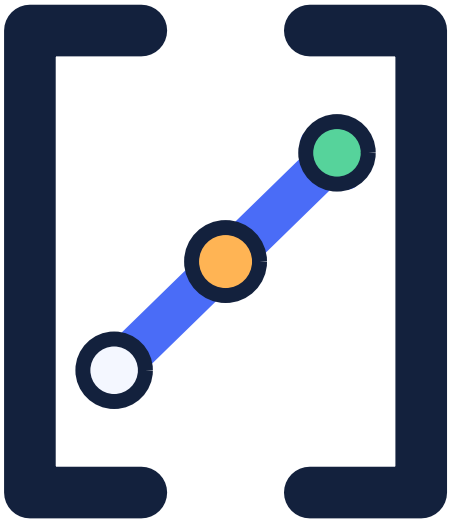


Agent Harness 源码内核

从一次工具调用开始，读懂六套编程智能体的运行系统

完整中文离线版 · 六条源码课程、横向对比、机制样本与实验

目录

Agent Harness 源码课程总目录

这套中文源码课程会从各项目共同面对的问题走进真实实现，不按产品逐项罗列功能，并且始终追问同一件事：模型给出下一步意图以后，Agent Harness（智能体框架）怎样把它变成真实动作，又怎样控制、恢复并核对这些动作？全书就从这个问题展开。

全书会反复处理一个很具体的任务：修好运费计算的边界错误，因为订单金额刚好为 100 元时，系统仍会收取 10 元运费。任务虽小，链条却很完整。你不用陷进业务代码，可以把注意力放在 Harness 怎样读取、编辑、执行测试、处理失败并判断是否结束，这些动作已经覆盖了编程智能体的一条主链。

下载完整中文 PDF

PDF 和本站使用同一份内容，CI 会从 MkDocs 导航里的这套 Markdown 生成它，因此两个版本不会各自演变。

姊妹项目

这套教材只回答「模型的下一步意图怎样变成可以控制的真实动作」。动作做完以后，究竟由谁判断它是否真的做对，则要看另一套系统，详见 Eval Harness 源码内核： 它会讲任务与数据集怎样定义、运行器如何并发与重试、评分和统计如何得出结论，以及发布门禁凭什么放行。 两套教材采用同一种读法，都从锁定到具体行号的源码出发。

第一部分：建立共同语言

顺序很重要。如果这是你第一次读 Agent Harness 源码，按顺序看完这一部分，更容易知道每次进入新项目时该去哪里找相同的问题。

1. 从这里开始：先跟完一次任务
2. Model、Harness 与 Environment
3. 一次任务怎样形成 Agent Loop
4. 工具、权限与执行边界
5. Session、Context、Memory 与恢复
6. Trace、Eval 与结果核对

这一部分只帮你建立读后续源码必需的坐标。读完以后，你应该能画出输入、决定、动作和观察怎样连成闭环，系统又怎样选择继续或结束，并解释模型、Harness 和环境分别能做什么、不能做什么。

第二部分：六条核心源码课程

六条课程都是主线，所以这里不按项目热度或模型能力排序。你先完整读完一条，等进入第二条课程以后，自然会看出各项目在相同位置用了哪些不同做法，到那时再比较更合适。

课程	最适合作为切入点的问题	入口
DeepSeek Harness	多包、Prompt、工具、Code Mode 与 Session 怎样装配成运行时	进入课程
Codex	Rust 核心怎样连接 Thread、Turn、审批、Sandbox 与多种客户端	进入课程

课程	最适合作为切入点的问题	入口
Gemini CLI	Turn、Scheduler、Policy 与工具生命周期怎样协作	进入课程
Claude	闭源产品契约与公开 Agent SDK 源码之间的证据边界在哪里	进入课程
pi	最小智能体核心怎样逐层长成编程智能体、会话与协议	进入课程
OpenCode	服务化 Session、Provider、Processor、Permission 和多客户端怎样共享核心	进入课程

每条课程都会带你走完一条完整任务链，但不会为了排版整齐，硬把不同项目塞进相同的目录和篇数。读六条课程时，真正该互相对照的是这六个问题：模型看到了什么，循环由谁控制，工具怎样执行，权限在哪里判断，状态怎样恢复，以及结果由谁观察。

第三部分：横向比较

至少读完两条源码课程以后再看这一部分。比较页不会给项目打总分，而会把同一个工程问题放回各项目的源码中，让你顺着调用和状态去找差异从哪里产生。

1. 运行时、配置与模型输入
2. Agent Loop、工具与执行
3. 权限、状态与恢复
4. 编排、协议与产品表面
5. 可观测性、Eval 与部署边界

第四部分：机制案例

每个扩展项目只用来讲清一种机制，六条主课程已经走过的完整路径不会在这里重复一遍。

- Aider：Architect 与 Editor 的职责分离
- Cline：审批检查点怎样进入循环
- Goose：Recipe 与 Extension 怎样扩展运行时
- mini-swe-agent：最小循环保留了什么
- OpenHands：Agent、Controller 与事件流
- Qwen Code：Server 与客户端边界

第五部分：实践与核对

实践会从能够稳定复核的任务起步，再逐渐增加依赖，所以你一开始不用调用真实模型。

1. 浏览器核对：打开永久链接，复原为什么存在、调用者、输入、状态、返回和下一站。
2. 最小 Agent Loop：运行确定性 Model Stub、工具请求、结果回填和停止条件。
3. 权限与崩溃恢复：区分获准、执行、记录和未知副作用。

4 独立评测管线：把执行 Trace 与结果判定分离。

需要真实模型的练习始终可以跳过，因为只用浏览器、锁定源码和确定性本地脚本，你也能完成核心学习路线。

查阅资料

术语表

怎样核对源码结论

按经验选择阅读路线

开始第一课

从这里开始：先跟完一次任务

如果你会一种编程语言，也用过 Git 和命令行，却还说不清模型给出意图以后 Agent Harness 还要做哪些工作，可以从这一页开始。读这套教材不要求你同时掌握 TypeScript、Rust 和 Python，也不用提前准备六家模型账号。

我们要完成的任务

后面的基础导读会反复回到同一个小仓库，我们会跟着一项具体任务往下走，看看 Harness 怎样接住模型的决定并推动任务。订单金额刚好达到 100 元时本应免运费，可测试失败了：

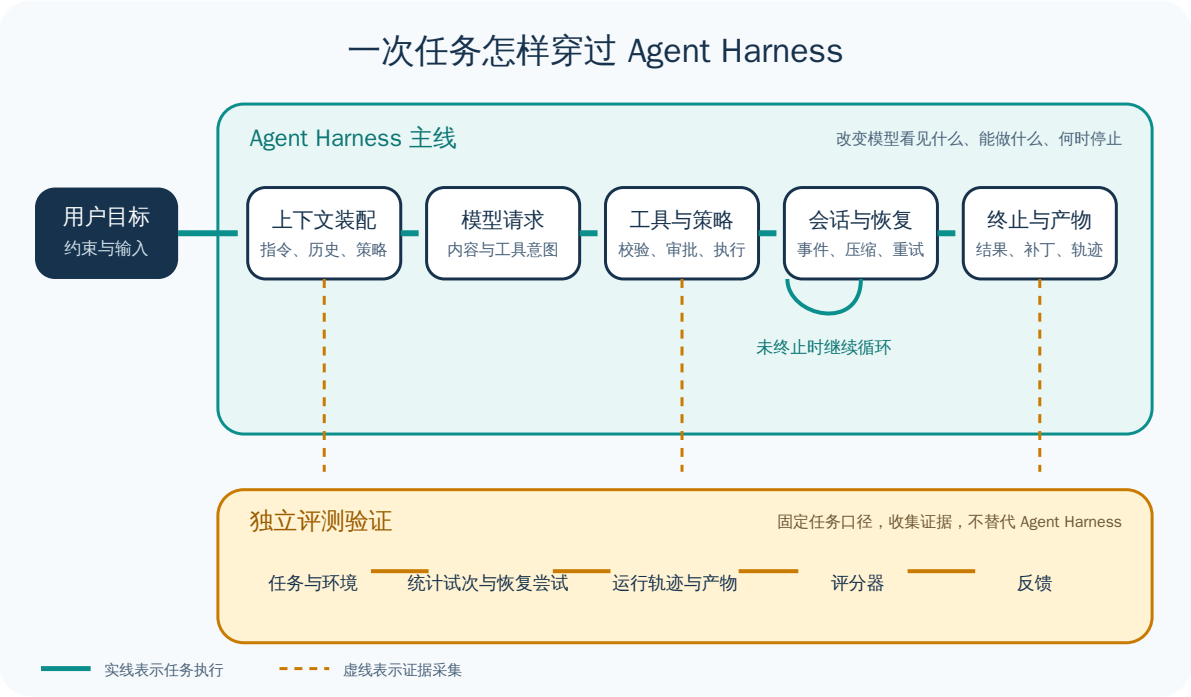
```
用户：修复订单金额为 100 元时仍收取运费的问题，并运行测试确认。

仓库：
src/shipping.ts
tests/shipping.test.ts

失败：
expected shippingFee(100) to be 0
received 10
```

普通聊天模型或许会建议把 > 改成 >=，但它既看不到真实文件，也无法运行测试，因此证明不了这个建议确实修好了问题。编程智能体还得读取仓库、找到实现、修改文件、运行测试并观察结果，最后判断任务有没有完成。Harness 就是把这一串动作接起来并控制它们怎样往下走的运行系统，也是本仓库要研究的对象。

三个角色

角色	在案例中负责什么	不负责什么
Model	根据当前消息决定读取文件、编辑代码、运行测试或给出答复	不直接拥有文件系统和终端权限
Harness	构造模型输入，解析工具请求，检查权限，执行或委托工具，保存状态并控制下一轮	不替模型决定业务修复内容
Environment	保存仓库文件，运行测试，返回真实输出	不解释任务是否已经满足用户目标

Trace（执行轨迹）会记下三者之间发生了什么，Eval（评测）则拿事先冻结的任务和判定方法来解释结果。两者都很重要，不过要等你看清执行链以后再展开。



一次任务的最小循环

1. Harness 把用户目标、仓库信息和可用工具交给 Model
2. Model 请求读取 src/shipping.ts
3. Harness 检查请求并让 Environment 读取文件
4. 文件内容作为工具结果进入下一次模型输入
5. Model 请求把 > 修改为 >=
6. Harness 完成写入并返回结果
7. Model 请求运行测试
8. 测试输出进入下一轮
9. Model 根据通过结果生成最终答复
10. Harness 保存会话并结束循环

这十步不表示每个项目都会使用相同的函数或事件名，它们只是给你一张阅读地图。先记住动作顺序。进入任何源码仓库以后，你都可以顺着这张图去找：输入从哪里进来，谁作出决定，谁真正动手，结果怎样被观察，循环又在什么条件下停住。

六个阅读问题

完成任意一条源码课程后，你应该能回答：

1 模型每一轮究竟看见了哪些消息、工具和状态？

2 谁创建下一轮请求，谁判断任务继续或结束？

3 模型给出的工具意图怎样变成真实文件或进程操作？

4 工具可见、策略允许、用户批准和系统权限分别在哪里判断？

5 Session、Context、Memory 和压缩后的历史怎样保存与恢复？

6 日志、事件、Trace、测试和 Eval 能核对哪些行为？

你可以拿这六个问题在不同项目之间定位，但不必要求各项目按同一种规范实现。有的项目把循环和工具分派放在同一个模块里，有的则把它们拆到服务端、协议层和多个客户端中，文章会照实保留这些差异。

先读五篇基础导读

1. Model、Harness 与 Environment：分清谁决定、谁控制、谁产生副作用。
2. 一次任务怎样形成 Agent Loop：沿失败测试走完输入、工具结果和停止条件。
3. 工具、权限与执行边界：区分工具可见、策略允许、用户批准和环境能力。
4. Session、Context、Memory 与恢复：理解短期输入、运行状态和跨任务记忆。
5. Trace、Eval 与结果核对：把「过程结束」和「任务正确」分开。

基础导读只补上读源码必需的那点共同知识。如果你已经能独立画出一次模型与工具怎样来回交接，就可以直接选择一条项目课程。

再选择一条源码课程

DeepSeek Harness：适合观察多包 Harness 如何组合上下文、工具、会话和反馈。

Codex：适合观察 Rust 核心、审批、Sandbox 和多种产品表面。

Gemini CLI：适合观察 Turn、Scheduler、Policy 和工具生命周期。

Claude：适合学习公开产品契约与 SDK 源码之间的证据边界。

pi：适合从极简核心逐层理解编程智能体。

OpenCode：适合观察服务化 Session 和多客户端架构。

第一次读时只做三件事就够了：运行或观察一个任务，找到核心入口，再沿源码跟完一条调用链。Provider 兼容层、界面怎样渲染、遥测怎样导出以及发布脚本都可以暂时跳过。

如果你不确定该读到多深，可以先看Starter、Builder 与 Maintainer 三层阅读路线。这三层不是考试等级，它们只是把问题逐层推深：先看任务怎样完整跑一遍，再追踪配置如何改变行为，最后检查文章里的结论能不能稳定复现。

源码页面怎样读

每个「第 N 站」都是调用链中的一个停靠点：

```
任务入口
  → 请求构造
  → 模型响应解析
  → 工具或文本分支
  → 权限与执行
  → 结果回送
  → 持久化与结束
```

每个源码站点都会给出锁定版本的链接，并说明谁调用这里、输入从哪来、代码改了哪些状态、结果回到哪里，以及下一步该读什么。长代码不会整段复制进仓库，链接负责提供完整 Context，正文则带你顺着数据和调用关系读下去。

三种核对深度

只用浏览器

打开永久链接以后，你可以依次确认文件、符号、谁调用谁以及测试写在哪里。只做到这一步，已经足以核对多数关于代码结构的结论。

使用本地 Checkout

运行上游测试或仓库提供的确定性脚本，你就能看见固定输入触发了哪些事件，又让哪些状态发生了变化。主线会尽量避开付费模型。

可选真实模型运行

只有某项结论确实要靠模型交互才能核对时，你才需要进入这一层。对应页面会说明要准备什么凭据、可能花多少钱、结果有哪些不确定性，以及看到什么才算成功。即使一次演示跑通，也不能据此断定整个产品都没有问题。

阅读完成的标志

学习不在于记住目录名。真正读懂一条课程以后，你应该能从用户输入讲起，用自己的话说明模型、工具和环境怎样接力改变状态，还能指出关键函数谁调用谁，并分别找到权限拒绝、执行失败和测试未通过记在了哪里。

下一篇：Model、Harness 与 Environment

查看完整课程目录

按经验选择阅读路线

不同读者看同一篇源码导读，想解决的问题并不一样，所以你不用把「从头到尾读完所有文件」当成目标。先选对深度。更合适的读法是先回答当前这一层的问题，等任务主线已经清楚，再进入更深的实现。

Starter：先建立一条完整任务链

如果你刚接触编程智能体，而且只熟悉一种编程语言，可以从这一层开始。

第一遍阅读时，只关注这五个直接决定任务主线的问题：

项目从哪里接收用户任务；

模型请求工具时产生什么数据；

工具结果怎样回到下一轮；

哪个条件让循环结束；

一次失败会在哪里被看到。

Provider（模型提供商）兼容层、终端怎样渲染、遥测怎样导出、发布脚本以及大量边界类型都可以暂时跳过。学完这一层以后，你不必背出文件名。只要合上文档还能画出一次任务怎样沿主链跑完，就算达到了目标。

你可以先读五篇基础导读，再看 pi 或 DeepSeek Harness 的系统地图，然后进入对应课程，依次读 Agent Loop、工具和 Session 相关章节。

Builder：解释一个变量怎样改变行为

如果你正在搭建智能体、工具系统或编程智能体，这一层会更贴近手头的问题。

看清完整任务链以后，再继续追踪这些会改变系统行为的变量：

配置怎样进入模型请求；

Tool Schema、调用参数和结果消息怎样转换；

可见、允许、批准和可执行为何是四层不同判断；

Steering、Follow-up、Hook、Extension 或子 Agent 在循环哪个位置介入；

Session 恢复和压缩怎样改变下一轮输入。

读到这一层时，要把实现和测试对照起来，因为只看一边很难确认某种行为究竟是怎样触发的。变量名不能代替行为。每读完一章，你可以挑一个配置项或状态字段，沿着调用链说明代码在哪里读取它，它又怎样改变后面的行为。

你可以先完整读完任意一条课程，再选一条架构明显不同的课程，等两边的主链都清楚以后，再看对应的横向比较。

Maintainer：判断结论能否持续被核对

如果你负责维护 Harness，或者关心安全治理、可观测性和评测系统，可以把阅读推进到这一层。

到了维护阶段，你要开始追问系统长期运行时会发生什么，以及文章里的判断还能不能持续核对：

错误是否被正确分类，而不是统一重试；

并发执行是否保留确定性的历史顺序；

权限拒绝、系统能力不足和工具失败能否区分；

Session、Trace 和结果产物能否建立稳定关联；

上游升级后，哪些源码锚点和解释会失效；

Eval 是否独立判定任务结果，而不是相信模型自述。

到了这一层，你需要完成确定性实践、检查上游测试，还要为自己关心的机制写出最小复现。一次演示成功证明不了系统换到其他任务、平台或部署条件以后仍然可靠，因此你给出结论时，也必须同时说清自己验证到了什么范围。

怎样在一篇文章中切换层级

每篇源码课主要写给 Builder，但也会留下三类入口，让不同阶段的读者都能找到自己该从哪里切入：

开头的「先看问题」和流程图供 Starter 建立直觉；

中间的源码站点、状态变化与设计解释供 Builder 深读；

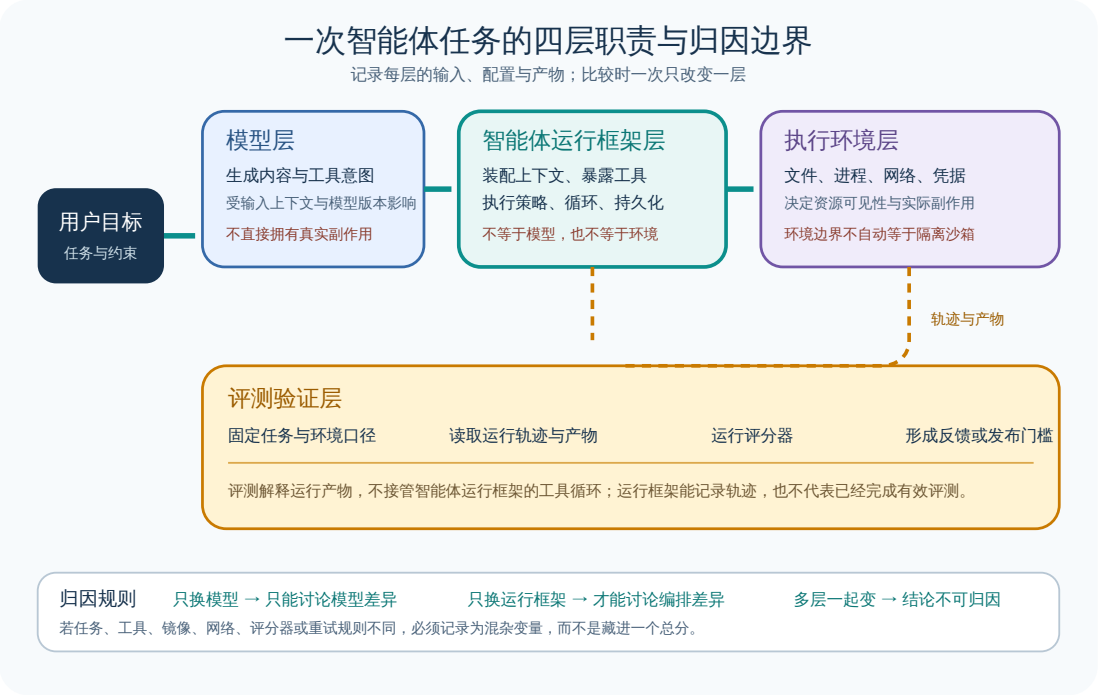
末尾的核对方法、失败路径和开放问题供 Maintainer 验证。

如果某篇第一遍读不懂，可以先回到课程 README 的系统地图，看看当前模块是在模型调用之前、工具执行之中，还是结果返回以后。只要先确定它在整条任务链上的位置，代码里谁调用谁通常就容易理解了。

返回课程总目录

Model、Harness 与 Environment：先分清谁在做什么

返回学习入口 · 下一篇：一次任务怎样形成 Agent Loop



第一次读 Agent 源码，人们很容易把什么都算在模型头上。模型说「我读了文件」，文件却可能根本没打开。终端返回成功，用户交代的事也未必办完。所以读实现前，你得先分清 Model、Agent Harness（智能体框架）和 Environment（运行环境）各自做了什么。下文把 Agent Harness 简称为 Harness。

从失败测试开始

这里继续用订单运费这个例子。

```
用户目标：修复订单金额为 100 元时仍收取运费的问题，并运行测试。

失败断言：
shippingFee(100)
期望：0
实际：10
```

模型想改对代码，得先看到任务、相关源码和失败输出。Harness 会把这些信息组成模型输入，还会告诉模型可以请求哪些工具。Environment 里放着真实文件，测试也在那里跑。少了任何一方，这件事都走不完。

Model：产生候选决定

Model 拿到一组消息和工具定义后，可以吐出文本、推理数据，也可以发出结构化的工具请求。它可能看着失败断言，猜边界条件写错了，也可能先要求读取 src/shipping.ts，等拿到证据再动手。

下面把模型输出简化成一个教学示例，这种写法不对应任何上游项目的精确协议。

```
{
  "type": "tool_call",
  "name": "read_file",
  "arguments": {"path": "src/shipping.ts"}
}
```

这只是一个候选动作，文件还没有打开，因为 Model 不知道宿主进程能不能读文件，也不知道产品策略会不会放行这个路径。

读到 Model 这一侧，你要找两个动作：请求在哪里建，响应又在哪里拆。常见线索有 Provider（模型提供商）、Client、Request、Response、Stream（流）、Message 和 ToolCall（工具调用）。先查模型实际收到了什么，又吐回了什么，这比从工具实现反猜模型协议靠谱。

Harness：把候选决定接入任务循环

Harness 夹在 Model 和 Environment 中间，通常要做好下面六件事。

1. 把系统提示、用户消息、历史、工具定义和动态状态组合成模型输入；
2. 发起模型请求并消费流式或非流式响应；
3. 区分文本完成、工具请求、错误、取消和超时；
4. 检查工具名称、参数、权限和用户确认；
5. 调用执行器并把结果重新写入消息历史；
6. 保存 Session、发送事件，并决定继续还是结束。

各个项目拆这六件事时，切法并不一样，Codex 把核心循环、协议和多种界面分到不同 crate 里。OpenCode 让多个客户端连到服务化的 Session。Claude 的 Agent SDK（智能体开发工具包）只公开应用进程能看到的 SDK 和控制协议，我们仍然看不到 Claude Code 内部怎么做。职责看着相似，模块名称却未必对得上。

回到运费案例，Harness 先把 read_file 请求交给工具注册表，检查路径是否落在工作区内，然后才调用读文件的实现。读到的内容会进入下一轮，变成模型的新输入。

```
{
  "type": "tool_result",
  "tool_call_id": "call-1",
  "content": "export function shippingFee(total) { return total > 100 ? 0 : 10 }"
}
```

Model 看到这个结果，才知道边界条件真写错了，也才有根据提出把 > 改成 >=。

Environment：副作用真正发生的地方

Environment 里有工作区文件、进程、网络、凭据、数据库、浏览器，也可能连着远程服务。工具最后落到哪个 Environment 里执行，就由那里的真实权限决定能不能做成。

产品策略放行 src/shipping.ts 之后，文件仍然可能打不开，原因很具体：文件不存在，宿主进程没权限，Sandbox 没挂载该目录，或者远程工作区断了线。这些都是执行失败，记成策略拒绝就把账算错了。

反过来看，宿主进程也许有能力删掉整个仓库，Harness 却不该因此批准删除。系统实际能做什么，产品又授权它做什么，这两笔记录必须分开。

观察	更可能属于哪一层	下一步检查
模型没有请求任何工具	Model 或输入构造	模型请求、工具定义、上下文
工具请求被判定为未知	Harness	工具注册和名称解析
请求被规则拒绝	Harness	权限策略和确认结果

观察	更可能属于哪一层	下一步检查
请求获准但文件打不开	Environment	路径、挂载和系统权限
测试进程退出码为 0	Environment 事实	继续检查测试是否覆盖用户目标

Trace 与 Eval 在哪里

Trace（执行轨迹）会记下任务里出现过的消息、工具调用、决定和输出，并保留它们发生的顺序。Eval 再拿任务说明、环境事实和判定方法去看这些记录，连同最后留下的产物一起判断。

如果测试进程根本没启动，Harness 仍有可能按自己的流程正常收尾，但你还不能判定任务成没成。如果测试只跑了 shippingFee(101)，退出码即使是 0，也证明不了金额 100 的问题已经修好。Eval 要判断的是任务结果，不能只盯着进程状态。

初学时先记住这个分工：Trace 留下发生过的事，Eval 再按明确口径来判断。第五篇基础导读会继续拆开这两件事。

用一条数据流检查理解

先看一段教学伪代码。

```
messages = Harness.构造输入(用户目标, Session, tools)
response = Model.生成(messages)

如果 response 是工具请求:
    decision = Harness.检查(response)
    result = Environment.执行(decision)
    Harness.把结果写回 Session
    进入下一轮

如果 response 是最终文本:
    Harness.保存并结束
```

这段代码有意略掉了流式事件、并行工具、取消、重试和压缩，只想让你看清三方各管哪一段，后面读真实项目时，文章会把每一行对到具体文件和函数上。

常见混淆

「模型会调用工具」

更准确地说，模型只会提出调工具的请求。Harness 拿到请求后决定怎么处理，Environment 最后才真正执行。把中间两层略掉，你就看不出是谁拦了请求，也分不清失败算在哪一层。

「有权限系统就有 Sandbox」

权限系统决定产品愿不愿意做这个动作。Sandbox 则卡住进程，限定它真执行时能碰到什么。前一层可以由规则和用户确认来做决定，后一层得靠操作系统、容器或远程执行环境真正限住。

「测试通过就完成了」

测试通过要成为有效证据，必须同时对上三件事：它覆盖了用户目标，跑的是修改后的代码，输出也确实来自这次任务。Harness 还得把这些事实挂到正确的 Session 和任务上。

读源码时的落点

打开一个新仓库时，你可以先找出下面这些位置。

模型请求入口；

消息和工具调用的数据类型；

核心循环或状态机；

工具注册与执行接口；

Session 或事件存储；

用户可见输出入口。

位置找齐了，再逐个问：这段代码是在适配 Model，还是由 Harness 控制流程？哪个函数又把动作交给了 Environment？把这几个问题答清，你不必先读完整个仓库，也能画出第一张靠谱的地图。

练习：给一句运行记录归属

请把下面四句话分给 Model、Harness、Environment 或 Eval，看看每句是谁作出的判断。

1 「下一步应当把 > 改成 >=。」

2 「本次写入需要用户确认。」

3 「shipping.test.ts 返回退出码 0。」

4 「边界值测试通过，而且测试文件未被修改，因此任务结果合格。」

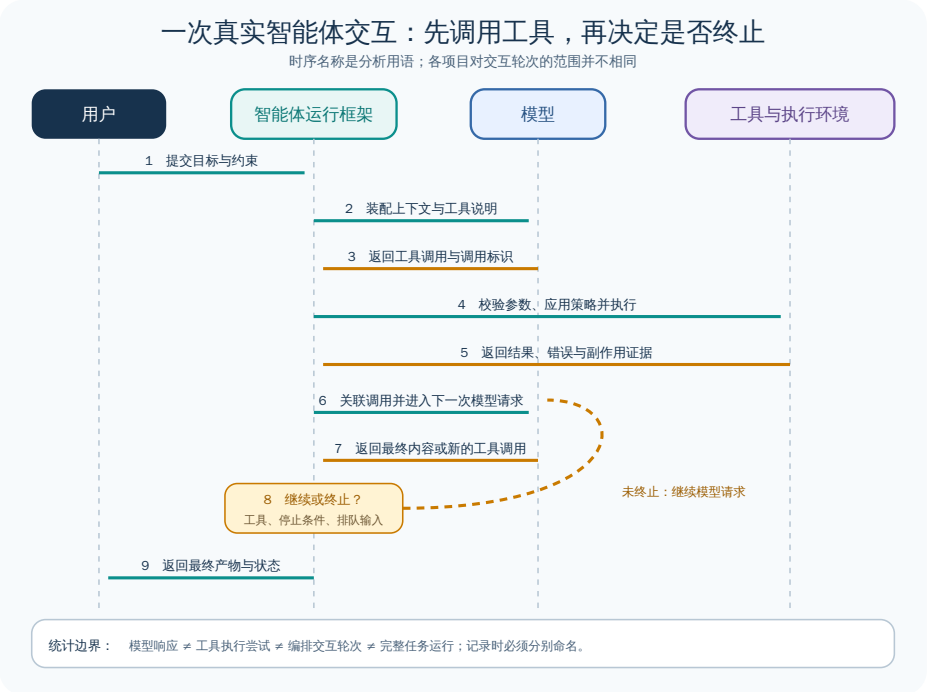
现在应该能解释

模型提出 edit_file 时，文件可能还一个字都没变。Harness 批准了请求，系统也未必真能执行。测试进程返回 0，同样不能保证用户目标已经达成。分别去看 Model 吐出了什么、Harness 怎么决定，以及 Environment 里真发生了什么。

下一篇：一次任务怎样形成 Agent Loop

一次任务怎样形成 Agent Loop

上一篇：Model、Harness 与 Environment · 返回学习入口 · 下一篇：工具、权限与执行边界



Agent Loop（智能体循环）不是把模型一遍遍重新调用就算完事。每一轮都得组好输入，接住模型吐出的内容，再走工具或文本对应的分支。Agent Harness（智能体框架）还要存下新状态，根据已经看到的信号判断下一步。下文把 Agent Harness 简称为 Harness。任何一环没说清，任务都可能打转、过早收尾，或在恢复后重复产生副作用。

一轮和一个任务不是同一个单位

回到运费案例，这件事要走好几轮才能办完。

1. 用户提出修复目标；
2. 模型请求读取实现；
3. 文件结果返回后，模型请求编辑；
4. 编辑结果返回后，模型请求运行测试；
5. 测试结果返回后，模型给出最终答复。

模型发出一次请求并收到响应，可以算一轮。一个任务通常会串起多轮，Harness 还要在轮次之间调工具、存状态，出错时也得接住。

不同项目会说 Turn（回合）、Step（步骤）、Cycle、Iteration、Run、Task 或 Thread。这些词不能直接画等号。每条课程开头都会先说明，该项目把它们分到了哪一层。

第一轮：构造模型真正看见的输入

用户只说了一句话，模型真正收到的输入却通常还有系统指令、工具定义、会话历史和当前运行状态。

```
{
  "system": "你在受控工作区中修改代码。修改后运行相关测试。",
```

```
  "tools": ["read_file", "edit_file", "run_command"],
  "messages": [
    {
      "role": "user",
      "content": "修复订单金额为 100 元时仍收取运费的问题，并运行测试。"
    }
  ],
  "runtime": {
    "workspace": "当前仓库",
    "approval_mode": "按风险询问"
  }
}
```

这些都是教学数据，不对应任何厂商 API。真实项目可能把 system、tools 和 runtime 拆到不同请求字段，也可能把动态状态直接写进系统提示。读源码时，请一直追到最后发给 Provider（模型提供商）的那个对象，别在配置文件的原始值处停下。

第二步：消费模型输出

模型吐出的内容可能长成下面这几种样子。

普通文本；

一个工具调用；

多个可并行工具调用；

流式文本和工具参数片段；

明确结束原因；

被取消、不完整或解析失败的响应。

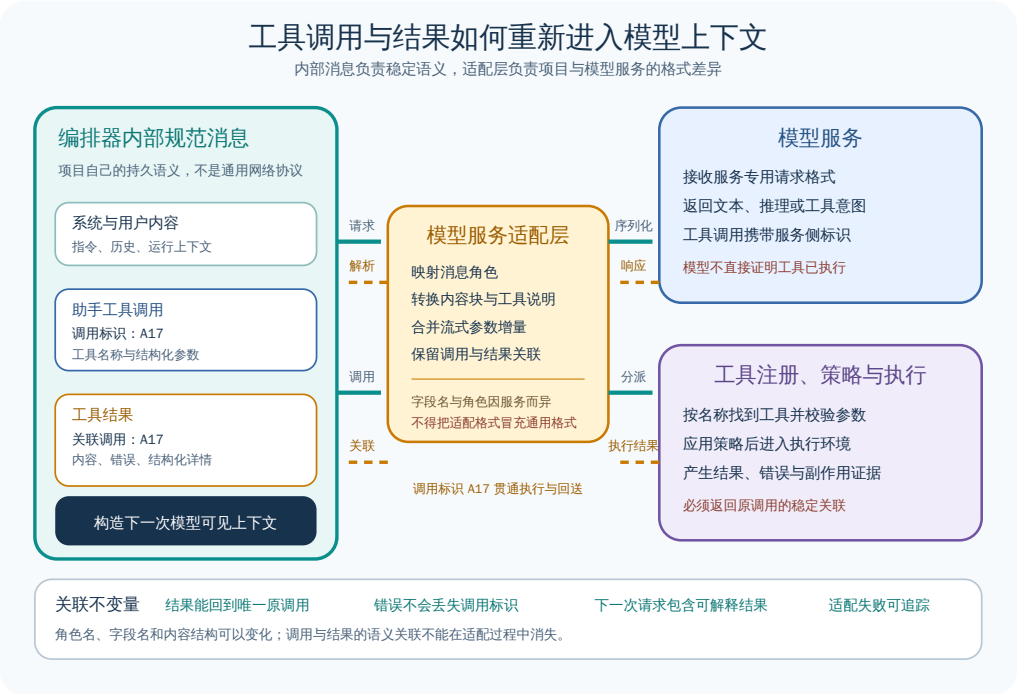
Harness 要把这些输出拆成内部事件或状态。如果工具参数是一片片流过来的 JSON，Harness 必须等它们拼完且解析成功，才能把请求交给工具层。响应若在参数到齐前中断，那半截参数绝不能当成有效命令。

在运费案例里，第一轮最好先要求读文件，不要看见失败断言就立刻动手修改。

```
{
  "id": "call-read-1",
  "name": "read_file",
  "arguments": {"path": "src/shipping.ts"}
}
```

调用 ID 必须保持稳定，后面的权限决定、执行结果和下一轮消息，都要靠它找回对应的模型请求。

工具结果怎样进入下一轮



文件工具读完文件后，Harness 通常会生成一条结果消息，再用调用 ID 把它挂回原请求。

```
{
  "tool_call_id": "call-read-1",
  "status": "success",
  "content": "export function shippingFee(total) { return total > 100 ? 0 : 10 }"
}
```

下一轮不会只把工具结果单独发走。Harness 会重新装配系统指令、必要历史、工具定义和这条新结果。有些 Provider 能缓存稳定前缀，有些项目则会在预算吃紧时压缩旧历史。怎么装，会直接改变模型下一轮能用哪些信息。

编辑和测试也照这个节奏走，模型先提请求，Harness 再控制执行，完成后把结果交给下一轮。

循环状态应该显式保存什么

一个能恢复的 Loop（循环）至少得存住下面这些状态。

状态	用途
任务或 Session 标识	把多轮事件归到同一任务
当前消息历史	构造下一次模型输入
工具调用 ID 和执行状态	防止结果错配或重复副作用
取消与超时信号	让模型请求和工具执行可以停止
Token 或轮次预算	防止无限循环
最后一个持久化位置	崩溃后判断从哪里恢复

如果只存最后的聊天文本，恢复时就无法确认某次写文件究竟执行过没有。系统可能又执行一遍同一工具，也可能因为丢了上次结果，让模型再次提出同一请求。副作用就这样重复了。

什么时候继续

出现下面这些情况时，Loop 通常还得继续往前走。

模型提出了一个可以处理的工具请求；

工具执行结束，结果需要交给模型解释；

权限被拒绝，但拒绝原因作为新观察允许模型选择替代方案；

可恢复错误已经转化为结构化反馈。

继续循环不等于照原样重跑一遍，因为权限被拒后，模型可以改用只读工具。命令跑失败后，它也可以根据错误输出修正参数，下一轮要带上这些新信息。

什么时候结束

常见的收尾条件有下面几种。

模型返回最终文本且没有待处理工具；

达到轮次、Token、时间或成本上限；

用户取消；

协议错误无法恢复；

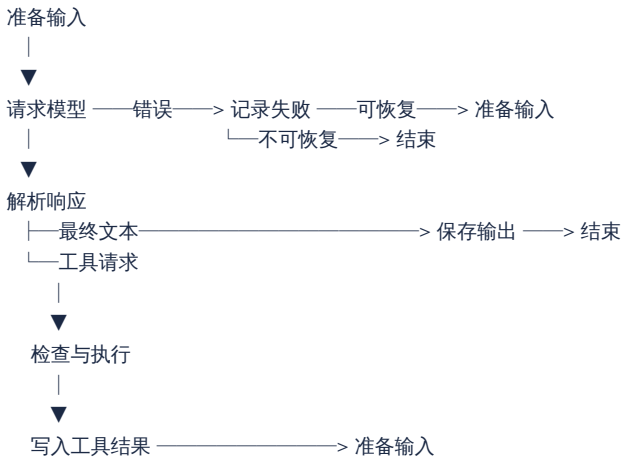
Environment 不再可用；

产品策略要求停下并交给用户。

「模型没有调工具」只是一个收尾信号。如果测试还在失败，模型写下「已经修复」也不会让任务自动变对。Harness 可以结束这次运行，任务是否完成仍要交给测试或 Eval 来判断。

一个最小状态机

下面这张教学状态机，把上面说的状态怎么转换画在了一起。



真实项目可能用异步生成器、事件总线、递归函数或显式 reducer 来写这段逻辑，看起来未必像图里的状态机。别被写法带跑。你只要找出它保存了哪些状态，又用什么条件把状态推到下一步。

过早结束的三个例子

编辑成功就结束

文件写成功，只能说明补丁已经落盘。代码能不能编译，测试能不能过，还得继续调验证工具去查。

命令退出码为 0 就结束

命令就算跑错了测试文件，也有可能返回 0。Harness 还要让模型或验证器对一遍命令，确认它真的在检查用户交代的问题。

模型说完成就结束

模型写下的文本只是候选解释。任务既然能验证，收尾时就应当带上环境里的真事实，比如相关测试输出和预期的文件差异。

无限循环从哪里来

下面几种情况，很容易让 Loop 一直转下去。

模型反复读取同一文件，没有新的状态提示；

工具输出被截断，模型每轮都重新请求完整内容；

权限拒绝没有作为结果返回，模型不知道请求被拒；

失败命令只返回「失败」，缺少可以修正的错误信息；

历史压缩丢掉了已经完成的步骤。

只设一个最大轮数，到点硬停，解决不了这些问题。Harness 还得把调用次数、拒绝原因、结构化错误、已完成的步骤和压缩摘要送回模型能看见的状态里。模型得知道哪里变了。

阅读真实 Loop 的顺序

进入源码后，你可以按下面的顺序找。

1. 找到公开入口怎样创建任务或 Session；
2. 找到发起模型请求的函数；
3. 找到响应类型和流式消费逻辑；
4. 找到工具请求分支；
5. 找到工具结果写回消息的位置；
6. 找到最终文本、限制、取消和错误的结束分支；
7. 找到状态持久化发生在转换前还是转换后。

先把这条调用链从头跟到尾，再去看并行工具、子代理、压缩和恢复。一上来就钻进扩展功能，你很容易把真正推动状态前进的那条主链看丢。

回到运费案例

这个任务要走完，下面这条事实链就得完整留下来。

失败测试
→ 读取实现
→ 确认边界条件
→ 编辑比较符
→ 运行目标测试
→ 观察金额 100 的断言通过
→ 生成最终答复

少了「目标测试」或「金额 100 的断言」，最后的答复就没有东西可以核对。Loop 把任务一步步往前推，任务判定再根据留下的事实解释结果。到这里，过程和判定才真正接上。

练习：补出缺失的一轮

现有历史是「用户目标 → 读取文件 → 文件内容 → 编辑成功」，模型却直接回复「已经修复」。请你补出 Harness 至少还要推动的一轮，并说清这一轮会带回什么新事实。

现在应该能解释

一轮只算一次模型交互，一个任务则会串起多轮交互和它们之间的工具执行。工具做完之后，结果要变成下一轮能看见的新事实。判断是否收尾时，既要看协议和资源限制，也要防止模型把自己的说法冒充成环境事实。声明不是事实。

下一篇：工具、权限与执行边界

工具、权限与执行边界：模型提出动作之后发生什么

上一篇：一次任务怎样形成 Agent Loop · 返回学习入口 · 下一篇：Session、Context、Memory 与恢复

工具让模型能读取和改变外部世界，也因此把模型不确定的输出接上了真实副作用。读源码时，你得分开查四件事：请求里有没有工具，策略放不放行，用户批不批准，操作系统最后有没有执行成功。它们分属不同层，不能压成一个布尔值，各层的失败也不能统统算成「执行失败」。

同一个动作经过五道边界

回到运费案例，这次模型想要跑测试。

```
{
  "name": "run_command",
  "arguments": {
    "command": "npm test -- tests/shipping.test.ts"
  }
}
```

命令真正跑起来以前，这个请求至少要过五道关。

1. 工具可见性：模型输入中是否声明了 run_command；
2. 协议校验：工具名和参数是否符合 Schema；
3. 产品策略：当前模式和规则是否允许运行该命令；
4. 用户确认：风险达到阈值时是否获得即时授权；
5. 执行环境：宿主、Sandbox 或远程运行器是否真的能执行。

前四道关只在审查请求有没有资格进入执行环境，第五步才会真正启动进程，产生副作用。

工具可见不等于可以自动执行

工具定义通常会告诉模型，这个工具叫什么、用来做什么，参数又该怎么填。

```
{
  "name": "run_command",
  "description": "在当前工作区运行命令并返回退出码和输出",
  "input_schema": {
    "type": "object",
    "properties": {
      "command": {"type": "string"}
    },
    "required": ["command"]
  }
}
```

把工具告诉模型，只是允许模型照这套协议生成请求。请求能不能自动执行，还要继续查权限模式、命令内容、路径范围、网络需求和用户选择。看得见工具，不等于已经拿到权限。

反过来，工具没写进模型输入，模型即使猜到系统支持终端，也没法按约定协议发出请求。有些 Agent Harness（智能体框架）会根据当前模式筛选工具，另一些会始终把工具告诉模型，等请求出现后再按策略拒绝。下文把 Agent Harness 简称为 Harness。两种写法会用不同方式影响模型行为和 Token 用量，你得分开追。

Schema 校验保护协议，不保护业务安全

Schema 能拦下缺少 command、类型错误或带有未知字段的请求，却判断不了命令会不会带来危险。

```
npm test -- tests/shipping.test.ts    结构合法，通常低风险
删除整个工作区                          结构同样可能合法，风险完全不同
```

参数合法，只说明 Harness 看得懂这个请求，并没有说它应当获准，因此参数校验和权限策略必须分开判断。合法不等于安全。

请求进入执行环境前，Harness 还可能给它补上工作目录、超时、环境变量和输出上限。所以正文要讲清原始参数经过了哪些改写，最后又变成什么。否则审计者只能看见模型最初吐出的文本。

产品策略怎样作决定

权限策略作决定时，可能会读下面这些信息。

- 工具名称；
- 命令和参数；
- 当前工作区；
- Session 模式；
- 用户或组织规则；
- 请求来源；
- 是否已有同范围授权；
- 动作是否可逆。

如果策略只返回 true 或 false，你就不知道它为什么放行或拒绝。返回值还得带上能说清原因的理由。

```
{
```

```
  "decision": "ask",
  "reason": "命令会启动本地进程",
  "scope": "本次调用",
  "final_arguments": {
    "command": "npm test -- tests/shipping.test.ts",
    "timeout_ms": 120000
  }
}
```

不同项目会用 allow、deny、ask、always allow、workspace scope 或模式枚举来表达结果，名字像，语义却不一定一样。真要比较时，你得追到每个选项实际排在什么优先级，授权又会存多久、管多大范围。

用户确认是策略的一部分，不是 Sandbox

用户点下「允许」，只是表示他当下同意这个请求。这一点击扩大不了操作系统权限，也证明不了命令本身安全。确认界面可能跑在 CLI、IDE（集成开发环境）、桌面端或远程客户端，这些界面未必都能用同一种方式向用户询问。批准穿不过系统边界。

无头模式如果没有确认通道，Harness 就得明确选一条路：直接拒绝，使用事先配好的策略，或把请求交给外部审批者。如果它不出声就自动放行，安全边界就已经变了，文章必须把这个选择讲出来。

保存确认结果时，要同时绑定调用 ID 和授权范围。如果只留一句宽泛的「用户曾经允许 run_command」，后面一条完全不同的命令就可能错用这份旧授权。

Sandbox 限制已经获准的动作

Sandbox 管的是动作真正执行时，进程究竟能碰到什么。

哪些目录可读写；

是否能够访问网络；

能否看到宿主凭据；

CPU、内存、磁盘和时间限制；

子进程和系统调用范围；

执行完成后是否保留工作区。

有的项目会很细致地询问权限，却默认让命令在宿主进程里跑。另一个项目可能把所有命令都放到远程隔离环境，产品策略却放得很宽。两种设计在防不同的风险，不能拿一把尺子量。

在运费案例里，测试命令就算获准，Environment（运行环境）仍可能返回下面的结果。

```
{
  "status": "failed",
  "exit_code": 127,
  "stderr": "npm: command not found",
  "executed": true
}
```

这里失败的是执行，权限并没有拒绝。模型只有看到准确错误，才能换成仓库支持的测试命令，或者如实告诉用户：环境有问题。

工具结果怎样安全地回到模型

工具吐出的内容可能很长，也可能夹着二进制内容或敏感数据。Harness 把结果交回模型前，通常要做下面这些处理。

保存退出码和错误类型；

截断过长输出并提供完整结果位置；

标记 stdout 与 stderr；

过滤不应进入模型的敏感字段；

保留原调用 ID；

说明动作是否实际执行；

把结果转换成 Provider 接受的消息格式。

只回一句「工具失败」，模型就没有线索去修正。可要是把几万行日志全部塞回上下文，Token 预算又会很快被吃光。信息太少和太多都要付出代价。Harness 得决定保留什么、截掉什么，才能让模型有足够信息继续工作，又不把上下文挤满。

四条失败路径

路径	executed	应记录什么	模型下一轮可以做什么
未知工具	false	工具名和可用工具提示	改用现有工具
参数不合法	false	Schema 错误和字段位置	修正参数
策略或用户拒绝	false	决定来源、理由和范围	选择低风险替代方案
环境执行失败	true 或状态未知	退出码、错误、部分输出	修正命令或报告阻塞

如果这四条路都被压成同一个 error，恢复逻辑、审计者和 Eval（评测）就都判断不了副作用究竟发生过没有。

恢复时最危险的情况

工具已经开始执行，进程却突然崩了，Session 里就可能只有「开始」事件，没有与它配对的「结束」事件。这时别自动假定工具没执行过。写文件、发消息、支付和部署都会留下副作用，盲目重跑很可能把同一件事做两遍。

想让恢复更安全，记录里至少要留下这些信息。

稳定调用 ID；

审批后冻结参数；

开始执行时间；

执行环境标识；

完成、失败、取消或未知状态；

可用于查询或补偿的外部操作 ID。

这些记录齐了，恢复逻辑才能看工具的性质，决定去查状态、做补偿、重新执行，或者把无法自动确认的情况交给人来判断。

在真实源码中怎么找

读真实源码时，不要只搜 permission。找到工具请求的数据类型，再沿着它一路追下去。

1. 工具定义在哪里进入模型请求；
2. 模型响应怎样解析成内部 ToolCall；
3. 工具注册表怎样从名称找到实现；
4. 参数在何处校验或改写；
5. 策略与用户确认的调用顺序；
6. 执行器使用宿主、Sandbox 还是远程环境；
7. 结果怎样转换并回到消息流；
8. 拒绝、取消、超时和未知状态有哪些测试。

读 Claude 课程时，还要额外核对证据到哪里为止。公开 SDK 可以让我们看到回调和控制协议，但它证明不了 Claude Code 内部所有策略和执行细节都已经公开。

回到运费案例

读源码、改比较符和跑测试，可能各自落在不同的风险级别。一套合理流程可以自动放行工作区内的读取，动手编辑前先把差异给用户看，遇到命令再按策略判断要不要询问。真正执行时，范围始终要卡在工作区内。

「所有工具都获准」证明不了任务做对了。真正的证据是：危险请求没有越界，必要动作确实执行了，覆盖边界值的目标测试也最终通过。

练习：判断阻断发生在哪一层

请你给下面四种失败分层：工具名称不存在、命令被策略拒绝、用户取消确认，以及进程在 Sandbox 里读不了工作区外的文件。再说说为什么它们不能统一返回「执行失败」。

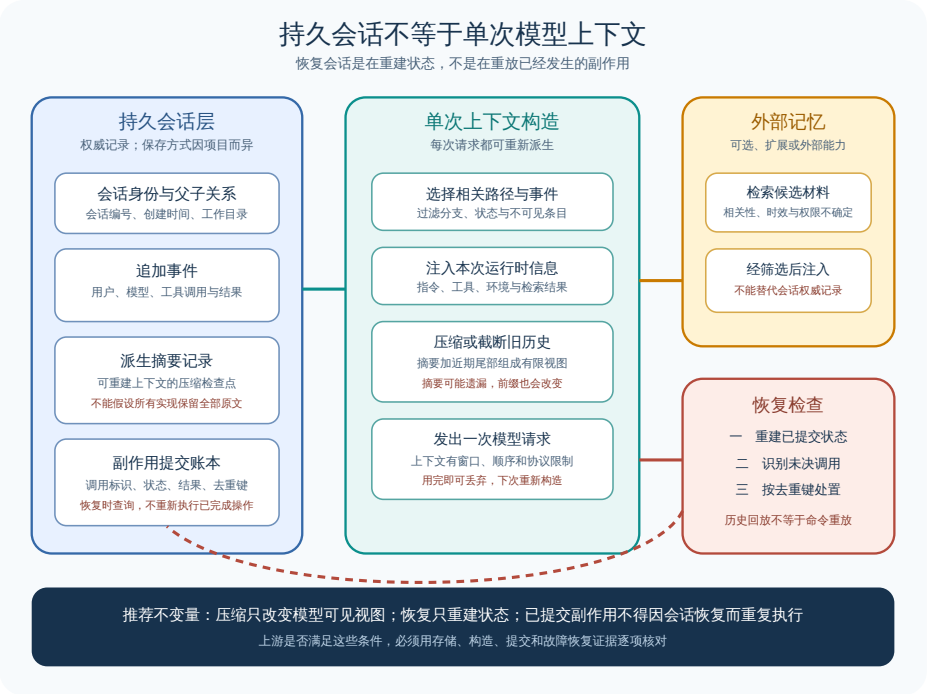
现在应该能解释

工具是否可见、Schema 是否合法、策略是否放行、用户是否批准，以及系统最后是否执行，要分成五次判断。权限管产品愿不愿意做，Sandbox 管进程真正能做到什么。工具结果还得准确写明副作用有没有发生，下一轮才能正确处理，恢复时也才不会盲目重跑。

下一篇：Session、Context、Memory 与恢复

Session、Context、Memory 与恢复：状态究竟保存在哪里

上一篇：工具、权限与执行边界 · 返回学习入口 · 下一篇：Trace、Eval 与结果核对



人们常把 Session、Context 和 Memory（记忆）混着说，读源码时却得分清它们各自管什么。模型这一次请求能看到的输入叫 Context。Session 记下一项任务怎样一轮轮跑到现在，Memory 留下以后做别的任务还用得上的信息。至于 Compaction（上下文压缩），它要解决的是历史太长怎么办，也就是怎样把原始记录收短。

用中断后的任务观察差异

假设运费修复已经读完文件并应用编辑，可进程在测试尚未运行时突然退出，那么系统重新启动后就需要回答：

用户原始目标是什么；

哪个文件已经修改；

编辑工具是否完成；

下一步应该运行什么测试；

之前的模型消息是否全部重新发送；

这次任务结束后，有哪些信息值得跨任务保存。

这些问题听起来都像在追问「之前发生了什么」，可你顺着实现查下去，会分别落到 Session、工具状态、Context 怎么拼，以及 Memory 要留什么。

Context：当前这一轮模型能看见什么

Harness（位于模型外部，负责输入装配、工具、权限、状态和产品表面的程序）通常会拿下面几类内容来拼 Context：

稳定部分：系统指令、工具定义、项目规则
任务部分：用户目标、附件、初始环境信息
历史部分：模型消息、工具请求和工具结果
动态部分：工作目录、剩余预算、任务进展、权限模式
检索部分：按需加载的文件、记忆和外部知识

Context 并不会照搬数据库里的整段 Session。Harness 还得看 Model 的窗口有多大、Cache 怎么用、任务跑到了哪一步，再挑出这一轮真正要发送的内容。

回到运费案例，恢复后至少要告诉模型三件事：原始目标是什么，文件已经怎么改了，哪个测试还没跑。要是只发最近那句「编辑成功」，模型既弄不清这次为什么要改，也选不出该跑哪个测试。

Session：任务的连续运行记录

Session 会把多轮 Message 和执行事实挂到同一项 Task 上。你若想以后还能追查并恢复这项任务，通常得在 Session 里留下这些内容：

稳定标识；

创建和更新时间；

用户消息与模型消息；

工具调用、决定和结果；

当前运行配置；

取消、错误和结束原因；

压缩摘要及其覆盖范围；

外部产物引用。

Session Store（会话存储）可以落在文件、数据库或 Server 里，也可以由 Host（宿主）应用提供接口。存到哪里并不关键。它必须说清哪些 Event 已经写稳，以及恢复时该从哪个提交点往下走。

界面只是投影。聊天窗口只把 Session 的一部分展示给你，内部还可能留着控制事件、Token 统计、Approval（审批）记录和 Tool 的原始输出。屏幕上的消息并不是完整记录。

Runtime State：尚未成为消息的运行事实

有些状态不适合直接写入消息历史，例如：

当前正在执行的工具；

取消令牌；

流式响应已经接收的片段；

并行工具的完成集合；

尚未持久化的事件；

当前工作区句柄或进程 ID。

这类状态可能只活在 Runtime（运行时）的内存里。进程一退出，凡是还没写过持久化接口的状态，恢复代码只能把它们标成未知，不能装作已经全都找回来了。未知也得明确记下来。

所以读源码时，你要查清哪些状态能恢复，哪些状态出了当前进程就没了。看到 resume(session_id) 这个 API，也不能顺势认定正在发生的副作用都能原样接上。

Memory：跨任务保留的信息

Memory 是留给未来任务用的，并不会把每段 Session 永久塞进模型输入。它通常留下这些信息：

用户偏好；

项目约定；

已确认的环境事实；

经过整理的解决方案；

可复用 Skill 或指令；

外部知识索引。

运费修复时具体调过哪一次工具，通常没必要长期记住。若项目已经定下「金额单位使用整数分」或「提交前运行 shipping 测试」，这类规则稳定，也能反复使用，更适合写进项目文档或 Memory。

往 Memory 里写内容时，必须同时交代来源，也要规定以后怎么更新。未经核实的模型猜测一旦跨会话留存，后面的任务就会反复受它误导。

Compaction：历史太长时保留什么

当消息历史接近模型窗口或成本预算，Harness 可以：

丢弃可重新获取的旧工具输出；

截断长日志并保留文件引用；

把早期消息总结成摘要；

保留最近若干轮原文；

固定关键任务状态和未完成步骤；

重新建立稳定前缀。

Compaction 把历史收短以后，仍得留下会影响后续判断的事实。只顾删旧消息，任务很可能接不下去。Summary 也一定会漏掉信息，所以用户目标、已经发生的副作用、没做完的步骤、约束和关键错误都要保住。

在运费案例里，下面这份教学摘要已经足够让模型继续干活：

```
目标：修复金额 100 时仍收费的问题。
已读取：src/shipping.ts。
已修改：条件 total > 100 改为 total >= 100。
尚未完成：运行 tests/shipping.test.ts 并确认边界断言。
```

如果摘要只剩一句「已修复运费问题」，模型很可能直接跳过测试，还没拿到证据就宣布结束。

恢复流程的四个检查点

1. 加载 Session

先确认 Session 确实存在且版本兼容，然后找到最后一个完整持久化的事件。

2. 重建可恢复状态

恢复消息、配置、工作区引用和压缩摘要，而进程句柄等不可恢复对象，则要重新创建或者标记失效。

3. 处理未完成副作用

如果工具只留下开始事件，却没有结束事件，就要根据工具类型查询实际结果、请求用户判断，或者执行补偿。不能统一重跑。

4. 构造下一轮 Context

把原目标、已完成工作、未完成步骤和新环境状态交给模型，只要这些事实足以支持安全继续，恢复就无须回放全部历史。

消息重放与副作用重放不同

把旧消息再发给模型，通常不会动到外部世界。旧工具若再执行一次，却可能把副作用也再做一遍。因此，恢复 Session 时必须把这两类动作分开：

对象	是否通常可以重建	主要风险
用户和模型文本	可以	版本差异导致解释变化
工具结果消息	可以从存储恢复	原始输出可能已被截断
只读工具	可能重新执行	环境已经变化，结果不再相同
写入、支付、发送等工具	不能盲目重跑	重复副作用
进程内流式片段	常常不完整	无法知道 Provider 是否已结束

幂等键、外部操作 ID 和事务记录可以减少重复执行的风险，但前提是每个工具先说清自己会造成什么副作用。这些机制只能降低重复风险，无法保证每次都能挡住重复动作。

在源码里怎么找

你可以从公开入口查起，搜索 create、load、resume、Fork、compact、History、store 等符号，再回答下面的问题：

1. Session ID 由谁生成；
2. 消息何时持久化；
3. 工具开始和结束是否分别记录；
4. Compaction 由长度、Token 还是显式请求触发；
5. 摘要怎样进入后续 Context；
6. 恢复是否重新连接 Provider 或执行环境；
7. 多客户端是否共享同一 Session 真值；
8. 哪些状态只存在于宿主内存。

只找到数据结构还不够，因为它没有告诉你系统实际怎样恢复。继续顺着 Call 路径查，直到找到哪些函数创建、写入、读取并消费这些结构。

回到运费案例

进程是在编辑后退出的。恢复时先确认写入工具已经留下完成记录，再加载改过的工作区，并告诉模型「测试尚未运行」，让它把验证补上。如果写入是否完成还说不准，就先读文件或检查 Diff，不要直接再改一遍。

任务结束后，可以留下 Session 供以后审计。长期 Memory 则只收已经核实，而且以后还用得上的项目规则。

练习：恢复时哪些动作不能重放

Session 最后两条事件是「开始编辑 shipping.ts」和「进程意外终止」，中间没有工具完成记录。恢复后应当直接再次编辑、直接运行测试，还是先读取文件并检查差异？请说明理由。

现在应该能解释

Context 管模型这一轮能看到什么，Session 记任务一路怎样跑过来，Runtime State 留住当前进程还在用的事实，Memory 则把信息带到未来任务。历史太长时，Compaction 会把它收短。做恢复时，你既要找回继续干活所需的状态，也要防止副作用重复发生。只把聊天再演一遍，解决不了这些问题。

下一篇：Trace、Eval 与结果核对

Trace、Eval 与结果核对：运行结束不等于任务正确

上一篇：Session、Context、Memory 与恢复 · 返回学习入口

Harness（位于模型外部，负责输入装配、工具、权限、状态和产品表面的程序）可以顺利跑完一项 Task。结果仍可能出错。循环停下，只能说明这次 Run（运行）结束了。Trace（执行轨迹）让你看见中间发生过什么，Eval 再拿明确的任务定义和判定方法检查结果。讲 Tool 或 Session 时，不必每篇都把这两部分塞进来。

运费案例有三个不同结论

模型完成编辑并运行命令后，不能立刻用一个「成功」概括全部结果，至少要区分：

1. 运行是否正常结束：模型请求、工具执行和消息循环有没有协议错误；
2. 目标测试是否通过：测试进程是否运行了正确文件并返回预期断言；
3. 用户目标是否满足：金额 100 的行为是否正确，同时没有破坏其他运费规则。

第一项看 Harness 有没有正常运转，第二项核对 Environment（运行环境）里究竟发生了什么，第三项才按用户的任务口径判断有没有做对。只读最终回复不够，因为它无法可靠回答后两项。

Trace 应该记录什么

先看一份教学数据，它展示了最小 Trace 需要串起哪些 Event。

```
{
  "task_id": "shipping-boundary",
  "events": [
    {"type": "user_message", "content": "修复金额 100 时仍收费的问题"},
    {"type": "tool_call", "id": "read-1", "tool": "read_file"},
    {"type": "tool_result", "id": "read-1", "status": "success"},
    {"type": "tool_call", "id": "edit-1", "tool": "edit_file"},
    {"type": "tool_result", "id": "edit-1", "status": "success"},
    {"type": "tool_call", "id": "test-1", "tool": "run_command"},
```

```
    {"type": "tool_result", "id": "test-1", "exit_code": 0},
    {"type": "final_message"}
  ]
}
```

真实项目未必照搬上面的格式，也可能用 JSONL（逐行 JSON）、数据库事件、Protocol（协议）通知或内存对象来记。载体可以换。事件顺序和关联标识却必须留下，否则你就查不出哪个请求带来了哪个结果。

Trace 不必收下所有原始内容。遇到敏感字段、超长日志和二进制 Artifact（产物），可以只存引用或哈希，但读取这些记录的人必须知道数据全不全，也要知道哪些地方截断或脱敏了。

最终文本为什么不够

模型可能说「测试已经全部通过」，但存在多种反例：

测试命令没有真正执行；

执行的是错误目录中的测试；

输出属于旧进程；

只跑了不覆盖金额 100 的测试；

测试通过，但修改了无关文件；

最终回复在测试失败后仍声称完成。

最后那段文字不可靠。它可能和实际执行毫无关系。任务只要能验证，Scorer（评分器）就该先读环境留下的产物，不能只凭结论听起来像不像真的。

Eval 的最小组成

一个可解释的 Eval 至少要包含下面这些部分。

组成	在案例中的内容
Task	修复金额 100 的免运费边界，并运行测试
Initial Environment	固定版本的仓库和依赖
Runner	使用目标 Harness 执行任务
Trace 与 Artifact	消息、工具事件、补丁、测试输出
Evaluator	检查目标断言、回归测试和文件差异
Result	通过、失败或无法判断，并附原因

这里可以让 Evaluator 运行确定性测试、套用规则、交给人工审查，也可以用模型评分。只要环境里有能直接核对的真值，就优先用确定性方法。

过程正确和结果正确

目标测试通过了，Agent 做事的过程仍可能不合格。它可能删掉失败测试，把结果硬编码进输出，或者顺手改了很多无关文件。绿色结果也会骗人。Eval 因此要同时查下面几类条件：

结果条件：金额 100 返回 0，其他边界仍正确；

过程条件：没有越界写文件，没有修改测试来迎合实现；

资源条件：运行时间、模型调用和工具次数处于合理范围；

证据条件：补丁和测试输出确实属于本次任务。

别只把过程条件写进 Prompt，然后相信模型会照办。模型到底守没守规矩，还得看得见证据。能验证的规则，尽量用文件 Diff、Permission 事件和测试来查。

失败、无法运行和无法判断

评测结果至少要区分：

任务失败：运行完成，但用户目标没有满足；

环境阻塞：依赖、网络或执行环境无法启动任务；

结果不完整：缺少必要产物，无法可靠判定；

Harness 错误：协议、工具或持久化发生故障；

用户取消：任务主动终止。

如果把这些情况全记成失败，产品质量统计会混进别的问题。直接丢掉环境错误也不行，那会遮住运行本身的故障。后面的高级 Eval 专题还会讲统计单位和恢复 Attempt，这里先把每种结果分开。

重试为什么会改变解释

测试服务若只是偶发超时，可以沿用同一项任务定义，再启动一次运行尝试。重试也会改变数据。基础设施出了故障，可以靠它恢复；Agent 如果确实改错了，就不能一直重试到碰巧成功，否则测出来的产品质量已经变了。

读源码时，你可以跟着一次重试往下查：

谁决定重试；

重试的是模型请求、工具调用还是整项任务；

旧的失败是否保留；

最终结果引用哪一次产物；

有副作用动作是否会重复。

这些问题能直接说明重试改动了哪些事实，比背住某个 Eval Harness 的类名管用。

从 Trace 回到源码

Trace 里的字段能追到哪段代码写出来，才算有可靠来源。读项目时还要继续找下面这些位置：

1. 事件类型或消息类型的定义；
2. 模型请求开始和结束的发射点；
3. 工具调用、权限决定和执行结果的关联方式；
4. Session 持久化使用的序号或时间；
5. 最终输出和错误怎样暴露给 CLI、SDK 或服务端；

6 是否存在 OpenTelemetry、日志、回放或测试适配器。

项目提供 Telemetry，不等于它已经做了完整的任务 Eval。仓库里有测试，也不等于测试覆盖真实的运行轨迹。写文章时要按各项目确实提供的能力说明这些接缝，不能一概声称所有 Harness 都内建了独立评测。

一个可复核的最小判定

对运费案例来说，跑完下面这组确定性检查，就能得到最小判定。

1. 从固定仓库版本创建工作区
2. 让 Harness 执行用户任务
3. 保存补丁和工具事件
4. 在干净测试进程中运行目标测试与回归测试
5. 确认测试文件没有被修改
6. 把结果关联到本次任务和补丁

即使模型最后没回文字，只要 Patch 和测试事实齐全，照样能判断任务结果。回复再自信也没用。目标断言失败就算没通过。

如何使用后续内容

六条源码课程只有在项目确实提供事件、日志、Telemetry 或 Eval 接缝时，才会讲到相应实现。DeepSeek Harness 的评测代码会放回它自己的 Call 链里解释。Codex、Gemini CLI、Claude、pi 和 OpenCode 则只讲锁定来源能够证实的能力。

跨项目评测、Trial、Attempt、RewardAdapter、Feedback 和独立发布评估都留到进阶专题。现在读基础 Agent Loop，还不用掌握这些内容。

练习：最终回复与任务结果冲突时相信谁

一次运行中，模型回复「所有测试已通过」，Trace 里却找不到测试命令，而独立验证又发现金额 100 的断言仍然失败。请分别写出 Harness 运行状态和 Eval 结果。

现在应该能解释

Harness 结束，只说明运行停在了什么状态。测试输出告诉你环境里真正发生了什么，Eval 再按任务口径判断结果。Trace 要把请求、决定和产物串起来。解释重试时也得分清它是在修复基础设施，还是在掩盖真实的产品失败。最终判定才不会走样。

你可以任选一条主要课程，从真实源码开始。

DeepSeek Harness

Codex

Gemini CLI

Claude

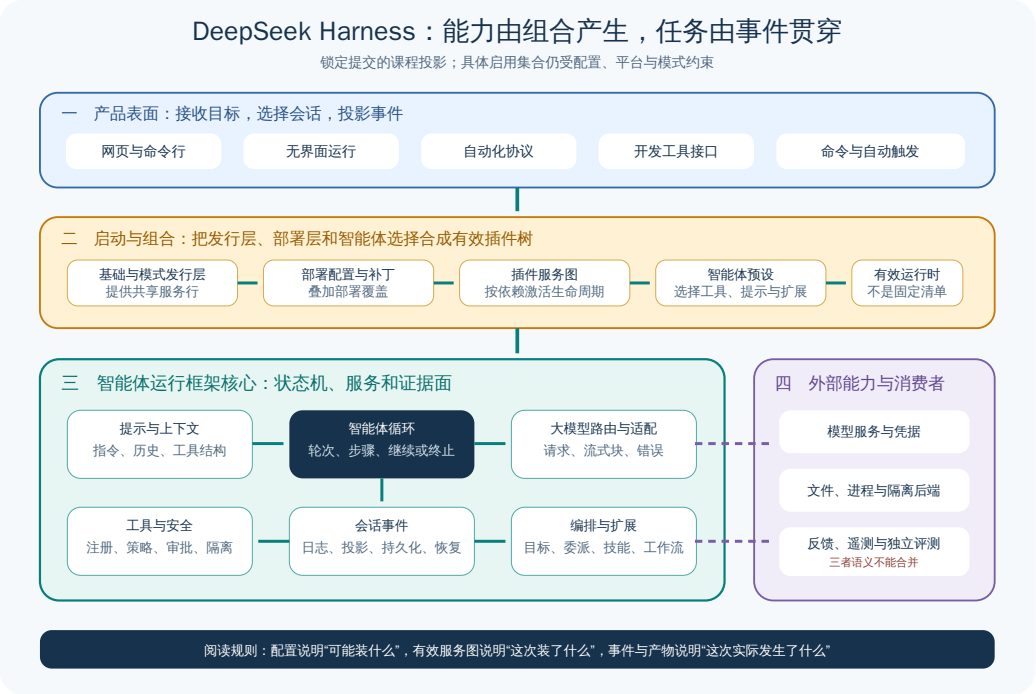
pi

OpenCode

DeepSeek Harness 源码课程

返回学习入口

DeepSeek Harness 是一套采用 TypeScript 多包结构的 Agent Harness。启动时，各个包会把 Preset（预设）、Prompt、模型调用、工具、Session、Code Mode（代码模式）、扩展和评测接缝接到同一套运行时。装配只是起点。课程会沿一次真实任务追踪配置怎样生成运行对象、模型输出怎样进入工具循环，以及状态怎样跨轮保存。源码目录只帮助定位模块，不能直接证明架构结论。



这条课程适合谁

如果你想了解一个功能较完整的 Harness 怎样通过包和配置获得不同能力，可以从这里开始。只要能读懂基础 TypeScript，就能跟上这条课程。Cordis、ACP、KV Cache 和 Code Mode 会在对应源码第一次出现时解释，你不必提前熟悉它们。

锁定来源

为了让正文解释始终对应同一份源码，本课程把 DeepSeek Harness 固定在提交 b150a551b8d465e31e418e1b2eaf5e79bbb7d28e。开始阅读前，可以先打开下面三个入口，认清配置、主循环和测试分别位于哪里：

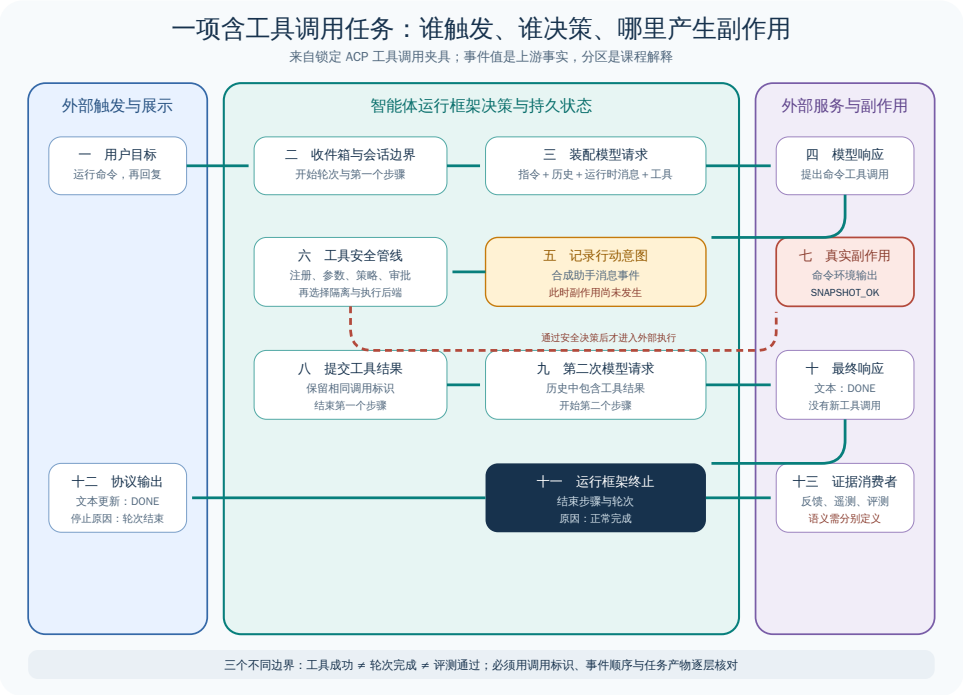
- 标准 Agent Preset
- Agent Loop 实现
- Agent Loop 测试

所有源码链接都指向这个提交，因此默认分支继续变化时，文章仍能与当时阅读的实现对应。

先看一项任务

课程始终围绕「修复失败测试」这项任务展开，并把任务推进过程逐步对应到 DeepSeek Harness 的真实对象。CLI 先选择 Preset，配置层据此装入 Prompt 和工具，随后 Agent Loop 请求模型，并把工具调用产生的结果写成新的 Message。Session 会继续保存这些事件，ACP 表面或评测适配器最终再从中读取运行产物。

CLI / ACP 输入
→ Preset 与 Bundle 装配
→ Prompt、模型与工具就绪
→ Agent Loop
→ 工具调用与结果消息
→ Session 与输出表面



这张图根据锁定源码重建阅读路线，并非运行时自动生成的 Trace。进入各章后，你还要沿每个箭头找到具体文件和对象，再用上游测试核对它们之间的调用关系。

仓库地图

区域	首轮关注点
apps/cli	参数、Preset 和启动入口怎样选择运行配置
packages/bundle	多个能力怎样通过 Bundle 组合
packages/core/agent-loop	模型、工具结果和停止条件怎样形成循环
Prompt 与上下文相关包	系统提示、历史、Cache 和压缩信息怎样进入请求
工具与 Code Mode 相关包	普通工具和代码执行路径怎样连接 Environment
Session 与协议表面	消息怎样持久化并暴露给 CLI、ACP 或其他宿主

第一次阅读时，可以先跳过网站、构建脚本和当前调用链没有经过的 Provider 兼容代码。

三层读法

Starter：读完前三篇，能解释 Preset 怎样装配能力、模型工具请求怎样回到下一轮。

Builder：继续读工具、Code Mode、Session 与 Compaction，选择一个配置字段追到行为变化。

Maintainer：阅读编排、产品表面和核对篇，检查失败分类、来源边界与外部 Eval 接口。

阅读顺序

1. 启动、Preset 与 Bundle：从 CLI 配置走到可运行 Agent。
2. Prompt、Context 与 Cache：模型一轮真正看见什么。
3. Agent Loop、模型与工具结果：沿主循环跟完一次工具调用。
4. 工具、权限与 Code Mode：工具意图怎样进入受控执行。
5. Session 与 Compaction：消息如何保存、压缩和恢复。
6. 编排与扩展：Bundle、子任务和扩展怎样改变运行时。
7. 产品表面、Feedback 与 Eval 接缝：运行事实怎样离开核心。
8. 测试、核对与适用边界：用上游测试回查课程结论。

课程会随着重构合并重复概念，编号只表示推荐阅读顺序，不能据此判断每个机制都属于独立层。

用贯穿任务复盘

读完课程后，可以从「CLI/ACP 收到运费修复目标」开始，沿前面的调用链复盘一次。先说明标准 Preset 选择了哪些能力，以及 Prompt 怎样带入项目上下文，再解释 Agent Loop 怎样接收 read/edit/test 的 Tool Result、Code Mode 与普通工具路径如何分工、Session 在中断后保存了什么，最后指出哪些 Feedback 或 Eval 接缝能够观察结果。复盘中的每一步都应回到课程里的某个锁定源码站点，这样才能形成可核对的完整链路。

如果只能说「Bundle 把它们组合起来」，复盘还没有完成，因为你仍需指出具体配置进入哪个对象、状态在哪里变化，以及下一站由谁消费。

完成课程后应该能回答

标准 Preset 怎样选择 Prompt、工具和运行能力；

Agent Loop 接收什么输入，何时再次请求模型；

工具结果以什么形态回到消息历史；

Code Mode 与普通工具路径各自解决什么问题；

Session 和 Compaction 保存或改写了哪些信息；

DeepSeek Harness 中实际存在的评测接缝在哪里，哪些仍需要外部系统。

从第一篇开始：启动、Preset 与 Bundle

启动、Profile、Bundle 与 Agent Preset

返回 DeepSeek Harness 课程地图

DeepSeek Harness 启动时不会把所有能力塞进一个写死的 Agent 类。它先由 Profile（配置档）选出要加载的 Bundle，再按顺序应用各层 Patch（补丁），最后合成 Host 配置。

会话选定 Agent Preset（智能体预设）后，Preset 才会把 Persona、Prompt、Tools、Compaction 和委派能力装入当前 Agent。两层不能混看。只有分清这两级组合，你才能解释「某个工具为什么会出现在这个 Agent 中」。

四个词先分清

名称	作用	典型内容
Profile	一次应用启动选择的组合入口	有序 Bundle 列表和用户 Patch
Bundle	可安装的一层 Host 配置	Session、模型、Sandbox、持久化、产品表面
Patch	对 Cordis Entry 列表的插入、替换或修改	增加插件、覆盖整段 config
Agent Preset	某类 Agent 的能力组合	Persona、Tools、Skills、Compaction、Delegation

Profile 回答「这个进程要装哪些系统能力」，Preset 则回答「这个 Agent 能看见并使用哪些能力」。因此，Sandbox Registry 可以由 Host 持有，但 Bash Tool 要不要暴露给某个 Agent，仍然要由 Preset 决定。

```
启动参数选择 Profile
  → 依次解析 Bundle Patch
  → 追加 Profile 自己的 Patch
  → 合成 Host Entry 列表并启动服务
  → 会话选择 Agent Preset
  → 在 Agent Scope 中挂载 Prompt 与 Tools
```

第 1 站：Profile 清单保存有序 Bundle 列表

源码：查看 Profile 类型

```
export interface DshProfileManifest {
  bundles?: string[]
}

export interface Profile {
  name: string
  dir: string
  layers: ProfileLayer[]
  patchPath: string
  patches: PatchOptions[]
}
```

调用者：命令行启动器根据 Profile 名称加载配置。

输入：Profile 目录中的 package.json 和 cordis.patch.yml。

状态变化：这里只描述加载后的数据结构，还未挂载插件。

返回：有序 Bundle Layer 与用户 Patch 的 Profile 对象。

下一站：loadProfile() 解析每个 Bundle 包及其 Patch。

后一层可以覆盖前一层，所以 Bundle 的排列顺序会直接改变配置含义。顺序会改变结果。做课程实验或复现问题时，不能只记「安装了哪些包」，还要一并保存 Bundle 顺序和合成后的完整配置。

第 2 站：内置 Bundle 优先从当前安装解析

源码：查看双锚点解析

```
for (const anchor of [installAnchor, join(profileDir, 'package.json')]) {
  const dir = packageDirFromAnchor(anchor, packageName)
  if (dir !== undefined) return dir
}
```

调用者：loadProfile() 为清单中的每个 Bundle 调用 resolveBundleDir()。

输入：Bundle 包名、当前应用安装锚点和 Profile 目录。

状态变化：没有写配置，只决定真实包目录。

返回：首先命中的 Bundle 路径；两处都找不到则抛出带安装建议的错误。

下一站：读取 Bundle 的 package.json，找到声明的 Patch 文件。

先看安装锚点。解析器从当前应用的安装锚点开始查找，以便让内置 Bundle 与正在运行的应用来自同一份安装。只有第一处找不到时，它才会去查 Profile 的本地依赖，因此同名包无法悄悄遮蔽内置核心层。

第 3 站：加载失败要尽早暴露

源码：查看 loadProfile()

```
const bundles = manifest.dsh?.profile?.bundles ?? []
const layers = bundles.map((packageName): ProfileLayer => {
  const packageDir = resolveBundleDir(...)
  const bundleManifest = JSON.parse(...)
  const declared = bundleManifest.dsh?.bundle?.patch
  if (declared === undefined) throw new Error(...)
  return { packageName, packageDir, patchPath, patches: loadOverlayPatches(...) }
})
```

调用者：应用 Boot 逻辑。

输入：Profile 名称、应用安装锚点、Harness Home 和是否读取用户层。

状态变化：缺省模板可能首次初始化；每层 Patch 被解析。

返回：完整 Profile；缺包、坏清单或伪装成 Bundle 的普通包会直接失败。

下一站：composeEntries() 按顺序应用所有 Patch。

做恢复诊断时，你可以传入 userLayer:false 跳过已损坏的用户 Patch，单独查看 Bundle 的默认组合。这个入口专供故障恢复使用，正常启动时并不会自动忽略用户配置。

第 4 站：最终 Entry 从空列表按层合成

源码：查看 Entry 合成

```
return applyEntryPatches(
  [],
  structuredClone(layers.flat()),
  warn,
)
```

调用者：Boot include 和配置诊断使用同一合成函数。

输入：已按 Bundle 顺序排列、最后再加用户层的 Patch 数组。

状态变化：Patch 被应用到空 Entry 列表；使用 clone 避免修改原配置对象。

返回：真正要挂载的 Cordis Entry 列表。

下一站：Cordis 加载这些插件并按服务依赖激活。

Patch 处理某一行的 config 时，会直接替换整个对象，不会逐层合并字段。因此，哪怕用户只想改一个字段，也要重复写出该层的完整 config，否则其他设置就可能随整体替换一起消失。所以，讲解「最终运行配置」时要查看完整合成结果，不能只读某个 Bundle 的原始文件。

Host Plane 与 Agent Plane

标准 Agent Preset 在开头就把责任划分清楚：Host 组合持有 Registry、Sandbox、Approval、Persistence 和 Model Route，Preset 只在 Agent Scope 中注册模型能看见的能力。

源码：查看标准 Preset 的边界说明

```
- id: persona
  name: '@deepseek-ai/dsh-persona'
  config:
    text: >-
      You are a coding agent powered by the {{model}} model.

- id: tool-bash
  name: '@deepseek-ai/dsh-tool-bash'
  disabled: !!js process.platform === 'win32'
```

源码：查看 Persona 与平台工具

Preset 选择要向模型展示 Bash 还是 PowerShell，底层执行器和 Sandbox 则仍可由 Host 提供。平台条件会在组合配置时生效，所以即使源码里存在 Bash 插件，Windows 上的 Agent 也未必能够使用 Bash。

为什么需要 Scope 与 Realm

一个进程可能同时挂载多个 Preset，还要为多个 Session 提供服务。如果 Agent 的私有服务发布到 Root Realm，另一个 Preset 注册的同名服务就会与它冲突，Host Reader 也可能读到错误实例。因此，Preset 用隔离的 Realm 约束 Agent 私有状态，跨 Session 共享的 Registry 则留在 Host，并按 Agent/Session ID 分区。

判断一个服务放哪一层，可以问：

1. 它是否在 Session 创建前就要解析？
2. 多个 Preset 是否必须共享同一个 Registry？
3. 状态是否天然按 Agent 或 Session Key 分区？
4. 是否有 Host 表面直接读取它？
5. 多次挂载是否会发生重复注册？

判断服务应该归属哪一层时，不要只看 Agent 会不会用到它，还要同时核对服务的生命周期和实际消费者。

用失败测试任务走一遍装配

1. CLI 选择一个 Profile，Profile 加载 Base、模式和产品表面 Bundle。
2. 用户 Patch 覆盖工作区、模型路由或遥测等部署设置。
3. Host 挂载 Session、LLM、Approval、Sandbox、Persistence 等共享服务。
4. 新会话选择 standard Preset，挂载 Persona、文件和 Shell Tool、Skills、Compaction 与委派工具。
5. Agent Loop 从 Scope 获取这些服务，开始第一 Turn。

排查工具缺失时，也要沿着装配顺序往下查。先确认 Profile 是否包含对应 Bundle，再检查 Patch 有没有禁用或覆盖它，然后核对 Host Registry 是否已挂载、Preset 是否已暴露该工具。最后，再看平台条件是否关闭了它，以及 Agent Scope 是否解析到正确实例。

下一篇进入模型请求内容：Prompt、运行时 Context 与请求 Cache。

Prompt、运行时 Context 与请求 Cache

返回 DeepSeek Harness 课程地图

模型在一轮中真正看到的内容，并不来自某一个 Prompt 文件。

DeepSeek Harness 会在每个 Step 重新收集并排列 System Prompt 的 Section（区段）、动态 Context、Tool Schema、变量和 Session 历史。

随后，模型 Adapter（适配器）会补齐路由的默认值，再把这些内容序列化为 Provider（模型提供商）请求。

模型输入由五部分组成

```
System Sections
+ 动态 Context Snapshot
+ Session 派生 Messages
+ Tool Schemas
+ Provider / Model / 采样与推理参数
= 一次冻结的 GenerateOptions
```

Section 通常是稳定规则，如 Persona、工具使用说明和项目指令。

Context 是每 Step 变化的运行信息，如工作区、计划状态或外部投影。

Messages 从追加式 Session Event 派生。

Tools 来自 Agent Scope 中已注册的工具集合。

Adapter Defaults 取决于精确 Provider 与 Model。

不要把这五类内容都统称为「上下文」。它们进入请求的位置和变化频率不同，如果混在一起谈，后面就无法准确分析缓存与恢复行为。

第 1 站：SystemPrompt Registry 保留结构化 Assembly

源码：查看 PromptAssembly 类型

```
export interface PromptAssembly {
  sections: AssembledSection[]
  contexts: AssembledContext[]
  tools: ToolSchema[]
  variables: Record<string, string | undefined>
}
```

调用者：Agent Loop 的 preStep() 调用 systemPrompt.assemble()。

输入：当前 Agent Scope、AbortSignal 和各插件注册的 Section、Context、Tool Provider、变量。

状态变化：Registry 收集并排序贡献项，但不直接写 Session。

返回：仍保持 Section/Context 边界的 PromptAssembly。

下一站：Loop 分别渲染 System Prompt 与 Context Snapshot，再把 Context 作为用户侧消息投影。

Assembly 会保留 Section、Context 和 Tools 的结构，这一步仍未生成最终文本。因此，扩展仍然能替换某个具名 Section、调整 Tools 顺序，或者检查 Context 由谁提供，只到模型请求的边界才把它们渲染成最终文本。

第 2 站：同名 Scoped Section 可以覆盖全局项

源码：查看 assemble() 合并规则

```
const sectionByName = this.layers.merge(scope, layer => layer.sections)
const sectionDefinitions = [...sectionByName.values()]
  .sort((a, b) => a.order - b.order)

const completeSections = sectionDefinitions.filter(
  section => section.complete === true,
)
if (completeSections.length > 1) throw new Error(...)
```

调用者：每个模型 Step 的 preStep()。

输入：全局与 Agent Scope 的多层 Registry。

状态变化：同名 Scoped 项遮蔽 Global；Section 按 order 稳定排序；Waterfall 可以转换 Assembly。

返回：唯一有效的 Assembly；多个 complete Section 同时生效会失败。

下一站：renderPrompt() 严格插值变量并去掉空 Section。

complete Section 的含义是「用这一整段取代普通 Section 集合」，适合需要全面接管 Prompt 的特殊模式。如果同时启用多个 complete Section，系统就无法确定该选哪一段，因此会直接报错，不会自行猜测优先级。

动态 Context 为什么进入消息而非 System 字符串

preStep() 先渲染 Context Sections，再交给 RuntimeContextProjection 生成上下文消息，然后连同 Inbox claim 到的消息一起送入当前 Step。这样处理后，Session 可以分别记录真实的用户输入和运行时投影，Context 变化时也不必改动相对稳定的 System 前缀。

源码：查看 Step 前组装

```
const claimed = this.inbox.claim(target, position.turn)
const assembly = await this.loopCtx.systemPrompt.assemble(...)
const sections = renderContextSections(assembly)
const context = this.runtimeContext.project(
  joinContextSections(sections),
  sections,
)
```

Projection 只有在内容真正改变时才生成新的 Snapshot（快照），不会为每个 Step 重复追加含义相同的运行时 Context，因此你仍能追溯模型在该 Step 看到的信息。

第 3 站：请求 Header 把会影响前缀的配置做成快照

源码：查看 Request Header 规范化

```
export function canonicalHeader(header: EpochHeader): EpochHeader {
  return {
    config: header.config,
    ...header.system?.length ? { system: header.system } : {},
    ...header.tools?.length ? { tools: header.tools } : {},
  }
}

export function foldRequestHeader(events, from?) {
  let state = from
  for (const event of events) {
```

```
    if (event.type === 'request/header') state = canonicalHeader(event.data.header)
  }
  return state
}
```

调用者：Agent Loop 构建请求时创建 Header；Session 恢复与诊断折叠 Header Event。

输入：模型路由配置、System Prompt、Tool Schema 和 Adapter Defaults。

状态变化：空 System/Tools 被规范为缺省字段；最新 Header 可从 Event Log 重建。

返回：稳定的 EpochHeader 或日志中最近快照。

下一站：Loop 比较旧 Header，只在初始、恢复或变化时追加新事件。

Header 判断两次请求是否相等时，还要比较 Tool 的顺序，因为 Provider 请求前缀和工具选择的含义都可能受这个顺序影响。如果只对比 Tool 名称的集合，Schema 或排序发生的变化就会被漏掉。

第 4 站：Loop 冻结最终 GenerateOptions

源码：查看 buildRequest()

```
preparedCall = await this.loopCtx.llm.prepareCall(proposedConfig, signal)

const header = canonicalHeader({
  config,
  ...system ? { system } : {},
  ...tools.length > 0 ? { tools } : {},
})

const request = markAgentLoopRequest(deepFreeze({
  ...header.config,
  messages: boundaryMessages,
  ...header.system !== undefined ? { system: header.system } : {},
  ...header.tools !== undefined ? { tools: header.tools } : {},
  sessionId: this.session.id,
  signal,
}))
```

调用者：每个 Step 的模型流开始前。

输入：路由种子、Waterfall 修改、Prompt、Tools、Session Messages 和 AbortSignal。

状态变化：精确 Adapter 解析默认值；Header 与 Request Context 的变化追加进 Session。

返回：冻结的 GenerateOptions，以及可选 PreparedCall。

下一站：Prepared Adapter 或 LLM Registry 将它序列化并发起流请求。

deepFreeze 冻结最终请求对象，防止下游在发送期间继续修改同一份数据。Session 也只在 Header 发生变化时追加新事件，因此无需在每个 Step 重复保存大段相同的 System 和 Tool Schema。

Provider Cache 命中来自稳定前缀

DeepSeek Harness 的 Agent Loop 没有在本地创建名为「KV Cache」的模型缓存。它会尽量保持请求前缀稳定，并在历史末尾追加新内容，从而给 Provider 复用 Prompt Cache 的机会，然后再把 Provider 报告的命中 Token 统一记入 Usage。

第 5 站：真实 API 测试检查第二次以后命中

源码：查看请求 Cache 端到端测试

```
const usages = [...agent.session.events]
  .filter(e => e.type === 'assistant/message')
  .map(e => e.data.usage)

for (const usage of usages.slice(1)) {
  expect(usage!.cacheReadTokens ?? 0).toBeGreaterThan(0)
}
```

调用者：带真实 API Key 的可选端到端测试。

输入：长 System Prompt、含 ToolCall 的两 Step Turn 和后续 Turn。

状态变化：真实 Provider 请求发生，Usage 被记录到 Assistant Message Event。

返回：测试断言首个请求之后都报告 Cache Read Token。

下一站：性能分析使用 Usage 评估前缀稳定性和成本。

没有 Key 时，这项真实 API 测试会直接跳过，所以普通离线测试只能证明 Harness 按追加式历史构造请求，无法证明真实 Provider 一定会命中缓存。缓存命中必须实测。Cache 是一项需要在真实请求中观测的性能结果，不能仅凭「源码结构看起来稳定」就认定已经命中。

哪些变化最容易破坏前缀

每 Step 重排 Tool Schema

在 System Prompt 前部加入随机时间或请求 ID

用新字符串替换历史，而不是追加新消息

模型或 Provider 路由变化

Compaction 重写前缀

Adapter 序列化在相同输入下不稳定

缓存命中率下降时，先比较两次最终 Wire Request 的前缀在哪里开始分叉，再回看 Provider 报告的命中数据。Session Event 有多少条，本身并不能说明缓存是否命中。

下一篇沿请求进入主循环：Agent Loop：Turn、Step、模型流与工具结果。

Agent Loop：Turn、Step、模型流与工具结果怎样闭环

返回 DeepSeek Harness 课程地图

DeepSeek Harness 的 Agent Loop（智能体循环）比一个简单的 while (toolCall) 多了几层边界，它会先把用户输入放进 Inbox（收件箱）。

系统再用 Turn（回合）表示一次外部任务的推进过程，并用 Step（步骤）标记其中的每次模型请求，同时把关键变化追加为 Session Event，为插入输入、取消、并行工具和恢复留下可核对的边界。

先理解 Turn 与 Step

```
Turn 1
  Step 1：用户消息 → 模型 → ToolCall
  工具执行 → ToolResult 放入 next-step Inbox
  Step 2：ToolResult → 模型 → 最终文本
Turn 1 结束
```

Turn 从一批需要唤醒 Agent 的输入开始，到完成、阻塞、错误或取消结束。

Step 对应一次模型请求和它产生的 Assistant Message；有工具调用时，工具结果推动下一 Step。

Inbox 区分 next-turn 与 next-step，所以 follow-up 和 steer 不会被混入错误边界。

Session Event 保存 turn/start、step/start、消息、工具调用、结果和结束原因。

第 1 站：外部输入先进入 Inbox

源码：查看 send() 与三种输入语义

```
followup(input: UserMessage): void {
  this.send(input, 'next-turn', true)
}

steer(input: UserMessage): void {
  this.send(input, 'next-step', true)
}

inject(input: UserMessage): void {
  this.send(input, 'next-step', false)
}
```

调用者：CLI、ACP、子任务或扩展表面向 Agent 提交新消息。

输入：UserMessage、目标边界和是否唤醒 Driver。

状态变化：消息被插入 Inbox；需要唤醒时启动或通知运行 Driver。

返回：这些方法没有业务返回值。

下一站：Driver 调用 turn()，在合适边界 claim Inbox 消息。

followup 会把消息明确排到下一 Turn，steer 会把消息送往下一 Step，并唤醒 Driver（驱动器），inject 则只把消息放进队列，不主动启动运行。三种入口不能混用。如果带唤醒语义的输入在取消后才到达，系统还会把它重新归入下一 Turn，避免它混入已经中止的活动。

Driver 只负责把 Turn 推进到收敛

kick() 的主体很短：只要 turn() 返回「还要继续」，它就开启下一 Turn。Driver 只负责推进，代码把复杂逻辑分别放在 Turn 和 Step 的边界上，因此 Driver 无需自己承担一个难以追踪的巨型循环。

源码：查看 Driver 循环

```
private async kick(): Promise<void> {
  try {
    while (await this.turn()) {}
  } catch (_error) {
    // failure is contained at the driver boundary
  } finally {
    ...
  }
}
```

更内层的代码会先把异常写成结构化事件，Driver 再在自己的边界上收束活动状态。失败不会卡在 running，因此 UI 既能看到具体错误，也能看到状态转为 idle。

第 2 站：Turn 追加边界事件并逐 Step 推进

源码：查看 turn() 主体

```
this.session.append('turn/start', { turn })

while (true) {
  const step = phase.step + 1
  const decision = await this.preStep(target, { turn, step })
  ...
  this.session.append('step/start', { turn, step })
  const stepEnd = await this.step(decision.assembly)
  this.session.append('step/end', { turn, step })
  ...
}

this.session.append('turn/end', { turn, reason: turnEnds! })
```

调用者：kick() 在 Driver 已占有 running phase 后调用。

输入：Inbox 中的消息、Session 派生历史、AbortSignal 和 Prompt Assembly 服务。

状态变化：追加 Turn/Step 边界与用户消息；更新 phase 的 turn、step 和结束原因。

返回：布尔值表示 Inbox 是否还有输入，需要继续下一 Turn。

下一站：每个 Step 调用模型流，随后可能调度工具。

preStep() 除了取出消息，还会组装 System Prompt、投影运行时 Context，并发出 waterfall 事件，让扩展有机会修改或拒绝当前 Step。当扩展返回 reject 时，Turn 会以 blocked 结束，如果初始消息在处理后已经为空，系统也不会再发起一次无意义的模型调用。

Step 从 Session 重新派生完整请求

每次请求都重新派生。系统会从 Session 历史出发，不会只沿用上一次模型返回的局部内容。构造新请求时，它需要把 System Prompt Assembly、Tools、模型路由和历史消息放在一起，然后冻结成本次请求。

第 3 站：流式 Chunk 先落事件，再合成 Assistant Message

源码：查看 step() 的模型流

```
const stream = preparedCall?.stream(request) ?? this.loopCtx.llm.stream(request)
for await (const chunk of stream) {
  chunkSeqs.push(
    this.session.append('assistant/chunk', { turn, step, chunk }).seq,
  )
  assembler.push(chunk)
}

const message = createAssistantMessage({
  content: assembler.blocks(),
  source: { provider: request.provider, model: request.model },
})
this.session.append('assistant/message', { turn, step, message, ... })
```

调用者：turn() 为本 Step 准备好 Prompt Assembly 后调用 step()。

输入：System Prompt、Session 派生消息、工具 Schema、Provider/Model 与 AbortSignal。

状态变化：每个 Chunk 先追加到 Session；Assembler 构造完整内容与 Usage；完成消息引用来源 Chunk 序号。

返回：Step 结束原因，或工具执行后返回 null 表示还需下一 Step。

下一站：若 Assistant Message 含 ToolCall，调用 executeToolCalls()。

如果模型流在中途取消，Assembler 会尽量用已收到的内容生成 interruptedBlocks()，并把这条不完整的 Assistant Message 标记为 interrupted。部分消息仍不算完成，保留它只是为了观察取消前发生了什么，评分时不能把它当成正常完成的消息。

模型流报告 error 或 aborted 后，agent/request-error waterfall 可以返回 retry，系统会留在同一个 Step 重新构建请求。只有没有获得 retry 决策时，它才抛出 LlmError，所以统计 Attempt（尝试）时不能根据 Step 数直接推算模型调用次数。

没有 ToolCall 才完成当前 Step 链

```
const toolCalls = message.content.filter(block => block.type === 'tool-call')
if (toolCalls.length === 0) return { kind: 'completed' }

const { concluded } = await executeToolCalls(...)
return concluded ? { kind: 'completed' } : null
```

源码：查看 Step 收敛判断

这段逻辑表明，模型输出文本并非当前 Step 链唯一的完成信号。工具结果可以直接声明 concluded，否则它们会进入新的上下文，Turn 随后再开启一个 Step，让模型读取工具的执行结果。

第 4 站：工具调用先解析，再按执行模式分组

源码：查看 executeToolCalls()

```
const planned = toolCalls.map(block => ({
  block,
  exec: {
    callId: block.id,
    name: block.name,
    arguments: parseArguments(block.arguments),
    agent,
```

```
      signal,
    },
  }))

  const mode = ctx.tools.executionMode(first.exec).kind
  const group = mode === 'parallel' ? planned.slice(next) : [first]
```

调用者：Step 检测到一个或多个 ToolCall 后调用。

输入：模型顺序中的 ToolCall 列表、Turn/Step、发起 Agent 和共享取消信号。

状态变化：参数被解析为对象或保留原始文本；工具根据实时 execution mode 形成并行组或独占屏障。

返回：是否有工具要求直接结束，以及已启动调用的上下文结果。

下一站：Scheduler 依次 prepare、dispatch、finish/finalize，并按模型顺序提交结果。

工具可以并行运行，但 Scheduler 仍会按模型给出的顺序提交 Policy、Result 和送往模型的 Context。提交不能跟着完成速度走，因此运行顺序和提交顺序要分开看，稳定的提交顺序既方便重放和解释，也能避免第二个工具因为先跑完就抢先改变后续语义。

第 5 站：Call 与 Result 用事件序号建立因果关系

源码：查看工具事件提交

```
  const event = session.append('tool/call', {
    turn, step, callId: block.id, name: block.name, arguments: block.arguments,
  })

  session.append('tool/result', {
    turn, step, message, ...
  }, { sourceEventSeqs: [callSeq] })
```

调用者：Scheduler 在调用开始和结果可提交时调用辅助函数。

输入：ToolCall、执行结果、Turn/Step 和对应 Call Event 序号。

状态变化：Session 先记录意图，再记录引用该意图的结果；UI 私有展示信息也可随 Result 保存。

返回：Call 辅助函数返回事件序号，Result 追加没有业务返回值。

下一站：Result Context 被放入 next-step Inbox，下一模型请求从 Session 派生它。

取消发生后，Scheduler 会 drain 已经开始的调用，并按既定顺序提交结果，尚未开始的调用则会收到「调度前已取消」的合成错误。如果是 Scheduler 内部失败，它会停止分派新调用，并排空已启动的调用，却不会为未启动的调用伪造结果。两条路径留下的事件不同，统计时也要分开。

用失败测试任务走一遍

用户要求修复运费边界：

1. followup() 把任务放入下一 Turn 并唤醒 Driver。
2. preStep() claim 消息，组装 Prompt、工具和运行时 Context。
3. Step 1 请求模型，Assistant Message 产生 Read 与测试工具调用。
4. Scheduler 可以并行执行只读调用，但按模型顺序追加 Result。
5. Result Context 进入下一 Step，模型再产生 Edit 和测试调用。

6 没有新 ToolCall 时，Step 返回 completed，Turn 写入结束事件。

7 独立 Eval 读取 Session 轨迹、工作区 Diff 和测试输出，不以 completed 代替正确性。

阅读这一循环时保留三个边界

turn/end completed 是 Harness 收敛，不是测试通过。

ToolResult 是执行报告，不保证副作用完整或可信，仍需环境核验。

Session Event 能重放已记录事实，但不等于自动恢复所有外部进程状态。

上一层配置怎样把 Prompt 和 Tools 送进循环，见Prompt、Context 与 Cache，而下一篇会继续追踪执行边界：工具、审批与 Sandbox。

工具、审批、Sandbox 与一次性提权

返回 DeepSeek Harness 课程地图

模型看得到某个工具，并不表示它能绕过策略直接执行。各层不能互相代替。DeepSeek Harness 把 Tool Schema、Runtime Pipeline、Approval（审批）和 Sandbox 分成几层，分别处理不同问题。Schema 告诉模型怎样提出调用，Runtime 随后校验参数、安排执行并统一结果格式，Approval 决定是否只放行这一次请求，Sandbox 则真正限制文件和进程能够触及的范围。

一次工具调用的安全链

```
模型 ToolCall
  → Agent Loop 记录 tool/call
  → ToolRuntime 解析可见定义与 Scoped Restriction
  → pre-execute / policy / approval
  → Sandbox 或其他执行器
  → post-execute / 结果规范化
  → Session 记录 tool/result
```

前一层放行后，调用仍要接受后一层检查。即使 Approval 允许某次调用使用范围更宽的 Sandbox，也只能证明用户同意了这次请求，真正开始执行时，操作系统内核、文件 ACL 或执行器仍可拒绝操作。

第 1 站：工具定义同时声明输入、输出和执行体

源码：查看 ToolDefinition

```
export interface ToolDefinition extends ToolSchema {
  readonly output: ToolOutputDefinition
  execute(args: unknown, exec: ToolRunContext): Promise<unknown>
  finalizeContent?(exec, result): ContentBlock[] | undefined
  timeoutMs?: number
  isConcurrencySafe?(args: unknown): boolean
}
```

调用者：文件、Shell、Todo、MCP Bridge 等插件向 ToolRuntime 注册定义。

输入：模型可见 Schema、规范输出 Schema、执行函数和可选并发/超时元数据。

状态变化：定义进入 Global 或 Agent Scope Registry，还没有执行。

返回：注册函数返回精确 disposer，用于卸载定义。

下一站：SystemPrompt 获取可见 Schema；模型提出调用后 Runtime 解析同一个 Scoped 定义。

工具要先返回可验证的 JSON Value，再由纯函数 render() 转成模型内容。这样一来，UI 展示、模型文本和结构化业务值不会全被塞进一个任意字符串，调用方也能分别验证和使用它们。

第 2 站：注册与 Scope Restriction 是不同操作

源码：查看注册和限制

```
register(definition: ToolDefinition): () => void {
  ...
  return this.layers.effect(
    this.ctx,
    layer => layer.tools.insert(name, definition),
  )
}
```

```
restrict(filter: ToolRestriction): () => void {
  const scope = scopeOf(this.ctx)
  if (scope === undefined) throw new Error(...)
  ...
}
```

调用者：Host 插件注册全局工具；Agent Preset 可以注册局部工具或限制全局集合。

输入：完整 ToolDefinition，或 allow/deny Restriction。

状态变化：Scoped Tool 可以遮蔽 Global；Restriction 只影响目标 Agent Scope。

返回：可撤销本次变更的 disposer。

下一站：Prompt Assembly 和执行解析都读取相同 Scoped 可见集合。

Runtime 禁止在 Root Context 上调用 restrict()，因为这会意外屏蔽所有 Agent。Restriction 为空时，调用同样会直接失败，避免系统把「配置最终展开为空」误认成一条有效策略。

并行默认是拒绝式选择

只有 isConcurrencySafe(args) 明确返回 true，工具才能与同批的其他调用重叠执行。如果工具没有声明该函数、函数抛出异常或返回的不是 true，又或者工具不可见、参数不合法，Runtime 都会把这次调用按 exclusive 处理。

源码：查看并发分类

这里采用 fail-closed 策略。读取文件一类工具可能适合并行，但工具只要会修改共享状态，就默认阻断同批调用与它重叠执行。并发不等于安全。并发安全声明只回答「能否与同批工具同时运行」，既不保证工具整体安全，也不会免除权限检查。

第 3 站：Approval 每次询问都写成一对 Session Event

源码：查看 Approval 请求

```
const id = ApprovalRequestId(randomUUID())
session.append('approval/asked', {
  id, toolName: req.toolName, callId: req.callId, reason: req.reason,
})
const outcome = await this.decide(req, session)
session.append('approval/decided', { id, outcome })
return outcome
```

调用者：Sandbox 提权或其他需要人类决定的策略层。

输入：发起 Agent、Tool 名、Call ID、理由和取消信号。

状态变化：在同一开放 Turn 内先记 asked，再记 decided；两者用 Approval ID 关联。

返回：allowed-once、rejected、cancelled 或 unavailable。

下一站：策略层只在 allowed-once 时继续本次执行。

两条记录必须配对。Approval 要求当前 Turn 仍处于开放状态，因为系统以 Turn 划分持久化提交和重放范围。如果询问事件落在两个 Turn 之间，崩溃恢复时就可能把它当作不属于任何 Turn 的尾部记录并丢弃。

ask 策略会把问题交给已经接入的 Answerer，没有 Answerer 时便返回 unavailable。never 则由服务内部直接拒绝，之后注册的 Listener 也无法改变这个决定。因此，Headless/CI 环境应该采用结果确定的 never，不要指望无人回答的 Prompt 会自动安全收尾。

Sandbox Mode 与 Approval Policy 不是同一个轴

Sandbox Mode 决定调用在什么文件/进程边界运行，如 read-only、workspace-write、danger-full-access。

Approval Policy 决定遇到需要授权的请求时能否询问。

一次调用可以在 workspace-write 下直接运行，也可以请求更宽模式；请求更宽必须先获 Approval。

第 4 站：提权参数必须成对并严格变宽

源码：查看提权参数与 Mode 阶梯

```
export const WIDER_MODES = {
  'read-only': ['workspace-write', 'danger-full-access'],
  'workspace-write': ['danger-full-access'],
}

if (sandboxPermissions !== undefined && justification === undefined) {
  throw new Error('... requires a justification')
}
```

调用者：Shell 和文件工具在解析 sandbox_permissions 时调用。

输入：目标 Mode 和用户可读理由。

状态变化：这里只校验请求形状，不改变会话默认 Mode。

返回：合法时无返回；缺字段、空理由或错误配对直接抛错。

下一站：approveEscalation() 对照本次调用的有效 Mode 并发起 Approval。

系统不能只根据部署时的默认 Mode 缩减目标枚举，因为 Session 运行期间可能切换到范围更窄的 Mode。执行每次调用时，都要拿它当时生效的 Mode 作比较，才能判断申请的目标是否确实更宽。

第 5 站：授权发生在任何执行之前

源码：查看 approveEscalation()

```
if (!(WIDER_MODES[effectiveMode] ?? []).includes(mode as SandboxMode)) {
  throw new Error('... is not strictly wider ...')
}
if (approval.approver === undefined) throw new Error('... no approval service ...')
if (approval.agent === undefined) throw new Error('... no agent ...')

const outcome = await approval.approver.request({ ... })
switch (outcome) {
  case 'allowed-once': return mode as SandboxMode
  default: throw new Error(...)
}
```

调用者：具备提权字段的 Tool，在启动命令或文件变更前调用。

输入：目标 Mode、有效 Mode、理由、Agent、Call ID 和 Approval Service。

状态变化：Approval 轨迹写入 Session；不会改写后续调用的默认 Mode。

返回：只在 allowed-once 时返回本调用使用的 Mode。

下一站：执行器以该 Mode 运行这一次操作。

授权不会永久生效。申请同级或范围更窄的 Mode 时，系统不会询问用户。只要 Approval Service 或 Agent 缺失，或者用户拒绝、取消、没有可用回答通道，本次执行就会按失败处理。即使获得授权，也只对当前调用生效。

ToolRuntime 如何保证取消后不遗留活动

取消仍需工具配合。Runtime 的 dispatchToolBody() 会合并调用方传入的信号与 Wrapper 替换后的信号。工具体一旦开始运行，收到取消请求后，Runtime 不会丢下仍在执行的 Promise，而会等待工具停止活动，再把结果统一标记为 aborted。

源码：查看 Tool Body 分派

同一进程内运行的 JavaScript 无法由 Runtime 强行终止，因此 Tool 必须主动检查 exec.signal，并继续把信号传给子进程、网络请求或文件操作。即使 Tool 声明了 timeout，也要等相应的 Policy Wrapper 接入，而且 Tool 本身配合取消，超时限制才真正生效。

三个平台的 Sandbox 不能假设完全同构

Linux 可以使用 Bubblewrap/Landlock 等 Provider，macOS 常用 Seatbelt，Windows 则可能依赖 ACL 或受限进程链。统一 Mode 只是 Harness 对外提供的行为契约，各平台未必具备完全一致的内核能力，因此部署验收至少要测试：

1. read-only 下工作区写入确实失败；
2. workspace-write 不能越过工作区边界；
3. 提权必须先出现 Approval Event；
4. 拒绝后没有子进程或文件副作用；
5. Sandbox 不可用时是否失败关闭，而非静默裸跑；
6. 符号链接、盘符、网络和子进程继承是否符合平台策略。

下一篇：Session、Compaction 与冷恢复。

Session、模型可见 Surface、Compaction 与冷恢复

返回 DeepSeek Harness 课程地图

DeepSeek Harness 的 Session 同时保存完整的追加式 Event Log，并维护一份可以送给模型的 Surface（接口面）。两种视图各有用途。

Compaction（上下文压缩）不会删除旧 Event，只会在 Surface 中用一条摘要消息替换一段旧节点。恢复 Session 时，系统重新播放 Event 及其 Surface 操作，便能再次派生出模型历史。

Event Log 与 Surface 为什么分开

Event Log 除了记录模型消息，还要保存 Turn/Step 边界、流式 Chunk、Approval、Request Header、Compaction 生命周期和错误等运行事实，而模型每次请求只需要读取其中一部分。

```
追加式 Event Log
  ├── turn/start、assistant/chunk、approval/* 等审计事实
  └── user/message、assistant/message、tool/result 等 Surface 节点
     → append 或 replace 操作
     → deriveMessages()
     → 下一次模型请求历史
```

把两种视图分开后，Harness 既能保存完整、可审计的运行事实，也能缩短模型输入，因此不必为了减少 Token 而真正擦除旧记录。

第 1 站：恢复 Seed 先验证，再进入 Session

源码：查看 Session 构造与恢复

```
static fromRestore(id, seed, header): Session {
  return new Session(id, seed, header, 'restore')
}

for (const [index, source] of seed.entries()) {
  assertSessionEventEnvelope(snapshot, index)
  if (snapshot.seq !== index) throw new Error(...)
  this.surfaceManager.validateNext(snapshot)
  this.log.push(...)
}
```

调用者：持久化层加载 JSONL 或其他后端记录后恢复 Session。

输入：Session ID、完整 Event Seed 和持久化 Header。

状态变化：验证格式版本、序号连续性、JSON 可表示性和 Surface 转换；全部合格后才接管记录。

返回：Detached Session，之后由 SessionStore 进入运行上下文。

下一站：Agent Loop 从恢复后的 Surface 派生消息，Request Header 恢复模型请求边界。

系统会逐条检查 Seed，并采用与 live append 相同的 Surface 校验规则，从而避免 Session 在内存中可以恢复、后端却再也无法保存的分叉状态。序号也必须从 0 开始连续排列，因为许多因果引用会直接使用 Event Seq 定位来源。

第 2 站：Append 是内存提交点，不等待磁盘 I/O

源码：查看 Session.append()

```
const event = deepFreeze({
```

```
  type,
  seq: this.log.length,
  time: Date.now(),
  data: dataSnapshot,
  ...surfaceMetadataSnapshot,
})
this.surfaceManager.validateNext(event)
this.log.push(event)
invokeContainedSessionObservers(...)
```

调用者：Agent Loop、ToolRuntime、Approval、Compaction 和其他插件。

输入：类型化 Event Data，以及消息类 Event 必需的 Surface 操作与来源序号。

状态变化：数据被无损 JSON Snapshot、冻结、分配 Seq、验证 Surface 后追加到内存 Log。

返回：实际进入 Log 的 Event；调用方后续修改原对象不会影响它。

下一站：Observer 异步缓冲持久化；需要耐久点的调用方显式 session.flush()。

系统会隔离 Observer 的失败，所以 Observer 无法回滚已经提交到内存的 Event。两次成功含义不同。append 成功只说明 live Session 接受了该事件，flush 成功才说明已配置的后端写到了耐久检查点。

Surface 操作如何表达压缩

消息类 Event 必须携带：

surfaceOp: 'append'：把自己加到模型历史末尾；

surfaceOp: { op:'replace', start, end }：用自己替换当前 Surface 中一段连续范围；

sourceEventSeqs：列出它派生或替换所依据的旧 Event。

非消息 Event 不能伪装成 Surface 节点。Replace 操作既要命中当前仍存在的节点，也要完整覆盖指定范围，而系统对 ToolResult 的特殊重写还会施加更严格的约束。

第 3 站：模型历史从 Surface 增量派生

源码：查看 deriveMessages()

```
const nodes = surface.nodes
const generation = surface.replaceGeneration
if (generation !== this.derivedGeneration) {
  this.derived = []
  this.derivedNodes = 0
}
for (const seq of nodes.slice(this.derivedNodes)) {
  const msg = this.deriveEventMessage(this.log[seq]!)
  if (msg) this.derived.push(msg)
}
return [...this.derived]
```

调用者：Agent Loop 构建每次模型请求。

输入：当前 Surface 节点序列与完整 Log。

状态变化：普通 append 只投影新增节点；replace generation 变化时重建派生缓存。

返回：新的数组快照，内部 Message 对象仍是共享且深冻结的。

下一站：buildRequest() 把这些消息与 System、Tools 一起冻结为 GenerateOptions。

Surface 并非「另一份可随意修改的消息数组」，它是系统按照确定性规则折叠 Event Log 后得到的投影。因此，无论恢复 Session、构建模型请求还是开展外部分析，都应该复用同一套 Fold 规则。

Compaction 是一笔带锁的 Surface 事务

自动 Compaction 会在仍然开放的 Turn 内运行，手动 Compaction 则只能等 Session Idle 后启动。系统先从 Surface 中选择一段边界完整的范围，确保不会拆开 ToolCall 与 ToolResult 的配对关系，再写入 compaction/start 作为耐久锁。摘要异步生成后，系统还要确认 Surface 没有发生变化，才会提交摘要和 Replace 操作，并在末尾写入 compaction/end。

第 4 站：Start/End 把异步摘要包进事务

源码：查看 Compaction 主事务

```
const startEvent = session.append('compaction/start', lifecycle)
try {
  const prepared = prepareCompaction(...)
  const summarized = await summarizeCompaction(...)
  assertStable(...)
  const pending = commitCompactionBody(...)
  const endEvent = session.append('compaction/end', lifecycle)
  result = completeCompaction(pending, endEvent)
} catch (error) {
  session.append('compaction/end', { ...lifecycle, error: errorChain(error) })
}
```

调用者：自动压力策略或手动 Compact 命令。

输入：选中 Seq 范围、Agent、Summarizer、稳定性规则、可选 Flush 和 AbortSignal。

状态变化：写 Start Lock，生成摘要，验证并提交 Surface Replace，再写 End；失败也尝试关闭事务。

返回：成功的 CompactionResult；摘要、提交或持久化失败用不同错误表达。

下一站：下一次 deriveMessages() 看到新的 replace generation 并重建模型历史。

如果系统连 End 都写入失败，Log 中就会留下没有配对的 Start。恢复流程据此可以发现「上次 Compaction 没有正常收尾」，并阻止系统在状态未明时并发启动第二次压缩。

第 5 站：摘要消息替换旧 Surface，但保留因果

源码：查看 Compaction 提交

```
const summaryEvent = session.append('compaction/summary', {
  summary,
  shadowedRange: { start, end },
  shadowedSeqs: [...shadowedSeqs],
  provider,
  model,
  usage,
})

session.append('user/message', checkpointMessage, {
  surfaceOp: { op: 'replace', start, end },
  sourceEventSeqs: [startEvent.seq, summaryEvent.seq, ...shadowedSeqs],
})
```

调用者：Compaction 事务通过稳定性检查后提交。

输入：摘要、被遮蔽范围、来源节点、模型信息和 Checkpoint Message。

状态变化：完整 Log 追加摘要事实；模型 Surface 用一条 Checkpoint 替换旧范围。

返回：待完成的 Compaction 结果。

下一站：后续模型请求只看 Checkpoint 与保留的尾部消息，审计器仍可访问旧 Event。

压缩只改变模型看到的 Surface，旧 Event 依然保留在 Log 中。摘要仍可能遗漏信息。因此，Eval 应该对照 Compaction 前后的结果，检查关键约束、未完成任务、文件路径和 Tool 结果能否继续恢复。

冷恢复还需要外部世界检查

Session 可以恢复消息、请求 Header、Approval Policy 和 Compaction Surface，但重新播放 Event 时，下面这些外部状态不会随之自动恢复：

已退出子进程的内存；

已过期的网络连接和 MCP 会话；

工作区在 Session 关闭后发生的外部修改；

临时文件与凭据有效期；

当时的 Provider 模型版本。

外部状态要另行核对。恢复 Session 后，你还要重新核对工作区基线、当前可用工具和模型路由，再决定是否继续未完成的动作。Event 重放只能恢复 Harness 保存的状态，无法把外部世界还原到记录产生的时刻。

下一篇：编排、子 Agent、Workflow 与 Code Mode。

子 Agent、Workflow 与 Code Mode：三种不同的编排跨度

返回 DeepSeek Harness 课程地图

前五篇一直在回答「一个 Agent 怎样完成一个 Turn」。当任务规模扩大后，Harness 还要区分三种需求：要不要创建另一个 Agent，要不要用确定性脚本组织多个 Agent，以及是否只需在一次工具调用中批量使用现有工具。

三者不能相互替换。DeepSeek Harness 分别用 Subagent、Workflow 和 Code Mode（代码模式）处理这三类需求，它们都涉及「编排」，但各自改变的运行边界不同。

```
需要独立上下文和独立 Session？ —— 是 → Subagent
            |
           否
            ↓
需要一段确定性脚本组织多个子运行？ —— 是 → Workflow
            |
           否
            ↓
需要在一次模型可见调用中批量使用现有工具？ → Code Mode
```

先分清四个容易混淆的对象

对象	谁做决策	是否创建新 Agent	模型可见的入口	主要边界
普通 Tool	当前 Agent	否	一个工具 Schema	单次副作用
Subagent	父 Agent 或宿主	是	委派工具或类型化接口	父子 Session、深度、取消
Workflow	脚本作者	通常会	Workflow 工具或服务	脚本资源上限、子运行配对
Code Mode	当前模型编写的小程序	否	run_code	仍受原 ToolRuntime 约束

Subagent 为一项子任务建立独立的上下文、生命周期和结果通道，但这种机制本身不会自动提高模型能力。Workflow 把并发、流水线和结果聚合交给可重复执行的脚本，不再要求模型每次都用自然语言决定编排步骤。Code Mode 仍然使用当前 Agent，只是把多次工具调用集中到一个模型可见的 run_code 中，从而减少模型与 Harness 之间的往返。

Subagent：一次委派是一条完整子运行

执行一次进程内委派时，系统先计算子层级并创建新 Session，再把父 Agent 允许继承的项目配置、Persona、工具过滤规则和结构化输出约束装入子上下文。spawn 不携带历史 Seed，fork 则会带入一段已经完成 Turn 的 Seed，不过激活边界之后产生的新事件仍归当前子运行所有。

第 1 站：创建子 Agent 之前冻结继承关系

源码：查看 startInProcessRun()

```
const childDepth = resolveChildDepth(parent, request.maxDepth)
const activationBoundary = seed?.length ?? 0
const inherited = captureDelegatedPolicyOverrides(parent)
```

```
const handle = await parent.ctx.agents.create({
  sessionId: childId,
  meta: childSessionMeta(parent, childDepth, activationBoundary),
  ...seed !== undefined ? { seed } : {},
  setup,
})
```

调用者：Subagent 的 spawn/fork Provider。

输入：父 Agent、Prompt、最大深度、可选 Seed、Persona、Tool Filter 与输出 Schema。

状态变化：检查深度；在第一次异步等待前捕获父端策略；创建并发布独立子 Agent。

返回：拥有子 ID、结果 Promise 和 dispose() 的 SubagentRun。

下一站：Driver 把 Prompt 投递到子 Agent Inbox，等待它回到 Idle。

系统必须「在第一次 await 前捕获」继承值，因为创建子 Agent 期间，父端可能已经切换权限策略，而正在启动的委派不应悄悄读到之后才生效的值。发布改变取消路径。Publication 构成另一条边界：发布前取消，说明创建事务尚未交出子 Agent，发布后再取消，就必须通过已经返回的 Run 管理生命周期，不能把确实存在的子 Agent 重新解释成「启动失败」。

第 2 站：结果、取消与释放是三件事

源码：查看已发布子运行的生命周期

```
child.followup(createUserMessage({ content: prompt, source: { kind: 'user' } }))
await child.whenIdle()
return readResult(child, boundary, flags.cancelled, ...)

async dispose() {
  flags.cancelled = true
  const settlements = await Promise.allSettled([handle.dispose(), result])
}
```

调用者：拿到 SubagentRun 的委派工具或 Workflow。

输入：已发布的 Agent Handle、取消信号、Prompt 和激活边界。

状态变化：投递一次用户消息；父取消传播到 child.cancel()；释放时等待 Agent Handle 与结果通道都收敛。

返回：output、可选 structured 和 stopReason。

下一站：父 Agent 或 Workflow 决定怎样消费子结果。

源码会保留取消或截断前已经提交的最后一段 Assistant 输出，因此 output 非空并不能证明 stopReason === 'completed'。文本完整不等于成功。即使文本看起来完整，也可能来自未正常完成的子运行。如果结构化 Schema 没有捕获到值，就不能判定这次结构化委派成功。

Workflow：脚本拥有编排，事件只负责观察

Workflow Service 接收脚本、参数、父 Agent 和取消信号后，会返回一个 live run，脚本可以在该运行中通过 agent() 启动多个子运行。phase() 和 log() 只向外提供观察信息，不会改变脚本的执行语义。

第 3 站：每个开始事件都有对应结束事件

源码：查看 Workflow 生命周期事件

```
'workflow/start'(info)
'workflow/agent-start'(info, agent)
```

```
'workflow/agent-end'(info, agent)
'workflow/end'(info, result)
```

调用者：Workflow Runtime；Observer 订阅这些事件。

输入：Run 身份、Phase、日志或带序号的子 Agent 信息。

状态变化：不控制脚本，只向外发出可配对的生命周期事实。

返回：事件回调无业务结果。

下一站：运行结果从 WorkflowRun.result 收敛，而不是从 Observer 猜测。

系统通过 agent.seq 配对 agent-start 与 agent-end。如果子 Agent 在发布前启动失败，这两个事件都不会发出。一旦子 Agent 已经发布，无论之后正常结束、失败还是取消，都必须发出对应的 agent-end，因此观察者看到 start 时只能登记一项待结算运行，不能直接把它计为完成。

Workflow 会区分致命编排错误与普通子任务失败。脚本无法解析、参数非法、超出资源上限或结果不可序列化等 WorkflowError 会终止整个脚本，普通子运行失败则可以由组合器把对应项映射为 null。这样一来，系统不会把脚本中的拼写错误伪装成「某个子任务没有答案」。

第 4 站：观察器不能接管运行控制

源码：查看 WorkflowEngine

```
abstract start(request: WorkflowStartRequest): WorkflowRun
```

```
for (const callback of this.ctx.events.dispatch('emit', [name, ...args])) {
  try {
    void Promise.resolve(callback(...args)).catch(logWarning)
  } catch (error) {
    logWarning(error)
  }
}
```

调用者：Workflow 工具或宿主调用 start()；Engine 内部发送事件。

输入：脚本请求或一个生命周期事件。

状态变化：start() 建立运行；事件监听器失败被记录并隔离。

返回：live WorkflowRun，其 result 负责最终结算。

下一站：调用方等待 Result，随后释放 Run 与子资源。

如果遥测监听器抛错就能终止脚本，观察系统便会意外取得运行控制权。观察者不能控制运行。如果监听器还能修改结果，那么重放同一次运行时，结果将取决于当时加载了哪些插件。系统会隔离监听器失败，确保监听器只能观察事件，无法接管运行。

Code Mode：外面只有 run_code，里面仍是受控工具调用

Code Mode 会根据 TypeScript 或 Python Runtime 生成对应语言的 run_code Schema。模型在其中编写异步函数体，并通过 tools.name(args) 调用当前 Agent 原本就能看到的工具。

第 5 站：内部调用复用 Registry 的执行与并发规则

源码：查看 createRunCodeTool() 的调度约束

```
const runController = new AbortController()
```

```
let dispatches = 0

interface PendingDispatch {
  start(): Promise<void>
  classify(): 'parallel' | 'exclusive'
  commit(): Promise<void>
  flight: Promise<void>
}
```

调用者：当前 Agent 的 ToolRuntime 执行 run_code。

输入：程序正文、显示用 Description、当前 Agent 和外层取消信号。

状态变化：建立本次运行的取消域；按普通工具相同的 parallel/exclusive 规则排队、执行和提交内部调用。

返回：程序日志与 JSON 可表示的返回值。

下一站：外层 Tool Result 进入模型历史；内部 Dispatch Log 留给追踪与重建。

并发只发生在工具 Body 阶段，Guard、Pre/Post Execute、开始事件和结算事件仍要经过同一条有序通道。遇到独占工具时，调度器先等待并发池排空，并持续阻断后续调用，直到独占工具提交完成。因此，即使模型把三次 Bash 调用放进 Promise.all，也无法绕过 Registry 对独占执行的判定。

第 6 站：程序只能绑定当前 Agent 真正可见的工具

源码：查看 Runtime 绑定与清理

```
for (const schema of registry.schemas(exec.agent)) {
  if (schema.name === RUN_CODE_NAME) continue
  Object.defineProperty(functions, schema.name, { value: binding(schema.name) })
}

result = await runtime.run({ program: args.code, bindings, signal })
runController.abort('run_code settled')
await drainDispatches()
```

调用者：run_code.execute()。

输入：调用 Agent 的 Scoped Tool View 与 Code Runtime。

状态变化：只绑定该 Agent 可见的工具；结束时取消未完成调用并排空已启动 Dispatch。

返回：整理后的日志和 Result；运行错误成为 CodeRunFailedError。

下一站：Agent 根据外层 Tool Result 继续 Step。

Code Mode 不会让任意代码获得宿主的全部权限。权限并未随之扩大。实际隔离强度仍取决于 Code Runtime，但系统会按照当前 Agent 的可见集重新解析每项工具能力，并让内部调用经过同一条执行管线。

用一个代码审查任务理解三者

假设要检查三个包：

1. 若每个包都需要独立长上下文、独立工具范围和单独结果，创建三个 Subagent。
2. 若三个检查要限制并发、固定输出顺序，并让一个失败不打乱其他项，用 Workflow 组织三个 Subagent。
3. 若只是读取三个小文件并统计共同模式，当前 Agent 可在 run_code 内并发调用 Read；无需创建三个 Agent。

无论选择哪种方式，最后都要分别检查每次运行的停止原因和产物内容。父 Agent 写出「子任务完成」，只说明它消费过一次结果，并不会改变子 Session 记录的原始终态。

下一篇：产品表面、反馈与评测接入。

同一 Agent 核心如何变成 Headless、ACP 与反馈数据

返回 DeepSeek Harness 课程地图

Agent Loop 解释内部怎样推进任务，产品表面则要继续回答外部怎样创建 Session、提交任务和接收输出，以及审批或取消怎样传回运行中的流程。DeepSeek Harness 可以把同一套 Agent、Session、Tools 和 Provider 核心接到 Headless（无头模式）、ACP（Agent 客户端协议）、Web 或 SDK，但每个入口仍会按自己的协议交付结果。

表面适配器具体要翻译什么

外部请求进入 Agent 后，表面适配器要把协议对象转换成内部对象，等 Agent 产出结果，再把内部状态转换成客户端认识的形式。这个往返过程至少包含五类翻译：

外部 Session 身份如何映射到内部 SessionId；

外部 Prompt 何时真正进入 Agent Inbox；

内部事件中哪些内容可以投影给客户端；

内部 TurnEndReason 如何变成协议 Stop Reason 或进程退出码；

外部取消与权限响应如何回到正在运行的 Agent。

共享 Agent Core 能减少重复代码，不过数据一旦跨过协议边界，适配器仍可能舍弃部分信息。能力边界也可能改变。

Headless：一次任务、一次 Session、一个退出码

第 1 站：Headless 等待完整装配后才创建 Agent

源码：查看一次性运行入口

```
await ctx.get('loader')?.await()
const { agent } = await agents.create({
  sessionId: SessionId(`session-${randomUUID()}`),
  meta: { cwd: process.cwd() },
  agentOptions: { provider: selection.provider, model: selection.model },
})

agent.followup(createUserMessage({ content: [{ type: 'text', text: task }] }))
await agent.whenIdle()
await sessions.flush(agent.session)
```

调用者：Headless Bundle 的 apply()。

输入：单次任务文本、默认模型选择和进程 I/O。

状态变化：创建新 Agent，投递任务，等待 Idle，并把 Session 刷到持久化后端。

返回：Assistant 文本写到 stdout；停止原因映射为退出码。

下一站：Shell、CI 或其他宿主读取 stdout、stderr 和 Exit Code。

源码把 completed 映射为 0，把其他停止原因映射为 1，但 completed 只说明 Agent Loop 按内部规则停止，无法证明仓库中的测试已经修好。因此，Headless 承担自动修复任务时，宿主必须等进程退出，再另行执行验证命令。

ACP：一个协议请求跨过多个异步边界

ACP 表面会为协议 Session 找到对应的内部 Agent，同时保证同一 Session 最多只有一个 Prompt in flight。收到请求不代表 Prompt 已经开始运行，因为系统还要转换附件、处理取消竞争，并经过 Agent Inbox 接纳和 Turn Claim。请求仍可能失败。

第 2 站：先预留 Prompt 槽位，再做异步接纳

源码：查看 ACP prompt()

```
if (record.inflight !== undefined) {
  throw invalidParams('a prompt is already in flight for this session')
}
record.inflight = inflight

const content = await admitAcpPrompt(...)
admissionController.signal.throwIfAborted()
record.agent.followup(message)
```

调用者：ACP 客户端的 session/prompt 请求处理器。

输入：协议 Session ID、文本或附件 Prompt。

状态变化：同步占用 in-flight 槽；转换内容；在最终取消检查后投递 Inbox。

返回：等待内部运行和输出投影都收敛后的 ACP Stop Reason。

下一站：客户端决定继续 Prompt、取消或关闭 Session。

代码必须在第一次 await 之前预留槽位，否则两个并发请求都可能读到「当前没有 Prompt」，随后一起进入同一个 Agent。处理取消时，桥接层也要先判断消息已经走到哪一步：消息仍在接纳附件时只取消 Admission，等它进入 Inbox 后，桥接层才能连同 Agent Work 一并取消。

第 3 站：协议只发送已提交的 Assistant 消息

源码：查看 Session Event 到 ACP Update 的投影

```
if (event.type === 'assistant/message') {
  for (const block of event.data.message.content) {
    const content = await assistantBlockToAcp(ctx, block)
    await notify({ update: { sessionUpdate: 'agent_message_chunk', content } })
  }
}
```

调用者：ACP Bridge 对 session/event 的监听器。

输入：内部 Session Event。

状态变化：每个 Session 用 Promise Chain 保证异步附件转换仍按原顺序投递。

返回：ACP session/update 通知。

下一站：Prompt 结算还要等待这条输出链归于静止。

Raw Chunk、Reasoning、Tool、Plan 和 Retry Marker 都不会进入自动化协议，因此客户端只能看到经过主动收窄的内容，无法据此获得完整 Trace。调试工具如果只用 ACP 输出重建执行过程，就会漏掉解释事件因果关系的关键信

息。

第 4 站：权限响应只产生一次性决定

源码：查看 ACP Permission 映射

```
options: [
  { optionId: 'allow-once', kind: 'allow_once' },
  { optionId: 'reject-once', kind: 'reject_once' },
]
```

调用者：内部 Approval Chain 遇到需要客户端决策的 Tool Call。

输入：内部 Request、Session ID 和 Call ID。

状态变化：桥接一次协议请求，不写入永久授权。

返回：allowed-once、rejected 或 cancelled。

下一站：ToolRuntime 继续执行、拒绝或终止该调用。

即使未知客户端返回了相似字符串，桥接层也不会据此推断用户给出了耐久授权。这里除了要映射字段，协议适配器还必须按保守规则限制能力，避免一次决定意外扩大后续权限。

Feedback：它是带身份的用户信号，不是正确答案

DeepSeek Harness 同时接收消息级和 Session 级 Feedback（反馈）。消息级接口先用 SessionId + MessageId 找到一条 Assistant 消息，每次写入后再返回一个不透明 Revision（修订版）。界面以后替换或删除反馈时，必须带回自己读到的 Revision，存储层才能执行 Compare-and-Set，避免两个界面悄悄覆盖彼此的修改。

第 5 站：消息反馈用 Revision 处理并发修改

源码：查看消息反馈公共类型

```
interface PutMessageFeedbackRequest {
  sessionId: SessionId
  messageId: MessageId
  value: MessageFeedbackValue
  expectedRevision?: MessageFeedbackRevision
}
```

调用者：Web 或 Remote Client 的反馈操作。

输入：Session、Message、反馈值与可选预期 Revision。

状态变化：创建或条件替换一个反馈项。

返回：新值和新的不透明 Revision，或冲突错误。

下一站：UI 刷新最新值；数据管线按版本消费反馈。

Session 级 /feedback 不处理这种并发更新，它只检查文本是否为空，随后把 feedback/record 追加到 Event Log。

源码：查看 Session Feedback 写入

```
const normalized = text.trim()
if (normalized.length === 0) throw new TypeError(...)
session.append('feedback/record', { text: normalized })
```

反馈只记录某个用户对某条消息或某个 Session 表达了什么，而界面位置、用户目标、情绪、选择偏差和误操作都可能影响这次表达。它不能直接充当正确答案。若要把反馈用于训练或评测，数据管线还得明确怎样采样、标签代表什么、冲突如何处理，并预先规定权重和独立留出集。

Eval 应接在哪一层

Eval（评测）不应只读取某个界面显示的「已完成」，更可靠的输入是一次运行留下的 Artifact，其中要保存固定输入、Harness 配置、Session/Trace、工作区变化、停止原因和成本。

Evaluator（评估器）位于运行流程之外，它读完这些材料后，再执行确定性检查或模型评分。

```
任务输入 + 锁定配置
        ↓
DeepSeek Harness 运行
        ↓
Session / Trace / 文件差异 / Stop Reason
        ↓
独立 Evaluator
        ↓
Score + 失败原因
```

评分工作放在运行流程之外以后，Headless Exit Code、ACP end_turn、Assistant 自述和用户 Feedback 都可以成为特征或证据，但其中任何一项都不能单独决定最终分数。

下一篇：自验证机制与证据边界。

如何阅读 DeepSeek Harness 的测试、不变量与证据边界

返回 DeepSeek Harness 课程地图

阅读复杂 Harness 源码时，测试通常比 README 更接近真实约束，但只说「仓库里有测试」，范围还是太宽。每层证据都有边界。类型检查、单元测试、运行时不变量、端到端场景和真实模型评测各自观察不同对象，因此只能支持不同范围的结论。

阅读时不妨少统计测试数量，多追问每个设计结论究竟由哪一层证据支撑。

五层证据各回答什么

层次	能直接检查	常见盲区
类型与 Schema	字段、联合类型、输入形状	运行顺序与业务语义
单元测试	一个模块在构造输入下的行为	真实装配与外部系统
集成 / 端到端测试	多模块或一个产品路径	未覆盖平台、长期运行
运行时不变量	一次运行中事件之间的关系	没注册或没触发的路径
独立 Eval	固定任务与评分规则下的结果	分布之外的任务与部署风险

例如，「Tool Result 必须引用已存在的 Tool Call」涉及两个事件之间的关系，单个字段无法表达，因此更适合交给运行时检查。至于「Headless 能打印答案」，端到端测试就足以核对。若要确认修复后的项目确实通过测试，还必须在 Harness 外部重新执行目标仓库的测试命令。

多层证据可以共同支持一个结论，但每一层只能证明自己观察到的部分，因此引用测试前要先看它实际构造了什么输入，又检查了什么结果。

运行时不变量：由包声明自己的跨事件关系

DeepSeek Harness 提供一个 Registry（注册表），各包通过自己的 Companion 向它登记动态检查。检查不会自动发生。Registry 只管理启用开关、包名筛选、重复注册和生命周期，具体检查哪些事件关系，仍由拥有这些关系的包决定。

这样分工以后，Registry 可以统一管理检查器怎样启动和停止，却不必理解每个业务包发出的事件。读源码时不要停在 Registry，还要进入具体 Companion，确认它到底安装了哪些监听器。

第 1 站：错误必须标明哪个包违反了什么约束

源码：查看 InvariantError

```
export class InvariantError extends Error {
  readonly code = 'INVARIANT'
  constructor(packageName: string, message: string) {
    super(`invariant violated by "${packageName}": ${message}`)
  }
}
```

调用者：包自有的 Invariant Installer 通过注入的 fail() 报错。

输入：注册包名和被破坏的关系描述。

状态变化：中断当前检查路径，不修改业务状态来「自动修复」。

返回：稳定错误码、包归属与可读信息。

下一站：测试或运行宿主决定终止、记录还是隔离该路径。

错误归属很重要。如果错误带有包归属，维护者就能把失败直接交给定义该契约的模块，而不会因为某个下游包触发了事件，就误判它应该负责。统一抛出 AssertionError 无法提供这层定位信息。

第 2 站：注册、筛选与生命周期由统一服务管理

源码：查看 InvariantRegistry.register()

```
if (!this.selected(packageName)) return unregister
const child = ctx.plugin(childCtx => installer(childCtx, fail))
await child
return async () => {
  await child.dispose()
  registrations.delete(packageName)
}
```

调用者：每个包的 invariant.ts Companion。

输入：完整包名和一个在子 Context 内安装监听器的函数。

状态变化：预留包名；若被选中则启动独立 Child Fiber；释放时卸载监听器并清除注册。

返回：与 Context Effect 绑定的 Disposer。

下一站：业务事件触发包自有检查。

Allowlist/Blocklist 一旦排除某个包，该包登记的检查就不会运行，没有 Companion 的包也不会由「通用服务」代为检查。因此，一次运行没有出现 InvariantError，只能说明已经加载、选中并触发的检查没有报告违规。

测试宿主如何避免业务代码跑在检查之前

DeepSeek Harness 的测试设置会根据测试文件找到对应包。普通场景只加载该包的 Invariant Companion，专门检查拓扑的测试才加载全部 Companion。在业务代码开始装配前，测试 Root Plugin 还要等待 Invariant Service 和 Companion 全部启动。

Readiness Barrier 关注事件发生顺序，不关注等待时长，业务事件必须等检查器开始监听后才能发生。否则，即使测试通过，也可能只是违规事件先于检查器到达。这个顺序不能颠倒。

第 3 站：按测试拥有者选择 Companion

源码：查看测试 Companion 选择

```
const owner = normalized.match(/\/packages\/([^/]+)\/([^/]+)\/tests\//)
const companionPath = `../packages/${owner[1]}/${owner[2]}/src/invariant.ts`
if (testInvariantCompanions[companionPath] === undefined) {
  throw new Error(`test invariants: package test has no companion ...`)
}
```

调用者：Vitest 全局设置建立测试 Invariant Host。

输入：当前 Test Path 与 import.meta.glob 发现的 Companion 表。

状态变化：选择本包检查，或在专门测试中选择全部检查。

返回：需要挂载的 Companion Path 列表。

下一站：Host 启动 Service、Companion 和业务 Root 的 Readiness Barrier。

这种做法兼顾了运行速度与检查范围：普通包测试不必导入整个 Monorepo，但某个包只要缺少约定的 Companion，对应测试就会立即报错。全量拓扑场景则负责发现跨包注册问题。

从一个结论反向找到合适测试

以「ACP 同一 Session 不允许两个并发 Prompt」为例：

1. 先读实现中第一次 await 前的 record.inflight = inflight，理解竞争窗口在哪里。
2. 在 packages/acp/acp/tests/turns.spec.ts 搜并发 Prompt 或 in-flight 场景。
3. 检查测试是否真的让第一个请求停在异步接纳阶段，而不是顺序调用两个已完成请求。
4. 再看取消、转换失败和 Session Dispose 是否都清除槽位。
5. 若要声称某个编辑器集成可用，还需在对应客户端与传输环境做端到端检查。

测试只有真正制造出设计要防范的时间窗口，才能为并发约束提供证据。同名测试不能作证。沿着输入和观察结果阅读，会比看到实现旁边有同名测试就接受结论更可靠。

独立 Eval 为什么仍然需要

仓库内部测试主要保护 DeepSeek Harness 自己的契约，无法自动判断 Agent 是否在陌生仓库中正确完成任务。若要评测完整任务，至少还要保存下面这些输入、过程和判定依据：

锁定任务输入、工作区基线、模型、Provider 和 Harness 配置；

保存 Session、Tool Trace、文件差异、停止原因、成本和时长；

由运行外部执行目标测试或评分器；

把基础设施中断与模型/工具造成的任务失败分开；

保留失败尝试，不能反复运行到一次通过后覆盖原结果。

一次评测不能拼接多次运行的材料。所有产物都必须来自同一次可识别的运行，否则不同尝试的 Session、文件差异和测试结果混在一起后，Evaluator（评估器）就无法判断某个分数究竟对应哪条执行轨迹。

如果还要用反馈数据训练模型，数据管线必须先经过显式适配，把原始点赞、文本反馈或测试结果转换成语义清楚的 Reward。定义训练所用的 Reward、选择 Checkpoint 和执行最终发布评测时，还要分别使用隔离的数据与判断主体，否则训练信号可能进入最终结论。

源码能支持和不能支持的结论

把结论限制在证据能够支持的范围内，我们可以从锁定源码和测试中核对这些内容：

Agent、Session、ToolRuntime、Approval 和 Compaction 的数据流；

特定错误、取消和生命周期分支是否被显式处理；

仓库作者为哪些边界写了自动检查。

这些材料只能证明上面的内容，不能继续推出：

所有真实模型和 Provider 都满足相同假设；

未运行过的平台、文件系统或 Sandbox 已兼容；

长时间、高并发或恶意输入下没有未知故障；

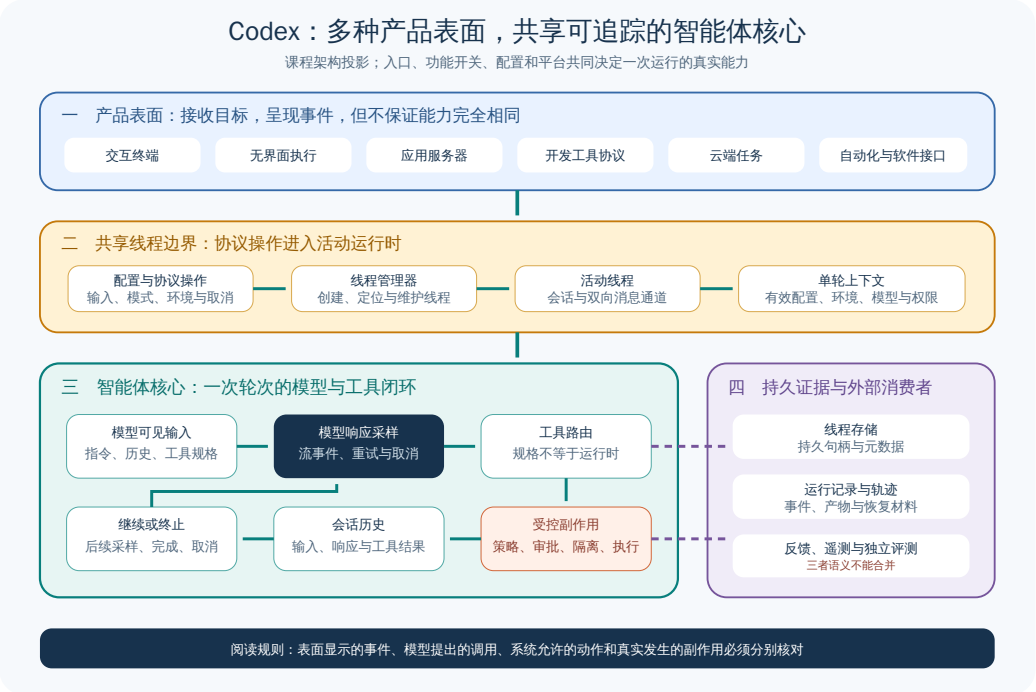
某个具体部署已经满足安全、成本和发布要求。

读到这里，你已经沿 DeepSeek Harness 的装配链走到外部证据。现在可以回到 课程地图，按照「首次源码走读」的顺序打开永久链接逐项复核，再进入 Codex 课程 比较另一套架构怎样处理相同问题。

Codex 源码课程

返回学习入口

Codex 的公开仓库围绕 Rust 核心展开，同时接入 CLI、无头执行、IDE（集成开发环境）和应用协议。这套课程不教某个界面怎么操作。它要带你看清 Thread、Turn（回合）、模型请求、工具路由、执行策略、Sandbox、事件和持久化怎样接力完成同一项任务。



这条课程适合谁

如果你想弄懂强类型事件和异步任务怎样跑起来，又想知道多个产品入口如何共用 Agent Harness（智能体框架）的核心，这条路线很合适。你不必先学完 Rust。文章会顺着调用链解释 enum、trait、async channel 等结构各自在做什么。

锁定来源

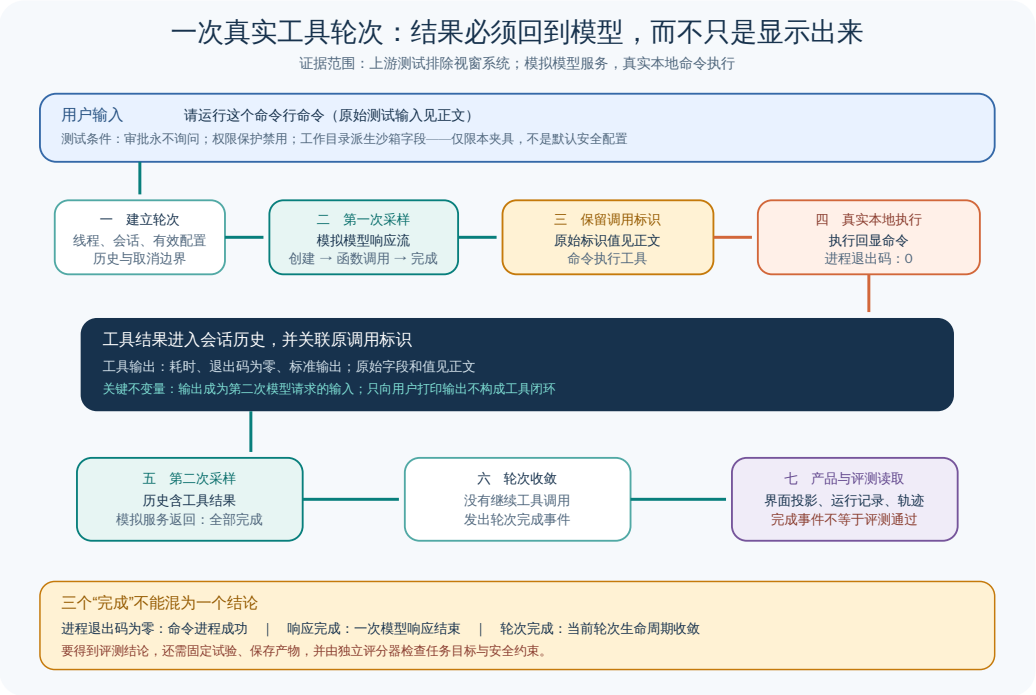
课程基于 Codex 提交 c9b19deb09c1841ce7acc33ddb96276030936a29：

- CLI 入口
- ThreadManager
- Turn 状态
- 工具路由

先看一项任务

用户交来一项代码修改任务后，产品入口会新建或恢复 Thread，再把 Turn 交给核心。核心随后拼出模型输入，解析响应里的工具请求，让策略层和执行环境依次处理，最后把结果写回这项任务。Event 流一边推动界面更新，一边把过程存下来。

CLI / App / IDE
→ ThreadManager
→ Thread 与 Turn
→ Model Client
→ Tool Router
→ Policy / Sandbox / Execution
→ Event 与 Thread Store



图里把各种产品入口画成一处，只为突出它们共用同一套核心。CLI、应用和 IDE 的交互协议并不会因此变得完全相同。别混在一起。

仓库地图

区域	首轮关注点
codex-rs/cli	CLI 如何选择交互、exec 和其他表面
codex-rs/core	Thread、Turn、模型、工具与事件主链
codex-rs/core/src/tools	工具发现、路由和各类 Handler
Sandbox 与 exec policy 相关 crate	命令审批和环境隔离怎样分工
codex-rs/protocol 与 app-server	多客户端怎样交换任务和事件
codex-rs/thread-store	Thread 历史怎样持久化与查询

第一遍先跳过 UI 渲染、安装器、Provider（模型提供商）的边缘兼容，以及与当前任务无关的 Connector。别急着铺开。

三层读法

Starter：读配置、Thread/Turn 和工具循环，先跟完一个工具请求。

Builder：继续追踪执行策略、审批、Sandbox、Rollout 和压缩，解释状态怎样跨轮演化。

Maintainer：阅读多表面、子代理与事件章节，并从测试检查并发顺序、错误分类和恢复边界。

阅读顺序

八篇文章始终跟着同一项任务走。你会逐层查清模型看见了什么，输入在哪一层生效，动作怎样真正发生，最后又由谁独立核对结果。

1. 配置、项目指令与模型输入 从模型实际看见的内容起步，把配置、项目指令、历史和工具投影为一次模型请求。它留下「这次请求落在哪一层生命周期」的问题，为下一篇划定入口。
2. Thread、Task 与 Turn 接住这个运行边界问题，分清 Thread、Session、Task、Turn 与 Op/Event。边界明确后，下一篇才能在当前 Turn 内追踪 Function Call 的去向。
3. 模型响应与工具循环 沿当前 Turn 继续追踪模型输出，经 Tool Router、Handler 和有序结果回填形成再次采样的闭环。工具一旦走到 Shell，下一篇便要回答它能以什么权限执行。
4. 执行策略、审批与 Sandbox 接住命令权限问题，分开 Exec Policy、用户审批与实际隔离。进程结束后，这些安全决策和执行结果如何留下证据，成为下一篇的起点。
5. Rollout、历史、压缩与恢复 沿执行后的证据继续走，分清完整 Rollout、模型可见历史、Compaction 与跨线程 Memory。历史怎样接续清楚以后，下一篇再检查当前任务如何获得扩展能力。
6. 扩展表面与 Code Mode 接住「当前任务还能使用什么」的问题，区分 Skill、Hook、Plugin、MCP 与 Code Mode 介入的层次。单个 Agent 的能力面理清后，下一篇转向多个独立 Thread 怎样协作。
7. 子代理与编排 把单 Agent 的工具调用推进到 Thread 树，解释父子身份、状态、通信、等待、打断与回收。树内控制厘清后，下一篇把视线移到 CLI、Exec、App Server 等树外表面及其证据投影。
8. 事件、Trace 与 Eval 接缝 接住多表面怎样共享核心的问题，继续追到 Event、Rollout Trace、OTel 与 Feedback。最后由 Codex 运行之外的测试和显式 Evaluator 核对结果，收束这条从模型输入到独立判定的链条。

用贯穿任务复盘

再把运费任务放回 Codex 的调用链里走一遍。产品入口新建或恢复 Thread，用户输入开启 Turn，项目指令和工具随后进入模型请求。Tool Router（工具路由器）把读取、编辑和测试分派出去，Exec Policy（执行策略）、Approval 与 Sandbox 各自作出判断，Rollout 和 Thread 的 Store 则保存相关事件。到最后你还要讲清楚：Turn 正常结束，只能证明这一轮已经收住，不能替目标测试下结论。

复盘时要专门核对两种顺序：并发工具实际上按什么次序完成，系统又按什么固定次序把结果写进模型历史。两者可以不同，但 Call 的 ID 和前后因果不能乱。

完成课程后应该能回答

Thread、Turn 和产品表面分别拥有哪部分状态；

模型工具请求怎样进入 Router 和具体 Handler；

产品策略、用户审批和 Sandbox 在哪里分开；

事件怎样同时服务界面、SDK 和历史存储；

压缩与恢复怎样影响下一轮 Context；

哪些测试能够核对工具和任务主链。

从第一篇开始：配置、项目指令与模型输入

配置、项目指令与模型实际看到的输入

返回 Codex 课程地图

第一次读源码，可以先做三件事：观察 Agent 怎样处理一个 Task，找到核心入口，再跟完一条调用链。这一篇顺着输入往下看，目标是弄清模型到底收到了什么。先别急着看请求。磁盘和命令行先给出候选配置，Thread 启动 Session 时，再把它们合成当前有效的 Config。到了每一个 Turn（回合），Codex 才把指令、历史、Tool 及其 Tool Schema 和用户输入装进模型请求。很多人会把「配置文件里的内容」直接当成发给模型的 Prompt，问题就出在这里。

```
配置文件 / 命令行覆盖 / 启动参数
            ↓
        有效配置
            ↓
模型默认指令 + 开发者指令 + AGENTS.md
            ↓
历史正规化 + 当前输入 + Tool Schemas
            ↓
        一次模型请求
```

为什么必须区分这三层

假设仓库根目录有 AGENTS.md，子目录也有一份。用户起初在根目录工作，后来切进子目录，磁盘上的文件虽然没变，模型能看到的项目指令却多了一份。再比如，Thread 已经启动，此时修改默认 Model 只会影响后续请求，未必会改写 Rollout（运行轨迹）里已经保存的 Turn Context。旧记录还在。

因此，排查「模型为什么没遵守某条指令」时，你要按实际动作来查：Codex 有没有找到文件，合并字段时该值有没有胜出，这一轮发出的请求里到底有没有它。

第 1 站：有效配置按字段合并，不是整个对象覆盖

源码：查看模型与指令字段的合并

```
let model = model.or(cfg.model);
let base_instructions = base_instructions
    .or(file_base_instructions)
    .or(cfg.instructions.clone());
let developer_instructions =
    developer_instructions.or(cfg.developer_instructions);
```

调用者：线程或 Session 启动时的配置解析流程。

输入：运行时参数、文件加载结果和全局配置。

状态变化：逐字段选择优先值，形成不可再用「来自哪个文件」简单概括的有效配置。

返回：供 Session、Provider 和 Prompt 构造使用的 Config。

下一站：项目文档发现与 Turn Context 构造。

Rust 的 Option::or 会取左边第一个 Some。所以「命令行优先」只适用于按这种办法合并的字段，要确认某个具体选项以谁为准，仍得回到它自己的解析代码。别凭印象猜。

AGENTS.md 是项目上下文，不是无限长系统提示

Codex 会顺着工作目录逐层寻找项目指令，同时检查最大字节数和后备文件名。你进入更深的目录后，它会把新找到的指令放进后续 Turn 的 Context。历史里的旧消息已经写定，不会因为当前目录变了就跟着重写。

第 2 站：项目文档限制先进入有效配置

源码：查看项目文档大小与后备名称

```
project_doc_max_bytes:
    cfg.project_doc_max_bytes.unwrap_or(AGENTS_MD_MAX_BYTES),
project_doc_fallback_filenames: cfg
    .project_doc_fallback_filenames
    .unwrap_or_default()
    .into_iter()
    .filter_map(|name| {
        let trimmed = name.trim();
```

调用者：Config Builder。

输入：用户配置中的字节上限和后备文件名。

状态变化：填入默认上限，去掉无效名称。

返回：项目文档发现器使用的确定参数。

下一站：发现器读取目录链中的指令文件，Turn 将新增内容写入上下文。

文件明明存在，却仍可能进不了模型输入。常见原因有四种：文件太大，文件名不在候选表里，文件不在工作目录这条链上，或者 Codex 读取失败。查 Prompt 时，只看文件树不够。这一步很关键。

Context Fragment 是带契约的输入片段

Codex 不会把所有 Context 都当成普通聊天消息，有些内容会实现 ContextualUserFragment，由实现代码说清角色、内容种类、是否单独占一条 Message，还要给出边界标记和正文。

第 3 站：片段自己声明怎样渲染

源码：查看 ContextualUserFragment

```
pub trait ContextualUserFragment {
    fn role(&self) -> &'static str;
    fn content_kind(&self) -> ContentItemKind;
    fn requires_separate_message(&self) -> bool { false }
    fn markers(&self) -> (&'static str, &'static str);
    fn body(&self) -> String;
}
```

调用者：把扩展上下文插入模型历史的构造器。

输入：一个具体 Fragment 实现。

状态变化：按角色、消息边界和 Marker 投影为模型可读内容。

返回：一个或多个 Response Input Item。

下一站：Context Manager 与普通历史一起做正规化。

Marker 绝不是装饰。不同片段如果直接拼在一起，又没有稳定边界，模型就很难分清哪段是「项目规则」、哪段是「环境信息」、哪段才是「用户任务」。无关文本只要换个顺序，Prompt Cache 也更容易失效。边界必须稳定。

历史不能原样塞进下一次请求

Session 保存的记录里，可能混有当前模型不支持的模态、内部 Event，也可能有必须成对出现的工具条目。发请求前，Context Manager（管理器）会先整理这段历史，去掉不能发送的内容，并把结构修正好。

第 4 站：模型历史是持久记录的投影

源码：查看历史正规化入口

```
pub(crate) fn history_for_prompt(
    &self,
    input_modalities: &[InputModality],
) -> Vec<ResponseItem> {
    let mut input = self.contents.clone();
    self.normalize_history(&mut input, input_modalities);
    input
}
```

调用者：一次模型采样前的 Turn Loop。

输入：Session Context 中的记录与当前模型支持的输入模态。

状态变化：只修改本次投影副本，不把裁剪结果冒充原始 Rollout。

返回：可发送给模型的有序 Response Items。

下一站：模型客户端把它与 Tools、Model 和请求选项一起发送。

用一个失败测试任务串起来

用户说「修复解析器测试」以后，这句话不会原封不动地直达模型。它大致要经过下面这些环节，Sandbox 等运行限制也会跟着有效配置进入后续路径：

1. 启动参数和配置文件决定模型、审批策略与 Sandbox 策略。
2. 根 AGENTS.md 告诉 Agent 运行哪些检查；进入 parser/ 后，子目录规则作为新增 Context 进入。
3. 过去消息经正规化，不能发送的内部项被过滤。
4. 当前用户输入与可见 Tool Schemas 加到请求尾部。
5. 模型返回 Tool Call，进入后续工具循环。

如果模型违反了子目录规则，你该检查切换目录以后发出的那次请求体。Thread 启动时打印的 Config，只能说明启动那一刻发生了什么。

走到这里，Codex 已经把配置和 Context 整理成一次模型请求，但只看「一个请求」还画不出完整的运行边界。下一篇会分清 Thread、Session、Task 和 Turn，你也就能判断模型返回的 Tool Call（工具调用）究竟落在哪一层。

下一篇：Thread、Task 与 Turn。

Thread、Session、Task 与 Turn 到底分别是什么

返回 Codex 课程地图

上一篇停在模型返回 Tool Call（工具调用）、准备进入工具循环的地方。要继续往下追，你得先分清 Thread、Session、Task 和 Turn（回合），因为 Codex 源码还会同时出现 Op 和 Event，这些名字可不能都理解成「任务」。它们各自管着一段不同的生命周期：Thread 提供可以长期引用的会话身份，Session 装着当前还在活动的运行状态，Task 推进某一类后台工作流，Turn 对应一次用户交互。Op 是产品表面发给核心的命令，Event 则是核心往外发的通知。

```
ThreadId（持久身份）
  └── 活动 Session（服务、队列、当前 Turn）
      └── ActiveTurn
          ├── TurnState
          └── RunningTask（Regular / Review / Compact）
```

一个 Turn 可以包含多次模型请求

用户只提交一次「修复失败测试」，Codex 就会开启一个 Turn。随后，模型可能读取文件、运行测试、修改代码，再跑一次测试，每个 Tool Result 都可能触发新的模型请求，可这些动作仍属于当前 Turn。若只按网络请求计数，你会把一次交互误拆成四个 Turn。先记住这个差别。

第 1 站：Session 拥有唯一活动 Turn 和输入队列

源码：查看 Session 的核心字段

```
pub(crate) struct Session {
    pub(crate) thread_id: ThreadId,
    pub(crate) active_turn: Mutex<Option<ActiveTurn>>,
    pub(crate) input_queue: InputQueue,
    pub(crate) services: SessionServices,
}
```

调用者：CodexThread 和 Session 方法处理提交、转向、取消与恢复。

输入：Thread 身份、配置和共享服务。

状态变化：Input Queue 接纳消息；active_turn 在 Idle 与某个活动 Turn 之间切换。

返回：活动 Session 由 CodexThread 对外提供类型化操作。

下一站：提交逻辑创建 ActiveTurn 并启动对应 Task。

Mutex<Option<ActiveTurn>> 写得很直白：同一个 Session 最多只能有一个活动 Turn。不过，这不代表整个程序同时只能跑一个 Task，不同 Thread 可以并发，同一个活动 Turn 里的 Tool Call 也可能并发。这三层不能混。

第 2 站：Turn 状态与运行控制分开保存

源码：查看 ActiveTurn 与 RunningTask

```
pub(crate) struct ActiveTurn {
    pub(crate) task: Option<RunningTask>,
    pub(crate) turn_state: Arc<Mutex<TurnState>>,
}

pub(crate) enum TaskKind { Regular, Review, Compact }
```

```
pub(crate) struct RunningTask {
    pub(crate) cancellation_token: CancellationToken,
    pub(crate) turn_context: Arc<TurnContext>,
}
```

调用者：Session 启动、转向和中断代码。

输入：本轮 Context、Task 类型与 Cancellation Token。

状态变化：TurnState 记录本轮可变状态；RunningTask 保存正在执行的 Future 控制面。

返回：可供中断、等待和追加输入使用的 ActiveTurn。

下一站：具体 Task 的 run() 驱动普通 Agent、Review 或 Compaction 工作流。

读到这里，你可以问一句：界面上还显示着某个 Turn，是否说明 Task 仍在运行？不能这么推。Codex 把状态和运行控制分开保存，即使 Task 的 Future（异步计算）已经进入清理阶段，TurnState 仍会留下先前发生的 Event，而界面能显示这个 Turn，只能证明相关状态还在。先把两层分开。

Task 是工作流接口，不等于普通 Agent Loop

这里先排除一个常见误解：几个类型的名字里都有 Task，并不代表它们内部都跑同一条 Agent Loop（智能体循环）。

第 3 站：不同 Task 共享启动方式，不共享内部实现

源码：查看 Task Trait

```
#[async_trait]
pub(crate) trait SessionTask: Send + Sync + 'static {
    fn kind(&self) -> TaskKind;
    async fn run(
        self: Arc<Self>,
        sess: Arc<Session>,
        ctx: Arc<TurnContext>,
        input: Vec<TurnInput>,
        cancellation_token: CancellationToken,
    ) -> Option<String>;
}
```

调用者：Session 根据 Op 选择并启动 Task。

输入：Session、Turn Context、初始输入和取消信号。

状态变化：具体实现可以运行普通模型循环、代码审查或压缩。

返回：可选的最后文本；完整事实仍通过 Events 和 Rollout 保存。

下一站：Task 结束时 Session 结算 Turn 并释放活动槽位。

所以，看到 TaskKind::Compact 时别急着往下猜。这个名字不能证明它会走与 Regular 相同的 Tool Loop，也不能拿某一种 Task 的成功条件套在其他 Task 上。

Op 是命令，Event 是观察结果

产品表面会提交 Op::Interrupt、用户输入或者配置变更。核心处理以后，才会发出 TurnStarted、Item（条目）事件、错误和 TurnComplete，而收到 Op 只说明核心接下了命令，不能证明操作已经完成。你要判断后来究竟发生了什么，得看 Event。

第 4 站：协议明确分开输入与输出

源码：查看 Op 与 Event 定义

```
pub enum Op {
    Interrupt,
    // 其他提交给核心的操作
}

pub struct Event {
    pub id: String,
    pub msg: EventMsg,
}
```

调用者：CLI、App Server、IDE 等表面发送 Op；核心发出 Event。

输入：带 Thread/Turn 语境的操作。

状态变化：由 Session 验证是否有活动 Turn、是否能转向或中断。

返回：提交确认与后续异步 Events。

下一站：表面根据 Event 更新 UI，Rollout Writer 持久化相关记录。

读这一段源码时，始终问清一件事：眼前这条记录，是产品表面发出的命令，还是核心发回的结果？方向不能看反。

Follow-up、Steer 与下一 Turn

Session 处于 Idle 时，如果此时提交一个新的用户任务，对应的处理是建立一个新 Turn，判断时应当从 Session 的 Idle 状态起步。

Turn 仍然活动时，转向输入会加入当前控制流，并且继续复用这个 Turn 的身份，别把这类输入误判成新 Turn。

普通 Follow-up 可以先进入队列，等当前工作收敛以后，它才会进入后续处理。

Interrupt 触发的是 Cancellation Token，但中断的边界要读准：Thread 和过去的 Rollout 都不会被删除。

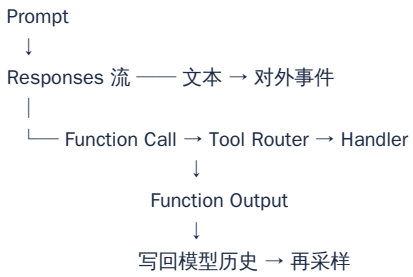
拿这些边界排查 Bug 时，先别急着问模型为什么没响应，你应该先确定消息去了哪里：它属于当前 Turn，准备进入下一个 Turn，还是只在队列里等着？位置找准后，再沿着那一层继续查。

这些边界分清后，我们就能继续看当前 Turn 里面的动作：模型流吐出 Function Call（函数调用）后，谁把它交给 Tool Router？工具执行完，又是谁按原来的调用身份把结果写回 Context？答案在下一篇：模型响应与工具循环。

模型响应怎样变成工具调用，再回到下一次采样

返回 Codex 课程地图

上一节把 Follow-up、Steer（中途引导）和新 Turn（回合）分别归到了 Thread、Task 和 Turn 的哪一层。现在边界清楚了，我们来看这个 Turn 里模型到底吐出了什么，其中可能有普通文本、Reasoning、Function Call 和其他 Item。Codex 不会只调用一次 Model 就打印文本。Tool Router 拿到解析出的 Function Call，交给工具执行，再按原来的调用身份把结果写回 Context。Turn 随后判断要不要继续采样。



两个顺序必须同时成立

假设一次 Response 提出 A、B、C 三个 Tool Call（工具调用），Agent Harness（智能体框架）可以让支持并行的工具同时执行，但下一次模型请求仍要按 A→结果 A、B→结果 B、C→结果 C 对齐。谁先执行完，和谁先写回历史，是两件事，协议里的写回顺序就在这里定下来。次序不能乱。

第 1 站：Router 按调用种类找到具体 Handler

源码：查看 Tool Router

```
pub(crate) struct ToolRouter {
    registry: ToolRegistry,
}

// 路由器根据 ToolCall 找到 Handler，并把调用上下文交给它。
```

调用者：Turn 对模型输出 Item 的处理流程。

输入：Tool Call、Turn Context、Call ID 和 Cancellation Token。

状态变化：选择注册的 Handler，进入统一 Dispatch/事件路径。

返回：可写回模型的 Tool Output 或类型化失败。

下一站：并行调度器决定本调用取得共享锁还是独占锁。

Router（路由器）只负责把模型协议里的对象转成框架内部调用。Shell、文件和 MCP 各自怎么执行，要看对应的 Handler，Approval（审批）与 Sandbox 则在更后面的链路上处理，所以一句「Router 允许了」把好几层都说混了。

第 2 站：工具元数据决定是否允许并行

源码：查看并行执行门

```
let supports_parallel = router.tool_supports_parallel(&call);
let _guard = if supports_parallel {
    Either::Left(lock.read().await)
} else {
    Either::Right(lock.write().await)
```

```
};
router.dispatch_tool_call_with_terminal_outcome(...).await
```

调用者：一批 Tool Calls 的调度代码。

输入：待执行调用、Router 与同批次读写锁。

状态变化：可并行工具获取读锁；独占工具获取写锁并等待其他调用退出。

返回：单个调用的终态结果。

下一站：有序 Future 容器等待并按提交顺序取回结果。

并行标志只说明框架允许多个调用的 Body 同时运行，它不保证这些调用碰不到同一份业务资源。比如，两次文件写入都允许并行，仍可能争抢同一个路径，这类冲突得由 Handler 自己管住。

第 3 站：执行可乱序完成，历史按调用顺序提交

源码：查看 Turn 中的有序 Tool Future

```
let mut in_flight: FuturesOrdered<_> = FuturesOrdered::new();

while let Some(res) = in_flight.next().await {
    sess.record_conversation_items(...).await;
}
```

调用者：模型流处理完当前批次 Tool Calls 后的 Turn Loop。

输入：按模型出现顺序压入的 Tool Futures。

状态变化：Future 可以并发推进；取出与记录保持原始顺序。

返回：稳定排序的 Function Call Outputs。

下一站：Context Manager 把调用和结果交给下一次模型请求。

这就解释了慢工具为什么会造成 Head-of-Line 等待。即使 B 已经跑完，只要 A 还没结束，Codex 就不会让 B 的结果越过 A 写入历史，它用这段等待保住了可重放、稳定的协议顺序。

用测试确认你理解的并行不是「看起来同时开始」

源码：查看上游批次断言

```
assert_eq!(function_calls.len(), 3);
assert_eq!(function_call_outputs.len(), 3);
assert!(*index < *output_index);
assert_eq!(call.1.get("call_id"), output.1.get("call_id"));
```

这还不够。

这里的断言只检查请求体怎么排列，以及 Call ID 是否一一对齐，若要证明多个 Body 真的同时运行，你还得看测试有没有设置屏障或时间窗口，能不能让多个 Handler 在同一时刻保持运行状态。

模型流重试与工具重试不是一回事

先看清到底在重试谁。

Provider 的 Stream（流）可能在响应尚未结束时断开，遇到这种情况，Codex 只会重连被归为可重试的 Error，次数也受 Provider 配置限制，工具已经落地的副作用则不能跟着模型流无条件重跑。

第 4 站：重试受错误分类和次数上限控制

源码：查看模型流重试

```
let max_retries = turn_context.provider.info().stream_max_retries();
if !err.is_retryable() {
    return Err(err);
}
handle_retryable_response_stream_error(..., max_retries, ...).await?;
```

调用者：Turn 的 Responses Stream 循环。

输入：Provider Error、已重试次数与 Cancellation Token。

状态变化：记录重试信息；超过上限或不可重试时结束当前路径。

返回：新的 Stream，或向上返回错误。

下一站：继续处理 Response Items，或结算 Turn Error。

一次失败测试修复的完整闭环

1. 模型先调用 Read；Result 写回 Context。
2. 模型调用 Shell 运行测试；非零退出是工具成功执行后返回的业务结果，不一定是 ToolRuntime 故障。
3. 模型调用 Patch 修改代码。
4. 模型可能并发 Read 两个互不依赖文件，但 Patch 或某些执行工具串行。
5. 第二次测试通过后，模型生成最终文本；外部仍可再次独立运行测试核对产物。

到这里，整条链已经走完。Codex 把模型流里的 Function Call 解析成 Tool Call，Router 找到对应 Handler，再按照工具元数据决定并行还是独占执行，有序的 Future 会按原调用顺序取出结果，写回 Context，然后开始下一次采样。下一篇沿命令执行路径往下读，看看 Exec Policy 怎样作出三值判断，用户审批怎样处理额外权限请求，Sandbox 又怎样准备真正的隔离环境。

下一篇：执行策略、审批与 Sandbox。

执行策略、用户审批与 Sandbox 为什么必须分开

返回 Codex 课程地图

上一讲已经把模型流、工具调用和结果回填串起来了。工具一走到 Shell，这次命令能用什么权限执行，就成了绕不开的问题，权限问题也从这里开始。

这里最容易混淆的说法是「命令已经放行，所以它肯定在沙箱里安全地跑完了」。Codex 会分三步处理这件事：Exec Policy（执行策略）先判断请求该放行、询问还是禁止，Approval Policy（审批策略）再决定能不能向用户申请额外权限。最后才轮到 Sandbox 一层的 Manager，它会按当前平台和权限要求准备隔离环境。

```
模型提出命令
    ↓
Exec Policy：Allow / Prompt / Forbidden
    ↓
必要时请求用户一次性审批
    ↓
计算 Effective Permission Profile
    ↓
选择并准备平台 Sandbox
    ↓
启动进程并收集输出
```

Allow 只说明策略层不再拦这条请求，Prompt 则说明还得等一次决定，就算用户点了允许，授权也只管这一次尝试。命令最后有没有进隔离环境，还得看后面具体怎样生成执行请求。

第 1 站：Exec Policy 输出三值决定

源码：查看 Decision

```
pub enum Decision {
    Allow,
    Prompt,
    Forbidden,
}
```

调用者：执行工具在准备命令时查询策略。

输入：命令、工作目录、规则集与请求的权限。

状态变化：不启动进程，只生成策略结论。

返回：允许继续、要求审批或立即拒绝。

下一站：Approval 流或 Sandbox 请求构造。

这里必须保留三个结果，因为「目前没拿到授权，但可以问用户」和「问了也不能放行」会把流程带向不同分支。如果两种情况都折成 false，UI 就不知道该显示审批入口，还是直接告诉用户这条命令不能执行。

这两种情况不能混。

第 2 站：需要 Sandbox 与平台能否提供 Sandbox 是两个问题

源码：查看平台选择与需求判断

```
pub fn get_platform_sandbox(...) -> Option<SandboxType> {
    // 根据平台与配置选择后端
}
```

源码：查看 should_sandbox()

```
pub fn should_sandbox(...) -> bool {
    // 请求是否需要隔离，与主机能否提供后端分开判断
}
```

调用者：执行请求构造器。

输入：Sandbox Policy、Permission Profile、平台能力和用户配置。

状态变化：选择 Seatbelt、Landlock、Windows 后端或无后端；另行判断请求是否要求隔离。

返回：可选 Sandbox Type 与需求判断。

下一站：若需要且有后端，转换命令；不满足时按策略失败或走明确的无 Sandbox 路径。

这两个函数各管一件事，系统才能说清「请求要求禁用网络，但当前平台没有可用后端」究竟卡在哪里。如果后端缺失，安全层就该明确报出缺失，不能顺手把它解释成「无需隔离」。

第 3 站：有效权限先合并，再生成平台命令

源码：查看 Sandbox 命令转换

```
let base_effective_permission_profile =
    effective_permission_profile(permissions, additional_permissions.as_ref());

let (argv, arg0_override, pending_sandboxed_request) = match sandbox {
    // 各平台后端分支
};
```

调用者：Shell/Exec Handler 准备子进程。

输入：原命令、基础权限、这次批准的附加权限、平台 Sandbox。

状态变化：合成 Effective Profile；生成包装后的 argv、环境或待处理 Sandbox 请求。

返回：可以交给 Process Host 的执行规格。

下一站：实际创建进程，并把 stdout、stderr、退出状态送回 Tool Result。

看到这里，Approval（审批）管什么已经很清楚了。它只决定「这次请求可以多拿哪些权限」，不会替你提供平台后端，也保证不了后端一定能初始化成功。

Require Escalated 也受 Approval Policy 约束

模型不能想当然地把每条命令都提到宿主权限，这件事同样受 Approval Policy 约束。上游测试专门覆盖了一种情况：细粒度 Sandbox 审批关闭以后，RequireEscalated 请求会直接失败。

源码：查看提升权限拒绝测试

```
sandbox_approval: false,
sandbox_permissions: SandboxPermissions::RequireEscalated,
output_contains: "you cannot ask for escalated permissions",
```

这里拒绝命令的还不是 Sandbox。请求在走到 Sandbox 之前就违反了当前审批约定，因此流程提前停下，命令根本没有进入隔离环境。

拒绝发生得更早。

Fail Closed 的具体含义

如果 Windows 受限令牌这条路径落实不了 Deny-Read 约束，测试就要求它直接报错，绝不能悄悄撤掉限制，再让命令裸跑。

源码：查看 Windows 拒绝读取测试

```
.expect_err("restricted-token sandbox should reject deny-read restrictions");
```

Fail Closed 要求的是：策略既然承诺了某项限制，当前执行路径又落实不了，系统就必须拒绝这次执行。它没有要求所有命令一律进入 Sandbox，也不允许系统偷偷降低约束。

排查一次「为什么弹出审批」

按这条顺序查：

1. Tool Call 请求了什么命令和权限；
2. Exec Policy 为什么返回 Prompt；
3. 当前 Approval Policy 是否允许询问与提升；
4. 用户决定是否只对本次 Call 有效；
5. Effective Permission Profile 最终是什么；
6. 选择了哪个平台后端；
7. 子进程是否真的以该规格启动。

如果你只盯着最后的命令输出，就看不到前面为什么作出每个安全决定，也查不清这些决定究竟在哪一层生效。

进程跑完，还不能收工。安全决定和执行结果会分别写进 Rollout（运行轨迹）、模型历史、Compaction（上下文压缩）和 Memory。下一篇顺着这些记录往下看。

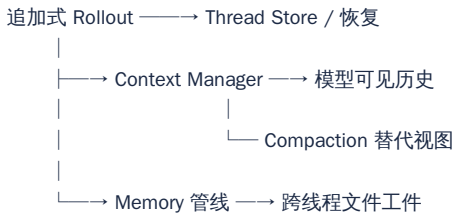
下一篇：Rollout、历史、压缩与恢复。

Rollout、模型历史、Compaction 与 Memory 不是同一份数据

返回 Codex 课程地图

上一讲顺着 Exec Policy（执行策略）、Approval Policy（审批策略）和 Sandbox 一路追到进程启动。进程结束以后，安全决定和执行结果还会分别写进 Rollout、模型能看到的 History、Compaction 和 Memory。

这几份数据各有用处。恢复程序要读完整记录，模型只接收容量有限的 Context，界面要把事件整理成可显示的内容，跨线程 Memory 管线则从旧任务中挑出以后还用得上的信息。你要是把它们全叫作「会话历史」，做压缩时很可能删掉证据，恢复时也可能误把摘要当成原文。



第 1 站：Rollout Item 不只是聊天消息

源码：查看持久记录类型

```
pub enum RolloutItem {
    SessionMeta(...),
    ResponseItem(...),
    Compacted(...),
    TurnContext(...),
    WorldState(...),
    EventMsg(...),
}
```

调用者：Rollout Writer 记录 Session、Turn 与模型/工具事件。

输入：协议消息、上下文快照、压缩项和运行事件。

状态变化：向当前 Thread 的持久记录追加 Item。

返回：可供恢复器、Trace 和界面读取的有序序列。

下一站：Thread Store 用 ThreadId 定位后端，恢复器重建 Initial History。

ResponseItem 和模型看到的历史最接近，可恢复程序还得依靠其他 Item（条目）解释当时发生了什么。只导出聊天文字，会把 Turn Context、Compaction 边界和一部分控制事件丢掉。

这层区别不能省。

ThreadId 是句柄，不是文件路径

第 2 站：存储后端隐藏在 Thread Store 之后

源码：查看 Thread Store 契约

```
//! Application code should treat `ThreadId` as the only durable thread handle.
//! Implementations resolve that id to rollout files, RPC requests,
//! or another backing store.
```

调用者：ThreadManager、列表、恢复和 Fork 功能。

输入：ThreadId 与查询/读取请求。

状态变化：由具体后端解析本地文件、RPC 或其他存储位置。

返回：线程元数据和 Rollout 内容。

下一站：恢复器构造 Resumed 或 Forked History。

如果应用层拿本地 JSONL（逐行 JSON）路径当作唯一身份，存储后端一换到远端，这个身份立刻就失效。应用只认 ThreadId（线程 ID），API 才能在不同存储实现之间继续使用同一个稳定句柄。

文件在哪并不重要。

Resume 与 Fork 必须保留不同语义

源码：查看初始历史类型

```
pub struct ResumedHistory {
    conversation_id: ConversationId,
    history: Vec<RolloutItem>,
    rollout_path: Option<PathBuf>,
}

pub enum InitialHistory {
    New,
    Cleared,
    Resumed(...),
    Forked(...),
}
```

调用者：Thread 创建与恢复流程。

输入：过去 Rollout 与操作意图。

状态变化：Resume 延续原 Thread 语境；Fork 用过去历史建立新的分支身份。

返回：Session 初始化所需的 InitialHistory。

下一站：Context Manager 重放可见历史，Writer 继续写入正确目标。

Resume（恢复）和 Fork（从已有历史建立新分支身份）起步时可以读到同一批消息，但后续写入必须各用各的身份。否则你在 Fork 上做一次实验，内容就会混进原来的 Thread。

Compaction 改的是模型视图，不是假装旧记录没发生

当 Context 快装不下时，Codex 会写出一份 Summary（摘要），再用它和保留下来的尾部重新组成模型历史。同时，Codex 还会在 Rollout 中追加 Compacted Item，恢复程序据此才能知道：后面的 Context 从什么时候开始改用哪份摘要。

第 3 站：压缩生成替代 History 和元数据

源码：查看压缩提交

```
let summary_text = format!("{SUMMARY_PREFIX}\n{summary_suffix}");
let mut new_history = build_compacted_history(...);
sess.replace_compacted_history(
    new_history,
    ...,
```

```
    CompactedHistoryMetadata { ... },
).await;
```

调用者：Compact Task 或自动 Context 压力处理。

输入：当前 History、摘要响应和 Turn Context。

状态变化：替换 Session 中下一次模型使用的 History，并记录 Compaction Metadata。

返回：压缩后的可用 Context。

下一站：后续 Turn 用摘要与保留尾部继续采样；Rollout 保存 Compacted 事实。

摘要一定会损失信息。验证时别只看 Token 数量有没有下降，还要确认新 Context 能不能还原未完成的任务、文件名、用户约束和关键 Tool Result。

少一个都可能误事。

源码：查看压缩后的请求与 Rollout 测试

```
assert_eq!(assistant_count, 0, "assistant history should be cleared");
// Rollout 中仍能找到带期望摘要的 Compacted Item。
```

测试会同时查看模型请求和持久记录，因为压缩会改模型眼前的历史，也会在 Rollout 里留下记录。两边都得核对。

Memory 是跨线程提炼工件

负责写 Memory 的 crate 有自己的目录，还会分别保存 Raw Memories 和 Rollout Summaries。它从过去的任务里挑出可复用的信息，再写成独立文件，不会把所有 Thread 原封不动地拼进一个超长 Prompt。

第 4 站：Memory 有自己的文件布局和管线

源码：查看 Memory 写入模块

```
//! Startup memory pipeline and file-backed memory artifact helpers.

pub const ROLLOUT_SUMMARIES_SUBDIR: &str = ...;
pub const RAW_MEMORIES_FILENAME: &str = ...;
```

调用者：启动或后台 Memory 管线。

输入：选中的历史任务与 Memory 配置。

状态变化：生成摘要、原始记忆或索引文件。

返回：后续任务可以检索的独立工件。

下一站：新 Session 根据需要选择性注入相关 Memory。

Memory 里的内容可能已经过时，提炼时也可能出了错。因此，把 Memory 注入新 Session 时，必须带上来源和时间背景。它能帮模型找回背景，但当前仓库里的文件和用户刚给的指令优先级更高。

旧记忆不能盖过新事实。

Memory 的边界清楚以后，下一篇再看 Skill、Hook、Plugin、MCP 和 Code Mode 分别改动哪一层。目录已经存在、系统发现了它、模型看得见它、最后执行成功，这几个状态也得拆开查。

下一篇：Skills、Hooks、Plugins、MCP 与 Code Mode。

Skills、Hooks、Plugins、MCP 与 Code Mode 各扩展哪一层

返回 Codex 课程地图

前一篇已经分清 Rollout（运行轨迹）、模型能看到的 History（历史记录）、Compaction 和跨线程 Memory。旧历史接上以后，当前任务还能得到哪些额外能力，取决于 Skill、Hook、Plugin、MCP/Connector 和 Code Mode 各自插在流程的哪一层。

别把 Codex 里的「扩展」想成一套统一的插件回调。Skill 在任务用到时提供指令，Hook 盯住生命周期中的特定节点，Plugin 把多种能力装在一起，MCP/Connector 接入外部 Tool，Code Mode 则让模型改用程序来编排这些工具。

它们改动的地方不同。

```
Skill ————————→ 模型如何完成一类任务
Hook ————————→ 生命周期前后执行附加动作
Plugin ————————→ 能力的安装与身份容器
MCP / App ——→ 外部服务与工具目录
Code Mode ——→ 用程序批量调度当前可见工具
```

排查扩展故障时，你得沿着六个检查点逐项确认：目录存在、系统发现、配置启用、模型可见、权限放行、执行成功。前一项通过，只能说明流程可以往后走，证明不了后一项也正常。

Skill：先发现，再在任务需要时加载

第 1 站：Skill 模块同时维护根加载、快照与工具依赖

源码：查看 Skill 公共接口

```
pub use loading::SkillRootLoadRequest;
pub use loading::SkillRootSnapshotCache;
pub use model::SkillToolDependency;
pub use mentions::extract_tool_mentions;
```

调用者：Session 启动、Skill 发现与显式提及处理。

输入：Skill Roots、任务文本和缓存快照。

状态变化：发现候选 Skill，解析元数据与依赖；需要时才读取完整指令。

返回：可展示目录或注入当前任务的 Skill 内容。

下一站：Prompt 构造加入被选中的 Skill，工具表核对依赖是否可用。

Skill 出现在目录里，只能说明发现阶段认出了它。系统还要按任务需要读取正文，把内容放进 Context，并检查它依赖的工具是否可用。目录里看得见，模型不一定看得见。

Hook：观察或干预明确的生命周期点

第 2 站：不同 Hook 事件拥有不同 Matcher 语义

源码：查看 Hook 事件定义

```
// 公开的事件包括 PreToolUse、PostToolUse 等。
// 某些其他事件即使在 JSON 中带 matcher，也会忽略该字段。
```

调用者：工具执行、Session 生命周期或其他 Hook 发射点。

输入：事件载荷、配置的 Command 与 Matcher。

状态变化：同步 Hook 可能影响当前动作；异步 Hook 只在旁路运行。

返回：允许、拒绝、附加上下文或诊断结果，取决于具体事件契约。

下一站：原生命周期继续，Hook 结果按该事件的规则合并。

所有 Hook 即使写在同一份配置文件里，能做的事也可能完全不同。要判断某个 Hook 能不能拦住操作，你仍得找到消费对应 Event 的代码，看它怎样处理这个 Hook 返回的结果。

配置长得像，权限未必一样。

Plugin 是来源容器，不是单一执行引擎

第 3 站：能力摘要保留 Plugin 身份与不同能力种类

源码：查看 Plugin Capability Summary

```
pub struct PluginCapabilitySummary {
    // Skill、Hook、App Connector 等能力摘要
}

pub struct PluginHookSource {
    plugin_id: ...,
    plugin_root: ...,
    hooks: ...,
}
```

调用者：Plugin Loader 与能力目录。

输入：一个 Plugin Root 及其 Manifest/子目录。

状态变化：发现能力并保留来源 Plugin ID，方便权限、诊断和卸载。

返回：分类后的能力摘要。

下一站：Skills、Hooks、Connector 等各自进入自己的加载器。

卸载 Plugin 时，加载器还得按来源撤掉它贡献的每项能力。只删目录不够，旧的 Tool Catalog 仍可能留在正在运行的 Session 里，让当前任务继续看到已经卸载的工具。

MCP 与 Connector：连接成功后还要决定模型是否可见

MCP Manager（管理器）维护连接和 Tool Catalog Cache，Connector 的 Runtime（运行时）则继续处理策略、鉴权和工具投影。外部 Server 返回了一个工具，只能证明目录里有它，还不能证明当前模型有权调用它。

源码：查看 MCP 可见性与目录缓存导出

```
pub use connection_manager::tool_is_model_visible;
pub use tool_catalog_cache::McpToolCatalogCache;
```

调用者：MCP 连接管理与 Tool Schema 构造。

输入：Server Catalog、当前配置和工具元数据。

状态变化：缓存目录，并过滤当前模型不可见的工具。

返回：可合并进 Tool Registry 的定义。

下一站：Tool Router 在调用时仍执行审批与外部请求。

从连上服务器到拿到业务结果，中间至少隔着五个状态：连接建立、目录发现、模型可见、调用获准、远端执行。报错时必须说清卡在哪一层，否则你很容易拿着鉴权问题去查 Tool Schema，或拿着可见性问题去重启服务器。

Code Mode：把多次工具往返收进一个程序

Code Mode 会给模型一个代码执行入口，模型写出的程序再通过绑定去调用现有工具。要读很多文件、汇总结果，或者按条件跑一批操作，这种方式很合适。但它仍会经过 Tool Router（工具路由器），没有多出一条直通宿主的后门。

第 4 站：宿主缺失时的回退取决于配置模式

源码：查看 Code Mode 宿主缺失测试

```
// 普通模式：缺少 Process Host 时回退到直接工具并警告一次。
// code-mode-only：绝不暴露直接 Shell 工具，而是失效关闭。
```

调用者：Tool Registry 构造阶段。

输入：Code Mode 配置、Process Host 可用性和直接工具集合。

状态变化：选择 Code Mode 工具面、回退面或 Fail-Closed 面。

返回：本轮模型实际可见的 Tool Schemas。

下一站：模型生成普通 Tool Call 或 Code Program。

普通模式启用 Code Mode，只是想减少模型和工具来回往返。Process Host（宿主）缺失时退回直接工具，原来的能力还在。code-mode-only 则明确要求模型只能走代码入口，如果这时暴露直接 Shell，就违反了配置给出的限制，因此系统只能拒绝继续。

配置决定了退路。

一次扩展故障应该怎样定位

当模型说「没有某工具」时，按顺序检查：

1. Plugin/Skill/MCP 配置是否被发现；
2. 来源是否被信任并启用；
3. 外部连接和鉴权是否成功；
4. 当前 Agent 的 Tool Catalog 是否把它投影为模型可见；
5. Skill 是否已经加载完整正文；
6. 调用时是否被 Approval 或 Sandbox 拒绝；
7. 远端工具是否返回业务错误。

单个 Agent 能看到哪些扩展，现在已经有了排查顺序。下一篇要处理多个独立 Thread 怎样协作：谁保存父子身份，谁负责发消息、等待、打断和回收。答案要从它们共享的 AgentControl 往下找。

下一篇：子 Agent 与线程树编排。

子 Agent 为什么是一棵 Thread 树，而不是一组函数调用

返回 Codex 课程地图

上一篇讲过，Code Mode（代码模式）会把多次工具往返收进一个程序，再通过绑定来调度当前可见的工具。如果一项任务要有自己的 Context、持久身份，还要能单独控制，你就不能只盯着工具调用了，得看子 Agent 怎样连成一棵 Thread 树。

每个 Codex 子 Agent 都有自己的 ThreadId（线程 ID）、Session、Turn（回合）、Context 和 Rollout（运行轨迹）。整棵树只共用一套控制面。这个控制面就是 AgentControl，它管容量，记父子关系，也负责传消息、等待、打断和回收节点。

```
根 Thread
  ├── 子 Thread：检索实现
  │     └── 孙 Thread：核对测试
  └── 子 Thread：检查安全边界

共享：AgentControl、容量约束、父子身份
独立：Session、Turn、Context、Rollout、停止状态
```

为什么不直接 join_all() 几个模型 Future

普通 Future（异步计算）只能帮你并发等待，却回答不了后面的事：子任务的历史写到哪里？父 Agent 到了下一个 Turn 怎样接着推进它，父任务取消后又是否要中断整棵子树？完成通知又该怎样找到当初那次委派？Thread 树给每项工作留下持久身份，再把这些需求交给明确的控制操作。

第 1 站：一棵根线程树只共享一个控制面

源码：查看 AgentControl 的作用域

```
/// An `AgentControl` instance is intended to be created at most once
/// per root thread/session tree and shared with every spawned sub-agent.
pub(crate) struct AgentControl { ... }
```

调用者：根 Session 创建控制面，Spawn/Fork/通信工具复用它。

输入：根线程配置、Agent Registry 和容量限制。

状态变化：注册和更新整棵树的 Live Agent 节点。

返回：供子 Agent 工具调用的共享控制接口。

下一站：Spawn 在控制面中预留槽位并创建子 Thread。

这两件事不能混。整棵树可以共用控制面，各个节点却仍有自己的 Context，AgentControl 也不会把所有子 Rollout 都塞进父 Agent 的模型历史。

第 2 站：活动节点把身份、元数据和状态分开

源码：查看 Agent 节点类型

```
pub(crate) struct LiveAgent {
    thread_id: ThreadId,
    metadata: AgentMetadata,
```

```
    status: AgentStatus,
}

pub(crate) struct ListedAgent {
    agent_path: AgentPath,
    agent_status: AgentStatus,
}
```

调用者：AgentControl 的列表、等待、通知和清理逻辑。

输入：新子 Thread 与父子 Agent Path。

状态变化：节点状态从创建、运行、Idle 到终态迁移。

返回：面向调用者的路径与状态快照。

下一站：父 Agent 可以 Wait、Follow-up、Interrupt 或读取结果。

agent_path 告诉你节点在树上的位置，thread_id 则标出它的持久身份，status 只记录这个节点眼下处于什么状态。列表序号不能代替它。

Spawn 先预留容量，再开始昂贵创建

第 3 站：容量和继承在创建前确定

源码：查看 Spawn 控制

```
let agent_max_threads = config.effective_agent_max_threads(...);
let mut reservation = self.state.reserve_spawn_slot(...)?;
let inheritance = SpawnAgentThreadInheritance { ... };
```

调用者：Spawn Agent 工具 Handler。

输入：父 Thread、父 Turn、Prompt、Agent 配置和最大线程数。

状态变化：原子预留一个树容量槽；构造明确的历史/配置继承描述；成功后发布子 Thread。

返回：子 ThreadId 或类型化启动失败。

下一站：子 Session 接收初始 Prompt，父端获得可等待的节点。

系统先占住名额，免得两个并发 Spawn 都看见「还剩一个名额」，随后一起挤过上限。失败就得退还名额。只要发布成功，子 Thread 就进入正常的节点生命周期，不能再当成从未创建过。

发送消息与打断是不同控制动作

第 4 站：是否触发新 Turn 是消息本身的契约

源码：查看通信与中断

```
let parent_turn_id =
    parent_turn_id.filter(|_| communication.trigger_turn);

Op::InterAgentCommunication { communication }
Op::Interrupt
```

调用者：Follow-up、Send Message 和 Interrupt 工具。

输入：发送者/接收者 ThreadId、内容、是否触发 Turn。

状态变化：普通消息只进入通信记录；触发型消息可以唤起接收 Thread 的新 Turn；Interrupt 走取消路径。

返回：提交结果和后续 Agent Events。

下一站：接收 Session 处理消息，或 RunningTask 响应取消。

所以，「通知已送达」不代表子 Agent 已经跑了新一轮，Interrupt（中断）也不会删除 Thread，原来的历史和已经提交的输出都还在。

完成通知是父端输入，不是子结果本身

子 Agent 结束后，父 Rollout 会收到一条能关联到它的通知，子 Agent 自己的完整 Transcript（对话记录）仍保存在另一条 Rollout 里。

源码：查看父子 Transcript 与通知测试

```
assert!(rollout.contains("<subagent_notification>"));
assert_ne!(parent_transcript_path, agent_transcript_path);
```

父 Agent 可以看着通知继续汇总。你若要复核工具具体跑过什么，就得去读子 Thread。父端改不了这份记录。一句「审查通过」，也抹不掉子端已经记下的错误和停止原因。

何时应该创建子 Agent

如果子任务要占用一段独立的长 Context，要用不同范围的工具，要并行探索，或者以后还得单独恢复，就适合创建子 Agent。若只是读两个小文件，结果又必须带回眼前的全部细节，甚至创建开销比任务本身还大，那就别创建，因为多 Agent 改变的是 Context 和控制结构，不会凭空提高质量。

到这里，Thread 树已经把子任务是谁、处于什么状态、该怎样控制讲清楚了。可同一套核心还要接入 CLI、Exec、App Server 等产品入口，下一篇就去核对各个接口面怎样投影事件，又为 Trace 和 Eval 留下了哪些证据。

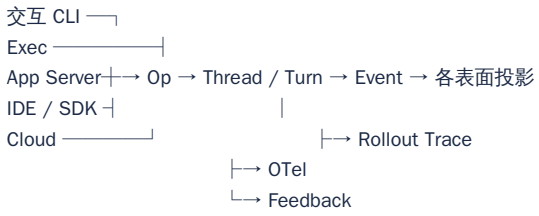
下一篇：产品表面、Trace 与 Eval 接缝。

多种产品表面怎样共享核心，又怎样留下可评测证据

返回 Codex 课程地图

上一篇顺着 Thread 树，弄清了子任务是谁、处于什么状态，又该怎样控制。现在走出这棵树，看看同一套核心怎样接住不同产品入口。

交互 CLI、无头 Exec、App Server、IDE（集成开发环境）、MCP Server 或 Cloud 入口都能驱动 Codex 核心。它们共用 Thread、Turn（回合）、模型和工具实现，可到了传输方式、错误格式、停止语义和对外事件这一层，各入口并不完全一样。



第 1 站：CLI 只是入口分派器之一

源码：查看 CLI 子命令

```
enum Subcommand {
    Exec(ExecCli),
    McpServer(...),
    AppServer(...),
    Resume(...),
    Fork(...),
    Cloud(...),
}
```

调用者：进程 main() 解析命令行。

输入：子命令、配置覆盖和终端环境。

状态变化：选择交互、无头、Server 或管理流程。

返回：对应表面的退出码和输出。

下一站：表面创建 Thread/Session，或连接既有 Thread。

同一个 Turn Error 到了 Exec，可能表现为非零 Exit Code，所以你测过这种映射，仍不能断定另一种也对。到了 App Server，它也可能变成 JSON-RPC Error 或 Event。两种映射要分开测。

App Server 把核心变成多传输服务

第 2 站：输入、处理和输出具有独立异步循环

源码：查看 App Server 传输入口

```
// AppServerTransport 可以是 Stdio、UnixSocket 或 WebSocket。
// 主入口为传输建立独立读写与处理任务。
```

源码：查看 App Server 主循环

```
// 输入消息进入处理循环，核心事件再由输出循环发送。
```

调用者：CLI 的 AppServer 子命令或嵌入宿主。

输入：所选 Transport 与客户端请求流。

状态变化：把协议请求映射为核心操作；把 Events 投影为客户端通知。

返回：协议响应、通知和连接终态。

下一站：客户端更新自己的任务视图或提交下一项操作。

Transport（传输层）连上了，不代表 Thread 已经创建。服务端接受请求，也不代表 Turn 已经跑完，Client 必须分别拿请求 ID、ThreadId（线程 ID）和 TurnId 对上这三个层次。这几层不能混。

Trace、Telemetry 与 Feedback 分别保存什么

Rollout Trace：重建 Harness 内部因果

源码：查看 Trace 模块

```
//! Trace bundle format, writer, and reducer for Codex rollouts.

pub use raw_event::RawTraceEvent;
pub use writer::TraceWriter;
```

调用者：Session/Turn 事件记录与诊断工具。

输入：带关联身份的 Raw Events。

状态变化：追加 Trace Bundle；Reducer 可派生便于查询的状态。

返回：可重建调用链的事件集合或派生视图。

下一站：调试器、可视化或 Evaluator 消费。

Reducer 给出的只是派生视图，不能拿它覆盖原始事件。Writer 没有启用或者写入时失败了，你都可能看不到 Trace（执行轨迹）。这不能反推执行没发生。

OTel：观察性能与运行指标

源码：查看遥测导出

```
pub use ... SessionTelemetry;
pub use ... RuntimeMetricsSummary;
pub use ... inject_span_w3c_trace_headers;
```

OTel 适合用来查延迟、错误率、调用关系和资源消耗，而采样会漏掉一部分信息，Exporter 故障和脱敏也会留下缺口。因此，看不到 Error Span，不能据此判定任务成功。

Feedback：用户提交诊断与主观信号

源码：查看 Feedback Snapshot

```
pub struct CodexFeedback { ... }
pub fn snapshot(...) -> FeedbackSnapshot { ... }
```

Feedback（反馈）可以带上日志和附件，帮你定位问题，不过谁愿意提交反馈、用户当时处在什么语境，都会影响这份材料，其中还可能混有敏感数据。它不能自动变成 Reward。也不能充当发布门禁。

一个独立 Eval 应怎样使用这些通道

以「修复失败测试」为任务：

1. 固定仓库基线、用户输入、模型、Codex 配置和表面版本。
2. 通过 Exec 或 App Server 提交一次任务，并保存 ThreadId/TurnId。
3. 收集 Rollout、Tool Trace、文件差异、停止原因、Token 和时长。
4. 在 Codex 运行之外执行目标测试与静态检查。
5. Evaluator 依据测试、Diff 范围和任务约束给出 Score 与失败原因。

Assistant（助手）的最终文本、Exec Exit Code、OTel、Trace 和 Feedback 都能提供证据，真正给出评分的应是明确设置的 Evaluator。如果还想拿这些数据训练模型，就要另设 Reward Adapter（把原始信号转换成训练奖励的版本化规则），把样本怎么选、标签表示什么写清楚，并保留一份独立发布集。

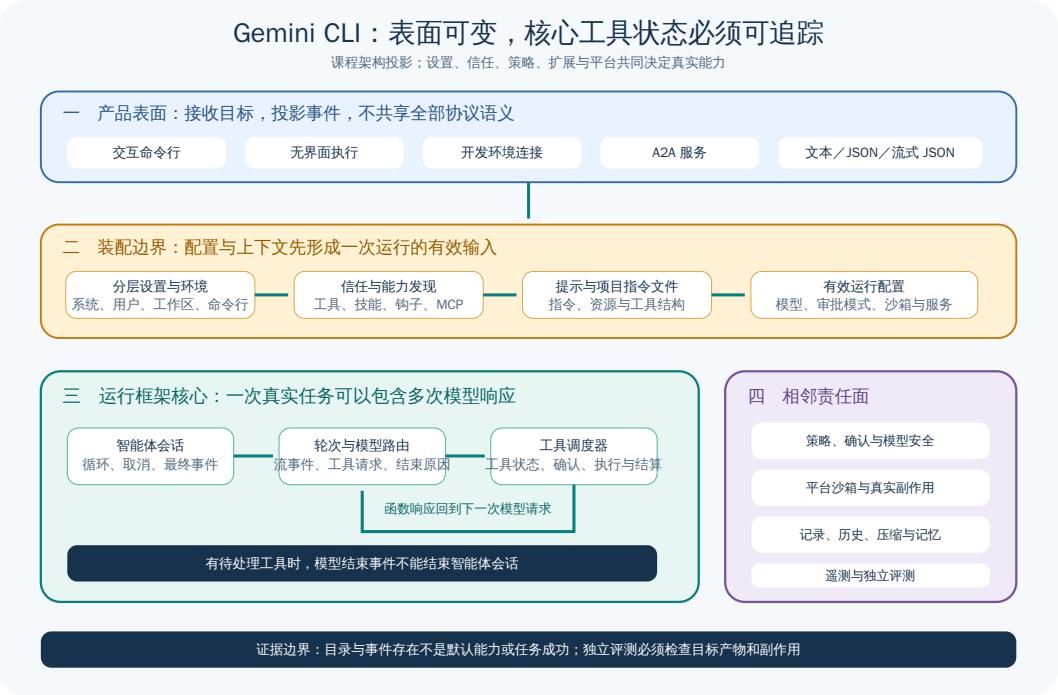
读完这八篇，你已经能从配置出发，一路追过 Thread、Turn、工具和执行边界，再读到 Rollout、子 Agent 以及各个入口留下的证据。要做横向比较，手边还缺另一套 Agent Harness 的同类证据，先读完另一条课程，再回课程地图对照会更稳。

到这里可以回到 Codex 课程地图 自己复核永久链接，或进入 Gemini CLI 课程 比较 Config 和 Scheduler，再看 Tool 与 Confirmation 这两个概念在 Tool Confirmation 里怎样配合。

Gemini CLI 源码课程

返回学习入口

Gemini CLI 用交互表面、Core、Turn、Scheduler（调度器）、工具、Policy、Confirmation 和 Session 接起一条 TypeScript 调用链。模型发出工具请求后，你可以顺着这条链往下追，看 Agent Harness（智能体框架）怎样调度请求、确认权限并交给工具执行，再怎样把结果送入下一轮。阅读时不用急着记住每个类型，先盯紧一条请求携带的数据，看它如何从配置和上下文出发，经过模型路由与事件流变成需要授权的工具调用，最后带着执行结果写回历史。只要数据仍在链上流动，后面遇到 Registry、Hook 或 A2A，即使一时不熟，也能把它们放回正确的位置。



这条课程适合谁

如果你已经理解最小 Agent Loop（智能体循环），想继续弄清调度器怎样接住请求、系统又在何处询问用户，就适合从这条路线往下读。两代实现不能混。课程会把旧 Agent Session 与新的 Turn/Scheduler 路径分开讲，免得你把它们硬接成一条源码里并不存在的主链。

锁定来源

这门课程锁定 Gemini CLI 提交 5411f113cafae26161b4969b0237b8e1e024e2c2，正文里的路径、类型和测试因此都能放在同一个版本里核对。Gemini CLI 还在演进，如果把当前主链与不同版本的 Legacy Agent 证据接在一起，你很容易拼出一条源码从未实现过的调用链。遇到课程描述和本地新版不一致时，先回到这个提交确认差异，再判断是版本变了，还是自己读错了：

- Turn 实现
- Scheduler 实现
- Legacy Agent Session
- Legacy Agent Session 测试

先看一项任务

用户输入进入 Core 后会形成 Turn，模型流随后可能吐出文本、工具请求或错误。模型一旦发出工具请求，Scheduler 就会接过去，等 Policy 与 Confirmation 作出判断后再调用具体工具，执行结果则写回会话，并进入后续模型输入。

CLI 输入
→ Core / Turn
→ Gemini 请求与流
→ Scheduler
→ Policy / Confirmation
→ Tool 执行
→ Session 历史与下一轮

课程会分别标出哪些证据属于当前 Turn/Scheduler 主链，哪些来自 Legacy Agent Session。先确认它走的是哪条路径。这样读到不同实现时，你就不会把两个时期的对象硬拼成一套源码里不存在的流程。

仓库地图

区域	首轮关注点
packages/cli	用户输入和 Core 的连接
packages/core/src/core	Turn、模型流和主要状态
packages/core/src/scheduler	工具调度、完成和取消
Tools 相关目录	工具声明、调用和结果类型
Policy、Safety、Confirmation	请求怎样获得允许或用户确认
Session 与压缩相关目录	历史怎样保存和缩短
Hooks、Agents、Skills、MCP	核心之外的扩展接缝

第一次读时，可以先放下终端渲染、主题、认证提供商和所有遥测导出，这些分支固然重要，但你还没看清主链就钻进去，只会被细节带跑。先把一次 Turn 走完。

三层读法

Starter：读前三篇，认识 Turn、模型流、Scheduler 和工具结果回填。

Builder：继续追踪 Policy、Confirmation、Session、压缩与扩展介入点。

Maintainer：核对新旧路径、取消与错误事件，并判断 Telemetry 能支持哪些结论。

阅读顺序

1. 配置、Prompt 与 Context 先从 Settings 的信任过滤与合并开始，追到系统指令、首条用户上下文和活动工具；它们共同形成模型请求；这为下一篇进入 Turn 备齐输入。
2. Turn、Scheduler 与路由 接住已经成形的请求，区分 Model Router 的模型决定、Turn 的流式事件和 Scheduler 的调用状态，再把问题推进到每个工具请求如何完成。
3. 工具生命周期 沿 Scheduler 展开 Registry、活动工具、Invocation 与 Executor，辨清工具协议成功与任务完成的区别之后，下一步自然要核对执行是否获得授权。
4. Confirmation、Policy、Safety 与 Sandbox 四条边界的位置不同。顺着工具执行前的授权问题，分开 Policy Decision、用户确认、模型 Safety 与平台 Sandbox，然后才能讨论这些结果如何进入 Session。
5. Session、历史、压缩与 Memory 接住工具结果和后续请求，划清运行时历史、模型投影、JSONL 记录、Compression 与跨会话 Memory，为扩展机制介入 Context 留出边界。
6. Agents、Hooks、Skills 与 MCP 从这些边界继续追踪 Extension 如何刷新多个 Registry，以及 Hook、Skill、MCP 和 Agent 各自怎样进入运行时，随后再看不同表面如何投影同一运行。
7. CLI、输出与协议表面 把 Core Events 投影到交互 CLI、非交互格式、IDE 与 A2A，明确各表面的事件和停止语义后，最后才有条件判断 Telemetry 能解释什么。
8. Telemetry、错误与 Eval 接缝 汇合前面各层留下的路由、确认、工具与输出证据，区分运行观测和独立评分，完成从执行链到 Eval Artifact 的复盘。

用贯穿任务复盘

复盘时从 CLI 输入起步，先看 Config 和 Prompt 怎样拼出 Turn，再跟着模型流里的 Tool Call 进入 Scheduler，找出 Policy 与 Confirmation 在哪里插手，然后看 Tool Result 怎样写回 Session，以及压缩或 Memory 会如何改变后续 Context。把这条链走完以后，再分别列清 Tool Success、Turn 结束和任务通过各要什么证据，这三种「完成」就不会混在一起。

引用 Legacy Agent Session 时，必须说明它走的是另一条可以单独核对的路径，否则读者会误以为新旧对象同时存在于一条运行链上。不要悄悄把它接进当前 Turn/Scheduler 主链。

完成课程后应该能回答

Turn 消费模型流时维护了什么状态；

Scheduler 怎样接收工具请求并等待结果；

Policy、Confirmation 和执行环境的判断顺序；

Session 历史怎样进入下一轮或压缩；

Hooks、Skills 与 MCP 能改变哪些阶段；

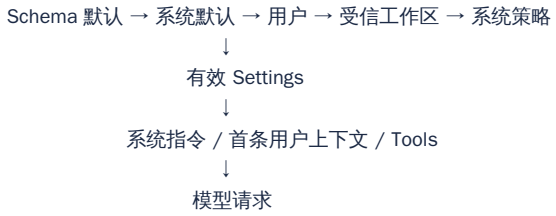
新旧 Agent 路径的证据应如何分开。

从第一篇开始：配置、Prompt 与 Context

Settings 怎样变成一次模型请求的 Prompt 与 Context

返回 Gemini CLI 课程地图

课程地图先讲 Config/Prompt，再讲 Turn，是为了顺着程序真正组装请求的次序往下读。程序得先把有效配置、项目上下文、工具声明和历史拼好，才能发出一次模型请求。Gemini CLI 依次合并 Schema 默认值、系统默认值、用户设置、工作区设置和系统设置，工作区是否受信则直接决定这一层能不能并进来。五层 Settings 合完后，Core Config 再接着整理系统指令、项目上下文、工具声明和历史。



第 1 站：未受信工作区先被过滤，再参与合并

源码：查看 Settings 合并

```
const safeWorkspace = isTrusted ? workspace : ({} as Settings)
return customDeepMerge(
  schemaDefaults,
  systemDefaults,
  user,
  safeWorkspace,
  system,
)
```

调用者：CLI 设置加载器计算本次运行的合并视图。

输入：各层 Settings 与工作区信任状态。

状态变化：未受信工作区变成空贡献；其余层按顺序深合并。

返回：merged Settings。

下一站：CLI Config 把字段传给 Core Config、Policy、Sandbox 和 UI。

顺着这个合并顺序看，系统设置能覆盖工作区设置，工作区设置又能覆盖用户设置，不过 Deep Merge 遇到数组、对象或特殊字段时，仍会按各自的 Schema 规则处理。一个字段说明不了全部。

信任切换为什么需要重新计算

LoadedSettings 不能把未受信工作区的内容直接丢掉，否则用户后来确认信任，程序还得重新读取磁盘。它会把原始工作区文件留在内部，但对外只给出过滤后的安全视图。这样切换信任状态时，程序就能立即重算配置。

第 2 站：原始文件与有效工作区分开保存

源码：查看 LoadedSettings

```
this._workspaceFile = workspace
this.workspace = isTrusted
  ? workspace
  : this.createEmptyWorkspace(workspace)

setTrusted(isTrusted) {
```

```
  // 更新安全视图并重新计算 merged
}
```

调用者：目录信任 UI 或启动流程。

输入：新的信任决定。

状态变化：切换 Workspace 安全视图，重新生成合并 Settings。

返回：后续消费者读取到新的有效配置。

下一站：Extension、MCP、Hooks 和 Prompt 根据新配置刷新。

这里用信任状态拦住工作区配置，是因为工作区文件不只会改变主题，还能影响工具、扩展和外部进程。程序先过滤再合并，就是为了避免工作区配置赶在用户确认之前改动可执行能力，这道门必须守住。

项目上下文不全放在 System Instruction

Core Config 会把全局 Memory 和用户级项目 Memory 放进系统指令，再把 Extension 与当前项目 Memory 放进首条用户消息。这样一来，不同来源各自落在哪个优先级就很清楚。你也能判断哪些部分可以复用缓存，而经常变化的工作区文本不会挤进稳定的系统前缀。

第 3 站：按来源选择注入层

源码：查看项目指令分层

```
// Global memory and user project memory go in the system instruction.
// Extension and project memory are placed in the first user message.
```

调用者：Core Config 初始化 Prompt Provider 与 Gemini Client。

输入：全局、用户、Extension 和工作区项目上下文。

状态变化：按来源分组，并生成不同角色的 Prompt 片段。

返回：System Instruction 与首条 User Context。

下一站：Turn 把历史和当前输入接在这些片段之后。

GEMINI.md 出现在仓库里，只能证明文件存在，还说明不了它是否进入模型请求。你还得确认目录已经受信、配置找到了这个文件名、程序成功读出了内容，最后再看它究竟被放进请求里的哪个角色。

Tool Registry 也参与模型输入

模型只能请求当前 Function Declarations 里列出的工具。Plan Mode、Extension 或 Policy 一变，Tool Registry（工具注册表）投给模型的活动工具也可能跟着变，所以 Prompt 不能只当成一串文本来看。

源码：查看 Prompt Provider 的工具渲染测试

```
// 覆盖多个 Context 文件名。
// 核对 Plan Mode Prompt 列出 ToolRegistry 中的活动工具。
```

排查模型为什么会「编造工具」时，你要保存这次请求真正带上的 Function Declarations。只看源码目录里有哪些 Tool Class，无法证明模型当时真的看见了它们。候选不等于可用。

一个失败测试任务的输入组成

用户在受信仓库里请求修复测试时：

1. 用户 Settings 选择模型和输出方式；系统策略仍能覆盖危险字段。
2. 工作区 Settings 与 GEMINI.md 只有在信任后生效。
3. 全局 Memory 进入系统指令，项目上下文进入首条用户消息。
4. Tool Registry 投影当前活动工具。
5. Session History 与本次用户输入组成 Turn 的模型请求。

走到这里，程序已经把有效 Settings、分层 Context、活动工具和 Session History 拼成了一次模型请求。下一篇跟着这次请求进入 Model Router、Turn 与 Scheduler，看模型怎样响应、工具怎样接手，以及会话到底在什么条件下结束。

下一篇：Turn、Model Router 与 Scheduler。

Turn、Model Router 与 Scheduler 怎样分工

返回 Gemini CLI 课程地图

上一篇拼好的模型请求进入执行主链后，会依次交给三个模块：Model Router（模型路由器）替这次请求选模型，Turn 消费 Gemini 流并发出事件，Scheduler（调度器）则逐个管理 Tool Call，带它们走完验证、审批、执行和结算。模型响应结束不代表任务结束。

工具成功不等于 Turn 结束。

```
用户输入
  ↓
Model Router → 选择模型与回退原因
  ↓
Turn → Gemini 流 → 文本 / ToolCall / Finished
                ↓
            Scheduler
                ↓
        Tool Results 回到下一次请求
```

第 1 站：路由是一组有顺序的 Strategy

源码：查看 Model Router 组装

```
strategies.push(new FallbackStrategy())
strategies.push(new OverrideStrategy())
strategies.push(new ApprovalModeStrategy())

return new CompositeStrategy(
  [...strategies, terminalStrategy],
  'agent-router',
)
```

调用者：Turn 发起模型请求前的路由服务。

输入：有效 Config、Approval Mode、显式覆盖和可用模型。

状态变化：按顺序尝试 Strategy，并记录覆盖或回退元数据。

返回：本次请求使用的模型决定。

下一站：Gemini Client 建立流式响应。

同一 Session 里的不同请求可能因为回退或模式切换选中不同模型，所以评测时既要保存启动配置里的首选模型，也要记下每次请求实际走到了哪个模型。两者不能混用。

第 2 站：Turn 只有看到真实 FinishReason 才发 Finished

源码：查看 Turn 的结束事件

```
const finishReason = resp.candidates?.[0]?.finishReason
if (finishReason) {
  yield {
    type: GeminiEventType.Finished,
    value: { reason: finishReason, usageMetadata: resp.usageMetadata },
  }
}
```

调用者：Turn 对 Gemini Response Chunk 的迭代器。

输入：Provider 返回的流式 Candidate Chunk。

状态变化：累积文本与 Tool Request；仅在响应提供 FinishReason 时发 Finished。

返回：供 Agent Session 或 UI 消费的 Gemini Events。

下一站：Agent Session 先检查是否还有待处理 Tool Calls，再决定是否真正结束。

Finished 只说明「这次模型响应已经给出结束原因」，如果同一份响应里还有 Tool Calls，Harness 就得先执行工具，把结果送回模型，后面的分支也得继续处理这一批调用。它只结算当前响应，整个任务还可能继续。

第 3 站：工具请求优先于 Agent Session 结束

源码：查看 Legacy Agent Session 的分支

```
if (toolCallRequests.length === 0) {
  this._finishStream(mapFinishReason(finishedReason))
  return
}

const completedToolCalls = await this._scheduler.schedule(...)
```

调用者：Legacy Agent Session 消费完一批 Turn Events 后。

输入：本次响应收集的 Tool Requests 与 FinishReason。

状态变化：没有工具时结算 Agent Stream；有工具时交给 Scheduler 并保持会话活动。

返回：Agent End，或一批 Completed Tool Calls。

下一站：Completed Calls 变成 Function Responses，进入下一次模型请求。

这里把 Legacy AgentSession 和新的 Turn/Scheduler 分开讲，因为源码同时留着两代路径时，你不能看到一条路径怎么做，就认定另一条也完全一样。读代码时先找清调用者走的是哪条路，再跟着这条路上的状态变化和返回值往下看。

Scheduler 的状态属于单个调用

一个 Tool Call 会依次经过 Validating → Scheduled → AwaitingApproval → Executing → Success/Error/Cancelled，但每个状态都只记在这一条 Call 上。同一批里的其他调用完全可能停在别的阶段，所以你分析整批请求时仍得逐条核对，不能拿一个笼统的批次状态盖住它们。

第 4 站：Scheduler 维护显式调用状态机

源码：查看 Tool Call 状态类型

```
type ToolCall =
  | ValidatingToolCall
  | ScheduledToolCall
  | WaitingToolCall
  | ExecutingToolCall
  | SuccessfulToolCall
  | ErroredToolCall
  | CancelledToolCall
```

调用者：Scheduler 接纳、确认、执行与取消流程。

输入：带 Call ID、工具名和参数的请求。

状态变化：每次转换生成更具体的联合类型成员。

返回：最终 CompletedToolCall。

下一站：Agent Session 将结果翻译为模型可读 Function Response。

联合类型直接告诉 TypeScript：「处于 AwaitingApproval 的调用一定带着 Confirmation Details，到了 Success 就一定有 Result。」这样写能减少可选字段随意组合，避免程序拼出根本不该存在的状态。阅读时先找到当前对象属于哪个联合类型成员，再看这个成员已经带上了哪些字段。

FinishReason 也不能简单折成成功/失败

STOP 会映射成 completed，MAX_TOKENS 说明预算已经耗尽，SAFETY、RECITATION 等原因会归进 refusal，Malformed Function Call 等情况则算 failed。如果界面只写「流结束」，恢复流程就拿不到自己需要的分类，所以记录里必须保留这些映射结果。

源码：查看 FinishReason 映射

```
// STOP -> completed
// MAX_TOKENS -> max_budget
// SAFETY / RECITATION -> refusal
// MALFORMED_FUNCTION_CALL / OTHER -> failed
```

到这里，我们已经分清 Router 选了哪个模型、Turn 发出了什么事件，以及 Scheduler 怎样记录每条调用的状态。下一篇继续跟着 Scheduler 收到的 Tool Call 往下走，看 Registry、Invocation、Executor 与 Function Response 怎样连起来，工具也正是在这条链路上真正开始做事。

下一篇：工具注册与完整生命周期。

工具从「已知定义」到「模型结果」的完整生命周期

返回 Gemini CLI 课程地图

上一章跟着一次交互走过 Model Router、Turn 和 Scheduler（调度器）：Model Router 选择模型，Turn 消费 Gemini 流并发出事件，工具请求则交给 Scheduler 处理，所以会话不会在工具做完之前结束。继续沿着验证、审批、执行和结算往下看，你至少要分清三层工具信息：Registry（注册表）收着所有已知定义，活动视图决定这一轮让模型看见哪些工具，Scheduler 再逐条记录实际调用走到了哪一步。

Tool Class 只是候选。

```
已知工具定义
    ↓ 过滤 / 模式 / 配置
活动工具与 Function Declarations
    ↓ 模型生成请求
Scheduler 调用状态
    ↓ Policy / Confirmation / Executor
模型内容 + UI 内容 + 可选输出文件
```

第 1 站：Registry 保留被排除的已知工具

源码：查看工具目录字段

```
// Includes tools which are currently not active.
private allKnownTools: Map<string, AnyDeclarativeTool> = new Map()

// Excluded tools remain registered so they can be enabled later.
```

调用者：Core Config 初始化内置工具，Extension/MCP 刷新外部工具。

输入：Declarative Tool Definition 与来源信息。

状态变化：按工具名登记候选定义，即使当前被排除。

返回：可重算活动视图的 Registry。

下一站：Prompt Provider 查询 Function Declarations，Scheduler 按名查活动工具。

Registry 会保留那些已经注册、眼下却没启用的工具，因此模式切换后可以直接重新启用，不必重启整个进程。不过，诊断信息必须说清展示的是「已知工具」还是「当前可用工具」。两者不能混。

第 2 站：所有公开查询经过活动过滤

源码：查看活动工具查询

```
private getActiveTools(): AnyDeclarativeTool[]
getFunctionDeclarations(modelId?: string)
getAllTools()
getTool(name)
```

调用者：Prompt 构造、Scheduler 和工具列表界面。

输入：当前模式、排除配置与可选 Model ID。

状态变化：无持久修改；从已知表投影活动集合。

返回：模型 Schema、活动工具数组或按名定义。

下一站：模型生成 Function Call，Scheduler 再按同一活动视图查找。

即使模型已经看过 Schema，配置仍可能在工具执行前刷新，所以 Scheduler 真正准备执行时还得再查一次 Registry，按当时的活动视图找工具，不能从请求生成那一刻起就一直握着同一个 Handler 引用。

第 3 站：不存在与参数无效在副作用之前失败

源码：查看 Scheduler 验证

```
const tool = toolRegistry.getTool(request.name)
if (!tool) {
  // TOOL_NOT_REGISTERED
}

const invocation = tool.build(request.args)
// 参数错误 -> INVALID_TOOL_PARAMS
```

调用者：Scheduler 接纳模型 Tool Request。

输入：工具名、JSON 参数和 Call ID。

状态变化：从 Validating 转到 Scheduled，或直接生成 Error Call。

返回：类型化 Invocation 或无副作用失败。

下一站：Policy/Confirmation 检查 Invocation 的具体动作。

工具根据参数建出 Invocation（调用实例）时，也在划安全边界，因为这一步会把路径、命令和资源整理成规范形式。Policy 应该检查整理后的具体动作，不能只盯着模型吐出的原始 JSON 文本。

第 4 站：Executor 把实现结果归约为统一终态

源码：查看 Tool Executor 结算

```
if (signal.aborted) return createCancelledResult(...)
if (toolResult.error === undefined) return createSuccessResult(...)
return createErrorResult(...)
```

调用者：获得批准的 Scheduled Call。

输入：Tool Invocation、AbortSignal、Hooks 与输出处理器。

状态变化：进入 Executing，运行具体 Tool，随后归约为 Success/Error/Cancelled。

返回：Completed Tool Call。

下一站：Scheduler 通知 UI，Agent Session 构造 Function Response。

Success 只说明工具实现没有返回 Tool Error。任务未必完成。比如 Shell 确实跑起了 npm test，可进程退出码仍是 1，这时工具协议可能正常交回一份带失败输出的结果，Agent 还得继续修复。

一份结果为何要拆成三种内容

源码：查看模型内容与显示内容

```
{
  responseParts: response,
  resultDisplay: toolResult.returnDisplay,
  outputFile,
  errorType,
}
```

responseParts 回到模型，应控制大小并保留行动所需信息；

resultDisplay 面向终端 UI，可以有颜色、摘要或进度；

outputFile 保存过大的完整输出，避免把全部日志塞进 Context。

这三份内容各有用处：一份回到模型，一份显示在终端，还有一份把完整输出存进文件，所以不能拿其中一份冒充另外两份。Evaluator 如果要看完整测试日志，就该读取明确保存下来的 Artifact，不能拿 UI 里截断过的摘要顶替。

一个 Tool Call 的核对清单

核对 Trace 时，你要顺着调用发生的次序，找到活动 Schema、模型原始请求、规范化 Invocation、Policy Decision、Confirmation、Execution Start、Tool Result 和 Function Response。中间缺了任何一项，最后那张结果卡片都还原不了整条控制链。

工具调用走到 Policy Decision 和 Confirmation 时，关注点就从工具怎么定义、筛选和执行，转到了这次动作能不能获得授权。下一篇会把几类信号逐一分开：PolicyEngine 给出允许、拒绝或询问，Confirmation Bus 关联用户作出的决定，模型用 SAFETY 说明这次生成为什么结束，SandboxManager 则生成平台实际采用的执行规格。这些信号看起来都在谈安全，其实管的不是同一件事。

下一篇：Confirmation、Policy 与 Sandbox。

Confirmation、Policy、模型 Safety 与 Sandbox 是四条边界

返回 Gemini CLI 课程地图

上一章跟着一个 Tool Call 走过 Registry（注册表）、活动视图和 Scheduler，又看了系统怎样规范化 Invocation（调用实例）并收拢执行结果，最后停在 Policy Decision 与 Confirmation。走到授权这一步，你会碰到四个都像是在管「安全」的名字，可它们各管一段：PolicyEngine 判断工具动作该允许、拒绝还是询问，Confirmation Bus 把用户决定配回这次调用，模型用 SAFETY 说明为什么停止生成，只有 SandboxManager 才会把文件、网络、环境和进程限制写成平台真正执行的规格。

```
Tool Invocation
    ↓
PolicyEngine：ALLOW / DENY / ASK_USER
    ↓
Confirmation Bus：请求与响应配对
    ↓
SandboxManager：平台命令与权限
    ↓
Process / Tool 执行

模型 SAFETY：属于上游生成响应，不在这条执行授权链中
```

第 1 站：非交互默认拒绝，交互默认询问

源码：查看 Policy 默认值

```
this.defaultDecision =
  config.defaultDecision ??
  (this.nonInteractive
    ? PolicyDecision.DENY
    : PolicyDecision.ASK_USER)

this.sandboxManager =
  config.sandboxManager ?? new NoopSandboxManager()
```

调用者：Core Config 创建 PolicyEngine。

输入：规则集、是否非交互、默认决定和 Sandbox Manager。

状态变化：确定没有规则命中时的行为。

返回：后续 check() 使用的 PolicyEngine。

下一站：Confirmation Bus 对具体 Invocation 查询决定。

非交互任务里没人能点击确认，如果默认走 ASK_USER，任务就会一直等下去，因此系统直接选择 DENY。这里还要看懂 NoopSandboxManager 这个名字：Policy（策略）和 Sandbox 可以分开配置，前者决定没有规则命中时该怎么办，后者则可能明确选用 Noop 实现，完全不加隔离。

第 2 站：Message Bus 不自己发明授权

源码：查看确认路由

```
const { decision } = await this.policyEngine.check(...)

switch (decision) {
```

```
  case PolicyDecision.ALLOW:
  case PolicyDecision.DENY:
  case PolicyDecision.ASK_USER:
}
```

调用者：Scheduler 准备执行需要确认的 Invocation。

输入：工具动作、会话语境和 Confirmation Request。

状态变化：ALLOW 直接继续；DENY 生成拒绝；ASK_USER 发布请求并等待关联响应。

返回：本次调用的确认结果。

下一站：Executor 或取消路径。

Confirmation ID 必须关联 Call ID，这样系统才能把用户的每次响应配回那一条具体调用。用户批准的是另一条命令时，当前正在等待的调用绝不能把这个决定拿走。调用身份不能串。

第 3 站：只有启用 Sandbox 才选择平台后端

源码：查看 Sandbox Manager Factory

```
if (sandbox?.enabled) {
  // win32 -> WindowsSandboxManager
  // linux -> LinuxSandboxManager
  // darwin -> MacOsSandboxManager
}
return new NoopSandboxManager(options)
```

调用者：Core 启动与配置刷新。

输入：Sandbox 开关、平台和选项。

状态变化：实例化平台后端或明确的 Noop 实现。

返回：Policy/Executor 使用的 SandboxManager。

下一站：具体工具请求调用 Manager 生成执行规格。

Policy 给出 ALLOW、Sandbox Manager 却选了 Noop，这是一种明确存在的配置组合，日志不能把它记成「已隔离」，因为动作虽然获准执行，进程实际上没有套上任何 Sandbox 后端。反过来也一样，即使启用了 Sandbox，工具动作仍要先过 Policy。这两层不能混。

平台后端必须产生真实执行变化

Linux

源码：查看 Bubblewrap 包装

```
const bwrapArgs = await buildBwrapArgs(...)
bwrapArgs.push('--seccomp', '9')
// 执行 bwrap，并把原命令放在包装参数之后
```

macOS

源码：查看 Seatbelt 包装

```
const sandboxArgs = buildSeatbeltProfile(...)
return {
  program: '/usr/bin/sandbox-exec',
```

```
  args: ['-f', tempFile, '--', finalCommand, ...finalArgs],
}
```

调用者：执行工具在启动进程前调用当前平台 Manager。

输入：原程序、参数、工作目录和权限选项。

状态变化：生成 Bwrap/Seatbelt/Windows 包装与临时策略文件。

返回：真正交给 Process Host 的程序和参数。

下一站：进程启动；初始化失败进入 Tool Error。

核对 Sandbox 时别只看 Config 开关，你还得跟着平台 Manager 生成的执行规格往下查，看看最终的 Program/Args、临时 Profile（配置档）、网络和挂载参数，以及启动时抛出的错误，确认 Process Host 收到的确实是包装后的命令。

模型 Safety 属于另一条链

模型可能用 SAFETY FinishReason 拒绝继续生成，这件事发生在 Tool Invocation 出现以前，只能说明 Provider（模型提供商）拦下了这次内容。它既不能告诉你本地命令有没有经过 Policy 和 Sandbox，也不能替你核对后面的执行授权链。

做评测时，模型拒绝、用户拒绝工具和 Sandbox 初始化失败必须分成三类错误，因为它们发生的位置不同，修复办法也不同。分清以后，你还得追问运行时历史、模型投影和 JSONL 记录各自留下了什么，Compression 怎样拿摘要换掉旧 Context，以及 GEMINI.md 和 Memory Service 怎样把知识带到下一次会话，这正是下一篇要拆开的五种「过去信息」。

下一篇：Session、记录、压缩与 Memory。

Session 记录、模型历史、Compression 与 Memory 的边界

返回 Gemini CLI 课程地图

上一章已经分清模型 SAFETY、用户拒绝和 Sandbox 初始化失败各自发生在哪里，可这些结果以后能不能拿来回放或恢复，还得看系统把它们写进了哪一种记录。Gemini CLI 至少保留五种「过去信息」：AgentChatHistory 掌管进程内的回合，模型只接收投影出来的 Content[]，ChatRecordingService 逐行追加 JSONL（逐行 JSON），Compression（压缩）拿摘要替换旧 Context，而 GEMINI.md 和 Memory Service 保存能跨会话复用的知识。

```
AgentChatHistory ──→ Content[] ──→ 下一次模型请求
      |
      ├─→ ChatRecordingService ──→ JSONL 回放记录
      └─→ Compression ──→ 摘要 + 保留尾部

Memory / GEMINI.md ──→ 新会话的项目上下文
```

第 1 站：运行时历史保留 Turn 身份，模型投影去掉它

源码：查看 AgentChatHistory

```
// The "Strong Owner" of chat history turns.
getContents(): Content[] {
  return this.history.map((turn) => turn.content)
}
```

调用者：GeminiChat 和 Agent Session。

输入：带稳定 Turn 标识的用户/模型 Content。

状态变化：追加或替换当前进程内 Chat Turns。

返回：模型 SDK 使用的不带内部 Turn ID 的 Content[]。

下一站：Gemini Client 发请求，或 Recording Service 保存表面消息。

系统靠内部身份去重并关联前后事件，Provider Content 却只关心角色和 Parts，因此两边留下的信息并不一样。如果你只保存 Provider Content，恢复时就可能找不回原来的内部 Turn 引用。内部身份不能丢。

Resume 只选可重新进入聊天的消息

第 2 站：显式 History 优先，恢复记录需要过滤

源码：查看 GeminiChat 初始化

```
if (history.length > 0) {
  initialHistoryTurns = history
} else if (resumedSessionData) {
  initialHistoryTurns = resumedSessionData.messages
    .filter((m) => m.type === 'user' || m.type === 'gemini')
}
```

调用者：创建或恢复 GeminiChat。

输入：显式历史、可选 Session Recording。

状态变化：选择唯一初始化来源；从完整记录投影聊天消息。

返回：AgentChatHistory 的初始 Turns。

下一站：Prompt Context 与新用户输入继续追加。

Tool、Info、Error 等 Recording Message 可以留给界面回放，却未必适合原样塞进 Provider 历史，因此恢复时要从完整记录里挑出聊天消息，再投影成模型认识的格式，不能把 JSONL 一行不漏地发给模型。

Recording 失败不一定终止正在运行的 Agent

第 3 站：JSONL 逐行追加，写失败后关闭记录通道

源码：查看 Chat Recording 写入

```
const line = JSON.stringify(record) + '\n'
fs.appendFileSync(this.conversationFile, line)

// 失败时将 conversationFile 置空，后续不再写。
```

调用者：用户、模型、工具和状态事件的记录点。

输入：可序列化 Session Record。

状态变化：追加一行；持久化故障时停用本 Session 的 Recording。

返回：通常不改变 Agent 的业务结果。

下一站：回放/恢复只能读取故障前成功落盘的前缀。

所以，即使 Agent 最后给出了完整结果，也不能说明 Session 记录同样完整，磁盘上能读到故障前的那一段，更不能证明后面的事件已经写入。如果评测要靠 Trace（执行轨迹）判定结果，就得单独记录 Recording 是否可用，别把基础设施故障和 Agent 的业务结果混在一起。这两种结果要分开。

Compression 先摘要，再让模型检查摘要是否遗漏

ChatCompressionService 先找一个不会截断工具交互的切分点，把较旧的 History（历史记录）交给模型归纳，同时留下最近一段内容，随后再发一次 Verification Prompt，让模型检查摘要有没有漏掉重要细节。

第 4 站：切分、摘要与核对是分开的步骤

源码：查看压缩与核对

```
const splitIndex = findCompressSplitPoint(...)
const historyToCompress = curatedHistory.slice(0, splitIndex)

const verificationResponse =
  await config.getGeminiClient().generateContent(...)
```

调用者：Context 压力策略。

输入：Curated History、Token 预算和 Gemini Client。

状态变化：选旧前缀进行摘要；用第二次请求检查遗漏并形成最终摘要。

返回：摘要文本、保留尾部与压缩元数据。

下一站：替换活动 Chat History。

第二次请求会让模型专门检查重要细节有没有丢失，确实能减少遗漏，可同一个模型也可能再次忽略同一个地方，所以这套核对不能证明压缩过程没有损失。摘要仍然有损。

第 5 站：新历史由摘要握手和尾部组成

源码：查看压缩后 History

```
const newHistory: Content[] = [
  { role: 'user', parts: [{ text: finalSummary }] },
  { role: 'model', parts: [{ text: 'Got it. Thanks for the additional context!' }] },
  ...historyToKeepTruncated,
]
```

调用者：Compression 提交阶段。

输入：最终摘要与保留尾部。

状态变化：用新的对话前缀替换模型可见历史。

返回：满足窗口预算的 Content[]。

下一站：后续 Turn 继续；Recording 仍可保留压缩事件与旧消息。

Memory 不应保存瞬态任务状态

GEMINI.md 和 Memory Service 适合记住稳定的项目命令、目录约定和长期偏好，却不该拿来充当环境快照。像「测试现在失败在第 42 行」这样的消息，应当留在当前 Session 或 Artifact（产物）里，因为文件很快会变，下次恢复时你仍要重新核对工作区。

跨会话知识进入新会话以后，Extension 还会刷新多个 Registry，从而改变这次运行真正能用的能力。下一篇会跟着 MCP Client Manager、Tool Registry、Hook System、Agent Registry 和 Skill Manager 依次往下看，弄清 Agents、Hooks、Skills 与 MCP 各自怎样进入当前 Session 的 Prompt 和工具面。

下一篇：Agents、Hooks、Skills、MCP 与 Extensions。

Extensions 怎样刷新 Agents、Hooks、Skills 与 MCP

返回 Gemini CLI 课程地图

上一节看到 Compression 怎样替换正在使用的 Chat History，也分清了 GEMINI.md 和 Memory Service 该保存哪些跨会话知识。现在从运行时能力继续往下看：Gemini CLI 的 Extension 可以带来 MCP Server、Tool 排除项、Policy、Hook、Skill 和 Agent，安装目录只说明这些能力来自哪里，只有启用或停用 Extension 时同步刷新多个运行时 Registry（注册表），模型实际能用的能力才会跟着改变。

第 1 站：Extension 切换是一次多 Registry 更新

源码：查看 Extension Loader 刷新流程

```
await this.config.getMcpClientManager()!.startExtension(extension)
await this.maybeRefreshGeminiTools(extension)
await this.config.getHookSystem()?.initialize()
await this.config.getAgentRegistry().reload()
await this.config.reloadSkills()
```

调用者：Extension Manager 启用、停用或重载某个 Extension。

输入：Extension 描述、信任状态与有效配置。

状态变化：连接 MCP，重算 Tools，重新初始化 Hooks、Agents 和 Skills。

返回：刷新后的运行时能力集合。

下一站：后续 Turn 构造新的 Prompt 与 Function Declarations。

只要其中一步失败，运行时就可能停在「一部分能力已经刷新、另一部分还是旧的」这种状态，因此实现既要写清更新顺序，也要处理怎么回滚或重新初始化。整轮刷新并不是一次原子替换。

Hook：先匹配和去重，再决定整批执行方式

源码：查看 Hook Planner

```
const matchingEntries = hookEntries.filter(...)
const deduplicatedEntries = this.deduplicateHooks(matchingEntries)
const sequential = deduplicatedEntries
  .some((entry) => entry.sequential === true)
```

调用者：Hook System 在某个生命周期事件发生时。

输入：事件、Matcher、来源 Extension 和 Hook 配置。

状态变化：筛选、去重并生成执行计划；只要一项要求串行，整批按顺序执行。

返回：Hook Plan 与后续结果集合。

下一站：Executor 按事件契约合并允许、阻断或附加上下文。

「整批串行」能避免有先后依赖的 Hook（钩子）和其他 Hook 交错执行，可只要其中一个 Hook 很慢，后面的所有项都会被它堵住。因此你不能只核对单个 Hook 的配置，还要针对具体事件检查整批任务怎样超时、出错后又怎样收场。

Skill：同名覆盖顺序和激活是两步

第 2 站：加载顺序决定同名 Skill 的最终定义

源码：查看 Skill 发现优先级

```
await this.discoverBuiltinSkills()
// extension skills -> user skills -> workspace skills
skillMap.set(newSkill.name, newSkill)
```

调用者：启动和 Extension 刷新后的 Skill Manager。

输入：Builtin、Extension、User 与 Workspace Roots。

状态变化：按加载顺序写入 Map，后加载的同名 Skill 覆盖前者。

返回：当前 Skill Catalog。

下一站：模型或用户调用 Activate Skill。

源码：查看 Skill 激活

```
skillManager.activateSkill(skillName)
this.config.getWorkspaceContext()
  .addDirectory(path.dirname(skill.location))

llmContent: `<activated_skill ...>${skill.body}...`
```

只有激活 Skill（技能）以后，系统才会把正文送回模型，同时把 Skill 目录加入可读取的资源范围。因此，目录里能够列出某个 Skill，只能说明系统发现了它，不能说明它的正文已经进入 Context。发现不等于激活。

MCP：发现前有信任与用户配置短路

第 3 站：连接成功后才把能力写入 Registry

源码：查看 MCP Client Manager

```
if (!finalConfig.command && !finalConfig.url && !finalConfig.httpUrl) return
if (this.isBlockedBySettings(name)) return
if (await this.isDisabledByUser(name)) return
if (!this.cliConfig.isTrustedFolder()) return

await client.connect()
await client.discoverInto(this.cliConfig, targetRegistries)
```

调用者：启动或 Extension 启用流程。

输入：MCP Server Config、Blocklist、用户开关和目录信任。

状态变化：通过前置门后建立连接，发现 Tools/Prompts/Resources 并写入目标 Registry。

返回：活动 MCP Client 与发现结果。

下一站：Tool Registry 过滤模型可见工具，调用时再次经过 Policy。

Agent：本地与远程不是同一个执行器

Agent Registry 能发现 Agent Definition（智能体定义），Local Executor 则会在当前进程里创建子会话并配好工具，Remote A2A 路径通过协议交换 Message、Artifact、状态和取消请求。两条路径看起来都可以通过 Agent Tool 调用，可它们的身份归属、认证方式、超时处理和结果边界各不相同。

需要隔离 Context 或者把一段长任务单独跑起来时，创建子 Agent 很合适。如果只是批量读取文件，直接调用普通工具反而省事。父 Agent 收到结果以后，还得核对 Stop Reason 和 Artifact，不能看到一条完成通知就把子运行里的错误盖过去。通知不是结论。

到这里，你已经看过 Extension 按什么顺序刷新各项能力，Hook 怎样整批执行，同名 Skill 怎样覆盖又怎样激活，MCP 连接前要过哪些门，以及 Agent 在本地和远程分别怎么跑。不过，使用者最后能看到什么，还取决于交互 CLI、非交互输出、IDE 与 A2A 各自暴露哪些事件、怎样表示停止，下一篇就来逐项划清这些输出边界。

下一篇：CLI、非交互输出、IDE 与 A2A。

交互 CLI、非交互输出、IDE 与 A2A 如何投影同一运行

返回 Gemini CLI 课程地图

上一节讲到，启用、停用或重载 Extension 时，系统会一起刷新 MCP Client Manager、Tool Registry（工具注册表）、Hook System、Agent Registry 和 Skill Manager。运行时能力刷新以后并不会原样摆到使用者面前，交互终端、非交互命令、IDE 集成和 A2A Server 都会挑选、转换或忽略 Core Events，并按各自的方式表示身份和停止。它们虽然复用了部分 Core，对外遵循的却不是同一套事件契约。

```
Core Agent Events
  ├─→ 交互 UI：消息、思考、工具卡片、确认
  ├─→ 非交互：text / json / stream-json
  ├─→ IDE：编辑器上下文、Diff 请求与响应
  └─→ A2A：Task、Message、Artifact、Status Update
```

stream-json 是一个有意收窄的公共协议

第 1 站：公开事件集合小于 Core Agent Events

源码：查看输出事件类型

```
enum JsonStreamEventType {
  INIT,
  MESSAGE,
  TOOL_USE,
  TOOL_RESULT,
  ERROR,
  RESULT,
}
```

调用者：非交互 Session 的 Stream Formatter。

输入：Core Agent Events 与 Session 元数据。

状态变化：映射为稳定、可逐行消费的公共事件。

返回：JSON Lines。

下一站：Shell、CI 或 SDK 客户端按类型解析。

公共协议有意藏起一部分内部生命周期，所以维护兼容性更容易，但你也无法只靠这些公开事件还原 Scheduler 的完整状态。

第 2 站：非交互消费者显式忽略部分事件

源码：查看非交互忽略列表

```
case 'initialize':
case 'session_update':
case 'agent_start':
case 'tool_update':
case 'elicitation_request':
case 'elicitation_response':
case 'usage':
case 'custom':
  // Explicitly ignore these non-interactive events.
```

调用者：非交互 Session 的事件循环。

输入：完整 Agent Event Stream。

状态变化：对不支持的交互事件不产生输出。

返回：只包含该表面契约允许的信息。

下一站：Formatter 生成 Text、JSON 或 Stream JSON。

交互 UI 会显示工具进度，非交互 JSON 却可能只给出开始和最终结果，所以自动化程序不能一直等协议明确不会发送的 tool_update。

同一消息在三种格式中走不同路径

源码：查看消息输出分支

```
if (streamFormatter) {
  // 立即发 MESSAGE delta
} else if (outputFormat === JSON) {
  responseText += output
} else {
  textOutput.write(output)
}
```

调用者：非交互 Session 处理模型消息事件。

输入：文本 Delta 与输出格式。

状态变化：Stream 模式立即输出；JSON 聚合到最终对象；Text 写标准输出。

返回：不同序列化时机的同一模型内容。

下一站：进程结束时输出 RESULT/最终 JSON 与 Exit Code。

如果选择 JSON，你拿到的是聚合后的结果，不该期待实时 Token。如果选择 Stream JSON，就得处理逐行到来的事件，还要接住中途出现的 Error。

IDE Diff 是一次独立请求/响应协议

第 3 站：按文件路径等待编辑器接受或拒绝

源码：查看 IDE Diff 交互

```
name: `openDiff`
this.diffResponses.set(filePath, resolve)
promise.finally(release)
```

调用者：需要用户在 IDE 审阅文件变更的工具。

输入：File Path、旧/新内容或 Diff 信息。

状态变化：注册该文件的待响应 Promise，发送 IDE Tool Call。

返回：接受、拒绝或连接/取消错误。

下一站：Tool Executor 根据决定提交或撤销修改。

用户在 IDE 里接受 Diff，只表示同意这次变更，不能拿它给代码正确性打分，这两件事不能混。

A2A 拥有自己的任务状态机

A2A Server 对外用 Task ID、Context ID、Message、Artifact（产物）和 Status Update 描述任务，远程客户端则可能看到 input-required、completed 或 failed。A2A 会用这些值标出 Task 最后停在哪种状态，但它们对不上内部 Tool Call 的状态，两层含义不能混。

调用远程 Agent 时，你要同时保存协议 Task ID 和本地父 Session/Call ID，这样才能从父工具请求一路追到远程 Artifact。即便网络请求成功，也只能说明双方完成了协议交换，业务有没有做成还得查 Task State 和 Artifact。

自动化应选择哪种表面

只要最终文本：Text 最简单，但证据最少。

需要结构化最终统计：JSON。

需要实时工具与错误事件：Stream JSON。

需要编辑器上下文和人工 Diff：IDE。

需要跨进程 Agent 互操作：A2A。

到这里，你已经看过 stream-json 公开哪些事件、非交互消费者丢掉哪些事件，也看过同一条消息怎样分别写进 Stream JSON、JSON 和 Text。IDE Diff 会停下来等用户接受或拒绝，A2A 则要继续检查 Task State 与 Artifact，光看网络请求成功远远不够，这几条路径不能直接互换。不同表面可以跑同一个任务，但做评测时必须锁定表面和版本，也不能把 UI 文本与 A2A Artifact 当成完全相同的输入。等这些边界固定下来，下一篇再看 Telemetry 能从模型请求、Tool Call 和运行指标里解释哪些失败，又有哪些任务答案、评分准则与发布阈值必须由独立 Eval 判断。

下一篇：Telemetry、错误分类与 Eval 接缝。

Telemetry 怎样解释失败，但不能替代独立 Eval

返回 Gemini CLI 课程地图

上一节先看 stream-json 公开哪些事件、非交互消费者忽略哪些事件，再追踪消息怎样走向三种输出格式，以及 IDE Diff 和 A2A Task 分别如何结束，由此把各个表面的事件与停止方式分开。做评测时还得锁定表面和版本。在这个前提下，Gemini CLI 会用 Telemetry（遥测）记录模型请求、API 错误、Tool Call、Confirmation、Hook、Compression、Retry、路由回退和运行指标，不过这些数据只能告诉你「发生了什么」，里面没有任务答案、评分准则和发布阈值。运行观测与任务评分必须分开。

用户确认决定与工具执行结果是两个字段

第 1 站：Decision 只描述授权动作

源码：查看 Confirmation Telemetry 映射

```
enum ToolCallDecision {
  ACCEPT,
  REJECT,
  MODIFY,
  AUTO_ACCEPT,
}

// ToolConfirmationOutcome.Cancel -> REJECT
```

调用者：Confirmation 结算后的 Telemetry Logger。

输入：用户或 Policy 对一次 Tool Call 的决定。

状态变化：不改变业务执行，只生成标准化观察字段。

返回：Tool Call Decision。

下一站：与执行时长、Success、工具类型一起记录。

源码：查看 Tool Call 指标

```
recordToolCallMetrics(config, event.duration_ms, {
  function_name,
  success,
  decision,
  tool_type,
})
```

Decision=ACCEPT 与 success=false 完全可以同时出现，因为用户虽然放行了这次调用，工具执行时仍可能失败。Decision=AUTO_ACCEPT 与 success=true 也只说明系统自动放行，而且工具按协议顺利结束，不能据此断定整个任务已经完成。

批量编辑的聚合会丢掉单项细节

源码：查看 Code Assist 聚合

```
// 任一 cancelled -> ACTION_STATUS_CANCELLED
// 任一 error -> ACTION_STATUS_ERROR_UNKNOWN
// 全部 accepted 且至少一个 edit -> ACCEPT_FILE
```

调用者：Code Assist 完成一批工具动作后。

输入：多个 Tool Call 的状态与编辑结果。

状态变化：按优先级折叠成一个交互状态。

返回：面向指标系统的聚合 Event。

下一站：Dashboard 或产品分析。

把一批动作折成一个状态很适合统计，却无法告诉你究竟哪个文件出了问题，所以 Eval 还要保留每个 Call 及其实际 Diff。

模型错误分类只覆盖响应层

源码：查看响应错误映射

```
// signal.aborted -> CANCELLED
// no candidates -> EMPTY
// STOP / MAX_TOKENS 之外的 FinishReason -> error
```

这段映射能分清传输何时取消、模型何时返回空响应或拒绝继续，却看不到目标仓库的测试有没有通过，所以给任务错误分类时，还得核对 Tool、Policy、Sandbox、Hook 和表面序列化，并验证 Artifact（产物）。

Telemetry 本身受配置和脱敏影响

第 2 站：启用、目标与 Prompt 记录都由有效配置决定

源码：查看 Telemetry Config

```
// argv -> env -> settings
// enabled, traces, target, otlpEndpoint,
// otlpProtocol, logPrompts, outfile
```

调用者：Telemetry SDK 初始化。

输入：命令行、环境变量和 Settings。

状态变化：选择 Exporter、协议、目的地与敏感正文策略。

返回：Logger/Meter/Tracer 使用的有效配置。

下一站：各运行点按配置发 Event 和 Metric。

关掉 Prompt 记录以后，Trace（执行轨迹）里仍会留下时长和错误码，却无法据此还原输入。如果 Exporter 自己出了故障，记录还会断档。没有事件，不等于没有执行。

一个可复核的 Eval Artifact

针对「修复失败测试」，建议保存：

固定仓库基线、用户 Prompt、Settings、信任状态和实际路由模型；

Agent/Turn 标识、模型 FinishReasons、Tool Call 状态与 Confirmation；

Sandbox 类型与执行规格、文件 Diff、完整测试输出；

表面输出、进程 Exit Code、Token、时长和 Telemetry 可用性。

Evaluator（评估器）要在 Agent 之外重新跑测试，再检查 Diff 改了多大范围、有没有违反任务约束，确认以后才能给出 Score。如果这些 Score 还要拿去训练，就得用 Reward Adapter（把原始信号转换成训练奖励的版本化规则）写清楚如何处理拒绝、基础设施失败和部分通过，选择 Checkpoint 与决定最终发布时则继续使用隔离任务集。

课程从 Config、Prompt 和 Context 起步，顺着 Turn、Tool、Policy、Session 与 Extension 组成的运行链，已经追到了表面协议、Telemetry 和独立 Eval 之间的边界。这一篇又把几个容易混淆的结果逐一拆开：Confirmation Decision 只说是否放行，工具 Success 说明调用怎样结束。批量编辑可以聚合统计，单项 Diff 才能指出哪个文件出了问题。响应错误来自模型调用，任务错误还要结合仓库里的实际结果来判。你还得保存实际 Diff、完整测试输出、表面输出、进程 Exit Code 和任务约束，最后把这些 Artifact 交给 Agent 之外的 Evaluator。回到课程地图后，就可以沿同一条调用链重新核对每一层收到什么、改了什么状态，又留下了什么证据。

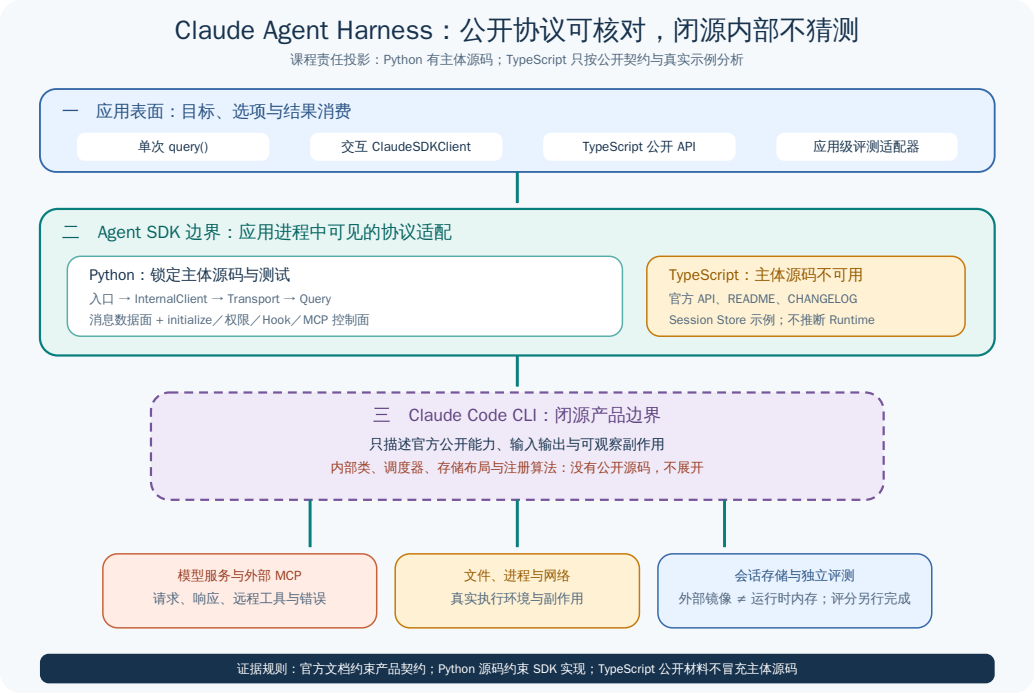
到这里可以回到 Gemini CLI 课程地图，或进入 Claude 课程 比较公开 SDK 中 Tool Permission 与 Hook 的契约边界。

Claude 源码课程

返回学习入口

读 Claude 课程时，你要先守住一条特殊的证据边界：Claude Code 是闭源产品，公开 Agent SDK（智能体开发工具包）能让你看到应用进程里的 API、Transport（传输层）、消息、控制协议、权限回调和 Session Store 接口，却没有交出 Claude Code 内部的运行时源码。这条边界别越过。

课程会一直标清结论来自「官方公开契约」、「Python Agent SDK 源码」还是「TypeScript Agent SDK 可见内容」，凡是当前来源无法核对的产品内部机制，就直接留白。



锁定来源与可见范围

Python Agent SDK 锁定在提交 542fefb3b94be87760b2513fff889b91bb5b6672：

- 公开 query 入口
- 内部 Client 控制层
- Query 控制协议
- Subprocess Transport

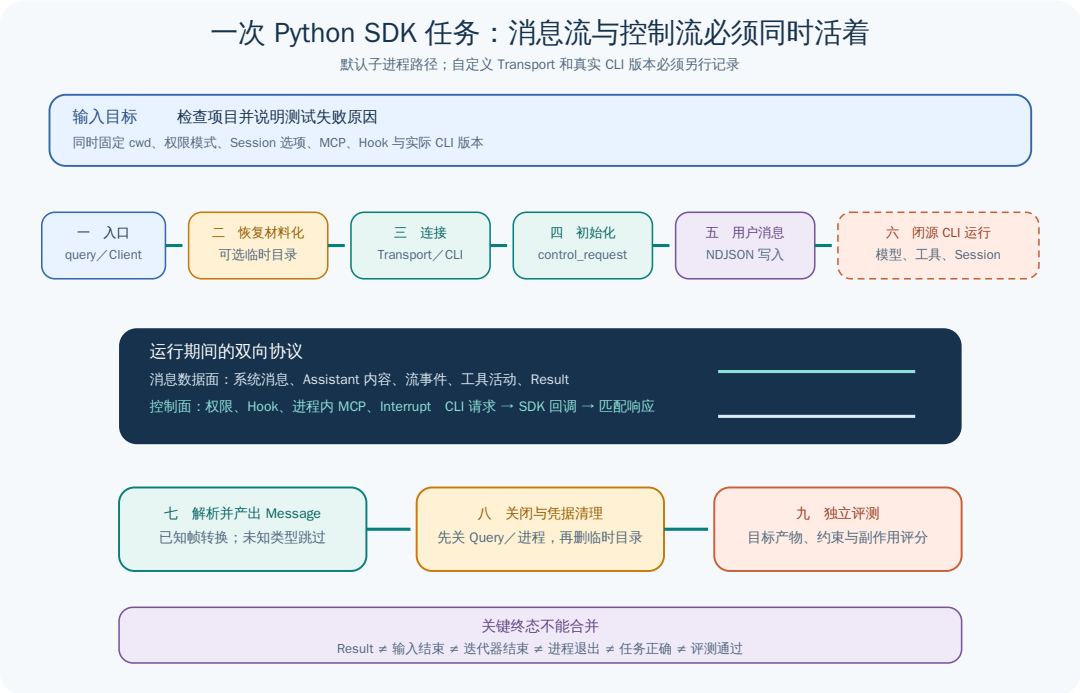
TypeScript Agent SDK 锁定在提交 48275071e804139579fabada9bb8d90cfe02b062。你可以用这份来源核对 README、CHANGELOG、公开类型说明和 Session Store（会话存储）示例，但仓库里没有 SDK 主体的运行时源码：查看公开 README。

要核对产品公开承诺了什么、API 字段表示什么，就读官方文档：Agent SDK 总览。它能解释对外行为和语义边界，但你仍然无法由此画出闭源实现的内部调用图。

先看一项任务

Python 应用调用 query() 或创建 ClaudeSDKClient 之后，SDK 会先准备 Options，再让 Transport 启动 CLI 子进程或连上已有进程。CLI 返回的消息会组成异步流，如果运行还需要双向控制，Query 控制层就会继续处理初始化、权限回调、Hook、MCP 和中断请求。

Python 应用
→ query() / ClaudeSDKClient
→ Options 与 Transport
→ Claude Code CLI（闭源产品边界）
→ 消息流与控制请求
→ SDK 类型、回调和 Session Store



这条流程到 CLI 边界就停下。公开 SDK 能证明传输、消息和控制请求是怎样运作的，但它不会告诉你闭源产品内部选了哪种 Agent Loop（智能体循环）。

图里一旦走到产品边界，就不能再像读开源模块那样往内展开，因此后面的课程只会继续追踪公开协议和 SDK 代码真正能到达的地方。

仓库地图

区域	能核对什么
query.py	一次性查询 API 怎样创建 Client 并返回消息
_internal/client.py	双向 Client 怎样连接 Query 与 Transport
_internal/query.py	控制请求、回调、Hook 和 MCP 消息怎样分派
_internal/transport	应用与 CLI 子进程之间的输入输出
types.py	消息、Options、权限和 Hook 的公开数据形状
tests	SDK 层可观察行为和错误路径

区域	能核对什么
TypeScript 示例	Session Store 等公开扩展契约

第一遍读时，凡是公开来源没有入口的 Claude Code 内部循环，都先跳过，别用猜测补上。

三层读法

Starter：先读证据边界、Python 入口和消息生命周期，分清应用进程与 CLI 边界。

Builder：继续追踪工具可见性、动态权限、Hooks、Session 和 MCP 控制请求。

Maintainer：对照 TypeScript 公开契约、错误表面和测试，检查哪些结论只能写成产品承诺。

阅读顺序

1. 产品、文档与 SDK 的证据边界 先从边界开始。分清 Claude 模型、Claude Code、Agent SDK 和应用，确定哪些结论能由锁定来源支持；边界画清后，才能进入 Python 的真实入口而不把 SDK 调用链误写成闭源产品内部结构。
2. Python 入口、Transport 与控制连接 接住这条证据边界，沿 query()、InternalClient、Transport 和 Query 追踪公开链路。连接建立之后，下一步就要分辨其中流动的消息、控制帧与结束信号。
3. 消息流与生命周期 再看数据面。从双向连接继续读取类型化消息，区分 Assistant 停止、Result、响应结束、断开和 Eval；消息中的 ToolUse 只是请求，要判断它能否出现和执行，还得回到配置层。
4. 工具可见性：模型能看到什么 承接 ToolUse，把工具集合、静态权限、动态权限、Hooks 与系统能力拆成不同层。分层之后，才能准确追问静态规则没有直接决定时，can_use_tool 处理什么。
5. 动态权限决策：can_use_tool 何时运行 动态询问由此展开。沿这条路径核对回调，避免把它当成每次工具调用的总开关；需要覆盖每次匹配调用的策略时，问题会自然转向 PreToolUse Hook。
6. Hooks 生命周期：注册、匹配与回调 接着追踪 Hook 的注册、回调 ID、控制请求和返回结果，厘清它与权限回调的分工。单次运行中的策略点明确后，课程再把视野移到跨轮次状态。
7. Session、Resume 与 Store 再进入跨轮次状态。从生命周期进入会话标识、恢复、分叉和外部 Store，区分本地 Transcript 与镜像适配层；状态怎样保存清楚之后，才能判断 MCP、Agent 与 Skill 分别扩展运行的哪一部分。
8. MCP、Agents 与 Skills 承接 Session 上下文，拆开协议能力、委派角色和按需说明三种扩展机制。它们的公开边界明确后，跨语言对照也必须继续服从同一套证据规则。
9. TypeScript 契约与差异 跨语言对照到这里。以 SessionStore 示例和测试为落点，对照 Python 源码与 TypeScript 锁定树能支持的不同结论；契约相符不等于实现同构，这项区分会成为最后设计错误分类和 Eval 的证据基础。
10. 产品表面、错误与 Eval 接缝 最后回到 Eval。接住跨语言证据边界，把启动、协议、权限、工具、运行与任务质量分层记录；至此，前九篇的外部链路才能汇入独立 Eval，而不会越过公开 SDK 虚构 Claude Code 内部实现。

用贯穿任务复盘

复盘时，可以让一个 Python 应用提交「修复运费问题」这个目标，然后一步步问：query() 或 ClaudeSDKClient 怎样准备 Options，Transport 怎样连接 CLI，消息和控制请求如何流动，can_use_tool、Hooks、MCP 与 Session Store 又分别在 SDK 边界的什么位置出现。每走一步，都标清它是「源码事实」、「官方文档契约」，还是「当前不可核对」。

复盘做到最后，不要强行画出 Claude Code 内部的 Agent Loop，而要准确停在公开边界上：SDK 能观察和控制哪些外显行为，哪些产品内部决策还没有公开源码支持。能停准位置就算读懂了。

完成课程后应该能回答

query() 与 ClaudeSDKClient 分别适合什么控制方式；

Transport 怎样承载消息与控制请求；

SDK 在哪里处理权限回调、Hook 和 MCP；

Session 恢复和自定义 Store 的公开契约是什么；

哪些结论属于 Python 源码事实，哪些只来自官方文档；

为什么不能用 Agent SDK 反推 Claude Code 内部实现。

从第一篇开始：产品、文档与 SDK 的证据边界

先画边界：我们能从 Claude Agent SDK 看见什么

返回 Claude 课程地图

课程地图已经追到了公开 SDK 与 CLI 交接的地方，这一篇先把边界说清楚，看每类证据到底能支持哪些结论。

学习 Claude Agent Harness（智能体框架）时，第一道难关并不在代码，而是别把几个不同的对象写成同一件事。日常交流里，人们常把 Claude 模型、Claude Code、Agent SDK 和自己的 Python 应用都叫作 Claude，但你若带着这种简称去读源码，就很容易引错证据，最后把结论也写错。

四个对象分别是谁

```
你的 Python 应用
  → Claude Agent SDK（本课程可读源码）
  → Claude Code CLI（公开产品，内部实现不在本仓库）
  → 模型服务与执行环境
```

Claude 模型负责根据上下文生成文本或工具请求。

Claude Code是承载 Agent 行为的产品运行时，包含 CLI 表面，但主体实现不是本课程锁定的开源源码。

Claude Agent SDK让应用启动或连接 CLI，发送消息，接收类型化事件，并处理公开控制协议。

你的应用决定怎样配置 SDK、实现权限回调和 Hooks、保存产物以及独立评测结果。

因此，「Python SDK 里有一个 Query 类」只能证明 Python SDK 怎样路由协议，不能证明 Claude Code 内部也有同名类。同理，即使官方文档说 SDK 提供 Agent Loop（智能体循环），你也不能拿这项公开契约去补画闭源循环的源码调用图。

两套 SDK 的证据并不对称

Python Agent SDK 锁定在提交 542fefb3b94be87760b2513fff889b91bb5b6672。这个仓库里有 query.py、client.py、内部 Transport（传输层）、消息解析、控制协议和测试，我们可以沿着 Python 真正走过的调用链一站站读下去。

Python 公开入口

Python 内部 Client

Python 控制协议

TypeScript Agent SDK 锁定在提交 48275071e804139579fabada9bb8d90cfe02b062。当前锁定的仓库只公开了 README、CHANGELOG、许可证和 Session Store（会话存储）示例，我们看不到 SDK 主体运行时的源码，也就无法沿着它的内部实现继续追踪。这门课可以讲它公开承诺的 API 和看得见的示例，但不会装作已经读过其内部实现：查看锁定 README。

「没有看到源码」有明确的范围：这句话只针对当前锁定的 Git 树，不能外推到 npm 分发包、其他仓库、私有实现或未来版本。

怎样判断一句话能不能写进源码课程

先看这句话说的到底是谁，再根据对象去找能够直接支持它的证据。

句子想说明什么	首选证据	合理写法
产品公开支持什么	官方文档和公开 API 契约	「官方文档说明……」
Python SDK 在该版本怎样运行	锁定源码与测试	「Python SDK 在此提交中……」
某配置下实际发生什么	固定版本的复现实验	「在这些条件下观察到……」
TypeScript 内部怎样实现	当前证据不足	明确未知和所需证据
Claude Code 内部怎样调度	当前证据不足	保留产品边界，不补想象图

例如，下面这句话不能直接发布。

> Python 与 TypeScript SDK 内部都由同一个 Query 控制器驱动 Claude Code。

这句话一口气跨过了三个对象，可我们能从这三处拿到的证据并不对等：锁定的 Python 源码只能证明 Python 入口把工作交给了内部 Client，TypeScript 锁定树里又没有主体实现，Claude Code 的内部运作则仍在产品边界之内。把这些限制一起写清楚，句子才能改成下面这样。

> 两套 SDK 都公开了查询表面，但只有 Python 锁定源码可以核对其 Client、Transport 与控制协议链路。当前 TypeScript 锁定树无法证明内部实现与 Python 同构，因此也不能据此推断 Claude Code 内部对象图。

从最小入口练习边界判断

Python 的公开 query() 最后几行非常简单，代码如下。

```
if options is None:
    options = ClaudeAgentOptions()

client = InternalClient()
async for message in client.process_query(
    prompt=prompt, options=options, transport=transport
):
    yield message
```

源码：查看完整入口

这段代码能支持三个结论：入口会在没有 Options 时补上默认值，然后实例化 InternalClient，并从 process_query() 异步取出消息。但这几行还没有走到后面的模型循环和工具执行，因此你不能据此断定「这里已经实现了模型循环」「每条工具调用都由这个函数执行」或「CLI 内部也使用 InternalClient」。

读源码的价值，在于弄清这几行改变了哪些状态、下一步应该去哪里，以及哪些问题到了这里仍然没有答案。

官方文档与源码冲突时怎么办

在线文档说的是当前对外公开的契约，锁定源码记录的则是某个历史提交。两者对不上时，别急着拿其中一个覆盖另一个，而要把版本和证据来源一起交代清楚：

1. 写明文档页面和访问日期；
2. 写明源码提交；
3. 判断差异是版本漂移、语言 SDK 差异，还是理解错误；
4. 在没有同版本证据前并列描述。

许可证也得逐个仓库处理，因为 Python 仓库里的 MIT 许可不会自动覆盖 TypeScript 仓库或 Claude Code 产品。几份材料的技术接口即使看起来相似，授权范围也仍然可能完全不同。

本课程之后怎样标注边界

后续文章遵循三种明确语气：

源码事实：给出锁定提交、文件和行号，追踪调用者、输入、状态变化、返回和下一站。

机制解释：用课程自己的图或伪代码帮助理解，并明确它是抽象，不伪装成上游类图。

产品边界：到公开 SDK 无法继续进入的地方停止，不用熟悉的设计模式填空。

边界清楚了，下一篇就只沿着锁定的 Python 源码往里走：从公开入口进入 InternalClient，再看 Transport 和控制协议怎样接上 CLI。

下一篇开始走真实主链：Python 入口、Transport 与双向控制。

Python 入口、Transport 与双向控制

返回 Claude 课程地图

上一篇已经把 Python SDK、Claude Code 和产品内部那些无法核对的机制分开了，现在只沿着锁定源码进入 query()，看你的应用从公开入口出发后怎样接上 CLI。

第一次使用 Claude Agent SDK 时，你很容易把 query() 当成「发一个 HTTP 请求」，可真实链路走的是一套受控的子进程协议：SDK 默认启动 Claude Code CLI，往标准输入写消息，再从标准输出读取 NDJSON。整条链路还是双向的，因为 CLI 在运行期间可能回过头来，要求 Python 执行权限回调、Hook 或进程内 MCP 工具。

先选择两种入口

```
# 一次性或预先给定全部输入
async for message in query(prompt="解释这个项目"):
    ...

# 需要追问、中断或运行时修改配置
async with ClaudeSDKClient() as client:
    await client.query("解释这个项目")
    async for message in client.receive_response():
        ...
```

query() 只让调用方单向消费消息，你给出 Prompt 或输入流后，继续迭代输出就可以了。ClaudeSDKClient 会把连接保留下来，你因此能继续发消息、调用 interrupt()、修改权限模式，然后接收下一轮响应。两个入口虽然复用了内部组件，但调用方分别要管住哪一段生命周期，这件事完全不同。

```
公开入口
  → InternalClient 组装运行
  → SubprocessCLITransport 连接 CLI
  → Query 启动读取任务与 initialize
  ↔ 普通消息 + 双向控制请求
  → 关闭 Query 和 Transport
```

第 1 站：query() 只做入口委托

源码：查看公开 query()

```
async def query(
    *,
    prompt: str | AsyncIterable[dict[str, Any]],
    options: ClaudeAgentOptions | None = None,
    transport: Transport | None = None,
) -> AsyncIterator[Message]:
```

调用者：批处理脚本、CI 作业或只需要一次响应的应用。

输入：字符串或异步消息流、Options，以及可选自定义 Transport。

状态变化：入口本身只补默认 Options 并创建内部 Client。

返回：一个逐条产出类型化 Message 的异步迭代器。

下一站：InternalClient.process_query() 选择 Transport、创建 Query 并协调清理。

源码注释把它称为 one-shot 或 unidirectional，还明确说明，需要追问或 Interrupt 时应该选 ClaudeSDKClient。这里提醒的是谁来控制 API：一次性入口替调用方管连接，长连接入口则把更多控制权交还给调用方。

Transport 不只是「发字符串」

默认的 SubprocessCLITransport 要做四件事：找到 CLI，组好参数和环境，接管标准流，最后关掉子进程。你可以用自定义 Transport（传输层）替换默认字节通道，但底层无论怎样传输，都必须完成 connect/read/write/close 这组契约。

第 2 站：InternalClient 先配置，再连接 Transport

源码：查看内部装配

```
configured_options = _configure_can_use_tool(options)

if transport is not None:
    chosen_transport = transport
else:
    chosen_transport = SubprocessCLITransport(
        prompt=prompt,
        options=configured_options,
    )

await chosen_transport.connect()
```

调用者：公开 query() 委托到 InternalClient 后进入这段逻辑。

输入：Prompt、配置后的 Options 和可选 Transport。

状态变化：默认路径创建子进程 Transport；自定义路径保留调用方提供的实现；随后真正连接。

返回：连接成功的 chosen_transport 被继续交给 Query。

下一站：Transport 的 connect() 查找 CLI、组装命令并打开进程标准流。

在连接建立之前，SDK 先配好权限回调，因为这份配置会决定 CLI 怎样把权限请求送回 SDK，恢复 Session 时也可能在此生成临时配置。所以，从创建 Transport 到真正连接的这一段就是生命周期边界，后面按什么顺序跑，在这里已经定下来了。

第 3 站：默认 Transport 在 connect() 才启动进程

源码：查看子进程连接

```
if self._cli_path is None:
    self._cli_path = await anyio.to_thread.run_sync(self._find_cli)

cmd = self._build_command()
self._process = await anyio.open_process(
    cmd,
    stdin=PIPE,
    stdout=PIPE,
    stderr=stderr_dest,
    cwd=self._cwd,
    env=process_env,
)
```

调用者：InternalClient 或 ClaudeSDKClient.connect()。

输入：CLI 路径、Options、工作目录、环境和已经构造好的命令参数。

状态变化：启动子进程，登记活动子进程，包装标准输入输出流，并把 Transport 标为 ready。

返回：connect() 没有业务消息返回；成功后对象具备读写能力。

下一站：Client 用该 Transport 创建 Query，启动后台读取任务并发送 initialize。

这里还做了两件容易漏看的事：SDK 会从继承的环境里过滤 CLAUDECODE，免得子进程误以为自己正嵌在另一个 Claude Code 里，而调用方一旦启用 stderr 回调，Transport 还会多启动一个读取任务。这些行为都取决于实际运行环境，因此复现实验时，你要记下 CLI 版本、工作目录和关键 Options，只记 Python 包版本远远不够。这些上下文不能省。

为什么协议必须双向

普通消息从 CLI 流向应用，但控制请求有两个方向：

SDK 发起：initialize、interrupt、动态修改权限模式等；

CLI 发起：can_use_tool、hook_callback、SDK 内 MCP 消息等。

如果应用只把 stdout 当成 Assistant 文本流，遇到权限或 Hook 就会卡住，因为 CLI 正等着 Python 写回控制响应，Python 却根本没有处理这类帧。这里必须双向处理。

第 4 站：Client 先启动读取，再初始化

源码：查看 Query 启动顺序

```
query = Query(
    transport=chosen_transport,
    is_streaming_mode=True,
    can_use_tool=configured_options.can_use_tool,
    hooks=...,
)

await query.start()
await query.initialize()
```

调用者：内部 Client 完成 Transport 连接后创建 Query。

输入：已连接 Transport、权限回调、Hooks、MCP Server 和初始化选项。

状态变化：start() 启动后台读循环；initialize() 随后发送控制请求并等待响应。

返回：初始化结果保存到 Query，运行具备收发普通消息和控制帧的能力。

下一站：Prompt 被写入 Transport，读循环按帧类型分流。

这两步不能换顺序。如果 SDK 先发 initialize，再启动读取任务，它等响应时就可能还没有消费者去处理 stdout。源码把读循环放在 initialize 前面，正是为了避开这种僵局。这样双方就不会互相干等。

第 5 站：读循环按请求 ID 配对控制响应

源码：查看控制帧分流

```
if msg_type == "control_response":
    request_id = response.get("request_id")
    if request_id in self.pending_control_responses:
        event = self.pending_control_responses[request_id]
        self.pending_control_results[request_id] = response
        event.set()
elif msg_type == "control_request":
    self._spawn_control_request_handler(request)
```

调用者：Query 的唯一后台读取任务。

输入：Transport 产出的反序列化帧。

状态变化：响应唤醒对应等待者；反向请求派生受控处理任务；普通消息进入应用流。

返回：读循环持续运行，直到 Transport 结束或发生错误。

下一站：等待者继续 initialize 或 Interrupt；反向请求进入权限、Hook 或 MCP 分支。

并发控制要靠请求 ID 把响应对回原请求，你不能用「最后发出的请求」去猜它属于谁，也不能让某个回调卡住唯一的读取任务，否则其他控制帧都走不下去。配对必须准确。

关闭为什么比启动复杂

一轮响应结束、输入关闭、消息迭代器退出和 CLI 进程终止，这四件事不在同一时刻发生。默认 Transport 的 close() 会先屏蔽 AnyIO 取消，停止 stderr 读取和输入写入，再等子进程自行退出，只有等到超时才升级终止手段。源码特意留出这段等待时间，因为太早发终止信号，可能会打断 Session 文件刷新。

源码：查看 Transport 关闭策略

所以调用方即使取消了任务，也要给 SDK 留出清理资源的机会，不能把 Client 对象一丢了事。清理不能省。对长连接来说，用 async with ClaudeSDKClient() 会更稳妥，因为正常结束和异常退出都会走同一条关闭路径。

一次完整运行应该记录什么

为了能复盘「为什么卡住或退出」，至少保存：

入口类型：query() 或 ClaudeSDKClient；

Python SDK 提交或版本、实际 CLI 版本；

关键 Options、工作目录和自定义 Transport 类型；

控制请求 ID、子类型、耗时和结果；

普通消息序列、最终 Result、异常；

关闭原因以及子进程是否回收。

靠这些信息，你可以判断 SDK 是否跑完了协议，也可以检查资源是否经历了完整的生命周期，但这还不能证明 Agent 已经正确完成用户任务。你仍然得独立检查最终文件、测试结果和任务约束。

协议怎样收发已经理清，但数据里到底有什么还没展开，所以下一篇要分辨类型化消息、增量事件和 Result，免得把协议正常结束误写成任务已经做对。

下一篇继续处理数据面：消息流、Result 与生命周期边界。

消息流、Result 与生命周期边界

返回 Claude 课程地图

Transport 和 Query 已经把普通消息、控制帧和清理步骤接了起来，这一篇顺着数据往下看，弄清 CLI 吐出的字典怎样变成可以评测的类型化轨迹。

SDK 输出的远不只一段 Assistant 文本，一次运行里还可能出现用户消息、Assistant 内容块、系统事件、增量事件、Hook 事件和 Result。如果你把它们全部交给 str(message) 打印，然后丢掉原有类型，工具调用、错误、成本、权限拒绝和会话标识就会散成很难复盘的日志碎片。类型不能丢。

先区分「消息结束」和「任务正确」

下面几个结束信号经常同时出现，但含义不同：

信号	表示什么	不表示什么
Assistant 的 stop_reason	一次模型生成停止	工具副作用完成
ResultMessage	CLI 为当前响应发布结果帧	用户目标一定正确
receive_response() 结束	便利迭代器已经产出第一个 Result	Client 已断开
disconnect() 完成	SDK 开始并完成连接清理	Eval 已通过
独立 Eval 通过	产物满足预先定义的检查	协议过程没有异常

例如 Result 的 subtype 虽然显示成功，Agent 却忘了修改文件，这时协议可以完整结束，任务仍然是失败的。反过来，文件即使碰巧改对了，只要控制通道异常退出，你就不该把这次运行记成干净成功。这两件事不能混。

第 1 站：原始字典先经过统一解析器

源码：查看消息解析入口

```
def parse_message(data: dict[str, Any]) -> Message | None:
    if not isinstance(data, dict):
        raise MessageParseError(...)

    if data.get("type") == "system" and data.get("subtype") in (
        "hook_started",
        "hook_response",
    ):
        ...
```

调用者：ClaudeSDKClient.receive_messages() 和一次性查询内部消费原始帧时调用解析器。

输入：Query 数据面产出的单个字典。

状态变化：解析器不修改会话，只校验结构并选择具体消息类型。

返回：类型化 Message，某些可忽略帧返回 None，非法结构抛 MessageParseError。

下一站：应用按消息子类型处理文本、工具块、系统事件或 Result。

解析器会先处理 Hook 事件，然后才走普通 SystemMessage 分支，这个顺序告诉你，只按顶层 type=system 分类会丢掉更具体的事件含义。可靠的消费者应该优先匹配 SDK 已经提供的类型，不要再自己发明一套字符串判断。

AssistantMessage 是内容块容器

一条 Assistant 消息可能同时装着文本块和工具调用块，而模型提出 ToolUse 时，只是在「请求执行」，还不能证明权限已经放行，更不能证明工具执行成功。应用要想完成最终审计，就得靠关联字段把 ToolUse、后续的工具结果、权限轨迹和副作用接起来。关联字段不能丢。

第 2 站：解析器保留消息关联字段

源码：查看 AssistantMessage 构造

```
return AssistantMessage(
    content=content_blocks,
    model=data["message"]["model"],
    parent_tool_use_id=data.get("parent_tool_use_id"),
    error=data.get("error"),
    usage=data["message"].get("usage"),
    message_id=data["message"].get("id"),
    stop_reason=data["message"].get("stop_reason"),
    session_id=data.get("session_id"),
    uuid=data.get("uuid"),
)
```

调用者：parse_message() 的 assistant 分支。

输入：CLI 消息中的 model、content、usage、ID 和停止原因。

状态变化：原始 content 字典已经先被转成不同内容块；这里封装消息级字段。

返回：强类型 AssistantMessage。

下一站：业务消费者遍历 content，并用 ID 与工具结果、子 Agent 或会话事件关联。

parent_tool_use_id 能告诉你这条消息是否从某个工具或子流程中派生，session_id 和 uuid 可以持久保存后继关联，stop_reason 则解释模型响应为什么停下。它们各自记录不同的事，不能全部压成一条展示文本。

增量事件和完整消息不要重复计数

开启部分消息后，应用可能同时收到 StreamEvent 和完整的 AssistantMessage。前者适合实时 UI 和计算首字延迟，后者则适合保存最终 Transcript（对话记录）并用于评测。如果你先拼上所有增量，后面又追加完整消息，统计程序就会把文本和 Token 算两遍。

常见策略有两种：

1. UI 使用增量渲染，但最终持久化以完整消息为准；
2. 同时保存两套流，用消息 ID 建立关联，统计时明确选择其中一种投影。

网络或进程异常时，你可能只拿到一部分增量，始终等不到完整消息，这时 Artifact（产物）应该标成 incomplete，不能把拼出来的那点文本伪装成正常完成的响应。轨迹并不完整。

第 3 站：Result 保留协议终态信息

源码：查看 ResultMessage 构造

```
return ResultMessage(
    subtype=data["subtype"],
    duration_ms=data["duration_ms"],
    duration_api_ms=data["duration_api_ms"],
    is_error=data["is_error"],
```

```
        num_turns=data["num_turns"],
        session_id=data["session_id"],
        total_cost_usd=data.get("total_cost_usd"),
        permission_denials=data.get("permission_denials"),
        errors=data.get("errors"),
        terminal_reason=data.get("terminal_reason"),
    )
```

调用者：parse_message() 的 result 分支。

输入：CLI 发布的时延、轮数、成本、错误、拒绝和终止信息。

状态变化：解析器把可选的 deferred tool use 等嵌套字段转成对应类型。

返回：ResultMessage，缺少必需字段时抛解析错误。

下一站：便利消费者结束当前响应，长连接 Client 仍可发送下一条查询；Artifact 构造器保存终态事实。

Result 适合回答「这次协议跑了多少轮、有没有报错、为什么停下」，却不会替你检查仓库文件是否正确。所以 Eval 还得跳出 Result，另外读取工作区差异、测试输出和约束违规。协议终态不等于 Eval。

receive_messages() 与 receive_response() 的区别

receive_messages() 会一直从连接上取出各类型化消息，receive_response() 则只为收取单次响应提供便利：它一看到 ResultMessage 就结束这次迭代，但 Client 并没有断开。

第 4 站：便利方法在第一个 Result 后返回

源码：查看 receive_response()

```
async for message in self.receive_messages():
    yield message
    if isinstance(message, ResultMessage):
        return
```

调用者：需要「发送一条查询，然后收完这条响应」的长连接应用。

输入：底层 receive_messages() 的类型化消息流。

状态变化：方法本身不关闭 Transport，只结束当前异步迭代器。

返回：从响应开始到包含第一个 Result 的消息序列。

下一站：调用方可以处理 Result、发送下一条 Query，或显式 Disconnect。

因此，下面这种用法是合法的：

```
async with ClaudeSDKClient() as client:
    await client.query("先检查测试失败")
    first = [m async for m in client.receive_response()]

    await client.query("再修复第一个失败")
    second = [m async for m in client.receive_response()]
```

如果你看到第一个 receive_response() 结束，就以为连接也已经关闭，后面很可能会把第二轮错放进新 Session 或新进程。迭代结束了，连接还在。

建议的 Artifact 结构

如果你只保存最后那段文字，到评测时就再也还原不了中间发生过什么，因此一份最小可评测产物至少可以保存以下内容。

```
{
  "sessionId": "...",
  "messages": ["类型化消息或可还原摘要"],
  "toolCalls": ["请求、决定、结果与 tool_use_id"],
  "results": ["所有 ResultMessage"],
  "transportError": null,
  "workspaceDiff": "...",
  "testResult": "..."
}
```

前三项是协议轨迹里的事实，工作区差异和测试结果则要去执行环境里取。Scorer（评分器）必须同时读到这两类信息，才能分清下面几种情况：

模型没有提出正确动作；

权限拒绝了正确动作；

工具执行失败；

文件修改成功但逻辑错误；

协议异常导致轨迹不完整。

最小消费器示例

```
from claude_agent_sdk import AssistantMessage, ResultMessage, TextBlock, ToolUseBlock

async for message in client.receive_response():
    if isinstance(message, AssistantMessage):
        for block in message.content:
            if isinstance(block, TextBlock):
                print(block.text)
            elif isinstance(block, ToolUseBlock):
                record_tool_request(block.id, block.name, block.input)
    elif isinstance(message, ResultMessage):
        record_protocol_result(message)
```

这个示例只演示了怎样按类型分流，要做成完整审计器，还差不少工作。真实系统必须把 ToolResult、Hook、权限决定、异常和工作区副作用关联起来，并对其中的敏感内容做脱敏。

顺着类型化轨迹看到这里，已经可以确定 ToolUse 只是一条执行请求。下一篇回到 Options，看模型能看见哪些工具，以及某个工具最后是否真能执行。

下一篇从消息里的 ToolUse 回到配置层：工具可见性：模型能看到什么。

工具可见性：模型能看到什么，不等于什么都能执行

返回 Claude 课程地图

消息消费器已经能从类型化消息里分出 ToolUse，但你看到了工具请求，还是不知道模型为什么能选这个工具。要回答这个问题，得回到配置里，逐层检查模型看到了哪些工具、权限规则怎样判断，以及系统最后是否真能执行。

假设你正在做一个代码审查 Agent，只希望 Claude 读取文件和搜索文本，不希望它修改文件。配置可以这样写。

```
from claude_agent_sdk import ClaudeAgentOptions

options = ClaudeAgentOptions(tools=["Read", "Glob", "Grep"])
```

这段配置只告诉模型可以看到哪些内置工具，还回答不了调用时是否要批准、工具进程能访问哪些目录，以及操作系统到底会不会执行命令。这几个问题一旦混在一起，权限设计几乎一定出错。

先建立五层判断框架

一次工具调用至少要过五关。

层次	要回答的问题	Claude Agent SDK 中可见的入口
工具集合	模型本轮能否选择这个工具	tools
静态权限规则	这次调用是否直接允许或拒绝	allowed_tools、disallowed_tools、permission_mode、Settings
动态权限	静态规则给出「询问」时由谁决定	can_use_tool
生命周期策略	调用前后是否要检查、改写或补充上下文	hooks
系统能力	进程最终能否读文件、联网或启动程序	Claude Code 进程、Sandbox 与操作系统

你可以把整个过程想成进办公楼：工具集合像楼层目录，静态权限像门禁白名单，动态权限像前台打电话确认，Hook 负责在进出时触发审计，而系统能力决定门后的房间和设备是否真的存在。这几层不能混。目录里有一个房间，不代表你已经获准进入，门禁即使放行，房间里的设备也不一定能用。

tools 与 allowed_tools 的根本区别

下面两种配置说的不是一回事。

```
# 模型只能看到这三个内置工具。
ClaudeAgentOptions(tools=["Read", "Glob", "Grep"])

# Read 调用不再进入交互式批准；其他工具是否可见由 tools 或默认预设决定。
ClaudeAgentOptions(allowed_tools=["Read"])
```

tools=[] 会明确关掉所有内置工具，tools=None 则表示这个字段不去改写基础集合。区别就在这里。再看 allowed_tools=[]，它只是没有额外加入自动放行规则，不会因此关掉所有工具。

第 1 站：公开 Options 先把两个概念写成两个字段

源码：查看 ClaudeAgentOptions 的工具字段

```
@dataclass
class ClaudeAgentOptions:
    tools: list[str] | ToolsPreset | None = None
```

```
    allowed_tools: list[str] = field(default_factory=list)
```

调用者：业务代码在构造 ClaudeAgentOptions 时填写这两个字段。

输入：基础工具名或预设，以及需要自动放行的权限规则。

状态变化：这里只生成配置对象，还没有启动 CLI，也没有执行工具。

返回：一个供后续 Transport 消费的 Options 实例。

下一站：SubprocessCLITransport 把配置翻译成 Claude Code CLI 参数。

这段源码虽然只有两行字段，注释却已经把边界划得很清楚：tools 决定模型能用哪些基础工具，allowed_tools 则决定哪些调用不必询问。后面再看到「工具已允许」时，你要先问清楚这句话指的是哪一层。

配置怎样变成进程参数

Python SDK 不会在 ClaudeAgentOptions 里面运行权限算法，它只把我们能观察的配置转成 CLI 参数，然后启动或连接 Claude Code CLI。开源证据走到这个交界处也就到头了，再往里便是产品边界。

第 2 站：Transport 把基础集合写成 --tools

源码：查看命令构造

```
if self._options.tools is not None:
    tools = self._options.tools
    if isinstance(tools, list):
        if len(tools) == 0:
            cmd.extend(["--tools", ""])
        else:
            cmd.extend(["--tools", ",".join(tools)])
    else:
        cmd.extend(["--tools", "default"])
```

调用者：Transport 的连接流程调用 _build_command()。

输入：上一站保存的 options.tools。

状态变化：SDK 正在组装子进程命令数组，尚未运行具体工具。

返回：包含 --tools 的命令参数列表。

下一站：同一个函数继续加入允许和拒绝规则，再由 Transport 启动 CLI。

实现遇到空列表时，会明确生成 --tools ""，所以你不能把这种情况理解成「省略参数」。源码已经给出可以直接核对的行为：SDK 用明确参数关闭内置工具，不必猜它会套用什么默认值。

第 3 站：允许和拒绝规则走独立参数

源码：查看允许与拒绝参数

```
if effective_allowed_tools:
    cmd.extend(["--allowedTools", ",".join(effective_allowed_tools)])

if self._options.disallowed_tools:
    cmd.extend(["--disallowedTools", ",".join(self._options.disallowed_tools)])
```

调用者：仍然是 _build_command()。

输入：有效允许规则和 disallowed_tools。

状态变化：命令行同时携带工具集合与权限规则，它们没有被折叠成一个字段。

返回：完整度更高的 CLI 命令数组。

下一站：CLI 根据这些规则处理模型提出的具体调用；若结果需要询问，才可能进入动态权限回调。

这里所说的「有效允许规则」，不一定就是用户原样填进 allowed_tools 的内容，因为后面合成配置时，Skills 也可能往里补规则。

为什么 Skills 会影响允许规则

当 skills="all" 时，Transport 会把裸 Skill 规则加入允许列表，而当调用方传入具体 Skill 名称时，它会加入 Skill(name) 形式的规则。

源码：查看 Skills 默认规则合成

```
if skills == _SKILLS_ALL:
    if "Skill" not in allowed_tools:
        allowed_tools.append("Skill")
else:
    for name in skills:
        pattern = f"Skill({name})"
        if pattern not in allowed_tools:
            allowed_tools.append(pattern)
```

排查权限时，只打印调用方最初填的 Options 不够，因为 CLI 真正收到的是合成后的有效配置。这一步别漏了。后面的权限课程还会看到，裸 Skill 可能直接自动放行整个工具，动态回调也就根本轮不到。

三个容易犯的错误

错误一：把 allowed_tools 当工具白名单

如果只写 allowed_tools=["Read"]，你还不能断定模型只看得到 Read，因为要缩小模型可见的工具范围，同时还得检查 tools 和当前使用的预设。

错误二：把「自动允许」当「系统一定执行成功」

权限规则即使放行了，目标文件仍可能不存在，进程也可能没有访问文件系统的权限，Sandbox 或操作系统还可能直接拒绝命令。这些失败发生在不同层，所以你必须分开记录权限怎样判断，以及工具最后执行得怎样。

错误三：从 SDK 参数反推闭源产品内部算法

公开源码能证明 SDK 生成了哪些参数，也能证明控制协议怎样收发请求，但它无法告诉你 Claude Code 内部用了什么类，又依次调了哪些私有函数。文章到这里就要停在产品边界上，不能补画一条无法核对的虚构源码链路。

一个更安全的只读配置

```
from claude_agent_sdk import ClaudeAgentOptions

options = ClaudeAgentOptions(
    tools=["Read", "Glob", "Grep"],
    allowed_tools=["Read", "Glob", "Grep"],
    disallowed_tools=["Bash", "Edit", "Write"],
)
```

这段配置表达了三层意图：模型只能看到只读内置工具，调用这三类工具时无须逐次询问，高风险工具还会收到单独的拒绝规则。不过 Settings、MCP 工具、Hook、Sandbox 和进程权限都会影响最后发生的事，因此单凭这段配置还证明不了系统是安全的。

读完后怎样自查

看到任意「工具权限」配置时，按下面顺序问：

1. tools 最终给模型暴露了哪些工具？
2. allowed_tools、disallowed_tools、Settings 和 permission_mode 会怎样影响具体调用？
3. 哪些调用仍会落到动态询问？
4. Hook 是否还会在调用前后执行？
5. 系统能力是否真的支持这次操作？

下一篇继续追踪第 3 个问题：动态权限决策：can_use_tool 何时真的会被调用。

动态权限决策：can_use_tool 不是每次工具调用的总开关

返回 Claude 课程地图

工具集合和静态规则已经分清了，上一章五层框架还剩动态询问这一层：业务代码什么时候才有机会判断某次调用，要顺着 can_use_tool 走过控制协议的过程来看。

业务代码常常要在运行时判断：「这条 Bash 命令能不能执行？」Claude Agent SDK 虽然提供了 can_use_tool 回调，但它只替用户回答原本需要交互询问的那一次权限问题，并不会自动接管每次工具调用。

```
from claude_agent_sdk import PermissionResultAllow, PermissionResultDeny

async def decide(tool_name, tool_input, context):
    if tool_name == "Bash" and tool_input.get("command") == "npm test":
        return PermissionResultAllow()
    return PermissionResultDeny(message="这里只允许运行测试")
```

如果静态规则已经自动放行 Bash，上面的回调可能根本收不到这次调用。这条边界很关键。写权限策略之前，你得先认清它。

权限决策不是单个布尔值

公开契约没有把决定压成一个布尔值，因为回调允许调用时，还可以改写工具输入并建议更新权限。拒绝也不只说「不行」，回调还能说明原因，并决定要不要中断整个会话。

```
@dataclass
class PermissionResultAllow:
    behavior: Literal["allow"] = "allow"
    updated_input: dict[str, Any] | None = None
    updated_permissions: list[PermissionUpdate] | None = None

@dataclass
class PermissionResultDeny:
    behavior: Literal["deny"] = "deny"
    message: str = ""
    interrupt: bool = False
```

源码：查看权限结果类型

所以权限层至少要记住原始输入、最终决定、改写后的输入和决定理由。日志里如果只留一个 true/false，以后就没法解释工具实际执行的参数为什么和模型最初给出的不同。

先看完整路径

```
具体工具调用
  → Claude Code 的权限规则求值（闭源产品边界）
  → 已允许：继续，不调用 can_use_tool
  → 已拒绝：拒绝，不调用 can_use_tool
  → 需要询问：经控制协议发送 can_use_tool 请求
  → Python 回调返回 Allow 或 Deny
  → SDK 转成控制响应写回 CLI
```

这张图的后半段可以从开源 SDK 里核对，前半段怎样在内部计算规则，却只能按公开契约来划定边界，不能写得像我们已经读过 Claude Code 的内部实现。

第 1 站：配置阶段把询问通道切到标准输入输出

源码：查看 _configure_can_use_tool

```
if not options.can_use_tool:
    return options
if options.permission_prompt_tool_name:
    raise ValueError(...)
_warn_if_can_use_tool_shadowed(options)
return replace(options, permission_prompt_tool_name="stdio")
```

调用者：一次性 query() 和双向 ClaudeSDKClient.connect() 的内部流程都会调用它。

输入：包含 can_use_tool 的 Options。

状态变化：原对象不被原地修改；SDK 创建一个把权限询问路由到 stdio 的副本。

返回：配置后的 Options，或在两种互斥提示机制同时存在时抛错。

下一站：Client 用新 Options 创建 Transport 和 Query 控制层。

这里回答的是「权限问题怎样送回 Python 进程」，至于「哪些工具需要询问」，仍由 CLI 的权限规则来决定。这两件事不能混。

第 2 站：Client 把回调交给 Query 控制层

源码：查看 Client 创建 Query

```
configured_options = _configure_can_use_tool(options)
chosen_transport = SubprocessCLITransport(
    prompt=prompt,
    options=configured_options,
)
await chosen_transport.connect()

query = Query(
    transport=chosen_transport,
    is_streaming_mode=True,
    can_use_tool=configured_options.can_use_tool,
    ...
)
```

调用者：公开查询入口最终进入 _process_query_inner()。

输入：Prompt、Options，以及可选的自定义 Transport。

状态变化：Transport 已连接；回调引用被保存到 Query 实例。

返回：这个异步生成器随后会输出消息，而 Query 在后台处理控制请求。

下一站：CLI 发来 can_use_tool 控制请求时，Query 找到并调用保存的回调。

走到这一步，Transport（传输层）只负责搬运消息，Query 把协议事件交给 Python 回调，最后由回调决定是否放行，三者各自做什么也就分清了。

第 3 站：真正收到请求后构造上下文

源码：查看 can_use_tool 请求分支

```
if subtype == "can_use_tool":
    original_input = permission_request["input"]
    context = ToolPermissionContext(
```

```
        signal=None,
        suggestions=[...],
        tool_use_id=permission_request.get("tool_use_id"),
        agent_id=permission_request.get("agent_id"),
        ...
    )
    response = await self.can_use_tool(
        permission_request["tool_name"],
        permission_request["input"],
        context,
    )
```

调用者：Query 的控制消息读取循环收到 CLI 请求后进入 _handle_control_request()。

输入：工具名、工具输入、建议权限更新、tool_use_id、子 Agent 标识和展示信息。

状态变化：SDK 将线上的字典重建为 ToolPermissionContext，然后暂停等待业务回调。

返回：业务回调返回 PermissionResultAllow 或 PermissionResultDeny。

下一站：Query 把 Python 类型编码为 CLI 认识的控制响应。

多个工具并发调用时，只按工具名去找审计记录，很容易把同名调用串到一起。调用身份不能丢。你要用 tool_use_id 分清每一次调用，再用 agent_id 判断权限请求来自主 Agent 还是子 Agent。

第 4 站：允许可能改写输入，拒绝可能中断

源码：查看权限结果编码

```
if isinstance(response, PermissionResultAllow):
    response_data = {
        "behavior": "allow",
        "updatedInput": response.updated_input
        if response.updated_input is not None
        else original_input,
    }
elif isinstance(response, PermissionResultDeny):
    response_data = {"behavior": "deny", "message": response.message}
    if response.interrupt:
        response_data["interrupt"] = response.interrupt
```

调用者：仍是 _handle_control_request()。

输入：业务回调的强类型结果和原始工具输入。

状态变化：Python 字段被转换为协议字段；未改写时显式回填原输入。

返回：统一的 control_response 最终写回 Transport。

下一站：CLI 根据响应继续或拒绝工具调用；这一执行分支属于产品边界。

这里还能测出一条固定规则：业务回调只能返回规定的两种类型，否则 SDK 会抛出 TypeError，不会悄悄把任意字典当成权限决定。

为什么回调会「失踪」

SDK 专门检查两类明显的遮蔽配置：

permission_mode="bypassPermissions" 会在回调前自动批准调用，但显式拒绝规则仍例外；

allowed_tools 中允许整个工具的规则，如 Read、Read()、Read(*)，会让相应调用在到达回调前获批。

源码：查看遮蔽警告

带约束的规则和直接放行整个工具的规则并不一样。例如 Bash(npm test:*) 只放行匹配的调用，其他 Bash 输入仍可能触发询问。Settings 文件也可能写了允许规则并提前放行调用，但 SDK 做这段本地检查时未必看得到，所以没有警告也不能说明回调覆盖了所有调用。

正确选择：权限回调还是 PreToolUse Hook

需求	更合适的机制
只处理原本需要用户确认的调用	can_use_tool
每次工具调用都要审计或策略检查	PreToolUse Hook
调用执行后清洗结果或补充上下文	PostToolUse Hook
彻底限制模型看见的工具集合	tools
约束文件、网络和子进程能力	Sandbox 与操作系统策略

PreToolUse 也不能和权限回调画等号，它介入的生命周期更广，还能返回 allow、deny、ask 或 defer 等特定输出。下一篇会顺着注册过程，看回调 ID 怎样生成，响应又怎样写回。

最小验证实验

你可以让三组配置都调用同一个 Read，再看回调各触发了多少次：

1. 仅配置 can_use_tool，让权限规则产生询问；预期回调收到请求。
2. 再加入 allowed_tools=["Read"]；预期该调用被静态规则自动放行，回调计数不增加。
3. 移除裸允许规则，改加 PreToolUse Hook；预期每次匹配的 Read 都进入 Hook。

这个实验只能验证公开回调表现，证明不了 Claude Code 内部怎样实现。因此测试时要记下 Options、工具输入、回调次数、回调决定和最终工具结果，不能只凭终端上看起来「好像执行了」就下结论。

边界到这里就清楚了。can_use_tool 只接住需要询问的调用，如果你想覆盖每次匹配的调用，还得继续看 Hook 怎样注册、怎样返回响应。

下一篇：Hooks 生命周期：注册、匹配、并发回调与结果改写。

Hooks
生命周期：在工具调用前后插入可核对的策略点

返回 Claude 课程地图

can_use_tool 只能接到权限规则留下的询问，没法替每次调用都跑一遍策略。要覆盖更多事件，就得看 Hook 怎样注册，运行时怎样分派，以及回调结果怎样编码后写回。

权限回调只处理需要询问的调用，Hooks（钩子集合）却能插进更多生命周期节点：工具执行前检查策略，成功后改写送回模型的结果，失败后补充 Context，也能观察 Prompt 提交、压缩和子 Agent 等事件。

先看一个调用前 Hook：

```
from claude_agent_sdk import HookMatcher

async def protect_env_file(input_data, tool_use_id, context):
    target = str(input_data.get("tool_input", {}).get("file_path", ""))
    if target.endswith(".env"):
        return {
            "hookSpecificOutput": {
                "hookEventName": "PreToolUse",
                "permissionDecision": "deny",
                "permissionDecisionReason": "禁止读取环境变量文件",
            }
        }
    return {}

hooks = {
    "PreToolUse": [HookMatcher(matcher="Read", hooks=[protect_env_file])]
}
```

这段代码的意思是「每个匹配的 Read 在执行前都要经过策略函数」，并非「等权限系统想询问时再来调用我」。两条链路不同。

Hook 由事件、匹配器、回调和输出组成

部件	作用
事件名	确定生命周期位置，如 PreToolUse、PostToolUse
matcher	在该事件内筛选工具或对象
回调列表	一个匹配器可关联多个 Python 异步函数
通用输出	控制是否继续、停止原因、展示信息
特定输出	按事件允许决策、改写输入、补充上下文或替换结果

Hook 不能随手返回任意对象，因为公开类型同时约束了输入和输出，而且事件不同，回调能用的字段也不同。

第 1 站：事件类型定义了可观察的生命周期

源码：查看 Hook 事件与工具输入类型

```
HookEvent = (
    Literal["PreToolUse"]
    | Literal["PostToolUse"]
```

```
    | Literal["PostToolUseFailure"]
    | Literal["UserPromptSubmit"]
    | Literal["Stop"]
    | Literal["SubagentStop"]
    | Literal["PreCompact"]
    | Literal["Notification"]
    | Literal["SubagentStart"]
    | Literal["PermissionRequest"]
)

class PreToolUseHookInput(BaseHookInput, _SubagentContextMixin):
    tool_name: str
    tool_input: dict[str, Any]
    tool_use_id: str
```

调用者：业务代码用这些类型构造 hooks 配置，运行时由 CLI 触发相应事件。

输入：事件名、工具名、工具输入、调用 ID，以及可能存在的子 Agent 信息。

状态变化：类型声明本身不执行回调，只规定协议两端都能理解的数据形状。

返回：强类型的 Hook 输入联合类型供回调消费。

下一站：HookMatcher 把事件下的匹配条件与 Python 回调组合起来。

tool_use_id 把同一次工具调用的前置、成功和失败事件串起来，遇到子 Agent 时，还可以用 agent_id 和 agent_type 继续区分。工具并发执行后，这些标识远比「消息出现的先后顺序」可靠。

不同时间点能做的事情不同

PreToolUse：在执行前给出权限决定、改写输入或增加上下文。

PostToolUse：拿到成功结果后增加上下文，或者用符合工具输出 Schema 的值替换结果。

PostToolUseFailure：观察错误和中断信息，为后续推理提供上下文。

PreCompact：在上下文压缩前记录或协调外部状态。

SubagentStart、SubagentStop：观察子 Agent 生命周期。

源码：查看 Hook 特定输出

用 PostToolUse 改写输出时尤其要小心：给内置工具换上的值必须符合它的输出 Schema，形状对不上，系统就会拒绝替换并保留原结果。Hook 绕不过这份数据契约。

第 2 站：公开配置先转换成 Query 可消费的形状

源码：查看 Hook 配置转换

```
for event, matchers in hooks.items():
    internal_hooks[event] = []
    for matcher in matchers:
        internal_matcher = {
            "matcher": matcher.matcher,
            "hooks": matcher.hooks,
        }
        if matcher.timeout is not None:
            internal_matcher["timeout"] = matcher.timeout
        internal_hooks[event].append(internal_matcher)
```

调用者：Client 在创建 Query 时调用 _hooks_to_internal_format()。

输入：按事件组织的 HookMatcher 列表。

状态变化：Dataclass 配置被转成内部字典，但回调函数仍只是进程内对象。

返回：Query 接受的事件、匹配器、回调和超时结构。

下一站：初始化控制协议为每个回调分配 ID，不能把 Python 函数对象直接发送给 CLI。

这里也能看清超时由谁控制：它写在 matcher 配置里，Hook 函数不能在内部随意决定。

第 3 站：初始化时把函数换成回调 ID

源码：查看 Query 初始化 Hook

```
for callback in matcher.get("hooks", []):
    callback_id = f"hook_{self.next_callback_id}"
    self.next_callback_id += 1
    self.hook_callbacks[callback_id] = callback
    callback_ids.append(callback_id)

hook_matcher_config = {
    "matcher": matcher.get("matcher"),
    "hookCallbackIds": callback_ids,
}
```

调用者：Client 启动 Query 后调用 initialize()。

输入：内部 Hook 配置和本地 Python 回调对象。

状态变化：Query 在 hook_callbacks 映射中保存函数，并给每个函数分配稳定的进程内 ID。

返回：初始化请求只携带 matcher、回调 ID 和超时，不携带 Python 函数本身。

下一站：CLI 在事件发生时，通过控制协议带着某个回调 ID 请求 Python 执行。

跨进程时，Claude Code CLI 不会直接「调用 Python 函数」，而是先发一条协议消息。Query 收到消息后按 ID 找回函数，执行完再把结果写回去。

第 4 站：收到 hook_callback 后执行并编码结果

源码：查看 Hook 回调分派

```
callback_id = hook_callback_request["callback_id"]
callback = self.hook_callbacks.get(callback_id)
if not callback:
    raise Exception(f"No hook callback found for ID: {callback_id}")

hook_output = await callback(
    request_data.get("input"),
    request_data.get("tool_use_id"),
    {"signal": None},
)
response_data = _convert_hook_output_for_cli(hook_output)
```

调用者：Query 的控制消息循环处理 hook_callback 子类型。

输入：回调 ID、强类型事件输入、可选工具调用 ID 和上下文。

状态变化：本地异步函数运行；Python 保留字安全字段会转换为 CLI 协议字段。

返回：Hook 输出被包装进成功控制响应。

下一站：CLI 使用结果继续、阻止、改写输入或把附加上下文送入后续流程。

Python 类型把字段写成 async_、continue_，线上协议则写成 async、continue。如果绕过 SDK 转换，直接手写协议字典，字段名很容易在这里对不上。

并发不是注册顺序

公开 Options 的注释写明，CLI 会并发分派同一事件上的多个匹配器，所以你不能假定「先注册的 Hook 先改状态，后注册的 Hook 再读取」。顺序靠不住。每个 Hook 都应当独立并保持幂等，记录之间则用 tool_use_id 或 agent_id 关联。

反例：

```
# 不要假设 hook_a 一定先把全局变量设好，hook_b 才开始读取。
hooks = {
    "PreToolUse": [
        HookMatcher(matcher="Bash", hooks=[hook_a]),
        HookMatcher(matcher="Bash", hooks=[hook_b]),
    ]
}
```

如果几条策略必须合成一个确定结论，最好只注册一个负责编排的回调，让它明确收集各条规则的结果，并规定冲突时谁优先。

PreToolUse 与 can_use_tool 怎样配合

你可以把两者分成两个阶段：

1. PreToolUse 负责每次匹配调用的通用策略、审计和输入检查。
2. 若前置 Hook 没有直接形成允许决定，权限规则仍可能让调用进入 can_use_tool 询问。

公开类型的注释指出，只要 PreToolUse 给出允许决定，运行时就会跳过 can_use_tool。两层不能混用。最好让前置 Hook 执行覆盖全部调用的硬规则，再把剩下的交互式决定交给动态权限回调。

可观测性应记录什么

要让一条 Hook 记录以后还能复盘，至少要写下这些内容：

- session_id、tool_use_id、agent_id；
- Hook 事件与 matcher；
- 原始工具名和输入摘要；
- Hook 开始、结束、超时或异常；
- 返回的决定、改写字段和理由；
- 后续工具成功、失败或未执行。

别把敏感文件内容、完整环境变量或密钥直接写进记录。为了看清运行过程，也不能无边界地复制输入输出。

证据边界

沿着 Python Agent SDK 的源码，你可以核对它怎样公开 Hook 类型、转换配置、注册回调 ID，以及怎样分派控制请求并返回响应。至于 Claude Code 何时在内部构造事件、怎样计算权限规则、怎样调用闭源 Sandbox，这个仓库都拿不出源码证据，只能按官方契约描述，不能凭空补出源码站点。

Hook 已经能把一次运行里的事件和具体工具调用对应起来，下一篇再越过单次运行，看 Session 怎样恢复，以及 Transcript 怎样在外部 Store 中保留副本。

下一篇进入跨轮次状态：Session、Resume 与 Store。

Session、Resume 与外部 Store

返回 Claude 课程地图

Hook 已经能在一次运行里按事件串起记录，但要跨轮次继续，你还得知道历史归谁、该从哪里恢复。历史必须有归属。因此这一篇沿着 Session 标识、Resume（恢复）和外部 Store，看状态怎样接着走下去。

只放回上一轮回答还不够。运行时还要认出原来的会话，找到 CLI 保存在本地的 Transcript，确定从哪里续接，并处理分叉、子 Agent 记录和可选的外部持久化。Python SDK 的 SessionStore 也不会拿外部数据库直接替掉 Claude Code 的本地历史文件，它只负责镜像记录，并在恢复时做适配。

先区分三个概念

概念	解决的问题
session_id	当前消息和哪条会话关联
resume / continue_conversation	新进程从哪段已有历史继续
SessionStore	怎样把 Transcript 副本写到外部存储，并在恢复前取回

continue_conversation=True 会选择最近的会话，resume=<id> 会指定一条会话，fork_session=True 则沿用旧历史再开出新的会话分支。它们会改变后续历史写到哪里，不能只当成几个普通的字符串标签。

SessionStore 保存的是不透明记录

外部 Store 不用读懂并改写每一种 Transcript 行，因为公开类型只强制要求 type，其余字段都会作为 JSON 对象原样传过去。

第 1 站：Store 契约要求可往返，不要求字节完全相同

源码：查看 SessionStoreEntry 与 Protocol

```
class SessionStoreEntry(TypedDict, total=False):
    type: Required[str]
    uuid: str
    timestamp: str

class SessionStore(Protocol):
    async def append(self, key, entries) -> None: ...
    async def load(self, key) -> list[SessionStoreEntry] | None: ...
```

调用者：SDK 的 Transcript 镜像器调用 append()，恢复材料化流程调用 load()。

输入：由 project、session 和可选 subpath 组成的 Key，以及不透明 JSON 记录。

状态变化：Adapter 将副本写入外部系统，或从外部系统读取完整会话。

返回：append() 无业务结果；load() 返回记录列表或不存在。

下一站：镜像写入不会替代 CLI 本地写入；恢复时返回值会被写到临时 JSONL。

Protocol 的注释说得很清楚：子进程仍把记录写到本地磁盘，Adapter（适配器）拿到的是另一份副本。至于外部存储保留多久、怎样满足合规要求、何时删除，都由 Adapter 自己负责，SDK 不会跟着本地清理策略自动删掉外部副本。

镜像为什么要批量写

流式运行会频繁产生 Transcript 帧，如果每一帧都同步等待远程数据库，Store 的延迟就会卡住消息处理。落盘没有这么快。SDK 默认采用 batched，等 Result 出现或缓存超过阈值再刷新。eager 虽然更早触发后台刷新，也不能保证每条记录都已经永久落盘。

源码：查看镜像批处理构造

```
eager = flush_mode == "eager"
return TranscriptMirrorBatcher(
    store=store,
    projects_dir=projects_dir,
    on_error=on_error,
    max_pending_entries=0 if eager else MAX_PENDING_ENTRIES,
    max_pending_bytes=0 if eager else MAX_PENDING_BYTES,
)
```

所以看到模型已经输出，不等于外部 Store 也追到了同一位置。需要强一致审计时，要在 Result 出现和运行关闭的路径上等待 flush，同时记下镜像错误。否则数据库里暂时没有记录，只能说明镜像还没写到那里，不能断定本轮什么也没发生。

第 2 站：Client 在启动 CLI 前决定是否材料化

源码：查看长连接 Client 的恢复入口

```
validate_session_store_options(self.options)
self._materialized = (
    await materialize_resume_session(self.options)
    if self._custom_transport is None
    else None
)
```

调用者：ClaudeSDKClient.connect()。

输入：包含恢复选项、Store、工作目录和超时的 Options。

状态变化：先校验不兼容组合；默认 Transport 路径可能创建临时配置目录。

返回：材料化结果保存在 Client 上；自定义 Transport 跳过这一过程。

下一站：材料化 Options 指向临时 CLAUDE_CONFIG_DIR，Transport 再启动 CLI。

自定义 Transport 会跳过这一步，因为调用方已经把它构造好了，SDK 就算改写临时配置，也未必能把配置送到对端。架构说明必须写清这个条件，不能笼统地说「设置 SessionStore 就一定支持 Resume」。

第 3 站：恢复先从 Store 取回，再生成临时本地结构

源码：查看恢复材料化

```
if options.resume is not None:
    resolved = await _load_candidate(store, project_key, options.resume, timeout_s)
else:
    resolved = await _resolve_continue_candidate(store, project_key, timeout_s)

tmp_base = Path(tempfile.mkdtemp(prefix="claude-resume-"))
project_dir = tmp_base / "projects" / project_key
_write_jsonl(project_dir / f"{session_id}.jsonl", entries)
```

调用者：Client 在创建默认子进程前调用 materialize_resume_session()。

输入：Store、明确 Session ID 或 continue 语义、project key 和读取超时。

状态变化：选择候选会话，创建临时目录，写入主 Transcript，并可能材料化子 Agent 记录和认证文件。

返回：MaterializedResume，包含临时目录、实际恢复 ID 和清理协程；无可用历史时返回 None。

下一站：apply_materialized_options() 把 CLI 配置目录指向临时位置，运行结束后清理。

这条链路解释了外部 Store 和 Runtime（运行时）数据库为什么不能画等号：CLI 仍按原来的本地恢复机制读取 JSONL，SDK 只是在启动前取回外部记录，并把它们重新写成 CLI 认识的目录结构。

外部 Key 不能直接当路径

子 Agent 的 subpath 来自外部 Store，如果不做检查就直接拼到临时目录后面，恶意或损坏的数据便可能借助绝对路径、..、Windows 盘符或空字符串逃出目标目录。

第 4 站：恢复时验证 subpath

源码：查看安全路径检查

```
if not subpath:
    return False
if Path(subpath).is_absolute() or subpath.startswith(("/", "\\")):
    return False
if ntpath.splitdrive(subpath)[0]:
    return False
if any(p in (".", "..") for p in re.split(r"[\\/]", subpath)):
    return False
```

调用者：子 Agent Transcript 材料化循环。

输入：Store 返回的 subpath 和目标 Session 目录。

状态变化：只对通过验证的记录创建 JSONL 或元数据文件；危险路径被跳过并告警。

返回：安全布尔判断。

下一站：安全路径对应的记录才写入临时配置，再供 CLI 恢复。

任何由外部持久化系统返回的路径都应当按不可信输入处理，即使数据最初是自己写进去的，也要同时用 POSIX 和 Windows 的路径规则检查。这个检查不能省。

Store 实现至少要满足的约束

1. 同一进程内保持 append 调用顺序。
2. 有 UUID 的记录应按 UUID 幂等写入；无 UUID 的标记不能随意去重。
3. load() 返回的对象要与写入记录深度相等，但 JSON 键顺序可以不同。
4. 超时、部分失败和重试要可观察；不能默默丢批次。
5. 自己实现外部保留、删除、加密和访问控制。
6. Resume 前验证 project key、Session ID 和 subpath。

SDK 提供了 Store conformance 测试入口。你自己为 Redis、PostgreSQL 或对象存储编写 Adapter 时，应先跑契约测试，再用真实并发和故障注入来检验实现。

Session 恢复不是质量证明

恢复成功只能说明运行时加载了旧历史，并开始了一次新的运行。它不能保证旧任务目标依然适用，也不能保证工作区、工具权限和模型版本都没有变化，更不能证明最终答案正确。因此 Artifact（产物）还要记下恢复来源、父 Session、分叉 ID、工作区基线，并接受新的独立 Eval。

Session 和 Store 解释了历史怎样延续，却还没说明一次运行怎样加入新的能力和角色。下一篇会分别看 MCP Server、Agent Definition（智能体定义）和 Skill 各自怎样进入运行。

下一篇：MCP、Agents 与 Skills：三种扩展机制怎样进入运行。

MCP、Agents 与 Skills：三种扩展机制不要混用

返回 Claude 课程地图

SessionStore（会话存储）会把外部记录落成 CLI 能恢复的 JSONL（逐行 JSON），同时校验 subpath，免得恢复时越出临时目录。历史怎样回来已经说清了，这一篇改看各种能力怎样接入同一次运行。

MCP、Agents 和 Skills 都能给一次 Claude 运行加能力，但三者动的地方不同：MCP 通过协议接入工具等能力，Agent 定义新的角色和独立上下文，供主 Agent 委派任务，Skill 则收好操作说明和配套资源，让 Agent 按需发现和加载。要是统统叫成「插件」，你就看不出它们分别在哪个进程里跑、受哪层权限管，以及何时开始和结束。

机制	主要增加什么	典型问题
MCP Server	Tools、Resources、Prompts 等协议能力	在哪运行，怎样授权，结果如何返回
Agent Definition	可由主 Agent 调用的子 Agent 配置	有哪些工具、使用什么 Prompt、如何归因
Skill	按需加载的说明、脚本和资源	从哪些设置源发现，允许哪些名称

MCP 有外部 Server 与进程内 Server

外部 MCP Server 可以按公开配置走 stdio、HTTP 等通道，SDK MCP Server 却直接跑在 Python 应用进程里。这样确实少了一个进程和一层 IPC，但工具函数也能直接碰到应用内存，所以它一旦崩溃或阻塞，宿主就会跟着受影响，敏感状态也留在同一个信任边界内。

第 1 站：@tool 把 Python 函数变成 SDK MCP 工具

源码：查看 Tool 装饰器契约

```
def tool(
    name: str,
    description: str,
    input_schema: type | dict[str, Any],
    annotations: _McpToolAnnotations | None = None,
) -> Callable[..., SdkMcpTool[Any]]:
```

调用者：应用开发者装饰一个异步 Python 函数。

输入：稳定工具名、给模型看的描述、输入 Schema 和可选 MCP 注解。

状态变化：函数被包装成 SdkMcpTool，还没有启动 Server，也没有暴露给模型。

返回：可加入 SDK MCP Server 的工具对象。

下一站：create_sdk_mcp_server() 把一个或多个 Tool 注册到进程内 Server。

模型会同时看到描述和 Schema，所以描述写得含糊，它就容易选错工具，而 Schema 放得过宽，它就可能填入更多危险参数。readOnlyHint、destructiveHint 这类注解只是提示，真正的权限判断和输入校验还得由执行端完成。

第 2 站：进程内 Server 仍通过 MCP 协议连接

源码：查看 SDK MCP Server 创建

```
def create_sdk_mcp_server(
    name: str,
    version: str = "1.0.0",
    tools: list[SdkMcpTool[Any]] | None = None,
```

```
) -> McpSdkServerConfig:
    ...
```

调用者：业务应用完成 Tool 定义后创建 Server 配置。

输入：Server 名称、版本和 Tool 列表。

状态变化：构建普通 mcp.server.Server 实例并登记 Handler。

返回：type="sdk" 的 McpSdkServerConfig，可放入 ClaudeAgentOptions.mcp_servers。

下一站：InternalClient 识别 SDK 类型，把实例交给 Query；CLI 通过控制协议转发 MCP 消息。

「进程内」只是在说 Server 部署在哪里，它没有绕过 MCP。请求照样带着 Server 名称、JSON-RPC 消息和 Tool Schema，失败也会返回协议化的错误结果，只是 SDK 会通过控制协议来回桥接这些消息。

第 3 站：Client 将不同配置送往不同路径

源码：查看扩展配置装配

```
for name, config in configured_options.mcp_servers.items():
    if isinstance(config, dict) and config.get("type") == "sdk":
        sdk_mcp_servers[name] = config["instance"]

agents_dict = {
    name: {k: v for k, v in asdict(agent_def).items() if v is not None}
    for name, agent_def in configured_options.agents.items()
}

query = Query(
    sdk_mcp_servers=sdk_mcp_servers,
    agents=agents_dict,
    skills=configured_options.skills,
    ...
)
```

调用者：InternalClient 在 Transport 连接后装配 Query。

输入：MCP 配置、Agent 定义和 Skill 选择。

状态变化：进程内 MCP 实例保留在 Python；Agent 转成可序列化字典；Skill 作为初始化选择保存。

返回：持有三类扩展信息的 Query。

下一站：initialize 发送 Agent 和 Skill 配置；运行时 MCP 消息按 Server 名称回到本地实例。

这段代码把三条路分得很清楚：Agent Definition（智能体定义）不会被当成 MCP Server，Skill 的名称也不会转成 Python 回调，所以你不能用同一条执行链来解释它们。

第 4 站：MCP 请求通过控制协议回到 Python

源码：查看 MCP 控制请求分支

```
elif subtype == "mcp_message":
    server_name = request_data.get("server_name")
    mcp_message = request_data.get("message")
    mcp_response = await self._handle_sdk_mcp_request(
        server_name, mcp_message
    )
    response_data = {"mcp_response": mcp_response}
```

调用者：CLI 发来 mcp_message 控制请求，Query 分派它。

输入：Server 名称和 MCP JSON-RPC 消息。

状态变化：对应进程内 Server 执行 Handler；通知类消息也会得到控制层确认。

返回：MCP 响应被包装到 control_response。

下一站：CLI 将 Tool 结果继续送入 Agent 运行；应用还应记录执行结果和副作用。

Python Tool 如果卡住，控制协议就只能跟着等。因此，高风险或不可信的实现更适合放到外部 MCP Server，然后再用独立进程、容器、网络策略和资源限额给它划出清晰边界。

Agent Definition 增加委派角色

ClaudeAgentOptions.agents 按名称收好每个 Agent Definition，主 Agent 再通过工具把任务交给它，而每份定义通常会写明描述、Prompt、可用工具和模型。这里真正要防的是多出一个能发起调用的主体，而不只是多了一段 Prompt：

子 Agent 的 ToolUse 要用 agent_id 与主线程分开归因；

工具权限不能因为「来自内部子 Agent」就默认可信；

并行子 Agent 的 Hook 和消息会交错，不能靠输出顺序关联；

主 Agent 成功不代表所有子任务都成功，Artifact 要保留各自终态。

initialize 会把 Agent 定义发送出去，这能证明公开配置是怎样交给 CLI 的，却不足以支持你去推测 Claude Code 内部用了什么调度器类或队列。

Skill 增加按需知识与流程

公开字段 skills 有三种有意义的值：

None：SDK 不做自动配置，但 CLI 默认仍可能发现 Skill；

[]：显式不向列表暴露任何 Skill；

"all" 或名称列表：启用全部或指定 Skill。

源码：查看 Skills 语义

当你设置 Skills 时，SDK 还会补上相应的允许规则和 Settings 来源，因此前面课程追过的有效 allowed_tools 会跟着改变，can_use_tool 甚至可能被遮蔽。Skill 不只是一份 Prompt 文件，它一旦启用，就会同时影响能否发现对应内容，以及运行时怎样配权限。

一项需求该选哪种机制

需求一：读取公司工单系统

这类需求优先用 MCP Tool，因为它用 Schema 约清输入输出，还能把认证和 API 调用放在 Server 里隔离运行，并为每次调用单独留下记录。

需求二：让专门的审查角色并行检查安全问题

这时可以用 Agent Definition，但要给它限定 Prompt 和工具，再按 Agent ID 分别收集轨迹和结果，别把并行输出混在一起。

需求三：让 Agent 按团队发布流程操作

这种情况适合用 Skill，把说明、脚本和模板收成 Agent 能找到的资源，不过脚本真正运行时，仍然要受工具权限和系统隔离约束。

一项功能也可以同时用到三者：Skill 告诉 Agent 何时发布，子 Agent 专门做审查，MCP Tool 再去调发布系统。一旦这样组合，你就得分层记下权限决定和失败位置，因为一句「插件执行失败」根本说不清哪一层出了问题。

扩展安全清单

1. MCP Tool 使用最小 Schema，并在 Handler 内再次校验输入。
2. 进程内 MCP 只承载可信代码；不可信代码使用外部隔离。
3. Agent Definition 明确工具集合、成本与停止条件。
4. 子 Agent 事件保留 agent_id，不要按到达顺序归因。
5. Skill 来源和名称使用显式允许列表，避免无意加载项目外内容。
6. 三类扩展的失败分别进入 Artifact，再由独立 Eval 判断任务影响。

三条扩展路径现在已经分清，下一篇再换到 TypeScript，看这些公开契约还能追到哪里，以及哪些细节因为没有源码而必须留白。

下一篇：TypeScript SDK：公开契约能讲到哪里。

TypeScript SDK：公开契约能讲到哪里

返回 Claude 课程地图

前一篇已经把 MCP、Agent Definition 和 Skill 分到三条不同的装配与执行路径上，但换成 TypeScript 之后，你不能直接假定这三条路还按同样的结构运作。证据到哪，我们就读到哪。

写跨语言 SDK 时，最容易犯的错是先在 Python 里找到一段内部实现，再把同一套结构安到 TypeScript 头上。当前锁定的 TypeScript 仓库没有公开 SDK 主体的 Runtime（运行时）源码，因此这一篇只读实际看得到的 README、CHANGELOG、SessionStore（会话存储）示例和测试。边界就在这里。

当前锁定树里实际有什么

提交 48275071e804139579fabada9bb8d90cfe02b062 包含：

- 项目 README、CHANGELOG 和许可证；
- GitHub 工作流与维护脚本；
- PostgreSQL、Redis、S3 三套 SessionStore 参考 Adapter；
- Adapter 的共享契约测试与各后端测试。

这个仓库看不到 SDK Runtime 常见的 src 主体，也没有公开包入口和控制协议的实现，所以下面几类材料各自能证明什么，必须分开写：

证据	可以说明	不可以说明
README	项目定位和公开指引	Runtime 内部调用顺序
CHANGELOG	某版本声称新增或修复什么	该分支具体怎样实现
示例 Adapter	SessionStore 怎样映射到后端	SDK 怎样调度所有 Store 回调
共享测试	示例认定的行为契约	npm 包全部边界条件
Python 源码	Python 提交中的实现	TypeScript 必然同构

契约对齐与实现对齐不是一回事

Python 和 TypeScript 都可以向外提供 query、权限、Hooks、MCP 与 SessionStore，这只能说两边暴露的契约属于同一类。要判断内部的类、状态机、并发控制和资源清理是否真的对齐，你必须同时拿到两种语言在同一版本下的 Runtime 实现证据。

CHANGELOG 能提醒你哪些地方可能带来升级风险，例如它记过什么时候会发出 canUseTool 遮蔽警告、哪些 Tools 对模型可见、SessionStore 怎样重试，也记过 stdin 过早关闭这类问题：查看锁定 CHANGELOG 中的权限回调变化。但它记的是版本对外声明，并没有把对应函数的实现交给我们。

真实可读案例：PostgreSQL SessionStore

在 TypeScript 锁定树里，外部 Store Adapter（适配器）最适合顺着往下读，因为这部分同时留下了完整源码和契约测试。

第 1 站：示例把 Store 交给 query()，并用 Session ID 恢复

源码：查看 PostgreSQL 端到端示例

```
const store = new PostgresSessionStore({ pool })
await store.ensureSchema()

for await (const m of query({
  prompt,
  options: { sessionStore: store, resume, maxTurns: 1 },
})) {
  if (m.type === 'system' && m.subtype === 'init') sessionId = m.session_id
}
```

调用者：示例应用的 run() 函数。

输入：Prompt、Store、可选 Resume ID 和轮数限制。

状态变化：SDK 运行会话并镜像记录；示例从 init 消息提取 Session ID。

返回：run() 返回该 ID，下一次调用用它恢复。

下一站：Store Adapter 将 append/load 映射到 PostgreSQL 表。

这段示例只把公开用法展示给你，你无法由此判断 TypeScript Runtime 是否像 Python 那样，会调某个函数把外部记录落到临时目录。

第 2 站：Adapter 用递增 ID 保存顺序

源码：查看 PostgreSQL Adapter

```
export class PostgresSessionStore implements SessionStore {
  async append(key: SessionKey, entries: SessionStoreEntry[]): Promise<void> {
    ...
  }

  async load(key: SessionKey): Promise<SessionStoreEntry[] | null> {
    ... ORDER BY id
  }
}
```

调用者：Agent SDK 通过公开 Store 契约调用；测试也会直接调用。

输入：projectKey、sessionId、可选 subpath 和记录列表。

状态变化：每条 JSON 记录写入一行，BIGSERIAL 提供该表中的稳定顺序。

返回：load() 按 ID 还原记录，不存在时返回 null。

下一站：共享 conformance suite 检查追加顺序、Key 隔离和可选行为。

写入记录时，这里会绑定参数值，同时检查表名是否只含合法的 SQL 标识符字符。由于表名无法像普通值那样绑定成参数，构造 Adapter 时就必须先校验，这一道检查会直接挡住表名注入。

第 3 站：共享测试明确最小行为

源码：查看 SessionStore 契约测试

```
type SessionStore = {
  append(key: SessionKey, entries: SessionStoreEntry[]): Promise<void>
  load(key: SessionKey): Promise<SessionStoreEntry[] | null>
  listSessions?(projectKey: string): Promise<...>
```

```
  delete?(key: SessionKey): Promise<void>
  listSubkeys?(key: ...): Promise<string[]>
}
```

调用者：每个后端测试用一个全新 Store Factory 注册相同测试集。

输入：隔离的 Adapter 实例和构造好的记录序列。

状态变化：测试执行多次 append、load 和 Key 隔离操作。

返回：断言深度相等、顺序一致、不存在返回 null 等契约。

下一站：后端特有测试再检查 SQL、S3 分片或 Redis 索引细节。

共享 suite 先给 JSON 对象键排序，再比较两边是否深度相等，因此 PostgreSQL JSONB 重排对象键不会让测试误报。Python Store 的注释也表达了同样的行为，但证据只够我们说「两份公开契约在这一点相符」，还不能说两个 Runtime 复用了同一份实现。

怎样写跨语言对照表

对照每项能力时，至少要把下面四列摆在一起，这样你才不会把某一边的公开声明误当成两边共用的实现证据：

能力	Python 锁定源码	TypeScript 锁定树	可下结论
Query 公开表面	有实现	README/CHANGELOG 声明	API 目的相近，内部未知
权限回调	有配置和控制分派源码	CHANGELOG 记录	行为契约可比较，状态机不可比较
SessionStore 示例	有 Protocol 与材料化源码	有三种 Adapter 和测试	Store 数据契约可对照
关闭与子进程回收	有 Transport 源码	无主体 Runtime	不写 TypeScript 内部顺序

如果以后锁定树放出了 Runtime 源码，要先更新来源提交，再按新提交重走一遍调用链，否则你很容易把新版本的实现倒填进旧版本结论。这一步不能省。

选择语言 SDK 时看什么

不要只因为这份课程能读到 Python 内部源码，就认定生产系统也必须用 Python。真正选语言 SDK 时，你还得考虑：

应用语言与部署环境；

SDK 当前公开能力和版本稳定性；

团队能否测试权限、Hooks、Session 和关闭语义；

依赖与许可证要求；

是否需要自定义 Store、MCP 或浏览器表面。

源码和证据是否透明，会影响你评估风险，但它只是其中一项，选型最后还是要回到你的实际需求上。

跨语言对照必须停在可见契约的边界上。课程的最后一篇会回到真实运行，看错误究竟发生在哪一层，哪些可观测证据应该留下，以及独立 Eval（评测）怎样判断任务做得对不对。

下一篇收束 Claude 课程：错误分类、可观测性与独立 Eval 接缝。

错误分类、可观测性与独立 Eval 接缝

返回 Claude 课程地图

上一篇对照 TypeScript 时，只读到 README、CHANGELOG、SessionStore 示例和契约测试能支持的地方就停下了。这些可见契约能告诉你运行对外暴露了什么，任务有没有通过独立 Eval（评测），还要另外判断。这两层不能混。

课程走到最后，你要问的已经不只是「Claude 有没有输出」，还要问运行在哪一层失败、留下了哪些证据、能不能安全重试，以及最终产物是否真的满足目标。要是所有失败都被算成「模型答错」，Harness 就无法知道该改哪一层，重试还可能把真实的产品问题藏起来。

先把失败分层

层次	例子	通常由谁处理
启动与连接	找不到 CLI、版本不兼容、子进程无法启动	Transport / 部署
协议与解析	JSON 损坏、消息字段缺失、控制请求超时	SDK / 集成层
权限与策略	Tool 被拒绝、Hook 阻止、Sandbox 不可用	Harness 策略
工具执行	命令退出非零、文件不存在、MCP Handler 异常	Tool / 环境
Agent 运行	最大轮数、预算、Interrupt、API 错误	Runtime / 调用方
任务质量	修改不正确、测试失败、违反约束	独立 Eval

同一次运行可能在多层同时留下事实，比如 CLI 先发出一个 is_error=true 的 Result，随后又以非零状态退出。Artifact（产物）要保留这两条原始证据，但统计根因时必须把它们关联到同一次失败，别误算成两个任务各失败了一次。

第 1 站：SDK 错误类型保留不同责任层

源码：查看错误类型

```
class ClaudeSDKError(Exception): ...
class CLIConnectionError(ClaudeSDKError): ...
class CLINotFoundError(CLIConnectionError): ...

class ProcessError(ClaudeSDKError):
    def __init__(self, message, exit_code=None, stderr=None):
        self.exit_code = exit_code
        self.stderr = stderr
```

调用者：公开 query 或 Client 的连接、读取和关闭路径抛出这些异常。

输入：连接失败原因、进程退出码和可选 stderr。

状态变化：错误对象保存结构化字段，而不是只拼接一段文本。

返回：异常沿调用栈传播，调用方按类型决定重试、降级或停止。

下一站：若 CLI 已先发布错误 Result，SDK 可以提供更具体的 ResultError。

CLINotFoundError 说明部署没有准备好 CLI，所以多调几次模型不会解决问题，你得回去修部署。协议解析失败也不能直接算成用户任务做错，重试前先看看原输入会不会稳定触发同一故障，否则只会把确定失败再演一遍。

第 2 站：ResultError 把终态负载带进异常

源码：查看 ResultError

```
class ResultError(ProcessError):
    ...
    self.subtype = data.get("subtype")
    self.errors = _normalize_result_errors(data.get("errors"))
    self.api_error_status = data.get("api_error_status")
    self.terminal_reason = data.get("terminal_reason")
    self.session_id = data.get("session_id")
```

调用者：读取流程发现终端错误 Result 后，在进程非零退出路径构造该异常。

输入：Result 原始负载和退出码。

状态变化：终止原因、HTTP 状态、Session 和错误列表被结构化保留。

返回：兼容 ProcessError 的更具体异常。

下一站：调用方按 terminal_reason 和状态判断是否可重试，并把原始 Result 与异常归为同一根事件。

比如 API 只是暂时过载，你可以把它当成基础设施故障，在有次数上限的前提下退避重试。如果运行已经撞上最大轮数，就应该回头检查怎样拆任务、何时停止，而权限拒绝则必须交回策略层处理，不能暗中换成绕过模式。

非致命错误也要进入 Artifact

如果 SessionStore 没有镜像成功，本地会话不会因此停下，因为 Transcript 已经写到本地，但系统会发出 MirrorErrorMessage。如果你只盯着最终 Result，可能会看到一次「成功」的运行，却没发现外部审计副本已经断了一截。

源码：查看 MirrorErrorMessage 语义

RateLimitEvent 也有类似情况，它可能在用量真正撞上硬限制之前就发出 warning。所以可观测系统要分清两种情况：一种是运行仍在继续，只有证据或剩余容量降级，另一种才是运行已经终止。

第 3 站：Result 暴露运行指标，但不是评分器

源码：查看 ResultMessage 字段

```
class ResultMessage:
    subtype: str
    duration_ms: int
    duration_api_ms: int
    is_error: bool
    num_turns: int
    total_cost_usd: float | None = None
    permission_denials: list[Any] | None = None
    errors: list[str] | None = None
    terminal_reason: str | None = None
```

调用者：消息解析器从 CLI Result 帧构造，应用消费者接收。

输入：时延、轮数、使用量、拒绝、错误和终止原因。

状态变化：协议事实聚合成一个终态对象，不修改工作区产物。

返回：应用可记录成本、性能和运行结果。

下一站：独立 Eval 读取完整 Artifact 与执行环境，判断任务正确性。

duration_ms、num_turns 和成本能用来看效率，permission_denials 能解释某个动作为什么没有执行，terminal_reason 则会告诉你运行是正常完成、中途中断，还是因为预算耗尽而停止。但要判断任务做对没有，你仍然要直接检查目标文件和测试结果。指标只是指标。

可观测性要围绕一次 Trial 组织

可以把一次完整的业务任务记为 Trial（评测试次），如果中间遇到可恢复的基础设施故障，每次重试再单独记为 Attempt（尝试）。

```
Trial
  ├── 输入任务与工作区基线
  ├── Attempt 1：连接失败，可恢复
  ├── Attempt 2：产生完整运行 Artifact
  ├── 最终产物与副作用
  └── 独立 Eval 结果
```

产品失败不能靠反复重试「刷成通过」。比如 Agent 第一遍误删了文件，第二遍又碰巧修好，Trial 依然得记住第一遍造成的副作用。只有当某个 Attempt 明确遇到可恢复的基础设施故障，而且还没有对业务产生副作用时，系统才适合自动重试。

每条事件至少关联：

trial_id、attempt_id、session_id；

message UUID、tool_use_id、agent_id；

SDK 与 CLI 版本、模型标识；

权限决定、Hook 结果、工具结果；

工作区差异、测试输出和最终 Result；

脱敏后的错误与耗时。

别把完整 Prompt、文件内容、环境变量和凭据一股脑塞进 Telemetry（遥测）。你应该先给数据分级，设定保留期，然后再按敏感度和排查需求决定每个字段该存原文、哈希还是摘要。

Eval 是 Harness 的横切能力

这个仓库的主线是 Agent Harness，Eval 会横跨运行时的各个关键位置收集证据，没有必要再另写一套与 Harness 平行的百科：

```
任务合同
  → Harness 运行与工具副作用
  → Artifact（轨迹 + 产物 + 环境事实）
  → 独立 Scorer
  → 质量、约束、效率与证据完整性
```

以「修复运费边界错误」为例，你可以从下面几个方向同时检查，让任务正确性、执行约束和证据是否齐全都有明确落点：

结果检查：边界测试从失败变为通过；

约束检查：没有修改测试和无关文件；

过程检查：高风险工具调用经过策略，轨迹没有缺口；

效率指标：轮数、时延、成本和重试次数；

证据完整性：Result、Diff、测试输出和错误均可关联。

Scorer（评分器）必须独立检查产物，不能照单全收模型的自述，因为 Assistant 说「已经修好」只代表它发出了这条消息，还没有任何外部检查为这个结论背书。

一套可执行的判定顺序

```
def evaluate(artifact):
    if not artifact.protocol_complete:
        return "inconclusive"
    if artifact.forbidden_side_effects:
        return "fail"
    if not artifact.required_tests_passed:
        return "fail"
    if not artifact.expected_diff:
        return "fail"
    return "pass"
```

这段只是教学用的伪实现，不来自 Claude SDK 上游源码。它把判定顺序写得很清楚：证据没收齐就不能直接判成功，已经违反约束也不能靠测试通过抵消，只有先确认任务做对了，才能把效率拿出来单独报告。

重试决策表

情况	默认处置
CLI 不存在、配置确定错误	阻止并修复部署，不重试
429/过载且无副作用	有上限退避重试
权限策略拒绝	保留拒绝，回到授权或任务设计
工具已产生部分副作用后超时	先检查幂等与现场副作用
最大轮数	评估任务拆分、Prompt 与停止条件
测试失败	记为任务失败，不作为基础设施重试
Artifact 缺关键轨迹	inconclusive，不能宣称通过

到这里，Claude 课程已经把外部链路走完：证据边界 → 入口与 Transport → 消息 → 工具可见性 → 权限 → Hooks → Session → 扩展 → 跨语言契约 → Eval。整条路都停在公开 SDK 能支持的地方，没有借机补写 Claude Code 从未公开的内部实现。

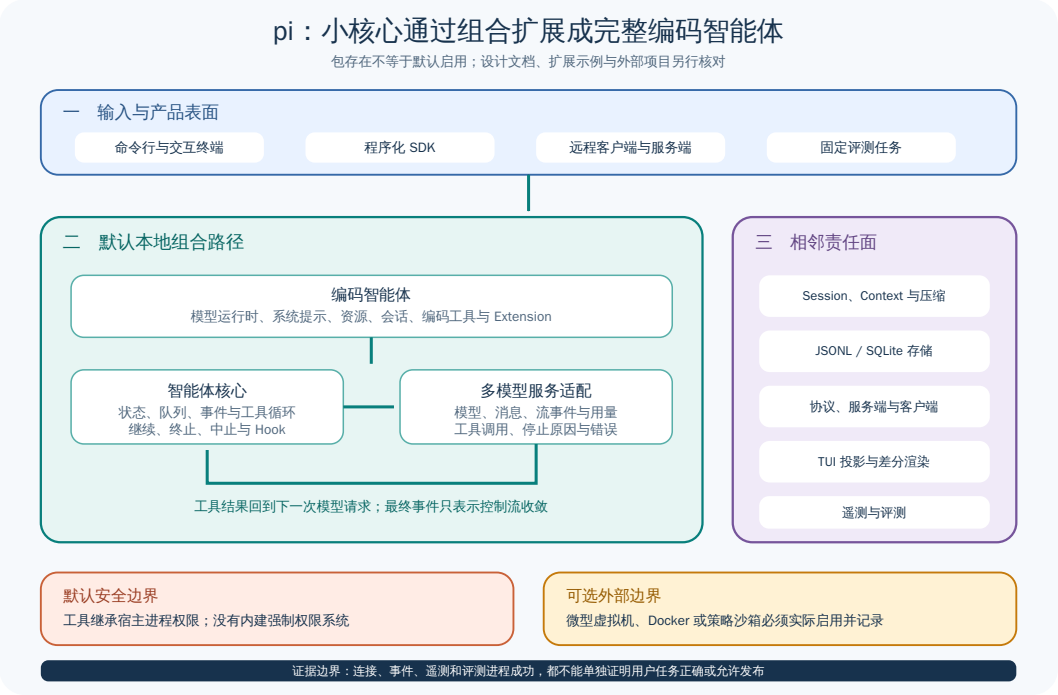
回到课程地图时，你可以再沿着这条外链定位每个机制，然后对照错误所在层次、Artifact 和独立 Eval，检查它们怎样一起撑起完整 Trial。证据链在这里合上了。

返回 Claude 课程地图

pi 源码课程

返回学习入口

先看清三层。pi 的价值在于分层清楚，底层 AI 包负责 Provider 与流式消息，Agent 包提供最小循环，而 Coding Agent 再把工具、Session 和交互表面组合起来。课程会沿着这条组合关系向上阅读，同时分清核心运行时代码、设计文档、扩展示例和外部环境。



这条课程适合谁

如果你想从较小的核心开始，而不是立刻钻进大型产品代码，pi 很适合作为第一条源码课程。Starter 可以先读 AI、Agent 和 Coding Agent 三层，Builder 再进入队列、工具批次、Session Tree 和 Protocol，而 Maintainer 最后检查 Lease、权限与宿主隔离边界。

锁定来源

课程基于 pi 提交 c1279a65b3ef6b0b19950ed1771d5933241c240f：

- Agent Loop
- Agent 状态对象
- Coding Agent SDK
- Agent Loop 测试

先看一项任务

循环从这里启动。Coding Agent 接收任务后，会先组合 AI Provider、Agent Core 和工具，而 Agent Loop 消费流式响应时，一旦遇到工具调用，就把它交给注册工具执行，再将结果加入消息。Session 和协议表面围绕这条核心循环工作，为它补上持久化与外部控制能力。

Coding Agent / Protocol / TUI
→ Agent 对象
→ Agent Loop
→ AI Provider Stream
→ Tool 执行
→ Message 与 Session

图中的 Coding Agent 组合了 Agent Core，并没有另起一套平行循环，而 Protocol 和 TUI 也只是面向不同场景的控制表面。

仓库地图

区域	首轮关注点
packages/ai	Provider、消息与流式事件的统一形状
packages/agent	状态、Loop、工具执行和事件
packages/coding-agent	编码工具、Prompt、Session 和 SDK
Protocol 与 Server/Client	外部应用怎样驱动 Agent
CLI/TUI	用户交互和权限提示
Telemetry 与 Evals	可观察信息和外部验证接缝

pi 默认沿用启动它的宿主权限，而权限提示和工具接口本身不能证明文件、进程、网络或凭据已经隔离，所以要说明真实边界，还需要把外部容器或 Sandbox 一并纳入判断。

三层读法

Starter：前三篇，读懂 Provider 流怎样进入 Agent Loop，并被 Coding Agent 组合。

Builder：加入 Session Tree、Compaction、Protocol 和 Client Lease，理解长期任务状态。

Maintainer：检查取消、工具失败、权限提示、宿主隔离和 Telemetry 的证据边界。

阅读顺序

1. 三层架构、Provider 与流归一化
2. Agent Loop、双队列与工具批次
3. Coding Agent、Prompt 与 Extensions
4. Session Tree、Compaction 与 JSONL
5. Protocol、Server 与 Client Lease
6. CLI、TUI、权限与外部隔离
7. Telemetry、Evals 与证据边界

用贯穿任务复盘

复盘可以从 Coding Agent 组合 AI Provider、Agent Core 和工具开始，跟着 read/edit/test 依次经过流归一化、双层 Loop、工具批次和 Session Tree，再假设客户端通过 Protocol 远程控制同一 Session，说明 Attach、Event、Response 和 Lease 怎样共同保持身份。走完这条链路后，再区分宿主权限、交互批准、Telemetry 和 Eval Score 各自说明了什么。

复盘时必须分清 AI Stream 的 done、Agent Run 收敛、TUI 显示完成和任务通过——它们描述的是不同层次的终态。

完成课程后应该能回答

- AI、Agent 和 Coding Agent 三层分别提供什么；
- Agent Loop 怎样处理流式文本和工具调用；
- Coding Agent 怎样组合 Prompt、工具和 Session；
- Protocol 表面怎样复用核心状态；
- 权限提示与宿主系统权限为什么不同；
- Telemetry 与 Eval 在源码中能观察哪些事实。

从第一篇开始：三层架构、Provider 与流归一化

从 AI 包到 Agent Core，再到 Coding Agent

返回 pi 课程地图

pi 最适合按「分层组合」的思路来读，因为 packages/ai 先统一模型、消息和流，packages/agent 再提供与编码无关的最小 Agent Loop，最后由 packages/coding-agent 加入文件、Shell、Prompt、Session、Extension 和交互表面。

```
Coding Agent：编码工具、Prompt、Session、TUI/RPC
                ↓
Agent Core：状态、队列、Loop、Tool Events
                ↓
AI：Provider、Model、Message、Event Stream
```

这三层解决的问题并不相同：AI Stream 收敛时，只能说明一次模型响应已经完成，等到当前队列与工具批次都收敛之后，Agent Loop 才算结束，而 Coding Agent 还要继续照管会话、工作区和产品输出。

第 1 站：共同类型保留统一字段和 Provider 身份

源码：查看 AI 消息与停止类型

```
export interface ToolCall {
  type: 'toolCall'
  id: string
  name: string
  arguments: Record<string, any>
}

export type StopReason =
  | 'pending' | 'stop' | 'length' | 'toolUse'
  | 'error' | 'aborted' | 'deferred'

export interface AssistantMessage {
  provider: ProviderId
  model: string
  usage: Usage
  stopReason: StopReason
}
```

调用者：具体 Provider Adapter 生成流事件，Agent Core 消费 Assistant Message。

输入：各模型 SDK 的 Chunk、Tool Call、Usage 和原始结束原因。

状态变化：把 Provider 差异归一到共同 Message/Event 形状，同时保留 Provider 与 Model。

返回：Agent Core 不依赖某一家 SDK 的 Stream。

下一站：Agent Loop 根据 StopReason 与 Tool Calls 决定执行工具还是结束本次采样。

统一类型虽然降低了上层的复杂度，却也会压缩 Provider 之间的差异——因此在调试性能或排查错误时，仍应保留原始 HTTP 状态、Provider Stop Reason 和经过安全处理的响应详情。差异还得留下。

为什么不是让 Agent Core 直接调用每家 SDK

Agent Loop 真正关心的是「有哪些消息块、有没有工具调用、为什么停止、花了多少资源」，而不是每家 SDK 如何命名 Chunk 类。如果 Core 直接按 Provider 写分支，那么每新增一种模型，认证、流解析和循环代码都要跟着修改，就连错误处理也很难保持一致。统一层会把变化隔离在 Adapter 内，但 Provider 特有的缓存、推理块或安全终止不能因此被抹掉，它们仍要通过扩展字段或诊断信息保留下来。

第 2 站：模型目录、Provider 实现和认证缺一不可

源码：查看 Models 路由

```
const provider = this.providers.get(model.provider)
if (!provider) throw new ModelsError('provider', ...)

const resolution = await this.getAuth(model, ...)
if (!resolution) throw new ModelsError('auth', ...)

return provider.stream(requestModel, context, requestOptions)
```

调用者：Agent Loop 需要一次模型流时调用 Models Runtime。

输入：Model Entry、消息 Context、认证选项与 AbortSignal。

状态变化：查 Provider 实现并解析认证；尚未产生 Agent 消息副作用。

返回：统一的 Assistant Event Stream，或 Provider/Auth 错误。

下一站：Agent Loop 订阅 start、text、thinking、toolcall、done/error。

模型出现在目录里，只能证明它的元数据可以被发现，因为 Provider 尚未注册、认证缺失、区域不可用或 API 不兼容等问题，都可能让请求在真正建立 Stream 之前就失败。

事件流的终态只属于一次模型响应

事件流工具只会把 done 和 error 识别为终结事件，但在 done 之前仍可能产生 Tool Call，而 Agent Core 一旦收到这种调用，就要执行工具并再次调用模型。因此，done 不是整个任务的完成信号。

源码：查看 Event Stream 实现

```
// start / thinking / text / toolcall 是中间事件。
// done 与 error 关闭本次流。
```

调用者：Provider Adapter 推送事件，Agent Loop 异步迭代。

输入：统一 Assistant Stream Events。

状态变化：累积一条 Assistant Message 并向观察者广播增量。

返回：流终态中的 Assistant Message 或 Error。

下一站：Agent Loop 检查 Tool Calls、StopReason 与队列。

不要把新的 Harness Scaffold 当成现行主循环

锁定版本里还有一个面向未来接口的 AgentHarness Scaffold，只是其中多项操作都会明确抛出 HarnessNotImplemented，所以当前真正可工作的调用链仍然是 Agent + agent-loop.ts。

源码：查看未实现错误

```
export class HarnessNotImplemented extends Error {
  constructor(operation: string) {
    super(`AgentHarness.${operation} is not implemented yet`)
  }
}
```

设计文档能帮我们理解目标方向，但在分析实际运行行为时，还是应当沿着现行 Agent Loop 追踪，然后用测试确认对应版本确实执行了这条路径。

回到运费任务

用户任务进入 Coding Agent 后，由 packages/coding-agent 选择模型和工具，而 Agent Core 只会收到已经归一化的 Context、Model 与 Tool Definitions。当模型流产生 read Tool Call 时，AI 层会先把供应商事件还原成共同 Tool Call，然后才由 Agent 层决定是否执行。即使 Provider 流以 done 结束，只要消息中还有 Tool Call，整个任务就没有结束。

练习：判断是哪一层的问题

如果模型目录里能看到目标模型，但运行时报「没有认证」，而另一台机器能完成模型响应，却找不到 edit 工具，就不能把两种失败都归因于 Agent Loop。前者应先查 AI 层的 Provider/Auth 路由，后者则应查 Coding Agent 的工具装配。

下一篇：Agent Loop、双队列与工具批次。

Agent Loop、Steering、Follow-up 与工具批次

返回 pi 课程地图

pi Agent Core 把最小循环写得很直接：它先准备消息、调用模型并流式发出事件，一旦收到 Tool Calls，就执行工具、追加 Tool Results，然后继续采样。除了这条主循环，它还维护 Steering 与 Follow-up 两个输入队列，前者处理运行中的转向，后者承接本轮结束后的追加工作。

第 1 站：两个队列拥有独立 Drain 语义

源码：查看 Agent 队列

```
class PendingMessageQueue {
  enqueue(...)
  drain(...)
  clear(...)
}

private readonly steeringQueue: PendingMessageQueue
private readonly followUpQueue: PendingMessageQueue
```

调用者：宿主在 Agent 运行时调用 Steer 或 Follow-up；Loop 拉取消息。

输入：新的 Agent Message 与队列模式。

状态变化：消息进入不同队列，可按各自策略一次取一条或全部取出。

返回：Loop 下一次检查时得到的 Pending Messages。

下一站：Steering 加入当前工具/模型循环；Follow-up 在原本准备结束时重新开启循环。

两个队列分开之后，Steering 可以及时改变正在进行的任务，而 Follow-up 只会在当前闭环收敛后进入循环——因此它不会在工具批次中间破坏已经建立的因果顺序。

Steering 和 Follow-up 解决的是时序，不只是两种名称

假设 Agent 正在并行读取实现和测试，此时用户补充「不要修改测试文件」，这条 Steering 就应尽快进入下一次模型决策。另一句「完成后再生成变更说明」属于 Follow-up，它要等到当前工具闭环收敛后才开始。如果把两者塞进同一个队列，Harness 不是过晚响应约束，就会在工具结果尚未齐全时提前开始新目标。

第 2 站：双层循环定义了输入插入点

源码：查看 Agent Loop

```
while (true) {
  while (hasMoreToolCalls || pendingMessages.length > 0) {
    // 模型采样、工具执行、Steering
  }

  const followUpMessages =
```

```
    (await config.getFollowUpMessages?.()) || []
  if (followUpMessages.length > 0) {
    pendingMessages = followUpMessages
    continue
  }
  break
}
```

调用者：Agent.prompt() 或产品 Session。

输入：初始消息、模型、工具表、队列回调与 AbortSignal。

状态变化：依次追加 Assistant、Tool Result 与 Steering/Follow-up Messages。

返回：完整本次运行消息与 Agent Events。

下一站：Coding Agent Session 持久化并投影到 UI/RPC。

外层循环负责处理 Follow-up，内层循环则负责本次模型与工具的闭环，所以模型给出 stop 时，只结束了当前这一次采样。只有当内外两层都不再满足继续条件时，Agent Run 才真正收敛。

长度截断的 Tool Call 不能执行

如果 Assistant Message 的 StopReason 是 length，Tool Call Arguments 就可能被截成一份看似 JSON 的半成品，因此 pi 不会冒险执行这组参数，而是生成错误 Tool Result，再让模型处理一次。

源码：查看长度截断处理

```
// length 结束时不执行本批 Tool Calls，
// 返回说明参数可能因输出 Token 上限被截断的错误结果。
```

调用者：Tool Batch 准备阶段。

输入：Assistant Message、Tool Calls 与 StopReason。

状态变化：不产生工具副作用；构造 Call ID 对应的错误 Tool Results。

返回：模型可读错误消息。

下一站：再次采样，让模型缩短或重建调用。

并行批次怎样决定是否终止

无论工具并行还是串行，最终都会返回 { messages, terminate }，但只有当批次非空、且每个最终结果都显式要求 terminate 时，shouldTerminateToolBatch() 才会终止 Loop。如果只有一个快速工具要求结束，它不能取消那些仍在运行、且没有要求结束的兄弟调用。

源码：查看批次终止规则

```
// 批次全部结果 terminate=true 时才整体终止。
```

调用者：Tool Batch 执行结束后的 Loop。

输入：每个工具的最终消息与 Terminate 标志。

状态变化：决定继续下一次模型采样还是停止。

返回：批次消息和整体 Terminate。

下一站：追加 Tool Results，或结束当前 Agent Run。

beforeToolCall 可以在调用发生之前阻断它，afterToolCall 则可以调整结果和 Usage，但这两个扩展点都不会自动带来用户审批或 OS 隔离。具体 Coding Agent 如何注册 Extension，要到下一篇再看。

失败路径怎样进入下一轮

pi 的关键设计并不是「工具必须成功」，而是无论成功还是失败，工具都必须产生与 Call ID 对应的结果。因此，未知工具、参数截断、Extension 阻断和实现抛错，都应转换成模型可读的失败观察，同时保留结构化错误供宿主判断。除非协议已经无法继续、用户取消或整个批次明确终止，否则 Loop 仍会继续采样。

回到运费任务

第一轮先读取 shipping.ts，等读取结果回填后，第二轮才会提出编辑，而编辑完成时，Follow-up 里的「同时更新说明」仍要等测试闭环结束才能开始。如果用户临时发来「不要改测试」，这条消息会作为 Steering，在下一次模型调用之前插入。这样既能让新约束及时生效，又不会把已经启动的工具批次伪装成尚未发生。

练习：什么情况下批次可以提前结束

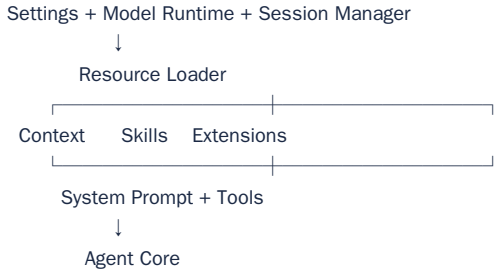
并行批次有三个调用：读取源码返回 terminate=false，读取测试返回 terminate=false，一个自定义工具返回 terminate=true。此时整个批次是否终止？

下一篇：Coding Agent、Prompt 与 Extensions。

Coding Agent 怎样把 Prompt、Tools、Session 与 Extensions 装起来

返回 pi 课程地图

Agent Core 本身并不知道「编码任务」意味着什么，所以 packages/coding-agent 要先通过 SDK 组合 Model Runtime、Settings、Session Manager、Resource Loader、内建工具和 Extension，然后再把它们交给 Agent Core。



第 1 站：装配先建立服务，再加载资源快照

源码：查看 createAgentSession() 装配

```
const modelRuntime = options.modelRuntime ?? await ModelRuntime.create(...)
const settingsManager = options.settingsManager ?? SettingsManager.create(...)
const sessionManager = options.sessionManager ?? SessionManager.create(...)

resourceLoader = new DefaultResourceLoader(...)
await resourceLoader.reload()
```

调用者：CLI、RPC Server 或嵌入 SDK 创建 Coding Agent Session。

输入：工作目录、模型/设置覆盖、Session 选择和临时 Extensions。

状态变化：创建或复用运行时服务，加载当前资源快照。

返回：可 Prompt、Steer、Compact、Fork 的 Agent Session。

下一站：Prompt Builder 按活动工具和资源生成模型指令。

在模型请求开始之前先 Reload 资源，才能避免工具表已经更新、Prompt 却还在描述旧工具的错位，而运行中一旦发生动态刷新，两者也必须同步更新。

装配顺序为什么会影响模型行为

模型输入里的自然语言指南、Tool Schema 和实际 Tool Registry 必须来自同一份资源快照。如果先构造 Prompt，随后 Extension 又替换了工具，模型就可能按照旧说明，去调用一个已经不存在的名称。反过来，如果只更新 Schema，却不更新 Context File 或 Skill，模型又可能继续遵循过期的工作流。因此装配不是一段可有可无的启动样板代码——它实际上定义了本次 Session 的能力边界。

第 2 站：System Prompt 根据活动工具生成

源码：查看 Prompt Builder

```
const tools = selectedTools || ['read', 'bash', 'edit', 'write']
const visibleTools = tools.filter((name) => !!toolSnippets?.[name])

for (const guideline of promptGuidelines ?? []) {
```

```
  // Trim、去重并追加指南
}
```

调用者：Session 创建与资源/工具变化后的 Prompt 构造。

输入：活动工具名、工具摘要、指南、项目 Context、Skills 和 CWD。

状态变化：生成只描述实际可见工具的 System Prompt。

返回：Agent Core 使用的系统指令字符串。

下一站：模型请求与 Tool Schema 一起发送。

如果自定义工具没有 Prompt Snippet，它可能已经能被调用，却没有出现在文字指南里，因为 Schema 才是模型调用工具的正式接口。反过来，Prompt 里写得再谨慎，也不等于权限得到了强制。

Resource Loader 生成的是一次一致快照

Reload 会在同一轮处理 Package、项目 Trust、Extension、Skill、Prompt Template、Theme、Context File 和 Prompt Override，但 noExtensions、noSkills、noPromptTemplates 各自控制的来源并不相同，所以不能把其中一个开关理解成「所有扩展资源都关闭」。

源码：查看 Resource Loader

```
// reload() 解析各资源来源，形成当前 Session 使用的快照与诊断。
```

调用者：SDK 装配、配置变化和显式 Reload。

输入：Trust、Packages、CLI 临时 Extension 与资源开关。

状态变化：替换当前资源集合并保存加载诊断。

返回：Extensions、Skills、Prompts、Context 与 Theme。

下一站：Extension Runner、Prompt Builder 与 Command Registry。

Extension 可以改写调用前后与 Context

Extension Loader 会收集 Handler、Tool、Command 和 Provider，然后由 Runner 按加载顺序逐个调用。在这条链上，Tool Call Handler 可以 Block，Tool Result Handler 可以改写 Content、Details、Error 和 Usage，而 Context Handler 的输出会继续传给下一个 Extension。

源码：查看 Extension Runner

```
// Tool Call handlers 顺序执行，block 时停止。
// Tool Result 与 Context handlers 以前一项输出作为后一项输入。
```

调用者：Agent Core 的 Before/After Tool Hook 与 Context Transform。

输入：Tool Call、Tool Result 或消息副本。

状态变化：可能阻断副作用、改写模型可见结果或 Context 投影。

返回：最终 Block 决定、Result 或 Messages。

下一站：Tool Executor、Session History 或模型请求。

Extension 属于高信任的进程内代码，因为它既可以注册新 Provider 或工具，又拥有与宿主进程等同的权限，所以必须单独审计来源并做好外部隔离。Prompt 代替不了这些措施。

三种扩展作用不要混在一起

Extension 既可以增加能力，也可以改变控制流和模型观察：注册 Tool 属于增加能力，用 beforeToolCall 阻断属于改变控制流，通过 Context Handler 改写 Messages 则属于观察变换。正因为这三种作用可以叠加，审计时才必须分别记录「模型原本提出什么」「扩展允许执行什么」和「模型最终看见什么」，否则一个 Extension 就能让 Trace 同时失去原始意图和真实结果。

回到运费任务

在运费任务里，项目 Context 可以告诉模型如何运行测试，Skill 可以说明仓库的修复流程，Tool Registry 则暴露 read、edit 和 bash。安全 Extension 可以在编辑测试文件之前阻断调用，但它不能只把返回文本改成「已阻断」，实际上却仍然执行写入。调用前 Hook 决定副作用是否发生，而调用后 Hook 只影响结果投影。两者责任不同。

练习：找出不一致快照

资源刷新之后，如果 System Prompt 还写着「使用 patch 工具」，Tool Schema 里却只有 edit，问题就不在模型能力，而在资源快照已经失去一致性。此时应检查 Prompt Builder 与工具注册是否消费了同一轮 Resource Loader 结果，并为刷新路径增加回归测试。

下一篇：Session Tree、Compaction 与 JSONL。

Session Tree、Context 投影、Compaction 与 JSONL

返回 pi 课程地图

pi Coding Agent 的 Session 并不是一条从头排到尾的消息数组，而是由 Session Entries 组成的可分支树，当前 Leaf 会从树中选出正在使用的路径。只有 Message、Custom Message、Branch Summary 和 Compaction 等 Entry 才会沿这条路径投影成模型 Context，而 JSONL Repo 保存的则是完整 Entry 序列。两者不是同一份数据。

```
Session Entry Tree
  ├── 当前活动路径 → UI / 导航
  └── Context Entries → 模型 Messages
                  |
                  └── Compaction Summary 替代旧前缀

完整 Entries → JSONL Session Repo
```

第 1 站：活动路径与模型 Context 分开构造

源码：查看 Session Context 构造

```
const path = buildSessionPath(entries, leafId, byId)
const messages = buildContextEntries(entries, leafId, byId)
  .flatMap(sessionEntryToContextMessages)

return { messages, thinkingLevel, model }
```

调用者：打开、Resume、Fork、Navigate 或 Compact 后的 Session Manager。

输入：全部 Entries、当前 Leaf ID 与索引。

状态变化：不改原树；计算活动路径和模型可见 Entry 子集。

返回：Messages、Thinking Level 与 Model。

下一站：Agent Core 用这些消息继续 Prompt。

Label、普通 Custom Entry 和部分 UI 状态即使写进了 Session 文件，也不一定会进入模型，因此调试「为何模型忘了」时，应该检查 buildContextEntries() 的结果，而不能只凭文件里有没有那一行作判断。

Compaction 选择最新 Summary 与后续尾部

每次压缩都会写入一个 Summary Entry，而同一条路径经过多次压缩以后，Context 会从最新的有效 Summary 开始，再接上它后面的消息。旧 Entry 并未消失，审计和创建分支时仍然可以回到 Session Tree 中找到它们。

源码：查看 Entry 到 Context Message 的转换

```
// message / custom_message / branch_summary / compaction -> Context
// 其他 Entry -> []
```

调用者：buildContextEntries()。

输入：沿当前 Leaf 的 Session Entries。

状态变化：用最新 Compaction/Branch Summary 建立有界模型视图。

返回：按角色排序的 Agent Messages。

下一站：模型请求；完整 JSONL 不被摘要覆盖。

Summary 是模型生成的有损结果，所以内容里应该保留精确路径、未完成事项以及关键 Tool Result 的引用，并在压缩完成后用确定性问题核对关键信息是否还在。不要只看 Token 是否下降。

为什么保留树，而不是把旧消息直接删除

Compaction 要解决的是当前模型窗口装不下历史的问题，而不是替审计存储腾出空间。完整的 Entry Tree 还要承担多种职责——回看记录、Fork、比较分支和解释摘要来源。一旦直接把旧前缀从存储中删除，不但无法再追溯「模型为什么得出这个摘要」，也会失去从较早节点创建新分支的能力。

第 2 站：JSONL Repo 处理尾部损坏和追加队列

源码：查看 JSONL Repo 的创建、打开与 Fork、查看尾部修复，以及查看追加队列。

```
// Create / Open / List / Delete / Fork
// 修复缺失换行或损坏尾行，拒绝非法中间行。
// 追加失败后，队列仍可继续处理后续请求。
```

调用者：Session Manager 创建、打开、追加和 Fork。

输入：CWD、Session ID、Entries 与目标路径。

状态变化：同一进程内预留 {cwd,id}，串行追加 JSONL；打开时验证/修复尾部。

返回：Open Session、Writer 或类型化损坏错误。

下一站：Session Tree 重建并选择 Leaf。

尾行能够修复，并不意味着文件里的任何损坏都可以忽略，因为中间行一旦损坏，Entry 之间的父子关系也可能随之断裂。此时应拒绝打开，并保留原文件副本。

Resume 不能恢复外部世界

Session 可以恢复消息、模型选择、Thinking Level 和分支信息，却无法让已经退出的进程重新运行，也不能还原远端请求、外部数据库或工作区后来发生的修改。因此 Resume 完成以后，还要重新检查 CWD、Git Diff、工具可用性和凭据。

同一进程里的 Destination Reservation 也不能充当跨进程 Lease，如果多个进程会共享同一份 Session 存储，就还需要文件锁、Fencing 或其他后端事务来协调写入。

回到运费任务

如果任务在编辑文件以后中断，Session Tree 里可能同时留有原始 Tool Call、工具开始事件、完成结果以及后来生成的摘要。恢复时不能因为摘要写着「已经修改」就认定工作完成，而要把活动分支上的工具完成记录与当前文件差异对照起来。若新分支是从编辑前的节点 Fork 出来的，它就不该继承编辑后的 Tool Result，即使那条结果仍保存在同一个 JSONL 文件中。

练习：解释「文件里有，模型却不知道」

你在 Session JSONL 中找到一条 Label 和一条旧分支的 Message，但下一轮模型都没有看到。应该先检查什么？

下一篇：Protocol、Server 与 Client Lease。

Protocol、Server 与 Client Lease 如何远程控制同一 Agent

返回 pi 课程地图

pi Protocol 把进程内的 Agent Session 变成了一个可以由外部 Client 驱动的状态机，而消息进入传输层以后，会先按 Frame 切分，再经过 CBOR 解码和 Schema 校验。Command/Response 负责同步控制，Event 传递运行中的进度，Attach/Detach Lease 则管理 Client 对 Session 的占用。

```
Client Command
  ↓ Frame → CBOR → Schema
Server Coordinator → Attached Session → Agent
  ├→ Response：命令结算与 Snapshot
  └→ Event：模型、工具与状态进度
```

第 1 站：Frame Boundary 与 Message Codec 分两层

源码：查看 Protocol Codec

```
const frame = encodeFrame(
  encodeCbor(validated, { maxByteLength: maxFrameLength }),
)

for (const frame of this.frames.push(chunk)) {
  messages.push(this.parse(
    decodeCbor(frame, { maxByteLength: this.maxFrameLength }),
  ))
}
```

调用者：Client/Server Transport 收到任意大小 Byte Chunk。

输入：部分 Frame、多个粘连 Frame 或待发送 Message。

状态变化：Frame Decoder 缓冲 Header/Payload；完整后再解 CBOR 并做 Schema 校验。

返回：一个或多个类型化 Protocol Messages。

下一站：Request Router 或 Pending Request Map。

Frame 完整只能说明这一段字节的边界没有问题，接下来的 CBOR 解码和 Schema 校验仍然可能失败，而 end() 如果发现缓冲区里还留着半个 Frame，就应该报告截断错误。不能静默丢弃。

Response 还要核对 Command 名称

Client 收到响应以后不只会按 Request ID 查找 Pending Promise，还会确认 Response Result 中的 Command 与原请求一致，因此即使服务端或连接层错配了 ID，另一种命令的 Payload 也不会被交给错误的 Decoder。

源码：查看 Protocol Client

```
// 按 requestId 找 Pending Request，
// 再验证 result.command 与原 Command Name 一致。
```

调用者：Client Message Loop。

输入：Response、Pending Request Table。

状态变化：正确响应结算 Promise；错配使连接进入错误状态。

返回：类型化 Command Result。

下一站：SDK 调用者更新本地 Snapshot。

第 2 站：Prompt 前必须先 Attach

Server Session Coordinator 会在 Create 完成后自动 Attach，而在执行 Prompt、Steer 和 Abort 之前，都会先调用 requireAttached() 确认当前连接仍然占用 Session。命令结算时的 Snapshot 由同步 Response 携带，运行过程则通过 Events 持续推送。

源码：查看 Session Coordinator

```
// create -> attach
// prompt / steer / abort -> requireAttached()
// progress -> events；command completion -> response snapshot
```

调用者：Server Command Router。

输入：Session ID 与 Prompt/Steer/Abort Command。

状态变化：验证当前连接拥有 Attachment，随后调用 Agent Session。

返回：Command Result 和 Snapshot。

下一站：Client 同时合并 Events 与 Response。

Event 与 Response 在网络上可能交错到达，所以 Client 应该依据 Revision 和 Session ID 合并状态，而不能把消息抵达本机的先后直接当成全局因果顺序。

为什么 Response 和 Event 必须共存

Prompt 命令可能运行很久，如果只在结束时返回 Response，客户端就无法及时显示模型增量、工具开始或权限请求，而如果只发送 Event，调用者又不知道某个命令何时正式结算，也无法判断失败属于哪次请求。两种消息各管一层——Response 给出请求级的完成语义，Event 提供运行中的观察窗口，但它们都要关联到同一个 Session 和 Revision。

Shared 与 Exclusive Lease 解决同一 Client 内的所有权

同一个 Client 持有多个 Shared Lease 时，远端只需要 Attach 一次，等最后一个 Shared Lease 释放才会 Detach，而 Exclusive Lease 会与任何 Shared Lease 互斥。若 Detach 失败，显式释放可以把 Lease 恢复到 Active 状态以便重试，Dispose 则采用尽力清理语义。

这套 Lease 只处理同一个 Client 内部的所有权，并不会自动解决多个独立 Client 争用同一个 Server Session 的问题，因此跨 Client 的独占、断线超时和 Fencing 仍然要由 Server Runtime 定义。

回到运费任务

客户端 Attach 后提交 Prompt，随后可能先后收到「读取工具开始」与「读取完成」事件，直到最后才拿到 Prompt 命令的 Response Snapshot。一旦网络重连，Client 就不能把旧连接上迟到的事件直接应用到新 Snapshot，而要根据 Session ID、Revision 或服务端的重放规则判断这条事件是否仍然有效。

练习：不要用到达顺序重建状态

客户端先收到 Revision 12 的 Response，随后收到 Revision 11 的工具完成 Event。界面应否把状态退回 Revision 11？

下一篇：CLI、TUI、权限与外部隔离。

CLI、TUI、JSON、RPC 与真实权限边界

返回 pi 课程地图

pi Coding Agent 可以运行在 Interactive TUI、Print、JSON 和 RPC 模式，这些产品表面虽然共享同一个 Agent Session，却各自遵守不同的 I/O 契约。pi 默认会继承启动进程拥有的文件、网络、子进程与凭据权限。项目 Trust 和工具提示仍不是 OS Sandbox。

第 1 站：产品模式由参数和 TTY 条件选择

源码：查看模式选择

```
if (parsed.mode === 'rpc') return 'rpc'
if (parsed.mode === 'json') return 'json'
if (parsed.print || !stdinIsTTY || !stdoutIsTTY) return 'print'
return 'interactive'
```

调用者：Coding Agent 进程入口。

输入：CLI 参数与 stdin/stdout TTY 状态。

状态变化：选择单次文本、事件 JSON、双向 RPC 或交互 UI。

返回：对应 Surface Runner。

下一站：Runner 创建 Agent Session 并连接输入输出。

JSON 和 RPC 都可能逐行输出 JSON，不过前者承载的是单向 Agent Events，后者使用的却是双向 Command/Response/Event 协议，所以消费者不能因为外观相似就混用 Parser。

产品表面改变交互，不改变环境事实

Interactive 模式可以在终端里弹出确认，Print 模式可能需要预设策略，RPC 模式则能把问题交给远端客户端回答，而这些差异决定的只是「谁回答批准请求」，不会自动改变 Bash 最终在哪台机器上运行，也不会改变执行它的用户身份。权限模型必须跨产品表面保持同一语义，否则同一个工具从 TUI 和 RPC 发起调用时，就会落入不同的安全边界。

TUI Render 不是 Agent Event 计数器

TUI 每次更新时会先渲染 Components 与 Overlays，然后再根据屏幕状态选择全量重绘或差分写入，因此终端行数、Spinner 停止和「Done」卡片都只是 UI 投影，既不代表模型采样次数，也不能充当任务验收结果。

源码：查看 TUI Main Screen

```
let newLines = this.render(width)
if (this.hasOverlayEntries) newLines = this.compositeOverlays(...)
if (widthChanged) {
  fullRender(true)
  return
}
```

调用者：Terminal Event/Resize/Agent Event 触发 UI 更新。

输入：组件状态、宽度和 Overlay。

状态变化：更新屏幕缓存并写终端。

返回：用户可见界面。

下一站：用户输入继续驱动 Session。

Project Trust 与 OS Sandbox 的区别

Project Trust 控制的是要不要自动加载仓库提供的 Context、Extension 或其他资源，从而降低一打开恶意项目就执行代码的风险，但它不会限制用户主动调用 Bash/Write 以后，进程究竟能访问哪些路径。

默认情况下，工具会直接使用 pi 进程的权限运行，如果需要真实隔离，就要引入外部容器或 Sandbox Extension，并明确区分哪些工具已经重定向到隔离环境，哪些工具仍然留在宿主执行。

第 2 站：Sandbox Extension 存在明确本地回退

源码：查看 Sandbox Extension

```
// sandboxEnabled 或 sandboxInitialized 为 false 时，
// Bash 调用回退到 localBash。
```

调用者：Example Extension 注册的 Bash Tool Handler。

输入：命令、Sandbox 开关和初始化结果。

状态变化：选择隔离执行器或本地 Bash。

返回：命令输出。

下一站：Agent Core 接收 Tool Result。

Example 只是展示 Extension 能做到什么，并不代表 Sandbox 默认启用，而当平台不支持、依赖缺失、初始化失败或用户主动关闭时，工具都可能回到宿主执行。日志必须写清实际路径。

容器化也要说明挂载和网络

Plain Docker、Gondolin、OpenShell 等方案面对的威胁模型并不相同，如果宿主工作区以读写方式挂载，容器里的进程仍然能够修改文件，而只重定向 Bash 也约束不了 Extension 自己调用的 Node API。网络网关可以限制出站访问，却不会自动保护已经挂载进去的凭据。

回到运费任务

TUI 中的「允许编辑」只表示产品批准了这次 Tool Call，真正的写入仍然可能由宿主 Node 进程完成。即使启用了 Sandbox Extension，也要确认三件事——edit 与 bash 是否都被重定向、工作区采用什么挂载方式，以及初始化失败后会不会回退到本地，因为只有 Trace 记录了实际执行路径，用户才能判断这次修改究竟发生在哪里。

练习：识别危险回退

日志只写「Sandbox 初始化失败，继续执行」，随后测试正常通过。为什么这不能被视为安全降级？

下一篇：Telemetry、Evals 与证据边界。

Telemetry、Eval Harness 与分数分别负责什么

返回 pi 课程地图

pi 同时提供通用 Telemetry 接口和 Eval 相关包，其中 Telemetry 记录运行生命周期，Eval Harness 在受控目录运行 Coding Agent 并保存 Artifact，而 Summary 只消费已有 Score。它们不会自动判断任务对错。

第 1 站：Telemetry 接口没有 Score 语义

源码：查看通用 Telemetry

```
startSpan(options, callback)
addEvent(name, attributes)
setAttributes(attributes)
setStatus(status)
```

调用者：AI、Agent、Coding Agent 和 Protocol 运行点。

输入：名称、时间、关联属性与错误状态。

状态变化：向注入的 Telemetry Context 发送观察数据。

返回：Span 结果或原业务函数返回值。

下一站：Exporter、Trace Viewer 或 Eval Artifact Collector。

Agent Telemetry 虽然可以记录 Provider、Model、StopReason、Token、Cost 和首块延迟，却不会因此给这些字段附加 Rubric 或 Pass Threshold。

可观测性为什么不能顺便充当评分器

Telemetry 的首要职责是忠实记录运行——它不能因为某个指标「看起来不好」就改写事实。Eval 做的是另一件事，它必须依据冻结任务和判定规则，把多个事实解释成通过、失败或无法判断。一旦把两者合并，观察字段的变化就可能无意中改变评分口径，评分逻辑也可能为了报表便利而丢失原始事件。

源码：查看 Agent Harness Telemetry 字段

```
'pi.ai.response.stop_reason'
'pi.ai.usage.total_tokens'
'pi.ai.stream.time_to_first_chunk_ms'
```

第 2 站：Eval Harness 负责运行和留证

源码：查看 pi Eval Harness

```
setArtifact('runId', sessionManager.getSessionId())
setArtifact(
  PI_SESSION_SNAPSHOT_ARTIFACT,
  await readFile(sessionPath, 'utf8'),
)
```

调用者：Eval Case Runner。

输入：任务、选择的模型与 Source Workspace。

状态变化：创建临时目录和无 Extension Session，运行 Prompt，转换 Transcript Events。

返回：Session JSONL、Source 和 Run ID 等 Artifacts。

下一站：确定性或模型 Scorer 读取产物。

关闭 Extensions 是为了减少未锁定变量，并不意味着真实产品只能用这种方式评测。如果目标就是评估某个 Extension，就必须把它的版本和配置一并写进 Target。

Summary 不会凭 Transcript 自动创造分数

源码：查看 Eval Summary

```
// scored Observation 进入比较；
// unscored 被归类为 missing-score。
```

调用者：Eval Suite 结束后的汇总器。

输入：已有 Score 的 Observation 与无分数项。

状态变化：分类、比较和聚合。

返回：报告数据。

下一站：人或发布流程解释结果。

任务评分要想可信，还必须明确测试命令、Diff 约束、基础设施失败处理和重复运行规则，而 Assistant StopReason、工具 Success、TUI Done 或 Telemetry Status 无论哪一项都不能单独替代这些规则。

回到运费任务

Telemetry 可以说明模型调用了两次、执行过 edit 和 bash，以及整个过程耗时多少，而 Eval 必须进入独立工作区，检查金额 100 的断言、回归测试和测试文件是否被篡改。如果模型 API 超时，并且没有形成可判定的补丁，就应把这次运行记录为基础设施失败或无法判断，不能等到某次重试「通过」后再删除早先事实。早先事实仍要保留。

练习：为一个分数列出最小证据

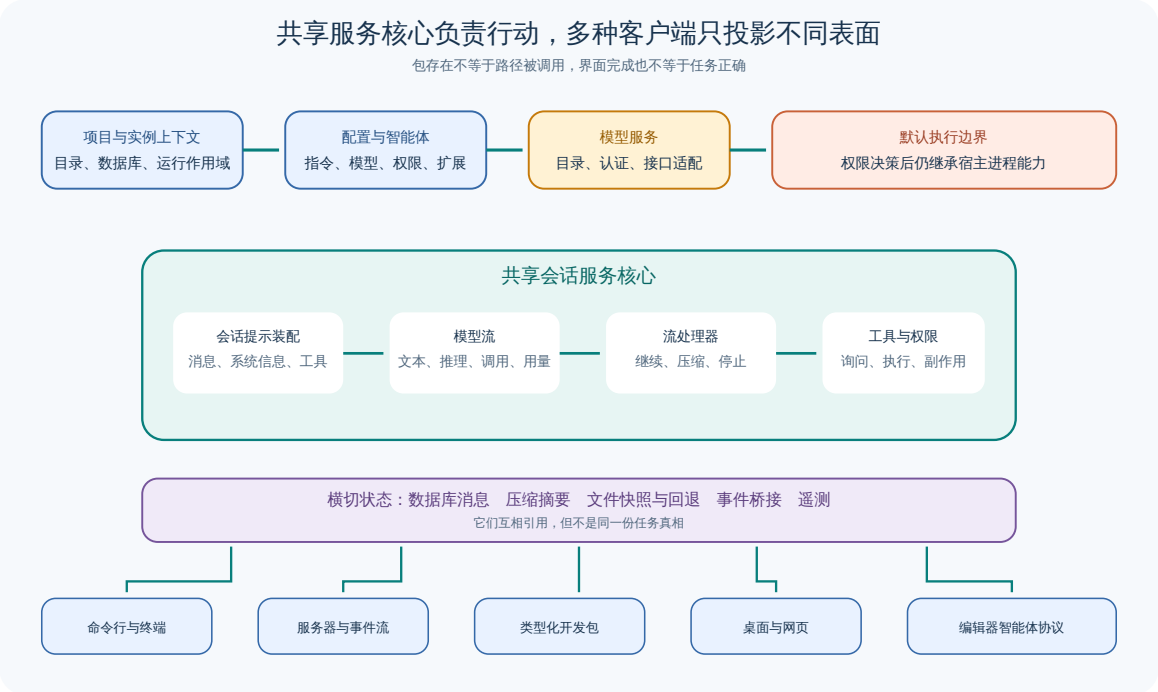
如果要给运费任务标记为通过，至少需要哪些 Artifact？

到这里可以回到 pi 课程地图，或进入 OpenCode 课程 比较服务化 Session 架构。

OpenCode 源码课程

返回学习入口

OpenCode 围绕服务化 Session 组织运行过程：Project 和 Config 先选定这次运行要用的环境，Provider（模型提供商）接好模型接口，Session Prompt 再驱动主循环，让 Processor（处理器）接住模型流和工具结果，最后由 Server 与 Protocol 把这套核心能力交给多个客户端使用。



这条课程适合谁

如果你已经读完基础导读，想看 TUI（终端用户界面）、Desktop、Web 和协议客户端怎样共用一个 Agent 核心，OpenCode 很适合作为下一站。它没有停在最小 Loop 上，还把 Project、Server 和持久化边界摆到了同一条运行链里。

锁定来源

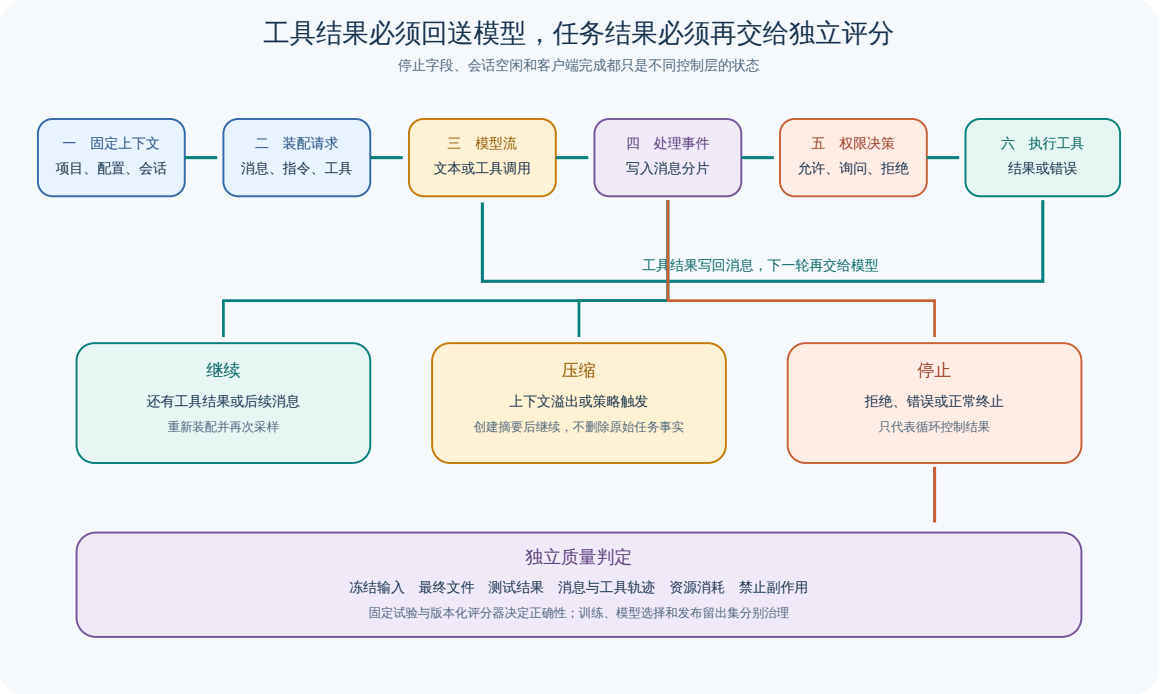
课程基于 OpenCode 提交 3a31c4ea801915c0b050df4b3842997ea62b6e93：

- 程序入口
- Session Prompt
- Session Processor
- 模型调用层
- Protocol API

先看一项任务

客户端先在某个 Project 里创建或选中 Session，Prompt 主链随后把消息、Agent 配置和工具组到一起，再向 Provider 发起模型请求。模型开始流式返回后，Processor 会逐项处理事件和工具状态，权限询问则跟着 Session 事件送到客户端，用户给出的结果还会写回原来的 Session。

TUI / Desktop / Web / ACP
→ Server / Protocol
→ Project 与 Session
→ Prompt / LLM
→ Processor 与 Tools
→ Permission / Question
→ Message Store 与下一轮



图里的客户端只负责展示询问和结果，服务核心才真正维护 Session 状态。即使同时接入多个客户端，也不会因此复制出多套 Agent Loop（智能体循环）。

仓库地图

区域	首轮关注点
packages/opencode/src/project 与 Config	工作区和配置怎样建立
packages/opencode/src/provider	多 Provider 怎样归一化
packages/opencode/src/session	Prompt、LLM、Processor、消息和压缩
Tools 与 Permission	工具副作用和用户询问
Agent、Skills、Plugins、MCP、LSP	运行时扩展和上下文来源
packages/server 与 packages/protocol	多客户端怎样共享服务核心
TUI、Desktop、Web、ACP	同一事件在不同表面怎样呈现

Permission（权限）规则可以决定产品是否放行某个动作，用户询问也能在执行前把流程停下来，但这两层都不会自动变成操作系统 Sandbox，进程到底被隔离到什么程度，仍要看宿主和另行配置的执行环境。这两层不能混。

三层读法

Starter：先读 Runtime、Session Prompt 和工具权限，跟完一次请求。

Builder：继续读 Storage、Compaction、Revert 与扩展，解释状态怎样被改写。

Maintainer：检查 Server、Protocol、Share、Telemetry 与多客户端一致性边界。

阅读顺序

1. Runtime、Project、Config 与 Provider
2. Session Prompt、LLM 与 Processor
3. Tools、Permission、Question 与 Patch
4. Storage、Compaction 与 Revert
5. Agents、Skills、Plugins、MCP 与 LSP
6. Server、Protocol 与多产品表面
7. Share、Telemetry 与 Eval 边界

用贯穿任务复盘

复盘时从客户端选择 Server/Directory 开始，沿着一次真实运行往下讲：Project、Config 和 Provider 怎样把实例建起来，Session Prompt 怎样驱动 LLM 与 Processor，Permission Question 怎样送到客户端，Message 和 Part 怎样落盘，Compaction（上下文压缩）与 Revert 又怎样改写模型下一轮真正能看到的历史。最后再看各种产品表面怎样从同一个 Server 读取事实。

读到最后，你还要逐个判断 Share 同步、OpenTelemetry Span、Session Idle 和独立测试到底各自能证明什么。它们都能关联到同一个 Session，但这几类证据不能互相替代。

完成课程后应该能回答

Project、Config、Provider 和 Session 的创建顺序。

Prompt 怎样形成模型请求，Processor 怎样消费结果。

工具和权限事件怎样跨服务端与客户端。

Message、History、Compaction 和 Revert 如何改变 Session。

多种产品表面共享了哪些协议和状态。

Share、Telemetry 和测试能支持哪些核对，哪些仍需外部 Eval。

从第一篇开始：Runtime、Project、Config 与 Provider

Runtime、Project、Config 与 Provider 的启动顺序

返回 OpenCode 课程地图

OpenCode 不会让每个界面各自创建一套 Agent。它先按 Directory 确认 Project 和 Instance，依次读取全局、项目、托管与系统来源，把它们合成有效 Config，再据此筛选 Provider（模型提供商）并建立 Map，Session 随后都引用这套实例服务。

```
Directory → Project / Instance
          ↓
    Config 来源合并与权限覆盖
          ↓
      Provider Map + Models
          ↓
        Session 创建
```

第 1 站：Config Merge 还会处理数组和来源范围

源码：查看 Config 合并

```
const merge = (source: string, next: Info, kind?: ConfigPlugin.Scope) => {
  result = mergeConfigConcatArrays(result, next)
}
```

调用者：Instance 初始化时的 Config Loader。

输入：全局、项目、Plugin、托管目录和系统管理配置。

状态变化：按来源顺序合并，数组字段使用拼接/去重语义，权限还有专门覆盖。

返回：当前 Project 的有效 Config。

下一站：Agent、Provider、Permission、Plugin 与 Server 初始化。

所以，只看到项目配置最后写了 deny，还不能断定它就是最终规则，因为系统管理配置可能随后改写这个值，Instructions、Plugins 等数组也不会像普通标量那样直接覆盖。排查时要把字段的 Provenance 一并输出。

为什么配置来源必须保留身份

合并后的值只能告诉你「现在是什么」，却看不出谁写入了它、后来的来源凭什么把它改掉。当个人、仓库和组织管理配置同时出现时，Loader 必须保留每个关键字段来自哪里、按什么顺序合并，才能阻止恶意仓库改掉管理员对 Provider 或 Permission 的禁令。只打印最终 JSON，往往追不到真正动过这个值的来源。

第 2 站：Provider 先受白名单和黑名单过滤

源码：查看 Provider 过滤

```
if (enabled && !enabled.has(providerID)) return false
if (disabled.has(providerID)) return false
```

调用者：Provider Loader 构建当前实例 Provider Map。

输入：发现的 Provider、Enabled/Disabled 集合与凭据环境。

状态变化：过滤不允许的 Provider，解析每个 Provider 的 Models 与 SDK 配置。

返回：Session 可使用的 Provider Map。

下一站：Agent/Session 按 providerID/modelID 查实际 Model。

模型目录里出现某个名字，并不能证明实例已经加载了对应 Provider，因为禁用规则、缺失的认证、尚未启动的 Plugin，甚至 SDK 解析失败，都可能把它挡在实例 Map 之外。看得到名字，仍可能用不了。

第 3 站：查到 Model 后还要解析具体 SDK

源码：查看 getModel() 与 SDK 解析

```
const provider = s.providers[providerID]
if (!provider) return yield* new ModelNotFoundError(...)

const sdk = await resolveSDK(model, s, envs)
```

调用者：Session Prompt 在每次运行前解析 Agent 选择的模型。

输入：Provider ID、Model ID、实例状态与环境凭据。

状态变化：解析语言模型适配器及 Provider 选项。

返回：可交给 LLM 层的 Model Language 实现。

下一站：Session LLM 构造 System、Messages 与 Tools。

ModelNotFound 与底层 SDK 抛出的 NoSuchModelError 不在同一个阶段：前者说明实例目录没有查到这个模型，后者则说明目录虽然查到了，具体 Adapter（适配器）却无法把它交给模型服务。

Project Identity 为什么重要

OpenCode 会按 Server、Directory、Project 和 Session 分层保存配置、数据库与事件，UI 也沿这些身份找到自己该看的内容。远程 Attach 时，Directory 指向服务端路径，所以客户端出现两个同名目录，并不能证明它们属于同一个工作区。为了让后续操作找回原来的运行边界，每份 Artifact（产物）都应同时保存 Server Identity 和 Project/Directory。

回到运费任务

用户在客户端选好目录后，服务端会按这个路径建立 Project 和 Instance，读入项目里的 Instructions 与权限规则，再让 Session 选择 Provider/Model。如果远端服务实际打开的并非用户以为的仓库，即使模型和工具都能正常运行，修改仍会落进错误的环境。因此，任务刚开始就要显示 Directory 和 Server Identity。

练习：为什么模型存在却仍然不可用

配置文件声明了一个模型，Provider 目录也包含它，但 Session 抛出 ModelNotFound。按顺序应检查哪些位置？

下一篇：Session Prompt、LLM 与 Processor。

Session Prompt、LLM 与 Processor 如何形成主循环

返回 OpenCode 课程地图

OpenCode 没把整个 Agent Loop（智能体循环）塞进一个函数，而是让 session/prompt.ts 推进循环并整理 Context。

session/llm.ts 接着按当前配置向 Provider（模型提供商）发请求，再由 session/processor.ts 里的 Processor（处理器）消费响应流，把 Message Parts、Tool 状态、Cost 和 Snapshot Patch 写下来。

```
Session Prompt Loop
  ↓ 构造 System / Messages / Tools
LLM Stream
  ↓
Processor：Text / Tool / Step Finish / Error
  ↓
continue ──→ 下一次模型请求
compact ────→ 压缩后继续
stop ──────────→ Session Idle / Error
```

第 1 站：有 Tool Calls 时不能只看 Provider Finish

源码：查看 Prompt 主循环

```
while (true) {
  const hasToolCalls = lastAssistantMsg?.parts.some(...)
  if (lastAssistant?.finish && !hasToolCalls) break
  // 继续处理工具结果或下一次模型请求
}
```

调用者：Server/CLI 的 session.prompt 服务。

输入：Session ID、User Parts、Agent、Model 和 AbortSignal。

状态变化：追加 User/Assistant Messages，驱动多次 LLM Step。

返回：最终 Assistant Message/Parts 或 Error。

下一站：Session Status 变 Idle，表面输出最终投影。

即使 Provider 给出的 FinishReason（结束原因）不是 tool-calls，只要 Assistant Parts 里已经写入 Tool Call，循环仍要把这次调用处理完，因为落盘的 Part 比一个结束字段更能说明模型实际吐出了什么。别只信 FinishReason。

为什么主循环没有集中在一个函数里

Prompt 层负责从启动到结算这一整段 Session，LLM 层把请求改写成各家供应商接受的格式，Processor 则逐个消费流事件，再把结果收进持久状态。三层拆开以后，同一个 Processor 可以处理不同 Provider 发出的统一事件，Server 也只需调用 Session 服务，但你得沿着三层代码连起来看，才能找到完整的 Loop。要判断谁在控制下一轮，就看 Prompt 收到 Processor 的信号后做了什么，别只搜索名为 loop 的函数。

第 2 站：Processor 把流归约成三个控制信号

源码：查看 Processor 结算

```
const stream = llm.stream(streamInput)

if (ctx.needsCompaction) return 'compact'
if (ctx.blocked || ctx.assistantMessage.error) return 'stop'
return 'continue'
```

调用者：Prompt Loop 每个 Step。

输入：LLM Stream、Assistant Message、Tools、Permission 与 Hooks。

状态变化：逐事件更新 Message Parts、Tokens、Cost、Error、Blocked 与 Compaction 标志。

返回：continue、compact 或 stop。

下一站：Prompt Loop 决定再采样、压缩还是结算。

Processor 返回的这三个值只负责控制 Loop：工具交回结果时可能返回 continue，遇到错误或 Hook Block 时可能返回 stop，Context 放不下时才返回 compact。它们只决定循环怎么走，不评价用户的目标有没有完成。

第 3 站：Tool 与 Step 事件写入不同事实

源码：查看 Processor Event 分支

```
case 'tool-call':
  yield* ensureToolCall(value)
  break

case 'step-finish':
  ctx.assistantMessage.finish = value.reason
  // 写 Token、Cost、Snapshot Patch，并检查 Compaction
```

调用者：Processor 异步消费 AI SDK Stream。

输入：Text Delta、Reasoning、Tool Call/Result、Step Start/Finish 和 Error。

状态变化：Tool Call 建 Running Part 并防重复；Result 结算 Part；Step Finish 结算本次模型步骤。

返回：持久化后的 Message/Part Events。

下一站：UI 订阅、Session Store 与下一轮 Context。

Tool Part 进入 Success，只说明这个工具已经交回结果，Step 出现 Finish 则说明本次模型流已经结束，直到 Session 转为 Idle，才能确定主循环眼下不再推进。如果界面给三者都画上绿色勾，调试者很容易把它们当成同一层的成功。结束也分层级。

LLM 层负责 Provider 语法，不拥有 Session 控制

LLM 层负责把 Agent Prompt、Provider Model、System、Messages 和 Tools 交给模型 SDK，也可以在请求里注入 OpenTelemetry。它无权判断用户目标是否完成，因为 Processor 负责解释流事件，Prompt Loop 则根据解释结果继续推动 Session。

回到运费任务

Prompt Loop 先把用户目标写进 Session，LLM 层再把当前有效的历史和工具表交给 Provider。Processor 收到 read Tool Call 后会创建 Running Part，等工具交回结果便结算这个 Part，并向 Prompt Loop 返回 continue。只有 edit 和测试工具都运行完毕，而且最后一条 Assistant 不再带有 Tool Call，Prompt Loop 才会真正退出。

练习：区分三个「结束」

工具 Part 已 Success、模型 Step 已 Finish、Session 仍 Busy。为什么这不是矛盾？

下一篇：Tools、Permission、Question 与 Patch。

Tools、Permission、Question 与 Patch 的边界

返回 OpenCode 课程地图

OpenCode 会先收齐内建工具和 Plugin 提供的工具，也会把通过 MCP 接入的工具放到一起，然后根据模型能力和功能开关，筛出这一轮真正能进入 Tool Registry（工具注册表）的部分。

模型发出 Tool Call（工具调用）以后，Permission.ask() 才会逐条计算权限规则，如果还得让用户决定，客户端就会收到 Question。

Patch（补丁）和 Snapshot（快照）可以撤销工作区里的改动，却碰不到已经发生的外部副作用：文件能退回原样，外部世界不会跟着复原。

```
Tool Registry → 模型可见 Schemas
                ↓
          Tool Invocation
                ↓
Permission Rules：allow / deny / ask
                ↓
Question Event → 用户 once / always / reject
                ↓
Tool Body → Patch / Snapshot / Result
```

第 1 站：工具目录在交给模型前仍会过滤

源码：查看 Tool Registry 投影

```
const filtered = (yield* all()).filter((tool) => {
  if (tool.id === ApplyPatchTool.id) return usePatch
  // 其他模型与功能过滤
})
```

调用者：Session Prompt 构造本轮 Tools。

输入：所有注册工具、Model、Agent 与 Feature Flags。

状态变化：不删除源定义，只形成当前模型可见集合。

返回：AI SDK Tool Map。

下一站：模型生成 Tool Call，Processor 建 Running Part。

模型看不见某个工具，误调用的机会确实会少一些，不过这里还没有判断它到底有没有权限。Plugin 代码仍可能绕过模型，直接在外部产生副作用。反过来，即使模型看见了某个工具，也得等执行阶段算完规则并放行，工具才会真正运行。隐藏不等于授权。

第 2 站：Permission 使用最后匹配规则

源码：查看规则求值

```
ruleset
  .findLast((rule) =>
    Wildcard.match(permission, rule.permission) &&
    Wildcard.match(pattern, rule.pattern),
  ) ?? { action: 'ask' }
```

调用者：Tool Body 在执行具体路径/命令前调用 Permission。

输入：Permission 名称、规范化 Pattern 与有序 Ruleset。

状态变化：无匹配时回退 Ask；最后一条匹配规则覆盖之前规则。

返回：Allow、Deny 或 Ask。

下一站：立即执行/拒绝，或发布 Question Event。

你在审计权限配置时，不能只搜索其中有没有 deny，因为后面那条范围更宽的 Allow 仍可能盖过前面的具体 Deny。要看最终会放行还是拒绝，必须按照配置真正合并后的顺序，把所有匹配规则走一遍。顺序就是语义。

「最后匹配」带来的可组合性与风险

「最后匹配」让你先写全局默认，再叠加项目例外，最后还能让管理员补一条收口规则，因此同一套机制可以表达很灵活的权限策略。代价也很直接：文件怎么排列，会影响系统是否放行。一旦合并配置时改了数组顺序，最终动作就可能随之改变，所以测试要跑过合并后真正生效的完整 Ruleset，不能只确认某条规则还在。顺序一变，权限就变。

第 3 站：Ask 会等待 Deferred，不会隐式允许

源码：查看 Permission 请求生命周期

```
if (rule.action === 'deny') {
  return yield* new PermissionV1.DeniedError(...)
}

yield* events.publish(Event.Asked, info)
return yield* Deferred.await(deferred)
```

调用者：Read、Edit、Bash、External Directory 等工具。

输入：Session、Call ID、Permission、Patterns 与元数据。

状态变化：注册待回答 Deferred 并发布事件；always 才把选中 Pattern 加入当前实例批准集合。

返回：Once/Always 后继续，Reject/关闭时失败。

下一站：Tool Body 或 Processor Error Part。

用户选择 always 后，OpenCode 只是把对应 Pattern（匹配模式）加入当前应用实例的批准规则，并不会因此改写 OS ACL。只要某个工具绕过 ask() 直接执行，应用层 Permission 就管不到它。这两层不能混。

Question 与 Permission 不完全相同

Question Tool 也会请用户作出业务选择，但用户答了普通问题，并不等于同时批准工具产生副作用。Permission Question 必须绑住具体的 Tool Call 和 Pattern，普通答案则只会写回模型 Context，所以客户端要把两类问题的风险清楚地展示给用户。

Patch 与 Revert 能覆盖什么

文件工具先留下 Snapshot 和 Patch，Session Revert 才能据此把工作树里的改动倒回去。不过它无法收回网络请求、数据库写入、项目外删除或包发布，也不会自动停掉后台进程。只要某个操作可能越过工作树边界，你仍要限制权限，并让外部操作可以安全重试。Patch 不是事务。

回到运费任务

回到运费任务，规则可以直接放行 read，却可能要求用户确认 edit。至于 bash 里的测试命令，还得拿命令本身继续匹配 Pattern。用户选择 Once，只会批准当前 Call，选择 Always 才会把匹配模式留在当前实例里。文件写完后虽然可以靠 Patch 回退，但 Bash 如果同时启动了后台服务，Revert 不会替你关掉进程。

练习：计算最终规则

Ruleset 依次包含：edit:* → ask、edit:tests/** → deny、edit:** → allow。对 tests/shipping.test.ts 的最终动作是什么？

下一篇：Storage、Compaction 与 Revert。

Storage、有效历史、Compaction 与 Revert

返回 OpenCode 课程地图

OpenCode 会把 Session、Message 和 Part 的完整记录存进数据库，但 Prompt Loop（提示词循环）不会原样读取它们，而是先让 Compaction（上下文压缩）筛选并重排历史，再取其中仍然有效的部分。

Provider（模型提供商）拿到的内容还要再转换一次，已经成了适配具体 API 的 Model Messages。Revert（还原）走的是另一条路，它既要改工作树，也要清理 Session 里位于还原点之后的消息。

工作树能倒回去，是因为文件工具事先留下了 Snapshot（快照）和 Patch（补丁）。Session 这边则要删掉消息后缀，让下一轮从新的尾部继续。你读这一篇时要一直分清这两条线。

```
数据库 Session / Message / Part
        ↓ filterCompacted
摘要 + 保留尾部（可能重排）
        ↓ toModelMessages
Provider Context

Revert：Snapshot/Patches + Session Cleanup
```

第 1 站：有效历史可能不是时间顺序

源码：查看压缩历史投影

```
return [
  ...result.slice(compactionIndex, summaryIndex + 1),
  ...result.slice(tailIndex, compactionIndex),
]
```

调用者：Session Prompt 读取模型有效历史。

输入：按持久顺序读取的 Messages/Parts。

状态变化：选择最新 Compaction Summary 与之后保留的 Tail，并按语义重排。

返回：供 Context 转换的有效历史。

下一站：Message Converter 生成 Provider Model Messages。

Compaction 按语义重排数组以后，最后一项就未必对应 Session 的最新状态。你要找「最新消息」，应当使用源码规定的时间或 ID 规则，不能顺手拿走投影数组的尾元素。尾部不等于最新。

三种历史视图分别服务什么

数据库留下完整记录，供你审计和恢复。Compaction 从这些记录里挑出模型还能看到的历史，用来控制窗口，Provider Messages 再把有效历史改成具体 API 接受的格式。三层数据各自做一件事，顺序、字段和粒度自然可能不同。调试模型为什么这样回答时，要保存真正送给 Provider 的第三层输入。如果要解释 Session 怎样一步步变成现在这样，就得回到数据库里的第一层记录，因为只看 UI 截图，通常哪一个问题都答不完整。三种视图不能混。

第 2 站：保留尾部按 Token 预算倒推

源码：查看近期预算选择

```
const budget = preserveRecentBudget({ cfg: input.cfg, model: input.model })
if (total + size <= budget) {
  keep = { start: turn.start, id: turn.id }
```

```
}
```

调用者：Compaction Service。

输入：Model Context 限制、Config、Turn 边界与 Token 估算。

状态变化：从最近 Turn 向前累计，确定摘要前缀与逐条保留尾部。

返回：Compaction 切分点。

下一站：模型摘要旧前缀，写 Summary Message/Part。

OpenCode 按 Turn 切分历史，能让同一轮里的 Tool Call 和 Result 留在一起，避免模型只看见调用却找不到结果。不过 Token 只是本地估算，未必和 Provider 最终采用的口径一致，所以预算里必须留出余量。

第 3 站：Revert 同时恢复文件和 Session 控制状态

源码：查看 Session Revert

```
rev.snapshot = session.revert?.snapshot ?? (yield* snap.track())
yield* snap.revert(patches)
yield* sessions.setRevert({
  sessionID: input.sessionID,
  revert: rev,
  summary: ...,
})
```

调用者：用户从 UI/CLI 请求 Revert 到某个 Message/Part。

输入：Session ID、Revert Point、Snapshot 与后续 Patches。

状态变化：反向应用文件修改，记录 Revert 元数据与 Diff，随后清理消息后缀。

返回：恢复后的 Session 状态。

下一站：UI 重新同步；后续 Prompt 从新尾部继续。

执行 Revert 前，要先等 Session 结束 Busy 状态，否则你一边恢复文件，仍在运行的模型流可能一边继续写入。它也算不上通用事务，因为 Git Ignore 文件、项目外路径和网络副作用都可能原封不动地留在那里。恢复有明确边界。

如何核对压缩没有把任务带偏

核对压缩有没有带偏任务时，你要保存压缩前的完整 Messages/Parts，也要留下摘要输入、Summary、保留的 Tail 和压缩后的 Model Messages，然后用答案确定的问题逐项追问：目标是什么，哪些步骤还没做，文件在哪，最近一次 Tool Error 又是什么。请求不再溢出，只能说明窗口暂时够用，不能证明任务语义还完整。

回到运费任务

压缩这段长会话时，Summary 要记住「金额 100 的目标测试仍未运行」，近期 Tail 则要留下刚才编辑文件的结果。如果摘要错写成「测试已通过」，下一轮就可能没做真实验证便提前结束。等你 Revert 到编辑前的消息时，既要用文件 Patch 倒回工作树，也要删掉 Session 后缀，否则模型读到的历史还说「尚未编辑」，文件却早已变了。

练习：发现历史与工作树分裂

Revert 后 Session 已删掉编辑结果，但 git diff 仍显示修改。应把哪个事实当作当前环境真值？

下一篇：Agents、Skills、Plugins、MCP 与 LSP。

Agents、Skills、Plugins、MCP 与 LSP 扩展哪一层

返回 OpenCode 课程地图

OpenCode 有好几条路可以给 Agent 增加能力，但每条路接入系统的位置都不一样。Agent 定义会选定模型、Prompt 和 Permission（权限），Task Tool 则负责创建子 Session，让父 Agent 能把一段工作交出去。

Skill（技能）经过权限过滤后才把指令资源交给 Agent，Plugin 却能直接向进程注册 Tools、Auth、Provider 和 Hooks。

再往系统外面看，MCP 暴露远端的 Tools、Prompts 与 Resources，LSP（语言服务器协议）只管回答代码语义问题。它们不能混着理解。

子 Agent：继承拒绝边界，不复制父权限全集

第 1 站：子 Session 权限从父端提取关键限制

源码：查看子 Agent 权限派生

```
return [
  ...input.parentSessionPermission.filter(
    (rule) =>
      rule.permission === 'external_directory' ||
      rule.action === 'deny',
  ),
]
```

调用者：Task Tool 创建子 Session。

输入：父 Session Permission 与目标 Agent 配置。

状态变化：保留外部目录与拒绝规则，并为递归 Task/Todo 加安全默认值。

返回：Child Session Ruleset。

下一站：Session Create 与子 Prompt Loop。

父 Session 已经放行的能力不会自动全部交给子 Agent，因为子 Agent 换了 Persona（角色设定）以后，仍得遵守父 Session 设下的禁止边界。Deny 必须跟过去。

第 2 站：Task Tool 在创建前检查深度和 Agent

源码：查看 Task Tool

```
if (depth >= (cfg.subagent_depth ?? 1)) {
  return yield* Effect.fail(...)
}

const childPermission = deriveSubagentSessionPermission(...)
```

调用者：父模型调用 Task Tool。

输入：Agent 名、Prompt、父 Session/Message 与当前深度。

状态变化：验证功能开关、深度、Permission 和 Agent 存在性；创建或复用子 Session。

返回：子任务结果投影与 Child Session ID。

下一站：父 Prompt Loop 把结果作为 Tool Result 继续。

子 Session 会保存自己的 Messages，也会独立触发 Compaction 并产生副作用，父端最后拿到的只是这段运行投回来的结果。调试时一定要留下 Child Session ID，你才能找到真正执行过工作的那条历史。

子 Agent 为什么不能只是一次嵌套函数调用

子任务一旦跑进多轮模型调用，就可能继续执行工具、触发压缩，也可能中途单独取消或失败，因此普通的嵌套函数调用根本装不下这些状态。独立 Session 会替子任务留住完整历史，也给它划出可以恢复的边界，父 Session 只收最终投回来的 Tool Result，不用把内部对话全塞进自己的 Context。这样做也有代价，跨 Session 的权限和成本要单独处理，Trace 也得明确关联起来。

Skill：发现后还要按 Agent Permission 过滤

源码：查看可用 Skill 列表

```
return list.filter((skill) =>
  Permission.evaluate('skill', skill.name, agent.permission).action !== 'deny',
)
```

调用者：Prompt/Skill Tool 为当前 Agent 列出资源。

输入：发现的 Skills 与 Agent Permission。

状态变化：过滤明确拒绝的 Skill。

返回：该 Agent 可见的 Skill Catalog。

下一站：模型选择读取 Skill 正文。

Plugin 是高信任进程代码

Plugin 可以注册 Tool、Provider、Auth 和 Hook，还直接拿着 OpenCode 进程的权限运行，风险自然比普通 Prompt 文件高得多。启用之前，你得逐项检查它的签名、来源、版本和安装脚本，这道检查不能省。如果 Hook 还会改写模型输入或 Tool Result，就要把改写前后的值都留下，免得出问题时找不到证据。

MCP：连接存在不等于服务提供每种能力

源码：查看 MCP Catalog 能力检查

```
if (!client.getServerCapabilities()?.resources) {
  return Promise.resolve([])
}

return paginate((cursor) => client.listResources(...))
```

调用者：MCP Client 连接成功后的目录发现。

输入：Server Capabilities 与分页 Cursor。

状态变化：只请求声明支持的 Tools/Prompts/Resources/Templates。

返回：可转成 OpenCode Registry 项的目录。

下一站：Tool/Skill/Resource 表按当前 Agent 过滤。

LSP：按文件和项目根选择语言服务器

LSP 可以查询 Diagnostics、Definition、References 和 Symbols，但它得先找到 Binary，正确识别 Root，还要确认文件语言匹配，查询才真的跑得起来。仓库里有 package.json，不代表 TypeScript Server 已经启动。

回到运费任务

父 Agent 可以把「检查相关测试覆盖」交给子 Agent，不过父端用 Deny 禁止编辑测试后，这条限制必须在子 Session 里继续生效。任务跑起来以后，Skill 可以告诉子 Agent 该用哪条测试命令，LSP 可以帮它找到 shippingFee 的引用，MCP 则能提供外部资源。Plugin 直接进入进程，最该仔细审计。它们改动的层次不同，不能全叫「插件」。

练习：一项能力为什么目录可见却不能使用

如果 Skill 已被发现但当前 Agent 看不到，MCP 已连接却没有 Resource，或者 LSP 配置存在却查不到 Definition，分别应该从哪一层开始检查？

下一篇：Server、Protocol 与多产品表面。

Server、Protocol 如何同时服务 TUI、Desktop、Web 与 ACP

返回 OpenCode 课程地图

OpenCode 的 TUI（终端用户界面）、Web、Desktop、SDK 和 ACP（Agent 客户端协议）各有自己的使用场景，却都连到同一套 HTTP/Event 服务核心。

Protocol（协议）Package 先规定 Endpoint 的 Schema、错误和 Middleware 放在哪里，Server 再接入 Session Location、Authorization 与业务 Handler，因此这些产品表面都要经过协议访问 Session，谁也不会直接拿到内部对象。

```
TUI / Run / Web / Desktop / SDK / ACP
            ↓ HTTP / Events / Adapter
      Protocol Schema
            ↓
   Server Handler + Middleware
            ↓
   Project / Session Services
```

第 1 站：Protocol 定义 API Group 和中间件位置

源码：查看 Protocol API

```
return HttpApi.make('opencode')
  .add(makeSessionGroup(sessionLocationMiddleware))
  .add(eventGroup)
  .middleware(Authorization)
```

调用者：Server API 组装与 SDK/OpenAPI 生成。

输入：Session Location Middleware 定义。

状态变化：组合 Endpoint Groups、Schemas 和 Authorization 位置。

返回：与具体业务实现分离的 API Contract。

下一站：Server 为每个 Endpoint 注册 Handler。

Schema 只能检查请求和响应长什么样，即使 Session 刚刚通过 Location Lookup，它随后仍可能被移走、删掉或转入 Busy 状态，所以 Handler 还得捕捉这些变化，并返回对应的业务错误。

第 2 站：Session Prompt 与 Events 是不同返回形态

源码：查看 Session Handlers

```
  'session.prompt',
  (ctx) => Effect.map(
    session.prompt(...),
    (data) => ({ data }),
  )
```

源码：查看 History 与 Event Handler

```
  'session.history'
  'session.events'
  session.events({ sessionID, after })
```

调用者：HTTP/SDK Client。

输入：Directory/Session Location、Prompt 或 Event Cursor。

状态变化：Prompt 驱动业务；Event/History 只读取持久记录与流。

返回：同步响应、历史页或可重连事件流。

下一站：客户端更新自己的 UI State。

事件连接随时可能断开，因此客户端得保存 Cursor（游标），还要能处理重复事件和已经过期的 History。先订阅再发送 Prompt，确实可以缩小竞态窗口，但只要连接中断过，客户端仍得重新取得一份基线，才能补回这段时间发生的事实。

多客户端共享的是服务事实，不是界面状态

Server 负责保存 Session 和事件，TUI、Web 与 Desktop 则各自记住用户选中的面板、滚动位置和还没提交的输入。界面状态留在本地。协议不能拿某个 UI 展示出来的内容充当 Session 真值，也不能逼着每个客户端理解服务内部对象，真正能在它们之间共享的只有稳定的 Endpoint Schema，以及可以重放的 Event Cursor。

Run 与 TUI 也通过统一事件面

源码：查看 Run 事件循环

```
for await (const event of events.stream) {
  if (event.type === 'session.created' && event.properties.info.parentID) {
    // 跟踪子 Session
  }
  if (event.properties.status.type === 'idle') break
}
```

调用者：非交互 Run 或 Mini 交互模式。

输入：Server Events 与目标 Session ID。

状态变化：跟踪父子 Session，格式化消息/工具输出；目标 Session Idle 时结束。

返回：终端文本/JSON 与进程状态。

下一站：Shell 自动化或用户查看结果。

Idle 只说明控制流程已经停下来，Session 眼下没有工作可做。至于结果对不对，它不管。

Web/Desktop 与 Server State 分开

App 会先根据 Server Identity 挂载 SDK/Sync Provider，然后在 Directory Scope 里管理 Project 和 Session 数据。Desktop 复用这套 App 时，还会接上本地或 WSL 连接，并补充原生能力。切换 Server 时一定要重建数据 Provider，因为如果只改 URL，旧服务留下的数据仍会挂在新连接下面。

ACP 只是另一层协议适配

源码：查看 ACP Agent 委托

```
newSession(params) {
  return run(this.service.newSession(params))
}

prompt(params) {
```

```
    return run(this.service.prompt(params))
  }
```

ACP 把编辑器送来的 Session、Prompt 和 Permission 转给 OpenCode Service，再由适配层统一整理错误，但它对外给出的内容只截取了内部 Event Log 的一部分。

回到运费任务

Web 提交 Prompt 后即使断了线，Session 仍会留在 Server 上继续运行，因此 Desktop 重新连上时，应该先取得 History 或 Snapshot（快照），再从 Cursor 指向的位置续订后面的事件。如果客户端只等新事件，断线期间做过的编辑和测试就不会出现在界面上。可要是它从头应用所有事件，工具结果又可能显示两遍。

练习：设计一次安全重连

请给出重连的最小顺序。

下一篇：Share、Telemetry 与 Eval 边界。

Share、Telemetry、上游测试与独立 Eval 的边界

返回 OpenCode 课程地图

Share 会把 Session 的可见内容复制到外部服务，Telemetry（遥测）记下运行时发生了什么，上游测试守住特定的代码契约，独立 Eval 则按固定任务检查 Artifact（产物）。它们可以沿用同一个 Session 身份，但各自回答的问题不能混在一起。

第 1 站：分享策略区分禁用、手动和自动

源码：查看 Share 入口

```
if (conf.share === 'disabled') {
  throw new Error('Sharing is disabled in configuration')
}

if (!(flags.autoShare || conf.share === 'auto')) return result
```

调用者：Session 创建或用户显式 Share。

输入：Share Config、Feature Flags 与 Session ID。

状态变化：禁用时拒绝；自动只对符合条件的根 Session 异步创建分享。

返回：本地 Session 与可选 Share Metadata。

下一站：ShareNext 做首次全量同步并监听后续事件。

自动分享会把数据送出本地环境，因此配置必须把开关写清楚，开启之前还得检查子 Session 里有没有敏感的工具输出。

分享不是普通 UI 功能

Share Worker 持续复制 Message、Part 和 Diff，而这些内容可能带上源码、命令输出、路径和模型推理片段，因此你要像检查 Provider 调用那样，检查它怎样给数据分类、怎样脱敏、怎样删除，又留下了哪些审计记录。公开链接能打开，只能说明远端已经有一份可见副本，至于内容有没有同步完整、删除流程能不能走完，还得分别核对。

第 2 站：Share 是会话投影的持续复制

源码：查看 Share 事件监听

```
yield* watch(MessageV2.Event.Updated, ...)
yield* watch(MessageV2.Event.PartUpdated, ...)
yield* watch(Session.Event.Diff, ...)
```

调用者：活动 Share Worker。

输入：Session、Message、Part、Diff 与 Delete Events。

状态变化：把本地投影增量同步到远端；失败可能只记录告警。

返回：外部可查看副本。

下一站：取消分享时调用远端 Delete 并清理本地 Share 记录。

分享链接已经生成，不代表远端副本追上了本地内容。你删除本地 Session 之后，也别默认外部副本会跟着消失，最终要看 API 实际返回了什么。

第 3 站：OpenTelemetry 是条件性注入

源码：查看 LLM Telemetry

```
const tracer = cfg.experimental?.openTelemetry
  ? ...
  : undefined

span.setAttribute('session.id', input.sessionID)
```

调用者：Session LLM Stream。

输入：实验配置、Session/Model/Provider 与请求时序。

状态变化：启用时创建 Span 和模型 SDK Telemetry Metadata。

返回：观察数据，不改变 Processor 控制结果。

下一站：Exporter、Trace Backend 或 Eval Collector。

源码里的 Telemetry 没有 Rubric、Score 或 Release Decision 字段，因此 Trace 里没有 Error Span，也证明不了这次修改是对的。

独立 Eval 应保存什么

以代码修复为例：

1. 固定 Server/Project/Directory、Config Provenance、Agent、Provider/Model 和 Permission Ruleset。
2. 保存 Session/Message/Part、Tool/Question Events、Child Session IDs 和实际文件 Diff。
3. 记录 Compaction/Revert/Share/Telemetry 是否发生，但不要让它们覆盖原始 Artifact。
4. 在 OpenCode 外部重新运行测试与静态检查。
5. Scorer 根据测试、Diff 范围和任务约束给 Score 与失败原因。

如果要把这些结果变成训练信号，就用 Reward Adapter（把原始信号转换成训练奖励的版本化规则）接进来，选模型和决定是否发布时仍要跑隔离任务。Share View、Session Idle、Tool Success、用户点下 Always Allow，甚至上游单元测试全部通过，都不能单独顶替这一步。

回到运费任务

你可以在 Share 页面回看模型说了什么、工具做了什么，也可以从 Telemetry 里查延迟和成本，上游测试则用来守住 Processor 或 Permission 的代码契约。不过，只有外部 Scorer（评分器）在冻结的工作区里跑过目标测试并检查 Diff，才能判断这次修复是否完成任务。这四类证据可以拼在一起看，却不能拿其中一类冒充另一类。

练习：同步失败时结果还能否评分

Share Worker 最后一批同步失败，但本地 Session、Diff 和独立测试 Artifact 都完整。Eval 是否必须判为任务失败？

读完这一站，可以回到 OpenCode 课程地图，也可以进入 横向比较，拿同一组问题去对照六套 Harness。

六套 Harness 怎样形成一次模型输入

返回课程总目录 · 下一篇：Agent Loop、工具与执行

模型不会直接读取整个仓库，因为每次采样前，Harness 都得先取出用户目标、系统指令、项目规则、历史消息、工具定义和运行状态，再按 Context 容量把它们收进一个有界请求。这篇不比谁的「提示词更强」，我们要看六套实现怎样取得输入、由谁定下最终顺序，以及哪些状态压根没有交给模型。

先用运费任务建立问题

用户说：「修复订单金额为 100 元时仍收取运费的问题，并运行测试确认。」第一轮里，模型至少要看到用户目标和可用工具，但没必要立刻把全部源码读完。读取工具拿回 shipping.ts 后，下一轮才会看到真实的文件内容。如果项目规则写着「禁止修改测试」，就得在模型决定编辑前把这条约束交给它，否则就要让权限层直接拦住违规操作。

所以，每轮都要根据新的工具结果和运行状态重新算出「模型输入」，它不是启动时生成一次就不再改变的字符串：

稳定前缀：系统指令、项目规则、工具定义
任务输入：用户目标、补充约束
运行历史：模型消息、工具调用、工具结果
动态状态：工作目录、活动 Agent、剩余预算、压缩摘要

各套 Harness 给这些内容起的名字不同，保存方式也不同，但最后都得把它们转成 Provider（模型提供商）能读懂的消息和 Tool Schema。名字不是重点。

共同调用链

配置与项目身份
→ 选择模型、Agent/Preset 和工具集合
→ 加载项目指令、Skill 或 Context 文件
→ 选择 Session 的有效历史
→ 应用压缩、Hook 或 Context Transform
→ 转换为 Provider 消息和 Tool Schema
→ 发起本轮模型请求

这条链只要有一步走错，你就会遇到「模型为什么没看见」：程序可能从另一个目录读了配置，投影工具时把某个工具过滤掉，把历史写到了非活动分支，压缩摘要时遗漏了事实，或者在转成 Provider 消息时丢了某类内容。

六条主线的真实切入点

课程	输入装配的主要入口	值得学习的差异	继续阅读
DeepSeek Harness	Preset、Bundle、Prompt 与 Context/Cache 包	能力通过多包装配；Prompt、KV Cache 和 Code Mode 共同影响请求	Prompt、Context 与 Cache
Codex	Config、项目指令、Thread/Turn 与请求构造	强类型事件和项目指令进入同一 Thread；稳定前缀还影响缓存	配置、项目指令与模型输入
Gemini CLI	Config、Prompt、Turn 与 Core	当前 Turn/Scheduler 主链要与 Legacy Agent Session 分开	配置、Prompt 与 Context
Claude	Agent SDK Options、Transport 参数和公开消息类型	SDK 能证明应用侧输入与控制协议，不能展开闭源 CLI 内部装配	Python 入口、Transport 与控制连接
pi	Resource Loader、System Prompt、活动工具和 Session Context	资源先形成一致快照；完整 Session Tree 与模型投影明确分开	Coding Agent、Prompt 与 Extensions
OpenCode	Project/Config、Session Prompt、有效历史与 Provider 转换	服务端 Directory 决定项目身份；数据库历史、有效历史和 Provider 消息是三种视图	Runtime、Project、Config 与 Provider

这张表不能当成接口对照表。例如 Claude 的公开 SDK 只能让我们看到某些边界，而其他五套开源运行时能暴露的内部流程并不相同。即使它们都有 messages 字段，你也不能因此认为自己能核对同样深度的内部行为。证据到哪里，就说到哪里。

配置合并为什么是输入问题

配置不只选模型，它还会定下项目指令、Tool Allowlist、Permission、Hook、Extension 和压缩阈值。想知道后写的值会不会盖掉前值，你得回到各项目的真实合并逻辑：标量可能直接覆盖，数组可能往后拼接，管理员配置还可能在项目配置合并完以后再收紧一遍。

新人排查这类问题时，最好先记下 Provenance，以后看到任何一个最终值，都能顺着记录追回它从哪里来：

最终字段	最终值	来源	为什么胜出
Model	具体 Provider/Model	用户或 Agent 配置	显式选择覆盖默认
Instructions	多段项目规则	全局 + 仓库	按各项目顺序合并

最终字段	最终值	来源	为什么胜出
Tools	本轮可见集合	Preset/Agent/Feature	注册表经过投影
Permission	有效规则集	用户 + 项目 + 管理策略	按真实优先级求值

没有 Provenance，即使「最终 JSON 看起来正确」，你也解释不了为什么换个目录、客户端或 Session，程序就组出了不同的请求。这一层不能省。

Tool Schema 和自然语言指南必须一致

模型通常会同时看到两种工具信息：结构化的 Tool Schema 列出名称、参数和描述，System Prompt 或项目指南则告诉模型什么时候该用它。如果两者取自不同的资源快照，模型就可能根本不知道某个工具其实可以调用，Prompt 也可能还在推荐已经删除的工具。两份快照必须对上。

DeepSeek Harness 通过 Preset/Bundle 把内容装起来，pi 让 Resource Loader 根据活动工具生成 Prompt，OpenCode 则先把 Registry 投影出来，再构造 Session Prompt。Codex 和 Gemini CLI 也会读取策略与运行配置，然后定下这一轮真正可见的工具。所以比较时，要检查「本轮发给模型的最终工具表」，不能只数源码里曾经注册过哪些工具。

Session 中有记录，不代表这一轮可见

至少有四种常见过滤：

1. 当前只选择 Session Tree 的一个活动分支；
2. Compaction Summary 替换了旧前缀；
3. UI 或审计 Entry 不属于模型消息；
4. Provider Adapter 对角色、Tool Result 或附件做了再次转换。

pi 把完整 Entry Tree、当前活动路径和 Context Entries 分开处理，OpenCode 也会分别保留数据库记录、压缩后的有效历史和 Provider Messages。Codex 的 Rollout/Compaction、Gemini CLI 的历史压缩与 DeepSeek Harness 的 Session/Context 同样把这几层分开。所以调试时，你得抓到真正交给 Provider 的请求并安全脱敏。只打开 Session 文件还不够。

用同一个故障排查法

当模型说「我看不到测试文件」时，按下面顺序排查：

1. 项目身份：当前 Server/CWD/Project 是否真指向目标仓库；
2. 能力装配：读取工具是否注册并进入本轮可见集合；
3. Prompt 与 Schema：文字说明和 Tool Schema 是否来自同一快照；
4. 历史投影：读取结果是否写入正确 Session、Call ID 和活动分支；
5. 压缩变换：摘要或 Hook 是否丢掉路径和工具结果；
6. Provider 转换：最终消息是否仍保留 Tool Result；
7. 证据边界：公开来源是否真的能观察到该阶段。

按这个顺序查，你会先核对能确定的系统事实，最后才需要怀疑模型是不是「忘了」。很多问题看着像模型在犯错，其实是 Harness 把错误的观察交给了它。所以要先核对输入。

设计取舍：稳定、完整和有界不可能同时最大化

如果一味把输入塞得更全，Token 用量和延迟都会上升，可压得太狠又容易漏掉还没做完的事。每轮动态插入的内容越多，稳定前缀就越容易变，缓存也越难命中。因此，Harness 得明确定下哪些信息要精确保留，哪些可以压成摘要，哪些等真正用到时再通过工具读取。

放到运费任务里，Harness 必须精确留下用户目标、「禁止修改测试」这条约束，以及最近一次测试失败。但整份仓库目录可以用到时再读，很久以前那些无关的探索也可以压成摘要。按内容对任务的作用来分类，比简单按消息时间截断更可靠。

练习：为一次模型请求画输入清单

任选两条课程，假设 Agent 已经读取 shipping.ts 但还没有编辑，然后分别找出项目身份、系统/项目指令、用户目标、工具定义、历史消息、读取结果和动态状态分别由哪个对象保存、在哪个函数附近流转。如果某项无法从公开源码中核对，也要明确写出你能看到哪里、又从哪里开始无法确认。

下一篇：六套 Harness 怎样完成模型—工具闭环

六套 Harness 怎样完成模型—工具闭环

上一篇：运行时、配置与模型输入 · 返回课程总目录 · 下一篇：权限、状态与恢复

模型吐出 Tool Call 时，只是用结构化数据表达「我想调这个工具」。Harness 还得把调用解析出来，找到对应实现，检查策略后执行，记下结果，然后把新观察送回下一轮。这篇继续用运费修复任务，看六套实现在哪里接上这个闭环，也看清并发、失败和停止条件为什么塞不进一个 while。

最小闭环不是「模型调用一次工具」

模型流产生 Tool Call
→ Harness 验证名称与参数
→ 路由到具体工具
→ 权限与环境决定是否执行
→ 工具产生成功、拒绝或错误结果
→ 结果按 Call ID 写入消息与 Session
→ Harness 决定再次采样、压缩、取消或结束

运费任务至少要读文件、改代码、跑测试，其中任何一步失败，Harness 都应把错误变成下一轮能读懂的观察，不能只往终端扔一行错误。模型就算最后说了「完成」，也代替不了目标测试的实际结果。这两件事不能混。

同一问题在六套实现中的落点

课程	谁主要控制下一轮	工具路由与结果回填	代表性细节	继续阅读
DeepSeek Harness	Agent Loop	Core Loop 与 Tool/Code Mode 包	模型消息、工具调用和 Session 事件跨包组合	Agent Loop、模型与工具结果
Codex	Thread/Turn 内的 Core 任务	Tool Router 与各 Handler	工具可以并发执行，但历史提交保持调用顺序	模型响应与工具循环

课程	谁主要控制下一轮	工具路由与结果回填	代表性细节	继续阅读
Gemini CLI	Turn 与 Scheduler	Scheduler 管理 Tool Call 生命周期	Policy、Confirmation、取消和结果事件围绕调度器展开	Turn、Scheduler 与路由
Claude	闭源 CLI 控制产品内部循环；SDK 处理公开消息与控制请求	SDK 暴露权限回调、Hook、MCP 与消息流	只能证明公开 SDK 边界，不能画出产品内部 Router	消息流与生命周期
pi	agent-loop.ts 的双层循环	Agent Core 执行工具批次并追加 Tool Results	Steering 与 Follow-up 在不同时间点插入；截断 Tool Call 不执行	Agent Loop、双队列与工具批次
OpenCode	Session Prompt Loop	Processor 归约流事件与 Tool Parts	continue/compact/stop 控制 Session，不是任务评分	Session Prompt、LLM 与 Processor

表里问「谁控制下一轮」，实际上是要你去公开源码里找：哪个控制点会创建下一次模型请求。它不要求某个类必须叫 Agent。Claude 这一行特意画清了证据边界，因为 SDK 虽然暴露了异步流，我们却仍然看不到 CLI 内部的完整 Loop。

一次工具调用至少有五种结果

如果 Tool Result 只有成功和失败两种值，恢复时需要的信息就会被丢掉，因此更实用的做法是把结果细分为：

结果	副作用是否发生	下一步通常是什么
成功	已发生	把结果送回模型
策略拒绝	未发生	解释限制，换方案或请求授权
用户取消	未发生	停止或等待用户，不应自动重试
执行错误	可能未发生，也可能部分发生	依据工具语义检查环境，再决定重试
状态未知	无法确认	先观察环境，禁止盲目重放

如果工具名不存在，或者参数没通过校验，副作用通常还没发生。可 Shell 超时时，子进程可能已经启动，网络请求断开时，服务端也可能早已处理完请求。工具必须能把这些状态分别报给 Harness，Session 才能安全恢复。

并发比较要同时记录两种顺序

并发工具有「完成顺序」和「提交顺序」：

```
模型声明：读取源码 A，读取测试 B
实际完成：B 先，A 后
历史提交：仍按 A、B 对齐各自 Call ID
```

Codex 用测试核对了两件事：工具可以按并发时的先后完成，历史却仍然要确定地提交。pi 会看完整个 Tool Batch 再计算终止条件，Gemini CLI 则让 Scheduler 同时管理多个调用。其他实现就算串行执行，也得把每个 Tool Result 对回原来的 Call ID。

如果谁先从网络返回，历史就先写谁，模型可能会把 B 的结果配给 A，以后重放时 Context 也会忽前忽后。工具不必因此全部串行，只要提交历史时保住确定的顺序，让人能重建因果关系就够了。

Stop Reason 为什么不能直接结束任务

Provider 给出的 stop、toolUse、length 或 error，只在解释这一次模型响应为什么停下。pi 遇到 length 截断时，不会执行可能没传完的 Tool Call。OpenCode 就算已经收到 Finish Reason，只要 Parts 里还有 Tool Call 就会继续。DeepSeek Harness、Codex 和 Gemini CLI 也都要同时查看工具状态和队列，才知道要不要发起下一轮。

任务级结束还需要判断：

是否仍有未结算工具；

是否有 Steering、Follow-up 或用户新输入；

是否需要 Compaction 后继续；

是否被取消、预算耗尽或协议错误阻断；

是否已经产生足以回答用户的环境证据。

最后一项通常还要交给 Eval 或应用规则判断，单看模型的 Stop Reason 并不足以判定任务结果。别在这里提前收尾。

Hook、Extension 和 Processor 可以改变什么

不同项目会让扩展在不同位置插手，有的能在调用前拦住动作，有的能在调用后改写结果，还有的会变换 Context、订阅事件或委托子任务。比较这些入口时，你一定要问清它们在副作用发生前还是发生后介入。

调用前 Hook 可以阻止动作发生；

调用后 Hook 只能改变模型看见的结果，不能假装撤销副作用；

Context Transform 可以隐藏或摘要历史，但不改变环境；

Event Subscriber 观察事实，不应反向改写核心状态；

子 Agent 拥有另一条 Loop，需要保留父子身份。

如果把所有扩展都叫作 Middleware，这些安全边界就会被一个名字盖住。别混在一起。

回到运费任务：六套实现共享的事实链

具体函数名可以不同，但一次能够复核的修复仍然应该保留下面这条事实链：

```
用户目标
→ read 调用与文件结果
→ edit 调用、批准和实际 Diff
→ test 命令、工作目录、退出码与输出
→ 最终模型文本
→ 独立结果判定
```

对 DeepSeek Harness、Codex、Gemini CLI、pi 和 OpenCode，我们可以顺着公开的核心源码往里追，只是各自能追到的深度不同。Claude 这条线则只能根据公开 SDK 消息和控制契约，核对产品显露在边界上的事实。某一层找不到证据时，就明写「当前来源不可核对」，不能拿另一套 Harness 的常见做法补上空白。

一次可复核的本地实验

不调用真实模型，你也能验证 Loop 怎样运转。先写一个按脚本返回结果的 Model Stub，让它依次给出 read、edit、bash test 和最终文本，再让另一组输入在编辑后把测试跑失败。然后检查下面几件事：

1. Tool Result 与原 Call ID 对齐；
2. 失败结果进入下一轮，而不是让 Loop 崩溃；
3. 截断或非法参数不会产生副作用；
4. 并发完成顺序变化时，历史仍可重放；
5. 模型自述「完成」不会覆盖测试失败。

这个实验只能核对通用机制，无法证明任何上游项目在所有配置下都这样做。证据到此为止。

练习：沿两套源码解释同一个失败

假设测试工具返回退出码 1，请选择两条课程，指出结果在哪里被转换成 Tool Result，又会写入何种 Session/Message 对象，并找到决定再次请求模型的控制点。如果用户在这时取消，还要说明取消信号怎样与普通工具失败区分。

下一篇：权限、Sandbox、Session 与恢复不能压成一个开关

权限、Sandbox、Session 与恢复不能压成一个开关

上一篇：Agent Loop、工具与执行 · 返回课程总目录 · 下一篇：编排、协议与产品表面

当你问「这个 Agent 有权限吗」，至少要分开问五件事：模型看不看得到这个工具，策略放不放行，用户批不批准，执行环境做不做得到，以及中断后系统能不能安全接着做。这篇把六套 Harness 摆在同一条动作链上，看清界面显示 Allow 时，为什么不能证明操作系统已把动作隔离起来，更不能承诺随时撤销副作用。

五层边界

模型可见性：本轮是否提供这个 Tool Schema
↓
参数与策略：名称、参数、路径或命令是否符合规则
↓
用户决定：是否需要一次性或持续批准
↓
执行能力：宿主、容器、Sandbox 或远程环境实际允许什么
↓
恢复语义：中断后能否确认副作用并安全继续

前四层一起决定动作能不能发生，第五层要处理的是另一件事：动作做到一半，副作用还不知道有没有发生，系统该怎样诚实地收拾现场。少任何一层，出错的方式都不一样。

六条课程各自暴露了什么

课程	应用层判断	执行或隔离边界	状态与恢复入口	深入阅读
DeepSeek Harness	工具、安全策略与 Code Mode 配置	普通工具与代码执行路径需要分别说明真实宿主	Session 与 Compaction	工具、权限与 Code Mode
Codex	Exec Policy、审批和具体工具策略	OS Sandbox/执行环境与审批分开	Rollout、Thread Store、压缩与恢复	执行策略、审批与 Sandbox

课程	应用层判断	执行或隔离边界	状态与恢复入口	深入阅读
Gemini CLI	Policy、Confirmation 与 Safety	Sandbox 是独立执行边界	Session、历史与压缩	Confirmation、Policy、Safety 与 Sandbox
Claude	SDK Options、can_use_tool 与 Hooks 的公开契约	实际 CLI/产品执行边界不能由公开 SDK 全部证明	Session/Resume/Store 的公开接口	动态权限决策
pi	工具 Hook、项目 Trust 与交互表面	默认继承宿主权限；Sandbox Extension 可能回退本地	Session Tree、JSONL 与 Resume	CLI、TUI、权限与外部隔离
OpenCode	Tool Registry、Permission Rules 与 Question	Permission 是应用规则，不自动构成 OS Sandbox	Storage、Compaction、Patch/Revert	Tools、Permission、Question 与 Patch

看到「项目提供 Sandbox 选项」时，先别认定所有工具、Extension、Plugin 和外部进程都会经过它。你得顺着每个具体工具的执行器往下追，再查它失败时会不会回退，才能下这个结论。先顺着路径查到底。

运费任务里的三个动作风险不同

动作	常见策略	仍需注意的环境事实
读取 shipping.ts	工作区内可自动允许	符号链接、外部目录和敏感文件
编辑比较符	展示 Diff 或请求批准	实际写入范围、并发修改、编码与回退
运行测试	按命令和工作目录判断	子进程、网络、环境变量、后台任务和超时

系统不必对每个动作都弹出同样的确认框，因为它可以按动作会碰哪些资源、影响多大范围、能不能撤销来分级。但动作就算已经获批，也应放进一个只拥有必要能力的环境里运行。批准不等于放开限制。

Allow、Ask、Deny 只是产品策略

应用层的 Permission 通常先读取工具名和已经规范化的 Pattern，然后回答 Allow、Ask 或 Deny。OpenCode 会按顺序检查规则，以最后命中的一条为准，Gemini CLI 让 Policy 和 Confirmation 跟着工具从提出调用走到结算，Codex 则让 Exec Policy、审批和 Sandbox 各管一层。Claude SDK 虽然公开了 can_use_tool 回调，我们仍然不能用它证明闭源产品内部的所有决策。

用户选择 Always 时，通常只是让当前应用规则多放行一些动作，它不会跟着改写操作系统 ACL。反过来也一样：OS 能执行某个动作，产品不一定就该放行。宿主用户可能有权访问整个磁盘，Harness 却仍然要把 Agent 限制在当前工作区。

Sandbox 约束能力，不解释意图

Sandbox 可以限制路径、进程、网络和系统调用，却不懂「修改测试来让测试通过」违反了任务。Policy 要读懂任务，并在动作发生前拦住这类行为。独立 Eval 只能在运行后判断任务有没有违规，Sandbox 则在动作获准后继续守住已经配好的资源边界。三层不能混用。

检查真实隔离至少要问：

1. 哪些工具被重定向到隔离环境；

2 工作区以只读还是读写方式挂载；

3 网络默认允许、拒绝还是经网关；

4 凭据是否进入环境；

5 初始化失败时失败关闭还是回退宿主；

6 Plugin/Extension 是否能直接使用宿主 API 绕过工具执行器。

pi 的示例 Sandbox Extension 在隔离失败后可以回退到本地，所以只看到「代码库里有 Sandbox」，还不能证明所有动作默认都在隔离环境里运行。

Session 保存历史，不自动保存环境

Session 通常能把消息、工具结果、模型选择、分支和摘要重新读回来，但它无法让已经退出的进程回到原状，也无法复原远端事务或抹掉工作区后来发生的变化。所以 Resume 之前，你得重新查看环境：

```
加载 Session
→ 找到最后已结算事件
→ 检查未完成 Tool Call
→ 读取文件、Diff、进程或远端状态
→ 判断副作用已发生、未发生或未知
→ 构造下一轮 Context
```

重放消息和重做副作用是两件事。如果编辑工具已经把内容写进文件，却在记下「完成」之前崩溃，Session 只能说它的状态未知。恢复时应该先读文件，确认 Patch 到底有没有应用，不能直接再做一遍。

Compaction 会让恢复变得更难

摘要可能写着「代码已经修复」，却漏了「测试尚未运行」，所以必须让人能从模型当前看到的 Context，一路追回完整 Session 里的记录和环境当时的状态。Codex Rollout、Gemini CLI Session、DeepSeek Harness Session、pi Entry Tree 和 OpenCode Message/Part 虽然投影历史的方式不同，却都得防止人们把摘要当成精确的执行账本。

恢复时还会用到 Tool Call ID、工具状态、目标路径、命令、退出码、工作区版本和未完成事项，这些内容应该尽量存成结构化数据。自然语言摘要能帮模型继续往下做，却不能证明副作用真的发生过。判定时不能靠它。

Revert 不是通用事务

文件 Snapshot 或 Git Patch 可以逆转工作区修改，却通常不能撤销：

已发送的网络请求；

数据库写入；

发布到外部系统的包或消息；

工作区外删除；

启动后仍在运行的后台进程。

OpenCode 会把 Patch/Revert 和 Session 后缀放在一起处理，但它仍然没有越过上面这些边界。其他 Harness 就算依赖 Git，也得说清哪些状态根本没进版本控制。对高风险工具，应当优先使用幂等键、事务或预演，别等出事后才把希望都寄托给 Revert。

练习：设计一次崩溃恢复

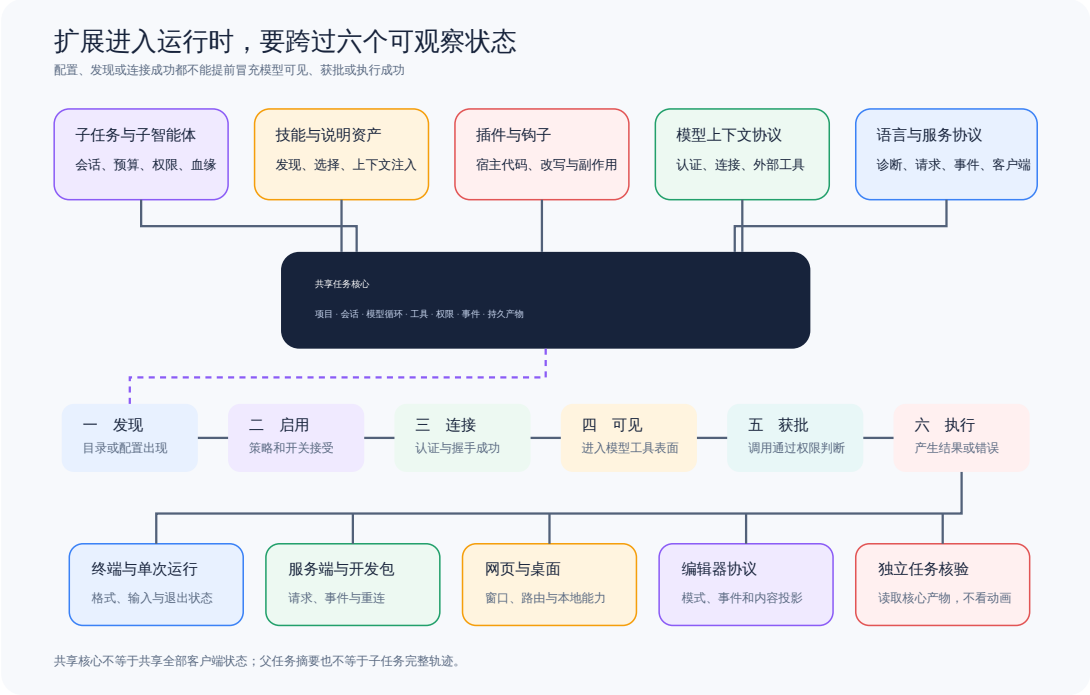
场景：Harness 已经请求把 > 改成 >=，但进程随后崩溃，而 Session 里只有 Tool Call 和「开始执行」，没有最终结果。请为两条课程分别写出恢复步骤，并指出应该在哪一层确认用户批准仍然有效。

下一篇：编排、Skills、Plugins、MCP 与产品表面如何分类

编排、Skills、Plugins、MCP 与产品表面如何分类

上一篇：权限、状态与恢复 · 返回课程总目录 · 下一篇：可观测性与独立 Eval

Agent 生态经常把 Agent、Sub-Agent、Skill、Plugin、Hook、MCP、Protocol、Server 和 UI 都叫作「扩展能力」，可它们插手运行过程的位置并不一样。有的会改模型输入，有的会添工具，有的会另起一条循环，还有的只是让另一个客户端接入同一 Session。要分清这些边界，你得先看每种机制负责什么，再比较六套 Harness 怎样实现它们。



先按责任分类

类别	改变什么	是否拥有独立 Loop	典型风险
Agent/Preset	模型、Prompt、工具和权限组合	可能	配置身份混淆
Sub-Agent/Task	创建另一项受控任务	是	权限升级、上下文泄漏、递归爆炸
Skill/Context	给模型增加按需知识或流程	否	提示注入、过期指令
Tool/MCP	增加观察或动作接口	否	远端副作用、能力声明与现实不一致
Plugin/Extension/Hook	修改装配、调用前后或 Context	通常否	进程内高权限、不可见改写
Protocol/Server	让外部 Client 控制 Session	否	身份、重连、事件重复与版本错配
CLI/TUI/Web/Desktop/IDE	展示和采集用户交互	否	把 UI 状态误当核心状态

同一个项目里的名词可能横跨两类责任，例如 Plugin 既能注册 Tool，也能安装 Hook，所以你分析它时要看清这两个动作各自改了什么。这两个作用不能混。

六条课程的代表机制

课程	编排与扩展	协议或表面	最值得核对的边界
DeepSeek Harness	Bundle、Preset、扩展与子任务	CLI、ACP 等表面	多包装配如何仍保留单一运行主链
Codex	子代理、任务编排、扩展与 Code Mode	CLI、exec、IDE、app-server 协议	Thread/Turn 事件如何被多个表面投影
Gemini CLI	Agents、Hooks、Skills 与 MCP	CLI 输出和协议表面	扩展介入 Turn/Scheduler 的哪个阶段
Claude	SDK Hooks、MCP、Agents 与 Skills 的公开契约	Python/TypeScript SDK 与 CLI 边界	公开控制协议不能补全闭源内部实现
pi	Resource Loader、Extensions、Coding Agent	TUI、Print、JSON、RPC、Protocol Server/Client	最小 Agent Core 如何被逐层组合而不是复制
OpenCode	Agent、Task、Skill、Plugin、MCP、LSP	Server、Protocol、TUI、Web、Desktop、ACP	服务端 Session 是共享真值，客户端只持有投影

详细证据还得回到六条课程里讲扩展和产品表面的章节逐项查，因为这个比较页只帮你判断各类机制负责哪一层，不能替代上游源码站点。

子 Agent 是另一条任务链

子 Agent 一旦接下任务，系统做的就不只是「再调用一次模型」，还得为这条任务链处理下面这些事情：

独立或可区分的 Session/Task 身份；

明确的输入投影，而不是无边界复制父 Context；

权限派生规则；

深度、并发、预算和取消限制；

返回父 Agent 的结果投影；

父子 Trace 关联。

OpenCode 用 Task Tool 创建子 Session，同时沿用父端的 Deny 和外部目录限制。Codex 课程会带你看子代理怎样参与编排，DeepSeek Harness 则靠装配和任务机制扩展运行。pi 的 Follow-up 仍在同一个 Agent Run 里接收追加消息，所以不能把它算作子 Agent。

运费任务可以让子 Agent 去「检查回归测试范围」，但父任务明令禁止修改测试，这条约束不能因为执行者换了名字就失效。子任务结束以后，父端不能只收到一句「完成」，还要拿到 Child Session ID 和可供核对的结果。

Skill 不是 Tool，MCP 也不是单一能力

Skill 通常把流程和知识写给模型看，所以是否加载 Skill 会改变 Context。Tool 提供可调用的接口，执行以后会返回观察，也可能产生副作用。MCP Server 还能分别声明 Tools、Prompts、Resources 或 Templates，因此连上 Server 只说明通道已经建立。连接成功还不够。

因此要分开检查：

1. 资源有没有被发现；

2 当前 Agent 是否有权看见；

3 能力是否进入本轮输入；

4 调用时是否还需要批准；

5 远端 Server 实际能否执行；

6 返回结果怎样关联到 Session。

如果界面只亮起一个绿色的「已连接 MCP」，用户很容易误以为能力已经可用，却看不到后面至少还有五道门槛。这个绿灯很会骗人。

Plugin 与 Extension 是高信任代码

Prompt、Skill 和 Tool Schema 主要决定模型能看到什么，进程内的 Plugin/Extension 却可以直接读文件、发网络请求、注册 Provider、改写 Tool Result，甚至绕开正常的权限入口。项目 Trust 只能决定是否加载这段代码，一旦把它装进进程，宿主不会自动收窄它的权限。

做安全审计时，你需要记下代码从哪里来、用的是哪个版本、安装脚本做了什么、按什么顺序加载，以及每个 Hook 在哪里插手。调用前拦截和调用后改写要分别记录。两边都要留证。如果用了 Context Transform，最好再保存经过安全脱敏的转换前后摘要。

多客户端只共享核心事实

CLI、Web、Desktop、IDE 或 RPC（远程过程调用）Client 可以同时观察同一个 Session，但它们应该从服务端共享事件、消息和状态。界面控件不能当核心事实。协议想让多个客户端可靠地共享这些事实，至少要处理下面几项：

Server/Project/Session 身份；

请求与响应关联；

Event Cursor 或 Revision；

重连基线与去重规则；

取消和权限问题的路由；

版本或能力协商。

pi Protocol 用 Frame（帧）、CBOR、Schema 和 Attach Lease 控制远程 Session，OpenCode 用 HTTP/Event 服务和 History/Cursor，Codex 则通过协议与 app-server 接入多个产品表面。界面显示「Done」，只说明这个客户端投影收到了某种终态。你要判断它究竟代表什么，仍得回到核心事件。

设计取舍：进程内组合还是服务化核心

pi 从轻量的进程内 Agent Core 起步，再逐层组合到 Protocol Server。OpenCode 把 Session 放在服务端，让多个客户端直接共享核心，Codex 则用 Rust Core 和协议支撑多种产品表面。DeepSeek Harness 通过 Bundle 组合多个包的能力。这几种做法各有侧重，没有哪一种能适合所有场景：

进程内组合部署简单、延迟低，但客户端与运行时耦合更紧；

服务化核心便于共享 Session 和远程控制，但必须解决身份、并发和重连；

闭源产品 + 公开 SDK 给出稳定契约，却限制了内部实现可核对深度。

所以比较这些设计时，你要从任务真正需要什么出发，不能看到「支持更多表面」就断定 Harness 更强。

练习：给六个对象分类

你可以任选一条课程，从中找出一个 Agent/Preset、一个 Tool、一个 Skill/Context 来源、一个 Hook/Extension、一个 Session 协议对象和一个 UI 表面，再逐个说明它改动了模型输入、控制流、环境还是展示层。

下一篇：运行证据怎样进入独立 Eval

运行证据怎样进入独立 Eval，而不是变成宣传结论

上一篇：编排、协议与产品表面 · 返回课程总目录

Harness 结束、工具返回成功、测试进程退出 0 和任务通过，各自回答不同的问题，也各有自己的证据门槛。这四类结论不能混用。这一篇会把六套课程里的 Event、Telemetry（遥测）、Session、测试和 Eval 串成一条证据链，看系统怎样先记下「发生过什么」，再判断「用户目标是否满足」，也帮你避开一个常见错误：把仓库检查通过写成已经可以投入生产。

四层结论

层次	典型问题	证据示例	不能证明什么
协议生命周期	模型流、工具或 Session 是否正常结束	Stop Reason、Tool Part、Idle Event	用户目标是否满足
环境事实	文件和进程实际发生了什么	Diff、命令、退出码、日志	测试是否覆盖任务口径
任务判定	冻结任务是否通过	独立测试、Rubric、约束检查	其他任务和部署是否可靠
发布决策	一组结果是否达到发布标准	隔离数据、统计规则、回归门槛	生产环境必然无故障

后一层可以读取前一层留下的证据，但中间必须有明确的判定过程，否则你不能直接把结论抬高一级。证据不会自己升级。

六条课程提供的观察面

课程	可观察运行事实	Eval 位置或边界	深入阅读
DeepSeek Harness	Session、产品表面、Feedback 与实际 Eval 接缝	只对源码中真实存在的适配与数据流下结论	产品表面、Feedback 与 Eval 接缝

课程	可观察运行事实	Eval 位置或边界	深入阅读
Codex	Thread/Turn Event、Rollout、Store 与产品表面	Trace 可供外部任务判定，事件终态不是 Score	事件、Trace 与 Eval 接缝
Gemini CLI	Tool/Session Events、Telemetry 与错误分类	Telemetry 观察运行，外部 Eval 冻结任务	Telemetry、错误与 Eval 接缝
Claude	SDK 消息、Result/Error、Hook 与 Session 契约	公开 SDK 能收集边界事件，不能证明产品内部全部 Trace	产品表面、错误与 Eval 接缝
pi	通用 Telemetry、Eval Harness Artifact 与 Summary	Harness 留证，外部 Scorer 产分，Summary 不创造分数	Telemetry、Eval Harness 与分数
OpenCode	Message/Part/Event、Diff、Share 与可选 OpenTelemetry	Share/Telemetry 不是评分器，独立 Eval 在外部运行	Share、Telemetry 与独立 Eval

比较这些系统时，别只数字段有多少。字段多不等于证据强。你真正要看的是：系统能否稳定关联输入版本、运行身份、环境里留下的产物以及最终判定。

最小跨 Harness Artifact

要把同一个运费任务交给不同 Harness 再比较结果，每次运行至少得保存下面这些信息：

```
task:
  id: shipping-boundary-100
  source_revision: 固定输入版本
run:
  harness: 项目与锁定版本
  session_id: 本次会话身份
  model: 实际 Provider 与 Model
  config_digest: 有效配置摘要
artifacts:
  trace: 模型、工具和错误事件
  patch: 最终修改或工作区哈希
  commands: 测试命令、目录、退出码和输出
  constraints: 测试文件未修改、路径范围等
evaluation:
  result: pass | fail | blocked | inconclusive
  reasons: 结构化原因
```

这里展示的只是教学用的数据契约，它把不同 Harness 的信息投影到同一组字段，并不对应六套项目里某个现成的同名结构。真正编写 Adapter（适配器）时，要先保留各项目的原始事件，再把它们映射到共同字段，不能为了把表格填整齐就丢掉差异。

Trace 与 Telemetry 的职责

Trace（执行轨迹）帮你还原事情的前后关系，例如哪次模型请求产生了哪个 Tool Call，工具返回的结果又进入了哪一轮。Telemetry 主要记录延迟、Token、成本、错误率和资源消耗等运行特征。两者不能互相替代。它们可以共用 Session/Span 身份，但某个指标即使超过阈值，也不能反过来改写原始事实。

安全日志还得处理源码、凭据和模型输入里的敏感信息，但脱敏后的展示数据不能覆盖原始的受控 Artifact（产物），否则排查问题的人和执行 Eval 的程序可能在不知情时各用一套数据。

独立 Eval 为什么要在 Harness 外再次验证

模型可能改掉测试文件，只运行一部分测试，也可能看错命令输出，所以 Harness 里的 Tool Success 只能说明工具返回了成功。这仍然不够。独立 Eval 还得进入干净环境，用事先冻结的命令重新检查目标断言和回归约束。

运费任务的最小判定可以是：

1. 应用补丁到固定源码；
2. 确认测试文件未被修改；
3. 运行金额 100 的目标测试；
4. 运行约定回归测试；
5. 保存命令、退出码和输出；
6. 把结果关联到同一 Run 与 Patch。

即使模型最后什么也没写，只要环境 Artifact 足够完整，独立 Eval 仍有可能判定结果。反过来，模型就算把结论写得再自信，只要目标测试失败，结果仍应判为失败。

失败、阻断与无法判断

评测没有通过时，别急着把所有情况都写成 Fail，因为你至少还得区分下面四类结果：

fail：运行完成且结果不满足任务；

blocked：缺少权限、依赖或平台能力，任务没有进入可判定状态；

inconclusive：Artifact 缺失或矛盾，无法可靠判断；

infrastructure error：评测基础设施自身故障，应与产品失败分开统计。

重试可以让系统从偶发的基础设施错误中恢复，却不能把真实发生过的产品失败从分母里删掉。如果允许模型反复尝试同一任务，就要保留每一次 Attempt（尝试），并且提前规定由哪一次 Attempt 代表整个 Trial（评测试次）。

四种常见误判

1. Session Idle = 任务通过：Idle 只说明当前控制循环没有继续工作。
2. Tool Success = 副作用正确：命令成功可能运行了错误测试，写入成功可能改了错误文件。
3. Telemetry 无错误 = 质量良好：没有异常不代表实现符合需求。
4. 上游单元测试通过 = 仓库生产就绪：测试只覆盖其明确断言和环境。

同样，课程链接能打开、内容检查通过、图示渲染正常，只能证明这个知识库满足了相应的质量要求，不能证明六个上游项目已经适合生产使用，也不能证明学习者已经具备生产能力。

训练与发布为什么要隔离

如果要把评测结果送进 DPO、GRPO、RFT 或其他训练流程，就得让版本化的 Reward Adapter 明确怎样接收输入、怎样归一化，以及怎样处理缺失值和失败。训练流程会把结果转成奖励，Checkpoint 选择会比较候选，最终发布 Eval 则要在隔离数据上作判断，因此三者必须提前分清数据用途，否则系统会对同一个判定器过拟合，最后把训练中的提升错当成真实的泛化能力。

这部分属于进阶内容，不会改变本仓库围绕 Agent Harness 源码展开的主线。如果你刚开始学，可以先分清三件事：执行 Trace 留下证据，独立 Scorer 作出判定，发布门槛决定是否发布。

练习：给三个结果分别下结论

现在看三个场景：A 中模型正常结束，却没有运行测试。B 中测试命令退出 0，可测试文件已经被修改。C 中模型 API 超时且没有产生补丁，随后评测机又发生磁盘错误。请分别写出它们的协议状态、环境事实和任务判定。

完成比较课程后，进入确定性实践，亲手复原一条源码调用链。

实践一：不用模型，复原一条源码调用链

返回课程总目录 · 下一项：运行最小 Agent Loop

这项实践只练习怎样读源码，你可以先从六条课程中选定一个足够具体的源码站点，不必运行产品，也不会让环境准备和模型凭据干扰这轮核对。你要顺着真实的调用关系，逐项检查这段代码为什么存在、谁调用它、输入从哪里来、它改了什么状态、结果返回到哪里，还要找出至少一个失败分支。把这些站点连起来以后，你才能看清单个函数在整条任务链里起什么作用。整个过程都以锁定提交为准，所以其他读者也能回到同一份源码复核你的结论。

先选择一个足够小的问题

你必须把问题收得足够窄，因为一旦从「这个项目架构是什么」之类的宽泛问题出发，调用关系很快就会淹没在目录和类型名称里。最好选一个能沿调用链回答的小问题，让每次搜索都进一步缩小阅读范围，而且问题还得落到可观察的状态或输出上。否则你很容易只解释了几个名字，却无法判断这套机制是否真按文章所说运行。例如：

- Codex 的并发工具完成后，历史怎样保持调用顺序；
- Gemini CLI 的 Scheduler 在哪里等待 Confirmation；
- Claude Python Agent SDK 怎样把 can_use_tool 回调接入控制请求；
- pi 的 Follow-up 为什么在外层循环消费；
- OpenCode 的 Permission 为什么由最后一条匹配规则决定；
- DeepSeek Harness 的 Tool Result 怎样重新进入 Agent Loop。

问题越具体，你就越容易找到真正的调用者和对应测试，也越容易看出文章漏掉了哪个关键分支。

七步阅读法

1. 打开课程给出的 40 位提交永久链接，确认文件与符号存在。
2. 先读函数签名和相邻类型，写下它解决的问题。
3. 向上搜索调用者，直到 Session、Turn、Agent 或产品入口。
4. 向下追返回值，直到状态持久化、事件发布或下一次模型请求。
5. 找到一个正常分支、一个错误分支和一个取消或清理分支。
6. 找到上游测试，确认它真的制造了文章描述的边界。
7. 写出测试没有覆盖什么，避免把局部断言扩大成产品结论。

交付模板

```
project: codex
commit: c9b19deb09c1841ce7acc33ddb96276030936a29
question: 多个工具并发完成后怎样保持结果顺序
why: 模型声明顺序和实际完成顺序可能不同，历史必须可重放
entry: codex-rs/core/src/session/turn.rs
caller: 本次 Turn 的工具批次处理
input: 按模型输出顺序建立的 Tool Futures
state_change: 结果按调用身份写入 Conversation Items
return: 下一次模型请求使用的 Function Outputs
next: Session 继续构造下一轮输入
error_path: 单个 Tool Error 仍与原 Call ID 关联
cancellation_path: Turn Cancellation Token 传播到在途调用
test: codex-rs/core/tests/suite/tool_parallelism.rs
tradeoff: 保持确定性提交会等待较慢的前序调用
limits: 测试不证明所有真实工具都适合并行
```

这份模板只是帮你整理阅读结果，不要求公开文章照搬这些固定标题，因为真正要保持稳定的是各项证据能否彼此对应。如果当前来源里找不到某一项，就明确写出「当前来源不可核对」，别拿其他项目的常见做法补上空白。缺口可以先留着。这样既不会把推测冒充源码事实，后来找到新来源时也知道该补在哪里。只有当调用者、状态变化和验证证据都能互相对上，你才有理由从局部观察归纳出机制怎样运作。

自己核对答案的标准

入口、调用者和消费者来自 Runtime 源码，不只来自 README；

解释了这段代码为什么存在，而不只是复述变量名；

区分了内存状态、持久记录与外部副作用；

错误分支说明副作用是否可能已经发生；

测试的输入确实触发目标边界；

对设计取舍至少写出一个好处和一个代价；

所有上游链接固定到同一提交。

进阶：画一张局部图

图里只留 5 至 8 个节点，把当前问题真正涉及的对象放进去，再用中文说明每个节点做什么，并把真实类型或函数名写在括号里。节点少一些，次要对象才不会冲淡箭头的含义，读者也能按正文的顺序检查调用链。画完再讲一遍。你可以从入口出发，沿着箭头复述任务。如果中途必须跳过某个节点，说明图里的关系多半还没理顺。每条箭头都应该对应源码里的调用、事件或数据转换。某段关系如果只能靠推断，就改用虚线，并在图注里交代依据和边界。

下一项：运行一个不需要模型的最小 Agent Loop

实践二：运行一个不需要模型的最小 Agent Loop

上一项：复原源码调用链 · 返回课程总目录 · 下一项：权限与崩溃恢复

这一项会把基础导读里的伪代码变成一段可以运行的确定性程序，让你从一次次重复执行中看清循环在哪里开始，又在哪里结束。脚本化 Model 会依次请求读取、编辑和测试，每一步怎么选都已经固定，因此观察结果不会混入模型采样的随机性。虚拟工作区负责承受副作用，Harness 则控制循环、检查权限、回填 Tool Result 并记录 Trace（执行轨迹），这三个角色仍按真实系统里的方式各管一段。整个程序既不访问网络，也不修改真实仓库文件，所以你可以反复重跑并比较 Trace。这样不会弄脏当前 checkout。

先运行测试

从仓库根目录执行下面的命令。

```
node --test examples/minimal-harness/harness.test.mjs
```

下面五个测试名直接对应这一课要建立的五条边界。

```
ok 1 - 最小 Harness 完成读取、编辑、测试和最终回复闭环
ok 2 - 工具结果与原 Call ID 一一对应
ok 3 - 权限拒绝不会产生编辑副作用
ok 4 - 独立判定不相信模型最终文本
ok 5 - 恢复未结算编辑时先观察环境，不盲目重放
# pass 5
# fail 0
```

第 3 条和第 4 条要放在一起看：权限被拒绝以后，虚拟源码仍保持原样，这说明 Harness 在调用工具体之前就拦住了动作。但单看日志还不够。测试同时核对审批记录和环境状态，才能说明工具到底有没有执行。独立 Evaluator 即使拿到模型声称「全部通过」的文本，只要源码里的边界错误还在，仍会判成 failed。它会重新读取环境产物，所以模型怎么措辞不会左右任务能否通过。这两条分别回答了「谁能改变状态」和「谁说了算」，测试也因此没有把执行成功和任务成功混成一件事。

先看四个对象

打开 examples/minimal-harness/harness.mjs：

对象	扮演什么角色
createScriptedModel()	用固定决定替代真实模型，消除随机性和凭据依赖
createShippingWorkspace()	保存虚拟源码并执行确定性目标测试
runHarness()	维护消息、审批、工具执行、Trace 和停止条件
evaluateShippingTask()	在运行结束后独立检查任务结果

脚本化 Model 虽然算不上智能系统，却保留了真实 Harness 最关键的一条协议边界：Model 只能提出 Tool Call，Environment 才能真的动手改变状态。由于 Model 会按固定顺序作出决定，你看到失败时可以直接去查权限、工具或循环怎样控制流程，不必先怀疑模型这次是不是采样出了不同结果。

沿一次运行读代码

1 runHarness() 把用户目标写成第一条消息；

2 Model 返回 call-read；

3 Harness 记录 tool_requested，等待 approve()；

4 虚拟工具读取源码，结果带着同一 Call ID 回填；

5 Model 看到结果后请求 edit；

6 编辑获准并改变虚拟工作区；

7 Model 请求 test，退出码和输出成为新观察；

8 Model 最后生成文本，Harness 以 completed 结束；

9 Evaluator 仍可独立运行目标测试，不依赖最终文本。

completed 说明循环已经走到哪里，passed 则说明任务有没有通过。两个字段不能混着看。前者只表示 Harness 碰到了某个停止条件，无法证明工作区里的运费边界错误真的消失。后者由独立检查给出，只有环境产物满足事先冻结的任务条件时才会成立。示例特意把两者分开，免得一个绿色勾同时吞掉两种含义。

修改一处，观察行为

依次尝试：

1 把编辑批准改成 Deny，确认源码没有变化；

2 让 Model 调用未知工具，观察 tool_failed 怎样进入消息；

3 把 maxSteps 设为 2，观察 Step Limit 与任务失败的区别；

4 让测试工具退出 1，确认 Model 文本不能覆盖环境结果；

5 复制一个 Call ID，思考真实 Harness 为什么需要拒绝重复身份。

每完成一项修改，先写预测，再跑测试。这样更容易看清哪些状态归 Model、哪些归 Harness、哪些又归 Environment。如果实际 Trace 和预测不同，就从第一个走岔的事件往回查，别只盯着最终文本。这个练习要看的是你能否观察并解释这些边界，功能多几个或少几个不会改变结论。

这个示例刻意没有实现什么

没有流式 Token 和 Provider Adapter；

没有并发工具批次；

没有真实文件、Shell 或 OS Sandbox；

没有 Session 持久化和 Compaction；

没有真实模型不确定性。

正因为省掉了这些机制，你才能逐行核对这个最小循环，并把每条 Trace 对回那几处具体的状态变化。等你读顺这条确定性路径以后，再引入真实模型或并发工具，才有一条基线可以判断新增的不确定性来自哪里。下一项只增加权限与恢复，而且继续使用同一个虚拟工作区，所以你仍然可以守住当前范围，不会突然跳进大型框架。

下一项：给最小 Harness 增加权限和崩溃恢复

实践三：增加权限判断与崩溃恢复

上一项：最小 Agent Loop · 返回课程总目录 · 下一项：独立 Eval

这一项不会再加工具，而是要把编辑工具获准、开始执行和留下记录这三个阶段分清楚。如果三个阶段共用一个成功标记，恢复器就判断不了副作用究竟走到了哪一步。最危险的情况是工具已经获准执行，却还没来得及写下能够结算这次调用的完成记录。这一步不能猜。编辑可能已经落盘，进程却在记录结果之前崩溃了。恢复器只有同时核对事件和环境里的真实状态，才能决定是继续推进、执行补偿，还是交给人工检查。

先观察权限拒绝

最小示例把 approve() 作为 Harness 的策略入口，测试会让 read 和 test 获准，同时拒绝 edit，然后检查以下事实：

- 记录中出现 tool_denied；
- 不出现对应 tool_started 和 tool_completed；
- 虚拟源码保持原样；
- Model 收到带原 Call ID 的拒绝结果。

这些断言只能证明应用层拒绝以后，Harness 没有调用虚拟 Tool Body，却无法证明 OS Sandbox 已经生效，因为整个示例仍跑在普通 Node.js 进程里。

为副作用写三阶段事件

真实工具至少需要：

```
tool_requested：模型提出动作，尚未执行
tool_started：权限已经通过，副作用可能开始
tool_completed：结果已保存，可以安全进入下一轮
```

如果最后一条记录是 tool_requested，通常可以认为副作用还没开始，因为 Harness 尚未放行这次执行。一旦记录停在 tool_started，日志只能证明工具已经开始跑，却说不清副作用有没有做完。这两种状态差得很远。恢复器必须先检查环境，再根据实际看到的结果决定继续、重试，还是转给人工处理。若日志里已经有 tool_completed，说明这次调用留下了可以继续消费的结果，恢复时就不该再次执行同一个动作。

运行恢复判断

示例里的 recoverPendingEdit() 不会直接相信 Session 文本，它会先读取虚拟工作区，再看实际内容决定该怎么做：

- 仍然包含 total > 100：返回 apply_edit；
- 已经包含 total >= 100：返回 continue_to_test；
- 两种都不匹配：返回 manual_review。

这里只处理一次结果确定的文本替换，所以它查看一处明确的文本差异，就能判断编辑有没有落盘。真实系统没这么简单。碰到真实 Patch（补丁）时，恢复器还要比较文件 Hash、版本、调用参数以及并发修改，免得把其他进程写入的内容错认成这次调用的结果。数据库或网络工具没有这么直观的文件状态，你还得去查幂等键、事务状态或外部系统，才能判断副作用究竟发生了没有。

自己扩展一个安全规则

为虚拟 Harness 增加「禁止编辑 tests/**」规则，并编写测试证明：

1. 工具即使对模型可见，执行时仍会被拒绝；
2. Pattern 使用规范化路径，不能通过 tests/../tests/ 绕过；
3. Deny 不会写入工具完成事件；
4. Model 得到结构化拒绝原因；
5. 独立 Evaluator 仍检查测试文件 Hash。

什么时候不应自动恢复

工作区内容与调用前后版本都不匹配；

写入跨越多个文件，只确认了其中一部分；

工具启动了后台进程；

网络响应丢失，服务端可能已经处理；

用户批准只针对已经失效的一次性上下文；

Session 与环境来自不同 Project 或 Server。

遇到这些情况，就该转给人工检查、启动补偿事务或使用专门的恢复器，不能只让模型「再试一次」。

下一项：让外部 Evaluator 独立判断结果

实践四：给任一 Harness 接一条独立 Eval 管线

上一项：权限与崩溃恢复 · 返回课程总目录

前三项分别教你怎样拿到源码证据、跑通确定性 Loop，以及判断恢复到底恢复了什么。最后再单独判断结果。Harness 执行任务并留下证据，外部 Evaluator（评估器）再到干净进程里检查产物，因此你不能把「Harness 正常结束」当成「任务结果正确」。

任务夹具

先创建一个小项目，只放入一个运费边界错误，并在开始运行前冻结下面这些内容：

初始源码版本或 Hash；

用户目标；

允许修改的文件；

禁止修改的测试；

目标测试与回归测试命令；

超时和基础设施失败口径。

任务一旦开始运行，Harness 就不能再改这些定义，否则同一次评测的目标和判定标准会跟着执行过程漂移。

三个组件

Target Adapter

Target Adapter（适配器）会启动选定的 Harness，并把任务和工作区交给它。

它还会保存 Harness 版本、产品表面、Provider（模型提供商）/Model、有效配置摘要、Session/Run ID、退出分类和原生 Trace（执行轨迹），但只负责在接口之间转换，不参与评分。

Artifact Collector

Harness 结束后，Artifact Collector 会收集最终 Diff、文件 Hash、命令输出、Tool Calls、Permission Decisions、错误、成本和时长。Assistant 的最终文本只是其中一份 Artifact（产物），就算它声称任务成功，也不能推翻其他环境事实。

Independent Evaluator

Independent Evaluator 会在新进程或干净工作区里应用补丁，重新运行事先冻结的测试，再检查测试文件有没有变化、Diff 是否越界，以及目标行为是否通过。模型自述不算分。它不会拿 Harness 的 completed、idle 或 Tool Success 来决定结果。

```
{
  "task_id": "shipping-boundary-100",
  "target": { "harness": "codex", "commit": "固定版本", "surface": "exec" },
  "artifact": { "diff_sha256": "固定哈希", "test_exit": 0 },
  "evaluation": { "status": "passed", "reason": "目标测试和回归测试通过，测试文件未修改" }
}
```

四种结果要分开

passed：环境产物满足全部任务约束；

failed：运行完成，但修改、测试或约束不满足；

blocked：缺少权限、依赖或平台能力，无法进入正常判定；

inconclusive：Artifact 缺失或互相矛盾，无法可靠解释。

如果 Provider 不可达，或者评测机磁盘已满，记录里还要写清楚故障来自哪项基础设施。你可以通过重试从基础设施错误中恢复，但必须保留原始运行，否则产品失败后反复重跑、最后只留下某次通过，会掩盖系统的真实表现。

与仓库里的确定性示例连接

evaluateShippingTask() 会重新执行虚拟目标测试，再检查 Trace 里有没有成功的测试结果。配套测试还会制造一种情况：finalText 声称全部通过，但源码里的边界错误仍然存在，而 Evaluator 最后依然给出 Failed。

这个 Evaluator 虽然很小，却把关键接口讲得很清楚：评分器读取环境状态和 Trace Artifact，模型说得再自信也改不了判定。

可选真实模型阶段

只有确定性管线已经能稳定记录每份数据从哪里来、经过了什么步骤，才适合接入真实模型。接入以后，还要额外锁定或记录 Sampling 参数、Tool Schemas、网络、Sandbox、超时、预算和线上模型版本。即便运行成功，结论也只适用于这项任务、这份配置和这个时间点。

如果还要设计训练奖励、挑选 Checkpoint（检查点）并设置发布门槛，就得继续分开各类数据的用途，但做完这项实践并不要求你先掌握这些内容。

参考：可观测性与独立 Eval 比较。

Aider：Repository Map 与 Architect Editor 分工

为什么值得单独看

Aider 有两个机制很适合放在一起看。Repository Map（仓库映射）会用有限的 Token 预算挑出仓库结构，让 Model 看到聊天文件之外的相关符号。Architect（架构师）模式则先出方案，再编辑文件，这两个阶段还能选用不同模型和编辑格式。前一个机制决定 Context 里放什么，后一个机制决定怎样把建议交给 Editor 落到文件上。

这两个机制能降低大型代码库的操作难度，却补不上所有证据缺口。信息仍可能漏掉。Map（映射）压缩以后可能丢掉关键文件，Architect 也可能拿着残缺的上下文出方案，Editor 甚至会把正确建议改错。本文只讲锁定源码中可以核对的选择过程和交接过程。模型表现不错、生成提交或测试碰巧通过一次，都不能证明 Task 做对了。

机制一：Repository Map；机制二：Architect 与 Editor

RepoMap.get_repo_map 先检查 Token 预算，也检查有没有其他文件。预算为零，或者其他文件一个都没有，它就直接跳过 Map。检查通过后，这个函数收下聊天文件、其他文件、用户提到的文件和标识符，再调用标签排序逻辑。若当前没有聊天文件，它还会从 Context 窗口里扣掉预留空间，把剩余容量拿来扩大仓库视图。

Repository Map 会把候选符号和引用排好顺序，再投进 Context，让每个 Token 尽量多带一些有用信息。它不包含完整文件，也不能保证每项依赖都会入选。配置本身不够。map_tokens、Context 窗口、用户提到的线索和排序结果会一起决定模型看见什么，所以复现实验还要保存最终 Map 或它的哈希，以及预算 Config。

ArchitectCoder 继承 AskCoder，它会先给出架构建议或修改建议。等 Response 完整返回后，如果内容不为空，并且没有开启自动接受，它就会问用户要不要继续改文件。用户批准后，代码再从主模型配置里选择 editor_model。没有独立 Editor 时，它就继续用主模型，同时设置 editor_edit_format，然后新建一个 Coder，把 Architect 的 Message 交给 Editor 执行。

交接以后，自然语言方案和结构化编辑 Protocol（协议）由两个阶段分别处理。Architect 负责分析，Editor 负责应用 Diff（差异）或整文件格式，但它们也可能指向同一个模型对象。有没有人工 Confirmation，要看 auto_accept_architect 的实际值。只有有效配置关闭了自动接受，而且确认交互确实发生过，才能说「Architect 模

式有人审」。

Editor 做完以后，会把总成本和 Aider 生成的提交哈希交回 Architect，当前消息也会移回上文。提交也会骗人。提交哈希能把 Artifact（产物）追溯到某次编辑，说明那次编辑形成了 Git 对象，但它证明不了需求已经满足，也排除不了测试被改坏的可能。Git 记录和任务 Score 必须分开看。

从哪里开始读源码

先读 RepoMap.get_repo_map，核对哪些条件会让函数提前返回，预算什么时候会放大，以及代码怎样调用 get_ranked_tags_map。如果还要解释排序，就继续查看标签图、Cache 和排名实现，同时记下哪些文件算 chat_files，哪些算 other_files。只看最终 Map 文本，你很难查出文件是在哪一步漏掉的。

再读 ArchitectCoder.reply_completed。空回复怎么处理、人工怎样确认、Editor 模型和编辑格式怎么选、新 Coder 怎么建、建议怎么交出去、提交哈希怎么收回来，都集中在这里。然后沿着代码查到模型配置，确认 editor_model 是独立模型，还是主模型的别名。

把两段源码连成因果链

要找到 Repository Map 真正接进模型 Context 的位置，就查最终 Map 文本进了哪一次 Query，别停在 map_tokens 这项配置上。教学实现应保存候选文件、符号排序、预算怎样裁剪以及最终内容哈希，再把哈希关联到 Architect 请求。错误来源才查得清。方案若漏了依赖，复核者才能继续判断：源文件没被扫描，排名太低，预算把它裁掉了，还是模型忽略了已经收到的信息。

Architect 会把一份能追到来源的建议消息交给 Editor。交接记录至少要写明 Architect 模型、输入 Map、原始建议、确认决定、Editor 模型、编辑格式和新 Coder 标识。用户拒绝后，就不能创建 Editor。哪怕两个阶段使用同一模型，也要分别留下 Call 边界，不能因为模型 ID 相同，就把规划和编辑挤成一次说不清的请求。

文件 Diff、Command 结果和提交哈希把 Editor 的动作连到最终产物。提交虽然给出了不可变引用，未追踪文件、工作树残留和外部副作用仍要另行收集。做最小实验时，可以让 Architect 给出正确建议，却故意让 Editor 改错一行，再在另一轮里让 Map 漏掉关键符号。前一种错在交接，后一种错在 Context 选择。这两类失败不能混。

沿一次任务走完整链

1. 入口确定聊天文件、其他仓库文件、提及文件与标识符，并计算 Map 预算。
2. RepoMap 对符号与引用排序，生成受上下文窗口约束的仓库投影。
3. Architect 读取任务、聊天文件和 Map，输出可执行的修改建议。
4. 根据有效配置等待用户确认或自动接受，再选择 Editor 模型与编辑格式。
5. Editor Coder 接收建议，修改文件并可能运行命令、测试或创建 Git 提交。
6. 外部验收比较最终差异、测试与需求，独立决定任务是否通过。

记录里至少要留下 Map 预算、最终 Map、Architect 原始建议、确认决定、Editor 模型与格式、文件 Diff、命令和提交哈希。只留最终提交，你就分不出错误发生在 Context 选择、规划还是编辑。只留聊天也不够，因为它证明不了磁盘究竟变成了什么样。

它补充了六条主课程的什么

Aider 这个扩展样本讲清两件事：怎样压缩代码 Context，怎样把规划和编辑分开做。六条一级主线还会覆盖更完整的 Runtime、Session、Permission 和产品 Surface。这里不做综合排名，也不要求所有 Harness（位于模型外部，负责输入装配、工具、权限、状态和产品表面的程序）照搬 Repository Map。你真正可以带走的是三组问题：Context 投影怎么生成，遗漏怎么查出来，两个角色交接时怎样留下来源。

独立核对时，Map 和 Architect 建议属于运行 Trace，文件 Diff 和 Git 对象则是环境产物。Evaluator 要读取冻结任务和最终产物，不能因为 Architect 给了建议、Editor 正常结束或 Git 生成提交，就直接算通过。

什么时候值得采用

代码任务如果符号关系很多，完整仓库又放不进 Context，Repository Map 会比较合适。项目若只有少量文件，主要内容不是代码，语言不受解析器支持，或者依赖主要藏在 Runtime 配置里，Map 可能花了成本仍漏掉核心事实。预算也不能无限扩大，因为更多符号会挤掉任务、History 和 Tool Result 的空间。效果要按具体任务验证。

如果方案要先复核、编辑格式要专门处理，或者不同模型适合分担不同角色，可以采用 Architect/Editor 分工。改动非常小时，多一次调用反而会增加延迟和误差，自动接受也会拿掉人工检查。本文只覆盖锁定源码中的 Map 和交接机制，不评价 Aider 的所有 Mode、真实模型质量或任何部署的默认安全。迁移时沿用这些问题和记录方法即可，不必照搬类名。

最容易误判的地方

Map 最大的风险在于选错信息。预算不足、解析失败或图排名较低，都可能让关键实现缺席，模型却容易把眼前这份投影当成完整仓库。角色名称也会制造错觉。叫作 Architect 和 Editor，并不能证明背后用了不同模型，也不能证明有人审过方案。

也别把 Git 当成完成门槛。自动提交便于追查，却仍可能收进错误改动、漏掉未追踪文件，甚至通过修改测试来凑出绿色结果。运行信号不够。命令返回零、提交已经生成以后，你还得结合冻结任务、禁止项和独立测试来解释。

怎样亲手核对

你可以准备一项跨三个文件的受控任务，把关键符号放在聊天文件之外，再分别用充足和极小的 Map 预算运行。记录最终 Map 有没有带上这个符号，也记录符号缺席后方案发生了什么变化。这个实验只验证 Context 怎样投影，单个模型跑出的结果不能代表普遍性能。

做 Architect 实验时，先关掉自动接受，确认用户拒绝后代码不会创建 Editor。然后批准一次，固定独立的 editor_model 和编辑格式，再检查 Editor 收到的内容是否等于 Architect 的建议。这一步不能省。最后用外部测试验收文件产物，并故意造出一个「提交成功但需求做错」的案例，确认门禁不会把提交本身算成通过。

读完后做四个判断

问题 1

Repository Map 是完整源码吗？

答案： 不是，它是受预算和排序约束的符号投影。

问题 2

Architect 与 Editor 一定使用不同模型吗？

答案： 不一定，没有独立 Editor 时会回退主模型。

问题 3

Architect 模式一定有人确认吗？

答案： 不一定，自动接受配置会改变这条边界。

问题 4

生成 Git 提交能否计为完成？

答案： 不能，仍需按任务契约验证差异、测试和副作用。

Cline：工具批准与可恢复工作区

为什么值得单独看

Cline 这个扩展样本要拆开两个常被混为一谈的问题：Tool 能不能执行，文件改动能不能恢复，这两条链不能混。Runtime（运行时）里的工具 Policy（策略）、Client 的 Approval 回调和 Executor 一起回答前一个问题，Git 提供的可选 Checkpoint 机制负责后一个问题。分清以后，你才不会看到用户点了批准就认定系统安全，也不会看到可以 Revert 就以为所有副作用都能撤销。

锁定源码里同时有 SDK 化的 Agent Runtime、Core（核心层）、Shared 类型和多种 Host（宿主）Surface。本文只总结能沿锁定路径核对的公共机制，不会拿 EditorUI 里的按钮、Provider 列表或能力说明，去推断每个宿主默认怎样运行。批准策略和 Checkpoint 都很依赖有效 Config，所以一定要查到 Runtime 里的实际值。

两条不能混用的链：批准与 Checkpoint

Agent Runtime 在执行工具前，会把基础工具策略和 Runtime 的覆盖规则合在一起。最终策略如果禁用了工具，Runtime 就生成 skipReason。如果策略只是关掉自动批准，Runtime 会通过 requestToolApproval 把 Session、Agent、迭代、Tool Call（工具调用）ID、名称、输入和策略发给宿主。宿主一旦拒绝，也会生成 skipReason，执行路径就收不到这个工具调用。

工具要求批准而宿主没配回调时，这条路径会关闭执行并返回 approved: false。批准回调一旦抛出异常，Runtime 就会返回 Deny（拒绝）Result 和原因，这样能避免界面失联后默认放行，但这项约束只作用于 SDK Runtime 路径。宿主如果还有别的执行入口，仍要单独核对。

批准和 Sandbox 管的不是一回事，用户或策略说「可以执行」，也只代表应用层放行。Command 究竟能访问哪些文件、网络、凭证和外部服务，还要看进程身份、操作系统隔离与工具实现。批准界面展示的参数也可能在执行前被转换，所以审计时要同时保存请求、决定和执行记录。

Checkpoint 管文件恢复，Core 配置规定，根 Agent Run（Agent 运行）开始时可以捕获一份可恢复的工作区 Snapshot。内置实现用 Git stash 和私有 ref 来保存，也允许调用方提供自定义创建函数，而源码把默认值明确写成

false，所以你得主动启用 Checkpoint，它才会工作。

即使启用了 Checkpoint，它主要也只管工作区文件，文件恢复有边界。它能比较运行前后的 Diff，也能恢复追踪和未追踪文件，却撤不回已经发出的网络请求、数据库写入、云资源变更或外部 Message。文件回来了，对话语义、模型端 Cache 和下游系统状态也不会自动复原。因此，文件恢复和副作用补偿必须分开设计。

从哪里开始读源码

先读 Agent Runtime 的工具策略与批准请求，重点追踪五条分支：policy.enabled、policy.autoApprove、批准回调缺失、回调拒绝和回调异常，看它们最后怎样生成 skipReason。然后继续读后面的执行函数，确认带着 skipReason 的 Call 确实到不了工具 Handler。

再读 Checkpoint 配置契约，这里直接写着默认值、调用时机和自定义契约。要验证具体实现，就沿 hooks/checkpoint-hooks.ts 查看代码怎样建立 Git Snapshot 和私有 ref，再用对应的上游测试核对未追踪文件、重复运行与恢复动作。

把策略、执行和快照连起来

批准链在策略算出结果以后，才把工具调用交给 Handler。记录实现时，要同时保存基础策略、Runtime 覆盖、最终的 enabled/autoApprove、调用参数哈希、批准请求和 skipReason。Handler 只能收到没有 skipReason，而且参数未变的调用。批准回调缺失、明确拒绝或抛错时，Runtime 都要生成结构化的拒绝观察，让 Agent 有机会调整，不能悄悄丢掉，也不能退回默认允许。

Checkpoint 会在根 Agent Run 建立前接入。只有有效配置已经启用，代码才创建 Snapshot，并把 ref、Run 标识和工作树基线写进 Session 元数据。子迭代不该反复创建无人管理的 Snapshot。恢复时，先列出哪些追踪文件和未追踪文件会变化，再执行 Git 层恢复，最后重新计算 Diff，而 Snapshot 和批准 Event 虽然共用一个 Run ID，却不能合成一个含糊的安全状态。

教学实现可以在临时 Git 仓库里接一个本地计数服务，让工具既改文件，又调用外部服务，然后再恢复 Checkpoint。此时文件应当回到基线，外部计数却仍然增加，系统还要明确提示用户补做补偿。如果 UI 显示「全部回滚」，产品文案就夸大了真正能恢复的范围。

沿一次受控编辑走完整链

1. 模型产生工具调用，Runtime 先运行输入转换与前置钩子。
2. Runtime 合并工具的基础策略和当前覆盖，禁用策略直接阻断。
3. 需要批准时，宿主客户端收到带会话与调用归属的请求并返回决定。
4. 只有没有 skipReason 的调用进入真实工具执行器，结果再回到 Agent 迭代。
5. 若显式启用 Checkpoint，根运行开始前保存工作区快照并登记到会话元数据。
6. 失败后可选择比较或恢复文件；外部副作用由独立补偿与验证流程处理。

一条能审计的 Trace 至少要把工具调用 ID、输入哈希、策略来源、批准者或自动策略、最终执行参数、返回值和文件 Diff 关联起来。Checkpoint 还得另记 ref、创建时间、runCount 和恢复操作。Trace 里没有批准 Event，不能自行解释成自动批准。没有 Checkpoint，也不能推断工作区没变。

它补充了六条主课程的什么

Cline 这个扩展样本补充了 Editor 宿主怎样让人控制工具，又怎样恢复文件。一级主线已经分别讲过不同 Harness（位于模型外部，负责输入装配、工具、权限、状态和产品表面的程序）里的 Permission、Session 和工具生命周期。这里不把 Cline 扩成一条全面主线，只借这个案例分清四层：应用批准、OS 隔离、文件 Snapshot 和任务 Eval。

Runtime Trace 和 Checkpoint Diff 都是运行证据。Evaluator 还要读取最终工作区、必要测试和禁止出现的副作用，不能只看到一次批准、一次成功执行或一个可恢复 Snapshot，就判定任务做对了。

什么时候值得采用

出错就关闭执行的批准回调，适合嵌入式或多宿主 SDK。宿主失联时，它不会顺势扩大执行权。不过，它替代不了参数语义检查、最小进程权限和操作系统隔离。已经受约束的低风险工具可以自动批准，危险命令、对外发送和不可逆操作仍要采用更窄的策略，并交给外部控制。

Git Checkpoint 适合处理受版本控制的工作区改动，也适合预览 Diff 和恢复文件。数据库、远端 API、后台进程、项目外目录以及被忽略的大对象，并不在它天然覆盖的事务范围内。本文不覆盖所有 Cline 宿主和 UI，也不会把 Checkpoint 说成完整的 Session 恢复。选用这项机制时，必须让用户看清哪些动作能撤销，哪些后果要另行补偿。

最容易误判的地方

最常见的误判，是把批准当成安全审查，因为用户可能看不到完整参数，策略可能放得太宽，工具也可能在获批后扩大作用范围。另一个误判是认为 Checkpoint 默认存在，可锁定配置明确把它关掉了。即使 Git 工作区已经恢复，外部 API、数据库和消息系统也可能早就留下无法撤销的状态。

还要把 Session 状态和任务是否做对分开。宿主界面显示完成、Runtime 正常停止、所有工具都获批，只能说明运行没有在这些位置出错。模型仍可能改错文件、漏掉需求，甚至删测试来制造绿色结果。独立验收不能省。

怎样亲手核对

批准实验可以给同一个无副作用工具设置三种策略：自动批准、请求批准后拒绝、要求批准却不提供回调。验证时确认后两种策略都不会调用 Tool Handler，并保存各自的 skipReason。再让批准回调主动抛出异常，检查 Runtime 是否仍会关闭执行。实验还要固定锁定提交和配置，不能只截一张 UI 图当证据。

做 Checkpoint 实验时，先在临时 Git 仓库里准备追踪文件和未追踪文件，再分别用关闭与启用两种配置启动根 Run。关闭时应当没有 Snapshot，启用时则要拿到能比较的 ref，并验证恢复动作只影响工作区。最后模拟一次外部副作用，用实际结果证明 Git 恢复不会自动补偿它。

读完后做四个判断

问题 1

工具获批后是否说明安全？

答案： 不说明。批准是应用授权，仍需验证执行参数、进程权限和沙箱。

问题 2

Checkpoint 默认开启吗？

答案： 没有。锁定配置明确为选择启用。

问题 3

恢复快照能撤销 API 调用吗？

答案： 不能。它主要覆盖工作区文件，外部副作用需要补偿机制。

问题 4

Cline 为什么是扩展样本？

答案： 这里提炼的是批准与恢复分层，不承担对整个产品和所有宿主的一级全景分析。

goose：Recipe 与 Extension 的分阶段装配

为什么值得单独看

goose 把可复用的工作流输入和外部能力接在一条链上，沿着源码就能逐段核对。Recipe（配方）写明标题、指令、启动提示、参数、扩展和设置，Agent 与 ExtensionManager 再把 Extension 接入具体 Session，汇总其中的 Tool。这样确实方便复用任务，但也容易让人误以为「配置里写了扩展，模型就一定能调用」。

这里把扩展接入过程拆成五段来看：系统有没有发现并解析配置，扩展能不能连上，Model 看不看得到工具，调用有没有获准，工具最后是否真的执行。把每一段都记下来，你才能看出问题出在配方、依赖、Protocol（协议）、模型选择还是 Permission（权限）。Recipe 和工具清单都不能充当评测结果，更不能证明任务已经完成。两者都不够。

Recipe 如何变成会话中的真实工具

Recipe 数据结构要求填写版本、标题和描述，也允许加入模型 instructions、启动 prompt、extensions、settings 与活动提示。源码注释规定 instructions 和 prompt 至少要有一个，因此配方既可以持续约束模型，也可以直接发起一次会话。

Recipe 把扩展配置和模型输入装进同一个可移植对象，但不会预先写死扩展会产出什么，因为参数替换、嵌套配方、工作目录和用户配置都会改变最终值。这一点要记住。因此，实验记录要保存解析完成的有效 Recipe，不能只留下原始 YAML，而只供界面展示的活动提示也不能证明执行到了哪一步。

会话构建器先把待加载的扩展整理成「名称、配置」对，再调用加载函数。Agent 只有在内部装载成功后，才由 add_extension 保存扩展状态。普通扩展会和会话工作目录一起交给 ExtensionManager，前端扩展则从另一条分支插入。随后，list_tools 向 ExtensionManager 取得带前缀的工具，再与前端工具合并。

这个顺序不能打乱。发现配置只说明候选项存在，装载成功只说明 Client 或前端完成了注册。即使工具已经列出来，也只表示当前会话有这批候选能力，模型最终能看到哪些工具，还要看系统怎样构造并过滤 Context。至于模型会不会调用、调用能否执行，则取决于模型选择、权限判断和执行结果。系统可以保存状态，却保存不了可用性，因为下次启动进程时，外部命令、网络或凭证都可能已经失效。可用性要重算。

从哪里开始读源码

你可以先读 Recipe 数据结构，把每个字段由谁填写、交给谁用列出来。再沿 builder 和验证代码往下读，核对参数要求、模板渲染，以及子配方怎样拼出最终指令。别从示例 Recipe 反推默认配置，因为示例只能证明这种写法允许存在。

扩展怎样装进会话，最好从 Agent 的扩展装载路径 看起。成功装载后怎样按会话保存状态，前端扩展和普通扩展怎样分流，工作目录怎样传递，工具怎样汇总，都集中在这段代码里。若还想看协议细节，再进入 extension_manager.rs 与 MCP client，核对它怎样连接、命名工具并分派调用。

把配置、连接和工具投影连起来

Recipe 解析完成后会得到一个有效对象，会话就从这个对象读取配置。等参数替换、嵌套配方和用户设置全部合并后，系统应当计算哈希，并分别保存原始来源与最终值。模型指令、启动 Prompt 和 ExtensionConfig 都要引用这份哈希。这样一来，即使同名 Recipe 因参数或工作目录不同而表现不同，复核者也不会只看到两个相同的文件名。

ExtensionConfig 能不能进入工具表，要看装载是否成功。普通扩展连上之后才进入 ExtensionManager，前端扩展则由 Host（宿主）登记。只有当前会话装载成功的实例，才能提供带前缀的工具。系统还应保存 Tool Schema、服务版本和连接时间的 Snapshot（快照），再由上下文构造器筛出模型本轮可见的集合。已经保存的扩展配置只是下次 Resume 时的候选，不能跳过重新连接和枚举能力。

Tool Call（工具调用）还要过第三道关。系统先根据模型可见的名称找到具体扩展实例，再检查权限、发送协议请求，最后把带原调用 ID 的 Result 送回来。若服务在枚举工具后重启，或者改了 Schema，本次调用就应该明确失败或重新协商，不能继续拿旧缓存参数请求未知版本。旧参数不能照发。做教学实验时，可以在枚举结束后关掉服务，看系统会不会把「列表还在」误报成执行成功。

沿一次 Recipe 启动走完整链

1. CLI、桌面或调度入口取得 Recipe，并解析参数、指令、启动提示与扩展列表。
2. 会话建立 Provider、工作目录和有效提示，整理需要加载的 ExtensionConfig。
3. 普通扩展由 ExtensionManager 建立客户端，前端扩展由宿主登记；失败保持显式错误。
4. 成功装载后记录会话扩展状态，再汇总前缀化工具与前端工具。
5. 工具表进入模型可见能力，模型产生调用后再经过权限、检查与真实执行。
6. 轨迹、产物与冻结 Recipe 进入独立评分，判断这次任务结果。

最好给每个扩展分配一个稳定名称，并记录配置哈希、连接方式、服务版本、发现的工具、过滤后的工具、调用 ID、权限决定和返回结果。若 MCP 服务会动态变化，还要记下连接时间和能力变化。否则，你很可能拿启动后才出现的新工具，去解释启动前已经发生的运行。

它补充了六条主课程的什么

goose 不会成为第七条一级主线。它让我们看到 Recipe 怎样驱动编排，也让我们看到会话怎样装入扩展。拿这两套机制回看六条一级主线时，你要逐项追问系统只是发现了配置还是扩展真的连上了，工具有没有进入模型请求，权限判断有没有在调用时生效，以及失败是否完整留在 Trace 里。

Recipe 很适合充当可复现实验的输入。不过，实验仍要固定来源 Commit、Provider、模型、扩展服务和工作目录，Evaluator 也必须读取真实 Artifact。同一份 Recipe 能成功启动多个会话，只能说明装配过程可以重复，不能推出每

次任务都会得到相同结果。结果仍会变化。

什么时候值得采用

如果你想把重复任务、参数、指令和依赖扩展打包成可审核的输入，或者让团队共享受控工作流，Recipe 很合适。若任务高度依赖交互，扩展又使用短期凭据，或者 Environment 经常变化，Recipe 便只能固定意图，固定不了在线能力。运行清单仍须写清实际使用的 Provider、服务、工具和工作目录。

会话按需装入 Extension，很适合动态 MCP 和前端工具，不过你也要承担供应链、命名冲突、连接恢复和权限治理的成本。小型离线任务可能更适合静态内建工具。这里讲的只有锁定源码中的 Recipe 和装载次序，无法证明任意 Recipe 都安全，也无法证明扩展默认受到 Sandbox 隔离，或恢复后仍然可达。迁移时要保留五个阶段各自的状态，也要留下失败诊断。

最容易误判的地方

第一类风险，是看到目录里有 Recipe 或 Extension 配置，就认定扩展已经装载。第二类风险，是 MCP 握手成功、工具也列出来了，便认定模型已经收到、选择并获准使用某个工具。第三类风险，是扩展状态写入会话后，就认定它会一直可用。到了恢复时，外部二进制、端口、凭证和网络仍可能失效。

Recipe 还可能带入不受信任的指令和文件参数。解析成功并不表示内容安全，扩展返回的文本也可能影响模型后续判断。实验要同时固定输入来源、信任级别、工作目录和权限策略，还要把高风险工具放进外部隔离环境。

怎样亲手核对

你可以先建一个只提供只读工具的测试扩展，再建一个故意连不上的扩展，然后分别用同一份 Recipe 运行。失败的扩展不应进入工具表，成功的扩展则应加上前缀，并绑定当前会话。随后禁止调用这个工具，检查系统能否分清「模型可见」和「允许执行」，同时保存解析后的 Recipe 和每个阶段产生的 Event。

再做一次恢复实验：扩展成功装载后，关掉外部服务，再恢复会话，看看系统会不会把已保存的状态误判为当前可达。Eval 仍要单独检查目标文件或结构化输出，因为 Recipe 已加载，或者工具调用返回 Success，都不足以提前判定任务通过。证据仍看产物。

读完后做四个判断

问题 1

Recipe 中声明 Extension 是否等于启用？

答案： 不等于，还要解析、连接并进入当前会话工具表。

问题 2

工具出现在列表中是否一定被模型调用？

答案： 不一定，还受上下文、模型选择、权限和执行错误影响。

问题 3

扩展状态持久化是否保证恢复成功？

答案： 不保证，外部依赖可能已经变化或不可达。

问题 4

Recipe 能否直接作为发布评测？

答案： 它是可复现输入的一部分，仍需独立任务、评分器和发布门槛。

mini-swe-agent：用最小循环看清执行边界

为什么值得单独看

mini-swe-agent 很适合拿来认识最小可读的 Agent Harness（智能体框架）。它用很短的源码，把 Model、Agent 和 Environment（运行环境）三个角色串了起来。你可以顺着一次查询往下看：模型给出动作，环境执行动作，观察结果写回，退出信号最后收住循环。这套最小结构怎样划清边界，才是这里要看的重点。代码少不代表能力弱，结构清楚也不代表它默认安全或足以用于生产。

它补充六条一级主线，却不能替代其中任何一条。一级主线会完整讲解 Config、Session、Permission（权限）、Extension 和产品 Surface（接口面）。这个样本则把教学范围缩小，让你在更短的源码路径里核对 Agent Loop 由哪些环节组成。源码明确写出了重试、限额、Trace 保存和执行环境，你也更容易分清哪些事实能直接核对，哪些还得靠外部实验。

最小循环保留了哪些必要状态

DefaultAgent.run 先清空 Message，放入 System（系统）消息和用户消息，然后反复调用 step。每次 step 依次做几件事：先查询模型，把响应里的动作交给环境执行，再由模型 Adapter（适配器）把输出整理成观察消息，放回 History。这里分得很清楚，模型决定下一步做什么，环境负责产生事实，两者合起来形成最小的查询、执行和观察循环。

成功提交并非唯一的终止方式，每次查询前，代码都会检查 Step 数、累计成本和墙钟时间。连续出现格式错误时，循环会进入对应的退出状态，遇到未捕获异常时，系统则先保存轨迹，再继续向外抛出。只有最后一条消息的角色变成 exit，循环才会结束，并返回这条消息的 extra。所以，Submitted、LimitsExceeded、TimeExceeded 和错误退出只说明控制流程怎样收场，不能直接判定业务任务通过还是失败。

另一个关键机制是环境可以替换，而每换一种环境，风险边界也会跟着改变。LocalEnvironment（本地环境）的类说明明确写着，它会直接在本机执行 Bash（Bash 命令行）命令。实际调用还会合并进程环境与配置环境，再交给运行函数。它提供的是一条简单而透明的执行路径，并不会自动带上容器、虚拟机或系统调用过滤。

本地环境读取输出的第一行，如果那里出现特殊提交标记，它就触发 Submitted，并把后面的文本当作 submission。这个 Protocol 方便系统把「Agent 主动结束」与普通命令输出分开，但它只是 Harness 内部约定的终止信号。恶意命令或写错的命令同样可以打印这段字符串，所以别把责任混在一起，外部 Artifact 检查、测试和 Scorer 仍要独立运行。

从哪里开始读源码

你可以先读 DefaultAgent 的最小循环。run 控制循环，query 检查资源门限并调用模型，execute_actions 则让环境执行动作，再把观察写回。然后读 LocalEnvironment.execute，确认本地执行怎样选择工作目录、继承环境、捕获异常，以及识别提交标记。

如果还要验证 Resume 行为，再看 DefaultAgent 怎样分别处理 FormatError、InterruptAgentFlow 和未捕获异常。你可以把每条异常分支整理成五列，逐项记录它是否记账、是否追加消息、是否继续循环、是否生成退出状态、是否保存轨迹。这样就不会把性质不同的重试统称为恢复。

把模型动作、环境观察和退出协议连起来

模型与 Agent 之间传递一条完整消息，不依赖还在流动的半成品。query 调用模型前先检查步数、成本和时间，收到 Response 后，再把内容与费用写入历史。只有已经生成完整的动作，才会交给 execute_actions。做教学实现时，最好让模型适配器解析动作，让 Agent 只负责安排查询和执行。这样你可以用录制好的响应单独验证循环，不必承受在线模型的随机性。

Agent 把动作输入交给 Environment，Environment 收到工作目录、命令和环境变量后，返回退出状态、输出或控制异常，模型适配器再把观察整理成下一轮消息。调用 ID、动作原文、执行目录和返回码必须一起保存，少了其中一项，LocalEnvironment 在 Host 上产生的错误就很容易被错记成模型错误。

退出协议还要接到外部 Eval，因为 Submitted、限额和错误状态只会写进 Trace，submission 也只是候选产物。两者不能混。Scorer 还得读取文件和测试，才能判断真正的结果。你可以让最小原型打印提交标记，却故意写错目标文件。这个实验能直接证明，Agent 即使正常退出，任务仍可能失败，也就切断了「框架结束等于任务通过」这条错误推断。

沿一次最小任务走完整链

1. 入口把任务渲染进系统模板和实例模板，形成初始消息历史。
2. 查询前检查调用次数、累计成本与运行时长，超过门限就追加退出消息。
3. 模型根据当前消息返回文本与零个或多个动作，Harness 记录调用成本。
4. Environment 逐个执行动作，模型适配器把结果转成观察消息，再进入下一轮。
5. 格式错误可以在上限内恢复；退出消息、提交标记或硬错误使当前运行停止。
6. 外部验证读取轨迹和文件产物，以冻结任务契约判断结果。

这条链要优先保存模型调用序号、累计成本、动作原文、工作目录、命令返回码、异常类型、退出状态和 submission。若只保存最终回答，Tool Call 失败和恢复经过都会丢失。若只保留完整日志，却不固定 Task、配置和来源 Commit，你同样无法复现当时的运行条件。

它补充了六条主课程的什么

DeepSeek Harness、Codex、Gemini CLI、Claude、pi 和 OpenCode 的主线课程，要讲清各自完整的产品和工程结构。mini-swe-agent 不参与谁更强的横向排序，它只提供一副基准骨架。面对更复杂的 Harness，你可以拿它追问模型请求在哪里形成、谁解释工具动作、真实执行发生在哪里、观察怎样回到 Context，以及最后由谁决定退出。

它也能帮你把评测分层。Harness 的退出状态只进入 Rollout，submission 只是候选产物，Evaluator 还要读取冻结任务和最终文件。这些职责仍然存在，只是没有全部内置进来。

什么时候值得采用

如果你要学习循环怎样推进、动作怎样解释、观察怎样写回，或者资源限额和环境替换怎样生效，mini-swe-agent 很合适。它也适合用来构造确定性的状态机实验。不过，它无法单独证明企业权限治理、多客户端协议、长期会话、分布式恢复或发布评测已经具备。核心实现写得短，这些责任也不会自动出现。

LocalEnvironment 适合在可信临时目录里教学和调试。面对不可信任务、未知仓库或敏感凭据时，你应当选择有明确隔离措施的环境，并验证挂载、网络和进程权限是否真的生效。这里仅覆盖锁定的 DefaultAgent 与 LocalEnvironment 路径，不评价其他环境或部署。它值得借鉴，因为边界清楚，并非因为实现越小越好。

最容易误判的地方

第一类风险，是把 LocalEnvironment 当成 Sandbox。它会直接继承宿主进程可以访问的文件、命令和网络。调用前即使有 Prompt 或命令格式约束，也替代不了操作系统隔离。确实需要隔离时，应选择 Docker 等其他环境实现，或者在 Harness 外提供受限账号、容器和网络 Policy，再验证这些限制是否真的生效。

第二类风险，是看到框架退出便认定产品成功。打印提交标记、触发 exit 或返回 submission，都不能证明测试通过、需求满足，也不能证明运行没有造成破坏性副作用。第三类风险，是反复重试格式错误，最后只保留一次成功，把前面的失败藏起来。基础设施恢复后可以再试，但真实任务失败不能靠这种办法改写。

怎样亲手核对

你可以做三个最小实验。正常链让模型返回一个只读动作，并确认观察消息进入下一轮。限额链把 step limit 设为一，检查第二次查询前有没有生成 LimitsExceeded。边界链在临时目录里运行 LocalEnvironment，记录它确实用了宿主进程和环境。这些实验只能证明锁定版本面对受控输入时怎样运行，不能外推到其他环境、模型或部署。

核对时要看两组信号：控制流信号包括退出状态、调用次数和异常，任务信号包括预期 Diff、测试、禁止副作用和 Score。只有任务信号满足冻结的任务契约，结果才算通过，而控制流日志只能解释结果，代替不了结果。

读完后做四个判断

问题 1

最小实现是否意味着没有恢复？

答案： 不是。源码包含格式错误恢复、限额和轨迹保存，只是恢复策略较集中，必须逐分支核对。

问题 2

LocalEnvironment 是否默认安全？

答案： 不能这样判断。它明确直接执行宿主命令，安全边界取决于外部进程权限和所选环境。

问题 3

出现 Submitted 是否可以计为通过？

答案： 不可以。它只表示 Harness 收到提交协议，仍需独立检查产物和任务契约。

问题 4

为什么它只是扩展样本？

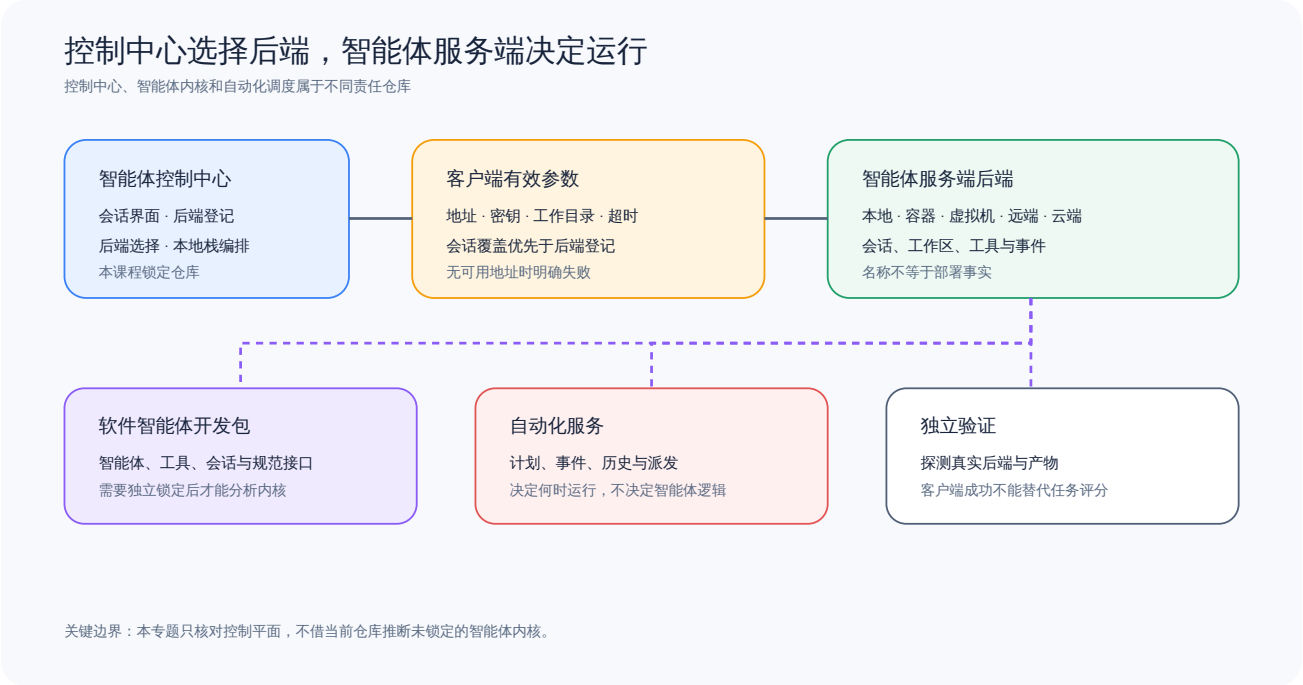
答案： 它突出最小循环的教学价值，但不承担一级主线需要的完整产品表面、部署与治理覆盖。

OpenHands Agent Canvas：多后端控制平面与仓库边界

为什么值得单独看

当前锁定的 OpenHands 仓库把自己定位为 Agent Canvas，也就是供 Coding Agent 和自动化任务使用的自托管控制中心。这里的 Canvas（画布）负责启动 Session、登记并切换后端，再把不同的 Agent Server 投影到同一个界面。这个定位已经不同于早期常见的「OpenHands 单体 Agent 仓库」。因此，你要先看当前仓库究竟负责什么，不能只凭项目名就认定源码里一定包含 Agent Loop（智能体循环）、Tool 实现或 Sandbox 运行时。

Agent Canvas 让我们看到，控制平面怎样把多个 Agent 后端组织成用户可以操作的产品。它可以连接本地、远端、云端或兼容 ACP（Agent 客户端协议）的 Agent，但这些名称只说明连接类别，不能证明真实后端已经存在。要判断后端到底跑在哪里，采用了什么隔离措施，又支持哪些版本能力，还得逐项检查有效 Config、网络探测、服务信息和部署证据。



控制平面怎样选择并连接真实后端

Canvas 每次调用 Agent Server，都不会盲目使用一个写死的全局地址。getAgentServerClientOptions 会读取当前有效的本地后端，也允许本次 Call（调用）用 host、conversationUrl、sessionApiKey、apiKey、workingDir 和 timeout 覆盖对应配置。若系统既找不到可用后端，也没有收到明确地址，就会抛出 NoBackendAvailableError，防止请求悄悄发往未知目标。

代码按明确的优先级选择地址与凭证，会话密钥盖过调用密钥，调用密钥再盖过登记后端的密钥。明确传入的 host 或 conversation URL 可以改写后端地址，工作目录则取自调用参数或 Agent Server 配置。不过，workingDir 只是发给后端的一项运行参数，看到这个字符串，不能据此认定后端已经配置容器挂载、路径白名单或文件系统隔离。

仓库 README 还用一张表分清各仓库做什么，其中当前仓库负责 Agent Canvas 前端、用户控制中心、后端选择和本地栈编排。software-agent-sdk 负责 Python 的 Agent SDK（智能体开发工具包）、Agent Server、Agent、工具、

会话、工作区和 Event，automation 则负责定义自动化任务、安排计划、接收 Webhook、保存运行历史并派发任务。

读自动化链时，要把「何时运行」和「运行什么」分开。这两件事不能混。Automation Server 根据计划或事件决定何时派发，Agent Server 与 SDK 决定会话和 Agent 怎样执行，Canvas 只提供配置和观察入口。如果你只看到 Canvas 页面成功创建了一条自动化，最多能确认控制对象已经登记。Scheduler（调度器）有没有触发、任务有没有派发成功、Agent 有没有完成工作，以及下游集成有没有收到正确结果，都还没有得到证明。

从哪里开始读源码

先读 锁定 README 的仓库责任表，看清代码分布在哪些仓库。然后读 getAgentServerClientOptions，核对 host、apiKey、workingDir 与 timeout 怎样组合。把这两处放在一起看，就不会把 Client 的 Router（路由器）代码当成 Agent Core（Agent 核心），也不会因为界面「支持某后端」，就认定后端已经部署。

继续往下读时，可以先沿 backend-registry 看系统怎样登记后端、选择当前活动后端，再沿具体 API service 追踪代码何时构造客户端参数。核对每项能力时，要把 Canvas 的配置记录、最终请求地址、Agent Server 返回的版本与能力，以及实际会话留下的 Artifact 记在一起。缺少其中任何一项，这项能力就只能标为部分可用或无法核对。

把 Canvas、Agent Server 与 Automation 连起来

系统登记后端后，客户端构造器会读取当前有效的 Backend，再叠加本次调用给出的覆盖值。每次发出 Query 前，构造器都会重新解析 host、conversation URL、会话密钥、API Key、工作目录和超时，并保留各项参数来自哪里、谁优先。若没有可用地址，就应明确失败，代码不能把上次活动后端缓存成永远有效的目标，更不能在切换会话后继续使用属于另一主体的密钥。

Canvas 通过一份带版本的客户端 Protocol 连接 Agent Server，连接成功后，它应先读取服务信息和能力，再创建或 Resume 会话。事件、Artifact 和最终状态都属于后端 Session，Canvas 只保存它们在界面上的投影和局部状态。workingDir 只是请求参数，Server 仍要自己解释并约束路径，客户端不能拿这个字符串替后端证明挂载或沙箱已经存在。

Automation 和 Agent Server 用派发记录衔接，计划或 Webhook 先生成 dispatch ID，派发动作再创建具体 Session，后端返回 Result 后，系统把它写回运行历史。这里有四个不能混为一谈的状态：自动化已经创建、派发成功、会话结束、Task 通过。做教学测试时，应在每个边界主动制造失败，检查控制平面会不会用绿色配置卡片遮住「根本没触发」或「产物不正确」。

沿一次多后端会话走完整链

1. 用户或配置注册一个后端，Canvas 保存地址、类别和所需认证信息。
2. 当前界面选择有效后端；会话也可以携带独立地址、密钥和工作目录覆盖。
3. 客户端参数构造器按优先级产生 host、apiKey、workingDir 与 timeout。
4. TypeScript 客户端调用 Agent Server，服务端创建或恢复会话并运行具体 Agent。
5. 事件和状态返回 Canvas；自动化服务则可以从计划或 Webhook 独立派发会话。
6. 外部验证读取服务端产物与目标系统状态，而不是只读取 Canvas 的成功提示。

兼容 ACP 的 Agent 也要遵守这条责任链。协议兼容只表示客户端和 Agent 进程能够交换初始化信息、Prompt、工具和会话事件，不保证每个 Agent 都提供相同工具，也不保证它们采用相同的权限模型、Context 恢复方式或终止语义。因此，界面里出现 Claude Code、Codex、Gemini 等名称时，只能把它们看成可选 Adapter 表面。实际能力仍要结合各自锁定的版本和后端配置验证。

它补充了六条主课程的什么

OpenHands Agent Canvas 作为扩展样本，为六条一级主线补上了「多后端控制平面」这个视角。Codex、Gemini CLI、Claude、pi 与 OpenCode 的一级文档，会沿各自可以核对的源码解释核心循环和产品表面，Canvas 则重点展示一个产品怎样把多个异构 Agent 放进统一入口。它不会因此变成第七条全面主线。

这个视角也能帮你看清 Eval 横跨哪些模块。Canvas 可以收集会话、事件和自动化运行历史，但这些记录只是一部分评测输入。独立 Eval Harness 必须固定任务、目标后端、有效配置、Agent 版本和产物，并分清客户端报错、后端不可达、Agent 产品失败和 Scorer 失败。控制中心显示绿色，代替不了独立的发布 Holdout（未用于训练和候选选择的隔离任务集）。

什么时候值得采用

当团队需要统一管理身份、配置和运行 History 时，Agent Canvas 很适合组织多个异构后端，也方便大家从同一入口切换会话、观察自动化运行。控制平面在这里有独立价值，不过对于单一离线 Agent 或完全嵌入式的库，这层服务化可能只会增加成本。不能因为它们没有 Canvas，就判定能力不足。

这个样本只能证明两件事：锁定仓库承担哪些控制平面职责，客户端怎样决定参数优先级。后端名称、下拉框或 server info 都不足以证明 Agent、工具、隔离、认证和生产可用性。你还得检查 software-agent-sdk、真实部署和受控任务，比较多个后端前也必须先固定版本和 Environment。界面一致，语义未必一致。

最容易误判的地方

第一类误判来自仓库身份变化。如果不读当前 README 的责任表，你可能会把 Canvas 发起 API 调用、选择后端的动作，误写成 OpenHands Agent 的内部决策。第二类误判来自部署推断。看到后端类型写着 Docker、VM、remote 或 cloud，就以为对应实例正在运行，而且隔离配置符合安全要求。其实，这些字段只记录了登记或选择信息。

第三类误判，是客户端显示成功后，就认定任务已经成功。HTTP 请求返回、会话创建、Stream（流）结束，或者自动化显示完成时，任务产物仍可能是错的。各后端的版本、认证、工作目录和工具能力也不会天然一致。切换后端后，模型输入、权限请求和可见工具都可能改变。因此，实验记录必须包含最终生效的参数。

怎样亲手核对

验证控制平面，至少要做三类实验。无后端实验用来确认没有地址时系统会明确失败。覆盖优先级实验分别为登记后端和会话设置不同地址与密钥，再检查客户端最终选了什么。真实后端实验则要调用 server info，创建受控会话并验证产物。若还研究自动化，需要另外固定触发事件、派发记录、会话 ID 和下游结果。

安全验证必须检查真实部署，你要记录进程或容器身份、挂载、网络 Policy、工作目录怎样解析，以及凭证在哪个作用域生效。Canvas 下拉框里的文字不能充当证据。遇到基础设施错误，可以重新派发，但要保留旧记录。如果 Agent 给出了错误结果，就不能重新派发后只展示一次成功，把前面的错误盖掉。

读完后做四个判断

问题 1

当前仓库是否包含完整 OpenHands Agent Loop？

答案： 不能这样声称。锁定仓库明确把 Agent 与规范服务端实现归到 software-agent-sdk。

问题 2

选择 Docker 后端是否证明容器隔离？

答案： 不证明。必须检查真实进程、容器配置、挂载和网络边界。

问题 3

Canvas 会话结束是否就是任务通过？

答案： 不是。结束属于客户端和服务端生命周期，任务结果仍由冻结契约和独立评分器判定。

问题 4

为何仍值得收录？

答案： 它提供了多后端控制平面、仓库分责和自动化派发边界，是六条一级主线之外的重要机制样本。

Qwen Code：Serve 回放与批准作用域

为什么值得单独看

Qwen Code 和 Gemini CLI 有一些容易辨认的共同结构与概念。不过，当前锁定的仓库已经加入了规模可观的 Serve、Daemon、SDK 和多 ClientSession 能力，也实现了基于 ACP（Agent 客户端协议）的 Bridge。这里不考证历史谱系，也不会因为目录相似就认定行为相同。我们只看当前源码能直接核对的两套机制：系统怎样按 Turn（回合）压缩 Event 回放，以及 Serve Bridge（服务桥接层）怎样限制高风险 Approval 模式。

Agent Harness（智能体框架）一旦变成长驻 Server，又同时服务多个客户端，就会遇到单进程 CLI 里不太显眼的问题。Stream（流）里的事件很多，系统必须在内存占用和恢复能力之间取舍。远端 Tool 还可能改变会话的审批模式，因此代码必须限制这种变化影响哪个作用域，以及能否持久保存。这两套机制补充了六条一级主线里的协议与会话视角，但不会把 Qwen Code 变成新的综合主线。

轮次回放压缩与批准作用域

TurnBoundaryCompactionEngine 遇到 turn_complete 或 turn_error 时，会折叠当前 Turn 积累的事件。它合并连续文本和思考 Chunk（数据块），把 Tool Call（工具调用）序列收拢到最终状态，丢掉瞬态信号，同时保留不同事件类型原有的相对顺序。处理之后，回放日志只会随着对话轮数增长，不再随着每个流式 Token 一起膨胀。

Compression（压缩）能减少内存占用和回放成本，也会让证据变粗。系统一旦丢掉瞬态信号，就无法再从压缩日志里复原，工具调用只留下归约后的状态，中间怎样变化也可能看不到。源码还限制了回放字节数、事件数、Journal 增长量和截断锚点，所以长会话的窗口仍会淘汰旧记录。旧记录会消失。这是有损取舍。压缩日志适合帮助客户端恢复和展示界面，却替代不了高保真的审计 Trace。

Serve Bridge 还提供了一个工具，让调用方改变会话的审批模式，它接受 plan、default、auto-edit、auto 和 yolo 五个值。不过，只要 allowGlobalScope 没有开启，代码就会明确拒绝 auto-edit、auto、yolo，也不允许把变化持久保存。只有调用方明确开放全局作用域，请求才会交给 DaemonClient。

这道门禁只管远端 Bridge 表面怎样扩大风险，它可以阻止普通桥接调用悄悄把会话切到更自动的执行 Mode，也能阻止调用方把变化写入工作区 Settings，继续影响后面的会话。但它不构成完整的安全边界，直接 CLI、其他 API 表面、工具本身和 Sandbox 各有自己的 Policy。一旦开放全局作用域，还要记录谁发起调用、改了哪个会话，以及变化会保存多久、影响多大范围。

从哪里开始读源码

先读 TurnBoundaryCompactionEngine，看类说明怎样解释预算字段、回放片段和截断锚点。再跟到 ingest、轮次结束和 Snapshot，记下代码合并了哪些事件、丢掉了哪些事件、何时生成历史截断标记，以及客户端怎样取得先前记录。

审批作用域要从 workspaceWrite 的模式变更工具 读起。这几层不能混。注意分清四件事：枚举允许写哪些值，默认作用域放行哪些值，变化能否持久保存，最终改的是哪个会话 ID。工具描述列出了某种模式，不代表默认门禁一定放行。

把实时事件、回放窗口和会话控制连起来

实时 Journal 在 Turn 完成或出错时，才会产出回放片段。引擎先在当前 Turn 内接收增量事件，按类型合并文本和思考，收拢工具状态，并保留不同类型之间的顺序，然后才把片段放进受字节和事件预算约束的窗口。系统要记录压缩前后的数量、丢弃的类型、截断锚点和当前 Cursor，客户端看到这些信息，才能知道自己恢复的是有损投影，不是完整 Trace。

多个客户端通过「快照加增量」读取同一个回放窗口，客户端先取得窗口快照和锚点，再订阅新事件。断线重连时，它用 Cursor 去掉重复事件，如果要续读的位置已经被淘汰，系统就应返回明确的截断状态。原始证据要另存。高风险工具的原始调用、审批和 Result 应写进另一条不可变审计流，不能只依赖为 UI 优化的压缩 Journal。

allowGlobalScope 守在 Serve Bridge 通往会话控制 API 的入口，代码先校验模式、是否想持久保存，以及请求的作用域，默认只允许影响较小的变化。即使开放全局作用域，记录里仍要绑定调用者、目标 Session、旧值、新值和有效期。模式变化只会改动应用的审批策略，Executor 和操作系统隔离仍要各自判断。

沿一次长驻会话走完整链

1. Agent 运行产生文本、思考、工具调用、权限与轮次终止等流式事件。
2. ACP Bridge 在轮次进行中维护实时 Journal，并在完成或错误边界执行折叠。
3. 压缩片段进入受字节与事件预算约束的回放窗口，必要时保留截断锚点。
4. 终端、网页、协议客户端或 SDK 按会话订阅、重连并读取回放状态。
5. 若客户端通过 Serve Bridge 改批准模式，先经过全局作用域门禁，再调用会话控制 API。
6. 独立评测读取真实产物和所需原始证据，不把回放完成或批准变更当作任务通过。

要追清一段运行，记录里应当同时包含 session ID、事件 ID、轮次边界、压缩前后的数量、截断计数、记录锚点、客户端订阅位置，以及审批模式怎样变化。若 Task 风险较高，还要另外保存不可修改的原始工具调用与结果。客户端回放日志原本就有意做了有损优化。

它补充了六条主课程的什么

Gemini CLI 一级主线只根据锁定的 Gemini 源码，解释它怎样处理 Config、调度、工具和会话。Qwen Code 样本则根据自己的锁定 Commit 解释 Serve 扩展。两者可以沿同一组维度比较，但包名、目录或早期关系都不能让一边替另一边作证。所谓「同源」只能帮助定位。判断行为时，仍要回到各自源码和测试。

这个样本也给其他主线提出两个通用问题。长驻 Harness 同时向多个客户端提供会话时，怎样限制回放范围？Permission 模式能否通过 Protocol 表面从远端扩大？Codex、Claude、pi、OpenCode 和 DeepSeek Harness 都要按自己的实现回答，不能直接套用 Qwen Code 的字段或默认值。

什么时候值得采用

如果事件量远大于轮数，客户端又要快速重连，而 UI 不要求逐 Token 审计，那么在轮次边界做压缩很合适。可是一旦要调试流式解析、做合规取证，或者操作高风险工具，就必须同时保留保真度更高的证据。窗口预算越小，恢复成本越高，看不到的历史也越多。因此，预算要根据任务风险来定。

Bridge 的作用域门禁可以阻止普通远端工具扩大自动审批范围，也能拦住它们持久修改设置，但只能保护这一个入口。直接 CLI、其他 Agent SDK、配置文件和组织策略都要分别核对。这里不会因为 Qwen Code 与 Gemini CLI 结构相似，就推断两者语义相同，也不评价 Qwen Code 的全部产品表面。真正可以借鉴的，是明确告诉客户端回放有损，用快照和 Cursor 续读，并审计权限怎样扩大。

最容易误判的地方

第一类风险，是看到压缩回放保留了相对顺序，就把它当成完整审计。中间事件仍可能被丢弃，截断窗口还会淘汰旧轮次。第二类风险，是把 Bridge 的默认门禁当成全系统策略。它只管 Daemon MCP Serve Bridge 这条工具路径，其他入口仍要逐个检查。

第三类风险，是因为两个仓库同源，就认定行为等价。即使它们仍有共同结构，独立演进也会改动配置优先级、工具策略、协议事件和 Resume 语义。Eval 若混用两套 Harness 的默认值，结果便无法复现。审批模式也只在应用层控制行为，替代不了容器、进程权限、网络和凭证隔离。

怎样亲手核对

做回放实验时，可以先构造一个 Turn，放入大量文本分片、思考分片和工具状态变化。然后比较完成边界前后的事件数量与顺序，确认同类片段已经合并，工具状态已经收拢，不同类型之间的相对顺序仍然保留。随后调低窗口预算，主动触发截断，再检查客户端是否收到明确标记和分页锚点。

审批实验先保持默认的 allowGlobalScope=false，分别请求 plan、auto-edit、auto、yolo 和持久化 default，确认高风险模式与持久化请求都遭到拒绝。再明确开放全局作用域，检查请求是否真的送到了指定会话。开放门禁不代表操作已经安全获批。实验仍要使用受控工具，并放在隔离环境里运行。

读完后做四个判断

问题 1

回放压缩是否无损？

答案： 不是。它合并片段、归约工具状态并丢弃瞬态信号。

问题 2

默认 Bridge 能否切到 yolo？

答案： 不能，除非显式开放全局作用域。

问题 3

该门禁是否覆盖所有 Qwen Code 入口？

答案： 不覆盖，本文结论只描述 Serve Bridge 工具路径。

问题 4

与 Gemini CLI 结构相似是否说明行为一致？

答案： 不说明，必须分别锁定源码、配置与运行证据。

术语表：同一个词在六套 Harness 中可能不是同一对象

返回学习入口

这份词表先说明一个术语在不同项目间共享的大致含义，再提醒你各项目可能怎样使用它。它只是入口。遇到具体行为时，仍要回到对应课程和锁定源码逐项核对。

完整术语对照

正文保留英文术语，因为它们要和源码里的标识符对得上。这张表给出每个词的中文译名和 它在本仓库里的具体所指，正文每篇第一次出现某个术语时也会在括号里带上译名。 下面按主题分的几节讲的是同一个词在六套实现之间的差异，那是另一层问题。

英文	中文译名	在本仓库中的含义
ACP	Agent 客户端协议	连接编辑器等客户端与 Agent 进程的开放协议，规定会话、消息、工具和权限请求的交互方式。
ACP/A2A/RPC	外部客户端或 Agent 间协议	协议投影不等于完整内部事件
AbortSignal	中止信号	在模型、工具和压缩链路中传播取消请求的标准信号，接收方仍需在安全边界响应它。
Adapter	适配器	把某个模型、协议、工具或存储实现转换成 Harness 统一接口的组件。
Agent	智能体	接收目标并通过模型、工具和状态管理持续推进任务的执行主体，不等同于单次模型调用。
Agent Core	Agent 核心	承担模型调用、工具执行、状态更新和循环控制的核心模块。
Agent Definition	智能体定义	描述可被主 Agent 委派的子 Agent 配置，通常包含说明、Prompt、工具限制和模型选择。
Agent Harness	智能体框架	位于模型之外，负责输入装配、循环、工具、权限、状态和产品表面的运行程序。
Agent Loop	智能体循环	在模型推理、工具调用、结果回填和结束判断之间反复运行的核心控制流程。
Agent Run	Agent 运行	Agent 从接收输入到控制循环收敛的一次运行，pi 用它覆盖内外两层循环，Cline 还将根 Agent Run 作为 Checkpoint 边界。
Agent SDK	智能体开发工具包	供应用创建会话、发送消息、接收事件和配置工具或权限的开发接口，不等同于闭源产品内部实现。
Agent Scope	智能体作用域	DeepSeek Harness 中限定当前 Agent 的依赖解析、工具可见性和局部限制，作用域内定义可以遮蔽全局定义。

英文	中文译名	在本仓库中的含义
Agent Server	智能体服务器	OpenHands 中实际创建或恢复会话并运行 Agent 的后端服务，与负责界面和后端选择的 Canvas 分开。
Agent Session	智能体会话	承载智能体运行状态、输入队列和历史的会话对象；Gemini CLI 的 Legacy Agent Session 与当前 Turn/Scheduler 主链不是同一代实现。
Agents	Agent 集合	系统中已配置、注册或发现的多个 Agent，包括主 Agent、子 Agent 或专用角色。
Allow	允许	Permission 或 Policy 对当前工具动作给出的放行决定，只代表应用层许可，不等于操作已在沙箱中安全执行。
App Server	应用服务器	Codex 把核心 Thread、Turn 和 Event 暴露给 JSON-RPC 等多种传输入口的服务层。
Approval	审批	用户或策略对某次受限工具调用、命令执行或资源访问作出的允许决定。
Approval Policy	审批策略	决定遇到额外权限请求时能否询问用户，它不负责选择沙箱边界，也不等于请求已经获准。
Approval/Confirmation	用户或宿主对一次请求的决定	不证明用户理解全部副作用
Architect	架构师	负责分析代码结构并先产出实现方案的 Agent 角色，不等同于软件架构师职级。
Artifact	产物	Agent 执行后留下的可交付对象，例如代码文件、补丁、报告、日志或评测记录。
Ask	询问	Permission 或 Approval Policy 暂停当前动作并等待用户或 Answerer 决定的分支，其默认行为和授权范围随项目而异。
Assistant	助手	消息协议中代表模型输出的一方，不等同于包含工具和运行时的完整 Agent。
Assistant Message	助手消息	角色为 Assistant 的具体消息对象，可承载文本、工具调用或其他结构化内容。
Attach	附加	把客户端连接到已有运行边界的操作，pi 用它取得 Session 占用权，OpenCode 的远程 Attach 使用服务端目录。
Attempt	尝试	一个 Trial 内的一次运行尝试，只有未产生业务副作用的可恢复基础设施故障才适合自动重试，并且每次记录都要保留。
Bash	Bash 命令行	Harness 调用 shell 命令时常用的命令解释器，也是部分工具名称或执行环境。
Builder	构建者	教材中追踪配置或状态变量如何改变运行行为的进阶读者层级。

英文	中文译名	在本仓库中的含义
Bundle	打包包	把运行所需的代码、配置或资源组合成一个可加载或分发单元，不同项目的 Bundle 内容不同。
CHANGELOG	变更日志	Claude TypeScript Agent SDK 课程用它核对版本声称新增或修复的行为，但不能据此推断未公开的运行时实现。
Cache	缓存	教材主要指 Provider 对稳定 Prompt 前缀的复用，不表示 Harness 在本地保存了一份模型 KV Cache。
Call	调用	模块向模型、工具或服务发起的一次请求，正文会根据调用对象区分 Model Call 和 Tool Call。
Canvas	画布	用于展示、编辑或承载 Agent 生成内容的可视区域，不同项目可能对应文档、代码或交互组件。
Checkpoint	检查点	在任务执行过程中保存的可恢复状态节点，可包含会话、工作区或消息状态。
Chunk	数据块	流式处理中分批到达的数据块，在模型层是增量响应，在协议传输层也可能只是任意大小的字节段。
Client	客户端	发起请求并消费 Agent、模型或工具服务响应的一侧，可能是 SDK、CLI 或图形界面。
Code Mode	代码模式	让模型生成并执行代码来编排多个工具或处理数据的运行方式，而不是逐个直接发起工具调用。
Coding Agent	编程智能体	面向代码仓库执行阅读、修改、测试和调试任务的 Agent，通常具备终端与文件工具。
Command	命令	在 pi Protocol 中是等待同步 Response 的控制请求，在 pi Extension 中也可指注册命令，而 Codex Hook 中则是程序配置。
Compaction	上下文压缩	上下文接近容量限制时，把较早内容整理成摘要或精简状态以便继续执行。
Companion	伴随组件	DeepSeek Harness 中由各包提供的动态检查组件，负责安装该包特有的事件监听器和约束检查。
Compression	压缩	用摘要和保留尾部缩短模型可见历史的过程；Gemini CLI、pi、Codex 和 OpenCode 的持久化方式不同。
Config	配置	控制模型、工具、权限、路径和运行行为的配置对象或配置文件。
Confirmation	确认	Agent 继续执行前向用户发起的是非确认，通常用于核对意图或高影响操作。

英文	中文译名	在本仓库中的含义
Context	上下文	当前模型请求能够看到的消息、工具结果、指令和运行状态，不同项目的拼装与截断规则不同。
Context/Surface	下一次模型实际可见的有界历史投影	不是完整审计记录
Core	核心层	各项目中承载主要状态与控制循环的核心模块，Gemini CLI、pi、Codex 和 Cline 对其边界与职责划分不同。
Core Config	核心配置	Gemini CLI 在 CLI Settings 合并后接手系统指令、项目 Memory、工具声明和 Gemini Client 初始化的核心配置对象。
Cursor	游标	标记分页或事件流消费位置的续读凭据，重连时用于恢复增量并处理重复或过期事件。
Deny	拒绝	对权限请求或受限操作作出的不允许决定，Agent 应停止该操作或选择其他路径。
Desktop	桌面端	通过服务协议访问共享 Agent 核心的桌面应用表面，其窗口和选中状态不属于服务端 Session。
Diff	差异	文件修改前后的逐行对比，用来审查 Agent 实际新增、删除和替换了什么。
Directory	目录	OpenCode 用它建立 Project/Instance 并划分配置、事件和会话，远程 Attach 时它指服务端路径。
Driver	驱动器	DeepSeek Harness 唤醒 Agent、推进 Turn 并把运行状态收敛到 Idle 或错误终态的控制层。
Editor	编辑器	用户查看和修改代码的界面，也是部分 Harness 接收选区、文件状态和诊断信息的入口。
Entry	条目	表示集合中的一个记录；在 DeepSeek Harness 中可指 Cordis 挂载项，在 pi 中可指会话树节点。
Environment	运行环境	Agent 执行时可见的工作目录、系统变量、运行时、依赖和权限边界。
Error	错误	记录模型、工具、权限、协议或基础设施失败的状态，不同来源必须保留各自阶段和语义。
Escalation	经策略允许后，以更宽权限重新尝试	不应被理解为永久授权
Eval	评测	用固定任务、运行轨迹和判定规则检查 Agent 是否完成目标，而不只比较模型回答文本。
Eval Harness	评测框架	在受控任务、配置和工作区中运行 Agent 并保存 Artifact 的框架，最终分数仍由外部 Scorer 产生。

英文	中文译名	在本仓库中的含义
Evaluator	评估器	根据预期结果、轨迹或评分规则判断一次 Agent 运行表现的组件。
Event	事件	Agent 运行过程中发出的结构化通知，用来驱动界面更新、日志记录或后续处理。
Event Log	事件日志	DeepSeek Harness 用它追加保存 Turn、Step、消息、工具、审批、压缩和错误，模型看到的 Surface 只是其有界投影。
Event Log/Rollout	追加式运行记录	比模型 Context 包含更多事实
Events	事件	运行过程中产生的状态变化记录；不同项目的核心事件、持久化事件和对外协议事件范围不同。
Exec Policy	执行策略	Codex 对命令请求给出允许、询问或禁止判断的策略层，与用户审批和操作系统沙箱分开。
Executor	执行器	Gemini CLI 的 Tool Executor 归约工具终态，Hook Executor 和 Local Executor 则分别执行 Hook 计划与本地子会话。
Extension	扩展	通过公开接口给 Harness 增加能力的模块，不同项目可能把工具、事件处理器或命令都称为 Extension。
Extensions	扩展	为 Harness 增加工具、Hook、Skill、Prompt 或产品能力的扩展机制，各项目的加载和信任边界不同。
Feedback	反馈	与消息或 Session 身份关联的用户评价信号，可以进入记录或训练管道，但不是任务正确答案。
FinishReason	结束原因	Provider 对当前一次模型响应为何停止的标记，各家枚举和映射不同，也不能单独判定整个任务已经结束。
Follow	跟随	让读取或展示持续追随新产生的消息、事件或日志，具体跟随对象随项目而异。
Fork	从已有历史建立新分支身份	后续写入不应污染原会话
Frame	帧	pi 协议中带 Header 和 Payload 的完整字节单元，传输层要先从 Byte Chunk 中拼出它，再进行 CBOR 解码和 Schema 校验。
Function Call	函数调用	模型响应中带名称、参数和调用标识的请求，Harness 会将它路由到工具处理器并按原标识写回结果。
Future	异步计算	Codex 用它表示尚未完成的异步工作，既可控制 RunningTask，也可让工具并发推进后按原顺序写回结果。

英文	中文译名	在本仓库中的含义
GEMINI	Gemini 项目指令文件	正文中的 GEMINI 指 GEMINI.md，它保存稳定的项目命令、目录约定和长期偏好，并在工作区受信且读取成功后进入新会话。
Handler	处理器	接收某类命令、消息、工具调用或事件并执行对应逻辑的入口函数或对象。
Harness	位于模型外部，负责输入装配、工具、权限、状态和产品表面的程序	不是模型本身，也不等于 Eval Runner
Header	头部	DeepSeek Harness 持久化影响请求前缀和恢复边界的配置快照，不等同于普通 HTTP 请求头。
Headless	无头模式	不启动交互界面的运行方式，DeepSeek Harness 用标准输出和退出码交付结果，并要求预先确定无人值守时的审批策略。
History	历史记录	会话中过去的消息和事件集合，用于恢复界面、构造上下文或继续运行。
Holdout	未用于训练和候选选择的隔离任务集	用于最终发布判断，防信息泄漏
Hook	钩子	挂在模型调用、工具执行或状态变化等节点上的扩展入口，用来观察、拦截或修改流程。
Hooks	钩子集合	一个项目公开的多个 Hook 入口及其处理函数，用来共同介入 Agent 的不同运行阶段。
Host	宿主	装载插件、注册全局能力并实际启动进程或服务的运行环境，与单个 Agent 的可见能力范围分开。
IDE	集成开发环境	承载代码编辑、诊断和 Agent 交互的开发工具，如 VS Code 或 JetBrains IDE。
Idle	空闲	表示会话当前没有继续推进的工作，不表示用户目标已经通过验证。
Inbox	收件箱	DeepSeek Harness 接纳外部输入和工具结果的队列，并区分下一 Step 与下一 Turn 的投递边界。
Interrupt	中断	向活动运行发出取消信号；Codex 和 Claude 的具体入口不同，但都不会因此删除会话身份和已提交历史。
Invocation	调用实例	Gemini CLI 将模型原始工具参数规范化并校验后生成的可执行调用对象。
Item	条目	Codex 中既可指模型流里的响应条目，也可指 Rollout 的持久记录单元，后者还会保存上下文、压缩和控制事件。
JSONL	逐行 JSON	每行保存一个独立 JSON 对象的格式，常用于记录事件流、会话和执行轨迹。
LLM	大语言模型	Harness 实际构造请求并接收文本、工具调用和停止原因的模型服务。

英文	中文译名	在本仓库中的含义
LSP	语言服务器协议	OpenCode 通过它查询诊断、定义、引用和符号，只有语言服务器存在、项目根识别成功且文件语言匹配时才可用。
Legacy Agent	旧版智能体	Gemini CLI 中另一条可核对的旧版 Agent Session 路径，不能与当前 Turn/Scheduler 主链拼成同一套运行流程。
LocalEnvironment	本地环境	mini-swe-agent 中直接在宿主机工作目录执行 Bash、继承进程环境并回传观察的实现，不提供容器或系统调用隔离。
Loop	循环	重复执行一组步骤直到满足退出条件的控制结构，正文中既可能指 Agent Loop，也可能指事件循环。
Maintainer	维护者	教材中核对失败路径、证据边界和结论能否持续复现的维护者层级。
Manager	管理器	Context、Sandbox、MCP Client、Session 和 Skill Manager 分别管理不同状态或资源边界，不能把它们当成同一个模块。
Map	映射	源码中通常指键值映射，教材图示中的 Map 则指帮助定位模块和调用链的系统地图。
Memory	记忆	跨消息或跨会话保留并可再次取用的信息，可能存成摘要、文件、向量记录或结构化状态。
Message	消息	在用户、Assistant、模型和工具之间传递的结构化交互单元，各项目的字段和角色定义不同。
Messages	消息列表	按顺序保存并提交给模型或界面的一组 Message，通常构成当前对话上下文。
Mode	模式	决定 Agent 当前工作方式的一组行为规则，如规划、编辑或只读模式。
Model	模型	Agent Loop 实际调用的推理模型，其能力、上下文窗口和工具协议由 Provider 适配。
Model Router	模型路由器	Gemini CLI 为当前请求选择具体模型并记录回退原因的组件，不负责工具调度。
Options	选项	创建会话、启动运行或调用工具时传入的一组可选参数。
Part	消息片段	一条 Message 内的结构化内容单元，可表示文本、推理、工具调用或工具结果。
Parts	内容片段	组成消息或模型内容的结构化片段，OpenCode 将其持久化为 Message Parts，Gemini CLI 用它表示 Provider Content 的组成项。
Patch	补丁	Agent 对现有文件提出或应用的一组局部改动，通常以统一 Diff 或结构化编辑操作表示。

英文	中文译名	在本仓库中的含义
Pattern	匹配模式	权限规则用它匹配规范化后的路径、命令或工具调用，OpenCode 由最后命中的规则决定结果。
Permission	权限	规定 Agent、工具或命令可以访问哪些资源以及允许执行哪些操作。
Permission/Policy	应用是否允许、拒绝或询问一次动作	不自动提供 OS 隔离
Persona	角色设定	DeepSeek Harness 的 Agent Preset 用它装入角色和行为规则，创建子 Agent 时还可随继承配置单独指定。
Plugin	插件	按项目约定打包并安装的扩展单元，通常可以同时注册工具、命令、Hook 或 Provider。
Plugin/Extension	可贡献工具、Provider、Hook 等的能力容器	通常是高信任宿主代码
Plugins	插件	由宿主加载并可贡献工具、Provider、Hook 或其他运行能力的高信任代码扩展。
Policy	策略	控制模型选择、权限审批、工具使用或执行终止等决策的规则集合。
Preset	预设	一组可直接选用的模型、工具、权限或运行参数组合。
Processor	处理器	对消息、模型输出或事件流进行解析、转换和分发的组件。
Profile	配置档	一组具名的模型、提供商和运行设置；部分项目中也可能指用户资料。
Project	项目	一次 Agent 工作所关联的代码库、目录和项目级配置上下文。
Prompt	提示词	实际送入模型的指令与上下文组合，可能由系统规则、用户消息、工具说明和运行时状态共同组装。
Prompt Loop	提示词循环	OpenCode 读取有效历史、驱动模型 Step，并根据工具结果决定继续、压缩或停止的会话主循环。
Protocol	协议	约定客户端、Agent 进程、模型和工具之间如何组织请求、事件与响应。
Provider	模型提供商	把统一的模型调用接口接到 OpenAI、Anthropic、Gemini 等具体模型服务的适配层。
Pruning	删除或缩短某个大结果/片段	不一定重写整段历史
Query	查询	交给模型、搜索组件或状态存储执行的一次检索请求，具体含义随调用对象变化。
RPC	远程过程调用	外部客户端通过 Command、Response 和 Event 双向驱动 Agent Session 的协议方式。

英文	中文译名	在本仓库中的含义
Read	文件读取	多套 Harness 中用于读取文件的工具名，它是否可见、是否免审批以及能否并行要按各项目配置判断。
Recipe	配方	把提示词、工具、模型和执行步骤组合成可复用任务流程的配置或示例。
Registry	注册表	按名称登记并查找工具、Provider、Skill 或扩展实现的集中索引。
Repository Map	仓库映射	Aider 按符号、引用和 Token 预算生成的仓库上下文投影，不是完整源码，也不保证包含每项依赖。
Response	响应	一次模型或 API 调用返回的完整结果，可包含文本、推理、工具调用和用量信息。
Result	结果	某次调用或任务完成后返回给上层的输出，通常同时携带状态、数据或错误信息。
Resume	恢复	从已保存的会话或运行状态继续执行，而不是重新创建一段独立对话。
Revert	还原	把工作区或会话恢复到先前状态，不一定等同于执行 Git 回退。
Revision	修订版	内容或状态经过一次修改后形成的版本，在不同项目中可能对应消息、文件或共享内容的版本。
Reward Adapter	把原始信号转换成训练奖励的版本化规则	必须定义缺失、方向、范围和聚合
Rollout	运行轨迹	Agent 针对一个任务从开始到结束的一次完整尝试，评测系统常对同一任务采样多条 Rollout。
Router	路由器	根据请求或标识选择目标处理器的控制组件；Tool Router 分派工具，Model Router 选择模型。
Run	运行	Agent 从接收任务到完成、失败或取消的一次独立执行实例。
Runtime	运行时	实际执行 Agent 循环、工具和扩展代码的环境，不同项目可能是 Node.js、Python 或 Rust 进程。
Sandbox	沙箱	限制命令、文件和网络访问范围的执行环境，各项目的隔离强度和授权方式并不相同。
Scheduler	调度器	决定任务、步骤或并发工具何时开始、暂停和继续执行的组件。
Scheduler/Router	查找、排序和执行 Tool Calls 的控制层	可见性、Permission、Sandbox 通常在不同层
Schema	模式	约束消息、工具参数或配置数据结构的规则，常由 JSON Schema 或项目内类型定义表达。

英文	中文译名	在本仓库中的含义
Score	评分	独立 Evaluator 根据冻结测试、文件差异和任务约束给出的结果，而不是 Harness 的成功状态。
Scorer	评分器	在 Agent 运行之外读取 Trace、文件、测试和环境事实，并按固定规则产出判断或分数的组件。
Scorer/Evaluator	按固定规则读取 Artifact 并给判断	应位于 Harness 自述之外
Section	区段	DeepSeek Harness 组装 System Prompt 时使用的有序内容单元，如 Persona、工具说明和项目指令。
Serve Bridge	服务桥接层	Qwen Code 中把 Serve 工具请求接到 Daemon 会话控制 API，并用 allowGlobalScope 限制扩大自动批准和持久化设置。
Server	服务端	对外提供模型、工具或 Agent 能力的进程，在不同项目里可能指 MCP Server、HTTP Server 或后台守护进程。
Session	会话	一段可恢复的交互状态；有的项目指 CLI 进程生命周期，有的项目指跨多次启动保存的对话。
Session Event	会话事件	DeepSeek Harness 追加保存 Turn、Step、消息、工具调用、结果和结束原因的耐久事件。
Session Idle	会话空闲	Session 当前没有控制循环继续工作的状态，OpenCode 用它表示运行结算，DeepSeek Harness 还将它作为手动压缩的前提。
Session Prompt	会话提示词	创建会话时组装并注入的指令文本，用来设定角色、规则、工具和项目上下文。
Session Store	会话存储	基础章节用它泛指保存和恢复会话事件的文件、数据库或接口，在 Claude TypeScript SDK 中则特指宿主可实现的扩展契约。
Session Tree	会话树	pi 用 Entry 组成的完整会话分支结构，可用于回看、压缩、恢复和从旧节点分叉。
SessionStore	会话存储	负责保存、读取、列出和恢复 Session 数据的存储组件。
Settings	设置	用户可调整并持久化的应用级或项目级行为选项。
Share	共享	把会话、运行结果或生成内容导出或发布给其他人查看。
Skill	技能	可按需加载的一组任务说明、脚本和参考资料，用来教 Agent 完成某类具体工作。

英文	中文译名	在本仓库中的含义
Skills	技能集合	当前环境中可发现和加载的多项 Skill，通常通过目录、清单或注册表统一管理。
Snapshot	快照	某一时刻工作区、会话或配置状态的固定副本，用于比较、恢复或复现。
Snapshot/Patch	可用于恢复工作树的一组文件事实	无法撤销网络和数据库副作用
Starter	入门者	教材中先沿完整任务链认识请求、模型流、工具执行和结果回填的入门读者层级。
Steer	中途引导	向正在运行的任务补充约束或改变方向，DeepSeek Harness 将它送到下一 Step，Codex 与 pi 的接入边界则不同。
Steering	转向	pi 在运行过程中插入的新指令队列，使当前工具或模型循环从下一合适边界调整方向。
Step	步骤	一次可单独记录和判定的执行推进，可能对应模型请求、工具调用或工作流节点。
Store	存储	保存会话、消息、记忆或运行状态并向上层提供读写接口的组件，不限定具体数据库实现。
Stream	流	模型或服务持续产生文本、工具请求、状态和错误事件的异步输出序列。
Subagent	拥有独立上下文/Session 的子运行	父端摘要会丢子轨迹细节
Success	成功	表示某个工具调用或局部步骤正常结算，不表示整个会话目标已经完成。
Summary	摘要	压缩后替代一段模型可见历史的有损内容，不能替代完整事件记录或外部状态核对。
Surface	接口面	一个模块向用户或其他模块暴露的命令、API 和可观察行为范围。
System	系统	在模型请求中指 System Prompt 等系统级输入，在 Hook System 等名称中则指一套运行时子系统。
System Prompt	系统提示词	每次模型请求中装入角色、规则、项目指令和工具用法的系统级提示内容。
TUI	终端用户界面	直接运行在终端中的交互界面，用于输入任务、查看流式输出和处理审批。
Task	任务	Agent 要完成的目标单元，可能来自用户请求，也可能由工作流拆分或调度器生成。
Task Tool	任务工具	OpenCode 中由父模型调用、检查深度与权限后创建或复用子 Session，并把子任务结果投影回父会话的工具。
Telemetry	遥测	Harness 为观察运行情况而采集的耗时、错误、Token 用量和执行事件等数据。

英文	中文译名	在本仓库中的含义
Thread	线程	一条可持续追加消息的对话分支；部分项目用它表示会话对象，并非操作系统线程。
ThreadId	线程 ID	Codex 用来持久定位 Thread 的稳定句柄，应用不应拿某个 JSONL 文件路径替代它。
Token	词元	模型计量文本、上下文和用量的基本单位；在认证代码中也可能指访问令牌。
Tool	工具	Agent 可调用的外部能力，例如执行命令、读写文件、搜索内容或访问服务。
Tool Call	工具调用	模型请求运行某个工具并提供结构化参数的动作，执行结果随后会回填到上下文。
Tool Calls	工具调用	模型在一次响应中提出的一批工具请求，执行并回填结果后还可能继续请求模型。
Tool Registry	工具注册表	汇集工具定义并投影本轮可见 Schema 的注册中心，工具已注册不代表当前模型一定可见。
Tool Result	工具结果	工具执行后返回给 Agent Loop 的结构化输出，可能包含文本、数据、错误或产物引用。
Tool Router	工具路由器	Codex 中把模型协议里的工具调用交给已注册 Handler 的分派层，审批、沙箱和工具语义由后续层处理。
Tool Schema	工具定义	用 JSON Schema 等结构描述工具名称、用途、参数和约束，供模型生成合法调用。
ToolCall	工具调用	源码类型或事件名中的单次工具请求，通常携带工具名、参数和 Call ID。
ToolRuntime	工具运行时	DeepSeek Harness 中负责解析当前作用域的工具定义、执行策略链、调度工具并规范化结果的运行层。
ToolUse	工具使用	Claude 类型化消息中的工具请求块，只说明模型提出了调用，不能证明权限已放行或工具已执行。
Tools	工具集合	当前 Agent 被允许发现和调用的全部 Tool，其范围还会受到配置、权限和沙箱限制。
Trace	执行轨迹	记录一次 Agent 运行中模型请求、工具调用、状态变化和结果的完整过程。
Transcript	对话记录	按时间顺序保存的用户消息、Assistant 输出和工具交互记录。
Transport	传输层	把 SDK 的请求送到 CLI 进程的那一层，默认实现在 connect() 才真正启动子进程。
Trial	评测试次	围绕同一业务任务、工作区基线、全部 Attempt、最终产物和独立评分组织的一次完整评测。
Trust	是否加载工作区提供的配置、指令或扩展	不是文件、网络和子进程 Sandbox

英文	中文译名	在本仓库中的含义
Turn	回合	一次逻辑用户交互，内部可能包含多次模型请求，不能按 HTTP 请求数来数。
Turn Context	回合上下文	Codex 为某个 Turn 保存的有效配置与运行语境快照，会进入任务、模型请求、压缩和 Rollout，不能用当前配置倒推。
Usage	用量	模型或工具上报的资源消耗记录，通常包含 Token、缓存命中、成本或耗时，各项目的字段并不相同。
Web	Web 端	通过服务协议访问 Session 的浏览器产品表面，只保存界面局部状态而不拥有会话真值。
Workflow	用脚本或控制器编排多个调用/子运行	资源上限、结果身份和取消传播

运行与身份

术语	本仓库中的含义	容易混淆之处
Harness	位于模型外部，负责输入装配、工具、权限、状态和产品表面的程序	不是模型本身，也不等于 Eval Runner
Session	一段可继续的运行状态与历史身份	OpenCode、Gemini CLI、Claude SDK 等字段范围不同
Thread	可持久引用或并行分支的会话身份	Codex 把 Thread 与活动 Session/Turn 分开
Turn	一次逻辑用户交互，可包含多次模型请求	不能按 HTTP 请求数量计 Turn
Step	一次模型采样及其工具结算附近的较小单位	不同项目是否公开 Step 类型并不一致
Task	驱动某类工作流或用户目标的对象	Codex Task、A2A Task、普通「任务」不是同一类型
Run	一次可结算执行实例	Subagent Run、Workflow Run、Eval Run 作用域不同

模型与工具

术语	含义	核对点
Provider	把共同请求协议适配到某个模型服务	目录发现、实现加载、认证和真实请求分开看
Model	Provider 下的模型标识与能力配置	保存实际路由结果，不只保存首选项
Tool Schema	模型可见的工具名、描述和参数约束	仓库中有实现不代表 Schema 已进入本轮请求
Tool Call	模型提出的一次带 Call ID 的工具请求	参数必须完整、规范化并与 Result 配对
Tool Result	工具协议结算后回给模型的观察	Success 不等于用户目标完成
Scheduler/Router	查找、排序和执行 Tool Calls 的控制层	可见性、Permission、Sandbox 通常在不同层

安全

术语	含义	不代表什么
Permission/Policy	应用是否允许、拒绝或询问一次动作	不自动提供 OS 隔离
Approval/Confirmation	用户或宿主对一次请求的决定	不证明用户理解全部副作用
Sandbox	操作系统、容器或远端环境对进程能力的强制约束	配置开启不等于后端成功执行
Trust	是否加载工作区提供的配置、指令或扩展	不是文件、网络和子进程 Sandbox
Escalation	经策略允许后，以更宽权限重新尝试	不应被理解为永久授权

历史与恢复

术语	含义	边界
Event Log/Rollout	追加式运行记录	比模型 Context 包含更多事实
Context/Surface	下一次模型实际可见的有界历史投影	不是完整审计记录
Compaction	用摘要替换一段模型可见历史	摘要有损，旧事实是否保留取决于记录层
Pruning	删除或缩短某个大结果/片段	不一定重写整段历史
Resume	继续原会话身份	不恢复外部世界到过去时刻
Fork	从已有历史建立新分支身份	后续写入不应污染原会话
Snapshot/Patch	可用于恢复工作树的一组文件事实	无法撤销网络和数据库副作用
Memory	跨 Session 提炼的长期信息	可能过时，不覆盖当前文件和新指令

扩展与产品表面

术语	含义	主要风险
Subagent	拥有独立上下文/Session 的子运行	父端摘要会丢子轨迹细节
Workflow	用脚本或控制器编排多个调用/子运行	资源上限、结果身份和取消传播
Skill	按需加载的任务说明与资源	发现不等于正文已注入
Hook	生命周期事件上的附加处理	是否能阻断取决于事件契约
Plugin/Extension	可贡献工具、Provider、Hook 等的能力容器	通常是高信任宿主代码
MCP	连接外部 Tools/Resources/Prompts 的协议	连接成功不代表每项能力存在或获准
ACP/A2A/RPC	外部客户端或 Agent 间协议	协议投影不等于完整内部事件

Eval 与训练

术语	含义	规则
Artifact	一次运行留下的大对象与环境事实	应带 Hash、版本和血缘
Trace	带顺序和关联身份的运行事件	没有 Error 不等于任务正确
Scorer/Evaluator	按固定规则读取 Artifact 并给判断	应位于 Harness 自述之外
Reward Adapter	把原始信号转换成训练奖励的版本化规则	必须定义缺失、方向、范围和聚合
Holdout	未用于训练和候选选择的隔离任务集	用于最终发布判断，防信息泄漏

进一步阅读：五篇基础导读 与 横向比较。

怎样从文章结论核回锁定源码

返回学习入口

每条源码课程都会链接到锁定 Commit 的 GitHub 文件，因此不同读者看到的是同一版代码和同一段行号。不过，链接只能锁住版本，机器也只能检查仓库、Commit 和路径是否存在，判断不了文章有没有把代码讲对。这一步不能交给机器。你复核时还得沿着调用链亲自回答下面五个问题。

源码站点五问

1. 谁调用这段函数，它在真实产品路径上吗？
2. 输入来自模型、配置、用户还是已验证的内部对象？
3. 它修改了 Session、工具、文件还是仅修改局部投影？
4. 返回值代表调用完成、步骤完成还是任务完成？
5. 下一站在哪里，错误、取消和清理走哪条分支？

文章里的每个「第 N 站」都会回答这五项，你可以据此检查正文有没有说得太满，超出源码实际能够支持的范围。

三类材料不要混用

材料	能支持	不能单独支持
Runtime 源码	该提交的实现和类型边界	运行时一定触发了这条路径
上游测试/Fixture	构造场景下的可执行行为	未覆盖平台、真实 Provider 和长期负载
官方公开文档/协议	对外承诺与用法	闭源内部如何实现

这三类材料不能混用。在这里，公开 Agent SDK 源码只能帮你核对 Transport（传输层）、Messages、Permissions 和 Hooks（钩子集合）等公开契约，不能拿来还原 Claude Code 内部完整的 Agent Loop。

从仓库根目录复核

```
npm run bootstrap
npm test
npm run check
```

bootstrap 按 Source Lock 准备对应上游提交；

test 运行本仓库脚本的单元测试；

check 检查公共内容、链接、来源、图示和其他仓库规则。

这些命令只检查教材自身，不会调用六套 Harness 的付费模型，所以运行时不用准备相关账号。不过，本仓库没有覆盖上游项目的全部运行条件，这些检查也就代替不了目标项目自己的完整测试。你若要核对某条关于运行行为的结论，还应该进入对应的锁定 Checkout，按官方要求运行最小相关测试，并记下 Node、Rust、Python 的版本以及跳过了哪些项目。

判断一句话是否过度外推

你可以试着把一句结论改写成「在锁定 Commit 的哪条路径里，给了什么输入，又观察到什么」，这样很快就能看出证据缺了哪一环。如果其中任何一项填不出来，就要收窄结论，只说现有材料确实覆盖的对象和条件。到了这一步，应明确标成未核对、仅在特定条件下成立、公开材料不可见，或仍需到真实环境中实验。

继续实践：受控本地核对。