AI CODING FOR TEAMS

AI Coding 团队化实战

把个人经验变成团队可接管的工程能力



好好用 AI　著

中文电子版 · 第 1 版
2026 年 10 月

AI CODING FOR TEAMS

AI Coding 团队化实战

把个人经验变成团队可接管的工程能力

好好用 AI　著

第 1 版 · 2026-10-24

版本与许可

AI Coding 团队化实战
把个人经验变成团队可接管的工程能力

作者：好好用 AI
公众号：好好用 AI
GitHub：DavidDingXu

电子版版本：1.0.0
修订日期：2026-10-24

原创正文与图示采用 CC BY-NC-SA 4.0，要求署名、非商业使用、相同方式共享。代码采用 PolyForm Noncommercial 1.0.0。第三方材料遵循各自许可。完整范围见仓库许可说明。

在线阅读：https://daviddingxu.github.io/ai-coding-for-teams/
配套仓库：https://github.com/DavidDingXu/ai-coding-for-teams

本版配套源码：738a77f3a573。书中源码链接指向该提交。工具与框架可能变化，复现时请结合对应版本说明。

本书为作者自行编排的电子版。字体使用 Noto CJK，依 SIL Open Font License 1.1 提供；图示、截图与示例的来源和适用范围随正文保留。

前言

团队中最早用好 AI 的人，往往也最了解业务、代码和风险。他知道什么时候该补充上下文，什么时候该停下来核对测试。但这种判断很难仅靠分享工具或复制提示词传给别人。

本书讨论的是这些判断如何进入团队的工程流程。规则需要有归属，需求需要有边界，任务需要留下产物，验证需要能被复查。工具、模型和界面会改变，这些要求仍然成立。

全书从共享运行时和规则体系开始，沿着一次需求走过设计、实现、验证与协作，再回到团队怎样积累经验。书中的目录、状态与数据结构是帮助理解方法的样本。读者可以保留有价值的边界，再用团队已有的工程约定承接它们。

如何阅读本书

本书适合技术负责人、研发效能团队，以及已经有个人 AI Coding 经验、正在开展团队协作的开发者。阅读不要求安装特定工具，也不要求先实现一套完整的 Agent 平台。

第一次阅读建议按五篇顺序推进。若正在准备试点，可先读第 12 章的最小落地包，再回查规则、上下文和验证治理。书中保留原课程编号：00 为序章，01—12 为正文，13 为结语；配套网页和仓库沿用同样的编号。

读到目录模板时，先判断它保存了什么事实、谁负责更新、其他人如何接管，再决定是否采用相同文件名。流程图在书中以静态图呈现，可以离线阅读；目录和命令示例仍保留原始文本。

每篇扉页给出阅读重点。章内图表使用“章号-序号”编号；章末配套资源供需要进一步查阅时使用。网页链接和配套文件不是理解正文的前置条件。

PDF 使用 A4 版式，较密的宽图独立横排，适合固定版式阅读与打印，EPUB 可随设备调整字号。两种版本均包含完整正文和配图，不依赖联网加载图片；外部文档、源码与问题反馈需要联网访问。

目录

前言 4

如何阅读本书 5

第1篇　从个人经验到团队系统 7

序章　个人会用 AI Coding，为什么团队还是用不好？ 8

第1章　共享运行时：不要让每个应用都重复建设 Agent 19

第2章　SPEC 体系：把规则从 Prompt 里拿出来 27

第2篇　从复杂需求到可审查交付 42

第3章　复杂需求拆分：大 PRD 不能直接丢给 Agent 43

第4章　有界上下文：不是给 Agent 更多，而是给得更准 60

第5章　阶段化交付：别让 AI 从一句话直接改代码 69

第6章　Task 化开发：把大 Diff 拆成可审查的工程任务 78

第3篇　从一次成功到持续复用 88

第7章　验证治理：AI 说“测试失败无关”是不够的 89

第8章　资产沉淀：让每次交付反哺下一次任务 98

第4篇　从单 Agent 到团队协作 106

第9章　Agent 分层：不要让一个万能 Agent 包办所有事 107

第10章　多工具适配：把共享语义放 Runtime，把工具差异放 Adapter 114

第11章　协作面：团队如何审查和接管 Agent 产物 122

第5篇　最小试点与收官 132

第12章　团队最小落地包：从一套目录、一条流程、三类规则开始 133

结语　AI Coding 团队化，最后还是工程问题 141

术语索引 149

章节与配套资源 150

PART 01

第1篇　从个人经验到团队系统

先分清什么需要团队共同维护，什么应留在具体应用中。共享运行时承接通用交付流程，SPEC 保存稳定规则，二者共同减少对某个人临场判断的依赖。

阅读重点

能解释运行时、规则和应用之间的职责，并为规则指定维护位置。

序章　个人会用 AI Coding，为什么团队还是用不好?

很多团队第一次看到 AI Coding 的效果，往往来自某个研发同学的个人实践：他把需求贴给模型，让模型帮忙读代码、生成实现、补测试，再自己做几轮修正，最后确实比过去快。这个过程很容易让团队产生一个判断：既然个人已经能提效，那接下来只要多推广工具、多沉淀 Prompt、多做几个 Agent，团队效率自然会起来。

真实情况通常没这么顺。个人会用 AI Coding，依赖的是个人对业务、代码、测试和风险的综合判断。这个人知道哪些上下文该给，知道模型哪里容易胡说，也知道测试失败是不是历史问题。换一个人、换一个应用、换一个需求规模，结果就会波动。团队真正要解决的，不是“让模型更聪明一点”，而是把个人临场判断里可工程化的部分固定下来。

团队 AI Coding 的关键转折点，是从“会用工具”走向“有工程系统承接工具”。需求从哪里来、PRD 是否确认、规则放在哪里、上下文如何选择、设计和计划怎么审查、验证证据如何记录、历史经验怎么沉淀，这些问题如果都靠人临场补救，AI Coding 就只能停留在个人经验层。

这个系列把这类系统称为 **团队 AI Coding 工程化框架**。它不是某个具体产品名，而是一组工程边界：共享运行时、任务状态、稳定产物、规则体系、有界上下文、复杂需求拆分、验证证据、资产沉淀和协作审查。下面用一个通用的实践样本来拆解这些边界如何落到目录、命令、产物和数据结构里。

> 一句话结论：**AI Coding 的团队化问题，不是 Prompt 问题，而是工程系统问题。**

序.1　一个典型场景：个人能跑通，团队接不住

假设一个业务团队要做“新增奖品权益类型”。熟悉该应用的研发 A 用 AI Coding 很顺：

- 他知道 PRD 里“权益类型”和代码里的 `PrizeType` 不是完全一一对应。
- 他知道新增类型不能只改枚举，还要看配置生成、查询展示、回滚兼容。
- 他知道历史测试里有一个用例长期失败，本次不用被它阻塞。
- 他知道哪个 Service 是主入口，哪些 Controller 只是薄封装。

所以他能把上下文裁剪得很准，模型输出也容易被他修正。但如果把同一个需求交给研发 B，或者交给另一个应用团队，问题就会出现：

- B 不知道要补哪些业务规则，只能把整篇 PRD 和一堆代码塞给模型。
- 模型生成的方案看起来完整，但没有说明哪些存量行为不能变。
- 计划按 Controller、Service、DAO 拆任务，看起来很工程化，但没有对应业务验收。

· 测试失败后，模型说“可能是历史失败”，但没有改前 baseline 证据。

· 需求完成后，关键经验只留在聊天记录和 PR 评论里，下次还要重新解释。

这不是某个人不会用 AI，而是团队缺少一个承接层。个人经验里有很多可以系统化的东西：规则可以结构化，需求可以切片，上下文可以选择，验证可以留证据，交付可以归档。团队 AI Coding 的建设重点，就是把这些东西从个人脑子里拿出来，变成团队共享的工程资产。

常见误区

团队推进 AI Coding 时，常见误区有五类。

表 序-1 常见误区

误区	短期看起来	长期问题
每个应用各写一套 Agent	贴近业务，启动快	通用流程重复建设，行为不一致，维护成本随应用数线性上升
规则全部塞进 Prompt	Agent 初期效果明显	Prompt 越来越长，规则过期无人发现，无法审查来源
直接让 AI 改代码	交互简单	缺少 PRD、设计、计划和验证边界，大 diff 难评审
上下文越多越好	AI 看起来知道更多	无关信息淹没关键约束，成本上升，输出不稳定
测试失败靠口头解释	进度不被阻塞	分不清历史失败和本次回归，质量判断不可审计

这些误区背后是同一个问题：团队没有把 AI Coding 纳入工程系统，而是把它当成一个更强的聊天窗口。

一个更容易传播的判断

团队内部讨论 AI Coding 时，可以先用下面这张表判断问题到底出在哪里。

表 序-2 一个更容易传播的判断

现象	表面问题	真正缺的工程能力
某个同学用得很好，其他人复现不了	个人能力差异	规则、上下文和验证经验没有资产化
Agent 每次都要重新解释业务背景	Prompt 不够长	没有可路由的团队规则和应用规则
大需求一交给 AI 就变成大 diff	模型执行太散	没有需求切片、阶段产物和 task contract
测试失败后没人敢信 AI 的解释	模型判断不可靠	没有改前 baseline 和验证 ledger
换工具后历史任务接不上	工具割裂	任务状态和产物没有放在共享运行时

现象	表面问题	真正缺的工程能力
交付结束后经验没有复用	知识库没人维护	缺少 summary、规则处置和 archive recall

这个判断比“要不要多写几个 Agent”更重要。团队化落地的关键，不是 Agent 数量，而是这些工程能力是否形成闭环。

序.2　团队 AI Coding 要补的八类能力

从个人使用走到团队使用，至少要补齐八类能力。

表 序-3　团队 AI Coding 要补的八类能力

能力	解决的问题	工程边界
共享运行时	避免每个应用重复建设 Agent	统一任务状态、工具、产物、上下文目录
状态真相源	避免任务依赖聊天上下文	当前任务指针、任务 JSON、阶段状态
稳定阶段产物	让交付过程可审查	PRD、设计、计划、总结、验证证据
规则体系	避免 Prompt 人工维护失控	Team SPEC、App SPEC、route page、focused Spec
有界上下文	避免全仓扫描和信息过载	PRD working set、code working set、selected rules
复杂需求拆分	避免大 PRD 直接变成大 diff	功能切片、需求契约、必要时分片交付
验证治理	避免测试失败靠口头判断	改前 baseline、改后验证、失败分类、ledger
资产沉淀	让每次交付反哺下一次任务	summary、Spec update、archive recall、research artifact

这些能力不是一次性做全的平台功能，而是一套可以逐步建设的工程边界。先把状态、产物、规则、验证这几件事固化下来，团队 AI Coding 才有持续演进的基础。

这里有一个很实用的顺序：先解决“人能不能接手”，再解决“AI 能不能自动”。如果一个任务换人之后连当前状态、需求、设计、计划和验证证据都找不到，过早做自动化只会把不可控流程跑得更快。

序.3　一个工程化框架的整体结构

团队 AI Coding 工程化框架的核心做法，是把 AI Coding 组织成一套项目内共享运行时。运行时承接通用流程，应用差异进入应用规则和历史案例，不同 AI 工具通过适配层接入。

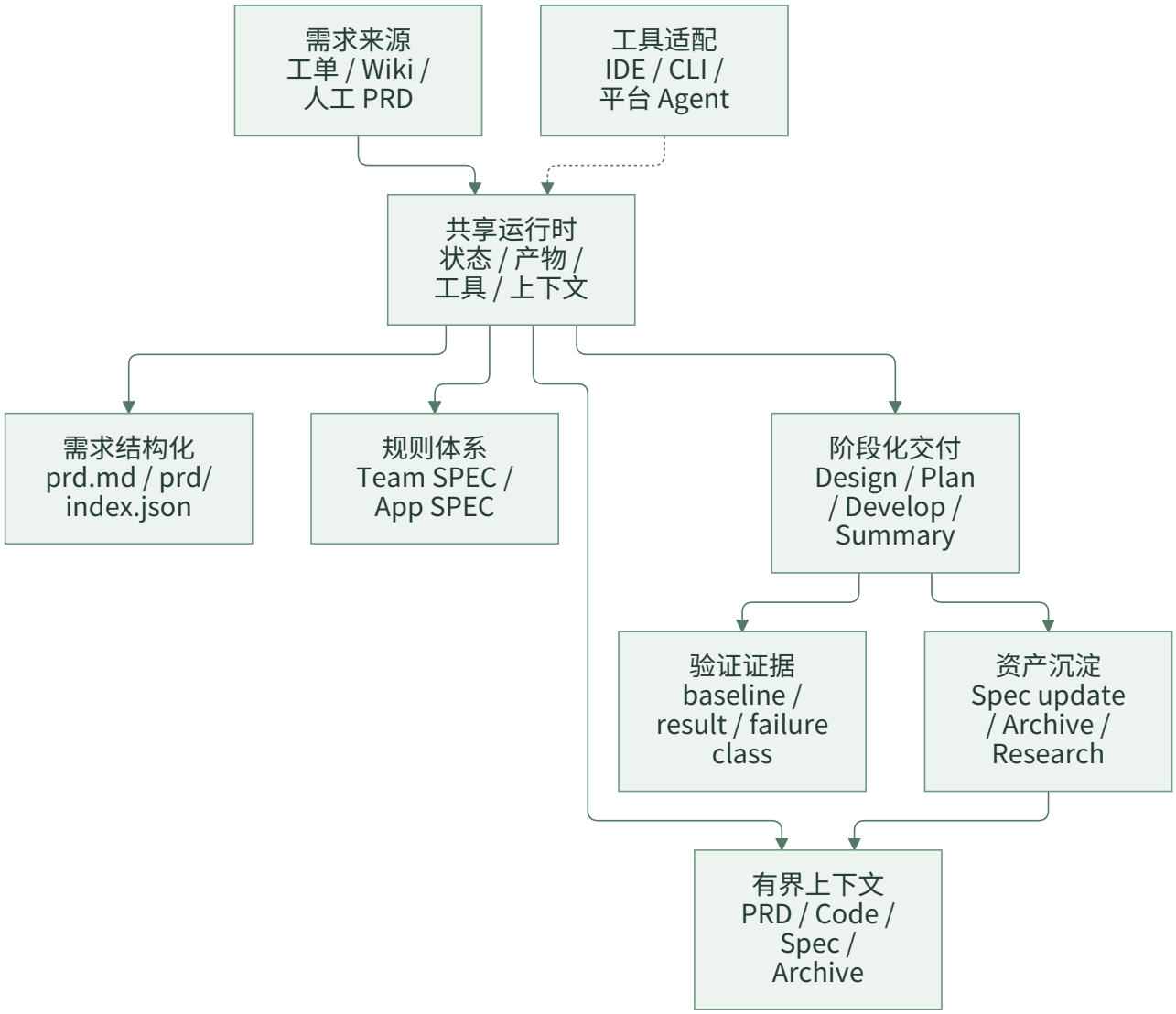


图 序-1　一个工程化框架的整体结构

对应到目录，一个通用实践样本可以这样组织：

示例 序-1　一个工程化框架的整体结构

```
.team-agent/
├── .current-task
├── config.json
├── workflow/
│   └── phase-contracts.json
├── tools/
│   ├── task.js
│   ├── build-prd.js
│   ├── prd-context.js
│   ├── code-index.js
│   ├── context-selection.js
│   └── spec-context.js
├── spec/
│   ├── team/
│   └── app/
```

```
├── code-index/
│   └── index.json
└── tasks/
    ├── {taskId}/
    │   ├── task.json
    │   ├── prd.md
    │   ├── design.md
    │   ├── plan.md
    │   ├── plan.tasks.json
    │   ├── summary.md
    │   ├── generated/context/
    │   ├── evidence/verification-ledger.jsonl
    │   └── context/archive-recall.json
    └── archive/
        └── index.jsonl
```

这套结构里有几个关键分工：

- `.current-task` 和 `task.json` 承接任务状态。
- `prd.md`、`design.md`、`plan.md`、`summary.md` 是稳定阶段产物。
- `generated/context/*.jsonl` 是可重建上下文，不是新的状态真相源。
- `spec/**` 承接团队和应用规则。
- `evidence/verification-ledger.jsonl` 承接验证证据。
- `archive/index.jsonl` 承接已完成任务的历史案例索引。

这些名字都不是重点。重点是把“状态、产物、规则、上下文、证据、归档”分开。只要边界清楚，目录名可以换成 `.ai-agent/`、`.coding-agent/` 或团队自己的平台目录。

这套框架背后的几个理论关键词

这套体系不是凭空设计出来的，它背后其实是几条很朴素的软件工程原则。

表 序-4　这套框架背后的几个理论关键词

理论关键词	在 AI Coding 团队化里的含义	落地动作
Single Source of Truth	任务状态不能散落在聊天、脚本和 UI 里	用 `task.json` 和当前任务指针承接状态
Bounded Context	Agent 每次只消费当前阶段必要证据	用 PRD 切片、规则选择、代码 working set 控制上下文
Artifact-first	关键判断不能只存在聊天里	PRD、设计、计划、总结先落文档再确认
Evidence-based Review	质量判断要基于证据链	改前 baseline、改后验证、ledger 和 staged diff review

理论关键词	在 AI Coding 团队化里的含义	落地动作
Control Plane / Execution Plane	调度和执行不能混在一个万能 Agent 里	控制面推进流程，worker 执行单个 task

这些词听起来像架构概念，但落到团队里就是很具体的问题：任务从哪里恢复、规则从哪里读、失败怎么分类、谁有权推进下一阶段。

序.4 状态真相源：任务不能依赖聊天记忆

个人使用 AI Coding 时，经常靠当前聊天窗口维持上下文。团队场景里，这很危险：会话断了、工具换了、人换了，任务就很难恢复。

一个最小 `task.json` 可以长这样：

清单 序-2 状态真相源：任务不能依赖聊天记忆

```
{
  "taskId": "REQ-123456",
  "workflowKind": "feature",
  "currentStepId": "tech_design",
  "status": "in_progress",
  "branch": "feature/prize-type",
  "artifacts": {
    "prd": "prd.md",
    "design": "design.md",
    "plan": "plan.md",
    "summary": "summary.md"
  },
  "meta": {
    "contextMode": "expanded",
    "contextRisks": ["large_prd", "multi_domain"]
  }
}
```

这个 JSON 不需要承载所有过程细节，它只保存任务恢复和流程推进必须知道的事实。上下文选择、代码索引、验证证据这类可重建或可追加的信息，应放在自己的产物里，不要都塞回任务状态。

一个健康的状态模型要满足三个要求：

1. **可恢复**：换一个会话，能知道当前任务是谁、走到哪一步、下一步该读哪些产物。
2. **可审查**：任务状态里不能混入大量模型总结，关键事实要能追溯到具体 artifact。
3. **可清理**：任务归档后，active task 指针要清掉，临时诊断文件不能变成长期状态。

一个不健康的状态模型通常长这样：`task.json` 里塞满模型总结、测试日志、候选代码、历史案例和临时判断。短期看信息很全，长期看没人知道哪个字段是事实、哪个字段是缓存、哪个字段已经过期。团队要尽早建立“状态只放状态，证据放证据，缓存可重建”的习惯。

从聊天驱动到产物驱动

个人使用 AI Coding 时，常见交互是：“帮我看下这个需求”“帮我实现一下”“测试失败你看下”。这种方式对熟悉上下文的人有效，但团队协作时很难交接和审查。

团队级流程需要把关键节点产物化：

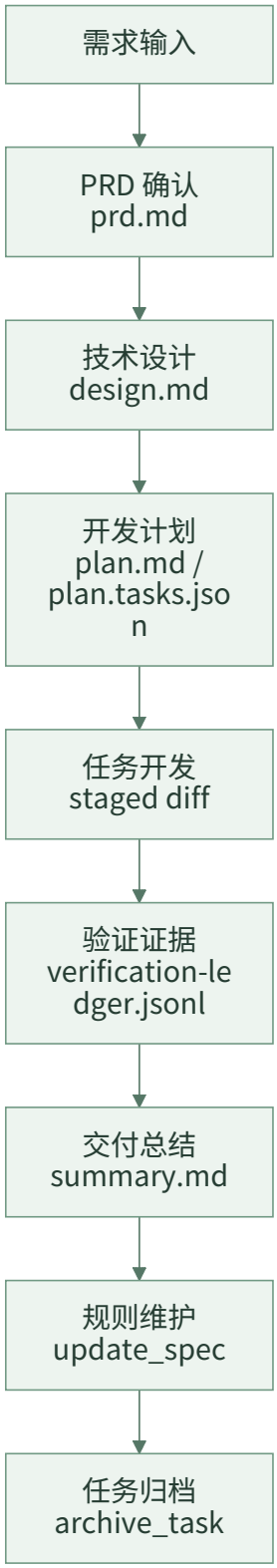


图 序-2 从聊天驱动到产物驱动

这个流程并不是为了增加仪式感，而是为了回答团队协作中的具体问题：

- 需求是否被确认过?
- 技术方案是否能被负责人审查?
- 开发计划是否拆到可执行 task?
- 每个 task 是否有验证命令和通过证据?
- 失败测试是否是历史问题?
- 这次交付有没有形成后续可用的规则或案例?

如果这些问题都只能从聊天记录里找答案，AI Coding 很难成为团队能力。

什么叫“足够干”的最小标准

如果一篇实践文章读完，团队只能得到“要重视上下文”“要做好验证”这类原则，它还不够。真正能落地的内容至少要能回答下面几个问题：

表 序-5　什么叫“足够干”的最小标准

问题	文章里应该给出的东西
我从哪里开始建?	最小目录和文件职责
我怎么判断做对了?	检查清单和阻塞条件
我怎么避免做重?	先后顺序和低配版路径
我怎么说服团队?	真实失控场景和可讨论问题
我怎么持续演进?	交付后规则和案例沉淀机制

这个系列后面的每一篇都会围绕这些具体答案展开，而不是停留在“理念正确”。

一个需求在框架里怎么流转

以“新增奖品权益类型”为例，完整流转可以拆成七个阶段。

表 序-6　一个需求在框架里怎么流转

阶段	输入	输出	人需要关注什么
需求结构化	工单、Wiki、人工 PRD	`prd.md`、`prd/index.json`、`prd/feature-*.md`	需求范围是否清楚，功能点是否漏拆
规则选择	PRD 切片、SPEC route page	selected SPEC refs	是否命中业务规则，是否有设计前必须确认的问题

阶段	输入	输出	人需要关注什么
技术设计	PRD、规则、代码 working set	`design.md`	方案是否覆盖核心链路、兼容、数据、稳定性
开发计划	`design.md`、代码引用	`plan.md`、`plan.tasks.json`	task 是否可执行，是否有验证命令和验收标准
任务开发	单个 task、allowed paths	staged diff、验证记录	diff 是否只覆盖当前 task，是否满足验收
交付总结	diff、验证证据、计划完成情况	`summary.md`	这次改动的边界、风险和后续事项
资产沉淀	summary、历史问题、规则变化	SPEC 更新、archive index	哪些经验下次还要用

这套流转的目的不是让 AI 每一步都自动化，而是让人和 AI 围绕同一批产物协作。AI 负责生成、搜索、初步修改和整理，人负责关键判断、边界确认和风险兜底。

序.5　低配版也能先跑起来

很多团队会担心：如果要做 code-index、selector、archive recall、workspace，听起来是不是太重。实际落地时可以从低配版开始。

示例 序-3　低配版也能先跑起来

```
.team-agent/
├── current-task
├── spec/
│   ├── team/
│   └── app/
├── tasks/
│   └── {taskId}/
│       ├── task.json
│       ├── prd.md
│       ├── design.md
│       ├── plan.md
│       ├── plan.tasks.json
│       ├── summary.md
│       ├── context/
│       └── evidence/
└── archive/
    └── index.jsonl
```

第一阶段只要求做到：

1. 每个 AI Coding 任务都有任务目录。
2. 需求、设计、计划、总结落成 Markdown。
3. 团队规则和应用规则不要写死在 Prompt 中。

4. 每个 task 都记录验证命令和结果。
5. 交付结束后判断是否需要更新规则或归档案例。

这五件事做好后，再考虑自动选择上下文、自动召回历史案例、多工具适配和协作工作台。不要一开始就把系统做成大平台，否则团队还没有形成稳定习惯，维护成本已经先上来了。

建设顺序建议

一个比较稳的建设顺序是：

1. **先产物化**：固定 `prd.md`、`design.md`、`plan.md`、`summary.md`，让团队从聊天驱动切到产物驱动。
2. **再规则化**：把 Prompt 里的长期规则拆到 `spec/team` 和 `spec/app`，建立 route page 和 focused Spec。
3. **再上下文化**：让 PRD 切片、SPEC、代码引用和历史案例进入 working set，而不是全量塞给模型。
4. **再证据化**：建立改前 baseline、改后验证、失败分类和 ledger。
5. **最后协作化**：当产物和证据稳定后，再做 Feature Room、Review Request、指标看板等协作面。

这条路径的好处是每一步都有独立收益。即使团队暂时没有做复杂 Agent 编排，只要先把产物、规则和验证固定下来，AI Coding 的稳定性就会明显提升。

序.6 自查清单

可以用下面的问题判断团队是否还停留在个人使用阶段：

- 换一个人能否继续同一个 AI Coding 任务?
- 需求、设计、计划、验证证据是否有稳定文件?
- 应用规则是否主要写在 Prompt 或聊天模板里?
- 相似需求的历史经验能否被后续任务重新找到?
- AI 修改代码前是否知道当前测试 baseline?
- 一个大需求是否会先拆成功能点边界?
- 负责人审查的是 artifact 和 diff，还是只能看聊天记录?
- 更换 AI 工具时，任务状态和产物是否还能保留?

如果多数答案是否定的，优先建设共享运行时、SPEC 体系和阶段产物，而不是继续增加单点 Agent 数量。

序.7 思考与实践

1. 你们团队现在 AI Coding 最大的问题，是个人经验难复用、规则难维护、上下文失控，还是验证不清?

2. 如果让另一个同事明天接手你当前的 AI Coding 任务，他需要哪些文件才能继续?
3. 你们团队最应该先补的是任务状态、规则体系、上下文工程，还是验证证据？欢迎带着具体场景讨论。

序.8　结尾

团队 AI Coding 的核心挑战，不是让每个人都学会更复杂的提示词，而是让团队有一套稳定工程系统承接 AI 的输入、输出和沉淀。

个人提效靠技巧，团队稳定靠系统。下一篇进入共享运行时：为什么不要让每个应用都重复建设 Agent，以及团队最小任务目录应该如何承接状态、产物和验证证据。

第1章　共享运行时：不要让每个应用都重复建设 Agent

很多团队推进 AI Coding 时，会自然地从应用出发：订单应用写一套 Agent，营销应用写一套 Agent，结算应用再写一套 Agent。每套 Agent 都有自己的 Prompt、Command、Skill、规则文档和任务目录。短期看，这种方式贴近业务，启动也快；长期看，它会把团队拖进重复建设。

真正重复的部分并不是业务知识，而是需求进入、任务状态、阶段产物、上下文组织、验证证据和任务归档这些基础能力。如果每个应用都重新实现一遍，团队会很快遇到行为不一致、质量口径不统一、历史经验难复用、工具升级成本高的问题。

共享运行时的目标，是把通用交付能力放到一套稳定边界里。应用差异通过规则、案例和少量定制命令扩展，而不是每个应用都重写一套 Agent 系统。

> 一句话结论：**应用差异应该资产化，通用流程应该运行时化。**

常见误区

表 1-1　常见误区

误区	表现	后果
按应用建设完整 Agent	每个应用都有自己的流程、状态、规则和验证	通用能力重复实现，行为难统一
把运行时放进某个工具	任务状态只存在某个 IDE 插件或聊天工具里	换工具、换会话、换人后难恢复
只沉淀 Prompt，不沉淀产物	Prompt 很强，但 PRD、设计、计划、验证都散落	交付过程不可审查
没有状态真相源	当前阶段靠聊天记忆判断	恢复、暂停、变更、归档都不可靠
运行时目录无限膨胀	每个能力都新增顶层文件	后期维护和归档成本上升

共享运行时要解决的是“通用交付能力应该放在哪里、由谁维护、如何被不同工具复用”。

1.1　为什么这件事容易被低估

很多团队一开始会觉得“先让每个应用自己跑起来就行”。这个判断在试点期没问题，但一旦进入团队规模化，重复建设会快速放大。

表 1-2 为什么这件事容易被低估

应用数	如果每个应用自建 Agent	最容易出现的问题
1 个	看起来很灵活	规则和流程还没暴露复用问题
3 个	每个应用开始有自己的 Prompt 和脚本	同一条工程规范出现多个版本
10 个	工具升级、规则变更、流程调整都要多处同步	团队质量口径失控
30 个以上	平台团队变成 Prompt 和脚本维护团队	新应用接入成本越来越高

共享运行时不是为了“平台化好看”，而是为了避免团队把通用交付能力重复实现几十遍。

设计原则

第一，**任务状态独立于 AI 工具**。Claude、OpenCode、Codex 或自研 IDE 插件都可以接入，但任务真相源不应该只存在某个工具的会话中。

第二，**稳定产物和派生产物分层**。PRD、设计、计划、总结是稳定产物；代码索引、上下文 JSONL、selector cache 是可重建产物。两类文件的 owner 和生命周期不能混在一起。

第三，**应用扩展不能破坏运行时边界**。应用可以有自己的 SPEC、Skill、Command，但不应该重新定义任务状态、阶段流程和验证证据格式。

1.2 案例拆解：一个共享运行时长什么样

这类框架通常会把共享运行时放在项目内 `.team-agent/` 目录。这个目录不属于某一个 Agent，也不属于某一个 AI 工具，而是所有适配层共同消费的任务工作区。

示例 1-1 案例拆解：一个共享运行时长什么样

```
.team-agent/
├── .current-task
├── config.json
├── workflow/
│   └── phase-contracts.json
├── tools/
│   ├── task.js
│   ├── build-prd.js
│   ├── code-index.js
│   ├── context-selection.js
│   └── spec-context.js
├── spec/
│   ├── team/
│   └── app/
├── code-index/
│   └── index.json
```

```
└── tasks/
    ├── {taskId}/
    │   ├── task.json
    │   ├── prd.md
    │   ├── design.md
    │   ├── plan.md
    │   ├── plan.tasks.json
    │   ├── summary.md
    │   ├── generated/context/
    │   ├── evidence/verification-ledger.jsonl
    │   └── context/archive-recall.json
    └── archive/
        └── index.jsonl
```

这套目录里有几类边界：

表 1-3 案例拆解：一个共享运行时长什么样

层级	代表文件	作用
任务指针	`.team-agent/.current-task`	指向当前 active task
状态真相源	`tasks/{taskId}/task.json`	保存当前阶段、任务元信息、恢复所需状态
稳定产物	`prd.md`、`design.md`、`plan.md`、`summary.md`	给人和 Agent 审查的交付文档
派生产物	`generated/context/*.jsonl`、`code-index/index.json`	可由 builder 重建的上下文材料
执行证据	`evidence/verification-ledger.jsonl`	记录验证命令、结果、失败分类
规则资产	`spec/team/**`、`spec/app/**`	团队规则和应用差异
历史资产	`tasks/archive/index.jsonl`	已完成任务的案例索引

一个最小任务状态可以这样表达：

清单 1-2 案例拆解：一个共享运行时长什么样

```
{
  "taskId": "123456",
  "workflowKind": "feature",
  "status": "in_progress",
  "currentStepId": "plan_delivery",
  "branch": "feature/prize-type",
  "steps": [
    { "id": "tech_design", "status": "completed" },
    { "id": "plan_delivery", "status": "in_progress" },
```

```
    { "id": "develop_feature", "status": "pending" }
  ],
  "meta": {
    "contextMode": "expanded",
    "contextRisks": ["large_prd"]
  }
}
```

这个状态只保存流程恢复需要的信息。它不保存完整 PRD，不保存代码索引，不保存所有规则正文，也不保存测试日志全文。这些内容属于对应 artifact。

文件 owner 要先定清楚

共享运行时最容易膨胀的原因，是每个人都觉得自己可以往任务目录里加文件。建议一开始就把 owner 和生命周期写清楚。

表 1-4 文件 owner 要先定清楚

文件或目录	Owner	生命周期	规则
`.current-task`	控制面	active task 期间	只保存当前任务 ID，归档后清理
`task.json`	控制面	active + archive	只保存流程状态和恢复必要事实
`prd.md` / `design.md` / `plan.md` / `summary.md`	对应阶段 Agent 或负责人	稳定产物	人可以审查，不能被 builder 随意覆盖
`generated/context/*`	context builder	可重建	缺失或过期时重建，不作为真相源
`evidence/*`	worker / verifier	追加型证据	不覆盖历史记录，失败也要记录
`spec/**`	团队或应用 owner	长期资产	需要来源、适用范围和维护入口
`archive/index.jsonl`	归档流程	长期索引	用于后续弱召回，不替代当前需求

这个表越早建立，后续越少出现“某个脚本顺手改了状态”“某个 Agent 覆盖了 PRD”“缓存文件被当成事实”的问题。

1.3 多工具为什么能共享运行时

共享运行时的一个重要价值，是把工具差异放到 adapter，而不是放到任务状态里。

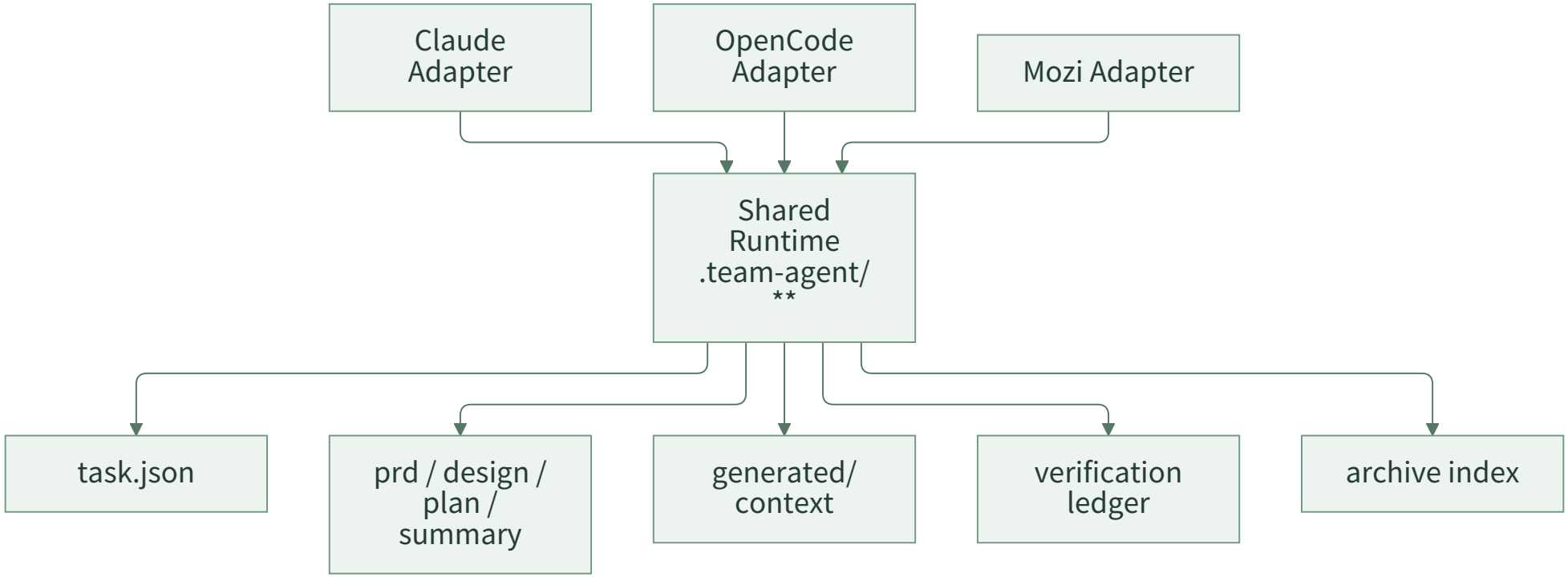


图 1-1　多工具为什么能共享运行时

在这个结构下，不同工具可以有自己的命令格式、hook 机制和上下文注入方式，但它们不重新定义任务目录。这样做有两个好处：

- 换工具时，任务状态和历史产物还能保留。
- 多个入口协作时，不会出现“这个工具认为任务完成，另一个工具不知道”的状态分裂。

一个判断方法是：如果明天把当前 AI 工具换掉，`tasks/{taskId}/` 里的 `task.json`、`prd.md`、`design.md`、`plan.md`、`summary.md` 和验证证据还能不能被新工具继续消费？如果不能，说明任务状态被工具私有实现绑住了。

运行时要避免什么

共享运行时不是把所有东西都塞进一个目录。它要控制文件预算和 owner。

表 1-5 运行时要避免什么

不建议	原因	更合适的做法
每个阶段都新增顶层状态文件	状态来源变多，恢复逻辑复杂	统一写入 `task.json` 或阶段 artifact
把 selector 结果写入任务状态	可重建上下文污染真相源	写入 cache 或 generated context
把测试日志全文写入 `task.json`	状态文件膨胀	写入 verification ledger 或 evidence 文件
把历史案例直接注入会话	上下文不可控	作为弱引用，在设计/计划前按预算召回
应用自定义流程覆盖主流程	团队质量口径不一致	应用通过 SPEC 和专项 Command 扩展

共享运行时越稳定，应用扩展越容易。反过来，如果运行时边界混乱，后续每个能力都会变成补丁。

运行时成熟度分级

团队可以用下面的分级判断自己该做到哪一步。

表 1-6 运行时成熟度分级

级别	运行时能力	适合阶段
L0 聊天驱动	只有聊天记录和代码 diff	个人试用
L1 产物驱动	每个任务有 PRD、设计、计划、总结	小团队试点
L2 状态驱动	有 `task.json`、当前任务指针、阶段状态	多人接力
L3 上下文驱动	有 PRD/code working set、规则选择、验证 ledger	多应用推广

级别	运行时能力	适合阶段
L4 协作驱动	有 review request、Feature Room、归档召回和指标	团队规模化

不要跳级。L1 没做好就做 L4，最后会变成一个展示漂亮但产物不可审查的面板。

1.4 团队参考做法

其他团队可以先定义一个最小运行时目录：

示例 1-3 团队参考做法

```
.team-agent/
├── current-task
├── tools/
├── workflow/
├── spec/
│   ├── team/
│   └── app/
├── tasks/
│   └── {taskId}/
│       ├── task.json
│       ├── prd.md
│       ├── design.md
│       ├── plan.md
│       ├── plan.tasks.json
│       ├── summary.md
│       ├── context/
│       └── evidence/
└── archive/
    └── index.jsonl
```

第一阶段只做四个约束：

1. 所有任务必须进入统一任务目录。
2. 当前任务只通过 `current-task` 指向。
3. 需求、设计、计划、总结必须是稳定 artifact。
4. 上下文和验证证据不能只留在聊天里。

第二阶段再引入 code-index、context selector、archive recall、多工具 adapter 和 Web 协作工作台。不要一开始就做大平台。共享运行时最早要解决的是状态和产物，而不是复杂 UI。

1.5 检查清单

- 团队是否有统一任务目录？
- 当前任务是否能脱离聊天记录恢复？

- PRD、设计、计划、总结是否有稳定文件?
- 派生产物是否可以重建，而不是写死在任务状态里?
- 不同 AI 工具是否共享同一套任务状态?
- 应用差异是否通过 SPEC 扩展，而不是重写流程?
- 任务完成后是否进入 archive?
- 运行时目录是否有文件预算和 owner 规则?

1.6 思考与实践

1. 你们团队现在 AI Coding 的任务状态在哪里？聊天记录、Issue、PR，还是统一任务目录?
2. 如果新接入一个应用，你们会复制一套 Agent，还是复用同一套运行时后只补应用规则?
3. 你们最小可接受的 `.team-agent/` 目录应该包含哪些文件?

1.7 结尾

共享运行时是团队 AI Coding 的底座。没有它，Agent 越多，状态越散，经验越难沉淀。先把任务状态、阶段产物、规则、上下文和验证证据放到同一个工程边界里，后续的 SPEC、上下文选择、验证治理和协作面才有承载点。

下一篇进入 SPEC 体系：为什么团队规则不应该继续堆在 Prompt 里，以及如何让规则变成可检索、可审查、可维护的工程资产。

第2章 SPEC 体系：把规则从 Prompt 里拿出来

很多团队刚开始做 AI Coding Agent 时，会把大部分规则写进 Prompt：代码风格、目录结构、日志要求、接口规范、业务规则、测试要求、上线注意事项都往里塞。这样做初期见效很快，因为 Agent 确实能在一次会话里“知道更多”。但随着应用增多、规则变多、业务变化变快，Prompt 很快会变成一个没人敢改、没人能审、也没人知道是否过期的大文件。

更麻烦的是，Prompt 往往会混在两类完全不同的内容：一类是执行流程，例如“先读 PRD，再写 design.md，再输出 JSON”；另一类是业务和工程知识，例如“新增奖品类型必须覆盖创建、查询、展示和回滚路径”。前者属于 Agent 行为契约，应该跟着工具和流程演进；后者属于团队知识资产，应该能被检索、审查、维护和复用。两类内容混在一起，后期一定会失控。

团队级 AI Coding 需要把长期规则从 Prompt 中剥离出来，形成可被 Agent 消费的 SPEC 体系。这里的 SPEC 可以理解为“面向 Agent 的项目知识库”，但它不是普通 Wiki 的搬运。一个可用的 SPEC 体系要有目录、路由、适用阶段、注入策略、来源说明和维护流程。

> 一句话结论：**Prompt 适合承接流程，SPEC 才适合承接团队知识。**

2.1 一个真实问题：Prompt 为什么会越写越乱

假设团队最初只有一个应用，Prompt 里写了这些规则：

- 代码必须遵循三层架构。
- Controller 不能写业务逻辑。
- 新增接口要补日志。
- 新增奖品类型要兼容存量类型。
- 退款需求要确认是否影响补偿链路。
- 单测失败要先判断是否历史失败。

这时看起来还可以。但当第二个应用接入时，问题出现了：第二个应用没有奖品类型，却有账户冻结和履约规则；第三个应用使用不同目录结构，日志规范也不一样。团队很容易走向两种做法：

1. 把所有规则继续塞进一个总 Prompt。
2. 每个应用复制一份 Prompt，然后各自修改。

第一种做法会让上下文越来越重，Agent 每次都读大量不相关规则。第二种做法会让同一条通用规则被复制到多个地方，后期维护时必然遗漏。更糟的是，交付过程中发现的新规则，经常只留在聊天记录、PR 评论或某个研发的脑子里，下一次需求又要重新解释。

SPEC 体系要解决的不是“文档不够多”，而是“团队知识没有工程边界”。

常见误区

表 2-1 常见误区

误区	表现	后果
把规则都写进 Prompt	Prompt 越写越长，所有应用共享一份大规则	应用差异无法表达，规则过期后难发现
每个应用维护一套 Agent Prompt	A 应用一套，B 应用一套，C 应用一套	通用规则重复维护，行为不一致
把知识库当文档仓库	Wiki 很全，但 Agent 不知道该读哪篇	检索和注入不可控，关键规则容易漏
把所有规则都强制注入	每次任务都加载全量规范	上下文预算浪费，不相关规则干扰判断
交付后不更新规则	踩坑经验留在聊天和 PR 里	下一次类似需求还要重新解释

规则体系的关键不是“文档更多”，而是让规则具备可路由、可选择、可验证、可演进的结构。

2.2 判断一条规则该不该进 SPEC

不是所有信息都应该沉淀成规则。很多团队的规则库失控，不是因为规则太少，而是因为把临时事实、一次性解释和长期规则混在一起。

可以用下面这张表判断：

表 2-2 判断一条规则该不该进 SPEC

内容	是否进 SPEC	更合适的位置
“新增权益类型必须覆盖创建、查询、展示路径”	是	`spec/app/business/`
“本次需求不做管理端入口”	否	当前 `prd.md` 或 `design.md`
“这个应用使用 Maven，单测命令是 xxx”	是	`spec/app/engineering/` 或 `profile.md`
“这次测试环境 Redis 不稳定”	否	`verification-ledger` 或 `summary.md`
“接口字段 A 是历史兼容字段，不能删除”	是	`spec/app/domain/` 或 `business/`
“某次实现选择了方案 B，因为工期短”	不一定	更适合 archive case

一个简单标准是：**下一次类似需求开始前，是否希望 Agent 自动看到这条信息？** 如果是，就考虑进入 SPEC；如果只是本次任务事实，就不要污染长期规则。

设计原则

SPEC 体系设计可以遵循三条原则。

第一，**流程规则和知识规则分离**。Agent Prompt 负责执行机制，例如如何调用工具、如何输出 JSON、什么时候写 artifact；SPEC 负责可复用知识，例如业务不变量、工程规范、接口约束、测试要求。

第二，**团队规则和应用规则分层**。团队通用规则只维护一份，应用差异放到应用自己的 SPEC。不要因为某个应用有特殊业务，就复制一套完整 Agent。

第三，**规则按任务语义选择，不全量注入**。默认只注入路由页和必要硬规则，其他规则通过当前 PRD、代码上下文、阶段和 Agent 职责选择进入 working set。

2.3 SPEC 目录怎么设计

一个通用的 SPEC 目录可以分成两层：`team/` 是团队基线规则，`app/` 是具体业务应用的规则。

示例 2-1 SPEC 目录怎么设计

```
.team-agent/spec/
├── team/
│   ├── index.md
│   ├── requirement.md
│   ├── architecture.md
│   ├── implementation-java.md
│   ├── middleware.md
│   ├── verification.md
│   └── delivery.md
└── app/
    ├── profile.md
    ├── index.md
    ├── .initialized.json
    ├── domain/
    │   └── index.md
    ├── business/
    │   └── index.md
    └── engineering/
        └── index.md
```

这套目录有几个明确边界：

- `spec/team/**` 承接团队基线规则，例如需求、设计、Java 实现、验证、交付。
- `spec/app/profile.md` 描述当前应用边界，例如应用名、技术栈、主要模块、测试入口。
- `spec/app/domain/**` 承接领域知识，例如订单、奖品、账户、履约等模型和不变量。

- `spec/app/business/**` 承接可复用业务需求开发规则，例如某类业务变更的流程、接口约束、设计前确认问题。
- `spec/app/engineering/**` 承接应用工程规范，例如分层目录、日志、异常、配置、测试、中间件接入。

应用级顶层目录建议保持封闭：只允许 `profile.md`、`index.md`、`.initialized.json`、`domain/`、`business/`、`engineering/`。封闭不是为了限制表达，而是为了防止知识目录越长越散，最后人和 Agent 都不知道规则应该放在哪里。

Route Page 和 Focused Spec

SPEC 体系里有两类文件。

第一类是 route page，也就是路由页。它的职责是告诉 Agent 和人“有哪些规则，什么时候去读哪一篇”。`index.md` 属于 route page，不带 frontmatter，不承接大段规则正文。

一个业务规则路由页可以长这样：

清单 2-2 Route Page 和 Focused Spec

```
# Business Spec Index

## Overview

业务规则目录承接可复用业务需求开发规则、验收约束和设计前确认问题。

## Available Specs

| Spec | When To Use | Path |
|---|---|---|
| Prize Type Development | 涉及奖品类型、权益类型、奖品配置扩展 | `.team-agent/spec/app/business/prize-type.md` |
| Order Refund Policy | 涉及订单退款、撤销、补偿规则 | `.team-agent/spec/app/business/order-refund.md` |

## Quick Triggers

- 奖品类型、权益类型、配置生成、权益展示
- 退款策略、补偿策略、履约回滚

## Pre-Development Checklist

- 当前需求是否命中已有业务规则?
- 是否存在设计前必须确认的问题?

## Quality Check

- 路由页只做导航，不重复 focused Spec 正文。
```

第二类是 focused Spec，也就是聚焦规则。它承接某一个具体主题的规则、示例、DO / DON'T、Checklist 和来源。

清单 2-3 Route Page 和 Focused Spec

```
---
mode: auto
phases: tech_design,plan_delivery,develop_feature
agents: design-agent,plan-agent,task-worker
---

# Prize Type Development

> Purpose: 说明奖品类型扩展在当前应用中如何设计、计划和实现。

## Overview

奖品类型扩展涉及配置、识别、查询展示和兼容逻辑。新增类型时需要确认是否影响存量奖品、是否需要新增权益枚举、是否需要补充管理端配置。

## How To Apply

- `tech_design` 阶段先确认新增类型是否改变既有状态机。
- `plan_delivery` 阶段拆分配置、领域模型、接口展示和测试任务。
- `develop_feature` 阶段优先补充枚举映射、配置解析和回归测试。

## Rules

| Rule | Reason |
|---|---|
| 新增奖品类型必须有存量兼容策略 | 防止旧类型查询或展示行为变化 |
| 新增类型必须覆盖创建、查询、展示和回滚路径 | 避免只实现写入不实现读取 |

## DO / DON'T

DO:

- 先确认业务枚举和数据库存储值的对应关系。
- 为存量类型补回归测试。

DON'T:

- 不要只在接口层新增枚举，忽略配置生成和展示路径。
- 不要把一次性需求说明写成长期规则。

## Checklist

- [ ] 是否确认新增类型的业务含义?
- [ ] 是否确认存量类型行为不变?
- [ ] 是否补充创建、查询、展示测试?

## Source Reference

- 来源：历史需求总结、代码评审结论、业务方确认。
```

这类文件带三个 metadata：

表 2-3 Route Page 和 Focused Spec

字段	含义
`mode`	注入模式，`auto` 表示语义命中后注入，`required` 表示匹配阶段强制注入
`phases`	适用阶段，例如 `tech_design`、`plan_delivery`、`develop_feature`
`agents`	适用 Agent，例如 designer、planner、task worker、reviewer

`mode: required` 要谨慎使用，只适合短硬规则，并且规则正文应该放在 `## Runtime Rules` 下。大段业务知识、完整模块地图、通用背景说明都不应该强制注入。

Required 和 Auto 怎么选

很多团队会把所有重要规则都标成 required，结果每次任务上下文都很重。更稳的做法是把“必须无条件遵守的短规则”和“命中语义后才展开的知识”分开。

表 2-4 Required 和 Auto 怎么选

类型	适合内容	不适合内容	示例
`required`	短硬规则、流程禁令、安全红线	大段业务背景、模块地图、案例说明	“不得跳过 baseline 验证”
`auto`	业务规则、领域知识、工程规范、案例化约束	每次都必须执行的流程动作	“新增奖品类型的设计和验证要求”

`required` 的正文最好只保留在 `## Runtime Rules` 下，几条就够。`auto` 才是大多数 focused Spec 的默认形态。这样 Agent 每次都能看到必要红线，但不会被大量不相关规则淹没。

2.4 规则如何进入 Agent 上下文

SPEC 不是全量写进 Prompt，而是经过选择后进入上下文。

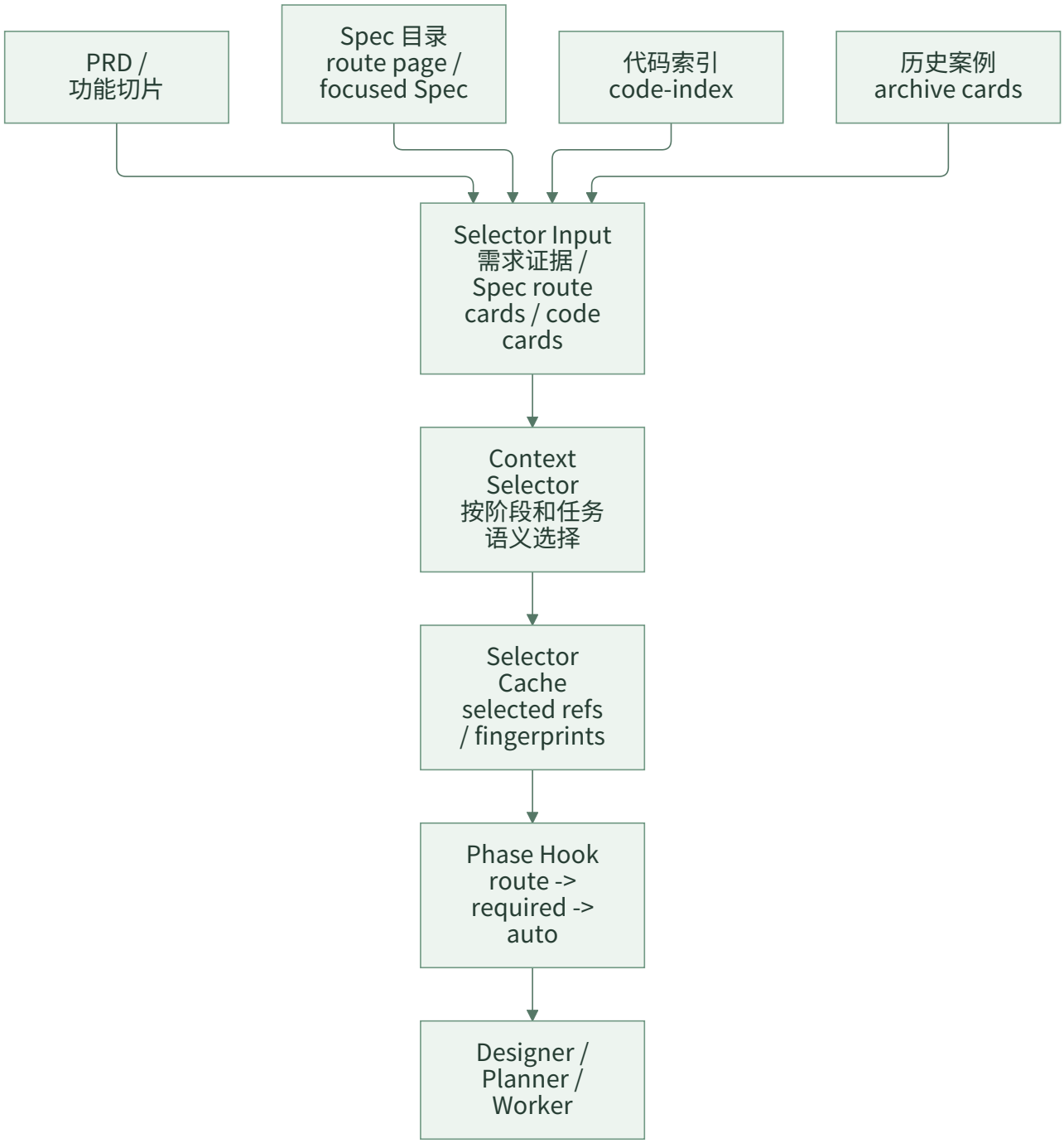


图 2-1 规则如何进入 Agent 上下文

在一个通用实现里，这条链路可以由几类工具和文件承接：

```
runtime/templates/tools/spec-context.js
runtime/templates/tools/context-selection.js
runtime/templates/tools/task.js
.team-agent/cache/tasks/{taskId}/context-selection.json
```

一个简化后的 selector cache 可以长这样：

清单 2-4 规则如何进入 Agent 上下文

```
{
  "phase": "tech_design",
  "agent": "design-agent",
  "selectedSpecRefs": [
    ".team-agent/spec/app/business/prize-type.md",
    ".team-agent/spec/app/engineering/api-contract.md"
  ],
  "recommendedBusinessSpecs": [
    ".team-agent/spec/app/business/prize-type.md"
  ],
  "confirmedBusinessSpecs": [
    ".team-agent/spec/app/business/prize-type.md"
  ],
  "fingerprints": {
    "selectorInput": "sha256:input",
    "selectorOutput": "sha256:output"
  }
}
```

这里的重点不是字段名，而是选择结果必须可追溯。Agent 在设计阶段读了哪些规则、为什么读这些规则、这些规则是否已经被负责人确认，都应该能从 cache 或对应产物中找到证据。

注入顺序建议采用 `route -> required -> auto`：

1. `route`：注入顶层和祖先 `index.md`，让 Agent 知道路由结构。
2. `required`：注入匹配当前阶段和 Agent 的硬规则，只取 `## Runtime Rules`。
3. `auto`：从 selector cache 的 selected refs 中按预算展开 focused Spec 正文。

这种做法解决两个问题：一方面，Agent 不会因为全量规则上下文过载；另一方面，选中的规则有来源、有路径、有选择结果，后续可以审查为什么这次任务读了这些规则。

一条规则从产生到复用的闭环

SPEC 不是写完就结束，它应该来自真实交付，再服务下一次交付。

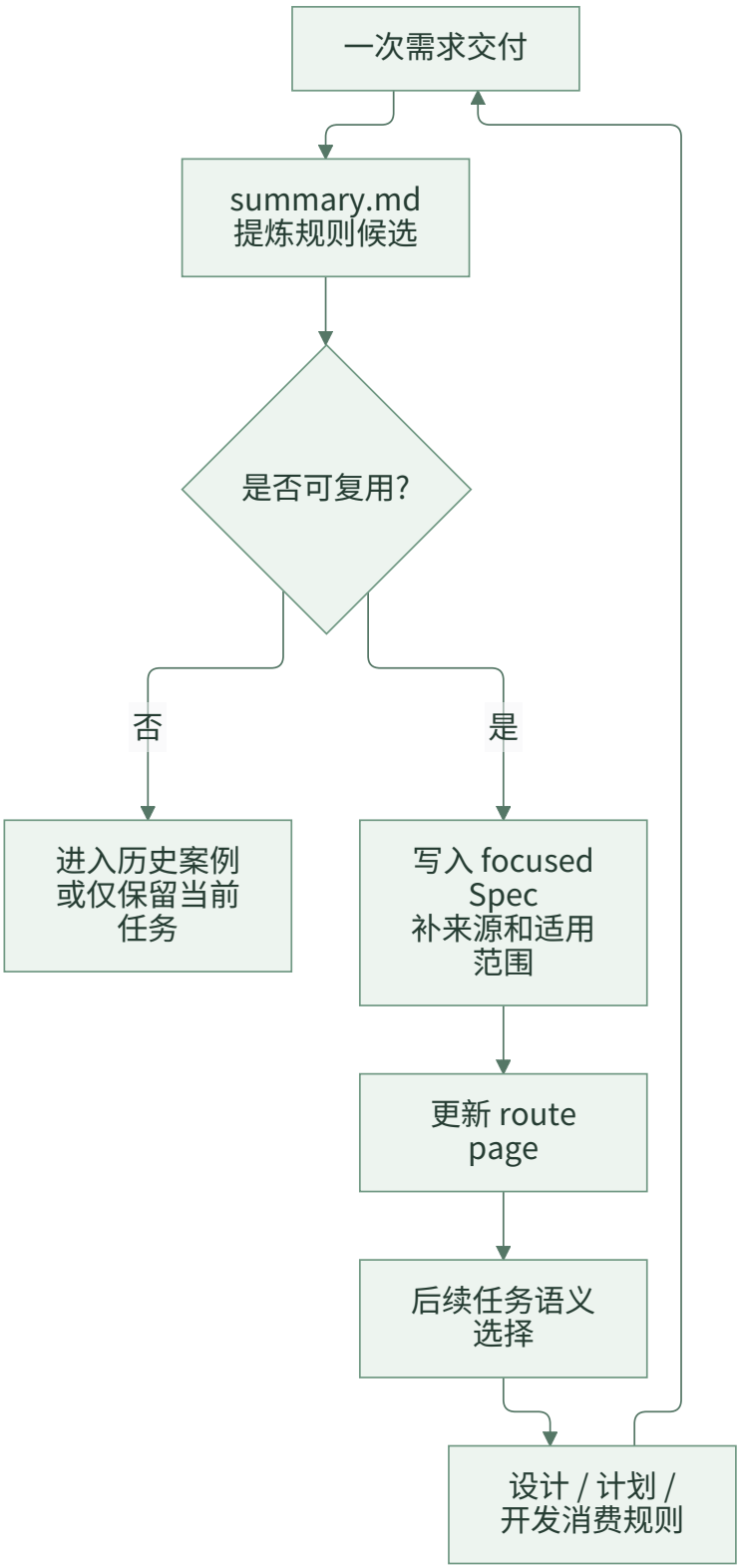


图 2-2 一条规则从产生到复用的闭环

这个闭环能避免两类问题：规则不是靠人拍脑袋定期整理，而是来自真实需求；规则也不是写进文档后没人用，而是会进入后续任务上下文。

设计前确认问题

SPEC 不只是告诉 Agent “怎么做”，也可以告诉流程“哪些问题必须先问清楚”。例如一个业务 focused Spec 里可以有这样的结构：

清单 2-5 设计前确认问题

```
## 设计前确认问题

1. 新增奖品类型是否允许存量订单查询展示?
2. 是否需要管理端配置入口?
3. 是否影响退款、补偿或履约回滚?
```

当当前需求命中这个 focused Spec 时，流程不应该直接进入技术设计，而是先把这些问题补齐。补齐后的答案应该写入 `design.md` 的前置事实区，例如：

清单 2-6 设计前确认问题

```
## 设计前置事实

- 新增奖品类型允许存量订单查询展示，展示名称沿用配置中心。
- 本次不新增管理端配置入口，只复用现有配置导入链路。
- 不影响退款和补偿链路，但需要补充查询展示回归测试。
```

这样做的价值很直接：业务规则不再只是“建议阅读”，而是能真正影响流程。缺关键事实时，Agent 不会自作主张生成方案。

SPEC 什么时候维护

规则体系真正难的不是建目录，而是后期维护。可以把维护入口放在三个位置。

表 2-5 SPEC 什么时候维护

入口	使用场景	产物
`/ai-init-spec`	应用首次接入，初始化应用 SPEC 拓扑	`profile.md`、根 `index.md`、三类子目录、`.initialized.json`
`update_spec` 阶段	一次 feature 交付结束后，判断是否有可复用规则	更新已有 focused Spec、推荐新增 Spec、或确认无需更新
`/ai-spec-add` / `/ai-spec-update`	任务外补充规则或修正规则	新增 focused Spec、更新 route page、修订已有 Spec

每次变更 `spec/app/**` 后，都应该跑校验：

清单 2-7 SPEC 什么时候维护

```
node .team-agent/tools/task.js check-app-spec
```

校验重点包括：

- app 顶层是否只包含允许的文件和目录。
- `profile.md` 和 focused Spec 是否带 `mode/phases/agents`。
- `index.md` 是否保持 route page，不误加 focused Spec frontmatter。
- `.initialized.json` 是否只在用户确认后写入。
- 业务规则是否放在 `business/`，工程规则是否放在 `engineering/`，领域知识是否放在 `domain/`。

维护动作的关键判断是：**这条信息下一次类似任务是否还应该被 Agent 使用？** 如果答案是肯定的，就应该进入 SPEC 或 archive；如果只是本次需求的临时事实，就留在 `prd.md`、`design.md` 或 `summary.md`。

交付结束后的规则维护流程

一次需求完成后，可以用下面的流程判断是否更新 SPEC。

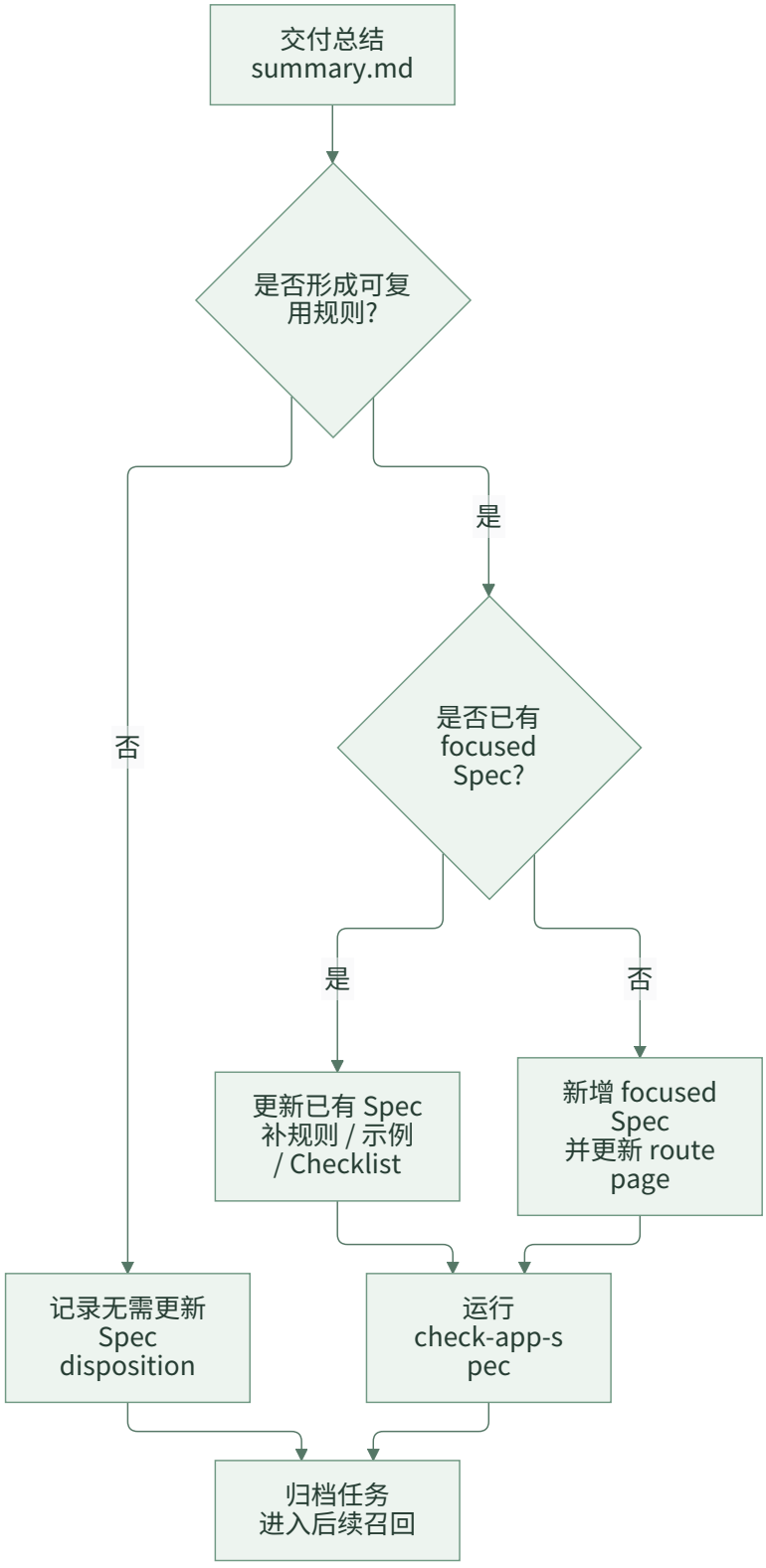


图 2-3 交付结束后的规则维护流程

几个判断标准可以帮助团队避免把 SPEC 写成垃圾桶：

表 2-6 交付结束后的规则维护流程

内容	应该放哪里	原因
本次需求的临时事实	`prd.md` / `design.md`	只对当前任务有效

内容	应该放哪里	原因
某类需求以后都要遵守的规则	focused Spec	后续任务需要自动消费
一次交付的取舍和踩坑	`summary.md` / archive	作为历史案例召回
代码文件的临时位置	context / code refs	可能随重构变化，不宜写成长期规则
团队统一工程规范	`spec/team/**`	不属于单个应用
应用业务不变量	`spec/app/domain/**` 或 `business/**`	属于应用知识资产

SPEC 维护要克制。一次需求结束后，如果没有形成可复用规则，可以明确记录“本次无需更新 SPEC”。强行为每个需求都新增规则，最后会让选择器和人都难以判断哪些规则重要。

一篇 focused Spec 的质量标准

一篇好的 focused Spec 不是“内容很全”，而是“能被后续任务正确使用”。

表 2-7 一篇 focused Spec 的质量标准

检查项	好的表现	差的表现
标题	对应一类业务或工程问题	对应某个文件名或一次需求名
适用范围	写清何时使用、何时不使用	只写背景介绍
规则	能指导设计、计划或实现	只有泛泛原则
示例	来自真实需求或代码约束	编造理想化例子
Checklist	能作为交付前检查	只是重复正文
Source Reference	能追溯来源	没有出处

如果一篇规则读完后，Agent 仍然不知道“当前任务该多做什么、少做什么、先确认什么”，这篇规则就还不够可执行。

2.5 团队参考做法

如果团队刚开始建设，可以先做一个简化版 SPEC 体系，不需要一开始就实现复杂 selector。

示例 2-8 团队参考做法

```
.team-agent/spec/
├── team/
│   ├── index.md
│   ├── architecture.md
```

```
│   ├── implementation.md
│   ├── verification.md
│   └── delivery.md
└── app/
    ├── profile.md
    ├── index.md
    ├── domain/
    │   └── index.md
    ├── business/
    │   └── index.md
    └── engineering/
        └── index.md
```

第一阶段只做四件事：

1. Prompt 只保留流程、工具和输出格式。
2. 团队通用规则放到 `team/`。
3. 应用差异放到 `app/domain`、`app/business`、`app/engineering`。
4. 每次需求结束后，明确判断“是否需要更新规则”。

第二阶段再做语义选择：

1. 为每个 focused Spec 加 metadata。
2. 根据 PRD 关键词、模块、代码路径初步选择候选规则。
3. 在设计或计划阶段显示 selected refs，让负责人可以确认。
4. 逐步引入预算控制和 required/auto 注入。

先把规则从 Prompt 中拆出来，收益已经很明显；不要一开始就追求复杂的知识图谱。

常见坑

表 2-8 常见坑

坑	表现	建议
route page 里写大段正文	index 越来越长，focused Spec 没人读	index 只做导航和触发条件
focused Spec 太细	每个接口、每个枚举都建一篇	一个 Spec 对应一类可复用规则，不对应代码文件
required 过多	每次任务都注入很多硬规则	required 只放短硬规则，其他用 auto
metadata 不维护	规则存在但选不中或总被选中	每次新增规则都确认 phases 和 agents
规则没有来源	不知道这条规则为什么存在	Source Reference 写历史需求、评审结论或业务确认
交付后只更新 summary	后续任务找不到长期规则	summary 负责案例，SPEC 负责长期规则

2.6 检查清单

可以用下面的问题检查团队 SPEC 体系是否健康：

- 团队规则是否还主要写在 Prompt 里？
- 同一条工程规范是否在多个应用 Agent 中重复维护？
- 应用业务规则是否有固定目录，而不是散在 Wiki 和聊天记录中？
- 每个规则文件是否能说明适用阶段、适用对象和来源？
- 是否区分 route page 和 focused Spec？
- 是否有“交付后是否更新规则”的固定动作？
- 变更规则后是否有校验机制？
- Agent 读取规则时是否能追溯“为什么读了这条规则”？

如果这些问题没有答案，优先建设 SPEC 目录和维护流程，再考虑更复杂的 Agent 能力。

2.7 思考与实践

1. 你们团队最常重复提醒给 AI 的三条规则是什么？它们现在写在 Prompt、Wiki、代码注释，还是有专门的规则文件？
2. 选一条近期反复踩坑的业务规则，试着判断它应该放在 `spec/team`、`spec/app/domain`、`spec/app/business` 还是 `spec/app/engineering`。
3. 如果每次需求结束都要判断“是否更新 SPEC”，你觉得团队最可能卡在哪一步？

2.8 结尾

SPEC 体系的目标不是让文档变多，而是让团队知识从聊天、Prompt 和个人经验中独立出来，成为 Agent 可以稳定消费、团队可以审查、后续可以维护的工程资产。

下一篇进入复杂需求拆分：当 PRD 足够大时，为什么不能直接把整篇需求交给 Agent，而要先变成功能点切片。

PART 02

第2篇　从复杂需求到可审查交付

复杂需求需要被拆成有边界、有验收标准的切片。随后为每个阶段选择上下文，把设计、计划、任务和代码变更连接起来，使一次交付可以逐步检查。

阅读重点

能沿着同一需求核对功能切片、阶段产物与任务变更。

第3章　复杂需求拆分：大 PRD 不能直接丢给 Agent

很多团队在 AI Coding 中遇到的不稳定，不是模型完全不懂业务，而是输入太大、边界太混。一个需求文档里可能同时包含新增配置、改接口、补查询、兼容存量、调整权限、增加监控，还涉及多个模块和多类测试。直接把整篇 PRD 丢给 Agent，结果往往是大方案、大计划、大 diff：看起来很完整，但很难评审，也很难确认每个功能点是否真的被实现和验证。

复杂需求需要先被结构化。这里的“拆分”不是项目管理意义上的排期拆分，也不是一上来就把一个需求拆成多个子项目，而是先把 PRD 变成 Agent 和评审人都能稳定消费的功能点切片。每个切片都应该能回答四个问题：这个功能点要做什么，涉及哪些范围，验收标准是什么，可能关联哪些代码和规则。

这里还有一个容易被低估的点：功能切片的真相源仍然是 PRD，但切片边界不能只看 PRD 文本。系统里已经维护的领域 SPEC、业务规则、工程约束、代码索引和历史案例，都可以用来校准切片边界。它们不能替 PRD 新增需求，但能帮助团队识别“这个功能属于哪个领域、有哪些既有规则、设计前还有哪些问题必须确认”。

功能点切片的价值在于让后续阶段围绕同一组边界工作：上下文选择按切片找证据，技术设计按切片说明影响面，开发计划按切片拆 task，验证证据按切片回到验收标准。没有这个结构，AI 很容易在每个阶段重新理解一次需求，最后前后不一致。

> 一句话结论：**复杂需求交给 AI 之前，先要把需求拆成可验收、可追溯、可进入上下文选择的功能切片。**

3.1　一个真实问题：大 PRD 为什么容易变成大 diff

假设 PRD 标题是“新增奖品权益类型支持”。正文里可能包含这些内容：

- 新增权益类型枚举。
- 配置导入支持新类型。
- 创建接口允许选择新类型。
- 查询接口展示新类型名称。
- 存量类型保持兼容。
- 管理端暂时不新增配置入口。
- 需要补充埋点和监控。
- 需要覆盖退款和履约链路不受影响。

如果直接把这份 PRD 给 Agent，Agent 可能会按技术层拆：

- 改 Controller。
- 改 Service。
- 改 DAO。
- 改枚举。
- 补测试。

这种拆法不是完全错，但它没有回答业务验收问题：新增类型到底是哪几个链路要支持？哪些链路只是回归验证？哪些模块是主改动，哪些只是兼容确认？如果计划阶段不清楚，开发阶段就会变成大范围探索式修改。最后 diff 很大，评审人也很难判断“这个需求是否完整交付”。

更合理的做法是先按功能点切片：

表 3-1 一个真实问题：大 PRD 为什么容易变成大 diff

功能切片	目标	验收
feature-01 新权益类型识别	系统能识别新增权益类型并生成配置	新类型可创建，非法类型被拦截
feature-02 查询展示兼容	查询接口能展示新类型名称，存量类型不变	新旧类型查询结果符合预期
feature-03 回归影响确认	退款、履约、监控链路不受影响	相关回归测试通过或有 baseline 证据

这样后续无论是上下文选择、设计、计划还是验证，都能围绕功能边界展开，而不是围绕代码文件猜测需求。

常见误区

表 3-2 常见误区

误区	表现	后果
整篇 PRD 直接进 Agent	Agent 自己理解、自己拆、自己设计	每个阶段可能拆法不同，前后不一致
功能点和执行切片混在一起	识别出 5 个功能点，就默认开 5 条开发流	过早复杂化，状态管理和验收成本上升
只按代码模块拆	Controller、Service、DAO 各一块	忽略业务验收边界，容易实现碎片化
只按 PRD 标题拆	标题像功能点就直接作为切片	无法保证验收、代码、规则能对应上
切片后不进入上下文选择	文件拆了，但 Agent 仍读全文和全仓	拆分没有减少复杂度

复杂需求拆分的关键，是先稳定需求结构，再让上下文和计划围绕这个结构工作。

一个判断：PRD 大不等于要分片交付

很多团队一看到大 PRD，就想把它拆成多个执行流。这个反应很自然，但不一定正确。PRD 大通常先说明“上下文需要拆”，不一定说明“交付也要拆”。

表 3-3 一个判断：PRD 大不等于要分片交付

情况	建议处理
PRD 很长，但属于同一业务目标	先做 context slice，帮助设计和计划聚焦
涉及多个业务 owner	考虑执行切片，但需要负责人确认边界
涉及多个发版窗口	考虑 program slice，分阶段验收
只是包含很多实现细节	不要拆执行流，把细节放到 design/plan
横跨多个领域模型且互相依赖	先生成 requirement contract，再判断是否 program

这条边界很关键：**切片首先是理解和上下文工具，其次才可能成为执行工具。**

设计原则

第一，**功能切片先是上下文单元，不默认是执行单元**。识别出功能点，只说明上下文可以更聚焦，不代表一定要拆成多个独立开发项目。

第二，**切片必须保留原始证据**。每个功能点都要能追溯到 PRD 原文片段，避免 Agent 或人把总结当成需求真相。

第三，**切片要连接验收、规则和代码**。只有标题和摘要不够，后续还要能关联验收项、候选模块、SPEC 和 code refs。

第四，**切片粒度要面向业务验收，而不是面向代码文件**。一个切片应该对应业务方能理解的一项能力或影响面，而不是一个方法、一个枚举或一个 Mapper。

3.2 切片前先做 PRD 消毒

把原始需求变成稳定 PRD 前，建议先做一次“消毒”。目的不是改写业务方意思，而是把混在一起的信息分层。

表 3-4 切片前先做 PRD 消毒

原始内容	处理方式
业务目标、范围、验收	进入 `prd.md` 正文
技术方案建议	保留在 source 或 design input，不直接变成 PRD 真相
临时沟通结论	标注来源和确认状态

原始内容	处理方式
不确定字段	写成 open question，不让 Agent 猜
明确非目标	写入 scope out，防止开发越界

这一步能减少后续常见问题：Agent 把业务方的“可能方案”当成硬要求，把未确认事实当成设计前提。

从原始需求到稳定 PRD

团队 AI Coding 工程化框架不应该把需求来源直接交给后续 Agent，而是先生成稳定 PRD 产物。

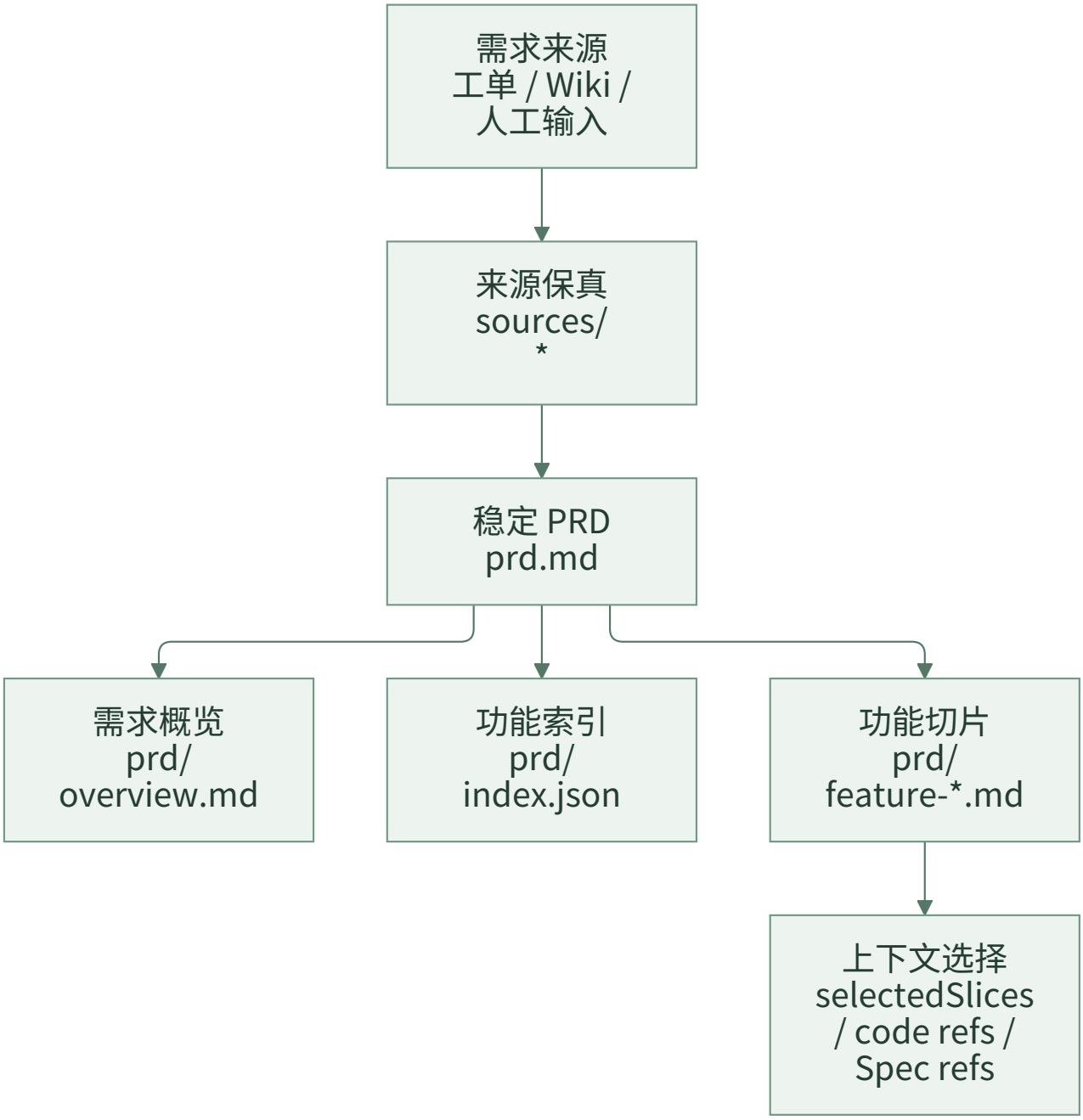


图 3-1 从原始需求到稳定 PRD

典型任务目录如下：

示例 3-1 从原始需求到稳定 PRD

```
.team-agent/tasks/{taskId}/
├── prd.md
├── sources/
│   ├── requirement.json
│   ├── wiki.md
│   └── manual-prd.md
└── prd/
    ├── overview.md
    ├── index.json
    ├── cross-cutting.md
    ├── feature-01.md
    ├── feature-02.md
    └── feature-03.md
```

这里有三个层次：

- `sources/*` 保存原始来源，解决追溯问题。
- `prd.md` 是用户确认后的稳定需求产物，解决需求真相源问题。
- `prd/overview.md`、`prd/index.json`、`prd/feature-*.md` 是可重建 companion 文件，解决后续上下文选择和功能点拆分问题。

如果团队暂时没有自动 builder，也可以人工维护这个结构。关键不是自动化程度，而是 PRD 确认后，后续阶段都围绕同一个稳定结构工作。

3.3 功能点切片如何生成

一个通用的 PRD builder 可以按五步生成候选功能切片：

1. 按 Markdown 二级标题拆分 PRD 顶层章节。
2. 识别标题中带有 `功能`、`feature`、`模块`、`场景`、`能力` 或编号结构的章节。
3. 为每个章节生成稳定 ID，例如 `feature-01`。
4. 从章节正文中提取摘要、验收项和候选模块。
5. 根据功能数量、PRD 长度、仓库复杂度判断上下文模式和风险信号。

候选切片生成后，还要用其他工程资产做一次校准。这里要分清每类输入的职责：

表 3-5 功能点切片如何生成

输入	用来做什么	不能做什么
PRD	决定本次要做什么、验收什么	不能被 SPEC 覆盖

输入	用来做什么	不能做什么
SPEC	校准领域边界、业务规则、设计前确认问题	不能凭空新增需求
Code Index	判断候选实现路径、测试入口和风险热点	不能反推业务目标
Archive Case	提醒历史相似风险和验证经验	不能替代当前验收标准

比较稳的顺序是：先从 PRD 生成候选切片，再用命中的 domain / business SPEC 补充 `specRefs`、设计前问题和规则约束，再用 code index 推断候选路径，最后用 archive case 提醒历史风险。这样切片既不脱离当前需求，也不会忽略系统已有知识。

生成的 `prd/index.json` 类似这样：

清单 3-2 功能点切片如何生成

```
{
  "taskId": "REQ-123456",
  "title": "新增奖品权益类型",
  "mode": "expanded",
  "source": "wiki",
  "overviewPath": ".team-agent/tasks/REQ-123456/prd/overview.md",
  "prdPath": ".team-agent/tasks/REQ-123456/prd.md",
  "crossCuttingPath": ".team-agent/tasks/REQ-123456/prd/cross-cutting.md",
  "features": [
    {
      "id": "feature-01",
      "title": "功能一：新增权益类型识别",
      "path": ".team-agent/tasks/REQ-123456/prd/feature-01.md",
      "summary": "支持新权益类型识别、配置生成和创建接口校验",
      "acceptance": ["新权益类型可创建", "非法类型被拦截"],
      "modules": ["权益类型", "奖品配置"],
      "specRefs": [".team-agent/spec/app/business/prize-type.md"]
    },
    {
      "id": "feature-02",
      "title": "功能二：查询展示兼容",
      "path": ".team-agent/tasks/REQ-123456/prd/feature-02.md",
      "summary": "查询接口展示新类型名称，存量类型展示不变",
      "acceptance": ["新类型可查询展示", "存量类型返回结构不变"],
      "modules": ["查询接口", "展示映射"],
      "specRefs": [".team-agent/spec/app/business/prize-type.md"]
    }
  ]
}
```

对应的功能切片文件会保留更适合人阅读的内容：

清单 3-3 功能点切片如何生成

```
# 功能切片 feature-01

## 基本信息

- 需求标题：新增奖品权益类型
- 功能标题：功能一：新增权益类型识别
- 来源：wiki

## 摘要

支持新权益类型识别、配置生成和创建接口校验。

## 相关模块

- 权益类型
- 奖品配置

## 验收与范围

- 新权益类型可创建
- 非法类型被拦截
- 存量权益类型创建行为不变

## 风险信号

- 新枚举可能影响存量配置解析。
- 创建链路和查询链路使用的类型映射可能不是同一个入口。

## 原始片段

这里保留 PRD 中与该功能点直接相关的原文。
```

注意，`summary`、`modules` 和 `specRefs` 都是帮助后续选择上下文的结构化信息，不替代原始片段。评审人和 Agent 必须能回到原文确认需求含义。

用 SPEC 校准切片边界

如果团队已经维护了领域或业务 focused SPEC，功能切片就不应该只靠 PRD 标题机械生成。SPEC 可以帮助回答几个问题：

表 3-6 用 SPEC 校准切片边界

问题	SPEC 能提供的帮助
这个功能属于哪个领域？	用已有领域模型和业务术语统一命名
这个功能有哪些长期规则？	找到创建、查询、展示、兼容、灰度、验证等固定要求
设计前有什么必须确认？	从 SPEC 的确认问题里提取待确认事实
哪些路径只是回归，不是新增需求？	用历史规则区分主改动和影响确认

比如 PRD 只写了“新增权益类型支持”，SPEC 里已经沉淀过“权益类型变更必须覆盖创建、查询、展示、存量兼容”。这时切片不应该只生成一个“新增类型”大块，而应该至少识别出主功能切片和回归影响切片。反过来，如果 SPEC 里有“查询展示必须回归”，但 PRD 明确说明“本次不涉及查询展示”，那就不能把它写成新增需求，只能作为风险提示或非目标确认问题。

这就是 PRD 和 SPEC 的边界：PRD 决定需求事实，SPEC 提供领域校准。

功能切片的命名技巧

切片命名不要像代码任务，也不要太抽象。推荐格式是：

```
feature-序号 + 业务能力 + 结果
```

好的例子：

- `feature-01` 新权益类型识别和创建校验
- `feature-02` 查询展示兼容
- `feature-03` 退款履约影响确认

不好的例子：

- `feature-01` 改 `Service`
- `feature-02` 补测试
- `feature-03` 后端逻辑

好的切片标题应该让业务方、研发、测试和 Agent 都能理解它对应哪一段需求。

切片粒度怎么判断

功能切片不是越多越好。粒度太粗，无法减少复杂度；粒度太细，计划和上下文会碎片化。可以用下面几个判断标准。

表 3-7 切片粒度怎么判断

判断问题	如果答案是“是”	处理方式
这个功能点能被业务方独立理解吗？	是	可以作为候选切片
它是否有独立验收标准？	是	应该保留为切片
它是否只对应一个代码文件或方法？	是	通常不应单独成切片
它是否只是另一个功能点的实现细节？	是	放入父切片的相关模块或 task
它是否跨 owner、跨领域或跨发版窗口？	是	可能需要升级为执行切片

判断问题	如果答案是“是”	处理方式
它是否只是全局约束，例如日志、监控、兼容？	是	放入 `cross-cutting.md` 或切片风险

经验上，一个中等复杂需求拆 3 到 7 个功能切片比较常见。超过 10 个切片时，要反过来检查 PRD 是否本身包含多个独立需求。

三种切片层级

复杂需求拆分最容易混淆的地方，是把“功能点切片”直接等同于“分片交付”。这两者不是一回事。

表 3-8 三种切片层级

层级	产物	作用	是否代表独立执行
Context Slice	`prd/index.json`、`prd/feature-*.md`	帮助 selector 选择 PRD evidence、code refs、SPEC	否
Delivery Support	`generated/delivery/requirement-contract.json`、`slice-manifest.json`	在 expanded 或 program 场景冻结需求目标、切片、依赖、候选路径	不一定
Program Slice	`generated/context/prd.slice-{sliceId}.jsonl`、`code.slice-{sliceId}.jsonl`、`verification-ledger.slice-{sliceId}.jsonl`	用户确认分片交付后，围绕当前切片执行	是

这个边界很重要。大多数复杂需求只需要 context slice：让设计、计划、上下文选择更聚焦。只有当需求确实跨 owner、跨发版窗口、跨领域，或者需要分段验收时，才应该升级为 program slice。

建议规则是：`large_prd`、`multi_domain`、`contextMode=expanded` 只触发更强的上下文支持，不自动进入 program mode。只有负责人显式确认分片执行边界后，切片才成为执行边界。

什么时候必须让人确认切片

不是所有切片都需要人工逐个确认。团队要避免把流程做得太碎。但下面几类情况应该停下来确认：

表 3-9 什么时候必须让人确认切片

信号	为什么要确认
切片之间有强依赖	执行顺序会影响设计和发布

信号	为什么要确认
切片跨不同 owner	需要明确谁验收、谁承担风险
某个切片包含高风险存量改动	可能需要独立回滚和灰度
PRD 中存在互相矛盾的验收项	不能让 Agent 自己裁剪需求
自动拆分超过 10 个切片	可能其实是多个需求

确认的目标不是让用户逐字审切片，而是确认“边界、顺序、验收和非目标”。

Requirement Contract 和 Slice Manifest

当需求足够复杂，需要冻结更明确的交付支持信息时，可以生成 `generated/delivery/*`。

```
.team-agent/tasks/{taskId}/generated/delivery/
├── requirement-contract.json
├── slice-manifest.json
└── manual-slice-input.md
```

`requirement-contract.json` 用来冻结需求目标和验收边界。它不是替代 PRD，而是给后续设计、计划、开发提供机器可读的需求契约。

清单 3-4 Requirement Contract 和 Slice Manifest

```
{
  "taskId": "REQ-123456",
  "title": "新增奖品权益类型",
  "goals": [
    {
      "id": "REQ-001",
      "featureId": "feature-01",
      "statement": "系统支持新增权益类型识别和配置生成",
      "acceptance": ["新权益类型可创建", "非法类型被拦截"]
    },
    {
      "id": "REQ-002",
      "featureId": "feature-02",
      "statement": "查询接口展示新类型名称且存量类型不变",
      "acceptance": ["新类型可查询展示", "存量类型返回结构不变"]
    }
  ],
  "crossCutting": [
    "需要确认存量类型兼容策略",
    "需要补充查询和展示回归验证"
  ]
}
```

`slice-manifest.json` 用来描述切片、依赖、候选规则和候选代码路径。它通常不是只从 PRD 生成，而是在 PRD 候选切片基础上合并已选择的 SPEC 和代码索引信号。

清单 3-5 Requirement Contract 和 Slice Manifest

```
{
  "taskId": "REQ-123456",
  "mode": "context",
  "source": {
    "mode": "auto",
    "inputs": ["prd"]
  },
  "calibrationSignals": ["selected_specs", "code_index"],
  "slices": [
    {
      "id": "feature-01",
      "title": "功能一：新增权益类型识别",
      "objective": "支持新权益类型识别、配置生成和创建接口校验",
      "requirementRefs": ["REQ-001"],
      "specRefs": [".team-agent/spec/app/business/prize-type.md"],
      "runtimeRuleRefs": [".team-agent/spec/app/business/prize-type.md#Runtime Rules"],
      "dependsOn": [],
      "riskSignals": ["large_prd"],
      "candidateDomains": ["权益类型"],
      "candidatePaths": ["src/main/java/..."]
    }
  ]
}
```

这两个文件的作用是让后续阶段有稳定边界：

- designer 可以围绕切片目标和跨切面约束写设计。
- planner 可以把任务拆到更清晰的功能边界。
- context selector 可以选择相关 SPEC 和 code refs。
- worker 在 program mode 下可以只消费当前 slice 的 working set。

切片如何进入上下文选择

功能切片只有进入上下文选择，才真正降低复杂度。

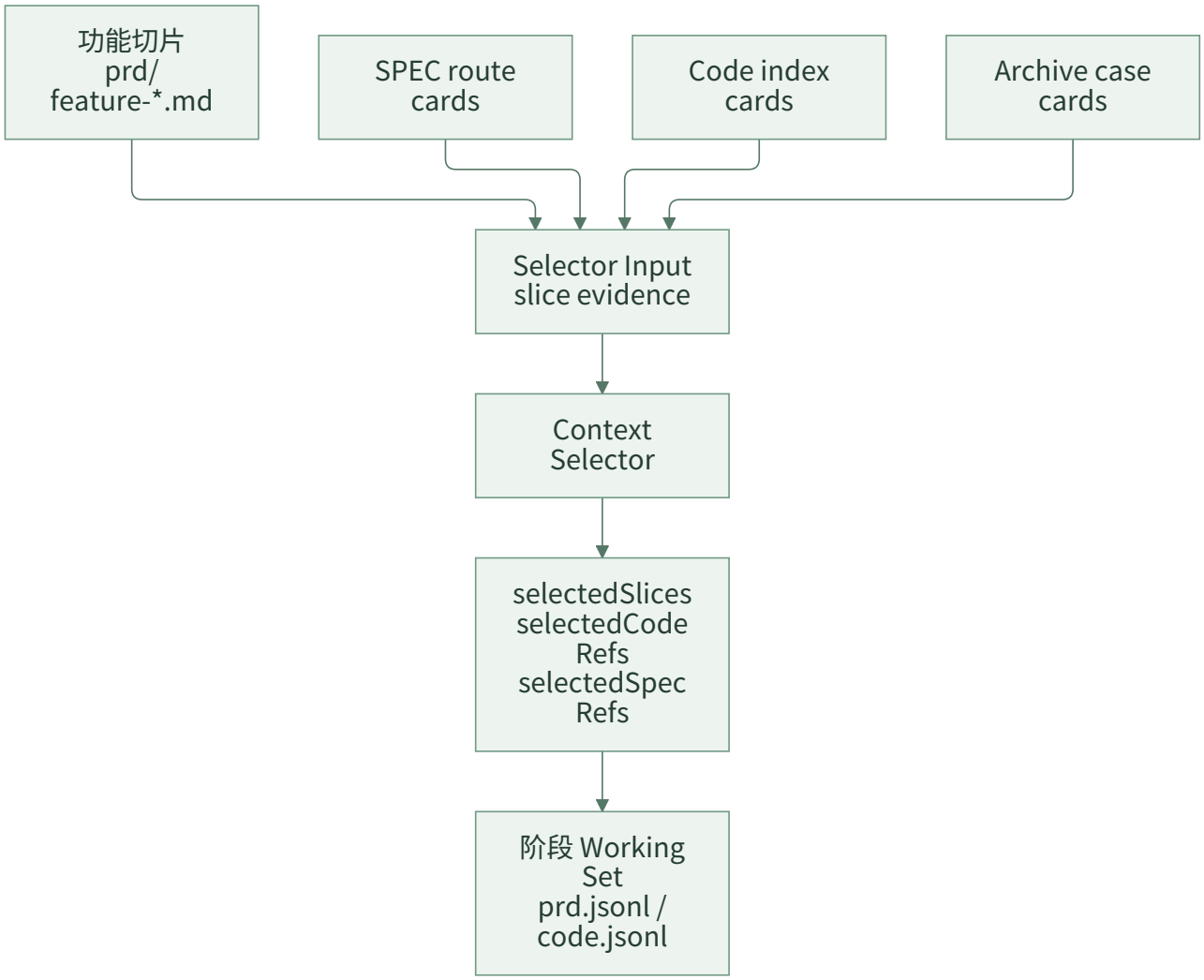


图 3-2 切片如何进入上下文选择

selector input 可以包含：

- 当前任务摘要。
- `prd/index.json` 中的 features。
- `slice-manifest.json` 中的 slices。
- SPEC route cards 和 business Spec candidates。
- code-index cards。
- archive case cards。
- 预算、阶段、Agent、fingerprint。

输出可以写入：

```
.team-agent/cache/tasks/{taskId}/context-selection.json
```

典型字段如下：

清单 3-6 切片如何进入上下文选择

```
{
  "selectedSlices": [
    {
      "id": "feature-01",
      "reason": "当前需求主要涉及新增权益类型识别和配置生成"
    }
  ],
  "selectedCodeRefs": [
    {
      "path": "src/main/java/com/example/prize/PrizeTypeService.java",
      "reason": "处理奖品类型识别和配置生成"
    }
  ],
  "selectedSpecRefs": [
    ".team-agent/spec/app/business/prize-type.md"
  ]
}
```

之后 runtime 会把这些选择转成阶段 working set，例如：

```
.team-agent/tasks/{taskId}/generated/context/
├── prd.jsonl
├── code.jsonl
├── prd.slice-feature-01.jsonl
└── code.slice-feature-01.jsonl
```

non-program 模式下，通常只消费共享的 `prd.jsonl` 和 `code.jsonl`；program mode 下，当前切片才会使用 slice 专属 working set。

切片如何影响设计和计划

切片不能只停留在 `prd/` 目录里，它要继续进入 `design.md` 和 `plan.md`。

`design.md` 可以按功能切片说明影响面：

清单 3-7 切片如何影响设计和计划

```
## 功能切片设计

### feature-01 新权益类型识别

- 目标：支持新权益类型识别、配置生成和创建接口校验。
- 影响模块：PrizeTypeService、PrizeConfigParser、CreatePrizeRequestValidator。
- 兼容策略：存量类型映射保持不变，新增类型走独立枚举分支。
- 风险：配置解析和查询展示使用不同映射入口，需要统一回归。

### feature-02 查询展示兼容
```

```
- 目标：查询接口展示新类型名称，存量类型展示不变。
- 影响模块：PrizeQueryService、PrizeTypeDisplayMapper。
- 验证：新增类型查询测试，存量类型查询回归测试。
```

`plan.tasks.json` 里每个 task 要能回到功能切片：

清单 3-8 切片如何影响设计和计划

```
{
  "tasks": [
    {
      "id": "TASK-001",
      "featureId": "feature-01",
      "title": "支持新权益类型识别和创建校验",
      "allowedPaths": [
        "src/main/java/com/example/prize/PrizeTypeService.java",
        "src/test/java/com/example/prize/PrizeTypeServiceTest.java"
      ],
      "verificationCommands": [
        "mvn -Dtest=PrizeTypeServiceTest test"
      ],
      "acceptanceCriteria": [
        "新权益类型可创建",
        "非法类型被拦截",
        "存量类型创建行为不变"
      ]
    }
  ]
}
```

这样做的好处是，后续代码 diff、测试证据和验收标准能形成闭环。如果 task 只写“修改 Service 逻辑”，评审人很难知道它服务于哪个需求目标。

3.4 团队参考做法

团队刚开始不需要完整实现 builder、selector 和 program mode，可以先做一个简化版需求切片机制。

第一阶段，先让 PRD 结构化：

示例 3-9 团队参考做法

```
.team-agent/tasks/{taskId}/
├── prd.md
└── prd/
    ├── overview.md
    ├── index.json
    └── feature-01.md
```

`prd/index.json` 只需要包含：

清单 3-10 团队参考做法

```
{
  "taskId": "T-001",
  "title": "需求标题",
  "features": [
    {
      "id": "feature-01",
      "title": "功能点标题",
      "summary": "功能点摘要",
      "acceptance": ["验收项 1", "验收项 2"],
      "modules": ["候选模块"],
      "specRefs": [".team-agent/spec/app/business/example.md"],
      "sourcePath": ".team-agent/tasks/T-001/prd/feature-01.md"
    }
  ]
}
```

第二阶段，把切片接入设计和计划：

- `design.md` 中按功能点说明影响范围。
- `plan.md` 中按功能点拆 task。
- `plan.tasks.json` 中每个 task 标注相关 `featureId`、`requirementRefs` 和 `specRefs`。
- 验证命令和验收标准要能回到 feature。

第三阶段，再考虑 program mode：

- 只有跨 owner、跨领域、跨发版窗口的需求才升级。
- 每个执行切片要有独立验收和验证证据。
- 切片之间依赖要显式记录。
- 不要因为 PRD 大就自动开多条执行流。

常见坑

表 3-10 常见坑

坑	表现	建议
按代码层拆切片	Controller / Service / DAO 各一片	切片按业务能力拆，代码层放在 candidatePaths
切片没有原文	只有模型总结	每个切片保留原始 PRD 片段
切片没有验收	只有标题和摘要	没有验收标准的切片不能进入计划
切片不看已有 SPEC	PRD 标题怎么写就怎么拆	用 SPEC 校准领域边界和设计前确认问题
切片过细	一个枚举一个切片，一个字段一个切片	合并到业务功能点，细节放 task

坑	表现	建议
切片过粗	一个切片包含 8 个业务目标	拆成多个 feature，或确认是否多个需求
自动升级执行流	看到多切片就开多条开发流	默认只做上下文切片，执行切片需显式确认
切片不进计划	`prd/feature-*.md` 写了，但 task 不引用	`plan.tasks.json` 增加 `featureId`

切片质量检查

一个好的功能切片至少要满足：

- 有稳定 ID，例如 `feature-01`。
- 有清晰标题和摘要。
- 有原始 PRD 片段。
- 有验收项。
- 有候选业务模块或领域。
- 能关联相关 SPEC。
- 能说明这些 SPEC 是规则校准，不是新增需求来源。
- 能帮助选择代码上下文。
- 能在计划中映射到 task。

切片不是越细越好。过细会导致上下文碎片化，评审人也难以把握整体需求。经验上，一个切片应该对应一个可以被业务方理解的功能点，而不是一个技术方法或一个代码文件。

3.5 检查清单

可以用下面的问题检查复杂需求拆分是否有效：

- 大 PRD 是否先生成了稳定 `prd.md`?
- 每个功能点是否保留原始需求片段?
- 切片是否有验收项，而不只是标题?
- 是否区分 context slice 和 program slice?
- 是否避免因为 `large_prd` 就自动进入多执行流?
- 设计和计划是否引用了功能切片?
- 上下文选择是否使用了 selectedSlices?
- 每个执行 task 是否能回到具体功能点、需求契约、SPEC 和验收标准?

如果这些问题没有答案，先不要急着让 Agent 开发。先把需求结构稳定下来，再进入设计、计划和代码修改。

3.6 思考与实践

1. 选一个你最近做过的大需求，能不能拆成 3-7 个业务功能切片？每个切片是否都有独立验收标准?
2. 你们团队现在更常按业务功能拆需求，还是按 Controller / Service / DAO 这类代码层拆任务?
3. 哪类需求在你们团队里应该升级为 program slice？是跨 owner、跨领域、跨发版窗口，还是其他原因?

3.7 结尾

复杂需求拆分的目标不是把流程做重，而是让 Agent 和人都围绕同一组功能边界工作。PRD 先结构化，上下文再选择，计划再拆 task，开发再围绕 diff 和验证证据收敛。

下一篇进入有界上下文：功能切片、SPEC、代码索引和历史案例如何共同组成 Agent 当前阶段真正需要的 working set。

第4章　有界上下文：不是给 Agent 更多，而是给得更准

AI Coding 的一个常见误区是：模型效果不好，就继续塞上下文。PRD 全文、Wiki 全文、代码目录、团队规范、历史问题、接口文档都塞进去。结果是上下文窗口被大量无关内容占满，Agent 输出看似全面，但经常选错模块、漏掉关键约束，或者在相似概念之间混淆。

团队级 AI Coding 不能依赖“让 Agent 自己全仓搜索”。真实业务仓库里，代码量大、模块多、历史包袱多、规则也多。更稳定的做法是先把需求、代码、规则和历史案例变成可选择的候选，再为当前阶段生成刚好够用的 working set。

有界上下文的目标，不是少给信息，而是给准信息，并且让选择过程可追溯、可重建、可审查。

> 一句话结论：**上下文不是越多越好，而是越准越好。**

常见误区

表 4-1　常见误区

误区	表现	后果
PRD 全文每阶段都塞	设计、计划、开发都重读整篇需求	关键功能点被淹没，阶段成本高
让 Agent 全仓扫描	交给模型自己找相关代码	容易被同名类、旧模块、测试代码误导
SPEC 全量注入	所有规范和业务规则都进上下文	预算浪费，不相关规则干扰判断
历史案例直接塞进会话	相似需求都作为背景材料	历史上下文过重，且无法判断是否适用
上下文失效不重建	PRD 改了，仍沿用旧 working set	设计、计划、实现基于过期事实

有界上下文的核心，是把“搜索和筛选”从 Agent 的临场发挥，变成运行时可观察的构建过程。

4.1　为什么“给更多上下文”会变差

模型不是不知道信息越多，而是很难判断哪些信息更重要。业务仓库里经常有同名类、历史实现、废弃接口、测试桩、临时兼容逻辑。全量上下文会让 Agent 看起来更“全面”，但实际更容易被噪音带偏。

表 4-2 为什么“给更多上下文”会变差

上下文类型	常见噪音	风险
PRD 全文	背景说明、历史沟通、非目标、技术建议	把未确认信息当需求
代码全仓	旧模块、测试桩、同名工具类、废弃入口	改错位置
规则全文	不相关业务规则、过期规范	忽略当前任务关键规则
历史案例	相似但边界不同的旧需求	套用错误方案

有界上下文的关键不是“少给”，而是给出证据优先级：当前 PRD 和确认事实优先，其次是命中的规则和代码，再其次才是历史案例。

设计原则

第一，**先结构化，再选择**。原始 PRD、代码、规则不能直接堆给 Agent，要先变成 feature、code card、Spec route card、archive card。

第二，**按阶段生成 working set**。设计阶段、计划阶段、开发阶段需要的上下文不同，不应该消费同一份大上下文。

第三，**上下文要可重建**。PRD 变化、代码变化、规则变化、历史召回变化后，working set 应该能通过命令重建，而不是手工修补。

第四，**历史案例只能弱引用**。历史案例用于提醒风险和复用思路，不能覆盖当前需求、当前规则和当前设计事实。

4.2 案例拆解：上下文构建流水线

这套框架的上下文构建可以分成 builder、selector、runtime 三层。

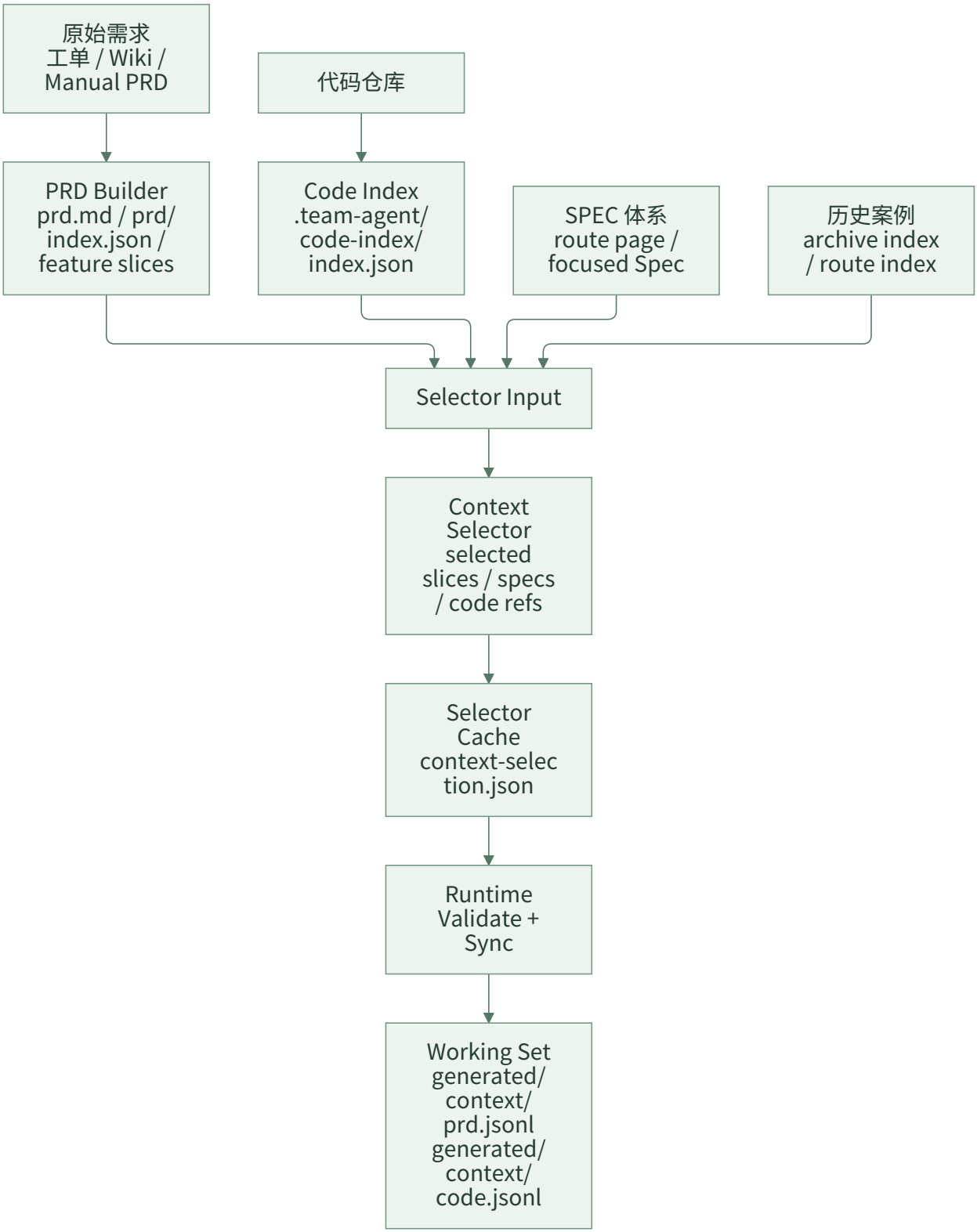


图 4-1 案例拆解：上下文构建流水线

几类关键产物如下：

示例 4-1 案例拆解：上下文构建流水线

```
.team-agent/
├── code-index/
│   └── index.json
├── cache/tasks/{taskId}/
│   └── context-selection.json
└── tasks/{taskId}/
    ├── prd/index.json
    ├── context/archive-recall.json
    └── generated/context/
        ├── prd.jsonl
        ├── code.jsonl
        ├── prd.slice-feature-01.jsonl
        └── code.slice-feature-01.jsonl
```

这条链路对应几个命令：

清单 4-2 案例拆解：上下文构建流水线

```
node .team-agent/tools/task.js build-code-index
node .team-agent/tools/task.js build-context-selection-input <taskId> --phase tech_design
node .team-agent/tools/task.js write-context-selection <taskId> <selection-json-path>
node .team-agent/tools/task.js sync-context <taskId>
```

`build-context-selection-input` 会把当前任务摘要、PRD feature slices、Spec route cards、business Spec candidates、archive case cards、code-index cards 和预算信息打包成 selector input。selector 输出后，runtime 会校验路径、预算、fingerprint 和引用合法性，再写入 selector cache。

Selector Cache 长什么样

一个简化的 `context-selection.json` 可以长这样：

清单 4-3 Selector Cache 长什么样

```
{
  "phase": "tech_design",
  "selectedSlices": [
    {
      "id": "feature-01",
      "reason": "当前需求主要涉及新增权益类型识别"
    }
  ],
  "selectedCodeRefs": [
    {
      "path": "src/main/java/com/example/prize/PrizeTypeService.java",
      "reason": "处理奖品类型识别和配置生成"
    }
  ],
  "selectedSpecRefs": [
```

```
    ".team-agent/spec/app/business/prize-type.md"
  ],
  "selectedCases": [
    {
      "archiveId": "2026-05-01-prize-type-extension",
      "reason": "历史需求包含相同奖品类型扩展路径"
    }
  ],
  "fingerprints": {
    "selectorInput": "sha256:input",
    "selectorOutput": "sha256:output"
  }
}
```

这个文件不是任务状态真相源，而是上下文选择结果。它的价值是让后续阶段知道：这次为什么选了这些功能点、规则、代码和历史案例。

Selector 也需要硬边界

如果团队引入模型做语义选择，要给 selector 明确边界。它只回答“哪些候选上下文相关”，不能顺手改状态、改产物或决定流程。

表 4-3 Selector 也需要硬边界

Selector 可以做	Selector 不应该做
从候选规则中选相关 Spec	决定 required 规则是否生效
从 PRD 切片中选当前阶段相关切片	修改 PRD 内容
从 code index 中选候选代码路径	直接改代码
从历史案例中选弱参考	覆盖当前需求事实
输出路径、理由和置信度	推进 phase 或写 task 状态

这条边界能避免 selector 从“上下文选择器”变成另一个隐形调度者。

Selector 输出必须被运行时校验

如果 selector 由模型完成，输出结果不能直接进入后续阶段。模型可以做语义判断，但引用合法性、路径边界和预算控制必须由确定性运行时校验。

表 4-4 Selector 输出必须被运行时校验

输出	运行时应该校验什么
`selectedSpecRefs`	是否来自可发现的 SPEC route cards，是否匹配当前 phase / agent，是否超出预算

输出	运行时应该校验什么
`selectedSlices`	是否引用已存在的 requirement / acceptance，是否超过切片数量限制，是否出现循环依赖
`selectedCodeRefs`	路径是否存在于 code index，是否落在切片允许路径内，是否重复
`selectedCases`	是否来自 archive index，是否在召回预算内
`fingerprints`	selector input 和 output 是否能追溯，输入变化后是否需要重建

这层校验很关键。selector 可以是概率性的，但“引用是否合法、路径是否越界、上下文是否过期”不能靠模型自觉。否则有界上下文会变成另一种形式的自由发挥。

4.3 Working Set 如何被 Agent 使用

最终交给阶段 Agent 的不是 selector cache 本身，而是 JSONL working set。

`generated/context/prd.jsonl` 可以包含：

清单 4-4 Working Set 如何被 Agent 使用

```
{"type":"prd_overview","path":".team-agent/tasks/123456/prd/overview.md","summary":"新增奖品权益类型"}
{"type":"prd_slice","sliceId":"feature-01","path":".team-agent/tasks/123456/prd/feature-01.md","reason":"selected slice"}
```

`generated/context/code.jsonl` 可以包含：

清单 4-5 Working Set 如何被 Agent 使用

```
{"path":"src/main/java/com/example/prize/PrizeTypeService.java","reason":"selected code ref","symbols":["PrizeTypeService"]}
{"path":"src/test/java/com/example/prize/PrizeTypeServiceTest.java","reason":"local companion test","symbols":["PrizeTypeServiceTest"]}
```

这样做的好处是：Agent 先从有限证据开始，而不是默认全仓扫描；人可以检查 working set 是否选错；PRD 或代码变化后，可以通过 `sync-context` 重建；program mode 下可以生成 slice 专属 working set。

这里的 `selectedSlices` 不能只写一个切片 ID。更稳的做法是同时带上 `requirementRefs` 和必要的 `acceptanceRefs`，让后续阶段知道这个切片对应哪些需求契约和验收项。`selectedCodeRefs` 如果绑定了某个 slice，也要受该 slice 的路径边界约束，不能因为“看起来相关”就把全仓文件拉进来。

Working Set 不是最终答案

working set 只是阅读入口，不是真相源。它应该告诉 Agent “先读这里”，而不是让 Agent 永远不回原文。

表 4-5 Working Set 不是最终答案

场景	Agent 应该怎么做
写设计决策	先读 working set，再回看 `prd.md` 和相关规则原文
拆计划任务	使用 code working set，但验证真实文件和测试入口
修改代码	以 selected code refs 起步，必要时沿调用链扩展
判断验收	回到 PRD 切片和 acceptance，而不是只看摘要
判断历史案例	只作为参考，不能替代当前设计证据

如果 working set 摘要和原文冲突，以原文和已确认事实为准。

4.4 上下文失效和重建

有界上下文不是一次生成就永久有效。

表 4-6 上下文失效和重建

变化	影响	动作
PRD 范围变化	feature slices、验收项、selected slices 可能过期	重建 PRD companion 和 context
代码结构变化	code-index 和 code working set 可能过期	重建 code-index 和 code context
SPEC 更新	selectedSpecRefs 可能变化	重建 selector input 和 selector cache
历史案例新增	archive recall 可能变化	重跑 archive recall
设计方案调整	research artifact 或 working set 可能不再适用	判断复用或重建

这套框架里 `sync-context` 承担显式重建入口。它不是一个新的交付阶段，而是 builder rebuild orchestrator：当上游输入变化时，重新生成当前阶段需要的派生产物。

上下文质量怎么评估

有界上下文不是生成了 `prd.jsonl` 和 `code.jsonl` 就算完成，还要检查质量。

表 4-7 上下文质量怎么评估

检查项	好的表现	风险信号
PRD 覆盖	覆盖当前功能切片和验收项	只有任务标题，没有验收
代码覆盖	包含主实现入口、调用链、测试入口	只有 Controller 或只有接口定义
规则覆盖	命中业务规则和工程规则	只注入团队通用规范
历史案例	少量高相关弱参考	一次塞入大量旧任务
可重建性	输入变化后能重建	手工拼接，没人知道来源

如果设计阶段只看到 API 层代码，却看不到 service、repository、mapper、entity 或测试入口，就要警惕：上下文可能看起来相关，实际不足以支撑设计。

4.5 团队参考做法

其他团队可以先从最小上下文流水线开始：

示例 4-6 团队参考做法

```
.team-agent/tasks/{taskId}/
├── prd/index.json
├── context-selection.json
└── context/
    ├── prd.jsonl
    └── code.jsonl
```

第一阶段只做四件事：

1. PRD 先拆成 overview 和 feature list。
2. 代码先生成简单 index，记录路径、模块、符号、测试文件。
3. 人或模型从 feature list、rule list、code index 中选出候选上下文。
4. 生成 `prd.jsonl` 和 `code.jsonl` 给设计、计划、开发使用。

不要一开始就追求复杂语义召回。先让上下文选择可见、可审查、可重建。

4.6 检查清单

- Agent 是否还在默认全仓扫描?
- PRD 是否被拆成 overview 和 feature slices?
- 代码是否有可排序的 index?
- SPEC 是否按任务语义选择，而不是全量注入?
- selector 输出是否经过运行时校验，而不是直接信任模型?

- 历史案例是否按预算召回，而不是直接塞进会话?
- working set 是否可以被人审查?
- PRD 或代码变化后，是否能显式重建上下文?
- selector 输出是否有路径、理由和 fingerprint?

4.7 思考与实践

1. 你们现在给 AI 上下文时，是手动复制，还是有明确的选择和预算机制?
2. 最近一次 AI 选错代码位置，是因为上下文太少，还是因为无关上下文太多?
3. 如果要先做一个低配版 working set，你会优先放 PRD 切片、代码索引、业务规则，还是历史案例?

4.8 结尾

有界上下文不是限制 Agent，而是让 Agent 从正确的证据开始工作。团队越复杂，越不能依赖模型临场搜索；越要把需求、代码、规则和历史案例变成可选择、可重建、可审查的 working set。

下一篇进入阶段化交付：为什么不能让 AI 从一句话直接改代码，以及 PRD、设计、计划、总结如何变成稳定产物。

第5章　阶段化交付：别让 AI 从一句话直接改代码

个人使用 AI Coding 时，一句“帮我实现一下”可能足够。因为提出需求的人通常也熟悉代码，能在过程中判断模型是否跑偏。但团队交付不是这样。需求负责人、开发负责人、评审人、测试人可能不是同一个人，任务可能跨会话、跨天、跨工具继续。如果从一句话直接进入代码修改，后面很难回答：需求是否确认过，技术方案是否合理，计划是否可执行，验证是否覆盖，哪些经验应该沉淀。

阶段化交付的目标，不是把 AI Coding 变成繁重流程，而是把关键判断点产物化。团队不应该只审查聊天记录，而应该审查 PRD、设计、计划、diff、验证证据和总结。

> 一句话结论：AI Coding 要进入团队流程，必须先从聊天结果变成可审查产物。

常见误区

表 5-1　常见误区

误区	表现	后果
需求没确认就设计	Agent 基于工单摘要直接做方案	方案可能建立在错误需求上
设计没落文档	方案只存在聊天里	负责人难审查，后续难恢复
计划只是自然语言	“先改 A，再改 B，最后测试”	worker 无法稳定执行，reviewer 无法检查
开发和验证混在一起	代码改完才想起测试	失败原因难分类
交付后不总结	代码合了就结束	规则和经验无法进入下一次任务

阶段化交付要解决的是：每个关键阶段都有稳定 artifact，且阶段之间有输入输出契约。

5.1　阶段化不是多问几次确认

很多人一听阶段化，就担心流程变慢。真正拖慢团队的不是产物，而是无效确认：每写一章问一次，每做一步问一次，用户只能不断点头。

好的阶段化有两个特征：

表 5-2　阶段化不是多问几次确认

做法	说明
关键产物完整写出来再确认	让用户审 `design.md`、`plan.md`，不是审聊天摘要

做法	说明
确认点少但不可省	PRD、设计、计划、风险选择、规则处置和归档前置条件

也就是说，阶段化不是“多打断用户”，而是“把真正需要人判断的地方做成可审查产物”。

设计原则

第一，**阶段产物优先于聊天结论**。PRD、设计、计划、总结必须落文件，聊天只作为交互通道。

第二，**每个阶段有明确输入和输出**。设计消费 PRD 和上下文，计划消费设计，开发消费计划，验证消费 task 和 diff。

第三，**确认点少但关键**。不需要每个小步骤都打断用户，但 PRD、设计、计划、Spec disposition、归档前置条件必须可审查。

5.2 案例拆解：阶段化主干

完整需求流的主干流程可以拆成：

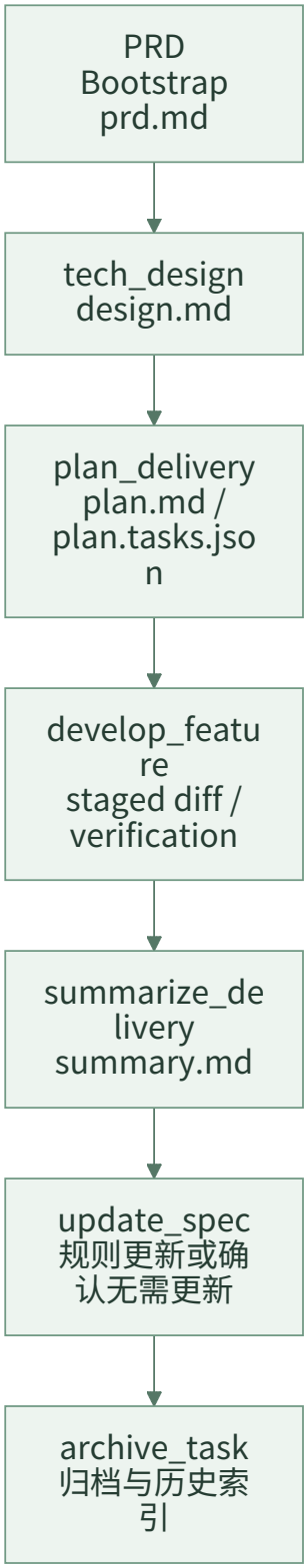


图 5-1 案例拆解：阶段化主干

任务目录中的稳定产物如下：

示例 5-1 案例拆解：阶段化主干

```
.team-agent/tasks/{taskId}/
├── prd.md
├── design.md
├── plan.md
├── plan.tasks.json
├── summary.md
├── evidence/verification-ledger.jsonl
└── context/archive-recall.json
```

其中 `prd.md`、`design.md`、`plan.md`、`summary.md` 是 stable artifacts。它们不是临时上下文，也不是缓存，而是团队审查和恢复任务时的主要入口。

每个阶段的进入和退出条件

阶段化真正有用，必须有 gate。否则只是多几个文件名。

表 5-3 每个阶段的进入和退出条件

阶段	进入条件	退出条件
PRD	有需求来源或人工输入	`prd.md` 已确认，范围和非目标清楚
Design	PRD、规则、代码 working set 可用	`design.md` 完整，重大风险和影响范围明确
Plan	设计已确认	`plan.md` 和 `plan.tasks.json` 对齐，每个 task 可执行
Develop	计划已确认	每个 task 有 diff、验证证据和 review 结果
Summary	开发完成并有验证证据	`summary.md` 包含范围、风险、验证和沉淀建议
Spec / Archive	summary 完成	规则处置明确，历史案例归档，active task 清理

这些 gate 可以先人工执行，后续再自动化。关键是团队要知道什么时候不能继续往下走。

PRD：需求真相源

PRD 阶段要回答：这次到底要做什么，来源是什么，是否已经被用户确认。

清单 5-2 PRD：需求真相源

```
# PRD：新增奖品权益类型

## 需求信息
```

```
- 需求 ID：123456
- 来源：Wiki
- 上下文模式：expanded

## 功能概览

支持新增权益类型识别、配置生成、查询展示，并保证存量权益类型行为不变。

## 功能切片

| Slice | Title | Acceptance |
|---|---|---|
| feature-01 | 新增权益类型识别 | 新权益类型可创建，存量类型行为不变 |

## 需求正文

这里保留用户确认后的需求正文。
```

PRD 阶段的关键不是写得漂亮，而是把需求真相和原始来源分开管理。原始来源可以追溯，确认后的 PRD 才进入后续交付。

PRD 里还要尽量写清楚非目标。很多 AI 跑偏不是因为不知道要做什么，而是不知道什么不做。

清单 5-3 PRD：需求真相源

```
## 非目标

- 本次不新增管理端配置入口。
- 本次不改退款补偿链路，只做回归验证。
- 本次不调整存量权益类型的展示文案。
```

Design：技术方案产物

技术设计阶段要回答：用什么方案实现，影响哪些模块，有哪些风险，如何保证稳定性。

`design.md` 至少应该包含：

清单 5-4 Design：技术方案产物

```
# 技术设计

## 需求概述
## 整体设计
## 功能模块设计
## 数据模型设计
## 稳定性设计
## 影响范围
## 风险与回滚
## 变更记录
```

设计阶段建议采用 artifact-first：先写完整 `design.md`，通过 design reviewer 后，再进行一次最终确认。复杂图示、表格和方案细节必须写入 artifact，而不是只靠聊天确认。

一个设计文档是否“够团队审”，可以看它是否回答了这几个问题：

- 影响哪些已有接口、调用方、数据表、配置、消息或定时任务？
- 哪些存量行为必须保持不变？
- 是否有灰度、回滚或兼容策略？
- 哪些地方需要回归测试？
- 哪些信息还没确认，是否明确标成待确认？

Plan：可执行计划

计划阶段要回答：怎么拆 task，每个 task 改哪些文件，怎么验证，失败和通过信号是什么。

一个简化的 `plan.tasks.json` 可以长这样：

清单 5-5 Plan：可执行计划

```
{
  "tasks": [
    {
      "id": "T1",
      "title": "新增权益类型枚举和映射",
      "featureId": "feature-01",
      "allowedPaths": [
        "src/main/java/com/example/prize/",
        "src/test/java/com/example/prize/"
      ],
      "verificationCommands": [
        "mvn test -Dtest=PrizeTypeServiceTest"
      ],
      "red": "新增测试先失败，证明当前不支持新权益类型",
      "green": "新增权益类型创建和查询测试通过",
      "regression": "存量权益类型回归测试通过",
      "expectedEvidence": ["测试命令输出", "verification-ledger 记录"],
      "acceptanceCriteria": ["新权益类型可创建", "存量权益类型行为不变"]
    }
  ]
}
```

计划必须可执行。只有“修改相关代码并补测试”这种描述，对 Agent 和 reviewer 都不够。

计划阶段最常见的问题是“看起来合理，但 worker 无法执行”。判断计划是否可执行，可以看每个 task 是否同时具备五件东西：`featureId`、`allowedPaths`、`verificationCommands`、`acceptanceCriteria`、`expectedEvidence`。缺任意一个，后续就容易变成自由发挥。

Develop：按 Task 执行

开发阶段不应该把完整需求再次交给一个万能 Agent。更稳定的方式是：控制平面读取 `plan.tasks.json`，一次只分配一个 task 给 worker。

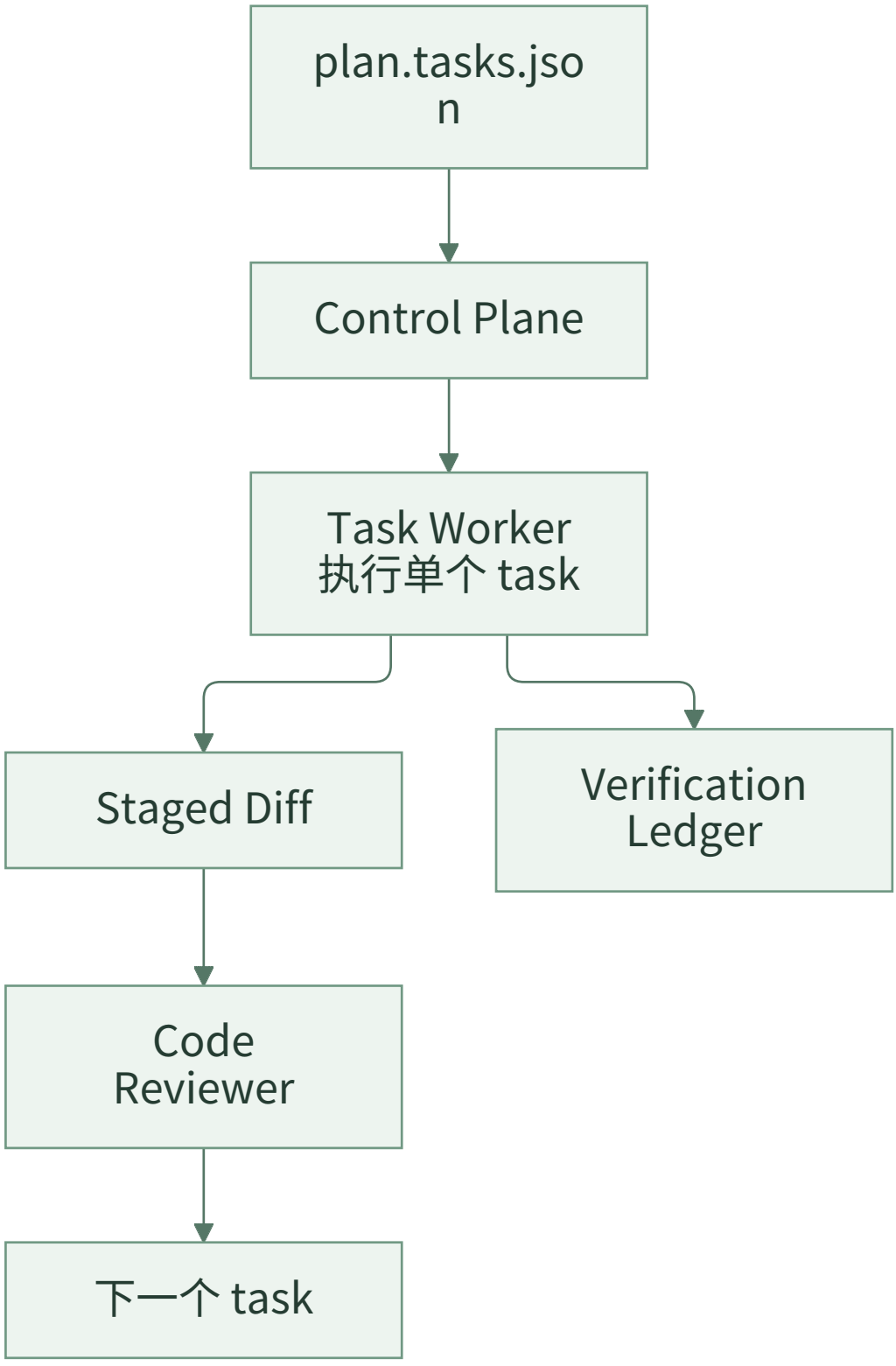


图 5-2 Develop：按 Task 执行

worker 只负责当前 task，不提交代码、不推进阶段、不修改任务状态。它返回 RED / GREEN / Regression 和验证证据后，控制平面再进行 staged diff review。

如果 worker 发现计划里的验证命令不存在，正确动作不是自己发明一个命令继续跑，而是把问题退回计划阶段。计划的命令、路径和验收标准属于 contract，执行端不能随意改 contract。

Summary、Spec 和 Archive

代码写完不代表交付结束。团队 AI Coding 的尾部动作同样重要：

- `summary.md` 总结本次交付范围、改动、验证、风险和未解决问题。
- `update_spec` 判断是否需要更新团队或应用规则。
- `archive_task` 把完成任务写入历史索引，供后续相似需求召回。

简化的 archive index entry 可以是：

清单 5-6 Summary、Spec 和 Archive

```
{
  "taskId": "123456",
  "title": "新增奖品权益类型",
  "domain": ["奖品", "权益类型"],
  "changedAreas": ["PrizeTypeService", "PrizeConfig"],
  "risks": ["存量类型兼容"],
  "summaryPath": ".team-agent/tasks/archive/123456/summary.md"
}
```

这个尾部动作解决的是经验沉淀问题：相似需求下一次出现时，不需要从零解释历史上下文。

5.3 团队参考做法

其他团队可以先采用最小阶段化流程：

```
需求确认 -> 设计 -> 计划 -> 开发 -> 验证 -> 总结 -> 规则维护 -> 归档
```

对应最小文件：

示例 5-7 团队参考做法

```
.team-agent/tasks/{taskId}/
├── prd.md
├── design.md
├── plan.md
├── plan.tasks.json
├── summary.md
└── evidence/verification-ledger.jsonl
```

第一阶段要做到：

1. PRD 未确认，不进入设计。
2. 设计未落文档，不进入计划。
3. 计划没有 task、路径、验证命令，不进入开发。
4. 验证没有证据，不算 task 完成。
5. 总结和规则处置未完成，不算交付完成。

5.4 检查清单

- 需求是否有确认后的 PRD artifact?
- 技术方案是否有稳定设计文档?
- 计划是否拆到 task，并包含文件范围和验证命令?
- worker 是否一次只执行一个 task?
- code review 是否基于 staged diff?
- 验证结果是否进入 ledger?
- 交付总结是否包含范围、风险和验证?
- 是否有规则更新或明确无需更新的动作?

5.5 思考与实践

1. 你们现在 AI 生成的内容里，哪些应该从聊天记录变成稳定 artifact?
2. 最近一次 AI Coding 任务，如果换人接手，`design.md` 和 `plan.md` 是否足够支撑继续开发?
3. 你们团队更缺 PRD 确认、设计审查、计划拆分，还是交付总结?

5.6 结尾

阶段化交付不是给 AI Coding 加流程负担，而是让团队能够审查、恢复、验证和沉淀。没有稳定产物，AI Coding 只能是聊天能力；有了稳定产物，AI 才能进入团队研发流程。

下一篇进入 Task 化开发：如何把一个大 diff 拆成能被逐个验证和审查的工程任务。

第6章　Task 化开发：把大 Diff 拆成可审查的工程任务

AI Coding 最容易让团队紧张的地方，是它能很快生成一个很大的 diff。这个 diff 可能同时改接口、改 service、改配置、改测试、改文档。模型觉得自己“完成了需求”，但评审人面对的是一大坨变化：不知道每一处改动对应哪个需求点，也不知道哪些测试覆盖了哪些行为。

团队级 AI Coding 不能让开发阶段变成“Agent 自由发挥”。更稳定的做法是先把计划拆成 task，每个 task 有明确范围、允许修改路径、验证命令、失败信号、通过信号和验收标准。开发时一次只执行一个 task，审查时只审当前 staged diff。

Task 化开发的目标，是把大需求拆成可执行、可验证、可审查的小交付单元。

> 一句话结论：团队不能接受一个不可控的大 diff，只能接受一组可验证的 task。

常见误区

表 6-1　常见误区

误区	表现	后果
计划只有自然语言	“实现核心逻辑并补测试”	worker 不知道边界，reviewer 无法判断是否完成
一次性交给 Agent 完整需求	Agent 自己决定改哪些文件	大 diff 难审查，风险集中
实现和测试拆成两个 task	先实现，后补测试	中间状态不可验证，测试容易漏
没有 allowed paths	Agent 可以改任意文件	容易越界修改和顺手重构
没有失败/通过信号	只写“运行测试”	无法判断验证是否覆盖新增行为

Task 化不是把任务拆碎，而是让每个小任务都能独立证明自己完成了什么。

6.1　什么样的 task 才算合格

很多计划看起来已经拆了 task，但其实只是把一句大需求拆成几句小描述。一个合格 task 至少要让 worker 和 reviewer 同时能回答下面的问题。

表 6-2 什么样的 task 才算合格

问题	必要字段
这个 task 服务哪个需求点?	`featureId` / `requirementRefs` / `acceptanceCriteria`
它必须遵守哪些长期规则?	`specRefs` / rule refs
它允许改哪些地方?	`allowedPaths`
它不应该改哪些地方?	scope / excluded paths / plan 说明
怎么证明新增行为有效?	GREEN 验证命令和 expected evidence
怎么证明存量行为没坏?	Regression 验证命令
失败时怎么判断是否阻塞?	baseline 和 failure classification

如果这些问题还需要 worker 临场判断，task contract 就还不够完整。

设计原则

第一，**每个 task 都要包含实现和验证**。不要把“写代码”和“补测试”拆成两个互相独立的 task。

第二，**每个 task 都要有修改边界**。allowed paths 和候选文件不是限制创造力，而是防止无关修改进入 diff。

第三，**每个 task 都要有证据要求**。RED、GREEN、Regression、Expected Evidence 和 Acceptance Criteria 必须写清楚。

6.2 案例拆解：plan.tasks.json

`plan_delivery` 阶段会产出 `plan.md` 和 `plan.tasks.json`。前者给人读，后者给运行时、worker 和 reviewer 使用。

```
.team-agent/tasks/{taskId}/
├── plan.md
├── plan.tasks.json
└── evidence/verification-ledger.jsonl
```

一个简化 task 如下：

清单 6-1 案例拆解：plan.tasks.json

```
{
  "id": "T1",
  "title": "新增权益类型枚举和配置解析",
  "featureId": "feature-01",
```

```
  "requirementRefs": ["REQ-001"],
  "specRefs": [".team-agent/spec/app/business/prize-type.md"],
  "dependsOn": [],
  "allowedPaths": [
    "src/main/java/com/example/prize/",
    "src/test/java/com/example/prize/"
  ],
  "verificationCommands": [
    "mvn test -Dtest=PrizeTypeServiceTest"
  ],
  "red": "新增权益类型解析测试在实现前失败",
  "green": "新增权益类型解析、配置生成测试通过",
  "regression": "存量权益类型解析回归测试通过",
  "expectedEvidence": [
    "RED 失败输出",
    "GREEN 通过输出",
    "verification-ledger 记录"
  ],
  "acceptanceCriteria": [
    "新权益类型可被识别",
    "配置生成包含新权益类型",
    "存量权益类型行为不变"
  ]
}
```

这个结构让 worker 和 reviewer 都有共同判断标准：改哪些范围、为什么改、遵守哪些规则、怎么证明。

`featureId` 只能说明 task 属于哪个功能切片，还不够细。`requirementRefs` 把 task 连接到机器可读的需求契约，`specRefs` 把 task 连接到长期业务规则或工程规则。这样 reviewer 看到 diff 时，不只是在问“是不是 feature-01”，还可以继续问“是否覆盖 REQ-001 的验收，是否符合对应 SPEC 的固定约束”。

Task 不应该怎么拆

下面这些拆法很常见，但会让 AI 开发不稳定。

表 6-3 Task 不应该怎么拆

错误拆法	为什么有问题	更好的拆法
`T1` 实现逻辑、`T2` 补测试	实现阶段不可验证，测试容易变成补丁	每个 task 同时包含实现和验证
`T1` 修改 `Controller`、`T2` 修改 `Service`	按代码层拆，脱离业务验收	按功能点和验收项拆
`T1` 处理所有兼容问题	范围太泛，worker 会自由发挥	明确具体兼容行为和验证命令
`T1` 重构相关代码	容易夹带无关改动	只有需求需要时才允许重构，并写清边界

错误拆法	为什么有问题	更好的拆法
T1 跑全量测试	纯验证 task 不产生交付闭环	把验证绑定到对应实现 task

好的 task 应该小到能审查，大到能闭环。它不是代码文件清单，而是一个可交付的小业务结果。

6.3 执行链路

Task 化开发的执行链路如下：

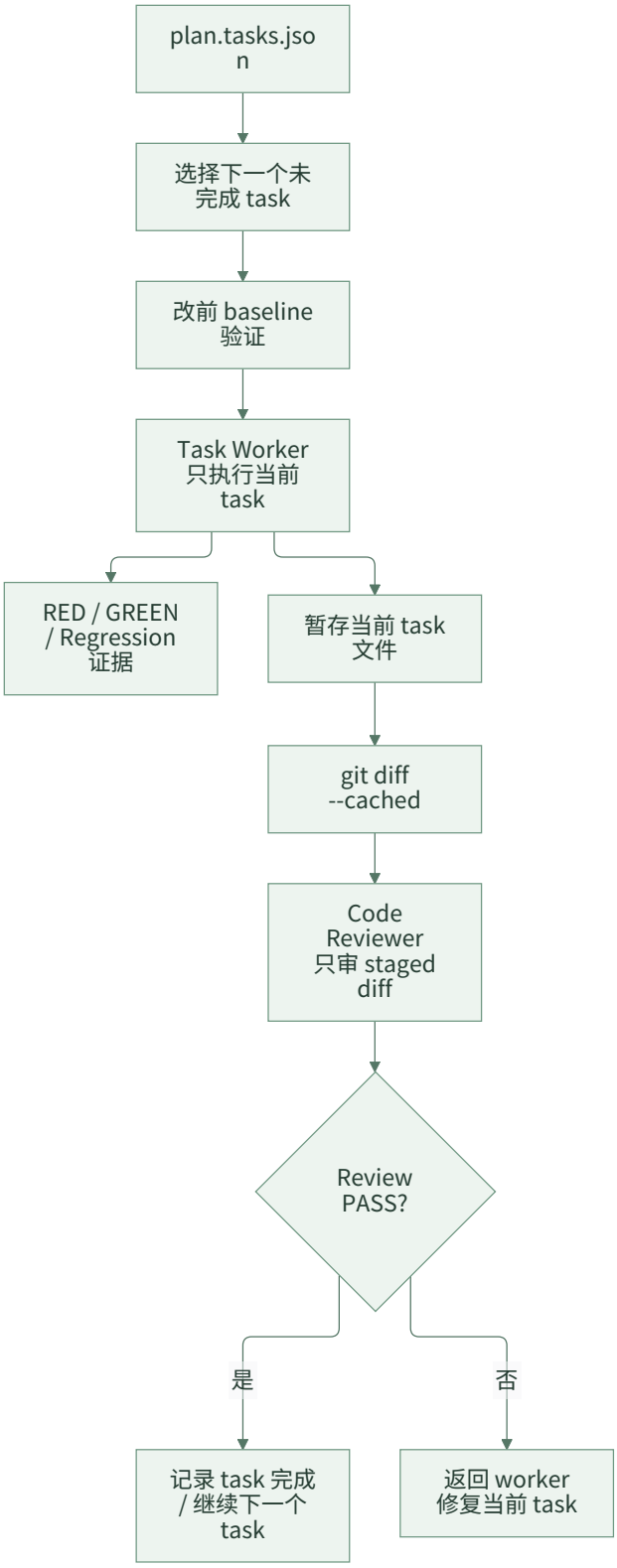


图 6-1 执行链路

这里有几个关键约束：

- worker 不推进 phase。
- worker 不提交代码。
- worker 不修改全局任务状态。

- 控制平面一次只分配一个 task。
- reviewer 只审 staged diff，不审整仓状态。

这些约束可以避免一个 Agent 既执行又调度又自评审。

Worker 返回结果也要结构化

worker 不应该只返回“完成了”。它至少要返回当前 task 的改动、验证和风险。

清单 6-2 Worker 返回结果也要结构化

```
{
  "taskId": "T1",
  "status": "completed",
  "changedFiles": [
    "src/main/java/com/example/prize/PrizeTypeService.java",
    "src/test/java/com/example/prize/PrizeTypeServiceTest.java"
  ],
  "verification": {
    "baseline": "passed",
    "green": "passed",
    "regression": "passed"
  },
  "evidencePaths": [
    ".team-agent/tasks/123456/evidence/verification-ledger.jsonl"
  ],
  "notes": [
    "存量权益类型行为通过回归测试"
  ]
}
```

如果证据不足，返回的也应该是结构化诊断，而不是继续扩范围修改。

清单 6-3 Worker 返回结果也要结构化

```
{
  "taskId": "T1",
  "status": "input_contract_unsatisfied",
  "missingEvidence": [
    "缺少查询展示路径的代码引用",
    "计划中的验证命令不存在"
  ]
}
```

6.4 Diff 审查看什么

Code reviewer 不只是看代码风格，而是把 diff 和 task contract 对齐。

表 6-4 Diff 审查看什么

审查项	问题
范围	diff 是否只修改 allowed paths?
需求对应	每个核心改动是否能回到 featureId、requirementRefs 和 acceptance?
规则对应	实现是否遵守 task 绑定的 specRefs?
测试覆盖	新增行为是否有正向测试?
回归保护	存量行为是否有回归验证?
越界修改	是否顺手重构或修改无关文件?
验证证据	ledger 是否记录对应命令和结果?
失败分类	失败是否被正确标记为新增失败或历史失败?

一个简化 review 结果可以是:

清单 6-4 Diff 审查看什么

```
{
  "taskId": "T1",
  "status": "FAIL",
  "findings": [
    {
      "severity": "major",
      "path": "src/main/java/com/example/prize/PrizeTypeService.java",
      "message": "新增类型只覆盖创建路径，查询展示路径未实现"
    }
  ],
  "requiredActions": [
    "补充查询展示路径实现",
    "补充对应回归测试"
  ]
}
```

存在 critical 或 major 问题时，当前 task 不应该进入完成状态。

Review 严重级别要有标准

为了避免 reviewer 的判断太随意，可以先约定严重级别。

表 6-5 Review 严重级别要有标准

级别	含义	处理
critical	会导致需求错误、数据风险、发布风险或安全问题	必须阻塞

级别	含义	处理
major	需求覆盖不完整、验证不足、越界修改、计划不一致	必须修复后重审
minor	命名、可读性、局部表达或非阻塞建议	可以记录，不阻塞

团队真正要拦的是 critical 和 major。minor 如果过度阻塞，会把 AI Coding 变成格式拉扯。

为什么不让 Worker 自己决定计划

很多团队会让一个 Agent 读 PRD 后直接实现。这样的问题是：计划、执行和验收都由同一个模型临场决定，团队无法判断它是否漏掉边界。

Task 化之后，职责更清楚：

表 6-6 为什么不让 Worker 自己决定计划

角色	负责	不负责
Planner	拆 task、定义范围和验证	写代码
Worker	执行单个 task	修改计划、提交代码、推进阶段
Reviewer	审 staged diff 和证据	自己修复代码
Control Plane	调度 task、记录状态	直接写业务代码

这种分工让问题更容易定位：计划错了回到 planner，执行错了回到 worker，审查不通过继续当前 task，不需要在一个万能 Agent 里混合修正。

6.5 团队参考做法

其他团队可以先把 `plan.tasks.json` 做到最小可用：

清单 6-5 团队参考做法

```
{
  "tasks": [
    {
      "id": "T1",
      "title": "任务标题",
      "featureId": "feature-01",
      "requirementRefs": ["REQ-001"],
      "specRefs": [".team-agent/spec/app/business/example.md"],
      "allowedPaths": ["src/main/java/...", "src/test/java/..."],
      "verificationCommands": ["mvn test -Dtest=..."],
      "acceptanceCriteria": ["验收项 1", "验收项 2"],
      "expectedEvidence": ["测试输出", "验证记录"]
```

```
    }
  ]
}
```

第一阶段要做到：

1. 每个 task 有唯一 ID。
2. 每个 task 绑定功能点、需求契约和验收项。
3. 每个 task 有 allowed paths。
4. 每个 task 有验证命令。
5. 每个 task 的 diff 单独审查。

第二阶段再加入 RED / GREEN / Regression、baseline-aware verification 和 reviewer JSON schema。

6.6 检查清单

- `plan.tasks.json` 是否能直接驱动 worker?
- 每个 task 是否同时包含实现和验证?
- 是否避免纯验证 task 和纯实现 task?
- 每个 task 是否有 `requirementRefs` 和必要的 `specRefs`?
- 是否定义 allowed paths?
- 是否定义验证命令和失败/通过信号?
- staged diff 是否只包含当前 task 改动?
- code review 是否基于 task contract?
- review FAIL 时是否回到当前 task，而不是继续后续任务?

6.7 思考与实践

1. 最近一次 AI 生成的大 diff，能否拆成 3-5 个可独立验证的 task?
2. 你们现在的 task 是否同时包含实现范围、验证命令和验收标准?
3. 如果 reviewer 只看 staged diff，哪些信息必须提前写进 `plan.tasks.json`?

6.8 结尾

Task 化开发的重点不是把需求拆得越碎越好，而是让每个工程任务都有边界、有证据、有审查入口。团队接受 AI Coding，不是接受一个不可控的大 diff，而是接受一组能被逐个验证和审查的 task。

下一篇进入验证治理：为什么 AI 说“测试失败无关”不够，以及如何用 baseline 和 ledger 建立证据链。

PART 03

第3篇　从一次成功到持续复用

一次测试通过不能代替稳定的验证方法。先建立基线并区分失败来源，再决定哪些交付经验值得进入规则和案例，避免积累无人维护的材料。

阅读重点

能说明完成判断的依据，并区分可复用规则与单次任务记录。

第7章　验证治理：AI 说“测试失败无关”是不够的

AI Coding 进入团队交付后，最敏感的问题之一是验证。模型很容易说：“这个测试失败看起来是历史问题”“这个失败应该和本次改动无关”“建议你本地再验证一下”。这些话在个人使用时也许可以接受，但在团队交付里不够。负责人需要看到证据：改前是什么状态，改后是什么状态，失败签名是否一致，失败是否进入本次改动文件，新增行为是否有通过证据。

验证治理的目标，是把“模型判断”变成“证据链判断”。如果无法证明失败是历史问题，就应该默认阻塞当前 task，而不是靠口头解释继续推进。

> 一句话结论：AI 说测试失败无关不算证据，改前 baseline 和改后 ledger 才算证据。

常见误区

表 7-1　常见误区

误区	表现	后果
只跑改后测试	不知道失败是否原本存在	无法区分历史失败和本次回归
失败靠自然语言解释	“应该无关”“看起来是环境问题”	质量判断不可审计
只看命令 exit code	不记录失败签名、数量、栈	下次无法比较
新增测试失败也标历史问题	只要老测试也失败就放过	新行为没有被验证
验证结果留在聊天里	后续 summary 和 review 找不到证据	交付链路断裂

验证治理要解决的是：每个 task 的质量判断都要能被复查。

7.1　为什么 baseline 不是“多跑一次测试”

很多团队觉得改前跑 baseline 很浪费时间。实际上，baseline 解决的是责任归因问题：没有改前状态，就无法判断改后失败是不是本次引入。

表 7-2　为什么 baseline 不是“多跑一次测试”

没有 baseline 时	有 baseline 时
只能说“看起来无关”	可以比较失败签名和数量
历史失败和新增失败混在一起	可以分类为 preexisting 或 new
reviewer 只能凭经验判断	reviewer 可以看证据

没有 baseline 时	有 baseline 时
summary 只能写“测试有失败”	summary 可以写清失败是否阻塞

baseline 不是为了追求每次都全量测试，而是为了让失败判断有证据。实际执行时可以先跑当前 task 相关命令，不一定每次都跑全仓。

设计原则

第一，**改前先跑 baseline**。没有改前状态，就没有资格判断改后失败是否历史存在。

第二，**失败分类要有硬条件**。只有满足签名一致、数量未增加、失败栈不进入改动文件、新增行为有通过证据等条件，才可以标记为历史失败。

第三，**证据必须落盘**。命令、时间、结果、失败分类、阻塞判断都要进入 ledger，而不是只留在聊天里。

7.2 案例拆解：baseline-aware verification

这套框架对每个 task 要求 baseline-aware verification。流程如下：

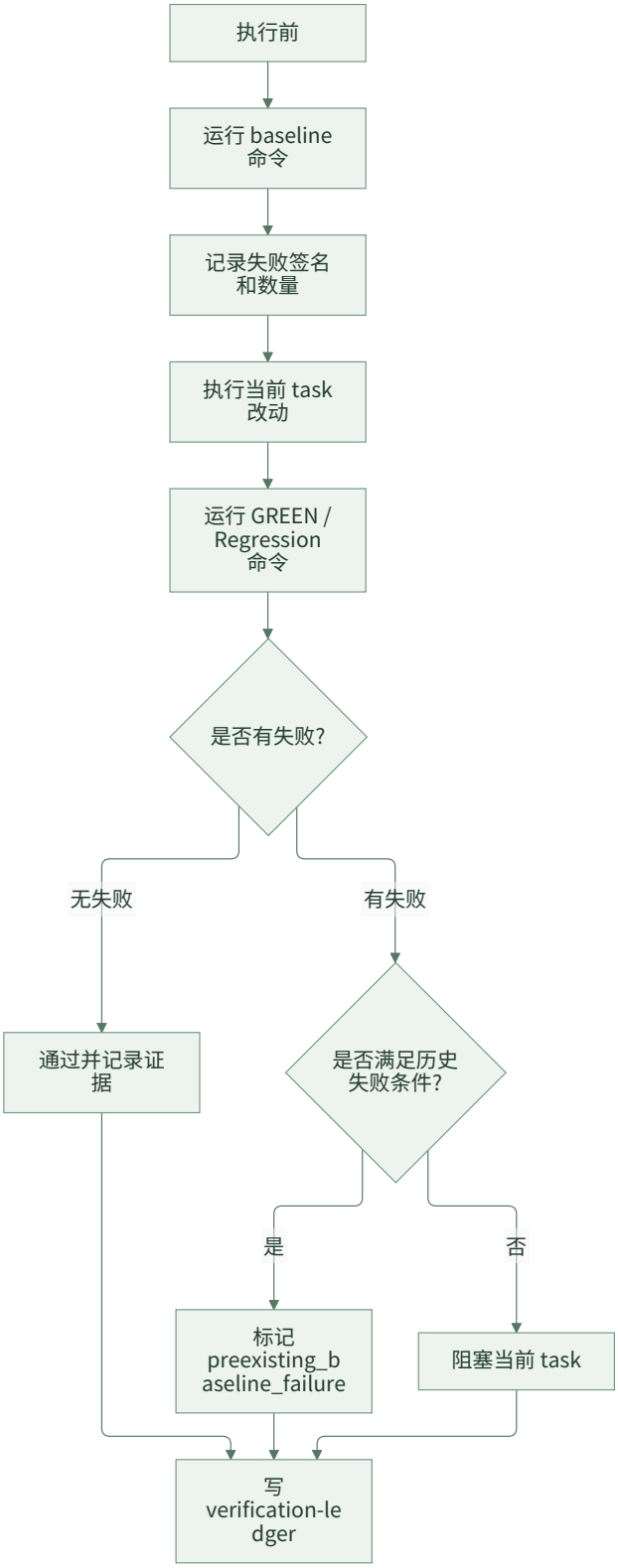


图 7-1 案例拆解：baseline-aware verification

关键条件可以写成表格：

表 7-3 案例拆解：baseline-aware verification

条件	含义
改前已失败	baseline 中已经存在同类失败

条件	含义
same signature	改后失败签名和改前一致
失败数量未增加	没有出现更多失败用例
失败不是新增测试	新增行为的测试不能被标成历史失败
栈不进入 changed files	失败栈没有指向本次改动文件
新增行为有通过证据	本次需求新增行为已经有正向通过测试

只要有一个关键条件不满足，就应该阻塞当前 task。

失败分类建议

团队可以先约定几类最小失败分类。

表 7-4 失败分类建议

分类	含义	是否阻塞
`passed`	命令通过	否
`preexisting_baseline_failure`	改前已失败，改后签名和数量不变，且不影响当前行为	否，但要记录 warning
`new_failure`	改前通过，改后失败	是
`increased_failure`	改前失败，改后失败数量增加	是
`changed_files_stack_hit`	失败栈进入本次改动文件	是
`new_test_failed`	新增或修改测试失败	是
`uncertain_failure`	证据不足以判断	是

最重要的一条规则是：**不确定默认阻塞**。团队可以接受历史失败不阻塞，但不能接受无法分类的失败被放过。

7.3 Verification Ledger 长什么样

验证证据进入 `evidence/verification-ledger.jsonl`。JSONL 的好处是可以追加，适合记录多个 task、多个命令和多次重试。

```
.team-agent/tasks/{taskId}/evidence/
└── verification-ledger.jsonl
```

一条简化记录可以是：

清单 7-1 Verification Ledger 长什么样

```
{
  "taskId": "T1",
  "phase": "develop_feature",
  "command": "mvn test -Dtest=PrizeTypeServiceTest",
  "stage": "green",
  "status": "failed",
  "exitCode": 1,
  "failureSignature": "PrizeTypeServiceTest.shouldKeepLegacyType",
  "baselineSignature": "PrizeTypeServiceTest.shouldKeepLegacyType",
  "failureCountBefore": 1,
  "failureCountAfter": 1,
  "changedFiles": [
    "src/main/java/com/example/prize/PrizeTypeService.java"
  ],
  "classification": "preexisting_baseline_failure",
  "blocksCurrentTask": false,
  "positiveEvidence": [
    "PrizeTypeServiceTest.shouldCreateNewPrizeType passed"
  ]
}
```

如果失败无法分类，就应该记录为阻塞：

清单 7-2 Verification Ledger 长什么样

```
{
  "taskId": "T2",
  "command": "mvn test -Dtest=PrizeQueryTest",
  "stage": "regression",
  "status": "failed",
  "classification": "new_or_uncertain_failure",
  "blocksCurrentTask": true,
  "reason": "失败栈进入本次改动文件，且 baseline 中不存在相同签名"
}
```

ledger 的价值不是替代 CI，而是让 AI Coding 任务本身有本地验证证据。

Ledger 里最少要记录什么

最小 ledger 不需要一开始很复杂，但以下字段建议保留。

表 7-5 Ledger 里最少要记录什么

字段	作用
`taskId`	关联当前 plan task
`command`	复现验证动作
`stage`	区分 baseline、green、regression
`status`	命令通过或失败

字段	作用
`failureSignature`	用于和 baseline 比较
`changedFiles`	判断失败是否进入改动范围
`classification`	失败分类
`blocksCurrentTask`	是否阻塞当前 task
`positiveEvidence`	新增行为是否有通过证据

只写“测试通过”四个字没有审计价值。能复现的命令、能比较的失败签名、能解释的阻塞判断，才是团队需要的证据。

7.4 验证和 Review 的关系

验证不是 worker 自己说了算。worker 返回验证证据后，控制平面再把 staged diff 交给 code reviewer。

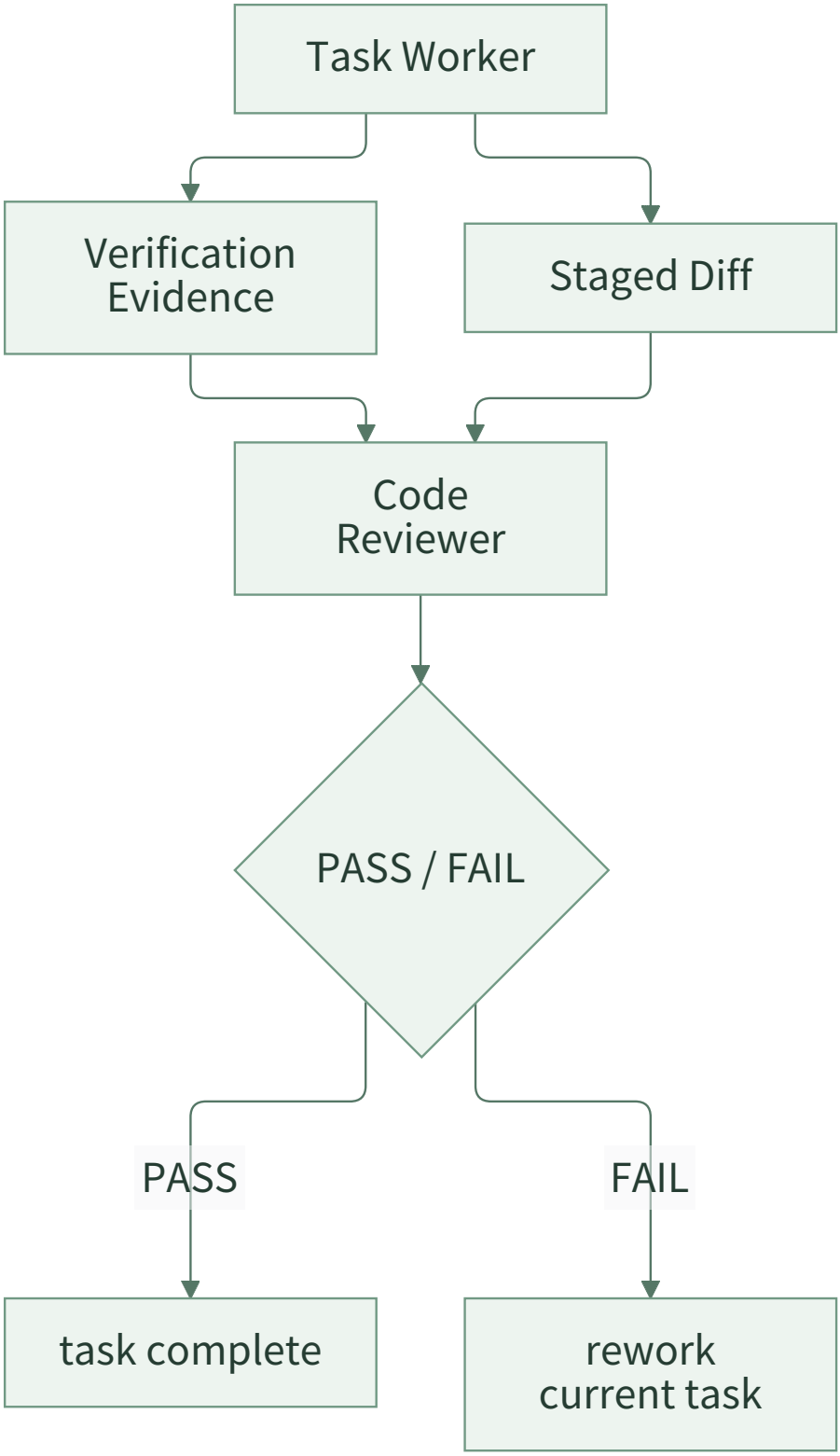


图 7-2 验证和 Review 的关系

Reviewer 要检查两件事：

- diff 是否满足 task contract。
- 验证证据是否足够支撑当前 task 完成。

如果新增行为没有正向通过证据，即使回归测试没有失败，也不应该轻易通过。

哪些情况不能让 AI 解释放过

下面几种情况应该默认阻塞，除非有非常明确的人为豁免。

表 7-6 哪些情况不能让 AI 解释放过

情况	原因
新增测试失败	说明新行为没有被证明
改前通过、改后失败	高概率是本次回归
失败数量增加	历史问题恶化
失败栈进入 changed files	和本次改动高度相关
验证命令不存在或跑错模块	计划 contract 不成立
没有正向通过证据	无法证明需求完成
分类为 uncertain	证据不足

AI 可以辅助分析失败，但不能替代团队的阻塞规则。

7.5 团队参考做法

其他团队可以先做一个最小 verification ledger：

清单 7-3 团队参考做法

```
{
  "taskId": "T1",
  "command": "npm test -- user-service",
  "stage": "baseline|green|regression",
  "status": "passed|failed",
  "failureSignature": "optional",
  "classification": "passed|new_failure|preexisting_failure|uncertain_failure",
  "blocksCurrentTask": true
}
```

第一阶段要求：

1. 每个 task 改前先跑 baseline。
2. 改后至少跑 GREEN 和 Regression。
3. 失败必须分类。
4. 不确定失败默认阻塞。
5. ledger 必须进入 summary 和 review 输入。

第二阶段再增强失败签名、changed files 栈分析、CI 结果关联和趋势统计。

7.6 检查清单

- 每个 task 是否有改前 baseline?
- 改后失败是否能和 baseline 比较?
- 是否记录失败签名和失败数量?
- 新增测试失败是否默认阻塞?
- 失败栈进入 changed files 时是否阻塞?
- 新增行为是否有正向通过证据?
- 验证记录是否进入 ledger?
- summary 是否引用验证证据，而不是只写“测试通过/失败”？

7.7 思考与实践

1. 你们团队遇到测试失败时，怎么判断是历史失败还是本次回归?
2. 如果要求每个 task 改前都跑 baseline，你们最大的阻力是耗时、环境不稳定，还是测试命令不清楚?
3. 哪类失败在你们团队里应该默认阻塞，而不能让 AI 口头解释放过?

7.8 结尾

团队 AI Coding 不能依赖“看起来无关”的判断。验证治理的核心是证据链：改前状态、改后状态、失败分类、阻塞判断和正向通过证据。只有验证证据可审计，团队才敢把 AI Coding 纳入真实交付流程。

下一篇进入资产沉淀：如何让每次 AI 交付反哺下一次任务，而不是让经验继续留在聊天记录里。

第8章　资产沉淀：让每次交付反哺下一次任务

很多 Agent 建设会卡在一个问题上：内容越来越多，但越来越不可信。Prompt 写了一版又一版，Skill 补了一堆，Wiki 也有不少规范，但真实业务规则、历史踩坑、代码约束和验证经验仍然靠人记。过一段时间，Agent 看起来更复杂了，实际知识却过期了。

团队级 AI Coding 不能只靠人工定期维护 Prompt。更可持续的方式，是把资产沉淀嵌入每次交付：需求做完后，哪些规则值得沉淀，哪些案例值得归档，哪些研究结论后续还会复用，哪些验证经验应该保留，都要成为交付终态的一部分。

资产沉淀飞轮的目标，是让每次交付都反哺下一次任务。

> 一句话结论：Agent 内容不能只靠人工维护，真实交付才是最好的知识更新来源。

常见误区

表 8-1　常见误区

误区	表现	后果
代码合了就结束	没有 summary、没有规则处置	经验留在人脑和 PR 里
定期人工整理 Prompt	依赖负责人记得维护	容易漏，且无法覆盖真实变化
历史案例留在聊天里	下次任务无法自动召回	相似问题反复解释
研究结论不落盘	中间件、框架、调用链分析只在对话中	后续计划和开发无法复用
所有经验都写进规则	临时事实变成长期约束	SPEC 膨胀且误导后续任务

资产沉淀的关键，是区分哪些是长期规则，哪些是历史案例，哪些只是本次任务事实。

8.1　资产沉淀不是“把东西都保存”

很多团队的知识库越来越乱，是因为只做保存，不做分层。一次交付结束后，至少要把信息分成四类。

表 8-2　资产沉淀不是“把东西都保存”

信息类型	应该去哪里	后续怎么用
长期规则	SPEC	后续任务按语义注入
一次交付案例	archive index + summary	相似任务弱召回
探索性研究	research artifact	后续阶段或类似任务按范围复用

信息类型	应该去哪里	后续怎么用
验证经验	verification ledger + summary	review、复盘和失败分类参考

如果所有东西都写进 Prompt 或 Wiki，最后会变成没人敢删、Agent 也不知道怎么读的“知识堆”。

设计原则

第一，**沉淀动作进入主流程**。总结、规则处置和归档不能依赖人想起来，而要成为交付终态。

第二，**规则、案例、研究、验证分层保存**。不同资产有不同 owner、生命周期和复用方式。

第三，**历史召回是弱引用**。历史案例可以提示相似风险，但不能覆盖当前 PRD、当前 SPEC 和当前设计事实。

8.2 案例拆解：交付尾链

完整需求流在开发完成后不应该直接结束，而是进入：

```
summarize_delivery -> update_spec -> archive_task
```

这条尾链对应的资产如下：

示例 8-1 案例拆解：交付尾链

```
.team-agent/tasks/{taskId}/
├── summary.md
├── evidence/verification-ledger.jsonl
├── research/
│   ├── explore/
│   └── middleware/
└── context/archive-recall.json

.team-agent/spec/
├── team/
└── app/

.team-agent/tasks/archive/
├── index.jsonl
└── route-index.json
```

沉淀飞轮可以这样理解：

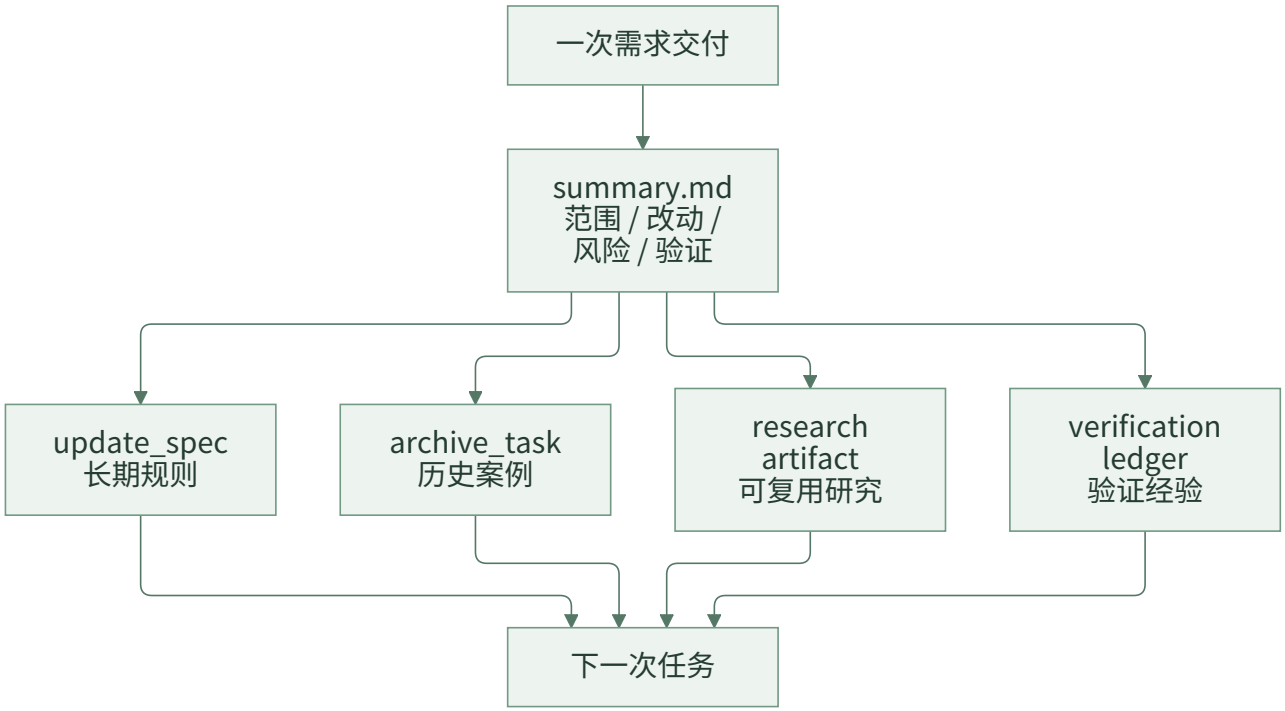


图 8-1 案例拆解：交付尾链

Summary 写什么

`summary.md` 不是流水账，它要把一次交付压缩成后续可消费的事实。

清单 8-2 Summary 写什么

```
# 交付总结

## 交付范围

- 新增权益类型识别和配置生成。
- 补充查询展示回归。

## 主要改动

| Area | Files | Reason |
|---|---|---|
| 领域逻辑 | PrizeTypeService | 新增权益类型识别 |
| 测试 | PrizeTypeServiceTest | 覆盖新增类型和存量类型 |

## 验证证据

- `mvn test -Dtest=PrizeTypeServiceTest` passed
- verification ledger: `evidence/verification-ledger.jsonl`

## 风险和后续关注

- 存量权益类型兼容仍需在集成环境观察。
```

```
## 可沉淀规则

- 新增权益类型必须覆盖创建、查询、展示路径。
```

summary 是 update_spec 和 archive_task 的输入。写得越清楚，后续沉淀越稳定。

Summary 不应该写成流水账

一份好的 summary 要面向未来任务，而不是复述本次聊天过程。

表 8-3 Summary 不应该写成流水账

不推荐	推荐
“完成了相关代码修改”	“完成新增权益类型识别和查询展示兼容，涉及 X/Y 模块”
“测试有一个失败，应该无关”	“回归命令失败签名与 baseline 一致，分类为 preexisting_baseline_failure”
“后续注意兼容”	“存量权益类型展示路径需要在类似需求中固定回归”
“无需更新规则”但不说明原因	“本次只调整一次性配置，不形成可复用规则”

summary 写得越像未来任务的索引，资产飞轮越容易转起来。

8.3 哪些内容进 SPEC

只有可复用规则才进入 SPEC。判断标准是：下一次类似需求是否应该让 Agent 自动看到这条规则。

表 8-4 哪些内容进 SPEC

内容	是否进 SPEC	原因
“新增权益类型必须覆盖创建、查询、展示路径”	是	可复用业务开发规则
“本次需求 ID 是 123456”	否	只属于当前任务
“这次测试环境数据库不可用”	否	临时环境事实
“配置变更必须补充灰度开关”	是	可复用工程约束
“某个历史 bug 的修复路径”	不一定	更适合作为 archive case，除非抽象成规则

`update_spec` 阶段应该允许三种结果：

1. 更新已有 SPEC。
2. 推荐新增 focused Spec。

3. 明确本次无需更新 SPEC。

第三种也很重要。不是每个需求都应该产生新规则，否则 SPEC 会膨胀。

Spec disposition 要成为交付动作

交付结束时，建议明确记录一次规则处置结果。

清单 8-3 Spec disposition 要成为交付动作

```
{
  "taskId": "123456",
  "specDisposition": {
    "decision": "update_existing_spec",
    "target": ".team-agent/spec/app/business/prize-type.md",
    "reason": "新增权益类型需要固定创建、查询、展示回归要求"
  }
}
```

如果无需更新，也要写清楚：

清单 8-4 Spec disposition 要成为交付动作

```
{
  "decision": "no_update_required",
  "reason": "本次只修正文案，不产生可复用业务或工程规则"
}
```

这能防止“规则维护”变成一句空话。

8.4 Archive 如何被召回

归档不是把任务目录挪走这么简单，而是要形成可检索的历史案例索引。

简化的 archive index entry：

清单 8-5 Archive 如何被召回

```
{
  "taskId": "123456",
  "title": "新增奖品权益类型",
  "domain": ["奖品", "权益类型"],
  "changedAreas": ["PrizeTypeService", "PrizeConfig"],
  "middleware": [],
  "risks": ["存量类型兼容"],
  "summaryPath": ".team-agent/tasks/archive/123456/summary.md"
}
```

下一次任务进入设计或计划前，可以根据当前 PRD、selected slices、SPEC 和 code refs 召回少量历史案例，生成：

```
.team-agent/tasks/{taskId}/context/archive-recall.json
```

历史召回要遵守几个边界：

- fail-open：召回失败不阻塞主流程。
- budgeted：只召回少量最相关案例。
- weak reference：历史案例只能作为参考，不能覆盖当前需求事实。
- 不写回 `task.json`：召回结果是上下文，不是任务状态。

一个好的 archive card 不需要很长，但要足够可检索：

清单 8-6 Archive 如何被召回

```
{
  "caseId": "case-20260501-prize-type",
  "title": "新增权益类型支持",
  "domains": ["奖品", "权益类型"],
  "changedAreas": ["配置解析", "查询展示", "回归验证"],
  "riskSignals": ["存量兼容", "枚举映射"],
  "useWhen": ["新增或调整权益类型", "查询展示兼容需求"],
  "doNotUseWhen": ["仅修改文案", "不涉及权益类型识别"],
  "summaryPath": ".team-agent/tasks/archive/123456/summary.md"
}
```

`useWhen` 和 `doNotUseWhen` 很重要。没有适用边界的历史案例，容易误导后续任务。

Research Artifact 的位置

复杂需求经常需要额外研究，例如中间件接入、调用链梳理、框架约束、历史实现比较。研究结果如果只留在聊天里，后续 phase 和后续任务都无法复用。

这套框架把研究结果放到：

```
.team-agent/tasks/{taskId}/research/
├── explore/
└── middleware/
```

研究 artifact 应该包含：

- 研究问题。
- 结论摘要。
- 引用文件和路径。
- 适用范围。
- 失效条件。

当需求范围或设计方案变化时，要判断研究结论是否仍然可复用；不能默认沿用旧研究。

资产也会失效

沉淀不是永久可信。规则、案例和研究结论都需要失效条件。

表 8-5 资产也会失效

资产	常见失效原因	应对
SPEC 规则	业务流程变化、技术栈升级	更新 Source Reference 和规则正文
Archive case	当前需求边界不同	作为弱参考，不直接套用
Research artifact	依赖版本变化、方案变更	标记不可复用或重新研究
Verification evidence	测试命令变化、环境变化	只作为历史证据，不替代当前验证

团队越依赖资产复用，越要明确资产的适用范围和失效条件。

8.5 团队参考做法

其他团队可以先建立四类资产：

示例 8-7 团队参考做法

```
.team-agent/
├── spec/
├── tasks/{taskId}/summary.md
├── tasks/{taskId}/research/
├── tasks/{taskId}/evidence/verification-ledger.jsonl
└── archive/index.jsonl
```

交付结束时固定问四个问题：

1. 这次交付有没有可复用规则？如果有，更新 SPEC。
2. 这次交付有没有可参考案例？如果有，写 archive index。
3. 这次交付有没有可复用研究？如果有，落 research artifact。
4. 这次验证有没有历史失败或特殊证据？如果有，进入 ledger 和 summary。

先做人工确认版沉淀，再逐步让 Agent 推荐更新内容。不要一开始就让模型自动改所有规则。

8.6 检查清单

- 交付完成后是否一定生成 summary？
- summary 是否包含范围、改动、验证、风险和可沉淀规则？
- SPEC 更新是否有明确入口？
- 是否允许“本次无需更新 SPEC”？
- 历史案例是否有 index，而不是只存完整目录？

- archive recall 是否按预算召回?
- research artifact 是否落盘?
- 资产是否区分长期规则、历史案例、临时事实?

8.7 思考与实践

1. 你们最近一次 AI Coding 的经验，后来进入了哪里？SPEC、Wiki、PR 评论，还是只留在聊天里?
2. 哪些内容应该沉淀成长期规则，哪些只适合放进历史案例?
3. 如果每次交付结束都固定做一次资产处置，你们团队会担心流程变重，还是担心规则质量不好?

8.8 结尾

Agent 内容不能只靠人工维护。团队要让每次真实交付都成为知识更新来源：规则进 SPEC，案例进 archive，研究进 artifact，验证进 ledger。这样 AI Coding 才会越用越贴近团队，而不是越用越依赖某几个人的经验。

下一篇进入 Agent 分层：为什么不要让一个万能 Agent 同时负责调度、设计、开发、验证和评审。

PART 04

第4篇　从单 Agent 到团队协作

参与者变多以后，需要明确职责、共享语义和接管位置。这一篇讨论 Agent 分层、工具适配，以及人怎样审查和接手任务。

阅读重点

能判断哪些状态必须跨工具保存，哪些差异可以留在适配器。

第9章 Agent 分层：不要让一个万能 Agent 包办所有事

团队做 AI Coding Agent 时，很容易从“一个超级 Agent”开始：它读需求、查代码、写设计、拆计划、改代码、跑测试、评审自己、总结归档。这个方案看起来简单，但真实运行时很快会失控。一个 Agent 既当调度者又当执行者，还要自我评审，职责混在一起后，问题很难定位：到底是上下文不够、计划不清、执行跑偏，还是评审没拦住?

团队级 Agent 架构需要分层。控制平面负责流程推进，builder 负责上下文构建，research 负责补充研究，phase agent 负责阶段产物，worker 负责单个 task，reviewer 负责审查。每一层只做自己的事，才能让系统可恢复、可验证、可演进。

> 一句话结论：**一个万能 Agent 看起来简单，真正稳定的是分层职责和清晰边界。**

常见误区

表 9-1 常见误区

误区	表现	后果
一个 Agent 包办所有事	需求、设计、开发、验证、评审全在一个 Prompt	失败原因难定位
leaf agent 调 subagent	执行单元又开始调度其他执行单元	控制权分散，状态不可控
builder 推进流程	上下文生成工具顺手改任务状态	派生产物和真相源混淆
reviewer 自己修代码	审查者同时修改代码	评审边界失效
research 留在对话里	探索结果不落盘	后续阶段无法复用

分层不是为了复杂，而是为了让每个失败点都能被定位和修复。

9.1 什么时候必须分层

不是所有团队第一天就需要一堆 Agent。可以用下面几个信号判断是否必须拆职责。

表 9-2 什么时候必须分层

信号	说明
一个 Agent 经常边写边改计划	计划和执行混在一起

信号	说明
测试失败后 Agent 自己解释并继续提交	执行和质量判断混在一起
设计方案缺证据但 Agent 自己全仓搜索	设计和上下文构建混在一起
review 只是 Agent 自我总结	审查边界失效
任务卡住后无法恢复	控制面和执行面混在一起

只要出现两三个信号，就说明“万能 Agent”已经不适合团队交付。

设计原则

第一，**控制和执行分离**。控制平面决定流程推进和调度，leaf agent 只完成被分配的阶段或 task。

第二，**构建和消费分离**。builder 生成上下文，phase agent 消费上下文，不要让 phase agent 临场重建运行时产物。

第三，**执行和评审分离**。worker 负责改代码和验证，reviewer 负责审查 diff 和证据。

9.2 案例拆解：职责分层

这套框架的职责分层可以概括为：

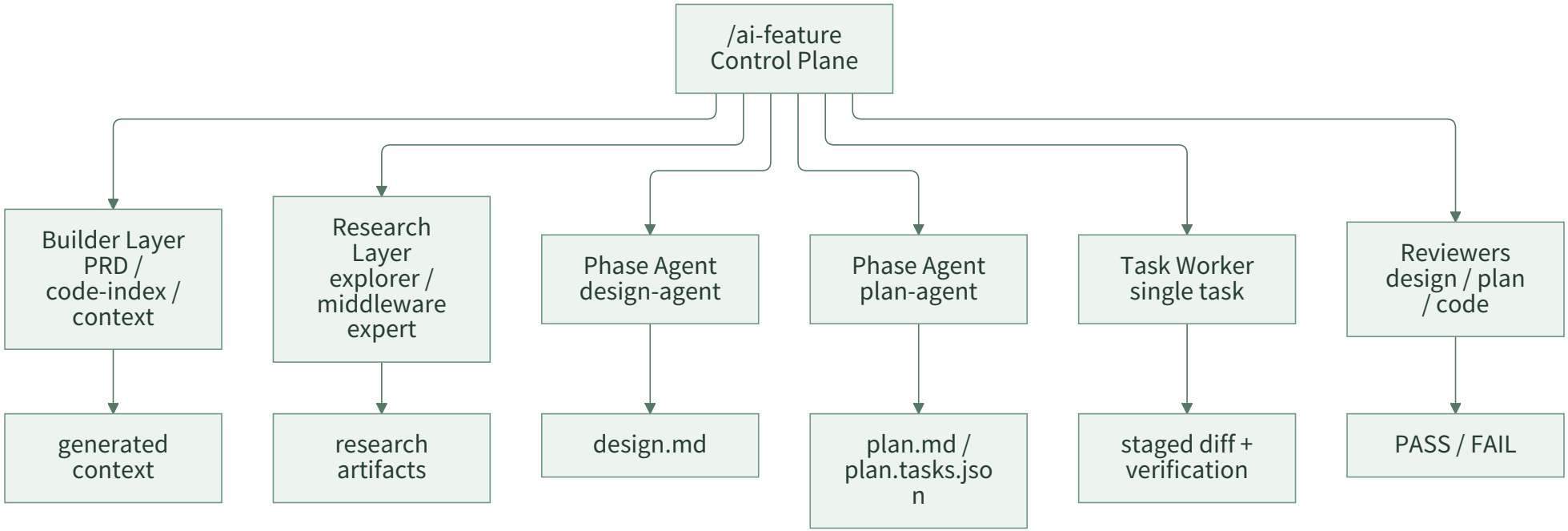


图 9-1　案例拆解：职责分层

几类角色的边界如下：

表 9-3 案例拆解：职责分层

层级	负责	不负责
Control Plane	phase 推进、task 调度、用户确认、状态变更	直接写业务代码
Builder Layer	PRD 构建、代码索引、上下文重建	推进 phase、做设计决策
Research Layer	仓库探索、中间件研究、结论落盘	修改任务状态、直接实现需求
Designer	写 `design.md`	拆执行 task、改代码
Planner	写 `plan.md` 和 `plan.tasks.json`	现场写代码
Task Worker	执行单个 task、产出验证证据	提交代码、推进 phase、调 subagent
Reviewer	审查 design、plan 或 staged diff	自评审、自修复

这套分工让系统出现问题时能回到明确 owner：设计问题回到 designer，计划问题回到 planner，执行问题回到 worker，上下文缺失回到 builder 或 research。

控制面最小职责

控制面不需要很复杂，但要掌握几件不能下放的事情：

表 9-4 控制面最小职责

职责	为什么不能下放给 worker
读取和更新任务状态	状态真相源只能有一个 owner
决定进入哪个阶段	避免执行单元越级推进
处理用户确认和阻塞	用户决策要回到主流程
调度 research 和 reviewer	防止 leaf agent 自己找证据、自我放行
暂存和提交当前 task diff	控制提交范围，避免混入无关改动

如果 worker 既能改代码又能推进阶段又能提交，后续一旦出问题，很难知道是计划错、执行错还是评审漏。

9.3 Leaf Agent 为什么不能调度

Leaf agent 指的是被控制平面分配具体任务的执行单元。它应该 path-first 工作：只接收路径、ID、短摘要、allowed paths、验证命令和 research artifact paths。

一个 worker handoff 可以是：

清单 9-1 Leaf Agent 为什么不能调度

```
{
  "taskId": "T1",
  "planPath": ".team-agent/tasks/123456/plan.tasks.json",
  "prdContextPath": ".team-agent/tasks/123456/generated/context/prd.jsonl",
  "codeContextPath": ".team-agent/tasks/123456/generated/context/code.jsonl",
  "allowedPaths": [
    "src/main/java/com/example/prize/",
    "src/test/java/com/example/prize/"
  ],
  "verificationCommands": [
    "mvn test -Dtest=PrizeTypeServiceTest"
  ]
}
```

worker 如果发现证据不足，不应该自己调 explorer 或改计划，而应该返回结构化诊断：

清单 9-2 Leaf Agent 为什么不能调度

```
{
  "status": "input_contract_unsatisfied",
  "taskId": "T1",
  "diagnostics": {
    "missingEvidence": [
      "缺少 PrizeConfig 写入路径的 code ref",
      "缺少权益类型配置来源说明"
    ]
  }
}
```

控制平面再决定是重建上下文、调用 research、回到计划，还是进入变更流程。

`input_contract_unsatisfied` 是好事

很多团队会把 Agent 返回“证据不足”看成失败。其实这恰恰是系统稳定的表现。它说明执行单元没有硬猜，也没有越权扩范围。

常见的证据不足可以这样路由：

表 9-5 input_contract_unsatisfied 是好事

缺什么	回到哪里
PRD 或上下文派生产物缺失	context builder / sync-context
代码调用链或测试入口缺失	research / explorer
中间件规则或接入约束缺失	专项研究或已有规则
验证命令不存在	plan_delivery

缺什么	回到哪里
业务事实未确认	用户确认 / 设计前事实

分层之后，Agent 不需要装作什么都知道。它只需要把缺口说清楚。

Research Layer 如何落盘

研究层的输出不能只留在对话里。这套框架把研究结论落到：

```
.team-agent/tasks/{taskId}/research/
├── explore/
│   └── {research-id}.md
└── middleware/
    └── {research-id}.md
```

研究产物应该包含：

- 研究问题。
- 结论摘要。
- 代码引用。
- 风险和适用范围。
- 后续阶段如何使用。

这样 planner 和 worker 可以通过路径消费研究结论，而不是依赖某次聊天上下文。

9.4 团队参考做法

其他团队可以先定义最小角色矩阵：

表 9-6 团队参考做法

角色	最小职责
Orchestrator	维护任务状态、推进阶段、分配 task
Context Builder	生成 PRD/code/rule working set
Designer	输出技术设计
Planner	输出计划和 tasks
Worker	执行单个 task
Reviewer	审查产物和 diff
Summarizer	总结和沉淀建议

第一阶段不要把每个角色都做成独立 Agent，可以先用命令、脚本和人工确认组合实现。关键是职责边界先稳定。

不是所有角色都要一开始产品化

一个实用落地顺序是：

1. 先拆 reviewer：让审查者和执行者分开。
2. 再拆 planner：让计划先形成 task contract。
3. 再拆 context builder：让上下文选择可见、可重建。
4. 最后拆 research：把复杂探索结论沉淀成 artifact。

这样不会一上来制造很多角色，但能逐步把最危险的职责混杂点拆开。

9.5 检查清单

- 是否存在一个 Agent 同时设计、开发、验证、评审?
- 控制平面是否是唯一推进状态的角色?
- builder 是否只生成上下文，不推进 phase?
- worker 是否一次只执行一个 task?
- reviewer 是否只审查不修改?
- research 结论是否落盘?
- leaf agent 证据不足时是否返回诊断，而不是自行扩展范围?
- 失败时能否定位到具体层级?

9.6 思考与实践

1. 你们现在的 Agent 更像万能助手，还是已经区分设计、计划、开发、评审职责?
2. 哪些职责绝对不应该让同一个 Agent 同时承担?
3. 如果只能先拆一个角色，你会优先拆 reviewer、planner，还是 context builder?

9.7 结尾

Agent 分层的意义不是增加角色数量，而是让团队 AI Coding 有清晰责任边界。一个万能 Agent 看起来简单，实际会把调度、执行、评审和状态混在一起。分层之后，每个环节都能被审查、替换和优化。

下一篇进入多工具适配：如果明天换 AI Coding 工具，团队的规则、任务状态和历史产物还能不能保留。

第10章　多工具适配：把共享语义放 Runtime，把工具差异放 Adapter

AI Coding 工具变化很快。今天团队可能主要用 Claude，明天有人想用 OpenCode，后天又要接入公司内部 IDE 插件。如果每接一个工具都重建一套 Agent、Prompt、任务状态和规则体系，团队会再次回到重复建设。

多工具适配的关键，是把共享语义放在 runtime，把工具差异放在 adapter。Runtime 承接任务状态、阶段产物、规则、上下文和验证证据；adapter 负责把某个具体工具的命令、hook、上下文注入和 UI 入口接到这套 runtime 上。

> 一句话结论：**工具可以换，任务状态、规则和交付语义不能跟着重建。**

常见误区

表 10-1　常见误区

误区	表现	后果
每个工具一套流程	Claude 一套，OpenCode 一套，IDE 插件一套	产物和状态不能共享
工具 Prompt 承接业务规则	换工具就要迁移规则	规则重复维护
adapter 修改任务状态语义	不同工具对完成、失败、暂停理解不同	恢复和归档混乱
只做 CLI，不考虑 hook/context	命令能跑，但上下文注入不一致	Agent 行为不稳定
追求所有工具能力完全一致	为少数工具特性污染公共 runtime	核心语义变复杂

适配不是把所有工具拉平，而是让不同工具共享同一套交付语义。

10.1　多工具适配真正要保住什么

工具可以换，但下面这些东西不能跟着换。

表 10-2　多工具适配真正要保住什么

必须保住	原因
任务状态	换工具后还要知道任务走到哪里
稳定产物	PRD、设计、计划、总结要能继续审查

必须保住	原因
规则体系	团队规则不能复制到多个工具 Prompt
上下文产物	working set 要能被不同工具读取
验证证据	质量判断不能随工具丢失
历史归档	相似任务经验要跨工具复用

如果一个工具接入后，必须复制一份规则、重建一套任务状态，说明适配层越界了。

设计原则

第一，**共享状态只在 runtime**。adapter 不定义新的任务状态真相源。

第二，**共享规则只在 SPEC**。adapter Prompt 只承接工具和运行时契约，不重复维护业务规则。

第三，**工具差异留在边缘**。命令格式、插件结构、hook 机制、UI 入口都属于 adapter，不应污染核心流程。

10.2　案例拆解：Runtime 和 Adapter 的关系

多工具适配可以简化成：

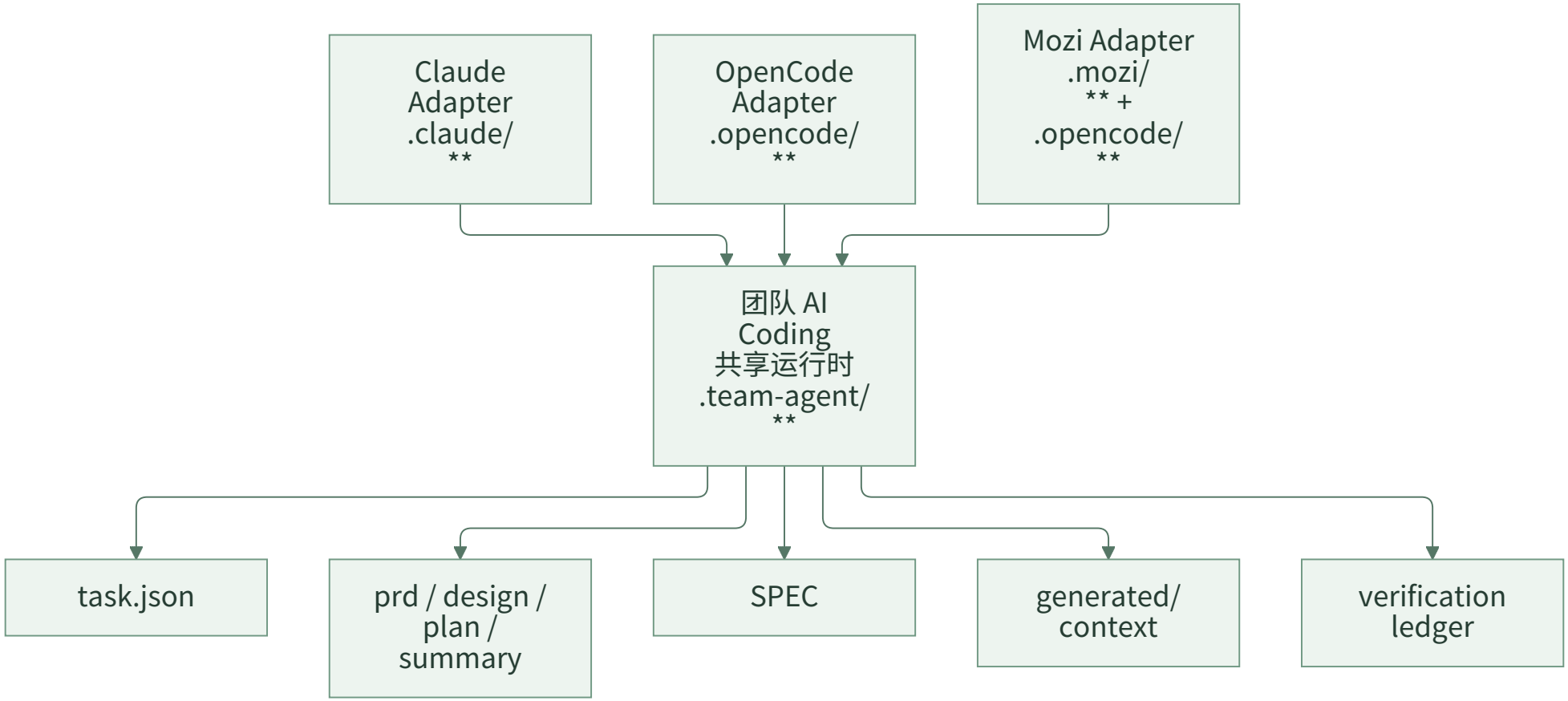


图 10-1　案例拆解：Runtime 和 Adapter 的关系

目录上大致是：

示例 10-1　案例拆解：Runtime 和 Adapter 的关系

```
src/
├── templates/runtime/
│   ├── tools/
│   ├── spec/
│   └── workflow/
└── adapters/
    ├── claude/
    ├── opencode/
    ├── mozi/
    └── codex/
```

`runtime/templates/**` 是共享 runtime 模板；`src/adapters/**` 是不同工具的生成模板和提示适配。业务项目初始化后，公共状态仍然落在 `.team-agent/**`。

Adapter 应该负责什么

Adapter 的职责可以归纳为四类：

表 10-3　Adapter 应该负责什么

职责	示例
命令暴露	把 `/ai-feature`、`/ai-quick` 等入口映射到工具支持的命令形态
Agent/Skill 模板	生成当前工具可识别的 agent、skill、prompt 配置
Hook/Context 注入	在会话或 subagent 启动时注入任务摘要、Spec、working set 路径
本地配置	生成工具需要的配置文件和本地 runtime 配置

Adapter 不应该负责：

- 定义新的任务状态。
- 改写 workflow 主干。
- 重复维护团队规范。
- 把工具专属能力变成核心 runtime 强依赖。

一个 adapter 生成结构可以类似：

示例 10-2　Adapter 应该负责什么

```
.claude/
├── agents/
├── commands/
├── skills/
└── settings.local.json

.opencode/
├── agents/
├── commands/
└── plugin/

.team-agent/
├── tasks/
├── spec/
├── tools/
└── workflow/
```

工具目录可以不同，但 `.team-agent/**` 仍是共享状态和产物位置。

10.3　Runtime 和 Adapter 的边界测试

接入新工具时，可以用三个问题做边界测试：

1. 删除这个工具的私有目录后，任务状态和历史产物是否还在？
2. 新工具是否只需要读取 `.team-agent/**` 就能恢复任务？
3. 团队规则是否没有复制到新工具的 Prompt 里？

如果答案是否，说明共享语义没有真正放到 runtime。

一个简化的 adapter manifest 可以这样表达：

清单 10-3　Runtime 和 Adapter 的边界测试

```
{
  "adapter": "opencode",
  "runtimeRoot": ".team-agent",
  "commands": [
    { "name": "ai-feature", "runtimeCommand": "/ai-feature <taskId>" },
    { "name": "ai-quick", "runtimeCommand": "/ai-quick <taskId>" }
  ],
  "generatedPaths": [
    ".opencode/agents/",
    ".opencode/commands/",
    ".opencode/plugin/"
  ],
  "sharedState": [
    ".team-agent/.current-task",
    ".team-agent/tasks/{taskId}/task.json",
    ".team-agent/spec/",
```

```
    ".team-agent/tasks/{taskId}/generated/context/"
  ]
}
```

这里的重点是 manifest 显式声明：adapter 暴露哪些入口、生成哪些工具私有文件、读取哪些共享 runtime 路径。它不定义新的任务状态。

Adapter 需要有生成清单

多工具接入最怕“生成了哪些文件没人知道”。建议每个 adapter 都维护 manifest 或生成清单。

表 10-4　Adapter 需要有生成清单

字段	作用
`adapter`	工具名称
`generatedPaths`	本 adapter 生成哪些私有文件
`sharedRuntimeRoot`	共享运行时根目录
`commands`	暴露哪些命令
`contextHooks`	什么时候注入任务摘要和规则
`doctorChecks`	初始化后检查哪些文件

这样后续升级 adapter 时，团队能判断是工具私有变更，还是影响了共享 runtime。

为什么 Prompt 不承接共享知识

多工具适配时，最容易犯的错误是把共享知识复制到每个工具 Prompt 里。这样一来，每次规则变化都要同步多个 adapter，迟早会不一致。

更稳的方式是：

表 10-5　为什么 Prompt 不承接共享知识

内容	放哪里
工具如何调用命令	adapter prompt
Agent 输出 JSON schema	adapter prompt
阶段交付流程	runtime workflow + adapter prompt
Java 实现规范	SPEC
应用业务规则	App SPEC
历史案例	archive

内容	放哪里
当前任务上下文	generated context

也就是说，adapter 只解释“这个工具如何接入流程”，不承接长期业务知识。

Tool lock-in 的早期信号

下面这些信号说明团队已经被某个工具绑住了：

表 10-6　Tool lock-in 的早期信号

信号	问题
任务状态只存在该工具会话里	无法跨工具恢复
规则写在该工具专属 Prompt 中	换工具要迁移知识
历史案例只能从聊天记录搜索	归档资产不可复用
验证证据没有落地文件	审查依赖工具界面
adapter 里有大量业务规则	工具层污染业务知识

越早把这些内容移到 runtime，后续越容易接入新工具。

10.4　团队参考做法

其他团队可以先定义三层：

示例 10-4　团队参考做法

```
runtime/
├── tasks/
├── spec/
├── context/
└── tools/

adapters/
├── cli/
├── ide-plugin/
└── web-runner/
```

第一阶段只需要支持一个工具，但也要提前把边界分清：

1. 任务状态写 runtime，不写工具私有目录。
2. 规则写 spec，不写工具 Prompt。
3. 工具 Prompt 只引用路径，不粘贴完整产物。
4. 换工具时，旧任务目录仍能被新工具读取。

第二阶段再增加更多 adapter。每增加一个 adapter，都检查它是否复用了 runtime，而不是复制 runtime。

10.5　检查清单

- 不同 AI 工具是否共享同一个任务目录?
- adapter 是否只负责工具接入?
- 规则是否只维护在 SPEC 中?
- Prompt 是否重复了大量业务规则?
- 工具切换后，旧任务是否能继续?
- hook/context 注入是否读取 runtime 产物?
- adapter 是否引入了核心 runtime 不需要的依赖?
- 新工具接入是否需要改 workflow 主干?

10.6　思考与实践

1. 如果你们团队明天从一个 AI Coding 工具切到另一个工具，历史任务和规则还能继续用吗?
2. 你们现在的 Prompt 里有多少内容其实属于共享规则，而不是工具适配?
3. 新接一个工具时，你们更担心命令适配、上下文注入，还是任务状态不一致?

10.7　结尾

多工具适配的目标不是让所有工具长得一样，而是让团队交付语义保持一致。共享状态、规则、上下文和证据放 runtime；命令、hook、插件和 UI 差异放 adapter。这样团队不会被单一工具绑定，也不会因为接入新工具重复建设 Agent。

下一篇进入协作面：当 AI 产出 design、plan、diff 和 evidence 后，团队应该如何审查、接管和治理这些产物。

第11章　协作面：团队如何审查和接管 Agent 产物

AI Coding 在个人场景里，聊天窗口就是主要工作界面。但团队协作不能只围绕聊天记录展开。需求负责人要看 PRD 和验收范围，架构师要看技术设计，开发负责人要看计划和 task，代码评审人要看 diff，测试或质量负责人要看验证证据。如果这些内容散落在聊天里，团队很难审查、评论、确认和接管。

协作面的目标，是把 AI 产物变成团队可协作的对象。人不需要进入同一个 AI 会话，也应该能看到一个需求的 PRD、设计、计划、diff、测试证据、评审意见和当前状态。

> 一句话结论：**团队审查的对象应该是产物、diff 和证据，而不是聊天记录。**

这类协作面不是替代本地 runner 或 AI 工具，而是围绕 feature 产物建立一个团队控制面。它让不同角色能在同一个需求空间里看到稳定产物、当前 diff、验证证据和待确认事项。

常见误区

表 11-1　常见误区

误区	表现	后果
团队只看聊天记录	设计、计划、证据都在对话里	审查困难，无法结构化评论
只看最终代码 diff	不看 PRD、设计和计划	不知道代码是否满足需求和方案
把协作面做成仪表盘	只有状态列表和统计	不能真正审查产物
浏览器里直接执行所有代码	控制面和执行面混在一起	权限、安全和环境复杂度上升
缺少接管机制	AI 卡住后只能重新开会话	人无法从稳定产物继续推进

团队协作要围绕 artifact、diff 和 evidence，而不是围绕聊天。

11.1　为什么聊天记录不适合团队审查

聊天记录有两个问题：一是线性，二是混杂。需求确认、方案讨论、工具输出、错误日志、模型解释都混在一起，后来的评审人很难快速找到该看的东西。

表 11-2　为什么聊天记录不适合团队审查

团队角色	真正想看	不想看
需求负责人	PRD、范围、非目标、验收	模型搜索过程
架构师	设计、影响面、风险、回滚	反复试错的聊天

团队角色	真正想看	不想看
开发负责人	task、路径、依赖、验证命令	模型自然语言计划
Code Reviewer	staged diff、task contract、ledger	Agent 自我总结
测试负责人	baseline、green、regression、失败分类	“应该无关”的解释

协作面要做的，就是把这些角色各自关心的对象从聊天里拆出来。

设计原则

第一，**一个需求一个协作空间**。团队成员围绕同一个 feature room 查看和讨论产物。

第二，**artifact-first，diff-first，evidence-first**。设计看 artifact，代码看 diff，质量看验证证据。

第三，**控制面和执行面分离**。浏览器负责协作和审查，代码执行仍由本地或 runner 承接。

11.2　案例拆解：Feature Room

一个 Feature Room 可以这样组织：

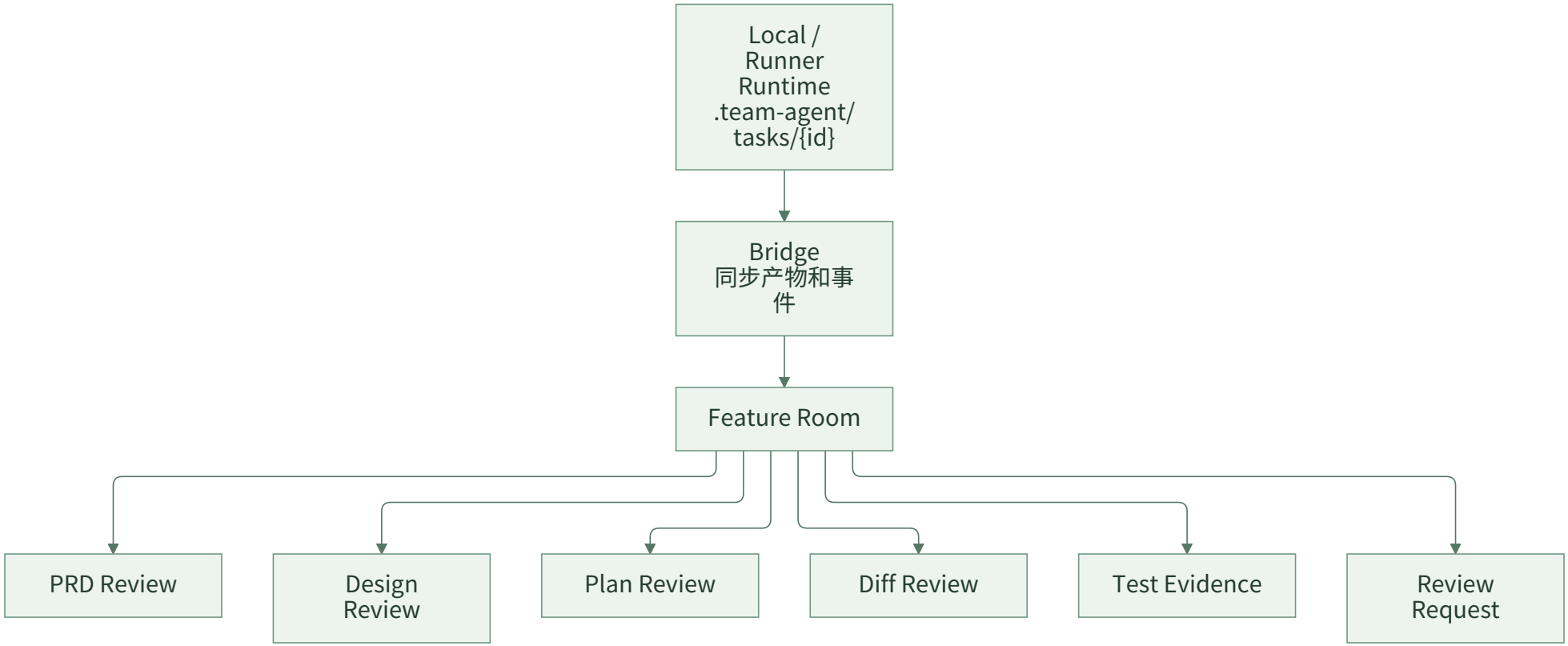


图 11-1　案例拆解：Feature Room

Feature Room 不是简单展示状态，而是把一个需求的关键产物组织到同一个工作面：

示例 11-1 案例拆解：Feature Room

```
Feature Room
├── Overview
│   ├── 当前阶段
│   ├── 需求摘要
│   └── 风险提示
├── Artifacts
│   ├── prd.md
│   ├── design.md
│   ├── plan.md
│   └── summary.md
├── Tasks
│   └── plan.tasks.json
├── Diff
│   └── 当前 staged / branch diff
├── Evidence
│   └── verification-ledger.jsonl
└── Review Requests
    ├── 设计确认
    ├── 计划确认
    └── 代码审查
```

这类界面解决的是“人如何参与 Agent 交付”的问题。

Feature Room 最小信息架构

一个可用的 Feature Room 不需要一开始很花哨，但至少要有这些区块：

表 11-3 Feature Room 最小信息架构

区块	最小内容
Overview	当前阶段、负责人、风险、下一步
Requirement	`prd.md`、切片、验收和非目标
Design	`design.md`、待确认事实、影响范围
Plan	`plan.md`、`plan.tasks.json`、task 状态
Diff	当前 task 的 staged diff 或 branch diff
Evidence	verification ledger、baseline 和失败分类
Decisions	review request、确认记录、阻塞问题
Archive	summary、规则处置、历史索引

这个信息架构比“做一个 AI 看板”更重要。看板只能告诉你任务在哪里，Feature Room 要能让人真正审查和接管。

11.3 团队审查应该看什么

Artifact-first 的含义是：团队审查稳定产物，而不是让每个人翻聊天记录。

表 11-4 团队审查应该看什么

阶段	审查对象	关注点
PRD	`prd.md`	需求是否正确，范围是否清楚
Design	`design.md`	方案是否合理，风险是否识别
Plan	`plan.md`、`plan.tasks.json`	task 是否可执行，验证是否充分
Summary	`summary.md`	交付范围、验证、风险、沉淀是否完整

评论应该挂在 artifact 上，而不是散在会话里。这样后续恢复、复盘和归档时，团队能看到完整决策链路。

Artifact review 的评论也应该结构化：

清单 11-2 团队审查应该看什么

```
{
  "artifact": "design.md",
  "section": "影响范围",
  "severity": "major",
  "comment": "设计没有说明存量权益类型查询展示是否保持不变",
  "requiredAction": "补充存量类型兼容策略和回归范围"
}
```

这样的评论可以回流到设计修订，而不是停留在聊天提醒。

Diff-first Review

代码审查应该围绕 diff，而不是围绕 Agent 的自然语言描述。

清单 11-3 Diff-first Review

```
{
  "file": "src/main/java/com/example/prize/PrizeTypeService.java",
  "taskId": "T1",
  "featureId": "feature-01",
  "reviewStatus": "changes_requested",
  "comments": [
    {
```

```
      "line": 42,
      "severity": "major",
      "message": "新增类型只覆盖创建路径，查询展示路径未覆盖"
    }
  ]
}
```

Diff review 要能关联：

- 当前 task。
- 对应 feature slice。
- 对应 acceptance criteria。
- 对应验证证据。

这样评审不只是“代码写得好不好”，而是“这段改动是否满足当前 task”。

Diff-first 还有一个好处：它天然限制审查范围。评审人只看当前 task 的 staged diff，而不是被一个大分支 diff 淹没。

Evidence-first Quality

验证证据应该在协作面可见。负责人不应该只看到“测试通过”四个字，而要看到命令、结果、失败分类和阻塞判断。

示例 11-4 Evidence-first Quality

```
Evidence Panel
├── Baseline
│   └── mvn test -Dtest=PrizeTypeServiceTest: passed
├── Green
│   └── shouldCreateNewPrizeType: passed
├── Regression
│   └── shouldKeepLegacyType: warning / preexisting_baseline_failure
└── Blocking
    └── none
```

这能让团队讨论质量时基于证据，而不是基于模型解释。

Evidence 面板里最值得突出的是阻塞判断：

表 11-5 Evidence-first Quality

状态	展示方式
passed	绿色通过
warning / preexisting failure	黄色提示，展示 baseline 对比
blocking failure	红色阻塞，展示失败签名和 changed files

状态	展示方式
missing evidence	红色阻塞，提示缺少验证

团队要让“没有证据”也可见，而不是只展示成功结果。

Review Request

协作面还需要明确确认机制。不是每个动作都需要审批，但关键节点要有 review request：

表 11-6 Review Request

Request	触发时机	审查对象
PRD Confirm	需求结构化后	`prd.md`
Design Review	技术设计完成后	`design.md`
Plan Review	开发计划完成后	`plan.md`、`plan.tasks.json`
Code Review	单个 task diff 完成后	staged diff、ledger
Spec Disposition	交付总结后	SPEC 更新建议或无需更新说明

review request 的结果应该回到运行时，作为流程继续、修改或阻塞的依据。

Review request 可以很轻，但必须有三件事：审查对象、决策结果、后续动作。

清单 11-5 Review Request

```
{
  "type": "plan_review",
  "artifactPaths": ["plan.md", "plan.tasks.json"],
  "decision": "changes_requested",
  "requiredActions": [
    "为 T2 补充回归验证命令",
    "把纯测试 task 合并到对应实现 task"
  ]
}
```

11.4 控制面和执行面分离

浏览器协作面不一定要直接执行代码。更稳的结构是：

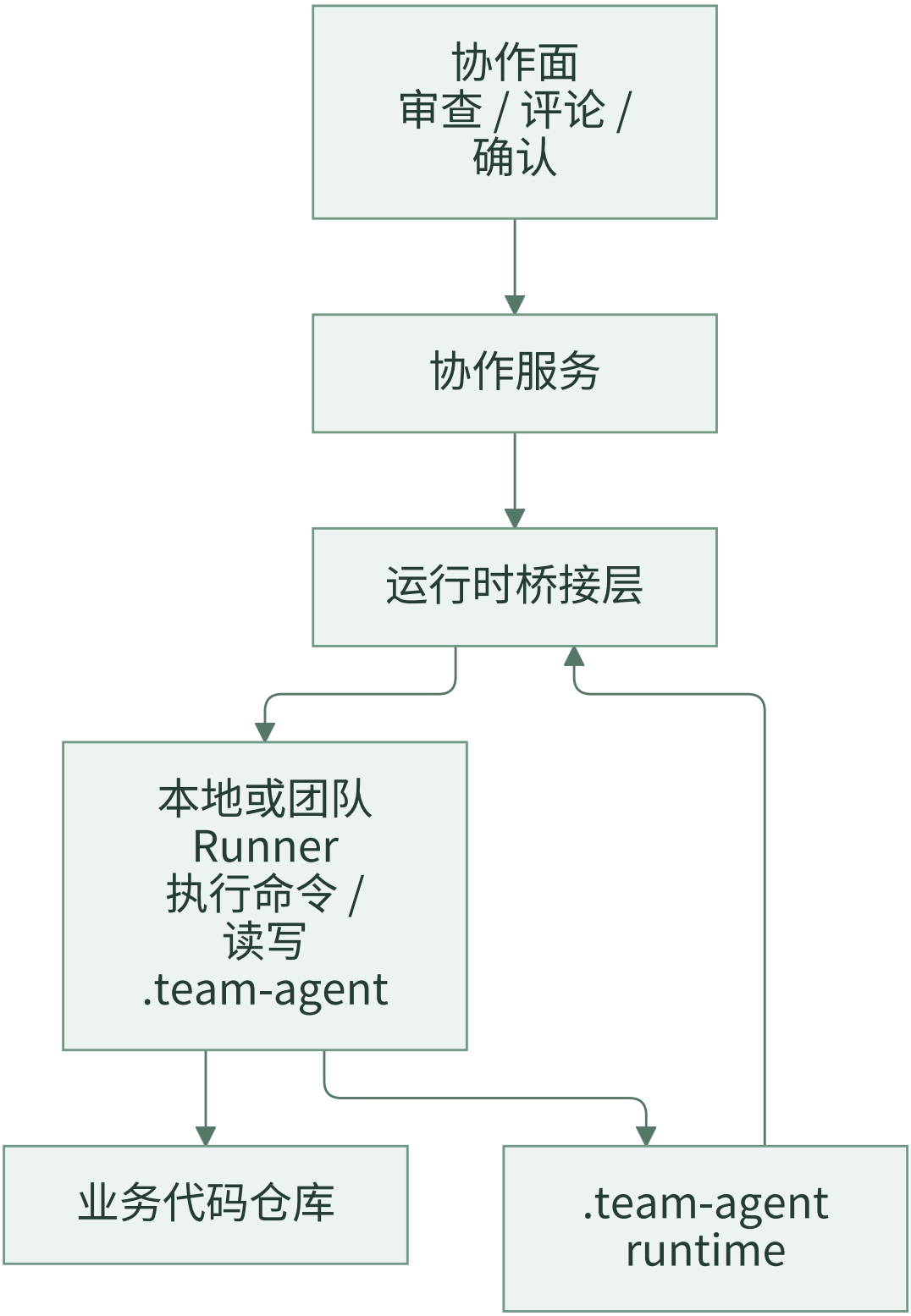


图 11-2 控制面和执行面分离

这样做有几个好处：

- 浏览器不直接持有业务仓库执行权限。
- 本地 runner 可以复用开发者环境。
- 团队 runner 可以承接更标准化的执行。

- 协作面专注审查和决策，不变成云 IDE。

控制面如果直接执行一切，很容易变成权限和环境的大泥潭。更稳的方式是让它只发起请求、展示产物、记录决策，把真正的代码执行留给本地或团队 runner。

11.5 团队参考做法

其他团队可以先做一个轻量协作面，不必一开始做完整产品。

第一阶段可以只是一个静态或半自动页面：

示例 11-6 团队参考做法

```
Feature Page
├── 任务状态
├── PRD
├── Design
├── Plan
├── Tasks
├── Diff
├── Verification Evidence
└── Review Comments
```

第二阶段再加入：

- review request。
- 评论和审批。
- runner 事件同步。
- 多人实时协作。
- 指标统计。

最小原则是：让人能围绕产物审查，而不是围绕聊天审查。

11.6 检查清单

- 一个需求是否有独立协作空间?
- 团队是否能直接查看 PRD、设计、计划和总结?
- 代码评审是否基于 diff?
- 验证证据是否可见?
- 评论是否能挂到 artifact 或 diff 上?
- 关键节点是否有 review request?
- 人是否能接管 Agent 卡住的任务?
- 控制面是否和代码执行面分离?

11.7 思考与实践

1. 你们负责人现在审查 AI 产物时，主要看聊天记录、代码 diff，还是 design / plan / evidence?
2. 如果一个 AI 任务卡住，人接管时最缺的信息是什么?
3. 你们团队更需要 artifact review、diff review、test evidence，还是 review request?

11.8 结尾

团队 AI Coding 的协作对象应该是产物，而不是聊天。Feature Room、artifact-first review、diff-first review、test evidence 和 review request 共同解决的是：人如何审查、确认、接管和治理 Agent 交付。

下一篇进入团队最小落地包：如果只给 30 天，应该从哪套目录、哪条流程、哪几类规则开始。

PART 05

第5篇　最小试点与收官

最后把前面的边界收束成一个可试用的流程。选择一个范围清晰、可验证的需求，观察哪些约定真正帮助了协作，再逐步扩展。

阅读重点

能组织一次有明确范围、验证依据和复盘记录的团队试点。

第12章　团队最小落地包：从一套目录、一条流程、三类规则开始

前面的文章分别讲了共享运行时、SPEC、复杂需求拆分、有界上下文、阶段化交付、Task 化开发、验证治理、资产沉淀、Agent 分层和多工具适配。真正落地时，团队不需要一次性实现全部能力。更现实的方式，是先做一个最小闭环：任务能进来，需求能确认，规则能被引用，计划能拆 task，验证有证据，交付后能总结和归档。

最小落地包的目标，是让团队用较低成本建立第一套 AI Coding 工程边界。它不追求平台完整，也不追求自动化程度最高；它只要求每次 AI Coding 任务都有状态、有产物、有规则、有验证、有沉淀。

> 一句话结论：**不要从大平台开始，先让一个真实需求完整跑完最小闭环。**

12.1　先选对试点需求

最小落地包成败，首先取决于试点需求选得对不对。不要一上来选“全链路重构”“跨十几个应用联调”“历史包袱极重”的需求，也不要选只改一个文案的需求。前者会把流程问题和业务复杂度混在一起，后者又看不出团队化价值。

更适合试点的需求通常有这些特征：

表 12-1　先选对试点需求

维度	推荐选择	不推荐选择
业务范围	一个明确功能点或一个小流程	多条业务线同时改造
代码范围	2-5 个核心模块，可列出主要文件	影响范围无法提前判断
验证方式	有单测、集成测试或明确回归命令	只能靠人工点页面
规则沉淀	有可复用业务规则或工程规范	一次性临时处理
风险级别	中低风险，可回滚	强资金、强合规、强线上实时风险

试点不是为了证明 AI “什么都能做”，而是为了证明团队能把 AI Coding 过程纳入工程闭环。

12.2　最小闭环

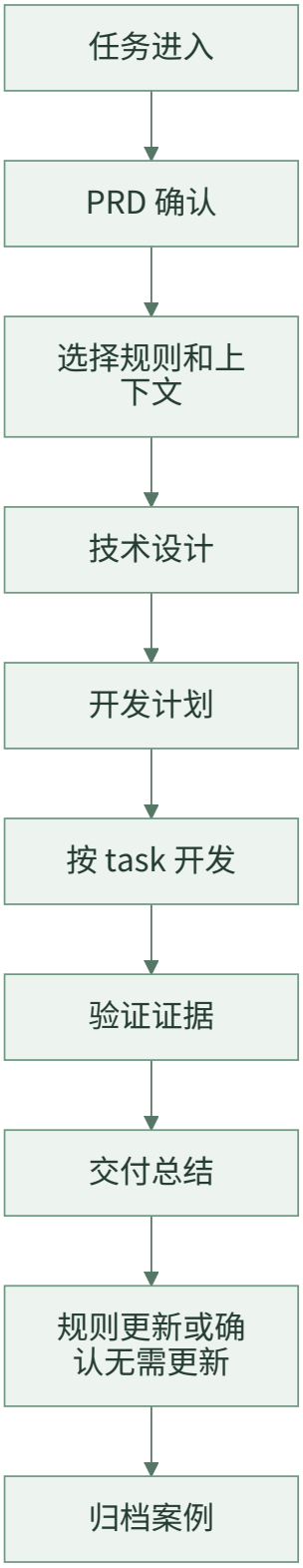


图 12-1　最小闭环

这条闭环里，自动化可以逐步增强，但产物边界不要省。

最小目录模板

可以从下面这个目录开始：

示例 12-1　最小目录模板

```
.team-agent/
├── current-task
├── workflow/
│   └── phases.json
├── tools/
├── spec/
│   ├── team/
│   │   ├── index.md
│   │   ├── architecture.md
│   │   ├── implementation.md
│   │   ├── verification.md
│   │   └── delivery.md
│   └── app/
│       ├── profile.md
│       ├── index.md
│       ├── domain/
│       ├── business/
│       └── engineering/
├── tasks/
│   └── {taskId}/
│       ├── task.json
│       ├── prd.md
│       ├── design.md
│       ├── plan.md
│       ├── plan.tasks.json
│       ├── summary.md
│       ├── context/
│       │   ├── prd.jsonl
│       │   └── code.jsonl
│       └── evidence/
│           └── verification-ledger.jsonl
└── archive/
    └── index.jsonl
```

这里最重要的是 owner 清晰：

- `task.json` 是状态。
- Markdown 是稳定产物。
- `context/` 是阶段上下文。
- `evidence/` 是验证证据。
- `spec/` 是长期规则。
- `archive/` 是历史案例。

最小数据结构

`task.json`：

清单 12-2　最小数据结构

```
{
  "taskId": "T-001",
  "status": "in_progress",
  "currentPhase": "plan_delivery",
  "branch": "feature/prize-type",
  "artifacts": {
    "prd": "prd.md",
    "design": "design.md",
    "plan": "plan.md",
    "summary": "summary.md"
  }
}
```

`plan.tasks.json`：

清单 12-3　最小数据结构

```
{
  "tasks": [
    {
      "id": "T1",
      "title": "实现新增权益类型识别",
      "featureId": "feature-01",
      "allowedPaths": ["src/main/java/...", "src/test/java/..."],
      "verificationCommands": ["mvn test -Dtest=PrizeTypeServiceTest"],
      "acceptanceCriteria": ["新权益类型可识别", "存量类型行为不变"]
    }
  ]
}
```

`verification-ledger.jsonl`：

清单 12-4　最小数据结构

```
{"taskId":"T1","stage":"baseline","command":"mvn test
-Dtest=PrizeTypeServiceTest","status":"passed"}
{"taskId":"T1","stage":"green","command":"mvn test
-Dtest=PrizeTypeServiceTest","status":"passed","classification":"passed","blocksCurrentTask":false
}
```

`archive/index.jsonl`：

清单 12-5　最小数据结构

```
{"taskId":"T-001","title":"新增权益类型","domain":["奖品"],"changedAreas":
["PrizeTypeService"],"summaryPath":".team-agent/archive/T-001/summary.md"}
```

这些结构不需要一开始很复杂，但要稳定。后续自动化、UI、统计都可以基于它们演进。

最小规则体系

先建三类规则：

表 12-2　最小规则体系

目录	先放什么
`spec/team/`	团队通用架构、实现、验证、交付规则
`spec/app/domain/`	应用领域模型、状态、不变量
`spec/app/business/`	可复用业务需求开发规则
`spec/app/engineering/`	应用工程规范、测试命令、中间件规则

每条 focused Spec 至少包含：

清单 12-6　最小规则体系

```
# 规则标题

## Overview

## How To Apply

## Rules

## DO / DON'T

## Checklist

## Source Reference
```

第一阶段可以先人工选择相关规则；第二阶段再做 selector。

最小流程规范

每个任务必须经过：

1. `prd.md`：确认后的需求。
2. `design.md`：技术方案。
3. `plan.md` + `plan.tasks.json`：可执行计划。
4. task 开发：每次只做一个 task。
5. `verification-ledger.jsonl`：验证证据。
6. `summary.md`：交付总结。
7. Spec disposition：更新规则或确认无需更新。

8. archive：归档历史案例。

如果团队暂时没有自动控制平面，也可以用人工 checklist 驱动。关键是产物和证据不能省。

12.3　每天怎么跑

试点阶段不要只设计流程，还要规定每天怎么用。一个简单节奏就够：

表 12-3　每天怎么跑

时间点	动作	产物
开始前 10 分钟	确认当前 task、阶段、阻塞项	`task.json`、当前阶段
设计前	检查 PRD 是否有范围、非目标、验收	`prd.md`
开发前	审计划和 task contract	`plan.md`、`plan.tasks.json`
每个 task 完成后	看 diff 和验证证据	staged diff、ledger
每天结束前	更新阻塞、风险和下一步	`task.json`、简短记录
交付后	补 summary、规则处置和归档	`summary.md`、archive index

这个节奏的价值不是增加仪式感，而是让团队每天都能回答：现在做到哪、为什么能继续、哪里不能继续。

12.4　30 天推进建议

表 12-4　30 天推进建议

时间	目标	产物
第 1 周	建目录和模板	`.team-agent/`、PRD/Design/Plan/Summary 模板
第 2 周	选 1-2 个真实需求试跑	task 目录、plan.tasks.json、ledger
第 3 周	沉淀第一批 SPEC	team spec、app profile、business spec
第 4 周	复盘和归档	summary、archive index、流程调整

不要一开始追求全自动。先让一个真实需求完整跑完，再根据痛点补自动化。

最小指标

试点要有指标，但不要一开始就做复杂报表。先观察五类信号：

表 12-5　最小指标

指标	看什么	判断方式
接手成本	换人后能否在 10 分钟内知道当前状态	看 task、artifact 和 ledger 是否完整
计划可执行性	worker 是否频繁退回计划阶段	统计 plan contract 缺失次数
Diff 可审查性	reviewer 是否能按 task 审 diff	看每个 task 是否有明确 allowed paths 和验收
验证可信度	测试失败是否能分类	看 baseline、green、regression 记录
资产复用	下个相似需求是否引用了历史案例或规则	看 summary、spec disposition、archive recall

这些指标不追求漂亮，重点是暴露团队 AI Coding 流程里最先需要补的短板。

失败预案

试点失败通常不是因为 AI 不够强，而是边界没立住。可以提前约定几条处理规则：

表 12-6　失败预案

失败信号	处理动作
PRD 反复变化	停止开发，回到 PRD 确认，不让 worker 继续改代码
计划 task 太大	退回计划阶段，按文件范围、验收标准和验证命令重新拆
验证命令缺失	不允许用“手动看起来正常”替代 ledger
Diff 超出 allowed paths	阻塞 review，要求解释或拆分
规则过期导致误导	在 summary 里记录 spec disposition，交付后更新或废弃

这几条看起来简单，但能避免试点变成“AI 写了一堆代码，最后靠人兜底”。

12.5　检查清单

- 是否有统一任务目录?
- 是否有状态真相源?
- 是否有 PRD、设计、计划、总结四类稳定产物?
- 是否有规则目录，而不是只靠 Prompt?
- 是否有 task 级开发计划?
- 是否有验证 ledger?
- 是否有交付后的规则处置?

- 是否有 archive index?
- 是否能用一个真实需求完整跑完闭环?

12.6　思考与实践

1. 如果只给 30 天，你们团队最应该先补任务状态、SPEC、上下文，还是验证 ledger?
2. 选一个即将开始的真实需求，你能否用这篇文章的最小目录把它完整跑一遍?
3. 如果你准备在团队试点，最需要争取谁的支持：研发负责人、架构师、测试负责人，还是效能团队?

12.7　结尾

团队 AI Coding 工程化不需要从大平台开始。先建立最小闭环：目录、流程、规则、task、验证、总结、归档。只要这个闭环稳定，后续再补上下文选择、多工具适配、协作工作台和指标化评估，系统就会自然长出来。

结语　AI Coding 团队化，最后还是工程问题



图 结-1　AI Coding 团队化工程闭环

这个系列写到这里，已经从共享运行时、SPEC、需求拆分、有界上下文、阶段化交付、Task 化开发、验证治理、资产沉淀、Agent 分层、多工具适配、协作面，一直写到了团队最小落地包。

如果只看目录，容易觉得这套东西有点“重”。AI Coding 不就是让 AI 帮我们写代码吗，为什么要讲这么多状态、规则、产物、上下文、证据和归档?

原因也很简单：个人用 AI Coding，靠的是个人判断；团队用 AI Coding，靠的是工程边界。一个人知道该给哪些上下文，知道模型哪里容易乱写，知道哪些测试失败是历史问题，也知道代码改到什么程度该停。但这些判断如果一直留在个人脑子里，团队就没有办法复制、审查和接手。

这个系列想讨论的，其实不是“怎么把 AI 用得更猛”，而是怎么让 AI Coding 进入团队已有的工程秩序。

> 一句话结论：AI Coding 团队化的终点，不是做出一个更大的 Agent，而是让需求、上下文、规则、验证和沉淀都变成团队可接手的工程资产。

结.1　这个系列真正想留下什么

如果要把前面所有文章压缩成一句话，我会这样说：

> 团队 AI Coding 不是把研发流程交给 AI，而是把 AI 放进一个可审查、可恢复、可验证的研发流程里。

这句话里有几个关键词。

表 结-1 这个系列真正想留下什么

关键词	解决什么问题	对应文章
可审查	不再只看聊天记录，而是看 PRD、设计、计划、diff 和证据	阶段化交付、Task 化开发、协作面
可恢复	换人、换会话、换工具后，任务还能继续	共享运行时、状态真相源、多工具适配
可验证	AI 不能只说“应该没问题”，必须留下命令、结果和失败分类	验证治理
可复用	每次交付后，经验要进入规则、案例和研究资产	SPEC、资产沉淀
可控制	上下文、任务范围、Agent 职责都有边界	有界上下文、需求拆分、Agent 分层

这些能力听起来不像“AI 技巧”，更像研发效能、架构治理和工程协作。但这正是团队化落地绕不开的部分。

工具会继续变，模型也会变强。今天是 IDE 插件，明天是 CLI，后天可能是团队级 Agent 平台。但只要团队想稳定地把 AI 放进研发流程，就绕不开这些问题：

- 当前需求到底确认了吗？
- AI 为什么读这些文件，而不是读全仓？
- 哪些规则是长期有效的，哪些只是本次任务事实？
- 这次代码改动对应哪个 task 和验收标准？
- 测试失败是历史失败、本次回归，还是缺少环境？
- 交付后有什么经验会进入下一次任务？
- AI 卡住时，人能不能接手？

能把这些问题回答清楚，团队才算真正开始建设 AI Coding 能力。

结.2 不要把这个系列读成“造平台指南”

这个系列反复讲运行时、目录、数据结构、协作面，很容易被误读成：团队必须先造一个平台，才能用好 AI Coding。

我不这么看。

大多数团队第一步不需要平台，甚至不需要自动化。第一步应该是把一个真实需求跑清楚：需求怎么确认，规则怎么引用，计划怎么拆，验证怎么留证据，交付后怎么总结。

真正危险的不是“工具太简陋”，而是下面几种情况：

表 结-2 不要把这个系列读成“造平台指南”

现象	看起来像	实际问题
Agent 能一口气改很多文件	效率高	缺少 task 边界，review 很难接住
Prompt 里写满规则	很全面	规则没有来源、生命周期和审查机制
每个应用都有自己的 Agent	很贴业务	通用流程重复建设，行为不一致
AI 解释测试失败无关	很懂项目	没有 baseline 和 ledger，无法审计
每次都能靠高手救回来	团队有经验	经验没有资产化，换人后继续波动

所以这个系列不是劝大家一上来做大系统。相反，它一直在强调最小闭环：先让状态、产物、规则、验证、总结这几件事稳定下来。

平台可以晚一点，自动化也可以晚一点。边界不能太晚。

结.3 AI Coding 不是替代工程习惯，而是放大工程习惯

AI Coding 很容易暴露一个团队原本就存在的问题。

如果需求本来就说不清，AI 会把不清楚的部分编成看似完整的方案。

如果代码边界本来就混乱，AI 会沿着混乱边界继续扩散改动。

如果测试本来就长期失败，AI 会给出一堆“可能无关”的解释。

如果规则本来就只存在老员工脑子里，AI 每次都要重新猜。

这不是 AI 特有的问题。只是 AI 写得快，放大得也快。

所以团队推进 AI Coding 时，最好不要只问“模型能不能做”。更应该问：

表 结-3 AI Coding 不是替代工程习惯，而是放大工程习惯

问题	如果答案不清楚，先补什么
需求边界清楚吗？	PRD 结构化和功能切片
规则在哪里？	Team SPEC 和 App SPEC
上下文为什么是这些？	Working set 和选择证据

问题	如果答案不清楚，先补什么
代码改动能拆开审吗？	task contract 和 allowed paths
验证结果可信么？	baseline、green、regression ledger
做完后能复用吗？	summary、spec disposition、archive index

AI 不是用来绕过这些工程习惯的。它更适合承担搜索、整理、草拟、局部实现、补测试、生成总结这些执行工作。边界、取舍、验收和风险判断，仍然要由团队负责。

一个比较稳的分工是：

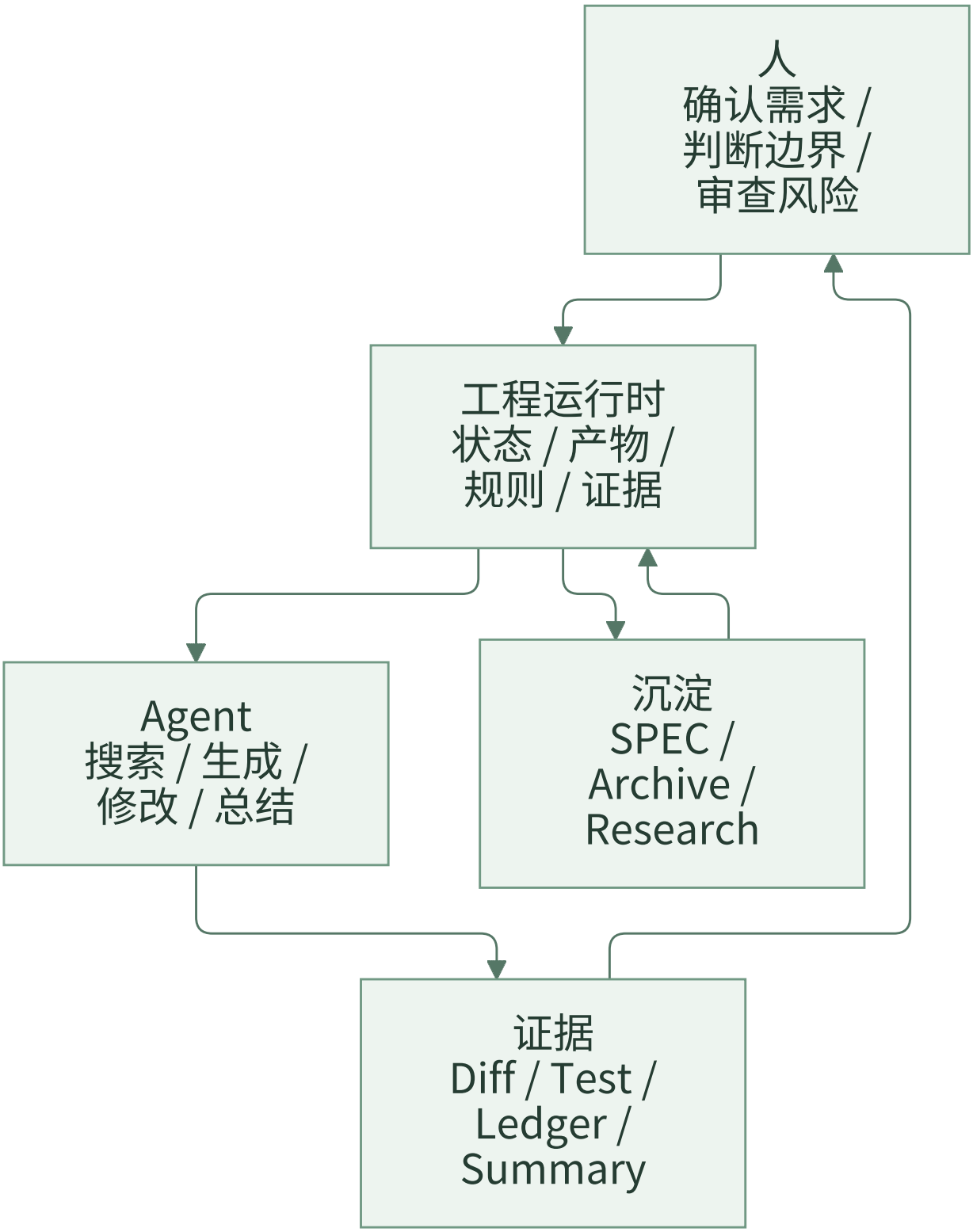


图 结-2　AI Coding 不是替代工程习惯，而是放大工程习惯

这张图里，AI 很重要，但它不是唯一中心。真正的中心是工程运行时和团队判断。

结.4　一个团队可以怎么继续往下走

如果读完这个系列，准备在团队里试一下，我建议不要从“建设完整 AI Coding 平台”开始。可以从一个 30 天小试点开始。

第一步：选一个真实需求

不要选太大的重构，也不要选只改文案的小需求。最好是一个中低风险、边界可控、能写出验收标准、能跑验证命令的真实功能点。

目标不是证明 AI 能写多少代码，而是证明团队能把一次 AI Coding 任务完整接住。

第二步：建最小目录

不用一开始把所有工具都做完。先有这些文件就够：

示例 结-1 第二步：建最小目录

```
.team-agent/
├── current-task
├── spec/
│   ├── team/
│   └── app/
├── tasks/
│   └── {taskId}/
│       ├── task.json
│       ├── prd.md
│       ├── design.md
│       ├── plan.md
│       ├── plan.tasks.json
│       ├── summary.md
│       └── evidence/
│           └── verification-ledger.jsonl
└── archive/
    └── index.jsonl
```

这套目录不是重点，职责才是重点：状态归状态，产物归产物，规则归规则，证据归证据，历史案例归历史案例。

第三步：先把人工流程跑通

第一轮可以完全人工驱动：

1. 需求负责人确认 `prd.md`。
2. 架构或负责人审 `design.md`。
3. 开发负责人审 `plan.md` 和 `plan.tasks.json`。
4. 每个 task 只允许改约定范围内的文件。
5. 每次验证写入 `verification-ledger.jsonl`。
6. 交付后补 `summary.md`。
7. 判断是否更新 SPEC，最后归档案例。

跑完一轮后，再决定哪些地方值得自动化。通常最先值得做的是模板生成、验证记录、上下文选择和归档索引。不要一开始就把所有流程都做成复杂系统。

第四步：看五个指标

试点结束后，不要只看“节省了多少人天”。更应该看这五个信号：

表 结-4 第四步：看五个指标

指标	问法
接手成本	换一个人，10 分钟内能知道任务状态和下一步吗？
计划质量	task 是否小到可执行、可审查、可验证？
Diff 收敛	代码改动是否围绕当前 task，没有顺手扩散？
验证可信度	测试失败是否能分类，是否有 baseline 对比？
资产复用	下一个相似需求能不能引用这次的规则或案例？

这些指标比“AI 写了多少代码”更接近团队能力。

结.5 最后应该坚持的几条底线

AI Coding 会继续变快，工具也会继续把流程包装得更顺。但团队要留几条底线。

第一，**需求没确认，不要急着改代码。** 模型越强，越容易把模糊需求补成一个看起来完整的方案。

第二，**规则不要只放 Prompt。** Prompt 可以承接流程，长期规则要有来源、适用边界和更新机制。

第三，**上下文不是越多越好。** 给 Agent 的上下文应该能解释为什么被选中，而不是把整个仓库都塞进去。

第四，**没有验证证据，就不要轻易说完成。** 测试命令、结果、失败分类和阻塞判断要留下来。

第五，**交付不是合代码就结束。** 总结、规则处置和归档，决定了下一次任务能不能少走弯路。

第六，**不要让一个万能 Agent 包办所有事。** 控制、设计、执行、评审、总结最好有清楚边界。

第七，**人不要从流程里消失。** AI 可以承担大量执行工作，但边界、优先级、风险和验收仍然需要人负责。

这几条不新鲜，甚至有点像工程常识。但 AI Coding 团队化最后拼的，恰恰就是这些常识能不能落到日常流程里。

结.6 思考与实践

1. 你们团队现在 AI Coding 最不稳定的环节，是需求、上下文、代码 diff、验证，还是交付沉淀？
2. 如果只选一个真实需求试点，你会先补 `task.json`、SPEC、`plan.tasks.json`，还是 verification ledger？

3. 你们团队有没有一个地方，可以让新人不看聊天记录，也能接手 AI 正在做的任务？

结.7　结尾

这个系列到这里就收住。

如果你只记住一件事，我希望是：AI Coding 的团队化，不是把每个人都训练成 Prompt 高手，也不是让一个 Agent 接管研发流程。更可靠的路径，是把个人经验里可复用、可审查、可验证的部分，变成团队共同拥有的工程资产。

从一个真实需求开始，把状态写清楚，把需求确认好，把规则放到该放的位置，把上下文选准，把计划拆小，把验证证据留下，把交付经验沉淀下来。

这件事不酷，但有用。

等这些基础动作稳定之后，团队再谈更强的 Agent、更顺的协作面、更自动化的运行时，才有意义。

术语索引

共享运行时

承接任务状态、阶段产物、上下文组织与验证的公共交付能力。 参见第1章。

SPEC

团队或应用需要持续维护、可被任务引用的稳定规则。 参见第2章。

功能切片

从 PRD 中提取的可验收需求单元，用于连接设计、实现与验证。 参见第3章。

有界上下文

围绕当前阶段和任务选择的必要材料及其范围。 参见第4章。

阶段产物

用于确认当前阶段结论、支持下一阶段和接管的持久文档或记录。 参见第5章。

Task

范围明确、可独立检查的工程任务。 参见第6章。

验证基线

改动前已知的检查结果，用于判断新出现的失败。 参见第7章。

规则沉淀

把有来源、有适用范围的经验转为可维护规则。 参见第8章。

Adapter

把工具入口、权限或能力差异映射到共同工程语义的适配层。 参见第10章。

接管

另一位参与者根据任务状态、产物和验证记录继续工作。 参见第11章。

章节与配套资源

正文中的规则、目录和流程示例可以直接阅读。准备在团队中试用时，可从第 12 章的最小落地包开始，结合具体需求选择需要的约定。

章节	配套入口
全书	团队最小落地包

勘误与交流

反馈时注明书名、电子版版本、章节和具体位置。代码问题请附运行环境与复现步骤。

https://github.com/DavidDingXu/ai-coding-for-teams/issues

微信公众号：好好用 AI

