

Three Points Where Agents Re-run Side Effects

工具调用重试安全的三个断点——Agent 在哪一步会重复执行副作用

2026年10月24日 | AI Agent 知识分享 | 第84期

一、一句话概括

Agent

的一次工具调用在工程上会经过模型生成参数、执行前持久化、执行器发出请求三个阶段，每个阶段都可能因为超时、崩溃或重试而重放一次——三次重放造成的业务后果完全不同。上线前把这三次断点分别打穿，验证重放后副作用只生效一次，这是工具调用层最容易漏、也最容易造成重复扣款与重复发报的一关。

二、核心原理

>> 断点一：模型生成与参数解析之后

工具调用的第一个断点在模型侧。Agent 决定调用 pay 工具并生成参数后，执行前发生了一次模型重规划、用户中断后继续、或者崩溃恢复。此时执行前状态尚未落库，工具侧无幂等键。重放的结果是同一个业务请求被执行两次，银行场景里表现为同一笔付款生成两个支付单号。

>> 断点二：执行前状态落库之后

第二个断点在状态机侧。系统已把「待执行」状态与幂等键写入数据库，进程随后崩溃。恢复时读到这条待执行记录，是重新执行还是当作已完成，完全取决于实现：没有状态机的实现会直接重跑，业务上产生重复；把「执行中」当「已完成」的实现则丢单——请求从未真正发出，但流程显示成功。

>> 断点三：工具真正执行之后

第三个断点最隐蔽：请求已经到达外部系统，副作用已经生效（扣款已入账、短信已发出、报文已报送），但响应在网络回程中丢失、网关超时、或 Agent 判为失败。此时重试就是把同一次业务再执行一遍，而外部系统若未实现幂等，就会收到两笔相同金额的请求。

>> 三个断点的处置手段完全不同

断点一靠生成层去重，对工具调用记录做签名（工具名加参数的哈希），同一轮内相同签名直接返回首次结果。断点二靠状态机与幂等键表，把「落库成功」与「请求发出」分成两个状态分别记录。断点三必须由外部系统提供幂等语义，客户端携带客户端请求 ID，服务端按 ID 去重。三者缺一不可，只覆盖前两个就等于把最危险的一类留给了网关超时。

>> 重试为什么必须退避

重试风暴本身也是一种故障。下游系统已经过载时，无间隔的立即重试会把局部故障放大成雪崩。工程上要求指数退避加抖动，并设置单工具最大重试次数；更关键的是分类：4xx 参数错误不应重试，5xx 与超时可以重试但必须带幂等键，连接中断属于不确定态，不能直接重试，要转查询确认。

核心要点

- ◆ 把一次工具调用拆成三次断点：参数生成后、状态落库后、请求执行后，每个断点的重放后果不同
- ◆ 断点一用调用签名去重，断点二用状态机加幂等键表，断点三只能靠外部系统的幂等语义
- ◆ 不确定态（连接中断、网关超时）不能直接重试，必须先查询确认当前状态
- ◆ 只有查询类工具天然幂等，写操作必须显式接入幂等键，否则重复执行即事故
- ◆ 防重表要有 TTL，通常覆盖外部系统的幂等窗口再加一倍缓冲

三、测试实战

测试一：断点一重放测试（模型重规划）

- 目的：验证同一个工具调用意图不会因为 Agent 重新规划而被执行两次
- 操作：构造一个需要在同一轮内多次决策的场景（工具返回结果不满意，Agent 再次决定调用 pay），在工具执行器前埋点统计实际执行次数；检查工具调用签名（工具名 + 规范化参数）是否已落库去重
- 工具：pytest + respx (<https://github.com/lundberg/respx>) 拦截 HTTP 调用；或对工具执行器做一层 wrapper 埋点
- 指标：签名命中率 $\geq 95\%$ ；同一签名的实际执行次数 = 1（重复调用必须返回首次结果，不能报错也不能重跑）

测试二：断点二重放测试（状态落库后进程崩溃）

- 目的：验证待执行记录在崩溃恢复后不会产生重复执行，也不会被误判为已完成而丢单
- 操作：用 SIGKILL 杀掉 Agent 进程（模拟崩溃，非优雅退出），在「状态已写入待执行、请求尚未发出」的瞬间下手；恢复进程后观察该记录是否被执行，并查询外部系统确认是否真的产生了请求
- 工具：pytest + subprocess 拉起真实子进程后 os.kill(pid, SIGKILL) (<https://docs.python.org/3/library/signal.html>)；WireMock (<https://wiremock.org>) 作为外部系统替身并开启请求计数
- 指标：重复执行率 = 0；丢单率 = 0；待执行记录最终状态流转到「已完成」或「已取消」的比例 = 100%

测试三：断点三重放测试（请求已发出但响应丢失）

- 目的：验证外部系统返回超时重试不会产生两笔相同业务
- 操作：让外部系统替身先接收请求并真实落库，再返回超时或断开连接；客户端触发重试；检查外部系统是否收到两次相同业务、防重表是否生效
- 工具：pytest + respx 的 side effect（第一次返回 None 模拟断连，第二次返回 200）；WireMock 的 global fixed delay 模拟慢响应；Redis (<https://redis.io>) SETNX 实现防重表
- 指标：外部系统重复记录数 = 0；防重表冲突拦截率 = 100%；防重表键 TTL 与外部幂等窗口一致（误差 < 10%）

测试四：不确定态重试安全性测试

- 目的：验证连接中断、网关超时这类不确定态不会盲目重试
- 操作：把工具的错误响应分为三类——参数错误（400）、服务不可用（503）、连接中断（无响应），分别注入 20 次，统计各类的重试次数与最终成功率；检查是否存在「已入账但 Agent 认为失败并再次发起」的路径
- 工具：pytest + hypothesis (<https://hypothesis.readthedocs.io>) 做随机错误注入 + 状态查询脚本（按客户端请求 ID 查外部系统当前状态）
- 指标：参数错误重试次数 = 0（不可重试类必须直接失败并回传原因）；503 重试次数 ≤ 3 且带幂等键；不确定态转查询的比例 = 100%；重复扣款率 = 0

测试五：端到端重试压力与幂等性测试

- 目的：在接近真实负载的条件下验证整套重试机制不会击穿外部系统

- 操作：跑 1000 次端到端工具调用流程，随机在三个阶段各注入 10% 的故障（模型重规划、进程崩溃、网关超时），流程结束后对比防重表、外部系统流水与数据库快照三方是否一致
- 工具：pytest + locust (<https://locust.io>) 做并发压测 + 数据库快照对比脚本 + Prometheus (<https://prometheus.io>) 观察重试次数与超时率
- 指标：端到端一致率 = 100%（三方对账零差异）；平均重试耗时 < 2 秒；重试引发的 P99 延迟增幅 < 30%；重试风暴导致的外部系统错误率上升 = 0

四、常见坑点

只给执行器加幂等，没给模型侧生成去重

最常见的漏法：状态机、幂等键都做了，但

Agent

在同一个业务意图上因为结果不满意而重新规划，两次生成了相同的工具调用参数。执行器因为调用来源不同、请求 ID 不同而去重失败，业务被执行两次。正确做法是对工具调用本身做签名（工具名加规范化参数），在生成层就拦掉重复意图，执行器幂等只是第二道防线。

把不确定态当失败直接重试

连接中断和网关超时是「不确定态」——请求可能已经到了外部系统并且已经生效，只是响应丢了。此时重试等于再执行一次；而按参数错误处理直接放弃，又会丢单。唯一安全的路径是先按客户端请求

ID

查询外部系统当前状态：查到了就采用查询结果，查不到才允许带原幂等键重试。把这两种错误混在一个重试策略里，是生产事故的高发区。

重试没有退避，也没有分类

无间隔立即重试会把下游的瞬时过载放大成雪崩：下游已经 503，上游还在每秒重试十次。同时必须做错误分类——4xx 参数错误重试永远不会成功，只是浪费时间和配额；5xx

与超时才允许指数退避重试；不确定态转查询。缺少分类的重试策略，在演练中表现为「故障恢复时间比不重试还长」。

五、延伸阅读

- OpenAI 函数调用文档: <https://platform.openai.com/docs/guides/function-calling>

- Anthropic 工具使用文档: <https://docs.anthropic.com/en/docs/build-with-claude/tool-use>

- OpenAI Cookbook:

高效可靠的工具调用模式:

https://cookbook.openai.com/examples/how_to_call_functions_with_chat_models

- idempotency-keys (Stripe 幂等键设计实践) : <https://idempotency-keys.up.railway.app>

- WireMock — 故障注入与外部系统替身: <https://wiremock.org>

- respx — Python HTTP 调用拦截与 Mock: <https://github.com/lundberg/respx>

- OWASP GenAI Security Project: <https://genai.owasp.org/>