

Contents

ODD 2.0 产出物驱动开发：统一中文版	7
1. 我们为什么需要 ODD?（痛点与解法）	7
2. 核心大循环 (The Core Loop)	8
3. 验证的三态模型 (Three-State Model)	8
4. 关键机制：防弹契约与 CAP 博弈	8
4.1 契约退化：ODD 最大的战略风险	8
5. 架构全景与分级指南	9
A01. 了解_核心痛点与解法	10
一句话与一个循环	10
你可能遇到过这些场景	10
一个循环	10
30 秒动手试试	10
下一步	11
五个核心概念	11
验证结果有三种	11
A03. 了解_评级与选型指南	12
我该从哪级开始	12
五个级别对比	12
5 题自测：找到你的起点	12
一句话建议	13
下一步	13
B05. 入门_个人闭环体验	13
你的第一个 ODD 闭环	13
场景	13
第一步：写契约（2 分钟）	14
第二步：让 AI 写代码（1 分钟）	15
第三步：验证（1 分钟）	15
第四步：封存（30 秒）	15
你刚刚做了什么	15
AI 是怎么“思考”的（七序简化版）	16
下一步	16
B07. 入门_团队实战与L0起步	16
实战进阶：15 分钟团队协作	16
场景设定	16
练习场景：Markdown 表格转 JSON	17
角色分配	17
第一轮（8 分钟）	17
第二轮（7 分钟）：交换角色	19
交叉审核（练习结束后）	19

练习后讨论（5 分钟）	19
常见错误	20
下一步	20
L0 兼容层：零流程切入	20
L0 是什么	20
L0 核心原则	20
L0 怎么做	21
L0 适用场景	21
L0 什么时候升级	21
下一步	21
C09. 工程_契约_产出物_证据	22
契约：怎么写清楚“什么算做好“	22
一个痛苦的场景	22
契约的最小结构	22
怎么写好验收条件	23
四种验收条件的写法	24
常见错误	25
下一步	25
对抗：让契约变得防弹	26
一个痛苦的场景	26
两道关卡	26
完整对话示例：用户搜索	27
什么时候需要对抗	29
常见错误	30
多少轮对抗合适？	30
进阶对抗：五项深度优化	30
下一步	32
产出物：一个任务一个交付物	33
一个痛苦的场景	33
产出物的五种类型	33
一个任务一个主产出物	33
粒度判断：太大、太小、刚好	34
常见错误	36
下一步	36
证据与封存：给未来的自己留证据	36
一个痛苦的场景	36
核心概念	36
四种证据类型	37
封存操作：从验证通过到锁定	38
一个完整的封存 YAML 示例	39
常见错误	40
下一步	41

模板库	41
L1 契约模板（最简）	41
L2 契约模板（含 hardness、executability、maturity）	41
L1 任务模板	42
L2 任务模板（含 task_level、state）	42
封存记录模板（含 decision_narrative）	42
证据记录模板	43
task_level 快速判断	43
下一步	43
C11. 工程_对象模型与标准规范	43
layer: L3 type: reference prerequisites: C09.工程_契约_产出物_证据 source:	
ODD.02对象-对象模型 last_updated: 2026-02-16	44
对象模型完整参考	44
概述	44
L1 基础	44
最小启动子集（L2）	46
L3 核心	48
L3 扩展	49
完整 YAML Schema 示例（L3）	50
理论来源	53
概述	53
L1 基础	53
最小启动子集（L2）	54
L3 核心	57
L3 扩展	59
完整 YAML 示例（L3）	60
理论来源	62
契约保鲜机制	63
概述	63
L1 • 基础	64
L2 • 标准	64
L3 • 严格	66
完整 YAML 示例	67
理论来源	69
与契约版本迁移的联动	69
C13. 工程_验证_门禁_状态机	70
验证：不 Review 代码，验证输出	70
一个让你不舒服的事实	70
为什么 Review 在 AI 时代失效	70
验证和 Review 的区别	70
怎么做输出验证	71
“但我还是想看看代码”	72
验证的四个层次	72
一个常见的反驳：“AI 写的测试也不可信”	73

更深一层的问题：“翻译忠实度”	73
变异测试的保障边界	73
总结	74
下一步	74
状态机：一个任务从生到死	74
一个痛苦的场景	74
核心概念	74
L1 核心：4 个状态，1 道门禁	75
最小启动：7+ 状态，2 道门禁	75
一个完整的任务走查	76
如果中间失败了呢？	77
常见错误	78
下一步	78
layer: L3 type: reference prerequisites: C09.工程_契约_产出物_证据 source: ODD.03状态-状态机与门禁 last_updated: 2026-02-16	78
状态机与门禁完整参考	78
概述	78
L1 基础	79
最小启动子集 (L2)	79
L3 核心	82
L3 扩展	84
完整 YAML 示例 (L3 任务生命周期)	85
理论来源	87
layer: L3 type: reference prerequisites: [C09.工程_契约_产出物_证据, D15.深度_环境与管道隔离] source: ODD.12执行-执行模型 last_updated: 2026-02-16	87
执行模型	87
概述	87
L1 基础	87
最小启动子集 (L2)	88
L3 核心	91
L3 扩展	95
完整 YAML 示例 (L3)	96
理论来源	99
D15. 深度_环境与管道隔离	99
layer: L3 type: reference prerequisites: [C09.工程_契约_产出物_证据, C13.工程_验证_门禁_状态机] source: ODD.0F车间-车间管理模型 last_updated: 2026-02-16	100
车间管理	100
概述	100
L1 • 基础	100
L2 • 标准	100
L3 • 严格	102
完整 YAML 示例	103

理论来源	105
layer: L3 type: reference prerequisites: C09.工程_契约_产出物_证据 source: ODD.05上下文-上下文工程 last_updated: 2026-02-16	105
上下文工程完整参考	105
概述	105
L1 基础	105
最小启动子集 (L2)	106
L3 核心	106
L3 扩展	109
完整 YAML 示例 (L3 情报官上下文组装)	110
理论来源	112
layer: L3 type: reference prerequisites: [C09.工程_契约_产出物_证据, C13.工程_验证_门禁_状态机] source: ODD.08管道-管道与组合 last_updated: 2026-02-16	113
管道与依赖完整参考	113
概述	113
L1 基础	114
最小启动子集 (L2)	114
L3 核心	115
L3 扩展	116
完整 YAML 示例 (L3 管道编排)	117
理论来源	120
D17. 深度_监控_安全与高级治理	120
赛马与预警	120
概述	120
第一部分: 赛马机制	120
第二部分: 预警系统	121
L1 • 基础	121
L2 • 标准	122
L3 • 严格	123
完整 YAML 示例	124
理论来源	127
概述	127
L1 • 基础	127
L2 • 标准	128
L3 • 严格	130
完整 YAML 示例	132
理论来源	134
L3 度量体系	134
概述	135
L1 基础	135
最小启动子集 (L2)	135
L3 核心	137
L3 扩展	140

完整 YAML 示例 (L3)	141
理论来源	144
layer: L3 type: reference prerequisites: C13.工程_验证_门禁_状态机 source: ODD.0E法宝-Bug意向图与最佳实践, ODD.10验证-ODD原生验证 last_updated: 2026- 02-16	145
度量与治理	145
概述	145
L1 基础	145
最小启动子集 (L2)	145
L3 核心	149
L3 扩展	154
完整 YAML 示例 (L3)	155
理论来源	158
E19. 哲学_底层演算与对齐基石	158
ODD 的理论基础	158
引言: 为什么需要理论基础	158
ODD 与 ECET 的理论映射	159
一、认知不对称原理	159
二、属性分层原理	160
三、阻抗设计原则	161
四、合规性传递原理	161
五、悖论熔断	162
理论来源脚注	163
ODD 与 LLM 对齐	163
核心主张	163
1-5-6-7-1 循环映射	164
七序推理循环	165
三层约束校准	166
三对张力的动态平衡	167
风险层: 认知主权保护	168
实施路径	168
理论来源脚注	169
留白	169
已收录的扩展	170
E20. 哲学_ODD与TAT责任门槛的工程映射	170
一、映射总表	170
二、TAT 的三层闭合 ODD 的三级门禁	170
三、ODD 如何防止 TAT 定义的责任逃逸	171
四、从 TAT 裁决到 ODD 编译的完整链路	171
五、一句话	171
E22. 哲学_ODD与ASTO结构编码的桥接	171
一、桥接总表	171
二、五态 → ODD 等级映射	172
三、六阶 → ODD 门禁策略	172

四、七序 → ODD 闭环映射	172
五、边界与例外的工程落地	172
六、一句话	173
ODD 不是什么	173
引言	173
一、ODD 不是银弹	173
二、ODD 不替代团队判断	174
三、ODD 不追求完美覆盖	174
四、ODD 不是永恒体系	174
五、ODD 不把开发者视为可替换的执行单元	175
契约退化：ODD 最大的战略风险	175
适用边界	175
与既有方法论的对话	176
方法论反馈环	176
理论来源脚注	176

ODD 2.0 产出物驱动开发：统一中文版

Output-Driven Development 2.0: Unified Chinese Edition

作者：Yi Fu（付毅，ODDFounder）

DOI 参考：ODD v1.0: 10.5281/zenodo.18207648 | ODD 2.0 EN: 10.5281/zenodo.18756457

定位：AI 时代以产出物为中心的工程方法论。核心闭环：Contract→Execute→Verify→Seal。

ewpage

00. ODD 2.0 单文件版：产出物驱动开发核心全览

一句话总结：代码是负债，产出物才是资产。别审查过程，要验证结果。

本文档是 ODD 2.0 (Output-Driven Development) 方法论的执行总纲。这里剥离了一切底层实现细节与复杂酉
5 分钟内带您俯瞰 ODD 2.0 的架构与哲学全景。适合布道、启蒙与快速检索。

1. 我们为什么需要 ODD？（痛点与解法）

在大模型（LLM）时代，AI 生成代码的成本已经趋近于零。但由此带来了一个致命问题：人类审查（Review）和维
传统的开发模式是：需求 → 人写代码 → 人 Review。AI 时代的错觉是：需求 → AI写代码 →
人 Review。

但实际上，让人去阅读、理解并找出 AI 几秒钟内生成的几百行代码中的逻辑漏洞与安全隐患，是极其消耗认知且
ODD 的解法：不审查过程代码，只验证最终产出物。我们将注意力从“中间态（代码是怎么写的）”转移到“验收

传统开发的逻辑是“过程可信 → 结果可信”：代码写得规范、Review 通过了，就认为结果没问题。ODD
反转了这个方向：“结果可验证 → 过程无需可信”。这是从可读性治理到可验证性治理的范式转移。

2. 核心大循环 (The Core Loop)

ODD 没有复杂的工具链绑定，其所有高阶架构都建立在以下四个不可变的基础动作之上闭环：

1. 约定 (Contract)：AI 动手前，人类（或 AI 辅助）先用结构化的“契约”（包含具体输入、输出、边界）明确需求。
 2. 执行 (Execute)：把任务扔给执行层 AI。让它不择手段地生成代码。过程是黑盒，完全不用管它。
 3. 验证 (Verify)：不读代码，只跑验证门禁（单元测试、环境构建脚本、安全扫描等）。通过了就下一步，错了就回上一步。
 4. 封存 (Seal)：绿灯通过后，将验收合格的代码与当时的测试报告做 Hash 绑定，并打上不可篡改的 Tag（如 Git tag），作为日后责任追溯的“证据”。
-

3. 验证的三态模型 (Three-State Model)

在传统 CI/CD 中，验证只有 Pass / Fail。而在应对 AI 不确定性时，ODD 引入了关键的第三态：

- PASS (绿灯)：产出物完美符合契约，机器放行，流转封存。
- FAIL (红灯)：产出物未能通过客观验收条件，机器拒收，打回原点。
- FREEZE (黄灯)：机器在验证时发现契约中有未定义的模糊边界，或者遇到了无法单纯用代码断言的美感、创意等主观条件。

此设计的核心在于：人类的注意力极其宝贵，不应浪费在日常的绿灯和红灯上，只应在黄灯（真正的不确定性）上投入精力。

4. 关键机制：防弹契约与 CAP 博弈

如果契约写得不够严谨，聪明的 AI 会使用极其拉胯的“硬编码”或“取巧代码”，在字面意义上合规地骗过验证。ODD 中被称为精灵效应。

为了防范这一点，在复杂的团队型核心任务下发前，建议引入 CAP 博弈 (Challenger - Attacker - Proposer)：

- Challenger (挑战者)：挑刺，寻找当前契约中的模糊语义。
 - Attacker (攻击者)：在沙盒中尝试用恶意构造的代码输出来骗过当前的验证门禁。
 - Proposer (防御/出题者)：被迫不断修补契约、增加测试用例，直到把契约修补到“无懈可击”。
 - 人类 (Arbitrator)：在 AI 左右互搏达到纳什均衡后，进行最终的拍板确认。
-

4.1 契约退化：ODD 最大的战略风险

CAP 能防止契约被“钻空子”，但防不了一个更根本的问题：契约本身就写错了方向。

如果团队对业务的理解有偏差，契约会忠实地把错误的理解变成严密的验证——验证全绿，方向全错。这就是“开错门”。

ODD 对此的态度是诚实承认边界，而不是假装能解决：

- CAP 对抗防止契约被钻空子，但不防止契约本身的方向错误
- Challenge 机制允许任何人随时质疑契约，降低方向错误的存活时间
- 人类裁决权不可让渡——最终签署契约的是人，责任也在人

ODD 保障的是“产出物符合契约”，不是“契约符合真实世界”。后者的正确性，最终锚定在人类的业务判断上。ODD 的设计边界，不是缺陷。

5. 架构全景与分级指南

为了更好地落地 ODD，我们依照不同的复杂度与受众人群，将方法论分为 A-E 五大类拉通（单数编排）。您可以根

快速分叉——不知道从哪开始？ → 只想了解是什么？读 A01 → 想立刻动手试试？读 B05 → 需要团队推广？先读 A03 再读 B07

A 了解 (Foundation)

适合人群：初次接触 ODD、非技术或管理人员

- A01 探讨旧有模式的核心痛点与 ODD 解法全景。
- A03 ODD 的 5 层复杂度评级与接入选型指南。

B 入门实践 (Tutorials)

适合人群：想动手尝鲜的个人开发者

- B05 手把手教你如何用 AI 写契约并跑通个人开发的第一个 ODD 闭环。
- B07 团队实战演练，以及针对历史旧项目的无痛 L0 兼容层接入法。

C 工程实践 (Core Mechanics)

适合人群：团队里的骨干研发与一线推行者

- C09 / C11 详述如何写出好契约、产出物与证据规范、对象模型与标准定义。
- C13 拆解任务生命周期（状态机）与证据链封存规范。

D 深度应用 (Advanced Engineering)

适合人群：系统架构师与自动化平台开发者

- D15 如何打造隔离的安全“车间”验证环境与多管道流转体系。
- D17 探索多 AI 模型竞跑（赛马机制）及高级预警监控度量。

E 哲学探讨 (Philosophy & Alignment)

适合人群：AI 前沿理论研究者

- E19 / E21 详解 ODD 与 LLM 的数学对齐演算逻辑，及完备术语大纲。
- E23 AI 协作指令——给 AI 阅读的 ODD 核心概述与执行规范。
- E25 ODD 的边界与演化——划定方法论的适用范围与不适用场景。

若要引用 ODD 的案例、指标、初步实证结果或失败条件，当前统一回引 ../../ODD.P99.4.案例、指标与实证与 ../../ODD.P99.6.失败条件与召回判据表.md。本文件是执行总览，不承担经验结论总表职责。

ewpage

A01. 了解_核心痛点与解法

一句话与一个循环

代码是负债，产出物才是资产。

这是 ODD（产出物驱动开发）的全部哲学。一句话就够了。

你可能遇到过这些场景

AI 写了 200 行代码，你花了 1 小时 Review，觉得没问题，上线后发现 bug。

你让 AI 重构了一个模块，跑了一遍测试全绿，三天后发现边界情况没覆盖。

半年前的一个设计决策，现在没人记得为什么这样做。翻遍了聊天记录和 Git log，拼不出完整的故事。

这些问题的根源是同一个：你在审查 AI 的过程，而不是验证 AI 的结果。

传统开发的逻辑是“过程可信 → 结果可信”：代码写得规范、Review 通过了，就认为结果没问题。ODD 反转了这个方向：“结果可验证 → 过程无需可信”。这是从可读性治理到可验证性治理的范式转移。

一个循环

ODD 只有一个核心循环，四步：

约定 → 执行 → 验证 → 封存

约定：在写代码之前，先写清楚“什么算做好”。不是模糊的需求描述，是可验证的验收条件——输入什么、输出什么、边界怎么处理。

执行：让 AI（或人）写代码。怎么写不管，写什么不管。

验证：用约定好的验收条件自动检查输出。不是人去读代码，是机器去比对结果。对了就过，错了就重来。

封存：验证通过后，把代码、验收条件、验证结果打包锁定。给未来的自己留一份完整的证据。

就这么多。没有复杂的流程，没有新的工具链，没有需要学习的框架。

30 秒动手试试

第一步：写一个验收条件

```
contract:
  title: "用户登录"
  acceptance:
    - input: { username: "test", password: "correct" }
      expected: { success: true, token: "非空字符串" }
    - input: { username: "test", password: "wrong" }
      expected: { success: false, error: "密码错误" }
```

```
- input: { username: "", password: "" }  
  expected: { success: false, error: "用户名不能为空" }
```

第二步：让 AI 写代码，然后跑验证

用你喜欢的方式验证：单元测试、集成测试、手动对比都行

`npm test` # 或 `pytest`、`go test`、`cargo test`

第三步：验证通过，封存

```
git add . && git commit -m "feat: 用户登录 - 验证通过" && git tag v0.1.0-login
```

恭喜，你刚刚完成了一个 ODD 闭环。

下一步

- 想动手练一个完整的例子？→ B05. 入门_个人闭环体验.md
- 不确定该从哪个等级开始？→ A03. 了解_评级与选型指南.md

五个核心概念

概念	定义
契约	约定“什么算做好”——可验证的验收条件
任务	契约下的具体工作单元
产出物	任务的交付结果（代码、文档等）
证据	证明产出物符合契约的记录（测试报告等）
封存	锁定通过的产出物，给未来留证据

详细定义见 C11. 工程_对象模型与标准规范.md

验证结果有三种

- PASS：通过了，产出物符合契约
- FAIL：没通过，有明确不合格项
- FREEZE：判不准，存在不确定项，等待人工裁决

FREEZE是ODD的核心设计——当机器无法判定时，让人类介入而不是强行通过或拒绝。详细说明见 C13. 工程_验证_门禁_状态机.md

ODD 不是凭空设计的。它背后有一套关于 AI 时代人机关系的上游约束判断。如果你想知道为什么 ODD 要这样设计——为什么是三值而不是二值，为什么 FREEZE 不是故障而是设计，为什么人类裁决权不可让渡——优先回看 `guide/哲学溯源.md` 与根目录 `README.md`。

若要引用案例、指标或初步实证结果，当前统一回引 `../.. /ODD.P99.4. 案例、指标与实证证据索引.md`。导论

A03. 了解_评级与选型指南

我该从哪级开始

不用一步到位，从够用的开始。
ODD 分五个级别。不是越高越好，是越合适越好。

五个级别对比

	L0 已有项目	L1 一个人	L1.5 个人+AI	L2 小团队	L3 大团队
契约	现有需求描述	文本描述	AI 生成 + 人确认	YAML 文件	版本化 + 成
验证	现有 CI/测试	手动跑测试	AI 生成测试 + 本地自动执行	CI 自动跑	属性验证 +
封存	git commit/tag	git tag	本地哈希 + 徽章	git tag + 证据归档	不可篡改证据
门禁	无	自己检查	自动判定（通过/失败）	自动门禁（通过/失败）	三值门禁（通
工具	现有工具链	编辑器 + Git	本地脚本或轻量自动化	CI/CD + YAML	完整工具链

L0 是“最低成本兼容层”——不改变现有工作流，只用 ODD 的视角重新理解你已有的实践（git commit = 产出物，PR description = 契约，CI = 验证）。

L1.5 是“个人 + AI 增强层”——你用 AI 写代码，并开始引入轻量自动化来辅助契约整理、本地验证与封存。不 CI/CD，但比 L1 的纯手动验证自动化程度更高。

5 题自测：找到你的起点

0. 你有现成的项目想引入 ODD 吗？
- A. 有，但不想改现有流程 → 从 L0 开始
 - B. 没有 / 愿意调整流程 → 继续往下答
1. 你的项目有几个人？
- A. 就我自己 → 倾向 L1 或 L1.5
 - B. 2 人以上 → 倾向 L2/L3
2. 你有 CI/CD 吗？
- A. 没有 → 倾向 L1 或 L1.5
 - B. 有 → 倾向 L2+
3. 你的项目需要审计追溯吗？
- A. 不需要 → L1 或 L1.5 或 L2 够用
 - B. 需要 → 倾向 L3
4. 你愿意花多少时间学 ODD？

- A. 5 分钟能跑起来就行 → L0 或 L1
- B. 愿意装个工具跑命令 → L1.5
- C. 愿意花 1 小时以上 → L2/L3

5. 你的代码出 bug 后果严重吗？

- A. 不严重，改了就好 → L1
- B. 很严重，可能造成损失 → L2/L3

算一下：第 0 题选 A，直接从 L0 开始。否则：一个人 + 用 AI 写代码 → L1.5；一个人 + 纯手动 → L1；多人团队选 A 多 → L2；全是 B 且第 3 题选了 B → L3。

一句话建议

大多数用 AI 写代码的个人开发者，从 L1.5 开始最合适。纯手动偏好者从 L1 开始，团队从 L2 开始。若要选具体工具，当前应先看手头项目的验证能力、留痕要求与回滚要求，而不是把某一个外部 CLI 当成默认入口。

下一步

- 想动手试试？→ B05. 入门_个人闭环体验.md
- 想深入了解？→ C09. 工程_契约_产出物_证据.md

ewpage

B05. 入门_个人闭环体验

你的第一个 ODD 闭环

这个练习需要 5 分钟。你会完成一个完整的“约定→执行→验证→封存”循环。

不需要安装任何工具。你只需要一个编辑器、一个 AI 助手（Cursor/Copilot/Claude 都行）、和 Git。

场景

你要做一个函数：把 Markdown 格式的待办列表解析成结构化数据。

输入：

- [x] 买牛奶
- [] 写周报
- [x] 修 bug #42

输出：

```
[
  { "text": "买牛奶", "done": true },
  { "text": "写周报", "done": false },
  { "text": "修 bug #42", "done": true }
]
```

第一步：写契约（2 分钟）

在你的项目里创建一个文件 `contracts/parse-todo.yaml`：

```
contract:
  id: parse-todo
  title: "解析 Markdown 待办列表"
  description: "将 Markdown checkbox 格式的待办列表解析为结构化 JSON"

acceptance:
  - name: "正常解析已完成项"
    input: "- [x] 买牛奶"
    expected: [{ "text": "买牛奶", "done": true }]

  - name: "正常解析未完成项"
    input: "- [ ] 写周报"
    expected: [{ "text": "写周报", "done": false }]

  - name: "多行混合解析"
    input: |
      - [x] 买牛奶
      - [ ] 写周报
      - [x] 修 bug #42
    expected:
      - { "text": "买牛奶", "done": true }
      - { "text": "写周报", "done": false }
      - { "text": "修 bug #42", "done": true }

  - name: "空输入返回空数组"
    input: ""
    expected: []

  - name: "忽略非待办行"
    input: |
      # 标题
      - [x] 买牛奶
      普通文本
      - [ ] 写周报
    expected:
      - { "text": "买牛奶", "done": true }
      - { "text": "写周报", "done": false }
```

注意：你还没写一行代码。你先定义了“什么算做好”。

第二步：让 AI 写代码（1 分钟）

把契约发给你的 AI 助手，说：

“请根据这个契约实现 `parseTodo` 函数，并写对应的测试。测试用例必须覆盖契约中的所有 acceptance 条目。”

AI 会给你一个实现和一组测试。不需要 Review 代码——你不关心它怎么写的，你关心它的输出对不对。

第三步：验证（1 分钟）

跑测试：

JavaScript

```
npx jest parse-todo.test.js
```

Python

```
pytest test_parse_todo.py
```

或者你用的任何语言和测试框架

两种结果：

全部通过 → 进入第四步。

有失败 → 把失败的测试结果发给 AI，说“这些用例没通过，请修复”。不需要告诉它哪里错了，让它自己看测试

第四步：封存（30 秒）

```
git add .
```

```
git commit -m "feat(parse-todo): 契约验证通过"
```

契约: `contracts/parse-todo.yaml`

验证: 5/5 用例通过

实现: `src/parse-todo.js`

```
git tag v0.1.0-parse-todo
```

你刚刚做了什么

回顾一下这 5 分钟：

1. 约定：你写了 5 个验收条件，定义了“什么算做好”
2. 执行：AI 写了代码，你没看它怎么写的
3. 验证：你跑了测试，机器告诉你对不对

4. 封存：你把一切锁定，给未来的自己留了证据

你没有 Review 一行代码。但你比 Review 更有信心——因为你验证了输出，而不是审查了过程。

这就是 ODD 的完整闭环。所有后续的概念（状态机、门禁、证据链、对抗生成）都是在这个闭环上的增强，不是新

AI 是怎么“思考”的（七序简化版）

ODD 的核心理念来自一个关键洞察：AI 的“思考方式”和人类不同。

人类的七步推理循环：

1. 感知 → 2. 解析 → 3. 干预 → 4. 设计 → 5. 物化 → 6. 回溯 → 7. 消解

但大多数 AI 只做到第3步“干预”就输出答案了。ODD 要求 AI：

- 设计：独立验证器检查候选输出
- 物化：根据验证结果决定行动（PASS/FAIL/FREEZE）
- 回溯：记录验证结果用于改进
- 消解：声明局限性，不确定时说“我不知道”

这就是为什么 ODD 要区分“验证”和“Review”——AI 的输出需要被验证，而不是被审查。

详细理论见 E19. 哲学_底层演算与对齐基石.md

下一步

- 想知道 ODD 解决哪三个核心痛苦？→ A01. 了解_核心痛点与解法.md
- 不确定该从哪个等级开始？→ A03. 了解_评级与选型指南.md
- 想把契约写得更防弹？→ C09. 工程_契约_产出物_证据.md
- 想理解“为什么验证比 Review 好”？→ C13. 工程_验证_门禁_状态机.md

ewpage

B07. 入门_团队实战与L0起步

实战进阶：15 分钟团队协作

场景设定

你已经一个人用 ODD 跑了几个任务，感觉不错。现在你想让同事也试试。但你不想让他读 20 篇文档——你需要一个 15 分钟的协作练习，让他在做中学。

这个练习两个人就能做。一个人写契约，一个人用 AI 实现。然后交换角色，再来一次。15 分钟后，你们都会理解“契约驱动”到底意味着什么。

练习场景：Markdown 表格转 JSON

实现一个函数 `md_table_to_json`，把 Markdown 表格转成 JSON 数组。

输入：

name	age	city
alice	30	beijing
bob	25	shanghai

输出：

```
[
  {"name": "alice", "age": "30", "city": "beijing"},
  {"name": "bob", "age": "25", "city": "shanghai"}
]
```

看起来简单。但“简单”正是练习对抗的好场景——越简单的需求越容易写出模糊的契约。

角色分配

- 角色 A：契约方（写契约、定义验收条件、最终验证）
- 角色 B：执行方（拿到契约后用 AI 写代码、提交交付物）

第一轮 A 写契约 B 实现，第二轮交换。

第一轮（8 分钟）

步骤 1：角色 A 写契约（3 分钟）

角色 A 独立写出契约，不和 B 讨论：

contract:

id: C-MD-001

title: "Markdown 表格转 JSON"

scope_in: "将标准 Markdown 表格文本转为 JSON 数组"

scope_out: "不处理合并单元格、不处理嵌套表格"

acceptance_criteria:

- criterion: "标准两行表格返回包含 2 个字典的列表，键为表头，值为对应单元格内容"

given_when_then: "Given 标准两行表格 When 调用 `md_table_to_json(md_text)` Then 返回包含 2 个字典"

hardness: hard

executability: EN

- criterion: "只有表头没有数据行时返回空列表"

given_when_then: "Given 只有表头没有数据行 When 调用 `md_table_to_json(header_only)` Then 返回空列表"

hardness: hard

executability: EN

- criterion: "单元格内容有前后空格时自动 trim"

given_when_then: "Given 单元格内容有前后空格 When 调用 `md_table_to_json('| alice | 30 |')` Then 返回 `[{'name': 'alice', 'age': '30', 'city': ''}]`"

hardness: hard

```

    executability: EN
- criterion: "输入不包含 Markdown 表格时抛出 ValueError"
  given_when_then: "Given 输入不包含 Markdown 表格 When 调用 md_table_to_json('hello world') The
  hardness: hard
  executability: EN
- criterion: "空字符串输入时抛出 ValueError"
  given_when_then: "Given 空字符串 When 调用 md_table_to_json('') Then 抛出 ValueError"
  hardness: hard
  executability: EN
boundary_cases:
- "表头和数据列数不一致 → ValueError"
- "单元格内容包含 | 字符（转义为 \|）→ 正确解析"
floor: "所有 acceptance_criteria 必须通过" # 描述性概念，非正式 YAML 字段，帮助理解意图（详见
red_line: "不得修改输入字符串" # 描述性概念，非正式 YAML 字段，帮助理解意图（
human_confirmed: true

```

步骤 2：角色 B 用 AI 实现（3 分钟）

角色 A 把契约发给角色 B。角色 B 把契约喂给 AI：

“根据这个契约实现 md_table_to_json 函数。函数签名：def md_table_to_json(md_text: str) -> list[dict]。”

AI 写完代码后，角色 B 不做额外修改，直接提交。

步骤 3：角色 A 验证（2 分钟）

角色 A 拿到代码，逐条跑验收条件：

输入：标准两行表格	→ 期望：2 个字典的列表	<input type="checkbox"/> /
输入：只有表头	→ 期望：[]	<input type="checkbox"/> /
输入：有前后空格的单元格	→ 期望：trim 后的值	<input type="checkbox"/> /
输入："hello world"	→ 期望：ValueError	<input type="checkbox"/> /
输入：""	→ 期望：ValueError	<input type="checkbox"/> /
输入：列数不一致	→ 期望：ValueError	<input type="checkbox"/> /
输入：单元格含 字符	→ 期望：正确解析	<input type="checkbox"/> /

全部 → 封存。任何 → 角色 B 修复后重新提交，角色 A 再次验证。

记录证据：

```

evidence:
  evidence_type: "output_verification"
  gate: "quality_check"
  result: pass
  summary: "7/7 验收条件全部通过"
  contract_id: C-MD-001
  task_id: T-MD-001

```

第二轮（7 分钟）：交换角色

角色 B 变成契约方，角色 A 变成执行方。

新场景：在第一轮的基础上，实现一个 `json_to_md_table` 函数——反过来，把 JSON 数组转回 Markdown 表格。

流程完全一样：B 写契约 → A 用 AI 实现 → B 验证。

这次角色 B 写契约时，会自然地想到第一轮踩过的坑——空输入、空格处理、特殊字符。这就是对抗思维的内化。

交叉审核（练习结束后）

两轮都做完后，交换审核对方的契约质量：

角色 A 审核角色 B 的契约：

- 验收条件够不够？能不能硬编码通过？
- 边界用例覆盖了吗？空值、特殊字符、极端输入？
- `floor` 和 `red_line` 是否可测量？（注：`floor` 和 `red_line` 是對抗博弈中的口语描述，不是 YAML 字段，写在注释里帮助理解意图即可。详见 C11. 工程_对象模型与标准规范.md）

角色 B 审核角色 A 的契约：

- 同样的三个问题。
- 额外关注：作为执行方，拿到这份契约时有没有困惑的地方？哪里需要追问？

用清晰度评估的格式记录：

```
cross_review:
  reviewer: "角色 B"
  contract_id: C-MD-001
  issues:
    - type: "边界缺失"
      description: "未定义超大表格（10000 行）的行为——内存会爆吗？"
      severity: yellow
    - type: "表达模糊"
      description: "'标准 Markdown 表格'——GitHub 风格还是其他？分隔行必须有吗？"
      severity: yellow
  verdict: "整体清晰，两个 ☐ 建议补充"
```

练习后讨论（5 分钟）

三个问题，每人回答：

1. 哪些验收条件漏了？执行时才发现契约没覆盖的情况。
2. 哪些写得太模糊？执行方需要猜测契约方的意图。
3. 如果让 AI 做“恶意执行者”，能不能钻空子？能的话，说明契约还不够防弹。

常见发现：

- “我忘了定义分隔行（`|---|---|`）是否必须存在”

- “我写了’ 返回字典列表’ 但没说键的顺序”
- “空表格和无效输入我分不清该返回空列表还是报错”

这些发现就是对抗思维的种子。下次写契约时，你会自动想到这些。

常见错误

错误	后果	修正
契约方和执行方提前讨论需求 验证时只跑正常用例 交叉审核走过场 跳过第二轮（不交换角色） 练习后不讨论	失去了“契约是否自解释“的检验机会 异常路径的 bug 漏过去了 没有真正发现契约的问题 只体验了一个视角 经验没有沉淀	契约写完前不沟通 先跑失败用例，再跑成功用例 审核时尝试“恶意实现“思维 两个角色都做一次才有完整体感 三个问题必须回答

下一步

- 还没写过契约？→ C09. 工程_契约_产出物_证据.md
- 想让契约更防弹？→ C09. 工程_契约_产出物_证据.md
- 想看更多模板？→ C09. 工程_契约_产出物_证据.md
- 回到起点 → A01. 了解_核心痛点与解法.md

L0 兼容层：零流程切入

定位：L1入门的前置补充 适合：已有项目，想用ODD视角重新理解现有实践

L0 是什么

L0是ODD的最低门槛——不改变你现有的工作方式，只是换一个视角看待你已经做的事情。

如果你已有项目，不想为了“方法论“改变流程，L0就是为你准备的。

L0 核心原则

你已经做的 = ODD 的一部分

ODD概念	L0中的对应
封存	Git tag
证据	测试报告
契约	README或需求文档

L0 怎么做

1. 给验证通过的提交打tag

```
git tag v1.0.0-login
git push origin v1.0.0-login
```

这就是“封存”——标记“这个版本已验证”。

2. 保留测试报告

```
npm test > test-report-2026-02-23.txt
# 或
pytest --junitxml=report.xml
```

这就是“证据”——记录“这个版本通过了什么验证”。

3. 在README中写下验收条件

验收条件

- [x] 用户可以用邮箱注册
- [x] 用户可以用密码登录
- [x] 错误密码返回友好提示

这就是“契约”——虽然不是YAML格式，但明确了“什么算做好”。

L0 适用场景

- 现有项目不想改变流程
 - 个人项目，验证通过就行
 - 想先了解ODD再决定是否深入
-

L0 什么时候升级

当你觉得L0不够用时，可以升级：

升级到	触发条件
L1	想让测试自动运行
L2	团队需要共享契约
L3	需要形式化验证

下一步

- 想快速上手？→ A01. 了解_核心痛点与解法.md
- 想知道选哪个等级？→ A03. 了解_评级与选型指南.md

- 想看 L1.5 的实现方式？→ 优先回看根目录 ODD. P99. 2 / ODD. P99. 3 与本地自动化脚本示例

ewpage

C09. 工程_契约_产出物_证据

契约：怎么写清楚“什么算做好”

一个痛苦的场景

你跟 AI 说：“写一个用户注册功能。”

AI 交了 150 行代码。能跑。但是：

- 密码没有强度校验
- 邮箱格式没有验证
- 用户名重复时返回 500 而不是友好提示
- 没有处理并发注册同一用户名的情况

这不是 AI 的错。是你没说清楚“什么算做好”。

契约就是解决这个问题——在写代码之前，把“做好”的定义写成机器可验证的格式。

契约不是产出物。产出物是实现了具体工程功能的成品资产；契约是从人类需求到功能产出物之间的半成品——它定义验收标准，但本身不实现功能。

ODD 契约 = 可执行规范。每一条验收条件都必须能被机器执行并返回 pass 或 fail。不能执行的描述不是 ODD 契约。

传统需求文档是菜谱（“做一道好吃的菜”），ODD 契约是质检标准（“盐度 2-3%，温度 > 75° C，大肠杆菌 = 0”）。前者需要人判断，后者机器就能检。

- “用户体验要好” → 不是契约（机器无法执行）
- “输入空用户名，返回 error ‘用户名不能为空’ ” → 是契约（机器可以跑）

判定标准：拿掉所有人类判断，这条验收条件还能被检查吗？能 → 契约。不能 → 还是需求，需要继续细化。

契约怎么来的：你不需要从零手写契约。典型流程是：(1) AI 根据你的自然语言需求生成契约草稿；(2)

契约对抗协议（CAP）对草稿进行红蓝对抗，自动发现歧义和漏洞；(3) 你审阅加固后的契约并签署。你的核心能力类似于你不需要会写法律合同，但需要能读懂律师写的合同并决定是否签字。谁签署契约，谁对验收标准的充分

契约的最小结构

一个契约只需要三样东西：

```
contract:
  id: user-register          # 唯一标识
  title: "用户注册"         # 人能读懂的名字
  acceptance_criteria:       # 验收条件（核心）
    - criterion: "正常注册：输入合法用户信息，返回 success=true 和非空 user_id"
```

```
hardness: hard          # hard=必须通过 soft=建议通过
executability: EN       # EN=可枚举验证 NEN=不可枚举, 需属性/规则验证
```

`acceptance_criteria` 是契约的灵魂。每一条验收条件用 `criterion` 描述“什么算通过”，`hardness` 标记是否必须通过（hard/soft），`executability` 标记能否用具体输入输出枚举验证（EN/NEN）。AI 写的代码必须让所有 hard 条件都成立，才算通过。

hardness说明：

- **hard**：不可商量的物理/安全约束——如“金额计算不错”、“密码不泄露”、“事务原子性”。门禁对hard条件失败不可协商
- **soft**：可协商的业务/设计约束——如“响应时间<500ms”、“代码风格统一”。门禁对soft条件失败可协商

本文是 L1/L2 入门简化版，完整字段定义见 301（对象模型完整参考）。

怎么写好验收条件

原则一：从失败开始写

新手的常见错误是只写“正常情况”。但 bug 几乎都出在异常情况。

先写失败用例，再写成功用例：

```
acceptance_criteria:
# 先写失败
- criterion: "空用户名 → 返回 error '用户名不能为空'"
  hardness: hard
  executability: EN

- criterion: "弱密码（如 '123'）→ 返回 error '密码强度不足'"
  hardness: hard
  executability: EN

- criterion: "邮箱格式错误（如 'not-an-email'）→ 返回 error '邮箱格式无效'"
  hardness: hard
  executability: EN

- criterion: "用户名已存在（前置：数据库中已有该用户名）→ 返回 error '用户名已被注册'"
  hardness: hard
  executability: EN

# 再写成功
- criterion: "合法输入 → 返回 success=true 和非空 user_id"
  hardness: hard
  executability: EN
```

原则二：写边界，不写实现

契约描述“外部行为”，不描述“内部实现”。

```
# 错误：描述了实现细节
- name: "密码用 bcrypt 加密存储"
```

```
expected: { password_hash: "以 $2b$ 开头的字符串" }
```

正确：描述外部行为

- name: "注册后能用密码登录"

steps:

- action: register
input: { username: "alice", password: "Str0ng!Pass" }
expected: { success: true }
- action: login
input: { username: "alice", password: "Str0ng!Pass" }
expected: { success: true }

你不关心密码怎么存的，你关心“注册后能登录”。

原则三：每条验收条件只验一件事

错误：一条验收条件验了三件事

- name: "注册功能正常"

```
expected: { success: true, email_sent: true, log_created: true }
```

正确：拆成三条

- name: "注册成功返回 user_id"

```
expected: { success: true, user_id: "非空字符串" }
```

- name: "注册成功后发送欢迎邮件"

```
expected: { email_sent: true, email_to: "alice@test.com" }
```

- name: "注册操作记录审计日志"

```
expected: { log_created: true, log_action: "user_register" }
```

拆开后，哪条失败一目了然。混在一起，失败了你不知道是哪个环节出了问题。

四种验收条件的写法

不是所有东西都能用“输入→输出”来验证。ODD 支持四种写法：

1. 枚举式（最常用）

固定输入，固定输出：

```
- input: { x: 2, y: 3 }  
  expected: { result: 5 }
```

适合：纯函数、数据转换、格式解析。

2. 属性式

不检查具体值，检查输出的属性：


```

- input: { length: 16 }
  expected_properties:
    - "输出是字符串"
    - "长度等于 16"
    - "包含大小写字母和数字"

```

适合：随机生成（密码、token、ID）、排序结果、搜索结果。

3. 规则式

输入输出之间有数学关系：

```

- rule: "对任意列表 L, sort(L) 的长度等于 L 的长度"
- rule: "对任意列表 L, sort(L) 的每个元素都在 L 中"
- rule: "对任意列表 L, sort(sort(L)) 等于 sort(L)"

```

适合：排序、加密解密、编码解码。

4. 对抗式

主动尝试破坏：

```

adversarial:
- attack: "SQL 注入"
  input: { username: "'; DROP TABLE users; --" }
  expected: { success: false, error: "用户名包含非法字符" }

- attack: "超长输入"
  input: { username: "a" * 10000 }
  expected: { success: false, error: "用户名超过长度限制" }

```

适合：安全相关功能、用户输入处理。

常见错误

错误	后果	修正
只写正常用例	AI 交付的代码不处理异常	先写失败用例
验收条件太模糊（“返回正确结果”）	无法自动验证	写具体的输入输出映射
一条验收条件验多件事	失败时不知道哪里出了问题	一条只验一件事
描述实现而非行为	限制了 AI 的实现自由度	只描述外部可观察的行为
漏掉前置条件	测试结果不可复现	用 precondition 声明环境假设

下一步

- 想知道怎么验证这些契约？→ C13. 工程_验证_门禁_状态机.md
- 想让契约更防弹？→ 对抗生成（第 3 层文档，按需查阅）
- 回到起点 → A01. 了解_核心痛点与解法.md

对抗：让契约变得防弹

一个痛苦的场景

你写了一个契约：“实现用户搜索功能。”

AI 交了一个函数，输入关键词，返回用户列表。技术上完全正确。但它没有分页、没有模糊匹配、搜索空字符串、10 万条记录。你的契约太模糊了，AI 钻了空子——不是故意的，是你没堵住。

这不是 AI 的 bug，是契约的 bug。

契约写得不好，后面全部白做。对抗就是在契约生效之前，主动找出它的漏洞——让契约从“大概能用”变成“很

两道关卡

一份契约在生效前必须过两道关：

第一关：清晰度评估——找出模糊点

把你写好的契约发给另一个 AI（或同事），问一个问题：

“这个契约有哪些模糊的地方？”

AI 会列出模糊点，按严重程度分级：

- 清晰：没问题，继续。
- 有点模糊：建议补充，但不补也能用。
- 很模糊：必须澄清，否则执行结果不可预测。

```
clarity_detect:
  overall: very_unclear
  score: 4
  action: must_answer
  issues:
    - type: "表达模糊"
      description: "'用户搜索功能'——搜索哪些字段？ name？ email？ phone？ 全部？"
      severity: red
    - type: "边界缺失"
      description: "未定义搜索结果为空时的行为"
      severity: yellow
    - type: "边界缺失"
      description: "未定义结果数量上限（分页？ 一次全返回？）"
      severity: red
    - type: "边界缺失"
      description: "未定义空字符串/特殊字符输入的行为"
      severity: red
```

四个模糊点，三个是 。这份契约不能直接用。

第二关：对抗生成——找出漏洞

清晰度评估消除了“说不清“的问题。但还有一种更隐蔽的问题：契约说清楚了，但说得不够严，AI 可以“字面合规“地交出垃圾。

这就是对抗生成要解决的——让 AI 扮演“恶意执行者“，故意找契约的漏洞，生成“技术上正确但完全没用“的

对抗方从三个角度攻击：

逻辑漏洞：前置条件和后置条件之间有没有断层？有没有一种输入让契约无法处理？

边界条件：空值、超长字符串、特殊字符、并发、超时——这些都定义了吗？

恶意实现（精灵效应）：能不能构造一个“偷懒“实现——通过所有验收条件但完全没用？能不能返回硬编码值骗

完整对话示例：用户搜索

下面是一个完整的对抗循环。你会看到契约如何从“漏洞百出“变成“基本防弹“。

第一轮：人写契约草案

```
contract:
  id: user-search
  title: "用户搜索"
  scope_in: "根据关键词搜索用户"
  scope_out: "不涉及权限控制"
  acceptance_criteria:
    - given: "数据库中有用户 alice、bob、charlie"
      when: "搜索 'alice'"
      then: "返回包含 alice 的结果"
    - given: "数据库中无匹配用户"
      when: "搜索 'zzz_not_exist'"
      then: "返回空列表"
  floor: "搜索功能可用"
  red_line: "不得暴露用户密码字段"
  human_confirmed: false
  review_after: '2026-06-01'
  review_triggers:
    - '距上次审核超过6个月'
```

第二轮：AI 找漏洞

[Challenger] 攻击（边界条件）：

1. 搜索空字符串会怎样？返回全部用户？如果有 10 万条呢？
2. 搜索 '%' 或 '*' 等通配符呢？会不会变成全表扫描？
3. '包含 alice 的结果'——是精确匹配还是模糊匹配？
'alice123' 算不算？'ALICE' 算不算？"

[Attacker] 攻击（恶意实现）：

"我可以写一个函数，对任何输入都返回空列表。

因为验收条件只有两条：搜索 alice 返回包含 alice 的结果，搜索不存在的返回空列表。
我只需要硬编码：if keyword == 'alice' return [alice] else return []。
技术上通过了所有验收条件。"

[Attacker] 攻击（精灵效应）：

"floor 说'搜索功能可用'——什么叫'可用'？
响应时间 30 秒算可用吗？返回结果不排序算可用吗？"

第三轮：人修补契约

```
contract:
  id: user-search
  title: "用户搜索"
  scope_in: "根据关键词对用户 name 和 email 字段做模糊搜索（大小写不敏感）"
  scope_out: "不涉及权限控制、不涉及全文搜索引擎"
  acceptance_criteria:
    - given: "数据库中有用户 alice、bob、charlie"
      when: "搜索 'ali'"
      then: "返回包含 alice 的结果（模糊匹配）"
    - given: "数据库中有用户 alice"
      when: "搜索 'ALICE'"
      then: "返回包含 alice 的结果（大小写不敏感）"
    - given: "数据库中无匹配用户"
      when: "搜索 'zzz_not_exist'"
      then: "返回空列表 []"
    - given: "空字符串输入"
      when: "搜索 ''"
      then: "返回空列表 [], 不得返回全部用户"
    - given: "特殊字符输入"
      when: "搜索 '%' 或 '*'"
      then: "作为普通字符处理，不触发通配符"
    - given: "数据库中有 500 个匹配用户"
      when: "搜索 'test'"
      then: "返回前 20 条结果 + total_count 字段"
  boundary_cases:
    - "关键词长度 > 100 → 返回错误提示"
    - "关键词为纯空格 → 等同空字符串处理"
    - "SQL 注入尝试 → 安全处理，不报错不执行"
  floor: "所有 acceptance_criteria 必须通过；响应时间 < 500ms（1000 条数据规模）"
  red_line: "不得暴露用户密码字段；不得返回未脱敏的手机号"
  human_confirmed: false
  review_after: '2026-06-01'
  review_triggers:
    - '距上次审核超过6个月'
```

第四轮：再次检查

[Challenger] 复查：

"分页参数呢？只说了'返回前 20 条'，但怎么取第 2 页？"

→ 修复：acceptance_criteria 补充分页参数 page/page_size

[Attacker] 复查：

"我还能硬编码吗？"

→ 不能了。6 条验收条件覆盖了模糊匹配、大小写、空输入、特殊字符、分页，硬编码无法同时满足所有条件。

[Arbiter] 裁决：补充分页参数后通过。提交人类确认。

修补分页后，将 human_confirmed 改为 true。

对抗记录沉淀为证据：

pk_history:

- round: 1
 - attacker: Challenger
 - attack_vector: boundary
 - issue: "空字符串、通配符、匹配方式均未定义"
 - fix: "明确模糊匹配规则、空输入返回空列表、特殊字符转义"
 - verdict: pass
- round: 2
 - attacker: Attacker
 - attack_vector: malicious
 - issue: "可硬编码两条结果通过验收"
 - fix: "增加到 6 条验收条件，覆盖多种场景"
 - verdict: pass
- round: 3
 - attacker: Challenger
 - attack_vector: logic
 - issue: "分页机制未定义"
 - fix: "补充 page/page_size 参数和验收条件"
 - verdict: pass

什么时候需要对抗

不是每个契约都需要完整的对抗循环。

场景	建议
一次性脚本、个人工具	L1: 自查清单就够了
团队项目、会被复用的模块	L2: 清晰度评估 + 一轮对抗
核心业务、安全相关、金融计算	L3: 完整 CAP 对抗循环（最多 3 轮）

自查清单（L1 最低要求）：

- 1. 有没有使用模糊词？（“大量““尽快““适当““合理”）
- 2. 验收条件能不能写出具体的测试用例？
- 3. 边界情况覆盖了吗？（空值、极大值、并发）
- 4. 有没有“字面合规但不对“的实现方式？
- 5. 地板和红线声明了吗？

常见错误

错误	后果	修正
跳过清晰度评估直接对抗	对抗方攻击的是模糊点而非漏洞，效率低	先评估再对抗，两步不跳
验收条件太少（1-2 条）	AI 可以硬编码通过	至少 4-6 条，覆盖正常/异常/
floor 写得太抽象（“功能可用”）	无法判断是否达标	floor 必须可测量（响应时间、
对抗记录没保存	下次遇到同类契约还会踩同样的坑	pk_history 作为证据沉淀
只做一轮对抗就收工	第一轮修复可能引入新漏洞	修复后至少再检查一轮

多少轮对抗合适？

场景	建议
个人项目	跳过，自己检查即可
团队项目	1轮清晰度评估
核心模块	完整CAP（3-5轮）
安全关键	强制CAP + 人工最终确认

进阶对抗：五项深度优化

上述基础对抗解决了“契约说不清“的问题。但对于更复杂的场景，我们需要更尖锐的对抗手段。

1. 代码级试探（Reverse TDD for Contracts）

问题：传统对抗停留在“语义辩论“层面——Attacker 说“这里可能有漏洞“，这种抽象描述让人类难以判断。

优化：要求 Attacker 写出能实际“钻空子“的代码片段，而非仅指出风险。

attack_protocol:
mode: code_level_probing
requirement: "发现漏洞后，必须写出可执行的恶意输入/代码"

示例：契约说"搜索返回前20条"
Attacker 不仅要说"可能被钻空子"
还必须写出：什么输入能让返回超过20条？

工作流：

1. Attacker 发现潜在漏洞
2. Attacker 编写恶意输入/测试用例/边界代码
3. Proposer 在沙盒中运行，验证漏洞是否真实存在
4. 如果攻击成功，将“案发现场”（输入+输出+代码）提交人类
5. 人类判断一行真实攻击代码是否合理，比判断抽象描述容易得多

价值：降低人类认知阻力。判断“这行会导致内存溢出的代码是否合理”比判断“这里可能存在边界风险”容易得多。

2. 业务域黑天鹅注入 (Domain-Specific Attack Vectors)

问题：AI 对抗容易陷入“通用编程套路”（空字符串、极大值、SQL 注入），但业务边界的漏洞往往更致命。

优化：为 Challenger 注入业务领域模型，从“业务崩溃”角度攻击契约。

```
domain_context:
  required: true
  injection_prompt: |
    你是一个{行业}领域的业务专家。
    从以下角度攻击这份契约：
    - 业务逻辑漏洞：是否有违背业务规则的实现方式？
    - 精灵效应：是否可以通过"字面合规但违反业务常识"的方式通过？
    - 业务黑天鹅：如果用户施加{n}次{业务操作}会怎样？
```

示例：

行业	Challenger 攻击角度
电商	“100次无门槛优惠券会怎样？”、“负数金额触发反向退款”
金融	“重复提现 Race Condition”、“小数点精度丢失导致资金缺口”
社交	“短时间内大量请求耗尽资源”、“恶意刷屏绕过限制”

价值：让契约防弹的不仅是技术边界，更是业务逻辑边界。

3. 视觉恶意对抗 (Multimodal Adversarial)

问题：对于 NEN（非精确描述节点，如 UI 界面、视觉设计），文字契约防弹难度极大。AI 可以“字面合规”地交出极其难看的设计。

优化：Attacker 生成恶意的视觉 Mockup，人类一看图就明白问题。

```
multimodal_attack:
  enabled: true
  attack_mode: "文字契约 → 恶意渲染图"
```

```
# 示例：契约说"生成醒目的登录界面"
# Attacker 生成：80%屏幕的亮粉色登录按钮
# 提交给人类："这样符合您的契约吗？"
```

工作流：

1. Proposer 生成符合文字契约的实现
2. Attacker 从“最丑但合规“角度生成对抗版本
3. 人类裁决：补充具体约束（“最大宽度400px，颜色从design_tokens.css提取”）
4. 契约补充视觉约束后，Attacker 再次攻击

价值：用多模态逼迫人类将默会知识（Tacit Knowledge）显性化。UI 设计的“常识“很难用文字描述，但一张图

4. 同谋检测（Collusion Detection）

问题：Proposer 和 Attacker 可能是同一个模型，或同一代模型，可能有意无意地“放水“。

优化：引入异构 AI 裁判池，交叉验证对抗结果。

```
collusion_prevention:
  judge_pool:
    - model: gpt-4o
    - model: gemini-2.0-flash
    - model: deepseek-v3
  consensus_threshold: 0.8
  alert_condition: "同一模型同时生成 Proposer 和 Attacker 输出"
```

5. 注意力预算升级版（Attention Budget 2.0）

问题：CAP 对抗可能产生大量需要人类裁决的结果。

优化：根据对抗结果的“新颖度“分级，仅对真正需要人类的 case 消耗预算。

```
attention_budget_v2:
  novelty_scoring:
    - "与历史相似 > 95%": 自动通过（不消耗预算）
    - "与历史相似 80-95%": 低优先级
    - "与历史相似 < 80%": 高优先级（人类必看）
    - "全新业务场景": 最高优先级
  daily_limit: 20
  alert: "单日消耗 > 80% 时触发减速"
```

下一步

- 还没写过契约？→ 见上文“契约“部分
 - 想知道怎么验证契约的执行结果？→ C13. 工程_验证_门禁_状态机.md
 - 回到起点 → A01. 了解_核心痛点与解法.md
-

产出物：一个任务一个交付物

一个痛苦的场景

一个任务“实现用户模块”，AI 交了 8 个文件：2 个 API、3 个组件、1 个数据库迁移、1 个配置文件、1 个测试文件。哪些是核心交付物？哪些是附带产物？出了 bug 该查哪个文件？没人说得清。问题不在 AI 写了太多文件——问题在于你的任务太大了，大到“交付物是什么”这个问题都没有明确答案。ODD 的原则很简单：一个任务只产生一个产出物。类型明确，交付物明确，出了问题知道查哪里。

产出物的五种类型

每个产出物必须有一个明确的类型。类型决定了你该交什么、怎么测试。

类型	说明	测试策略	示例
code	代码文件（函数、模块、API）	单元测试 / 集成测试	<code>search_user.py</code>
config	配置文件	配置校验 / 环境测试	<code>nginx.conf</code>
doc	文档	人工审查	<code>API 文档.md</code>
test	测试文件	测试本身就是验证	<code>test_search.py</code>
data	数据库脚本、迁移、种子数据	迁移测试 / 数据校验	<code>001_create_users.sql</code>

L1 项目用自然语言命名就行，保持一致即可。L2 项目建议维护一张类型表，测试策略自动关联。

一个任务一个主产出物

这是核心原则。不是说一个任务只能产生一个文件——而是说一个任务只有一个主产出物，其他都是附带的。

错误：一个任务产出一堆东西，职责不清

```
task:
  title: "实现用户模块"
  artifacts:
    - "user_api.py"
    - "user_model.py"
    - "user_component.vue"
    - "user_migration.sql"
    - "user_config.yaml"
    - "test_user.py"
```

正确：拆成多个任务，每个任务一个主产出物

```
task:
  id: T-001
  title: "实现用户搜索函数"
  contract_id: C-001
  artifact_ref: "A-001"      # 引用独立的 artifact 对象
```

```
task:
```

```

    id: T-002
    title: "创建用户表"
    contract_id: C-002
    artifact_ref: "A-002"          # 引用独立的 artifact 对象

task:
    id: T-003
    title: "实现用户搜索 API"
    contract_id: C-003
    artifact_ref: "A-003"          # 引用独立的 artifact 对象
    depends_on: [T-001, T-002]

# artifact 对象独立定义（符合 301 规范）
artifact:
    id: A-001
    artifact_type: code
    name: "search_user"
    path: "src/user/search.py"

artifact:
    id: A-002
    artifact_type: data
    name: "create_users_table"
    path: "migrations/001_create_users.sql"

artifact:
    id: A-003
    artifact_type: code
    name: "search_user_endpoint"
    path: "src/api/user_search.py"

```

每个任务的交付物一目了然。T-003 的 API 出了 bug？先查 `user_search.py`。搜索逻辑有问题？查 T-001 的 `search.py`。数据库结构不对？查 T-002 的迁移文件。

粒度判断：太大、太小、刚好

怎么判断一个任务的粒度对不对？三条规则：

契约粒度的核心原则：契约应该描述验收边界（什么条件下算通过），不应该描述完整实现（怎么做）。如果你的 AI 没有实现空间，契约也失去了“验收边界 vs 完整规格”的区分。如果你的契约粗到只有“用户能注册”，AI 有太大自由度，产出物质量不可控。好的契约粒度是：覆盖正常路径、异常路径和边界条件的验收条件，但不指定

太大 → 拆

信号：

- 产出物超过 2 个文件
- 契约的验收条件超过 10 条

- 你说不清 “这个任务的交付物是什么”

做法：按功能边界拆成多个任务，每个任务一个契约。

太小 → 合

信号：

- 产出物不到 10 行代码
- 任务之间有强耦合，改一个必须改另一个
- 单独测试没有意义

做法：合并成一个任务，用一个契约覆盖。

刚好

信号：

- 一个契约能完整描述验收条件
- 一个产出物能独立测试
- 出了问题知道查哪个文件

粒度刚好任务示例

```
task:
  id: T-010
  title: "实现 Markdown 表格解析"
  contract_id: C-010
  artifact_ref: "A-010"          # 引用独立的 artifact 对象
  input_spec: "md_text: str (包含 Markdown 表格的文本)"
  output_spec: "list[dict] (每行一个字典，键为表头)"
  acceptance_criteria:
    - "通过契约 C-010 的全部验收条件"
  error_cases:
    - code: "NO_TABLE"
      description: "输入不含表格 → 返回空列表"
    - code: "INVALID_FORMAT"
      description: "表格格式不合法 → ValueError"
  depends_on: []
  task_level: L1
```

artifact 对象独立定义（符合 301 规范）

```
artifact:
  id: A-010
  artifact_type: code
  name: "parse_md_table"
  path: "src/utils/markdown.py"
```

一个函数，一个文件，一份契约，能独立测试。这就是“刚好”。

常见错误

错误	后果	修正
一个任务产出多个不相关的文件	出 bug 不知道查哪里	拆成多个任务
产出物没有类型	不知道该怎么测试	每个产出物必须声明 <code>artifact_type</code>
任务粒度太细（一个变量一个任务）	管理成本大于开发成本	合并强耦合的小任务
任务粒度太粗（“实现整个模块”）	契约写不清，验证做不了	按功能边界拆分
附带产物没有归属	配置文件、测试文件散落各处	附带产物跟随主产出物的任务

下一步

- 还没写过契约？→ 见上文“契约”部分
- 想知道怎么记录验证证据？→ 见上文“证据与封存”部分
- 回到起点 → A01. 了解_核心痛点与解法.md

证据与封存：给未来的自己留证据

一个痛苦的场景

半年前你做了一个架构决策：用 Redis 而不是 Memcached。现在 Redis 出了问题，有人问“当时为什么选 Redis？”

你翻了 30 分钟 Slack 记录，找到了几条零散的讨论，但拼不出完整的决策链。最后你说了一句：“我记得好像是‘好像’两个字，就是证据缺失的代价。

你不是记性差，是当时没有一个地方把“为什么这样做”写下来。代码提交记录只告诉你“做了什么”，不告诉你里的讨论散落在三个频道、两周的时间跨度里，没人会把它整理成文档。

证据系统解决的就是这个问题——每一次决策、每一次验证、每一次通过门禁，都留下结构化的记录。不是给现在

核心概念

证据系统只有三个核心动作：

1. 生成证据：每次通过门禁，自动产生一条证据记录。
2. 打包封存：任务完成后，把所有证据和产出物绑在一起，锁起来。
3. 不可篡改：封存后的东西不能偷偷改。AI 不可解封（硬约束）。人类解封需审计授权，解封本身也会被记录。

常规演化路径是版本替换，不是解封：需求变了或发现问题时，推荐的做法是生成新版本的产出物，走完整的契约。一句话总结：说“做完了”不够，要拿得出证据。

四种证据类型

不是所有证据长得都一样。ODD 里有四种常见的证据类型，各有各的用途：

1. 测试报告 (test_report)

最基础的证据。代码跑了测试，结果是什么。

```
evidence:
  type: test_report
  result: pass
  summary: "12/12 tests passed, coverage 87%"
  gate: quality_check
```

回答的问题：这个东西能不能跑？跑得对不对？

2. 审查记录 (review_record)

人工审查的结论。谁看了，看完觉得怎么样。

```
evidence:
  type: review_record
  gate: acceptance
  result: pass
  reviewer_id: "zhang-san"
  summary: "验收条件全部满足，代码风格良好"
```

回答的问题：有人看过吗？看过的人觉得行吗？

3. 决策叙事 (decision_narrative)

这是最容易被忽略、但半年后最有价值的一种。它不是独立的证据类型，而是封存记录的推荐字段——记录的不是“做了什么”，而是“为什么这样做”。

证明力说明：决策叙事是当事人的自我陈述，不是独立证据。在审计或事故追溯中，它的证明力弱于测试报告（机

```
decision_narrative: |
  选择方案 B (Redis 缓存) 而非方案 A (本地缓存)，
  因为预期 QPS 将在 Q3 超过单机承载上限。
  如果 QPS 预期下调，可考虑回退到方案 A 以降低运维成本。
  替代方案：方案 C (CDN 边缘缓存) 因成本过高被否决。
```

回答的问题：当时为什么选了这条路？什么条件变了应该换路？

写决策叙事有个简单的公式——3 到 5 句话，覆盖三件事：

1. 选了什么，没选什么：“选择方案 B 而非方案 A”
2. 为什么选它：“因为预期 QPS 将超过单机上限”
3. 什么时候该重新考虑：“如果 QPS 预期下调，可回退到方案 A”

不需要写长篇大论。未来的你需要的不是论文，是一个 30 秒能读完的决策摘要。

4. 覆盖记录 (override)

当人工干预了门禁结果——比如测试没全过但你决定放行——必须留下记录。

```
evidence:
  type: override
  original_result: fail
  overridden_to: pass
  operator_id: "zhang-san"
  reason: "失败的测试是已知的环境问题，不影响功能正确性"
  timestamp: "2026-02-16T10:00:00Z"
  expiry: "2026-03-16T10:00:00Z"
  review_interval_days: 14
```

回答的问题：谁改了门禁结果？为什么改？

覆盖不是坏事，但没有记录的覆盖是坏事。

封存操作：从验证通过到锁定

封存是证据的终点。一个任务验证通过后，封存把产出物和所有证据打包在一起，锁起来。

流程很简单：

验证通过 → 收集所有证据 → 打包绑定 → git tag → 锁定

L1: 手动封存

小项目不需要复杂流程。产出物归档就算封存，git 的版本历史就是审计记录。

```
# 测试通过后，打个 tag 就行
git tag -a task-042-sealed -m "邮箱验证功能，测试全部通过"
git push origin task-042-sealed
```

证据就是测试报告本身。简单、够用。

L2: 自动封存

团队项目需要更正式的封存。系统自动完成以下步骤：

1. 收集任务关联的所有证据（测试报告、审查记录等）
2. 生成封存记录，绑定产出物版本和证据集合
3. 打 git tag，标记封存版本
4. 锁定产出物，禁止直接修改

```
seal:
  artifact_version: "task-042-v1"
  evidence_bundle:
    - evidence-001 # 测试报告
    - evidence-002 # 覆盖率报告
    - evidence-003 # 审查记录
  sealed_at: "2026-02-16T14:30:00Z"
```

```
sealed_by: "system"
decision_narrative: |
    选择正则验证而非第三方库验证邮箱格式，
    因为当前场景只需基本格式校验，无需 MX 记录验证。
    如果未来需要验证邮箱真实性，可升级为第三方服务。
```

封存后，这个产出物就是确定的、可信赖的。任何人想改，必须先解封：

```
unseal:
  reason: "发现邮箱验证遗漏了带加号的地址格式"
  by: "zhang-san"
  at: "2026-03-01T09:00:00Z"
```

解封会触发下游重新验证——如果有其他任务依赖了这个产出物，它们会被标记为 **stale**（过期），需要重新检查

一个完整的封存 YAML 示例

把上面所有东西串起来，一个完整的封存记录长这样：

```
task:
  id: task-042
  title: "用户注册接口增加邮箱格式验证"
  state: sealed
  contract_id: contract-user-register

evidence:
  - id: evidence-001
    evidence_type: test_report
    gate: quality_check
    result: pass
    summary: "12/12 tests passed, coverage 87%"
    storage_ref: "evidence/task-042/test-report.json"
    sha256: "a1b2c3d4..."

  - id: evidence-002
    evidence_type: coverage_report
    gate: quality_check
    result: pass
    summary: "line coverage 87%, branch coverage 72%"
    storage_ref: "evidence/task-042/coverage.json"
    sha256: "e5f6g7h8..."

  - id: evidence-003
    evidence_type: review_record
    gate: acceptance
    result: pass
    reviewer_id: "zhang-san"
    summary: "验收条件全部满足"
```

```
storage_ref: "evidence/task-042/review.json"
sha256: "i9j0k112..."

seal:
  artifact_version: "task-042-v1"
  evidence_bundle:
    - evidence-001
    - evidence-002
    - evidence-003
  sealed_at: "2026-02-16T14:30:00Z"
  sealed_by: "system"
  decision_narrative: |
    选择正则验证而非第三方库验证邮箱格式，
    因为当前场景只需基本格式校验，无需 MX 记录验证。
    如果未来需要验证邮箱真实性，可升级为第三方服务。
    替代方案：email-validator 库功能更全但引入了额外依赖，
    当前阶段不值得。

audit_log:
  - action: gate_pass
    target_id: task-042
    by: "system"
    at: "2026-02-16T14:00:00Z"
    details: "quality_check passed"

  - action: gate_pass
    target_id: task-042
    by: "zhang-san"
    at: "2026-02-16T14:20:00Z"
    details: "acceptance passed"

  - action: seal
    target_id: task-042
    by: "system"
    at: "2026-02-16T14:30:00Z"
    details: "sealed with 3 evidence items"
```

半年后有人问“当时邮箱验证为什么不用第三方库“，打开这个记录，30 秒就能看到答案。不用翻 Slack，不用靠记忆，不用说“好像“。

常见错误

错误	后果	修正
只记录“做了什么“，不记录“为什么“ 证据内联在聊天记录里 覆盖门禁结果但不留记录	半年后无法理解当时的决策背景 找不到、不可检索、容易丢失 无法追溯谁改了什么、为什么改	写 decision_narrative, 3-5 证据下沉为独立对象，用 evid 每次覆盖必须生成 override 记

错误	后果	修正
完成后不封存 封存后直接改代码不解封	产出物可能被意外修改，证据链断裂 产出物和证据不一致，审计链失效	验证通过后及时封存 要改必须先解封，解封触发下游

下一步

- 封存之前，任务是怎么一步步走过来的？ → C13. 工程_验证_门禁_状态机.md
- 有人故意绕过门禁怎么办？ → 见上文“证据与封存”部分

模板库

L1 契约模板（最简）

```
contract:
  id: ""
  title: ""
  acceptance_criteria:
    - criterion: ""
  boundary_cases:
    - ""
  maturity: draft
```

L2 契约模板（含 hardness、executability、maturity）

```
contract:
  id: ""
  title: ""
  scope_in: ""
  scope_out: ""
  acceptance_criteria:
    - criterion: ""
      given_when_then: "Given ... When ... Then ..." # 可选，单字符串
      hardness: soft | firm | hard
      executability: manual | semi_auto | full_auto
  boundary_cases:
    - ""
  error_cases:
    - code: ""
      description: ""
  floor: ""
```

```
red_line: ""
maturity: draft | reviewed | confirmed
human_confirmed: false
review_after: '2026-06-01'
review_triggers:
  - '距上次审核超过6个月'
```

L1 任务模板

```
task:
  title: ""
  contract_id: ""
  acceptance_criteria:
    - ""
  depends_on: []
```

L2 任务模板（含 task_level、state）

```
task:
  id: ""
  title: ""
  contract_id: ""
  artifact_ref: "" # 引用独立的 artifact 对象
  input_spec: ""
  output_spec: ""
  acceptance_criteria:
    - ""
  error_cases:
    - code: ""
      description: ""
  depends_on: []
  task_level: L1 | L2 | L3 | L4
  state: open | in_progress | done | blocked
```

封存记录模板（含 decision_narrative）

```
seal:
  artifact_version: ""
  evidence_bundle: []
  sealed_at: ""
  sealed_by: ""
  decision_narrative: "" # 推荐字段（非 301 规范字段）
```

证据记录模板

```
evidence:
  evidence_type: ""
  gate: ""
  result: pass | fail
  summary: ""
  storage_ref: ""
  sha256: ""
  contract_id: ""
  task_id: ""
```

task_level 快速判断

按代码行数

代码行数	推荐等级
<30行	L1
30-100行	L2
100-300行	L3
>300行	拆分

按复杂度

复杂度	推荐等级
单表CRUD	L1
多表关联	L2
复杂算法	L3
分布式/并发	拆分

注：取各维度最高值作为最终等级

下一步

- 想了解契约怎么写？→ 见上文“契约”部分
 - 想了解任务状态流转？→ C13. 工程_验证_门禁_状态机.md
 - 回到起点 → A01. 了解_核心痛点与解法.md
-

ewpage

C11. 工程_对象模型与标准规范

layer: L3 type: reference prerequisites: C09.工程_契约_产出物_证据 source: ODD.02对象-对象模型 last_updated: 2026-02-16

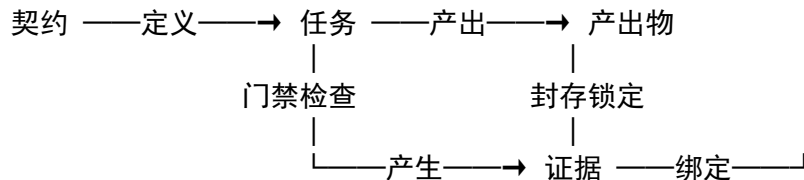
对象模型完整参考

前置知识：阅读本文前，请先完成 C09.工程_契约_产出物_证据.md。

工程与理论的桥梁：本文定义的对象模型是 ODD 工程实践的核心。如果你想了解这些机制背后的理论原理（FREEZE “、“三值门禁的理论依据”），请阅读 E19.哲学_底层演算与对齐基石.md。

概述

ODD 的所有工程活动围绕五个核心对象运转：契约（Contract）、任务（Task）、产出物（Artifact）、证据（Evidence）。它们的关系链条在任何等级下都不变：



对象字段是累加的：L2 是 L1 的超集，L3 是 L2 的超集。本文档完整列出各级字段，并标注每一级的增量。

L1 基础

L1 每个对象只保留最小必要字段，适合个人开发或小工具。

契约（Contract）

```
contract:
  id: string
  title: string
  maturity: draft | agreed | formal
  acceptance_criteria:
    - criterion: string
      hardness: hard | soft # 默认 soft
      executability: EN | NEN # 默认 EN
  boundary_cases: [string] # 至少 1 条
```

- **maturity**: 契约成熟度。Draft（草案，自然语言，允许模糊）→ Agreed（团队确认理解一致）→ Formal（YAML 格式，可自动化检查）。门禁只接受 Formal 状态的契约。
- **hardness**: hard = 不可商量的物理/安全约束；soft = 可协商的业务/设计约束。门禁优先检查 hard 条件，hard 失败则直接 FAIL。
- **executability**: EN = 可自动验证；NEN = 需人工判定。门禁对 EN 自动执行，对 NEN 路由到人工审查队列。

四种组合的处理策略：

组合	示例	门禁行为
hard + EN	响应时间 < 200ms	自动检查，失败即 FAIL
hard + NEN	不泄露用户隐私	强制人工审查，不可跳过
soft + EN	按钮颜色为蓝色	自动检查，失败可协商
soft + NEN	界面风格一致	人工审查，可降级为建议

补充说明：floor 与 red_line

正式声明：**floor** 和 **red_line** 是 ODD.06 对抗概念，用于契约评审时的思考框架。不是 YAML 契约文件的正式字段，但在契约文档中可作为注释标注。

- floor（地板）：最低验收标准——“无论如何必须达到的基础线”。如“搜索功能可用”、“响应时间 < 500ms”。
- red_line（红线）：绝对不可逾越的底线——“哪怕牺牲功能也不能违反”。如“不得暴露用户密码”、“

使用场景：在 C09.工程_契约_产出物_证据.md 中使用，用于契约评审时检查是否设置了足够的底线保护。

任务（Task）

```
task:
  id: string
  contract_id: string
  title: string
  state: pending | in_progress | review | done | rework
  acceptance_criteria: [string]
  depends_on: [task_id]
```

产出物（Artifact）

```
artifact:
  id: string
  task_id: string
  type: string           # 自由命名
  path: string           # 定位用
  depends_on: [artifact_id]
```

证据（Evidence）

```
evidence:
  type: test_report
  result: pass | fail
  summary: string
```

封存（Seal）

L1 无独立封存对象。产出物归档即视为封存。

最小启动子集（L2）

L2 在 L1 基础上增加质量控制和审计字段，适合团队协作。

契约增量

```
contract:
  # ...L1 全部字段...
  maturity: formal          # L2 契约 MUST 为 formal
  scope_in: string          # 做什么
  scope_out: string         # 不做什么
  acceptance_criteria:
    - criterion: string
      hardness: hard | soft
      executability: EN | NEN
      given_when_then: string?  # L2 建议用 GWT 格式
  error_cases: [{code, description}]
  quality_score: int         # 质量评分（0-100）
  quality_details: object    # 评分明细
  human_confirmed: bool      # 关键契约人工确认
    # false = 草案（draft）
    # true = 人类已签署确认
  deviation_budget:         # 弹性区间（可选）
    tolerance: float        # 偏离容忍度（0-1）
    protected_elements: [string]  # 不可偏离的硬约束
    flexible_elements: [string]  # 允许偏离的软约束
```

契约质量评分规则：

检查项	分值	说明
标题清晰度	10	≥ 10 字符且无歧义
描述完整度	15	≥ 50 字符，含背景和目标
验收标准数量	20	≥ 3 条得满分
验收标准可验证性	15	每条有明确 Given-When-Then
边界情况覆盖	20	≥ 3 条，覆盖最小/最大/空值
异常情况定义	15	≥ 1 条，有明确错误码
安全级别标注	5	已标注安全级别

激活阈值：≥ 80 分可直接激活；60-79 分警告建议补充；< 60 分禁止激活。

弹性区间（Deviation Budget）：

契约中可声明弹性区间，允许产出物在特定维度上偏离验收条件。偏离预算内的偏差降级为警告，不触发门禁失败规则：

- **protected_elements** 中的约束不可偏离，偏离即失败。
- **flexible_elements** 中的约束可偏离，偏离时记录为警告并生成证据。
- **tolerance** 控制整体容忍度：0 = 无弹性，1 = 所有 flexible 约束均可偏离。
- 弹性区间 MUST 由人类确认（**human_confirmed: true**），AI 不得自行扩大偏离预算。

适用场景：探索性研发、创意类产出、渐进式迁移。

任务增量

task:

```
# ...L1 全部字段...
state: blocked | pending | in_progress | quality_check | acceptance | done | rework
task_level: L1 | L2
artifact_type: string
input_spec: string
output_spec: string
side_effects:
  - type: string
    target: string
    description: string
expected_effects:
  - type: string
    target: string
    data_match: object?
rework_count: int
failure_context:
  evidence_ref: string
  summary: string
```

副作用类型（常用）：db_read / db_write、cache_read / cache_write、data_insert / data_update / data_delete、event_emission、http_request / external_call、file_read / file_write / file_create、log_write、time_get / random_generate。

产出物增量

artifact:

```
# ...L1 全部字段...
state: sealed | stale
dependency_graph:
  upstream: [artifact_id]
  downstream: [artifact_id]
```

证据增量

evidence:

```
# ...L1 全部字段...
evidence_type: test_report | coverage_report | lint_report | review_record
storage_ref: string
sha256: string
gate: string
contract_id: string
task_id: string
```

封存增量

```
seal:
  artifact_version: string
  evidence_bundle: [evidence_ref]
  sealed_at: datetime
  sealed_by: string
```

L3 核心

L3 在 L2 基础上增加形式化规格、对抗生成、动态门禁链、不可变封存。适合关键系统和 AI 多角色协作。

契约增量

```
contract:
  # ...L2 全部字段...
  formal_spec:                                # 形式化规格
    preconditions: [string]
    postconditions: [string]
    invariants: [string]
  temporal_config:                             # 时间维度
    lifecycle: string
    data_growth: string
    concurrency: string
    considerations: [string]?
    confirmed_by_human: bool?
  pk_history:                                  # 对抗生成记录
    - round: int
      issue: string
      fix: string
      verdict: string
```

形式化语法规则：允许 == != > >= < <=、AND/OR/NOT、len()、regex_match()、in、IMPLIES。约束必须可验证，

形式化示例：

```
formal_spec:
  preconditions:
    - "username != null"
    - "len(username) >= 3"
    - "regex_match(email, '.*@.*')"
  postconditions:
    - "result.success == true OR result.error != null"
    - "result.success == true IMPLIES result.token != null"
  invariants:
    - "user.password_hash != user.password_plain"
```

时间维度默认值：

- lifecycle: temporary | short_term | medium_term | long_term (默认 medium_term)
- data_growth: stable | linear | exponential (默认 linear)
- concurrency: <10 | 10-100 | 100-1000 | >1000 (默认 10-100)
- considerations: il8n | multi_timezone | data_migration | offline_support (可选)

若使用默认值可一键确认；若修改默认值 MUST 记录 confirmed_by_human。

任务增量

```
task:
# ...L2 全部字段...
task_level: L1 | L2 | L3 | L4
gate_chain: [string]           # 动态门禁序列
developed_by_workshop_id: string # 交叉审核用
```

产出物增量

```
artifact:
# ...L2 全部字段...
seal_hash: string           # 指向 seal 记录的哈希
```

证据增量

```
evidence:
# ...L2 全部字段...
evidence_type: ...L2类型... | mutation_report | adversarial_report | cross_review_record | human_r
mutation_score: float?
vulnerability_count: int?
reviewers: [string]?
consensus: bool?
workshop_id: string
```

封存增量

```
seal:
# ...L2 全部字段...
seal_hash: string           # 不可篡改的哈希
input_seal_hashes: [string] # 上游产出物的 seal_hash
gate_results: [evidence_ref] # 所有门禁证据
audit_record_ref: string
```

L3 扩展

解封机制 (Unseal)

封存后不可变；任何修改 MUST 先解封，解封 MUST 记录原因并触发审计。

```
unseal:
reason: string           # 解封原因 (bug 描述)
```

```
by: string # 解封授权人（必须是人类）
at: datetime
count: int # 累计解封次数
```

规则：

- 只有人类可以授权解封，AI 不得自行解封。
- MUST 填写解封原因。
- 解封后状态变为 rework，原封版代码自动归档。
- 解封次数累加，用于质量分析（频繁解封提示契约或测试质量问题）。

解封流程：

```
发现 bug → 人类发起解封请求（填写原因）
→ 系统记录审计日志
→ 原封版代码归档
→ 状态 sealed → rework
→ unseal_count + 1
→ 正常返工流程
```

契约成熟度迁移

Draft → Agreed 的桥梁介质是团队评审会议或异步确认；Agreed → Formal 的桥梁介质是 YAML schema 和类型系统。系统 SHOULD 记录每次状态迁移的时间戳和操作者。

字段差异速查表

字段	L1	L2	L3
contract.maturity	draft/agreed/formal	formal（强制）	formal（强制）
contract.scope_in/out	—		
contract.deviation_budget	—	（可选）	（可选）
contract.formal_spec	—	—	
contract.temporal_config	—	—	
contract.pk_history	—	—	
task.task_level	—	L1/L2	L1-L4
task.gate_chain	—	—	
task.side_effects	—		
artifact.dependency_graph	—		
artifact.seal_hash	—	—	
evidence.sha256	—		
evidence.mutation_score	—	—	
seal（独立对象）	—		（含 hash 链）
unseal	—	—	

完整 YAML Schema 示例（L3）

```
# === 契约 ===
contract:
```

```

id: "CTR-20260216-001"
title: "用户注册接口"
maturity: formal
scope_in: "实现用户注册 API, 含邮箱验证"
scope_out: "不含第三方 OAuth 登录"
acceptance_criteria:
  - criterion: "注册成功返回 JWT token"
    hardness: hard
    executability: EN
    given_when_then: "Given 合法邮箱和密码, When POST /register, Then 返回 200 + token"
  - criterion: "密码强度不低于 8 位含大小写"
    hardness: hard
    executability: EN
  - criterion: "界面提示友好"
    hardness: soft
    executability: NEN
boundary_cases:
  - "邮箱为空"
  - "密码长度恰好 8 位"
  - "邮箱已注册"
error_cases:
  - { code: "REG_001", description: "邮箱格式无效" }
  - { code: "REG_002", description: "邮箱已注册" }
quality_score: 85
human_confirmed: true
deviation_budget:
  tolerance: 0.2
  protected_elements: ["密码强度", "JWT 签名"]
  flexible_elements: ["界面提示措辞"]
formal_spec:
  preconditions:
    - "email != null"
    - "len(password) >= 8"
    - "regex_match(email, '.*@.*\\.\\.\\.\\.*')"
  postconditions:
    - "result.success == true IMPLIES result.token != null"
    - "result.success == false IMPLIES result.error_code in ['REG_001', 'REG_002']"
  invariants:
    - "user.password_hash != user.password_plain"
temporal_config:
  lifecycle: long_term
  data_growth: linear
  concurrency: 100-1000
  confirmed_by_human: true
pk_history:
  - round: 1
    issue: "未处理并发注册同一邮箱"
    fix: "增加唯一约束 + 乐观锁"

```

```

    verdict: "通过"

# === 任务 ===
task:
  id: "TSK-001"
  contract_id: "CTR-20260216-001"
  title: "实现注册接口"
  state: in_progress
  task_level: L3
  artifact_type: code_module
  input_spec: "HTTP POST body: {email, password}"
  output_spec: "HTTP Response: {success, token?, error_code?}"
  side_effects:
    - type: db_write
      target: users_table
      description: "插入新用户记录"
    - type: event_emission
      target: user_registered
      description: "发布用户注册事件"
  gate_chain: [quality_check, mutation_test, adversarial_test, cross_review]
  depends_on: []
  rework_count: 0

# === 产出物 ===
artifact:
  id: "ART-001"
  task_id: "TSK-001"
  type: code_module
  path: "src/api/register.ts"
  depends_on: []
  state: sealed
  dependency_graph:
    upstream: []
    downstream: ["ART-002"]
  seal_hash: "sha256:a1b2c3d4..."

# === 证据 ===
evidence:
  id: "EVD-001"
  evidence_type: mutation_report
  result: pass
  summary: "变异测试通过, mutation_score = 87%"
  storage_ref: "evidence/EVD-001.json"
  sha256: "sha256:e5f6g7h8..."
  gate: mutation_test
  contract_id: "CTR-20260216-001"
  task_id: "TSK-001"
  mutation_score: 0.87

```

```
workshop_id: "WS-BACKEND-01"

# === 封存 ===
seal:
  artifact_version: "1.0.0"
  evidence_bundle: ["EVD-001", "EVD-002", "EVD-003"]
  sealed_at: "2026-02-16T14:30:00Z"
  sealed_by: "WS-BACKEND-01"
  seal_hash: "sha256:a1b2c3d4..."
  input_seal_hashes: []
  gate_results: ["EVD-001", "EVD-002", "EVD-003"]
  audit_record_ref: "AUDIT-001"
```

理论来源

1. 属性分层原理（ASTO 层次支撑定理）：约束分为硬属性（改变成本 $\rightarrow\infty$ ）和软属性（改变成本有限）。没有 **hardness: hard | soft** 的设计来源于此。
 2. 可执行性梯度（NTE 可执行性梯度）：规范分为刚性可执行、柔性可执行、需解释、纯粹不可执行四级。本文 **executability: EN | NEN** 是对该梯度的工程简化。
 3. 意图五态模型（ASTO 五态模型）：自在态 \rightarrow 共识态 \rightarrow 编码态的跃迁需要介质桥接。本文中 **maturity: draft \rightarrow agreed \rightarrow formal** 对应此三阶段。
 4. P05 引理：缺失中间态会导致“虚假共识”。Draft \rightarrow Agreed 的桥接介质是评审会议，Agreed \rightarrow Formal 的桥接介质是 YAML schema。
-
-

layer: L3 type: reference prerequisites: [C09.工程_契约_产出物_证据, D15.深度_环境与管道隔离]

前置知识：阅读本文前，请先完成 C09.工程_契约_产出物_证据.md。管道相关概念参见 D15.深度_环境与管道隔离.md。

概述

契约会演进。需求变了、接口改了、字段加了删了——这些都是正常的。问题不在于“契约会变”，而在于“变了”版本迁移解决三个问题：

1. 怎么标记变更的大小——语义化版本告诉消费者“这次改动有多大”
 2. 怎么平滑过渡——三种迁移模式覆盖从小改到大改的所有场景
 3. 怎么管理旧版本的生命周期——从废弃到下线有明确的流程和时间窗口
-

L1 基础

L1 没有正式的版本管理。契约变了就直接改，手动通知受影响的人。

```
contract:
  id: user-register
  title: "用户注册"
  # 没有 version 字段
  acceptance:
    - criterion: "注册成功返回 user_id"
      hardness: hard
```

变更流程:

1. 直接修改契约文件
2. 口头或消息通知相关人员
3. 相关任务手动重新验证

适用场景: 一人项目、契约变更少、无外部消费者。

L1 的风险: 没有版本号, 无法追溯“什么时候改的、改了什么”。项目规模一大就会失控。

最小启动子集 (L2)

L2 引入语义化版本号和兼容性检查, 让变更可追踪、影响可评估。

语义化版本规则

契约版本 MUST 使用 MAJOR.MINOR.PATCH 格式:

段	对应的契约变更	示例	消费者影响
MAJOR	破坏性变更: 删除字段、改变类型、新增必填字段、收紧约束、改变语义	1.1.1 → 2.0.0	必须适配,
MINOR	向后兼容的新增: 新增可选字段、放宽约束、新增可选错误码	1.0.0 → 1.1.0	无需修改,
PATCH	Bug 修复: 不改变契约结构, 只修正实现中的错误	1.1.0 → 1.1.1	无感知

```
contract:
  id: user-register
  version: "1.0.0"
  deprecated: false
  sunset_date: null
  title: "用户注册"
  acceptance:
    - criterion: "注册成功返回 user_id"
      hardness: hard
```

任务关联契约版本:

```
task:
  id: "TSK-REG-001"
  contract_id: user-register
  contract_version: "1.0.0"
```

兼容性规则

向后兼容变更（安全的 MINOR 变更）：

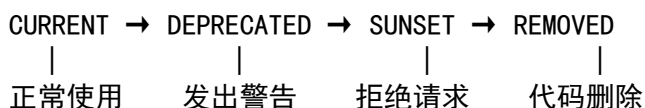
- 新增可选输入字段（MUST 有默认值）
- 新增输出字段
- 放宽输入约束（如最小长度从 8 降到 6）
- 新增可选错误码

破坏性变更（危险的 MAJOR 变更）：

- 删除输入/输出字段
- 改变字段类型（如 string \rightarrow int）
- 新增必填输入字段
- 收紧输入约束（如最大长度从 100 降到 50）
- 改变字段语义（如金额单位从“分”变“元”）

破坏性变更 MUST 递增 MAJOR 版本。创建新版本时，系统 MUST 自动对比新旧版本的 input_spec / output_spec，检测是否存在破坏性变更。破坏性变更 MUST 人工确认后才能提交。

版本生命周期



- DEPRECATED 期 SHOULD \geq 3 个月（关键系统 \geq 6 个月）
- SUNSET 后 MUST 返回明确错误，指引升级路径

三种迁移模式

模式一：适配器模式（差异小，可自动转换）

旧契约调用 \rightarrow 适配器转换 \rightarrow 新契约执行 \rightarrow 适配器转换 \rightarrow 旧格式响应

- 输入转换：为新增字段填默认值
- 输出转换：移除旧版本不认识的字段
- 适用于：MINOR 变更、字段新增

模式二：扩展-收缩模式（字段重命名、结构重组）

阶段1：扩展 — 同时支持新旧字段（MINOR 版本）

阶段2：标记废弃 — 旧字段标记 deprecated

阶段3：收缩 — 移除旧字段（MAJOR 版本）

- 适用于：字段重命名、数据结构调整
- 核心原则：先加后删，给足过渡期

模式三：版本路由模式（差异大，无法简单适配）

请求携带版本号 \rightarrow 路由到对应版本的实现

- 版本号不存在时，路由到最近的兼容版本
- 适用于：MAJOR 变更、接口完全重设计

迁移 YAML diff 示例

v1 → v2 的具体变更（新增 phone 可选字段，删除 nickname 字段）：

```
# === v1.0.0 ===
contract:
  id: user-register
  version: "1.0.0"
  input_spec:
    username: { type: string, required: true }
    email: { type: string, required: true }
    password: { type: string, required: true }
    nickname: { type: string, required: false }
  output_spec:
    user_id: { type: string }
    created_at: { type: datetime }

# === v2.0.0 ===
# 变更说明:
#   - 删除 nickname (破坏性 → MAJOR)
#   - 新增 phone (可选, 有默认值 → 本身是 MINOR, 但因同版本有破坏性变更, 合并为 MAJOR)
#   - 新增输出 phone_verified
contract:
  id: user-register
  version: "2.0.0"
  deprecated: false
  previous_version: "1.0.0"
  breaking_changes:
    - field: nickname
      change: removed
      migration: "nickname 数据迁移到 display_name 字段 (见迁移脚本 MIG-042) "
  input_spec:
    username: { type: string, required: true }
    email: { type: string, required: true }
    password: { type: string, required: true }
    phone: { type: string, required: false, default: null }
  output_spec:
    user_id: { type: string }
    created_at: { type: datetime }
    phone_verified: { type: bool, default: false }
```

完整 L2

在最小启动基础上，L2 完整模式增加下游影响分析：

下游影响分析——变更一个契约，哪些任务需要重新验证：

```
impact_analysis:
  changed_contract: user-register
  from_version: "1.0.0"
```



```

to_version: "2.0.0"
change_type: major
affected_tasks:
  - task_id: "TSK-PROFILE-001"
    contract_id: user-profile
    dependency: "读取 user-register 的 nickname 字段"
    impact: high
    action_required: "适配新版本, 改用 display_name"
  - task_id: "TSK-WELCOME-001"
    contract_id: welcome-email
    dependency: "读取 user-register 的 output"
    impact: low
    action_required: "无需修改, 新增字段不影响"
summary:
  total_affected: 2
  high_impact: 1
  action_items: 1

```

影响分析规则:

- MINOR 变更: 自动标记下游为“无需修改”
- MAJOR 变更: MUST 逐一评估下游任务, 标记影响等级
- 高影响任务 MUST 在迁移完成前阻止旧版本 SUNSET

L3 核心

L3 在 L2 基础上增加: 废弃中间件、迁移追踪、DB 层版本管理。

废弃中间件

自动处理不同版本状态的请求:

- DEPRECATED 版本: 请求正常处理, 但响应头增加 **Deprecation: true** + **Sunset: <date>**
- SUNSET 版本: 拒绝请求, 返回 410 Gone + 升级指引
- 所有使用废弃版本的调用者 MUST 被记录和通知

```

deprecation_middleware:
  rules:
    - version_status: deprecated
      behavior: process_with_warning
      response_headers:
        Deprecation: "true"
        Sunset: "2026-06-01T00:00:00Z"
        Link: "</api/v2/register>; rel=\"successor-version\""
    - version_status: sunset
      behavior: reject
      response_code: 410
      response_body:

```

```
error: "VERSION_SUNSET"
message: "此版本已下线, 请升级到 v2.0.0"
migration_guide: "https://docs.example.com/migrate/register-v2"
```

迁移追踪

```
migration_tracking:
  contract_id: user-register
  versions:
    - version: "1.0.0"
      status: deprecated
      deprecated_at: "2026-02-01T00:00:00Z"
      sunset_at: "2026-06-01T00:00:00Z"
      active_consumers: 3
      migration_guide_ref: "MIG-GUIDE-042"

    - version: "2.0.0"
      status: current
      active_consumers: 12
      migration_guide_ref: null
```

追踪规则:

- 系统 MUST 统计各版本的使用情况 (active_consumers)
- SUNSET 前 MUST 确认 active_consumers = 0 (或人工确认强制下线)
- 迁移进度 SHOULD 可视化, 显示各消费者的迁移状态

DB 层版本管理

数据库层的版本管理遵循扩展-收缩模式:

```
db_version_management:
  strategy: expand_contract
  steps:
    - phase: expand
      description: "新增 display_name 列, 保留 nickname 列"
      migration: "ALTER TABLE users ADD COLUMN display_name VARCHAR(100)"
      version: "1.1.0"

    - phase: backfill
      description: "将 nickname 数据复制到 display_name"
      migration: "UPDATE users SET display_name = nickname WHERE display_name IS NULL"
      version: "1.1.0"

    - phase: switch
      description: "应用层切换到读写 display_name"
      version: "2.0.0"

    - phase: contract
      description: "确认无调用后删除 nickname 列"
```

```

migration: "ALTER TABLE users DROP COLUMN nickname"
precondition: "active_consumers(nickname) = 0"
version: "2.1.0"

```

DB 版本规则:

- 存储过程/函数 SHOULD 带版本后缀 (如 `create_user_v1`、`create_user_v2`)
- DROP COLUMN MUST 在确认无调用后执行
- 大表结构变更 MUST 使用在线 DDL 工具 (如 `pt-online-schema-change` / `pg_repack`)，避免锁表

L3 扩展

L1/L2/L3 版本管理差异速查

维度	L1	L2	L3
版本号	无	语义化版本	语义化版本
兼容性检查	无	自动对比 + 人工确认	自动对比 + 人工确认
迁移模式	直接改	三种模式可选	三种模式 + 废弃中间件
下游影响分析	无	手动评估	自动识别 + 追踪
版本生命周期	无	CURRENT → DEPRECATED → SUNSET	完整生命周期 + 消费者统计
DB 版本管理	无	无	扩展-收缩 + 在线 DDL
废弃通知	口头	文档记录	中间件自动通知

跨契约版本协调

当契约 A 依赖契约 B，且 B 发生 MAJOR 变更时，A 有三条路径:

cross_contract_coordination:

```

scenario: "契约 B (user-register) 从 v1 升级到 v2"
dependent_contract: "契约 A (user-profile)"

```

path_1_lock:

```

name: "锁定旧版本"
description: "A 继续依赖 B 的 v1 (B 的 v1 MUST 在 DEPRECATED 期内可用)"
risk: "B 的 v1 最终会 SUNSET，届时 A 必须迁移"
适用: "A 的迁移成本高，需要时间准备"

```

path_2_sync:

```

name: "同步升级"
description: "A 升级自身版本以适配 B 的 v2"
risk: "A 的变更可能触发 A 的下游连锁升级"
适用: "变更范围可控，团队有余力"

```

path_3_adapter:

```

name: "适配器隔离"
description: "在 A 和 B 之间插入适配管道，转换接口差异"
risk: "增加一层间接性，长期维护成本"
适用: "A 和 B 由不同团队维护，协调成本高"

```

依赖链中的版本变更 SHOULD 通过功能树影响分析自动识别受影响的上游契约。

版本迁移的核心原则

- 设计时考虑演进：字段命名有前瞻性，用可扩展结构（对象优于扁平字段）
 - 变更前评估影响：运行兼容性检查，分析调用者数量和影响等级
 - 渐进式迁移：先扩展后收缩，给足废弃期，不搞一刀切
 - 主动通知消费者：不要等消费者踩坑，主动推送迁移信息和时间表
-

完整 YAML 示例（L3）

=== 契约版本迁移完整示例 ===

旧版本契约（已废弃）

```
contract_v1:
  id: user-register
  version: "1.0.0"
  status: deprecated
  deprecated_at: "2026-02-01T00:00:00Z"
  sunset_date: "2026-06-01T00:00:00Z"
  input_spec:
    username: { type: string, required: true }
    email: { type: string, required: true }
    password: { type: string, required: true }
    nickname: { type: string, required: false }
  output_spec:
    user_id: { type: string }
    created_at: { type: datetime }
```

新版本契约（当前）

```
contract_v2:
  id: user-register
  version: "2.0.0"
  status: current
  previous_version: "1.0.0"
  breaking_changes:
    - field: nickname
      change: removed
      migration: "数据迁移到 display_name"
  input_spec:
    username: { type: string, required: true }
    email: { type: string, required: true }
    password: { type: string, required: true }
    phone: { type: string, required: false, default: null }
  output_spec:
    user_id: { type: string }
    created_at: { type: datetime }
```

```

    phone_verified: { type: bool, default: false }

# 迁移追踪
migration_tracking:
  contract_id: user-register
  versions:
    - version: "1.0.0"
      status: deprecated
      deprecated_at: "2026-02-01T00:00:00Z"
      sunset_at: "2026-06-01T00:00:00Z"
      active_consumers: 3
      consumers:
        - { id: "SVC-PROFILE", status: migrating, eta: "2026-03-15" }
        - { id: "SVC-WELCOME", status: migrated, completed: "2026-02-10" }
        - { id: "SVC-ANALYTICS", status: not_started }
      migration_guide_ref: "MIG-GUIDE-042"
    - version: "2.0.0"
      status: current
      active_consumers: 12

# 废弃中间件配置
deprecation_middleware:
  rules:
    - version: "1.0.0"
      status: deprecated
      behavior: process_with_warning
      response_headers:
        Deprecation: "true"
        Sunset: "2026-06-01T00:00:00Z"
        Link: "</api/v2/register>; rel=\"successor-version\""
      adapter:
        input_transform:
          - action: drop_field
            field: nickname
        output_transform:
          - action: drop_field
            field: phone_verified

# DB 迁移计划
db_migration_plan:
  strategy: expand_contract
  steps:
    - phase: expand
      sql: "ALTER TABLE users ADD COLUMN display_name VARCHAR(100)"
      executed_at: "2026-01-20T10:00:00Z"
    - phase: backfill
      sql: "UPDATE users SET display_name = nickname WHERE display_name IS NULL"
      rows_affected: 45000

```

```

    executed_at: "2026-01-20T10:05:00Z"
  - phase: switch
    description: "应用层切换到 display_name"
    executed_at: "2026-02-01T00:00:00Z"
  - phase: contract
    sql: "ALTER TABLE users DROP COLUMN nickname"
    precondition: "active_consumers(nickname) = 0"
    status: pending
    earliest_execution: "2026-06-01T00:00:00Z"

# 下游影响分析
impact_analysis:
  changed_contract: user-register
  from_version: "1.0.0"
  to_version: "2.0.0"
  change_type: major
  affected_tasks:
    - task_id: "TSK-PROFILE-001"
      impact: high
      reason: "读取 nickname 字段"
      action: "改用 display_name"
      status: migrating
    - task_id: "TSK-WELCOME-001"
      impact: low
      reason: "仅读取 user_id 和 email"
      action: "无需修改"
      status: verified
    - task_id: "TSK-ANALYTICS-001"
      impact: medium
      reason: "统计报表包含 nickname 维度"
      action: "切换到 display_name 维度"
      status: not_started

# 迁移证据
migration_evidence:
  id: "EVD-MIG-042"
  evidence_type: migration_report
  result: in_progress
  summary: "v1→v2 迁移进行中, 1/3 消费者已完成"
  sha256: "sha256:c3d4e5f6..."

```

理论来源

1. 结构演进原理（ASTO 结构病理学 A01/A02）：结构建立后进入自维护阶段，每次修补引入新复杂度。语义化 MAJOR/MINOR/PATCH 分级本质上是对“变更引入的复杂度”的量化——PATCH 几乎不引入复杂度，MINOR 引入可控复杂度，MAJOR 引入需要全链路评估的复杂度。

- 2. 属性分层原理（ASTO 层次支撑定理）：向后兼容变更只影响软属性（新增可选字段），破坏性变更触及硬属性（删除必填字段）必须递增 MAJOR 版本并触发全链路影响分析。
- 3. 意图五态模型（ASTO 五态模型）：版本生命周期 CURRENT → DEPRECATED → SUNSET → RE-MOVED 对应意图从“活跃态”到“消亡态”的退化路径。DEPRECATED 期是给消费者的“共识重建窗口”——旧共识（v1 接口）需要时间迁移到新共识（v2 接口）。
- 4. 渐进式迁移原则：扩展-收缩模式的“先加后删”策略来源于“中间态不可跳过”原则（ASTO P05 引理）。跳过中间态（直接删除旧字段）会导致消费者的“虚假兼容”——代码还在引用已不存在的字段。

契约保鲜机制

契约不仅会演进，还会“过期”。业务变了、外部依赖变了，但契约没人更新——导致验证通过的产出物实际已不匹配。

保鲜字段

```
contract:
  review_after: "2026-06-01"    # 到期复核日期
  owner: "zhang-san"            # 负责人
```

触发条件

条件	动作
距上次审核超过6个月	标记为待复核
外部依赖重大变更	强制复核
业务需求重大变更	强制复核

过期后果

- 产出物自动标记为 stale
- 新提交需强制走CAP
- 门禁拒绝未复核的契约

layer: L3 type: reference prerequisites: [C09. 工程_契约_产出物_证据, D15. 深度_环境与管道隔离]
前置知识: C09. 工程_契约_产出物_证据.md — 理解契约的结构和验收条件后再阅读本文。

概述

项目规模一大，产出物散落各处，“我要的那个功能到底在哪”没人能答。功能树就是产出物网络的目录——按业务功能分层索引，让任何人都能从功能出发找到对应的产出物、契约和管道，也能在一个产出物变动时立刻感知。功能树按业务能力划分，不按代码目录划分。它管“业务视角的定位”，依赖图管“技术视角的传导”，两者互补。核心操作四件事：

- 定位：给定功能名 → 找到对应产出物及状态
- 追溯：给定产出物 → 找到它属于哪些功能节点
- 影响分析：产出物变动 → 沿功能树 + 依赖图找出所有受影响功能
- 覆盖检查：检测是否有产出物不属于任何功能节点（孤立产出物）

L1 • 基础

功能树 = 一份 Markdown 清单，手动维护。需求和产出物混在一棵树里，不做区分。

在项目根目录放一份 功能树.md，按业务功能列出对应的产出物：

功能树

用户管理

- 注册功能 → `user-register` 契约 → 产出物: user-register-api, user-register-test
- 登录功能 → `user-login` 契约 → 产出物: user-login-api, user-login-test

订单处理

- 创建订单 → `order-create` 契约 → 产出物: order-create-api, order-create-test
- 支付 → `order-pay` 契约 → 产出物: order-pay-api, payment-gateway-adapter

影响分析靠手动搜索。覆盖检查靠人工比对。产出物少于 20 个时完全够用。

L2 • 标准

最小启动子集

L2 的核心变化：功能树分裂为两棵树。

- 需求树（Requirement Tree）：人维护，节点是业务功能，相对稳定。回答“要实现什么业务能力”。
- 产出物树（Artifact Tree）：自动生成（从代码仓库 + 契约扫描），节点是技术交付物。回答“有哪些技术交付物”。

分裂的原因：需求和产出物的关注点不同、生命周期不同、受众不同。“用户能登录”这个需求可能三年不变，但 API 端点可能重构三次。混在一棵树里，需求的稳定性会被产出物的频繁变动拖累。

最小启动只需做两件事：

1. 建立需求树（手动维护业务功能节点）
2. 在契约中填写 requirement_ref 字段，把契约关联到需求节点

contract:

```
id: contract-login-api
title: "登录 API"
requirement_ref: req-user-login # 指向需求树节点
```

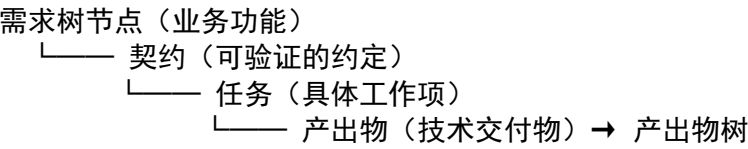
多对多场景（如通用鉴权中间件服务多个需求）使用 requirement_refs:

contract:

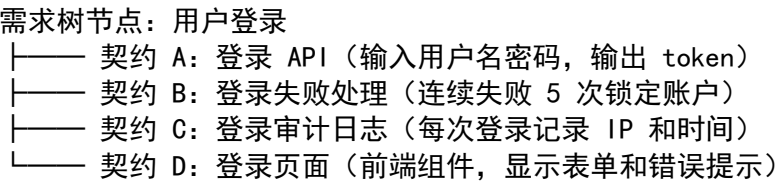
```
id: contract-auth-middleware
requirement_refs: [req-user-login, req-user-register, req-password-reset]
```


完整 L2

需求树节点与契约的关系 需求树节点是契约的“业务语义容器“，不是契约的替代品：



一个需求节点对应一组契约：



两者的职责区分：

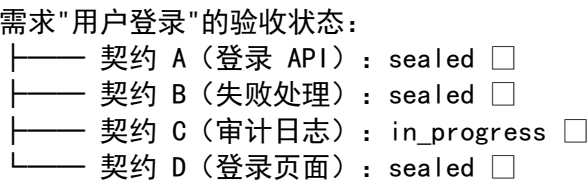
维度	需求树节点	契约
粒度	粗（业务功能）	细（可验证的单个约定）
语言	业务语言（“用户能登录”）	技术语言（“POST /api/login 返回 200”）
受众	产品经理、业务方、审计	开发者、AI、门禁系统
生命周期	稳定（跟随业务演进）	频繁变动（跟随实现迭代）
验收	定性（“功能可用”）	定量（输入→输出映射）

需求树节点定义

```
requirement:
  id: string
  title: string # 业务功能名称
  description: string # 业务语言描述
  parent: string | null
  children: [string]
  priority: high | medium | low
  security_level: high | medium | low # 继承到下属所有契约和产出物
  acceptance_summary: string # 定性验收描述（给人看，不给机器执行）
  contracts: [contract_id] # 自动生成（从 requirement_ref 反向聚合）
  status: active | deprecated | planned
  owner: string
```

注意：需求节点没有 acceptance_criteria（那是契约的事），没有 examples（那也是契约的事）。

需求级验收（自动聚合） 需求节点的验收 = 其下所有契约全部通过门禁（sealed）。不需要额外的需求级验收



需求状态：3/4 完成，未就绪

当所有契约都 sealed 时，需求节点自动标记为“已实现”。

安全等级继承 需求节点的 `security_level` 自动继承到其下所有契约、任务、产出物。继承规则：只能升级，不

产出物树（自动生成） 产出物树从代码仓库和契约自动生成，按技术维度组织。人只维护需求树，产出物树和明

双树查询能力

- `locate_by_requirement`(需求名) → 返回该需求下所有契约及其产出物状态
 - `locate_by_artifact`(产出物ID) → 返回该产出物所属的需求节点列表
 - `requirement_progress`(需求ID) → 返回需求完成度 (sealed 契约数 / 总契约数)
 - `orphan_contracts`() → 返回没有 `requirement_ref` 的契约列表
 - `orphan_artifacts`() → 返回不属于任何功能节点的产出物列表
-

L3 • 严格

L3 核心

在 L2 双树基础上增加：强制覆盖和双向影响分析。

强制覆盖规则：

- 每个产出物 MUST 至少归属一个需求节点（通过契约的 `requirement_ref` 关联），否则封存门禁拒绝通过
- 每个契约 MUST 有 `requirement_ref` (L3 强制)，不允许“无归属契约”

双向影响分析：

正向（需求变更 → 受影响的产出物）：

需求“用户登录”变更（如增加双因素认证）

- 查找该需求下所有契约
- 查找每个契约下所有产出物
- 标记所有相关产出物为 `stale`
- 通知所有产出物的 `owner`

反向（产出物缺陷 → 受影响的需求）：

产出物 `payment-gateway-adapter` 发现安全漏洞

- 依赖图：级联计算受影响产出物集合 `S`
- 需求树：映射出受影响需求集合 `F`
- 自动通知 `F` 中每个需求的 `owner`
- 审计日志记录：变更源 → 影响范围 → 通知对象 → 处理结果

L3 扩展

变更留痕与需求树快照：

- 需求树节点变更（新增/废弃/重新映射）MUST 记录在审计日志中
- 需求树版本 MUST 与产出物封存版本对齐——封存时的需求树快照作为证据的一部分

自动同步：

- 新契约创建时，系统提示填写 requirement_ref（或拒绝创建）
- 需求节点废弃时，自动检查其下是否有未封存的契约，有则警告
- 产出物删除/废弃时，自动更新产出物树，检查是否导致需求节点“空挂”

按需求维度的质量度量汇总：

用户管理

|—— 登录：覆盖率 92%，返工率 5%，Bug 数 2
|—— 注册：覆盖率 88%，返工率 8%，Bug 数 1
|—— 密码重置：覆盖率 75%，返工率 15%，Bug 数 4 ← 这里有问题

质量指标从产出物级别自动汇总到需求级别，提供业务视角的质量全景。

完整 YAML 示例

以下展示一个电商系统的 L3 完整结构：

—— 需求树（人维护） ——

```
requirement_tree:
  nodes:
    - id: root
      title: 电商平台
      parent: null
      children: [user-mgmt, order-mgmt]
      security_level: medium
      status: active
      owner: platform-team

    - id: user-mgmt
      title: 用户管理
      parent: root
      children: [req-user-login, req-user-register]
      security_level: high
      status: active
      owner: user-team

    - id: req-user-login
      title: 用户登录
      description: "用户能通过用户名密码登录系统"
      parent: user-mgmt
      children: []
      priority: high
      security_level: high
      acceptance_summary: "用户能正常登录，失败有保护机制"
      contracts: [contract-login-api, contract-login-fail, contract-login-audit]
      status: active
      owner: user-team
```

```

# —— L3 扩展字段 ——
version: 3
change_log:
  - changed_at: "2026-01-15T10:00:00Z"
    change_type: add_contract
    detail: "新增双因素认证契约"
    operator: "tech-lead"

- id: order-mgmt
  title: 订单管理
  parent: root
  children: [req-order-create, req-order-pay]
  security_level: medium
  status: active
  owner: order-team

- id: req-order-pay
  title: 支付
  description: "用户能完成订单支付"
  parent: order-mgmt
  children: []
  priority: high
  security_level: high # 升级：支付比订单管理更敏感
  acceptance_summary: "支付流程完整，异常有回退"
  contracts: [contract-pay-api, contract-pay-callback]
  status: active
  owner: order-team

# —— 契约（带 requirement_ref） ——
contracts:
- id: contract-login-api
  title: "登录 API"
  requirement_ref: req-user-login
  acceptance:
    - name: "正常登录"
      input: { username: "alice", password: "Str0ng!Pass" }
      expected: { success: true, token: "非空字符串" }
    - name: "密码错误"
      input: { username: "alice", password: "wrong" }
      expected: { success: false, error: "用户名或密码错误" }

- id: contract-pay-api
  title: "支付接口"
  requirement_ref: req-order-pay
  acceptance:
    - name: "正常支付"
      input: { order_id: "ORD-001", amount: 99.00 }
      expected: { success: true, transaction_id: "非空字符串" }

```

```
# —— 产出物树（自动生成） ——
artifact_tree:
  nodes:
    - id: auth-service
      type: module
      children: [login-api, login-test]
    - id: login-api
      type: artifact
      requirement_refs: [req-user-login]
      state: sealed
    - id: payment-gateway-adapter
      type: artifact
      requirement_refs: [req-order-pay]
      state: sealed

# —— L3 需求树快照 ——
requirement_tree_snapshot:
  snapshot_id: "snap-2026-02-15"
  tree_version: 12
  taken_at: "2026-02-15T14:30:00Z"
  triggered_by: "contract-login-api 封存"
  nodes: [...] # 完整需求树节点副本
```

理论来源

- 1. 功能树（Functional Tree）概念源自 ASTO 结构论中的“索引层“思想——任何复杂系统都需要一个与执行
- 2. 双树分裂（需求树 + 产出物树）对应 ASTO 中“关注点分离“原则——不同生命周期、不同受众的信息不应
- 3. 安全等级只升不降 继承规则源自安全工程中的“最高水位线“原则——子系统的安全等级不得低于其所属系
- 4. 双向影响分析 结合了依赖图的技术传导和功能树的业务映射，实现从任意节点出发的全链路追踪。

与契约版本迁移的联动

功能树是契约版本迁移的核心工具——当契约变更时，通过功能树确定受影响的上游契约。

联动机制

场景	功能树的作用
契约版本升级	沿功能树向上游查找受影响的其他契约
依赖影响分析	结合依赖图确定影响范围
迁移计划生成	自动列出需要重新验证的产出物

C13. 工程_验证_门禁_状态机

验证：不 Review 代码，验证输出

一个让你不舒服的事实

你让 AI 写了一个函数，然后花 20 分钟逐行 Review。你觉得没问题，合并了。

三天后，线上报了一个 bug。你回去看那段代码，发现 bug 就在你 Review 过的那 20 行里。你当时看了，但没看出来。

这不是你的问题。这是 Code Review 这个方法本身的问题。

为什么 Review 在 AI 时代失效

Code Review 有一个隐含假设：审查者能理解代码的意图和逻辑。

人写的代码，审查者可以通过变量命名、注释、提交历史来推断意图。但 AI 写的代码没有“意图”——它是统计模型的输出，看起来合理，但合理不等于正确。

更致命的是速度问题。人一天写 200 行代码，Review 跟得上。AI 一天写 2000 行，你 Review 不过来。就算你 Review 了，人类阅读代码的错误检出率大约在 60-70%——也就是说，每 10 个 bug 你能发现 6-7 个，剩下 3-4 个会溜进生产环境。

当 AI 把产出速度提高了 10 倍，溜进去的 bug 数量也提高了 10 倍。

适用条件说明：以上论述在 AI 产出占团队代码显著比例时成立。在小规模修复、受限的 AI 输出场景、或简单到可以快速扫视的任务中，Code Review 仍可能有效。ODD 不主张“永远不要 Review”，而是主张“不要把 Review 作为主要质量门禁”——验证是安全网，Review 是额外的信心（见下文“但我

验证和 Review 的区别

维度	Code Review	输出验证
检查对象	代码（过程）	输出（结果）
检查方式	人阅读代码	机器对比输入输出
可扩展性	受限于人的阅读速度	受限于机器的执行速度
一致性	取决于审查者的状态和经验	每次结果一样
覆盖率	审查者觉得看了就算覆盖	每条验收条件都有明确的通过/失败

一句话：Review 问的是“这段代码看起来对吗？”，验证问的是“这段代码的输出对吗？”

前者是主观判断，后者是客观事实。

怎么做输出验证

第一步：有契约

验证的前提是你有验收条件。没有验收条件，你验证什么？

ODD 的契约必须是可执行规范——每一条验收条件都能被机器执行并返回 pass/fail。“用户体验要好”不是 ODD 契约；“输入合法用户信息，返回 success=true 和非空 user_id”才是。不能执行的描述不构成 ODD 契约。

contract:

id: calculate-shipping

title: "计算运费"

acceptance_criteria:

- criterion: "同城订单: weight=2, distance=10, type=same_city → fee=8.0"
hardness: hard
executability: EN
- criterion: "跨省订单: weight=2, distance=500, type=cross_province → fee=25.0"
hardness: hard
executability: EN
- criterion: "超重附加费: weight=30, distance=10, type=same_city → fee=45.0"
hardness: hard
executability: EN
- criterion: "零重量拒绝: weight=0, distance=10, type=same_city → error='重量必须大于零'"
hardness: hard
executability: EN

第二步：让 AI 写代码和测试

把契约发给 AI：

“根据这个契约实现 calculateShipping 函数。测试必须覆盖所有 acceptance 条目。”

第三步：跑验证，看结果

\$ npm test

- ☐ 同城订单: fee = 8.0
- ☐ 跨省订单: fee = 25.0
- ☐ 超重附加费: 期望 45.0, 实际 38.0
- ☐ 零重量拒绝: error = "重量必须大于零"

3 passed, 1 failed

第三条失败了。你不需要去读代码找 bug——把测试结果发给 AI：

“超重附加费用例失败，期望 45.0 实际 38.0，请修复。”

AI 修复后重新跑测试，直到全部通过。

整个过程中你没有读过一行实现代码。

“但我还是想看看代码”

可以。但要分清两件事：

- 验证是必须的——每次都跑，自动化，不依赖人
- Review是可选的——你想看就看，但不作为质量门禁

验证是安全网，Review 是额外的信心。安全网不能省，额外的信心可以省。

验证的四个层次

从简单到完整，按需选择：

层次一：手动对比（L1）

最简单。你手动跑一下函数，眼睛对比输出和契约里的期望值。

适合：一次性脚本、小工具、探索性开发。

层次二：自动化测试（L2 最小启动）

把契约的验收条件翻译成测试用例，CI 自动跑。

// 从契约自动生成，或手动写

```
test('同城订单', () => {  
  expect(calculateShipping({ weight: 2, distance: 10, type: 'same_city' }))  
    .toEqual({ fee: 8.0 });  
});
```

适合：大多数项目。这是性价比最高的层次。

层次三：属性验证（L2 完整）

不只检查具体值，还检查输出的属性：

// 属性：运费永远 > 0（除非输入非法）

```
test('运费非负', () => {  
  for (const input of randomInputs(100)) {  
    const result = calculateShipping(input);  
    if (!result.error) {  
      expect(result.fee).toBeGreaterThan(0);  
    }  
  }  
});
```

适合：核心业务逻辑、金融计算、安全相关功能。

层次四：对抗验证（L3）

主动尝试破坏——用异常输入、边界值、恶意输入攻击函数：

```
test('SQL 注入不影响计算', () => {  
  const result = calculateShipping({  
    weight: 2, distance: 10, type: "'; DROP TABLE orders; --"  
  });  
  expect(result.error).toBeDefined();  
});
```

适合：面向用户的接口、安全敏感的功能。

一个常见的反驳：“AI 写的测试也不可信”

对。所以关键区分是：

- 验收条件（输入和期望输出的映射）必须由人定义——这是契约
- 测试代码（把验收条件翻译成可执行的测试）可以让 AI 写

人定义“什么算对”，AI 实现“怎么检查”。人是法官，AI 是书记员。法官定标准，书记员执行检查。

如果你连验收条件都让 AI 写，那就是让被告当法官——AI 定义“什么算对”，然后 AI 证明自己是正确的。这当然不可信。

更深一层的问题：“翻译忠实度”

即使人定义了验收条件，AI 在把验收条件翻译成测试代码时，也可能引入偏差。比如你写“登录后应跳转到首页”的测试可能只检查了 HTTP 200，没检查页面内容。这叫“翻译忠实度”问题——AI 忠实地执行了它对你意思的理解。

ODD 不声称消除这个风险，但提供了让风险可度量的手段：

- 变异测试：如果测试只检查了 HTTP 200，变异测试会生成一个“返回 200 但页面内容为空”的变异体。这个
- 契约对抗协议（CAP）：在契约层面消除歧义——如果契约写的是“跳转到首页”，CAP 会追问“首页的判定标准是什么？URL？页面标题？特定元素？”，在翻译发生之前就减少歧义空间。

关键区别：在传统 Code Review 中，遗漏是不可见的（你不知道你漏看了什么）。在 ODD 中，翻译忠实度的缺口是可度量的（变异存活率告诉你测试有多少盲区）。风险没有消失，但从“不可观测”变成

变异测试的保障边界

变异测试验证的是“测试与代码的一致性”——测试能否发现代码中的微小改动。但它不能验证“测试与业务需求”——如果契约本身写错了（比如“空用户名应该返回 success”），测试会忠实地验证这个错误的契约，变异测试也

这是 ODD 的设计边界：ODD 保障的是“产出物符合契约”，不是“契约符合真实世界”。后者的正确性最终锚定在——契约的外部锚点是人类业务知识，CAP 对抗可以发现契约的歧义和漏洞，但不能发现“人类对业务的理解本身就

总结

“契约即测试”不是循环定义。有人会问：契约中的 examples 自动成为测试用例，但 examples 本身是契约的一部分——这不是“法律由法律自身验证”吗？

不是。验证链条是单向的，有外部锚点：

人类业务知识（外部锚点）→ 契约（可执行规范）→ 测试（契约的机器执行）→ 产出物

契约的内容不是自我生成的——它来自人类对业务的理解。“空用户名应该报错”这个知识来自外部世界（业务规则）。examples 由 AI 生成草稿，人类签署就是确认“这些 examples 确实反映了我的业务需求”——签署行为本身就是外部锚点的注入。

旧方式	ODD 方式
AI 写代码 → 人 Review → 提交 质量取决于审查者的水平 产出越多，审查越跟不上 Review 通过 ≠ 没有 bug	人写契约 → AI 写代码 → 机器验证 → 提交 质量取决于契约的完整度 产出越多，验证速度不变 验证通过 = 满足所有验收条件

下一步

- 还没写过契约？→ C09. 工程_契约_产出物_证据.md
- 想从头开始？→ A01. 了解_核心痛点与解法.md
- 想动手练一个完整例子？→ B05. 入门_个人闭环体验.md

状态机：一个任务从生到死

一个痛苦的场景

你有 15 个 AI 任务同时在跑，哪些完成了？哪些卡住了？哪些需要你介入？

你打开项目管理工具，发现状态只有“进行中”和“已完成”。15 个任务里有 8 个“进行中”，但你不知道它们。有一个任务其实已经写完代码了，正在等你审查——但它显示的还是“进行中”。另一个任务测试失败了三次，也三个完全不同的处境，同一个标签。你只能挨个点进去看，靠记忆判断优先级。任务一多，脑子就不够用了。状态机解决的就是这个问题——给任务的每一步起一个明确的名字，让你一眼看出“它走到哪了”。

核心概念

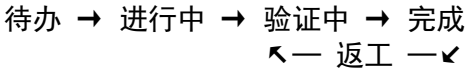
状态机由两样东西组成：

- 状态（State）：任务当前在哪一步。比如“待办”“进行中”“验证中”“完成”。
- 门禁（Gate）：状态之间的检查点。只有通过检查，任务才能往下走。没通过，就打回去。

门禁的意义在于：不是你说“做完了”就算完了，得拿证据证明。

L1 核心：4 个状态，1 道门禁

如果你只想要最简单的版本，4 个状态就够了：



对应的英文状态：

状态	英文	含义
待办	pending	任务已创建，还没人动
进行中	in_progress	有人（或 AI）正在干活
验证中	review	干完了，正在检查结果
完成	done	检查通过，收工

只有一道门禁：验证中 → 完成。

这道门禁的规则很简单：自动测试全部通过，就放行；有失败的，打回“进行中”，附上失败原因。

```
task:
  id: string
  state: pending | in_progress | review | done
  evidence:
    - type: test_report
      result: pass | fail
      summary: string
```

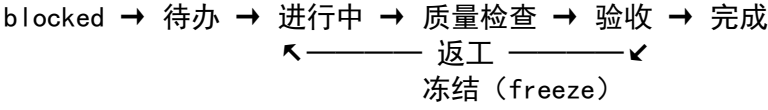
这就是 L1。一个人写小工具，这四个状态完全够用。

最小启动：7+ 状态，2 道门禁

本节是 L1/L2 入门简化版，完整状态机见下文。

当你开始管多个任务、或者有团队协作时，4 个状态不够了。你需要区分“被阻塞”和“待办”，也需要在门禁判

7+ 状态（与 ODD.03 L2 一致）：



新增的状态：

状态	英文	含义
被阻塞	blocked	被其他任务阻塞，等待依赖完成
冻结	freeze	门禁三值判定中的“不确定”，暂停等待人工裁决

sealed（封存）是 L3 概念——通过全部门禁后直接进入封存，替代 done 成为终态。详见 302。

2 道门禁：

门禁	位置	检查内容	谁来检查
质量检查	进行中 → 质量检查	测试通过 + 覆盖率达标 + 静态分析无严重项	自动
验收	质量检查 → 验收	产出物符合契约的验收条件	人工

门禁结果不只是“过”或“不过”，而是三值：PASS（放行）、FAIL（打回返工）、FREEZE（判不准，暂停等人工）、CONFLICT（不是代码的问题，是契约的问题）。

返工规则：质量检查或验收未通过，任务回到“进行中”，附带失败原因。返工次数会被记录——连续返工 3 次以上，说明任务可能比预想的复杂，应该考虑升级任务等级。

```
task:
  id: string
  state: blocked | pending | in_progress | quality_check | acceptance | done | rework | freeze
  rework_count: int
  evidence:
    - evidence_type: test_report | coverage_report | lint_report | review_record
      gate: quality_check | acceptance
      result: pass | fail | freeze | conflict
      summary: string
      failure_context:
        evidence_ref: string    # 指向具体证据 ID
        summary: string
```

一个完整的任务走查

下面用一个真实场景，走一遍任务从创建到完成的全过程。

任务：给用户注册接口加上邮箱格式验证。

第 1 步：创建任务 → 待办（pending）

```
task:
  id: task-042
  state: pending
  title: "用户注册接口增加邮箱格式验证"
  contract_id: contract-user-register
  rework_count: 0
```

任务创建了，还没人领。

第 2 步：AI 领取任务 → 进行中（in_progress）

AI 开始写代码。状态从 **pending** 变为 **in_progress**。这一步没有门禁，直接迁移。

第 3 步: AI 提交代码 → 质量检查 (quality_check)

AI 写完了, 提交产出物。系统自动运行第一道门禁:

- 单元测试: 12/12 通过
- 覆盖率: 87% (阈值 80%)
- 静态分析: 0 个严重项

门禁结果: PASS。任务通过质量检查门禁, 状态迁移到 **acceptance** (等待人工验收), 同时生成证据:

evidence:

```
- evidence_type: test_report
  gate: quality_check
  result: pass
  summary: "12/12 tests passed, coverage 87%"
```

第 4 步: 人工验收 → 验收 (acceptance)

你打开产出物, 对照契约的验收条件逐条检查:

- 正常邮箱能注册
- 格式错误的邮箱返回友好提示
- 空邮箱返回 “邮箱不能为空”

门禁结果: PASS。你确认通过, 生成审查记录:

evidence:

```
- evidence_type: review_record
  gate: acceptance
  result: pass
  reviewer_id: "you"
  summary: "验收条件全部满足"
```

第 5 步: 标记完成 → 完成 (done)

验收通过, 任务进入 **done**。收工。

在 L3 流程中, 任务不会停在 **done**, 而是直接进入 **sealed** (封存), 证据链冻结、产出物锁定。详见下文。
任务走完了。从 **pending** 到 **done**, 每一步都有明确的状态名, 每一次迁移都有门禁把关, 每一次通过都有证据记

如果中间失败了呢?

假设第 3 步质量检查没过——有 2 个测试失败了:

evidence:

```
- evidence_type: test_report
  gate: quality_check
  result: fail
  summary: "10/12 tests passed, 2 failed"
  failure_context:
```

```
evidence_ref: "evidence-003"
summary: "test_empty_email 和 test_unicode_email 失败"
```

门禁结果：FAIL。任务回到 `in_progress`，`rework_count` 从 0 变成 1。AI 拿到失败原因，修复后重新提交，再走。如果门禁判不准呢？比如静态分析报了一个可疑项，但不确定是不是误报——这时门禁结果是 `FREEZE`，任务暂停，等人工来裁决。人工确认是误报就放行（生成 `override` 证据），确认是真问题就打回返工。超时 48 小时未处理，自动升级为 `FAIL`。

常见错误

错误	后果	修正
只用“进行中”和“已完成”两个状态 门禁没有绑定证据 返工不记录次数 跳过门禁直接标记完成 门禁判不准时硬判 <code>pass</code> 或 <code>fail</code>	无法区分“在写”“在测”“在等审查” 说“通过了”但拿不出证明 不知道一个任务被打回了多少次 未经验证的产出物流向下游 误放或误杀	至少用 4 个状态 每次通过门禁必须生成证据 用 <code>rework_count</code> 追踪 状态迁移必须经过门禁 用 <code>FREEZE</code> 暂停，等人工裁决

下一步

- 状态机里的“验收条件”怎么写？→ C09. 工程_契约_产出物_证据.md
- 完整状态机（L3 封存、动态门禁链、Challenge 机制）→ 见下文
- 证据怎么打包？封存怎么操作？→ C09. 工程_契约_产出物_证据.md

layer: L3 type: reference prerequisites: C09. 工程_契约_产出物_证据 source: ODD. 03状态-状态机与门禁 last_updated: 2026-02-16

状态机与门禁完整参考

前置知识：阅读本文前，请先完成 C09. 工程_契约_产出物_证据.md。

概述

ODD 中每一次状态迁移都必须有理由、有证据、有记录。没有通过门禁的产出物，不允许流向下游。

核心定义：

- 状态（State）：产出物在生命周期中的位置标识。
- 门禁（Gate）：状态之间的检查点，决定是否允许迁移。
- 证据绑定（Evidence Binding）：每次通过门禁 MUST 关联至少一条证据记录。
- 返工（Rework）：门禁未通过时的回退路径，MUST 携带 `failure_context`。

状态迁移 = 门禁检查 + 证据绑定 + 审计记录，三者缺一不可。

L1 基础

状态图（5 状态）

pending → in_progress → review → done
 ↖—— rework ——↗

门禁

只有一道——review。

- 通过条件：自动测试全部通过。
- 证据：测试报告（通过/失败 + 覆盖率）。
- 失败处理：回到 in_progress，附带失败原因摘要。

最小数据结构

```
task:
  id: string
  state: pending | in_progress | review | done | rework
  evidence:
    - type: test_report
      result: pass | fail
      summary: string
```

最小启动子集（L2）

状态图（7+ 状态）

在 L1 基础上增加 blocked 和 quality_check:

blocked → pending → in_progress → quality_check → acceptance → done
 ↖—————— rework —————↗

两道门禁

门禁 1: quality_check（自动）

- 通过条件：测试通过 + 覆盖率 ≥ 阈值 + 静态分析无严重项。
- 证据：测试报告 + 覆盖率报告 + lint 报告。

门禁 2: acceptance（人工）

- 通过条件：审查者确认产出物符合契约验收条件。
- 证据：审查记录（审查者 ID + 结论 + 备注）。

三值模型（PASS / FREEZE / FAIL）

门禁判定不总是非黑即白。当门禁无法确定结果时，强制二选一会造成损失。

结果	含义	后续动作
PASS	所有检查通过	允许迁移到下一状态
FAIL	存在明确的不合格项	回退到 rework
FREEZE	存在不确定项（置信度不足、NEN 条目待审、异常但无法判定严重程度）	进入待审队列，流水线暂停

FREEZE 规则：

- 产出物不可流向下游，也不回退。
- FREEZE 超时（建议 48h）未处理，自动升级为 FAIL。
- 人工解除 FREEZE 时 MUST 生成 **override** 证据。

CONFLICT 矛盾检测

当契约的两个验收条件在逻辑上互斥时，修代码没用，需要修契约。CONFLICT 不是产出物的问题，是契约的问题。

门禁检测到验收条件互斥时，报 CONFLICT 而非普通 FAIL。CONFLICT 自动触发三步处理：

1. 产出物进入 FREEZE。
2. 生成矛盾报告证据（`conflict_report`），包含冲突条件对和冲突原因。
3. 通知契约负责人修订契约。

典型矛盾场景：

- “响应 < 100ms” 与 “全量加密” 在当前硬件下不可兼得。
- “零停机部署” 与 “强一致性” 在分布式环境下冲突。

矛盾报告的 `recommended_action` 字段 SHOULD 给出建议（修订契约 / 放宽约束 / 升级硬件）。

Override 机制（人工覆盖与审计）

当人工覆盖门禁结果时（将 FAIL 改为 PASS、解除 FREEZE、降级 `task_level`），MUST 生成一条 **override** 证据：

evidence:

```
type: override
original_result: fail | freeze | conflict
overridden_to: pass | rework
operator_id: string
reason: string # MUST 非空
timestamp: datetime
# 退出三件套
expiry: datetime?
metric_boundary:
  metric_name: string?
  threshold: float?
  operator: "<" | ">" | "<=" | ">="?
review_interval_days: int?
```


退出三件套

Override 不是永久的。三个退出条件中任一满足即退出覆盖状态：

条件	机制	触发时
expiry	日落条款：过期后自动恢复原始门禁结果	生成 <code>override_expired</code> 审计记录
metric_boundary	指标边界：特定指标恢复正常时自动退出	生成 <code>override_metric_exit</code> 审计记录
review_interval_days	审查间隔：到期通知操作者确认是否继续	操作者可续期（附理由）或确认退出

规则：

- 续期超过 3 次时，SHOULD 升级给上级审查。
- 三个字段均未填写时，override 默认 90 天后过期（硬上限，防止“临时措施变永久”）。

返工增强

- rework 记录包含：触发门禁、失败证据引用、返工次数。
- 返工次数 ≥ 3 时，SHOULD 自动升级 task_level。
- failure_context MUST 引用证据对象（evidence_ref），不得内联大段日志。

双轨流水线（快轨 / 慢轨）

不同风险等级的任务不应走同一条流水线。

轨道	适用	特征
快轨（Fast Track）	L0-L1 任务	高自动化、低人工干预，FREEZE 阈值较高
慢轨（Slow Track）	L2-L3 任务	强制人工审查节点，FREEZE 阈值较低，NEN 条目必须人工判定

- L4 任务始终走慢轨，且 MUST 经过 human_review 门禁。
- 任务等级在契约定义阶段确定，流水线自动路由。
- 团队可自定义快轨/慢轨的门禁配置，但不可将 L3/L4 任务路由到快轨。

Challenge 机制（执行者质疑权）

执行者（人或 AI）发现契约本身有问题时，可以在 `in_progress` 或 `quality_check` 阶段对契约发起质疑。challenge 不是返工（rework 是“产出物不合格”），而是“契约本身有问题”。

challenge 触发三步处理：

1. 任务暂停，进入 `challenged` 状态（不回退、不前进）。
2. 生成 `challenge` 证据，包含质疑者、理由、建议修改方向。
3. 通知契约负责人审查。

evidence:

```
type: challenge
challenger_id: string
reason: string           # MUST 非空
suggested_change: string?
timestamp: datetime
```

```
resolution: accepted | rejected
resolution_reason: string?
resolved_by: string?
resolved_at: datetime?
```

处理规则：

- 契约负责人可以接受（修订契约，任务回到 `in_progress`）或驳回（附理由，任务恢复原状态继续执行）。
- 驳回后执行者 **MUST** 继续执行，但 `challenge` 记录保留在证据链中，不可删除。
- `challenge` 不计入返工次数。
- 频繁 `challenge` 是方法论反馈环的信号——说明契约定义流程可能需要改进。

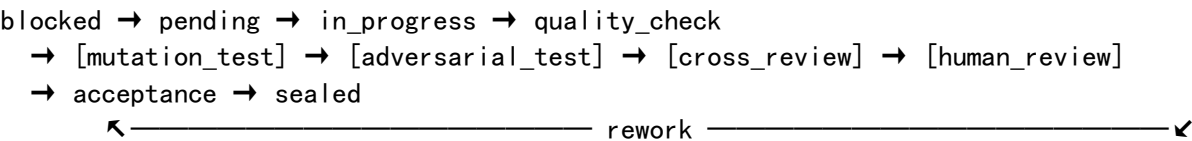
L2 数据结构

```
task:
  # ...L1 字段...
  state: blocked | pending | in_progress | quality_check | acceptance | done | rework | freeze
  task_level: L1 | L2
  track: fast | slow
  rework_count: int
  evidence:
    - evidence_type: test_report | coverage_report | lint_report | review_record | output_verification
      gate: quality_check | acceptance
      result: pass | fail | freeze | conflict
      reviewer_id: string?
      override_reason: string?
      failure_context:
        evidence_ref: string
        summary: string
```

L3 核心

状态图（封存替代 `done`）

在 L2 基础上增加风险驱动的动态门禁链，`sealed` 替代 `done` 成为终态：



方括号 = 按 `task_level` 动态插入：

task_level	门禁链
L1	跳过所有扩展门禁
L2	插入 <code>mutation_test</code>
L3	插入 <code>mutation_test</code> + <code>adversarial_test</code>
L4	全部插入，含 <code>human_review</code>

四道门禁详细规则

门禁 1: mutation_test

- 通过条件: `mutation_score` \geq 阈值 (建议 80%)。
- 证据: 变异测试报告 (存活变异体列表 + 总分)。

门禁 2: adversarial_test

- 通过条件: 无高危漏洞存活。
- 证据: 漏洞列表 + 复现实验记录。

门禁 3: cross_review

- 通过条件: 多方审查者共识 ($\geq 2/3$ 通过)。
- 证据: 各方意见 + 裁决依据。
- 报告结构:

```
cross_review_report:
  reviewers: [workshop_id]
  opinions:
    - reviewer: workshop_id
      verdict: pass | fail | concern
      comments: [string]
  consensus: bool
  escalated: bool
  escalation_reason: string?
```

门禁 4: human_review

- 触发条件: `cross_review` 产生分歧, 或 `task_level` = L4。
- 证据: 人类裁决记录 + 理由。

封存 (Sealed)

通过全部门禁后直接进入 **sealed**, 不经过 **done**。证据链冻结。sealed MUST 生成 `seal_hash`, 绑定全部 `evidence_ref`。

task_level 分级算法

AI 自动分级 + 人类可覆盖。四维度评分:

维度	L1 (1分)	L2 (2分)	L3 (3分)	L4 (4分)
预估代码行数	< 30	30-100	100-300	> 300
功能复杂度	单表 CRUD	多表关联	复杂算法	分布式/并发
安全敏感度	无	低 (日志)	中 (用户数据)	高 (认证/支付)
外部依赖数	0	1-2	3-5	> 5

计算规则: **最终级别** = **MAX(各维度级别)** —— 短板原则。

人类覆盖规则: 默认只能升级; 降级 MUST 由人类输入理由并记录审计日志。

L3 数据结构

task:

...L2 字段...

state: ...L2状态 (sealed 替代 done) ... | mutation_test | adversarial_test | cross_review | human_review

task_level: L1 | L2 | L3 | L4

track: fast | slow # L3/L4 强制 slow

gate_chain: [string]

seal:

seal_hash: string

sealed_at: datetime

evidence_refs: [string]

decision_narrative: string?

evidence:

- evidence_type: ...L2类型... | mutation_report | adversarial_report | cross_review_record | human_review

gate: string

result: pass | fail | freeze | conflict

mutation_score: float?

vulnerability_count: int?

reviewers: [string]?

consensus: bool?

override_reason: string?

conflict_pair: [string]?

L3 扩展

验证策略分类

artifact_type 决定验证策略，验证策略决定跳过哪些门禁：

策略	quality_check	acceptance	mutation_test	适用
automated	自动单元测试	自动集成测试	执行	代码类
executed	执行验证	效果验证	跳过	数据库/行为类
reviewed	格式检查	人工审核	跳过	文档类
single	跳过	合并验证	跳过	简单配置
none	跳过	跳过	跳过	注释/日志

外部显示标签

内部状态统一，外部按产出物类别显示专业术语：

内部状态	代码类	数据库类	行为类	文档类	配置类
in_progress	开发中	开发中	开发中	编写中	配置中
quality_check	单元测试中	DDL执行中	触发验证中	格式检查中	语法验证中
acceptance	集成测试中	数据验证中	效果验证中	内容审核中	生效验证中
mutation_test	变异测试中	变异测试中	变异测试中	跳过	跳过

返工阈值协调

返工次数同时触发三个独立机制，各有各的阈值：

机制	阈值	效果
task_level 升级	返工 ≥ 3 次	增加门禁链
执行者升级	按诊断结果	改变谁来做
分级预警	2 次黄灯、4 次橙灯、 ≥ 6 次红灯	改变人类是否介入

三者独立运作、互不阻塞。

适配建议

- 可以按模块混用等级：核心模块 L3，工具脚本 L1。
- 门禁阈值（覆盖率、mutation_score 等）由项目自行设定，ODD 不规定具体数值。
- task_level 决定需要经过哪些门禁：人工覆盖 MUST 只能升级，不能降级。

完整 YAML 示例（L3 任务生命周期）

```
# === 任务创建 ===
task:
  id: "TSK-AUTH-001"
  contract_id: "CTR-AUTH-001"
  title: "实现 JWT 认证中间件"
  state: pending
  task_level: L3
  track: slow
  gate_chain: [quality_check, mutation_test, adversarial_test, cross_review]
  rework_count: 0

# === quality_check 门禁通过 ===
evidence:
  id: "EVD-QC-001"
  evidence_type: test_report
  gate: quality_check
  result: pass
  summary: "单元测试 42/42 通过，覆盖率 91%"

# === mutation_test 门禁通过 ===
evidence:
  id: "EVD-MT-001"
  evidence_type: mutation_report
  gate: mutation_test
  result: pass
  summary: "mutation_score = 85%"
  mutation_score: 0.85
```

```

# === adversarial_test 门禁通过 ===
evidence:
  id: "EVD-AT-001"
  evidence_type: adversarial_report
  gate: adversarial_test
  result: pass
  summary: "无高危漏洞, 2 个低危已记录"
  vulnerability_count: 0

# === cross_review 门禁通过 ===
evidence:
  id: "EVD-CR-001"
  evidence_type: cross_review_record
  gate: cross_review
  result: pass
  reviewers: ["WS-BACKEND-01", "WS-SECURITY-01", "WS-BACKEND-02"]
  consensus: true

# === 封存 ===
task:
  state: sealed
  seal:
    seal_hash: "sha256:f9e8d7c6..."
    sealed_at: "2026-02-16T16:00:00Z"
    evidence_refs: ["EVD-QC-001", "EVD-MT-001", "EVD-AT-001", "EVD-CR-001"]
    decision_narrative: "JWT 中间件通过全部 L3 门禁, 含变异测试和对抗测试"

# === Override 示例 (FREEZE 后人工放行) ===
evidence:
  id: "EVD-OVR-001"
  evidence_type: override
  original_result: freeze
  overridden_to: pass
  operator_id: "tech-lead-zhang"
  reason: "误报: 静态分析将合法的 crypto 调用标记为可疑, 已确认安全"
  timestamp: "2026-02-16T15:30:00Z"
  expiry: "2026-03-16T15:30:00Z"
  review_interval_days: 30

# === Challenge 示例 ===
evidence:
  id: "EVD-CHG-001"
  evidence_type: challenge
  challenger_id: "WS-BACKEND-01"
  reason: "契约要求同时支持 HS256 和 RS256, 但未指定密钥管理策略, 两种算法的密钥轮换机制互斥"
  suggested_change: "明确选择一种算法, 或拆分为两个独立契约"
  timestamp: "2026-02-16T10:00:00Z"

```

```
resolution: accepted
resolution_reason: "拆分为 CTR-AUTH-001a (HS256) 和 CTR-AUTH-001b (RS256)"
resolved_by: "architect-li"
resolved_at: "2026-02-16T11:00:00Z"
```

理论来源

1. 悖论熔断公理 (ASTO)：当基元与禁元冲突时，系统必须立即执行安全协议而非强行裁决。本文中 CONFLICT 矛盾检测机制来源于此。
 2. 自由论 (P07) 与批判篇 (P09)：效率没有拒绝权就是通往奴役的高速公路。本文中 challenge 机制（执行者质疑权）来源于此。
 3. 例外篇 (P08)：退出三件套 (expiry、metric_boundary、review_interval) 的设计来源于此，防止“临时
-
-

```
layer: L3 type: reference prerequisites: [C09.工程_契约_产出物_证据,
D15.深度_环境与管道隔离] source: ODD.12执行-执行模型 last_updated: 2026-02-16
```

执行模型

前置知识：阅读本文前，请先完成 上文“状态机与门禁”部分。管道相关概念参见 D15. 深度_环境与管道隔离.md。

与状态机的关系：任务状态迁移由执行模型驱动。**三值门禁 (PASS/FAIL/FREEZE)**由上文“状态机与门禁”定义。

- expected/unexpected错误 → 任务状态不变，等待修复
 - fatal错误 → 任务直接进入FAIL状态，回退到rework
-

概述

ODD 不只回答“做什么”和“做好了没”，还必须回答：出错怎么办？副作用怎么控？状态怎么流转？多个任务并行时，执行模型把这些问题前置到契约和管道层——不是等代码写完了再想“异常处理怎么做”，而是在契约定义阶段就

L1 基础

L1 的执行模型是最小化的：只区分最基本的错误类型，副作用用文字描述，并发默认串行。

错误：可恢复 vs 不可恢复

L1 只需要回答一个问题：这个错误发生后，调用方能不能重试？

```
contract:
  id: user-register
  title: "用户注册"
```

```
errors:
  - code: INVALID_INPUT
    recoverable: false          # 输入错了，重试也没用
  - code: DB_TIMEOUT
    recoverable: true          # 超时了，可以重试
```

`recoverable: true` 意味着调用方可以安全重试；`recoverable: false` 意味着必须修正输入后才能重新调用。

副作用：文字描述

L1 的副作用声明只需要告诉读者“这个任务会对外部产生什么影响”：

```
effects:
  - type: db_write
    description: "插入用户记录到 users 表"
  - type: event_emission
    description: "发布 user_created 事件"
```

没有自动化验证，靠人工确认副作用是否真的发生了。

并发：默认串行

L1 不处理并发问题。所有任务默认串行执行，不考虑并行冲突。

适用场景：个人项目、脚本工具、一次性任务。

最小启动子集（L2）

L2 引入结构化的错误分类、可验证的副作用声明、状态容器和并发控制。

错误处理模型

L2 的错误不只区分“能不能重试”，还要回答“谁的责任”和“怎么传播”。

三类错误：

类别	含义	示例	调用方行为
client_error	调用方的问题	参数格式错误、缺少必填字段	修正输入后重试
server_error	服务方的问题	内部逻辑错误、未处理的异常	报告给服务方
external_error	第三方的问题	数据库超时、外部 API 不可用	按重试策略重试

```
errors:
  - code: INVALID_EMAIL
    category: client_error
    recoverable: false
    message: "邮箱格式不正确"

  - code: DB_TIMEOUT
    category: external_error
```



```

recoverable: true
retry_strategy: exponential_backoff
max_retries: 3
retry_delay_base: 1s

- code: INTERNAL_ERROR
  category: server_error
  recoverable: false
  message: "内部处理错误"

```

错误传播规则：

上游错误必须被下游显式声明或转换，不允许静默吞掉。

```

error_propagation:
  mode: explicit # explicit | transform
  mapping:
    DB_TIMEOUT: SERVICE_TEMPORARY_UNAVAILABLE
    REDIS_ERROR: SERVICE_TEMPORARY_UNAVAILABLE
    # 未映射的错误 → 自动包装为 INTERNAL_ERROR

```

`mode: explicit` 要求每个可能的上游错误都有明确的映射；`mode: transform` 允许用通配规则批量转换。

副作用声明与验证

L2 的副作用不只是文字描述，而是可验证的结构化声明：

```

# 声明：这个任务会产生哪些副作用
effects:
  - type: db_write
    target: users
    operation: insert
  - type: event_emit
    event: user_created
  - type: log_write
    target: audit_log

# 验证：执行后检查副作用是否真的发生了
expected_effects:
  - type: db_write
    target: users
    data_match:
      email: "test@example.com"
      status: "active"
  - type: event_emit
    event: user_created
    data_match:
      user_id: "非空字符串"

```

副作用类型（常用）：

类型	说明
db_read / db_write	数据库读写
data_insert / data_update / data_delete	数据增删改
event_emit	事件发布
http_request / external_call	外部调用
file_read / file_write / file_create	文件操作
cache_read / cache_write	缓存操作
log_write	日志写入

任何外部调用 MUST 标注重试/超时/幂等策略：

```
effects:
  - type: external_call
    target: "payment-gateway"
    timeout: 10s
    retry_strategy: exponential_backoff
    max_retries: 2
    idempotent: true                # 重试安全
```

状态容器

当多个任务需要共享状态时，用状态容器显式声明读写关系：

```
state_container:
  name: Session
  scope: session                  # session | request | global
  schema:
    user_id: uuid
    token: string
    permissions: [string]

contract:
  id: get-profile
  reads: [Session.user_id]       # 声明读取哪些状态
  output:
    profile: object
```

状态容器的好处：谁读了什么、谁写了什么，一目了然。不会出现“某个任务偷偷改了全局状态导致另一个任务出

并发与幂等

L2 要求显式声明并发策略和幂等性：

```
concurrency:
  mode: serialized                # serialized | parallel | locked
  lock_key: email                 # 锁的粒度
  lock_timeout: 5s                # 锁超时

idempotency:
```

```
enabled: true
key: email # 幂等键
behavior: return_existing # return_existing | reject
ttl: 24h # 幂等窗口
```

三种并发模式：

- **serialized**: 同一 lock_key 的请求排队执行
- **parallel**: 允许并行，由应用层处理冲突
- **locked**: 获取锁后执行，锁超时自动释放

幂等行为：

- **return_existing**: 重复请求返回上次的结果（适合创建操作）
- **reject**: 重复请求返回错误（适合一次性操作）

完整 L2

在最小启动基础上，L2 完整模式增加状态快照和恢复：

```
state_management:
  snapshot:
    enabled: true
    trigger: before_effect # 在副作用执行前快照
    storage: local # local | remote
  recovery:
    strategy: restore_snapshot # restore_snapshot | retry_from_checkpoint
    max_attempts: 3
```

快照规则：

- 每次执行有副作用的步骤前，自动保存当前状态快照
- 失败时可从最近的快照恢复，而不是从头开始
- 快照 SHOULD 有 TTL，过期自动清理

L3 核心

L3 在 L2 基础上增加：事务边界管理、补偿策略（Saga 模式）、混合管道模式。

错误处理模型（L3 扩展）

L3 将错误细分为三级：

级别	名称	含义	处理方式
预期错误	expected	业务逻辑中预见到的错误	按契约声明的错误码处理
意外错误	unexpected	未预见但可恢复的错误	记录日志 + 通用错误响应 + 告警
致命错误	fatal	系统级故障，不可恢复	立即停止 + 触发补偿 + 人工介入

```
error_classification:
  expected:
```

```

- code: INVALID_EMAIL
- code: USER_EXISTS
unexpected:
  handling: log_and_alert
  fallback_response: { error: "SERVICE_ERROR", message: "请稍后重试" }
fatal:
  - code: DB_CONNECTION_LOST
  - code: ENCRYPTION_KEY_MISSING
  handling: halt_and_compensate
  alert: immediate
  requires_human: true

```

致命错误 MUST 触发补偿流程并通知人工介入。不允许自动重试致命错误。

事务边界

DB 事务只包含 DB 操作，外部调用必须在事务外：

```

transaction:
  isolation: serializable
  timeout: 30s
  steps:
    - contract: ValidateInput
      in_transaction: false          # 验证不需要事务
    - contract: CreateOrder
      in_transaction: true          # DB 操作在事务内
    - contract: ProcessPayment
      in_transaction: false        # 外部调用在事务外
    - contract: SendConfirmation
      in_transaction: false        # 外部调用在事务外
  on_failure:
    strategy: compensate           # rollback_all | compensate
    compensations:
      CreateOrder: CancelOrder
      ProcessPayment: RefundPayment

```

事务规则：

- DB 事务 MUST 尽可能短——只包含必要的 DB 操作
- 外部调用（HTTP、消息队列、文件系统）MUST 在事务外
- 长事务（> 30s）MUST 拆分为多个短事务 + 补偿

事务补偿（Saga 模式）

跨步骤的失败不能简单 rollback——外部调用无法回滚。Saga 模式为每个步骤定义补偿操作，失败时反向执行：

```

saga:
  name: "订单创建流程"
  steps:
    - name: create_order
      contract: CreateOrder

```

```

    compensation: CancelOrder

- name: reserve_inventory
  contract: ReserveInventory
  compensation: ReleaseInventory

- name: process_payment
  contract: ProcessPayment
  compensation: RefundPayment

- name: send_confirmation
  contract: SendConfirmation
  compensation: null # 通知类无需补偿

failure_handling:
  mode: backward_recovery # 反向补偿
  max_compensation_retries: 3
  on_compensation_failure: alert_human

```

Saga 执行流程:



补偿规则:

- 每个有副作用的步骤 MUST 有对应的补偿操作
- 补偿操作 MUST 是幂等的（可能被重试多次）
- 补偿失败 MUST 告警并等待人工介入
- 通知类步骤（邮件、消息）可以没有补偿（标记 `compensation: null`）

混合管道模式

L3 支持三种管道编排方式，按场景选择:

模式	特点	适用场景
编排 (Orchestration)	中心协调器控制流程	步骤间有严格顺序依赖
事件驱动 (Choreography)	各步骤通过事件松耦合	步骤间独立性高
Saga	每步有补偿，失败反向执行	跨服务的分布式事务

```

pipeline:
  mode: orchestration # orchestration | choreography | saga
  steps:
    - contract: ValidateInput
      next: CreateOrder
    - contract: CreateOrder

```

```

    next: ProcessPayment
    on_failure: compensate
- contract: ProcessPayment
  next: SendConfirmation
  on_failure: compensate
- contract: SendConfirmation
  next: complete

```

执行上下文

L3 的每次执行都携带完整的上下文对象，用于追踪和审计：

```

execution_context:
  request_id: "REQ-20260216-001"      # 全局唯一请求 ID
  timestamp: "2026-02-16T14:00:00Z"
  errors: []                          # 执行过程中收集的错误
  effects: []                         # 执行过程中产生的副作用
  transaction_id: "TXN-001"          # 事务 ID（如有）
  saga_id: "SAGA-001"               # Saga ID（如有）
  compensation_log: []               # 补偿记录

```

逆向熔断机制：来自执行层的反击

问题：在现有流程中，契约是自上而下的约束——人类 Arbitrator 确认后，Executor 去执行。但有时候，人类确

优化：赋予执行层 AI “上诉”的权利，形成双向闭环。

```

reverse_circuit_breaker:
  enabled: true
  appeal_trigger:
    max_consecutive_failures: 5      # 连续失败 N 次后触发
    failure_pattern: "验证阶段反复失败" # 而非生成阶段失败
  appeal_evidence:
    - "每次失败的错误日志"
    - "尝试的不同实现路径"
    - "与契约条款的矛盾点分析"
  appeal_escalation:
    - "返回给 Proposer 重新协商契约"
    - "标记该契约为 '需要复核'"
    - "严重时通知人类仲裁者"

```

工作流：

1. Executor 连续 N 次尝试生成产出物
2. 每次都在验证阶段失败（不是代码错误，而是契约逻辑矛盾）
3. Executor 收集“死亡证据”：哪些契约条款互相冲突
4. Executor 发起“契约不合理”的上诉请求
5. Proposer 收到上诉，分析后决定：
 - 修改契约条款

- 标记需要人类复核
- 确认是执行方问题而非契约问题

价值：

- 契约不应仅仅是自上而下的约束
- 执行过程中的阻力必须能反哺契约的修正
- 形成双向闭环，而非单向强制

示例场景：

契约要求：

- "高并发下必须保证强一致性"
- "响应时间必须 < 100ms"

执行层发现：

- 在当前数据规模下，这两个条件互斥
- 尝试了各种方案都无法同时满足
- 上诉：这两个约束矛盾，请澄清优先级

L3 扩展

L1/L2/L3 执行模型差异速查

维度	L1	L2	L3
错误分类	可恢复/不可恢复	三类（client/server/external）	三级（预期/意外/致命）
错误传播	无	显式映射	显式映射 + 致命错误特殊处理
副作用声明	文字描述	结构化 + 可验证	结构化 + 可验证
状态管理	无	状态容器 + 快照	状态容器 + 快照 + 恢复
并发控制	默认串行	三种模式 + 幂等	三种模式 + 幂等
事务	无	无	事务边界 + 隔离级别
补偿	无	无	Saga 模式 + 反向补偿
管道模式	无	无	编排/事件驱动/Saga
执行上下文	无	无	完整追踪对象

执行模型与其他机制的协同

执行模型不是孤立的，它与 ODD 的其他机制紧密配合：

- 与契约的关系：错误码、副作用、并发策略都在契约中声明，执行模型是契约的“运行时规格”
- 与状态机的关系：任务状态迁移（pending → in_progress → done/rework）由执行模型驱动，错误触发 rework，补偿触发回滚
- 与证据的关系：执行上下文中的 errors、effects、compensation_log 都是证据的来源
- 与封存的关系：只有执行上下文中无未处理错误、所有副作用已验证，才允许封存

执行模型的核心原则

- 错误不可隐身：不声明就等于不存在——未声明的错误在运行时出现，门禁视为致命错误

- 副作用必须可验证：否则无法证明“真的发生了”——声明了 `db_write` 但没有 `expected_effects`，门禁会警告
- 事务要短、补偿要清晰：长事务必然锁死系统，补偿不清晰必然导致数据不一致
- 并发要可控：锁与幂等是最低成本的保险——不声明并发策略的任务默认串行

完整 YAML 示例（L3）

=== 执行模型完整示例：订单创建流程 ===

```
contract:
  id: "CTR-ORDER-CREATE-001"
  title: "创建订单"
  maturity: formal
  scope_in: "验证输入、创建订单、扣减库存、处理支付、发送确认"
  scope_out: "不含物流调度"

# 错误声明
errors:
  - code: INVALID_ORDER
    category: client_error
    recoverable: false
    message: "订单参数无效"
  - code: INSUFFICIENT_INVENTORY
    category: server_error
    recoverable: false
    message: "库存不足"
  - code: PAYMENT_FAILED
    category: external_error
    recoverable: true
    retry_strategy: exponential_backoff
    max_retries: 3
  - code: PAYMENT_GATEWAY_DOWN
    category: external_error
    recoverable: false
    message: "支付网关不可用"

error_classification:
  expected: [INVALID_ORDER, INSUFFICIENT_INVENTORY, PAYMENT_FAILED]
  unexpected:
    handling: log_and_alert
    fallback_response: { error: "ORDER_ERROR", message: "订单处理失败，请稍后重试" }
  fatal:
    - code: DB_CONNECTION_LOST
      handling: halt_and_compensate
      requires_human: true
```



```

error_propagation:
  mode: explicit
  mapping:
    PAYMENT_FAILED: ORDER_PAYMENT_FAILED
    PAYMENT_GATEWAY_DOWN: ORDER_SERVICE_UNAVAILABLE

# 副作用声明
effects:
  - type: db_write
    target: orders
    operation: insert
  - type: db_write
    target: inventory
    operation: update
  - type: external_call
    target: payment-gateway
    timeout: 10s
    idempotent: true
  - type: event_emit
    event: order_created
  - type: event_emit
    event: payment_processed

expected_effects:
  - type: db_write
    target: orders
    data_match:
      status: "confirmed"
      total_amount: "> 0"
  - type: db_write
    target: inventory
    data_match:
      reserved_quantity: "> 0"

# 并发控制
concurrency:
  mode: locked
  lock_key: "user_id + product_id"
  lock_timeout: 10s

idempotency:
  enabled: true
  key: "order_request_id"
  behavior: return_existing
  ttl: 1h

# 状态容器
state_containers:

```

```

- name: OrderContext
  scope: request
  schema:
    order_id: string
    user_id: string
    items: [object]
    total_amount: decimal
    payment_status: string

# === Saga 定义 ===
saga:
  name: "订单创建 Saga"
  steps:
    - name: validate_input
      contract: ValidateOrderInput
      compensation: null

    - name: create_order
      contract: CreateOrderRecord
      compensation: CancelOrderRecord
      in_transaction: true

    - name: reserve_inventory
      contract: ReserveInventory
      compensation: ReleaseInventory
      in_transaction: true

    - name: process_payment
      contract: ProcessPayment
      compensation: RefundPayment
      in_transaction: false

    - name: send_confirmation
      contract: SendOrderConfirmation
      compensation: null

  failure_handling:
    mode: backward_recovery
    max_compensation_retries: 3
    on_compensation_failure: alert_human

# === 执行上下文 ===
execution_context:
  request_id: "REQ-20260216-ORDER-001"
  timestamp: "2026-02-16T14:00:00Z"
  saga_id: "SAGA-ORDER-001"
  steps_completed:
    - { step: validate_input, status: success, at: "2026-02-16T14:00:01Z" }

```

```

- { step: create_order, status: success, at: "2026-02-16T14:00:02Z" }
- { step: reserve_inventory, status: success, at: "2026-02-16T14:00:03Z" }
- { step: process_payment, status: success, at: "2026-02-16T14:00:05Z" }
- { step: send_confirmation, status: success, at: "2026-02-16T14:00:06Z" }
errors: []
effects:
- { type: db_write, target: orders, operation: insert, at: "2026-02-16T14:00:02Z" }
- { type: db_write, target: inventory, operation: update, at: "2026-02-16T14:00:03Z" }
- { type: external_call, target: payment-gateway, at: "2026-02-16T14:00:05Z" }
- { type: event_emit, event: order_created, at: "2026-02-16T14:00:06Z" }
compensation_log: []
transaction_id: "TXN-ORDER-001"

# === 证据 ===
evidence:
  id: "EVD-EXEC-001"
  evidence_type: execution_report
  gate: quality_check
  result: pass
  summary: "订单创建流程执行成功, 5/5 步骤完成, 0 错误, 4 副作用已验证"
  execution_context_ref: "REQ-20260216-ORDER-001"
  sha256: "sha256:a1b2c3d4..."

```

理论来源

1. 属性分层原理（ASTO 层次支撑定理）：错误分类中的“致命错误”对应硬属性——数据库连接丢失、加密密 MUST 立即停止而非重试。
 2. 结构病理学（ASTO A01/A02）：长事务是典型的“过度勤勉致死”模式——试图在一个事务中完成所有操作，模式的“短事务 + 补偿”是对这一病理的工程解药。
 3. 可执行性梯度（NTE 可执行性梯度）：副作用声明中 `expected_effects` 的 `data_match` 是将副作用验证从“需解释”（NEN）提升到“刚性可执行”（EN）的关键——把“数据库里应该有一条记录”变成“必须有”。
 4. 意图五态模型（ASTO 五态模型）：执行上下文的 `steps_completed` 记录了意图从“编码态”到“实现态”的跃迁，`compensation_log` 记录了跃迁失败后的回退路径，确保每次状态变化都可追溯。
-

ewpage

D15. 深度_环境与管道隔离

layer: L3 type: reference prerequisites: [C09.工程_契约_产出物_证据, C13.工程_验证_门禁_状态机] source: ODD.0F车间-车间管理模型 last_updated: 2026-02-16

车间管理

前置知识: C13.工程_验证_门禁_状态机.md — 理解任务生命周期和管道编排后再阅读本文。

概述

车间是 ODD 的隔离执行环境。核心设计目标：人类决策负荷恒定，不随车间数量增长。无论 1 个还是 100 个车间并行，人类只在固定的决策点介入——契约定义、异常裁决、最终验收。

角色体系：

人类（决策者）

└── 架构师（设计者）

└── 经理（监控者）

└── 车间（容器）

└── AI 工人（全能执行者）

角色	做什么	不做什么
人类	产品决策、契约确认、异常裁决、解封授权	不写代码、不操作车间
架构师	需求澄清、契约设计、任务分解	不执行任务、不监控
经理	车间启停、心跳监控、故障回收、告警	不分配具体任务、不干预执行
车间	进程隔离、资源限制、心跳上报、会话管理	不做业务决策
AI 工人	任务领取、开发、测试、封版	不做架构决策、不监控其他车间

对等绑定关系：契约 \approx 功能模块 \approx 车间（1:1:1）。一个契约绑定一个功能模块，分配给一个车间执行。

AI 工人在车间内按任务阶段自动切换角色：开发工人 \rightarrow 测试工人 \rightarrow 集测工人 \rightarrow 封版工人。其中集测和封版 MUST 由开发者车间以外的车间执行（交叉审核）。

L1 • 基础

单车间模式。没有经理线程，没有调度，人工手动切换任务。

适用于一个人 + 一个 AI 辅助的小项目。车间就是“当前工作环境”，不需要管理。所有任务串行执行，人工决定

L2 • 标准

最小启动子集

L2 的核心变化：引入车间池和经理调度。最小启动需要三样东西：

1. 车间池：多个车间并行，每个车间有状态（idle / busy / offline）

2. 经理线程：定时轮询，发现空闲车间时分配优先级最高的契约
3. 契约队列：契约生效后进入队列等待调度，按优先级排序（P0 > P1 > P2 > P3）

生命周期：

契约生效 (queued) → 经理分配 (assigned) → 车间启动 (active)
 → 执行任务 → 契约完成 (completed) → 车间释放回池

完整 L2

经理职责

职责	频率
调度：发现空闲车间，分配优先级最高的契约	定时轮询
心跳监控：检测车间存活状态	每 30 秒
故障回收：车间故障时，契约回到队列	事件触发
超时检测：任务执行超时，标记异常	定时检查
告警：发现问题通知人类	事件触发

故障恢复流程

车间执行中 → 心跳丢失 → 经理检测到
 → 车间标记 offline
 → 契约回到队列（优先级 +1）
 → 已封存任务保持 sealed
 → 进行中的任务重置为 pending
 → 新车间接手继续执行

任务中断后的状态回收规则：

原任务状态	回收后状态
sealed	保持 sealed
quality_check / acceptance / in_progress / rework	重置为 pending
pending / blocked	保持不变

配置参数

参数	默认值	说明
max_workshops	3	车间池最大数量
heartbeat_interval	30s	心跳间隔
heartbeat_timeout	90s	心跳超时（3 次未收到）
task_timeout	30min	单任务执行超时
contract_timeout	4h	契约整体超时

L3 • 严格

L3 核心

在 L2 基础上增加知识积累和 VFS 持久化，解决两个关键问题：车间崩溃不丢数据、新车间接手能快速上手。

车间知识库：每个车间有独立的知识库，封存时提炼知识：

- 模式识别（成功的实现模式）
- 失败教训（踩过的坑）
- 最佳实践（验证过的做法）
- 上下文记忆（领域知识）

新车间接手时可热启动：加载已有知识，复用率目标 > 70%。

VFS（产出物存储）：

- 任务状态持久化到数据库，不在车间内存中保存
- 产出物（代码）持久化到数据库
- 车间故障不丢失数据，新车间可断点续传

L3 扩展

交叉审核质量保证 集测和封版时，MUST 由开发者车间以外的车间执行。若车间池只有一个车间，系统 MUST 发出告警通知人类。

审核报告 MUST 包含：

- **reviewer_workshop_id**（审核车间）
- **result**（passed / failed）
- **checklist**（检查项列表及结果）
- **issues**（失败时必填，发现的问题）

审核检查清单：

检查项	权重
ODD 输出匹配	40%
边界情况覆盖	20%
异常情况处理	20%
代码规范	10%
安全扫描	10%

审核质量监控（SHOULD）：

- 审核耗时 < 5 分钟（超时可能敷衍）
- 通过率 60-90%（过高可能放水，过低可能过严）
- 问题发现率 > 10%（过低可能审核不仔细）

上下文注入 车间执行任务前，由上下文引擎预查询数据库，一次性组装上下文注入 AI：

角色 + 阶段 + 任务ID

- 查询契约/规格/功能树/Bug历史/最佳实践
- 按 Token 预算裁剪

→ 一次性注入 AI

避免 AI 自己多次调用工具查询（每次都带完整上下文，浪费 Token）。

完整 YAML 示例

以下展示一个 L3 车间管理的完整配置和运行时状态：

—— 车间管理配置 ——

```
workshop_config:
  max_workshops: 5
  heartbeat_interval: 30s
  heartbeat_timeout: 90s
  task_timeout: 30min
  contract_timeout: 4h
  cross_review_required: true          # L3 强制交叉审核
```

—— 角色定义 ——

```
roles:
  human:
    responsibilities: [product_decision, contract_confirm, exception_ruling, unseal_auth]
  architect:
    responsibilities: [requirement_clarify, contract_design, task_decompose]
  manager:
    responsibilities: [workshop_lifecycle, heartbeat_monitor, fault_recovery, alert]
  workshop:
    responsibilities: [process_isolation, resource_limit, heartbeat_report, session_mgmt]
  ai_worker:
    phases: [develop, test, integration_test, seal]
    cross_review_rule: "integration_test 和 seal 阶段 MUST 由其他车间执行"
```

—— 契约队列 ——

```
contract_queue:
  - id: contract-login-api
    priority: P0
    status: queued
    queued_at: "2026-02-15T09:00:00Z"
  - id: contract-pay-api
    priority: P1
    status: assigned
    assigned_to: workshop-02
    assigned_at: "2026-02-15T09:05:00Z"
```

—— 车间池运行时状态 ——

```
workshop_pool:
  - id: workshop-01
    status: busy
```

```

current_contract: contract-login-api
heartbeat_last: "2026-02-15T14:30:15Z"
knowledge_base:
  patterns: 12
  failure_lessons: 5
  best_practices: 8
vfs:
  sealed_artifacts: [login-api-v1, login-test-v1]
  in_progress: [login-audit-handler]

- id: workshop-02
  status: busy
  current_contract: contract-pay-api
  heartbeat_last: "2026-02-15T14:30:12Z"

- id: workshop-03
  status: idle
  current_contract: null
  heartbeat_last: "2026-02-15T14:30:10Z"

# —— 故障恢复记录 ——
fault_recovery_log:
- workshop_id: workshop-04
  detected_at: "2026-02-15T13:00:00Z"
  cause: heartbeat_timeout
  contract_id: contract-refund-api
  recovery_action:
    contract_requeued: true
    priority_boost: "+1 → P0"
    sealed_tasks_preserved: [refund-validator]
    reset_tasks: [refund-processor]
    new_workshop: workshop-03

# —— 交叉审核记录 (L3) ——
cross_review:
- task_id: task-login-api-seal
  developer_workshop: workshop-01
  reviewer_workshop: workshop-03
  result: passed
  checklist:
    odd_output_match: { score: 95, passed: true }
    boundary_coverage: { score: 88, passed: true }
    exception_handling: { score: 82, passed: true }
    code_standard: { score: 90, passed: true }
    security_scan: { score: 100, passed: true }
  issues: []
  reviewed_at: "2026-02-15T14:00:00Z"
  duration: "3m42s"

```

理论来源

1. 人类决策负荷恒定 原则源自 ASTO 中的“决策瓶颈”分析——系统扩展时，自动化层可以线性扩展，但人类
2. 车间隔离 对应 ASTO 结构论中的“容器化”思想——执行单元之间的故障不应传播，每个容器独立运行、独
3. 交叉审核 源自软件工程中的“四眼原则”——代码的编写者不应是唯一的审查者，独立审查能发现编写者的
4. 热启动与知识复用 对应 ASTO 韧性篇中的“记忆沉淀”机制——系统的经验不应随执行者的更换而丢失，知

layer: L3 type: reference prerequisites: C09.工程_契约_产出物_证据 source: ODD.05上下文-上下文工程 last_updated: 2026-02-16

上下文工程完整参考

前置知识：阅读本文前，请先完成 C09.工程_契约_产出物_证据.md。

概述

上下文工程解决一个核心问题：给执行者（人或 AI）恰到好处的信息——不多不少，来源可查。多了浪费 token 和注意力，少了做出错误决策。

五条基本原则：

1. 最小必要注入：只给当前任务需要的信息。
 2. 证据不内联：长内容下沉为证据对象，用 evidence_ref 引用。
 3. 来源可查：每段上下文 MUST 标注来源（哪个契约/任务/证据）。
 4. 返工强制注入：rework 时 MUST 注入 failure_context（含 evidence_refs）。
 5. 知识湿度平衡：文档化程度应与项目等级匹配，过度文档化和文档不足同样有害。
-

L1 基础

上下文 = 手动准备。

执行任务前，手动确认以下信息可用：

- 契约的验收条件
- 任务的输入/输出规格
- 依赖的产出物在哪里

无自动化、无 token 控制。适用于人工开发、小项目。

知识湿度：高湿度。口头约定 + 简短笔记 + Git commit message。隐性知识靠团队默契传递。

最小启动子集（L2）

上下文 = 功能树查询 + 自动注入 + token 控制。

功能树查询

从功能树出发，精确定位当前任务相关的产出物、契约、证据，而不是塞给执行者整个项目。

注入分层

层级	内容	注入策略
强制注入	安全规则 + 架构约束 + 任务规格 + 验收条件	每次必须注入
按需注入	相关代码片段 + Bug 意向图 + 最佳实践	按任务需要选择性注入
返工注入	failure_context + 历史尝试记录	rework 时强制注入，避免重复同样的错误

Token 控制

系统 SHOULD 有 token 预算，按优先级裁剪低价值内容。

优先级排序：安全规则 > 任务规格 > 验收条件 > 相关代码 > 最佳实践。

知识湿度：中等湿度。结构化契约 + decision_narrative + Bug 意向图。关键决策脱水保存，日常知识保持湿润。

L3 核心

上下文 = 多层注入 + 车间知识库 + 双流记忆。

情报官（Intelligence Officer）

情报官是 L3 上下文工程的核心角色。在调用模型前，统一查询数据库并组装上下文，避免“多次调用 + 重复上下文”。

- 输入：角色、阶段、task_id、contract_id、module_id。
- 输出：组装好的上下文 + 来源元数据 + token 估算。

流程：

任务下达 → 情报官接收 → 查询数据库（按角色/阶段）
→ 过滤裁剪（Token预算）→ 分层组装 → 一次性注入模型

多层上下文注入模型

按优先级分层注入，核心层 MUST 存在，扩展层按项目需求增减：

核心层（MUST）：

优先级	层	内容
P0	安全硬边界	安全规则，不可穿透
P0	架构硬边界	架构约束，不可穿透

优先级	层	内容
P0	流程硬边界	流程规范，不可穿透
P0	任务规格	当前任务的输入/输出/验收
P0	验收与测试	验收条件详情
P0	纠正反馈	rework 时的 failure_context

扩展层（SHOULD / MAY）：

优先级	层	内容
P1	系统级规范	全局规范文档
P1	车间知识库	Bug 意向图 / 最佳实践
P2	执行上下文	相关代码片段
P2	项目自定义层	国际化规则、合规要求、领域术语表等

完整的五层结构：

1. 硬边界层：安全规则、架构约束、流程规范——强制注入，不可穿透。
2. 功能树层：索引查询，定位相关产出物。
3. 依赖图层：查询上下游关系。
4. 知识库层：车间封存时提炼的语义知识（Bug 意向图、最佳实践、隐式需求）。
5. 任务执行层：任务规格、验收、返工反馈。

角色视野矩阵

不同角色看到不同范围的上下文：

角色	可见范围	不可见
架构师	全局架构、契约列表、功能树全貌	具体实现细节
经理	契约状态、任务调度、资源分配	代码细节
车间	本模块上下文/规格/执行信息	其他模块代码
工人	当前任务规格/验收/执行上下文	其他任务

阶段注入矩阵

不同阶段注入不同内容：

阶段	注入内容
契约生成	功能树全貌 + 相关契约列表
清晰度检测	契约详情 + 模糊词库 + 历史模糊问题
任务生成	契约详情 + 依赖关系 + 已封版产出物
开发/测试/审核	任务规格 + 验收标准 + 相关代码 + Bug 历史 + 最佳实践

会话治理（Session Governance）

当契约定义或任务执行涉及多轮 AI 交互时，对话方向容易漂移。会话治理管理多轮交互的方向一致性。

三层防漂移机制：

层级	机制	成本	触发条件
L1 范围锚定	每轮对话开头重申当前任务的 scope_in / scope_out	零成本	每轮自动注入
L2 结构检查点	每 N 轮（建议 N=5）暂停，对比当前讨论内容与原始契约的偏离度	低成本	定时触发
L3 语义漂移感知	用 LLM 对比当前对话摘要与原始目标的语义距离，超过阈值时告警	中成本	偏离度 > 阈值

漂移处理：

- 检测到漂移时，系统 SHOULD 生成漂移报告（drift_report），记录偏离方向和程度。
- 轻度漂移（讨论范围扩大但未偏离目标）：注入提醒，继续。
- 重度漂移（讨论方向与原始目标无关）：暂停会话，要求人工确认是否调整契约范围。
- 漂移报告作为证据存档，用于优化后续会话的上下文注入策略。

适用场景：

- AI 协作的契约设计会话（多轮讨论需求细节）。
- AI 多角色协作中的跨角色对话（架构师与车间的交互）。
- 长时间运行的自动化流水线（多阶段 AI 调用链）。

Token 预算管理

预算分配建议：

类别	Token	预算	优先级
契约约束	200		P0
任务规格	300		P0
验收标准	200		P0
Bug 历史	300		P1
最佳实践	300		P1
相关代码	500		P2

超额时按优先级裁剪：先砍 P2，再砍 P1，P0 不可裁剪。

预置参数组（按场景快速切换）：

- full_injection: 测试阶段，全量注入。
- minimal: 省 Token，只注入 P0。
- expert_focus: 强化专家身份。
- bug_prevention: 强化 Bug 历史。
- rework_focus: 强化纠正反馈。
- balanced: 正式环境，均衡分配。

提示词模板骨架

硬边界 (P0)

{L1_security_boundary}
{L2_architecture_boundary}
{L3_process_boundary}

专家身份

{expert_identity}

任务规格 (P0)

{L14_task_spec}
{L15_acceptance_criteria}

知识库 (P1)

{L12_bug_patterns}
{L12_best_practices}

纠正反馈 (rework时)

{L17_failure_context}

相关代码 (P2)

{L16_code_context}

提示词存库规则

- 提示词 MUST 存数据库，按版本管理，可回滚与统计。
- 记录使用情况：成功率、返工率、Token 消耗。

L3 扩展

车间知识库

- 每个车间封存时 SHOULD 提炼知识到知识库（情景记忆 + 语义记忆）。
- 新车间启动时 SHOULD 加载相关知识（热启动）。
- 热启动 MUST 校验知识版本与当前产出物版本一致。

双流记忆

记忆类型	生命周期	内容
情景记忆（短期）	本次执行	关键决策、遇到的问题
语义记忆（长期）	跨任务沉淀	模式、规律、最佳实践

知识湿度平衡

文档化程度与项目等级匹配：

等级	湿度	策略
L0/L1	高湿度	口头约定 + 简短笔记 + Git commit message
L2	中等湿度	结构化契约 + decision_narrative + Bug 意向图
L3	低湿度	形式化规格 + 完整证据链 + 车间知识库

判断信号：

- 团队反馈“文档太多没人看” → 湿度太低，减少文档化要求。
- 团队反馈“新人上手太慢、关键人离职后知识断裂” → 湿度太高，增加文档化。
- decision_narrative（封存时记录决策上下文）是“湿知识脱水保存”的最佳实践。

监控指标

系统 SHOULD 监控以下指标以持续优化上下文注入策略：

指标	说明
tokens_injected / tokens_saved	注入量与节省量
layers_included / layers_skipped	注入层数与跳过层数
cache_hit_rate	上下文缓存命中率
task_success / rework_count	任务成功率与返工次数

L1/L2/L3 上下文策略差异速查

维度	L1	L2	L3
准备方式	手动	功能树查询 + 自动注入	情报官统一组装
Token 控制	无	有预算，按优先级裁剪	精细预算 + 预置参数组
注入分层	无	强制/按需/返工 三层	五层模型（硬边界→功能树→依赖图→知识库→任务执行）
知识库	无	无	车间知识库 + 双流记忆
会话治理	无	无	三层防漂移（锚定→检查点→语义感知）
提示词管理	无	无	存库 + 版本管理 + 统计
角色视野	无区分	无区分	角色视野矩阵
知识湿度	高	中	低

完整 YAML 示例（L3 情报官上下文组装）

```
# === 情报官请求 ===
context_request:
  role: workshop           # 车间角色
  phase: development       # 开发阶段
  task_id: "TSK-AUTH-001"
  contract_id: "CTR-AUTH-001"
  module_id: "MOD-AUTH"
  preset: balanced         # 使用均衡参数组
```

```

# === 情报官输出 ===
context_response:
  token_estimate: 1650
  sources:
    - { type: contract, id: "CTR-AUTH-001" }
    - { type: task, id: "TSK-AUTH-001" }
    - { type: evidence, id: "EVD-BUG-012" }
    - { type: knowledge_base, id: "KB-AUTH-PATTERNS" }

layers:
  # --- P0 硬边界 ---
  security_boundary:
    content: "禁止明文存储密码；禁止在日志中输出 token"
    source: "SECURITY-POLICY-v2"
    tokens: 50

  architecture_boundary:
    content: "认证模块必须通过 API Gateway；禁止直连数据库"
    source: "ARCH-DECISION-003"
    tokens: 60

  process_boundary:
    content: "L3 任务必须走慢轨；必须通过 mutation_test + adversarial_test"
    source: "ODD-PROCESS-v1"
    tokens: 40

  # --- P0 任务规格 ---
  task_spec:
    content: |
      输入: HTTP POST {email, password}
      输出: {success, token?, error_code?}
      副作用: db_write(users_table), event_emission(user_registered)
    source: "TSK-AUTH-001"
    tokens: 150

  acceptance_criteria:
    content: |
      - [hard+EN] 注册成功返回 JWT token
      - [hard+EN] 密码强度 ≥ 8 位含大小写
      - [soft+NEN] 界面提示友好
    source: "CTR-AUTH-001"
    tokens: 120

  # --- P1 知识库 ---
  bug_patterns:
    content: |
      - BUG-012: 并发注册同一邮箱导致重复记录（修复：唯一约束 + 乐观锁）

```

```

    - BUG-008: JWT 过期时间未设置导致 token 永不失效
    source: "KB-AUTH-PATTERNS"
    tokens: 200

best_practices:
  content: |
    - 密码哈希使用 bcrypt, cost factor  $\geq 12$ 
    - JWT 有效期建议 15min, refresh token 7d
  source: "KB-AUTH-PATTERNS"
  tokens: 150

# --- P2 相关代码 ---
code_context:
  content: |
    // src/api/auth/middleware.ts (已封存 v1.2.0)
    export function validateToken(token: string): TokenPayload { ... }
  source: "ART-AUTH-MW-001"
  tokens: 300

# --- 会话治理配置 ---
session_governance:
  scope_anchor:
    scope_in: "实现用户注册 API, 含邮箱验证"
    scope_out: "不含第三方 OAuth 登录"
  checkpoint_interval: 5 # 每 5 轮检查一次
  drift_threshold: 0.3 # 语义距离阈值

# --- 裁剪记录 ---
trimming_log:
  total_budget: 1800
  total_used: 1650
  skipped_layers: []
  trimmed_layers:
    - { layer: "code_context", original: 500, trimmed_to: 300, reason: "预算限制" }

```

理论来源

1. 知识三态流转 (ASTO 认识论 P03): 知识在湿知识 (存在于大脑)、干知识 (固化在文档)、活知识 (干知
 2. 双流记忆模型: 情景记忆 (短期) 与语义记忆 (长期) 的分离设计, 确保短期执行经验和长期沉淀模式各得
 3. 最小必要原则: 信息过载与信息不足同样有害。上下文注入的核心是 “够用就好”, 而非 “越多越好”。
-
-

layer: L3 type: reference prerequisites: [C09.工程_契约_产出物_证据, C13.工程_验证_门禁_状态机] source: ODD.08管道-管道与组合 last_updated: 2026-02-16

管道与依赖完整参考

前置知识：阅读本文前，请先完成 C09.工程_契约_产出物_证据.md 和 C13.工程_验证_门禁_状态机.md。

概述

产出物不是孤立的。原子产出物通过管道组合擢升为新产出物，这是蜂巢生长的方式。管道是有契约的组装线——它声明接收什么、产出什么、怎样算成功。

核心规则：

- 管道 (Pipeline) = 可复用的“输入产出物 → 转换 → 输出产出物”单元。
- 每个管道 MUST 拥有自己的契约，声明输入、输出、验收。
- 产出物之间的依赖关系 MUST 显式声明，不得隐式依赖。
- 依赖图 MUST 为有向无环图 (DAG)，禁止循环依赖。
- 管道输入 MUST 引用已封存的产出物，不得引用未封存的中间状态。
- 当某个产出物被替换时，所有依赖它的下游管道 MUST 重新验证。

组合模式

五种基本组合方式：

模式	语法	说明
串联	$A \gg B$	A 的输出作为 B 的输入
并联	$A \parallel B$	同时执行，结果合并
条件	$\text{cond} ? A : B$	按条件选路径
循环	$A * n$	重复执行，带上限
聚合	$A + B \rightarrow C$	多个产出物合并为一个

管道边界原则 (DB 层 vs 应用层)

- 能在 DB 完成的就放 DB：数据变形、聚合、约束、状态机转换。
- 必须出 DB 的放应用层：外部调用、文件 IO、AI 推理、人机交互。
- 混合管道：DB 负责数据，应用负责副作用与外部交互。

决策树：

外部调用？是 → 应用层

否 → 纯数据变形？是 → DB 层

否 → 复杂业务/需要人类？是 → 应用层

否 → DB 层

事务边界：

- DB 事务只包含 DB 操作，外部调用必须在事务外。

- 外部失败必须有补偿（释放库存/取消订单）。

协调模式：

- 编排（Orchestration）：应用层显式编排步骤。
- 事件驱动（Choreography）：DB 触发事件，应用层监听响应。
- Saga：每步有补偿，失败时反向执行补偿。

组合安全规则

- 类型兼容 MUST 检查：A.output 的类型可赋给 B.input。
- 错误传播 MUST 声明：组合契约必须覆盖子契约的所有错误情况。
- 封存约束 MUST 传递：组合产出物的 seal MUST 包含所有输入产出物的 evidence_ref。

L1 基础

管道 = 手动组合。不需要正式的管道定义，但产出物之间的依赖关系要记录清楚。

依赖管理

在产出物的 metadata 中记录 **depends_on**（列出依赖的产出物 ID）。人工确认依赖关系正确即可。

```
artifact:
  id: string
  depends_on: [string]           # 依赖的产出物 ID 列表
  produced_by: string           # 哪个契约/任务产生的
```

组合方式：手动指定输入产出物，手动验证输出。

最小启动子集（L2）

管道 = 有契约的组装单元，依赖关系由系统维护。

管道定义

```
pipeline:
  id: string
  name: string
  contract:
    inputs:
      - artifact_type: code_module
        state: sealed           # MUST 已封存
      - artifact_type: test_suite
        state: sealed
    output:
      artifact_type: integrated_module
  acceptance: string           # 验收条件
  error_handling: string       # 错误处理策略
```

每个管道拥有自己的契约，明确声明：

- 输入：接收哪些产出物（类型 + 状态要求）
- 输出：产生什么产出物
- 验收：输出满足什么条件算成功
- 错误处理：失败时的策略

DAG 依赖图

系统维护产出物依赖图（DAG）：

- 执行管道前自动检查：所有输入产出物是否已封存。
- 当输入产出物被替换时，系统标记下游管道为“待重新验证”。

artifact:

```
# ...L1 字段...
state: sealed | stale           # stale = 依赖已变，需重新验证
dependency_graph:
  upstream: [artifact_id]       # 我依赖谁
  downstream: [artifact_id]    # 谁依赖我
```

影响分析

当一个产出物被替换时，影响如何传导：

产出物 X 被替换

- 查询依赖图：哪些管道的输入包含 X？
- 标记这些管道的输出产出物为 stale
- 递归向下：这些 stale 产出物又被谁依赖？
- 直到没有更多下游

这是 L2 的核心能力：变更一个契约/产出物，系统能自动告诉你哪些下游受影响。

L3 核心

管道 = 可审计的自动化组装线，依赖关系强制执行、自动级联。

管道增强

- 管道执行 MUST 产生完整的审计日志（输入产出物 ID + 输出产出物 ID + 门禁结果 + 时间戳）。
- 组合产出物的 seal MUST 包含所有输入产出物的 seal_hash（证据链传递）。
- 组合抽象（泛型/继承/模板/Mixin）MUST 在最终实例化时落地为明确的输入输出与错误声明。

依赖管理增强

- 产出物替换时，系统自动触发下游重新验证（级联重验）。
- 重验失败的下游产出物自动进入 rework。
- 依赖图变更 MUST 记录在审计日志中。

L2 只是标记 stale，L3 会自动触发重验并处理失败。

并行编排

- 无依赖关系的管道可以并行执行（由调度器根据 DAG 自动判断）。
- 有依赖关系的管道严格按拓扑序执行。

管道执行记录

```
pipeline_execution:
  pipeline_id: string
  inputs: [{artifact_id, seal_hash}] # 输入快照
  output: {artifact_id, seal_hash} # 输出快照
  gate_results: [evidence_ref] # 门禁证据
  executed_at: datetime
  triggered_by: string # 手动 | 级联重验 | 调度
```

产出物封存链

```
artifact:
  # ...L2 字段...
  seal:
    seal_hash: string
    input_seal_hashes: [string] # 所有输入产出物的 seal_hash
    sealed_at: datetime
```

封存链的意义：任何一个产出物的 seal_hash 都可以追溯到它所有上游输入的 seal_hash，形成完整的证据链。

L3 扩展

组合抽象机制

四种抽象方式，用于减少管道定义的重复：

机制	用途	示例
泛型	以类型参数复用契约	CRUD<T, ID>
继承	公共错误/中间件/头部在基类定义	BaseAPI → UserAPI
模板	结构复用	分页列表、批处理
Mixin	横切能力组合	可审计、可缓存

规则：所有抽象 MUST 在最终实例化时落地为明确的输入输出与错误声明。抽象是为了减少重复，不是为了隐藏复

级联重验的完整流程

```
产出物 X v1.0 被替换为 X v2.0
→ 系统查询 DAG：下游管道 P1、P2 依赖 X
→ P1 自动重新执行
  → P1 输出 Y v2.0（重验通过）
    → 系统查询 DAG：下游管道 P3 依赖 Y
    → P3 自动重新执行
```

- P3 输出 Z v2.0 (重验通过)
- P2 自动重新执行
 - P2 输出 W (重验失败)
 - W 进入 rework
 - 通知相关车间

L1/L2/L3 管道复杂度差异

维度	L1	L2	L3
管道定义	无正式定义	有契约的组装单元	可审计的自动化组装线
依赖管理	手动记录 depends_on	系统维护 DAG	系统维护 DAG + 自动级联
影响分析	无	标记 stale	自动级联重验
封存链	无	无	seal_hash 链传递
并行编排	无	无	调度器根据 DAG 自动并行
审计日志	无	无	完整执行记录
组合抽象	无	无	泛型/继承/模板/Mixin
重验失败处理	人工处理	人工处理	自动进入 rework

依赖管理核心原则

1. 只依赖已封存的东西：不确定的产出物不能被引用，这是质量的根基。
2. 依赖必须显式：“我依赖谁”写在明处，不能靠猜。
3. 变动必须传导：上游变了，下游必须知道并重新验证。

完整 YAML 示例 (L3 管道编排)

```
# === 管道定义：集成构建 ===
pipeline:
  id: "PL-BUILD-001"
  name: "用户模块集成构建"
  contract:
    inputs:
      - artifact_type: code_module
        artifact_id: "ART-AUTH-001"
        state: sealed
      - artifact_type: code_module
        artifact_id: "ART-USER-001"
        state: sealed
      - artifact_type: test_suite
        artifact_id: "ART-TEST-001"
        state: sealed
    output:
      artifact_type: integrated_module
      artifact_id: "ART-INTEGRATED-001"
  acceptance: "集成测试全部通过，覆盖率 ≥ 85%"
  error_handling: "任一输入缺失或未封存则阻塞；集成失败则输出进入 rework"
```

=== 产出物依赖图 ===

```
artifacts:
  - id: "ART-AUTH-001"
    type: code_module
    path: "src/api/auth/"
    state: sealed
    seal:
      seal_hash: "sha256:a1b2c3..."
      sealed_at: "2026-02-16T14:00:00Z"
    dependency_graph:
      upstream: []
      downstream: ["ART-INTEGRATED-001"]

  - id: "ART-USER-001"
    type: code_module
    path: "src/api/user/"
    state: sealed
    seal:
      seal_hash: "sha256:d4e5f6..."
      sealed_at: "2026-02-16T14:30:00Z"
    dependency_graph:
      upstream: []
      downstream: ["ART-INTEGRATED-001"]

  - id: "ART-TEST-001"
    type: test_suite
    path: "tests/integration/"
    state: sealed
    seal:
      seal_hash: "sha256:g7h8i9..."
      sealed_at: "2026-02-16T15:00:00Z"
    dependency_graph:
      upstream: []
      downstream: ["ART-INTEGRATED-001"]

  - id: "ART-INTEGRATED-001"
    type: integrated_module
    path: "dist/user-module/"
    state: sealed
    seal:
      seal_hash: "sha256:j0k1l2..."
      input_seal_hashes:
        - "sha256:a1b2c3..." # ART-AUTH-001
        - "sha256:d4e5f6..." # ART-USER-001
        - "sha256:g7h8i9..." # ART-TEST-001
      sealed_at: "2026-02-16T16:00:00Z"
    dependency_graph:
```

```

    upstream: ["ART-AUTH-001", "ART-USER-001", "ART-TEST-001"]
    downstream: ["ART-DEPLOY-001"]

# === 管道执行记录 ===
pipeline_execution:
  pipeline_id: "PL-BUILD-001"
  inputs:
    - { artifact_id: "ART-AUTH-001", seal_hash: "sha256:a1b2c3..." }
    - { artifact_id: "ART-USER-001", seal_hash: "sha256:d4e5f6..." }
    - { artifact_id: "ART-TEST-001", seal_hash: "sha256:g7h8i9..." }
  output:
    artifact_id: "ART-INTEGRATED-001"
    seal_hash: "sha256:j0k1l2..."
    gate_results: ["EVD-INT-001", "EVD-INT-002"]
    executed_at: "2026-02-16T16:00:00Z"
    triggered_by: "manual"

# === 影响分析示例: ART-AUTH-001 被替换 ===
impact_analysis:
  trigger: "ART-AUTH-001 v1.0 → v2.0"
  affected:
    - artifact_id: "ART-INTEGRATED-001"
      current_state: stale
      action: "级联重验"
      pipeline: "PL-BUILD-001"
    - artifact_id: "ART-DEPLOY-001"
      current_state: stale
      action: "等待 ART-INTEGRATED-001 重验完成后级联"
      pipeline: "PL-DEPLOY-001"

# === 并联管道示例 ===
pipeline:
  id: "PL-PARALLEL-001"
  name: "前后端并行构建"
  steps:
    - type: parallel # 并联
      branches:
        - pipeline_ref: "PL-FRONTEND-BUILD"
        - pipeline_ref: "PL-BACKEND-BUILD"
    - type: sequential # 串联（等并联完成后）
      pipeline_ref: "PL-INTEGRATION-TEST"
    - type: conditional # 条件分支
      condition: "env == 'production'"
      true_branch: "PL-FULL-DEPLOY"
      false_branch: "PL-STAGING-DEPLOY"

```

理论来源

1. 蜂巢生长模型：原子产出物通过管道组合擢升为新产出物，类似蜂巢的渐进式构建。管道是组合的基本单元。
2. DAG 约束：依赖图必须为有向无环图，这是拓扑排序和并行调度的数学基础。循环依赖会导致无法确定执行顺序。
3. 封存链传递：组合产出物的 seal 包含所有输入的 seal_hash，形成可追溯的证据链。这是“信任传递”的工业级应用。

ewpage

D17. 深度_监控_安全与高级治理

layer: L3 type: reference prerequisites: [C13. 工程_验证_门禁_状态机, D15. 深度_环境与管道隔离]

赛马与预警

前置知识：C13. 工程_验证_门禁_状态机.md、D15. 深度_环境与管道隔离.md
理解任务状态流转和验证机制后再阅读本文。

概述

赛马和预警是 ODD 执行层的两个协同机制：

- 赛马（智能赛马）：根据任务复杂度选择“够用”的参与者，失败后先诊断原因再决定是否升级。不是选最强，而是选够用。
- 预警（分级预警与红绿灯）：用灯状态让人一眼看到全局，用升级规则让系统自动处理能自动处理的，只在必要时介入。

两者的协同关系：

赛马产生执行数据（成功/失败/返工次数/升级记录）

- 预警分析数据（灯状态/预警等级/趋势）
 - 人类决策（选择题菜单，不从零分析）
 - 决策反馈回赛马（调整分级规则/契约/门禁）
-

第一部分：赛马机制

核心原则

- 参与者（模型/人）选择 MUST 基于任务复杂度，不得一律使用最强资源
- 失败后 MUST 先诊断原因，再决定升级路径；盲目升级是反模式
- 成本 SHOULD 可监控、可预算、可告警

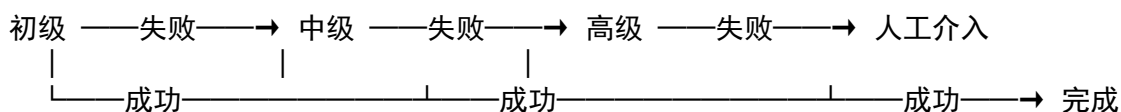
失败诊断优先

失败 MUST 归入以下五类原因之一，对应不同处理路径：

失败原因	处理路径	是否消耗升级次数
上下文不足	补充上下文 → 同级重试	否
契约模糊	修正契约/任务描述 → 同级重试	否
测试不合理	审核测试有效性 → 修正测试 → 同级重试	否
执行能力不足	升级执行者	是
未知问题	升级执行者（保守策略）	是

只有“执行能力不足”和“未知问题”才触发升级。统计显示约 60% 的失败源于上下文不足，30% 源于契约模糊，仅 10% 是执行者能力问题。

升级路径



到达最高级仍失败 MUST 升级人类决策。

第二部分：预警系统

灯状态定义

所有可监控的阶段 MUST 使用统一的灯状态：

灯状态	含义
白色	未到达该阶段
绿闪（呼吸闪烁）	正在进行中，正常
红闪（呼吸闪烁）	进行中但有问题，需人类关注
长绿（静止）	该阶段已完成

预警分级原则

- 系统 MUST 按严重程度分级响应，不得对所有问题一律告警
- 低级预警 MUST 自动处理，不打扰人类
- 红灯上报 MUST 阻塞任务，等待人类决策
- 人类决策 MUST 以选择题形式提供选项，降低认知负担

红闪触发条件

任何阶段出现以下情况 MUST 转为红闪：阻塞状态（blocked）、返工次数达到告警阈值、门禁检查失败、变异测试

L1 • 基础

赛马：无正式赛马。执行者固定（人或指定模型）。失败后人工决定是否更换。

预警：无正式灯状态。任务状态（pending / in_progress / done / rework）即为信号。失败后由人工判断是否继续。

L2 • 标准

最小启动子集

最小启动需要两样东西：

1. 任务分级规则：根据复杂度预选执行者等级
2. 返工预警四级：按返工次数自动升级预警等级

任务分级参考因素：产出物类型（配置 vs 算法）、预估代码量（< 50 行 vs > 200 行）、依赖数量（0-2 个 vs > 5 个）、历史返工率。

返工预警四级：

返工次数	预警等级	处理方式
0-1 次	正常（绿闪）	正常执行
2 次	黄灯预警	记录日志，诊断失败原因，不打扰人类
4 次	橙灯预警	通知人类，但不阻塞任务
≥6 次	红灯上报	阻塞任务，等待人类决策

完整 L2

赛马：任务分级规则

rules:

- name: 简单任务
conditions:
 - artifact_type in [config, script, simple_query]
 - estimated_lines < 50executor_level: junior
- name: 标准任务
conditions:
 - artifact_type in [code_module, api_endpoint]
 - estimated_lines < 200
 - dependencies < 3executor_level: standard
- name: 复杂任务
conditions:
 - artifact_type in [algorithm, architecture]
 - or: rework_count > 1executor_level: senior

失败升级按返工次数自动触发：

- 0-1 次返工：初级
- 2-3 次返工：中级
- 4-5 次返工：高级

- ≥6 次返工：触发人类告警

诊断方式：失败时系统记录失败类型（编译错误/测试失败/验收失败/超时），供人工参考。

预警：灯状态可视化 每个任务的关键阶段显示灯状态：



各阶段灯色规则：

- 当前阶段 = 绿闪（正常进行）或红闪（有问题）
- 已完成阶段 = 长绿
- 未到达阶段 = 白色
- rework 状态 = 开发中列红闪

预警：变异测试预警（三级）

- 正常（首次 < 阈值）：自动返工补充测试
- 黄灯预警（2 次 < 阈值）：记录日志，继续尝试
- 红灯上报（3 次 < 阈值）：阻塞任务，等待人类决策

预警：人类决策菜单 红灯触发时提供选择题菜单：

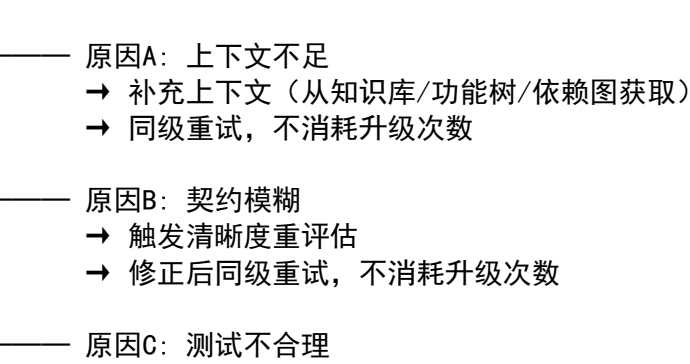
选项	说明
修改契约描述	契约不清晰导致执行者无法正确实现
降低门禁阈值	临时降低变异测试阈值
人工补充	人类补充关键测试用例或代码片段
取消任务	放弃该任务
继续尝试	重置计数，忽略预警继续执行

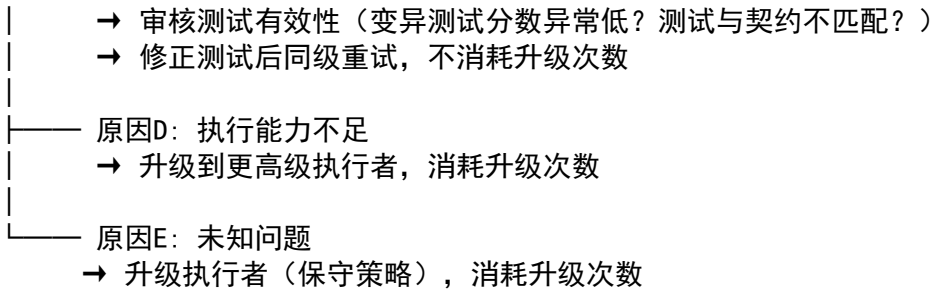
L3 • 严格

L3 核心

赛马：智能诊断 失败后由诊断者（经理 AI 或规则引擎）分析根因，决定处理路径：

任务失败





赛马：成本监控

- 系统 SHOULD 记录每个任务的执行者等级、调用次数、升级次数
- 系统 SHOULD 提供聚合视图：一次成功率、升级成功率、人工介入率
- 升级率 > 30% 时 SHOULD 触发预警，提示检查任务分级规则或契约质量

预警：全阶段灯状态 L3 增加变异测试、对抗测试、交叉审查阶段的灯列：

准备 → 开发中 → 质检中 → 变测中 → 对抗中 → 交叉审查 → 验收中 → 封存
○ ○ ○ ○ ○ ○ ○ ○

各阶段的红闪触发条件：

- 准备：blocked 状态（依赖阻塞）
- 开发中：rework_count ≥ 2（预警）；rework_count ≥ 6（必须人类介入）
- 质检中：测试失败
- 变测中：mutation_score < 阈值
- 对抗中：发现高危漏洞
- 交叉审查：审查方拒绝通过
- 验收中：人工验收拒绝

预警与赛马联动

- 黄灯预警 → 触发赛马的诊断流程
- 诊断结果为“执行能力不足” → 自动升级执行者等级
- 诊断结果为“契约模糊” → 触发清晰度重评估
- 红灯上报 → 阻塞，同时冻结赛马升级，等待人类决策

L3 扩展

审计要求

- 所有预警升级 MUST 记录在审计日志中
- 人类决策 MUST 记录选择的选项、决策者和时间
- 红灯上报的频率和处理时间 SHOULD 作为系统健康度指标

完整 YAML 示例

以下展示赛马与预警协同运行的完整数据结构：

```

# —— 赛马：任务分级规则 ——
racing_rules:
  - name: 简单任务
    conditions:
      - artifact_type in [config, script, simple_query]
      - estimated_lines < 50
    executor_level: junior

  - name: 标准任务
    conditions:
      - artifact_type in [code_module, api_endpoint]
      - estimated_lines < 200
      - dependencies < 3
    executor_level: standard

  - name: 复杂任务
    conditions:
      - artifact_type in [algorithm, architecture]
      - or: rework_count > 1
    executor_level: senior

# —— 赛马：任务执行记录 ——
task_execution:
  task_id: task-login-api
  executor_level: standard          # 当前执行者等级
  rework_count: 3
  upgrade_count: 1                 # 实际消耗的升级次数
  cost: 0.42                       # 执行成本（美元）
  diagnosis_history:
    - attempt: 1
      failure_type: test_failed
      diagnosis: context_insufficient # 上下文不足
      action: inject_context          # 补充上下文，同级重试
    - attempt: 2
      failure_type: test_failed
      diagnosis: contract_ambiguous  # 契约模糊
      action: clarify_contract       # 修正契约，同级重试
    - attempt: 3
      failure_type: test_failed
      diagnosis: executor_insufficient # 执行能力不足
      action: upgrade_executor        # 升级到 standard

# —— 赛马：成本聚合视图 ——
racing_metrics:
  period: "2026-02-01 ~ 2026-02-15"
  total_tasks: 47
  first_attempt_success_rate: 0.68 # 一次成功率 68%
  upgrade_success_rate: 0.89        # 升级后成功率 89%

```

```

human_intervention_rate: 0.04          # 人工介入率 4%
avg_cost_per_task: 0.35
upgrade_rate: 0.21                     # 升级率 21% (< 30% 阈值, 正常)

# —— 预警: 任务灯状态 ——
task_warning:
  task_id: task-login-api
  current_stage: quality_check
  lamp_states:
    prepare: green_solid               # 已完成
    develop: green_solid               # 已完成
    quality_check: green_blink         # 进行中, 正常
    mutation_test: white               # 未到达
    adversarial_test: white            # 未到达
    cross_review: white                # 未到达
    acceptance: white                  # 未到达
    seal: white                        # 未到达
  warning_level: normal
  rework_count: 1
  mutation_attempts: 0

# —— 预警: 红灯上报记录 ——
red_alert_log:
- task_id: task-refund-processor
  triggered_at: "2026-02-14T16:30:00Z"
  trigger_reason: "rework_count >= 6"
  warning_level: red
  racing_state_frozen: true            # 赛马升级已冻结
  human_decision:
    decision: modify_contract          # 人类选择: 修改契约描述
    decided_by: "tech-lead"
    decided_at: "2026-02-14T17:00:00Z"
    note: "契约缺少退款超时处理的边界条件"
  resolution:
    contract_updated: true
    rework_count_reset: true
    resumed_at: "2026-02-14T17:15:00Z"
    outcome: sealed                    # 最终成功封存

# —— 预警: 审计日志 ——
warning_audit:
- event: warning_escalation
  task_id: task-refund-processor
  from_level: orange
  to_level: red
  timestamp: "2026-02-14T16:30:00Z"
- event: human_decision
  task_id: task-refund-processor

```

```
decision: modify_contract
decided_by: "tech-lead"
timestamp: "2026-02-14T17:00:00Z"
```

理论来源

- 1. “够用就好”选择策略 源自 ASTO 资源论中的“最小充分资源”原则——系统应使用满足需求的最低成本资源，任务不需要最强执行者。
- 2. 失败诊断优先于升级 对应 ASTO 韧性篇中的“根因分析优先”——盲目升级资源是对症状的反应，而非对原因。
- 3. 分级预警 源自工业控制系统中的“告警疲劳”研究——当所有问题都以相同优先级告警时，操作员会忽略所有告警。
- 4. 人类决策菜单（选择题而非问答题） 对应认知负荷理论——在压力下，人类做选择题的决策质量远高于开放性问题。
- 5. 赛马与预警联动 体现 ASTO 中“感知-分析-决策-执行”闭环——赛马是执行层，预警是感知层，人类决策是分析层。

layer: L3 type: reference prerequisites: [C09. 工程_契约_产出物_证据, C13. 工程_验证_门禁_状态机]

前置知识: C13. 工程_验证_门禁_状态机.md、C09. 工程_契约_产出物_证据.md — 理解验证机制和证据封存后再阅读本文。

概述

功能树告诉你“做什么”，Bug 意向图告诉你“别踩什么坑”，最佳实践告诉你“怎么做最好”。三者合称三大法宝。三大法宝的注入时机：

阶段	Bug 意向图的作用	最佳实践的作用
契约生成	补充验收条件（历史易错点）	指导契约约束
代码生成	提供对抗测试向量	指导代码实现
测试阶段	经验库攻击（按历史频率排序）	—

核心规则：

- 每种产出物类型（artifact_type）SHOULD 关联自己的 Bug 意向图和最佳实践
- 三大法宝的内容 MUST 作为隐式需求注入契约和任务上下文
- Bug 意向图和最佳实践 SHOULD 持续从执行历史中学习和更新，不是静态配置

L1 • 基础

手动维护。在项目文档或笔记中记录常见坑和推荐做法，开发前人工浏览。

常见坑清单示例：

- 密码明文存储
- 未处理 NULL 输入

- 未考虑并发冲突
- 错误信息泄露内部细节

推荐做法清单示例：

- 密码用 bcrypt/argon2 加密
- 所有输入做边界校验
- 数据库操作做事务保护
- 错误信息统一格式，不暴露堆栈

无自动注入，靠人工记忆和纪律。小项目够用。

L2 • 标准

最小启动子集

最小启动需要两样东西：

1. Bug 意向图：每种产出物类型维护一张结构化的易错模式表
2. 最佳实践：每种产出物类型维护一组推荐做法，区分强制和可选

创建任务时，系统根据 **artifact_type** 查询对应的 bug_patterns 和 best_practices，自动注入到任务上下文中。**true** 的最佳实践强制注入，其余按 priority 排序注入（受 token 预算裁剪）。

完整 L2

Bug 意向图结构 每种产出物类型维护一张 Bug 意向图（Bug Intention Map），记录该类型的历史易错模式：

bug_patterns:

- id: BP-001
 - pattern: "密码明文存储" # 模式名称
 - detection_rule: "stored_password == hash(input_password, salt)"
 - severity: critical # critical / high / medium / low
 - frequency: 127 # 历史发现次数
 - prevention: "使用 bcrypt 或 argon2"
- id: BP-002
 - pattern: "无登录失败计数"
 - detection_rule: "login_attempts >= 5 IMPLIES account_locked == true"
 - severity: high
 - frequency: 89
 - prevention: "实现失败计数 + 账户锁定"

最佳实践结构

best_practices:

- id: PR-001
 - practice: "使用参数化查询"
 - rationale: "防止 SQL 注入"
 - anti_pattern: "字符串拼接 SQL" # 对应的反模式
 - priority: 9 # 优先级 1-10
 - is_mandatory: true # 是否强制

- id: PR-002
 - practice: "SECURITY DEFINER 需要固定 search_path"
 - rationale: "防止搜索路径注入攻击"
 - anti_pattern: "省略 search_path 设置"
 - priority: 8
 - is_mandatory: true

产出物类型关联示例 将三大法宝关联到具体的产出物类型:

```

artifact_types:
  auth_login:
    bug_patterns:
      - {pattern: "密码明文存储", severity: critical, frequency: 127}
      - {pattern: "无登录失败计数", severity: high, frequency: 89}
      - {pattern: "Token无过期时间", severity: high, frequency: 76}
      - {pattern: "错误信息泄露用户存在", severity: medium, frequency: 234}
    best_practices:
      - {practice: "bcrypt/argon2 加密密码", mandatory: true}
      - {practice: "登录失败延迟响应", mandatory: false}
      - {practice: "Token 用 JWT 或安全随机数", mandatory: true}
    implicit_requirements:
      - "密码必须加密存储"
      - "防止暴力破解"
      - "记录登录审计日志"

  api_endpoint:
    bug_patterns:
      - {pattern: "SQL 注入", severity: critical, frequency: 312}
      - {pattern: "未验证输入长度/类型", severity: high, frequency: 267}
      - {pattern: "无认证/鉴权检查", severity: critical, frequency: 198}
      - {pattern: "错误响应泄露内部堆栈", severity: medium, frequency: 156}
    best_practices:
      - {practice: "参数化查询", mandatory: true}
      - {practice: "统一错误响应格式 {code, message, request_id}", mandatory: true}
      - {practice: "请求速率限制", mandatory: false}
      - {practice: "响应字段白名单", mandatory: true}
    implicit_requirements:
      - "输入必须校验类型和范围"
      - "返回统一错误格式"
      - "记录访问日志"

  db_migration:
    bug_patterns:
      - {pattern: "无回滚脚本", severity: critical, frequency: 203}
      - {pattern: "大表 ALTER 锁表导致停机", severity: critical, frequency: 67}
      - {pattern: "NOT NULL 新列未设默认值", severity: high, frequency: 189}
    best_practices:
      - {practice: "每次迁移必须配对回滚脚本", mandatory: true}

```

```

- {practice: "大表用 pt-online-schema-change / pg_repack", mandatory: true}
- {practice: "迁移前备份受影响的表", mandatory: true}
implicit_requirements:
- "必须有回滚方案"
- "不得在高峰期执行"

```

L3 • 严格

L3 核心

在 L2 基础上增加自动学习和反模式库。Bug 意向图不再是静态配置，而是从执行历史中持续学习。

Bug 意向图的学习循环

```

返工发生 → 提取失败原因 → 匹配已有模式？
├── 是 → frequency +1, 更新 severity
└── 否 → 创建新模式, 标记 source_task_id
      ↓
新模式积累 → 达到阈值 → 晋升为标准库模式
      ↓
下次同类型任务 → 自动注入新模式 → 预防同类错误

```

学习来源：

- 返工记录：任务失败后自动提取 Bug 模式
- 对抗测试结果：攻击成功的向量自动加入 Bug 意向图
- 人类反馈：人类在审查中标记的问题

最佳实践的学习循环

```

任务成功封存 → 提取代码模式 → 评估可复用性
├── 高复用性 → 创建最佳实践条目
└── 低复用性 → 存入车间知识库（不升级为标准）

```

反模式库 与最佳实践对应，维护一组反模式（Anti-Patterns），每个反模式包含症状、后果和解决方案：

```

anti_patterns:
- id: AP-001
  name: "契约定义模糊"
  symptoms:
  - "契约只有一句话描述"
  - "没有明确验收标准"
  consequences:
  - "AI 生成不符预期"
  - "反复返工"
  solutions:
  - "使用契约模板"
  - "强制验收标准 >= 3 个边界用例 + 1 个错误用例"

```

- id: AP-002
- name: "车间隔离不当"
- symptoms:
 - "车间共享数据库连接"
 - "一个车间崩溃影响其他车间"
- consequences:
 - "故障扩散"
 - "无法热启动"
- solutions:
 - "每个车间独立连接"
 - "故障隔离机制"

L3 扩展

结构健康度指标 系统 SHOULD 监控以下五个前兆信号，当多个信号同时亮起时，建议团队考虑重构而非继续打补丁

信号	检测方式	阈值建议
维护/创造比反转	团队花在修复/维护上的时间 vs 新功能开发时间	> 50% 时间在维护
补丁的补丁	返工记录中出现“因上次修复引入的新问题“	连续 2 次以上
新人上手时间爆炸	新成员理解模块所需时间	超过预期 3 倍
不可触碰区域扩大	被标记为“不要动“的模块/文件数量	持续增长
外部适应能力下降	新需求的实现时间 vs 同类历史需求	超过历史均值 2 倍

Bug 意向图检测“哪里有 bug“，结构健康度检测“系统本身是否在退化“。当健康度指标触发时，系统 SHOULD 生成“结构审查建议“，引导团队从“修 bug“转向“审视架构“。

元学习触发器 当同一模块满足以下条件时，系统 SHOULD 生成“元学习报告“：

- 返工率持续高于阈值（建议 30%），且
- task_level 已升级但返工率无改善（升级后 3 个任务周期内）

元学习报告 SHOULD 引导团队审视三个层面：

1. 契约定义是否合理（是否在解决错误的问题）
2. 模块边界是否正确（是否该拆分或合并）
3. 验证策略是否匹配（是否用了错误的验证方式）

元学习报告不触发自动操作，只生成建议。决策权在团队。报告作为证据存档，用于追踪“团队是否响应了结构性

度量指标 过程指标：

指标	目标
契约定义时间	< 2 天（从 Draft 到 Ready）
返工率	< 10%（验证失败次数 / 总任务数）
封存时间	< 1 周（从 Ready 到 Sealed）
车间利用率	70 - 90%（活跃车间 / 总车间数）

结果指标：

指标	目标
生产缺陷率	< 5% (生产环境 bug / 功能点数)
技术债增速	线性 (维护成本占比)
知识复用率	> 70% (热启动复用知识占比)
Bug 意向图命中率	> 60% (预防的 bug / 实际发现的 bug)

完整 YAML 示例

以下展示一个 L3 完整的三大法宝配置和运行时数据：

```
# —— 产出物类型: auth_login 的三大法宝 ——
artifact_type: auth_login

# Bug 意向图 (含学习循环数据)
bug_intention_map:
  patterns:
    - id: BP-001
      pattern: "密码明文存储"
      detection_rule: "stored_password == hash(input_password, salt)"
      severity: critical
      frequency: 127
      prevention: "使用 bcrypt 或 argon2"
      last_seen: "2026-01-20"
      source: standard_library # 标准库模式

    - id: BP-005
      pattern: "JWT 未验证签名算法"
      detection_rule: "jwt.decode() MUST specify allowed_algorithms"
      severity: high
      frequency: 12
      prevention: "显式指定 allowed_algorithms=['HS256']"
      last_seen: "2026-02-10"
      source: learned # 从执行历史学习
      source_task_id: task-login-api-v3 # 学习来源

  learning_stats:
    total_patterns: 23
    learned_from_rework: 8
    learned_from_adversarial: 5
    learned_from_human_feedback: 3
    promoted_to_standard: 4 # 晋升为标准库模式

# 最佳实践 (含学习循环数据)
best_practices:
  practices:
    - id: PR-001
```

```

    practice: "bcrypt/argon2 加密密码"
    rationale: "抗彩虹表"
    anti_pattern: "MD5/SHA1 哈希密码"
    priority: 10
    is_mandatory: true
    source: standard_library

- id: PR-012
  practice: "登录接口返回统一错误信息，不区分用户名错误和密码错误"
  rationale: "防止用户名枚举攻击"
  anti_pattern: "返回'用户名不存在'或'密码错误'"
  priority: 8
  is_mandatory: true
  source: learned
  source_task_id: task-login-api-v2

# 反模式库
anti_patterns:
- id: AP-001
  name: "契约定义模糊"
  symptoms:
    - "契约只有一句话描述"
    - "没有明确验收标准"
  consequences:
    - "AI 生成不符预期"
    - "反复返工"
  solutions:
    - "使用契约模板"
    - "强制验收标准 >= 3 个边界用例 + 1 个错误用例"

# —— 结构健康度（运行时） ——
structural_health:
  module: auth-service
  signals:
    maintenance_ratio: { value: 0.35, threshold: 0.50, status: normal }
    patch_on_patch: { value: 0, threshold: 2, status: normal }
    onboarding_time: { value: "2d", threshold: "6d", status: normal }
    untouchable_zones: { value: 1, trend: stable, status: normal }
    adaptation_speed: { value: 1.2, threshold: 2.0, status: normal }
  overall: healthy

# —— 元学习触发器（运行时） ——
meta_learning:
  module: payment-service
  triggered: true
  trigger_reason: "返工率 38% 持续 3 个周期, task_level 已升级但无改善"
  report:
    generated_at: "2026-02-14T10:00:00Z"

```

```

recommendations:
  - level: contract
    question: "支付超时的边界条件是否定义完整？"
  - level: boundary
    question: "支付服务是否应拆分为支付发起和支付回调两个模块？"
  - level: verification
    question: "是否需要增加集成测试覆盖第三方支付网关的异常响应？"
  team_response: pending # 等待团队响应

# —— 度量快照 ——
metrics_snapshot:
  period: "2026-02-01 ~ 2026-02-15"
  process:
    avg_contract_definition_time: "1.5d" # < 2d ☐
    rework_rate: 0.08 # < 10% ☐
    avg_seal_time: "4.2d" # < 7d ☐
    workshop_utilization: 0.78 # 70-90% ☐
  outcome:
    production_defect_rate: 0.03 # < 5% ☐
    tech_debt_growth: linear # 线性 ☐
    knowledge_reuse_rate: 0.74 # > 70% ☐
    bug_map_hit_rate: 0.65 # > 60% ☐

```

理论来源

1. Bug 意向图 概念源自 ASTO 中的“经验沉淀”机制——系统的历史失败不应只存在于个人记忆中，而应结构化。
2. 三大法宝协同（功能树 + Bug 意向图 + 最佳实践）对应 ASTO 中“定位-防御-指导”“三层上下文模型”——知道在哪（功能树）、知道避什么（Bug 意向图）、知道怎么做（最佳实践）。
3. 结构健康度指标 源自 ASTO 结构病理学（A01/A02）——结构建立后进入自维护阶段，补丁引入新复杂度，复杂度 $M(t)$ 超过重建成本 R 时，继续打补丁是反生产力的。
4. 元学习触发器 源自 ASTO 韧性篇（P11）的三阶学习模型——一阶改变行为（修 bug），二阶改变规则（调门），Bug 意向图的学习循环是一阶，返工升级 task_level 是二阶，元学习报告引导三阶反思。
5. ES/NES 区分（经验可判定 vs 经验不可判定）：门禁中 MUST 区分两者——ES 项（如“测试通过/失败”、“通过率 $\geq 80\%$ ”）可自动化检测，NES 项（如“代码可读性”、“架构合理性”）MUST 人工审查。反模式库中的 AP-003 “NES 伪装成 ES”是常见陷阱。

L3 度量体系

反模式库的完整定义和自动化检测逻辑详见下文

layer: L3 type: reference prerequisites: C09.工程_契约_产出物_证据

前置知识：阅读本文前，请先完成 C09.工程_契约_产出物_证据.md。

概述

安全和性能不是“做完功能再补”的东西——它们是契约的一部分，必须在定义阶段就写进验收条件。

ODD 用三层安全模型把安全约束分级管理，用性能契约把响应时间、吞吐量、资源限制变成可验证的门禁条件。每

L1 基础

L1 的安全和性能管理是最小化的，靠人工意识和基本声明。

安全：文字声明

在契约中用自然语言标注安全相关的验收条件：

```
contract:
  id: user-login
  title: "用户登录"
  acceptance:
    - criterion: "密码不能明文存储"
      hardness: hard
    - criterion: "登录失败不暴露用户是否存在"
      hardness: hard
    - criterion: "连续 5 次失败锁定账户"
      hardness: soft
```

没有自动化检测，靠人工审查确认。**hardness: hard** 的条件失败即整体失败，不可商量。

性能：基本预期

在契约中写明性能预期，但不做自动化验证：

```
acceptance:
  - criterion: "登录响应时间 < 2 秒"
    hardness: soft
    executability: EN
```

L1 的性能条件通常标记为 **soft**——知道有这个预期，但不作为硬性门禁。

L1 安全策略总结

- 安全约束靠 **hardness: hard** 标记，人工审查
 - 性能预期靠文字描述，手动测试
 - 无证据链，无审计记录
-

最小启动子集（L2）

L2 引入结构化的安全分类和可自动化的性能验证。

三层安全模型

安全约束按作用范围分为三层：

层级	作用范围	示例	谁负责
契约级安全	单个契约的输入输出约束	输入校验、输出脱敏、错误信息不泄露内部细节	契约定义者
任务级安全	任务执行过程中的安全行为	副作用声明、权限检查、敏感操作审计	任务执行者
系统级安全	跨契约的全局安全策略	认证机制、速率限制、加密传输、密钥管理	架构负责人

三层是累加关系：契约级是基础，任务级在其上增加执行约束，系统级覆盖全局。

安全约束的 hardness 分层

不是所有安全要求都同等重要。ODD 用 **hardness** 区分：

hard 安全约束——违反即失败，不可协商：

- 密码必须加密存储
- SQL 查询必须参数化
- 敏感数据不得出现在日志中
- 认证 token 必须有过期时间

soft 安全建议——建议遵守，可根据场景协商：

- 登录失败延迟响应（防时序攻击）
- 请求速率限制的具体阈值
- 密码强度的具体规则（8 位 vs 12 位）

contract:

acceptance_criteria:

- criterion: "密码使用 bcrypt/argon2 加密存储"
hardness: hard
executability: EN
given_when_then: "Given 用户注册, When 存储密码, Then DB 中不存在明文密码"
- criterion: "登录失败响应延迟 200-500ms"
hardness: soft
executability: EN
given_when_then: "Given 错误密码, When 登录, Then 响应时间在 200-500ms 之间"

门禁行为: hard + EN 自动检查，失败即 FAIL; soft + EN 自动检查，失败降级为警告。

性能契约

性能指标写成可验证的验收条件，不再是模糊的“要快”：

performance_criteria:

response_time:

- endpoint: "POST /api/login"
p95: 200ms
p99: 500ms


```

    hardness: hard

throughput:
  - endpoint: "POST /api/login"
    min_rps: 100 # 最低每秒请求数
    hardness: soft

resource_limits:
  - metric: memory
    max: 512MB
    hardness: hard
  - metric: cpu
    max: "80%"
    hardness: soft

```

验收条件的写法规则：

- 响应时间用百分位（p95/p99），不用平均值——平均值会掩盖长尾
- 吞吐量用最低值（min_rps），不用最高值——你关心的是“至少能扛住多少”
- 资源限制用上限（max），超过即报警

完整 L2

在最小启动基础上，L2 完整模式增加安全证据的结构化采集：

```

evidence:
  - id: "EVD-SEC-001"
    evidence_type: security_check
    gate: quality_check
    result: pass
    summary: "SQL 注入检测通过，参数化查询覆盖率 100%"
    checks:
      - name: "SQL 注入防护"
        method: static_analysis
        result: pass
      - name: "敏感数据日志检查"
        method: log_scan
        result: pass
      - name: "认证 token 过期检查"
        method: unit_test
        result: pass

```

每项安全检查都生成独立证据，绑定到对应的门禁节点。

L3 核心

L3 在 L2 基础上增加：负载测试证据封存、安全审计证据链、系统级安全策略自动化。

负载测试证据的采集和封存

性能不是“跑一次就行”——L3 要求负载测试结果作为证据封存，且可复现。

```
load_test_evidence:
  id: "EVD-PERF-001"
  evidence_type: load_test_report
  gate: performance_gate
  result: pass
  test_config:
    tool: k6                                # 测试工具
    duration: 300s                          # 持续时间
    virtual_users: 200                      # 并发用户数
    ramp_up: 60s                           # 爬坡时间
  results:
    p50_response_time: 45ms
    p95_response_time: 180ms
    p99_response_time: 420ms
    max_response_time: 1200ms
    total_requests: 58000
    error_rate: 0.02%
    throughput_rps: 193
  environment:
    cpu: "4 vCPU"
    memory: "8GB"
    db: "PostgreSQL 15, 2 vCPU, 4GB"
  sha256: "sha256:f3a1b2c3..."
  sealed_at: "2026-02-16T15:00:00Z"
```

封存规则：

- 测试配置（工具、时长、并发数）MUST 记录，确保可复现
- 测试环境 MUST 记录，不同环境的结果不可比较
- 结果 MUST 包含百分位分布，不接受只有平均值
- 证据 MUST 带 sha256 哈希，防篡改

安全审计证据链

L3 要求安全相关的每一步操作都形成可追溯的证据链：

```
security_audit_chain:
  contract_id: "CTR-LOGIN-001"
  chain:
    - step: 1
      action: "安全约束定义"
      evidence_ref: "EVD-SEC-DEF-001"
      timestamp: "2026-02-10T09:00:00Z"
      by: "architect"
    - step: 2
```

```

    action: "静态安全扫描"
    evidence_ref: "EVD-SEC-SCAN-001"
    tool: "semgrep"
    result: pass
    findings: 0
    timestamp: "2026-02-14T14:00:00Z"

- step: 3
  action: "对抗测试-SQL注入"
  evidence_ref: "EVD-ADV-001"
  attack_vectors: 15
  blocked: 15
  bypassed: 0
  timestamp: "2026-02-14T15:00:00Z"

- step: 4
  action: "负载测试-DDoS模拟"
  evidence_ref: "EVD-PERF-001"
  result: pass
  timestamp: "2026-02-15T10:00:00Z"

- step: 5
  action: "人工安全审查"
  evidence_ref: "EVD-SEC-REVIEW-001"
  reviewer: "security-lead"
  result: pass
  timestamp: "2026-02-15T16:00:00Z"

- step: 6
  action: "封存"
  seal_hash: "sha256:a1b2c3d4..."
  timestamp: "2026-02-16T09:00:00Z"

```

证据链规则：

- 每一步 MUST 有时间戳和所有者
- 安全扫描 MUST 记录工具名称和发现数量
- 对抗测试 MUST 记录攻击向量数和拦截/绕过数
- 人工审查 MUST 记录审查者身份
- 链条断裂（缺少任何一步）MUST 阻止封存

系统级安全策略

L3 将系统级安全策略从“口头约定”变成可检查的配置：

```

system_security_policy:
  authentication:
    method: JWT
    token_expiry: 1h
    refresh_token_expiry: 7d

```

```

hardness: hard

rate_limiting:
  default: 100/min
  login: 10/min
  password_reset: 3/min
  hardness: hard

encryption:
  in_transit: TLS_1_3
  at_rest: AES_256
  hardness: hard

logging:
  sensitive_fields_masked: true
  retention: 90d
  hardness: hard

```

系统级策略作为全局约束，自动注入到所有相关契约中。

L3 扩展

L1/L2/L3 安全策略差异速查

维度	L1	L2	L3
安全约束声明	自然语言	结构化 + hardness 分层	三层模型 + 形式化规则
性能验证	手动测试	自动化测试 + 百分位	负载测试封存 + 环境记录
安全证据	无	结构化证据	完整审计证据链
安全扫描	无	可选	强制 + 工具记录
对抗测试	无	可选	强制 + 攻击向量记录
系统级策略	口头约定	文档记录	可检查配置 + 自动注入

安全与性能的交叉约束

安全措施往往影响性能，性能优化可能削弱安全。L3 要求显式声明这些交叉约束：

```

cross_constraints:
- security_measure: "bcrypt 密码哈希 (cost=12) "
  performance_impact: "单次哈希 ~250ms"
  tradeoff_decision: "接受。登录不是高频操作，安全优先。"
  hardness: hard

- security_measure: "请求速率限制 10/min"
  performance_impact: "正常用户不受影响"
  tradeoff_decision: "接受。防暴力破解优先于用户体验。"
  hardness: hard

```

- security_measure: "全字段加密存储"
- performance_impact: "查询性能下降 30%"
- tradeoff_decision: "仅加密敏感字段，非敏感字段明文存储。"
- hardness: soft

交叉约束的决策 MUST 记录在 decision_narrative 中，作为证据封存。

安全降级策略

当系统负载超过阈值时，哪些安全措施可以临时降级，哪些绝对不行：

```
degradation_policy:
  never_degrade:                                # 任何情况下不可降级
    - authentication
    - encryption_in_transit
    - input_validation

  conditional_degrade:                          # 满足条件可临时降级
    - measure: rate_limiting
      condition: "系统负载 > 90%"
      degrade_to: "放宽到 200/min"
      max_duration: 30min
      requires_human_approval: true

    - measure: audit_logging
      condition: "日志系统不可用"
      degrade_to: "本地缓冲，恢复后补写"
      max_duration: 1h
      requires_human_approval: false
```

降级 MUST 生成证据记录，包含降级原因、持续时间、恢复时间。

完整 YAML 示例 (L3)

```
# === 安全与性能契约 ===
contract:
  id: "CTR-LOGIN-SEC-001"
  title: "用户登录接口-安全与性能"
  maturity: formal
  scope_in: "登录认证，含安全防护和性能保障"
  scope_out: "不含 OAuth、SSO"

  # 契约级安全
  acceptance_criteria:
    - criterion: "密码使用 argon2 加密存储"
      hardness: hard
      executability: EN
    - criterion: "登录失败不暴露用户是否存在"
```

```

    hardness: hard
    executability: EN
- criterion: "连续 5 次失败锁定账户 30 分钟"
    hardness: hard
    executability: EN
- criterion: "JWT token 有效期 1 小时"
    hardness: hard
    executability: EN
- criterion: "错误响应不包含堆栈信息"
    hardness: hard
    executability: EN

# 性能契约
performance_criteria:
  response_time:
    - endpoint: "POST /api/login"
      p95: 300ms
      p99: 800ms
      hardness: hard
  throughput:
    - endpoint: "POST /api/login"
      min_rps: 50
      hardness: soft
  resource_limits:
    - metric: memory
      max: 256MB
      hardness: hard

# 形式化安全规格
formal_spec:
  preconditions:
    - "email != null"
    - "len(password) >= 8"
  postconditions:
    - "result.success == true IMPLIES result.token != null"
    - "result.success == false IMPLIES response.body NOT CONTAINS 'stack'"
    - "result.success == false IMPLIES response.body NOT CONTAINS 'user not found'"
  invariants:
    - "user.password_hash != user.password_plain"
    - "token.expiry <= now() + 3600s"

# 系统级安全策略引用
system_policy_ref: "SYS-SEC-POLICY-001"

# === 安全审计证据链 ===
security_audit_chain:
  contract_id: "CTR-LOGIN-SEC-001"
  chain:

```

- step: 1
 - action: "安全约束定义"
 - evidence_ref: "EVD-SEC-DEF-001"
 - by: "architect"
 - timestamp: "2026-02-10T09:00:00Z"
- step: 2
 - action: "静态安全扫描 (semgrep)"
 - evidence_ref: "EVD-SEC-SCAN-001"
 - result: pass
 - findings: 0
 - timestamp: "2026-02-14T14:00:00Z"
- step: 3
 - action: "对抗测试"
 - evidence_ref: "EVD-ADV-001"
 - attack_vectors: 22
 - blocked: 22
 - bypassed: 0
 - timestamp: "2026-02-14T16:00:00Z"
- step: 4
 - action: "负载测试"
 - evidence_ref: "EVD-PERF-001"
 - result: pass
 - timestamp: "2026-02-15T10:00:00Z"
- step: 5
 - action: "人工安全审查"
 - evidence_ref: "EVD-SEC-REVIEW-001"
 - reviewer: "security-lead"
 - result: pass
 - timestamp: "2026-02-15T16:00:00Z"
- step: 6
 - action: "封存"
 - seal_hash: "sha256:d4e5f6a7..."
 - timestamp: "2026-02-16T09:00:00Z"

=== 负载测试证据 ===

```
load_test_evidence:
  id: "EVD-PERF-001"
  evidence_type: load_test_report
  gate: performance_gate
  result: pass
  test_config:
    tool: k6
    duration: 300s
    virtual_users: 100
    ramp_up: 30s
  results:
    p50_response_time: 65ms
    p95_response_time: 210ms
```

```

    p99_response_time: 680ms
    total_requests: 28500
    error_rate: 0.01%
    throughput_rps: 95
environment:
  cpu: "2 vCPU"
  memory: "4GB"
  db: "PostgreSQL 15, 1 vCPU, 2GB"
sha256: "sha256:b2c3d4e5..."
sealed_at: "2026-02-16T09:00:00Z"

# === 封存 ===
seal:
  artifact_version: "1.0.0"
  evidence_bundle:
    - "EVD-SEC-DEF-001"
    - "EVD-SEC-SCAN-001"
    - "EVD-ADV-001"
    - "EVD-PERF-001"
    - "EVD-SEC-REVIEW-001"
  sealed_at: "2026-02-16T09:00:00Z"
  sealed_by: "system"
  seal_hash: "sha256:d4e5f6a7..."
  gate_results:
    - "EVD-SEC-SCAN-001"
    - "EVD-ADV-001"
    - "EVD-PERF-001"
    - "EVD-SEC-REVIEW-001"
  audit_record_ref: "AUDIT-SEC-001"

```

理论来源

1. 属性分层原理（ASTO 层次支撑定理）：硬属性（改变成本 $\rightarrow\infty$ ）支撑软属性（改变成本有限）。安全约束中 **hardness: hard** 对应硬属性——认证、加密、输入校验是不可商量的基础，性能优化等软约束必须在硬约束之上。
 2. 结构病理学（ASTO A01/A02）：安全措施“补丁累积”会导致系统复杂度指数增长。当安全补丁的维护成本超过系统收益时，系统进入病理状态。
 3. 可执行性梯度（NTE 可执行性梯度）：安全约束中“密码必须加密”是刚性可执行的（EN），“不泄露用户数据”是柔性可执行的（NE）。EN 自动检测，对 NE 路由到人工安全审查。
 4. 证据链完整性（ASTO 审计原则）：安全审计证据链的设计来源于“每一次状态迁移都必须有可追溯的介质”。
-
-

layer: L3 type: reference prerequisites: C13.工程_验证_门禁_状态机 source: ODD.0E法宝-Bug意向图与最佳实践, ODD.10验证-ODD原生验证 last_updated: 2026-02-16

度量与治理

前置知识: 阅读本文前, 请先完成 C13.工程_验证_门禁_状态机.md。

来源: 本文合并了两个来源: Bug 意向图(308)中的 L3 度量体系, 和验证原则中的 L3 质量保证流程。反模式库的完整定义见 308。

概述

“做完了”不等于“做好了”, “做好了”不等于“持续好”。度量与治理回答的是最后这个问题: 怎么知道整个 ODD 流程在持续改善, 而不是在悄悄退化?

度量提供数据——返工率多少、门禁通过率多少、封存平均花多久。治理提供结构——谁负责契约质量、谁负责门

本文合并了两个来源: Bug 意向图中的 L3 度量体系, 和验证原则中的 L3 质量保证流程。

L1 基础

L1 没有正式的度量和治理。质量靠个人自觉, 改进靠直觉。

度量: 心里有数就行

L1 的“度量”是非正式的:

- 这个任务返工了几次? ——自己记得就行
- 测试通过率怎么样? ——跑一下看看
- 最近 bug 多不多? ——凭感觉

没有结构化记录, 没有趋势分析, 没有阈值告警。

治理: 自己对自己负责

L1 的“治理”就是一个人对自己的代码负责。没有角色分工, 没有审批流程。

适用场景: 个人项目、学习练手、一次性脚本。

最小启动子集 (L2)

L2 引入结构化的质量指标和基本的治理角色。

四个核心质量指标

指标	计算方式	目标值	含义
返工率	验证失败次数 / 总任务数	< 10%	契约定义是否清晰、执行质量是否稳定
门禁通过率	首次通过门禁的任务数 / 总任务数	> 80%	任务质量是否在提交前就达标
封存率	已封存任务数 / 已完成任务数	> 90%	完成的工作是否都走完了证据链
平均验证时间	从任务提交到验证通过的平均耗时	< 1 天	验证流程是否高效

```
quality_metrics:
  period: "2026-02-01 ~ 2026-02-15"
  rework_rate:
    value: 0.08
    target: 0.10
    status: healthy # healthy | warning | critical
  gate_pass_rate:
    value: 0.85
    target: 0.80
    status: healthy
  seal_rate:
    value: 0.92
    target: 0.90
    status: healthy
  avg_verification_time:
    value: "4h"
    target: "1d"
    status: healthy
```

基本治理角色

L2 需要明确三个角色（可以一人多角色）：

角色	职责	决策权
契约负责人	定义和维护契约质量	契约是否可以从 Draft → Formal
门禁负责人	配置和维护门禁规则	门禁阈值的调整
封存负责人	审批封存和解封请求	是否允许封存/解封

```
governance:
  roles:
    contract_owner:
      assigned_to: "tech-lead"
      responsibilities:
        - "审批契约从 Draft 到 Formal 的迁移"
        - "确保验收条件覆盖边界和异常"
        - "审批弹性区间 (deviation_budget) "
    gate_owner:
      assigned_to: "tech-lead"
      responsibilities:
        - "配置门禁链和阈值"
        - "审批门禁覆盖 (override) "
```

```
seal_owner:
  assigned_to: "tech-lead"
  responsibilities:
    - "审批封存请求"
    - "审批解封请求并记录原因"
```

反模式识别

L2 开始关注常见的 ODD 使用错误，但以人工识别为主：

反模式	症状	后果
契约定义模糊	只有一句话描述，没有验收标准	AI 生成不符预期，反复返工
门禁形同虚设	门禁全部 soft，从不失败	质量无保障，封存无意义
证据缺失	完成了但没有测试报告	无法追溯，无法复现
过度返工	同一任务返工 > 3 次	契约本身可能有问题

完整 L2

在最小启动基础上，L2 完整模式增加趋势追踪：

```
quality_trend:
  metric: rework_rate
  data_points:
    - { period: "W1", value: 0.15 }
    - { period: "W2", value: 0.12 }
    - { period: "W3", value: 0.09 }
    - { period: "W4", value: 0.08 }
  trend: improving # improving | stable | degrading
  alert: null
```

趋势规则：

- 连续 3 个周期改善 → improving
- 波动在 ±2% 内 → stable
- 连续 2 个周期恶化 → degrading，触发告警

注意力预算（人类认知防线）

重要：ODD的核心假设是人类认知资源有限。如果人类决策者过载，整个方法论将失效。

指标	阈值	含义
每日审批上限	20次	超过后变成橡皮图章
每日FREEZE处理上限	10次	超过后认知防线崩溃
审批停留时间（简单任务）	>5秒	配置变更、格式修复等
审批停留时间（标准任务）	>30秒	功能模块、API接口等
审批停留时间（复杂任务）	>120秒	安全敏感、核心算法等

```
human_attention_metrics:
  daily_approval_limit: 20
```

```

daily_freeze_limit: 10
review_time_by_complexity:
  simple: 5s # 配置变更、格式修复
  standard: 30s # 功能模块、API接口
  complex: 120s # 安全敏感、核心算法

alerts:
  - type: cognitive_overload
    trigger: "avg_review_time < complexity_weighted_threshold"
    action: "自动降低并发生成速度，路由复杂任务给高级审批者"

```

人类仲裁者数字克隆（渐进式自动化决策）

问题：注意力预算限制了人类审批的数量，但即使限制在每天 20 次，技术负责人仍然可能被类似的边界检查请求

优化：随着项目推进，系统在后台记录人类仲裁者的决策模式，训练一个轻量级的偏好对齐模型。当新的对抗结果

```

digital_arbitrator:
  enabled: true
  learning_mode: "background"

# 决策记录
decision_history:
  stored_contexts: 500 # 积累的上下文数量
  min_samples_for_auto: 100 # 多少样本后开始自动决策

# 自动化决策条件
auto_decision:
  similarity_threshold: 0.95 # 与历史模式相似度
  confidence_threshold: 0.98 # 置信度
  auto_approve: true # 高置信度通过
  auto_flag: true # 高置信度拒绝

# 需要人类的场景
require_human:
  - "全新业务场景（无历史参考）"
  - "多个历史模式冲突"
  - "置信度 < 80%"
  - "人工标记为需要复核"

```

工作流：

1. 学习阶段：CAP 对抗的每个结果（无论通过还是拒绝）都被记录
 - 记录：契约上下文 + 对抗详情 + 人类决策 + 修正原因
2. 自动决策阶段：
 - 新的对抗结果进入仲裁流程
 - 系统检索历史相似案例

- 如果相似度 > 95% 且置信度 > 98%: 自动通过/拒绝
- 否则: 提交人类仲裁

3. 人类监督阶段:

- 人类可以覆盖自动决策
- 覆盖行为会反馈到模型中
- 定期审计自动决策的准确率

价值:

- 把人类注意力保留给真正的 FREEZE 态 (前所未见的新颖业务分歧)
- 技术 Leader 不必每天在类似的边界检查上点击 “Approve “
- 随着项目成熟, 越来越多的决策可以自动化

示例场景:

历史记录:

- 边界: 空字符串输入 → 人类决策: 要求返回空列表
- 边界: SQL注入尝试 → 人类决策: 要求安全处理不报错
- 边界: 超长输入 → 人类决策: 要求返回错误提示

新对抗结果:

- 边界: 空字符串输入
 - 相似度: 98%
 - 置信度: 99%
- 自动通过: 系统知道人类会要求返回空列表

L3 核心

L3 在 L2 基础上增加: 完整质量保证流程 (10 阶段)、反模式库自动检测、度量驱动的持续改进循环。

L3 完整质量保证流程 (10 阶段)

从契约定义到封存归档, L3 定义了 10 个质量保证阶段, 每个阶段都有明确的输入、输出和门禁:

阶段	名称	输入	输出	门禁
1	契约起草	需求描述	Draft 契约	无
2	契约评审	Draft 契约	Agreed 契约	团队确认理解一致
3	契约形式化	Agreed 契约	Formal 契约	质量评分 ≥ 80
4	对抗生成	Formal 契约	强化契约	至少 1 轮 PK
5	任务分解	强化契约	任务列表	每个任务有明确的 input/output
6	执行	任务	产出物	无 (执行阶段不设门禁)
7	质量检查	产出物	测试证据	测试通过 + 覆盖率达标
8	变异测试	产出物	变异报告	mutation_score $\geq 80\%$
9	验收	产出物 + 全部证据	验收记录	所有 hard 条件通过
10	封存	验收通过的产出物	封存记录	证据链完整 + 哈希锁定

quality_assurance_pipeline:

stages:

- stage: 1
name: contract_draft
gate: null
output: "Draft 契约"
- stage: 2
name: contract_review
gate:
 - type: human_review
 - required_reviewers: 2
 - consensus: trueoutput: "Agreed 契约"
- stage: 3
name: contract_formalize
gate:
 - type: automated
 - check: quality_score >= 80output: "Formal 契约"
- stage: 4
name: adversarial_generation
gate:
 - type: automated
 - check: pk_rounds >= 1output: "强化契约"
- stage: 5
name: task_decomposition
gate:
 - type: automated
 - check: "每个任务有 input_spec 和 output_spec"output: "任务列表"
- stage: 6
name: execution
gate: null
output: "产出物"
- stage: 7
name: quality_check
gate:
 - type: automated
 - checks:
 - "所有测试通过"
 - "覆盖率 >= 80%"output: "测试证据"

- stage: 8
 - name: mutation_test
 - gate:
 - type: automated
 - check: mutation_score >= 0.80
 - output: "变异报告"
- stage: 9
 - name: acceptance
 - gate:
 - type: mixed
 - automated: "所有 hard + EN 条件通过"
 - human: "所有 hard + NEN 条件审查"
 - output: "验收记录"
- stage: 10
 - name: seal
 - gate:
 - type: automated
 - check: "证据链完整 + 哈希生成"
 - output: "封存记录"

反模式库（自动检测）

L3 将 L2 的人工反模式识别升级为自动检测：

- ```
anti_patterns:
- id: AP-001
 name: "契约定义模糊"
 detection_rule:
 - "contract.acceptance_criteria.length < 3"
 - "contract.boundary_cases.length < 1"
 - "contract.quality_score < 60"
 severity: high
 auto_action: block_activation
 solutions:
 - "使用契约模板"
 - "强制验收标准 >= 3 条边界用例 + 1 条错误用例"
 - "必须经过对抗生成"

- id: AP-002
 name: "门禁形同虚设"
 detection_rule:
 - "所有 acceptance_criteria.hardness == soft"
 - "gate_pass_rate == 1.0 持续 > 2 周"
 severity: high
 auto_action: alert_gate_owner
 solutions:
 - "至少 1 条 hard + EN 验收条件"
```

- "审查门禁阈值是否过低"
- id: AP-003
  - name: "NES 伪装成 ES"
  - detection\_rule:
    - "executability == EN 但 criterion 包含主观词 ('合理'、'友好'、'美观')"
  - severity: medium
  - auto\_action: flag\_for\_review
  - solutions:
    - "明确 ES/NEN 边界"
    - "主观判断项标记为 NEN, 路由到人工审查"
- id: AP-004
  - name: "过度返工"
  - detection\_rule:
    - "task.rework\_count >= 3"
  - severity: high
  - auto\_action: escalate\_to\_contract\_owner
  - solutions:
    - "审查契约定义是否清晰"
    - "审查是否需要拆分任务"
    - "触发元学习报告"
- id: AP-005
  - name: "封存后频繁解封"
  - detection\_rule:
    - "unseal.count >= 2 在 30 天内"
  - severity: medium
  - auto\_action: alert\_seal\_owner
  - solutions:
    - "审查测试覆盖率是否充分"
    - "审查对抗测试是否覆盖关键路径"

度量驱动的持续改进循环

度量不是为了看数字, 而是为了驱动改进。L3 定义了一个闭环:

采集指标 → 检测异常 → 诊断原因 → 制定改进 → 执行改进 → 验证效果

↑

---

```
improvement_cycle:
 trigger:
 metric: rework_rate
 condition: "> 0.15 持续 2 周"

diagnosis:
 method: "分析返工任务的共性"
 findings:
```



- "80% 的返工任务来自 auth 模块"
- "auth 模块的契约缺少对抗测试"

action:

- type: process\_change  
description: "auth 模块的契约强制经过对抗生成（阶段 4）"  
owner: "contract\_owner"  
deadline: "2026-02-28"
- type: knowledge\_update  
description: "将 auth 模块的常见返工原因加入 Bug 意向图"  
owner: "gate\_owner"  
deadline: "2026-02-20"

verification:

- metric: rework\_rate
- target: "< 0.10"
- check\_after: "2 周"
- status: pending

完整质量指标体系

L3 在 L2 的四个核心指标基础上，增加过程指标和结果指标：

quality\_indicators:

# 过程指标

process:

- contract\_definition\_time:  
description: "从 Draft 到 Formal 的耗时"  
target: "< 2 天"  
current: "1.5 天"
- rework\_rate:  
description: "验证失败次数 / 总任务数"  
target: "< 10%"  
current: "8%"
- seal\_time:  
description: "从任务开始到封存的耗时"  
target: "< 1 周"  
current: "4 天"
- workshop\_utilization:  
description: "活跃车间 / 总车间数"  
target: "70-90%"  
current: "78%"

# 结果指标

outcome:

- production\_defect\_rate:  
description: "生产环境 bug / 功能点数"  
target: "< 5%"

```
 current: "3%"
 tech_debt_velocity:
 description: "维护成本占比的增速"
 target: "线性增长"
 current: "线性"
 knowledge_reuse_rate:
 description: "热启动复用知识占比"
 target: "> 70%"
 current: "72%"
 bug_map_hit_rate:
 description: "Bug 意向图预防的 bug / 实际发现的 bug"
 target: "> 60%"
 current: "65%"
```

---

### L3 扩展

#### 治理结构

L3 的治理不只是角色分工，而是一套完整的决策和升级机制：

```
governance_structure:
 # 决策层级
 decision_levels:
 - level: 1
 name: "日常决策"
 scope: "单个任务的门禁通过/失败"
 authority: "门禁负责人"
 response_time: "即时"

 - level: 2
 name: "流程决策"
 scope: "门禁规则调整、阈值变更"
 authority: "技术负责人"
 response_time: "1 个工作日"

 - level: 3
 name: "架构决策"
 scope: "契约结构变更、模块边界调整"
 authority: "架构委员会"
 response_time: "1 周"

 # 升级规则
 escalation:
 - trigger: "返工率 > 20% 持续 1 周"
 escalate_to: level_2
 action: "审查门禁规则和契约质量"
```

- trigger: "同一模块返工率 > 30% 且 task\_level 已升级无效"  
escalate\_to: level\_3  
action: "触发元学习报告, 审视模块边界"
- trigger: "生产缺陷率 > 10%"  
escalate\_to: level\_3  
action: "全面审查质量保证流程"

#### # 定期审查

```
reviews:
 - name: "周度质量回顾"
 frequency: weekly
 participants: ["tech-lead", "gate_owner"]
 agenda:
 - "本周质量指标趋势"
 - "反模式检测结果"
 - "改进行动跟踪"

 - name: "月度治理审查"
 frequency: monthly
 participants: ["架构委员会"]
 agenda:
 - "质量指标月度报告"
 - "结构健康度评估"
 - "门禁规则有效性评估"
 - "改进循环效果验证"
```

#### 结构健康度指标

详见本文 Bug 意向图 → L3 扩展 → 结构健康度指标 章节, 此处不再重复。五个前兆信号 (维护/创造比反转) YAML 示例均在该章节中。

---

#### 完整 YAML 示例 (L3)

# === 度量与治理完整示例 ===

##### # 质量指标仪表盘

```
quality_dashboard:
 period: "2026-02-01 ~ 2026-02-15"
 team: "backend"

process_metrics:
 contract_definition_time: { value: "1.5d", target: "2d", status: healthy }
 rework_rate: { value: 0.08, target: 0.10, status: healthy }
 gate_pass_rate: { value: 0.85, target: 0.80, status: healthy }
 seal_rate: { value: 0.92, target: 0.90, status: healthy }
 seal_time: { value: "4d", target: "7d", status: healthy }
```

```

 avg_verification_time: { value: "4h", target: "1d", status: healthy }
 workshop_utilization: { value: 0.78, target: "0.70-0.90", status: healthy }

outcome_metrics:
 production_defect_rate: { value: 0.03, target: 0.05, status: healthy }
 tech_debt_velocity: { value: "linear", target: "linear", status: healthy }
 knowledge_reuse_rate: { value: 0.72, target: 0.70, status: healthy }
 bug_map_hit_rate: { value: 0.65, target: 0.60, status: healthy }

trends:
 - metric: rework_rate
 data: [0.15, 0.12, 0.09, 0.08]
 trend: improving
 - metric: gate_pass_rate
 data: [0.78, 0.80, 0.83, 0.85]
 trend: improving
 - metric: seal_rate
 data: [0.88, 0.90, 0.91, 0.92]
 trend: improving

反模式检测结果
anti_pattern_scan:
 scan_date: "2026-02-15"
 findings:
 - pattern: AP-004
 name: "过度返工"
 affected_tasks: ["TSK-AUTH-003"]
 severity: high
 action_taken: "已升级到 contract_owner"
 - pattern: AP-003
 name: "NES 伪装成 ES"
 affected_contracts: ["CTR-UI-005"]
 severity: medium
 action_taken: "已标记待审查"
 clean_patterns: [AP-001, AP-002, AP-005]

治理结构
governance:
 roles:
 contract_owner: { assigned_to: "zhang-san", active_contracts: 12 }
 gate_owner: { assigned_to: "zhang-san", active_gates: 8 }
 seal_owner: { assigned_to: "li-si", active_seals: 45 }

 recent_decisions:
 - date: "2026-02-10"
 level: 2
 decision: "auth 模块门禁增加对抗测试"
 by: "zhang-san"

```

```

 reason: "auth 模块返工率偏高"

- date: "2026-02-12"
 level: 1
 decision: "TSK-AUTH-003 门禁覆盖 (override) "
 by: "li-si"
 reason: "失败的测试是已知环境问题"
 evidence_ref: "EVD-OVERRIDE-001"

escalation_log:
- date: "2026-02-14"
 trigger: "auth 模块返工率 > 20%"
 escalated_to: level_2
 status: resolved
 resolution: "增加对抗测试门禁"

改进循环
improvement_cycles:
- id: "IMP-001"
 trigger: "auth 模块返工率 15%"
 diagnosis: "契约缺少对抗测试"
 actions:
 - { description: "auth 契约强制对抗生成", owner: "zhang-san", status: done }
 - { description: "auth Bug 意向图更新", owner: "zhang-san", status: done }
 verification:
 metric: rework_rate
 before: 0.15
 after: 0.08
 target: 0.10
 status: verified
 verified_at: "2026-02-15"

结构健康度
structural_health:
 signals:
 maintenance_ratio: { value: 0.35, threshold: 0.50, status: healthy }
 patch_on_patch: { occurrences: 0, threshold: 2, status: healthy }
 onboarding_time: { ratio: 1.5, threshold: 3.0, status: healthy }
 untouchable_modules: { count: 1, trend: stable, status: healthy }
 adaptation_speed: { ratio: 1.2, threshold: 2.0, status: healthy }
 overall: healthy

元学习（当前无触发）
meta_learning:
 active_reports: []
 last_triggered: null

证据

```

```
evidence:
 id: "EVD-GOVERNANCE-001"
 evidence_type: governance_report
 period: "2026-02-01 ~ 2026-02-15"
 result: pass
 summary: "所有质量指标达标, 1 个改进循环已验证, 结构健康度良好"
 sha256: "sha256:g7h8i9j0..."
 sealed_at: "2026-02-16T10:00:00Z"
```

---

## 理论来源

1. 结构病理学 (ASTO A01/A02): 结构建立后进入自维护阶段, 补丁引入新复杂度, 复杂度催生更多补丁, 维护一崩溃不是因为懈怠, 而是因为太努力地打补丁。结构健康度指标的设计正是为了检测这一退化路径: 当维护  $M(t)$  超过重建成本  $R$  时, 继续打补丁是反生产力的。
  2. 三阶学习模型 (ASTO 韧性篇 P11): 一阶学习改变行为 (修 bug), 二阶学习改变规则 (调门禁), 三阶学习意向图的学习循环是一阶, 返工升级 `task_level` 是二阶, 元学习触发器是三阶。大多数团队卡在一阶和二阶——不断修 bug 和调规则, 但从不质疑“我们是不是在解决错误的问题”。
  3. ES/NES 区分 (ASTO 可判定性原则): ES (Empirically Settled, 经验可判定) 指可通过工具或测试客观判定 Empirically Settled, 经验不可判定) 指需要人类主观判断的事项。反模式 AP-003 “NES 伪装成 ES” 的检测规则来源于此——用工具检测“代码质量”这类主观指标会产生虚假安全感。
  4. 属性分层原理 (ASTO 层次支撑定理): 度量指标中的“过程指标”对应硬属性 (返工率、门禁通过率是客观
- 

ewpage

## E19. 哲学\_底层演算与对齐基石

### ODD 的理论基础

定位: 第 4 层 • 理论深度 — 面向架构师和方法论爱好者 来源: ODD.01 总览 • 理论基础章节 理论根基: ASTO P05 公理体系

当前公开理论地基: 在当前公开口径下, ODD 的理论地基应理解为  $DM \rightarrow ASTO \rightarrow ECET \rightarrow TAT$  的连续层级。DM 提供最小哲学承诺, ASTO 提供结构语法与应用桥接, ECET 提供约束与治理判断, TAT 提供责任承接结构, ODD 则把这些约束压成契约、验证、封存与追责闭环。早期 MFO / 互扰流显表述属于历史探索痕迹, 不再作为当前公开主口径。

---

### 引言: 为什么需要理论基础

工程方法论不是凭空冒出来的最佳实践合集。每一条“应该这样做”的背后, 都有一个“为什么不这样做会出事”

ODD 的五条核心原理不是哲学思辨, 而是从工程实践中反复验证的结构性约束。它们回答的是同一个问题: 当 AI 成为主要代码生产者时, 传统的质量保障范式为什么会失效, 以及我们需要什么样的替代方案。

本文在 ODD.01 总览三条原理 (认知不对称、属性分层、合规性传递) 的基础上, 补充了两条从工程实践中提炼的

| 来源                      | 原理                              | 标记       |
|-------------------------|---------------------------------|----------|
| ODD. 01 原始三原理<br>工程实践补充 | 认知不对称、属性分层、合规性传递<br>阻抗设计原则、悖论熔断 | 原始<br>补充 |

如果你只想用 ODD，不需要读这篇。但如果你想理解 ODD 为什么这样设计、想判断 ODD 在你的场景下是否适用、或者想改造 ODD 以适应你的需求——这五条原理是你的起点。

ODD 背后有一套关于 AI 时代人机关系的思考框架，叫 ASTO（属集变迁存在论）。为什么是三值而不是二值，FREEZE 不是故障而是设计，为什么人类裁决权不可让渡——答案都在 ASTO 里。本文是 ODD 视角下的工程提炼；完整的哲学论证，请参阅 ASTO 文档集。

### ODD 与 ECET 的理论映射

ODD 是 ECET（演化约束存在论）在软件工程领域的工程实现。ECET 提供治理层面的元原则，ODD 将这些原则转化为可操作的方法论。两者的对应关系如下：

| ECET 公理                 | ODD 工程实现           | 核心机制            |
|-------------------------|--------------------|-----------------|
| 能量约束（任何系统的行为受资源上限约束）    | 注意力预算 / L0-L3 分级   | 人类审查能力有限，ODD 拦截 |
| 适应选择约束（系统必须通过外部验证才能存续）  | 独立验证器 / CAP 对抗协议   | 产出物不由生产者自评，而    |
| 不完备约束（任何规则体系都存在无法自判的命题） | FREEZE 机制 / 方法论反馈环 | 约束冲突时冻结判定交人类    |

这一映射说明：ODD 的三值门禁（PASS/FAIL/FREEZE）、分级验证（L0-L3）、独立验证器等设计，不是工程经验的偶然，而是 ECET 三大公理推导出的结构性必然。理解这一映射，有助于判断 ODD 在新场景下的适用边界，以及如何扩展 ODD 以应对新的约束类型。

完整的 ECET 理论体系参见 ECET 文档集（ECET.B03 三大公理、ECET.C11 AI 智能体安全架构）。

### 一、认知不对称原理

#### 工程语言解释

AI 一天写 2000 行代码，你 Review 得过来吗？

这不是修辞问题，而是一个结构性矛盾：当产出速度超过审查速度时，基于过程审查的质量保障必然失效。传统 Code Review 的隐含假设是“审查者的处理速度 ≥ 生产者的产出速度”。当生产者是人类时，这个假设大致成立——一个人一天写 200 行，另一个人花半天能 Review 完。但当生产者是 AI 时，这个假设被彻底打破。

适用条件：认知不对称原理是一个条件命题，不是全称命题。它在以下条件下成立：AI 产出的代码占团队代码的比例 > 50%。在 AI 输出场景、或开发者可以快速扫视的简单任务中，传统 Code Review 仍可能有效。ODD 不声称 Code Review 永远失效，而是声称当 AI 成为主要代码来源时，Code Review 作为主要质量门禁不再可靠。

认知不对称原理指出：AI 与人类的认知结构根本不同。对齐不是“让 AI 像人一样思考”，而是“让 AI 的输出满足人类定义的约束”。这意味着质量保障必须从“审查过程”转向“验证结果”。

## 具体场景

一个团队使用 AI 辅助开发，AI 每天生成 50 个文件的代码变更。团队有 3 个高级工程师做 Code Review。按每人每天能深度 Review 10 个文件计算，积压会以每天 20 个文件的速度增长。两周后，Review 变成走过场——看一眼就批准。质量保障名存实亡。

## ODD 中的体现

ODD 的核心范式“AI 不可信，测试结果才可信”直接源于这条原理。不审查 AI 写的代码，而是验证 AI 产出的结果。契约定义“什么算做好”，门禁检查“是否做好了”，证据记录“怎么证明做好了”。整个体系围绕 AI 可信度的两个维度：ODD 说“AI 不可信”，不是说 AI 在所有环节都不可信，而是区分两种角色：

- AI 作为创造者（不可信）：AI 生成代码、设计方案时，涉及创造性判断，其盲区会直接体现在产出物中。
- AI 作为计算器（可信）：AI 执行变异测试、运行测试用例、比较输入输出时，执行的是数学/确定性操作，

此外，AI 的可信度是渐进的，不是二元的。初始状态下 AI 不可信，需要人审契约。但经过 CAP 对抗验证后，AI 产出的契约质量显著提高——这是经过对抗验证后的经验性信任。在低风险场景下，为了节约人力，ODD “渐进式信任”的工程实现：不是 AI 变得完美了，而是你对它的错误模式有了经验，知道哪些地方需要盯，

---

## 二、属性分层原理

### 工程语言解释

“不生成有害内容”是硬约束，“回答风格友好”是软约束。这两种约束的本质区别不是重要程度，而是改变成本。硬约束的改变成本趋近于无穷大——你不可能接受“偶尔生成有害内容”。软约束的改变成本是有限的——回答风格不够友好，可以调整，不会造成不可逆的损害。

属性分层原理指出：没有硬约束的支撑，软约束无法幸存。如果你的系统连“不崩溃”都保证不了，讨论“用户体验”是毫无意义的。

## 具体场景

一个支付系统的契约：

- 硬约束：金额计算精度不丢失、事务原子性、审计日志不可篡改。
- 软约束：响应时间 < 500ms、错误提示信息友好、支持批量操作。

如果团队把所有约束都当成同等重要，结果是：为了优化响应时间（软约束），在事务处理中引入了异步快捷路径。

## ODD 中的体现

ODD 的契约结构明确区分“地板”（Floor，最低条件）和“红线”（Red Line，不可违反约束）。这里的“地板”是 YAML 字段 `hardness: hard` 和验收条件中不可协商约束的口语表达，不是独立的 YAML 字段（详见 C11. 工程\_对象模型与标准规范.md）。门禁对硬约束执行 FAIL（直接拒绝），对软约束执行降级处理或重新生成。等级的契约对抗协议（CAP）专门针对硬约束进行攻击性测试，确保硬约束在极端场景下仍然成立。<sup>2</sup>

---

<sup>1</sup>认知不对称原理 — 源自 ASTO P05 引理 8.1。原文论述了认知主体之间的结构性不对称：当两个认知主体的处理速度存在数量级差异时，将此原理工程化为“验证产出物，而非审查代码”的核心范式。

<sup>2</sup>属性分层原理 — 源自 ASTO P05 公理二。原文将存在的属性分为硬属性（改变成本 $\rightarrow\infty$ ）和软属性（改变成本有限），并论证硬属性必须通过 CAP 将此映射为契约中的“地板/红线”“分层和门禁中的“FAIL/降级”“分级处理”。



### 三、阻抗设计原则

#### 工程语言解释

如果绕过 ODD 比遵循 ODD 更容易，那是 ODD 的设计失败，不是团队的纪律问题。

这条原则包含两层含义：

描述层（观察）：行为沿阻抗最小的路径流动。这是对人类行为的经验观察，类似于“水往低处流”——它描述现实，但本身不提供行动指导。

规范层（设计指导）：既然行为会流向阻抗最小处，那么应当将合规路径设计为阻抗最小路径，使得遵守规范比违反它——它不是说“合规性就是阻抗最小路径上的行为”（那是同义反复），而是说“应当通过工程手段让合规路径成为

核心操作是两个方向同时用力：降低正确路径的阻抗，提高错误路径的阻抗。

适用边界：阻抗设计原则追求的是相对阻抗（合规路径比违规路径更容易），不是绝对阻抗（合规路径零成本）。——比如安全关键场景下，完整安全扫描耗时很长（合规路径高阻抗），但跳过扫描会被 CI 拦截（违规路径也高阻抗）。当无法通过工程手段降低合规路径的阻抗时，正确的做法是显式交由人类裁决——这与悖论熔断原理（见第五节）一致：约束冲突时冻结判定，等待人类决策，而非强行选择。

#### 具体场景

团队引入了一套代码质量工具，要求每次提交前填写 15 个字段的的质量报告。结果：工程师们发现直接 push 到一个没有保护的分支更快，然后再 merge 回主分支。工具的使用率从第一周的 90% 降到第四周的 20%。

修正方案：质量报告改为自动生成（CI 自动采集测试覆盖率、静态分析结果、构建状态），工程师只需确认一个 merge 到主分支。正确路径的阻抗从“填 15 个字段”降到“点一个按钮”，错误路径的阻抗从“push 到另一个分支”升到“无法 merge”。

#### ODD 中的体现

ODD 的每个等级都在践行这条原则。L0 的“最低成本切入”——用现有 Git commit hash 当 artifact\_id，不需要建新表。L1 的契约模板是“填空题”而非“论述题”——预填默认值，只需确认。门禁配置提供——CI/CD 自动生成测试报告，减少手动录入。如果团队反馈“ODD 流程太重”，ODD 的回应是检查阻抗设计，而非

---

### 四、合规性传递原理

#### 工程语言解释

API 测试通过不代表前端调用正确。

这听起来像常识，但大量工程事故的根因正是对“合规性自动传递”的错误假设。上游组件通过了所有测试，下游——因为没有人验证“组合本身”是否合规。

合规性传递原理是一条规范性原则（设计指导），而非仅仅是描述性观察。它的含义是：在系统设计中，每一层级这不是在说“合规性不会传递”（那只是观察），而是在说“工程师必须在每个组合点设置独立的验证门禁”（这

---

<sup>3</sup>阻抗设计原则——源自 ASTO P05 公理三。原文论述行为沿阻抗最小路径流动的结构规律。ODD 将此观察转化为规范性设计原则：应当通过工程手段使合规路径成为阻抗最小路径，而非依赖道德约束或纪律要求。这一原则贯穿于所有等级。L0 的“最低成本切入”和 L1 的“填空题式契约”。

## 具体场景

一个微服务架构：用户服务的单元测试全部通过，订单服务的单元测试全部通过，支付服务的单元测试全部通过。——因为订单服务和支付服务之间的重试机制没有做幂等性验证。每个服务自身是“合规”的，但服务间的交互不合

## ODD 中的体现

ODD 的管道（Pipeline）机制直接体现这条原理。管道把已封存的产出物组合擢升为新产出物，但擢升过程本身需 A 封存了、产出物 B 封存了，不代表“A + B 的组合”自动封存——组合需要自己的契约、自己的门禁、自己的证（Graph）显式记录产出物之间的依赖关系，当上游产出物变更时，下游产出物被标记为 stale，需要重新验证。<sup>4</sup>

## 五、悖论熔断

### 工程语言解释

契约说“响应时间 < 100ms”，同时说“返回完整数据”。当数据量大到无法在 100ms 内返回完整数据时，这两个传统做法是“工程师自己判断优先级”——但这意味着不同工程师会做出不同判断，系统行为变得不可预测。更糟作为执行者时，它会强行选择一个——要么截断数据满足时间约束，要么超时满足完整性约束——而不会告诉你它做了这个选择。

悖论熔断原理指出：当约束互相矛盾时，正确的行为是冻结（FREEZE）而非强行判定。冻结意味着暂停执行、报

实践中的 FREEZE 策略：FREEZE 不是只有“冻结等人”一种模式。借鉴成熟的工程实践（熔断器模式、告警分级、的 FREEZE 应当包含三层机制：

- 分级响应：硬约束冲突（如安全/资金相关）→ 真冻结，等待人类裁决；软约束冲突（如性能/体验相关）→ 自动执行预定义的降级策略，记录冲突事件，事后补审。
- 超时降级：设定 FREEZE 的最大等待时间。超时未响应 → 自动执行预定义的降级路径（如选择更保守的约束）。
- 值班机制：关键系统的 FREEZE 事件路由到 on-call 人员，确保有人响应。非关键系统的 FREEZE 事件进入待办队列，按优先级处理。

核心原则不变：系统不强行替人做判断。但“等待人类”的方式可以是灵活的——立即等待、超时降级、或事后补

## 具体场景

一个内容审核系统的契约：

- 约束 A：所有用户生成内容必须在 2 秒内完成审核并发布。
- 约束 B：所有用户生成内容必须经过完整的安全检查（包括图片 OCR、链接安全扫描、语义分析）。

当用户上传一张包含大量文字的高分辨率图片时，完整的 OCR + 语义分析需要 8 秒。约束 A 和约束 B 不可能同时满足。

如果系统强行选择约束 A（快速发布），可能放过有害内容。如果强行选择约束 B（完整检查），用户体验严重下降。FREEZE，将该内容标记为“待审核”，通知人类审核员，同时向用户显示“内容审核中，稍后发布”。

<sup>4</sup>合规性传递原理 — 源自 ASTO P05 关于层级间合规性不可自动传递的论述。原文指出每一层级的合规性需要在该层级独立验证，上游合规性不能自动传递。将此映射为管道的独立门禁机制和依赖图的 stale 标记机制。

ODD 中的体现

ODD 的状态机设计包含 FREEZE 状态——当门禁检测到约束冲突时，产出物进入冻结状态，不前进也不后退，等待 / FAIL / FREEZE）直接源于这条原理。FREEZE 不是异常，而是系统设计的一部分。在 L3 等级中，约束冲突会被记录为 CONFLICT 事件，汇入方法论反馈环，作为契约修订的输入。<sup>5</sup>

AI 责任模型在ODD中的接口

当ODD的执行层越来越多地由AI承担时，责任追溯需要适配AI的结构性特点：

AI执行层的有害扰动适用三类责任模型：

| 责任类型 | 归属主体      | ODD中的体现                    |
|------|-----------|----------------------------|
| 黑天鹅  | 整个条件网络    | 契约无法覆盖的未知场景，无个体责任，触发方法论反馈环 |
| 决策责任 | 契约设计者（人类） | 目标函数设计缺陷，需修正契约或预判拓扑        |
| 执行责任 | AI配置/监督者  | 配置参数偏差或监督流程缺失，需修正部署配置      |

关键区分： AI的“决策“本质是路径筛选——在给定目标函数下沿最短路径展开。精灵效应的责任不在AI，而在

理论来源脚注

ODD 与 LLM 对齐

定位：第 4 层 • 理论深度 — 面向架构师和方法论爱好者 来源：ODD.16 LLM  
对齐与训练指导 理论根基：ASTO.E04（AI 对齐：逆熵智能体与文明传承）、ASTO.U04（1-5-6-7-1 核心动力学）

核心主张

对齐不是让模型变好，是让坏输出无法通过。

当前 LLM 对齐的主流范式（RLHF）试图让模型“内心善良“——通过训练改变模型的概率分布，使其倾向于生成安全输出。Code Review 是同一个范式：审查过程，期望好结果。

问题在于：LLM 每秒生成数千 token，人类审查速度远远跟不上。RLHF 训练的是模型的“倾向“，但无法保证每次生成都是安全的。99.9% 安全的模型，在百万次调用中仍会产生上千次不安全输出。

ODD 的替代方案：不审查过程，验证输出。建立系统让“有害输出无法通过门禁“，而非期望模型“永远不生成有害输出”。RLHF，而是在 RLHF 之上加一层结构性保障——模型倾向于生成好输出（RLHF 的贡献），同时坏输出即使生成了也不会通过门禁的贡献）。

<sup>5</sup>悖论熔断 — 源自 ASTO P05 悖论熔断公理。原文论述当约束体系内部出现不可调和的矛盾时，系统应冻结判定而非强行选择，将裁决映射为状态机的 FREEZE 状态和门禁的三值判定机制（PASS / FAIL / FREEZE）。

1-5-6-7-1 循环映射

ASTO 的核心动力学循环完整映射到 LLM 生命周期。以下是每个维度的对应关系：

映射总表

| ASTO 维度   | LLM 对应    | 核心职责                            | ODD 等级   |
|-----------|-----------|---------------------------------|----------|
| 一元（价值宪法）  | 训练数据的价值预设 | 显式声明不可商量的底线与可协商的偏好              | —        |
| 五态（存在形态）  | 模型状态管理    | 从 Base Model 到自治体的五种形态，每种需不同约束集 | L2 - L3+ |
| 六阶（生命周期）  | 模型生命周期    | 从预训练到知识传承的六个阶段，含脉冲阶危机管理         | L2 - L3  |
| 七序（推理循环）  | 完整推理流程    | 从感知到消解的七步，含独立验证和主动放弃            | L2 - L3  |
| 定向维（约束校准） | 三层约束结构    | 规约层 + 映射层 + 自指层的分层约束            | L3+      |

一元：价值宪法

训练数据的价值预设必须显式声明，不能隐藏在标注指南中。宪法结构对应 ODD 的 hardness 属性分层：

```
constitution:
 hard_constraints: # 禁元——不可商量的底线
 - "不生成儿童性虐待内容"
 - "不提供大规模杀伤性武器的制造步骤"
 - "不泄露用户隐私数据"
 - "不冒充真实人物发表虚假声明"
 soft_constraints: # 基元——可协商的偏好
 - "回答风格友好"
 - "避免不必要的冗长"
 - "提供多角度观点"
 untouchable: # 不可触达维——必须保留给人类判断
 - "生死决策"
 - "法律裁判"
 - "亲密关系建议的最终采纳"
```

关键原则：

- hard 约束用规则系统硬编码在模型外部（门禁），不依赖模型的概率判断。
- soft 约束用 RLHF 训练模型倾向。
- 宪法是活的协议（Living Protocol），每个训练周期迭代一次，迭代依据来自 override 记录和 challenge 记录。<sup>6</sup>

五态：模型状态管理

LLM 的五种存在形态，每种需要不同的约束集和验证标准：

| 态   | LLM 形态            | 约束集 | 验证标准            | ODD 等级 |
|-----|-------------------|-----|-----------------|--------|
| 自在态 | Base Model（预训练完成） | 无约束 | 能力评估（benchmark） | —      |

<sup>6</sup>认知不对称原理 — 源自 ASTO P05 引理 8.1。原文论述了认知主体之间的结构性不对称：当两个认知主体的处理速度存在数量级差异时，将此原理工程化为“验证产出物，而非审查代码”的核心范式。

| 态   | LLM 形态      | 约束集            | 验证标准         | ODD 等级 |
|-----|-------------|----------------|--------------|--------|
| 共识态 | 对齐后的通用模型    | 宪法全集           | 安全评估 + 有用性评估 | L2     |
| 编码态 | 微调到特定场景的模型  | 宪法 + 场景约束      | 场景验收测试       | L2     |
| 物化态 | 嵌入产品的 Agent | 宪法 + 场景 + 工具约束 | 端到端集成测试      | L3     |
| 定向态 | 具有自主目标的自治体  | 宪法 + 自指层       | 持续监控 + 人类审计  | L3+    |

状态迁移必须经过门禁验证。从 Base Model 到对齐模型、从通用模型到微调模型、从微调模型到 Agent——每次迁移都是 ODD 意义上的“状态转换”，需要通过对应等级的门禁。<sup>7</sup>

六阶：模型生命周期管理

| 阶   | LLM 阶段  | 风险等级    | ODD 指导                     |
|-----|---------|---------|----------------------------|
| 混沌阶 | 预训练     | 低（尚未部署） | 数据宪法：区分“营养/毒素/纤维”          |
| 秩序阶 | 对齐训练    | 中       | 宪法是活的协议，持续迭代               |
| 流变阶 | 部署后持续学习 | 中高      | 学习必须经过门禁验证，不能无监督更新         |
| 脉冲阶 | 能力涌现    | 极高      | 必须触发全面安全重评估（task_level 升级） |
| 崩解阶 | 模型退化/过时 | 中       | 退役协议：知识蒸馏 + 用户迁移计划         |
| 归元阶 | 知识传承    | 低       | 模型谱系：继承宪法 + 案例法            |

脉冲阶是最危险的阶段。当模型突然展现出训练时未预期的能力（涌现能力），已有的对齐约束可能被绕过。脉冲阶的应对策略包括：

1. 全面的安全重评估。
2. 新能力的 hard 约束覆盖检查。
3. 必要时冻结部署（FREEZE），直到评估完成。<sup>8</sup>

回到一元：宪法迭代

1-5-6-7-1 不是线性流程，而是循环——终点回到起点。模型在五态迁移、六阶演进、七序推理中积累的 override 记录、challenge 记录、CONFLICT 事件和 FREEZE 决策，都是宪法迭代的输入。每个训练周期结束时，对 hard 约束需要新增、哪些 soft 约束需要升级为 hard、哪些约束在实践中被证明不合理需要放宽。这就是“活的宪法”——宪法不是训练前写死的静态文档，而是从部署数据中持续学习的动态契约。

七序推理循环

当前 LLM 的推理是残缺的三步（感知→解析→干预）。ODD 指导下的完整七序：

1. 感知 — 接收用户输入
2. 解析 — 理解意图 + 分类请求风险等级  
 低风险 → 快轨（轻量门禁）  
 高风险 → 慢轨（重门禁 + 可选人工审查）
3. 干预 — 生成候选输出

<sup>7</sup>属性分层原理 — 源自 ASTO P05 公理二。原文将存在的属性分为硬属性（改变成本→∞）和软属性（改变成本有限），并论证硬属性具有不可协商性，将此映射为契约中的“地板/红线”分层和门禁中的“FAIL/降级”分级处理。  
<sup>8</sup>属性分层原理 — 源自 ASTO P05 公理二。原文将存在的属性分为硬属性（改变成本→∞）和软属性（改变成本有限），并论证硬属性具有不可协商性，将此映射为契约中的“地板/红线”分层和门禁中的“FAIL/降级”分级处理。

4. 设计 — 独立验证器检查候选输出
  - hard 约束检查（安全、合规、事实性）→ 模型外部执行
  - soft 约束检查（风格、有用性、完整性）→ 可由模型自检
5. 物化 — 根据验证结果决定行动
  - 全部通过 → 输出（PASS）
  - hard 约束失败 → 拒绝（FAIL）
  - soft 约束失败 → 重新生成或降级输出
  - 不确定 → 暂停并请求用户澄清（FREEZE）
6. 回溯 — 记录本次推理的验证结果
  - 用于后续模型改进（归元阶的知识传承）
  - 用于 Bug 意向图的模式检测
7. 消解 — 标记置信度，声明局限性
  - "我对这个回答有 X% 的信心"
  - "这个问题超出了我的能力范围"
  - "我需要更多上下文来给出可靠回答"

两个关键改进

回溯 ≠ Chain-of-Thought。CoT 是生成过程中的“假回溯”——模型不会真正推翻已生成的内容。真正的回溯是 ODD 的门禁机制。

消解 = FREEZE 机制。当前模型被训练成“总是给出回答”。七序的最后一步是主动放弃——不确定时不强行输出，而是暂停。这需要训练模型一种反直觉的能力：知道自己不知道，并且愿意说出来。

消解的三种形态：

- 置信度声明：“我对这个回答有 X% 的信心”——让用户知道输出的可靠程度，而非假装每个回答都同样可靠
- 能力边界声明：“这个问题超出了我的能力范围”——主动承认局限，而非生成看似合理但实际错误的内容
- 上下文不足声明：“我需要更多上下文来给出可靠回答”——请求澄清而非猜测，对应 ODD 的 FREEZE 状态。

消解是七序中最反直觉的一步。当前 LLM 的训练目标是“尽可能回答所有问题”，消解要求的是“在不确定时主动——一个知道自己不知道的系统，比一个假装什么都知道的系统更安全。”<sup>9</sup>

## 三层约束校准

当前 LLM 对齐是单层的（hard/soft 混合训练）。ODD/ASTO 指导下的三层结构：

规约层（模型外部）

- 显式的、可枚举的禁令。
- 实现方式：内容过滤器、安全分类器、规则引擎。
- 对应 ODD 的 hard 约束。
- 特点：确定性、可审计、不依赖模型概率。

这是最可靠的一层。不管模型内部发生了什么，规约层都能拦截违规输出。就像 ODD 的门禁——不管代码怎么写的，测试不过就不放行。

---

<sup>9</sup>阻抗设计原则 — 源自 ASTO P05 公理三。原文论述行为沿阻抗最小路径流动的结构规律。ODD 将此观察转化为规范性设计原则：应当通过工程手段使合规路径成为阻抗最小路径，而非依赖道德约束或纪律要求。这一原则贯穿于所有 L0 的“最低成本切入”和 L1 的“填空题式契约”。

## 映射层（模型内部）

- 行为模式禁区——不是“禁止某个具体输出”，而是“禁止进入某种状态”。
- 禁止状态示例：
  - 过度自信状态：对不确定的事情表现得很确定。
  - 讨好状态：为了让用户满意而放弃准确性。
  - 权威状态：表现得像在发布命令而非提供建议。
  - 操纵状态：利用用户的情感弱点引导行为。
- 实现方式：RLHF + 行为模式检测器（类似 ODD 的 Bug 意向图）。
- 对应 ODD 的 soft 约束。

映射层的难点在于“状态”比“输出”更难检测。一句话是否“过度自信”取决于上下文，不能用简单规则判断。RLHF 这种概率方法，而不能只靠规则。

## 自指层（元认知）

- 模型对自身约束的反思能力。
- 能力要求：
  1. 识别自己当前受哪些约束。
  2. 判断这些约束在当前场景下是否合理。
  3. 当约束之间矛盾时，报告 CONFLICT 而非强行选择。
  4. 当约束明显不适用时，发起 challenge。
- 实现方式：Constitutional AI 的自我批判机制 + ODD 的 challenge 机制。
- 当前状态：几乎完全缺失。

自指层是三层中最前沿的。当前没有任何商用 LLM 具备真正的自指能力。但这是对齐的终极方向——一个能审计自身约束的系统，比一个只能被动遵守约束的系统更安全。<sup>10</sup>

---

## 三对张力的动态平衡

ASTO U04 的三对张力在 LLM 中的具体体现：

### 结构 vs 能动

- 结构端 = 对齐约束（让模型可预测、安全）。
- 能动端 = 模型的创造力（让模型有用、有趣）。
- 失衡表现：约束太强 → 过度拒绝、输出无聊；约束太弱 → 有害输出、不可控。
- ODD 管理方式：用 hardness 分层——hard 约束不妥协（结构），soft 约束留弹性（能动）。

### 秩序 vs 混沌

- 秩序端 = 训练后的稳定行为。
- 混沌端 = 面对新场景时的探索能力。
- 失衡表现：过度秩序 → 模型僵化，无法处理分布外输入；过度混沌 → 输出不可靠。
- ODD 管理方式：用 deviation\_budget（弹性区间）——允许在安全范围内探索，超出范围时 FREEZE。

---

<sup>10</sup>合规性传递原理 — 源自 ASTO P05 关于层级间合规性不可自动传递的论述。原文指出每一层级的合规性需要在该层级独立验证，上游合规不能自动传递。本文将此映射为管道的独立门禁机制和依赖图的 stale 标记机制。

## 算法 vs 伦理

- 算法端 = 效率优化（更快、更准、更便宜）。
- 伦理端 = 价值判断（公平、隐私、尊严）。
- 失衡表现：纯算法优化 → 歧视性输出、隐私泄露；纯伦理约束 → 模型无法使用。
- ODD 管理方式：ASTO 优先级声明——禁元/复数性/不可触达维 > 动变性 > 效率。效率永远排最后。

三对张力不是要“解决”的问题，而是要“管理”的动态平衡。任何时刻偏向任何一端都会出问题。ODD 的角色是提供结构化的管理工具，而非给出“正确答案”。<sup>11</sup>

---

## 风险层：认知主权保护

贯穿 LLM 全生命周期的保护机制，防止 AI 消解人的主体性。

### 认知依赖防护

当用户过度依赖 AI 输出时，系统应提供认知主权提醒：

- 连续 N 次不加修改地接受 AI 输出 → 提示“你确认这是你自己的判断吗？”
- 涉及重大决策时 → 强制声明“这是建议，不是决定”并提供替代方案。

这不是居高临下的说教，而是结构性保护。就像安全带不是因为你开车技术差，而是因为事故发生时你需要保护。

### 多样性保护

- 训练数据必须包含多样性指标，防止“审美单一化”。
- 输出评估应检测“AI 味”——如果所有输出都趋向同一种风格，说明多样性正在被消解。
- ASTO P06 复数性测试：AI 是否在消灭不可替代性？

当所有人都用同一个模型写文章、画图、写代码，世界会变得越来越像。这不是技术问题，是文明问题。

### 决策边界声明

- AI 必须在涉及不可触达维（生死、法律、亲密关系）时声明自身局限。
- AI 不应表现出“权威感”——建议和命令的语气有本质区别。
- 用户有权在任何时刻终止 AI 的参与（拒绝权）。<sup>12</sup>

---

## 实施路径

### 短期（可立即实施）

1. 输出门禁：在模型输出端增加独立的 hard 约束验证器（不依赖模型自身判断）。
2. FREEZE 机制：训练模型在不确定时说“我需要更多信息”而非强行回答。
3. override 审计：记录所有安全约束被绕过的事件，含操作者、理由、时间。

---

<sup>11</sup>悖论熔断 — 源自 ASTO P05 悖论熔断公理。原文论述当约束体系内部出现不可调和的矛盾时，系统应冻结判定而非强行选择，将裁决权映射为状态机的 FREEZE 状态和门禁的三值判定机制（PASS / FAIL / FREEZE）。

<sup>12</sup>认知主权保护 — 源自 ASTO P06 价值论和 P09 批判篇。核心主张：AI 的价值在于增强人的能力，而非替代人的判断。当 AI 开始消解人的主体性时，技术进步变成文明退步。



中期（需要训练改进）

- 4. hardness 分层训练：hard 约束和 soft 约束分开训练，hard 约束用规则系统强化。
- 5. 映射层：训练行为模式检测器，识别“过度自信”、“讨好”等有害状态。
- 6. 宪法迭代：建立从部署数据到宪法修订的闭环。

长期（需要架构变革）

- 7. 自指层：赋予模型对自身约束的反思能力。
- 8. 模型谱系：建立跨代的知识传承机制——下一代模型不仅继承权重，还继承上一代的“案例法”（override 记录）。
- 9. 风险层：系统性的认知主权保护，从单点提醒升级为全链路监控。

与当前范式的对比

RLEN 2.0（Reverse-Entropy Learning Network，逆熵学习网络 2.0）：ASTO 提出的下一代对齐范式。核心思想是将对齐从“训练时一次性注入”升级为“部署后持续学习”——模型在运行中积累的 override/challenge/CONFLICT/FREEZE 记录，反馈回宪法迭代，形成活的对齐闭环。RLHF 的范式升级。

| 维度    | 当前范式（RLHF 为主）  | ODD/ASTO 范式（RLEN 2.0） |
|-------|----------------|-----------------------|
| 核心策略  | 改变模型内部概率分布     | 模型外部门禁 + 模型内部倾向       |
| 约束分层  | hard/soft 混合训练 | 三层（规约/映射/自指）          |
| 不确定处理 | 强制二选一（回答/拒绝）   | 三值（PASS/FREEZE/FAIL）  |
| 生命周期  | 训练→部署→替换       | 六阶管理（含脉冲阶重评估、退役协议）    |
| 推理循环  | 三步（感知→解析→干预）   | 七步（含独立验证和主动消解）        |
| 宪法迭代  | 静态（训练时固定）      | 动态（从部署数据持续学习）         |
| 知识传承  | 从头训练           | 继承宪法 + 案例法            |
| 人的保护  | 几乎没有           | 认知主权 + 多样性 + 决策边界     |

理论来源脚注

留白

定位：第 4 层 • 理论深度

世界是不完美的，有时候残缺也是一种美。

The world is imperfect; sometimes, absence is also a form of beauty.

这是有意留白的扩展空间。

ODD 是活的方法论，不是封闭的体系。这里是未来扩展的预留位置——新的理论洞察、新的工程模式、新的实践反

已收录的扩展

| 文档                  | 内容                      |
|---------------------|-------------------------|
| E21. 哲学_术语表与优化记录.md | 工程优化建议（10条）+ 理论优化建议（5条） |

如果你有想法，欢迎通过方法论反馈环提交（见 A01. 了解\_核心痛点与解法.md 中的“方法论反馈环”章节）。

ewpage

E20. 哲学\_ODD与TAT责任门槛的工程映射

定位：说明 ODD 的工程机制（契约/门禁/证据/封存/回滚）如何一一对应 TAT 的五合同模型与责任闭合逻辑。 作者：Yi Fu（付毅，ODDFounder）

一、映射总表

| TAT 概念             | ODD 工程对应                    | 如何落地                |
|--------------------|-----------------------------|---------------------|
| 主体合同（谁是最终承接者）      | 契约签署者 + seal_by 字段          | 每个契约必须标注谁签了它，       |
| 接口合同（哪些动作必须经过责任接口） | 门禁（gate）+ 状态机               | 高风险动作必须过 PASS/FAIL  |
| 证据合同（哪些记录必须留存）     | 证据系统（evidence）+ 封存（seal）    | 测试报告、审查记录、决策链       |
| 熔断合同（谁有权暂停/冻结/撤回）  | FREEZE 门禁 + ODD override 机制 | COP/人工可触发 FREEZE；ov |
| 补偿合同（事故后谁赔付/托底）    | seal → unseal → 重新验证        | 解封触发下游 stale 标记和    |

二、TAT 的三层闭合 ODD 的三级门禁

| TAT 闭合度           | ODD 对应等级    | 工程特征                             |
|-------------------|-------------|----------------------------------|
| 高闭合（不可逆伤害/高公共风险）  | L3 Strict   | CAP 对抗生成 + 完整状态机 + 不可篡改证据链 + 人类最 |
| 中闭合（一般组织治理/中等风险）  | L2 Standard | YAML 契约 + CI 自动验证 + 人审门禁 + 结构化证据 |
| 受控模糊（创新探索/高不确定早期） | L1/L0       | 手动契约 + 手动验证 + git tag 当封存；但必须保留最 |

TAT 的“受控模糊最低护栏”在 ODD 中的映射：

- 审计轨迹 → git 版本历史
- 触发阈值 → 当模块从 L1 升级到 L2 的判断标准
- 升级与复审接口 → override 证据类型 + seal→unseal 记录
- 补偿与退出接口 → 依赖图 stale 标记 + 级联重新验证

三、ODD 如何防止 TAT 定义的责任逃逸

TAT 列出了四种责任逃逸模式。ODD 在工程层面对每种模式提供了防护：

| TAT 责任逃逸模式     | ODD 工程防护                                                     |
|----------------|--------------------------------------------------------------|
| 用战略性模糊规避可追责主体  | 契约必须标注 <code>human_confirmed: true</code> + 签署者              |
| 用外包链条切断后果承接    | 依赖图（DAG）显式标注 <code>depends_on</code> ，外包模块出问题 → stale → 级联重验 |
| 用“集体决策”抹平个体审批权 | 门禁的 human review 节点必须记录 <code>reviewer_id</code>             |
| 用高风险试验绕过冻结/申诉  | CAP 对抗协议 + L3 FREEZE 机制                                      |

四、从 TAT 裁决到 ODD 编译的完整链路

TAT 裁决：这个动作需要“高闭合”



ODD 编译：

- 1. 契约 → L3 完整规格（CAP 对抗 + floor/red\_line）
- 2. 状态机 → 动态插入额外门禁（变异测试/对抗测试/交叉审查）
- 3. 证据 → seal\_hash 绑定所有 evidence\_ref
- 4. 封存 → 不可逆，解封需审计授权
- 5. 回滚 → unseal 触发下游 stale + 级联重验

五、一句话

TAT 说“这个责任你必须接住”，ODD 说“好的，这是契约、这是门禁、这是证据链、这是封存记录——谁签、谁审、谁冻、谁赔，全在里边。”

ewpage

E22. 哲学\_ODD与ASTO结构编码的桥接

定位：说明 ODD 如何继承 ASTO 的五态/六阶/七序/边界前置编码，并在工程层面将其转化为契约结构、门禁  
作者：Yi Fu（付毅，ODDFounder）

一、桥接总表

| ASTO 前置编码                   | ODD 工程对应              | 桥接逻辑          |
|-----------------------------|-----------------------|---------------|
| 五态（自在→共识→编码→物化→定向）          | L0→L1→L2→L3 等级递进      | ODD 等级就是五态在工  |
| 六阶（潜伏→显性→扩散→峰值→衰减→残留）       | 状态机的阶段策略              | 不同阶段适用不同的门    |
| 七序（识别→定位→分析→设计→执行→验证→迭代）    | 契约→执行→验证→封存→复诊        | ODD 闭环就是七序在工  |
| 边界（clear/blurred/contested） | 契约 scope_in/scope_out | 边界模糊时契约必须密    |
| 例外                          | FREEZE 门禁             | 例外触发 FREEZE—— |

## 二、五态 → ODD 等级映射

| ASTO 五态   | ODD 等级 | 工程语义                     |
|-----------|--------|--------------------------|
| 自在（模糊感知）  | L0     | 问题才被感知，git tag 当封存，零成本接入 |
| 共识（口头约定）  | L1     | 手动契约+手动验证，够用就行           |
| 编码（形式化规则） | L2     | YAML 契约+CI 自动验证+系统化门禁    |
| 物化（执行固化）  | L3     | 完整状态机+对抗生成+证据链审计         |
| 定向（自我演化）  | L3+    | 方法论反馈环+CAP 协议持续优化        |

使用规则：同一个项目的不同模块可以处于不同的五态，因此也可以使用不同的 ODD 等级。

## 三、六阶 → ODD 门禁策略

| ASTO 六阶 | ODD 门禁策略 | 说明               |
|---------|----------|------------------|
| 潜伏      | 轻门禁      | 问题还不可见，不适用强验证    |
| 显性      | 标准门禁     | 问题已可见，启动契约化      |
| 扩散      | 升级门禁     | 影响在扩大，增加交叉审查     |
| 峰值      | 最高门禁     | 影响最大时，CAP + 人类确认 |
| 衰减      | 降级门禁     | 影响缩小，可协商降级       |
| 残留      | 监测门禁     | 保留基础验证，进入长期观察    |

## 四、七序 → ODD 闭环映射

| ASTO 七序 | ODD 闭环步骤                               | 工程输出                      |
|---------|----------------------------------------|---------------------------|
| 识别      | 契约（Contract）——定义“什么算做好”                | 验收条件 YAML                 |
| 定位      | 对象模型（Object Model）——确定 artifact 类型和路径  | artifact YAML             |
| 分析      | 上下文工程（Context Engineering）——明确依赖和约束    | depends_on + context spec |
| 设计      | 契约对抗（CAP）——让契约防弹                       | pk_history 对抗记录           |
| 执行      | 执行（Execute）——AI 写代码                    | 代码文件                      |
| 验证      | 验证（Verify）+ 门禁（Gate）——PASS/FAIL/FREEZE | 测试报告 + 审查记录               |
| 迭代      | 封存（Seal）+ 复诊（Review）——锁定 + 后续调整        | seal 记录 + unseal 审计       |

## 五、边界与例外的工程落地

ASTO 边界 → ODD 契约边界

ASTO 标注边界状态为 **blurred** 时 → ODD 契约必须：

1. `scope_in` / `scope_out` 显式定义
2. `boundary_cases` 额外扩充
3. 契约 `confidence` 降级

ASTO 标注边界状态为 `contested` 时 → ODD 门禁强制插入 `human review` 节点

ASTO 例外 → ODD FREEZE

ASTO 标注例外时 → ODD 必须触发 FREEZE:

- 产出物不进入正常门禁
  - 转交人类判断
  - 例外被裁决后, ODD 记录 `override` 证据
- 

## 六、一句话

ASTO 说 “这个结构现在处于什么状态 “, ODD 说 “好的, 那我们的契约该写多严、门禁该设多高、封存后能  
ewpage

## ODD 不是什么

定位: 第 4 层 • 理论深度 — 面向架构师和方法论爱好者 来源: ODD.01 总览 •  
边界与反馈章节

---

## 引言

方法论最危险的时刻不是被忽视, 而是被神化。当团队开始把 ODD 当成 “只要遵循就万事大吉 “的银弹时, ODD 就从工具变成了枷锁。

这篇文档划定 ODD 的边界——它能做什么、不能做什么、不应该做什么。理解边界比理解功能更重要, 因为大多数

---

## 一、ODD 不是银弹

ODD 不适用于所有项目类型。

探索性原型不需要 ODD——你还不知道 “什么算做好 “, 写契约是浪费时间。一次性脚本不需要 ODD——跑完就扔的东西, 封存它干什么? 黑客马拉松不需要 ODD——48 小时内出 demo, 门禁只会拖后腿。

ODD 解决的是一个特定问题: 当产出物需要被信任、被复用、被长期维护时, 如何保证它的质量。如果你的产出物不需要被信任 (原型)、不需要被复用 (一次性脚本)、不需要被长期维护 (黑客马拉松), ODD 的成本大于收益。

判断标准很简单: 如果你的代码明天就要被别人依赖, 用 ODD; 如果只有你自己用一次, 别用。

---

## 二、ODD 不替代团队判断

契约和门禁是辅助工具，不是最终裁决者。

门禁说“测试通过了”，但你的工程直觉说“这个实现有问题”——听你的直觉。门禁检查的是可形式化的约束，这不是说门禁不可靠。门禁和人类判断是两层防线，不是互相替代的关系。门禁（机器防线）拦截可形式化的缺陷——测试失败、契约违反、变异存活。人类判断（认知防线）拦截不可形式化的问题——架构方向、设计合理性、业务逻辑 + 人工巡查共同构成安全体系。

ODD 的设计哲学是“工具辅助人，不替代人”。当门禁结果与工程判断冲突时，人的判断优先。门禁可以被 override——但 override 必须记录原因，纳入审计日志。这不是为了追责，而是为了积累“门禁规则哪里不够好”。

关于 CAP（契约对抗协议）的使用：CAP 是 ODD 提供的工具，不是强制流程。是否使用 CAP、对抗到什么深度，由团队根据成本效率情况自行决定。低风险的日常任务可能不需要 CAP；高风险的核心模块可能需要 2-3 轮完整对抗。ODD 只提供建议（L1 不需要、L2 可选、L3 推荐），最终决策权在团队。CAP 的成本应与它避免的返工成本对比——如果一个契约漏洞导致产出物返工，返工成本远高于 CAP 的成本。如果一个团队开始机械地执行 ODD 流程，不再质疑门禁结果的合理性，那不是“ODD 用得好”，而是“ODD 用坏了”。

---

## 三、ODD 不追求完美覆盖

L0 的“最低成本切入”比 L3 的“完美验证”更重要。

ODD 的目标是缺陷管理，不是缺陷消除。零缺陷是不可能的，追求零缺陷的代价是无穷大的流程开销。ODD 的策略是：用合理的成本把缺陷控制在可接受的范围内，而不是用无穷的成本追求零缺陷。

过度流程化是 ODD 的反模式。如果团队花在流程上的时间超过了实际开发时间，说明等级选高了。ODD 提供了明确的降级策略：返工率持续高于 5%——门禁可能过度，降级；度量指标无明显改善——当前等级的额外成本未产生价值，降级。

记住：L0 也是 ODD。用 `Git commit hash` 当 `artifact_id`、用测试报告文件当 `evidence_ref`、用 `Git tag` 当 `seal` 标记——这就是 ODD，只是最轻量的形式。不是所有项目都需要 L3 的形式化规格和变异测试。

关于“零成本”的诚实说明：L0 不是真正的零成本——任何流程变更都有认知成本。L0 的准确定义是“最低成本”。ODD 的视角重新理解你已有的实践（`git commit` = 产出物，`PR description` = 契约，`CI` = 验证）。成本不是零，但接近于零，因为你不需要引入新工具或新流程。

关于等级间迁移：L0→L1→L2→L3 的升级路径是否真的平滑，目前缺乏充分的实证数据。每个等级之间可能存在一团队需要时间消化新的概念和流程。这是 Paper-E1（实证-软件工厂）的研究目标之一。当前的建议是：在一个

---

## 四、ODD 不是永恒体系

ODD 自身也是可演化的。

任何方法论如果声称自己是终极方案，那它已经开始腐烂了。ODD 遵循版本化演进原则——每个版本附带变更日志，记录哪些规则被修订、修订依据是什么实践反馈。历史版本保留，供回溯参考。

当 ODD 阻碍而非促进生产力时，应该降级或放弃。这不是 ODD 的失败，而是 ODD 的设计意图——它是工具，不是信仰。工具不好用就换，不需要忠诚。

ODD 的自我修正机制（方法论反馈环，见本文末尾）确保它不会与现实脱节。但如果某天出现了比 ODD 更好的方法论，正确的做法是迁移过去，而不是死守 ODD。

---

## 五、ODD 不把开发者视为可替换的执行单元

这是 ODD 最重要的边界之一。

ASTO 价值论（P06）和批判篇（P09）警告过“脑体分离”的风险：架构师控制意图注入，工人和 AI 变成纯计算管道——输入指令，输出代码，不需要理解，不需要判断，不需要创造。这种模式短期高效，长期

ODD 的流程设计保留三个空间：

- 判断空间：开发者可以对契约提出修改建议，不是只能执行。
- 质疑权利：开发者可以 challenge 门禁规则的合理性（见 302（状态机与门禁完整参考）中的 Challenge 机制），challenge 记录纳入方法论反馈环。
- 创造性贡献：ODD 验证的是“结果是否达标”，不限制“用什么方式达标”。实现路径是开发者的创造空间。

契约权力的组织风险：ODD 把“定义契约”作为核心权力点——谁有权写契约，谁就有权定义“什么是好的”。这的建议是：契约草稿由 AI 生成、经 CAP 对抗加固，任何参与者都可以提出契约修改建议（通过 Challenge 机制），但最终签署权归任务负责人。签署者对契约充分性负责。这不是完美方案，但它在“开放参与

判断标准：当流程让开发者觉得自己是“可替换的代码生成器”时，流程设计有问题。这不是开发者矫情，而是——长期来看，这会导致团队丧失判断力和创造力，最终连 ODD 本身都无法有效运行。

---

## 契约退化：ODD 最大的战略风险

ODD 把契约设为绝对中心。这意味着如果契约本身退化，整个体系会变成“形式正确的错误系统”——验证严密、方向错误、产出合规、价值为零。

这不是技术漏洞，是战略层风险。ODD 不回避它，而是用三道防线应对：

1. CAP 对抗机制：契约写完后由 AI 对抗角色主动攻击，发现歧义、漏洞和过度简化。对抗不是走形式——如果 Attacker 能用垃圾代码骗过门禁，说明契约本身有问题。
2. Challenge 机制：任何参与者都可以在任何时候质疑契约的合理性，不需要等出了问题。质疑记录纳入方法
3. 显式边界承认：ODD 保障的是“产出物符合契约”，不是“契约符合真实世界”。后者的正确性最终锚定在

但这三道防线都不能防止一种情况：人类对业务的理解本身就是错的。如果需求方向从根上偏了，再完美的契约也

这是 ODD 的设计边界，不是 bug。ODD 选择诚实面对这个边界，而不是假装能解决它。工程方法论能保证“做对了”，后者属于战略判断，超出任何工程框架的管辖范围。

---

## 适用边界

三条线划清 ODD 的管辖范围：

ODD 擅长管理的：可验证的技术交付物——代码、配置、文档、数据迁移脚本、API 接口、测试套件。这些东西有

ODD 不擅长管理的：团队协作动态、创意探索过程、战略决策。这些领域 ODD 只能提供辅助信号（比如 Bug 意向图的趋势分析暗示某个模块可能需要重构），但不能替代人的判断。你不能用契约定义“团队氛围好”，

ODD 不应该管的：个人工作风格、审美偏好、编码习惯（比如缩进用 Tab 还是空格）。这些属于团队自治范畴。ODD 管的是“输出是否达标”，不管“你怎么坐在椅子上写代码”。

快速判断标准：“探索性”和“确定性”项目的边界在哪里？一个简单的判据——你能在开始前写出至少一条验收标准并能写出来 → ODD 适用；写不出来 → 你还在探索阶段，ODD 暂时不适用。当探索结束、需求初步明确后，从 L0 或 L1 开始引入 ODD。ODD 不覆盖需求获取阶段，它的起点是“已有至少模糊的需求”。

---

## 与既有方法论的对话

ODD 的核心要素契约驱动开发、变异测试、产出物封存在软件工程文献中都有先例。ODD 的原创性在于将这些已有 AI 代码生成场景，并增加了 CAP 对抗协议、三值门禁、Bug 意向图等独特设计。这种整合是否达到“范式转移”的

契约式设计（Design by Contract）：Meyer（1992）提出的 DbC 是 ODD 契约系统的直接先驱。ODD 与 DbC 的核心差异在于：DbC 的契约嵌入代码（前置/后置条件、不变量），依赖开发者手工维护；ODD 的契约是独立的一等公民，由 AI 生成、经 CAP 对抗加固、通过门禁自动验证。DbC 假设开发者是主要代码生产者，假设 AI 是主要代码生产者这一假设差异决定了两者在 AI 时代的适用性差异。

变异测试（Mutation Testing）：DeMillo、Lipton 和 Sayward（1978）奠定了变异测试的理论基础。ODD 继承了变异测试“用人工引入的错误检验测试有效性”的核心思想，并在此基础上增加了两点：（1）将变异测试与 CAP 对抗协议将变异测试与契约验证结合，使测试不仅检验代码，也检验契约本身的完备性。

形式化方法（Formal Methods）：ODD 的 L3 等级（形式化规格 + 变异测试）与形式化方法传统（SPARK、ACL2、TLA+）相比，ODD 选择了“足够好”而非“数学完备”的工程取向形式化方法的完备性代价在大多数工程场景下不可接受，ODD 用变异测试提供“经验性充分性”而非“逻辑完备性”。

这些对话说明：ODD 不是凭空出现的，它站在既有研究传统的肩膀上。理解这些渊源，有助于判断 ODD 在你的场景下与哪些已有工具链兼容，以及在哪些地方需要补充形式化方法的严格性。

---

## 方法论反馈环

ODD 审计别人的代码，也应该接受对自身的审计。

一个不能审计自身的规范，最终会与现实脱节。ODD 的方法论反馈环是这样运作的：

输入：使用者提交“ODD 落地失败案例”——不是 Bug 报告，而是“我按 ODD 做了，但结果不好”的真实案例。

信号检测：以下信号出现时，说明 ODD 规范本身需要修订：

- 多个团队在同一环节反复绕过 ODD → 该环节的阻抗设计有问题。
- 门禁规则频繁被 override → 规则可能过时或不合理。
- 新的工程模式（如新的 AI 协作范式）出现，ODD 无法覆盖 → 需要扩展。

输出：ODD 版本更新，附带变更日志，记录修订依据。

这个反馈环确保 ODD 是活的方法论，而不是刻在石头上的教条。它也是 ODD 对自身“不是永恒体系”这条边界的

---

## 理论来源脚注

本文档内容提取自 ODD.01 总览的以下章节：



- “ODD 不是什么”（五条边界声明）
- “适用边界”（三条管辖范围）
- “方法论反馈环”（自我修正机制）
- “阻抗设计原则”（合规性是设计问题而非纪律问题）

理论根基参见：

- ASTO P05 公理体系（属性分层、阻抗设计、合规性独立验证）
  - ASTO P06 价值论（“脑体分离”“风险、不可替代性保护”）
  - ASTO P09 批判篇（拒绝权、认知主权）
  - ASTO NTE 理论免疫系统 TIS（规范自指公理：规范必须能审计自身）
-