CODEX 源码研究

读懂 Codex：从大模型到智能编程助手

连续阅读十章总览，先建立从模型、运行时、工具到恢复系统的完整主线。

主干学习版

版本日期：2026-08-04
Codex 源码提交：fe01054a28fa4bd04716d9ceadb410f2443a50ce
纸张：A4 · 离线图表 · 分章目录

主干目录

完整参考版另收录 285 个源码级专题单元。

第一章　会接话的机器，怎样开始做事

用户向聊天模型输入一句“资料页点击保存没有反应，请修一下”，模型可以列出常见原因，也可以给出排查步骤。但如果它看不到项目目录，不能读取日志，也不能修改文件，这些判断就只能停留在建议层面。

同一句话交给 Coding Agent，执行过程会不同。Coding Agent 是能够围绕编程目标读取项目、调用开发工具，并根据执行结果继续工作的系统。它可能先搜索资料页组件，再检查事件处理函数和网络请求；找到故障后修改代码、运行测试，并根据测试结果决定是否继续调整。用户给出的是目标，实际执行路径则在观察与行动的往复过程中逐步形成。

这项能力不是语言模型单独提供的。它来自模型、上下文、工具和运行时的协作。理解 Codex 的第一步，就是把模型负责的“生成下一步意图”与外层系统负责的“执行并反馈结果”分开。

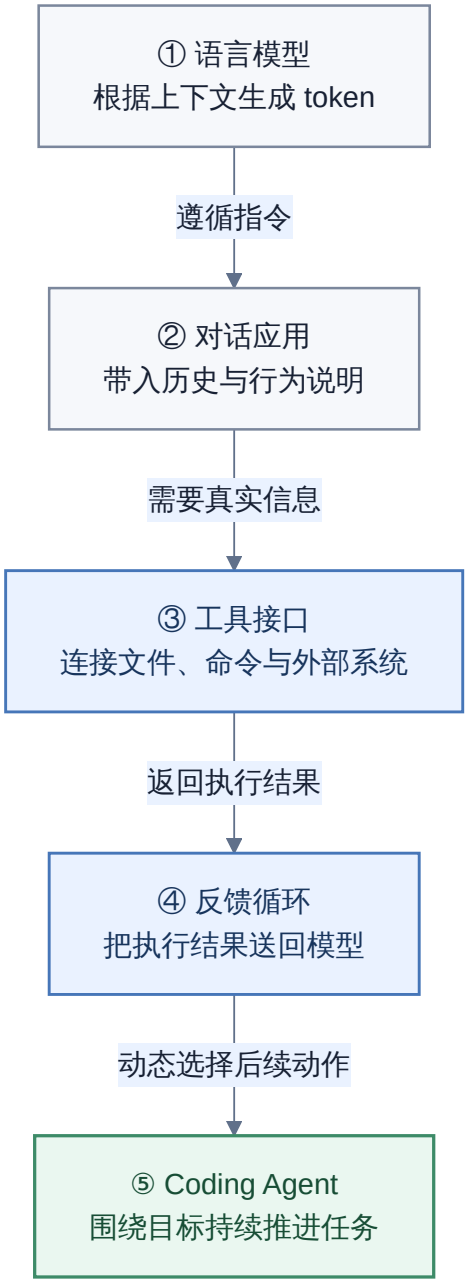

图 1-1：每一层都补上前一层缺少的能力。语言模型负责生成，对话应用维护上下文，工具连接外部环境，反馈循环让系统能够根据执行结果继续工作。

语言模型生成下一个 token

语言模型接收一段已有序列，计算下一个 token 的条件概率分布。token 是模型处理输入和输出的离散单位。中文里的一个 token 可能对应一个字、词的一部分或标点；源代码里的关键字、括号和空白也会被分词器编码为 token。

假设输入是“乌云越来越低，看样子马上要”，模型可能给“下雨”分配较高概率，也可能认为“天黑”可以接在后面。“吃饭”与当前上下文的关系较弱，概率通常更低。

模型给出候选分布后，解码算法从中选择一个 token。贪心解码总取概率最高的候选，采样算法则保留一定随机性。选中的 token 被追加到已有序列末尾，模型再计算下一位置。这个过程不断重复，直到产

生结束标记、达到长度限制，或完成协议要求的输出项。

早期统计语言模型依赖词频和相邻词组合。它们可以处理见过的局部模式，但很难表示长距离依赖。神经网络语言模型使用可训练的向量表示词语，并通过多层网络学习更复杂的关系。

2017 年提出的 Transformer 使用自注意力机制处理序列中的依赖关系。以“杯子放在桌上，它已经空了”为例，模型可以在计算“它”的表示时，为“杯子”和“桌子”分配不同权重。这里的“注意力”是一种数值计算，不表示模型具有人类的注意或意识。

Transformer 还适合在训练阶段并行处理序列中的多个位置。研究者可以使用更大的数据集和更多计算资源训练通用语言模型，再将模型用于问答、摘要、翻译和代码生成等任务。

GPT 是 Generative Pre-trained Transformer 的缩写，即生成式预训练 Transformer。训练过程中，优化算法根据预测误差调整模型参数。训练完成后，一次普通推理通常不会修改这些参数；模型根据已有参数和当前输入生成结果。

预训练模型怎样学会遵循指令

预训练主要教会模型预测文本。仅经过预训练的模型收到一个问题时，可能继续生成更多问题，也可能模仿网页内容补出作者、日期和评论。这些续写在统计上合理，却不一定符合用户意图。

后训练把模型的生成能力调整到指令遵循场景。监督微调使用“指令与理想回答”示例训练模型。偏好优化则利用人类或其他评估器对多个回答的比较结果，让模型的输出更符合用户意图。具体训练方法并不只有一种，但它们解决的是同一类问题：怎样让文本续写服务于用户给出的任务。

经过后训练，模型能够把“总结这份文档”“解释这段代码”或“找出报错原因”识别为需要完成的指令。不过，它的输入和输出仍然是 token 序列。持续对话与外部操作都由模型之外的系统提供。

对话连续性来自上下文

聊天应用在每次请求模型前组织一份输入。它通常包含当前用户消息、相关历史消息以及系统提供的行为说明。模型根据这份输入生成回答。

这些随请求一起交给模型的内容称为上下文。上下文还可以包含用户粘贴的文档、代码片段、工具返回结果和项目规则。模型在一次推理中只能使用当前上下文窗口内的信息。

应用可以在模型外部保存长期记录，也可以从数据库或搜索系统中检索资料。但检索到的内容必须重新进入上下文，或者通过工具接口提供给模型，才能影响本轮生成。模型不会在两次请求之间自行回忆上一次对话。

因此，对话过长时，应用可能删除、压缩或摘要较早的内容。新会话如果没有重新加载旧记录，模型也不会自动知道之前讨论过什么。对话的连续性由应用维护，不是模型参数在聊天过程中发生了变化。

上下文解决了连续交流问题，但不能直接改变外部环境。模型可以描述怎样整理目录，却无法仅靠一段文本移动文件。系统还需要把模型的意图转换为受控操作。

工具连接模型与外部环境

工具是外层程序向模型开放的操作接口。读取文件、搜索代码、执行命令、应用补丁和查询数据库都可以封装为工具。

每个工具需要定义名称、参数格式和返回结果。模型需要读取文件时，可以生成类似下面的结构化请求，而不是只输出一句自然语言说明。

```
{
  "name": "read_file",
  "arguments": {
    "path": "src/profile.tsx"
  }
}
```

模型并不会自行调用文件系统。运行时解析这个请求，检查参数、权限和执行条件，再把它交给对应工具。文件内容或错误信息随后作为工具结果写回会话；模型只有在下一次采样中读到这个结果，才能根据项目的实际状态继续判断。

工具结果可能改变原来的计划。指定路径可能不存在，测试命令可能失败，日志也可能指出另一个模块的问题。模型必须观察每次执行结果，再决定后续动作。单次“请求与回答”无法覆盖这种过程。

反馈循环构成 Agent

Agent 通常译为“智能体”。本书把它定义为：围绕一个目标观察环境、选择行动，并根据行动结果继续推进的系统。这个定义只描述工作机制，不讨论意识或人格。

一次简化的 Agent 循环如下。

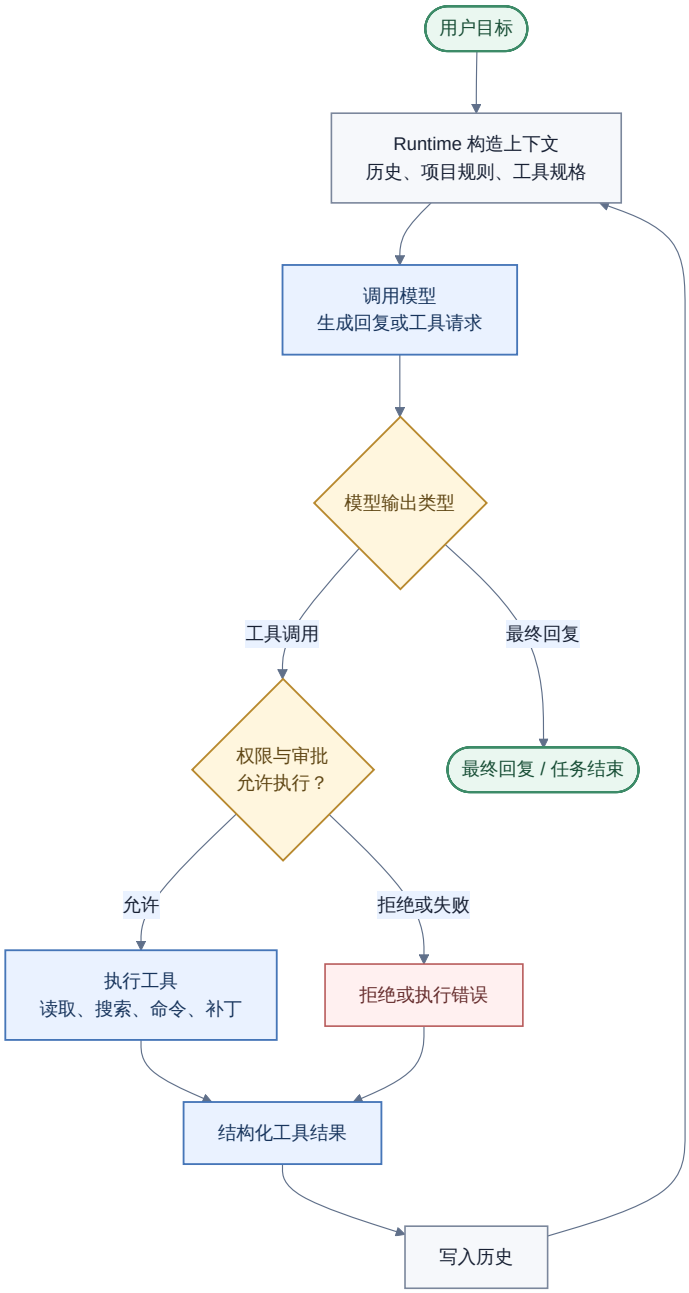

图 1-2：模型输出工具调用时，Runtime 先经过权限与审批边界，再把成功结果或错误结果写回历史。只有最终回复不再要求工具时，循环才结束。

模型可能在一轮中请求一个工具，也可能请求多个可以并行执行的工具。运行时收集这些结果后，判断是否需要再次调用模型。循环在模型给出最终回复、系统遇到无法恢复的错误、达到资源限制或收到中止请求时结束。

Agent 与自动化脚本之间没有绝对边界。脚本的主要路径通常由程序员预先写入代码；Agent 允许模型根据刚得到的观察结果选择后续动作。复杂脚本也能包含动态分支，Agent 也可以只调用一次工具。判断重点是执行路径是否会根据环境反馈在运行时改变。

动态选择使系统能够处理程序员没有预先枚举的错误路径，也增加了执行成本。循环可能调用模型多次，工具可能失败，上下文会持续增长，停止条件也可能判断错误。运行时需要处理超时、取消、重试和上下文限制，而不能只负责转发模型请求。

Coding Agent 在代码仓库中运行循环

软件项目能为 Agent 提供密集而具体的反馈。代码搜索返回定义位置，编译器报告语法和类型错误，测试暴露行为偏差，版本控制记录文件变化。这些结果可以验证模型的判断，也可以推翻它的初始假设。

以“资料页点击保存没有反应”为例，Coding Agent 可以按下面的顺序推进任务：

1. 查看项目结构，确认前端代码和测试的位置。
2. 搜索保存按钮及其事件处理函数。
3. 沿调用链检查请求参数、认证信息和错误处理。
4. 运行相关测试或复现命令，根据输出缩小范围。
5. 修改代码，再运行测试、类型检查和差异检查。

这组步骤会随反馈变化。搜索结果决定下一步读取哪个文件，测试输出决定是否修改原来的判断。如果现有测试没有覆盖故障路径，Agent 还需要补充验证方法，并在最终回复中说明证据缺口。

反馈也不能保证修改正确。测试可能缺失，日志可能不完整，需求本身也可能含糊。Coding Agent 的优势在于它能够获取和使用工程反馈，而不是仅凭生成的文字宣称任务已经完成。

Runtime 负责维持执行过程

模型不负责保存会话状态、路由工具、管理并发或记录任务进度。这些职责属于 Runtime，也就是运行时。

Runtime 为每次模型调用构造上下文，并向模型提供当前可用的工具说明。模型返回工具请求后，Runtime 选择对应执行器，收集输出，再把结果写入后续上下文。如果模型需要继续处理，Runtime 发起下一次模型调用；如果任务被取消，Runtime 停止仍在运行的操作并发出终止事件。

Codex 是一套面向软件开发任务的 Agent 系统。模型负责理解输入和生成下一步意图，工具负责读取或修改外部环境，Runtime 维护反馈循环。终端、编辑器和其他客户端负责接收用户操作并展示执行事件。

从用户请求到任务结束，可以先把主链路概括为：

```
用户提交任务
    ↓
Codex 组织上下文与工具说明
    ↓
模型生成回复或工具请求
    ↓
Runtime 路由并执行工具
    ↓
工具结果写回上下文
    ↓
模型继续处理，直到任务结束
```

沙箱与审批限制工具执行

模型生成的命令可能包含错误，也可能受到仓库中不可信内容的影响。工具执行因此需要明确的安全边界。

Codex 使用沙箱和审批策略控制外部操作。两者处理不同问题。

机制	处理的问题	作用
沙箱	命令在技术上可以访问什么	限制可读写路径、网络和其他系统资源
审批策略	哪些动作必须由用户决定	在执行越界或高风险操作前暂停并请求确认

会产生外部副作用的工具调用先经过策略判断，再在指定执行环境中运行。执行成功时，输出返回模型。沙箱错误可以返回模型用于调整方案；审批被拒绝时，系统可以取消当前动作或中止任务，具体行为由审批策略决定。安全控制位于 Agent 循环内部，因为工具调用正是系统产生外部副作用的位置。

权限放得过宽会扩大误操作和数据泄露风险，限制过严则会让任务频繁中断。具体配置取决于代码仓库的可信程度、任务需要的资源和用户允许的自动化范围。

章内索引

延伸阅读

- Attention Is All You Need
- Improving Language Understanding by Generative Pre-Training
- Training Language Models to Follow Instructions with Human Feedback
- ReAct: Synergizing Reasoning and Acting in Language Models
- Agent approvals and security

第二章　走进 Codex：一句“帮我修好它”是怎样被执行的

用户在 Codex 终端中输入：

> 资料页点击保存没有反应，请修一下。

界面很快开始显示状态：任务已经启动，Codex 正在搜索代码，随后运行命令、修改文件并汇报测试结果。用户看到的是一条连续的执行记录，内部却没有一个从头运行到尾的“修复函数”。这项任务会经过客户端协议、线程状态、输入队列、模型流、工具执行和事件输出等多个边界。

这条请求的完整执行路径由五类边界组成：客户端协议、Thread 与 Session 状态、模型采样、工具执行和事件投影。每层都有明确的输入、输出与状态所有者；模型调用和工具调用在同一个 Turn 内形成反馈循环。

客户端先建立 Thread

Codex 有多种交互入口。终端界面适合持续对话，`codex exec` 适合脚本和 CI，编辑器或其他富客户端可以通过 App Server 接入。App Server 是 Codex 为富客户端提供的双向协议服务。各类客户端负责收集输入和展示结果，不负责运行 Agent 循环。

在当前开源实现中，终端界面和 `codex exec` 都使用 App Server Client。这个客户端既可以连接进程内的 App Server，也可以连接远程 App Server。对上层界面来说，两种连接方式使用同一组请求和事件。

客户端开始工作前要取得一个 Thread。Thread 是用户与 Codex 之间的一段可持续会话，其中可以包含多次请求及其执行记录。新任务可以调用 `thread/start` 创建 Thread；继续旧任务时，可以使用 `thread/resume` 恢复已有 Thread。

Thread 除了在界面中组织消息，还确定后续输入追加到哪段历史，并为模型配置、工作目录、工具状态和持久化记录提供共同的生命周期。用户第二次说“把刚才的测试也补上”时，Codex 必须知道“刚才”属于哪段执行历史。

Core 是承载 Agent Runtime 的核心层。其中的 ThreadManager 负责创建、恢复和查找 Thread。每个正在运行的 Thread 都有一个长期存在的 Session。Session 持有对话历史、当前任务、配置快照和多种共享服务。客户端不会直接操作 Session，而是通过 CodexThread 提交操作并接收事件。

各层关系如下：

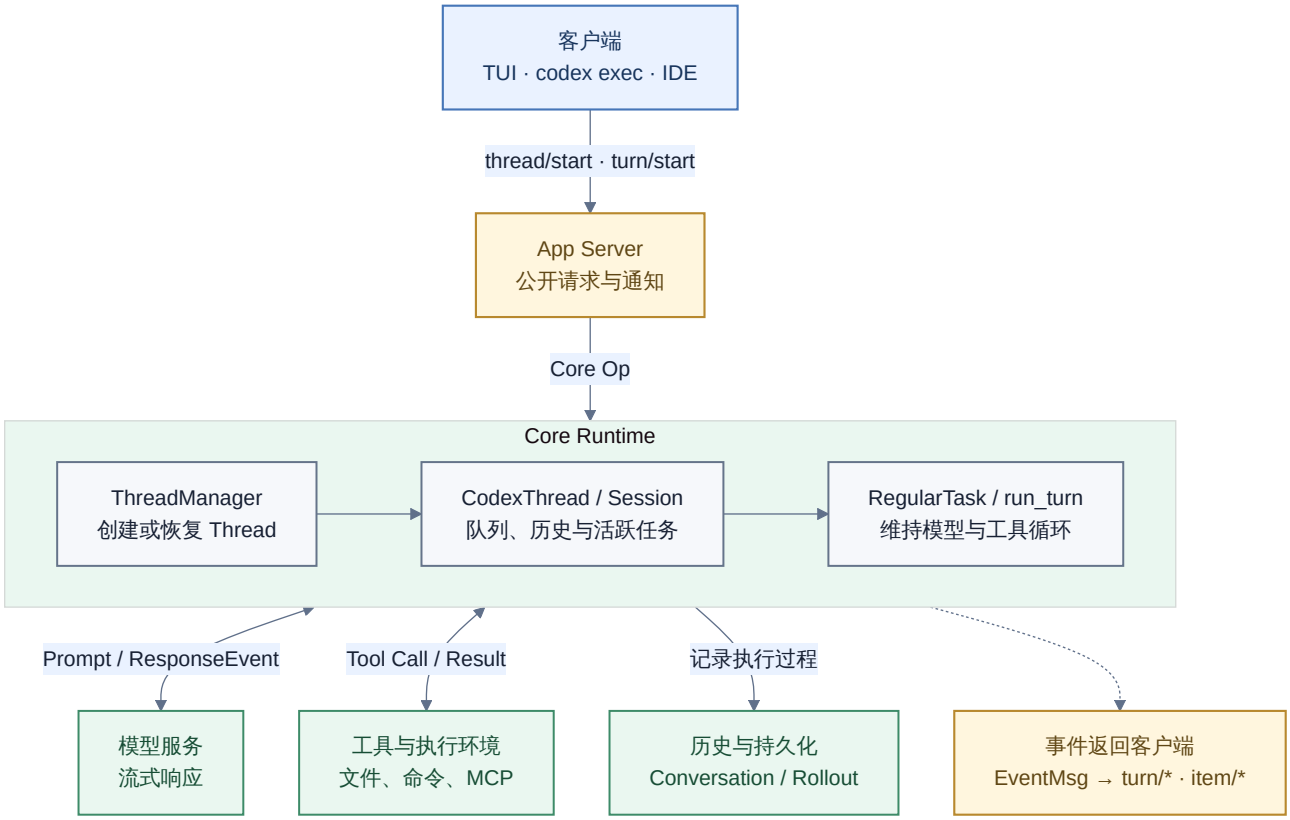

图 2-1：客户端只处理请求与展示；App Server 映射公开协议；Core Runtime 持有 Thread、任务循环、模型与工具状态。实线表示主要调用或数据流，虚线表示事件返回。移动端可横向滑动，点击可查看 SVG 原图。

App Server 位于客户端协议和 Core Runtime 之间。它校验公开请求，把请求转换为 Core 能处理的操作，再把内部事件投影成客户端可以稳定消费的通知。这一层使客户端不必依赖 Runtime 内部的 Rust 类型。

`turn/start` 把用户请求送入 Core

Thread 准备好以后，客户端用 `turn/start` 提交当前请求。按照 App Server 的公开协议，一次请求可以写成：

```
{
  "method": "turn/start",
  "id": 30,
  "params": {
    "threadId": "thr_123",
    "input": [
      {
        "type": "text",
        "text": "资料页点击保存没有反应，请修一下。"
      }
    ],
    "cwd": "/workspace/project",
    "approvalPolicy": "unlessTrusted",
    "sandboxPolicy": {
      "type": "workspaceWrite",
```

```
      "writableRoots": ["/workspace/project"],
      "networkAccess": false
    }
  }
}
```

请求中除了用户文字，还可以携带工作目录、模型、推理强度、审批策略和沙箱策略等覆盖项。App Server 先确认 Thread 存在，再检查输入大小和这些设置是否合法。校验通过后，它把公开输入转换为 Core 输入项，并构造 `Op::UserInput`。

`Op` 是 Operation 的缩写，表示客户端希望 Runtime 执行的一种操作。用户输入只是其中一种。中断、关闭、审批答复、回滚和上下文压缩也会以不同的 `Op` 进入同一条控制通道。

Core 不直接把 `Op::UserInput` 传给模型。CodexThread 先把它包装成 `Submission`。Submission 包含一个唯一 ID 和具体操作，随后进入 Session 的有界输入队列。这个 ID 也会成为当前 Turn 的 ID。

Turn 表示一次用户请求以及 Codex 为完成该请求开展的全部工作。一次 Turn 可以包含多次模型调用、多次工具调用和大量流式事件。App Server 在操作成功入队后立即返回一个状态为 `inProgress` 的 Turn，不会等待修复任务结束后才响应 `turn/start`。

输入队列提供了两个保证。第一，来自客户端的操作有明确顺序。用户输入、审批结果和中断信号不会以任意顺序同时修改 Session。第二，队列容量有限，生产速度超过 Runtime 的接收能力时会形成背压，而不是无限积累输入。

队列按顺序分派操作，不代表整个 Agent 只能串行工作。分派循环接到用户输入后会启动异步 Task，随后继续处理新的控制操作。模型响应可以流式到达，工具可以在满足约束时并行执行，中断和审批答复也能在任务运行期间进入 Session。

Session 为 Turn 启动一个 Task

Session 的提交循环收到 `Op::UserInput` 后，先检查当前是否已有活跃 Turn。空闲状态下，它为新请求建立 TurnContext，并启动 RegularTask。若已有任务正在运行，新输入会进入活动 Turn 的输入队列，由当前执行阶段决定何时吸收。

TurnContext 是本次 Turn 的主要运行配置。它包含模型、工作目录、权限策略和请求标识等信息。这里需要强调“主要”两个字：部分环境和工具状态可以在同一 Turn 的后续模型调用之间刷新，不能把整个执行过程理解成一份永远不变的配置。

Task 是 Session 中的一种执行策略。普通用户请求使用 RegularTask，代码审查、上下文压缩等工作可以使用其他 Task。Task 启动时，Session 会登记活跃 Turn，创建取消令牌，并把实际工作放进异步任务。

RegularTask 首先发出 `TurnStarted` 事件，然后进入 `run_turn`。取消令牌会沿调用链传给模型流和工具运行时。用户触发中断后，各层在可取消的等待点停止工作，Session 再清理活跃状态并发出中止事件。这种方式属于协作式取消：执行组件必须显式观察取消信号，Runtime 不能假定所有外部操作都能瞬间终止。

用户输入由此完成从产品协议到运行时任务的转换：

```
turn/start
    ↓
App Server 校验并映射输入
    ↓
Op::UserInput
    ↓
Submission 进入 Session 输入队列
    ↓
创建 TurnContext 和 RegularTask
    ↓
发出 TurnStarted，进入 run_turn
```

第一次模型调用发生在 Turn 的运行所有权建立之后。

一次 Turn 包含多个 Step

`run_turn` 负责维持模型与工具之间的反馈循环。一次模型采样及其输出处理构成一个 Step。这里的“采样”是一次完整模型请求，不是生成单个 token 的采样动作。

进入第一个 Step 前，Runtime 会处理必要的上下文压缩，记录用户输入，加载本轮涉及的技能、插件和扩展信息，并建立 ModelClientSession。ModelClientSession 是当前 Turn 内复用的模型传输会话。后续 Step 可以复用连接和路由状态，不需要把每次模型请求都当作互不相关的网络调用。

每次采样前，Session 都会捕获一份 StepContext。它表示本次模型请求实际看到的执行快照，其中包括当前环境、可用工具以及由这些工具构造出的路由器。Runtime 使用同一份 StepContext 完成三项工作：

1. 生成当前环境的 World State，也就是模型需要了解的工作目录、权限和其他运行状态。
2. 生成模型可见的工具规格，让模型知道工具名称、参数和用途。
3. 构造 ToolRouter，使模型随后发出的工具调用能够落到与这些规格对应的执行器。

工具规格和工具路由必须来自同一份快照。假设模型看到一个名为 `run_command` 的工具，但执行时 Runtime 已经换成另一套路由配置，模型生成的合法参数也可能无法执行。StepContext 把“模型看见什么”和“Runtime 能执行什么”绑定在同一个 Step 内。

准备完成后，Runtime 从对话历史生成模型输入，加入基础指令、当前 World State 和工具规格，再通过 ModelClientSession 发起流式请求。模型返回的内容不是只有最终文字，还可能包含推理摘要、普通消息、工具调用和用量信息等结构化响应项。

工具结果触发下一次采样

第一次模型采样时，模型还没有读取项目文件。对于“保存没有反应”这类请求，合理输出通常是一个工具调用，例如搜索页面组件。Runtime 收到完整工具调用后解析参数，通过 ToolRouter 找到执行器，再

由 ToolCallRuntime 管理并行和取消边界。`run_turn` 负责收集执行结果。

一次可能的执行过程如下：

Step	模型本次获得的新增信息	模型输出	Runtime 的处理
1	用户请求、项目规则、工具规格	搜索保存按钮相关代码	执行代码搜索并记录匹配位置
2	搜索结果	读取资料页组件和请求封装	读取文件并记录内容
3	文件内容	运行相关测试或复现命令	执行命令并记录退出状态与输出
4	测试错误	修改认证参数并再次测试	应用补丁，执行验证命令
5	修改差异和验证结果	给出最终说明	结束 Turn

表中的步骤只是示例。实际模型可能一次请求多个文件，也可能并行调用几个互不依赖的工具。关键约束没有变化：工具结果必须先进入历史，才能成为下一次模型采样的输入。

每个工具调用都有调用 ID。Runtime 将工具结果与原调用配对，记录到对话历史和持久化执行记录中。模型在下一次请求里会收到与调用 ID 对应的结构化结果，界面无须把执行结果拼成一段说明。成功输出、命令退出码和可恢复错误都可以通过这条路径返回模型。

模型发出工具调用后，当前采样结果会标记为需要继续。`run_turn` 收集仍在执行的工具结果，检查是否有用户追加输入或扩展要求续写，然后捕获下一份 StepContext 并再次调用模型。循环持续到以下条件都满足：模型没有留下需要处理的工具调用，没有待处理输入，也没有 Hook 要求继续。

因此，一次 Turn 和一次模型请求之间没有一一对应关系。Turn 是用户任务的边界，Step 是模型观察当前状态并决定下一步的边界。工具调用位于两次 Step 之间，负责把代码仓库中的新事实带回模型。

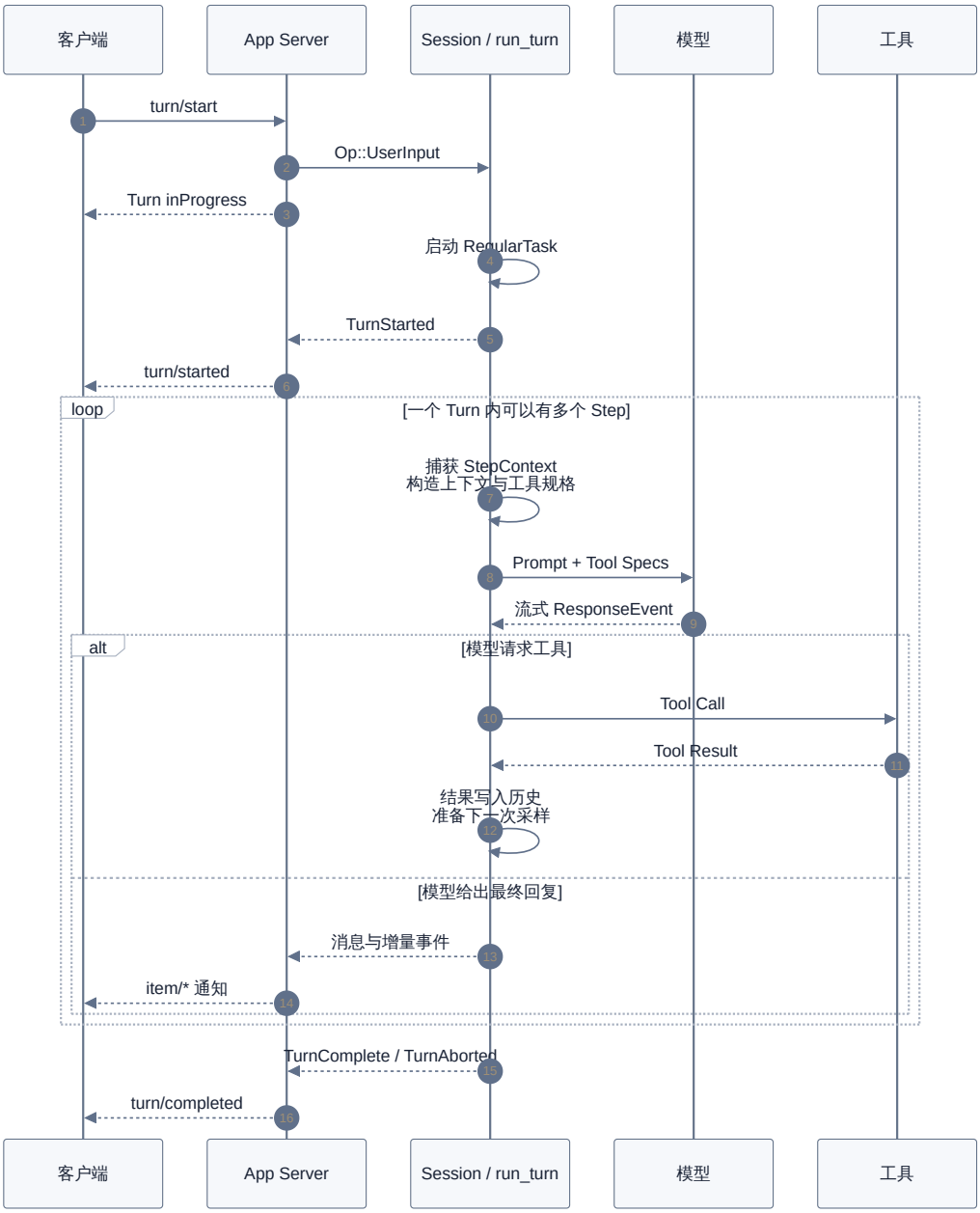

图 2-2：`turn/start` 只启动 Turn。虚线框内的模型采样和工具执行可以重复多次，直到模型给出最终回复、任务失败或用户中断。移动端可横向滑动，点击可查看 SVG 原图。

执行进度通过事件返回客户端

模型流和工具执行都在异步进行，客户端不能等到最后才知道状态。Core 使用 Event 输出运行中的变化。Event 包含关联 ID 和具体的 EventMsg，内容覆盖 Turn 生命周期、消息增量、命令执行、文件修改、审批、计划、差异、Token 用量、警告和错误。

App Server 持续读取这些 Core 事件，并将其转换为公开通知。例如：

```
Core EventMsg                    App Server 通知
──────────────────────────────   ──────────────────────────────
TurnStarted                  →   turn/started
AgentMessageContentDelta     →   item/agentMessage/delta
```

```
命令或文件项开始、完成            →  item/started、item/completed
TurnComplete / TurnAborted      →  turn/completed（带最终状态）
```

客户端根据通知类型更新界面。文本增量追加到当前消息，命令事件显示执行状态，文件事件更新差异视图，审批请求暂停相关操作并等待用户决定。客户端无须从自然语言中猜测“Codex 现在是否正在运行命令”。协议已经把状态和内容分开。

这种事件投影也隔离了内部实现变化。Core 可以增加更细的运行时事件，App Server 决定哪些字段成为公开协议。代价是两层状态必须保持一致：漏掉一个结束事件，客户端可能一直显示任务进行中；同一事件重复投影，也会造成消息或工具项重复。

完成、中断和失败走不同路径

正常情况下，最后一次模型采样只返回助手消息，也没有待处理输入。`run_turn` 退出后，Session 统计本轮用量，清理活跃 Task，并发出 `TurnComplete`。App Server 将它转换为状态为 `completed` 的 `turn/completed` 通知。

错误发生在哪一层，会决定 Turn 是否还能继续。

情况	Runtime 的处理	客户端看到的结果
`turn/start` 参数无效或 Thread 不存在	App Server 拒绝请求，不启动 Task	JSON-RPC 错误
搜索无结果、命令退出非零、工具参数错误	作为工具结果写回，允许模型调整方案	工具项失败，Turn 通常继续
命令需要审批	挂起相关工具调用，等待客户端答复	审批请求和待处理工具项
用户中断当前 Turn	触发取消，停止可取消的模型与工具工作	`turn/completed`，状态为 `interrupted`
模型流中断且重试耗尽、内部状态损坏	记录错误并终止当前 Turn	错误事件及状态为 `failed` 的完成通知

把工具错误返回模型很重要。测试失败可能正是定位问题所需的证据，文件不存在也可能提示模型先搜索正确路径。如果所有非零退出都直接终止 Turn，Agent 无法利用失败结果修正计划。

可恢复并不等于无限重试。模型请求有传输重试和连接回退策略，工具也可以报告错误，但 Runtime 仍要受到上下文窗口、资源限制、超时和取消信号约束。失败路径必须最终收敛为可观察的事件和明确的 Turn 状态。

分层带来的收益与代价

从客户端到 Core 的协议边界，让 TUI、非交互命令和富客户端共享同一套任务语义。Submission 队列让控制操作按顺序进入 Session。异步 Task 使运行中的 Turn 仍可接收中断、审批和追加输入。

StepContext 保证每次模型采样使用一致的环境与工具视图。结构化事件则让客户端能够稳定展示进度。

这些机制也引入了工程成本。一次用户请求可能产生多次模型调用，延迟和 Token 消耗随循环增长。工具并行、流式响应和用户中断会产生复杂的时序组合。公开通知与内部事件之间还需要维护一套状态投影。Runtime 的职责因此覆盖模型调用、操作顺序、状态快照、一致性、取消和错误传播。

完整执行路径如下：

```
用户输入
  → 客户端发送 turn/start
  → App Server 映射为 Op::UserInput
  → Submission 进入 Session
  → RegularTask 启动 Turn
  → run_turn 构造 StepContext 和模型请求
  → 模型返回消息或工具调用
  → 工具结果写回历史
  → 需要时进入下一个 Step
  → Core 事件经 App Server 返回客户端
  → Turn 完成、失败或被中断
```

章内索引

入口、配置、Thread、协议、普通与特殊 Turn，以及三种 Surface 的适配边界分别见：

1. CLI、TUI、Exec 与 App Server 的进程入口
2. 配置文件、Profile、命令行覆盖和受管配置的合并顺序
3. Feature Gate 怎样决定实际启用的 Runtime 分支
4. ThreadManager 的创建、恢复、派生与注册
5. CodexThread 的提交端、事件端与状态查询端
6. Session 初始化时并行建立的服务和失败收尾
7. Op 协议的类别、所有者和分派入口
8. EventMsg、Raw Response Item 与界面事件的层次
9. 普通 User Turn 的端到端调用链
10. 工具 Turn 的端到端调用链
11. Compact、Review、UserShell 等非普通 Task 的入口
12. TUI 怎样把用户动作逐层翻译到 Core Op
13. App Server 怎样把 JSON-RPC 翻译为 Core 操作和通知
14. Exec Server 与远程执行环境的协议边界

延伸阅读

- Codex App Server
- Codex CLI
- Codex 开源组件
- Agent approvals and security

第三章　一项任务的一生：Thread、Turn 与 Step

Codex 正在修复资料页的保存按钮。它已经找到组件代码，准备运行测试。此时用户又补充了一句：

> 不要改接口，只修前端。

这句话应该加入正在执行的任务，还是创建一项新任务？如果用户紧接着点击中断，正在等待的模型请求、已经启动的测试进程和尚未答复的审批又该由谁停止？测试停下以后，之前的对话历史是否还保留？

这些问题不能只靠一个反复调用模型的循环解决。Runtime 必须知道哪些状态属于整段会话，哪些只属于当前请求，哪些在一次模型采样结束后就应失效。Codex 用 Thread、Session、Turn、Task 和 Step 把这些时间尺度分开。

状态所有权、生命周期切换、运行中追加输入和协作式取消共同决定一项任务怎样开始、继续与终止。

章内索引

19. Session Shutdown：按所有权回收后台任务、子进程与长期服务
20. 终态所有权：为什么一个 RunningTask 只提交一次 Complete 或 Aborted
21. Rollout Flush：终态附近为什么需要多道持久化屏障
22. 生命周期变体：Regular、Review 与 Realtime 为什么不能共用一套状态机

五个名称对应五种时间尺度

五个名称对应不同的状态范围与存活时间。

名称	它表示什么	典型生命周期
Thread	一段可以继续、恢复和持久化的会话身份	从创建开始，可跨多次 Turn，停止后还可以恢复
Session	一个已加载 Thread 的内存 Runtime	从 Thread 被加载到关闭或进程退出
Turn	一次用户请求及为完成它开展的全部工作	从 `TurnStarted` 到完成、中断或失败
Task	执行某类 Turn 的运行策略	从 Session 启动后台任务到清理结束
Step	模型观察当前状态并采样一次的内部边界	从捕获 StepContext 到本次模型输出处理完成

这五层不是五种写法不同的“任务”。Thread 回答“这段对话是谁”，Session 回答“现在由哪个活体 Runtime 管理它”，Turn 回答“用户这次要求做什么”，Task 回答“Runtime 用哪种策略执行”，Step 回答“模型这一次实际看到了什么”。

它们的所有权关系如下。

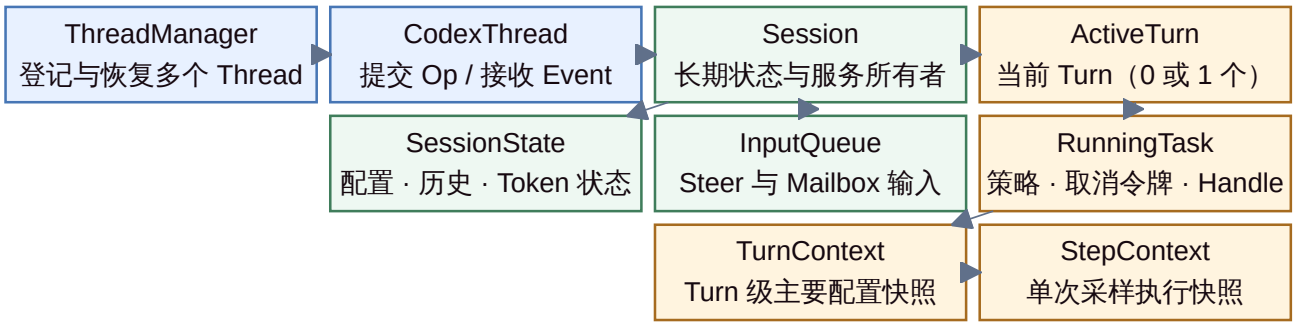

图 3-1：ThreadManager 管理多个可用 Thread；每个已加载 Thread 由一个 Session 承载。Session 同时至多有一个活跃主 Task，而同一 Turn 可以先后捕获多份 StepContext。移动端可横向滑动，点击可查看 SVG 原图。

Thread 是身份，Session 是活体 Runtime

`ThreadManager` 负责创建、恢复和查找 Thread。它在进程内维护一张从 Thread ID 到 `CodexThread` 的表，并共享模型管理器、环境管理器、MCP、Skills、Plugins 和存储等服务。

`CodexThread` 是外部调用者接触 Core 的双向接口。它包装一个 `Session` 和一组输入输出通道：调用者通过它提交 `Op`，再读取 Core 发出的 `Event`。它本身不运行模型循环，真正持有会话状态的是 Session。

Session 中既有跨 Turn 保留的状态，也有只在运行期间存在的资源。前者包括当前配置、对话历史、Token 统计和附加上下文；后者包括事件发送端、输入队列、工具与扩展服务以及当前活跃 Turn。

Thread 和 Session 的区别在恢复场景中最清楚。一个正在运行的 Thread 已经有 Session，再次请求恢复时可以直接复用。如果 Thread 只剩持久化记录，Runtime 会从历史重新构造一个 Session。Thread ID 和历史仍然延续，但原来的内存对象、异步任务和网络连接不会被“复活”。

因此，不能把 Session 当成 Thread 的另一个名字。Thread 是可恢复的会话身份，Session 是该身份当前加载出来的运行实例。

Session 串行处理控制操作

Session 启动时会创建一条 Submission 通道，并运行长期存在的 `submission_loop`。用户输入、中断、审批答复、设置变更、上下文压缩和关闭请求都会先成为 `Submission`，再由这条循环按接收顺序分派。

串行分派解决的是控制顺序，不代表模型和工具只能串行执行。循环收到用户输入后，会把实际工作放进独立的异步 Task，自己继续接收后续操作。因此，模型仍在流式返回内容时，Session 依然能够处理审批答复或 `Interrupt`。

这里还有另一条容易与 Submission 通道混淆的队列：`InputQueue`。

队列	保存什么	什么时候消费
Submission 通道	`UserInput`、`Interrupt`、审批答复、设置变更等控制操作	Session 主循环逐项分派
Turn InputQueue	运行中追加的用户输入、附加上下文和 Agent Mailbox 消息	当前 RegularTask 在模型采样边界读取

前者决定 Runtime 先处理哪个操作，后者决定哪些新信息应进入当前 Turn 的下一次模型请求。把它们合并成一条队列，会让“中断当前任务”和“把一句补充要求交给模型”失去清晰边界。

没有活跃任务时，UserInput 创建新 Turn

Session 收到 `Op::UserInput` 后，先根据 Thread 当前配置和本次覆盖项构造 `TurnContext`。它包含 Turn ID、模型、推理设置、协作模式、权限、环境选择、Skills 快照和终态元数据等信息。

TurnContext 是一次 Turn 的主要配置快照。这里的“主要”很重要：模型和权限等决策需要在本次任务中保持一致，但 MCP 连接、环境就绪状态和项目指令仍可能在后续采样前刷新。Codex 没有把所有运行状态一次复制后永久冻结。

如果 Session 此时没有活跃 Task，它会创建 RegularTask，并把两类对象登记进 `ActiveTurn`：

- `RunningTask` 保存 Task 类型、取消令牌、后台 Tokio Handle 和 TurnContext；
- `TurnState` 保存待处理输入、审批等待者、动态工具响应、临时权限和本 Turn 统计。

Session 的 `active_turn` 是可选值，而且一个 Session 同时至多运行一个主 Task。这个约束让审批、追加输入和终态事件都能明确关联到当前 Turn。工具执行仍然可以在 Task 内并发，这与“只有一个主 Task”并不冲突。

Task 决定 Turn 怎样运行

Task 不是操作系统进程，也不是 Thread 的子会话。它是 Session 对一类工作的执行策略。统一接口的核心职责是报告任务类型、执行 `run`，并在需要时通过 `abort` 完成取消后的专用清理；追踪名称等运行信息也由 Task 提供。

Codex 当前把主要任务分成三类：

Task	用途	是否接受运行中追加输入
RegularTask	普通对话、代码修改和工具反馈循环	接受
ReviewTask	运行受限制的代码审查流程并投影结果	不接受
CompactTask	压缩上下文，重建可继续使用的历史	不接受

普通用户请求由 RegularTask 执行。它先发出 `TurnStarted`，取得启动阶段预热的模型传输会话（如果可用），然后进入 `run_turn`；同一次 `run_turn` 内的后续采样会复用这份会话。ReviewTask 和 CompactTask 仍然使用同一套 Session 生命周期设施，但它们各自实现不同的执行逻辑与清理动作。

把 Task 单独抽象出来有两个直接作用。第一，Session 不需要在主循环里堆叠普通对话、审查和压缩的全部分支。第二，所有 Task 共享启动、取消、统计、持久化和终态事件规则，客户端不必为每种内部实现重新理解生命周期。

代价也很明确：Task 的业务结束不等于生命周期已经结束。Runtime 仍要刷新执行记录、计算用量、发送终态事件并安全地清除 ActiveTurn。把这些动作散落到每个 Task 中，很容易产生重复完成事件或残留状态，因此统一收口在 Session 的完成处理里。

一个 Turn 为什么需要多个 StepContext

RegularTask 进入 `run_turn` 后，才开始第一次模型采样。采样前，Session 捕获一份 StepContext。它把这一次请求实际使用的动态状态绑定在一起，包括：

- 当前环境是否已经就绪；
- 此刻加载到的 `AGENTS.md`；
- 执行环境提供的能力发现结果；
- 当前 MCP Binding 和模型可见工具列表；
- 与工具列表对应的 ToolRouter。

模型看到的工具规格和 Runtime 随后使用的路由来自同一份 StepContext。即使 MCP 配置在采样期间刷新，已经发出的工具调用仍按产生它的那份执行视图解释，不会在半途中换到另一套路由。

当工具结果进入历史、用户追加输入到达或上下文发生变化时，下一次采样会捕获新的 StepContext。新的快照可以看见已经刷新的环境和工具状态，旧快照则保持不变。

因此，Step 不是一个长期保存的业务对象，也没有独立的客户端生命周期。它是一次模型采样的内部一致性边界。一个 Turn 可以只有一个 Step，也可以经历“搜索—读取—测试—修改—再测试—总结”等多次 Step。

RegularTask 和 `run_turn` 的控制关系可以写成下面的伪代码：

```
async def run_regular_task(turn: TurnContext) -> str | None:
    while True:
        result = await run_turn(turn)
        if not input_queue.has_pending_input(turn.id):
            return result


async def run_turn(turn: TurnContext) -> str | None:
    while True:
        step = await capture_step_context(turn)
        outcome = await sample_model(step)
        await execute_and_record_tool_calls(outcome, step)

        if not outcome.needs_follow_up and not input_queue.has_pending_input(turn.id):
            return outcome.last_message
```

内层循环处理模型与工具的反馈，外层检查完成边界是否又出现了输入。两层检查不是重复劳动：用户可能恰好在最后一次采样结束、`run_turn` 准备返回时补充要求，外层检查可以避免这条输入被遗漏。

运行中输入会 Steer 当前 Regular Turn

在示例场景中，Codex 正在运行测试时，用户补充“不要改接口，只修前端”。Session 不会立即再启动一个并行 Turn，而是先检查当前 ActiveTurn。

如果活跃 Task 是 RegularTask，这条输入会作为 pending input 加入当前 TurnState。`steer_input` 返回当前 Turn ID，表明它属于原来的工作边界。`run_turn` 在下一次模型采样前把输入写入历史，模型随后可以根据新要求调整方案。

Steer 不是把一句话硬塞进正在传输的模型响应。已经发出的采样使用原 StepContext 完成；追加要求在安全的采样边界生效。这避免了同一请求的 Prompt 和工具视图在传输中途变化。

Runtime 还会检查调用者提供的 expected Turn ID。如果界面以为自己正在调整 Turn A，而 Session 实际已经进入 Turn B，输入会被拒绝，不会误写到新任务。ReviewTask 和 CompactTask 也不接受 Steer，因为把普通用户要求混入审查或压缩流程会破坏它们的输入契约。

如果根本没有 ActiveTurn，`steer_input` 会把输入原样退回给调用路径，Session 再用已经构造的 TurnContext 启动新的 RegularTask。同一种 `UserInput` 因此可以根据 Session 当前状态表现为“创建 Turn”或“调整当前 Turn”。

完成和中断都必须经过状态收口

普通 Turn、追加输入和中断的状态关系如下。

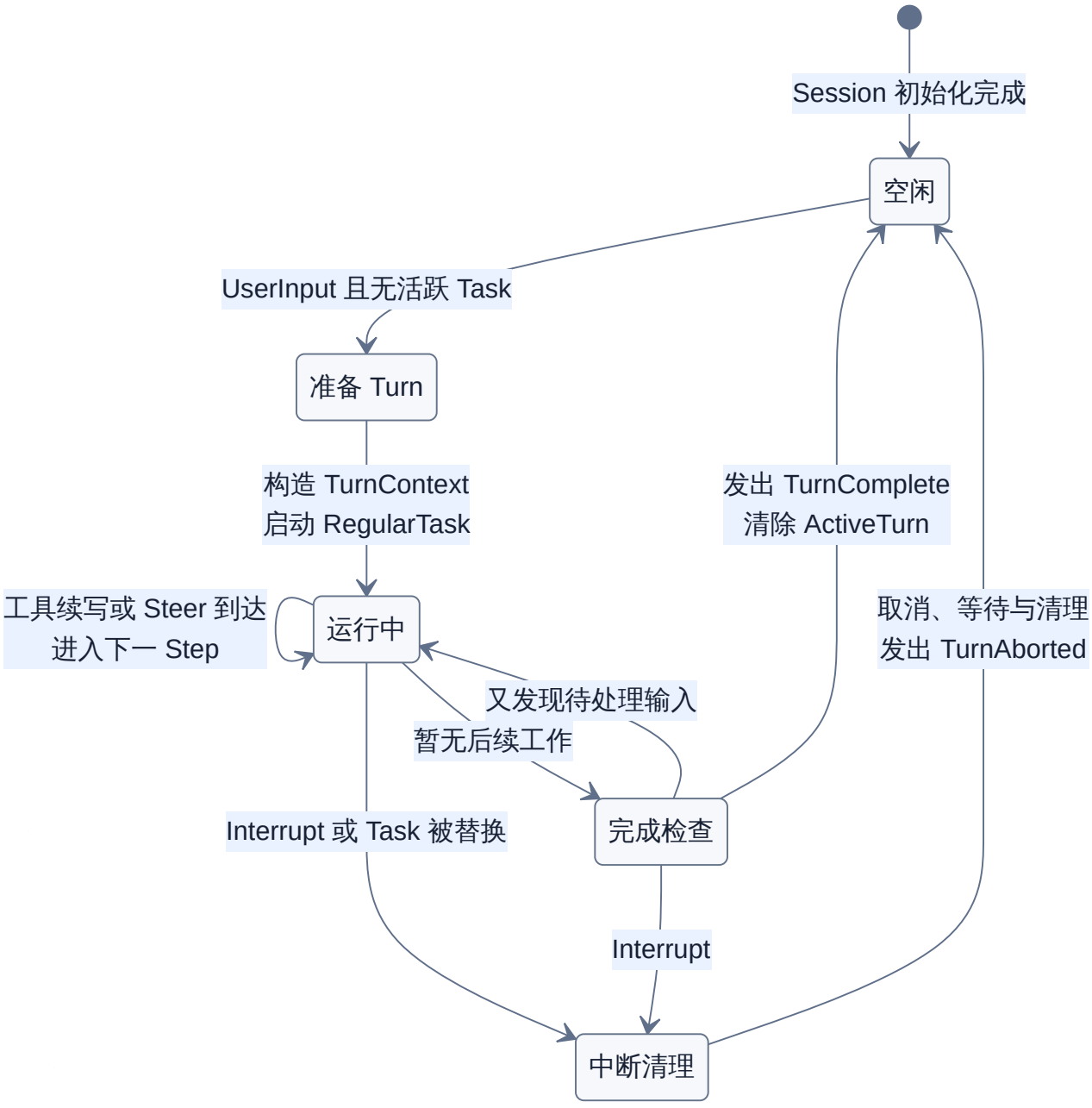

图 3-2：Regular Turn 可以在工具反馈或 Steer 输入到达后继续采样。正常完成与中断都回到同一个 Session 空闲态，因此结束 Turn 不等于结束 Thread。移动端可横向滑动，点击可查看 SVG 原图。

正常完成时，Task 的 `run` 先返回结果，Runtime 刷新执行记录，再由 Session 统一完成以下动作：处理完成边界残留的输入，计算本 Turn 的 Token 和工具调用统计，发出 `TurnComplete`，清除

RunningTask，最后把 ActiveTurn 设回空闲。

清理 ActiveTurn 时还要确认它仍然对应刚完成的 TurnState。这个检查防止一个已经结束的异步任务误删后来建立的新状态。只有清理成功后，Session 才发出 Thread Idle 生命周期，并检查是否有排队工作需要唤醒新的 Turn。

中断走另一条路径。`Op::Interrupt` 到达 Session 主循环后，Runtime 先原子地取走 ActiveTurn，使新的 Steer 不再写入旧任务。随后取消 RunningTask 的 CancellationToken，给模型流、工具执行和审批等待一个协作退出的窗口；如果 Task 没有及时结束，再中止它的后台 Handle，并调用 Task 自己的 `abort` 清理。

对于用户中断，Codex 会先把模型可见的中断标记写入历史并完成持久化刷新，再发出 `TurnAborted`。这样恢复后的模型知道上一轮不是自然结束，客户端收到中断事件后重新读取记录时也不会看见缺失的历史。待审批者、待输入和其他 Turn 级等待项随后被清除。

无论正常完成还是中断，Session 和 Thread 都继续存在。用户下一句话会在同一个 Thread 中启动新 Turn，并继承此前已经提交到历史的内容。只有 Shutdown 或 Thread 被卸载，当前 Session 的生命周期才结束。

生命周期边界换来可控的并发

这套设计维持了几条关键不变量：

1. 一个 Session 同时至多有一个主 Task，但 Task 内部可以运行并发工具和流式模型请求。
2. TurnContext 固定一次 Turn 的主要决策，StepContext 固定一次采样真正使用的动态视图。
3. 运行中的普通 Turn 可以吸收 Steer，特殊 Task 明确拒绝不兼容输入。
4. 取消先切断 ActiveTurn，再通知异步工作退出，旧任务不能继续接收新输入。
5. Turn 的完成、中断和持久化状态最终都由 Session 收口。

这些边界让 Codex 能在同一 Thread 中持续工作，同时避免多个主任务争抢历史、权限和终态事件。它们也增加了实现成本：输入可能落在采样、工具执行或完成检查之间；取消依赖每个异步组件配合；Turn 级配置和 Step 级动态状态必须维护清晰的刷新规则。

延伸阅读

- Codex Core 的 Session 实现
- Codex 的 Task 实现
- ThreadManager 源码

第四章　模型眼中的世界：提示词、上下文与历史

假设用户对 Codex 说：

> 修好资料页的保存按钮。不要改后端接口，只运行前端测试。

模型要正确完成这件事，不能只看到这一句话。它还得知道当前工作目录、项目规范、此前读过的文件、刚才的测试结果、哪些命令需要审批，以及这一轮可以调用哪些工具。

这些信息也不能在会话开始时一次性写死。Codex 可能切换到另一个工作目录，权限可能变化，工具可能在下一次采样前刷新；与此同时，已经完成的工具调用和用户刚补充的要求又必须留在历史中。

因此，模型每次采样前看到的内容不是一份静态“提示词模板”，而是 Runtime 根据当前 Session、历史和 StepContext 编译出来的一次结构化请求。编译过程必须确定请求组成、动态状态更新和 History 的发送前投影。

Prompt 不是一段拼接好的长字符串

许多简单 Agent 会这样构造输入：

```
系统提示词 + 项目规则 + 对话历史 + 工具说明 + 用户问题
```

这种写法适合建立直觉，却不足以描述 Codex 的实际请求。Codex Core 内部的 `Prompt` 至少包含以下字段：

字段	内容	主要来源
`base_instructions`	模型的顶层基础指令	SessionConfiguration
`input`	有类型的上下文和对话历史	ContextManager
`tools`	本次采样允许模型调用的工具规格	当前 StepContext 的 ToolRouter
`parallel_tool_calls`	是否允许并行提出工具调用	当前模型能力
`output_schema`	最终回答需要满足的结构	TurnContext

这些字段不会被提前压成同一段文本。工具规格仍然是 `ToolSpec`，历史仍然是 `ResponseItem` 列表，输出约束仍然是 Schema。到了模型客户端一层，它们才会被映射成具体 API 所需的请求格式。

这也解释了为什么“模型看到的 Prompt”不能只理解为界面上显示的系统提示词。更准确的说法是：Prompt 是一次采样所需的完整输入包，文字指令只是其中一部分。

Base Instructions 走一条独立通道

`base_instructions` 不在普通对话历史里。Session 创建时按三层优先级选择它：

1. 配置显式提供的覆盖值；
2. 恢复会话保存的 Base Instructions；
3. 当前模型自带的默认指令。

恢复记录排在当前模型默认值之前，是为了让继续执行的 Session 保持原来的基础行为，而不是仅因进程重启就悄悄换掉底层规则。选定以后，指令由 SessionConfiguration 持有；每次构造请求时，采样路径再从 Session 读取。

模型切换和 Personality 变化还需要另一层处理。若当前模型与此前模型不同，World State 可以生成一条带边界标记的模型切换说明；Personality 可能已经被编入顶层模型指令，也可能作为单独的上下文更新注入。于是同一个概念会经过两条不同的数据线：

- 顶层 Base Instructions 确定本 Session 的基础行为；
- 历史中的模型或 Personality 更新告诉模型当前状态发生了什么变化。

如果把两者混成一段“系统消息”，恢复、切换和差异更新的语义都会变得含糊。

History 保存的是有类型的项目

ContextManager 保存的不是聊天窗口文本，而是一列从旧到新的 `ResponseItem`。其中既有 Developer、User 和 Assistant 消息，也有 Reasoning、工具调用、工具结果以及压缩相关项目。

例如，一次“先搜索再回答”的过程，在历史中更接近下面的结构：

```
Message(role=user, "保存按钮为什么失效？")
Reasoning(...)
FunctionCall(call_id="call_17", name="search_files", ...)
FunctionCallOutput(call_id="call_17", output="...")
Message(role=assistant, "问题出在表单提交条件。")
```

调用和结果通过 `call_id` 建立关系。这个关系不仅方便 Runtime 找到工具结果，也是模型理解因果顺序的必要条件：先有一次工具意图，后有该意图对应的观察结果。

项目规则、权限和环境信息同样会进入 `ResponseItem`，但它们并不是普通用户发言。Codex 用 `ContextualUserFragment` 表示这类 Runtime 注入片段。每个片段自己声明角色、正文、起止标记，以及是否必须单独成为一条消息。

标记的作用是让 Runtime 以后能认出“这段文字由哪类上下文生成”。没有这种边界，历史压缩或版本迁移时只能用模糊的文本匹配，很容易把用户恰好说过的相似句子误当成系统状态。相邻、同角色且允许合并的片段可以装进同一条消息；要求隔离的片段仍保持独立。

第一次采样先建立完整世界

一个新 Session 还没有上下文基线。第一次真实采样前，Runtime 会先收集 Developer Instructions、Skills 目录、扩展贡献和完整 World State，再按片段角色写入历史。用户的实际任务随后与这些上下文一起进入模型输入。

World State 可以理解为“这次采样中，模型需要知道的当前运行状态”。它由多个有稳定 ID 的分区组成，常见内容包括：

分区	告诉模型什么
Model / Personality	当前模型身份以及必要的行为切换说明
AGENTS.md	当前项目和目录范围内适用的开发规则
Permissions	可访问范围、审批策略和执行限制
Collaboration Mode	当前协作方式和工作流程约束
Environments	工作目录、执行环境和可用能力
Apps / Plugins / Tools	当前可见的扩展能力或延迟工具
Context Window Guidance	在有限上下文预算下应遵循的行为

这些分区不是配置文件目录的镜像，而是模型可见状态的边界。每个分区负责保存足够比较的 Snapshot，并负责把变化渲染成带角色的自然语言片段。

首次构造和后续采样的数据流如下。

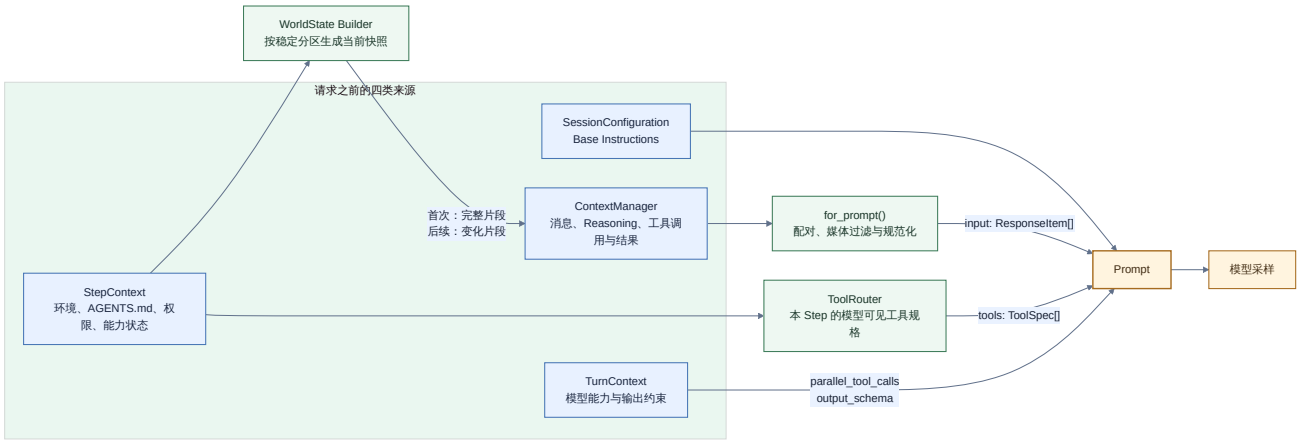

图 4-1：World State 的完整或增量片段先进入 History，History 再按模型能力规范化；Base Instructions、工具规格和输出约束沿独立通道进入同一个 Prompt。移动端可横向滑动，点击可查看 SVG 原图。

图中有一条容易忽略的约束：World State 和工具规格都来自同一份 StepContext。模型得到“当前允许哪些操作”的文字说明时，它看到的工具列表和 Runtime 随后执行调用所用的 ToolRouter 也属于同一个采样快照。不能用旧权限说明配新工具表，也不能在请求发出后临时换掉路由。

AGENTS.md 是有目录作用域的动态状态

项目规则是 World State 中最容易观察的一类。Codex 从当前工作目录向上寻找项目根，默认用 `.git` 等根标记确定边界；然后从项目根到当前目录逐层寻找规则文件。

同一层如果存在 `AGENTS.override.md`，它优先于 `AGENTS.md`。项目还可以配置后备文件名。最终内容按“根目录规则在前、靠近当前目录的规则在后”的顺序组合，并受总字节预算限制。搜索不会越过项目根。

这套规则解决了两个不同问题：

- 仓库根规则可以约束整个项目；
- 子目录规则可以补充当前组件、语言或测试方式。

加载结果由 `AgentsMdManager` 缓存。环境选择和工作目录没有变化时，可以复用已有结果；选择变化后会重新发现，再把结果放入新的 StepContext。因此，不能把它描述成“每次采样都无条件重读磁盘”，也不能把它当作 Session 启动后永远不变的常量。

当适用的项目规则变化时，Codex 不只是再追加一份新文本。`AgentsMdState` 会明确告诉模型：新内容替换此前的 AGENTS.md 指令。如果当前范围不再有规则，它会明确撤销旧指令。否则，模型可能同时保留两套互相冲突的目录规则，却不知道哪套已经失效。

后续 Step 只发送变化

完整注入能建立正确起点，但不能在每次采样前原样重复。项目规则、权限和工具说明可能很长，重复发送既占上下文，也会破坏稳定的请求前缀。

World State 为此保留一份紧凑 Snapshot。后续 Step 构造新状态后，Runtime 按分区比较前后快照：

- 分区没有变化：不生成模型消息；
- 分区新增：生成首次说明；
- 分区变化：生成更新或替换说明；
- 分区消失：生成撤销说明。

与此同时，当前 Snapshot 会与旧 Snapshot 计算一份 RFC 7386 JSON Merge Patch，用于 Rollout 持久化。这里有一个必须分清的边界：模型收到的是各分区渲染出的上下文片段，Rollout 保存的是推进状态基线的结构化 Patch。JSON Patch 本身不是发给模型阅读的 Prompt。

更新顺序也有意固定。Codex 先把模型可见变化记录到 History，再持久化描述该变化的 World State Patch。这样恢复过程不会先看到“基线已经改变”，却找不到让模型知道这一变化的上下文消息。

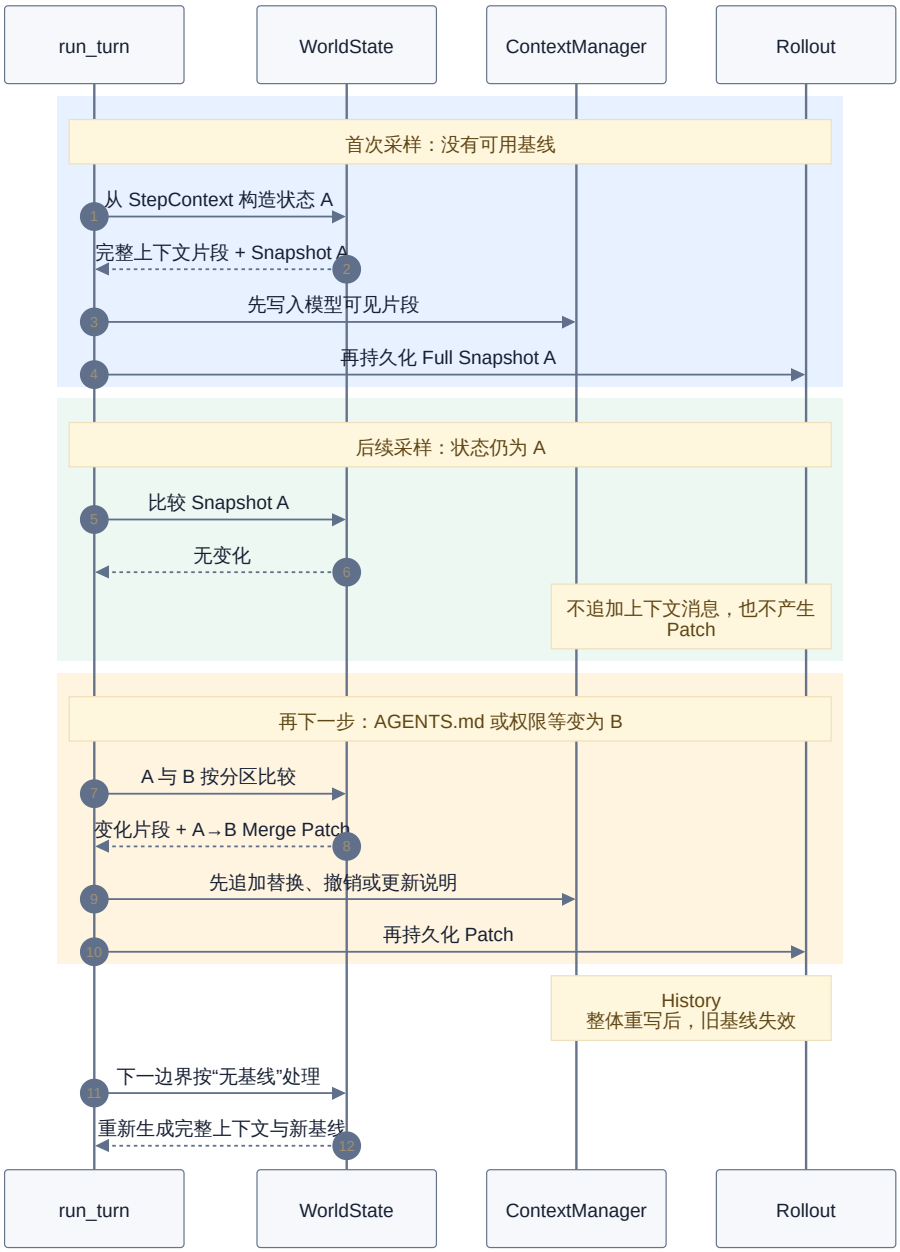

图 4-2：首次采样建立完整基线；状态不变时不追加任何内容；状态变化时先写模型可见片段，再持久化 Patch。History 整体重写后旧基线失效，下一边界重新建立完整上下文。

核心组装算法可以写成下面的伪代码：

```
async def prepare_prompt(turn: TurnContext) -> Prompt:
    step = await capture_step_context(turn)
    current = await build_world_state(step)

    fragments, patch = history.diff_world_state(current)
    if fragments:
        history.record(merge_by_role(fragments))
    if patch is not None:
        await rollout.append(patch)

    model_input = history.snapshot().for_prompt(
        input_modalities=turn.model.input_modalities
```

```
    )
    return Prompt(
        base_instructions=session.base_instructions,
        input=model_input,
        tools=step.tool_router.model_visible_specs(),
        parallel_tool_calls=turn.model.supports_parallel_tool_calls,
        output_schema=turn.final_output_schema,
    )
```

实现还区分首次 Turn、同一 Turn 内的后续 Step 和压缩后的新上下文窗口，但顺序不变：先捕获一致的 Step，更新模型可见状态，再从 History 生成请求快照。

发送前还要修复 History 的结构

History 已经按顺序记录，为什么不能直接交给模型？因为持久化历史允许出现对恢复有意义、却不一定满足模型 API 契约的边界状态。

最典型的情况是工具调用被中断。历史里可能已有 `FunctionCall`，进程却在写入结果前停止。如果下一次请求只带这半对记录，模型和 API 都无法判断调用是否仍在执行。

`ContextManager.for_prompt()` 会在发送副本上执行规范化：

1. 调用缺少结果时，紧跟调用插入一个内容为 `aborted` 的合成结果；
2. 结果找不到对应调用时，移除这条孤立结果；
3. 裁剪历史时，调用和结果作为一对处理；
4. 当前模型不支持图片或音频时，按输入能力剥离或替换相关内容。

这些合成修复只服务于本次 Prompt，不会反向伪造持久化历史。合成结果的 ID 又由原调用 ID 稳定派生，所以同一份历史在重试或恢复时会生成同一个修复项。这不仅维持结构正确，也减少了请求前缀无意义变化对 Prompt Cache 的影响。

大型工具输出还会在记录历史时按模型策略截断。对 Coding Agent 而言，这一点很实际：一次测试可能输出几万行日志，模型通常只需要错误附近的部分。如果每次都把完整 stdout 带回后续 Step，真正重要的项目规则和用户意图反而会被挤出上下文窗口。

压缩不是简单删除旧消息

上下文窗口终究有限。Codex 会结合 Base Instructions 和 History 估算当前占用，并在达到相应边界时进入压缩流程。压缩可以把较长历史替换成摘要和保留项目，但它也改变了此前用于差异比较的前缀。

因此，ContextManager 只要整体替换 History，就会提升 `history_version` 并清空 World State 基线。新的上下文窗口会插入规范的初始上下文，或者由下一次正常 Turn 在发现无引用基线后进行完整重注入。回滚如果裁掉了建立基线的混合上下文，也必须采取同样做法。

这个选择看起来保守，却比继续使用过期基线安全。假如摘要里已经没有旧权限说明，而 Runtime 仍认为模型“早就知道”，后续又只发送一个很小的差异，模型实际看到的状态就会缺一块。重建完整上下文要多花一些 Token，但能重新对齐模型视图和 Runtime 视图。

增量更新与稳定合成 ID 对缓存友好，压缩和回滚则会主动牺牲一部分前缀稳定性来换取正确性。两者并不矛盾：缓存只能优化一份语义完整的 Prompt，不能成为保留错误基线的理由。

调试 Prompt 要沿同一条编译链

检查 Prompt 时，只打印用户最后一句话没有意义。Codex 内部的调试入口会创建临时 Session，捕获 StepContext，记录初始上下文和用户输入，再调用 `for_prompt()` 与 `build_prompt()`。它复用了正常请求的关键准备链路，而不是另写一个“看起来差不多”的拼接器。

遇到模型行为异常时，也可以沿这条链逐层定位：

现象	优先检查
项目规则没有生效	当前环境选择、工作目录、AGENTS.md 发现结果和替换消息
模型仍按旧权限行动	StepContext 是否刷新，World State 是否生成权限差异
工具结果像是丢失	History 中调用与结果的 `call_id`，发送前是否被规范化
压缩后模型忘记运行环境	History 替换后是否重建引用基线和完整 World State
模型能看到工具却无法执行	ToolSpec 与 ToolRouter 是否来自同一个 StepContext

“记忆”与“现状”

- Base Instructions 提供 Session 级基础行为；
- History 保存已经发生的消息、推理、调用和观察；
- World State 描述当前 Step 仍然成立的动态事实；
- ToolSpec 与输出 Schema 保留结构化协议；
- ContextManager 在发送前保证历史满足模型输入契约。

History 回答“此前发生了什么”，World State 回答“现在仍然是什么”。前者主要追加，后者必须能够替换和撤销。把它们分开后，Codex 才能在长期会话中既保留行动轨迹，又不会让过期的项目规则、权限和环境永久生效。

章内索引

1. Base Instructions：Session 行为基线怎样选择、持久化和下沉到请求
2. Developer 与 User Instructions：权威不靠拼接顺序模拟
3. AGENTS.md 发现：项目根、候选优先级与有序并发探测

4. 嵌套 AGENTS.md：每环境预算、来源标签、选择缓存与替换通知
5. ContextualUserFragment：渲染、识别与消息合并是三份合同
6. 初始上下文：新窗口的状态基线，不是一条用户消息
7. WorldState：同一状态怎样生成 Prompt 差异与可恢复 Patch
8. Environment Context：cwd、Shell、Readiness 与权限轮廓
9. Permissions World State：把执行策略翻译成模型合同
10. Model 与 Personality：稳定 Session 基线之上的切换协议
11. Collaboration Mode 与 Multi-Agent Mode：两套相邻但独立的状态机
12. Skills：发现目录、压缩元数据与注入正文是三条路径
13. Plugins 与 Apps：安装事实、用法提示、显式提及和推荐列表
14. 动态能力进入模型的三条通道
15. ContextManager：把持久化事实与单次请求投影分开
16. History 变换：追加、头删、整体替换与按 Turn 回滚
17. Function Call 与 Output：在请求副本中修复因果对
18. Custom Tool、Local Shell 与 Tool Search：四套配对协议
19. 媒体规范化：同一 History 如何投影给不同模态的模型
20. Token 估算：从模型可见字节到三层预算
21. Context Window：策略提示、窗口身份、剩余量与一次性提醒
22. Local Compaction：普通采样怎样变成新的 History 检查点
23. Remote Compaction：候选 transcript 怎样通过客户端提交边界
24. Compaction 提交：History、World State 与 Turn Context 必须一起换代
25. Prompt Cache：Key 只定义归属，稳定前缀才决定复用
26. `previous_response_id`：把完整 Prompt 安全压成 WebSocket 尾部 delta
27. 从 Prompt 到 Responses Request：逐字段编译
28. 怎样证明模型真正看见了什么：Prompt 的分层测试体系

延伸阅读

- Codex Prompt 与模型客户端公共类型
- Codex ContextManager
- Codex World State
- Codex AGENTS.md 发现逻辑

第五章　回答是怎样流出来的：模型连接与事件

模型连接层接收一份结构化 `Prompt`，其中包含基础指令、规范化历史、本次可用的工具和输出约束。它不能被简化为“调用一次模型然后取回字符串”。

一次采样期间，界面可能先显示几段文字，随后出现一项工具调用；网络可能在中途断开，重连后请求又被发送一次；同一 Turn 还可能执行工具并继续采样。要让这些行为可恢复、可取消且不污染历史，Runtime 至少要分清三个边界：

- 哪些内容只是尚未完成的显示增量；
- 哪个事件交付了一项可记录的完整模型输出；
- 什么信号才表示整次响应已经成功结束。

Codex 的模型调用层就在这些边界之间工作。它把内部 `Prompt` 映射成 Responses 请求，通过 SSE 或 WebSocket 接收线协议事件，再把两种传输统一成 Runtime 可消费的 `ResponseEvent`。

一次采样返回的不是一个字符串

一次采样从 Runtime 向模型发送一个 `Prompt` 开始，以收到该响应的 `response.completed` 结束。模型可以在这段时间内返回多个有类型的输出项，例如：

```
Response
├── Reasoning
├── Message(role=assistant, ...)
└── FunctionCall(call_id="call_17", name="exec_command", ...)
```

这些输出项统称为 `ResponseItem`。一项 Assistant Message 内部可以继续产生多个文字增量，一项工具调用也可以逐段产生参数。因而下面三个概念不能混用：

概念	表示什么	能否直接写入 History
Delta	某个输出项尚未完成的一小段内容	否
`ResponseItem`	已完成的一项消息、Reasoning 或工具调用	是
Response Completed	整次模型响应已结束，并附带响应 ID、Token 用量等信息	它提交响应边界，不代替具体 Item

如果模型输出工具调用，一次采样会先产生完整的 Tool Call Item，再以 Completed 结束。Runtime 随后执行工具，并把结果加入 History，开始同一 Turn 中的下一次采样。模型连接层的责任到完整 Tool Call 提交为止，真实副作用由本地工具运行时负责。

Prompt 在客户端层被编译成 Responses 请求

内部 `Prompt` 仍然与具体网络协议无关。`ModelClient` 收到它后才构造 `ResponsesApiRequest`：

内部数据	Responses 请求字段
当前模型	`model`
Base Instructions	`instructions`
规范化 History	`input`
ToolSpec 列表	`tools`
并行工具能力	`parallel_tool_calls`
推理强度与摘要配置	`reasoning`
输出 Schema 与回答详细度	`text`
Session 级缓存身份	`prompt_cache_key`

请求还会设置 `tool_choice: "auto"` 和 `stream: true`。这一步保留了内部数据的类型边界：History 仍以 `ResponseItem[]` 发送，工具仍以结构化定义发送，JSON Schema 进入 `text.format`，而不是先拼成一篇巨大的文字提示词。

Responses Lite 使用另一种编码方式。它把工具定义包装成 Developer 角色的 `AdditionalTools` 输入项，把 Base Instructions 包装成 Developer Message，并清空顶层 `instructions` 和 `tools`；同时关闭并行工具调用。也就是说，Lite 是请求能力适配，不只是给同一份 JSON 换一个地址。

内容	标准 Responses	Responses Lite
基础指令	顶层 `instructions`	Input 前缀中的 Developer Message
工具定义	顶层 `tools`	Input 前缀中的 `AdditionalTools`
并行工具调用	按模型和 Prompt 决定	关闭
Reasoning Context	使用服务默认行为	显式覆盖全部 Turn

这层适配让上游 Runtime 继续面对同一个 `Prompt`，协议差异被限制在模型客户端内部。

两层客户端对应两种生命周期

连接复用最容易引入跨 Turn 状态污染。Codex 没有把所有状态都塞进一个长期客户端，而是分成 `ModelClient` 和 `ModelClientSession`。

对象	生命周期	主要状态
`ModelClient`	整个 Codex Session	Provider、认证、Thread ID、Prompt Cache Key、传输回退状态、可复用 WebSocket 状态

对象	生命周期	主要状态
`ModelClientSession`	一个 Turn	本 Turn 的流式请求、增量请求基线引用、`x-codex-turn-state`
`WebsocketSession`	可在 Turn 之间转移	物理连接、上次完整请求、上次完成响应及连接复用信息

这里的关键字段是 `x-codex-turn-state`。服务端可以在 Turn 开始时返回这枚粘性路由令牌，客户端在同一 Turn 后续的重试、增量追加或继续采样中原样带回。它不能进入下一个 Turn。

因此，每个 Turn 都创建新的 `ModelClientSession` 和空的 `turn_state`。Turn 结束时，仍然健康的 WebSocket 连接等传输状态可以放回 `ModelClient` 缓存，供下一 Turn 取用。复用的是连接，重置的是 Turn 语义状态。

这种设计比“每次请求都重新连接”复杂，但它明确回答了状态所有权问题：

- HTTP 回退是否继续生效，由 Session 级 `ModelClient` 决定；
- 当前 Turn 应送回哪枚路由令牌，由 Turn 级 `ModelClientSession` 决定；
- 连接是否仍可复用，由 `WebsocketSession` 决定。

SSE 与 WebSocket 在事件层汇合

当前 Provider 支持 Responses WebSocket，并且本 Session 没有进入 HTTP 回退状态时，`ModelClientSession::stream` 优先选择 WebSocket；否则通过 HTTP POST `/responses` 建立 SSE 流。

两条路径在线路上不同：

- SSE 由一次 HTTP 请求承载，服务端持续返回 `text/event-stream`；
- WebSocket 先建立双向连接，再发送 `response.create`，后续响应以文本帧返回。

但它们不会把两套协议一路泄漏到 `run_turn`。SSE Data 和 WebSocket Text Frame 都先反序列化为 `ResponsesStreamEvent`，再经过同一个事件映射函数，变成统一的 `ResponseEvent`。

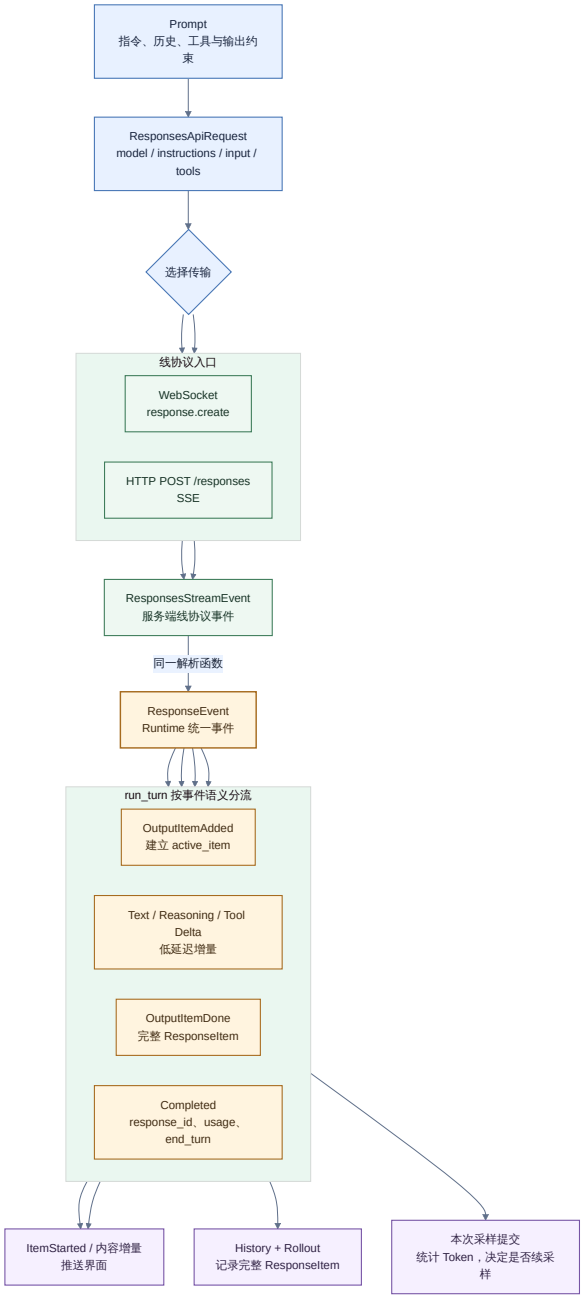

图 5-1：传输差异在 codex-api 层收敛。Delta 进入低延迟展示，OutputItemDone 交付可持久化的完整项目，Completed 提交整次采样。

这条统一事件线有两个直接收益。第一，Runtime 不需要为 SSE 和 WebSocket 各写一套消息、Reasoning 和工具调用处理器。第二，传输可以在失败后切换，而上游仍消费同一种事件类型。

五类事件构成一次响应的状态机

与内容主链最相关的事件可以按生命周期排列：

1. `Created`：服务端已创建响应；
2. `OutputItemAdded`：一个输出项开始，Runtime 建立当前活动项；
3. `OutputTextDelta`、Reasoning Delta、Tool Input Delta：内容逐段到达；

4. `OutputItemDone(ResponseItem)`：该输出项已完整；
5. `Completed`：整个响应完成。

`run_turn` 在收到 `OutputItemAdded` 后设置 `active_item`。此后的文字 Delta 必须属于这个活动项；如果没有活动项却收到文字增量，实现会把它视为协议或状态错误。界面此时可以收到 Item Started 和内容增量，所以用户无需等待整条回答生成完毕。

`OutputItemDone` 到达后，Runtime 清理该项的流式解析状态，并处理服务端给出的完整 `ResponseItem`。Assistant Message 会成为完成的 Turn Item；Tool Call 会先作为完整调用记录下来，再进入工具处理路径。

`Completed` 则处理响应级工作：发送 Raw Response Completed 事件、记录 Token Usage、刷新限流信息，并检查 `end_turn`。服务端若明确给出 `end_turn: false`，Runtime 会要求继续采样。

除了这条内容主线，`ResponseEvent` 还承载 Rate Limit、服务端实际模型、Models ETag、安全缓冲和账户验证等元数据。它们不会变成对话消息，而是更新 Session 状态或转成界面事件。

Delta 负责即时反馈，Done Item 负责事实

流式系统很容易犯的错误，是把每个文字 Delta 直接追加到持久化 History。网络若在一半断开，History 就会留下一个模型从未完成的句子；重试请求又可能在这个半句之后继续，模型视图与服务端响应边界从此错位。

Codex 把两种职责分开：

- Delta 只驱动界面展示和流内解析；
- `OutputItemDone` 提供完整、带类型的 `ResponseItem`；
- 完整 Item 通过 `record_conversation_items` 进入 History 和 Rollout；
- `Completed` 再确认整次响应正常结束。

这不是说 Item Done 之前的内容没有价值。它们降低了首字延迟，还能让工具参数差异消费者提前观察参数变化。只是这些内容仍是暂态，不应冒充已经提交的对话事实。

当前源码也没有一个统一、官方命名为 `ResponseAccumulator` 的核心对象。一次响应的暂态分散在 `active_item`、Assistant Message 流式解析器、Plan Mode 状态和工具参数差异消费者中，完整项则由 `OutputItemDone` 交付。后面实现 Mini Codex 时可以主动把这些职责收进一个较小的 `ResponseAccumulator`，但那是教学实现的取舍，不是官方类型的翻译。

预热减少首个请求的准备时间

WebSocket 可以按需连接，但首次 Turn 若把认证、握手和请求准备全部串在用户提交之后，首个事件会更晚出现。Codex 因此提供启动预热。

Session 启动后，Runtime 会尽力构造一个启动用 TurnContext 和 StepContext，建立当前工具 Router，并用 Base Instructions、工具规格和空 Input 生成一份 Prompt。预热请求使用：

```
{
  "type": "response.create",
  "generate": false
}
```

`generate: false` 表示建立连接和服务端响应链，而不是执行一次应计入正常 Rollout 的模型推理。客户端会等待该预热响应的 Completed，然后把得到的连接、完整请求基线和 Response ID 留给首个正常 Turn。

预热是尽力而为的优化。它有超时和取消边界；失败后不会阻止用户任务，而是让正常请求重新建立连接。这样可以把性能优化与正确性解耦：预热成功时少等一段，失败时协议语义不变。

Previous Response 只在严格匹配时使用

持久 WebSocket 解决了重复握手，但还没有解决重复发送长 Prompt 的问题。同一 Turn 的后续采样通常是在此前 History 末尾增加一个工具结果。如果每次都重传完整 Input，随着会话增长，线上请求会越来越大。

WebSocket 请求可以携带 `previous_response_id`，并只发送新增 Input。不过 Codex 不会只凭“这是同一连接”就使用增量。它先检查：

1. 模型、Instructions、Tools、Tool Choice、Reasoning、Service Tier、Prompt Cache Key、输出控制等非 Input 属性是否一致；
2. 当前 Input 的前缀是否严格等于“上次请求 Input + 服务端上次返回的完成项”；
3. 上次响应是否存在有效 Response ID。

条件全部成立时，客户端发送：

```
{
  "type": "response.create",
  "previous_response_id": "resp_123",
  "input": [
    {"type": "function_call_output", "call_id": "call_17", "output": "..."}
  ]
}
```

若基础指令、工具表、模型、输出 Schema 或历史前缀发生变化，客户端退回完整请求。这样做牺牲了一些潜在复用，换来一个明确不变量：增量请求与完整逻辑请求必须让模型看到同一份内容。

预热后的首个真实请求还有一个特例。若真实请求与预热基线完全匹配，可以引用预热 Response ID 并发送空 Input 增量。传输层虽然省略了重复内容，Rollout Trace 仍记录完整逻辑请求，保证以后调试或回放时看到的是模型实际上下文，而不是线路压缩后的残片。

连接、Previous Response 与 Prompt Cache 是三种复用

这三种机制经常被统称为“缓存”，但它们优化的对象不同：

机制	省掉什么	失效条件
WebSocket 连接复用	再次握手和建立传输	连接关闭、超时或被重置
`previous_response_id`	重发已有 Input 与已知返回项	请求属性变化或 Input 前缀不匹配
`prompt_cache_key`	服务端对稳定 Prompt 前缀的重复计算	由服务端缓存策略、Key 和前缀变化决定

`prompt_cache_key` 默认使用 Session ID，也可以被显式覆盖。它是请求中的缓存身份，不是 Response ID。稳定的 History 前缀有助于 Prompt Cache，增量请求减少线上载荷，持久连接减少传输准备。三者可以同时生效，但不能互相替代。

`x-codex-turn-state` 又是第四种不同的状态：它服务于同一 Turn 的粘性路由，不是缓存键，也不能跨 Turn 延续。

只有 Completed 才能证明流成功

TCP 连接正常关闭，不等于模型响应成功。服务端可能在只返回一半 Item 后断开，代理可能因空闲超时切断 SSE，WebSocket 也可能在完成事件前收到 Close Frame。

因此，SSE 和 WebSocket 都执行相同的终止检查：

- 收到 `response.completed`：正常结束；
- 收到 `response.failed`：按错误码映射为具体错误；
- 收到 `response.incomplete`：产生流错误；
- 连接在 Completed 前关闭：产生流错误；
- 超过空闲等待时间：产生流错误；
- 用户取消：终止当前 Turn，不作为网络重试。

`response.failed` 也不是一个笼统的“请求失败”。解析层会区分上下文窗口超限、配额不足、用量限制、无效请求、策略拒绝、服务过载和普通可重试错误。上下文窗口超限需要压缩或调整输入，配额问题需要用户或账户状态变化；盲目重发同一请求不会解决它们。

服务端实际使用的模型若因后端路由与请求不同，会通过 `ServerModel` 元数据事件报告。这与 WebSocket 回退 HTTP 完全不同：前者描述服务端模型选择，后者描述客户端传输选择。

重试会重建 Prompt，回退会改变 Session 状态

可重试错误进入采样外层循环。第一次尝试使用已经准备好的 Input；后续重试会从最新 History 再生成一次规范化 Prompt，而不是永久保存并机械重放旧请求。

这一点对流式响应尤其重要。某个完整 `OutputItemDone` 可能已经写入 History，随后连接才断开。重试从最新 History 构造请求，至少不会假装这些已经完成的项目不存在。尚未 Done 的界面 Delta 仍是暂态，不会进入新 Prompt。

等待时间优先采用错误中的 Retry-After；没有明确延迟时使用带抖动的退避。WebSocket 的普通重试预算耗尽后，Runtime 会：

1. 在整个 Codex Session 范围禁用 WebSocket；
2. 清空当前 WebSocket 连接和增量基线；
3. 重置重试计数；
4. 通过 HTTP SSE 重放请求。

HTTP 426 Upgrade Required 是更直接的信号，不需要先耗尽普通流重试预算。回退一旦生效，后续 Turn 也继续走 HTTP，避免每轮都重复尝试已知不兼容的 WebSocket。

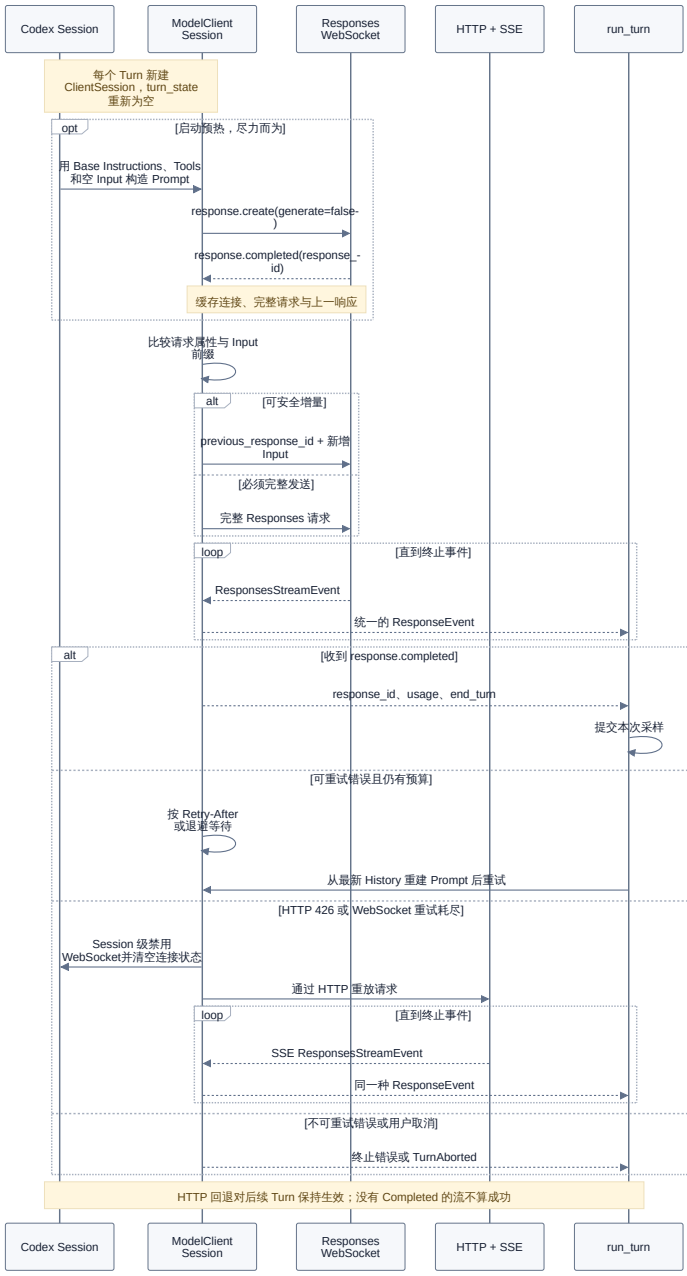

图 5-2：预热和 Previous Response 是可失效的优化；Completed 是成功边界。可重试错误先按预算重试，WebSocket 不再可靠时切换为 Session 级 HTTP 回退。

核心控制可以写成接近 Python 的伪代码：

```
async def sample(prompt: Prompt, turn: TurnContext) -> SamplingResult:
    retries = 0

    while True:
        try:
            stream = await model_session.stream(prompt)

            async for event in stream:
                match event:
                    case OutputItemAdded(item):
                        stream_state.start(item)
                    case TextDelta(text):
```

```
                    await ui.publish_delta(text)
                case OutputItemDone(item):
                    await history.record(item)
                case Completed(response_id, usage, end_turn):
                    await usage_store.record(usage)
                    return SamplingResult(response_id, end_turn)

        raise IncompleteStream("missing response.completed")

    except Cancelled:
        raise TurnAborted
    except NonRetryableError:
        raise
    except RetryableError as error:
        if retries >= turn.max_stream_retries:
            if model_session.enable_http_fallback():
                retries = 0
                prompt = await rebuild_prompt_from_history()
                continue
            raise

        retries += 1
        await asyncio.sleep(error.retry_after or backoff(retries))
        prompt = await rebuild_prompt_from_history()
```

伪代码省略了认证刷新、Reasoning 解析、工具参数增量和多项并发处理，但保留了四个关键不变量：取消不重试，Delta 不落历史，Completed 才成功，重试使用最新 History。

调试模型流要同时看内容和传输

当回答流异常时，只检查最终聊天消息往往不够。可以按现象定位：

现象	优先检查
首个字符很慢	启动预热是否完成、WebSocket 是否复用、认证是否阻塞
文本显示了一半后重新出现	是否在 Completed 前断流，界面显示的是否只是暂态 Delta
History 中缺少模型调用	是否收到 `OutputItemDone`，而不只是收到参数 Delta
每次都发送完整长请求	请求非 Input 属性是否变化，History 前缀是否严格匹配
WebSocket 失败后每轮都再次尝试	Session 级 `disable_websockets` 是否正确保持
日志显示模型变化	区分 Server Model 事件与 HTTP 传输回退
用量没有更新	是否真正收到带 Usage 的 Completed

调试记录也应同时保留两个视角：

- 逻辑请求：模型完整看到的 Instructions、Input、Tools 与输出约束；
- 线路请求：这次 WebSocket 是否只发送 Previous Response 和 Input Delta。

如果只记录线路请求，预热后的空增量看起来像“模型没有收到 Prompt”；如果只记录逻辑请求，又无法解释为什么网络载荷很小。Codex 的 Inference Trace 会在传输复用预热响应时补记完整逻辑请求，就是为了不混淆这两个视角。

模型调用层是一道协议防火墙

模型调用层维护三层边界：

- `Prompt → ResponsesApiRequest`：隔离 Runtime 领域模型与 Provider 请求格式；
- `ResponsesStreamEvent → ResponseEvent`：隔离 SSE、WebSocket 与 Runtime 事件处理；
- `Delta → Done Item → Completed`：隔离即时展示、可持久化事实与响应提交。

连接预热、Previous Response 和 Prompt Cache 都只优化这条主链，不能改变它的正确性条件。任何优化失效，都应退回完整请求或 HTTP 传输；任何流没有 Completed，都不能伪装成一次成功响应。

章内索引

22. 传输错误与 Provider 错误归一化
23. ContextWindowExceeded 与压缩入口
24. Output Schema 与本地结果验证
25. 模型流取消与协作式收尾
26. Realtime 与普通 Responses 的协议差异

延伸阅读

- Codex ModelClient 与 ModelClientSession
- Codex Responses 公共请求与 ResponseEvent
- Codex Responses SSE 事件解析
- Codex Turn 中的事件消费
- Codex WebSocket 增量请求测试
- Codex WebSocket 回退测试

第六章　Agent 的手：工具、命令、权限与沙箱

一项已经完成的模型输出可以包含工具调用：

```
FunctionCall(
    call_id="call_17",
    name="exec_command",
    arguments={"cmd": "npm test"}
)
```

这还不是一次命令执行。它只是模型提出的结构化意图。真正改变文件、启动进程或访问网络之前，Codex 还必须回答四个问题：

1. 模型调用的工具，是否就是这个 Step 曾经展示给它的那一个；
2. 工具名称和参数应交给哪个执行器；
3. 这次操作是否需要审批，最终允许接触哪些资源；
4. 执行结果怎样成为下一次模型采样可以读取的事实。

这四个问题共同构成工具执行链。它不是一个巨大的 `if name == ...`，也不是“用户允许后直接运行命令”。Codex 把工具描述、路由、生命周期、命令策略、审批、权限和沙箱分成不同层，再用同一份 Step 快照把它们接起来。

工具有给模型看的一面，也有给 Runtime 用的一面

模型不会直接持有 Rust 函数。它看到的是 `ToolSpec`：工具名称、说明和参数格式。当前实现支持几种不同形状：

ToolSpec	模型怎样提交调用	典型用途
`Function`	按 JSON Schema 生成参数	`exec_command`、`write_stdin`
`Freeform`	生成一段自定义语法文本	`apply_patch`
`Namespace`	在命名空间内选择具体工具	MCP、扩展工具和多工具分组
`ToolSearch`	搜索暂未直接展示的工具	Deferred Tool 发现
`WebSearch`	由 Responses 服务执行	Hosted Tool

Runtime 需要的是另一面：哪个名字由谁处理，能否并行，怎样执行。`ToolExecutor` 把两面绑在同一个接口里：

```
tool_name()                    精确路由名称
spec()                         模型可见定义
exposure()                     在什么界面暴露
```

```
supports_parallel_tool_calls() 是否允许并行
handle(invocation)             真正的执行入口
```

这个绑定很重要。如果工具规格和执行函数由两套互不相关的注册表维护，规格更新而执行器没更新时，模型可能生成 Runtime 无法解释的参数；执行器换了语义而规格仍旧时，副作用会比模型理解的更大。当前源码在 `codex-rs/tools/src/tool_executor.rs::ToolExecutor` 的注释中直接把“规格与可执行 Runtime 保持绑定”写成接口目标。

不是每个已经注册的工具都要立刻塞进 Prompt。`ToolExposure` 规定了四种暴露方式：

暴露方式	初始工具列表	可被本地 Registry 调度	Code Mode 中的含义
`Direct`	有	有	可作为嵌套工具
`Deferred`	无	有	先经 Tool Search 发现
`DirectModelOnly`	有	有	只保留普通模型调用入口
`Hidden`	无	有	仅供内部兼容或间接调度

还有一类 Hosted Tool 只有模型可见规格，没有本地 Handler，因为副作用由 Responses 服务完成。由此可见，“模型看得见”和“客户端注册了”是两个集合，不能用一张工具名称列表代替。

Deferred Tool 解决的是 Prompt 体积问题。工具数量较多时，Runtime 可以只展示 `tool_search`，把带搜索元数据的工具留在 Registry 中；模型找到所需工具后再调用它。Hidden Tool 则不允许模型发现，只保留内部调度能力。两者虽然都不在初始列表中，安全含义完全不同。

同一个 Step 同时生成规格和执行表

`StepContext` 冻结一次模型采样实际使用的环境、MCP Binding、能力发现结果和 AGENTS.md。工具执行使用其中的 `tool_router` 字段；源码注释把它定义为：

> 为这一次采样请求展示并执行的最终工具计划。

建立 Step 时，`capture_step_context_with_required_mcp_servers` 先刷新已选环境的就绪状态，再捕获 MCP 和扩展能力，最后调用 `built_tools`。工具规划器从同一批 `ToolExecutor` 生成两个结果：

```
ToolRouter
├── model_visible_specs: Vec<ToolSpec>
└── registry: ToolRegistry
    └── ToolName → CoreToolRuntime
```

`build_prompt` 从这个 Router 读取 `model_visible_specs`；`ToolCallRuntime` 则保存同一个 Router 和整个 `Arc<StepContext>`，以后用其中的 Registry 执行调用。即使工具 Future 稍后才运行，

它也不会偷偷改用下一次采样刚生成的工具表。

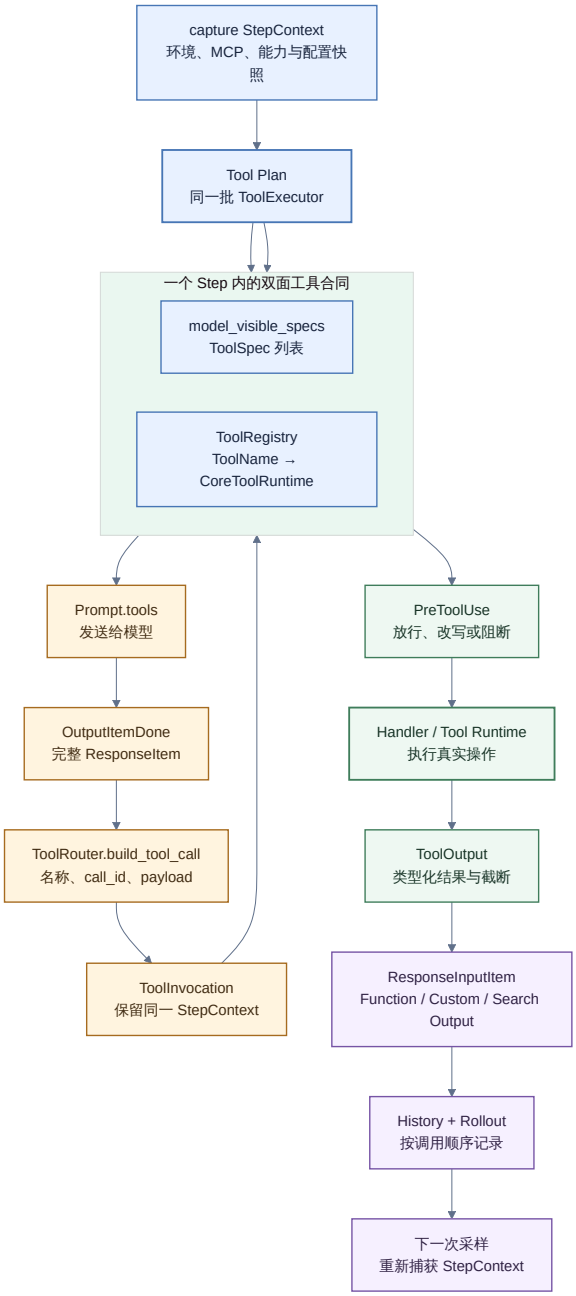

图 6-1：一个 Step 的工具计划同时产生 Prompt 中的规格和 Runtime 中的执行表。工具调用保留原 StepContext，结果进入 History 后，下一次采样才重新捕获工具与环境。

这条约束解决了一个真实的时序问题。假设一次 Turn 中，远程环境在第一轮采样时仍未就绪，第二轮才变成可用：

```
Step A：只展示本地 exec_command
Step B：展示带 environment_id 的多环境 exec_command
```

Step A 返回的调用必须按 Step A 的参数合同解析，不能因为执行发生得晚，就拿 Step B 的环境和 Schema 解释它。下一次采样可以采用新的计划，已经发出的调用不能跨快照漂移。

完整 ResponseItem 才进入工具路由

工具执行从 `OutputItemDone` 开始，而不是从参数 Delta 开始。`handle_output_item_done` 收到完整 `ResponseItem` 后调用 `ToolRouter::build_tool_call`，当前客户端执行路径主要处理三种输入：

- `FunctionCall` 变成 JSON 参数形状的 `ToolPayload::Function`；
- `CustomToolCall` 变成自由文本形状的 `ToolPayload::Custom`；
- `execution == "client"` 的 `ToolSearchCall` 变成 `ToolPayload::ToolSearch`。

Hosted Tool Search 不进入本地调度。工具名也不是一个扁平字符串：Namespace 和 Name 一起组成规范的 `ToolName`。因此 `mcp__calendar / create_event` 不会与本地同名函数碰撞。

解析成功后，Runtime 先把模型生成的 Tool Call Item 写入 History 和 Rollout，再创建工具 Future，并把 `needs_follow_up` 设为真。这里的顺序保证即使用户随后取消 Turn，历史里仍保留“模型曾经请求了什么”，不会只剩一个来历不明的工具结果。

工具 Future 经过 `ToolRegistry` 时还有几道结构检查：

1. Registry 按完整 `ToolName` 查找 Handler；
2. 未注册的名称变成模型可读的“unsupported call”结果；
3. Handler 检查 Payload 形状是否匹配；
4. 名称存在但 Function/Custom 形状不兼容，被视为 Runtime 内部契约错误；
5. 检查通过后才发送 Tool Start 生命周期事件。

“工具不存在”和“程序把错误 Payload 交给了已存在工具”性质不同。前者可能是模型使用了过期或错误的工具名，返回结果后模型还能改正；后者意味着规格、解析器和 Registry 之间出现了程序错误，不能假装成普通命令失败。

Hooks 包围执行，但执行后的阻断不能回滚副作用

Registry 不会一查到 Handler 就立刻调用。它先运行 `PreToolUse` Hook。Hook 可以：

- 继续使用原参数；
- 返回新输入，让 Handler 重建 `ToolInvocation`；
- 阻断调用，并把原因反馈给模型。

Handler 成功后，Registry 再运行 `PostToolUse` Hook。Post Hook 可以补充上下文、把反馈文字替换成模型可见结果，或阻断结果继续返回。不过此时真实操作已经发生。源码在 `registry.rs` 中专门提醒：PostToolUse 阻断的是结果，不是已经完成的工具执行。

例如 `apply_patch` 已经修改文件后，Post Hook 决定不把原结果交给模型，并不会自动恢复旧文件。需要事务语义的工具必须由自己的 Runtime 提供提交、补偿或回滚机制，不能把生命周期 Hook 当成数据库事务。

多个工具可以并行执行，但结果仍按调用顺序回灌

一次模型响应可以完成多个 Tool Call。Codex 把对应 Future 放进 `FuturesOrdered`，并通过一个共享 `RwLock` 控制执行：

- 声明支持并行的工具取得读锁，可以与其他并行工具同时运行；
- 不支持并行的工具取得写锁，会等待已有并行调用结束，并阻止其他调用进入；
- 即使实际完成时间不同，`FuturesOrdered` 仍按模型发出调用的顺序交付结果。

因此需要区分“执行顺序”和“History 顺序”。两个只读搜索可能并行，第二个先完成，但结果仍按调用次序与各自的 `call_id` 写入历史。这样既缩短等待，又给下一次采样一个稳定转录。

当前流程还有一个容易误读的细节：`OutputItemDone` 只是把 Future 排入队列。采样事件循环结束后，`drain_in_flight` 才轮询并等待这些 Future。工具不会在参数还没完成时抢跑，也不会把执行输出插进同一个尚未 Completed 的模型响应。

取消同样沿这层传播。每次调用获得子 `CancellationToken`。普通 Handler 可被直接中止；声明需要 Runtime 清理的 Handler 会先得到机会终止进程或释放资源。最终 Runtime 生成带失败语义的 Aborted Tool Output，让下一次模型采样知道操作没有正常完成，而不是让调用在 History 中永久悬空。

`exec_command` 不只是“一次 shell 调用”

命令工具最容易把工具路由、安全策略和进程生命周期同时暴露出来。当前 Unified Exec 组合通常向模型展示：

- `exec_command`：启动命令，等待一小段时间并返回当前输出；
- `write_stdin`：向仍存活的会话写入字符，或空写入以继续轮询。

旧的 `shell_command` 可以继续注册为 Hidden 兼容入口，但具体暴露仍受模型能力、Feature 和 Tool Mode 影响，不能把某一套工具表写成所有会话的固定常量。

`exec_command` 的 Handler 先从原 Step 的环境快照选择目标环境。没有 `environment_id` 时使用主环境；指定 ID 时只允许选择该 Step 中已经就绪的环境。工作目录相对环境自身的 cwd 解析，远程环境还使用它报告的 Shell 类型，而不是假定远端与 Codex 宿主机相同。

接下来命令被交给 Unified Exec Process Manager。短命令在初始等待窗口内结束时，结果直接带回退出码；仍在运行时，Manager 会先把进程存入 Session 级 Process Store，再返回 `process_id`：

```
exec_command
  ├── 已退出 → output + exit_code
  └── 仍运行 → output snapshot + process_id
                                  │
                                  └── write_stdin / 空轮询
```

`write_stdin` 使用工具参数名 `session_id` 接收这个进程 ID。对同一终端的读写会取得会话级交互锁，避免两个轮询同时抽走同一个输出缓冲区；不同终端仍可并行。空字符表示只等待新输出，非空字符写入 TTY。非 TTY 进程不能任意续写，只保留有限的中断处理。

`write_stdin` 也不会再次运行命令级 PreToolUse Hook，因为它只是已经批准并启动的命令的传输通道。当一次轮询观察到原命令最终结束时，PostToolUse 仍使用原 `exec_command` 的调用 ID 和命令内容，保持生命周期配对。

这套设计的代价是状态明显增加：Process Store、输出缓冲、交互锁、超时、终止、网络审批和清理都要跨多次工具调用保持一致。收益则是编译、测试服务器和交互式程序不必被伪装成一个超长的同步函数调用。

命令策略先判断“是否应该执行”，沙箱再约束“执行时能碰什么”

命令到达进程创建之前，`ExecPolicyManager` 会把 Shell 命令尽量拆成可判断的命令片段，然后同时参考显式 Exec Policy、危险命令启发式、已知安全命令和当前审批模式。结果被规整为三类：

结果	含义
`Skip`	不需要弹出审批；是否绕过沙箱由额外字段和权限策略决定
`NeedsApproval`	需要 Hook、自动 Reviewer 或用户给出决定
`Forbidden`	当前策略不允许执行，也不能通过询问来改变

例如审批模式为 `Never` 时，Runtime 不会询问用户，但这不代表所有命令都被允许：危险命令可以直接变成 `Forbidden`，普通命令则可以依赖受限沙箱运行。`OnRequest` 配合受限文件系统时，普通且未请求升级的命令通常直接进入沙箱；只有请求越过边界或命中策略时才需要审批。

显式 Allow 规则也有严格条件。只有解析出的每个命令片段都被 Exec Policy 明确允许，`Skip` 才可以携带 `bypass_sandbox=true`。模型给出的 `prefix_rule` 只是一个建议：Runtime 会拒绝过宽或不能覆盖整条命令的前缀，不能靠 `["bash"]` 这类泛化规则把任意脚本变成可信命令。

这里至少有四个不能合并的概念：

层	回答的问题	典型数据或组件
命令策略	这条命令应允许、询问还是禁止	Exec Policy、危险/安全启发式
审批流程	这一次具体动作由谁决定	Permission Hook、Guardian、User
权限配置	如果运行，允许读写和联网到哪里	`PermissionProfile`、附加权限
沙箱执行	怎样在操作系统或目标环境强制落实边界	`SandboxManager`、Exec Server

用户批准一次命令，是审批流程的结果，不会自动删除 Permission Profile；沙箱是执行机制，也不负责决定什么时候应该弹窗。

ToolOrchestrator 把审批、沙箱和升级组织成两次有界尝试

Shell、Unified Exec 和 Apply Patch 都把高风险执行交给 `ToolOrchestrator`。其正常顺序如下：

1. 用当前 Workspace Roots 物化 Permission Profile；
2. 计算 `Skip`、`NeedsApproval` 或 `Forbidden`；
3. 需要决定时，先运行 PermissionRequest Hook，再路由到 Guardian 或用户；
4. 根据文件、网络策略和工具偏好选择首个 Sandbox Attempt；
5. 执行并收集成功、普通错误或 Sandbox Denied；
6. 只有工具允许升级且策略允许时，才审批并执行第二次尝试。

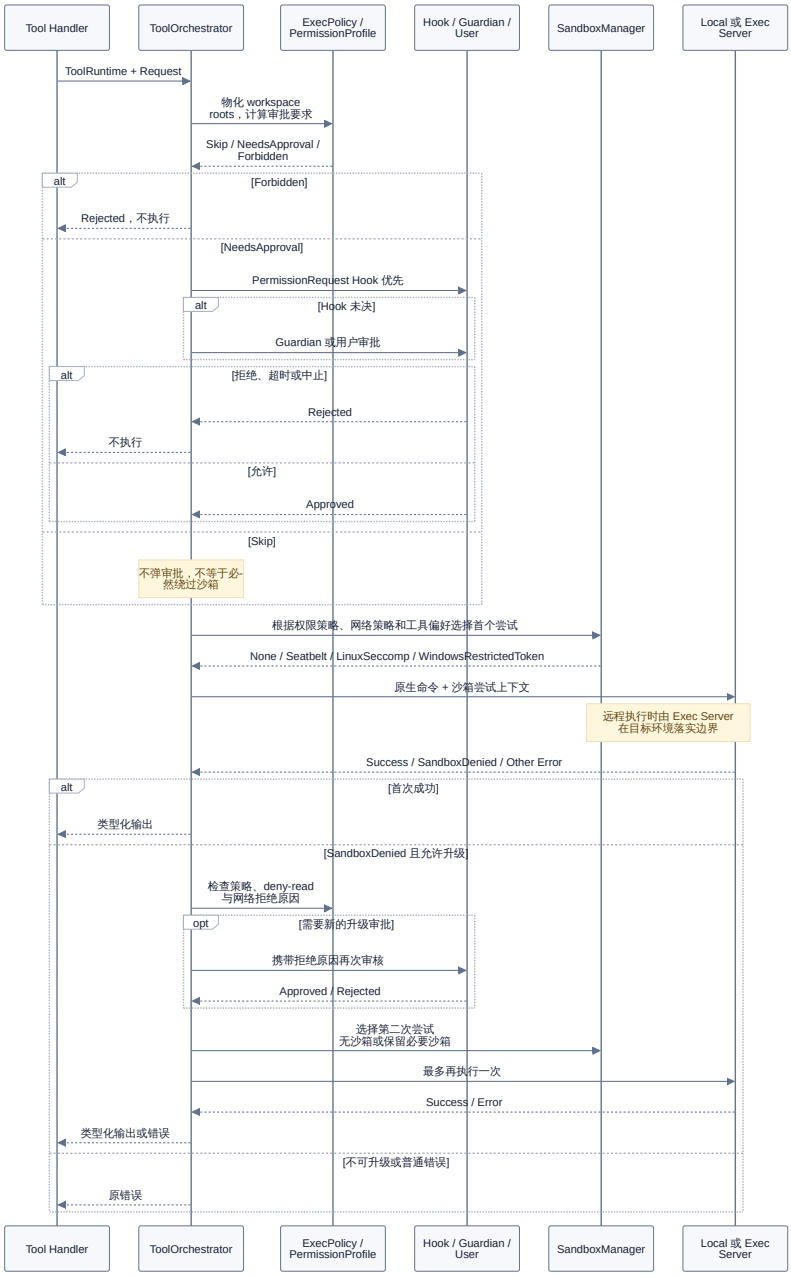

图 6-2：审批是决策，Permission Profile 是资源边界，Sandbox 是强制执行。首次沙箱拒绝只有在工具和策略都允许时才进入一次升级尝试。

`SandboxManager` 在当前平台选择具体实现：macOS 使用 Seatbelt，Linux 路径在类型中名为 `LinuxSeccomp` 并由 Codex Linux Sandbox 根据权限配置构造启动参数，Windows 可使用 Restricted Token；不需要或无法选择平台沙箱时为 `None`。

本地执行会把命令转换成平台对应的启动请求。远程执行不能让 Codex 宿主机用自己的路径和 Sandbox Wrapper 包住另一台机器的命令，因此 `env_for_exec_server` 保留原生命令，把 Permission Profile、Workspace Roots、cwd 和沙箱请求作为上下文发给 Exec Server，由目标环境落实边界。

Sandbox Denied 也不等于“自动去掉沙箱再跑一次”。Orchestrator 先检查：

- 该工具是否允许失败升级；
- 当前审批模式是否允许无沙箱审批；
- 拒绝是否来自可解释的网络策略；
- 文件策略是否含有 denied-read 限制；
- 首轮批准能否覆盖权限更大的第二轮。

denied-read 尤其关键。这类限制只能由文件系统沙箱执行。如果为了升级直接无沙箱重跑，原本禁止读取的路径反而全部暴露。因此只要存在 denied-read，Runtime 就认为无沙箱执行不能表示当前权限策略；即使命令规则允许或调用明确请求升级，也必须保留沙箱和拒读边界。

严格自动审查下，首轮的 Guardian 批准只覆盖沙箱内执行，改成无沙箱的第二轮需要新审核。其他模式可以在动作已经获得适当批准且没有网络升级时复用决定。无论哪种情况，当前 Orchestrator 的升级都是一次有界的第二次尝试，不会无限循环“失败—放宽—再失败”。

`apply_patch` 是结构化文件操作，不是把文本丢给 Shell

`apply_patch` 使用 Freeform Tool，因为补丁语法比嵌套 JSON 更适合模型生成：

```
*** Begin Patch
*** Update File: src/app.py
@@
-old_value = 1
+new_value = 2
*** End Patch
```

Handler 首先解析语法，再针对同一个 Step 选中的文件系统验证补丁。验证会确认目标文件、上下文和移动操作等信息，之后才评估权限并委托 Runtime 执行。直接工具调用和 `exec_command` 中识别出的 `apply_patch` 命令最终都会进入这条专用路径，后者不会真的启动一个 Shell 子进程来修改文件。

补丁审批按原子目标建立 Key：

```
ApplyPatchApprovalKey = environment_id + path
```

一次补丁可能有多个 Key；移动文件还会同时加入源路径和目标路径。只有所有 Key 都已经获得 Session 级批准，后续补丁才可跳过询问。若用户选择“本会话允许”，Runtime 分别缓存每个路径，因此以后只修改其中一个已批准文件也能命中缓存。

在 Workspace 内本来可写的目录不需要额外权限。目标父目录不在当前可写范围时，Handler 才构造附加写权限请求。真正应用补丁时，`ApplyPatchRuntime` 仍使用 Orchestrator 选定的 Sandbox Attempt 和环境文件系统；远程文件也通过同一抽象处理。

这种实现比 `echo ... > file` 复杂，但它知道“将修改哪些路径”，可以提前展示 Diff、按路径审批、追踪已提交 Delta，并在部分失败时把已经发生的变化交给生命周期事件。通用 Shell 无法可靠提供这些语义。

工具结果不是日志，它是下一次采样的输入

Handler 返回的是实现 `ToolOutput` 的类型，而不是随意向标准输出打印一段文字。不同工具会生成对应的 `ResponseInputItem`：

调用形状	回灌形状
Function Tool	`FunctionCallOutput`
Freeform Custom Tool	`CustomToolCallOutput`
Client Tool Search	`ToolSearchOutput`

结果保留原 `call_id`，模型因此能把输出与此前的调用配对。普通可恢复错误也会转成失败结果，例如不支持的工具、审批拒绝或参数错误；只有破坏 Runtime 契约的 Fatal Error 才终止整个采样链。

命令输出在进入模型上下文前还会受双重限制：Process Manager 先限制采集缓冲，`ToolOutput` 再按模型截断策略生成上下文文本。返回值可以携带原始 Token 估计、被省略的字节数、Chunk ID、退出码和仍存活的进程 ID。遥测预览也有独立上限，不能把“日志里只看到一小段”误判为 Handler 只产生了这一小段。

采样结束后，`drain_in_flight` 把每个结果转换成 `ResponseItem`，按调用顺序写入 History。外层 `run_turn` 看到 `needs_follow_up=true` 后，不是复用旧 Step，而是重新捕获 StepContext，再用已经包含 Tool Call 和 Tool Result 的 History 发起下一次采样。

主循环可以写成接近 Python 的伪代码：

```
async def run_agent_step(session: Session) -> bool:
    step = await session.capture_step_context()
    prompt = build_prompt(
        history=session.history.for_prompt(),
        tools=step.tool_router.model_visible_specs,
    )

    ordered_calls: list[Awaitable[ToolResult]] = []
    needs_follow_up = False
```

```
    async for event in model.stream(prompt):
        if isinstance(event, OutputItemDone):
            await session.history.append(event.item)
            call = step.tool_router.try_parse(event.item)
            if call is not None:
                ordered_calls.append(
                    step.tool_runtime.execute(call, session.cancel_token)
                )
                needs_follow_up = True

    for result in await_in_call_order(ordered_calls):
        await session.history.append(result.to_response_item())

    return needs_follow_up
```

这段伪代码省略了界面事件、Hooks、网络审批和错误分类，但保留了三个不变量：使用同一 Step 的规格和 Registry，调用项先于结果进入 History，工具结果只在下一次模型采样中生效。

调试工具链要先判断失败发生在哪一层

“命令没执行”不是一个足够精确的故障描述。可以按下面的信号定位：

现象	优先检查
模型始终不调用某工具	它是否为 Deferred、Hidden，或被 Code Mode/Feature Gate 隐藏
模型生成了工具名但提示 unsupported	调用是否来自旧 Step，Namespace 是否匹配 Registry
参数看似正确却触发 Fatal	ToolPayload 是 Function 还是 Custom，Handler 的 `matches_kind` 是否一致
调用在执行前被挡住	PreToolUse Hook、Exec Policy、审批模式与 Reviewer 决定
用户批准后仍被拒绝	Permission Profile 是否仍限制路径或网络，Sandbox 是否正确落实
沙箱拒绝后没有无沙箱重试	工具是否允许升级，审批策略和 denied-read 是否允许绕过
命令返回 `process_id` 但无退出码	进程仍存活，应通过 `write_stdin` 继续轮询或输入
第二个并行工具先完成却后出现	执行并行，但 `FuturesOrdered` 按调用顺序交付
Patch 移动目标没有获批	审批 Key 是否同时包含源路径与目标路径
模型只看到部分命令输出	区分采集上限、模型上下文截断和遥测预览

观察日志时还应把 `tool_name`、`namespace`、`call_id`、Step/Turn ID、环境 ID、审批来源、Sandbox Type、退出码和原始输出大小放在同一条 Trace 中。只记录 Shell 文本，无法解释它为什么走到了某个远程环境，也无法区分“策略拒绝”和“操作系统拒绝”。

工具执行形成受控闭环

一项 Tool Call 的完整执行链如下：

```
模型可见规格
→ 完整 Tool Call
→ 同 Step 路由
→ Hook 与 Handler
→ 命令策略和审批
→ Permission Profile 与 Sandbox
→ 本地或远程副作用
→ 类型化 Tool Result
→ History
→ 下一次采样
```

这条链的核心不是工具数量，而是四个边界始终可核对：规格与执行器来自同一 Step，副作用发生前有明确决策，批准和强制权限没有混为一谈，结果与原调用一起进入历史。

章内索引

Shell 与进程工具

- 6.20 `shell_command` 的模型规格和命令数组语义
- 6.21 Shell 参数解析、登录 Shell 与平台差异
- 6.22 ShellCommandHandler 的审批、执行和输出
- 6.23 Shell Runtime 的 Sandbox 选择与升级重试
- 6.24 Zsh Fork Backend 的会话复用机制
- 6.25 `exec_command` 的规格、yield time 和 session ID
- 6.26 Unified Exec ProcessManager 的进程注册表
- 6.27 子进程启动、stdout/stderr 异步读取和 HeadTailBuffer
- 6.28 命令首次 Yield、后台化与退出事件
- 6.29 `write_stdin` 写入、空轮询和会话关闭
- 6.30 Unified Exec 的超时、取消和进程回收
- 6.31 User Shell Command 为什么不经过模型 Tool Call

Apply Patch

- 6.32 Apply Patch Freeform Tool Spec 和 Lark Grammar
- 6.33 Patch 文本的 Add、Update、Delete 与 Move 语法
- 6.34 Parser 怎样定位 Hunk、上下文和文件操作
- 6.35 路径规范化、工作区边界和符号链接风险
- 6.36 ApplyPatchHandler 的审批和直接执行路径
- 6.37 Shell 中 `apply_patch` 命令的识别与截获
- 6.38 Patch 的文件写入顺序、失败原子性和错误报告
- 6.39 TurnDiffTracker 怎样记录改动并生成 Turn Diff
- 6.40 Apply Patch 流式参数预览和完成事件

图像、计划和交互工具

- 6.41 `view_image` 的环境文件读取与图像输入边界
- 6.42 原图 Detail 能力、尺寸限制和模型内容项
- 6.43 `update_plan` 的状态约束与 Event 更新
- 6.44 `request_user_input` 的模式门控、问题 Schema 和等待器
- 6.45 `request_permissions` 的权限请求、响应和 TurnContext 更新
- 6.46 `wait_for_environment` 的共享启动结果
- 6.47 `clock.curr_time` 的可替换时间源与输出视图
- 6.48 `sleep` 的取消和时间范围限制
- 6.49 `new_context` 怎样触发新上下文窗口
- 6.50 `get_context_remaining` 怎样读取 Token Budget

Tool Search、Code Mode 与扩展工具

- 6.51 Deferred Tool 的 SearchInfo 和延迟暴露
- 6.52 `tool_search` 的查询匹配、来源分组和返回规格
- 6.53 Tool Search Result 怎样进入下一次模型请求
- 6.54 Code Mode 的启用条件和工具面重写
- 6.55 Code Mode Execute 的 JavaScript Cell 生命周期
- 6.56 Code Mode Nested Tool 的名称规范化和桥接
- 6.57 Code Mode Wait 的 Cell 等待、取消与结果
- 6.58 Code Mode 与 DirectModelOnly/Excluded Namespace
- 6.59 Dynamic Function/Namespace Tool 的注册和执行
- 6.60 Extension ToolContributor 到 Core Runtime Adapter
- 6.61 Hosted Web Search 与 Standalone `web.run` 的选择
- 6.62 Image Generation Extension 的能力和账户门控

MCP 与插件工具

- 6.63 MCP Server 的工具目录、命名空间和旧名称兼容
- 6.64 MCP Function Call 的参数解析、审批和执行
- 6.65 MCP 结果中的文本、图片、资源和错误编码
- 6.66 MCP 输出截断、耗时信息和 Hook Payload
- 6.67 `list_mcp_resources` 的分页和 Server 过滤
- 6.68 `list_mcp_resource_templates` 的模板目录
- 6.69 `read_mcp_resource` 的 URI 路由和内容转换
- 6.70 `list_available_plugins_to_install` 的候选来源
- 6.71 `request_plugin_install` 的批准、安装和能力刷新

安全策略与系统沙箱

- 6.72 Permission Profile 从配置到 TurnContext 的解析
- 6.73 Approval Policy 的 Never、OnRequest 和升级语义
- 6.74 Exec Policy 怎样解析命令风险和保存前缀规则
- 6.75 Guardian 审查请求、策略 Prompt 与判定
- 6.76 SandboxPolicy、SandboxType 与执行器选择
- 6.77 macOS Seatbelt Profile 的文件和网络规则
- 6.78 Linux Landlock、Seccomp 与网络命名空间
- 6.79 Windows Sandbox Token、ACL 和进程限制
- 6.80 Network Proxy、域名审批和运行期网络规则
- 6.81 Sandbox Denied 的识别、升级和重试

- 6.82 审批、策略、沙箱和 Hook 各自能防住什么

延伸阅读

- ToolExecutor 与 ToolExposure
- ToolRouter 的工具调用解析与调度
- 工具规格与 Registry 的共同规划
- ToolRegistry 的 Hooks 与生命周期
- ToolOrchestrator 的审批、沙箱和升级
- Unified Exec Process Manager
- Apply Patch Handler

第七章　从一个 Agent 到一支小队

多 Agent 工具从同一条 Tool Router 进入 Runtime，但它产生的副作用不同于普通文件、进程或网络操作：`spawn_agent` 创建的不是一个后台函数，而是一条新的 Codex Thread。

这条 Thread 有自己的 Session、历史、Turn、模型采样和工具调用。父 Agent 可以继续当前工作，子 Agent 同时处理另一项任务；子 Agent 还可以继续派生下一层。它们共用工作目录，因此一个 Agent 写入的文件会立刻被其他 Agent 看见，但它们不会共用同一段模型上下文。

这组边界决定了 Codex 中“协作”的真实含义：共享外部世界，分离思考过程，通过显式协议交换任务和结果。

Agent 不是套在模型外面的一个新类

在当前实现中，找不到一个包办一切的 `Agent` 对象。一个可运行的 Agent 是几部分状态的组合：

```
Agent
├── ThreadId / AgentPath          身份和寻址
├── CodexThread + Session         操作入口与长期运行状态
├── SessionSource                 父 Thread、深度、角色、昵称
├── Config + Role                 模型、指令、权限和运行配置
├── History + Turn + Task         独立的思考与执行过程
└── shared AgentControl           派生、通信、容量和拓扑控制
```

因此，创建子 Agent 不能简化为“再调用一次模型”。新的模型请求必须附着在一条可持续运行的 Thread 上，否则它无法接收后续消息、被中断、恢复历史或再次执行任务。

父子 Agent 之间也不是对象字段直接调用。每条 Thread 的 Session 都持有一个 `AgentControl`，而同一根节点派生出的所有子 Agent 共享同一个控制实例。这个实例内部再连接 Agent Registry、V2 驻留管理器、执行容量限制器和整棵树共享的 Rollout 预算。

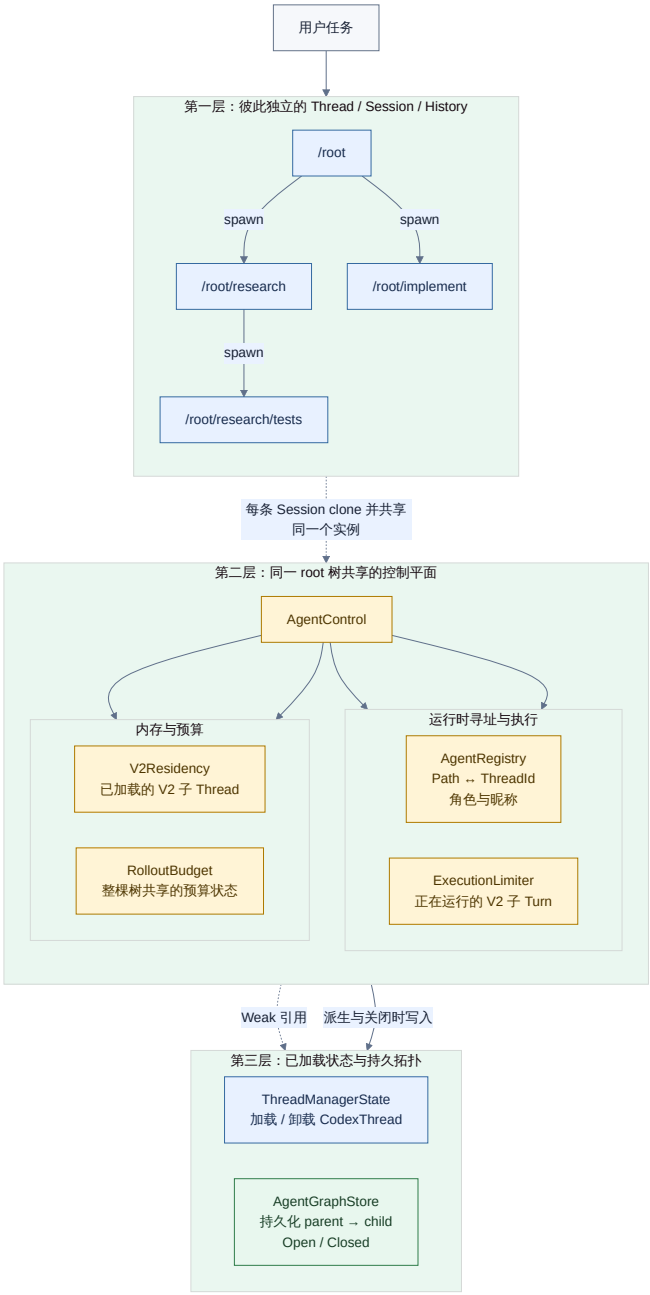

图 7-1：每个 Agent 独占 Thread、Session 和模型历史；同一 root 树共享 AgentControl。AgentRegistry 负责运行时寻址，ThreadManager 持有当前加载的 Thread，AgentGraphStore 另存可恢复的父子边。

`AgentControl` 只保存指向 `ThreadManagerState` 的弱引用。原因不是语法偏好，而是所有权边界：如果控制器反过来强持有全局 Thread Manager，就会形成 `ThreadManager → CodexThread → Session → AgentControl → ThreadManager` 的引用环，让已经关闭的会话继续存活，甚至造成影子持久化。

AgentPath 是任务树中的地址

V2 不再要求模型记住一串 UUID。根 Agent 的规范地址是 `/root`。如果它用任务名 `research` 派生子 Agent，新地址就是：

```
/root/research
```

`research` 再派生 `tests`，地址变成：

```
/root/research/tests
```

在 `/root/research` 内使用相对目标 `tests`，会解析到自己的子节点；若要联系兄弟节点 `/root/implement`，就要给出完整地址。路径段只允许小写字母、数字和下划线，`root`、`.`、`..` 以及含 `/` 的任务名都会被拒绝。

这不是展示层的别名。`AgentRegistry` 同时维护 `AgentPath → ThreadId` 与 `ThreadId → AgentPath`，消息工具先解析路径，再把操作提交给目标 Thread。Spawn 还会提前预留路径，两个并发调用不能侥幸创建出同名子节点。

`SessionSource::SubAgent(ThreadSpawn)` 则把关系写入子 Session：父 Thread ID、派生深度、AgentPath、昵称和角色都随 Thread 保存。AgentPath 负责可读寻址，ThreadId 负责稳定身份，二者不能互相替代。

Spawn 是一项分阶段提交协议

`spawn_agent` 的 Handler 先把模型参数转成子 Agent 配置和派生来源，再交给 `AgentControl`。控制层按下面的顺序工作：

1. 判断当前会话实际使用 V1 还是 V2；
2. 检查这次派生是否还有执行容量；
3. V2 预留一个驻留槽，V1 预留总 Thread 名额；
4. 在 Registry 中预留任务路径和昵称；
5. 从父 Session 继承环境快照，以及允许继承时的 Exec Policy；
6. 创建全新 Thread，或从父 Rollout 派生历史；
7. Thread 成功后才提交 Registry 和驻留预留；
8. 尽力把父子边以 `Open` 状态写入 AgentGraphStore；
9. 向子 Session 发送第一条 `InterAgentCommunication`，并触发它的第一个 Turn。

路径、昵称和容量预留都由带 `Drop` 清理语义的 Reservation 管理。Thread 身份提交之前，创建、历史加载或配置应用失败会让未提交的预留自动归还；身份提交之后，Edge 写入或首个任务投递失败不会回滚已经存在的 live Thread。这里的回滚边界只覆盖预留阶段，不能扩展成整个 Spawn 的原子性保证。

用接近 Python 的伪代码表达，这项事务大致是：

```
async def spawn_agent(request: SpawnRequest) -> AgentHandle:
    check_execution_capacity(request.parent)
```

```
    async with residency.reserve() as resident_slot:
        async with registry.reserve(request.task_name) as agent_slot:
            child_config = inherit_runtime_config(request.parent)
            history = await build_child_history(request.fork_turns)
            child = await thread_manager.create_thread(
                config=child_config,
                history=history,
                source=request.thread_spawn_source,
                agent_control=shared_agent_control,
            )

            agent_slot.commit(child.thread_id)
            resident_slot.commit(child.thread_id)
            try:
                await graph.upsert(request.parent.thread_id, child.thread_id, "open")
            except GraphStoreError as error:
                log_warning(error)
            await child.mailbox.send(request.initial_task, trigger_turn=True)
            return AgentHandle(child.thread_id, agent_slot.agent_path)
```

伪代码合并了 V1/V2 分支，但保留了三个提交阶段：先预留，创建成功后提交身份，最后尽力写入拓扑并投递首个任务。最后两个动作失败时，已提交身份仍然存在。

Fork 复制的是可用上下文，不是父 Agent 的全部内部轨迹

V2 的 `fork_turns` 有三种取值：

参数	子 Agent 获得的父历史	适用情况
`none`	不复制对话历史	任务自包含，只需要初始消息和共享工作区
`all`	复制经过清洗的完整有效历史；也是默认值	子任务依赖当前讨论的来龙去脉
正整数，例如 `3`	只保留最近三个有效 Turn，再清洗	需要近期上下文，但不想携带整段会话

“完整历史”仍不是字节级克隆。派生前，父 Thread 会先把异步 Rollout 写入刷新到存储；截取所需 Turn 后，Runtime 再按模型上下文规则过滤条目：

- 保留 System、Developer、User 消息；
- Assistant 消息只保留最终回答阶段；
- 丢弃 Reasoning、函数调用、工具输出、Tool Search 和 Shell 调用等中间轨迹；
- 丢弃旧的 Agent 间通信，避免把发给父 Agent 的消息误当成子 Agent 的新任务；
- 保留 Compaction 和必要的 Session 元数据；
- 只有安全的全量 Fork 才沿用父 Thread 的 TurnContext、WorldState 参考基线，截断 Fork 在第一次子 Turn 中重新构造动态上下文。

这样处理同时服务两个目标。第一，子 Agent 能理解用户目标和父 Agent 已经确认的结论；第二，它不会继承一批已经完成的工具调用，再次面对无法配对的 Call/Result，或把父 Agent 的隐藏推理当成自己的待执行计划。

角色指令也不能简单复制。V2 会移除父 Agent 的协作提示，并在可识别时把父 Developer Instructions 片段替换成子 Agent 指令。压缩历史中的 replacement history 也使用同一清洗规则。若 `fork_turns=all`，源码禁止额外覆盖 `agent_type`，因为全量参考上下文与另一套角色指令可能产生冲突；`none` 或最近 N 个 Turn 的 Fork 才允许显式选择另一角色。

子任务越独立，越适合 `none`；越依赖父 Agent 刚刚建立的概念和约束，越需要最近 N 个 Turn 或 `all`。Fork 越多并不天然越好，它会增加上下文成本，也会扩大子 Agent 需要重新辨认的历史范围。

消息进入 mailbox，是否启动 Turn 由一个位决定

V2 的通信协议不是“向目标 Prompt 追加字符串”。`InterAgentCommunication` 至少携带作者路径、接收者路径、内容和 `trigger_turn`。`send_message` 与 `followup_task` 走同一个提交函数，关键差异只有投递模式：

工具或事件	`trigger_turn`	目标空闲时	典型用途
Spawn 的初始任务	`true`	立即启动第一个 Turn	创建并开始工作
`send_message`	`false`	只进入 mailbox，不开始采样	给正在工作的 Agent 补充信息
`followup_task`	`true`	启动新 Turn	让已完成或空闲的 Agent 继续任务
子 Agent Result	`false`	只进入父 mailbox	把完成结果交还父 Agent

目标 Agent 正在运行时，消息先进入 Session 级 InputQueue。Runtime 会在可接收输入的消息边界，或当前工具调用完成后，把待处理内容交给活动 Turn；若当前阶段不允许同 Turn 投递，就留到下一 Turn。`followup_task` 还能在目标空闲时启动新 Turn，`send_message` 不能。

这个区别避免了两种常见浪费。补一句“测试文件在 `tests/runtime.rs`”不值得额外发起一次模型请求；而“现在根据审查结果修复问题”显然需要目标 Agent 再运行。是否唤醒由发送方明确表达，不让接收方靠猜测决定。

V2 还会在投递前调用 `ensure_v2_agent_loaded`。如果目标此前因驻留容量不足被卸载，Runtime 会从已存储 Rollout 恢复原 ThreadId、SessionSource、角色和历史，再把通信放进 mailbox。对发送方来说，目标仍由同一个 AgentPath 寻址，不需要显式执行 `resume_agent`。

子 Agent 的最终回答怎样回到父 Agent

子 Agent 每次发出事件时，Session 都会更新 `AgentStatus`：

```
TurnStarted                          → Running
TurnComplete(last_agent_message)     → Completed(message)
```

```
TurnAborted(Interrupted/Budget)     → Interrupted
其他 TurnAborted / Error            → Errored(reason)
ShutdownComplete                    → Shutdown
```

`Interrupted` 在这里不是最终状态。它表示当前 Turn 停下了，但这条 Agent Thread 还能收到后续任务。`Completed` 也只是“这一轮已经给出可交付结果”，不等于 Thread 被销毁；一次 `followup_task` 可以让它重新进入 `Running`。

V2 子 Session 遇到 `TurnComplete` 或 `TurnAborted` 时，会检查状态是否真正终结。如果是 `Completed`、`Errored` 或 `Shutdown`，它把最后回答或错误包装成标准 Result 通信，然后投递到直接父 Session。这个 Result 的 `trigger_turn=false`，所以它不会为了宣布完成而强行启动父 Agent 的新 Turn。

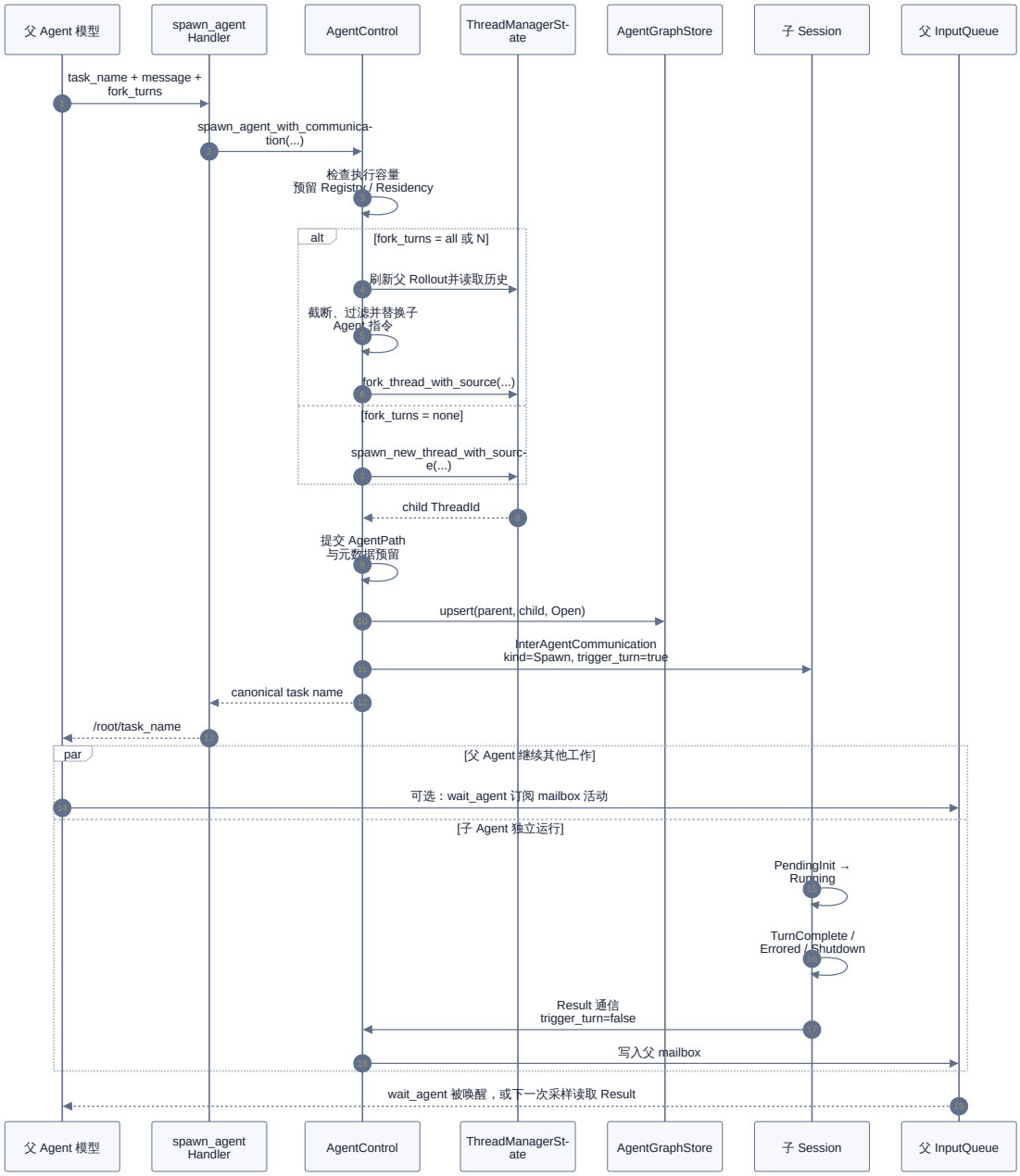

图 7-2：Spawn 的初始通信会唤醒子 Agent，完成 Result 只写入父 mailbox。wait_agent 订阅的是 mailbox 或新输入活动，不直接轮询指定子 Agent 的状态。

这里最容易被工具名称误导。V1 的 `wait_agent(targets)` 会订阅指定 Thread 的状态，直到其中至少一个进入最终状态；V2 的 `wait_agent` 没有目标参数，它等待当前父 Session 出现两类活动之一：

- mailbox 收到 Agent 消息或结果；
- 用户输入对当前 Turn 进行了 Steer。

超时只表示这段时间没有活动。`wait_agent` 返回的也是“完成、被新输入打断或超时”，真正的子 Agent Result 仍是 mailbox 中的模型输入。这样 V2 不需要让父 Agent 同时维护一份“我正在等哪些 UUID”的列表；任务地址和消息信封已经给出了来源。

因此不能把 `wait_agent` 当成必须调用的 Join。父 Agent 完全可以 Spawn 后继续检查其他文件，子 Agent 完成时 Result 自然进入 mailbox。只有下一步确实依赖结果、父 Agent 已没有别的可推进工作时，等待才有意义。

Interrupt、Close 和卸载是三种不同操作

“让 Agent 停下”至少有三种语义：

操作	当前 Turn	Thread 是否保留	父子边	能否继续
`interrupt_agent`	发送 `Op::Interrupt`，状态可变为 `Interrupted`	保留	保持 `Open`	可以接收新任务
V1 `close_agent`	Flush 后 Shutdown，并清理活动后代	从内存 Registry/Manager 移除	标记为 `Closed`	需按已存 Rollout 显式 Resume
V2 Residency 卸载	只选择无活动 Turn、无 mailbox 的终态/中断 Thread	从 ThreadManager 卸载，身份元数据保留	保持 `Open`	下次消息前自动恢复

V2 暴露的协作工具里有 `interrupt_agent`，没有 V1 的 `close_agent` 与 `resume_agent`。这不是少做了生命周期管理，而是把“暂时不占内存”和“从协作拓扑明确关闭”分开：普通空闲 Agent 由 Residency 自动换出和恢复；中断只停止当前工作；持久边的关闭仍是更强的内部生命周期操作。

关闭一个父 Agent 时，控制层还会遍历当前内存中的派生后代并逐一 Shutdown，防止留下失去控制入口的活动 Thread。中断不会级联，也不会释放路径或容量计数。

并发槽和驻留槽限制的不是同一件事

多 Agent 最容易出现的错误模型是“配置最多四个 Agent，所以第五个一定不能存在”。V2 实际把资源分成两个维度。

第一个维度是执行容量。只有正在运行 Turn 的 V2 子 Agent 占用 `AgentExecutionGuard`；根 Agent不计入这个 Guard。Guard 与 `RunningTask` 生命周期绑定，Task 完成或取消后通过 `Drop` 归还。`UserInput` 和 `trigger_turn=true` 的通信在启动空闲 Turn 前都要检查容量；只入队的 `send_message` 不消耗新的执行槽。

第二个维度是驻留容量。它限制当前加载在 ThreadManager 中的 V2 子 Thread 数量。容量用尽时，Residency 按近似 LRU 顺序寻找可卸载对象，但候选必须同时满足：

- 状态为 `Completed`、`Errored` 或 `Interrupted`；
- 没有活动 Turn；
- mailbox 中没有待处理消息。

卸载前必须物化并刷新 Rollout，再 Shutdown Session、移出 ThreadManager。Registry 中的 AgentPath 和持久化 `Open` 边不因此消失，所以后续通信仍能用原路径恢复它。

两种容量解决不同问题：

```
Execution capacity = 同时有多少个子 Agent 正在花模型和工具计算
Residency capacity = 同时有多少条子 Thread 占用运行时内存
Persistent graph   = 总共有多少条仍可恢复的协作关系
```

如果所有执行槽都在 Running，新 Spawn 或 Followup 会被拒绝，即使某个已加载 Thread 理论上可以卸载；如果只是驻留满了，但存在干净的已完成 Agent，Runtime 可以先卸载它，再创建或恢复另一条 Thread。

V1 的限制更直接：Registry 用总 Thread 计数和 `agents.max_depth` 限制派生。默认深度为 1，达到深度后连 Spawn 工具都不会继续暴露。V2 的工具规划明确忽略这项 V1 深度上限，允许子 Agent继续派生；它依靠执行容量、驻留容量、共享预算和调用规范约束资源。因此在解释配置时，必须先确认会话运行的是 V1 还是 V2。

Registry 管现在，Graph Store 管重启以后

AgentRegistry 与 AgentGraphStore 都保存“Agent 关系”，但权威范围不同。

Registry 是一棵 root 会话树共享的内存索引。它负责路径解析、ThreadId 反查、角色、昵称、并发 Spawn 预留和运行时计数。V2 Residency 卸载 Thread 时不会释放这份身份，因此消息仍能找到待恢复的目标；真正 Shutdown 或关闭才会释放运行时注册。

AgentGraphStore 是存储中立的持久化边界，只记录定向父子关系与 `Open/Closed` 状态：

```
parent_thread_id ──Open──> child_thread_id
parent_thread_id ─Closed─> child_thread_id
```

`Open` 表示子 Thread 仍处于活动或可恢复的协作图中，`Closed` 表示这条入边已经被明确关闭。它不保存当前是否 Running，也不替代 AgentStatus。查询后代时，状态过滤会应用到遍历的每条边；关闭父边后，不能绕过它继续把下层节点当成开放后代。

V1 从 Rollout 恢复一棵 Agent 树时，会沿 Graph Store 的开放边递归恢复后代。V2 则更多采用按需恢复：Registry 提供地址，存储提供历史，`ensure_v2_agent_loaded` 在通信前重新加载目标。

共享工作区不等于共享记忆

同一棵 Agent 树通常使用相同 cwd、文件系统和工具能力。父 Agent 派出一个实现者和一个测试者时，两者可以同时读同一仓库，也可能同时修改同一文件。Codex 没有因为它们属于一支小队，就自动提供数据库事务或文件锁。

这带来两条工程约束：

1. 子任务的写入范围应尽量不重叠；否则最后写入者可能覆盖另一方的改动，父 Agent 还要额外合并冲突。
2. 共享文件变化不等于共享语义。一个 Agent 新建了 `src/cache.rs`，另一个 Agent 能在文件系统中看到它，但除非重新读取文件或收到消息，它的模型上下文并不知道为什么要这样设计。

反过来，Agent 的 History、InputQueue 和活动 Turn 都是独立的。发送消息会形成有作者和接收者的通信记录，不会把发送方整段对话偷偷合并进接收方 History。Fork 只发生在创建时，而且使用前述清洗策略。

这种“外部状态共享、内部上下文隔离”比共享一条长对话更容易控制 Token 和故障范围，代价是任务拆分与结果整合必须显式完成。

Role、Skill 和 MCP 不是三种 Agent

多 Agent 机制经常与扩展能力混在一起讨论，但它们改变 Runtime 的位置不同：

机制	改变什么	是否创建独立 Thread	是否有独立状态与 mailbox
Agent Role	Spawn 时叠加子 Agent 配置、模型或指令	角色本身不创建；由 Spawn 创建	取决于对应子 Agent
Skill	向当前 Agent 注入一套任务说明和资源	否	否
MCP / Plugin / App	给当前 Runtime 增加工具、资源或能力入口	否	否
Multi-agent Spawn	创建新的 Thread、Session、历史和任务循环	是	是

Role 回答“这条新 Thread 应按什么配置工作”，Skill 回答“当前 Agent 完成某类任务应遵守什么流程”，MCP 和 Plugin 回答“当前 Runtime 还能调用什么能力”。只有 Spawn 改变 Agent 树的拓扑。

这也是为什么“安装更多 Skill”不能替代并行 Agent，而“多开几个 Agent”也不会自动获得新的外部系统能力。前者扩充同一个执行者的方法，后者增加彼此隔离的执行者。

调试协作链要同时看地址、状态和队列

多 Agent 故障通常不是一句“子任务没回来”能够定位的。可以按现象拆开检查：

现象	优先检查
Spawn 提示路径已存在	当前 AgentPath 下是否重复使用 task_name，失败预留是否已释放
子 Agent 缺少背景	`fork_turns` 是否为 `none`，初始 message 是否自包含
Fork 后出现旧工具噪声	是否错误绕过了 Rollout 清洗，Call/Result 是否仍成对
`send_message` 后目标没有开始工作	该工具本来就不触发 Turn；需要任务时用 Followup
`wait_agent` 很快返回但没有指定状态表	当前是否为 V2；它等待 mailbox/Steer，不是 V1 目标状态订阅
完成 Result 没有唤醒空闲父 Agent	Result 的设计就是 queue-only；检查 mailbox，或让父 Turn主动等待
Agent 显示 Interrupted 后仍可用	Interrupted 不是最终状态，也没有关闭父子边
路径能解析但 Thread 不在 Manager	可能被 Residency 卸载，应检查自动恢复和 Rollout 读取
达到上限但 Agent 总数看起来不多	区分正在 Running 的执行槽与已加载 Thread 的驻留槽
两个 Agent 的代码互相覆盖	文件系统共享但无自动写冲突协调，重新划分写入范围

日志至少要关联 root Session ID、ThreadId、AgentPath、parent ThreadId、Turn ID、通信 ID、`trigger_turn`、AgentStatus，以及 Registry/Residency 的变化。只记 Agent 昵称不够：昵称可以重复轮换，AgentPath 才是 V2 工具使用的规范地址。

一支小队仍然由同一个 Runtime 约束

Codex 的多 Agent 没有绕开基础 Runtime 机制。Spawn、Message 和 Interrupt 首先仍是 Tool Call；每条子 Thread 仍按 Turn 和 Step 运行；工具仍受原有权限与沙箱约束；结果仍要写进 History 才能影响后续采样。

新增的只是另一层控制协议：

```
Tool Call
→ 预留路径、容量与驻留槽
→ 创建或 Fork 子 Thread
→ 持久化 Open 父子边
→ trigger_turn 任务通信
→ 子 Agent 独立运行
→ 终态 Result 写入父 mailbox
→ 父 Agent 整合结果
```

这条链中最重要的分界有四条：Agent 共享工作区但不共享模型历史；Full Fork 也要清洗上下文；消息投递与启动 Turn 是两个动作；运行时 Registry、加载中的 Thread 和持久拓扑分别回答不同问题。

章内索引

- 7.3 AgentRegistry 的注册、活动索引与 SpawnReservation
- 7.4 AgentStatus 的事件投影、终态和 Interrupted 语义
- 7.5 Spawn 深度、注册容量、执行容量与驻留容量
- 7.6 V2 Residency 的 LRU、Evict、Resume 与冷 Thread
- 7.7 Spawn 前的角色解析、模型选择和子 Thread 创建事务
- 7.8 Fork History 怎样过滤 Reasoning、Tool Call 和旧通信
- 7.9 父 Agent 指令怎样替换为子角色指令
- 7.10 `spawn_agent` V1 的参数、创建与 UUID 返回
- 7.11 `send_input` V1 的 interrupt、queue 与新 Turn 选择
- 7.12 `resume_agent` V1 的 Rollout 恢复与开放后代
- 7.13 `wait_agent` V1 的多目标状态订阅与超时
- 7.14 `close_agent` V1 的边关闭、后代 Shutdown 与资源释放
- 7.15 `spawn_agent` V2 的 AgentPath、fork_turns 与元数据隐藏
- 7.16 `send_message` 的 QueueOnly Mailbox 投递语义
- 7.17 `followup_task` 怎样向空闲 Agent 启动新 Turn
- 7.18 `wait_agent` V2 等待 Mailbox/Steer Activity 与超时范围
- 7.19 `interrupt_agent` 的目标校验、previous status 与幂等取消
- 7.20 `list_agents` 的路径前缀、树形投影与 loaded-only 快照
- 7.21 Result Mailbox 的终态通知、QueueOnly 与父 Agent 唤醒
- 7.22 AgentGraphStore 的唯一入边、Open/Closed 与广度优先恢复
- 7.23 V1 与 V2 协作协议的工具、地址、等待和恢复差异
- 7.24 Skill 根目录、层级优先级与有界发现算法
- 7.25 Skill 元数据怎样进入 Prompt，全文怎样按需注入
- 7.26 Skill 的 scripts/assets 读取边界与 MCP 依赖安装
- 7.27 MCP ConnectionManager 的并发启动、连接复用、认证与关闭
- 7.28 MCP 工具目录刷新、Binding 快照与 Catalog Revision
- 7.29 MCP Resource、Resource Template 与 Elicitation 的边界
- 7.30 Plugin Manifest、Bundle、Marketplace 与路径解析
- 7.31 Plugin 安装、升级、删除与启动同步的事务边界
- 7.32 App MCP Routing、Backend Auth 门控与同名 Server 去重
- 7.33 Hooks 配置发现、Trust、Matcher 与 Command Runner
- 7.34 Session、Prompt、Tool、Compact 与 Subagent Hook 事件负载
- 7.35 Hook 输出解析、Additional Context、输入改写与阻断结果
- 7.36 ExtensionRegistry 与 Session、Thread、Turn、Step Data
- 7.37 ContextContributor 的 Prompt Slot、Turn Context 与 WorldState Section
- 7.38 Tool、MCP、TurnInput 与 TurnItem Contributor 的接入点

- 7.39 Thread 与 Turn Lifecycle Contributor 的调用时机和状态清理
- 7.40 Hooks 与进程内 Extension 的信任、能力和失败边界

延伸阅读

- AgentControl 与共享控制平面
- Spawn、Fork 与 V2 自动恢复
- AgentRegistry 与 Spawn Reservation
- V2 执行容量与 Residency
- V2 消息工具和 mailbox
- AgentGraphStore 持久拓扑接口

第八章　工作怎样被记住：持久化、恢复与回滚

一条子 Agent Thread 可以被卸载，随后又以同一个身份恢复。进程内的 `Session` 已经消失，恢复依据来自进程外保存的规范记录。

最容易产生的误解，是把答案想成“Codex 定期把 Session 保存到数据库”。当前实现并不是对象快照系统。Codex 会把足以重建会话语义的记录按顺序写入 Rollout；恢复时创建新的 Session，再重放这些记录，重新计算模型历史、Turn 设置、World State 和上下文窗口。

持久化系统必须分开三条边界：什么是规范历史，什么只是查询投影，什么副作用根本不属于会话历史。它不能抽象成一个 `save()` 和一个 `load()`。

章内索引

- 8.24 不完整 Turn 和 Prompt-only Call/Output 修复
- 8.25 Compact Checkpoint 的 Replacement History
- 8.26 Local/Remote Compact 在恢复语义上的差异
- 8.27 Legacy Rollback Marker 的追加式回退
- 8.28 Copied Fork 的历史筛选和新 SessionMeta
- 8.29 Reference-backed Fork 的冻结前缀
- 8.30 Rollout Lineage 的祖先拼接和删除约束
- 8.31 Archive、Delete 与被引用历史
- 8.32 State DB 中 Thread、Goal、Memory 和 Log 的职责
- 8.33 AgentGraphStore 恢复与 Thread History 恢复的协作
- 8.34 崩溃点矩阵：模型、工具、日志和投影分别可能留下什么

先分清五种看起来都像“记忆”的东西

一条 Thread 运行时，会同时碰到几种状态：

名称	保存什么	主要读者	是否是重放权威
Context History	当前要交给模型的 `ResponseItem`	下一次模型请求	否，只存在于运行中的 Session
Rollout	经过持久化策略筛选的有序 `RolloutItem`	Resume、Fork、Rollback、诊断	是，本地实现的规范重放日志
ThreadStore	创建、追加、刷新、读取、派生 Thread 的接口	Core Runtime	它是边界，不是某一种文件格式
SQLite State / History Projection	Thread 列表、预览、cwd、Git 信息、Turn/Item 索引	列表、搜索、分页读取	否；部分元数据可修复，History Projection 可重建
AgentGraphStore	`parent_thread_id → child_thread_id` 及 Open/Closed	多 Agent 恢复	只对 Agent 拓扑权威

`Event Stream` 也不能等同于 Rollout。模型文字增量、命令输出增量、审批请求、警告和界面生命周期事件要及时送给客户端，但没有必要全部成为未来模型上下文的一部分。Rollout 的目标是重建，不是录屏。

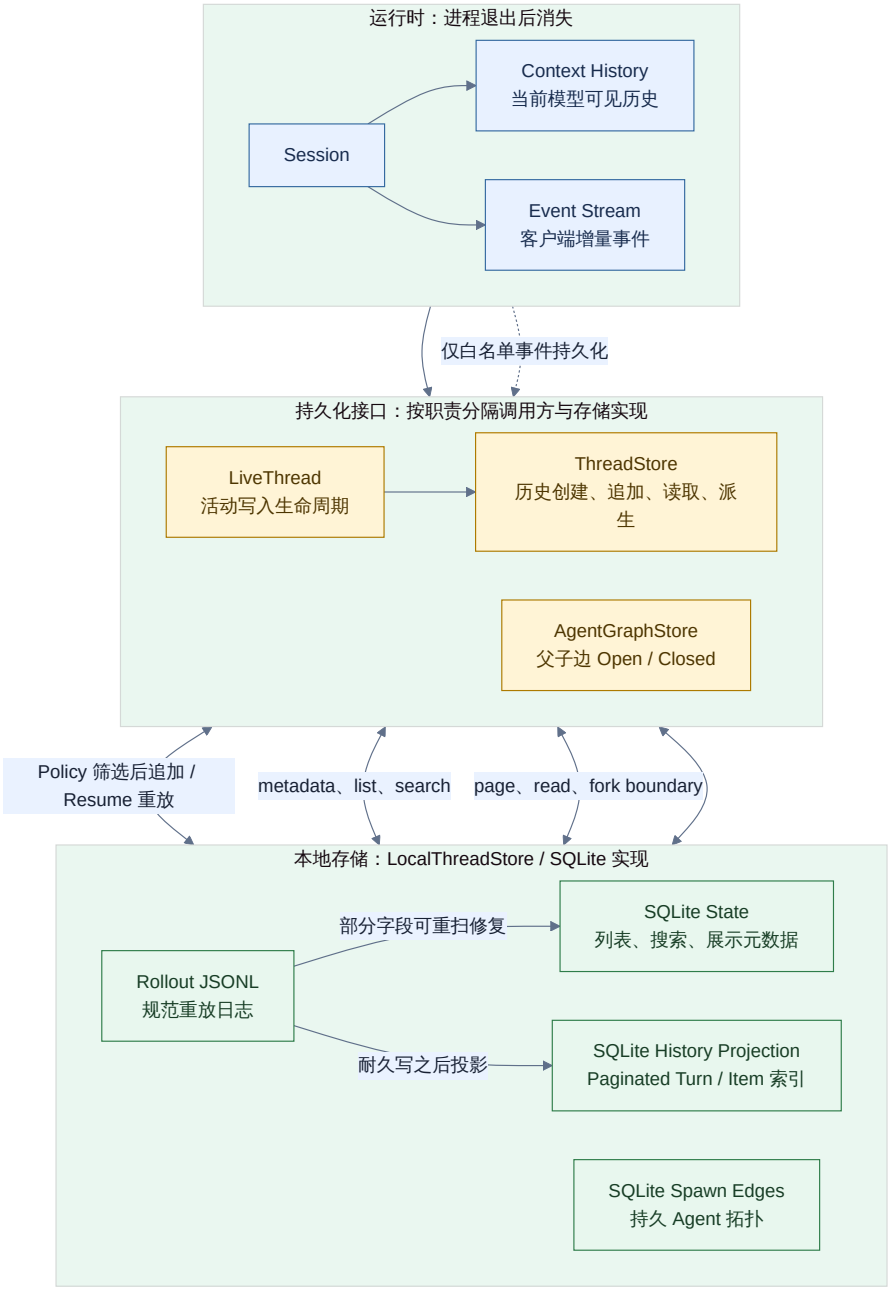

图 8-1：Rollout JSONL 保存规范重放记录；SQLite 保存便于查询的元数据和 Paginated 历史投影；AgentGraphStore 只保存父子拓扑。客户端见过的瞬时事件并不会全部进入 Rollout。

这张图也解释了为什么数据库中缺一行和 Rollout 丢一行不是同等级故障。前者可能让列表暂时显示旧预览，随后可以通过扫描日志修复；后者可能改变恢复出来的模型历史。

Rollout 是经过筛选的追加日志

持久化的基本单位是 `RolloutItem`。它不是只有聊天消息，当前协议还包括：

```
RolloutItem
├── SessionMeta
├── ResponseItem
├── InterAgentCommunication / Metadata
```

```
├── Compacted
├── TurnContext
├── WorldState
└── EventMsg
```

这些类型分别回答不同问题。`ResponseItem` 重建模型见过的消息、Reasoning、Tool Call 和 Tool Output；`TurnContext` 恢复上一个真实 Turn 的模型、配置哈希和 Realtime 状态；`WorldState` 恢复动态环境基线；`Compacted` 表示历史曾被一个替代历史接管；Turn Started、Complete、Aborted 等少量 `EventMsg` 则给出回合边界。

写入前，Persistence Policy 会再次筛选条目。Function Call、Tool Output、Shell Call、最终消息和 Compaction 等需要重放的 `ResponseItem` 会保留；`AdditionalTools`、`CompactionTrigger` 等运行中辅助项不保留。Event 的筛选更明显：

- `TurnStarted`、`TurnComplete`、`TurnAborted`、Token Count 和 Rollback Marker 会持久化；
- Paginated 模式用 `ItemCompleted(TurnItem)` 保存结构化项目；Legacy 模式保留一组旧事件；
- 文本 Delta、Reasoning Delta、命令输出 Delta、审批请求、Warning、Error 和大量 Begin 事件是瞬时信号，不进入规范 Rollout。

这里有一个容易忽略的后果：不能仅凭 Rollout 逐字重放用户曾经看到的界面，也不能拿 Event Stream 直接喂给模型。两条流部分重叠，但服务的是不同消费者。

Rollout 使用 JSONL，每一行是一条带时间戳的记录。Paginated 模式还写入单调递增的 ordinal，使一段历史可以用“结束序号 + 字节偏移”表达稳定边界。冷日志可以压缩成 `.jsonl.zst`；恢复读取会透明处理压缩格式，需要继续追加时再物化回普通 `.jsonl`。

写入顺序比“用了数据库”更重要

当 Session 收到一个新的模型消息或工具结果时，它先更新自己的 Context History，然后通过 `LiveThread` 把原始 `RolloutItem` 交给 `ThreadStore`。本地 Store 对同一 Thread 串行化写入，应用持久化策略，再交给 `RolloutRecorder`。

`RolloutRecorder` 背后有一个专用写任务和 `pending_items` 队列。普通 append 先排队；`flush()` 才是等待此前写入完成的屏障。新建而尚无内容的 Thread 还可以延迟创建文件，直到出现实际记录或调用 `persist()` 明确要求物化。

三种操作因此不能混为一谈：

操作	语义
`append_items`	按顺序增加新的规范记录，并由 LocalThreadStore 完成一次 flush
`persist_thread`	即使还没有 Turn 内容，也把延迟状态和 SessionMeta 物化
`flush_thread`	等待已经排队的记录越过写入屏障，不凭空增加业务记录

如果第一次写失败，Recorder 会丢弃文件句柄、重新打开再试一次。只有成功写出的前缀才会从 `pending_items` 删除，未写出的后缀仍留在内存中，后续 `persist`、`flush` 或 `shutdown` 可以继续重试。这避免了“调用方收到错误，于是整批再写一次”造成的前缀重复。

Paginated 模式还要把 JSONL 物化成 SQLite 中的 Turn/Item 投影。这里的顺序是硬约束：

```
先完成 JSONL 写入屏障
再更新 SQLite Projection
```

投影失败会记录警告，但不会让已经成功的规范写入倒退。因此 SQLite 可以落后于 JSONL，却不允许领先于 JSONL。元数据同步也位于规范历史追加之后：只有这批历史成功写入，`LiveThread` 才根据它更新 preview、首条用户消息、cwd、Git 信息和最近活动时间。

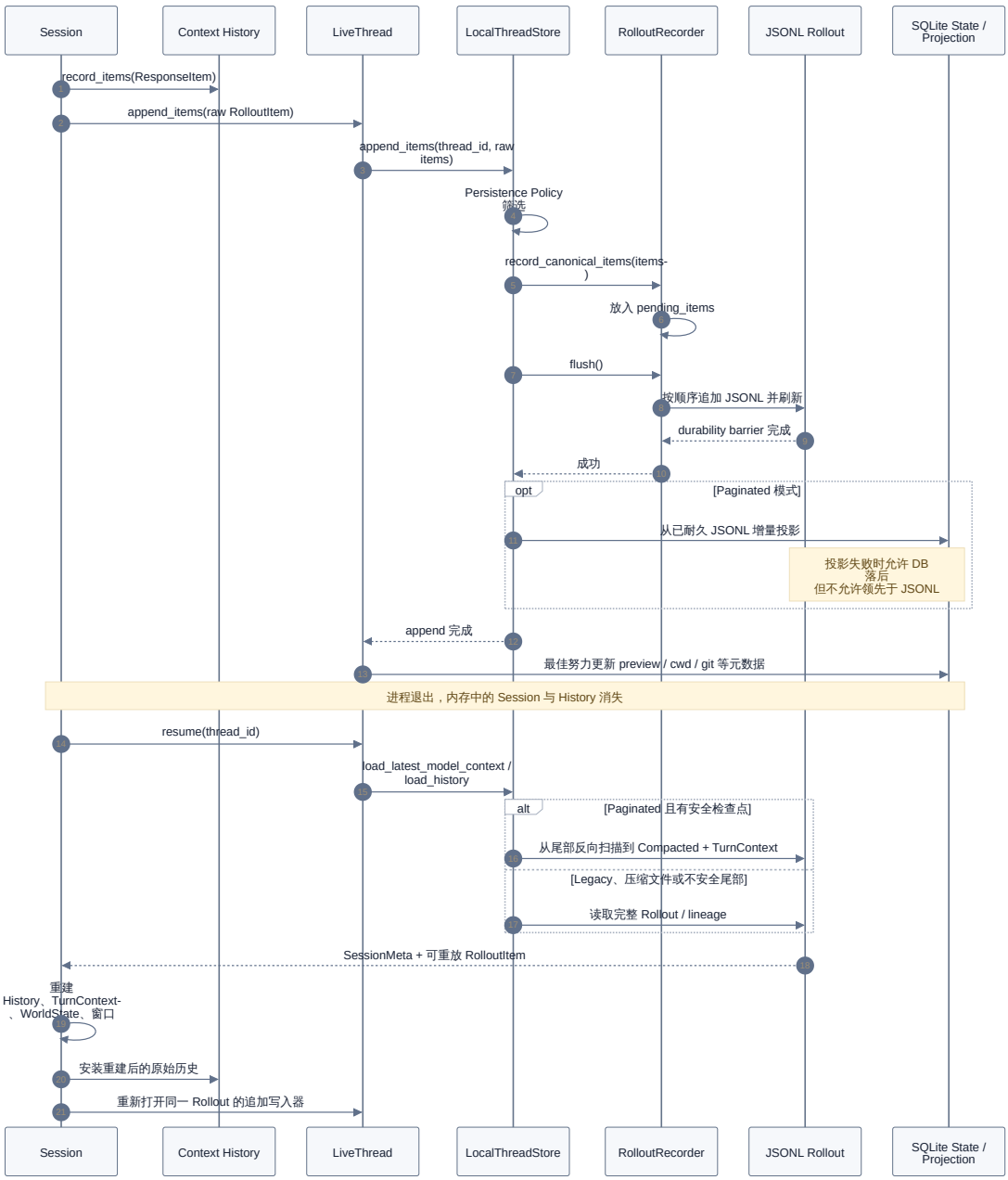

图 8-2：写入的权威顺序是内存历史、规范 JSONL、SQLite 投影；恢复则从 ThreadStore 读取可重放项，构造新的 Session 状态。SQLite 投影不参与决定 JSONL 已经发生过什么。

Turn 的终止边界还会额外刷新。正常 Task 结束时，Runtime 先刷新主体记录，再追加 `TurnComplete`，然后再次刷新；中断路径会先写入模型可见的 interrupted marker，再持久化 `TurnAborted` 并刷新。正常写入成功时，客户端收到终止事件后立即重读 Thread 就能看见完整边界；若 I/O 屏障失败，Runtime 会记录警告并保留待重试项，但不会撤回已经送出的客户端事件。

这仍不是跨文件系统和数据库的分布式事务。Codex 保证的是单条 Thread 重放日志内部的顺序，以及“投影不能领先规范历史”的关系。突然断电、磁盘损坏或应用在屏障失败后被强制杀死，仍可能使最后一段工作不可恢复。

Resume 是重放，不是对象反序列化

进程重启后，旧的 Mutex、Channel、Task、网络流和子进程句柄都不存在。Resume 会用原 ThreadId 创建新的 Session，重新打开 Rollout 的追加写入器，并把已存记录交给 `reconstruct_history_from_rollout`。

重建器需要同时恢复几类结果：

```
Rollout replay
├── model-visible history
├── previous turn settings
├── reference TurnContext
├── WorldState baseline
└── context-window number and IDs
```

它不是简单过滤出所有 `ResponseItem`。重建过程从新到旧识别 Turn 边界、Rollback Marker 和最新可用的 Compaction Checkpoint，再按时间顺序重放仍然有效的后缀。`WorldState` 也有自己的规则：完整快照建立基线，后续 merge patch 才能在该基线上应用；遇到 Compaction 时旧基线失效，新的窗口要重新建立完整快照。

Paginated Rollout 可以避免每次 Resume 都读取整个大文件。LocalThreadStore 从日志尾部反向扫描，找到以下两项后便可以停止：

1. 带 `replacement_history` 和 `window_number` 的可用 `CompactedItem`；
2. 一个已完成真实用户 Turn 的兼容 `TurnContextItem`。

前者给出新的模型历史起点，后者给出恢复设置和参考基线。扫描结果再补上文件头部的规范 `SessionMeta`，按时间顺序交给同一个重建器。

有些形状不能安全截断。旧 Compaction 没有替代历史或窗口号、日志中出现 Rollback Marker，或者无法证明 TurnContext 属于真实用户 Turn 时，扫描必须一直回到开头。Legacy 历史和已压缩的 Rollout 目前也走完整读取路径。这里的优化原则很明确：只有能证明较老记录不会改变结果，才允许少读。

读取 JSONL 时，单行 JSON 解析失败会被计数并跳过，而不是让整个文件立即不可读。这个策略提高了剩余历史的可用性，但它不是无损修复：如果坏掉的恰好是 Tool Output、Compaction Checkpoint 或

Turn 边界，重建语义仍可能变化。调试时不能只看“Resume 成功”，还要检查 parse error 和重建后的历史形状。

未完成工具调用在送给模型前被修成合法对话

进程可能恰好死在 Tool Call 已写入、Tool Output 尚未写入的时刻。把这样的历史原样发给 Responses API，会破坏“每个 Call 都有对应 Output”的协议不变量。

Codex 没有伪造工具真的执行成功。`ContextManager::for_prompt` 在生成模型输入副本时执行规范化：

- Function Call、Custom Tool Call、Local Shell Call 或客户端 Tool Search 缺少结果时，在调用后插入稳定 ID 的合成 Output；
- Function 和自定义工具的合成文本是 `aborted`，Tool Search 得到空工具集合；
- 只有 Output、找不到对应 Call 的孤儿结果会从模型输入中移除；
- 不支持的图片或音频内容也在同一步被替换或剥离。

关键是“模型输入副本”。源 Call 带 Item ID 时，合成 Output 会由它派生稳定 UUID，重复 Resume 和 Retry 不会不断改变 Prompt Cache Key；旧历史中的 Call 若没有 Item ID，则沿用无 Output ID 的兼容行为。两种情况都不会反写到原始 Rollout，把一次恢复性假设冒充成真实工具执行记录。

中断路径还有更高层的边界：Runtime 会记录 `TurnAborted`，必要时插入 interrupted guidance。Fork 若截到一个仍在进行的 Turn，可以丢弃未完成后缀，或为明确的 Interrupted Snapshot 附加同类中断边界。Call/Output 配对解决协议合法性，TurnAborted 解决回合语义；二者不是同一层修复。

Compact 不删除旧历史，只改变重放起点

Context 太长时，Codex 会构造一个更短的 replacement history。Local Compact 通常保留必要的用户消息并加入 Summary；Remote Compact 可以直接返回压缩后的 Transcript。安装新历史时，Session 同时完成三件事：

1. 用 replacement history 替换当前 Context History；
2. 追加一个 `CompactedItem`，其中保存相同 replacement history 和新窗口身份；
3. 在需要时追加新的完整 WorldState 与 TurnContext 基线。

旧 ResponseItem 没有从 JSONL 删除。恢复器反向找到最新有效 Compaction 后，把它的 replacement history 当成完整历史基座，只重放更新的后缀。于是 Compact 同时做到两件事：模型不再承担全部旧 Token，审计日志仍保留压缩之前发生过的记录。

这个设计的成本是日志不会因为 Compact 自动变小；多次压缩还可能累积摘要误差。Compact 是上下文窗口管理，不是磁盘清理，也不是事实数据库的无损归档。

Rollback 回退的是会话历史，不是工作区

Legacy `thread/rollback(num_turns=N)` 的实现也是追加式的。它先确认当前没有活动 Turn，刷新 Rollout，读取规范历史，然后在内存中把 `ThreadRolledBack { num_turns: N }` 临时接到末尾进行完整重建。重建成功后，才把同一个 Marker 追加到 Rollout 并刷新。

重放器从后向前跳过最近 N 个真实用户 Turn。没有 User Message 的独立维护任务不消耗这个计数；不完整但已经出现用户边界的 Turn 仍可以被排除。之前的原始记录保留在日志中，后续 Resume 看到 Marker 后得到相同的逻辑历史。

当前 App Server 明确拒绝对 Paginated Thread 调用 `thread/rollback`，并且该 RPC 已标记为即将移除的旧接口。因此不能把 Legacy Rollback 描述成所有 History Mode 都具备的通用能力。

更重要的是，Rollback 不执行以下操作：

- 不把被覆盖的源文件恢复到旧字节；
- 不撤销已经提交的 Git Commit；
- 不杀掉历史 Turn 启动而仍存活的外部服务；
- 不追回已经发送的网络请求或远程副作用。

它回退的是“下一次模型还应看见哪些会话 Turn”。真正要撤销代码，应使用 Git、反向 Patch、备份或目标系统自己的补偿操作。把会话 Rollback 当成工作区 Undo，可能得到一个忘记自己改过文件、但文件仍然已经改变的 Agent。

Fork 冻结一个前缀，然后产生新身份

Fork 与 Compact、Rollback 的共同点，是都从已有 Rollout 计算新的有效历史；不同点是 Fork 会创建一个新的 ThreadId。

Legacy 或复制式 Fork 会把处理后的历史写进子 Rollout。Paginated 的 Reference-backed Fork 可以只保存一个 `history_base`，其中的 `HistoryPosition` 指向某条物理 Rollout 的稳定前缀：

```
HistoryPosition
├── thread_id
├── end_ordinal_exclusive
└── end_byte_offset
```

`prepare_fork` 先持久化源 Thread，把所需段投影到 SQLite，解析 Latest、ThroughTurn 或 BeforeTurn 边界，再加载该边界对应的模型上下文。返回的 `PreparedFork` 还持有 Store 管理的源预留，在子 Thread 的引用变得耐久之前阻止源历史被删除。

子 Rollout 只写自己的本地增量，读取时 ThreadStore 沿 lineage 拼接祖先前缀和子增量。这样不需要为每个分支复制一份大历史，也意味着删除父 Thread 不再只是删除一个独立文件：Store 必须先检查仍在引用它的后代。

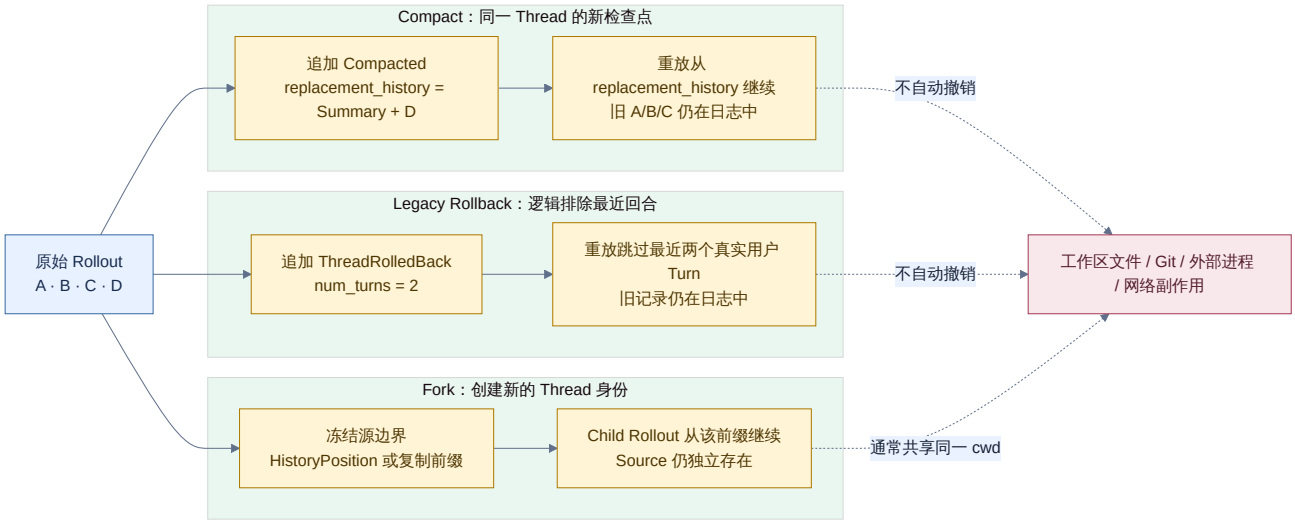

图 8-3：Compact 与 Rollback 都保留旧 Rollout 记录，只改变重放结果；Fork 则以冻结前缀创建新 Thread。三者都不提供文件系统或网络副作用的自动回滚。

多 Agent Fork 通常还会过滤 Reasoning、Tool Call 和旧通信，并替换子角色指令。Reference-backed Fork 解决“历史存在哪里”，Context Filter 解决“哪些内容适合交给子 Agent”，两者不能互相代替。

一条可操作的故障排查顺序

持久化问题最好按权威层级排查，而不是先删数据库：

1. 确认 ThreadId、History Mode，以及当前是否仍有 Live Writer；
2. 找到对应 `.jsonl` 或 `.jsonl.zst`，检查第一条 `SessionMeta` 是否属于该 Thread；
3. 检查尾部是否有完整 `TurnStarted → TurnComplete/TurnAborted` 边界，以及 Rollback、Compacted、TurnContext、WorldState 的相对顺序；
4. 查看是否存在 JSON parse error、ordinal 断裂或压缩/物化切换问题；
5. 再比较 SQLite metadata 和 history projection 是否只是落后；
6. 对恢复后的模型请求，检查 Prompt Normalize 是否插入了 `aborted` Output 或删除了孤儿 Output；
7. 最后单独核对工作区、Git、后台进程和远程服务，因为它们不在 Rollout 回退边界内。

也可以用几条不变量快速判断实现是否偏离：

```
JSONL canonical position >= SQLite projected position
terminal event path => append terminal marker and attempt an explicit flush
every model-visible tool call => a compatible output
every model-visible output => a compatible call
rollback changes effective conversation history, not external state
referenced child durable => its ancestor prefix cannot be deleted underneath it
```

这些不变量比“保存成功”更具体。它们分别约束写入顺序、终止可见性、Prompt 协议、回滚范围和 Fork 生命周期。

持久化边界闭合运行链

Codex 的主运行链由持久化边界闭合：Thread 接收输入，Turn 驱动模型和工具，事件把过程送给客户端，Rollout 把足以重建的语义留到进程之外。Session 可以消失，因为身份和规范历史没有与它绑定在同一段内存里。

源码导航

- ThreadStore 持久化接口
- LiveThread 与元数据同步
- LocalThreadStore 的规范写入与 SQLite 投影顺序
- RolloutRecorder 的队列、Flush 与失败重试
- Rollout Persistence Policy
- Session Rollout 重建
- Prompt 历史的 Call/Output 修复
- Paginated Model Context 反向扫描
- Reference-backed Paginated Fork

第九章　用 Python 搭建 Mini Codex

Mini Codex 把 Thread、Turn、模型流、工具路由、安全边界和 Rollout 重新组合成一个可运行的教学 实现，使各层之间的契约可以通过代码和测试直接验证。

目标不是写一个“模型返回 function call 就执行函数”的演示脚本，而是回答四个运行时问题：工具 执行后，下一次采样能否看到新文件；工具目录在采样期间变化时，规格和 Handler 是否仍然一致； 进程死在 Call 与 Result 之间时，恢复后怎样生成合法 Prompt；Interrupt 到达时，怎样保证不会同时 收到 Completed 和 Aborted。

配套项目位于 `examples/mini-codex` 。它使用 Python 3.12、`asyncio` 、`dataclass` 、`Protocol` 、 httpx、JSONL、pytest 和 mypy strict；不依赖模型 SDK。默认演示和 27 项测试完全离线，只有同时 设置 `OPENAI_API_KEY` 与 `MINI_CODEX_LIVE_MODEL` 时，额外的真实 Responses smoke test 才会运行。 目标不是复刻 Codex 的产品规模，而是让关键架构约束变成可执行断言。

先划定实现和安全边界

当前教学版只支持单进程、单事件循环和一个活动主 Thread。一个 Thread 中每次只能运行一个 Turn，但一个 Turn 可以包含多次模型采样和多次工具执行。它保留以下能力：

- `Op → Session → Event` 双向协议；
- 生命周期不同的 `TurnContext` 与 `StepContext` ；
- 流式 `ModelEvent` 和完整 `ResponseItem` ；
- 分层 Prompt、History 与动态 World State；
- 同一个快照产生的 ToolSpec 和 ToolRouter；
- Shell、持续进程 Unified Exec 和 Add/Update/Delete Patch Grammar；
- Exec Policy、Approval、可替换执行环境和 Tool Hook；
- 专用 JSONL Writer Task、Interrupt、Resume、Compact、Rollback 和复制式 Fork；
- 真实 Responses/SSE Adapter、录制回放模型、最小 MCP 目录与 Agent Mailbox 控制面。

仍未实现 TUI、App Server、Provider 能力协商、WebSocket、完整 MCP 连接/认证/资源协议、真正的子 Agent Session 循环、远程执行环境、SQLite 投影、分页 Lineage 或跨平台系统沙箱。这里的 MCP 只覆盖动态 Tool Catalog/Call 适配，AgentPool 只覆盖身份、容量和 Mailbox；不能把它们外推为完整 MCP Client 或多 Agent Runtime。

尤其要明确威胁模型。`WorkspacePolicy` 能阻止 Apply Patch 使用 `../` 逃出工作区，也会限制 Shell 的 `cwd` ，但它无法阻止 `python -c` 、编译器或其他命令自行访问工作区外的文件和网络。 应用层路径检查只是误操作防线，不是针对恶意代码的 OS 级沙箱。

包结构先表达依赖方向

项目没有把所有逻辑塞进一个 Agent 类，而是按协议、运行时、上下文、模型、工具、策略和持久化拆分：

```
src/mini_codex/
├── protocol.py
├── runtime/        # Session、Thread、Turn/Step、取消、Agent Mailbox
├── context/        # Prompt、History、World State
├── models/         # Protocol、Responses/SSE、Accumulator、录制回放
├── tools/          # Plan/Router、Shell、Unified Exec、Patch、Hook、MCP
├── execution/      # 本地进程、持续进程与 Sandbox Adapter
├── policy/         # Workspace、Exec Policy 与 Approval
├── persistence/    # Writer Task、JSONL、Resume/Fork
└── cli.py
```

Runtime 不直接导入 OpenAI、Anthropic 或其他 SDK。它只依赖一个 `ModelClient` Protocol。工具也不直接从全局变量取得当前工作区和审批设置，而是接收一次调用所需的 `ToolInvocationContext`。这样测试可以替换边界适配器，同时保留生产代码经过的完整主链。

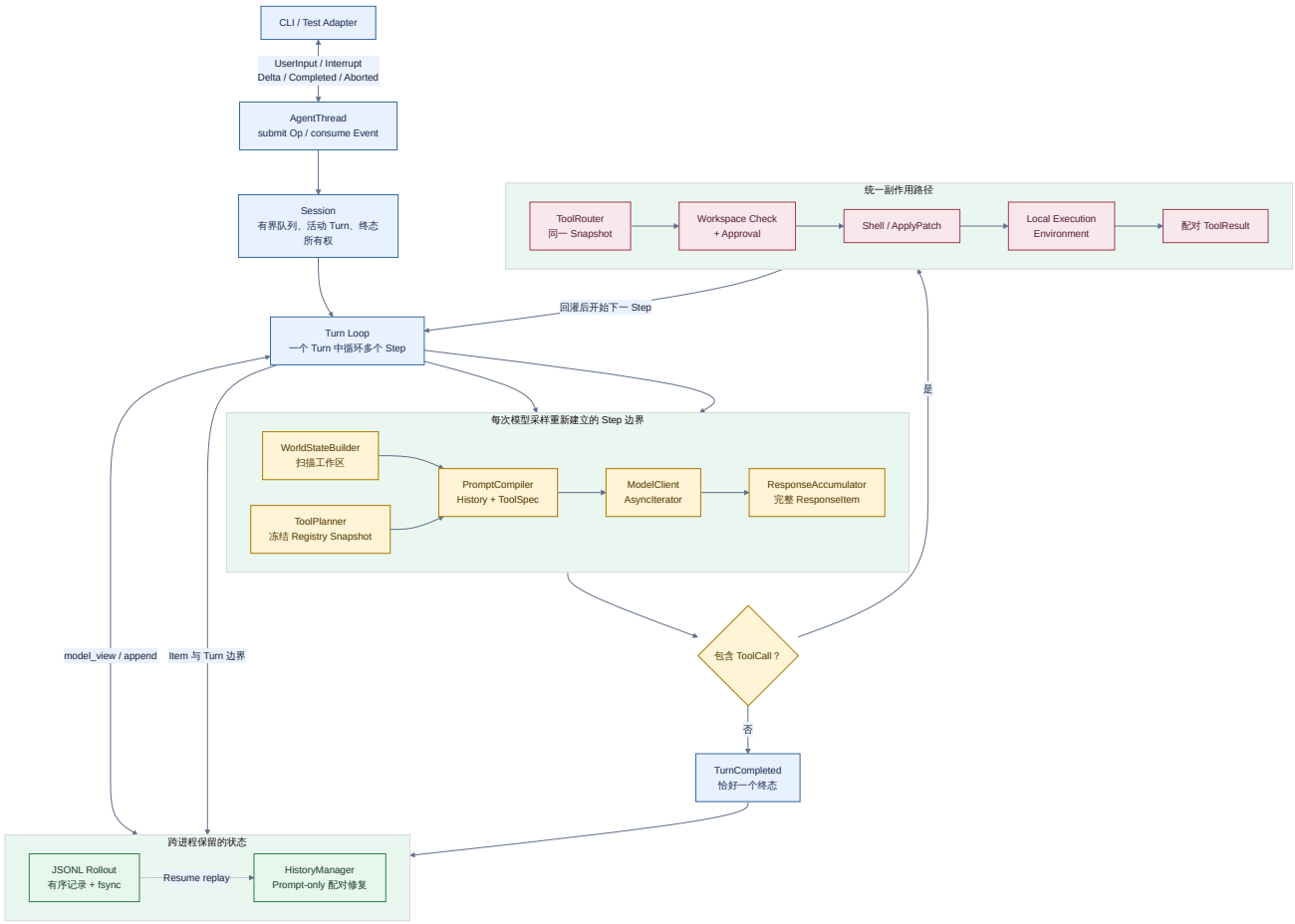

图 9-1：Turn Loop 是控制中心；黄色区域在每次模型采样前重建，红色区域统一处理副作用，绿色区域负责跨进程重放。ToolPlan 的规格与 Router 来自同一快照。

组件之间存在三种时间尺度：Session 跨多个 Turn 存活，Step 快照只服务一次模型采样，JSONL 则在进程退出后继续存在。混淆这三个时间尺度，环境刷新、动态工具和恢复都会产生隐蔽错误。

Op 和 Event 是 Runtime 的窄腰

输入协议只定义三个操作：

```
@dataclass(frozen=True, slots=True)
class UserInputOp:
    text: str

@dataclass(frozen=True, slots=True)
class InterruptOp:
    pass

@dataclass(frozen=True, slots=True)
class ShutdownOp:
    pass
```

输出协议分为瞬时增量和完整边界：`TurnStarted`、`TextDelta`、`ItemCompleted`、`TurnCompleted`、`TurnAborted` 与 `RuntimeErrorEvent`。每个 `AgentEvent` 都带 submission ID，调用方可以把事件归回某次提交。

Session 内部使用两个容量默认为 32 的 `asyncio.Queue`。输入队列只有 Session 主循环消费，输出队列由 CLI 或测试适配器消费。有界队列不是性能装饰：如果客户端处理事件太慢，生产者会在 `put()` 处等待，内存不会无限增长。代价也很直接——单个消费者必须持续排空事件，否则 Runtime 的推进和关闭都会受到反压。

主循环没有直接 `await` 完整 Turn，而是把 Turn 创建为活动 Task。这样它仍能继续从输入队列取出 `InterruptOp` 并设置取消信号。第二个用户输入在已有 Turn 运行时会得到错误事件，第一版不暗中排队多个 Turn，也不把它解释成 steer。

这里还有一个很小但真实的竞态：调用方可能在 `submit(UserInputOp)` 后立刻等待 idle。如果只由主循环在稍后清除 idle，等待者可能读到上一个 Turn 留下的“空闲”。实现因此在用户输入入队前就 清除 idle，只有终态发出后才重新设置。

一个 Turn 为什么仍然需要多个 Step

Turn 是对一次用户请求负责的完整工作单元；Step 是其中一次模型采样及其随后触发的工具处理。 两者的数据快照故意不同：

```
@dataclass(frozen=True, slots=True)
class TurnContext:
    id: str
    base_instructions: str
```

```
    cancellation: CancellationToken

@dataclass(frozen=True, slots=True)
class StepContext:
    id: str
    turn_id: str
    sequence: int
    world_state: WorldState
```

顶层指令和取消 Token 在当前 Turn 内保持一致。World State 则在每次采样前重新捕获。第一版只扫描工作区的文件名；如果文件集合发生变化，就追加一条 Developer Message，并在构造模型请求前刷新 Rollout。

这已经足以验证关键行为。假设 Step 1 的 Apply Patch 创建 `notes.txt`：Step 2 不能继续使用 Turn 开始时的旧文件清单，而必须重新扫描并把 `notes.txt` 放进新 Prompt。官方 Codex 的 World State 还包含权限、协作模式、项目指令等分区，并维护完整快照和增量 Patch；Mini Codex 只保留刷新时机，不冒充完整实现。

Turn Loop 的核心可以缩成以下结构：

```
for sequence in range(1, max_steps_per_turn + 1):
    cancellation.raise_if_cancelled()
    step = await capture_step(turn, sequence)
    tool_plan = tool_planner.plan()
    request = compile_prompt(turn, step, history.model_view(), tool_plan.specs)
    response = await collect_model_stream(request)
    await record_and_flush(response.items)

    if not response.tool_calls:
        await complete_turn()
        return

    results = await tool_plan.router.dispatch(response.tool_calls, invocation_context)
    await record_and_flush(results)
```

`max_steps_per_turn` 默认是 12。它防止一个始终要求调用工具的异常模型让 Turn 永远循环。超过限制 不是“正常完成”，而是进入失败路径并发出 `TurnAborted`。

模型流不能直接当历史保存

`ModelClient` 只承诺返回一个异步事件迭代器：

```
class ModelClient(Protocol):
    def stream(
        self,
        request: ModelRequest,
        cancellation: CancellationToken,
    ) -> AsyncIterator[ModelEvent]: ...
```

`ModelEvent` 包括文本 Delta、完整 Assistant Message、完整 Tool Call 和 `ModelCompleted`。TextDelta 一到达 Session 就可以转发给客户端，但它不会逐块写进 History。`ResponseAccumulator` 按 item 首次出现的顺序组装完整 `ResponseItem`，确认收到 `ModelCompleted` 后才允许 `finish()`。如果传输在完成标记前断开，半截响应不会被伪装成一个成功模型结果。

这层分离同时服务三个消费者：UI 要低延迟 Delta，Tool Router 要结构完整的 Call，下一次 Prompt 和 Resume 要稳定的完整 Item。官方 Codex 并没有一个统一同名的 `ResponseAccumulator` 类型；相关职责分布在流式解析和 active item 等状态中。Python 版本把它们收拢，是为了让协议边界更容易测试，不是声称官方 Rust 也使用同一对象。

DemoModelClient 仍是默认离线适配器。第二阶段已经增加 `ResponsesModelClient`：它把 Message、Function Call/Output 与 ToolSpec 编译成真实 `POST /v1/responses` payload；`HttpxResponsesTransport` 负责认证、HTTP status、typed SSE 和取消。Runtime 没有因此导入 httpx 类型。另有 Recording 与 Replay Adapter 用请求指纹固定离线事件序列。

OpenAI 官方文档要求 Responses 流消费者按 typed SSE event 的 `type` 分支，文本流至少处理 `response.output_text.delta`、`response.completed` 与 `error`，函数流还会出现 arguments delta/done；Mini 的完整映射和 UTF-8/SSE chunk 测试在 9.6、9.7 展开。

ToolPlan 必须同时冻结“告诉模型什么”和“真正执行什么”

工具 Registry 允许 Handler 更新，并给每一版分配 generation。每次 Step 开始时，Planner 只取一次快照：

```
def plan(self) -> ToolPlan:
    snapshot = self._registry.snapshot()
    return ToolPlan(
        generation=snapshot.generation,
        specs=tuple(handler.spec for handler in snapshot.handlers.values()),
        router=ToolRouter(snapshot.handlers),
    )
```

如果先生成 ToolSpec，等模型返回后再去全局 Registry 查 Handler，就会出现时间穿越：模型按旧参数定义生成 Call，Runtime 却用新 Handler 执行。Mini Codex 的测试会故意在模型采样期间替换同名 Handler，断言当前 Step 仍由旧 Router 处理，下一 Step 才看到新 generation。

Router 对每个已接收 Call 都产生同 `call_id` 的 Result。未知工具、审批拒绝、参数错误、执行异常和取消都作为 `is_error=True` 的 ToolResult 反馈，而不是让 Call 从历史中悬空。工具失败不等于 Runtime 失败：模型看见失败结果后，仍可能修改参数或选择另一条路径。

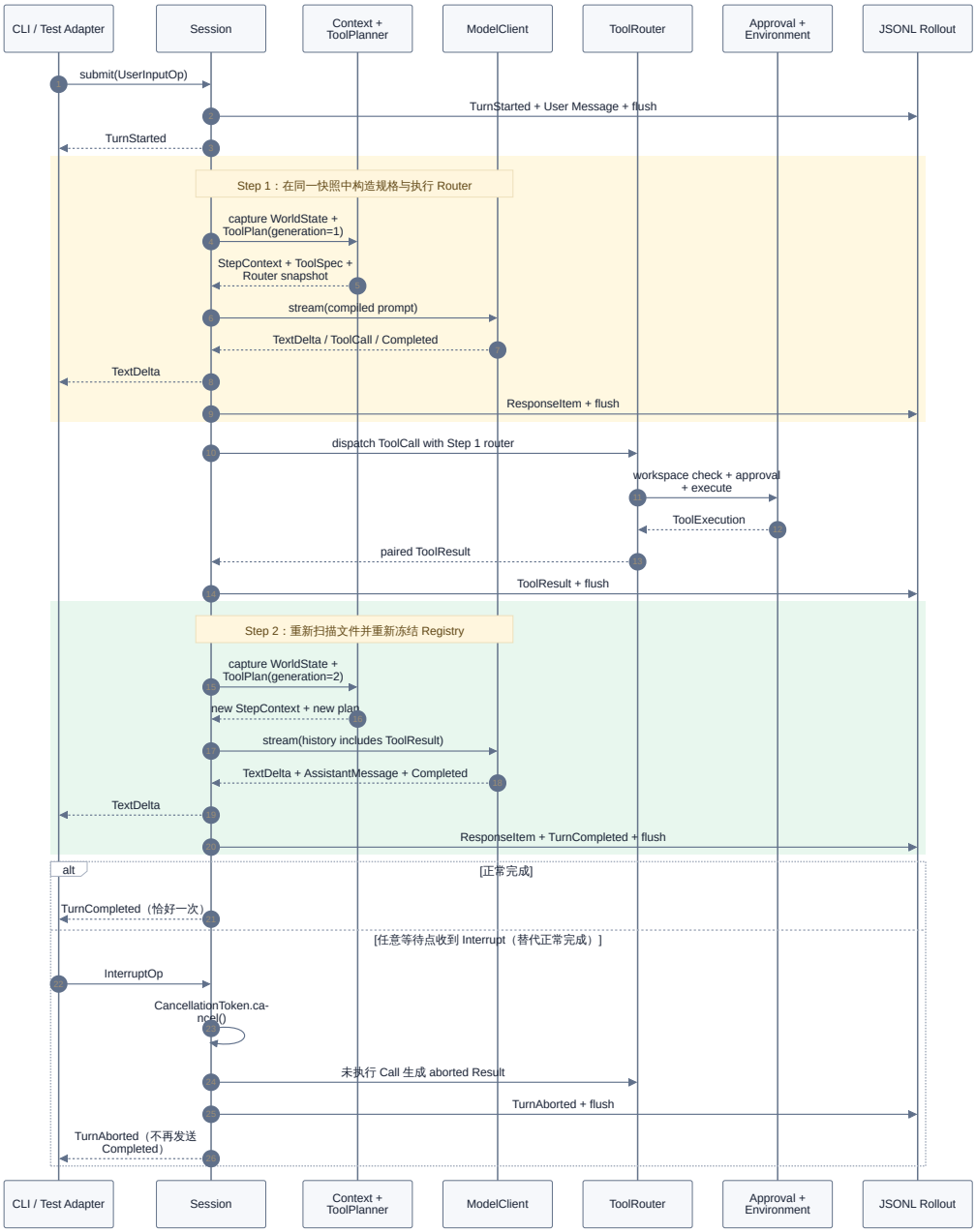

图 9-2：Tool Call 结束第一轮模型采样，配对 Result 落盘后才开始第二轮；文本增量即时发给客户端。正常完成和中断终态互斥。

这条顺序使第二次模型请求必然包含 Tool Result，也使进程崩溃后的日志能区分“模型尚未请求工具” 和“已经请求，但结果还没写入”。如果为了减少 I/O 把 Call、Result 和 TurnCompleted 全部拖到最后 一次写入，Resume 就无法准确判断故障落点。

Shell、Patch、审批与执行环境

Shell 接受 argv 数组，而不是让 Runtime 先拼成一条 Shell 字符串。它先验证参数和 cwd，再由 Exec Policy 产生 allow/prompt/forbid；只有 prompt 才请求 Approval，最后交给可替换 ExecutionEnvironment。执行环境负责超时、协作式取消、kill/reap 和 stdout/stderr 截断。

Apply Patch 现在解析 `*** Begin Patch` / `*** End Patch`，支持多文件 Add、Update hunk 和 Delete。 Parser 先生成 Operation，Planner 再解析全部 workspace 路径、存在性与唯一 context；所有操作规划 成功并经 Approval 后，才 stage 临时文件和 commit。Mini 仍不支持官方 Grammar 的 Move、End of File、宽松 shell 截取和 streaming preview；多个 `os.replace` 也不是跨文件事务。

Unified Exec 把长进程注册到 ProcessManager。`exec_command` 首次 yield 返回 `session_id`，后续 `write_stdin` 可写输入或轮询增量输出；正常退出、timeout 和 Interrupt 都必须移除 Registry 并清理 reader/watcher Task。9.10 单独给出状态机和 stdin 测试。

审批是 Protocol，不是写死在工具中的 `input()`。CLI 使用 `InteractiveApproval`，测试可以注入 `AlwaysApprove` 或 `AlwaysDeny`。审批拒绝会形成 ToolResult，例如：

```
shell command rejected by approval policy
```

模型因此知道动作没有发生。若只把拒绝显示给 UI 而不回灌模型，下一次采样可能误以为命令已经 执行。相反，批准也不代表安全：它是用户决策记录，真正的文件系统、网络和进程隔离仍需要执行器或操作系统提供。

Interrupt 必须收束到一个终态

取消 Token 内部使用 `asyncio.Event`。模型适配器、执行环境和 Tool Router 在自己的等待点检查它； Session 收到 Interrupt 后只修改这个共享取消状态，不从外部直接拼接一个“已中断”回答。

终态由 `_run_owned_turn` 统一拥有：正常执行完成时发送一个 `TurnCompleted`；捕获 `TurnCancelled` 时先追加并刷新 `turn_aborted`，再发送一个 `TurnAborted`；其他异常发送错误事件 后也以 Aborted 收束。finally 只处理没有任何路径拿到终态所有权的防御情况。

Tool Router 还有一条局部不变量：同一批 Call 中，取消发生前已执行的 Call 保留真实 Result，尚未 执行的 Call 生成 `aborted` Result。随后取消继续向上抛出，Turn 进入 Aborted。这样模型可见历史的 Call/Result 结构仍合法，客户端也不会同时收到 Completed。

取消只能保证 Runtime 知道“不再等待”。若子进程、网络请求或外部系统已经产生副作用，`TurnAborted` 不会自动撤销它们。需要可回滚效果时，工具本身必须提供事务或补偿操作。

JSONL 保存语义，不保存 Python 对象

Rollout 记录 session meta、完整 Item、Turn Started、Completed、Aborted、Compacted 和 RolledBack Checkpoint。它不保存 Queue、Task、文件句柄或模型连接。`flush()` 把 immutable batch 交给专用 Writer Task；File Sink 逐行 flush/fsync 并返回 confirmed count。若第 k 行失败，`PrefixWriteError` 只确认已写前缀，pending 保留未确认后缀供下一次重试。

一段最小日志形状如下：

```
{"type":"session_meta","version":1,"thread_id":"thread_demo"}
{"type":"turn_started","turn_id":"turn_...","submission_id":"submission_..."}
{"type":"item","item":
{"id":"item_user","type":"message","role":"user","content":"...","turn_id":"turn_..."}}
{"type":"item","item":
{"id":"item_call","type":"tool_call","call_id":"call_...","name":"shell","arguments":
{},"turn_id":"turn_..."}}
{"type":"item","item":
{"id":"item_result","type":"tool_result","call_id":"call_...","output":"...","is_error":
false,"turn_id":"turn_..."}}
{"type":"turn_completed","turn_id":"turn_...","final_message":"..."}
```

Resume 顺序读取这些记录，遇到 Compacted 或 RolledBack 时用 replacement history 替换此前历史；某个 TurnStarted 到文件结尾仍没有匹配终态，就返回 `incomplete_turn_id`。单行坏 JSON 会被计数并 跳过，这提高剩余日志的可用性，但不能保证语义无损。

如果崩溃发生在 Tool Call 写入后、Result 写入前，`HistoryManager.model_view()` 会在模型输入副本 中插入稳定 ID 的 `aborted` Result；孤儿 Result 则从该副本移除。原始 Rollout 不被改写，恢复性假设不会伪装成真实工具执行。稳定 ID 由源 Item ID 通过 UUIDv5 派生，多次 Resume 不会每次改变 Prompt 形状。

Compact 接受一段人工摘要和需要保留的最近用户 Turn；Rollback 按 User Message 边界删除最近 N 个 Turn；两者都把 replacement history 写入 checkpoint。Fork 把当前有效历史复制到一个新 Thread 的 JSONL。这三个实现足以验证重放起点和新身份， 但没有官方实现中的自动摘要、Window/WorldState 基线、Ordinal、压缩日志、SQLite 投影或 Reference-backed Lineage。

即使调用了 `fsync()`，当前实现也不是完整故障恢复系统。它会在追加前为无换行坏尾补 delimiter，但没有目录 fsync、batch checksum、跨进程 writer lock，更不存在 JSONL 与其他存储之间的事务。 这里验证的是 confirmed prefix、重试与重放算法，不宣称任意断电场景绝对无损。

用测试证明架构约束

项目现有 27 项默认离线测试和 1 项 opt-in 真实模型测试。它们不是按函数做机械覆盖，而是针对会跨模块失效的不变量：

测试场景	需要同时成立的约束
提交后立即等待 idle	等待的是新 Turn 的终态，不会误读上一次留下的 idle 状态
Function Tool 线格式	`type/name/description/strict/parameters` 与 Responses 结构一致
Function Tool 内部元数据	`output_schema` 不会误发进普通工具数组，Deferred 标记按需发送
单次采样	Delta、完整 Item、恰好一个 Completed 的事件顺序
工具后续采样	配对 Result 在第二次请求前进入 History
Registry 热更新	当前 Step 的 Spec 与 Router 同源，下一 Step 才更新
Patch 创建文件	第二次采样重新捕获 World State

测试场景	需要同时成立的约束
审批拒绝	不执行副作用，仍生成模型可见失败 Result
路径穿越	`../` 解析后不能逃出工作区
不完整 Turn Resume	报告故障边界，只在 Prompt 副本补稳定 aborted Result
Compact 与 Fork	checkpoint 重放及新 Thread 使用有效历史
Interrupt	取消传播并且只有一个 Aborted 终态
SSE chunk 与 typed event	UTF-8/frame 边界、错误事件和 completed gate
Unified Exec	首次 yield、stdin 续写、timeout kill/reap 与 Registry 清理
Patch 全计划	后置非法操作不会让前置 Add 提前落盘
Exec Policy 与 Hook	longest prefix、参数改写、生命周期与错误回灌
Writer fault injection	部分确认后只重试未确认后缀
Rollback	replacement checkpoint 重放得到正确前缀
MCP Catalog	一个 server 的目录按 generation 整体替换
Agent Mailbox	capacity 拒绝、final 通知和 bounded backpressure
Model record/replay	只有 request fingerprint 一致才回放

在 `examples/mini-codex` 中运行：

```
uv sync --extra dev
uv run pytest -q
uv run mypy src
uv run python benchmarks/runtime_baseline.py
```

当前结果是 27 passed、1 skipped；mypy strict 对 `src/mini_codex` 下 46 个 Python 文件无报错。20 次离线单 Turn 本机样本中位数约 14.46 ms、P95 约 19.06 ms；这只是 2026-08-03 当前机器的观测值，不是跨机器 SLA。 还可以启动离线交互：

```
uv run mini-codex \
  --workspace ./demo-workspace \
  --rollout ./demo-rollout.jsonl
```

普通文本由离线模型回答；`shell python -c "print('hello')"` 会走 Shell Tool；下面的命令会通过 Patch Grammar 创建文件：

```
patch *** Begin Patch\n*** Add File: a.txt\n+hello\n*** End Patch
```

CLI 提交普通输入后会立即继续接收命令，因此活动 Turn 中仍可输入 `/interrupt`；`/wait` 用来等待 Runtime 回到空闲。Shell 和 Patch 默认逐次询问批准，`/compact 摘要`、`/rollback 1` 与 `/quit` 分别测试压缩、回退和关闭边界。

Mini Codex 保留的架构契约

Mini Codex 的代码量远小于官方 Codex，但没有把 Agent 简化成一个无限 `while True`。实现保留的不是 Rust 类型名，而是几条可以迁移到其他 Coding Agent 的契约：

1. 客户端通过 Op/Event 与长期 Runtime 交互，不直接操纵模型循环；
2. Turn 内稳定配置和 Step 级动态环境必须使用不同快照；
3. 流式展示事件、完整模型项和耐久记录不能混为一种数据；
4. 工具描述和执行绑定必须来自同一个 Step 快照；
5. 拒绝、失败和取消也要维持 Call/Result 配对；
6. 恢复依赖可重放的语义记录，不依赖反序列化旧 Session；
7. 路径策略、人工审批和 OS 隔离是三层不同的安全机制。

同时，这个实现也暴露了成本：有界队列需要明确消费者，显式 flush 增加 I/O，动态 Step 快照增加扫描和编译工作，取消必须贯穿每个等待边界，恢复规则又会把协议不变量带进持久化层。它们不是为了 让架构图更漂亮，而是为失败、并发和时间变化付出的复杂度。

Mini Codex 也提供了一把判断架构必要性的尺子：一个机制若删掉后，某项测试不变量立刻失效，它就不是单纯的代码组织偏好；若删除后行为契约不变，它更可能只是产品规模、兼容性或特定执行环境带来的实现选择。

章内索引

- 9.17 MCP Adapter 与动态工具目录
- 9.18 子 Agent、Mailbox 和容量控制
- 9.19 端到端真实模型测试与录制回放测试
- 9.20 故障矩阵、类型检查、并发测试和性能基线

源码与实现导航

- 官方 Op/Event 协议
- 官方 CodexThread 双向接口
- 官方 run_turn 主循环
- 官方 TurnContext
- 官方 StepContext
- 官方 ToolRouter
- 官方 Prompt History 规范化
- 官方 Rollout Recorder
- 配套 Mini Codex README
- Mini Codex Session
- Mini Codex 测试

第十章　Agent Runtime 的可迁移设计

Codex 生产实现与 Python Mini Codex 共同揭示了两类复杂度：一类来自任何工具型 Agent 都会遇到的并发、故障和时间一致性；另一类由 Codex 的产品规模、兼容性和扩展生态触发。

设计另一个 Runtime 时，需要判断哪些合同必须从第一版建立，哪些能力应等到需求出现后再引入，以及每一层复杂度怎样用失败残留和可执行测试证明。

源码说明当前实现怎样工作，测试约束具体行为，Mini Codex 验证这些合同能够跨语言表达。三类证据都不能替代对目标系统需求的判断。

Runtime 合同与产品层

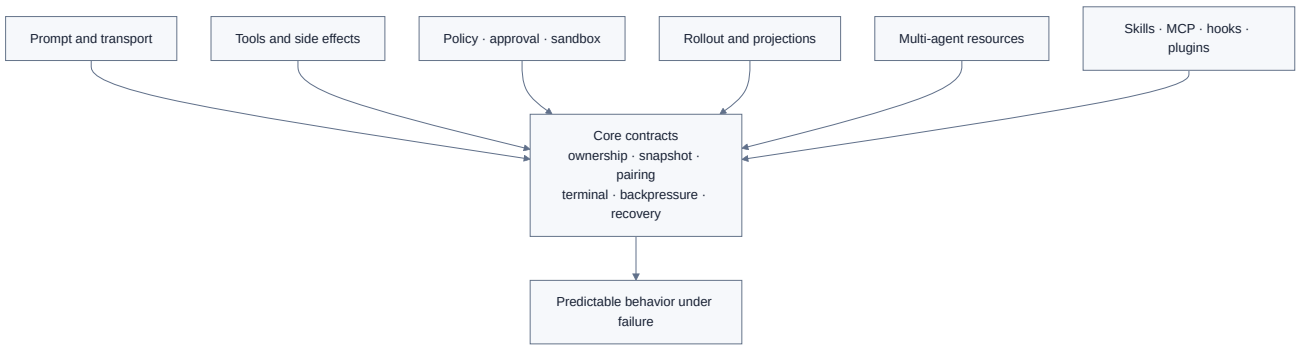

图 10-1：中心是故障下仍需成立的 Runtime 合同；外圈能力只有在动态工具、不可信执行、跨进程恢复或多 Agent 等需求出现时才加入。

中心的合同数量并不多：输入和事件有明确类型；状态有唯一所有者；一次请求使用一致快照；模型可见 ToolSpec 与执行 Handler 同源；ToolCall 与 ToolResult 按 call_id 配对；副作用前完成校验和授权；Queue、进程和 Agent 都有上限与终态；持久恢复从规范记录开始，而不是从 UI 日志猜测。

这些合同在只有一个模型和两个工具时也有意义，因为它们处理的不是产品规模，而是时间：调用发生前 知道什么，等待期间谁还能修改状态，失败后已经留下什么，下一次采样又应看见什么。

外圈的产品层则由明确需求触发。没有第二个客户端，不需要 App Server 版本投影；没有动态第三方工具， 不需要 MCP 目录刷新；不执行不可信进程，未必需要复制三平台沙箱矩阵；没有分页和搜索，不应先引入 SQLite 投影；没有可恢复子 Agent，不需要 Residency 和 AgentGraph。

事件驱动不是风格选择

一个同步函数可以完成“发送 Prompt，得到文本”。它无法在模型仍流式输出时接收 Interrupt，也无法同时等待审批回答和后台进程，更无法让慢客户端对事件生产者施加反压。

Codex 的 `Op → Session → EventMsg` 结构把控制输入和观察输出拆开。Session 是活动 Turn 的所有者， Turn 又负责让 Completed、Aborted 和 Failed 收敛为唯一终态。Delta 可以低延迟送到界面，但只

有完整 ResponseItem 才进入模型历史和 Rollout。

值得迁移的不是 `EventMsg` 的全部 variant，而是以下条件：

```
活动任务期间仍需接收控制输入
→ 输入分派与任务执行必须解耦
→ 增量观察与规范历史必须分层
→ 取消在状态所有者处收敛
→ 每个 Turn 只产生一个终态
```

若目标程序完全同步、没有流式输出、审批和后台资源，直接返回值比事件总线更清楚。一旦上述任一条件出现，继续把所有事情塞进一个 `run()` 通常只是把状态机藏在异常和布尔变量中。

快照解决的是时间一致性

TurnContext 与 StepContext 不是面向对象层级，而是两种稳定期。TurnContext 固定本次用户请求的模型、 cwd、权限基线和取消范围；StepContext 在每次模型采样前重新捕获工具计划、MCP binding 和 World State。

工具修改文件后，下一 Step 应看见新 World State。MCP 目录刷新后，下一 Step 可以暴露新工具。但一个 已经由旧 Schema 产生的 ToolCall 必须继续交给旧 Step 的 Router；否则热更新会改变过去请求的含义。

因此快照原则可以写成一句更精确的话：旧请求保持自洽，新请求看见新世界。它也解释了为什么 stream retry 不应该偷偷重建工具面，为什么 Handler 不应从全局变量读取 cwd 和权限。

Prompt 是编译产物，不是字符串

模型请求同时包含不同稳定期的内容：基础指令通常稳定，AGENTS 指导受目录作用域约束，History 按 Thread 增长，World State 按 Step 变化，ToolSpec 又必须与当前 Router 同快照。Compact 还会用一个 replacement checkpoint 替换旧前缀。

一个可靠的 PromptCompiler 先生成确定的语义快照，再让 HTTP/SSE 或 WebSocket Transport 选择完整或 增量编码。`previous_response`、prompt cache key 和稳定序列化都属于传输与性能层，不能决定模型应看 什么。权限、工具或历史真的变化时，允许 cache miss；错误复用旧语义比缓存失效严重得多。

这条设计即使没有 Provider Prompt Cache 也有收益：测试快照更稳定，录制回放可以校验 request fingerprint，故障诊断能比较具体输入层，而不是面对每次顺序都不同的大字符串。

ToolSpec 和 Handler 是一份计划的两面

模型看见 JSON Schema，Runtime 执行 Handler。只按名称在全局 Registry 中查找是不够的，因为工具可能来自 Feature、MCP、Plugin、Extension 或远程环境，而且目录能在运行时刷新。

Codex 的 ToolPlan 同时构造 `model_visible_specs` 和 Registry，并把 Router 放进 StepContext。Hosted 工具可能只有模型面，Deferred 工具可能暂不显示；真正的不变量不是两个集合相等，而是每个需要本地 执行的模型调用都能在同一 generation 中找到唯一兼容实现。

ToolCall 被接受后，生命周期继续经过参数校验、Pre Hook、Policy、Approval、ExecutionEnvironment、输出截断、Post Hook 和 ToolResult 提交。所有能阻止副作用的步骤都必须位于执行前。Post Hook 可以 记录或阻断后续流程，却不能回滚已经启动的进程、写入的文件或远端服务动作。

这里最容易被忽略的是 Result 写入失败：外部副作用可能已经发生，而历史还没有同 call_id Result。恢复动作应先补足协议事实，不能因为“没有成功记录结果”就盲目重跑工具。

安全不是一张 allow/deny 表

应用 Policy 判断动作风险，Approval 记录用户是否授权，OS Sandbox/Network Proxy 强制真实资源边界。 三者不能互换：Policy 不限制 syscall；用户批准不自动缩小文件系统权限；Sandbox 不理解删除操作是否 符合用户意图。

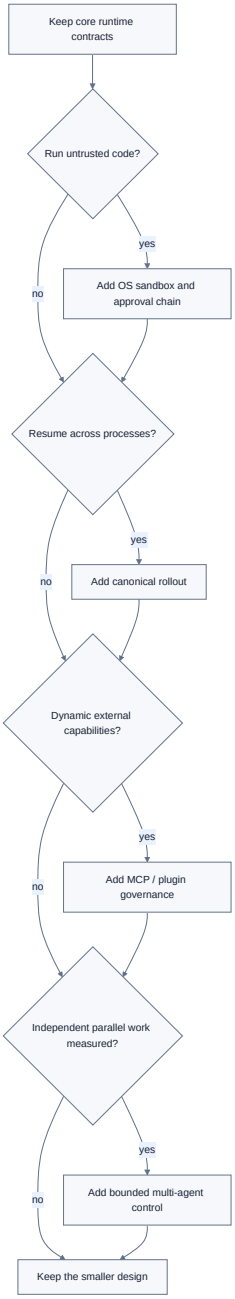

图 10-2：新增层必须由风险、持久化、生态、协作或客户端需求触发；每层都要说明所有者、失败矩阵和退化路径。

最小安全链应先把模型参数解析成具体 argv、cwd、路径和网络需求，再用 Permission Profile 缩小请求， 由 Policy 决定是否可执行/可询问，Approval 只授权这个具体范围，最后由执行器物化为平台强制约束。

Workspace 路径检查可以阻止 Patch 使用 `../`，却阻止不了 `python -c` 自行打开工作区外文件。Mini Codex 因此把默认 Adapter 明确命名为无 OS isolation 的 Local Adapter。说清没有什么，比把应用检查 叫作沙箱更重要。

多份持久化状态为什么不是重复建设

Context History 回答下一次模型应看什么；Rollout 回答进程退出后怎样重放会话语义；ThreadStore 和 State DB 服务分页、搜索与元数据；AgentGraphStore 保存不能从聊天文本可靠推断的父子拓扑。

多权威的必要条件是每层回答不同问题，并且重建方向明确。规范 JSONL 已 Flush、SQLite 尚未更新时， 后者是可追赶投影；不能让派生索引反向覆盖 Rollout。Graph 写入失败时也不能从一条“已创建 Agent” 文本猜出完整拓扑。

这种分层的代价是真实的：短暂不一致、迁移、压缩引用保留、跨进程锁和更多故障点。小型 Runtime 可以从内存 History + JSONL 开始；只有分页、搜索、归档或可恢复 Agent 拓扑出现后再增加其他层。

无论存储多少层，Thread Rollback 都只替换有效会话历史。文件改动、已运行进程和远端操作不属于对话日志事务，不能被一并撤销。

多 Agent 首先是一组资源约束

子 Agent 能把探索、测试和日志分析移出主上下文，也可以并行处理互不依赖的任务。每个子 Agent 同时 复制模型采样、工具资源、历史和恢复状态，因此消耗的总 token 通常更多。

Codex 将多 Agent 资源拆成注册容量、活动执行容量、已加载 Residency 和持久 AgentGraph。Agent “已经创建”“正在运行”“当前驻留内存”和“可以恢复”是四种不同状态。一个 `max_agents` 计数器无法 表达这些约束，也无法安全处理并发 Spawn reservation。

值得迁移的是独立性门槛、有界 Mailbox、单一 final notification 和资源分层。写热点相同、依赖链 串行或结果无法压缩的任务，单 Agent 配合工具并发往往更合适。父 Agent 必须继续拥有需求、冲突合并和最终验证，不能把三份未核对摘要直接拼成结论。

扩展能力必须保留信任边界

Skill 是工作流指令，MCP 是外部服务协议，Hook 是生命周期外部命令，Plugin 是安装和分发单元，Extension Contributor 是宿主进程内的强类型能力。它们可以组合，但运行位置和失败半径不能因为统一 “插件体验”而消失。

只需指导模型时用 Skill；需要认证服务和工具 Schema 时用 MCP；需要在已有生命周期点阻断/通知时用 Hook；需要分发组合能力时用 Plugin；只有完全受信且必须贡献宿主对象的能力才做进程内 Extension。

不可信代码进入 Extension 可以崩溃或阻塞整个 Session。反过来，用 Hook 模拟原生 Tool 也缺少正确的 typed executor 与 ToolPlan 绑定。选择扩展面的第一问题是“代码在哪里运行、能做什么、失败由谁吸收”， 而不是 API 看起来是否统一。

Mini Codex 的测试边界

Mini Codex 当前有 27 项离线测试、1 项需要凭据才运行的 live smoke test，46 个 Python 源文件通过 mypy strict。它还提供 request fingerprint 录制回放和 20 Turn 单机微基准。

这些证据证明的是具体不变量：Interrupt 只产生一个 Aborted 终态；Writer 部分成功后只重试未确认 后缀；Registry 热更新不改变旧 ToolPlan；进程超时会 kill/reap；Mailbox 有反压。它们不证明完整 MCP、跨平台 OS sandbox、多 Agent Session、长时间压力或生产 SLA。

测试的正确读法是：

```
源码命题
→ 最小跨语言实现
→ 在提交点注入故障
→ 断言完整事件和残留状态
→ 结论只返回原命题
```

如果一段最小实现无法表达取消由谁拥有、写入何时确认或旧 ToolCall 使用哪个 Router，那么问题通常不是 Python 不够像 Rust，而是研究还没有提炼出真正合同。

从零搭建时的建议顺序

先建立领域 Item、Operation/Event 和 Session owner，再增加 Cancellation 与 terminal-once；随后把 Prompt 做成纯编译，把 request/step 状态做成快照；再用同一 ToolPlan 生成 specs/handlers，建立完整 Call/Result 配对和副作用屏障。

只有需要执行不可信动作时加入 OS 隔离，需要跨进程恢复时加入规范日志，需要查询时加入派生索引， 需要动态外部能力时加入 MCP/Plugin，需要被测量证明的并行收益时加入多 Agent。

这个顺序没有要求复制 Codex 的 Rust 类型和 crate。换成 TypeScript、Go、actor model 或数据库队列， 只要以下问题仍有确定答案，合同就还在：谁拥有状态；变化何时可见；副作用何时发生；失败后剩什么； 恢复从哪份权威记录开始。

不照搬，也不能过度简化

Codex 的多 Surface、Transport、三平台沙箱、legacy Rollout、动态扩展和多 Agent 兼容矩阵，都有产品 需求背景。目标系统没有这些需求时，不应预先支付相同成本。

但“保持简单”不能删除核心正确性：Call/Result 配对不是产品装饰，ToolPlan 同快照不是大规模专属，取消终态和副作用屏障也不能等出事故后再补。正确裁剪位于两种极端之间：不复制未被需求触发的产品 层，也不删除故障下维持协议自洽的 Runtime 合同。

这套审查方法不规定唯一架构图。每增加一个层次，都要写清触发需求、状态所有者、协议变化、失败残留、迁移路径和可执行证据。回答不了其中任一项，就暂时不应把这层复杂度加入系统。

章内索引