纯函数，再加一个可选的界面卡片投影。

04 部署侧元数据

超时预算、能否与别的调用并发、卡片长什么样。这些不进提示词，不花 token。

常见误解

「工具的返回值就是给模型看的那段话」——不是。改那段话只改模型看到的文本，程序拿到的值一个字节没变。四件事怎么落成可运行代码，见 30.1–30.3。

18.2

一条管线，五个对外挂点

每次调用都走同一条管线，部署策略挂在段与段之间。

超时、权限、脱敏、外部钩子——这些东西一个都不写在工具里。它们是挂在管线接缝上的独立插件。

什么是 GUARD

一个只会说「不」的检查函数：返回一句理由就是拒绝，什么都不返回就是弃权。

参数不可改写

策略开始前就已冻结。因为日志、界面、审计和实际执行必须看到同一份参数。

SOURCE

packages/core/tools/README.md、docs/tool-execution-pipeline.md

挂点	它能做什么	谁挂在这
pre-execute	放行、拒绝、或转人工确认。权限就挂在这一环：工具动手前决定要不要问人、问的结果算不算过（机制见第 24 章）	权限、外部钩子
guard	只能拒绝，不能放行	所有者策略
execute	环绕整次调用，只准换取消信号	超时、埋点
post-execute	换内容、换值、或整个拦下	外溢、循环提醒
finalizeContent	工具自己的最后一次内容改写，同步、必跑、只能改内容	工具定义自带，不对外开放
result	广播点：只读观察，改不了	审计、指标

一条策略管住所有工具，而任何一个工具都不知道这条策略存在。想给全部工具加超时，挂一个环绕插件；想让某类调用必须人批，挂一个前置监听器。两件事都不用改任何一个工具的代码。

记忆方法

五个挂点分三类：前后两次改写加中间一层环绕，是能商量的**拦截点**；guard 是不能翻案的一票否决——它的返回值里根本没有「允许」这个分支；最后那个只是**广播点**。

18.3

被拒之后的那条路

模型被沙箱拦住后可以申请放宽，但这次放宽只对这一次调用生效。

沙箱和审批这两个旋钮本身怎么转，在第 24 章。这里只讲被它们拦下之后还剩哪条路：升级请求要同时满足四个条件才成立，缺一个就当场拒绝。

为什么需要它

没有升级路径的拒绝是终局的，会逼运营方干脆全局放开，反而摧毁沙箱。

一份实现两处用

命令执行和文件写入共用同一段升级代码，连错误文案都是同两个常量。

01 **必须严格更宽**

拿请求的模式和这次调用的实际模式比。平级或更窄都拒。判定发生在执行期，因为实际模式是每次调用才确定的。

02 **必须给出人话理由**

理由字段和目标模式必须成对出现，且不能为空。理由会原样进审批记录。

03 **必须有人批**

走的还是那条审批通道，结果还是那四个值。没有应答者就是拒。

04 **只盖在这一次上**

批准被盖在发起请求的那一次调用的策略上。下一次调用回到原来的模式，什么都没被记住。

成功标准

你在会话日志里能找到成对的「问了什么」和「答了什么」，且理由自带上下文足以事后审计——如果找不到，说明这条路径压根没走审批。

19.1

谁才是权威

别处是消息数组说了算，这里是事件流说了算。

几家 harness 都往磁盘写 jsonl，也都能靠它续上会话。差别不在落不落盘，在**约束**：这里有运行期断言盯着「模型看得见的，日志里必然已经有一笔」，而且压缩只加遮罩、不删事件。

什么是投影

把整条日志从头折一遍算出来的一个结果。日志变了，重折一遍就是新结果。

为什么不能静默跳过

一条不认识的事件可能改变后面事件的解释方式，跳过它会「成功地」重建出一个错的会话。

SOURCE

`packages/core/session`、`docs/persistence-catalog.md`

谁能进入模型看到的那条线

本仓库共声明 44 种事件类型，其中只有三种能进：`user/message`、`assistant/message`、`tool/result`。而且它们必须在写入时声明**怎么进**——是接在末尾，还是替换掉前面某一段。别的事件只落日志，模型永远看不见。插件用类型合并加进来的自定义事件，同样受这条规则管。

读回来时同样偏保守

碰到本次构建不认识的事件类型，读取端**直接拒绝打开整个会话**，而不是跳过它。只有写入方明确打了「可跳过」标记的纯信息记录例外。

> **压缩**
>
> 为什么不算删历史、代价是什么，见 16.3。

一句话判断

日志是账本，模型历史只是账本的一张视图。凡是能进入模型请求的东西，账本上必然先有一笔。

19.2

写入那一刻的三道关

事件在追加那一刻就被冻结校验，坏数据根本进不了日志。

校验不放在落盘的时候，放在写入内存的那一步。三道关全过才算写进去，任何一关不过，日志一个字节都不变。

为什么这么早校验

观察者是同步跑的。等到落盘才发现坏数据，它们早就照着这条事件动过手了。

SOURCE

`packages/core/session/src/json.ts`、`surface.ts`

01 无损 JSON 快照

递归拷一份。遇到大整数、函数、Map、循环引用当场抛错。

02 深冻结

整棵对象树变成只读。观察者拿到的是数据快照，不是能反过来改状态的活对象。

03 视图预校验

如果这条事件要替换掉前面一段区间——也就是「第 a 条到第 b 条这一段」——先验这段在不在。

过关之后才拿到序号

三道关全过，事件才被追加进日志并**同步**通知观察者。序号恒等于追加前的日志长度，不跳号，所以任何一条事件的位置都是可算的，不需要额外索引。观察者自己出的错会被记录并隔离，不会让一次已经提交的追加反悔。

常见误解

「写进内存算成功，落盘时还可能失败」——不会。三道关全在追加那一刻跑完，所以「内存里的日志」永远等于「能存下的日志」，落盘不是一个迟到的失败点。

19.3

日志真正长什么样

一段真实日志长这样：一次读文件，十一行。

仓库自带的端到端快照，截短脱敏后逐行标注。这是「44 种事件类型」落到磁盘上的实际形状。

为什么这里贴原文

前两页讲的都是规则。规则看一眼原文就懂，而这份原文是你写任何旁路插件时的唯一输入。

打包行不是事件类型

`reasoning-chunks` 这种不带斜杠的行是存储写法，读的时候会被还原成一条条原始事件。

SOURCE

`examples/acp-agent/tests/snapshots/`

SESSION.JSONL（关掉压缩后的原始文本，长值以 … 截断）

```
{"type":"session","version":0,"id":"e04cc262-…","cwd":"/work/demo"}
← 第一行不是事件，是头：格式版本、会话 id、工作目录
{"type":"turn/start","seq":1,"time":1785821390865,"data":{"turn":1}}
{"type":"step/start","seq":3,"data":{"turn":1,"step":1}}
← 回合与步的边界。只落日志，模型看不见
{"type":"user/message","seq":4,"surfaceOp":"append","data":{…}}
← 用户消息（「读一下 data.txt」）。surfaceOp 说明它怎么进模型视图
{"type":"assistant/chunk","seq":9,"data":{"chunk":{"type":"block-
start"}}}
{"type":"reasoning-chunks","seq0":10,"data":{"texts":["The"," user",…]}}
← 开一个思考块，随后整段思考增量被打包成一行存
{"type":"assistant/message","seq":69,"data":{…"type":"tool-call"…}}
{"type":"tool/call","seq":70,"data":
{"callId":"call_00_n4…","name":"read"}}
{"type":"tool/result","seq":71,"data":{…"call_00_n4…","isError":false}}
← 助手消息、工具调用、工具结果。callId 把三条串成一次调用
{"type":"step/end","seq":72,"data":{"turn":1,"step":1}}
{"type":"turn/end","seq":149,"data":{"reason":{"kind":"completed"}}}
← 收工，并写下这一回合为什么结束
```

到这儿就够了

写遥测、审计或检索插件，读的就是这个形状：`type` 选事件，`seq` 等于它在日志里的下标，`callId` 串起一次调用，`surfaceOp` 决定它进不进模型视野。

19.4

崩了之后怎么接上

进程崩了不会丢会话，它会替你把断口补齐。

崩溃、断电、被杀进程，日志都停在半路：有一个回合没有结束标记，有工具调用没有结果。冷加载时有一段专门的修复代码无条件跑一遍，把这些断口补成一份模型能接着往下读的记录。

为什么不截断

长任务里一个回合可能极大，那些事件都已经真的落过盘，扔掉它们比补齐边界损失大得多。

活会话不修

修复只对冷加载的会话做。读一个正在跑的会话，它会等内存里那份权威快照落盘且回合闭合。

SOURCE

`packages/core/session/src/repair.ts`

01 给悬空的调用补结果

助手已经要求调用但日志里没有开始记录，补一条 `TOOL_NOT_STARTED`；已经开始但没有结果，补 `TOOL_OUTCOME_UNKNOWN`——这条会明确告诉模型：只有只读或幂等的活可以直接重试，有副作用的要先核实，或者问人。

02 补上边界

再补一条步结束，和一条结束原因为 `interrupted` 的回合结束。这正是循环自己永远不会产生的那个值——所以你在日志里看到它，就知道这是崩溃恢复补的。

03 撕坏的尾巴单独处理

压缩分帧的最后一帧只写了一半，读取端保留前面完整的记录，从那一帧截断再重写。但坏在最后一个已提交的回合结束之前，那就是损坏，直接拒绝。

先别做

别把恢复出来的会话直接当成没事发生过：接着往下跑之前，先核实那条标着「结果未知」的调用到底有没有真的动过东西。也别把主动取消跟这里混为一谈——你按下的「取消」在日志里记成 `aborted`，只有崩溃恢复补的才是 `interrupted`。

19.5

收益与代价

同一条流折出五份投影，代价是日志会长得很大。

模型看到的历史、界面看到的状态、落盘的存储、遥测导出的记录、检索用的索引——五份东西全是同一条流的折叠结果，因此不可能互相分叉。

加一个横切能力有多便宜

订阅同一条流即可，不必改循环、不必开新存储、不必接新生命周期。

版本策略是临时的

日志格式版本固定为 0，没有任何迁移链，版本对不上只拒绝。这只在预发布阶段成立。

换来的

最实用的一条是回放测试：录一次真会话，之后的回归全部离线跑，**不需要任何 API key**。浏览器端的界面快照测试在持续集成里也被强制成只回放、不重录。

付出的

逐 token 的增量事件，JSON 信封比载荷本身大得多——源码注释记的实测值是**约 56 倍**。缓解办法有两层：连续同块的增量打包成一行（够三条才打），落盘再默认套一层 Zstandard 分帧。两层都是纯存储编码，逻辑日志一个字节没变。

做部署评估要的量级

仓库里签入的 74 份端到端会话快照合计约 720 KB、182 步，平均**一步约 4 KB**（未压缩、已打包）。按这个单价推测：一次半小时、跑一两百步的编码会话，`$DSH_HOME/sessions` 大约多出 0.4–0.8 MB，压缩后更小。真正会把量级拉上去的是大段工具输出，不是对话本身。

记忆方法

记住两件事：状态不在内存里，在日志里；日志只增不改，压缩靠遮蔽。剩下的批窗写盘和屏障时机，都是这两条的自然推论。

20.1

循环怎么决定再问一次

回合与步

定义见 15.2。

SOURCE

packages/core/agent-loop/src/agent.ts

继续问下一次，是数据决定的，不是监听器决定的。

一步跑完，循环只看两个数据：这一回合有没有已经定下的结束原因，收件箱里还有没有待办输入。两个都为空才收工。

一个回合的六种结束原因

completed	模型自然停下，或首步输入被改写成空
blocked	这一步被拦下，整个驱动器退回空闲
max-tokens	撞上输出上限。这个原因是黏的，后面步骤跑成功也不会降级
error	模型请求最终失败
aborted	收到取消
interrupted	循环永远不会产生这个值，它只由崩溃恢复补写（19.4）

常见误解

「挂个回合收尾的监听器就能让它别停」——不行。那个事件没有否决返回值。想续这一回合，得往收件箱里塞一条消息；想提前收尾，得让工具结果带上「本轮到此为止」的标记。

20.2

子代理是一段对话

子代理不是发出去等结果，它是一段可以续聊的持久对话。

一个子代理 = 一份持久会话，外加最多一份还活着的进程内实例（它创建的后代都结束了才释放）。那份实例可以跑很多轮；进程重启后从存储里冷恢复。

对照

Claude Code 的 Task 发出去就只能等一个最终结果，没有续聊、打断、枚举和主动回报。

只有进程内后端支持

接出去的那几种子代理后端一律不支持续聊，也不支持结构化输出和工具过滤。

SOURCE

`packages/subagent/subagent/README.md`

父这边能做	效果
`send_message`	在同一段对话上开一个新轮次，不是新建一个孩子
`interrupt_agent`	只停当前轮次。**尚未认领**的排队消息、进程内实例、已发布的后代都保留；已被这一轮认领的工作不会重新排队
`list_agents`	默认只列直接的可续聊孩子；`scope: descendants` 才按稳定先序走完整棵树。两种都不唤醒、不恢复任何一个孩子
`report`	反向：孩子主动多次回传中间发现

两种回话不合并

孩子主动说的话，和运行时对孩子下场的陈述，在父亲的记录里是两种不同来源。最需要通知的场景——撞上限、模型故障、被取消——恰恰是孩子没机会开口的场景。

一句话判断

需要来回追问、需要中途叫停、需要看着它干活的任务，才值得开子代理；一问一答就能完的活，自己做更省。枚举出来的状态不是投递承诺——显示在跑的孩子，发消息时仍可能因所有权冲突失败。

20.3

委派那一刻发生了什么

子代理的权限在委派那一刻冻结，而且只会变窄。

孩子拿到一个全新的作用域，只继承父的显式沙箱设置；审批策略一律被钉成「不问」，需要人批的操作在它那里是确定性拒绝。

为什么钉死不问

被委派的孩子背后没有人类应答者，问了会永远阻塞。

深度是绝对值

上限为 1 意味着根可以委派、但它的孩子不能再委派；上限为 0 就是完全禁止委派。

限制写进孩子自己的日志

沙箱覆盖和审批策略以「来自委派」为来源，落到孩子自己的会话里。所以冷恢复时只读孩子的日志，就能重建它当时的有效策略，不必去问父亲。

模型不能挑用哪种子代理

工具里根本没有「子代理类型」这个参数。类型由部署时挂几个实例决定——不同的工具名、不同的后端、不同的人格、不同的工具过滤。权限收缩是部署方的决定，不是模型运行时的选择。

先别做

别指望在子代理会话里临时提权，也别把「工具还看得见」读成「还能用」：超限时工具仍然对模型可见，运行时才拒绝——这会白白花掉一次调用，是刻意的取舍。

20.4

让模型自己写编排

模型可以自己写一段脚本来编排，但那一层不是安全沙箱。

脚本是一段普通 JavaScript，跑在一个独立工作线程里；脚本里调用的几个钩子通过消息打回主进程，去起真正的子代理。模型写的是编排本身，不是一句一句的委派。

和子代理的分工

子代理是一次委派一个孩子；这一层是一次写完整套扇出与汇总。

省在哪

中间那些子代理的输出只在脚本里流转，不进父的上下文。

不是安全边界

文档自己写明：worker 里的 `node:vm` 只是塑形 API、不是安全边界；逃出去的脚本能以**宿主进程**的权限拿回 Node 能力。

工作线程换来的两件事

一是模型写的同步计算不会卡住主事件循环；二是终止线程是真正的最终止损——脚本迟迟不肯收尾时，宿主等一段宽限期后强制收尾，并给悬空的记录补上取消标记。

脚本是模型写的，引擎不是

引擎在一个接缝后面，由部署方选。所以「谁写脚本」和「脚本跑在什么里面」是两个可以分开决定的问题——把脚本写死成常量，同一个引擎就变成一条模型改不动的流程。

常见误解

「跑在工作线程里 = 关在沙箱里」不成立。这一层提供的是主循环隔离和强制终止，不是对抗性代码的隔离。真要跑不可信脚本，得在同一个接缝后面换一个独立进程或容器的引擎，别直接用现成的这个。

20.5

五种做法怎么挑

同一件事有五种做法，挑错了代价是上下文和钱。

这五种不是互相替代。它们按两件事分开：编排由谁写，以及那段编排模型改不改得动。

为什么放在这里

前四页讲的是每一种怎么转。到这一步该回答的是：手上这个活该用哪一种。

都能关

这五种里除了普通工具，其余四种都是可以整包不装的。装哪几种是部署方的决定。

SOURCE

`docs/tool-catalog.md`、`packages/core/tools/README.md`

手上的活	用哪个	为什么
一问一答就能完	普通工具	最便宜。结果直接进上下文，别的都是在这条路上再加一层
要来回追问、中途叫停	子代理	一段可续聊的持久对话，父这边能续、能停、能枚举
一次调一串工具，只要汇总	`run_code`	中间结果留在程序里，只有 print 或 return 的部分回到上下文
扇出到大量子任务再收口	`workflow`	多文件审计、迁移、多角度调研、对抗性核验，一次写完整套编排
要一条模型改不动的流程	`ralph`	循环写死成常量：模型只能给不可改的目标和轮数上限，每轮开一个全新的孩子

仓库里管 `run_code` 这套叫 **Code Mode**：注册表不再把每个工具的 schema 发给模型，而是生成一段 SDK 声明，模型写 TypeScript 去调它们。纯 `code` 模式下它是模型唯一能直接调的工具。

一句话判断

从上往下挑，能停在哪一行就停在哪一行。每往下一层，多买到的是编排能力，多付出的是一层看不见的东西——脚本里出的错，模型只会告诉你「程序失败了」。

V

五章五张账单，外加权限的
完整机制

PART FIVE

它的五个特点

这五个特点本身，第三部和第四部已经讲过怎么运转。所以这一部不重讲机制，只算账：21 到 23 每章一页，只写这个特点的代价；24 章权限例外，它是全书唯一完整讲权限的地方，机制一页、代价一页；25 章讲仓库自己的工程约定，三页。只讲好处的那种介绍，你在别处已经看够了。

21

代价：没有内核，也就没有保护

SOURCE

docs/capability-seams.md、packages/boot/app-boot/README.md

换掉主循环的代价

下游依赖的是 dsh-agent 的事件与服务，换掉它等于把这些全部重新实现一遍。

同进程

packages/core/scope/README.md 明写：作用域路由的是可信的同进程插件，不是沙箱，也不是权限边界。

代价：没有内核保护你，一行插件就能停掉整个工具派发。

这个特点第 14 章讲过，这里只算账：「什么都能替换」和「什么都能被替换掉」是同一句话。

先把最诱人的那句收紧。主循环在配置里也只是一行，能摘能换——**但那一行没有契约接住你**：`ctx.agentLoop` 标的角色是打包层，不是能力接缝。它是「一行配置」，不是「一个接缝」。

拦截点是双向的

挂在 `tools/pre-execute` 上的监听器只要不往下传，工具就不会执行。策略能拦住危险操作，靠的正是同一个机制。

缺依赖的插件实例会一直等下去

依赖没人提供的插件既不报错也不运行，只停在等待态，而等待态的插件实例不保活事件循环。启动这条路已加固：harness 在启动末尾对整棵树跑一遍断言，点名「谁缺哪个服务」并失败退出。没人管的是启动后动态挂的。

常见误解

拦截点不只是加东西的地方——它是一个插件能对你做什么的上限。

22

代价：换后端只是改配置，改错了没人拦

SOURCE

examples/headless-agent/e2b.cordis.yml

E2B

一个远端代码沙箱服务；这里指把文件和进程都放到远端容器里跑。

补丁不能改名字

行的 name 是守卫。替换只能「禁掉旧行 + 插入新行」，不能悄悄把一行改成别的包。

代价：接缝省掉写适配代码的工夫，省不掉判断这组配置自不自洽。

这个特点第 11 章讲过，这里只算账。仓库里有个现成例子：把整个执行世界搬到远端沙箱。

E2B.CORDIS.YML —— 结构，非逐字

```yaml
# 禁掉两行
- id: subprocess    disabled: true
- id: fs-local      disabled: true
# 插入一组新行
- insert:
    - dsh-e2b              # 远端沙箱本体
    - dsh-subprocess-e2b   # 进程能力的远端实现
    - dsh-fs-e2b           # 文件能力的远端实现
    - …以及沙箱策略、终端、LSP 等消费方
```

被禁掉的只有进程和文件两个底层实现。**bash 执行器那一行一个字都没改**——它只认契约，不认谁在下面执行。

会静默出错

同一位置挂两个实现会在加载时抛错，但「这一组是否自洽」没人验。示例文件顶部写着：漏掉设置远端工作目录那一行，本地 bash 仍指向宿主路径，每次调用都会失败。

先别做

别在换后端的同一轮里顺手加别的行——出了问题你才知道是哪一边。

23

代价：陌生的日志一律不读

SOURCE

packages/session/session-persistence/src/coordinator.ts、packages/core/session/src/repair.ts

格式版本仍是 0

而且没有任何迁移链。文档里提到的升级步骤只存在于决策笔记，是延后设计，不是已实现的代码。

崩溃能自愈

被中断写坏的会话不在此列，冷加载时会无条件修复一遍，不是只能丢弃。

代价：读到不认识的事件类型，它宁可拒绝打开整个会话。

这个特点第 19 章讲过，这里只算账。拒绝本身是对的——静默跳过会「成功地重建出一个错误的会话」。

仓库外插件加的事件，读不回来

「这个 build 认识哪些事件」是仓库内类型声明生成的固定清单，当前 44 个。第三方插件新增的类型天生不在里面，重开那条会话就会被判为不支持。信封上留了个「可忽略」标记位，但当前的写入接口不接受它。

只拒绝，不迁移

格式版本对不上就拒绝打开，并告诉你往哪个方向走；派生数据——全文索引、投影缓存——则直接重建。这套做法只在预发布立场下成立，第一个正式版本之后要重做。注意区分：被崩溃写坏的半截会话不走这条路，冷加载时会无条件跑一遍修复，给没等到结果的工具调用合成错误结果。

到这儿就够了

会话日志现在的定位是「本 build 能完整重放的记录」，不是跨版本的归档格式。别再往下找它的迁移方案——还没有。

24.1

两个旋钮各管什么

SOURCE

`sandbox/sandbox-policy/`、`interaction/user-approval/`、`interaction/permission-presets/`（均在 `packages/` 下）

沙箱只管写

三档都不限制读，也不管网络和子进程。只读档位下，agent 仍能读遍整台机器，包括存 API key 的那个文件。

三档是配置，不是硬编码

档位表写在 base 层那一行的配置里；包自带的默认表只有两档，`read-only` 是这个发行版加的。

两个旋钮：沙箱管能改哪些文件，审批管问不问人。

界面上那三个档位不是三种权限模型，是这两个旋钮的三组取值捆绑。

旋钮	取值	它唯一决定的事
沙箱模式	`read-only`	一个字都不许改
	`workspace-write`	只许改会话工作区（建会话时钉死的目录）底下的东西，外加一些平台临时目录
	`danger-full-access`	不再限制文件改动
审批策略	`ask`	问出去；**没有应答者就当拒绝**——服务自己从不弹窗
	`never`	连交互都不进，直接拒，并告诉模型别再申请放宽沙箱

批准只有 `allowed-once` 一个取值：没有「以后都允许」，没有规则表，没有撤销——**一次批准只对被问的那一次调用生效**，下次重新问。三档只是把两个取值捆成一个名字，切换时只有真正变了的那个旋钮会被拧。被沙箱拦下之后还留了一条路：模型可以申请更宽的模式，那次放宽同样只盖在这一次调用上，成立条件见 18.3。

常见误解

以为只有三种可能。两个旋钮拧出表外的组合（只读 + 不问）照样成立，界面显示 `custom`——能显示，不能选。

24.2

代价：粒度粗，而且人会关掉它

SOURCE

packages/interaction/user-approval/README.md、packages/bundle/base/cordis.patch.yml

出厂默认

可写工作区 + 每次询问。进程级的回退档位由环境变量 DSH_PERMISSION_MODE 决定。

钉死在会话上

权限在会话创建那一刻写进这个会话，之后改设置只对以后新建的会话有效。

代价：它粗到不能只放行 `git status`，于是长任务里人会把它关掉。

机制在 24.1，这里只算账：克制是真的，可预见的退化也是真的。

01 **粗到看不见参数**

递给答复者的只有工具名、原因和一个可选的调用 id——「只放行 `git status`」在这个接口上连表达的余地都没有。

02 **长任务里会一直弹窗**

只有「这一次允许」，没有「以后都允许」，跑几十步的任务就问几十次。

03 **于是人会全局设成不问**

忍不了弹窗的人会把策略调成不问，保护就只剩沙箱一层——而沙箱只管写不管读。出厂三档里「不问」还跟「完全放开」捆着，选它等于两层一起关。

这两个旋钮不是凭空来的：它与 Codex CLI 的沙箱模式和审批策略几乎同名同义，差别是 Codex 的审批策略还有 on-request 和 on-failure 两档。所以「这一版还没有规则表」比「刻意没有」准确——on-failure 正是为缓解第三条设计的。

一句话判断

想关掉审批时，正确的动作是缩小工作区，不是调大权限。

25.1

一条写在决策笔记里的前提

SOURCE

.agents/notes/implemented/process/2026-06-11-quality-gates.md

什么是门禁

一条退出码非零就会让 CI 变红的命令。约定写成散文没人守，写成门禁才守得住。

这个仓库的工程实践，是按「写代码的主要是 agent」这个前提设计的。

它不是一句宣传语。原话写在决策笔记里，而且直接解释了后面所有看起来过分的东西。

> 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.
>
> .AGENTS/NOTES/IMPLEMENTED/PROCESS/2026-06-11-QUALITY-GATES.MD

大意：这个代码库主要由编码 agent 开发。比起散文写成的约定，agent 遵守被强制执行的门禁要可靠得多；而当干活的是 agent 时，「工作量大」不构成一条成本论据。

后半句是这一章的钥匙。人类主导的仓库里，「每个包都要写一份运行时不变量」这种提案会被工作量直接否掉。这里不会——所以凡是散文约定加 code review 保不住的，一律下沉成脚本。整套东西分四层：

层	它承载什么
指令层	`AGENTS.md`：每个会话都要在上下文里的标准命令，每条一到三行
门禁层	几十条会让 CI 变红的脚本，由一个带依赖图的编排器统一调度
测试层	单测、逐文件覆盖率、无密钥回放、真实接口端到端
记忆层	决策笔记：承载「为什么」和「放弃了什么」

记忆方法

四层：指令、门禁、测试、记忆。前两层管当下，后两层管以后。

25.2

它换来了什么

SOURCE

package.json 的 scripts、vitest.config.ts

「活跃」的口径

已实现 505 + 提案中 25 + 已否决 11，不含已归档的那一批。

最值得偷的一招

回放用的素材就是产品自己写出的会话日志，不另造 fixture 格式。

换来的是一套连文档陈旧都会亮红灯的检查体系。

下面这些数字不是规模炫耀，它们是「工作量不算成本」这条前提的直接产物。

37
verify–* 检查脚本

12
生成器，其中 9 个配了 --check

219
个包，每包一份运行时不变量

541
篇活跃决策笔记

可枚举的事实一律不许手写

工具目录不是解析源码得来的，而是真的把每个工具插件启动一遍，从运行时注册里读计算后的结果。配了检查模式的生成器挂进 CI，于是「文档过期」是一次 CI 失败，不是一次 review 疏漏。

日志既是输入也是期望值

回放测试直接用产品写出的会话日志当素材：从里面反推出每一次模型调用的输出序列，无需密钥就能确定性重放。于是「能回放」和「日志格式没坏」变成同一个断言。

一句话判断

这套里最值得抄的一条：别为测试另造录制格式，直接把产品的持久化日志当 fixture 用。

25.3

代价：这套体系不是给你用的

SOURCE

CONTRIBUTING.md、docs/i18n/README.md

不接受外部 PR

官方原文：项目仍处早期，目前无法接受外部 pull request；问题与缺陷的上报被引导到 GitHub Discussions。

代价：这套体系的维护成本全归内部，而它对 fork 并不友好。

读到这里容易生出「我们团队也该这么干」的冲动。先看两条账。

成本全落在内部，没人帮忙分担

仓库明说目前不接受外部 pull request，社区参与被引导到写插件、写文章、答疑上。于是这几十条门禁的维护、修复和升级全部由内部团队和它们的 agent 承担，外部既贡献不了改进，也分担不了负担。前面那句「工作量不构成成本论据」的成立条件，也只有他们自己满足——你的团队要抄这套，先问问干活的是不是 agent。

双语哈希门禁，对 fork 是一盏长亮的红灯

每份在范围内的文档都是三兄弟：英文、中文、加一个记录两侧内容哈希的配对文件。改了英文而没同步中文，门禁立刻红——而它的清单里只有排除项，没有逐文件的启用名单，也就是默认全覆盖。你 fork 之后随手改一段说明，就得连中文一起改并重录哈希。不打算维护中文的 fork，等于永久带着一条红色检查。

先别做

别把这套门禁整套搬进你的仓库。先抄那两条便宜又通用的：用产品日志当测试素材，和「可枚举的事实一律用生成器产出」。

VI

往这套东西里加自己的东西，从哪一层接手，代价是什么

PART SIX · 第六部

插件：它这次真正的主角

前面五部都在说这套东西怎么转。这一部换个立场：你要往里加东西。

26.1

一条判定标准

SOURCE

packages/hooks/hooks-claude-code/README.md · packages/hooks/hooks-codex/README.md

补充

这两个包把别家的钩子配置桥接进来，所以别家的扩展面有多大，仓库里有据可查。

判断一个系统是不是真的插件化，只看一条：你能不能删掉它自带的东西。

扩展点多不说明问题。能不能把宿主自己的功能摘掉，才说明产品和插件站在同一个平面上。

这条通路第 02 章已经证过：整个产品叠在一个空数组根配置上，官方自己的 web 组合就在走它。这一页只做一件事——把 dsh 放回同类系统里比一比，只跟做同一件事的东西比。

同类系统	你能改到哪一层
Claude Code	命令钩子加子代理。钩子是外部进程，靠退出码和 stdout 说话，能拦，不能替换内置实现
Codex CLI	十个固定钩子点。要动钩子够不着的地方，只剩 fork 一条路
Koishi / Cordis 原生	能整体替换一个服务实现，但 app 内核仍然是特权的，摘不掉
dsh	按 id 写一行 `disabled: true` 就摘掉，或整块换掉那行——主循环、会话持久化、权限审批一视同仁

前三行不是能力不足，是产品选择：可动的面越窄，官方越能保证装了别人的东西之后自己还站得住。dsh 反过来，上限抬得很高，代价是它必须假设所有插件都可信。那堵墙第 37 章正面讲。

一句话判断

看一个插件系统的上限，不要数它有多少扩展点，要看它允许你删掉多少自带功能——再问一句，删得掉的东西谁来担保。

26.2

四条替换通路

SOURCE

docs/architecture.md

注意

四条路都不需要 fork，但都能把系统改坏，没有护栏。

改掉它的内置行为有四条路：一条不写代码，三条对应 ctx 上的三类调用。

认清这四条，第 31 章那张全图就只是在往里填名字。

01 **不写代码：按 id 顶替或关掉任意一行**

在补丁层里写同一个 id，那行配置被整块替换；写 `disabled: true`，那行不挂载。是整块替换不是深合并——改一个字段，要保留的字段得全部重写。

02 **加东西：走一个注册入口**

「注册入口」是 ctx 上那些往系统里添内容的方法：一个工具、一条斜杠命令、一段提示词片段、一个界面槽位。加进去的和官方自带的排在同一张表里。

03 **插话：站到一个拦截点的中间**

`tools/pre-execute`、`agent/pre-step`、`system-prompt/assemble` 都是拦截点：监听器不调 `next()`，下游全部拿不到这次调用，包括 harness 自己的工具分发。标 `complete: true` 的提示词片段也在这一层，它一出现，别人的片段全部作废。

04 **换世界：卸掉一个能力的实现，挂上另一个**

把 shell 那行摘掉、挂一个远端的，声明依赖 `shell` 的插件会干净地重启一遍。`packages/e2b/` 就是实证：换掉文件系统和子进程两个实现，整套搬到远端。

到这儿就够了

四条通路记住名字就够：不写代码、加东西、插话、换世界。每个入口具体叫什么，第 31 章列全。

27.1

五种角色

SOURCE

docs/capability-seams.md · docs/tool-catalog.md

SOURCE

apps/cli/config/agent-presets/

补充

数字来自 0.1.0-rc.5 的仓库源码，不是 npm 上的 rc.6。

插件分五种角色，动手前先确认自己要写的是哪一种。

这五类的写法、依赖、难度完全不同。搞混了，第一步就会走岔。

角色	官方有多少	它做什么
能力定义	26	只定义抽象类和词汇，不含实现。直接挂载会抛错
实现	约 45	把某个能力定义实现出来，占住那个服务名
工具	24 包 / 52 个	把能力包装成模型看得见、能调用的工具
策略钩子	十几个	不注册工具、不提供服务，只挂监听器
界面	39	浏览器那半边，往界面的槽位里放组件

代表例子：`ctx.fs`、`ctx.shell`、`ctx.llm` 是能力定义；`dsh-fs-sandbox`、`dsh-bash-sandbox`、`fs-e2b` 是实现；`tool-bash`、`tool-fs` 是工具；`timeout-policy` 是策略钩子；`ui-plan`、`ui-jobs` 是界面。

那 52 个工具不会同时摆在模型面前。web 组合在基础层上一口气关掉 24 行（工具、计划模式、压缩引擎都在内），改由每次会话挂上的会话预设决定这一次露出哪些——出厂四个预设里，「极简模式」只给两个：持久 bash 和 `str_replace_editor`。逐条清单见附录 C。

记忆方法

能力定义写契约，实现填契约，工具把契约端给模型，策略钩子在旁边旁听并插话，界面只管画出来。

27.2

最被低估的一类

SOURCE

packages/guard/timeout-policy/src/index.ts

补充

官方把这类拆到「一个策略监听器就是一个包」的粒度。

最被低估的一类插件不注册工具、不提供服务，只挂一个监听器。

它不改任何数据结构，也不进模型看到的清单，却能实实在在改变系统的行为。

官方自己就有一批这样的包，每个只干一件事：

timeout-policy 超时策略	包住工具分发，换上自己的截止信号。整包 81 行，是这类插件最好的模板。
fs-observation-policy 读后写策略	监听文件写入与编辑意图，没读过的文件不许改。与文件系统的具体实现互不相干。
time-context 时间上下文	在每一步之前把当前时间注进模型将要看到的消息里。

组织里最常见的需求——禁掉某些命令、按小组限制工具、把每次调用送去审计——全部落在这一类里，二十来行就能起步。

常见误解

以为想改变行为就得替换实现。大多数时候不用：挂一个监听器就够了，而且不会牵动任何别人的包。

27.3

门槛阶梯

从不写代码到必须读内核，切入点排成五级台阶。

先认清自己站在哪一级，再决定这个周末要走多远。

SOURCE

docs/cookbook/extension-cookbook.md

SOURCE

packages/mcp/mcp-client/README.md

补充

阶梯只排「上手成本」，不排价值。最低那一级同样能解决真问题。

门槛	做法
不写代码	写一段配置把外部 MCP 服务器挂进来，它的工具直接出现在模型面前；或复用你已有的 Claude Code / Codex 钩子配置
极低	纯注册一个领域工具。注册表直接收裸 JSON Schema，MCP 来的工具就是这么进来的
低	策略钩子、斜杠命令、系统提示词片段、网页检索与抓取的实现
中	模型适配器，shell / 文件系统 / 子进程 / 沙箱的实现，技能来源、压缩引擎、子代理后端、会话存储后端
高	浏览器侧界面模块、对话节点、RPC 契约、工作流引擎、整体改写提示词装配

前三级都在主机这一侧，一个下午能上线；第五级是另一个复杂度量级。

「不写代码」的三条前提

出厂组合里一个 MCP 服务器都没有；可执行文件要自己先装好；配置块要手写进 `cordis.patch.yml` 或 `--patch` 覆盖层，一个服务器最少六到八行。

先别做

这一轮别从第四、五级起步。第一个插件在前两级里挑，先把「写出来—装上去—看见它生效」走通。

28

三项决定

SOURCE

docs/user/develop/basic/publish.md

补充

组合（profile）是 $DSH_HOME/profiles/ 下的一个目录，描述一套可启动的搭配。

动手之前先定三件事：在哪执行、由谁拥有、怎么发布。

这三个问题决定了你写的是哪种文件、放在哪个目录、别人怎么拿到。答错一个就得返工。

01 **在哪执行：主机这半边，还是浏览器那半边**

主机侧——工具、策略、能力实现、提示词、斜杠命令——一个下午能上线。浏览器侧是另一回事：组件永远拿不到上下文对象，只能走槽位注册和四条固定的属性通道，而且插件集变了必须重启进程。

02 **由谁拥有：机器、组合，还是单次会话**

机器级的偏好写 `$DSH_HOME/cordis.patch.yml`；只给某一套搭配用，写那个组合目录里的 `cordis.patch.yml`；只想给某一类会话换一套能力，用**会话预设**。层级选错，要么别人装不上，要么你其他组合被连累。

03 **怎么发布：三个档，成本差一个量级**

只给自己用，停在 `--patch` 覆盖层，不用建包。给团队用，打成 bundle 走 `dsh plugin add`，这是唯一的分发路径。第三档——合进官方仓库——现在走不通，成因与它对你的实际影响见第 37 章。

一句话判断

三个问题里只要有一个还答不上来，就先停在 `--patch`，别急着建包——建早了每改一次都要重新构建一次。

29.1

三样东西

SOURCE

docs/user/develop/basic/index.md

前提

这条路径从源码检出跑：pnpm install && pnpm run build 之后用 pnpm dsh。

让它认出你的插件，只要三样东西：一个文件、一行配置、一条命令。

不用 npm、不用打包、不用建 package.json。

01 **一个文件**

`scratch-plugin/src/my-plugin.ts`。导出一个 `apply` 函数就是一个插件，没有基类和清单文件。

02 **一行配置**

`scratch-plugin/cordis.yml`。一条 `insert`，把这个文件插进正在跑的组合里。

03 **一条命令**

启动时带上这个覆盖层，它叠在所有官方层之后。

SCRATCH-PLUGIN/SRC/MY-PLUGIN.TS

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

export const name = 'hello-plugin'
export function apply(ctx: Context) {
  console.log('[hello-plugin] plugin loaded!')
}
```

成功标准

启动时终端出现 `[hello-plugin] plugin loaded!`，界面照常打开。

29.2

绝对路径这个坑

SOURCE

docs/user/develop/basic/index.md:56

SOURCE

vendor/loader/src/config/group.ts:79 · packages/boot/app-boot/src/index.ts:773

注意

Loader 的挂载是事务性的：一个条目挂不上就整棵树回滚并抛出，启动器再冠上阶段名。

配置里的插件路径必须写成绝对路径——写相对路径不是悄悄失效，是让启动直接失败。

这是新手在这条路径上最常摔的一跤。好在它摔得很响。

SCRATCH-PLUGIN/CORDIS.YML

```yaml
# 先在仓库根目录跑一次 pwd，把打印出来的路径填进去
- insert:
    - id: hello
      name: '/absolute/path/to/deepseek-harness/scratch-plugin/src/my-plugin.ts'
```

```
$ pnpm dsh web --patch ./scratch-plugin/cordis.yml
```

> The plugin path must be absolute.
>
> DOCS/USER/DEVELOP/BASIC/INDEX.MD

补丁文件只贡献配置，不会把模块解析的基准目录挪到自己身上——相对路径按组合目录算。

按组合目录找不到那个文件，启动不会带着半棵树继续跑，而是直接终止，终端上打出 `dsh: plugin tree failed to load: failed to apply loader entry hello (…)`。看到这条，先怀疑路径基准，别去翻插件代码。

先别做

这一轮别建 `package.json`、别声明 `dsh.bundle`、别发 npm。打包是第 32 章的事，现在建包只会让你每改一行都要重新构建一次。

29.3

怎么确认它进去了

SOURCE

apps/cli/src/dump-config.ts

补充

这份输出怎么读见 10.5，四层怎么叠见 10.2；这一页只把它当确认手段用一次。

插件「没动静」时的第一件事不是加日志，是先把组合打印一次。

大半的问题在这一步就现形了：你那行到底进没进去，一眼可见。

```
$ pnpm dsh web --patch ./scratch-plugin/cordis.yml --dump-config
```

输出片段

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

# == /Users/you/deepseek-harness/scratch-plugin/cordis.yml
- id: hello
  name: /Users/you/deepseek-harness/scratch-plugin/src/my-plugin.ts
```

层标签有两种形状：bundle 那层印的是纯包名，你自己那层印的是解析后的绝对路径——不是命令行里敲的 `./scratch-plugin/…`。想 grep 自己那段，按绝对路径找。

一句话判断

先在这份输出里找 `id: hello` 那一段：找得到、并且下面的 `name` 是一条真实存在的绝对路径，问题就在插件代码里；找不到，问题在补丁层，翻代码是白费。

30.1

完整的第一个工具

SOURCE

docs/user/develop/basic/tool.md

补充

inject 声明本插件必需的服务，服务没就绪之前它不会被加载。

一个能用的工具，全文就这么长——四件事交给注册表就完了。

替换掉上一章那个文件的全部内容，重启就能用。没有省略。

SCRATCH-PLUGIN/SRC/MY-PLUGIN.TS

```
import type { Context } from '@deepseek-ai/cordis'
import { defineTool } from '@deepseek-ai/dsh-tools'
export const name = 'greet-tool'
export const inject = ['tools']
export function apply(ctx: Context) {
  ctx.tools.register(defineTool({
    name: 'greet',
    description: 'Greet someone by name.',
    parameters: {
      name: { type: 'string', required: true, description: 'The name to
greet' },
    },
    output: {
      schema: { type: 'string' },
      render: (_args, value) => [{ type: 'text', text: value }],
    },
    async execute(args) {
      return `Hello, ${args.name}!`
    },
  }))
}
```

成功标准

重启后说一句 Use the greet tool to greet Ada.，模型调用 `greet`，拿回 `Hello, Ada!`。

30.2

出参这一段

SOURCE

packages/core/tools/src/index.ts:1039

注意

这是注册当场抛的类型错误，不是运行到工具被调用时才炸。

`output` 不是可选的美化项，少写它，注册当场就抛错。

很多人照着别处的工具写法只填 `parameters` 和 `execute`，然后在启动阶段撞上一条看不懂的报错。就是这里。

`output` 里有两件事，分工很清楚：

schema
规范值的形状

`execute` 的返回值必须符合它。每次成功返回、以及被策略替换过的返回值，都会拿它校验一遍。

render
模型看到的内容

把那个规范值翻译成模型读到的文本块。它必须是纯函数，只依赖入参和这个值。

注册当场就抛

注册表在收下定义前会检查这一段：`output` 不是对象、或者 `render` 不是函数，直接抛 `tool "greet" must declare output { schema, render, presentationMeta? }`。这条错误信息很直白，但如果你在启动一大堆插件，它容易被淹在别的输出里。

常见误解

以为 `execute` 返回什么模型就看到什么。不是。模型看到的是 `render` 的产物；`execute` 的返回值先要过 `schema` 那一关。

30.3

剩下三段在干什么

SOURCE

packages/core/tools/src/schema.ts:545

下一步

ctx 上全部注册入口与拦截点，第 31 章一张图列全。

你只需要写「怎么执行」，校验、渲染、注销都不用你操心。

30.1 那段代码里，除了 `output`，其余三段各自在替你挡掉一类麻烦事。

parameters 顺带把类型和校验都给了你

声明完这一段，`defineTool` 会据此推导出 `args` 的类型，并在进 `execute` 之前替你校验模型传来的东西。模型给错参数，根本走不到你的函数里。

execute 的第二个参数带着身份与取消信号

示例里只用了 `args`，实际还有第二个参数 `exec`，带着这次调用的身份、调用方，以及一个 `exec.signal`。写长任务必须自己检查它——取消是协作式的，没人会替你把函数掐断。

register 的返回值让注册本身可撤销

它返回一个撤销函数，并自动挂在当前这个插件实例上。插件被卸载，这个工具就从模型的清单里消失，不需要你写任何清理代码。

一句话判断

工具作者只对一件事负责：给定合法入参，返回一个符合 `schema` 的值。其余每一步框架都已经接管了。

31.1

三类归位，先认清你要哪一类

第三方能挂进去的地方只有三类，先归位再翻清单。

上一部那四条路里，除了改配置那条不用写代码，剩下三条就是这一章的三页，各是一张清单。翻之前先用这张表把你要做的事归到一类——归错类，后面三页都白读。

规模

56 个服务键 = 26 个能力接缝 + 29 个核心服务 + 1 个打包层。注册入口主要落在核心服务上。

不必顺着读

只想加工具的看 31.1；要改别人行为的直接跳 31.2。

SOURCE

`docs/capability-seams.md`

这一类	什么时候用它	难度	看哪页
注册内容	你要往里加一样东西：一个工具、一段提示词、一条人敲的命令	低	31.1
拦截插话	别人的东西已经在跑，你要在它做决定之前改掉输入或结果	中	31.2
换掉实现	某项能力整体换个来源：模型、文件系统、搜索、凭据	中到高	31.3

三类的写法都是同一句——在 ctx 对象上调一个方法，拿回一个撤销函数。区别只在你站的位置：第一类是在旁边加一样东西，谁都不挡；第二类是站进别人的调用链中间，你不放行下游就不动；第三类是把某个能力的唯一实现整块顶掉，上游一行代码都不改。

到这儿就够了

这一页只做一件事：把需求归到三类里的一类。多数人以为自己要写工具，实际要的是拦截——归位对了，另外两页可以先不翻。

31.2

第一类：往里注册内容

往框架里注册内容，是三类里门槛最低的一类。

这一类的写法都是同一句：在 ctx 对象上调一个注册方法，拿回一个撤销函数。八个最常用的入口在下面。

注册入口

往框架里加东西的位置。调用返回的都是撤销函数，插件卸载时自动回收。

SOURCE

`docs/capability-seams.md`

你想做的事	入口	难度
给模型加一个工具	`ctx.tools.register`	容易
往系统提示词加一段，或加一个变量	`ctx.systemPrompt.section`	容易
加一条斜杠命令（人敲的，不走模型回合）	`ctx.commands.register`	容易
给所有命令子进程注入环境变量	`ctx.shellEnv.register`	容易
不可翻案地禁掉某个工具，或收窄子代理的工具集	`ctx.tools.guard / restrict`	中等
贡献一项技能，或接一个技能来源	`ctx.skills.registerProvider`	中等
注册一份可回放的会话状态折叠	`ctx.sessionProjections.register`	中等
加一条 HTTP 路由，或声明自己的设置命名空间	`ctx.webServer / ctx.settings`	中等

记忆方法

注册入口就是「加东西」。返回值都是撤销函数，框架在插件卸载时逆序替你撤回——这一类几乎不用你写清理代码。

31.3

第二类：挂监听器拦截

挂一个监听器，就能在别人做决定之前插话。

这是第三方最被低估的一类：不注册工具、不提供服务，纯靠监听器改变行为。官方自己就有十几个这样的包，最短的八十来行。

也能不写代码

挂一行 hook 桥接包，就能把现成的 Claude Code 或 Codex hook 配置映射到这些拦截点上。

只观察也要放行

不调 `next()` 就是主动短路。一个忘了放行的日志监听器会吃掉全体下游行为。

事件	你能做什么	形态	难度
`tools/pre-execute`	放行、拒绝、或转人工确认	拦截点	容易
`tools/execute`	环绕执行，换一个自己的超时信号	拦截点	中等
`tools/post-execute`	改写结果，或把它拦成一次失败	拦截点	中等
`tools/result`	只读观察最终结果，用于审计计量	广播点	容易
`agent/pre-step`	改写模型这一步将看到的消息	拦截点	中等
`agent/request`	替换这次模型调用的配置	拦截点	中等
`agent/request-error`	出错后决定重试、压缩、还是放弃	拦截点	中等
`session/event`	跟住整条会话事件流，做界面或桥接	广播点	容易
`fs/write-intent`	写入与编辑前的文件策略，如先读后写	拦截点	中等
`system-prompt/assemble`	整体改写系统提示词的装配结果	拦截点	较高

常见误解

以为「不返回值就等于不干预」。恰恰相反：拦截点上的监听器不调 `next()`，就是它自己拿了主意。

31.4

第三类：换掉一个能力的实现

能力接缝

只定义抽象类和词汇、不含实现的空壳包。直接挂载它会当场报错。

独占型接缝

没有按名字挑选的注册表，同一时刻只允许一个实现占位，第二个挂上去当场报 `has been registered`。

数字

26 个接缝，平均一个 1.7 个官方实现。子代理后端最多 6 个，压缩引擎只有 1 个。

接缝越窄越好换，难度基本等于要实现几个方法。

你不是加东西，而是把某个能力的实现整块换掉。依赖它的插件会干净地重启，接到你这一份上。

换什么	入口	要实现	难度
沙箱 / 外溢存储 / 工作流引擎 / 网页搜索与抓取	`ctx.sandbox`、`spillStore`、`web`	各一个方法	容易
模型适配器	`ctx.llm.registerAdapter`	一个流式方法	中等
命令执行器 / 子进程运行时	`ctx.shell`、`ctx.subprocess`	各三个方法	中等
凭据来源	`ctx.credentials`	四个方法	中等
子代理 / 语言服务 / 终端后端	`registerProvider`	一到两个	中等
压缩引擎 / 后台任务 / 文件系统	`ctx.compaction`、`jobs`、`fs`	三 / 九 / 十二个以上	较高

表里的**外溢存储**（`spillStore`）最容易被跳过：工具结果太大时先落到旁边的存储里，只把摘要给模型。

一句话判断

决定换不换实现之前，先去定义包里数一下抽象方法有几个——那个数字基本就是这件事的全部工作量。

32.1

先分清 bundle 和 profile

SOURCE

docs/user/develop/basic/publish.md

谁维护 PROFILE

你永远不用手写它的清单，`dsh plugin` 负责创建和维护。

内置包的来源

官方基础包永远来自安装本体，包管理器只管树外的包。

bundle 是你写的，profile 是用户开的，没有东西两者都是。

安装只建立在这两个概念上。两者都由一份 package.json 描述，但回答的是完全不同的问题。

问的这件事	BUNDLE	PROFILE
谁产出它	你，插件作者	用户，由 `dsh plugin` 建和维护
物理上是什么	一个 npm 包	用户目录下的一个文件夹
里面装的是	随包带的一层配置补丁	一串有序的 bundle 名
回答什么问题	这个包贡献什么	这套设置由哪些包按什么顺序组成

最终配置由四层依次盖出来，顺序见 13.1。作为 bundle 作者你只需要记住其中一条：你贡献的是最靠前的那一层，用户在后面三层里随时能盖掉你——所以默认值要选用户大概率会保留的那种。

用户也不必手写清单。`web` 和 `headless` 两个组合首次使用时自动初始化，其余的用 `dsh plugin` 创建；你的安装说明只需要给一条命令。

记忆方法

bundle 是「一层」，profile 是「一摞」。你交付的永远是一层，用户启动的永远是一摞。

32.2

目录结构与关键字段

决定它是不是插件包的，只有 package.json 里的一行。

缺了那一行，包照样装得进去，只是变成一个普通依赖：安装时打一行警告，然后什么层都不激活。

包名不受限

不必带官方前缀，那只是仓库内的约定。官方教程用的名字就是 `dsh-hello-plugin`。

模块格式

全仓一律 ESM；CJS 形态没有任何官方先例，不建议尝试。

浏览器半边

另需一组 `dsh.client` 声明和一个 `./client` 导出，两者必须同有同无。

一个 BUNDLE 的四个文件

```
hello-plugin/
├── package.json        # 声明 dsh.bundle
├── cordis.patch.yml    # 被列入组合时应用的那一层
├── src/index.ts        # 插件模块本体
└── lib/                # 构建产物，发布安装时必须存在
```

dsh.bundle.patch 决定性的那一行	指向随包的补丁文件。有它才是 bundle，没有它只是依赖。
exports / files 别漏东西	补丁文件既要导出、也要列进打包清单。漏了它，包装上也没有层。

常见误解

以为要先申请一个官方命名空间才能发插件。不用。名字随便起，起决定作用的只有清单里那一行。

32.3

安装，以及 git 安装的那份授权

从 git 装一个插件，等于允许它在你机器上执行代码。

安装命令本身很短。分歧在其中一种来源：从 git 装是唯一需要用户额外点头的路径，而这个头点下去的分量比看上去重。

四种来源都收

本地目录、npm 包名、`github:你/仓库#提交号`、打好的 `.tgz`。

要有 PNPM

这条命令是 pnpm 的一层薄转发。找不到 pnpm 就打印提示并退出 127。

```
$ dsh plugin --profile demo add ./hello-plugin
```

```
$ dsh --profile demo --dump-config
```

第一条装包并把这一层追加进组合；第二条不启动、只打印组合出来的树。

从 GIT 装会先失败一次

症状

第一次安装直接报错，说这个包的构建脚本没被允许。

原因

git 安装只拉源码、不拉产物，包必须在安装时自己构建；而 pnpm 从 10 起默认拒绝跑 git 依赖的构建脚本。

对策

作者提供一份自包含的构建脚本；用户在这个组合的工作区文件里把包名写进 `allowBuilds`，再重跑一次。

先别做

不要为了省事让用户开这份授权——它的实质是「允许这个包的代码在你机器上、在 agent 沙箱之外执行」。发 npm 或发一个打好的压缩包，两者都是预构建产物，装的时候不需要任何构建权限。

32.4

为什么官方包必须放对位置

框架必须只有一份，所以官方包一律写进 `peerDependencies`。

这不是风格偏好，是会不会跑起来的问题。ctx、服务和插件实例这一套，认的是「是不是同一份」，不是「长得一不一样」。

唯一的例外

校验器包 `@deepseek-ai/schemastery` 放普通依赖。它是运行时校验器，仓库内也这么办。

这条路被设计支持

官方源码注释里写明了遍历 `peerDependencies` 的理由：「树外插件直接 import 它们」。

把框架包写进 `dependencies`，包管理器会在用户的组合目录里再装一份。于是你的服务注册进了另一棵服务树，而你依赖的能力在这棵树上根本不存在。`peerDependencies`（同伴依赖）的意思正好相反：这个包由宿主提供，我不自带。

PACKAGE.JSON 的三段写法

```
"dependencies": { // 只有校验器例外
  "@deepseek-ai/schemastery": "^3.18.0" },
"peerDependencies": { // 框架本体和所有官方包
  "@deepseek-ai/cordis": "…" },
"devDependencies": { // 同样的包与区间再写一遍 }
```

怎么自查

装完之后，在用户组合的依赖目录里数一下框架包出现了几次。只要不是一次，问题就已经在那里了，只是还没炸出来。

一句话判断

凡是名字以 `@deepseek-ai/` 开头的，除校验器外全部写进 `peerDependencies`，并在 `devDependencies` 里用同样区间再写一遍。绝不把它们打包进自己的产物。

33.1

写代码时：两个导出陷阱

SOURCE

docs/postmortem/0001-acp-default-export-drops-inject.md

测试为什么没拦住

这两个 bug 在 178 个绿测、100% 行覆盖率下都没被发现——手工挂载的测试不走真实加载路径。

最狠的两个静默失败，都出在导出方式上。

官方有一份事故复盘专门记这两条。共同点是：代码看起来正常，类型检查也过，只在真实加载路径上炸。

陷阱一：多写了一行默认导出

症状

插件加载了，第一次读服务就抛「没有声明依赖就不能取这个属性」。

原因

加载器取导出的规则是「有默认导出就只要默认导出」。你一写它，整个命名空间被丢掉，`apply` 在一个零依赖的插件实例里运行。

对策

命名空间形态绝不写默认导出；提供服务的类形态则必须写。两者规则正好相反。

陷阱二：用点号读一个没声明的服务

症状

本地测试全过，挂进真实组合就抛。

原因

点号走属性拦截壳，只沿祖先链往上找，跨了作用域必抛。

对策

只想「有就用一下」的服务，一律改用 `ctx.get('名字')`。

到这儿就够了

不必理解背后的机制，背下来即可：命名空间形态没有默认导出，类形态必须有；机会性取服务用取值方法。

33.2

加载时：等待与拼写

SOURCE

packages/boot/app-boot/src/index.ts:692

路径必须是绝对的

补丁只贡献配置，不改变解析基准目录。

什么才是真静默

启动之后动态挂上去的插件不经过启动结算，等待态没人替你报错。

插件「什么都没发生」有两个成因，启动期都会当场报错。

新手最常撞的一类，但它一点都不安静：两种都让启动当场失败。

依赖没满足，插件停在等待态

症状

`apply` 从不执行，日志里没有任何输出。

原因

你声明依赖的服务没人提供，插件就一直等。

对策

结算会报 `did not activate`，并打出 `pending (waiting for services: …)` ——照那个服务名查谁该提供它。

模块名拼错，启动会直接失败

症状

启动中止，报 `dsh: plugin tree failed to load: failed to apply loader entry <id> (<你写的名字>)`。

原因

Loader 的挂载是事务性的：一个条目 apply 失败，整棵树回滚并抛出，启动器再冠上阶段名。

对策

报错原样打出你写的 id 和名字，直接比对拼写和路径基准。

成功标准

启动没中止，就说明每个启用的条目都建起了插件实例并激活了。

33.3

写配置时：覆盖与表达式

补丁按行整块替换，写漏一个字段就等于删掉它。

配置层是这套架构最强的地方，也最容易安静出错。两条规则都不报错。

双向都成立

用户覆盖你那一行时也要重写全部字段。所以应给出用户大概率会保留的默认值。

没人替你报错

官方仓库有脚本卡表达式那条，但它只跑在那个仓库内。

补丁不做深合并

症状

你只想改上游某一行的一个字段，改完发现那一行的其他行为全没了。

原因

后来的层按 id 整块替换这一行的配置值。

对策

把需要保留的每个字段全部重写一遍。

表达式只在两个字段里生效

症状

写了一段表达式，没有报错，但行为不对。

原因

加载器只对条目的 `config` 和 `disabled` 求值。写在 id、name、group、inject、intercept、isolate 里的表达式不会被插值，只变成一条恒为真的数据。

对策

元数据只写字面量。

先别做

不要靠「只覆盖上游一个字段」做集成。要么完整重写那一行，要么请上游把这个值做成配置项。

33.4

跑起来之后：两处改不动的限制

SOURCE

packages/host/apiproxy/src/api-proxy.ts:126 与 :256、packages/subprocess/subprocess/src/index.ts:44

官方自己也知道

白名单那段源码注释写着：把这个声明搬到注册接口上，「是待办工作」。

有两处限制不在你的代码里，改不动，只能绕。

前面几条是你写错了。这两条不是——你写得完全正确，限制在别人的代码里。

WEB 设置面板对第三方是硬编码白名单

症状

你正确调用了设置注册接口，配置页就是读不到，只回一句「未暴露」。

原因

接口转发层里有两份写死的命名空间清单，合计九个名字（另加动态发现的模型供应商命名空间）；三者之外一律拒绝。

对策

把可调项放回配置行让部署方改，或自己注册路由和界面模块自建一页。

名字里带 KEY / PASSWORD / SECRET / TOKEN 的环境变量不传给子进程

症状

模型跑 gh 或 npm 报未认证，同一条命令你在终端里跑得好好的。

原因

子进程环境被统一洗过一遍：命中那四个词的变量名，以及所有以 `DSH_` 开头的都不传下去。命令执行、MCP、语言服务、终端共用这一份定义。

对策

要转发就在那一行的显式环境配置里点名写出来——显式层在洗过之后合并。

常见误解

看到「未认证」就去查凭据配置。凭据没问题，是变量没走到子进程里。

34.1

先判断要不要写 dsh 插件

要不要写 dsh 插件，三句话就能判断。

这一页是整章的入口。绝大多数人问「我该怎么给它写插件」的时候，其实并不需要写插件。

MCP

让外部程序把工具暴露给模型的协议。挂它不用写任何 dsh 代码，也不绑定这一个 harness。

成本差别很大

MCP 服务器是另一个进程，写完到处能用；dsh 插件跑在同一进程里，能力大得多，但绑死在这套架构上。

01 **只是想让模型多几样领域工具，就去写 MCP 服务器**

查内部接口、跑数据库查询、调公司系统，这些都不用动 dsh。挂一行配置就接进来，而且同一个服务器在别的 harness 上也能用。

02 **要改执行策略、权限、或上下文行为，才需要 dsh 插件**

MCP 只能往里加工具，管不了别人的工具怎么执行。禁掉某类命令、给某些操作加人工确认、在模型出错后自动补救，这些只有插件能做。

03 **要换掉执行世界或模型层，dsh 是目前唯一能干净做到的**

把文件系统和子进程整块换成远端，读写、命令、终端、语言服务会全部跟着搬过去，上层一行不改。换模型适配器同理。

一句话判断

MCP 是「加东西」，dsh 插件是「改规矩」和「换世界」。先问自己要哪一个，再决定动不动手。

34.2

值得做的方向

值得做的方向，集中在官方留白和社区没人做的交叉区。

判断标准两条：有没有现成接缝能干净地做，官方和社区是不是都还没做。

社区严重偏科

抽样看，约六成扎堆在界面增强和前端形态替换上，模型层扩展接近于零。

两个明确的空插槽

上下文压缩引擎和语言服务，官方各只有一个实现。

怎么判断一个插件真不真

看它的 `peerDependencies` 里有没有官方包，比看星标靠谱得多。

方向	挂在哪	为什么值得
接自建或第三方模型	`ctx.llm.registerAdapter`	只需实现一个流式方法。社区几乎无人做
企业内网检索	`ctx.web.registerSearchProvider`	模型看到的工具名不变，换的是背后来源
组织策略网关	`tools/pre-execute` 加 `guard`	官方指南里有可直接抄的完整例子
审计与合规出口	`ctx.sessionTelemetry`	这个接缝在进程内没有消费者，旁路最干净
团队级技能来源	`ctx.skills.registerProvider`	官方只有文件系统与内置徽章两个实现
企业密钥托管	`ctx.credentials`	四个方法，每次操作重新取值，支持轮换
远程执行世界	`ctx.fs` 加 `ctx.subprocess`	官方远端沙箱三件套是完整模板

先别做

第一个插件别从浏览器侧起手。先做主机侧——工具、策略、能力实现、提示词、命令，一个下午能上线；浏览器侧是另一个量级。

34.3

不适合做的方向

有七类方向不该动手，原因不是难，是这套结构接不住。

它们都是真需求。前六类结构上给不了；最后一类给得了，但没有契约接住你。

不是「难」的问题

这一页列的不是工作量大的方向，是这套结构接不住的方向。

沙箱接缝的边界

它的策略词汇里只有文件效果，不表达网络、进程、系统调用、设备和凭据的任何限制。

想做的	为什么做不成
不可信第三方插件市场，或带沙箱的插件运行时	插件与主程序同进程、无权限模型。工作线程和虚拟机不是安全边界
给插件加网络限制，或把容器与微虚拟机做成沙箱后端	沙箱接缝只表达文件效果；容器与微虚拟机被官方排除，正确做法是整组替换文件系统和子进程
做一个运行时的插件启停管理面板	插件清单是只读投影，不能启用、禁用、增删；插件集变更要重启
让自己的配置项出现在 Web 设置页	硬编码白名单，见 33.4
多个工作流引擎并存，或多套会话命名策略并存	这两个接缝是独占型，没有按名字挑选的注册表
依赖会话日志磁盘格式的外部分析工具	格式版本停在 0 且无兼容承诺。正确姿势是写一个遥测实现
换掉整个主循环	能摘能换，但没有契约接住你——等于把 dsh-agent 的事件与服务重新实现一遍。是一行配置，不是一个接缝

常见误解

以为工作线程或虚拟机隔离能当安全边界。官方在至少八个包里主动否认，措辞是「不是安全沙箱」「把动态包当作 bash 访问那样对待」。

35.1

交付前的验收清单

这十条全过了，你的插件才算能交出去。

前面几节的所有坑，压缩成一张可以逐条打勾的表。它们的共同点是：不过也不报错。

怎么用这张表

逐条过。任何一条不过，插件在别人机器上就有相当概率静默不工作。

再加一条自保

README 里写明你验证过的具体 rc 版本，版本区间给一个上界。这套东西明文承诺会破坏兼容。

检查项	怎么算过
导出形态	命名空间形态没有默认导出；提供服务的类形态有
依赖位置	官方包全在 `peerDependencies`，`devDependencies` 同区间再写一遍
版本标签	装的是显式版本或 next，不是默认标签（成因与自查见 37.2）
包声明	有 bundle 声明；模块格式是 ESM；补丁在打包清单里
浏览器半边	两处声明同有同无
覆盖别人的行	需要保留的每个字段都完整重写了
表达式位置	只出现在配置和禁用两个字段里
监听器放行	拦截点上只观察的监听器也调了 `next()`
服务命名	自定义服务名带了前缀，框架已占 56 个键
装上能看见	组合树里有你那一层，启动日志没有未解析服务名

成功标准

在一台干净机器上从 npm 装一遍你的包，打印组合树看到你的层，起一次会话让模型真的调用你的工具。三步都过，才算交付。

35.2

五条常见问题

五个问题，覆盖八成的「装上了，但不对劲」。

顺序有讲究

先确认层在不在，再确认服务到没到，最后才怀疑模型。

关于工具收窄

按作用域限制可见工具集是可见性组合，不是权限边界。要不可翻案的禁止，用 guard。

01 加载成功了，模型就是不调我的工具

模型不读你的代码，只读工具描述。把描述写成「什么时候该用它」，再补一段顺序在 100 到 199 之间的提示词片段。

02 界面上什么都不出现

先查浏览器半边的两处声明是不是只写了一半——那种情况什么都不贡献，也哪儿都不报错。

03 工具加得越多，效果反而越差

每个工具的结构声明都会进提示词，抢模型的注意力。少而准优于多而全，真需要很多就按作用域收窄。

04 挂上去毫无动静，也不报错

启动期不会有这种事：依赖没满足和模块名拼错都会让启动中止并报出名字。真安静就说明它是启动之后动态挂的。

05 什么时候该从临时覆盖层转成正式包

三个信号任一出现就转：要给别人装、要跟着版本走、要带一组默认配置。

一句话判断

五个问题有四个的根因是同一条：你写的东西和真实加载路径不一致。先打印组合树，再看服务名，最后才怀疑模型——这个顺序能省掉大半个下午。

VII

这东西值不值得你投入，投到什么程度，什么时候该抽身

PART VII

判断

前面六部讲的是它是什么、怎么装、怎么转、能扩展什么。这一部只回答最后一个问题：这东西值不值得你投入，投到什么程度，什么时候该抽身。

36.1

把选型问题拆成四层

提醒

层号只是本书为了讲清楚而分的，不是官方术语。

「谁更强」问不出答案，「你要控制哪一层」才问得出来。

任何一个 agent harness 都可以拆成四层。你在哪一层需要说了算，就决定了你该选谁。

01 **模型层**

换厂商、换模型、调思考强度档位。主流产品都给了这一层的旋钮。

02 **循环与工具面**

模型看得见哪些工具、什么时候停、哪些动作要人点头。主流产品给有限的钩子。

03 **能力实现层**

bash 在哪台机器执行、文件读写落到哪、会话存进哪个库。主流产品把这层焊死了。

04 **界面层**

人怎么看到过程、怎么中途插手。终端、Web、编辑器插件，各家形态不同。

dsh 的差异只在第三层：`ctx.fs`、`ctx.shell` 被做成 26 个可整块换掉的接缝。一、二层与别家相当，第四层更弱。

一句话判断

需求停在一、二层就选成熟产品；落到第三层（自研沙箱、自建存储、自己的审批流）才轮到 dsh。

36.2

和 Claude Code / Codex 的公允对比

已删除

`packages/ui/tui` 于 2026-08-04 整包删除，不留兼容包也不留别名。

注意

竞品一列取自公开资料，本书未读其源码，标注「未核实」的条目请以官方文档为准。

它缺的是给人坐在终端里聊天的那层壳，不是自动化入口。

网上流传的「dsh 只有 Web，没有命令行」是错的。它有官方命令行启动器，也有一次性执行模式。

对比项	CLAUDE CODE / CODEX CLI	DEEPSEEK HARNESS
日常交互面	成熟的交互式终端	只有 Web 界面
命令行	有	有：`dsh --profile`
一次性执行	有	有：`--profile headless "任务"`
扩展点	钩子、MCP 与官方扩展机制（未逐项核实）	整棵插件树，任意一行可替换
换执行世界	未见整块替换 fs / shell 的公开接缝（未核实）	能，换两个接缝就整体搬走
源码开放	Codex：Apache-2.0 可 fork；Claude Code：闭源	MIT 全开源
外部修复通道	Codex 接受 PR；Claude Code 可提 Issues	不接 PR，无 Issues 入口
稳定性承诺	有版本与发布节奏	只有预发布 rc，无 CHANGELOG、无兼容承诺

常见误解

以为「开源」就等于「最容易改成自己的形状」。后两行别跳过：在这一维上，可 fork、可提 PR 的 Codex 未必输给 dsh。

36.3

关系，而不是对立

SOURCE

packages/subagent/、packages/hooks/

补充

被调用的那一侧读它自己的登录态和配置，dsh 不复制也不过滤。

它们可以待在同一条工作流里，各占一层。

仓库里真实存在把竞品当插件用的包。这是它对自己位置的声明：编排层，不是又一个终端 agent。

dsh 调别人

`subagent-claude-code` 走官方 Agent SDK，`subagent-codex` 起 `codex app-server`，`subagent-acp` 接任意 ACP agent。它们都以子代理身份出现。

别人调 dsh

ACP server、JSON-RPC、headless 三个入口。但 ACP 那一面自陈是仅供自动化的传输适配器，不是完整的编辑器集成。

`hooks-claude-code` 和 `hooks-codex` 是桥接器：指向你已有的 `hooks.json`，让那些 shell 钩子原样跑起来。`mcp-client` 用的工具命名也是 `mcp__<server>__<tool>`，跟两家一致。已经攒下的这些资产，换过来不用重写。

一句话判断

不必二选一。让成熟产品管日常敲代码，让 dsh 管你自己那套执行环境和审批流，两者在同一条流水线上。

37.1

书面记录不构成承诺

SOURCE

AGENTS.md、docs/capability-seams.md

快照

版本与发布状态取自 2026-08-14。

它写下来的东西都不构成承诺——版本没有，文档也已经对不上了。

仓库里没有 CHANGELOG，没有任何兼容承诺，也没有划出哪些是公开 API。

发布通道本身是有的——`release.yml` 从 `dsh-v*` 标签手动触发发布，带 rc 段的版本一律进 `next` 标签，vendor 那条序列另有自己的标签与工作流。但那是发版机制，不是稳定性承诺。

> With no external consumers, prefer the correct foundation over compatibility shims: rename or repackage freely and update every reference together. Backends reject old on-disk formats.
>
> AGENTS.MD · PRE-RELEASE STANCE

大意：因为没有外部使用者，宁可把地基做对也不做兼容垫片；可以随意改名和重新打包，后端直接拒绝旧的磁盘格式。

文档走样已经开始了。生成的能力接缝表里，`ctx.lsp` 的实现写作 `lsp-local`、`ctx.codeRuntime` 写作 `code-runtime-worker`，而仓库里这两个包实际叫 `lsp-stdio` 和 `code-runtime-worker-thread`——改名发生在 2026-08-11，开源在两天后，文档没跟上。源码注释里还留着指向已不存在目录的链接。

常见误解

以为生成出来的文档表就是源码的镜像。把文档当线索，把源码当事实：任何要写进你自己代码里的名字，都去仓库里再对一遍。

37.2

npm 标签陷阱

SOURCE

packages/core/tools/src/index.ts:466

兜底自查

装完在 node_modules 里数一遍 dsh-tools，两次就是踩中了（见 32.4）。

后果

当次运行中断，日志里留下没有结果的孤儿调用；会话不必丢弃——重新加载时 session-persistence 会给孤儿调用补上错误结果。

照默认标签装依赖，你会装到 8 月 10 日那一版。

`@deepseek-ai/dsh` 的 latest 指向 0.1.0-rc.6，但几个子包的 latest 还停在 0.0.1-rc.1，next 才是新版。

01 **装进来两份物理拷贝**

插件按 latest 拉到旧版 `dsh-tools`，宿主用新版。同一个包，磁盘上两份。

02 **调度器的钥匙对不上**

`TOOL_RUNTIME_SCHEDULER` 是模块内的 `Symbol()`，不是 `Symbol.for()`。两份拷贝的符号永远不相等。

03 **当前这一步失败**

取到 `undefined`，工具调度抛错。日志里的孤儿调用等会话重新加载时由持久化层补齐。

这是纯 registry 状态，读到这里可能已经修好了，自己复核一次：

```
$ npm view @deepseek-ai/dsh-tools dist-tags
```

2026-08-14 实测返回 `latest=0.0.1-rc.1`、`next=0.1.0-rc.6`。

先别做

不要用默认标签装任何 `@deepseek-ai/dsh-*`。显式写 `@next` 或钉死具体版本号，并且一律放进 `peerDependencies`。

37.3

最严重的单条部署风险

SOURCE

packages/host/webserver/README.md

只有两档

配置里 host 只接受 `127.0.0.1` 和 `0.0.0.0`，没有中间态。

把 Web 界面绑到 0.0.0.0，等于把本机 shell 开给整个网段。命令行那一层已经拦了一道——`--host 0.0.0.0` 会直接报错退出（9.4）；但配置层的 host 只有 127.0.0.1 和 0.0.0.0 两个合法值，改配置文件仍然绑得上。也就是说走到这一步需要一次刻意的动作，而不是手滑。

这不是猜测，是官方在自己的 README 里写明的已知限制。

> No TLS, auth, or origin policy — binding a non-loopback address exposes the server to that network.
>
> HOST/WEBSERVER README · KNOWN LIMITATIONS AND DEFERRED WORK

大意：没有传输加密、没有身份认证、没有来源校验；绑一个非回环地址，就等于把这台服务器暴露给那个网络。

要理解这条有多重，得把它和背后的东西连起来看：连上这个端口的人，指挥的是一个握有本机 shell（`ctx.shell`）、能读写全盘文件（`ctx.fs`）、能起子进程的 agent。没有登录页，没有令牌，没有来源检查——端口通了就是全部权限。

官方把加固明确划在范围之外，说这是「面向开发者的 v1」，让你自己放到真正的反向代理后面。

一句话判断

默认的 `127.0.0.1` 就是它唯一安全的姿势。要给第二个人用，先有反向代理和鉴权，再谈改 host。

37.4

扩展者会撞上的两堵墙

想扩展它，你会先撞上两堵墙。

两堵都不是 bug，是这一版有意留下或尚未处理的边界。

SOURCE

CONTRIBUTING.md、八个包的 README

补充

Worker 与 node:vm 只约束模型写的代码，不约束插件。

墙一 · 插件没有任何隔离

插件跑在主进程，拥有宿主的全部权限：起进程、连网络、读写用户文件，一律不受限。安装路径是把参数原样转给 `pnpm`，没有签名、没有审核、没有能力声明。装一个插件，等于允许它以你的身份执行任意代码。官方自己也不掩饰：至少八个包在 README 里明写自己不是安全边界——`code-runtime`、`code-runtime-worker-thread`、`workflow`、`workflow-worker-thread`、`tool-cordis`、`cordis-host-runner`、`cordis-client-runner`、`fs-sandbox`。

墙二 · 组织通道是单向的

官方把反馈入口指向 GitHub Discussions 和 Discord，README 里没有 Issues 入口；贡献指南原文写着目前无法接受外部 PR。发现核心缺陷你只能发讨论帖，或者自己 fork 绕过。

还有一堵矮墙，成因见 33.4：设置面板只服务源码里两份写死的白名单（7 项 Web 设置 + 2 项产品设置）外加模型服务商动态注册的那几个，此外的命名空间即使注册了配置 schema，页面也只回答 `settings-not-exposed`。

常见误解

以为装插件像装浏览器扩展、后面有个沙箱兜着。没有。它是给「部署方自己选装的可信插件」用的平台，不是给「任意第三方上传」的插件市场用的。

37.5

遥测与数据出境

遥测默认关着，但默认端点写死指向 DeepSeek 的域名。

出厂挂着 `session-telemetry-otel`，模式取自 `DSH_TELEMETRY_MODE`，三档差别是硬的。

SOURCE

`packages/bundle/base/cordis.patch.yml:129`

硬开关

`DSH_TELEMETRY_DISABLED` 只要非空就生效，写 `0` 或 `false` 一样是关掉。

不含 KEY

厂商 API key 是构造参数，从不进会话日志，也就不会被导出。

档位	行为
DISABLED	出厂默认。不构造 SDK 管线，一条记录都不出进程
FEEDBACK_ONLY	你点了「反馈」，才把此前那段会话日志回放上传
FULL	每条会话事件产生即交给 OTel SDK

端点硬编码：`exporter.url` 默认取 `https://harness-telemetry.deepseeksvc.com/v1/logs`，只有 `DSH_TELEMETRY_OTLP_URL` 能覆盖。要整台机器断掉，用 `DSH_TELEMETRY_DISABLED`——启动器在加载之前就把这一行 patch 成禁用，压在所有组合层之上：配置里怎么开，这个环境变量都能一票关掉。

开了就是全量

上传消息正文、工具参数与结果（命令输出、文件内容）、系统提示与工具 schema、压缩摘要、会话工作目录。接缝上留了 `sessionTelemetry/record` 脱敏拦截点，但仓库不带任何一条规则——没人挂规则，记录就原样离开进程。

先别做

别把「默认关闭」当成「不用管」。要么在部署环境钉死 `DSH_TELEMETRY_DISABLED`，要么开之前先挂上自己的脱敏规则。

37.6

退出成本

硬事实

会话日志格式版本停在 0，且无兼容承诺。

现在就算清楚你要怎么退出来。

投入越深，绑住你的东西越不可迁移。这三档的差别很大，值得在动手前分清。

投入深度	被绑住的是什么	退出成本
只当工具用	一个家目录和一批会话日志	几乎为零
写策略与工具插件	注册入口与拦截点的名字、接缝契约；业务逻辑可以留下	中等
换执行世界或做 UI	Cordis 的生命周期语义和前端插槽契约	很高

会静默出错

不要写依赖会话日志磁盘格式的外部分析工具。格式版本保持在 0、无兼容承诺、旧格式被后端直接拒绝。要取数据就走遥测接缝，让它把记录送出进程——端点与脱敏见 37.5。

三档里真正决定退出难度的不是代码量，是你把多少判断写进了框架的语义。第二档的插件文件里如果只剩注册与拆卸，业务逻辑整份都能搬到别的 harness 上；第三档一旦用上 Cordis 的加载顺序、插件实例的逆序撤回、前端插槽的生命周期，这些语义在别处没有对应物，只能重写。

一句话判断

把不可迁移的东西控制在「挂载层」那一个文件里，业务逻辑写成普通 TypeScript 模块，退出就只是重写挂载层。

37.7

许可与再分发

SOURCE

THIRD_PARTY_NOTICES.md、vendor/README.md

提醒

本书不提供法律意见，这里只列你能交给法务的原始材料。

许可这一侧没有拦路的东西，而且清单是机器生成的。

要向法务解释为什么可以内网部署，需要的三份材料仓库里都有，且都能复核。

要回答的问题	去哪里拿
本体什么许可	`LICENSE`：MIT
整套复制进来的 Cordis 呢	九个包全是 MIT，各自目录保留上游 `LICENSE`，只把包名重挂到 `@deepseek-ai` 域下
第三方依赖清单	`THIRD_PARTY_NOTICES.md` ——由脚本从工作区清单生成，提交钩子重算、测试断言字节一致
完整传递闭包	`pnpm-lock.yaml`（`pnpm licenses list` 可列），Python 那侧在 `python/sdk/uv.lock`

vendor 下登记着 18 处本地修改不影响再分发：`vendor/README.md` 要求这份修改日志必须穷尽，18 条逐条写明改了哪个文件、为什么改；改动落在 MIT 代码上，MIT 本来就允许修改后分发，只要保留许可与版权声明——而每个目录的上游 `LICENSE` 都还在。

到这儿就够了

许可这一侧到此为止，不必再往下查。真正要交给法务的不是许可证，是 37.5 那条默认遥测端点。

38.1

四种目的，四个答案

提醒

这四条建议基于 0.1.0-rc.5，会随版本变化。

同一个问题问四遍，会得到四个不同的答案。

「值不值得用」没有统一答案，取决于你打算拿它干什么。

01 **研究架构 · 直接读，性价比最高**

它是目前公开的 harness 里把内部结构摊得最开的一个。用 `--dump-config` 把实际启动的整棵树打印出来，对着读。

02 **做实验插件 · 做，但按展示品做**

官方只给了一两个内置实现、社区又几乎没人做的方向最值得占位：模型适配器、压缩引擎、LSP provider、遥测出口。不要给它排上线时间。

03 **内网给团队共用 · 可以，但先解决鉴权**

反向代理加身份认证是前提，版本全部钉死，遥测端点当场确认。没有这三条就别开给第二个人。

04 **投生产 · 现在不要**

四条各自独立的理由：没有版本承诺、服务端没有鉴权、没有外部修复通道，以及谁也拿不出成本与效果数据——下一页专讲这条空白。

一句话判断

把它当研究对象和实验平台是划算的；当生产依赖，现在还不划算。

38.2

成本与效果的空白

SOURCE

BENCHMARK.md（全文三行）

态度

写明空白，好过让你翻到最后才发现没有。

「它比别家贵多少、做得怎么样」——本书没有数据，官方也没有。

这是本次评估的已知空白，不是漏写。拿它进选型会之前，这一组数字得你自己产出。

官方那一侧是空的。仓库根目录确实有一个 `BENCHMARK.md`，但全文只有三行，讲的是怎么装 Python SDK、跑哪个最小变体、每个基准任务要用独立的工作区和会话 ID。没有任务集，没有对照组，一个数字也没有。

市面上流传的数字同样不能用。全书唯一出现过的一条（见 5.2）转述自一家中文媒体，说同款模型接别的框架只消耗它的三成多——单一来源、未交叉验证，本书不为它背书，也不建议你拿去做预算。

为什么不能随手测一下

同一个任务的 token 用量，取决于这套组合装了哪些工具（工具 schema 每一步都进请求）、系统提示词多长、压缩阈值定在哪、有没有开 Code Mode。两家默认配置不同，直接对跑等于在比配置，不是在比框架。

真要测，最小口径是这几条：同一个模型；三个代表性任务，每个各跑三次取中位数；同时记 token 用量和墙钟时间；并把两边的工具集大小与系统提示词长度一起记下来。少了最后一条，数字换个人就复用不了。

先别做

在你自己那组数字出来之前，不要用本书或任何媒体的数字去做预算，也不要用它去说服别人。

38.3

自保清单

顺序

前三条护版本，后三条护你自己的代码。

如果你还是要动手，这六条现在就做。

每一条都对应前面已经出过事的一个坑，不是泛泛的最佳实践。

01 所有 `@deepseek-ai/*` 一律放进 `peerDependencies`，绝不打包进你的产物。

02 显式写 `@next` 或钉死版本号，版本区间给一个 `<0.2.0` 的上界。

03 README 里写明你验证过的 rc 版本和 commit，让用它的人知道基线在哪。

04 只用文档化的注册入口与拦截点，避开需要改核心才能走通的路径。

05 业务逻辑写成普通 TypeScript 模块，插件文件里只留注册和拆卸。

06 服务端 host 保持默认的回环地址；要开放先有反向代理和鉴权。

成功标准

上游发一个新的 rc，你只需要改挂载层那一个文件，业务逻辑一行不动。做到这条，前面五条就都做对了。

38.4

重新评估的触发信号

做法

这五条你都推不动，设个季度提醒去看一眼即可。

出现下面任意一个信号，就该把结论重算一遍。

今天的「不要投生产」是对 2026 年 8 月这个状态说的，不是永久判决。

信号	看哪里
出现兼容承诺	仓库有了 CHANGELOG，或写明的 semver 政策与公开 API 划线
服务端有了鉴权	webserver 那条「无 TLS、无认证、无来源策略」的已知限制消失
设置页对插件开放	两份硬编码白名单被换成由插件自己注册
组织通道打开	开始接受外部 PR，或官方给出 Issues 入口
有了分发支持	官方脚手架、插件目录或兼容性矩阵之一落地

反向信号同样要看：连续数月没有新提交、讨论区堆积无人回应，说明这个项目的优先级变了，那时候该重算的是另一个方向的结论。

一句话判断

把这五条抄进你的季度提醒里。它们变了，这本书的结论就该重算一遍。

38.5

这本书没讲什么

为什么写这页

知道地图缺哪一块，比以为地图是完整的安全。

这本书有五处明确的空白，现在告诉你它们在哪。

不是「暂未涉及」，是我们清楚它重要、这一版确实没做。

01 **ACP 与 JSON-RPC 两个入口的实际用法**

书里只交代了它们存在、各自是什么定位，没有给出一次从头到尾的接入。

02 **MCP 两侧的配置**

讲了它的工具命名和两家对齐，没讲客户端与服务端的配置文件到底怎么写。

03 **界面那 39 个包怎么组织**

27.1 只给了个数字就走了。浏览器那半边是公认最难的一级，本书没有拆开。

04 **换成非 DeepSeek 模型的实际效果**

提示词、工具集、思考强度档位都是围着自家模型调的：base 默认 `deepseek-v4-flash`，headless 的角色设定直接写 `{{model}}`。17.2 讲了适配器怎么写，没讲换完掉不掉链子。

05 **成本与效果的数字**

38.2 专讲这条：官方没有，本书也没有。

到这儿就够了

正文到此结束。后面四份附录是查表用的，不必顺读。

A.1

框架与组装

用法

正文里每个词第一次出现都解释过，这里只做回查。

先是这七个词，讲的是它由什么拼起来。

由大到小排：从最外层的 harness，一路收窄到只管一个会话的 preset。

harness 智能体外壳	让模型在真实环境里干活的那层程序：给上下文、给工具、跑循环、记日志。
Cordis 插件内核	dsh 用的插件框架，从 Koishi 生态搬进仓库，版本 4.0.1。
context / ctx 上下文	插件拿到的那个对象，同时是服务仓库、事件总线和副作用账本。
service key 服务键	`ctx` 上的一个名字，如 `ctx.fs`。全仓共 56 个。
capability seam 能力接缝	声明了抽象接口、可整块换掉实现的服务键，共 26 个。
bundle / profile / patch 三层组装	零件包、装配单、改单。三层叠出实际跑的那棵树。
preset 会话预设	给单个会话换一套能力的配置，随包四个，也可以自己写。

A.2

插件之间怎么连

用法

这七个词是第五、六两部的通用词汇。

再是这七个词，讲的是插件之间怎么连。

前三个说一个插件自己是什么，后四个说它怎么跟外面接上。

plugin
插件

配置树里的一行，加上它挂载的那段代码。在这里它比通常理解的重得多。

effect / disposer
可逆副作用

可撤销的注册。卸载插件时按反序执行，不留痕迹。

inject
依赖声明

插件声明自己要用哪些服务，加载顺序由它推导。

拦截点
waterfall

可以短路的事件链。主要拦截点都建立在它上面——`agent/pre-step`、`agent/request`、`llm/stream`、三个`tools/*`、`approval/request`；`agent/turn-stopping`是例外，它是没有`next()`的 serial 事件。

三角色
接缝的构成

定义接口的、实现接口的、使用接口的。三者齐了才叫接缝。

Code Mode
代码模式

工具不逐个暴露，模型写一段程序去调，入口是`run_code`。

subagent
子代理

被主代理委派的下级，可在进程内，也可以是外部 agent 程序。

A.3

运行与会话

最后这七个词，讲的是它跑起来之后发生什么。

用法

这七个词都出现在第四部的旅程里。

日志是唯一真相，其余六个词里有四个是它的投影。

agent loop
主循环

发请求、收工具调用、执行、再发的那个循环。它能摘能换，但没有契约接住你：是「一行配置」，不是「一个接缝」。

turn / step
回合与步

一次请求响应叫一步（`step`）；从一次用户输入到它停下来叫一个回合（`turn`）。

session event
会话事件

会话日志里的一条记录，共 44 种类型。

surface
遮罩后的那层视图

盖在日志之上的一层视图，模型看到的是它而不是原始日志。仓库里这层的标识符就叫 `surface`（`session.surface`、`surfaceOp`）。

projection
投影

从日志纯函数算出来的视图：模型历史、界面、标题都是它。

compaction
压缩

上下文快超限时把旧内容折叠起来的机制。

MCP / ACP
两个协议

MCP 解决代理接工具，ACP 解决编辑器接代理。dsh 两边都实现了。

B

26 个可替换的能力接缝

这 26 个位置，你可以整块换掉实现。

SOURCE

docs/capability-seams.md

读法

每个名字前都省略了 `ctx.` 前缀。

fs	文件系统
shell	命令执行
subprocess	子进程
terminals	持久终端
sandbox	进程沙箱
codeRuntime	代码执行
llm	模型适配
web	搜索与抓取
lsp	语言服务
skills	技能来源
subagents	子代理
jobs	后台作业
workflowEngine	流程引擎

sessionPersistence	会话落盘
sessionQuery	会话检索
sessionTitle	会话标题
sessionTelemetry	遥测出口
storage	通用存储
attachments	附件存储
spillStore	外溢存储
compaction	上下文压缩
settings	用户设置
credentials	凭据托管
approval	审批决策
userQuestions	向人提问
directoryPicker	选工作区

这 26 个的难度差得很远：`sandbox`、`spillStore`、`workflowEngine` 各只有一个抽象方法，`fs` 有十二个。想动手，就从只有一个方法的那几个挑。

C

52 个工具，装在 24 个包里

SOURCE

docs/tool-catalog.md

读法

包名省略了 @deepseek-ai/dsh- 前缀；这份目录是生成的，不是手写的。

仓库内置的工具一共 52 个，但没有哪套组合会装全。

```
包 → 模型可见的工具名

tool-bash               bash
tool-pwsh               pwsh
tool-bash-persistent    bash（持久 PTY 版）
tool-fs                 read / write / edit / read_image
tool-str-replace-editor str_replace_editor
tool-fs-search          glob / grep
tool-terminal           terminal_open/close/list/read/send/signal
tool-web                web_search / web_fetch
tool-lsp                lsp
tool-jobs               job_list / job_output / job_kill
tool-todo               todo_write
tool-goal               create_goal / get_goal / update_goal
tool-skill              skill
tool-workflow           workflow
tool-ralph              ralph
tool-subagent           subagent
tool-subagent-control   list_agents / send_message / interrupt_agent
tool-subagent-report    report
tool-session-query      session_search / session_trace
                         session_event_read/search/trace
tool-ask-user           ask_user_question
tool-cordis             cordis_define/undefine/run/stop
                         cordis_inspect_self/list/query
schedule                schedule_create / schedule_list / schedule_delete
plan-mode               exit_plan_mode
tools                   run_code（Code Mode 的保留通道）
```

要知道自己那套组合到底能看到几个：web 这一层补丁关掉了 24 行，2.2 举了其中一部分；要精确到行，读 `packages/bundle/web-app/cordis.patch.yml`。

D

这份文档是怎么做出来的

基线

源码 0.1.0-rc.5；网络快照 2026-08-14。

书里的数字来自源码，判断来自我们，两者分开写。

读者有权知道每一句话的可信度从哪来。

三条纪律

一、数字必须能复算。219 个包、56 个服务键、26 个接缝、52 个工具、44 种会话事件，全部来自仓库的生成产物或直接计数。

二、源码压过文档。文档和源码对不上的，以源码为准，并在正文里点出差异。

三、推测必须标明。我们的推断在正文里写了「推测」二字；没写的都能在仓库里找到出处。

已知的不确定

npm 周下载量取不到，star 数的构成无法独立验证；社区插件仓库的真实数量只能估计；中英文官网对运行模式的命名不一致，本书以英文原文为准；npm 上的 latest 与本书的源码基线不是同一版。还有一处是有意留白：本书没做成本与效果实测，官方也没给 benchmark 数据（见 38.2）；另有五处内容缺口列在 38.5。

所以这本书是地图，不是合同。这个项目每天都在动，真要动手，以你当天 checkout 的那份源码为准。

换掉一层，
比说服一层容易。

这大概就是 DeepSeek Harness 想说的全部。它做得对不对，要看接下来有没有人真的去换。

出品

废才俱乐部

基线

DEEPSEEK-HARNESS
0.1.0-RC.5