方案一

> **Postgres 发号 + 占号 + Redis 按 submitter 加锁（可降级）+ receipt 异步终态**

工作原理

Redis实现（可选锁方案）

> 选择的理由：为了解决同一个 submitter 在高并发下，大量请求同时进入 DB 导致行锁竞争、阻塞/超时（甚至拖垮连接池）的问题。

- 使用`SET NX EX(PX)`获取 Redis 锁
- 使用Lua脚本保证"校验owner+删除锁"的原子性（`GET==owner`才`DEL`）
- 业务处理完释放锁 / 使用事务提交和回滚的钩子函数中释放锁，避免事务未提交就放锁

PostgreSQL实现（主方案）

- 使用数据库行锁加锁获取nonce游标，如果没有就初始化该 submitter 的 nonce 游标。
- 分配超时回收
- 优先复用最小 gap nonce；如果没有可复用的 gap nonce 则使用`next_local_nonce`并将其自增回写
- 成功获取nonce后携带nonce处理业务
- 成功后`markSubmit`→`submitted`（同txHash幂等）；
- 失败/放弃`markRecyclable`→`RECYCLABLE`

receipt 异步闭环（终态）

- 业务成功只 markSubmitted 写入 txHash，仍保持 RESERVED
- 后台轮询查链是否达到确认条件；达到确认条件后 markUsed
- 如果失败/链分叉导致的回滚

故障转移机制（代码实际）

- 仅支持"Redis失败则降级为纯DB"这一种：由`nonce.degrade-on-redis-failure`控制
- 连续失败N次切换 / 定时探活 / Redis恢复自动切回

优点

- 强一致性：nonce 分配在数据库事务里完成，天然避免了数据不一致问题

- 实现简单且易于维护和排错。

缺点

- DB 仍可能成为瓶颈，并发上限会被 DB 吞吐限制

方案二

redis发号 + 数据库占号

工作原理

Redis实现（发号主链路）

- 使用INCR命令原子性生成递增序列号（nonce/seq）
- Redis仅负责“发号”，不承担“占号/使用中”状态存储
- 生成失败（Redis不可用）时进入PostgreSQL降级发号链路

PostgreSQL实现（占号主链路）

- 在数据库写入“占号记录”，用**唯一索引**约束nonce不可重复占用
- 使用事务保证“占号写入”的原子性
- 并发冲突时（唯一索引冲突）通过**重试**获取下一个Redis序列号再尝试占号
- 定时清理超时占用，避免异常未释放导致永久占用

一致性与取舍（方案特点）

- “发号（Redis）”与“占号（DB）”是跨系统两步：若Redis已INCR成功但DB占号失败，DB会产生**号段空洞**
- 优点：占号状态在DB可审计、可查询、可恢复；并发安全主要依赖DB唯一约束
- 缺点：每次占号都要落库，性能受DB写入能力约束；跨系统两步带来空洞与重试成本

故障转移机制

- Redis连续失败N次后切换到PostgreSQL，发号 + 占号全走DB
- 定期探活Redis（如每5秒一次），恢复后自动切回“Redis发号”
- PostgreSQL不可用时无法完成占号直接失败，避免产生“已发未占”的不可控状态

优点

- redis发号可以原子性生成递增序列号，避免使用分布式锁 / 数据库锁

缺点

- “发号（Redis）”与“占号（DB）”是跨系统两步：若Redis已INCR成功但DB占号失败，DB会产生**号段空洞**
- 每次占号都要落库，性能受DB写入能力约束；跨系统两步带来空洞与重试成本。

方案三

redis实现发号 + 占号；异步刷盘

工作原理

Redis实现（主方案）

- 使用`INCR`命令原子性生成序列号
- 使用`SET NX EX`标记nonce为使用中
- 使用Lua脚本保证"获取+标记"的原子性
- 释放时删除标记，使nonce可重用

PostgreSQL实现（降级方案）

- 使用`SELECT FOR UPDATE`行锁保证并发安全
- 使用数据库事务保证原子性
- 使用唯一索引防止nonce重复使用
- 支持nonce的释放和重用

故障转移机制

- 连续失败 N 次后切换到PostgreSQL
- 每 N 秒检测一次Redis是否恢复
- Redis恢复后自动切回

优点

- 性能/吞吐更高、延迟低。
- Redis 横向扩展（分片/集群）后，发号能力可随资源线性提升

缺点

- 主从复制是异步的：你在主上成功写入，不代表从已同步；故障切换可能带来回滚窗口。
- 一致性与可靠性风险更大，处理不好会从“性能问题”升级成“一致性问题”（比如 Redis ↔ DB 可能会由于各种原因导致的刷盘没跟上，导致重复发号/重复占号）

方案四

> **GCP按照submitter路由请求，PostgreSQL 权威 + 租约栅栏（fencing_token）；本地缓存加速；后台异步推进交易状态**

工作原理

worker-queue模式（ 节点内串行）

1) 正确性边界：PostgreSQL + Fencing

- **每个 submitter 只能有一个 leader 节点写入**（租约 lease）。
- **所有关键写入都带 token 校验**（owner、未过期、token 匹配）。
- **写入更新行数=0 就视为被“栅栏拒绝”**：旧节点迟到回写不会覆盖新节点。

2) 发号：max(chain, cache, db)

- **db 游标**：记录下一次可分配 nonce。
- **chain 兜底**：用链上 pending nonce 防止“分低了”。
- **cache 加速**：节点内缓存下一次 nonce，减少查库/查链。
- **批次失败就清缓存**：避免“缓存推进但 DB 没提交”带来错号。

3) 交易闭环：提交≠终局（异步推进）

- **Writer**：写入交易记录（含 nonce），异步提交拿到 txHash。
- **ReceiptChecker**：后台轮询回执；查不到就延迟再查，出错就退避。
- **FinalityManager**：回执到手后推进终局（确认数够/检测分叉与否）。
- **ResubmitScheduler**：到期就重提；先“占位/claim”再发链，避免多节点重复动作。
- **StuckHook**：卡住时让业务决定策略（继续等/重提/标记 STUCK/补救动作）。

PostgreSQL实现（降级方案）

- 使用数据库行锁加锁获取nonce游标，如果没有就初始化该 submitter 的 nonce 游标。
- 分配超时回收
- 优先复用最小 gap nonce；如果没有可复用的 gap nonce 则使用`next_local_nonce`并将其自增回写

- 成功获取nonce后携带nonce处理业务
- 成功后`markSubmit`→`submitted`（同txHash幂等）；
- 失败/放弃`markRecyclable`→`RECYCLABLE`

优点

- **一致性强**：旧节点写入能被硬拒绝，不会乱写状态。
- **性能可控**：worker 分片 + batch + cache，减少冲突与 DB 往返。
- **可运维**：交易从创建到终局都有状态与"完成列表"可轮询。

缺点

- 业务需要立马拿到 txhash，这种异步处理没有办法与项目现有实现兼容
- 需要路由规则的支持

方案五

> **本地缓存 + 号段（segment）发号；批量刷盘；本地锁；GCP 按 submitter 路由**

工作原理

本地号段实现（主方案）

- GCP 层按 `submitter` 路由到固定实例，确保同一 submitter 的请求天然"单写入点"
- 实例内为每个 submitter 维护一个 **号段缓存**：`[start, end)` + `next`
- 发号时走 **本地锁（per submitter）**：返回 `next++`，号段用完再申请新号段
- 号段申请使用 **批量读写**（一次拿一段）：减少 DB 往返
- 占号/终局状态在内存里先记账（inflight/used），再 **异步批量刷盘**（合并写）

故障处理机制

- 实例重启恢复 next 时，取 max(本地缓存快照, DB 已刷盘最大, 链上最新 nonce) 作为基线
- 路由漂移/扩缩容：靠"同 submitter 固定路由"减少并发写；如发生双写，以 DB 的原子提升为准（可能出现跳号）
- 刷盘积压：触发背压（暂停申请新号段/降速），避免内存无限增长

优点

- 发号几乎全在本地：**延迟低、吞吐高**，数据库访问大幅减少（按号段粒度）。

- 批量刷盘：写放大更小，适合高并发 submitter 场景。

缺点

- 强依赖路由假设：submitter 路由不稳会导致**双发/乱序风险上升**，需要额外保护，
- 乱序情况下冲突带来的频繁缓存失效和重建反而会拖慢性能。

> tips：号段本身不吃内存，吃内存的是“异步刷盘前的状态堆积”；必须配套容量上限、过期清理和背压策略。

回执 + 确认数 / 终局

> - 区块通知驱动确认；异步查回执；达到确认阈值后判定最终
> - 数据对齐，避免数据库nonce滞后
> - 错误分层要做对：区分“暂时查不到”（未上链/节点未同步） vs “真实错误”（节点故障/限流/参数错），否则会误判失败或引发重试风暴。
> - 链头不稳定与可回退：明确“未终局可回退”；遇到重组要能重算并覆盖旧结果，而不是只会追加。

工作原理

回执获取（主流程）

- 链上节点持续推送“新区块/区块哈希”通知
- 一旦发现某笔交易“进入了某个区块”，就触发一次回执查询
- 回执查询是**异步工作池**处理：不会阻塞主流程
- 回执暂时查不到：视为“尚未可见”，继续等待后续触发

确认数推进（Finality 判定）

- 拿到“交易所在区块”后，从该区块往后统计连续的后继区块数
- 只认可“父子关系能对上”的链路；一旦对不上，认为可能发生分叉/重组，停止本轮推进
- 当确认数达到阈值 N 时，判定“已足够稳定”（final），并输出最终通知
- 确认阈值为 0 时：拿到定位信息即可直接视为最终

分叉/重组处理

- 如果链路发生变化，会发出“分叉发生”的信号
- 下游收到后应当把之前保存的确认链视为可被替换，而不是简单追加
- 对“稳定区块流”（用于事件流的已确认区块）若检测到不稳定，采用“清空视图后重建”的简单策略保证正确性

持久化策略（性能取向）

- 回执与确认链的写入通常是**异步批量落库**，降低请求链路延迟与数据库压力

优点

- 对链上分叉友好：确认链按父子关系校验，能识别重组并纠正结果
- 吞吐好：回执查询异步化，多 worker 并行，不阻塞主处理
- 输出有序：尽量按链上顺序交付“确认推进/最终”事件

缺点

- 最终性有天然延迟：必须等待 N 个后续区块
- 对节点依赖更强：需要额外查询区块/回执，节点不稳定时会增加重试与延迟
- 短暂不一致窗口：在“未最终”阶段结果可能变化（尤其遇到重组时）