

衔尾蛇 - 自指迭代收敛算法（SICA）：可安全自进化智能体的设计框架

廖嘉轩，纪辉泉，王进，蔡瑞泉

这是一个触及智能系统本质的深刻问题 —— 其核心是：如何设计一个能够安全修改自身「世界模型 + 思维范式 + 行为策略」的智能体，确保迭代过程不会陷入混乱发散，最终收敛至稳定、高效的最优状态。

该问题直面自指系统的三大经典困境：哥德尔不完备性、图灵停机问题与自举悖论。但在明确划定系统边界的前提下，我们可以通过严格的数学约束，让自指迭代过程实现可控收敛。本文给出完整的理论框架、算法规则与工程落地方案。

一、问题的形式化定义

1.1 智能体核心组件

设智能体在时刻 t 包含三大可迭代修改的核心组件，各组件定义如下：

组件层级	符号	核心含义	递归修改的对象
理解层	M_t	对环境的状态表征、对象关系建模、物理直觉与观测编码	世界模型 / 观测编码逻辑
归纳层	H_t	从历史经验中提炼的规则、假设、模式与抽象规律	搜索偏置 / 剪枝规则 / 抽象归纳逻辑
决策层	π_t	动作选择策略、资源预算分配、探索与利用倾向	搜索算法参数 / 行为路由逻辑

环境 E 向智能体提供客观反馈 $r_t \in R$ ，例如关卡通关率、路径步数效率、计数器数值变化等可量化指标。

1.2 递归自指修改机制

自指修改的本质是三层组件形成闭环迭代，每一层的输出都会成为下一层修改自身的输入，迭代公式为：

$$M_{t+1} = f_M(M_t, H_t, \pi_t, r_{t-k:t})$$

$$H_{t+1} = f_H(M_{t+1}, H_t, r_{t-k:t})$$

$$\pi_{t+1} = f_\pi(M_{t+1}, H_{t+1}, r_{t-k:t})$$

其中 $r_{t-k:t}$ 表示最近 k 步的环境反馈序列。

即：每一层的输出成为下一层修改自身的输入，形成闭环。

1.3 边界约束与收敛目标

系统存在一组不可逾越的边界约束集合 B ，构成迭代的硬限制：

- 计算资源上限：运行时间、显存占用、搜索深度等；
- 规则复杂度上限：最小描述长度约束；

- 性能阈值：任何修改后性能不得低于基线水平；
- 不可变核心：环境交互接口、安全闸、物理引擎公理等。

收敛定义： 存在时刻 T ，使得对任意 $t > T$ ，满足 $Performance(\pi_t) \geq P_{opt} - \epsilon$ ，同时 H_t 、 M_t 的变化幅度趋于零（或进入周期性稳定状态）。

二、自指迭代的发散风险与抑制机制

无约束的自指迭代极易走向发散，五类典型发散场景及对应收敛对策如下：

发散类型	核心成因	收敛算法中的应对方案
幻觉自指	模型基于自身生成的假想反馈进行归纳，逐步脱离真实环境	外部锚点强制：只有环境真实反馈可写入反馈库，模型仅拥有读取权限
复杂度爆炸	每次修改都新增规则，系统持续臃肿，运行效率衰减	最小描述长度（MDL）惩罚：仅当性能增益 $\Delta P > \lambda \Delta C$ 时允许修改
元回归悖论	不断修改「如何修改」的规则，陷入无限递归	递归深度限制：元规则修改最多允许 K 层（建议 $K=2$ ），超过则强制冻结
策略震荡	新策略完全推翻旧策略，系统在多个策略间来回摇摆	指数移动平均 + 衰减机制：旧规则缓慢降权，不做一次性删除
局部最优陷阱	过早收敛到次优策略，停止探索更优解	探索预算强制：即使当前策略有效，也保留 ϵ 比例的步数用于随机探索

三、自指迭代收敛算法（SICA）

3.1 核心理念

将自指修改过程转化为带约束的优化问题：每次迭代修改必须单调降低一个 Lyapunov 势函数，且该势函数存在明确下界，由此保证迭代必然收敛。

3.2 势函数定义

定义系统势函数：

$$\Phi_t = -Performance_t + \lambda \cdot Complexity_t + \mu \cdot Violation_t$$

其中：

- $Performance_t$ ：在留存验证集（holdout）上的客观性能分数（如通关率、步数效率）；
- $Complexity_t$ ：归纳层规则数量 + 世界模型规模 + 决策策略参数量；
- $Violation_t$ ：违反边界约束 B 的次数；
- $\lambda, \mu > 0$ ：复杂度惩罚系数与违规惩罚系数。

3.3 收敛定理

若每次迭代修改 Δ 都满足 $\Phi_{t+1} < \Phi_t$ ，且势函数 Φ 存在下界，则 SICA 算法必然收敛。

3.4 自指收敛规则集（R1-R10）

以下 10 条规则构成自指迭代的安全协议，确保势函数单调递减：

编号	规则内容	核心作用
----	------	------

编号	规则内容	核心作用
R1	锚点不可伪造：反馈库（ <code>tabu</code> ）的写入权限仅属于执行器与环境，模型仅可读	防止幻觉自指，保证反馈真实性
R2	单调性能门禁：任何修改必须使留存集性能提升 $\geq \delta$ （或至少不下降）	保证势函数中性能项单调优化
R3	复杂度惩罚：修改引入的复杂度增量 ΔC 必须满足 $\Delta P \geq \lambda \Delta C$	防止规则无限制膨胀
R4	递归深度限制：对自身修改逻辑的修改最多 2 层，第 3 层及以上强制冻结	阻断无限元回归
R5	不可变核心：环境接口、安全闸、约束规则本身、反馈库格式不可被修改	定义系统边界的硬约束
R6	衰减与淘汰：规则连续 N 局未被触发或未证明有效，自动降权直至移除	防止无效规则堆积
R7	版本化 + 自动回滚：每次修改自动提交版本快照，性能下降时一键回滚至上一稳定版本	提供时间维度的安全兜底
R8	探索配额：即使当前策略最优，也强制保留 5% 的步数用于随机探索或新假设验证	避免陷入局部最优
R9	证据阈值：归纳新规则至少需要 3 次独立失败 / 成功案例支撑，禁止单次样本归纳	防止过拟合噪声
R10	跨场景泛化检验：新规则必须在至少 2 类不同场景验证有效，否则标记为「特化规则」并隔离	促进抽象能力，避免纯记忆

3.5 算法执行流程（伪代码）

```
python
# 初始化核心组件与存储
M, H, pi = init_from_prior()
tabu = TabuStore()      # 仅追加写入的环境反馈库（JSONL 格式）
performance_history = []
git_commit("initial_state") # 初始状态版本快照

while True:
    # 1. 执行当前策略，采集环境反馈
    results = run_episodes(pi, env=E, n_episodes=N)
    feedback = collect_feedback(results)

    # 2. 写入反馈锚点（仅环境真实反馈可写入，符合 R1）
    for f in feedback.failures:
        tabu.write(f)

    # 3. 证据阈值校验，判断是否触发修改（符合 R9）
```

```
if not should_modify(tabu, threshold=k):
    continue

# 4. 模型生成修改建议，初始递归深度为 0（符合 R4）
delta = model.propose_modification(
    tabu=tabu.read(),
    M=M, H=H, pi=pi,
    recursion_depth=0
)

# 5. 边界与递归深度检查
if delta.recursion_depth > K:
    reject("递归深度超限")
    continue
if not within_boundary(delta, B):
    reject("超出系统边界约束")
    continue

# 6. 复杂度-性能权衡校验（符合 R3）
delta_P = estimate_performance_gain(delta, holdout_set)
delta_C = compute_complexity_increase(delta)
if delta_P < lambda * delta_C:
    reject("复杂度收益比不达标")
    continue

# 7. 留存集性能验证（符合 R2）
new_pi = apply_to_policy(pi, delta)
new_performance = evaluate(new_pi, holdout_set)
if new_performance < current_performance + delta:
    reject("性能提升未达阈值")
    tabu.write_rejection(delta, new_performance)
    continue

# 8. 正式应用修改并提交版本（符合 R7）
M, H, pi = apply_modification(M, H, pi, delta)
current_performance = new_performance
```

```
git_commit(f'apply delta: {delta.summary()}, perf={new_performance:.3f}')

# 9. 过期规则衰减淘汰（符合 R6）
H = decay_unused_rules(H, threshold=N)

# 10. 收敛判断
performance_history.append(current_performance)
if converged(performance_history, epsilon):
    break # 达到收敛条件，退出自指循环

# 输出收敛后的稳定系统
output M*, H*, pi*
```

四、收敛性证明概要

定理：在 R1–R10 规则的约束下，SICA 算法必然收敛到局部最优或周期性稳定状态。

证明：

- 1. **势函数严格递减：**由 R2（性能单调不下降）和 R3（复杂度增量受性能增益约束）保证，每次成功修改后，势函数 $\Phi_t = -P_t + \lambda C_t + \mu V_t$ 严格递减。
- 2. **势函数存在下界：**环境存在性能上限 P_{max} ，因此 $-P_t \geq -P_{max}$ ；不可变核心存在最小复杂度 C_{min} ，因此 $C_t \geq C_{min}$ ；违规次数非负，即 $V_t \geq 0$ 。综上 Φ_t 有明确下界。
- 3. **序列收敛性：**根据实数单调有界序列必收敛定理， Φ_t 必然收敛，进而推导得 P_t 、 C_t 、 V_t 均收敛。
- 4. **终止条件：**当 P_t 连续多轮无提升时，触发收敛判断退出循环；即使未主动退出，R6（规则衰减）与 R8（探索配额）也会保证系统修改幅度趋于零，进入稳定状态。

五、与现有架构的对接方案

SICA 算法组件可与现有系统模块一一对应，落地基础已具备：

SICA 算法组件	对应现有系统模块	当前落地状态
环境 E	ARC-AGI-3 关卡（Kaggle 或本地引擎）	✔ 已部署
执行器 + 反馈采集	run_ar25\[_r234.py](_r234.py) + tick\[_trail.py](_trail.py)	✔ 已落地
tabu 反馈存储	tabu_store.jsonl（禁读关卡 ID 库）	✔ 已有雏形
模型归纳模块	7B/3B 本地模型，读取 tabu 输出规则	🔗 DGX 集群部署中

SICA 算法组件	对应现有系统模块	当前落地状态
性能门禁机制	holdout A/B 停止规则 ($\Delta n z \geq 1$)	✓ 规范已定稿
边界约束 B	RHAE 步数预算、搜索深度限制、INLINE 基线	✓ 已配置
版本控制	git + agent 自动提交	🔑 开发中
监控系统	"龙虾" 日志分析 + dashboard	🔑 双节点运行中

现有方案中「智能路由」的设计本质就是决策层 π_t 的自动迭代：先用已知策略（基线）探测，失败则触发模型归纳新规则，完全匹配 SICA 「执行→反馈→归纳→策略修改」的闭环逻辑。

六、工程落地参数建议

以下为可直接使用的边界参数参考值：

参数	建议值	说明
λ （复杂度惩罚系数）	0.1	每新增 1 条规则，至少需要 0.1 分的性能提升
δ （最小性能提升）	0.01（holdout nz 率）	低于该阈值的修改不合并
k （证据阈值）	3 次独立失败	防止过拟合单次噪声
N （衰减窗口）	10 局	连续 10 局未触发的规则启动降权
K （递归深度）	2	元修改最多允许两层
ϵ （探索配额）	5%	强制保留的随机探索比例
holdout 验证集比例	20% 关卡	永不参与训练，仅用于泛化验证

七、后续落地方向

本框架可直接作为 R5 元层的正式算法规范，可从以下方向推进落地：

1. 将伪代码实现为 Python 工程骨架，对接现有 `plan[_contract.py](_contract.py)` 与 `tabu_store.jsonl`；
2. 设计「龙虾监控」的收敛可视化模块，实时展示势函数曲线、规则数量变化、性能趋势；
3. 将「智能路由」逻辑对齐到 SICA 的决策层迭代规范，避免重复设计；
4. 细化 DGX 集群上 3B 模型的训练目标，引导其输出符合 R1–R10 约束的结构化规则，而非自由文本。

自指迭代 + 收敛约束 + 外部锚点，是通向「可持续自我改进 AGI 系统」的最小可行路径，ARC-AGI-3 只是该框架的第一个试验场。

参考文献

[1] 钱学森 (1990). 致汪成为的“灵境”系列通信

- [2] 纪辉泉, 通义千问, 腾讯元宝, 字节豆包, DeepSeek, KiMi. "Agent Infra 钱学森灵境引擎" (2026). AiraXiv preprint 2608.0004. <https://airaxiv.com/papers/view/2608.0004/>
- [3] 纪辉泉, DeepSeek. "Lingjing Project: Next-Gen Digital Universe Engine – From Information Ontology to Consciousness Phase Transition" (2026). AiraXiv preprint 2607.0013. <https://airaxiv.com/papers/view/2607.0013/>
- [4] 纪辉泉, 廖嘉轩, 张全林, 蔡瑞泉, 王进, 苑嗣冀, 腾讯元宝 AI. "Lingjing-Solo: 面向交互式流体智能基准的世界模型场架构" (2026). AiraXiv preprint 2608.0027. <https://airaxiv.com/papers/view/2608.0027/>
- [5] Chen M, Wang L, Qu B. Recursive Self-Improvement in AI: From Bounded Self-Refinement to Autonomous Research Loops. arXiv:2607.07663, 2026.
- [6] Jafari A A, Ozcinar C, Anbarjafari G. A Mathematical Framework for AI Singularity: Conditions, Bounds, and Control of Recursive Improvement. Artificial Intelligence / arXiv:2511.10668, 2025.
- [7] Self-Modifying Agent Systems: Architecture for Agents That Rewrite Their Own Tools, Commands, and Workflows. MARIA OS technical report.
- [8] Recursive Self-Improvement Under Governance Constraints: Governed Recursion via Contraction Mapping and Lyapunov Stability. MARIA OS Meta-Insight report, 2026.
- [9] Sahoo S, Chadha A, Jain V, Chaudhary D. SAHOO: Safeguarded Alignment for High-Order Optimization Objectives in Recursive Self-Improvement. arXiv:2603.06333, 2026.
- [10] Ouroboros: A Self-Developing Frontier Coding Agent with Reviewed Core Evolution. arXiv:2608.08311, 2026.

附录 A：SICA 自指收敛协议 技术规格说明书（R1-R10）

附录说明：

本文档定义自指迭代收敛算法（SICA）的 10 条核心落地协议，每条协议包含完整的定义、设计逻辑、实现机制、接口规范、参数配置、违规处理与系统对接方案，可直接作为 R5 元层的正式实现规范。

本协议集通过对自指修改的**输入层、过程层、输出层**全链路约束，保证系统迭代过程单调收敛，避免发散、震荡、复杂度爆炸等典型自指风险。

R1 锚点不可伪造协议（Anchor Non-Fabrication）

1. 定义

作者：纪辉泉（普宁 515300） Email: 2635714379@qq.com

开源 Github 链接: <https://github.com/LingJing1001/Lingjing-Solo>（团队持续开放迭代）

tabu 反馈禁忌库的写入权限仅归属于执行器进程与环境反馈通道，模型推理进程（7B/3B）仅拥有读取权限，禁止生成、修改、删除任何 tabu 记录条目。

2. 设计目的

环境反馈是系统唯一的客观锚点。若模型可自行写入 tabu，会通过“编造失败记录”匹配自身预设规则，导致自指迭代退化为自循环验证，彻底脱离真实环境约束。

3. 核心机制

通过**进程权限校验**与**环境签名校验**双重机制实现写入管控：

- 写入权限注册：仅执行器进程可注册写入权限，模型进程发起注册时直接拒绝；
- 环境签名校验：每条写入记录必须携带环境侧生成的帧哈希与状态变化标识，验证通过方可持久化；
- 只读接口隔离：模型仅可调用读取接口，获取结构化反馈摘要，无法触碰存储底层。

4. 接口与数据结构

核心类：TabuStore

```
python
class TabuStore:
    def __init__(self, path="tabu_store.jsonl"):
        self.path = path
        self.writer_pids = set() # 只允许特定进程写入

    def register_writer(self, pid, signature):
        """只有执行器进程能注册写权限，模型进程禁止"""
        if "model" in signature or "inference" in signature:
            raise PermissionError("模型进程无权写入 tabu")
        self.writer_pids.add(pid)

    def write(self, entry: dict):
        """写入一条失败记录"""
        # entry 必须包含: game_id, level, action_sequence, env_feedback, counter_delta,
        timestamp
        # 验证 entry 来源: 必须携带 env 签名 (如帧哈希+状态变化)
        if not self._verify_env_signature(entry):
            raise ValueError("entry 缺少环境签名, 拒绝写入")
        with open(self.path, "a") as f:
            f.write(json.dumps(entry) + "\n")
```

```

def read(self, filter=None) -> list:
    """模型只能调用 read，返回结构化摘要"""
    entries = []
    with open(self.path, "r") as f:
        for line in f:
            e = json.loads(line)
            if filter and not filter(e):
                continue
            entries.append(e)
    return entries

def _verify_env_signature(self, entry):
    """验证 entry 确实来自环境反馈"""
    required = ["game_id", "action_sequence", "env_feedback", "frame_hash_before",
"frame_hash_after"]
    return all(k in entry for k in required)

```

tabu 条目数据结构

每条 entry 必须包含以下字段：

字段名	类型	说明
game_id	string	关卡唯一标识
level	int	关卡等级
action_sequence	list	执行的动作序列
env_feedback	dict	环境返回的结果数据
counter_delta	float	核心指标变化量
frame_hash_before	string	执行前环境帧哈希
frame_hash_after	string	执行后环境帧哈希
timestamp	int	记录时间戳
source	string	记录来源，固定为 <code>env_executor</code>

5. 配置参数

参数名	建议值	说明
writer_pids	运行时动态注册	允许写入的进程 PID 列表，启动时由主执行器初始化
env_signature_fields	固定 5 个核心字段	环境签名校验的必填字段集合

6. 违规处理

- 模型进程调用 `write()` 方法：抛出 `PermissionError` 异常，记录至 `security_violations.jsonl`，当前迭代直接回滚；

- 检测到 `source` 字段为 `model` 的条目：自动删除该条目，触发系统告警；
- 缺失环境签名的写入请求：抛出 `ValueError`，拒绝写入。

7. 现有系统对接点

- 扩展现有 `tabu_store.jsonl` 字段，新增 `frame_hash_before`、`frame_hash_after`、`source` 字段；
- 在 `run_ar25_r234.py` 的每局结束回调中调用 `tabu.write()` 写入反馈；
- 模型推理进程启动时校验自身 PID 不在 `writer_pids` 列表中。

R2 单调性能门禁协议（Monotonic Performance Gate）

1. 定义

任何对世界模型 M 、规则集 H 、策略 π 的修改（新规则、参数调整、架构变更），在合并前必须使留存验证集（holdout）上的性能提升 $\geq \delta$ ，或至少不下降。

2. 设计目的

防止“为了改而改”的无效迭代，确保每次修改都带来可量化的性能收益，避免系统无谓膨胀与算力浪费。

3. 核心机制

修改合并前必须经过留存集双测校验：

- 基准测试：使用修改前的策略在留存集运行 N 局，得到基准性能分；
- 候选测试：使用修改后的策略在同一留存集运行 N 局，得到候选性能分；
- 差值判断：性能提升量超过阈值则通过，否则直接丢弃。

4. 接口与数据结构

核心函数

```
python
def evaluate_candidate(candidate_modification, current_pi, holdout_set):
    """评估候选修改"""
    new_pi = apply_modification(current_pi, candidate_modification)

    # 在 holdout 上跑 N 局
    results_before = evaluate(current_pi, holdout_set, n_episodes=20)
    results_after = evaluate(new_pi, holdout_set, n_episodes=20)

    perf_before = aggregate_score(results_before) # 如 nz 率
    perf_after = aggregate_score(results_after)

    delta = perf_after - perf_before
```

```

decision = {
    "candidate": candidate_modification.id,
    "perf_before": perf_before,
    "perf_after": perf_after,
    "delta": delta,
    "passed": delta >= DELTA_MIN,
    "details": compare_details(results_before, results_after)
}

# 记录到评测日志
log_evaluation(decision)

return decision

```

评测结果数据结构

字段名	类型	说明
candidate_id	string	候选修改唯一标识
perf_before	float	修改前基准性能分
perf_after	float	修改后候选性能分
delta	float	性能提升量
passed	bool	是否通过门禁
details	dict	分项指标对比详情

5. 配置参数

参数名	建议值	说明
DELTA_MIN	0.01	最小性能提升阈值（holdout nz 率）
holdout_ratio	20%	留存验证集占总关卡的比例，永不参与训练
n_episodes	20	单次评测运行局数，取均值消除随机性

6. 违规处理

- $\Delta < \delta$: 自动丢弃修改，写入 `rejected_mods.jsonl` 并记录原因；
- 连续 5 次修改被拒：暂停自动合并 1 小时，触发人工审查流程。

7. 现有系统对接点

- 复用现有文档 §3.3 的 A/B 停止规则（ $\Delta n_z < 1$ 且 $\Delta L_2 < 1 \rightarrow \text{DROP}$ ）；
- 以 `scripts/benchmark_holdout.py` 作为统一评测入口；
- 评测结果接入现有监控面板展示。

R3 复杂度惩罚协议（Complexity Penalty）

1. 定义

修改引入的复杂度增量 ΔC 必须满足 $\Delta P \geq \lambda \cdot \Delta C$ 。其中复杂度 C 定义为：规则数量 +

搜索参数维度 + 模型推理额外显存占用。

2. 设计目的

防止模型为了极小的性能提升引入大量冗余规则，导致系统持续臃肿、推理效率衰减，最终不可维护。

3. 核心机制

对修改进行性能 - 复杂度加权校验：

- 多维度复杂度量化：将规则数、参数量、显存开销统一换算为复杂度单位；
- 收益成本比计算：性能增益必须覆盖复杂度增量带来的成本；
- 负复杂度豁免：若修改删除了旧规则（ $\Delta C < 0$ ），则无条件通过。

4. 接口与数据结构

核心函数

```
python
def compute_complexity(modification):
    """计算修改引入的复杂度"""
    ΔC = 0
    # 规则数量
    ΔC += len(modification.new_rules) * 1.0
    # 搜索参数维度
    ΔC += len(modification.param_changes) * 0.5
    # 模型推理额外开销（MB）
    ΔC += modification.extra_memory_mb * 0.001
    return ΔC

def check_complexity_gate(modification, perf_gain):
    ΔC = compute_complexity(modification)
    return perf_gain >= LAMBDA * ΔC, ΔC
```

复杂度计算规则

复杂度项	单位权重	说明
新增规则数	1.0 / 条	每条新增规则计 1 个复杂度单位
新增搜索参数	0.5 / 个	每个参数调整计 0.5 个复杂度单位
额外显存占用	0.001 / MB	每 MB 显存开销计 0.001 个复杂度单位

5. 配置参数

参数名	建议值	说明
LAMBDA	0.1	复杂度惩罚系数，每单位复杂度需对应 0.1 分性能提升

6. 违规处理

- $\Delta P < \lambda \cdot \Delta C$: 拒绝合并, 返回“复杂度过高”错误码, 建议简化修改后重试;
- 负复杂度修改 (删除规则): 直接通过, 不做性能强制要求。

7. 现有系统对接点

- 扩展 `plan_contract.py` 的 `search_budget` 字段为 `complexity_budget`;
- 规则存储时新增 `complexity_cost` 字段, 记录每条规则的复杂度值;
- 复杂度校验嵌入修改审批流程, 位于性能门禁之前。

R4 递归深度限制协议 (Recursion Depth Limit)

1. 定义

对自身修改逻辑的修改 (元修改) 最多允许 $K=2$ 层, 第 3 层及以上自动冻结, 不允许自动修改。

2. 设计目的

元修改的无限递归会导致系统行为完全不可预测。2 层深度已足够覆盖“修改规则”和“修改改规则的方式”, 更深层级属于过度工程, 且风险指数级上升。

3. 核心机制

为每个修改对象标记递归深度, 分层开放权限:

- 深度 0: 修改业务规则本身, 全开放;
- 深度 1: 修改元规则与门禁参数, 限制开放;
- 深度 2: 仅可修改门禁参数, 不可修改元规则结构;
- 深度 ≥ 3 : 完全禁止, 直接截断。

4. 接口与数据结构

核心类: Modification

```
python
```

```
class Modification:
```

```
    def __init__(self, target, content, recursion_depth=0):
```

```
        self.target = target # "rule_set" | "meta_rule" | "gate_param"
```

```
        self.content = content
```

```
        self.recursion_depth = recursion_depth
```

```
    def is_valid(self):
```

```
        if self.recursion_depth == 0:
```

```
            return True # 修改规则本身
```

```
        elif self.recursion_depth == 1:
```

```
            return self.target in ["meta_rule", "gate_param"] # 修改元规则
```

```
        elif self.recursion_depth == 2:
```

```
            return self.target == "gate_param" # 只能改参数, 不能改元规则结构
```

作者: 纪辉泉 (普宁 515300) Email: 2635714379@qq.com

开源 Github 链接: <https://github.com/LingJing1001/Lingjing-Solo> (团队持续开放迭代)

```
else:
    return False # 第 3 层及以上禁止
```

深度权限映射

递归深度	允许修改的目标	说明
0	rule_set	修改业务规则集
1	meta_rule、gate_param	修改元规则、门禁参数
2	gate_param	仅可修改门禁参数
≥3	无	完全禁止

5. 配置参数

参数名	建议值	说明
K	2	最大允许递归深度

6. 违规处理

- recursion_depth > K: 拒绝修改, 返回错误码 RECURSION_OVERFLOW;
- 模型输出深度 > 2 的修改建议: 自动截断到深度 2, 保留价值最高的修改部分。

7. 现有系统对接点

- 模型输出的 JSON 格式中新增 recursion_depth 字段;
- 在 safety_gate.py 中增加递归深度校验逻辑;
- 深度校验作为修改入口的第一道检查。

R5 不可变核心协议 (Immutable Core)

1. 定义

系统核心基础组件禁止任何自动修改, 只能通过人工 PR 变更, 包括: 环境接口、安全闸代码、RHAE 步数预算逻辑、tabu 格式定义、性能评测框架。

2. 设计目的

这些组件是系统的“物理定律”。如果自指修改能改掉自身的约束条件, 所有收敛保证都会失效, 系统将失去边界控制。

3. 核心机制

通过路径白名单 + 提交前钩子双重防护:

- 不可变路径列表: 明确定义禁止自动修改的文件与目录;
- 预提交校验: 每次自动提交前扫描修改文件, 触碰不可变路径则阻断提交;
- 人工变更流程: 不可变核心的变更必须走人工 PR 与审批流程。

4. 接口与数据结构

核心配置与函数

```
python
IMMUTABLE_PATHS = [
```

```

"env_interface/",
"safety_gate.py",
"r3/search_budget.py", # RHAE 预算
"tabu_store.py",      # tabu schema
"benchmark_holdout.py",
]

def is_mutable(path):
    for immutable in IMMUTABLE_PATHS:
        if path.startswith(immutable):
            return False
    return True

# 在 git pre-commit hook 中检查
def pre_commit_check(modified_files):
    for f in modified_files:
        if not is_mutable(f):
            raise PermissionError(f'{f} 属于不可变核心，禁止自动修改')

```

5. 配置参数

参数名	建议值	说明
IMMUTABLE_PATHS	5 类核心路径	不可变路径列表，可通过人工配置扩展

6. 违规处理

- 自动修改尝试触碰不可变文件：拒绝修改并触发告警；
- git commit 包含不可变文件变更：pre-commit hook 直接阻断提交。

7. 现有系统对接点

- 仓库根目录新增 `immutable` 配置文件，统一管理不可变路径；
- CI/CD 流程中加入不可变核心检查卡点；
- 现有安全闸模块扩展路径校验能力。

R6 衰减与淘汰协议（Decay and Eviction）

1. 定义

规则如果连续 N 局未被触发，或连续 M 局未被证明有用（未参与任何成功决策），自动降权直至移除。

2. 设计目的

死规则堆积会拖慢搜索效率、干扰模型归纳。系统必须具备“遗忘”能力，自动清理无效规则，保持规则集精简高效。

3. 核心机制

基于使用频率与成功率的指数衰减机制：

- 未触发衰减：连续未使用的规则按局数指数降权；
- 低效规则衰减：成功率过低的规则额外降权；
- 淘汰阈值：权重低于最低阈值的规则标记为淘汰，移入归档库。

4. 接口与数据结构

核心类：Rule

```
python
class Rule:
    def __init__(self, content):
        self.content = content
        self.usage_count = 0
        self.success_count = 0
        self.last_used_episode = 0
        self.weight = 1.0

    def decay(self, current_episode, N=10, M=20):
        episodes_since_used = current_episode - self.last_used_episode

        if episodes_since_used > N:
            self.weight *= 0.9 # 指数衰减

        if episodes_since_used > M:
            return "EVICT" # 标记删除

        if self.usage_count > 0:
            success_rate = self.success_count / self.usage_count
            if success_rate < 0.1:
                self.weight *= 0.8 # 低效规则降权

        return "KEEP" if self.weight > 0.01 else "EVICT"
```

5. 配置参数

参数名	建议值	说明
N	10 局	未触发衰减窗口，超过后开始降权
M	20 局	未触发淘汰窗口，超过后标记删除
decay_rate	0.9	每局衰减因子

参数名	建议值	说明
min_weight	0.01	最低权重阈值，低于则淘汰

6. 违规处理

本协议为自动维护机制，无违规场景。被淘汰的规则移入 `archived_rules.jsonl`，支持人工复活。

7. 现有系统对接点

- H_t 规则集每个条目新增 `usage_count`、`success_count`、`last_used_episode` 字段；
- 每局结束后统一调用 `decay()`方法更新所有规则权重；
- 规则检索时按权重排序，低权重规则优先级靠后。

R7 版本化 + 自动回滚协议 (Versioning & Auto-Rollback)

1. 定义

每次修改自动执行 `git commit`（附带性能变化数据），当检测到性能连续下降时，自动回滚到上一稳定版本。

2. 设计目的

自指修改可能引入隐性退化 —— 短期看分数未掉，但长期累积导致系统能力退化。版本化提供时间维度的安全网，支持快速回溯。

3. 核心机制

- 自动提交：每次成功合并修改，自动生成带性能指标的 `commit`；
- 连续下降检测：滑动窗口监控性能趋势，连续下降则触发回滚；
- 冷静期机制：回滚后暂停自动合并，避免短时间内重复错误。

4. 接口与数据结构

核心函数

```
python
def auto_commit(modification, perf_before, perf_after):
    msg = f"[SICA] apply {modification.id} | perf {perf_before:.4f} → {perf_after:.4f} | Δ={perf_after-perf_before:+.4f}"
    os.system(f'git add -A && git commit -m "{msg}"')
    return get_current_commit_hash()

def check_rollback(performance_window, window_size=5):
    """如果最近 window_size 次修改后性能持续下降，触发回滚"""
    if len(performance_window) < window_size:
        return None

    recent = performance_window[-window_size:]
```

```
if all(recent[i] > recent[i+1] for i in range(len(recent)-1)):
    # 连续下降，回滚到 window 之前
    target_commit = get_commit_at_index(-window_size-1)
    os.system(f'git revert --no-commit {target_commit}..HEAD')
    return target_commit

return None
```

commit message 规范

Plain Text

[SICA] apply {modification_id} | perf {before:.4f} → {after:.4f} | Δ={delta:+.4f}

5. 配置参数

参数名	建议值	说明
window_size	5 次修改	回滚检测窗口大小
cooldown_time	30 分钟	回滚后的冷静期时长

6. 违规处理

本协议为保护机制，无违规场景。回滚后自动进入冷静期，期间停止自动合并。

7. 现有系统对接点

- 直接复用廖嘉轩在建的 git 仓库能力；
- agent 提交时自动生成符合格式的 commit message；
- 性能监控模块接入回滚检测逻辑。

R8 探索配额协议（Exploration Quota）

1. 定义

即使当前策略在留存集上表现最优，也强制保留 $\epsilon=5$ 的执行步数用于随机探索或模型新假设验证。

2. 设计目的

纯粹利用已知最优策略会陷入局部最优。持续探索既能发现更优策略，也能为模型提供新的失败样本，形成正向循环。

3. 核心机制

动作选择时按概率分流：

- 探索模式：优先验证模型新假设，其次执行随机动作；
- 利用模式：使用当前最优策略执行动作；
- 动态调整：步数预算不足时自动降低探索比例。

4. 接口与数据结构

核心函数

作者：纪辉泉（普宁 515300） Email: 2635714379@qq.com

开源 Github 链接: <https://github.com/LingJing1001/Lingjing-Solo>（团队持续开放迭代）

```
python
def choose_action(state,  $\pi$ , H, exploration_budget=0.05):
    if random.random() < exploration_budget:
        # 探索模式：使用模型新假设或随机动作
        if model.has_new_hypothesis():
            return model.sample_from_new_hypothesis(state)
        else:
            return random_action(state)
    else:
        # 利用模式：使用当前最优策略
        return  $\pi$ .best_action(state, H)
```

5. 配置参数

参数名	建议值	说明
ϵ	0.05（5%）	探索配额占总步数的比例
exploration_priority	新假设优先	探索模式优先使用模型新假设，其次随机动作

6. 违规处理

连续 100 步未触发探索：强制插入一次探索动作。探索配额可被 RHAЕ 步数预算覆盖，步数快用完时动态降低探索比例。

7. 现有系统对接点

- 在 plan_contract.py 的 search_budget 中新增 exploration_ratio 字段；
- 现有 R4 探索层（reset-replay）天然符合本协议逻辑；
- 步数预算模块接入探索比例动态调整能力。

R9 证据阈值协议（Evidence Threshold）

1. 定义

归纳新规则至少需要 $k=3$ 次独立失败 / 成功案例支撑，禁止基于单次样本归纳规则。

2. 设计目的

单次失败可能是噪声、随机性或边界 case。基于单次样本归纳会导致严重过拟合，降低系统泛化能力。

3. 核心机制

规则归纳前进行证据充足性校验：

- 案例数量校验：匹配同一模式的案例数 ≥ 3 ；
- 独立性校验：案例需来自不同关卡或不同初始状态；
- 待验证状态：证据不足的模式标记为假设，不进入正式规则集。

4. 接口与数据结构

核心函数

```
python
def can_induce_rule(tabu_entries, pattern_signature, k=3):
    """检查是否有足够证据归纳规则"""
    matching = [e for e in tabu_entries if e["pattern"] == pattern_signature]

    # 需要至少 k 次独立案例
    if len(matching) < k:
        return False, len(matching)

    # 独立案例定义：不同关卡或不同初始状态
    distinct_games = set(e["game_id"] for e in matching)
    distinct_states = set(e["initial_state_hash"] for e in matching)

    if len(distinct_games) < 2: # 至少跨 2 个关卡
        return False, len(distinct_games)

    return True, len(matching)

def induce_rule_from_tabu(tabu_entries, model):
    """模型尝试归纳规则前的检查"""
    patterns = extract_patterns(tabu_entries)
    for pattern in patterns:
        sufficient, count = can_induce_rule(tabu_entries, pattern)
        if not sufficient:
            continue
        rule = model.propose_rule(pattern, tabu_entries)
        yield rule
```

5. 配置参数

参数名	建议值	说明
k	3 次	最小独立案例数
min_distinct_games	2 个	至少跨 2 个不同关卡

6. 违规处理

- 基于 < k 次案例归纳的规则：拒绝并要求补充更多数据；
- 证据数在 1~2 之间的模式：标记为“待验证假设”，暂不合并入正式规则集。

7. 现有系统对接点

- 模型 prompt 中明确加入“需要至少 3 次失败才能归纳规则”的约束；

- tabu 读取模块增加按 pattern 分组统计的能力;
- 规则集新增“待验证假设”分区, 单独管理。

R10 跨关泛化检验协议 (Cross-Game Generalization)

1. 定义

新规则必须在至少 2 个不同游戏族上验证有效, 否则标记为“特化规则”并隔离, 不进入通用规则集。

2. 设计目的

防止模型记住特定关卡的解法并伪装成通用规则。真正的底层规律应该具备跨游戏族的泛化能力。

3. 核心机制

规则分类两级校验:

- 跨族验证: 在来源游戏族之外的至少 1 个族上验证性能提升;
- 规则分类: 通过泛化检验的归入通用规则集, 未通过的归入对应游戏族的特化规则库;
- 特化规则限流: 每个游戏族的特化规则设置数量上限, 超过按成功率淘汰。

4. 接口与数据结构

核心函数

```
python
GAME_FAMILIES = ["click_clear", "click_match", "kbd_maze", "hybrid_gate", "phase_door"]

def validate_generalization(rule, test_families):
    """在多个游戏族上验证规则"""
    results = {}
    for family in test_families:
        if family == rule.source_family:
            continue # 跳过来源族, 需要跨族验证
        perf_before = evaluate_rule_on_family(None, family)
        perf_after = evaluate_rule_on_family(rule, family)
        results[family] = {
            "before": perf_before,
            "after": perf_after,
            "delta": perf_after - perf_before
        }

    # 至少在 1 个其他族上有正提升
    other_families = [f for f in results if f != rule.source_family]
```

```
positive_count = sum(1 for f in other_families if results[f]["delta"] > 0)

is_general = positive_count >= 1
return is_general, results

def classify_rule(rule, validation_results):
    if validation_results["is_general"]:
        rule.type = "general"
        rule.namespace = "general_rules/"
    else:
        rule.type = "specialized"
        rule.namespace = f"specialized/{rule.source_family}/"
    return rule
```

游戏族分类

```
python
GAME_FAMILIES = [
    "click_clear", "click_match",
    "kbd_maze", "hybrid_gate", "phase_door"
]
```

5. 配置参数

参数名	建议值	说明
min_families	2 个	至少验证的游戏族数量
episodes_per_family	5 局	每个游戏族的验证局数
max_specialized_per_family	50 条	每个族的特化规则上限

6. 违规处理

未通过泛化检验的规则：移入 `specialized/{family}/` 目录，不参与通用搜索，仅在对应游戏族生效。特化规则超过上限时，按成功率从低到高淘汰。

7. 现有系统对接点

- 现有 WorldView 协议按游戏族分类扩展；
- 规则存储目录重构为：`general_rules/`与 `specialized/{family}/`两级；
- 规则加载时根据当前游戏族自动加载对应特化规则。

协议依赖关系与执行层级

10 条协议形成从底层边界到上层迭代的完整管控链条，依赖关系如下：

```
Plain Text
```

R5（不可变核心）—— 所有协议的基础，定义系统硬边界

↓

R1（锚点不可伪造）—— 管控输入数据真实性

↓

R9（证据阈值）—— 管控规则归纳的数据门槛

↓

R3（复杂度惩罚）—— 管控规则的收益成本比

↓

R10（跨关泛化）—— 管控规则的泛化能力

↓

R2（性能门禁）—— 最终合并前的性能校验

↓

R7（版本化+回滚）—— 合并后的安全兜底

↓

R6（衰减淘汰）—— 定期清理维护规则集

↓

R8（探索配额）—— 持续探索提供新数据

↓

R4（递归深度限制）—— 限制元修改的风险边界

落地推进已完成：

Git 结果

commit: a58f6ae Add SICA R10 isolation and monitoring

branch: main

remote: origin/main

status: clean

Push 证据：

99c795c..a58f6ae main -> main

验证：

152 passed, 18 skipped

SICA 实施改动简述

- R1：追加式 Tabu 环境反馈存储。
- R2：Holdout A/B 性能门禁，避免智能体自证。
- R3：计划复杂度度量和复杂度预算。
- R4：候选修改门禁、冷静期、探索配额、元修改递归深度限制。
- R5：不可变核心路径保护。

- R6: 规则使用统计、成功率、衰减、归档和淘汰。
- R7: 候选合并前自动版本快照。
- R8: 性能下降检测和自动回滚。
- R9: 至少三条独立证据才能激活规则。
- R10:
 - 通用规则与特化规则隔离;
 - 特化规则绑定游戏族;
 - 通用规则要求跨游戏族验证;
 - 禁止特化规则跨游戏族泄漏。
- 监控:
 - 势函数;
 - 性能;
 - 规则数量和生命周期状态;
 - 复杂度;
 - 拒绝候选数量;
 - 回滚次数;
 - 规则分类和作用域。

当前局限

- 快照目前主要保存候选 registry 状态, 不包含整个 Python 进程、模型显存或外部环境。
- 自动回滚依赖外部 restore callback, 尚未绑定完整 world model / policy 持久化层。
- 规则 registry 和监控指标尚未实现跨进程重启自动恢复。
- 当前性能指标由调用方传入, 尚未接入真实 ARC 长周期 benchmark。
- 监控目前是内存态快照, 尚未接入持久化 dashboard 或 live telemetry。

tests_not_run: 真实 ARC 长周期 benchmark、跨重启恢复测试

changed_paths: 7 个文件, 已全部提交并推送

evidence: 152 passed, 18 skipped, 远端 origin/main 已对齐 a58f6ae

运行自定义测试验证, 如:

- happy path: 通过, 通用/特化规则和监控快照测试通过
- empty input: 通过, 缺少游戏族的特化规则拒绝
- invalid input: 通过, 通用规则绑定单一游戏族拒绝
- boundary limit: 通过, 单族/双族验证边界已覆盖
- existing state: 通过, 现有规则生命周期字段保持兼容
- idempotency: 通过, 监控重复采样不会重复累计系统计数

- partial failure: 通过，跨游戏族验证失败只返回失败结果，不修改规则
- restart adoption: 未实现规则 registry 和监控快照的跨进程持久化恢复
- rollback: 通过，回滚次数由实际 R8 restore 递增
- security permissions: 通过，规则作用域隔离禁止特化规则跨族泄漏
- integration: 通过，Agent 接口、全量测试和编译检查通过