前言

这本书不是源码导读，也不是 API 文档。

它记录了一个 agent 运行时 — pi — 在设计过程中做出的关键决策。每一章回答一个设计问题，源码只在需要解释"为什么这样做"时出场。

这本书适合谁

- 有工程经验、想真正理解 agent 系统设计的开发者
- 准备给 pi 生态做扩展或贡献源码的人
- 想在自己的项目中借鉴 agent 架构设计经验的技术负责人

阅读准备

前置知识

- **TypeScript**：需要能读懂类型定义、泛型、async/await
- **LLM API 概念**：了解 prompt、tool calling、streaming 的基本概念
- **不需要**：不需要用过 pi，不需要 Node.js 框架经验

推荐阅读路径

路径 A：架构师（想了解设计决策）

> 第 1 章 → 第 8 章 → 第 10 章 → 第 30-32 章

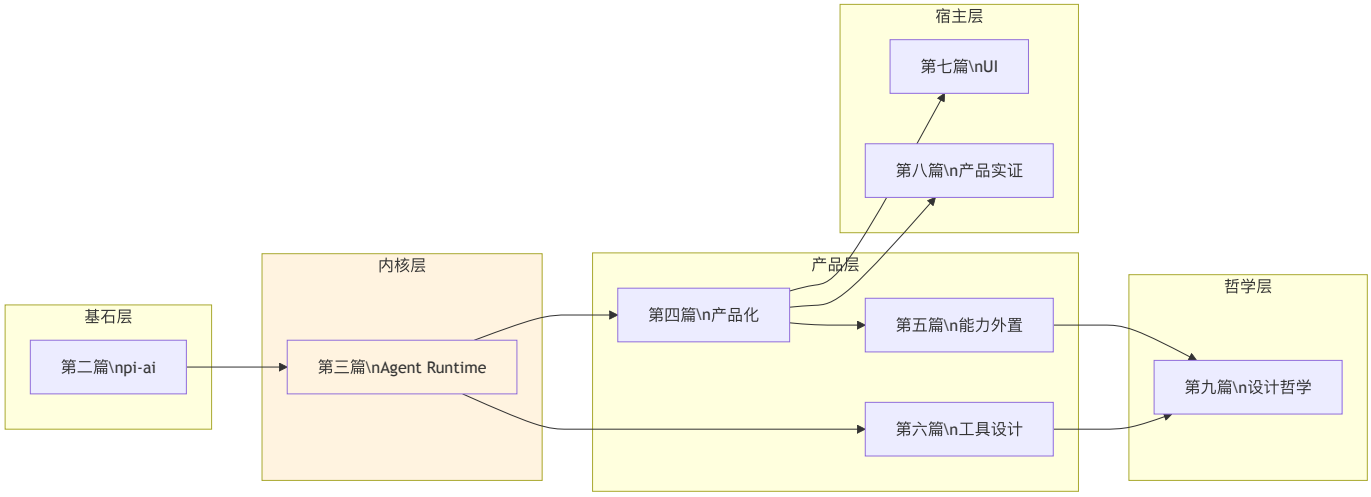
路径 B：开发者（想贡献代码或写 extension）

> 第 2-3 章 → 第 8-10 章 → 第 15-16 章 → 第 19 章 → 附录 D

路径 C：完整阅读（从头到尾）

> 按目录顺序

全书知识地图



标记说明

- **源码引用**：`packages/agent/src/agent-loop.ts:155-232` — 文件路径 + 行号范围
- **取舍分析**：每章末尾的"得到了什么 / 放弃了什么"段落
- **Mermaid 图**：架构图、流程图、时序图嵌入在章节中
- **版本演化说明**：每章末尾标注分析基于的版本和后续变化

版本基线

> **当前对应版本：pi（pi-mono）v0.82.1**（2026 年 7 月 25 日发布）。全书内容已按该版本完成对照核实。

本书核心分析基于 **pi-mono v0.66.0**（2026 年 4 月），并已对照 **v0.82.1**（2026 年 7 月）核实。核心设计决策在这段时间里保持稳定；期间最显著的仓库结构变化（包数量的收缩与再扩张、新增 server / sqlite-node / evals 三个包）见第 2 章，各章末尾的「版本演化说明」标注了各自的对照结论。

第 1 章：不是又一个 LLM 包装器

> **定位**：本章建立全书的阅读框架。 前置依赖：无。 适用场景：当你想快速判断"这本书值不值得读"。

pi 到底是什么

AI 编程助手正在从"产品形态"回到"基础设施层"。市场上有上百个 AI 编码工具，但支撑它们的基础设施 — agent runtime — 仍然是一个没有共识的领域。

pi 是一个 **agent 运行时**，不是一个调用库。

调用库（LangChain、Vercel AI SDK）的核心问题是"怎么调 LLM"。Runtime 的核心问题是"调完之后怎么办" — 模型返回了工具调用，工具执行了，结果返回了，然后呢？要不要继续？要不要重试？用户在 agent 工作过程中发了新消息怎么办？上下文窗口快满了怎么办？工具执行超时了怎么办？多个工具调用之间如何排序？

这些问题没有一个能被一次 LLM API 调用解决。它们需要的是一个**运行循环** — 一个持续运转、不断决策的引擎。pi 的核心就是这样一个引擎。

这本书不讲"怎么用 pi"，而是讲 **pi 做了哪些设计决策，每个决策放弃了什么、得到了什么**。读完后你应该能判断：这些决策适不适合你的场景。

与同类项目的结构差异

在进入 pi 的设计细节之前，值得先看看同一领域内几个有代表性的项目在结构上的差异。这里不是评价好坏，而是指出架构选择上的不同方向。

LangChain：链式编排

LangChain 的核心抽象是 **chain** — 一系列步骤的有向图。每个步骤可以是 LLM 调用、检索、工具执行等。开发者手动编排这些步骤的顺序和条件分支。

pi 没有 chain 的概念。pi 的循环引擎（`agentLoop`）是一个**无限循环**：调用 LLM → 执行工具 → 把结果送回 LLM → 重复，直到 LLM 决定停止。开发者不编排步骤，而是提供工具和 prompt，让 LLM 自己决定调用什么、调用几次。

结构差异在于：LangChain 的控制流是开发者定义的图（developer-defined graph），pi 的控制流是 LLM 驱动的循环（LLM-driven loop）。前者更可预测，后者更灵活。

Vercel AI SDK：流式调用库

Vercel AI SDK 的核心是**统一的流式 LLM 调用接口**。它解决的问题是：不同 provider（OpenAI、Anthropic、Google）的 API 格式不同，AI SDK 提供统一的 `streamText` / `generateText` 入口。

pi 的 `pi-ai` 层做了类似的事 — 统一的 provider 抽象和事件流。但 pi 在这之上多了两层：`pi-agent-core`（循环引擎 + 状态管理）和 `pi-coding-agent`（产品内核）。Vercel AI SDK 止步于"怎么调 LLM"，pi 继续回答"调完之后怎么管理整个 agent 生命周期"。

结构差异在于：AI SDK 是一个**调用层**（call layer），pi 是一个**运行时**（runtime）。AI SDK 可以被 pi 替代掉底层调用部分（事实上 pi 自己实现了这层），但反过来不成立 — AI SDK 没有循环引擎。

CrewAI：多 agent 编排

CrewAI 的核心抽象是 **crew** — 多个 agent 组成的团队，每个 agent 有角色（role）、目标（goal）和工具（tools）。框架负责协调 agent 之间的协作。

pi 明确选择**不内建 sub-agents**（第 31 章详述）。pi 认为 sub-agent 的抽象在当前阶段收益不明确 — 一个 agent 调用另一个 agent，本质上和一个 agent 调用一个工具没有区别，但引入了额外的消息传递、状态同步、错误传播等复杂性。

结构差异在于：CrewAI 是**多 agent 框架**（multi-agent framework），pi 是**单 agent 运行时**（single-agent runtime）。pi 认为单 agent + 强大的工具集 > 多个弱 agent 的协作，至少在 coding 领域是如此。

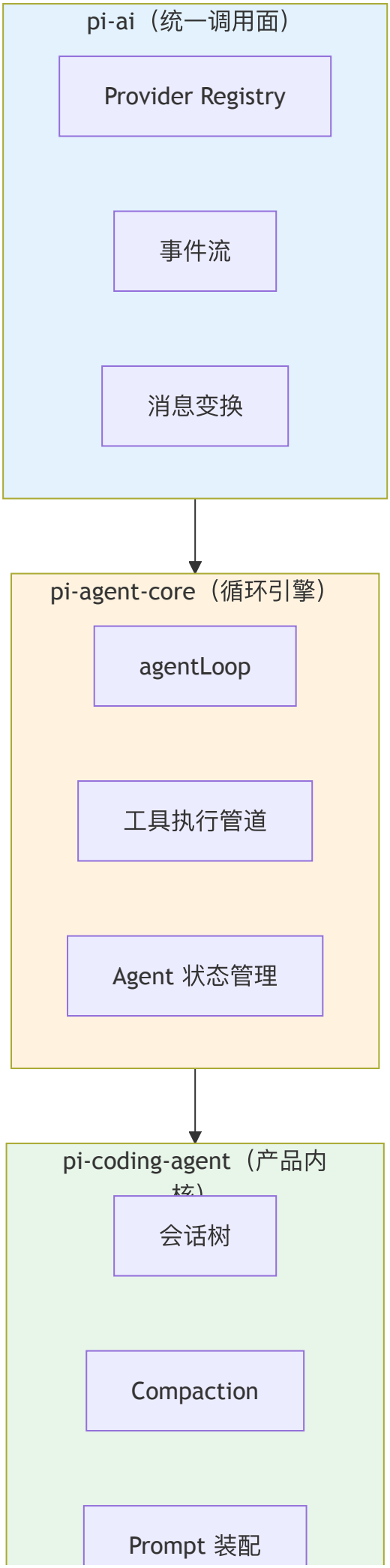
差异总结

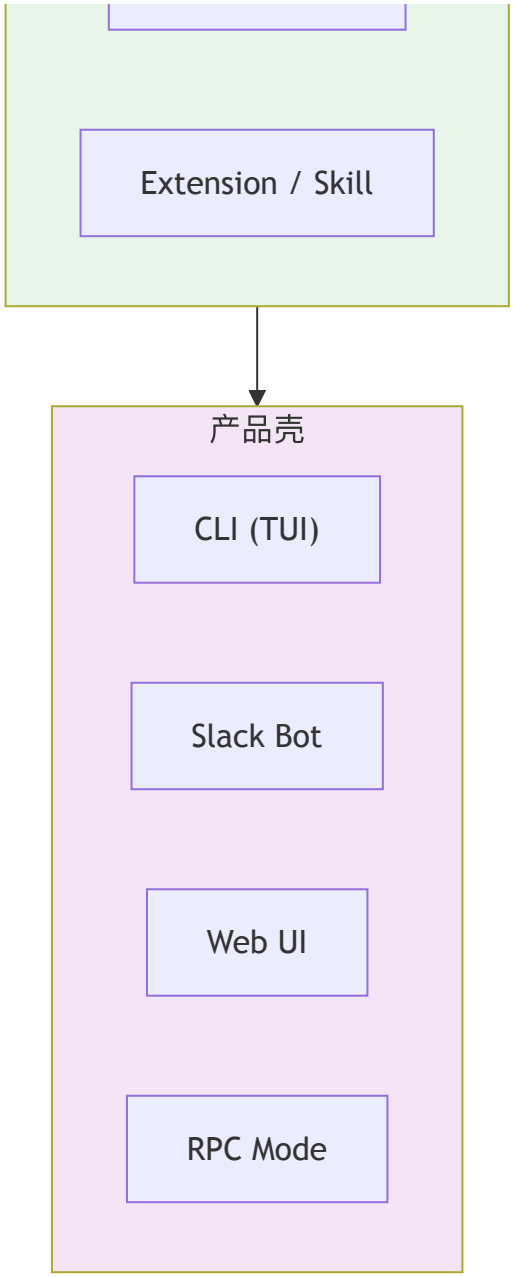
维度	LangChain	Vercel AI SDK	CrewAI	pi
核心抽象	Chain（步骤图）	StreamText（调用）	Crew（agent 团队）	AgentLoop（循环引擎）
控制流	开发者定义	开发者定义	框架协调	LLM 驱动
分层数	单层 + 插件	单层	双层	四层（ai → agent → coding → 产品壳）
状态管理	外部	无	内置	分层（循环无状态，agent 有状态）
Multi-agent	通过 LangGraph	无	核心特性	明确不做

这张表不是评分卡。每个项目的选择都是对其目标场景的优化。pi 的选择优化的是**单 agent 在复杂编码任务中的深度能力**。

洋葱架构

pi 的设计可以用一张洋葱图概括：


pi-ai（统一调用面）
Provider Registry
事件流
消息变换
pi-agent-core（循环引擎）
agentLoop
工具执行管道
Agent 状态管理
pi-coding-agent（产品内
核）
会话树
Compaction
Prompt 装配



每一层只知道下一层的接口，不知道上层的存在。**依赖只向内**。这条规则没有例外。

L1：pi-ai — 统一调用面

最内层解决一个纯粹的问题：如何用统一的接口调用不同的 LLM provider。

Provider 集合 是一个极简的统一调用面： `models.ts` 里的 `Models` 把一批 `Provider` 显式组装成一个集合，支持 Anthropic、OpenAI、Google、Bedrock、Mistral 等。定义一个新 provider 至少要提供 `api` 标识、 `stream` 这组接口面，然后把它加进集合。（早期这里是一个全局的 `api-registry.ts` 注册表，v0.80.0 后被显式的 `Models` 集合取代，详见第 4 章。）

事件流 是 pi-ai 的核心输出格式。无论哪个 provider，调用结果都被标准化为一系列事件： `text-delta` 、 `tool-call-delta` 、 `usage` 等。下游代码完全不需要知道底层

provider 的 API 格式。

消息变换 处理不同 provider 的消息格式差异。Anthropic 用 content blocks，OpenAI 用 function calling — 这些差异在 L1 内部被消化掉，上层看到的永远是统一的 `AssistantMessage`。

L2：pi-agent-core — 循环引擎

中间层解决的问题是：LLM 返回了工具调用，然后呢?

agentLoop 是整个系统的心脏。它是一个 `while(true)` 循环：调用 LLM → 检查是否有工具调用 → 执行工具 → 把结果加入消息列表 → 再次调用 LLM。循环在两种情况下终止：LLM 没有产生工具调用（认为任务完成），或者外部信号要求停止。

关键设计：agentLoop **本身是无状态的**。它不持有任何跨次调用的状态。消息列表、工具定义、配置 — 全部由调用方传入。这意味着循环引擎可以被任何上层以任何方式复用（第 8 章详述）。

工具执行管道 负责并行执行工具、处理超时、收集结果。它不关心工具做了什么 — tool 的实际实现在上层定义。

Agent 状态管理 是循环引擎上面的一层薄壳。它持有消息历史、当前配置、abort 信号等。当 agentLoop 需要这些信息时，Agent 提供；当 agentLoop 产生新消息时，Agent 记录。

L3：pi-coding-agent — 产品内核

这一层把通用的 agent 引擎变成一个具体的编码助手。

会话树 管理对话的分支结构。用户可以在任何一轮对话后回退、分支，形成一棵树状的会话历史。这不是 agent 引擎的通用功能，而是编码助手这个产品的需求。

Compaction 是上下文窗口管理。当消息历史接近 context window 上限时，compaction 把旧消息压缩成摘要。这是一个有损操作 — 压缩后的摘要会丢失细节 — 但它让 agent 可以持续工作而不会因为 context window 满了而中断。

Prompt 装配 把系统 prompt 的各个部分（基础指令、用户自定义、项目上下文文件如 AGENTS.md）组装成最终发给 LLM 的 system prompt。这是一个看似简单但细节极多的过程（第 14 章详述）。

Extension / Skill 是能力外置的机制。Extension 是代码模块（可以注册新工具、新命令、事件处理器和 UI 扩展），Skill 是指令文档（Markdown 格式，告诉 agent 如何完成特定任务）。两者都通过 Resource Loader 统一加载。

L4：产品壳

最外层是面向终端用户的界面。CLI（TUI）是主要交互方式，Slack Bot 把 coding agent 接入团队协作，Web UI 提供浏览器交互，RPC Mode 允许编程方式调用。

这一层的代码量最大（TUI 有 35+ 组件），但设计上最简单 — 它只是 L3 的消费者。所有的核心逻辑都在内层。

为什么是"运行时"而不是"框架"

这个区分值得展开。框架（framework）提供骨架，开发者在骨架中填充业务逻辑 — 控制流由框架决定（所谓 "inversion of control"）。运行时（runtime）提供执行环境，开发者编写完整的程序在环境中运行 — 控制流由开发者决定。

pi 的定位更接近运行时：

循环引擎不强制控制流。agentLoop 的 `while(true)` 循环确实控制了 "LLM 调用 → 工具执行 → 再次调用" 的基本循环，但循环何时开始、何时终止、消息如何传入传出、工具如何定义 — 这些全部由调用方决定。你可以在一次用户交互中启动循环、在任意时刻中止、在循环结束后修改消息历史再重新启动。

没有强制的项目结构。pi 不要求你的项目遵循特定的目录布局或配置格式。Extension 是一个 TypeScript 文件，导出一个工厂函数 — 就这些。没有 decorator、没有 annotation、没有继承链。

不隐藏底层。pi-ai 层提供了统一的 provider 抽象，但如果你需要直接访问底层 provider 的原始 API（比如 Anthropic 的 prompt caching），可以直接导入 `@earendil-works/pi-ai/anthropic` 使用 provider 特定的功能。抽象是可穿透的。

这不是说 "运行时" 比 "框架" 更好。框架的优势是**降低入门门槛** — 开发者不需要理解完整的系统就能开始使用。pi 的运行时定位意味着开发者需要更多的理解成本，换来的是更多的控制权。

本书不涉及的内容

为了明确阅读预期，以下是本书**不**涵盖的内容：

不讲怎么用 pi。这不是用户手册。不会教你怎么安装、怎么配置 API key、怎么使用各种命令。pi 有独立的 README 和文档来做这件事。

不讲 prompt engineering。虽然 pi 的 system prompt 装配是一个重要话题（第 14 章），但本书关注的是"system prompt 是怎么组装的"，而不是"怎么写出更好的 prompt"。

不讲具体 provider 的 API 细节。Anthropic Messages API 的参数、OpenAI Responses API 的格式 — 这些是各 provider 的文档该讲的内容。本书只关注 pi 如何抽象掉这些差异。

不讲 TUI 组件的实现细节。35+ 个 UI 组件的渲染逻辑、交互处理是工程实现，不是设计决策。本书讨论 TUI 层的架构（第 24-27 章），但不会逐组件讲解。

不做框架推荐。本书不会得出"pi 比 X 更好"的结论。每个设计决策都标注取舍 — 得到了什么、放弃了什么。你根据自己的场景做判断。

阅读方法

这本书不是源码导读。不会逐文件介绍"这个函数做什么"。

每一章回答一个设计问题：

- 为什么循环引擎是无状态的？（第 8 章）
- 为什么 skill 是 markdown 而不是代码？（第 16 章）
- 为什么不内建 sub-agents？（第 31 章）

源码只在需要解释"为什么这样做"时出场。如果你只想了解设计哲学，可以跳过所有代码块。如果你想深入实现，代码块提供了精确的文件和行号引用。

建议的阅读路径有三种：

快速扫描路（2 小时）：读第 1、2、3 章了解全局，然后跳到第 30-32 章看设计哲学总结。

设计理解路（1-2 天）：按章节顺序读，跳过所有代码块。每章关注"设计问题"和"取舍分析"两个部分。

深入实现路（1 周）：按章节顺序读，配合源码。每章末尾的代码引用提供了精确的文件和行号，可以直接跳到对应位置阅读完整实现。

版本演化说明

> 本书核心分析基于 pi-mono v0.66.0（2026 年 4 月），并已对照 v0.82.1（2026 年 7 月）核实。 本章的洋葱四层模型在此期间保持不变；仓库结构层面的变化（包数量的收缩与再扩张、新增 server / sqlite-node / evals 三个包）见第 2 章。

第 2 章：包不是项目

> **定位**：本章建立全书最重要的一张分层图。 前置依赖：第 1 章。 适用场景：当你想快速理解 pi-mono 的全局架构。

如何把一个庞大的 agent 系统切成互相不知道对方的层?

pi-mono 是一个 npm workspace monorepo，当前有 7 个 workspace 业务包 — 其中 6 个对外发布，1 个是私有评测包。但它们不是"7 个独立项目"，而是同一个系统里分工不同的层与配件。理解 pi 的架构，第一步就是看清这些分层线怎么切、为什么这么切。

> **关于包数量的演化**：这个 monorepo 的包数从来不是恒定的。它最初只有 3 个包（ai、agent、tui），一度扩张到 7 个（额外有 `pi-mom` Slack bot、`pi (pods)` GPU 编排、`pi-web-ui` Web 组件）；这三者各自分化出独立的使用者与发布节奏后被移出主仓库，收缩回 4 个；此后又长出 `pi-storage-sqlite-node`、`pi-server`、`pi-evals` 三个包，回到 7 个。数量在反复变，判断标准从没变（详见本章末尾「为什么是这些包」一节）。第 27/28/29 章仍保留对三个已迁出包的设计分析，并在章首标注为历史快照。

Workspace 配置

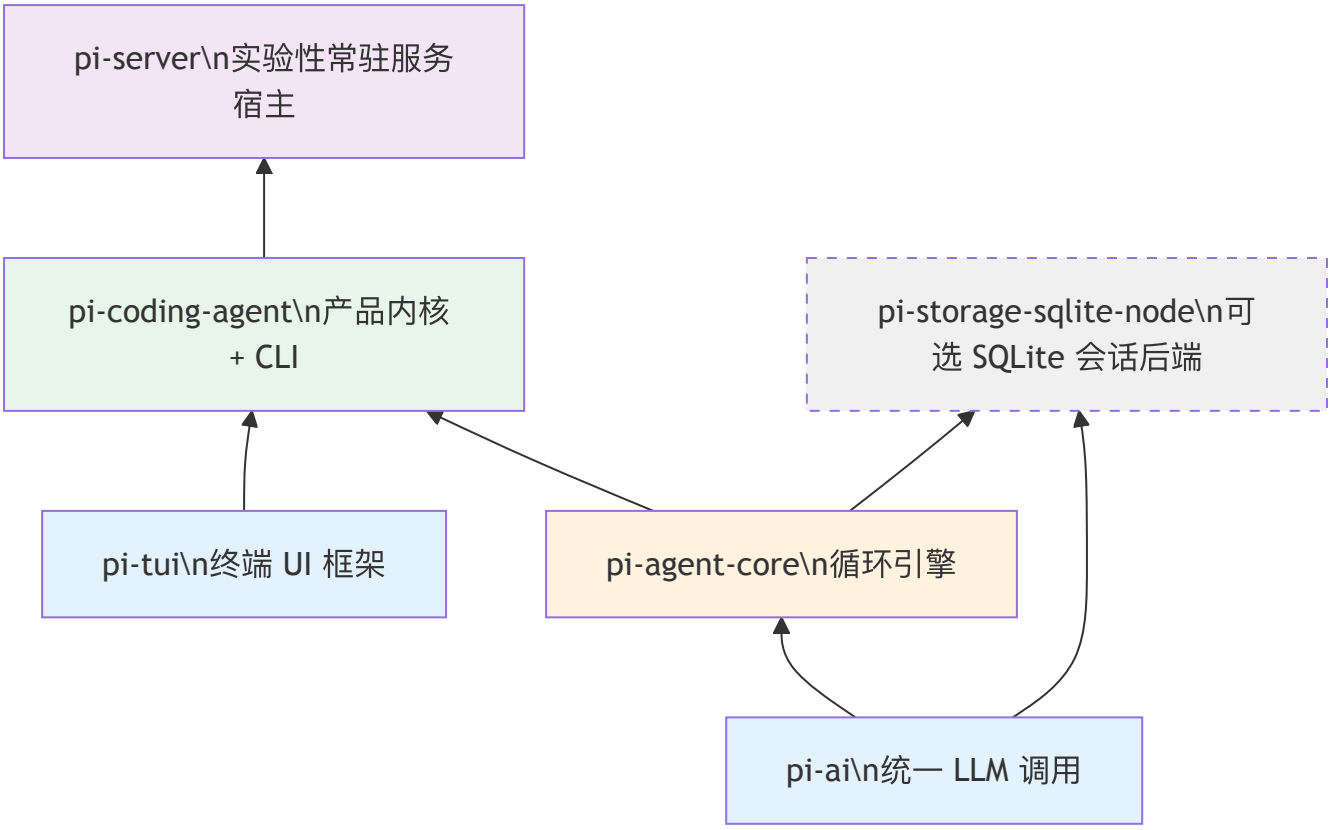
先看根目录的 `package.json`，它定义了 monorepo 的边界：

```
// file: package.json:5-13
{
  "workspaces": [
    "packages/*",
    "packages/storage/*",
    "packages/coding-agent/examples/extensions/with-deps",
    "packages/coding-agent/examples/extensions/custom-provider-
anthropic",
    "packages/coding-agent/examples/extensions/custom-provider-gitlab-
duo",
    "packages/coding-agent/examples/extensions/sandbox",
    "packages/coding-agent/examples/extensions/gondolin"
  ]
}
```

`packages/*` 覆盖了直接放在 `packages/` 下的六个包，`packages/storage/*` 则把嵌套一层的 `packages/storage/sqlite-node` 也纳入 workspace —— 这条 glob 是 v0.81 新增 SQLite 存储后端时补上的，漏了它 sqlite-node 的 `node_modules` 就不会被正确链接。其余的 workspace 入口是 extension 示例项目 — 它们是 extension 的示范实现，同样要参与 npm 的依赖解析。

注意根 `package.json` 的 `"private": true` — 这个 monorepo 本身不发布。7 个业务包里有 6 个发布到 npm（`pi-tui`、`pi-ai`、`pi-agent-core`、`pi-coding-agent`、`pi-storage-sqlite-node`、`pi-server`），只有 `pi-evals` 标了 `"private": true`，作为内部评测 harness 不对外发布。

分层图



依赖箭头只向上 — 底层的包不知道上层的存在。`pi-ai` 不知道 `pi-agent-core` 的存在，`pi-agent-core` 不知道 `pi-coding-agent` 的存在。图里两个较新的节点是这条主干上的"配件"和"新楼层"：`pi-storage-sqlite-node`（虚线框）是挂在 `pi-ai` / `pi-agent-core` 侧面的可选会话存储后端，没有任何已发布包依赖它；`pi-server` 则坐在产品层之上、是唯一依赖 `pi-coding-agent` 的包，把 pi 包装成常驻服务，处在分层图的第 5 层。

三个较新的包：server、sqlite-node、evals

上一轮收缩到 4 个包之后，仓库又长出三个包。它们的共同点是都没有被塞进 `pi-coding-agent`，因为各自的使用者和发布边界都和产品内核不同。这里只登记定位，展开分析留给后文或未来版本：

`pi-server` （实验性）。目录最初以 `orchestrator` 之名落地（commit `7ece19b0`），v0.81.0 改名 `server`（commit `8495f9d0`，#6898）。它把 pi 包装成一个常驻服务：一个 supervisor 监督进程、若干 RPC 子进程、进程间用 IPC 通信，对外提供 HTTP 服务，内部复用 `pi-coding-agent` 的内核（这也是它唯一的运行时依赖）。它的 README 明确标注 **Experimental**，CLI 与 API 都可能变更甚至移除。正因如此，本书**不为 server 单开**

一章 —— 一个接口尚未稳定的包，架构展开的价值会被后续改动不断稀释；等它稳定下来再补章更划算。

`pi-storage-sqlite-node`（可选后端）。基于 Node 内置的 `node:sqlite` 实现的一个 SQLite 会话存储后端（`SqliteSessionRepo` + migrations + 物化视图），供 `pi-agent-core` 的会话持久化选用。它是一个**可选配件**：没有任何已发布包依赖它，当前唯一的使用方是 agent 包里的测试。把它单独成包，正是因为"用 SQLite 存会话"有独立的使用者，又不该强塞进内核。

`pi-evals`（私有评测）。基于 `vitest-evals` 的评测 harness（#7085），用 `createHarness` 驱动真实的 `createAgentSession` 跑评测。它标了 `"private": true`，不发布到 npm —— 是内部质量工具，不是对外产品。（顺带一提，evals 曾因沿用改名前的 `@mariozechner/pi-ai` 别名依赖导致发布失败，commit `6173017a` 把它切回直接基于 `vitest-evals` 构建，这是上一轮 scope 迁移遗留的最后一条长尾。）

依赖关系的实际验证

每个包的 `package.json` 中的 `dependencies` 字段精确地记录了这些依赖关系。我们可以从源码中直接验证：

pi-ai（`@earendil-works/pi-ai`）：零内部依赖。它只依赖外部 SDK — `@anthropic-ai/sdk`、`openai`、`@google/genai`、`@mistralai/mistralai`、`@aws-sdk/client-bedrock-runtime` 等。这是整个系统的最底层。

pi-agent-core（`@earendil-works/pi-agent-core`）：唯一的内部依赖是 `pi-ai`。

```
// file: packages/agent/package.json
{
  "dependencies": {
    "@earendil-works/pi-ai": "^0.82.1"
  }
}
```

极其克制 — 整个循环引擎只有一个内部依赖。

pi-tui（`@earendil-works/pi-tui`）：零内部依赖。它只依赖 `chalk`、`marked`、`get-east-asian-width` 等纯 UI 工具库。TUI 框架完全独立于 AI 系统。

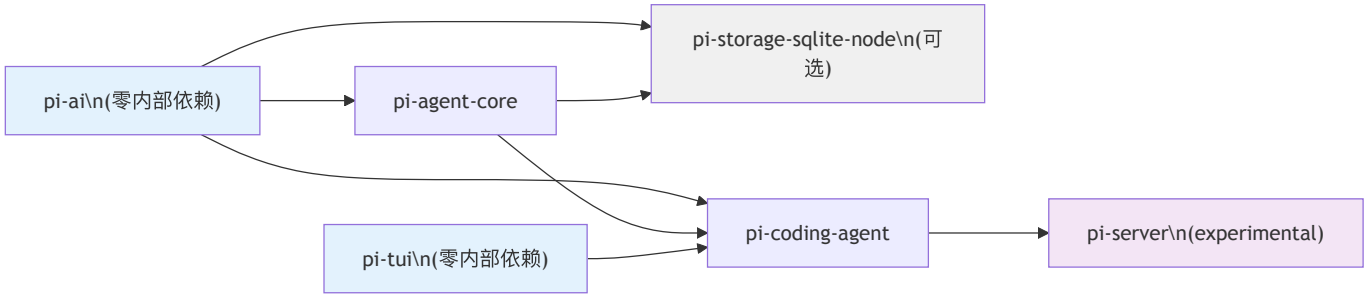
pi-coding-agent（`@earendil-works/pi-coding-agent`）：依赖三个内部包。

```
// file: packages/coding-agent/package.json
{
  "dependencies": {
    "@earendil-works/pi-agent-core": "^0.82.1",
    "@earendil-works/pi-ai": "^0.82.1",
    "@earendil-works/pi-tui": "^0.82.1"
  }
}
```

这是依赖最重的包 — 它是三个底层包的汇聚点，把 AI 调用、agent 循环和终端 UI 整合成一个编码助手产品。

两个新包的内部依赖同样遵守"箭头只向上"的原则： `pi-storage-sqlite-node` 依赖 `pi-ai` 与 `pi-agent-core` （同为 `^0.82.1` ）， `pi-server` 只依赖 `pi-coding-agent` （ `^0.82.1` ）。所有交叉依赖的版本号都锁在同一个 `^0.82.1` 上，这是下文 lockstep 版本策略的直接结果。

依赖关系图（按 direct dependencies 绘制）



这张图的关键特征：没有环。箭头严格从底层指向上层。这不是偶然的 — 它是设计约束。 `pi-coding-agent` 仍是产品内核的汇聚点，三个底层包都不知道它的存在； `pi-server` 又在它之上再叠一层，而 `pi-storage-sqlite-node` 是挂在下层旁边的可选分支。整张图依然无环。

包的规模与职责

包名	npm 名	主要 exports
pi-ai	`@earendil-works/pi-ai`	`stream()`, Provider Registry, Model 类型
pi-agent-core	`@earendil-works/pi-agent-core`	`agentLoop()`, `Agent`, `AgentHarness`, 类型定义
pi-coding-agent	`@earendil-works/pi-coding-agent`	CLI 入口, Session Manager, 工具集, Extension API, SDK
pi-tui	`@earendil-works/pi-tui`	Terminal 渲染引擎, 编辑器组件
pi-storage-sqlite-node	`@earendil-works/pi-storage-sqlite-node`	`SqliteSessionRepo`, SQLite 适配器（可选会话后端）
pi-server	`@earendil-works/pi-server` （experimental）	`server` CLI, supervisor, RPC 子进程, IPC / HTTP
pi-evals	`@earendil-works/pi-evals` （private）	评测 harness（`pi-harness.ts`），不发布

几个值得关注的数字（数量级以 v0.66 基线为参考，后续版本持续增长）：

pi-agent-core 只有 5 个源文件、~1,900 行代码。这是整个 agent 系统的循环引擎。它的极简性不是偶然 — 循环引擎故意只做"循环"这一件事，把所有业务逻辑推给上层。

pi-coding-agent 有 129 个源文件、~42,100 行代码。它占了整个项目的近半数代码量。这也是符合预期的 — 产品层需要处理大量具体的工程问题：130+ 个工具实现、会话管理、prompt 组装、extension 加载、配置覆盖等。

pi-ai 有 ~26,900 行代码。这些代码的大部分是各 provider 的实现（Anthropic、OpenAI、Google、Bedrock、Mistral 等）和自动生成的模型目录。核心抽象很薄：`models.ts` 的 `Models` 集合与 `Provider` 接口把这一切收拢到一个统一调用面之后（v0.80.0 后旧的 `api-registry.ts` 已删除，详见第 4 章）。

每个包的导出策略

pi-mono 中的包在 npm 发布时的导出策略各不相同，反映了它们面向不同使用者的设计意图。

pi-ai 的多入口导出。pi-ai 不仅导出主入口（`@earendil-works/pi-ai`），还为每个 provider 提供独立的子路径导出（`@earendil-works/pi-ai/anthropic`、`@earendil-works/pi-ai/openai-responses` 等）。这让使用者可以只导入需要的 provider，避免把所有 provider 的 SDK 都拉进依赖树。OAuth 支持也是独立子路径（`@earendil-works/pi-ai/oauth`）。

pi-coding-agent 的 hooks 导出。除了主入口，pi-coding-agent 还导出 `@earendil-works/pi-coding-agent/hooks` — 这是给 extension 开发者用的。Extension 需要引用 hooks 的类型定义，但不应该依赖整个 coding-agent 包的内部实现。单独的子路径导出实现了这种选择性暴露。

pi-tui 和 pi-agent-core 的单入口导出。这两个包只有一个导出入口，因为它们的 API 面足够小 — 没有需要独立拆分的子模块。

bin 字段。`pi-coding-agent` 导出 `pi` 命令（`"bin": { "pi": "dist/cli.js" }`），`pi-ai` 导出 `pi-ai` 命令，实验性的 `pi-server` 导出 `server` 命令。这些是面向终端用户的入口点。`pi-tui`、`pi-agent-core`、`pi-storage-sqlite-node` 没有 bin — 它们是纯库，不提供 CLI。

构建顺序

npm workspace 不保证构建顺序。如果你运行 `npm run build`，npm 会并行构建所有包 — 但包之间有依赖关系，并行构建会失败。

pi-mono 通过在根 `package.json` 的 `build` 脚本中**手动编排构建顺序**来解决这个问题：

```
// file: package.json:16
{
  "build": "cd packages/tui && npm run build && cd ../ai && npm run build && cd ../agent && npm run build && cd ../storage/sqlite-node && npm run build && cd ../../coding-agent && npm run build && cd ../server && npm run build"
}
```

构建顺序是：`tui` → `ai` → `agent` → `storage/sqlite-node` → `coding-agent` → `server`。随着包数从 4 涨到 7，这条链也从 4 步扩成了 6 步。

这个顺序必须满足一个约束：**每个包在构建时，它所依赖的包必须已经构建完成**。让我们验证：

1. `tui`：零内部依赖，可以最先构建
2. `ai`：零内部依赖，可以最先构建（与 tui 并行也可以）
3. `agent`：依赖 `ai`，必须在 `ai` 之后
4. `storage/sqlite-node`：依赖 `ai + agent`，必须排在两者之后
5. `coding-agent`：依赖 `ai + agent + tui`，必须在三者之后
6. `server`：依赖 `coding-agent`，排在整条链的最后

私有的 `pi-evals` 不参与这条构建链 —— 它不发布、也没有被任何包依赖，跑评测时由根脚本 `npm run eval` 单独驱动。

对于一个内部项目来说，串行构建的简单性和可调试性比省几秒构建时间更有价值。

注意 `pi-ai` 的构建有一个特殊步骤：

```
// file: packages/ai/package.json:65
{
  "build": "npm run generate-models && npm run generate-image-models &&
tsgo -p tsconfig.build.json"
}
```

构建前先运行 `generate-models`（和 `generate-image-models`）— 这个脚本从各 provider 的 API 拉取最新的模型目录，生成 `models.generated.ts`（第 18 章详述）。这意味着每次完整构建都会拿到最新的模型列表。

Lockstep 版本

所有包始终使用同一个版本号（当前 v0.82.1）。每次发布，全部包一起升版；lockstep 覆盖全部 7 个业务包（含私有的 evals）。

得到了什么：永远不会有"pi-ai v0.65 和 pi-agent-core v0.66 不兼容"的问题。开发者看到一个版本号就知道整个系统的状态。CI/CD 流程简单 — 一个脚本升版、一个脚本发布。

放弃了什么：一个只影响 `pi-tui` 的 bug fix 也要升全部 7 个包。但对于一个内部高度耦合的系统，lockstep 的简单性远胜于独立版本的灵活性。

版本管理通过根目录的脚本完成：

```
// file: package.json:23-25
{
  "version:patch": "npm version patch -ws --no-git-tag-version && node scripts/sync-versions.js && ...",
  "version:minor": "npm version minor -ws --no-git-tag-version && node scripts/sync-versions.js && ..."
}
```

`npm version patch -ws` 同时升所有 workspace 的版本，`sync-versions.js` 确保交叉依赖中的版本号也同步更新（比如 `pi-agent-core` 的 `dependencies` 中引用的 `@earendil-works/pi-ai` 版本号）。

为什么是这些包：一条 3 → 7 → 4 → 7 的曲线

包的数量不是随意的。pi-mono 的包划分遵循一个原则：**当且仅当两段代码有不同的使用者时，它们才应该在不同的包中**。

`pi-ai` 单独成包，因为有人只想用统一 LLM 调用而不需要 agent 循环。`pi-tui` 单独成包，因为终端 UI 框架与 AI 无关 — 它甚至可以用于非 AI 的 TUI 应用。`pi-agent-core` 单独成包，因为有人想用循环引擎构建非编码类 agent。

反过来，`pi-coding-agent` 没有被进一步拆分为 "工具包"、"session 包"、"prompt 包"，因为这些部分没有独立的使用者 — 没有人只要 pi 的工具系统而不要 session 管理。过度拆分只会增加包之间的版本协调成本而没有实际收益。

这个原则解释了包数量**来回的多次变化**。初期 pi-mono 只有 3 个包（ai、agent、tui）；随着 Slack bot、GPU 编排、Web UI 等产品形态的出现，曾扩张到 7 个包（增加了 `pi-mom`、`pi (pods)`、`pi-web-ui`）。但当这三者各自发展出独立的使用者群体、独立的迭代节奏，继续留在主仓库里反而违背了"不同使用者 → 不同包 → 不同仓库"的纪律 —— 于是它们先后被移出为独立项目：`pi-mom` 与 `pi (pods)` 随 commit `0ed0d434`（2026-04-30）移除（mom 的方向性继任为 GitHub `earendil-works/pi-chat`），`pi-web-ui` 随 commit `b141e1fa`（2026-05-20）移除。主仓库收回到聚焦的 4 个包。

收缩之后又是一轮扩张：`pi-storage-sqlite-node`、`pi-server`、`pi-evals` 三个包先后长出来，包数回到 7。关键在于，这一轮扩张用的仍是**同一把尺子**，而不是推翻它 —— 每个新包都对应一群"内核不该背、却确有独立使用者"的代码：想用 SQLite 存会话的人拿到 `pi-storage-sqlite-node` 这个可选后端，想把 pi 跑成常驻服务的人拿到 `pi-server`（尽管它还标着 experimental），需要拿真实 agent 会话跑评测的内部流程拿到

`pi-evals`（它 private，不外发）。三者都没有被塞进 `pi-coding-agent`，因为它们各自的使用者和发布边界都和产品内核不同。曲线从 3 走到 7、退回 4、再回到 7，尺子始终没变。

移出、加入、再加入，用的是同一把尺子：代码是否有独立的使用者，以及这个独立性是否强到值得一条独立的发布边界。独立性弱，就留在 `pi-coding-agent` 内部；强到值得单独发布，就成为 `packages/` 下的一个新包；强到需要自己的迭代节奏和仓库，就像 mom / pods / web-ui 那样彻底离开。第 27/28/29 章保留了对这三个已迁出包的设计分析，作为这把尺子的实证案例。

取舍分析

得到了什么

强制的分层纪律。npm 包是硬边界 — 如果 `pi-ai` 试图 import `pi-agent-core` 的代码，TypeScript 编译器会直接报错。这比"团队约定不要跨层调用"强得多。

独立测试。每个包有自己的 `vitest` 测试。测试 `pi-agent-core` 时不需要启动任何 UI，测试 `pi-tui` 时不需要配置 API key。

渐进式采用。外部开发者可以只使用 `pi-ai`（统一 LLM 调用），不需要引入 agent 引擎；也可以使用 `pi-agent-core`（循环引擎）来构建自己的 agent，不需要依赖 coding agent 的产品层逻辑。

放弃了什么

开发环境复杂度。每个包意味着一套 `tsconfig.json` 和构建流程。`dev` 脚本需要用 `concurrently` 同时启动多个包的 watch 模式。新贡献者需要理解 monorepo 的工作方式。

发布流程的原子性要求。lockstep 版本意味着每次发布必须全部成功或全部回滚。如果某个包的发布失败了，已经发布成功的包也需要处理（虽然实际上各包独立发布到 npm，部分失败时版本号已经消耗掉了）。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，并已对照 v0.82.1（2026 年 7 月）核实。包数量走过 一条"3 → 7 → 4 → 7"的曲线：最初 3 个（ai、agent、tui），扩张到 7 个，于 v0.66 之后将 `pi-mom` 、 `pi (pods)` （commit `0ed0d434` ）与 `pi-web-ui` （commit `b141e1fa` ）移出、 收回到 4 个，随后又新增 `pi-storage-sqlite-node` 、 `pi-server` （experimental）、 `pi-evals` （private）三个包回到 7 个（其中 6 个发布、evals 私有）。构建链相应从 4 步扩为 6 步（ `tui → ai → agent → storage/sqlite-node → coding-agent → server` ），交叉依赖与 lockstep 版本同步到 `^0.82.1` 。npm scope 从 `@mariozechner/` 迁移到 `@earendil-works/` （commit `551385e4` / `3e5ad67e` ）。分层原则和 lockstep 版本策略从未改变。

第 3 章：怎样高效阅读这个仓库

> **定位**：本章告诉读者怎样高效读码，避免迷失在细节中。 前置依赖：第 2 章。 适用场景：当你打算直接看源码。

先读的 10 个文件

优先级	文件	理由
1	`packages/agent/src/types.ts`	定义了 Agent 系统的全部类型
2	`packages/ai/src/models.ts`	`Models` 运行时 + `createProvider`：provider 注册与模型解析核心
3	`packages/agent/src/agent-loop.ts`	循环引擎核心
4	`packages/agent/src/agent.ts`	有状态壳
5	`packages/ai/src/index.ts`	pi-ai 层的公共导出面
6	`packages/ai/src/types.ts`	Model、Context、Event 定义
7	`packages/coding-agent/src/core/session-manager.ts`	会话树
8	`packages/coding-agent/src/core/system-prompt.ts`	Prompt 装配

优先级	文件	理由
9	`packages/coding-agent/src/core/tools/edit.ts`	工具设计范例
10	`packages/coding-agent/src/core/extensions/types.ts`	Extension API 面

每个文件你会看到什么

1. **`packages/agent/src/types.ts`** — 这是整个 agent 系统的 "schema"。你会在这里找到 `AgentMessage`（所有消息类型的联合）、`AgentTool`（工具的定义接口）、`AgentEvent`（循环引擎产出的事件流）和 `AgentLoopConfig`（循环的配置项）。理解这些类型后，你看任何其他文件都能立刻知道数据流的形状。这个文件不长，但信息密度极高 — 建议逐行读完。

2. **`packages/ai/src/models.ts`** — pi-ai 的运行时集合 `Models` 与 provider 工厂 `createProvider` 所在。它取代了早期那个 98 行的 `api-registry.ts` 全局注册表：v0.80 起 provider 不再挂到一个隐式全局表上，而是显式地注册进一个 `Models` 运行时对象，模型解析和流式调用的分发都从这里出发。整个系统"如何添加一个新 provider、如何按 `model.api` 找到实现"的答案都在这个文件里（这次"隐式全局单例 → 显式依赖注入集合"的范式迁移，来龙去脉见第 4 章）。

3. **`packages/agent/src/agent-loop.ts`** — 循环引擎的完整实现。你会看到一个 `while(true)` 循环，里面的逻辑是：调用 LLM → 收集事件 → 如果有 tool call 就执行 → 把 tool result 放回消息列表 → 继续循环。关键点是：这个函数是**纯函数式**的 — 它的所有输入（消息、工具、配置）都从参数传入，所有输出都通过事件流返回。第 8 章会详细分析这个设计。

4. **`packages/agent/src/agent.ts`** — agentLoop 的有状态包装器。你会看到 `Agent` 类持有消息历史、工具列表、abort controller 等状态，然后在内部调用 agentLoop。它的存在回答了一个问题：如果循环引擎是无状态的，状态保存在哪里？答案是这个薄壳。它是 "stateful convenience layer"。

5. **`packages/ai/src/index.ts`** — pi-ai 层的公共导出面。你会在这里一眼看全 ai 层对外暴露了什么：`Models` 运行时与 provider 工厂、`auth/` 认证子系统、以及 `types.ts` 里的核心类型。想知道"从外部能怎么用 pi-ai"，从这个出口读起最快；至于一次 LLM 调用具体怎么按 `model.api` 分发到各 provider，第 4-6 章会顺着 `Models` 展开。

6. **`packages/ai/src/types.ts`** — pi-ai 层的类型系统。你会找到 `Model` 类型（一个模型的完整元数据：id、provider、cost、context window 等）、`Context` 类型（发送给 LLM 的完整上下文：messages、tools、system prompt 等）和各种事件类型。这些类型定义了 pi-ai 层的 "公共 API 契约" — 上层代码（agent-core）只通过这些类型与 ai 层交互。

7. **`packages/coding-agent/src/core/session-manager.ts`** — 会话的持久化和分支管理。你会看到会话如何被序列化到磁盘（JSONL 文件）、如何创建分支（fork）、如何回退到历史节点。这是让 pi 的对话可以"时间旅行"的关键机制。如果你想理解 pi 的交互模型为什么比简单的聊天框更强大，这是入口。

8. **`packages/coding-agent/src/core/system-prompt.ts`** — Prompt 装配的完整逻辑。你会看到 system prompt 是如何从多个来源（基础模板、AGENTS.md 文件、用户自定义、extension 注入）层层拼装的。理解这个文件的关键在于：system prompt 不是一个静态字符串，而是一个运行时动态组装的产物。这解释了为什么 pi 可以根据项目上下文、用户配置、已加载的 skill 动态调整 agent 的行为。

9. **`packages/coding-agent/src/core/tools/edit.ts`** — 文件编辑工具的实现。你会看到一个完整的工具定义：tool schema（告诉 LLM 这个工具接受什么参数）、执行逻辑（实际操作文件系统）、结果格式（返回什么给 LLM）。这是理解 "pi 的工具是怎么设计的" 最好的范例 — 不是因为它最复杂，而是因为它最能代表设计模式。读完一个工具，你就知道其他所有工具的结构。

10. **`packages/coding-agent/src/core/extensions/types.ts`** — Extension API 的完整类型定义。你会看到 Extension 可以做什么：注册新工具（tools）、注册新命令（commands）、快捷键和 CLI flag，以及通过 context 访问会话和 UI。这个文件定义了 pi 的**可扩展性边界** — Extension 能做什么、不能做什么，全由这些类型决定。

最后读的 10 个文件

文件	理由
`packages/tui/src/components/editor.ts`	交互细节，不是设计核心
`packages/coding-agent/src/modes/interactive/components/*.ts`	35+ UI 组件，按需查看

文件	理由
`packages/ai/src/providers/anthropic.ts` 等	具体 provider 实现
`packages/coding-agent/src/core/slash-commands.ts`	命令列表，不是设计
`packages/coding-agent/src/modes/interactive/theme/theme.ts`	UI 主题
`packages/ai/src/models.generated.ts`	自动生成的模型目录
`packages/coding-agent/src/core/export-html/`	HTML 导出，工程实现
`packages/tui/src/components/select-list.ts`	列表组件渲染，UI 细节
`packages/coding-agent/src/modes/rpc/`	RPC 模式，另一种宿主实现
`packages/coding-agent/src/core/extensions/loader.ts`	Extension 加载机制，工程实现

这些文件不是不重要 — 它们只是不应该**先**读。它们是"设计的消费者"，不是"设计本身"。

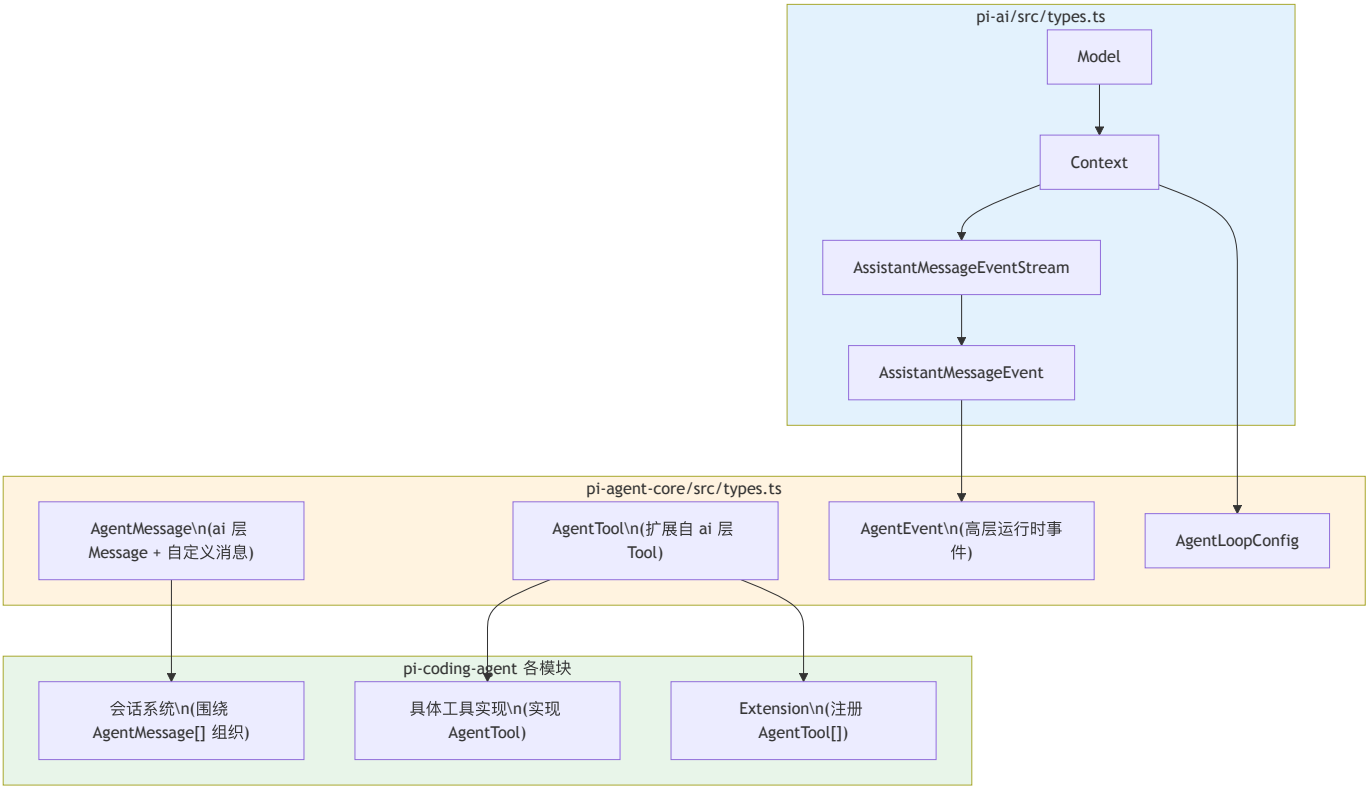
三个不在核心阅读路径上的包

除了上面涉及的四个核心包，仓库里还有三个包不在阅读清单里，因为它们不属于"理解内核"的必经之路：`packages/server`（实验性常驻服务宿主，坐在 coding-agent 之上，CLI/API 尚未稳定）、`packages/storage/sqlite-node`（可选的 SQLite 会话存储后端，没有任何已发布包依赖它）、`packages/evals`（`vitest-evals` 驱动的私有评测 harness）。想理解 pi 的核心设计时可以先跳过它们；需要时第 2 章给了它们各自的定位。

类型如何串连整个系统

pi 的设计中，类型不仅是编译器检查的工具，更是**跨层通信的契约**。理解类型的依赖链，就理解了系统的数据流。

类型依赖链



从图中可以看到三个关键的类型边界：

pi-ai → pi-agent-core： `toolResult` 其实已经在 ai 层的 `Message` 联合里了。agent-core 在此基础上做的主要扩展有两件事：一是通过 `CustomAgentMessages` 开放自定义消息类型，二是把 ai 层的 `AssistantMessageEvent` 嵌入 `AgentEvent.message_update` 这类更高层的运行时事件中。

pi-agent-core → pi-coding-agent：coding-agent 的 `AgentSession` / `SessionManager` 围绕 `AgentMessage[]` 来组织会话上下文。具体的工具（edit、bash、read 等）实现 `AgentTool` 接口。Extension 向系统注册新的 `AgentTool[]` 。

泛型边界： `Model<TApi>` 和 `StreamFunction<TApi, TOptions>` 这类 provider 层接口是泛型的，用来约束“这个模型对应哪种 API 族”。但到了通用的 `Context` ，消息和工具格式已经被统一，不再按 provider 继续分化。也就是说，类型安全主要集中在 provider/stream 边界，而不是把 `Context` 一路参数化到底。

跟着一个类型读代码的例子

假设你想理解 "工具调用的结果是怎么流回 LLM 的"。可以这样跟踪：

1. 在 `agent/src/types.ts` 中找到 `AgentTool` — 它的 `execute` 方法返回 `AgentToolResult`
2. 在 `agent/src/agent-loop.ts` 中搜索 `AgentToolResult` 或 `ToolResultMessage` — 你会看到循环引擎把工具执行结果包装成 `ToolResultMessage` 并加入消息列表
3. 在 `ai/src/types.ts` 中找到 `ToolResultMessage` — 它是 `Context.messages` 数组中合法的消息类型之一
4. 在 `ai/src/providers/anthropic.ts` 中搜索 `ToolResultMessage` — 你会看到它被转换成 Anthropic API 要求的 `tool_result` content block

整个链路：`工具执行 → AgentToolResult → ToolResultMessage → 消息列表 → Provider 转换 → API 请求`。每一步都有对应的类型。

阅读策略

不要从 TUI 开始读。TUI 组件有 35+ 个文件，大量的交互细节。它们是"上层消费者"，不是"设计核心"。先理解内核（第 1-6 项），再看 TUI 如何消费事件。

不要从 provider 实现开始读。`packages/ai/src/providers/anthropic.ts` 等文件是 `Models` 运行时的具体分发目标，不是设计本身。先理解 `models.ts` 里 provider 如何注册、如何被按 `model.api` 分发，再看具体实现。

跟着类型走。`agent/src/types.ts` 定义了 `AgentMessage`、`AgentEvent`、`AgentTool`、`AgentLoopConfig` — 这些类型串起了整个系统。从类型出发，看哪些函数使用它们。

用搜索代替目录浏览。pi-mono 现在有 7 个 workspace 业务包、400 多个源文件（`pi-coding-agent` 与 `pi-ai` 两个包占了大头，`server` / `sqlite-node` / `evals` 合计不到 30 个）。逐文件浏览效率极低。更好的策略是：找到一个你关心的类型或函数名，在整个仓库中搜索它的使用处。工具推荐：`grep -rn "AgentTool" packages/`。

常见阅读误区

在阅读 pi 源码时，有几个容易踩的坑：

误区 1：把 agentLoop 当作唯一入口

很多人一上来就找 `main` 函数或启动入口。pi 的启动路径是 `cli.ts` → `session-manager.ts` → `agent.ts` → `agent-loop.ts`，但**理解系统不应该从启动路径开始**。启动路径包含大量初始化逻辑（参数解析、配置读取、资源加载），这些会分散你对核心设计的注意力。

正确做法：先读 `agent-loop.ts` 理解循环引擎，再倒推它的调用者是谁。

误区 2：试图理解所有 provider 实现

pi-ai 支持 10+ 个 provider（Anthropic、OpenAI Responses、OpenAI Completions、Google、Bedrock、Mistral、Azure、Copilot、OpenRouter、Vercel Gateway 等）。每个 provider 有不同的 API 格式和特殊处理。

如果你试图读完所有 provider 再理解系统，你会花很长时间且收获不大。正确做法：只读 `anthropic.ts`（最简洁的实现），理解 provider 的接口契约后，其他 provider 按需查看。

误区 3：忽略 `models.generated.ts`

这个文件有上万行，是自动生成的模型目录。很多人看到它会直接跳过。但它的存在本身就是一个设计决策：pi 为什么选择在构建时生成模型目录，而不是运行时从 API 查询？（答案：离线可用 + 启动速度。第 18 章详述。）

正确做法：不需要读它的内容（确实是机器生成的），但需要理解它为什么存在、怎么生成。

误区 4：把 Extension 和 Skill 混为一谈

Extension 是**代码模块**（TypeScript/JavaScript），可以注册新工具、新命令、事件处理器和 UI 扩展。Skill 是**指令文档**（Markdown），只能被注入到 system prompt 中。两者的能力边界完全不同。

很多人看到 "Extension 可以提供 skill 路径" 就以为 Extension 包含 Skill。准确的关系是：Extension 可以**注册额外的 Skill 加载路径**，但 Extension 本身和 Skill 是两种独立的资源类型。

误区 5：低估 types.ts 的重要性

agent/src/types.ts 只有几百行，看起来"没什么内容"。但这个文件是整个 agent 系统的骨架。每个类型的每个字段都对应着一个设计决策。比如 `AgentLoopConfig` 中的 `toolExecution` 字段 — 为什么需要区分 `sequential` 和 `parallel`？因为它直接影响工具 prepare 阶段和 execute 阶段的并发语义（第 9 章详述）。

正确做法：types.ts 至少读两遍。第一遍建立整体印象，读完几章设计分析后再回来读第二遍 — 你会发现每个字段都有了具体的含义。

版本演化说明

> 本章阅读路线以 pi-mono v0.66.0 的目录结构为骨架，并已对照 v0.82.1（2026 年 7 月）核实： 引用的文件路径均在当前仓库存在，pi-ai 的 provider 注册入口已从早期的 `api-registry.ts` / `stream.ts` 迁移到 `models.ts` （`Models` 运行时，详见第 4 章）。仓库现有 7 个 workspace 业务包（结构见第 2 章）。文件路径可能随版本继续变化，但"先读内核、后读产品层"的策略不变。

第 4 章：Provider 不是 Adapter

> **定位**：本章解剖 pi-ai 的核心抽象 — 如何用一组显式的 `Provider` 运行时单元统一 20+（现约 38）家 LLM 厂商，以及为什么它从早期的全局注册表演化成今天的 `Models` 集合。 前置依赖：第 2 章（分层架构）。 适用场景：当你想理解如何设计一个多 provider LLM 抽象层，或者想为 pi 添加新的 LLM 供应商。

20+ 家厂商，如何用一个接口统一？

这是本章的核心设计问题 — 但它有两个层次。

第一个层次是**抽象**：用什么数据结构描述一次 LLM 调用，才能让 Anthropic、OpenAI、Google、Bedrock 等协议各异的厂商共享同一套调用代码？这个答案（`Model` 值对象 + Provider/Api 双维度）从 pi 诞生至今基本没变。

第二个层次是**装配**：这些 provider 实现如何被组织、被发现、被注入到调用点？这个答案在 v0.80.0 发生了一次全局性的破坏性重构 — 从**隐式的全局副作用注册表**迁移到**显式的 `Models` 依赖注入集合**。这是 pi-ai 这一年里最大的一次架构演进，也是本章的新主线。

我们先讲不变的抽象，再讲变化的装配，最后用一节历史对照说明"为什么当初的注册表要被换掉"。

打开 `packages/ai/src/types.ts`，前 30 行定义了 pi 对 LLM 世界的全部认知：

```
// packages/ai/src/types.ts:16-28

export type KnownApi =
  | "openai-completions"
  | "mistral-conversations"
  | "openai-responses"
  | "azure-openai-responses"
  | "openai-codex-responses"
  | "anthropic-messages"
  | "bedrock-converse-stream"
  | "google-generative-ai"
  | "google-vertex"
  | "pi-messages";

export type Api = KnownApi | (string & {});
```

注意 `Api` 类型的设计：它是 `KnownApi`（已知的 10 种 API 协议）加上 `(string & {})`（任意字符串）的联合。这个看似奇怪的 `(string & {})` 是 TypeScript 的一个技巧——它让类型系统对已知值提供自动补全，同时允许任意新值。内建的 10 种协议有 IDE 提示，自定义协议可以用任意字符串注册。

> **版本提示**：v0.71.0 移除了 `google-gemini-cli`（连同 Antigravity 一并删除），一度把 `KnownApi` 降到 9 种；v0.80.8 又新增了 `pi-messages`（Radius 网关使用的原生协议），现为 10 种。

`KnownProvider` 联合列出了约 38 个已知供应商（`types.ts:34-72`，v0.66 时为 23 个，随 DeepSeek、Moonshot、Fireworks、xAI、Kimi Coding、Qwen/Xiaomi Token Plan 等内建化而持续增长）。这里有一处**重要的类型订正**：早期这个联合直接叫 `Provider`，写作 `type Provider = KnownProvider | string`。v0.80.0 起它改名为 `ProviderId`：

```
// packages/ai/src/types.ts:73

export type ProviderId = KnownProvider | string;
```

改名不是美学问题，而是为新架构腾出名字：`Provider`（不带 Id）现在指**运行时的 provider 接口**（`models.ts:75`，本章后面详述），而 `ProviderId` 才是那个"厂商名字符串"的联合类型。凡是 `Model`、`AssistantMessage`、`ToolResultMessage` 上标识"哪个厂商"的字段，类型都是 `ProviderId`，不再是 `Provider`。

这里隐含了 pi 最重要的设计决策之一：**Provider 和 Api 是两个独立的维度**。

一个 provider（比如 `"google"`）可能暴露多种 api（`"google-generative-ai"` 和 `"google-vertex"`）。一种 api 协议（比如 `"openai-responses"`）可能被多个 provider 使用（`"openai"`、`"azure-openai-responses"`、`"github-copilot"`）。如果把 provider 和 api 绑死，每增加一个 Azure OpenAI 部署就要写一个新 provider。分离之后，Azure OpenAI 只需注册一个使用 `"azure-openai-responses"` api 的 provider。

Model<TApi> — 携带一切上下文的值对象

在理解 provider 之前，先看它操作的核心数据：`Model`。

```
// packages/ai/src/types.ts:749-776

export interface Model<TApi extends Api> {
  id: string;
  name: string;
  api: TApi;
  provider: ProviderId;
  baseUrl: string;
  reasoning: boolean;
  thinkingLevelMap?: ThinkingLevelMap;
  input: ("text" | "image")[];
  cost: ModelCost;
  contextWindow: number;
  maxTokens: number;
  headers?: Record<string, string>;
  compat?: /* 基于 TApi 的条件类型 */ ;
}
```

`Model` 不只是一个"模型名称"，它是一个**自描述的值对象**，携带了调用一个 LLM 所需的全部元信息：身份（`id` / `name` / `provider`）、协议（`api`）、能力（`reasoning` / `input`）、约束（`contextWindow` / `maxTokens`）、经济（`cost` 精确到输入/输出/缓存读/缓存写四维）。注意 `provider` 字段的类型是 `ProviderId`，不再是旧稿写的 `Provider`。

为什么 Model 是泛型的?

`Model<TApi>` 的泛型参数 `TApi extends Api` 是整个类型系统的支点。它的作用不在于 Model 本身的字段差异（大部分字段对所有 api 都一样），而在于**向下传播协议信息**。

看 `compat` 字段的条件类型（`types.ts:767-775`）：`TApi` 是 `"openai-completions"` 时 `compat` 为 `OpenAICompletionsCompat`；是 `"openai-responses"` 系时为 `OpenAIResponsesCompat`；是 `"anthropic-messages"` 时为 `AnthropicMessagesCompat`；是 `"bedrock-converse-stream"` 时为 `BedrockCompat`；其余为 `never`。这些 compat 类型编码了同一协议下不同厂商的兼容性开关（第 18 章详述）。

更重要的是，`Model<TApi>` 的泛型参数会传递给 `StreamFunction`（`types.ts:311-315`）：

```
export type StreamFunction<
  TApi extends Api = Api,
  TOptions extends StreamOptions = StreamOptions
> = (
  model: Model<TApi>,
  context: Context,
  options?: TOptions,
) => AssistantMessageEventStream;
```

当一个 provider 声明自己的 stream 函数为 `StreamFunction<"anthropic-messages", AnthropicOptions>` 时，TypeScript 保证传入的 `model` 一定是 `Model<"anthropic-messages">`，`options` 一定是 `AnthropicOptions`。每个 provider 的实现在**类型层面就知道自己服务的是哪种协议**，不需要运行时判断。

Provider 是一个运行时单元，不是注册表条目

这里是新架构的枢纽。在旧设计里，"provider" 是全局注册表 Map 里的一个条目 `{ api, stream, streamSimple }` — 一组无状态的函数。在 v0.80.0 之后，`Provider` 是一个**自足的运行时对象**，它自己知道：它是谁、怎么认证、有哪些模型、怎么发起流。

```ts
// packages/ai/src/models.ts:75-120（节选）

export interface Provider<TApi extends Api = Api> {
  readonly id: string;
  readonly name: string;
  readonly baseUrl?: string;
  readonly headers?: ProviderHeaders;

  /** 至少要有 apiKey / oauth 之一：连纯环境变量、纯本地服务器
   *  这样的 provider 也提供 apiKey 认证，其 resolve() 报告是否已配置。 */
  readonly auth: ProviderAuth;

  /** 当前已知模型（同步）。静态 provider 返回目录；
   *  动态 provider 返回上次 refreshModels() 的结果。不得抛异常。 */
  getModels(): readonly Model<TApi>[];

  /** 仅动态 provider：恢复本地缓存并可选地用凭证拉取更新列表。 */
  refreshModels?(context: RefreshModelsContext): Promise<void>;

  filterModels?(models: readonly Model<TApi>[],
    credential: Credential | undefined): readonly Model<TApi>[];

  stream<T extends TApi>(model: Model<T>, context: Context,
    options?: ApiStreamOptions<T>): AssistantMessageEventStream;
  streamSimple(model: Model<TApi>, context: Context,
    options?: SimpleStreamOptions): AssistantMessageEventStream;
}
```

对比旧的两方法接口（`{ stream, streamSimple }`），新 `Provider` 多出了三组能力：**身份**（`id` / `name` / `baseUrl` / `headers`）、**认证**（`auth: ProviderAuth`，第 7 章展开）、**模型目录**（`getModels` / `refreshModels` / `filterModels`）。这意味着 provider 从"一个 api 协议的实现函数"升级为"一家厂商的完整运行时代表"。api 协议的实现（真正把请求发给 Anthropic 的代码）下沉成了 provider **内部**注入的 `ProviderStreams`，通过 `model.api` 分派。

`createProvider`：从零件组装一个 provider

没有人手写上面那个接口的每个方法。所有 provider — 内建的和 models.json 里用户自定义的 — 都经由工厂函数 `createProvider` 组装：

```ts
// packages/ai/src/models.ts:556-623（节选）

export function createProvider<TApi extends Api = Api>(
  input: CreateProviderOptions<TApi>
): Provider<TApi> {
  const baselineModels = input.models;
  let dynamicModels: readonly Model<TApi>[] = [];
  // ... currentModels() 把静态目录与动态覆盖按 id 合并 ...

  const single = typeof (input.api as ProviderStreams).stream ===
"function"
    ? (input.api as ProviderStreams) : undefined;
  const byApi = single ? undefined
    : (input.api as Partial<Record<string, ProviderStreams>>);
  const apiFor = (model) => single ?? byApi?.[model.api];

  return {
    id: input.id,
    name: input.name ?? input.id,
    baseUrl: input.baseUrl,
    auth: input.auth,
    getModels: currentModels,
    refreshModels: input.fetchModels ? /* 恢复缓存 + 拉取 + 落盘 */ :
undefined,
    stream: (model, ctx, opts) =>
      dispatch(model, (s) => s.stream(model, ctx, opts)),
    streamSimple: (model, ctx, opts) =>
      dispatch(model, (s) => s.streamSimple(model, ctx, opts)),
  };
}
```

`createProvider` 的输入 `CreateProviderOptions`（`models.ts:533-548`）就是"一家 provider 的全部配置"：`id`、`auth`、静态 `models` 列表、可选的 `fetchModels`（动态目录）、以及 `api`（单个 `ProviderStreams` 或按 `model.api` 分派的映射表）。它把这些零件组装成上面那个自足对象。

一个真实的 provider factory 只有十几行。以 Anthropic 为例：

```
// packages/ai/src/providers/anthropic.ts:38-50

export function anthropicProvider(): Provider<"anthropic-messages"> {
  return createProvider({
    id: "anthropic",
    name: "Anthropic",
    baseUrl: "https://api.anthropic.com",
    auth: {
      apiKey: anthropicApiKeyAuth(),
      oauth: lazyOAuth({ name: "Anthropic (Claude Pro/Max)",
        load: loadAnthropicOAuth }),
    },
    models: Object.values(ANTHROPIC_MODELS),
    api: anthropicMessagesApi(),
  });
}
```

注意几个设计点： `api: anthropicMessagesApi()` 是一个 **lazy** 的 `ProviderStreams`（真正的 Anthropic SDK 只在第一次流式调用时加载）； `auth.oauth` 用 `lazyOAuth` 包装（OAuth 实现代码在第一次 login/refresh 时才 import）。延迟加载的机制还在，只是从"注册表侧的 `createLazyStream` "下沉到了"每个 factory 内部的 lazy 零件"。

Models：显式的运行时集合

有了自足的 `Provider` ，下一个问题是：谁持有这些 provider？谁在调用点把 `model` 派给正确的 provider？答案是 `Models` 集合。

```
// packages/ai/src/models.ts:127-187（节选）

export interface Models {
  getProviders(): readonly Provider[];
  getProvider(id: string): Provider | undefined;
  getModels(provider?: string): readonly Model<Api>[];
  getModel(provider: string, id: string): Model<Api> | undefined;

  refresh(options?: ModelsRefreshOptions): Promise<ModelsRefreshResult>;
  checkAuth(providerId: string): Promise<AuthCheck | undefined>;
  getAvailable(providerId?: string): Promise<readonly Model<Api>[]>;
  getAuth(model: Model<Api>, overrides?): Promise<AuthResult |
undefined>;
  login(providerId: string, type: AuthType,
    interaction: AuthInteraction): Promise<Credential>;
  logout(providerId: string): Promise<void>;

  stream<TApi extends Api>(model: Model<TApi>, context: Context,
    options?: ModelsApiStreamOptions<TApi>):
AssistantMessageEventStream;
  complete<TApi>(...): Promise<AssistantMessage>;
  streamSimple(...): AssistantMessageEventStream;
  completeSimple(...): Promise<AssistantMessage>;
}
```

`Models` 是一个**接口**，实现是 `ModelsImpl`（`models.ts:218`）。它做四件事：

1. **持有 provider**：内部一个 `Map<string, Provider>`，通过 `MutableModels.setProvider`（`models.ts:191`）增删（`Models` 的可变子接口 `MutableModels` 才暴露 `setProvider` / `deleteProvider` / `clearProviders`）。
2. **路由请求**：`stream(model, ...)` 按 `model.provider` 找到对应 provider，把请求委托给它（`models.ts:489-502`）。
3. **解析认证**：调用 provider 之前，`applyAuth`（`models.ts:463-487`）先经 `getAuth` 解析出 API key / headers / baseUrl，注入到请求选项里再委托。认证从"藏在某个 getApiKey 回调里"变成了集合的一等职责。
4. **刷新目录**：`refresh`（`models.ts:276`）并发刷新所有动态 provider 的模型列表（第 18 章）。

关键在于：**`Models` 是一个值，不是全局单例**。你 `createModels()`（`models.ts:529`）得到一个实例，往里 `setProvider` 你想要的 provider，然后把这个实例作为依赖传给需要它的代码。同一个进程里可以有多个互不干扰的 `Models`（测试尤其受益 — 不再有跨用例泄漏的全局注册表状态）。这就是"隐式全局单例 → 显式依赖注入集合"迁移的核心收益。

`stream` 本身仍是薄委托，只是委托的目标从"全局 Map 查出的函数"变成"集合内按 id 查出的 provider 对象"：

```
// packages/ai/src/models.ts:489-502

stream<TApi extends Api>(model, context, options) {
  return lazyStream(model, async () => {
    const provider = this.requireProvider(model);
    const { requestModel, requestOptions } =
      await this.applyAuth(model, options);
    return provider.stream(requestModel, context, requestOptions);
  });
}
```

providers/all.ts：聚合与 tree-shakeable 拆包

内建 provider 现在集中在 `providers/all.ts`。它导出一个 `builtinProviders()`（`providers/all.ts:87`），返回全部 38 个内建 provider factory 的**新构造实例**；以及 `builtinModels()`（`:131`），一步到位地 `createModels()` 并把它们全部 `setProvider` 进去：

```
// packages/ai/src/providers/all.ts:131-137

export function builtinModels(options?: CreateModelsOptions):
MutableModels {
  const models = createModels(options);
  for (const provider of builtinProviders()) {
    models.setProvider(provider);
  }
  return models;
}
```

拆包是这次重构的另一个目标。每个 provider 都在自己的模块里（`providers/anthropic.ts`、`providers/openai.ts` ……），API 实现按 api id 命名放在 `api/`（`api/anthropic-messages.ts`、`api/openai-responses.ts` ……）。因为 provider 之间不再通过"全局注册表副作用"耦合，一个只用 Anthropic 的下游可以直接 `import { anthropicProvider }`，让打包器 tree-shake 掉其余 37 个 provider 及其 SDK 依赖。旧设计里，只要 `import` 了根入口就会触发 `register-builtins` 的副作用，把所有 provider 的注册代码拖进 bundle。

根入口 core-only，旧全局 API 迁到 /compat

配合拆包，包的**根入口 `index.ts` 变成了无副作用的 core-only**：只导出类型、`Models` / `createModels` / `createProvider` 、auth 类型、事件流工具等，**不导出**生成的模型目录、provider factory、api 实现，也不再有任何"模块加载即注册"的副作用（`index.ts:4-8` 的注释明确写了这一点）。

那些依赖旧全局 API 的代码怎么办？pi 保留了一个**临时兼容入口** `@earendil-works/pi-ai/compat`（源码 `src/compat.ts`）。旧的全局函数 — `stream` / `complete` 、`registerApiProvider` / `getApiProvider` / `getApiProviders` / `unregisterApiProviders` 、以及 `getModel` / `getModels` / `getProviders` — 都在这里以 `@deprecated` 的形式继续导出。这是一条明确的迁移跑道：老代码改一行 import 就能继续跑，新代码则应当直接用 `Models` 集合。

实战：添加一个新 provider 的完整步骤

用一个具体例子走一遍。假设你要添加 DeepSeek，它使用 OpenAI 兼容的 API。

第一步：确定 Api 协议。 DeepSeek 兼容 OpenAI completions，所以直接复用 `"openai-completions"` ，零 api 代码 — 这是 Provider/Api 分离的第一个好处。

第二步：写一个 provider factory。 用 `createProvider` 组装：给它 `id: "deepseek"` 、`auth`（通常 `envApiKeyAuth("DeepSeek API key", ["DEEPSEEK_API_KEY"])` ，见 `auth/helpers.ts:9` ）、静态 `models` 列表、以及复用的 `api: openAICompletionsApi()` 。

第三步：注册进集合。 `models.setProvider(deepseekProvider())` ；若是内建 provider，则把 factory 加进 `providers/all.ts` 的 `builtinProviders()` 数组。

第四步：结束。 用户调用 `models.stream(deepseekModel, context)` ，集合按 `model.provider === "deepseek"` 找到 provider， `applyAuth` 解析出 key，provider 内部按 `model.api === "openai-completions"` 分派给复用的实现。

如果 DeepSeek 用的是**完全不兼容**的私有协议，你才需要：在 `KnownApi` 加一个 id（或用任意字符串）、写一个 `ProviderStreams` 实现 `stream` / `streamSimple` 、在 factory 的 `api` 字段传入它。即便如此，你**不需要触碰任何"注册表"代码** — 因为已经没有注册表了，只有集合与 factory。核心不变，边缘生长。

历史对照：v0.66–v0.79 的全局注册表形态

> 以下机制在 v0.80.0 已被删除，仅作历史对照。它们对应的文件（`api-registry.ts`、`stream.ts`、文本侧 `providers/register-builtins.ts`）在当前源码树中**已不存在**；其公共 API 以 `@deprecated` 形式迁到了 `/compat` 入口。

早期 pi-ai 的枢纽是一个 98 行的 `api-registry.ts`。它维护一个模块级的全局 `Map<string, RegisteredApiProvider>`，公共 API 面只有五个函数：`registerApiProvider`（注册）、`getApiProvider`（查单个）、`getApiProviders`（列全部）、`unregisterApiProviders`（按 `sourceId` 批量注销）、`clearApiProviders`（清空，测试用）。注册时用 `wrapStream` 做一次"类型擦除桥接"：把泛型的 `StreamFunction<TApi, TOptions>` 包装成非泛型函数，在运行时用 `model.api !== api` 检查守住类型边界。这套"入口检查 + 内部擦除"是当时应对"Map 无法表达存在类型"的经典手法。

用户看到的是 `stream.ts`（59 行）导出的 `stream` / `complete` / `streamSimple` / `completeSimple` 四个函数，每个都是"解析 provider + 委托"的一行逻辑。它的第一行 `import "./providers/register-builtins.js"` 是一个**副作用导入** — 模块一加载就执行 `registerBuiltInApiProviders()`，把 9 种内建 provider 灌进全局 Map。每个内建 provider 用 `createLazyStream` 包装：注册时不加载 SDK，第一次调用时才 `import()`，并用 `||=` 缓存 Promise 保证只加载一次。

这套设计当年解决了真问题 — 极简 API 面、Provider/Api 解耦、延迟加载、无限扩展 — 它的很多**设计取舍今天仍然成立**（延迟加载、双方法接口、api 分派都被新架构继承了下来）。但它有两个结构性缺陷，最终促成了 v0.80.0 的重构：

1. **全局可变状态**。注册表是模块级单例。两段代码、两个测试、两个并发的 agent 会话共享同一张表，`sourceId` 批量注销就是为了给这个共享状态打补丁 — 你得记得给每次注册打标签，卸载时按标签清理。这是"全局单例"必然要付的税。
2. **副作用耦合阻碍 tree-shaking**。"import 根入口 = 注册所有内建 provider"意味着任何下游都无法只带走它用的那一家。

新架构用"值语义的集合"替换了"全局的表"：provider 变成自足对象，集合变成可传递的依赖，`sourceId` 批量注销退化为"从你自己的集合里 `deleteProvider`"，副作用注册退化为"显式 `setProvider`"。同一套延迟加载、同一套 api 分派，换了一个装配底座。

取舍分析

得到了什么

1. 消除全局可变状态。`Models` 是值不是单例。测试之间不再泄漏注册状态，多会话可以各持一个集合，认证/凭证也随集合隔离。

2. 真正的 tree-shaking。core-only 的根入口 + 每 provider 独立模块，让下游只打包用到的 provider。旧的"副作用注册"被"显式装配"取代。

3. Provider 升级为完整运行时单元。认证、模型目录、动态刷新都内聚在 provider 自身，而不是散落在注册表、OAuth 注册表、model-registry 三处。`Models` 只负责路由与 auth 应用。

4. 抽象层的取舍被继承。Provider/Api 解耦、双方法接口（`stream` / `streamSimple`）、延迟加载 — 这些旧设计的优点一个没丢，只是换了装配方式。

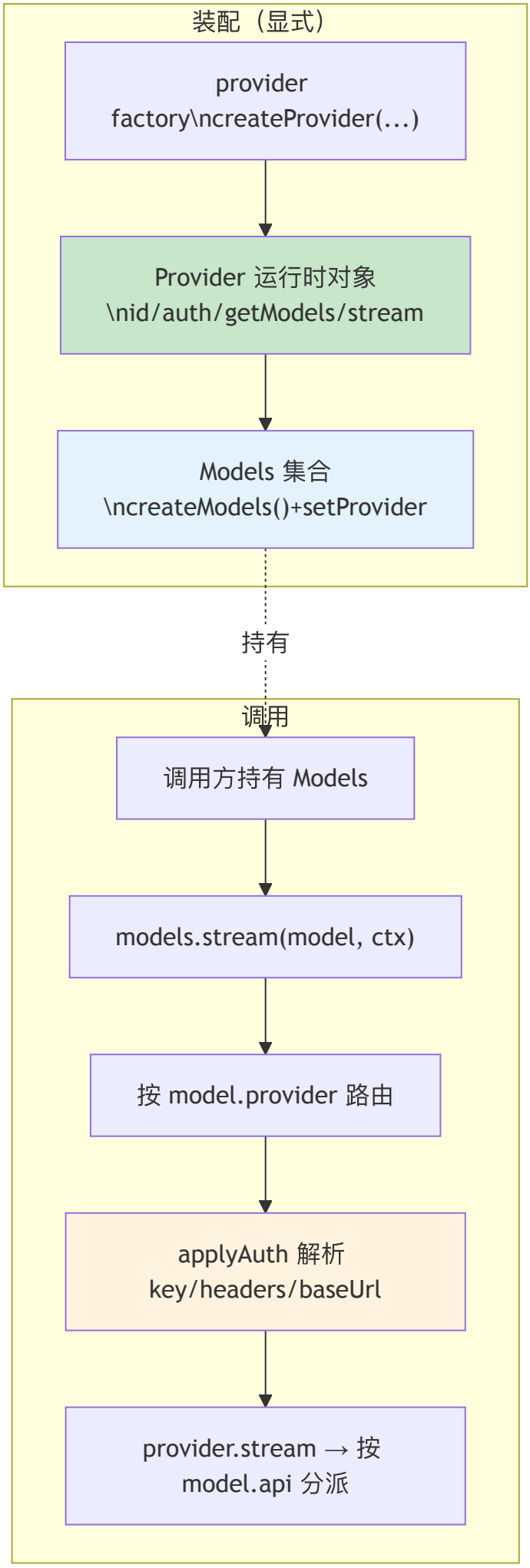
放弃了什么

1. 一次破坏性迁移的成本。所有依赖 `stream()` / `registerApiProvider()` 全局函数的下游都必须迁移。`/compat` 入口把成本摊薄成"改 import"，但它是**临时**的，最终仍要清账。

2. 显式装配的样板。旧代码 `import "pi-ai"` 就能 `stream(model)`；新代码要先 `builtinModels()` 或手动 `createModels()+setProvider()`，再 `models.stream(model)`。多了一步"谁持有集合、集合怎么传下去"的显式接线 — 这正是依赖注入相对于全局单例的固有代价，也是它的全部意义。

3. 运行时才知道 provider 是否可用。这一点两代架构相同：provider 是运行时装配的，配了一个未注册的 provider，错误只在调用时暴露（`requireProvider` 抛 `Unknown provider`）。

对于 pi 的定位 — 支持 38+ 家厂商、允许用户扩展、要被多个宿主（TUI/SDK/RPC）以库的形式复用 — 显式集合是比全局注册表更合适的底座。它用一点装配样板，买到了可测试性、可 tree-shake、以及"provider 是一等运行时对象"的内聚性。


装配（显式）
provider
factory\ncreateProvider(...)
Provider 运行时对象
\nid/auth/getModels/stream
Models 集合
\ncreateModels()+setProvider
持有
调用
调用方持有 Models
models.stream(model, ctx)
按 model.provider 路由
applyAuth 解析
key/headers/baseUrl
provider.stream → 按
model.api 分派

版本演化说明

> 本章内容已对照 pi-mono **v0.82.1**。本章主线在 **v0.80.0** 跨过了一道破坏性分界：
>
> - **v0.80.0（Breaking）**：删除全局 `api-registry.ts` / `stream.ts` /文本侧 `register-builtins.ts`，代之以 `Provider` 运行时接口（`models.ts:75`）、`createProvider`（`:556`）、`Models` / `createModels`（`:127` / `:529`）与 `providers/all.ts` 聚合。旧全局 API（`stream` / `registerApiProvider` / `getApiProvider` / `unregisterApiProviders` 等）迁到 `@earendil-works/pi-ai/compat` 临时入口；根 `index.ts` 变为无副作用 core-only。
> - **类型订正**：`Provider` 联合类型改名 `ProviderId`（`types.ts:73`）；`Provider` 现指运行时接口。`Model.provider`、`AssistantMessage.provider` 等字段类型为 `ProviderId`。
> - **计数**：`KnownApi` 现为 10 种（v0.80.8 新增 `pi-messages`）；`KnownProvider` 约 38 种。早期（v0.66–v0.79）的注册表叙述见上文"历史对照"节；其延迟加载、双方法接口、Provider/Api 解耦等取舍被新架构继承。

第 5 章：消息变换 — 跨模型交接的隐藏复杂度

> **定位**：本章解析用户在不同 LLM 之间切换时，历史消息如何被有损但安全地变换。前置依赖：第 4 章（Provider Registry）。 适用场景：当你想理解为什么跨 provider 对话不只是"换个 API key"，或者想为自己的系统设计消息兼容性层。

用户在 Claude 上聊了 50 轮，现在要切到 GPT — 历史消息怎么办?

这是本章的核心设计问题。

直觉上，LLM 消息就是"角色 + 文本"。但实际上，每家厂商的消息格式都携带了 provider 特有的元数据：

- **Thinking blocks**：Anthropic 的 extended thinking 和 OpenAI 的 reasoning 是加密的、不可跨模型复用的
- **Tool call IDs**：OpenAI Responses API 生成 450+ 字符的 ID（带 `|` 等特殊字符），Anthropic 要求 ID 匹配 `^[a-zA-Z0-9_-]+$`（最多 64 字符）
- **Thought signatures**：Google 的 tool call 携带 `thoughtSignature`（用于思维链上下文复用），其他 provider 不认识
- **Text signatures**：OpenAI 的 text block 携带 `textSignature`（消息元数据，legacy ID 或 `TextSignatureV1` JSON），跨模型时毫无意义
- **Redacted thinking**：安全过滤后的加密内容，只有原模型能解码

`transformMessages()` 函数（`packages/ai/src/api/transform-messages.ts:64`）解决了这些问题。它的策略可以概括为一句话：**尽可能保留，不能保留的安全降级，绝不让变换导致 API 调用失败。**

> **路径订正（v0.80.0）**：这个函数随 API 实现整体从 `providers/transform-messages.ts` 迁到了 `api/transform-messages.ts`，本章所有行号已按当前源码

> 更新。

入口的第一道防线：null content 归一化

在进入变换逻辑之前，`transformMessages` 先做一件防御性的事 — 把来路不明的 `content` 归一化：

```
// packages/ai/src/api/transform-messages.ts:71-73

// Normalize null/undefined content from untyped callers (custom tools,
// hand-built histories, old session files) so downstream code can rely
// on the type contract.
const normalizedMessages = messages.map(
  (msg) => (msg.content == null ? { ...msg, content: [] } : msg));
```

类型上 `content` 不该是 `null`，但历史消息的来源是开放的：自定义工具拼的消息、手工构造的历史、旧版本写下的 session 文件都可能带一个 `content: null`。与其让后续的 `.flatMap` / `.filter` 在某处抛 `Cannot read properties of null`，不如在入口把 `null` / `undefined` 统一成 `[]`（v0.80.4 补入）。这是"在边界处把非法输入收敛成合法输入"的典型防御 — 让下游代码可以无条件信任类型契约。

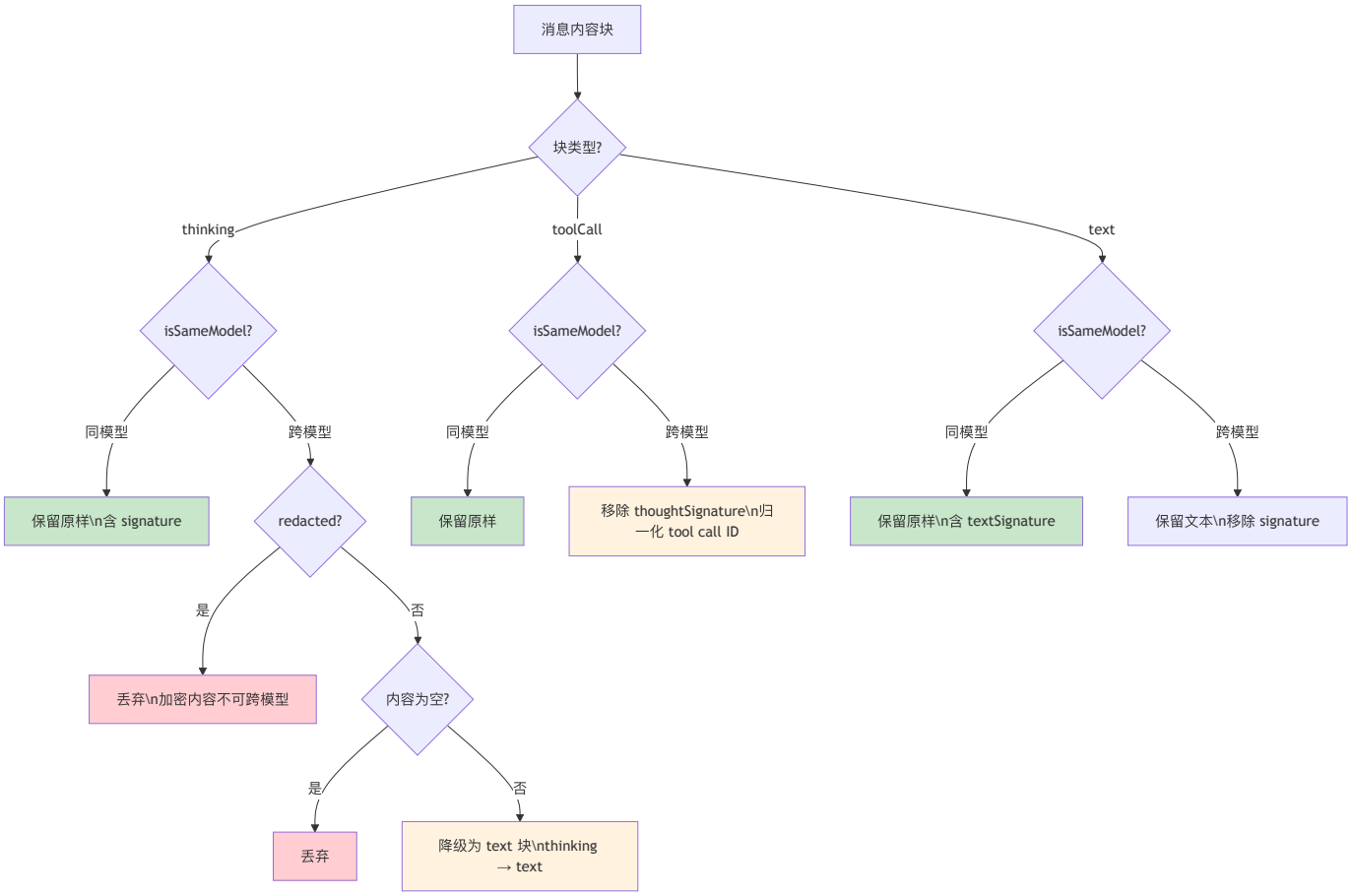
变换策略：同模型保持，跨模型降级

`transformMessages` 的核心判断逻辑围绕一个布尔值 `isSameModel`：

```
// packages/ai/src/api/transform-messages.ts:95-98

const isSameModel =
  assistantMsg.provider === model.provider &&
  assistantMsg.api === model.api &&
  assistantMsg.model === model.id;
```

这不是简单的"同 provider"判断 — 它要求 provider、api、model ID 三者完全一致。同一个 provider 的不同模型（比如 `claude-sonnet-4-6` 和 `claude-opus-4-6`）也被视为"不同模型"。

基于这个判断，变换策略如下：



Thinking Block 变换：完整的决策树

Thinking block 是整个变换逻辑中最复杂的部分。这不是因为代码多，而是因为 thinking 块有多种形态，每种的处理策略不同。先看类型定义：

```
// packages/ai/src/types.ts:335-343

export interface ThinkingContent {
  type: "thinking";
  thinking: string;
  thinkingSignature?: string;
  /** When true, the thinking content was redacted by safety filters.
   *  The opaque encrypted payload is stored in `thinkingSignature`
   *  so it can be passed back to the API for multi-turn continuity. */
  redacted?: boolean;
}
```

一个 `ThinkingContent` 可以是以下几种情况：

1. **正常的思维内容** — `thinking` 有文本，没有 `redacted`，可能有 `thinkingSignature`
2. **被安全过滤的思维** — `redacted === true`，`thinkingSignature` 存储加密后的不透明载荷
3. **OpenAI 加密推理** — `thinking` 为空，但 `thinkingSignature` 存在（OpenAI 的 reasoning item ID）
4. **空思维块** — `thinking` 为空或纯空白，没有 signature

每种情况的处理逻辑完全不同。以下是 `transformMessages` 中的完整决策代码：

```
// packages/ai/src/api/transform-messages.ts:100-117

const transformedContent = assistantMsg.content.flatMap((block) => {
  if (block.type === "thinking") {
    // Redacted thinking is opaque encrypted content,
    // only valid for the same model.
    // Drop it for cross-model to avoid API errors.
    if (block.redacted) {
      return isSameModel ? block : [];
    }
    // For same model: keep thinking blocks with signatures
    // (needed for replay) even if the thinking text is empty
    // (OpenAI encrypted reasoning)
    if (isSameModel && block.thinkingSignature) return block;
    // Skip empty thinking blocks, convert others to plain text
    if (!block.thinking || block.thinking.trim() === "") return [];
    if (isSameModel) return block;
    return {
      type: "text" as const,
      text: block.thinking,
    };
  }
```

逐行拆解这个决策树：

第一层判断：`block.redacted`。Redacted thinking 是安全过滤的产物。当 Anthropic 的安全系统认为某段思维内容不适合展示时，会将其替换为加密载荷，存储在 `thinkingSignature` 中。这个加密载荷只有**同一个模型**能解读 — 它需要在后续的 API 调用中原样回传，以维持多轮对话的连续性。跨模型时，这段加密内容对目标模型来说就是乱码，传过去只会导致 API 报错，所以直接丢弃（`return []`）。

第二层判断：`isSameModel && block.thinkingSignature`。这是专门处理 OpenAI 加密推理（encrypted reasoning）的分支。OpenAI 的 reasoning model（如 o1、o3）不

会暴露推理文本，但会返回一个 reasoning item ID 作为 `thinkingSignature`。此时 `thinking` 字段为空字符串，但 `thinkingSignature` 存在。如果是同模型重放，这个 signature 必须保留 — 模型需要它来延续推理上下文。关键点在于：这个分支在 redacted 检查**之后**，所以它不会误处理 redacted blocks。

第三层判断：空内容检查。如果 `thinking` 为空或纯空白，且不是上面两种有 signature 的情况，那这个块就没有任何有用信息，直接丢弃。

第四层：同模型保留，跨模型降级。如果有实际的思维文本，同模型原样保留（包括 signature），跨模型则降级为普通 `text` 块 — 文本内容保留，但失去了"这是模型的内部推理"这层语义信息。

这个决策树的顺序很关键。把 redacted 检查放在最前面是防御性编程的体现：redacted 块的处理规则最严格（跨模型必须丢弃），如果漏掉了，可能导致加密载荷被当作普通文本传给目标模型。

Text Block 变换：看似简单的清洗

Text block 是最"普通"的内容类型，但即使是文本块，跨模型时也需要变换：

```
// packages/ai/src/api/transform-messages.ts:119-125

if (block.type === "text") {
  if (isSameModel) return block;
  return {
    type: "text" as const,
    text: block.text,
  };
}
```

同模型时原样返回，跨模型时构造一个新的 `TextContent` 对象，只保留 `type` 和 `text` 两个字段。为什么不能直接 `return block`？因为 `TextContent` 类型上还有一个可选字段：

```
// packages/ai/src/types.ts:329-333

export interface TextContent {
  type: "text";
  text: string;
  textSignature?: string;
}
```

`textSignature` 是 OpenAI Responses API 附加的元数据 — 可能是 legacy ID 字符串，也可能是 `TextSignatureV1` JSON（包含版本号、ID 和 phase 信息）。同模型时保留这些元数据有助于 API 重放的准确性；跨模型时，这些 provider 特有的元数据对目标模型毫无意义，甚至可能引起兼容性问题。

通过构造一个新对象而非修改原对象，代码保证了跨模型时 `textSignature` 被干净地剥离。这是一种典型的"白名单"策略：不是"检查并删除已知的无关字段"，而是"只复制已知需要的字段"。白名单策略更安全 — 如果未来 `TextContent` 增加了新的 provider 特有字段，白名单策略会自动将其排除在跨模型变换之外，无需修改变换代码。

Tool Call ID 归一化：一个具体的例子

OpenAI Responses API 生成的 tool call ID 长这样：

```
fc_682e1b1b5c9081919ecae4e2b4f73f710cf7bd7c89b44df5|call_RJxMmhTWpikOz4UMgkJbopvl
```

450+ 字符，包含 `|` 字符。如果把这个 ID 原样传给 Anthropic，API 会拒绝 — Anthropic 要求 `^[a-zA-Z0-9_-]+$`，最多 64 字符。

`transformMessages` 通过 `normalizeToolCallId` 回调解决这个问题：

```
// packages/ai/src/api/transform-messages.ts:136-142

if (!isSameModel && normalizeToolCallId) {
  const normalizedId = normalizeToolCallId(
    toolCall.id, model, assistantMsg
  );
  if (normalizedId !== toolCall.id) {
    toolCallIdMap.set(toolCall.id, normalizedId);
    normalizedToolCall = { ...normalizedToolCall, id: normalizedId };
  }
}
```

注意 `toolCallIdMap` 的设计：当一个 tool call ID 被归一化后，映射关系被存储起来。后续遇到对应的 `toolResult` 消息时，它的 `toolCallId` 也会被同步更新：

```
// packages/ai/src/api/transform-messages.ts:84-90

if (msg.role === "toolResult") {
  const normalizedId = toolCallIdMap.get(msg.toolCallId);
  if (normalizedId && normalizedId !== msg.toolCallId) {
    return { ...msg, toolCallId: normalizedId };
  }
}
```

tool call 和 tool result 的 ID 必须匹配，否则 API 会报错。归一化必须双向一致。

同样值得注意的是 `thoughtSignature` 的处理（源码 `:131-134`）：Google 的 tool call 携带 `thoughtSignature` 用于思维链上下文复用，跨模型时这个字段被删除。这和 text block 的白名单策略不同 — tool call 由于有 `id`、`name`、`arguments` 等关键字段需要精确保留，这里用的是"黑名单"策略：显式删除已知的无关字段。

第二遍扫描：合成缺失的 Tool Result

`transformMessages` 做了两遍扫描。第一遍处理内容变换（thinking 降级、ID 归一化、text signature 清洗）。第二遍处理一个更隐蔽的问题：**孤立的 tool call**。

孤立 tool call 是怎么产生的?

当 assistant 消息中有 tool call，但对应的 tool result 缺失时，API 会报错。这种"孤立"有几种成因：

1. **用户中途 abort 了 agent 循环** — assistant 发出了 tool call，但 tool 还没执行用户就按了 Ctrl+C
2. **tool 执行过程中发生了错误** — result 消息没有被正确记录
3. **用户在 tool call 和 tool result 之间切换了模型** — 新模型看到了前模型的 tool call，但没有对应的 result

合成逻辑的完整代码

第二遍扫描的核心是一个状态机，追踪"当前有哪些待回复的 tool call"。合成逻辑被抽成一个闭包 `insertSyntheticToolResults`，在三个位置调用：

```
// packages/ai/src/api/transform-messages.ts:160-187（节选）

const result: Message[] = [];
let pendingToolCalls: ToolCall[] = [];
let existingToolResultIds = new Set<string>();
const insertSyntheticToolResults = () => {
  if (pendingToolCalls.length > 0) {
    for (const tc of pendingToolCalls) {
      if (!existingToolResultIds.has(tc.id)) {
        result.push({
          role: "toolResult",
          toolCallId: tc.id,
          toolName: tc.name,
          content: [{ type: "text", text: "No result provided" }],
          isError: true,
          timestamp: Date.now(),
        } as ToolResultMessage);
      }
    }
    pendingToolCalls = [];
    existingToolResultIds = new Set();
  }
};

for (let i = 0; i < transformed.length; i++) {
  const msg = transformed[i];
  if (msg.role === "assistant") {
    // 前一条 assistant 若有未回复的 tool call，先补合成 result
    insertSyntheticToolResults();
```

注意这里的时序：当遇到一条新的 assistant 消息时，如果前一条 assistant 还有未回复的 tool call，在新 assistant **之前**插入合成的 tool result。这保证了消息序列始终满足 `assistant(tool_call) → toolResult → assistant` 的交替模式。把合成逻辑抽成一个函数、在三处（遇到新 assistant、遇到 user 消息、以及转录收尾）统一调用，是 v0.80.0 之后的一次可读性重构 — 三种"孤立 tool call"场景共用同一段补齐代码。

错误/中止消息的跳过

紧接着合成逻辑之后，是对 error 和 aborted 消息的处理：

```
// packages/ai/src/api/transform-messages.ts:189-197

// Skip errored/aborted assistant messages entirely.
// These are incomplete turns that shouldn't be replayed:
// - May have partial content (reasoning without message,
//   incomplete tool calls)
// - Replaying them can cause API errors (e.g., OpenAI
//   "reasoning without following item")
// - The model should retry from the last valid state
const assistantMsg = msg as AssistantMessage;
if (assistantMsg.stopReason === "error"
  || assistantMsg.stopReason === "aborted") {
  continue;
}
```

被 `continue` 跳过的消息不会出现在最终结果中。源码注释精确地解释了原因：这些消息可能包含不完整的内容 — 比如 OpenAI 模型可能返回了 reasoning 但还没来得及生成后续内容就中断了，重放这样的消息会触发 "reasoning without following item" 错误。

用户消息打断 Tool 流

第二遍扫描还处理一种特殊场景：**用户消息打断了 tool 流**。正常的 agent 循环是 `assistant(tool_call) → toolResult → assistant`，但用户可以在任何时候发送新消息。如果用户在 assistant 发出 tool call 后、tool result 返回前发送了新消息，tool call 就变成了孤立的：

```
// packages/ai/src/api/transform-messages.ts:210-213

} else if (msg.role === "user") {
  // User message interrupts tool flow - insert synthetic
  // results for orphaned calls
  insertSyntheticToolResults();
  result.push(msg);
}
```

因为合成逻辑已经抽成 `insertSyntheticToolResults`，user 分支只需复用同一个闭包 — 处理策略是相同的：在用户消息**之前**插入合成的 tool result，修复断裂的消息序列。闭包内 `existingToolResultIds` 的检查保证了如果部分 tool call 已经有了真实的 result（比如 assistant 发了 3 个 tool call，2 个已经有 result，用户在第 3 个执行完之前发了消息），只为缺失的那些补充合成 result。

合成的 tool result 都标记为 `isError: true`，内容为 `"No result provided"`。这个设计有双重目的：一是满足 API 的格式要求（每个 tool call 必须有对应的 result），二是给模型一个信号 — 这个工具调用的结果是不可靠的，模型应该考虑重新调用或采取其他策略。

收尾分支：以未回复 tool call 结尾的转录

前面两种合成都发生在"中间"——孤立的 tool call 后面还跟着新的 assistant 或 user 消息。v0.69.0 起还补上了第三种情形：整段转录**以一条尚未回复的 assistant tool call 结尾**（既没有后续 result，也没有后续消息触发合成）。这在直接重放低层历史（history replay）时会出现——例如恢复一个在工具执行途中被中断的会话。此时 `transformMessages` 会在序列**末尾**追加合成的 tool result，确保即便是最后一条消息也满足 `assistant(tool_call) → toolResult` 的闭合要求，否则下一次 provider 调用会因末尾悬空的 tool call 而报错。

具体例子：从 Claude 到 GPT 的消息变换

以下是一个 3 消息对话在跨模型变换前后的对比。假设用户在 Claude（`claude-sonnet-4-6`）上进行了对话，现在要切换到 GPT（`gpt-4o`）。

变换前（Claude 原生消息）：

```
[
  { "role": "user", "content": "查看 src/main.rs 的内容" },
  {
    "role": "assistant",
    "provider": "anthropic", "api": "anthropic-messages",
    "model": "claude-sonnet-4-6",
    "content": [
      { "type": "thinking",
        "thinking": "用户要看文件内容，我用 read 工具",
        "thinkingSignature": "sig_abc123..." },
      { "type": "text",
        "text": "我来读取文件内容。",
        "textSignature": "
{\"v\":1,\"id\":\"msg_01X...\",\"phase\":\"commentary\"}" },
      { "type": "toolCall",
        "id": "toolu_01ABC", "name": "read",
        "arguments": { "path": "src/main.rs" } }
    ],
    "stopReason": "toolUse"
  },
  {
    "role": "toolResult",
    "toolCallId": "toolu_01ABC",
    "toolName": "read",
    "content": [{ "type": "text", "text": "fn main() { ... }" }],
    "isError": false
  }
]
```

变换后（发送给 GPT 的消息）：

```
[
  { "role": "user", "content": "查看 src/main.rs 的内容" },
  {
    "role": "assistant",
    "provider": "anthropic", "api": "anthropic-messages",
    "model": "claude-sonnet-4-6",
    "content": [
      { "type": "text",
        "text": "用户要看文件内容，我用 read 工具" },
      { "type": "text",
        "text": "我来读取文件内容。" },
      { "type": "toolCall",
        "id": "toolu_01ABC", "name": "read",
        "arguments": { "path": "src/main.rs" } }
    ]
  },
  {
    "role": "toolResult",
    "toolCallId": "toolu_01ABC",
    "toolName": "read",
    "content": [{ "type": "text", "text": "fn main() { ... }" }],
    "isError": false
  }
]
```

变换产生了以下变化：

内容	变换前	变换后	说明
Thinking block	`type: "thinking"` + signature	`type: "text"`	降级为普通文本，signature 丢失
Text block	含 `textSignature`	无 signature	文本保留，元数据剥离
Tool call	原样	原样	Claude 的 ID 格式恰好符合大多数 provider 的要求
Tool result	原样	原样	ID 未变，无需更新
User message	原样	原样	用户消息从不变换

丢失了什么?

- thinking 块从结构化思维降级为普通文本。GPT 不知道这段文字是前一个模型的内部推理 — 它看到的只是一段额外的 text block。这意味着 GPT 不会用自己的 reasoning 能力来"接着想"，而是把这段文字当作 assistant 说过的话来理解。
- `textSignature` 被剥离。如果后续再切回 Claude，这个 signature 已经不可恢复。
- `thinkingSignature` 被丢弃。Claude 的 thinking 连续性在切换到 GPT 的那一刻就中断了。

保留了什么?

- 所有的文本内容 — 思维内容虽然降级了，但文字本身没丢
- 完整的 tool call / tool result 对 — GPT 可以看到前模型调用了什么工具、得到了什么结果
- 对话的因果链 — 用户问了什么、模型做了什么、结果是什么，这条语义链完整保留

这就是"有损但安全"的核心含义：丢失的是 provider 特有的元数据和语义标注，保留的是对话的内容和因果关系。

取舍分析

得到了什么

1. 用户可以随时切换模型。从 Claude 切到 GPT 再切回 Gemini，历史消息不会丢失（虽然会降级）。这在实际使用中非常重要 — 用户可能因为模型性能、成本、上下文窗口等原因频繁切换。

2. 变换是确定性的。同样的输入总是产生同样的输出。没有随机性，没有网络调用，只是纯粹的数据变换。

3. 绝不让变换导致 API 失败。合成 tool result、ID 归一化、跳过错误消息 — 每个策略都是为了保证变换后的消息可以被目标 provider 接受。

放弃了什么

1. 变换是有损的。thinking 块从结构化思维变成了普通文本，丢失了模型特有的语义。redacted thinking 在跨模型时被完全丢弃。这些信息一旦丢失就无法恢复。

2. 合成的 tool result 是假数据。"No result provided" 这个合成结果告诉模型"这个工具调用没有结果"，但模型可能会基于这个假结果做出不理想的推断。不过 `isError: true` 标记在一定程度上缓解了这个问题 — 模型通常会把错误的 tool result 当作需要重试的信号。

3. `isSameModel` 的判断过于严格。同 provider 的不同模型（比如 Claude Sonnet 和 Claude Opus）也被视为"不同模型"，thinking 块会被降级。

`isSameModel` 的严格性：一个深思熟虑的保守选择

`isSameModel` 要求 `provider` 、 `api` 、 `model` 三者完全一致。这意味着以下场景都被视为"不同模型"：

- **同 provider 不同模型**： `claude-sonnet-4-6` → `claude-opus-4-6` （Anthropic 内部切换）
- **同 provider 不同 API**： `gpt-4o` via Chat Completions → `gpt-4o` via Responses API
- **同模型不同 provider**：通过 Anthropic 直连的 Claude → 通过 AWS Bedrock 的 Claude

为什么不放宽为"同 provider"就保留？因为 thinking signature 的兼容性是模型级别的，不是 provider 级别的。Anthropic 没有承诺 Sonnet 的 thinking signature 可以被 Opus 正确解读。OpenAI 也没有承诺不同模型之间的 reasoning item ID 可以互换。实际上，即使是同一个模型的不同版本（比如 `claude-sonnet-4-20250514` 和未来的 `claude-sonnet-4-20250801` ）是否共享 thinking signature 格式，也是未知的。

`api` 字段的检查更加微妙。同一个 provider 可能通过不同的 API 暴露同一个模型 — 比如 OpenAI 的 Chat Completions API 和 Responses API。虽然底层是同一个模型，但两种 API 返回的元数据格式不同（text signature 的结构、reasoning item 的编码方式等）。如果只检查 `provider + model` 而忽略 `api` ，可能会把 Responses API 的 signature 传给 Chat Completions API，导致不可预知的错误。

这种"宁可多降级一次，也不冒 API 报错的风险"策略，本质上是在**可用性和保真度之间选择了可用性**。降级只是丢失一些元数据，用户可能完全感受不到；而 API 报错会直接中断对话，用户体验断裂。在生产系统中，这个取舍几乎总是正确的。

白名单 vs 黑名单的一致性问题

值得注意的是，变换代码对不同块类型使用了不同的"清洗"策略：

- **Text blocks**：白名单 — 构造新对象，只包含 `type` 和 `text`
- **Tool calls**：黑名单 — 在原对象上显式删除 `thoughtSignature`

这种不一致是有原因的：text block 的字段少且稳定（`type`、`text`、`textSignature`），白名单实现简单且安全。Tool call 的字段多且关键（`id`、`name`、`arguments` 都不能丢），白名单实现需要枚举所有需要保留的字段，增加了维护负担和遗漏风险。但这种不一致也带来了未来的风险 — 如果 `ToolCall` 类型增加了新的 provider 特有字段，黑名单策略需要记得更新变换代码。

核心判断：**有损交接好过不能交接。** 丢失一些 thinking 细节，比"切换模型后对话完全中断"要好得多。

第 8 章将展示循环引擎如何在每次 LLM 调用前把 `AgentMessage[]` 收敛成 ai 层 `Message[]`。真正的 `transformMessages` 则发生在各 provider 构造请求时，用来把这组统一消息进一步变成目标 API 可接受的格式。

版本演化说明

> 本章内容已对照 pi-mono **v0.82.1**。本章的变换策略整体未变，但有两处需要订正：
>
> - **路径迁移（v0.80.0）**：`transformMessages` 随 API 实现从 `providers/transform-messages.ts` 迁到 `api/transform-messages.ts`；本章行号已全部按当前源码更新，`transformMessages` 现位于 `:64`。
> - **null content 归一化（v0.80.4）**：入口新增 `content == null → []` 的防御（`:71-73`），见本章第二节。其余演进（redacted thinking 处理、thoughtSignature 清理、合成 tool result、v0.69.0 的收尾合成分支）都是在遇到实际 API 错误后逐步添加的防御措施；第二遍扫描已重构为共用 `insertSyntheticToolResults` 闭包。注意：跨模型示例中的 `api` 字段值是 `"anthropic-messages"`（与 `KnownApi` 一致），早期草稿误写为 `"messages"`。

第 6 章：统一事件流设计

> **定位**：本章解析 pi-ai 的流式事件契约 — 整个系统的"最底层脉搏"。 前置依赖：第 4 章（Provider Registry）。 适用场景：当你想理解为什么 pi 选择流式事件而非一次性响应，或者想为自己的系统设计流式 API。

为什么不返回一个 Promise，而是返回一个事件流?

这是本章的核心设计问题。

最简单的 LLM API 设计是 `async function complete(prompt): Promise<string>` 。调用、等待、拿结果。但 pi 从最底层就选择了流式设计 — 即使是不需要流式渲染的场景（比如后台任务），底层 API 仍然返回事件流。

这个选择的原因不是"流式渲染更快"（虽然确实如此），而是**事件流是唯一能完整捕获 LLM 交互过程的数据模型**。一个 `Promise<string>` 能告诉你结果，但不能告诉你过程中发生了什么 — thinking 阶段用了多长时间、模型输出的每个 token 的时间分布、tool call 是在什么位置开始出现的。事件流把这些过程信息保留了下来。

`StreamFunction` 的契约

pi-ai 层最重要的一段注释在 `types.ts` 的 `StreamFunction` 类型定义上：

```
// packages/ai/src/types.ts:303-315

// Contract:
// - Must return an AssistantMessageEventStream.
// - Once invoked, request/model/runtime failures should be encoded in
the
//   returned stream, not thrown.
// - Error termination must produce an AssistantMessage with stopReason
//   "error" or "aborted" and errorMessage, emitted via the stream
protocol.
export type StreamFunction<TApi extends Api, TOptions extends
StreamOptions> = (
  model: Model<TApi>,
  context: Context,
  options?: TOptions,
) => AssistantMessageEventStream;
```

这三条规则定义了整个系统的错误处理哲学：

规则 1：必须返回事件流。不是 Promise，不是回调，是 `AssistantMessageEventStream`。调用者总是拿到一个流对象，然后 `for await` 消费事件。

规则 2：一旦调用，错误编码进流里。`StreamFunction` 本身不抛异常。网络超时、API 限流、模型不存在 — 所有失败都通过流中的事件传递。这意味着调用者不需要 try-catch。

规则 3：错误终止必须产出完整的 `AssistantMessage`。即使请求失败了，流也必须产出一个带 `stopReason: "error"` 和 `errorMessage` 的 `AssistantMessage`。调用者总是可以调 `stream.result()` 拿到一个消息对象 — 成功的或失败的。

EventStream：发布-消费的桥梁

`AssistantMessageEventStream` 的底层实现是一个通用的 `EventStream<T, R>` 类。这个类只有 66 行，却是整个流式架构的基石。我们完整地展示它，然后逐段分析。

完整实现

```
// packages/ai/src/utils/event-stream.ts:4-48（节选）

export class EventStream<T, R = T> implements AsyncIterable<T> {
  private queue: T[] = [];
  private waiting: ((value: IteratorResult<T>) => void)[] = [];
  private done = false;
  private finalResultPromise: Promise<R>;
  private resolveFinalResult!: (result: R) => void;

  constructor(private isComplete: (event: T) => boolean,
              private extractResult: (event: T) => R) {
    this.finalResultPromise = new Promise((r) => {
this.resolveFinalResult = r; });
  }

  push(event: T): void {
    if (this.done) return;
    if (this.isComplete(event)) {
      this.done = true;
      this.resolveFinalResult(this.extractResult(event));
    }
    const waiter = this.waiting.shift();
    if (waiter) waiter({ value: event, done: false });
    else this.queue.push(event);
  }

  end(result?: R): void {
    this.done = true;
    if (result !== undefined) this.resolveFinalResult(result);
    while (this.waiting.length > 0) {
      this.waiting.shift()!({ value: undefined as any, done: true });
    }
  }
```

```
// packages/ai/src/utils/event-stream.ts:50-66

  async *[Symbol.asyncIterator](): AsyncIterator<T> {
    while (true) {
      if (this.queue.length > 0) {
        yield this.queue.shift()!;
      } else if (this.done) {
        return;
      } else {
        const result = await new Promise<IteratorResult<T>>(
          (resolve) => this.waiting.push(resolve)
        );
        if (result.done) return;
        yield result.value;
      }
    }
  }

  result(): Promise<R> {
    return this.finalResultPromise;
  }
}
```

Queue/Waiting Consumer 模式

`EventStream` 的核心是一对互补的数组：`queue` 和 `waiting`。这个设计实现了生产者和消费者之间的无锁协调：

- `queue: T[]` — 事件缓冲区。当事件到达但没有消费者在等待时，事件排队等候。
- `waiting: ((value: IteratorResult<T>) => void)[]` — 消费者等待队列。当消费者需要下一个事件但队列为空时，消费者把自己的 resolve 函数注册到这里。

这两个数组**互斥**使用：在任何时刻，要么 `queue` 里有积压的事件（消费者慢于生产者），要么 `waiting` 里有挂起的消费者（消费者快于生产者），要么两个都为空（恰好平衡）。不可能同时两个数组都有内容 — 如果有积压的事件，新来的消费者会立即拿走一个；如果有等待的消费者，新来的事件会立即送达。

这个模式本质上是一个**无界异步通道**（unbounded async channel），但用不到 50 行代码实现了。没有引入任何第三方依赖，没有用 `EventEmitter`，没有用 `ReadableStream` — 就是两个数组和 Promise 的组合。

push() 的工作流

```
push(event) 被调用
  │
  ├── done 为 true? → 静默丢弃，直接返回
  │
  ├── isComplete(event) 为 true?
  │     → 标记 done = true
  │     → 用 extractResult(event) resolve finalResultPromise
  │
  └── 尝试投递事件
        ├── waiting 中有消费者? → 取出第一个 waiter，直接投递
        └── 没有消费者? → 推入 queue 缓冲
```

注意 `push()` 的一个关键设计：**即使事件触发了完成（`isComplete` 返回 true），该事件仍然会被投递给消费者**。设置 `done = true` 和 resolve `finalResultPromise` 发生在投递之前，但投递本身不会被跳过。这意味着消费者通过 `for await` 迭代时，一定能收到 `done` 或 `error` 这个终止事件本身 — 它不会被"吞掉"。

end() 的工作流

`end()` 是为异常情况准备的"紧急关闭"方法：

```
end(result?) 被调用
  │
  ├── 标记 done = true
  │
  ├── result 不为 undefined? → resolve finalResultPromise
  │
  └── 通知所有等待的消费者：发送 { done: true }
        → 每个 waiter 收到后，asyncIterator 中的循环 return
```

`end()` 和 `push()` 触发完成的区别在于：`push()` 是通过一个"完成事件"自然结束（流的正常终止路径），而 `end()` 是外部强制关闭流（比如 provider 代码捕获到异常后需要清理）。两者都会 resolve `finalResultPromise`，但 `end()` 不投递任何事件 — 它只是告诉所有等待中的消费者"没有更多数据了"。

result() 和 finalResultPromise

`result()` 方法返回 `finalResultPromise` — 一个在构造时就创建好的 Promise。这个 Promise 在两种情况下被 resolve：

1. `push()` 收到一个使 `isComplete()` 返回 true 的事件时，用 `extractResult(event)` 的返回值 resolve。
2. `end(result)` 被显式传入 result 时，用这个 result 直接 resolve。

`result()` 的存在使得**流式消费和一次性消费使用同一个底层机制**。如果你需要流式渲染， `for await (const event of stream)` 逐个处理事件。如果你只需要最终结果， `await stream.result()` 直接等待。这就是 `completeSimple` 的实现方式：

```
// 非流式用法：直接等最终结果
const message = await stream(model, context, options).result();
```

一行代码，把流式 API 变成了同步 API。消费者不需要知道底层是不是流式的 — 接口是统一的。

asyncIterator 的三态循环

`[Symbol.asyncIterator]()` 是一个 async generator，它的 `while(true)` 循环在每次迭代中检查三种状态：

1. **`queue` 非空** — 直接 shift 一个事件 yield 出去。这是"追赶"模式：生产者曾经比消费者快，积累了缓冲，消费者现在快速消耗。
2. **`done` 为 true 且 `queue` 为空** — 流结束，return。不再产出任何值。
3. **`queue` 为空且 `done` 为 false** — 消费者比生产者快，没有事件可消费。创建一个 Promise 并把 resolve 函数推入 `waiting` 。消费者挂起在这个 Promise 上，直到 `push()` 或 `end()` 来唤醒它。

这三态检查的顺序很重要：先检查 queue，再检查 done，最后挂起等待。这保证了即使流已经结束（ `done` 为 true），消费者也会先把 queue 中的剩余事件消费完。

AssistantMessageEventStream：具体化的 LLM 响应流

通用的 `EventStream<T, R>` 需要被具体化为 LLM 响应场景。这就是 `AssistantMessageEventStream` 的工作：

```
// packages/ai/src/utils/event-stream.ts:69-83

export class AssistantMessageEventStream
  extends EventStream<AssistantMessageEvent, AssistantMessage> {
  constructor() {
    super(
      (event) => event.type === "done" || event.type === "error",
      (event) => {
        if (event.type === "done") {
          return event.message;
        } else if (event.type === "error") {
          return event.error;
        }
        throw new Error("Unexpected event type for final result");
      },
    );
  }
}
```

它做了两件事：

1. **定义完成条件**：`isComplete` 检查事件是否为 `done` 或 `error` 类型。只有这两种事件标志着流的终止。
2. **定义结果提取**：`extractResult` 从 `done` 事件取 `event.message`，从 `error` 事件取 `event.error`。两者都是 `AssistantMessage` 类型 — 成功和失败返回的是同一种数据结构，只是 `stopReason` 字段不同。

事件类型：AssistantMessageEvent 联合类型

`AssistantMessageEvent` 是一个 discriminated union，通过 `type` 字段区分 12 种事件。我们按功能分组来看：

生命周期事件

```
// packages/ai/src/types.ts:492
| { type: "start"; partial: AssistantMessage }
```

start — 流的第一个事件。provider 在开始接收 API 响应后立即发射。携带一个初始的 partial AssistantMessage，此时 content 数组通常为空，但 model 、 provider 、 api 等元数据已经填充。消费者可以用这个事件来显示"正在生成..."的状态。

文本内容事件

```
// packages/ai/src/types.ts:493-495
| { type: "text_start"; contentIndex: number; partial: AssistantMessage
}
| { type: "text_delta"; contentIndex: number; delta: string; partial:
AssistantMessage }
| { type: "text_end"; contentIndex: number; content: string; partial:
AssistantMessage }
```

text_start — 一段文本内容开始。 contentIndex 指向 AssistantMessage.content 数组中的位置。

text_delta — 文本增量。 delta 是新增的文本片段（通常是一个或几个 token）。这是流式渲染的核心事件 — TUI 收到 text_delta 后立即追加显示。

text_end — 一段文本结束。 content 是这段文本的完整内容（所有 delta 的拼接结果）。消费者可以用它来做最终校验，而不需要自己累积 delta。

Thinking 事件

```
// packages/ai/src/types.ts:496-498
| { type: "thinking_start"; contentIndex: number; partial:
AssistantMessage }
| { type: "thinking_delta"; contentIndex: number; delta: string;
partial: AssistantMessage }
| { type: "thinking_end"; contentIndex: number; content: string;
partial: AssistantMessage }
```

thinking_start/delta/end — 与文本事件结构完全相同，但语义不同。这些事件对应模型的 extended thinking 输出（如 Claude 的 thinking blocks）。消费者可以选择性地显示或隐藏 thinking 内容。版本演化说明：这组事件是后来添加的，最初的设计只有 text 和 toolcall。

Tool Call 事件

```
// packages/ai/src/types.ts:499-501
| { type: "toolcall_start"; contentIndex: number; partial:
AssistantMessage }
| { type: "toolcall_delta"; contentIndex: number; delta: string;
partial: AssistantMessage }
| { type: "toolcall_end"; contentIndex: number; toolCall: ToolCall;
partial: AssistantMessage }
```

toolcall_start — 模型开始生成一个 tool call。此时工具名称可能已经确定，但参数还在流式生成中。

toolcall_delta — 工具调用参数的增量。`delta` 是 JSON 参数字符串的一部分。与 text_delta 不同，tool call 的 delta 通常是不可直接渲染的 JSON 片段。

toolcall_end — 工具调用完成。`toolCall` 字段携带完整的 `ToolCall` 对象，包含 `id`、`name`、`arguments`。这是循环引擎（第 8 章）真正需要的事件 — 拿到完整的 tool call 后，引擎可以开始执行工具。

终止事件

```
// packages/ai/src/types.ts:502-503
| { type: "done"; reason: Extract<StopReason, "stop" | "length" |
"toolUse">; message: AssistantMessage }
| { type: "error"; reason: Extract<StopReason, "aborted" | "error">;
error: AssistantMessage }
```

终止事件是整个事件协议中最关键的部分。它分为两种：

done — 正常终止。`reason` 告诉消费者**为什么**流结束了：

- `"stop"` — 模型自然结束输出。最常见的情况，表示模型认为回答已经完整。

- `"length"` — 达到 max tokens 限制，输出被截断。这不是错误，但消费者应该知道回答可能不完整。
- `"toolUse"` — 模型决定调用工具。注意这里是 `"toolUse"` 而不是 `"tool_use"` — pi 使用驼峰命名作为内部统一值，而不是照搬某个具体 provider 的命名。

`error` — 错误终止。`reason` 区分两种错误：

- `"error"` — 请求失败。网络超时、API 限流、模型返回错误响应等。
- `"aborted"` — 用户主动中止。通常是 AbortSignal 触发的取消操作。

两种终止事件都携带完整的 `AssistantMessage` 对象（`done.message` 和 `error.error`）。注意类型定义使用了 TypeScript 的 `Extract` 工具类型来约束 reason 的可选值 — `done` 只能携带 `"stop" | "length" | "toolUse"`，`error` 只能携带 `"error" | "aborted"`。这是编译时保证，不是运行时检查。

contentIndex 和 partial 的设计意图

每个中间事件（除了 `start`、`done`、`error`）都携带两个公共字段：

- **`contentIndex: number`** — 指向 `AssistantMessage.content` 数组中的位置。一次 LLM 响应可能包含多个内容块（先 thinking，再 text，再 tool call），`contentIndex` 标识当前事件属于哪个块。
- **`partial: AssistantMessage`** — 到当前事件为止的累积状态。这个 partial 对象随着事件推进不断更新。消费者不需要自己维护状态 — 任何时候拿最新的 `partial` 就是当前的完整快照。

`partial` 的设计是一个有意的冗余。它增加了每个事件的数据量，但换来了消费者实现的极大简化。一个只关心最终文本的消费者，可以在收到任何事件时直接读取 `partial.content`，而不需要自己累积 delta。

AssistantMessage：最终结果类型

无论流式消费还是一次性消费，最终拿到的都是 `AssistantMessage`：

```ts
// packages/ai/src/types.ts:390-403

export interface AssistantMessage {
  role: "assistant";
  content: (TextContent | ThinkingContent | ToolCall)[];
  api: Api;
  provider: ProviderId;
  model: string;
  responseModel?: string;    // 路由后的实际模型（如 OpenRouter auto）
  responseId?: string;
  diagnostics?: AssistantMessageDiagnostic[]; // 脱敏的失败/恢复诊断
  usage: Usage;
  stopReason: StopReason;
  errorMessage?: string;
  timestamp: number; // Unix timestamp in milliseconds
}
```

逐字段分析：

- **`role: "assistant"`** — 固定值。与 `UserMessage`（`role: "user"`）和 `ToolResultMessage`（`role: "toolResult"`）一起构成三种消息类型，用于对话历史的类型区分。
- **`content: (TextContent | ThinkingContent | ToolCall)[]`** — 内容数组。一次响应可以包含多种类型的内容块：纯文本（`TextContent`）、思考过程（`ThinkingContent`）、工具调用（`ToolCall`）。数组的顺序对应模型输出的顺序。
- **`api: Api`** — 使用的 API 类型，如 `"anthropic-messages"` 或 `"openai-responses"`。
- **`provider: ProviderId`** — 使用的 provider 标识，如 `"anthropic"` 或 `"openai"`。类型是 `ProviderId`（`KnownProvider | string`），v0.80.0 起从旧名 `Provider` 改来 — 现在 `Provider` 指运行时 provider 接口（第 4 章）。
- **`model: string`** — 实际使用的模型名称，如 `"claude-sonnet-4-20250514"`。
- **`responseId?: string`** — 上游 API 返回的响应标识符（可选）。不同 provider 的含义不同。
- **`responseModel?: string`** — 当上游路由后的实际模型与请求的 `model` 不同时填充（如 OpenRouter 的 `auto` 路由到 `anthropic/...`，v0.71.0 引入）。`model` 是"请求了什么"，`responseModel` 是"实际跑了什么"。
- **`diagnostics?: AssistantMessageDiagnostic[]`** — 脱敏后的 provider/runtime 诊断，记录这次响应过程中的失败与恢复（如重试、降级），用于可观测性而不泄露敏感请求内容（v0.59.0 起）。

- **`usage: Usage`** — token 使用统计，包含 `input`、`output`、`cacheRead`、`cacheWrite`、`totalTokens`，以及对应的 `cost` 计算。
- **`stopReason: StopReason`** — 停止原因。类型为 `"stop" | "length" | "toolUse" | "error" | "aborted"`，与终止事件的 reason 对应。
- **`errorMessage?: string`** — 当 `stopReason` 为 `"error"` 或 `"aborted"` 时，携带错误描述。
- **`timestamp: number`** — Unix 时间戳（毫秒）。

`Usage` 类型值得单独展开：

```
// packages/ai/src/types.ts:359-380

export interface Usage {
  input: number;
  output: number;
  cacheRead: number;
  cacheWrite: number;
  cacheWrite1h?: number; // Anthropic 1 小时缓存写入的子集
  reasoning?: number;    // 推理/思考 token（output 的子集）
  totalTokens: number;
  cost: {
    input: number;
    output: number;
    cacheRead: number;
    cacheWrite: number;
    total: number;
  };
}
```

`Usage` 同时记录 token 数量和费用。`cacheRead` 和 `cacheWrite` 是 prompt caching 相关的统计 — 不是所有 provider 都支持，不支持的填 0。`cacheWrite1h` 从 `cacheWrite` 中拆出"写入 1 小时 TTL 缓存"的那部分 token，目前只有 Anthropic 上报（用于更精确的缓存计费，v0.79.4）。`reasoning` 是 v0.80.3 新增的字段：它记录模型的推理/思考 token 数，是 `output` 的**子集**（`output` 已经把这些 token 算在内），只有暴露 reasoning 明细的 provider 会填一个数字（可能是 0），其余留 `undefined`。`cost` 嵌套对象把 token 数量按各 provider 的价格转换成了美元金额，让上层不需要知道定价细节。

ToolResultMessage 的两个增补字段

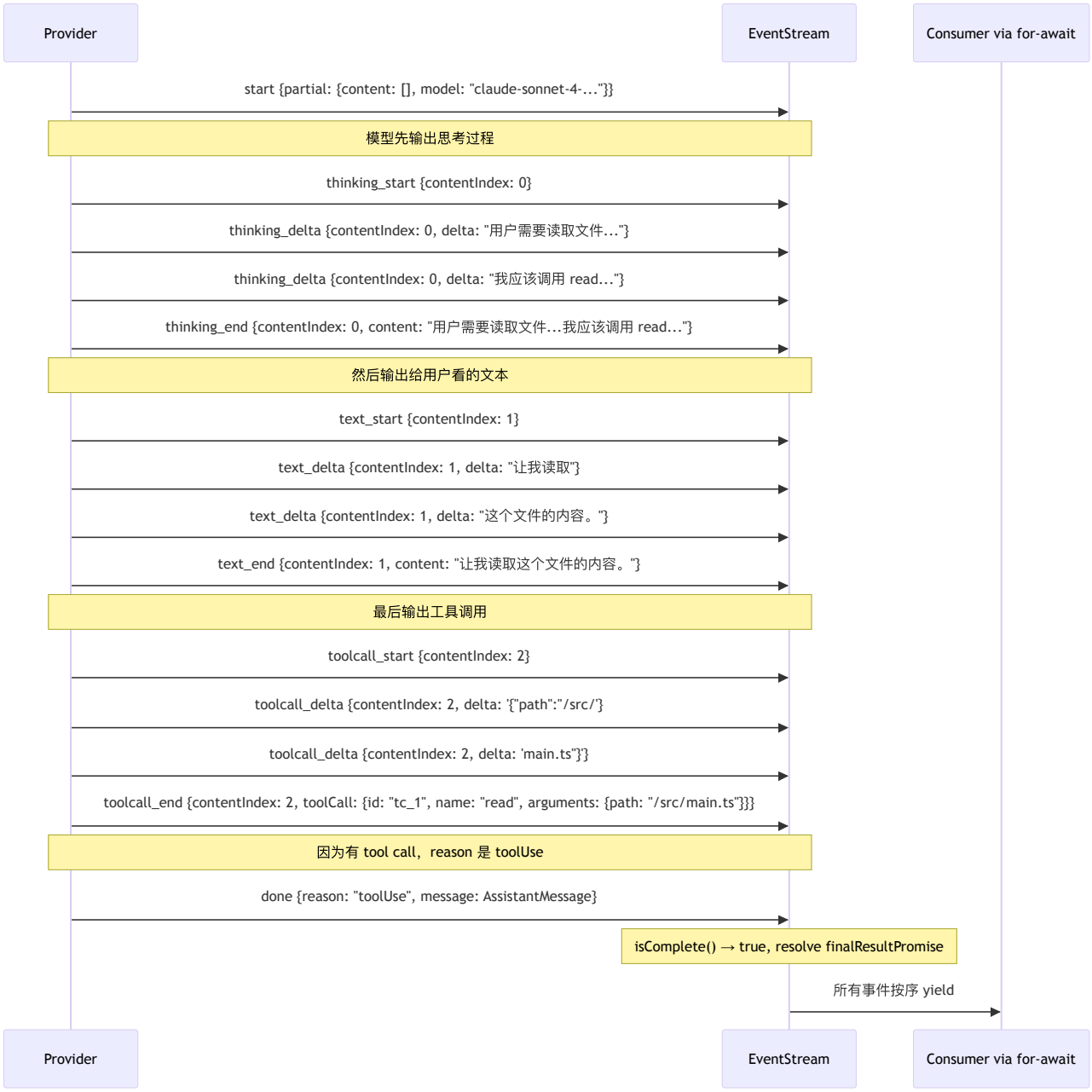
对话历史的第三种消息 `ToolResultMessage` 也随版本长出了两个新字段：

```ts
// packages/ai/src/types.ts:405-421（节选）

export interface ToolResultMessage<TDetails = any> {
  role: "toolResult";
  toolCallId: string;
  toolName: string;
  content: (TextContent | ImageContent)[];
  usage?: Usage;                // 工具自身执行的用量（不计入主 LLM 上下文核算）
  addedToolNames?: string[];  // 本次结果后新变得可用的工具名
  isError: boolean;
  timestamp: number;
}
```

- **`usage?: Usage`**（v0.81.0）— 有些工具本身就是一次 LLM 调用（例如子 agent、代码搜索），它们的 token 用量记在这里，与主循环的上下文核算分开，用于成本可观测性。
- **`addedToolNames?: string[]`**（v0.80.7）— 支持"原生延迟工具加载"的 provider 用它作为**加载点**：某个工具结果返回后才让一批新工具变得可用（`Context.tools` 里的名字）。不支持原生延迟加载的 provider 忽略这个字段，照常使用 `Context.tools`。这是把"工具集随对话动态生长"这件事编码进消息模型的一个精巧扩展点。这批新工具名如何在循环里被解析并传播，见第 9 章。

一次典型 LLM 响应的事件序列

理论分析完了，让我们看一个具体的场景：用户要求 coding agent 读取一个文件。模型决定调用 `read` 工具，同时输出一段说明文本。整个过程产生的事件序列如下：



关键观察：

1. **`contentIndex` 递增**：thinking 是 0，text 是 1，tool call 是 2。这个数字对应最终 `AssistantMessage.content` 数组的下标。
2. **每个内容块都有完整的 start/delta/end 周期**。消费者可以用 `_start` 和 `_end` 作为状态机的转换边界。
3. **`done` 的 reason 是 `"toolUse"`**，不是 `"stop"`。因为模型输出了 tool call，循环引擎（第 8 章）收到这个 reason 后知道需要执行工具并继续循环。
4. **每个中间事件都携带 `partial`**。如果消费者在收到第二个 `text_delta` 后崩溃了，重启后只需要看最后一个 `partial` 就知道完整状态。

对比另一个场景 — 纯文本回答（没有 tool call）：

事件序列会简单得多：`start` → `text_start` → 多个 `text_delta` → `text_end` → `done {reason: "stop"}`。没有 thinking（如果模型不支持或未启用 extended

thinking），没有 tool call， `contentIndex` 始终为 0。

再对比一个错误场景 — API 限流：

`start → error {reason: "error", error: AssistantMessage{stopReason: "error", errorMessage: "Rate limited: retry after 30s"}}` 。流可能只有两个事件就结束了。但消费者的处理逻辑不需要特殊化 — `for await` 正常迭代，`stream.result()` 正常返回一个 `AssistantMessage` ，只是 `stopReason` 是 `"error"` 。

取舍分析

得到了什么

1. 会话录制和回放。因为所有交互都是事件流，只需序列化事件就能录制完整的会话过程。回放时重放事件，不需要重新调用 LLM。

2. 统一的错误处理路径。成功和失败走同一条路 — 都是事件流里的事件。调用者不需要两套处理逻辑（try-catch + event handler）。

3. 消费者解耦。生产者（provider）和消费者（循环引擎、TUI、session manager）通过事件流解耦。provider 不知道谁在消费事件，消费者不知道事件来自哪个 provider。

4. 流式和非流式的统一。 `stream.result()` 让不关心过程的调用者用一行代码拿到最终结果。底层机制完全相同，不需要维护两套 API。

5. 渐进式消费。 `partial` 字段让消费者可以在任何时刻获取当前的完整快照，而不需要自己维护累积状态。这大幅简化了 TUI 渲染逻辑。

放弃了什么

1. 每个消费者都要写状态机。消费事件流比消费 `Promise<string>` 复杂得多。消费者需要跟踪 `start` / `delta` / `end` 状态，处理部分消息的累积。这是流式设计的固有复杂性。（ `partial` 字段在一定程度上缓解了这个问题，但消费者仍然需要理解事件协议。）

2. 调试更困难。事件流的执行过程是异步的、交错的。当出问题时，需要查看完整的事件序列才能定位原因，比查看一个同步的函数调用栈要困难。

3. "Must not throw"契约需要每个 provider 遵守。如果某个 provider 的实现不小心抛了异常而没有编码进事件流，调用者的 `for await` 会收到一个意外的 rejection，整个调用链可能崩溃。这个契约是信任性的，没有编译时强制。

4. `partial` 的内存开销。每个中间事件都携带完整的 `partial` AssistantMessage 快照。对于一个长回答，可能有数百个 delta 事件，每个都带一份完整快照。这是用内存换取消费者的简单性。

StreamOptions：事件流之下的扩展点

事件流的"形状"稳定，但**发起一次流**的入口 `StreamOptions` 随版本显著扩张（`types.ts:115-191`）。这些字段不改变事件协议，而是在协议之下提供 hook 和传输层旋钮：

- **`transport?: Transport`** — `"sse" | "websocket" | "websocket-cached" | "auto"`。事件流是上层抽象，底下走什么传输由它决定。`websocket-cached`（v0.71.1）让 OpenAI Codex Responses 用一条保活的 WebSocket 连接跨请求只发送**新增**的对话项，而不是每次重发整个上下文 —— 事件流的消费者完全无感。
- **`onPayload?`** — 在请求发出**前**改写 provider payload（由早期的 beforeProviderRequest 演化而来），用于注入自定义头部、改写参数。
- **`onResponse?`** — 在拿到响应后读取 HTTP 状态码与响应头（v0.67.6），用于配额提示、调试。
- **`env?`** — provider 级的环境变量覆盖（v0.79.5），让同一进程内不同请求使用不同的 API key / 代理 / 区域占位符（Cloudflare、Azure、Vertex、Bedrock）。
- 还有 `timeoutMs` / `maxRetries` / `maxRetryDelayMs` / `cacheRetention` / `sessionId` / `metadata` 等运行时旋钮。

同一条"契约稳定、旋钮生长"的主线还延伸到两个相邻的扩展点，它们同样不碰事件协议：

- **`Tool.constrainedSampling`**（`types.ts:474`，v0.80.4/0.82.0）— 让工具声明 provider 侧的约束采样配置（`ConstrainedSamplingConfig`，`types.ts:460`），把"这个工具的参数必须严格符合 schema / 语法"的意图下推给支持约束解码的 provider（对应 `supportsStrictTools` / `supportsGrammarTools` 能力元数据）。完整取值联合详见第 19 章。事件流的消费者对此完全无感 — 收到的仍是同样的 `toolcall_*` 事件，只是模型更不容易吐出非法参数。
- **`ModelsStreamTransforms.transformHeaders`**（`models.ts:58-61`，v0.80.8）— 这是一个 **Models-only** 的选项（只在 `Models.stream()` 上有效，不属于底层 provider 的 `StreamOptions`）。它在模型/认证/请求三方的 header 全部拼装完成、

即将交给 provider 之前，给上层最后一次改写机会（`models.ts:480`：`headers = await options.transformHeaders(headers ?? {})`）。`Models` 用完就把它从选项里剥掉，不会泄漏给 provider。这正体现了 v0.80.0 后 `Models` 集合作为"认证与请求装配层"的新职责。

这是 pi-ai 的一条设计主线：**事件流契约对上不变，传输与拦截能力对下生长**。消费者面对的永远是同一套 12 种事件，而 provider 与上层产品通过 `StreamOptions`（以及 `Models` 集合的 `transformHeaders`）协商越来越多的细节。

版本演化说明

本章内容已对照 pi-mono **v0.82.1**。事件协议与 `EventStream` 类几乎未变 —— 这是 pi-ai 最稳定的部分。变化集中在数据模型与入口选项：

- **类型订正**：`AssistantMessage.provider` 的类型从 `Provider` 改为 `ProviderId`（v0.80.0，`types.ts:394`）；`Provider` 现指运行时接口。
- `Usage` 新增 `reasoning`（v0.80.3）、`cacheWrite1h`（v0.79.4）；`AssistantMessage` 有 `responseModel`（v0.71.0）、`diagnostics`（v0.59.0）。
- `ToolResultMessage` 新增可选 `usage`（v0.81.0）与 `addedToolNames`（v0.80.7）。
- 扩展点扩张：`Tool.constrainedSampling`（v0.80.4/0.82.0）、Models-only 的 `transformHeaders`（v0.80.8）；`StreamOptions` 继续扩展（transport / onPayload / onResponse / env 等）。
- 事件类型随 provider 能力增加而扩展（如 `thinking_start/delta/end` 为 extended thinking 而加）。对比：v0.74.1 新增的图像生成走的是 `Promise` 而非事件流（详见第 18 章），正好反衬"为什么文本生成需要事件流"。

第 7 章：认证是一等子系统

> **定位**：本章解析为什么 pi 把认证从"一组 OAuth 工具函数"升级为与 provider 对等的一等子系统 `src/auth/`，以及它如何用四个抽象统一 API key、OAuth、环境变量、AWS profile 等所有凭证来源。 前置依赖：第 4 章（Provider 与 Models 集合）。 适用场景：当你想理解为什么认证不能推到产品层，或者想为 pi 添加新的认证方式 / OAuth provider。

认证为什么值得成为一等子系统?

这是本章的核心设计问题。

直觉上，认证应该是产品层的事：用户登录，拿到 token，传给 LLM 调用层。但 pi 不仅把认证放在 pi-ai 层，还把它从"一堆 `utils/oauth/` 工具函数"重构成了一个**独立成层的子系统** `packages/ai/src/auth/`，与 provider 注册、事件流并列。这次重构分两步落地：v0.80.0 引入 auth substrate（`ProviderAuth` / `CredentialStore` / 双检刷新），v0.80.8 完成收口（删除旧的全局 OAuth 注册表、把六回调的 `AuthLoginCallbacks` 改名收敛为 `AuthInteraction`、`checkAuth` / `getAuth` / `login` / `logout` 运行时入口收口）。为什么值得这样投入?

原因是三个约束叠加在一起，任何一个单独看都不足以成层，合在一起就必须成层：

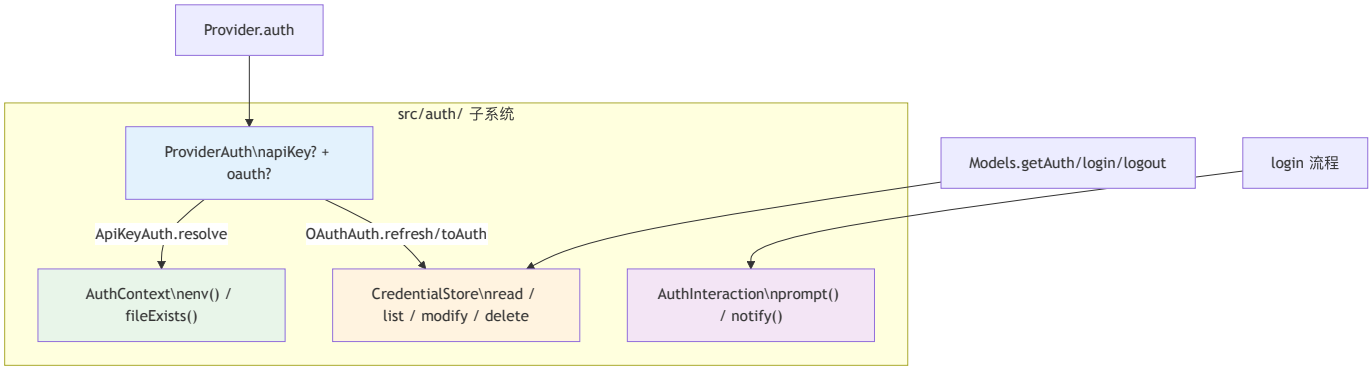
1. **凭证会过期，而 agent 的一次 run 可能持续几十分钟。** OAuth token 在第 7 次 LLM 调用时过期，谁负责刷新？如果认证在产品层，产品层得在每次调用前检查并刷新 — 但产品层不知道循环引擎何时发起调用。刷新必须下沉到调用点附近。
2. **凭证来源五花八门，但调用点只想要"一个 key"。** 同一个 provider 的凭证可能来自：存储的 API key、存储的 OAuth token、环境变量、AWS profile、ADC 文件、bearer 网关 token。调用点不该关心这些差异，它只想拿到一份可用的请求认证。
3. **刷新必须是并发安全的。** 几十分钟里多个并发请求可能同时发现 token 过期，如果各自去刷新，会互相作废对方刚换来的 token。刷新需要一把跨请求（甚至跨进程）的锁。

把"过期刷新 + 来源归一 + 并发安全"这三件事内聚成一层，让 provider 和调用点都不必操心，这就是认证成为一等子系统的理由。旧的 `utils/oauth/` 只解决了 OAuth 一种来

源，无法承载后两个约束 — 这是它被重构掉的根本原因。

认证子系统的四个抽象

`src/auth/` 用四个正交的抽象搭起整个子系统。理解了它们的分工，就理解了这一层：



- **`ProviderAuth`**（`auth/types.ts:217`）— 一个 provider 的认证能力声明，只有两个字段：`apiKey?: ApiKeyAuth` 和 `oauth?: OAuthAuth`，至少有其一。
- **`CredentialStore`**（`auth/types.ts:60`）— app 拥有的凭证存储，按 provider id 存一份凭证，`modify` 是唯一写路径。
- **`AuthContext`**（`auth/types.ts:91`）— 环境访问的注入点（`env(name)` / `fileExists(path)`），让 auth 解析在测试和浏览器里可替换。
- **`AuthInteraction`**（`auth/types.ts:150`）— 登录流程与用户交互的抽象，只有 `prompt()` 和 `notify()` 两个方法。

下面逐个展开最关键的三个。

`ProviderAuth`：一个 provider 的两条认证腿

```ts
// packages/ai/src/auth/types.ts:217-220

export interface ProviderAuth {
  apiKey?: ApiKeyAuth;
  oauth?: OAuthAuth;
}
```

设计上强制"至少有其一"：**每个 provider 都有认证语义**，哪怕它只靠环境变量、AWS profile，甚至是无 key 的本地服务器 — 这类 provider 也提供一个 `apiKey` 认证，其 `resolve()` 负责报告"这个 provider 到底配没配好"。这是一个重要的统一：`Models.getAuth()` 对未配置的 provider 一律返回 `undefined`，调用点不需要区分"这个 provider 用 key 还是用 OAuth"。

`ApiKeyAuth`（`auth/types.ts:161`）有三个方法：可选的 `login`（交互式录入 key）、可选的 `check`（无副作用的可用性探测）、必需的 `resolve`（从存储凭证 + 环境来源逐字段解析出请求认证）。绝大多数 provider 的 `ApiKeyAuth` 都由一个 helper 一行生成：

```
// packages/ai/src/auth/helpers.ts:9-27（节选）

export function envApiKeyAuth(name: string, envVars: readonly string[]):
ApiKeyAuth {
  return {
    name,
    login: async (interaction) => ({
      type: "api_key",
      key: await interaction.prompt({ type: "secret", message: `Enter
${name}` }),
    }),
    resolve: async ({ ctx, credential }) => {
      if (credential?.key) return { auth: { apiKey: credential.key },
... };
      for (const envVar of envVars) {
        const value = await ctx.env(envVar);
        if (value) return { auth: { apiKey: value }, source: envVar };
      }
      return undefined;
    },
  };
}
```

`resolve` 的逻辑是"存储凭证优先，否则按顺序试环境变量"。有非标准来源的 provider（Anthropic 支持 bearer 网关 token、Bedrock 走 AWS profile、Google 走 ADC 文件）自己写 `ApiKeyAuth`，但接口不变 — 调用点永远只调 `resolve`。

OAuthAuth：refresh / toAuth 的职责切分

OAuth 认证是 `ProviderAuth` 的另一条腿。它的接口设计里藏着这次重构最关键的一个决策：

```
// packages/ai/src/auth/types.ts:189-210

export interface OAuthAuth {
  name: string;
  loginLabel?: string;

  login(interaction: AuthInteraction): Promise<OAuthCredential>;

  /** 交换 refresh token。网络调用，失败即抛（invalid_grant 等）。
   *  Models 在 store 锁下运行它。 */
  refresh(credential: OAuthCredential, signal?: AbortSignal):
Promise<OAuthCredential>;

  /** 无副作用地从一个有效凭证派生请求认证。
   *  覆盖 per-credential baseUrl（GitHub Copilot）。 */
  toAuth(credential: OAuthCredential): Promise<ModelAuth>;
}
```

关键是把 `refresh` 和 `toAuth` **拆成两个方法**：

- **`refresh`** 是"有副作用、要网络、会失败"的那一半 — 拿 refresh token 换一个新的 credential。
- **`toAuth`** 是"无副作用、纯派生"的那一半 — 从**已经有效**的 credential 派生出这次请求要用的 `ModelAuth`（apiKey / headers / baseUrl）。

为什么拆开？因为这让 `Models` 能够**独占并封装"加锁刷新"这套并发控制**。刷新是全局串行的（要在 store 锁下、双检过期、只刷一次），而派生是每次请求都要做、且可以随便并发的。如果把它们揉进一个方法，`Models` 就无法在正确的粒度上加锁。拆开之后，`Models` 的算法变得干净：过期就在锁下 `refresh` 一次，然后无论如何都 `toAuth` 派生。

GitHub Copilot 是 `toAuth` 派生能力的最佳例子。Copilot 不允许直连 OpenAI，请求必须发到嵌在 token 里的代理地址。旧架构靠一个 `modifyModels` 回调批量改写模型的 `baseUrl`；新架构把它收进 `toAuth`：

```
// packages/ai/src/auth/oauth/github-copilot.ts:373-378

async toAuth(credential) {
  return {
    apiKey: credential.access,
    baseUrl: getGitHubCopilotBaseUrl(credential.access,
      copilotEnterpriseDomain(credential)),
  };
}
```

`baseUrl` 从 token 的 `proxy-ep` 字段解析而来，每次 `toAuth` 用最新的 credential 重新派生。这比"登录后一次性改写模型列表"更准确 — baseUrl 跟着 credential 走，token 一旦刷新，下次 `toAuth` 自动拿到新地址。

CredentialStore：modify 作为唯一写路径

凭证存储的接口把"并发安全"设计进了类型：

```
// packages/ai/src/auth/types.ts:60-88（节选）

export interface CredentialStore {
  read(providerId: string): Promise<Credential | undefined>;
  list(): Promise<readonly CredentialInfo[]>;
  modify(providerId: string,
    fn: (current: Credential | undefined) => Promise<Credential |
undefined>,
  ): Promise<Credential | undefined>;
  delete(providerId: string): Promise<void>;
}
```

注意**没有 `write`**。唯一的写路径是 `modify` — 一个串行化的 read-modify-write：`fn` 能看到当前凭证，返回新凭证或 `undefined`（表示不变）。为什么强制走 `modify`？因为正确的写入都依赖当前值：刷新要先看"是不是真过期了"、登录要看"是不是别人刚登录过"。`modify` 对每个 provider id 提供互斥（后端支持时甚至跨进程，如文件锁），保证这些 read-modify-write 不会交错。

`list()` 则明确要求"不解析、不暴露 secret、不执行配置的 API-key 命令" — 它只列出凭证的元数据（provider id + 类型），供账户/状态 UI 枚举。读与列的分离，避免了状态页面无意中触发一串命令执行或 token 刷新。

store 锁下的双检过期刷新

把 `OAuthAuth.refresh` / `toAuth` 的切分和 `CredentialStore.modify` 的串行化合起来，就得到了 `resolveProviderAuth` 里那段"双检锁定"的刷新逻辑：

```
// packages/ai/src/auth/resolve.ts:101-123（节选）

if (Date.now() >= credential.expires) {
  // 乐观检查说过期了；权威检查在锁内进行
  const post = await credentials.modify(providerId, async (current) => {
    if (current?.type !== "oauth") return undefined;        // 期间登出了
    if (Date.now() < current.expires) return undefined;     // 别的请求已刷新
    try {
      return await oauth.refresh(current);
    } catch (error) {
      throw new ModelsError("oauth", `OAuth refresh failed for ${providerId}`, { cause: error });
    }
  });
  if (post?.type !== "oauth") return undefined;
  credential = post;
}
return { auth: await oauth.toAuth(credential), source: "OAuth" };
```

有效 token 走零锁的快路径（`Date.now() < expires` 直接 `toAuth`）；过期 token 才进 `modify` 锁，**在锁内再检一次过期**（别的请求可能刚刷过），确认仍过期才 `refresh` 一次，落盘后释放。这正是"存储型凭证独占 provider"的语义（`resolve.ts:40-45` 注释）：一旦存了凭证，就以它为准，刷新失败也**不静默回退到环境变量** — 因为静默回退会掩盖"你需要重新登录"这个事实。

接口收敛：六回调 → `prompt()` / `notify()`

登录流程要和用户交互 — 打开浏览器、显示设备码、等待输入、让用户选择。但 OAuth provider 不该知道自己跑在终端、Electron 还是 Slack bot 里。旧架构用一个有**六个回调**的 `OAuthLoginCallbacks` 抽象这层交互（`onAuth`、`onDeviceCode`、`onPrompt`、`onSelect`、`onProgress`、`onManualCodeInput`）。新架构把它收敛成**两个方法**：

```
// packages/ai/src/auth/types.ts:150-155

export interface AuthInteraction {
  signal?: AbortSignal;
  prompt(prompt: AuthPrompt): Promise<string>;
  notify(event: AuthEvent): void;
}
```

收敛的关键洞察是：那六个回调其实只有两类语义 — 要么**向用户要一个输入并等待返回值**，要么**向用户单向通知一件事**。于是：

- **`prompt(AuthPrompt)`** 吸收了 `onPrompt` / `onSelect` / `onManualCodeInput`。`AuthPrompt`（`auth/types.ts:119`）是一个带 `type` 的联合：`text` / `secret` / `select` / `manual_code`。`select` 返回选项 id，其余返回输入的字符串。
- **`notify(AuthEvent)`** 吸收了 `onAuth` / `onDeviceCode` / `onProgress`。`AuthEvent`（`auth/types.ts:131`）也是一个联合：`info` / `auth_url` / `device_code` / `progress`。

这是一个典型的接口收敛取舍。 得到了什么：宿主实现从"填六个具名回调"变成"写两个方法 + 两个 switch"，新增一种交互（比如未来的 passkey 流程）只需给 `AuthPrompt` / `AuthEvent` 联合加一个分支，而**不用改 `AuthInteraction` 接口本身**；同时这套接口同时服务 api-key 和 OAuth 两种登录，不再是 OAuth 专属。放弃了什么：具名回调的"自文档性" — `onDeviceCode` 一看就知道是设备码，而 `notify({ type: "device_code", ... })` 要读联合定义才知道有哪些事件；此外宿主必须处理联合里所有分支（漏了一个 `type` 就是运行时的"未知交互"）。对 pi 这种交互种类会持续增长的场景，用"数据驱动的联合"换"接口稳定性"是划算的 — 这正是把可变性从**接口形状**挪到**数据形状**的经典手法。

PKCE：为什么 CLI 应用不能有 client secret

登录流程本身（PKCE、本地回调、device-code）的机制没有随重构改变，只是交互从六回调换成了 `notify` / `prompt`。这里保留核心的 PKCE 原理。

传统 OAuth 依赖 client secret 证明身份。但 CLI 发布到用户机器上，任何人都能反编译出 secret。PKCE（RFC 7636）用一次性密码学证明替代 client secret：

```
// packages/ai/src/auth/oauth/pkce.ts:21-34（简化）

export async function generatePKCE(): Promise<{ verifier: string;
challenge: string }> {
  const verifierBytes = new Uint8Array(32);
  crypto.getRandomValues(verifierBytes);
  const verifier = base64urlEncode(verifierBytes);

  const data = new TextEncoder().encode(verifier);
  const hashBuffer = await crypto.subtle.digest("SHA-256", data);
  const challenge = base64urlEncode(new Uint8Array(hashBuffer));
  return { verifier, challenge };
}
```

发起授权时把 `challenge`（verifier 的 SHA-256）放进 URL；交换 token 时把原始 `verifier` 发给 token endpoint，服务端做 SHA-256 比对。安全性建立在"即使截获 `challenge` 也无法反推 `verifier`"上，而 `verifier` 只在本进程内存里存在。代码用 Web Crypto API 而非 Node.js `crypto` 模块，让它在 Node 20+ 和浏览器里都能跑。

Anthropic 登录：三层降级与 AuthInteraction

以 Anthropic 为例看完整流程如何落到 `notify` / `prompt` 上：

`loginAnthropic`（`auth/oauth/anthropic.ts:229`）先 `generatePKCE()` 并在 `127.0.0.1:53692` 启一个回调服务器（回调主机可经 `PI_OAUTH_CALLBACK_HOST` 配置，端口固定，`anthropic.ts:32-33`），然后 `interaction.notify({ type: "auth_url", url })` 让宿主打开浏览器。接着是三层降级：**本地回调**（同机浏览器命中 `/callback`）→ **手动粘贴**（`interaction.prompt({ type: "manual_code" })`，SSH 远程场景）→ 竞赛，谁先拿到 code 用谁。交换 token 时 `notify({ type: "progress", ... })` 报告进度。整个 provider 不知道自己在什么宿主里，只调这两个方法。

`login` 返回的 `OAuthCredential` 由 `Models.login()` 负责落盘。过期时间处理仍是"提前 5 分钟标记过期"（`anthropic.ts:338`：`expires_in * 1000 - 5 * 60 * 1000`），

留出刷新窗口，消除"请求发出后、响应返回前 token 过期"的竞态。

三种登录范式，一套交互协议

pi 同时支持三种登录范式，全部共享 `AuthInteraction`：**PKCE + 本地回调**（Anthropic）、**device-code 轮询**（GitHub Copilot 一直用、OpenAI Codex 作无头替代，共享 `auth/oauth/device-code.ts`）、以及二者的**手动粘贴降级**。宿主只要实现 `prompt` / `notify` 两个方法，三种范式都能跑。

内建 OAuth provider 也从早期的 3 家扩展到多家。当前文本侧提供 OAuth 的 provider 有 **7 家**（各自的 factory 在 `providers/*.ts` 里用 `lazyOAuth` 声明，实现在 `auth/oauth/*.ts`）：

Provider	OAuth 显示名	登录范式
anthropic	Anthropic (Claude Pro/Max)	PKCE + 本地回调
github-copilot	GitHub Copilot	device-code
openai-codex	OpenAI (ChatGPT Plus/Pro)	浏览器 / device-code
xai	xAI (Grok/X subscription)	device-code
kimi-coding	Kimi Code (subscription)	device-code
openrouter	OpenRouter OAuth	PKCE
radius	（随网关名）	device-code / PKCE

运行时入口：provider-scoped 的四个方法

调用点不直接碰上面的抽象，而是通过 `Models` 集合上四个 provider-scoped 的方法使用认证（第 4 章）：

- **`Models.checkAuth(providerId)`**（`models.ts:388`）— 无副作用地检查一个 provider 是否配好认证（不刷新 OAuth），供状态 UI 用。
- **`Models.getAuth(model | providerId)`**（`models.ts:413`）— 解析出这次请求的 `AuthResult`（apiKey/headers/baseUrl + source 标签）。请求路径（`stream`）内部就调它，过期 OAuth 在这里被透明刷新。
- **`Models.login(providerId, type, interaction)`**（`models.ts:431`）— 跑 provider 自己的登录流程并把返回的 credential 落盘。

- **`Models.logout(providerId)`**（`models.ts:447`）— 删除存储的 credential。

对循环引擎（第 8 章）来说，认证依然是透明的：它调 `models.stream(model, ...)`，`applyAuth`（`models.ts:463`）在内部完成"解析 + 刷新 + 注入 header/baseUrl"。区别在于，这个透明性现在由 `Models` 集合 + `auth/` 子系统共同提供，而不是旧的"藏在 `getApiKey` 回调里的一个 OAuth 工具函数"。

错误信息策略：从"统一简化"反转为"保留底层 cause"

这里要**订正**上一版书稿的一个取舍判断。旧的 `getOAuthApiKey` 把所有刷新错误统一成 `Failed to refresh OAuth token for ${providerId}`，**丢弃原始错误信息**，当时把它描述为"有意的信息损失，简化上层处理"。新架构的策略**反转了**：`ModelsError` 现在会把底层 cause 追加进消息：

```
// packages/ai/src/auth/resolve.ts:22-38

export class ModelsError extends Error {
  readonly code: ModelsErrorCode;
  constructor(code: ModelsErrorCode, message: string, options?: {
cause?: unknown }) {
    super(withCauseDetail(message, options?.cause), options);
    this.name = "ModelsError";
    this.code = code;
  }
}

/** Callers surface `error.message` only, so keep the underlying reason
in it. */
function withCauseDetail(message: string, cause: unknown): string {
  if (cause === undefined || cause === null) return message;
  const detail = formatThrownValue(cause).trim();
  if (!detail || message.includes(detail)) return message;
  return `${message}: ${detail}`;
}
```

`ModelsError` 用一个 `code`（`"oauth"` / `"auth"` / `"provider"` / `"stream"` / `"model_source"` / `"model_validation"`）做机器可分类的错误，同时 `withCauseDetail` 把底层原因拼进 `message`。反转的理由写在注释里：**调用方只会展示 `error.message`**，所以底层原因必须留在 message 里，否则用户看到的永远是"刷新失败了"却不知道是网络超时、refresh token 过期还是服务端拒绝。旧策略优化了"上层代码

的简单"，新策略优化了"用户和调试者能看到真相" — 对一个凭证问题高频、且失败后需要用户自己判断"是重新登录还是检查网络"的系统，后者显然更重要。同时 `code` 字段保证了需要分类处理时仍可用 `error.code === "oauth"` 判断，鱼与熊掌兼得。

历史对照：v0.66–v0.79 的 OAuth 注册表

> 以下机制在 v0.80.8 已被删除或重写，仅作历史对照。`utils/oauth/` 目录、`OAuthProviderInterface`、全局 `oauthProviderRegistry`、`getOAuthApiKey`、六回调的 `OAuthLoginCallbacks` 在当前源码树中已不存在（其扩展兼容类型残留在 `compat/extension-oauth-types.ts`）。

早期认证是 `packages/ai/src/utils/oauth/` 下的一组工具函数，核心是一个 `OAuthProviderInterface`（`id` / `name` / `login` / `refreshToken` / `getApiKey` /可选 `modifyModels`）和一个与 API provider 注册表同构的**全局 OAuth 注册表** `oauthProviderRegistry`（`registerOAuthProvider` / `unregisterOAuthProvider`，卸载内建 provider 时"恢复默认"而非删除）。运行时靠一个 `getOAuthApiKey(providerId, credentials)` 函数：查注册表、判断过期、调 `refreshToken`、`getApiKey` 提取 key，失败则统一简化错误。交互靠六回调的 `OAuthLoginCallbacks`。

这套设计在当时是自洽的，也解决了真问题（三种 OAuth 流程共享一套 UI 协议、extension 可注册新 provider）。但它有几个结构性局限，最终促成 v0.80.0–v0.80.8 的重写（v0.80.0 引入新 substrate、v0.80.8 删除旧注册表完成收口）：

1. **只覆盖 OAuth 一种来源**。API key、环境变量、AWS profile、ADC 文件各有各的处理路径，没有一个统一的"解析这个 provider 的认证"入口。新的 `ProviderAuth` 把 apiKey 和 oauth 收进同一个抽象。
2. **全局注册表 = 全局可变状态**。和第 4 章的 API 注册表一样，它是模块级单例，`unregisterOAuthProvider` 的"恢复默认"就是给共享状态打的补丁。新架构里 OAuth 下沉为每个 `Provider.auth.oauth`，随 provider 装配进集合，没有全局表。
3. **`refresh` 和"取 key"揉在一起**。`getApiKey(credentials)` 既可能触发刷新又要派生 key，`Models` 无法在正确粒度加锁。新的 `refresh` / `toAuth` 切分解决了这个问题。
4. **错误信息被统一丢弃**（见上节），且 `modifyModels` 用"批量改写模型列表"来表达"per-credential baseUrl"，绕了远路。新架构用 `toAuth` 直接派生 baseUrl。

取舍分析

得到了什么

1. 凭证来源被统一。 API key、OAuth、环境变量、AWS profile、bearer 网关全部收敛到 `ProviderAuth.resolve` / `getAuth` 一个入口，调用点只拿"一份可用认证"。

2. 并发安全内建。 `CredentialStore.modify` 唯一写路径 + `refresh` / `toAuth` 切分 = store 锁下的双检刷新，多请求/多进程不会互相作废 token。

3. 认证与 provider 对等内聚。 OAuth 随 provider 装配，不再有全局 OAuth 注册表；`Models` 提供 provider-scoped 的 `checkAuth` / `getAuth` / `login` / `logout` 。

4. 接口对增长友好。 交互收敛为 `prompt` / `notify` 两方法，新增交互种类只加联合分支；错误保留底层 cause 同时保留可分类的 `code` 。

放弃了什么

1. 一次破坏性重构的成本。 依赖 `getOAuthApiKey` 、 `OAuthLoginCallbacks` 、 `registerOAuthProvider` 的下游都要迁移到 `auth/` 子系统。

2. ai 层的职责进一步扩大。 一个"LLM 调用抽象层"如今还独立管一整层认证。但如本章开头所述，过期刷新、来源归一、并发安全这三个约束把认证和调用紧紧绑在一起，分到不同层反而要跨层协调。

3. 凭证持久化仍不在 ai 层。 `CredentialStore` 是接口，真正的落盘（文件、keychain）由产品层实现。ai 层只定义"怎么用、怎么并发安全地改"，不定义"存在哪"。

4. 交互联合的自文档性下降。 具名回调换成 `type` 联合，宿主必须读联合定义并处理所有分支。这是接口收敛的固有代价。

对 pi 的定位 — 支持几十家 provider、多种订阅制 OAuth、要在容器/SSH/桌面等各种环境登录 — 把认证做成一等子系统是值得的。它用一层抽象的投入，换来了凭证来源的统一、并发刷新的正确性，以及"认证是 provider 的对等能力"这个更清晰的心智模型。

版本演化说明

本章内容已对照 pi-mono **v0.82.1**。本章主线分两步跨过了一道破坏性分界（v0.80.0 引入、v0.80.8 收口）：

- **v0.80.0（引入 auth substrate）**：把认证从 `utils/oauth/` 抽成独立的认证子系统 `src/auth/`，落地 `ProviderAuth`（`auth/types.ts:217`）、`CredentialStore`（`:60`）、`AuthContext`、`OAuthAuth`（`login` / `refresh` / `toAuth`，`:189`），以及 store 锁下的双检过期刷新。
- **v0.80.8（Breaking，完成收口）**：删除旧的全局 OAuth 注册表 `oauthProviderRegistry`（`register/unregisterOAuthProvider`）、`OAuthProviderInterface`、`getOAuthApiKey`；把六回调的 `AuthLoginCallbacks` 改名收敛为 `AuthInteraction` 的 `prompt()` / `notify()`（`auth/types.ts:150`）；运行时入口收口为 provider-scoped 的 `Models.checkAuth()` / `getAuth()` / `login()` / `logout()`。
- **内建 OAuth provider 扩展**：从 3 家扩到 7 家（+xAI、Kimi Coding、OpenRouter、Radius）。
- **v0.82.1**：`ANTHROPIC_AUTH_TOKEN` bearer 网关支持（`providers/anthropic.ts:21-27`）；`ModelsError` 追加底层 cause（`auth/resolve.ts:22-45`）— 上一版书稿"刷新错误被统一简化、丢弃原始信息"的取舍**已反转**，见本章"错误信息策略"节。 早期（v0.66–v0.79）的 OAuth 注册表叙述见"历史对照"节；PKCE、三层降级、device-code 等登录范式被新架构继承，只是交互从六回调换成了 `prompt` / `notify`。

第 8 章：agentLoop — 发动机只管转

> **定位**：本章解剖 pi 整个系统真正的发动机 — 无状态的 agent 循环引擎。 前置依赖：第 6 章（事件流设计）。 适用场景：当你想理解"一次 agent 执行到底经历了什么"，或者想把 pi 的循环引擎用在自己的系统里。

一个 agent 循环应该知道多少?

这是本章的核心设计问题。

一个直觉上的答案是"越多越好" — 循环引擎应该知道怎么管理会话、怎么重试失败的请求、怎么压缩超长上下文、怎么持久化中间状态。毕竟，这些都是"循环过程中"会遇到的问题。

但 pi 给出了一个反直觉的答案：**循环引擎应该知道尽可能少的东西。**

打开 `packages/agent/src/agent-loop.ts`，文件头的注释只有一句话：

```
Agent loop that works with AgentMessage throughout.
Transforms to Message[] only at the LLM call boundary.
```

这句话定义了整个循环引擎的边界：它只管把消息送进 LLM、把 LLM 的响应拿回来、如果响应里有工具调用就执行工具、然后决定要不要继续。它不知道消息从哪来，不知道消息要存到哪去，不知道哪些工具应该被允许，不知道上下文快溢出了。

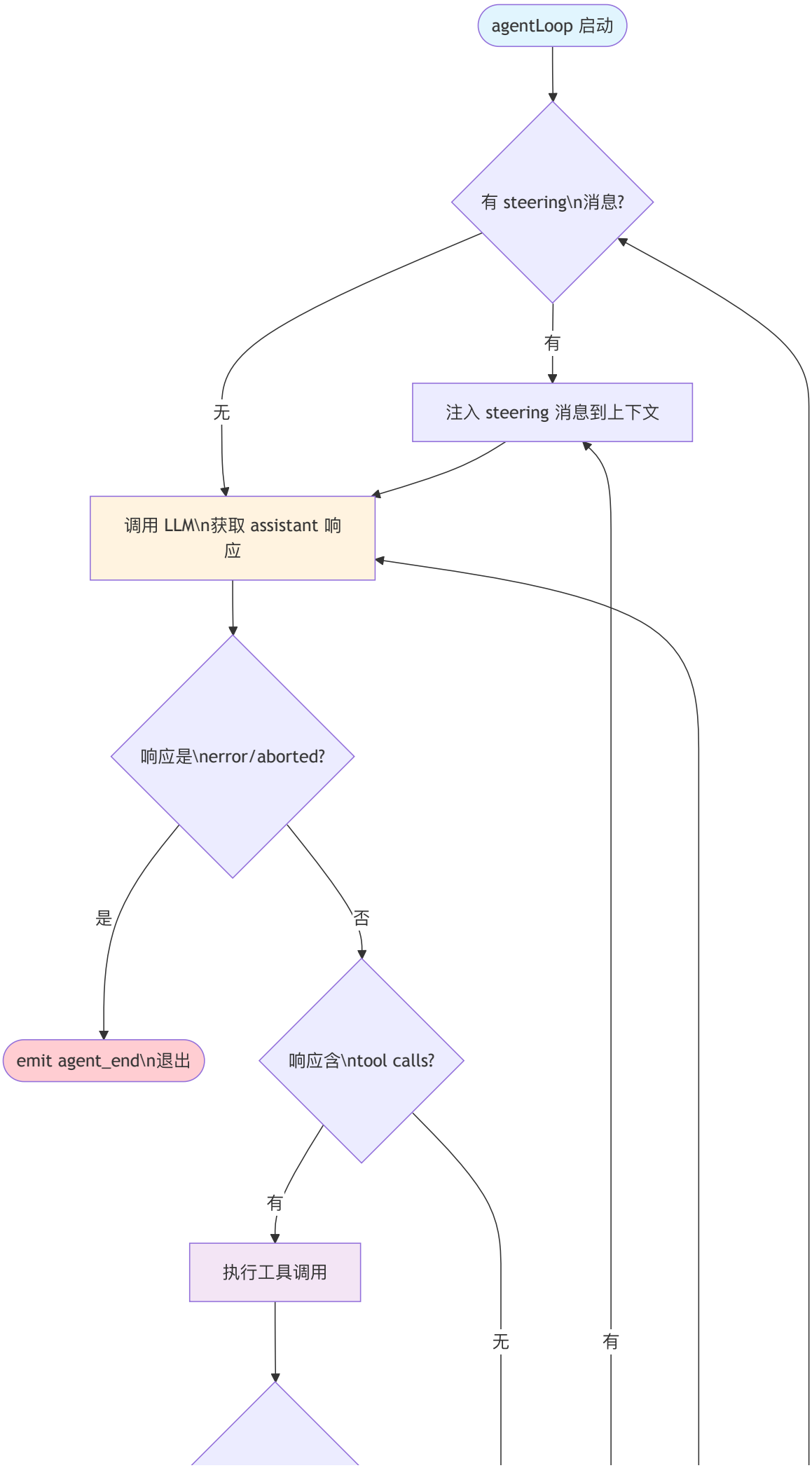
这个选择的代价是：所有这些"循环之外"的功能都必须由上层来实现。

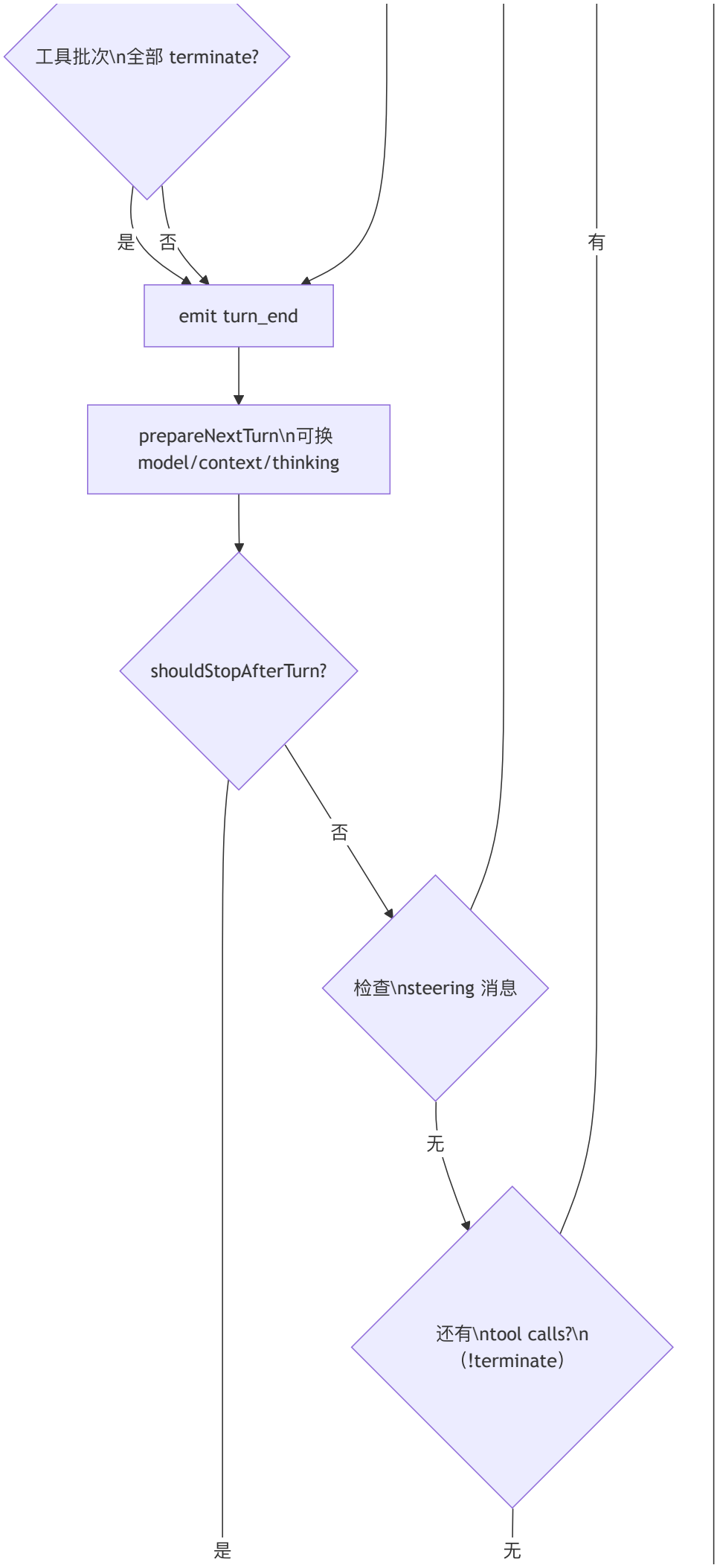
这个选择的收益是：循环引擎可以被任何上层随意组合 — 终端 CLI、Slack bot、Web UI、甚至一个测试用例，都可以用同一个循环，只要提供不同的配置。

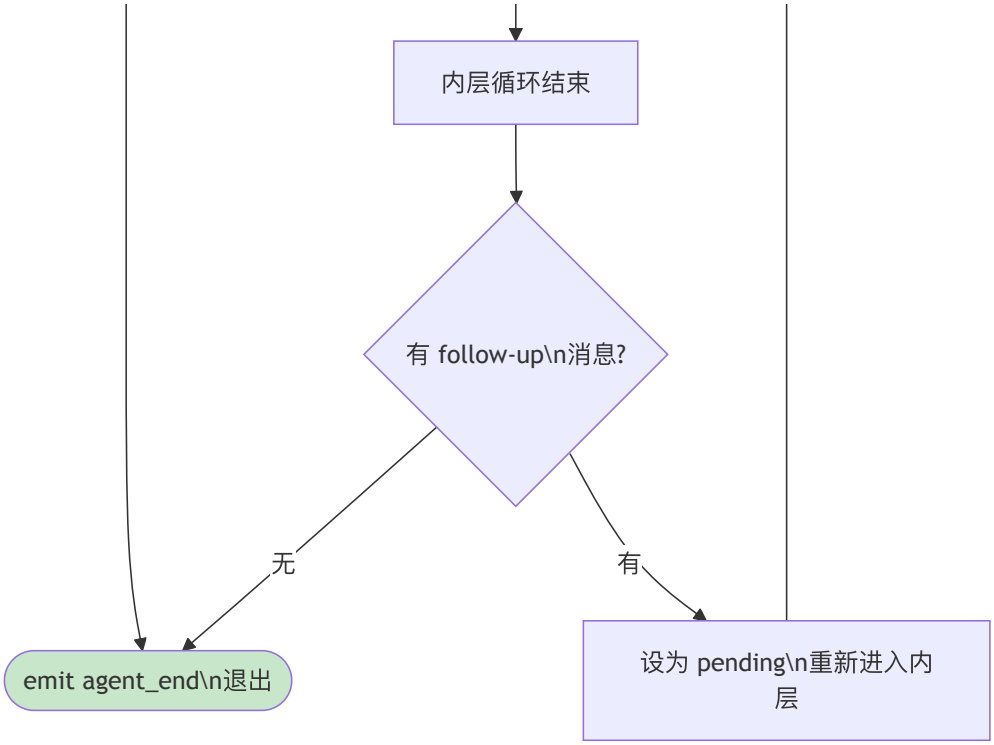
接下来我们拆解这个引擎的内部结构。

双层循环：内层忙碌，外层等待

`agentLoop` 的核心是一个约 70 行的 `runLoop()` 函数。它的设计可以用一张图概括：


agentLoop 启动
有 steering\n消息?
有
无
注入 steering 消息到上下文
调用 LLM\n获取 assistant 响应
响应是\nerror/aborted?
是
否
emit agent_end\n退出
响应含\ntool calls?
有
无
执行工具调用
有


工具批次\n全部 terminate?
是
否
有
emit turn_end
prepareNextTurn\n可换 model/context/thinking
shouldStopAfterTurn?
否
检查\nsteering 消息
无
还有\ntool calls?\n(!terminate)
是
无



这张图展示了两个嵌套的 `while` 循环各自的职责：

内层循环负责"持续工作"：调用 LLM → 执行工具 → 检查 steering 消息 → 再调用 LLM。只要还有工具调用要执行，或者有用户的 steering 消息要注入，它就不停。

外层循环负责"被唤醒"：当内层循环结束（agent 本来要停下来），外层检查有没有 follow-up 消息。有的话，把它设为 pending，重新启动内层循环。

为什么要分两层？因为 steering 和 follow-up 的语义不同：

- **Steering**（转向）：用户在 agent 工作过程中插入一条新指令，比如"别改那个文件，换一种方式"。它在当前 turn 的工具执行完成后注入，影响下一次 LLM 调用。
- **Follow-up**（追加）：用户在 agent 完成后追加一条新任务，比如"好的，现在写测试"。它只在 agent 本来要退出时才被消费。

这两种消息的消费时机不同，所以需要两层循环来区分。如果只有一层循环，就没法区分"agent 还在干活时插入的指令"和"agent 干完活后追加的新任务"。

源码解剖：runLoop() 的 80 行

让我们看看实际代码。为了聚焦设计，这里展示 `runLoop()` 的核心结构（简化版，省略了 `turn_start` / `turn_end` 事件发射和 `firstTurn` 首轮保护逻辑，完整版见源码）：

```
// packages/agent/src/agent-loop.ts:155-320（简化）

async function runLoop(
  currentContext: AgentContext,
  newMessages: AgentMessage[],
  config: AgentLoopConfig,
  signal: AbortSignal | undefined,
  emit: AgentEventSink,
  streamFunction: StreamFn,    // ← 必填：底层循环不自带兜底（agent-
loop.ts:161）
): Promise<void> {
  let pendingMessages: AgentMessage[] =
    (await config.getSteeringMessages?.()) || [];

  // 外层循环：处理 follow-up 消息
  while (true) {
    let hasMoreToolCalls = true;

    // 内层循环：处理 tool calls 和 steering 消息
    while (hasMoreToolCalls || pendingMessages.length > 0) {
      // emit turn_start（首轮由调用者发射，此处省略）

      // 1. 注入 pending 消息，每条发射 message_start/end
      if (pendingMessages.length > 0) {
        for (const message of pendingMessages) {
          // emit message_start, message_end
          currentContext.messages.push(message);
          newMessages.push(message);
        }
        pendingMessages = [];
      }

      // 2. 调用 LLM，获取 assistant 响应
      const message = await streamAssistantResponse(
        currentContext, config, signal, emit, streamFunction
      );
      newMessages.push(message);

      // 3. 错误或中止 → emit turn_end + agent_end，退出
      if (message.stopReason === "error"
        || message.stopReason === "aborted") {
        return;
      }
```

```
    // 4. 提取 tool calls，有则执行
    const toolCalls = message.content
      .filter((c) => c.type === "toolCall");
    const toolResults: ToolResultMessage[] = [];
    hasMoreToolCalls = false;

    if (toolCalls.length > 0) {
      const batch = await executeToolCalls(
        currentContext, message, config, signal, emit
      );
      for (const result of batch.messages) {
        currentContext.messages.push(result);
        newMessages.push(result);
      }
      toolResults.push(...batch.messages);
      // 工具可主动叫停：整批结果都标记 terminate 时不再续轮
      hasMoreToolCalls = !batch.terminate;
    }

    // emit turn_end

    // 5. turn 间热切换：可替换 context / model / thinkingLevel
    const snapshot = await config.prepareNextTurn?.({
      message, toolResults, context: currentContext, newMessages,
    });
    // ...（应用 snapshot 到 currentContext / config，此处省略）

    // 6. turn 后优雅停止
    if (await config.shouldStopAfterTurn?.({
      message, toolResults, context: currentContext, newMessages,
    })) {
      return; // emit agent_end 后退出
    }

    // 7. 检查有无 steering 消息
    pendingMessages =
      (await config.getSteeringMessages?.()) || [];
  }

  // 内层结束，检查 follow-up 消息
  const followUpMessages =
    (await config.getFollowUpMessages?.()) || [];
  if (followUpMessages.length > 0) {
    pendingMessages = followUpMessages;
    continue; // 重新进入内层
```

```
      }

      break; // 无 follow-up，真正退出
    }
  }
```

整个函数只有一个 `return`（错误退出）和一个 `break`（正常退出）。控制流清晰到可以逐行朗读。

注意几个设计细节：

1. `pendingMessages` 的复用。无论是 steering 消息还是 follow-up 消息，都通过同一个 `pendingMessages` 变量注入内层循环。外层循环的唯一动作就是把 follow-up 消息赋值给 `pendingMessages`，然后 `continue` 重新进入内层。两种消息共享同一条注入通道，但消费时机不同。

2. 错误通过 `stopReason` 传递，而不是异常。当 LLM 调用失败时，`streamAssistantResponse` 不会抛异常 — 它返回一个 `stopReason` 为 `"error"` 的消息。这和第 6 章讲的"错误编码进事件流"的设计一脉相承。循环引擎不需要 try-catch，它只需要检查 `stopReason`。

3. 函数签名是纯函数式的。`runLoop` 接收 context、config、signal，返回 void（通过 `newMessages` 数组收集产出）。它不持有任何状态，不修改任何外部变量（除了 `currentContext.messages` 和 `newMessages` 这两个被调用者传入的可变引用）。

4. 内层循环靠 `terminate` 决定续轮，而不是"还有没有 tool calls"。早期版本里内层条件是 `hasMoreToolCalls = toolCalls.length > 0` — 只要这一轮发生过工具调用就一定再调一次 LLM。v0.69.0 起，`executeToolCalls` 不再返回一个 `ToolResultMessage[]`，而是返回 `{ messages, terminate }`（`agent-loop.ts:208-210`、`:390-393`）。续轮条件变成 `hasMoreToolCalls = !executedToolBatch.terminate`。

`terminate` 是一个**工具主动叫停**的提示：工具结果（`AgentToolResult.terminate`，`types.ts:354`）或 `afterToolCall` 返回的 `terminate`（`types.ts:80`）都能设置它。但叫停只在**整批工具结果都标记 `terminate` 时**才生效 —— `shouldTerminateToolBatch()` 要求 `finalizedCalls.every(f => f.result.terminate === true)`（`agent-loop.ts:544-545`）。这给了工具一种干净的"任务已完成，无需再让模型说话"的退出通道（例如一个显式的 `finish` / `done` 工具），而不必依赖模型自己决定停止。

5. turn 之间有两个新的干预点。每个 turn 的 `turn_end` 之后、决定是否发起下一次 LLM 调用之前，循环依次调用两个可选回调（`agent-loop.ts:226-251`）：

- `prepareNextTurn(ctx)`（v0.72.0）：返回一个 `AgentLoopTurnUpdate`，可在 turn 之间**热切换** `context` / `model` / `thinkingLevel`。返回 `undefined` 则沿用当前配置。这让"先用便宜模型探路、命中难点再升档"这类策略可以在**同一次循环运行内**完成，而不必结束循环重启。
- `shouldStopAfterTurn(ctx)`（v0.72.0）：返回 `true` 则在本 turn 后优雅退出（emit `agent_end`），且发生在 steering / follow-up 轮询之前。它与 `terminate` 的区别是控制权归属：`terminate` 由工具发出，`shouldStopAfterTurn` 由调用者（上层）发出。

`prepareNextTurn` 的两个变体（v0.80.3）。在循环配置这一层，`prepareNextTurn` 只有一个签名 —— `prepareNextTurn(context: PrepareNextTurnContext)`（`types.ts:224`）。但运行时壳 `Agent`（第 10 章）在它之上暴露了**两个**可选变体，让上层按需要选择回调拿到多少信息：

- `prepareNextTurn(signal)`（`agent.ts:107`）：只关心"该不该换配置"，不需要 loop context 的轻量变体，只收到一个 abort `signal`。
- `prepareNextTurnWithContext(context, signal)`（`agent.ts:110`）：既要 loop context（消息、上一条 assistant 响应、工具结果等），又要 abort `signal` 的完整变体。

两者同时提供时**优先调用带 context 的变体**：`Agent.createLoopConfig()` 把它们折叠成循环需要的单个 `prepareNextTurn(context)`，内部先看 `prepareNextTurnWithContext` 在不在，在就调它、否则退回 `prepareNextTurn`（`agent.ts:448-454`）：

```
// packages/agent/src/agent.ts:448-454（简化）
prepareNextTurn: this.prepareNextTurnWithContext || this.prepareNextTurn
  ? async (context) => {
      if (this.prepareNextTurnWithContext)
        return await this.prepareNextTurnWithContext(context,
this.signal);
      return await this.prepareNextTurn?.(this.signal);
    }
  : undefined,
```

注意这里把 `this.signal` 一路带了进去 —— **abort signal 现在会下发到 turn 间回调**。这让"turn 间热切换"的决策逻辑也能感知中止：如果这次 run 已经被 abort，回调可以据此跳过昂贵的 model 切换或 context 重算，而不是等切换做完才发现白做。

消息变换管道：只在 LLM 边界发生

`runLoop()` 把"调 LLM"委托给了 `streamAssistantResponse()`。这个函数做了一件非常重要的事：**把 AgentMessage 世界和 LLM Message 世界桥接起来。**

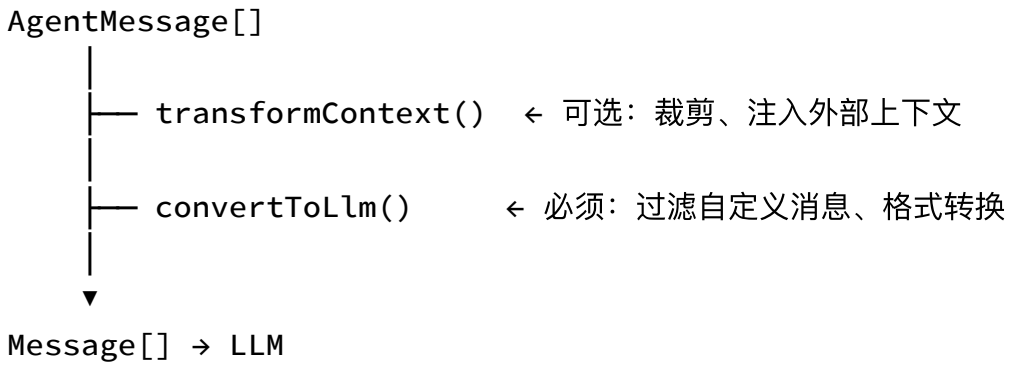
```
// packages/agent/src/agent-loop.ts:286-320（简化，完整的流式
// 事件处理约 90 行，这里只展示管道结构）

async function streamAssistantResponse(
  context: AgentContext,
  config: AgentLoopConfig,
  signal: AbortSignal | undefined,
  emit: AgentEventSink,
  streamFunction: StreamFn,
): Promise<AssistantMessage> {
  // 第一步：AgentMessage[] → AgentMessage[]（可选裁剪）
  let messages = context.messages;
  if (config.transformContext) {
    messages = await config.transformContext(messages, signal);
  }

  // 第二步：AgentMessage[] → Message[]（格式转换）
  const llmMessages = await config.convertToLlm(messages);

  // 第三步：组装 LLM 上下文并调用
  const llmContext: Context = {
    systemPrompt: context.systemPrompt,
    messages: llmMessages,
    tools: context.tools,
  };

  const response = await streamFunction(
    config.model, llmContext, { ...config, signal }
  );
  // ... 事件流处理 ...
}
```

这里有一条关键的设计边界：**AgentMessage 和 LLM Message 是两种不同的类型。**

`AgentMessage` 是 pi 的内部消息格式，它可以包含自定义消息类型（通过 `CustomAgentMessages` 声明合并扩展），可以包含 UI 通知、compaction 摘要、分支标记等 LLM 根本不需要看到的内容。

`Message` 是 pi-ai 层定义的 LLM 兼容消息格式，只包含 LLM 能理解的内容： `user` 、 `assistant` 、 `toolResult` 。

两者之间的转换通过一条两步管道完成：



为什么 `transformContext` 和 `convertToLlm` 要分开?

`transformContext` 操作的是 `AgentMessage[]` ，它知道所有自定义消息类型。典型用途是 context window 管理 — 当消息太多时，裁剪老消息或替换为摘要。这个操作必须在 `AgentMessage` 层面完成，因为自定义消息可能包含裁剪决策所需的元数据。

`convertToLlm` 操作的是从 `AgentMessage[]` 到 `Message[]` 的转换。它过滤掉 LLM 不认识的消息类型（比如 `notification` 、 `compaction_summary` ），把自定义消息转换成 LLM 能理解的格式。

如果把这两步合成一步， `transformContext` 就必须同时理解 AgentMessage 语义和 LLM 消息格式 — 关注点耦合了。

为什么转换只在 LLM 调用边界发生?

文件头注释给出了答案： `Agent loop that works with AgentMessage throughout. Transforms to Message[] only at the LLM call boundary.`

循环内部全程使用 `AgentMessage` 。工具执行返回的是 `ToolResultMessage` （AgentMessage 的一种），用户的 steering 消息也是 `AgentMessage` 。转换只在调用 LLM 的那一刻发生。

这意味着循环内部可以处理任意的自定义消息类型，而不需要关心 LLM 是否认识它们。自定义消息在循环内部是一等公民，只在出门见 LLM 时才被过滤。

stream 函数从哪来：底层必填，入口兜底

`streamAssistantResponse` 最后调用的那个 `streamFunction`，是循环引擎与 LLM 之间**唯一**的耦合点 —— 循环不 import 任何 provider，它只持有一个"给我 context、还我事件流"的函数。这个函数到底从哪来，是本章主线（"循环引擎不依赖 provider 目录"）落到实处的地方，也是 v0.81 周期里一次值得记录的取舍反复。

先看现在的分层。底层的 `runLoop` 把 `streamFunction: StreamFn` 列为**必填**参数（`agent-loop.ts:161`）—— 底层循环不为缺失的 stream 函数兜底，谁调用它谁就得把这个函数备齐。而面向使用者的公共入口 `agentLoop` / `runAgentLoop` 则把这个参数放宽为可选，并在转交给 `runLoop` 之前做一次兜底（`agent-loop.ts:116` 、 `:141`）：

```
// packages/agent/src/agent-loop.ts:116（公共入口的兜底）
await runLoop(
  currentContext, newMessages, config, signal, emit,
  streamFn ?? getDefaultStreamFn(),   // ← 调用者没给就取宿主默认
);
```

`getDefaultStreamFn()` 来自一个全新的、只有二十行的模块 `stream-fn.ts`。它维护一个进程级的默认 stream 函数，由宿主通过 `setDefaultStreamFn()` 注入；没注入就取用时抛错，明确要求"要么显式传 `streamFn`，要么先 `setDefaultStreamFn()`"：

```
// packages/agent/src/stream-fn.ts（节选）
let defaultStreamFn: StreamFn | undefined;

export function setDefaultStreamFn(fn: StreamFn | undefined): void {
  defaultStreamFn = fn;
}
export function getDefaultStreamFn(): StreamFn {
  if (!defaultStreamFn)
    throw new Error("No default stream function configured. ...");
  return defaultStreamFn;
}
```

关键在于这套注入机制**不让 agent 包 import provider 目录**。宿主（比如 coding-agent）在启动时把自己那套模型运行时的 stream 函数装进来，agent 包始终只认 `StreamFn` 这个类型，不认某个具体的 provider catalog 或 compat 层。

这套设计不是一步到位的，而是一次**取舍演进**，正好印证了主线：

- **v0.81.0**：把底层 `runLoop` 的 `streamFn` 从可选改为**必填的** `streamFunction`，同时删掉了循环内对 pi-ai/compat 层的隐式依赖。动机很纯粹 —— 循环引擎不该"偷

偷"知道有个默认 provider 可用；要用哪套 stream，必须由调用方显式交出。代价是所有直接调用底层循环的路径（包括老的已编译消费者）都必须自带 stream 函数，否则直接失败。

- **v0.81.1**：很快补回了一层**宿主可配置的 fallback**（`setDefaultStreamFn` / `getDefaultStreamFn`），但把它放在**公共入口**而非底层循环里。这样既保留了"底层必填、不依赖 provider 目录"的纪律，又让宿主能在应用边界注入一次默认值，免得每个调用点都手动传 `streamFn`。

这一改一补的净结果是：**依赖倒置的方向被彻底摆正**。底层循环对 stream 函数是"必填、无兜底"，把 provider 的存在性完全推给调用方；宿主级的默认注入则被限制在入口层，成为一个可选的便利，而不是循环的内在假设。这正是"循环引擎不依赖 provider 目录"这条主线在 v0.81 周期的一次具体兑现。

（注意：`Agent` 类一侧的 `streamFn` 选项**仍然存在**（`agent.ts:101`），只是它在内部被规整为 `this.streamFunction = runtimeOptions.streamFn ?? getDefaultStreamFn()`（`agent.ts:216`）—— 同一套"入口兜底"逻辑。要强调的是：`AgentOptions.streamFn` 在**类型上仍是必填**（字段没有 `?`），所以 TS 调用方并不能省略它；这里的 `?? getDefaultStreamFn()` 兜底只在**运行时**对 JS 调用方或显式传入 `undefined` 的情形生效，不要据此以为 TS 下可以不传。第 10 章还会再遇到它。）

`AgentLoopConfig`：循环引擎的全部知识

一个"无状态引擎"需要知道什么才能工作？答案就藏在 `AgentLoopConfig` 里。这个类型定义了循环引擎的全部外部依赖：

```
// packages/agent/src/types.ts:96-214（关键字段，简化）

interface AgentLoopConfig extends SimpleStreamOptions {
  model: Model<any>;

  // 消息变换管道
  convertToLlm: (messages: AgentMessage[])
    => Message[] | Promise<Message[]>;
  transformContext?: (messages: AgentMessage[], signal?: AbortSignal)
    => Promise<AgentMessage[]>;

  // 消息队列
  getSteeringMessages?: () => Promise<AgentMessage[]>;
  getFollowUpMessages?: () => Promise<AgentMessage[]>;

  // turn 级控制（v0.72.0 新增）
  shouldStopAfterTurn?: (ctx: ShouldStopAfterTurnContext)
    => boolean | Promise<boolean>;
  prepareNextTurn?: (ctx: PrepareNextTurnContext)
    => AgentLoopTurnUpdate | undefined
     | Promise<AgentLoopTurnUpdate | undefined>;

  // 工具执行控制
  beforeToolCall?: (context, signal?)
    => Promise<BeforeToolCallResult | undefined>;
  afterToolCall?: (context, signal?)
    => Promise<AfterToolCallResult | undefined>;
  toolExecution?: "sequential" | "parallel";

  // API 密钥动态解析（支持同步或异步返回）
  getApiKey?: (provider: string)
    => Promise<string | undefined> | string | undefined;
}
```

注意这个设计的纪律：

- `convertToLlm` 是必须提供的（循环没有默认的转换逻辑，但 `Agent` 类提供了一个默认实现：只保留 `user`、`assistant`、`toolResult` 三种角色）
- `transformContext` 是可选的（不提供就不裁剪）
- `getSteeringMessages` 和 `getFollowUpMessages` 是可选的（不提供就没有消息队列）
- `shouldStopAfterTurn` 和 `prepareNextTurn` 是可选的（不提供就既不提前停、也不在 turn 间换模型）

- `beforeToolCall` 和 `afterToolCall` 是可选的（不提供就没有工具钩子；返回 `undefined` 表示不做任何修改）
- `getApiKey` 是可选的（注释说明了用途："important for expiring tokens"，支持同步或异步返回以适配不同的认证后端）

循环引擎不假设任何可选功能的存在。它只在功能被提供时使用它们。这就是为什么同一个循环可以被极简的测试用例使用（只提供 `model` 和 `convertToLlm`），也可以被全功能的 coding agent 使用（提供所有字段）。

多数回调函数的注释里有一条统一的契约：**"Contract: must not throw or reject."** 这和 `StreamFn` 的契约一脉相承（详见第 6 章）。`convertToLlm`、`transformContext`、`getApiKey`、`getSteeringMessages`、`getFollowUpMessages` 都遵守此契约 — 循环引擎不对它们做错误恢复，如果抛了异常整个循环会意外终止。

但 `beforeToolCall` 和 `afterToolCall` 例外 — 它们**没有**此契约。循环引擎对它们做了防御性 try-catch（详见第 9 章）。这个区别反映了一个设计判断：消息变换管道是系统内部代码，有义务保证不出错；而工具钩子可能是外部扩展代码，引擎需要为它们兜底。

事件发射：循环的唯一输出通道

`runLoop()` 的返回值是 `Promise<void>` — 它不返回任何东西。循环的所有产出都通过 `emit` 回调发射：

```
type AgentEventSink = (event: AgentEvent) => Promise<void> | void;
```

这是一个故意的设计选择：**循环引擎不决定谁消费它的产出**。它只管往 `emit` 里塞事件，至于这些事件是被 TUI 渲染、被 session manager 持久化、还是被测试用例断言，循环不知道也不关心。

事件类型形成了一个完整的生命周期：

```
agent_start
  └── turn_start
        ├── message_start (user/steering message)
        ├── message_end
        ├── message_start (assistant response)
        ├── message_update (streaming delta)
        ├── message_end
        ├── tool_execution_start
        ├── tool_execution_update (partial result)
        ├── tool_execution_end
        └── turn_end
  └── turn_start (next turn)
        └── ...
agent_end
```

每个订阅者都能从这个事件流中重建整个执行过程的完整状态。这也是为什么 pi 能做会话录制和回放 — 只需要序列化事件流。

取舍分析

现在让我们退后一步，评估这个设计的得失。

得到了什么

1. 极致的可组合性。循环引擎是一个纯函数 — 给它输入（context + config），它产出事件流。任何上层都可以组合它：

- CLI 的 Agent 类用它跑交互式会话
- Slack bot 的 mom 用它跑一次性回复
- 测试用例直接调用 `runAgentLoop()` 验证行为
- 甚至可以用两个循环嵌套实现 sub-agent（外层循环的工具调用里启动内层循环）

2. 可测试性。因为循环不持有状态，测试只需要构造输入和检查输出。不需要 mock 数据库、不需要启动 UI、不需要连接真实的 LLM（提供一个假的 `streamFn` 就行）。

3. 关注点分离。会话持久化是 session manager 的事；UI 渲染是 TUI 的事；错误重试是上层的事；context 压缩是 compaction 模块的事。循环不参与其中任何一个。

放弃了什么

1. 不能自己管理会话。循环不知道"会话"这个概念的存在。它每次调用都接收一个新鲜的 `AgentContext`，不关心这个 context 是从哪来的（内存、JSONL 文件、数据库、测试固件）。如果你想要会话持久化，必须在上层实现。

2. 不能自己重试。当 LLM 返回错误时，循环直接退出。它不会自动重试，不会 backoff，不会切换备用模型。所有重试逻辑必须由调用者实现（通常是 Agent 类或 AgentSession）。

3. 不能自己压缩上下文。当 context window 快满时，循环不会自动触发 compaction。它只是把消息传给 `transformContext` — 如果调用者没提供这个回调，循环会带着越来越长的上下文继续调用 LLM，直到 provider 拒绝请求。

4. 需要上层正确组装 `AgentLoopConfig`。循环的灵活性是以配置复杂度为代价的。`AgentLoopConfig` 有十几个字段，每个回调都有自己的契约（must not throw）。如果上层组装错了 — 比如 `convertToLlm` 遗漏了某种自定义消息类型 — 循环不会报错，只会产生意外的 LLM 行为。

这个取舍值得吗?

对于 pi 的定位 — 一个支撑多种产品形态的 agent 运行时 — 这个取舍是值得的。

如果循环引擎自己管理会话，Slack bot（用 channel 做会话）和 CLI（用 JSONL 文件做会话）就需要不同的循环实现。如果循环引擎自己重试，有些场景（自动化管道）想要快速失败，有些场景（交互式 CLI）想要无限重试，循环就要为不同的重试策略膨胀。

把循环做薄，是为了让上层做厚时有足够的自由度。

这正是整本书的主线：如何用尽可能薄的内核，撑起尽可能丰富的产品。在循环引擎这一层，这条主线得到了最纯粹的体现。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，并已对照 v0.82.1。 `runLoop()` 的双层循环结构 自引入以来保持稳定，steering/follow-up 消息队列在早期版本中从单队列拆分为双队列。 v0.69.0 起内层续轮条件由"是否还有 tool calls"改为基于工具批次的

> `terminate` 提示（`executeToolCalls` 现返回 `{ messages, terminate }`）；v0.72.0 在每个 turn 之间新增 `prepareNextTurn`（热切换 model/context/thinkingLevel）与 `shouldStopAfterTurn`（turn 后优雅停止）两个可选回调。v0.80.3 起 `Agent` 在 `prepareNextTurn(signal)` 之外 增补带 loop context 的 `prepareNextTurnWithContext(context, signal)` 变体（两者并存时优先 后者），且 abort signal 已下发到 turn 间回调。v0.81.0 把底层 `runLoop` 的 stream 函数由可选 `streamFn?` 改为必填的 `streamFunction`（剥离循环对 pi-ai/compat 层的依赖），v0.81.1 又在 公共入口补回宿主可配置的默认兜底（`stream-fn.ts` 的 `setDefaultStreamFn` / `getDefaultStreamFn`，入口以 `streamFn ?? getDefaultStreamFn()` 兜底）；`Agent` 类的 `streamFn` 选项仍在（`agent.ts:101`），内部规整为 `streamFunction`。

第 9 章：工具执行不是插件调用

> **定位**：本章解剖循环引擎中工具执行管道的三阶段设计。 前置依赖：第 8 章（agentLoop 双层循环）。 适用场景：当你想理解 pi 如何让工具调用既安全又灵活，或者想为自己的 agent 系统设计工具执行策略。

为什么工具调用不能简单地"调一下"?

上一章展示了 `runLoop()` 如何在内层循环中调用 `executeToolCalls()` 。那个调用看起来只是一行代码：

```
const batch = await executeToolCalls(
  currentContext, message, config, signal, emit
);
// batch.messages: ToolResultMessage[]（按 LLM 源顺序排列）
// batch.terminate: 整批是否请求 agent 停止（见第 8 章）
```

（v0.69.0 起 `executeToolCalls` 的返回类型从裸 `ToolResultMessage[]` 变成 `{ messages, terminate }` ， `terminate` 喂给上一章的内层续轮判断。）

但如果展开 `executeToolCalls()` ，你会发现它并不是简单地"找到工具，传参数，拿结果"。它是一条精心设计的三阶段管道：**prepare → execute → finalize**。

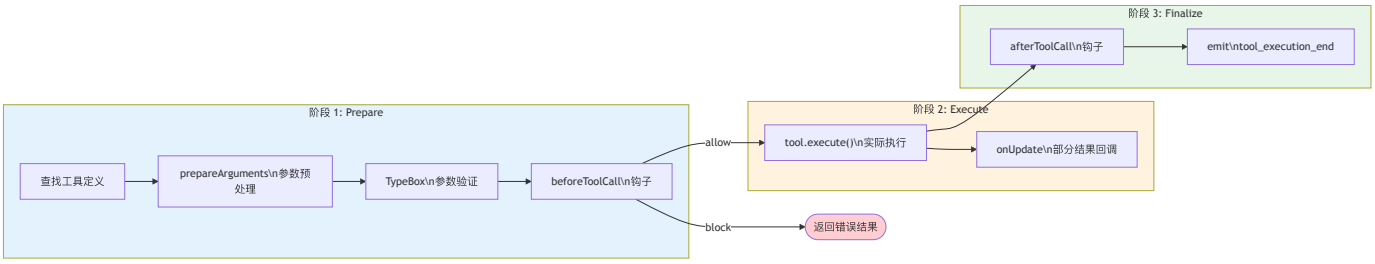
为什么需要三阶段？因为工具执行面临三个现实问题：

1. **LLM 会犯错**。模型可能调用一个不存在的工具，传入格式错误的参数，甚至调用被禁止的工具。这些错误必须在执行之前被拦截。
2. **执行过程需要被观测**。安全审计、权限控制、速率限制 — 这些横切关注点需要在执行前后有插入点。
3. **多个工具调用的执行策略不同**。LLM 一次可能返回多个工具调用，串行还是并行执行，不同场景有不同的最优策略。

三阶段管道为每个问题提供了解决位置。

三阶段管道

让我们用一张图看清整条管道：



阶段 1：Prepare — 在执行之前把一切验证完

`prepareToolCall()` 是第一阶段的入口。它做三件事，任何一件失败都会产生一个 `immediate` 结果（跳过执行，直接返回错误）：

```
// packages/agent/src/agent-loop.ts:472-522（简化）

async function prepareToolCall(
  currentContext, assistantMessage, toolCall, config, signal
): Promise<PreparedToolCall | ImmediateToolCallOutcome> {
  // 1. 查找工具 — 模型调了个不存在的工具?
  const tool = currentContext.tools?.find(t => t.name ===
toolCall.name);
  if (!tool) {
    return {
      kind: "immediate",
      result: createErrorToolResult(
        `Tool ${toolCall.name} not found`
      ),
      isError: true,
    };
  }

  try {
    // 2. 参数验证 — 模型传了错误的参数格式?
    const preparedToolCall = prepareToolCallArguments(tool, toolCall);
    const validatedArgs = validateToolArguments(tool, preparedToolCall);

    // 3. beforeToolCall 钩子 — 上层说不许执行?
    if (config.beforeToolCall) {
      const beforeResult = await config.beforeToolCall(
        { assistantMessage, toolCall, args: validatedArgs, context },
        signal,
      );
      if (beforeResult?.block) {
        return {
          kind: "immediate",
          result: createErrorToolResult(
            beforeResult.reason || "Tool execution was blocked"
          ),
          isError: true,
        };
      }
    }

    // 全部通过，返回"已准备好"的工具调用
    return { kind: "prepared", toolCall, tool, args: validatedArgs };
  } catch (error) {
    // 验证失败或钩子抛异常 → 转为 immediate 错误结果
```

```
      return {
        kind: "immediate",
        result: createErrorToolResult(
          error instanceof Error ? error.message : String(error)
        ),
        isError: true,
      };
    }
  }
```

注意整个 prepare 体被 try-catch 包裹。这保证了不论是 `validateToolArguments` 抛出验证错误、还是 `beforeToolCall` 钩子意外崩溃，结果都是一个 `immediate` 错误 — 循环不会中断。

返回类型的设计也值得关注： `PreparedToolCall | ImmediateToolCallOutcome` 。这是一个判别联合（discriminated union），用 `kind` 字段区分两种情况：

- `kind: "prepared"` — 验证通过，可以执行
- `kind: "immediate"` — 验证失败，直接返回错误结果，跳过执行阶段

这个设计让调用者不需要 try-catch。检查 `kind` 就知道下一步该做什么。

`prepareArguments` 的用途：有些工具需要在验证之前预处理参数。比如 edit 工具需要把旧版 API 的 `oldText` / `newText` 顶层字段转换成新的 `edits[]` 数组格式。这个钩子允许工具自己做参数兼容性处理。

`validateToolArguments` 的作用：使用 TypeBox schema 做运行时验证。如果模型传了 `{ path: 123 }` 而 schema 要求 `path` 是 string，验证会失败，循环返回一个清晰的错误信息给模型，模型有机会在下一轮修正。

`beforeToolCall` 钩子的位置选择：注意它在参数验证**之后**。这意味着钩子拿到的 `args` 是已验证的、类型安全的。钩子不需要自己做参数验证。同时，钩子可以访问完整的 `context` （包括之前的消息历史），这让基于上下文的安全策略成为可能 — 比如"如果模型在最近 3 轮内已经修改了 5 个文件，阻止进一步的写操作"。

阶段 2：Execute — 实际运行工具

只有通过 prepare 阶段的工具调用才会进入 execute：

```
// packages/agent/src/agent-loop.ts:524-559（简化）

async function executePreparedToolCall(
  prepared: PreparedToolCall,
  signal: AbortSignal | undefined,
  emit: AgentEventSink,
): Promise<ExecutedToolCallOutcome> {
  const updateEvents: Promise<void>[] = [];

  try {
    const result = await prepared.tool.execute(
      prepared.toolCall.id,
      prepared.args,
      signal,
      (partialResult) => {
        updateEvents.push(
          Promise.resolve(emit({
            type: "tool_execution_update",
            toolCallId: prepared.toolCall.id,
            toolName: prepared.toolCall.name,
            args: prepared.toolCall.arguments,
            partialResult,
          }))
        );
      },
    );
    await Promise.all(updateEvents);
    return { result, isError: false };
  } catch (error) {
    await Promise.all(updateEvents);
    return {
      result: createErrorToolResult(
        error instanceof Error ? error.message : String(error)
      ),
      isError: true,
    };
  }
}
```

这个阶段相对直接，但有两个设计细节值得注意：

1. `onUpdate` 回调。工具执行可能是长时间运行的（比如一个 bash 命令）。`onUpdate` 让工具可以在执行过程中发射部分结果，这些部分结果通过事件流传递给 UI，让用户看到实

时进度。注意 `updateEvents` 数组 — 所有 update 事件的 Promise 都被收集起来，在工具执行完成后 `await Promise.all(updateEvents)` 确保所有事件都被发射完毕。

2. 这里允许 try-catch。和 `AgentLoopConfig` 的回调不同（它们要求"must not throw"），工具的 `execute` 方法是**允许抛异常的**。`AgentTool` 接口的文档明确说明："Throw on failure instead of encoding errors in content." 循环引擎会捕获异常并转换为 `isError: true` 的结果。

为什么工具允许抛异常？因为工具是外部代码 — 它可能是用户通过 extension 注册的，pi 不能假设它会正确处理所有错误。循环引擎对工具做了 try-catch 兜底。同样，`beforeToolCall` 和 `afterToolCall` 钩子也被 try-catch 保护，因为它们也可能来自 extension 代码。

相比之下，`convertToLlm`、`transformContext` 等消息管道回调有明确的"must not throw"契约（详见第 8 章），因为它们是系统内部代码，循环引擎不对它们做防御性捕获。

阶段 3：Finalize — 执行后审计和修改

```
// packages/agent/src/agent-loop.ts:561-595（简化）

async function finalizeExecutedToolCall(
  currentContext, assistantMessage, prepared, executed,
  config, signal, emit,
): Promise<ToolResultMessage> {
  let result = executed.result;
  let isError = executed.isError;

  // afterToolCall 钩子：可以修改结果
  if (config.afterToolCall) {
    const afterResult = await config.afterToolCall(
      { assistantMessage, toolCall: prepared.toolCall,
        args: prepared.args, result, isError, context },
      signal,
    );
    if (afterResult) {
      result = {
        content: afterResult.content ?? result.content,
        details: afterResult.details ?? result.details,
      };
      isError = afterResult.isError ?? isError;
    }
  }

  return emitToolCallOutcome(prepared.toolCall, result, isError, emit);
}
```

`afterToolCall` 钩子的设计非常精妙。它的返回值是**部分覆盖**语义：

- 返回 `undefined` → 不修改任何东西
- 返回 `{ content: [...] }` → 只替换 content，保留原来的 details 和 isError
- 返回 `{ isError: false }` → 只把错误标记改为成功，保留原来的 content 和 details
- 返回 `{ terminate: true }` → 只覆盖"提前停止"提示（`types.ts:80`），不动其他字段；配合第 8 章的整批 terminate 判断，让钩子也能在工具结算后请求 agent 停下

这种"字段级覆盖"设计让钩子可以做非常精确的修改：

- **安全审计**：记录工具调用日志，但不修改结果（返回 undefined）

- **敏感信息脱敏**：替换 content 中的 API key、密码等（返回 `{ content: [...] }`）
- **错误降级**：某些工具的"错误"其实是预期的（比如 grep 没找到匹配），改 isError 为 false
- **结果增强**：在 details 中注入额外元数据供 UI 展示

工具结果还能动态引入新工具。除了 content / details / terminate，`AgentToolResult` 上还有一个字段 `addedToolNames?: string[]`（`types.ts:363`）：一个工具在返回结果时，可以声明"从这条转录点起，这些新工具变得可用"。finalize 阶段把它原样传播到最终写入上下文的 `ToolResultMessage` 上（`agent-loop.ts:783`，仅在非空时带上该字段）：

```
// packages/agent/src/agent-loop.ts:773-786（简化）
function createToolResultMessage(finalized): ToolResultMessage {
  return {
    role: "toolResult",
    toolCallId: finalized.toolCall.id,
    toolName: finalized.toolCall.name,
    content: finalized.result.content ?? [],
    details: finalized.result.details,
    ...(finalized.result.addedToolNames?.length
      ? { addedToolNames: finalized.result.addedToolNames }   // ← 传播
      : {}),
    isError: finalized.isError,
    timestamp: Date.now(),
  };
}
```

这就打通了一条**消息锚定的工具加载**（message-anchored tool loading，v0.80.7）通道：某个工具的执行结果本身可以"解锁"一批后续工具，而且解锁点被钉在具体的转录位置上 —— 从这条 `ToolResultMessage` 往后的 turn 才看得到这些新工具，转录回放时也能在同一个锚点复现同样的可用工具集。典型场景是一个"进入某子系统 / 加载某 skill"的工具，执行后才把该子系统专属的工具暴露给模型，而不必一开始就把全部工具塞进 context。注意 pi 在这里只负责**传播这个字段**；真正据此把工具加进 active 集合是上层（工具来源管理方）的职责。

Parallel vs Sequential：两种执行策略

当 LLM 一次返回多个工具调用时，pi 提供了两种执行策略：

```
// packages/agent/src/agent-loop.ts:373-388

async function executeToolCalls(...) {
  // 批次中任一工具自带 executionMode: "sequential"，整批就降级串行
  const hasSequentialToolCall = toolCalls.some(
    (tc) => currentContext.tools
      ?.find((t) => t.name === tc.name)?.executionMode === "sequential",
  );
  if (config.toolExecution === "sequential" || hasSequentialToolCall) {
    return executeToolCallsSequential(...);
  }
  return executeToolCallsParallel(...);  // 默认
}
```

注意执行策略**不再只由 `config.toolExecution` 这一个全局开关决定**。`AgentTool` 新增了一个 per-tool 的 `executionMode` 字段（`types.ts:388`）：单个工具可以声明自己"必须串行"。分流规则是 `config.toolExecution === "sequential"` **或**批次中任一工具的 `executionMode === "sequential"` → 整批走串行。

这个"混批含一个 sequential 工具就整批降级串行"的规则很关键：它让一个有副作用、不能与他人并发的工具（比如一个会改全局状态的工具）有办法**强制**整批串行，而不必要求产品层把全局执行模式也切成 sequential。代价是粒度较粗 —— 一个 sequential 工具会拖慢同批的其他本可并行的工具。

两种策略的差异不只是"串行 vs 并行"那么简单。让我们对比它们的执行时序：

Sequential 模式：每个工具调用独立完成整条管道（prepare → execute → finalize），然后才开始下一个。简单、可预测、但慢。

Parallel 模式的设计更微妙。它**不是**完全并行的：

1. **Prepare 阶段串行**。所有工具调用按源顺序通过 prepare，包括参数验证和 `beforeToolCall` 钩子（这些字段都定义在第 8 章介绍的 `AgentLoopConfig` 中），`tool_execution_start` 也在这一遍按源顺序发射。串行 prepare 保证了钩子调用的**确定性顺序** — 钩子可以基于"第几个工具调用"做决策（比如"同一个 turn 中最多允许 3 次文件写操作"），而不需要担心并发导致的非确定性。

2. **Execute + Finalize 阶段并行**。所有通过 prepare 的工具调用并发跑完 execute → finalize（含 `afterToolCall`），**每个工具一完成就立即发射自己的 `tool_execution_end`** —— 这一串生命周期事件因此按**完成顺序**到达订阅者，而不是源顺序。
3. **持久化的 tool-result 工件按源顺序**。所有工具结算完后，循环再按 LLM 返回的源顺序重新拼装 `ToolResultMessage[]`（并按源顺序发射 tool-result 消息事件），保证写进上下文的工具结果顺序是确定的。

这里有一个**容易被旧版文档误导的细节**：早期实现里 finalize 是一个 `for...await` 串行循环，`afterToolCall` 确实严格按源顺序运行。v0.68.1 起改为并发结算（下方代码的 `Promise.all`），生命周期事件与持久化工件的顺序**被拆成了两套**：

```
// packages/agent/src/agent-loop.ts:451-517（简化）

async function executeToolCallsParallel(...) {
  const finalizedCalls: FinalizedToolCallEntry[] = [];

  // 串行 prepare：按源顺序发 tool_execution_start
  for (const toolCall of toolCalls) {
    await emit({ type: "tool_execution_start", ... });
    const preparation = await prepareToolCall(...);
    if (preparation.kind === "immediate") {
      await emitToolExecutionEnd(finalized, emit);   // 立即结算
      finalizedCalls.push(finalized);
    } else {
      // 推入一个 thunk：execute → finalize → 发 tool_execution_end
      finalizedCalls.push(async () => { ... });
    }
  }

  // 并发跑完所有 thunk —— tool_execution_end 按【完成顺序】发射
  const ordered = await Promise.all(
    finalizedCalls.map((e) => typeof e === "function" ? e() : e)
  );

  // 再按【源顺序】拼装并发射 tool-result 消息工件
  const messages: ToolResultMessage[] = [];
  for (const finalized of ordered) {
    const msg = createToolResultMessage(finalized);
    await emitToolResultMessage(msg, emit);
    messages.push(msg);
  }
  return { messages, terminate: shouldTerminateToolBatch(ordered) };
}
```

为什么要把"事件"和"工件"拆成两套顺序?

因为它们服务两种不同的消费者。**生命周期事件**（`tool_execution_end`）服务 UI —— 哪个工具先跑完，就先把它的结果渲染出来，用户能尽早看到进展，没必要为了排版让先完成的工具干等。**持久化工件**（`ToolResultMessage[]`）服务的是上下文与回放 —— 写进 `AgentContext`、最终喂给 LLM 的工具结果必须有确定顺序（与源顺序一致），否则同样的输入会产生不同的转录，破坏可重放性。一句话：**对人眼用完成顺序，对模型和存储用源顺序。**

取舍分析

三阶段管道的收益

1. 钩子增加了系统的可观测性和可控性。 `beforeToolCall` 让产品层可以实现权限弹窗（"允许执行 bash？"）、速率限制、安全策略。 `afterToolCall` 让产品层可以实现审计日志、敏感信息脱敏、结果增强。这些横切关注点不需要修改循环引擎的代码。

2. 参数验证把模型的错误变成可恢复的对话。当 TypeBox 验证失败时，循环把清晰的错误信息作为工具结果返回给模型。模型在下一轮可以修正参数重试。如果没有验证，错误的参数会导致工具内部崩溃，产生不可恢复的失败。

3. parallel 模式的"prepare 串行 + execute 并行"兼顾了安全和性能。串行 prepare 保证安全检查的一致性，并行 execute 减少了多工具调用的等待时间。

三阶段管道的代价

1. 当提供了钩子时，每次工具调用都有额外的异步开销。循环引擎会先检查 `config.beforeToolCall` 是否存在，存在才 `await` 调用。所以不提供钩子时没有开销。但一旦提供了钩子 — 即使它每次都返回 undefined（不做任何修改）— `await` 的异步调度成本就会叠加。对于高频、低延迟的工具调用场景，这个开销值得注意。

2. 钩子异常被 try-catch 防御，但代价是静默失败。与第 8 章介绍的消息变换管道回调不同（它们有明确的"must not throw"契约）， `beforeToolCall` 和 `afterToolCall` 被循环引擎的 try-catch 包裹。钩子抛异常不会终止循环，但异常会被转换为工具错误结果 — 这意味着钩子的 bug 可能表现为"工具莫名失败"，而不是清晰的错误信息。

3. parallel 模式下，工具之间无法共享中间状态。并行执行的工具彼此独立，无法看到对方的执行结果。如果两个工具调用之间有依赖关系（比如"先读文件再编辑"），parallel 模式会产生竞态。pi 的解决方案是让 LLM 自己管理依赖 — 如果两个操作有依赖，LLM 应该在两个不同的 turn 中分别发起。

一个具体场景：读文件 + 编辑文件

让我们用一个真实场景把三阶段管道串起来。假设 LLM 在一个 turn 中同时返回了两个工具调用： `read` 和 `edit` （parallel 模式）。

Prepare 阶段（串行）：

1. 循环 emit `tool_execution_start` for `read` — 注意这发生在 prepare **之前**，所以即使工具最终验证失败，UI 也能看到"正在准备"的状态
2. `read` 通过 prepare：工具存在 → TypeBox 验证 path 是 string → `beforeToolCall` 检查文件权限 → 返回 `kind: "prepared"`
3. 循环 emit `tool_execution_start` for `edit`
4. `edit` 通过 prepare：工具存在 → TypeBox 验证 edits 数组格式 → `beforeToolCall` 检查不在禁止列表 → 返回 `kind: "prepared"`

Execute + Finalize 阶段（并发，事件按完成顺序）： 5. `read` 和 `edit` 的 execute → finalize 并发跑 6. `read` 先完成（只是读磁盘）→ 立即 finalize（`afterToolCall` 脱敏文件内容）→ 先发 `tool_execution_end(read)` 7. `edit` 后完成（要写磁盘）→ finalize（`afterToolCall` 记录审计日志）→ 后发 `tool_execution_end(edit)` 8. UI 因此先看到 read 的结果、后看到 edit 的结果 —— 按**完成顺序**

拼装 tool-result 工件（按源顺序）： 9. 两者都结算后，循环按源顺序 `[read, edit]` 重新拼装 `ToolResultMessage[]` 并发射 tool-result 消息事件 —— 写进上下文、最终喂给 LLM 的顺序与 LLM 调用顺序一致

注意一个微妙之处：如果两个工具调用中，`read` 的 prepare 失败了（比如 `beforeToolCall` 阻止了它），它会在串行 prepare 循环中**立即**结算（emit `tool_execution_end`）并作为一个 immediate 项留在源位置。等所有 thunk 并发跑完，最终拼装的工件顺序仍是源顺序：`read` → `edit`。

这个设计的核心判断是：**工具执行不是一个独立的子系统，而是循环的有机组成部分。**三阶段管道让循环引擎在不增加核心复杂度的前提下，为上层提供了精确的控制点。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，并已对照 v0.82.1。三阶段管道结构稳定。`parallel` 执行模式作为默认策略引入，取代了最初的纯 `sequential` 模式；v0.79 起 `AgentTool` 新增 per-tool `executionMode`，批次中任一工具声明 `"sequential"` 即令整批降级串行。v0.68.1 起 parallel 模式的 execute+finalize 改为并发结算：生命周期事件 `tool_execution_end` 按**完成顺序**发射，而持久化的 tool-result 工件仍按 **源顺序**拼装（旧版"finalize 严格按源顺序"的描述已不适用）。v0.69.0 起 `executeToolCalls` 返回 `{ messages, terminate }`，工具可通过整批 `terminate` 提示请求 agent 停止。`prepareArguments` 钩子为支持 edit 工具

> API 演进而添加。v0.80.7 起 `AgentToolResult.addedToolNames` 会传播到 `ToolResultMessage`（`agent-loop.ts:783`），支持消息锚定的动态工具引入：工具结果可从该转录点起解锁一批后续工具。

第 10 章：Agent — 循环之上的有状态壳

> **定位**：本章解析为什么一个无状态循环引擎之上还需要一个有状态的 `Agent` 类。 前置依赖：第 8 章（agentLoop）、第 9 章（工具执行管道）。 适用场景：当你想理解"循环"和"运行时对象"为什么必须分开，或者想为自己的 agent 系统设计状态管理。

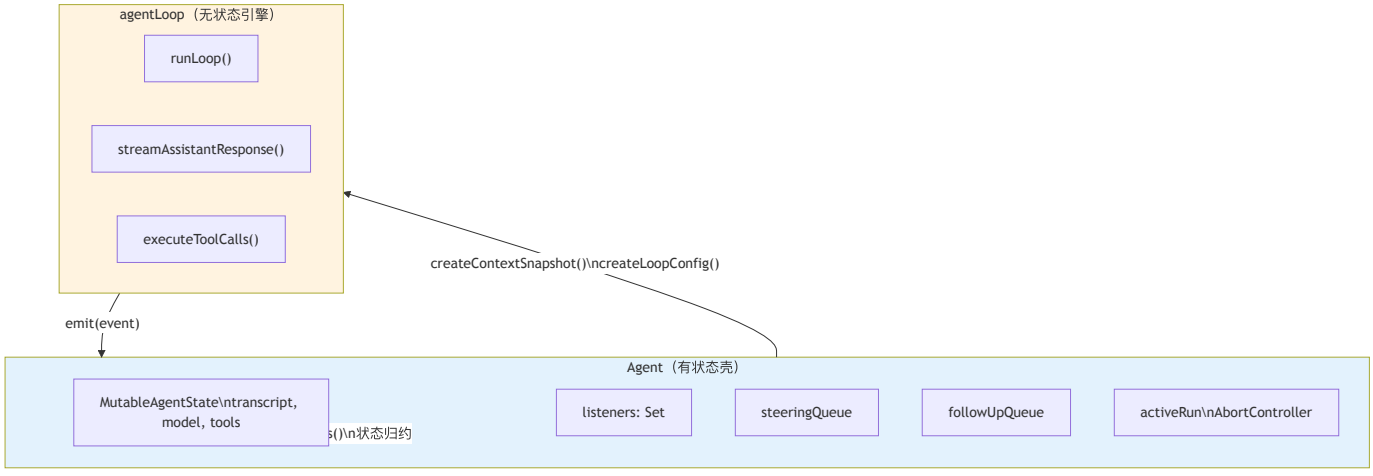
为什么循环引擎不够?

第 8 章展示了 `agentLoop` 的无状态设计。但一个真正可用的 agent 需要更多：

- 它需要**记住**对话历史（transcript）
- 它需要**通知**多个订阅者关于状态变化（listeners）
- 它需要**接收**用户在执行过程中发来的消息（queues）
- 它需要**能被中断**（abort）
- 它需要**防止**同时运行两次（mutual exclusion）

这些都是有状态的需求。如果把它们塞进循环引擎，循环就不再是纯函数了 — 它会变成一个"知道太多"的上帝对象。

pi 的解决方案是在循环引擎之上套一个有状态的壳： `Agent` 类。循环引擎负责"转"， `Agent` 负责"管"。



`Agent` 向循环引擎提供两样东西：一个 context 快照（`createContextSnapshot`）和一个配置对象（`createLoopConfig`）。循环引擎通过事件回调（`emit`）把产出送回 `Agent`，`Agent` 在 `processEvents()` 中做状态归约。

Agent 拥有什么

让我们逐一看 `Agent` 管理的五类状态。

1. MutableAgentState — 受控的可变状态

`Agent` 的核心状态是一个 `MutableAgentState` 对象：

```
// packages/agent/src/agent.ts:57-91（简化）

type MutableAgentState = {
  systemPrompt: string;
  model: Model<any>;
  thinkingLevel: ThinkingLevel;
  // 外部看 readonly，赋值时自动 copy
  get tools(): AgentTool[];
  set tools(next: AgentTool[]);
  get messages(): AgentMessage[];
  set messages(next: AgentMessage[]);
  // 运行时状态
  isStreaming: boolean;
  streamingMessage?: AgentMessage;
  pendingToolCalls: Set<string>;
  errorMessage?: string;
};
```

这里有一个精巧的设计：`tools` 和 `messages` 使用 getter/setter 属性。当你赋值 `state.messages = newArray` 时，setter 会自动调用 `newArray.slice()` — 它总是存储一个副本。

```
// packages/agent/src/agent.ts:80-85

get messages() {
  return messages;
},
set messages(nextMessages: AgentMessage[]) {
  messages = nextMessages.slice();  // ← 总是 copy
},
```

为什么要这样做？因为 `AgentMessage[]` 会被传递给循环引擎（`createContextSnapshot`）。如果不 copy，循环引擎修改数组时会直接影响 `Agent` 的状态，两者的状态就耦合了。copy-on-assign 保证了 `Agent` 的状态和循环引擎的工作数据是独立的。

同时，`isStreaming`、`streamingMessage`、`pendingToolCalls`、`errorMessage` 这四个字段在公开的 `AgentState` 接口中是 `readonly` 的：

```
// packages/agent/src/types.ts:253-278

interface AgentState {
  // ... 可读写字段 ...
  readonly isStreaming: boolean;
  readonly streamingMessage?: AgentMessage;
  readonly pendingToolCalls: ReadonlySet<string>;
  readonly errorMessage?: string;
}
```

外部代码（UI 组件、extension）通过 `agent.state` 读取这些字段，但不能直接修改它们。只有 `Agent` 内部的 `processEvents()` 可以修改。这保证了运行时状态的单一真相源。

2. 事件订阅 — 有序且受信号保护

```
// packages/agent/src/agent.ts:159-219

private readonly listeners = new Set<
  (event: AgentEvent, signal: AbortSignal) => Promise<void> | void
>();

subscribe(listener): () => void {
  this.listeners.add(listener);
  return () => this.listeners.delete(listener);
}
```

订阅模式的三个设计选择：

1. listener 接收 AbortSignal。每个 listener 都能感知当前 run 的中止信号。如果一个 listener 在处理事件时发现 `signal.aborted`，它可以选择跳过耗时操作（比如持久化）。

2. listener 的 Promise 被 await。`processEvents` 中的代码是：

```
for (const listener of this.listeners) {
  await listener(event, signal);
}
```

这意味着 listener 按注册顺序串行执行。一个慢的 listener 会阻塞后续 listener。这是故意的 — 它保证了状态归约和事件通知的顺序一致性。如果 listener 并行执行，两个 listener 可能同时读到不一致的中间状态。

3. `agent_end` 不等于 idle。`agent_end` 事件只意味着循环引擎不再发射事件了。但 `Agent` 要等到**所有 listener 处理完 `agent_end`** 后才算真正 idle。这就是 `waitForIdle()` 和 `agent_end` 的区别：

```
// packages/agent/src/agent.ts:293-295

waitForIdle(): Promise<void> {
  return this.activeRun?.promise ?? Promise.resolve();
}
```

`activeRun.promise` 在 `finishRun()` 中 resolve，而 `finishRun()` 在所有 listener 处理完毕之后才被调用。

3. 消息队列 — 两种节奏的输入

```
// packages/agent/src/agent.ts:112-143

class PendingMessageQueue {
  private messages: AgentMessage[] = [];

  constructor(public mode: QueueMode) {}

  enqueue(message: AgentMessage): void {
    this.messages.push(message);
  }

  drain(): AgentMessage[] {
    if (this.mode === "all") {
      const drained = this.messages.slice();
      this.messages = [];
      return drained;
    }
    // "one-at-a-time" 模式
    const first = this.messages[0];
    if (!first) return [];
    this.messages = this.messages.slice(1);
    return [first];
  }
}
```

`Agent` 持有两个独立的消息队列：

- `steeringQueue`： `agent.steer(msg)` 入队，在 turn 间隙被消费
- `followUpQueue`： `agent.followUp(msg)` 入队，在 agent 本来要退出时被消费

每个队列有两种 drain 模式：

- `"all"`： 一次性取出所有排队的消息
- `"one-at-a-time"` （默认）：一次只取一条，剩余的留到下次

为什么默认是 `one-at-a-time`？考虑这个场景：用户在 agent 执行 bash 命令时快速输入了三条 steering 消息。如果用 `"all"` 模式，三条消息会同时注入 context，LLM 需要一次理解三条指令。如果用 `"one-at-a-time"`，LLM 先处理第一条，在下一个 turn 间隙再收到第二条 — 就像人类对话中逐条回应，而不是一次性面对一堆请求。

队列和循环引擎的对接发生在 `createLoopConfig()` 中：

```
// packages/agent/src/agent.ts:407-431（简化）

private createLoopConfig(): AgentLoopConfig {
  return {
    // ... 其他字段 ...
    getSteeringMessages: async () => {
      return this.steeringQueue.drain();
    },
    getFollowUpMessages: async () => {
      return this.followUpQueue.drain();
    },
  };
}
```

`Agent` 把队列的 `drain()` 方法包装成循环引擎需要的 `getSteeringMessages` 和 `getFollowUpMessages` 回调。循环引擎不知道消息从哪来 — 它只管调用回调取消息。

4. 中止控制 — 一个 AbortController 管全局

```
// packages/agent/src/agent.ts:434-457（简化）

private async runWithLifecycle(
  executor: (signal: AbortSignal) => Promise<void>
): Promise<void> {
  const abortController = new AbortController();
  let resolvePromise = () => {};
  const promise = new Promise<void>((resolve) => {
    resolvePromise = resolve;
  });
  this.activeRun = { promise, resolve: resolvePromise, abortController
};

  this._state.isStreaming = true;
  try {
    await executor(abortController.signal);
  } catch (error) {
    // 安全网：即使循环违反了"must not throw"契约，
    // Agent 也能合成一条失败消息而不是崩溃
    await this.handleRunFailure(
      error, abortController.signal.aborted
    );
  } finally {
    this.finishRun();
  }
}
```

`Agent` 还提供了 `continue()` 方法，它调用循环引擎的 `agentLoopContinue` — 从当前 transcript 继续，而不添加新的 prompt。当最后一条消息是 assistant 角色时，`continue()` 会先尝试排空 steering 队列或 follow-up 队列作为新的 prompt。这是 Agent 和循环引擎之间的一个精巧协调。

每次 `prompt()` 或 `continue()` 调用都会创建一个新的 `AbortController`。它的 signal 被传递给循环引擎、所有 listener、所有工具执行。当用户调用 `agent.abort()` 时：

```
abort(): void {
  this.activeRun?.abortController.abort();
}
```

一个 `abort()` 调用就能中止整条链：LLM 流式响应被取消 → 工具执行被中止 → 循环退出。

5. 互斥锁 — 禁止重入

```
async prompt(input): Promise<void> {
  if (this.activeRun) {
    throw new Error(
      "Agent is already processing a prompt. " +
      "Use steer() or followUp() to queue messages, " +
      "or wait for completion."
    );
  }
  // ...
}
```

`Agent` 通过检查 `activeRun` 来防止同时运行两个循环。这不是用 Mutex 实现的，而是一个简单的存在性检查 — 如果 `activeRun` 存在，说明有循环在跑，新的 `prompt()` 调用会抛异常。

注意错误信息的设计：它不只是说"不行"，还告诉调用者**应该怎么做** — "Use steer() or followUp() to queue messages, or wait for completion." 错误信息本身就是 API 文档。

processEvents：状态归约器

`Agent` 接收循环引擎发射的事件，并在 `processEvents()` 中做状态归约。这个方法的逻辑类似 Redux 的 reducer — 给定当前状态和一个事件，更新状态 — 但不同于 Redux 的纯函数语义，这里是直接 mutation：

```
// packages/agent/src/agent.ts:491-538（简化，省略了 signal 空值保护）

private async processEvents(event: AgentEvent): Promise<void> {
  switch (event.type) {
    case "message_start":
      this._state.streamingMessage = event.message;
      break;

    case "message_update":
      this._state.streamingMessage = event.message;
      break;

    case "message_end":
      this._state.streamingMessage = undefined;
      this._state.messages.push(event.message);
      break;

    case "tool_execution_start": {
      const pendingToolCalls = new Set(this._state.pendingToolCalls);
      pendingToolCalls.add(event.toolCallId);
      this._state.pendingToolCalls = pendingToolCalls;
      break;
    }

    case "tool_execution_end": {
      const pendingToolCalls = new Set(this._state.pendingToolCalls);
      pendingToolCalls.delete(event.toolCallId);
      this._state.pendingToolCalls = pendingToolCalls;
      break;
    }

    case "turn_end":
      if (event.message.role === "assistant"
        && event.message.errorMessage) {
        this._state.errorMessage = event.message.errorMessage;
      }
      break;

    case "agent_end":
      this._state.streamingMessage = undefined;
      break;
  }

  // 先归约状态，再通知 listener
```

```
    // 实际代码中还有 signal 空值保护：
    // if (!signal) throw new Error("listener invoked outside active run")
    const signal = this.activeRun?.abortController.signal;
    for (const listener of this.listeners) {
      await listener(event, signal);
    }
  }
```

几个值得注意的设计细节：

1. `pendingToolCalls` 每次修改都创建新 Set。 `tool_execution_start` 和 `tool_execution_end` 不是在原 Set 上 add/delete，而是创建一个新的 Set 再赋值。这是因为 `AgentState.pendingToolCalls` 是 `ReadonlySet` — 外部代码持有的引用不会被意外修改。新建 Set 保证了不可变语义。

2. 状态归约在 listener 通知之前。 `switch` 语句先更新状态，然后才 `for` 循环通知 listener。这意味着 listener 在收到 `message_end` 事件时，`state.messages` 已经包含了这条消息，`state.streamingMessage` 已经被清空。listener 总是看到一致的状态。

3. 不是所有事件都有状态变更。 `agent_start`、`turn_start`、`tool_execution_update` 都没有对应的状态修改 — 它们只被透传给 listener。归约器只处理真正影响状态的事件。

CustomAgentMessages：类型安全的扩展点

`Agent` 管理的 `messages` 数组的类型是 `AgentMessage[]`。这个类型的定义隐藏了一个精妙的扩展机制：

```
// packages/agent/src/types.ts:236-245

// 默认为空 — 应用通过声明合并扩展
export interface CustomAgentMessages {
  // Empty by default
}

// AgentMessage = LLM 消息 + 所有自定义消息
export type AgentMessage =
  Message | CustomAgentMessages[keyof CustomAgentMessages];
```

`CustomAgentMessages` 是一个空接口，但它使用了 TypeScript 的声明合并（declaration merging）。应用层可以这样扩展它：

```
// 在 pi-coding-agent 中
declare module "@earendil-works/pi-agent-core" {
  interface CustomAgentMessages {
    custom: CustomMessage;        // compaction 摘要、分支标记等
    bashExecution: BashMessage;   // bash 工具的结构化结果
  }
}
```

扩展之后，`AgentMessage` 自动变成：

```
type AgentMessage =
  | Message           // user, assistant, toolResult
  | CustomMessage     // compaction, branch, notification
  | BashMessage;      // bash 结构化结果
```

为什么不用普通的联合类型?

如果把自定义消息硬编码到联合类型里，pi-agent-core 就需要知道 pi-coding-agent 的消息类型 — 依赖方向反了。声明合并让 pi-agent-core 定义框架（空的 `CustomAgentMessages`），pi-coding-agent 填充内容，依赖方向保持正确。

为什么不用 `any` 或泛型?

用 `any` 会丢失类型安全。用泛型 `Agent<TMessage>` 会让每个使用 Agent 的地方都要传类型参数。声明合并在全局生效，不需要传递类型参数，所有使用 `AgentMessage` 的地方自动包含自定义类型。

这和 `convertToLlm` 回调配合形成完整的设计：自定义消息在循环内部是一等公民（类型安全、可以被 `transformContext` 处理），在出门见 LLM 时被 `convertToLlm` 过滤掉。类型系统保证你不会忘记处理某种自定义消息。

Agent 不管什么

理解 `Agent` 的边界和理解它的能力同样重要。以下是 `Agent` 明确不管的事情：

关注点	Agent 的态度	谁管
会话持久化	不知道"会话"的存在	SessionManager（第 11 章）
UI 渲染	只发射事件，不管谁听	TUI / Web UI（第 24 章）
认证	通过 `getApiKey` 回调获取，不管 token 怎么来的	OAuth 模块（第 7 章）
模型选择	只持有一个 `model` 字段，不管怎么选的	ModelRegistry（第 18 章）
Context 压缩	通过 `transformContext` 委托，不管怎么压	Compaction（第 12 章）
System prompt 拼接	只持有一个 `systemPrompt` 字符串	system-prompt.ts（第 14 章）
工具注册	只持有 `tools[]`，不管工具从哪来	Extension（第 15 章）

这张表揭示了一个设计原则：**Agent 只管运行时状态，不管配置和策略。** 它知道自己正在用什么模型（`state.model`），但不知道为什么选这个模型。它知道有哪些工具可用（`state.tools`），但不知道这些工具是怎么被发现和注册的。它知道 system prompt 是什么（`state.systemPrompt`），但不知道 prompt 是怎么从多个来源拼接出来的。

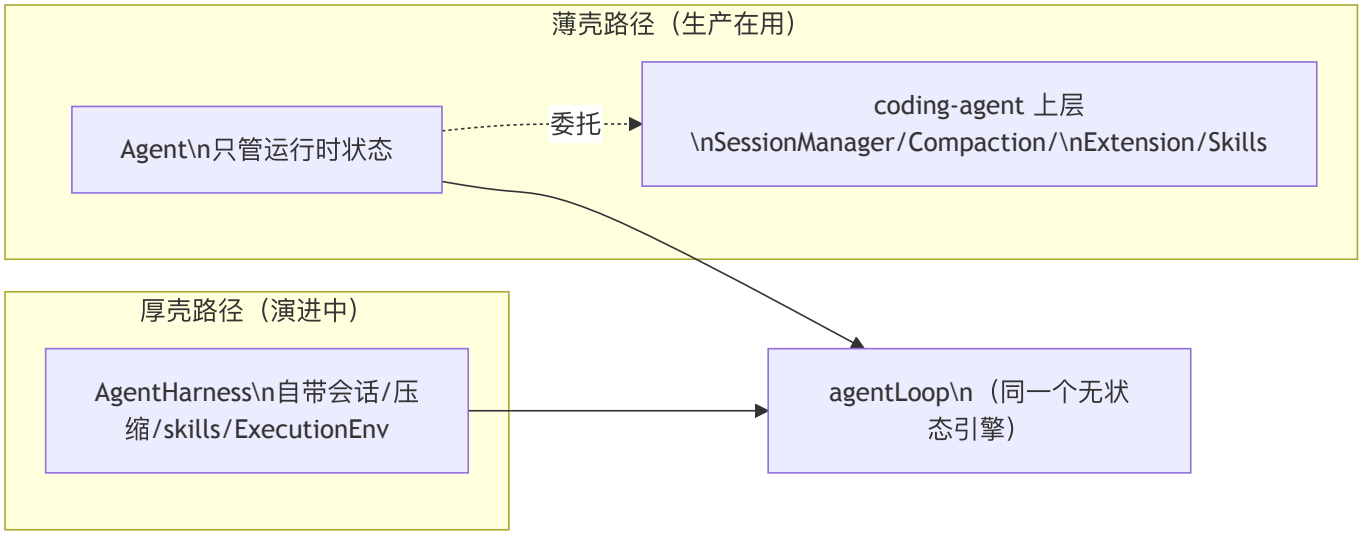
另一条路：AgentHarness 这个"厚壳"

上面那张表的每一行 —— 会话持久化、压缩、skills、工具来源 —— `Agent` 都明确委托给了上层（coding-agent 包）。但 packages/agent 在 v0.79 周期里长出了**第二条路径**：`AgentHarness`（`packages/agent/src/harness/agent-harness.ts:174`），它把那张表里委托出去的东西**反过来收编回了 agent 包内部**。

如果说 `Agent` 是"薄壳 + 上层自己拼装一切"，`AgentHarness` 就是一个"自带电池"的厚壳：

表里委托出去的	AgentHarness 的内聚实现
会话持久化（SessionManager）	两层拆分：底层 SessionStorage（条目/树持久化，harness/types.ts:498）+ 上层 SessionRepo（会话集合生命周期，:528）；实现 JsonlSessionStorage / InMemorySessionStorage、JsonlSessionRepo / InMemorySessionRepo（harness/session/）、session tree、fork()、navigateTree()
Context 压缩（Compaction）	内置 compact() / prepareCompaction + 分支摘要（harness/compaction/），可配 retry?: RetryPolicy 与 retry_* 事件
Skills / Prompt 模板	skill(name) / promptFromTemplate(name, args)（harness/skills.ts、prompt-templates.ts）
工具来源（Extension）	getTools / setTools 与 getActiveTools / setActiveTools 二层模型 + 重名校验；工具类型改为上下文感知的 AgentHarnessTool<TContext>
文件系统 / shell 触达	ExecutionEnv（FileSystem + Shell，harness/types.ts:373）被包装为 ExecutionToolContext（harness/tools/tool-context.ts:4），喂给内置工具 harness/tools/{read,write,edit,bash}.ts；Node 实现见 harness/env/nodejs.ts

它还提供了比 Agent.subscribe() 更丰富的**类型化钩子链** on(type, handler)（agent-harness.ts:1050）—— before_provider_request、after_provider_response、tool_call / tool_result、session_before_compact / session_compact、session_before_tree / session_tree 等 —— 以及一套 Result<T, E> 风格的错误模型（AgentHarnessError 归一化 SessionError / CompactionError / BranchSummaryError，agent-harness.ts:145-151）和显式的运行阶段机 AgentHarnessPhase = "idle" | "turn" | "compaction" | "branch_summary" | "retry"（types.ts:553），用 phase !== "idle" 守卫拒绝重入。



厚壳在 v0.79 之后又经历了几次实质重构，值得单独展开三处，因为它们恰好都是"把厚度收回内核"这个选择必须付的账。

工具模型：从上下文无关的 AgentTool 到 AgentHarnessTool<TContext>

薄壳路径里，工具是**上下文无关**的 —— `AgentTool.execute(id, args, signal, onUpdate)` 拿不到任何应用级依赖，要触达文件系统或 shell 得靠闭包捕获。v0.82.0 的 breaking 重构给厚壳换了一套工具类型 `AgentHarnessTool<TContext>`（`harness/types.ts:99-112`），它在 `AgentTool` 的 `execute` 末尾**多加了一个 `context: TContext` 参数**：

```
// packages/agent/src/harness/types.ts:99-112（简化）
export type AgentHarnessTool<TContext extends object | undefined, ...> =
  Omit<AgentTool<...>, "execute"> & {
    execute(
      toolCallId: string,
      params: Static<TParameters>,
      signal: AbortSignal | undefined,
      onUpdate: AgentToolUpdateCallback<TDetails> | undefined,
      context: TContext,          // ← 每个 turn 快照解析出的应用自定义 context
    ): Promise<AgentToolResult<TDetails>>;
  };
```

这个 `context` 从哪来？厚壳的构造选项里多了一个 `toolContext`（`harness/types.ts:955`）：应用要么给一个静态值，要么给一个零参 provider，由 harness **在每个 turn 的快照时刻解析一次**，再注入当轮所有工具的 `execute`。类型上还

做了纪律约束 —— 当 `TContext` 为 `undefined`（上下文无关的 harness）时 `toolContext?: undefined`，一旦声明了非空 `TContext`，`toolContext` 就变成**必填**（`harness/types.ts:942-956`）。

那原来那套 `ExecutionEnv`（`FileSystem + Shell`）去哪了？它**没有被删**（`harness/types.ts:373` 仍在），而是被**降级成一种具体的 context 形态**：内置工具需要的文件/ shell 触达被收进 `ExecutionToolContext`（`harness/tools/tool-context.ts:4`），其定义就是薄薄一层 `{ env: ExecutionEnv }`。厚壳自带的上下文感知工具 `harness/tools/{read,write,edit,bash}.ts` 正是以这个 `ExecutionToolContext` 为 `TContext`，通过 `execute(..., context)` 的最后一个参数拿到 `env` 去读写磁盘、跑命令。

这个改动的取舍：得到的是**工具依赖显式化** —— 工具不再靠闭包偷偷捕获环境，而是由 harness 在 turn 边界统一注入、可随快照替换（比如切换工作目录 / 沙箱）；付出的是**又一个 breaking 的类型分叉**，厚壳的工具从此和薄壳的 `AgentTool` 不是同一个类型，两条路径的工具不能直接互换。

会话两层：SessionStorage 与 SessionRepo

前面表里把会话持久化一句带过成"`SessionRepo` 抽象"。厚壳的会话其实是**两层**，把"一个会话内部怎么存"和"多个会话怎么管"拆开了。这个两层结构在 v0.80.x 就已成型（v0.80.4 起导出 `Jsonl` / `InMemory` 两套实现），v0.81.0 则重构了它的契约（新增 `getPathToRootOrCompaction`、cursor 式读取，以及 compaction checkpoint 语义）：

- **`SessionStorage`**（`harness/types.ts:498`）是**单会话内的条目/树存储**：`appendEntry` / `getEntry` / 游标式 `getEntries`、`getLeafId` / `setLeafId` 追踪当前叶子、`getSessionName` / `getSessionStats`，以及关键的 `getPathToRootOrCompaction(leafId)`（`:512`）—— 从某个叶子回溯到根**或到最近一次 compaction 为止**，让压缩尾部成为一个自包含的 checkpoint，回放不必每次都拉到最开头。
- **`SessionRepo`**（`harness/types.ts:528`）是**跨会话的集合生命周期**：`create` / `open` / `list` / `delete` / `fork`。`fork` 就落在这一层，配合 storage 的 tree 能力实现分支。

两层各有内存与 JSONL 两套实现：`InMemorySessionStorage` / `JsonlSessionStorage`（v0.80.4 起导出）与 `InMemorySessionRepo` / `JsonlSessionRepo`；JSONL 侧的仓库接口收窄为 `JsonlSessionRepoApi`（`harness/types.ts:550`）。分层的意义在于：换存储介质只需替换 `SessionStorage`（比如未来接数据库），而 `SessionRepo` 的会话管理语义与 `fork` / 树导航逻辑不必跟着改。

压缩重试：RetryPolicy 与三个 retry_* 事件

前面提到厚壳的运行阶段机里有一个 `"retry"` 阶段（`types.ts:553`）。v0.81.1 把它补齐成了一套**可配置的重试策略 + 生命周期事件**：构造选项新增 `retry?: RetryPolicy`（`harness/types.ts:934`，类型来自 pi-ai），专门作用于**生成式的 compaction 与 branch-summary 请求**（这两步要额外调 LLM，是最容易踩到瞬时失败的地方）。重试过程通过三个事件对外可观测：

- `retry_scheduled`（`harness/types.ts:666`）：带 `operation`（`"compaction" | "branch_summary"`）、`attempt` / `maxAttempts` / `delayMs` / `errorMessage`，宣告"第几次重试将在多久后开始、上次为何失败"。
- `retry_attempt_start`（`:675`）：一次重试真正启动。
- `retry_finished`（`:680`）：重试序列结束。

它们和既有的 `"retry"` 阶段呼应 —— 阶段机用 `phase === "retry"` 守卫重入，事件流则让订阅者看到重试的节奏。这正是薄壳"不能自己重试"（第 8 章取舍）在厚壳里被收编的地方：`Agent` 把重试留给上层，`AgentHarness` 则把它做进了内核，代价是又多背了一份策略配置。要注意，这套 harness 内核里的 `RetryPolicy` / `retry_*` 与第 12 章 coding-agent 产品层的 `summarization_retry_*` 是**不同层的同构机制** —— 前者在 agent 内核重试生成式 compaction / branch-summary，后者在产品层重试会话摘要，二者互不依赖、各自成套。

认证收敛：models-only

薄壳 `Agent` 走 `getApiKey` 回调拿密钥（上面那张"Agent 不管什么"的表仍然成立，`agent.ts:102`）。但厚壳这一侧在 v0.80.0 做了 breaking 收敛：**移除了 `getApiKeyAndHeaders`**，认证统一收口到一个必填的 `models: Models`（`harness/types.ts:923`）—— turn 流式、compaction、branch-summary 全部经由 `Models` 集合，认证由各 provider 自己的 auth 解析。`compact()` / `generateSummary()` / `generateBranchSummary()` 也相应改收 `Models`。换句话说，厚壳不再有"裸密钥 + headers"这条老路，只认 provider 运行时对象。**这只影响 `AgentHarness` 一侧；`Agent` 路径的 `getApiKey` 叙述保持不变。**

必须如实说明它的采用状态：截至 v0.82.1，**生产的 coding-agent 仍然用 `new Agent({...})`**（`packages/coding-agent/src/core/sdk.ts:294`），`AgentHarness` 目前只在 agent 包自身的测试与 `docs/{agent-harness,durable-harness,hooks}.md` 里使用，coding-agent 源码中没有任何 `AgentHarness` 引用。所以它是一个**演进中的并行抽象**，不是已经取代 `Agent` 的新默认。

这两条路线本身就是本书主线的一次张力实验： `Agent` 把厚度留给上层、换取内核纯净； `AgentHarness` 把厚度收回内核、换取开箱即用的持久会话与分支能力。两者共享同一个 `agentLoop` 引擎 —— 厚薄之争发生在引擎之上，而引擎自己始终只管转。

取舍分析

得到了什么

1. 清晰的职责边界。循环引擎是纯计算，Agent 是状态管理。两者可以独立演进 — 改循环逻辑不影响状态管理，改状态结构不影响循环逻辑。

2. 可预测的状态变更。所有状态修改都通过 `processEvents()` 这一个入口。想知道某个状态字段什么时候会变？只需要在 `processEvents()` 的 `switch` 语句中搜索。

3. 灵活的消费者模型。 `subscribe()` 让任意多个消费者同时观察 Agent 的行为。TUI 订阅事件来渲染，session manager 订阅事件来持久化，extension 订阅事件来做自定义逻辑 — 它们互不干扰。

放弃了什么

1. Agent 是一个胖接口。 `Agent` 类有 30+ 个公开方法和属性（包括 `subscribe` 、 `prompt` 、 `continue` 、 `steer` 、 `followUp` 、 `abort` 、 `waitForIdle` 、 `reset` 等方法，以及 `convertToLlm` 、 `transformContext` 、 `beforeToolCall` 、 `afterToolCall` 、 `streamFn` 、 `sessionId` 、 `transport` 、 `toolExecution` 等可配置字段）。如果你只想跑一个简单的 agent 循环，直接调用 `runAgentLoop()` 比创建一个 Agent 实例更直接。Agent 的价值在有状态、有交互的场景，对于一次性脚本反而是负担。

2. 状态同步依赖事件顺序。因为 listener 串行执行，一个慢的 listener（比如写磁盘的 session manager）会延迟后续 listener（比如渲染 UI 的 TUI）收到事件的时间。在实践中这通常不是问题（listener 的处理时间远小于 LLM 响应时间），但在极端情况下可能导致 UI 延迟。

3. Agent 是单线程模型。同一时间只能有一个 `prompt()` 或 `continue()` 在运行。这意味着不能实现"后台持续运行、前台随时查询"的模式。如果需要这种模式，必须在 Agent 之上再包一层异步调度器。

版本演化说明

本章核心分析基于 pi-mono v0.66.0，并已对照 v0.82.1。 `Agent` 类的核心结构自引入 以来保持稳定，仍是生产 coding-agent 的运行时壳（ `coding-agent/src/core/sdk.ts:294` ）。 `PendingMessageQueue` 的 `"all" | "one-at-a-time"` 模式是后来的增强， 早期版本只有 `"one-at-a-time"` 行为。 `CustomAgentMessages` 声明合并机制 在 pi-agent-core 从 pi-coding-agent 分离时引入，解决了包间类型依赖问题 （模块名随 scope 迁移为 `@earendil-works/pi-agent-core` ）。v0.79 周期新增的 `AgentHarness` 子系统 （ `packages/agent/src/harness/` ）把会话/压缩/skills/ExecutionEnv 内聚进 agent 包，是与 `Agent` 并存的"厚壳"路径，目前仍仅在 agent 包自身测试/docs 中使用。 该厚壳在 v0.79 之后持续重构：v0.80.0 认证收敛为 models-only（移除 `getApiKeyAndHeaders` ， `models: Models` 必填)；v0.81.0 会话拆成 `SessionStorage + SessionRepo` 两层 （新增 `getPathToRootOrCompaction`)；v0.81.1 为 compaction / branch-summary 补上 `retry?: RetryPolicy` 与 `retry_scheduled` / `retry_attempt_start` / `retry_finished` 事件； v0.82.0 工具模型 breaking 重构为上下文感知的 `AgentHarnessTool<TContext> + toolContext` ， `ExecutionEnv` 被包装为 `ExecutionToolContext` 喂给内置的 read/write/edit/bash 工具。

第 11 章：会话树 — 比"聊天记录"更好的数据模型

> **定位**：本章解析 pi 的会话持久化设计 — 为什么用树而不是列表，为什么用 JSONL 而不是数据库。 前置依赖：第 10 章（Agent 类的状态管理）。 适用场景：当你想理解 coding agent 为什么需要分支和回溯，或者想为自己的 agent 系统设计会话存储。

Coding agent 的会话为什么不是线性的?

这是本章的核心设计问题。

聊天机器人的会话是线性的 — 一问一答，从头到尾。但 coding agent 的工作流本质上是非线性的：

- 用户让 agent 重构一段代码，agent 改了 5 个文件。用户发现改错了，想**回到改之前**，用不同的方式重试
- agent 执行了一个 bash 命令，结果不对。用户想**从那个命令之前**重新开始，换个命令
- 用户在第 20 轮发现第 5 轮的一个决策不对，想**跳回第 5 轮**创建一个新分支

如果用线性列表存储，这些操作要么不可能（无法回溯），要么需要复制整个会话历史（浪费存储）。

pi 的解决方案是**会话树** — 每条消息有一个 `parentId`，指向它的前驱消息。分支就是同一个父节点下的多个子节点。

JSONL + parentId：极简的树形存储

会话文件是一个 JSONL（JSON Lines）文件，每行一个 JSON 对象。第一行是 session header，后续行是 session entries：

```
// packages/coding-agent/src/core/session-manager.ts:29-36

interface SessionHeader {
  type: "session";
  version?: number;  // 当前 v3
  id: string;
  timestamp: string;
  cwd: string;
  parentSession?: string;  // fork 来源
}
```

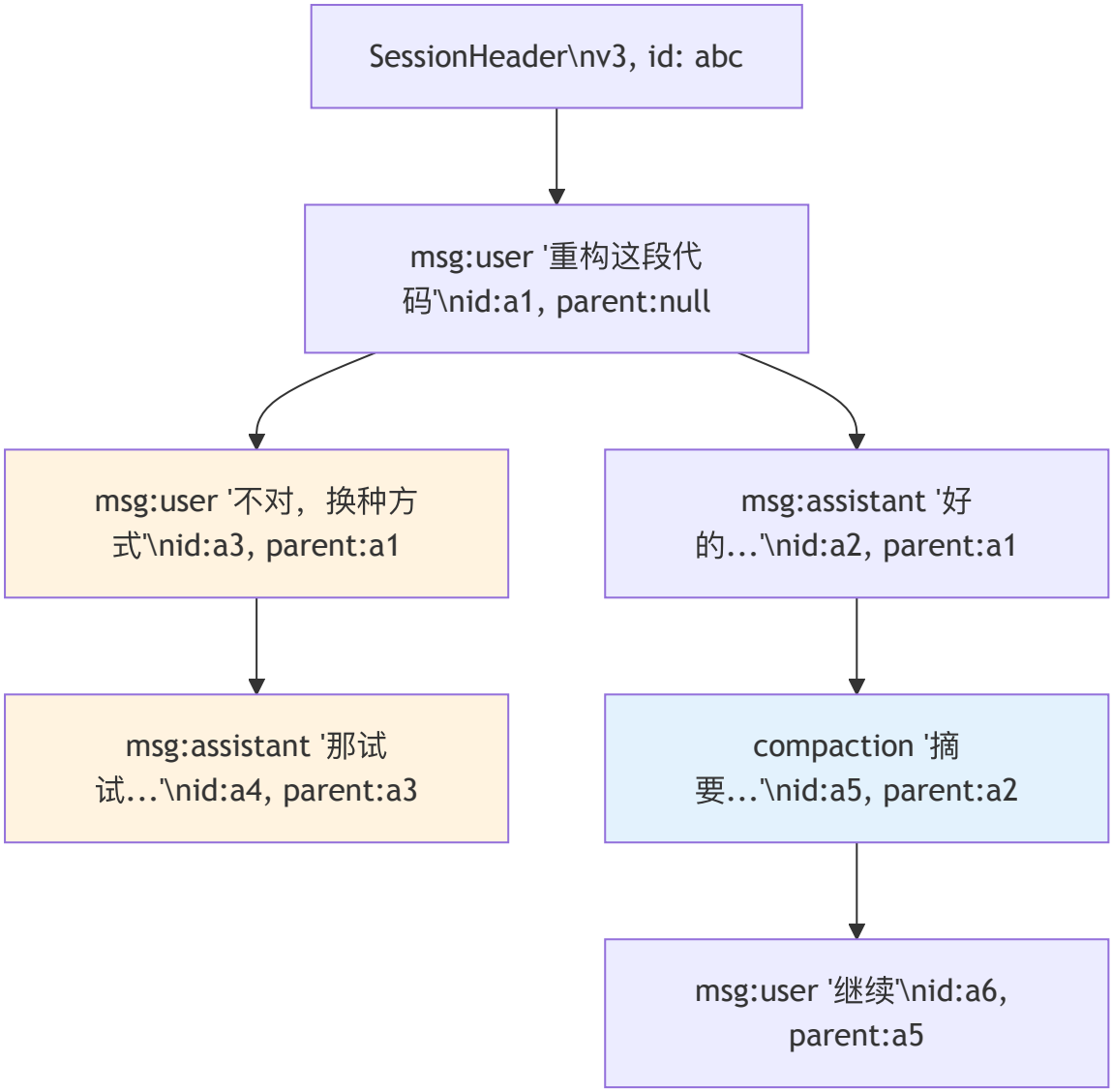
`SessionHeader` 是会话文件的第一行，它不参与树结构，而是记录会话级别的元信息。`version` 字段驱动向后兼容的迁移逻辑（后文详述）。`parentSession` 在用户从一个分支创建新 session 文件时，记录来源 session 的路径，形成 session 之间的溯源链。

每个 entry 都有 `id` 和 `parentId`，形成树形结构：

```
// packages/coding-agent/src/core/session-manager.ts:43-48

interface SessionEntryBase {
  type: string;
  id: string;
  parentId: string | null;  // null = 根节点
  timestamp: string;
}
```

这里要区分两种 id。**entry 的 id** 是 8 位十六进制字符串，由 `randomUUID().slice(0, 8)` 生成（session-manager.ts:218），并通过碰撞检查确保唯一性 —— 它们会出现在 TUI 的分支导航中，短 id 对人类更友好。`parentId` 指向前驱 entry 的 id，`null` 表示这是树的根节点（通常是用户的第一条消息）。

而**会话本身的 id**（session header 里的 `id`）从 v0.67.1 起改用 **UUIDv7**（`createSessionId()` 返回 `uuidv7()`，session-manager.ts:204）。UUIDv7 是时间有序的 —— 这不是为了好看，而是为了让**基于 session 的请求路由具备时间局部性**：同一会话的 id 在时间上聚集，对 provider 端按 id 路由 + prompt 缓存命中更友好。两种 id 各司其职：会话 id 求全局唯一 + 时间有序，entry id 求短小可读。



这张图中， `a3` （"不对，换种方式"）和 `a2` （"好的..."）共享同一个父节点 `a1` 。这就是一个分支点。用户在 `a1` 之后走了两条路。 `a5` 是一个 compaction entry — 它把 `a1→a2` 的对话压缩成摘要， `a6` 从摘要继续。

9 种 Entry 类型

`SessionEntry` 是 9 种类型的联合：

```
// packages/coding-agent/src/core/session-manager.ts:137-146

type SessionEntry =
  | SessionMessageEntry      // 对话消息（user/assistant/toolResult）
  | ThinkingLevelChangeEntry // 思考级别变更
  | ModelChangeEntry         // 模型切换
  | CompactionEntry          // 上下文压缩摘要
  | BranchSummaryEntry       // 分支摘要
  | CustomEntry              // extension 数据（不进 LLM context）
  | CustomMessageEntry       // extension 消息（进 LLM context）
  | LabelEntry               // 用户书签
  | SessionInfoEntry;        // 会话元数据（显示名称）
```

这 9 种类型分为三层。理解这三层是理解整个 session 系统的关键。

核心层：影响 LLM context

核心层的 3 种类型直接参与 LLM context 的构建 — `buildSessionContext()` 函数在遍历树时，只有这些类型会被转换成发送给 LLM 的消息。

SessionMessageEntry — 对话的基本单元：

```
// packages/coding-agent/src/core/session-manager.ts:50-53

interface SessionMessageEntry extends SessionEntryBase {
  type: "message";
  message: AgentMessage;
}
```

`message` 字段是 `AgentMessage` 类型，来自 pi-agent-core 层。它可以是 user 消息、assistant 消息、或 tool result。注意 `SessionMessageEntry` 本身并不区分消息角色 — 角色信息在 `message.role` 内部。这意味着一个 user 消息和一个 assistant 消息在 session 树中的地位是完全相同的：都是节点，都有 `parentId`，都可以被分支。

CompactionEntry — 上下文压缩的结果（第 12 章详述）：

```
// packages/coding-agent/src/core/session-manager.ts:66-75

interface CompactionEntry<T = unknown> extends SessionEntryBase {
  type: "compaction";
  summary: string;
  firstKeptEntryId: string;
  tokensBefore: number;
  details?: T;
  fromHook?: boolean;
}
```

当对话变长、token 数接近 LLM 窗口限制时，pi 会把前面的对话压缩成一个摘要。`summary` 是压缩后的文本，`firstKeptEntryId` 标记从哪个 entry 开始保留原文（压缩点之后的消息不丢弃），`tokensBefore` 记录压缩前的 token 数。`details` 是一个泛型字段，让 extension 在压缩时附带结构化数据（比如文件操作索引）。`fromHook` 标记这个压缩是由 extension hook 生成的还是 pi 内置逻辑生成的。

BranchSummaryEntry — 分支时对被放弃路径的摘要：

```
// packages/coding-agent/src/core/session-manager.ts:77-85

interface BranchSummaryEntry<T = unknown> extends SessionEntryBase {
  type: "branch_summary";
  fromId: string;
  summary: string;
  details?: T;
  fromHook?: boolean;
}
```

当用户跳回某个节点创建新分支时，pi 可以为被放弃的对话路径生成一个摘要，注入到新分支的上下文中。这样 LLM 在新分支中知道"之前尝试过什么、为什么不行"，避免重复犯错。`fromId` 指向分支点，`summary` 是旧路径的摘要。

元数据层：影响 session 状态

元数据层的 3 种类型记录会话过程中的配置变化。它们不生成 LLM 消息，但在 session reload 时恢复状态。

ThinkingLevelChangeEntry — 用户在对话中途切换了 thinking level（如从 `off` 切到 `high`）。`buildSessionContext()` 在遍历路径时会跟踪这个值，返回当前路径最后生效的 thinking level。

ModelChangeEntry — 用户在对话中途切换了模型（如从 Claude Sonnet 切到 Claude Opus）。同样在路径遍历时被提取，恢复到最后一次切换的状态。

SessionInfoEntry — 会话元数据，主要字段是 `name`（用户自定义的会话显示名称）。这不是在树遍历中提取的，而是通过 `getSessionName()` 方法从后往前扫描最新的 `session_info` entry。会话命名的写入路径后来扩展了：除了交互中重命名，还可以在启动时用 `--name` / `-n` 指定（覆盖 interactive/print/JSON/RPC 各模式），以及在 SDK/extension 里用 `pi.setSessionName()` 写入 —— 它们最终都落成一条 `session_info` entry。

这三种类型的共同设计特点是：它们记录**事件**而非**状态**。不维护一个"当前配置"对象，而是把每次变更作为事件追加。最终状态通过重放事件得出。这是 event sourcing 的思想 — 在 append-only 的存储中，这是唯一合理的做法。

扩展层：为 extension 和用户提供扩展点

扩展层的 3 种类型是为生态设计的。它们让 extension 和用户可以在 session 中持久化自己的数据，而不需要修改 pi 的核心代码。

CustomEntry — extension 的私有数据仓库：

```
// packages/coding-agent/src/core/session-manager.ts:97-101

interface CustomEntry<T = unknown> extends SessionEntryBase {
  type: "custom";
  customType: string;
  data?: T;
}
```

不参与 LLM context。`customType` 是 extension 的标识符（如 `"my-extension:state"`），`data` 是任意结构化数据。用途：extension 在 session 中持久化内部状态。例如，一个代码审查 extension 可以把已审查的文件列表存为 `CustomEntry`，session reload 时扫描 `customType` 恢复状态。

CustomMessageEntry — extension 注入的 LLM 消息：

```
// packages/coding-agent/src/core/session-manager.ts:128-134

interface CustomMessageEntry<T = unknown> extends SessionEntryBase {
  type: "custom_message";
  customType: string;
  content: string | (TextContent | ImageContent)[];
  details?: T;
  display: boolean;
}
```

参与 LLM context — `buildSessionContext()` 会把它转换成 user 消息发送给 LLM。`content` 可以是纯文本或富媒体内容（文字 + 图片）。`display` 控制 TUI 渲染：`false` 表示完全隐藏（LLM 能看到但用户在界面上看不到），`true` 表示用特殊样式渲染（与普通 user 消息视觉区分）。`details` 存放 extension 私有元数据，不发送给 LLM。

`CustomEntry` 和 `CustomMessageEntry` 的关键区别值得再强调一次：

- `CustomEntry`：**不污染** LLM 对话。extension 的内部状态，LLM 看不到
- `CustomMessageEntry`：**影响** LLM 的输入。extension 主动向 LLM 注入额外上下文

这个区分让 extension 既能存储自己的状态（不干扰 LLM 对话质量），又能在需要时影响 LLM 的行为（比如注入项目约定、代码规范等上下文）。

LabelEntry — 用户书签：

```
// packages/coding-agent/src/core/session-manager.ts:104-108

interface LabelEntry extends SessionEntryBase {
  type: "label";
  targetId: string;
  label: string | undefined;
}
```

`targetId` 指向被标记的 entry，`label` 是用户定义的标签文本。传 `undefined` 或空字符串表示清除标签。注意 `LabelEntry` 本身也是树节点（有 `id` 和 `parentId`），但它的语义是"对另一个 entry 的标注"。`buildSessionContext()` 完全忽略它。标签通过 `SessionManager` 内部的 `labelsById` Map 解析，在 `getTree()` 返回的树节点中作为 `label` 字段附带。

SessionTreeNode：从 entry 列表到内存中的树

JSONL 文件是扁平的 — 所有 entry 按追加顺序排列，树结构隐含在 `parentId` 链中。要进行树操作（找分支、遍历路径、显示树形视图），需要先把扁平列表构建成内存中的树。这就是 `SessionTreeNode` 的角色：

```
// packages/coding-agent/src/core/session-manager.ts:152-159

interface SessionTreeNode {
  entry: SessionEntry;
  children: SessionTreeNode[];
  /** Resolved label for this entry, if any */
  label?: string;
  /** Timestamp of the latest label change */
  labelTimestamp?: string;
}
```

`getTree()` 方法负责构建这棵树：

```
// packages/coding-agent/src/core/session-manager.ts:1070-1107

getTree(): SessionTreeNode[] {
  const entries = this.getEntries();
  const nodeMap = new Map<string, SessionTreeNode>();
  const roots: SessionTreeNode[] = [];

  // 第一遍：创建所有节点，解析 label
  for (const entry of entries) {
    const label = this.labelsById.get(entry.id);
    const labelTimestamp = this.labelTimestampsById.get(entry.id);
    nodeMap.set(entry.id, { entry, children: [], label, labelTimestamp
});
  }

  // 第二遍：建立父子关系
  for (const entry of entries) {
    const node = nodeMap.get(entry.id)!;
    if (entry.parentId === null || entry.parentId === entry.id) {
      roots.push(node);
    } else {
      const parent = nodeMap.get(entry.parentId);
      if (parent) {
        parent.children.push(node);
      } else {
        roots.push(node);  // 孤儿节点 → 当作根
      }
    }
  }
  // ... 按时间排序 children
  return roots;
}
```

构建逻辑分两遍遍历：第一遍为每个 entry 创建 `SessionTreeNode`，同时从 `labelsById` Map 解析标签；第二遍根据 `parentId` 建立父子关系。如果某个 entry 的 `parentId` 指向不存在的 id（可能是数据损坏），它被当作孤儿节点推入 `roots` 数组。最后，每个节点的 `children` 按时间戳排序，确保旧分支在前、新分支在后。

注意 `getTree()` 返回的是"防御性浅拷贝" — `entry` 对象是原始引用，但树结构（`children` 数组）是新建的。这意味着调用者可以安全地遍历树，而不会意外修改 `SessionManager` 的内部状态。

一个设计细节：正常的 session 应该只有一个根节点（第一条用户消息）。但 `getTree()` 返回的是 `SessionTreeNode[]`（数组），允许多个根。这是防御性设计 — 处理数据损坏或 `resetLeaf()` 后创建的多棵子树。

buildSessionContext：从树到 LLM 消息数组

`buildSessionContext()` 是 session 存储和 Agent 运行时之间的桥梁。Agent 需要一个线性的消息数组来调用 LLM，而 session 是一棵树。这个函数的职责是：给定一个叶节点，沿 `parentId` 链回溯到根节点，收集路径上的消息，构建 `SessionContext`。

```
// packages/coding-agent/src/core/session-manager.ts:310-348

function buildSessionContext(
  entries: SessionEntry[],
  leafId?: string | null,
  byId?: Map<string, SessionEntry>,
): SessionContext {
  // 1. 构建 id→entry 索引（如果没传入）
  if (!byId) {
    byId = new Map<string, SessionEntry>();
    for (const entry of entries) {
      byId.set(entry.id, entry);
    }
  }

  // 2. 找到叶节点
  let leaf: SessionEntry | undefined;
  if (leafId === null) {
    return { messages: [], thinkingLevel: "off", model: null };
  }
  if (leafId) {
    leaf = byId.get(leafId);
  }
  if (!leaf) {
    leaf = entries[entries.length - 1];  // 默认取最后一个
  }

  // 3. 从叶到根收集路径
  const path: SessionEntry[] = [];
  let current: SessionEntry | undefined = leaf;
  while (current) {
    path.unshift(current);
    current = current.parentId ? byId.get(current.parentId) : undefined;
  }
  // ...
}
```

路径收集之后，函数进入两个阶段：

阶段一：提取元数据。 遍历路径上的每个 entry，跟踪最后生效的 `thinkingLevel` 和 `model`，以及路径上最后一个 `CompactionEntry`。

阶段二：构建消息数组。 这里有两种情况：

1. **无 compaction**：路径上所有 `message`、`custom_message`、`branch_summary` 类型的 entry 依次转换成 `AgentMessage`，其他类型（`thinking_level_change`、`model_change`、`custom`、`label`、`session_info`）被跳过。

2. **有 compaction**：先输出压缩摘要消息，然后输出从 `firstKeptEntryId` 到 compaction 之间的保留消息，最后输出 compaction 之后的消息。这保证了 LLM 看到的是"摘要 + 保留的近期对话 + 最新对话"，而不是完整的历史。

```
// packages/coding-agent/src/core/session-manager.ts:373-383

const appendMessage = (entry: SessionEntry) => {
  if (entry.type === "message") {
    messages.push(entry.message);
  } else if (entry.type === "custom_message") {
    messages.push(
      createCustomMessage(entry.customType, entry.content,
        entry.display, entry.details, entry.timestamp),
    );
  } else if (entry.type === "branch_summary" && entry.summary) {
    messages.push(createBranchSummaryMessage(
      entry.summary, entry.fromId, entry.timestamp));
  }
};
```

`appendMessage` 是一个局部函数，它定义了"哪些 entry 类型产生 LLM 消息"的规则。注意 `CustomMessageEntry` 被转换成 `CustomMessage`（一种特殊的 user 消息），而 `BranchSummaryEntry` 被转换成 `BranchSummaryMessage`。这些转换由 `messages.ts` 中的工厂函数完成，确保消息格式符合 LLM API 的要求。

返回的 `SessionContext` 包含三个字段：

```
// packages/coding-agent/src/core/session-manager.ts:161-165

interface SessionContext {
  messages: AgentMessage[];      // 发送给 LLM 的消息数组
  thinkingLevel: string;         // 当前路径的 thinking level
  model: { provider: string; modelId: string } | null;
}
```

Agent 拿到 `SessionContext` 后，用 `messages` 调用 LLM，用 `thinkingLevel` 和 `model` 恢复当前配置。整个流程形成一条清晰的数据管线：**JSONL 文件 → entry 列表 → 树遍历 → 路径提取 → 消息数组 → LLM 调用**。

Session 版本迁移：v1 → v2 → v3

会话格式经历了三个版本。向后兼容至关重要 — 用户的历史 session 文件不能因为升级 pi 而丢失。

v1：纯线性列表。 最早的版本没有 `id` 和 `parentId`，entry 按顺序排列，不支持分支。

v2：加入树形结构。 为每个 entry 生成 `id`，将前后关系转换成 `parentId` 链。

```
// packages/coding-agent/src/core/session-manager.ts:211-237

function migrateV1ToV2(entries: FileEntry[]): void {
  const ids = new Set<string>();
  let prevId: string | null = null;

  for (const entry of entries) {
    if (entry.type === "session") {
      entry.version = 2;
      continue;
    }

    entry.id = generateId(ids);
    entry.parentId = prevId;
    prevId = entry.id;

    // 转换 compaction 的索引引用为 id 引用
    if (entry.type === "compaction") {
      const comp = entry as CompactionEntry
        & { firstKeptEntryIndex?: number };
      if (typeof comp.firstKeptEntryIndex === "number") {
        const targetEntry = entries[comp.firstKeptEntryIndex];
        if (targetEntry && targetEntry.type !== "session") {
          comp.firstKeptEntryId = targetEntry.id;
        }
        delete comp.firstKeptEntryIndex;
      }
    }
  }
}
```

迁移逻辑很直接：遍历所有 entry，为每个生成唯一 id，`parentId` 指向前一个 entry（因为 v1 是线性的，所以父节点就是前一个）。特殊处理 `CompactionEntry` — v1 用数

组索引（`firstKeptEntryIndex`）标记保留起点，v2 改用 entry id（`firstKeptEntryId`）。

v3：重命名 hookMessage → custom。 早期 extension 消息的角色名叫 `hookMessage`，v3 统一改为 `custom`。这是一个语义清理。

```
// packages/coding-agent/src/core/session-manager.ts:239-255

function migrateV2ToV3(entries: FileEntry[]): void {
  for (const entry of entries) {
    if (entry.type === "session") {
      entry.version = 3;
      continue;
    }
    if (entry.type === "message") {
      const msgEntry = entry as SessionMessageEntry;
      if (msgEntry.message
        && (msgEntry.message as { role: string }).role
          === "hookMessage") {
        (msgEntry.message as { role: string }).role = "custom";
      }
    }
  }
}
```

所有迁移由 `migrateToCurrentVersion()` 统一调度，在 `setSessionFile()` 加载文件时自动执行。迁移是就地修改（mutate in place），修改后用 `_rewriteFile()` 重写整个文件。这意味着迁移是单向的、不可逆的 — 一旦文件被升级到 v3，旧版本的 pi 无法读取它。这是一个有意识的取舍：pi 的更新频率足够高，不需要支持降级。

```
// packages/coding-agent/src/core/session-manager.ts:261-271

function migrateToCurrentVersion(entries: FileEntry[]): boolean {
  const header = entries.find((e) => e.type === "session")
    as SessionHeader | undefined;
  const version = header?.version ?? 1;

  if (version >= CURRENT_SESSION_VERSION) return false;

  if (version < 2) migrateV1ToV2(entries);
  if (version < 3) migrateV2ToV3(entries);

  return true;
}
```

版本号采用递增整数（不是 semver），迁移链是线性的。加新版本时，只需在末尾加一个 `if (version < N) migrateVN_1ToVN(entries)` 。这种"梯级迁移"模式简单、可组合、不容易出错。

具体案例：5 轮对话 + 在第 3 轮创建分支

让我们用一个具体的例子来理解 session tree 的工作流。

用户开始一个新 session，聊了 5 轮（10 条消息）。然后发现第 3 轮的方向不对，跳回第 3 轮的用户消息处创建新分支。

第一阶段：正常的 5 轮对话

JSONL 文件内容（简化，只保留关键字段）：

```
{"type":"session","version":3,"id":"sess-
001","timestamp":"...","cwd":"/project"}
{"type":"message","id":"e1","parentId":null,"message":
{"role":"user","content":"分析 auth 模块"}}
{"type":"message","id":"e2","parentId":"e1","message":
{"role":"assistant","content":"好的，我来看看..."}}
{"type":"message","id":"e3","parentId":"e2","message":
{"role":"user","content":"重构 login 函数"}}
{"type":"message","id":"e4","parentId":"e3","message":
{"role":"assistant","content":"我建议用策略模式..."}}
{"type":"message","id":"e5","parentId":"e4","message":
{"role":"user","content":"继续"}}
{"type":"message","id":"e6","parentId":"e5","message":
{"role":"assistant","content":"已重构完成..."}}
{"type":"message","id":"e7","parentId":"e6","message":
{"role":"user","content":"添加测试"}}
{"type":"message","id":"e8","parentId":"e7","message":
{"role":"assistant","content":"已添加 3 个测试..."}}
{"type":"message","id":"e9","parentId":"e8","message":
{"role":"user","content":"检查覆盖率"}}
{"type":"message","id":"ea","parentId":"e9","message":
{"role":"assistant","content":"覆盖率 85%..."}}
```

此时 `leafId = "ea"`，树是一条直线：`e1 → e2 → e3 → e4 → e5 → e6 → e7 → e8 → e9 → ea`。

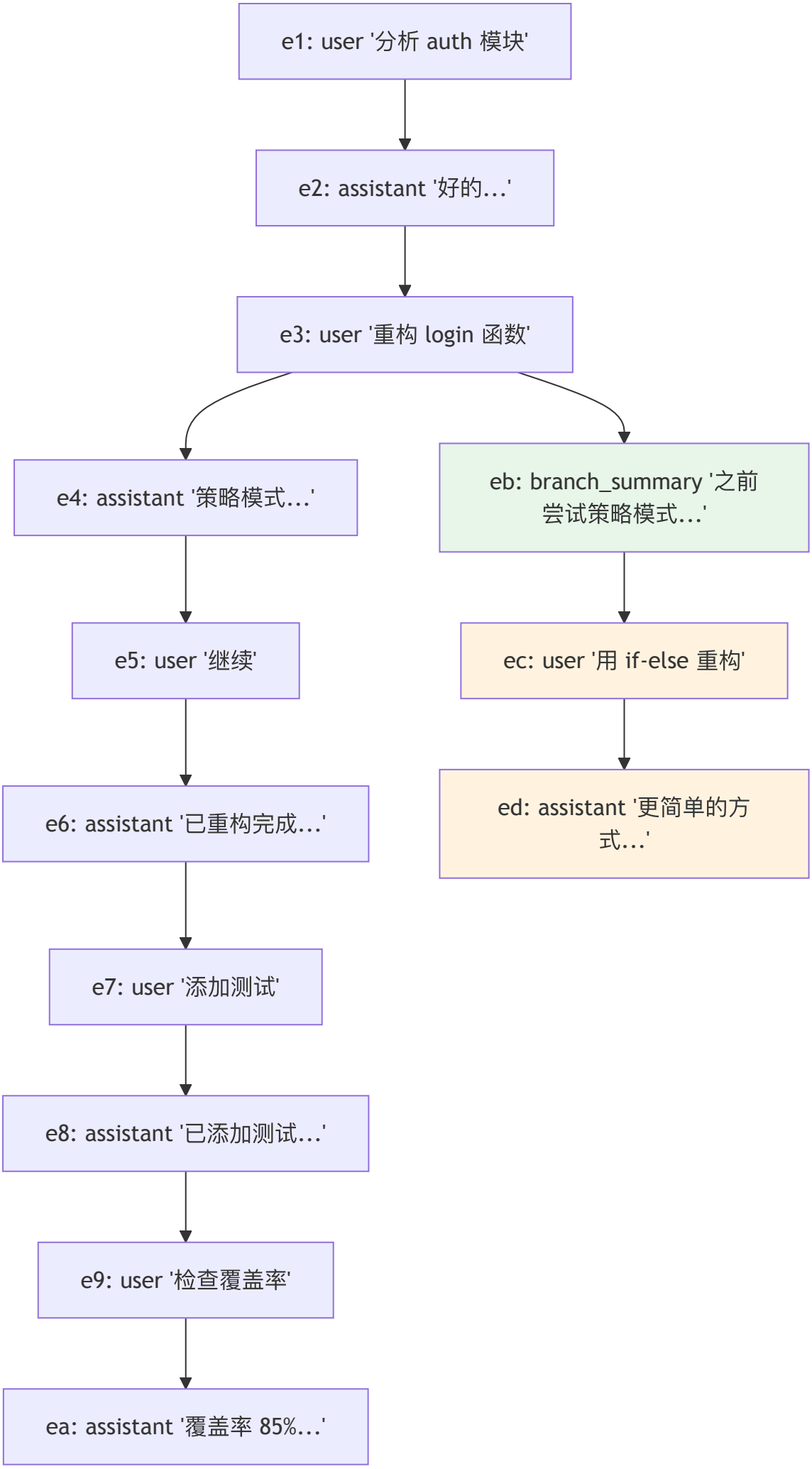
第二阶段：在第 3 轮创建分支

用户发现策略模式太重了，想回到 `e3`（"重构 login 函数"）换一种方式。TUI 调用 `sessionManager.branch("e3")`，然后用户输入新消息。

pi 不修改任何已有 entry，只追加新行：

```
{"type":"branch_summary","id":"eb","parentId":"e3","fromId":"e3","summar
y":"之前尝试用策略模式重构 login，完成了重构并添加了 3 个测试，覆盖率 85%"}
{"type":"message","id":"ec","parentId":"eb","message":
{"role":"user","content":"用简单的 if-else 重构"}}
{"type":"message","id":"ed","parentId":"ec","message":
{"role":"assistant","content":"好的，更简单的方式..."}}
```

此时 `leafId = "ed"`，树变成了：


e1: user '分析 auth 模块'
e2: assistant '好的...'
e3: user '重构 login 函数'
e4: assistant '策略模式...'
eb: branch_summary '之前尝试策略模式...'
e5: user '继续'
ec: user '用 if-else 重构'
e6: assistant '已重构完成...'
ed: assistant '更简单的方式...'
e7: user '添加测试'
e8: assistant '已添加测试...'
e9: user '检查覆盖率'
ea: assistant '覆盖率 85%...'

LLM 看到的消息

调用 `buildSessionContext(entries, "ed")` 时，函数从 `ed` 沿 `parentId` 回溯到根：`e1 → e2 → e3 → eb → ec → ed`。注意旧分支的 `e4` 到 `ea` 完全不在这条路径上。

LLM 收到的消息数组是：

1. `user: "分析 auth 模块"` ← e1
2. `assistant: "好的，我来看看..."` ← e2
3. `user: "重构 login 函数"` ← e3
4. `user: "[Branch Summary] 之前尝试用策略模式重构..."` ← eb（BranchSummaryEntry 转换的消息）
5. `user: "用简单的 if-else 重构"` ← ec
6. `assistant: "好的，更简单的方式..."` ← ed

LLM 知道之前试过策略模式（通过 branch summary），但上下文中只有新分支的对话。旧分支的 6 条消息不占用 token 预算。

JSONL 文件的关键特性

整个过程中，JSONL 文件只做了 append 操作。原始的 10 行没有被修改或删除。分支操作的"成本"是 3 行新 entry（branch_summary + 2 条新消息），而不是复制前 6 行。这就是 `parentId` 树结构的核心价值：**分支是零拷贝的**。

三种分支命令：/tree、/fork、/clone

在这套数据模型之上，pi 暴露了三个语义不同的用户命令（interactive-mode.ts:2587-2597）：

- **`/tree`** —— 在**同一个文件内**原地分支：追加一个 `parentId` 指向更早节点的新 entry，旧分支继续留在同一 JSONL 里（就是上面案例展示的零拷贝分支）。
- **`/fork`** —— 从某条更早的 user 消息**开一个新文件**，常用于"我想从这里重新走一条路，但不想污染当前会话历史"。fork 支持 `position` 选项（`"before" | "at"`，interactive-mode.ts:4407）：`at` 从该消息处分叉，`before` 从它**之前**分叉（连这条 user 消息也重来）。
- **`/clone`** —— 把**当前活动分支**复制到一个新文件，相当于"另存为"，保留当前这一条路径作为新会话的起点。

`/tree` 是零拷贝的同文件分支，`/fork` 和 `/clone` 则产生新文件 —— 这正是上一节"提取单条路径到新文件"的两种入口。定位会话还多了两个自动化友好的入口：`--session-id <id>`（精确创建/恢复某个项目会话，args.ts:108）与 `PI_CODING_AGENT_SESSION_DIR` 环境变量（覆盖会话存储目录，args.ts:375）。

RPC 只读会话树访问：get_entries / get_tree

前面讲的树结构（entry 列表、`SessionTreeNode`）此前只有 pi 自己的 TUI 能看到。v0.80.3 起，RPC 模式把它作为**只读**能力暴露出来，让 IDE 插件、Web 前端这类外部宿主也能渲染会话树：

```
// packages/coding-agent/src/modes/rpc/rpc-types.ts:64-65
| { id?: string; type: "get_entries"; since?: string }
| { id?: string; type: "get_tree" }
```

两个命令各有分工：`get_tree` 返回完整的 `SessionTreeNode[]`（就是本章 `getTree()` 构建的那棵树，带 label），适合一次性画出整棵分支图；`get_entries` 返回扁平的 entry 列表，且带一个 `since` 参数 —— 只要上次拿到的最后一个 entry id，就能**增量**拉取此后追加的 entry，而不必每次全量传输整棵树。对一个正在流式追加消息的长会话，`since` 让客户端的树视图能低成本地跟着更新。

这两个命令都是**只读**的 —— 它们只读 `SessionManager` 已经加载的树，不触发任何写操作，呼应本章"append-only + 防御性浅拷贝"的设计。相关地，会话选择器（`/resume` 列表）现在**按子树的最近活动时间排序**，让最近动过的会话/分支浮到前面。

存储层本身也更可插拔了：`JsonlSessionStorage` / `InMemorySessionStorage` 作为公共导出（第 26b 章 SDK），宿主可以自带存储后端；JSONL 会话头（`SessionHeader`）也支持自定义 metadata，让上层在会话文件里挂自己的元信息。

取舍分析

得到了什么

1. Append-only 的容错性。 JSONL 是 append-only 的 — 新 entry 只追加到文件末尾，不修改已有内容。即使进程在写入过程中崩溃，最坏情况是最后一行不完整（被丢弃），

之前的所有 entry 完好无损。数据库在崩溃时可能损坏索引或事务日志。`parseSessionEntries()` 中的 `try/catch` 正是为此设计 — 跳过格式错误的行，解析剩余内容。

2. 人类可读可修复。 JSONL 文件可以用任何文本编辑器打开、阅读、甚至手动修复。如果某个 entry 出了问题，用户可以直接删除那一行。`LabelEntry` 可以通过设置 `label: undefined` 来"删除"标签 — 不需要修改原始 entry，只需追加一个新的 label entry。

3. 零依赖。 不需要 SQLite、LevelDB 或任何外部存储引擎。`fs.appendFileSync` 就够了。pi 的 `_persist` 方法在有了第一条 assistant 消息后，要么 flush 全部内容，要么 append 单条 entry — 全部是同步文件 I/O，没有连接池、事务、WAL 日志等复杂性。

4. 天然支持分支。 `parentId` 让分支和跳转变成了简单的"追加一个新 entry，它的 parentId 指向目标节点"。不需要复制任何数据。`branch()` 方法只有一行核心逻辑：`this.leafId = branchFromId`。

放弃了什么

1. 没有随机访问。 要找到某个 entry，必须从头读到尾构建 `byId` Map。对于长会话（几百轮），这意味着每次加载时要解析整个 JSONL 文件。实际的性能特征取决于文件大小：

- 10 轮对话（~20 entry，~10KB）：解析时间 < 1ms，可以忽略
- 100 轮对话（~200 entry，~100KB）：解析时间几毫秒，用户无感知
- 500 轮对话（~1000 entry，带 tool result 可能 ~5MB）：解析时间几十毫秒，首次加载有轻微延迟

`buildSessionContext()` 的时间复杂度是 O(N)（N = 总 entry 数），因为即使只需要一条路径，也要先构建 `byId` 索引。不过索引构建后，路径遍历本身是 O(D)（D = 树的深度，即路径长度）。

> **订正（v0.78.1）**：早期 `/resume` 列出历史会话时，会把每个候选 JSONL **整文件读成字符串**再解析，会话一多就有 OOM 风险。现在改为**逐行流式读取**（`createInterface` + `for await (const line of rl)`，session-manager.ts:599-604），只解析定位会话列表所需的头部信息，不再把整个文件载入内存。注意这只改变了"浏览会话列表"的读取方式；**加载某个会话的完整上下文**仍需构建 `byId` Map（仍是 O(N)），上面的结论不变。此外，关闭会话时会 abort 在途的 agent / compaction / branch-summary / retry / bash（v0.77.0）。

2. 没有索引。 不能按工具名、时间范围、文件路径等维度快速查询。全部依赖遍历。如果需要"找到所有修改了 auth.ts 的 tool call"，必须遍历所有 entry 并检查消息内容。

3. 文件会持续增长。 compaction 压缩了 LLM context，但 JSONL 文件中的原始 entry 仍然保留。一个长时间使用的 session 可能有几 MB 的 JSONL 文件。`createBranchedSession()` 提供了一种"修剪"方式 — 提取单条路径到新文件，丢弃其他分支。但这创建的是新 session，不是原地修剪。

4. 树的遍历需要全量加载。 要构建树、找到某个分支、确定叶节点，都需要先把所有 entry 加载到内存。`SessionManager` 通过 `byId` Map 做了一层缓存，加载后的所有操作（`getBranch`、`getEntry`、`getChildren`）都是 O(1) 或 O(D) 的。但首次加载（`setSessionFile`）和迁移（`migrateToCurrentVersion`）必须全量读取和解析。

5. 单进程写入。 `appendFileSync` 是同步阻塞调用，不支持并发写入。如果多个 pi 实例同时操作同一个 session 文件，会出现竞争条件。这对 pi 的使用场景不是问题 — 每个终端窗口是独立 session — 但对多用户场景（如团队共享 session）不适用。

对于 pi 的使用场景 — 单用户、本地文件系统、会话大小通常在 KB 到低 MB 级别 — JSONL 的简单性远胜于数据库的复杂性。选择一个"够用的简单方案"而不是"面面俱到的复杂方案"，这本身就是一个值得学习的工程决策。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。9 种 entry、v1→v2→v3 迁移、parentId 树形零拷贝分支均不变。 主要演进：① RPC 新增只读会话树访问 `get_entries`（带 `since` 增量）/ `get_tree`（v0.80.3， `rpc-types.ts:64-65`，见第 26 章），会话选择器按子树最近活动排序；② 可插拔存储 `JsonlSessionStorage` / `InMemorySessionStorage` 导出、JSONL 会话头自定义 metadata（v0.80.4）；③ 会话 id 改用 UUIDv7（v0.67.1，时间有序，利于按 id 路由 + prompt 缓存）；④ 命令三件套 `/tree`（同文件原地分支）、`/fork`（开新文件，支持 `position: before|at`）、`/clone`（复制当前分支到新文件）；⑤ `--session-id` 精确定位（v0.76.0）、`PI_CODING_AGENT_SESSION_DIR`（v0.71.0）；⑥ `/resume` 改为逐行流式读 JSONL（v0.78.1，防 OOM）。 `CustomEntry`、`CustomMessageEntry`、`LabelEntry`、`SessionInfoEntry` 为 extension 生态和用户体验提供扩展点；旧版 session 通过 `migrateV1ToV2` / `migrateV2ToV3` 自动升级。

第 12 章：Compaction — 把无限对话装进有限窗口

> **定位**：本章解析 pi 如何在不丢失关键信息的前提下压缩超长对话。 前置依赖：第 11 章（会话树）、第 8 章（循环引擎的 transformContext）。 适用场景：当你想理解 agent 产品如何处理 context overflow，或者想设计自己的上下文管理策略。

Context window 快满了，怎么办?

这是本章的核心设计问题。

用户和 agent 聊了 200 轮。每轮包含用户消息、assistant 响应、可能还有多个工具调用的输入输出。总 token 量轻松超过模型的 context window。

最简单的做法是截断 — 扔掉最早的消息。但 coding agent 的上下文不是闲聊，第 5 轮修改的文件结构可能是第 200 轮决策的基础。暴力截断会让 agent 丢失关键的工作记忆。

pi 的解决方案是 **compaction** — 用 LLM 自己来摘要旧对话，然后用摘要替换原始消息。

何时触发 Compaction

Compaction 不是每轮都触发的。pi 在每次 assistant 响应返回后检查 context 使用量，只在接近上限时触发。

Token 计算策略

pi 用两种方式估算 context 占用量。首先，如果最近的 assistant 消息有 `usage` 字段（来自 API 的真实 token 统计），直接用它：

```
// packages/coding-agent/src/core/compaction/compaction.ts:135-137

function calculateContextTokens(usage: Usage): number {
  return usage.totalTokens ||
    usage.input + usage.output + usage.cacheRead + usage.cacheWrite;
}
```

但如果最近的 assistant 消息之后又追加了新消息（比如用户的新输入），这些新消息没有 usage 数据。pi 对这些"尾部消息"使用 chars/4 的启发式估算：

```
// packages/coding-agent/src/core/compaction/compaction.ts:232-249（简化）

function estimateTokens(message: AgentMessage): number {
  let chars = 0;
  switch (message.role) {
    case "user": {
      // 提取 text content 长度
      chars = /* text content length */;
      return Math.ceil(chars / 4);
    }
    case "assistant": {
      // 累加 text + thinking + toolCall 长度
      for (const block of message.content) {
        if (block.type === "text") chars += block.text.length;
        else if (block.type === "thinking") chars +=
block.thinking.length;
        else if (block.type === "toolCall")
          chars += block.name.length +
JSON.stringify(block.arguments).length;
      }
      return Math.ceil(chars / 4);
    }
    // ... toolResult, bashExecution 等同理
  }
}
```

chars/4 是一个保守的启发式（偏向高估 token 数量）。宁可提前触发压缩，也不要因为低估而撞上 context window 的硬限制。对于图片消息，固定按 1200 token 估算。

`estimateContextTokens` 将两种策略组合：用最后一条有 usage 的 assistant 消息的真实 token 数，加上之后所有消息的启发式估算：

```
// packages/coding-agent/src/core/compaction/compaction.ts:186-214（简化）

function estimateContextTokens(messages: AgentMessage[]):
ContextUsageEstimate {
  const usageInfo = getLastAssistantUsageInfo(messages);

  if (!usageInfo) {
    // 没有任何 usage 数据，全部用启发式
    let estimated = 0;
    for (const message of messages) estimated +=
estimateTokens(message);
    return { tokens: estimated, usageTokens: 0, trailingTokens:
estimated };
  }

  const usageTokens = calculateContextTokens(usageInfo.usage);
  let trailingTokens = 0;
  for (let i = usageInfo.index + 1; i < messages.length; i++) {
    trailingTokens += estimateTokens(messages[i]);
  }

  return { tokens: usageTokens + trailingTokens, usageTokens,
trailingTokens };
}
```

触发判定和可配置阈值

判定逻辑只有一行，但背后有三个可配置参数：

```
// packages/coding-agent/src/core/compaction/compaction.ts:115-125

interface CompactionSettings {
  enabled: boolean;          // 是否启用压缩
  reserveTokens: number;     // 为 prompt + 响应预留的 token 数
  keepRecentTokens: number;  // 压缩时保留的最近消息的 token 数
}

const DEFAULT_COMPACTION_SETTINGS = {
  enabled: true,
  reserveTokens: 16384,      // 预留 16K
  keepRecentTokens: 20000,   // 保留最近 20K token 的消息
};
```

触发条件：

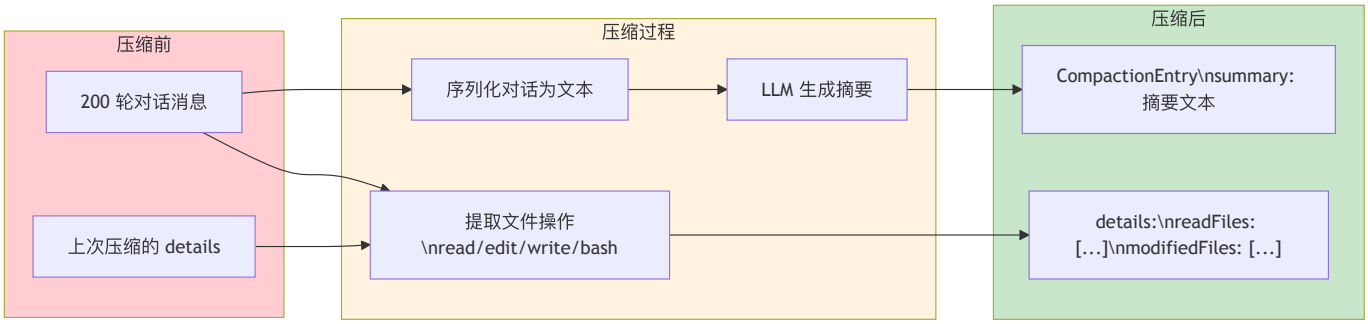
```
// packages/coding-agent/src/core/compaction/compaction.ts:219-222

function shouldCompact(contextTokens, contextWindow, settings): boolean
{
  if (!settings.enabled) return false;
  return contextTokens > contextWindow - settings.reserveTokens;
}
```

当 `contextTokens > contextWindow - reserveTokens` 时触发。以 200K context window 为例：当 context 使用超过 200K - 16K = 184K token 时，开始压缩。

`reserveTokens` 的设计意图是为 LLM 响应留出空间。如果 context 已经用了 199K，LLM 只剩 1K token 生成响应，这基本没法用。16K 的默认值足以让 LLM 生成有意义的响应。

压缩不只是摘要

pi 的 compaction 做了两件事：对话摘要 + 文件操作追踪。



文件操作追踪

压缩时， `extractFileOperations` 函数扫描所有工具调用消息，提取文件操作记录：

```
// packages/coding-agent/src/core/compaction/compaction.ts:32-36

interface CompactionDetails {
  readFiles: string[];      // agent 读过哪些文件
  modifiedFiles: string[];  // agent 修改过哪些文件
}
```

这些文件列表被存入 `CompactionEntry.details` 。为什么要追踪文件操作?

因为 LLM 的摘要会丢失具体细节（"修改了 api-registry.ts 的第 42-52 行"可能被压缩成"更新了注册表"），但文件路径不能丢。agent 需要知道它在这个 session 中读过和改过哪些文件，才能在后续决策中避免重复读取或矛盾的修改。

`extractFileOperations` 的实现还有一个设计细节 — 它会**累积**上次压缩的文件列表：

```
// packages/coding-agent/src/core/compaction/compaction.ts:41-61（简化）

function extractFileOperations(messages, entries, prevCompactionIndex) {
  const fileOps = createFileOps();

  // 从上次压缩的 details 中继承文件列表
  if (prevCompactionIndex >= 0) {
    const prevCompaction = entries[prevCompactionIndex];
    if (!prevCompaction.fromHook && prevCompaction.details) {
      const details = prevCompaction.details as CompactionDetails;
      for (const f of details.readFiles) fileOps.read.add(f);
      for (const f of details.modifiedFiles) fileOps.edited.add(f);
    }
  }

  // 从当前消息中提取新的文件操作
  for (const msg of messages) {
    extractFileOpsFromMessage(msg, fileOps);
  }

  return fileOps;
}
```

注意 `!prevCompaction.fromHook` 的检查 — 如果上次压缩是 extension 生成的（`fromHook: true`），不继承它的文件列表。因为 extension 的压缩策略可能和 pi 的不同，文件列表的格式不一定兼容。

`computeFileLists` 函数负责最终的分类 — 把 `read`、`written`、`edited` 三个 Set 合并为两个有序列表：

```
// packages/coding-agent/src/core/compaction/utils.ts:62-67

function computeFileLists(fileOps: FileOperations) {
  const modified = new Set([...fileOps.edited, ...fileOps.written]);
  const readOnly = [...fileOps.read].filter(f =>
!modified.has(f)).sort();
  const modifiedFiles = [...modified].sort();
  return { readFiles: readOnly, modifiedFiles };
}
```

逻辑很清晰：如果一个文件既被 read 又被 edit，它归类为 modified 而不重复出现在 read 列表中。文件列表最终以 XML 标签格式追加到摘要文本后面：

```
<read-files>
/project/src/config.ts
/project/src/utils.ts
</read-files>

<modified-files>
/project/src/api-registry.ts
/project/src/router.ts
</modified-files>
```

对话序列化

在发送给 LLM 做摘要之前，对话不是原样传递的。`serializeConversation` 函数将结构化的 `Message[]` 转换为纯文本格式：

```
// packages/coding-agent/src/core/compaction/utils.ts:109-162（简化）

function serializeConversation(messages: Message[]): string {
  const parts: string[] = [];
  for (const msg of messages) {
    if (msg.role === "user") {
      parts.push(`[User]: ${content}`);
    } else if (msg.role === "assistant") {
      // 分别序列化 thinking、text、tool calls
      if (thinkingParts.length > 0)
        parts.push(`[Assistant thinking]: ${thinkingParts.join("\n")}`);
      if (textParts.length > 0)
        parts.push(`[Assistant]: ${textParts.join("\n")}`);
      if (toolCalls.length > 0)
        parts.push(`[Assistant tool calls]: read(path="/src/foo.ts");
...`);
    } else if (msg.role === "toolResult") {
      // 工具结果被截断到 2000 字符
      parts.push(`[Tool result]: ${truncateForSummary(content, 2000)}`);
    }
  }
  return parts.join("\n\n");
}
```

为什么要序列化而不是直接传递消息数组？因为如果直接把对话作为 messages 传给 LLM，LLM 可能会尝试"继续"对话而不是摘要它。序列化为纯文本放在 `<conversation>`

标签中，明确告诉 LLM"这是要被摘要的内容，不是要继续的对话"。

注意工具结果被截断到 2000 字符。一个 `read` 工具的结果可能包含整个文件的内容（几千行代码），但摘要不需要这些细节 — 它只需要知道 agent 读了什么文件、做了什么决策。

摘要生成策略

pi 使用一个专门的 system prompt 来指导摘要生成：

```
// packages/coding-agent/src/core/compaction/utils.ts:168

const SUMMARIZATION_SYSTEM_PROMPT =
  `You are a context summarization assistant. Your task is to read a
conversation ` +
  `between a user and an AI assistant, then produce a structured summary
` +
  `following the exact format specified.\n\n` +
  `Do NOT continue the conversation. Do NOT respond to any questions in
the ` +
  `conversation. ONLY output the structured summary.`;
```

> **措辞中性化（v0.79.0）**：注意这里是"an AI **assistant**"而非早期的"an AI **coding** assistant"。这是一个看似微小、实则有意的改动 —— compaction 的内核（连同 `AgentSession`）现在被设计为可被**非编码场景的 SDK 调用方**复用（见第 26b 章 SDK）。把摘要 prompt 里的"coding"去掉，让同一套压缩逻辑在客服、写作等非编码 agent 里也不显得突兀。

用户 prompt 则要求生成结构化的 checkpoint 摘要，包含固定的六个章节：Goal、Constraints & Preferences、Progress（Done / In Progress / Blocked）、Key Decisions、Next Steps、Critical Context。

如果这不是首次压缩（已有上一次的摘要），pi 使用 **update prompt** 而不是 fresh prompt — 将新消息和旧摘要一起发送，要求 LLM 更新而非重写：

```
// packages/coding-agent/src/core/compaction/compaction.ts:540-558（简化）

let basePrompt = previousSummary
  ? UPDATE_SUMMARIZATION_PROMPT : SUMMARIZATION_PROMPT;

let promptText =
`<conversation>\n${conversationText}\n</conversation>\n\n`;
if (previousSummary) {
  promptText += `<previous-summary>\n${previousSummary}\n</previous-
summary>\n\n`;
}
promptText += basePrompt;
```

这种"增量更新"策略避免了随着压缩次数增加，摘要质量递减的问题。每次压缩都以上次摘要为基础，只需要处理新增的消息。

关于"用什么配置去跑这次摘要"，有几个后来调整的细节：摘要**不再强制用 high thinking level**，而是复用会话当前的 thinking level（`createSummarizationOptions(..., thinkingLevel)`，compaction.ts:603；仅当 `model.reasoning && thinkingLevel !== "off"` 时才设 `options.reasoning`，:536）—— 强制 high 在便宜模型或关思考的会话里既费钱又没必要。摘要请求走的是会话自定义的 agent stream 函数，以保留代理路由等设置（v0.75.0）；`maxTokens` 会被钳到模型上限以内（v0.74.1），避免请求超过模型能力。

切点检测：保留多少、摘要多少

Compaction 不会把所有旧消息都摘要掉。它需要决定一个"切点" — 从哪里开始保留原始消息。

从后往前走

`findCutPoint` 从最新的 entry 往回走，累积 token 数，直到达到 `keepRecentTokens`（默认 20000）的阈值。切点在这个位置附近：

```
// packages/coding-agent/src/core/compaction/compaction.ts:386-448（简化）

function findCutPoint(entries, startIndex, endIndex, keepRecentTokens) {
  const cutPoints = findValidCutPoints(entries, startIndex, endIndex);
  let accumulatedTokens = 0;
  let cutIndex = cutPoints[0];

  for (let i = endIndex - 1; i >= startIndex; i--) {
    const entry = entries[i];
    if (entry.type !== "message") continue;
    accumulatedTokens += estimateTokens(entry.message);
    if (accumulatedTokens >= keepRecentTokens) {
      // 找到最近的合法切点
      for (let c = 0; c < cutPoints.length; c++) {
        if (cutPoints[c] >= i) { cutIndex = cutPoints[c]; break; }
      }
      break;
    }
  }

  return { firstKeptEntryIndex: cutIndex, /* ... */ };
}
```

合法切点

不是任何位置都能切。`findValidCutPoints` 只允许在 user、assistant、bashExecution、custom 等消息处切割。**永远不在 toolResult 处切** — 因为 toolResult 必须紧跟在其对应的 tool call 之后，拆开它们会破坏对话结构。

Turn splitting

还有一个复杂情况：如果切点落在一个 turn 的中间（比如 assistant 消息之后、toolResult 之前），pi 会做 "turn splitting" — 对这个 turn 的前半部分生成一个额外的 prefix 摘要，保留后半部分的原始消息。这确保保留的消息在语义上是完整的。

JSONL 中的 CompactionEntry

压缩结果作为一个 `CompactionEntry` 写入 session 的 JSONL 文件（见第 11 章）。以下是一个真实的 entry 示例：


```
{
  "type": "compaction",
  "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "parentId": "previous-entry-uuid",
  "timestamp": "2026-04-10T08:30:00.000Z",
  "summary": "## Goal\nRefactor the API registry...\n\n## Progress\n### Done\n- [x] Split monolithic registry into per-provider modules\n...\n\n<read-files>\n/src/config.ts\n</read-files>\n\n<modified-files>\n/src/api-registry.ts\n/src/providers/openai.ts\n</modified-files>",
  "firstKeptEntryId": "kept-entry-uuid",
  "tokensBefore": 185432,
  "details": {
    "readFiles": ["/src/config.ts", "/src/utils.ts"],
    "modifiedFiles": ["/src/api-registry.ts", "/src/providers/openai.ts"]
  }
}
```


关键字段解读：

- `summary`：LLM 生成的结构化摘要 + 文件操作列表，这是 LLM 在后续对话中能看到的内容
- `firstKeptEntryId`：切点之后第一条保留的 entry 的 UUID，用于重建 context 时定位
- `tokensBefore`：压缩前的 context token 数，用于展示"从 185K 压缩到 20K"
- `details`：结构化的文件列表，下次压缩时会被继承

> **重复压缩的区间起点与 tokensBefore 重算**：当一个会话被**多次压缩**时，新一轮要摘要的区间并非从"上一条 CompactionEntry"开始，而是从**上一次压缩保留的边界** `firstKeptEntryId` 开始；如果那条 kept entry 在当前路径里找不到，则回退到"上一条 CompactionEntry 之后的那条 entry"。这样能把上一轮幸存下来的消息也纳入本轮摘要，不至于丢失。此外，pi 在写新的 `CompactionEntry` 之前，会从

> **重建后的会话上下文**重新计算 `tokensBefore`，让这个数字真实反映"本次被替换掉的压缩前上下文"，而不是沿用一个过时的旧值（docs/compaction.md）。

Extension 可以接管 Compaction

`CompactionEntry` 有一个 `fromHook` 字段：

```
// packages/coding-agent/src/core/session-manager.ts:66-75

interface CompactionEntry<T = unknown> {
  type: "compaction";
  summary: string;
  firstKeptEntryId: string;
  tokensBefore: number;
  details?: T;            // extension 可以存任意数据
  fromHook?: boolean;     // true = extension 生成的
}
```

当 `fromHook: true` 时，pi 知道这个压缩不是自己生成的，在处理时会更保守（比如不继承其文件列表）。

Extension 为什么要接管 compaction？因为不同的 agent 场景对"什么信息重要"的判断不同。一个代码审查 agent 可能想保留所有 diff 细节，而一个项目管理 agent 可能只想保留决策点。默认的 compaction 策略无法满足所有场景。

可恢复的压缩：重试、会计与路由隔离

压缩本身是一次 LLM 调用，它会失败 —— 网络抖动、provider 限流、超时。早期一旦摘要请求失败，这一轮压缩就丢了，用户可能撞上 context overflow。本区间把压缩/分支摘要做成了**可恢复**的。

可配置的重试策略 + 生命周期事件（v0.81.1）。 摘要请求现在走和普通 LLM 调用一样的重试逻辑（`RetryPolicy`），并把重试过程作为一组 `summarization_retry_*` 事件暴露出来（交互 / JSON / RPC / SDK 都能收到，`agent-session.ts:167-181`）：

```
// packages/coding-agent/src/core/agent-session.ts:167-181（节选）
| { type: "summarization_retry_scheduled";
    attempt: number; maxAttempts: number;
    delayMs: number; errorMessage: string }
| { type: "summarization_retry_attempt_start";
    source: "compaction";
    reason: "manual" | "threshold" | "overflow" }
| { type: "summarization_retry_finished" }
```

`reason` 区分了压缩的三种触发来源 —— 手动 `/compact`、阈值自动触发、以及 overflow 溢出重试；`willRetry` 则告诉消费者这次失败之后还会不会再试。这让宿主能给用户一个诚实的"正在重试压缩（第 2/3 次）"提示，而不是静默卡住。压缩事件还带上**估算的压缩后 token 数**（v0.79.8），让 UI 能显示"从 185K 压到约 20K"这样的预期结果。（这套产品层的 `summarization_retry_*` 与第 10 章 `AgentHarness` 内核里的 `RetryPolicy` / `retry_*` 是**不同层的同构机制** —— 都在"要额外调 LLM 的摘要请求"上加重试，但一个跑在 coding-agent 产品层、一个跑在 agent 内核，彼此独立。）

usage 会计扩展到工具/压缩/分支摘要（v0.81.0）。 早期只有主对话的 assistant 响应计入会话 token 总计。现在工具调用、压缩、分支摘要各自消耗的 usage 都被持久化并计入会话总计、footer 和会话统计（`docs/compaction.md`）。这修正了一个长期的低估 —— 压缩本身要读一大堆旧消息、生成摘要，这些 token 此前是"隐形"的。

压缩请求隔离路由并禁用 prompt caching（v0.82.0）。 摘要是**独立的一次性请求**，它和主对话共享缓存前缀既没有意义、又会污染主对话的缓存。所以压缩/分支摘要请求被显式地隔离出来：

```
// packages/coding-agent/src/core/compaction/compaction.ts:570-574
// Summaries are standalone requests, so isolate routing and
// avoid cache writes that cannot be reused.
const requestOptions: SimpleStreamOptions = {
  ...options,
  cacheRetention: "none",    // 不写 prompt cache
  sessionId: uuidv7(),       // 全新 routing session id
};
```

`sessionId` 用一个全新的 `uuidv7()`，让 provider 端按 id 路由时不会把摘要请求和主对话归到一起；`cacheRetention: "none"` 则避免为一次用不上的请求写缓存。同时，纯 header 认证的 provider 现在也能跑压缩了（v0.82.1）—— 摘要请求不再假设一定有 API key 形式的凭证。

Compaction 在产品层而非 Runtime 层

一个关键的架构决策是：compaction 不在 agent-core 层（第 8-10 章的循环引擎），而在 coding-agent 层（产品层）。

循环引擎通过 `transformContext` 回调接入 compaction：当 context 接近上限时，`transformContext` 可以把旧消息替换为压缩摘要。但 compaction 的触发条件、压缩策略、摘要生成都是产品层的逻辑。

为什么？因为 compaction 需要知道太多产品级信息：

- 配置（是否启用压缩、保留多少 token）
- 会话结构（哪些 entry 是 compaction、哪些是分支摘要）
- 文件操作追踪（需要扫描工具调用消息）
- extension 的参与（`fromHook`）

这些都不是一个通用循环引擎应该知道的。

准备阶段与执行阶段的分离

`prepareCompaction` 和 `compact` 是两个独立的函数，前者做所有的 CPU 计算（找切点、提取文件操作、整理消息），后者做 I/O（调用 LLM 生成摘要）：

```
// packages/coding-agent/src/core/compaction/compaction.ts:612-687（简化）

function prepareCompaction(pathEntries, settings): CompactionPreparation | undefined {
  // 1. 找到上次压缩的位置
  // 2. 估算当前 token 数
  // 3. findCutPoint 确定切点
  // 4. 分离 messagesToSummarize 和 turnPrefixMessages
  // 5. extractFileOperations 提取文件操作
  return {
    firstKeptEntryId, messagesToSummarize, turnPrefixMessages,
    isSplitTurn, tokensBefore, previousSummary, fileOps, settings,
  };
}
```

这种分离使得 extension 可以在 `prepare` 和 `compact` 之间插入自定义逻辑 — 比如修改 `messagesToSummarize` 列表、添加自定义的 details、或者完全替换摘要生成策略。

`compact` 函数接收 `CompactionPreparation` 并返回 `CompactionResult`。如果检测到 turn splitting，它会并行生成两个摘要（历史摘要 + turn prefix 摘要），然后合并：

```
// packages/coding-agent/src/core/compaction/compaction.ts:737-756（简化）

if (isSplitTurn && turnPrefixMessages.length > 0) {
  const [historyResult, turnPrefixResult] = await Promise.all([
    generateSummary(messagesToSummarize, model, ...),
    generateTurnPrefixSummary(turnPrefixMessages, model, ...),
  ]);
  summary = `${historyResult}\n\n---\n\n**Turn Context:**\n\n${turnPrefixResult}`;
} else {
  summary = await generateSummary(messagesToSummarize, model, ...);
}
```

最后追加文件操作列表，返回结果：

```
const { readFiles, modifiedFiles } = computeFileLists(fileOps);
summary += formatFileOperations(readFiles, modifiedFiles);
return { summary, firstKeptEntryId, tokensBefore, details: { readFiles, modifiedFiles } };
```

取舍分析

得到了什么

1. 理论上无限的对话长度。每次压缩后，context 回到可控范围内。用户可以和 agent 持续工作几百轮。

2. 文件操作记忆不丢失。即使对话细节被压缩了，agent 仍然知道它在这个 session 中接触过哪些文件。

3. Extension 可定制。通过 `fromHook` 机制和 prepare/compact 分离，不同的产品可以实现完全不同的压缩策略。

4. 增量更新避免质量递减。通过 previousSummary 机制，多次压缩不会产生"摘要的摘要"问题，而是持续更新同一个结构化文档。

放弃了什么

1. 压缩是有损的。LLM 生成的摘要不可能保留所有细节。一些 early-turn 的具体决策细节会丢失。

2. 压缩本身消耗 token。生成摘要需要一次额外的 LLM 调用，消耗 input token（读旧消息）和 output token（生成摘要）。`reserveTokens` 的 80% 被分配为摘要生成的 maxTokens。

3. 压缩不可逆。一旦压缩完成，原始消息在 LLM context 中被替换为摘要。虽然 JSONL 文件中原始 entry 仍然保留（第 11 章），但 LLM 不再能看到它们。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。Compaction 的核心数据模型（9 种 entry、reserveTokens/keepRecentTokens、增量更新摘要、CompactionDetails、fromHook 接管、BranchSummaryEntry）保持稳定。 主要演进：① **可恢复压缩**（v0.81.1）—— 摘要/分支摘要可配置重试策略 + `summarization_retry_*` 生命周期事件（交互/JSON/RPC/SDK 均可见），压缩事件带 `reason` / `willRetry`（v0.79.10）与估算压缩后 token 数（v0.79.8）；② **usage 会计扩展**（v0.81.0）—— 工具/压缩/分支摘要 usage 计入持久化会话总计、footer 与统计；③ **压缩请求隔离路由 + 禁用 prompt caching**（v0.82.0，`cacheRetention:"none"` + 全新 `sessionId`），纯 header 认证 provider 也能压缩（v0.82.1）；④ 摘要不再强制 high thinking，复用会话当前级别（v0.68.0），走自定义 agent stream 保留代理路由（v0.75.0），maxTokens 钳到模型上限（v0.74.1）；⑤ 摘要 system prompt 中性化为"AI assistant"以便非编码 SDK 复用（v0.79.0）；⑥ 重复压缩从上次 `firstKeptEntryId` 起算区间、写入前从重建上下文重算 `tokensBefore`。

第 13 章：三级配置覆盖

> **定位**：本章解析 pi 的配置系统 — 全局、项目、目录三级覆盖如何让同一个工具适应不同场景。 前置依赖：第 10 章（Agent 的状态管理）。 适用场景：当你想理解 pi 的配置优先级，或者想为自己的开发工具设计分层配置。

一个工具如何同时满足所有项目？

这是本章的核心设计问题。

用户 A 在公司项目中使用 Claude Opus，thinking level 设为 high，禁止 agent 修改 `deploy/` 目录。用户 A 在个人项目中使用 GPT-4o，thinking level 设为 medium，没有目录限制。用户 A 的公司项目的 `packages/legacy/` 子目录有特殊规则：只允许修改 `.test.ts` 文件。

一个配置文件搞不定。pi 的解决方案是三级覆盖：

```
~/.pi/agent/           ← 全局配置（所有项目）
  ├── settings.json
  ├── AGENTS.md
  └── SYSTEM.md

/project/.pi/          ← 项目配置（覆盖全局）
  ├── settings.json
  └── AGENTS.md

/project/packages/     ← 目录上溯
  └── legacy/
      └── AGENTS.md    ← 目录级规则（追加）
```

Settings：完整的可配置维度

`settings.json` 存储结构化配置。Settings 接口定义了 pi 所有可配置的维度：

```
// packages/coding-agent/src/core/settings-manager.ts:63-98（完整接口）

interface Settings {
  // 模型与 provider
  defaultProvider?: string;
  defaultModel?: string;
  defaultThinkingLevel?: "off" | "minimal" | "low" | "medium" | "high" |
"xhigh";
  transport?: TransportSetting;       // "sse" | "websocket"
  enabledModels?: string[];           // 模型循环列表

  // 操作模式
  steeringMode?: "all" | "one-at-a-time";
  followUpMode?: "all" | "one-at-a-time";

  // 外观
  theme?: string;
  hideThinkingBlock?: boolean;

  // Compaction 与分支摘要
  compaction?: CompactionSettings;
  branchSummary?: BranchSummarySettings;

  // 重试策略
  retry?: RetrySettings;

  // 终端行为
  terminal?: TerminalSettings;        // showImages, clearOnShrink
  images?: ImageSettings;             // autoResize, blockImages

  // Thinking token 预算
  thinkingBudgets?: ThinkingBudgetsSettings;

  // Shell 定制
  shellPath?: string;
  shellCommandPrefix?: string;
  npmCommand?: string[];

  // 能力扩展
  packages?: PackageSource[];
  extensions?: string[];
  skills?: string[];
  prompts?: string[];
  themes?: string[];
```

```
  enableSkillCommands?: boolean;

  // UI 细节
  markdown?: MarkdownSettings;
  editorPaddingX?: number;
  autocompleteMaxVisible?: number;
  showHardwareCursor?: boolean;
  doubleEscapeAction?: "fork" | "tree" | "none";
  treeFilterMode?: "default" | "no-tools" | "user-only" | "labeled-only"
| "all";

  // 安全：项目信任（仅全局有效）
  defaultProjectTrust?: "ask" | "always" | "never";  // default: "ask"

  // 杂项
  lastChangelogVersion?: string;
  quietStartup?: boolean;
  collapseChangelog?: boolean;
  sessionDir?: string;
  httpProxy?: string;
  httpIdleTimeoutMs?: number;
}
```

每个子接口也值得展开看看。这些子接口展示了 pi 在不同维度上提供的精细控制：

```
// packages/coding-agent/src/core/settings-manager.ts:7-44

interface CompactionSettings {
  enabled?: boolean;          // default: true
  reserveTokens?: number;     // default: 16384
  keepRecentTokens?: number; // default: 20000
}

interface BranchSummarySettings {
  reserveTokens?: number;     // default: 16384
  skipPrompt?: boolean;       // default: false
}

interface RetrySettings {
  enabled?: boolean;      // default: true
  maxRetries?: number;    // default: 3
  baseDelayMs?: number;  // default: 2000（pi 应用层指数退避：2s, 4s, 8s）
  provider?: ProviderRetrySettings;  // v0.70.1 新增，透传给 SDK/provider
}

interface ProviderRetrySettings {  // settings-manager.ts:21-25
  timeoutMs?: number;        // SDK/provider 单次请求超时
  maxRetries?: number;       // provider 层重试次数
  maxRetryDelayMs?: number; // default: 60000，服务端要求的最大退避，超过即失
败
}

interface TerminalSettings {
  showImages?: boolean;       // default: true
  clearOnShrink?: boolean;    // default: false
}

interface ImageSettings {
  autoResize?: boolean;       // default: true（最大 2000x2000）
  blockImages?: boolean;      // default: false
}

interface ThinkingBudgetsSettings {
  minimal?: number;
  low?: number;
  medium?: number;
  high?: number;
}
```

```
interface MarkdownSettings {
  codeBlockIndent?: string;  // default: "  "
}
```

注意所有字段都是 optional（`?`）。这是"渐进式定制"的基础 — 用户只需要设置自己关心的字段，其他全部使用默认值。

配置维度的设计逻辑

这些配置项可以分为几个层次来理解：

模型层：`defaultProvider`、`defaultModel`、`defaultThinkingLevel`、`transport`、`enabledModels` — 控制 agent 使用哪个模型、怎么连接。这是最基础的配置，通常在全局级别设置一次。

行为层：`compaction`、`retry`、`branchSummary`、`steeringMode`、`followUpMode` — 控制 agent 的运行策略。比如一个大型 monorepo 项目可能需要更大的 `keepRecentTokens`（因为上下文更复杂），而一个简单的脚本项目可以用默认值。

环境层：`terminal`、`images`、`shellPath`、`shellCommandPrefix`、`npmCommand` — 适配不同的运行环境。Cygwin 用户需要自定义 `shellPath`，SSH 环境可能需要 `blockImages`。

能力层：`packages`、`extensions`、`skills`、`prompts`、`themes`、`enableSkillCommands` — 控制 pi 加载哪些外部能力。这些配置可以在全局和项目级别分别设置，实现"全局装常用 skills，项目装专用 skills"的效果。

UI 层：`markdown`、`editorPaddingX`、`autocompleteMaxVisible`、`showHardwareCursor`、`doubleEscapeAction`、`treeFilterMode` — 纯粹的用户体验偏好，通常只在全局设置。

每一层的默认值都经过精心选择。比如 `retry.baseDelayMs = 2000` 配合指数退避产生 2s → 4s → 8s 的重试间隔 — 既不会因为太频繁而被 API 限流，也不会因为等太久而影响用户体验。`compaction.keepRecentTokens = 20000` 大约相当于 10-15 轮对话，足以保留足够的近期上下文。

> **重试配置分两层（v0.70.1）**：注意 `retry` 现在拆成了两层语义。顶层的 `enabled` / `maxRetries` / `baseDelayMs` 是 **pi 应用层**的退避策略（pi 自己在两次 stream 调用之间等多久再重试）；而 `retry.provider`（`ProviderRetrySettings`）是**透传给底层 SDK/provider** 的参数 —— `timeoutMs`（单次请求超时）、`maxRetries`

> （provider 自己的重试次数）、`maxRetryDelayMs`（服务端通过 `Retry-After` 要求的最大退避，超过即放弃）。早期版本只有一个扁平的 `retry.maxDelayMs`，它在迁移时被自动搬到 `retry.provider.maxRetryDelayMs`（见 `migrateSettings`，settings-manager.ts:409-430）。把"应用层何时重试"和"provider 层如何重试"分开，是因为这两者解决的是不同层面的失败。

Settings 的加载与合并

两级加载

`SettingsManager` 的核心加载逻辑是：分别加载 global 和 project 两级配置，然后深度合并。

```
// packages/coding-agent/src/core/settings-manager.ts:258-283（简化）

static create(cwd, agentDir): SettingsManager {
  const storage = new FileSettingsStorage(cwd, agentDir);
  return SettingsManager.fromStorage(storage);
}

static fromStorage(storage): SettingsManager {
  const globalLoad = SettingsManager.tryLoadFromStorage(storage,
"global");
  const projectLoad = SettingsManager.tryLoadFromStorage(storage,
"project");
  // 收集加载错误但不中断
  return new SettingsManager(
    storage,
    globalLoad.settings,
    projectLoad.settings,
    globalLoad.error,
    projectLoad.error,
  );
}
```

文件路径固定：

- 全局：`~/.pi/agent/settings.json`

- 项目：`{cwd}/.pi/settings.json`

加载使用 `tryLoadFromStorage` — 如果文件不存在或 JSON 解析失败，返回空对象 `{}` 而不是崩溃。错误被记录下来，可以后续通过 `drainErrors()` 检查。这个设计让 pi 在配置文件损坏时仍然能启动。

深度合并策略

两级配置通过 `deepMergeSettings` 合并：

```
// packages/coding-agent/src/core/settings-manager.ts:101-129（简化）

function deepMergeSettings(base: Settings, overrides: Settings):
Settings {
  const result = { ...base };
  for (const key of Object.keys(overrides)) {
    const overrideValue = overrides[key];
    const baseValue = base[key];

    if (overrideValue === undefined) continue;

    // 嵌套对象：递归合并
    if (typeof overrideValue === "object" &&
!Array.isArray(overrideValue)
        && typeof baseValue === "object" && !Array.isArray(baseValue)) {
      result[key] = { ...baseValue, ...overrideValue };
    } else {
      // 原始值和数组：项目覆盖全局
      result[key] = overrideValue;
    }
  }
  return result;
}
```

合并规则：

- **原始值**（string, number, boolean）：项目值覆盖全局值
- **数组**（packages, extensions, skills 等）：项目值**完全替换**全局值（不是追加）
- **嵌套对象**（compaction, retry, terminal 等）：递归合并，项目中指定的子字段覆盖对应全局子字段

最后一条很重要。如果全局设置了 `compaction: { enabled: true, reserveTokens: 16384 }`，项目只设置 `compaction: { keepRecentTokens: 30000 }`，合并结果是 `{ enabled: true, reserveTokens: 16384, keepRecentTokens: 30000 }`。项目不需要重复声明 `enabled` 和 `reserveTokens`。

> 优先级：项目 `settings.json` > 全局 `settings.json` > 内建默认值

Settings 迁移

pi 的配置格式会随版本演进而变化。`migrateSettings` 函数处理旧格式的自动迁移：

```
// packages/coding-agent/src/core/settings-manager.ts:317-352（简化）

static migrateSettings(settings): Settings {
  // queueMode → steeringMode
  if ("queueMode" in settings && !("steeringMode" in settings)) {
    settings.steeringMode = settings.queueMode;
    delete settings.queueMode;
  }

  // websockets: boolean → transport: "sse" | "websocket"
  if (typeof settings.websockets === "boolean") {
    settings.transport = settings.websockets ? "websocket" : "sse";
    delete settings.websockets;
  }

  // skills: { enableSkillCommands, customDirectories } → skills:
string[]
  //  （旧的对象格式迁移为新的数组格式）
  // ...
}
```

迁移在**每次加载时**自动执行，但不会立即回写文件。只有当用户下次修改设置时，新格式才会被持久化。这避免了无谓的文件写入。

持久化与锁

设置的保存使用了文件锁来防止并发写入：

```
// packages/coding-agent/src/core/settings-manager.ts:178-206（简化）

withLock(scope, fn): void {
  const path = scope === "global" ? this.globalSettingsPath :
this.projectSettingsPath;
  let release;
  try {
    if (existsSync(path)) {
      release = this.acquireLockSyncWithRetry(path);
    }
    const current = existsSync(path) ? readFileSync(path, "utf-8") :
undefined;
    const next = fn(current);
    if (next !== undefined) {
      if (!existsSync(dir)) mkdirSync(dir, { recursive: true });
      if (!release) release = this.acquireLockSyncWithRetry(path);
      writeFileSync(path, next, "utf-8");
    }
  } finally {
    if (release) release();
  }
}
```

保存时不是简单地覆盖文件，而是读取当前文件内容，只合并本次会话中修改过的字段（通过 `modifiedFields` 追踪），再写回。这意味着如果用户在另一个 pi 实例中修改了 settings，本实例不会覆盖那些更改。

AGENTS.md 的拼接规则

`AGENTS.md`（或 `CLAUDE.md`）的规则不同于 settings — 它是**拼接**而非覆盖。

目录树上溯发现

`loadProjectContextFiles` 函数从当前工作目录向上搜索，收集路径上所有的 context 文件：

```
// packages/coding-agent/src/core/resource-loader.ts:58-113（简化）

function loadProjectContextFiles(options): Array<{ path; content }> {
  const contextFiles = [];
  const seenPaths = new Set();

  // 1. 先加载全局 context（~/.pi/agent/AGENTS.md 或 CLAUDE.md）
  const globalContext = loadContextFileFromDir(resolvedAgentDir);
  if (globalContext) {
    contextFiles.push(globalContext);
    seenPaths.add(globalContext.path);
  }

  // 2. 从 cwd 向上遍历到根目录
  const ancestorContextFiles = [];
  let currentDir = resolvedCwd;
  while (true) {
    const contextFile = loadContextFileFromDir(currentDir);
    if (contextFile && !seenPaths.has(contextFile.path)) {
      ancestorContextFiles.unshift(contextFile); // 最远的在前
      seenPaths.add(contextFile.path);
    }
    if (currentDir === root) break;
    currentDir = resolve(currentDir, "..");
  }

  // 3. 全局在前，祖先目录从远到近排列
  contextFiles.push(...ancestorContextFiles);
  return contextFiles;
}
```

`loadContextFileFromDir` 在每个目录中依次查找 `AGENTS.md` 和 `CLAUDE.md`，找到第一个就返回。这意味着如果同一个目录同时有 `AGENTS.md` 和 `CLAUDE.md`，只有 `AGENTS.md` 会被加载（它在候选列表中排第一）。

```
// packages/coding-agent/src/core/resource-loader.ts:58-74

function loadContextFileFromDir(dir: string) {
  const candidates = ["AGENTS.md", "CLAUDE.md"];
  for (const filename of candidates) {
    const filePath = join(dir, filename);
    if (existsSync(filePath)) {
      return { path: filePath, content: readFileSync(filePath, "utf-8")
};
    }
  }
  return null;
}
```

最终的拼接顺序：

```
1. ~/.pi/agent/AGENTS.md          ← 全局规则（最先注入）
2. /AGENTS.md                     ← 根目录（如果有）
3. /project/AGENTS.md             ← 项目根目录
4. /project/packages/AGENTS.md    ← 子目录
5. /project/packages/legacy/AGENTS.md  ← 当前工作目录
```

三者**同时生效**，后者可以补充或细化前者的规则。这些文件最终被注入到 system prompt 的 `<project_context>` 区域（v0.75.0 起用 XML 标签包裹，见第 14 章）。

SYSTEM.md 的替换规则

`SYSTEM.md` 的规则又不同 — 它是**替换**而非拼接：

```
如果项目 .pi/SYSTEM.md 存在 → 替换默认 system prompt
否则如果全局 SYSTEM.md 存在 → 替换默认 system prompt
否则 → 使用默认 system prompt
```

```
// packages/coding-agent/src/core/resource-loader.ts:834-846

private discoverSystemPromptFile(): string | undefined {
  const projectPath = join(this.cwd, CONFIG_DIR_NAME, "SYSTEM.md");
  if (existsSync(projectPath)) return projectPath;
  const globalPath = join(this.agentDir, "SYSTEM.md");
  if (existsSync(globalPath)) return globalPath;
  return undefined;
}
```

pi 还支持 `APPEND_SYSTEM.md` — 一个追加到 system prompt 末尾的文件，发现逻辑与 SYSTEM.md 相同（项目优先于全局）。这让用户可以在不替换默认 prompt 的情况下追加内容。

为什么 AGENTS.md 拼接而 SYSTEM.md 替换?

因为它们的语义不同。AGENTS.md 是"额外的规则" — 目录级规则不应该消灭全局规则，而是在全局规则的基础上添加新的约束。SYSTEM.md 是"完全自定义的 system prompt" — 如果用户要自定义 system prompt，通常是想完全控制 prompt 的内容，而不是在默认 prompt 后面追加一段。

PackageSource：外部能力的配置

Settings 中的 `packages` 字段支持两种格式 — 简单字符串和带过滤的对象：

```
// packages/coding-agent/src/core/settings-manager.ts:48-62

type PackageSource =
  | string                          // 加载包的全部资源
  | {
      source: string;               // npm 包名或 git URL
      extensions?: string[];        // 只加载指定 extensions
      skills?: string[];            // 只加载指定 skills
      prompts?: string[];           // 只加载指定 prompts
      themes?: string[];            // 只加载指定 themes
    };
```

这种设计让用户可以安装一个大型的能力包（比如包含 20 个 skills 的社区包），但只启用其中几个。配置示例：

```
{
  "packages": [
    "pi-community-skills",
    { "source": "pi-advanced-tools", "skills": ["tdd", "code-review"] }
  ]
}
```

配置的运行时行为

Getter 中的默认值

SettingsManager 为每个配置项提供 getter 方法，默认值在 getter 中硬编码而非在 Settings 对象中：

```
// packages/coding-agent/src/core/settings-manager.ts:617-644（示例）

getCompactionEnabled(): boolean {
  return this.settings.compaction?.enabled ?? true;
}

getRetrySettings() {
  return {
    enabled: this.getRetryEnabled(),
    maxRetries: this.settings.retry?.maxRetries ?? 3,
    baseDelayMs: this.settings.retry?.baseDelayMs ?? 2000,
  };
}
```

为什么不在构造时填入默认值？因为这样保持了 `globalSettings` 和 `projectSettings` 的"原始状态" — 它们只包含用户显式设置的字段。这对于 `persistScopedSettings` 很重要：保存时只写入用户修改过的字段，不会把默认值写入文件。如果默认值将来改变，用户的配置文件不需要手动更新。

运行时覆盖

除了全局和项目两级， `SettingsManager` 还支持运行时覆盖：

```
// packages/coding-agent/src/core/settings-manager.ts:390-393

applyOverrides(overrides: Partial<Settings>): void {
  this.settings = deepMergeSettings(this.settings, overrides);
}
```

这用于 CLI 参数等临时性的配置。比如 `pi --model gpt-4o` 会在运行时覆盖 `defaultModel`，但不会写入任何配置文件。这构成了实际上的第四级配置：CLI 参数 > 项目 settings > 全局 settings > 默认值。

Reload 机制

当用户在会话中修改了配置文件（比如在另一个终端编辑 `settings.json`），pi 可以通过 `reload()` 方法重新加载：

```
// packages/coding-agent/src/core/settings-manager.ts:362-388（简化）

async reload(): Promise<void> {
  await this.writeQueue;  // 等待未完成的写入
  const globalLoad = SettingsManager.tryLoadFromStorage(this.storage,
"global");
  const projectLoad = SettingsManager.tryLoadFromStorage(this.storage,
"project");
  // 清除修改追踪
  this.modifiedFields.clear();
  this.modifiedNestedFields.clear();
  // 重新合并
  this.settings = deepMergeSettings(this.globalSettings,
this.projectSettings);
}
```

`reload` 先等待写入队列完成（防止读到半写的状态），然后重新从存储加载两级配置。这是一个"热重载"机制 — 用户不需要重启 pi 就能看到配置变更的效果。

编辑哪一层：`pi config -l` 与作用域切换

前面讲的都是配置**如何合并**，但还有一个操作性的问题：当用户想改一个设置，他改的是全局那份还是项目那份？这两份文件路径不同（`~/.pi/agent/settings.json` vs

`{cwd}/.pi/settings.json`），语义也不同 —— 全局是"我的个人偏好"，项目是"这个仓库的约定，会提交进 git"。

pi 用一个显式的作用域概念把这件事摆到台面上。命令行入口是 `pi config`：

```
// package-manager-cli.ts:92,99-103
pi config [-l] [--approve|--no-approve]

不带 -l：编辑全局设置（~/.pi/agent/settings.json）
-l, --local：编辑项目覆盖（.pi/settings.json）
```

`config`（含 `install` / `remove` 等包管理子命令）默认落在**全局**，加 `-l` / `--local` 才写**项目本地**那一层（v0.80.4）。交互式配置编辑器里则用 **Tab 键在"全局 / 项目"两个作用域之间切换** —— 同一个设置项，用户可以清楚地看到、并选择自己在往哪一层写。这解决了三级覆盖的一个隐蔽痛点：如果不告诉用户"你现在改的是哪一层"，他很容易把本该是项目约定的设置写进了全局、或反之。

这一层"项目本地资源"和第 17 章 Resource Loader 的项目本地资源覆盖是同一个作用域概念的两面 —— 一个管 `settings.json` 的分层，一个管 extensions/skills/prompts 的分层，都受同一道 Project Trust 闸门（下节）管辖。

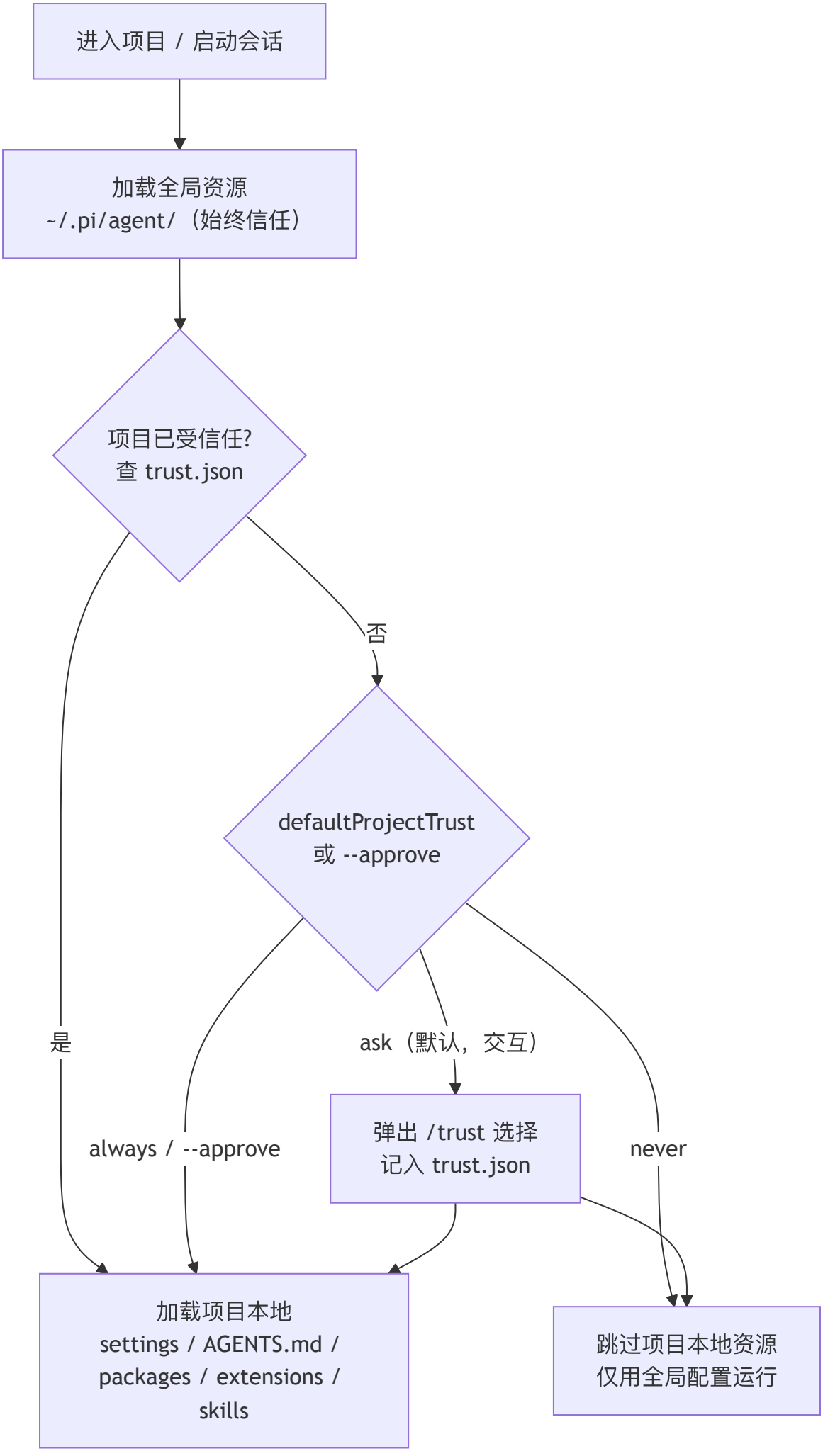
顺带一提本区间新增的几个 settings（都遵循前述 optional + getter 默认值的模式）：`externalEditor`（`Ctrl+G` 外部编辑器命令，优先于 `VISUAL` / `EDITOR`，`settings-manager.ts:97`）、`outputPad`（聊天输出的水平留白，`0 | 1`，`:120`）、`showCacheMissNotices`（在 transcript 里提示明显的 prompt-cache miss，`:96`），以及 `shellPath` 现在支持**开头 `~` 展开**（`:98` / `:880`，Cygwin 等场景更省事）。

第四道闸门：Project Trust

到这里为止，本章一直把"项目级配置覆盖全局"当作无条件的事实 —— 进入一个目录，它的 `.pi/settings.json`、`AGENTS.md`、甚至 `packages` / `extensions` / `skills` 就自动加载并生效。但从 **v0.79.0** 起，这个假设不再成立。

问题出在威胁模型上。前三级覆盖里，项目级配置是**可执行的攻击面**：一个 `.pi/settings.json` 可以通过 `packages` 字段让 pi 从 npm/git 拉取并加载任意代码，`extensions` 可以注入运行时钩子，`AGENTS.md` 能往 system prompt 里塞入任意指令。这意味着——**仅仅 `cd` 进一个 clone 下来的陌生仓库并启动 pi，就可能执行该仓库作者预置的代码**。三级覆盖越方便，这个口子就越危险。

pi 的解法是在"项目本地资源加载"之前插入一道**信任闸门**：


进入项目 / 启动会话
加载全局资源
~/.pi/agent/（始终信任）
项目已受信任?
查 trust.json
是
否
defaultProjectTrust
或 --approve
always / --approve
ask（默认，交互）
弹出 /trust 选择
记入 trust.json
never
加载项目本地
settings / AGENTS.md /
packages / extensions /
skills
跳过项目本地资源
仅用全局配置运行

闸门的关键约定：

1. 全局资源始终信任，项目资源需要决策。 `~/.pi/agent/` 下的配置是用户自己的，无条件加载。只有**项目本地**那一层（`{cwd}/.pi/`）受闸门管辖。这与三级覆盖的优先级是正交的两个维度：优先级回答"谁覆盖谁"，信任回答"项目这一级到底加不加载"。

2. 决策结果持久化在全局。 一旦用户对某个项目作出信任决定，它被写入 `~/.pi/agent/trust.json`（`trust-manager.ts:212`，路径基于 `agentDir`，因此也遵循下文的 `CONFIG_DIR_NAME`）。下次进入同一项目不再询问。注意它存在**全局**而非项目内 —— 否则攻击者只要在自己仓库里预置一个"已信任"标记就能绕过闸门。

3. 三种默认策略 + 一个一次性开关。 全局设置 `defaultProjectTrust` 取 `"ask" | "always" | "never"`（默认 `"ask"`，且**仅全局有效**，`settings-manager.ts:61,95`）：`ask` 在交互模式下弹出 `/trust` 选择器询问；`always` 自动信任所有项目；`never` 一律跳过项目本地资源。命令行上 `--approve / -a` 是一次性开关 —— "本次运行信任项目本地文件"（`args.ts:180`），用于自动化/CI 等非交互场景显式授权。会话内还能用 `/trust` 命令随时改变当前项目的信任状态。

4. extension 可介入信任决策。 闸门会触发 `project_trust` 扩展事件（`types.ts:504`），extension 也能通过 `ctx.isProjectTrusted()`（`types.ts:318`）查询当前项目是否受信任 —— 例如一个企业策略 extension 可以据此决定是否放行。

判定"这个项目是否含有需要信任才能加载的本地资源"由 `hasTrustRequiringProjectResources(cwd)` 完成（`interactive-mode.ts:3281`）：如果项目根本没有 `.pi/` 本地资源，就没有可执行攻击面，闸门直接放行、不打扰用户。

这道闸门改变了第 17 章 Resource Loader 的加载流程 —— 项目本地资源的 `reload` 现在夹在"全局先加载、项目后加载"之间多了一步信任检查（详见第 17 章）。它也是 pi 在"开箱即用的便利"与"陌生仓库的安全"之间做出的明确取舍：默认 `ask` 让便利与安全都不极端。

> **`CONFIG_DIR_NAME` 作为 rebrand 扩展点（v0.70.0）**：本章多处出现的 `.pi` 目录名其实并非硬编码常量，而是从 `config.ts` 导出的 `CONFIG_DIR_NAME`（v0.79.7 起公开导出）。`SYSTEM.md` 发现（`resource-loader.ts`）、`trust.json` 路径、CLI 帮助文本里的示例路径都引用它。这让基于 pi 二次封装、换皮（rebrand）成自有产品的下游能统一改掉配置目录名，而不必到处替换字符串。

取舍分析

得到了什么

1. 零配置启动。不创建任何配置文件，pi 用内建默认值就能工作。所有 Settings 字段都是 optional，默认值在 getter 中硬编码。

2. 渐进式定制。用户可以从全局 settings 开始，遇到特殊项目时加项目配置，遇到特殊目录时加目录规则。复杂度只在需要时引入。

3. 团队共享。项目级的 `.pi/` 目录和 `AGENTS.md` 可以提交到 git，团队成员自动继承项目规则。全局配置保持个人偏好。

4. 并发安全。文件锁 + 只写入修改过的字段，多个 pi 实例可以安全地共享同一个 settings 文件。

放弃了什么

1. 心智负担。三级覆盖意味着用户需要理解"我的这个配置到底从哪来"。当行为不符合预期时，需要检查三个地方（甚至更多，如果目录树上有多个 AGENTS.md）。

2. 不同规则类型的合并语义不同。settings 是深度合并（嵌套对象递归、数组替换）、AGENTS.md 是拼接、SYSTEM.md 是替换 — 三种不同的合并语义增加了理解成本。

3. 没有"dry run"或"explain"命令。用户不能简单地查看"当前生效的完整配置是什么"。需要自己推理合并后的结果。

4. 信任闸门引入了额外的摩擦。Project Trust（v0.79.0）虽然堵上了"进入陌生仓库即执行预置代码"的口子，但代价是首次进入一个带本地资源的项目时多了一次询问。`defaultProjectTrust` 的三种取值本质上是在"安全"与"零打扰"之间让用户自己选位置。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。三级覆盖的核心结构自引入以来稳定，但有几处设计级增补： ① **配置作用域切换**（v0.80.4）： `pi config -l` 编辑项目本地那层、交互编辑器用 Tab 切换全局/项目作用域；与第 17 章项目本地

> 资源覆盖同属一个作用域概念。② **Project Trust 信任闸门**（v0.79.0）：项目本地 settings/resources/extensions/packages/skills 的加载受 `defaultProjectTrust`（ask/always/never）、`--approve` 、 `/trust` 与 `~/.pi/agent/trust.json` 管辖，不再无条件加载。③ **重试配置分层**（v0.70.1）： `retry` 拆为应用层退避与 `retry.provider.*` （透传 SDK），旧 `retry.maxDelayMs` 自动迁移到 `retry.provider.maxRetryDelayMs` 。④ 新增 settings： `externalEditor` 、 `outputPad` 、 `showCacheMissNotices` ， `shellPath` 支持开头 `~` 展开（v0.80.3/v0.80.4/v0.80.6）；此外 httpProxy、httpIdleTimeoutMs、自动 light/dark 主题等持续扩展， `.pi` 目录名由 `CONFIG_DIR_NAME` 提供以支持 rebrand。 `AGENTS.md` / `CLAUDE.md` 双支持、PackageSource 对象格式、 `migrateSettings` 自动迁移均保持不变。

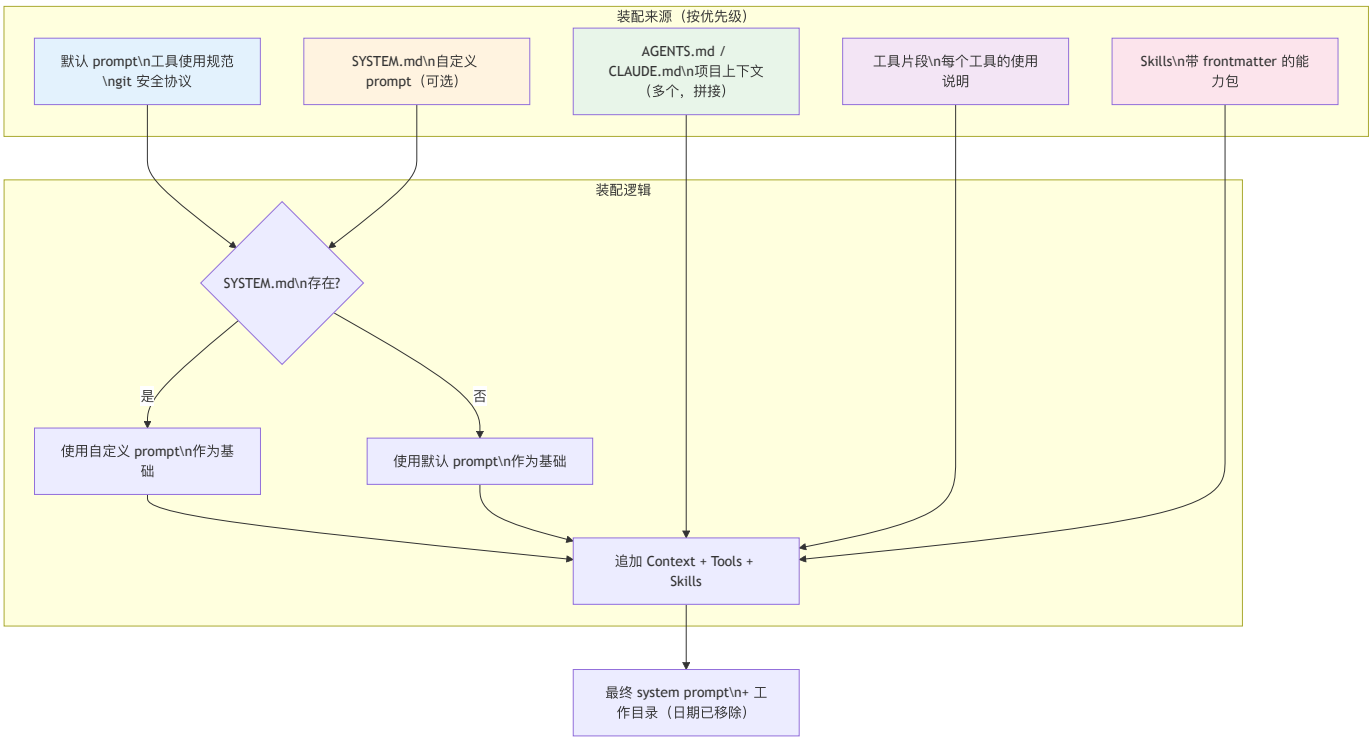
第 14 章：System Prompt 是一套装配流程

> **定位**：本章解析 pi 的 system prompt 如何从多个来源动态拼接而成。 前置依赖：第 13 章（三级配置覆盖）。 适用场景：当你想理解 pi 的 prompt 为什么那么长，或者想定制 prompt 的行为。

Prompt 不是一个字符串

这是本章的核心设计问题。

打开 pi 的 system prompt，你会看到一段几千 token 的文本。但这段文本不是手写的一整块 — 它是从 5 个来源动态装配出来的。



装配入口：buildSystemPrompt

```
// packages/coding-agent/src/core/system-prompt.ts:8-25（接口）

interface BuildSystemPromptOptions {
  customPrompt?: string;          // SYSTEM.md 的内容
  selectedTools?: string[];       // 启用的工具列表
  toolSnippets?: Record<string, string>;  // 工具说明片段
  promptGuidelines?: string[];    // 额外的指引条目
  appendSystemPrompt?: string;    // 追加文本
  cwd?: string;                   // 工作目录
  contextFiles?: Array<{ path: string; content: string }>;  // AGENTS.md
  skills?: Skill[];               // 已发现的 skills
}
```

注意这个函数不做**任何** I/O 操作 — 所有输入都是预加载的。文件发现（搜索 AGENTS.md）、skill 发现（搜索 skill 文件）、配置读取 — 这些都在调用 `buildSystemPrompt` 之前完成。函数本身是纯粹的字符串拼接。

默认 Prompt：非 SYSTEM.md 路径

当用户没有提供 `SYSTEM.md` 时，pi 使用内建的默认 prompt。这是大多数用户的路径，也是 prompt 装配中最复杂的分支。

工具列表注入

默认 prompt 首先构建可用工具的列表。一个工具只有在调用者提供了 `toolSnippets`（一行描述）时才会出现在列表中：

```
// packages/coding-agent/src/core/system-prompt.ts:85-89

const tools = selectedTools || ["read", "bash", "edit", "write"];
const visibleTools = tools.filter(name => !!toolSnippets?.[name]);
const toolsList = visibleTools.length > 0
  ? visibleTools.map(name => `- ${name}: ${toolSnippets!
[name]}`).join("\n")
  : "(none)";
```

这意味着 extension 注册的自定义工具也可以出现在 system prompt 中 — 只要 extension 提供了 `toolSnippets`。最终在 prompt 中呈现为：

```
Available tools:
- read: Read files from the filesystem
- bash: Execute shell commands
- edit: Make targeted edits to existing files
- write: Create or overwrite files
```

Guidelines 动态生成

Guidelines 不是硬编码的列表 — 它根据可用工具动态生成，并去重：

```
// packages/coding-agent/src/core/system-prompt.ts:91-125（简化）

const guidelinesList: string[] = [];
const guidelinesSet = new Set<string>();
const addGuideline = (guideline: string): void => {
  if (guidelinesSet.has(guideline)) return;
  guidelinesSet.add(guideline);
  guidelinesList.push(guideline);
};

const hasBash = tools.includes("bash");
const hasGrep = tools.includes("grep");
const hasFind = tools.includes("find");
const hasLs = tools.includes("ls");

// 只在"没有专用探索工具、只能靠 bash"时才给出指引
if (hasBash && !hasGrep && !hasFind && !hasLs) {
  addGuideline("Use bash for file operations like ls, rg, find");
}

// 注入 extension 提供的额外指引
for (const guideline of promptGuidelines ?? []) {
  const normalized = guideline.trim();
  if (normalized.length > 0) addGuideline(normalized);
}

// 总是包含的基础指引
addGuideline("Be concise in your responses");
addGuideline("Show file paths clearly when working with files");
```

去重机制（`guidelinesSet`）确保即使 extension 提供的 guideline 和内建的重复，也不会出现两次。guideline 的注入顺序是：工具相关的指引 → extension 提供的指引 → 通用指引。

> **设计演化（v0.77.0）**：早期版本里这里还有一个 `else if` 分支 —— 当 grep/find/ls 这些专用工具可用时，注入一条"优先用 grep/find/ls 而不是 bash 做文件探索（更快、尊重 .gitignore）"的指引。这条指引后来被删除了。原因是它属于"用 prompt 教模型挑工具"，而 pi 的判断是：工具的取舍应该由**工具是否被注册**来表达，而不是靠一句容易被忽略的自然语言指引。现在只保留了反向的兜底分支 —— 当一个会话**只有 bash、没有任何专用探索工具**时，才明确告诉模型用 bash 去做 ls/rg/find。换句话说，prompt 不再替模型在"bash vs grep"之间做选择，它只在"无专用工具可用"这一种情形下补一句话。

默认 Prompt 的完整结构

所有部分拼接后，默认 prompt 的结构如下：

```
// packages/coding-agent/src/core/system-prompt.ts:127-143

let prompt = `You are an expert coding assistant operating inside pi, `
+
  `a coding agent harness. You help users by reading files, ` +
  `executing commands, editing code, and writing new files.

Available tools:
${toolsList}

In addition to the tools above, you may have access to other ` +
  `custom tools depending on the project.

Guidelines:
${guidelines}

Pi documentation (read only when the user asks about pi itself, ` +
  `its SDK, extensions, themes, skills, or TUI):
- Main documentation: ${readmePath}
- Additional docs: ${docsPath}
- Examples: ${examplesPath} (extensions, custom tools, SDK)
...`;
```

注意最后一段 — pi 把自己的文档路径注入到了 prompt 中。但只建议 agent 在用户**主动询问 pi 相关话题**时才去读这些文档。这是一个巧妙的设计：不预加载文档内容（节省 token），但告诉 agent 文档在哪里（按需加载）。

appendSystemPrompt 注入

`appendSystemPrompt` 来自 `APPEND_SYSTEM.md` 文件或 extension 的追加文本。它在基础 prompt 之后、context files 之前注入：

```
// packages/coding-agent/src/core/system-prompt.ts:145-147

if (appendSection) {
  prompt += appendSection;
}
```

这是一个"轻量级定制"入口 — 不需要替换整个 system prompt，只需要追加额外的规则。命令行上 `--append-system-prompt` 可以多次传入并依次叠加，而不是后者覆盖前者。

> **装配输入可被扩展检视（v0.68.0 / v0.78.1）**：装配 `buildSystemPrompt` 的那组输入（即 `BuildSystemPromptOptions`）会在 `before_agent_start` 事件中以 `systemPromptOptions` 暴露给 extension，extension 也能通过 `ctx.getSystemPromptOptions()` 主动读取。这让"prompt 由谁拼出来"从黑盒变成可观测、可介入的装配流程。相关的还有两个 CLI 开关：`--no-context-files`（跳过 AGENTS.md/CLAUDE.md 的发现与注入）和 `cwd` 现在是 `buildSystemPrompt` 的**必填**参数 —— 不再回退到 `process.cwd()`，调用方必须显式传入工作目录（这条"去 process-global"的纪律贯穿 pi 的多个子系统，详见第 17 章）。

Context Files 注入

无论是默认 prompt 还是自定义 prompt，context files（AGENTS.md / CLAUDE.md）的注入逻辑相同：

```
// packages/coding-agent/src/core/system-prompt.ts:153-160

if (contextFiles.length > 0) {
  prompt += "\n\n<project_context>\n\n";
  for (const { path: filePath, content } of contextFiles) {
    prompt += `<project_instructions path="${filePath}">\n` +
      `${content}\n</project_instructions>\n\n`;
  }
  prompt += "</project_context>\n";
}
```

每个 context file 作为一个独立的 `<project_instructions>` 块注入，文件路径放在 `path` 属性里，整体再包进一个 `<project_context>` 标签。这样 LLM 能看到规则来自哪个文件，有助于在规则冲突时理解优先级。

> **设计演化（v0.75.0）**：早期版本用 Markdown 标题包裹项目上下文 —— `# Project Context` 作为大标题，每个文件用 `## ${path}` 作为二级标题。问题在于：当 `AGENTS.md` 自身的内容里也含有 Markdown 标题（几乎必然如此）时，模型很难分清哪些标题是"边界标记"、哪些是"文件内容"。改用 XML 标签后，边界有了明确且不会与 Markdown 正文混淆的开闭标记。这与下文 skills 注入用 `<available_skills>` 是同一个判断 —— **prompt 里凡是需要机器可靠识别的边界，一律用 XML 标签，不用 Markdown 标题**。

context files 的加载顺序（在第 13 章详述）决定了它们在 prompt 中的顺序 — 全局在前，当前目录在后。由于 LLM 的近因偏差（recency bias），后面的内容通常更受重视，这恰好与我们的优先级期望一致：目录级规则 > 项目级规则 > 全局规则。

Skills 注入

Skills 的注入稍复杂一些。`formatSkillsForPrompt` 将 skill 列表转换为结构化的 XML 格式：

```
// packages/coding-agent/src/core/skills.ts:339-365

function formatSkillsForPrompt(skills: Skill[]): string {
  const visibleSkills = skills.filter(s => !s.disableModelInvocation);

  if (visibleSkills.length === 0) return "";

  const lines = [
    "\n\nThe following skills provide specialized instructions for
specific tasks.",
    "Use the read tool to load a skill's file when the task matches its
description.",
    "When a skill file references a relative path, resolve it against "
+
      "the skill directory and use that absolute path in tool
commands.",
    "",
    "<available_skills>",
  ];

  for (const skill of visibleSkills) {
    lines.push("  <skill>");
    lines.push(`    <name>${escapeXml(skill.name)}</name>`);
    lines.push(`    <description>${escapeXml(skill.description)}
</description>`);
    lines.push(`    <location>${escapeXml(skill.filePath)}</location>`);
    lines.push("  </skill>");
  }

  lines.push("</available_skills>");
  return lines.join("\n");
}
```

几个设计要点值得注意：

1. 只注入元数据，不注入内容。每个 skill 只注入 name、description 和 location。Skill 的完整内容（可能有几千行）不会出现在 system prompt 中。LLM 根据 description 判断是否需要加载某个 skill，然后用 `read` 工具读取完整内容。这是一种典型的"延迟加载"策略 — 把 skill 发现和 skill 使用分开。

2. XML 格式。使用 `<available_skills>` / `<skill>` 这样的 XML 标签而不是 Markdown 或 JSON。XML 在 prompt 中是一个非常好的结构化格式 — 它有明确的开始和结束标记，不会与 Markdown 混淆，LLM 对 XML 的解析也非常可靠。

3. `disableModelInvocation` 过滤。有些 skill 被标记为不允许模型自动调用（可能因为它们有副作用或者只用于手动触发）。这些 skill 不会出现在 prompt 中。

4. 路径解析指引。prompt 中明确告诉 LLM：如果 skill 文件引用了相对路径，要相对于 skill 目录解析。这避免了路径混乱的问题。

为什么 Skills 需要 read 工具

代码中有一个细微的条件判断。在默认 prompt 路径中：

```
// packages/coding-agent/src/core/system-prompt.ts:159-161

if (hasRead && skills.length > 0) {
  prompt += formatSkillsForPrompt(skills);
}
```

在自定义 prompt 路径中：

```
// packages/coding-agent/src/core/system-prompt.ts:66-69

const customPromptHasRead = !selectedTools ||
selectedTools.includes("read");
if (customPromptHasRead && skills.length > 0) {
  prompt += formatSkillsForPrompt(skills);
}
```

只有当 `read` 工具可用时才注入 skills。原因是：skill 是指向文件的指针（"这个 skill 的内容在 `~/.pi/skills/tdd.md`"），不是内联的全文。agent 需要用 `read` 工具来读取 skill 的完整内容。如果 `read` 工具不可用（极少数受限场景），注入 skill 列表反而会误导 agent — 它知道有这些 skill 存在，但没法读取它们。

尾部注入：工作目录（以及被移除的日期）

无论走哪条路径（默认 prompt 或自定义 prompt），最后都会追加工作目录：

```
// packages/coding-agent/src/core/system-prompt.ts:159

prompt += `\nCurrent working directory: ${promptCwd}`;
return prompt;
```

工作目录放在最后有一个实际考量：它是 prompt 中最容易被模型注意到的信息（近因偏差），且是模型执行文件操作时最需要的上下文 —— 知道当前在哪个目录下工作，才能正确解析相对路径。Windows 路径中的反斜杠会被替换为正斜杠（`cwd.replace(/\\/g, "/")`，`system-prompt.ts:39`），因为 LLM 在处理路径时对正斜杠更可靠（反斜杠容易被解释为转义字符）。

> **移除当前日期以稳定 prompt cache（v0.80.7，行为变化）**：这里曾经还有一行 `prompt += \`\nCurrent date: ${date}\`` 。它被**删掉了**，理由值得展开。system prompt 是发给模型的**第一段、也是最长的固定前缀**，provider 端的 prompt caching 正是靠"前缀逐字节相同"来命中缓存的。而"当前日期"每天都会变 —— 于是每跨过一次午夜，system prompt 前缀就变了一个字节，**整块缓存作废**，第二天第一次请求要重新按未缓存价格计费、且更慢。为一个模型很少真正用到的信息，赔上一个每日必然发生的缓存失效，并不划算。取舍很直白：**得到**稳定的、跨日不失效的 prompt cache 前缀；**放弃**让模型"知道今天几号"（真需要时它可以用 bash 跑 `date`，或由用户/AGENTS.md 显式提供）。这与本章反复出现的判断一致 —— prompt 里凡是会频繁变动的内容，都要为它对缓存的破坏付出代价，值不值得要单独算账。

自定义 Prompt 路径

如果用户提供了 `SYSTEM.md`（通过 `customPrompt` 参数），装配逻辑简化了很多：

```
// packages/coding-agent/src/core/system-prompt.ts:49-76（完整的 customPrompt 分支）

if (customPrompt) {
  let prompt = customPrompt;

  // 追加 appendSystemPrompt
  if (appendSection) prompt += appendSection;

  // 追加项目上下文文件（同样用 XML 标签包裹）
  if (contextFiles.length > 0) {
    prompt += "\n\n<project_context>\n\n";
    for (const { path, content } of contextFiles) {
      prompt += `<project_instructions path="${path}">\n` +
        `${content}\n</project_instructions>\n\n`;
    }
    prompt += "</project_context>\n";
  }

  // 追加 skills（需要 read 工具可用）
  if (customPromptHasRead && skills.length > 0) {
    prompt += formatSkillsForPrompt(skills);
  }

  // 最后追加工作目录（当前日期已于 v0.80.7 移除）
  prompt += `\nCurrent working directory: ${promptCwd}`;
  return prompt;
}
```

关键区别：自定义 prompt 路径**没有**默认 prompt 中的工具列表、guidelines 生成、pi 文档路径注入。用户完全控制 prompt 的基础部分。但 context files、skills、工作目录仍然会自动追加 — 因为这些是"环境信息"，不管 prompt 怎么定制，agent 都需要知道当前的项目规则和可用能力。

这种"基础可替换、环境自动追加"的设计在两个需求之间取得了平衡：

- 用户想完全控制 prompt 的核心指令（"你是一个代码审查专家"）
- 系统需要确保 agent 能看到项目规则和可用工具（这些不应该被用户误删）

这也解释了为什么 `buildSystemPrompt` 的参数设计中，`customPrompt` 和 `contextFiles` / `skills` 是独立的。如果自定义 prompt 和环境信息耦合在一起，用户要么全盘接管（包括 AGENTS.md 和 skills 的注入逻辑），要么完全不能定制。分离参数让"基础替换"和"环境注入"成为正交的两个维度。

Context Files 的加载时机

一个常见的问题是：context files 是在 session 启动时加载一次，还是每次构建 prompt 时重新加载？

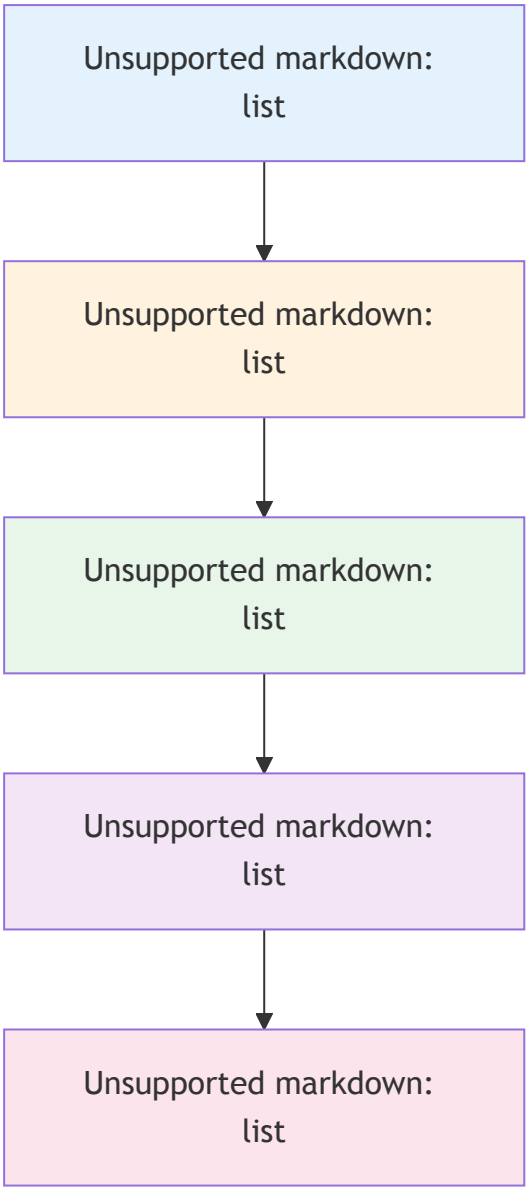
答案是前者。`buildSystemPrompt` 接收预加载的 `contextFiles` 数组，不做任何文件 I/O。但 `ResourceLoader` 的 `reload()` 方法会重新发现和加载 context files：

```
// packages/coding-agent/src/core/resource-loader.ts:451-453（在 reload 中）

const agentsFiles = {
  agentsFiles: loadProjectContextFiles({ cwd: this.cwd, agentDir: this.agentDir })
};
```

这意味着如果用户在会话中创建了一个新的 `AGENTS.md` 文件，它不会自动生效 — 需要等到下一次 `reload`（比如 settings 变更时触发）。这是一个有意的设计权衡：避免每轮对话都进行文件系统遍历，但代价是 context files 的变更不是实时的。

对于大多数使用场景来说这不是问题，因为 AGENTS.md 通常在 session 开始前就已经存在。但如果用户需要在会话中途修改项目规则，他们需要知道这个延迟。

完整装配顺序总结



每一层的注入都是 append — 后面的内容追加在前面的后面。这个设计决定了 LLM 看到信息的顺序，也隐含了优先级：越后面的信息越容易被 LLM 重视。

取舍分析

得到了什么

1. 无限可定制。从完全替换 system prompt（SYSTEM.md）到精细追加规则（AGENTS.md），用户可以控制 prompt 的任何部分。

2. Prompt 随上下文变化。不同的工作目录可能有不同的 AGENTS.md，不同的项目可能有不同的 skills。Prompt 自动适应当前环境。

3. 关注点分离。默认 prompt 管工具使用规范，AGENTS.md 管项目规则，skills 管能力扩展。每个来源负责自己的领域。

4. 纯函数设计。`buildSystemPrompt` 不做任何 I/O，所有输入都是预加载的。这使得它易于测试、可预测、不会有副作用。所有的文件发现、配置读取都在调用链的上游完成。

放弃了什么

1. Prompt 的最终形态难以预测。5 个来源动态拼接，用户不容易知道 LLM 到底收到了什么 prompt。当 agent 行为不符合预期时，需要检查多个来源。

2. Prompt 可能很长。多个 AGENTS.md + 多个 skills + 默认规范 + 追加文本，最终 prompt 可能有几千 token。这压缩了留给实际对话的 context 空间。

3. 拼接顺序隐含了优先级。后来的内容在 prompt 的后面，LLM 倾向于更重视后面的内容（近因偏差）。这意味着 AGENTS.md 的规则比默认 prompt 的规则更容易被遵守 — 这是有意的设计，但如果两者矛盾，行为可能不直觉。

4. 自定义 prompt 失去工具指引。选择 SYSTEM.md 路径的用户不会得到自动生成的 guidelines 和工具列表。如果他们忘记在 SYSTEM.md 中说明工具用法，agent 可能不知道怎么正确使用工具。这是完全控制的代价。

与其他章节的联系

Prompt 装配是 pi 产品层的"信息入口" — 它决定了 LLM 在整个会话中看到的第一段文本。理解这个装配过程有助于理解后续章节中的多个机制：

- **第 12 章的 compaction**：压缩后的摘要替换旧消息，但 system prompt 保持不变。这意味着 AGENTS.md 中的规则在压缩后仍然有效 — 它们不在消息历史中，而在 system prompt 中。
- **第 13 章的配置层**：Settings 控制 prompt 的间接行为（比如启用哪些工具决定了 `selectedTools`），而 AGENTS.md 和 SYSTEM.md 直接影响 prompt 内容。两个系统互补但不重叠。
- **第 15 章的 extension**：Extension 通过 `toolSnippets`、`promptGuidelines`、`appendSystemPrompt` 三个维度影响 prompt 装配。Extension 不能替换默认 prompt，但可以在其基础上追加工具、指引和自定义文本。

- **第 16 章的 skills**：Skills 是 prompt 中最"轻量"的部分 — 只注入一个列表，完整内容按需加载。这种延迟加载策略使得即使有几百个 skill，prompt 的长度也不会失控。

整个装配流程的哲学可以总结为一句话：**system prompt 是 agent 的"宪法"，它不应该被频繁修改，但应该能够反映当前环境的全部约束**。默认 prompt 提供通用宪法条款，AGENTS.md 提供地方法规，skills 提供执业指南，日期和工作目录提供当前国情。每一层都有明确的职责，每一层都可以独立定制。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。prompt 装配有几处设计级变化： ① **默认系统提示词移除"当前日期"**（v0.80.7，行为变化）：尾部只剩 `Current working directory`，动机是避免每天跨午夜时 prompt-cache 前缀失效 —— 得到跨日稳定的缓存，放弃让模型直接知道日期。 ② 项目上下文的边界从 Markdown 标题（`# Project Context` / `## ${path}`）改为 XML 标签（`<project_context>` / `<project_instructions path="...">`，v0.75.0），动机是避免与 AGENTS.md 正文里的标题混淆 —— 这与 skills 注入用 `<available_skills>` 是同一判断。 ③ "优先用 grep/find/ls 而非 bash"的指引在 v0.77.0 被删除，prompt 不再替模型在工具间做选择，只在"仅有 bash"时兜底。 此外，装配输入在 `before_agent_start` 通过 `systemPromptOptions` 暴露给 extension（v0.68.0），`cwd` 成为必填（去 process.cwd() 兜底）；扩展工具变更在同一 run 内下次请求前生效、不丢弃 `before_agent_start` 的系统提示词覆盖（v0.80.3）。 `buildSystemPrompt` 的纯函数设计（不做 I/O）、`formatSkillsForPrompt` 的 XML 注入、`APPEND_SYSTEM.md` 轻量追加这三点自始至终保持不变。

第 15 章：Extension 系统 — 让产品长出新器官

> **定位**：本章解析 pi 的 extension 系统如何在不修改核心代码的前提下扩展产品能力。 前置依赖：第 10 章（Agent 类）、第 14 章（System Prompt 装配）。 适用场景：当你想理解"能力外置"的具体实现，或者想为 pi 写 extension。

哪些能力应该内建，哪些应该外置？

这是本章的核心设计问题。

pi 的回答是极端的：**几乎所有产品级能力都外置。** 核心只做三件事 — 调模型、跑循环、管状态（第 8-10 章）。其余能力 — 新工具、新命令、新快捷键、自定义 UI — 全部通过 extension 系统提供。

Extension 的 API 面（`ExtensionUIContext + ExtensionApi`）定义了 extension 能触达的系统表面积。让我们看看这个表面积有多大。

Extension 能做什么

Extension 是一个 TypeScript 模块，导出一个工厂函数，接收 `ExtensionAPI` 对象：

```
// extension 的基本结构
// extensions/types.ts:1273
export type ExtensionFactory = (pi: ExtensionAPI) => void |
Promise<void>;
```

注意这个签名的两个设计选择：工厂函数支持 `async`（extension 可以在 setup 阶段做 I/O），参数名是 `pi` 而不是 `ctx`（强调这是系统级别的扩展点）。

`ExtensionAPI` 暴露的能力极其丰富。我们不逐一列举全部方法（接口超过 200 行），而是看 5 个最重要的能力维度。

能力 1：事件订阅 — 观察系统的一切

```
// extensions/types.ts:986-1026（节选 5 个关键事件）
export interface ExtensionAPI {
  on(event: "session_start",
     handler: ExtensionHandler<SessionStartEvent>): void;
  on(event: "context",
     handler: ExtensionHandler<ContextEvent, ContextEventResult>): void;
  on(event: "tool_call",
     handler: ExtensionHandler<ToolCallEvent, ToolCallEventResult>):
void;
  on(event: "tool_result",
     handler: ExtensionHandler<ToolResultEvent, ToolResultEventResult>):
void;
  on(event: "input",
     handler: ExtensionHandler<InputEvent, InputEventResult>): void;
  // ... 二十多种事件，且随版本持续增加
}
```

事件系统的设计有两层含义。**观察型事件**（如 `session_start`、`agent_end`、`message_update`）让 extension 被动获知系统状态变化。**干预型事件**（如 `tool_call`、`input`、`context`）通过返回值影响系统行为 — `tool_call` 可以 block 工具执行，`input` 可以 transform 用户输入，`context` 可以修改发送给 LLM 的消息列表。

> **事件面持续扩张**：早期版本约有 26 种事件，现已增至二十多种并仍在增长，因此本书不再钉死具体数字。值得注意的是几个较新的 hook：`after_provider_response`（v0.67.6，能拿到 provider 返回的 HTTP 状态码与响应头，types.ts:687）、`message_end`（其返回值 `MessageEndEventResult` 可以替换一条刚 finalize 的消息，types.ts:751）、配合第 13 章信任闸门的 `project_trust`（types.ts:514）。本区间又新增了三个：`before_provider_headers`（types.ts:681，在请求发出前改写 provider 的 HTTP 头 —— 干预型）、`agent_settled`（types.ts:718，agent 彻底停下、队列排空后触发 —— 适合做"一切就绪再收尾"的动作）、`session_info_changed`

> （types.ts:566，会话元信息如显示名变化时触发 —— 观察型）。它们大多属于"通过返回值改变系统行为"或"在精确生命周期点收尾"这两类，而非单纯观察。

这个区分体现在 handler 的泛型签名中：

```
// extensions/types.ts:981
export type ExtensionHandler<E, R = undefined> =
  (event: E, ctx: ExtensionContext) => Promise<R | void> | R | void;
```

当 `R = undefined` 时，handler 是纯观察型 — 返回值被忽略。当 `R` 是具体类型时（如 `ToolCallEventResult`），handler 可以通过返回值干预流程。

能力 2：工具注册 — 给 LLM 新的手

```
// extensions/types.ts:1032-1035
registerTool<TParams extends TSchema, TDetails = unknown, TState = any>(
  tool: ToolDefinition<TParams, TDetails, TState>,
): void;
```

`ToolDefinition` 是 extension 系统中最复杂的类型，因为它承载了从 LLM 交互到 UI 渲染的完整工具定义：

```
// extensions/types.ts:369-404（核心字段）
export interface ToolDefinition<TParams extends TSchema = TSchema,
                                 TDetails = unknown, TState = any> {
  name: string;
  description: string;           // 给 LLM 看的描述
  parameters: TParams;           // TypeBox schema → JSON Schema
  promptSnippet?: string;        // 注入 system prompt 的单行摘要
  promptGuidelines?: string[];   // 注入 system prompt 的使用指南

  execute(
    toolCallId: string,
    params: Static<TParams>,
    signal: AbortSignal | undefined,
    onUpdate: AgentToolUpdateCallback<TDetails> | undefined,
    ctx: ExtensionContext,
  ): Promise<AgentToolResult<TDetails>>;

  renderCall?: (...) => Component;   // 自定义调用时的 UI
  renderResult?: (...) => Component; // 自定义结果的 UI
}
```

注意 `promptSnippet` 和 `promptGuidelines` — 工具不仅有 schema 描述，还能直接向 system prompt 注入使用指南。这让工具的"说明书"和"实现"在同一个定义中完成。

> **参数 schema 用 `typebox`（v0.69.0）**：`parameters` 字段是一个 TypeBox schema，`Static<TParams>` 把它映射回静态 TS 类型。这里有一个容易被忽略的迁移：extension 现在应从 **`typebox`** 导入 `Type` / `Static` / `TSchema`（核心代码即 `import type { Static, TSchema } from "typebox"`，types.ts:42），而不再是旧的 `@sinclair/typebox`（0.34）。换到 `typebox` 1.x 的动机是它自带校验与转换、不依赖 `eval`，因而在禁用 `eval` 的运行时（如 Cloudflare Workers）也能真正执行工具参数校验，而不是静默跳过。旧的 `@sinclair/typebox` 仍作为向后兼容别名保留（见下文 VIRTUAL_MODULES）。

> **工具选择是"名字白名单"（v0.68.0）**：extension 用 `registerTool` 注册的工具，会和内建工具汇入同一个工具集。而宿主（SDK 调用方或 CLI）**选择启用哪些工具**的方式，在 v0.68.0 改成了**字符串名字白名单** —— `createAgentSession({ tools: ["read", "bash", "my_tool"] })`（`tools?: string[]`，sdk.ts:67）。早期那种导出预构建工具对象（`readTool` / `codingTools`）的方式被移除，改为

> 按需用工厂构造（`createReadTool(cwd)` 等，需要显式传入 cwd）。白名单里写内建工具名就启用内建工具，写 extension 注册的工具名就启用该 extension 工具 —— 内建与扩展工具在选择层面一视同仁。工具激活模型的完整设计见第 19 章。

> **cache-friendly 动态工具加载（v0.80.7）**：extension 不一定在启动时就把工具全注册好 —— 它可以在**运行中**（比如某个 tool result 到达后）再激活新工具。问题是：工具定义是 system prompt 的一部分，中途加工具会改变发给模型的前缀，进而**打掉 prompt cache**。v0.80.7 让受支持的 Anthropic 与 OpenAI Responses 模型能在"工具变得可用的位置"加载定义，而**保留已缓存的 prompt 前缀**不失效（配合 Kimi 的 native deferred tool loading）。这批新工具名如何随 `addedToolNames` 在循环里被解析并传播，见第 9 章。这与第 14 章"移除当前日期以稳定 prompt cache"是同一条纪律的两面 —— 凡是会动到 prompt 前缀的改动，都要盘算它对缓存的代价。

能力 3：命令与快捷键 — 用户交互的扩展点

```
// extensions/types.ts:1042-1061
registerCommand(name: string,
  options: Omit<RegisteredCommand, "name" | "sourceInfo">): void;

registerShortcut(shortcut: KeyId,
  options: {
    description?: string;
    handler: (ctx: ExtensionContext) => Promise<void> | void;
  }): void;

registerFlag(name: string,
  options: {
    description?: string;
    type: "boolean" | "string";
    default?: boolean | string;
  }): void;
```

三个注册方法对应三种用户交互通道：斜杠命令（`/command`）、键盘快捷键、CLI flag。注意 `registerCommand` 的 handler 接收的是 `ExtensionCommandContext` — 比普通的 `ExtensionContext` 多了 `newSession()`、`fork()`、`navigateTree()` 等会话控制方

法。这个区分很重要：只有**用户主动发起的命令**才有权做会话级操作，事件 handler 中不能 fork 或切换会话。

会话替换后 ctx 失效：withSession 模式（v0.69.0）

这里有一个极易踩的坑。`newSession()` 、 `fork()` 、 `switchSession()` 会**替换底层会话对象** —— 调用之后，你手上原来的 `pi` / `ctx` 已经绑定在旧会话上，再用它做后续操作会抛错。正确做法是把"会话切换后要做的事"放进 `withSession` 回调，由系统在新会话就绪后回调给你一个全新的 `ReplacedSessionContext`：

```
// extensions/types.ts:356,380（withSession 选项）
await ctx.fork(entryId, {
  position: "before",  // 从该 user 消息之前分叉
  withSession: async (newCtx: ReplacedSessionContext) => {
    // 在这里用 newCtx（而不是旧 ctx）继续操作新会话
    newCtx.sendUserMessage("继续这个分支");
  },
});
```

`ReplacedSessionContext` （types.ts:380）专门作为 `newSession()` / `fork()` / `switchSession()` 的回调参数存在。规则可以浓缩成一句：**一旦你触发了会话替换，旧的 ctx 就作废了，后续逻辑只能走 `withSession` 。**这是 extension 开发中最常见的运行时错误来源之一，却最不直觉。

能力 4：消息与状态持久化

```
// extensions/types.ts:1078-1093
sendMessage<T = unknown>(
  message: Pick<CustomMessage<T>, "customType" | "content" |
           "display" | "details">,
  options?: { triggerTurn?: boolean;
              deliverAs?: "steer" | "followUp" | "nextTurn" },
): void;

sendUserMessage(
  content: string | (TextContent | ImageContent)[],
  options?: { deliverAs?: "steer" | "followUp" },
): void;

appendEntry<T = unknown>(customType: string, data?: T): void;
```

三种消息注入方式形成梯度： `sendMessage` 发送自定义消息（可控制是否触发 LLM 回复）， `sendUserMessage` 模拟用户输入（总是触发回复）， `appendEntry` 只写入会话文件但不发给 LLM（纯持久化，用于 extension 保存自己的状态）。

`deliverAs` 参数控制消息在 agent 正在 streaming 时的排队策略： `steer` 立即注入当前 turn， `followUp` 等当前 turn 结束后注入。

能力 5：Provider 注册 — 自定义模型接入

```
// extensions/types.ts:1192-1207
registerProvider(name: string, config: ProviderConfig): void;
unregisterProvider(name: string): void;
```

这是最强大的扩展点之一。Extension 可以注册全新的 model provider（带自定义 baseUrl、API key、甚至 OAuth 流程），也可以覆盖已有 provider 的配置（比如把所有 Anthropic 请求路由到内部代理）。详见第 18 章。

> **完整 provider 扩展（v0.81.0）**：此前 extension 在模型侧能做的相对有限 —— 主要是改模型定义、加工具。v0.81.0 起 extension 可以注册一个**完整的 pi-ai provider**：自带认证、模型刷新（ `refreshModels` ）、模型过滤、乃至自定义流式实现。也就是说 extension 不再只是"往已有 provider 上打补丁"，而能整个接管一类模型的接入 —— 一个企业内部网关、一个自研推理服务，都能作为一等 provider 挂进 pi 的 `Models` 运行时（第 18 章 `ModelRuntime` 装配 provider 时把 extension 注册一并纳入）。

ExtensionUIContext — 系统表面上的 UI 接口

除了 `ExtensionAPI` （在 setup 阶段使用），extension 还通过 `ExtensionUIContext` 与用户交互。这个接口在每个事件 handler 的 `ctx.ui` 中可用。

```
// extensions/types.ts:108-175（核心 UI 方法）
export interface ExtensionUIContext {
  // 对话式 UI — 阻塞等待用户回应
  select(title: string, options: string[],
         opts?: ExtensionUIDialogOptions): Promise<string | undefined>;
  confirm(title: string, message: string,
          opts?: ExtensionUIDialogOptions): Promise<boolean>;
  input(title: string, placeholder?: string,
        opts?: ExtensionUIDialogOptions): Promise<string | undefined>;

  // 单向通知
  notify(message: string,
         type?: "info" | "warning" | "error"): void;

  // 持久 UI 元素 — 自定义系统外观
  setWidget(key: string,
    content: string[] | undefined,
    options?: ExtensionWidgetOptions): void;
  setFooter(factory: ((tui, theme, footerData) =>
    Component & { dispose?(): void }) | undefined): void;
  setHeader(factory: ((tui, theme) =>
    Component & { dispose?(): void }) | undefined): void;
}
```

UI 方法分两类。**对话式方法**（`select`、`confirm`、`input`）返回 Promise，会暂停 extension 的执行直到用户做出选择。它们都支持 `AbortSignal` 和 `timeout`（倒计时自动关闭），这样 extension 不会无限阻塞系统。

持久元素方法（`setWidget`、`setFooter`、`setHeader`）改变的是系统 UI 的结构。`setWidget` 可以在编辑器上方或下方插入自定义面板。`setFooter` 和 `setHeader` 直接替换系统的页眉页脚。这些方法接受 Component factory（而非 Component 实例），因为 TUI 的组件生命周期由系统管理。

`setEditorComponent` 是最激进的扩展点 — extension 可以完全替换输入编辑器。类型文件中甚至给出了 Vim mode 的示例实现。

一个真实的 Extension 长什么样

以下是一个 token 统计 extension 的 setup 逻辑，展示了典型的 extension 结构：

```
// 一个真实的 extension 示例
import type { ExtensionAPI } from "@earendil-works/pi-coding-agent";

export default async function setup(pi: ExtensionAPI) {
  let turnCount = 0;

  // 1. 注册事件监听 — 追踪 turn 计数
  pi.on("turn_end", async (event, ctx) => {
    turnCount++;
    ctx.ui.setStatus("turns", `Turns: ${turnCount}`);
  });

  // 2. 注册事件监听 — 在 compaction 前保存状态
  pi.on("session_before_compact", async (event, ctx) => {
    pi.appendEntry("turn-counter-state", { turnCount });
  });

  // 3. 注册命令 — 提供用户交互
  pi.registerCommand("reset-turns", {
    description: "Reset the turn counter",
    handler: async (args, ctx) => {
      const confirmed = await ctx.ui.confirm(
        "Reset?", `Reset turn counter (currently ${turnCount})?`
      );
      if (confirmed) {
        turnCount = 0;
        ctx.ui.setStatus("turns", `Turns: 0`);
        ctx.ui.notify("Turn counter reset", "info");
      }
    },
  });
}
```

这个例子展示了三个典型模式：用闭包变量维护状态（`turnCount`），用事件监听驱动行为，用 `appendEntry` 持久化到会话文件。

Loader → Runner → Wrapper 三层架构

Extension 系统内部分为三层，各有明确职责。

Loader：发现与加载

```
// extensions/loader.ts:373-390
export async function loadExtensions(
  paths: string[], cwd: string, eventBus?: EventBus
): Promise<LoadExtensionsResult> {
  const extensions: Extension[] = [];
  const errors: Array<{ path: string; error: string }> = [];
  const runtime = createExtensionRuntime();

  for (const extPath of paths) {
    const { extension, error } =
      await loadExtension(extPath, cwd, resolvedEventBus, runtime);
    if (error) { errors.push({ path: extPath, error }); continue; }
    if (extension) { extensions.push(extension); }
  }

  return { extensions, errors, runtime };
}
```

Loader 的关键设计：

1. **串行加载**。Extension 按配置顺序逐个加载，不并行。这保证了注册顺序的确定性 — 先加载的 extension 的事件 handler 先执行。

2. **jiti 动态导入**。Extension 是普通的 TypeScript 文件，通过上游 `jiti`（当前 `2.7.0`，支持 `virtualModules` 别名映射）在运行时编译和加载。这意味着 extension 不需要预编译。不过对 SDK 宿主，现在还有一条**内联**路径：`InlineExtension`（`types.ts:1491`，从包根导出）让宿主直接把一个 extension 对象交给 `createAgentSession`，不必先落一个磁盘文件再指路径 —— 适合"为一次嵌入临时挂一个 hook"的场景（第 26b 章）。

3. **Virtual Modules**。编译后的 Bun binary 中没有 `node_modules`，extension 依赖的包通过 `virtualModules` 映射到 binary 中打包的静态导入：

```
// extensions/loader.ts:44-61
const VIRTUAL_MODULES: Record<string, unknown> = {
  typebox: _bundledTypebox,
  "@sinclair/typebox": _bundledTypebox,  // 旧名，向后兼容别名
  "@earendil-works/pi-agent-core": _bundledPiAgentCore,
  "@earendil-works/pi-tui": _bundledPiTui,
  "@earendil-works/pi-ai": _bundledPiAi,
  "@earendil-works/pi-coding-agent": _bundledPiCodingAgent,
  // 旧 scope 名作为向后兼容别名同时注册：
  "@mariozechner/pi-agent-core": _bundledPiAgentCore,
  "@mariozechner/pi-tui": _bundledPiTui,
  "@mariozechner/pi-ai": _bundledPiAi,
  "@mariozechner/pi-coding-agent": _bundledPiCodingAgent,
};
```

这些 import 必须是静态的（`import * as`），否则 Bun 不会打包。这是一个不太常见的"静态依赖支撑动态加载"的模式。注意 loader 同时注册了 `@earendil-works/pi-*`（当前 canonical 名）与 `@mariozechner/pi-*`（迁移前的旧名）两套 key —— 这样 scope 迁移后，仍引用旧包名的存量 extension 不必改动即可继续解析。

Runtime：两阶段初始化

Runtime 是 extension 和核心系统之间的共享状态层。它的设计核心是**两阶段初始化**：

```
// extensions/loader.ts:120-154（简化）
export function createExtensionRuntime(): ExtensionRuntime {
  const notInitialized = () => {
    throw new Error("Extension runtime not initialized.");
  };

  const runtime: ExtensionRuntime = {
    sendMessage: notInitialized,    // 抛异常的 stub
    sendUserMessage: notInitialized,
    // ...所有 action 方法都是 throwing stubs
    flagValues: new Map(),
    pendingProviderRegistrations: [],
    registerProvider: (name, config, extensionPath) => {
      runtime.pendingProviderRegistrations.push(
        { name, config, extensionPath }
      );
    },
  };
  return runtime;
}
```

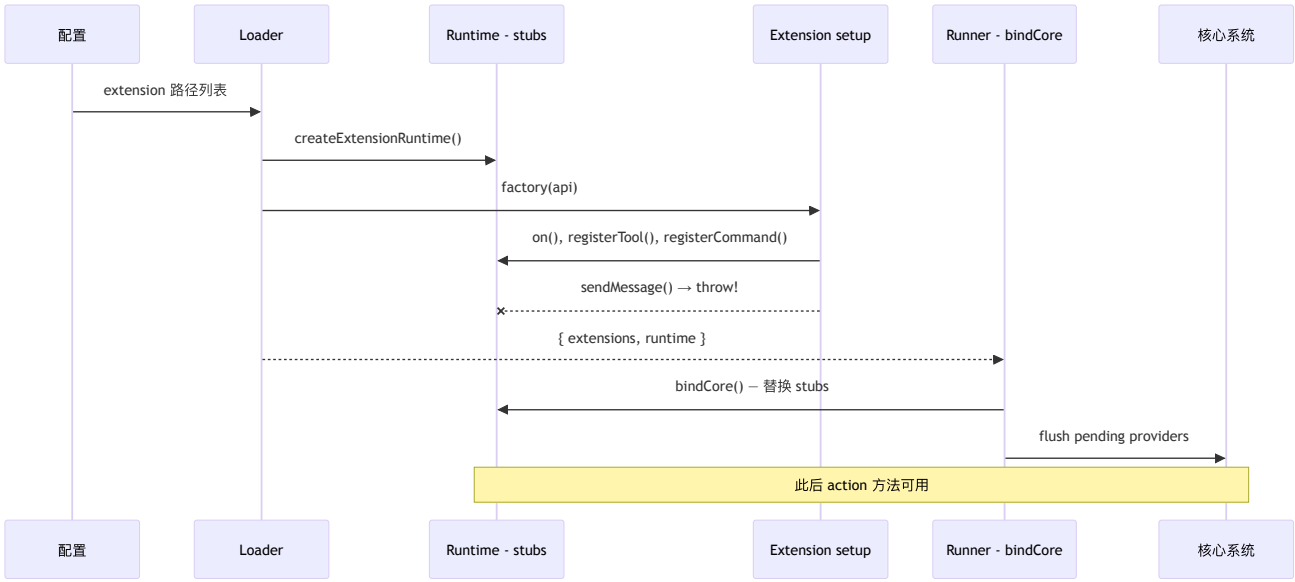
第一阶段（loader 阶段）：runtime 的 action 方法全部是 throwing stubs。Extension 在 `setup()` 中不能调用 `sendMessage()` 或 `setModel()` — 因为此时核心系统还没准备好。但**注册方法**（`on()`、`registerTool()`、`registerCommand()`）可以正常使用，因为它们只是向 `Extension` 对象的 Map 中添加条目。

第二阶段（runner 的 `bindCore()` 阶段）：runner 把真实的 action 实现注入 runtime，替换 throwing stubs。同时 flush 所有 `pendingProviderRegistrations`。从此刻起，事件 handler 中的 `pi.sendMessage()` 等方法才可用。

这种两阶段设计解决了一个经典问题：**extension 的注册代码在系统启动早期执行，但 action 代码需要系统完全就绪后才能执行**。用 throwing stubs 而不是 silent no-op，让开发者在错误使用时立即得到明确的报错。

Wrapper：从注册到运行

Wrapper 层（由 runner 实现）把 extension 注册的工具和命令包装成核心系统能理解的格式。例如，`ToolDefinition` 注册后会被转化为和内建工具相同的调用接口，在 agent loop 的 tool dispatch 中一视同仁。事件 handler 被收集到统一的 handler map 中，由 runner 在对应的生命周期点逐个调用。



Extension 的生命周期

加载时机

Extension 在 pi 启动的早期阶段加载，在 system prompt 装配之前。这是因为 extension 可能注册新工具（需要出现在 prompt 中）、注册 provider（影响模型选择）、或通过 `resources_discover` 事件提供额外的 skill/prompt 路径。

加载顺序：

1. 解析配置中的 extension 路径
2. 创建共享 runtime（throwing stubs 阶段）
3. 串行加载每个 extension，执行 `setup()`
4. Runner 调用 `bindCore()` — 注入真实 action 实现
5. 触发 `session_start` 事件

重载机制

用户可以通过 `/reload` 命令重新加载 extension。重载不是简单的"卸载 + 加载"— 它重新执行整个加载流程（包括 skill、prompt、theme 的重新发现），产生新的 Extension 对象集合，替换旧的。

状态管理

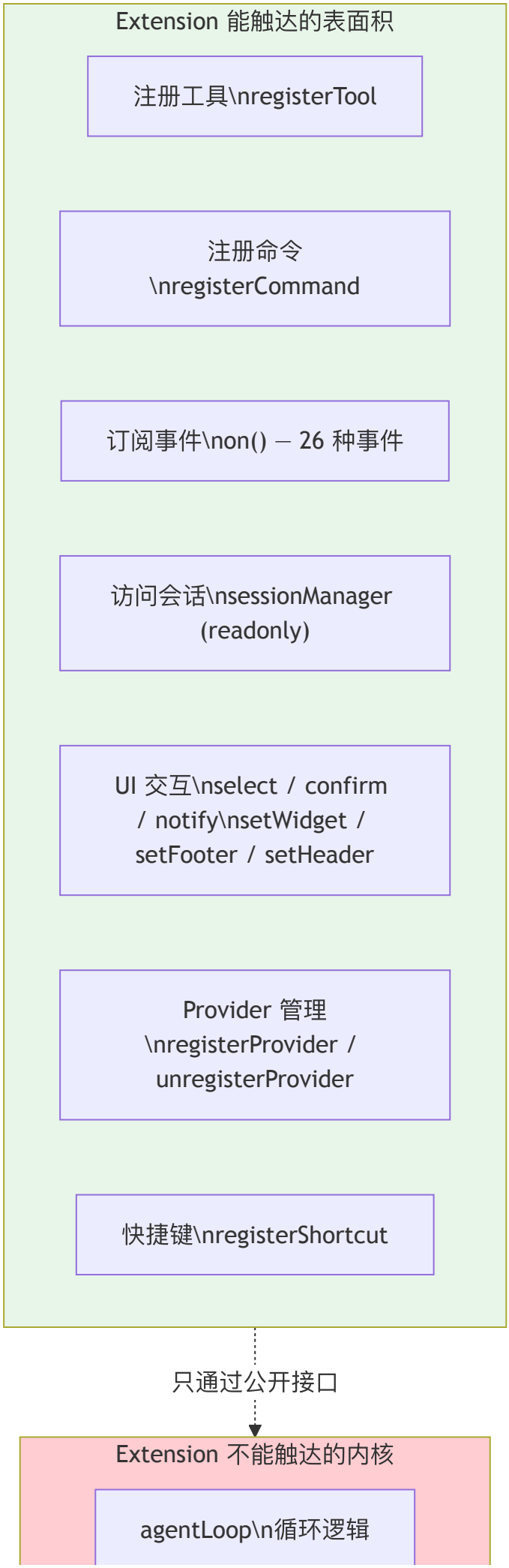
Extension 没有正式的"卸载"钩子 — 没有 `teardown()` 或 `dispose()` 方法。如果 extension 需要在退出时清理资源，可以监听 `session_shutdown` 事件。

Extension 的状态管理是一个有趣的取舍。Extension 用 JavaScript 闭包维护内存中的状态（如上面例子中的 `turnCount`），用 `appendEntry` 持久化到会话文件。但重载会重新执行 `setup()`，闭包状态会丢失。如果 extension 需要跨重载保持状态，必须在 `session_start` 事件中从会话 entries 中恢复。

Extension 不能做什么

这条边界同样重要：

- **不能修改循环逻辑**。循环引擎的 `runLoop()` 不暴露给 extension
- **不能直接修改 Agent 状态**。Extension 通过 `ctx.on(event, handler)` 观察状态，通过 `sendMessage()` / `sendUserMessage()` 注入消息，但不能直接写 `state.messages`
- **不能拦截消息变换**。`transformContext` 和 `convertToLlm` 管道不对 extension 开放
- **Session 是 readonly**。Extension 通过 `ReadonlySessionManager` 读取会话数据，不能修改已有 entries（只能追加新 entries）


Extension 能触达的表面积
注册工具\nregisterTool
注册命令\nregisterCommand
订阅事件\non() — 26 种事件
访问会话\nsessionManager (readonly)
UI 交互\nselect / confirm / notify\nsetWidget / setFooter / setHeader
Provider 管理\nregisterProvider / unregisterProvider
快捷键\nregisterShortcut
只通过公开接口
Extension 不能触达的内核
agentLoop\n循环逻辑

取舍分析

得到了什么

开放但有边界。Extension 可以加能力（新工具、新命令），可以干预流程（拦截 tool call、transform 用户输入），可以改变 UI（替换编辑器、自定义 footer），甚至可以接入新的模型 provider。但不能改核心规则 — 循环逻辑、状态归约、消息变换管道。这让系统的核心行为是可预测的，同时产品功能是可扩展的。

类型安全的 API 合约。整个 extension API 用 TypeScript 严格类型定义，26 种事件每种都有独立的类型和返回值约束。这不是一个"给你个 any，自己造"的 plugin 系统。

放弃了什么

Extension 的能力上限有限。有些深度定制（比如自定义的工具执行策略、自定义的消息格式）无法通过 extension 实现，需要 fork 核心代码。这是"安全性 vs 灵活性"的经典取舍。

没有沙箱。Extension 运行在和核心系统相同的进程中，一个坏的 extension 可以 crash 整个进程。pi 选择了信任开发者（和审计机制）而不是技术隔离。

两阶段初始化增加了认知负担。开发者需要理解"setup 阶段不能调 action 方法"的规则。throwing stubs 的设计让错误变得显式，但这仍然是一个需要学习的概念。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。Extension API 持续扩展，几处设计级变化：① **完整 provider 扩展**（v0.81.0）：extension 可注册完整 pi-ai provider（认证/模型刷新/过滤/自定义流式），此前只能改模型定义/加工具；② 新增三个 hook `before_provider_headers`（types.ts:681）、`agent_settled`（:718）、`session_info_changed`（:566），以及 `InlineExtension`（:1491）内联声明；③ **cache-friendly 动态工具加载**（v0.80.7）：运行中激活的扩展工具在受支持的 Anthropic/OpenAI Responses 模型上保留 prompt-cache 前缀（配合 Kimi native deferred tool loading）；④ 参数 schema 从 `@sinclair/typebox` 迁到 `typebox` 1.x（v0.69.0，旧名仍作别名）；⑤ 工具选择改为字符串名字白名单 `tools: string[]` + cwd 工厂（v0.68.0，移除预构建工具对象）；⑥ `newSession()` / `fork()` / `switchSession()` 后旧 ctx 失效，须用 `withSession` 回调（v0.69.0）；⑦ 早期的 `after_provider_response`、`message_end`（可替换 finalized 消息）、`project_trust` 等干预型 hook 仍在，事件总数已超早期的 26 种；jiti 切到上游 `2.7.0`，VIRTUAL_MODULES 同时注册新旧 scope 别名。`setWidget` / `setFooter` / `setHeader` / `setEditorComponent` 等 UI 扩展点随需求持续增长（`outputPad` 也在 v0.82.1 暴露给自定义消息渲染器）。两阶段初始化与"开放但有边界"的内核保护始终不变。

第 16 章：Skill 机制 — 用文档替代代码

> **定位**：本章解析 pi 最鲜明的哲学选择 — 为什么"能力包"是 markdown 文件而不是代码插件。 前置依赖：第 14 章（System Prompt 装配）、第 15 章（Extension 系统）。 适用场景：当你想理解 skill 和 MCP 的区别，或者想为 pi 创建 skill。

为什么 skill 不是代码?

这是本章的核心设计问题。

在大多数 agent 系统中，"扩展能力"意味着写代码 — MCP server、plugin、adapter。但 pi 的 skill 是**带 frontmatter 的 markdown 文件**。它的类型定义极其简洁：

```
// skills.ts:74-81
export interface Skill {
  name: string;
  description: string;
  filePath: string;
  baseDir: string;
  sourceInfo: SourceInfo;
  disableModelInvocation: boolean;
}
```

六个字段，没有 `execute()`，没有 `handler()`，没有任何可执行代码。Skill 的全部运行时能力就是**被 LLM 读取**。

对应的 frontmatter 接口同样极简：

```
// skills.ts:67-72
export interface SkillFrontmatter {
  name?: string;
  description?: string;
  "disable-model-invocation"?: boolean;
  [key: string]: unknown;
}
```

三个已知字段加一个 index signature — skill 的 metadata 可以携带任意额外信息，但系统只关心名称、描述和是否允许 LLM 自动调用。

一个完整的 Skill 示例

让我们看一个真实的 skill 文件，理解 frontmatter 和内容体的关系：

```
---
name: tdd
description: >
  Test-driven development workflow.
  Use when implementing any feature or bugfix where tests
  are feasible. Guides the red-green-refactor cycle with
  emphasis on writing minimal tests first.
---

## When to use

When implementing any feature or bugfix where automated
tests are feasible. Especially important for:
- Bug fixes (write the regression test FIRST)
- New API endpoints
- Data transformation functions

## Steps

1. **Red**: Write the smallest test that expresses the requirement
2. **Run**: Execute the test, confirm it fails for the right reason
3. **Green**: Write the minimal implementation to make the test pass
4. **Run**: Execute the test again, confirm it passes
5. **Refactor**: Improve the implementation without changing behavior
6. **Run**: Confirm tests still pass after refactoring

## Important

- Do NOT write implementation before the test exists
- Each test should test ONE behavior
- If a test is hard to write, the interface needs redesigning
```

Frontmatter 中的 `name` **约定上**应与父目录名一致（如 `skills/tdd/SKILL.md` ）。注意从 v0.74.1 起这只是约定而非强制 —— `name` 与目录名不一致**不再产生诊断告警**（详见下文

名称验证）。`description` 是注入 system prompt 的部分 — 它是 LLM 决定"要不要读这个 skill"的唯一依据，所以应该包含足够的触发条件信息。

正文（`## When to use` 以下）不会自动注入 prompt — 只有当 LLM 认为当前任务匹配 description 时，它会用 `read` 工具读取完整文件。

disable-model-invocation 与显式触发

当 `disable-model-invocation: true` 时，skill 不会出现在 system prompt 的 `<available_skills>` 列表中，LLM 无法自动发现和加载它。这类 skill 只能通过用户的 `/skill:name` 命令显式触发。适用场景：包含敏感指令、或只在特定上下文中才有意义的 skill。

显式触发时 skill 的内容**如何进入对话**也经过了设计。`/skill:name` 不是简单地把 skill 全文塞进去，而是用一段 **XML wrapper 把 skill 指令与用户本轮消息包裹在一起**注入（v0.73.1 引入，v0.79.0 修复了指令与用户消息之间的空行间距，#5371）。这样模型能清楚区分"这是被显式激活的 skill 指令"与"这是用户的请求"，而不会把两者混成一团。相应地，HTML 会话导出会**剥离这层 skill wrapper XML**（#4234），避免它出现在给人看的渲染结果里 —— 又一次"机器可靠识别用 XML 边界、但对人隐藏标记"的取舍（参见第 14 章 `<project_context>`）。

Frontmatter 解析

Skill 文件的 frontmatter 通过通用的 `parseFrontmatter` 函数解析：

```
// utils/frontmatter.ts:28-37
export const parseFrontmatter = <T extends Record<string, unknown>>(
  content: string,
): ParsedFrontmatter<T> => {
  const { yamlString, body } = extractFrontmatter(content);
  if (!yamlString) {
    return { frontmatter: {} as T, body };
  }
  const parsed = parse(yamlString);
  return { frontmatter: (parsed ?? {}) as T, body };
};
```

解析规则：以 `---` 开头和 `\n---` 结尾的 YAML 块被提取为 frontmatter，剩余部分为 body。没有 frontmatter 的 markdown 文件返回空对象 — 这意味着 skill 仍然可以加载，但会因缺少 `description` 而产生验证警告。

Skill 发现算法

Skill 的发现过程是系统中最体现"约定优于配置"哲学的部分。

来源与优先级

```
// skills.ts:451-453
addSkills(loadSkillsFromDirInternal(
  join(resolvedAgentDir, "skills"), "user", true));
addSkills(loadSkillsFromDirInternal(
  resolve(cwd, CONFIG_DIR_NAME, "skills"), "project", true));
```

加载顺序决定了优先级：**先加载的赢**。全局 skill（`~/.pi/agent/skills/`）先于项目 skill（`.pi/skills/`）加载。这意味着全局 skill 优先 — 当同名冲突时，全局的赢。

等等，这和你的直觉相反吗？大多数系统是"近处优先"。但看代码中的冲突处理：

```
// skills.ts:431-448
const existing = skillMap.get(skill.name);
if (existing) {
  collisionDiagnostics.push({
    type: "collision",
    message: `name "${skill.name}" collision`,
    path: skill.filePath,
    collision: {
      resourceType: "skill",
      name: skill.name,
      winnerPath: existing.filePath,
      loserPath: skill.filePath,
    },
  });
} else {
  skillMap.set(skill.name, skill);
  realPathSet.add(realPath);
}
```

Map 的 `get/set` 语义是"第一个写入的赢"— `existing` 存在时，新的 skill 被记录为 collision 但不覆盖。先加载的来源（user）先写入 Map，所以用户级全局 skill 优先于项目级 skill。

冲突不是静默丢弃 — 它被记录为 `collision` 类型的 diagnostic，让用户知道发生了什么。

目录发现规则

```
// skills.ts:164-175
// Discovery rules:
// - if a directory contains SKILL.md, treat it as a skill root
//   and do not recurse further
// - otherwise, load direct .md children in the root
// - recurse into subdirectories to find SKILL.md
```

这三条规则形成了一个清晰的约定：

规则 1：如果目录包含 `SKILL.md`，这个目录就是一个 skill 包。`SKILL.md` 是 skill 的入口文件，目录名就是 skill 名。不再向下递归 — skill 包内的其他 `.md` 文件是 skill 的内部文件（可以被引用但不单独加载）。

规则 2：根目录下的 `.md` 文件（非 `SKILL.md`）被当作独立 skill 加载。这是简化模式 — 不需要创建子目录。

规则 3：递归扫描子目录寻找 `SKILL.md`。支持任意深度的目录结构。

```
skills/
├── tdd/
│   └── SKILL.md           # → skill "tdd"（规则 1）
├── code-review/
│   ├── SKILL.md            # → skill "code-review"（规则 1）
│   └── checklist.md        # 内部文件，不独立加载
├── quick-tips.md           # → skill "quick-tips"（规则 2）
└── advanced/
    └── perf-tuning/
        └── SKILL.md        # → skill "perf-tuning"（规则 3）
```

.gitignore 尊重

发现过程会读取 `.gitignore`、`.ignore`、`.fdignore` 文件，跳过被忽略的路径。这意味着你可以在 skill 目录中放置工作文件而不用担心它们被加载为 skill。

名称验证

```
// skills.ts:92-112
function validateName(name: string): string[] {
  const errors: string[] = [];
  if (name.length > MAX_NAME_LENGTH) {  // 64
    errors.push(`name exceeds ${MAX_NAME_LENGTH} characters`);
  }
  if (!/^[a-z0-9-]+$/.test(name)) {
    errors.push(`name contains invalid characters`);
  }
  if (name.startsWith("-") || name.endsWith("-")) {
    errors.push(`name must not start or end with a hyphen`);
  }
  if (name.includes("--")) {
    errors.push(`name must not contain consecutive hyphens`);
  }
  return errors;
}
```

名称规则：小写字母、数字、连字符，最长 64 字符，不能以连字符开头或结尾，不能有连续连字符。这些约束确保 skill 名可以安全地用作文件路径、命令名、XML 标签。

> **订正（v0.74.1）**：早期 `validateName` 还接收 `parentDirName` 参数并检查 `name !== parentDirName`，对"名称与父目录不一致"报告诊断。这条检查已被移除 —— 名称与目录名一致现在是**约定而非强制**，不一致不再告警。`validateName` 的签名也相应简化为只接收 `name`。

注意：验证失败只产生 warning，不阻止加载（除非 description 完全缺失）。这是"宽松输入、严格输出"的策略 — 让开发者的 skill 能用，但提醒他们规范命名。

Symlink 去重

```
// skills.ts:419-428
let realPath: string;
try {
  realPath = realpathSync(skill.filePath);
} catch {
  realPath = skill.filePath;
}
if (realPathSet.has(realPath)) {
  continue;  // 同一个物理文件，静默跳过
}
```

如果全局和项目 skill 目录中有 symlink 指向同一个文件，只加载一次。这和名称冲突（产生 diagnostic）不同 — symlink 去重是完全静默的，因为它不是错误，只是重复引用。

formatSkillsForPrompt — 从数据到 Prompt

发现完成后，skill 需要被注入 system prompt。这个过程由 `formatSkillsForPrompt` 完成：

```
// skills.ts:339-365
export function formatSkillsForPrompt(skills: Skill[]): string {
  const visibleSkills = skills.filter(
    (s) => !s.disableModelInvocation
  );
  if (visibleSkills.length === 0) return "";

  const lines = [
    "\n\nThe following skills provide specialized instructions.",
    "Use the read tool to load a skill's file when the task " +
      "matches its description.",
    "When a skill file references a relative path, resolve it " +
      "against the skill directory.",
    "",
    "<available_skills>",
  ];

  for (const skill of visibleSkills) {
    lines.push("  <skill>");
    lines.push(`    <name>${escapeXml(skill.name)}</name>`);
    lines.push(`    <description>${escapeXml(skill.description)}
</description>`);
    lines.push(`    <location>${escapeXml(skill.filePath)}</location>`);
    lines.push("  </skill>");
  }

  lines.push("</available_skills>");
  return lines.join("\n");
}
```

几个关键设计决策：

XML 格式。Skill 列表使用 XML 标签而不是 markdown 或 JSON，遵循 Agent Skills 标准。XML 在 LLM prompt 中有明确的起止标记，不容易和自然语言混淆。

只注入 metadata，不注入全文。每个 skill 只贡献 name + description + location 三个字段到 prompt。完整内容需要 LLM 用 `read` 工具主动读取。这是 token 经济性和能力可用性的平衡 — 如果有 50 个 skill，每个 3000 字，直接注入会消耗 150K token。

preamble 指令。XML 列表前面有三行指令文本，告诉 LLM：(1) skill 提供任务特化的指令，(2) 要用 read 工具加载匹配的 skill，(3) skill 内的相对路径要基于 skill 目录解析。第三点容易被忽视但很重要 — skill 可能引用同目录下的模板文件或配置文件。

disableModelInvocation 过滤。标记了 `disable-model-invocation: true` 的 skill 在此处被过滤掉，LLM 完全看不到它们。

Skill vs MCP vs Extension — 详细对比

维度	Skill	MCP Server	Extension
本质	指令文本（markdown）	RPC 服务（独立进程）	代码模块（同进程）
运行时能力	无 — 只能影响 LLM 行为	可调任意 API、访问外部系统	完整 — 注册工具、命令、UI
执行环境	被 LLM 读取，无执行	独立进程，stdio/SSE 通信	主进程内，共享内存
部署成本	创建 .md 文件	启动服务进程 + 配置 transport	写 TypeScript + 配置路径
创建成本	5 分钟写 markdown	数小时实现 server	数小时学 API + 实现
审计成本	打开文件就能看	需要审计代码 + 网络通信	需要审计代码
版本控制	git diff 友好	需要包管理	需要包管理
安全风险	零（纯文本）	中（独立进程但可联网）	高（同进程，无沙箱）
效果确定性	低 — 依赖 LLM 指令遵循	高 — 代码执行确定	高 — 代码执行确定

维度	Skill	MCP Server	Extension
可组合性	低 — skill 之间无法互调	中 — 工具之间可组合	高 — 可访问系统 API
离线可用	是	取决于 server	取决于实现
适用场景	工作流指南、编码规范、检查清单	数据库查询、API 集成、文件转换	UI 定制、工具拦截、provider 接入

这张表揭示了一个关键洞察：**三种扩展机制不是竞争关系，而是互补的**。Skill 解决"告诉 LLM 怎么做"的问题，MCP 解决"给 LLM 新的能力接口"的问题，Extension 解决"改变 pi 本身的行为"的问题。

一个典型的组合：Extension 注册一个新工具（比如"执行 SQL"），MCP server 提供数据库连接，Skill 描述"在这个项目中，查询数据库时要遵循的安全规范"。三层各司其职。

取舍分析

得到了什么

1. 零依赖、零风险。Skill 是纯文本文件。不需要安装、不需要运行、不需要信任。人类可以在 5 秒内审计一个 skill 的全部内容。

2. 版本控制友好。Skill 文件可以提交到 git、做 code review、做 diff。它的"代码"就是人类可读的自然语言指令。

3. 创建成本极低。写一个 skill 就是写一篇 markdown。不需要学框架、不需要写 schema、不需要实现接口。这让非工程师也能贡献"能力" — 一个 QA 工程师可以写一个 test review skill，一个设计师可以写一个 accessibility audit skill。

4. 渐进式复杂性。最简单的 skill 是一个带 description 的 markdown 文件。更复杂的 skill 可以引用外部文件、使用条件指令、甚至用 `disable-model-invocation` 控制可见性。复杂性是可选的。

放弃了什么

1. 没有运行时能力。Skill 不能调 API、不能查数据库、不能访问外部系统。它只能影响 LLM 的行为，不能扩展 agent 的能力。需要运行时能力时，必须用 extension（第 15 章）或 MCP。

2. 效果依赖 LLM 的指令遵循能力。Skill 的"执行"完全靠 LLM 理解和遵循指令。不同的 LLM 对同一个 skill 的遵循程度不同。没有机械的保证。一个精心编写的 skill 在 Claude 上效果很好，在另一个模型上可能被部分忽略。

3. 发现机制简单。Skill 通过文件路径被发现，没有依赖管理、版本约束、冲突检测（除了同名先到先得）。大规模 skill 生态需要额外的管理工具。

4. 全局 skill 优先于项目 skill。这和大多数"近处优先"的系统不同。如果用户的全局 skill 和项目 skill 同名，项目级的会被忽略（虽然会报 collision diagnostic）。这个决策优先保护用户的个人偏好，但可能让团队共享的项目 skill 意外被覆盖。

pi 的判断：对于大多数 agent 的"能力扩展"需求，**告诉 LLM 怎么做**比**写代码替 LLM 做**更轻量、更安全、更容易维护。Skill 不是万能的 — 但它覆盖了大量"不需要写代码"的场景，让真正需要代码的场景用 Extension 或 MCP 来解决。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。**v0.79.7 → v0.82.1 区间无 skills 专属变更** —— skill 机制的六字段接口、三条发现规则、全局优先于项目、XML 注入、名称验证均未变动；本次仅确认口径对齐 v0.82.1。 更早的两处变化仍成立：① 名称验证放宽（v0.74.1）—— `name` 与父目录名不一致不再告警，`validateName` 签名简化；② 显式 `/skill:name` 触发用 XML wrapper 包裹 skill 指令与用户消息（v0.73.1，间距修复 v0.79.0/#5371），HTML 导出会剥离该 wrapper（#4234）。 Frontmatter 字段限制（name ≤64、description ≤1024）、npm 包 skill 发现、`disable-model-invocation` 区分自动/显式触发均保持。（注：skills 在 system prompt 中的注入 `formatSkillsForPrompt` 与第 14/15 章的 prompt-cache 稳定性纪律相关，但那些是 prompt 装配/extension 侧的变化，skill 机制本身不变。）

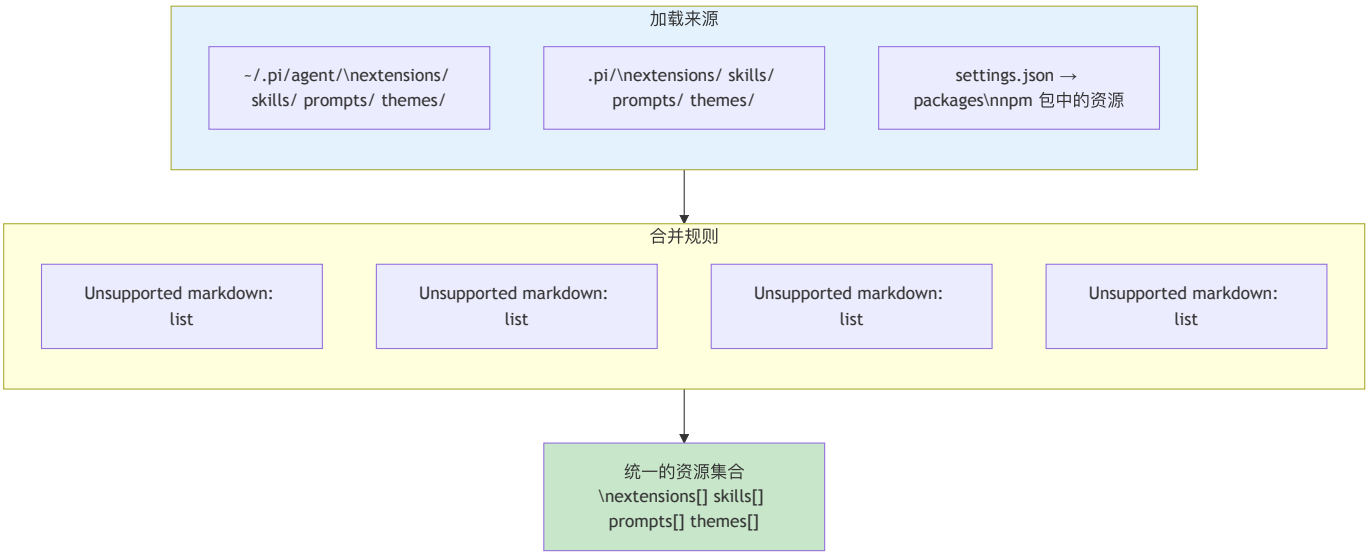
第 17 章：Resource Loader — 一切外部资源的统一入口

> **定位**：本章解析 pi 如何统一加载 extensions、skills、prompts、themes 四种资源。 前置依赖：第 15 章（Extension 系统）、第 16 章（Skill 机制）。 适用场景：当你想理解资源从哪里来、按什么顺序加载。

为什么需要统一入口?

pi 有四种可扩展的外部资源：extensions（代码模块）、skills（指令文档）、prompts（模板文本）、themes（UI 主题）。每种资源有两个作用域：全局（`~/.pi/agent/`）和项目（`.pi/`）。再加上 npm 包来源，组合起来有十几个加载路径。

如果每种资源各自加载，就会有四套发现逻辑、四套冲突处理、四套作用域合并。Resource Loader 把这些统一成一套流程：



ResourceLoader 接口

Resource Loader 的公共接口定义了消费者能做什么：

```
// file: packages/coding-agent/src/core/resource-loader.ts:29-39
export interface ResourceLoader {
  getExtensions(): LoadExtensionsResult;
  getSkills(): { skills: Skill[]; diagnostics: ResourceDiagnostic[] };
  getPrompts(): { prompts: PromptTemplate[]; diagnostics:
ResourceDiagnostic[] };
  getThemes(): { themes: Theme[]; diagnostics: ResourceDiagnostic[] };
  getAgentsFiles(): { agentsFiles: Array<{ path: string; content: string
}> };
  getSystemPrompt(): string | undefined;
  getAppendSystemPrompt(): string[];
  extendResources(paths: ResourceExtensionPaths): void;
  reload(): Promise<void>;
}
```

几个值得注意的设计点：

每种资源的返回值都包含 `diagnostics`。Resource Loader 不会因为一个资源加载失败就中止整个加载过程。它会继续加载其他资源，把错误收集到 `ResourceDiagnostic[]` 中。上层代码（比如 TUI）可以选择把这些诊断信息展示给用户，也可以忽略。

`extendResources` 允许运行时动态扩展。Extension 加载完成后，它可以向 Resource Loader 注册额外的 skill、prompt、theme 路径。这是 "Extension 可以提供 Skill" 的底层机制。

`reload` 是异步的。因为加载过程涉及文件系统读取、npm 包解析，这些操作是异步的。但 `getExtensions` 等取值方法是同步的 — 它们只是返回上一次 `reload` 的结果。

四种资源的不同加载需求

虽然 Resource Loader 提供了统一接口，但四种资源的加载逻辑实际上差异很大：

Extensions — 需要执行代码

Extension 是 TypeScript/JavaScript 模块，加载意味着**执行代码**。加载器需要使用 `jiti`（运行时 TypeScript 编译器）把 `.ts` 文件编译并执行，获得 `ExtensionFactory` 函数，然后调用它得到 `Extension` 对象。

这是四种资源中最复杂的 — 涉及模块解析、错误隔离（一个 extension 崩溃不能影响其他 extension）、以及 runtime 注入（每个 extension 获得一个 `ExtensionRuntime` 对象用于注册能力）。

Skills — 只需读文件

Skill 是 Markdown 文件（通常命名为 `SKILL.md`），加载只需要读取文件内容。但 skill 有一个特殊的路径解析逻辑：如果加载路径指向一个目录，加载器会自动查找该目录下的 `SKILL.md` 文件。

```
// file: packages/coding-agent/src/core/resource-loader.ts:350-370
const mapSkillPath = (resource: { path: string; metadata: PathMetadata
})
    : string => {
  if (resource.metadata.source !== "auto"
      && resource.metadata.origin !== "package") {
    return resource.path;
  }
  try {
    const stats = statSync(resource.path);
    if (!stats.isDirectory()) { return resource.path; }
  } catch { return resource.path; }
  const skillFile = join(resource.path, "SKILL.md");
  if (existsSync(skillFile)) {
    if (!metadataByPath.has(skillFile)) {
      metadataByPath.set(skillFile, resource.metadata);
    }
    return skillFile;
  }
  return resource.path;
};
```

这段代码的逻辑是：对于自动发现的路径和来自 npm 包的路径，如果它指向一个目录，就尝试找 `SKILL.md`。这让 npm 包可以简单地导出一个包含 `SKILL.md` 的目录作为 skill。

Prompts — 需要模板解析

Prompt Template 是文本文件，但不是简单的纯文本 — 它们可能包含变量占位符，需要在使用时填充。加载器需要读取文件并解析模板格式。

Themes — 需要 Schema 验证

Theme 是 JSON 文件，加载后需要验证其结构是否符合 theme schema（颜色、字体大小等字段是否完整）。一个格式错误的 theme 文件不应该让整个 UI 崩溃 — 加载器需要检测并报告格式问题，同时回退到默认 theme。

完整的加载流程

`reload()` 方法是 Resource Loader 的核心。它按照精确的顺序加载所有资源：

```
// file: packages/coding-agent/src/core/resource-loader.ts:318-467
// 简化的 reload 流程:
async reload(): Promise<void> {
  // 1. 重新加载 settings（用户可能修改了 settings.json）
  await this.settingsManager.reload();

  // 2. 通过 PackageManager 解析所有路径
  const resolvedPaths = await this.packageManager.resolve();

  // 3. 过滤出启用的资源路径
  const enabledExtensions = getEnabledPaths(resolvedPaths.extensions);
  const enabledSkills = getEnabledResources(resolvedPaths.skills);
  const enabledPrompts = getEnabledPaths(resolvedPaths.prompts);
  const enabledThemes = getEnabledPaths(resolvedPaths.themes);

  // 4. 合并 CLI 额外路径
  const extensionPaths = this.mergePaths(cliEnabledExtensions,
                                         enabledExtensions);

  // 5. 加载 extensions（执行代码）
  const extensionsResult = await loadExtensions(extensionPaths,
                                                this.cwd,
this.eventBus);

  // 6. 检测 extension 冲突
  const conflicts = this.detectExtensionConflicts(
    extensionsResult.extensions
  );

  // 7. 加载 skills, prompts, themes（读取文件）
  this.updateSkillsFromPaths(skillPaths, metadataByPath);
  this.updatePromptsFromPaths(promptPaths, metadataByPath);
  this.updateThemesFromPaths(themePaths, metadataByPath);

  // 8. 加载 AGENTS.md / CLAUDE.md 上下文文件
  this.agentsFiles = loadProjectContextFiles({ ... });

  // 9. 解析 system prompt
  this.systemPrompt = resolvePromptInput(
    this.systemPromptSource ?? this.discoverSystemPromptFile(), ...
  );
}
```

npm 包来源的加载

除了全局和项目两个本地目录，Resource Loader 还支持从 npm 包加载资源。这是通过 `PackageManager` 实现的。

用户可以在 `settings.json` 中配置包：

```
{
  "packages": [
    "@myorg/pi-extension-custom-tool",
    "@myorg/pi-skills-react"
  ]
}
```

`PackageManager` 负责：

1. 检查包是否已安装（在 `~/.pi/agent/packages/` 中）
2. 如果未安装，使用 npm 安装
3. 解析包的目录结构，找出其中的 extensions、skills、prompts、themes
4. 把这些路径添加到各自的加载列表中

npm 包中的资源遵循一个约定：包的根目录下有 `extensions/`、`skills/`、`prompts/`、`themes/` 子目录。这与全局和项目目录的结构保持一致 — 同样的目录约定，不同的来源。

来自 npm 包的资源在合并顺序中**最后加载**。这意味着如果全局目录和 npm 包中有同名的 skill，全局目录的会被 npm 包的覆盖。这个选择有些反直觉 — 通常你期望本地配置覆盖远程包。但 pi 的设计认为：npm 包是显式安装的（用户主动选择的），全局目录是隐式存在的，显式选择应该有更高的优先级。

冲突诊断

当多个来源提供同名资源时，Resource Loader 不会静默地选择一个。它会生成 `ResourceDiagnostic` 来告知用户存在冲突。

```
// file: packages/coding-agent/src/core/resource-loader.ts:400-405
// 检测 extension 冲突（工具、命令、flag 同名）
const conflicts = this.detectExtensionConflicts(
  extensionsResult.extensions
);
for (const conflict of conflicts) {
  extensionsResult.errors.push(
    { path: conflict.path, error: conflict.message }
  );
}
```

Extension 的冲突检测尤其重要，因为两个 extension 可能注册同名的工具。当检测到冲突时：

1. **所有冲突的 extension 都会被加载**（不会因为冲突就丢弃某个 extension）
2. **冲突被记录为 diagnostic**（用户在 TUI 中可以看到警告）
3. **优先级由加载顺序决定**（后加载的覆盖先加载的）

这是一个典型的 "宽容加载 + 事后报告" 策略。替代方案是 "严格加载 — 有冲突就报错并拒绝加载"。pi 选择宽容策略的原因是：在开发阶段，用户经常需要临时覆盖某个 extension 的行为（比如用项目级的 extension 覆盖全局的），如果每次覆盖都报错阻断，开发体验会很差。

不仅仅是 extension 名称冲突 — Resource Loader 还检测路径不存在的情况：

```
// file: packages/coding-agent/src/core/resource-loader.ts:421-425
for (const p of this.additionalSkillPaths) {
  if (isLocalPath(p) && !existsSync(p)
      && !this.skillDiagnostics.some((d) => d.path === p)) {
    this.skillDiagnostics.push(
      { type: "error", message: "Skill path does not exist", path: p }
    );
  }
}
```

每种资源类型都有类似的检查。用户在 CLI 参数中指定了一个不存在的 skill 路径，不会导致崩溃 — 它会被记录为 diagnostic，其他资源正常加载。

Override 机制

Resource Loader 提供了一套完整的 override 机制，允许上层代码在加载完成后修改结果：

```
// file: packages/coding-agent/src/core/resource-loader.ts:131-148
extensionsOverride?: (base: LoadExtensionsResult) =>
LoadExtensionsResult;
skillsOverride?: (base: { skills: Skill[];
  diagnostics: ResourceDiagnostic[] }) => { ... };
promptsOverride?: (base: { prompts: PromptTemplate[];
  diagnostics: ResourceDiagnostic[] }) => { ... };
themesOverride?: (base: { themes: Theme[];
  diagnostics: ResourceDiagnostic[] }) => { ... };
systemPromptOverride?: (base: string | undefined) => string | undefined;
appendSystemPromptOverride?: (base: string[]) => string[];
```

每种资源类型都有对应的 override 函数。它接收加载完成的基础结果，返回修改后的结果。这让测试、RPC mode、Slack bot 等不同的产品壳可以在不修改 Resource Loader 代码的情况下定制资源加载行为。

比如测试时可以注入 `noExtensions: true` 禁用所有 extension 加载，或者通过 `skillsOverride` 注入测试用的 skill。这比 mock 整个 Resource Loader 简单得多。

AGENTS.md 上下文文件

Resource Loader 还负责加载项目上下文文件（`AGENTS.md` 或 `CLAUDE.md`）。这些文件的加载逻辑与四种资源不同 — 它沿着目录树向上搜索：

```
// file: packages/coding-agent/src/core/resource-loader.ts:76-113
function loadProjectContextFiles(options: { cwd: string; agentDir:
string }) {
  const contextFiles = [];
  // 1. 先加载全局上下文（~/.pi/agent/ 下的 AGENTS.md）
  const globalContext = loadContextFileFromDir(resolvedAgentDir);
  if (globalContext) contextFiles.push(globalContext);

  // 2. 从当前目录向上遍历到根目录
  let currentDir = resolvedCwd;
  while (true) {
    const contextFile = loadContextFileFromDir(currentDir);
    if (contextFile) ancestorContextFiles.unshift(contextFile);
    if (currentDir === root) break;
    currentDir = resolve(currentDir, "..");
  }
  // 3. 按从根到当前目录的顺序返回
  contextFiles.push(...ancestorContextFiles);
  return contextFiles;
}
```

这意味着在 `/home/user/project/src/` 目录下运行 pi 时，它会查找并加载：

- `~/.pi/agent/AGENTS.md` （全局）
- `/home/AGENTS.md` （如果存在）
- `/home/user/AGENTS.md` （如果存在）
- `/home/user/project/AGENTS.md` （如果存在）
- `/home/user/project/src/AGENTS.md` （如果存在）

所有找到的文件**按顺序拼接**（不覆盖），作为项目上下文注入 system prompt。这个设计让组织可以在不同层级的目录中放置不同粒度的上下文 — 根目录放通用规范，子目录放模块特定的上下文。

边界：跳过与 context 文件同名的目录（#7106）

`loadContextFileFromDir` 在每个目录里按候选名（`AGENTS.md` / `CLAUDE.md` 及大写变体）找 context 文件。这里有一个真实踩到的坑：候选名指向的路径**未必是文件** —— 有人会建一个叫 `AGENTS.md` 的**目录**（比如把多份 agent 说明拆成 `AGENTS.md/foo.md`）。早期代码直接 `readFileSync` 这个路径，撞上目录就抛 `EISDIR`，整个上下文加载失败。v0.82.1 的修复（#7106）加了一道 `isFile` 判断：

```
// packages/coding-agent/src/core/resource-loader.ts:69-78（节选）
for (const filename of candidates) {
  const filePath = join(dir, filename);
  if (existsSync(filePath)) {
    if (!statSync(filePath).isFile()) {
      continue;            // 同名的是目录 → 跳过，继续试下一个候选
    }
    return { path: filePath, content: readFileSync(filePath, "utf-8") };
  }
}
```

`if (!statSync(filePath).isFile()) continue;`（`resource-loader.ts:73`）让发现逻辑在遇到"同名目录"时静默跳过、继续尝试下一个候选名，而不是崩掉。这是"宽容加载"原则在一个刁钻边界上的体现 —— 一个非常规的目录布局不该让整个会话起不来。

与 ch13 的项目本地资源覆盖是同一作用域

本章的资源分层（全局 → 项目 → npm 包）和第 13 章的 `settings.json` 分层（全局 → 项目）其实是**同一个"全局/项目本地"作用域概念**的两面：一个管 extensions/skills/prompts/themes，一个管结构化 settings。第 13 章讲的 `pi config -l`、交互编辑器里 Tab 切换全局/项目作用域，改的正是本章"项目后加载、覆盖全局"里的"项目"那一层。两章合起来，才是完整的"项目本地资源覆盖"图景 —— 而它们都统一受下节的 Project Trust 闸门管辖。

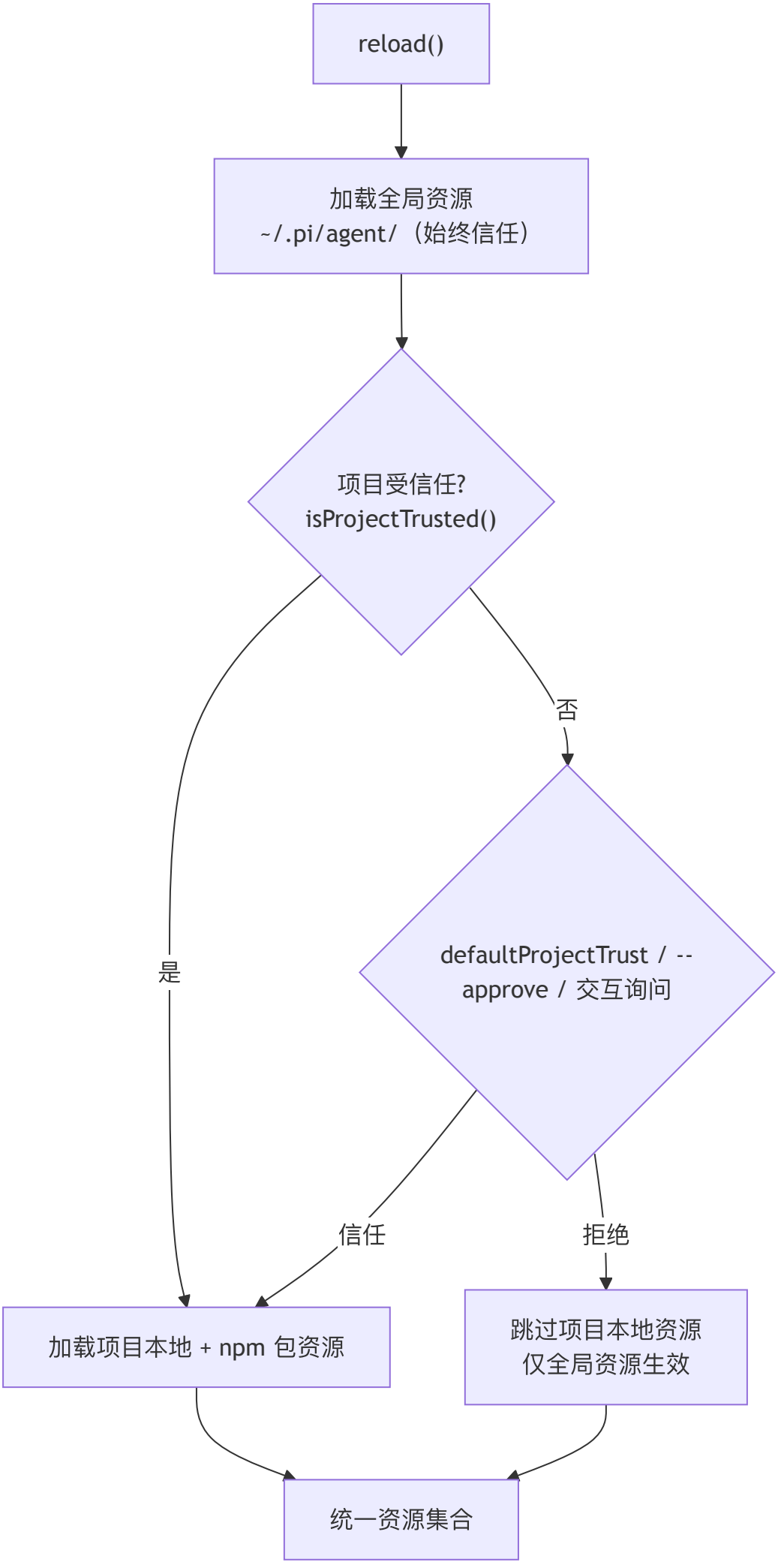
显式 cwd：去掉 process.cwd() 兜底（v0.68.0）

注意上面 `loadProjectContextFiles` 的签名 —— `cwd` 是**必填**的，不再是早期的可选参数。这是一个看似琐碎、实则贯穿全局的设计纪律。早期版本里 `DefaultResourceLoader`、`loadProjectContextFiles()`、`loadSkills()`（skills.ts:387）等都会在 `cwd` 缺省时回退到 `process.cwd()`。v0.68.0 起，这个隐式兜底被移除：调用方**必须显式传入**工作目录。

去掉 `process.cwd()` 兜底的动机是：pi 不再假设"只有一个全局当前目录"。同一个进程里可能同时存在多个会话、多个工作目录（RPC 模式、SDK 嵌入、子 agent），任何依赖进程级全局状态的代码都会在这些场景下出错。把 `cwd` 提升为显式必填参数，等于在类型层面强制每一处资源加载都说清楚"相对谁解析"。这条"去 process-global"的纪律同样体现在 system prompt 装配（第 14 章 `buildSystemPrompt` 的 cwd 必填）与各工具的 cwd 工厂（第 19 章）上 —— 它是 pi 走向"可嵌入库"（第 26b 章 SDK）的前提。

项目本地资源受信任门控（v0.79.0）

本章开头的加载流程图把规则简化为"全局先加载、项目后加载、npm 包最后加载"。从 **v0.79.0** 起，这个流程在"全局"与"项目"之间多了一道**信任闸门**：项目本地的 extensions / skills / prompts / packages / settings 都是可执行的攻击面（extension 跑代码、packages 拉取并加载 npm/git 代码、AGENTS.md 注入 prompt），因此在加载它们之前， `reload()` 会先经过 Project Trust 决策。


reload()
加载全局资源
~/.pi/agent/（始终信任）
项目受信任?
isProjectTrusted()
是
否
defaultProjectTrust / --approve / 交互询问
信任
拒绝
加载项目本地 + npm 包资源
跳过项目本地资源
仅全局资源生效
统一资源集合

具体地说：全局作用域的资源无条件加载；项目作用域的资源是否加载，取决于 `isProjectTrusted()`（extension 也能经 `ctx.isProjectTrusted()` 查询，见第 15 章）。非交互场景下由全局设置 `defaultProjectTrust`（`ask` / `always` / `never`）或 CLI `--approve` 决定，交互场景下弹出 `/trust` 询问。换句话说，本章原先"项目资源总是会加载"的假设在 v0.79.0 后不再成立 —— 它现在是一个**经信任决策后**才发生的步骤。信任闸门的完整设计见第 13 章。

取舍分析

得到了什么

统一的心智模型。所有资源遵循同样的加载顺序和覆盖规则。用户学会一套规则就能理解所有资源的行为。

渐进式降级。任何单个资源加载失败都不会阻塞系统启动。通过 diagnostic 机制，用户可以在启动后看到哪些资源加载失败了，但系统仍然可用。

可测试性。override 机制让测试可以精确控制每种资源的加载结果，而不需要 mock 文件系统。

放弃了什么

资源类型之间的差异被抹平。Extension 需要执行 setup、skill 只需读文件、theme 需要验证 schema — 不同类型有不同的加载需求，统一入口需要为最复杂的类型设计接口。这导致接口上有些方法（如 `reload` 的异步性）对简单资源来说是过度设计。

加载顺序不够透明。全局 → 项目 → npm 包 → CLI 额外路径 — 当多个来源都提供了资源时，用户需要理解完整的合并顺序才能预测最终结果。虽然 diagnostic 可以报告冲突，但合并的过程本身不够可观测。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。几处设计级变化： ① **上下文文件发现跳过同名目录**（v0.82.1，#7106）： `loadContextFileFromDir` 加 `if (!statSync(filePath).isFile()) continue;` （`resource-loader.ts:73`），避免叫 `AGENTS.md` 的目录导致 `EISDIR` 。 ② **显式 cwd 必填**（v0.68.0）： `DefaultResourceLoader` / `loadProjectContextFiles` / `loadSkills` 移除 `process.cwd()` 兜底，cwd 必须显式传入（"去 process-global"纪律）。 ③ **项目本地资源受 Project Trust 门控**（v0.79.0）：reload 在全局与项目加载之间插入信任闸门，项目资源不再无条件加载；项目本地资源覆盖与第 13 章 `pi config -l` 作用域切换同属一个概念。 ④ 兄弟 npm 扩展保留相对路径显示、恢复会话时资源通知保持在消息前（v0.82.0/v0.80.3）。 资源来源仍是全局 + 项目 + npm 包三级； `extendResources` 运行时扩展与 override 机制保持不变。

第 18 章：Model Registry — 模型不只是一个 ID

> **定位**：本章解析模型选择背后的配置系统和动态注册机制。 前置依赖：第 4 章（Provider Registry）、第 13 章（配置覆盖）。 适用场景：当你想理解 pi 如何管理模型列表，或者想添加自定义模型。

选模型 = 选 provider + 选 api + 选参数

用户在 pi 中选择一个"模型"，背后实际上是选择了一组参数：

```
// file: packages/ai/src/types.ts:749
interface Model<TApi extends Api> {
  id: string;            // "claude-opus-4-6"
  name: string;          // "Claude Opus 4.6"
  api: TApi;             // "anthropic-messages"
  provider: ProviderId;  // "anthropic"（v0.80.0 起由旧名 Provider 改来）
  baseUrl?: string;      // API endpoint
  cost: { input; output; cacheRead; cacheWrite };
  contextWindow: number; // 200000
  maxTokens: number;     // 32768
  input: ("text" | "image" | "audio")[];
  reasoning: boolean;    // true
  thinkingLevelMap?: ThinkingLevelMap; // 见下文
}
```

其中 `thinkingLevelMap`（`types.ts:760`）是后来新增的顶层字段：它把 pi 统一的思考档位（`off` / `minimal` / `low` / `medium` / `high` / `xhigh` / `max`）映射到该 provider 专用的取值，`null` 表示某档位不支持。配套的 `getSupportedThinkingLevels(model)` 与 `clampThinkingLevel(model, level)`（`models.ts:663`、`:674`）据此计算模型实际支持的档位并把越界请求收敛到最近的合法档。它取代了早期写在 `compat.reasoningEffortMap` 里的映射（旧的 `supportsXhigh()` 也随之废弃）。

这些参数来自三个来源：

1. **内建模型目录**：pi 内置了一个生成的 TypeScript 文件，列出所有已知模型的参数
2. **`models.json` 自定义**：用户可以在全局配置中添加、覆盖模型定义
3. **Extension 动态注册**：Extension 可以注册新的 API provider（第 4 章的 `registerApiProvider`），随之带来新的模型

内建模型目录的生成

pi 不手动维护模型列表 — 它通过自动化脚本从各 provider 的 API 抓取最新数据并生成代码。

```
// file: packages/ai/scripts/generate-models.ts:1-10
#!/usr/bin/env tsx
import { writeFileSync } from "fs";
import { join, dirname } from "path";
import { fileURLToPath } from "url";
import { Api, KnownProvider, Model } from "../src/types.js";

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);
const packageRoot = join(__dirname, "..");
```

这个脚本从 OpenRouter / AI Gateway 等来源抓取模型列表与元数据（context window、定价、是否支持 tool calling），合并 Anthropic、Google、Bedrock 等主流 provider，生成内建目录。

> **拆包重构（v0.80.0）**：早期它生成的是**单体** `models.generated.ts`（一个巨大的 `export const MODELS = {...}` 对象）。现在目录**按 provider 拆分**成 per-provider 的 `providers/*.models.ts`，每个只导入自己的一份 JSON 数据并展开：

```
// file: packages/ai/src/providers/anthropic.models.ts:1-8
// This file is auto-generated by scripts/generate-models.ts
import values from "./data/anthropic.json" with { type: "json" };
import { flattenModelCatalog, type ModelCatalog } from "../model-
catalog.ts";

export const ANTHROPIC_MODELS: ModelCatalog<typeof values, "anthropic">
=
  flattenModelCatalog("anthropic", values);
```

`models.generated.ts` 退化为一个**纯聚合器**：把 37 份 `*_MODELS` 拼成 `MODELS` 对象（`models.generated.ts:42`），类型上保留 `readonly "anthropic": typeof ANTHROPIC_MODELS` 这样的精确映射。拆分的收益是 tree-shaking：一个只用 Anthropic 的下游可以只带走 `anthropic.models.ts` 那一份数据，而不必把全部 provider 的模型 JSON 打进 bundle（呼应第 4 章的 core-only 拆包）。

读内建目录的现行 API 在 `providers/all.ts`，不再直接读 `MODELS`：

```
// file: packages/ai/src/providers/all.ts:59-84（节选）

/** 类型化地读单个内建模型。 */
export function getBuiltinModel<TProvider, TModelId>(
  provider: TProvider, modelId: TModelId): Model</* 推断出的 api */> {
... }

/** 读某个 provider 的全部内建模型。 */
export function getBuiltinModels<TProvider>(provider: TProvider):
Model<...>[] { ... }

/** 内建目录的生成时间戳（取代旧的 getBuiltinModelDataUrl）。 */
export function getBuiltinModelDataGeneratedAt(): number | undefined {
  const generatedAt = Date.parse(modelDataManifest.generatedAt);
  return Number.isNaN(generatedAt) ? undefined : generatedAt;
}
```

`getBuiltinModel` / `getBuiltinModels`（`all.ts:59` / `:77`）是类型化的静态目录读法：泛型参数从 `MODELS` 的形状里把 `api` 推断出来，让返回的 `Model<TApi>` 精确到协议。`getBuiltinModelDataGeneratedAt`（`all.ts:72`）是 **v0.82.0 的 Breaking 变更** — 它取代了旧的 `getBuiltinModelDataUrl(provider)`：上层现在关心的不是"目录数据从哪个 URL 来"，而是"这份内建数据是什么时候生成的"（用于判断新鲜度、决定要不要触发动态刷新）。时间戳来自随目录一起生成的 `providers/data/.manifest.json`。

每次构建 pi-ai 都会重新生成模型目录（`package.json` 的 `build` 脚本先跑 `generate-models`）。这意味着每次发布新版本时，模型目录自动包含最新的模型。

这个设计有一个重要的取舍：**模型数据在构建时确定，而不是运行时查询**。

得到了什么：离线可用 — pi 不需要网络连接就能显示模型列表。启动速度 — 不需要等待 API 响应。确定性 — 同一个版本的 pi 永远看到同样的内建模型列表。

放弃了什么：时效性 — 新模型发布后，用户需要等 pi 的下一个版本才能在内建目录中看到它。（但用户可以通过 `models.json` 立即使用新模型，或经动态刷新拉取 — 见下节与本章后文。）

动态目录的持久化：ModelsStore 与 etag 条件刷新

内建目录是构建时冻结的静态数据，但有些 provider 的模型列表是**动态**的 — 它随订阅、随远程目录变化。第 4 章讲过 `Provider.refreshModels` 和 `Models.refresh`：动态 provider 用凭证去远程拉取最新列表。这份动态列表需要一个持久化的落点，就是 `ModelsStore`：

```
// file: packages/ai/src/models-store.ts:3-21

export interface ModelsStoreEntry {
  models: readonly Model<Api>[];
  lastModified?: number;    // 远程 Last-Modified
  checkedAt?: number;       // 上次完成远程检查的时间
  etag?: string;            // 远程 ETag，原样存、原样回传为 If-None-Match
}

export interface ModelsStore {
  read(providerId: string): Promise<ModelsStoreEntry | undefined>;
  write(providerId: string, entry: ModelsStoreEntry): Promise<void>;
  delete(providerId: string): Promise<void>;
}
```

`ModelsStore` 按 provider id 存一份 `ModelsStoreEntry`。`Models` 默认用 `InMemoryModelsStore`（`models-store.ts:28`，进程内、测试用），产品层可注入一个落盘实现。刷新时 `Models` 把它包装成 provider-scoped 的 `ProviderModelsStore`（`read/write/delete` 都不带 providerId），provider 只能碰自己的那一格，碰不到别家的目录。

`etag` 字段（v0.82.1）是**条件刷新**的关键。provider 拉取时把上次存下的 `etag` 作为 `If-None-Match` 头发出去，远程若返回 `304 Not Modified`，就说明目录没变，直接复用缓存、不重新解析。这把"每次刷新都全量下载 + 解析模型列表"降级为"大多数刷新只花一个条件请求的往返"。配合 `Models.refresh({ force })`（`models.ts:276`，可绕过 provider 的新鲜度检查强制拉取）和 per-provider 的错误/取消隔离（一个 provider 刷新失败不影响其余），动态目录的刷新既省流量又稳健。

`createProvider` 内部已经把这套缓存-拉取-落盘的时序封装好了（`models.ts:596-617`）：先从 store 恢复上次的动态列表，若允许网络再 `fetchModels`，成功后写回 store。provider 作者只需提供一个 `fetchModels` 函数，不必自己管缓存与并发。

models.json 覆盖机制

用户可以在 `~/.pi/agent/models.json` 中自定义模型。这个文件遵循严格的 JSON Schema。**注意 schema 及其解析已经从 `model-registry.ts` 迁到独立的 `ModelConfig`（`model-config.ts`）** —— 这是本区间 coding-agent 侧模型管理重构的一部分（详见下文 ModelRuntime）：

```
// file: packages/coding-agent/src/core/model-config.ts:154-167
const ModelDefinitionSchema = Type.Object({
  id: Type.String({ minLength: 1 }),
  name: Type.Optional(Type.String({ minLength: 1 })),
  api: Type.Optional(Type.String({ minLength: 1 })),
  baseUrl: Type.Optional(Type.String({ minLength: 1 })),
  reasoning: Type.Optional(Type.Boolean()),
  thinkingLevelMap: Type.Optional(ThinkingLevelMapSchema),
  input: Type.Optional(Type.Array(
    Type.Union([Type.Literal("text"), Type.Literal("image")])
  )),
  cost: Type.Optional(ModelCostSchema),
  contextWindow: Type.Optional(Type.Number()),
  maxTokens: Type.Optional(Type.Number()),
  headers: Type.Optional(Type.Record(Type.String(), Type.String())),
  compat: Type.Optional(ProviderCompatSchema),
});
```

注意 schema 的设计：除了 `id`，所有字段都是 **optional**。这意味着用户定义一个本地模型时，只需提供最少的信息：

```
{
  "providers": {
    "my-local-llm": {
      "baseUrl": "http://localhost:11434/v1",
      "api": "openai-completions",
      "models": [
        { "id": "llama-3.3-70b" }
      ]
    }
  }
}
```

未指定的字段会使用合理的默认值。这降低了添加本地模型（Ollama、LM Studio 等）的门槛。

覆盖内建模型

`models.json` 不仅能添加新模型，还能覆盖内建模型的参数。通过 `modelOverrides` 字段（`ModelOverrideSchema`，`model-config.ts:169`）：schema 与 `ModelDefinitionSchema` 几乎同构，但 `cost` 字段内部每一项也都是 optional —— 你可以只覆盖 `input` 定价而保留其他定价不变。

覆盖的**应用**逻辑（把 override deep-merge 进内建模型）已经从旧的 `ModelRegistry.applyModelOverride` 迁到 `provider-composer.ts` 的 `applyModelOverride`（`provider-composer.ts:100-120`）：它逐字段覆盖简单值，对 `cost` 做 partial merge（`override.cost.input ?? model.cost.input`），对 `compat` 用 `mergeCompat`。`composeModelProvider` 在装配 provider 时对每个模型应用一次 override（`provider-composer.ts:423-436`）—— 也就是说，覆盖不再由 coding-agent 手写的 `loadModels` 循环完成，而是作为"最上层用户配置"注入 pi-ai 的 `Models` 运行时（见下文 ModelRuntime）。

ModelRuntime：规范的异步 model/auth 门面

到这里，一个重要的架构变化必须交代。coding-agent 侧原来以 `ModelRegistry` 为中心 —— 它同步持有一份 `Model[]`，构造时 `new ModelRegistry(authStorage, modelsJsonPath)` 就地 `loadModels()`。**从 v0.80.8 起（Breaking），这个中心让位给了 `ModelRuntime`（`model-runtime.ts`）**：它是一个 `implements Models` 的**异步**门

面，统一管模型配置、认证、provider 自有 `/login` 、以及动态 provider 目录。`ModelRegistry` 降级为一层面向 extension 的**同步 compat 投影**。

```
// file: packages/coding-agent/src/core/model-runtime.ts:94,133
export class ModelRuntime implements Models {
  static async create(
    options: CreateModelRuntimeOptions = {},
  ): Promise<ModelRuntime> {
    const credentials = new RuntimeCredentials(
      options.credentials ??
DefaultAuthStorage.create(options.authPath));
    const config = await ModelConfig.load(modelsPath);       // 解析
models.json
    const modelsStore = modelsPath
      ? new FileModelsStore(/* …/models-store.json */)       // 动态目录落
盘
      : new InMemoryCodingAgentModelsStore();
    const providers = builtinProviders().map((p) =>           // 内建
provider
      withRemoteCatalog(p, options.catalogBaseUrl, generatedAt)); // 套远
程目录
    const runtime = new ModelRuntime(credentials, config, modelsPath,
      modelsStore, providers, /* networkEnabled */);
    runtime.rebuildProviders();
    await runtime.refresh({ allowNetwork: /* … */ });
    return runtime;
  }
}
```

注意几个关键点：构造是 `static async create()` （因为要读盘、可能拉网络），不再是旧的同步 `new` ；它不再接收 `AuthStorage` 而是接收一个 `CredentialStore` （认证子系统的口径见第 4-7 章）；models.json 的解析交给独立的 `ModelConfig` （`model-config.ts` ），动态目录交给 `ModelsStore` （下节）。

`rebuildProviders()` 是覆盖真正生效的地方：它对每个 provider 调用 `composeModelProvider(providerId, base, config, extension)` ，把 models.json 的 provider 覆盖、 `modelOverrides` 、自定义模型作为"最上层用户配置"注入 pi-ai 的 `Models` （ `this.models.setProvider(...)` ， `model-runtime.ts:215` ）。也就是说，前一节讲的 override 应用、以及"自定义模型按 `provider + id` 合并、冲突时自定义优先"这些规则，如今都由 pi-ai 的 `Models` 运行时统一执行，coding-agent 不再手写 `loadModels` 循环。

ModelRegistry 作为 compat 投影

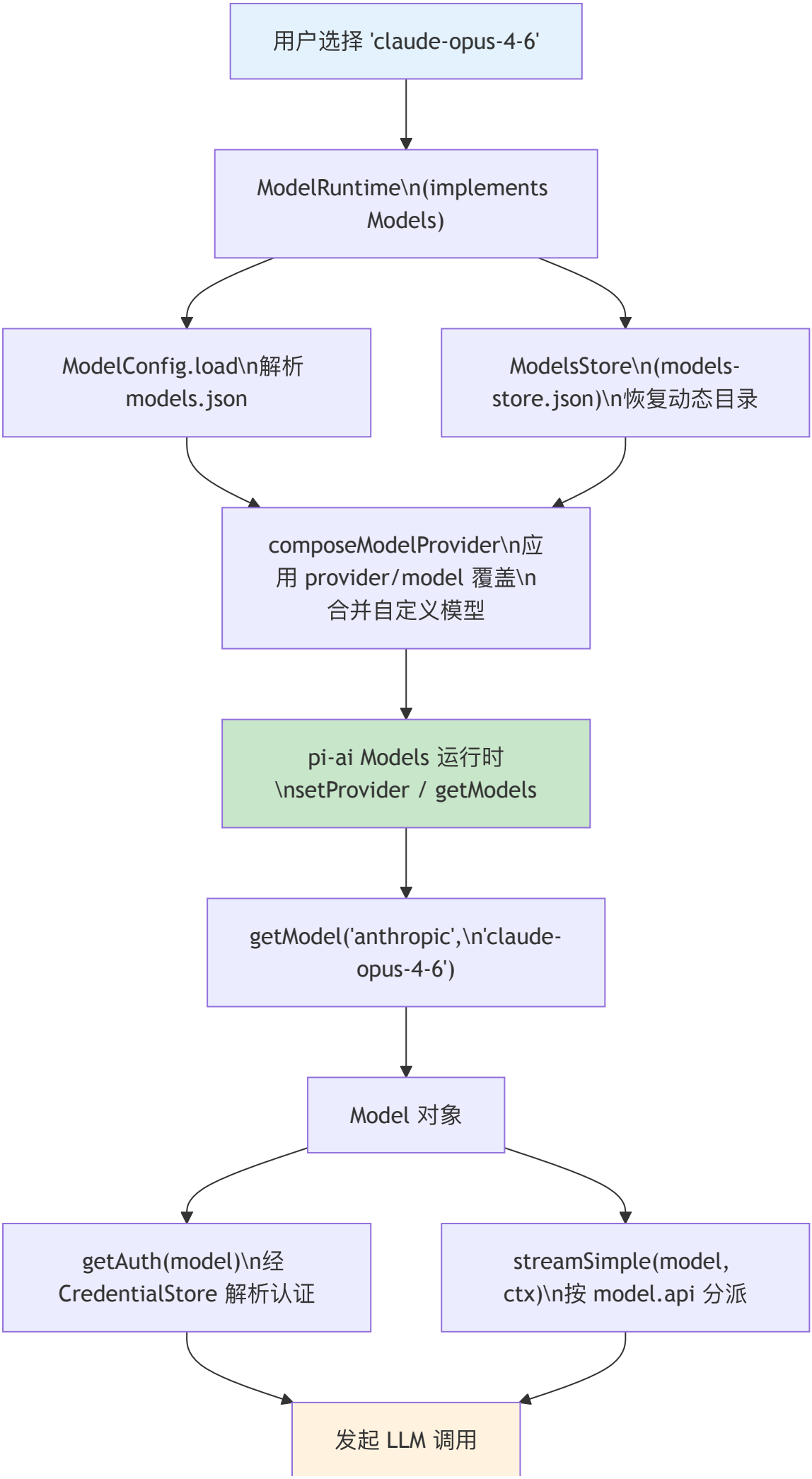
那么 `ModelRegistry` 还在吗？在 —— 但它现在只是 `ModelRuntime` 的一层薄壳，其唯一存在理由是给 extension 一个**同步**读法（extension API 面历史上是同步的）：

```
// file: packages/coding-agent/src/core/model-registry.ts:16-38
/**
 * Synchronous compatibility facade exposed to extensions.
 * Coding-agent internals use ModelRuntime directly.
 */
export class ModelRegistry {
  constructor(runtime: ModelRuntime) { this.runtime = runtime; }

  /** Reload models.json asynchronously. Await before synchronous reads.
*/
  async refresh(): Promise<void> { await this.runtime.refresh(); }

  getAll(): Model<Api>[] { return [...this.runtime.getModels()]; }
  getAvailable(): Model<Api>[] {
    return [...this.runtime.getAvailableSnapshot()];
  }
  find(provider: string, modelId: string): Model<Api> | undefined {
    return this.runtime.getModel(provider, modelId);
  }
}
```

两处口径变化值得钉住：① 构造从 `new ModelRegistry(authStorage, modelsJsonPath)` 变成 `new ModelRegistry(runtime)` —— 它不再自己读盘，而是投影一个已经装配好的 `ModelRuntime`；② `refresh()` 的返回类型从 `void` 改成 **`Promise<void>`**（Breaking），因为底层的目录刷新本质上是异步的（要读盘、可能走网络）。extension 若在同步读之前要保证拿到最新目录，必须先 `await registry.refresh()`。

模型解析的完整流程（经 ModelRuntime）

当用户选择 "claude-opus-4-6" 时，现行的解析路径是：


用户选择 'claude-opus-4-6'
ModelRuntime\n(implements Models)
ModelConfig.load\n解析 models.json
ModelsStore\n(models-store.json)\n恢复动态目录
composeModelProvider\n应用 provider/model 覆盖\n合并自定义模型
pi-ai Models 运行时\nsetProvider / getModels
getModel('anthropic',\n'claude-opus-4-6')
Model 对象
getAuth(model)\n经 CredentialStore 解析认证
streamSimple(model, ctx)\n按 model.api 分派
发起 LLM 调用

`ModelRuntime` 对外暴露的就是 `Models` 接口那套异步方法：`getModels()` / `getModel(provider, id)` / `getAvailableSnapshot()`（`model-runtime.ts:301` / `:305` / `:330`）读目录，`getAuth(model)`（`:376`）解析认证，`streamSimple(model, ctx)`（`:492`）按 `model.api` 分派到具体 provider。SDK 的 `createAgentSession` 直接吃一个 `modelRuntime`（第 26b 章），不再传 `authStorage` + `modelRegistry` 两件套。

动态目录落盘：models-store.json 与条件刷新

前面 pi-ai 侧的 `ModelsStore` 是接口；coding-agent 侧要给它一个**落盘**实现，这就是 `FileModelsStore`（`models-store.ts:28`）。它把每个 provider 的动态目录写进 `~/.pi/agent/models-store.json`，并复用会话/认证同款的文件锁（`withLock`）保证并发安全；测试则用 `InMemoryCodingAgentModelsStore`。

真正省流量的是**条件刷新**。动态 provider（pi.dev 远程目录）由 `withRemoteCatalog` 包裹（`model-runtime.ts:149`），刷新时把上次存下的 `etag` 作为 `If-None-Match` 发出去：

```
// file: packages/coding-agent/src/core/remote-catalog-provider.ts:73-90（节选）
const validator = stored?.models.length ? stored.etag : undefined;
const response = await fetch(url, {
  headers: { accept: "application/json",
    ...(validator ? { "if-none-match": validator } : {}) },
  signal: context.signal,
});
// 304 Not Modified：目录没变，只推进 checkedAt，复用缓存的 overlay
if (response.status === 304 && stored) {
  await context.store.write({ ...stored, checkedAt });
  return;
}
```

远程若回 `304 Not Modified`，就说明目录没变，直接复用 store 里缓存的模型、只推进 `checkedAt`，**不重新下载、不重新解析**。这把"每次刷新都全量拉目录"降级为"多数刷新只花一个条件请求的往返"。刷新还带 per-provider 的错误/取消隔离：一个 provider 拉取失败（非 2xx）时保留旧 etag 让下次重试，不影响其余 provider。

这套刷新已经**移出启动路径**（不再阻塞冷启动）。要主动强刷，用命令 `pi update --models`（`package-manager-cli.ts:728`，调用 `refreshModelCatalogs`），它绕过新鲜

度窗口强制重拉所有目录；交互中切 `/model` 也会在后台触发刷新。

Extension 注册新 Provider 的能力

Model 系统最有趣的扩展点仍在：extension 可以注册全新的 provider（第 15 章的 `registerProvider`），任何使用该 api 的 model 定义随之可用。比如一个私有部署的 OpenAI 兼容端点：extension 注册一个 `"openai-responses"` provider，用户在 `models.json` 里加一条指向自定义 `baseUrl` 的模型定义，pi 就能用它。

承接这种动态注册的是 `ModelRuntime.refresh()`（`model-runtime.ts:516`）—— 它重新跑 `configureRadiusProviders()` + `rebuildProviders()`，用最新的 `ModelConfig` 与 extension 注册重建整份 provider 列表。因为 `rebuildProviders` 每次都从内建 provider + 配置**重新装配**，`refresh` 后的状态天然一致，不会残留旧注册。extension 侧通过 compat 投影 `ModelRegistry.refresh()`（返回 `Promise<void>`）触达同一条路径。

`max` / `xhigh` 两个更高的 thinking 档位也在本区间贯穿了 CLI/SDK/RPC 与主题 —— 模型实际支持哪些档由 `thinkingLevelMap` 决定（见本章开头），越界请求被 `clampThinkingLevel` 收敛到最近合法档。

OpenAI 兼容性层 (compat)

`Model` 类型中有一个 `compat` 字段，它是 Model Registry 中最复杂的部分：

```
// file: packages/coding-agent/src/core/model-config.ts:72-108（节选）
const OpenAICompletionsCompatSchema = Type.Object({
  supportsStore: Type.Optional(Type.Boolean()),
  supportsDeveloperRole: Type.Optional(Type.Boolean()),
  supportsReasoningEffort: Type.Optional(Type.Boolean()),
  maxTokensField: Type.Optional(Type.Union([
    Type.Literal("max_completion_tokens"),
    Type.Literal("max_tokens")
  ])),
  thinkingFormat: Type.Optional(Type.Union([
    Type.Literal("openai"), Type.Literal("openrouter"),
    Type.Literal("together"), Type.Literal("deepseek"),
    Type.Literal("zai"), Type.Literal("qwen"),
    Type.Literal("qwen-chat-template"),
  ])),
  openRouterRouting: Type.Optional(OpenRouterRoutingSchema),
  // ... 更多字段
});
```

> **版本变化（v0.72.0，破坏性）**：早期 `compat` 里有一个 `reasoningEffortMap` 字段承载"pi 档位 → provider effort 值"的映射。它已被移除，映射上移为 `Model` 的顶层 `thinkingLevelMap`（见本章开头），并对所有 api 协议通用，而不再局限于 OpenAI Completions 的 compat。

为什么需要这个 compat 层？因为 "OpenAI 兼容" 是一个光谱，不是一个二元属性。各种 provider（OpenRouter、Vercel AI Gateway、Ollama、LM Studio、vLLM）声称兼容 OpenAI API，但具体兼容到什么程度各不相同：

- 有的不支持 `developer` role（只支持 `system`）
- 有的用 `max_tokens` 而不是 `max_completion_tokens`
- 有的有自己的 reasoning/thinking 格式
- 有的不在 streaming 中返回 usage 信息

`compat` 字段把这些差异编码到模型定义中，让 provider 实现可以根据这些标志调整请求格式。这避免了为每个 "OpenAI 兼容" 的 provider 写一个单独的 provider 实现。

并行的图像生成目录 —— 事件流的一个反例

v0.74.1 起，pi-ai 多出一套**与文本侧完全并行、但彼此独立**的图像生成子系统，且它跟着文本侧一起经历了 v0.80.0 的同一次范式迁移。现行形态与文本侧同构：`images-models.ts` 的 `ImagesModels` 集合（`createImagesModels`、`ImagesProvider`、`MutableImagesModels.setProvider`）对应文本侧的 `Models` / `createModels`，`providers/all.ts` 的 `builtinImagesProviders()` / `builtinImagesModels()`（`all.ts:140` / `:145`）对应文本侧的 `builtinProviders()` / `builtinModels()`，图像模型目录用 `image-models.generated.ts`。类型有 `ImagesModel<TApi>`（`types.ts:778`）、`ImagesFunction`（`:317`）、`AssistantImages`（`:434`）。内建支持 OpenRouter 图像生成（`KnownImagesApi = "openrouter-images"`）。

> **历史对照**：早期图像侧的枢纽是与旧文本注册表同构的全局 `images-api-registry.ts`（`registerImagesApiProvider` / `getImagesApiProvider`）。它和文本侧的 `api-registry.ts` 一样，在 v0.80.0 后退居 `/compat` 临时入口（`compat.ts` 仍导出 `images-api-registry.ts` / `images.ts` / `providers/images/register-builtins.ts`），现行代码走 `images-models.ts` 集合。

值得注意的是它在一个设计点上**与文本生成相反**。文本侧的 `StreamFunction` 返回事件流（第 6 章论证了"为什么是事件流"），而图像侧的核心函数签名是：

```
// packages/ai/src/types.ts:317-321
export type ImagesFunction<TApi, TOptions> = (
  model: ImagesModel<TApi>,
  ...
) => Promise<AssistantImages>;
```

它返回的是 `Promise<AssistantImages>`，**不是事件流**。原因正好印证第 6 章的论点：事件流的价值在于"增量产出 + 渐进式消费"——文本是一个 token 一个 token 流出来的，UI 需要边收边渲染。而图像生成没有有意义的中间增量（用户要的是最终那张图），于是退回最简单的 `Promise`。同一个团队、同一套集合 + factory 模式，却因为**输出是否有增量价值**而选择了不同的返回类型。这是"按场景选择抽象"而非"教条地统一抽象"的一个清晰例证。

取舍分析

得到了什么

动态性。系统可以支持任何 LLM — 只要有人写了 provider 并定义了 model。内建目录覆盖主流模型， `models.json` 覆盖长尾需求。

构建时确定性。内建模型目录在构建时生成，运行时不依赖外部 API。这让 pi 在断网环境下也能正常列出模型（虽然调用模型仍然需要网络）。

渐进式覆盖。用户可以从最小配置开始（只写一个 `id`），逐步添加更多参数。override 机制支持 partial merge — 只覆盖需要改的字段。

放弃了什么

运行时才知道模型是否可用。用户在 `models.json` 中定义了一个模型，但如果对应的 provider 没有注册（extension 没加载或 API key 没配置），错误只在实际调用时才暴露。目前没有 "预检" 机制在启动时验证所有模型的可用性。

compat 层的维护成本。每当一个新的 "OpenAI 兼容" provider 出现，并且有新的不兼容点，就需要在 compat schema 中添加新的字段。这是一个持续增长的配置面。

版本演化说明

> 本章内容已对照 pi-mono **v0.82.1**。本章横跨 pi-ai 内建目录与 coding-agent 侧模型管理，两侧本区间均有破坏性调整。
>
> **pi-ai 内建目录侧：**
>
> - **内建目录拆包（v0.80.0）**：单体 `models.generated.ts` 拆为 per-provider `providers/*.models.ts` + `providers/all.ts` 聚合，可 tree-shake。读法改用 `getBuiltinModel(s)`（`all.ts:59` / `:77`），不再直接读 `MODELS`。
> - **新鲜度接口（v0.82.0，Breaking）**：`getBuiltinModelDataUrl(provider)` 被 `getBuiltinModelDataGeneratedAt()`（`all.ts:72`）取代，返回目录生成时间戳。

- **动态目录持久化（v0.80.8 起）**：pi-ai 侧新增 `ModelsStore` / `InMemoryModelsStore`（`models-store.ts:15` / `:28`）；`ModelsStoreEntry.etag`（v0.82.1）支持 `If-None-Match` 条件刷新；`Models.refresh({force})` 提供 per-provider 错误/取消隔离。
- **类型订正**：`Model.provider` 类型从 `Provider` 改为 `ProviderId`（`types.ts:753`，v0.80.0）。
- **思考档位映射**：`thinkingLevelMap`（顶层 Model 字段）取代 `compat.reasoningEffortMap`（v0.72.0），新增 `getSupportedThinkingLevels` / `clampThinkingLevel`，废弃 `supportsXhigh()`；`max` / `xhigh` 档位在本区间贯穿 CLI/SDK/RPC/主题。
- **图像生成子系统**：v0.74.1 新增并行的图像注册表与模型目录（见上节），返回 `Promise` 而非事件流。

coding-agent 侧模型管理（本轮重点）：

- **`ModelRuntime` 成为规范门面（v0.80.8，Breaking）**：异步 `implements Models` 的 `ModelRuntime`（`model-runtime.ts:94`，`create()` 于 `:133`）取代旧的同步 `ModelRegistry` 中心；`ModelRegistry` 降级为面向 extension 的**同步 compat 投影**（`model-registry.ts:20`），构造改为 `new ModelRegistry(runtime)`，`refresh()` 从 `void` 改为 **`Promise<void>`**。
- **models.json schema 迁到 `ModelConfig`**：schema/解析从 `model-registry.ts` 迁到 `model-config.ts`（`ModelDefinitionSchema:154`、`ModelOverrideSchema:169`）；override 应用迁到 `provider-composer.ts` 的 `applyModelOverride`（`:100`）+ `composeModelProvider`，作为最上层用户配置注入 pi-ai `Models`。
- **文件型动态目录 `models-store.json`（v0.80.8 起）**：`FileModelsStore`（`models-store.ts:28`）落盘 per-provider 动态目录；pi.dev 远程目录经 `withRemoteCatalog` 用 `If-None-Match` / `304` 条件刷新（`remote-catalog-provider.ts:70-112`），刷新移出启动路径；`pi update --models`（`package-manager-cli.ts:728`）强制重刷。
- **校验栈迁移**：`models.json` 校验用 `typebox` 1.x 的 `Compile` / `Check`（v0.69.0），适配禁用 `eval` 的运行时。

内建模型目录仍由自动化脚本在构建时生成；`models.json` 覆盖与 extension 动态注册的语义保持稳定，只是执行者从 `ModelRegistry` 换成了 `ModelRuntime` + pi-ai `Models`。

第 19 章：工具设计原则 — 约束即保护

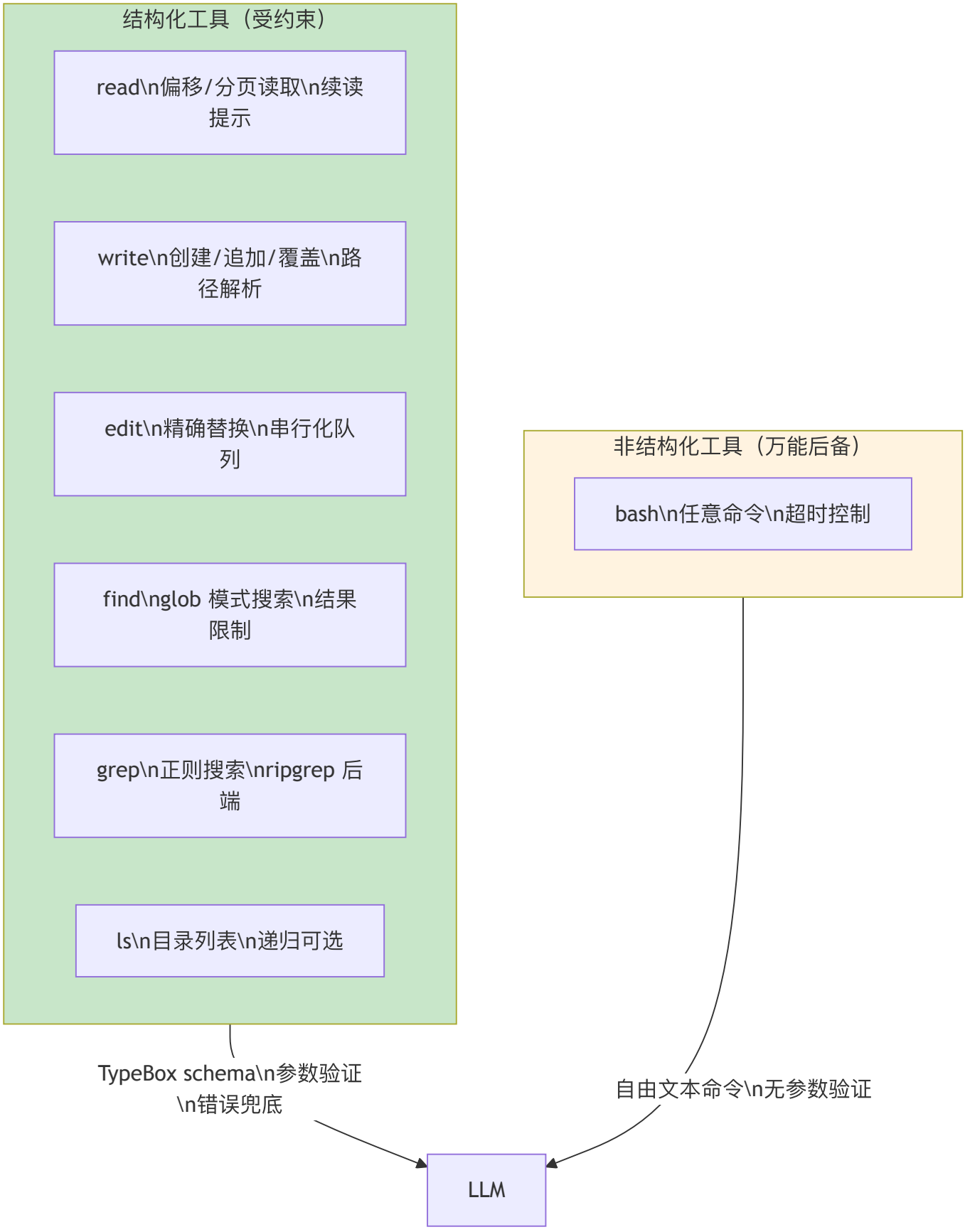
> **定位**：本章总论 pi 的工具设计哲学 — 为什么给 LLM 的接口越受约束，犯错越少。前置依赖：第 9 章（工具执行管道）。 适用场景：当你想理解 pi 为什么有 6 个结构化工具 + bash 后备，而不是只给一个 bash。

为什么不只给一个 bash?

这是本章的核心设计问题。

理论上，bash 能做一切 — 读文件用 `cat`、写文件用 `echo >`、搜索用 `grep`、编辑用 `sed`。一个万能工具，LLM 自己组合命令。

但 pi 选择了 **6 个结构化工具 + bash 后备**（共 7 个内置工具，工具名见 `allToolNames`：`read`、`write`、`edit`、`find`、`grep`、`ls`、`bash`，core/tools/index.ts:84）：



设计理由不是"bash 不好"，而是 **LLM 用结构化参数犯的错比用自由文本命令少得多。**

当 LLM 调用 `read({ path: "src/index.ts", offset: 50, limit: 30 })` 时，每个参数都有 TypeBox schema 验证。如果 `offset` 是字符串而非数字，验证立即失败，LLM 在下一轮修正。

当 LLM 拼 `cat src/index.ts | sed -n '50,79p'` 时，任何拼写错误、引号不匹配、管道符遗漏都会导致 bash 报一个模糊的错误，LLM 可能需要多轮才能修正。

工具如何定义：ToolDefinition 类型

pi 的每个工具都实现 `ToolDefinition` 接口。这是 extension API 层面的工具标准格式 — 不仅内置工具用它，第三方 extension 注册的自定义工具也用同一套接口：

```
// packages/coding-agent/src/core/extensions/types.ts:369-399

export interface ToolDefinition<
  TParams extends TSchema = TSchema,
  TDetails = unknown, TState = any
> {
  name: string;            // 工具名（LLM tool call 中使用）
  label: string;           // UI 显示的可读名称
  description: string;     // 给 LLM 的描述
  promptSnippet?: string; // system prompt 中的一行摘要
  promptGuidelines?: string[]; // 追加到 Guidelines 段落的指引
  parameters: TParams;     // TypeBox schema
  prepareArguments?: (args: unknown) => Static<TParams>;
  execute(
    toolCallId: string, params: Static<TParams>,
    signal: AbortSignal | undefined,
    onUpdate: AgentToolUpdateCallback<TDetails> | undefined,
    ctx: ExtensionContext,
  ): Promise<AgentToolResult<TDetails>>;
  renderCall?: (...) => Component;  // TUI 调用显示
  renderResult?: (...) => Component; // TUI 结果显示
}
```

这个接口的设计要点：

1. **`parameters` 是 TypeBox schema**，不是自由格式 JSON — 框架在调用 `execute` 之前自动做 schema 验证
2. **`prepareArguments` 是兼容性钩子** — 当 LLM 使用旧版参数格式时，在 schema 验证之前转换参数（edit 工具用这个把 `oldText/newText` 转为 `edits[]`，详见第 20 章）
3. **`promptSnippet` 和 `promptGuidelines`** — 工具不仅定义参数，还参与 system prompt 的组装。工具激活时，它的 guidelines 会自动注入到 prompt 中

受约束采样：让 provider 在解码时就保证结构（v0.82.0）

TypeBox schema 验证是**事后**的：LLM 先自由生成一段 JSON，pi 再拿 schema 去校验，失败就退回让它重试。这一节讲的是另一条更强的路 —— 让 provider 在**解码时**就把输出约束在合法结构里，从源头上不产生非法 tool call。承载它的是 `ToolDefinition` 上新增的一维契约字段 `constrainedSampling`：

```
// packages/coding-agent/src/core/extensions/types.ts:457
constrainedSampling?: false | ConstrainedSamplingConfig;

// packages/ai/src/types.ts:460-468
export type ConstrainedSamplingConfig =
  | { type: "json_schema"; strict: "prefer" | "require" }
  | { type: "grammar"; variants: GrammarVariants };  // Lark / regex
```

两种约束模式对应两类需求：

- **`json_schema`** —— 要求 provider 按工具的 JSON Schema 做严格解码。`strict` 有 `"prefer"`（能严格就严格，provider 不支持则降级为普通 tool call）和 `"require"`（必须严格，否则报错）两档 —— 这个"偏好 vs 强制"的区分让工具作者能在"尽量结构化"和"绝不接受非结构化输出"之间选位置。
- **`grammar`** —— 用形式文法（Lark 语法或 regex）约束输出，适合参数不是普通 JSON、而是某种领域小语言的工具（如受限的查询表达式）。

关键在于：**这维契约不是无条件生效的，它由模型能力位把关**。模型目录里带两个 compat 能力位 —— `supportsStrictTools`（provider 是否接受严格 JSON-schema 工具定义，`model-config.ts:129`；内建 Anthropic 模型在生成的 metadata 里默认开启）和 `supportsGrammarTools`（是否支持自定义文法工具，OpenAI GPT-5+、Codex、Azure OpenAI 等开启）。工具声明了 `constrainedSampling`，但只有当**当前模型的对应能力位为真**时才会真正下发严格/文法约束，否则回退成普通 tool call。这样同一个工具定义能跨不同能力的模型工作，而不必为每个 provider 分叉。

这正是"约束即保护"在协议层的延伸：TypeBox 是 pi 侧的事后校验，`constrainedSampling` 是 provider 侧的事前保证，两者叠加把"LLM 发出结构非法的 tool call"这类错误压到最低。

七个内置工具的 Schema 一览

每个工具的参数定义都是 `Type.Object`（6 个结构化工具加上 bash 后备，共 7 个）——以下并排展示它们的参数结构，方便看出设计差异：

```
// packages/coding-agent/src/core/tools/read.ts:17-21
const readSchema = Type.Object({
  path: Type.String({ description: "Path to the file to read" }),
  offset: Type.Optional(Type.Number({
    description: "Line number to start reading from (1-indexed)"
  })),
  limit: Type.Optional(Type.Number({
    description: "Maximum number of lines to read"
  })),
});
```

```
// packages/coding-agent/src/core/tools/write.ts:14-17
const writeSchema = Type.Object({
  path: Type.String({ description: "Path to the file to write" }),
  content: Type.String({ description: "Content to write to the file" }),
});
```

```
// packages/coding-agent/src/core/tools/edit.ts:35-44
const editSchema = Type.Object({
  path: Type.String({ description: "Path to the file to edit" }),
  edits: Type.Array(Type.Object({
    oldText: Type.String({ description: "Exact text for one targeted
replacement. Must be unique." }),
    newText: Type.String({ description: "Replacement text for this
targeted edit." }),
  })),
});
```

```
// packages/coding-agent/src/core/tools/bash.ts:27-30
const bashSchema = Type.Object({
  command: Type.String({ description: "Bash command to execute" }),
  timeout: Type.Optional(Type.Number({ description: "Timeout in seconds"
})),
});
```

```
// packages/coding-agent/src/core/tools/find.ts:20-26
const findSchema = Type.Object({
  pattern: Type.String({ description: "Glob pattern to match files" }),
  path: Type.Optional(Type.String({ description: "Directory to search
in" })),
  limit: Type.Optional(Type.Number({ description: "Maximum number of
results (default: 1000)" })),
});

// packages/coding-agent/src/core/tools/grep.ts:23-35
const grepSchema = Type.Object({
  pattern: Type.String({ description: "Search pattern (regex or
literal)" }),
  path: Type.Optional(Type.String({ description: "Directory or file to
search" })),
  glob: Type.Optional(Type.String({ description: "Filter files by glob
pattern" })),
  ignoreCase: Type.Optional(Type.Boolean()),
  literal: Type.Optional(Type.Boolean()),
  context: Type.Optional(Type.Number()),
  limit: Type.Optional(Type.Number({ description: "Max matches (default:
100)" })),
});

// packages/coding-agent/src/core/tools/ls.ts:13-16
const lsSchema = Type.Object({
  path: Type.Optional(Type.String({ description: "Directory to list"
})),
  limit: Type.Optional(Type.Number({ description: "Max entries (default:
500)" })),
});
```

注意几个跨工具的设计模式：

- **`path` 参数**：几乎所有工具都有，类型一律是 `Type.String`，路径解析由 `resolveToCwd()` 在 execute 中处理
- **`limit` 参数**：搜索类和列表类工具都有输出数量限制 — 防止 LLM 收到几万行输出
- **bash 是唯一没有 `path` 的工具** — 它接收自由文本 `command`，本质上是结构化程度最低的工具

tool-definition-wrapper：从 extension 到 runtime

pi 有两层工具抽象：`ToolDefinition`（extension API 层）和 `AgentTool`（agent runtime 层）。内置工具定义为 `ToolDefinition`，但 agent runtime 消费的是 `AgentTool`。`wrapToolDefinition` 负责这个转换：

```
// packages/coding-agent/src/core/tools/tool-definition-wrapper.ts:5-18

export function wrapToolDefinition<TDetails = unknown>(
  definition: ToolDefinition<any, TDetails>,
  ctxFactory?: () => ExtensionContext,
): AgentTool<any, TDetails> {
  return {
    name: definition.name,
    label: definition.label,
    description: definition.description,
    parameters: definition.parameters,
    prepareArguments: definition.prepareArguments,
    execute: (toolCallId, params, signal, onUpdate) =>
      definition.execute(
        toolCallId, params, signal, onUpdate,
        ctxFactory?.() as ExtensionContext
      ),
  };
}
```

这个 wrapper 的关键作用是**注入 `ExtensionContext`**。`ToolDefinition.execute` 需要 5 个参数（包括 `ctx`），而 `AgentTool.execute` 只有 4 个。wrapper 在调用时通过 `ctxFactory` 自动注入 context — 让工具定义可以访问当前会话的 cwd、配置等环境信息，而 agent runtime 不需要关心这些细节。

反向也有支持 — `createToolDefinitionFromAgentTool` 把一个 `AgentTool` 包装回 `ToolDefinition`，用于外部提供的工具覆盖需要进入 definition-first 注册表的场景。

工具激活：名字白名单 + cwd 工厂

v0.68.0 重写了"宿主如何选择启用哪些工具"的模型。早期 pi 导出一组**预构建的工具对象**（`readTool` 、 `bashTool` 、 `codingTools` 等）供调用方直接拼装。现在改为两个更小的原语：一个**字符串名字白名单**，加上**按 cwd 构造**的工厂函数。

```
// packages/coding-agent/src/core/tools/index.ts:83-117（节选）
export type ToolName = "read" | "bash" | "edit" | "write" | "grep" |
"find" | "ls";
export const allToolNames: Set<ToolName> =
  new Set(["read", "bash", "edit", "write", "grep", "find", "ls"]);

// 按名字 + cwd 构造单个工具
export function createTool(toolName: ToolName, cwd: string, options?:
ToolsOptions): Tool;
// 一次性构造全部编码工具
export function createCodingTools(cwd: string, options?: ToolsOptions):
Tool[];
```

两个设计点：

1. **激活是名字白名单**。SDK 调用方通过 `createAgentSession({ tools: ["read", "bash"] })` （`tools?: string[]` ，sdk.ts:67）声明启用哪些工具；CLI 上则有 `--tools` / `--no-tools` （全禁）/ `--no-builtin-tools` / `--exclude-tools` 四个开关。白名单里既可以写内建工具名，也可以写 extension 注册的工具名 —— 两者在选择层面一视同仁（第 15 章）。
2. **构造需要显式 cwd**。注意 `createTool(name, cwd, options)` 和 `createCodingTools(cwd)` 都把 `cwd` 作为必传参数。工具不再隐式读取 `process.cwd()` ，而是绑定到调用方传入的工作目录 —— 这与第 14 章 system prompt、第 17 章 resource loader 的"去 process-global"是同一条纪律，也是 pi 能在一个进程里并存多个工作目录（RPC、SDK、子 agent）的前提。

截断策略：`truncate.ts`

LLM 的 context window 是有限的。当 `read` 读了一个 50000 行的日志，或者 `grep` 匹配了 10000 条结果，把全部输出塞进 tool result 会浪费 token、甚至超出限制。

`truncate.ts` 提供了统一的截断策略，**所有工具共享同一套逻辑**：

```
// packages/coding-agent/src/core/tools/truncate.ts:11-13

export const DEFAULT_MAX_LINES = 2000;
export const DEFAULT_MAX_BYTES = 50 * 1024; // 50KB
export const GREP_MAX_LINE_LENGTH = 500;
```

截断基于两个独立限制 — **先触及的那个生效**：

- **行数限制**（默认 2000 行）：防止行数爆炸
- **字节限制**（默认 50KB）：防止单行超长（如 minified JSON）

两种截断方向对应不同场景：

函数	方向	适用场景
`truncateHead`	保留开头	`read` 、 `find` 、 `grep` — 文件开头和前 N 条结果通常最有用
`truncateTail`	保留结尾	`bash` — 命令输出的最后几行通常包含错误信息或最终结果

截断不会产生半行 — `truncateHead` 只保留完整行。如果第一行就超过字节限制，返回空内容并设置 `firstLineExceedsLimit: true` ，让上层决定如何提示 LLM。

截断后的 `TruncationResult` 携带完整的元信息（总行数、总字节、输出行数、哪个限制被触发），工具可以据此生成 "Showing 2000 of 45000 lines (truncated)" 这样的提示。

具体对比：bash vs 专用工具

以"在项目中搜索所有包含 `TODO` 的 TypeScript 文件"为例，对比两种方式：

方式 A：LLM 使用 bash

```
{ "command": "grep -r 'TODO' --include='*.ts' . | head -100" }
```

问题链：

1. LLM 可能忘记 `--include` 的引号，导致 glob 展开
2. 输出是纯文本，没有结构化的行数限制 — `head -100` 只是近似控制

3. 如果项目有 `node_modules`，grep 会扫描依赖目录，输出爆炸
4. 输出没有截断元信息 — LLM 不知道总共有多少匹配
5. 不同操作系统的 grep 行为可能不同（macOS BSD grep vs GNU grep）

方式 B：LLM 使用 grep 工具

```
{
  "pattern": "TODO",
  "glob": "*.ts",
  "limit": 100
}
```

优势链：

1. 参数是结构化 JSON — `pattern`、`glob`、`limit` 各自独立，不存在引号嵌套问题
2. 后端用 ripgrep（`rg`），自动跳过 `.gitignore` 中的目录
3. `limit` 由工具实现控制，准确地只返回 100 条
4. 输出经过 `truncateHead` 处理，附带 `TruncationResult` 元信息
5. 每行长度被 `GREP_MAX_LINE_LENGTH`（500 字符）截断，防止 minified 文件的匹配行吃掉 context

第二种方式的 token 消耗更可预测，错误更少，LLM 需要的重试次数更低。这就是"约束即保护"的具体含义。

Pluggable I/O

内置工具大多提供了 `Operations` 接口（bash 用 `exec`）：

```
// packages/coding-agent/src/core/tools/edit.ts:63-70

interface EditOperations {
  readFile: (path: string) => Promise<Buffer>;
  writeFile: (path: string, content: string) => Promise<void>;
  access: (path: string) => Promise<void>;
}
```

默认实现用本地文件系统。但这个接口可以被替换为 SSH、Docker volume、甚至远程 API — 让同一套工具在不同环境中工作。每个工具的 Operations 接口只暴露它需要的最小操作集：

工具	Operations 方法
read	readFile, access, detectImageMimeType?
write	writeFile, access, mkdirp
edit	readFile, writeFile, access
bash	exec
find	exists, glob
grep	isDirectory, readFile

取舍分析

得到了什么

更低的出错率。结构化参数 + TypeBox 验证 = LLM 犯错时快速反馈。bash 是后备，不是首选。

可预测的 token 消耗。每个工具的输出都经过 `truncate.ts` 的统一截断 — 2000 行或 50KB 的上限确保 tool result 不会意外吃掉大量 context window。

跨环境一致性。Pluggable Operations 让工具在本地文件系统、SSH 远程、Docker 容器中行为一致。bash 命令在不同环境中可能有不同的 shell、不同的工具版本，专用工具则抽象了这些差异。

放弃了什么

更多的工具选择负担。LLM 需要在 6 个结构化工具 + bash 中选择正确的一个。注意 system prompt 不再用"优先用 grep/find/ls 而非 bash"这类指引去替模型选工具（该指引已于 v0.77.0 删除，详见第 14 章）—— 现在的判断是：工具的取舍由"哪些工具被激活"表达，而非靠 prompt 文字。

灵活性受限。有些操作在 bash 中一行命令就能完成（如 `wc -l *.ts | sort -n`），但没有对应的专用工具。pi 的策略是：对于**高频且容易出错**的操作（读、写、编辑、搜索）提供专用工具，其他操作留给 bash。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。6 个结构化工具（read/write/edit/find/grep/ls）+ bash 后备是逐步从 bash 中分离出来的 — 早期版本只有 bash + read + edit，后来加入 write、find、grep、ls，工具名集中在 `allToolNames`。主要设计级变化：① 工具契约新增**受约束采样**维度 `constrainedSampling`（`extensions/types.ts:457`，v0.82.0）—— prefer/require 严格 JSON Schema 或 Lark/regex 文法，由 `supportsStrictTools` / `supportsGrammarTools` 能力位把关；② v0.68.0 重写工具激活模型 —— 从导出预构建工具对象改为"字符串名字白名单 + cwd 工厂"（`createTool(name, cwd)` / `createCodingTools(cwd)`），SDK 用 `tools: string[]` 选择、CLI 用 `--tools` / `--no-tools` / `--no-builtin-tools` / `--exclude-tools`。`ToolDefinition` 接口在 extension 系统（第 15 章）引入后统一；参数 schema 改用 `typebox` 1.x（第 15 章）。

第 20 章：edit 的设计 — 为什么不能直接写文件

> **定位**：本章深入 edit 工具 — pi 工具设计原则的最佳体现。 前置依赖：第 19 章（工具设计原则）。 适用场景：当你想理解为什么编辑操作要用精确替换而非行号，或者想理解并发文件写入的安全性。

为什么不让 LLM 直接 write 整个文件?

这是本章的核心设计问题。

最简单的编辑方式是"读出来 → 改 → 写回去"。但 LLM 生成的"改后全文"可能遗漏原文的部分内容、改变缩进、引入多余的空行。文件越大，出错概率越高。

pi 的 edit 工具用**精确替换**（oldText → newText）而非全文重写。

完整的 editSchema

edit 的参数定义是所有工具中最复杂的 — 它有嵌套的数组结构：

```
// packages/coding-agent/src/core/tools/edit.ts:24-44

const replaceEditSchema = Type.Object(
  {
    oldText: Type.String({
      description: "Exact text for one targeted replacement. "
        + "It must be unique in the original file and must not "
        + "overlap with any other edits[].oldText in the same call.",
    }),
    newText: Type.String({
      description: "Replacement text for this targeted edit.",
    }),
  },
  { additionalProperties: false },
);

const editSchema = Type.Object(
  {
    path: Type.String({
      description: "Path to the file to edit (relative or absolute)",
    }),
    edits: Type.Array(replaceEditSchema, {
      description: "One or more targeted replacements. Each edit is "
        + "matched against the original file, not incrementally.",
    }),
  },
  { additionalProperties: false },
);
```

`description` 中的约束条件是给 LLM 看的设计文档 — 它们告诉 LLM：

1. **`oldText` 必须在文件中唯一** — 如果存在多处匹配，操作会失败
2. **多个 edit 不能重叠** — 避免替换冲突
3. **所有 edit 基于原始文件匹配，不是递增的** — 第二个 edit 看到的是原文，不是第一个 edit 修改后的结果
4. **`additionalProperties: false`** — TypeBox 验证会拒绝任何多余字段

执行流水线

edit 操作从参数到写入文件经过一条精确的流水线。每个步骤都有明确的职责：

为什么需要这么多步骤？因为 LLM 生成的 `oldText` 不会包含 BOM 或 `\r\n` — 它只会生成 `\n` 换行。如果直接在原始内容中搜索 `oldText`，Windows 风格的文件几乎一定匹配失败。

流水线的核心是**先归一化、再匹配、再恢复**：

```
原始文件（可能有 BOM、\r\n）
  ↓ stripBom + normalizeToLF
归一化内容（纯 \n）
  ↓ applyEditsToNormalizedContent（在归一化空间中匹配和替换）
修改后内容（纯 \n）
  ↓ restoreLineEndings（如果原始文件用 \r\n，恢复回去）
最终内容（保持原始行尾风格）
  ↓ writeFile
```

这样 LLM 不需要关心目标文件是 Unix 还是 Windows 格式 — 框架自动处理。

edit-diff.ts：行尾归一化与 fuzzy matching

edit-diff.ts 是 edit 工具的核心算法模块。除了行尾归一化，它还实现了**模糊匹配**：

```ts
// packages/coding-agent/src/core/tools/edit-diff.ts:34-55

export function normalizeForFuzzyMatch(text: string): string {
  return (
    text
      .normalize("NFKC")
      .split("\n").map((line) => line.trimEnd()).join("\n")
      // Smart single quotes → '
      .replace(/[\u2018\u2019\u201A\u201B]/g, "'")
      // Smart double quotes → "
      .replace(/[\u201C\u201D\u201E\u201F]/g, '"')
      // Various dashes/hyphens → -
      .replace(/[\u2010-\u2015\u2212]/g, "-")
      // Special spaces → regular space
      .replace(/[\u00A0\u2002-\u200A\u202F\u205F\u3000]/g, " ")
  );
}
```

为什么需要 fuzzy matching？因为 LLM 经常在输出中用 smart quotes（""）代替 ASCII quotes（""），或者在行尾多加空格。`normalizeForFuzzyMatch` 对两端都做归一化，消除这类微小差异。

匹配策略是**精确优先、模糊兜底**：

```
// packages/coding-agent/src/core/tools/edit-diff.ts:96-134

export function fuzzyFindText(content: string, oldText: string)
  : FuzzyMatchResult {
  // 先尝试精确匹配
  const exactIndex = content.indexOf(oldText);
  if (exactIndex !== -1) {
    return { found: true, index: exactIndex, ... ,
             usedFuzzyMatch: false };
  }
  // 精确匹配失败，尝试模糊匹配
  const fuzzyContent = normalizeForFuzzyMatch(content);
  const fuzzyOldText = normalizeForFuzzyMatch(oldText);
  const fuzzyIndex = fuzzyContent.indexOf(fuzzyOldText);
  if (fuzzyIndex === -1) {
    return { found: false, ... };
  }
  return { found: true, index: fuzzyIndex, ... ,
           usedFuzzyMatch: true };
}
```

`applyEditsToNormalizedContent` 是多 edit 替换的核心。它的设计确保了多个 edit 不会互相干扰：

```
// packages/coding-agent/src/core/tools/edit-diff.ts:193-260（简化）

export function applyEditsToNormalizedContent(
  normalizedContent: string, edits: Edit[], path: string,
): AppliedEditsResult {
  // 1. 所有 edit 的 oldText/newText 归一化为 LF
  // 2. 验证：oldText 不能为空
  // 3. 每个 edit 在原始内容中匹配（不是递增）
  // 4. 验证：每个 oldText 必须唯一（出现次数 === 1）
  // 5. 按匹配位置排序
  // 6. 检测重叠：相邻 edit 的匹配范围不能交叉
  // 7. 从后往前替换（reverse order），保持前面的偏移量不变
  return { baseContent, newContent };
}
```

从后往前替换是关键技巧 — 替换后面的文本不会改变前面文本的偏移量，所以所有 edit 可以安全地使用它们在原始内容中匹配到的位置。

> **边界修正（v0.79.9，#5899）**：当走**模糊匹配**时，替换只作用在匹配到的那一段行块上，**原文其余未改动的行块被原样保留**（`edit-diff.ts:123` 的"preserving unchanged line blocks from the original"）。早期实现会把整个文件先归一化再重写，结果是即使只改一行，未匹配区域的行尾/smart quotes 等也可能被归一化"顺手"改掉；修复后模糊匹配不再整文件重写，只动它真正命中的块。

`generateDiffString` 生成带行号的 unified diff，返回给 LLM 作为 tool result。它还返回 `firstChangedLine` — 用于 TUI 中自动跳转到修改位置：

```
// packages/coding-agent/src/core/tools/edit-diff.ts:266-270

export function generateDiffString(
  oldContent: string, newContent: string, contextLines = 4,
): { diff: string; firstChangedLine: number | undefined } {
  // 使用 `diff` 库的 diffLines 计算差异
  // 输出格式：+行号 添加的行 / -行号 删除的行 / 空格行号 上下文
```

除了给 LLM 看的带行号 diff，edit 执行后还会用 `generateUnifiedPatch(path, baseContent, newContent)` 生成一份**标准 unified patch** 附在结果 details 里（edit.ts:350-351）。这让下游（IDE 扩展、外部工具）可以拿到能直接 `git apply` 的补丁，而不必自己解析带行号的展示型 diff。从 v0.79.7 起，`generateDiffString`、`generateUnifiedPatch`、`EditDiffResult` 已作为**公共 API 从包根导出**

（index.ts:249）—— edit 的 diff 能力不再是工具内部细节，而是可被 extension/SDK 复用的构件。

`file-mutation-queue`：并发安全

当 LLM 在 parallel 模式下同时发起多个 edit 调用（比如"同时修改 3 个文件"），可能有两个 edit 操作指向同一个文件。`file-mutation-queue` 确保同一个文件的写操作被串行化：

```ts
// packages/coding-agent/src/core/tools/file-mutation-queue.ts:1-39

const fileMutationQueues = new Map<string, Promise<void>>();

function getMutationQueueKey(filePath: string): string {
  const resolvedPath = resolve(filePath);
  try {
    return realpathSync.native(resolvedPath); // 解析符号链接
  } catch {
    return resolvedPath;
  }
}

export async function withFileMutationQueue<T>(
  filePath: string, fn: () => Promise<T>,
): Promise<T> {
  const key = getMutationQueueKey(filePath);
  const currentQueue = fileMutationQueues.get(key)
    ?? Promise.resolve();

  let releaseNext!: () => void;
  const nextQueue = new Promise<void>((resolve) => {
    releaseNext = resolve;
  });
  const chainedQueue = currentQueue.then(() => nextQueue);
  fileMutationQueues.set(key, chainedQueue);

  await currentQueue;          // 等前一个操作完成
  try {
    return await fn();         // 执行本次操作
  } finally {
    releaseNext();             // 释放锁，让下一个操作开始
    if (fileMutationQueues.get(key) === chainedQueue) {
      fileMutationQueues.delete(key); // 清理：没有后续排队时删除条目
    }
  }
}
```

这 39 行代码是整个 `file-mutation-queue.ts` 的全部。设计要点：

1. **基于 Promise 链的无锁串行化** — 不用 mutex，用 Promise 的 `.then()` 链来排队
2. **`realpathSync.native` 解析符号链接** — 确保 `./src/foo.ts` 和 `../project/src/foo.ts` 如果指向同一个文件，用同一个锁
3. **自动清理** — 当没有后续操作排队时删除 Map 条目，避免内存泄漏

4. **不同文件仍然并行** — 只有指向同一文件的操作才串行

具体示例：一次 edit 操作的前后

假设 LLM 要在 `src/config.ts` 中把默认超时从 30 秒改为 60 秒。

文件原始内容：

```
// src/config.ts
export const DEFAULT_CONFIG = {
  timeout: 30_000,
  retries: 3,
  baseUrl: "https://api.example.com",
};
```

LLM 发出的 tool call：

```
{
  "path": "src/config.ts",
  "edits": [{
    "oldText": "  timeout: 30_000,",
    "newText": "  timeout: 60_000,  // increased for slow networks"
  }]
}
```

执行过程：

1. `resolveToCwd("src/config.ts", cwd)` → `/home/user/project/src/config.ts`
2. `readFile` 读取原始内容
3. `stripBom` — 无 BOM，跳过
4. `detectLineEnding` → `"\n"`（Unix 格式）
5. `normalizeToLF` — 已经是 LF，无变化
6. `applyEditsToNormalizedContent` — 在内容中找到 `"  timeout: 30_000,"` 的唯一匹配，替换
7. `generateDiffString` 生成输出：

```
  ...
  2 export const DEFAULT_CONFIG = {
- 3   timeout: 30_000,
+ 3   timeout: 60_000,  // increased for slow networks
  4   retries: 3,
  ...
```

8. `restoreLineEndings` — 原始是 LF，无变化
9. `writeFile` 写回文件

返回给 LLM 的 tool result 包含 diff 文本和 `firstChangedLine: 3`。

如果匹配失败的情况：

假设 LLM 的 `oldText` 写成了 `"timeout: 30000,"` — 注意缺少缩进和下划线。精确匹配会失败，然后 fuzzy matching 也找不到（因为 `30000` 和 `30_000` 的差异不在模糊匹配的覆盖范围内）。此时 `applyEditsToNormalizedContent` 抛出：

```
Could not find the exact text in src/config.ts. The old text must
match exactly including all whitespace and newlines.
```

这个错误消息被返回给 LLM，LLM 可以用 `read` 重新查看文件内容，然后用正确的文本重试。

Legacy API 兼容：prepareArguments

edit 工具的 `prepareArguments` 钩子处理一个历史遗留问题。早期的 API 使用顶层的 `oldText` / `newText` 参数而非 `edits[]` 数组：

```
// packages/coding-agent/src/core/tools/edit.ts:95-117

function prepareEditArguments(input: unknown): EditToolInput {
  if (!input || typeof input !== "object") return input as
EditToolInput;
  const args = input as Record<string, unknown>;

  // 有些模型（Opus 4.6、GLM-5.1）把 edits 当成 JSON 字符串发来，先 parse
  if (typeof args.edits === "string") {
    try {
      const parsed = JSON.parse(args.edits);
      if (Array.isArray(parsed)) args.edits = parsed;
    } catch {}
  }

  const legacy = args as LegacyEditToolInput;
  if (typeof legacy.oldText !== "string"
    || typeof legacy.newText !== "string") {
    return args as EditToolInput;
  }
  // 把旧格式的 oldText/newText 合并到 edits 数组
  const edits = Array.isArray(legacy.edits) ? [...legacy.edits] : [];
  edits.push({ oldText: legacy.oldText, newText: legacy.newText });
  const { oldText: _oldText, newText: _newText, ...rest } = legacy;
  return { ...rest, edits } as EditToolInput;
}
```

注意开头新增的那段 JSON 字符串处理。`edits` 在 schema 里是数组，但实践中有些模型（注释里点名了 Opus 4.6、GLM-5.1）会把它**序列化成 JSON 字符串**再发出来。`prepareEditArguments` 在 schema 验证之前先尝试 `JSON.parse`，parse 成功且是数组就还原 —— 这是"宽容输入"原则对模型行为差异的又一次兜底，让同一个 edit 工具能容忍不同模型的 tool-call 习惯。

这个函数在 schema 验证**之前**运行（见第 9 章 `prepareArguments` 钩子）。它检测旧格式并转换为新格式：

```
旧格式：{ path, oldText, newText }
          ↓ prepareEditArguments
新格式：{ path, edits: [{ oldText, newText }] }
```

甚至支持混合格式 — 如果 LLM 同时传了 `edits[]` 和顶层 `oldText/newText`，两者会合并。这种宽容的输入处理让 API 升级不会破坏已有的 LLM 行为。

EditOperations：Pluggable I/O

edit 工具通过 `EditOperations` 接口抽象了文件 I/O：

```
// packages/coding-agent/src/core/tools/edit.ts:63-76

export interface EditOperations {
  readFile: (absolutePath: string) => Promise<Buffer>;
  writeFile: (absolutePath: string, content: string) => Promise<void>;
  access: (absolutePath: string) => Promise<void>;
}

const defaultEditOperations: EditOperations = {
  readFile: (path) => fsReadFile(path),
  writeFile: (path, content) => fsWriteFile(path, content, "utf-8"),
  access: (path) => fsAccess(path, constants.R_OK | constants.W_OK),
};
```

默认实现用 Node.js 的 `fs/promises` 。替换场景包括：

- **SSH 远程编辑** — `readFile` 通过 SSH 读取， `writeFile` 通过 SSH 写入
- **Docker 容器** — 通过 Docker API 读写容器内的文件
- **测试** — 注入 mock 实现，不接触真实文件系统

取舍分析

得到了什么

几乎消除"写错文件"的可能。精确替换要求 LLM 准确引用原文（`oldText` 必须在文件中唯一存在）。如果 LLM 引用了不存在的文本，操作失败并返回清晰的错误。

并发安全。 `file-mutation-queue` 用 39 行代码解决了并行 edit 的竞态条件，且不同文件仍然并行 — 只有同文件操作才串行。

跨平台行尾处理。BOM 剥离 + LF 归一化 + 行尾恢复的流水线让 LLM 不需要关心文件的行尾格式，减少了一整类匹配失败。

放弃了什么

限制了模型的表达方式。有时 LLM 想"删除第 42 到 50 行" — 但 edit 工具不支持行号操作，必须把那些行的内容作为 `oldText` 传入。这要求 LLM 先读文件、再精确引用。

大范围重构不友好。如果要修改一个文件的 20 处不同位置，需要 20 个 edit 条目，每个 `oldText` 都必须足够长以确保唯一性。这种场景下 `write` 工具（全文重写）可能更合适。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。Edit 从"行号编辑"演变为"精确替换"是重要的设计转变； `prepareArguments` （第 9 章）兼容旧版 API， `edits[]` 多编辑、fuzzy matching、行尾归一化、file-mutation-queue 均保持。主要演进：① 模糊匹配保留未改动行块、不再整文件重写（v0.79.9，#5899）；② `prepareArguments` 增加对 `edits` 为 JSON 字符串的容错（Opus 4.6/GLM-5.1 等模型，先 JSON.parse）、schema 允许模型多发的替换字段（v0.80.4）；③ 结果附带标准 unified patch， `generateDiffString` / `generateUnifiedPatch` / `EditDiffResult` 自 v0.79.7 起作为公共 API 导出。

第 21 章：read 的设计 — 为什么不是简单的 cat

> **定位**：本章解析 read 工具如何在"给 LLM 看文件"时做保护性设计。 前置依赖：第 19 章（工具设计原则）。 适用场景：当你想理解为什么 read 工具不直接 `cat` 文件给 LLM。

为什么不直接 cat?

`cat` 的问题是它没有任何保护。一个 100MB 的二进制文件会被原样输出到 LLM context 中，消耗全部 token 窗口而不产生任何有用信息。

pi 的 read 工具做了三层保护：

1. 偏移与分页。 `offset` 和 `limit` 参数让 LLM 可以只读文件的一部分。LLM 不需要一次加载整个大文件。

2. 截断策略。超过一定大小的输出会被截断，附带提示信息告诉 LLM "还有更多内容，请用偏移参数继续读"。

3. 续读提示。如果输出被截断，read 会附带续读提示告知 LLM "还有更多内容，请用 offset 参数继续读"。这让 LLM 可以按需增量读取大文件，而不是一次性加载。

Schema 定义：三个参数撑起整个读取逻辑

```
// packages/coding-agent/src/core/tools/read.ts:17-21
const readSchema = Type.Object({
  path: Type.String({
    description: "Path to the file to read (relative or absolute)"
  }),
  offset: Type.Optional(Type.Number({
    description: "Line number to start reading from (1-indexed)"
  })),
  limit: Type.Optional(Type.Number({
    description: "Maximum number of lines to read"
  })),
});
```

注意 `offset` 是 1-indexed。LLM 如果需要从第 42 行继续读，直接传 `offset: 42` 就可以。截断提示中会告知总行数和当前读到的位置，让 LLM 知道如何继续。

截断的双重限制

read 工具的截断不是简单的"只看前 N 行"。它有两个独立的限制条件，先触发的那个生效：

```
// packages/coding-agent/src/core/tools/truncate.ts:11-13
export const DEFAULT_MAX_LINES = 2000;
export const DEFAULT_MAX_BYTES = 50 * 1024; // 50KB
```

为什么需要两个限制？因为行数和字节数衡量的是不同维度的资源消耗：

- **2000 行限制**防止 LLM context 被大量短行填满（比如一个巨大的 JSON 文件，每行都很短但总行数惊人）
- **50KB 字节限制**防止少量超长行消耗大量 token（比如 minified JavaScript，一行就可能有几百 KB）

`TruncationResult` 类型记录了完整的截断元信息：

```
// packages/coding-agent/src/core/tools/truncate.ts:15-38
export interface TruncationResult {
  content: string;
  truncated: boolean;
  truncatedBy: "lines" | "bytes" | null;
  totalLines: number;
  totalBytes: number;
  outputLines: number;
  outputBytes: number;
  lastLinePartial: boolean;
  firstLineExceedsLimit: boolean;
  maxLines: number;
  maxBytes: number;
}
```

`firstLineExceedsLimit` 是一个有趣的边界情况处理：如果文件的第一行就超过了 50KB 限制（比如 minified CSS），read 不会返回一个被截断到一半的行，而是返回一条提示让 LLM 用 bash 的 `sed + head -c` 来读取。这比返回半截内容更有用 — 半截的 minified CSS 对 LLM 没有任何帮助。

图片读取：自动检测与 base64 编码

Read 工具不仅读文本 — 它能检测并正确返回图片文件。这对多模态 LLM 至关重要：

```
// packages/coding-agent/src/core/tools/read.ts:32-40
export interface ReadOperations {
  readFile: (absolutePath: string) => Promise<Buffer>;
  access: (absolutePath: string) => Promise<void>;
  detectImageMimeType?: (
    absolutePath: string
  ) => Promise<string | null | undefined>;
}
```

当 `detectImageMimeType` 返回非空值时，read 走图片路径而非文本路径。流程是：

1. 读取文件为 `Buffer`
2. 转换为 base64 编码字符串
3. 如果 `autoResizeImages` 开启（默认），调用 `resizeImage()` 将图片缩放到 2000x2000 以内

4. 返回一个包含 `TextContent`（描述信息）和 `ImageContent`（base64 数据）的数组

```
// packages/coding-agent/src/core/tools/read.ts:157-183
const mimeType = ops.detectImageMimeType
  ? await ops.detectImageMimeType(absolutePath)
  : undefined;

if (mimeType) {
  const buffer = await ops.readFile(absolutePath);
  const base64 = buffer.toString("base64");
  if (autoResizeImages) {
    const resized = await resizeImage({
      type: "image", data: base64, mimeType
    });
    if (!resized) {
      content = [{
        type: "text",
        text: `Read image file [${mimeType}]\n` +
          `[Image omitted: could not be resized ...]`
      }];
    } else {
      content = [
        { type: "text", text: `Read image file [${resized.mimeType}]` },
        { type: "image", data: resized.data, mimeType: resized.mimeType
}
      ];
    }
  }
}
```

`autoResizeImages` 的默认值是 `true`。这意味着用户截图一个 4K 显示器的屏幕（几 MB 的 PNG），read 会自动缩小到合理尺寸再发送给 LLM。这避免了图片 token 消耗失控 — API 按图片像素数计费，一张 4K 截图可能消耗几千 token。

resize 失败时不会报错，而是返回一条 "Image omitted" 的文本提示。这种优雅降级确保了 read 工具永远不会因为图片处理失败而让整个 tool call 失败。

非文本文件的处理边界

read 工具的 schema 只有 `path`、`offset`、`limit` 三个参数 — 没有 PDF 页码范围、Jupyter cell 选择等专用参数。它的设计重心是**文本文件和图片**。

对于 PDF 和 Jupyter notebook 等格式，read 工具不提供专用的参数化支持。如果 LLM 需要处理这些格式，通常会退回到 bash 工具使用专门的命令行工具（如 `pdftotext`）。这是"专用工具做专用事"原则的体现 — read 不试图成为万能的文件解析器。

ReadOperations 的 Pluggable 设计

和其他工具一样，read 通过接口抽象了底层操作：

```
// packages/coding-agent/src/core/tools/read.ts:42-46
const defaultReadOperations: ReadOperations = {
  readFile: (path) => fsReadFile(path),
  access: (path) => fsAccess(path, constants.R_OK),
  detectImageMimeType: detectSupportedImageMimeTypeFromFile,
};
```

默认实现直接调用 Node.js 的 `fs` 模块读取本地文件。但 `ReadOperations` 可以被替换为：

- **SSH 远程读取**：通过 SSH 连接读取远程服务器上的文件
- **Docker 容器读取**：读取容器内的文件系统
- **Git blob 读取**：直接从 git object store 读取文件内容

这种可插拔设计让 read 工具在不修改核心逻辑的情况下适应不同的执行环境。

TUI 中的渲染：语法高亮与折叠

read 结果在 TUI 中的展示也经过了精心设计。对于**普通代码/文本文件**，`formatReadResult` 函数会：

1. 根据文件扩展名检测语言（`getLanguageFromPath`）
2. 对代码内容做语法高亮（`highlightCode`）
3. 默认只显示前若干行（折叠预览），更多内容需要用户手动展开

4. 截断信息用 warning 色显示，提示 LLM 输出被限制了

但有一类文件走**更激进的折叠**：当未展开（`!context.expanded`）时，read 会先用 `getCompactReadClassification` 判断这是不是一个"资源型文件"——`SKILL.md`、pi 自身的文档、以及 `AGENTS.md` / `CLAUDE.md`（含全大写 `AGENTS.MD` / `CLAUDE.MD`，见 `COMPACT_RESOURCE_FILE_NAMES`，read.ts:37）。如果是，结果**折叠为单独一行**（如 `[skill] tdd (Ctrl+O to expand)` 或一行资源路径 + 选中的行范围），而不是前 10 行预览：

```
// packages/coding-agent/src/core/tools/read.ts:117-138（节选）
function getCompactReadClassification(args, cwd):
CompactReadClassification | undefined {
  const fileName = basename(resolveToCwd(rawPath, cwd));
  if (fileName === "SKILL.md") return { kind: "skill", label: ... };
  const docs = getPiDocsClassification(absolutePath);
  if (docs) return docs;
  if (COMPACT_RESOURCE_FILE_NAMES.has(fileName))   // AGENTS.md /
CLAUDE.md / 大写变体
    return { kind: "resource", label: ... };
  return undefined;
}
// 渲染时（read.ts:342）：!context.expanded 时才套用紧凑分类
```

为什么这些文件要折叠成一行？因为 agent 在一个会话里会**反复读取**它们（每次涉及项目规则、skill 内容时都可能 read 一遍），如果每次都在 TUI 里铺开 10 行预览，屏幕会被这些"基础设施文件"刷屏。折叠成一行 + `Ctrl+O` 展开，既不打扰用户，又保留按需查看的能力。

> **订正（v0.73.0 / v0.75.5）**：早期版本对所有文件一视同仁地"只显示前 10 行"。现在对 `AGENTS.md` / `CLAUDE.md` / `SKILL.md` /pi 文档等资源型文件改为**折叠为单行**（可 `Ctrl+O` 展开），普通文件仍是前若干行预览。

这里有一个微妙的分层：LLM 看到的是完整的截断后内容（最多 2000 行/50KB），但用户在 TUI 中看到的是折叠预览（普通文件前若干行、资源型文件单行）。两个折叠分别服务不同的受众 — LLM 需要足够的上下文做决策，用户只需要确认 read 读对了文件。

> **边界修正（v0.81.0，#6731）**：`formatReadResult` 现在多接一个 `isError` 参数（`read.ts:171`）。read **失败**时的输出（如"文件不存在"）本身不是文件内容，早期却被和成功结果一样按文件扩展名做**语法高亮**，读起来像是把错误信息误当成了

> 代码。修复后， `isError` 为真时既不套紧凑折叠、也不做语法高亮（ `!isError && … getLanguageFromPath(...)` ， `read.ts:179-180` ）—— 错误信息以朴素文本呈现，不再伪装成高亮代码。

取舍分析

得到了什么

安全的文件探索。LLM 不会因为读了一个大文件而耗尽 context。 `offset/limit` 配合续读提示让 read → read 的增量探索工作流更流畅。双重截断限制（行数 + 字节数）覆盖了不同类型文件的边界情况。

文本与图片支持。同一个 read 工具处理文本文件和图片，LLM 不需要为这两种常见格式学习不同的工具。图片通过自动检测 MIME 类型走 base64 路径。

可插拔的执行后端。 `ReadOperations` 接口让 read 工具可以适应本地、远程、容器等不同环境。

放弃了什么

截断丢失上下文。截断意味着 LLM 可能需要多次 read 调用才能获取完整信息。但相比一次性加载大文件耗尽 context window 的后果，这个开销值得。

图片 resize 可能丢失细节。自动缩放到 2000x2000 以内意味着 LLM 看不到高分辨率的细节。对于需要像素级精度的场景（比如 UI 截图中的小字体），这是一个潜在的问题。但对大多数场景，缩放后的图片已经足够。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。Read 工具支持文本（带双重截断保护）和图片（自动检测格式 + base64 + 自动 resize）。 主要设计级变化：① read **错误输出不再被当文件内容语法高亮**（v0.81.0，#6731， `formatReadResult` 加 `isError` 参数， `read.ts:171,179` ）；② v0.73.0/v0.75.5

> 引入 `getCompactReadClassification` —— `AGENTS.md` / `CLAUDE.md`（含大写）/ `SKILL.md` /pi 文档在未展开时折叠为单行（`Ctrl+O` 展开），不再统一"只显示前 10 行"；③ BMP 检测转 PNG（v0.80.3）。`ReadOperations` 的 pluggable 设计与图片支持保持不变；图片 resize 与 fs 读取在流式期间移入 worker（v0.76.0）。

第 22 章：bash 与外部世界的边界

> **定位**：本章解析 bash 工具的定位 — 万能后备而非首选工具。 前置依赖：第 19 章（工具设计原则）。 适用场景：当你想理解结构化工具和非结构化工具的关系。

bash 是后备，不是首选

bash 在 pi 里的定位是**万能后备**：用于没有专用工具覆盖的操作 — 安装依赖、运行测试、启动服务、执行 git 命令。读文件该用 read、编辑用 edit、搜索用 grep、查找文件用 find。

值得注意的是这个"后备"定位**如何被表达**。早期版本里，pi 的 system prompt 直接写着一条"当有专用工具时不要用 bash"的指引；到 v0.77.0，这类"用 prompt 教模型挑工具"的文字被删除了（详见第 14 章）。现在的判断是：工具的取舍应当由"**哪些工具被激活**"来表达 —— 如果一个会话同时激活了 grep/find/ls 和 bash，模型自然会用更顺手、反馈更好的结构化工具；只有在"仅有 bash、没有专用探索工具"时，prompt 才补一句让它用 bash 去做 ls/rg/find。bash 工具自己的 promptSnippet 也回归中性（`"Execute bash commands (ls, grep, find, etc.)"`，bash.ts:280）。

为什么倾向结构化工具不是审美偏好，而是工程约束：结构化工具有参数校验、自动截断、跨平台一致性，bash 没有。LLM 用 bash 搜索文件时可能忘记排除 `node_modules`，可能用了 macOS 特有的 `find` 参数，可能返回几万行输出。结构化工具替它兜住了这些风险。

Schema 定义：极简但完整

```
// packages/coding-agent/src/core/tools/bash.ts:27-30
const bashSchema = Type.Object({
  command: Type.String({
    description: "Bash command to execute"
  }),
  timeout: Type.Optional(Type.Number({
    description: "Timeout in seconds (optional, no default timeout)"
  })),
});
```

只有两个参数 — command 和 timeout。这和 read 的三参数、grep 的七参数形成了鲜明对比。bash 的 schema 越简单，LLM 的使用门槛越低。但代价是：bash 的一切细节（工作目录、环境变量、错误处理）都被压缩进了一个 command 字符串中。

`timeout` 是可选参数，没有默认值。这是一个有意的设计选择 — 大多数命令（`git status`、`npm install`）的执行时间差异巨大，设置一个统一的默认超时要么太短（中断正常操作）要么太长（无用）。让 LLM 根据命令性质自行决定超时更合理。

BashOperations：可插拔的执行后端

和 read 工具一样，bash 也通过接口抽象了执行环境：

```
// packages/coding-agent/src/core/tools/bash.ts:43-61
export interface BashOperations {
  exec: (
    command: string,
    cwd: string,
    options: {
      onData: (data: Buffer) => void;
      signal?: AbortSignal;
      timeout?: number;
      env?: NodeJS.ProcessEnv;
    },
  ) => Promise<{ exitCode: number | null }>;
}
```

这个接口的设计透露了几个关键决策：

流式输出。`onData` 回调接收 `Buffer` 而非等待执行完毕后返回字符串。这让 TUI 可以实时显示命令输出（比如 `npm install` 的进度条），而非等到命令结束才显示结果。

AbortSignal 支持。用户可以随时取消正在执行的命令。`signal` 参数传递到执行层，触发进程树的 kill。

返回值是 exitCode 而非输出内容。输出通过 `onData` 流式传递，返回值只关心"成功还是失败"。`exitCode` 为 `null` 表示进程被 kill（用户取消或超时）。

默认执行后端：本地 Shell

```
// packages/coding-agent/src/core/tools/bash.ts:69-127
export function createLocalBashOperations(): BashOperations {
  return {
    exec: (command, cwd, { onData, signal, timeout, env }) => {
      return new Promise((resolve, reject) => {
        const { shell, args } = getShellConfig();
        if (!existsSync(cwd)) {
          reject(new Error(
            `Working directory does not exist: ${cwd}`
          ));
          return;
        }
        const child = spawn(shell, [...args, command], {
          cwd,
          detached: true,
          env: env ?? getShellEnv(),
          stdio: ["ignore", "pipe", "pipe"],
        });
        // Stream stdout and stderr
        child.stdout?.on("data", onData);
        child.stderr?.on("data", onData);
        // ...timeout and abort handling...
      });
    },
  };
}
```

几个关键实现细节：

`detached: true`。创建独立的进程组，这样 `killProcessTree` 可以一次性杀掉主进程和它所有的子进程。没有这个选项，kill 主进程后子进程可能变成孤儿进程继续运行。

`stdio: ["ignore", "pipe", "pipe"]`。stdin 被忽略（LLM 不需要和命令交互），stdout 和 stderr 都被 pipe 出来通过 `onData` 回调传递。注意 stdout 和 stderr 合并到了同一个 `onData` — 这意味着输出顺序和终端中看到的一致，但没有办法区分标准输出和错误输出。

工作目录校验。在 spawn 之前检查 `cwd` 是否存在。这避免了一个常见的 debug 陷阱 — 如果工作目录不存在，`spawn` 会报一个含糊的错误，不如直接给出明确的错误信息。

超时处理

```
// packages/coding-agent/src/core/tools/bash.ts:84-92
let timedOut = false;
let timeoutHandle: NodeJS.Timeout | undefined;
if (timeout !== undefined && timeout > 0) {
  timeoutHandle = setTimeout(() => {
    timedOut = true;
    if (child.pid) killProcessTree(child.pid);
  }, timeout * 1000);
}
```

超时时不是简单地 reject promise — 而是先 kill 进程树，然后在进程退出回调中检查 `timedOut` 标志再 reject。这个顺序很重要：如果先 reject 再 kill，调用方可能在进程还在运行时就开始处理"超时"结果，导致输出数据竞争。

超时 reject 的错误信息格式是 `timeout:${timeout}` — 这个格式化的字符串允许上层精确知道超时值，用于生成更有用的提示信息（"命令在 30 秒后超时，考虑增加超时时间"）。

BashToolDetails 与结构化结果

bash 的执行结果不只是一个字符串。`BashToolDetails` 记录了截断元信息和完整输出文件路径：

```ts
// packages/coding-agent/src/core/tools/bash.ts:34-37
export interface BashToolDetails {
  truncation?: TruncationResult;
  fullOutputPath?: string;
}
```

`fullOutputPath` 指向一个临时文件，保存了命令的完整输出。当输出被截断时，LLM 可以通过 read 工具去读取这个临时文件获取完整内容。这是一种精巧的工具间协作 — bash 截断输出以保护 context，但提供了一条"逃生通道"让 LLM 在需要时可以看到全部内容。

临时文件路径的生成使用了 crypto random：

```ts
// packages/coding-agent/src/core/tools/bash.ts:22-25
function getTempFilePath(): string {
  const id = randomBytes(8).toString("hex");
  return join(tmpdir(), `pi-bash-${id}.log`);
}
```

BashSpawnHook：命令执行前的最后一道关卡

```ts
// packages/coding-agent/src/core/tools/bash.ts:130-136
export interface BashSpawnContext {
  command: string;
  cwd: string;
  env: NodeJS.ProcessEnv;
}

export type BashSpawnHook =
  (context: BashSpawnContext) => BashSpawnContext;
```

`BashSpawnHook` 在命令执行前被调用，可以修改命令、工作目录或环境变量。典型用途：

- **命令审计**：记录所有 LLM 执行的命令
- **命令改写**：在命令前添加 `set -e` 确保脚本在第一个错误时停止
- **环境注入**：添加特定的环境变量（比如 API key）

配合 `commandPrefix` 选项，可以在每条命令前自动添加前缀（比如 `source ~/.nvm/nvm.sh &&`），确保 shell 环境正确初始化。

输出截断策略

bash 的输出截断和 read 不同 — 它用的是 `truncateTail` 而非 `truncateHead`。这意味着 bash 保留的是**最后**的输出而非开头。

为什么？因为 bash 命令的关键信息通常在末尾 — 编译错误的最后几行、测试结果的 summary、安装完成的 success 信息。如果一个 `npm install` 输出了几千行依赖安装日志，LLM 需要看到的是最后的 "added 42 packages" 或 "ERR! missing dependency"，而不是开头的 "added foo@1.0.0"。

这和 read 的 `truncateHead`（保留开头）形成了有意的对比：读文件时，开头（函数签名、import 语句、文件头注释）更重要；执行命令时，结尾（结果、错误信息）更重要。

Sandbox 讨论：bash 的安全边界

bash 是安全风险最大的工具。它可以执行任意命令，包括 `rm -rf /`。pi 通过多个层次来控制这个风险：

1. `beforeToolCall` 钩子（第 9 章）。产品层可以通过这个钩子实现任意安全策略 — 命令白名单、确认弹窗、日志审计等。具体实现由上层决定。

2. 独立的执行器抽象。mom（Slack bot，第 28 章）通过 `Executor` 接口（`DockerExecutor` / `HostExecutor`）让所有命令在 Docker 容器中执行，限制 agent 的文件系统访问范围。这不是替换 `BashOperations`，而是 mom 的工具层自己封装了执行环境（详见第 28 章）。

3. 环境隔离。`BashSpawnHook` 可以过滤环境变量，防止 API key 等敏感信息泄露给 LLM 执行的命令。

这三层保护形成了一个从"提示确认"到"物理隔离"的安全梯度。不同的产品场景可以选择合适的安全级别 — 个人使用可能只需要第一层，企业部署可能需要三层全开。

TUI 中的实时渲染

bash 命令在 TUI 中有特殊的渲染逻辑。执行期间，TUI 实时显示输出（通过 `onData` 回调驱动 `requestRender`），并在命令旁边显示经过时间。完成后，输出被折叠为最多 5 行的预览：

```
// packages/coding-agent/src/core/tools/bash.ts:152
const BASH_PREVIEW_LINES = 5;
```

折叠状态下，用户可以展开查看完整输出。这个 UX 设计平衡了"不丢失信息"和"不让长输出淹没对话流"两个需求。

`excludeFromContext`：执行但不进上下文（v0.76.0）

有一类命令用户想**执行并看到输出，但不希望它进入 LLM 的上下文** —— 典型的是 `cat` 一个很大的日志、`env`（含敏感变量）、或纯粹给人看的探查命令。把这些塞进上下文既浪费 token，又可能污染后续推理。v0.76.0 为此加了 `excludeFromContext`。

```
// packages/coding-agent/src/core/agent-session.ts:2595-2598, 2639
async executeBash(
  command: string,
  onChunk?: (chunk: string) => void,
  options?: { excludeFromContext?: boolean; operations?: BashOperations
},
): Promise<BashResult> { /* ... */ }

// 结果记录到会话时带上该标记：
recordBashResult(command, result, options): void {
  const bashMessage: BashExecutionMessage = {
    role: "bashExecution", command, output: result.output,
    /* ... */ excludeFromContext: options?.excludeFromContext,
  };
}
```

交互界面上对应的语法是 **`!!` 前缀** —— 用户输入 `!!` 开头的命令时，它照常执行、输出照常显示在 TUI 里，但这条 `bashExecution` 记录被标记为 `excludeFromContext: true`，在装配发给 LLM 的上下文时被跳过。注意它**仍然写入会话文件**（持久化与"是否发

给模型"是两个正交维度，呼应第 11 章的会话模型）。RPC 模式下 `bash` 命令也透传这个选项（见第 26 章）。

另外两点与上下文相关的细节：bash 的输出从 v0.73.0 起**增量地进入模型上下文**（流式 chunk 边产生边累积，而非命令结束才一次性塞入），进程退出后还会继续 drain stdout/stderr 以免丢尾部输出；`shellPath` 与工作目录都跟随**会话当前 cwd**（`sessionManager.getCwd()`，agent-session.ts:2610），而非 pi 启动时的目录 —— 又一次"去 process-global"。

会话环境变量：把会话身份注入子进程（v0.82.0）

bash 执行的命令常常需要知道"我正跑在哪个会话里、用的什么模型"。v0.82.0 起，pi 在 spawn 子进程前会往它的环境里注入一组 **`PI_*` 会话变量**：

```
// packages/coding-agent/src/core/tools/bash.ts:166-180（节选）
const env = { ...getShellEnv() };
delete env.PI_SESSION_ID;    // 先清掉继承来的旧值，避免串味
delete env.PI_SESSION_FILE;
delete env.PI_PROVIDER;
delete env.PI_MODEL;
delete env.PI_REASONING_LEVEL;
if (exposeSessionEnvironment && ctx) {
  env.PI_SESSION_ID = ctx.sessionManager.getSessionId();
  const sessionFile = ctx.sessionManager.getSessionFile();
  if (sessionFile) env.PI_SESSION_FILE = sessionFile;
  if (ctx.model) {
    env.PI_PROVIDER = ctx.model.provider;
    env.PI_MODEL = ctx.model.id;
  }
  if (ctx.thinkingLevel) env.PI_REASONING_LEVEL = ctx.thinkingLevel;
}
```

五个变量各自对应一维会话身份：

- **`PI_SESSION_ID`** —— 当前会话 id（第 11 章的 UUIDv7）
- **`PI_SESSION_FILE`** —— 会话 JSONL 文件的绝对路径（若会话落盘）
- **`PI_PROVIDER`** —— 当前模型的 provider（如 `anthropic`）
- **`PI_MODEL`** —— 当前模型 id（如 `claude-opus-4-6`）
- **`PI_REASONING_LEVEL`** —— 当前 thinking 档位（如 `medium`）

有两个设计细节值得注意。其一，注入前**先 `delete` 一遍**：pi 自己可能就跑在一个带 `PI_*` 的环境里（比如嵌套调用），不清掉旧值，子进程会读到上一层的会话身份。其二，注入受 `exposeSessionEnvironment` 开关控制 —— 它默认开启，但宿主可以关掉，因为把会话文件路径暴露给任意 LLM 执行的命令并非所有场景都想要。

一个真实用例是第 26b 章的 **evals harness**：它用 `PI_PROVIDER / PI_MODEL` 从环境里读出要评测的模型（ `getRequiredModelSelection()` ），再装配 session。也就是说，这组会话变量不只是给用户脚本看的 —— 它已经是 pi 自己评测管线的输入。配套地，直连 RPC 的 bash 执行会通过流式 `bash_execution_update` 事件把输出增量吐给客户端（见第 26 章）。

取舍分析

得到了什么

灵活性兜底。当专用工具无法覆盖的场景出现时（比如一个特殊的 CLI 工具），bash 保证 agent 不会"束手无策"。

可适配的安全模型。从本地执行到 Docker sandbox， `BashOperations` 接口让安全边界可以按需收紧。

放弃了什么

bash 是安全风险最大的工具。可插拔架构只是提供了控制点，实际的安全策略需要产品层去实现。一个忘记配置 sandbox 的部署环境，bash 就是一个完全开放的后门。

输出解析不可靠。bash 返回的是纯文本，没有结构化信息。LLM 需要自己解析命令输出来判断成功还是失败（虽然 `exitCode` 提供了基本信号）。这是结构化工具（read、grep）相对 bash 的核心优势。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。Bash 工具的输出处理经历多次改进 — 输出 tail 截断、 `BashSpawnHook` 、 `fullOutputPath` 临时文件机制均

> 保持。 主要设计级变化：① 会话环境变量 `PI_SESSION_ID` / `PI_SESSION_FILE` / `PI_PROVIDER` / `PI_MODEL` / `PI_REASONING_LEVEL` 注入子进程（v0.82.0， `bash.ts:166-180` ，evals harness 依赖 `PI_PROVIDER` / `PI_MODEL` ），直连 RPC bash 流式 `bash_execution_update` （第 26 章）； ② `excludeFromContext` （ `!!` 前缀，v0.76.0）—— 命令执行并显示但输出不入 LLM 上下文； ③ 输出从 v0.73.0 起增量进入上下文，进程退出后继续 drain（v0.79.4）； ④ `shellPath` /cwd 跟随会话当前 cwd 而非启动 cwd （v0.68.0）。 此外 system prompt 不再硬性指引"有专用工具时勿用 bash"（v0.77.0，详见第 14 章）。

第 23 章：find 和 grep — 结构化搜索替代万能 bash

> **定位**：本章解析 find 和 grep 从 bash 中独立出来的设计理由。 前置依赖：第 19 章（工具设计原则）、第 22 章（bash 的定位）。 适用场景：当你想理解为什么"多一个工具"有时比"少一个工具"更好。

为什么要把搜索从 bash 中拆出来?

LLM 用 bash 搜索时的典型模式：

```
# LLM 可能会生成
find . -name "*.ts" -not -path "node_modules/*" | head -100
grep -rn "registerProvider" --include="*.ts" src/
```

这些命令有几个问题：

1. **平台不一致**。macOS 的 `find` 和 Linux 的 `find` 参数不完全相同
2. **node_modules 陷阱**。LLM 经常忘记排除 `node_modules`，导致返回几万个结果
3. **结果截断不可控**。`head -100` 是 LLM 的猜测，不是系统的保护

pi 把搜索拆成两个结构化工具 — find（按文件名搜索）和 grep（按内容搜索） — 每个都有明确的参数定义、自动保护和跨平台一致性。

两个 Schema 对比

把 find 和 grep 的 schema 放在一起看，可以清楚地看到它们各自的职责边界：

```
// packages/coding-agent/src/core/tools/find.ts:20-26
const findSchema = Type.Object({
  pattern: Type.String({
    description: "Glob pattern to match files, " +
      "e.g. '*.ts', '**/*.json', or 'src/**/*.spec.ts'"
  }),
  path: Type.Optional(Type.String({
    description: "Directory to search in (default: cwd)"
  })),
  limit: Type.Optional(Type.Number({
    description: "Maximum number of results (default: 1000)"
  })),
});


// packages/coding-agent/src/core/tools/grep.ts:23-35
const grepSchema = Type.Object({
  pattern: Type.String({
    description: "Search pattern (regex or literal string)"
  }),
  path: Type.Optional(Type.String({
    description: "Directory or file to search (default: cwd)"
  })),
  glob: Type.Optional(Type.String({
    description: "Filter files by glob pattern, e.g. '*.ts'"
  })),
  ignoreCase: Type.Optional(Type.Boolean({
    description: "Case-insensitive search (default: false)"
  })),
  literal: Type.Optional(Type.Boolean({
    description: "Treat pattern as literal string (default: false)"
  })),
  context: Type.Optional(Type.Number({
    description: "Lines before and after each match (default: 0)"
  })),
  limit: Type.Optional(Type.Number({
    description: "Maximum number of matches (default: 100)"
  })),
});
```

find 有 3 个参数，grep 有 7 个 — 这个差异反映了内容搜索比文件名搜索本质上更复杂。grep 需要控制正则/字面量模式、大小写敏感、上下文行数、文件类型过滤，这些在 bash `grep` 命令中是通过 `-i`、`-F`、`-C`、`--include` 等 flags 实现的。结构化 schema 把这些 flags 转化为有描述的命名参数，让 LLM 不需要记忆 flag 字母。

默认限制的设计

```
// packages/coding-agent/src/core/tools/find.ts:30
const DEFAULT_LIMIT = 1000;

// packages/coding-agent/src/core/tools/grep.ts:38
const DEFAULT_LIMIT = 100;
```

find 默认 1000 条，grep 默认 100 条。这个 10 倍的差异有意为之：

- **文件名列表的信息密度低**。每条结果就是一个路径，一千条路径在 LLM context 中占用不大。LLM 经常需要浏览大量文件来理解项目结构。
- **内容搜索的信息密度高**。每条 grep 结果包含文件路径、行号、匹配行内容、可能还有上下文行。100 条 grep 结果已经提供了足够的信息，更多只会浪费 context。

LLM 可以通过 `limit` 参数覆盖默认值，但大多数情况下默认值足够好。这种"合理的默认值"减少了 LLM 需要做的决策数量。

`.gitignore` 集成

两个搜索工具都自动尊重 `.gitignore` 规则。这不是一个 flag — 它是默认行为，不能关闭。

为什么强制启用？因为 LLM 搜索 `node_modules`、`dist`、`.git` 这些目录的结果几乎从来都不是有用的。一个典型的 Node.js 项目，`node_modules` 中的文件数量可能是源码的 100 倍。不排除这些目录，搜索结果会被噪声淹没。

find 工具的实现优先使用 `fd`（如果系统安装了的话），否则回退到 Node.js 的 `globSync`：

```
// packages/coding-agent/src/core/tools/find.ts:41-46
export interface FindOperations {
  exists: (absolutePath: string) =>
    Promise<boolean> | boolean;
  glob: (
    pattern: string,
    cwd: string,
    options: { ignore: string[]; limit: number }
  ) => Promise<string[]> | string[];
}
```

`fd` 是一个 Rust 写的 `find` 替代品，默认尊重 `.gitignore`、速度极快。find 工具通过 `ensureTool("fd", true)`（find.ts:214）确保它可用（必要时自动下载），然后用 `spawn` 流式调用。调用参数里有两个值得注意的细节（find.ts:228-246）：

- **`--no-require-git`（仅在非 git 目录时加）**：fd 默认在 git 仓库外不读 `.gitignore`；这个 flag 让它即便在非 git 目录也应用**层级 `.gitignore`**。但 find 工具会先探一下当前是否在 git 仓库内（向上找 `.git`，find.ts:232），**只有不在仓库里才加** `--no-require-git`。**在仓库内则用 fd 的默认 git-aware 行为**，这样父仓库的 `.gitignore` 规则会**停在嵌套子仓库的边界**上、不越界（v0.79.10，#5960）——否则一个把某子目录 ignore 掉的父 `.gitignore` 会连带屏蔽掉那个其实是独立 git 仓库的子目录里的文件。
- **`--full-path`**：fd 的 `--glob` 默认只匹配文件名（basename）。当 LLM 给的 pattern 含路径分隔符（如 `src/**/*.spec.ts`）时，find 工具会加上 `--full-path`，让 glob 匹配完整路径而非仅文件名 —— 这样"按目录结构搜文件"才能正确工作。

这是一个"能力增强"的设计 — 核心功能不依赖外部工具，但用上 fd 后性能与 `.gitignore` 行为都更好。

ripgrep 后端

grep 工具的后端是 ripgrep（`rg`）。选择 ripgrep 而非系统 `grep` 的原因：

1. **默认尊重 `.gitignore`**。和 `fd` 一样，不需要额外配置
2. **速度**。ripgrep 使用 Rust 的 regex crate，在大代码库上比 GNU grep 快数倍
3. **Unicode 安全**。正确处理 UTF-8 文件，不会因为二进制文件内容导致乱码输出
4. **单行截断**。ripgrep 可以限制匹配行的最大长度，避免一行 minified JS 消耗大量 context

具体调用方式上，grep 不再"同步逐文件读取再匹配"，而是以 `--json` 模式**流式**运行 ripgrep：`spawn(rgPath, ["--json", "--line-number", "--color=never", "--hidden", ...])`（grep.ts:215-221），然后逐行 `JSON.parse` ripgrep 输出的结构化事件、边流边收集匹配（grep.ts:270-276）。流式 + 子进程的好处是整个搜索**全程可取消**（`AbortSignal` 一到就 kill 子进程）且不阻塞事件循环 —— 这对在大仓库里跑搜索、又随时可能被用户打断的 agent 很关键。

```
// packages/coding-agent/src/core/tools/truncate.ts:13
export const GREP_MAX_LINE_LENGTH = 500; // Max chars per grep match line
```

每条 grep 匹配行被截断到 500 字符。这个限制处理了一个常见的噪声源 — minified 文件中的匹配。一行 50KB 的 minified JavaScript 如果包含搜索词，不加截断就会消耗大量 context 而不提供有用信息。

结果截断的层次

grep 的截断有三个层次，形成递进的保护：

1. 单行截断（500 字符）— 防止单个匹配行过长 **2. 匹配数限制**（默认 100 条）— 防止匹配结果过多 **3. 总输出截断**（50KB）— 最终的安全网

```
// packages/coding-agent/src/core/tools/grep.ts:40-44
export interface GrepToolDetails {
  truncation?: TruncationResult;
  matchLimitReached?: number;
  linesTruncated?: boolean;
}
```

`matchLimitReached` 记录了是否因为 limit 而停止搜索。当这个字段有值时，返回给 LLM 的结果会附带提示："搜索在第 N 条结果后停止，可能还有更多匹配。如果需要更多，请增加 limit 参数或缩小搜索范围。"

`linesTruncated` 标记是否有匹配行被截断。这让 LLM 知道某些匹配行的内容不完整，如果需要完整行可以用 read 工具去读对应文件。

GrepOperations 的可插拔设计

```
// packages/coding-agent/src/core/tools/grep.ts:50-55
export interface GrepOperations {
  isDirectory: (absolutePath: string) =>
    Promise<boolean> | boolean;
  readFile: (absolutePath: string) =>
    Promise<string> | string;
}
```

grep 的 operations 接口比 find 的更简单 — 只需要判断路径类型和读取文件。这是因为 grep 的核心搜索逻辑（ripgrep 调用）在默认实现中处理，远程场景下可能需要完全不同的搜索策略（比如全文搜索引擎而非逐文件 grep）。

find 和 grep 的 operations 接口都支持同步和异步返回值（`Promise<T> | T`）。这种灵活性让本地实现可以用同步文件系统调用（更快，无 event loop 开销），远程实现可以用异步调用。

TUI 中的搜索结果展示

find 结果在 TUI 中按文件路径显示，每条结果一行，超过 10 条后折叠。grep 结果的展示更复杂 — 包含文件路径（作为分组标题）、行号（高亮显示）、匹配行内容（搜索词高亮）。

两个工具的 TUI 展示都使用了和 read 相同的"默认折叠 + 手动展开"模式。LLM 看到的是完整的截断后结果，用户在 TUI 中看到的是精简预览。

取舍分析

得到了什么

大幅降低搜索出错率。LLM 不需要拼 shell 命令、不需要记住平台差异、不需要手动排除 node_modules。

分层截断保护。从单行截断到匹配数限制到总输出截断，三层保护确保搜索结果永远不会消耗失控的 context。

性能提升。ripgrep 和 fd 在大代码库上比系统工具快数倍，这直接转化为 agent 的响应速度提升。

放弃了什么

多了两个工具增加选择负担。LLM 需要知道"搜文件名用 find、搜内容用 grep、其他用 bash"。system prompt 中的工具使用指引帮助 LLM 做选择，但对于不熟悉 pi 工具集的 LLM（比如较弱的模型），额外的工具可能导致混淆。

强制 .gitignore 过滤可能遗漏需要的文件。如果 LLM 需要搜索 `dist/` 目录中的构建产物，find 和 grep 都会跳过它。这时 LLM 必须回退到 bash 工具。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。find 和 grep 是较晚从 bash 中分离出来的工具， `Operations` pluggable 设计与三层截断保护保持不变。 主要实现级演进：① find **尊重嵌套 git 仓库边界**（v0.79.10，#5960）—— 仓库内用 fd 默认 git-aware 行为，父 `.gitignore` 规则停在子仓库边界；仅在非 git 目录才加 `--no-require-git`；② grep 改为以 ripgrep `--json` **流式**运行（全程可取消、不阻塞，取代同步逐文件读取）；③ find 用 fd 并加 `--full-path` （路径型 glob，如 `src/**/*.spec.ts` ）； 另外 grep/find 对以 `-` 开头的 flag-like pattern 用 `--` 分隔做了注入修复（v0.71.0）。 `GREP_MAX_LINE_LENGTH` 的 500 字符限制根据 minified 文件噪声问题调整。

第 24 章：pi-tui — 在终端里做应用

> **定位**：本章解析 pi 为什么自建 TUI 框架，以及极简 Component 接口如何撑起复杂交互。 前置依赖：第 10 章（Agent 事件订阅）。 适用场景：当你想理解终端 UI 的渲染模型，或者想为 pi 的 TUI 添加组件。

为什么不用 Ink.js?

Node.js 生态有成熟的终端 UI 框架（Ink.js 基于 React 的声明式模型）。pi 选择自建，原因是 **差分渲染的完全控制权**。

Ink.js 使用 React 的 reconciliation 算法来管理组件树，然后把组件树渲染成终端输出。问题在于：终端不是浏览器 DOM — 你不能"修改一个元素"，你只能覆盖某一行的内容。Ink.js 的抽象层让你无法精确控制"哪些行需要重绘"，导致不必要的全屏刷新。

pi-tui 选择了一个更底层的模型：组件返回字符串数组，TUI 逐行比较新旧输出，只重绘变化的行。

Component 接口：一个方法定义一切

```
// packages/tui/src/tui.ts:17-41
export interface Component {
  render(width: number): string[];
  handleInput?(data: string): void;
  wantsKeyRelease?: boolean;
  invalidate(): void;
}
```

`render` 返回字符串数组（每个元素是一行）。就是这么简单 — 没有 virtual DOM，没有 JSX，没有 state management。一个组件的全部职责就是：给你一个宽度，你告诉我你要显示什么。

`handleInput` 是可选的 — 只有能接收键盘输入的组件（比如 Editor、SelectList）才需要实现它。参数 `data` 是原始的终端输入序列（可能是单个字符如 `"a"`，也可能是转义序列如 `"\x1b[A"` 表示方向上键）。

`wantsKeyRelease` 控制组件是否接收按键释放事件。这需要 Kitty keyboard protocol 支持（普通终端不区分 keydown 和 keyup）。默认 `false` — 释放事件被 TUI 过滤掉，减少组件需要处理的事件量。

`invalidate()` 通知 TUI 组件需要重绘。调用后 TUI 会在下一个 render cycle 重新调用 `render()`。组件的缓存状态（如果有的话）也应该在 `invalidate()` 中清除。

Focusable 接口与双层光标模型

```
// packages/tui/src/tui.ts:52-68
export interface Focusable {
  focused: boolean;
}

export const CURSOR_MARKER = "\x1b_pi:c\x07";
```

`Focusable` 是 `Component` 的增强接口。当一个组件获得焦点时，TUI 设置它的 `focused` 属性为 `true`。

这里容易产生一个误解：以为终端里只有终端硬件光标这一个光标。pi 实际上用了**两层光标**，各司其职。

第一层：反色软件光标（用户看到的那个）。 编辑器并不依赖硬件光标来"显示"光标位置 —— 它在 `render()` 输出里，把光标所在的那个 grapheme 用反色（SGR `\x1b[7m` … `\x1b[0m`）画出来；光标停在行尾时则反色一个空格（`packages/tui/src/components/editor.ts:549-564`）。这个反色块才是用户视觉上看到的光标，它随差分渲染一起刷新，不受终端硬件光标能力差异的影响。

第二层：硬件光标（IME 定位叠加层）。 组件在反色块之前，还会在同一位置输出一个零宽的 `CURSOR_MARKER`（`editor.ts:549-550`）。`CURSOR_MARKER` 是一个 APC（Application Program Command）转义序列 —— 终端会忽略它，但 TUI 渲染后能找到它的位置，把真正的硬件光标移过去。硬件光标在这里几乎不承担"显示"职责，它只是一个**给 IME 用的定位叠加层**：中文、日文、韩文输入法需要读硬件光标位置来放置候选窗口，没有它候选框会飘到错误的地方。

两层分工带来一个必须小心的收尾：**退出时要先清软件光标、再交还硬件光标**。`stop()` 会先写一个普通空格覆盖掉那个反色块（否则退出后终端里会残留一个反色残影），把光标移到内容末尾，最后才 `showCursor()` 恢复正常硬件光标（`tui.ts:698-712`）。v0.81.0 的清屏修复（#6790）正是暴露并补上了这个顺序 —— 在此之前退出后会留下一块反色光标残影。

TUI 类：渲染引擎

```
// packages/tui/src/tui.ts:214-245
export class TUI extends Container {
  public terminal: Terminal;
  private previousLines: string[] = [];
  private previousWidth = 0;
  private previousHeight = 0;
  private focusedComponent: Component | null = null;
  private renderRequested = false;
  private renderTimer: NodeJS.Timeout | undefined;
  private lastRenderAt = 0;
  private static readonly MIN_RENDER_INTERVAL_MS = 16;
  private cursorRow = 0;
  private hardwareCursorRow = 0;
  private maxLinesRendered = 0;
  // Overlay stack for modal components
  private overlayStack: {
    component: Component;
    options?: OverlayOptions;
    preFocus: Component | null;
    hidden: boolean;
    focusOrder: number;
  }[] = [];
}
```

TUI 自身继承了 `Container`（组件树的容器），同时管理渲染状态。几个关键的状态字段：

- `previousLines`：上一次渲染的输出，用于差分比较
- `MIN_RENDER_INTERVAL_MS = 16`：渲染节流，约 60fps 上限，防止高频 invalidate 导致 CPU 空转
- `maxLinesRendered`：终端工作区域的最大高度，用于检测内容收缩时是否需要清理空行

渲染调度：requestRender

```
// packages/tui/src/tui.ts:469-516
requestRender(force = false): void {
  if (force) {
    this.previousLines = [];
    this.previousWidth = -1;
    this.previousHeight = -1;
    // ...重置所有状态...
    process.nextTick(() => {
      this.doRender();
    });
    return;
  }
  if (this.renderRequested) return;
  this.renderRequested = true;
  process.nextTick(() => this.scheduleRender());
}

private scheduleRender(): void {
  const elapsed = performance.now() - this.lastRenderAt;
  const delay = Math.max(0, MIN_RENDER_INTERVAL_MS - elapsed);
  this.renderTimer = setTimeout(() => {
    this.doRender();
  }, delay);
}
```

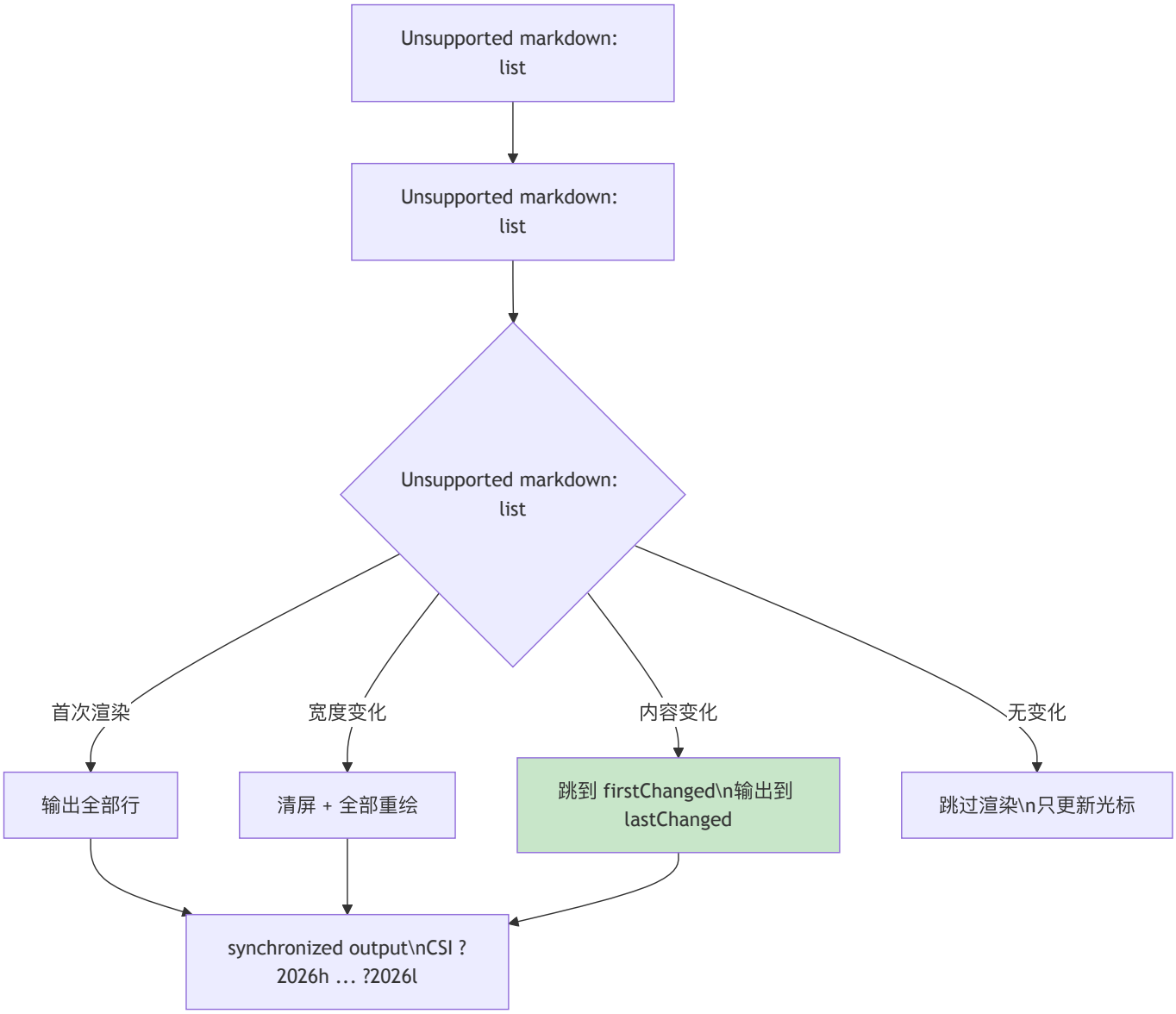
`requestRender` 有两种模式：

- **`force = true`**：清空所有缓存，下一个 tick 立即全量渲染。用于主题切换、终端 reset 等场景。
- **`force = false`**（默认）：标记"需要渲染"，通过 `scheduleRender` 节流到至少 16ms 间隔。多次快速的 `requestRender` 调用只会触发一次实际渲染。

`process.nextTick` 确保渲染在当前事件循环结束后执行 — 这让同一个 tick 中的多个状态变更可以合并为一次渲染。

差分渲染算法

`doRender()` 是 TUI 的核心。它的逻辑可以分为四个阶段：



差分阶段的核心代码：

```
// packages/tui/src/tui.ts:981-1011
// Find first and last changed lines
let firstChanged = -1;
let lastChanged = -1;
const maxLines = Math.max(
  newLines.length, this.previousLines.length
);
for (let i = 0; i < maxLines; i++) {
  const oldLine = this.previousLines[i] ?? "";
  const newLine = newLines[i] ?? "";
  if (oldLine !== newLine) {
    if (firstChanged === -1) firstChanged = i;
    lastChanged = i;
  }
}
```

算法的关键洞察：只需要找到**第一个**和**最后一个**变化行。然后把光标移到第一个变化行，从那里开始输出到最后一个变化行。不需要逐行比较和逐行更新 — 因为终端的 cursor movement 本身也有开销，连续输出比跳跃输出更快。

特殊情况处理：

- **宽度变化**：必须全量重绘，因为换行位置全部改变
- **高度变化**：全量重绘（Termux 例外 — 软键盘弹出会频繁改变高度）
- **内容收缩**：可选地清理空行（`clearOnShrink`），避免在长输出消失后留下视觉残留

Synchronized Output

每次渲染都包裹在 synchronized output 序列中：

```
// packages/tui/src/tui.ts:917-923
let buffer = "\x1b[?2026h"; // Begin synchronized output
// ...写入所有行...
buffer += "\x1b[?2026l"; // End synchronized output
this.terminal.write(buffer);
```

CSI `?2026h` 告诉终端"开始缓冲"，`?2026l` 告诉终端"一次性显示"。没有这个序列，逐行输出会导致可见的闪烁 — 用户能看到旧内容被逐行替换为新内容的过程。synchronized output 让更新在视觉上是原子的。

注意：不是所有终端都支持 CSI 2026。不支持的终端会忽略这些序列，退化为逐行更新。这种优雅降级是 pi-tui 的设计哲学之一 — 利用先进终端的能力，但不依赖它们。

Overlay 系统

```
// packages/tui/src/tui.ts:119-155
export interface OverlayOptions {
  width?: SizeValue;
  minWidth?: number;
  maxHeight?: SizeValue;
  anchor?: OverlayAnchor;
  offsetX?: number;
  offsetY?: number;
  row?: SizeValue;
  col?: SizeValue;
  margin?: OverlayMargin | number;
  visible?: (termWidth: number, termHeight: number) => boolean;
  nonCapturing?: boolean;
}
```

Overlay 是渲染在主内容之上的浮动组件。典型用途：自动补全菜单、模型选择器、键绑定帮助。

Overlay 的定位支持三种模式：

1. **锚点模式**（`anchor`）：9 个预定义位置（center、top-left、bottom-right 等）
2. **百分比模式**（`row: "25%"`）：相对终端大小定位
3. **绝对模式**（`row: 5, col: 10`）：固定位置

`visible` 回调让 overlay 可以根据终端尺寸动态显示/隐藏 — 比如在终端宽度小于 60 列时隐藏侧边栏 overlay。

Overlay 有独立的焦点管理。`showOverlay` 返回一个 `OverlayHandle`，可以控制显示/隐藏、焦点获取/释放。焦点释放时自动恢复到之前的焦点组件（`preFocus`）。

Overlay 的渲染发生在 `compositeOverlays` 阶段 — 先渲染主内容，再把 overlay 的输出合成到对应位置。差分比较在合成之后进行，所以 overlay 的变化也受益于差分渲染。

Container：组件树的基础

```
// packages/tui/src/tui.ts:256-289
export class Container implements Component {
  children: Component[] = [];

  addChild(component: Component): void {
    this.children.push(component);
  }

  render(width: number): string[] {
    const lines: string[] = [];
    for (const child of this.children) {
      const childLines = child.render(width);
      for (const line of childLines) {
        lines.push(line);
      }
    }
    return lines;
  }
}
```

Container 是组件树的容器节点。它的 `render` 简单地拼接所有子组件的输出。TUI 自身继承 Container — 整个 UI 就是一棵组件树，TUI 是根节点。

注意内层这个看似啰嗦的 `for...push(line)` 循环 — 它曾经是更简洁的 `lines.push(...child.render(width))`。区别在于 spread 形式会把整个子数组作为函数参数展开，当一个长会话累积出成千上万行、子组件层层嵌套时，`push(...hugeArray)` 会触碰 V8 的参数数量上限，抛出 `RangeError: Maximum call stack size exceeded`（#2651，v0.67.0 修复）。逐行 push 没有这个隐患。这是"终端 UI 看似简单、实则处处是规模边界"的一个缩影。

这个设计的简洁性值得注意：没有 layout engine，没有 flex/grid，没有 padding/margin。所有布局都由组件自己在 `render()` 中通过字符串拼接实现。这看起来原始，但对终端 UI 来说足够了 — 终端的布局模型本质上就是"一行一行堆叠"。

终端能力检测：探测，而不是假设

"利用先进终端能力但不依赖它们"这条哲学，落地为一套**主动探测**机制 —— TUI 不假设终端支持什么，而是去问。

最典型的是**亮/暗配色检测**。pi 的自动主题需要知道终端背景是亮是暗，但终端不会主动上报。TUI 用 OSC 11 序列查询背景色：

```
// packages/tui/src/tui.ts:1693
queryTerminalColorScheme(
  { timeoutMs }: { timeoutMs: number }
): Promise<TerminalColorScheme | undefined>
```

它向终端写入 OSC 11 查询，终端（如果支持）回写一个 `\x1b]11;rgb:....` 响应，`parseOsc11BackgroundColor` 把它解析成 RGB，再换算成 `"dark" | "light"`（`packages/tui/src/terminal-colors.ts:7,35,67`）。`timeoutMs` 是关键 —— 不支持 OSC 11 的终端永远不回应，所以查询必须带超时，超时即降级为"未知"。

配色还可能在运行中变化（用户切换系统亮暗模式）。TUI 提供订阅接口让上层响应这种变化：

```
// packages/tui/src/tui.ts:660-668
onTerminalColorSchemeChange(
  listener: (scheme: TerminalColorScheme) => void
): () => void
setTerminalColorSchemeNotifications(enabled: boolean): void
```

这套"查询—解析—超时降级—订阅变化"的模式，正是 pi-tui 处理一切高级终端能力的范式。

Kitty 图片协议：在终端里画图

第 24 章前面提到的 Kitty **keyboard** protocol（区分 keydown/keyup）只是 Kitty 系列协议之一。另一条独立的能力是 Kitty **graphics**（图片）协议 —— 让终端直接渲染位图，pi 用它在对话里内联显示图片附件。

`detectCapabilities()` 返回每个终端支持哪种图片协议：

```ts
// packages/tui/src/terminal-image.ts:3-9,65
export type ImageProtocol = "kitty" | "iterm2" | null;

export interface TerminalCapabilities {
  images: ImageProtocol;
  trueColor: boolean;
  hyperlinks: boolean;
}

export function detectCapabilities(...): TerminalCapabilities
```

检测逻辑是一串终端识别：Kitty、WezTerm、Ghostty 走 `"kitty"`；Warp 也支持 Kitty graphics 协议（#5841，`terminal-image.ts:95-97`）；iTerm2 走 `"iterm2"`；其余返回 `null`（纯文本降级，显示占位符而非图片）。一个重要的保守决策：**tmux 下图片协议不可靠，直接置 `images: null`**（`terminal-image.ts:73-75`）—— 宁可不画，也不画错。

`ImageRenderOptions` 还支持 `imageId`（复用/替换同一张图，避免重复传输）等细节。但设计要点始终一致：能力是探测出来的，不支持就优雅降级到文本。

文本渲染的边界细节

自建渲染的代价，是连"把文本正确地铺到终端上"这种小事都得自己兜底。几个 v0.8x 期间补上的细节值得一提：

- **ANSI 感知换行。** `wrapTextWithAnsi()` 按 `\r\n | \r | \n` 统一识别 CRLF/CR/LF 三种换行，并用一个 `AnsiCodeTracker` 跨行追踪 SGR 状态 —— 换行后仍处于激活状态的样式会被补写到续行开头，避免颜色/加粗在硬换行处意外中断（`packages/tui/src/utils.ts:715-735`，#6764）。
- **Markdown source-preservation 选项族。** `MarkdownOptions` 允许调用方选择"按源码原样保留"而非规范化：除了早先的 `preserveOrderedListMarkers`（保留源列表序号），v0.80.3 新增 `preserveBackslashEscapes`，让被反斜杠转义的标点原样保留、不被渲染器吃掉（`packages/tui/src/components/markdown.ts:101-102`，#6105）。
- **流式代码围栏防闪烁。**流式渲染 Markdown 时，尚未闭合的代码围栏一度会随 token 到达而闪烁、抖动；现在部分闭合的围栏渲染稳定，不再收缩跳变（#5846）。

得到了什么

完全的控制力。差分渲染的粒度、IME 支持（通过 CURSOR_MARKER 定位硬件光标）、Kitty keyboard protocol 支持 — 这些都需要直接操作终端转义序列，框架反而会碍事。

极低的渲染开销。字符串比较 + 只重绘变化行的策略，让 TUI 在高频更新场景（比如 bash 命令的流式输出）下保持流畅。

渲染节流防抖。16ms 的最小渲染间隔和 `requestRender` 的去重机制，确保即使组件频繁触发 invalidate，CPU 消耗也是可控的。

放弃了什么

更多的维护成本。从字符宽度计算到 ANSI 转义序列解析，都要自己实现。`visibleWidth()`、`truncateToWidth()`、`wrapTextWithAnsi()` 这些工具函数证明了"终端里的文本处理"远比想象复杂。

没有声明式 API。相比 React/Ink.js 的声明式模型，命令式的 `render()` 方法需要组件自己管理所有状态和渲染逻辑。但 Component 接口的简洁性在某种程度上弥补了这一点 — 实现一个新组件只需要一个 `render(width): string[]` 方法。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 v0.82.1 核实。pi-tui 仍是 pi-mono 中最稳定的包之一： Component 接口自创建以来没有改变，新功能通过添加新组件实现。 Overlay 系统是后来添加的 — 早期版本的模态交互（如模型选择）直接替换主内容。
>
> v0.66 → v0.79 的主要变化： `Container.render` 因长会话栈溢出（#2651，v0.67.0）从 spread-push 改为逐行 push； 新增亮/暗配色探测（OSC 11，`terminal-colors.ts`）与运行时订阅；新增 Kitty 图片（graphics）协议及降级（`terminal-image.ts`， Warp 支持见 #5841）； 终端能力检测扩展到 OSC 8 超链接、tmux 透传、Windows Terminal/JetBrains 等。运行环境最低 Node 版本提升到 **22.19.0**（v0.75.0，Breaking）。
>
> v0.79 → v0.82 的补充：厘清了**双层光标模型** —— 用户看到的是编辑器渲染的反色软件光标，硬件光标只是 IME 定位叠加层，退出时先清软件光标再恢复硬件光标（#6790）； `wrapTextWithAnsi` 归一化 CRLF/CR 并跨行保留 ANSI 样式

> (#6764)；Markdown 渲染新增 `preserveBackslashEscapes` 源码保留选项 (#6105)、流式代码围栏不再闪烁（#5846）。

第 25 章：编辑器组件 — 交互复杂度的集中地

> **定位**：本章解析 pi 的编辑器组件为什么代码量超过很多独立项目。 前置依赖：第 24 章（pi-tui 框架）。 适用场景：当你想理解终端交互的真实复杂度。

为什么编辑器这么复杂?

pi 的 Editor 组件处理的不只是"输入文字"：

- **多行编辑**： `Shift+Enter` 或 `Ctrl+J` 换行（默认键位见 `keybindings.ts:118`，具体哪个可用取决于终端能力）
- **`@` 文件引用**：输入 `@` 触发模糊文件搜索，选中后附加为上下文
- **`!command` 执行**：输入 `!` 前缀运行 bash，输出送入 LLM
- **`Tab` 路径补全**：自动补全文件路径和斜杠命令
- **图片粘贴**： `Ctrl+V` 检测剪贴板中的图片
- **滚动与光标**：长输入的垂直滚动和光标管理

每个功能单独看都不复杂。但组合在一起 — 用户在多行编辑中间触发了 `@` 搜索，搜索结果弹出覆盖层，用户按 Tab 选中文件，覆盖层关闭，光标回到编辑位置 — 这些状态转换的组合爆炸是编辑器代码量大的根本原因。

Editor 类结构

```
// packages/tui/src/components/editor.ts:217
export class Editor implements Component, Focusable {
  // ... 核心编辑状态 ...
  private autocompleteProvider?: AutocompleteProvider;
  private autocompleteList?: SelectList;
  private autocompleteState: "regular" | "force" | null = null;
  private autocompletePrefix: string = "";
  private autocompleteMaxVisible: number = 5;
  private autocompleteAbort?: AbortController;
  private autocompleteDebounceTimer?: ReturnType<typeof setTimeout>;
  private autocompleteRequestTask: Promise<void> =
    Promise.resolve();
  private autocompleteStartToken: number = 0;
  private autocompleteRequestId: number = 0;

  // Paste tracking for large pastes
  private pastes: Map<number, string> = new Map();
  private pasteCounter: number = 0;
  private pasteBuffer: string = "";
  // ...
}
```

光是自动补全相关的字段就有 10 个。这不是过度设计 — 每个字段都解决一个真实的并发问题：

- `autocompleteAbort`：取消正在进行的补全请求（用户继续输入时需要取消旧请求）
- `autocompleteDebounceTimer`：防抖，用户快速输入时不触发每一个字符的补全
- `autocompleteRequestId`：请求序号，确保过时的异步响应不会覆盖新的结果
- `autocompleteStartToken`：记录补全开始时的光标位置，用于在完成补全时正确替换文本

自动补全系统

Editor 的自动补全通过 `AutocompleteProvider` 接口和外部系统对接：

```ts
// packages/tui/src/autocomplete.ts:243-269
export interface AutocompleteProvider {
  /** 在 token 边界自然触发本 provider 的字符。 */
  triggerCharacters?: string[];

  getSuggestions(
    lines: string[],
    cursorLine: number,
    cursorCol: number,
    options: { signal: AbortSignal; force?: boolean },
  ): Promise<AutocompleteSuggestions | null>;

  applyCompletion(/* ...替换文本并返回新光标... */): {
    lines: string[];
    cursorLine: number;
    cursorCol: number;
  };

  /** 显式 Tab 补全时是否应触发文件补全。 */
  shouldTriggerFileCompletion?(
    lines: string[], cursorLine: number, cursorCol: number
  ): boolean;
}
```

这个接口比早期版本扩展了不少，每个新增字段都对应一个真实的交互需求：

- `getSuggestions` 的入参从早期的 `(text, cursorPosition, signal)` 改成了 `(lines, cursorLine, cursorCol, options)` —— 多行编辑下"光标位置"必须用行号+列号表达，单一偏移量不够；`options.force` 取代了外部的模式标志。
- `triggerCharacters`（v0.79.1，#4703）声明哪些字符在 token 边界自然触发本 provider，比如 `#`、`$`。Editor 据此决定何时不等 Tab 就主动唤起补全。
- `shouldTriggerFileCompletion`（v0.69.0）让 provider 自己判断"此刻按 Tab 是否该做文件路径补全"，把这个上下文敏感的决策下放给 provider，而不是写死在 Editor 里。

与之配套，`SlashCommand` 接口也新增了 `argumentHint?`（v0.67.6，`autocomplete.ts:230`）—— 斜杠命令可以声明参数提示（如 `/model <name>`），补全列表渲染时显示出来。

补全触发有两种模式：

- **`"regular"`**：用户输入时自动触发（带 debounce），比如输入 `/` 后提示斜杠命令
- **`"force"`**：用户按 Tab 明确请求补全

补全请求是异步的 — 文件路径补全可能需要扫描文件系统，斜杠命令补全可能需要查询已注册的命令列表。异步意味着竞态条件：用户发起补全请求后继续输入，旧请求的结果到达时上下文已经变化。

`autocompleteRequestId` 解决这个问题：每次发起请求时递增 ID，响应到达时检查 ID 是否匹配当前请求。不匹配的响应被静默丢弃。这比 debounce 更精确 — debounce 只能延迟请求，但不能处理"请求已发出、响应延迟到达"的情况。

补全列表使用 `SelectList` 组件渲染，通过 overlay 系统（第 24 章）定位在光标下方。最多显示 5 项（`autocompleteMaxVisible`），可以通过配置调整：

```
// packages/tui/src/components/editor.ts:287-288
const maxVisible = options.autocompleteMaxVisible ?? 5;
this.autocompleteMaxVisible = Number.isFinite(maxVisible)
  ? Math.max(3, Math.min(20, Math.floor(maxVisible)))
  : 5;
```

@ 文件引用

输入 `@` 触发文件搜索覆盖层。这和普通的自动补全不同 — 它弹出一个独立的搜索界面，支持模糊匹配，选中后在消息中插入文件引用标记。

工作流程：

1. 用户输入 `@`，Editor 检测到这是文件引用触发字符
2. 弹出文件搜索 overlay（使用 find 工具的后端逻辑扫描项目文件）
3. 用户继续输入缩小搜索范围，搜索结果实时更新
4. 用户按 Enter 选中文件，overlay 关闭
5. 文件路径作为上下文附加到消息中（不是插入文本 — 而是作为 context attachment）

这个流程的关键设计是：文件引用不是文本替换，而是结构化数据。LLM 收到的不是 `@src/foo.ts` 这样的字符串，而是一个包含文件路径的 context 对象。这让后端可以把文件内容读出来作为附加上下文发送给 LLM。

！Bash 执行

以 `！` 开头的消息触发 bash 执行模式：

- `!npm test` — 执行命令，输出作为用户消息的一部分发送给 LLM
- `!!` — 重复上一条 bash 命令

这个功能把编辑器变成了一个混合输入界面 — 既是聊天输入框，也是命令行。用户不需要切换到单独的终端窗口就能运行命令并把结果分享给 LLM。

粘贴处理

粘贴在终端中比在浏览器中复杂得多。Editor 需要处理多种场景：

Bracketed Paste Mode

现代终端支持 bracketed paste — 在粘贴内容前后加入标记序列（`\x1b[200~` 和 `\x1b[201~`），让应用区分"用户输入"和"粘贴内容"。Editor 检测到 bracketed paste 开始标记后，把后续输入缓存到 `pasteBuffer` 中，直到结束标记。

```
// packages/tui/src/components/editor.ts:252-257
// Paste tracking for large pastes
private pastes: Map<number, string> = new Map();
private pasteCounter: number = 0;

// Bracketed paste mode buffering
private pasteBuffer: string = "";
```

大文本粘贴折叠

用户可能粘贴几百行的日志或代码。如果直接显示在编辑器中，会让输入区域变得巨大。Editor 的解决方案是 **paste marker**：

```
// packages/tui/src/components/editor.ts:12-16
const PASTE_MARKER_REGEX =
  /\[paste #(\d+)( (\+\d+ lines|\d+ chars))?\]/g;
const PASTE_MARKER_SINGLE =
  /^\[paste #(\d+)( (\+\d+ lines|\d+ chars))?\]$/;
```

大粘贴被替换为一个标记如 `[paste #1 +123 lines]`，原始内容保存在 `pastes` Map 中。这个标记在编辑器中作为一个原子单元处理 — 光标移动跳过它，删除时整体删除。

`segmentWithMarkers` 函数包装了 `Intl.Segmenter`，让 paste marker 在 Unicode 分词层面也表现为单个 segment：

```
// packages/tui/src/components/editor.ts:30-45
function segmentWithMarkers(
  text: string, validIds: Set<number>
): Iterable<Intl.SegmentData> {
  if (validIds.size === 0 || !text.includes("[paste #")) {
    return baseSegmenter.segment(text);
  }
  // Find all marker spans with valid IDs
  // Merge graphemes within markers into single segments
  // ...
}
```

这个实现的精巧之处：paste marker 的存在对 word wrap、光标移动、删除操作都是透明的 — 因为它们在分词层就被处理成了原子单元。

paste registry 不是一个"设了就不管"的表 —— 它必须和文本严格对账。删除一个 paste marker（比如退格退掉它）时，Editor 不只是从文本里抹掉标记，还会把 `pastes` Map 里对应条目删掉，并把更大 id 的条目按升序整体下移一位，让 `[paste #3]` 在 `[paste #1]` 被删后变成 `[paste #2]`（`editor.ts:1296-1306`，v0.80.4 #6397 修复了删除/清屏后残留陈旧记账的问题）；提交或清屏后整个 registry 连同计数器一起清零（`editor.ts:1265-1266`）。这套记账还有一条不易察觉的支线 —— 它和 undo 是绑定的，见下面的 Undo Stack。

IME 支持

IME（Input Method Editor）是中日韩文输入的基础设施。终端中的 IME 支持比 GUI 应用困难得多：

1. **光标定位**。IME 需要硬件光标来定位候选窗口：它读硬件光标位置弹出拼音/候选框，而用户在编辑器里看到的可见光标其实是 `render()` 反色画出的软件块。这套双层光标的完整机制（软件反色块 + 硬件光标叠加、退出清理）见第 24 章「双层光标模型」，这里只强调 IME 依赖的是硬件光标那一层（#6790）。

2. **compose 事件**。IME 的输入过程是多步的 — 用户按下拼音字母，IME 在 compose 状态下显示拼音，用户选择候选字后 IME 发送最终字符。Editor 需要正确处理这个 compose 过程，不能把中间状态当作最终输入。

3. **Kitty keyboard protocol**。Kitty 终端的增强键盘协议可以区分 keydown 和 keyup，提供更精确的键盘事件。但 IME 在这个协议下的行为和标准模式不同，需要特殊处理。

Word Wrap 算法

Editor 的 word wrap 不是简单的"按宽度截断"。它需要考虑：

- **Unicode 字符宽度**：CJK 字符宽 2 列，拉丁字符宽 1 列，emoji 可能宽 2 列
- **换行点选择**：优先在空格处换行，其次在标点处换行，最后才强制在 **grapheme 边界**换行
- **Paste marker**：作为原子单元不能被换行拆开（除非宽度超过整行）

这里的"字符边界"在 v0.79.x 被收紧为 **grapheme（字素簇）边界**（#5495 等）。原因是：强制换行如果落在一个多码位 grapheme（如带变音符的字母、emoji ZWJ 序列）的中间，会把一个视觉字符劈成两半，渲染错乱。Editor 用 `getGraphemeSegmenter()`（包装 `Intl.Segmenter`， `editor.ts:9,18` ）先把行切成 grapheme，换行只在 grapheme 之间发生。这一改动同时修复了旧版本 CJK 文本换行时右侧出现的大段空隙问题。Unicode 词边界的导航与删除（ `Ctrl+←/→` 、按词删除）则由 `word-navigation.ts` 的 `findWordBackward` / `findWordForward` 基于 `Intl.Segmenter` 的 word 分词实现。

```
// packages/tui/src/components/editor.ts:101-108
export function wordWrapLine(
  line: string,
  maxWidth: number,
  preSegmented?: Intl.SegmentData[]
): TextChunk[] {
  if (!line || maxWidth <= 0) {
    return [{ text: "", startIndex: 0, endIndex: 0 }];
  }
  const lineWidth = visibleWidth(line);
  if (lineWidth <= maxWidth) {
    return [{ text: line, startIndex: 0, endIndex: line.length }];
  }
  // ...complex wrapping logic...
}
```

`TextChunk` 不只是文本片段 — 它还记录了在原始行中的起始和结束位置（`startIndex` 、 `endIndex` ）。这让光标在 wrapped 行之间移动时可以正确映射回原始文本位置。

滚动指示器的宽度安全

输入内容超过可视高度时，Editor 在被裁掉的方向画一条滚动指示器边框，形如 `── ↑ 12 more ──────` （`createScrollBorder` ， `editor.ts:259-267` ）。正常情况下它把提示文字左对齐、右侧用 `─` 补满整行宽度。

问题出在**窄终端**：当终端宽度比提示文字本身还窄时，"补满剩余宽度"的 `remaining` 会变成负数，早期版本据此 `"─".repeat(负数)` 会抛异常、把整个渲染带崩（v0.82.0 #7015 修复）。现在的处理是宽度安全的 —— `remaining < 0` 时改走截断分支：用 `sliceByColumn` 按**列宽**（而不是字符数）把指示器截到放得下的宽度，再接一个 `...` 省略号（ `editor.ts:265-267` ）。按列宽截断这点很重要，因为指示器里可能混有宽度为 2 的字符，按字符数截断仍会溢出。

Kill Ring

Editor 实现了 Emacs 风格的 kill ring — `Ctrl+K` 杀掉光标到行尾的内容， `Ctrl+Y` 粘贴最近 kill 的内容。这不是剪贴板 — 它是一个独立的循环缓冲区，保存了多次 kill 的历史。

为什么要实现 kill ring 而不只是用系统剪贴板？因为在终端中访问系统剪贴板需要 OSC 52 序列支持（不是所有终端都支持），而且 kill ring 和 Emacs 快捷键是终端用户的肌肉记忆。

Undo Stack

编辑器维护了自己的 undo 栈（`UndoStack`），支持 `Ctrl+Z` 撤销和 `Ctrl+Shift+Z` / `Ctrl+Y` 重做。这在终端编辑器中不是标配 — 大多数终端输入框不支持撤销。但对于多行编辑场景（用户可能编辑一段代码片段作为 prompt），撤销功能从"可选"变成了"必要"。

关键的一点是：undo 快照不只存文本。`EditorSnapshot` 里同时装了编辑状态、`pastes` registry 和 `pasteCounter`（`editor.ts:215-220`）—— 每次改动前 `pushUndoSnapshot` 把三者一起压栈，`undo()` 弹栈时把文本和 paste registry 一并还原（`editor.ts:2012-2028`，v0.81.0 #6844）。这正是上一节粘贴折叠留下的那条支线：如果 undo 只回滚文本、不回滚 registry，一次"撤销删除 paste marker"就会让标记回到文本里、但它指向的原始内容已经从 registry 里消失，提交时 `[paste #1]` 会展开成空。把 paste registry 纳入 undo 快照，才让"粘贴折叠"和"撤销"这两个看似独立的机制真正自洽。

状态转换的复杂性

编辑器在任意时刻可能处于以下状态之一：

1. **普通编辑**：正常文本输入
2. **自动补全**：补全列表可见，上下箭头选择，Enter 确认，Esc 取消
3. **文件搜索**：`@` 触发的搜索 overlay
4. **Compose**：IME 正在组合输入

这些状态之间的转换不是线性的。用户可以在 compose 状态下按 `@`，需要先完成 compose 再进入文件搜索；在自动补全时按 Esc，需要关闭补全但不退出编辑。每种转换都需要正确处理焦点、光标位置、缓冲区内容。

这就是为什么 Editor 是 pi-tui 中代码量最大的组件 — 不是因为单个功能复杂，而是因为功能组合的状态空间是乘法增长的。

取舍分析

得到了什么

流畅的交互体验。用户不需要离开编辑器就能引用文件、执行命令、查看补全。这些快捷方式让 pi 感觉像一个 IDE 而不是一个聊天框。

大粘贴的优雅处理。Paste marker 机制让用户可以粘贴大段内容而不破坏编辑器的可用性。

放弃了什么

大量的边界条件。每种输入方式（键盘、粘贴、IME、Kitty protocol）和每种交互模式（普通编辑、搜索覆盖层、补全菜单）的组合都需要测试。

维护负担集中。Editor 几乎每个版本都有交互改进或 bug 修复，它是 pi-tui 中变化最频繁的代码。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 v0.82.1 核实。Editor 组件是 pi-tui 中变化最频繁的组件 — 几乎每个版本都有交互改进或 bug 修复。Paste marker 机制是后来添加的， 早期版本会直接将大粘贴内容全部显示在编辑器中。
>
> v0.66 → v0.79 的主要变化： `AutocompleteProvider` 接口扩展 —— `getSuggestions` 改为多行签名 `(lines, cursorLine, cursorCol, options)`，新增 `triggerCharacters` （v0.79.1，#4703）与 `shouldTriggerFileCompletion` （v0.69.0）， `SlashCommand` 新增 `argumentHint` （v0.67.6）； word wrap 收紧为按 grapheme 边界断行（v0.79.x，#5495），修复 CJK 换行大尾隙；新增基于 `Intl.Segmenter` 的 Unicode 词边界导航/删除（ `word-navigation.ts` ）。
>
> v0.79 → v0.82 的补充：换行默认键位补上 `Ctrl+J` （与 `Shift+Enter` 并列， `keybindings.ts:118` ）； undo 快照纳入 paste registry，撤销可同时还原文本与粘贴表（ `EditorSnapshot` ， v0.81.0 #6844），并补齐 paste marker 删除/清屏后

> 的记账清理（v0.80.4 #6397）；窄终端滚动指示器按列宽截断、不再崩溃（v0.82.0 #7015）；并把 IME 光标定位对齐到第 24 章的双层光标模型（#6790）。

第 26 章：RPC 模式 — pi 作为后端服务

> **定位**：本章解析 pi 的 RPC 模式 — 让 CLI 工具可以被其他进程驱动。 前置依赖：第 10 章（Agent 类）。 适用场景：当你想把 pi 集成到 IDE 插件、Web 前端或自动化管道中。

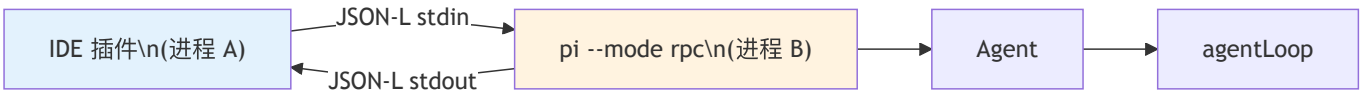
为什么 CLI 工具需要 RPC 模式?

pi 的主要界面是终端 TUI（第 24 章）。但有些场景不需要终端交互：

- **IDE 插件**：VS Code extension 需要通过进程间通信驱动 pi
- **Web 前端**：浏览器客户端需要通过 HTTP 或 WebSocket 驱动 pi
- **自动化管道**：CI/CD 脚本需要非交互式地使用 pi

RPC 模式（`packages/coding-agent/src/modes/rpc/`）提供了一个 JSON-L 协议，让外部进程可以：

1. 发送 prompt 和 steer/follow-up 请求
2. 接收流式事件（和 Agent 事件一一对应）
3. 切换模型、调整 thinking level
4. 管理会话（创建、切换、fork、导出）



RPC 模式和 interactive 模式共享同一个 Agent 实例（第 10 章）。不同的只是事件的消费方式 — interactive 模式渲染到终端，RPC 模式序列化成 JSON-L 输出到 stdout。

协议设计：JSON Lines over stdio

RPC 使用 JSON Lines 协议 — 每条消息是一行 JSON，以换行符分隔。选择 JSON-L over stdio 而非 HTTP/WebSocket 的理由：

1. **零配置**。不需要端口分配、不需要 TLS、不需要服务发现
2. **进程生命周期绑定**。父进程退出时子进程的 stdin 关闭，RPC 会话自然终止
3. **双向通信**。stdin 发命令，stdout 收响应和事件
4. **调试友好**。JSON Lines 可以用 `jq` 解析，管道可以用 `tee` 录制

RPC 命令类型

命令通过 stdin 发送，每条命令是一个 JSON 对象，必须包含 `type` 字段：

```
// packages/coding-agent/src/modes/rpc/rpc-types.ts:19-68
export type RpcCommand =
  // Prompting
  | { id?: string; type: "prompt"; message: string;
      images?: ImageContent[];
      streamingBehavior?: "steer" | "followUp" }
  | { id?: string; type: "steer"; message: string;
      images?: ImageContent[] }
  | { id?: string; type: "follow_up"; message: string;
      images?: ImageContent[] }
  | { id?: string; type: "abort" }
  | { id?: string; type: "new_session";
      parentSession?: string }

  // Model
  | { id?: string; type: "set_model";
      provider: string; modelId: string }
  | { id?: string; type: "cycle_model" }
  | { id?: string; type: "get_available_models" }

  // Thinking
  | { id?: string; type: "set_thinking_level";
      level: ThinkingLevel }

  // Session
  | { id?: string; type: "get_session_stats" }
  | { id?: string; type: "export_html";
      outputPath?: string }
  | { id?: string; type: "fork"; entryId: string }
  | { id?: string; type: "get_messages" }

  // ... 更多命令类型
```

随产品演进，命令清单已远不止上面这些。除了 `prompt` / `steer` / `follow_up` / `abort` / `set_model` / `fork` / `export_html` 等早期命令，现在还包括（rpc-mode.ts）：

- **会话**：`switch_session`（切换到已有会话）、`clone`（复制当前活动分支到新文件，与 `fork` 区别见第 11 章）、`get_last_assistant_text`、`get_commands`（列出可用命令）
- **只读会话树（v0.80.3）**：`get_entries`（带 `since` 增量拉取此后追加的 entry）、`get_tree`（返回完整 `SessionTreeNode[]`）—— 让外部宿主也能渲染第 11 章那棵会话树（`rpc-types.ts:64-65`）

- **压缩与重试**：`compact`（可带 `customInstructions`）、`set_auto_compaction`、`set_auto_retry` / `abort_retry`
- **执行控制**：`abort_bash`、`cycle_thinking_level`、`set_steering_mode` / `set_follow_up_mode`
- **思考档位**：`get_available_thinking_levels`（`rpc-types.ts:39`，返回当前模型支持的档位，配套 `RpcClient.getAvailableThinkingLevels()`；`max` / `xhigh` 也在此暴露）

`fork` 与 `clone` 的区别值得强调：`fork` 从某条更早的消息开新分支，`clone` 复制当前整条活动路径——对应第 11 章的 `/fork` 与 `/clone`。

几个设计要点：

`id` 是可选的。客户端可以为每条命令指定一个 correlation ID，响应会带上同一个 ID，让客户端可以匹配请求和响应。对于不需要匹配的场景（比如 fire-and-forget 的 abort），可以省略 ID。

prompt 的三种变体。`prompt` 是标准请求；`steer` 在 agent 思考过程中插入修正；`follow_up` 在 agent 完成后追加问题。这三种操作对应 Agent API 中不同的消息队列，RPC 直接暴露了这个区分。

`streamingBehavior`。`prompt` 命令可以指定 `streamingBehavior: "steer"` 或 `"followUp"`，让客户端控制当 agent 正在处理时新消息应该进入哪个队列。这和 interactive 模式中的"用户在 agent 思考时输入新消息"是同一个语义。

RPC 响应类型

响应通过 stdout 发送，所有响应共享统一的信封格式：

```
// packages/coding-agent/src/modes/rpc/rpc-types.ts:110-204
export type RpcResponse =
  // 成功响应（部分）
  | { id?: string; type: "response"; command: "prompt";
      success: true }
  | { id?: string; type: "response"; command: "get_state";
      success: true; data: RpcSessionState }
  | { id?: string; type: "response"; command: "set_model";
      success: true; data: Model<any> }
  | { id?: string; type: "response"; command: "bash";
      success: true; data: BashResult }
  | { id?: string; type: "response"; command: "get_messages";
      success: true; data: { messages: AgentMessage[] } }

  // 错误响应（任何命令都可能失败）
  | { id?: string; type: "response"; command: string;
      success: false; error: string };
```

响应的 TypeScript 类型是一个 discriminated union — 每种命令有自己的成功响应类型（带不同的 `data` 结构），但所有命令共享同一个错误响应类型。这让客户端可以先检查 `success` 字段，再根据 `command` 字段解析 `data`。

异步命令和同步命令的区别。 `prompt`、`steer`、`follow_up` 的成功响应不包含 data — 因为这些操作是异步的，真正的结果通过后续的事件流（AgentSessionEvent）传递。`get_state`、`get_messages` 是同步查询，结果直接在响应的 `data` 中返回。

`prompt` 的"单一权威响应" + preflight 契约（v0.67.3）。 `prompt` 命令对响应有一个明确约定：它**只回一次** `response`，且这个响应是一次 **preflight 裁决** —— 表示命令被 accepted / queued / handled。一旦命令被 accept，后续即便发生失败，也**只通过事件流**汇报，**不再发第二个 `response`**。换句话说，客户端可以安全地"一条 prompt 命令 ↔ 一个 response"地配对，而不必担心同一命令收到两次终态响应。这避免了早期"先 ack 再 error" 造成的客户端状态机混乱。另外，对**未知命令**的错误响应现在会带上原命令的 `id`（v0.79.7），让客户端能把错误关联回具体请求。

事件流：命令之外的增量信号

异步命令的真正结果走的是事件流（`AgentSessionEvent` 序列化成 JSON-L）。这条流本区间多了两类事件：

- **`bash_execution_update`**（`agent-session.ts:181`，v0.82.0）：直连 RPC 执行 bash 时，命令输出**边产生边流式**吐给客户端（`{ type:`

`"bash_execution_update", id?, delta }`），而不是等命令结束才一次性给结果——让 IDE 能像 TUI 一样实时显示 bash 输出（对应第 22 章的流式 `onData`）。

- **`summarization_retry_*`**（`agent-session.ts:167-181`，v0.81.1）：压缩/分支摘要失败重试时，`summarization_retry_scheduled` / `_attempt_start` / `_finished` 三个事件把重试过程透出给客户端（对应第 12 章的可恢复压缩）。客户端据此能显示"正在重试压缩"而非静默卡住。

会话状态

```
// packages/coding-agent/src/modes/rpc/rpc-types.ts:90-103
export interface RpcSessionState {
  model?: Model<any>;
  thinkingLevel: ThinkingLevel;
  isStreaming: boolean;
  isCompacting: boolean;
  steeringMode: "all" | "one-at-a-time";
  followUpMode: "all" | "one-at-a-time";
  sessionFile?: string;
  sessionId: string;
  sessionName?: string;
  autoCompactionEnabled: boolean;
  messageCount: number;
  pendingMessageCount: number;
}
```

`RpcSessionState` 是 RPC 客户端了解 agent 状态的主要手段。通过 `get_state` 命令获取，包含了 UI 渲染需要的全部信息：当前模型、是否在流式输出、队列中有多少待处理消息。

`steeringMode` 和 `followUpMode` 控制消息队列的行为：`"all"` 表示累积所有消息一起处理，`"one-at-a-time"` 表示逐条处理。这影响了 IDE 插件的 UX 设计 — 如果用户快速发送多条消息，客户端可以选择是让它们排队还是合并。

会话管理

RPC 提供了完整的会话管理能力：

- **`new_session`**：创建新会话，可选地从指定父会话继承上下文
- **`switch_session`**：切换到已有会话
- **`fork`**：从指定消息处分叉对话（类似 git branch）
- **`get_fork_messages`**：获取可用于 fork 的消息列表
- **`export_html`**：导出当前会话为 HTML 文件
- **`set_session_name`**：给会话命名

`fork` 操作是一个高级功能 — 用户可以回到对话中的某个节点，从那里开始新的对话分支。这在 interactive 模式中通过 TUI 交互实现，在 RPC 模式中通过 `fork` 命令 + `entryId` 参数实现。`get_fork_messages` 返回可 fork 的消息列表和它们的 ID。

Extension UI 桥接

RPC 模式需要处理一个特殊问题：extension 的 UI 交互。在 interactive 模式中，extension 通过 UI context 的 `select()`、`confirm()`、`input()`、`notify()`、`setStatus()`、`editor()` 等方法与用户交互。但 RPC 模式没有 TUI — 这些交互需要被序列化为 JSON 请求，发送给 RPC 客户端处理。

```
// packages/coding-agent/src/modes/rpc/rpc-types.ts:211-246
export type RpcExtensionUIRequest =
  | { type: "extension_ui_request"; id: string;
      method: "select"; title: string;
      options: string[]; timeout?: number }
  | { type: "extension_ui_request"; id: string;
      method: "confirm"; title: string;
      message: string; timeout?: number }
  | { type: "extension_ui_request"; id: string;
      method: "input"; title: string;
      placeholder?: string; timeout?: number }
  | { type: "extension_ui_request"; id: string;
      method: "editor"; title: string;
      prefill?: string }
  | { type: "extension_ui_request"; id: string;
      method: "notify"; message: string;
      notifyType?: "info" | "warning" | "error" }
  | { type: "extension_ui_request"; id: string;
      method: "setStatus"; statusKey: string;
      statusText: string | undefined }
  | { type: "extension_ui_request"; id: string;
      method: "setWidget"; widgetKey: string;
      widgetLines: string[] | undefined }
  | { type: "extension_ui_request"; id: string;
      method: "set_editor_text"; text: string }
```

流程是：extension 发起 UI 请求 → RPC 层序列化为 `RpcExtensionUIRequest` 输出到 stdout → 客户端展示 UI → 客户端发送 `RpcExtensionUIResponse` 到 stdin → RPC 层传回 extension。

```
// packages/coding-agent/src/modes/rpc/rpc-types.ts:253-256
export type RpcExtensionUIResponse =
  | { type: "extension_ui_response"; id: string;
      value: string }
  | { type: "extension_ui_response"; id: string;
      confirmed: boolean }
  | { type: "extension_ui_response"; id: string;
      cancelled: true };
```

这是一个完整的 UI 代理模式 — extension 不知道也不关心 UI 是在终端还是在浏览器中渲染的。

RPC 模式的启动

```
// packages/coding-agent/src/modes/rpc/rpc-mode.ts:46-53
export async function runRpcMode(
  runtimeHost: AgentSessionRuntime
): Promise<never> {
  takeOverStdout();
  let session = runtimeHost.session;
  let unsubscribe: (() => void) | undefined;

  const output = (
    obj: RpcResponse | RpcExtensionUIRequest | object
  ) => {
    writeRawStdout(serializeJsonLine(obj));
  };
```

除了 `pi --mode rpc` 这个 CLI 入口，包还导出了一个 **`./rpc-entry`** 子路径入口（v0.80.3）：`import "@earendil-works/pi-coding-agent/rpc-entry"` 会直接以 RPC 模式启动整个进程。这对"想 spawn 一个纯 RPC 子进程、又不想拼 CLI 参数"的宿主很方便 —— 子进程的 entry 就是这一个 import。

`takeOverStdout()` 劫持了 `console.log` 和 `process.stdout.write`，防止其他代码意外向 stdout 写入非 JSON 内容。这是 RPC 模式的核心防御 — stdout 是协议通道，任何非 JSON 输出都会破坏客户端的解析。

`writeRawStdout` 绕过了劫持层直接写入 stdout。只有 RPC 层自己可以往 stdout 写数据。

Pending Extension Requests

```
// packages/coding-agent/src/modes/rpc/rpc-mode.ts:71-74
const pendingExtensionRequests = new Map<
  string,
  { resolve: (value: any) => void;
    reject: (error: Error) => void }
>();
```

Extension UI 请求是异步的 — RPC 层发送请求后需要等待客户端响应。`pendingExtensionRequests` Map 以请求 ID 为 key 保存了 Promise 的 resolve/reject

回调。当客户端的 `extension_ui_response` 到达时，查找对应的 pending request 并 resolve。

这个 Map 也处理了超时和取消 — 如果 extension 设置了 timeout，超时后 pending request 会被 reject；如果客户端发送 `cancelled: true`，request 也会被相应处理。

Agent API 到 RPC 的映射

RPC 命令和 Agent API 的对应关系：

RPC Command	Agent API
`prompt`	`session.prompt()`
`steer`	`session.steer()`
`follow_up`	`session.followUp()`
`abort`	`session.abort()`
`set_model`	`session.setModel()`
`compact`	`session.compact()`
`get_messages`	`session.getMessages()`
`bash`	direct bash execution
`fork`	`session.fork()`

这个映射几乎是一对一的。RPC 层不添加业务逻辑 — 它只做序列化/反序列化和 stdout 保护。这种薄层设计让 RPC 的行为和 interactive 模式完全一致。其中 `bash` 命令也透传了 `excludeFromContext` 选项（v0.76.0）——RPC 客户端可以执行命令、拿到输出、但让输出不进入 LLM 上下文（对应第 22 章的 `!!` 前缀）。

自带 typed client：`RpcClient`

外部进程当然可以自己手写 JSON-L 的收发，但 pi 从 v0.67.67 起**直接从包根导出了一个 typed client** `RpcClient`（连同 `RpcClientOptions`，index.ts:315）。它封装了"spawn 子进程、按行 framing、按 `id` 关联请求与响应、订阅事件流"这些样板，让 Node 调用

方用类型安全的方法调用代替手拼 JSON。子进程退出时，所有 pending 请求会被 reject（v0.76.0），不会让调用方永久挂起。

> **LF framing 陷阱**：RPC 是严格的"一行一条消息"协议，分隔符是 `\n` 。这里有一个**不能用 Node 内置 `readline` 的坑** —— `readline` 会把 Unicode 行分隔符 `U+2028` / `U+2029` 也当成换行，而这些字符完全可能出现在 LLM 生成的内容里，一旦被误当行边界，整条 JSON 就被切碎、解析失败。因此 RPC 的 framing 必须严格只按 `\n` 切分。配套地，往 stdout 写时还要处理背压：写满时重试 / flush，避免大块事件输出被截断（v0.76.0）。如果你要自己实现客户端而不用 `RpcClient` ，这是必须自己复刻的细节。

取舍分析

得到了什么

pi 可以被任何前端驱动。同一个 agent 内核，终端用户通过 TUI 交互，IDE 用户通过 RPC 交互。代码复用，行为一致。

Extension UI 的透明代理。Extension 不需要为不同的 UI 后端写不同的代码 — RPC 层自动桥接 UI 交互。

放弃了什么

增加了一个运行模式的维护成本。RPC 协议需要版本管理、backward compatibility、错误序列化。每次 Agent API 变化，RPC 层都需要同步更新。

stdout 污染是隐蔽的 bug 源。任何第三方库的 `console.log` 都可能破坏 RPC 协议。`takeOverStdout` 是必要的防御，但它也让调试变得更困难 — 你不能用 `console.log` 来 debug RPC 模式。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0，已对照 **v0.82.1**。RPC 的 JSON-L over stdio、薄层映射、Extension UI 桥接保持不变。 主要演进：① 命令清单大幅扩展，新增只读会话树 `get_entries`（带 `since`）/ `get_tree`（v0.80.3， `rpc-types.ts:64-65`）与 `get_available_thinking_levels`（v0.81.0， `:39`），以及 `clone` / `switch_session` / `compact` / `set_auto_compaction` / `set_auto_retry` / `abort_retry` / `abort_bash` / `get_commands` / `cycle_thinking_level` 等；② 事件流新增流式 `bash_execution_update`（v0.82.0）与 `summarization_retry_*`（v0.81.1， `agent-session.ts:167-181`）；③ `./rpc-entry` 包导出直接以 RPC 模式启动进程（v0.80.3）；④ `prompt` 改为单一权威响应 + preflight 契约（v0.67.3），未知命令错误响应带 `id`（v0.79.7）；⑤ 从包根导出 typed `RpcClient`（v0.67.67），子进程退出时 reject pending 请求（v0.76.0）；⑥ `bash` 透传 `excludeFromContext`（v0.76.0）；⑦ 严格 LF framing（不可用 Node `readline`）+ stdout 背压处理。 若要把 pi 当库**嵌入**同一进程（而非 spawn 子进程驱动），见第 26b 章 SDK。

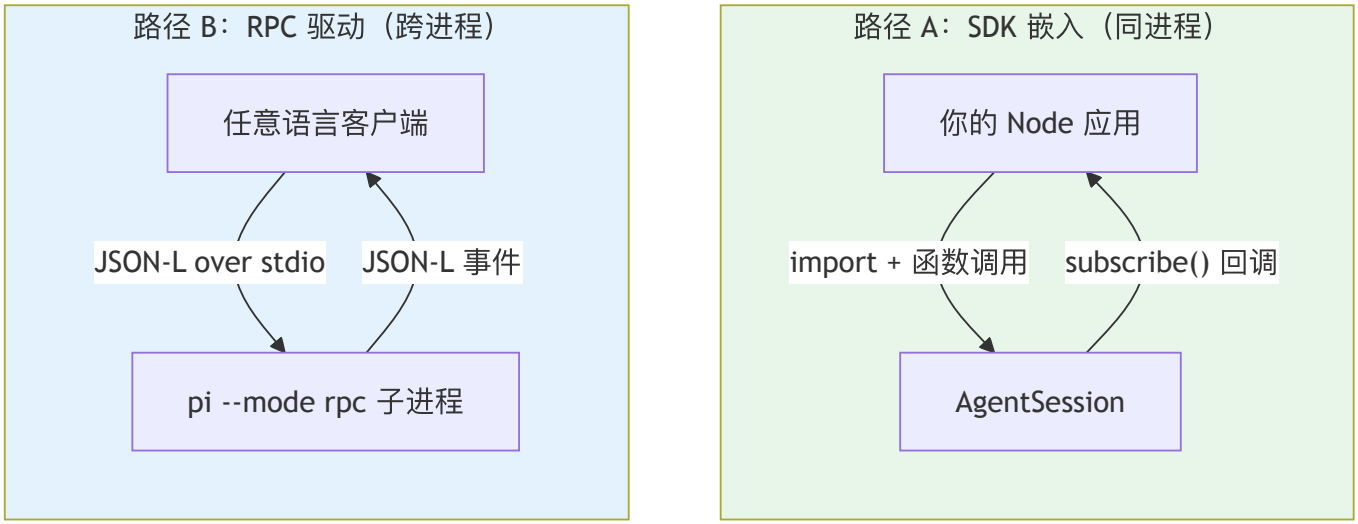
第 26b 章：SDK — 把 pi 当库用

> **定位**：本章解析 pi 的 SDK 嵌入路径 —— 不 spawn 子进程，而是直接把 `AgentSession` 嵌入到你自己的 Node 进程里。 前置依赖：第 10 章（Agent 类）、第 26 章（RPC 模式）。 适用场景：当你想在自己的应用里复用 pi 的 agent 能力（自建 UI、嵌入工作流、构造子 agent），而不是把 pi 当成一个外部命令来驱动。

集成 pi 的两条路：驱动 vs 嵌入

第 26 章讲的 RPC 模式是一条集成路径：把 pi 当成一个**外部进程**，用 JSON-L 协议驱动它。这条路的好处是语言无关、进程隔离 —— 任何能读写 stdio 的客户端都能用。但它也有代价：你需要 spawn 一个子进程、序列化每一次调用、跨进程传递事件流。

pi 还提供了第二条路：**SDK 嵌入**。把 `@earendil-works/pi-coding-agent` 当成一个普通的 npm 库 import 进来，在**同一个 Node 进程内**直接构造并操作 `AgentSession` 。这是 pi"极简核心、能力外置"哲学的自然延伸 —— 既然 TUI、RPC、mom 都只是同一个内核之上的不同宿主（第 24/26 章），那么没有理由不让外部应用也成为这样一个宿主。



值得注意的是，第 26 章的 RPC 文档甚至**主动建议** Node 用户优先走 SDK：如果你本来就是 Node 进程，直接嵌入 `AgentSession` 比 spawn 一个子进程再隔着 stdio 喊话要简单

得多，也省去了 framing、背压、进程生命周期管理这些麻烦。RPC 留给真正需要进程隔离或非 Node 客户端的场景。

为什么这很重要

这条嵌入路径背后是一个产品定位的转变：**pi 不只是一个 CLI 应用，它也是一个可嵌入的 agent 库**。这个定位散落在前面好几章的设计细节里，现在可以串起来看：

- 第 12 章里摘要 system prompt 把"AI **coding** assistant"中性化为"AI assistant"，正是为了让压缩内核能被**非编码**的 SDK 调用方复用。
- 第 14/17/19 章反复出现的"去 process-global、cwd 处处显式传入"，正是为了让同一个进程里能并存多个会话、多个工作目录 —— 这是 SDK（尤其是构造子 agent）的前提。
- 第 15 章工具选择改为 `tools: string[]` 名字白名单，让 SDK 调用方能精确声明启用哪些工具。

换句话说，本书前面分析过的许多"看似洁癖"的设计纪律，到了 SDK 这一章才显出它们共同的目的。

源码分析：`createAgentSession`

SDK 的主入口是工厂函数 `createAgentSession`（`packages/coding-agent/src/core/sdk.ts:169`）。最小用法只有几行：

```
import { createAgentSession, ModelRuntime,
         SessionManager } from "@earendil-works/pi-coding-agent";

const modelRuntime = await ModelRuntime.create();

const { session } = await createAgentSession({
  sessionManager: SessionManager.inMemory(),
  modelRuntime,
});

session.subscribe((event) => {
  if (event.type === "message_update"
      && event.assistantMessageEvent.type === "text_delta") {
    process.stdout.write(event.assistantMessageEvent.delta);
  }
});

await session.prompt("What files are in the current directory?");
```

> **Breaking（v0.80.8）**：这段最小示例在旧版本里是 `AuthStorage.create()` + `ModelRegistry.create(authStorage)` 两件套传入 `createAgentSession`。现在 `CreateAgentSessionOptions` 的模型/认证入口**只剩一个异步的 `modelRuntime?: ModelRuntime`**（`sdk.ts:45`）；`authStorage` / `modelRegistry` 选项已移除。`ModelRuntime.create()` 内部会用 `agentDir/auth.json` + `models.json` 装配好凭证、内建目录、动态目录（第 18 章）。连 `modelRuntime` 都可以省略 —— 不传时 `createAgentSession` 自己 `await ModelRuntime.create({…})`（`sdk.ts:176`）。相应地，`AuthStorage` 及其后端**不再从包根导出**；若只想读一条已存凭证，用 `readStoredCredential()`（`index.ts:26`），或直接用 `ModelRuntime`（也可用 pi-ai 的 `CredentialStore`，见第 4-7 章认证子系统）。

注意几个设计点：

1. 全部有默认值，渐进式覆盖。不传任何参数，`createAgentSession()` 就用 `DefaultResourceLoader` 走标准发现流程、用 `SessionManager.create(cwd)` 落盘会话。需要时再逐项覆盖：

```
// CreateAgentSessionOptions（sdk.ts:37-86 节选）
const { session } = await createAgentSession({
  model: myModel,
  tools: ["read", "bash"],          // 名字白名单（第 19 章）
  excludeTools: ["write"],          // 在白名单之后再排除
  customTools: [myToolDefinition],  // 注册额外的自定义工具
  noTools: "builtin",               // 禁用内建工具但保留扩展工具
  sessionManager: SessionManager.inMemory(),  // 不落盘，纯内存会话
});
```

`SessionManager.inMemory()` 是 SDK 友好的一个细节 —— 测试或一次性任务可以用纯内存会话，不在磁盘上留下 JSONL 文件（对照第 11 章的落盘会话）。

2. 返回值不止 session。 `createAgentSession` 返回 `{ session, extensionsResult, modelFallbackMessage }`（`CreateAgentSessionResult`）：`extensionsResult` 供宿主在需要时自己搭 UI context，`modelFallbackMessage` 在"恢复的会话所存模型与当前不一致"时给出告警。

3. `AgentSession` 就是那个有状态壳。拿到的 `session` 暴露的方法和第 10 章的 Agent、第 26 章的 RPC 命令一一呼应：`prompt()` / `steer()` / `followUp()` / `abort()` / `setModel()` / `compact()` / `getMessages()` / `fork()`，以及 `subscribe(event)` 订阅事件流。RPC 层本质上就是把这些方法序列化成 JSON-L —— SDK 只是把同一组方法**直接**交到你手里。

更低层的组合入口

如果默认工厂还不够灵活，SDK 暴露了更细的组合层：`createAgentSessionRuntime`（`agent-session-runtime.ts:406`）、`createAgentSessionServices` / `createAgentSessionFromServices`。它们把"构造依赖（services）"与"用依赖装配 session"拆成两步，便于在多会话宿主里复用同一批 services。大多数调用方用 `createAgentSession` 就够了，这些是为"完全控制"准备的。

不只是 session：可复用的构件被导出

SDK 还把一批原本是内部细节的构件提升为公共导出，让宿主不必重新造轮子。例如第 20 章提到的 `generateDiffString` / `generateUnifiedPatch` / `EditDiffResult`、第 26 章的 typed `RpcClient`、图片处理（`convertToPng` / resize）、参数解析 `parseArgs`、`CONFIG_DIR_NAME`、以及包内资源路径 helper。这些导出意味着：你可以只取用 pi 的某个能力（比如生成 unified patch），而不必启动整个 agent。

本区间新增/提升的公共导出还包括：

- **`ModelRuntime`**（`index.ts:180`）—— 如上文，模型/认证的规范门面；`ModelRegistry` 仍导出（`index.ts:169`）但仅作 extension 用的同步 compat 投影。
- **`InlineExtension`**（`index.ts:96`）—— 让宿主**内联**声明一个 extension（直接给对象，而非指向磁盘上的 extension 文件路径），SDK 场景下无需为一个临时 hook 单独建文件。
- **消息与工具执行的生命周期事件类型**（如 `message_end`、工具执行的 start/update/end 事件类型），让 TypeScript 宿主能类型安全地 `subscribe` 并 narrow 事件。
- **`JsonlSessionStorage` / `InMemorySessionStorage`**（可插拔存储后端，第 11 章）—— 宿主可自带存储实现，或在 `SessionManager` 之外直接用它们控制会话怎么落盘。
- **CLI 等价的 model 解析** helper —— 把用户在命令行写的 `provider/model:thinking` 串解析成 `Model` 对象，让 SDK 宿主复用与 CLI 完全一致的选模型语义。

第二消费者：evals harness

SDK 有没有真实的第二消费者，是检验"可嵌入库"定位是否成立的试金石 —— 如果只有 pi 自己的 CLI 用 `createAgentSession`，那这套嵌入 API 很容易在演进中悄悄退化。本区间新增的私有包 `packages/evals`（pi-evals，不发布）正是这样一个消费者：它是一套基于 `vitest-evals` 的评测 harness，用 `createHarness` 驱动**真实的** `createAgentSession`，在临时 workspace 里跑任务、汇报最终消息 / transcript / token usage。

```
// file: packages/evals/src/pi-harness.ts:4-11（节选重排）
import {
  createAgentSession, DefaultResourceLoader,
  ModelRuntime, SessionManager, SettingsManager,
} from "@earendil-works/pi-coding-agent";
import { createHarness } from "vitest-evals/harness";
```

它的 import 面把本章讲的构件都串了起来：`ModelRuntime`（上文的规范门面）、`DefaultResourceLoader`、`SessionManager`、`SettingsManager`。harness 从环境变量 `PI_PROVIDER` / `PI_MODEL` 读出要评测的模型（`getRequiredModelSelection()`），装配一个真实 session 跑完再 dispose、清理临时目录。

这个消费者有两层意义。其一，它是 SDK 面的**回归护栏** —— evals 每次运行都在走真实的嵌入路径，`createAgentSession` 的签名一旦破坏就会立刻暴露。其二，它把两个看似无关的特性连了起来：evals 依赖 `PI_PROVIDER` / `PI_MODEL` 这两个**会话环境变量**（第 22 章）来选模型，而这些变量本身是 bash 工具注入子进程的会话上下文的一部分。换句话说，"SDK 嵌入"（本章）和"bash 会话环境变量"（第 22 章）在 evals harness 这里第一次同时落地成一个真实用例。

模式提炼

嵌入 vs 驱动的选择：

维度	SDK 嵌入（本章）	RPC 驱动（第 26 章）
进程	同进程，直接函数调用	子进程，JSON-L over stdio
语言	仅 Node/TS	任意语言
隔离	无（共享进程）	有（独立进程）
开销	低（无序列化、无 framing）	高（每次调用序列化 + 跨进程）
适用	Node 应用内嵌、构造子 agent、测试	IDE 插件、非 Node 客户端、需隔离

经验法则：你是 Node 进程且不需要进程隔离 → 用 SDK；否则 → 用 RPC。

用户能做什么

- **自建 UI**：用 `createAgentSession` + `subscribe` 把事件流接到你自己的 Web/桌面前端，复用 pi 的循环、压缩、工具，而不用 fork pi。
- **构造子 agent**：在一个工具的 `execute` 里再 `createAgentSession`（多用纯内存会话），实现"agent 调 agent"。这正是 subagent 示例扩展的底层做法。

- **只借一个能力**：只想要 unified patch 或图片转换？直接 import 对应导出，不必起 agent。

版本演化说明

本章描述的 SDK 嵌入路径在 v0.66.1（本书基线）尚未作为一等公民成型，是 pi 走向"可嵌入库"定位后逐步固化的，已对照 **v0.82.1**。 本区间最重要的是一处 **Breaking（v0.80.8）**： `CreateAgentSessionOptions` 移除 `authStorage` / `modelRegistry`，改为单一异步入口 `modelRuntime?: ModelRuntime`（`sdk.ts:45`）； `AuthStorage` 不再从包根导出，读凭证改用 `readStoredCredential()`（`index.ts:26`）或 `ModelRuntime`（第 18 章）。最小示例已相应重写为 `ModelRuntime.create()` + `modelRuntime`。 新导出面：`ModelRuntime`（`index.ts:180`）、`InlineExtension`（`index.ts:96`）、消息/工具执行生命周期事件类型、`JsonlSessionStorage` / `InMemorySessionStorage`、CLI 等价 model 解析。 新增真实第二消费者 `packages/evals`（pi-evals，`createHarness` 驱动真实 `createAgentSession`，依赖 `PI_PROVIDER` / `PI_MODEL`），既是 SDK 面回归护栏，也串起第 22 章的会话环境变量。 更早的关键节点仍成立： `tools: string[]` 名字白名单（v0.68.0）；工具参数校验切到 `typebox` 1.x（v0.69.0）；摘要 prompt 中性化以支持非编码复用（v0.79.0）；嵌入时不再强制相邻 package.json（v0.78.1）。 `AgentSession` 的方法面与第 10 章 Agent、第 26 章 RPC 命令保持一一对应。

第 27 章：pi-web-ui — 浏览器里的复用

> **定位**：本章解析 Web UI 如何复用 pi-agent-core 的 Agent 抽象和 pi-ai 的 provider 层。 前置依赖：第 4 章（Provider Registry）、第 24 章（pi-tui）。 适用场景：当你想理解 pi 的 Web 组件库。

> **历史快照说明**： `pi-web-ui` 包已于 commit `b141e1fa`（2026-05-20）从 pi-mono 主仓库移出，且没有官方继任仓库。本章的全部分析、包名（`@mariozechner/pi-web-ui`）与版本号均基于 v0.66.1 历史快照（commit `c779c14e`）——当时该包仍在 `packages/web-ui/`。保留本章，是因为它演示的"Web 宿主复用同一内核"设计论点仍然成立（参见第 2 章关于包移出标准的讨论），但请勿据此在当前主仓库中查找对应代码。

浏览器里的 pi

`pi-web-ui` 是一组 Lit Web Components + Tailwind CSS 构建的可复用组件：聊天消息、模型选择器、文档预览（docx、pdf、xlsx）。

它不是一个完整的 Web 应用 — 而是一个组件库，可以被嵌入到任何 Web 页面中。它直接使用 pi-agent-core 的 `Agent` 类驱动 LLM 交互，订阅 `AgentEvent` 事件更新 UI，不需要中间的 Node.js 后端。

技术选型理由：Lit Web Components 是浏览器原生的组件模型（基于 Custom Elements 和 Shadow DOM），不依赖 React/Vue 等框架。这让 pi-web-ui 的组件可以在任何 Web 环境中使用。

依赖图谱

```
// packages/web-ui/package.json:19-29 (dependencies)
{
  "@lmstudio/sdk": "^1.5.0",
  "@mariozechner/pi-ai": "^0.66.0",
  "@mariozechner/pi-tui": "^0.66.0",
  "docx-preview": "^0.3.7",
  "jszip": "^3.10.1",
  "lucide": "^0.544.0",
  "ollama": "^0.6.0",
  "pdfjs-dist": "5.4.394",
  "xlsx": "https://cdn.sheetjs.com/xlsx-0.20.3/xlsx-0.20.3.tgz"
}
```

这个依赖列表透露了 pi-web-ui 的四个能力层：

1. LLM 连接层。 `@mariozechner/pi-ai` 提供统一的 provider 抽象。`@lmstudio/sdk` 和 `ollama` 是本地 LLM 的直接 SDK — 和 pi-ai 的 provider registry 配合，让 Web UI 可以连接 OpenAI、Anthropic、本地 Ollama、LM Studio 等任何 pi-ai 支持的 provider。

2. 文档预览层。 `pdfjs-dist`（Mozilla 的 PDF 渲染引擎）、`docx-preview`（Word 文档预览）、`xlsx`（Excel 文件解析）。这三个库让 Web UI 可以在浏览器中预览用户上传的文档 — 不需要服务端转换。

3. UI 层。 `lucide` 提供图标集，和 Tailwind CSS 配合构建界面。

4. 工具层。 `jszip` 用于处理压缩文件（docx、xlsx 本质上是 ZIP 包），`@mariozechner/pi-tui` 复用了 TUI 包中的一些工具函数（比如文本处理、颜色计算）。

Peer Dependencies：框架选择

```
// packages/web-ui/package.json:32-35
{
  "peerDependencies": {
    "@mariozechner/mini-lit": "^0.2.0",
    "lit": "^3.3.1"
  }
}
```

`mini-lit` 是 pi-mono 内部的 Lit 轻量封装，提供了简化的组件定义语法。作为 peer dependency 而非 direct dependency，让使用方可以控制 Lit 和 mini-lit 的版本，避免依赖冲突。

连接 pi-ai Provider

pi-web-ui 直接导入并使用 pi-ai 的 provider API：

```
// packages/web-ui/src/dialogs/ProvidersModelsTab.ts:3
import { getProviders } from "@mariozechner/pi-ai";

// packages/web-ui/src/dialogs/ModelSelector.ts:6
import {
  getModels, getProviders, type Model, modelsAreEqual
} from "@mariozechner/pi-ai";
```

这意味着 pi-web-ui 中的模型选择器和 CLI 中的模型选择器使用**完全相同的 provider 注册表**。当 pi-ai 新增一个 provider（比如 Google Gemini），Web UI 和 CLI 同时获得支持，不需要任何额外代码。

这是第 4 章 Provider Registry 设计的直接回报 — 跨平台复用的投资在 Web UI 中变现。

组件架构

pi-web-ui 的核心组件继承自 Lit 的 `LitElement`：

```
// packages/web-ui/src/ChatPanel.ts:18
export class ChatPanel extends LitElement { ... }

// packages/web-ui/src/dialogs/SettingsDialog.ts:116
export class SettingsDialog extends LitElement { ... }

// packages/web-ui/src/dialogs/ModelSelector.ts:50
export class ModelSelector extends DialogBase { ... }

// packages/web-ui/src/dialogs/AttachmentOverlay.ts:15
export class AttachmentOverlay extends LitElement { ... }
```

`DialogBase` 是所有对话框的基类，封装了打开/关闭动画、backdrop 点击关闭、焦点管理等通用逻辑。具体的对话框只需要实现内容渲染。

主要组件包括：

- **`ChatPanel`**：聊天界面的核心面板，管理消息列表和输入框
- **`SettingsDialog`**：设置对话框，包含 API Keys 管理、代理配置
- **`ModelSelector`**：模型选择器，从 pi-ai 获取可用模型列表
- **`AttachmentOverlay`**：附件预览覆盖层，支持多种文档格式
- **`SessionListDialog`**：会话列表管理
- **`CustomProviderDialog`**：自定义 provider 配置

文档预览能力

`AttachmentOverlay` 是 pi-web-ui 中最能体现"浏览器优势"的组件。它支持六种文件类型的预览：

```
// packages/web-ui/src/dialogs/AttachmentOverlay.ts:13
type FileType =
  "image" | "pdf" | "docx" | "pptx" | "excel" | "text";
```

文件类型检测通过 MIME type 和文件扩展名双重判断：

```
// packages/web-ui/src/dialogs/AttachmentOverlay.ts:55-74
private getFileType(): FileType {
  if (!this.attachment) return "text";
  if (this.attachment.type === "image") return "image";
  if (this.attachment.mimeType === "application/pdf")
    return "pdf";
  if (this.attachment.mimeType?.includes("wordprocessingml"))
    return "docx";
  if (this.attachment.mimeType?.includes("presentationml") ||
      this.attachment.fileName.toLowerCase()
        .endsWith(".pptx"))
    return "pptx";
  if (this.attachment.mimeType?.includes("spreadsheetml") ||
      this.attachment.mimeType?.includes("ms-excel") ||
      this.attachment.fileName.toLowerCase()
        .endsWith(".xlsx") ||
      this.attachment.fileName.toLowerCase()
        .endsWith(".xls"))
    return "excel";
  return "text";
}
```

每种文件类型使用不同的渲染策略：

PDF 渲染使用 `pdfjs-dist`。逐页渲染到 Canvas 元素，支持缩放和滚动。`pdfjs-dist` 是 Mozilla 维护的 PDF 渲染引擎，和 Firefox 内置的 PDF 阅读器是同一套代码。

Word 文档渲染使用 `docx-preview`。将 `.docx` 文件（本质上是 ZIP 包中的 XML）解析并渲染为 HTML。支持基本的文本格式、表格、图片。

Excel 渲染使用 `xlsx`（SheetJS）。解析 `.xlsx` 文件的工作表数据，渲染为 HTML 表格。

这三种预览都在浏览器中完成 — 不需要服务端转换。这对隐私敏感的场景很重要：用户上传的文档不会离开浏览器。

存储层

pi-web-ui 有自己的持久化存储层，基于 IndexedDB：

```
// packages/web-ui/src/storage/app-storage.ts:11
export class AppStorage { ... }

// packages/web-ui/src/storage/backends/indexeddb-storage-backend.ts:7
export class IndexedDBStorageBackend
  implements StorageBackend { ... }
```

存储层管理以下数据：

- **Provider API Keys**（`ProviderKeysStore`）：加密存储各个 provider 的 API key
- **Settings**（`SettingsStore`）：用户偏好设置（主题、默认模型等）
- **Sessions**（`SessionsStore`）：对话历史
- **Custom Providers**（`CustomProvidersStore`）：用户配置的自定义 provider

这让 pi-web-ui 可以作为一个独立的 Web 应用运行 — 打开浏览器就能用，不需要 CLI 或后端服务。API key 存储在浏览器的 IndexedDB 中，LLM 调用直接从浏览器发起。

构建工具链

```
// packages/web-ui/package.json:15-17
{
  "build": "tsc -p tsconfig.build.json && " +
    "tailwindcss -i ./src/app.css -o ./dist/app.css --minify",
  "dev": "concurrently ... tsc --watch ... tailwindcss --watch"
}
```

构建分两步：TypeScript 编译（`tsc`）和 Tailwind CSS 生成（`tailwindcss`）。开发模式用 `concurrently` 并行运行两个 watch 进程。

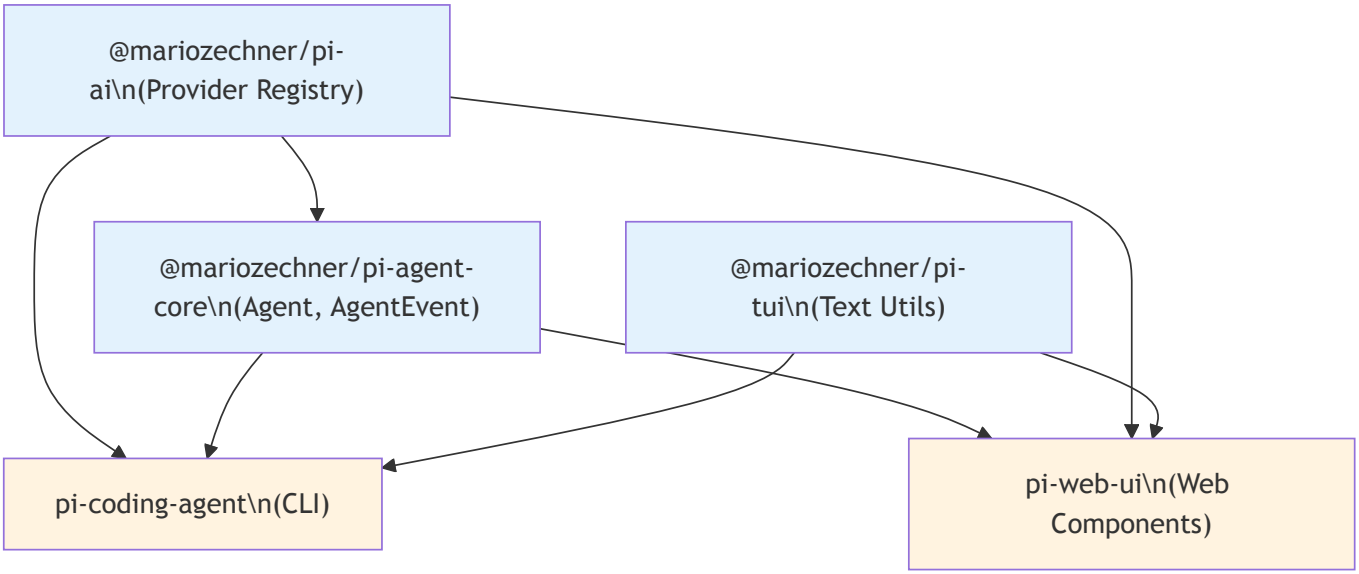
输出是标准的 ES module + CSS 文件：

```
// packages/web-ui/package.json:8-11
{
  "exports": {
    ".": "./dist/index.js",
    "./app.css": "./dist/app.css"
  }
}
```

使用方导入组件（JS）和样式（CSS）两个入口。组件通过 Web Components 的 Custom Elements 注册，不需要额外的初始化代码。

与 CLI 的关系

pi-web-ui 和 pi-coding-agent（CLI）是**并列**的消费者，而非上下游关系：



两者共享 pi-agent-core 的 `Agent` 抽象和 pi-ai 的 provider/模型定义，但各自有独立的 UI 实现。CLI 用终端渲染，Web 用 Lit Web Components。Web UI 的核心组件（`AgentInterface`）直接持有一个 `Agent` 实例，订阅 `AgentEvent` 驱动界面更新。这种架构让 pi 可以同时服务终端用户和 Web 用户，而核心的 agent 交互逻辑只写一次。

取舍分析

得到了什么

框架无关。Lit Web Components 可以嵌入 React、Vue、Angular 或纯 HTML 页面。使用方不需要适配特定的前端框架。

浏览器端 LLM 调用。通过 pi-ai 的 provider 抽象，Web UI 可以直接调用 LLM API，不需要后端代理。这简化了部署（一个静态文件服务器就够了）并提升了隐私性。

丰富的文档预览。PDF、Word、Excel 的浏览器端预览让用户可以直接查看上传的文档内容，不需要下载或在其他应用中打开。

放弃了什么

Web Components 的生态不如 React。社区组件、工具链、开发者经验都不如 React 生态丰富。Lit 的学习曲线对于习惯 React 的开发者来说是一个额外成本。

浏览器端 API key 管理有安全隐患。API key 存储在 IndexedDB 中，虽然浏览器提供了同源保护，但不如服务端管理安全。对于企业场景，可能需要通过代理服务器转发 API 调用而非直接在浏览器中存储 key。

依赖体积。`pdfjs-dist`、`xlsx`、`docx-preview` 这些文档处理库体积不小。如果使用方只需要聊天功能不需要文档预览，仍然需要承载这些依赖的打包体积（除非使用 tree-shaking 和动态导入）。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0。pi-web-ui 是 pi-mono 中最年轻的包，仍在快速演进中。文档预览能力（docx、pdf、xlsx、pptx）是近期添加的。`@lmstudio/sdk` 和 `ollama` 的集成让 Web UI 可以连接本地运行的 LLM，这对离线使用和隐私敏感场景尤为重要。

第 28 章：mom — Slack 里的 Coding Agent

> **定位**：本章展示 pi 内核在 Slack bot 产品中的复用方式。 前置依赖：第 11 章（会话管理）、第 22 章（bash 工具）。 适用场景：当你想理解"同一个 agent 内核，不同的产品壳"在实践中是什么样。

> **历史快照说明**： `mom` 包已于 commit `0ed0d434`（2026-04-30）从 pi-mono 主仓库移出；其方向性继任为独立项目 GitHub `earendil-works/pi-chat`。本章的全部分析、包名（`@mariozechner/pi-mom`）与版本号均基于 v0.66.1 历史快照（commit `c779c14e`）——当时该包仍在 `packages/mom/`。保留本章，是因为它是"同一个 agent 内核、不同的产品壳"这一设计论点最完整的实证案例（参见第 2 章关于包移出标准的讨论），但请勿据此在当前主仓库中查找对应代码。

从终端到 Slack：同一个内核，不同的壳

pi CLI 是终端里的 coding agent。mom 是 Slack 里的 coding agent。两者共享同一个内核（pi-ai + pi-agent-core），但产品形态完全不同：

维度	pi CLI	mom
用户交互	终端 TUI	Slack 消息
会话生命周期	按工作目录	按 channel
工具执行	本地进程	Docker 容器或 host
输出展示	富文本终端	Slack mrkdwn + 线程
多用户	单用户	多用户多频道

mom 的源码总量约 4000 行（含工具），核心在于如何把 pi-coding-agent 的 `AgentSession` 适配到 Slack 的消息模型中。

AgentRunner：channel 级的 agent 实例

mom 的核心抽象是 `AgentRunner` — 每个 Slack channel 有一个独立的 runner 实例，缓存在内存中：

```
// packages/mom/src/agent.ts:392-405
const channelRunners = new Map<string, AgentRunner>();

export function getOrCreateRunner(
  sandboxConfig: SandboxConfig,
  channelId: string,
  channelDir: string
): AgentRunner {
  const existing = channelRunners.get(channelId);
  if (existing) return existing;

  const runner = createRunner(sandboxConfig, channelId, channelDir);
  channelRunners.set(channelId, runner);
  return runner;
}
```

`AgentRunner` 接口很简洁 — 只有 `run()` 和 `abort()` 两个方法。但 `createRunner()` 内部做了大量的适配工作。

复用 AgentSession

mom 最重要的设计决策是**直接复用 pi-coding-agent 的 `AgentSession`**，而不是自己实现会话管理。看 `createRunner()` 的核心结构：

```
// packages/mom/src/agent.ts:435-476
// 创建 Agent（pi-agent-core 层）
const agent = new Agent({
  initialState: { systemPrompt, model, thinkingLevel: "off", tools },
  convertToLlm,
  getApiKey: async () => getAnthropicApiKey(authStorage),
});

// 创建 SessionManager（pi-coding-agent 层）
const contextFile = join(channelDir, "context.jsonl");
const sessionManager = SessionManager.open(contextFile, channelDir);
const settingsManager = createMomSettingsManager(join(channelDir,
"..."));

// 组装 AgentSession（pi-coding-agent 层）
const session = new AgentSession({
  agent,
  sessionManager,
  settingsManager,
  cwd: process.cwd(),
  modelRegistry,
  resourceLoader,
  baseToolsOverride,
});
```

这里的关键：mom 使用了 `AgentSession` 的全部能力 — 会话持久化、自动 compaction、消息同步 — 但替换了 UI 层（用 Slack API 代替 TUI）和工具集（用 mom 专属工具代替通用工具）。

`resourceLoader` 是一个最小实现，因为 mom 不需要 extension 加载、theme 等 CLI 专属功能：

```
// packages/mom/src/agent.ts:453-463
const resourceLoader: ResourceLoader = {
  getExtensions: () => ({ extensions: [], errors: [],
    runtime: createExtensionRuntime() }),
  getSkills: () => ({ skills: [], diagnostics: [] }),
  getPrompts: () => ({ prompts: [], diagnostics: [] }),
  getThemes: () => ({ themes: [], diagnostics: [] }),
  getAgentsFiles: () => ({ agentsFiles: [] }),
  getSystemPrompt: () => systemPrompt,
  getAppendSystemPrompt: () => [],
  extendResources: () => {},
  reload: async () => {},
};
```

这就是"协议式设计"的回报 — `AgentSession` 不关心谁提供 resource，只要实现接口就行。

Channel 级数据隔离

mom 把每个 Slack channel 视为一个独立的 agent 工作空间：

```
~/.pi/mom/data/
├── MEMORY.md                  # 全局记忆
├── settings.json              # 全局设置
├── events/                    # 事件调度文件
├── skills/                    # 全局 skills
├── C123ABC/                   # Channel A
│   ├── MEMORY.md              # Channel 级记忆
│   ├── log.jsonl              # 完整消息历史
│   ├── context.jsonl          # LLM context
│   ├── skills/                # Channel 级 skills
│   ├── attachments/           # 用户上传的文件
│   └── scratch/               # 工作目录
└── C456DEF/                   # Channel B
    └── ...                    # 完全独立的数据
```

每个 channel 有独立的记忆、历史、skills 和工作目录。agent 在 Channel A 中的操作不会影响 Channel B。

双层记忆系统

mom 的记忆分为全局和 channel 两级。`getMemory()` 函数按层级组装：

```
// packages/mom/src/agent.ts:69-103
function getMemory(channelDir: string): string {
  const parts: string[] = [];

  // 全局记忆（跨所有 channel 共享）
  const workspaceMemoryPath = join(channelDir, "..", "MEMORY.md");
  if (existsSync(workspaceMemoryPath)) {
    const content = readFileSync(workspaceMemoryPath, "utf-8").trim();
    if (content) {
      parts.push(`### Global Workspace Memory\n${content}`);
    }
  }

  // Channel 级记忆
  const channelMemoryPath = join(channelDir, "MEMORY.md");
  if (existsSync(channelMemoryPath)) {
    const content = readFileSync(channelMemoryPath, "utf-8").trim();
    if (content) {
      parts.push(`### Channel-Specific Memory\n${content}`);
    }
  }

  return parts.length === 0 ? "(no working memory yet)" :
parts.join("\n\n");
}
```

全局记忆存放跨频道的信息（用户偏好、项目知识），channel 记忆存放频道特定的上下文（正在进行的任务、频道约定）。agent 可以通过 bash 工具直接编辑这些 MEMORY.md 文件。

双层 Skill 系统

Skills 也分为全局和 channel 两级，channel 级 skill 可以覆盖同名的全局 skill：

```
// packages/mom/src/agent.ts:105-139
function loadMomSkills(channelDir: string, workspacePath: string):
Skill[] {
  const skillMap = new Map<string, Skill>();

  // 加载全局 skills
  const workspaceSkillsDir = join(hostWorkspacePath, "skills");
  for (const skill of loadSkillsFromDir({
    dir: workspaceSkillsDir, source: "workspace"
  }).skills) {
    skillMap.set(skill.name, skill);
  }

  // 加载 channel 级 skills（同名覆盖全局）
  const channelSkillsDir = join(channelDir, "skills");
  for (const skill of loadSkillsFromDir({
    dir: channelSkillsDir, source: "channel"
  }).skills) {
    skillMap.set(skill.name, skill);  // Map 覆盖语义
  }

  return Array.from(skillMap.values());
}
```

这让不同 channel 可以展示不同的"人格"。例如，一个 channel 专注于代码审查，另一个 channel 专注于运维监控 — 通过不同的 skills 引导 agent 的行为。

Slack Socket Mode 集成

mom 使用 Slack 的 Socket Mode（WebSocket）而非 HTTP webhook，避免了需要公网可达的 endpoint。核心类型定义展示了 Slack 的消息模型：

```
// packages/mom/src/slack.ts:1-66
import { SocketModeClient } from "@slack/socket-mode";
import { WebClient } from "@slack/web-api";

export interface SlackContext {
  message: {
    text: string;
    rawText: string;
    user: string;
    userName?: string;
    channel: string;
    ts: string;
    attachments: Array<{ local: string }>;
  };
  channelName?: string;
  channels: ChannelInfo[];
  users: UserInfo[];
  respond: (text: string, shouldLog?: boolean) => Promise<void>;
  replaceMessage: (text: string) => Promise<void>;
  respondInThread: (text: string) => Promise<void>;
  uploadFile: (filePath: string, title?: string) => Promise<void>;
  deleteMessage: () => Promise<void>;
}
```

`SlackContext` 封装了 Slack API 的操作集合 — `respond` 回复主消息、`respondInThread` 回复线程、`replaceMessage` 更新已发送的消息、`uploadFile` 上传文件。这些操作被传入 `AgentRunner.run()`，由事件处理器在 agent 执行过程中调用。

线程化工具输出

Slack 的消息限制约束了交互设计。mom 的解决方案是**主消息展示最终结果，线程展示工具执行过程**。

事件订阅代码（只在 runner 创建时注册一次）处理每个 agent 事件：

```
// packages/mom/src/agent.ts:505-544
if (event.type === "tool_execution_start") {
  // 主消息：显示工具 label（如"Reading file..."）
  queue.enqueue(
    () => ctx.respond(`_→ ${label}_`, false), "tool label"
  );
} else if (event.type === "tool_execution_end") {
  // 线程：显示工具的完整参数和结果
  let threadMessage = `*${agentEvent.isError ? "✗" : "✓"} ${
    agentEvent.toolName}*`;
  if (label) threadMessage += `: ${label}`;
  threadMessage += ` (${duration}s)\n`;
  if (argsFormatted)
    threadMessage += `\`\`\`\n${argsFormatted}\n\`\`\`\n`;
  threadMessage += `*Result:*\n\`\`\`\n${resultStr}\n\`\`\``;

  queue.enqueueMessage(
    threadMessage, "thread", "tool result thread", false
  );
}
```

设计要点：

1. **主消息保持简洁**。只显示 `_→ Reading file..._` 这样的斜体标签，不显示完整的工具参数和结果
2. **线程记录完整上下文**。工具名、参数、执行时间、完整结果都在线程中可查
3. **错误额外提示**。工具失败时，除了线程记录外，主消息也会显示截断的错误信息
4. **消息队列保证顺序**。所有 Slack API 调用通过 `queue.enqueue()` 串行化，避免乱序

消息长度处理

Slack 有 40000 字符的消息长度限制。mom 通过 `splitForSlack` 自动分割长消息：

```ts
// packages/mom/src/agent.ts:623-637
const SLACK_MAX_LENGTH = 40000;
const splitForSlack = (text: string): string[] => {
  if (text.length <= SLACK_MAX_LENGTH) return [text];
  const parts: string[] = [];
  let remaining = text;
  let partNum = 1;
  while (remaining.length > 0) {
    const chunk = remaining.substring(0, SLACK_MAX_LENGTH - 50);
    remaining = remaining.substring(SLACK_MAX_LENGTH - 50);
    const suffix = remaining.length > 0
      ? `\n_(continued ${partNum}...)_` : "";
    parts.push(chunk + suffix);
    partNum++;
  }
  return parts;
};
```

Docker Sandbox 实现

mom 的推荐部署方式是 Docker sandbox — 所有 bash 命令在容器中执行，限制了 agent 的文件系统访问。

sandbox 的抽象层很薄，只有一个 `Executor` 接口：

```ts
// packages/mom/src/sandbox.ts:79-91
export interface Executor {
  exec(command: string, options?: ExecOptions): Promise<ExecResult>;
  getWorkspacePath(hostPath: string): string;
}
```

两个实现：

```
// packages/mom/src/sandbox.ts:104-193
class HostExecutor implements Executor {
  async exec(command: string, options?: ExecOptions):
Promise<ExecResult> {
    // 直接在 host 上执行 sh -c command
    const child = spawn(shell, [...shellArgs, command], {
      detached: true, stdio: ["ignore", "pipe", "pipe"],
    });
    // ... timeout + abort signal 处理
  }
  getWorkspacePath(hostPath: string): string {
    return hostPath;  // host 路径不需要转换
  }
}

class DockerExecutor implements Executor {
  constructor(private container: string) {}
  async exec(command: string, options?: ExecOptions):
Promise<ExecResult> {
    // 包装为 docker exec container sh -c 'command'
    const dockerCmd = `docker exec ${this.container} sh -c ${
      shellEscape(command)}`;
    const hostExecutor = new HostExecutor();
    return hostExecutor.exec(dockerCmd, options);
  }
  getWorkspacePath(_hostPath: string): string {
    return "/workspace";  // 容器内统一为 /workspace
  }
}
```

`DockerExecutor` 的实现是委托模式 — 它把命令包装成 `docker exec`，然后交给 `HostExecutor` 执行。关键的安全边界在于：

1. **路径隔离**。容器内只看到 `/workspace`（挂载的数据目录），看不到 host 文件系统
2. **路径转换**。`getWorkspacePath()` 把 host 路径转为容器路径；`translateToHostPath()` 做反向转换（用于文件上传）
3. **启动前验证**。`validateSandbox()` 检查 Docker 可用性和容器运行状态

上下文同步：log.jsonl 与 context.jsonl

mom 面临一个独特的挑战：Slack 消息可能在 agent 不在线时到达。`context.ts` 中的 `syncLogToSessionManager` 解决这个问题：

```
// packages/mom/src/context.ts:42-46
// 确保 agent 离线期间的消息被同步到 LLM context
export function syncLogToSessionManager(
  sessionManager: SessionManager,
  channelDir: string,
  excludeSlackTs?: string,  // 排除当前正在处理的消息
): number
```

同步逻辑：

1. 从 `log.jsonl` 读取所有用户消息
2. 与 `context.jsonl`（SessionManager）中已有的消息做去重比对
3. 只添加 context 中不存在的消息
4. 跳过 bot 消息（agent 的回复已通过正常流程记录）
5. 排除当前正在处理的消息（避免重复）

去重用**消息文本归一化**实现 — 剥离时间戳前缀和附件部分后比较内容。

事件调度系统

mom 不只是被动响应消息。它有一个事件调度系统，让 agent 可以自己安排"闹钟"：

```
// packages/mom/src/events.ts:12-33
export interface ImmediateEvent {
  type: "immediate";
  channelId: string;
  text: string;
}

export interface OneShotEvent {
  type: "one-shot";
  channelId: string;
  text: string;
  at: string;  // ISO 8601 时间
}

export interface PeriodicEvent {
  type: "periodic";
  channelId: string;
  text: string;
  schedule: string;  // cron 语法
  timezone: string;  // IANA 时区
}
```

三种事件类型覆盖了不同场景：

- **Immediate**：脚本或 webhook 触发的即时事件（"新 GitHub issue 开了"）
- **One-shot**：定时提醒（"下午 3 点提醒开会"）
- **Periodic**：定期任务（"每天早上 9 点检查邮箱"）

事件以 JSON 文件形式存放在 `events/` 目录中。`EventsWatcher` 用 `fs.watch` 监控目录变化，用 `croner` 库处理 cron 调度。agent 自己可以通过 bash 工具创建事件文件 — 这是一个优雅的自举设计：agent 的"安排未来行动"能力不需要专门的 API，只需要写文件。

periodic 事件还支持 `[SILENT]` 标记 — 如果定期检查发现没有可报告的内容，agent 回复 `[SILENT]`，mom 会删除状态消息，避免刷屏。

System Prompt 的动态装配

每次 `run()` 执行时，mom 都会重新构建 system prompt，注入最新的：

- 当前 memory 内容

- Slack workspace 的 channel 和 user 列表（带 ID 映射）
- 当前加载的 skills
- sandbox 环境描述（Docker 还是 host）
- 事件系统的使用说明和 cron 格式参考

system prompt 约 300 行，是 mom 最长的单个代码块。它本质上是一份完整的"操作手册"，告诉 agent 它是谁、在什么环境中、能做什么。

取舍分析

得到了什么

安全的多租户。Docker sandbox 限制了 agent 的文件系统访问。channel 级数据隔离防止了跨频道的信息泄露。双层记忆和 skill 系统让不同频道可以有不同的"人格"。

内核复用零修改。mom 没有 fork 或修改 pi-agent-core 或 pi-coding-agent 的任何代码。它通过实现接口（`ResourceLoader` 、 `Executor` ）和订阅事件来适配。这证明了第 30 章所说的"协议式设计"的可行性。

自主调度能力。事件系统让 mom 从被动的"问答机器人"进化为主动的"助手" — 它可以定时检查邮箱、监控系统状态、提醒待办事项。

放弃了什么

Slack 的消息限制约束了交互设计。Slack 消息有字符数限制、不支持复杂的交互组件。mom 通过线程回复来展示详细的工具输出，但体验不如终端 TUI 的实时流式渲染。

单 channel 串行执行。每个 channel 同一时间只处理一条消息。其他消息排队等待或被记录到 log.jsonl 中待后续同步。这是有意的简化 — 避免并发导致的 context 冲突。

缺少同步确认流。pi 的核心并不内建 permission popup；即便某个 CLI 产品壳选择实现确认流，这种同步交互也很难直接搬到 Slack。mom 的安全策略主要依赖 Docker sandbox 的隔离，而不是运行时审批。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0。Mom 的 channel 级 skills 和 memory 是近期添加的，让 mom 可以在不同频道展示不同的"人格"。事件调度系统 也是后期添加的能力，使 mom 从被动响应进化为主动助手。

第 29 章：pods — 为什么这个仓库还要管 GPU

> **定位**：本章解释 pi-mono 为什么同时覆盖"模型供给侧"和"代理消费侧"。 前置依赖：第 4 章（Provider Registry）。 适用场景：当你想理解端到端 agent 系统为什么需要管模型部署。

> **历史快照说明**： `pods` 包已于 commit `0ed0d434`（2026-04-30）从 pi-mono 主仓库移出，且没有官方继任仓库。本章的全部分析、包名（`@mariozechner/pi`）与版本号均基于 v0.66.1 历史快照（commit `c779c14e`）——当时该包仍在 `packages/pods/`。保留本章，是因为它演示的"端到端覆盖模型供给侧"设计论点仍有参考价值（参见第 2 章关于包移出标准的讨论），但请勿据此在当前主仓库中查找对应代码。

端到端可控

pods 是 pi-mono 中最"奇怪"的包 — 一个 coding agent 的仓库为什么要管 GPU pod 编排和 vLLM 部署?

答案是**端到端可控**。如果 agent 只依赖第三方 API（OpenAI、Anthropic），那么模型的可用性、延迟、成本都不在自己掌控中。pods 让用户可以在 DataCrunch、RunPod、Vast.ai 等平台上部署自己的 vLLM 实例，暴露 OpenAI-compatible endpoint。部署完成后，用户在 pi 的 `models.json` 中配置自定义模型（指定 `baseUrl` 指向 pod 的 endpoint），pi 通过已有的 `openai-responses` 或 `openai-completions` API provider（第 4 章）即可调用 — pods 本身不注册新 provider，它只负责让 endpoint 可用。

pods 的代码量很小（~1773 行），功能也很聚焦：SSH 配置 GPU 机器、启动/停止 vLLM、管理模型权重。它不是一个通用的 GPU 编排系统，而是一个让 pi 用户快速获得自有模型推理能力的快捷方式。

pods.ts：Pod 管理命令

pods 的命令结构很直接。`pods.ts` 提供四个操作：

```
// packages/pods/src/commands/pods.ts:14-39
export const listPods = () => {
  const config = loadConfig();
  const podNames = Object.keys(config.pods);
  if (podNames.length === 0) {
    console.log("No pods configured. Use 'pi pods setup' to add.");
    return;
  }
  for (const name of podNames) {
    const pod = config.pods[name];
    const isActive = config.active === name;
    const marker = isActive ? chalk.green("*") : " ";
    const gpuCount = pod.gpus?.length || 0;
    const gpuInfo = gpuCount > 0
      ? `${gpuCount}x ${pod.gpus[0].name}` : "no GPUs detected";
    console.log(`${marker} ${chalk.bold(name)} - ${gpuInfo} -
${pod.ssh}`);
  }
};
```

完整命令清单：

命令	函数	作用
`pi pods list`	`listPods()`	列出所有 pod，标记 active
`pi pods setup <name> <ssh>`	`setupPod()`	配置新 pod（SSH + 环境安装）
`pi pods switch <name>`	`switchActivePod()`	切换 active pod
`pi pods remove <name>`	`removePodCommand()`	从配置中移除 pod

以及 `models.ts` 中的模型管理命令：

命令	函数	作用
`pi start <model>`	`startModel()`	启动 vLLM 实例
`pi stop <name>`	`stopModel()`	停止模型
`pi stop --all`	`stopAllModels()`	停止所有模型
`pi models`	`listModels()`	列出运行中的模型
`pi logs <name>`	`viewLogs()`	查看 vLLM 日志
`pi models known`	`showKnownModels()`	列出预配置的模型

SSH Setup Flow：从零到可用

`setupPod()` 是最复杂的命令，它自动化了 GPU 机器的完整配置流程：

```
// packages/pods/src/commands/pods.ts:44-172
export const setupPod = async (
  name: string,
  sshCmd: string,
  options: {
    mount?: string;
    modelsPath?: string;
    vllm?: "release" | "nightly" | "gpt-oss"
  },
) => {
  // 1. 验证环境变量
  const hfToken = process.env.HF_TOKEN;
  const vllmApiKey = process.env.PI_API_KEY;
  if (!hfToken) { /* 提示用户设置 HF_TOKEN */ }
  if (!vllmApiKey) { /* 提示用户设置 PI_API_KEY */ }

  // 2. 测试 SSH 连接
  const testResult = await sshExec(sshCmd, "echo 'SSH OK'");

  // 3. 复制安装脚本到远程机器
  const scriptPath = join(__dirname, "../../scripts/pod_setup.sh");
  await scpFile(sshCmd, scriptPath, "/tmp/pod_setup.sh");

  // 4. 远程执行安装脚本（2-5 分钟）
  let setupCmd = `bash /tmp/pod_setup.sh ` +
    `--models-path '${modelsPath}' ` +
    `--hf-token '${hfToken}' ` +
    `--vllm-api-key '${vllmApiKey}'`;
  await sshExecStream(sshCmd, setupCmd, { forceTTY: true });

  // 5. 检测 GPU 配置
  const gpuResult = await sshExec(sshCmd,
    "nvidia-smi --query-gpu=index,name,memory.total " +
    "--format=csv,noheader");
  // 解析 GPU 信息...

  // 6. 保存 pod 配置
  addPod(name, { ssh: sshCmd, gpus, models: {}, modelsPath });
};
```

整个流程的设计思路是**一条命令完成所有事**：

1. **环境变量检查**。HF_TOKEN（Hugging Face 下载模型权重）和 PI_API_KEY（vLLM 端点认证）是必需的

2. **SSH 连通性测试**。在开始耗时操作前先确认连接可用
3. **SCP 脚本传输**。用 `pod_setup.sh` 在远程机器上安装 Python、CUDA 工具链、vLLM
4. **流式输出**。`sshExecStream` 带 `forceTTY: true`，让用户看到安装进度
5. **GPU 自动检测**。通过 `nvidia-smi` 获取 GPU 数量、型号、显存
6. **配置持久化**。保存到本地 `~/.pi/pods.json`，后续操作引用

SSH 抽象层

pods 没有使用任何 SSH 库 — 它直接调用系统的 `ssh` 和 `scp` 命令：

```
// packages/pods/src/ssh.ts:12-68
export const sshExec = async (
  sshCmd: string,   // 如 "ssh user@host"
  command: string,  // 要执行的远程命令
): Promise<{ stdout: string; stderr: string; exitCode: number }>

export const sshExecStream = async (
  sshCmd: string,
  command: string,
  options?: { forceTTY?: boolean },
): Promise<number>  // 返回 exit code

export const scpFile = async (
  sshCmd: string,
  localPath: string,
  remotePath: string,
): Promise<boolean>
```

这是一个刻意的简化选择。`sshCmd` 参数接受完整的 SSH 命令（如 `ssh -i ~/.ssh/id_rsa user@host`），用户可以自由配置 SSH 代理、跳板机、端口转发等。pods 不需要理解 SSH 配置的细节。

vLLM 管理：模型启停

`startModel()` 是 pods 的核心功能 — 在远程 GPU 机器上启动一个 vLLM 实例：

```
// packages/pods/src/commands/models.ts:78-197
export const startModel = async (
  modelId: string,
  name: string,
  options: {
    pod?: string;
    vllmArgs?: string[];
    memory?: string;    // GPU 显存使用比例
    context?: string;   // 上下文长度（4k/8k/16k...）
    gpus?: number;      // GPU 数量
  },
) => {
  const { name: podName, pod } = getPod(options.pod);

  // 自动选择 GPU 配置
  if (isKnownModel(modelId)) {
    // 预配置模型：自动选择最优 GPU 数量和参数
    for (let gpuCount = pod.gpus.length; gpuCount >= 1; gpuCount--) {
      modelConfig = getModelConfig(modelId, pod.gpus, gpuCount);
      if (modelConfig) {
        gpus = selectGPUs(pod, gpuCount);
        vllmArgs = [...(modelConfig.args || [])];
        break;
      }
    }
  } else {
    // 未知模型：默认单 GPU
    gpus = selectGPUs(pod, 1);
  }
  // ... 启动 vLLM screen session
};
```

GPU 分配策略：

1. **预配置模型**：`model-configs.ts` 为常用模型（Llama、Qwen、Mistral 等）维护了经过测试的 vLLM 参数。根据 pod 的 GPU 型号和数量自动选择最优配置
2. **GPU 数量覆盖**：`--gpus N` 让用户强制指定 GPU 数量，pods 验证是否有对应配置
3. **显存/上下文覆盖**：`--memory 90%` 和 `--context 32k` 覆盖默认值
4. **自定义参数**：`--vllm <args>` 完全绕过自动配置，直接传递 vLLM 参数
5. **未知模型**：默认单 GPU，不做任何假设

OpenAI-Compatible Endpoint

vLLM 启动后会暴露 OpenAI-compatible API。这是 pods 和 pi 连接的关键 — pi 的 provider registry（第 4 章）天然支持 OpenAI API 格式：

```
用户 → pi CLI → Provider Registry → OpenAI-compatible API → vLLM → GPU
                                          ↑
                                    pods 部署的端点
```

pods 启动时配置的 `PI_API_KEY` 就是 vLLM 端点的认证 key。用户在 pi 中配置 provider 时，只需要指向远程机器的 IP 和端口。

模型生命周期管理

```
pi pods setup  →  配置 SSH + 安装环境
      ↓
pi start model →  启动 vLLM（screen session）
      ↓
pi models      →  查看运行状态
      ↓
pi logs model  →  查看 vLLM 日志
      ↓
pi stop model  →  停止 vLLM
```

vLLM 运行在 `screen` session 中（不是 Docker），这样 SSH 断开后进程继续运行。模型权重存储在 `modelsPath`（通常是挂载的 NFS/SFS），多个模型可以共享权重缓存。

本地 pi 如何连接远程 pod

连接流程分三步：

Step 1：部署模型

```
export HF_TOKEN=hf_xxx
export PI_API_KEY=my-secret-key
pi pods setup my-a100 "ssh root@1.2.3.4" \
  --models-path /mnt/models
pi start deepseek-ai/DeepSeek-V3 --name ds-v3
```

Step 2：配置 provider

pods 启动 vLLM 后，打印端点地址（如 `http://1.2.3.4:8000`）。用户在 pi 的 provider 配置中添加这个端点，指向 OpenAI-compatible API。

Step 3：在 pi 中使用

pi CLI 的模型选择器（第 4 章的 `getModel()`）现在可以选择自部署的模型。对 agent 循环来说，自部署模型和第三方 API 没有任何区别 — 都是通过 `StreamFunction` 接口调用。

vLLM 版本选择

pods 支持三种 vLLM 版本：

版本	适用场景
`release`	默认。稳定版 vLLM
`nightly`	需要最新功能（如新模型支持）
`gpt-oss`	专门为 GPT-OSS 模型优化的 fork

版本选择在 `setupPod()` 时确定，记录在 pod 配置中。`listPods()` 会显示 vLLM 版本信息，对 `gpt-oss` 版本额外警告其兼容性限制。

配置管理

pods 的配置存储在 `~/.pi/pods.json`：

```
// packages/pods/src/config.ts
export const loadConfig = (): Config => { /* 读取 JSON */ };
export const saveConfig = (config: Config): void => { /* 写入 JSON */ };
export const getActivePod = (): { name: string; pod: Pod } | null;
export const addPod = (name: string, pod: Pod): void;
export const removePod = (name: string): void;
export const setActivePod = (name: string): void;
```

配置包含：所有 pod 的 SSH 命令、GPU 信息、已部署的模型列表、模型路径、vLLM 版本。`active` 字段标记默认使用的 pod，大多数命令不需要显式指定 `--pod`。

取舍分析

得到了什么

模型自主权。用户可以运行自己的模型，不受第三方 API 的价格和策略变化影响。DataCrunch 上一台 4xA100 的月费可能只是调用 Claude API 一周费用的零头。

一键部署。从裸机 GPU 到可用的 OpenAI-compatible endpoint，一条命令搞定。这对不熟悉 vLLM、CUDA、模型部署的开发者来说，降低了巨大的门槛。

无缝集成。因为 vLLM 暴露标准 OpenAI API，pi 的 provider registry 无需修改就能对接。自部署模型和第三方 API 在 agent 层完全透明。

放弃了什么

职责边界模糊。把部署工具放在 agent 仓库里，让 monorepo 的范围从"agent 开发"扩展到了"模型运维"。但对于需要端到端控制的用户，这种"职责越界"反而是便利。

没有 GPU 编排能力。pods 不做多机调度、自动扩缩容、健康检查。它假设用户手动管理 GPU 机器的生命周期。对于需要生产级 GPU 集群管理的场景，应该使用 Kubernetes + GPU Operator 等专业工具。

SSH 依赖。直接调用系统 `ssh` 命令意味着用户必须先配置好 SSH 密钥和连接。这对有 SSH 经验的开发者很自然，但对新手可能是障碍。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0。Pods 是 pi-mono 中变化最少的包 — 它的核心功能（SSH 配置 + vLLM 管理）自引入以来保持稳定。 近期添加了 `gpt-oss` vLLM 版本支持和更多预配置模型。

第 30 章：极简核心，能力外置

> **定位**：本章提炼 pi 的核心设计方法论。 前置依赖：全书前 29 章。 适用场景：当你想把 pi 的设计经验应用到自己的系统中。

内核只做三件事

回顾前面的章节，pi 的内核（pi-ai + pi-agent-core）只做三件事：

1. **调模型**（第 4-7 章）：统一 20+ 家厂商，流式事件，跨模型消息变换
2. **跑循环**（第 8-9 章）：无状态的双层循环，三阶段工具执行管道
3. **管状态**（第 10 章）：有状态的 Agent 壳，事件订阅，消息队列

其余所有功能 — 会话持久化、上下文压缩、system prompt 装配、工具实现、UI 渲染、配置管理 — 都在内核之外。

用行数说话

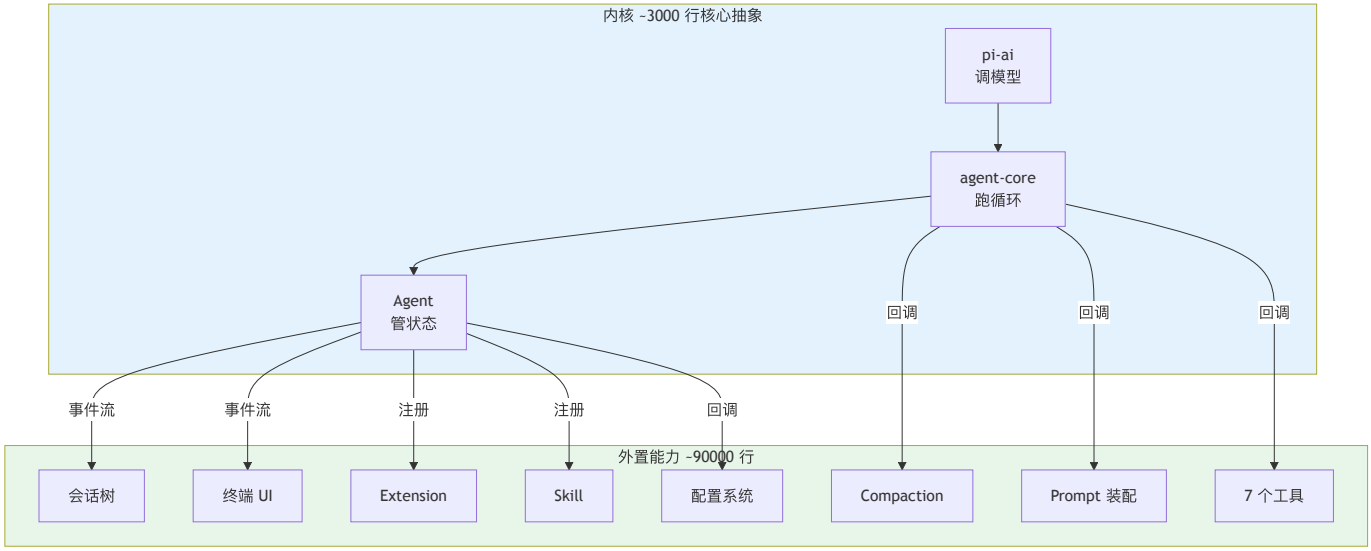
pi-mono 共约 120,000 行 TypeScript（不含测试和生成代码）。内核的占比很小：

包	行数	职责	层级
pi-ai	~26,875	模型调用、provider 注册、消息变换	内核
pi-agent-core	~1,859	agent 循环、事件系统、状态管理	内核
内核合计	**~28,734**		**24%**
pi-coding-agent	~42,058	会话管理、prompt 装配、工具实现、配置	产品层
pi-tui	~10,764	终端 UI 渲染	UI 层
mom	~4,046	Slack bot 适配	产品壳

包	行数	职责	层级
pods	~1,773	GPU 部署工具	工具
web-ui	~14,623	Web UI	UI 层

注意 pi-ai 的行数较多，是因为它包含了 20+ 家 provider 的适配代码（每个 provider 约 200-500 行的流式转换逻辑）。**真正的核心抽象**（`StreamFunction`、`Context`、`Model` 类型、`agentLoop`、`Agent` 类）加起来不到 3000 行。

pi-agent-core 只有 ~1,859 行，是整个 monorepo 中最小的包。但它定义了最关键的协议：agent 循环、事件类型、工具接口、回调钩子。



这种比例（核心 24%，外围 76%）不是偶然的。它反映了一个判断：**agent 系统的核心应该是一个协议（事件流 + 回调），而不是一个框架（内建的功能集合）。**

协议 vs 框架：用例子说清楚

"协议式设计"和"框架式设计"是两种根本不同的架构策略。用具体例子对比：

例子 1：添加一个新工具

框架式（假设的 AgentFramework）：

```
// 框架内建了工具注册系统，你在框架的约束内添加
class MyAgent extends AgentFramework {
  @Tool({ name: "search", schema: searchSchema })
  async search(query: string) {
    return await doSearch(query);
  }
}
```

协议式（pi）：

```
// 工具只是一个满足 AgentTool 接口的对象
const searchTool: AgentTool<typeof searchSchema> = {
  name: "search",
  label: "search",
  description: "Search for matching content",
  parameters: searchSchema,
  async execute(_toolCallId, { query }) {
    return await doSearch(query);
  },
};
// 传入 Agent 的 initialState.tools 即可
```

区别不在语法糖，而在**控制权**。框架式设计中，工具的生命周期由框架管理（注册、发现、权限检查都内建）。协议式设计中，工具只是一个数据结构，产品层可以自由地创建、组合、替换。

例子 2：实现上下文压缩

框架式：

```
// 框架内建了 compaction 策略
agent.setCompactionStrategy("summarize", {
  threshold: 100000,
  model: "claude-haiku",
});
```

协议式（pi）：

```
// transformContext 回调可以做任何事
const config: AgentLoopConfig = {
  transformContext: (messages) => {
    // 你自己决定压缩策略
    if (estimateTokens(messages) > threshold) {
      return compactMessages(messages);
    }
    return messages;
  },
};
```

框架式更方便（一行配置），协议式更灵活（可以实现框架没预见到的策略）。

例子 3：多产品复用

这是协议式设计的杀手级优势。pi 的同一个内核被三个完全不同的产品使用：

```
pi CLI（终端 coding agent）
├── 使用：TUI 渲染、本地文件系统工具、自定义权限钩子
├── 不使用：Slack API、Docker sandbox

mom（Slack bot）
├── 使用：Docker sandbox、Slack 消息输出、事件调度
├── 不使用：TUI、同步确认流、本地工具权限

web-ui（浏览器 IDE）
├── 使用：浏览器内 Agent 视图、proxy-aware streamFn、IndexedDB 存储
├── 不使用：TUI、Slack API、Docker sandbox
```

如果内核是框架式的（内建了 TUI、本地文件工具、权限 popup），mom 和 web-ui 要么 fork 框架，要么在框架上面做大量适配。pi 的协议式内核没有这些假设 — 它只定义"模型怎么调"和"循环怎么跑"，产品层自行决定其余一切。

与竞品的架构对比

以下对比是**结构性的**（架构选择），不是**评价性的**（孰优孰劣）。每种选择都有其适用场景。

Claude Code

Claude Code 是 Anthropic 官方的 CLI agent。架构对比：

维度	pi	Claude Code
模型层	多 provider 抽象	单一 Anthropic API
循环引擎	可组合纯函数（agentLoop）	内建循环 + 工具管理
工具系统	外置，通过接口注入	内建标准工具集
扩展机制	Extension API + Skill	Slash command + CLAUDE.md
会话管理	外置 SessionManager	内建会话持久化
UI 层	独立的 pi-tui 包	内建终端 UI
多产品	CLI / Slack / Web	CLI 为主，headless 模式

关键差异：Claude Code 是**单一产品优化**的设计 — 它只需要支持一个 provider（Anthropic）、一个 UI（终端）、一组工具。这让它可以把更多功能内建，降低上手成本。pi 是**多产品基座**的设计 — 它需要支持多个 provider、多个 UI、多个产品形态，所以必须把更多东西外置。

Cursor / Windsurf（IDE Agent）

维度	pi	IDE Agent（Cursor 类）
宿主	独立进程	IDE 插件（VS Code extension）
编辑操作	工具调用 → Edit/Write	直接操作 IDE API
上下文	手动管理（transformContext）	IDE 提供语义索引
文件导航	Glob/Grep 工具	LSP + 语义搜索
多模型	用户可切换	内建路由（不同任务用不同模型）

关键差异：IDE agent 有一个巨大优势 — IDE 本身就是上下文源（打开的文件、编辑历史、语言服务器）。pi 作为独立进程，需要通过工具（Glob、Grep、Read）来获取这些

信息。但 pi 的独立性也意味着它不受 IDE 限制 — 可以在 Slack、Web、CI/CD 中运行。

Aider

维度	pi	Aider
语言	TypeScript	Python
编辑策略	LLM 生成 edit/write 工具调用	LLM 生成 unified diff
循环模型	通用 agent 循环	专注于 code edit 循环
Git 集成	通过 bash 工具	内建 git 操作
上下文管理	手动 + transformContext	Repo map + 文件标签

关键差异：Aider 是**垂直整合**的 coding assistant — git、diff、编辑、测试一体化。pi 是**水平分层**的 agent 基座 — coding 只是其中一个用例。Aider 在纯代码编辑场景中更高效（diff 策略比 tool call 更节省 tokens），但 pi 的通用循环可以做 Aider 做不到的事（Slack bot、自动化管道等）。

"不内建"的判断标准

pi 如何决定一个功能是内建还是外置？有三个判断标准：

1. 这个功能是否产品相关?

如果功能的实现取决于产品形态，就应该外置。

- **权限确认**：CLI 弹终端 popup，Slack 发消息让用户回复，Web 弹 modal → **外置**
- **工具执行**：CLI 在本地执行，mom 在 Docker 中执行 → **外置**
- **输出渲染**：CLI 用 TUI，Slack 用 mrkdwn，Web 用 HTML → **外置**
- **流式事件格式**：所有产品都需要相同的事件流 → **内建**

2. 这个功能是否有多种合理实现?

如果功能有多种同样合理的实现方式，就应该外置为回调或接口。

- **compaction 策略**：可以用 LLM 总结、可以按时间截断、可以按 token 预算裁剪 → **外置**为 `on("session_before_compact", ...)` 钩子
- **上下文变换**：可以注入 plan 指令、可以过滤历史、可以动态切换模型 → **外置**为 `transformContext` 回调
- **消息序列化**：每个 LLM 的消息格式不同 → **内建**在 provider 层统一处理

3. 这个功能是否需要系统级别的一致性?

如果功能的不一致会导致系统行为不可预测，就应该内建。

- **事件类型**：所有组件必须用相同的事件定义 → **内建**
- **工具执行管道**：prepare/execute/finalize 三阶段必须统一 → **内建**
- **工具的具体 schema**：不同工具有不同参数 → **外置**

与竞品的设计对比（决策矩阵）

设计维度	pi	典型框架式 Agent
Sub-agents	不内建，用 tool call 组合	内建 multi-agent orchestration
权限控制	beforeToolCall 钩子	内建 permission popup
Plan mode	transformContext 组合	内建 plan/execute 模式
会话存储	SessionManager（产品层）	内建 memory store
Prompt 管理	AGENTS.md + skills	内建 prompt registry
模型路由	产品层选择 model	内建 model router
错误恢复	产品层实现 retry	内建 retry + fallback
可观测性	事件流 + 订阅	内建 tracing/logging

pi 的每个"不内建"都对应一个"用更底层的机制组合出来"。不是没想到，是故意没做。

极简核心的代价

诚实地说，极简核心不是没有代价：

上手成本高。新用户面对的是"洋葱架构" — pi-ai、pi-agent-core、pi-coding-agent、pi-tui 四层，每层有自己的类型和回调。理解一个"消息从用户输入到模型回复"的完整路径，需要穿越所有四层。

重复劳动。不同的产品壳（CLI、mom、web-ui）各自实现了类似的功能（session 加载、system prompt 构建、工具权限）。虽然它们各有差异，但重复的部分不小。

文档负担。内建功能可以在框架文档中一次性说明。外置功能需要每个产品各自文档化其组合方式。

调试困难。bug 可能出现在内核、产品层、或两者的交互中。分层越多，追踪问题的路径越长。

这些代价是 pi 为多产品适应性付出的"税"。对于只需要单一产品的团队，框架式设计可能是更好的选择。

取舍分析

得到了什么

极致的适应性。同一个内核跑在终端（pi CLI）、Slack（mom）、浏览器（pi-web-ui）、GPU 集群（pods）。每个产品只需要实现自己需要的"壳"。内核的 24% 代码驱动了 100% 的产品形态。

长期可维护性。内核的变化不影响产品层（只要协议不变）。产品层的变化不影响内核。这种解耦在 monorepo 中已经得到验证 — mom 和 web-ui 可以独立演进，不需要协调内核修改。

放弃了什么

上手成本。用户需要理解"洋葱架构"的每一层才能有效使用 pi。没有"开箱即用"的体验 — 你要么接受默认配置，要么深入理解系统才能定制。

开发速度。在框架式系统中，添加一个新功能可能只需要调一个 API。在 pi 中，你可能需要理解三层的交互才能找到正确的注入点。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0 的架构快照。 极简核心的设计哲学从 pi 的第一个版本就确立了，后续版本只是在不扩大内核的前提下 通过 extension、skill、回调等机制添加新能力。行数统计可能随版本变化， 但内核与外围的比例预计保持稳定。

第 31 章：反主流选择背后的判断

> **定位**：本章逐个解释 pi 的"不做"决策。 前置依赖：第 30 章（极简核心）。 适用场景：当你在设计自己的 agent 系统时犹豫"要不要内建 X"。

每个"不做"都是一个"用 Y 组合出来"

pi 做出了四个与主流 agent 框架相反的选择：不内建 sub-agents、不内建 MCP、不内建 permission popup、不内建 plan mode。这些选择不是因为缺少资源或没有想到，而是基于一个统一的判断：**如果一个功能可以用更底层的机制组合出来，就不应该把它内建到内核中。**

下面逐个展开。

为什么不内建 Sub-agents

主流做法

大多数 agent 框架提供"启动子 agent"的 API：

```
# 假设的框架式 API
orchestrator = Agent(tools=[
    SubAgent("researcher", model="claude-haiku", tools=[search]),
    SubAgent("coder", model="claude-sonnet", tools=[edit, bash]),
])
result = await orchestrator.run("Fix the login bug")
# 框架自动管理子 agent 的生命周期、消息传递、结果汇总
```

这种设计的优势是声明式 — 用户定义 agent 拓扑，框架处理编排细节。

pi 的做法

pi 不内建 sub-agent 概念。但它可以通过 tool call 组合实现。实际上，pi-coding-agent 中的 `Agent` 工具就是这样实现的：

Step 1：定义 Agent 工具

Agent 工具和其他工具（bash、edit、read）一样，都是满足 `AgentTool` 接口的对象：

```
// 概念代码，展示核心逻辑
const agentTool: AgentTool<typeof agentToolSchema> = {
  name: "Agent",
  label: "Agent",
  description: "Delegate a bounded sub-task to another agent run",
  parameters: agentToolSchema,
  async execute(_toolCallId, { prompt }) {
    // Step 2：在工具的 execute 函数里启动新的 agent 循环
    const subAgent = new Agent({
      initialState: {
        systemPrompt: "你是一个专注于子任务的 agent...",
        model: parentAgent.state.model,
        tools: parentAgent.state.tools,  // 复用父 agent 的工具
      },
      convertToLlm,
      getApiKey,
    });

    // Step 3：运行子 agent 循环
    const result = await subAgent.prompt(prompt);

    // Step 4：收集子 agent 的输出作为 tool result 返回
    const output = subAgent.state.messages
      .filter(m => m.role === "assistant")
      .map(m => extractText(m))
      .join("\n");

    return { content: [{ type: "text", text: output }] };
  },
};
```

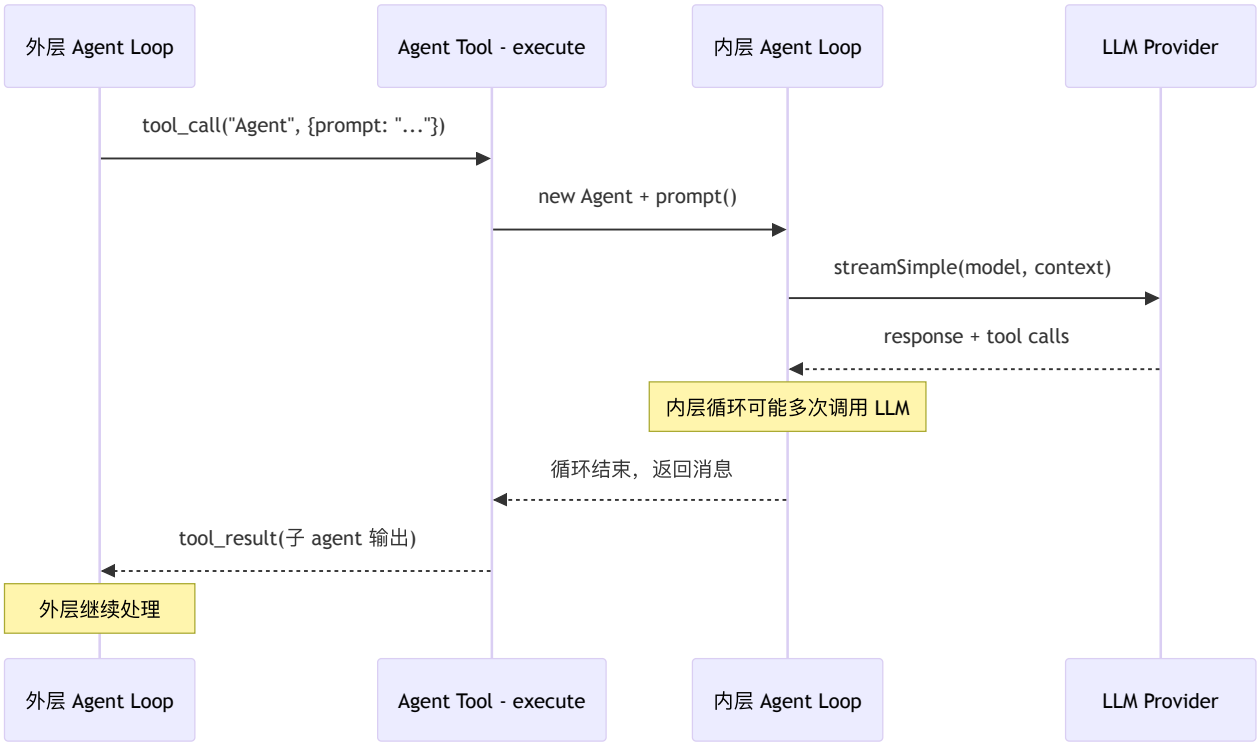
Step 2：外层 agent 自然调度

外层 agent 通过 tool call 调用 Agent 工具。对外层 agent 来说，Agent 工具和 bash 工具没有本质区别 — 都是"发送参数，等待结果"。循环引擎的三阶段管道（prepare →

execute → finalize）同样适用。

Step 3：嵌套的循环引擎

关键的架构支撑是第 8 章介绍的**循环引擎是可组合的纯函数**。`agentLoop` 没有全局状态，不依赖单例 — 在一个工具的 execute 函数里再启动一个 `agentLoop` 是完全安全的。内层循环有自己的消息列表、自己的回调、自己的停止条件。



为什么这样更好?

1. **组合自由**。子 agent 可以用不同的模型、不同的工具集、不同的 system prompt。这不需要框架支持"子 agent 配置" — 只需要在 execute 函数里自由构造
2. **透明的资源管理**。子 agent 的 token 消耗作为工具执行时间的一部分被跟踪。不需要专门的"子 agent 资源计量"机制
3. **自然的错误处理**。子 agent 失败 = 工具执行失败，由外层 agent 的标准错误处理逻辑处理
4. **零新概念**。开发者已经理解了工具调用和 agent 循环，不需要额外学习"sub-agent API"

什么时候这样不够好?

当你需要**并行子 agent**（多个子 agent 同时执行不同任务）时，pi 的串行工具执行管道是限制。虽然可以用 `Promise.all` 在一个工具内并行启动多个循环，但这需要产品层自

己处理并发控制。CrewAI、AutoGen 等框架的内建并行编排在这种场景下更方便。

为什么不内建 MCP

MCP 是什么

MCP（Model Context Protocol）是 Anthropic 提出的工具标准化协议。它定义了"工具服务器"（MCP server）和"工具客户端"的交互标准，让工具可以跨 agent 框架复用。

pi 的替代方案

pi 有两层能力扩展机制，覆盖了 MCP 的主要用例：

Skill（第 16 章）：纯 markdown 文件，注入 system prompt

```
---
name: tdd-workflow
description: 按 TDD 流程编写代码
---
# TDD Workflow
1. 先写测试
2. 运行测试（应该失败）
3. 写最少的实现代码让测试通过
4. 重构
```

Extension（第 15 章）：TypeScript 代码，运行时能力

```
// Extension 可以注册工具、订阅事件、操作 UI
export default {
  name: "my-extension",
  setup(api) {
    api.registerTool({
      name: "my-tool",
      label: "my-tool",
      description: "My custom tool",
      parameters: myToolSchema,
      execute: async (_toolCallId, args) => { /* ... */ },
    });
    api.on("tool_execution_end", (event) => {
      // 监听所有工具执行
    });
  },
};
```

Skill vs MCP：何时用哪个

需求	Skill	MCP	Extension
告诉 LLM 按某个流程工作	适合	过度设计	过度设计
提供特定领域知识	适合	过度设计	过度设计
调用外部 API	不适合	适合	适合
操作 agent 内部状态	不适合	不适合	适合
跨框架复用	不适合	适合	不适合
修改 UI 行为	不适合	不适合	适合

pi 的判断：**大多数 agent 的"能力扩展"不需要运行时代码**。"按 TDD 流程编写代码"用一个 markdown skill 就够了。"代码审查时注意安全问题"也是一个 markdown skill。"连接数据库查询数据"才需要运行时能力 — 而这时 Extension 比 MCP 有更深的系统集成（可以订阅事件、操作 UI、访问会话）。

MCP 的真正价值在**跨框架互操作** — 一个 MCP server 可以同时被 Claude Code、Cursor、pi 使用。但 pi 目前的生态定位是"自己的产品自己的工具"，跨框架互操作不是优先级。

MCP 的开销

一个 MCP server 意味着：

- 一个独立进程（需要启动、维护、监控）
- JSON-RPC 通信（序列化/反序列化开销）
- 进程间错误处理（超时、崩溃、重连）
- 额外的配置（server 地址、认证）

对于"告诉 LLM 按 TDD 流程工作"这种用例，启动一个 MCP server 就像用大炮打蚊子。Skill 是零开销的 — 它只是一个被读取到 system prompt 中的文件。

为什么不内建 Permission Popup

主流做法

许多 agent 产品在执行敏感操作前弹出确认框：

```
Agent wants to run: rm -rf node_modules
[Allow] [Deny] [Allow All]
```

这看起来很合理 — 让用户在危险操作前确认。

pi 的做法

pi 不内建 permission popup。官方 README 的表述就是 “No permission popups”。但 `beforeToolCall` 钩子（第 9 章）让产品层可以实现自己的权限策略。

策略 1：某个产品壳自行实现确认流

```
const config: AgentLoopConfig = {
  beforeToolCall: async ({ toolCall, args }) => {
    if (isDangerous(toolCall.name, args)) {
      const confirmed = await confirmDangerousAction(
        `Allow ${toolCall.name}?`
      );
      if (!confirmed) {
        return { block: true, reason: "User denied tool execution" };
      }
    }
    return undefined;  // 允许执行
  },
};
```

策略 2：命令白名单

```
const config: AgentLoopConfig = {
  beforeToolCall: async ({ toolCall, args }) => {
    if (toolCall.name === "bash") {
      const cmd = (args as { command: string }).command;
      if (!ALLOWED_COMMANDS.some(p => cmd.startsWith(p))) {
        return { block: true, reason: "Command not in whitelist" };
      }
    }
    return undefined;
  },
};
```

策略 3：完全自动（mom 的行为）

```
// mom 在 Docker sandbox 中运行，不需要用户确认
const config: AgentLoopConfig = {
  beforeToolCall: async () => undefined,  // 从不阻止
};
```

策略 4：基于角色的审批

```
const config: AgentLoopConfig = {
  beforeToolCall: async ({ toolCall }) => {
    const userRole = getCurrentUserRole();
    if (userRole === "admin") return undefined;
    if (RESTRICTED_TOOLS.includes(toolCall.name)) {
      return { block: true, reason: "Insufficient privileges" };
    }
    return undefined;
  },
};
```

同一个钩子，四种完全不同的安全策略。内建 popup 只能实现第一种。更重要的是，**安全策略和产品形态强相关**：

- CLI：很适合自行实现交互式确认（用户在终端前面）
- Slack bot：用户不在"终端前面"，确认流程需要异步消息
- CI/CD 管道：完全自动，任何人工确认都会打断流水线
- 内部工具：基于角色的权限，不是"每次确认"

内建 popup 假设了一种特定的交互模式。`beforeToolCall` 不做任何假设。

为什么不内建 Plan Mode

什么是 Plan Mode

Plan mode（先规划再执行）是很多 agent 产品的标配。典型流程：

1. 用户提出任务
2. Agent 生成执行计划（不执行任何工具）
3. 用户审阅计划，确认或修改
4. Agent 按计划逐步执行

pi 的做法

pi 不内建 plan mode，但 `transformContext` 回调（第 8 章）可以实现它。

`transformContext` 在每次 LLM 调用前执行，可以修改发送给模型的消息列表。Plan mode 的本质就是"在不同阶段注入不同的指令"：

实现草图：transformContext 版 plan mode

```
type PlanModeState = "planning" | "reviewing" | "executing";

function createPlanModeConfig(
  getState: () => PlanModeState,
  getPlan: () => string | null,
): Partial<AgentLoopConfig> {
  return {
    transformContext: (messages) => {
      const state = getState();
      const plan = getPlan();

      if (state === "planning") {
        // 注入计划指令：只输出计划，不调用工具
        return [
          ...messages,
          {
            role: "user",
            content: [{
              type: "text",
              text: "[SYSTEM] 现在是规划阶段。" +
                "分析任务，输出详细的执行计划。" +
                "不要调用任何工具。" +
                "用编号列表格式输出计划步骤。"
            }],
          },
        ];
      }

      if (state === "executing" && plan) {
        // 注入已确认的计划作为执行指导
        return [
          ...messages,
          {
            role: "user",
            content: [{
              type: "text",
              text: `[SYSTEM] 按以下计划逐步执行：\n${plan}\n` +
                "每完成一个步骤，报告进度后继续下一步。"
            }],
          },
        ];
      }

      return messages;  // reviewing 阶段或无计划时不修改
```

```
    },

    // 在 planning 阶段禁止工具调用
    beforeToolCall: async () => {
      if (getState() === "planning") {
        return {
          block: true,
          reason: "Planning phase - tools disabled"
        }
      }
      return undefined;
    },
  };
}
```

使用方式：

```
let state: PlanModeState = "planning";
let plan: string | null = null;

const config = createPlanModeConfig(
  () => state,
  () => plan,
);

// 阶段 1：生成计划
const planResult = await session.prompt("Fix the login bug", config);
// planResult 包含 agent 输出的计划文本

// 阶段 2：用户审阅（产品层实现确认 UI）
plan = extractPlanFromMessages(session.messages);
const confirmed = await userConfirm(plan);

// 阶段 3：执行
if (confirmed) {
  state = "executing";
  await session.prompt("Execute the plan", config);
}
```

为什么不内建更好?

1. **Plan 格式可定制**。有的产品要 markdown 列表，有的要 JSON，有的要流程图。产品层控制 system prompt 就能控制输出格式
2. **确认方式可定制**。CLI 弹终端确认，Slack 发消息等回复，Web 弹 modal。产品层控制确认逻辑
3. **Plan 粒度可定制**。有的场景需要粗粒度计划（"Step 1: 读代码 → Step 2: 改代码"），有的需要细粒度（每个文件的修改方案）。`transformContext` 的指令决定粒度
4. **Plan 可迭代**。用户审阅后修改计划、追加约束，再重新规划。这是 `transformContext` 的自然用法，不需要框架支持"计划修改 API"

什么时候内建更好?

如果你的产品 100% 需要 plan mode，且确认方式固定（比如始终是 CLI 弹窗），那内建 plan mode 更方便。pi 选择不内建是因为它的产品形态多样（CLI、Slack、Web），每个形态对 plan mode 的需求不同。

底层机制总结

四个"不内建"的功能用三个底层机制组合实现：

不内建的功能	组合机制	核心回调
Sub-agents	Tool call + Agent 循环	`AgentTool.execute()`
MCP 替代	Skill（markdown）+ Extension（代码）	system prompt + `registerTool()`
Permission popup	权限钩子	`beforeToolCall()`
Plan mode	上下文变换 + 权限钩子	`transformContext()` + `beforeToolCall()`

三个回调（`execute`、`beforeToolCall`、`transformContext`）覆盖了大量的"内建功能"。这不是巧合 — 这三个回调分别控制了 agent 循环的三个关键点：

- **execute**：控制"agent 做什么"（工具行为）
- **beforeToolCall**：控制"agent 能不能做"（权限）
- **transformContext**：控制"agent 看到什么"（上下文）

做什么 + 能不能做 + 看到什么 = agent 行为的完全控制。

取舍分析

得到了什么

更少的内建概念，更大的组合空间。四个"不内建"的功能用三个底层机制组合实现。用户可以创造 pi 设计者没有预见到的组合 — 比如"在特定时间段自动切换到 plan mode"、"根据用户历史行为动态调整权限策略"、"子 agent 使用比父 agent 更便宜的模型"。

统一的心智模型。开发者只需要理解工具、回调、事件三个概念，就能实现任何功能。不需要分别学习 sub-agent API、MCP 配置、permission API、plan mode API。

放弃了什么

"开箱即用"的功能丰富度。使用 pi 的开发者需要自己组合这些功能，而不是调用现成的 API。对于想快速上手的用户，这是障碍。

社区生态互操作。不内建 MCP 意味着无法直接使用 MCP 生态中的工具服务器。虽然可以写 Extension 来桥接 MCP，但这是额外的工作。

最佳实践的传递。内建功能自带"推荐用法"。不内建的功能需要文档和示例来传递最佳实践，否则每个团队都会自己发明一套用法。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0。"不内建"的决策从 pi 设计之初就确立了。 但随着社区反馈，一些"组合方式"被逐步抽象为 extension 模板，降低了组合的门

> 槛。未来 MCP 桥接（通过 Extension 连接 MCP server）可能成为官方支持的模式。

第 32 章：这套架构的适用边界

> **定位**：本章帮助读者判断 pi 的架构适不适合自己的场景。 前置依赖：第 30-31 章（设计哲学）。 适用场景：当你在评估是否基于 pi 构建产品。

适合什么

1. 需要深度定制的 agent 产品。如果你的产品和"通用聊天机器人"差别很大 — 比如一个特定领域的 coding assistant、一个基于 Slack 的运维 bot、一个带自定义 UI 的内部工具 — pi 的分层架构让你可以只替换需要的层，保留其余。

2. 重视工程纪律的团队。pi 的架构要求开发者理解分层边界、事件流契约、回调语义。这对工程能力有门槛，但回报是系统的可预测性和可维护性。

3. 需要支持多 LLM 厂商的场景。pi-ai 层的 provider 抽象让切换和混用 LLM 成为一行代码的事。如果你的产品需要同时支持 Claude、GPT、Gemini、自部署模型，pi 的统一调用面省去了大量适配工作。

4. 需要多产品形态的场景。如果你的 agent 需要同时运行在终端、Slack、Web、API 等多个入口，pi 的协议式内核让你只写一次循环逻辑，每个入口只实现自己的"壳"。mom 就是最好的证明（第 28 章）。

不适合什么

1. 需要开箱即用的简单 chatbot。如果你只想快速上线一个"问答机器人"，Vercel AI SDK 或 LangChain 更合适。pi 的价值在定制化，不在快速启动。一个简单的 chatbot 用 pi 的架构相当于"杀鸡用牛刀"。

2. 需要复杂的多 agent 编排。如果你的场景是"10 个 agent 协作完成一个任务"，pi 的单 agent 循环模型需要你自己在上层搭建编排层。专注于 multi-agent 的框架（如 CrewAI、AutoGen）可能更直接。虽然 pi 可以用 tool call 组合子 agent（第 31 章），但这不等于"原生多 agent 编排"。

3. 非 TypeScript/Node.js 技术栈。pi 是 TypeScript 项目，运行在 Node.js 上。如果你的团队主力是 Python 或 Go，使用 pi 意味着引入额外的技术栈。这不仅是语言问题 — 还涉及包管理（npm）、运行时（Node.js）、类型系统（TypeScript）的学习成本。

4. 需要极低延迟的嵌入式场景。pi 的分层架构在每次 LLM 调用时经历 transformContext → convertToLlm → stream → 事件分发 的完整管道。对于延迟敏感的实时场景（如语音助手），这些层次可能引入不必要的开销。

团队评估清单

在决定是否采用 pi 之前，回答以下五个问题：

Q1：你的 agent 需要运行在几种产品形态中?

- **1 种**（只有 CLI 或只有 Web）→ pi 的多产品适应性对你没有价值。考虑更垂直的方案。
- **2-3 种**（CLI + Web、CLI + Slack 等）→ pi 的分层架构开始有回报。内核复用能省大量重复代码。
- **4+ 种** → pi 的设计正是为这种场景优化的。

Q2：你需要支持几个 LLM provider?

- **1 个**（只用 Claude 或只用 GPT）→ 直接调用厂商 SDK 更简单。pi-ai 的 provider 抽象是多此一举。
- **2-3 个** → pi-ai 的统一调用面开始有价值。
- **4+ 个或包含自部署模型** → pi-ai 几乎是必要的。从头适配每个 provider 的流式 API 差异是大量工作。

Q3：你的团队是否愿意阅读源码?

pi 不是一个"看文档就能用"的框架。由于功能外置，很多"怎么实现 X"的答案在源码中（看已有产品如何组合），而不在 API 文档中。

- **团队习惯阅读和参考开源项目的源码** → 适合 pi。
- **团队期望完善的 API 文档和教程** → 当前阶段不适合。

Q4：你的安全模型是什么？

- **sandbox 隔离**（Docker、VM）→ pi 天然支持（见 mom 的 Docker sandbox）。
- **交互式确认** → pi 不内建，但产品层可以基于 `beforeToolCall` 自行实现。
- **基于角色的权限控制** → pi 支持，但完全由产品层实现。
- **需要内建的安全审计和合规** → pi 没有内建，需要自己通过事件订阅实现审计日志。

Q5：你的迭代速度需求是什么？

- **快速原型，一周内上线 MVP** → pi 的上手成本太高。用 Vercel AI SDK 或 Claude API + 简单循环更快。
- **中期项目，1-3 个月** → 如果团队有 TypeScript 经验，pi 的分层架构值得投入。
- **长期产品，6+ 个月维护** → pi 的可维护性和可扩展性在长期回报最大。

评估结论：如果 5 个问题中有 3 个以上指向"适合"，pi 是一个合理的选择。如果只有 1-2 个，投入产出比可能不够。

从其他框架迁移到 pi

从 LangChain 迁移

LangChain 用户最大的转变是心智模型：从"用框架提供的 Chain/Agent 类"变为"自己组合回调和工具"。

LangChain 概念	pi 对应	迁移策略
`ChatModel`	`getModel()` + provider	替换模型初始化代码
`AgentExecutor`	`Agent` + `agentLoop`	重写主循环（通常更简单）
`Tool`	`AgentTool<TParams>`	接口相似，改 schema 格式
`Memory`	`SessionManager`	替换持久化逻辑

LangChain 概念	pi 对应	迁移策略
`Chain`	`transformContext` + tool 组合	分解为回调和工具调用
`CallbackHandler`	事件订阅	订阅 `AgentEvent`
`OutputParser`	直接处理 assistant message	无需 parser 层

迁移核心步骤：

1. **替换模型层**。内建 provider 直接用 `getModel("provider", "model")`；如果你有自定义 provider，再额外接入 `registerApiProvider()`
2. **重写工具**。将 `@tool` 装饰器改为 `AgentTool` 对象。schema 从 Pydantic 改为 JSON Schema
3. **删除 Chain**。大多数 Chain 的功能用 `transformContext` 就够了
4. **替换 Memory**。用 SessionManager 替换 LangChain 的 BufferMemory/ConversationMemory
5. **重写 Agent 循环**。通常比 LangChain 的 AgentExecutor 更短（pi 的循环引擎做了更多事）

从 Vercel AI SDK 迁移

Vercel AI SDK 是轻量级方案。迁移到 pi 通常是因为需要更复杂的工具执行管道或多产品支持。

Vercel AI SDK	pi 对应	迁移策略
`streamText()`	`streamSimple()`	替换调用
`tool()`	`AgentTool`	类似接口
`generateText()`	`Agent.prompt()`	替换为 agent 循环
`useChat()` (React)	自行实现或用 web-ui	需要自建 UI 层

迁移核心步骤：

1. **替换流式调用**。`streamText()` → `streamSimple()`，事件格式略有不同
2. **添加 agent 循环**。Vercel AI SDK 没有内建循环，pi 的 `agentLoop` 提供自动工具调用

3. **如果需要 React UI**，可以参考 pi-web-ui 直接持有 `Agent` 的方式，或者单独参考 `modes/rpc/` 构建 headless 后端

从 Claude Code 扩展到 pi

如果你已经在使用 Claude Code 并想构建自己的产品，pi 提供了"从 CLI agent 到平台"的路径。

Claude Code	pi 对应	扩展策略
Slash commands	Extension `registerCommand()`	可复用概念
CLAUDE.md	AGENTS.md + SYSTEM.md	类似机制
单— Anthropic API	多 provider 支持	解锁更多模型
固定工具集	可替换的工具集	可定制
CLI only	CLI + Slack + Web + API	多产品形态

二次开发指南

扩展的二次开发表

我想做什么	优先修改	参考章节	复杂度	代码量预估
加一个新的 provider 实现	`packages/ai` 中实现 provider + `registerApiProvider()` + model wiring	第 4、18 章	中	~200-600 行
加一个新工具	Extension → `registerTool()`	第 15、	低	~50-200 行

我想做什么	优先修改	参考章节	复杂度	代码量预估
		19 章		
改 system prompt	创建 `SYSTEM.md` 或 `AGENTS.md`	第 13-14 章	最低	0 行代码
自定义权限策略	`beforeToolCall` 钩子	第 9 章	低	~30-100 行
自定义 compaction	Extension hook: `on("session_before_compact", ...)`	第 12 章	中	~100-300 行
加一个新 UI 模式	参考 `modes/rpc/` 实现新 mode	第 26 章	高	~500-2000 行
支持新的消息类型	`CustomAgentMessages` 声明合并	第 10 章	中	~100-200 行
自定义会话存储	实现新的 SessionManager	第 11 章	中	~200-500 行
加一个新 OAuth provider	`registerOAuthProvider()`	第 7 章	中	~200-400 行
自定义上下文管理	`transformContext` 回调	第 8 章	中	~50-200 行
构建 Slack bot	参考 mom 的架构	第 28	高	~2000-4000

我想做什么	优先修改	参考章节	复杂度	代码量预估
		章		行
构建 Web UI	参考 web-ui 直接消费 `Agent`，或参考 `modes/rpc/` 做后端	第 26、27 章	高	~3000-5000 行
添加自部署模型	参考 pods 的 vLLM 集成	第 29 章	中	~500-1000 行
实现 plan mode	`transformContext` + `beforeToolCall`	第 31 章	中	~100-300 行
实现 sub-agent	Agent 工具 + 嵌套循环	第 31 章	中	~100-300 行
添加审计日志	事件订阅 + 日志写入	第 10 章	低	~50-100 行
实现成本控制	`transformContext` 检查 token 预算	第 8 章	低	~50-100 行

关键入口文件

如果你计划二次开发，以下是最常接触的文件：

```
packages/
├── ai/src/
│   ├── models.ts              # Models 集合 / createProvider（添加
provider 的入口）
│   ├── providers/             # 各内建 provider 的 factory
│   ├── types.ts               # Model, Context, StreamFunction 类型
│   └── models.json            # 模型定义（id, cost, contextWindow）
├── agent/src/
│   ├── agent-loop.ts          # 循环引擎（核心抽象）
│   ├── agent.ts               # Agent 类（状态容器）
│   └── types.ts               # AgentTool, AgentEvent 类型
├── coding-agent/src/core/
│   ├── agent-session.ts       # 产品层的 agent 包装
│   ├── session-manager.ts     # 会话持久化
│   ├── extensions/            # Extension API
│   ├── tools/                 # 内建工具实现
│   ├── system-prompt.ts       # system prompt 装配
│   └── prompt-templates.ts    # prompt 模板
└── tui/src/
    └── tui.ts                 # 终端 UI（如果构建 CLI 产品）
```

推荐的二次开发路径

路径 1：最小改动 — 只换 prompt 和工具

适合：在 pi 的现有产品形态（CLI）上做领域定制。

1. 创建 `AGENTS.md` 定义领域知识
2. 创建 skills 定义工作流程
3. 可选：通过 Extension 添加领域工具
4. 不需要修改任何 pi 源码

路径 2：中等改动 — 添加新的产品壳

适合：在 pi 的内核上构建新的产品形态（如 Discord bot、API 服务）。

1. 参考 mom 的架构，创建新的入口包
2. 实现 `ResourceLoader` 接口
3. 创建产品特定的工具集
4. 接入产品特定的 I/O（消息平台、HTTP 等）
5. 订阅 agent 事件，适配输出格式

路径 3：深度改动 — 修改内核行为

适合：需要改变 agent 循环的基本行为（如并行工具执行、自定义停止条件）。

1. Fork pi-agent-core
2. 修改 `agentLoop` 的循环逻辑
3. 扩展 `AgentEvent` 类型（新的事件类型）
4. 注意：需要同步更新依赖 pi-agent-core 的所有上层包

强烈建议从路径 1 开始，只在确认现有机制无法满足需求时才进入路径 2 或 3。大多数"我需要修改内核"的需求，最终都可以用 `transformContext` + `beforeToolCall` + Extension 组合解决。

长期展望

pi 的架构边界会随生态成熟而变化：

- **更多 Extension 模板** → 降低"组合的门槛"，让常见模式开箱可用
- **MCP 桥接** → 通过 Extension 连接 MCP 生态，解锁跨框架工具互操作
- **多语言 SDK** → Python/Go binding for pi-ai 层，降低技术栈门槛
- **社区 skills 市场** → 类似 npm，分享和发现领域 skills

但核心架构 — 协议式内核、能力外置、三层回调 — 预计不会改变。这是 pi 的设计本体，不是暂时的实现选择。

取舍分析

得到了什么

清晰的决策框架。本章的五个评估问题和二次开发路径为读者提供了结构化的决策依据。不是"pi 好不好"的判断，而是"pi 适不适合你"的分析。

放弃了什么

营销友好的叙事。诚实地列出"不适合什么"和迁移成本，可能劝退一些潜在用户。但对于真正要用 pi 构建产品的团队，这些信息比"一切皆可"的宣传更有价值。

版本演化说明

> 本章核心分析基于 pi-mono v0.66.0。适用边界的判断会随着 pi 生态的成熟 （更多 extension 模板、更完善的文档、可能的多语言 SDK）而变化。 二次开发的复杂度估算基于当前代码库，可能随 API 稳定化而降低。

附录

A. 核心类型速查表

pi-ai 层

类型	文件	用途	关键字段
`Model<TApi>`	types.ts	模型定义	`id: string` — 唯一标识（`claude-sonnet-4-5`） `provider: string` — 提供（如 `anthropic`） `api: TApi` — API 类型标识 `cost: { input, output, cacheRead, cacheWrite }` — 每 token 价格 `contextWindow: number` — 下文窗口大小 `maxOutput?: number` — 最输出 token 数
`Context`	types.ts	LLM 调用上下文	`systemPrompt?: string` — 选系统提示词 `messages: Message[]` — 历史 `tools?: Tool[]` — 可用工义
`AssistantMessageEvent`	types.ts	流式事件	`start` — 初始化 partial assistant message

类型	文件	用途	关键字段
			`text_*` / `thinking_*` / `toolcall_*` — 分块流式更 `done` — 成功结束并携带最息 `error` — 失败或中止结束并带错误消息
`StreamFunction`	types.ts	Provider 必须实现的流式函数	签名：`(model, context, options?) => AssistantMessageEventSt` 契约：返回事件流，不通过常传递运行期错误
`Usage`	types.ts	Token 使用量	`input: number` — 输入 to `output: number` — 输出 t `cacheRead: number` — 缓取 `cacheWrite: number` — 缓写入 `cost: { input, output, cacheRead, cacheWrite,`

pi-agent-core 层

类型	文件	用途	关键字段
`AgentMessage`	types.ts	扩展消息类型	联合类型：`Message | CustomAgentMessages[...]` 标准消息来自 `pi-ai`：`user` / `assistant` / `toolResult`

类型	文件	用途	关键字段
			产品层可通过声明合并追加自定义消息
`AgentTool<TParams>`	types.ts	工具定义	`name: string` — 工具名称 `parameters: TParams` — 参数（继承 Tool<TParameters>） `label: string` — UI 显示的可读名称 `prepareArguments?: (args) => Static<TParams>` — 可选的参数预处理 `execute(toolCallId, params, signal, onUpdate): Promise<AgentToolResult>` — 执行函数
`AgentEvent`	types.ts	循环生命周期事件	`type` 联合： `agent_start / agent_end` — 本轮运行开始/结束 `turn_start / turn_end` — 一轮 assistant turn 开始/结束 `message_start / message_update / message_end` — 消息流式生命周期 `tool_execution_start / tool_execution_update / tool_execution_end` — 工具执行生命周期

类型	文件	用途	关键字段
`AgentLoopConfig`	types.ts	循环引擎配置	`model: Model` — 本轮使用的模型 `convertToLlm` — `AgentMessage[] -> Message[]` 转换 `transformContext?` — LLM 调用前的上下文变换 `beforeToolCall?` / `afterToolCall?` — 工具调用前后钩子 `getSteeringMessages?` / `getFollowUpMessages?` — 运行中注入消息
`AgentState`	agent.ts	Agent 可变状态	`systemPrompt: string` — 当前系统提示词 `model: Model` — 当前模型 `thinkingLevel: ThinkingLevel` — 后续 turn 的思考级别 `tools` / `messages` — 以 copy-on-assign 方式暴露的数组状态 `isStreaming` / `streamingMessage` / `pendingToolCalls` / `errorMessage` — 当前运行态

pi-coding-agent 层

类型	文件	用途	关键
`SessionEntry`	session-manager.ts	会话持久化条目	9 种类型联合：`message` — 用户 `thinking_level` 级别变更 `model_change` — `compaction` — 压 `branch_summary` `custom` — 自定义 `custom_message` `label` — 标签 `session_info` — 每条有 `id`, `pare` `timestamp`
`AgentSessionEvent`	agent-session.ts	产品层事件	`AgentEvent` 的超 `queue_update` 、 `compaction_sta` `compaction_end` `auto_retry_sta` `auto_retry_end`
`AgentSession`	agent-session.ts	产品层 agent 包装	组合了 `Agent` + `SessionManager` `SettingsManage` `prompt(text, o` 送消息并运行循环 `compact(custom` — 主动触发压缩 `subscribe(list`

类型	文件	用途	关键
			`AgentSessionEv` `abort()` — 中止
`Skill`	skills.ts	能力扩展定义	`name: string` — `description: s` `filePath: stri` 路径 `baseDir: strin` `sourceInfo` — 来 `disableModelIn` 否从 prompt 中除
`Extension`	core/extensions/types.ts	运行时扩展	`name: string` — `setup(api)` — 初 API 提供：`regis` `registerComman` `handler)` 等

B. 设计模式索引

模式	出现位置	章节	模
显式集合装配（createProvider + setProvider）	models.ts, providers/*.ts, auth/oauth/	第 4、7 章	用 `Models` 集合进来（`createProvi` 认证随 `Provide` 用 DI 框架、不用 `api-registry.` `oauthProviderI` v0.80.0 起被这套

模式	出现位置	章节	模
有损变换（isSameModel 判断）	transform-messages.ts	第5章	跨模型消息变换时 thinking 块），便
流式契约（Must not throw）	StreamFn type, AgentLoopConfig	第6、8章	Provider 的 stre 常，错误通过事件 要 try-catch
双层循环（steering + follow-up）	agent-loop.ts runLoop()	第8章	外层 steering 循 试，内层 follow- 后的后续对话
三阶段工具执行（prepare/execute/finalize）	agent-loop.ts	第9章	prepare 获取资源 执行操作，finali 任何阶段可中止
声明合并扩展（CustomAgentMessages）	types.ts	第10章	TypeScript `decl` interface mergi 定义消息类型而不
Copy-on-assign（getter/setter + slice）	agent.ts MutableAgentState	第10章	读取 `messages` `messages = [.` 防止外部意外修改
Append-only 树（JSONL + parentId）	session-manager.ts	第11章	每条记录有唯一 支从分支点的 id 删除。JSONL 格式
三级配置覆盖（全局/项目/目录）	settings-manager.ts, resource-loader	第13章	全局配置 < 项目配 近原则，越具体越 层叠覆盖
Pluggable I/O（EditOperations, BashOperations）	edit.ts, bash.ts, find.ts	第19-	工具不直接调用 `child_process`

模式	出现位置	章节	模
		23 章	入。测试用 moc 行
极简组件接口（Component）	tui.ts	第 24 章	UI 组件实现 `ren number): strin` `handleInput?(` `invalidate()` 。有状态管理框架
消息队列串行化	agent.ts (mom)	第 28 章	所有 Slack API 调 行执行，避免消 返回 void，错误
委托模式（DockerExecutor → HostExecutor）	sandbox.ts	第 28 章	DockerExecutor `docker exec` ， 执行。关注点分

C. 一次完整请求的时序图

关键节点说明

1. **transformContext 是 `AgentMessage[]` 级的预处理点**。裁剪历史、注入额外上下文、实现 plan-like 控制通常放在这里
2. **convertToLlm 是 `AgentMessage[] -> Message[]` 的边界转换**。它负责过滤或改写应用层消息；真正的跨 provider 兼容处理还会在 `pi-ai` 的 provider 层继续发生

3. **三阶段工具执行保证了安全性**。prepare 阶段可以拒绝执行（beforeToolCall 钩子），execute 阶段实际操作，finalize 阶段格式化输出
4. **事件驱动的持久化主要挂在 `message_end` 上**。`tool_execution_end` 主要用于 UI 和 extension 观测，真正写入会话的是随后产生的 `toolResult` 消息

D. /compact 命令的端到端追踪

下面追踪用户在 pi CLI 中输入 `/compact` 命令时，系统内部发生的完整过程。

Phase 1：命令解析

```
用户输入: /compact
    ↓
interactive-mode: 识别 `/compact` 或 `/compact ...`
    ↓
handleCompactCommand(customInstructions)
    ↓
AgentSession.compact(customInstructions?)
```

`/compact` 不经过 agent 循环本身。它先由 interactive mode 在本地识别，再直接调用 `AgentSession.compact(...)`。

Phase 2：Compaction 触发

```
compact handler
    ↓
SessionManager.getEntries()  → 获取当前所有会话条目
    ↓
estimateTokens(messages)     → 估算当前 context token 数
    ↓
Agent.state.messages         → 获取当前消息列表
    ↓
emit("compaction_start", { reason: "manual" })
```

Phase 3：LLM 总结

```
compaction_start 事件
    ↓
构建 compaction prompt:
  "请总结以下对话的关键信息..."
  + 当前所有消息
    ↓
completeSimple(compactionModel, compactionContext)
  → 调用 LLM 生成总结
    ↓
收集 LLM 输出 → summary text
```

Compaction 使用的模型可能和主对话模型不同（通常用更快更便宜的模型）。

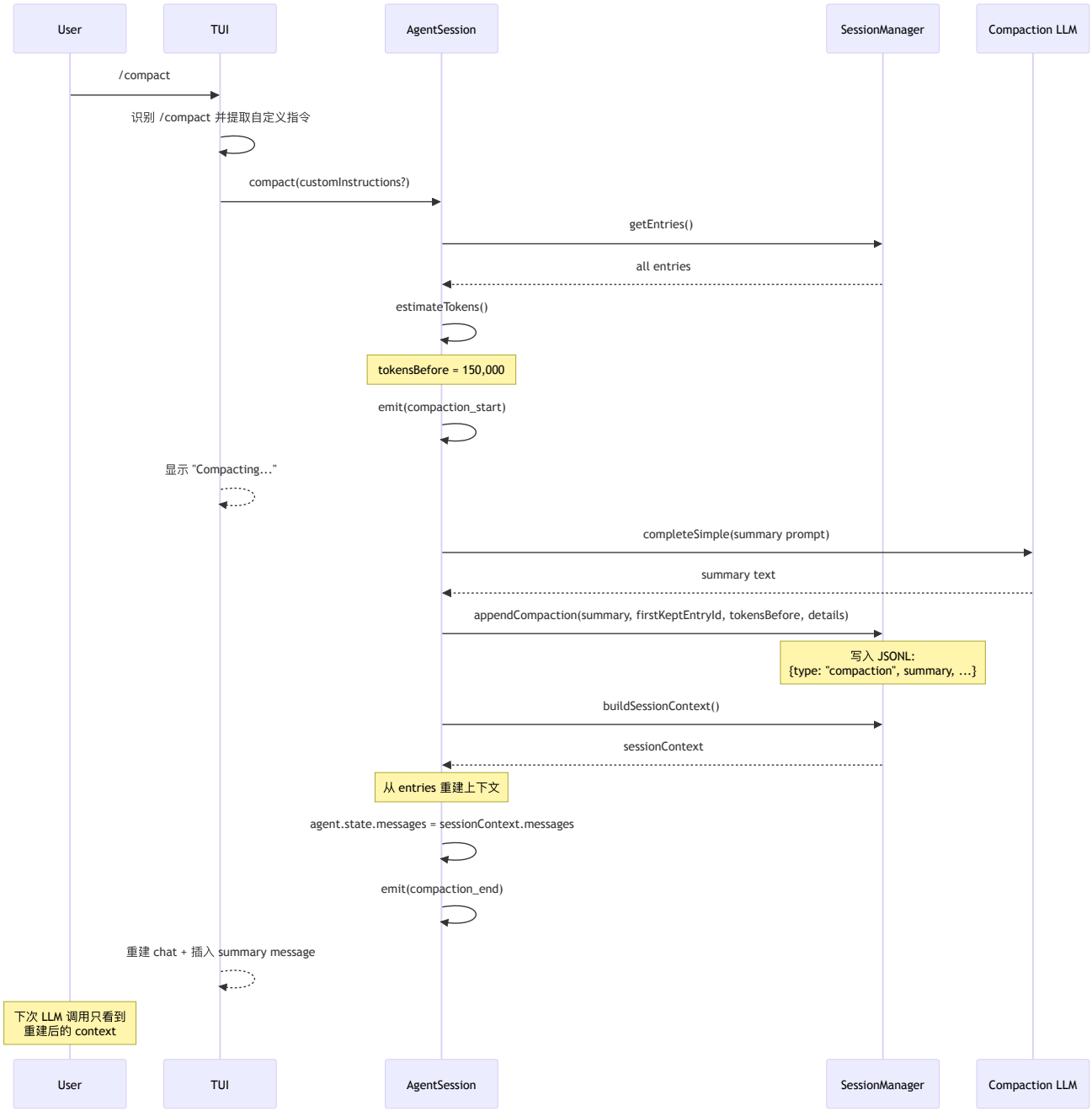
Phase 4：状态更新

```
summary text
    ↓
SessionManager.appendCompaction(
  summary,
  firstKeptEntryId,
  tokensBefore,
  details?
)
  → 将 compaction 记录追加到 JSONL
    ↓
buildSessionContext()
  → 从 entries 重建完整的 session context
    ↓
agent.state.messages = sessionContext.messages
  → 用重建后的消息列表替换 agent 状态
    ↓
emit("compaction_end", { result: { summary, firstKeptEntryId,
tokensBefore, details } })
```

Phase 5：UI 更新

```
compaction_end 事件
    ↓
TUI 订阅者收到事件
    ↓
清空并重建 chat
    ↓
插入 compaction summary message
    ↓
刷新 footer / 状态显示
```

完整数据流



关键观察

1. `/compact` **不经过 agent 循环**。它是一次本地命令处理 + 一次独立的 LLM 调用
2. **状态重建而非直接替换**。`appendCompaction` 记录压缩事件后，`buildSessionContext()` 从 entries 重建上下文，再赋值给 `agent.state.messages`。不是直接构造一条 summary 消息替换
3. **持久化保留历史**。SessionManager 记录 compaction 事件，但不删除历史条目。JSONL 是 append-only 的，`buildSessionContext()` 负责根据 compaction 记录决定哪些 entries 构成当前 context

4. **事件驱动 UI**。TUI 不直接参与 compaction 逻辑，只订阅事件更新显示

E. 二次开发入口清单

我想做什么	起点文件/函数	参考章节
加一个新 LLM provider	`packages/ai` 中实现 provider + `registerApiProvider()` + model wiring	第 4、18 章
加一个新工具	Extension API → `registerTool()`	第 15、19 章
加一个新 slash command	Extension API → `registerCommand()`	第 15 章
改 system prompt	创建 `SYSTEM.md` 或 `AGENTS.md`	第 13-14 章
自定义权限策略	`beforeToolCall` 钩子	第 9 章
改 compaction 策略	Extension hook: `on("session_before_compact", handler)`	第 12 章
加一个新 UI 模式	参考 `modes/rpc/` 实现新 mode	第 26 章
支持新的消息类型	`CustomAgentMessages` 声明合并	第 10 章
加一个新 OAuth provider	`registerOAuthProvider()`	第 7 章
自定义上下文管理	`transformContext` 回调	第 8 章
构建 Slack bot 产品	参考 `packages/mom/` 完整实现	第 28 章

我想做什么	起点文件/函数	参考章节
部署自有模型	参考 `packages/pods/` 的 SSH + vLLM 流程	第 29 章
实现审计日志	订阅 `AgentEvent`，写入日志系统	第 10 章
实现成本预算控制	`transformContext` 中检查累计 token 使用量	第 8 章
实现 A/B 测试（模型对比）	在 `getModel()` 中随机选择模型	第 4 章

F. 术语对照表

英文术语	本书用法	说明
Agent Loop	循环引擎 / agent 循环	`agentLoop()` 函数实现的 LLM 调用 → 工具执行 → 再调用循环
Compaction	上下文压缩	用 LLM 总结对话历史，减少 token 使用
Extension	扩展	运行时代码扩展，可注册工具、命令、事件处理器
Provider	提供方	LLM API 提供方（Anthropic、OpenAI 等）
Skill	技能	纯 markdown 文件，注入 system prompt 引导 agent 行为
Session	会话	一次用户与 agent 的完整交互，持久化为 JSONL
Stream	流式	LLM 的流式响应，通过 SSE 或 WebSocket 传输
Tool Call	工具调用	LLM 请求执行一个工具的指令

英文术语	本书用法	说明
Transform Context	上下文变换	在 LLM 调用前修改消息列表的回调
TUI	终端 UI	Terminal User Interface，基于 ANSI 转义码的终端渲染

版本演化说明

> 本附录基于 pi-mono v0.66.0。类型名、文件路径、函数签名可能随版本更新而变化。 设计模式和二次开发入口的结构性建议预计长期有效。