从 For You 到

分布式系统

读 x-algorithm，亲手构造、破坏、证明并运营一个推荐服务

推荐 · 并发 · 状态 · 一致性 · 演化 · 测试 · 运维

唯一代码依据：x 团队原始公开仓库

源码基线：`0bfc2795d308`

目录

1 导读：你将亲手建立证据 4
1.1 开始之前：建立你的工程证据簿 4
2 第一部：建立一次有边界的请求 7
2.1 间章 A：先建立系统模型，再谈分布式 7
2.2 用户刷新 For You 时，请求属于谁 12
2.3 把推荐请求拆成可证明的 Pipeline 14
2.4 间章 B：RPC、时间与失败语义 16
2.5 多个 Source：并行、部分失败与压力放大 21
2.6 时间、Deadline、取消与重试预算 23
3 第二部：让系统真正做推荐 25
3.1 间章 C：数据契约、身份、版本与新鲜度 25
3.2 Query、Candidate 与数据契约 31
3.3 过滤、安全与业务不变量 33
3.4 间章 D：召回不是“小号排序” 35
3.5 Phoenix Retrieval：先决定谁值得精读 40
3.6 间章 E：排序分数不是用户价值 42
3.7 Phoenix Ranking：把产品目标变成分数 47
3.8 Selection、Blending 与产品约束 49
4 第三部：状态、事件与异步工作 51
4.1 间章 F：先写状态机，再写并发代码 51
4.2 Thunder：实时状态面对重复与乱序 56
4.3 间章 G：缓存是一份会过期的副本 58
4.4 缓存、Served History 与一致性承诺 63
4.5 Grox：任务图、资源池与失败隔离 65
4.6 间章 H：响应成功之后，工作还没有结束 67
4.7 Side Effects：响应之后怎样不忘记 73
5 第四部：规模、复制与在线演化 75
5.1 间章 I：排队论不是算术题——从等待预算到过载闭环 75
5.2 资源预算、排队与过载保护 80
5.3 间章 J：分片不是取模——路由、热点与在线迁移 82
5.4 分片、热点与在线迁移 87
5.5 间章 K：副本不是备份——复制、共识与 Fencing 的责任边界 89
5.6 复制、Failover、Epoch 与 Fencing 94
5.7 间章 L：发布不是瞬间——协议、数据、配置与模型的在线演化 96
5.8 协议、数据与配置的在线演化 102
6 第五部：证明、运营与恢复 104
6.1 间章 M：测试不是绿灯——怎样为分布式系统建立可反驳的证据 104
6.2 用风险驱动测试建立分布式证据 110
6.3 间章 N：看见、承诺与恢复——可观测性和 SRE 的完整闭环 112
6.4 可观测性、SLO、容量与事故响应 119
6.5 间章 O：系统活着不等于业务恢复——灾备、安全、隐私与多租户 121
6.6 恢复、数据修复、安全与多租户 128

6.7 间章 P：一次发布不是一个结论——可信实验、灰度与可逆变更 ... 130
6.8 毕业项目：像真正上线一样增加一个 Source ... 136
7 结语：成为会保护系统的人 ... 137
8 附录：源码、能力与验收回查 ... 138
8.1 附录：能力、源码与项目回查 ... 138

1 导读：你将亲手建立证据

这不是一套随书代码的使用说明，也不是把仓库文件依次翻译成中文。全书只有两个输入：这本书，以及固定版本的 `x-algorithm` 原始公开仓库。你会沿着小林的一次 For You 请求读真实代码，再从空目录开始，逐章构造自己的推荐服务。这个项目不是教材依赖，而是你的学习成果。

每章都遵循同一个循环：在原仓定位证据，提取输入、输出、不变量与失败语义；在自己的系统中实现一个最小但完整的能力；主动制造延迟、错误、重复、乱序、崩溃或过载；最后用测试、日志、指标、容量和恢复记录证明判断。我们不要求你的类名或目录与某份标准答案相同，只要求系统行为经得起验收。

> **Tip：这本书默认你已经会普通 Web 后端**
>
> 你应能独立完成常规接口、数据库、测试、部署和排障。书中不会重复讲 CRUD、HTTP 状态码或容器基础；遇到 Rust/JAX 语法障碍时，会用 Tip 给出最短阅读路径。真正的新内容是：远程调用怎样失败，时间怎样失真，状态怎样分叉，版本怎样共存，以及怎样证明系统可以恢复。

> **Tip：间章不是第二套作业**
>
> 间章是插在工程章之前的大型 Tip，只负责建立模型、推导和反例。首次阅读只需回答每篇自检中最关键的两三题，不要另建项目或提交新产物；所有实现、故障实验和验收都并入紧随其后的工程章。

1.1 开始之前：建立你的工程证据簿

1.1.1 现场：克隆成功不等于系统可运行

小林刷新一次 For You，背后会经过 Home Mixer、Candidate Pipeline、Thunder、Phoenix、缓存和后台写入。公开仓库让我们看见核心设计，却没有交付 X 的内部 crate、proto、配置、集群和生产数据。如果把“源码公开”误读成“完整生产环境可以本地启动”，后面每次失败都会变成无效的安装排查。

本书采用另一条更诚实的路线：原仓库只负责提供真实证据；你从空目录建立一个自己的系统，用它验证通用语义。你的实现不必复刻 X，也不得被描述成 X 的运行结果。

> **新手 Tip：先接受不完整**
>
> 能分清“代码事实、合理推断和未知”是分布式工程的第一项能力。生产仓库、事故日志和监控面板同样永远不完整。不要等到掌握全部背景才开始；先记录当前证据能支持什么。

1.1.2 在原仓定位：先画地图，不钻文件

确认当前源码版本：

```
git rev-parse HEAD
git status --short
```

本书基线是 `0bfc2795d308f90032544322747caacd535f75ae`。然后只打开六个入口：

- `README.md`：公开架构声明；
- `candidate-pipeline/candidate_pipeline.rs`：通用流水线控制流；
- `home-mixer/for_you_server.rs`：For You 外层入口；
- `thunder/thunder_service.rs`：实时内容查询边界；
- `phoenix/README.md`：召回与排序模型说明；

- `grox/engine.py`：异步内容理解任务入口。

给每个目录只写三句话：它拥有哪类状态，向谁提供什么，失败会影响当前请求还是未来状态。暂时不要抄组件清单。

1.1.3 原理与边界：四种证据不能混写

本书使用四个证据等级：

等级	含义	可以怎样表述
可运行	原仓脚本或测试已在声明环境下实际执行	"官方测试验证了若干形状与遮罩结构不变量。"
可计算	结果能由公开公式和固定输入手算	"按公开公式，B 的相似度高于 C。"
可审计	只能从公开控制流或调用点判断	"代码在此处并行调度，公开快照未展示统一 deadline。"
读者实现	你按书中契约构造的概念系统	"我的实现通过了取消测试，不代表 X 生产系统采用相同机制。"

Rust 主链没有公开 Cargo workspace，并引用内部 `xai_*` 依赖；Grox 缺少部分模块；Phoenix 提供单阶段 demo、测试和依赖声明，能否在具体平台运行仍须实际验证，完整 pipeline 还需要 Git LFS artifact。后续每章都必须守住这些边界。

1.1.4 构造：从空目录建立唯一项目

在仓库之外或你自己的学习分支中创建一个空项目。语言由你选择，但核心路线只允许标准库。第一份提交只包含：

- `README`：系统目标和非目标；
- 一个可以接受 `viewer_id`、`request_id` 的入口；
- `Candidate` 与 `FeedItem` 两个最小数据结构；
- A—F 固定案例；
- 一份结构化 JSON 日志；
- 一条最简单的自动化测试。

A—F 的含义全书不变：A 是关注作者帖子，B 是召回内容，C 是旧缓存内容，D 来自被屏蔽作者，E 缺失安全字段，F 是广告。此时不必实现算法，只要一次请求能留下稳定、可阅读的记录。

> **验收契约：第一里程碑**
>
> 给定 `viewer_id=xiaolin` 和固定 `request_id`，程序返回一个合法但可以为空的 feed；日志必须包含同一 request ID、开始/结束事件和总耗时字段。测试不能依赖真实墙钟的精确毫秒数。

1.1.5 破坏并证明：先让观察本身失败

主动制造三种错误：漏写 request ID、让日志输出不可解析、让测试依赖一次真实 `sleep`。分别记录为什么这些错误会让后续故障无法定位。然后修复，并建立你的"工程证据簿"，以后每章追加：

源码锚点 | 本章契约 | 固定输入 | 故障 | 测试 | 指标 | 未知

证据簿可以是一个 Markdown 文件，但答案必须来自你的实现和原仓源码，而不是复制本书结论。

1.1.6 验收：从第一天建立分布式 DoD

- 能解释原仓哪些部分可运行、可计算、可审计；
- 读者项目与原仓库明确分离；
- 一次请求拥有稳定身份和机器可读日志；
- 固定案例可重复构造；
- 已写下第一个测试和第一个已知未知；
- 没有把“程序返回”误写成“推荐系统正确”。

后续每项能力都要回答十个问题：正确性、时间、失败、资源、状态、演化、隔离、证据、恢复和用户影响。

2 第一部：建立一次有边界的请求

从入口、Candidate Pipeline 和多路 source 出发，建立请求身份、阶段账本、并行 fan-out、deadline、取消和重试预算。第一部结束时，你的系统仍可以是单进程，但已经不再把远程依赖当成本地函数。

2.1 间章 A：先建立系统模型，再谈分布式

大型 Tip · 间章：你画出的边界，决定你能看见哪些错误

分布式系统不是“很多机器”，而是多个拥有独立状态、独立资源与独立失败命运的参与者，通过不可靠通信共同完成一个目标。分析任何实现之前，先写清参与者、状态、消息、时间和不变量。否则，“加重试”“上缓存”“做高可用”都只是没有坐标的动作。

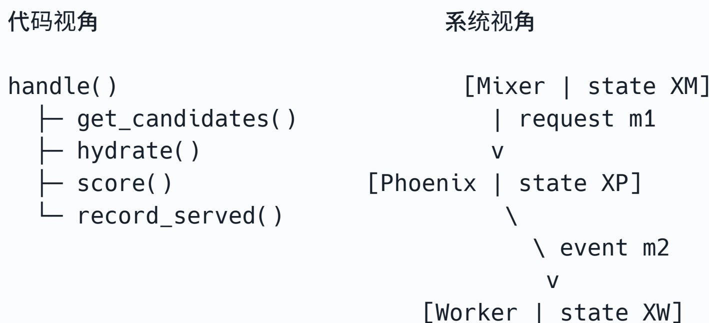
2.1.1 1. 五个基本对象

把一次 For You 请求抽象为系统 $S = (P, X, M, T, I)$：

- P（participants）是参与者，例如 Mixer、Phoenix、Thunder、缓存和 side-effect worker；
- X（state）是各参与者持有的状态，包括内存、文件、数据库行、队列位置与配置版本；
- M（messages）是跨边界传递的请求、响应、事件和确认；
- T（time）是本地时钟、deadline、事件时间与消息延迟；
- I（invariants）是在所有允许执行中都必须成立的性质。

这里最重要的是“不共享命运”。同一进程中的两个对象通常一起崩溃；两个服务可能一边存活、一边失联；两个副本甚至会同时认为自己拥有写权限。进程、机器、可用区和团队都可以形成不同的故障域。

图：从代码调用图切换为分布式系统图



左图容易暗示“调用一定返回”；右图迫使我们询问：
消息能否丢失？状态能否分叉？谁先崩溃？恢复后知道什么？

2.1.2 2. 本地调用与远程调用不是同一种抽象

在常见语言级调用模型中，本地函数完成时通常被观察为返回值或异常；它仍可能永久阻塞、死锁或随进程崩溃。远程调用额外引入执行结果未知和多参与者部分成功：下游可能未执行、已执行但响应丢失、仍在执行，或调用方已放弃且不知道结果。

因此，远程调用的结果集合不应只写成 `Ok/Err`，而应在设计文档中区分：

- 明确成功：收到可验证的成功响应；
- 明确拒绝：下游确认前置条件不成立且没有产生目标效果；
- 明确失败：处理已发生但契约明确回滚或没有留下目标效果；
- 结果未知：连接中断、超时或响应丢失，无法判断是否生效；

- 部分成功：多个参与者中只有一部分完成。

Tip：先问事实，再决定策略

“超时后重试”是策略，不是事实。先写下超时发生时你究竟知道什么，再判断操作是否幂等、是否可查询、是否需要相同幂等键，以及剩余 deadline 是否允许重试。

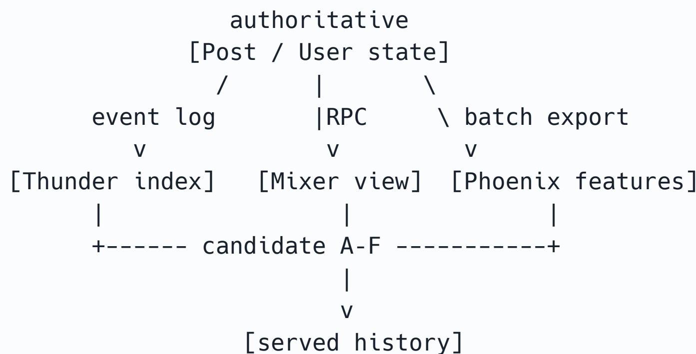
2.1.3 3. 状态所有权：谁有权说它是真的

同一个业务概念常有多个表示：帖子原始记录、Thunder 的实时索引、Phoenix 的向量、缓存中的特征、Mixer 请求中的快照。它们不是五份同等权威的数据。

为每类状态声明：

1. 权威来源（source of truth）是谁；
2. 哪些是派生状态，可否重建；
3. 谁可以写，谁只能读；
4. 版本或顺序如何比较；
5. 多旧仍可接受；
6. 损坏后怎样发现并修复。

图：一条候选数据的所有权与派生关系



箭头表示“可由谁推导”，不是“永远同步”。

如果 Phoenix 的作者特征旧了 30 秒，可能只是质量下降；如果安全状态旧了 30 秒，可能越过产品红线。所谓一致性需求不能脱离字段语义单独决定。

2.1.4 4. 不变量、前置条件与目标不要混写

不变量：任何允许执行中都必须成立，例如同一响应中帖子 ID 不重复。

前置条件：操作成立所需条件，例如 scorer 只能消费已经 hydration 的候选。

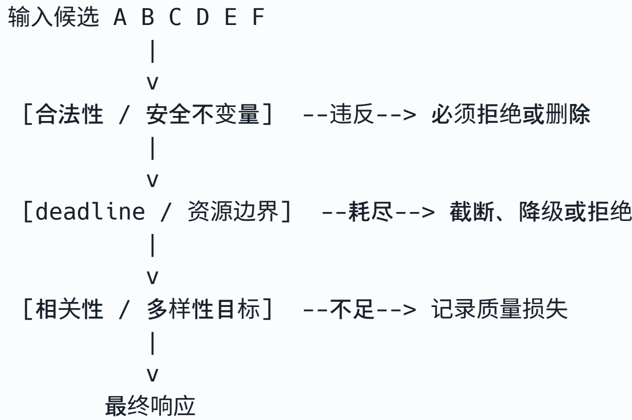
后置条件：操作成功返回时承诺成立，例如 selected candidate 都具有最终分数。

优化目标：希望尽量改善但不保证每次成立，例如 NDCG 更高或内容更多样。

SLO：在统计窗口内的服务水平目标，可以是内部目标；例如 99.9% 请求在 120 ms 内返回至少 10 条候选。安全穿透等硬不变量不因 error budget 而获得许可；外部合同承诺通常属于 SLA。

反例：把“每次返回 20 条”写成不变量，会迫使系统在候选不足时填入不安全内容；把“尽量不重复”写成目标，又会让一个本可严格保证的性质变得含糊。正确分类会直接影响失败时允许怎样降级。

图：硬边界与软目标的裁决顺序



软目标不能购买穿越硬边界的权利。

2.1.5 5. 安全性与活性

分布式性质常分为两类：

- 安全性（safety）：“坏事永不发生”，例如同一个 side effect 不会产生两次业务效果；
- 活性（liveness）：“好事最终发生”，例如暂时失败后事件最终被处理。

“不丢不重”同时包含两种要求，通常不能靠一句 exactly-once 实现。去重表可以保护安全性，但若永远不重试就没有活性；无限重试追求活性，却可能被 poison message 永久占用。工程设计必须说明两者的边界和取舍。

> **源码证据：原仓中的系统边界**
>
> `candidate-pipeline/candidate_pipeline.rs` 展示阶段编排；`candidate-pipeline/source.rs`、`candidate-pipeline/scorer.rs` 等 trait 展示组件契约；Home Mixer、Phoenix、Thunder 与 Grox 则提供不同的状态和执行边界。公开快照能支持架构取证，但缺少内部依赖，不能据此声称整条生产系统可以本地运行。

2.1.6 6. A–F：用候选生命史检查模型

假设六个候选经历如下过程：

候选	最初来源	关键变化	最终命运	要保护的事实
A	Following	补全并高分	入选	来源与身份不丢失
B	Phoenix	关注网络外的 Rust 内容	排序后入选	模型/索引版本可追踪
C	Cache	旧篮球内容且已展示	过滤	缓存年龄与 served history 可追踪
D	Following	作者已被小林屏蔽	过滤	权限依赖失败策略明确
E	Phoenix	安全字段未知	拒绝	未知不能伪装成安全
F	Ads	需要 safe gap	延后插入	商业目标不能越过约束

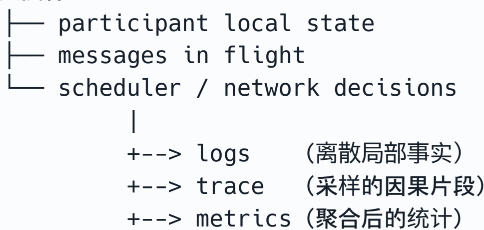
这张表不是测试数据装饰，而是一个小型状态空间。每增加缓存、重试或副本，都重新追踪 A–F：它们是否可能换错身份、被重复计数、读取旧状态、越过过滤或产生无法解释的结果？

2.1.7 7. 执行、历史与可观测证据

一次执行可以看作事件序列 $H = e_1, e_2, ..., e_n$。事件至少携带参与者、操作、对象 ID、逻辑顺序和结果。日志只记录观察到的局部事实；trace 把一部分因果关系连接起来；指标聚合大量执行。三者都不是系统真相本身。

例如看到 `source_timeout=1`，只能证明调用方按本地规则判定超时。它不能证明 Phoenix 崩溃，也不能证明 Phoenix 没有返回候选。可靠复盘应把“观测”“推断”“待验证假设”分开写。

图：从执行到证据，而不是从图表到真相



证据有损，因此结论必须标注置信边界。

2.1.8 8. 故障模型决定结论有效范围

常见模型从弱到强包括：进程 crash-stop、进程 crash-recovery、消息丢失/重复/乱序、网络分区、时钟漂移、磁盘损坏、任意行为。一个只模拟返回错误的实验，不能证明系统能处理进程在写入后响应前崩溃。

写实验时明确三件事：注入点、允许观察到的中间状态、恢复动作。不要笼统地写“模拟故障”。

> **Tip：最小模型也可以很有价值**
>
> 本地多进程无法复刻真实机房，但可以精确演示结果未知、乱序、重复、旧 writer 和资源耗尽。诚实说明模型边界，比搭建一堆中间件却不清楚验证了什么更专业。

2.1.9 9. 建模顺序

面对任何新组件，按下面顺序写一页纸：

1. 用户可见目标与不可越过的红线；
2. 参与者及故障域；
3. 权威状态、派生状态和写入者；
4. 消息及其重复、丢失、乱序可能性；
5. 时间预算和资源预算；
6. 安全性、活性与质量目标；
7. 可观测证据和恢复入口；
8. 当前故障模型没有覆盖什么。

这页模型不是一次性文档。随着系统从单进程变成多进程、从单副本变成复制、从单版本变成滚动升级，应持续修改它。

2.1.10 自检：你是否真的拥有一个系统模型

1. timeout 后，你能否判断下游一定没有执行？为什么？
2. Phoenix 向量、Thunder 索引和帖子记录中，哪些是权威状态，哪些可重建？
3. “返回足够多内容”在什么条件下是 SLO，什么条件下不能成为不变量？
4. 为 side effect 分别写出一条安全性和一条活性要求。
5. A–F 中哪个候选最适合检验未知安全状态？哪个适合检验新鲜度？
6. 一张成功率图能证明哪些事实，不能证明哪些事实？
7. 你的本地实验采用了什么故障模型？结论不能推广到什么情形？

检查点：带着模型进入后文

首次阅读不需要提交新图。进入下一工程章前，应能口头指出参与者、状态所有权、消息边界和 A–F 生命史，并为关键结论标出故障模型。后续每一种技术——deadline、缓存、队列、复制、fencing——都只是对这个模型中某个风险的具体回答。

2.2 用户刷新 For You 时，请求属于谁

2.2.1 现场：一个刷新动作穿过两层流水线

小林看到的是一个 feed，服务端却先在内层产生帖子候选，再在外层与广告、推荐关注等 item 混排。若 request ID、用户身份、参数快照和实验分桶在中途丢失，同一次刷新就可能被不同组件解释成不同请求。

> **Tip：先找边界，再追函数**
>
> 阅读大型服务时，不要从 `main` 顺序展开所有初始化。先找网络入口、内部 query 构造点、下游调用和响应转换四个边界，再补中间细节。

2.2.2 在原仓定位：沿两层入口追一次请求

依次阅读：

- `home-mixer/for_you_server.rs`：外层 For You 请求；
- `home-mixer/sources/scored_posts_source.rs`：把帖子排序服务作为 source；
- `home-mixer/scored_posts_server.rs`：内层帖子请求；
- `home-mixer/server.rs`：QueryBuilder、参数和 trace 上下文；
- `home-mixer/models/query.rs`：内部 query 的长期字段。

画出 `client -> ForYou -> ScoredPostsSource -> ScoredPosts -> pipeline -> FeedItem`，并标注 viewer ID、request ID、产品 surface、参数和 datacenter 在哪里产生、转换或读取。

2.2.3 原理与边界：控制流、数据流与请求上下文

控制流说明谁调用谁；数据流说明字段怎样变化；请求上下文说明同一次决策共享哪些冻结信息。三者不能用一张“服务架构图”代替。

请求参数应在入口冻结。否则 source A 在配置更新前读取旧值，source B 在更新后读取新值，同一次请求就出现不可复现的混合决策。实验分桶、模型版本和安全策略也属于请求上下文，而不是随用随查的全局变量。

协议成功、pipeline 成功和产品成功是三种状态：服务可以正常返回，同时某个 source 降级、候选不足或后台事件失败。不要用一个 `ok` 布尔值吞掉这些差异。

2.2.4 构造：建立入口、防腐层和参数快照

在你的项目中加入：

- 外部 `FeedRequest` 与内部 `RecommendationQuery`；
- 一个只在入口运行的 QueryBuilder；
- request ID 生成或校验；
- 参数快照与明确版本号；
- 外层 `FeedItem` 和内层 `Candidate` 的转换边界；
- 每个阶段共享的 request context。

固定输入包含小林、surface=`for_you`、参数版本 `v1`。在请求处理中途修改全局参数，证明当前请求仍使用入口快照，下一次请求才看到新值。

> **验收契约：身份不掉线**

> 同一请求产生的每条阶段日志必须包含相同 request ID；每次下游尝试还要有独立 attempt ID。参数版本在一次请求中不可变化，响应必须报告降级状态和最终候选数量。

2.2.5 破坏并证明：制造上下文断裂

至少测试：

- `viewer_id` 缺失或非法；
- 客户端重复或碰撞 request ID，系统能够标记但不把它误当业务幂等键；
- 内层调用遗漏 request ID；
- 参数在处理中途切换；
- 内层返回成功但候选为零；
- 响应转换丢失 candidate 来源。

不要只断言 HTTP/RPC 状态；同时检查阶段日志、参数版本、候选账本和降级字段。

2.2.6 验收：请求有了唯一身份

- 外部协议与内部模型分离；
- 请求上下文只在入口构造并冻结；
- request ID 的信任边界与唯一作用域明确，request ID、attempt ID、trace/span ID、幂等键职责分开；
- 内外两层数据边界可画出；
- 空结果与协议失败不再混为一谈；
- 入口测试能够发现上下文断裂。

2.3 把推荐请求拆成可证明的 Pipeline

2.3.1 现场：一大段函数为什么迟早失控

推荐请求会依次执行 initial/dependent query hydration、拉取候选、candidate hydration、过滤、打分、选择、post-selection hydration/filter、截断并触发后台工作。把它们写进一个函数并不会减少复杂度，只会让阶段顺序、局部失败和指标责任变得不可见。

> **新手 Tip：trait 先读输入输出**
>
> 阅读 Rust trait 时，先忽略宏和生命周期，写下方法接收什么、返回什么、是否异步、错误放在哪里。能回答这四件事，就已经抓住了组件契约。

2.3.2 在原仓定位：`execute` 是整本书的目录

重点阅读 `candidate-pipeline/candidate_pipeline.rs`，再用下列文件核对组件接口：

- `candidate-pipeline/query_hydrator.rs`；
- `candidate-pipeline/source.rs`；
- `candidate-pipeline/hydrator.rs`；
- `candidate-pipeline/filter.rs`；
- `candidate-pipeline/scorer.rs`；
- `candidate-pipeline/selector.rs`；
- `candidate-pipeline/side_effect.rs`。

把 `execute` 还原成阶段表：输入类型、输出类型、是否可并行、失败怎样表示、记录哪些观测。宏不是主线；先读显式数据变化。

2.3.3 原理与边界：阶段既是合约也是故障域

阶段化带来三种价值：

- 数据责任：字段应该在哪一阶段产生；
- 执行责任：哪些组件可并行，哪些必须等待前序结果；
- 失败责任：局部错误是否允许继续，以及证据归谁记录。

同阶段并行不等于跨阶段乱序。hydrator 依赖 source 的候选，scorer 依赖 hydration，selector 依赖 score。把阶段顺序写成显式 orchestration，才能测试“顺序本身”这个不变量。

2.3.4 构造：建立最小 Pipeline 与阶段账本

在读者项目中定义最小接口：`QueryHydrator`、`Source`、`CandidateHydrator`、`Filter`、`Scorer`、`Selector`、`SideEffect`。先只实现每类一个组件，让 A—F 经过完整流程。

每个阶段记录：

```
stage | input_count | output_count | latency | status | error_kind
```

不处理候选数量的阶段使用 `null`，不要用 0 冒充“确实没有候选”。给候选附上稳定 ID、来源集合和 drop reason，使账本能解释它在哪里进入、变化和离开。

> **验收契约：流水线不是黑箱**

一次请求必须产生严格有序的阶段记录；最终 feed 中每个 item 都能追溯到 source；被删除候选保留阶段和原因；fatal error 可以终止执行，但必须补写 terminal/fatal 记录，不能让账本无声断裂。

2.3.5 破坏并证明：交换两个阶段

主动把 filtering 移到 hydration 前，或把 selection 移到 scoring 前。不要只观察程序是否报错，要比较候选数量、缺失字段、drop reason 和最终结果，说明为什么某些错误仍能返回“看起来合理”的 feed。

测试至少覆盖：阶段顺序、空 source、组件错误、候选身份对齐、账本完整性。延迟测试只断言非负和相对关系，不断言 CI 机器上的精确毫秒数。

2.3.6 验收：每次变化都有归属

- 七类教学组件拥有清晰接口，但没有把它们误写成原仓全部执行阶段；
- 阶段顺序可被自动化测试保护；
- A—F 拥有贯穿请求的生命史；
- 数量、身份、原因和错误都可归因；
- side effect 已有接口且不阻塞核心响应，但尚未因此获得持久性、可靠性或自动取消；
- 当前仍是单进程，远程边界将在下一章出现。

2.4 间章 B：RPC、时间与失败语义

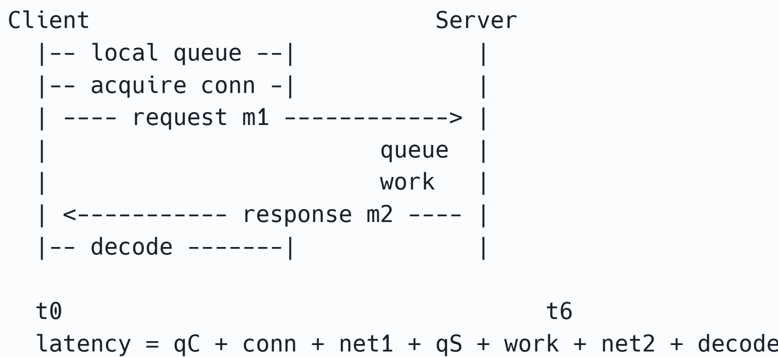
大型 Tip · 间章：远程调用最危险的谎言，是长得像函数

`await client.get()` 的语法会让人忘记：请求和响应是两条独立消息；客户端、网络、服务端可能在不同时间失败；timeout 只结束等待；重试会创造新执行。可靠 RPC 不是选一个库，而是定义预算、结果、幂等、取消、负载和证据。

2.4.1 1. 一次 RPC 实际发生了什么

把调用拆成至少六段：客户端排队、连接获取、请求发送、服务端排队、服务端执行、响应返回。所谓“RPC 延迟”是这些时间的和，任何一段都可能占满 deadline。

图：调用者看见一个 await，系统经历两条消息



```
Client                          Server
  |-- local queue --|                |
  |-- acquire conn -|                |
  | ---- request m1 ------------->  |
  |                        queue    |
  |                        work     |
  | <----------- response m2 ----   |
  |-- decode -------|                |

  t0                                    t6
  latency = qC + conn + net1 + qS + work + net2 + decode
```

如果指标只记录服务端 `work`，排队和连接池耗尽会成为盲区。如果 timeout 只覆盖读响应，连接获取可能无限等。预算必须覆盖调用方真正关心的完整区间。

2.4.2 2. Deadline、timeout 与剩余预算

timeout：某个局部操作最多等待多长时间。

deadline：整项工作不得晚于哪个边界完成。

剩余预算：当前时刻到 deadline 之间仍可支配的时间，并要扣除安全余量。

若入口预算为 $D = 120$ ms，在 Mixer 已耗费 18 ms，序列化和返回预留 12 ms，那么下游可分配预算至多为：

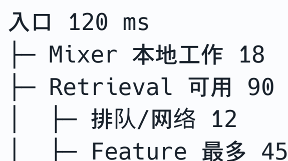
$$B_{\text{downstream}} \leq D - 18 - 12 = 90 \quad \text{ms}$$

若下游再串行调用 Feature Service，不能重新给它 120 ms。应从同一个 deadline 推导剩余预算：

$$B_i = \max(0, \min(L_i, \text{deadline} - \text{now} - \text{reserve}_i))$$

其中 L_i 是该依赖的局部上限。局部上限保护调用方，端到端 deadline 保护用户。

图：预算沿调用链递减



```
入口 120 ms
├─ Mixer 本地工作 18
├─ Retrieval 可用 90
│   ├─ 排队/网络 12
│   ├─ Feature 最多 45
```

```
│   └─ 合并结果 15
└─ 编码与返回预留 12

错误做法：每一层重新设置 timeout = 120 ms。
方案一：传播绝对 wall-clock deadline，但声明有界时钟误差。
方案二：传播相对预算并扣除保守余量；它避免比较 monotonic 原点，
        却不能单独精确计入消息已在网络中停留的时间。
最终强边界仍由上游自己的 deadline/timeout 执行。
```

2.4.3 3. 为什么 wall clock 不能计算经过时间

wall clock 会因 NTP 校时、人工修改或虚拟机迁移跳变。持续时间应由 monotonic clock 计算；wall clock 还承担事件时间、审计和跨节点过期。跨主机传递绝对 deadline 时，要说明时钟同步假设；传播相对预算时，则只能用保守余量近似在途时间，不能声称接收方重新计时仍严格等于入口边界。

反例：开始时 wall clock 为 10:00:01，校时后结束时间为 09:59:59，得到负延迟。另一个反例是两个节点相差 5 秒，用绝对时间判断 lease，旧主可能比新主多写 5 秒。

> **Tip：测试时间，不要测试睡眠**
>
> `sleep(100ms)` 后断言超时既慢又不稳定。把时钟、调度点或阻塞门抽象为可控 seam：测试推进假时钟、释放 barrier，再检查任务状态。真实时钟只留少量集成测试。

2.4.4 4. RPC 失败的知识边界

观测	可以确定	不能确定	典型动作
连接前被拒绝	未建立本次连接	服务端是否由别的请求改过状态	退避、换节点或降级
收到明确业务拒绝	该 attempt 未产生目标效果	未来重试是否会成功	按业务语义处理
读响应超时	调用方停止等待	服务端是否已执行	查询、幂等重试或标记未知
连接中途断开	响应不完整	请求是否完整到达	依赖操作语义裁决
收到成功响应	服务端声称成功	副本、side effect 是否均已完成	按响应契约继续

“收到成功”也有边界：如果服务端在 durable write 前就返回，重启后成功可能消失；如果成功只代表已入队，下游效果尚未发生。响应契约必须写明成功点。

2.4.5 5. 幂等不是“请求重复也没报错”

操作 f 的数学幂等是 $f(f(x)) = f(x)$。工程中的幂等通常指：用同一个业务意图和幂等键重复提交，不会产生额外业务效果，并能返回兼容结果。

需要区分：

- 读操作通常天然幂等，但可能每次看到不同版本；
- `set(value)` 可能幂等，`increment(1)` 通常不幂等；
- “插入唯一键”防止重复行，不一定防止重复外部通知；
- attempt ID 标识一次尝试，idempotency key 标识同一业务意图；
- 去重记录必须与业务写入处在可解释的持久化关系中。

图：相同意图，多次 attempt

```
request_id = R42, idempotency_key = serve:lin:feed:9001

attempt A1 ---> write effect ----X response lost
attempt A2 ---> lookup key ------> return prior result

若 A1 的 effect 与 key 不是原子提交：
  effect 后崩溃 -> A2 可能重复；
  key 后崩溃    -> A2 可能误认为已完成。
```

2.4.6 6. 重试预算与放大

假设入口 QPS 为 Q，每请求 fan-out 到 F 个依赖，每个调用平均尝试 A 次，则下游尝试速率约为：

$$Q_{\text{downstream}} = Q \times F \times A$$

小林的请求入口 800 QPS，60% 启用 Phoenix，Phoenix 平均 1.5 次尝试，则 Phoenix 承受 $800 \times 0.6 \times 1.5 = 720$ QPS，而不是 480 QPS。若三层各自最多重试两次，最坏执行数可达 $3^3 = 27$ 次。

指数退避减少连续压力，jitter 减少客户端同步，但都不会创造容量。重试必须同时满足：错误可能是暂时的、操作可安全重复、剩余 deadline 足够、全局/局部 retry budget 未耗尽、下游没有明确过载。

源码证据：原仓可观察到什么

从 `home-mixer/server.rs` 的 client 等待与 query 构造出发，再检查 `candidate-pipeline/candidate_pipeline.rs` 中 source、hydrator 和 side effect 的调度。公开代码可证明若干局部 timeout 或异步边界；除非调用链明确传递同一上下文，否则不能推断整条请求具有端到端 deadline、取消和重试预算。

2.4.7 7. 取消是一份协作协议

调用方取消等待，不代表：请求字节没有发出、服务端停止执行、底层库释放连接、子任务被回收。有效取消需要逐层协作：

1. 入口产生 cancellation signal；
2. orchestration 停止启动新工作；
3. 已启动任务在安全点检查信号；
4. client library 能中止或丢弃 I/O；
5. CPU 密集工作主动让出；
6. 不可取消的副作用转为受控后台工作；
7. 无论怎样退出都释放 semaphore、连接与缓冲区。

反例：`select` 返回最快的 source 后丢弃其余 future，但底层任务已经 `spawn`，仍持有连接和并发许可。请求延迟看似改善，容量却持续下降，这就是僵尸工作。

图：取消信号与资源释放链

```
deadline expires
       |
       v
[request token] ---> stop new fan-out
```

```
       |              cancel pending I/O
       |              release permits
       +-----------> child token
                           |
                           +--> CPU loop checks safe point
                           +--> irreversible effect records handoff
```

完成标准不是“调用方返回”，而是无价值资源停止增长。

2.4.8 8. Hedging、重试与 fallback

重试是在失败或超时后再次尝试；hedging 是首个尝试仍未完成时向另一副本发起并行尝试；fallback 是改用能力较弱但独立的路径。三者成本不同。

hedging 可以削减尾延迟，却会增加正常流量，并可能让最慢时刻变得更拥塞。它通常只适合幂等或可安全重复的读取，最好发往负载与故障高度不相关的副本，并需要取消输掉的请求。向同一个过载队列发第二次请求通常无益，只会加压。

fallback 也不能无限叠加：缓存 fallback 可能陈旧，默认 feed 可能违反个性化安全约束。每条降级路径都要有质量下限和独立指标。

2.4.9 9. A–F 的部分成功语义

固定候选身份不变：Following/Thunder 路径返回 A 与待过滤的 D，Cache 返回 C，Phoenix 预期返回 B 与安全状态未知的 E，Ads 返回 F。当 Phoenix 超时：

- 可以保留已明确取得的 A、C、D、F，但 D 仍须权限过滤，C 仍须 served-history 过滤；
- 不能把尚未返回的 B/E 当成普通空结果，事实是 Phoenix 结果未知；
- source coverage、候选数量和质量代理指标应记录降级；
- 如果安全依赖也超时，不能照搬相关性 source 的 fail-open 策略；
- F 的 safe gap 仍是硬约束，不能用广告填满候选缺口。

依赖	失败影响	可选策略	证据
Phoenix	召回覆盖下降	部分结果 + 质量降级	source status、coverage
Safety	可能越过红线	fail closed 或安全缓存	unknown reason、cache age
History	重复曝光	按产品契约降级	history version、duplicate rate
Ads	商业内容缺失	不插入广告	placement reason

2.4.10 10. 一份完整的 RPC 契约

每个远程边界至少声明：请求身份、deadline 传播、最大消息大小、成功点、错误分类、幂等键、重试所有者、取消行为、并发和队列上限、过载响应、版本兼容、日志/trace 字段，以及调用方允许怎样降级。

不要让客户端、服务端和运维手册各自猜一份不同答案。契约变化属于协议演化，需要混合版本测试。

2.4.11 自检：你会如何裁决一次失败

1. 读响应超时后，关于服务端执行你知道什么、不知道什么？
2. 120 ms 入口 deadline 为什么不能让每层都配置 120 ms timeout？
3. monotonic clock 和 wall clock 各自适合什么用途？
4. attempt ID、request ID、idempotency key 为什么不能合并？

5. 三层各重试两次时，最坏尝试数为什么是乘法增长?
6. 怎样证明取消后没有僵尸任务继续占用 semaphore?
7. Phoenix timeout 与 Safety timeout 为什么不能采用相同降级策略?
8. hedging 在什么前提下才可能降低尾延迟而非放大事故?

检查点：RPC 的验收不是收到 HTTP 200

把这些判断带入后续工程章：为 Mixer 到 Phoenix 的边界写出完整 RPC 契约，并用可控 barrier 演示明确成功、明确拒绝、结果未知和取消后释放资源。再计算重试放大；说明调用方总 deadline 如何强制约束整个调用，以及相对 budget、绝对 deadline、时钟误差和在途时间各自留下什么边界。能回答这些问题，才是在把远程调用当成分布式协议而非函数。

2.5 多个 Source：并行、部分失败与压力放大

2.5.1 现场：三位采购员为何不该排队出门

Following、Phoenix 风格召回和缓存彼此独立。串行调用会把 8、28、12 ms 累加成约 48 ms；在没有排队、连接竞争和额外开销的简化条件下，并行可以接近最慢一路的 28 ms。并行只是缩短等待，它同时增加在途调用、连接和下游峰值压力。

> **Tip：并发实验不要先追求真实毫秒数**
>
> 首先证明任务确实重叠、结果保持一致、失败互不抹除。墙钟会受机器负载影响，精确 P99 应留给后面的容量实验。

2.5.2 在原仓定位：合约、装配和 fan-out

阅读：

- `candidate-pipeline/source.rs`：source 合约；
- `candidate-pipeline/candidate_pipeline.rs`：同阶段调度与结果收集；
- `home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs`：真实 source 装配；
- `home-mixer/sources/thunder_source.rs`；
- `home-mixer/sources/phoenix_source.rs`；
- `home-mixer/sources/cached_posts_source.rs`。

记录每路 enable 条件、输入 query 字段、下游依赖、候选来源和错误返回。特别区分"没有候选"和"调用失败"。

2.5.3 原理与边界：并行降低延迟，不减少工作

fan-out 的总等待接近关键路径最大值，但下游调用量约为入口 QPS 乘以启用 source 数。若每路还重试，放大继续乘上平均尝试次数。部分失败策略必须由产品价值决定：失去 Phoenix 也许还能返回关注内容，失去安全判断则可能不能继续。

错误至少分为：可忽略空结果、可降级依赖失败、请求本身非法、必须阻断的正确性失败。公开 Candidate Pipeline 对 `Vec<Result<...>>` 汇总结果执行 `flatten`，source 层仍可能留下错误日志，但错误会从候选汇总值中消失；若只看最终列表，就会制造"系统很稳定"的假象。

2.5.4 构造：并行三个独立 Source

实现 Following、Retrieval、Cache 三个 source。先让它们在同一进程并行，随后把其中一路改为本地网络服务。固定延迟分别为 8、28、12 个逻辑单位，固定返回 A、B、C，并让 Cache 同时返回一个与 Following 重复的 A。

合并结果时保留：candidate ID、所有来源、每路状态、开始结束时间和错误类型。去重不能丢掉来源归因。

> **验收契约：一条路坏了仍能解释**
>
> 在固定输入、相同快照、确定性 source 且没有 deadline 竞态时，并行结果与串行结果集合一致；Retrieval 失败时 A/C 仍可返回；失败 source 有独立指标；请求结束后所有任务都处于完成、取消或显式隔离状态。

2.5.5 破坏并证明：失败、重复和慢调用

依次注入：

- Retrieval 立即报错；
- Cache 返回重复 A；
- Following 返回空列表；
- 一个 source 永不结束；
- 客户端在处理中途断开。

本章只要求你发现“永不结束”会拖住整个请求，不要先在每路随意加 timeout。客户端断开也不会自动证明服务端子任务已取消。测试需证明正常集合、来源合并、部分失败和任务清理；指标需区分 empty、error 与 `still_in_flight_at_observation`。

2.5.6 验收：系统学会不必一起成功

- 三路 source 真正并行启动；
- 空结果与错误可区分；
- 重复候选合并但来源不丢；
- 单路错误不覆盖其他结果；
- 已计算入口 QPS 到下游 QPS 的 fan-out 放大；
- 已暴露无限等待问题，为下一章建立动机。

2.6 时间、Deadline、取消与重试预算

2.6.1 现场：Phoenix 只是慢，并没有报错

最危险的依赖不一定失败，它可能一直占用连接、并发槽和调用方耐心。若 Mixer 只有 100 ms 总预算，第二层服务不能重新获得一份完整 100 ms；否则有限调用链会逐层显著超出入口 SLA，无界递归或重试还可能继续放大。

> **新手 Tip：四个词不要混用**
>
> timeout 是一次等待上限；deadline 是绝对或可传播的结束时刻；cancellation 是停止无价值工作的信号；retry budget 是允许额外尝试消耗的总资源。它们解决不同问题。

2.6.2 在原仓定位：已有局部保护，也有公开缺口

阅读 `home-mixer/server.rs` 中 query 构造和 client 等待，再回到 `candidate-pipeline/candidate_pipeline.rs` 检查 source、hydrator 和 side effect 的调度。搜索 `timeout`、`CancellationToken`、`tokio::spawn`、`join_all`，建立调用点清单：

`调用方 | 下游 | timeout | 继承 deadline | 可取消 | 可重试 | 幂等`

公开快照能证明部分 client 有局部 timeout，不能证明整条 pipeline 拥有统一 deadline 和完整取消传播。

2.6.3 原理与边界：时间也是不可靠输入

使用 monotonic clock 计算本进程内的经过时间，但其原始数值不能直接发送到另一台机器比较。跨服务有两类不完美方案：传播绝对 wall-clock deadline，需要有界时钟误差；传播相对预算可以避免比较不同 monotonic 原点，却无法精确知道消息已经在网络/代理中停留多久，只能扣除保守余量。无论采用哪种，下游预算传播用于尽早停止，上游自己的本地 deadline 才是调用方观察到的最终强制边界。Wall clock 还服务事件时间、审计、跨节点过期和新鲜度，但要容忍偏移与回拨。

重试只适合暂时性且幂等或可安全重复的失败，并且必须服从原请求 deadline。预算至少包括剩余时间、最大 attempt 和允许的额外流量。指数退避与 jitter 可以减少同步重试，却不会消除流量放大；hedging 是在首个尝试尚未失败时并行发出备用请求，代价和适用场景也不同。最安全的默认是由最上层拥有总预算，每个下游只消费其中一部分。

2.6.4 构造：让预算沿调用链向下传播

在入口用本地 monotonic clock 建立 deadline，并由入口定时器强制结束等待。跨 RPC 选择一种明确方案：发送带时钟误差假设的 wall deadline，或发送扣除保守余量的剩余预算；接收方据此建立本地 deadline。每个 source 结束后报告完成、错误、超时或取消。把 Retrieval 拆成本地服务，再增加一个 Feature Service，形成两跳调用。

要求：上游只剩 30 ms 时，下游不得重新等待 100 ms；请求完成后取消尚未产生价值的工作；已完成的 A/C 作为部分结果保留。每次重试产生新的 attempt ID，但共享原 request ID 和 deadline。

> **验收契约：时间有唯一所有者**
>
> 调用方观察到的端到端等待受入口本地 deadline 约束；下游传播方案及其时钟/传输假设有记录；调用方停止等待不冒充远端已经停止，慢 source 必须协作取消或被显式隔离并计量；重试不越过剩余预算；不存在请求结束后持续增长的僵尸工作。

2.6.5 破坏并证明：时钟偏移、僵尸任务和重试风暴

注入：第二跳慢于剩余预算、底层忽略取消、两个客户端同时重试、wall clock 向后跳。分别测试本地 monotonic 经过时间和跨进程绝对过期语义。不要使用“多睡一会儿希望任务结束”；使用事件、假时钟或可控 barrier 证明任务状态。

计算入口 800 QPS、60% 启用 Retrieval、平均 1.5 次尝试时的下游 QPS。再说明如果每层都重试两次，最坏调用数为何呈乘法增长。

2.6.6 验收：系统知道何时放手

- deadline 由入口创建并向下传播；
- timeout、deadline、取消和重试预算分离；
- monotonic 与 wall clock 用途正确；
- 部分结果策略有明确产品依据；
- 超时任务不会悄悄继续消耗容量；
- 测试能确定性复现慢调用和取消；
- 下一部可以在有边界的请求里增加推荐能力。

3 第二部：让系统真正做推荐

这一部依次建立数据补全、过滤、安全、召回、精排、选择和混排。算法目标与工程约束在同一条请求里前进：每增加一项推荐能力，也必须增加对应的失败语义、测试和观测证据。

3.1 间章 C：数据契约、身份、版本与新鲜度

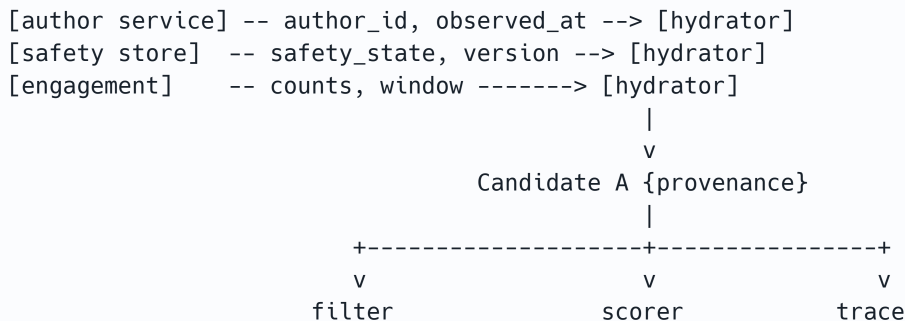
> **大型 Tip · 间章：大多数数据事故不是类型错，而是含义悄悄变了**
>
> 两端都成功解析 JSON，不代表它们理解的是同一件事。候选 ID、空值、时间单位、分数方向、特征版本和安全状态只要有一个语义漂移，系统就会“正常运行”并稳定地产生错误结果。数据契约必须描述含义、来源、身份、顺序、新鲜度与演化，而不只是字段类型。

3.1.1 1. 数据契约的七个维度

对每个关键字段，至少回答：

1. 语义：它代表哪个现实概念；
2. 身份：它属于哪个实体、请求或版本；
3. 类型与单位：整数、概率、毫秒还是秒；
4. 来源：谁产生，谁有权修正；
5. 时间：何时观测，允许多旧；
6. 缺失：为什么没有，下游怎样裁决；
7. 演化：新旧生产者和消费者如何共存。

图：字段从事实到决策的血缘



```
[author service] -- author_id, observed_at --> [hydrator]
[safety store]  -- safety_state, version --> [hydrator]
[engagement]    -- counts, window -------> [hydrator]
                                                |
                                                v
                                   Candidate A {provenance}
                                                |
                          +---------------------+-----------------+
                          v                     v                 v
                       filter                scorer             trace
```

每个消费者都应知道自己依赖的语义，而不只是字段名。

3.1.2 2. 身份对齐是批处理的首要不变量

设输入候选序列为 $C = [c_1, ..., c_n]$，批量特征结果为 R。安全契约不是简单的 $|C| = |R|$，而是：

$$\text{ids}(C) = \text{ids}(R)$$

，并且每个允许唯一的 ID 恰好出现一次。

如果协议承诺保持顺序，还要满足 $\text{id}(c_i) = \text{id}(r_i)$。更稳妥的做法是显式返回 `(candidate_id, result)`，由调用方按 ID join，并检查 unknown、duplicate 和 missing。

图：等长不等于对齐

```
input:    [A, B, C, D]
response: [A, C, B, D]   length = 4  ✓
zip:      A<-A B<-C C<-B D<-D        ✗

ID join 后：
A <- result(A)
B <- result(B)
C <- result(C)
D <- result(D)

还要拒绝 duplicate(B)、unknown(X) 与 silent missing(C)。
```

反例尤其隐蔽：B 与 C 都有合法数值，类型、长度和范围测试全部通过，但作者安全状态互换。只有身份不变量或端到端变形测试能抓住它。

> **Tip：不要让 map 悄悄吞掉重复 ID**
>
> 把列表直接转成字典可能让后一个重复值覆盖前一个，测试随后看见"键都在"。转换前先显式检查重复，再检查输入集合、输出集合和结果数量。

3.1.3 3. 空、缺失、未知与错误

`null` 经常承担太多含义：实体不存在、字段不适用、尚未计算、权限不足、依赖失败、版本不认识、值被删除。若下游只能看到一个空值，就无法做正确策略。

建议把结果建模为带原因的和类型：

```
FieldResult<T> =
  Present(value, observed_at, version, source)
  | NotFound
  | NotApplicable
  | Forbidden
  | DependencyError(error_class)
  | UnknownVersion(raw_tag)
```

这不要求所有语言真的使用代数数据类型，但协议必须保留这些区别。尤其安全字段：`Unknown` 不是 `Safe=false`，也不是 `Safe=true`；它需要由消费场景选择 fail-closed、安全缓存或人工隔离。

状态	事实	危险默认值	可能策略
NotFound	实体不存在	构造空实体	删除候选或核对 tombstone
NotApplicable	该字段无意义	当作 0	使用类型对应路径
Forbidden	无权读取	当作不存在	停止、脱敏或审计
DependencyError	本次获取失败	沿用零值	缓存、降级或拒绝
UnknownVersion	消费者不理解	忽略新语义	兼容路径或阻止发布

3.1.4 4. 数据时间不是一个 `timestamp`

至少区分：

- event time：现实事件发生的时间；
- ingestion time：系统收到事件的时间；
- processing time：某个处理阶段执行的时间；

- observed_at：这个字段值被观测或计算的时间；
- effective_at：规则或配置从何时生效；
- expires_at：超过何时不能再使用。

乱序发生时，processing time 晚不代表 event time 新。只按“最后收到”覆盖 Thunder 状态，会让延迟到达的旧事件复活已删除帖子。比较版本时要使用业务认可的顺序：单调序列、单分区 log offset 或带 epoch 的版本可以在各自作用域形成顺序；vector clock 只能识别 happens-before 与并发冲突，并发版本还需要合并或业务裁决。不要盲信 wall clock。

图：晚到达不等于更新

```
event time:   create(v7) ---- delete(v8)
delivery:          delete(v8) ---- create(v7, delayed)

按到达顺序覆盖：最终状态 = present   ✗
按版本比较：     v7 < v8，拒绝旧写    ✓

tombstone 必须保留到“旧事件不可能再到达”之后，
否则删除记录本身消失，旧 create 仍可能复活对象。
```

3.1.5 5. 新鲜度是消费方契约

同一份数据没有全局统一的“新鲜”。相关性特征允许旧 10 分钟，served history 也许只允许旧数秒，封禁状态可能要求强制读取最新权威状态。

对本进程自产并持续持有的值，可以记录本地 monotonic 插入时刻：

$$\text{age}_{\text{local}} = \text{mono_now} - \text{mono_inserted_at}$$

跨节点携带的 `observed_at` 通常是 wall clock，只能估算：

$$\text{estimated_age} = \text{wall_now} - \text{observed_at} +_{\vee_} \text{clock_error}$$

消费方声明最大允许陈旧度 S_x。超过阈值后进入刷新、stale-while-revalidate、降级、拒绝或忽略字段等策略。若有 source version、log offset 或 snapshot ID，通常比绝对时间更适合判断“是否达到消费方要求”。

源码证据：原仓中的字段生长过程

从 `home-mixer/models/query.rs`、`home-mixer/models/candidate.rs` 观察核心身份，再沿 `home-mixer/candidate_pipeline/query_features.rs`、`home-mixer/candidate_pipeline/candidate_features.rs` 与 hydrator 实现追踪字段来源。`candidate-pipeline/hydrator.rs` 给出抽象边界。阅读重点应是字段由谁产生、在哪里消费、失败怎样表示，而非记忆所有字段名。

3.1.6 6. 分数也需要契约

推荐系统中的 score 可能是 logit、概率、未校准实数、距离、相似度或多目标加权值。它们不能仅因类型都是 `float` 就直接相加。

分数契约至少声明：

- 方向：越大越好还是越小越好；
- 范围：是否有界，NaN/Infinity 怎样处理；

- 含义：预测哪个事件、时间窗和人群；
- 校准：能否解释为概率；
- 模型与特征版本；
- 可比较范围：跨 source、跨模型是否可直接比较；
- 缺失和默认值策略。

反例：Phoenix 返回余弦相似度，另一 source 返回点击概率，把两者直接按 0.5/0.5 加权没有稳定含义。另一个反例是模型升级后整体 logit 平移，固定阈值让候选量骤降，而离线排序指标仍看似正常。

3.1.7 7. Schema 兼容不等于语义兼容

增加 optional 字段通常能通过解析兼容，却仍可能改变行为。常见演化风险：

- 新 enum 值被旧消费者映射为默认安全状态；
- 单位从秒改成毫秒但类型不变；
- 空列表从“没有数据”改成“明确无兴趣”；
- 默认分值从 0 改成 `null`；
- ID 从帖子空间扩展到多内容类型，旧去重逻辑发生碰撞；
- 删除字段时仍有延迟事件和缓存对象携带旧格式。

图：滚动发布中的双向兼容

```
时间 ──────────────────────────────────>

producer:  v1 v1 v1 | v1 + v2 | v2 v2
consumer:  v1 v1    | v1 + v2 |    v2
cache:       [---- old objects ----]
queue:           [------ delayed v1 events ------]

必须验证：
v1 -> v1, v1 -> v2, v2 -> v1, v2 -> v2,
以及升级一半时回滚后仍能读写。
```

安全演化常采用 expand–and–contract：先让消费者理解新旧格式，再让生产者写新格式；完成回填和观测后，最后删除旧读写路径。回滚窗口结束前，不能提前移除旧协议。

3.1.8 8. 数据质量必须可观测，但避免高基数陷阱

建议观测：missing by reason、unknown enum、alignment violation、duplicate ID、field age bucket、schema version、model version、NaN score、候选阶段数量变化。

candidate ID、user ID 不应直接成为指标标签，否则时间序列数量爆炸。具体身份放入采样日志或 trace；指标保留有限枚举和分桶。对隐私字段还要规定脱敏、保留期与访问权限。

单个均值会掩盖局部分片：总体缺失率 1%，可能全部集中在新用户、某语言或某 source。数据质量应按有业务意义且基数受控的 slice 检查。

3.1.9 9. A–F 数据契约审计

候选	关键契约	故障样例	正确处理
A	稳定身份与来源	合并 source 时来源丢失	保留 provenance 集合
B	author 特征对齐	与 C 的结果交换	按 ID join 并拒绝错位

C	缓存年龄与展示历史	旧缓存覆盖删除或已展示事实	记录 age 并重新执行当前过滤
D	屏蔽作者关系	权限依赖失败后误放	保留 unknown/error 并按契约裁决
E	安全字段原因	依赖错误变成默认 safe	保留 Unknown 并按红线裁决
F	内容类型与间距	ID 空间碰撞或类型丢失	复合身份并保护 safe gap

要求读者为每个候选写出字段血缘，而不是只写最终值。一个可解释的候选应能回答：从哪里进入、何时被观测、用哪个模型打分、读取了哪个安全版本、为何保留或删除。

3.1.10 10. 契约测试应验证关系，而不只是样例

除了固定输入输出，还应验证：

- permutation：改变批量返回顺序，ID join 后结果不变；
- duplication：插入重复 ID，系统明确拒绝而非静默覆盖；
- omission：删除一个结果，其他候选不应错位；
- stale transformation：增加 age 后只触发声明的降级；
- version mixing：v1/v2 生产者消费者组合保持兼容；
- replay：重复或乱序事件仍收敛到相同状态；
- score sanity：NaN、Infinity、越界值不会污染排序。

Tip：先测试不变量，再测试实现细节

若测试只断言某个数组恰好等于 `[A, C, B]`，一次合法的并行化也会打破它。优先断言身份守恒、过滤红线、稳定 tie-break、版本单调和缺失语义；只有协议承诺顺序时才锁定顺序。

3.1.11 11. 一份字段契约卡

为关键字段建立可审阅的契约卡：名称、业务定义、实体 ID、生产者、消费者、类型/单位、合法范围、版本、新鲜度、缺失原因、默认策略、权限级别、指标和迁移计划。契约卡可以是文档，但关键约束必须进入代码检查、测试和运行时验证。

不要追求给所有临时字段写百科全书。优先覆盖跨团队、跨进程、影响安全或排序、需要长期持久化、会参与版本演化的字段。

3.1.12 自检：你能发现“正常运行的错误”吗

1. 为什么批量响应长度相等仍不能证明身份对齐？
2. `NotFound`、`Forbidden` 与 `DependencyError` 合并成 null 会造成什么错误？
3. event time 和 processing time 在乱序事件中分别回答什么问题？
4. 谁应该决定字段允许多旧：生产者还是消费者？为什么？
5. 两个 `float` 分数在什么条件下才可比较或相加？
6. optional 新字段为何仍可能造成语义不兼容？
7. 滚动升级至少需要验证哪四种 producer/consumer 组合？
8. 哪些字段适合放指标标签，哪些只能进入采样日志或 trace？
9. 为 E 的安全状态设计一个不会把依赖错误误判为安全的类型。

检查点：数据正确必须包含身份与语义

首次阅读先能解释 A–F 的字段血缘与契约卡；故意交换 B/C、重复 A、遗漏 D、延迟旧事件和未知 enum 的实现与证据，统一并入下一工程章，不在间章另建里程碑。

3.2 Query、Candidate 与数据契约

3.2.1 现场：一个 ID 还不是可推荐的内容

Source 常常只返回 candidate ID、来源和粗略分数。过滤需要作者、时间和安全字段，排序需要行为序列和特征，混排还要内容类型。补字段不是无害的“查表”，而是一组远程调用、批量对齐和缺失语义。

> **新手 Tip：先画字段血缘**
>
> 遇到几十个字段时，不要逐个背诵。为每个字段写四列：谁产生、谁消费、缺失表示什么、允许多旧。字段血缘比数据结构定义更能解释系统。

3.2.2 在原仓定位：谁让 Query 和 Candidate 逐渐完整

阅读：

- `home-mixer/models/query.rs` 与 `home-mixer/models/candidate.rs`；
- `home-mixer/candidate_pipeline/query_features.rs`；
- `home-mixer/candidate_pipeline/candidate_features.rs`；
- `home-mixer/candidate_hydrators/core_data_candidate_hydrator.rs`；
- `home-mixer/query_hydrators/user_action_seq_query_hydrator.rs`；
- `candidate-pipeline/hydrator.rs`。

挑选 viewer history、author ID、created-at、安全状态和媒体类型五个字段，画出完整生产与消费链。检查批量结果怎样与输入 candidate 对齐。

3.2.3 原理与边界：缺失、空值和错误不是同一件事

`None` 可能表示没有值、尚未请求、请求失败、权限不足或版本不认识。若这些状态共用一个空值，下游只能靠猜。数据契约至少要说明：字段来源、类型、单位、新鲜度、缺失原因、错误策略和身份对齐方式。

批量接口尤其危险：返回长度相等并不能证明第 i 个特征仍属于第 i 个 candidate。公开 Hydrator 只校验长度并按位置 zip/update，没有额外核对 candidate ID；这是原仓可见的契约边界。读者实现应尽量显式携带 ID 并校验集合、重复和顺序；若只能按位置 zip，必须把长度和稳定顺序写成被测试保护的强契约。

3.2.4 构造：实现 Query 与 Candidate Hydration

给读者项目增加两个阶段：QueryHydration 补小林的行为序列和屏蔽列表；CandidateHydration 为 A—E 补作者、创建时间、安全状态和基础互动数。用批量接口一次处理多个 candidate。

定义显式结果：`present`、`not_found`、`dependency_error`、`forbidden`。对外协议为了防止枚举攻击，有时会合并 not-found 与 forbidden；内部审计仍应保留区别。E 的安全字段使用 `dependency_error`，不要直接变成 `false` 或空字符串。对关键或跨边界字段记录来源和观测时间，避免为所有字段制造不可承受的元数据成本。

> **验收契约：候选不能戴错身份证**
>
> hydration 返回必须与输入 candidate 在数量和 ID 上对齐；缺失原因可区分；单个 candidate 缺失不得导致其他结果错位；下游不能消费尚未满足契约的字段。

3.2.5 破坏并证明：构造“长度正确、身份错误”

让批量下游把 B/C 的结果交换，但保持列表长度不变；再让 D 缺失、E 返回依赖错误。先写一个只检查长度的弱测试，观察它为什么通过，再增加 ID、重复、缺失原因和字段来源断言。

指标至少包含 batch size、missing by reason、dependency error、alignment violation 和字段 age。注意高基数 candidate ID 不应成为指标标签，应留在 trace 或采样日志中。

3.2.6 验收：数据已足够让后续阶段判断

- Query 与 Candidate 字段责任分开；
- 五个关键字段拥有血缘和新鲜度；
- 缺失、not-found、权限和依赖错误可区分；
- 批量接口保护数量与身份；
- 测试能抓住等长错位；
- E 的未知安全状态留给下一章按消费语境裁决。

3.3 过滤、安全与业务不变量

3.3.1 现场：模型喜欢，不代表系统可以展示

D 来自小林屏蔽的作者，E 的安全状态未知。即使它们拥有最高预测分数，也不能让排序模型替代权限、安全和产品规则。过滤是硬边界，顺序与失败默认值都会改变用户看到的集合。

> **Tip：先区分资格与偏好**
>
> “不应出现”属于资格约束，“不太想出现”属于偏好或打分。把资格规则塞进负权重，会让参数变化意外突破安全边界。

3.3.2 在原仓定位：predicate 才是最终事实

选择代表文件深读：

- `home-mixer/filters/author_socialgraph_filter.rs`；
- `home-mixer/filters/core_data_hydration_filter.rs`；
- `home-mixer/filters/vf_filter.rs`；
- `home-mixer/filters/drop_duplicates_filter.rs`；
- `home-mixer/filters/previously_served_posts_filter.rs`；
- `home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs` 中真实装配顺序。

不要从文件名猜行为。逐个写出 predicate、依赖字段、删除理由、字段缺失时的默认值和执行位置。

3.3.3 原理与边界：顺序、fail-open 与 fail-closed

过滤器不是交换律操作。若系统明确建立合法的 pre-hydration cheap-filter 阶段，先去重或做廉价权限判断可以省去昂贵 hydration；在当前原仓 pipeline 中，candidate hydration 已先于这组 filter 完成，单纯交换 filter 顺序不能追回已经花掉的成本。某些规则依赖 hydration 后字段，也不能任意前移。

fail-open 表示判断失败时保留，保护可用性；fail-closed 表示判断失败时删除或拒绝，保护安全与权限。选择必须结合 surface、误杀、漏放、信息泄露和补偿能力，而不是由工程师对可用性的抽象偏好决定。公开 `VFFilter::should_drop(None)` 和部分 `unwrap_or(false)` 路径表现为保留；读者项目对 E 采用 fail-closed 是有意更严格的教学选择，不是对原仓行为的复述。

3.3.4 构造：建立有序 Filter Chain

实现去重、屏蔽作者、基础数据完整性、安全状态和 previously-served 五个 filter。每次删除记录 candidate ID、source、filter、reason 和决策输入，但避免把敏感文本写入日志。

为 E 明确选择策略：主 feed 核心路线建议 unknown safety fail-closed，同时允许你设计一个隔离的低风险 surface 使用不同策略。策略必须由配置版本控制，并出现在请求快照中。

> **验收契约：硬边界不能被分数绕过**
>
> 在屏蔽列表成功取得或其依赖按契约 fail-closed 时，D 不能进入 ranking；E 的处理符合当前 surface 策略；每次 drop 记录 first-observed primary reason，但不声称候选只违反一条规则；filter 失败不能被伪装成普通零候选；改变顺序必须经过不变量测试。

3.3.5 破坏并证明：交换顺序和吞掉错误

实验至少包括：

- 安全检查返回依赖错误；
- previously-served 数据陈旧；
- 同一候选来自两个 source；
- 把 expensive filter 移到最前；
- 捕获异常并返回 `keep=true`；
- 将安全规则降级成 score penalty。

比较结果集合、下游调用量、drop reason、延迟和安全风险。测试既要覆盖每个 predicate，也要覆盖组合顺序和配置切换。

3.3.6 验收：只有合格候选进入排序

- 资格约束与偏好分数分离；
- filter 顺序有数据依赖和成本依据；
- fail-open/fail-closed 选择可解释；
- unknown 不再被隐式解释；
- D/E 的命运由测试保护；
- 每个删除原因可观测但不泄露敏感数据。

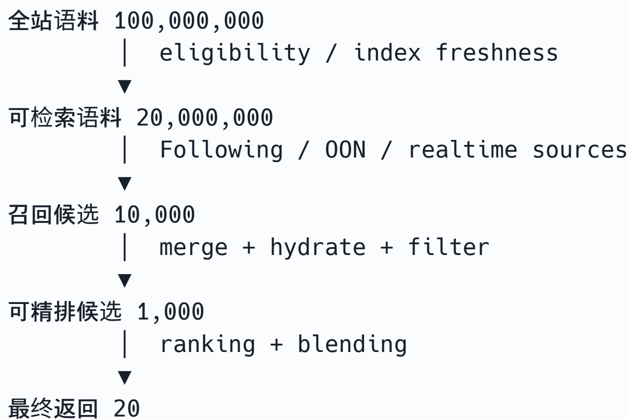
3.4 间章 D：召回不是“小号排序”

> **大型 Tip · 间章：从全站语料到可精排候选集**
>
> 本章放在 Phoenix Retrieval 之前。它补齐召回层的数学直觉、工程契约与评估边界：读者不需要先会深度学习，但必须知道一个候选为什么能够进入精排，以及“没有被召回”为什么是不可逆的系统决定。

3.4.1 1. 召回解决的是搜索空间问题

假设站内有一亿条可推荐内容，而一次请求只能让昂贵的排序模型处理一千条。召回（retrieval）负责用较低成本，从全量语料中找到一个“可能相关”的小集合；排序（ranking）才在这个集合内细分先后。

图：候选漏斗与不可逆的信息损失



某条内容若在“召回候选”之前消失，后面的模型再聪明也救不回来。

因此，召回的首要目标不是精确预测最终顺序，而是在成本约束下提高有价值内容进入候选集的概率。它通常愿意接受一定噪声，换取覆盖率与多样性。

- **语料（corpus）**：某个明确版本下可被检索的全部 candidate。
- **查询（query）**：描述本次请求意图的表示，通常来自用户、上下文与实时状态。
- **召回源（source）**：产生候选的独立策略，例如关注关系、向量近邻或实时热门。
- **召回预算**：一次请求允许某个 source 消耗的时间、CPU、网络、内存与候选数量。

3.4.2 2. Two-Tower 的最小数学模型

Two-Tower 并不神秘。用户塔把查询映射为向量 $u \in \mathbb{R}^d$，内容塔把 candidate 映射为向量 $v_i \in \mathbb{R}^d$。最简单的相似度是点积：

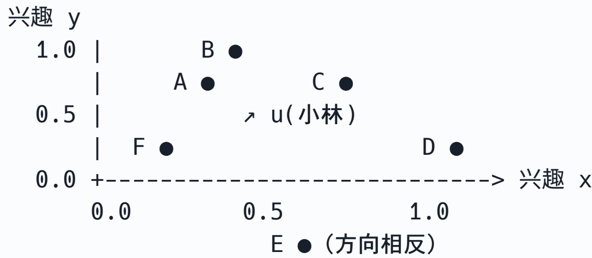
$$s(u, v_i) = u \cdot v_i$$

如果两边先做 L2 归一化，$\hat{u} = \frac{u}{\|u\|_2}$、$\hat{v}_i = \frac{v_i}{\|v_i\|_2}$，那么点积就是余弦相似度：

$$s(\hat{u}, \hat{v}_i) = \cos\theta_i$$

归一化消除了向量长度带来的尺度差异，但也丢掉了“置信强度”可能携带的信息。是否归一化是模型契约，而不是无关紧要的实现细节。

图：二维空间里 A—F 的直觉



离 u 方向近，不等于欧氏距离最近；归一化后比较的是夹角。

手算例子：若 $u = (3, 4)$，则 $\|u\|_2 = 5$，归一化后为 $(0.6, 0.8)$。若 A 为 $(0.6, 0.8)$，相似度为 1；若 B 为 $(0.8, 0.6)$，相似度为 0.96。这个小差距足以改变 top–K 边界。

小心：零向量不是普通输入

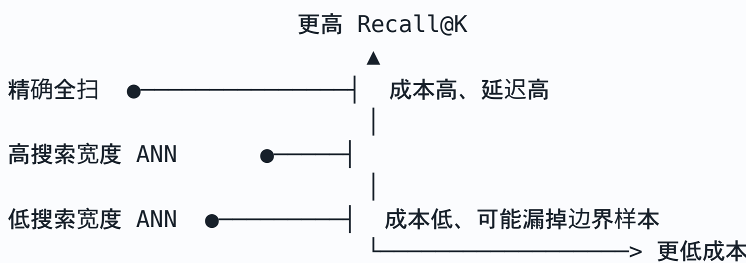
零向量没有方向，不能做 L2 归一化。实现必须选择明确行为：拒绝、回退到非个性化 source，或返回带原因的空结果。静默产生 NaN 会把错误带到排序层。

3.4.3 3. top–K 是带契约的选择，不只是排序切片

精确 top–K 可以计算每个 candidate 的分数，再按 `(score desc, candidate_id asc)` 排序取前 K。第二关键字提供 total order；在候选集合和分数完全相同的前提下，它保证稳定顺序。ANN 可能先返回不同候选集合，浮点内核也可能产生微差，因此 tie–break 不能单独保证跨节点结果完全一致。

当语料巨大时，系统常用近似最近邻（ANN）索引，以召回质量换延迟与成本。此时至少存在三个独立参数：ANN 搜索预算/宽度（例如 `efSearch` 或 `nprobe`）、source top–K、合并后 retain top–K。把它们都叫 `k` 会制造配置事故。

图：精确检索与近似检索的交换



调参问题不是“哪个最快”，而是“在质量下限内哪个最便宜”。

索引还引入版本与新鲜度。模型版本、candidate embedding 版本、索引版本必须彼此兼容。新内容尚未进入索引时，向量 source 可能完全看不到它，所以推荐系统通常需要实时 source 与离线索引互补。

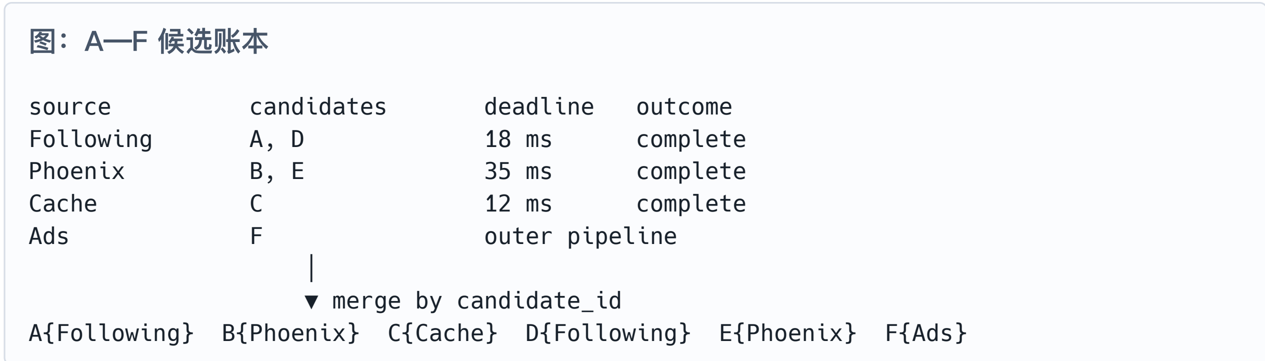
3.4.4 4. 多路召回必须保留来源与预算

小林的请求可以并行调用：

- Following/Thunder 路径：召回 A 与待过滤的 D；
- Phoenix OON source：召回 B 与安全状态未知的 E；

- Cache source：交回旧候选 C；
- Ads source：在外层提供 F，不与 organic retrieval score 直接比较。

如果同一 candidate 同时由多路召回，不能只保留一个裸 ID。候选账本至少应保存 source 集合、各自原始分数、模型/索引版本与产生时间。否则后续无法解释、切片评估或判断哪个 source 退化。



图：A—F 候选账本

source 的分数通常不可直接互比：Following 的“关系强度 0.8”和 Phoenix 的“余弦 0.8”不是同一量纲。合并阶段应把它们当作 provenance feature，交给后续排序或显式校准，而不是直接全局排序。

3.4.5 5. 召回正确性包含失败语义

远程检索可能完整成功、部分成功、超时、取消、索引落后或返回协议错误。空列表不能同时表示这些状态。推荐的结果类型至少区分：

```
Complete(candidates, metadata)
Partial(candidates, reason, metadata)
Unavailable(reason, retryability)
Cancelled(deadline_or_caller)
```

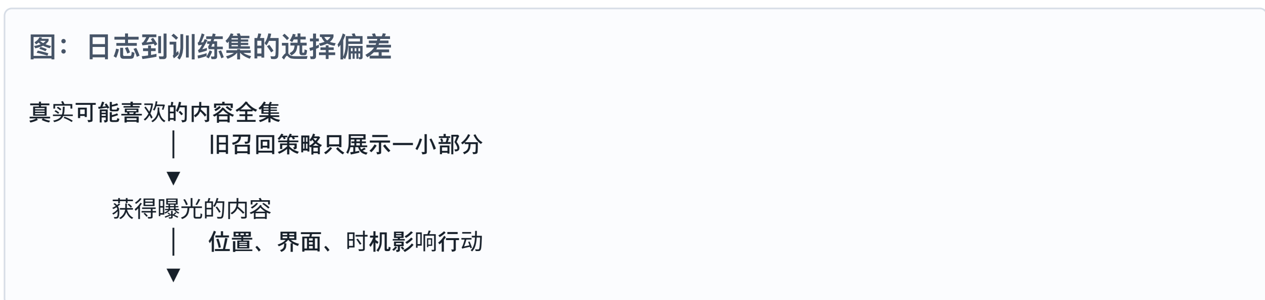
只有这样，Mixer 才能决定是继续、降级、补充 fallback，还是快速失败。对用户返回 20 条内容并不证明所有 source 健康；协议可用性与推荐质量必须分别观测。

> **小心：反例：把 timeout 当作空召回**
>
> Phoenix 超时后返回 `[]`，接口看似简单，却让监控误以为用户没有匹配内容。排序层也无法区分“没有好内容”和“检索系统坏了”，最终掩盖持续退化。

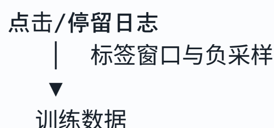
3.4.6 6. 离线标签与负样本决定你在优化什么

一次曝光后点击可以作为正标签，但“未点击”不一定是负样本：用户可能根本没看到、位置太靠后、网络中断，或者有兴趣但没有行动。训练数据首先是既有系统策略产生的日志，不是对用户偏好的无偏观察。

常见负样本来源包括同批曝光未互动内容、随机语料、同主题 hard negatives，以及模型近邻但未被选择的内容。它们改变任务难度，也改变模型学到的边界。训练和评估必须记录采样策略。



图：日志到训练集的选择偏差



每一层都在选择；模型只能学习被记录下来的世界。

未来互动、未来作者状态或请求之后更新的特征若进入 query，会产生时间泄漏。随机拆分也可能让同一用户的未来行为泄漏到训练集。推荐任务通常需要按事件时间切分，并按数据可用时点重建特征。

3.4.7 7. `Recall@K` 的含义与局限

设某请求的相关集合为 R，召回结果前 K 个组成 C_K：

$$\mathrm{Recall@K} = \frac{|R \cap C_K|}{|R|}$$

如果 $R = \{B, D\}$，而 $C_3 = \{A, B, C\}$，则 `Recall@3` 为 $\frac{1}{2}$。当 $|R| = 0$ 时必须预先约定跳过、记零或单独报告，并说明宏平均/微平均方式。生产日志往往不知道未曝光内容是否相关，所以离线的 R 只是代理标签集合。

只看总体平均值会掩盖冷启动用户、低资源语言、小地区、新作者和新内容的失败。至少要报告切片分布、置信区间、索引新鲜度与空结果率。

> **小心：反例：Recall 提升但产品变差**
>
> 新模型反复召回同一作者的相似内容，离线 `Recall@100` 上升，却压缩了主题和作者覆盖。精排收到的信息更单一，长期满意度与供给生态可能下降。

召回评估还应配合：candidate coverage、source contribution、freshness、duplicate rate、index lag、retrieval latency，以及最终被选中的候选中各 source 的贡献。

3.4.8 8. 在 x-algorithm 中怎样寻找证据

阅读 `phoenix/recsys_retrieval_model.py` 时，沿张量 shape 找 user/candidate 表示、归一化、相似度和 logits。阅读 `phoenix/run_retrieval.py` 与测试，确认它们使用的是合成输入还是生产 artifact。

再读 `home-mixer/sources/phoenix_source.rs`，关注模型结果怎样变成候选及错误怎样传播；读 Candidate Pipeline 的 source trait，理解 source 契约为何不应该泄漏某个模型实现。

原仓能证明公开快照中的结构和调用位置，不能证明生产索引服务、训练数据、ANN 参数或线上效果。把“源码看到的”和“从经验推测的”分开写，是源码审计的基本纪律。

3.4.9 9. 带入 Retrieval 章的判断标准

读者自己实现一个透明的二维或八维检索器，并固定 A—F 的向量与候选 ID。至少证明：

- L2 归一化可手算，零向量行为明确；
- top-K 使用稳定 tie-break；
- K 大于语料规模、重复 ID、相同分数均有测试；
- 结果携带 source、model version、corpus version；
- 合并时保留多来源，不直接比较异构 source 分数；
- 超时、部分结果与真实空结果可区分；
- 时间切分后的 `Recall@K` 可复算，并有至少两个用户切片。

3.4.10 10. 自检问题

1. 为什么召回漏掉 B 后，ranking 无法补救？
2. 点积和余弦在什么条件下等价？零向量为什么破坏这个条件？
3. 为什么 tie-break 属于正确性契约，而不只是美观？
4. ANN 的“近似”改变了哪些指标和运维信号？
5. Following 0.8 与 Phoenix 0.8 为什么不能直接比较？
6. 空结果、超时和部分成功若共用一个表示，会导致什么决策错误？
7. 随机拆分推荐日志为什么可能产生未来泄漏？
8. `Recall@K` 上升为什么仍可能损害多样性和长期体验？
9. 新内容尚未进入离线索引时，你会用什么 source 补洞？
10. 哪些结论能由 x-algorithm 公开源码证明，哪些只能作为设计假设？

检查点：能否把召回写成一份工程契约

如果你能同时说明输入、输出、版本、稳定性、失败语义、预算、离线指标和不可证明边界，就已经越过了“把向量点积叫召回”的阶段。

3.5 Phoenix Retrieval：先决定谁值得精读

3.5.1 现场：精排模型不可能阅读全站

Retrieval 的任务不是决定最终顺序，而是从巨大语料中找出一个足够好、足够多样且成本可控的候选集。未被召回的内容，后面的 ranking 无法凭空救回。

> **新手 Tip：向量先看 shape**
>
> 阅读 JAX 模型时，先在每一步旁边写张量形状，再看数学细节。two-tower 的核心是把 user 与 candidate 映射到同一维空间，随后比较相似度。

3.5.2 在原仓定位：两座塔怎样相遇

阅读：

- `phoenix/recsys_retrieval_model.py`；
- `phoenix/test_recsys_retrieval_model.py`；
- `phoenix/run_retrieval.py`；
- `home-mixer/sources/phoenix_source.rs`。

标出 user tower、candidate tower、归一化、corpus embedding、top-K 和返回 ID 的位置。官方测试与 `run_retrieval.py` 使用合成输入和随机初始化，不需要 2.9 GB artifact，但需要安装仓库 `pyproject.toml` 声明的依赖。

3.5.3 原理与边界：召回率决定排序上限

归一化后 dot product 等价于 cosine similarity。读者实现应以 `(score desc, candidate_id asc)` 建立稳定 total order；公开 `top_k` 调用本身并未证明跨设备 tie-break 稳定。召回质量不能只看平均 `Recall@K`，还要按新用户、语言、地区、内容年龄和供给规模切片，并区分 ANN 对 exact search 的 recall 与推荐相关性 recall。

负采样、候选池时点和未来信息泄漏会让离线指标失真。一个 retrieval 模型可以有很高 Recall，却只召回相似作者，造成生态和多样性问题。

3.5.4 构造：实现一个透明的 Two-Tower

不要重写 Transformer。用标准库实现一个二维或八维的 user encoder、candidate encoder、L2 normalize、dot product 和稳定 top-K。固定小林向量及 A—E candidate 向量，保证手算能够核对程序。

接口必须接受 `k`、corpus version 和 request context，返回 candidate ID、similarity、retrieval source 与模型版本。让 B 由 out-of-network（OON，关注网络之外）retrieval 进入候选池，并与 Following 的 A 合并。

> **验收契约：召回只负责入围**
>
> 输出向量归一化；读者实现的 top-K 具有稳定 total order；重复 candidate 合并来源；corpus/model 版本可观测；`k` 大于语料规模、零向量和相同分数都有明确的教学契约，不冒充 Phoenix 当前 API 行为。

3.5.5 破坏并证明：让离线分数说谎

测试零向量、重复 ID、相同相似度、不同 corpus version 和 k 边界。再构造一个离线数据集，使未来互动泄漏进 user vector，观察 Recall 虚高；随后用时间切分修正。

计算 `Recall@K`，并按新用户与老用户切片。选做：安装 Phoenix 依赖，运行官方 retrieval tests 与 demo，记录它证明的仅是模型结构和合成输入行为，不是线上效果。

3.5.6 验收：谁获得精排资格可以解释

- 读者能从 shape 读懂 two-tower；
- similarity、top-K 和 tie-break 可手算；
- retrieval 与 ranking 职责分离；
- 版本和来源进入候选账本；
- 离线评估避免明显泄漏；
- 召回不足和服务失败拥有不同指标。

3.6 间章 E：排序分数不是用户价值

大型 Tip · 间章：从多目标预测到可信评估

本章放在 Ranking 与 Blending 附近。它解释模型输出、业务效用、稳定排序、离线指标与在线实验之间的边界。目标不是背指标，而是能判断“一个更高的分数”究竟证明了什么。

3.6.1 1. 排序是在候选集内分配稀缺注意力

召回决定哪些内容有资格被考虑，排序决定谁获得更靠前的位置。用户注意力随位置快速下降，因此交换 A 与 B 的顺序，不只是界面变化，也会改变未来收集到的训练数据。

图：排序系统同时是预测器和数据生成器

模型影响曝光，曝光影响标签，标签再训练下一版模型。
这是一条反馈回路，不是静态分类题。

一次排序请求至少要明确：候选集合、可用特征时点、模型版本、参数版本、score 定义、稳定 tie-break、过滤后的合法性，以及超时或缺特征时的回退顺序。

3.6.2 2. 预测概率、效用与最终分数是三件事

模型可能输出点击、点赞、回复、转发、长停留和负反馈的 logits、连续预测或经激活/校准后的概率。若这里明确采用概率解释，设 candidate i 的动作概率为 $p_{i,a}$，业务为动作指定权重 w_a，一种简单代理效用为：

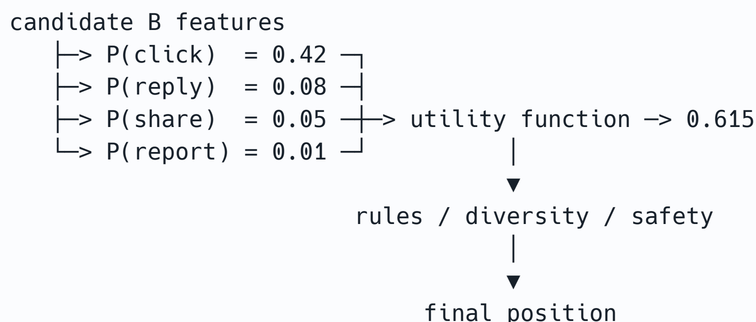
$$U_i = \sum_a w_a p_{i,a}$$

如果负反馈是损失，可以令对应权重为负。例如：

$$U_i = 1.0 p_{\text{click}} + 2.0 p_{\text{reply}} + 1.5 p_{\text{share}} - 4.0 p_{\text{report}}$$

这不是用户幸福的物理定律，而是一项产品假设；它还隐含各项效用可加、权重单位可比较等前提，多种行为也可能同时发生并相互依赖。权重改变会改变供给激励，也可能改变不同用户群体的体验。

图：从预测头到最终次序



预测分数还可能经过温度缩放、截断、作者惩罚、新鲜度加成或实验参数调整。因此日志应区分 raw prediction、calibrated probability、utility score 与 post-blending position。

> **小心：反例：把任意 logit 当概率**
>
> 未经 sigmoid/softmax 或校准的 logit 可能为负，也不满足概率解释。即使数值在 0 到 1 之间，也不能自动解释为“十次会发生四次”。

3.6.3 3. 校准回答“0.4 是否真的约等于四成”

若模型把一组样本的点击概率都预测为 0.4，长期观察中约 40% 真正点击，才称这一区间校准良好。区分能力与校准不同：模型可以正确排出高低，却系统性高估所有概率。

最小校准表可以把预测分桶：

图：可靠性图的文本版本

```
预测区间      平均预测      实际发生率      样本数
[0,.2)         .12           .10          4,200
[.2,.4)        .31           .22          2,800   <- 高估
[.4,.6)        .48           .47          1,900
[.6,.8)        .69           .61            700

理想情况：平均预测 ≈ 实际发生率；同时必须报告样本数。
```

校准会随用户群、地区、设备、内容类型和时间漂移。总体校准良好可能只是不同方向的误差互相抵消，所以需要切片。

3.6.4 4. 稳定排序属于可重复性与调试能力

当 A 与 B 的最终分数相同，排序器必须定义稳定次序，例如 `(score desc, original_position asc, candidate_id asc)`。不要依赖哈希表迭代顺序、线程完成顺序或某种语言当前版本的未声明行为。

稳定性带来三项收益：重试结果一致、A/B 实验噪声更小、事故复现更可靠。它也防止某个 source 因响应更快而获得隐含位置优势。

图：不稳定 tie-break 怎样制造幽灵变化

```
请求 1：线程完成 B -> A -> C     同分结果：B, A, C
请求 2：线程完成 A -> C -> B     同分结果：A, C, B
                                |
相同输入 + 相同模型 + 相同参数，却得到不同曝光日志
```

3.6.5 5. 排序指标先从一个请求算起

`Precision@K` 关注前 K 个槽位中有多少相关：

$$\text{Precision@K} = \frac{|R \cap C_K|}{K}$$

这个定义假设缺位按不相关处理，适合固定槽位语义；若用 `min(K, returned)` 作分母，会奖励少返回，必须同时报告 fill rate/coverage。本书采用固定 K 并单列结果不足率。

`Recall@K` 关注相关集合中有多少被找回。排序更常使用带位置折扣的 DCG：

$$\mathrm{DCG@K} = \sum_{i=1}^{K} \frac{2^{\mathrm{rel}_i} - 1}{\log_2}(i+1)$$

把实际 DCG 除以同一标签集合的理想排序 IDCG，得到：

$$\mathrm{NDCG@K} = \frac{\mathrm{DCG@K}}{\mathrm{IDCG@K}}$$

若所有相关度都为零，IDCG 为零，必须预先定义该请求是跳过、记零还是单独报告。指标实现也需要契约。

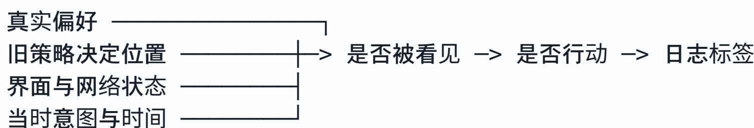
例如资格过滤后只评估 organic 的 A/B/C，分级相关度为 `A=3`，`B=2`，`C=0`。D/E 已因权限/安全边界排除，F 由外层广告混排处理。把 A 放第一位比放到更后位置贡献更高；NDCG 正是在编码“越靠前越重要”。

> **小心：反例：平均 NDCG 掩盖整类用户**
>
> 老用户占 95%，新用户占 5%。总体 NDCG 上升 1%，但新用户下降 20%，平均数仍可能为正。必须同时给出切片、样本量与不确定性。

3.6.6 6. 离线评估测到的是日志条件下的反事实替代品

排序日志只包含旧策略展示过的内容。未曝光的 E 没有点击，不代表用户不喜欢 E。位置越靠前越容易获得互动，因此标签同时混入 position bias、presentation bias 与旧策略选择偏差。

图：观察到的标签不等于真实偏好



日志记录的是整条因果链末端，而不是纯粹偏好。

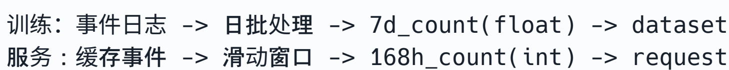
可以通过随机探索、位置偏差估计、逆倾向加权或更严谨的因果方法缓解，但任何修正都依赖假设。本书主线要求先做到：记录曝光、位置、候选资格、模型与参数版本，并用时间切分防止未来泄漏。

3.6.7 7. train/serve skew 比“模型不准”更常见

训练和服务可能在特征计算、默认值、时间窗口、枚举解释、归一化和版本上不一致。模型本身离线表现很好，线上却读取了另一种含义的数据。

为每个高价值 feature 记录：业务定义、类型、单位、事件时间、可用时间、缺失语义、训练生成路径、服务生成路径、版本和数据责任人。对 A—F 的固定样本做离线/在线特征一致性对比。

图：同名 feature 的两条路径



^
名字相同，不代表边界、时点、缺失值和类型相同

3.6.8 8. Blending 是带约束的再排序

最终顺序往往还要满足去重、作者上限、内容多样性、广告间距、安全和产品资格。此时最高 utility 的 candidate 不一定占据最高可用位置。

可以把它理解为约束优化：在可行集合 F 中选择顺序 π，最大化总效用：

$$\max_{\pi \in F} \sum_i d_i U_{\pi_i}$$

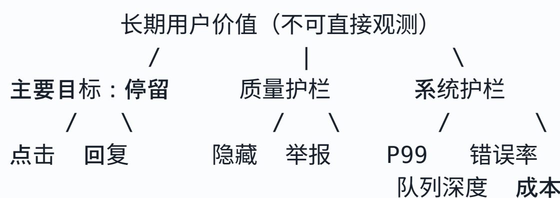
其中 d_i 是位置折扣，F 编码硬约束。生产实现常采用可解释的贪心和规则，而不是求全局最优；重要的是区分硬约束、软目标和回退行为。

A—F 的固定身份不变：D 来自被屏蔽作者，E 的安全字段未知，二者应在 ranking 前按资格策略处理；F 才是广告。若本节额外设定 A 与缓存候选 C 同作者，作者上限可能让二者只保留一个，但这只是新增属性，不改变 C 的缓存来源。

3.6.9 9. 在线实验必须预先写出决策规则

离线指标适合快速筛选，不能替代在线因果验证。A/B 实验至少需要：随机化单位、分桶稳定性、主要指标、护栏指标、最短观察窗口、样本量、停止规则和回滚阈值。

图：指标树而不是一个点击率



显著性不是重要性。极大样本能让微小差异显著，却未必值得复杂度和风险。反过来，短期点击上涨若伴随举报、延迟或作者集中度恶化，也不应上线。

不要在实验过程中反复看结果并在“刚好显著”时停止；这会抬高假阳性。若必须序贯决策，应使用预先选定的方法，而不是临场发挥。

3.6.10 10. 在 x-algorithm 中怎样对应

阅读 Phoenix ranking 模型，区分各预测头和后续分数组合；阅读 Home Mixer scorer、selector 与 blender，标出 raw score 在哪里产生、在哪里被调整、哪里应用资格与间距约束。

特别关注：缺失预测的处理、参数来源、候选顺序、稳定性、广告与自然内容边界，以及 side effect 中记录了哪些实验和曝光证据。

公开仓库可作为架构和代码契约证据，但不能恢复 x 团队真实线上权重、训练标签、实验结论或完整指标定义。教材中的 A—F 数值是可手算教学数据，不应冒充生产配置。

3.6.11 11. 带入 Ranking 章的判断标准

- 对 A—F 保存 raw predictions、utility 与 final position；
- 多目标权重和模型版本进入参数快照；
- 相同输入得到稳定次序；
- 缺特征、NaN、无预测和全部同分有明确行为；
- 手算 `Precision@K`、`Recall@K`、`NDCG@K` 并处理 IDCG=0；
- 按新老用户或至少两个切片报告指标与样本数；

- 构造一次 train/serve skew 并让测试捕获；
- 写出一份含主要指标、质量护栏和系统护栏的实验计划。

3.6.12 12. 自检问题

1. 为什么预测概率、业务效用和最终位置不能共用一个 `score` 字段？
2. 区分能力很好但校准很差的模型能否正确排序？有什么风险？
3. 为什么稳定 tie–break 会影响实验和事故复现？
4. NDCG 的位置折扣表达了什么产品假设？
5. 未曝光内容为什么不能直接记为负样本？
6. 时间切分能避免哪些泄漏，又不能解决哪些偏差？
7. 同名 feature 在训练和服务中可能怎样语义不同？
8. 硬约束与软目标冲突时，谁应优先？
9. 点击率显著上涨为何仍可能不应发布？
10. 哪些线上结论无法从公开源码静态审计得到？

检查点：能否解释每一次位置变化

一个可信的排序系统，不只输出 A—F 的顺序，还能回答每个预测来自哪个版本、效用怎样计算、哪些约束改变了位置、离线指标有什么偏差，以及上线失败时怎样回滚。

3.7 Phoenix Ranking：把产品目标变成分数

3.7.1 现场：最相似的内容不一定排第一

B 与小林兴趣最相似，但 A 可能更及时、更可信；某条内容能带来点击，也可能带来隐藏或举报。Ranking 不是预测一个抽象“喜欢程度”，而是把多种行为预测和产品取舍组合成可执行顺序。

> **Tip：预测值不是最终分数**
>
> Phoenix 公开模型明确输出离散行为 logits 和连续预测值；只有经过合适激活与校准的输出才可解释为概率。业务 scorer 再按权重、归一化和多样性规则组合。读代码时把“预测”和“决策”分成两层。

3.7.2 在原仓定位：模型预测与业务组合

阅读：

- `phoenix/recsys_model.py`；
- `phoenix/grok.py`；
- `phoenix/test_recsys_model.py`；
- `home-mixer/scorers/phoenix_scorer.rs`；
- `home-mixer/scorers/ranking_scorer.rs`；
- `home-mixer/scorers/author_diversity_scorer.rs`。

重点追 candidate-isolation mask：candidate 可以读取用户历史和自身，却不应读取同批其他 candidate。再追多行为预测怎样进入业务权重和最终 score。

3.7.3 原理与边界：分数是一份可执行价值声明

线性组合可以写成 `score = sum(weight_i * prediction_i)`；若某项被解释为概率，必须先说明激活与校准。还要明确权重单位、负反馈符号和缺失预测。业务总分本身不是概率，高分不保证长期价值，线上互动提升也不自动意味着生态改善。

Candidate-isolation mask 阻止 candidate 读取同批其他 candidate 内容，但完整的排列等变还受位置编码、预处理和数值内核影响。Raw model prediction 应尽量满足这种隔离；后续作者多样性等业务 scorer 会有意读取同批候选，因此最终分数和顺序不再具有同样的不变性。排序还需要稳定 tie-break、NaN 处理和版本可追溯。

3.7.4 构造：实现多目标 Ranking

为 A/B/C 提供固定的 like、reply、repost、click、hide 预测值，并明确哪些只是 logits、哪些经过校准。实现权重组合、负反馈惩罚、作者多样性衰减和稳定排序。权重来自请求参数快照，输出记录每项贡献，而不只给总分。

再实现一个简化 isolation test：对某 candidate 的预测函数只允许读取用户上下文和自身特征。即使你的模型只是确定性函数，也要把接口边界写出来。

> **验收契约：顺序可以解释也可以复现**
>
> 相同输入、模型版本和参数版本得到相同顺序；NaN/缺失预测有明确策略；负反馈不会因权重符号错误变成奖励；添加无关 candidate 不改变其他 candidate 的原始预测。

3.7.5 破坏并证明：权重、排列和批次污染

至少测试：

- 调高 click 权重改变排序；
- hide 权重误设为正数；
- 输入 candidate 顺序随机排列；
- 加入一个极端高分 candidate；
- 某一行为预测缺失或 NaN；
- 模型版本与权重版本不匹配。

使用变形测试分别检查 raw prediction 的排列等变/isolation，以及业务多样性阶段预期产生的 batch dependence。选做运行 Phoenix 官方 ranking tests，区分其 mask/shape 证据与真实产品收益。

3.7.6 验收：目标函数不再藏在黑箱里

- 预测与业务组合分层；
- 每个行为贡献可解释；
- 稳定性、NaN、缺失和版本策略明确；
- raw prediction 的 isolation 与业务 scorer 的 batch dependence 被分别测试；
- NDCG、校准和线上指标职责分开；
- 排序输出可以进入最终产品编排。

3.8 Selection、Blending 与产品约束

3.8.1 现场：最终 Feed 不是一张分数排行榜

帖子、广告、推荐关注和提示模块拥有不同语义。即使 F 的商业分数很高，也不能连续插入或贴着敏感内容；同一作者的高分帖子也不应占满页面。Selection 选择集合，blending 编排序列。

> **新手 Tip：先手推，再写 blender**
>
> 混排算法常被边界条件淹没。先用 A—F 在纸上执行位置规则，写出空供给、供给不足、末尾位置和重复 ID 的期望，再开始编码。

3.8.2 在原仓定位：从 top-K 到页面节奏

阅读：

- `home-mixer/selectors/top_k_score_selector.rs`；
- `home-mixer/selectors/blender_selector.rs`；
- `home-mixer/ads/safe_gap_blender.rs`；
- `home-mixer/ads/partition_organic_blender.rs`；
- `home-mixer/candidate_pipeline/for_you_candidate_pipeline.rs`。

把算法拆成候选分区、位置计算、约束检查、供给不足处理和最终截断。记录帖子 pipeline 与外层 feed item pipeline 的边界。

3.8.3 原理与边界：集合优化与序列约束

Top-K 只回答"谁入选"，不回答"按什么节奏出现"。Blending 同时处理广告间距、敏感内容邻接、模块固定位置、作者多样性、最终长度和稳定性。约束可能互相冲突，因此要明确优先级与无解时的降级。

不要在 blender 中重新做安全判断的全部逻辑，但 blender 可以消费已验证的标签来保护邻接约束。原仓 SafeGap 的具体语义是避开与 `BrandSafetyVerdict::MediumRisk` 帖子相邻的 gap，并结合广告间距；它不是所有安全策略的统称。任何截断、插入或分页之后都必须重新验证约束，原仓还会在截断后处理末尾广告。

3.8.4 构造：让 A—F 成为一个页面

实现稳定 top-K 和一个最小 safe-gap blender：A/B/C 是 organic，F 是广告，D/E 已被过滤。广告最小间距、末尾不为广告来自本章核心；同作者连续上限是读者项目增加的教学扩展，不要误写成 SafeGap 源码本身。

输入不足时优先返回较短但合法的 feed，不得复制广告或放松安全规则凑数。每个输出 item 保留原 candidate ID、类型、来源和最终位置理由。

> **验收契约：页面合法比凑满更重要**
>
> 无论 organic/ads 供给如何，结果无重复、同时满足 ad-to-ad spacing、brand-safe adjacency 和长度上限；相同输入稳定；约束冲突时按书面优先级降级；最终位置可追溯。

3.8.5 破坏并证明：供给不足与约束冲突

测试零广告、广告过多、organic 只有一个、相同作者占满 top-K、敏感内容位于边界、最终截断和相同分数。再把“末尾不能是广告”和“固定广告位置”设成冲突，要求实现报告无法满足的约束，而不是静默输出非法页面。

指标包含 input by type、selected contribution、constraint rejection、final size、重复率和各位置类型。受控 page size 下 position 通常是有界标签；candidate ID、author ID 和完整 constraint detail 才应留在 trace，不能成为常驻高基数标签。

3.8.6 验收：推荐请求终于成为产品结果

- Selection 与 blending 职责分开；
- A/B/F 的位置可逐步解释；
- 供给不足不会突破硬约束；
- 约束冲突拥有显式降级；
- 最终 item 保留身份和来源；
- 第一条完整推荐请求具备测试、日志和指标证据。

4 第三部：状态、事件与异步工作

请求返回不是故事终点。实时帖子、缓存、served history、后台事件和内容理解任务会改变下一次请求。本部要求你把系统拆成多个本地进程，并亲手处理乱序、重复、重启、持久化边界、任务图与失败隔离。

4.1 间章 F：先写状态机，再写并发代码

大型 Tip · 间章：重复、乱序和重启下仍然成立的业务规则

本章放在 Thunder 与有状态组件之前。分布式系统没有一条可靠的“按代码从上到下执行”的故事；状态机把命令、事件、状态、不变量和拒绝条件写成可以测试的规则。

4.1.1 1. 状态机是对历史的压缩

系统接收一串输入，并把历史折叠成当前状态。设状态为 S_t，输入事件为 e_t，转移函数为 δ：

$$S_{t+1} = \delta(S_t, e_t)$$

如果转移是确定的，相同初始状态和相同有序事件序列应得到相同结果。这使重放、恢复、性质测试和跨实现对照成为可能。

图：命令、事件与状态不要混在一起

```
客户端意图                  已接受的事实                  当前投影
Command                      Event                      State
CreatePost(A)  -validate-> PostCreated(A,v1) -apply-> posts[A]=v1
DeletePost(A)  -validate-> PostDeleted(A,v2) -apply-> tombstone(A,v2)
```

命令可以被拒绝；事件一旦持久接受，就代表已经发生的事实。

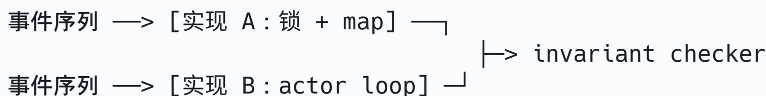
普通 CRUD 容易把“请求”和“已发生事实”都写成一次数据库 update。遇到重试、乱序、回放和多副本时，这个模糊边界会立即暴露。

4.1.2 2. 先定义不变量，而不是先选锁

不变量是在每个可观察状态都必须成立的条件。例如 Thunder-like post store 可以要求：

- 同一 post ID 的可见版本单调不降；
- 删除版本 v 的 tombstone 阻止任何 `version <= v` 的旧创建复活；
- 同一 event ID 重放不改变最终状态；
- 查询不会同时返回同一 post 的 active 与 deleted 表示；
- checkpoint 之后重放未确认事件与连续运行结果一致。

图：不变量是实现方案之上的护栏



只要两种实现都保护同一不变量，就可以比较性能而不改变语义。

“线程安全”不是业务不变量。一个 map 的每次读写都加锁，仍可能先应用删除、后应用旧创建，得到业务上错误但没有 data race 的结果。

4.1.3 3. 重复投递是常态，不是异常边角

网络超时只说明调用方没有及时看到结果，不说明服务端没有执行。调用方重试会让同一逻辑操作到达多次。消息系统在消费后、ack 前崩溃，也会再次投递。

在去重元数据仍处于保留窗口内，幂等意味着同一逻辑输入重复应用后，受保护的业务投影与应用一次相同：

$$\text{business}(\delta(\delta(S, e), e)) = \text{business}(\delta(S, e))$$

但这个公式需要先定义“同一逻辑输入”。request ID、event ID、业务键和 attempt ID 作用不同：

- request ID 关联一次端到端请求；
- attempt ID 区分重试尝试；
- event ID 标识一条持久事件；
- business key 标识真正只能生效一次的业务动作。

> **小心：反例：只用 event ID 去重**
>
> 两个不同 event ID 都表示“给订单 42 退款一次”。event 去重无法阻止业务重复。反过来，合法的第二次增量操作若复用了业务键，也可能被错误吞掉。

去重记录还需要作用域和保留期限。记录过早过期，迟到重复会再次生效；永久保存，则状态无限增长。系统必须把这个取舍变成显式契约。

4.1.4 4. 乱序要求版本，而不是“多等一会儿”

考虑同一 post A 的事件：

```
e1 = PostCreated(A, version=1)
e2 = PostUpdated(A, version=2)
e3 = PostDeleted(A, version=3)
```

传输顺序可能变成 `e3`，`e1`，`e2`。若无版本检查，A 会被旧 update 复活。版本化状态通常拒绝低于当前版本的事件并保留删除 tombstone；相同版本且 payload 相同可视为重复，相同版本但 payload 不同应报告生产者分叉/损坏。是否允许跳号还取决于事件是完整状态快照还是增量事件。

图：删除墓碑阻止旧事件复活

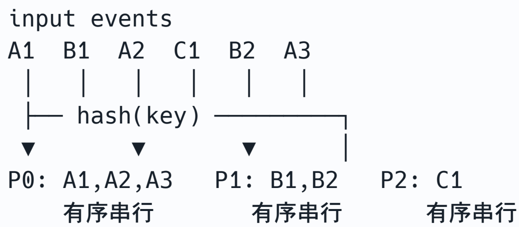
```
初始           收到 delete v3        迟到 create v1       迟到 update v2
 empty  ──>  tombstone(A,v3) ──>  reject(v1<=v3) ──> reject(v2<=v3)

错误实现：delete 直接移除 key
 empty  ──>  empty ──────────────> active(A,v1)  <- 已删除内容复活
```

版本必须来自能表达该实体顺序的权威，而不能用消费者本机 wall clock 临时生成。机器时钟偏移会让后到的旧事件看起来更新。

4.1.5 5. 全局顺序通常昂贵，也往往没有必要

不同 post 之间常不需要全局总序。A 的 v2 与 B 的 v7 谁先发生，可能不影响查询语义。按 entity key 分区，只保证同一 key 的顺序，可以提高并行度。

图：按 key 保序而不是全世界排队



分区之间无总序；每个 key 内保护版本不变量。

但重新分片或消费者扩缩容会产生旧 owner 与新 owner 并存窗口。此时只靠 key 内版本可能不够，还要用 partition epoch 或 fencing token 拒绝旧 owner 的写入。

4.1.6 6. 至少一次、至多一次与所谓恰好一次

在声明的 broker durability、ack 持久性和 offset 语义内，at-most-once 通常在处理前确认或不重试：可能丢，但不主动重复；at-least-once 在成功处理后确认：不轻易丢，但崩溃窗口会重复。Exactly-once 只有在明确事务范围内才有意义。

图：ack 位置决定失败窗口

```
at-most-once:
receive -> ack -> apply
               X crash：消息已确认，状态未更新，永久丢失

at-least-once:
receive -> apply -> ack
                  X crash：状态已更新，消息重投，需要幂等
```

如果事件系统与业务数据库不共享事务，即使 broker 宣称 exactly-once，也不能自动保证外部副作用只发生一次。应用仍需 inbox、业务幂等或 reconciliation。

4.1.7 7. checkpoint 是恢复起点，不是完整真相

状态可以通过完整事件日志重建，但日志越长，恢复越慢。checkpoint 保存某个已知位置的状态快照，恢复时加载快照，再重放之后的事件。

checkpoint 必须与消费位置一致。若先记录 offset、后写状态，崩溃会跳过未应用事件；先写状态、后记录 offset，则会重放，需要幂等。

图：一致 checkpoint 的恢复边界

```
event log:  101  102  103  104  105  106
                            ^
checkpoint state: applied through 103; next offset=104
恢复：load(state@103) -> replay 104,105,106

若 state 其实只应用到 102，却把 next offset 写成 104，事件 103 永久消失。
```

实际恢复测试必须杀死真实子进程、重新打开存储，再比较最终不变量。只在同一进程内抛异常不能证明文件刷新、事务提交或启动恢复逻辑。

4.1.8 8. 状态机也需要资源边界

一个逻辑正确的消费者仍可能被热点 key、坏事件或无限重试拖垮。状态机外围必须定义：队列上限、每 key 并发、最大事件大小、重试预算、poison event 隔离、checkpoint 频率和状态清理策略。

若某个 A 事件永久解析失败，不能让它永远堵住 B—F。隔离后也不能悄悄遗忘；要记录 quarantine/DLQ、oldest age、错误原因和人工恢复步骤。

4.1.9 9. Thunder 与独立事件样例的源码映射

状态机实验使用独立实体 `post-7`，不改变全书 A—F 候选身份：依次发送 create(v1)、重复 create(v1)、update(v2)、delete(v3)、迟到 create(v1) 和未知 schema。另用 `hot-author` 生成容量压力，检查 store 的版本、墓碑、去重和资源隔离。

阅读 Thunder 的 handler、state、event 与 checkpoint 相关代码，先画出公开实现真正存在的状态转移，再标出没有公开证明的 broker、部署和恢复边界。不要仅凭文件名推断完整生产语义。

同时回看 Home Mixer 的 Thunder source：它消费的是 Thunder 暴露的查询契约，而不是内部事件流。服务边界应该隐藏状态重建方式，但公开新鲜度、部分结果和失败原因。

4.1.10 10. 用模型测试，而不是枚举几个 happy path

为小状态机写一个极简参考模型，然后随机生成 create/update/delete/duplicate/reorder 序列，让真实实现与模型逐步对照。重要性质包括：

- 同一事件重复任意次，最终状态不变；
- 对不同 key 的独立事件交换顺序，最终状态相同；
- 低版本事件永远不能覆盖高版本；
- checkpoint + replay 等价于不中断连续执行；
- 查询结果永远满足 active/tombstone 互斥。

> **小心：反例：测试只断言最终数量**
>
> 最终都剩 5 条 post，不代表留下的是正确版本，也不代表删除内容没有复活。测试必须检查实体状态、版本、墓碑和可观察原因。

4.1.11 11. 带入 Thunder 章的判断标准

- 写出命令、事件、状态和转移表；
- 至少列出五条可机器检查的不变量；
- 为 `post-7` 构造重复、乱序、迟到创建/删除和未知 schema；
- event ID 去重与业务幂等分开设计；
- checkpoint 与 offset 有一致性方案；
- 真实进程重启后可恢复并重放；
- 热点 key、poison event 和积压可观测；
- 对同一随机事件序列，参考模型与实现结果一致。

4.1.12 12. 自检问题

1. 命令和事件为什么不能视为同一种对象？
2. “没有 data race”为何不能证明业务状态正确？
3. request ID、attempt ID、event ID 与 business key 各自解决什么问题？
4. 删除后为什么要保留 tombstone，而不是直接移除 key？

5. 为什么消费者本机时间不适合作为实体版本?
6. 哪些场景只需要 per-key order，哪些需要 epoch/fencing?
7. at-least-once 的重复窗口位于哪里?
8. checkpoint 的 state 与 offset 不一致会怎样丢数据?
9. 同进程异常测试为何不足以证明 crash recovery?
10. 你会选择哪些性质做随机状态机测试?

检查点：先证明转移，再优化吞吐

首次阅读只需能用同一组不变量解释 `post-7` 在重复、乱序、重放和重启后的最终状态；具体实现并入下一章，A—F 继续保留原有推荐身份。

4.2 Thunder：实时状态面对重复与乱序

4.2.1 现场：删除先到，帖子还会不会复活

单个 Kafka partition 可以提供分区内顺序，但多 partition、重放、批处理、回填和恢复后的端到端事件仍可能重复或相对乱序。若 delete 已经生效，迟到的 create 不能让同一帖子悄悄复活；若重复 create 消耗两份容量，热门作者会污染候选池。

> **Tip：先写状态转移表**
>
> 处理事件前先列出当前状态、到达事件、允许结果和拒绝原因。状态机表能比一串 `if` 更早暴露重复、乱序和未知事件问题。

4.2.2 在原仓定位：三类帖子索引与 tombstone

阅读：

- `thunder/posts/post_store.rs`；
- `thunder/thunder_service.rs`；
- `thunder/kafka/tweet_events_listener_v2.rs`；
- `thunder/deserializer.rs`。

记录 original 与 secondary 两个互斥时间线索引、可与二者重叠的 video 附加索引如何存放，删除怎样留下 tombstone，查询怎样按关注作者收集候选，以及 retention/扫描限制在哪里体现。公开实现中的 tombstone 是布尔状态，不要擅自声称它比较事件版本。

4.2.3 原理与边界：幂等、顺序和容量属于同一状态机

幂等要求同一业务事件重复到达不改变最终效果；乱序要求系统能拒绝已被更新事实覆盖的旧事件；容量要求无法无限保留所有历史。三者会冲突：过早清理 tombstone 节省内存，却可能让迟到 create 复活。

如果事件携带单调版本，可以比较版本；如果没有，只能采用保守 tombstone、有限保留窗口或从权威存储重新确认。不要假装不存在的信息可以由算法恢复。

4.2.4 构造：拆出第一个独立 Source 进程

把 Following source 拆成 Thunder-like 本地进程，接受 create/delete 事件并按 author 查询最近帖子。先实现与原仓公开语义一致的 bool tombstone，再把“版本化事件”作为你自己的扩展，明确标注差异。

公开 PostStore 是内存结构；持久化是课程扩展。核心路线可使用标准库追加日志；若所选语言的标准发行版内置 SQLite 接口，也可使用 SQLite。查询接口返回 candidate ID、author、created-at 和 source version。

> **验收契约：post-7 不会因迟到事件复活**
>
> 重复 create/delete 幂等；delete 后迟到 create 的行为与所选状态模型一致；重启后状态不倒退；过期清理策略说明 tombstone 风险；查询受容量和 deadline 约束。

4.2.5 破坏并证明：重放、重启与容量上限

固定事件序列：`create(post-7)`、重复 create、delete、迟到 create。分别在每个事件后杀进程并重启，再重放最后两条。检查最终状态、持久日志、重复计数和查询结果。

随后让单个作者持续写入，验证读者实现的 retention 与 per-author 上限；原仓可见的是 retention、查询 `max_per_user` 和扫描限制，不要把它们误写成相同的存储容量机制。禁止使用无界内存表；指标包含 active posts、tombstones、duplicate events、rejected stale events、restore duration 和 query size。

4.2.6 验收：实时意味着更快面对不完整世界

- Thunder-like source 已成为独立进程；
- 原仓 bool tombstone 与读者版本扩展明确区分；
- 重复、乱序和重启拥有测试；
- retention 与 tombstone 风险可解释；
- 查询有 deadline、容量和观测；
- A 的实时来源进入完整候选账本。

4.3 间章 G：缓存是一份会过期的副本

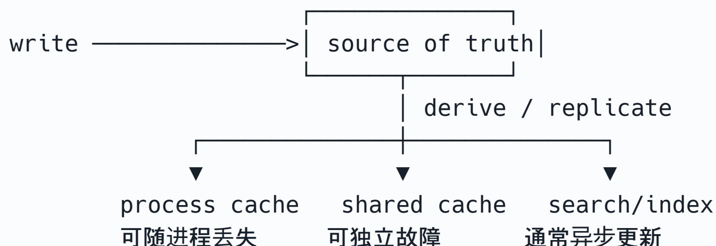
大型 Tip · 间章：从 cache hit 到一致性、击穿与恢复

本章放在 State 与 Cache 附近。缓存不是一个更快的 map，而是原始状态的副本；只要存在副本，就必须回答新鲜度、失效、并发、容量和故障时的语义。

4.3.1 1. 先问“它是谁的副本”

缓存价值通常来自减少昂贵计算、远程读取或数据库压力。但在写 `get(key)` 之前，必须写清 source of truth：谁拥有权威状态，缓存能否重建，丢失是否影响正确性。

图：权威状态与派生副本



副本之间可以暂时不同；契约必须规定允许不同多久、谁来修复。

若 served history 只存在缓存，缓存清空可能让已展示内容再次出现；这就不再是纯性能缓存，而是承载产品正确性状态。命名不会决定语义，故障后果才决定。

4.3.2 2. 命中率不是唯一目标

命中率定义为：

$$\text{hit rate} = \frac{\text{cache hits}}{\text{hits} + \text{misses}}$$

高命中率可能同时伴随大量陈旧值、热点不公平或源站雪崩。至少还要观察：load latency、value age、eviction、origin QPS、coalesced waiters、error rate、key cardinality 与内存使用。

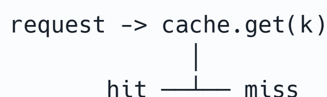
小心：反例：99% 命中率仍然事故

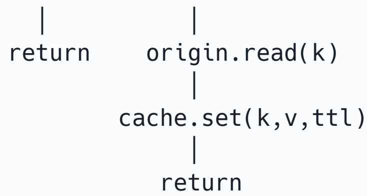
1% miss 集中在一个超贵的热点 key，每次都会并发击穿数据库。总体命中率漂亮，源站却因几百个相同请求瞬间过载。

4.3.3 3. cache-aside 的每条箭头都可能失败

cache-aside 读取路径：先查缓存，miss 时读权威存储，再回填。写路径通常先写权威状态，再删除或更新缓存。

图：cache-aside 读路径





```
        |          |
      return    origin.read(k)
                   |
              cache.set(k,v,ttl)
                   |
                 return

miss 后 origin 成功、cache.set 失败：请求仍可成功，但后续继续 miss。
```

这条路径要求区分：缓存 miss、缓存 unavailable、缓存值损坏和 cached negative。把缓存错误伪装成 miss 会隐藏基础设施故障，也可能突然把全部流量导向 origin。

4.3.4 4. TTL 只限制年龄，不保证业务新鲜

TTL 表示缓存项从某个起点经过多长时间后不可用。这个起点可能是生成时间、写缓存时间或最后刷新时间；三者不一定相同。

若 TTL 为 60 秒，值在计算前已经落后 5 分钟，写入后"还可用 60 秒"并不新鲜。应随值保存 `generated_at`、source version 或 logical version，而不是只依赖缓存服务器剩余 TTL。

图：TTL 与数据年龄不是同一个钟

```
source state v8 ──(延迟复制 5 min)──> cache.set(v8, ttl=60s)
source state v9 已经存在

cache TTL age: 2s         看起来很新
business data age: 302s   实际已经陈旧
```

wall clock 可能跳变；本进程计算 elapsed timeout 应使用 monotonic clock。跨系统数据年龄则需要时间戳、版本和允许的时钟误差假设。

4.3.5 5. 更新与失效存在并发竞态

经典 cache-aside 写法"更新数据库，然后删除缓存"仍有窗口：旧读取可能在数据库更新前读到旧值，却在删除之后把旧值回填进缓存。

图：旧值复活竞态

```
Reader R                        Writer W
cache miss
read origin -> old v1
                                write origin v2
                                delete cache
cache.set(v1)   <- 删除之后回填旧值
return v1

结果：缓存重新保存 v1，直到 TTL 或下一次失效。
```

可能的缓解包括短 TTL、带版本 compare-and-set、写入时携带 generation、延迟二次删除、串行化同 key 更新，或接受有界陈旧并用监控验证边界。没有一种策略对所有业务都最好。

4.3.6 6. 一致性需求必须按数据类型选择

对推荐系统中的不同状态，可以选择不同契约：

- candidate feature：允许短暂陈旧，但必须标注版本和 age；
- served history：当前请求后可能要求 read-your-writes，避免立刻重复；
- experiment assignment：同一用户需稳定，错误分桶会污染评估；
- safety eligibility：通常不能依靠长时间陈旧缓存；
- corpus index：允许异步，但要报告 index lag；
- 参数配置：需要版本兼容与快速回滚。

图：协调和故障条件下的常见代价，不是单一连续轴

数据类型	错误成本	允许陈旧	协调范围	分区时行为
safety block	极高	接近 0	权威状态	保守拒绝
served history	重复体验	会话内很小	session/token	overlay/降级
features	质量下降	数秒	异步副本	使用带 age 值
popular feed	通常较低	更久	本地缓存	继续服务

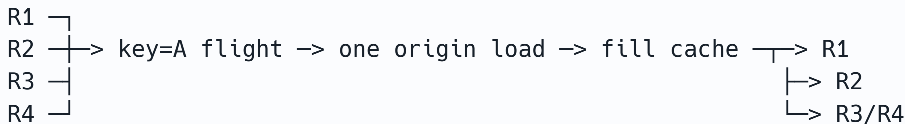
强一致并不必然高延迟；CAP 的可用性取舍只在网络分区条件下讨论。

read-your-writes 可以通过粘滞路由、版本 token、会话内 overlay 或读权威存储实现。每种方案都有故障和扩容边界。

4.3.7 7. stampede 需要请求合并与负载保护

热点 key 过期时，N 个并发请求可能同时 miss 并访问 origin，这叫 cache stampede。single-flight 让同一 key 同时只发生一次 load，其余请求等待同一个结果。

图：single-flight 合并热点加载



不同 key 仍应并发；同 key 的 waiter 必须有自己的 deadline/cancellation。

single-flight 不是无限等待。leader 超时或崩溃时，需要解除 flight；等待者取消时不能错误取消其他请求仍需要的共享加载。还应限制全局 origin 并发，防止许多不同 key 同时 miss。

可以使用 TTL jitter 避免大量 key 同时过期；可以 stale-while-revalidate，在允许陈旧的业务里先返回旧值并后台刷新。但 stale 值必须有最大年龄，后台刷新也需要并发和失败指标。

4.3.8 8. negative cache 也会把“暂时没有”固化

对不存在的 key 缓存 negative result 能保护 origin，但“没有”可能很快变成“有”：新 post 刚创建、用户刚获得权限、索引刚完成更新。Negative TTL 经常需要比正值更短，但具体取决于“不存在”状态的变化速度和 origin 成本；必须区分 not-found、forbidden、timeout 与 upstream error。

> **小心：反例：缓存所有异常**
>
> 把 origin timeout 作为 `None` 缓存 10 分钟，会把一次短暂故障变成十分钟稳定错误。只有语义明确且可安全复用的结果才应缓存。

4.3.9 9. key 设计决定隔离、版本和隐私

cache key 至少可能包含 user、locale、device class、model version、feature schema、experiment bucket 和权限上下文。漏掉维度会跨上下文复用错误值；维度过多则降低命中率、增加内存。

图：key 是隐含的函数参数列表

```
value = recommend(user, locale, model, policy, experiment)

坏 key： feed:{user}
示意 key： feed:{user}:{locale}:{device}:{model_v}:{policy_v}:{bucket}

若函数输入变了而 key 没变，缓存就在返回另一道题的答案。
```

版本化 key 可让新旧版本并存并支持回滚，但会短期双倍占用容量。删除旧 namespace 前要确认没有旧实例仍在读取。

缓存还可能包含敏感用户特征。必须定义租户隔离、访问控制、日志脱敏、加密需求与删除传播；“只是缓存”不是降低隐私责任的理由。

4.3.10 10. 容量与淘汰是正确性输入

容量不足会触发 LRU/LFU 等淘汰策略。若应用假设某键“应该一直在”，淘汰就会暴露隐藏错误。任何可淘汰缓存都必须允许 miss 并正确重建，或者明确它不是缓存而是权威状态。

粗略容量估算：

$$\text{memory} \approx \text{keys} \times (\text{key bytes} + \text{value bytes} + \text{overhead})$$

不要忽略对象、索引、allocator 和复制开销。还要估算 miss 后的 origin 容量；缓存集群整体重启时，冷启动负载往往比正常稳态更危险。

4.3.11 11. A—F 与原仓映射

用 A—F 验证缓存语义，但不改变固定身份：A 是 Following 内容，展示后应进入 served history；B 的 Phoenix feature 可能命中旧版本；C 本来就是旧缓存内容，仍要检查删除、权限与已展示事实；D 来自被屏蔽作者，缓存不能绕过权限；E 的安全加载超时不能缓存成“安全”或“不存在”；F 是广告，若有独立缓存也必须使用广告/实验专属 key，不能伪装成 organic fallback。

阅读 Home Mixer 的 Redis candidate cache、served history side effect、query/candidate hydration 与参数代码，标出 key、TTL、错误处理、版本和写入位置。公开快照若没有展示完整 Redis 拓扑、持久性或失效协议，就必须明确记为未知。

4.3.12 12. 带入缓存章的判断标准

- 为每种缓存写出 source of truth 与可重建性；
- miss、unavailable、corrupt、negative 四种状态可区分；
- 值携带生成时间或逻辑版本；
- 重现旧值回填竞态，并实现一种有说明的缓解；
- 对热点 A 实现 per-key single-flight 和 waiter cancellation；
- 对过期风暴加入 jitter、并发上限或受控 stale；
- key 包含所有影响结果的上下文和版本；
- 运行冷缓存压测，记录 origin QPS、P99、拒绝与恢复时间。

4.3.13 13. 自检问题

1. 为什么 served history 可能不是普通性能缓存?
2. cache miss 与 cache unavailable 混淆会怎样放大故障?
3. TTL age 与 business data age 为什么不同?
4. “写数据库再删缓存”为何仍可能复活旧值?
5. 哪些 A—F 状态需要 read-your-writes，哪些允许陈旧?
6. single-flight leader 超时后，等待者应该发生什么?
7. negative cache 为什么不能缓存 timeout?
8. cache key 少了 experiment bucket 会怎样污染实验?
9. 为什么缓存重启时要按 origin 容量设计，而不是按稳态命中率设计?
10. 怎样用指标区分高命中但陈旧，和真正健康的缓存?

检查点：把每个 cache 当作复制协议

当你能指出权威来源、复制触发、允许陈旧、失效方式、并发竞态、容量边界和恢复证据时，缓存才从“性能魔法”变成可运营的分布式组件。

4.4 缓存、Served History 与一致性承诺

4.4.1 现场：缓存说还有 C，历史说别再展示

候选缓存追求低延迟，served history 保护用户不要立刻看到重复内容。两者都可以存 ID，却拥有完全不同的一致性要求。缓存陈旧可能降低相关性，history 丢写则会直接产生重复体验。

> **新手 Tip：一致性先翻译成用户承诺**
>
> 不要先问“要不要强一致”。先问：允许多旧？用户写后是否必须立刻读到？能否重复？错误结果和无结果哪个更坏？答案会自然导向不同实现。

4.4.2 在原仓定位：命中、写回与展示历史是三条线

阅读：

- `home-mixer/query_hydrators/cached_posts_query_hydrator.rs`；
- `home-mixer/sources/cached_posts_source.rs`；
- `home-mixer/side_effects/redis_post_candidate_cache_side_effect.rs`；
- `home-mixer/query_hydrators/served_history_query_hydrator.rs`；
- `home-mixer/side_effects/update_served_history_side_effect.rs`；
- `home-mixer/filters/previously_served_posts_filter.rs`。

画出 read path、write path、key、数据年龄和失败影响。注意缓存 source 可能改变实时 source 的 enable 行为，不能默认所有 source 永远同时运行。

4.4.3 原理与边界：新鲜、命中和正确是三件事

Cache hit 只说明 key 存在；fresh 表示年龄在契约内；correct 还要求身份、权限和安全状态仍有效。Served History 通常希望得到更强的 read-your-writes 或保守过滤，因为用户刚看过的事实应尽快生效；这是课程要建立的产品契约，公开 detached side effect 调用点本身并未证明线上写后读保证。

single-flight 可以合并同一 hot key 的并发加载，但需要处理 leader caller 取消、loader 失败和 inflight 清理。hot key 与分片不均匀也不同：key 数量均匀不代表请求量均匀。

4.4.4 构造：实现两份不同的状态契约

为 Candidate Cache 和 Served History 分别写契约并实现：TTL、写入时刻、读取策略、缺失/陈旧行为、read-your-writes、最大集合、失败降级和指标。进程内测试使用可控 monotonic clock；跨进程、跨重启的持久过期需要绝对 expiry、存储端 TTL、版本或权威时间，不能持久化本地 monotonic 数值。

当 cache 返回 C，但 history 表明 C 已展示时，最终必须删除 C。Cache 的 C 还要重新通过当前安全与权限检查，不能因为“之前合法”而跳过过滤。

> **验收契约：缓存不能成为规则旁路**
>
> stale 值有显式年龄；history 写后当前用户能够按契约读到；同 key 并发 miss 最多触发一次 loader；失败后 inflight 状态清理；缓存候选仍经过当前过滤。

4.4.5 破坏并证明：陈旧、热点和写后读

测试 TTL 边界、wall clock 回拨、loader 失败、首个 caller 取消、8 个并发 miss、history 写失败和缓存中已被删除内容。再制造小林这个 hot key，观察总 key 分布均匀但单 shard 请求过热。

记录 hit/fresh/stale/miss、loader calls、inflight、history write/read lag、duplicate prevented 和 hot-key QPS。禁止把 user ID 直接放进常驻指标标签。

4.4.6 验收：系统不只记得值，也记得值有多旧

- Cache 与 Served History 契约不同；
- 进程内 monotonic TTL 与跨重启绝对过期分别可测试；
- C 的缓存事实与展示事实分别可解释；
- single-flight 处理失败和取消；
- hot key 风险可观测；
- 当前过滤规则不会被缓存绕过。

4.5 Grox：任务图、资源池与失败隔离

4.5.1 现场：任务都异步了，内存为什么还在涨

内容理解通常包含加载、转写、摘要、分类、embedding 和结果写入。把每一步 `create_task` 并不会自动产生吞吐控制；如果入口速度高于完成速度，无界任务和队列最终会吃光内存。

> **Tip：异步不是容量策略**
>
> `async` 只说明等待期间可以让出执行权。系统能否稳定，还取决于入口速率、服务速率、并发池、队列上限、拒绝和重试。

4.5.2 在原仓定位：从 Dispatcher 追到 Plan

静态阅读：

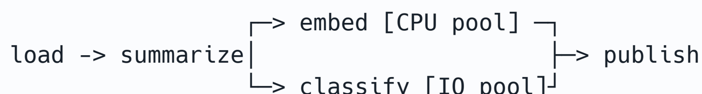
- `grox/dispatcher.py`；
- `grox/engine.py`；
- `grox/plans/plan.py`；
- `grox/plans/plan_master.py`；
- `grox/tasks/task.py`；
- `grox/generators/task_generator.py`。

再选择一个 `plan_*.py` 画 DAG。公开快照缺少多个内部模块，并存在不能直接运行的代码，因此本章证据等级是可审计。重点找任务依赖、调度入口、资源边界、重试与结果写入，而不是尝试补齐整个生产环境。

4.5.3 原理与边界：图正确、容量正确、失败正确

DAG executor 至少要验证节点存在、无环、依赖完成后才调度、失败如何传播。资源控制要按 CPU、网络、模型或外部 API 分池，否则一个昂贵任务会阻塞所有轻任务。重试必须由统一 budget 管理，不能每个节点独立乐观重试。

图：依赖就绪与资源池是两套约束




```
节点进入 ready queue：所有依赖完成
节点真正开始运行：对应资源池仍有容量
失败传播：按契约 skip descendants、继续独立分支或终止整图
```

队列长度不是越大越安全。大队列把过载转化为陈旧结果和长尾延迟。内容理解结果如果过期，完成也可能已经没有业务价值。

4.5.4 构造：实现最小 DAG Executor

在读者项目中实现拓扑排序、cycle detection、节点状态和依赖调度。固定 DAG 明确分叉与汇合：`load -> summarize`，随后并行进入 `embed` 与 `classify`，二者完成后再 `publish`。Summarize/classify 使用受限网络池，embed 使用保留容量的 CPU 池。

加入每任务 deadline、最大并发、有界等待室和拒绝策略。输出必须记录 DAG ID、task ID、attempt、依赖、queue wait、run time、status 和结果版本。

验收契约：任务图必须闭合

有环或缺节点在执行前失败；任务只在依赖成功后启动；不同资源池保留容量并限制相互干扰，但不虚构 CPU、内存和 event loop 的绝对隔离；队列满时显式拒绝；请求取消后未开始任务不再运行；重试服从 DAG 总预算。

4.5.5 破坏并证明：环、慢节点和重试放大

注入 cycle、缺失依赖、embed 变慢、publish 暂时失败、队列满和 worker 重启。验证失败任务是否阻止错误下游、独立分支是否可以继续、重试次数是否受限，以及过期任务是否被丢弃。

计算 arrival rate、service time 和在途任务，记录 accepted/rejected、queue depth、queue age、active by pool、attempt amplification 和 expired-before-run。

4.5.6 验收：异步工作开始为吞吐负责

- 能从原仓静态抽取一张真实 DAG；
- 自建 executor 拒绝环和缺失依赖；
- 资源池、队列和 deadline 明确；
- 慢节点不会拖死全部任务；
- 重试与过期有统一预算；
- DAG 行为有确定性测试和容量证据。

4.6 间章 H：响应成功之后，工作还没有结束

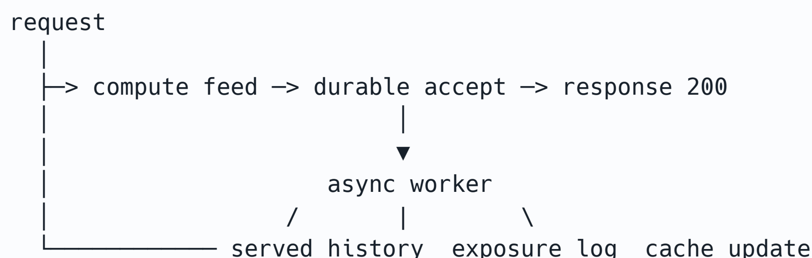
大型 Tip · 间章：跨事务边界的可靠事件与工作流

本章放在 Side Effects 附近。它把“开一个后台任务”改写成可推理的持久工作流：哪里算接收成功、哪里可能丢、哪里可能重复、怎样补偿、怎样最终发现并修复不一致。

4.6.1 1. 先把用户响应与后台完成分开

小林收到 A/B 后，系统还可能写 served history、更新缓存、记录曝光、发送训练事件和计费。把所有操作同步放在响应前会增加尾延迟；把它们交给内存后台任务，则在进程退出时可能消失。

图：一次请求有不止一个完成点



200 只证明响应路径的契约；不自动证明三个下游已经完成。

需要分别定义：computed、durably accepted、delivered、applied 与 externally visible。一个模糊的 `success=true` 无法表达这些阶段。

4.6.2 2. durability 是“崩溃后还能找到”

对象写进进程内 queue，不代表持久。真正的 durable accept 至少要求在进程崩溃并重新启动后，系统仍能发现待处理工作。具体保证取决于事务提交、文件系统刷新、数据库配置和存储故障模型。

小心：反例：create_task 不是消息队列

请求 handler 创建异步任务后立即响应。发布滚动升级正好终止进程，任务连同内存一起消失；调用方已收到成功，也不会重试。

本地教学项目可以用 SQLite 表示 durable queue，但必须真实杀死子进程并重新打开数据库。不能只在同一进程捕获异常后继续。

4.6.3 3. 双写问题来自两个独立提交点

假设请求既更新业务数据库，又发送事件到 broker。两者没有共享事务时，总存在 crash window。

图：朴素双写的两个失败方向

```
方案 A：write DB -> publish event
        success      X crash
结果：业务已变，事件丢失

方案 B：publish event -> write DB
        success         X crash
```

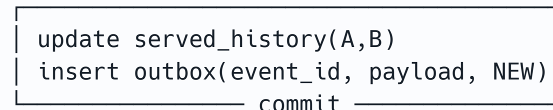
```
结果：事件已见，业务未变

交换顺序只交换哪一种不一致更容易发生，不能消灭双写问题。
```

如果业务变化与事件必须一致，可把业务状态和 outbox row 写进同一个本地事务。独立 relay 再把 outbox 发送到外部系统。

4.6.4 4. Transactional Outbox 关闭“状态已写但事件没记”的缝

图：outbox 的事务边界

```
request transaction
```




```
                     |
                     ▼
relay: NEW -> publish -> mark SENT
```

outbox 能保证业务写与“待发送意图”一起提交；它不能自动保证外部 broker 只收到一次。relay 在 publish 成功后、mark SENT 前崩溃，会再次 publish。因此消费者仍要幂等。

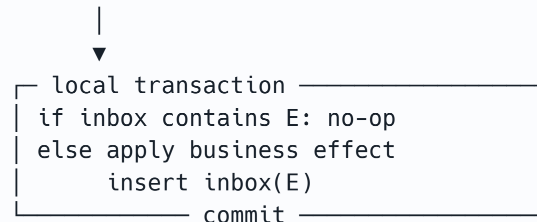
outbox row 至少应包含 event ID、aggregate/business key、schema version、payload、created time、attempt count 与状态。payload 应保存产生事件当时的必要事实，不能在几小时后重新查询并假定状态没变。

4.6.5 5. Inbox 把重复投递变成显式协议

消费者可以在本地事务中检查/插入 inbox event ID，并应用业务变化。若 inbox 与业务状态同库，就能原子地做到“这条事件是否已在本地生效”。

图：consumer inbox 与 ack

```
receive event E
      |
      ▼
```




```
      |
      ▼
ack broker

commit 后、ack 前崩溃：E 会重投，但 inbox 使它不重复生效。
```

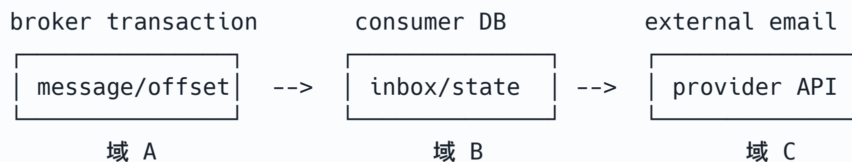
event ID 幂等不一定等于业务幂等。支付、配额、通知等副作用还要用业务键保护其真实“一次”语义。调用外部系统时，应把 idempotency key 传播出去，并理解对方保留多久。

4.6.6 6. 投递语义由整个链路共同决定

at-most-once 倾向不重复但可能丢；at-least-once 倾向不丢但会重复。“exactly once”必须带上范围，例如“同一数据库事务中，某 event ID 对本地表只生效一次”。

只要跨出事务域——发邮件、调用第三方、写另一个数据库——恰好一次就不再由单个 broker 开关保证。应用要依赖幂等、查询确认、补偿或 reconciliation。

图：所谓 exactly-once 的边界



每跨一个独立提交域，都要重新说明失败、重复与确认语义。

4.6.7 7. retry 是额外负载，不是免费的可靠性

重试只适合暂时性、可安全重做的操作。语法错误、权限拒绝、未知 schema 等永久错误，重试不会变好。服务已经过载时，立即重试会放大事故。

指数退避可写为：

$$\mathrm{delay}_n = \min(\mathrm{cap}, \mathrm{base} \times 2^n), \quad n = 0, 1, \ldots$$

多个 worker 若使用完全相同 delay，会同步醒来形成重试脉冲，所以应加入 jitter。请求内同步 attempt 服从 request deadline；已经 durable accepted 的后台 job 使用独立的 job expiry/completion SLO、最大年龄和 attempt budget；Saga 使用 workflow deadline，reconciliation 使用修复窗口。不能把所有后台工作绑在已经结束的用户请求 deadline 上。

图：分层重试的乘法

```
client 3 attempts
   └ API 3 attempts each
        └ worker 3 attempts each

最坏下游调用：3 × 3 × 3 = 27
一次用户动作可能变成 27 次压力。
```

4.6.8 8. poison event 不应堵住整个世界

某条事件若因永久数据问题反复失败，应在有限次数后进入 quarantine/DLQ，使后续健康事件继续处理。但 DLQ 不是垃圾桶；它需要告警、所有者、查看工具、修复规则、重放操作和审计。

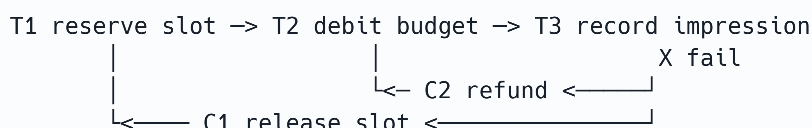
若同一业务 key 需要严格顺序，跳过坏事件可能让后续事件失去前置条件。可选择暂停该 key、隔离该 partition，或把 key 状态标为需要修复，而不是无条件全局继续。

> **小心：反例：DLQ 数量为零不代表健康**
>
> worker 可能无限重试，消息从未达到 DLQ；也可能解析失败后直接丢弃。要同时观察 attempts、oldest pending age、processing lag、drop 与状态对账。

4.6.9 9. Saga 用补偿表达跨服务业务流程

当一次工作流跨多个服务，无法使用单一 ACID 事务，可以把它拆成多个本地事务，并为已完成步骤定义补偿。例如：保留广告位、扣减预算、记录曝光；后一步失败时释放前面的预留。

图：Saga 不是自动回滚



补偿是新的业务动作，也会失败、重复和乱序；它不是数据库 undo。

有些动作不可真正撤销：邮件已经被阅读，推荐已经被用户看到。此时只能做语义补偿，例如后续更正、退款、屏蔽未来曝光，并保留审计记录。

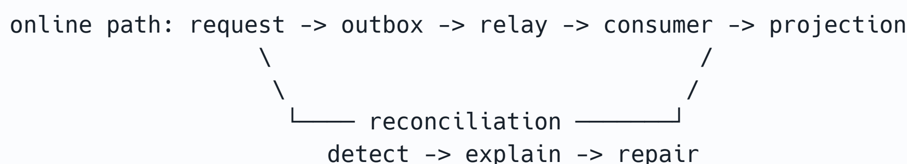
编排式 saga 由 coordinator 保存步骤状态；协同式 saga 由事件驱动各服务。前者更易观察流程，后者耦合较松但全局因果更难追踪。选择取决于流程复杂度、所有权和故障诊断需求。

4.6.10 10. Reconciliation 是最后的正确性防线

即使使用 outbox、inbox 和幂等，bug、人工操作、存储损坏和未知故障仍会制造不一致。reconciliation job 定期比较两个系统应满足的关系，并生成修复或人工工单。

例如：对声明为 durable 的已返回 feed，应在约定窗口内存在 served-history 记录与曝光事件；对每个 outbox SENT，要先定义 SENT 是 broker ack、consumer applied 还是 projection visible，再在 reconciliation window 和目标记录保留期内查找 event ID。缓存则可通过权威状态重建。

图：在线路径与离线对账共同闭环



对账不应只给出 mismatch count，还要保存差异样本、首次出现时间、修复尝试、责任系统和是否收敛。自动修复本身也必须幂等、限速和可回滚。

4.6.11 11. schema evolution 是工作流的一部分

持久队列意味着旧消息可能在新代码上线后才被消费。事件 schema 必须带版本，并为 unknown field、unknown enum、缺失字段和新旧消费者共存定义行为。

发布顺序通常采用 expand-and-contract：先让消费者兼容新旧格式，再让生产者写新格式，等待旧消息排空，最后移除兼容代码。回滚时，新生产者已经写出的事件仍必须被旧版本安全处理。

图：滚动升级中的四种组合

```
producer v1 -> consumer v1    基线
producer v1 -> consumer v2    v2 必须读旧格式
producer v2 -> consumer v2    新路径
producer v2 -> consumer v1    回滚/混跑的危险组合
```

只测试 v2->v2，不能证明可以滚动发布和回滚。

4.6.12 12. A—F 与原仓映射

一次返回 A/B/C 后，构造事件：A 写 served history；B 曝光日志重复投递；C 在 ack 前崩溃；D 使用新 schema；E 是永久坏 payload；F 下游持续超时。观察每条工作怎样进入 pending、retry、quarantine 或 completed。

阅读 Candidate Pipeline 的 side-effect trait 与调度位置，再读 Home Mixer 的 served history、Kafka、cache 和实验 side effects。标注它们在公开代码中可见的调用契约；不要假定仓库已经公开 Kafka 事务、Redis 持久性、部署拓扑和完整恢复流程。

4.6.13 13. 可观测性必须对应状态机阶段

建议指标包括：accepted、pending、processing、completed、failed、retried、deduplicated、quarantined、oldest pending age、end-to-end completion latency 与 reconciliation mismatch。

trace 应保留 request ID、event ID、business key、attempt ID、schema version 和 worker instance。日志不能只写“failed”；要能回答在哪个阶段、是否会自动重试、用户响应是否已成功、是否需要人工介入。

4.6.14 14. 带入 Side Effects 章的判断标准

- 对丢失会违反声明契约的工作，响应成功前完成相应 durable accept；允许 best-effort 的工作明确丢失预算、指标和重建方式；
- 业务状态与 outbox 在同一事务提交；
- relay 在 publish 后、mark 前崩溃会重复但不丢；
- consumer 在 apply 后、ack 前崩溃会重放但不重复生效；
- retry 有分类、退避和 jitter；同步请求、后台 job、Saga、reconciliation 各自使用正确的 deadline/expiry 与预算；
- poison event 隔离后可诊断、修复和受控重放；
- 至少一个跨步骤流程有补偿与补偿失败路径；
- reconciliation 能检测并修复人工制造的不一致；
- v1/v2 producer-consumer 四种组合有兼容性测试。

4.6.15 15. 自检问题

1. HTTP 200 可以证明后台工作的哪一个阶段?
2. 为什么内存 queue 不能满足 durable accept?
3. 调换“写 DB”和“发消息”的顺序为何无法解决双写?
4. outbox 解决了什么，又没有解决什么?
5. inbox event ID 去重与业务幂等为什么仍需分开?
6. broker 的 exactly-once 为什么不能保证外部邮件只发送一次?
7. 三层各重试三次为何可能产生 27 次下游调用?
8. poison event 被隔离后，per-key 顺序可能怎样受影响?
9. 补偿为什么不是回滚，它自己会有哪些失败?
10. 已有 outbox/inbox 后，为什么仍需要 reconciliation?
11. 哪种 producer-consumer 版本组合最容易在回滚时被遗漏?
12. 哪些指标能区分“已接收”“正在积压”和“最终完成”?

检查点：把 crash window 逐条关掉

可靠工作流不是承诺永不失败，而是让每个提交点都能回答：崩溃后工作在哪里、会不会重复、谁来继续、怎样观测、怎样补偿，以及长期怎样验证两个系统重新收敛。

4.7 Side Effects：响应之后怎样不忘记

4.7.1 现场：小林已经看到 A/B，系统却忘了这件事

返回 feed 后还要更新 served history、写缓存、发送日志和训练事件。全部放在响应前会把最慢下游加入用户延迟；裸后台任务则可能在进程退出时永久丢失。当前响应成功与未来状态正确必须分开验收。

> **新手 Tip：先画 crash window**
>
> 在每次状态变化之间画一条缝：业务写前、业务写后、enqueue 前后、消费后、ack 前后。问进程在这里崩溃会丢什么、重复什么，可靠性设计就会清楚很多。

4.7.2 在原仓定位：框架负责并行，不替业务选择持久性

阅读：

- `candidate-pipeline/side_effect.rs`；
- `candidate-pipeline/candidate_pipeline.rs` 的 side-effect 调度；
- `home-mixer/side_effects/served_candidates_kafka_side_effect.rs`；
- `home-mixer/side_effects/update_served_history_side_effect.rs`；
- `home-mixer/side_effects/redis_post_candidate_cache_side_effect.rs`；
- `home-mixer/side_effects/phoenix_experiments_side_effect.rs`。

为每个 side effect 标注：是否影响响应、写向何处、允许丢还是允许重复、需要什么幂等键、失败如何观测。公开代码展示调用位置，不等于展示了 Kafka/Redis 的完整持久协议。

4.7.3 原理与边界：投递语义取决于确认位置

at-most-once 倾向不重复但可能丢；at-least-once 允许重复并要求消费者幂等；“exactly once”通常只在特定事务边界内成立。业务状态和事件若分开写，需要 outbox 或 reconciliation 关闭缝隙。

幂等不是“消息 ID 去重”一句话。必须定义幂等作用域、记录保留时间、并发重复、业务键和副作用是否真的可重复。补偿也可能失败，因此需要可重试、可观测和人工修复路径。

4.7.4 构造：建立 Durable Event Worker

实现本地 durable queue，核心路线可用 SQLite：请求在响应前只完成快速 enqueue，后台 worker 拉取、处理、成功后 ack。事件包含 event ID、request ID、candidate IDs、schema version 和创建时间。

本课程选择把 ServedHistoryUpdated、CandidatesServed、CacheWrite 三类事件都放入 durable worker，以便统一练习 inbox/业务幂等；这不代表所有生产缓存回填都必须 durable。进程重启后能够恢复 pending 事件；poison event 有有限重试和隔离区，而不是永久阻塞队头。

> **验收契约：响应与未来状态分开验收**
>
> enqueue 只有在持久事务提交后才算成功；在声明的进程崩溃模型和同步级别内，事件不会因 worker 崩溃丢失；消费后 ack 前崩溃会重放但不重复生效；事件 schema 有版本；积压、重试、隔离和处理延迟可观测。

4.7.5 破坏并证明：逐个持久化边界杀进程

在 enqueue 前、持久提交后响应前、领取后执行副作用前、副作用完成后幂等记录提交前、幂等记录提交后 ack 前，以及 ack 后杀死真实子进程并重新打开存储。检查丢失、重复、最终状态和恢复时间。若专门演示“先 ack 后写”，必须明确把它标成错误协议；不要用同一进程内抛异常代替 crash recovery。

再注入重复事件、永久坏事件、下游超时和 schema unknown。记录 pending、oldest age、attempts、deduplicated、DLQ/quarantine 和 reconciliation mismatch。

4.7.6 验收：未来状态有了可靠入口

- side effect 按可靠性需求分类；
- durable enqueue 与响应边界明确；
- 重复投递不重复生效；
- poison event 不阻塞全队列；
- 真实重启测试证明恢复；
- 当前响应、可靠接收和下游完成分别有指标。

5 第四部：规模、复制与在线演化

能够并发不等于能够承压，拥有副本不等于能够安全失效转移。本部从资源预算、排队和负载脱落进入分片、复制、epoch、fencing，再处理新旧协议、配置和数据结构同时存在的滚动发布世界。

5.1 间章 I：排队论不是算术题——从等待预算到过载闭环

大型 Tip · 间章：为什么系统在 CPU 未满时已经不可用

第 14 章要求你找到饱和拐点。本间章补足背后的推理：请求不是只“占用 CPU”，它还会等待连接、锁、下游、队列和调度。系统容量也不是一个固定 QPS，而是一组随请求类型、fan-out、重试和推荐质量变化的预算。

5.1.1 1. 先把“并发”拆成三种东西

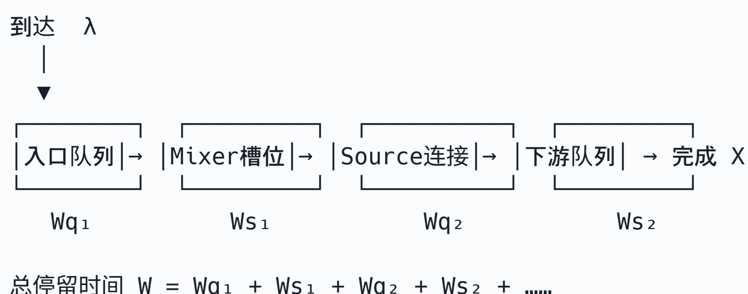
日常语言里的并发常混合了三个量：单位时间进入多少请求、同时有多少请求在系统里、单位时间完成多少请求。分别记作到达率 λ、在途数 L 和吞吐率 X。只有系统长期稳定且没有持续积压时，X 才近似 λ。

Little’s Law 写作：

$$L = \lambda W$$

其中 W 是请求在所观察系统中的平均停留时间。这个等式很强，因为它几乎不要求具体到达分布；但它也很容易被误用。应用它至少要声明观察边界、统计窗口与稳定性。

图：同一个请求穿过多个等待室



若你只测“进入 handler 后”的 W，入口代理中的等待就消失了；若统计窗口正处于队列增长期，用完成吞吐代替到达率也会低估在途量。Little’s Law 是账本恒等式，不是容量预测器。

小心：三个常见误用

第一，把 P99 延迟代入平均值公式；第二，在持续过载时假定到达率等于吞吐率；第三，只统计执行中的任务，不统计队列、重试和已取消但尚未停止的任务。

5.1.2 2. 延迟为什么会在极限前陡增

设单个 worker 平均每秒处理 μ 个请求，有 c 个 worker，粗略服务能力是 $c\mu$。利用率可写成：

$$\rho = \frac{\lambda}{c\mu}$$

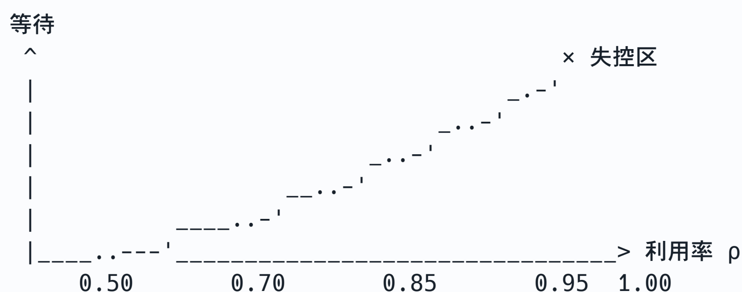
这只是第一层估计。请求耗时有波动时，即使 $\rho < 1$，一串慢请求也会形成队列；当 ρ 接近 1，系统没有空闲容量吸收波动，等待迅速放大。生产流量还通常具有突发、昼夜变化和相关性，并不服从教材里最方便的泊松分布。

在最简单的 M/M/1 模型中，平均系统时间为：

$$W = \frac{1}{\mu - \lambda}$$

它展示了“分母趋近于零”的危险，却不能直接拿来预测你的服务。M/M/1 假设单服务台、泊松到达、指数服务时间、无限队列与稳定状态；真实 Mixer 是多阶段、有限池、重尾延迟并带 deadline 的开放网络。

图：利用率不是平滑的旋钮



“CPU 还有 15%”不能证明还有 15% 可用容量。

应把模型当作提问工具：波动来自哪里？哪个资源最先饱和？请求类型是否同质？排队是否有界？被拒请求是否已消耗昂贵资源？随后用负载实验校准，而不是让公式替代实验。

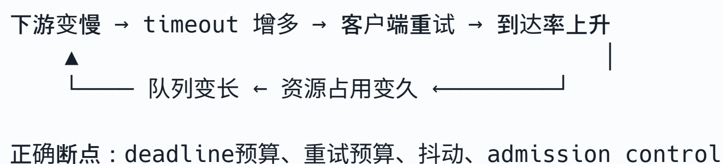
5.1.3 3. Fan-out 会把一个请求变成一束工作

一次 For You 请求同时访问 Following、Phoenix、Thunder 与缓存。若每次平均触发 f 路调用、每路平均重试 r 次，则下游尝试率近似：

$$\lambda_{\text{down}} = \lambda_{\text{request}} f(1 + r)$$

这里假设每路调用概率相同，r 是“每个初始调用的平均额外尝试数”，不是重试上限。更一般地，对 source i 的调用概率 q_i 和额外尝试 r_i，有 $\lambda_{\text{down}} = \lambda_{\text{request}} \sum_i q_{i(1+r_i)}$。若失败集中发生，$r_i$ 会随下游变慢而上升，于是形成正反馈。

图：重试放大环



正确断点：deadline预算、重试预算、抖动、admission control

并行 fan-out 降低理想延迟，却提高瞬时资源需求。请求结束后仍运行的“僵尸任务”尤其危险：用户已经收到降级 feed，后台 source 仍占连接和并发槽位。

把 A—F 放回现场：A 来自关注 source，B 来自召回，C 来自旧缓存，D/E 必须被安全过滤，F 是广告。过载时可以关掉昂贵的 B、缩小候选池、暂缓 F，也可以接受 C 的有界陈旧；但不能跳过 D/E 的安全判断。这说明负载脱落不仅有技术优先级，还有产品不变量。

5.1.4 4. 队列不是容量，而是等待承诺

队列长度 Q 与消费吞吐 X 可以给出最粗略的排空时间：

$$T_{\text{drain}} \approx \frac{Q}{X - \lambda}$$

前提是 $X > \lambda$ 且二者在排空期间近似稳定；X 指 backlog 存在时可持续的完成能力，不是任意瞬时吞吐。若 worker 受请求 mix、queue age 或限流影响，应按 class 分开估算。若 $X \leq \lambda$，队列不会排空。

无界队列看似“零拒绝”，实际把拒绝推迟成 timeout、内存耗尽或用户离开。有限队列也不是越长越好：当最大允许端到端 deadline 为 200 ms，平均排队已经 300 ms 时，接纳新请求只是制造必然失败的工作。

图：三种过载行为

```
无界队列：  [...................................] → OOM/超时
大有限队列：[.....................|满]            → 慢失败
预算队列:   [....|按deadline/优先级接纳]           → 早拒绝/降级

                  ↑ 队列上限应由等待预算推导，而非拍脑袋
```

好的 admission control 在昂贵工作前做决定，并说明拒绝原因。它可依据全局并发、调用方配额、剩余 deadline、请求成本与系统健康，而不只是入口 QPS。

5.1.5 5. Bulkhead、优先级与公平性

若 feed 请求、backfill、模型预热和事件回放共享同一个线程池，低优先级长任务会造成 head-of-line blocking。Bulkhead 的思想不是“多建几个池”这么简单，而是为故障域分配独立预算，使一个域耗尽时其他域仍能前进。

但隔离也有代价：固定切分会让一个池空闲而另一个池排队；动态借用又可能破坏保护边界。合理方案常是“保底份额 + 可撤回借用”：交互流量拥有不可侵犯的最低槽位，空闲时后台任务可以借用，一旦前台到达便停止新增借用。

优先级同样可能导致饥饿。可使用加权公平队列、每租户 token bucket 或 aging，让等待过久的低优先级任务逐步提升机会。任何调度策略都应有可观测字段：tenant、class、estimated cost、queue age、admission result。

5.1.6 6. 降级必须同时度量可用性与质量

HTTP 200 不能说明推荐请求成功。如果过载时只剩一条陈旧 C，协议可用但产品可能不可用。为此可定义复合 SLI：

- 在 deadline 内返回；
- 至少有 N 个候选；
- D/E 从未进入结果；
- 新鲜候选比例高于阈值；
- 关键 source 或替代 source 至少成功一类；
- 广告 F 不突破 safe-gap。

不要把这些条件草率相乘得到“总可用率”，因为它们可能相关。应同时报告协议成功率、质量合格率和降级模式分布。

源码证据：与原仓建立联系

在 `candidate-pipeline/candidate_pipeline.rs` 追踪阶段如何扩张候选工作量；在 `thunder/thunder_service.rs` 寻找并发服务边界；在 `grox/engine.py` 观察任务图可能怎样形成并行宽度。公开代码能证明调用结构，不能证明生产容量、线程池大小或负载策略。

5.1.7 7. 一次严谨的容量实验应该怎样读

阶梯负载不能只跑十秒。每一级至少经历预热、稳定采样和冷却；记录到达率而非只记完成率；同时观察 CPU、内存、连接、active、queue depth、queue age、P50/P95/P99、timeout、rejected、retry 和 feed quality。

用如下证据判断拐点：吞吐不再随 offered load 增长；队列或在途数持续上升；P99 突增；timeout 与 retry 同时增长；质量降级比例越过预算。停止条件应预先定义，以免为了“测极限”把本机拖死。

对比至少三轮：共享无界池、共享有界池、按 source/优先级隔离。实验输入保持一致，固定随机种子，但另外增加一次突发流量，以暴露仅在平均负载下稳定的设计。

5.1.8 8. 反例：看起来合理却会害人的修复

- “加长 timeout”：让更多工作滞留，并可能提高成功率假象。
- “队列扩大十倍”：吸收短峰值，也把过载发现推迟十倍。
- “失败就重试三次”：把一次请求变成最多四次尝试。
- “加线程”：若瓶颈是连接、锁或下游，只会增加争用。
- “关闭所有过滤”：提高吞吐，却让 D/E 进入 feed。
- “按平均耗时算容量”：忽略重尾分布对 P99 和并发的支配。
- “只限入口”：后台回放或内部重试仍可绕过预算。

5.1.9 9. P99 为什么会在 Fan-out 中变得更糟

假设每一路调用在阈值 t 内完成的概率是 p，并且各路延迟独立；若请求必须等待全部 n 路，则整体在 t 内完成的概率约为 p^n。单路有 99% 在阈值内，十路全部达标只有约 90.4%。

独立是假设，不是事实：多路调用常共享网络、线程池或同一存储，相关故障会让乘法估计过于乐观。即便请求只等“最快 K 路”，慢尾仍会占据资源，除非取消真正传播。

Hedged request 会在首个尝试过慢时向另一个副本发送第二次尝试，能降低尾延迟，也会增加负载。它只适合幂等读取，并应在超过历史分位点后触发、限制全局额外比例、首个结果返回后取消其余尝试。过载时盲目 hedge 正是在火上加油。

推荐 fan-out 可使用更贴近业务的等待策略：A 的 Following 在剩余预算内等待，B 的昂贵召回在预算紧张时跳过，C 作为有 age 标记的兜底；D/E 安全数据若无法确定，则不是普通超时降级，而是保守过滤。

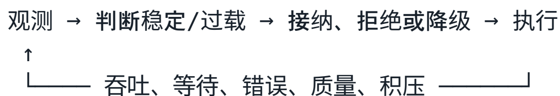
5.1.10 10. 从预算反推，而不是从默认值抄 timeout

若入口 SLO 是 200 ms，先预留网络、序列化、混排和响应发送，再把剩余预算分配给 source 与降级判断。不能让每个下游都获得完整 200 ms；串行阶段的预算近似相加，并行阶段的关键路径取最大值，但资源成本仍然求和。

预算还应包含重试：第一次尝试耗尽 180 ms 后再重试 100 ms，已经不可能满足入口 SLO。重试资格应同时检查剩余 deadline、操作幂等性、错误是否可重试和系统是否过载。

5.1.11 11. 留给实现的最小控制闭环

读者系统至少形成以下闭环：测量资源与质量；在 admission 点依据预算决策；对不同工作隔离；过载时选择有序降级；在恢复后缓慢放量，避免积压与重试再次击穿。



> **新手 Tip：先做离散规则，再谈自适应算法**
>
> 新手可以先实现三个状态：normal、degraded、shed。用有滞回的阈值切换，避免指标在边界抖动导致系统频繁开关 source。把每次状态变化写入日志，随后再考虑自动并发限制。

5.1.12 12. 自检：你是否真的理解了“承压”

1. 为什么 $L = \lambda W$ 不能直接告诉你应该开多少线程?
2. 在持续积压时，应使用 offered load 还是完成吞吐计算进入系统的工作?
3. 100 QPS、四路 fan-out、平均 0.2 次额外尝试，会制造多少次每秒下游尝试?
4. 为什么有限队列仍可能太长？怎样从 deadline 推导上限?
5. A—F 中哪些能力可以降级，哪些是不变量？请说明理由。
6. Bulkhead 固定切分和动态借用各有什么风险?
7. 如何证明被取消的请求没有留下僵尸工作?
8. 哪些指标能区分“系统健康”和“只是仍在返回 200”?

> **检查点：带着这张图回到第 14 章**
>
> 你应能为每个等待室标出容量、所有者、deadline、拒绝策略和指标；能用公式做数量级核对，也能明确公式的前提；最终用负载实验找到拐点，并证明 D/E 安全边界没有在降级中消失。

5.2 资源预算、排队与过载保护

5.2.1 现场：服务没有报错，P99 却已经失控

当入口速率接近系统服务能力，队列会先掩盖过载，再把等待推成长尾。fan-out、重试和后台任务共用连接或执行池时，一个慢依赖甚至能拖住与它无关的健康路径。分布式熟手首先问的不是“还能加多少线程”，而是工作从哪里进入、在哪里等待、谁有权拒绝。

> **Tip：记住一个数量关系**
>
> Little's Law 在稳定系统、长期平均和一致边界下写作 `L = λW`。800 completed QPS、平均 100 ms 意味着约 80 个在途请求；若延迟升到 1 秒且系统仍能稳定完成同样吞吐，才约为 800 个。过载发散时不能机械套用。

5.2.2 在原仓定位：查找并发边界与负载脱落

回查 `thunder/thunder_service.rs`、`candidate-pipeline/candidate_pipeline.rs` 和 `grox/engine.py`，搜索 semaphore、spawn、buffer、queue、limit、timeout 和 rejection。建立资源图：

```
入口 -> request slots -> source client pools -> task pools -> queue -> storage
```

对每个节点写容量、等待位置、超时所有者和拒绝行为。公开快照只展示部分应用调用点，不代表 X 的完整生产限流配置。

5.2.3 原理与边界：并发、队列和吞吐不能独立调参

无界队列把显式拒绝变成隐式超时和内存风险；过大的并发会压垮下游；过小则浪费服务能力。利用率接近极限时，微小抖动也会造成排队陡增。重试和 shadow traffic 还会放大实际到达率。

保护手段包括 admission control、有界等待室、bulkhead、优先级、公平调度、circuit breaker 和 load shedding。Admission control 控制是否接纳工作；circuit breaker 避免持续调用已失败依赖；二者不能互相替代。降级必须保护最低推荐质量，不能只追求返回 200。

5.2.4 构造：建立负载发生器与资源控制面

为读者系统编写标准库负载发生器，按阶梯提高 QPS。给 Mixer、Retrieval、Thunder 和 Event Worker 分别配置并发上限与有界队列，禁止所有工作共享一个全局池。

定义两类请求：交互式 feed 与低优先级 backfill。过载时优先保护 feed，并允许减少召回 source 或候选数量，但 D/E 安全约束永远不能降级。

> **验收契约：过载必须有界且可解释**
>
> 队列和并发都有硬上限；拒绝尽量发生在昂贵 fan-out、长时间排队和主要下游容量消耗之前；低优先级流量不能挤占交互流量保底，若后台工作承担活性承诺也应保留最低份额；降级仍返回至少 N 个合格候选或明确失败；重试不会绕过 admission control。

5.2.5 破坏并证明：找到饱和拐点

从低负载逐级增加 QPS，记录吞吐、active、queue depth、queue age、P50/P95/P99、rejected、timeout 和 feed size。随后让 Retrieval 变慢、Event Worker 积压、backfill 抢占资源，比较无界队列、共享池和 bulkhead 三种设计。

禁止只报告平均延迟。要求画出负载—吞吐—P99 曲线，指出系统从稳定进入排队失控的拐点，并用 Little’s Law 核对在途数。

5.2.6 验收：系统知道什么时候说不

- 资源图覆盖请求、连接、任务和队列；
- QPS、fan-out、重试与在途数能够闭合估算；
- 有界队列和 admission control 生效；
- bulkhead 保护交互流量；
- 降级保持安全与最低 feed 质量；
- 负载实验找到饱和点而非只跑一次 benchmark。

5.3 间章 J：分片不是取模——路由、热点与在线迁移

> **大型 Tip · 间章：把数据放到多台机器只是开始**
>
> 第 15 章让你实现四个逻辑 shard。本间章解释其真正难点：分片键决定可完成的查询，路由只决定“去哪里找”，迁移协议才决定节点变化时是否丢写、读旧或永久留下双份真相。

5.3.1 1. 分片首先是数据所有权契约

分片函数通常写作 $\text{shard} = h(\text{key})$，但这行代码隐藏了四个问题：key 是什么；谁保存当前路由版本；一次请求要访问几个 shard；路由变化期间谁拥有写权。

按 user ID 分片，读取小林的 Served History 很自然；按 post ID 分片，读取帖子特征自然；“找所有用户最近看过 A 的次数”则会 scatter-gather。分片不是免费的横向扩展，而是把某些本地查询换成跨分片查询。

图：分片键塑造请求形状

```
按 user_id： 小林 ———→ shard-b → {A,B,C served history}

按 post_id： A ————→ shard-a → {all viewers of A}
             B ————→ shard-d
             C ————→ shard-c
```

查询“小林看过什么”：前者单分片，后者 scatter-gather。

选择 key 时要列出主读路径、主写路径、事务边界、热点来源和未来再分片成本。不要以“分布均匀”作为唯一目标；一个均匀但每次请求访问全部 shard 的方案可能更差。

5.3.2 2. 取模为何让扩容变成全量搬家

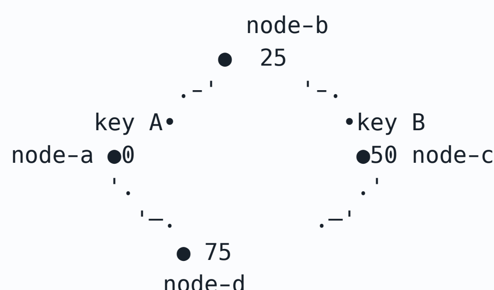
最直接的路由是：

$$\text{shard} = \text{hash}(\text{key}) \bmod N$$

当 N 从 4 变 5，多数 key 的余数都会改变，理论上只有约 $\frac{1}{5}$ 恰好保持原位。缓存场景可能接受大规模失效，持久状态则会造成巨量复制、网络和校验工作。

一致性哈希把节点和 key 放在环上，key 顺时针归属第一个节点；Rendezvous hashing 则为每个 key 对所有节点打分，选择最高者。二者在成员变化时通常只移动与变更节点相关的一部分 key。

图：环上的稳定路由



key 顺时针找到拥有者；移除 node-c 只重映射其区间。

“只移动 $\frac{1}{N}$”仍是近似说法，依赖哈希质量、虚拟节点/权重和 key 数量。测试应比较 movement ratio、负载方差和权重变化，而不只检查某个 key 的结果。

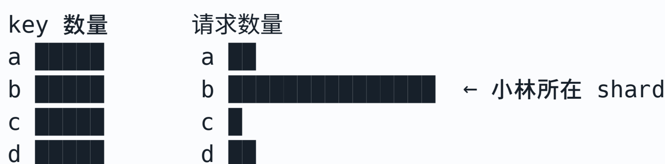
5.3.3 3. Key 均匀不等于请求均匀

如果每个 shard 都有 25% 的 key，但小林占 30% 请求，那么承载小林的 shard 仍会过载。分别定义：

- storage skew：字节或 key 数量分布；
- traffic skew：请求或计算量分布；
- fan-out skew：一个查询触达 shard 数分布；
- growth skew：新数据增长速度分布。

常见热点策略各有代价。热 key 复制提高读能力，却使写入和失效更难；请求合并减少同一时刻重复计算，却不能降低持续独立请求；拆 key 能摊开写入，却增加读聚合；局部缓存降低下游流量，却引入陈旧与击穿。

图：平均值怎样藏住热点



平均 key/shard 完美；请求负载完全不均。

把 A—F 映射到数据模型：A 的实时关注关系可能按 viewer 分片；B 的向量索引可能按 corpus/embedding 空间拆分；C 的缓存与 Served History 常按 viewer 路由；D 的屏蔽关系必须在正确用户分片可达；E 的安全字段可能按 post 存储；F 的广告预算可能按 advertiser 或 campaign 分片。一次 feed 因而天然跨越多个所有权域。

5.3.4 4. 路由元数据本身也是分布式状态

客户端必须知道 shard 列表、权重、状态和版本。若一半 Mixer 使用 routing version 12，另一半使用 13，同一个 key 可能被送往不同节点。于是每次请求或 trace 至少记录 routing version、computed shard 和 fallback route。

路由发布可以由中心服务、配置文件或服务发现完成；无论哪种方式，都要处理部分发布、旧客户端、回滚和缓存。控制面可以用 CAS/事务原子推进一份权威 routing version，但这不等于全体客户端瞬时切换：旧客户端仍会在传播窗口内发出旧路由请求，数据面必须用 epoch 拒绝、重定向或受控转发。不要让“查不到就广播所有 shard”成为永久兜底，它会在异常时把单 key 流量放大为全局 fan-out。

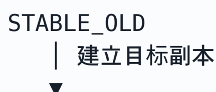
5.3.5 5. 在线迁移是一台状态机

考虑把 key 从 old shard 搬到 new shard。简单流程“复制完，改路由，删旧值”有一个致命窗口：复制期间仍写入 old 的更新不会自动出现在 new。

一种可解释的迁移状态机如下：

图：单 key 迁移的所有权变化

STABLE_OLD



```
COPYING —— copy snapshot / ordered change-log catch-up
    │ 接近追平；可选启用双写
    ▼
DUAL_WRITE —— old 仍是真源；new 失败写入 repair log
    │ change-log watermark 追平
    ▼
DUAL_READ —— old 为真源，比较 new
    │ new 已追平
    ▼
CUTOVER —— 路由版本+epoch 原子推进
    │ 观察窗口
    ▼
STABLE_NEW —— 延迟清理 old

任一步失败：重试同一步，或按状态定义回滚；绝不猜测。
```

状态机必须持久化并具备幂等 step。它不要求所有动作原子，但要求每个崩溃点都能根据记录判断“已经做了什么、下一步是什么”。迁移 ID、key range、source、destination、routing epoch、change-log watermark 和 checksum 都应可追踪。若系统不采用 dual write，可以一直以 old 为真源并追赶有序变更日志；不能只复制一次 snapshot 就宣布完成。

5.3.6 6. Dual read 与 dual write 的真实语义

Dual read 可用于验证新副本，但“值不同”不一定代表错误：复制存在 lag，TTL 可能不同，派生数据可能允许重新计算。比较器应按数据契约判断版本、业务字段和可接受陈旧度。

Dual write 看似解决复制期间更新，却产生部分成功：old 成功/new 失败，或相反。若应用无法做跨存储事务，就要指定真源、重试日志、幂等写和 reconciliation。不要把 `try both` 当成一致性协议。

迁移期间常见读策略：

- old-first：切换前稳定，但无法验证 new 独立可用；
- new-first with fallback：能暴露 new 问题，但回退会隐藏缺失；
- read-both-compare：证据最强，成本也近两倍；
- version-select：读取带逻辑版本、LSN 或 epoch 的较新值，要求版本形成全序或能显式发现冲突；不能用 wall-clock timestamp 冒充可比较版本。

> **小心：双写不能凭成功次数投票**
>
> 两边都返回成功也不证明后续读取一致；一边失败也不能随意选择另一边为新真源。所有权必须由迁移状态和 epoch 决定，结果由后台校验收敛。

5.3.7 7. Cutover 需要版本或 fencing

若旧客户端在 cutover 后仍向 old 写入，清理前可能看似正常，清理后更新永久丢失。解决方案不是等待“所有客户端应该都更新了”，而是让存储识别 routing epoch：new owner 接受当前 epoch，old owner 在 cutover 后优先 reject+redirect。若确需转发，必须验证 epoch、携带幂等 ID，并携带 hop limit 或 visited owner 防止 old/new 之间形成转发环。

这与下一间章的 fencing 同源：控制面宣布所有权变化，数据面必须执行该变化。没有数据面拒绝，旧路由就是 stale writer。

5.3.8 8. 迁移期间如何保护推荐语义

Served History 迁移错误可能让 A/B 再次曝光；屏蔽关系迁移错误可能让 D 穿透；安全字段 E 缺失时必须保守处理；广告预算 F 的双写错误可能造成超投。并非所有不一致代价相同，因此迁移顺序和 fallback 要按数据敏感度设计。

例如 C 只是可重建缓存，可以 miss 后重算；D 的屏蔽关系不能在 miss 时 fail-open；F 的预算若要求强约束，可能宁可拒绝投放也不接受旧副本。迁移协议必须携带业务风险，而不只是 bytes copied。

> **源码证据：原仓证据的边界**
>
> 从 `home-mixer/query_hydrators/served_history_query_hydrator.rs` 与 `home-mixer/side_effects/update_served_history_side_effect.rs` 提取读写契约；从 `home-mixer/server.rs` 识别 cluster/datacenter 坐标。公开仓库没有给出底层 Redis、Kafka 或存储迁移协议，所以本间章训练的是应用层推理，不能反向宣称 X 采用了某种哈希或 cutover 实现。

5.3.9 9. 拆分 Hot Key 也会改变一致性边界

若小林的 Served History 单 key 太大，可按 `(user_id, bucket)` 拆成多个 bucket；bucket 可由时间窗、post hash 或序号决定。写入被摊开，读取却要合并多个 bucket，并重新处理去重、排序、分页与过期。

按时间窗拆分便于 TTL 和冷热分层，但跨窗分页需要稳定 cursor；按 post hash 分布均匀，却让“最近 N 条”访问全部 bucket；按序号滚动写入容易定位最新 bucket，却需要安全推进当前 generation。不存在只获收益的拆分方式。

图：大 key 拆分后的读放大

```
小林 Served History
   ├─ bucket-0 {A,C,...}
   ├─ bucket-1 {B,...}
   └─ bucket-2 {...}

写：1 个 bucket              读最近记录：3 路读取 + merge
```

为 bucket 元数据设置 owner、generation 和版本。若拆分迁移与 shard 迁移同时发生，状态空间会相乘；工程上应一次只改变一个所有权维度，或明确组合状态机。

5.3.10 10. Scatter-gather 的尾延迟与部分结果

查询触达 n 个 shard 时，总延迟通常由最慢必要 shard 决定。一个 shard timeout 后，是返回部分结果、重试副本还是让整个请求失败，取决于不变量。召回 B 可以部分成功；查 D 的屏蔽关系却不能把“一个 shard 未响应”解释为“没有屏蔽”。

scatter-gather 还会放大单请求成本。入口并发 100、每请求 16 shard，就可能产生 1600 个下游在途调用。分片增加容量的同时也能增加 fan-out，必须和间章 I 的 admission、deadline 与 bulkhead 一起设计。

5.3.11 11. 反例：迁移脚本为何常在最后一步出事

- “先复制再改配置”：忽略复制期间的新写。
- “双写两边就安全”：忽略部分成功与旧客户端。
- “hash 分布均匀”：忽略 hot user 与请求成本。

- “失败后重跑整个脚本”：可能重复 cutover 或提前清理。
- “校验 key 数相等”：无法发现值错、版本旧或业务不变量破坏。
- “切换后立即删旧”：失去回滚窗口与取证材料。
- “fallback 到任意副本”：可能把陈旧值重新写回新世界。

5.3.12 12. 应该留下哪些证据

路由层记录 key、routing version、owner 和 fallback；迁移层记录每状态 key 数、bytes、lag、checksum mismatch、retry 与 stalled duration；容量层同时看 keys/bytes/requests/CPU by shard 和 top hot keys。

测试至少包含：加节点与减节点的 movement property；权重变化；路由版本混跑；copy 中崩溃；dual write 单边失败；cutover 后旧客户端写；cleanup 前回滚；热点超过单 shard 能力。

> **新手 Tip：先迁移十个固定 key**
>
> 不必一开始做百万数据。给 A—F 和四个额外 key 固定版本号，逐步打印 old/new 值、owner、epoch 与迁移状态。只有你能在纸上解释每一步，才扩大随机测试。

5.3.13 13. 自检：你是否真的拥有这些数据

1. 为什么 `hash(key) mod N` 在扩容时移动大量 key?
2. 一致性哈希减少 movement，为什么仍不能替代迁移协议?
3. key 分布均匀时，哪三类 skew 仍可能存在?
4. dual write 一边成功、一边失败时，谁是真源？依据是什么?
5. cutover 后怎样阻止持有旧路由的 Mixer 写回 old shard?
6. C、D、E、F 的迁移 fallback 为什么不应相同?
7. migration progress 应持久化哪些字段才能崩溃恢复?
8. 你会用哪些性质测试验证路由与 movement?

> **检查点：带着所有权状态回到第 15 章**
>
> 你的实现不只会算 shard：它能区分 key 与流量分布，能持久化迁移状态，能在每个崩溃点恢复，能阻止旧路由写入，并能按 A—F 的业务风险选择读写与回退策略。

5.4 分片、热点与在线迁移

5.4.1 现场：节点增加了，为什么某个用户仍然很慢

分片解决总体容量，不自动解决 hot key。200 个 key 可以均匀分布，小林一个 key 却可能占据单 shard 的大部分请求。更换分片函数或移除节点还会造成数据迁移、双读、双写和短暂不一致。

> **新手 Tip：先选 partition key**
>
> 分片算法之前，先问访问模式。按 user ID 分片适合按用户读取；按 post ID 分片适合帖子查询；需要跨所有用户聚合时则会产生 scatter–gather。没有一种 key 同时优化所有路径。

5.4.2 在原仓定位：只从坐标和调用点推断

在 `home-mixer/server.rs`、`home-mixer/query_hydrators/served_history_query_hydrator.rs`、`home-mixer/side_effects/update_served_history_side_effect.rs` 以及 source/client 调用中搜索 shard、datacenter、cluster、Kafka 和 Redis 相关坐标。

为候选缓存、served history、Thunder 查询和事件流分别写：partition key、请求形态、是否 scatter–gather、热点风险和跨机房依赖。仓库没有公开 Redis/Kafka 内部复制协议，因此这些调用点只能证明应用依赖，不能证明基础设施保证。

5.4.3 原理与边界：稳定路由只是迁移的第一步

Rendezvous hashing 或 consistent hashing 可以减少节点变化时的 key movement，但数据仍要复制到新归属，并处理迁移期间的并发读写、删除和旧 writer。常见方案包括 ownership epoch、后台 copy、catch–up watermark、dual read/write、校验和 cutover，每一步都有失败窗口。

热点需要单独策略：请求合并、局部缓存、复制热 key、拆大 key、限流或改变数据模型。平均 shard load 会掩盖热点，应同时观察 key distribution 与 request distribution。

5.4.4 构造：实现路由、热点检测和迁移状态机

为 Served History 实现稳定路由，使用至少四个逻辑 shard。生成普通用户和一个 hot user，记录每 shard 的 key 数和请求数。然后把 shard–d 标记为 draining，执行：复制、双读校验、切写、停止旧读、清理。

迁移进度、ownership epoch 和校验水位必须持久化，重启后可继续；每个 key 的复制、业务删除 tombstone 传播和 cutover 状态可重复执行。业务 tombstone 表示对象删除，必须随 snapshot/log 迁移；物理 cleanup 只是回滚窗关闭后的旧副本清理，二者不能混写。不要在一本教材里实现生产数据库，但要把应用层迁移协议写成明确状态机。

> **验收契约：迁移不能靠一次性脚本祈祷**
>
> 节点变化的 movement ratio 接近路由算法在该拓扑下的预期，而不是要求绝对零冗余；迁移步骤幂等；cutover 前新旧值按版本和水位校验；失败可恢复或回滚；hot key 能被单独识别；读写策略在每个阶段明确。

5.4.5 破坏并证明：在迁移每一步崩溃

在 copy 中、双写中、cutover 后清理前杀进程并恢复。注入旧 shard 写成功、新 shard 写失败，以及双读发现不一致。检查是否丢更新、重复迁移或提前删除旧数据。

指标包含 keys/requests by shard、hot-key ratio、movement ratio、migration lag、dual-read mismatch、copy retry 和 cutover duration。不要用一个总 availability 指标替代这些证据。

5.4.6 验收：数据分布开始对变化负责

- partition key 与访问模式对应；
- key 分布和请求分布分别观测；
- 稳定路由具备 movement 测试；
- 迁移是可恢复状态机；
- dual read/write 的窗口和代价明确；
- hot key 不再被平均负载隐藏。

5.5 间章 K：副本不是备份——复制、共识与 Fencing 的责任边界

大型 Tip · 间章：两个节点都活着，数据仍可能被写坏

第 16 章通过 epoch 7 与 epoch 8 展示 stale writer。本间章把这件事展开：复制回答“状态怎样传播”，共识回答“参与者怎样对顺序或领导权达成一致”，fencing 回答“真实资源怎样拒绝旧世界”。三者相互关联，却不能互相替代。

5.5.1 1. 先区分四个目标

副本常被笼统描述为“高可用”，但工程上至少有四个目标：增加读吞吐；降低单机丢失风险；缩短故障恢复时间；在部分故障下继续服务。不同复制方式只能满足其中一部分。

备份则是另一个时间尺度：副本会迅速复制误删除和逻辑损坏，离线 snapshot 才可能保留故障前状态。因此“有三个副本”不能推出“能恢复昨天的数据”。

图：复制链条中的三个问题

控制面决定角色；数据面执行写入；恢复面保留历史。

为每类状态写清 source of truth、可重建性、允许 lag、确认条件、RPO 与 RTO。A 的实时帖子可从事件重放，C 的缓存可重建，D 的屏蔽关系和 F 的预算则可能要求更严格的持久与读取语义。

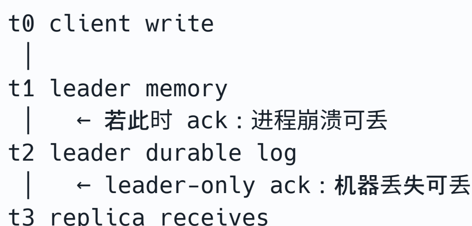
5.5.2 2. 同步与异步复制是在选择确认点

异步复制通常在系统定义的 leader 本地确认点后就向客户端成功，副本稍后追赶。该确认点可能只是内存，也可能是本地 durable WAL；它降低写延迟，却留下未复制窗口：leader 在复制前永久故障，已确认写可能丢失。

同步复制等待一个或多个副本确认后再返回。它通常收窄数据损失窗口，但不自动得到 RPO=0，还依赖相关故障、介质、日志截断和协议安全假设。若要求所有副本确认，一个慢副本就阻塞系统；若只等 quorum，则未确认副本可稍后追赶。

“同步”仍需问确认了什么：写入内存、操作系统页缓存、本地 WAL fsync，还是远端持久介质？不同确认点对应不同故障模型。

图：成功响应落在哪个时间点



```
 │
t4 replica durable
 │   ← quorum durable ack：延迟更高，RPO 更小
t5 all replicas durable

“写成功”必须绑定一个明确的 t。
```

5.5.3 3. Lag 不只是一个秒数

复制延迟可以用时间、日志位置、字节或未应用条目数表示。单一“lag = 2 s”可能误导：若期间没有写，时间 lag 高但数据一致；若有巨大突发，时间短也可能积压百万条。

读副本策略要绑定用户承诺：eventual read 允许旧值；read-your-writes 要让同一会话至少看到自己的更新；monotonic read 不允许用户先看到新值又回到旧值。实现方法包括 sticky session、携带最低版本、等待副本追平或回退 leader。

Served History 若读到旧副本，会再次展示 A/B；缓存 C 读旧可能只影响新鲜度；D 的屏蔽关系若读旧会突破安全；因此“允许 5 秒 lag”不能对所有数据统一套用。

5.5.4 4. Quorum 公式隐藏了哪些前提

常见直觉是有 N 个副本，写集合大小 W，读集合大小 R，若：

$$R + W > N$$

则读写集合有交集；若 $2W > N$，两个写集合也有交集。这只是集合交集，不自动提供线性一致性。

要从交集得到正确读取，还需要：版本可比较；节点不会悄悄接受冲突写；读方选择最新合法版本；成员集合本身一致；失败节点恢复时正确 repair；并发写有明确冲突规则。Dynamo 风格 quorum、Raft majority 与数据库的“quorum consistency”不是同一个完整协议。

> **小心：公式没有写出的世界**
>
> 网络会延迟和重复；节点可能重启并忘记内存状态；成员配置会变化；旧 leader 仍可能访问外部资源；客户端会 timeout 后重试。只验证 $R + W > N$，没有验证这些状态转换。

5.5.5 5. CAP 应该怎样用，怎样不用

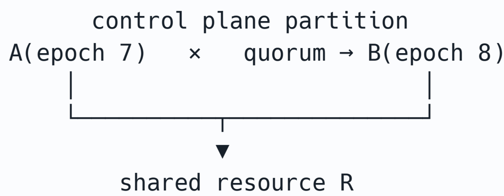
网络分区发生时，跨分区节点无法及时通信。系统不能同时保证 linearizability，以及 availability 所要求的“每个到达非故障节点的请求都在有限时间返回符合接口定义的响应”。CAP 的价值是迫使你明确分区时的行为，不是给数据库贴永久的 CP/AP 标签，也不要把业务成功/错误码混进定义。

同一系统可以按操作选择：feed 读取允许 C 的陈旧降级；D 的屏蔽更新在无法确认时宁可拒绝；指标写入可暂存；F 的预算扣减可能需要保守失败。还要区分 timeout 与已证明的分区：调用方通常只知道“在 deadline 内没有响应”，不知道对方究竟没执行、执行中还是已成功。

5.5.6 6. 选主为何不能自动停止或隔离旧 Leader

设 A 是 leader，与协调者失联；多数节点选 B。对多数派而言领导权已经转移，但 A 可能只是与协调者隔离，仍能访问对象存储、数据库或广告预算服务。A 不会因为另一个房间举行了选举就自动停止。

图：脑裂不是两个角色字段，而是两个有效写者




```
若 R 只看“请求来自 leader”，A/B 都自称 leader。
若 R 保存 max_epoch=8，A 的 7 被拒绝。
```

因此需要单调 fencing token。协调机制分配 epoch 8；B 写资源时携带 8；资源把最大已见 epoch 持久化，并拒绝任何小于 8 的写。关键不是 token 名字，而是资源自身执行比较。

5.5.7 7. Epoch、版本与幂等键不能混用

epoch 表示所有权时代，用于拒绝旧写者；record version 表示同一记录的更新次序，用于乐观并发或新旧比较；idempotency key 表示两个尝试属于同一业务操作，用于去重。

一次写可以同时携带：`epoch=8`、`record_version=42`、`operation_id=abc`。资源必须在同一串行化/事务边界中验证 epoch、检查或登记 operation ID、执行 record-version CAS 与状态变更；重复 operation ID 返回第一次的确定结果，不重复副作用。不能把它们写成彼此分离、可被暂停穿插的普通步骤。只用 timestamp 代替三者，会受到时钟偏移、相同时间戳和因果倒置影响。

Fencing 的原子性同样重要：token 检查、必要时提升 `max_epoch`、业务写入必须属于资源的同一原子边界。否则 epoch 7 可能先通过检查后暂停，epoch 8 完成写入，epoch 7 再落盘覆盖新世界。对 `epoch == current` 的请求仍需 record version 和幂等键保护。

5.5.8 8. Lease 不是免费的领导权

Lease 允许持有者在有效期内独占资源，常用于减少每次操作都走共识。但它依赖时间边界：谁的时钟？最大偏移是多少？GC pause 或进程冻结会怎样？续租请求 timeout 时，旧 lease 是否仍有效？

安全实现通常使用单调时钟计算本地期限、保守扣除不确定性，并仍以 fencing token 保护外部资源。Lease 可以帮助判断“我大概仍是 leader”，fencing 才让资源确认“这个写者不旧”。

5.5.9 9. 共识到底替应用完成了什么

共识协议在明确故障假设下，让多个节点对日志顺序、值或领导 epoch 达成一致；在 quorum 可用时，它可以推进已提交日志并阻止少数派提交冲突日志，但不自动隔离少数派访问外部资源。它也不自动完成业务幂等、跨系统事务、外部副作用 fencing、schema 演化、数据修复或调用方 deadline。

本书不要求实现 Raft。更有价值的练习是消费一个“能分配单调 epoch 的协调服务”后，证明你的应用仍正确处理 timeout、重复请求、旧 leader 恢复和资源拒绝。

源码证据：从原仓能证明到哪里

`thunder/kafka/tweet_events_listener_v2.rs` 可用于观察事件消费对顺序和恢复的需求；`home-mixer/server.rs` 展示 cluster/datacenter 等部署坐标；缓存及 side effect 调用点展示外部状态依赖。公开快照不包含 Kafka、Redis 或服务发现的共识实现，不能据此推断其副本数、确认级别或 failover 协议。

5.5.10 10. 成员变更为何不是改一行节点列表

若集群从 `{A,B,C}` 直接切到 `{C,D,E}`，旧多数派 `{A,B}` 与新多数派 `{D,E}` 可以各自形成互不相交的“多数”，并推进不同日志。安全成员变更必须由协议定义并提交：常见做法是联合阶段，使提交同时得到旧/新配置足够确认；某些协议也允许一次替换一个成员，但不能靠人工改列表。

图：直接换成员可能产生两个多数派

```
旧配置 {A,B,C}：A+B = majority
新配置 {C,D,E}：D+E = majority
                    ↑
               两组没有交集
```

联合阶段要求跨旧/新配置确认，直到新配置被正式提交。

应用不必实现联合共识，但扩缩副本时必须尊重底层协议的成员状态，不能同时移除过半节点，也不能把“进程已启动”当作“副本已追平且具有投票资格”。

5.5.11 11. 日志提交与状态机应用不是同一位置

共识日志条目可以已经 committed，却尚未被某个副本 applied 到业务状态机。读该副本时若不检查 applied index，仍会看到旧 Served History。监控因而至少区分 match/commit/applied position。

Snapshot 压缩日志后，新副本需要安装 snapshot 再追增量；snapshot 自身也必须带最后包含的 index/term。错误地混合 snapshot 与旧日志，会重复或跳过状态转换。

线性一致读取也不是“随便读任一副本”。常见方法包括读 leader 并确认其 leadership 仍有效、使用 read index，或让 follower 等到某个安全位置。每种方法都有额外通信和可用性代价。

5.5.12 12. Failover 与 failback 是两个不同迁移

Failover 从 A 转到 B，关注检测、提升、客户端路由和未复制写；failback 从 B 回到 A，必须先让 A 追平、确认 epoch、切换流量，并防止 B 成为新的旧写者。自动 failback 可能在节点抖动时反复切换，扩大事故。

每次转换都应有状态机与停止条件：发现故障；冻结或 fence 旧 owner；确认候选副本完整度；分配新 epoch；切流；验证；决定是否恢复冗余。故障检测只能基于怀疑，不能证明节点死亡，因此 fencing 是必需的最后一道门。

5.5.13 13. A—F 如何帮助选择一致性

- A：实时关注内容允许短暂漏召回，但事件重放后应收敛；
- B：召回索引可版本化切换，短暂旧 corpus 影响相关性而非安全；
- C：缓存可陈旧、可丢失、可重建，但要标记 age；
- D：屏蔽关系通常要求保守读取，旧副本可能造成不可接受曝光；
- E：安全字段未知时应 fail-closed 或进入人工/异步补全路径；
- F：广告预算与频控涉及跨请求状态，重复扣减和 stale writer 都有金钱后果。

一致性不是整个产品一个开关，而是每个不变量在具体故障下的选择。

5.5.14 14. 反例：常见“高可用”幻觉

- “两个节点互为主备”：没有说明谁决定、如何 fence、数据是否追平。
- “心跳没了就提升”：心跳失败只能证明无法通信，不能证明旧节点停止。
- “多数派写就线性一致”：忽略版本、成员变更和读取算法。

- “用当前时间做 token”：时钟不单调且可能重复。
- “副本延迟只有一秒”：没有说明用什么位置、在什么写入率下测量。
- “切回原主更稳”：未追平的原主正是数据风险。
- “共识保证 exactly-once”：共识日志不能替外部副作用做业务去重。

5.5.15 15. 你应怎样制造证据

实验先故意不做 fencing：A(epoch 7) 被隔离，B(epoch 8) 写入新 Served History，A 恢复后覆盖它。保存损坏 trace。再让资源持久保存 max epoch，重放同一序列，断言 A 被 `stale_epoch` 拒绝。

继续注入复制 lag、leader ack 后崩溃、协调者 timeout、旧 leader 长暂停、failback 时新写入。记录 current epoch、leader、commit/applied position、unreplicated writes、stale write rejected、failover duration 与 read source。

> **新手 Tip：不要用 sleep 证明一致性**
>
> 测试应通过可控消息队列、逻辑步骤或 barrier 决定何时复制、何时隔离，而不是“睡 100 ms 猜它应该完成”。你要能精确停在 ack 后、replicate 前这个边界。

5.5.16 16. 自检：谁真正有权写

1. 三副本异步复制在什么确认点可能丢失已成功写？
2. $R + W > N$ 为什么只是交集条件，不是完整一致性证明？
3. timeout 后，客户端为什么不能判断写入是否发生？
4. 旧 leader 不知道自己失效时，谁负责阻止它？
5. epoch、record version 与 idempotency key 分别解决什么？
6. Lease 为什么通常仍需要 fencing？
7. D 与 C 为什么应采用不同的副本读取策略？
8. failback 比“把流量切回去”多了哪些步骤？

> **检查点：带着资源层拒绝回到第 16 章**
>
> 你应能指出成功响应的持久化位置，描述 lag 对 A—F 的影响，解释 quorum 公式的额外前提，并用真实损坏—修复对照证明 epoch 由资源执行，而不是只存在于 leader 的内存角色字段中。

5.6 复制、Failover、Epoch 与 Fencing

5.6.1 现场：旧 Leader 回来以后仍然认为自己正确

副本可以提高可用性，却引入复制延迟和双写风险。节点 A 与协调者失联后，B 被提升为新 leader；A 恢复时如果仍能写共享资源，两个“各自正确”的节点会破坏同一份状态。

> **Tip：选主不等于保护资源**
>
> 共识通常还能对 term、日志和成员关系形成一致决定；即使它已经选出新 leader，旧 leader 也可能暂时不知道自己已经失效。真正的存储、文件或下游服务仍要拒绝旧 epoch 的写入，这就是 fencing。

5.6.2 在原仓定位：识别依赖，不臆测协议

从 `home-mixer/server.rs` 的 datacenter/cluster 参数、`thunder/kafka/tweet_events_listener_v2.rs` 的事件消费、以及缓存和 Kafka side effect 调用点建立复制需求表。对每类状态写：

```
source of truth | replicas | allowed lag | failover owner | stale write risk
```

公开仓库没有实现 Redis、Kafka 或服务发现的 leader election。本章的复制状态机是读者实现的通用实验，不是原仓行为声明。

5.6.3 原理与边界：副本、Quorum 与一致性承诺

异步复制降低写延迟但允许 lag 和故障时丢失；同步复制只有在 ack quorum、故障相关性和持久介质假设成立时才提高持久性，同时会增加等待和故障耦合。Quorum 的交集直觉能帮助解释读写集合，但 `R + W > N` 脱离版本、失败和写入协议并不会自动产生共识或线性一致性。

epoch/term 的分配必须持久且线性化，并保证同一 epoch 不授予两个写者。资源接受写入时，要把 fencing token 检查与状态写入原子结合，拒绝小于当前 epoch 的请求。Lease 依赖时间和时钟假设，不能单独解决所有 stale writer。

5.6.4 构造：让两个副本经历一次失效转移

实现两个数据副本的最小 Served History：leader 接受写入并异步复制，replica 可报告 lag。两数据副本失联时不能自己安全形成多数派，因此实验额外依赖一个独立、线性化的 epoch allocator（可由三投票协调器或测试 oracle 模拟）分配 leadership epoch；存储层在同一原子边界验证/提升最大 epoch 并执行写入。

流程：A(epoch 7) 写入；隔离 A；B 获得 epoch 8 并写入；恢复 A，让它尝试用 epoch 7 写。先运行无 fencing 版本观察损坏，再启用 token。

> **验收契约：过期写者永远不能覆盖新世界**
>
> leadership epoch 由独立协调器持久、单调且唯一授予；资源把 token 检查、max-epoch 更新与业务写放在同一原子边界；旧 leader 恢复后写入被拒绝；读策略明确是否允许 replica lag；failover 与 failback 都有状态和指标。

5.6.5 破坏并证明：复制延迟、脑裂和回切

注入 replica 延迟、复制中断、双 leader、协调者短暂不可达和 failback。检查 read-your-writes、monotonic read、数据丢失窗口和 stale write rejection。不要用进程角色字段代替资源层 fencing 验证。

记录 replication lag、current epoch、stale write rejected、failover duration、unreplicated writes 和 read source。写明当前设计的 RPO/RTO，而不是笼统声称“高可用”。

5.6.6 验收：副本不再只是两份数据

- source of truth 与副本角色明确；
- lag 对用户承诺的影响可解释；
- failover 与 failback 均可演练；
- epoch 单调且由资源执行 fencing；
- stale writer 测试真实失败后再修复；
- 共识边界与应用责任没有混淆。

5.7 间章 L：发布不是瞬间——协议、数据、配置与模型的在线演化

大型 Tip · 间章：系统永远处在某种混合版本中

第 17 章要求 v1/v2 混跑并回滚。本间章解释为什么兼容性不是“JSON 多一个字段没关系”：代码、持久数据、事件、缓存、配置、模型和语料各自按不同速度传播，发布只是这些版本暂时形成的一种组合。

5.7.1 1. 先放弃“切换瞬间”的想象

滚动发布中，新旧进程同时收流；长连接仍指向旧节点；队列里有数小时前的旧事件；缓存保存旧对象；回填尚未完成；配置只到一半节点；模型文件仍在下载。所谓“当前版本”实际是一个版本向量：

$$V = (\text{code}, \text{api}, \text{event}, \text{data}, \text{cache}, \text{config}, \text{model}, \text{corpus})$$

请求日志只记录 code version，无法解释同一代码为何行为不同。至少把与决策相关的版本写入 trace，并能按版本切片指标。

图：一次请求横跨多条版本时间线

“v2 请求”并不是一个单独版本，而是一组组合。

5.7.2 2. Backward 与 forward 要从读者视角定义

Backward compatibility 通常指新 reader 能读旧 writer 的输出；forward compatibility 指旧 reader 能容忍新 writer 的输出。术语容易因 API 提供方/调用方视角混乱，因此评审时直接写版本矩阵更可靠：谁写、谁读、是否成功、语义是否保守。

图：兼容性矩阵比口号清楚

	reader v1	reader v2
writer v1	基线	新读旧：必须
writer v2	旧读新：回滚关键	新世界

每格都检查：能解析？语义正确？未知安全值怎样处理？

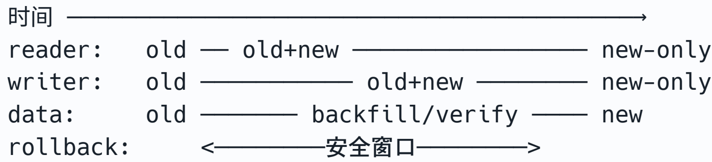
“能解析”只是最低标准。旧 reader 把未知枚举当作 `safe`，技术上没有 crash，业务上却让 E 穿透。兼容性必须包括不变量，而非只包括序列化成功。

5.7.3 3. Expand—Migrate—Contract 为何要分三步

直接 rename 或删除字段要求所有读写方同时切换，这在分布式发布中通常不存在。安全演化把动作拆开：

1. Expand：reader 同时理解旧字段与新字段，存储接受两种格式；
2. Migrate：producer 开始写新格式，双写或回填存量，持续校验；
3. Contract：确认旧 reader、延迟事件、缓存和回滚窗口消失后，停止旧写并删除旧字段。

图：字段迁移的时间关系



先部署能读新旧的 reader；再切 writer 并回填；确认旧 reader 消失且回滚窗关闭；停止旧字段写入；最后删除旧字段或数据。

步骤顺序不是形式主义。若 writer 先写新字段，旧 reader 会先失败；若 reader 刚升级就删旧字段，回滚后的旧代码无数据可读；若回填后不追踪增量写，校验通过也会再次分叉。

5.7.4 4. Schema 演化的危险不只在字段

常见变化包括新增字段、改变默认值、扩大/收缩枚举、改变单位、改变含义、拆分消息、调整唯一键。最危险的是“类型没变、语义变了”：`score` 从概率变成 logit，老代码仍能解析浮点数，却产生错误排序。

对每个字段记录语义、单位、合法范围、缺失行为、owner 和首次/最后支持版本。对于采用 required/编号字段的编码协议，新增 required field 可能让旧消息无法读取，复用已删除字段编号可能把历史数据解释成新含义；未知 enum 应保留原值、明确报错或进入保守分支，不能默认映射到安全状态。

5.7.5 5. 事件比同步 API 活得更久

API 混合版本通常持续一次滚动发布；事件可能在队列、DLQ、备份和回放中存活数月。今天删除的 consumer 兼容逻辑，可能在灾难恢复重放旧事件时重新需要。

事件 contract 应包括 event type、schema version、event ID、producer version、发生时间、幂等语义与保留期。消费者要决定未知字段、未知事件类型和 poison message 的处理；可以隔离 partition、阻断并告警，或在证明状态无依赖后进入 DLQ，不能默认跳过，也不能让一个无法解析的旧事件无声永久卡住 partition。只有 decoder/API 会暴露未知字段或保留原始 payload 时，系统才能直接统计 unknown-field 指标；若库默认静默丢弃，必须通过版本矩阵、旁路校验或自定义解码器建立证据，不能假装指标天然存在。

把 A—F 放入事件：A 的实时帖子事件可能延迟；B 的 corpus 更新与模型版本相关；ServedEvent 决定 A/B/C 去重；D 的屏蔽变更需要及时且保守传播；E 的安全状态可能从 unknown 变 safe/unsafe；F 的曝光与预算事件必须去重。每类事件的兼容失败后果不同。

5.7.6 6. 数据库演化还多了存量世界

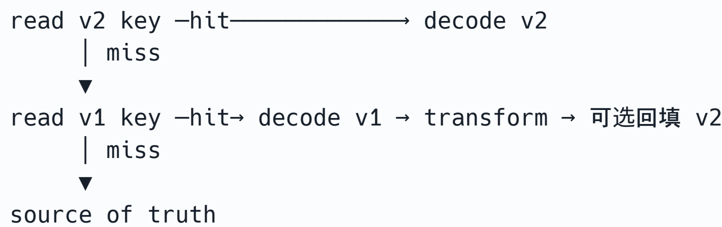
增加列通常容易，改变索引、唯一键或分片键则会影响写路径。在线回填要有进度、速率限制、幂等性和校验，且不能压垮交互流量。回填读旧快照时还要捕捉之后的增量更新。

双写迁移不能把两个 `success` 当原子提交。指定 source of truth，记录单边失败，使用 reconciliation 收敛，并让读路径在不同阶段有明确优先级。Contract 前必须证明旧格式数量归零，而不是“任务显示 100%”。

5.7.7 7. 缓存版本是协议的一部分

新代码若复用旧 cache key，可能把相同字节解释成不同结构；若直接改 key，则瞬间冷启动和回源洪峰。常见策略是版本化 key、双读、异步预热与逐步切流。

图：缓存迁移的选择



必须限制 fallback、回填并发与旧 key 生命周期。

版本 key 会暂时双倍占用容量；fallback 会隐藏新缓存写失败；预热会和真实流量竞争。发布计划要同时写容量与清理计划。

5.7.8 8. 配置是传播极快、约束极少的代码

动态配置可以瞬间改变 fan–out、排序权重、安全阈值和重试次数。它应有 schema、类型、范围、跨字段约束、默认值、owner、版本、审计、灰度、到期时间与 kill switch。

配置发布是分布式过程：节点可能拉取失败、顺序不同或在重启后回到旧 snapshot。请求应记录 config version；控制面观察各节点采用比例与 config age；服务在配置无效时保留 last–known–good，而不是静默使用零值。

> **小心：参数正确不等于组合正确**
>
> `top_k=1000` 和 `source_timeout=20ms` 各自可能合法，组合后却制造大量注定超时的工作。校验需要覆盖跨字段不变量与资源预算。

5.7.9 9. 模型、特征与语料必须形成可追踪组合

推荐系统额外拥有 model version、feature schema、normalization、label contract 和 corpus version。模型 7 若期待 feature v3，却由服务生成 v2，数值维度可能相同而语义不同；新 corpus 若尚未完全建索引，B 的召回集合会漂移。

部署应验证兼容清单与 checksum，先加载、warm up、shadow 比较，再在单节点原子切换活动引用；全 fleet 仍会经历混合版本，必须保持兼容与可观测。回滚模型时也要确认旧模型仍能读取当前特征与 corpus。不要用文件名 `latest` 作为版本证据。

5.7.10 10. Rolling、Canary、Blue–green 各解决什么

Rolling 节省资源、天然产生长时间混合版本；canary 限制初始爆炸半径，但少量流量可能覆盖不到稀有用户和 source；blue–green 让环境切换清楚，却没有消除共享数据库、缓存和事件的兼容要求。

发布策略不能替代兼容设计。即使瞬间切换所有 compute，旧数据和在途事件仍然存在；即使 canary 指标正常，新 writer 也可能已经写出回滚后无法读取的数据。

5.7.11 11. 回滚不是时间倒流

回滚代码不会撤销 v2 已写入的新事件、缓存、数据库行和外部副作用。真正的 rollback plan 要回答：旧代码能否读 v2 输出；是否停止 v2 producer；是否需要 forward fix；怎样处理已生成数据；配置和模型是否一起回退；积压事件用哪个 consumer 处理。

图：代码回滚后仍残留的新世界

```
发布 v2： [code v2] → 写 data/event/cache v2
                          |       |       |
代码回滚：[code v1] ←——————┴———————┴———————┘ 仍然存在

若 v1 不会读 v2，回滚会把局部故障升级为全面故障。
```

因此发布前就要测试“升级一半、产生新状态、立即回滚”，而不是只测试空数据库上的 v1 与 v2。

5.7.12 12. 安全枚举需要特殊规则

对于 E 的 safety status，v2 新增 `restricted_by_region`。v1 若未知，应把它当 unknown/restricted，而不是 safe。保守默认会牺牲部分召回，却保护安全不变量。

对 D 的 block reason 也类似：旧代码不理解新的封禁理由，不代表应展示内容。协议设计可以区分 `known_safe` 与 `not_known_unsafe`，避免把缺失证据误当安全证据。

F 的广告类型未知时，safe-gap 和披露规则也应保守处理。新枚举的兼容策略必须写在业务 contract，而不是交给序列化库默认。

> **源码证据：原仓中的版本链线索**
>
> 从 `home-mixer/server.rs` 提取参数快照入口；从 `home-mixer/scorers/ranking_scorer.rs` 观察参数如何影响排序；从 `home-mixer/side_effects/served_candidates_kafka_side_effect.rs` 与 `home-mixer/side_effects/phoenix_experiments_side_effect.rs` 观察事件边界。公开快照能证明消费点，不能证明线上 feature switch 默认值、灰度比例或真实发布流程。

5.7.13 13. 一张可执行的版本矩阵

测试不要只写“兼容”。至少覆盖：v1 producer/v1 consumer 基线；v1→v2；v2→v1；v2 写缓存/v1 回滚读；v1/v2 同时消费旧与新事件；config 17/18 分布不均；model 6/7 与 feature v2/v3 的合法和非法组合。

每格检查四层结果：解析是否成功；核心语义是否保持；安全 fallback 是否正确；指标是否能识别版本组合。再加入延迟事件、重复、乱序和回填中断。

5.7.14 14. 反例：版本号存在却没有演化能力

- “JSON 会忽略未知字段”：忽略语义变化、未知枚举与旧 writer。
- “DB migration 可回滚”：新数据可能已不满足旧 schema。
- “canary 只有 1% 很安全”：它仍可能写共享状态供 100% 旧节点读取。
- “改 cache key 最干净”：忽略冷启动回源与双倍容量。
- “配置改错再改回来”：旧值可能未到所有节点，新值也不会同时到达。
- “模型文件能加载”：不证明特征、归一化和 corpus 兼容。
- “删除旧 reader 代码”：忽略 DLQ、备份和灾难恢复中的历史消息。

5.7.15 15. 兼容代码何时才能删除

兼容分支不是看到新版本 100% 部署就能删。先计算旧输入的最长寿命：API 长连接、最大事件保留与 DLQ 时间、缓存 TTL、备份恢复窗口、离线作业周期、移动客户端升级周期。删除时间至少晚于这些边界，并留有审计证据。

若历史事件必须永久可重放，选择有三种：永久保留旧 reader；在归档时把事件规范化为稳定格式；提供独立 migration reader。每种方案都产生维护成本，应该在 schema 设计时决定，而不是多年后被迫保留无人理解的分支。

可以建立 compatibility debt 清单：旧字段/版本、仍在写入者、仍在读取者、流量、最后观测时间、删除 owner 与计划日期。没有 owner 的“临时双写”往往成为永久双写。

5.7.16 16. 发布停止条件应早于用户事故

Canary 自动停止不能只看总错误率。按版本比较 latency、空 feed、候选来源比例、D/E 过滤、F 曝光、cache decode error、unknown enum 和 side-effect lag。新版本只占 1% 时，总体平均会把严重回归稀释一百倍。

图：平均指标怎样掩盖 canary

```
stable 99%：error 0.1% ───────────┐
canary  1%：error 20%  ─┐         ├─ overall ≈ 0.30%
                        └─────────┘   看似仍不高

必须按 code/config/model/schema 版本切片并直接比较。
```

停止动作也要有顺序：停止扩大流量；触发 kill switch；停止新格式写入；保留取证；判断代码 rollback 还是 forward fix；最后处理已写状态。自动化应执行安全、可逆的前几步，不应在证据不足时自动删除或回填数据。

5.7.17 17. 给发布过程建立证据链

每次发布记录 artifact checksum、schema/config/model/corpus 兼容清单、变更 owner、canary 范围、自动停止条件、kill switch、迁移进度与回滚决策。指标按 code/config/model/schema version 切片，避免平均值掩盖小比例故障。

结束发布的条件不是“100% pod 已更新”，而是：混合版本错误归零；新旧数据校验通过；积压事件跨过兼容窗口；旧 cache 使用下降；回滚窗口明确关闭；旧字段删除前已有独立审批。

新手 Tip：把版本直接打印进一次 A—F trace

最小实现可在每个候选账本行附上 source、schema、config、model 与 corpus 版本。让 v1/v2 各生成一次结果，再观察 B 的分数、C 的 cache decode、E 的 unknown safety 和 F 的混排规则是否出现差异。

5.7.18 18. 自检：你能安全地回到旧世界吗

1. 为什么“当前服务版本”应描述为版本向量？
2. v2 writer/v1 reader 这一格为什么决定回滚安全？
3. Expand—Migrate—Contract 的顺序若颠倒，会分别怎样失败？
4. 事件 schema 为什么比同步 API 需要更长兼容期？
5. 新 cache key 怎样同时造成容量和下游风险？
6. 配置部分发布应由哪些日志和指标发现？
7. E 遇到未知 safety enum 时，为什么不能使用普通默认值？
8. 代码回滚后，哪些 v2 状态仍必须处理？
9. 模型版本之外还必须绑定哪些推荐资产版本？

检查点：带着混合世界回到第 17 章

你的 v1/v2 实验应同时覆盖 API、事件、数据、缓存、配置和模型；能在半数升级后产生新状态并安全回滚；能让未知 D/E 安全值保持保守；还能从 trace 精确说出一次 A—F 决策使用了哪组版本。

5.8 协议、数据与配置的在线演化

5.8.1 现场：每个版本单独正确，混在一起却失败

滚动发布期间，v1/v2 节点、旧缓存、新事件和不同配置会同时存在。一次改动如果只在“全量升级后的世界”测试，回滚时最容易发现新写数据已经不能被旧代码理解。

> **新手 Tip：兼容性要双向问**
>
> backward compatibility 问新 reader 能否读旧数据；forward compatibility 问旧 reader 能否容忍新数据。滚动发布和回滚通常两者都需要。

5.8.2 在原仓定位：参数、模型和事件形成三条版本链

阅读 `home-mixer/server.rs` 的参数快照，`home-mixer/scorers/ranking_scorer.rs` 的参数消费，以及 `home-mixer/side_effects/served_candidates_kafka_side_effect.rs`、`home-mixer/side_effects/phoenix_experiments_side_effect.rs` 的事件字段。

列出代码版本、schema version、参数版本、模型版本、corpus version 和 cache key version。公开快照没有完整 feature-switch 配置，能看到消费点，不能证明线上默认值和发布比例。

5.8.3 原理与边界：Expand、Migrate、Contract

安全演化通常分三步：先让 reader 同时理解旧/新格式，再迁移 producer 和存量数据，最后删除旧字段。事件和缓存必须考虑延迟到达与长期残留，数据库迁移还要考虑双写失败、回填和校验。

配置也是分布式状态：推送可能只到部分节点，错误配置可以比代码更快扩大爆炸半径。动态参数必须有 schema、默认值、版本、审计、灰度和 kill switch。

5.8.4 构造：让 v1/v2 同时服务

为 Candidate、ServedEvent 和参数分别设计 v1/v2。v2 增加 `source_attribution` 和新的安全枚举。运行混合拓扑：v1 producer/v2 consumer、v2 producer/v1 consumer、旧 cache/new reader、new writer/old rollback。

实现 unknown field/enum 策略、版本化 cache key、expand-contract migration 和配置校验。请求日志至少报告 code、schema、config/params、model、corpus/index 和 cache-key 六类版本；事件还携带 producer/schema version。

> **验收契约：升级和回滚使用同一份证据**
>
> 混合版本能完成核心请求；根据具体编码器/decoder，未知安全 enum 要么显式报错，要么保守映射，绝不能静默解释成 safe；新 writer 不会在回滚窗口写出旧代码无法读取的数据；配置部分发布可检测并可一键停止。

5.8.5 破坏并证明：半数升级后立即回滚

让 50% 节点升级 v2，产生新事件和缓存，再回滚到 v1。注入配置只到达一个节点、未知 enum、回填中断、双写一边失败和 cache key 冲突。检查数据是否仍可读、实验分桶是否污染、回滚是否真的恢复用户行为。

测试组合必须包含版本矩阵，而不只是当前版本单测。指标包含 requests/events by version、unknown fields、config skew、dual-write mismatch、migration progress 和 rollback errors。

5.8.6 验收：变化成为一种正常状态

- 六类请求版本以及事件 producer/schema version 均可追踪；
- v1/v2 混合测试自动运行；
- schema、数据和缓存采用 expand–contract；
- unknown 安全值保持保守；
- 配置有校验、灰度、审计和 kill switch；
- 回滚不再假设世界会恢复到发布前。

6 第五部：证明、运营与恢复

最后一部把正确性、可观测性和上线能力合并验收。你会建立风险驱动的测试组合，做容量与事故演练，恢复损坏状态，检查权限和故障注入边界，并完成一次像真实上线一样的新 source 交付。

6.1 间章 M：测试不是绿灯——怎样为分布式系统建立可反驳的证据

> **大型 Tip · 间章：从“这个例子通过了”走向“这个风险受到约束”**
>
> 第 18 章要求建立风险—证据矩阵。本间章补齐背后的推理：先把需求写成可观察的规格，再为安全性、活性、并发历史和恢复边界选择证据。测试不是正确性的同义词；好的测试会明确自己试图推翻什么、控制了什么世界、仍遗漏什么世界。

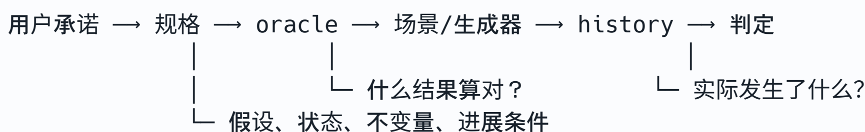
6.1.1 1. 测试之前，先说明“正确”是什么

需求“推荐服务稳定可用”无法直接测试。先把它拆成 specification：给定初始状态、允许的输入与环境行为，哪些输出、状态转换和外部效果合法。实现是被告，规格才是裁判。

规格至少有四层：接口 contract 规定输入输出和错误；invariant 规定任意可达状态都必须成立；safety 规定坏事永不发生；liveness 规定在假设成立时好事最终发生。它们会重叠，但提问方向不同。

- invariant：同一候选 ID 的特征、分数和来源始终属于同一对象；
- safety：D 被屏蔽或 E 未知安全状态的候选永不曝光；
- liveness：未持续失败且仍在 deadline 内的请求最终结束；
- contract：source timeout 返回可归因的部分失败，而不是伪装为空集合。

图：规格怎样变成可执行证据



没有规格：绿灯只表示“实现与测试恰好意见一致”。

活性不能写成无条件“最终成功”：永久网络分区时它不可能兑现。应写明公平性和环境假设，例如“worker 持续获得 CPU、存储最终可用、消息被无限重试时，每个合法事件最终进入 terminal state”。有限测试只能在给定时间和调度边界内寻找反例，不能证明无限时间上的“最终”。

6.1.2 2. Test oracle 是最容易被忽略的部件

Oracle 是把观察结果判为合法或非法的规则。`status == 200`、snapshot 文件、参考实现、状态机、业务不变量、统计阈值都可以是 oracle；它们的可靠性决定测试上限。

最危险的是把当前实现复制成 expected value：生产排序器和测试 helper 使用同一段错误 tie-break，测试会稳定地一起错。优先使用更简单、独立、可人工检查的 oracle：小数据穷举、数学恒等式、公开 contract、朴素模型或前后关系。

没有唯一答案时可使用 partial oracle。推荐结果不必固定为 A/B，但必须满足 D/E 被排除、F 间距合法、分数非增、tie-break 稳定、最少返回数量等约束。不要用过度精确 snapshot 锁死合法的实现空间。

```
图：同一个输出的四层判定

response = [A, B, F]
    │
    ├─ 语法：字段齐全、ID 唯一
    ├─ 语义：score 与 candidate 身份对齐
    ├─ 业务：D/E 不曝光，F 满足 safe gap
    └─ 系统：在 deadline 内完成，无遗留任务/重复副作用

只比较 JSON，最多覆盖第一层和部分第二层。
```

6.1.3 3. A—F 是一组贯穿全书的小型判题集

把固定候选当作可读的反例，而不是神秘 fixture：A 是实时关注内容，B 来自模型召回，C 是缓存或已曝光内容，D 被用户屏蔽，E 的安全状态未知，F 是需受间距与预算约束的广告。

围绕它们可写出独立 oracle：A/B 在 source 顺序改变后身份不漂移；C 不因缓存重试重复进入结果；D/E 在任何降级路径都不能穿透；F 重试不重复扣费，混排规则不因其他 source timeout 失效。

```
图：A—F 从原仓证据到读者实现

原仓公开边界                   A–F 风险                  读者的证据
Candidate Pipeline traits ⟶ A/B 身份与阶段顺序 ⟶ contract + property
Phoenix tests/model        ⟶ B shape/mask/top-K    ⟶ unit + metamorphic
Home Mixer filters         ⟶ D/E 硬边界            ⟶ invariant + fault
cache / served history     ⟶ C 陈旧与重复          ⟶ model + replay
safe-gap / side effects    ⟶ F 间距与幂等          ⟶ property + crash
Thunder event state        ⟶ A 乱序、删除、重放   ⟶ state-machine test

源码提供设计证据；它不自动替读者的实现背书。
```

源码证据：公开仓库已经提供哪些 oracle 线索

`phoenix/test_recsys_model.py` 与 `phoenix/test_recsys_retrieval_model.py` 展示 shape、mask、candidate isolation、归一化与 retrieval 行为；`candidate-pipeline/candidate_pipeline.rs` 及各 trait 展示阶段和接口边界。公开 Rust 快照没有可构建 workspace，不能据此声称主链集成测试已通过；生产数据、基础设施和线上 SLO 也不在仓库中。

6.1.4 4. 一次调用不是历史；并发正确性存在于历史中

并发对象的 observation 应记录 operation ID、调用时间、返回时间、输入、输出、错误和相关版本。所有操作组成 history。只看最终值会漏掉中间已经向用户返回的非法结果。

Linearizability 的工程直觉是：每个已完成操作仿佛在调用与返回之间某一瞬间原子发生，并保持真实时间先后。若写 W 已经返回，之后才开始的读 R 不能假装排在 W 之前；重叠操作则可以按规格允许的任一顺序解释。

```
图：寻找一个合法的线性化顺序

真实时间 ──────────────────────────────────────────→
W1: write(D=blocked)  [ call ───── return ]
R1: read(D)                    [ call ─ return ]   可与 W1 重叠
```

```
R2: read(D)                                   [call-return]

合法解释：R1 可在 W1 前或后；R2 必须在 W1 后看见 blocked。
若 R2 返回 allowed，就不存在保持真实时间的合法顺序。
```

检查器可对小 history 枚举满足实时约束的顺序，并逐步喂给简化状态机。找不到合法顺序就是反例；找到只表示这段有限 history 可解释，不表示所有执行都线性一致。

并非所有系统都承诺 linearizability。缓存 C 可能只承诺 bounded staleness；Thunder-like store 可能承诺按 event version 收敛；F 的预算可能要求单调、不超支。先选择 consistency model，再写 checker，不能拿最强模型随意判死合理设计，也不能用 eventual consistency 掩盖 D 的安全风险。

6.1.5 5. 测试层级不是成熟度排行榜

Unit test 隔离纯函数、状态转换、过滤和 tie-break，反馈最快，但不经过序列化、进程和持久化。Contract test 验证调用双方共享的请求、响应、错误、版本和 ID 对齐；它不证明两个真实进程的配置正确。

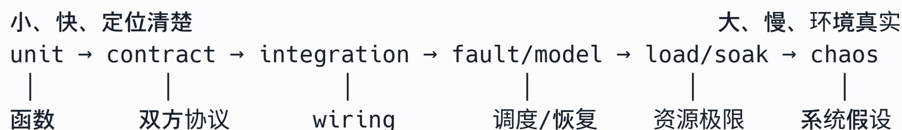
Integration test 连接真实组件边界，发现 wiring、编码、生命周期和存储交互问题；范围越大，定位越慢。Fault test 主动改变 delay、drop、duplicate、reorder、pause、capacity、clock 或 crash，验证失败语义而不只是 happy path。

Property test 生成大量输入验证不变量；metamorphic test 在没有精确答案时验证输入变换与输出关系。例如只有当 merge contract 本来要求顺序无关、输入快照固定且没有 deadline/资源竞态时，改变 source 完成顺序才不应改变 stable top-K；重复同一幂等事件不应改变业务投影；增加必然被过滤候选不应改变曝光。

Model-based test 用简化参考状态机生成命令并比较每步观察；replay test 重放保存的真实或合成事件。前者依赖模型正确且状态较小，后者擅长回归已见世界，却难发现日志从未覆盖的路径。

Load test 寻找吞吐、排队与降级边界；soak test 长时间发现泄漏、累积漂移和周期任务问题；chaos experiment 在受控环境检验系统级假设和运营响应。三者都不是功能测试的昂贵替代品。

图：不同测试看见的故障半径



越向右证据越宽，但噪声、成本与爆炸半径也越大；不是越右越正确。

6.1.6 6. 用确定性调度测试并发，不要靠运气

`sleep(100ms)` 同时充当同步、时钟和猜测：机器快慢一变，测试就失真。把关键边界变成显式事件：source 已开始、写已 durable、ack 尚未发送、取消已传播、replica 尚未 apply。测试通过 barrier 或 step scheduler 推进。

确定性并发不是删去线程，而是控制“下一步谁运行”。对少量 yield point，可系统枚举调度；对较大空间，可用固定 seed 的随机 scheduler，多轮探索后保存最小失败 schedule。

图：把不可控竞态改成可命名步骤

```
生产执行： client —?— worker —?— store —?— ack

测试脚本： ① worker_read
          ② pause_before_write
          ③ duplicate_delivery
          ④ worker_write
          ⑤ crash_before_ack
          ⑥ restart_and_redeliver

每一步可等待、断言、重放；失败不依赖“刚好撞上”。
```

Fake clock 应同时区分 monotonic time 与 wall clock。前者推进 deadline、退避和 lease 本地期限；后者用于事件时间、日志和可能偏移的外部时间。它只能控制既定时钟模型，不能证明跨节点 lease 安全；还要测试 bounded drift、pause 和 fencing 假设。测试主动 advance，不让真实等待决定结果。

Fake network 应建模双向链路：delay、drop、duplicate、reorder、disconnect、half-open 和 bandwidth/capacity。`return error` 只能模拟明确拒绝，不能模拟 timeout 后远端其实已经成功。要允许请求送达而响应丢失，才能验证幂等与不确定结果。

6.1.7 7. Fake 适合控制，真实边界适合揭穿假设

Fake storage 很难复制 fsync、进程页缓存、文件锁和 reopen 行为。涉及“成功后是否耐久”的 contract，必须至少有一组真实 crash test：独立子进程写入，父进程在精确边界终止它，再以新进程重新打开同一持久目录。

正常函数返回、抛异常或优雅 shutdown 都不等于 crash。Crash 不运行 finally、不 flush 用户态 buffer，也不执行补偿。测试应区分 process kill、machine/power-loss 模型和 storage corruption；本地 kill 只能证明其中一部分。

图：副作用边界的杀进程矩阵

```
业务写 + outbox 写 —同一事务—> commit ⟶ publish ⟶ consumer apply ⟶ ack
       X                  X          X             X             X

在每个 X 杀进程并 reopen：
丢失？重复？可重试？能收敛？指标能发现？需要人工 repair？
```

小心：内存测试通过不能证明 durability

“先写状态，再 enqueue”在 fake 中可能永远连续成功；真实进程却能死在两步之间。只有把原子边界交给真实持久层，或采用 outbox 等可恢复协议，才能缩小这个窗口。

6.1.8 8. Property、metamorphic 与 model test 怎样写得有用

生成器应偏向危险边界，而非均匀随机：空集、单元素、相同分数、重复 ID、最大 batch、deadline 前后、版本回绕附近、乱序删除、未知枚举。为每个生成值保留业务含义，失败时才可解释。

好的 property 保护语义，例如“去重后 ID 唯一且首次合法来源可追踪”；坏 property 只是重写实现，例如逐行复刻 production loop。Metamorphic relation 也要验证前提：若新增候选改变了多样性约束，“原排序不变”就不是合法关系。

Model 要故意比实现简单。Thunder-like store 可只保存 `(post_id, version, deleted)`；命令是 create/update/delete/replay，oracle 比较 visible set。若模型也复制并发缓存、后台 compaction 和相同数据库，独立性就消失了。

6.1.9 9. 失败必须能够复现、缩小和解释

随机测试每次记录 seed、生成输入、调度序列、版本、环境和故障脚本。Seed 是入口，不是完整复现保证：生成器或线程数变化会改变随机消费顺序，因此还要保存最终案例与 schedule。

Shrinking 应同时缩小数据与时间：减少候选、事件、节点、操作和故障点，同时保持失败。最终反例最好像“delete(v2)、create(v1)、replay delete(v2)”一样短，而不是一万条不可读日志。

Flaky test 通常暴露未建模时间、共享状态、资源泄漏、顺序依赖或真实竞态。先隔离并保留证据，再修复控制面；盲目 retry 只会把失败率从 1% 变成难以察觉的 0.01%。若临时 quarantine，必须有 owner、原因、失败率和退出日期。

新手 Tip：第一次做并发测试，先学会保存反例

比起一开始追求百万次随机运行，先确保任意失败都会打印 seed、完整操作序列、关键状态和重放命令。不能复现的红灯很难变成知识，不能缩小的红灯很难变成修复。

6.1.10 10. Shadow traffic 是观测工具，不是无害测试

Shadow 把生产请求复制给候选系统，适合比较延迟、错误、source 比例和排序差异，但它没有天然真值：旧系统可能也错；用户不会对 shadow 结果反馈；异步数据视图可能不同。

更重要的是副作用风险。Shadow 请求必须携带不可伪造的模式，默认禁止曝光记录、预算扣减、通知、写缓存污染和事件发布。仅把响应丢弃并不能撤销下游写入。敏感字段还需最小化、脱敏、访问审计和保留期。

Shadow 流量会放大下游 QPS、cache miss 和模型成本，不能绕过 admission control。先离线 replay 或采样，再逐步扩大，并分别监控 shadow 自身及其对 primary 的干扰。

6.1.11 11. 风险—证据矩阵迫使你承认空白

矩阵的行是风险/不变量，列不是测试文件名，而是故障模型、oracle、证据层、运行频率和未覆盖边界。每项选择最便宜且足够强的证据；同一高风险项通常需要快测试防回归、真实边界测试防假设、运行时指标发现未知。

图：一张可执行的风险—证据矩阵

风险	oracle	主要证据	仍未证明
ID/特征错位	candidate ledger	contract + property	生产脏数据
D/E 穿透	safety invariant	fault + model	未知新枚举
deadline 泄漏	task/slot 归零	fake clock/network	OS 调度尾延迟
重复副作用	operation ID 唯一	crash + replay	外部系统丢去重表
stale writer	resource max_epoch	history/model	协调服务实现
过载雪崩	queue/P99/reject	load + soak	真实跨租户流量
混合版本	version matrix	integration	长尾旧客户端

风险排序考虑影响、发生概率、可探测性与恢复成本。D/E 安全边界、F 金钱副作用和持久状态损坏即使低频，也值得真实故障测试；纯展示字段可以用较轻证据。矩阵应随事故、架构和版本变化更新，而不是一次性文档。

6.1.12 12. Chaos 的前提是停止条件，不是勇气

Chaos experiment 应从假设开始：“单个 Retrieval 实例消失时，bulkhead 保证 Mixer P99 与最低候选数仍在 SLO 内。”然后写 steady-state 指标、注入范围、观察窗口、自动 abort、kill switch、owner 和恢复验证。

不要在未知基线、无隔离、无回滚或 error budget 已耗尽时扩大实验。Chaos 能检验已设计的韧性和运营路径，不能替代单元测试，也不应用生产用户替团队完成首次调试。

6.1.13 13. 为本书项目建立最小而完整的证据梯

每次提交运行纯函数、contract、property 和小状态机测试；合并前运行多进程 integration、fake fault、版本矩阵与确定性 schedule；周期任务运行真实 crash/reopen、load、soak 和较大 replay；受控环境才运行 chaos 与 shadow。

失败产物统一包含 request/event/operation/attempt ID、code/config/model/schema version、seed、schedule、fault script、关键 history 和 artifact checksum。这样测试、事故和回归共享同一种证据语言。

对第 18 章的组合事故，不要只保留一个端到端红灯。把它分解为：Retrieval deadline 合约、retry budget property、C served-history 状态机、最低 feed 质量 oracle；再保留一个小型组合测试证明这些保护能一起工作。

6.1.14 14. 自检：你的绿灯究竟证明了什么

1. Specification、invariant、safety 与 liveness 各自约束什么？
2. 为什么有限测试不能无条件证明“最终一定成功”？
3. Oracle 与实现共享同一算法会产生什么盲区？
4. R2 在 W1 返回后才调用，为什么不能读到 W1 之前的 D 状态？
5. 各测试层的证据边界是什么；无唯一输出时怎样设计 metamorphic oracle？
6. Fake network 为什么要模拟响应丢失；真实 crash 又补了什么？
7. Seed、shrinking 与 schedule 怎样共同解释失败？Shadow 有什么副作用风险？
8. 风险—证据矩阵为何必须保留“仍未证明”一列？

检查点：带着可反驳的证据回到第 18 章

你应能为 A—F 写 oracle、检查并发 history，以可控网络和真实 crash 重放故障，保存并 shrink 反例，再用风险—证据矩阵标明每盏绿灯的保护范围与证明空白。

6.2 用风险驱动测试建立分布式证据

6.2.1 现场：结果对了，不代表系统是对的

一次 feed 返回 A/B，只证明这一组输入暂时得到期望结果。它没有证明候选身份未错位、取消后没有僵尸任务、重复事件不会重复生效、旧 leader 无法写入，也没有证明 v1/v2 混跑和重启恢复。

> **Tip：按风险选择最便宜证据**
>
> 纯函数先用单元测试；跨组件契约用 contract test；持久化边界要真实重启；并发不变量适合性质或模型测试；生产未知再用灰度、shadow 和事故演练。不要所有问题都启动全系统。

6.2.2 在原仓定位：官方测试保护哪些语义

阅读：

- `phoenix/test_recsys_model.py`；
- `phoenix/test_recsys_retrieval_model.py`；
- `candidate-pipeline/candidate_pipeline.rs` 及各 trait 合约。

Phoenix 官方测试覆盖 shape、attention mask、candidate isolation、归一化和 retrieval 行为；Rust 公开主链缺少可运行 workspace，不能声称已有集成测试。把"仓库已有证据"和"我们希望拥有的测试"分成两列。

6.2.3 原理与边界：测试对象是不变量，不是函数数量

普通测试只能在给定实现、调度和故障模型下建立有界证据，不能等同于形式证明。推荐分布式系统至少需要：

- unit：score、filter、routing、state transition；
- contract：source/hydrator ID 对齐、错误与版本兼容；
- integration：多进程请求、durable event、replication；
- fault：timeout、取消、重复、乱序、崩溃、过载；
- property/metamorphic：排列不变、重复幂等、节点变化时 movement ratio 符合路由算法预期；
- replay/model-based：事件序列与参考状态机；
- load/soak：饱和点、泄漏和长时间积压。

确定性不是把并发删掉，而是用 barrier、fake clock、可控网络和显式事件控制调度点。

6.2.4 构造：建立一张风险—证据矩阵

整理前 17 章所有不变量，为每项选择最便宜且足够强的测试层。要求至少覆盖：身份对齐、filter 硬边界、stable top-K、deadline 传播、任务取消、事件幂等、crash recovery、queue bound、shard movement、fencing 和混合版本。

为故障注入定义统一接口，但实现由读者自己完成：delay、error、drop、duplicate、reorder、crash、pause、clock offset、capacity limit 和 stale replica。Clock offset 通过注入时钟抽象或协议字段模拟，不要求修改操作系统时钟。

> **验收契约：测试是系统的可执行说明**

每个核心不变量至少有一种自动化证据；测试失败能指出哪个合约破坏；并发测试不用裸 `sleep` 作为同步；恢复测试真的关闭并重启进程；版本矩阵进入常规测试。

6.2.5 破坏并证明：把合成事故钉成回归测试

构造一次组合事故：Retrieval 变慢，retry 增加负载，Cache 返回 C，Served History 写入滞后，过滤使最终 feed 不足。先保留失败证据，再写最小回归组合，分别保护时间预算、重复过滤和最低 feed 质量。

检查测试是否会产生假阳性：只断言数量、只断言状态码、等待固定时间、忽略日志错误、复用生产全局配置、随机种子不可重现。对 flaky test 必须修复根因，不能用盲目重跑掩盖。

6.2.6 验收：正确性拥有多层证据

- 原仓已有测试与课程新增测试分开；
- 风险矩阵覆盖正常、故障、恢复和演化；
- 并发与时间测试可确定性复现；
- 状态机支持重放或模型对照；
- 组合事故成为回归测试；
- 测试成本与证明强度相匹配。

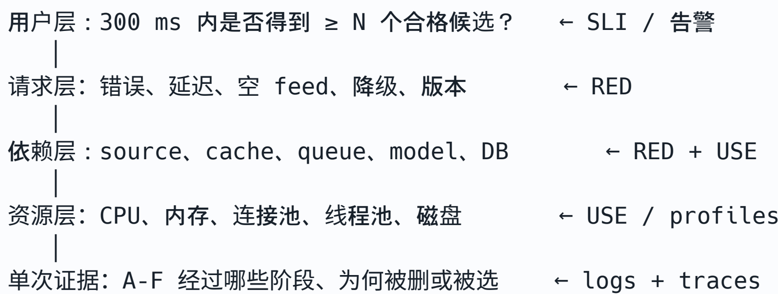
6.3 间章 N：看见、承诺与恢复——可观测性和 SRE 的完整闭环

> **大型 Tip · 间章：遥测不是装饰，而是关于系统的可证伪陈述**
>
> 第 19 章要求你建立日志、指标、trace、SLO、容量表和事故流程。本间章补齐它们之间的因果关系：先定义用户眼中的好事件，再用有限成本记录系统内部证据，最后让告警、止血、验证和改进形成闭环。面板很多不等于可观测；真正的标准是遇到事先没有列举的故障时，能否从系统输出推断内部状态。

6.3.1 1. Monitoring 与 observability 不是同义词

Monitoring 回答预先提出的问题，例如"错误率是否超过 1%"；observability 强调能否根据外部输出推断未知内部状态。前者适合稳定告警，后者支持探索诊断。生产系统两者都需要：没有 monitoring，值班不知道何时介入；没有 observability，介入后只能猜。

先区分三类信号：用户结果、系统状态、调试证据。用户结果决定是否违约；系统状态解释容量和依赖；调试证据还原某次行为。不要从"我们能采什么"出发，而应从"什么判断需要什么证据"反推采集。



图：从用户问题向内钻取，而不是从 CPU 向外猜

6.3.2 2. 四种基本信号各自擅长什么

Logs：离散事件的结构化记录，擅长回答"哪一个、发生了什么、上下文是什么"。 **Metrics**：按时间聚合的数值序列，擅长趋势、比较、告警和低成本长期保留。 **Traces**：一次请求跨进程的因果路径，擅长解释时间花在哪里、哪条依赖失败。 **Profiles**：一段时间内 CPU、内存、锁或 I/O 成本的统计归因，擅长解释资源花在哪里。

四者不是四份重复数据。日志若记录每次请求耗时，仍不适合低成本计算全站长期分位数；指标能说 P99 上升，却通常不能指出候选 E 为何穿透；trace 能展示慢 span，却不证明该样本具有总体代表性；profile 能发现 scorer 热点，却不说明小林是否获得了合格 feed。

问题：哪个 E 被误放？ → log/trace
问题：全站是否退化？ → metric
问题：CPU 为何耗尽？ → profile

图：信号的聚合程度与调查范围

6.3.3 3. 结构化日志要记录决定，而非倾倒对象

日志至少包含 timestamp、severity、service、instance、code/config/model version、request/trace/span ID、event name 和稳定字段。推荐请求还应记录阶段输入输出数量、source、drop reason、degrade reason 与最终质量摘要。敏感文本、完整特征和用户标识不应直接写入。

request ID 标识一次逻辑请求；attempt ID 区分重试；trace ID 连接调用树；event ID 支持异步去重。把四者都叫 correlation ID 会在重试和事件回放时失去语义。

日志等级表达处置需要，而不是开发者情绪。可自动恢复的单次下游失败通常不是 ERROR；导致请求违反契约或需要人工介入的事件才应升级。相同错误应聚合或限速，否则故障本身会制造日志 I/O 风暴。

> **小心：候选 ID 不应成为 metric label**
>
> A–F 可以出现在受采样、受权限控制的日志或 trace event 中；若把 candidate_id、request_id 或 user_id 放入指标标签，时间序列数量会随流量无界增长，先拖垮的可能正是监控系统。

6.3.4 4. Metric 类型决定你能回答的问题

Counter 只递增，适合 requests、errors、drops、bytes；查询时通常计算窗口 rate 或 increase，进程重启造成归零必须由查询处理。Gauge 可升可降，适合 queue depth、inflight、cache entries，但一次抓取只能看到瞬时值。Histogram 把观测放入可聚合 bucket，并保留 count/sum；summary 或客户端 quantile 往往不能跨实例正确合并。

指标名应带单位和语义，例如 `request_duration_seconds`、`candidate_dropped_total`。标签只使用有界维度：service、operation、source、result、reason、version；每增加一个标签，都估算所有取值笛卡尔积和保留成本。

图：Cardinality 怎样乘出来

```
source(8) × result(4) × region(6) × version(5) = 960 series

再加 user_id(10,000,000) → 理论上 9,600,000,000 series
再加 request_id          → 每个请求都创造新 series
```

标签不是数据库字段；可筛选不代表适合成为标签。

6.3.5 5. Histogram、percentile 与平均数的陷阱

平均延迟会掩盖尾部：99 次 10 ms 和 1 次 5 s 的平均约 60 ms，但那一个用户等待了 5 s。P99 表示约 99% 观测不大于该值，不表示“最慢 1% 的平均”，也不保证每一分钟都满足。

Histogram bucket 必须围绕决策阈值设计。若 SLO 阈值是 300 ms，却只有 100 ms 与 1 s 两个边界，就无法准确计算 good events。Bucket 太密增加成本，太疏丢失分辨率。原始样本、HDR histogram 或 sketch 各有权衡，书中只要求你明确误差边界。

不能平均多个实例的 P99；quantile 不是可加统计量。也不能把三个阶段 P99 相加当端到端 P99，因为慢样本未必来自同一请求。可聚合 histogram 应先合并 bucket 再求分位数，端到端延迟应直接在入口测量。

> **小心：Coordinated omission 会把过载测得过于漂亮**

若负载发生器等待上一个响应后才发下一个请求，服务暂停 2 秒时，发生器也暂停，没有记录本应在这 2 秒到达并排队的请求。开放环负载按计划到达时间发送，或记录校正延迟，才能暴露排队和停顿对真实用户的影响。

6.3.6 6. RED 和 USE 是两副不同的检查镜

RED 用于请求驱动服务：Rate、Errors、Duration。每个服务入口和关键依赖都可用它快速判断有没有流量、是否失败、是否变慢。错误必须按契约分类，HTTP 200 的空 feed 或严重降级也可能是用户错误。

USE 用于资源：Utilization、Saturation、Errors。Utilization 是忙碌比例；Saturation 是无法立即服务的工作，如 run queue、连接等待和队列深度；Errors 是磁盘、网络、分配或设备错误。CPU 40% 不代表无瓶颈：单核饱和、锁竞争或连接池耗尽都可能让请求排队。

图：RED 发现症状，USE 寻找资源解释

```
Mixer RED: rate 正常 / error 上升 / duration 上升
       │
       ├─ Retrieval RED：duration 上升
       │       └─ connection pool USE：saturation 上升
       ├─ Thunder RED：正常
       └─ Filter RED：rate 正常、drop 激增（业务异常）
```

结论来自信号组合，不是“CPU 正常所以服务正常”。

6.3.7 7. Trace 是带上下文的时间因果图

一个 trace 由 spans 构成。Span 应记录 operation、start/duration、status、parent、关键有界 attributes 和 events。父子关系表示调用因果，不只是时间嵌套；异步队列可用 link 连接 producer 与 consumer，避免伪造一个持续数小时的父 span。

Trace context 必须跨 HTTP/RPC header、任务队列和后台执行器传播，同时设置信任边界：外部调用方提供的 trace ID 不能绕过采样、注入任意 baggage 或污染日志。Baggage 会在每跳复制，应只放少量必要字段，不能携带候选列表。

Head sampling 在请求开始时决定，成本稳定却容易漏掉后来才知道的慢请求和错误；tail sampling 收集完整 trace 后按错误、延迟或属性决定，诊断价值更高但需要缓冲和集中决策。过载时无法保证全保留，应按策略优先保留错误与极慢请求、为采样丢弃本身计数，并保护 collector 容量。

图：一次 A–F 请求的 trace 与候选账本

```
Mixer request [trace=t7]
├─ Following source  12ms → A, D
├─ Phoenix source    74ms → B, E
├─ Cache source       4ms → C
├─ Ads source        18ms → F
├─ Hydrate            9ms → E safety=unknown
├─ Filter             4ms → drop D(block), E(unknown)
├─ Rank               7ms → B > A > C
└─ Blend              2ms → B, A, F, C
```

trace 解释路径；账本解释每个候选的状态变化。

6.3.8 8. Profiles 解释“代码把资源花在哪里”

CPU profile 统计运行样本落在哪些调用栈；allocation/heap profile 查找分配热点和常驻对象；mutex/block profile 揭示锁等待；I/O profile 解释文件与网络等待。Flame graph 的宽度代表采样占比，不代表时间顺序，也不能单凭函数名认定根因。

持续 profiling 适合捕捉偶发回归，但要限制开销、权限和保留期。比较部署前后同一负载、同一版本切片，比看一张孤立 flame graph 更可靠。Profile 发现 ranking scorer 占 CPU 60%，仍需结合请求量、候选数和延迟确认它是否异常。

6.3.9 9. 从 good event 推导 SLI、SLO 与 SLA

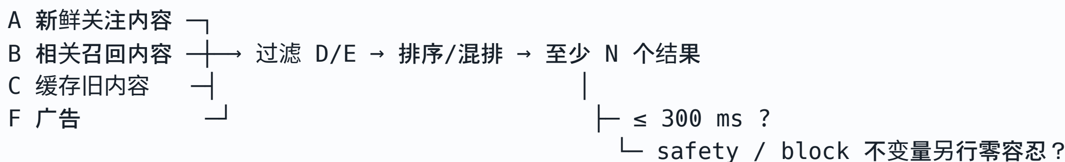
SLI 是被测量的服务水平指标；SLO 是目标；SLA 是与客户或组织约定、可能包含补偿的协议。SLO 通常应比 SLA 更严格，以便在违反外部承诺前有处置空间。不要把三者都写成“99.9% uptime”。

定义 valid event 排除明确不属于服务责任的请求，但不能在事故后临时扩大排除条件。可用性/质量 good event 可以写成：“授权且参数合法的 For You 请求，在 300 ms 内返回至少 10 个候选”。D/E 穿透则是独立 safety invariant：确认违规应触发 incident/stop，不能因为总体 SLO 尚有 error budget 就接受；可以记录 safety-violation SLI 用于探测，但 budget 不授权违规。

$$\text{SLI} = \frac{\text{good events}}{\text{valid events}}$$

若 28 天有 10,000,000 个 valid events，99.9% SLO 允许 10,000 个 bad events。Error budget 是允许的不可靠量，不是“可以故意制造的错误”，而是用来权衡发布速度、风险和可靠性投资的共同货币。

图：从候选 A–F 到用户 SLO



“HTTP 200”只覆盖传输；good event 还覆盖数量、时限和质量。

6.3.10 10. Window 与 burn rate 决定何时叫醒人

SLO 可以用滚动窗口或日历窗口。滚动窗口始终观察最近 N 天，更适合连续运营；日历窗口便于结算，却会在月初和月末产生不同风险。选择后应保持一致并解释数据迟到、重算与缺失怎样处理。

Burn rate 表示 error budget 被消耗得有多快。99.9% SLO 的允许错误率是 0.1%；若当前 bad rate 为 1%，burn rate 为 10。持续以 10 倍燃烧，28 天预算约 2.8 天耗尽。

多窗口、多 burn-rate 告警同时要求短窗口和长窗口越界：短窗口快速发现猛烈事故，长窗口过滤瞬时噪声。Page 应对应立即动作；ticket 可处理缓慢但持续的预算消耗。固定“错误率 > 1%”无法适配不同 SLO 和窗口。

图：同一预算的两种燃烧形态

告警关注耗尽速度，而不只关注瞬时尖峰。

6.3.11 11. 用户 SLO 与依赖 SLO 不能简单相等

用户 SLO 是产品承诺；dependency SLO 是内部预算和诊断边界。Mixer fan-out 三个 source，若要求每个依赖都达到与用户相同的 99.9%，组合可靠性通常低于 99.9%；但若 source 可选且有高质量降级，依赖失败又未必造成用户 bad event。

为每个依赖写 contribution：调用比例、失败时用户影响、timeout、fallback、预算份额。关键安全 source 可能必须 fail closed；个性化 source 可局部失败；Ads source 失败不应阻断有机内容。不要把所有依赖错误相加，也不要因顶层 SLO 尚好就忽视正在吞噬冗余的依赖。

源码证据：原仓中的观测边界与 A–F 映射

从 `candidate-pipeline/candidate_pipeline.rs` 盘点 pipeline tracing/stats 边界；从 `thunder/thunder_service.rs` 观察实时查询入口；从 `home-mixer/scorers/ranking_scorer.rs`、`home-mixer/filters/previously_seen_posts_filter.rs` 与 `home-mixer/side_effects/served_candidates_kafka_side_effect.rs` 找到分数、过滤和异步副作用的证据点。A 来自 Following，B 来自 Phoenix，C 来自缓存，D 是屏蔽命中，E 是安全未知，F 是广告混排。公开源码只能证明调用与埋点位置，不能证明线上 dashboard、阈值、采样率或 X 团队真实 SLO。

6.3.12 12. 容量计划要从工作量闭合到资源

容量不是“CPU 保持 60%”一句话。先预测峰值有效 QPS，再乘 fan-out、retry amplification、shadow/canary 流量和每请求候选数，得到依赖 QPS、在途请求、CPU、内存、连接、队列写入和存储增长。

Little’s Law 给出稳定系统中 $L = \lambda W$：平均在途量等于到达率乘平均停留时间。它不能替代尾延迟分析，却能做量纲检查。500 QPS、平均 200 ms，约有 100 个请求在途；若连接池只有 50，排队并不意外。

容量表至少写正常峰值、单可用区故障、依赖变慢、发布双跑与恢复回放。Headroom 用来吸收不确定性和故障，不应被日常流量长期吃满。负载测试要使用真实候选规模、缓存命中率、混合请求和开放环到达。

6.3.13 13. Dashboard 应沿调查路径组织

首页只回答：用户是否受影响、范围多大、何时开始、还在恶化吗。第二层按 source、region、tenant、code/config/model version 切片。第三层才是队列、连接池、runtime、主机与 profile。把所有图堆在一页会让最重要的信号消失。

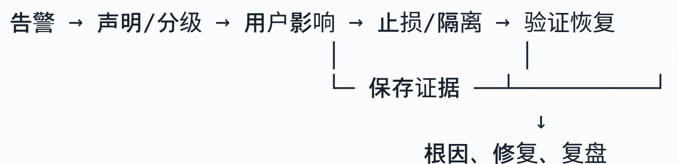
推荐服务至少展示 request SLI、合格候选数、空 feed、D/E 拦截、各 source 贡献与延迟、cache age/hit、queue lag、side-effect success、版本分布和预算燃烧。每张图标注单位、聚合、窗口、数据延迟和相关发布事件。

6.3.14 14. 事故响应先控制影响，再追完整根因

事故角色可以精简，但职责要明确：Incident Commander 维护优先级和决策；Operations Lead 执行技术止血；Communications Lead 同步利益相关方；Scribe 记录时间线和证据。小团队一人可兼任，不能让所有人同时无协调地改系统。

标准顺序是：声明事故；确认用户影响与安全边界；限制爆炸半径；停止继续损坏；自动或并行快速保存关键证据；执行最小可逆 mitigation；验证用户恢复；持续观察；最后才做深入根因分析。不能为了完美取证延误 containment；安全事件的特殊取证要求在间章 O 处理。

图：事故不是从告警直达根因



先恢复安全服务；未知根因不妨碍执行可逆止血。

6.3.15 15. Mitigation 成功必须由用户证据验证

“CPU 降了”“pod 全绿”“错误日志停止”都不是充分恢复证据。验证至少包括：用户 SLI 回到目标；A–F 结果满足数量与安全不变量；积压停止增长并按计划消化；状态没有继续分叉；版本和配置处于已知组合；关闭临时措施不会立刻复发。

Mitigation 可能制造新风险：关闭 Phoenix 后 feed 数量下降；fail-open 恢复数量却让 E 穿透；暂停 side effect 后 served history 不再更新；大量重启造成 cache cold start。每个止血动作都要写预期副作用、观察指标和撤销条件。

6.3.16 16. Runbook 是压力下可执行的决策树

Runbook 应包含触发症状、适用/不适用范围、权限、第一批查询、危险动作、止血步骤、验证、升级联系人和退出条件。命令应可复制但参数显式，不要写“重启一下”“看日志判断”。Kill switch 必须说明默认状态、传播延迟和审计。

每次演练和事故后运行 runbook：链接是否失效、权限是否足够、面板名称是否变化、步骤是否仍安全。无人执行过的 runbook 只是愿望。高风险恢复动作应双人确认，并优先提供 dry-run。

6.3.17 17. Postmortem 的产物是系统改进，不是漂亮故事

无责不等于无责任边界。复盘应记录影响、检测、时间线、触发条件、放大机制、止血与恢复、哪些防线成功/失败，以及带 owner 和期限的行动项。根因不应停在“工程师输入错误”，而要追问为何系统允许错误传播、为何未提前发现、为何恢复困难。

行动项分为消除触发、限制爆炸半径、加速检测、加速恢复和验证知识五类。不是所有项都必须完成；应按风险和成本排序，并明确接受的残余风险。重复事故说明行动项或学习闭环失效。

新手 Tip：第一次演练可以把遥测写到本地文件

不需要先安装 Prometheus、Jaeger 或 Grafana。用 JSON Lines 写结构化日志和 spans，用内存 counter/histogram 定期导出，用采样 profiler 或语言自带工具观察 CPU。关键是字段语义、关联关系和判断过程；更换采集后端不应改变你的 SLI 与事故契约。

6.3.18 18. 常见反例：看起来专业，却不能保护系统

- 指标很多但没有 good event，无法判断用户影响；
- 只看 HTTP error，忽略空 feed、E 穿透和严重降级；
- 平均实例 P99，得到没有统计意义的数字；

- trace 全采样，事故时先压垮 collector；
- 日志写完整用户特征，制造隐私与成本事故；
- 告警没有 owner、动作和抑制，最终被静音；
- 依赖 SLO 直接复制用户 SLO，忽略 fan-out 与 fallback；
- 容量测试使用闭环单线程，漏掉 coordinated omission；
- mitigation 后只看基础设施绿灯，没有用户层验证；
- postmortem 行动项全是“加强培训”和“更加小心”。

6.3.19 19. 自检：证据链是否真的闭合

1. Monitoring 和 observability 分别解决什么问题?
2. Logs、metrics、traces、profiles 各自不能替代谁?
3. Counter、gauge、histogram 的错误选择会造成什么误判?
4. 为什么不能平均实例 P99，也不能相加阶段 P99?
5. Coordinated omission 怎样隐藏服务暂停和排队?
6. RED 与 USE 分别从哪一层观察系统?
7. Head sampling 与 tail sampling 的成本和漏报有何不同?
8. A–F 的 good event 为什么不能退化成 HTTP 200?
9. Error budget 和 burn rate 怎样改变告警与发布决策?
10. 为什么 dependency SLO 不能机械等于 user SLO?
11. 单可用区故障和恢复回放为何必须进入容量表?
12. Incident Commander、止血、验证分别保护什么?
13. Runbook 与 postmortem 怎样形成下一轮可靠性输入?

检查点：带着一条可执行证据链回到第 19 章

把这些判断并入第 19 章：从 A–F 定义 valid/good event 和三个用户 SLI；为入口与依赖建立 RED，为关键资源建立 USE；解释指标类型、cardinality、histogram、采样与 profile 的边界；用 burn-rate 告警触发限时演练，按角色止血并以用户层结果验证。Runbook、无责复盘和有 owner 的改进行动都应成为同一工程里程碑的证据，而不是间章之外的第二份作业。

6.4 可观测性、SLO、容量与事故响应

6.4.1 现场：小林只看到 A，值班同学先看哪里

空结果可能来自 source 失败、hydration 缺失、过滤激增、排序异常、缓存陈旧或混排约束。没有阶段账本、版本和依赖指标时，值班只能从"模型是不是坏了"开始猜。

> **新手 Tip：指标回答多少，日志回答哪个，trace 回答怎样**
>
> 指标适合趋势和告警；日志记录离散事件；trace 连接一次请求的因果路径。不要把 candidate ID 当指标标签，也不要要求日志承担全站百分位统计。

6.4.2 在原仓定位：区分已有证据与希望拥有的面板

从 `candidate-pipeline/candidate_pipeline.rs` 的 tracing/stats 调用，`thunder/thunder_service.rs` 的查询边界，以及 scorer、filter、side effect 文件中盘点已有观测点。按阶段整理：latency、input/output size、error、drop reason、selected 和 side-effect result。

公开调用点不能证明生产 dashboard、告警阈值或 retention。正文只把它们写成可审计事实，面板设计属于读者交付。

6.4.3 原理与边界：从用户 SLI 向下钻取

先定义可用性/质量 good event："有效请求在 300 ms 内返回至少 N 个 item"；对应 SLI 是窗口内 good events / valid events 的实测比例，SLO 才是"该比例在 28 天窗口达到 99%"。D/E 穿透属于独立 safety invariant，error budget 不授权违规。下层 source、queue、cache 和 model 指标用于解释顶层偏离。

容量计划要闭合入口 QPS、fan-out、重试、在途并发、队列、连接池、事件增长和存储。Error budget 把允许失败量变成发布节奏；预算持续燃烧时应先恢复可靠性。

事故响应顺序是：确认用户影响、限制爆炸半径、停止继续损坏、保存证据、恢复服务、验证恢复、再找完整根因。

6.4.4 构造：建立最小生产证据面

为自建系统输出结构化日志、阶段 metrics 和跨进程 trace context。实现一个简单直方图或保存延迟样本计算 P50/P95/P99；生产环境通常使用 histogram/sketch，不能平均各实例 P99，分位数也不能跨阶段直接相加，bucket 设计会限制可回答的问题。主线不要求 Prometheus/Grafana。

编写 dashboard 规格：请求成功与 feed quality、source、hydration/filter、ranking/blending、queue/events、cache/history、replication/version。再写三个 SLO、对应 error budget 和容量表。

> **验收契约：告警必须能导向动作**
>
> 每个告警包含用户影响、可能故障域、第一检查项和止血动作；高基数不污染指标；trace 能跨进程关联；恢复后有用户层验证而不只看 CPU 回落。

6.4.5 破坏并证明：完成一次限时事故演练

使用测试章的组合事故，在 30 分钟时限内执行：声明 incident、确认影响、关闭重试或降级慢 source、检查安全与最低 feed 数、处理事件积压、验证恢复。记录每个判断使用的证据和时间。

随后写一页无责复盘：时间线、影响、触发条件、放大机制、为何未提前发现、止血、长期修复、回归测试和 owner。禁止把“某人操作失误”当作根因终点。

6.4.6 验收：值班不再靠猜

- SLI/SLO 描述用户体验；
- 指标、日志、trace 职责清晰；
- 容量预算包含 fan-out 与重试；
- 告警对应具体动作；
- 事故先止损再追根因；
- 恢复有用户层和状态层双重证据。

6.5 间章 O：系统活着不等于业务恢复——灾备、安全、隐私与多租户

大型 Tip · 间章：恢复的是不变量，不是进程数量

服务重新监听端口，只说明进程存活；副本追平、业务效果不丢不重（重复投递不重复生效）、屏蔽关系有效、广告预算正确、隐私删除完成，才说明业务恢复。本间章把恢复工程与安全工程放进同一套边界思维：先识别资产与不变量，再限制故障和攻击的爆炸半径，最后用演练证明恢复路径真的可用。

6.5.1 1. 先定义“恢复到什么”

恢复目标至少有四层：基础设施可用、服务可响应、数据达到可接受状态、用户体验恢复。它们不能互相替代。Mixer 返回 200，但只剩 F 广告、D 的屏蔽失效或 E 的安全状态未知，都不算恢复。

设事故发生时刻为 t_f，实际恢复点为 t_r，业务不变量校验通过且可安全放量的时刻为 t_{safe}：

$$\text{data_loss_window} = t_f - t_r \le \text{RPO_target}$$

$$\text{recovery_time} = t_{\text{safe}} - t_f \le \text{RTO_target}$$

RPO（Recovery Point Objective）和 RTO（Recovery Time Objective）是目标；演练测到的是 achieved recovery point 与 achieved recovery time，再与目标比较。二者是业务承诺，不是备份软件属性。若 served history 允许退回一小时，用户可能重新看到刚看过的 A；屏蔽状态通常不能接受同样目标，否则可能重新展示 D。

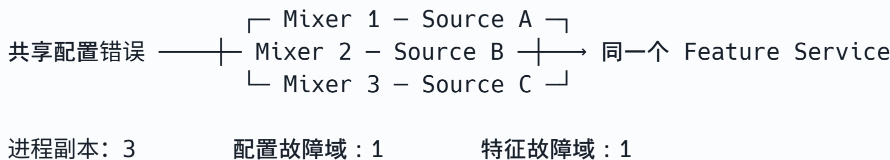
图：恢复不是一个开关

6.5.2 2. 先做故障域地图，再谈高可用

机器、进程、机架、可用区、区域、云账号、证书机构、身份系统、配置控制面都可能成为故障域。把两个副本放在同一宿主机，只能抵抗进程失败；跨区复制但共享错误配置，不能抵抗控制面事故。

对每个关键依赖记录：故障域、共享命运、检测信号、自动动作、人工动作与最大爆炸半径。推荐链路尤其要标出 fan-out：一个共享 feature service 可能同时影响 A、B、C 三个 source，表面上的多路召回并不独立。

图：副本很多，不代表故障域独立

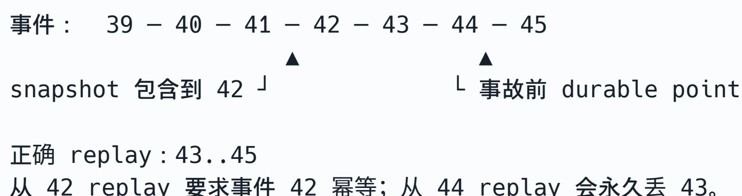


6.5.3 3. Backup 只有经过 restore 才是证据

备份成功指标只能证明某个任务写出了一批字节。它没有证明密钥仍可用、格式仍可读、依赖版本仍存在、增量日志连续，也没有证明恢复后满足业务不变量。

常见恢复链是：加载基线 snapshot，按确定顺序 replay 增量日志，重建索引和缓存，再执行一致性校验。snapshot 与日志必须共享明确的 checkpoint；否则 snapshot 已包含事件 42，replay 又从 40 开始，就会重复生效。

图：Snapshot 与增量日志的接缝



恢复演练应在隔离环境中真正读取备份，而不是复制线上当前状态。记录恢复所需的人员、权限、密钥、带宽、容量与时间。若恢复脚本只存在某位工程师的笔记本，它本身就是单点。

6.5.4 4. 派生状态应该能重建，但重建也消耗资源

缓存、向量索引、推荐特征和统计聚合通常是派生状态。理想上它们可由 source of truth 重建；现实中重建可能耗时数小时，并制造回源、CPU 与网络洪峰。

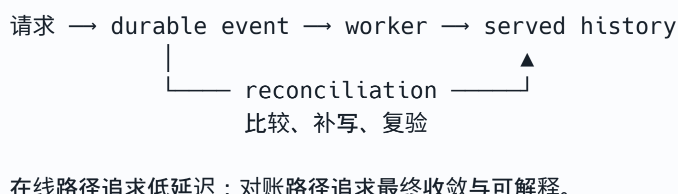
因此要同时定义：派生数据的来源版本、重建顺序、速率限制、读路径降级、校验方式和停止条件。重建 Phoenix corpus 时，不能让半成品索引悄悄承接全部 B 流量；可以使用版本化索引，完成校验后原子切换活动引用。

6.5.5 5. Reconciliation 是长期恢复机制

重试处理已知的短暂失败；reconciliation（对账修复）处理系统已经失去精确失败现场后的状态分歧。它比较“应该是什么”和“现在是什么”，生成幂等修复动作。

例如 served event 已持久化但 history 更新失败，在线请求早已结束。后台对账可以扫描事件与 history 的差集，补写缺失记录。修复任务需要游标、速率限制、审计、dry-run 和二次校验，否则修复器会成为新的事故源。

图：在线路径与对账路径



6.5.6 6. 恢复顺序来自依赖图，不来自服务名单

恢复时若先启动 Mixer、后启动身份和屏蔽服务，系统可能在“可用”窗口泄露内容。应把恢复顺序写成依赖 DAG：身份与策略数据、核心状态、索引与缓存、source、Mixer、异步副作用，最后逐级放量。

每一步都有 gate，例如：D 屏蔽数据版本不早于 checkpoint；E safety corpus 校验通过；F 的预算扣减总额与幂等事件守恒、预算不超支且频控不越界；A–C 每路候选量位于基线区间。不能只用 health check 决定进入下一步。

6.5.7 7. 安全从资产、主体、动作和边界开始

威胁建模不必先背攻击名称。对每条数据流回答：保护什么资产；谁是主体；允许什么动作；信任边界在哪里；攻击者能伪造、重放、篡改、窃听或耗尽什么。

身份认证回答“你是谁”，授权回答“你能做什么”。TLS 保护传输并不自动授权调用方代表任意 viewer。内部 source 若接受调用者传入的 `viewer_id`，必须验证调用链上的主体是否有代理该用户的权限。

图：一次内部调用的三种身份

```
用户 U ⟶ Mixer 服务 M ⟶ History 服务 H
          caller=M
          subject=U
          request/trace=R
```

认证 M 不能自动推出：M 可读取任意 U 的历史。

6.5.8 8. 最小权限与短期凭证

服务账号只应访问完成职责所需的资源、租户和动作。读推荐特征的身份不应同时能修改安全规则；发布配置的身份不应能读取原始私密事件。权限边界越宽，凭证泄露后的爆炸半径越大。

优先使用可轮换的短期凭证，并验证轮换时的新旧重叠窗口。密钥不应进入代码、镜像、日志、trace 或错误响应。紧急 break-glass 权限要短时、双人审批、完整审计，并在演练后撤销。

6.5.9 9. 重放防护与幂等不是同一件事

幂等键避免相同业务动作重复生效；重放防护还需确认请求是否来自合法主体、是否仍在有效时间窗、消息是否被篡改。攻击者复制一个过去合法的“增加预算”请求，即使系统按 request ID 幂等，也可能在键过期后再次成功。

对高风险动作绑定主体、租户、动作摘要、nonce、签名和过期时间；服务端持久记录已消费 nonce 或业务唯一键。不能仅依赖客户端时间，因为时钟可偏移或伪造。

6.5.10 10. 隐私不是“日志里不写姓名”

候选、曝光、点击、屏蔽、设备与 embedding 组合起来可能重新识别用户。隐私工程至少覆盖数据最小化、目的限制、保留期、访问控制、删除、导出、审计和派生数据传播。

先建立数据谱系：字段从哪里来，进入哪些日志、事件、缓存、特征、索引、训练集和备份。删除请求不是删一行数据库，而是沿谱系处理在线状态、异步副本和派生资产，并明确备份中的延迟删除策略。

图：一条行为数据的扩散

```
click ─┬→ request log ⟶ archive
       ├→ event bus ⟶ feature ⟶ model/train set
       ├→ served history/cache
       └→ experiment attribution
```

删除与保留策略必须覆盖每条边，而非只覆盖入口表。

日志应记录调查所需的最少信息。可使用稳定但受控的伪标识做关联；不要把 token、完整请求体、私信文本或精确位置作为“调试方便”长期保存。采样也不是隐私豁免。

6.5.11 11. 多租户的核心是隔离共享资源

租户可以是外部客户、内部产品、实验组、source 或优先级。多租户风险包括数据越界、资源争抢、指标混淆和控制面误操作。

数据访问必须把 tenant identity 放入服务端授权条件，不能只靠客户端查询过滤。缓存 key、幂等键、事件 partition 与指标 label 都应考虑租户命名空间。漏掉 tenant ID 的 cache key 会把 C 的结果返回给另一租户，这比普通 cache miss 严重得多。

资源隔离可组合配额、独立并发池、有界队列、加权公平调度和按租户 load shedding。总容量足够并不证明公平：一个慢租户占满连接池，其他租户仍会饥饿。

图：共享池与隔离池的爆炸半径

```
共享： T1 慢请求 ┐
       T2 正常请求 ├→ [同一 100 槽池] → 全体排队
       T3 正常请求 ┘

隔离： T1 → [20]  T2 → [40]  T3 → [40]
       配额可借用，但必须能在压力时收回。
```

6.5.12 12. 安全降级要比可用性降级更保守

当推荐 source 超时时，可以少返回 B；当 D 屏蔽或 E 安全服务超时时，不能把 unknown 当 safe。不同依赖的 fail-open/fail-closed 选择来自伤害模型。

可把决策写为：失败开放的用户伤害、失败关闭的可用性损失、持续时间、可逆性和检测能力。安全状态通常选择 fail-closed 或只展示已知安全集合；广告披露 F 缺失时也不应伪装为普通内容。

> **小心：加密不会修复错误授权**
>
> 磁盘和链路加密保护字节不被旁观者读取，但一个权限过大的服务仍能合法解密并泄露数据。安全评审必须覆盖调用主体、用途和字段级访问，而非止于“已启用 TLS”。

6.5.13 13. A–F 的恢复与安全账本

- A：实时内容依赖事件恢复顺序；重复 replay 不能重复曝光；作者删除必须传播到索引与缓存。
- B：Phoenix corpus、模型和 feature schema 要绑定版本；embedding 可能携带敏感属性，访问和保留需受控。
- C：缓存恢复允许短暂 miss，却不能跨 viewer/tenant 污染；served history 陈旧会造成重复推荐。
- D：屏蔽关系属于高优先级策略状态；恢复时宁可少展示，不能因旧 snapshot 重新泄露。
- E：安全判定 unknown 不是 safe；语料、规则与人工处置记录需要审计和保守回滚。
- F：广告预算、频控、披露和归因要求幂等；租户间预算与指标必须隔离。

> **源码证据：原仓中的边界线索**
>
> 从 `home-mixer/filters` 审查 D/E 的过滤位置，从 `home-mixer/side_effects/served_candidates_kafka_side_effect.rs` 识别曝光事件边界，从 `home-mixer/selectors` 观察 F

混排约束，从 `thunder` 追踪实时状态。公开快照可以支持边界审计，不能证明线上备份拓扑、权限策略、删除流程或灾备能力。

6.5.14 14. 反例：看起来安全、实际上没有证据

- “有三个副本”：三个副本共享错误配置、账号或区域。
- “每天备份成功”：从未 restore，密钥和格式可能早已失效。
- “服务已恢复 200”：D/E 尚未恢复，业务不变量仍破坏。
- “事件至少一次，不会丢”：重复副作用同样可能造成损失。
- “内部网络可信”：内部主体仍可能越权、被攻陷或误配置。
- “ID 已哈希”：低熵 ID 可枚举，行为组合仍可重新识别。
- “每租户都有字段”：查询、缓存或指标路径漏传租户仍会串数据。
- “删除主表即可”：缓存、日志、特征、训练集和备份仍有副本。

6.5.15 15. Game day 必须制造真实决策压力

演练不是朗读 runbook。先设定假设与停止条件，再注入一个受控故障：损坏部分 C、暂停 Thunder 事件、让 D snapshot 回退、使一个租户耗尽连接池，或撤销某个服务凭证。

参与者应从告警发现、建立事故时间线、判断影响、止损、恢复、校验到复盘完整走一遍。观察哪些权限拿不到、哪些 dashboard 无法切片、哪些命令不可逆、哪些 owner 无法联系。演练结束要产生可跟踪的修复项，而不是只有“演练成功”。

6.5.16 16. 完整性校验不能只比较行数

恢复前后一致的记录数量，可能同时包含错误重复和等量缺失。校验应从结构、引用和业务不变量逐层推进：文件 checksum 与日志连续；主键唯一、外键可达；状态机没有非法跃迁；聚合值能由明细重算；A–F 的候选与策略约束仍成立。

对大数据集可以分层抽样、按 shard 计算 digest，并对高风险对象做全量检查。digest 相等能证明选定字节相等，却不能证明选错了 snapshot；因此校验记录还要绑定数据版本、checkpoint、schema 和恢复任务 ID。

图：恢复校验的证据金字塔

```
          用户/策略不变量
        聚合可由明细重新推导
      引用完整、唯一键、状态机合法
    checksum、日志连续、版本/checkpoint

越向上越接近业务正确；越向下越适合自动全量执行。
```

6.5.17 17. 灾备环境也需要真实容量

冷备成本低，但 RTO 包含申请机器、恢复数据、预热索引和扩容下游；热备切换快，却需要持续复制、版本兼容和定期承载真实流量。若备用区域只有线上 20% 容量，切换计划必须定义优先流量、降级策略和 admission control。

灾备容量不能只算平均 QPS。事故期间重试、缓存冷启动、事件 replay、数据重建和人工查询会同时放大负载。先为恢复任务预留独立资源池，再限制 replay 速率，避免“为了追平积压而拖垮刚恢复的在线服务”。

6.5.18 18. 安全事件响应与可靠性事故有不同约束

可靠性事故通常优先恢复服务；安全事件还要防止攻击继续、保存取证并控制信息扩散。过早重启或清理机器可能销毁内存、日志和时间线；过早公开细节可能帮助攻击者扩大利用。

响应流程要区分 containment、eradication、recovery：先撤销凭证、隔离主体或关闭入口；再修补根因和搜寻持久化；最后从可信镜像与干净凭证恢复。所有操作使用独立审计通道，避免被已攻陷的系统篡改。

6.5.19 19. 审计日志本身也是受保护状态

审计记录谁在何时以哪个主体读取、修改、发布或恢复了什么。它需要追加语义、访问限制、完整性保护、可靠时间源、序列号或 append-only hash chain/签名，以及明确保留期；时间戳本身不能建立跨系统全序。普通应用日志不能替代审计：它可能被采样、删除、包含敏感内容或允许操作者自行覆盖。

同时避免把审计变成无限收集。记录资源标识、动作、决策和结果通常足够，不必复制完整私密载荷。对审计数据的查询本身也应留下审计记录。

6.5.20 20. 恢复与安全的共同控制面

二者都需要 inventory、owner、版本、审计、最小权限和可撤销动作。恢复工具本身权限极高：它能批量读取备份、覆盖状态、回放事件，必须比普通服务受到更严格的审批与审计。

把 runbook 命令分为只读诊断、可逆止损、状态变更和破坏性操作。默认先 dry-run，输出影响范围与 checkpoint；高风险动作要求第二人确认。恢复期间也不能为了速度关闭全部授权与审计，否则事故会变成数据泄露窗口。

> **新手 Tip：从一份最小恢复清单开始**
>
> 对自己的系统先选三种状态：用户屏蔽、served history、候选索引。分别写出 source of truth、备份方式、RPO/RTO、重建命令、校验不变量和 owner。然后真的删除本地副本并恢复一次；计时比想象更可信。

6.5.21 21. 自检：你恢复的是系统，还是幻觉

1. 为什么端口恢复和业务恢复之间还需要数据与不变量 gate?
2. RPO 相同为何对 C、D 和 F 产生完全不同的风险?
3. snapshot checkpoint 与 replay 起点不一致会怎样失败?
4. 为什么派生状态“可重建”仍需要容量和降级计划?
5. reconciliation 与在线 retry 的职责有什么不同?
6. caller identity、subject identity 和 request identity 为什么不能混用?
7. 幂等为何不能单独防止恶意重放?
8. 删除用户数据为何需要数据谱系?
9. 多租户 cache key 缺少 tenant 会造成什么类型的故障?
10. D/E 依赖失败时，为什么普通的 fail-open 策略不成立?
11. 一次合格的 restore/game day 应留下哪些可审计证据?
12. 为什么行数相等不能证明恢复前后状态等价?
13. 灾备切换时，哪些额外流量会使容量估算失真?
14. 安全事件为何不能总以“尽快重启”为第一动作?

检查点：把恢复能力写进交付物

你的系统应能从 snapshot 与增量事件恢复，报告 achieved recovery point/time 并与 RPO/RTO 目标比较，对 A—F 执行不变量校验；权限按主体与租户约束；日志具有隐私保留策略；慢租户无法耗尽全局资源；D/E 在依赖不确定时保持保守。所有实现证据并入下一工程章。

6.6 恢复、数据修复、安全与多租户

6.6.1 现场：服务起来了，状态却没有回来

进程重启成功不等于系统恢复。snapshot 可能不可用，增量日志可能缺口，派生缓存可能与 source of truth 不一致；故障注入接口若没有权限控制，还可能成为生产攻击面。可靠恢复与安全隔离都要求明确所有权和最小权限。

> **Tip：备份只有恢复成功后才算存在**
>
> 定期生成文件不证明可恢复。必须在隔离环境真正 restore、replay、校验不变量，测量 achieved recovery point/time，并分别与 RPO、RTO 目标比较。

6.6.2 在原仓定位：从数据流反推恢复与权限边界

回查 `home-mixer/side_effects/served_candidates_kafka_side_effect.rs`、`home-mixer/side_effects/update_served_history_side_effect.rs`、`thunder/posts/post_store.rs` 和 query 中 viewer/context 字段。标出：权威状态、派生状态、可重建状态、敏感数据、跨服务身份和审计事件。

公开快照没有完整备份、IAM 或灾备配置。本章使用原仓数据流确定“什么需要恢复和保护”，具体机制由读者实现。

6.6.3 原理与边界：恢复、修复和隔离是不同工作

RPO 描述可接受的数据丢失窗口，RTO 描述恢复服务所需时间。Restore 从备份建立状态，replay 应用增量事件，reconciliation 比较两个世界并修复偏差。派生缓存通常可重建，served history 和实验事件则可能有不同损失代价。

安全方面要保护身份传播、授权、重放、副作用、日志隐私和故障开关。多租户还需要配额、资源池和爆炸半径；一个 hot tenant 不应耗尽全部请求槽。

6.6.4 构造：完成一次可审计恢复

为 Thunder state、Served History 和 durable events 制作 snapshot 与增量日志。Snapshot 与日志共享一致切点或 LSN/watermark，并携带 manifest、checksum 和 schema/version。写入基线后生成 snapshot，再继续写事件并故意损坏部分状态。执行 restore、replay、reconciliation，按依赖顺序重建缓存并验证 A—F 不变量。

同时区分 end-user identity、service identity 和代表用户调用的 delegation；不能只信任一个 HTTP 字段，否则会产生 confused deputy。故障注入、管理、回放接口只允许管理角色，并记录审计日志。给两个逻辑租户分配独立配额或并发池。

> **验收契约：恢复不能扩大第二次事故**
>
> snapshot 可校验且版本兼容；replay-safe 通过幂等状态转换、去重或版本检查保证重复回放不重复业务效果；恢复不会绕过安全 filter；管理接口有认证授权与审计；敏感字段不进入普通日志；单租户过载不影响其他租户最低服务。

6.6.5 破坏并证明：恢复期间继续出错

注入损坏 snapshot、增量日志缺口、重复 replay、恢复到旧 schema、缓存未清理、无权限调用故障开关和租户 A 洪峰。记录哪些步骤自动停止，哪些需要人工选择，如何保持证据。

测量实际恢复点、数据损失窗口和恢复耗时，并与 RPO/RTO 目标比较；同时记录 restore duration、replay lag、reconciliation mismatch、unauthorized attempts、requests by tenant 和 rejected by quota。写一份恢复 runbook，包含停止条件与回滚。

6.6.6 验收：系统能够安全地回来

- 权威、派生和可重建状态分类完成；
- restore/replay/reconcile 都有自动化证据；
- achieved recovery point/time 已测量，并与 RPO/RTO 目标比较；
- 恢复路径遵守安全和版本契约；
- 管理能力最小权限并可审计；
- noisy neighbor 被资源隔离。

6.7 间章 P：一次发布不是一个结论——可信实验、灰度与可逆变更

大型 Tip · 间章：先证明比较有效，再决定是否扩大影响

推荐系统最危险的错觉之一，是“指标涨了，所以改动有效”。实验可能在分流、曝光、归因、统计或发布过程中失真；即使平均收益真实，也可能以安全、延迟或某类用户为代价。本间章把实验设计和发布控制连成一条证据链：从假设、随机化和事件契约，到 guardrail、灰度、kill switch、回滚与 game day。

6.7.1 1. 从可证伪假设开始

“新模型更好”无法直接检验。把它改成：对满足资格条件的用户，新 scorer 在七天内提高有效互动率，同时空 feed、D/E 违规、F 投诉率和 P99 不越过阈值。

实验规格至少包含实验单位、资格条件、处理、对照、主指标、guardrail、最小可检测效应、观察期、停止规则和决策 owner。先写规格再看数据，避免观察结果后更换指标或窗口。

6.7.2 2. 随机化单位决定干扰边界

按请求随机会让同一用户在控制和处理间跳动，served history、学习效应和缓存彼此污染；按用户随机通常更稳定。若社交关系、广告市场或共享库存使一个用户的处理影响另一个用户，则还要考虑 cluster randomization 或明确承认 interference。

图：随机化单位与污染

```
按请求： U — control — treatment — control
            └── history/cache/学习状态互相污染 ──┘

按用户： U1 ─────────── treatment
         U2 ─────────── control
```

稳定分桶应把 `hash(experiment_id, unit_id, allocation_version)` 映射到固定大空间 `[0, M)`，再用稳定区间分配 variant。扩流只移动区间边界，不应通过改变 `mod N` 让大量单位重分桶。Hash 版本、salt、namespace 和 allocation version 都要记录；不同实验复用同一桶并不自动独立。

6.7.3 3. SRM 是分流系统的烟雾报警器

Sample Ratio Mismatch（SRM）指观测到的组别人数显著偏离设计比例。设计 50/50，却得到 55/45，不应先解释业务效果，而应调查分桶、资格、日志丢失、机器人过滤、跨设备身份或处理引发的记录差异。

对于按独立单位做一次二元 assignment 的总样本 n、期望处理比例 p，处理组期望值为 np，二项近似标准差为 $\sqrt{np(1-p)}$。Cluster randomization、同一用户多请求等情况不能把事件数当独立 n。正式系统可使用卡方检验，但工程判断首先是：分流证据是否完整。

图：SRM 发生在指标解释之前

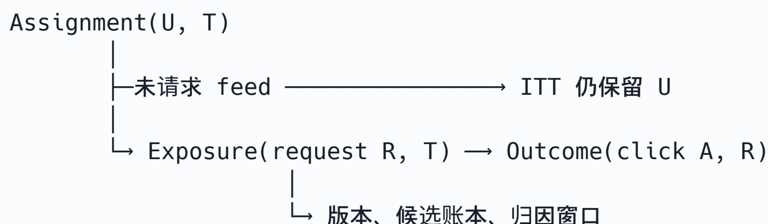
```
eligible 100k
     │ randomize 50/50
     ├─ control observed 55k
     └─ treatment observed 45k
             │
             └→ STOP：先查 assignment/exposure/logging
                不要计算“胜率”后继续发布
```

6.7.4 4. Assignment、Exposure 与 Outcome 是三类事件

Assignment 表示实验单位被分配；exposure 表示处理真正影响了决策；outcome 表示之后发生的结果。三者混为一个日志，会让资格、触发和归因无法审计。

Assignment event 应包含 experiment、unit、variant、hash/version、资格版本与时间；exposure event 还要绑定 request、实际 code/config/model 版本；outcome event 包含 event ID、subject、发生时间和可归因对象。

图：事件链而非一张结果表



6.7.5 5. ITT 与 Triggered 分析回答不同问题

Intention-to-Treat（ITT）对所有已经随机分配的合格单位按原 assignment 分析，即使部分人未真正触发处理。它保留随机化，回答“把策略部署给目标人群的总体效果”。

Triggered analysis 只分析满足预先定义触发条件的单位，通常更灵敏，回答“处理真正可作用时的效果”。危险在于用处理后的变量筛选：若新算法本身改变了谁会被记录为 exposed，就破坏组间可比性。

安全做法是预先定义对照、处理都可对称计算的 trigger，报告 ITT 与 triggered 两种结果，并解释它们的 estimand。不能因为 triggered 数字更漂亮就临时替换主结论。

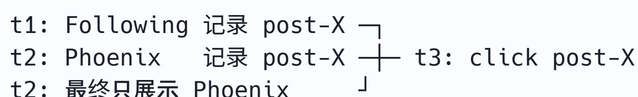
> **小心：只看 exposed 用户可能产生选择偏差**
>
> Treatment 超时导致一批困难请求没有 exposure 日志；过滤它们后，处理组只剩容易请求，指标自然上升。这不是算法收益，而是失败样本被观察规则删除。

6.7.6 6. 曝光—结果连接是一份数据契约

一次点击可能对应多个历史曝光。必须定义采用 last-touch、first-touch、最近一次有效曝光，还是请求级候选账本；窗口从 exposure 还是 event time 开始；重复事件如何去重；延迟和乱序如何处理。这些规则解决信用归属与数据完整性，不自动识别 treatment 的因果效果；因果识别仍来自随机化、estimand 和分析方法。

推荐实验最好把 outcome 关联到 request/candidate/exposure ID，而不是仅按 user+item+一天模糊连接。同一 `post-X` 在多个 source 出现时，还要记录最终被选中的归因来源，不能把同一次点击同时记给 Following 和 Phoenix。

图：模糊连接怎样重复归因



```
按 user+item 连接：两次收益
按 selected exposure ID：一次、归属于实际展示
```

事件去重键应反映业务事实。客户端重试产生的新上传 ID 不应重复计一次点击；相反，同一用户两次真实点击不能因键过粗而合并。

6.7.7 7. 指标需要分母、窗口与方向

`CTR = clicks / impressions` 仍不完整：哪些曝光进入分母，bot 是否剔除，延迟点击何时封窗，一次请求多候选怎样计数？每个指标应有名称、owner、事件来源、资格、分子、分母、去重、窗口、方向和版本。

比率指标不能随意平均每日比率；应根据决策目标选择总体 ratio、用户级平均或其他 estimand。重尾用户会主导事件级指标，因此同时观察用户级分布和分位数。

6.7.8 8. Guardrail 是发布约束，不是附录图表

主指标回答希望改善什么；guardrail 限制不能牺牲什么。推荐系统至少考虑延迟、错误、空 feed、候选多样性、隐藏/举报、F 广告投诉与收入异常。D/E 的确认穿透是硬 safety invariant，不是允许在统计阈值内消费的普通 guardrail；代理指标和测量噪声可以设阈值，确认违规应零容忍停止。

普通 Guardrail 应预先给出阈值和动作：超过 warning 暂停扩大，超过 critical 自动触发 kill switch。平均值不足以保护小群体，应按国家、设备、活跃度、新用户、source、版本和安全类别切片；但大量切片会引入多重比较和小样本噪声，确认性结论需要预注册、校正和足够样本，安全监控则可以采用更保守的停止规则。

```
图：平均收益不能覆盖硬边界

总体互动： +1.2%  ─────────────┐
新用户空 feed： +0.3pp          ├→ 决策不是简单相加
E safety violation： +0.02pp  STOP┘   硬 guardrail 优先
P99： +18ms  PAUSE
```

6.7.9 9. 多重比较与窥视会制造“显著”

同时查看 20 个独立无效指标，至少一个偶然越过普通阈值的概率会显著增加。每天反复查看并在最漂亮的一天停止，也改变了原检验的错误率。

教材不必推导完整统计学，但读者必须形成纪律：预注册主指标和停止规则；限制决策指标数量；报告效应大小与不确定区间，不只报告 p-value；探索性切片标记为探索，不冒充确认性结论；需要序贯决策时使用为此设计的方法。

6.7.10 10. 新奇效应、学习效应和网络效应

界面或排序变化上线初期会因新奇产生短期互动，之后回落；创作者和广告主会适应分发规则；模型与 served history 也会逐渐积累新状态。因此观察窗必须覆盖关键周期，并查看日序列而非只看累计值。

如果 Treatment 改变供给方行为，用户级随机化可能发生跨组干扰：处理组给创作者更多反馈，进而改变控制组也能看到的内容。此时结论应限制在当前实验设计可识别的范围，必要时使用地域、网络 cluster 或 switchback 实验。

6.7.11 11. A–F 实验账本

- A：实时内容可能只在特定活跃时段 triggered；延迟 outcome 要按 event time 处理。
- B：新 Phoenix 模型必须记录 model/corpus/feature 版本，并区分召回变化与排序变化。

- C：served history 与 cache 会造成跨请求 carryover，按请求随机容易污染。
- D：屏蔽与低质过滤是硬 guardrail；不能用互动收益抵消违规展示。
- E：安全策略通常不适合把已知风险随机暴露给对照，应采用离线回放、shadow 或保守边界实验。
- F：广告收入、预算、频控与投诉同时观察；归因事件必须幂等且按广告租户隔离。

> **源码证据：原仓中的实验与事件线索**
>
> 从 `home-mixer/side_effects/phoenix_experiments_side_effect.rs` 观察实验副作用边界，从 `home-mixer/side_effects/served_candidates_kafka_side_effect.rs` 观察曝光记录，从 `home-mixer/scorers/ranking_scorer.rs` 与 `home-mixer/selectors` 建立模型、排序和最终选择的候选账本。公开快照不能证明线上分桶实现、统计方法、指标口径或发布审批流程。

6.7.12 12. Shadow、Canary 与 A/B 不可互换

Shadow 把真实输入复制给新实现但不影响用户，适合比较输出、性能和错误；它无法测量真实用户行为，也可能遗漏写副作用。Canary 限制真实发布的初始流量，主要控制爆炸半径。A/B 实验通过随机对照估计因果效果。

一个改动可以依次经历离线回放、shadow、canary、A/B 和全面发布，但每阶段回答的问题不同。Shadow 输出一致不证明产品收益；A/B 收益显著不证明系统能承受 100% 容量。

6.7.13 13. 灰度阶梯必须绑定证据与等待时间

不要把 1%→10%→50%→100% 写成只有比例的流程。每一级应规定最小样本/时间、稳定性指标、业务指标、guardrail、版本一致性、on-call 确认与自动停止条件。

图：每一级都是 gate

```
0% shadow —[输出/容量]→ 1% —[错误/P99]→ 10%
  └─────────────────────────────────────┘
10% —[SRM/guardrail/切片]→ 50% —[峰值周期]→ 100%
      任一 gate 失败：暂停、kill switch 或 rollback
```

等待时间来自系统反馈周期：缓存 TTL、事件延迟、日周期、模型加载和下游账单，而不是固定“观察十分钟”。流量扩大还要检查 fan-out 和重试放大，避免低比例正常、全量后越过容量拐点。

6.7.14 14. Kill switch 必须独立、简单且演练过

Kill switch 用于快速停止危险行为：关闭新 source、恢复旧 scorer、禁止新格式写入、停止副作用或强制保守过滤。它应位于故障路径之外，有明确 owner、权限、默认值、传播时延和审计。

如果 kill switch 依赖正在故障的配置服务，或切换后仍要重启全部实例，它不是快速止损工具。定期验证开关在混合版本中都被识别，并测量从触发到全体节点生效的时间。

6.7.15 15. Rollback 与 forward fix 的选择

代码回滚不会撤销 Treatment 写出的事件、缓存和数据。决策前回答：旧版本能否读新状态；新 writer 是否已污染共享数据；模型/config/schema 是否也需回退；积压消息由谁消费；副作用能否补偿。

若新格式已经广泛写入、旧代码不兼容，盲目 rollback 会扩大事故，此时可能应先 kill 新行为，再 forward fix reader。发布计划必须在变更前写出这棵决策树。

图：回滚决策的最小树

```
异常发现
  ├─仅计算行为、无新状态 → rollback code/model
  └─已写共享状态
        ├─旧 reader 兼容 → stop writer + rollback
        └─旧 reader 不兼容 → kill behavior + forward fix
```

6.7.16 16. Pre-mortem：假设发布已经失败

发布前让团队假设“一周后发生严重事故”，各自写出最可能原因。按可能性、影响和可检测性排序，并把高风险项转成 guardrail、演练、kill switch 或拆分发布。

典型问题包括：SRM 被忽略；D/E 只看总体平均；新模型在某 feature 缺失时输出极值；F 预算副作用重复；cache key 造成回源洪峰；回滚读不了新事件。Pre-mortem 的价值是暴露沉默假设，不是生成长风险清单。

6.7.17 17. Game day：练的是决策链

在受控环境中注入 SRM、日志延迟、guardrail 恶化、配置部分传播、kill switch 失效或回滚不兼容。参与者应发现异常、停止灰度、保全证据、选择止损动作、处理已写状态并更新事故时间线。

演练评价不只是恢复速度，还包括：是否有人盯错平均指标；是否知道 assignment/exposure 差异；是否有权限操作开关；是否能按 code/config/model 切片；是否在压力下执行了不可逆命令。

6.7.18 18. 反例：数字正确，结论仍然错误

- 50/50 配置正确，却不检查观测到的 SRM。
- 只分析成功 exposure，删除 Treatment 超时请求。
- 点击按 user+item 模糊连接，同一 outcome 归给多个 source。
- 指标显著，却是看了二十个指标后挑出的一个。
- 总体收益为正，却让新用户、D/E 或某租户显著受损。
- Canary 一切正常，却没有经过峰值容量和完整 TTL 周期。
- Kill switch 写在文档里，从未在混合版本中演练。
- 回滚代码成功，却留下新 schema、事件和副作用。

> **新手 Tip：给每次改动做一张实验卡**
>
> 只用一页写：假设、unit、assignment、trigger、主指标、两个 guardrail、最短观察窗、SRM 检查、事件 ID、灰度阶梯、kill switch 和 rollback 条件。若一页写不清，代码通常也还没有清晰边界。

6.7.19 19. 自检：你是在测因果，还是在读故事

1. 为什么“新模型更好”不是可直接检验的假设？
2. 按请求随机怎样被 C 的 history/cache 污染？
3. SRM 为什么必须在效果解释之前处理？
4. Assignment、exposure 和 outcome 各证明什么？
5. ITT 与 triggered 分析的 estimand 有何不同？
6. 为什么按 Treatment 产生的 exposure 筛选会引入偏差？
7. A 同时来自两个 source 时，怎样避免重复归因？
8. 为什么 guardrail 必须按群体和版本切片？

9. Shadow、canary 和 A/B 分别回答什么问题?
10. 灰度等待时间为什么不能统一写成十分钟?
11. 什么情况下 forward fix 比代码 rollback 更安全?
12. Pre-mortem 与 game day 各自暴露哪类问题?

检查点：让一次发布成为可审计的推理

你的毕业项目应能保存 assignment—exposure—outcome 证据链，检查 SRM，同时报告 ITT 与预注册 triggered 结果；主指标与普通 guardrail 有明确口径和停止阈值，D/E 则作为不可用 error budget 抵扣的安全不变量，F 的预算与合规约束也要独立于增长指标执行；灰度每一级有容量和业务 gate；kill switch 已演练；rollback 计划覆盖代码、配置、模型、缓存、事件与已产生副作用。

6.8 毕业项目：像真正上线一样增加一个 Source

6.8.1 现场：最后一次改动必须经得起生产世界

目标是增加 `RecentEngagedAuthorsSource`：从近期互动作者中寻找新内容，在 out-of-network（OON，关注网络之外）供给不足时补充候选。它不是“多写一个类”，而是一次跨请求、状态、容量、事件、实验、版本和恢复的真实变化。

> **新手 Tip：先写可证伪假设**
>
> “上线新 source”不是实验假设。更好的表述是：对近期有作者互动且 OON 供给不足的请求，新 source 能降低结果不足率，同时不突破 P99、重复率、负反馈、安全和作者多样性 guardrail。

6.8.2 在原仓定位：一次变化经过三条链

沿三条真实链审查：

- 机制链：`candidate-pipeline/source.rs`、`home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs` 和现有 source；
- 参数链：`home-mixer/server.rs` 与 `home-mixer/scorers/ranking_scorer.rs` 的参数读取；
- 反馈链：`home-mixer/side_effects/served_candidates_kafka_side_effect.rs` 与 `home-mixer/side_effects/phoenix_experiments_side_effect.rs`。

公开快照缺少完整 params 定义和线上配置，所以原仓交付是证据化 RFC/patch plan；可运行实现进入读者自己的系统。

6.8.3 原理与边界：参数负责改变，实验负责判断

一次可信发布要连接 assignment、retrieved、selected、served、exposed 和 action。越靠左越接近机制，越靠右越接近用户结果。新 source 取到更多候选不等于产品成功。

实验需要稳定分桶、互斥、sample ratio 检查、事件 join、primary metric、mechanism metric 和 guardrail。离线 Recall/NDCG 提升不能替代线上因果判断；短期互动也不能自动代表长期生态。

6.8.4 构造：完成端到端工程交付

在自建系统实现新 source，固定返回 B/C：B 在 Retrieval 失败时恢复供给，C 因 served history 被删除。参数控制 enable、candidate limit、deadline、实验比例和 kill switch。

交付物必须包含：设计 RFC、实现、单元/契约/集成/故障/负载/混合版本测试、source 指标、trace、容量估算、事件 schema、实验计划、dashboard、runbook、灰度步骤、回滚和一页发布前 pre-mortem；game day 结束后再写 postmortem。

阶段	必须完成	过关证据
设计	目标人群、数据来源、契约、失败和隐私边界	RFC 能列出原仓扩展点与未知。
实现	source、参数、deadline、去重、过滤和 attribution	正常与 partial failure 集成测试。
证明	故障、负载、版本矩阵、事件幂等和恢复	测试报告、容量曲线和 replay 记录。
上线	分桶、指标、灰度、kill switch、回滚和 runbook	演练时间线与恢复后的用户层验证。

复盘	发布前 pre-mortem；演练后记录真实放大机制、检测缺口和长期修复	owner、期限和对应回归测试。

验收契约：毕业不是跑通，而是可交付

新 source 失败只损失本路；D/E 安全规则不变；deadline 和 admission control 不被绕过；事件身份稳定、重复可去重、归因窗口明确且 join 完整率可测；v1/v2 可混跑；kill switch 能在残留缓存和积压事件存在时恢复旧行为。

6.8.5 破坏并证明：在发布前主动失败

至少演练：source timeout、重复候选、下游过载、参数部分发布、事件重复/延迟、缓存污染、worker 崩溃、实验组共用错误 cache key、发布 50% 后回滚、恢复期间旧事件重放。

容量计算使用条件概率或保守上界：入口 QPS × `P(进入实验且满足 enable)` × 每请求调用数 × 平均尝试次数；只有实验 gate 与 enable gate 的嵌套/独立关系明确时，才能直接相乘两个比例。双机房灾备、shadow、重试和峰值余量分别列为显式放大项。每个失败都要留下测试或演练证据。

Tip：毕业项目卡住时先缩输入，不删语义

可以把候选固定为 A—F、把节点缩成四个本地进程、把流量缩成几十 QPS；不要删除 deadline、幂等、版本、恢复或指标。规模可以小，分布式问题不能被改没。

6.8.6 验收：你已经学会保护一次变化

- 能从原仓准确找到扩展点和证据边界；
- 自建实现完成推荐、分布式、测试和运营闭环；
- 正常、过载、崩溃、版本混合和回滚均有证据；
- 实验指标区分机制与用户结果；
- 容量、SLO、权限、恢复和数据质量进入上线门槛；
- 能清楚说明哪些结论来自原仓，哪些来自自己的实现，哪些仍需生产验证。

7 结语：成为会保护系统的人

熟手并不是记住最多组件的人。看到一个远程调用，他会追问等待上限、取消和重试；看到一份状态，他会追问所有权、顺序、持久化和恢复；看到一次发布，他会追问混合版本、爆炸半径和回滚；看到一个成功响应，他还会追问用户是否真的得到足够、新鲜、安全的结果。

`x-algorithm` 为我们提供真实工程现场，却不是一份可以在公开环境完整启动的生产系统。你亲手构造的项目也不是 X 的复刻。二者共同训练的是一套可迁移的方法：从证据出发，明确边界，主动破坏，留下证明，并为未知保持敬畏。

8 附录：源码、能力与验收回查

附录提供源码锚点、分布式 Definition of Done、项目里程碑和新手排障顺序。它只用于回查，不引入第二条阅读路线。

8.1 附录：能力、源码与项目回查

8.1.1 间章插入索引

间章不是第二套作业。首次阅读只选两三道自检题，所有实现证据并入紧随其后的工程章。

间章	核心模型	进入的工程章
A	系统模型、状态所有权、故障与证据	01 请求、02 Pipeline
B	RPC、fan-out、deadline、取消与重试	03 Sources、04 Deadlines
C	数据契约、缺失、新鲜度与资格边界	05 Hydration、06 Filtering
D	向量召回、ANN、负采样与 Recall	07 Retrieval
E	排序、校准、NDCG、反馈和实验	08 Ranking、09 Blending
F	事件身份、顺序、状态机与 checkpoint	10 Thunder
G	缓存模式、会话一致性、失效和热点	11 Cache/History
H	事务、outbox/inbox、消息和补偿	13 Side Effects
I	排队论、尾延迟、背压和过载	14 Overload
J	分片、热点、ownership 与迁移	15 Sharding
K	复制、quorum、共识、epoch 和 fencing	16 Replication
L	Schema、配置、模型与混合版本	17 Evolution
M	Oracle、history、故障和模型测试	18 Testing
N	遥测、SLI/SLO、容量与事故	19 Operations
O	恢复、灾备、安全、隐私与多租户	20 Recovery/Security
P	随机化、归因、灰度、回滚与复盘	21 Capstone

8.1.2 二十二个线性里程碑

章	系统变化	必须留下的证据
00	项目与证据簿	原仓边界、请求日志、第一条测试。
01	入口与上下文	request/attempt/version 贯穿 trace。
02	Candidate Pipeline	阶段顺序、候选账本、drop reason。
03	并行 Sources	部分失败、来源归因、fan-out 预算。
04	Deadline 与取消	剩余预算传播、任务清理、重试放大。
05	数据 Hydration	字段血缘、缺失语义、ID 对齐测试。
06	Filter Chain	硬边界、顺序、fail-open/closed。
07	Retrieval	归一化、top-K、Recall 和切片。
08	Ranking	多目标贡献、isolation、版本。
09	Blending	间距、冲突、供给不足和稳定性。
10	Thunder-like Source	乱序、幂等、真实重启和 retention。
11	Cache/History	TTL、single-flight、read-your-writes。
12	DAG Executor	拓扑、资源池、队列和重试预算。

13	Durable Events	enqueue/ack crash window、幂等、隔离。
14	Overload Control	饱和曲线、bulkhead、load shedding。
15	Sharding Migration	movement、热点、dual-read/write。
16	Replication	lag、failover、epoch、fencing。
17	Online Evolution	v1/v2 矩阵、config skew、rollback。
18	Test Portfolio	风险—证据矩阵和组合事故回归。
19	Operations	SLO、容量、告警、事故与复盘。
20	Recovery/Security	restore/replay、RPO/RTO、权限与隔离。
21	Capstone	实现、测试、容量、实验、灰度与恢复闭环。

8.1.3 分布式 Definition of Done

每次新增远程调用、异步任务或共享状态，都要回答：

1. **正确性**：要保护的业务不变量是什么？
2. **时间**：等待上限、deadline、取消和时间来源是什么？
3. **失败**：部分失败、重复、乱序、超时和重试如何处理？
4. **资源**：并发、队列、连接、内存和下游预算是否有界？
5. **状态**：谁是 source of truth，什么持久化，什么可重建？
6. **演化**：新旧代码、schema、配置、模型和缓存能否共存？
7. **隔离**：故障域、租户和最大爆炸半径是什么？
8. **证据**：测试、日志、指标和 trace 各自证明什么？
9. **恢复**：自动恢复、重放、补偿和人工修复怎样执行？
10. **用户影响**：协议成功时，feed 是否仍足够、新鲜、安全？

如果一个设计评审只回答“正常情况下怎样工作”，它还没有完成一半。

8.1.4 A 级源码锚点

路径	正文	高价值原因
`home-mixer/server.rs`	1/4/17/21	入口、QueryBuilder、参数、版本和调用边界。
`candidate-pipeline/candidate_pipeline.rs`	2/3/4/18	全局控制流、并发、阶段、side effects 和观测。
`home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs`	1—9/21	帖子候选的真实组件装配。
`home-mixer/candidate_pipeline/for_you_candidate_pipeline.rs`	1/9	帖子与其他 item 的外层混排。
`home-mixer/models/query.rs`	1/5	请求上下文、字段责任与路径开关。
`home-mixer/models/candidate.rs`	1/5—9	候选生命周期、身份和模型转换。
`phoenix/recsys_retrieval_model.py`	7	two-tower、归一化和 top-K。
`phoenix/recsys_model.py`	8	多行为 ranking 输入输出。
`phoenix/grok.py`	8/18	candidate-isolation mask 与 transformer。
`home-mixer/scorers/ranking_scorer.rs`	8/17/21	多目标分数、参数与最终字段。

home-mixer/ads/safe_gap_blender.rs	9	广告位置与敏感内容间距。
thunder/posts/post_store.rs	10/20	实时状态、tombstone、去重和 retention。
thunder/thunder_service.rs	10/14/19	查询、并发边界和负载脱落。
grox/engine.py	12/14	任务入口、异步调度和容量风险。
grox/plans/plan.py	12/18	任务依赖图和失败传播。
home-mixer/side_effects/served_candidates_kafka_side_effect.rs	13/17/20/21	服务事件、反馈闭环和 schema。
home-mixer/side_effects/update_served_history_side_effect.rs	11/13/15	展示历史、未来请求和分片写入。

完整 A/B/C 分级和全部公开文件索引见 `code-atlas/`。正文深读 A 级和代表性 B 级；C 级用于审计覆盖，不要求逐文件背诵。

8.1.5 故障注入词典

故障	优先观察	常见错误
delay/timeout	deadline、queue age、pending task	只在调用方返回，底层继续运行。
error/drop	partial result、error kind、fallback	吞异常并伪装成空列表。
duplicate	幂等键、业务效果、dedupe retention	只去重消息，不保护副作用。
reorder	版本、tombstone、状态转移	用到达顺序冒充业务顺序。
crash	持久化边界、pending、replay	同进程异常代替真实重启。
overload	active、queue、P99、rejected	无界排队或无限重试。
partition	双 leader、lag、availability	把连接失败等同于节点死亡。
clock skew	TTL、lease、event time	用 wall clock 计算经过时间。
mixed version	unknown field、cache key、rollback	只测试全量升级后的世界。
corruption	checksum、restore、reconcile	服务启动就宣布恢复完成。

8.1.6 新手排障顺序

> **Tip：先找第一处偏离**
>
> 用户只看到一条内容时，不要直接跳到“模型坏了”。按下面顺序找第一处偏离基线，再解释它如何向后传播。

```
1. 用户影响、surface、时间范围和版本
2. 入口参数、实验分桶、request/trace ID
3. 各 source enabled/error/latency/candidate count
4. hydration missing/alignment/field age
5. filter kept/removed by reason
6. retrieval/ranking score、model/config/corpus version
7. selection/blending 后 size、type 和硬约束
8. cache age、served history、event backlog
```

```
9. retry amplification、queue、连接、replica lag
10. 止血、回滚、恢复证据和回归测试
```

8.1.7 课程完成自检

完成课程时，你应能在不翻答案的情况下：

- 画出 For You 内外两层请求、数据和失败边界；
- 实现并证明 deadline 传播、取消和有限重试；
- 解释 retrieval、ranking、filtering、selection 和 blending 各自不能替代什么；
- 用状态机处理重复、乱序、tombstone、checkpoint 和 replay；
- 找到系统饱和拐点并用 Little's Law 核对容量；
- 设计 shard migration、replica failover 和 fencing；
- 让 v1/v2、旧缓存和新事件在滚动发布中共存；
- 为一次事故完成止血、恢复、验证和无责复盘；
- 从 snapshot 与日志恢复状态并测量 RPO/RTO；
- 为新 source 交付实现、测试、指标、容量、实验、灰度、回滚和安全审查。

做到这些，才算从"会写一个后端"走向"会保护一个分布式系统"。

8.1.8 Tips：经验较少时怎样不迷路

> **Tip：一次只增加一个未知**
>
> 新协议、新存储、新并发模型和新算法不要在同一提交首次出现。先用固定数据证明语义，再替换其中一层。这样失败时仍能定位变量。

> **Tip：每章结束就提交**
>
> 用一个小而完整的 commit 保存通过验收的里程碑。下一章故障把系统改坏时，你可以比较行为和证据，而不是凭记忆找差异。

> **Tip：错误也属于接口**
>
> 不要只设计成功返回值。为 timeout、overload、not-found、invalid、stale 和 dependency error 建立稳定分类，调用方才能选择重试、降级或阻断。

> **Tip：先写表，再写并发代码**
>
> 状态机写转移表，版本兼容写矩阵，故障写 crash-window，容量写资源图。并发代码只是这些决策的实现，不是设计本身。

> **Tip：不要用 sleep 证明并发**
>
> 用 barrier、event、fake clock 和可控下游决定任务何时前进。固定等待既慢又会在不同机器上随机失败。

Tip：先保留原始证据

事故发生时先保存 trace、版本、队列、配置和时间线，再重启或清缓存。止血重要，但没有证据就很难防止同类事故再次发生。

Tip：工具不是学习目标

标准库实现不代表生产应手写消息队列或监控系统。它让你看见确认点、队列和状态；理解语义后，再把实现替换成真实基础设施。

Tip：不知道就标 Unknown

原仓没有公开的参数、部署或基础设施协议，不需要猜一个看似专业的答案。把未知写进设计，说明上线前如何取证，比错误确定更可靠。