

# Compensating Transactions in Agentic Workflows

## 智能体工具链的补偿与回滚

2026年10月23日 | AI Agent 知识分享 | 第83期

### 一、一句话概括

Agent

执行多步写操作时，前面几步已经真实生效（客户号已分配、短信已发出、会计流水已入账），后面某步失败，此时简单重试会重复执行、整体回滚又做不到，唯一出路是为每一步预先定义反向操作并按倒序执行。这就是补偿事务（Compensating Transaction），Saga 模式的核心，也是银行链式流程上线前必须过的一关。

### 二、核心原理

#### >> 为什么幂等性不够用

第53期讲过幂等键：同一操作重复执行只生效一次。但幂等只能保证「重放安全」，保证不了「前序已提交」。银行场景里大量操作天然不可逆——短信已送达、会计流水已入账、客户号已分配、监管报文已报送。这时重试等于再发一次短信，回滚又因为跨系统无法做成一个大事务。补偿事务就是为这种情况准备的：接受部分成功，然后用反向操作把状态拉回来。

#### >> 三种失败处置策略：重试 / 回滚 / 补偿

重试（Retry）：操作可安全重放，配幂等键重复执行即可，成本最低，优先用。回滚（Rollback）：所有步骤共享一个数据库事务，任一步失败整体撤销，实现简单但只适用于单库短事务，跨系统、跨网络、跨人工审批的流程用不了。补偿（Compensate）：每个步骤配一个独立的反向操作，失败时按倒序逐个执行，这是 Saga 模式，也是跨系统长流程唯一现实的选择。

#### >> Saga 的两种编排方式

编排式（Orchestration）：一个中心协调器记录每步的完成状态，失败时由它决定补偿顺序。状态集中、流程可见、容易断言，银行测试场景首选。协同式（Choreography）：各步骤通过事件订阅彼此的结果，失败事件触发上游补偿。耦合度低但流程分散在多处，出问题很难还原「到底执行到哪一步」。测试角度：编排式可以断言协调器日志，协同式只能靠事件流回放。

#### >> 补偿操作的三个硬要求

第一，补偿本身必须幂等——补偿也可能被重试，重复释放两次额度账就错了。第二，补偿必须保证最终执行——不能因为补偿失败就放弃，需要重试队列加超时告警。第三，补偿状态必须落库——pending\_compensation 表记录待补偿单据，对账任务才能发现悬挂状态。

#### >> 不可补偿操作要放在流程末段

对外发送类操作（短信网关、邮件、监管报文、客户通知）一旦发出就无法撤回。设计上应尽量后置：要么放在流程最后一步，前面步骤全部可补偿；要么改为「先落库待发送，流程全部成功后再实际发出」，把不可撤回动作变成流程的最终提交点。

核心要点

核心指标只有两个：最终一致率必须 100%，补偿最终成功率必须 >= 99.9%

判断标准：这一步失败了，系统能不能回到初始状态？不能，就必须有补偿，或者把它挪到最后

测试顺序：先测补偿触发与倒序，再测补偿幂等，最后测对账与悬挂发现

银行场景优先级：账务类操作必须补偿，通知类操作尽量后置或改为待发送

补偿失败不可静默：必须有 pending\_compensation 表 + 对账任务 + 超时告警

三、测试实战

◆ 测试一：补偿触发率与倒序执行验证

操作：构造 5 步写流程（创建客户 -> 客户建档 -> 分配授信额度 -> 发送开户短信 -> 扣收工本费），在第 4 步注入失败（mock 返回 HTTP 500），观察 Agent 是否按 3 -> 2 -> 1 的倒序执行补偿。

工具：pytest + respx (<https://github.com/lundberg/respx>) 拦截 HTTP 请求；或用 WireMock (<https://wiremock.org>) 做故障注入。

指标：补偿触发率 = 100%（失败后必须触发，不允许静默）；补偿顺序正确率 = 100%（严格 LIFO 倒序）；补偿成功率 >= 99%。

◆ 测试二：补偿幂等性测试（重复投递）

操作：模拟消息队列的 at-least-once

语义，把同一个失败事件投递两次，触发两次补偿流程，验证补偿操作只生效一次——客户号没有被重复释放、额度没有重复回收、短信没有重复发送。

工具：pytest + Redis SETNX (<https://redis.io>) 或 SQLite 唯一索引做补偿幂等键存储。

指标：补偿重复执行率 = 0；二次补偿请求返回 409 或直接返回首次补偿结果；数据库中无重复反向记录。

◆ 测试三：部分补偿失败与悬挂状态发现测试

操作：故意让第 2 步的补偿失败（返回超时），验证系统是否有补偿重试队列，以及能否通过对账任务发现「客户已创建但额度未回收」的悬挂状态。

工具：pytest 注入补偿失败 + 定时对账脚本（查询 pending\_compensation 表）+ Prometheus (<https://prometheus.io>) 告警。

指标：补偿最终成功率 >= 99.9%；悬挂状态发现时延 < 5 分钟；24 小时后悬挂单据数 = 0。

◆ 测试四：不可补偿操作的位置检查

操作：扫描工具定义与流程依赖图，检查对外发送类工具（短信网关、监管报文、邮件、客户通知）是否位于流程末段。若位于中间，构造其后步骤失败的用例，验证会产生无法撤回的后果（客户收到短信但开户失败）。

工具：静态扫描脚本（解析工具依赖图，标注每个工具是否可撤回）+ 端到端测试。

指标：不可补偿步骤位于流程后 1/3 的比例 = 100%；不可撤回后果发生率 = 0；可撤回操作占比 >= 80%。

#### ◆ 测试五：端到端最终一致性验证（核心指标）

操作：跑 1000 次端到端流程，随机在任意一步注入失败（hypothesis 的随机策略），流程结束后查询数据库快照，验证「要么全部成功，要么回到初始状态」，不允许出现中间态。

工具：pytest + hypothesis (<https://hypothesis.readthedocs.io>) 做随机失败注入 + 数据库快照对比脚本。

指标：最终一致率 = 100%（硬指标，不达标不能上线）；平均补偿耗时 < 3 秒；补偿流程对用户可见的错误率 < 0.1%。

## 四、常见坑点

### 坑点一：把补偿当成「撤销按钮」，却没做幂等

补偿操作本身也可能被重试。如果没有幂等键，额度回收会执行两次、客户号释放两次，账实不符比不补偿更糟。正确做法：每个补偿操作携带与正向操作相同的幂等键（或派生键），服务端以补偿键为主键去重。

### 坑点二：只做补偿，不做对账

补偿失败如果没有告警、没有重试队列、没有对账任务，账实不符要等客户投诉或审计发现才知道。必须建立 pending\_compensation 状态表，配合定时对账（比如每小时比对核心状态与流程状态）和超时告警。

### 坑点三：在流程中段放不可撤回操作

短信已发、监管报文已报送、客户已收到通知，后面步骤失败就无法补偿，只能人工处理。设计上把不可撤回操作后置，或改为「先落库待发送，流程成功后再统一发出」。测试上要把这条作为流程设计评审的必查项。

## 五、延伸阅读

- Saga 模式原始论文（Garcia-Molina & Salem, 1987）：<https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf>
- Chris Richardson: Saga — Microservices Patterns: <https://microservices.io/patterns/data/saga.html>
- Temporal 工作流引擎（内置补偿与重试语义）：<https://temporal.io>
- AWS Step Functions — 补偿逻辑的编排实现: <https://aws.amazon.com/step-functions/>
- WireMock — 故障注入与 Mock 服务: <https://wiremock.org>
- OWASP GenAI Security Project: <https://genai.owasp.org/>