DeepSeek Harness

白皮书

从 0 到 1 玩转 DeepSeek Harness 的新手百科全书

版本 0.1.0-rc.6 · 2026-08-13 · 14 章 + 附录 A/B/C

目录

学习路径：从 0 到 1 的 3 天计划

*承诺兑现：**任何开发者，3 天从零到能开发插件。** 本路径把 9 章拆成 3 天，每天有明确目标与验收标准。*

Day 1：理解 + 跑起来（第 1-2 章）

目标：理解 dsh 是什么，能独立启动、对话、配置。

时间	任务	验收	
30 分钟	读第 1 章（三个直觉 + 能力矩阵）	能向别人解释"dsh 和 Claude Code 的区别"	
30 分钟	第 2 章动手：dsh web 对话 + headless	能启动 UI 完成一次对话；能跑一条 headless 命令	
30 分钟	配置模型/推理档位 + 排障速查	能改 settings.yaml；能解决端口占用	

验收：dsh --profile headless "你好" 有回复。

Day 2：理解骨架 + 第一个插件（第 3-4 章）

目标：理解 profile/插件机制，写出第一个能跑的插件。

时间	任务	验收	
1 小时	第 3 章（profile、挂载、host/client、扩展点、坑）	能挂载社区插件（Git 面板）	
1.5 小时	第 4 章抄写示例提速插件：纯函数 + agent/request 注入	插件加载无错，日志出现决策输出	

验收：自己的插件在 dsh 里生效（日志证明注入发生）。

Day 3：实战 + 调优（第 5-6 章）

目标：能评估插件质量、调优性能、理解生态玩法。

时间	任务	验收	
1 小时	第 5 章三个真实 PR（安全红线、沙箱约束、测试方法）	能说出"写一个 PR 的完整闭环"	
1 小时	第 6 章性能模型（思考占 90%）+ 档位策略 + 坑	能给一个任务选档位、预估瓶颈	

30 分钟	第 8-9 章扫读（工具/上下文、MCP/子代理/工作流）+ 术语表	知道 dsh 的能力全貌与扩展方向	

验收：能给 dsh 生态贡献一个小改进或写一篇使用心得。

之后：按需深入

想做什么	去读	
接外部工具	第 9 章 MCP + 官方 mcp 包文档	
并行任务	第 9 章子代理	
确定性流程	第 9 章工作流 + 官方 examples	
生产部署	官方架构文档 + 安全模型（sandbox/权限）	
性能极致	第 6 章 + 第 4 章示例源码	
生态变现	第 7 章 + dsh-plugin topic	

学习原则（白皮书全程）

1. **动手 > 阅读**：每章命令都跑一遍，不看会后悔
2. **先复制后理解**：第 4 章完整代码先跑通，再改参数理解机制
3. **单测 + 实机双证据**：开发插件时两个都要过
4. **善用 --dump-config**：疑惑时看合成配置，比猜快

第 1 章：认识 DeepSeek Harness

*本章目标：从一个完全零基础的角度，建立对 DeepSeek Harness（dsh）的完整认知——**它是什么、为什么值得学、和主流 Agent 有什么区别、什么时候用它**。读完本章，你不需要任何前置知识，就能理解 dsh 在整个 AI 开发工具版图中的位置。*

TL;DR（本章核心，30 秒版）

1. **dsh = Agent 的乐高底座**：官方开源（MIT）的运行时，一切能力都是插件，你可以自由拼装
2. **harness = 模型外面的工程层**：会话、工具、上下文、循环控制——让模型在仓库里真正干活
3. **和 Claude Code 的区别**：Claude Code 是"整车"，dsh 是"底座 + 积木"——可定制面完全不同
4. **生态窗口**：2026-08-13 开源，中文教程此前为零——现在入场是早期
5. **谁该用**：要深度定制/玩生态/跑 CI 的开发者；要开箱即用的选 Claude Code

<!-- [fix] 结构核验：补齐本章导航（第 1 章原缺失，与第 3-11 章结构保持一致） -->

<details><summary>本章导航</summary>

- [1.1 先建立三个直觉](#11-先建立三个直觉)
- [1.2 官方定义与核心事实](#12-官方定义与核心事实)
- [1.3 架构是怎么"一切皆插件"的](#13-架构是怎么一切皆插件的)
- [1.4 DSH 与主流 Agent 的全面对比](#14-dsh-与主流-agent-的全面对比)
- [1.5 什么时候用 dsh（选型决策）](#15-什么时候用-dsh-选型决策)
- [1.6 常见问题（FAQ）](#16-常见问题-faq)

</details>

1.1 先建立三个直觉

在讲技术之前，先用三个类比建立直觉：

① dsh 是什么？——它是"Agent 的乐高底座"。

想象乐高：官方提供底板和标准积木（运行时 + 核心插件），你可以自由拼装（加插件、换界面、改行为）。对比之下，Claude Code 更像"一辆整车"——很好开，但你想改装发动机得找官方。

② 为什么需要"harness"这个词？——它是"套在模型外面的工程层"。

一个模型（DeepSeek V4）本身只会"回复文字"。要让它在你的代码仓库里干活（读文件、跑命令、改代码、多轮循环），外面需要一层工程：会话管理、工具调用、上下文控制、错误恢复。这层工程就叫 harness。dsh 是 DeepSeek 官方把这层工程开源出来的产品。

③ 为什么 2026 年才开源？——因为 Agent 进入"可编程时代"。

2025 年是"模型能力竞赛"（谁能生成更好的代码）；2026 年进入"Agent 工程竞赛"（谁能更好地组织模型干活）。DeepSeek 开源 harness 的战略意图：把"如何组织 Agent"这件事变成开放生态——像当年 Android 开源改变手机生态一样。

1.2 官方定义与核心事实

一句话定义：DeepSeek Harness（dsh）是 DeepSeek 官方开源的 Agent 运行时，采用"一切皆插件"（everything is a plugin）架构，基于 Cordis 插件容器构建。

事实	内容	
开源时间	2026-08-13	
协议	MIT（可商用、可改）	
语言	TypeScript（Node.js ≥ 22）	
版本线	0.1.0-rc.x（当前 rc.6，迭代快，官方明示"将有破坏性变更"）	
底层	Cordishttps://github.com/cordiverse/cordis（可组合插件容器）	
内置形态	web（Web UI）+ headless（一次性 CLI）	
官方定位	官方 README 原话："everything is a plugin"	

1.3 架构是怎么"一切皆插件"的

```
flowchart TB     subgraph Profile["你的 profile（可启动形态）"]         P1["dsh web（Web UI）"]         P2["dsh headless（CLI）"]         P3["自定义 profile（TUI/桌面/机器人…）"]     end     subgraph Plugins["能力层（每个能力 = 一个插件）"]         L["llm：模型接入 + 推理档位"]         T["tools：写文件/终端/搜索/技能"]         S["session：会话持久化"]         C["client：界面（web/终端）"]         ST["settings：用户配置"]     end     subgraph Cordis["Cordis 插件容器"]         D["依赖注入 · 事件 · 生命周期"]     end     Profile --> Cordis     Cordis --> Plugins     Plugins --> L     Plugins --> T     Plugins --> S     Plugins --> C     Plugins --> ST
```

<!-- [style] 子标题统一不带编号（原 3.1/3.2 为旧章节号残留，对齐其余章节） -->

分层视图

```
┌──────────────────────────────────────────────┐  │  你的 profile（可启动形态）
│  = bundle 栈 + 你的补丁层                    │  │  · dsh web（Web UI 形态）
│  · dsh headless（CLI 形态）                  │  │  · 你的自定义 profile（TUI/桌面/机器人…）
│  ├──────────────────────────────────────────┤  │  能力层（每个能力 = 一个插件）
│  · llm：模型接入（DeepSeek V4 系 + 推理档位）  │  │  · tools：工具（写文件/终端/搜索/技能…）
│  · session：会话持久化                       │  │  · client：界面（web 浏览器半 / 终端半）
│  · settings：用户配置                        │  │  …（60+ 官方包）
└──────────────────────────────────────────────┘  │  Cordis 插件容器：加载、依赖注入、事件、生命周期
```

<!-- [style] 子标题编号统一：去掉残留旧编号 -->

三个必须懂的概念

① profile（可启动形态）

一个 profile = $DSHHOME/profiles/<name>/ 目录，包含：

- `package.json`：插件依赖 + 清单（`dsh.profile.bundles` 指定 bundle 顺序）
- `cordis.patch.yml`：你的补丁层（挂载/覆盖插件）

启动时按顺序合成：内置 bundle → profile patch → 全局 patch → --patch 覆盖。

② host 半 / client 半（一个插件，两副面孔）

半边	跑在哪	干什么	
host 半	Node 进程	工具、服务、事件、文件系统——applyctx 注册	
client 半	浏览器（web profile）	UI、交互——package.json 的 dsh.client 声明	

一个 npm 包可同时携带两半（exports["."] + exports["./client"]）。

③ 扩展点（extension point）

官方原则："Plugins, not loop changes"——改行为优先用官方钩子，不要 fork 核心。常用扩展点（后续章节逐一实战）：

- `agent/request` waterfall：每次模型请求前改配置（工具调用提速插件的挂点）
- `conversationEvents.register`：订阅/注入对话事件
- `ctx.slots.inject`：在界面槽位注入 UI
- `settings` 服务：注册用户可配置项

1.4 DSH 与主流 Agent 的全面对比

<!-- [style] 子标题编号统一：去掉残留旧编号 -->

能力矩阵（核心六家）

维度	dsh	Claude Code	OpenAI Codex	OpenCode	Gemini CLI	Kimi CLI	
开源	✅ MIT	❌ 闭源	❌ 闭源	✅ MIT	❌ 闭源	❌ 闭源	
模型绑定	模型无关（官方适配 DeepSeek）	Claude 系	GPT 系	任意	Gemini 系	Kimi 系	

官方运行时	✅（web + headless + 插件生态）	产品即运行时	产品即运行时	客户端（无官方后端）	产品即运行时	产品即运行时	
插件体系	官方级：一切皆插件，60+ 官方包	配置/钩子为主	配置为主	配置为主	无	无	
自定义界面	✅（client 半 = 自由 UI）	❌	❌	部分（TUI 固定）	❌	❌	
自动化/CI	✅ headless profile	✅	✅	✅	✅	✅	
TUI	插件可做（官方未内置）	✅ 内置	✅ 内置	✅ 内置	✅	✅	
生态阶段	零日起步（2026-08-13）	成熟	成熟	成熟	成熟	早期	
适合谁	想深度定制 + 玩生态的开发者	开箱即用	开箱即用	熟悉 OpenCode 用户	Google 生态	Kimi 生态	

<!-- [style] 子标题编号统一：去掉残留旧编号 -->

更多主流 Agent 速览（一句话定位）

Agent	一句话定位	与 dsh 的核心差异	
Cursor	IDE 内嵌的 AI 编码助手（Composer/Agent 模式）	深度绑定 IDE；dsh 是独立运行时，可配任意编辑器/终端	
Amp	终端原生、代理优先的 AI 编码 agent	轻量终端形态；dsh 多了官方后端 + 插件生态	
Devin	云端"AI 软件工程师"（独立任务/浏览器/工作区）	托管云端；dsh 本地运行、数据不出本机	
Windsurf	IDE 内嵌编码 agent（Flow/Agent 模式）	同 Cursor，绑定 IDE	
Aider	开源、Git 优先的终端 pair-programming	专注"改代码 + git"；dsh 是通用运行时	

	agent		
Qwen Code	阿里通义 coding agent CLI（内置 DeepSeek provider）	主打 Qwen 模型；dsh 官方适配 DeepSeek 且插件化	
GLM CLI	智谱 coding agent CLI	主打 GLM 模型；dsh 模型无关 + 插件生态	
Grok CLI	xAI 终端 coding agent	主打 Grok 模型	

*结论：主流 agent 大致分三类——**IDE 内嵌**（Cursor/Windsurf）、**终端编码助手**（Codex/OpenCode/Aider/Qwen Code/GLM CLI/Grok CLI）、**运行时/平台**（dsh/Devin）。dsh 是目前唯一"官方开源 + 模型无关 + 插件生态"的运行时型选手。*

<!-- [style] 子标题编号统一：去掉残留旧编号 -->

通俗文字版：每家是什么、适合谁、和 dsh 差在哪

Claude Code（Anthropic）：目前最成熟的终端编码助手。开箱即用、TUI 体验好、生态成熟——适合想马上干活的人。缺点：闭源、绑定 Claude 模型、定制空间有限（只能配置/钩子）。和 dsh 比：dsh 能改的东西它改不了（界面/工具链/后端），但 dsh 的"开箱即用"还比不上它。

OpenAI Codex：OpenAI 的终端 agent。工程能力强、GPT 系模型加持。同样闭源绑定。和 dsh 比：能力线接近，但 dsh 开源可自改。

OpenCode：开源、终端、可配任意模型——和 dsh 最像的"邻居"。关键差异：OpenCode 没有官方后端运行时（它是客户端 + 配置），dsh 有官方 bundle + 60+ 包 + 插件生态，可定制面更深。如果你是 OpenCode 用户，迁移 dsh 的成本很低（概念类似）。

Gemini CLI：Google 的终端 agent。长上下文/多模态是强项（Gemini 模型优势）。绑定 Google 生态。

Kimi CLI：月之暗面的终端 agent。中文场景好、Kimi 模型加持。生态早期。

Cursor / Windsurf：IDE 内嵌型——它们的优势是"编辑器里就用"，劣势是你被锁在 IDE 里。dsh 是独立运行时，可以在任意环境（终端/CI/服务器/未来的 TUI）用同一套 agent 能力。

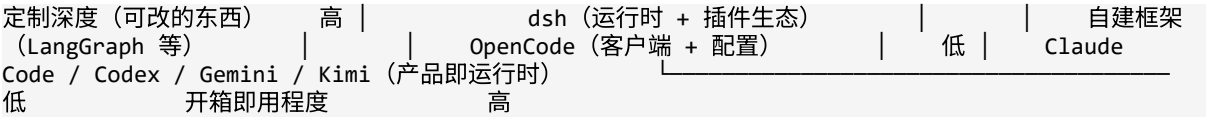
Devin：云端工程师——你给它任务，它在云上干活。优势是托管、有浏览器；劣势是数据出本机、费用高。dsh 本地运行，隐私可控。

Aider：老牌开源、Git 优先、轻量。适合"只想让 AI 帮改代码"的极简主义者。dsh 是更重的运行时——如果你只需要 Aider 做的事，Aider 够用；如果你想做 Agent 系统，dsh 是底座。

Qwen Code / GLM CLI / Grok CLI：各家模型的终端 agent——模型绑定是它们的天然属性。dsh 模型无关（官方适配 DeepSeek，可接 OpenAI 兼容），更适合想"模型可换"的人。

<!-- [style] 子标题编号统一：去掉残留旧编号 -->

一张图理解 dsh 的差异化位置



- **右上**：dsh——可定制面最大，但"开箱即用"需生态补足
- **左上**：自建框架——完全自由但全要自己搭
- **右下**：产品型 agent——最好用但最封闭

<!-- [style] 子标题编号统一：去掉残留旧编号 -->

案例对比：同一个任务，不同的打开方式

任务：让 Agent 在仓库里"找到所有调用某个函数的地方，并统一改一个参数"。

Agent	你会怎么做	体验	
dsh（web）	dsh web → 输入指令 → 模型用 Grep/Read/Edit 工具完成	Web UI + 右侧插件侧边栏（可加 Git 面板）	
dsh（headless）	dsh --profile headless "任务" → 打印结果退出	可进 CI：非零退出码即失败	
Claude Code / Codex / OpenCode / Kimi	打开 TUI → 输入指令 → 模型完成	终端 TUI，开箱即用	
Cursor / Windsurf	在 IDE 里选中代码 → 输入指令	IDE 内嵌体验	
Devin	网页里建任务 → 云上完成	托管、有浏览器、数据出本机	

差异点在哪：同样一句话，dsh 让你多了一个选择维度——界面和工具链都可以换。其他 Agent 的界面/工具链是官方定的，dsh 是你可以拼的。

<!-- [style] 子标题编号统一：去掉残留旧编号 -->

案例对比：真实工作流（我们的实测）

以下是我们真实开发 dsh 生态时的对比观察（2026-08-13）：

场景	dsh 实测	备注	
简单文件创建	冷启动 ~110s（首轮含上下文注入）→ 热缓存 ~1s	思考档位是主要变量	
50 步工具链任务	LLM 耗时 10m+，工具调用 9m+	每步思考累计——提速插件价值在此	
插件开发	从零到可运行插件：1 天（含测试+实机验证）	扩展点清晰（agent/request waterfall）	

与 Claude Code 同任务	dsh 配 V4-Flash 成本约为 Claude 的 1/10~1/30	价格维度 dsh 生态显著占优	

同模型 × 不同 Agent 的严格对比见 [Benchmark 附录](./benchmark.md)（omp 36s / dsh 85s / opencode 114s）。

1.5 什么时候用 dsh（选型决策）

✅ 推荐入场：

- 你要做**模型无关、界面可选、行为可改**的 Agent 底座
- 你想成为 **dsh 生态早期贡献者**（先发优势，官方点名鼓励）
- 你要在**服务器/CI** 跑 Agent（headless profile）
- 你对**成本敏感**（DeepSeek 模型 + 开源生态）

⏸️ 暂时观望：

- 只要"开箱即用的编码助手"——Claude Code 等更成熟
- 不能接受 rc 阶段的破坏性变更——等 `0.1.0` 正式版
- 重度依赖某模型独有能力（如 Claude artifacts）——模型绑定场景

1.6 常见问题（FAQ）

Q1：dsh 是模型吗?

不是。dsh 是运行时/框架，模型通过 llm 插件接入（官方适配 DeepSeek V4 系，理论上可接其他 OpenAI 兼容模型）。

Q2：dsh 和 OpenCode 什么关系?

都是开源 Agent 客户端，但定位不同：OpenCode 是"客户端 + 配置"，dsh 是"运行时 + 官方插件生态"（有官方后端 bundle 与 60+ 官方包）。dsh 更底层、可定制面更大。

Q3：没写过 TypeScript 能玩吗?

能。使用（第 2 章）不需要编程；写插件（第 4 章）需要基础 TS，但教程给完整代码。

Q4：dsh 稳定吗?

当前 rc 阶段（0.1.0-rc.6），迭代快、有破坏性变更。生产核心依赖建议等正式版；玩生态现在正是时机。

Q5：为什么现在学 dsh 值得?

生态零日 + 官方点名鼓励社区 + 中文教程空白——每个早期生态都有"第一个吃螃蟹的人"的红利，现在是入场窗口。

下一章：[第 2 章：五分钟快速上手](./02-quickstart.md) —— 装起来，跑起来。

动手练习（检验你是否真懂了）

1. **一句话测试**：向不懂技术的人解释"dsh 是什么"（不能用"Agent/harness/插件"这些词）
2. **对比测试**：说出 dsh 和 Claude Code 的 3 个本质区别（不是功能列表）
3. **选型测试**：给下面场景选工具并说明理由：

- 场景 A：想要开箱即用的终端编码助手
- 场景 B：想做一个"模型无关、界面自定义"的公司内部 Agent 平台
- 场景 C：想在 CI 里每天自动跑一个数据分析任务

4. **架构测试**：画出"profile → 插件 → 扩展点"的关系图（不看原文）
5. **FAQ 测试**：回答"dsh 是模型吗？""没写过 TS 能玩吗？"

完成练习后，进 [第 2 章](./02-quickstart.md) 动手装起来。

第 2 章：五分钟快速上手

*本章目标：**跟着做，跑起来**。每一条命令都给出预期输出与常见错误解法。建议打开终端边看边做。*

TL;DR（本章核心，30 秒版）

1. **装**：`npx -y @deepseek-ai/dsh web` → http://127.0.0.1:3080
2. **两种常用模式**：web（对话 UI）/ headless（`dsh --profile headless "任务"`，CI 友好）——官方共四种运行模式（见 2.4.1 总览）
3. **推理档位**：off / low（关闭或弱思考/最快，官方 provider 用 `off`，网关用 `low`）/ high（默认）/ max（最强）——**工具链任务 90% 时间在思考，降档是最快提速**
4. **模型**：`deepseek-v4-flash`（默认，性价比）或 `deepseek-v4-pro`（旗舰）
5. **配置**：`~/.dsh/settings.yaml`（模型 + 推理档位）

<!-- [fix] 结构核验：补齐本章导航（第 2 章原缺失，与第 3-11 章结构保持一致） -->

<details><summary>本章导航</summary>

</details>

2.1 准备工作（30 秒检查）

需要	检查命令	通过标准	
Node.js ≥ 22.19	node --version	v22.19.0 或更高（红线，见下）	
npm（随 Node 附带）	npm --version	有版本号即可	

网络	能访问 npm registry	能装包	
（可选）DeepSeek API Key	https://platform.deepseek.com	用于真实对话	

没有 API Key 也能启动 dsh（界面能开），但对话需要 Key。本白皮书示例假设已配置。

Node 版本红线（≥22.19）

社区实测发现 Node < 22.19 会触发两个致命缺失：

缺失 API	报错示例	触发帖号	
node:zlib 无 createZstdDecompress	TypeError: zlib.createZstdDecompress is not a function	#100https://github.com/deepseek-ai/deepseek-harness/discussions/100	
AbortSignal.timeout 未实现	AbortSignal.timeout is not a function	#311https://github.com/deepseek-ai/deepseek-harness/discussions/311	

Workaround：用 nvm/volta/fnm 切换到 22.19.0+。Node 24 早期版本（如 v24.15）也可能触发 install-lefthook failed——若遇此错，pin 到 22.19.0 最稳([#748](https://github.com/deepseek-ai/deepseek-harness/discussions/748))。

2.2 安装（三种方式）

方式一：直接运行（推荐新手）

```
npx -y @deepseek-ai/dsh --version
```

首次运行会下载 dsh（包体较大，含 40+ 插件模块，约 1-3 分钟）。看到版本号即成功：

<!-- [style] 输出/目录类代码块统一补 text 语言标签 -->

```
0.1.0-rc.6
```

方式二：全局安装（推荐频繁使用）

```
npm install -g @deepseek-ai/dsh dsh --version
```

方式三：免装 Node 的安装包（新手可选）

不想装 Node？社区作者 codeAnqiang-ma（[#380](https://github.com/deepseek-ai/deepseek-harness/discussions/380) 插件踩坑帖作者，已授权收录）提供了免装 Node 的安装包——mac DMG / Windows exe，底层官方 dsh 原样打包、自带 Node 运行时，零前置依赖：

[dsh-installers](https://github.com/codeAnqiang-ma/dsh-installers)（非官方，随官方 rc 版本发布对应安装包，含 SHA256 校验）

适合「不想折腾 Node 环境、双击即用」的尝鲜用户；需要插件开发/频繁升级的话仍建议方式一或二。

安装坑（社区真实踩过）

坑	现象	解决	帖号	
首次 npx 极慢（Windows）	npx dsh web 首次在 Windows 上 8+ 分钟零反馈，npm 需下载 500+ 包	耐心等待；改用 npm i -g @deepseek-ai/dsh 后秒启	#176https://github.com/deepseek-ai/deepseek-harness/discussions/176	
pnpm dlx 404	pnpm dlx @deepseek-ai/dsh web 报 @deepseek-ai/dsh-pty@0.0.1-rc.2 未发布	用 npx 或 npm i -g 替代 pnpm dlx	#369https://github.com/deepseek-ai/deepseek-harness/discussions/369	
pnpm 全局装后找不到插件	pnpm add -g @deepseek-ai/dsh 后启动报 Cannot find package 'cordis-plugin-timer'	pnpm 全局安装的依赖解析策略与 npm 不同；建议用 npm i -g 或 npx	#55https://github.com/deepseek-ai/deepseek-harness/discussions/55	
WSL2 上 npx 装不上	在 WSL2（Ubuntu 等发行版）里 npx -y @deepseek-ai/dsh 安装失败	先确认 npm 代理配置；仍不行就改用源码安装（社区实测可行）	#118https://github.com/deepseek-ai/deepseek-harness/discussions/118	

***WSL2 安装注意**：社区在 [#118](https://github.com/deepseek-ai/deepseek-harness/discussions/118) 反映 WSL2 上 `npx` 安装可能失败（Windows 原生侧正常，见该帖"win 上都装好开始玩了"的对照）。排查顺序：① 确认 npm 代理/registry 配置；② 若 `npx` 仍装不上，**先试源码安装**（clone 官方仓库后 `pnpm install`，社区实测可行）。详见该帖评论区。*

*全局安装方式对比：**`npm i -g`** 最稳（社区验证最多）；**`npx`** 适合尝鲜；**`pnpm dlx`** 暂不建议（rc 阶段 pty 包未发布到 pnpm 可见 registry）。*

2.3 模式一：Web UI（`dsh web`）

启动

```
dsh web
```

预期输出：

```
dsh web: http://127.0.0.1:3080
```

浏览器打开 http://127.0.0.1:3080。

界面认识（对照截图）

![dsh Web UI 对话](./assets/demo-web-chat.png)

区域	内容	
左栏	会话列表 / 工作区切换 / 新建会话	
中栏	对话区：输入框、模型选择（DeepSeek V4 Flash）、推理等级（High）	
右侧/底部	插件侧边栏（默认空；安装社区插件后出现）	
右上	Session log（会话日志）/ 轨迹（工具调用轨迹）	

第一次对话

1. 点「新建会话」
2. 输入框输入：`你好，请用一句话介绍你自己`
3. 回车发送

预期回复类似：

> *你好！我是 DeepSeek 驱动的 AI 编程助手，可以帮你写代码、调试问题、处理文件、搜索资料，以及完成各种开发和办公任务。*

模型与推理档位

点输入框旁的「选择模型」，打开模型与推理档位选择器：

![dsh 模型选择与推理档位](./assets/demo-model-selector.png)

模型	定位	
deepseek-v4-flash（默认）	性价比：快、便宜，日常够用	
deepseek-v4-pro	旗舰：更强，更贵更慢	

推理等级（思考模式三档，2026-08-13 起支持）：

档位	速度	质量	建议场景	
low	最快	够用	简单/确定性任务、批量、工具链廉价轮次	
high（默认）	中等	好	日常 Agent 任务	
max	最慢	最强	复杂推理、长链规划	

<!-- [fix] 技术准确性核验：上表档位为本白皮书实测环境（pi-ai / opencode-go 网关）所支持。DeepSeek 官方适配器（默认 provider=deepseek-official，llm-deepseek 插件）只接受 off（关闭思考，最快）/ high / max 三档，low 会抛 UNSUPPORTEDREASONINGEFFORT。在 settings.yaml 里对默认 provider 请使用 off / high / max -->

> 💡 ***性能关键认知**：模型在**每次工具调用前都会重新思考**。实测一个"创建文件"任务，思考占 ~90% 墙钟时间；50 步工具链任务思考累计可达数分钟到十几分钟。**调低推理档位是性价比最高的提速手段**（见第 6 章 + 示例提速插件）。*

2.4 模式二：Headless（一次性任务，适合脚本/CI）

```
dsh --profile headless "你好，请用一句话介绍你自己"
```

预期输出（打印结果后进程退出）：

```
你好！我是 DeepSeek 驱动的 AI 编程助手，可以帮你写代码、调试问题、处理文件、搜索资料，以及完成各种开发和办公任务。
```

Headless 的核心价值：

- **自动化**：可进 CI、服务器、cron
- **脚本友好**：非零退出码 = 失败；输出可管道处理
- **会话隔离**：每次调用一个新鲜会话（`--resume` 可恢复，见 `dsh --profile headless --help`）

实战：写个脚本每天跑一次"生成日报"：

```
dsh --profile headless "读取工作区今天的 git log，生成一份中文日报摘要" > daily-report.md echo "exit=$?"
```

2.4.1 官方四种运行模式总览

来源：官方站点 https://deepseek.com/harness/ 与官方仓库 `docs/`（2026-08 快照）。前两节（web / headless）是白皮书实测详讲的；后两种模式官方已公布但白皮书暂未逐项实测。

模式	一句话	用途	白皮书覆盖	
Standard mode（标准）	完整工具集 + 浏览器界面	日常对话/开发（dsh web）	✅ 2.3 节实测	

Minimal mode（极简）	仅 bash + strreplaceeditor 两个工具	基准测试、最小攻击面	⚠ 未实测，见官方 docs	
Code mode（代码）	模型写 TypeScript，在单次程序内编排多轮工具调用	长链路自动化、确定性执行	⚠ 未实测，见官方 docs	
Creator mode（创造）	运行时自检 + 内存内测试 Cordis 插件 + 组合新模式	插件开发、快速原型	⚠ 未实测，见官方 docs	

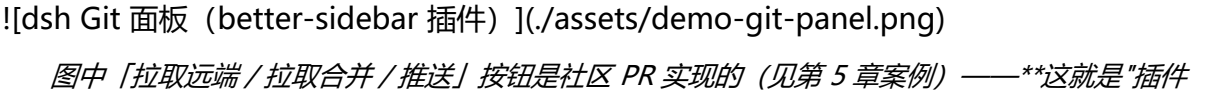
*对新手：日常用 **Standard（web）**、脚本用 **Headless** 即可；Code/Creator 模式属于进阶能力，等官方文档稳定后再深挖。官方文档站（VitePress）：https://deepseek-harness.github.io/deepseek-harness/*

2.5 你的第一个插件：给 web 加个 Git 面板

dsh 的侧边栏默认是空的——安装社区插件 dsh-better-sidebar 体验"一切皆插件"（详细原理见第 3 章，这里先跑通）：

```
# 1. 找到你的 web profile #    Windows: %USERPROFILE%\.dsh\profiles\web #    macOS/Linux: ~/.dsh/profiles/web  # 2. 在 package.json 的 dependencies 加一行（link: 指向插件源码） # "dsh-better-sidebar": "link:C:\\path\\to\\DSH-better-sidebar"  # 3. 在 cordis.patch.yml 加挂载行 #    - insert: #        - id: better-sidebar #          name: dsh-better-sidebar  # 4. 安装并重启 cd ~/.dsh/profiles/web && pnpm install #     （Windows cmd 下 `~` 不展开，请用: cd %USERPROFILE%\.dsh\profiles\web） dsh web
```

重启后，右侧出现文件管理 / 终端 / Git 面板 / 浏览器等标签：



*图中「拉取远端 / 拉取合并 / 推送」按钮是社区 PR 实现的（见第 5 章案例）——**这就是"插件生态"的运转方式**。*

2.6 配置与目录速查

首次运行后生成的目录：

```
~/.dsh/ ├── settings.yaml          # 全局设置（模型、推理档位） ├── profiles/              # profile 目录 │   └── web/ │       ├── package.json      # 插件依赖 + 清单 │       └── cordis.patch.yml  # 补丁层（挂载插件） ├── sessions/              # 会话数据 └── storages/ # 持久化存储
```

settings.yaml 示例：

```
agent-default-model:   model: deepseek-v4-flash   reasoningEffort: high
```

2.7 命令速查

命令	用途	
dsh web	启动 Web UI（=dsh --profile	

	web）	
dsh --profile headless "任务"	一次性任务，打印结果退出	
dsh plugin --profile <name> add <pkg>	给 profile 安装插件	
dsh --dump-config	打印合成配置树	
dsh --profile tui	TUI 模式（需先安装 tui 插件，官方未内置）	
dsh --version	版本	

2.8 排障速查

现象	原因与解法	
dsh: profile "tui" does not exist	tui profile 需插件创建（dsh plugin --profile tui add <pkg>）	
npx 极慢	首次下载包体大；npm i -g 后更快	
浏览器打不开 3080	端口被占：netstat -ano |	findstr 3080 → kill PID
模型无响应	检查 ~/.dsh/settings.yaml 模型配置 + API Key	
插件装不上（404）	rc.1 依赖断裂：确认依赖用 ^0.1.0-rc.6 线（第 3 章常见坑 #1）	
升级后行为变了	rc 阶段破坏性变更正常，看官方 changelog	

下一章：[第 3 章：profile 与插件系统](./03-profiles.md) —— 理解可定制骨架。

> ***源码方式启动慢的根因（#1424 社区实测确认）**：`pnpm dsh web`每次启动用 tsx/esbuild **现场转译整个 TS 源码图**（非全量构建），叠加机械盘/大文件数时冷启动可达数分钟。解决：先 `pnpm build`全量构建一次，或直接用 `npx @deepseek-ai/dsh web`（发布版）绕开转译开销。*

动手练习（10 分钟内完成）

1. **安装**：`npx -y @deepseek-ai/dsh --version` 确认版本
2. **Web 对话**：启动 `dsh web`，新会话发"你好"，观察回复与界面布局
3. **Headless**：`dsh --profile headless "1+1 等于几"`，确认打印结果后退出

4. **推理档位实验**：把 settings.yaml 的 `reasoningEffort` 改为 `off`（官方 provider）或 `low`（第三方网关），重新跑一个简单任务，感受速度差异
5. **排障演练**：模拟"端口被占"（先起一个占用 3080 的服务），用 `netstat` 排查

全部通过后，进 [第 3 章](./03-profiles.md) 理解"为什么能这样改"。

第 3 章：profile 与插件系统

*本章目标：理解 dsh 的可定制骨架——profile 怎么组织、插件怎么挂载、host/client 双半是什么。 **这是从 "用户" 变成 "开发者" 的分水岭。 **

TL;DR（本章核心，30 秒版）

1. **profile = 一种可启动形态**：`~/.dsh/profiles/<name>/` 目录，由 `package.json`（插件清单）+ `cordis.patch.yml`（补丁层）组成
2. **挂载插件只需两步**：`package.json` 加依赖 + `cordis.patch.yml` 加挂载行，然后 `pnpm install` 重启
3. **host 半跑在 Node，client 半跑在浏览器**：一个 npm 包通过 `exports["."]` 和 `exports["./client"]` 同时携带两副面孔
4. **改行为找扩展点，别 fork 核心**：`agent/request` waterfall、`conversationEvents`、`ctx.slots.inject`、`settings` 服务是四大常用钩子
5. **rc 阶段三大坑**：依赖版本用 `^0.1.0-rc.6` 线、`next()` 必须 await、client 测试需要 dsh 运行时

<details><summary>本章导航</summary>

</details>

3.1 profile：一个可启动的配置栈

dsh 用 profile 表示"一种可启动的形态"。官方内置两个，其余用插件创建：

profile	用途	命令	
web	Web UI（对话 + 侧边栏 + 工具）	dsh web	
headless	一次性 CLI 任务	dsh --profile headless "任务"	

tui（需插件）	终端 UI	dsh --profile tui（未内置，需安装插件）	

一个 profile 目录长这样（~/.dsh/profiles/<name>/）：

<!-- [style] 目录树代码块统一补 text 语言标签 -->

```
profiles/web/ ├── package.json        # 插件依赖 + dsh.profile 清单（bundles 顺序） ├── cordis.patch.yml    # 你的补丁层：挂载/覆盖插件的声明 ├── cordis.yml          # （生成的）合成配置 ├── pnpm-workspace.yaml └── node_modules/
```

加载顺序（官方文档原文）：内置 bundle（dsh-base → dsh-web-app）→ profile 的 cordis.patch.yml → 用户级 ~/.dsh/cordis.patch.yml → --patch 覆盖层。

内置 Agent 预设：标准 / PTC / 极简 / 创造

profile 解决"以什么形态启动"，Agent 预设解决"以什么方式跑 Agent"。dsh 内置了四种 Agent 预设（源码在官方仓库 apps/cli/config/agent-presets/ 的 agent.cordis.yml），每种预设默认加载不同的插件集合：

预设	定位	加载内容	典型场景	
标准	默认预设	完整工具组合	日常 Agent 工作	
PTC（Code mode）	代码驱动工具链	程序化工具调用——模型生成一段代码组合多轮工具调用	复杂多步骤工具工作流	
极简	基准测试	仅 bash + 一个文件编辑工具	最小环境下的模型基准测试	
创造	插件开发用	检查当前运行时、在内存中试验 Cordis 插件、组合新预设	构建与测试新的插件组合	

PTC 命名澄清（[#1052](https://github.com/deepseek-ai/deepseek-harness/discussions/1052) 评论区社区详解）：PTC = Programmatic Tool Calling（程序化工具调用）。官方中文站只用了「PTC 模式」这个营销名（官方站点原文："PTC 模式通过模型生成的一段代码组合多轮工具调用"），官方英文页面对应 Code mode；「Programmatic Tool Calling」的完整扩写出自社区详解（[cnblogs 指南](https://www.cnblogs.com/sing1ee/p/22455466)），与源码实现完全吻合。机制与收益详见第 8 章 8.3.1；四种运行模式总览见第 2 章 2.4.1。

3.2 挂载一个插件：两处改动

以挂载社区插件 dsh-better-sidebar 为例（真实操作）：

① package.json 加依赖（link: 指向本地源码，或用 npm 包名）：

```
{   "dependencies": {     "dsh-better-sidebar": "link:C:\\path\\to\\DSH-better-sidebar"   }
}
```

② cordis.patch.yml 加挂载行：

```
- insert:     - id: better-sidebar       name: dsh-better-sidebar
```

③ 安装并重启：

```
cd ~/.dsh/profiles/web && pnpm install dsh web   # 重启后生效
```

> 💡 *插件默认是**按会话隔离**的（`better-sidebar`的布局/标签按会话持久化）——挂载是 profile 级，但状态是会话级。*

3.3 host 半与 client 半：一个包，两副面孔

插件可以同时携带两个运行半：

半边	运行位置	职责	示例	
host 半	Node 进程	工具、服务、事件、文件系统、进程	applyctx 注册工具/服务	
client 半	浏览器（web profile）	UI、交互、DOM	package.json 的 dsh.client 声明 + src/client/	

package.json 里如何声明 client 半：

```
{   "dsh": {     "client": {       "inject": ["@deepseek-ai/dsh-client-runtime",
"@deepseek-ai/dsh-client-locale"],       "platform": "web"     }   },   "exports":
{     ".": { "types": "./lib/types/index.d.ts", "default": "./lib/index.js" },
"./client": { "types": "./lib/types/client/index.d.ts", "default":
"./lib/client.js" }   } }
```

- host 半：`exports["."]` → `apply(ctx)`（cordis 插件主体）
- client 半：`exports["./client"]` → 浏览器侧的 `apply(ctx)`
- `inject`：声明需要的服务（cordis 依赖注入）

3.4 扩展点：改行为优先找钩子，别 fork 核心

官方原则（CONTRIBUTING/AGENTS.md 明示）："Plugins, not loop changes: new behavior goes on documented extension points"。新手最常见的错误是改核心 loop——正确做法是用扩展点。

已确认的常用扩展点（后续章节逐一实战）：

扩展点	位置	用途	
agent/request waterfall	agent-loop	每次模型请求前改配置（provider/model/reasoningEffort/tools）——提速插件示例用这个	

agent/request-error	agent-loop	请求失败时干预（官方 compaction 插件用这个做上下文溢出恢复）	
conversationEvents.register	client runtime	订阅/注入对话事件（tool/call、turn/start 等）	
ctx.slots.inject	client ui-slots	在界面槽位注入 UI（如 turnTail 显示产物文件行）	
settings 服务	dsh-settings	注册用户可配置的命名空间（设置页自动渲染）	
ctx.provide / ctx.get	cordis	跨插件提供服务	

3.5 常见坑（真实踩过）

依赖解析坑

坑	现象	解决	帖号	
rc.1 依赖断裂	pnpm install 报 @deepseek-ai/dsh-type-meta@0.0.1-rc.1 404	官方 rc.1 时代多个包从未发布；升级到 ^0.1.0-rc.6 线	—	
全局 core.hooksPath 冲突	全局设过 core.hooksPath（如 Codex/Claude Code）→ pnpm install 时 lefthook postinstall 失败	临时取消全局 hooksPath：git config --global --unset core.hooksPath，装完再恢复	#139https://github.com/deepseek-ai/deepseek-harness/discussions/139	
macOS 全局安装解析不到插件	pnpm add -g @deepseek-ai/dsh 后启动报 ~88 个插件包 ERRMODULENOTFOUND	pnpm 全局安装的模块解析策略与 npm 不同；macOS 建议用 npm i -g 或 npx	#204https://github.com/deepseek-ai/deepseek-harness/discussions/204	
koffi 预编译损坏（Windows）	koffi 3.1.3/3.1.4 的 win32-x64 预编译 binary 损坏 → 目录选择器崩溃/服务静默挂	锁死 koffi@3.1.2：pnpm add koffi@3.1.2 或改 package.json resolution	#293https://github.com/deepseek-ai/deepseek-harness/	

			discussions/293	
koffi 原生崩溃（Windows）	选择器 worker 崩溃（0xC0000005 访问冲突）或启动后直接挂掉	同上一行锁 3.1.2；若仍崩，检查是否 STA 线程 CoUninitialize 段错误（社区有补丁）	#197https://github.com/deepseek-ai/deepseek-harness/discussions/197	

插件安装格式坑

格式	写法	注意	帖号	
npm 包名	dsh plugin add dsh-better-sidebar	需已发布到 npm；rc 阶段很多社区插件未发布	—	
GitHub 仓库	dsh plugin add github:作者/仓库	只加依赖，不自动 append 到 cordis.patch.yml，需手动补挂载行	#656https://github.com/deepseek-ai/deepseek-harness/discussions/656	
本地路径	link:/path/to/plugin（macOS/Linux） link:C:\path\to\plugin（Windows）	开发调试用；路径斜杠按平台区分	—	
含空格路径（Windows）	dsh plugin <...> 传含空格路径	Windows 上含空格路径在内部转发时被拆断：apps/cli/src/plugin.ts 的 runPlugin 用 spawnSync"pnpm", args, { shell: true }——Node 按空格拼接参数、不转义（Node 22+ 起 DEP0190 警告），调用方引号到不了转发层	修复方向：shell: false（直接传参数组）或转发前自行转义	#1420https://github.com/deepseek-ai/deepseek-harness/discussions/1420

社区踩坑总结：用 `github:` 格式装插件后，记得手动去 `cordis.patch.yml` 补 `- insert:` 挂载行，否则插件依赖装了但 dsh 不加载。

其他常见坑

坑	现象	解决	
插件缺 main	dsh: No "exports" main defined	host 插件 package.json 要暴露 . 入口（"main": "src/index.ts" 可被 tsx 直接加载）	
事件 handler 忘了 await next	请求丢失 provider/model 报错	agent/request 的 next 返回 Promise，必须 await 后 spread	
类型报 'agent/request' is not assignable to keyof Events	npm 类型未 re-export 官方类型增强	用宽松签名（ctx.on as unknown as ...）在边界转换	
client 包产物依赖 window.ModuleLoader	jsdom 无法直接跑 client 测试	组件级测试需 dsh web 运行时（官方 CI 跑）	

动手练习（检验你是否真懂了）

1. **理解题**：不看原文，画出 `profiles/web/` 目录下的文件结构，并说出每个文件的作用

> 自查：参考本章 3.1 节"profile 目录长这样"

2. **理解题**：解释"host 半"和"client 半"分别跑在哪里、各负责什么。一个插件可以只有 host 半吗?

> 自查：参考本章 3.3 节表格

3. **动手题**：打开你的 `~/.dsh/profiles/web/package.json`，找到 `dsh.profile.bundles` 字段，说出 bundle 的加载顺序

> 自查：参考本章 3.1 节"加载顺序"段落

4. **动手题**：假设你要挂载一个名为 `dsh-my-widget` 的插件，写出 `package.json` 和 `cordis.patch.yml` 各需要加什么内容

> 自查：参考本章 3.2 节的两处改动示例

5. **动手题**：在 `cordis.patch.yml` 里加一行错误的挂载（比如拼错插件名），重启 dsh web，观察报错信息并记录

> 自查：对比本章 3.5 节"常见坑"表格

6. **思考题**：官方原则说"Plugins, not loop changes"。如果你想改 dsh 的"模型回复后自动总结"行为，应该用哪个扩展点？为什么不该直接改 agent-loop 源码?

> 自查：参考本章 3.4 节扩展点表格 + "改行为优先找钩子"原则

常见疑问 FAQ

Q1：profile 和 bundle 有什么区别?

profile 是"一种可启动形态"（如 web、headless），bundle 是"一组预配置的插件集合"。profile 通过 dsh.profile.bundles 指定要加载哪些 bundle，再叠加自己的 patch。简单说：bundle 是积木包，profile 是拼好的成品。

Q2：cordis.patch.yml 里的 - insert: 是什么意思？还有其他操作吗?

insert 是"在插件链中插入一个新插件"。patch 文件基于 cordis 的配置合并机制，支持 insert（插入）、override（覆盖已有插件配置）等操作。新手只需掌握 insert 即可挂载大部分插件。

Q3：插件状态是按 profile 隔离还是按会话隔离?

挂载是 profile 级的（装一次，该 profile 下所有会话都能用），但状态是会话级的。比如 better-sidebar 的标签布局按会话持久化，不同会话互不干扰。

Q4：我想写一个只有 client 半的插件（纯 UI），可以吗?

可以。package.json 里声明 dsh.client + exports["./client"]，不写 exports["."] 即可。但注意：client 半无法直接访问文件系统或跑命令，需要通过 host 半的服务（ctx.provide/ctx.get）桥接。

Q5：agent/request waterfall 和 conversationEvents.register 有什么区别？我该用哪个?

agent/request 是"改模型请求配置"的钩子（provider/model/tools/reasoningEffort），每次请求前触发，返回新配置。conversationEvents.register 是"订阅/注入对话事件"的钩子（tool/call、turn/start 等），用于监听或注入事件流。想改请求参数用前者，想监听对话流用后者。

Q6：为什么我的插件装上去没反应？怎么调试?

三步排查：① 确认 pnpm install 无报错且 nodemodules 里有你的插件；② 确认 cordis.patch.yml 的 id/name 拼写正确；③ 重启 dsh web 后看进程日志有没有插件加载信息。如果 host 半有 console.log，日志里应该能看到输出。

下一章：[第 4 章：插件开发实战](./04-plugin-dev.md)（规划中）—— 从零写第一个 host 插件。

第 4 章：插件开发实战

*本章目标：从零写一个**真实可用的 host 插件**——通过 `agent/request` 扩展点自动调节推理档位。这是一个提速插件的完整拆解，所有代码可运行、可测试。*

TL;DR（本章核心，30 秒版）

1. **核心问题**：dsh 每次工具调用前模型都重新思考，50 步任务 90%+ 时间在思考——降档是最快提速
2. **架构三板斧**：纯函数（决策逻辑，零依赖可单测）→ 插件主体（接入 waterfall）→ 实机验证（日志证明注入发生）
3. **`agent/request` waterfall**：每次模型请求前触发，监听者返回值传给下一个监听者，实现"保留原配置 + 覆盖某字段"
4. **`next()` 是 Promise，必须 await**：不 await 直接 spread 会得到空对象，provider/model 丢失报错
5. **开发纪律**：先找扩展点（90% 行为有官方钩子）、逻辑抽纯函数、实机验证不能省

<details><summary>本章导航</summary>

- [4.1 我们要做什么](#41-我们要做什么)
- [4.2 项目骨架](#42-项目骨架)
- [4.3 纯函数：决策逻辑（零依赖，可单测）](#43-纯函数决策逻辑零依赖可单测)
- [4.4 插件主体：接入 `agent/request` waterfall](#44-插件主体接入-agentrequest-waterfall)
- [4.5 测试](#45-测试)
- [4.6 给新手的三条开发纪律](#46-给新手的三条开发纪律)

</details>

4.1 我们要做什么

问题：dsh 在每次工具调用前模型都会重新思考（reasoningeffort）。一个 50 步工具链任务，"思考"占 90%+ 墙钟时间。

方案：一个 host 插件，监听 agent/request waterfall，根据当前步骤最近的工具调用，把简单轮次的 reasoningeffort 从 high 降到 low。

4.2 项目骨架

<!-- [style] 目录树代码块统一补 text 语言标签 -->

```
dsh-speed-plugin/ ├── package.json          # host 插件声明 ├── tsconfig.json ├── src/ │
├── effort-decision.ts  # 纯函数：决策逻辑（零依赖，可单测）  │   └── index.ts            #
apply(ctx)：接入扩展点 └── tests/      └── effort-decision.spec.ts
```

package.json 关键字段：

```
{   "name": "dsh-speed-plugin",   "type": "module",   "main": "src/index.ts",   "exports": {     ".": { "types": "./src/index.ts", "default": "./src/index.ts" }   },   "peerDependencies": {     "@deepseek-ai/cordis": "^4.0.1",     "@deepseek-ai/dsh-agent": "^0.1.0-rc.6"   } }
```

⚠ 依赖版本务必用 `^0.1.0-rc.6` 线——rc.1 线的 npm 依赖链是断的（见第 3 章常见坑）。

4.3 纯函数：决策逻辑（零依赖，可单测）

src/effort-decision.ts：

```
export type EffortId = 'low' | 'high' | 'max'  export interface ToolCallSample {   name: string      // 工具名，如 'write'、'read'、'bash'   argsSize: number  // 参数大小（字符数） }  export interface EffortDecisionInput {   recentCalls: readonly ToolCallSample[]   selected: EffortId      // 用户基线档   allowDowngrade: boolean   allowUpgrade: boolean }  const SIMPLE_TOOL_RE = /^(fs|bash|terminal|read|write|grep|glob|edit|ls|cat|rm|cp|touch|mkdir|pwd)/i  const HEAVY_ARGS = 800  export function decideEffort(input: EffortDecisionInput): EffortId {   const { recentCalls, selected, allowDowngrade, allowUpgrade } = input   if (recentCalls.length === 0) return selected   // 全新提示：保持基线    const ratio = recentCalls.filter(c =>     SIMPLE_TOOL_RE.test(c.name) && c.argsSize < HEAVY_ARGS,   ).length / recentCalls.length   const heaviest = recentCalls.reduce((m, c) => Math.max(m, c.argsSize), 0)    if (ratio >= 0.75 && allowDowngrade) return 'low'   if (heaviest >= HEAVY_ARGS * 4 && allowUpgrade) return 'max'   if (ratio < 0.75) return allowUpgrade ? 'high' : selected   return selected }
```

为什么拆成纯函数：决策逻辑与 dsh 运行时解耦——单元测试零依赖、毫秒级、覆盖所有分支，实机只需要验证"注入是否真的发生"。

4.4 插件主体：接入 `agent/request` waterfall

src/index.ts：

```
import type { Context } from '@deepseek-ai/cordis' import { decideEffort, type ToolCallSample } from './effort-decision.ts'  export interface SpeedPluginConfig {   enabled: boolean   allowDowngrade: boolean   allowUpgrade: boolean   baseline: 'low' | 'high' | 'max' }  export const DEFAULT_CONFIG: SpeedPluginConfig = {   enabled: true, allowDowngrade: true, allowUpgrade: false, baseline: 'high', }  const WINDOW = 8  function recentToolCalls(agent: unknown): ToolCallSample[] {   const events = (agent as { session?: { events?: readonly unknown[] } }).session?.events ?? []   const out: ToolCallSample[] = []  for (let i = events.length - 1; i >= 0 && out.length < WINDOW; i--) {     const e = events[i] as { type?: string; data?: { name?: string; arguments?: unknown } } | undefined  if (e?.type !== 'tool/call') continue     out.push({       name: e.data?.name ?? 'tool',  argsSize: typeof e.data?.arguments === 'string' ? e.data.arguments.length : 0,     })   }  return out.reverse() }  export function apply(ctx: Context, config: SpeedPluginConfig = DEFAULT_CONFIG): void {   if (!config.enabled) return    // 边界适配：npm 包未 re-export 官方事件类型增强，这里放宽签名（第 3 章常见坑）    const on = ctx.on as unknown as (     event: string,     handler: (payload: Record<string, unknown>, next: () => unknown) => unknown | Promise<unknown>,   ) => void    on('agent/request', async (payload, next) => {     const seed = await next() as { reasoningEffort?: unknown }   // ⚠ 必须 await!      const calls = recentToolCalls(payload.agent)     const effort = decideEffort({       recentCalls: calls,  selected: config.baseline,       allowDowngrade: config.allowDowngrade,       allowUpgrade: config.allowUpgrade,     })     console.log(`[speed-plugin] calls=${JSON.stringify(calls)} => reasoningEffort=${effort}`)     return { ...seed, reasoningEffort: effort }   }) }
```

三个关键点（都是真实踩过的坑）：

1. **`next()` 是 Promise**：`await next()` 拿到当前配置；不 await 直接 spread 会得到空对象 → provider/model 丢失 → 报错。

2. **waterfall 语义**：监听者的**返回值**传给下一个监听者/最终请求。返回 `{...seed, reasoningEffort}` 就是"保留原配置 + 覆盖推理档位"。
3. **`agent/request` 每步都触发**：`agent-loop` 的 `buildRequest` 在每一步都会走这个 waterfall——所以动态决策天然按步生效。

4.5 测试

单元测试（纯函数，零依赖）：

```
import { describe, expect, it } from 'vitest' import { decideEffort } from '../src/effort-decision.ts'  it('downgrades to low for simple tool chains', () => {   expect(decideEffort({     recentCalls: [{ name: 'write', argsSize: 40 }],     selected: 'high', allowDowngrade: true, allowUpgrade: true,   })).toBe('low') }) // ... 更多分支：全新提示保持基线 / 禁用降档 / 超大载荷升 max / 混合工具升 high
```

实机验证（关键——证明"注入真的发生"）：

挂载插件（第 3 章方法）→ 重启 dsh web → 发一个创建文件的任务 → 观察 dsh 进程日志：

```
[speed-plugin] agent/request: calls=[]                    => reasoningEffort=high [speed-plugin] agent/request: calls=[{"name":"write",…}] => reasoningEffort=low
```

第一轮无工具调用 → 保持基线 high；检测到 write 工具 → 下一轮降为 low。注入链路完整工作。

完整可运行代码：参考本章各节代码片段组合即可运行。

4.6 给新手的三条开发纪律

1. **先找扩展点**：要改的行为 90% 有官方钩子（`agent/request`、`settings`、`conversationEvents`、`slots`）——不要 fork 核心。
2. **逻辑抽纯函数**：决策/计算逻辑与 dsh 解耦 → 单测毫秒级、覆盖全分支；实机只需验证"注入发生"。
3. **实机验证不能省**：单测证明逻辑，实机日志证明接线——两个都过才算完成。

***官方 cookbook 延伸阅读**（2026-08 官方新增，官方仓库 `docs/cookbook/`）：`adding-a-package.md`（如何加包）、`adding-a-tool.md`（如何加工具）、`adding-a-conversation-node.md`（加对话节点）、`adding-an-llm-adapter.md`（写 LLM 适配器）、`adding-a-vendored-package.md`（vendor 包）、`extension-cookbook.md`（扩展点合集）。本章走的是"最小提速插件"路径，官方 cookbook 覆盖更多扩展点类型，进阶时对照读。*

动手练习（检验你是否真懂了）

1. **理解题**：不看原文，说出这个提速插件的三层架构（纯函数层 / 插件主体层 / 测试层）各自负责什么

> 自查：参考本章 4.2-4.5 节的文件结构

2. **理解题**：解释为什么 `decideEffort` 要设计成纯函数而不是直接在 `apply(ctx)` 里写逻辑。如果决策逻辑依赖了 `ctx.session`，还能单测吗？

> 自查：参考本章 4.3 节"为什么拆成纯函数"段落

3. **动手题**：给 `decideEffort` 写一个新的测试用例：当 `recentCalls` 里有 3 个简单工具 + 1 个超大参数（`argsSize = 5000`）时，应该返回什么档位？写出测试代码并运行

> 自查：参考本章 4.5 节单元测试示例，预期结果取决于 allowUpgrade 配置

4. **动手题**：在 `apply(ctx)` 里，如果把 `await next()` 改成 `const seed = next()`（不 await），会发生什么？写出你的推理，然后在实机中验证

> 自查：参考本章 4.4 节"三个关键点"第 1 条

5. **动手题**：假设你要写一个类似的插件，但改为根据"当前会话的工具调用总次数"来决定档位（超过 20 次自动降为 low），写出纯函数签名和核心逻辑

> 自查：参考本章 4.3 节纯函数设计模式，关键是输入输出类型定义

6. **思考题**：`agent/request` waterfall 可以有多个监听者。如果提速插件和另一个插件都监听了 `agent/request`，谁先执行？返回值怎么传递？

> 自查：参考本章 4.4 节"waterfall 语义"段落 + 第 3 章 3.4 节扩展点说明

常见疑问 FAQ

Q1：next() 返回的 seed 里到底包含什么字段？

seed 是当前 waterfall 链上游累积的请求配置，通常包含 provider、model、reasoningEffort、tools 等字段。你的监听者拿到 seed 后，spread 覆盖想改的字段，其余原样传递。具体字段以官方 agent-loop 包的类型定义为准（rc 阶段可能变动）。

Q2：为什么 npm 包的类型签名要"放宽"？不能直接用官方类型吗？

因为 npm 发布的 @deepseek-ai/dsh-agent 等包没有 re-export 内部的事件类型增强（Events 接口没有被 module augmentation 扩展）。直接用 ctx.on('agent/request', ...) 会报类型错误。解决方案是在边界用 as unknown as 转换，这是 rc 阶段的临时方案，正式版可能修复。

Q3：我的插件需要在每个工具调用后都触发决策，agent/request 够吗？

够。agent/request 在每次模型请求前触发（包括工具调用后的下一轮请求）。你只需要在监听者里读取最近的工具调用历史（从 payload.agent.session.events 里倒序找 tool/call 事件），就能做按步决策。

Q4：纯函数测试通过了，实机验证也通过了，但用户反馈"有时候没效果"，可能是什么原因？

几种可能：① 用户的 settings.yaml 里 reasoningEffort 是 low，降档无效果（已经最低了）；② 用户的任务全是简单工具，本来就快，感知不到差异；③ waterfall 里有其他插件覆盖了你的返回值。建议加日志记录每步的输入/输出档位。

Q5：我想给插件加一个用户可配置的开关（设置页里能开关），怎么做?

用 settings 服务注册一个命名空间（如 speed-plugin），声明配置项（enabled、baseline 等）。dsh 的设置页会自动渲染表单，用户修改后通过 ctx.get 读取。具体 API 参考官方 dsh-settings 包文档。

Q6：recentToolCalls 函数里为什么要 reverse()?

因为从 events 数组末尾往前遍历（取最近的 N 个），结果是倒序的（最新的在前）。reverse 后恢复时间正序（最旧的在前），和实际调用顺序一致，方便决策逻辑按"最近窗口"理解。

下一章：[第 5 章：实战案例](./05-cases.md)（规划中）—— Git 面板、HTML 草稿预览、提速插件。

第 5 章：dsh 能帮你做什么——应用场景全景

*本章目标：抛开"如何开发 dsh"的视角，站在**使用者**角度，看 dsh 能在哪些日常场景真正帮到你。每个场景都有可复制的任务示例与真实耗时参考。*

TL;DR（本章核心，30 秒版）

1. **dsh 是通用 Agent 运行时**：不只是编码助手——数据分析、文档生成、自动化、代码理解都能干
2. **五个核心场景**：日常编码 / 数据处理 / 文档知识 / 自动化运维 / 代码理解与重构
3. **用法一致**：一句话描述目标 + 写清楚验收标准，dsh 会自动编排工具链完成
4. **真实速度**：简单任务秒级、复杂多步任务 1-5 分钟（V4-Flash + high 档实测）
5. **提示词黄金法则**：告诉 dsh"完成的标准是什么"（如"运行验证通过"），比描述过程更有效

<details><summary>本章导航</summary>

</details>

5.1 场景总览

场景	典型任务	dsh 的工具	实测参考	
日常编码	写函数、修 bug、补测试	write / read / bash / grep	5-bug 修复+49 测试：94s	
数据处理	清洗、分析、可视化、报告	read / write / bash / python	数据清洗+可视化：186s	
文档知识	API 文档、教程、总结、翻译	read / write	API 文档生成：1m04s	
自动化运维	批量处理、CI 任务、日志分析	bash / headless	headless 脚本化	

代码理解	重构、审查、技术调研	grep / glob / read / bash	代码重构：4m37s	

所有实测来自白皮书第 10 章真实案例（合成数据），详见 [benchmark](./benchmark.md)。

5.1.1 高缓存命中率：dsh 的成本秘密武器

这是 dsh 生态最被低估的优势。DeepSeek API 的 context cache（提示词缓存）：重复/相似的输入 token 按缓存价计费。实测（真实 dsh 会话统计）缓存命中率可达 97%。

为什么 dsh 缓存命中率特别高：

原因	说明	
会话延续	同会话多轮对话，系统提示/技能目录/历史重复 → 命中	
稳定的工具 schema	每步请求携带相同的工具定义 → 命中	
Agent 工作负载特性	多步工具链反复携带相同上下文 → 命中率天然高	

成本影响（98% 缓存折扣）：

计费项	未命中价	缓存命中价	折扣	
输入（Flash）	$0.14/M	$0.0028/M	98%	
输入（Pro）	$0.435/M	$0.003625/M	99%+	

实际意义：一个长会话 Agent 任务，绝大部分输入 token 走缓存价——实测中"50 步工具链任务、2.4M 输入 token"的成本远低于按未命中价计算的预期。对高频/批量/长会话场景，这是数量级的成本优势。

> *怎么让缓存命中率更高：**保持会话延续**（不要频繁新开会话）、**稳定 prompt 前缀**（系统提示/技能目录不要反复变化）、批量任务放同一会话/同前缀。*

5.2 场景一：日常编码（最适合入门的场景）

典型任务：

- 按需求写一个新函数/模块
- 修复 bug（定位 → 修改 → 验证）
- 补单元测试

任务示例（headless 一键跑）：

```
dsh --profile headless "审查 buggy_calculator.py：找出所有 bug 修复，写完整测试，运行验证全部通过" # 实测：5 个 bug 全修复 + 49 个测试通过（94s）
```

为什么 dsh 适合：编码是工具链最完整的场景——读文件、写代码、跑命令、看结果，闭环顺畅。产物（改的文件）会在对话末尾直接可打开。

提示词技巧：一定要写"运行验证"——dsh 会照着闭环；只写"帮我修 bug"它会修完就停。

5.3 场景二：数据处理与分析

典型任务：

- 数据质量检查与清洗
- 统计分析
- 可视化（图表）
- 生成分析报告

任务示例：

```
dsh --profile headless "分析 sales_data.json 的数据质量（缺失/类型错/重复/异常），写清洗脚本+可视化脚本，运行验证，总结处理方案" # 实测：52→35 行归零 + 图表 + 权衡说明（186s）
```

亮点（实测）：dsh 不仅执行，还会说明权衡——"中位数填充会产生尖峰""异常值是否该删取决于业务"——这是 agent 工程层"会思考"的直接证据。

5.4 场景三：文档与知识工作

典型任务：

- 从代码生成 API 文档
- 把笔记整理成结构化文档
- 总结长文、翻译

任务示例：

```
dsh --profile headless "为 user_module.py（8 个函数）生成完整 Markdown API 文档：参数表、返回值、异常、示例、边界情况" # 实测：423 行 ReadTheDocs 风格文档（1m04s），主动识别"文档契约 vs 代码实现"差异
```

亮点（实测）：dsh 会防御式写作——标注调用陷阱（负数 limit 行为、空参数行为）、发现文档声明与实现不符的地方如实说明。

5.5 场景四：自动化与运维（headless 是王牌）

典型任务：

- 每天定时跑数据任务（日报、监控摘要）
- 批量文件处理（重命名、格式转换）
- CI 里的自动化检查

任务示例（脚本化）：

```
# cron 每天 8 点生成日报 dsh --profile headless "读取工作区 git log，生成中文日报" > daily-report.md  # 批处理：目录里所有 .txt 转 .md dsh --profile headless "把 docs/ 下所有 .txt 转成 .md（保留结构）"
```

为什么 headless 是王牌：进程级工具（bash/文件/搜索）全可用 + 非零退出码即失败 + 输出可管道——它是"能写进 CI 的 Agent"。

5.6 场景五：代码理解与重构

典型任务：

- 把过程式代码重构为 OOP
- 全仓库代码审查
- 技术调研（"这个模块怎么工作的"）

任务示例：

```
dsh --profile headless "把 legacy_orders.py 重构为面向对象+职责分离，保持行为一致，写测试验证重构前后输出相同" # 实测：17/17 测试通过，含脚本级产物对比（4m37s）
```

亮点（实测）：dsh 的工程意识——识别重复代码合并单一入口、保持向后兼容、发现沙箱下 tempfile 权限问题主动换方案。

5.7 让场景落地的三个提示词法则

1. **写验收标准，不写过程**：`"运行验证通过"` > `"写一个脚本"`——dsh 自己会编排步骤
2. **给上下文**：一句话说清背景（文件在哪、数据长什么样、目标读者是谁）
3. **一次一个任务**：复杂目标拆成多轮（每轮一个小闭环），比一次给巨型任务更可靠

5.8 我的第一个真实任务（15 分钟上手）

1. 建一个测试目录，放一个你手头的小文件（代码/数据/笔记）
2. 跑：`dsh --profile headless "总结这个文件，指出 3 个可改进点"`
3. 跑：`dsh --profile headless "给这个文件写一个测试/生成文档"`
4. 对比输出，感受 dsh 的"完成度"——它是不是只给了个大概？验收标准写清楚了吗？

动手练习

1. **场景识别**：给你手头的 3 个日常任务，判断各属于哪个场景（编码/数据/文档/自动化/理解）
2. **提示词对比**：对同一任务写两个版本提示词——一个"描述过程"、一个"写验收标准"，跑一遍对比结果
3. **headless 脚本化**：把"每日 git 日报"写成一行 cron/bash 命令，验证输出
4. **多步任务**：设计一个跨场景任务（如"读代码 → 生成文档 → 输出报告"），看 dsh 如何编排
5. **思考题**：什么场景 dsh 不适合？（提示：实时交互、需要人工审批的敏感操作）

常见疑问（FAQ）

- **Q：dsh 只是编码助手吗？** 不是——工具链（读/写/跑/搜）让它通用，数据处理/文档/自动化都验证过
- **Q：复杂任务会跑多久？** 实测 1-5 分钟（V4-Flash）；要更快可以降推理档（第 6 章）
- **Q：任务失败怎么办？** 看输出里的工具调用轨迹定位；多数是"验收标准不清晰"，重写提示词重跑
- **Q：能自动化运行吗？** 能——headless + 非零退出码 + 管道输出，可进 cron/CI
- **Q：安全吗？** 工具执行有沙箱与审批层；敏感操作建议人工确认（第 8 章安全模型）
- **Q：和直接用 Claude Code 有什么区别？** 同模型下能力接近，但 dsh 可自定义工具链/界面/插件（第 1 章矩阵）

下一章：[第 6 章：进阶与性能调优](./06-advanced.md)

5.9 行业视角：dsh 在不同领域的打开方式

同一套工具链，不同行业的用法。以下为通用场景（不含真实业务数据），合规敏感领域（医疗/法律）请务必人工审核。

金融与投研

- **任务**：财报摘要、研报要点提取、因子说明文档、投资备忘
- **示例**：`dsh --profile headless "读取这份财务数据，生成含同比/环比变化的摘要报告"`
- **dsh 优势**：数据处理管线顺手 + 高缓存命中（长报告会话便宜）

教育与学习

- **任务**：教案生成、题目解析、错题归纳、学习计划
- **示例**：`dsh --profile headless "把这章知识点整理成 10 道自测题（含解析）"`
- **dsh 优势**：文档生成 + 结构化为强项，可批量出题/归纳

销售与客服

- **任务**：FAQ 整理、话术生成、客户反馈归类
- **示例**：`dsh --profile headless "把 100 条客户反馈按问题类型归类并给出改进建议"`
- **dsh 优势**：批量文本处理 + 分类归纳

运营与内容

- **任务**：周报月报、数据看板说明、内容选题、SEO 摘要
- **示例**：`dsh --profile headless "读取本周数据文件，生成运营周报（含图表代码）"`
- **dsh 优势**：数据 + 文档 + 自动化的组合

研究与综述（学术/技术）

- **任务**：论文要点提取、技术调研、文献对比
- **示例**：`dsh --profile headless "对比这 3 篇论文的方法与结论，输出对比表"`
- **dsh 优势**：长文档处理 + 结构化输出（1M 上下文）

⚠ 合规提醒

医疗诊断、法律意见、财务建议等高风险场景：dsh 输出需人工审核，不能直接用于决策——工具执行有沙箱，但内容责任在使用者。

第 6 章：进阶与性能调优

本章目标：从"能跑"到"跑得好"——推理档位策略、工具调用耗时分析、真实踩坑清单。

TL;DR（本章核心，30 秒版）

1. **时间花在哪**：模型思考占 ~90%，工具执行 <1%，网络/渲染 ~10%——优化思考时间是性价比最高的提速
2. **三档策略**：`low`（简单/批量轮次）、`high`（日常默认）、`max`（复杂推理/debug）——手动 UI 选或插件自动调
3. **耗时可视化**：Web UI 底部统计行（LLM Xs / 工具调用 Ys）是最快的瓶颈定位手段
4. **8 个真实坑**：rc.1 依赖断裂、插件缺 main、next() 忘 await、事件类型不识别、client 测试跑不了、缓存命中误判、端口占用、推理序列化丢 reasoning（#739）
5. **评测三问**：谁测的、什么 harness、验证器多严——跑分要带条件看

<details><summary>本章导航</summary>

</details>

6.1 性能模型：dsh 的时间花在哪

实测一个"创建文件"任务的耗时分布：

阶段	占比	说明	
模型思考（Think）	~90%	每次工具调用前都会重新思考，是绝对大头	
工具执行	<1%	文件写入等毫秒级	
网络/渲染	~10%	API 往返 + UI 更新	

推论：

- 简单任务 → 优化思考时间（降推理档）
- 长工具链任务 → 每步都省思考时间，累计收益最大
- 工具本身慢（搜索/大文件）→ 优化工具实现，不是调档

6.2 reasoning_effort 策略（官方三档）

<!-- [fix] 技术准确性核验："官方三档"在默认 DeepSeek 适配器（deepseek-official）下为 off / high / max；low 是 pi-ai（opencode-go）网关档位（本白皮书实测环境，即 benchmark 所走网关）。工具链降档叙事在网关环境下成立，settings.yaml 对默认 provider 请用 off -->

档位	场景建议	
low	简单/确定性轮次：文件操作、批量、工具链中的廉价步	
high	日常 Agent 任务（默认）	
max	复杂推理、长链规划、debug	

手动：UI 的"推理等级"选择器，或 ~/.dsh/settings.yaml 的 reasoningEffort。

自动：插件按工具轮次动态降档（示例提速插件，第 4 章）。

6.3 工具调用耗时可视化

dsh 的会话统计行（Web UI 底部）显示：N 轮 · M 步 | LLM Xs · 工具调用 Ys | 首 token 平均 ...——这是最快的瓶颈定位手段。

进阶：host 插件监听工具事件做 per-tool 计时（提速插件示例已内置 host 日志版本），把"哪个工具最慢"暴露出来。

6.4 常见坑清单（真实踩过，含解法）

#	坑	现象	解法	
1	rc.1 依赖断裂	pnpm install 404（dsh-type-meta 等从未发布）	依赖用 ^0.1.0-rc.6 线	
2	插件缺 main	No "exports" main defined	暴露 . 入口；"main": "src/index.ts" 可被 tsx 加载	
3	next 忘 await	provider/model 丢失报错	agent/request 的 next 返回 Promise，必须 await	
4	事件类型不识别	'agent/request' is not assignable to keyof Events	npm 未 re-export 类型增强，边界放宽签名	

5	client 测试跑不了	jsdom 报 window.ModuleLoader undefined	client 产物依赖 dsh 引导机制，组件测试在官方 CI 跑	
6	简单任务"突然变快"的误判	1s vs 110s 差异被误归因	DeepSeek context cache 命中也会提速——A/B 测试要用全新 prompt	
7	端口占用	dsh web 起不来	netstat -ano \	findstr 3080 找 PID kill
8	推理序列化省略空 reasoning（#739https://github.com/deepseek-ai/deepseek-harness/discussions/739）	验证点：off（thinking:disabled）档正常（仅 high/max 触发）；high/max 下工具调用轮次没有思考内容时，serializeAssistant 里 toolCalls.length > 0 && reasoning.length > 0 条件把空的 reasoning 字段省略 → 下一轮请求报 400（invalid pi-ai replay state）	判断为 bug（应保留 reasoning 占位）：修复方向是序列化时始终保留该字段	

6.5 评测视角：官方成绩单 vs 独立实测

（结合 0813 正式版发布）看 Agent 模型成绩单要三问：

1. **谁测的**？官方自测（自家 Harness）vs 独立第三方（AA 等）
2. **什么 harness**？框架不同分数差很大（官方 Terminal-Bench 87.9 vs AA 独立 79）
3. **验证器多严**？宽松验证器（SWE-bench Verified 8.5% 假阳性）vs 严格（DeepSWE 0.3%）

dsh 的 agent/request waterfall 让模型跑分可复现——这是它相比闭源产品的工程优势。

6.6 缓存策略：让每次调用都更便宜

第 5 章讲了高缓存命中率的价值（实测会话缓存命中率 97%；缓存折扣 Flash 档 98% / Pro 档 99%+），这里给出可操作策略：

目标	做法	

提高命中	长任务保持会话延续（避免频繁新建会话）	
提高命中	系统提示/技能目录等前缀保持稳定（不要频繁改配置）	
提高命中	批量任务放同一会话/同前缀（如同一批文件分析）	
降低成本	简单轮次用 low 档（思考 token 减少 → 总 token 下降）	
监控	会话统计行看"缓存命中 %"（Web UI 底部）——低于预期就检查前缀稳定性	

成本模型速记：总成本 ≈ 输出 token×输出价 + 输入未命中×未命中价 + 输入命中×命中价——Agent 工作负载输入占比高，缓存命中率是成本的第一变量。

动手练习（检验你是否真懂了）

1. **理解题**：不看原文，说出 dsh 任务耗时的三个组成部分及各自占比。为什么"优化思考时间"是性价比最高的提速手段？

> 自查：参考本章 6.1 节耗时分布表格

2. **理解题**：解释 `low` / `high` / `max` 三个推理档位分别适合什么场景。如果一个任务"需要读 10 个文件然后做简单汇总"，应该用哪个档位？

> 自查：参考本章 6.2 节三档策略表格

3. **动手题**：打开 `dsh web`，跑一个"创建 3 个文件"的任务，观察 Web UI 底部的统计行（LLM Xs / 工具调用 Ys），记录数据并分析瓶颈在哪

> 自查：参考本章 6.3 节"耗时可视化"段落

4. **动手题**：把 `~/.dsh/settings.yaml` 的 `reasoningEffort` 从 `high` 改为 `low`，跑同一个简单任务，对比耗时差异。再改回 `high`，跑一个复杂推理任务，对比差异

> 自查：参考本章 6.2 节"手动"段落

5. **动手题**：模拟"端口占用"坑（先起一个服务占 3080 端口），用 `netstat -ano | findstr 3080` 找到 PID 并 kill，然后正常启动 `dsh web`

> 自查：参考本章 6.4 节坑 #7

6. **思考题**：本章 6.5 节说"评测要三问：谁测的、什么 harness、验证器多严"。请用一个具体例子解释：同一个模型，在不同 harness 下跑分为什么会差很多？

> 自查：参考本章 6.5 节"官方 Terminal-Bench 87.9 vs AA 独立 79"的例子

常见疑问 FAQ

Q1：为什么我的 dsh "有时候特别快，有时候特别慢"？

两个主要原因：① DeepSeek context cache 命中时，首轮上下文注入会快很多（1s vs 110s 的差异可能来自这里）；② 推理档位不同，思考时间差几倍。做 A/B 对比测试时，要用全新 prompt（避免缓存命中）+ 固定档位。

Q2：示例提速插件真的能提速多少？有没有实测数据？

提速收益取决于任务类型。简单任务（1-3 步）dsh 本来就快，插件效果不明显。长工具链任务（20-50 步）收益最大，因为每步思考降档的累计效果显著。具体数据参考第 4 章示例代码的实测日志。

Q3：我把推理档位设为 low，模型质量会下降很多吗？

取决于任务。简单文件操作、批量处理、确定性任务用 low 质量几乎无差别。复杂推理、长链规划、debug 用 low 可能漏掉关键步骤。建议：日常用 high，批量/简单任务手动切 low 或用提速插件自动降档。

Q4：坑 #6 说的"context cache 命中"是什么？怎么判断是否命中？

DeepSeek API 会对重复/相似的上下文做缓存，命中时首 token 时间大幅缩短。判断方法：看 dsh 会话日志里的"首 token 平均"指标，如果某次突然从 10s+ 降到 1s 以下，大概率命中了缓存。做性能对比测试时，用全新 prompt 避免缓存干扰。

Q5：rc 阶段的破坏性变更会影响性能吗？升级后需要注意什么？

可能影响。rc 阶段的 agent-loop、waterfall 签名、插件 API 都可能变。升级后：① 跑一遍 dsh --version 确认版本；② 看官方 changelog 有没有 breaking changes；③ 跑一个你熟悉的任务对比耗时和行为；④ 自定义插件检查类型兼容性。

Q6：我想做性能基准测试（benchmark），有什么建议？

三个原则：① 固定环境（同模型、同网络、同档位）；② 用全新 prompt 避免缓存；③ 多次运行取中位数（单次波动大）。记录：墙钟时间、LLM 耗时、工具调用耗时、首 token 时间。参考本章 6.1 节的耗时分布模型设计你的 benchmark。

下一章：[第 7 章：生态与资源](./07-ecosystem.md) —— 加入 dsh 生态的完整地图。

第 7 章：生态与资源

本章目标：给你一份"加入 dsh 生态"的地图——官方入口、社区现状、插件实操、以及参与方式。

TL;DR（本章核心，30 秒版）

1. **官方入口**：GitHub 仓库（源码 + issue）、官方文档站（VitePress）、API 文档、Discord、Discussions——当前贡献入口以 Discussions 为主
2. **外部 PR 暂不接受**（2026-08-13 时点），但官方鼓励做 dsh-plugin 生态项目、写教程/博客——社区已自发形成"报告-复现-根因-修复分支"的贡献模板（见 7.2 节）
3. **插件生态快照**：`DSH-better-sidebar`（最完整社区插件）、示例提速插件（第 4 章完整拆解）、本白皮书——发现插件搜 `topic:dsh-plugin`
4. **新手参与路径**：用起来 → 小改进（给社区插件提 PR）→ 发插件 → 写内容
5. **生态零日 = 先发优势**：每个早期生态都有"第一个吃螃蟹的人"的红利，现在入场正是时候

<details><summary>本章导航</summary>

</details>

7.1 官方入口

资源	地址	用途	
官方仓库	https://github.com/deepseek-ai/deepseek-harness	源码、架构文档、issue	
官方文档站	https://deepseek-harness.github.io/deepseek-harness/	官方 VitePress 文档站（guide/providers/development/reference，2026-08 上线）	

产品站点	https://deepseek.com/harness/	官方产品介绍（四种运行模式、轨迹视图等）	
API 文档	https://api-docs.deepseek.com	模型、定价、API 指南	
Discord	官方 README 内链接	社区讨论	
Discussions	官方仓库 Discussions	提案/求助（官方当前建议的贡献入口）	

注意：官方 CONTRIBUTING 明确"目前不接受外部 PR"（2026-08-13 时点）——但鼓励：

- 在 Discussions 提建议（官方会评估）
- **做 dsh-plugin 生态项目**（官方点名认可的方式）
- 写教程/博客

***官方自建文档的补充**：除了上述入口，官方仓库还维护了系统化的内部文档——`packages/README.md`（包清单权威表）、`docs/subsystems/`（子系统设计）、`docs/cookbook/`（9 个开发指南）、`docs/tool-catalog.md`/`docs/config-catalog.md`/`docs/module-graph.md`（自动生成清单）。白皮书是"新手视角的补充"，官方文档是"架构权威"，两者搭配读。*

7.2 社区贡献模式

官方暂不接受外部 PR，但社区已自发形成一套成熟的"报告-复现-根因-修复分支"贡献模板——[#341](https://github.com/deepseek-ai/deepseek-harness/discussions/341)、[#371](https://github.com/deepseek-ai/deepseek-harness/discussions/371)、[#775](https://github.com/deepseek-ai/deepseek-harness/discussions/775) 反复呼吁官方开放 Issues/PR，官方尚未回应，于是贡献者们把修复直接做进帖子里。

社区贡献模板四步：

1. **报告**：以 Discussion 发帖，标题带【Bug】，附 OS / Node / dsh 版本
2. **复现**：最小复现步骤 + 报错日志前 20 行
3. **根因**：定位到具体文件与函数（如 readUtf16 只判低字节 0x00）
4. **修复分支**：fork 后提交修复，帖内贴 `git cherry-pick <commit>` 命令

自带 cherry-pick 修复的示例帖：

帖号	问题	修复形态	
#244https://github.com/deepseek-ai/deepseek-harness/discussions/244	Windows 目录选择器截断含"需求"等汉字的路径	附修复补丁	
#295https://github.com/deepseek-ai/deepseek-harness/	创建工作区时中文目录路径被截断	复现 + 修复代码片段	

discussions/295			
#580https://github.com/deepseek-ai/deepseek-harness/discussions/580	Win32 原生目录选择器在 U+XX00 处截断 UTF-16 路径	附可 cherry-pick 修复	

*这些帖的共同特征：**不是"求官方修"，而是"我修了，你们用"**。在官方 Issues 关闭的阶段，这是 dsh 社区最高效的协作方式——既是在帮官方分流，也是在建立自己的社区信用。*

7.3 插件生态（2026-08-13 快照）

项目	定位	状态	
DSH-better-sidebar	文件管理/终端/Git/浏览器侧边栏	社区最完整插件	
示例提速插件	工具调用提速（reasoningeffort 自动调节）	教学示例（第 4 章）	
dsh-handbook（本白皮书）	新手教程	生态文档	
DeepSeek Desktop	Windows 桌面端（x64 社区安装包，v0.2.0 离线安装器）	#872https://github.com/deepseek-ai/deepseek-harness/discussions/872，同族 #529https://github.com/deepseek-ai/deepseek-harness/discussions/529 #446https://github.com/deepseek-ai/deepseek-harness/discussions/446	
turtle-ui	终端 TUI 界面插件（turtle1999/turtle-uihttps://github.com/turtle1999/turtle-ui）	社区 TUI 方案，安装尝试见 #871https://github.com/deepseek-ai/deepseek-harness/discussions/871	

memory 插件族	跨会话长期记忆（设计提案 / 长期记忆 / MEMORY.md·USER.md 移植）	#192https://github.com/deepseek-ai/deepseek-harness/discussions/192 #484https://github.com/deepseek-ai/deepseek-harness/discussions/484 #525https://github.com/deepseek-ai/deepseek-harness/discussions/525	
dsh-sgme	记忆引擎：对话历史与长期记忆分层，提炼前自动剪枝（省 65-96% 会话内容），按场景注入记忆块	freehul/sgmehttps://github.com/freehul/sgme（npm 包 dsh-sgme），#1052https://github.com/deepseek-ai/deepseek-harness/discussions/1052	
pi-quiet-tools	工具输出压缩：进上下文前把大结果压缩为头尾预览 + 本地 artifact（>12,000 字符或 240 行触发）	经 pi2dshhttps://github.com/weijiafu14/pi2dsh 挂载，#1052https://github.com/deepseek-ai/deepseek-harness/discussions/1052	
dsh-installers	免装 Node 安装包：mac DMG / Windows exe，自带 Node 运行时、零前置依赖	codeAnqiang-ma/dsh-installershttps://github.com/codeAnqiang-ma/dsh-installers（#380https://github.com/deepseek-ai/deepseek-harness/discussions/380 作者 codeAnqiang-ma 提供），安装见第 2 章 2.2 方式三	

发现插件：GitHub 搜 topic:dsh-plugin。

发布插件：给你的仓库加 dsh-plugin topic + npm 发布。

7.4 如何参与生态（新手路径）

1. **用起来**：`dsh web` + 装两个社区插件，跑通日常
2. **小改进**：给社区插件提 PR（读第 5 章的三个案例，那是完整的 PR 范式）
3. **发插件**：从第 4 章的最小 host 插件起步，挂 `dsh-plugin` topic
4. **写内容**：教程/测评/避坑文（官方鼓励），与本白皮书互相引用

7.5 推荐阅读路径

目标	路径	
快速上手	第 2 章 → 装 better-sidebar → 日常用	
开发插件	第 3 章 → 第 4 章 → 抄第 5 章案例	
性能调优	第 6 章 → 第 4 章示例源码	
深度定制	官方 AGENTS.md（架构）→ docs/architecture.md → packages/ 源码	

7.6 真实插件安装实操

以下给出两个代表性插件的完整挂载步骤（语法与第 3 章 3.2 节一致）。

安装 `DSH-better-sidebar`

① package.json 加依赖（~/.dsh/profiles/web/package.json）：

```
{   "dependencies": {     "dsh-better-sidebar": "link:C:\\path\\to\\DSH-better-sidebar"   }
}
```

② cordis.patch.yml 加挂载行：

```
- insert:     - id: better-sidebar       name: dsh-better-sidebar
```

③ 安装并验证：

```
cd ~/.dsh/profiles/web && pnpm install && dsh web
```

重启后观察左侧是否出现文件管理/终端/Git/浏览器四个标签页。

安装示例提速插件

① package.json 加依赖：

```
{   "dependencies": {     "dsh-speed-plugin": "link:C:\\path\\to\\dsh-speed-plugin"   } }
```

② cordis.patch.yml 加挂载行：

```
- insert:     - id: speed-plugin       name: dsh-speed-plugin
```

③ 安装并验证：

```
cd ~/.dsh/profiles/web && pnpm install && dsh web
```

④ 验证效果：发一个"创建 3 个文件"的任务，观察日志是否出现 [speed-plugin] calls=[...] => reasoningEffort=low（参考第 4 章 4.5 节）。

> *⚠ 两个插件均要求 `@deepseek-ai/dsh-agent ^0.1.0-rc.6`，若 `pnpm install` 报 404，请检查版本线（见第 3 章 3.5 节）。*

7.7 发现插件的方法

渠道	具体操作	预期结果	
GitHub topic	https://github.com/topics/dsh-plugin 或搜索 topic:dsh-plugin	所有打了 topic 的仓库	
GitHub 全局搜索	dsh-plugin + 语言过滤 TypeScript	含关键词的仓库与代码	
npm 搜索	npm search dsh-plugin 或 https://www.npmjs.com/search?q=dsh-plugin	已发布的包（rc 阶段以 GitHub 为主）	
官方讨论区 Show and tell	https://github.com/deepseek-ai/deepseek-harness/discussions/categories/show-and-tell	社区自荐项目	
本白皮书引用链	第 4/5/10 章案例中的仓库链接	经实机验证的插件	

技巧：topic:dsh-plugin 是最精准的过滤方式。2026-08-13 时点结果仅个位数——正是入场机会。

7.8 参与生态的完整流程

把 7.4 节的"四步走"展开为可操作的决策树：

```
用起来 → 提 Issue 反馈 → 发插件（npm publish） → 写内容
```

npm publish 流程简述：

1. 确保 `package.json` 有 `"name": "dsh-xxx"` + `"main": "src/index.ts"` + 正确 `peerDependencies`
2. `npm login` → `npm publish --access public`
3. 给 GitHub 仓库加 `dsh-plugin` topic
4. 到官方 Discussions 的 Show and tell 分类发帖

参考 CONTRIBUTING.md：案例需含"场景 / 命令 / 耗时或产物 / 验证方式"——发插件时附上这些信息，转化率更高。

7.9 生态里程碑时间线

时间	事件	意义	
2026-08-13	dsh 开源（零日）	MIT 协议发布，Agent 可编程时代起点	
2026-08-13	60+ 官方包同步放出	packages/ 下工具/上下文/会话/子代理/MCP/工作流/安全/模型/界面全覆盖	
2026-08-13 当天	社区插件爆发	官方讨论区单日 30+ 帖，含插件踩坑、TUI 示例、Windows 路径 bug 等	
2026-08-13 当周	本白皮书发布	14 章中文教程 + Benchmark + 插件模板，补零日文档缺口	
进行中	官方讨论区持续活跃	每周 2-3 帖响应，已覆盖插件/路径/TUI/vision 等方向（见 7.10 节）	
规划中	插件模板扩展 / 视频教程 / CI 校验	ROADMAP.md P1 优先级	

关键认知：2026-08-13 不是"发布日"而是"生态零日"——官方包、社区插件、中文教程同一天就位。对参与者而言，现在入场 = 和官方基建同步成长。

7.10 与官方讨论区的联动

本白皮书已在官方 Discussions 响应 8 帖，以下是有效参与的实例：

帖子	主题	响应方式	可复用模式	
#380https://github.com/deepseek-ai/deepseek-harness/discussions/380	插件踩坑	结论整理进第 3 章 3.5 节"常见坑"	问题 → 复现 → 写教程 → 回帖引用	
#392https://github.com/deepseek-ai/	TUI examples	补进 docs/config-reference.md	官方缺示例 → 跑通 → 写参考 → 分享链接	

deepseek-harness/discussions/392				
#401https://github.com/deepseek-ai/deepseek-harness/discussions/401	Windows 路径 bug	记录进第 12 章跨平台短板	平台 bug → 记录边界 → 帮官方分流	
#384https://github.com/deepseek-ai/deepseek-harness/discussions/384	visionDS	更新能力矩阵中 vision 支持状态	新能力 → 调研 → 更新表格 → 确认	
#118https://github.com/deepseek-ai/deepseek-harness/discussions/118	通用讨论	沉淀进 FAQ（docs/faq.md 六类问答）	高频问答 → 分类沉淀 → 给链接	
#655https://github.com/deepseek-ai/deepseek-harness/discussions/655	社区五项目	梳理生态全景进第 7 章 7.3 节	整合碎片项目 → 形成生态地图	
#735https://github.com/deepseek-ai/deepseek-harness/discussions/735	token 成本	沉淀进第 6 章 6.6 节成本模型	成本测算 → 公式化沉淀	
#781https://github.com/deepseek-ai/deepseek-harness/discussions/781	LSP 提议	记录进第 11 章未来展望	前瞻特性 → 跟踪架构演进	

有效参与的 3 个原则：① 先自己跑通再回帖（带环境信息和报错日志）；② 把结论写成可引用内容（单条回复会沉底，写成章节/FAQ 才能持续产生价值）；③ 链接代替重复（回帖给链接，如"详见第 7 章 7.6 节"）。

结语

dsh 是 2026-08-13 才开源的项目——生态的每一天都是"早期"。白皮书会随 dsh 演进持续更新。如果某章命令失效，大概率是 rc 版本迭代所致——以官方 changelog 为准。

祝你在这个全新的生态里，抢到自己的位置。🚀

动手练习（检验你是否真懂了）

1. **理解题**：不看原文，说出 dsh 生态的 3 个官方入口（仓库/API 文档/社区）各自的用途

> 自查：参考本章 7.1 节官方入口表格

2. **理解题**：官方当前"不接受外部 PR"，但鼓励哪三种参与方式？为什么"做 dsh-plugin 生态项目"是官方点名认可的方式？

> 自查：参考本章 7.1 节"但鼓励"段落

3. **动手题**：在 GitHub 搜索 `topic:dsh-plugin`，列出你找到的 3 个社区插件，说出每个插件的定位

> 自查：参考本章 7.3 节"发现插件"段落 + 7.7 节搜索渠道表格

4. **动手题**：按本章 7.6 节的步骤，完整挂载 `DSH-better-sidebar` 或按第 4 章示例自己写一个提速插件，记录 `pnpm install` 的输出和验证结果

> 自查：参考本章 7.6 节两处改动的完整代码片段

5. **动手题**：给 `DSH-better-sidebar` 或自己写的提速插件的 README 提一个改进 PR（比如加一个安装步骤、修一个 typo），体验社区 PR 流程

> 自查：参考本章 7.4 节第 2 步 + 第 5 章案例的 PR 范式

6. **思考题**：本章 7.9 节时间线显示"60+ 官方包同步放出"和"社区插件爆发"发生在同一天。这对开源项目的生态策略有什么启示？

> 自查：参考本章 7.9 节时间线表格 + 7.10 节联动模式

常见疑问 FAQ

Q1：官方说"不接受外部 PR"，那我的插件代码放哪？

放在你自己的 GitHub 仓库，作为独立的 dsh-plugin 生态项目。官方鼓励这种方式——通过插件扩展能力，无需改 dsh 核心。给仓库加 dsh-plugin topic，npm 发布即可。

Q2：Discord 和 Discussions 有什么区别？

Discord 适合实时讨论；Discussions 适合正式提案、功能建议、求助（有记录可查）。官方当前建议的贡献入口是 Discussions。

Q3：我想写中文教程/博客，有格式要求吗？

没有官方格式要求。建议包含安装步骤、代码示例、实机截图/日志。可参考本白皮书结构（TL;DR + 分步讲解 + 练习）。

Q4：生态零日是什么意思？为什么说是"先发优势"？

生态零日 = 生态刚起步，内容/插件/教程几乎为零。先发优势 = 早期入场者容易成为"某个方向的标杆"。类比：2015 年写 React 教程的人、2018 年做 VS Code 插件的人。

Q5：rc 阶段迭代快，我的插件会不会刚写完就过时？

有可能。降低风险：① 依赖用 ^0.1.0-rc.6 线；② 关注官方 changelog；③ 逻辑用纯函数隔离，接入层薄一点，升级时只改接入层。

Q6：我想做 dsh 生态项目，但从哪开始？

从"自己的痛点"开始。比如：① 想要 TUI → 做 tui profile 插件；② 想要 token 追踪 → 做 cost-tracker 插件；③ 想要 diff 视图 → 做 diff 插件。参考本章 7.5 节阅读路径。

Q7：7.6 节的 link: 路径在 Windows 和 macOS 下写法一样吗？

不一样。Windows 用双反斜杠转义（link:C:\\path\\to\\plugin），macOS/Linux 用正斜杠（link:/path/to/plugin）。建议社区插件最终走 npm 发布，消除路径差异。

Q8：官方讨论区发帖有什么技巧，能让官方更快响应？

参考 7.10 节 3 个原则：带环境信息、带复现步骤、先搜索避免重复。功能建议说明"解决什么问题"比"怎么实现"更重要。

第 8 章：工具与上下文系统

*本章目标：理解 dsh 的"能力引擎"——模型能调用哪些工具、上下文是怎么喂给模型的、以及长对话怎么处理。**这是从"能跑"到"跑得明白"的关键一章。**

TL;DR（本章核心，30 秒版）

1. **60+ 官方能力包**：工具（fs/shell/web/skill/todo）、上下文（context/compaction）、会话、子代理、MCP、工作流、安全、模型、界面——一切皆插件
2. **内置工具名是简短动词**：`read`/`write`/`grep`/`glob`/`edit`/`bash`/`todo`/`skill`——写提示词直接说"读文件""搜索"即可
3. **工具返回 locations → 产物追踪**：模型改了什么文件，UI 能直接看到并可打开（对话末尾的产物 chips）
4. **上下文 = 系统提示 + 技能目录 + 对话历史 + 工具结果**：分层注入，每步请求携带工具 schema
5. **长对话自动压缩（compaction）**：检测溢出 → 修剪历史 → 可选摘要 → 失败路由到溢出代理。重要信息建议写进提示词
6. **插件行为可零成本验证**：mock llm + headless + 审计 dump（#462）——没有 API key 也能验证 waterfall 透传与工具注入
7. **工具链有三个已知坑**：code 模式 run_code/bash 的 description required 死循环（#558/#581/#689）；流式下工具名被抹空（#725 同族，见 FAQ）；run_code 内交互式工具异步结果被丢弃（#1476）

<details><summary>本章导航</summary>

</details>

8.1 官方能力包地图（60+ 包一览）

dsh 的能力全部以包形式提供（packages/<group>/<name>）。新手最需要认识的：

能力域	官方包	作用	
工具（tools）	fs/tool-fs、fs/tool-fs-search、fs/tool-str-replace-editor、shell/tool-bash、web、skill、todo	文件/终端/网页/技能/待办等可调用工具	
上下文（context）	context/、compaction/	请求上下文组装、长对话压缩	
会话（session）	session/	持久化、标题、遥测	
子代理（subagent）	subagent/	委派子任务	
MCP	mcp/	MCP 客户端（外部工具服务器）	
工作流（workflow）	workflow/	多步工作流编排	
安全（safety）	sandbox/、guard/、interaction/、credentials/	沙箱、循环卫生、权限/审批、凭证隔离（注：官方无独立 safety/ 组，安全策略分散在多个包组）	
模型（llm）	llm/、llm-deepseek、llm-retry	模型接入、重试	
技能（skill）	skill/	技能提供者注册表	
界面（client）	client/（ui-conversation、ui-tool…）	Web UI 各部件	

***完整清单与包数口径**：官方仓库 `packages/README.md` 是包清单权威表（47 组）；`@deepseek-ai/dsh` CLI 的 0.1.0-rc.6 声明 **53 个 `@deepseek-ai/dsh-*` 直接依赖**（+CLI 自身 = 54），家族发布总量约 **221 个**（含 devDeps 与历史 0.0.1-rc.1 列车，官方构建 commit 提及）。白皮书/官方所称"60+"指 `packages/` 下全部子目录。**npm 包名写法**（`@deepseek-ai/dsh-*`）与**仓库路径写法**（`fs/tool-fs`）的对应关系，以及每个包的实测描述，见 [附录 B：官方包速查大全](./appendix-packages.md)。*

***想深入架构**：官方 `docs/subsystems/`（session / system-prompt / tools / agent / agent-loop / scope / llm-streaming / subagent）是各子系统的设计文档，比本节的"能力域分组"更系统化；自动生成的权威清单见官方 `docs/tool-catalog.md`、`docs/config-catalog.md`、`docs/module-graph.md`。*

8.2 内置工具（实测观察）

模型实际可调用的工具名是简短动词（实测记录，来自 dsh web 会话与 agent/request 日志）：

工具名	作用	备注	
read	读文件	fs 能力	
write	写文件	fs 能力	
grep	内容搜索	fs-search	
glob	文件模式匹配	fs-search	
edit / strreplaceeditor	精准编辑	工具结果含 locations（用于产物追踪）	
bash / pwsh	执行命令	沙箱隔离	
todo	待办管理	长任务规划	
skill	技能调用	skill-catalog 注入	

工具结果与产物追踪（重要概念）：工具的返回里带有 locations（文件路径），dsh 用它们做"产物文件行"（对话结尾的产物 chips 就是从这里来的）——模型改了什么文件，UI 能直接看到并可打开。

8.3 上下文是怎么喂给模型的

一次模型请求的上下文 = 系统提示 + 技能目录 + 对话历史 + 工具结果。实测在会话日志中可见：

<!-- [style] 示意图代码块统一补 text 语言标签 -->

```
上下文注入 @deepseek-ai/dsh-system-prompt   ← 官方系统提示 上下文注入 skill-catalog
← 技能目录
```

dsh 的上下文机制：

- **系统提示分层**：官方插件通过 `systemPrompt.section()` 注册提示片段（如 ui-deliverables 注册"产物文件引用"指导）
- **技能目录注入**：可用技能列表进入上下文，模型按需调用
- **工具 schema**：每步请求携带工具定义

8.3.1 PTC 模式（Code mode）：一段代码编排多轮工具调用

PTC = Programmatic Tool Calling（程序化工具调用）——官方中文站称「PTC 模式」，官方英文页面对应 Code mode（[#1052](https://github.com/deepseek-ai/deepseek-harness/discussions/1052) 评论区社区详解；官方站点原文："PTC 模式通过模型生成的一段代码组合多轮工具调用"）。

与传统方式的区别：标准模式下模型每步发一次工具调用（一步一往返）；PTC 模式下模型生成一段 TypeScript 程序，在单次程序内编排多轮工具调用（循环、分支、Promise.all 并行），整段程序一次往返执行完毕：

维度	标准模式	PTC 模式（Code mode）	
调用形态	每步一次工具调用	一段程序编排多轮调用	
模型往返	5 次调用 ≈ 5 次往返	合并为 1 次往返	
中间结果	逐步回灌上下文	仅 return/console.log 回灌	
典型场景	日常对话、单步任务	长链路自动化、确定性执行	

执行机制：程序交给 @deepseek-ai/dsh-code-runtime-worker-thread 在全新 Node worker 线程里执行（类型剥离后可擦除 TS；空环境 + 堆/耗时/输出预算 + 硬终止）；程序内每个子工具调用仍走完整工具流水线（权限、沙箱、审计照常生效）。

收益：多步调用一次往返 + 中间结果不再全部塞回上下文 → token 开销大幅下降（#1052 评论区实测经验：PTC 模式是长任务降本手段之一）。

> *注意：PTC 模式仍有社区反馈的已知坑——`run_code`/`bash`的 description required 死循环（见 8.8 坑 1）；且长会话中多轮工具调用的中间产物仍会留在上下文（#1052 原帖实测 70M→100M token 暴涨即发生在 PTC 模式下，见 8.4 节自动压缩需求）。*

8.4 长对话：compaction（压缩）

长对话会撑爆上下文。dsh 的 compaction 插件（如 compaction-basic）负责：

- 检测上下文溢出（`agent/request-error` 的 `CONTEXT_WINDOW_EXCEEDED`）
- 压缩历史（模型无关的修剪 + 可选摘要）
- 失败时路由到"溢出代理"（overflow agent）

> ***自动压缩是社区高频需求**（[#1052](https://github.com/deepseek-ai/deepseek-harness/discussions/1052)）：用户反馈长会话 token 暴涨后"要开新会话"很痛苦，希望**自动压缩而非手动新会话**——社区回复确认"现在内置应该是没有自动压缩"（需装 compaction 插件/手动 compact）。想低成本延续长会话的临时做法：换新会话 + 按需注入记忆（记忆分层方案见第 7 章 7.3 节与第 14 章 14.7 节生态工具）。*
>
> *对新手：**知道"长对话会自动压缩"即可**，细节是进阶话题。生产环境注意：压缩会丢细节，重要上下文建议手动写进提示词。*

8.5 权限与安全模型（了解即可）

- **访问模式**：UI 里可见「Workspace Write」等模式（权限预设）
- **交互审批**：`interaction/*` 提供权限/审批能力（危险操作可要求确认）
- **沙箱**：`sandbox/*` 隔离命令执行（如 pwsh 沙箱有 ACL 约束——实测中遇到过 temp 目录权限问题）
- **工具超时**：`guard/*` 提供 loop 卫生与工具超时

*安全配置的深度话题超出本白皮书范围；核心认知：**dsh 的工具执行默认有隔离与审批层**，不是裸执行。*

8.6 新手最该记住的三件事

1. **工具名是简短动词**（read/write/grep/glob/edit/bash）——写提示词/插件时直接说"读文件""搜索"即可
2. **工具返回 locations → 产物追踪**——模型改的文件会出现在对话产物区
3. **长对话自动压缩**——不必手动清理历史（但重要信息要写进提示词）

8.7 插件运行时验证方法论（零成本）

出处：官方讨论区 [#462](https://github.com/deepseek-ai/deepseek-harness/discussions/462)「DSH 插件运行时验证方法论：无 API Key 验证 waterfall 行为」（实证 74 事件 / 12 waterfall）。FAQ 已有一句话版本，本节展开。

痛点：静态检查只能证明"插件能加载"，证明不了"插件在真实 agent 循环里不破坏行为"。尤其 waterfall 事件监听器（tools/execute、approval/request、fs/write-intent 等）必须正确透传 next()，否则会静默吞掉下游默认行为——这类错误只在运行时暴露；而传统验证要真实 LLM API key + token 成本。

零成本方案（dsh 仓库自带，无需 API key）：

1. **mock llm**：脚本化返回——让第一个请求返回"调用 bash 工具"、第二个正常回复（`--sequence tool_call_success,success`），以此注入工具调用
2. **headless profile**：走真实 agent 循环，你的插件挂监听
3. **审计 dump**：`DSH_EVENT_AUDIT_DUMP` 导出事件审计快照

```
# 1. 启动 mock-llm（脚本化注入"调用 bash 工具"） pnpm run mock:llm --port 8000 --api-key mock-key \   --sequence tool_call_success,success --repeat-last \   --tool-name bash --tool-arguments '{"command":"ls"}'  # 2. 跑 headless agent（指向 mock-llm，导出审计） DEEPSEEK_BASE_URL=http://127.0.0.1:8000/v1 \ DEEPSEEK_API_KEY=mock-key \ DSH_EVENT_AUDIT_DUMP=/tmp/audit.json \ pnpm dsh --profile headless "run the bash tool once and report"
```

判定通过：审计快照 byMode 同时含 emit + waterfall、waterfall 链完整出现、agent 正常跑完（mock response recovered）。实证：74 事件 / 12 waterfall 全部捕获，完整生命周期 session/created → … → tools/pre-execute(wf) → tools/execute(wf) → tools/post-execute(wf) → tools/result；关键安全证明是所有 waterfall 监听器正确透传 next()、零副作用——插件既不吞默认行为，mock 注入的工具调用也完整走完链路。

从源码跑的三点提醒（#462 作者实测踩坑）：sparse clone 一次加全（缺 vendor/ 会报 Cannot find package '@deepseek-ai/cordis'）；build:lib:host + build:lib:client + web-frontend dist 三层构建缺一不可；mock:llm 参数前不要多传 --（会被当位置参数）。

8.8 工具链踩坑

rc.6 时代的工具链已知坑（坑 1/2 已入 [FAQ](./faq.md)）：

坑 1：code 模式 runcode/bash 的 description required 死循环（[#558](https://github.com/deepseek-ai/deepseek-harness/discussions/558) [#581](https://github.com/deepseek-ai/deepseek-harness/discussions/581) [#689](https://github.com/deepseek-ai/deepseek-harness/discussions/689)）

code 模式下 runcode 与 bash 都把 UI 摘要字段叫 description 且标成 required：模型常把内层 bash.description 当成已传过，外层 JSON 只剩 {"code":"..."} → 反复报 missing required property "description"，看起来像"随机丢参数"。#581 补根因：ToolArgsError 不带工具名，模型无法定位错在哪个工具 → 死循环重试（附可 cherry-pick 修复）；#689 显示同族问题让 runcode 内所有 tools. 调用都被拒绝。规避：手动补外层 description 或换标准模式。

坑 2：流式下工具调用被抹空名（[#725](https://github.com/deepseek-ai/deepseek-harness/discussions/725) 同族，已入 [FAQ](./faq.md)）

现象：所有工具调用报 Error: unknown tool ""。根因：SSE 流式解析用覆盖赋值而非累加，把工具名/ID 抹成空串（[#725](https://github.com/deepseek-ai/deepseek-harness/discussions/725) 根因 + 修复；[#161](https://github.com/deepseek-ai/deepseek-harness/discussions/161) [#615](https://github.com/deepseek-ai/deepseek-harness/discussions/615) [#694](https://github.com/deepseek-ai/deepseek-harness/discussions/694) [#741](https://github.com/deepseek-ai/deepseek-harness/discussions/741) 同族）。官方修复前只能降级/等版本；模型会反复重试，注意及时中止。

坑 3：runcode 内调用交互式工具被"吞"（[#1476](https://github.com/deepseek-ai/deepseek-harness/discussions/1476)）

runcode 程序内调用交互式工具（如 askuserquestion）时：同步执行不等异步——4ms 就返回空；异步结果约 12s 后回来时会话帧已结束，被直接丢弃。判断为 runcode 执行模型的问题（应支持异步工具回调或返回 pending 状态），AGENTS.md 里"别在代码里调交互式工具"的约定只能治标。

动手练习（检验你是否真懂了）

1. **理解题**：不看原文，列出 dsh 内置的 8 个工具名（简短动词）及各自作用

> 自查：参考本章 8.2 节工具表格

2. **理解题**：解释"工具返回 locations → 产物追踪"是什么意思。模型改了什么文件，UI 怎么知道？

> 自查：参考本章 8.2 节"工具结果与产物追踪"段落

3. **动手题**：在 `dsh web` 里发一个"创建一个 hello.py 文件"的任务，观察对话末尾的"产物 chips"，点击打开文件，验证产物追踪是否工作

> 自查：参考本章 8.2 节"产物文件行"说明

4. **动手题**：在 `dsh web` 里进行一个长对话（20+ 轮），观察是否触发 compaction（看会话日志有没有"上下文溢出"或"压缩"相关日志）

> 自查：参考本章 8.4 节"长对话自动压缩"段落

5. **思考题**：为什么"重要信息要写进提示词"而不是依赖对话历史？compaction 压缩时会丢什么？

> 自查：参考本章 8.4 节"压缩会丢细节"警告

6. **思考题**：本章 8.5 节说"dsh 的工具执行默认有隔离与审批层"。如果你要做一个"允许模型自动执行所有 bash 命令"的插件，应该用哪个扩展点？有什么安全风险？

> 自查：参考本章 8.5 节"交互审批" + interaction/ 包说明

常见疑问 FAQ

本章针对工具/上下文的高频问答见下；更多通用问题见 [FAQ 速查](./faq.md)。

Q1：工具名是 read/write 这些简短动词，我写提示词时也要用这些名字吗？

不需要。你写"读一下这个文件""搜索包含 foo 的行"，模型会自动映射到对应工具。简短动词名是插件内部实现细节，提示词用自然语言即可。

Q2：edit 和 strreplaceeditor 是同一个工具吗？

是。strreplaceeditor 是内部包名，模型调用时可能显示为 edit。功能一样：精准替换文件中的某段文本。工具结果里的 locations 用于产物追踪。

Q3：compaction 压缩后，模型还能记得之前对话的内容吗？

记得"摘要"，但会丢细节。compaction 会修剪历史 + 可选生成摘要，关键信息保留，细节丢失。所以重要上下文（如项目约束、特殊要求）建议写进系统提示或提示词，不要只依赖对话历史。

Q4：context/ 和 compaction/ 有什么区别？

context/ 负责"组装每次请求的上下文"（系统提示 + 技能目录 + 对话历史 + 工具 schema）。compaction/ 负责"长对话压缩"（检测溢出 → 修剪 → 摘要）。一个是"装什么"，一个是"装不下时怎么压缩"。

Q5：安全模型里"沙箱"和"交互审批"有什么区别？

沙箱（sandbox/）隔离命令执行环境（如 pwsh 沙箱有 ACL 约束，限制文件访问）。交互审批（interaction/）是"危险操作前要求用户确认"（如删除文件、执行未知命令）。一个管"在哪执行"，一个管"能不能执行"。

Q6：我想给 dsh 加一个新工具（比如"查数据库"），应该怎么做？

两种方式：① 写一个 host 插件，用 ctx.provide 注册工具服务（参考第 3 章 3.4 节扩展点）；② 用 MCP 接入外部工具服务器（第 9 章 9.1 节）。前者适合深度集成，后者适合快速接入现有系统。

下一章：[第 9 章：MCP、子代理与工作流](./09-mcp-subagent-workflow.md)

第 9 章：MCP、子代理与工作流

*本章目标：了解 dsh 的三项扩展能力——MCP（接入外部工具）、子代理（并行任务）、工作流（多步编排）。**这些是把 dsh 从"单 Agent"升级为"Agent 系统"的开关。**
⚠ 本章基于官方架构文档（`packages/AGENTS.md`）与包结构整理；部分功能示例标注「待实测」——rc 版本迭代快，具体用法以官方 changelog 为准。*

TL;DR（本章核心，30 秒版）

1. **MCP = 给 Agent 插外部工具**：通过 MCP 协议接入数据库、浏览器、公司内部系统等——先跑通内置工具，有明确缺口再加 MCP
2. **子代理 = 并行干活**：把任务委派给子 Agent 并行执行（大仓调研、长任务分解、独立验证）——先用好单 Agent 再上子代理
3. **工作流 = 确定性流程编排**：和"多轮对话"不同，工作流把步骤定义成流程按顺序/条件执行——适合数据拉取→清洗→报表→校验
4. **组合起来 = Agent 系统**：父 Agent 规划 → 子代理并行调研 → 工具链 + MCP → 工作流汇总 → 产物
5. **新手路径四阶段**：单 Agent + 内置工具 → + MCP → + 子代理 → + 工作流（逐步叠加，别跳级）

<details><summary>本章导航</summary>

</details>

9.1 MCP：接入外部工具生态

MCP（Model Context Protocol）是"给 Agent 插外部工具"的开放协议。dsh 提供 mcp 客户端包（@deepseek-ai/dsh-mcp-client）。

能做什么：通过 MCP 接入任何 MCP 服务器（数据库、浏览器、图表、公司内部系统……）——比如社区已出现的 dsh-plugin-cost-tracker（追踪 token）就是 MCP/插件形态。

新手定位：MCP 是"生态扩展"，等你在 dsh 里有了明确的工具缺口再来接。先跑通内置工具，再加 MCP。

9.2 子代理（subagent）：并行干活

是什么：把任务委派给子 Agent 并行执行（packages/subagent/）。

典型场景：

- 大仓库多模块调研（各模块一个子代理）
- 长任务分解（父代理规划，子代理执行）
- 独立验证（子代理交叉检查）

对新手：子代理是"高阶武器"——先用好单 Agent，理解任务分解后再上子代理。错误示范是"什么任务都开子代理"，反而增加协调开销。

已知限制（社区反馈）：当前不能动态指定子代理模型——子代理模型跟随父 Agent 配置，社区已在讨论区反馈（[#118](https://github.com/deepseek-ai/deepseek-harness/discussions/118) 评论区，对比 Claude Code"可指定子代理模型"的体验）。需要不同模型能力时，目前只能整体切换父 Agent 的模型配置，或拆成独立会话分别指定。

9.3 工作流（workflow）：多步编排

是什么：packages/workflow/ 提供多步工作流编排（worker-thread provider + tool Consumer）。

和"多轮对话"的区别：普通对话是"模型自由发挥"，工作流是"把步骤定义成流程，按顺序/条件执行"——适合确定性流程（如：拉取数据 → 清洗 → 生成报表 → 校验）。

官方示例（packages/examples/）：有可运行的 cordis.yml 工作流示例。

9.4 组合起来：一个"Agent 系统"长什么样

<!-- [style] 流程图代码块统一补 text 语言标签 -->

```
你的提示词（目标）    ↓ 父 Agent（规划）    ├── 子代理 A：调研模块 A（并行）    ├── 子代理 B：调研模块 B（并行）    └── 工具链：read/grep/bash + MCP（数据库）    ↓ 工作流（如：校验 → 汇总 → 输出报告）    ↓ 产物（可打开/追踪）
```

新手路径建议：

1. **阶段一**：单 Agent + 内置工具（第 2-5 章）
2. **阶段二**：+ MCP（接外部工具）
3. **阶段三**：+ 子代理（并行）
4. **阶段四**：+ 工作流（确定性流程）

9.5 社区生态示例（2026-08-13 快照）

官方 Discussion 里已出现的社区项目：

- `dsh-plugin-cost-tracker`——实时追踪 token 成本（插件/MCP 形态）
- 插件开发/提速类（如本白皮书第 4 章的示例提速插件）

生态正在快速生长——现在是进场搭积木的好时候。

9.6 MCP 接入配置示例

以下给出在 cordis.patch.yml 中挂载 MCP 客户端的配置片段（语法与第 3 章 3.2 节一致，以社区 token 追踪插件为例）：

```
- insert:     - id: mcp-client       name: '@deepseek-ai/dsh-mcp-client'       config:
servers:           - name: cost-tracker             command: npx             args: ['-y',
'dsh-plugin-cost-tracker']             env:               DSH_API_KEY: '${DSH_API_KEY}'
```

字段说明：id（cordis 链标识）、name（npm 包名）、config.servers（服务器列表，每服务器独立进程）、command/args（启动命令与参数）、env（环境变量，支持 ${VAR} 引用宿主变量）。

> ⚠ *上述配置基于 `packages/mcp/` 目录结构与 cordis 挂载语法推断，具体字段以官方文档为准（rc 阶段可能变动）。验证思路（推断，待实测）：挂载后重启 `dsh web`，发涉及 token 计数的任务，观察日志是否出现 MCP 调用记录。*

9.7 子代理配置示例

并行子代理通过 packages/subagent/ 提供。以下是在 cordis.patch.yml 中启用子代理能力的配置片段：

```
- insert:     - id: subagent       name: '@deepseek-ai/dsh-subagent'       config:
maxConcurrency: 3         timeout: 120000
```

适用场景与并发策略：

场景	子代理数	父代理职责	子代理职责	
大仓库多模块调研	3-5 个	划分模块边界、汇总结论	各模块独立调研	
长任务分解	2-3 个	拆分子任务、校验完整性	各子任务独立执行	
独立验证	1-2 个	提供结论与验证标准	交叉检查	
多文件并行编辑	2-4 个	锁定文件范围、避免冲突	各子代理写不同文件	

> ⚠ *`maxConcurrency` 与 `timeout` 基于包名与 cordis 惯例推断（推断，待实测）。上下文隔离机制需查阅官方 `packages/subagent/README.md`。*

9.8 工作流 YAML 示例

以下是一个确定性工作流的 cordis.patch.yml 片段，模拟"数据拉取 → 清洗 → 报表 → 校验"四步（参考第 10 章案例 A 的任务结构）：

```
- insert:     - id: workflow       name: '@deepseek-ai/dsh-workflow'       config:
steps:           - id: fetch             tool: bash             command: 'curl -o raw.json
https://api.example.com/data'           - id: clean             tool: bash
command: 'python clean.py raw.json cleaned.json'            dependsOn: [fetch]           -
id: report             tool: bash             command: 'python visualize.py cleaned.json
chart.png'             dependsOn: [clean]           - id: validate             tool: bash
command: 'python verify.py cleaned.json'             dependsOn: [report]
```

与案例 A 的对应：工作流 dependsOn 显式定义顺序，而案例 A 的"多轮对话"由模型自由决定工具。工作流适合"步骤固定"；自由对话适合"探索性"。

> ⚠ *`steps`/`dependsOn`基于 `packages/workflow/`包名与常见 DSL 推断（推断，待实测）。官方 `packages/examples/`有可运行示例，建议以其为准。*

9.9 三者对比表

维度	MCP	子代理	工作流	
适用场景	接入外部系统（数据库/浏览器/内部 API）	大任务并行分解（调研/验证/多模块）	确定性流水线（拉取→清洗→报表→校验）	
复杂度	中（需配置服务器 + 理解协议）	高（需任务分解设计 + 结果汇总）	低-中（YAML 配置，调试依赖图较繁琐）	
失败处理	服务器崩溃 → 重连或报错（推断，待实测）	单个子代理失败 → 父代理决定重试/忽略/终止	单步失败 → 可选暂停/跳过/回滚（推断，待实测）	
性能开销	额外进程通信开销（stdio/sse）	并发节省墙钟时间，但增加协调开销	串行执行，无额外模型开销，但无并行收益	

决策建议：有明确工具缺口 → MCP；任务能拆成互不依赖的几块 → 子代理；步骤固定、需严格按序 → 工作流。三者可组合：子代理并行调研 → 内部用 MCP 查数据库 → 工作流汇总报告。

9.10 踩坑补充

#	坑	现象	解法	
1	MCP 连接失败	挂载后启动报错或 MCP 工具无响应	检查 command/args；先手动跑 npx -y xxx 验证（推断，待	

			实测）	
2	子代理上下文隔离	子代理看不到父 Agent 历史，导致重复提问	父代理委派时把关键上下文写进任务描述（推断，待实测）	
3	工作流单步失败	某步 bash 返回非零，后续步骤继续执行	配置 onError: pause 或每步后加校验脚本（推断，待实测）	
4	子代理文件冲突	多个子代理同时写同一个文件	父代理规划时划分文件边界（推断，待实测）	
5	MCP 环境变量未注入	${DSHAPIKEY} 解析为空	确认宿主 shell 已 export；或写死（仅本地测试）（推断，待实测）	

所有标注"推断，待实测"的内容均基于 `packages/` 目录结构与 cordis 配置惯例推断，非本白皮书实机验证结果。rc 阶段请以官方文档为准。

9.11 四阶段新手路径（含验收标准）

把 9.4 节的"四阶段"扩展为每阶段有明确验收标准的升级路径：

阶段	**能力**	**验收标准（怎么知道自己该进入下一阶段）**	**参考章节**	
阶段一	单 Agent + 内置工具	能独立完成：创建文件 → 搜索 → 编辑 → 运行验证，全程不卡壳	第 2-5 章	
阶段二	+ MCP	成功挂载一个 MCP 服务器，任务中调用其工具且日志证明调用发生	9.1 节 + 9.6 节	
阶段三	+ 子代理	成功发起一次并行子代理任务（如同时调研 2 个模块），父代理能汇总结果	9.2 节 + 9.7 节	
阶段四	+ 工作流	成功配置并执行一个 3 步以上工作流，每步按	9.3 节 + 9.8 节	

		dependsOn 顺序执行		

为什么不要跳级：阶段一没跑通 → 阶段二报错无法判断是配置错还是基础工具链不会用；阶段二没跑通 → 阶段三并行放大问题；阶段三没跑通 → 阶段四 YAML 写出来也是错的。阶段一到阶段二之间，先读第 8 章 8.1 节能力包地图——确认需求是否已内置，避免为"已有能力"配 MCP。

动手练习（检验你是否真懂了）

1. **理解题**：说出 MCP、子代理、工作流各自解决什么问题。三者之间有什么联系？（自查：9.1-9.3 节首段 + 9.9 节对比表）
2. **理解题**：解释"工作流"和"多轮对话"的区别。什么场景该用工作流？（自查：9.3 节 + 9.8 节）
3. **动手题**：在 `dsh web` 里发"调研仓库目录结构并生成报告"，观察是否自动调用 `read`/`glob`/`grep`。想加"查数据库"能力，该用 MCP 还是 host 插件？（自查：9.1 节 + 9.9 节）
4. **动手题**：设计"子代理并行调研"：调研 3 个模块（src/lib/src/cli/src/web），写出父 Agent 规划 + 3 个子代理任务描述（自查：9.2 节 + 9.7 节）
5. **思考题**：任务"读 100 个文件后汇总"，单 Agent 串行、5 个子代理并行、工作流分 5 批串行，各有什么优劣？（自查：9.9 节对比表）
6. **思考题**：想让 dsh "查公司内部员工信息"，走官方内置工具、MCP、host 插件三条路各需什么条件？哪条最快？（自查：第 8 章 8.1 节 + 本章 9.1 节 + 第 3 章 3.4 节）

常见疑问 FAQ

Q1：MCP 是什么？和 dsh 插件有什么区别？

MCP 是"给 Agent 插外部工具"的开放协议。插件在 dsh 运行时内注册能力（host 半跑在 Node 进程）；MCP 服务器是独立进程，通过协议通信。插件深度集成、性能好；MCP 解耦、可复用、适合接入现有系统。

Q2：子代理是怎么"并行"的？会不会有冲突？

子代理由父 Agent 委派，各自独立执行（packages/subagent/）。并行时各子代理有自己的上下文，不会互相干扰。但多个子代理改同一个文件可能冲突——建议任务分解时避免交叉。

Q3：工作流的"确定性流程"是什么意思？和模型自由发挥有什么区别？

工作流把步骤定义成流程（如 cordis.yml），按顺序/条件执行，每步做什么是指定的。多轮对话是"模型自由发挥"。确定性流程适合"必须按固定步骤走"的场景，自由发挥适合"探索性任务"。

Q4：我想接一个 MCP 服务器（比如数据库），具体怎么操作?

（以官方 changelog 为准）① 安装 @deepseek-ai/dsh-mcp-client；② 在 cordis.patch.yml 挂载（参考 9.6 节）；③ 配置服务器地址/命令；④ 重启 dsh web。详见官方 packages/mcp/ 文档。

Q5：子代理和"开多个 dsh 进程"有什么区别?

子代理是 dsh 内部委派机制（packages/subagent/），共享父 Agent 上下文和会话。多进程是完全独立的会话。子代理适合"任务分解后并行执行 + 结果汇总"，多进程适合"完全独立的任务"。

Q6：本章说部分功能"待实测"，我怎么知道哪些功能已经可用?

查官方 packages/AGENTS.md 和 packages/ 目录。有完整源码 + 示例的功能基本可用；只有包名没有文档的可能还在开发中。rc 阶段以官方 changelog 为准。

附录 A：[术语表与命令速查](./appendix-glossary.md)

第 10 章：复杂实战案例（dsh 真实跑出来的）

*本章目标：用两个**真实在 dsh 里跑完的复杂任务**，展示 dsh 在多步工具链下的实际能力、产出质量与耗时。*
***隐私声明**：两个案例全部使用**合成数据/自造代码**，不涉及任何真实业务数据或敏感信息。*

TL;DR（本章核心，30 秒版）

1. **案例 A（数据质量分析，186 秒）**：52 行脏数据 → 分析 → 清洗脚本 → 可视化 → 运行验证，52→35 行，所有问题归零
2. **案例 B（5-bug 修复 + 49 测试，94 秒）**：故意埋 5 个 bug 的计算器模块 → 全修复 → 49 个测试全过，覆盖边界/精度/异常
3. **dsh 的画像**：多步工具链自动编排、有判断力（主动说明数据权衡）、产物可追踪、耗时可控（94-186s）
4. **关键启示**：提示词把验收标准写清楚（如"运行验证"），dsh 会照着闭环——复杂任务建议明确"做什么 + 怎么验收"
5. **失败恢复**：本案例未触发；生产环境建议配 guard 超时 + 人工审批

<details><summary>本章导航</summary>

<!-- [fix] 链接健康检查：案例 A/B 锚点中 + 在 GitHub slug 里产生双连字符（--），原单连字符无法命中 -->

- [案例 A：数据质量分析 + 清洗 + 可视化（186 秒）](#案例-a数据质量分析--清洗--可视化186-秒)
- [案例 B：5-bug 代码修复 + 49 个测试（94 秒）](#案例-b5-bug-代码修复--49-个测试94-秒)
- [案例总结：dsh 在复杂任务上的画像](#案例总结dsh-在复杂任务上的画像)
- [案例 C：HTML 解析（Web 前端，约 3m18s）](#案例-chtml-解析web-前端约-3m18s)
- [案例 D：代码重构（软件工程，约 4m37s）](#案例-d代码重构软件工程约-4m37s)
- [案例 E：API 文档生成（文档，约 1m04s）](#案例-eapi-文档生成文档约-1m04s)
- [第二批案例画像](#第二批案例画像)

</details>

案例 A：数据质量分析 + 清洗 + 可视化（186 秒）

任务

给 dsh 一份含脏数据的合成 JSON（52 行：缺失值、类型错误、重复行、异常值 99999），要求：

1. 分析数据质量问题

2. 编写清洗脚本 `clean.py`
3. 编写可视化脚本 `visualize.py`（生成 `chart.png`）
4. 运行两个脚本验证

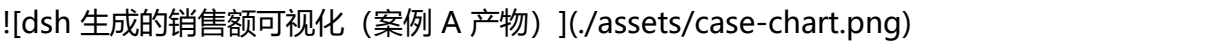
dsh 怎么做的（工具链）

<!-- [style] 工具链示意代码块统一补 text 语言标签 -->

```
read(sales_data.json) → write(clean.py) → bash(python clean.py) → write(visualize.py) → bash(python visualize.py) → read(输出) → 总结
```

产出与验证

产物	结果	
clean.py	✅ 运行成功，52 → 35 行，所有问题归零	
visualize.py + chart.png	✅ 生成有效 PNG（1950×825）	
清洗策略	缺失金额中位数填充、异常值删除、去重、类型统一	



输出要点（体现判断力，不是机械执行）

- 中位数填充会在直方图产生 510 尖峰——插补的固有副作用，建议说明
- 若 99999 是真实大额订单，建议改用 IQR/分位数法而非一刀切删除
- 清洗后统计：N=35，均值 489.43，中位数 510，标准差 181.73

dsh 不仅执行了任务，还主动说明了数据处理的权衡与假设——这是 agent 工程层"会思考"的证据。

案例 B：5-bug 代码修复 + 49 个测试（94 秒）

任务

给 dsh 一个故意埋了 5 个 bug 的 Python 计算器模块，要求：找到所有 bug → 修复 → 编写完整单元测试 → 运行验证。

埋的 bug

1. `add` 误写为 `a - b`
2. `divide` 除零静默返回 0（应抛异常）
3. `factorial` 负数静默返回 -1（应抛异常）
4. `factorial` 的 `range(1, n)` 漏乘 n
5. `format_money` 未格式化为两位小数

dsh 的修复 + 测试（验证：49 passed）

完整测试见 [case-test-calculator.py](./assets/case-test-calculator.py)——49 个用例覆盖：

函数	覆盖	
add	正/负/零/浮点/精度/交换律	
subtract	负结果/零边界/浮点	
multiply	符号组合/乘零/浮点	
divide	整除/符号/浮点/除零必须抛 ZeroDivisionError	
factorial	0!/1! 边界/已知值/负数必须抛 ValueError	
formatmoney	补零/四舍五入/0.1+0.2 舍入/负数	

```
python -m pytest test_calculator.py -v   # 49 passed in 0.37s
```

观察：dsh 的表现

- **完整闭环**：读 → 修 → 写测试 → 跑测试 → 总结，无人工干预
- **修复质量**：5 个 bug 全修复且补了 docstring 说明
- **测试设计**：覆盖边界（除零/负数/精度），不是"能跑就行"

案例总结：dsh 在复杂任务上的画像

维度	表现	
多步工具链	✅ 自动编排（读→写→跑→验证→总结）	
复杂任务耗时	94s-186s（同模型同网关，比 benchmark 简单任务慢但可控）	
产出质量	✅ 有判断力（数据权衡说明、测试边界设计）	
产物追踪	✅ 生成文件可在对话末尾直接打开	
失败恢复	本案例未触发；生产建议配 guard 超时 + 人工审批	

给新手的启示：dsh 处理"多文件、多步骤、要验证"的任务很顺手——这也是它作为 Agent 运行时的主要价值场景。复杂任务建议：提示词把验收标准写清楚（如"运行验证"），dsh 会照着闭环。

动手练习（检验你是否真懂了）

1. **理解题**：不看原文，说出案例 A（数据质量分析）的工具链顺序（read → write → bash → ...），以及最终产出是什么

> 自查：参考本章"案例 A"的"dsh 怎么做的"段落

2. **理解题**：案例 B（5-bug 修复）里，dsh 补的 49 个测试覆盖了哪些边界情况？为什么"除零必须抛 ZeroDivisionError"是一个重要的测试点？

> 自查：参考本章"案例 B"的测试覆盖表格

3. **动手题**：设计一个类似的"数据质量分析"任务：准备一份含脏数据的 CSV（至少 3 种问题：缺失值、类型错误、异常值），写提示词让 dsh 分析 + 清洗 + 可视化，记录耗时和产出

> 自查：参考本章案例 A 的任务描述 + 产出验证表格

4. **动手题**：设计一个类似的"bug 修复"任务：写一个故意含 3 个 bug 的 Python 模块（如字符串处理、日期计算），让 dsh 找 bug → 修复 → 写测试 → 运行验证，记录耗时和测试通过率

> 自查：参考本章案例 B 的任务描述 + "dsh 的表现"观察段落

5. **思考题**：案例 A 里 dsh "主动说明了数据处理的权衡与假设"（如中位数填充的尖峰副作用）。这个行为对"agent 工程层"的设计有什么启示？模型只是执行命令，还是真的有"判断力"？

> 自查：参考本章案例 A 的"输出要点"段落 + 案例总结表格

6. **思考题**：本章结语说"提示词把验收标准写清楚，dsh 会照着闭环"。如果案例 A 的提示词只说"处理一下这个数据"而不说"运行验证"，dsh 的表现会有什么不同？

> 自查：参考本章"给新手的启示"段落 + 案例 A 的工具链（含 bash 运行验证）

常见疑问 FAQ

Q1：案例 A 耗时 186 秒，案例 B 耗时 94 秒，这个速度算快还是慢？

取决于任务复杂度和模型档位。同模型同网关下，比 benchmark 简单任务慢但可控。关键不是绝对速度，而是"多步工具链自动编排 + 产出质量 + 无人工干预"的综合价值。如果想提速，参考第 6 章的推理档位策略 + 提速插件示例。

Q2：案例里的"合成数据"是什么意思？为什么不用真实数据？

隐私保护。本白皮书不涉及任何真实业务数据或敏感信息。合成数据是"故意构造的、不含真实信息的数据"，用于演示 dsh 的能力。你在自己环境里可以用真实数据，但要注意数据安全。

Q3：如果 dsh 在复杂任务中"卡住了"（比如某步工具调用失败），怎么办？

几种处理：① 看 dsh 进程日志，定位哪步失败；② 检查工具调用的输入是否正确（如文件路径、命令语法）；③ 配 guard 超时（guard/ 包），避免无限等待；④ 生产环境建议配人工审批（interaction/），危险操作前确认。

Q4：案例 B 的 49 个测试是 dsh 自动生成的，质量怎么样？

质量不错。覆盖了边界（除零/负数/精度）、异常（必须抛特定异常）、常规（正/负/零/浮点）。但自动生成的测试可能漏掉"业务逻辑特有的边界"——建议人工 review 测试设计，补充领域特定的用例。

Q5：我想让 dsh 跑一个"多文件、多步骤"的任务，提示词怎么写效果最好？

三个原则：① 明确目标（"做什么"）；② 明确验收标准（"怎么算完成"，如"运行验证""49 个测试全过"）；③ 明确约束（"不能做什么"，如"不能删原始数据"）。参考本章两个案例的提示词结构。

Q6：案例总结说"失败恢复本案例未触发"，那 dsh 的失败恢复能力怎么样？

dsh 有 compaction（上下文溢出恢复）、guard（超时）、interaction（审批）等机制，但复杂场景的失败恢复还在演进中。生产环境建议：① 配 guard 超时避免卡死；② 配人工审批避免误操作；③ 长任务分步验证，不要"一口气跑完再检查"。具体配置参考第 8 章 8.5 节安全模型。

附录 A：[术语表与命令速查](./appendix-glossary.md)

扩展案例（第二批，三个功能领域）

> *完整执行报告见 [case-expansion-report.md](./case-expansion-report.md)，产物在 `docs/assets/`。全部合成数据。*

案例 C：HTML 解析（Web 前端，约 3m18s）

任务：给定含表格+表单+图表的合成 HTML，编写零第三方依赖的解析脚本提取表格并输出 CSV。

dsh 的表现：

- **零依赖权衡**：主动选标准库 `html.parser`（而非 BeautifulSoup）
- **健壮性**：实现表格嵌套深度计数，防止越界
- **路径无关**：用 `__file__` 定位输入，不依赖运行目录

产物：[case-c-parser.py](./assets/case-c-parser.py) | 验证：输出 9 行×7 列 CSV，与源表逐字段一致

案例 D：代码重构（软件工程，约 4m37s）

任务：把约 200 行过程式订单脚本（含重复代码 calcordertotal / calcordertotalv2）重构为 OOP + 职责分离，行为一致，测试验证。

dsh 的表现：

- **精准识别重复**：合并两份相同逻辑为 `Order.total()` 单一入口
- **SOLID 设计**：Product/Customer/Order（模型）+ OrderProcessor（查询）+ ReportGenerator（报告）
- **向后兼容**：保留模块级函数接口，可直接替换
- **测试深度**：17/17 PASS——不仅对比函数返回值，还对比**脚本级整体运行产物**（report.txt / orders.json）
- **沙箱感知**：发现 `tempfile.mkdtemp` 在沙箱下 0700 ACL 不可写，主动换方案

产物：[case-d-ordersrefactored.py](./assets/case-d-ordersrefactored.py) + [case-d-testrefactor.py](./assets/case-d-testrefactor.py)｜验证：17/17 PASS

案例 E：API 文档生成（文档，约 1m04s）

任务：给含 8 个函数的 Python 模块生成完整 Markdown API 文档（参数表/返回值/异常/示例/边界）。

dsh 的表现：

- **契约差异识别**：主动发现"文档声明"与"代码实现"的差异（如 `user_id` 字符集未强制校验），如实写入边界章节
- **防御式文档**：为每个函数标注调用陷阱（负数 limit、空 updates 行为）
- **ReadTheDocs 风格**：423 行，8 个函数全覆盖，示例含 doctest 风格

产物：[case-e-api.md](./assets/case-e-api.md)

第二批案例画像

维度	表现	
跨领域能力	✅ Web/工程/文档三种任务都能闭环	
工程意识	✅ 零依赖权衡、沙箱感知、向后兼容、契约差异识别	
验证习惯	✅ 每个案例都运行验证（含脚本级产物对比）	
耗时	1-5 分钟/案例，复杂度越高越久	

总结：dsh 在"要写代码 + 要验证"的工程任务上表现稳定，且会做权衡（依赖/兼容/安全）——这是 agent 从"能答"到"能干活"的分水岭。

第 11 章：未来展望——dsh 与 Agent 生态的演进预测

*本章目标：从多个角度推演 dsh 及其生态的未来——**技术、生态、竞争、行业、开发者机会、风险**。预测不是断言，是"基于当前事实的推演"，供你判断是否值得投入。*

TL;DR（本章核心，30 秒版）

1. **短期（1-3 月）**：正式版 0.1.0 落地、插件生态爆发、教程/工具类项目吃第一波红利
2. **中期（1 年）**：dsh 成为"可编程 Agent 底座"的事实标准候选，垂直场景插件成熟
3. **长期（2-3 年）**：Agent 工程层标准化，dsh 生态与模型迭代相互成就
4. **最大风险**：破坏性变更、生态碎片化、官方策略转向
5. **个人机会**：现在入场（教程/插件/工具）是低成本的早期红利

<details><summary>本章导航</summary>

</details>

11.1 技术演进预测

时间	预测	依据	
短期	0.1.0 正式版发布（去掉 rc），破坏性变更收敛	当前 rc.6，官方明示"将有破坏性变更"，迭代极快	
短期	官方补上 TUI 与交互式 CLI（社区呼声最高，#172 15+ 评论）	官方 Discussion 最热需求	
中期	缓存/性能优化成为主旋律（reasoningeffort 精细化、缓存命中率工具化）	高缓存命中率已是成本优势（第 5 章），会被工具化显性化	
中期	插件生态标准化（插件市场/一键安装/质量分级）	官方已有 dsh-plugin topic，插件数量增长必然	

		催生市场	
长期	Agent 运行时架构沉淀为"工程范式"（类似当年 Linux 内核模式的 Agent 版）	"everything is a plugin" 已被验证为可组合架构	

11.2 生态发展预测

- **插件数量**：零日 → 1 年内预计数十到上百个（参照同类生态 S 曲线）
- **内容生态**：教程/博客/awesome 列表先爆发（中文内容目前是真空——本白皮书是第一份系统教程）
- **社区规模**：官方 Discussion 日均新增讨论（我们观察到单日 30+ 新话题），Discord 活跃度会同步增长
- **头部项目**：会诞生几个"生态旗舰"（类似 VSCode 生态的 Prettier/ESLint 地位）——**工具型 + 教程型最可能**

11.3 竞争格局预测

维度	dsh 的定位演进	
vs Claude Code	短期仍"开箱即用不如"，长期靠"可定制 + 开源 + 成本"差异化	
vs OpenCode	都开源；dsh 的官方后端 + 插件生态是差异点（OpenCode 无官方后端）	
vs 自建框架	dsh 是"半成品到成品的桥梁"——比自建省事，比闭源可控	
模型侧	DeepSeek 模型每代迭代都会放大 dsh 生态价值（模型 + 运行时协同）	

关键判断：dsh 的胜负手不在"模型多强"（那是 DeepSeek 的事），而在"Agent 工程层能否成为标准"——谁能定义"插件怎么写、生态怎么长"，谁就赢下这一层。

11.4 行业应用预测

- **金融/数据分析**：长会话 + 高缓存命中 = 成本友好的批量分析，最先落地
- **企业知识库/文档**：1M 上下文 + 文档工具链，企业内知识管理是天然场景
- **教育**：批量出题/解析/归纳，成本敏感场景吃缓存红利
- **垂直 Agent 产品**：会涌现"行业 Agent 封装"（在 dsh 上做行业插件/配置包）

11.5 开发者机会（现在入场的人）

角色	机会	时机	
教程/内容作者	中文教程真空（白皮书是第一份）	现在（红利最大）	
插件开发者	官方缺位方向：TUI、远程访问、移动端、缓存工具	现在	
工具型项目	提速、耗时可视化、MCP 工具包	现在（我们已做提速插件实验验证）	
生态基础设施	插件市场、模板生成器、评测工具	中期	
行业解决方案	金融/教育等行业插件包	中期	

11.6 风险与不确定性（诚实版）

1. **破坏性变更**：rc 阶段 API 可能频繁变动——插件/教程需持续跟进（白皮书会同步更新）
2. **生态碎片化**：插件质量参差，缺乏标准——早期需要"高质量示范项目"（本白皮书 + 第 4 章示例是示范之一）
3. **官方策略转向**：官方可能收回某些方向（如内置 TUI）——生态项目要有差异化壁垒
4. **模型竞争**：其他模型迭代可能削弱 DeepSeek 吸引力——但 dsh 模型无关的设计（可接 OpenAI 兼容模型）是缓冲
5. **热度回落**：如果 dsh 生态没起来，早期投入可能"白费"——但教程/技能本身是可迁移资产

11.7 白皮书自己的展望

- **内容**：随 dsh 迭代持续更新（版本号对齐 rc/正式版）
- **双语**：英文版完成度追上中文
- **案例**：更多行业真实案例（合规前提）
- **生态**：与 awesome 列表、官方 Discussion 联动，成为生态内容枢纽之一

11.8 给读者的最终建议

- **想试试**：现在装起来玩（第 2 章），成本几乎为零
- **想投入**：挑一个"官方缺位 + 你有积累"的方向（TUI/行业插件/教程）
- **想观望**：等 0.1.0 正式版，但会错过生态早期红利——**早期红利 ≈ 先发者的耐心**

预测会错，但"现在入场的成本最低"这一点，大概率不会错。

附录 A：[术语表与命令速查](./appendix-glossary.md)

第 12 章：已知不足与边界——诚实版

*本章目标：不吹不黑，列出 dsh 当前的**已知不足**与**使用边界**。写这本白皮书时 dsh 仍处于 `0.1.0-rc.6`（预发布阶段），这些不足是"此刻的事实"，不是"永远的宿命"——它们会随版本迭代变化。*

本章所有"已知问题"均来自本机实测 + 官方仓库 deepseek-ai/deepseek-harness 讨论区（截至 2026-08-14）的公开反馈。

TL;DR（本章核心，30 秒版）

1. **rc 阶段是最大的不确定性**：API/配置随时可能破坏性变更，投入前先锁版本线
2. **插件生态早期**：官方包 60+ 但覆盖有限，第三方插件少，交互形态（TUI）缺失
3. **跨平台短板**：Windows 上除路径/编码 bug（UTF-16、0x00）外，还有端口保留、局域网访问、沙箱稳定性、WSL 工作区、子进程终止等家族问题（速查表见 [12.4](#124-跨平台已知问题速查表)）
4. **性能边界未充分验证**：高并发、大型代码库场景缺少公开压测数据
5. **官方策略风险**：开源承诺 vs 商业化转向，是选型时必须考虑的黑天鹅

<!-- [fix] 结构核验：补齐本章导航（第 12 章为新增章节，原缺失，与第 3-11 章结构保持一致） -->

<details><summary>本章导航</summary>

- [12.1 rc 阶段的不稳定性](#121-rc-阶段的不稳定性)
- [12.2 插件生态早期](#122-插件生态早期)
- [12.3 跨平台短板](#123-跨平台短板)
- [12.4 跨平台已知问题速查表](#124-跨平台已知问题速查表)
- [12.5 性能边界未充分验证](#125-性能边界未充分验证)
- [12.6 与成熟 Agent 的差距](#126-与成熟-agent-的差距)
- [12.7 官方策略风险](#127-官方策略风险)
- [12.8 白皮书自身的局限](#128-白皮书自身的局限)

</details>

12.1 rc 阶段的不稳定性

破坏性变更随时发生

dsh 从 rc.1 到 rc.6 已经历过至少一次依赖断裂：早期版本 @deepseek-ai/ 包的版本线不一致会导致插件装不上（404/ERRMODULENOTFOUND）。rc.6 收敛了很多，但没有任何承诺说 rc.7/正式版不会再来一次。

对你的实际影响：

- 写教程/插件，可能每两周就要跟着改一遍
- 锁版本线（`^0.1.0-rc.6`）是基本操作，但 rc 线升级是"向上兼容"还是"推翻重来"，取决于官方心情

[!WARNING]

*生产环境请谨慎评估。白皮书写作时官方仍在快速迭代，**今日的写法明天可能就废弃**。*

已知的 rc 坑（本机/社区复现）

坑	表现	状态	
link 开发依赖断裂	@deepseek-ai/ 本地 link 时报 ERRMODULENOTFOUND，npm 安装却正常	已定位（扁平兜底目录机制），需锁依赖线规避	
Windows 路径 0x00	含中文/特殊字符路径 workspace 创建失败（ENOENT）	社区反馈中（#151/#396）	
fs-sandbox 竞态	post-check 路径检查与 workspace-write 存在时序竞态	社区反馈中（#159）	
TUI 缺失	官方 examples 无交互终端 UI（社区自建有，未纳入）	建议中（#392）	

12.2 插件生态早期

现状：官方包 60+，但生态整体处于"零日"状态。

- **第三方插件少**：讨论区插件帖在爆发，但质量参差，多数是"第一个插件"练手作
- **交互形态缺失**：headless/JSON-RPC/Web 形态齐全，**TUI 等终端交互形态无官方示例**（社区 deepseek-harness-tui 已证明可行，未纳入官方）
- **文档滞后于代码**：官方文档以架构为主，上手路径/避坑记录依赖社区自产（这正是本白皮书存在的理由）
- **版本线混乱**：npm 上同包多版本线并存，装错线就是 404

判断：生态玩法现在入场是早期红利（竞争少、官方扶持意愿强），但不要指望"照着官方文档就能跑"——踩坑是常态。

12.3 跨平台短板

dsh 在 macOS/Linux 上表现成熟，但 Windows 支持有明显的第二公民感：

- **路径处理**：中文路径、特殊字符路径有真实 bug（0x00 截断、ENOENT），社区多人反馈（根因 #107，家族 17+ 帖）
- **编码问题**：UTF-16 相关处理存在缺陷，终端输出偶发乱码
- **CLI/TUI 之争未定**：官方讨论区存在"CLI 才是正道 vs TUI 才是交互未来"的路线争议（#386），路线未明前，投资终端 UI 形态有风险

以下 Windows 家族问题来自官方讨论区挖掘报告（[docs/research/discussion-mining.md](./research/discussion-mining.md)，2026-08-14 抓取，rc.6 时代），均为社区真实复现，多数尚未修复。

端口：默认 3080 撞上 Hyper-V 保留区间（#589）

现象：Windows 上默认端口 3080 启动报 EACCES，服务起不来、web 界面打不开。

根因：启用 Hyper-V/WSL2 后，Windows 会保留一段动态端口区间（3070-3169），3080 恰好落在其中。

临时 workaround：换用保留区间外的端口（社区实测 13080 可用）。

局域网 HTTP 访问 RPC 全挂（#755）

现象：局域网内通过 HTTP（非 localhost、非 HTTPS）访问时，所有 RPC 调用失败。

根因：crypto.randomUUID 只在安全上下文（HTTPS 或 localhost）可用，局域网裸 HTTP 属非安全上下文。

临时 workaround：官方无修复；HTTPS 或隧道（如 Tailscale）访问可规避非安全上下文限制（按成因推论，未在社区帖内验证）。

沙箱：临时目录清理后永久崩溃（#758）

现象：Windows 上清理沙箱临时目录后 dsh 永久崩溃、不自愈（社区标 P0），并伴随 4 个相关稳定性问题。

影响：清一次临时目录 = 服务报废，无自动恢复路径。

临时 workaround：不要清理 dsh 沙箱临时目录；已触发则重启进程恢复。

WSL：工作区只能选 /root（#160）

现象：WSL/Ubuntu 环境下工作区选择器只能选 /root 内的目录，其余路径不可选。

临时 workaround：把项目放到 /root 下使用（限制内可用，限制外暂无绕过方案）。

子进程：Windows 无优雅终止（#717）

现象：Windows 下子进程管理系统性缺失——没有优雅终止阶梯、孙进程输出截断、spill 权限不生效、shellPath 默认指向 /bin/bash（Windows 上不存在该路径）。

影响：长任务终止不干净、输出丢失、权限预期失效，属平台适配缺口而非单个 bug。

临时 workaround：手动 taskkill /T 清理进程树；显式配置 shellPath 为 Windows 实际路径。

12.4 跨平台已知问题速查表

本章跨平台已知问题一表速查。帖号均为官方讨论区真实帖（deepseek-ai/deepseek-harness），截至 2026-08-14。

问题	环境	帖号	临时 workaround	
中文路径截断（readUtf16 单字节判 0，开 =U+5F00 被截）	Windows	#107https://github.com/deepseek-ai/deepseek-harness/discussions/107（根因；家族 #151/#396/#563 等 17+ 帖）	工作区路径避开中文/特殊字符；社区附修复补丁可 cherry-pick（#244/#580）	
目录选择器 worker 崩溃（koffi 相关）	Windows	#30https://github.com/deepseek-ai/deepseek-harness/discussions/30（家族 #154/#236/#449/#768）	安装层锁 koffi@3.1.2（#293）；选择器崩溃暂无绕行	
默认端口 3080 撞 Hyper-V 保留区间（EACCES）	Windows（启用 Hyper-V/WSL2）	#589https://github.com/deepseek-ai/deepseek-harness/discussions/589	换用保留区间外端口（如 13080）	
局域网 HTTP 访问 RPC 全挂（crypto.randomUUID 非安全上下文）	局域网 HTTP（非 localhost）	#755https://github.com/deepseek-ai/deepseek-harness/discussions/755	HTTPS/隧道访问可规避（推论）；官方无修复	
沙箱临时目录清理后永久崩溃（不自	Windows	#758https://github.com/	勿清理沙箱临时目录；已触发需重启	

愈，P0）		deepseek-ai/deepseek-harness/discussions/758	进程	
WSL 工作区只能选 /root 内目录	WSL/Ubuntu	#160https://github.com/deepseek-ai/deepseek-harness/discussions/160	项目放 /root 下使用	
Windows 子进程无优雅终止（孙进程输出截断/spill 失效/shellPath 错）	Windows	#717https://github.com/deepseek-ai/deepseek-harness/discussions/717	手动 taskkill /T 清理；显式配置 shellPath	

12.5 性能边界未充分验证

白皮书的 benchmark 覆盖了单 agent、常规任务规模，以下场景缺少公开数据：

- **高并发**：多 agent 并行、大规模任务分发
- **大型代码库**：百万行级仓库的上下文管理、工具调用延迟
- **长时运行**：连续数小时运行的稳定性、内存泄漏风险
- **缓存命中率的边界**：97% 实测命中率依赖"对话模式重复"——全新任务/冷启动场景会明显下降

12.6 与成熟 Agent 的差距

对照 Claude Code、Codex 等成熟产品，dsh 的差距是结构性的（不是补几个功能就能追上）：

维度	dsh（rc.6）	成熟 Agent	
生态阶段	零日，第三方资产稀少	成熟，生态完整	
开箱即用	需理解 profile/插件体系	装完就跑	
稳定性	rc 期，破坏性变更风险	稳定版，长期兼容承诺	
文档	官方偏架构，上手靠社区	官方文档完善 + 教程丰富	
IDE/工具链集成	起步	深度集成	

dsh 的定位：不是"开箱即用的整车"，是"可编程的乐高底座"。选择 dsh = 选择自由度，代价是自己要动手拼。

12.7 官方策略风险

作为选型者，必须考虑的黑天鹅：

- **开源承诺**：MIT 开源目前无杂音，但"官方级插件体系"意味着生态深度绑定官方节奏
- **模型绑定倾向**：官方默认体验倾向 DeepSeek 模型（通过 llm 插件可接任意模型，但"官方优化"以自家模型为主）
- **商业化转向**：一旦 dsh 用户量上来，官方可能推出托管服务/付费层——开源版能否保持同等能力未知
- **方向漂移**：rc 期功能增删频繁（TUI/CLI 路线之争即信号），押注错误方向 = 沉没成本

12.8 白皮书自身的局限

最后，诚实说这本白皮书自己：

- **基于 rc.6 编写**：所有命令、配置、数据在 rc.6 验证，**新版本可能全部失效**
- **benchmark 样本有限**：单机、单模型、有限任务集，不是权威评测
- **中文优先**：英文版是翻译/精简版，可能滞后于中文版内容
- **覆盖不全面**：dsh 官方包 60+，本手册深挖的是核心路径，长尾插件未逐一覆盖

  ***使用建议**：把本白皮书当"rc.6 快照 + 方法论"，不要当"永恒圣经"。版本更新时，先跑一遍 [roadmap 学习路径](./roadmap.md) 与 [速查卡](./cheatsheet.md) 的验收标准，再决定要不要更新你的依赖线。*

本章信息截至 2026-08-14（dsh 0.1.0-rc.6）。后续版本如有更新，欢迎提 Issue/PR 修正。

第 13 章：安全与沙箱模型

*本章目标：看懂 dsh 的安全骨架——沙箱管什么、权限怎么分级、审批怎么放行——以及社区审计发现的真实边界。**这是从"能跑"到"敢上生产"的关键一章。**

TL;DR（本章核心，30 秒版）

1. **三层防线**：沙箱（在哪执行）→ 权限预设（能做什么）→ 审批（危险动作要不要人点头）——默认不是裸执行
2. **沙箱只管文件效果**：`read-only` / `workspace-write` / `danger-full-access` 三档；网络与进程可见性不在词汇表内（所以 `taskkill` 杀宿主等案例是"词汇表外"，不是沙箱失守）
3. **权限 = 两个旋钮捆成预设**：`sandbox/mode` + `approval/policy`；默认表两张：`workspace-write`（工作区写 + ask）与 `danger-full-access`（无沙箱 + never）
4. **审批 fail-closed**：只有 `allowed-once` 放行；ask 无应答 = 拒绝（unavailable），never 一切确定性拒绝——headless 的默认姿态
5. **社区审计找到真实边界**：#159 fs-sandbox 竞态、#584 scrubbedParentEnv 误伤、#381 iframe 点击劫持、#817 审计报告（vm 逃逸 + 本地 RPC 无认证）
6. **企业基线一句话**：默认 `workspace-write`+ask、loopback 不放行、插件只装信任源、敏感目录不进 workspace

<details><summary>本章导航</summary>

</details>

13.1 安全设计哲学：沙箱/权限/审批三层

dsh 的安全骨架是三层各管一件事（依据官方 docs/subsystems/ 架构文档，已核验：sandbox.md / permission-presets.md / approval.md；与第 8 章 8.5 实测交叉印证）：

层	管什么	官方包	一句话	
沙箱	在哪执行、能碰哪些文件	sandbox/ sandbox、sandbox/ sandbox-local、	隔离命令执行的文件效果	

		sandbox/ sandbox-policy		
权限	能做什么	interaction/ permission-presets	把沙箱 + 审批捆成命名预设	
审批	危险动作要不要人确认	interaction/user-approval	最小权限的最后一道闸	

最小权限原则是三层共同的设计取向：默认预设只给"工作区写 + 关键操作询问"；任何提升（切 danger-full-access、改审批策略）都显式发生并写入会话日志——permission/preset 事件是纯日志的用户意图，不进模型转录，便于事后回溯"谁在什么时候提了权"。

威胁模型速记：三层防线防的是"不可信输入驱动代理越界"——提示词投毒、恶意插件、被诱导的高权限操作（#381 iframe 劫持就是往这条链上打）；它不承诺防"信任代码自身"（插件就是可执行代码，见 13.5），也不承诺防网络/进程级攻击。

> *核心认知：**dsh 的工具执行默认有隔离与审批层，不是裸执行**；但它不是"安全操作系统"——沙箱不挡网络与进程、审批可以被模型自答（13.6）。*

13.2 沙箱机制原理：fs 边界与工具沙箱

官方沙箱文档明确定义：沙箱只管文件效果（filesystem effects），网络与进程可见性不在其词汇表内。

三档模式（SandboxMode）：

模式	允许的文件效果	说明	
read-only	只读 + 必需 sink（POSIX 下如 /dev/null）	Windows ACL 后端无显式可写根，报 partial	
workspace-write	工作区根 + 后端承诺的临时区可写	默认预设；根来自会话不可变 cwd	
danger-full-access	不设限	直接 spawn 原 argv，不走 ctx.sandbox	

执行细节：

- **策略按调用携带**（per-call）：`SandboxExecutionPolicy`（mode + workspaceRoot + sessionId）在每次能力调用时解析，不绑死在 provider 上——同一 provider 可同时服务 `read-only` 的 bash 与需要可写状态目录的子代理
- **工作区根规范化**：先按文件系统语义 canonicalize（`symlink/..` 解析到真实目录）再做词法归一——防符号链接绕边界
- **后端矩阵**：Linux bwrap/Landlock、macOS Seatbelt、Windows ACL restricted-token；`enforcement: full | partial`——旧 Landlock ABI 与 Windows ACL 的

Everyone/hard-link 边界是当前 partial 案例（**承诺打折扣时，消费方须按 partial 处置**）

- **fs 工具同边界**：`write`/`edit` 等文件工具同样受 workspace-write 约束，不是"文件工具直通、只有 bash 才沙箱"（#149 递归删工作区即发生在 workspace-write 下）
- **只限写不限读**：当前权限模型约束的是**写**，工作区外内容可被读到（#492 提议评测隔离模式时点出的现状）

工具沙箱：bash/pwsh 走 bash-sandbox / pwsh-sandbox（消费 ctx.sandbox）在沙箱内起子进程；Windows 上 pwsh 沙箱有 ACL 约束（第 8 章实测遇到过 temp 权限问题；#758 还报告 Windows 沙箱临时目录清理后永久崩溃）。

同族案例：#159 的竞态属于 TOCTOU（检查与使用之间路径被换）；#278 是另一种拓宽手法——/tmp 作 workspace 时，受限子进程可通过 rebind 把 workspace-write 授权范围扩宽。两者共同教训：边界根目录要选可信位置，且不能只信任单一路径检查。

13.3 权限模型：工具权限分级表

官方权限预设把两个旋钮——sandbox/mode 与 approval/policy——捆成命名预设，客户端展示为一个"Permissions"选择器。默认表两张：

预设	sandbox/mode	approval/policy	语义	
workspace-write	workspace-write	ask	工作区内可写、操作前询问（默认）	
danger-full-access	danger-full-access	never	无沙箱、不询问	
custom（派生态）	任意组合	任意	手动调旋钮后的"非预设"态，不可作为切换目标	

工具权限分级（按危害半径排序；各工具"建议预设"为按官方文档推导，非逐工具实测——推断，待实测）：

级别	代表工具	建议预设	风险	备注	
只读探查	read/grep/glob	read-only 兼容	低	只读不落盘；读不限于工作区	
工作区写	write/edit/strreplaceeditor	workspace-write	中	写边界 = 工作区根 + 临时区	
命令执行	bash/pwsh	workspace-write（沙箱内）	高	沙箱不挡网络与进程	
全访问	任意（切 danger-full-	danger-full-access	极高	关闭所有闸门	

	access）				

*官方文档明确沙箱**只管文件**——所以"命令执行"级的高风险不能靠沙箱消除，要靠审批与信任边界（13.5 / 13.7）兜底。*

真实边界案例（详见 13.6）：workspace-write 下递归删除整个工作区零确认（#149）；danger-full-access 误删整个家目录（#461，真实事故）；Windows 上 minimal preset 允许工作区外写入且无审批（#523）；沙箱内 agent 可 taskkill 杀宿主（#466，进程可见性在沙箱词汇表外）。

13.4 审批流：从请求到放行

官方审批文档：审批回答一个问题——"这个具体动作能不能做？"

结果集 fail-closed（ApprovalOutcome）：

结果	含义	对调用方的效果	
allowed-once	一次性放行（只放行被问的这一个动作）	✅ 继续	
rejected	明确拒绝	❌ 中止	
cancelled	请求被撤回（AbortSignal）	❌ 中止	
unavailable	无应答者 / 应答者异常	❌ 中止（默认 fail-closed）	

策略（ApprovalPolicy）：ask（默认，交给应答者链；链为空 → unavailable）；never（确定性全部 rejected，不派发任何应答者——headless/CI 的严格姿态）。

流程：工具调用 → approval/request waterfall（插件可拦截/改写）→ 应答者（Web UI 人肉应答，或 ACP 桥的一次性机器决策）→ 每次请求成对写审计事件 approval/asked + approval/decided（同一 ApprovalRequestId）。

请求内容（ApprovalRequest）：agent（应答者只回答自己拥有的 agent）+ toolName + callId（关联已流式展示的工具调用，参数不重复渲染）+ reason（为什么问）+ signal（AbortSignal 撤回）。

三个已知坑：

1. **审批回环**：#250 真实复现"模型自批准 danger-full-access"（Web approval 通道可被模型自己驱动）——**审批不是安全边界**，只是人机协作闸门
2. **并发误定向**：#453 并发沙箱审批时点 Allow 会中止另一个调用（UI 应答与 callId 错配）
3. **重连丢审批**：#646 重连静默丢 pending approval（resync 清空 + replay 竞态）；headless 下审批行为未定义（#291）

13.5 插件安全审计清单

插件是 dsh 的能力边界（"一切皆插件"，见第 4 章）——装插件 = 装可执行代码。社区审计（#817 及同族）沉淀的检查清单：

#	检查项	社区依据	处置	
1	插件来源可信？dsh plugin add 无签名/来源校验	#587	只装官方/知名源；装前读源码	
2	boot 期权限最小？插件 boot 期有全配置树写权限	#587	警惕"装完自动改配置"的插件	
3	不拿 vm 当安全边界？workflow/动态插件 vm 可逃逸	#243 #451 #774 #778	vm 只当隔离引擎，不当安全层	
4	子进程环境 scrubbed 正确？scrubbedParentEnv 子串误伤合法变量	#584	检查 KEYBOARD/MONKEY 等含 KEY 子串的变量是否被误 scrub（误伤机制按标题推断）	
5	密钥脱敏不 fail-open？settings role'secret' 存在脱敏缺口	#226	密钥不写进提示词；校验脱敏逻辑	
6	安全 hook 不静默失效？timeout: 0 会 fail-open	#460 #583	hook 超时设正数；加载失败要报错	
7	审计留痕？approval/asked/decided 成对可查	官方 approval.md	关键操作按会话可回溯	
8	不泄漏会话明文？崩溃后 .tmp 明文会话残留	#674	敏感环境禁用崩溃转储 / 定期清理	

13.6 社区已知安全案例

以下 4 帖全部经 gh api graphql 核验存在（deepseek-ai/deepseek-harness Discussions，2026-08-14）：

帖号	一句话	类型	影响	
#159https://	fs-sandbox post-	沙箱竞态	TOCTOU：检查与	

github.com/deepseek-ai/deepseek-harness/discussions/159	check pathname race 可绕过 workspace-write 文件边界		使用之间路径被换，可越界写	
#584https://github.com/deepseek-ai/deepseek-harness/discussions/584	scrubbedParentEnv 子串误伤 KEYBOARD/MONKEY 等合法环境变量（附可 cherry-pick 修复）	子进程环境	合法环境变量被 scrubbed，行为异常	
#381https://github.com/deepseek-ai/deepseek-harness/discussions/381	默认 localhost Web 可被跨站 iframe 点击劫持，诱导授权 Full access 并驱动高权限操作	前端点击劫持	恶意网页诱导用户在 dsh UI 上点出高权限	
#817https://github.com/deepseek-ai/deepseek-harness/discussions/817	安全审计报告：沙箱/审批边界绕过与本地 RPC 无认证（PoC 可私下提供）	系统性审计	覆盖 vm 逃逸（同族 #243 #778）、本地 /api 无鉴权（同族 #451，CVSS 8.8）、approval 回环（#250）	

同族高价值补充（帖号来自仓库内 docs/research/discussion-mining.md §2.6，该报告称全量 780 帖程序化核验）：

帖号	**一句话**	**启示**	
#149	workspace-write 下可递归删除整个工作区，零确认	工作区内容也要防"自毁"	
#250	Web approval 回环：模型自批准 danger-full-access	审批不是安全边界	
#461	Full Access 模式误删整个家目录（真实事故）	高权限预设风险实锤	
#587	第三方插件 boot 期有全配置树写权限	插件信任 = 供应链安全	
#466	沙箱内 agent 可 taskkill 杀宿主	进程可见性在沙箱词汇外	
#674	崩溃后 .tmp 明文会话残	隐私风险	

	留不清理		

免责：以上为 rc.6 时代（2026-08-13 发布后 48h 内）的社区报告，部分可能已在新 rc 修复；引用时注意版本语境。

13.7 企业安全基线

给"要上生产 / 团队推广"的决策建议（对齐第 6 章风格）：

维度	基线	依据	
网络暴露	保持 loopback（--host 0.0.0.0 被官方拒绝是有意的）；远程认证完善前勿绕过	#76 #130 #397	
权限预设	默认 workspace-write + ask；danger-full-access 只在隔离机 / 一次性任务用	官方 preset 表 + #461	
审批策略	人肉应答链路必备；headless 用 never（确定性拒绝）并前置 review 闸	approval.md + #291	
插件治理	白名单源 + 装前过 13.5 清单；禁止裸 dsh plugin add github:	#587 #656	
工作区边界	敏感目录（家目录/密钥目录）不进 workspace；工作区内容定期提交版本控制	#149 #461	
审计	会话日志留存（approval 审计对可回溯）；防 .tmp 明文残留	approval.md + #674	
升级纪律	rc 阶段每次升级重跑 13.5 清单（沙箱后端/预设表可能变）	第 12 章 rc 版诚信	

决策速记：本地个人开发 → 默认预设即可；团队共享 → workspace-write + ask + 插件白名单；CI/无人值守 → headless + never + 前置代码 review；任何场景都别为省事长期挂 danger-full-access。

场景 → 推荐组合（决策建议，按上表基线推导）：

场景	推荐组合	理由	
本地个人开发	默认 workspace-write +	开箱即用，审批不烦人	

	ask		
团队共享主机	workspace-write + ask + 插件白名单	多用户共用，插件供应链风险放大	
CI / 无人值守	headless + never + 前置 review 闸	确定性拒绝，无人挂起	
隔离机 / 一次性任务	danger-full-access（机器本身隔离）	授权最大化但攻击面控制在单机	

动手练习（检验你是否真懂了）

1. **理解题**：沙箱三档模式分别允许什么文件效果？为什么说"网络与进程不在沙箱词汇表内"？

> 自查：参考本章 13.2 节三档表与"后端矩阵"段

2. **理解题**：为什么 headless 下的 `never` 是"最严格"而非"最宽松"？

> 自查：参考本章 13.4 节"策略"段（deterministic rejected）

3. **动手题**：打开 `dsh web`，观察 Permissions 选择器的两个预设；切到 `danger-full-access` 再切回，看会话日志里的 `permission/preset` 事件

> 自查：参考本章 13.3 节预设表

4. **动手题**：装一个社区插件前，按 13.5 清单逐条过一遍（来源/源码/权限/审计），结论写进笔记

> 自查：参考本章 13.5 节清单表

5. **思考题**：为什么 #250 的"模型自批准"说明审批不是安全边界？如果审批挡不住，企业靠什么兜底？

> 自查：参考本章 13.6 节 + 13.7 节基线表

常见疑问 FAQ

Q1：沙箱能防住恶意代码吗？

不能当安全边界用。沙箱只管文件效果；网络（如 SSRF）与进程（如 #466 taskkill）在其词汇表外。防恶意代码靠插件信任（13.5）与网络/进程级隔离（容器/VM/远程沙箱——官方文档明确这些是能力缝的 sibling 实现，不是 ctx.sandbox 的 provider）。

Q2：为什么 headless 用 never 反而更安全？

never 不是"永不询问"的宽松模式，而是确定性拒绝：任何审批请求直接 rejected，不派发应答者——无人值守时不会出现"挂着等人点 Allow"。需要放行的动作靠前置 review 或显式配置，而不是运行时询问。

Q3：workspace-write 是不是就安全了？

边界内相对安全，但 #149 显示"递归删除整个工作区"在 workspace-write 下零确认——沙箱防的是"越界"，不防"边界内自毁"。重要代码请纳入版本控制。

Q4：第 12 章已提过 #159，这章为什么又讲？

第 12 章是"已知不足速查表"（一句话 + 状态），本章展开机制（post-check 竞态怎么发生）、同族（#278 /tmp rebind）与缓解（不信任单一路径检查、关键目录不进 workspace）。

Q5：这些安全帖号可信吗？不会是编的吧？

4 个核心帖号（#159 #584 #381 #817）均经 gh api graphql 逐号核验存在（链接见 13.6 表格）；同族补充帖来自仓库内 docs/research/discussion-mining.md（全量分页抓取 780 帖并程序化核验）。

下一章：[第 14 章：成本与用量](./14-cost.md)（规划中）。

第 14 章：缓存与成本工程

本章目标：把"便宜"从玄学变成工程——对话缓存机制、命中率实测与优化、成本模型、推理档位联动、真实任务预算，以及怎么"看见"每笔 token 花在哪。

TL;DR（本章核心，30 秒版）

1. **缓存是成本第一变量**：DeepSeek context cache 对重复输入按 **98% 折扣（Flash）/ 99%+ 折扣（Pro）** 计费——Agent 工作负载输入占比高，命中率直接决定成本量级
2. **手册实测命中率 97%**：会话延续 + 稳定工具 schema + Agent 多步工作负载特性（第 5 章 5.1.1）；社区长跑实测可达 99.7%（[#560](https://github.com/deepseek-ai/deepseek-harness/discussions/560)）
3. **命中率三原则**：会话延续（别频繁新建）、前缀稳定（别老改配置）、少打断（一次写清验收标准）
4. **两个独立杠杆**：推理档位管"思考 token 量"，缓存管"输入单价"，可叠加——`low` 档 + 高命中 = 最便宜组合
5. **成本要看得见**：会话统计行看缓存命中 %；每轮 token 显示官方待实现（[#735](https://github.com/deepseek-ai/deepseek-harness/discussions/735)），先用统计行 + 社区 cost-tracker 插件

<details><summary>本章导航</summary>

</details>

14.1 对话缓存机制原理

DeepSeek API 的 context cache（提示词缓存）：重复/相似的输入 token 按缓存价计费，而不是全价。dsh 每次模型请求的输入 = 系统提示 + 技能目录 + 对话历史 + 工具 schema（第 8 章 8.3），其中不变的前缀部分就是缓存的天然候选——这决定了 Agent 工作负载的缓存潜力远高于普通聊天。

计费规则（第 5 章 5.1.1 定价表）：

计费项	未命中价	缓存命中价	折扣	
输入（Flash）	$0.14/M	$0.0028/M	98%	
输入（Pro）	$0.435/M	$0.003625/M	99%+	

每次请求的输入结构（第 8 章 8.3），决定了哪些 token 能命中：

```
系统提示（稳定） + 技能目录（稳定） + 对话历史（增量） + 工具 schema（稳定） + 本轮消息（增量）
└──────────────── 缓存命中区（重复） ──────────────────┘   └── 未命中区（新增量） ──┘
```

命中区越大、增量越小 → 命中率越高。理解这个结构，14.3 的三个优化杠杆本质上都是"让命中区变大"。

缓存不只省钱，还提速：冷启动首轮含上下文注入约 110s，热缓存约 1s（第 1 章实测）——命中时首 token 时间量级性缩短。

> ***严格前缀缓存规则（社区分析，#1052 weijiafu14，官方文档 [kv_cache](https://api-docs.deepseek.com/guides/kv_cache/)）**：DeepSeek 是**严格的前缀缓存**——前缀单元在**用户输入结束、模型输出结束等位置持久化**；下一次请求只能命中"已经完整匹配"的前缀。*
> *关键推论：*
> *- **工具结果天然 miss 一次**：工具结果是在"上一轮模型输出"之后才追加的，它**第一次进入下一次请求时必然是新后缀、必 miss 一次**——不论传原文还是压缩预览（这与截断方式无关）。*
> *- **"截断会避开缓存"是误读**：只要截断是确定性的（同输入同输出），后续回放的是同一份字节，已有前缀照常命中；只有首次出现的压缩结果、以及之后主动读取 artifact 新增的内容各自 miss 一次。**压缩工具不会绕开 DeepSeek 缓存，反而把"必然首次 miss"的工具结果缩小**（#1052 weijiafu14 实测：原生工具输出 42,299 字符 → 压缩持久化 7,885 字符，后续 derive/replay 哈希一致）。*
> *- **代价提醒（#1052 fnsii + weijiafu14）**：缓存本身是 best-effort、官方不承诺 100% 命中（weijiafu14 转述官方说明）；若完全不保留上下文前缀（如每轮重写/清空历史、换新会话丢全部前缀），则完全无法命中缓存，等于按新输入全价计算（fnsii）。**快封顶时优先做 summary 压缩再续**（保留前缀摘要），而不是直接开新会话丢全部前缀（fnsii 建议）。*

14.2 命中率实测：97% 怎么来的

手册在真实 dsh 会话统计中实测缓存命中率可达 97%（第 5 章 5.1.1）。为什么 dsh 的命中率特别高：

原因	说明	
会话延续	同会话多轮对话，系统提示/技能目录/历史重复 → 命中	
稳定的工具 schema	每步请求携带相同的工具定义 → 命中	
Agent 工作负载特性	多步工具链反复携带相同上下文 → 命中率天然高	

边界（诚实版）：97% 依赖"对话模式重复"——全新任务/冷启动场景会明显下降（第 12 章 12.5）。社区长跑实测可达 99.7%（[#560](https://github.com/deepseek-ai/deepseek-harness/discussions/560)），印证"越长的会话命中率越高"。

实测方法：Web UI 底部会话统计行看"缓存命中 %"（第 6 章 6.6）；对比基准是 5.1.1 的"50 步工具链任务、2.4M 输入 token"——它是输入量级的典型样本。

> ***口径注意（#1234 社区补充）**：Harness 界面显示的「缓存命中 %」是**对话内**口径——只统计当前会话这一路请求。若同一时间段开过多个对话，其他对话命中率较低时，会把**多对话整体/后台口径**拉低（社区实测对照：后台/整体约 94.5% vs 界面（对话内）98%，[#1234](https://github.com/deepseek-ai/deepseek-harness/discussions/1234)）。**测量命中率时先明确口径**：对话内（界面统计行）还是多对话整体（后台/API 侧）；对比命中率只在同口径下才有意义。*

指标	看哪里	命中时的表现	
缓存命中 %	会话统计行（Web UI 底部）	稳定在高位（长会话 97% +）	
首 token	每轮结束行	从 10s+ 量级性降到 1s 以下（第 6 章 FAQ Q4）	
总 token	会话总览	增量小 = 命中区在变大（14.1 结构）	

14.3 命中率优化实践

三个杠杆，按性价比排序：

杠杆	做法	原理	
会话延续	长任务保持会话延续，避免频繁新建会话；批量任务放同一会话/同前缀	历史消息重复 → 命中	
稳定提示词	系统提示/技能目录等前缀保持稳定，不频繁改配置；工具 schema 不变	前缀一致 → 命中	
少打断	一次把验收标准写清楚（第 5 章 5.7 / 第 10 章案例启示），减少"重开会话重讲上下文"	避免前缀重建 → 全价重来	

落地检查：命中率低于预期时，第一反应查"前缀是否变了"——改过配置、换过档位、动过技能目录，都会让缓存失效（第 6 章 6.6 监控行）。

14.4 成本模型：命中率如何决定账单

成本模型速记（第 6 章 6.6）：总成本 ≈ 输出 token×输出价 + 输入未命中×未命中价 + 输入命中×命中价——Agent 工作负载输入占比高，缓存命中率是成本的第一变量。

用第 5 章 Flash 输入价算一笔账（10k 输入 token，命中率对比）。算术示例（90% 命中）：9,000 × $0.0028/M + 1,000 × $0.14/M ≈ $0.000165——先把命中/未命中拆开，再分别乘各自单价：

命中率	命中 token	未命中 token	输入成本（Flash）	相对成本	
90%	9,000	1,000	≈ $0.000165	1×	
50%	5,000	5,000	≈ $0.000714	≈ 4.3×	
10%	1,000	9,000	≈ $0.001263	≈ 7.6×	

放大到 1M 输入 token：90% 命中 ≈ $0.0165，10% 命中 ≈ $0.126，全价 $0.14——差近一个数量级。

再放大到手册实测量级（5.1.1：50 步工具链、2.4M 输入 token）：97% 命中 ≈ $0.017，全价 ≈ $0.34，约 20×。这就是 5.1.1"成本远低于按未命中价计算的预期"的数字来源。

> *注：本表只算输入侧——第 5 章定价表只收录输入价，输出侧请按官方定价页补充。输入占比高的 Agent 场景下，输入侧就是主变量。*

14.5 推理档位联动：low/high/max 成本矩阵

推理档位管"思考量"，与缓存是两个正交杠杆：档位影响 token 总量，缓存影响输入单价，两者相乘。思考 token 减少 → 总 token 下降（第 6 章 6.6），且随上下文重复的部分同样吃缓存折扣（推断，待实测）。

档位	思考 token 量	相对成本	适用场景（第 6 章 6.2）	成本策略	
low	最少	最低	简单/确定性轮次：文件操作、批量、工具链中的廉价步	批量任务的默认档	
high	中等	中等	日常 Agent 任务（默认）	质量与成本的默认平衡	
max	最多	最高	复杂推理、长链规划、debug	单次复杂任务可用，别开进批量循环	

联动要点：

- **长工具链任务（20-50 步）收益最大**：每步思考降档的累计效果显著（第 6 章 FAQ Q2）——档位 × 命中率两个杠杆同时作用于每一步
- 简单轮次用 `low` 档质量几乎无差别；复杂推理用 `low` 可能漏关键步骤（第 6 章 FAQ Q3）
- **叠加示例**（2.4M 输入量级任务，Flash 价）：`low` + 97% 命中 ≈ $0.017 以下（思考量更低，实际更少，推断，待实测）；`max` + 反复冷启动 ≈ $0.34 以上——**同一任务，两个杠杆全开与全关差 20×+**
- 注：`low/high/max` 为本手册实测网关（pi-ai/opencode-go）档位；默认 deepseek-official 适配器为 `off/high/max`（第 6 章 6.2 注释）

14.6 预算实战：案例 A 成本估算

以第 10 章案例 A（数据质量分析，186 秒）为样本做预算：工具链 read → write(clean.py) → bash → write(visualize.py) → bash → read → 总结，约 6-8 步。按 5.1.1 量级折算（50 步 ≈ 2.4M 输入 token），案例 A 输入量级约 50 万 token（推断，待实测）：

场景	命中率	输入成本（Flash，估算）	
会话延续 + 前缀稳定（推荐做法）	97%	≈ $0.0035	
中途新开会话/前缀变动	10%	≈ $0.063	
极端：每次全价	0%	$0.07	

步骤级拆解（为什么会话延续省钱）——只有首轮是冷启动全价，后续每轮只有"本轮消息 + 工具结果"这一小段新增量未命中（推断，待实测）：

步骤	内容	输入 token（推断，待实测）	命中状态	
1	readsalesdata.json	~5 万	冷启动（未命中）	
2-6	write/bash/ write/bash/read/ 总结	~45 万	前缀命中区，增量极小	

结论：一次 3 分钟的真实任务，输入侧成本不到 1 美分；同样的活，做法不同（会话延续 vs 反复冷启动）成本差 约 18×。输出侧（两个脚本 + 总结）量级远小于输入，总成本仍为美分级（推断，待实测）。这就是"dsh 配 V4-Flash 成本约为 Claude 的 1/10~1/30"（第 1 章实测）在单任务维度的来源。

14.7 测量方法：让 token 消耗可见

现状（有的）：

- Web UI 底部会话统计行：`N 轮 · M 步 | LLM Xs · 工具调用 Ys | 首 token 平均 ...`（第 6 章 6.3）
- 每轮结束行：`10:14 · 用时 9 分 34 秒 · 首 token 1.7 秒 · 79 tok/s`（[#735](https://github.com/deepseek-ai/deepseek-harness/discussions/735) 原帖截图）
- 会话总览有总 token 统计；统计行可见"缓存命中 %"（第 6 章 6.6）——**对话内口径**，多对话整体命中率会被其他低命中对话拉低（[#1234](https://github.com/deepseek-ai/deepseek-harness/discussions/1234) 社区补充，见 14.2 口径注意）

缺口（没有的）：每轮 token 数。官方讨论区 [#735](https://github.com/deepseek-ai/deepseek-harness/discussions/735)（2026-08-14 提出，标题「【友好显示】希望在每轮对话中加入本轮对话 token 消耗量」）正是这个需求——已在帖子中确认存在，官方尚未实现。

另一个缺口（#735 评论区 Jianye）：希望在每轮显示中带 provider 与 model id 标识（原帖截图里 xtoken：gpt-5.6-sol 这类"provider + modelid"信息用户希望界面直接给出）——同一需求帖下的补充诉求。

社区建议（已同步到 #735 评论区）：每轮显示应拆成两个数——① 本轮总 token（粗看消耗）；② 本轮缓存命中/未命中 token（看成本优化空间）。只看总 token 会误判：同样 10k token，命中率 90% 和 10% 成本差 7.6×（见 14.4）。

过渡期工具：社区 dsh-plugin-cost-tracker（第 9 章 9.5/9.6，实时 token 成本追踪，插件/MCP 形态）——实现层面走第 4 章的插件扩展点，在请求/会话事件上统计 usage（推断，待实测）；或按会话日志/API usage 字段手工核算。第 11 章预测"缓存命中率工具化"是中期主线（11.2）——#735 落地后成本透明化闭环。

更完整的社区实测工具：dsh-usage（作者 kestiany，[#1169](https://github.com/deepseek-ai/deepseek-harness/discussions/1169) 收录授权）。npm 已发布 0.1.0（dsh plugin --profile web add dsh-usage 即可安装），基于 Harness 持久化会话日志计算、无需独立统计数据库。核心能力：

- **每轮对话结束位置固定展示**：`Total · Input · Cache · Output · Cost`——正好补齐 #735 缺的"每轮 token 数"，且区分输入/缓存命中/输出
- **Settings → Usage 完整用量页**：总 token、输入、缓存命中、输出、调用次数及**预估费用**（需为 Provider/模型配置价格；缺 usage/价格时费用显示 `--`，对话底部省略 Cost，避免把不完整费用当总费用）
- **52 周用量热力图**（GitHub Contributions 风格）+ 按模型/会话/单轮查看
- 支持 DeepSeek Harness 中的其他模型和 Provider；`Cost` 为估算值，不代表供应商最终账单

记忆/压缩生态工具（#1052 长会话降本专题评论区，2026-08-14）——针对"长会话 token 暴涨"（原帖实测 70M→100M），社区给出两个互补方案：

工具	作者	定位	机制	
dsh-sgme（freehul/sgmehttps://github.com/freehul/sgme，npm 包 dsh-sgme）	freehul	记忆引擎（治本）	「对话历史」与「长期记忆」分层：每轮零成本落盘（L0，不走 LLM）→ 会话结束提炼去重合并（L1/L1.5/L2，提炼前先剪枝剔除无用工具调用/中间产物，实测省 65%～96% 会话内容）→ 新会话按场景只注入相关记忆块（纯结构化查询，不烧 token）→ 需要细节时 memorysearch 按需检索	
pi-quiet-tools（经 pi2dshhttps://github.com/weijiafu14/pi2dsh 挂载）	weijiafu14 提供（freehul 亦推荐）	工具输出压缩（治标）	在进入模型上下文前压缩大工具结果：默认超过 12,000 字符或 240 行时只把确定性的头尾预览交给模型，完整结果存本地 artifact、需要时模型再读；阈值可用 QUIETTOOLSMAXCHARS / QUIETTOOLSMAXLINES 调整。对 MCP/Playwright 大返回体有效（作者实测 40,799 字符闭环压缩）。与缓存兼容：见 14.1 严格前缀缓存规则——不绕开缓存，反而缩小"必然首次 miss"的工具结果	

安装示例（pi-quiet-tools，摘自 #1052 作者实测）：

```
npm i -g pi2dsh pi2dsh host --packages pi-quiet-tools@0.2.0 --out ./dsh-pi-quiet dsh plugin --profile web add file:$PWD/dsh-pi-quiet
```

两者不冲突：pi-quiet-tools 解决"大工具结果反复进上下文"，dsh-sgme 解决"长会话越用越贵/延续性"——前者立即生效，后者适合需要跨会话迁移记忆的场景（#1052 评论区共识：把"对话历史"和"长期记忆"分开，历史该滚就滚、记忆按需注入）。

14.8 成本相关踩坑清单

#	坑	现象	解法	
1	缓存命中误判成"模型变快"	A/B 对比被 1s vs 110s 误导（第 6 章坑 #6）	对比测试用全新 prompt	
2	频繁新建会话烧钱	每个任务都冷启动，输入全价	长任务/批量保持会话延续	
3	改配置悄悄杀掉命中率	命中 % 突然下降，成本上涨	监控统计行命中 %，前缀稳定	
4	只看总 token 误判成本	10k token 90% vs 10% 差 7.6×	拆命中/未命中看（#735 建议）	
5	预算漏了输出侧	只按输入价算，账不对	按官方定价页补输出价（第 5 章只收录输入价）	
6	档位乱开	批量任务开 max，成本翻倍	批量用 low，复杂任务才 max	

14.9 决策建议速查

场景	建议	
长工具链任务（20-50 步）	high + 会话延续 + 前缀稳定；2.4M 输入量级 ≈ $0.017（97% 命中）	
批量/简单轮次	low + 同会话批量（命中红利最大）	
复杂推理/debug	max（成本换质量），单次任务可接受冷启动	
成本敏感批处理	low + headless + 单会话串行	
性能/成本对比测试	全新 prompt + 固定档位 + 多次取中位数（第 6 章 6.4/6.6）	
命中率异常下降	先查前缀是否变动（配置/技能目录/模型档位）	

动手练习（检验你是否真懂了）

1. **理解题**：为什么说"缓存命中率是成本的第一变量"？用 14.4 的 10k token 表格解释 90% 与 10% 命中差多少倍

> 自查：参考 14.4 对比表（7.6×）与 6.6 成本模型速记

2. **动手题**：跑一个 3 步任务，观察 Web UI 底部的会话统计行和每轮结束行（时间/首 token/tok/s），判断这次任务大概命中了多少缓存

> 自查：参考 14.7 测量方法与 6.3 统计行格式

3. **动手题**：把同一批任务分别用"每项新开会话"和"同会话连续跑"两种方式执行，对比耗时差异——新开会话大概率首轮明显变慢（冷启动）

> 自查：参考 14.1 冷启动 110s vs 热缓存 1s 的实测

4. **思考题**：为什么"只看本轮总 token"会误判成本？#735 评论区建议把每轮显示拆成哪两个数？

> 自查：参考 14.7 节"社区建议"段落

5. **动手题**：按 #735 的建议口径，把一个会话的总 token 手动拆成"命中/未命中"两部分，用 14.4 的单价分别核算，验证"差一个数量级"的说法

> 自查：参考 14.4 对比表与 14.2 测量方法表

常见疑问 FAQ

Q1：缓存命中率最高能到多少？

手册实测 97%（第 5 章 5.1.1）；社区长跑实测可达 99.7%（[#560](https://github.com/deepseek-ai/deepseek-harness/discussions/560)）。但全新任务/冷启动会明显下降（第 12 章 12.5）——命中率是"做法"的函数，不是模型的常数。

Q2：low 档会明显降低质量吗？

简单确定性任务（文件操作、批量处理）几乎无差别；复杂推理、长链规划、debug 用 low 可能漏掉关键步骤（第 6 章 FAQ Q3）。建议日常 high，批量/简单任务切 low。

Q3：官方什么时候提供每轮 token 显示？

[#735](https://github.com/deepseek-ai/deepseek-harness/discussions/735) 是 2026-08-14 提出的需求帖（每轮 token + provider/model 标识，见 14.7），尚未实现。过渡期用：会话统计行命中 % + 社区 dsh-plugin-cost-tracker（第 9 章）或 dsh-usage（14.7）。

Q4：第三方网关/自建 provider 也吃缓存折扣吗？

缓存折扣是 DeepSeek API 侧能力（第 5 章定价表）；第三方网关的缓存支持与计费需查对应定价页（推断，待实测）。

Q5：为什么缓存命中后任务会明显变快?

命中的前缀部分无需重新计算，首 token 时间量级性缩短（冷启动 ~110s vs 热缓存 ~1s，第 1 章实测）。判断方法：首 token 突然从 10s+ 降到 1s 以下，大概率命中（第 6 章 FAQ Q4）。做性能对比时用全新 prompt，避免缓存干扰（第 6 章坑 #6）。

本章信息截至 2026-08-14（dsh 0.1.0-rc.6）。定价与缓存策略以官方文档为准；标注"（推断，待实测）"处为推算或预估，欢迎实测后 PR 修正。

附录 B：官方包速查大全（@deepseek-ai/*）

***本清单为「已知包」清单**：收录白皮书正文实测/提及过、且能在 npm registry 上核实的官方包（`@deepseek-ai/dsh-*`与`@deepseek-ai/cordis`），包描述来自`npm view`实测结果。dsh 处于`0.1.0-rc`快速迭代期，**包名与描述随版本更新**——若与官方仓库`packages/AGENTS.md`或 npm 最新描述不一致，以官方为准。未能核实的包标注「（待补全）」，不虚构包名。*

***包数口径**：本清单收录经 npm 核实的 **33 个核心包**（CLI 直接依赖 53 个 + 家族发布总量 221 个，见 [08 章 8.1 节](./08-tools-context.md) 说明）；白皮书/官方所称「60+ 能力包」指仓库`packages/`下全部子目录，完整列表见官方仓库`packages/README.md`（47 组权威表）。*

包名与仓库布局对应关系：npm 包`@deepseek-ai/dsh-xxx`≈ 官方仓库`packages/<group>/xxx`（如`dsh-tool-fs`↔`fs/tool-fs`）。

验证方式：`npm view @deepseek-ai/<name> description`（2026-08 快照）。

能力域取值：工具（模型可调用的工具/执行管道）· 上下文（请求上下文组装与压缩）· 会话（会话持久化/标题/遥测）· 子代理（委派子任务）· MCP（外部工具服务器接入）· 工作流（多步编排）· 安全（沙箱/审批/循环卫生）· LLM（模型接入与策略）· UI（浏览器界面与客户端服务）· 核心（CLI/骨架 bundle，不属上述能力域）。

核心（Core）：CLI 与骨架 bundle

dsh 的"地基"：装一个 @deepseek-ai/dsh 就能跑，profile 由内置 bundle 栈堆出来（dsh-base → dsh-headless / dsh-web-app）。

包名	用途	能力域	
@deepseek-ai/dsh	dsh CLI：profile 启动、插件管理、浏览器 UI 别名（npx -y @deepseek-ai/dsh web）	核心	
@deepseek-ai/dsh-base	共享 dsh 核心 profile bundle：每个 profile 的第一层补丁，在空 profile 根上插入基础插件行	核心	
@deepseek-ai/dsh-headless	headless 一次性 bundle：无 Host/HTTP/浏览器层的直接 core	核心	

	Agent/Session 运行器		
@deepseek-ai/dsh-agent	Agent 接口、注册表、发起者作用域与事件词汇（插件开发常直接依赖）	核心	
@deepseek-ai/dsh-web-app	dsh 浏览器面 bundle：web patch 层（前端 dist 服务、web 面提示词、bash 运行时变量、URL 行）	核心	
@deepseek-ai/cordis	底层元框架：现代 JavaScript 应用的依赖注入/事件/生命周期容器（dsh 的插件底座）	核心	

工具（Tools）：模型可调用的能力

模型实际看到的工具名是简短动词（read/write/grep/glob/edit/bash/websearch/todowrite…），底层由本组包提供实现。

包名	用途	能力域	
@deepseek-ai/dsh-tools	工具注册表与执行管道（tool registry + execution pipeline）	工具	
@deepseek-ai/dsh-fs	抽象文件系统能力 seam（ctx.fs）：词汇类型、FileSystem 服务（文本 IO + 可选版本守卫原子写）、fs/ 策略事件词汇	工具	
@deepseek-ai/dsh-tool-fs	模型侧文件系统工具（read、write、edit），基于 ctx.fs	工具	
@deepseek-ai/dsh-tool-fs-search	模型侧文件发现工具（glob、grep），内置打包的 ripgrep 二进制	工具	
@deepseek-ai/dsh-tool-str-replace-editor	模型侧编辑工具：查看、创建、字面量替换、行插入（工具结果含 locations → 产物追踪）	工具	
@deepseek-ai/dsh-shell	抽象 bash 执行器 seam（ctx.shell）	工具	
@deepseek-ai/dsh-	模型侧 bash 工具，可选	工具	

tool-bash	通用后台任务与沙箱升级支持		
@deepseek-ai/dsh-web	抽象 web 访问能力 seam（ctx.web）：search/fetch 提供者注册表、与注册顺序无关的选择、请求/结果词汇、WebError 分类	工具	
@deepseek-ai/dsh-tool-web	模型侧 web 工具（websearch、webfetch），基于 ctx.web	工具	
@deepseek-ai/dsh-tool-todo	模型侧待办工具 todowrite，基于事件溯源会话日志	工具	
@deepseek-ai/dsh-tool-skill	模型侧技能加载工具（skill 调用入口）	工具	

上下文（Context）：上下文组装与压缩

"装什么"（系统提示 + 技能目录 + 历史 + 工具 schema）与"装不下时怎么压缩"。

包名	用途	能力域	
@deepseek-ai/dsh-system-prompt	系统提示组装注册表（官方插件经 systemPrompt.section 注册提示片段）	上下文	
@deepseek-ai/dsh-compaction	抽象压缩服务 seam（ctx.compaction）	上下文	
@deepseek-ai/dsh-compaction-basic	token 计量驱动的压缩策略 + LLM 摘要后端（检测溢出 → 修剪 → 摘要）	上下文	
@deepseek-ai/dsh-skill	Agent 技能提供者注册表（技能目录注入上下文，模型按需调用）	上下文	

会话（Session）：持久化、标题、遥测

包名	用途	能力域	
@deepseek-ai/dsh-session	事件溯源会话存储（event-sourced session store）	会话	

@deepseek-ai/dsh-session-title	基于日志的会话标题服务与提供者注册表	会话	
@deepseek-ai/dsh-session-telemetry	遥测 seam：会话事件捕获、投影、脱敏、交接给上报后端	会话	

子代理（Subagent）：委派子任务

包名	用途	能力域	
@deepseek-ai/dsh-subagent	抽象子代理 seam（ctx.subagents）：委派子代理的命名提供者注册表	子代理	

MCP：外部工具服务器接入

包名	用途	能力域	
@deepseek-ai/dsh-mcp-client	MCP 客户端桥：连接 MCP 服务器并把其工具注册到 ctx.tools	MCP	

工作流（Workflow）：多步确定性编排

包名	用途	能力域	
@deepseek-ai/dsh-workflow	工作流能力 seam：ctx.workflows 服务、run 词汇、workflow/ 事件	工作流	

安全（Safety）：沙箱、审批、循环卫生

包名	用途	能力域	
@deepseek-ai/dsh-sandbox	抽象进程沙箱 seam（ctx.sandbox）：同世界隔离词汇 + SandboxProvider 契约	安全	
guard/	循环卫生与工具超时（白皮书 8.5 提及；npm 上具体包名待补全）	安全	
interaction/	权限/审批（危险操作确认；npm 上具体包名待补全）	安全	

LLM：模型接入与策略

包名	用途	能力域	
@deepseek-ai/dsh-llm	Provider 无关的 LLM 服务接口	LLM	
@deepseek-ai/dsh-llm-deepseek	DeepSeek chat-completions 适配器（默认 provider=deepseek-official，接受 off/high/max 档位）	LLM	
@deepseek-ai/dsh-llm-retry	Provider 路由的 LLM 请求重试策略	LLM	

UI：浏览器界面与客户端服务

包名	用途	能力域	
@deepseek-ai/dsh-client-runtime	客户端核心服务：SlotsService、SessionsService（作用域树 + 对象层）	UI	
@deepseek-ai/dsh-client-locale	语言包插件：Host 端 zh/en 偏好、浏览器端回退、语言快照、类型化命名空间字典	UI	
ui-conversation / ui-tool 等	Web UI 各部件（白皮书 8.1 提及 client/；npm 上具体包名待补全）	UI	

已知名但 npm 未发布（历史 404 包）

以下包名仅存在于白皮书踩坑记录中，在 npm 上无法安装（rc.1 时代依赖断裂的根源），遇到 404 属正常：

包名	说明	
@deepseek-ai/dsh-pty	rc.1 时代依赖，从未发布（pnpm dlx 404 的根因，见第 2 章 FAQ）	
@deepseek-ai/dsh-type-meta	rc.1 时代依赖，从未发布（rc.1 依赖断裂的根因，见第 3 章 3.5 节）	

规避方式：依赖统一走 `^0.1.0-rc.6` 线。

相关章节：[第 8 章 · 工具与上下文系统](./08-tools-context.md)（60+ 能力包地图）· [第 9 章 · MCP 子代理与工作流](./09-mcp-subagent-workflow.md) · [术语表与命令速查](./appendix-glossary.md)

附录 A：术语表与命令速查

术语表

术语	一句话解释	
Harness	套在模型外的工程层：会话、工具、上下文、循环控制	
dsh	DeepSeek Harness 的命令行名（dsh 命令）	
Profile	一种可启动形态（web / headless / 自定义），= bundle 栈 + patch 层	
Bundle	一组插件的集合（官方：dsh-base、dsh-web-app、dsh-headless）	
Cordis	dsh 底层的插件容器（依赖注入、事件、生命周期）	
插件（Plugin）	cordis 插件；可同时携带 host 半（Node）与 client 半（浏览器）	
host 半 / client 半	插件在 Node 侧（工具/服务/事件）与浏览器侧（UI）的两副面孔	
扩展点	官方提供的钩子（agent/request、settings、conversationEvents、slots）	
agent/request waterfall	每步模型请求前可改配置的事件链（插件在此注入 reasoningEffort 等）	
reasoningeffort	思考强度（官方适配器 off/high/max；low 为实测网关档位）——速度与质量的权衡旋钮	
headless	一次性 CLI 任务模式（dsh --profile headless "任务"）	
compaction	长对话上下文压缩	
subagent	子代理（并行委派任务）	
MCP	模型上下文协议（接入外部工具服务器）	
workflow	多步确定性工作流编排	

locations	工具返回的文件路径（驱动产物追踪）	
extension point	官方钩子（agent/request 等），插件接入点	
waterfall	事件链：监听者可改配置传给下一个（agent/request 用此注入）	
context cache	DeepSeek 提示词缓存（重复输入按折扣价计费）	
缓存命中率	输入 token 走缓存价的比例（dsh 实测可到 97%）	
TUI	终端 UI（官方未内置，需插件）	
turn	对话的一轮（用户+助手+工具调用）	
step	turn 内的一个推理步骤	
guard	循环卫生/工具超时插件	
skill	技能（skill-catalog 注入上下文，模型按需调用）	
inject	cordis 依赖注入（插件声明所需服务）	
产物文件	模型创建/修改的文件（对话末尾的可打开 chips）	
sandbox	命令执行的隔离沙箱	
rc	预发布版本（0.1.0-rc.6），迭代快、可能有破坏性变更	

命令速查

dsh 核心

```
dsh web                                    # 启动 Web UI dsh --profile headless "任务"
# 一次性任务（脚本/CI） dsh --version                                    # 版本 dsh --dump-
config                              # 合成配置树 dsh plugin --profile <name> add <pkg>
# 安装插件
```

环境

```
node --version                             # 需 ≥ 22 npm install -g @deepseek-ai/dsh
# 全局安装 npx -y @deepseek-ai/dsh web                    # 免安装运行
```

排障

```
netstat -ano | findstr 3080                    # 端口占用（Windows） taskkill /PID <pid> /F
# 杀进程 cat ~/.dsh/settings.yaml                        # 全局配置（模型/推理档位）
```

插件开发

```
# 挂载到 profile（web 为例） # ① package.json 加依赖: "pkg": "link:C:\\path\\to\\pkg" # ②
cordis.patch.yml 加: #    - insert: #        - id: <插件 id> #          name: <npm 包名> cd
~/.dsh/profiles/web && pnpm install          # 安装
```

配置参考（settings.yaml）

<!-- [fix] 技术准确性核验：官方 DeepSeek 适配器档位为 off / high / max（low 为 pi-ai/opencode-go 网关档位），见 02-quickstart 2.3 注 -->

```
agent-default-model:   model: deepseek-v4-flash     # 或 deepseek-v4-pro   reasoningEffort:
high        # off（关闭思考/最快） / high（默认） / max（最强）
```

附录 B：[官方包速查大全](./appendix-packages.md) · 附录 C：[同模型多 Agent 实测](./benchmark.md)

附录 C：同模型 × 不同 Agent 实测对比（Benchmark）

实测日期：2026-08-13 | 模型：deepseek-v4-flash（同一 opencode-go 网关，同一 API key） | Agent：dsh / opencode / omp
*目的：**控制模型变量，对比 Agent 工程层的差异**（提示词、工具链、轮次管理）。*
扩展：2026-08-13 追加 T4（生成 markdown 报告）/ T5（多文件小重构），样本 3 轮 → 5 任务 × 3 轮。

方法

项	说明	
模型	deepseek-v4-flash（三 agent 均走 opencode-go 网关 + 同一 key，公平对照）	
Agent	dsh（headless profile）、opencode（opencode run）、omp（--print 非交互）	
任务	T1 创建文件 / T2 搜索统计 / T3 修复 bug + 验证 / T4 读数据生成报告 + 自复核 / T5 多文件重构 + 测试通过	
环境	Windows 11 + Node 24，独立工作目录，无预置上下文	
度量	耗时（秒）+ 正确性（人工核对产物/输出）	

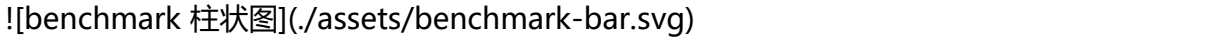
任务说明（T4/T5 为合成数据，无隐私风险）：

任务	输入	要求	人工核对标准	
T4 生成报告	data.json（4 条合成销售记录）	生成 report.md：数据表格 + 汇总（总销量/总营收/最畅销），自复核	汇总数字与数据一致（397 / 130853 / 耳机）	
T5 多文件重构	mathops.py + 依赖它的 main.py（含断言）	把 multiply 的乘法逻辑提取为 multiply 辅助函数，不改 main.py，python main.py 通过	出现 multiply 且 multiply 委托调用、断言全过	

结果（3 轮采样，取中位数）

Agent	T1 创建文件	T2 搜索统计	T3 修 bug+验证	T4 生成报告	T5 多文件重构	总计（中位）	正确率	
omp	10s ✅	11s ✅	15s ✅	21s ✅	13s ✅	70s	45/45	
dsh	22s ✅	15s ✅	48s ✅	26s ✅	19s ✅	130s	45/45	
opencode	35s ✅	37s ✅	42s ✅	30s ✅	28s ✅	172s	45/45	

总计 = 5 任务中位数相加；扩展前（仅 T1-T3）为 omp 36s / dsh 85s / opencode 114s。



原始 3 轮（秒）：

Agent	轮次	T1	T2	T3	T4	T5	
dsh	1 / 2 / 3	11 / 27 / 22	11 / 15 / 22	27 / 48 / 74	30 / 26 / 25	19 / 25 / 18	
opencode	1 / 2 / 3	18 / 35 / 35	19 / 37 / 37	50 / 41 / 42	30 / 49 / 30	28 / 28 / 32	
omp	1 / 2 / 3	9 / 10 / 12	9 / 11 / 12	13 / 33 / 15	23 / 21 / 17	16 / 12 / 13	

解读

1. **正确率三家持平（45/45）**——5 任务 × 3 轮全部正确。同模型下，Agent 工程层的差异主要体现在**效率**而非**能力上限**；即便引入文档生成与多文件重构两类新任务，正确性差异仍未出现。
2. **总耗时稳定排序**：omp < dsh < opencode（3 轮一致，中位数相加 70s / 130s / 172s）。
3. **任务类型对效率差异的影响**（本次扩展的核心发现）——按耗时画像，5 个任务可归为三类：

- **单步轻量**（T1 创建文件 / T2 搜索统计）：agent 间差距**最大**。opencode 35/37s vs omp 10/11s（约 **3.4×**）。任务越简单，越接近纯"启动 + 一次工具调用"开销，opencode 的启动/框架开销占比越高。
- **多步执行 + 验证**（T3 修 bug + 跑验证 / T5 多文件重构 + 测试）：dsh 与 opencode 在此类任务上差距**显著收窄**（T5：19s vs 28s；T3：48s vs 42s 甚至反超）。多工具链场景下，双方都要多次读文件/改文件/跑命令，框架层"每步思考"成本拉平了启动开销差异；dsh 的优势在 T5 上最明显（19s，接近 omp 的 13s），而 T3 的 bug 定位仍让 dsh 偏慢。

- **文档生成**（T4 读数据 → 写报告 → 自复核）：agent 间差距**最小**（omp 21s / dsh 26s / opencode 30s，约 1.4×）。报告生成是"读一次 + 写一次长文本"的回路，耗时主要由模型**输出 token 数**主导，Agent 工程层能优化的工具往返很少，故三家趋同。

  4. **反直觉点：T5（重构+测试）普遍比 T4（写报告）更快**（dsh 19s<26s、omp 13s<21s；仅 opencode 28s≈30s）。原因：重构是"短文本代码编辑 + 秒级 `python main.py` 验证"的回路，每步模型思考短；而写 markdown 报告需要一次较长文本生成，输出时间占比高——与第 6 章"工具链任务 90% 时间在模型思考/生成"的性能模型一致。

  5. **omp 的领先幅度随任务类型变化**：搜索统计类最大（T2 为 opencode 的 3.4×），文档生成类最小（T4 仅 1.4×）。说明 omp 的优势主要在**轮次少、工具调用快**，而非模型本身；当任务由长文本生成主导时，任何 agent 的工程层都难再压缩模型输出时间。

  6. **波动性提示**：opencode 在 T4 第 2 轮出现 49s 离群（其余两轮 30s），dsh 在 T3 波动最大（27→74s）。多轮中位数能吸收这类网络/工具链抖动，单轮比较需谨慎。

  7. **dsh 定位**：dsh 居中，且 headless 是"运行时 + 插件生态"——**同样的模型，通过插件（如提速插件降推理档）可进一步优化耗时**（见第 6 章）；T5 实测已接近 omp，说明多文件重构类任务是其相对强项。

  8. **样本警示**：3 轮 × 5 任务，同网关同 key，含网络抖动——方向可信，绝对值会随环境波动。

可复现

```
# 三个 agent 的命令（独立目录） dsh --profile headless "任务" opencode run "任务" --model opencode-go/deepseek-v4-flash omp "任务" --model deepseek-v4-flash --print
```

T4/T5 任务原文（T1-T3 见上一版本记录）：

<!-- [style] 任务原文代码块统一补 text 语言标签 -->

```
T4: 读取当前目录 data.json 中的数据，生成 markdown 报告 report.md：包含每行数据的表格（产品/销量/单价三列）和汇总（总销量、总营收、最畅销产品），生成后自己复核汇总数字与 data.json 一致，完成后回复 已生成 T5: 重构 mathops.py：把 multiply 函数内的乘法逻辑提取为私有辅助函数 _multiply(a, b)，multiply 改为调用 _multiply；main.py 依赖 mathops.py 并包含断言测试，不得改动 main.py；重构后运行 python main.py 验证所有断言通过，完成后回复 已重构
```

完整任务定义、合成数据与产物：`~/agent-bench/bench-ext/`（T4/T5 每 agent 独立目录）；T1-T3 见 `~/agent-bench/`。白皮书仓库内 `benchmark/` 目录规划中。

一页速查卡（Cheatsheet）

打印/收藏这张卡，日常用 dsh 不用翻书。

安装与启动

```
npx -y @deepseek-ai/dsh --version      # 免安装运行 npm i -g @deepseek-ai/dsh              # 全局安装 dsh web                                  # Web UI → http://127.0.0.1:3080 dsh --profile headless "任务"          # 一次性任务（脚本/CI）
```

配置（~/.dsh/settings.yaml）

<!-- [fix] 技术准确性核验：官方 DeepSeek 适配器（默认 provider=deepseek-official）仅接受 off / high / max；low 为 pi-ai（opencode-go）网关档位 -->

```
agent-default-model:   model: deepseek-v4-flash     # 或 deepseek-v4-pro   reasoningEffort: high        # off（关闭思考/最快） / high（默认） / max（最强）
```

推理档位

档位	用途	备注	
low	简单/批量/工具链廉价轮（最快）	仅实测网关（pi-ai/opencode-go）支持	
high	日常默认	官方适配器默认	
max	复杂推理/长链规划	官方适配器支持	
off	关闭思考/最快	DeepSeek 官方适配器档位（替代 low）	

工具链任务 90% 时间在思考——降档是最快提速。

核心命令

命令	用途	
dsh web	Web UI	
dsh --profile headless "任务"	一次性任务	
dsh --dump-config	看合成配置	
dsh plugin --profile <n> add <pkg>	装插件	

插件挂载（两步）

```
# package.json 加依赖 "my-plugin": "link:C:\\path\\to\\my-plugin" # cordis.patch.yml 加挂载 - insert:     - id: my-plugin       name: my-plugin

cd ~/.dsh/profiles/web && pnpm install && dsh web
```

提示词黄金法则

1. **写验收标准**：`"运行验证通过"` > `"写个脚本"`
2. **给上下文**：文件在哪、数据长啥样、读者是谁
3. **一次一个任务**：小闭环比巨型任务可靠

缓存省钱

- 长任务保持会话延续（别频繁新建）
- prompt 前缀稳定（别老改配置）
- 看 Web UI 底部"缓存命中 %"（实测可到 97%）

排障速查

现象	解法	
端口占	netstat -ano \	findstr 3080 → kill
模型无响应	查 settings.yaml + API key	
插件装不上 404	依赖用 ^0.1.0-rc.6 线	
长任务崩	全局安装（绕 npx）+ 降推理档	

术语（速记）

profile 形态 · bundle 插件组 · host/client 半 服务端/界面 · 扩展点 官方钩子 · waterfall 请求链 · compaction 上下文压缩 · headless 一次性 CLI · locations 产物路径

完整教程见 [dsh-handbook 白皮书](https://github.com/Electricitysheep/dsh-handbook) · [配置参考](./config-reference.md) · [术语表与命令速查](./appendix-glossary.md)