读懂 x-algorithm

推荐系统工程导读

沿着真实项目讲清推荐系统、算法模型、异步服务和分布式工程

Typst 源码入口：`docs/recsys-beginner-book/main.typ`

目录

1 前言：读这本书的路线

这本书现在按“从一次 For You 请求读完整个 x-algorithm 项目”的顺序组织。每个一级章回答一个稳定问题：这一层为什么存在、吃什么输入、产出什么结果、依赖哪些服务和模型、失败后怎样降级。分布式、异步和算法知识都放回具体代码场景里讲，服务于读懂项目，而不是抢走主线。

1.1 这本书怎么学

这本书把当前仓库当成教材，而不是把推荐算法讲成一组孤立公式。读者的起点可以很低：不需要先系统学完分布式系统、Rust 或 JAX；本书会在读到相关代码时补上必要的异步调用、服务依赖、模型结构和生产工程知识。

推荐系统第一次看起来复杂，通常不是因为每个概念都难，而是因为很多层同时出现：模型、特征、缓存、Kafka、RPC、过滤、排序、监控、实验、延迟预算。新手容易误以为必须先学完整个分布式系统才能入门。更好的方式是先抓住一条主线：

> **新手视角：一条主线**
>
> 一个用户打开 For You 页面，系统先把“这个用户是谁、最近喜欢什么、哪些内容不能看”补齐；再从多个地方拿到候选内容；再补充候选内容的元信息；再过滤掉不合适的内容；再用模型打分；最后选择一批内容返回。

本书围绕仓库里的四个目录展开：

目录	它在系统里的角色	你应该带走的理解
`candidate-pipeline/`	通用推荐流水线框架	推荐系统不是一段单独的模型调用，而是一串可组合阶段。
`home-mixer/`	For You 请求编排层	真实业务把帖子、广告、关注推荐、缓存和可见性规则混在一个响应里。
`phoenix/`	召回与排序模型示例	召回先缩小范围，排序再精排；两者追求的目标不同。
`thunder/`	实时站内内容候选源	“你关注的人刚发了什么”需要低延迟、近实时的数据结构。
`grox/`	内容理解任务管线	推荐不只预测兴趣，也需要理解内容安全、摘要、嵌入和标签。

1.1.1 读法

第一遍不需要追求每个类型和每个 trait 都看懂。你只需要能回答三个问题：

- 当前阶段输入是什么？
- 当前阶段输出是什么？
- 为什么这个阶段可以并行，或者必须顺序执行？

第二遍再补模型细节，例如 embedding、two-tower、transformer、candidate isolation、multi-action prediction。第三遍再看生产系统问题，例如缓存、监控、副作用、实验、Kafka、RPC。

> **提示：技术知识跟着项目走**

> 异步、RPC、缓存、Kafka、two-tower、transformer、实验和监控都不是单独的知识清单。它们会在项目代码需要它们时出现：source 调外部服务时讲异步和超时，Phoenix 模型里讲召回和排序，side effect 里讲日志、缓存和数据闭环。

1.1.2 学完后的目标

读完本书，你应该能从 `CandidatePipeline::execute` 出发，解释一次推荐请求如何穿过候选源、hydrator、filter、scorer、selector 和 side effect；也能解释 Phoenix 为什么分成 retrieval 和 ranking；还能看懂这个项目里哪些代码负责算法效果，哪些代码负责系统可靠性和延迟。

2 读代码前的地图

本章先建立全局坐标。新手读推荐系统最容易在模型、服务、缓存和实验之间来回跳；所以开头只做两件事：看清项目分层，并把后续学习路线放回这张地图。

2.1 先画地图，再读代码

推荐系统的代码不能从模型文件开始读。模型只回答“这个候选内容有多可能触发某些行为”，但系统还要回答“候选内容从哪里来、哪些内容不能展示、哪些副作用要记录、响应里还要混入什么”。所以第一站是项目地图。

2.1.1 一次请求的鸟瞰图

```
用户打开 For You
    |
    v
home-mixer/server.rs
    |
    v
ForYouCandidatePipeline
    |-- ScoredPostsSource  -> ScoredPostsServer
    |                         |
    |                         v
    |                    PhoenixCandidatePipeline
    |                         |-- ThunderSource
    |                         |-- PhoenixSource / PhoenixTopicsSource / PhoenixMOESource
    |                         |-- TweetMixerSource / CachedPostsSource
    |
    |-- AdsSource
    |-- WhoToFollowSource
    |-- PromptsSource
    |-- PushToHomeSource
    v
BlenderSelector
    |
    v
Feed response
```

这里有两个层次。内层 `PhoenixCandidatePipeline` 主要产出“已打分的帖子候选”。外层 `ForYouCandidatePipeline` 把帖子、广告、关注推荐、提示等 feed item 混合起来，形成最终 feed。

> **检查点：不要把 Phoenix 等同于整个推荐系统**
>
> Phoenix 是模型组件；推荐系统还包括候选源、过滤、缓存、可见性、安全、日志和实验。只懂模型，通常无法解释线上推荐为什么慢、为什么空、为什么某类内容被过滤。

2.1.2 代码入口

当前仓库的 README 已经说明 For You 系统由 Home Mixer、Thunder、Phoenix、Candidate Pipeline 组成。本书把这几个入口连起来：

入口	读它时要问的问题
`candidate-pipeline/candidate_pipeline.rs`	通用流水线怎样定义阶段？哪些阶段并行？哪些阶段顺序？

`home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs`	For You 的帖子候选实际配置了哪些 hydrator、source、filter、scorer?
`home-mixer/candidate_pipeline/for_you_candidate_pipeline.rs`	最终 feed 除了帖子，还混入了哪些业务 item?
`phoenix/recsys_retrieval_model.py`	召回模型怎样把用户和候选映射到同一个向量空间?
`phoenix/recsys_model.py`	排序模型接收哪些输入？怎样输出多种行为的预测?
`thunder/posts/post_store.rs`	近实时帖子如何保存在内存结构里，供 in-network 召回使用?
`grox/engine.py`	内容理解任务如何异步调度和执行?

2.1.3 推荐系统里的几种“数据”

新手读代码时，最容易把所有数据都叫“特征”。为了降低混乱，本书先区分五类数据：

Query 数据：和本次请求、当前用户有关。例如 user id、IP、关注列表、近期行为序列、已经曝光过的内容。

Candidate 数据：和某个候选内容有关。例如 post id、作者、文本、媒体、语言、视频时长、是否来自关注网络。

Model 输入：被整理成模型能吃的张量、hash、embedding、action 序列、product surface。

Decision 数据：模型打分之后，业务层用于过滤、排序、混排、可见性判断的数据。

Side-effect 数据：响应之后还要写出去的数据。例如曝光日志、缓存、统计、Kafka 事件。

这五类数据会在后续章节反复出现。你读一个函数时，先判断它正在处理哪类数据，理解速度会快很多。

2.1.4 为什么项目会分这么多阶段

一个简单推荐 demo 可能只有：

```
candidates = retrieve(user)
scores = model(user, candidates)
return top_k(candidates, scores)
```

真实系统不能这么写，原因有三点。

第一，候选源很多。关注网络、全局语义召回、主题召回、缓存、广告和运营内容并不来自同一处。

第二，延迟很紧。能并行的外部请求必须并行，否则一次请求会被几十个远程调用串起来拖慢。

第三，失败是常态。某个 hydrator 或候选源失败时，系统经常需要降级，而不是让整个 feed 空掉。

`candidate-pipeline` 的意义就在这里：把“推荐请求的常见骨架”抽出来，让具体业务只配置组件。

2.2 学习路线、练习和术语表

这一节把前面内容整理成一套可以执行的学习计划。你可以按周推进，也可以把每一节当成代码阅读 checklist。

2.2.1 七天入门路线

天	主题	完成标准

1	项目地图	能说出 `candidate-pipeline`、`home-mixer`、`phoenix`、`thunder`、`grox` 各自角色。
2	流水线阶段	能不看代码复述 `execute` 的阶段顺序。
3	并行与顺序	能解释为什么 source/hydrator 常并行，filter/scorer 常顺序。
4	Phoenix Retrieval	能画出 user tower、candidate tower、dot product 检索。
5	Phoenix Ranking	能解释 `RecsysBatch`、candidate isolation、多行为预测。
6	服务化组件	能解释 Thunder、Grox、side effects 为什么不是“模型细节”但仍影响推荐。
7	端到端复盘	从用户打开 feed 到返回结果，完整讲一遍数据流。

2.2.2 代码阅读 checklist

读任何一个组件时，按这个顺序写笔记：

- 组件名是什么？属于 source、hydrator、filter、scorer、selector 还是 side effect?
- 输入类型是什么？只读 query，还是会读 candidates?
- 输出写到哪里？更新 query、更新 candidate、返回 removed、写外部系统?
- 它是否 async？如果 async，等待的是模型服务、存储、Kafka、RPC，还是后台任务?
- 失败时发生什么？返回空、默认值、错误，还是跳过?
- 它对用户体验的影响是什么？相关性、安全、延迟、多样性、去重、商业化、观测性?

2.2.3 推荐算法术语表

Candidate：候选内容。进入排序前，它只是“可能展示”的对象，不代表最终会展示。

Query Hydrator：补齐本次请求上下文的组件，例如用户历史、关注列表、屏蔽列表。

Candidate Hydrator：补齐候选内容信息的组件，例如 core data、作者资料、媒体、语言、安全标签。

Source：产生候选的组件。一个 source 可能来自实时内存库、向量召回、缓存、广告系统或其他服务。

Filter：删除不合格候选的组件。过滤不只是安全，也包括去重、年龄、可见性、订阅资格、已曝光等。

Scorer：给候选写入分数或排序信号的组件。模型调用和加权组合都可以是 scorer。

Selector：从打分后的候选里选择最终返回的一批，可能包含排序、截断、混排和多样性规则。

Side effect：不直接改变当前返回内容，但会写日志、缓存、统计或实验数据的操作。

Retrieval：召回。目标是从大规模候选中快速找出一批可能相关的内容。

Ranking：排序。目标是对召回后的较小候选集做更精细的预测和排序。

Embedding：把离散对象或行为表示成向量，让模型能计算相似度或进行神经网络运算。

Candidate isolation：排序 transformer 中候选不能互相 attention，只能看用户、历史和自己。

Backfill / degradation：某个依赖失败时，用缓存、默认值或较弱策略继续服务，避免整个请求失败。

2.2.4 三个进阶项目

1. 给 `PhoenixCandidatePipeline` 画一张真实组件图。把 query hydrator、source、hydrator、filter、scorer、side effect 全部列出来，并标记哪些可能访问外部服务。
2. 写一个最小候选流水线伪代码。只实现两个 source、两个 filter、一个 scorer、一个 selector，用数组模拟候选流动。
3. 对 Phoenix 排序输出做一次手算。假设某候选 `favorite=0.2`、`reply=0.03`、`not_interested=0.01`，给定三个权重，算出加权分，并解释负反馈为什么要进入公式。

2.2.5 继续扩写本书的方向

这版书稿先建立“能读懂项目”的主线。后续可以继续补：

- 逐文件导读：每章带读一个真实 source、filter、hydrator、scorer。
- 实验系统：feature switch、decider、参数权重、A/B 分析。
- 可观测性：tracing span、stats receiver、空结果率、延迟分桶。
- 数据闭环：曝光日志、用户行为、训练样本、模型更新。
- 本地实验：使用 `phoenix/run_pipeline.py` 跑一次 retrieval -> ranking demo。

> **检查点：真正的掌握标准**
>
> 不是能背出所有组件名，而是能拿到任意一个新组件，判断它处在流水线哪一段、依赖什么数据、失败会影响什么、怎样验证它是否按预期工作。

3 Candidate Pipeline：请求主干和阶段边界

学完地图之后，主线进入框架层。这里的重点不是 Rust 语法，也不是某个异步关键字，而是一次推荐请求如何被拆成可并行、可顺序、可观测、可降级的阶段。理解这一章，后面所有 source、hydrator、filter、scorer 都会有位置。

3.1 服务化推荐的分布式最小模型

这一节不是把异步编程当成主角，而是解释为什么这个项目不能只按“本地函数调用”来读。推荐请求会依赖用户行为服务、社交图、帖子元数据服务、模型服务、Kafka、Redis、可见性服务等外部系统。所谓线上推荐系统，不只是模型公式，而是一张服务依赖图。

3.1.1 一次外部调用背后的五件事

看到下面这行，不要只把它翻译成语法上的“等待”：

```
let response = client.get_in_network_posts(request).await?;
```

它背后至少有五个工程问题：

- 请求发给谁？是本进程、同机服务、同机房服务，还是跨机房服务？
- 最坏要等多久？有没有超时、重试、fallback？
- 失败时当前请求应该失败，还是跳过这个结果继续？
- 返回的数据是否可信？是否可能为空、过期、部分缺失？
- 这个等待是否阻塞其他独立工作？

推荐系统的复杂度就在这里。一个用户刷新 feed，服务端可能同时访问几十个外部依赖。它们每个都可能慢、空、失败、返回旧数据。新手入门时不要急着背 RPC 框架名，先把这五个问题问清楚。

3.1.2 串行等待和并行等待

先看串行等待：

```
let a = call_a().await;
let b = call_b().await;
let c = call_c().await;
```

如果每个调用 50ms，总耗时接近 150ms。再看并行等待：

```
let futures = vec![call_a(), call_b(), call_c()];
let results = join_all(futures).await;
```

如果三个调用互不依赖，总耗时接近最慢的那个，例如 50ms 到 70ms。`candidate-pipeline` 在 query hydrator、source、candidate hydrator 中大量使用 `join_all`，就是为了把独立的外部等待并行化。

> **提示：并行不是免费午餐**
>
> 并行会降低一次请求的等待时间，但会增加下游系统的瞬时压力。如果每个请求都同时打十几个服务，下游容量、限流、缓存命中率都会变成设计问题。

3.1.3 依赖关系决定能否并行

能并行的典型例子：

- 获取关注列表和获取屏蔽列表通常都只依赖 user id。
- Thunder source 和 Phoenix source 都读取已补齐的 query，并各自返回候选。
- 多个候选 hydrator 都读取同一批 candidates，并各自补不同字段。

不能随便并行的典型例子：

- dependent query hydrator 必须等第一批 query hydrator 完成。
- filter 必须顺序缩小候选集合。
- scorer 可能依赖前一个 scorer 写入的字段，例如先写 Phoenix 预测，再计算最终分。

判断标准很朴素：后一步是否需要前一步的输出。如果需要，就顺序；如果只读同一份输入并写不同字段，才可能并行。

3.1.4 失败不是异常情况，而是常态

`Source::run` 的默认实现会把错误记录下来，然后返回 `Err`。但 `CandidatePipeline::fetch_candidates` 收集结果时用了：

```
for mut candidates in results.into_iter().flatten() {
    collected.append(&mut candidates);
}
```

这意味着失败的 source 不会贡献候选，但成功的 source 仍然能继续。这个设计体现了推荐系统常见的降级思路：某个召回源挂了，不一定让整个 feed 挂掉。

hydrator 的设计也类似。某个候选的某个字段补不出来，可能只是这个字段保持默认值，后续由 filter 判断是否移除。这样可以避免一个候选或一个字段拖垮整批请求。

> **小心：降级要被观测**
>
> 降级不是忽略错误。系统可以继续返回结果，但必须记录错误率、空结果率、缓存命中率、过滤率和延迟。否则线上质量下降时很难定位。

3.1.5 后台任务和当前响应

`run_side_effects` 使用 `tokio::spawn`：

```
tokio::spawn(async move {
    let futures = side_effects.iter().map(|se| se.run(input.clone()));
    let _ = join_all(futures).await;
});
```

这说明 side effect 不阻塞当前返回。用户不必等曝光日志写完、Kafka 发送完、缓存更新完才看到 feed。但这也带来另一个问题：后台失败不会自然反馈给当前请求，所以必须靠指标和日志发现。

这类代码体现的是服务化系统的边界划分。你会看到系统把工作分成两类：

- 必须完成才能返回：例如核心候选、必要过滤、排序。
- 可以返回后继续：例如日志、统计、某些缓存写入。

3.1.6 延迟预算

线上推荐有一个隐含约束：用户等待时间有限。假设一次请求目标是 300ms，不能把它平均分给所有阶段，因为最慢的尾部请求会拖垮体验。一个简单预算可以写成：

阶段	预算	主要风险
Query hydration	40ms	社交图、用户行为服务慢或超时。
Sources	80ms	某个召回源慢，导致候选不足或整体等待。
Candidate hydration	70ms	批量元数据服务慢、缓存命中低。

Scoring	80ms	模型服务延迟、候选过多。
Selection + post filters	20ms	规则过多、排序数据结构低效。
余量	10ms	序列化、调度、线程切换、网络抖动。

真实系统不会这么简单，但新手可以先用这种表理解“为什么召回不能无限多、为什么 hydration 要缓存、为什么 side effect 要后台化”。这些知识都是为了读懂项目的服务边界和成本结构。

3.1.7 本节练习

1. 在 `candidate-pipeline/candidate_pipeline.rs` 里找出 source、hydrator、scorer 和 side effect 访问外部系统的位置，区分单个调用和 fan-out 调用。
2. 任选一个 source，写下它失败时当前 pipeline 会发生什么。
3. 任选一个 side effect，解释它为什么不应该阻塞当前响应，以及它失败后应该通过什么指标被发现。

3.2 CandidatePipeline，推荐请求的主干

这一节只盯一个文件：`candidate-pipeline/candidate_pipeline.rs`。它是本项目最适合新手入门的地方，因为它把推荐请求拆成了清楚的阶段。

3.2.1 `execute` 是整条路的目录

`CandidatePipeline::execute` 的主流程可以直接搜索 `async fn execute` 定位。简化后是这样：

```
async fn execute(&self, query: Q) -> PipelineResult<Q, C> {
    let hydrated_query = self.hydrate_query(query).await;
    let hydrated_query = self.hydrate_dependent_query(hydrated_query).await;

    let candidates = self.fetch_candidates(&hydrated_query).await;
    let hydrated_candidates = self.hydrate(&hydrated_query, candidates).await;

    let (kept_candidates, mut filtered_candidates) =
        self.filter(&hydrated_query, hydrated_candidates.clone());

    let scored_candidates = self.score(&hydrated_query, kept_candidates).await;
    let selected_candidates = self.select(&hydrated_query, scored_candidates);

    let post_selection_hydrated_candidates =
        self.hydrate_post_selection(&hydrated_query, selected_candidates).await;

    let (mut final_candidates, post_selection_filtered_candidates) =
        self.filter_post_selection(&hydrated_query, post_selection_hydrated_candidates);

    self.run_side_effects(input);
    PipelineResult { ... }
}
```

这个函数的价值不是语法，而是顺序。推荐系统把一次请求拆成：

阶段	输入和输出	新手理解
Query hydration	query -> 更完整的 query	先补用户上下文。没有上下文，后面不知道该召回什么。
Source	query -> candidates	从多个地方拿候选内容。

Candidate hydration	candidates –> 更完整的 candidates	补内容元信息，例如作者、媒体、语言、安全标签。
Filter	candidates –> kept + removed	先删掉明显不能展示的内容，减少后续模型成本。
Score	candidates –> scored candidates	调用模型或规则，得到排序分数。
Selector	scored candidates –> selected + rest	排序、截断、混排。
Post-selection	selected –> final	最终可见性检查和响应前补充。
Side effect	final + rest –> 异步写日志/缓存	返回响应不一定要等所有写入完成。

新手视角：先看阶段，不急着看泛型

`Q` 可以先理解成 query 类型，`C` 可以先理解成 candidate 类型。trait、泛型和 boxed dyn 是为了让同一个流水线框架服务不同业务；第一次读时，不需要把 Rust 类型系统全部展开。

3.2.2 哪些地方并行

你会在这个文件里反复看到 `join_all`。它的意思是：把一组异步任务同时发出去，然后等它们都返回。

例如 `hydrate_query` 会筛选启用的 query hydrator，然后把每个 hydrator 的 `run` 变成 future：

```
let hydrate_futures = hydrators.iter().map(|h| h.run(&query));
let results = join_all(hydrate_futures).await;
```

`fetch_candidates` 对 source 做同样的事：

```
let source_futures = sources.iter().map(|s| s.run(query));
let results = join_all(source_futures).await;
```

这里要带走的重点是阶段依赖：query hydrator 和 source 可以 fan-out 并行，filter 和 scorer 通常按顺序收敛。异步语法只是实现手段，真正影响系统行为的是依赖关系、候选数量和失败策略。

提示：判断能否并行的标准

如果组件之间只读同一份 query，彼此不依赖，就适合并行。例如多个 source 同时取候选。若后一个组件必须基于前一个组件的输出，就必须顺序。例如 filter 会逐个缩小候选集。

3.2.3 哪些地方顺序

过滤器是顺序执行的。`run_filters` 会把候选列表交给第一个 filter，拿到 kept 和 removed；再把 kept 交给下一个 filter。核心结构是：

```
for filter in filters.iter().filter(|f| f.enable(query)) {
    let result = filter.run(query, candidates);
    candidates = result.kept;
    all_removed.extend(result.removed);
}
```

为什么不并行？因为过滤器之间会改变候选集合。去重之后，年龄过滤面对的输入已经变少；可见性过滤可能依赖 hydration 结果。如果并行跑所有过滤器，就要处理“同一个候选被多个过滤器同时移除”的归因问题，也很难记录每个过滤器到底删了多少。

scorer 也是顺序执行的。`score` 会依次运行启用的 scorer，并把结果更新回 candidates：

```
for scorer in all.iter().filter(|s| s.enable(query)) {
    let scored = scorer.run(query, &candidates).await;
    scorer.update_all(&mut candidates, scored);
}
```

这说明 scorer 不一定只有模型。一个 scorer 可以写入 Phoenix 的预测；另一个 scorer 可以把多种预测加权成一个最终分；再一个 scorer 可以做作者多样性衰减。

3.2.4 Home Mixer 怎样配置这条流水线

抽象框架本身不决定业务。真正的业务配置在 `home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs`。

在 `build_with_clients` 里，帖子候选流水线配置了：

- query hydrators：补 scoring sequence、retrieval sequence、关注列表、屏蔽列表、IP、话题、订阅关系等。
- sources：`ThunderSource`、`TweetMixerSource`、`PhoenixSource`、`PhoenixTopicsSource`、`PhoenixMOESource`、`CachedPostsSource`。
- hydrators：补 in-network 标记、core data、quote、视频时长、媒体、订阅、作者资料、语言等。
- filters：去重、年龄、自发帖、转帖去重、订阅资格、已看、已服务、静音关键词、社交图关系、视频、话题等。
- scorers：`PhoenixScorer`、`RankingScorer`、`VMRanker`。
- post-selection hydrators 和 filters：可见性、安全、广告 brand safety、会话去重等。
- side effects：实验、Kafka、Redis 缓存、统计、请求缓存等。

这份配置是读推荐系统的宝藏：它告诉你线上 feed 并不只是“拿候选、模型打分、top k”。真正的系统要在算法效果、产品规则、安全合规和工程延迟之间折中。

3.2.5 本节练习

1. 打开 `candidate-pipeline/candidate_pipeline.rs`，只读 `execute`，手写一遍阶段名。
2. 在 `hydrate_query`、`fetch_candidates`、`run_hydrators` 中找出 `join_all`，写下它们并行的对象分别是什么。
3. 在 `home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs` 中数一数 `sources` 有几个；想一想如果其中一个 source 失败，系统为什么不应该直接让整个 feed 失败。

3.3 请求生命周期，从 gRPC 到 Pipeline

这一节跟踪一次 scored posts 请求如何进入 Home Mixer，怎样变成 `ScoredPostsQuery`，再交给 pipeline 执行。

3.3.1 入口：get_scored_posts

`home-mixer/server.rs` 中实现了 gRPC 方法 `get_scored_posts`。它做的第一件事是从 metadata 提取 tracing 信息：

```
let b3_info = extract_b3_info(request.metadata());
```

然后调用 `query_builder.build(...)`，把 proto query 转成内部 query。

这说明请求入口不应该直接进入算法。它先要做校验、trace、上下文构造和参数求值。

3.3.2 QueryBuilder::build

`QueryBuilder::build` 的关键步骤：

1. 检查 viewer_id。
2. 如果是 trace user，强制采样。
3. 获取 viewer data。
4. 判断是否 `in_network_only`。
5. evaluate feature switches。
6. 生成 prediction_id 和 request_id。
7. 构造 `ScoredPostsQuery`。
8. 创建 root span。

这一步把外部请求转换成内部系统能处理的上下文。

3.3.3 Viewer data timeout

`fetch_viewer_data` 使用 timeout：

```
tokio::time::timeout(
    Duration::from_millis(VIEWER_ROLES_TIMEOUT_MS),
    self.gizmoduck_client.get_viewer_data(viewer_id),
).await
```

超时或错误会返回默认 ViewerData。这是入口阶段的降级。它保护请求延迟，但也意味着角色、muted keywords、follower count 等信息可能缺失。

> 这里再次体现默认值风险：入口降级后，后续系统仍能跑，但个性化和过滤可能变弱。

3.3.4 Feature switches

`evaluate_feature_switches` 用 user id、country、language、client app、datacenter、账号年龄、手机号状态和 user roles 构造 recipient。然后匹配 feature switches，并应用 overrides。

> 也就是说参数不是全局常量，而是按用户、地区、客户端和实验动态求值。

3.3.5 ScoredPostsServer::run_pipeline

`home-mixer/scored_posts_server.rs` 的 `run_pipeline` 是进入候选 pipeline 的地方：

```
let pipeline_result = self.phoenix_candidate_pipeline.execute(query).await;
```

执行前它会处理 test user，记录请求信息；执行后记录响应统计，并把 `PostCandidate` 转成 `ScoredPost`。

> 这一步是算法 pipeline 和服务 API 的边界：

- pipeline 输出内部候选。
- server 负责日志、响应转换和 debug JSON。

3.3.6 Debug JSON

`build_debug_json` 把 query、retrieved_candidates、filtered_candidates、selected_candidates 和 stats 序列化出来。这对学习和排查很有用，因为它展示了候选在各阶段的状态。

> 一个好的 debug 输出应该能回答：

- query 最终补了哪些字段?
- source 总共拿到多少候选?
- 哪些候选被过滤?
- 最终选中了哪些候选?

- 各阶段数量是否合理?

3.3.7 请求生命周期图

```
gRPC request
  -> extract tracing metadata
  -> QueryBuilder::build
      -> fetch viewer data
      -> evaluate feature switches
      -> construct ScoredPostsQuery
      -> root span
  -> ScoredPostsServer::run_pipeline
      -> PhoenixCandidatePipeline::execute
      -> candidates_to_scored_posts
  -> gRPC response
  -> side effects continue in background
```

3.3.8 本节练习

1. 在 `QueryBuilder::build` 中标出所有可能影响 query 的外部输入。
2. 解释 viewer data timeout 后为什么不能直接失败整个请求。
3. 打开 `build_debug_json`，说明它对排查空结果有什么帮助。
4. 画出 `ScoredPostsQuery` 从 proto query 到 hydrated query 的变化过程。

3.4 数据模型剖析，Query 和 Candidate

前面的章节按阶段讲。这一节换一个角度：看两个核心数据结构 `ScoredPostsQuery` 和 `PostCandidate`。它们是推荐流水线中最重要的“运输容器”。

3.4.1 ScoredPostsQuery：一次请求的上下文

`home-mixer/models/query.rs` 中的 `ScoredPostsQuery` 很长，因为它承载了整次请求的上下文。可以分成几类：

类别	字段例子
请求身份	`user_id`、`client_app_id`、`request_id`、`prediction_id`
地域和客户端	`country_code`、`language_code`、`ip_address`、`user_agent`、`client_version`
请求模式	`in_network_only`、`is_bottom_request`、`is_top_request`、`is_preview`、`is_polling`
用户历史	`scoring_sequence`、`retrieval_sequence`、`served_history`
候选排除	`seen_ids`、`served_ids`、`topic_ids`、`excluded_topic_ids`、`exclude_videos`
用户特征	`user_features`、`user_demographics`、`subscription_level`
控制面	`params`、`decider`
缓存	`cached_posts`、`has_cached_posts`

这个结构说明 query 不是一个简单请求对象，而是一个逐步被 hydrator 填满的上下文。

3.4.2 Query 的构造

`ScoredPostsQuery::new` 负责把 proto query、viewer data、feature switch、设备状态等合成初始 query。很多字段一开始是默认值：

```
scoring_sequence: None,
retrieval_sequence: None,
cached_posts: vec![],
has_cached_posts: false,
served_history: vec![],
```

后续 query hydrator 会逐步填这些字段。读 query 时要区分：

- 请求入口已经知道的字段。
- viewer data 带来的字段。
- feature switch/decider 控制字段。
- hydrator 后补字段。

3.4.3 PostCandidate：候选的生命周期

`home-mixer/models/candidate.rs` 中的 `PostCandidate` 也很长。它从 source 阶段开始是瘦对象，经过 hydration、filter、scoring 后变胖。

可以按生命周期分类：

阶段	字段例子
Source 初始字段	`tweet_id`、`author_id`、`served_type`、reply/retweet ids
Core hydration	`tweet_text`、`retweeted_user_id`、`quoted_tweet_id`
Media/user hydration	`min_video_duration_ms`、`has_media`、`author_screen_name`
Safety hydration	`visibility_reason`、`safety_labels`、`brand_safety_verdict`
Social hydration	`author_blocks_viewer`、`mutual_follow_jaccard`
Scoring	`phoenix_scores`、`weighted_score`、`score`、`prediction_request_id`
Response/log	`ancestors`、`tweet_type_metrics`、`following_replied_user_ids`

这就是为什么 hydrator 和 scorer 要保持顺序和字段边界。一个字段写错，会影响多个后续阶段。

3.4.4 CandidateHelpers

`CandidateHelpers` 提供几个语义方法：

```
fn get_original_tweet_id(&self) -> u64 {
    self.retweeted_tweet_id.unwrap_or(self.tweet_id)
}

fn get_original_author_id(&self) -> u64 {
    self.retweeted_user_id.unwrap_or(self.author_id)
}
```

这解决 retweet 场景：展示 tweet 和原始 tweet 不是同一个 id。模型请求、去重、日志可能需要原始 tweet id，而 UI 展示又需要当前 tweet id。

`as_tweet_info` 会把 PostCandidate 转成模型服务需要的 `TweetInfo`，包括原始 tweet、retweeting tweet、quote、reply、视频时长、计数、语言和 bool features。

3.4.5 默认值语义

数据模型里有很多 `Option`，也有一些普通字段默认 0 或空字符串。读代码时要特别注意：

- `author_id=0` 在当前 filter 语境中代表候选缺少有效作者 id，会被当作完整性失败处理。
- `score=None` 在 selector 里是负无穷。

- `visibility_reason=None` 在 VFFilter 中表示不 drop。
- `has_media=None` 在 TweetInfo 中会转成 false。
- `retweeted_tweet_id=None` 表示不是 retweet。

默认值不是实现细节，而是业务语义。

3.4.6 从 Candidate 到 ScoredPost

`home-mixer/scored_posts_server.rs` 的 `candidates_to_scored_posts` 把内部候选转成响应对象。它会把 Option 转成 proto 默认值：

```
score: candidate.score.unwrap_or(0.0) as f32,
in_network: candidate.in_network.unwrap_or(false),
prediction_request_id: candidate.prediction_request_id.unwrap_or(0),
```

这一步是内部语义到外部 API 的边界。内部可以用 Option 表示缺失，响应通常需要具体默认值。

3.4.7 本节练习

1. 给 `PostCandidate` 的字段按 source/hydrator/filter/scorer/response 分类。
2. 找出三个 `Option` 字段，说明 None 在下游代表什么。
3. 解释为什么 retweet 需要 `get_original_tweet_id` 和 `tweet_id` 两套语义。
4. 对比 `PostCandidate` 和 `ScoredPost`，写出哪些字段在响应时被默认化。

4 Phoenix：从召回到精排

有了流水线骨架，下一步看模型在系统里具体提供什么能力。Phoenix 不是整套推荐系统本身，它负责把用户历史和候选内容变成召回向量、行为预测和可排序信号。

4.1 Phoenix Retrieval，先从海量内容里捞一小桶

排序模型再强，也不能对全站每一条内容逐条打分。召回阶段的任务是把“几百万甚至更多候选”缩小成“几百或几千个值得精排的候选”。本项目里，Phoenix Retrieval 用 two-tower 思路做这件事。

4.1.1 召回要解决什么问题

想象你要给用户推荐帖子。最直接的方法是：

```
for post in all_posts:
    score = expensive_model(user, post)
return top_k(score)
```

这在小 demo 里成立，在线上系统里不可用。候选太多，模型太贵，延迟太紧。召回阶段因此追求一个不同目标：不要一开始就精确排序，只要快速找出一批“可能相关”的内容。

> **新手视角：召回和排序的分工**
>
> 召回像先从图书馆里找到 200 本可能相关的书；排序像再认真读摘要、目录和上下文，把最适合你的 20 本排到前面。

4.1.2 Two-tower 的基本形状

Phoenix Retrieval 的关键文件是 `phoenix/recsys_retrieval_model.py`。里面的 `PhoenixRetrievalModel` 把系统拆成两座塔：

塔	输入	输出
User Tower	用户 hash、历史行为、product surface 等	用户向量 `u`
Candidate Tower	帖子和作者 embedding	候选向量 `v`

两边输出都被放到同一个向量空间里。召回时只需要算相似度：

```
score(user, item) = dot(user_vector, item_vector)
```

当向量都做过归一化，点积越大，方向越接近，系统就越认为这个候选和用户当前兴趣相近。

4.1.3 为什么候选塔可以提前算

two-tower 的工程优势在于：候选塔只依赖帖子和作者，不依赖某个具体用户。于是候选内容的向量可以提前离线或准实时算好，放进向量索引里。在线请求来时，只需要：

1. 用 User Tower 算当前用户向量。
2. 在候选向量索引里查 top K。
3. 把这些候选交给 ranking 阶段。

这就是召回能处理大规模语料的核心原因。它把“每个用户对每个帖子跑大模型”变成“用户向量查索引”。

4.1.4 读 `CandidateTower`

`CandidateTower` 在 `phoenix/recsys_retrieval_model.py` 中。搜索类名即可定位。它接收 post + author 的 embedding，并投影到共享空间：

```
hidden = jnp.dot(post_author_embedding.astype(proj_1.dtype), proj_1)
hidden = jax.nn.silu(hidden)
candidate_embeddings = jnp.dot(hidden.astype(proj_2.dtype), proj_2)

candidate_norm_sq = jnp.sum(candidate_embeddings**2, axis=-1, keepdims=True)
candidate_norm = jnp.sqrt(jnp.maximum(candidate_norm_sq, EPS))
candidate_representation = candidate_embeddings / candidate_norm
```

这段代码可以分成三层理解：

- `post_author_embedding`：把内容和作者信息拼起来。
- 两层投影加激活：把原始 embedding 变成适合检索的向量。
- L2 normalization：把向量长度归一，方便用点积比较方向相似度。

> **小心：不要把向量检索当成最终答案**
>
> 召回分数只负责“先捞出来”。它通常不包含所有业务约束，也不适合直接决定最终顺序。最终展示还要经过 hydration、filter、ranking、selector 和可见性检查。

4.1.5 Retrieval sequence 和 scoring sequence

在 `home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs` 中，query hydrator 里同时出现了 `ScoringSequenceQueryHydrator` 和 `RetrievalSequenceQueryHydrator`。这说明召回和排序可能使用不同形态的用户历史。

新手可以这样理解：

- retrieval sequence 更关心“用什么历史快速表达用户兴趣，以便找候选”。
- scoring sequence 更关心“精排模型要看哪些上下文，以便预测多种行为”。

它们都来自用户行为，但服务于不同模型阶段。

4.1.6 召回阶段的常见误区

误区一：召回只要越多越好。实际不是。召回太少会漏掉好内容；召回太多会拖慢 hydration 和 ranking，还会把噪声交给后面。

误区二：召回模型越复杂越好。召回阶段通常更重视吞吐、索引效率和稳定性；复杂模型如果不能高效预计算候选向量，可能线上不可用。

误区三：召回能替代过滤。不能。召回模型只负责相关性，屏蔽、静音、可见性、安全、重复内容等仍然要由后续阶段处理。

4.1.7 本节练习

1. 打开 `phoenix/recsys_retrieval_model.py`，找到 `CandidateTower`，标出“拼接、投影、归一化”三步。
2. 在 `home-mixer/sources/phoenix_source.rs` 里观察 Phoenix source 怎样调用 retrieval client。
3. 用一句话解释：为什么召回阶段适合用 ANN 或向量索引，而不是对所有候选逐个精排。

4.2 Phoenix Ranking，给候选内容做精排

召回阶段已经把范围缩小。Ranking 阶段的任务是更认真地看用户上下文和候选内容，预测用户可能做什么，然后把这些预测变成最终排序分。

4.2.1 Ranking 输入长什么样

关键类型在 `phoenix/recsys_model.py` 的 `RecsysBatch`。它包含：

- `user_hashes`：用户标识的 hash。
- `history_post_hashes`、`history_author_hashes`：用户历史行为涉及的帖子和作者。
- `history_actions`：历史行为类型，例如喜欢、回复、转发、停留。
- `history_product_surface`：行为发生在哪个产品场景。
- `candidate_post_hashes`、`candidate_author_hashes`：本次要打分的候选。
- `candidate_product_surface`、候选时间信息、可选 IP hash 等。

这不是“把文本直接塞进模型”的接口，而是把用户、历史、候选和行为整理成张量。`RecsysEmbeddings` 则保存已经查表得到的 embedding。

4.2.2 三段序列拼成一个模型输入

`PhoenixModel.build_inputs` 做了关键拼接；在 `phoenix/recsys_model.py` 中搜索函数名即可定位：

```
embeddings = jnp.concatenate(
    [user_embeddings, history_embeddings, candidate_embeddings], axis=1
)
padding_mask = jnp.concatenate(
    [user_padding_mask, history_padding_mask, candidate_padding_mask], axis=1
)
candidate_start_offset = user_padding_mask.shape[1] + history_padding_mask.shape[1]
```

模型看到的序列大致是：

```
[ User ][ History 1 ][ History 2 ] ... [ Candidate 1 ][ Candidate 2 ] ...
```

`candidate_start_offset` 记录候选从哪里开始。这个位置很重要，因为 attention mask 会用它区分“用户和历史”和“候选”。

4.2.3 Candidate isolation：候选之间不能互相偷看

在普通 transformer 里，一个 token 可能通过 attention 看到其他 token。推荐排序里，这会带来一个问题：如果 Candidate A 的分数会受 Candidate B 是否在同一批里影响，那么 A 的分数就不稳定，也不容易缓存或解释。

本项目在 `phoenix/grok.py` 的 `make_recsys_attn_mask` 里解决这个问题：

```
causal_mask = jnp.tril(jnp.ones((1, 1, seq_len, seq_len), dtype=dtype))
attn_mask = causal_mask.at[:, :, candidate_start_offset:, candidate_start_offset:].set(0)
candidate_indices = jnp.arange(candidate_start_offset, seq_len)
attn_mask = attn_mask.at[:, :, candidate_indices, candidate_indices].set(1)
```

含义是：

候选可以看：用户、历史、自己
候选不能看：其他候选

> **提示：为什么这对线上系统重要**

如果候选之间能互相 attention，同一个帖子放在不同候选批次里可能得到不同分数。Candidate isolation 让每个候选的分数主要由“用户上下文 + 它自己”决定，排序更稳定。

4.2.4 输出不是一个分数，而是一组行为预测

`PhoenixModel.__call__` 最后取出候选位置的输出，并映射成多种行为的 logits 和 continuous prediction：

```
candidate_embeddings = out_embeddings[:, candidate_start_offset:, :]
logits = jnp.dot(candidate_embeddings.astype(unembeddings.dtype), unembeddings)
continuous_preds = jax.nn.sigmoid(continuous_logits).astype(self.fprop_dtype)
```

这意味着模型不会只说“相关”或“不相关”。它会预测多个行为，例如 favorite、reply、repost、click、dwell、follow author，以及一些负反馈。

真正的最终分在 Rust 侧继续计算。`home-mixer/scorers/phoenix_scorer.rs` 负责调用 Phoenix prediction client，把预测写回候选；`home-mixer/scorers/ranking_scorer.rs` 再按权重组合这些预测。

简化公式是：

```
final_score = sum(weight_i * P(action_i))
```

其中正反馈权重可以提高分数，负反馈权重可以压低分数。`RankingScorer` 还会做作者多样性衰减，避免同一个作者连续占据太多位置。

4.2.5 为什么要多行为预测

如果只预测“用户会不会点赞”，系统会偏向容易点赞的内容，却可能忽略停留、回复、分享、关注作者等目标；也可能忽略“不感兴趣、屏蔽、举报”这类负反馈。

多行为预测让产品可以调权重：某个阶段更重视深度互动，另一个阶段更重视负反馈规避，或者对视频内容单独调整 video quality view 的权重。

小心：权重不是随便调的旋钮

权重改变会影响内容生态和用户体验。工程上它只是参数，产品和算法上它代表目标函数的变化。调权重前要有实验、监控和回滚方案。

4.2.6 本节练习

1. 打开 `phoenix/grok.py`，用自己的话解释 `candidate_start_offset` 为什么存在。
2. 打开 `home-mixer/scorers/ranking_scorer.rs`，列出三个正反馈字段和三个负反馈字段。
3. 想一想：如果候选之间可以互相 attention，为什么同一条帖子在不同请求批次里可能分数不同？

4.3 Phoenix 本地实战，从 artifacts 跑完整链路

前面讲 Phoenix retrieval 和 ranking 的模型结构。这一节看 `phoenix/run_pipeline.py`，它是本仓库最适合作为本地实验入口的文件，因为它把 retrieval -> ranking 串成一个可运行脚本。

4.3.1 这个脚本解决什么

`run_pipeline.py` 的 docstring 写得很清楚：它加载导出的 checkpoint、预计算 corpus、用户行为序列，然后执行：

1. Retrieval：编码用户历史，与 corpus 向量做点积，取 top K。
2. Ranking：对 top K 候选运行 engagement model。

3. Display：打印按加权 engagement 排序的结果。

这和线上系统的关系是：它不包含 Home Mixer 的所有 hydrator、filter、side effect，但能帮助你理解 Phoenix 模型本身如何从用户历史走到候选排名。

4.3.2 artifacts 结构

脚本期望的目录大致是：

```
artifacts/
  retrieval/
    model_params.npz
    embedding_tables.npz
    config.json
  ranker/
    model_params.npz
    embedding_tables.npz
    config.json
  sports_corpus.npz
  example_sequence.json
```

这些文件分别对应模型参数、embedding table、模型 config、候选 corpus 和用户行为序列。

新手要注意：本地 pipeline 不是训练脚本，而是 inference demo。它用已经导出的参数和 corpus 跑一遍前向推理。

4.3.3 hash 函数

`build_hash_functions` 根据 config 里的 hash 参数构造 user/item/author hash。模型不直接拿原始 id 做 embedding lookup，而是把 id 映射到 hash bucket。

这有两个意义：

- embedding table 大小可控。
- 不同实体类型可以共享一个统一 embedding table 的不同区间。

代码中 `build_unified_emb_table` 把 user、item、author 三张表拼到一个大表里，并预留 padding 区间。

4.3.4 加载模型

`load_model_params` 在 `phoenix/runners.py` 中。它把 `.npz` 里的 key 拆成 Haiku 参数树：

```
parts = key.split("/")
module_path = "/".join(parts[:-1])
param_name = parts[-1]
params.setdefault(module_path, {})[param_name] = jnp.array(data[key])
```

`load_embedding_table` 则读取 embedding table。理解这两步后，你就知道 JAX/Haiku 模型推理需要两类东西：

- params：神经网络参数。
- embeddings：输入 id 查表后的向量。

4.3.5 构造用户历史

脚本从 `example_sequence.json` 中读取：

- `user_id`
- `history`
- 每条历史里的 `post_id`

- `author_id`
- `actions`

然后填入固定长度数组：

```
history_post_ids = np.zeros(hist_len, dtype=np.uint64)
history_author_ids = np.zeros(hist_len, dtype=np.uint64)
history_actions = np.zeros((hist_len, num_actions), dtype=np.float32)
```

真实系统中的 query hydrator 做的事情更复杂，但概念相同：把用户历史整理成模型能读的张量。

4.3.6 Retrieval 前向

脚本构造 retrieval batch 和 embedding batch 后，调用 retrieval model 得到 `user_repr`：

```
user_repr = ret_fn.apply(ret_params, batch, emb_batch, dummy_gn, dummy_gn)
scores = corpus_repr @ np.asarray(user_repr[0])
```

这里 `corpus_repr` 是预计算候选向量。`@` 是矩阵乘法，等价于一次性计算用户向量和所有候选向量的点积。然后通过 `argpartition` 取 top K。

这正是 two-tower 召回的工程优势：候选向量提前算好，在线只算用户向量和相似度。

4.3.7 Ranking 前向

retrieval 返回 top K 后，脚本按 `candidate_seq_len` 分批送入 ranker：

```
for i in range(0, TOP_K, cand_len):
    ...
    out = rank_fn.apply(rank_params, rb, re)
    probs = jax.nn.sigmoid(out.logits)
```

ranker 输出 logits，脚本用 sigmoid 变成概率。然后用一个简单权重组合：

```
weighted = fav * 1.0 + reply * 0.5 + rt * 0.3 + dwell * 0.2
```

线上 `RankingScorer` 的权重更完整，也会包含负反馈、多样性、OON 调整。本地脚本的目标是演示链路，不是复制全部线上排序策略。

4.3.8 读输出

脚本打印：

- retrieval score
- favorite/reply/retweet/dwell/VQV 概率
- weighted score
- topic
- post URL

看输出时，不要只盯最终 rank。你应该比较 retrieval score 和 ranking weighted score：retrieval 认为相近的内容，ranking 不一定排最高，因为 ranking 看的是更细的多行为预测。

4.3.9 本节练习

1. 改 `--top_k_retrieval`，观察 ranking 输入候选数量如何变化。
2. 改 `--top_k_display`，确认它只影响打印数量，不影响模型推理。
3. 在本地权重公式中加入 reply 或 dwell 的更高权重，观察排序会偏向哪类候选。
4. 对比本地脚本和 `PhoenixSource` + `PhoenixScorer`：一个是离线 demo，一个是线上服务调用。

4.4 Phoenix 内部，embedding、mask 和输出头

前面讲了 Phoenix 的 retrieval/ranking 分工。本节更细看模型内部的几个关键概念：hash embedding、输入拼接、attention mask、输出头。

4.4.1 Hash embedding

Phoenix 不直接为每个 user id、tweet id、author id 建独立 embedding。它使用多组 hash，把实体映射到 embedding table。好处是：

- 控制 embedding table 大小。
- 处理大量稀疏 id。
- 允许导出固定大小的表。

代价是 hash collision。多个实体可能共享 bucket。多 hash 和模型训练可以缓解，但不能完全消除。

4.4.2 User、history、candidate 三段输入

`PhoenixModel.build_inputs` 把三段拼起来：

```
[ user ][ history actions ... ][ candidates ... ]
```

每一段都来自不同 reduce 函数：

- `block_user_reduce`
- `block_history_reduce`
- `block_candidate_reduce`

这些函数把多个 hash embedding、行为 embedding、product surface、时间 bucket 等压成统一维度。

4.4.3 Product surface

`history_product_surface` 和 `candidate_product_surface` 告诉模型行为或候选发生在哪个产品场景。相同用户行为在不同 surface 上含义可能不同，例如 Home、搜索、视频页、通知入口。

> 新手可以把 product surface 理解成“上下文标签”，让模型知道这次预测发生在哪个产品场景。

4.4.4 Post age bucket

排序模型会计算候选年龄 bucket：

```
post_age_minutes = (impr_ts_sec - post_creation_ts_sec) // 60
bucket = (post_age_minutes // granularity_mins) + 1
```

帖子年龄影响推荐。新鲜内容和旧内容的互动模式不同。用 bucket 而不是原始秒数，可以让模型更稳定地学习时间段效果。

4.4.5 Candidate isolation mask

`make_recsys_attn_mask` 是 Phoenix ranking 最关键的工程设计之一。它让候选能看用户和历史，也能看自己，但不能看其他候选。

```
candidate_i -> user/history: allowed
candidate_i -> candidate_i: allowed
candidate_i -> candidate_j: blocked
```

这样同一候选的分数不会因为批次里放了哪些其他候选而变化太大。

4.4.6 输出头

排序模型输出两类结果：

- discrete logits：多种离散行为，例如 favorite、reply、repost。

- continuous preds：连续行为，例如 dwell time。

代码上是取 candidate 位置的 transformer 输出，再乘以 unembedding matrix：

```
candidate_embeddings = out_embeddings[:, candidate_start_offset:, :]
logits = jnp.dot(candidate_embeddings, unembeddings)
```

这和语言模型的“hidden state –> vocab logits”很像，只是这里的 vocab 换成了 action types。

4.4.7 Retrieval user tower 和 ranking 的关系

Retrieval user tower 使用类似 transformer 结构来编码用户历史，但目标不同：它输出 user representation，用来和候选向量点积。Ranking 则输出每个候选的行为预测。

可以这样记：

- Retrieval 输出一个用户向量。
- Ranking 输出每个候选的一组行为预测。

4.4.8 本节练习

1. 解释为什么 candidate isolation 能提高排序分数稳定性。
2. 找出 `RecsysBatch` 中哪些字段属于 user、history、candidate。
3. 说明 post age bucket 为什么比直接使用秒数更适合作为模型输入。
4. 对比 retrieval 输出和 ranking 输出的形状和用途。

4.5 召回和排序实验课

这一节给出几个不依赖完整线上环境的思考实验。它们帮助你理解 top K、权重、多样性和负反馈如何改变结果。

4.5.1 实验一：召回 top K

假设 corpus 有 100000 条内容，retrieval 返回 top 200。Ranking 只能处理这 200 条。即使某条内容在 ranking 模型下会排第 1，如果 retrieval 没召回它，它也不会出现。

这说明召回质量决定上限，排序质量决定最终顺序。

练习：把 top K 从 200 改成 50，会发生什么？

- Ranking 成本下降。
- 漏召回风险上升。
- 后续 hydration 和 filter 成本下降。
- 如果 filter 移除率高，最终候选可能不足。

4.5.2 实验二：过滤对召回深度的影响

假设 retrieval top 200 中：

- 20 条重复。
- 30 条太旧。
- 15 条被屏蔽或静音。
- 25 条可见性不通过。

剩下只有 110 条。此时 selector 的 top K 再高也没用，因为候选已经被过滤掉。

所以 source 的 max_results 要考虑过滤率。高过滤率场景需要更深召回，或提升 source 质量。

4.5.3 实验三：多行为权重

假设两个候选：

```
A: favorite=0.20, reply=0.02, dwell=0.30, not_interested=0.01
B: favorite=0.12, reply=0.08, dwell=0.45, not_interested=0.03
```

如果 favorite 权重高，A 可能排前。 如果 reply/dwell 权重高，B 可能排前。 如果 not_interested 权重很负，B 可能被压下去。

这说明“好内容”不是固定概念，而是目标函数决定的。

4.5.4 实验四：作者多样性

假设 top 5 原始分：

```
1. author X, score 0.90
2. author X, score 0.88
3. author Y, score 0.86
4. author X, score 0.84
5. author Z, score 0.80
```

作者多样性会衰减 X 的第 2 条、第 3 条，使 Y 或 Z 有机会提前。它不是认为 X 的内容不好，而是优化 feed 序列体验。

4.5.5 实验五：缓存路径

假设 Redis 缓存有 700 条候选，超过阈值。系统会：

- `has_cached_posts=true`
- `CachedPostsSource` 启用
- Thunder/Phoenix/CoreDataHydrator 等跳过或减少工作

如果缓存只有 100 条候选，低于阈值，系统回到实时路径。这个实验说明缓存命中也有质量门槛。

4.5.6 实验六：side effect 失败

假设 `UpdateServedHistorySideEffect` 失败，但当前请求成功返回。用户下一次刷新可能再次看到相同内容。 假设 `ServedCandidatesKafkaSideEffect` 失败，用户体验当下不变，但训练和实验数据缺失。

这说明后台失败有延迟影响，不一定立刻显性。

4.5.7 本节练习

1. 构造 5 个候选，手算不同权重下的排序。
2. 构造一个 filter pipeline，计算每一层后剩余候选数。
3. 解释为什么召回 top K 和过滤率要一起调。
4. 设计一个实验，观察 author diversity 对用户体验的影响。

5 Home Mixer：把算法变成产品响应

这一章把模型能力接回线上服务。Home Mixer 的关键价值是编排：它把帖子候选、非帖子模块、广告、安全规则和 side effect 放进一次 For You 响应。

5.1 从模型到服务，推荐系统怎样活在线上

前几章讲的是“候选帖子怎样被召回和排序”。这一节把视角拉回线上服务：一次 feed 请求不只返回帖子，也不只调用一个模型。

5.1.1 外层 For You 流水线

`home-mixer/candidate_pipeline/for_you_candidate_pipeline.rs` 配置的是最终 feed item 的混排。它的 source 包括：

- `ScoredPostsSource`：来自内层帖子打分服务。
- `AdsSource`：广告。
- `WhoToFollowSource`：关注推荐。
- `PromptsSource`：提示或运营类内容。
- `PushToHomeSource`：推送到 Home 的内容。

然后它用 `BlenderSelector` 做混排，并在 side effects 中记录广告注入、曝光、客户端事件、served history 等。

> **新手视角：内层和外层的区别**
>
> 内层 `PhoenixCandidatePipeline` 更像“帖子推荐算法”；外层 `ForYouCandidatePipeline` 更像“feed 产品编排”。线上用户看到的是外层结果。

这就是为什么推荐算法工程师不能只看模型输出。模型给出的是候选帖子的排序信号；产品响应还要处理广告、关注推荐、历史去重、日志和缓存。

5.1.2 Thunder：实时 in-network 候选

Thunder 负责“你关注的人最近发了什么”。在 `thunder/posts/post_store.rs` 中，`PostStore` 用多个 `DashMap` 保存帖子：

```
posts: Arc<DashMap<i64, LightPost>>,
original_posts_by_user: Arc<DashMap<i64, VecDeque<TinyPost>>>,
secondary_posts_by_user: Arc<DashMap<i64, VecDeque<TinyPost>>>,
video_posts_by_user: Arc<DashMap<i64, VecDeque<TinyPost>>>,
deleted_posts: Arc<DashMap<i64, bool>>,
```

这个结构体现了两个线上需求：

- 查某个作者最近内容要快，所以按 user id 组织队列。
- 删除和过期要及时处理，所以有 deleted map 和 retention trim。

`start_auto_trim` 会后台定期清理旧帖子。`start_stats_logger` 会定期记录用户数、帖子数、删除数等指标。这些看起来不是算法，却决定了召回源是否健康。

> **提示：后台任务也是系统能力**

`interval.tick().await` 这行代码表达的是周期性调度，而不是用户请求路径。后台清理和指标上报看起来离算法远，但它们决定实时候选源能否长期保持健康。

5.1.3 Grox：内容理解不是排序，但会影响推荐

`grox/` 不是主排序流水线，但它代表内容理解任务：分类、摘要、多模态 embedding、安全策略等。`grox/engine.py` 里的 `Engine` 会从队列取任务，然后异步执行：

```
task = await self._poll_task()
if task is None:
    await asyncio.sleep(0.1)
    continue
asyncio.create_task(self._run_task(task))
```

这说明内容理解通常不是“请求来了才全部现算”。很多结果会提前写入存储，供推荐侧 hydration、filter 或模型输入使用。

对新手来说，先记住一句话：推荐系统不只需要知道“用户喜欢什么”，还需要知道“内容是什么、是否安全、是否可展示、是否适合这个场景”。

5.1.4 Side effects：响应之后还要做的事

在 `CandidatePipeline::execute` 的最后，系统调用 `run_side_effects(input)`。实现里用 `tokio::spawn` 把 side effects 放到后台任务里：

```
tokio::spawn(async move {
    let futures = side_effects
        .iter()
        .filter(|se| se.enable(input.query.clone()))
        .map(|se| se.run(input.clone()));
    let _ = join_all(futures).await;
});
```

side effect 的例子包括：

- 写曝光和候选日志到 Kafka。
- 更新 served history，避免短时间重复展示。
- 写 Redis 缓存，给后续请求复用。
- 记录统计指标，帮助发现空结果、延迟和异常。
- 触发实验相关数据收集。

这些操作重要，但很多不应该阻塞用户响应。否则用户看到 feed 的时间会被日志系统、缓存系统或统计系统拖慢。

小心：后台不代表不重要

side effect 失败可能不会让当前请求失败，但会影响后续推荐质量、实验分析和问题排查。线上系统需要指标、重试、降级和告警来管理这些失败。

5.1.5 新手需要的服务化系统最小模型

先不要背复杂术语。读这个项目时，只需要带着下面五个问题：

1. 这个组件依赖什么外部资源？
2. 这个依赖可以和其他工作并行吗？如果不能，依赖是什么？

3. 失败时是丢弃部分结果、使用默认值，还是让整个请求失败？
4. 这个结果会不会被缓存？缓存过期或不一致会怎样？
5. 这个阶段影响当前响应，还是影响后续请求和分析？

这五个问题足够支撑你读懂多数服务化推荐代码。

5.1.6 本节练习

1. 在 `home-mixer/candidate_pipeline/for_you_candidate_pipeline.rs` 里找到所有 source，判断哪些是帖子，哪些不是帖子。
2. 在 `thunder/posts/post_store.rs` 里找出删除、过期清理和统计上报对应的方法。
3. 在 `grox/engine.py` 中解释 `asyncio.create_task` 和直接等待任务完成的区别。

5.2 组件目录，给代码建立索引

读大型推荐系统时，最容易迷路的原因不是代码太难，而是不知道每个文件属于哪一段流程。本节把 `PhoenixCandidatePipeline` 和 `ForYouCandidatePipeline` 中出现的主要组件整理成目录。它不是最终 API 文档，而是读码索引。

5.2.1 帖子候选流水线的组件

`home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs` 配置了帖子候选的主流程。

阶段	组件	主要作用
Query	`ScoringSequenceQueryHydrator`	补排序模型使用的用户行为序列。
Query	`RetrievalSequenceQueryHydrator`	补召回模型使用的用户行为序列。
Query	`BlockedUserIdsQueryHydrator`	补 viewer 屏蔽列表。
Query	`MutedUserIdsQueryHydrator`	补 viewer 静音列表。
Query	`FollowedUserIdsQueryHydrator`	补关注列表，供 Thunder 和新用户逻辑使用。
Query	`SubscribedUserIdsQueryHydrator`	补订阅关系。
Query	`CachedPostsQueryHydrator`	读取缓存候选，支持降延迟或回退。
Query	`MutualFollowQueryHydrator`	补互关信息。
Query	`UserDemographicsQueryHydrator`	补用户画像类上下文。
Query	`FollowedGrokTopicsQueryHydrator`	补关注的话题。
Query	`FollowedStarterPacksQueryHydrator`	补 starter pack 相关上下文。
Query	`InferredGrokTopicsQueryHydrator`	补推断话题。
Query	`ImpressionBloomFilterQueryHydrator`	补曝光去重相关结构。
Query	`IpQueryHydrator`	补 IP 地理信息。
Query	`UserInferredGenderQueryHydrator`	补推断性别信息。

5.2.2 Source 目录

Source	候选来源	新手重点
`ThunderSource`	关注网络的实时帖子	依赖 followed_user_ids，适合理解 in-network。
`TweetMixerSource`	其他帖子混合服务	适合理解跨服务候选接入。

`PhoenixSource`	Phoenix 普通 out-of-network 召回	重点看 retrieval_sequence 和 enable 条件。
`PhoenixTopicsSource`	话题相关 Phoenix 召回	关注 topic request 场景。
`PhoenixMOESource`	专家模型或 MoE 召回	关注多路召回策略。
`CachedPostsSource`	缓存候选	关注缓存命中后如何跳过昂贵阶段。

Source 的共同问题是：它返回多少候选、候选带什么初始字段、失败时是否允许降级。

5.2.3 Candidate Hydrator 目录

Hydrator	补充字段
`InNetworkCandidateHydrator`	标记候选是否来自关注网络。
`CoreDataCandidateHydrator`	帖子文本、作者、回复、转帖等核心数据。
`QuoteHydrator`	quote 相关帖子与作者信息。
`VideoDurationCandidateHydrator`	视频时长。
`HasMediaHydrator`	是否有媒体。
`SubscriptionHydrator`	订阅内容相关字段。
`GizmoduckCandidateHydrator`	作者用户资料。
`BlockedByHydrator`	作者是否屏蔽 viewer。
`FilteredTopicsHydrator`	候选的话题过滤信息。
`LanguageCodeHydrator`	语言信息。

post-selection hydrator 则更靠近最终展示：

- `VFCandidateHydrator`：可见性过滤需要的信息。
- `AdsBrandSafetyHydrator`、`AdsBrandSafetyVfHydrator`：广告安全相关。
- `TweetTypeMetricsHydrator`：帖子类型指标。
- `FollowingRepliedUsersHydrator`：回复关系。
- `MutualFollowJaccardHydrator`：互关相似度。

5.2.4 Filter 目录

filter 数量多，说明线上推荐需要大量硬约束。可以按意图分类：

类别	组件
去重	`DropDuplicatesFilter`、`RetweetDeduplicationFilter`、`DedupConversationFilter`
数据完整性	`CoreDataHydrationFilter`
时间	`AgeFilter`
用户关系	`SelfTweetFilter`、`AuthorSocialgraphFilter`
曝光历史	`PreviouslySeenPostsFilter`、`PreviouslySeenPostsBackupFilter`、`PreviouslyServedPostsFilter`

内容资格	`IneligibleSubscriptionFilter`、`VideoFilter`、`TopicIdsFilter`、`NewUserTopicIdsFilter`
用户控制	`MutedKeywordFilter`
可见性	`VFFilter`、`AncillaryVFFilter`

读 filter 时要问：它保护谁？保护 viewer、作者、平台安全、产品体验，还是系统成本？

5.2.5 Scorer 和 Selector 目录

帖子候选流水线的 scorer：

- `PhoenixScorer`：调用 Phoenix prediction client，写入多行为预测。
- `RankingScorer`：加权多行为预测，加入多样性和 OON 调整，写入最终 score。
- `VMRanker`：调用外部 ranker，作为额外排序信号或重排环节。

selector：

- `TopKScoreSelector`：按 `candidate.score` 排序并截断。

外层 For You pipeline 的 selector：

- `BlenderSelector`：先 partition posts/ads/prompts/wtf/push-to-home，再按产品规则混排。

5.2.6 Side Effect 目录

帖子候选流水线 side effects：

- `PhoenixExperimentsSideEffect`
- `RerankingKafkaSideEffect`
- `RedisPostCandidateCacheSideEffect`
- `ScoredStatsSideEffect`
- `MutualFollowStatsSideEffect`
- `PhoenixRequestCacheSideEffect`

外层 feed side effects：

- `AdsInjectionLoggingSideEffect`
- `PublishSeenIdsToKafkaSideEffect`
- `ServedCandidatesKafkaSideEffect`
- `ClientEventsKafkaSideEffect`
- `ForYouResponseStatsSideEffect`
- `UpdatePastRequestTimestampsSideEffect`
- `UpdateServedHistorySideEffect`
- `TruncateServedHistorySideEffect`

side effect 是理解闭环的关键。它们把“本次推荐发生了什么”写回系统，供后续请求、训练、实验和分析使用。

5.2.7 如何使用这个目录

遇到新文件时，先把它放到上面的分类里。然后按统一模板阅读：

阶段：
输入字段：
输出字段：
外部依赖：

enable 条件：
失败策略：
关键指标：
用户可见影响：

这样你不会被文件数量淹没，也不会只停留在函数名层面。

5.3 Query Hydration，先把用户上下文补齐

推荐请求刚进入系统时，query 往往只有一部分信息：用户是谁、请求场景是什么、客户端带了哪些参数。它还不知道用户最近喜欢什么、关注谁、屏蔽谁、已经看过什么、是否有话题偏好。Query hydrator 的职责就是补齐这些上下文。

5.3.1 QueryHydrator 的合约

核心 trait 在 `candidate-pipeline/query_hydrator.rs`：

```
#[async_trait]
pub trait QueryHydrator<Q>: Any + Send + Sync
where
    Q: PipelineQuery,
{
    fn enable(&self, _query: &Q) -> bool { true }
    async fn hydrate(&self, query: &Q) -> Result<Q, String>;
    fn update(&self, query: &mut Q, hydrated: Q);
}
```

这个合约有两个动作：`hydrate` 负责去外部系统拿数据，返回一个只填了相关字段的新 query；`update` 负责把这些字段合并回主 query。

> **新手视角：为什么不直接修改原 query**
>
> 并行运行多个 query hydrator 时，如果大家同时写同一个对象，会出现竞争和合并问题。这个框架让 hydrator 先各自返回结果，再由 pipeline 顺序 merge，降低共享可变状态的复杂度。

5.3.2 例子一：ScoringSequenceQueryHydrator

`home-mixer/query_hydrators/scoring_sequence_query_hydrator.rs` 会访问用户行为聚合服务。它根据 user id、窗口时间、最大序列长度、聚合类型、是否包含实时行为等参数，拿到 scoring sequence。

简化后：

```
let result = self
    .user_action_aggregation_client
    .fetch_aggregated_sequence(...)
    .await?;

Ok(ScoredPostsQuery {
    scoring_sequence: Some(result.sequence),
    columnar_scoring_sequence: result.columnar_bytes,
    ..Default::default()
})
```

这个调用访问的是用户行为服务。它不是模型调用，但它决定了模型能看到什么历史。如果这个序列缺失，Phoenix ranking 可能只能返回默认分或无法正常请求。

5.3.3 例子二：FollowedUserIdsQueryHydrator

`home-mixer/query_hydrators/followed_user_ids_query_hydrator.rs` 更简单。它访问 social graph，拿当前用户关注的人：

```
let followed_user_ids = self
    .socialgraph_client
    .get_followed_user_ids(query.user_id)
    .await?;
```

这份数据至少影响两件事：

- ThunderSource 用它请求 in-network posts。
- 某些新用户逻辑会根据关注数判断用户状态。

这说明一个 query 字段可能服务多个后续阶段。读代码时不要只看 hydrator 自己，还要追踪这个字段被哪些 source、filter、scorer 使用。

5.3.4 初始 hydrator 和 dependent hydrator

`CandidatePipeline::execute` 先跑 `hydrate_query`，再跑 `hydrate_dependent_query`。这给了框架一个表达依赖的方式。

如果某个 hydrator 只依赖原始 query，它应该放在第一批。例如关注列表、屏蔽列表、用户行为序列。 如果某个 hydrator 依赖第一批补出来的字段，它应该放在 dependent 阶段。

这个划分比“全部顺序跑”更有效，也比“全部并行跑”更安全。

5.3.5 Query hydration 的失败策略

Query hydrator 失败后，pipeline 在 `hydrate_query` 中只会在 `Ok(hydrated)` 时调用 `update`：

```
for (hydrator, result) in hydrators.iter().zip(results) {
    if let Ok(hydrated) = result {
        hydrator.update(&mut hydrated_query, hydrated);
    }
}
```

这意味着失败字段会保持默认状态。默认状态是否安全，要看后续组件如何处理。

例如：

- scoring sequence 缺失时，PhoenixScorer 会返回默认 candidate。
- followed_user_ids 缺失时，ThunderSource 可能拿不到 in-network 候选。
- blocked_user_ids 缺失时，如果后续没有其他保护，可能影响用户安全体验。

> **小心：默认值不是天然安全**
>
> 写 hydrator 时必须思考默认值的语义。空列表可能表示“用户没有关注任何人”，也可能表示“社交图服务失败”。这两种情况后续处理不一定相同。

5.3.6 如何读一个 query hydrator

读任何 query hydrator，都按下面模板记笔记：

问题	说明
它补哪个字段?	看 `Ok(ScoredPostsQuery { ... })` 和 `update`。

它依赖哪个系统?	看调用的 client、请求参数和错误处理。
它如何选择参数?	看 `query.params.get(...)` 和 `query.decider`。
失败后默认值是什么?	看 pipeline 合并逻辑和下游组件。
谁会使用这个字段?	用 `rg` 搜字段名。

5.3.7 本节练习

1. 在 `phoenix_candidate_pipeline.rs` 中列出所有 query hydrator，并给每个标注它大概补什么字段。
2. 用 `rg "scoring_sequence"` 查找这个字段的所有使用位置，区分“写入者”和“读取者”。
3. 选一个你认为默认值风险较高的 hydrator，写下如果它失败，后续可能出现什么用户可见问题。

5.4 Sources，候选内容从哪里来

Source 是推荐系统里最像“找素材”的阶段。它不负责把内容讲完整，也不负责最终排序；它只回答一个问题：给这个 query，先拿到哪些候选?

5.4.1 Source 的合约

`candidate-pipeline/source.rs` 的 trait 很短：

```
#[async_trait]
pub trait Source<Q, C>: Any + Send + Sync
where
    Q: PipelineQuery,
    C: PipelineCandidate,
{
    fn enable(&self, _query: &Q) -> bool { true }
    async fn source(&self, query: &Q) -> Result<Vec<C>, String>;
}
```

source 返回的是一批 candidate。它可以来自内存、RPC、缓存、向量检索、广告系统或其他服务。

在 `CandidatePipeline::fetch_candidates` 中，所有启用的 source 会并行运行，然后把成功结果 append 到同一个候选列表里。失败 source 会被跳过。

5.4.2 例子一：ThunderSource

`home-mixer/sources/thunder_source.rs` 负责 in-network 候选。它读取 query 里的 `followed_user_ids`，向 Thunder 请求这些关注作者的近期帖子。

核心请求包含：

- `user_id`：当前用户。
- `following_user_ids`：关注列表。
- `max_results`：最多拿多少结果。
- `exclude_tweet_ids`：已经看过或需要排除的帖子。
- `algorithm`：Thunder 内部算法选择。

返回后，它把 `LightPost` 转成 `PostCandidate`，写入 `tweet_id`、`author_id`、reply/retweet 信息和 `served_type`。

> **提示：source 不需要补全所有字段**

ThunderSource 返回的 candidate 只是初始形态。文本、作者资料、视频时长、语言、安全标签等可以留给后续 hydrator。

5.4.3 例子二：PhoenixSource

`home-mixer/sources/phoenix_source.rs` 负责 out-of-network 的 Phoenix retrieval 候选。它读取 `retrieval_sequence`，调用 Phoenix retrieval client：

```
let response = self
    .phoenix_retrieval_client
    .retrieve(...)
    .await?;
```

它的 `enable` 条件比 ThunderSource 更复杂：

- topic request 的处理不同。
- 新用户话题召回可能绕开普通 PhoenixSource。
- `in_network_only` 时不启用。
- 已有 cached posts 时不启用。

这说明 source 不只是“调用外部服务”。它还承担“当前请求是否适合这个候选源”的判断。

5.4.4 候选源的互补性

一个 feed 请求通常不会只依赖一个 source：

Source	擅长什么	可能的问题
Thunder	关注网络、实时性、低延迟	候选范围受关注列表限制，新用户可能不足。
Phoenix Retrieval	全局语义发现、out-of-network	依赖行为序列和模型服务，可能受索引新鲜度影响。
Topics/MOE	按话题或专家模型扩展候选	需要请求上下文和策略判断。
CachedPosts	复用上次结果，降低延迟	可能新鲜度较差，需谨慎处理曝光历史。
Ads/Prompts/WTF	满足产品和商业需求	需要混排和安全约束。

互补性是推荐系统可靠性的来源之一。某个 source 质量下降时，其他 source 仍然可能提供可用结果。

5.4.5 Source 的调试方式

当 feed 空了，source 是最早要检查的地方。一个实用排查顺序：

1. query hydrator 是否补齐了 source 需要的字段？例如 `followed_user_ids`、`retrieval_sequence`。
2. source 的 `enable` 是否返回 false？检查 params、decider、request type。
3. source 的下游调用是否失败？看错误日志和 latency。
4. source 返回了多少原始候选？不要等到 filter 后才看。
5. 候选是否带了必要 id？例如 tweet id、author id、served type。

5.4.6 候选数量不是越多越好

source 多拿候选可以提高召回率，但也会增加后续成本：

- hydrator 要补更多候选字段。

- filter 要扫描更多候选。
- scoring 模型要处理更多 candidate。
- selector 和 side effect 的数据量也会变大。

因此 source 的 `max_results` 是效果和成本之间的权衡。读参数时要问：这个值是为了召回质量、延迟预算，还是下游容量?

5.4.7 本节练习

1. 在 `phoenix_candidate_pipeline.rs` 里找出 `sources` 列表，按 in-network、out-of-network、cache、其他系统分类。
2. 打开 `ThunderSource`，标出它依赖 query 的哪些字段。
3. 打开 `PhoenixSource`，解释它的 `enable` 条件分别在保护什么场景。

5.5 Candidate Hydration，把候选补成可判断的对象

Source 返回的候选通常很瘦。它可能只有 tweet id、author id、served type。这样的对象还不能安全展示，也不能充分排序。Candidate hydrator 的职责就是批量补字段。

5.5.1 Hydrator 的合约

`candidate-pipeline/hydrator.rs` 里有一个非常重要的约束：

```
/// IMPORTANT: The returned vector must have the same candidates in the same order as the input.
/// Dropping candidates in a hydrator is not allowed - use a filter stage instead.
async fn hydrate(&self, query: &Q, candidates: &[C]) -> Vec<Result<C, String>>;
```

这句话是理解 hydrator 的关键。hydrator 只能补字段，不能删候选。删除候选是 filter 的职责。

> **新手视角：为什么必须保持顺序**
>
> pipeline 会用 `zip` 把原候选和 hydration 结果配对。如果 hydrator 少返回一个、换了顺序或偷偷删除候选，后续字段就会写错对象。这类 bug 很隐蔽，所以框架显式检查长度。

5.5.2 update 和 update_all

hydrator 返回的不是完整 candidate，而是“只填了自己负责字段”的 candidate。`update` 决定如何把这些字段合并回原 candidate：

```
fn update(&self, candidate: &mut C, hydrated: C);
```

默认 `update_all` 会遍历每个结果，只在 `Ok(hydrated)` 时更新。失败结果会跳过，原 candidate 保持已有字段。

这个设计让每个 hydrator 有清晰边界：CoreData 只写文本和基础关系，VideoDuration 只写视频时长，LanguageCode 只写语言，不互相覆盖。

5.5.3 例子：CoreDataCandidateHydrator

`home-mixer/candidate_hydrators/core_data_candidate_hydrator.rs` 从 TES 获取帖子 core data。它补的字段包括：

- `retweeted_user_id`
- `retweeted_tweet_id`
- `in_reply_to_tweet_id`
- `tweet_text`

注意当前代码里的一个细节：`CoreDataCacheValue` 保存了 `author_id`，但 `CoreDataCandidateHydrator::update` 写回 candidate 时只更新转帖、回复关系和文本，不写回 `author_id`。因此读这一节时不要把“cache value 里有字段”和“update 一定写字段”混为一谈；真正决定字段是否生效的是 `update`。

它实现的是 `CachedHydrator`，所以先查本地缓存；缓存没有命中时，再批量调用 TES。

```
match self.cache_store().get(&key).await {
    Some(value) => results[index] = Some(Ok(self.hydrate_from_cache(value))),
    None => missing_candidates.push(candidate.clone()),
}
```

缓存命中的结果直接转成 hydrated candidate；缓存未命中的候选集中起来，交给 `hydrate_from_client` 批量请求。

5.5.4 为什么 hydration 要缓存

候选元数据经常被重复请求。同一条帖子可能在多个用户、多个刷新场景、多个召回源中出现。没有缓存时，元数据服务会承受大量重复读取。

缓存的收益：

- 降低外部服务压力。
- 降低 P50/P90 延迟。
- 在下游短暂抖动时提高成功率。

缓存的风险：

- 数据可能过期，例如帖子被删除、作者状态变化。
- 缓存 key 设计错误会污染字段。
- 缓存命中率低时，缓存层只增加复杂度。

因此缓存 hydrator 同时记录 cache hit/miss 指标。读这类代码时，指标不是附属品，而是判断设计是否有效的证据。

5.5.5 Hydration 缺失后的下一步

CoreData 可能找不到某条帖子的 metadata。它会返回默认 candidate，而不是直接删除。后续 `CoreDataHydrationFilter` 会用 `author_id != 0` 做完整性检查；这要求 source 或前序阶段已经提供有效作者字段。

这是一条清晰的职责线：

- hydrator：我尽力补字段，补不到就留下默认状态。
- filter：我根据字段状态决定候选能不能继续。

把这两者混在一起会让排查困难。你不知道候选是没被召回、补字段失败，还是被规则移除。

5.5.6 批量请求

hydrator 往往对一批候选发批量请求：

```
let tweet_ids: Vec<u64> = candidates.iter().map(|c| c.tweet_id).collect();
let post_features = client.get_tweet_core_datas(tweet_ids.clone()).await;
```

批量请求比逐个请求更适合线上推荐，因为它减少网络往返和服务端调度开销。但批量也有问题：批太大可能导致单次请求慢，批太小又浪费吞吐。候选数量、批大小和超时策略需要一起设计。

5.5.7 本节练习

1. 在 `candidate-pipeline/hydrator.rs` 中找出长度检查逻辑，解释如果 hydrator 少返回一个结果，框架如何处理。
2. 打开 `CoreDataCandidateHydrator`，列出 cache key、cache value 和 update 写入的字段。
3. 找一个非 cached hydrator，比较它和 CoreData 的失败处理方式。

6 从候选到最终顺序

候选被召回和补全后，系统还不能直接展示。它必须先删除不该出现的内容，再把预测变成业务分数，最后按产品约束选出可返回的序列。

6.1 Filtering，哪些候选不能继续

推荐系统不只是挑喜欢的内容，也要排除不该展示的内容。Filter 阶段负责把候选分成两类：继续参与后续流程的 kept，以及被移除的 removed。

6.1.1 Filter 的合约

`candidate-pipeline/filter.rs` 的核心类型是：

```
pub struct FilterResult<C> {
    pub kept: Vec<C>,
    pub removed: Vec<C>,
}

pub trait Filter<Q, C>: Any + Send + Sync {
    fn filter(&self, query: &Q, candidates: Vec<C>) -> FilterResult<C>;
}
```

filter 和 hydrator 最大的区别是：filter 可以改变候选集合大小。它拿走一批 candidates，返回 kept 和 removed。

pipeline 会顺序运行所有启用的 filter。每个 filter 只看上一个 filter 留下来的 candidates。

6.1.2 例子一：DropDuplicatesFilter

`home-mixer/filters/drop_duplicates_filter.rs` 是最容易理解的 filter：

```
let mut seen_ids = HashSet::new();

for candidate in candidates {
    if seen_ids.insert(candidate.tweet_id) {
        kept.push(candidate);
    } else {
        removed.push(candidate);
    }
}
```

它解决的是多 source 召回重叠。Thunder、Phoenix、缓存都可能返回同一条帖子。如果不去重，同一内容可能在后续阶段重复补字段、重复打分，甚至重复展示。

6.1.3 例子二：AuthorSocialgraphFilter

`home-mixer/filters/author_socialgraph_filter.rs` 处理社交关系约束。它会检查：

- viewer 是否屏蔽 author。
- viewer 是否静音 author。
- author 是否屏蔽 viewer。
- quote 或 retweet 涉及的作者是否有屏蔽关系。

这类 filter 不是“相关性”问题，而是用户控制和安全体验问题。模型认为相关，也不能越过用户明确的屏蔽和静音关系。

> **小心：模型分高不等于可以展示**

推荐系统里有硬约束和软目标。用户屏蔽、可见性、安全、删除状态通常是硬约束；相关性、多样性、商业目标是软目标或可调目标。

6.1.4 顺序为什么重要

filter 顺序会影响性能、归因和最终结果。通常有几个原则：

- 便宜的、确定性的过滤尽量靠前，例如去重。
- 依赖 hydration 字段的过滤必须放在相关 hydrator 之后。
- 安全和可见性过滤不能被省略，只能选择在合适阶段执行。
- 高移除率的 filter 靠前可以减少后续成本。

例如 `DropDuplicatesFilter` 很便宜，靠前能减少重复候选。`CoreDataHydrationFilter` 必须等 CoreDataHydrator 补完字段。`VFFilter` 在 post-selection 阶段运行，针对即将展示的候选做最终检查。

6.1.5 removed 不是垃圾数据

pipeline 保存了 filtered candidates。side effect 也能拿到 selected 和 non_selected。被移除的候选很有诊断价值：

- 哪个 filter 移除最多？
- 某次请求为什么空结果？
- 某个 source 返回的候选是否大部分不可展示？
- 某次参数调整是否导致过滤率异常上升？

因此 `Filter::run` 会记录 kept_count、removed_count、filter_rate。排查问题时，只看最终空不空是不够的，要看候选在哪一段消失。

6.1.6 Filter 里的默认值风险

filter 经常根据 `Option` 字段判断。如果默认值语义不清，就会出问题。例如：

```
let author_blocks_viewer = candidate.author_blocks_viewer.unwrap_or(false);
```

这表示“字段缺失时按没有屏蔽处理”。这个选择可能是为了避免过度过滤，也可能有安全风险。是否合理要结合上游 hydrator 的可靠性和后续可见性过滤一起判断。

新手读 filter 时，遇到 `unwrap_or(false)`、空列表、默认 candidate，要停下来问：缺失数据被当成允许、拒绝，还是未知？

6.1.7 本节练习

1. 在 `phoenix_candidate_pipeline.rs` 中把 filters 列成表，标注每个 filter 依赖哪些字段。
2. 找出一个只能在 hydration 后运行的 filter，说明原因。
3. 假设某次请求最终空结果，写出你会查看的三个过滤指标。

6.2 Scoring 和 Selection，从预测到最终顺序

过滤之后，剩下的候选还没有最终顺序。Scoring 阶段负责写入排序信号，Selection 阶段负责选择和混排。

6.2.1 Scorer 的合约

`candidate-pipeline/scorer.rs` 和 hydrator 很像，也要求返回结果长度和输入一致：

```
async fn score(&self, query: &Q, candidates: &[C]) -> Vec<Result<C, String>>;
fn update(&self, candidate: &mut C, scored: C);
```

scorer 不删除候选。它只写字段，例如 Phoenix 预测、加权分、最终分。

pipeline 顺序运行 scorer：

```
for scorer in all.iter().filter(|s| s.enable(query)) {
    let scored = scorer.run(query, &candidates).await;
    scorer.update_all(&mut candidates, scored);
}
```

顺序运行的原因是 scorer 之间可能有依赖。`RankingScorer` 需要读取 `PhoenixScorer` 写入的 `phoenix_scores`。

6.2.2 PhoenixScorer：调用模型服务

`home-mixer/scorers/phoenix_scorer.rs` 做三件事：

1. 根据 query 选择 Phoenix cluster。
2. 把 query 和 candidates 构造成 prediction request。
3. 调用 prediction client，把返回的 per-candidate score 写回 candidate。

它还包含一些线上逻辑，例如新用户阈值、decider override、egress sidecar fallback。新手不必一次看懂所有参数，但要知道这些不是模型结构，而是线上流量治理和实验控制。

6.2.3 RankingScorer：把多行为预测变成分数

`home-mixer/scorers/ranking_scorer.rs` 把 Phoenix 的多行为预测加权：

```
let combined_score = favorite * favorite_weight
    + reply * reply_weight
    + retweet * retweet_weight
    + not_interested * not_interested_weight
    + block_author * block_author_weight
    + ...;
```

然后它会：

- normalize score。
- 做作者多样性调整。
- 对 out-of-network 候选乘以 OON 权重。
- 写入 `weighted_score` 和 `score`。

这一步体现了推荐系统的目标组合：模型预测行为概率，业务参数决定如何权衡这些行为。

> **提示：分数是产品目标的编码**
>
> 分数不是自然存在的真理。它是多种目标的加权结果：喜欢、回复、停留、关注作者、负反馈、多样性、OON 探索等。调分就是调产品目标。

6.2.4 作者多样性

`RankingScorer` 中的 `apply_author_diversity` 会按分数排序，然后对同一作者的第 2 条、第 3 条内容做衰减。它的目标是避免一个作者连续占据太多位置。

这类规则说明推荐系统不只是最大化每条内容的独立分数。最终 feed 是一个序列，序列体验会受到重复作者、重复话题、广告间隔、会话结构等影响。

6.2.5 Selector 的合约

`candidate-pipeline/selector.rs` 负责排序和截断：

```
fn score(&self, candidate: &C) -> f64;

fn sort(&self, candidates: Vec<C>) -> Vec<C> {
    sorted.sort_by(|a, b| self.score(b).partial_cmp(&self.score(a)).unwrap_or(...))
}
```

`TopKScoreSelector` 很简单：读取 `candidate.score`，选择前 K 个。

```
fn score(&self, candidate: &PostCandidate) -> f64 {
    candidate.score.unwrap_or(f64::NEG_INFINITY)
}
```

如果候选没有 score，它会排到最后。这也是一个默认值语义：没有分数不是 0 分，而是负无穷。

6.2.6 BlenderSelector：最终 feed 不是只有帖子

`home-mixer/selectors/blender_selector.rs` 是外层 For You pipeline 的 selector。它先把 FeedItem 分成 posts、ads、who-to-follow、prompts、push-to-home，然后：

- 用 ads blender 混入广告。
- 插入 prompts。
- 插入 who-to-follow。
- 把 push-to-home 固定到开头。

这说明最终 selector 不一定只是按分数排序。不同 item 类型会有固定位置、间隔规则和产品约束。

6.2.7 本节练习

1. 打开 `RankingScorer`，找出正反馈、负反馈、多样性、OON 权重分别在哪段代码。
2. 修改一个假想候选的 favorite/reply/not_interested 分数，手算加权分变化。
3. 对比 `TopKScoreSelector` 和 `BlenderSelector`，说明它们为什么服务不同层级的 pipeline。

6.3 Side Effects 和观测性，返回之后系统还在工作

当 pipeline 选出最终候选后，请求并没有真正结束。系统还要写日志、更新缓存、记录 served history、发送 Kafka 事件、统计实验数据。这些操作通常不改变当前响应，但会影响后续推荐和问题排查。

6.3.1 SideEffect 的合约

`candidate-pipeline/side_effect.rs` 定义了 side effect 输入：

```
pub struct SideEffectInput<Q, C> {
    pub query: Arc<Q>,
    pub selected_candidates: Vec<C>,
    pub non_selected_candidates: Vec<C>,
}
```

它能看到：

- 当前 query。
- 最终选中的候选。
- 没有被选中的候选。

side effect 可以据此写曝光日志、缓存候选、统计被丢弃的数据，或者记录实验对照信息。

6.3.2 为什么不阻塞响应

`CandidatePipeline::run_side_effects` 用 `tokio::spawn` 后台执行：

```
tokio::spawn(async move {
    let futures = side_effects
        .iter()
        .filter(|se| se.enable(input.query.clone()))
        .map(|se| se.run(input.clone()));
    let _ = join_all(futures).await;
});
```

这样做的动机是降低用户可见延迟。当前响应已经有最终候选，不应该等 Kafka 或 Redis 写入完成。

但后台执行也带来风险：如果 side effect 持续失败，当前请求可能看起来成功，长期数据却坏了。比如 served history 没写进去，用户可能重复看到同一批内容；曝光日志缺失，训练数据和实验分析会偏。

6.3.3 例子：ServedCandidatesKafkaSideEffect

`home-mixer/side_effects/served_candidates_kafka_side_effect.rs` 会把 selected feed items 序列化成 thrift，然后发送到 Kafka。

它先根据 query 构造 request_info，再遍历 selected items：

```
for item in items {
    if let Some(entry_info) = build_entry_info(item) {
        let served = ServedEntry { request: request_info.clone(), entry:
Some(Box::new(entry_info)) };
        let bytes = serialize_compact(&served)?;
        payloads.push(bytes);
    }
}
```

最后并行发送：

```
let futs: Vec<_> = payloads.iter().map(|bytes| self.kafka_client.send(bytes)).collect();
let results = futures::future::join_all(futs).await;
```

这是一条典型日志链路：把线上展示事实写出去，供后续分析、训练、审计或回放使用。

6.3.4 观测性三件套

本项目里常见的观测方式包括：

- tracing span：记录阶段名、组件名、输入数量、输出数量、过滤率等。
- stats receiver：记录指标，例如 result size、cache hit/miss、filter kept/removed。
- structured log：记录错误、延迟、移除摘要等。

观测性不是上线后再补的装饰。推荐系统依赖太多，只有把每段输入输出都量化，才能知道问题发生在哪里。

6.3.5 常见线上问题和对应指标

问题	优先查看
feed 空结果	source candidate_count、filter removed_count、final result_size。
延迟变高	query hydrator/source/hydrator/scorer latency 分布。
重复内容	DropDuplicatesFilter removed、served history 写入、cached posts 逻辑。
某类内容突然减少	对应 source 返回量、topic filter、visibility filter、参数开关。

训练数据异常	served candidates Kafka side effect 错误率、曝光日志数量。

6.3.6 Side effect 的可靠性设计

后台任务至少要考虑：

- 是否需要重试?
- 是否需要限流?
- 失败是否会积压内存?
- 是否需要 dead letter queue?
- 是否要把错误按组件名打点?
- 是否会泄露用户隐私或敏感字段?

这些问题超出了“推荐算法”的狭义范围，但它们决定系统能否长期稳定运行。

> **检查点：推荐系统的闭环**
>
> 当前请求的 side effect 往往会成为未来请求的 query hydration 数据，也会成为未来训练样本的一部分。返回之后的写入，最终会影响下一轮推荐。

6.3.7 本节练习

1. 在 `phoenix_candidate_pipeline.rs` 和 `for_you_candidate_pipeline.rs` 中列出 side effects，按缓存、日志、统计、实验分类。
2. 选一个 side effect，解释它失败对当前请求和后续请求分别有什么影响。
3. 设计一个 dashboard：至少包含 source 返回量、filter 过滤率、scorer 延迟、final result size、side effect 错误率。

6.4 Selector 和混排，feed 是一个序列

Selector 不只是“取 top K”。在帖子候选流水线里，selector 可以很简单；在最终 feed 流水线里，selector 要处理广告、关注推荐、提示和置顶内容。本节对比 `TopKScoreSelector` 和 `BlenderSelector`。

6.4.1 TopKScoreSelector：帖子候选层

`home-mixer/selectors/top_k_score_selector.rs` 很短：

```
fn score(&self, candidate: &PostCandidate) -> f64 {
    candidate.score.unwrap_or(f64::NEG_INFINITY)
}

fn size(&self) -> Option<usize> {
    Some(params::TOP_K_CANDIDATES_TO_SELECT)
}
```

它依赖 `RankingScorer` 已经写入 `candidate.score`。如果 score 缺失，使用负无穷，避免未打分候选排到前面。

这是内层帖子推荐的典型 selector：排序、截断、把剩下的放到 non_selected。

6.4.2 BlenderSelector：最终 feed 层

`home-mixer/selectors/blender_selector.rs` 处理的是 `FeedItem`，不只是帖子。它先 partition：

```
let PartitionedFeedItems {
    posts,
```

```
    ads,
    wtf_modules,
    prompts,
    push_to_home,
} = partition_feed_items(candidates);
```

然后选择广告混排策略：

```
let blender: &dyn AdsBlender = match blender_type.as_str() {
    "safe_gap" => &self.safe_gap_blender,
    _ => &self.partition_organic_blender,
};
```

最后插入 prompts、who-to-follow、push-to-home。

6.4.3 为什么最终 feed 不能只按分数排序

最终 feed 是多种 item 的组合。不同 item 有不同目标：

- 帖子：相关性、互动、内容质量。
- 广告：商业投放、间隔、安全。
- Who to follow：关系增长。
- Prompt：产品引导或运营目标。
- Push to home：特殊入口或强策略内容。

如果统一用一个 score 排序，会很难表达固定位置、间隔、曝光频控和产品优先级。

6.4.4 混排的三个层次

混排可以分成三层：

层次	说明
候选内排序	帖子之间按 relevance score 排序。
异类 item 插入	广告、关注推荐、提示按规则插入。
最终约束修正	安全间隔、置顶、去重、位置修正。

`TopKScoreSelector` 主要处理第一层；`BlenderSelector` 处理第二、三层。

6.4.5 non_selected 的意义

Selector 返回 `SelectResult`：

```
pub struct SelectResult<C> {
    pub selected: Vec<C>,
    pub non_selected: Vec<C>,
}
```

non_selected 不是无用数据。它可以用于：

- side effect 记录哪些候选没展示。
- 缓存较高分但本次没展示的候选。
- 分析 selector 丢弃率。
- 训练时构造展示边界附近样本。

`BlenderSelector` 还会为被丢弃的 posts/ads 构造 placeholder，保留统计语义。

6.4.6 广告混排的特殊性

广告不是普通帖子。它需要：

- 遵守插入间隔。
- 尊重 brand safety。
- 记录 impression id。
- 处理商业投放约束。
- 不破坏用户体验。

因此 ads 先作为 source 进入外层 pipeline，再由 BlenderSelector 控制位置。这比把广告直接塞进帖子排序模型更清晰。

6.4.7 Selector 排查模板

```
selector:
input_count:
selected_count:
non_selected_count:
score field:
missing score count:
item type distribution before:
item type distribution after:
fixed position rules:
ads dropped:
posts dropped:
```

对最终 feed 问题，必须看 item type distribution。只看帖子分数不够。

6.4.8 本节练习

1. 对比 `TopKScoreSelector` 和 `BlenderSelector` 的输入类型。
2. 解释为什么 `BlenderSelector::score` 返回 0 也没问题。
3. 设计一个规则：每 8 条 organic post 后最多 1 条广告。写出它属于混排的哪一层。
4. 思考：如果 push-to-home 插入开头，会不会影响其他 item 的 position 语义?

7 组件深读：逐文件形成证据链

前面建立了阶段顺序，本章开始逐文件深读。每一节都按同一套问题走：谁启用它、依赖哪些字段、修改哪些字段、失败时保留什么证据。

7.1 Source 深度导读，三种候选入口

前面已经讲过 Source 的合约。这一节带读三个代表性 source：`ThunderSource`、`PhoenixSource`、`CachedPostsSource`。它们分别代表实时关注网络、模型召回、缓存回放三种候选入口。

7.1.1 读 Source 的四层结构

任何 source 都可以按四层拆开：

1. `enable`：当前请求是否应该运行。
2. 输入字段：它从 query 读取哪些信息。
3. 外部调用：它访问哪个系统、超时和错误如何处理。
4. candidate 构造：它给候选写入哪些初始字段。

这四层比函数名更重要。只看 source 名字，你会知道“它从哪里来”；看完四层，你才知道“它什么时候来、依赖什么、失败会怎样、返回什么”。

7.1.2 ThunderSource：关注网络召回

`home-mixer/sources/thunder_source.rs` 的 `enable` 很短：

```
fn enable(&self, query: &ScoredPostsQuery) -> bool {
    !query.has_cached_posts
}
```

如果已经命中 cached posts，它不运行。这是一个成本优化：缓存足够好时，不再打实时 Thunder。

它读取的 query 字段包括：

- `query.user_id`
- `query.user_features.followed_user_ids`
- `query.seen_ids`
- `query.params.get(ThunderMaxResults)`
- `query.params.get(ThunderAlgorithm)`
- `query.in_network_only`

其中 `followed_user_ids` 来自 query hydration。如果关注列表缺失，ThunderSource 的请求会退化，甚至拿不到有效 in-network 候选。

7.1.3 ThunderSource 的外部调用

ThunderSource 通过 gRPC client 调用：

```
let response = client
    .get_in_network_posts(request)
    .await
    .map_err(|e| format!("ThunderSource: {}", e))?;
```

这里的等待是线上延迟预算的一部分。它不是本地函数调用，而是跨服务请求。失败后整个 ThunderSource 返回 `Err`，pipeline 会跳过它的候选，但其他 source 仍可继续。

7.1.4 ThunderSource 的 candidate 构造

返回的 `LightPost` 被转成 `PostCandidate`：

```
PostCandidate {
    tweet_id: post.post_id as u64,
    author_id: post.author_id as u64,
    in_reply_to_tweet_id,
    retweeted_tweet_id,
    ancestors,
    served_type: Some(served_type),
    ..Default::default()
}
```

这些字段都是后续阶段的重要输入：

- `tweet_id` 用于去重、hydration、打分、日志。
- `author_id` 用于社交图过滤和作者多样性。
- reply/retweet/ancestors 用于会话处理和 served history。
- `served_type` 用于日志、分数分析和最终响应。

> **提示：Source 只写必要字段**
>
> ThunderSource 没有写文本、语言、安全标签和最终分。它只写 source 阶段确定知道的字段。其他信息交给后续 hydrator 和 scorer。

7.1.5 PhoenixSource：模型召回入口

`home-mixer/sources/phoenix_source.rs` 的 `enable` 更复杂：

```
(!query.is_topic_request() || query.is_bulk_topic_request())
    && (!query.params.get(EnableNewUserTopicRetrieval) || !query.has_new_user_topic_ids())
    && !query.in_network_only
    && !query.has_cached_posts
```

这段条件包含四类业务判断：

- topic request 场景是否应该走普通 Phoenix。
- 新用户话题召回是否接管。
- ranked following 请求是否禁止 OON。
- cached posts 是否已经接管。

读这种 enable 条件时，不要急着化简布尔表达式。更好的方法是把每一项翻译成业务保护。

7.1.6 PhoenixSource 的输入依赖

PhoenixSource 最关键的输入是 `retrieval_sequence`：

```
let sequence = query
    .retrieval_sequence
    .as_ref()
    .ok_or_else(|| "PhoenixSource: missing retrieval_sequence".to_string())?;
```

没有 retrieval sequence，它直接失败。这很合理：Phoenix Retrieval 需要用户历史来构造 user representation。如果行为序列缺失，召回请求没有足够上下文。

它还读取：

- `query.user_id`
- `query.columnar_retrieval_sequence`
- `query.params.get(PhoenixMaxResults)`

- client_context
- user_context
- cluster 和 decider

7.1.7 PhoenixSource 的新用户 cluster

resolve_cluster 会根据历史长度切换 cluster：

```
let action_count = query.retrieval_sequence
    .as_ref()
    .and_then(|s| s.metadata.as_ref())
    .map(|m| m.length)
    .unwrap_or(0);
```

如果 action_count 小于阈值，则使用新用户 cluster。这是推荐系统常见做法：新用户历史少，普通模型可能不稳，需要专门策略或模型。

7.1.8 CachedPostsSource：缓存回放

home-mixer/sources/cached_posts_source.rs 是最短的 source：

```
fn enable(&self, query: &ScoredPostsQuery) -> bool {
    query.has_cached_posts
}

async fn source(&self, query: &ScoredPostsQuery) -> Result<Vec<PostCandidate>, String> {
    Ok(query.cached_posts.clone())
}
```

它不访问外部服务，因为缓存读取已经在 CachedPostsQueryHydrator 中完成。它只把 query 里的 cached_posts 放回 source 阶段。

这个设计看起来绕，但很统一：无论候选来自实时服务、模型召回还是缓存，后续 pipeline 都通过 source 阶段接收 Vec<PostCandidate>。

7.1.9 三种 source 的对比

Source	候选来源	关键依赖	主要风险
ThunderSource	实时 in-network	关注列表、Thunder gRPC	关注列表缺失、Thunder 慢、候选太少。
PhoenixSource	OON 模型召回	retrieval sequence、模型召回服务	行为序列缺失、cluster 错误、召回质量下降。
CachedPostsSource	Redis 缓存结果	cached_posts query 字段	缓存过期、候选重复、上下文不匹配。

7.1.10 Source 排查模板

```
source name:
enabled:
disabled reason:
input fields:
external dependency:
timeout/failure behavior:
candidate count:
fields written:
served_type:
downstream filters likely to remove:
```

这个模板能帮你把 source 的行为从“它返回了一些候选”变成可排查事实。

7.1.11 本节练习

1. 给 `ThunderSource` 填一份 source 排查模板。
2. 给 `PhoenixSource` 填一份 source 排查模板。
3. 假设 `query.has_cached_posts=true`，写出哪些 source 会跳过，为什么。
4. 解释为什么 `CachedPostsSource` 放在 source 阶段，而不是直接跳过 pipeline 返回。

7.2 字段追踪，从一个 query 字段看完整链路

大型系统里，理解一个字段比理解一个函数更重要。字段从哪里来、被谁读取、默认值是什么、失败后会怎样，决定了系统行为。本节用几个字段示范追踪方法。

7.2.1 字段追踪方法

追踪字段时按四步走：

1. 搜写入者：`rg "field_name ="` 或看 hydrator 的 `update`。
2. 搜读取者：`rg "field_name"`。
3. 区分阶段：query hydration、source、filter、scorer、side effect。
4. 记录默认值语义：空、None、0、false 分别代表什么。

不要只搜类型定义。字段真正的语义来自读写位置。

7.2.2 `followed_user_ids`

写入者是 `home-mixer/query_hydrators/followed_user_ids_query_hydrator.rs`：

```
query.user_features.followed_user_ids = hydrated.user_features.followed_user_ids;
```

读取者包括：

- `ThunderSource`：构造 `GetInNetworkPostsRequest.following_user_ids`。
- `RankingScorer`：判断新用户是否有足够关注数。
- 某些 filter 或 hydrator 可能使用关注关系判断 in-network reply。

这个字段的默认值是空列表。空列表可能有两种含义：

- 用户真的没有关注任何人。
- Social graph 调用失败，hydrator 没有更新字段。

后续代码如果无法区分这两种情况，就可能把服务失败当成新用户状态。

7.2.3 `retrieval_sequence`

写入者是 `RetrievalSequenceQueryHydrator`。读取者是 `PhoenixSource`。

PhoenixSource 对它的态度很强硬：

```
.ok_or_else(|| "PhoenixSource: missing retrieval_sequence".to_string())?
```

也就是说缺失时 PhoenixSource 失败，不返回候选。pipeline 仍可依赖 Thunder 或缓存等其他 source，但 OON 召回会缺失。

字段语义：

- `Some(sequence)`：可以构造用户向量。
- `None`：召回缺少上下文，不应该静默发空序列。

7.2.4 `scoring_sequence`

写入者是 `ScoringSequenceQueryHydrator`。读取者是 `PhoenixScorer`。

PhoenixScorer 对缺失 scoring sequence 的处理是：

```
if query.scoring_sequence.is_none() {
    return vec![Ok(PostCandidate::default()); candidates.len()];
}
```

这意味着它不会失败整个 scorer，而是给每个候选返回默认更新。后续 `RankingScorer` 仍会运行，但缺少 Phoenix 预测时，许多 action score 是 None，最终分会受影响。

这和 `retrieval_sequence` 的策略不同。原因是它们处在不同阶段：retrieval sequence 缺失时 source 无法产生候选；scoring sequence 缺失时候选已经存在，系统可能仍希望用默认分或其他 scorer 继续。

7.2.5 `has_cached_posts`

写入者是 `CachedPostsQueryHydrator`。它读取 Redis，只有 cached posts 数量达到阈值才设为 true：

```
let has_cached_posts = cached_posts.len() >= MIN_CACHED_POSTS_THRESHOLD;
```

读取者很多：

- `ThunderSource::enable`：cached true 时跳过。
- `PhoenixSource::enable`：cached true 时跳过。
- `CoreDataCandidateHydrator::enable`：cached true 时跳过。
- `CachedPostsSource::enable`：cached true 时运行。
- `RedisPostCandidateCacheSideEffect::enable`：cached true 时不再写缓存。

这个字段相当于一个模式切换开关。它让 pipeline 从“实时召回 + hydration + scoring”切到“缓存候选回放”。

7.2.6 `author_id`

`author_id` 主要由 source 初始写入。`CoreDataCandidateHydrator` 的 cache value 保存了 `author_id`，但当前 `update` 不把它写回 candidate，所以读代码时要以 `update` 为准。这个字段被多个阶段使用：

- `CoreDataHydrationFilter` 用 `author_id != 0` 判断 core data 是否有效。
- `AuthorSocialgraphFilter` 用它检查屏蔽和静音。
- `RankingScorer` 用它做作者多样性。
- 日志和 served history 需要记录 author id。

因此 `author_id=0` 不是普通作者，而是缺失或无效信号。这个约定必须贯穿 source、hydrator 的 update 逻辑和 filter。

7.2.7 `score`

`score` 由 `RankingScorer` 写入，`TopKScoreSelector` 读取：

```
candidate.score.unwrap_or(f64::NEG_INFINITY)
```

缺失 score 被当成负无穷，而不是 0。这个默认值很重要：没有打分的候选不应该意外排在正常候选前面。

7.2.8 字段追踪表

字段	写入者	主要读取者	缺失风险
`followed_user_ids`	FollowedUserIdsQueryHydrator	ThunderSource、RankingScorer	in-network 候选不足，新用户判断失真。

retrieval_sequence	RetrievalSequenceQueryHydrator	PhoenixSource	OON 召回失败。
scoring_sequence	ScoringSequenceQueryHydrator	PhoenixScorer	模型预测缺失，最终分质量下降。
has_cached_posts	CachedPostsQueryHydrator	多个 enable 条件	pipeline 路径切错。
author_id	Source, CoreData cache value 只作辅助证据	Filter/Scorer/Log	误过滤、误排序、日志缺字段。
score	RankingScorer	TopKScoreSelector	候选无法正常排序。

7.2.9 本节练习

1. 用 `rg "has_cached_posts"` 搜索所有读取者，画出 cached 模式切换图。
2. 用 `rg "author_id"` 搜索 `home-mixer/filters`，找出哪些 filter 依赖作者字段。
3. 选择一个字段，写出它的“默认值是否安全”分析。

7.3 CachedHydrator 深度导读

Hydrator 经常访问外部服务。为了降低延迟和下游压力，框架提供了 `CachedHydrator`。这一节带读它的控制流，并解释为什么缓存逻辑要写在抽象层。

7.3.1 CachedHydrator 的角色

普通 Hydrator 只定义：

```
async fn hydrate(&self, query: &Q, candidates: &[C]) -> Vec<Result<C, String>>;
```

CachedHydrator 拆得更细：

```
fn cache_key(&self, candidate: &C) -> Self::CacheKey;
fn hydrate_from_cache(&self, value: Self::CacheValue) -> C;
async fn hydrate_from_client(&self, query: &Q, candidates: &[C]) -> Vec<Result<C, String>>;
fn cache_value(&self, hydrated: &C) -> Self::CacheValue;
```

它把“查缓存、找 miss、批量打 client、写回缓存”的模板统一起来。具体 hydrator 只需要定义 key/value 和如何访问 client。

7.3.2 控制流

`CachedHydrator` 的默认实现大致是：

```
for each candidate:
  key = cache_key(candidate)
  if cache hit:
    results[index] = hydrate_from_cache(value)
  else:
    collect missing candidate/key/index

if missing not empty:
  hydrated_missing = hydrate_from_client(missing_candidates)
  for each hydrated missing:
    if ok:
      cache_store.insert(key, cache_value(hydrated))
    results[index] = hydrated

return results in original order
```

关键点是保存 `missing_indices`。因为 client 只处理 miss 子集，返回结果必须放回原 candidates 的位置。

7.3.3 为什么要保持原顺序

假设输入候选是：

`[A, B, C, D]`

其中 A、C 命中缓存，B、D miss。client 只会收到 `[B, D]`。返回后必须还原成：

`[A_result, B_result, C_result, D_result]`

如果错位，B 的文本可能写到 C 上，后续 filter 和 scorer 会基于错误内容决策。这类错误很难通过类型系统发现，所以框架显式用 index 合并。

7.3.4 CoreDataCandidateHydrator 的 key/value

`CoreDataCandidateHydrator` 的 cache key 是 `tweet_id`：

```
fn cache_key(&self, candidate: &PostCandidate) -> u64 {
    candidate.tweet_id
}
```

cache value 包含：

- author id
- retweeted user id
- retweeted tweet id
- in reply to tweet id
- tweet text

这说明缓存内容是 core data 的一个子集，不是完整 PostCandidate。缓存 value 越小，存储和序列化成本越低；但字段太少，后续可能仍需访问 client。

7.3.5 缓存命中指标

`CachedHydrator::stat_cache` 会记录 cache hit 和 miss。读缓存逻辑时必须看指标，因为缓存是否有效不能靠感觉判断。

如果 miss 很高，可能说明：

- cache key 设计不稳定。
- TTL 太短。
- 候选重复度低。
- cache store 不健康。
- 某些请求绕过了缓存写入。

如果 hit 很高但数据错误，可能说明：

- cache value 过期。
- key 缺少上下文维度。
- 更新或删除没有同步失效。

7.3.6 缓存和 correctness

缓存不是单纯性能优化，它会改变系统行为。CoreData 里的文本、作者关系、删除状态等如果过期，会影响过滤和展示。哪些字段可以缓存，取决于它们变化频率和错误代价。

经验上：

- 稳定字段更适合缓存，例如历史帖子文本。
- 频繁变化或安全敏感字段要谨慎缓存。
- 可见性、删除、屏蔽关系通常需要更强的新鲜度保证。

7.3.7 本节练习

1. 在 `hydrator.rs` 中画出 CachedHydrator 的 hit/miss 控制流。
2. 分析 `CoreDataCacheValue` 中每个字段的变化频率。
3. 假设 tweet text 缓存过期，会影响哪些 filter 或日志?
4. 设计一个新的 cached hydrator，写出 cache key、cache value、失效风险。

7.4 Filter 深度导读，硬约束怎样落地

Filter 是推荐系统里的硬约束执行层。这一节带读三个代表：`CoreDataHydrationFilter`、`AuthorSocialgraphFilter`、`VFFilter`。

7.4.1 CoreDataHydrationFilter：数据完整性

`home-mixer/filters/core_data_hydration_filter.rs` 只有一行核心逻辑：

```
let (kept, removed) = candidates.into_iter().partition(|c| c.author_id != 0);
```

它把 `author_id != 0` 当成候选作者信息是否有效的代理信号。为什么不是检查 tweet_text? 因为 author id 是更基础的关系字段，后续社交图过滤、作者多样性、日志都依赖它。

这个 filter 展示了一个常见模式：hydrator 负责补字段，filter 负责检查字段是否足够完整。

7.4.2 AuthorSocialgraphFilter：用户控制

`home-mixer/filters/author_socialgraph_filter.rs` 先把 blocked/muted list 转成 HashSet：

```
let viewer_blocked_user_ids: HashSet<i64> =
    query.user_features.blocked_user_ids.iter().copied().collect();
```

然后对每个 candidate 检查多个关系：

- author 是否被 viewer muted。
- author 是否被 viewer blocked。
- author 是否 block viewer。
- quoted author 是否 block viewer。
- viewer 是否 block quoted author。
- viewer 是否 block retweeted user。

这说明社交图过滤不只看候选作者。quote、retweet、reply 都可能把其他用户带进展示上下文。

7.4.3 VFFilter：可见性最终检查

`home-mixer/filters/vf_filter.rs` 使用 `visibility_reason`：

```
fn should_drop(reason: &Option<FilteredReason>) -> bool {
    match reason {
        Some(FilteredReason::SafetyResult(safety_result)) => {
            matches!(safety_result.action, Action::Drop(_))
        }
        Some(_) => true,
        None => false,
    }
}
```

这个逻辑很值得细看：

- `SafetyResult` 且 action 是 Drop，则移除。
- 其他 Some(reason) 也移除。
- None 不移除。

这意味着 `visibility_reason` 的存在通常代表已经有过滤理由；没有理由则默认保留。这个默认是否安全，取决于上游 VFCandidateHydrator 是否可靠，以及是否还有其他可见性保护。

> **小心：安全相关默认值要特别审查**
>
> 对推荐相关字段，缺失可能只是效果下降；对安全和可见性字段，缺失可能造成错误展示。读这类代码时要明确 None 的语义。

7.4.4 Filter 顺序案例

一个合理顺序可能是：

```
DropDuplicatesFilter
CoreDataHydrationFilter
AgeFilter
SelfTweetFilter
AuthorSocialgraphFilter
...
VFFilter (post-selection)
```

前面的 filter 通常便宜、确定、能减少候选。后面的 filter 可能依赖昂贵 hydration 或只需要对选中的候选做最终检查。

7.4.5 过滤率解读

过滤率高不一定坏。比如 DropDuplicatesFilter 高，可能说明多 source 重叠高；VFFilter 高，可能说明上游候选质量差或可见性策略变严。关键是结合 source 和阶段看。

排查时不要只问“为什么删了这么多”，要问：

- 哪个 source 的候选被删得最多？
- 删除发生在 scoring 前还是 post-selection？
- filter 输入量是否异常？
- 删除是否和参数或模型发布同一时间变化？

7.4.6 本节练习

1. 分析 `CoreDataHydrationFilter` 为什么用 `author_id != 0`。
2. 给 `AuthorSocialgraphFilter` 画一张关系检查图。
3. 解释 `VFFilter` 中 `Some(_) => true` 的含义和风险。
4. 设计一个 filter 指标面板，能区分 source 质量和 filter 策略变化。

7.5 Scorer 深度导读，从模型预测到最终分

Scorer 阶段最容易被新手误解成“模型调用”。实际线上 scorer 还包含 cluster 选择、fallback、权重组合、归一化、多样性和特殊人群逻辑。

7.5.1 PhoenixScorer 的职责边界

`PhoenixScorer` 做的是模型服务调用，不做最终加权排序。它的输出是 `phoenix_scores`，包含多种 action 的预测。

关键步骤：

1. 根据 query 解析 cluster。
2. 如果新用户历史少，切到新用户 cluster。
3. 根据 decider 做 override。
4. 构造 prediction request。
5. 选择 egress client 或普通 phoenix client。
6. 如果 egress 失败，fallback 到普通 client。
7. 把每个候选的预测写回 `PostCandidate`。

这个流程说明模型服务上线后也需要流量治理。cluster、sidecar、fallback 都是生产系统能力。

7.5.2 RankingScorer 的职责边界

`RankingScorer` 不调用 Phoenix 服务。它读取已有 `phoenix_scores`，计算最终排序分。

核心顺序是：

```
Phoenix 多行为分数
  -> weighted score
  -> normalize
  -> author diversity
  -> OON weight
  -> final score
```

这些步骤都可能改变最终排序。排查排序问题时，要逐层看，而不是只看 Phoenix 原始预测。

7.5.3 多行为权重

`ScoringWeights::from_params` 从参数系统读取权重。正反馈包括 favorite、reply、retweet、click、dwell、follow_author 等。负反馈包括 not_interested、block_author、mute_author、report、not_dwelled 等。

这让系统可以表达复杂目标：

- 提高 favorite 权重，会偏向轻互动内容。
- 提高 reply 权重，可能增加讨论性内容。
- 提高 dwell 权重，可能偏向长内容或视频。
- 加强负反馈权重，会更保守地避开用户讨厌的内容。

权重调节必须通过实验验证，因为行为之间会相互影响。

7.5.4 offset_score 的目的

`RankingScorer` 里有 `offset_score`。它处理 combined score 可能为负的情况，并把分数映射到适合后续排序的区间。

新手不必一开始记住公式，只需要理解：多行为加权不是简单加完就结束，还要处理负反馈、归一化和分数尺度。否则不同类型候选的分数不可比。

7.5.5 作者多样性

`apply_author_diversity` 会先按 weighted score 排序，再按作者出现次数衰减分数。它有两个参数：

- decay factor：重复作者衰减速度。

- floor：最低保留比例。

这个设计避免同一作者刷屏，同时不至于把同一作者后续内容直接清零。

7.5.6 OON 权重

`effective_oon_weight` 会根据 topic request、新用户状态和默认参数选择 OON 权重。它说明最终分不仅与内容相关，还与候选来源有关。

OON 权重太低，用户会困在关注网络里；太高，用户可能看到过多陌生内容。这个权衡是 feed 产品的核心问题之一。

7.5.7 Scorer 排查模板

```
候选 id:
served_type:
phoenix_scores 是否存在:
weighted_score:
normalized score:
author diversity multiplier:
oon multiplier:
final score:
selector rank:
```

用这个模板抽样 10 个候选，通常能快速定位排序异常发生在哪一步。

7.5.8 本节练习

1. 手写一个候选的 favorite、reply、not_interested 加权分。
2. 给同一作者的三条候选模拟 diversity 衰减。
3. 比较 in-network 和 out-of-network 候选在 OON 权重后的分数变化。
4. 解释为什么 PhoenixScorer 和 RankingScorer 分成两个 scorer。

8 状态、内容理解和安全边界

推荐系统不是无状态函数。实时关注网络、内容理解、缓存、served history、可见性和广告安全都会改变候选能否进入最终 feed。本章把这些“模型之外但决定体验”的能力串起来。

8.1 Thunder 深入，实时关注网络候选

Thunder 是 in-network 候选源。它解决的问题不是“全局找相似内容”，而是“用户关注的人最近发了什么，并且要很快返回”。

8.1.1 PostStore 的数据结构

`thunder/posts/post_store.rs` 中的 `PostStore` 保存几类数据：

```
posts: Arc<DashMap<i64, LightPost>>,
original_posts_by_user: Arc<DashMap<i64, VecDeque<TinyPost>>>,
secondary_posts_by_user: Arc<DashMap<i64, VecDeque<TinyPost>>>,
video_posts_by_user: Arc<DashMap<i64, VecDeque<TinyPost>>>,
deleted_posts: Arc<DashMap<i64, bool>>,
```

这里有一个重要设计：完整帖子数据按 `post_id` 存，用户时间线只存 `TinyPost`，也就是 post id 和 created_at。这样按作者查最近内容很快，同时避免在多个队列里重复保存完整帖子。

8.1.2 写入路径

`insert_posts` 会先过滤过旧或来自未来的帖子，再按时间排序，最后调用 `insert_posts_internal`。内部逻辑会：

- 跳过已经删除的帖子。
- 把完整 `LightPost` 写入 `posts`。
- 根据 original/reply/retweet/video 放入不同的 per-user 队列。

这是一种典型的服务端内存索引：写入时多维护几份轻量索引，读取时降低扫描成本。

8.1.3 删除和过期

`mark_as_deleted` 会从 `posts` 移除帖子，并写入 `deleted_posts`。`trim_old_posts` 会定期清理超过 retention 的帖子。

删除和过期都很重要：

- 删除处理保护内容正确性和用户安全。
- 过期清理控制内存大小。
- deleted_posts 还能处理 create/delete 事件乱序。

新手容易只看“如何插入”，但线上系统必须同等重视“如何删除”和“如何收缩”。

8.1.4 gRPC 服务层

`thunder/thunder_service.rs` 暴露 `get_in_network_posts`。它先用 semaphore 限制并发：

```
let _permit = match self.request_semaphore.try_acquire() {
    Ok(permit) => permit,
    Err(_) => return Err(Status::resource_exhausted("Server at capacity, please retry")),
};
```

这是一条非常重要的保护线。没有并发限制时，突发流量可能把内存、CPU 或下游服务打满，导致所有请求一起超时。拒绝一部分请求比拖垮整个服务更可控。

8.1.5 following list 的来源

Thunder 请求可以直接带 `following_user_ids`。如果请求没带，并且 debug 条件满足，服务会尝试从 Strato 拉取关注列表。线上 Home Mixer 的 ThunderSource 通常会通过 query hydrator 先补关注列表，再传给 Thunder。

这说明同一个数据可以在不同层补：

- Home Mixer 先补，Thunder 只负责查帖子。
- Thunder 自己 fallback 拉取，适合 debug 或特殊场景。

边界越清晰，系统越容易排查。

8.1.6 指标

Thunder 记录了很多指标，例如：

- 请求数和延迟。
- following list 大小。
- exclude list 大小。
- 返回帖子新鲜度。
- reply ratio。
- unique authors。
- posts per author。
- rejected requests。

这些指标回答的问题不是“模型准不准”，而是“实时候选源健康吗”。如果 Thunder 返回的帖子太旧、作者太少、回复比例异常，Home Mixer 后面的排序再好也会受影响。

8.1.7 Thunder 和 Phoenix 的互补

Thunder 偏实时、偏关注网络；Phoenix Retrieval 偏全局发现、偏语义相似。一个健康 feed 往往需要二者互补：

- Thunder 保证关注关系和新鲜内容。
- Phoenix 帮用户发现关注网络之外的内容。
- RankingScorer 再统一打分和做 OON 权重调整。

如果用户关注的人很少，Thunder 候选不足；如果 Phoenix 召回质量下降，feed 可能缺少新发现。排查时要分开看 in-network 和 out-of-network 的供给。

8.1.8 本节练习

1. 在 `PostStore` 中画出 `posts`、`original_posts_by_user`、`deleted_posts` 的关系。
2. 解释为什么 per-user 队列只保存 `TinyPost`。
3. 打开 `ThunderServiceImpl::get_in_network_posts`，找出并发保护、输入限制、指标记录三个位置。

8.2 Grox 深入，内容理解怎样进入推荐

Grox 不在主 feed 排序路径里直接返回候选，但它代表推荐系统的另一条重要链路：内容理解。摘要、分类、安全标签、多模态 embedding、回复排序等结果，都会影响后续候选生成、过滤、排序或训练。

8.2.1 Engine：任务执行入口

`grox/engine.py` 的 `Engine` 从任务队列取 `TaskPayload`，然后创建异步任务执行：

```
task = await self._poll_task()
if task is None:
```

```
    await asyncio.sleep(0.1)
    continue
asyncio.create_task(self._run_task(task))
```

这和 feed 请求的同步链路不同。Grox 更像后台任务系统：不断拉任务、执行计划、写结果。

8.2.2 PlanMaster：多个计划并行试配

`grox/plans/plan_master.py` 中有 `ALL_PLANS`：

```
ALL_PLANS = [
    PlanInitialBanger(),
    PlanPostSafety(),
    PlanSpamComment(),
    PlanPostEmbeddingWithSummary(),
    ...
]
```

`exec` 会并行执行所有 plan：

```
results = await asyncio.gather(*[p.execute(task) for p in cls.ALL_PLANS])
```

每个 plan 自己判断是否 eligible。不适合当前任务的 plan 返回 None，适合的 plan 产出 `TaskResult`。最后 `merge_results` 合并内容分类、embedding、reason、success/error。

8.2.3 Plan：有依赖的任务图

`grox/plans/plan.py` 的 `Plan` 支持 `TASK_DEPENDENCIES`。执行时，它为依赖任务创建 future，然后并行调度所有任务：

```
await asyncio.gather(
    *[self._execute_task(t, ctx, dependencies) for t in self.TASKS.keys()]
)
```

每个 task 在执行前等待自己的依赖：

```
dep_futures = [dependencies[d] for d in deps]
dep_results = await asyncio.gather(*dep_futures)
```

这比简单的顺序任务列表更灵活。没有依赖的任务可以并行，有依赖的任务等前置结果。

8.2.4 内容理解结果如何影响推荐

Grox 的结果可能进入推荐系统的多个位置：

- 安全分类进入 visibility 或 safety filter。
- 摘要和 embedding 进入 retrieval corpus 或相似度模型。
- 话题分类进入 topic source、topic filter 或用户兴趣建模。
- 回复排序信号进入 reply ranking 或会话展示。
- 多模态描述帮助模型理解图片、视频和文本之外的内容。

这些结果不一定在用户刷新 feed 时实时计算。更多时候，它们提前生成并写入存储，推荐请求只读取这些结果。

8.2.5 后台内容理解的优点和风险

优点：

- 把昂贵模型计算从在线请求中移走。
- 允许重试、批处理和异步补偿。
- 让多个下游系统复用同一份内容理解结果。

风险：

- 结果有延迟，新内容可能暂时缺标签或 embedding。
- 任务失败会造成数据缺口。
- 多个计划合并时要处理部分成功。
- 内容理解模型更新会影响推荐效果，需要版本和回放。

8.2.6 Grox 和 Candidate Hydration 的关系

Candidate hydrator 经常读取内容理解结果。例如安全标签、摘要、话题、embedding。Grox 更偏“生产这些结果”，Home Mixer 更偏“消费这些结果”。把生产和消费分开，可以降低在线延迟，但也引入数据新鲜度和一致性问题。

8.2.7 本节练习

1. 在 `PlanMaster.ALL_PLANS` 中选三个 plan，推测它们处理哪类任务。
2. 打开一个具体 plan，画出它的 `TASKS` 和 `TASK_DEPENDENCIES`。
3. 解释为什么内容安全分类更适合提前计算，而不是每次 feed 请求时现算。

8.3 缓存和状态，推荐为什么要记住过去

推荐系统不是无状态函数。它要记住刚刚展示过什么、缓存哪些高分候选、用户最近请求过几次、哪些内容已经看过。本节看 cached posts 和 served history 两条状态链路。

8.3.1 CachedPostsQueryHydrator：读缓存

`home-mixer/query_hydrators/cached_posts_query_hydrator.rs` 从 Redis 读缓存候选。它有两个关键参数：

```
const MIN_CACHED_POSTS_THRESHOLD: usize = 500;
const REDIS_GET_TIMEOUT: Duration = Duration::from_millis(300);
```

这说明缓存命中不是只看 Redis 有无 payload。只有候选数量达到阈值，才认为 `has_cached_posts=true`。

为什么需要阈值？因为缓存里如果只有很少候选，直接走 cached path 可能导致 feed 供给不足。阈值让系统在缓存不够完整时回到实时召回路径。

8.3.2 读缓存的失败策略

Redis GET 被 `timeout` 包住：

```
tokio::time::timeout(REDIS_GET_TIMEOUT, self.redis_client.get(cache_key.clone())).await
```

超时或错误时 hydrator 返回 Err。pipeline 不会更新 cached_posts，也不会设 `has_cached_posts`。于是后续 source 会按实时路径运行。

这是一种合理降级：缓存是优化，不是唯一数据源。

8.3.3 CachedPostsSource：把缓存重新接回 source 阶段

如果 `has_cached_posts=true`，`CachedPostsSource` 运行，其他昂贵 source 通常跳过。这样 pipeline 后面仍然处理 `Vec<PostCandidate>`，不需要为缓存路径写另一套主流程。

缓存路径的好处：

- 降低实时召回和模型服务压力。
- 降低延迟。
- 在部分下游不稳定时提供可用结果。

缓存路径的风险：

- 新鲜度下降。
- 用户刚看过的内容可能重复。
- 参数或请求上下文变了，缓存候选可能不再适合。

8.3.4 RedisPostCandidateCacheSideEffect：写缓存

`home-mixer/side_effects/redis_post_candidate_cache_side_effect.rs` 在响应后写缓存。它会从 selected 和 non_selected 中挑选 weighted_score 大于 0 的候选，并按分数排序截断。

这说明缓存的不只是最终展示内容，也可以缓存高质量但未展示候选。下一次请求可以更快拿到一批候选。

它还使用 zstd 压缩，并设置 TTL：

```
const REDIS_TTL_SECONDS: u64 = 180;
const ZSTD_COMPRESSION_LEVEL: i32 = 6;
```

TTL 代表缓存只服务短期复用，不是长期存储。

8.3.5 Served history：避免短期重复

`home-mixer/side_effects/update_served_history_side_effect.rs` 会把最终 FeedItem 写入 served history。它支持不同 item 类型：

- Post
- Ad
- WhoToFollow
- Prompt
- PushToHome

对 Post，如果有 ancestors，会把祖先和当前 tweet 都写入。这能帮助后续请求避免展示同一会话的重复分支。

8.3.6 状态的两种用途

状态	用途	风险
cached posts	复用高分候选，降低延迟和成本。	过期、重复、上下文不匹配。
served history	短期去重和会话控制。	写入失败导致重复展示。
past request timestamps	控制请求频率或刷新逻辑。	状态不准导致策略误判。
seen ids	排除已看内容。	客户端和服务端视图不一致。

8.3.7 状态一致性

状态系统经常是最终一致的。side effect 写入在后台执行，可能晚于响应，也可能失败。下一次请求可能读到旧状态。

因此状态逻辑要能容忍：

- 刚写的状态还没读到。
- 某次写入失败。
- 不同存储之间不一致。
- 缓存内容和最新可见性不一致。

这也是为什么最终可见性 filter 不能因为 cached posts 命中就完全跳过。

8.3.8 本节练习

1. 画出 cached posts 的读路径和写路径。
2. 解释 `MIN_CACHED_POSTS_THRESHOLD` 保护了什么。
3. 解释为什么缓存 selected 和 non_selected 都有意义。
4. 分析 served history 写入失败会造成哪些用户可见问题。

8.4 安全和可见性，推荐系统的硬边界

推荐系统不能只追求相关性。删除、屏蔽、静音、安全策略、广告 brand safety、可见性过滤都是硬边界。本节把这些机制放在同一张图里。

8.4.1 安全和相关性的区别

相关性回答：这个内容用户可能喜欢吗？ 安全和可见性回答：这个内容能展示吗？

两者的优先级不同。一个内容即使相关性很高，只要违反可见性规则，也必须被移除。反过来，一个内容通过安全检查，也不代表它值得推荐。

8.4.2 可见性数据从哪里来

可见性通常由 hydrator 补字段，再由 filter 执行移除：

```
VFCandidateHydrator
  -> candidate.visibility_reason
  -> VFFilter
  -> kept / removed
```

这种拆分让系统能记录“可见性服务返回了什么”和“filter 根据它做了什么”。

8.4.3 VFFilter 的决策

`VFFilter` 中：

```rust
Some(FilteredReason::SafetyResult(safety_result)) => {
    matches!(safety_result.action, Action::Drop(_))
}
Some(_) => true,
None => false,
```

可以理解为：

- 明确 drop：删除。
- 其他过滤理由：删除。
- 没有过滤理由：保留。

这个逻辑依赖上游正确填充 `visibility_reason`。如果 hydrator 没运行或失败，None 会被保留。因此安全链路需要额外的监控和可能的兜底策略。

8.4.4 社交关系安全

`AuthorSocialgraphFilter` 保护用户控制：

- viewer blocked author。
- viewer muted author。
- author blocked viewer。
- quote/retweet 涉及的作者关系。

这类过滤通常在 scoring 前执行，因为没必要给用户明确不想看的内容打分。

8.4.5 广告 brand safety

外层 feed 还要处理广告安全。广告出现在某些内容旁边可能有额外限制。相关 hydrator 包括：

- `AdsBrandSafetyHydrator`
- `AdsBrandSafetyVfHydrator`

广告安全说明推荐系统服务的不只是 viewer，还要考虑广告主、平台政策和内容上下文。

8.4.6 内容理解和安全

Grox 这类内容理解系统会生成安全分类、摘要、embedding、标签。安全标签可能被 Home Mixer 的 hydrator 或 filter 消费。由于内容理解可能异步生成，系统要处理“标签还没到”的情况。

常见策略：

- 对高风险内容默认保守。
- 对缺标签内容降权或延迟进入某些 source。
- 对最终展示执行实时可见性检查。
- 记录缺标签率和安全服务错误率。

8.4.7 安全链路排查

如果不该展示的内容出现，排查顺序：

1. source 是否返回了该候选?
2. core data 是否识别了正确作者、quote、retweet、reply 关系?
3. social graph filter 是否运行?
4. visibility hydrator 是否返回 reason?
5. VFFilter 是否运行?
6. post-selection 后是否又插入了未检查内容?
7. served history 和日志是否记录了该展示?

如果大量内容被误杀，排查顺序类似，但重点看 visibility_reason 分布和 filter_rate。

8.4.8 本节练习

1. 比较 `AuthorSocialgraphFilter` 和 `VFFilter`，说明它们保护的对象不同。
2. 解释为什么安全相关过滤不应该只依赖模型分数。
3. 设计一个可见性 dashboard：至少包含 VF hydrator 错误率、VFFilter 移除率、None 比例、按 source 的移除率。

8.5 广告混排和安全间隔

广告混排是 feed 系统里的特殊主题。它既要满足商业投放，也要保护用户体验和内容安全。本节看 `SafeGapAdsBlender`。

8.5.1 AdsBlender 的位置

广告不是帖子 scorer 的一部分。它在外层 For You pipeline 作为 `AdsSource` 返回 `FeedItem::Ad`，然后由 `BlenderSelector` 混入最终 feed。

这样设计有几个好处：

- 帖子排序和广告插入解耦。
- 广告有独立的候选来源和安全约束。
- 混排可以根据产品规则调整，而不影响帖子 ranker。

8.5.2 SafeGapAdsBlender

`home-mixer/ads/safe_gap_blender.rs` 的核心：

```
let safe_gaps = find_safe_gaps(&scored_posts);
let spacing = compute_spacing(&ads);
let first_ideal = ads[0].insert_position.max(0) as usize;
let placements = assign_ads_to_gaps(&safe_gaps, ads.len(), &spacing, first_ideal);
```

它不是简单按广告要求的位置插入，而是先找 safe gaps，再根据 spacing 分配广告位置。

8.5.3 safe gap 的意义

safe gap 可以理解为“广告可以插入的位置”。某些内容附近不适合放广告，例如安全风险内容、敏感上下文、或产品上不希望打断的结构。

广告安全不只是广告本身安全，还包括广告附近的 organic 内容是否合适。

8.5.4 spacing

广告之间要有间隔。间隔太小，用户体验差；间隔太大，投放不足。`assign_ads_to_gaps` 会考虑 ideal 位置和 min 位置：

```
let ideal = prev_ideal + spacing.requested;
let min = (prev_ideal + spacing.min).max(last_actual + DEFAULT_SPACING.min);
```

这说明混排是约束求解问题：既要接近理想位置，又要满足最小间隔和 safe gap。

8.5.5 如果没有足够 safe gaps

`find_best_gap` 找不到合适位置时，后续广告不会插入。这比强行插入更安全。

这类逻辑体现了商业目标和用户体验之间的边界：广告收益重要，但不能破坏安全和体验约束。

8.5.6 广告日志

广告 item 在 served candidates side effect 中会记录 `promotedTweet`、advertiser id、insert position、impression id 等。广告日志比普通帖子更复杂，因为它还服务计费、归因和投放分析。

8.5.7 本节练习

1. 解释为什么广告混排放在外层 For You pipeline。
2. 设计一个有 20 条帖子和 3 条广告的 safe gap 分配例子。
3. 如果 safe gaps 只有 1 个，系统应该插入几条广告？为什么？
4. 讨论广告插入如何影响 organic post 的 position。

8.6 Thunder ingestion，实时数据怎样进入内存索引

Thunder 不只是查询服务，还需要持续接收帖子创建和删除事件。虽然本节不完整展开 Kafka listener 的每个细节，但要建立实时 ingestion 的基本图景。

8.6.1 实时索引的目标

Thunder 的目标是低延迟返回关注作者近期内容。为此，它需要：

- 快速接收新帖子事件。
- 快速处理删除事件。
- 维护按作者分组的时间队列。
- 控制内存增长。
- 在请求时快速按关注列表扫描。

这和离线 corpus 不同。离线 corpus 更关注批量计算和向量索引；Thunder 更关注新鲜度和内存结构。

8.6.2 写入路径

Kafka listener 消费事件后，会把 `LightPost` 写入 `PostStore::insert_posts`。PostStore 会：

1. 过滤过旧和未来时间。
2. 按 created_at 排序。
3. 跳过已删除帖子。
4. 写入 `posts`。
5. 写入 per-user 队列。

这个路径把事件流转成可查询索引。

8.6.3 删除路径

删除事件调用 `mark_as_deleted`：

```
self.posts.remove(&post.post_id);
self.deleted_posts.insert(post.post_id, true);
```

deleted_posts 的存在是为了处理事件乱序。如果先收到 delete，再收到 create，insert 时会检查 deleted_posts 并跳过。

> 实时系统必须假设事件可能乱序、重复或延迟。

8.6.4 自动清理

`start_auto_trim` 后台定期执行：

```
interval.tick().await;
let trimmed = self.trim_old_posts().await;
```

清理旧帖子可以控制内存，并保证 Thunder 返回的内容足够新。

8.6.5 请求时扫描

ThunderService 接收 following list 后，会从 PostStore 按作者取候选。为了防止过载，它使用 semaphore 限制并发。为了观察质量，它记录新鲜度、作者数、reply ratio、返回比例等指标。

> 这说明实时召回不是简单 HashMap 查询。它还要处理容量保护和质量监控。

8.6.6 Ingestion 和 Query Hydration 的关系

Home Mixer 通过 `FollowedUserIdsQueryHydrator` 补关注列表，然后 ThunderSource 把列表发给 Thunder。Thunder 只负责“这些作者最近有什么内容”。

> 边界清晰：

- Home Mixer 知道 viewer 上下文。
- Thunder 知道实时帖子索引。

8.6.7 本节练习

1. 解释为什么 Thunder 要同时有 `posts` 和 `original_posts_by_user`。
2. 解释 deleted_posts 如何处理事件乱序。
3. 设计一个指标判断 Thunder 数据新鲜度是否下降。
4. 思考：如果 Kafka ingestion 延迟 10 分钟，For You feed 会出现什么变化？

9 数据闭环、参数和实验

当一次请求返回后，系统并没有结束。曝光、点击、负反馈、缓存写入和实验分桶会回到下一次请求、离线评估和后续模型。这里讨论“系统如何变化”，而不是只讨论一次调用如何成功。

9.1 数据闭环，从一次曝光到下一版模型

推荐系统不是一条单向链路。用户这次看到什么、点了什么、停留多久、屏蔽了什么，都会变成未来的训练样本和线上上下文。本节把“当前请求”放进更大的闭环里。

9.1.1 闭环的基本形状

```
线上请求
  -> 召回、过滤、排序、展示
  -> 曝光日志和行为日志
  -> 数据清洗和样本构造
  -> 模型训练或持续训练
  -> 导出 checkpoint / embedding / config
  -> 部署到 retrieval / ranking 服务
  -> 下一次线上请求
```

本书前面主要讲第一段：线上请求如何产生 feed。本节关注后半段：这些线上事实如何回到模型和系统。

9.1.2 为什么曝光日志重要

如果只记录用户点击了什么，而不记录系统展示了什么，就无法知道用户没有点击的内容是什么。训练推荐模型需要正负样本，曝光日志提供了“候选被展示但未互动”的上下文。

例如：

- 展示后 favorite：强正反馈。
- 展示后 reply：强互动，但可能有争议。
- 展示后 dwell：弱正反馈或兴趣信号。
- 展示后 not interested：明确负反馈。
- 展示后无互动：不能简单等于负反馈，要结合位置、停留时间、可见性。

这就是 side effect 和 Kafka 日志为什么重要。它们不是附属工程，而是训练数据的入口。

9.1.3 当前请求里的闭环写入

Home Mixer 的 side effects 会写多类数据：

- served candidates：系统展示了什么。
- seen ids：用户已经看到或即将看到哪些内容。
- served history：短期去重。
- client events：客户端侧行为或展示事件。
- request cache：复用本次请求信息。
- scored stats：候选分数和分布统计。

这些数据有的服务下一次请求，有的服务训练和分析，有的服务实验评估。

9.1.4 用户行为序列从哪里来

`ScoringSequenceQueryHydrator` 和 `RetrievalSequenceQueryHydrator` 都会访问用户行为聚合服务。这个服务的输入通常来自历史日志：用户看过什么、点过什么、停留多久、是否负反馈。

因此 query hydration 不是凭空产生上下文，而是在读取数据闭环沉淀下来的用户历史。

9.1.5 模型 artifacts 如何回到本地 demo

`phoenix/run_pipeline.py` 使用的 artifacts 可以看作模型闭环的一个切片：

- `model_params.npz`：训练后导出的模型参数。
- `embedding_tables.npz`：训练得到的 hash embedding table。
- `config.json`：模型结构、hash 参数、序列长度等配置。
- `sports_corpus.npz`：预计算的候选表示。
- `example_sequence.json`：示例用户行为历史。

线上系统会用服务化方式加载和调用这些能力；本地 demo 直接从文件加载，便于学习。

9.1.6 数据闭环里的延迟

并不是所有用户行为都会立刻影响下一次推荐。数据闭环有不同延迟：

数据	延迟	影响
served history	秒级到分钟级	短期去重，避免刚看过又出现。
实时行为序列	秒级到分钟级	让模型快速感知新兴趣。
离线训练样本	小时到天级	影响下一版模型。
候选 corpus embedding	分钟到小时级	影响召回新鲜度。
内容安全标签	秒级到小时级	影响可见性和过滤。

理解延迟后，你就不会期待“刚点了一条内容，所有模型立刻重新训练”。更多时候，系统通过实时序列、缓存和短期状态快速响应，再通过训练闭环慢慢更新模型。

9.1.7 数据质量问题

闭环系统常见数据质量问题：

- 曝光日志缺失，导致训练样本偏差。
- 行为日志延迟，导致 query hydration 看不到最新兴趣。
- 负反馈定义不稳定，导致模型目标漂移。
- 去重状态写入失败，导致重复展示。
- 内容标签缺失，导致候选被过度过滤或错误放行。
- 实验分桶不一致，导致指标分析失真。

这些问题都不是模型结构能单独解决的。推荐系统的质量来自模型、日志、特征、服务和实验的一致性。

9.1.8 本节练习

1. 选一个 side effect，说明它写出的数据会在闭环中服务哪一段。
2. 解释为什么“未点击”不一定等于负样本。
3. 画出从 `ServedCandidatesKafkaSideEffect` 到下一轮模型训练的假想链路。

9.2 参数、开关和实验，线上系统怎样安全变化

真实推荐系统不能每次改一点策略都重新发布代码。很多行为由参数、feature switch、decider 和实验系统控制。本项目里你会反复看到 `query.params.get(...)` 和 `query.decider`。

9.2.1 参数在哪里出现

在 source、hydrator、filter、scorer 中，参数控制很多行为：

- source 的最大结果数，例如 `PhoenixMaxResults`、`ThunderMaxResults`。
- 模型 cluster，例如 `PhoenixInferenceClusterId`。
- 新用户阈值，例如 `PhoenixRankerNewUserHistoryThreshold`。
- 排序权重，例如 favorite、reply、dwell、not interested。
- 多样性参数，例如 `AuthorDiversityDecay`、`AuthorDiversityFloor`。
- 过滤阈值，例如最大帖子年龄。
- 混排策略，例如 `AdsBlenderType`。

参数的价值是把可调策略从代码里抽出来。风险是参数组合变多，系统行为更难推断。

9.2.2 Decider override

`PhoenixScorer::resolve_cluster` 中有 decider override：

```
if let Some(decider) = &query.decider {
    match configured_cluster {
        PhoenixCluster::Experiment1Fou if decider.enabled("override_qf_use_lap7") => {
            return PhoenixCluster::Experiment1Lap7;
        }
        ...
    }
}
```

这说明即使参数配置了某个 cluster，decider 也可能按实验或开关把流量导到另一个 cluster。读线上代码时，不能只看默认参数，还要看实验开关和请求上下文。

9.2.3 为什么需要实验

推荐系统目标复杂，很多改动无法只靠离线指标判断。例如提高 reply 权重可能增加互动，但也可能增加争议内容。上线前需要 A/B 实验观察：

- 用户停留和互动。
- 负反馈。
- 内容多样性。
- 新用户留存。
- 作者生态。
- 延迟和错误率。

实验不是证明代码能跑，而是判断产品目标是否真的变好。

9.2.4 参数改动的风险

参数看起来比代码安全，但也可能造成严重影响：

- 把负反馈权重设错符号。
- 把 retrieval top K 调太大，导致 scoring 延迟上升。
- 把某个 source 关闭，导致候选不足。
- 把新用户阈值调错，导致大量用户走错模型 cluster。
- 把可见性相关开关关闭，造成安全风险。

因此重要参数需要审查、监控和回滚方案。

9.2.5 阅读参数代码的步骤

看到 `query.params.get(X)`，按下面步骤追：

1. 这个参数在哪个阶段读取?

2. 它控制 enable、数量、阈值、权重，还是下游 cluster?
3. 默认值是什么?
4. 它是否和 decider 或 request type 共同作用?
5. 改它会影响延迟、召回量、过滤率，还是最终排序?
6. 有没有指标能验证它生效?

9.2.6 实验分析的基本单位

一次推荐实验至少要同时看四类指标：

类别	例子
效果	favorite、reply、dwell、retention、follow author。
负反馈	not interested、mute、block、report。
供给	source candidate count、filter rate、in-network/OON 占比。
系统	latency、error rate、cache hit、side effect failures。

只看效果指标容易忽略系统成本；只看系统指标又无法判断用户体验是否变好。

9.2.7 本节练习

1. 在 `home-mixer/scorers/ranking_scorer.rs` 中找出所有权重参数，按正反馈和负反馈分类。
2. 在 `PhoenixSource` 和 `PhoenixScorer` 中比较 new user threshold 的作用。
3. 设计一个实验：把 OON 权重提高 10%。写出你会观察的五个指标。

9.3 训练和评估，线上目标怎样变成模型

本项目主要提供推理和服务代码，但 README 和 Phoenix demo 已经给出训练闭环的入口线索。本节从新手角度解释：线上行为如何变成训练目标，模型如何评估，为什么离线指标不能替代线上实验。

9.3.1 样本从哪里来

训练样本通常来自：

- 展示日志：用户看到了哪些候选。
- 行为日志：用户 favorite、reply、repost、click、dwell、not interested 等。
- 候选上下文：当时的 user history、candidate、position、surface。
- 内容理解结果：安全标签、topic、embedding、摘要。

没有展示日志，就很难知道“用户没有点”的候选是什么。没有行为日志，就无法定义正负反馈。没有上下文，就无法复现模型当时应该看到的信息。

9.3.2 Label 的复杂性

推荐系统的 label 不只是 0/1：

行为	建模含义
favorite	明确正反馈，但可能偏轻量。
reply	强互动，可能代表兴趣，也可能代表争议。
repost/quote	传播意愿。
click/profile click	探索行为。
dwell	停留信号，常作为连续值或弱正反馈。

not interested	明确负反馈。
block/mute/report	强负反馈和安全信号。

多行为模型的意义就是同时学习这些目标，而不是压成一个模糊的 relevance。

9.3.3 离线评估

离线评估可以看：

- AUC、log loss、calibration。
- top K recall。
- 多行为预测质量。
- 分桶指标，例如新用户、重度用户、视频、话题请求。
- 负反馈预测能力。

离线评估的优点是快、便宜、可重复。缺点是它依赖历史数据分布，无法完全预测线上用户对新排序策略的反应。

9.3.4 线上评估

线上 A/B 实验需要看：

- 互动和留存。
- 负反馈和安全事件。
- 内容多样性。
- 作者生态。
- 延迟和错误率。
- 日志完整性。

线上实验更接近真实目标，但成本高、周期长、需要防护。推荐系统的成熟度体现在：能把离线评估、线上实验和系统指标连起来决策。

9.3.5 训练和服务的一致性

常见问题是训练和服务不一致：

- 训练用的 feature 和线上 query hydration 字段不同。
- 训练样本里的 action 定义和线上 scorer 权重不一致。
- 训练时没有模拟 candidate isolation。
- 线上 hash 参数和导出 config 不一致。
- 训练数据缺少某类请求或某类用户。

本地 `run_pipeline.py` 通过加载 config、hash 参数、embedding table 和 checkpoint，帮助你理解推理一致性的重要性。

9.3.6 评估新模型的检查清单

```
模型版本：
训练数据时间窗：
行为 label 定义：
hash/config 是否匹配：
离线指标：
分桶指标：
负反馈指标：
推理延迟：
内存和吞吐：
```

线上实验分桶：
回滚方案：

9.3.7 本节练习

1. 选择 favorite、reply、dwell、not interested 四个 label，说明它们可能冲突的场景。
2. 解释为什么离线 AUC 提升不保证线上体验提升。
3. 设计一个新 ranker 上线前的最小评估表。

9.4 参数调试手册

参数是线上推荐系统最常见的控制面。本节给出一份面向新手的调试手册：当你看到某个参数时，如何判断它影响什么、如何安全修改、如何验证。

9.4.1 参数类型

常见参数可以分为：

类型	例子
数量	source max results、top K、cache max posts。
阈值	新用户历史长度、最小视频时长、年龄阈值。
权重	favorite/reply/dwell/not interested/OON。
开关	是否启用 cached posts、是否启用某 source。
策略名	ads blender type、cluster id。
超时	Redis GET timeout、viewer data timeout。

不同类型参数的风险不同。数量和超时影响成本与延迟；权重影响排序目标；开关和策略名可能改变整条路径。

9.4.2 修改参数前的五个问题

1. 这个参数在哪些组件读取?
2. 它影响候选数量、过滤率、打分，还是混排?
3. 它是否只对某些用户或请求生效?
4. 有没有现成指标验证它生效?
5. 出问题时如何回滚?

如果回答不了，不应该直接调。

9.4.3 权重参数

排序权重最敏感。调整 favorite、reply、dwell、negative feedback 会改变产品目标。建议：

- 一次只改少数权重。
- 先做小流量实验。
- 同时看正反馈和负反馈。
- 看分桶，不只看总体。
- 保留回滚配置。

负反馈权重尤其要小心。符号错误或绝对值过小都可能造成体验问题。

9.4.4 数量参数

`PhoenixMaxResults`、`ThunderMaxResults`、top K 这类参数影响候选供给和成本。

调大：

- 可能提高召回覆盖。
- 会增加 hydration、filter、scoring 成本。
- 可能增加重复和低质候选。

调小：

- 降低成本和延迟。
- 增加候选不足风险。
- 在高过滤率请求上更危险。

数量参数要和过滤率、最终 result size 一起看。

9.4.5 超时参数

超时不是越长越好。超时长能等到更多结果，但会拉高尾延迟；超时短能保护用户体验，但会增加降级率。

调超时时要看：

- 下游 P95/P99。
- 当前请求延迟预算。
- 降级后是否有可用 fallback。
- 超时错误是否可观测。

9.4.6 开关参数

开关参数风险最大，因为它可能改变路径。例如 `EnableCachedPosts` 会使 cached path 接管，影响 source、hydrator、side effect 的 enable。

开关上线要明确：

- 默认关还是默认开。
- 哪些用户生效。
- 是否支持快速回滚。
- 关闭后是否有残留状态。

9.4.7 参数变更记录模板

参数名：
旧值：
新值：
影响组件：
影响用户范围：
预期变化：
观察指标：
上线时间：
回滚方式：
负责人：

这份记录看起来繁琐，但能避免事故后没人知道系统为什么变了。

9.4.8 本节练习

1. 选择一个 source max results 参数，写出调大和调小的影响。
2. 选择一个 negative feedback 权重，设计实验指标。
3. 找一个 enable 开关，画出打开后 pipeline 路径变化。

10 生产工程：排查、观测、测试和降级

理解推荐系统，最终要能解释线上现象。本章把前面的阶段模型落到 runbook、dashboard、测试策略和降级决策上，帮助读者从“看懂代码”走向“能判断系统是否健康”。

10.1 生产排查 Runbook

这一节把前面的概念整理成可执行的排查手册。它不追求覆盖所有事故，只覆盖新手最常遇到的五类问题：空结果、慢请求、重复内容、排序异常、日志缺失。

10.1.1 Runbook 一：空结果

症状：某类请求返回 0 个候选，或 final result size 明显下降。

排查步骤：

1. 确认请求类型：initial/top/bottom、topic request、in_network_only、cached posts。
2. 查看 query hydration 是否成功：scoring_sequence、retrieval_sequence、followed_user_ids、blocked/muted ids。
3. 查看每个 source 的 enabled_count 和 candidate_count。
4. 对比 Thunder 和 Phoenix 的候选量，判断是 in-network 还是 OON 供给问题。
5. 查看 hydration missing/error，尤其 CoreData 和 visibility 相关字段。
6. 查看 filters 的 removed_per_filter。
7. 查看 scorer 是否写入 score。
8. 查看 selector input_count 和 selected_count。
9. 查看 post-selection filters 是否移除全部候选。
10. 对比最近参数、decider、模型 cluster、下游错误率变化。

结论要写成“候选在哪一段消失”，而不是“推荐坏了”。

10.1.2 Runbook 二：慢请求

症状：P95/P99 延迟上升，用户刷新 feed 变慢。

排查步骤：

1. 先看整体 execute latency。
2. 拆 stage：query_hydrators、sources、hydrators、filters、scorers、selector、post-selection。
3. 对慢 stage 按 component name 排序。
4. 看候选数量是否异常增长。
5. 看 cache hit/miss 是否变化。
6. 看下游错误率和重试率。
7. 看并发限制是否触发，例如 Thunder 的 rejected requests。
8. 看是否有新参数启用了额外 source 或 hydrator。

慢请求经常是“候选量变大 + 缓存命中下降 + 某个下游慢”叠加出来的。

10.1.3 Runbook 三：重复内容

症状：同一帖子、同一会话或同一作者重复出现。

排查步骤：

- 同一 tweet 重复：看 `DropDuplicatesFilter` 是否运行，tweet_id 是否正确。
- retweet 重复：看 `RetweetDeduplicationFilter` 和 retweeted_tweet_id 是否 hydrated。
- conversation 重复：看 `DedupConversationFilter` 和 conversation/ancestor 字段。

- 作者重复：看 RankingScorer 的 author diversity 参数和输出。
- 短时间重复曝光：看 served history side effects 是否成功写入。

重复问题通常跨 source、hydrator、filter、side effect。只看 selector 往往不够。

10.1.4 Runbook 四：排序看起来不合理

症状：明显低质量内容排高、负反馈内容变多、某类内容突然占比上升。

排查步骤：

1. 查看 PhoenixScorer 是否调用了预期 cluster。
2. 检查 prediction request 中用户历史和候选是否合理。
3. 抽样查看 per-action probabilities。
4. 检查 RankingScorer 权重参数是否变化。
5. 检查负反馈权重是否被错误设置为正或过小。
6. 检查 OON weight、topic weight、新用户逻辑。
7. 检查 author diversity 是否生效。
8. 检查 BlenderSelector 是否改变最终位置。

排序异常要拆成两层：模型预测是否异常，预测到最终分的组合是否异常。

10.1.5 Runbook 五：日志或训练数据缺失

症状：用户能看到内容，但曝光日志、served candidates、训练样本或实验报表缺失。

排查步骤：

1. side effect 是否被启用？看 enable 条件。
2. selected_candidates 是否为空?
3. 序列化是否失败?
4. Kafka publisher 是否报错?
5. 后台 task 是否被调度?
6. topic 或下游 consumer 是否健康?
7. 是否只有 shadow/prod/某类请求受影响?
8. 对比请求量和日志量是否成比例。

这类问题不会立刻表现为 feed 失败，但会破坏长期闭环。

10.1.6 事故复盘模板

问题摘要：
用户影响：
首次发现时间：
发现方式：
影响范围：
直接原因：
根因：
为什么监控没有更早发现：
修复动作：
回滚方案：
长期防护：
需要补充的指标：
需要补充的测试：

推荐系统事故复盘必须写清楚“当前请求影响”和“数据闭环影响”。后者经常被低估。

10.2 观测性 Dashboard，怎样知道系统正在变坏

推荐系统依赖太多组件，不可能靠用户反馈才发现问题。你需要 dashboard 把候选流动、延迟、过滤、打分、缓存、side effect 全部串起来。

10.2.1 Dashboard 的核心思想

每个阶段都要记录三类信息：

- 量：输入多少、输出多少。
- 时：花了多久。
- 质：错误率、缺失率、过滤率、分数分布。

只看延迟不够，只看最终结果数也不够。推荐系统的问题经常是“结果还在，但质量已经变差”。

10.2.2 顶层面板

顶层面板应该回答：系统整体是否健康?

指标	意义
request count	流量是否异常。
execute latency P50/P95/P99	用户等待是否变慢。
final result size	是否空结果或供给不足。
error rate	整体错误趋势。
in-network/OON ratio	候选结构是否变化。
negative feedback rate	用户不满是否上升。

10.2.3 Stage 面板

每个 stage 都应该有 latency 和 size：

- query hydrators latency。
- source candidate_count。
- hydrator missing/error count。
- filter kept/removed/filter_rate。
- scorer latency 和 score missing。
- selector selected_count。
- side effect error count。

这些指标能告诉你候选在哪一段变少，延迟在哪一段变高。

10.2.4 Component 面板

stage 只告诉你哪一层有问题，component 面板告诉你具体是谁：

```
query_hydrator.name -> latency/error
source.name -> candidate_count/error
hydrator.name -> latency/missing/cache_hit
filter.name -> removed_count/filter_rate
scorer.name -> latency/error
side_effect.name -> error/latency
```

`candidate-pipeline` 中的 tracing span 已经记录了许多 component name，这是构建 dashboard 的基础。

10.2.5 分布面板

推荐系统不能只看总量，还要看分布：

- 每个 source 的候选占比。
- 每个 served_type 的最终占比。
- 每个 author 的重复度。
- score 分布。
- action probability 分布。
- filter reason 分布。
- visibility reason 分布。

分布变化往往比均值更早暴露问题。

10.2.6 告警设计

告警不要太多，但要覆盖核心失败：

- final result size 接近 0。
- execute P99 超阈值。
- 某个核心 source candidate_count 急剧下降。
- CoreData missing 激增。
- VFFilter 移除率异常。
- PhoenixScorer 错误率升高。
- served candidates Kafka side effect 失败。
- cache hit rate 突降。

告警要能指向阶段，否则值班同学只能看到“feed 坏了”，不知道从哪里查。

10.2.7 调试截图模板

事故排查时，建议固定截取：

```
1. 总请求量和总延迟
2. final result size
3. source candidate_count by source
4. filter removed_count by filter
5. hydrator missing/error by hydrator
6. scorer latency/error
7. side effect failures
8. 最近参数和发布变化
```

这组截图能支持多数复盘。

10.2.8 本节练习

1. 为 `CandidatePipeline::execute` 画一个 dashboard 草图。
2. 设计一个空结果告警，要求能区分 source 供给不足和 filter 误杀。
3. 设计一个缓存异常告警，包含 hit rate、Redis latency、cached path result size。

10.3 测试策略，推荐系统该测什么

推荐系统测试不能只测“函数返回不报错”。它要保护阶段合约、字段语义、候选数量、排序不变量和模型推理形状。本节从已有 Phoenix 测试出发，扩展到 Home Mixer 流水线应该怎样测。

10.3.1 Phoenix 测试给出的启发

`phoenix/test_recsys_model.py` 中有一组 `make_recsys_attn_mask` 测试。它们不是只测 shape，而是逐项验证语义：

- user/history 保持 causal attention。
- candidates 能看 user/history。
- candidates 能看自己。
- candidates 不能看其他 candidates。
- single candidate 和 all candidates 边界情况。

这是好测试的特征：它保护模型设计意图，而不是只保护实现细节。

10.3.2 Retrieval 测试

`phoenix/test_recsys_retrieval_model.py` 测 `CandidateTower`：

- 输出 shape 正确。
- 输出 L2 normalized。
- mean pooling 模式没有参数。
- full retrieval model 输出 user representation 和 top K。

这些测试保护 two-tower 的核心不变量：向量维度、归一化、top K 输出。

10.3.3 Pipeline 合约测试

对 `candidate-pipeline`，最重要的是合约测试：

合约	测试
Hydrator 长度一致	模拟 hydrator 少返回一个结果，确认框架返回错误并不错位更新。
Scorer 长度一致	模拟 scorer 结果长度错误，确认候选 score 不被错误写入。
Filter 顺序	两个 filter 顺序运行，第二个只看到第一个 kept。
Source 降级	一个 source 返回 Err，另一个 source 成功，最终仍有候选。
Side effect 后台	side effect 不改变当前 selected candidates。

这些测试比具体业务组件更基础，一旦破坏会影响所有 pipeline。

10.3.4 组件单元测试

每类组件都有典型测试：

- QueryHydrator：mock client 返回数据，验证 query 字段被 update。
- Source：mock downstream 返回 candidates，验证 PostCandidate 字段和 enable 条件。
- Hydrator：输入候选，mock client 返回字段，验证 update 只写自己负责字段。
- Filter：构造 kept/removed 边界样本。
- Scorer：构造 PhoenixScores，验证 weighted_score 和 final score。
- Selector：构造 score 列表，验证排序和截断。
- SideEffect：mock publisher/client，验证 payload 和 enable 条件。

10.3.5 集成式测试

推荐系统还需要小规模集成测试。可以用 mock pipeline：

```
query hydrator -> 2 sources -> 1 hydrator -> 2 filters -> 1 scorer -> selector
```

验证：

- 候选数量每阶段符合预期。
- 被过滤候选进入 filtered_candidates。
- selected 和 non_selected 正确。
- side effect 收到正确输入。

这类测试不需要真实模型和真实 Redis，但要保护数据流。

10.3.6 回归测试样本

对于线上 bug，建议把最小复现转成 fixture：

```
bug: repeated retweet appears twice
fixture:
  source A returns original tweet
  source B returns retweeting tweet
expected:
  retweet dedup filter removes one
```

每个生产事故都应该沉淀一个测试或至少一个 runbook 检查项。

10.3.7 测试不该做什么

不要把测试写成“复制实现”。例如 filter 里用了 HashSet，测试不需要关心 HashSet，只需要关心重复 tweet id 被移除。

> 不要在单元测试里依赖真实外部服务。source/hydrator/scorer 的外部 client 应该 mock。
>
> 不要只测 happy path。推荐系统的核心能力之一是部分失败时继续服务。

10.3.8 本节练习

1. 给 `DropDuplicatesFilter` 写 3 个测试用例。
2. 给 `CoreDataCandidateHydrator` 写一个 cache hit 和一个 cache miss 测试。
3. 给 `PhoenixSource` 写 enable 条件测试表。
4. 把一次空结果 runbook 转成集成测试 fixture。

10.4 错误处理和降级策略

推荐系统的外部依赖很多，错误处理是主线能力。一个成熟系统不会因为任何一个下游失败就完全不可用，也不会静默吞掉所有错误。本节整理常见降级模式。

10.4.1 Source 失败：跳过一路候选

`fetch_candidates` 对 source 结果使用 `flatten` 收集成功值。失败 source 不贡献候选，其他 source 继续。

> 适用场景：

- 多路召回互相补充。
- 单个 source 不是唯一候选来源。
- 下游短暂失败时仍可返回 feed。

风险：

- 某路 source 长期失败，用户体验变差但请求不报错。
- 如果没有 per-source candidate_count 告警，问题可能被隐藏。

10.4.2 Hydrator 失败：保留默认字段

hydrator 对单个候选返回 Err 时，`update_all` 不更新该候选。后续 filter 决定是否移除。

适用场景：

- 某些字段非关键。
- 后续有完整性 filter。
- 允许部分候选缺字段。

风险：

- 默认值被误解释为真实值。
- 安全相关字段缺失却放行。

10.4.3 Scorer 失败：分数缺失或默认

PhoenixScorer 如果 prediction 请求失败，会给每个候选返回 Err；update 时不会写 phoenix_scores。RankingScorer 可能仍运行，但 score 质量会下降。

对核心 scorer，通常需要更强监控。因为请求可能仍有结果，但排序质量已经坏了。

10.4.4 Query hydrator 失败：上下文缺失

query hydrator 失败时，对应字段保持默认。这个模式最危险，因为默认值可能与真实空值混淆。

例如空关注列表可能是“没有关注”，也可能是“社交图失败”。如果后续没有错误标记，就难以区分。

10.4.5 Side effect 失败：当前成功，未来受损

side effect 在后台执行，失败不影响当前响应。但它会影响：

- served history 去重。
- 缓存复用。
- 训练数据。
- 实验分析。
- 审计和问题排查。

side effect 必须有错误率指标和必要的重试或补偿机制。

10.4.6 降级决策表

失败点	当前请求	后续风险
单个 source	可继续	候选结构变化。
核心 query hydrator	可继续	上下文缺失，个性化下降。
core data hydrator	可继续到 filter	候选可能被完整性过滤。
PhoenixScorer	可能继续	排序质量下降。
VFFilter 数据	风险高	安全放行或误杀。
served history side effect	当前成功	重复展示。
Kafka 日志 side effect	当前成功	训练和分析缺失。

10.4.7 本节练习

1. 找一个 source，写出它失败后的 pipeline 行为。
2. 找一个 query hydrator，分析默认值是否安全。
3. 设计一个 side effect 失败告警，说明为什么不能只看当前请求成功率。

10.5 上线就绪清单

任何推荐系统改动上线前，都应该经过一份固定清单。它能减少“代码能跑但线上不可控”的问题。

10.5.1 功能清单

- 是否明确属于哪个阶段?
- 输入字段是否已经由上游保证?
- 输出字段是否只写自己负责的部分?
- 是否有 enable 条件?
- 是否有参数控制?
- 是否有默认值语义说明?
- 是否有失败策略?

10.5.2 性能清单

- 是否增加外部依赖?
- 是否在 per-candidate 循环里发单个 RPC?
- 是否支持批量请求?
- 是否有超时?
- 是否会增加 source fan-out?
- 是否会增加 model candidate count?
- 是否影响 P95/P99?

10.5.3 可靠性清单

- 下游失败时是否降级?
- 部分失败是否会错位更新候选?
- 是否记录错误率?
- 是否有重试或明确不重试的理由?
- 是否会造成后台任务堆积?
- 是否有回滚开关?

10.5.4 安全清单

- 是否可能绕过 visibility filter?
- 是否依赖安全字段默认值?
- 是否处理 block/mute/report 相关关系?
- 是否影响广告 brand safety?
- 是否记录必要审计信息?

10.5.5 实验清单

- 是否能分桶实验?
- 是否有对照组?
- 是否定义主要指标和 guardrail?
- 是否看负反馈?

- 是否看延迟和错误率?
- 是否有实验结束后的复盘计划?

10.5.6 文档清单

- README 或本书组件目录是否更新?
- 新参数是否有说明?
- runbook 是否更新?
- dashboard 是否有新指标?
- 测试 fixture 是否覆盖关键路径?

10.5.7 上线记录模板

改动名称：
阶段：
影响用户：
参数/开关：
新增依赖：
预期收益：
主要风险：
监控链接：
回滚方式：
负责人：

10.5.8 本节练习

用这份清单审查“综合项目”一节的 `RecentEngagedAuthorsSource` 设计，列出至少五个上线前必须补充的点。

11 实战工作坊

前面的章节已经给出读法，本章把读法变成练习。每个练习都要求从代码和指标推导，而不是凭经验猜测。

11.1 端到端读码工作坊

前面的章节按组件类型拆开讲。这一节反过来：给出几个真实排查场景，练习如何把 query、source、hydrator、filter、scorer、selector 和 side effect 连起来。

11.1.1 场景一：用户打开 For You，系统返回空

排查空结果时，不要先猜模型坏了。按数据流向下走：

1. 请求是否进入正确的 server 方法？
2. query hydrator 是否成功补齐用户行为、关注列表、屏蔽列表？
3. source 是否启用？每个 source 返回多少候选？
4. candidate hydrator 是否大量失败？
5. 哪个 filter 移除了最多候选？
6. PhoenixScorer 是否返回预测？
7. selector 前是否已经没有候选？
8. post-selection filter 是否把候选全部移除？

把这八步写成一张表，空结果问题通常会很快缩小范围。

位置	看什么	可能结论
Query hydration	scoring/retrieval sequence 是否存在	用户行为服务失败或新用户冷启动。
Sources	candidate_count	召回源未启用、下游失败、参数过小。
Hydration	missing/error count	元数据服务失败，候选无法通过后续 filter。
Filters	removed_per_filter	可见性、年龄、重复、社交图规则移除过多。
Scoring	score 是否写入	模型服务失败或默认分导致排序异常。
Selection	selected_count	top K 前候选不足或混排规则丢弃。

11.1.2 场景二：feed 变慢

延迟问题要区分“一个慢调用”和“一组调用的尾延迟”。如果 source 并行运行，总耗时接近最慢 source；如果 filter 顺序运行，总耗时是多个 filter 的累加。

排查顺序：

- 看 `execute` 总延迟。
- 拆 query hydrators、sources、hydrators、scorers 的 stage latency。
- 在慢 stage 内按 component name 看 latency。
- 看输入 size 是否异常变大。
- 看缓存命中率是否下降。
- 看外部依赖错误率是否上升，重试是否放大延迟。

一个常见误判是：只看平均延迟。推荐系统更关心 P95/P99，因为用户体验常被尾部依赖拖慢。

11.1.3 场景三：同一个作者出现太多

这个问题可能出现在多个层次：

- source 本身返回了大量同作者候选。
- filter 没有去掉会话或 retweet 重复。
- RankingScorer 的作者多样性参数太弱。
- Selector 或 BlenderSelector 插入规则改变了相对位置。

排查时先看候选进入 scorer 前的作者分布，再看 `apply_author_diversity` 后的分数变化，最后看 selector 输出序列。

11.1.4 场景四：新用户推荐不好

新用户问题通常不是单个模型能解决的。你要检查：

- `retrieval_sequence` 和 `scoring_sequence` 是否足够长。
- `PhoenixRetrievalNewUserHistoryThreshold` 是否触发新用户 cluster。
- 是否启用了 topic retrieval 或 fallback source。
- 关注列表是否足够支持 ThunderSource。
- OON 权重是否对新用户有特殊处理。

这类场景适合画出决策树，而不是只看最终分数。

11.1.5 场景五：曝光日志缺失

如果用户看到了内容，但日志缺失，问题通常在 side effect 链路：

- side effect 是否启用？检查 `enable` 条件。
- selected_candidates 是否为空？
- serialization 是否失败？
- Kafka client 是否发送失败？
- 后台任务是否有错误指标？

这类问题不会影响当前响应，但会影响训练数据、实验分析和审计。

11.1.6 读码模板

以后读任何新组件，都用下面模板：

```
组件名：
阶段：
输入：
输出：
是否 async：
等待的外部系统：
enable 条件：
失败策略：
写入字段：
下游依赖：
关键指标：
用户可见影响：
```

这个模板可以逼迫你把“看懂代码”变成“看懂系统行为”。

11.1.7 本节练习

1. 用上面的模板分析 `PhoenixSource`。
2. 用上面的模板分析 `CoreDataCandidateHydrator`。
3. 用上面的模板分析 `RankingScorer`。
4. 写一个空结果排查 runbook，限制在 12 步以内。

11.2 手写一个最小推荐流水线

读完真实系统后，最好自己写一个小版本。这个练习不追求性能，只追求把 source、hydrator、filter、scorer、selector 的边界刻进脑子里。

11.2.1 目标

实现一个内存版 feed：

- 两个 source：关注作者 posts、全局热门 posts。
- 一个 query hydrator：补关注列表。
- 一个 candidate hydrator：补文本和作者名。
- 两个 filter：去重、屏蔽作者。
- 一个 scorer：按兴趣词和新鲜度打分。
- 一个 selector：取 top K。
- 一个 side effect：打印 served ids。

这个小系统可以用 Python、Rust 或 TypeScript 写。语言不重要，阶段边界重要。

11.2.2 数据模型

```
Query:
  user_id
  followed_author_ids
  blocked_author_ids
  interests

Candidate:
  post_id
  author_id
  text
  created_at
  source
  score
```

source 初始只返回 `post_id`、`author_id`、`source`。hydrator 再补 `text` 和 `created_at`。这样你能练习“候选先瘦后胖”的思路。

11.2.3 伪代码

```
async def execute(query):
    query = await hydrate_query(query)

    source_results = await gather(
        followed_source(query),
        trending_source(query),
    )
    candidates = flatten(source_results)

    candidates = await hydrate_candidates(query, candidates)

    kept, removed = drop_duplicates(query, candidates)
    kept, removed2 = author_block_filter(query, kept)
    removed += removed2

    scored = await score(query, kept)
    selected = top_k(scored, k=20)
```

```
    create_task(write_served_ids(query, selected))
    return selected
```

这个伪代码就是 `CandidatePipeline::execute` 的缩小版。

11.2.4 验证用例

至少写五个测试：

1. 两个 source 返回同一 post，最终只保留一个。
2. 被屏蔽作者的 post 被移除。
3. 包含兴趣词的 post 分数更高。
4. source 之一失败时，另一个 source 的候选仍然返回。
5. 没有 score 的候选不会排在已打分候选前面。

这些测试对应真实系统中的关键不变量。

11.2.5 扩展任务

完成最小版后，再加三个功能：

- 缓存 hydrator：post metadata 命中缓存时不访问 client。
- 过滤率统计：记录每个 filter 移除多少候选。
- side effect 失败计数：即使不阻塞响应，也要记录错误。

这三个扩展能帮你理解为什么真实系统需要缓存、观测性和后台任务。

11.2.6 对照真实代码

小实验	真实项目对应
followed_source	`ThunderSource`
trending_source	`PhoenixSource` 或其他 source
metadata_hydrator	`CoreDataCandidateHydrator`
drop_duplicates	`DropDuplicatesFilter`
author_block_filter	`AuthorSocialgraphFilter`
score	`PhoenixScorer` + `RankingScorer`
top_k	`TopKScoreSelector`
write_served_ids	`ServedCandidatesKafkaSideEffect` 或 seen ids side effect

11.2.7 本节练习

1. 用你熟悉的语言实现这个最小 pipeline。
2. 给每个阶段打印输入数量和输出数量。
3. 故意让一个 source 抛错，验证系统是否仍然返回另一个 source 的候选。
4. 把 hydrator 改成随机少返回一个结果，观察为什么真实框架要做长度检查。

11.3 综合项目，做一次端到端代码审计

最后一个实践项目：选择一次假想改动，按线上工程标准完成设计、实现、验证和回滚计划。你不需要真的改生产系统，但要用真实代码结构思考。

11.3.1 项目题目

假设你要新增一个 source：`RecentEngagedAuthorsSource`。它从用户最近互动过的作者里取新帖子，作为 Thunder 和 Phoenix 的补充。

目标：

- 对历史互动作者做轻量召回。
- 不替代 Thunder 和 Phoenix。
- 不显著增加 P95 延迟。
- 能被参数开关控制。
- 能记录 source candidate_count 和错误率。

11.3.2 设计文档模板

```
背景：
目标：
非目标：
输入字段：
输出 candidate 字段：
外部依赖：
enable 条件：
最大候选数：
超时策略：
失败策略：
需要的 hydrator：
可能被哪些 filter 移除：
scoring 是否复用现有 PhoenixScorer：
selector 影响：
side effect 影响：
指标：
实验方案：
回滚方案：
```

这个模板强迫你从系统角度思考，而不是只写一个 source 文件。

11.3.3 代码落点

可能需要修改或新增：

- `home-mixer/sources/recent_engaged_authors_source.rs`（新增文件）
- `home-mixer/sources/mod.rs`
- `home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs`
- 参数定义文件
- mock client 或测试 fixture
- README 或本书组件目录

如果 source 需要新的 query 字段，还要新增 query hydrator。不要让 source 自己偷偷访问太多上下文服务，否则边界会变乱。

11.3.4 验证计划

至少做五类验证：

1. unit test：enable 条件。
2. unit test：外部 client 返回候选后，PostCandidate 字段正确。
3. failure test：client 失败时 source 返回 Err，pipeline 仍能使用其他 source。
4. integration-like test：加入 pipeline 后 candidate_count 增加但 filter 正常工作。

5. metrics test：source name、candidate_count、error 能被观测。

如果没有测试框架，也要写清楚人工验证步骤和日志检查方法。

11.3.5 实验计划

实验不要只看总体互动。至少分桶：

- 新用户 vs 老用户。
- 关注数少 vs 关注数多。
- in-network only vs For You。
- 高活跃 vs 低活跃。
- source 贡献候选数量。

主要指标：

- final engagement。
- negative feedback。
- source candidate_count。
- latency P95/P99。
- duplicate rate。
- author diversity。

11.3.6 风险清单

可能风险：

- 最近互动作者过度集中，降低多样性。
- source 返回过多旧内容。
- 和 Thunder 重叠高，增加去重成本。
- 外部依赖慢，拉高 source stage latency。
- 新用户没有历史互动，source 贡献为 0。
- 负反馈用户被错误解释为“互动作者”。

每个风险都要对应一个缓解方案，例如 max_results、年龄过滤、去重、超时、负反馈排除。

11.3.7 复盘要求

上线后要回答：

- 这个 source 实际贡献了多少最终 selected candidates?
- 它带来的候选有多少被 filter 移除?
- 它和 Thunder/Phoenix 重叠多少?
- 它对延迟的影响是多少?
- 它对互动和负反馈的影响是多少?
- 是否对某些用户分桶明显更好或更差?

如果回答不了这些问题，说明观测性不足。

11.3.8 本节交付物

完成项目时应产出：

- 一页设计文档。
- source 代码或伪代码。
- 测试计划。

- 指标列表。
- 实验方案。
- 回滚方案。
- 复盘模板。

这比“写出能编译的代码”更接近真实推荐系统工作。

11.4 练习工作簿

这一节集中给出可执行练习。建议读者每学完一组章节，就做对应练习。目标不是背答案，而是训练“从代码证据推导系统行为”的能力。

11.4.1 练习 1：画出端到端流程

输入：用户打开 For You。 要求：画出从 gRPC request 到 final FeedItem 的流程图。

必须包含：

- QueryBuilder
- ScoredPostsServer
- PhoenixCandidatePipeline
- ForYouCandidatePipeline
- side effects

检查标准：能说清楚哪一层返回 PostCandidate，哪一层返回 FeedItem。

11.4.2 练习 2：追踪 `has_cached_posts`

用搜索工具找到 `has_cached_posts` 的所有读写位置。写出：

- 谁写它。
- 谁读它。
- true 时哪些 source 跳过。
- true 时哪些 side effect 跳过。
- 它如何改变延迟和新鲜度。

11.4.3 练习 3：模拟 filter pipeline

构造 10 个候选：

- 2 个重复 tweet id。
- 1 个 author_id=0。
- 2 个 blocked author。
- 1 个 visibility drop。

手动计算每个 filter 后剩余多少候选。

11.4.4 练习 4：手算分数

给三个候选：

```
A fav=0.2 reply=0.01 dwell=0.3 not_interested=0.01
B fav=0.1 reply=0.08 dwell=0.5 not_interested=0.02
C fav=0.05 reply=0.02 dwell=0.2 not_interested=0.10
```

自定义权重，计算排序。然后把 not_interested 权重加大，观察变化。

11.4.5 练习 5：设计 source

设计一个新 source。写出：

- enable 条件。
- query 输入。
- 外部依赖。
- 返回候选字段。
- 失败策略。
- 指标。

不要写代码也可以，但设计必须能放进 `CandidatePipeline`。

11.4.6 练习 6：写一份空结果 Runbook

限制 12 步以内。必须覆盖：

- query hydration
- source candidate count
- hydration missing
- filter removed count
- scorer score missing
- selector selected count

11.4.7 练习 7：审查安全默认值

找三个安全或可见性相关字段，分析 None/false/empty 的语义。判断默认值是否安全，并写出需要的监控。

11.4.8 练习 8：读一个 side effect

选择一个 side effect，回答：

- enable 条件。
- 输入 selected 还是 non_selected。
- 写哪个外部系统。
- 当前失败是否影响响应。
- 后续失败影响什么闭环。

11.4.9 练习 9：本地 Phoenix demo

阅读 `phoenix/run_pipeline.py`，解释：

- artifacts 目录有哪些文件。
- retrieval 如何得到 top K。
- ranking 如何得到 action probability。
- 本地 weighted score 和线上 RankingScorer 有什么区别。

11.4.10 练习 10：写复盘

假设 `PhoenixSource` 因 retrieval_sequence 缺失导致 OON 候选下降。写一份复盘：

- 影响范围。
- 根因。
- 为什么监控没发现。
- 修复。

- 长期防护。

11.5 调试场景题库

这一节提供更多场景题。每个场景都要求你用本书的方法定位问题，而不是凭直觉猜。

11.5.1 场景 1：OON 内容突然消失

现象：For You 里几乎全是关注网络内容。

优先检查：

- `PhoenixSource::enable` 是否为 false。
- `retrieval_sequence` 是否缺失。
- `PhoenixRetrievalInferenceClusterId` 是否变化。
- `PhoenixMaxResults` 是否被调小。
- `in_network_only` 是否被 viewer data 或请求字段置 true。
- OON weight 是否被压得太低。

结论格式：

```
供给层：PhoenixSource 返回量
排序层：OON 候选分数
选择层：OON 入选数量
最终占比：selected 中 OON ratio
```

11.5.2 场景 2：视频内容突然变少

检查：

- `exclude_videos` 是否为 true。
- `VideoDurationCandidateHydrator` 是否失败。
- `VideoFilter` 移除率。
- `vqv_weight` 和视频时长阈值。
- 视频候选 source 返回量。

这类问题可能发生在 source、hydrator、filter、scorer 任一层。

11.5.3 场景 3：缓存命中后质量下降

检查：

- `has_cached_posts` 是否 true。
- cached posts 数量是否刚好超过阈值。
- 缓存 TTL 是否过长。
- cached candidates 是否经过必要 post-selection filter。
- served history 是否仍然生效。
- 缓存写入是否包含 non_selected 高分候选。

缓存路径要同时看延迟收益和新鲜度损失。

11.5.4 场景 4：新用户结果空

检查：

- followed_user_ids 数量。
- retrieval/scoring sequence 长度。
- new user cluster 是否启用。

- topic retrieval 是否有 topic ids。
- Thunder 是否有关注网络候选。
- cached path 是否误启用。

新用户问题通常需要 fallback source 和特殊权重。

11.5.5 场景 5：负反馈增加

检查：

- not_interested/block/mute/report 权重。
- Phoenix 负反馈预测分布。
- 某个 source 是否引入高风险候选。
- VFFilter 和 safety label 是否异常。
- OON weight 是否过高。
- 广告或 prompt 插入是否改变用户体验。

负反馈问题不能只看模型，还要看供给和混排。

11.5.6 场景 6：P99 延迟升高但平均值正常

检查：

- 哪个 stage P99 高。
- source/hydrator fan-out 是否变多。
- Redis cache miss 是否升高。
- Thunder rejected requests 是否变化。
- PhoenixScorer 是否 fallback。
- side effect 是否意外阻塞当前路径。

P99 问题通常来自尾部依赖或容量边界。

11.5.7 场景 7：实验指标无法解释

检查：

- 分桶是否正确。
- feature switch 是否按预期命中。
- decider override 是否生效。
- 日志 side effect 是否完整。
- selected/non_selected 是否都被记录。
- 实验组和对照组是否走了相同缓存策略。

实验异常经常是数据链路问题，不一定是策略本身。

11.5.8 本节练习

选择三个场景，写出 10 步以内的排查计划，并标明每一步需要看的文件或指标。

12 新手读码工具箱

最后一章收束为通用能力：怎样读 Rust trait、怎样读 Python/JAX 模型、怎样识别常见误区，以及哪些设计模式值得保留。

12.1 给新手的 Rust 读法

这个项目的服务层大量使用 Rust。新手如果只会 Python 或 JavaScript，第一次读 trait、generic、Arc、Box、async_trait 会很吃力。本节只讲读这个项目需要的最小 Rust 知识。

12.1.1 Trait 是组件合约

`Source<Q, C>`、`Hydrator<Q, C>`、`Filter<Q, C>`、`Scorer<Q, C>` 都是 trait。你可以先把 trait 理解成“组件必须实现的方法列表”。

例如 Source：

```
async fn source(&self, query: &Q) -> Result<Vec<C>, String>;
```

任何实现 Source 的组件，都能被 pipeline 当作候选源运行。pipeline 不关心具体是 ThunderSource 还是 PhoenixSource，只关心它能返回 `Vec<C>`。

12.1.2 泛型 Q 和 C

`Q` 是 query 类型，`C` 是 candidate 类型。这样同一个 `candidate-pipeline` 框架能服务不同业务：

- `ScoredPostsQuery + PostCandidate`
- `ScoredPostsQuery + FeedItem`

新手读泛型时，不要被语法吓到。回到具体 pipeline 实例，把 Q/C 替换成真实类型即可。

12.1.3 `Box<dyn Trait>`

`Vec<Box<dyn Source<ScoredPostsQuery, PostCandidate>>>` 表示一个列表，里面可以放不同具体类型的 source，只要它们都实现同一个 trait。

这让 `PhoenixCandidatePipeline` 可以把 `ThunderSource`、`PhoenixSource`、`CachedPostsSource` 放在同一个 `sources` 列表里。

12.1.4 `Arc`

`Arc<T>` 是线程安全引用计数指针。项目里很多 client 用 `Arc` 包起来，因为多个组件或异步任务需要共享同一个 client。

例如：

```
Arc<dyn PhoenixPredictionClient + Send + Sync>
```

可以理解成“可在线程间共享的 Phoenix client 接口”。

12.1.5 `async_trait`

Rust 原生 async trait 有限制，因此项目使用 `tonic::async_trait`。你会看到：

```
#[async_trait]
impl Source<ScoredPostsQuery, PostCandidate> for PhoenixSource {
    async fn source(&self, query: &ScoredPostsQuery) -> Result<Vec<PostCandidate>, String>
{
        ...
    }
}
```

读法很简单：这个组件实现了异步 source 方法。

12.1.6 `Result` 和 `Option`

`Result<T, String>` 表示成功或错误。`Option<T>` 表示有或没有。

常见模式：

```
.await.map_err(|e| e.to_string())?
```

表示等待外部调用，失败时把错误转成字符串并返回。

```
candidate.score.unwrap_or(f64::NEG_INFINITY)
```

表示 score 缺失时使用默认值。读默认值时要问业务语义。

12.1.7 `..Default::default()`

构造部分字段时常见：

```
PostCandidate {
    tweet_id,
    author_id,
    ..Default::default()
}
```

这表示其他字段使用默认值。推荐系统里这很常见，因为每个阶段只写自己负责的字段。

12.1.8 `partition`

filter 中常见：

```
let (kept, removed) = candidates.into_iter().partition(|c| predicate(c));
```

注意返回顺序取决于 predicate。`partition(|c| c.author_id != 0)` 得到 `(kept, removed)`；`partition(|c| should_drop(...))` 得到的第一个是 removed，需要看变量名。

12.1.9 本节练习

1. 把 `Source<Q, C>` 中的 Q/C 替换成 `ScoredPostsQuery/PostCandidate`，手写一遍签名。
2. 找一个 `..Default::default()`，列出哪些字段被默认化。
3. 找三个 `unwrap_or`，解释默认值的业务含义。

12.2 给新手的 Python/JAX 读法

Phoenix 模型用 Python、JAX 和 Haiku 写。新手不必先成为 JAX 专家，但需要看懂几个模式：函数式模型、参数树、张量 shape、jit 风格和 numpy/JAX 互转。

12.2.1 JAX 数组和 numpy 数组

代码里同时出现 `np` 和 `jnp`：

- `numpy` 常用于文件加载、预处理、CPU 侧数组操作。
- `jax.numpy` 用于模型前向和可加速计算。

`run_pipeline.py` 中常见转换：

```
jnp.asarray(user_hashes)
np.asarray(user_repr[0])
```

读法：进入模型前转成 JAX 数组；拿出来做普通 numpy 操作时转回 numpy。

12.2.2 Haiku transform

Haiku 模型通常写成函数：

```
def rank_forward(b, e):
    return rank_model_config.make()(b, e)

rank_fn = hk.without_apply_rng(hk.transform(rank_forward))
```

`transform` 把函数变成可初始化、可应用的模型。`apply(params, ...)` 使用已有参数做前向推理。

本地 demo 加载导出的 params，所以主要用 `apply`。

12.2.3 参数树

`load_model_params` 把 `.npz` key 拆成：

```
module_path / param_name
```

再转成 Haiku params dict。模型代码里 `hk.get_parameter("name", shape, ...)` 会从参数树取对应权重。

如果参数名不匹配，模型无法加载或前向失败。这就是 config、代码和 checkpoint 必须一致的原因。

12.2.4 Shape 是第一层语义

JAX 模型读法从 shape 开始。Phoenix 常见 shape：

```
B: batch size
S: history sequence length
C: candidate count
D: embedding dimension
A: action count

user_hashes: [B, num_user_hashes]
history_post_hashes: [B, S, num_item_hashes]
candidate_post_hashes: [B, C, num_item_hashes]
embeddings: [B, 1 + S + C, D]
logits: [B, C, A]
```

看懂 shape，模型主线就清楚一半。

12.2.5 Mask 测试为什么重要

`test_recsys_model.py` 里对 attention mask 写了很多测试。因为 mask 的 shape 正确不代表语义正确。候选之间如果误相互 attention，模型仍可能跑通，但排序稳定性被破坏。

机器学习系统里，最危险的 bug 往往不是崩溃，而是悄悄改变训练或推理语义。

12.2.6 向量归一化

Retrieval 测试检查 CandidateTower 输出 L2 norm 接近 1。这是因为 retrieval 用点积做相似度。归一化后，点积更接近 cosine similarity，避免向量长度主导结果。

如果忘记归一化，top K 可能偏向向量范数大的候选，而不是方向相似的候选。

12.2.7 本节练习

1. 在 `recsys_model.py` 中标注 B/S/C/D/A 对应位置。
2. 解释 `hk.transform` 和 `apply(params, ...)` 的关系。
3. 运行或阅读 mask 测试，说明每个测试保护哪个推荐语义。
4. 解释 retrieval 为什么要测试 L2 normalized。

12.3 扩展术语表

本节把前面反复出现的术语集中整理。它适合在读源码时作为速查表。

Candidate Pipeline：把推荐请求拆成 query hydration、source、hydration、filter、scorer、selector、side effect 的通用框架。

Query：本次请求的上下文，包括用户、设备、场景、参数、历史行为、缓存状态等。

Candidate：候选内容。它可能来自不同 source，只有经过过滤和选择后才会展示。

Hydration：补字段的过程。Query hydration 补请求上下文，candidate hydration 补候选信息。

Source：候选来源，例如 Thunder、Phoenix Retrieval、缓存、广告系统。

Filter：硬约束或规则移除阶段，把候选分成 kept 和 removed。

Scorer：写入排序信号的组件，可以是模型调用，也可以是加权组合或多样性调整。

Selector：根据分数或混排规则选择最终候选。

Side effect：响应后或响应边界附近写日志、缓存、状态、实验数据的操作。

In-network：来自用户关注网络的内容，典型 source 是 Thunder。

Out-of-network：关注网络之外发现的内容，典型 source 是 Phoenix Retrieval。

Retrieval：从大规模内容中快速找一批可能相关候选的阶段。

Ranking：对较小候选集做精细预测和排序的阶段。

Two-tower：分别编码用户和候选到同一向量空间，用点积或相似度做召回的模型结构。

Candidate isolation：排序 transformer 中候选不能互相 attention，保证候选分数更稳定。

Product surface：行为或请求发生的产品场景。

Served type：候选来源或展示类型的标记，用于响应、日志和分析。

Feature switch：按用户、地区、客户端等动态求值的参数系统。

Decider：用于实验或开关控制的运行时决策系统。

Fallback：主路径失败时使用替代路径或默认值继续服务。

Degradation：部分能力不可用时降低质量但保持服务可用。

Tail latency：P95/P99 等尾部延迟，常决定用户体感。

Cache hit：缓存中找到可用数据。

Cache miss：缓存没有找到数据，需要访问下游或实时计算。

TTL：缓存存活时间。过长影响新鲜度，过短降低命中率。

Visibility filtering：基于安全、删除、策略、屏蔽等规则判断内容是否可展示。

Brand safety：广告展示上下文是否符合品牌安全要求。

Served history：记录用户近期看到的 feed item，用于去重和上下文。

Impression：一次内容展示事实，是训练和分析的重要输入。

Negative feedback：用户不感兴趣、屏蔽、静音、举报等负向信号。

Calibration：模型预测概率与真实发生频率的一致程度。

A/B experiment：把用户分到不同策略，比较线上指标。

Runbook：生产问题的标准排查步骤。

12.3.1 使用方法

遇到不懂的术语时，不要只背定义。回到代码里找：

- 它是哪一阶段的概念？
- 对应哪个字段或组件？
- 是否影响当前响应？
- 是否影响后续闭环？

12.4 按角色选择阅读路径

不同读者关注点不同。推荐算法新手、后端工程师、机器学习工程师、数据分析师、产品经理读这本书时，不需要完全同一条路线。

12.4.1 后端新手

推荐路线：

1. 前言和项目地图。
2. 服务化推荐的分布式最小模型。
3. Candidate Pipeline。
4. Query hydration、sources、hydrators、filters。
5. 错误处理和降级。
6. 生产 runbook。

重点问题：

- 哪些调用是并行的？
- 哪些阶段必须顺序？
- 失败时系统如何继续？
- 如何通过指标定位问题？

先不要深挖 Phoenix 数学。你需要先掌握服务链路。

12.4.2 推荐算法新手

推荐路线：

1. 项目地图。
2. Phoenix Retrieval。
3. Phoenix Ranking。
4. Phoenix 内部。
5. 训练和评估。
6. 参数和实验。
7. 数据闭环。

重点问题：

- 召回和排序为什么分开？
- 多行为预测如何变成最终分？
- 训练 label 从哪里来？
- 离线指标和线上实验有什么差异？

12.4.3 机器学习工程师

推荐路线：

1. Phoenix 本地实战。
2. Python/JAX 读法。
3. Phoenix 内部。
4. Retrieval/Ranking 实验课。
5. Scorer 深度导读。
6. 训练和评估。

重点问题：

- 模型输入 shape 是否正确?
- candidate isolation 是否被保持?
- artifacts、config、hash 参数是否一致?
- 模型输出如何被业务权重使用?

12.4.4 Feed 产品或策略同学

推荐路线：

1. 项目地图。
2. Scoring 和 Selection。
3. Selector 和混排。
4. 参数、开关和实验。
5. 数据闭环。
6. 生产 runbook。

重点问题：

- 分数代表什么产品目标?
- 负反馈如何进入目标函数?
- 广告和非帖子 item 如何插入?
- 实验应该看哪些指标?

12.4.5 数据分析师

推荐路线：

1. Side effects 和观测性。
2. 数据闭环。
3. 训练和评估。
4. 观测性 Dashboard。
5. 生产 runbook。

重点问题：

- 展示日志在哪里产生?
- selected 和 non_selected 有什么区别?
- 训练样本可能有什么偏差?
- 指标分桶应该如何设计?

12.4.6 安全和可见性相关读者

推荐路线：

1. Filtering。
2. Filter 深度导读。
3. 安全和可见性。
4. Grox 深入。
5. 生产 runbook。

重点问题：

- 哪些内容是硬约束？
- visibility_reason 如何被消费？
- None/false 默认值是否安全？
- 内容理解延迟如何影响线上过滤？

12.4.7 读者自测

如果你能回答下面问题，就说明入门主线已经掌握：

- 为什么 source 可以并行，而 filter 通常顺序？
- cached posts true 时 pipeline 发生了什么？
- Phoenix Retrieval 和 Phoenix Ranking 的输入输出分别是什么？
- 一个候选从 source 到 response 经过哪些字段变化？
- side effect 失败为什么可能影响未来训练？

12.5 常见误区

本节列出新手读推荐系统源码时最常见的误区。遇到困惑时，可以回到这里检查自己是否落入了某个误区。

12.5.1 误区一：把模型等同于推荐系统

Phoenix 很重要，但推荐系统还包括 query hydration、source、hydrator、filter、selector、side effect、缓存、实验和监控。模型只能解释一部分行为。

修正方法：从 `CandidatePipeline::execute` 看端到端，而不是从模型文件开始。

12.5.2 误区二：把所有外部调用都当成顺序慢

单次外部调用可能是顺序等待；fan-out 收集一组 future 则是并行等待。判断延迟要看依赖关系、并发结构、超时和下游容量。

修正方法：标出每个组件依赖的外部系统，并判断它是否和其他调用并行、是否有超时和降级。

12.5.3 误区三：把空列表当成真实空

空关注列表、空缓存、空行为序列可能是真实空，也可能是上游失败后的默认值。

修正方法：追踪字段写入者和错误处理，确认空值语义。

12.5.4 误区四：把 filter 当成模型质量问题

候选被过滤不代表模型不喜欢，而可能是删除、可见性、屏蔽、重复、年龄、订阅资格等硬约束。

修正方法：看 removed_per_filter，不要只看最终 selected。

12.5.5 误区五：忽略 side effect

side effect 不阻塞当前响应，但影响缓存、去重、训练数据和实验分析。忽略 side effect 会看不懂长期问题。

修正方法：每次分析请求，都问有哪些数据被写回。

12.5.6 误区六：认为 top K 越大越好

召回更深可以提高覆盖，但也增加 hydration、filter、scoring 成本，并可能带来更多噪声。

修正方法：把 top K 和过滤率、延迟、最终 result size 一起看。

12.5.7 误区七：只看平均延迟

用户体验常被 P95/P99 拖慢。并行 fan-out 系统尤其容易受最慢依赖影响。

修正方法：始终看尾延迟，并按 component 拆分。

12.5.8 误区八：把 None 都当安全

有些 None 可以默认保留，有些 None 应该保守处理。安全和可见性字段尤其不能随便默认放行。

修正方法：对每个 Option 字段写默认值语义。

12.5.9 误区九：把权重当简单旋钮

权重改变的是产品目标。提高某个正反馈可能同时提高负反馈或降低多样性。

修正方法：任何权重变更都配实验和分桶指标。

12.5.10 误区十：只看 happy path

真实系统里失败是常态。只看成功路径无法理解降级、缓存和监控设计。

修正方法：每读一个组件，都问失败时发生什么。

12.5.11 自查清单

我是否知道这个字段从哪里来？
我是否知道这个组件失败后会怎样？
我是否知道这个默认值的业务含义？
我是否知道这个阶段的输入输出数量？
我是否知道哪个指标能验证它？

12.6 推荐系统设计模式

本项目中有一些可复用设计模式。理解它们，可以帮助你阅读其他推荐系统。

12.6.1 分阶段流水线

把请求拆成固定阶段：hydrate query、source、hydrate candidate、filter、score、select、side effect。

收益：

- 组件边界清晰。
- 易于并行化。
- 易于观测。
- 易于复用。

代价：

- 字段在多个阶段逐步变化，需要清晰数据模型。
- 组件太多时需要目录和规范。

12.6.2 只写自己负责的字段

hydrator 和 scorer 返回默认 candidate，只填自己负责的字段，再由 update 合并。

收益：

- 避免组件互相覆盖。
- 便于局部失败。
- 便于测试。

风险：

- 默认值语义必须清楚。
- update 漏字段会导致下游缺数据。

12.6.3 并行 fan-out + 顺序决策

query hydrator、source、hydrator 常并行；filter、scorer 常顺序。

原则：

- 只读同一输入的独立组件可以并行。
- 会改变候选集合或依赖前一步输出的组件顺序执行。

12.6.4 缓存作为可降级路径

cached posts 不是旁路返回，而是重新接入 source 阶段。这样后续 filter/selector/side effect 仍能复用。

收益：

- 降延迟。
- 降下游压力。
- 保持 pipeline 结构统一。

12.6.5 硬约束后置保护

即使缓存命中或模型分高，也要经过可见性和安全过滤。这是推荐系统的硬边界。

12.6.6 后台 side effect

日志、缓存、served history 放后台执行，保护当前响应延迟。

风险是未来闭环受损，所以必须有观测性。

12.6.7 参数化策略

用 feature switches 和 decider 控制数量、权重、cluster、开关，让策略可以实验和回滚。

风险是路径变多，所以需要参数记录和 dashboard。

12.6.8 本节练习

1. 在项目中找出每个设计模式的一个例子。
2. 选择一个模式，说明它的收益和风险。
3. 想一想：如果没有分阶段流水线，Home Mixer 会变成什么样?

12.7 推荐系统反模式

反模式是看起来方便、长期会伤害系统的做法。本节列出本项目努力避免的问题。

12.7.1 在 hydrator 里删除候选

问题：hydrator 应该保持输入输出长度一致。如果它偷偷删除候选，后续 update 可能错位。

正确做法：hydrator 补字段，filter 删除候选。

12.7.2 在 source 里做太多业务过滤

问题：source 如果做了大量过滤，后续无法观测候选为什么消失。

正确做法：source 只做必要约束，把可解释过滤放到 filter 阶段。

12.7.3 把安全当成排序权重

问题：安全和可见性是硬约束，不应该靠模型低分解决。

正确做法：安全信息 hydrator + visibility filter。

12.7.4 没有 enable 条件

问题：所有组件对所有请求运行，会浪费资源，还可能在不适用场景产生错误。

正确做法：明确 enable 条件，并记录 disabled 组件。

12.7.5 默认值语义不清

问题：空列表、None、0、false 混用，导致失败和真实空值不可区分。

正确做法：字段文档化，必要时增加错误标记或指标。

12.7.6 缺少 per-component 指标

问题：只知道总结果坏了，不知道哪一段坏。

正确做法：stage + component 双层指标。

12.7.7 把 side effect 当不重要

问题：当前响应成功掩盖日志、缓存、训练数据失败。

正确做法：side effect 后台执行，但错误必须打点。

12.7.8 调参数不做实验

问题：权重和阈值改变系统目标，容易造成生态和体验副作用。

正确做法：小流量实验、guardrail、回滚方案。

12.7.9 本节练习

1. 选择三个反模式，说明项目里哪个设计避免了它。
2. 设计一个“坏 hydrator”例子，并说明会造成什么 bug。
3. 解释为什么 source 不应该吞掉所有过滤原因。

13 附录 A：代码索引和审校清单

附录不再承载主叙事，而是用于回查和维护。读者完成正文后，可以用这里的索引、审校清单和维护指南检查自己的理解。

13.1 代码路径索引

这是本书常用代码路径的速查索引。遇到问题时，可以从这里跳到对应文件。

13.1.1 框架层

路径	用途
`candidate-pipeline/candidate_pipeline.rs`	pipeline 主执行流程。
`candidate-pipeline/query_hydrator.rs`	query hydrator 合约。
`candidate-pipeline/source.rs`	source 合约。
`candidate-pipeline/hydrator.rs`	hydrator 和 cached hydrator 合约。
`candidate-pipeline/filter.rs`	filter 合约。
`candidate-pipeline/scorer.rs`	scorer 合约。
`candidate-pipeline/selector.rs`	selector 合约。
`candidate-pipeline/side_effect.rs`	side effect 合约。

13.1.2 Home Mixer

路径	用途
`home-mixer/server.rs`	gRPC 入口和 QueryBuilder。
`home-mixer/scored_posts_server.rs`	帖子候选 pipeline 运行和响应转换。
`home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs`	内层帖子候选 pipeline 配置。
`home-mixer/candidate_pipeline/for_you_candidate_pipeline.rs`	外层 feed item pipeline 配置。
`home-mixer/models/query.rs`	ScoredPostsQuery 数据模型。
`home-mixer/models/candidate.rs`	PostCandidate 数据模型。

13.1.3 Phoenix

路径	用途
`phoenix/recsys_retrieval_model.py`	two-tower retrieval model。
`phoenix/recsys_model.py`	ranking model。
`phoenix/grok.py`	Transformer 和 recommendation attention mask。
`phoenix/run_pipeline.py`	本地 retrieval + ranking demo。
`phoenix/runners.py`	模型参数和 runner 工具。
`phoenix/test_recsys_model.py`	ranking mask 和位置测试。
`phoenix/test_recsys_retrieval_model.py`	retrieval model 测试。

13.1.4 Thunder 和 Grox

路径	用途
`thunder/posts/post_store.rs`	实时帖子内存索引。
`thunder/thunder_service.rs`	in-network gRPC 服务。
`thunder/kafka/`	Kafka ingestion 相关。
`grox/engine.py`	内容理解任务执行入口。
`grox/plans/plan_master.py`	多 plan 并行执行和合并。
`grox/plans/plan.py`	有依赖的任务图执行。

13.1.5 使用方式

遇到 bug 时，先判断阶段，再去索引找入口文件。不要从全局搜索结果随便跳，因为推荐系统同名概念很多。

13.2 最终审校清单

这本书扩写到较完整版本后，需要做一次工程化审校。审校目标不是文风润色，而是确认它真的能作为学习材料使用。

13.2.1 编译检查

必须通过：

```
typst compile docs/recsys-beginner-book/main.typ docs/recsys-beginner-book/book.pdf
```

如果编译失败，先修 Typst 语法，再谈内容质量。

13.2.2 文件引用检查

用 `rg` 或 `find` 确认书中引用的路径存在：

- `candidate-pipeline/candidate_pipeline.rs`
- `home-mixer/candidate_pipeline/phoenix_candidate_pipeline.rs`
- `home-mixer/candidate_pipeline/for_you_candidate_pipeline.rs`
- `phoenix/recsys_model.py`
- `phoenix/recsys_retrieval_model.py`
- `thunder/posts/post_store.rs`
- `grox/engine.py`

如果代码文件移动，书中路径必须更新。

13.2.3 章节覆盖检查

完整教程版应覆盖：

- 项目地图。
- 服务化推荐的分布式最小模型。
- CandidatePipeline 阶段。
- QueryHydrator、Source、Hydrator、Filter、Scorer、Selector、SideEffect。
- Phoenix Retrieval 和 Ranking。
- Thunder 和 Grox。
- 缓存、状态、可见性、安全。
- 参数、实验、数据闭环。

- 生产 runbook。
- 练习和综合项目。

缺任何一项，都会影响新手形成完整系统图。

13.2.4 新手友好检查

每章至少应该回答：

- 这个阶段解决什么问题?
- 输入是什么?
- 输出是什么?
- 关键代码在哪里?
- 失败会怎样?
- 如何验证?

如果一章只有概念，没有代码路径和练习，应该补。

13.2.5 技术准确性检查

重点检查：

- source/hydrator/filter/scorer 的职责是否混淆。
- cached posts 路径是否描述准确。
- candidate isolation 是否解释准确。
- side effect 是否说明不阻塞当前响应。
- safety/visibility 默认值是否没有过度承诺。
- 本地 Phoenix demo 和线上 Home Mixer 是否区分清楚。

13.2.6 页数和结构检查

目标是约 120 页，不需要精确等于 120。更重要的是：

- 章节结构清晰。
- 每章长度适中。
- 表格和代码块不过度拥挤。
- 目录能反映学习路径。
- PDF 可读。

13.2.7 后续出版检查

如果要把它变成更正式的书，还需要：

- 统一章节编号和文件名。
- 增加图示。
- 增加术语索引。
- 增加代码引用行号或版本号。
- 增加 changelog。
- 对中文表述做二次编辑。

13.2.8 本节练习

1. 按本清单审查一章，记录三个改进点。
2. 随机选一个代码路径，确认书中描述仍和当前代码一致。
3. 编译 PDF 后检查目录和页码是否正常。

13.3 维护这本书

这本书应该跟随项目演进。否则代码一变，教程就会从学习材料变成误导材料。本节说明如何继续维护。

13.3.1 目录原则

每章围绕一个稳定问题，而不是围绕短期实现细节：

- 请求如何构造?
- 候选从哪里来?
- 字段如何补齐?
- 哪些内容会被过滤?
- 分数如何生成?
- 状态如何写回?
- 如何排查生产问题?

即使具体组件名字变了，这些问题仍然成立。

13.3.2 更新流程

每次项目代码大改后，按下面流程维护：

1. 重新编译 Typst，确认书仍可生成。
2. 用 `rg` 检查书中提到的文件是否存在。
3. 对照 `phoenix_candidate_pipeline.rs` 更新组件目录。
4. 对照 `for_you_candidate_pipeline.rs` 更新外层 feed 章节。
5. 对照 Phoenix README 更新模型和 artifacts 章节。
6. 更新页数和 README。
7. 抽查 3 个章节的代码引用是否仍准确。

13.3.3 引用风格

书中应该优先引用路径和组件名，而不是脆弱的行号。行号很容易随代码变动失效。必要时可以在最终出版版中加入行号，但维护版以路径为主。

13.3.4 扩写标准

新增内容要满足三个条件：

- 能指向当前仓库的真实代码或明确概念。
- 能帮助新手理解输入、输出、失败策略或验证方法。
- 不只是重复 README。

如果一段内容不能回答“读者看完能做什么”，就不该加入。

13.3.5 图表 TODO

后续可以补更多图：

- Pipeline 阶段图。
- Query/Candidate 字段生命周期图。
- Cached posts 读写图。
- Candidate isolation mask 图。
- Thunder ingestion 图。
- 数据闭环图。
- 生产 dashboard 图。

Typst 可以用文本图、表格或后续引入绘图库。当前版本先保证内容和结构。

13.3.6 版本记录建议

建议在 README 里维护：

版本：
页数：
主要新增：
验证命令：
适配代码 `commit`：
已知缺口：

这能帮助后来者判断书和代码是否同步。

13.3.7 本节练习

1. 找出书中一个可能随代码变化的组件名，写出维护风险。
2. 更新组件目录时，你会优先检查哪个文件？
3. 给本书设计一个 changelog 格式。

13.4 后续路线

这本书已经覆盖了从新手到能读懂主链路的内容。后续如果继续完善，可以朝三个方向扩展。

13.4.1 方向一：更多真实组件逐文件导读

可以继续增加：

- `AgeFilter`
- `MutedKeywordFilter`
- `PreviouslyServedPostsFilter`
- `QuoteHydrator`
- `LanguageCodeHydrator`
- `PhoenixTopicsSource`
- `PhoenixMOESource`
- `VMRanker`

每个文件按同一模板讲：输入、输出、外部依赖、enable、失败策略、指标、测试。

13.4.2 方向二：图示化

当前版本以文字和表格为主。后续可以补：

- 全链路流程图。
- Query 字段生命周期图。
- Candidate 字段生命周期图。
- Candidate isolation mask 可视化。
- 缓存读写图。
- 数据闭环图。
- Dashboard 示例图。

图示能降低新手理解成本。

13.4.3 方向三：本地可运行实验

可以新增脚本或 notebook：

- 运行 Phoenix demo。

- 构造 toy candidate pipeline。
- 可视化 attention mask。
- 手算 ranking weights。
- 模拟 source failure 和 filter rate。

把练习变成可运行代码，会让学习体验更完整。

13.4.4 方向四：评估和训练深挖

当前书只做了训练评估入门。后续可以深入：

- 样本构造。
- 曝光偏差。
- 多任务损失。
- calibration。
- counterfactual evaluation。
- A/B 实验分析。

这部分适合读者已经掌握服务主链路之后学习。

13.4.5 方向五：维护自动化

可以写一个简单检查脚本：

- 检查书中引用路径是否存在。
- 编译 Typst。
- 统计页数。
- 输出章节行数。
- 检查 TODO。

这样每次代码变化后都能快速确认教程没有明显过期。

13.4.6 结束语

推荐系统入门的关键不是一次性记住所有组件，而是形成稳定读法：

```
输入是什么？
输出是什么？
等了谁？
失败怎么办？
谁会读取这个结果？
怎样验证它？
```

只要你坚持这六个问题，再大的推荐系统也能被拆开理解。

14 附录 B：复习题

14.1 复习题

这一节提供一组短问题。建议读者不看答案，先口头回答，再回到代码验证。

14.1.1 架构

1. Home Mixer 和 Phoenix 的边界是什么?
2. 为什么 For You 有内外两层 pipeline?
3. `PostCandidate` 和 `FeedItem` 的区别是什么?
4. Thunder 和 Phoenix Retrieval 分别解决什么候选问题?

14.1.2 Pipeline

1. `CandidatePipeline::execute` 的阶段顺序是什么?
2. 哪些阶段并行，哪些阶段顺序?
3. 为什么 filter 不能并行地独立删除候选?
4. side effect 为什么放在最后?

14.1.3 Query

1. `ScoredPostsQuery` 中哪些字段来自原始请求?
2. 哪些字段由 query hydrator 补齐?
3. `params` 和 `decider` 分别控制什么?
4. `has_cached_posts` 为什么是路径切换字段?

14.1.4 Candidate

1. 一个 source 最少应该写哪些字段?
2. `author_id=0` 在 CoreDataHydrationFilter 中代表什么?
3. `get_original_tweet_id` 为什么必要?
4. `score=None` 在 selector 中如何处理?

14.1.5 Phoenix

1. Retrieval 为什么用 two-tower?
2. Ranking 为什么需要 candidate isolation?
3. 多行为预测比单一 relevance 有什么优势?
4. 本地 `run_pipeline.py` 和线上 Home Mixer 差异在哪里?

14.1.6 生产

1. source 失败时 pipeline 如何继续?
2. query hydrator 失败有什么风险?
3. side effect 失败为什么影响未来?
4. 你会如何排查 final result size 突降?

14.1.7 开放题

1. 如果要新增一个 source，你会先写哪些指标?
2. 如果负反馈上升，你会从哪三层查?
3. 如果缓存命中后用户看到重复内容，可能原因有哪些?
4. 如果 P99 延迟升高但 P50 不变，说明什么?

15 附录 C：习题参考解答

15.1 练习参考答案

本附录给的是“可核对的参考思路”，不是唯一答案。推荐系统题目通常要结合代码、参数和指标判断；如果你的答案能指出代码位置、输入输出和失败影响，即使表达不同，也可以视为合格。

15.1.1 使用方式

先自己答，再看参考答案。每道题至少补三类证据：

- 代码证据：对应文件、trait、字段或函数。
- 数据证据：输入数量、输出数量、过滤率、错误率、延迟或缓存命中率。
- 推理证据：为什么这个现象会从上一阶段传到下一阶段。

只写“模型坏了”“缓存坏了”“下游慢了”都不够。合格答案要能沿着 query –> candidates –> hydrated candidates –> filtered candidates –> scored candidates –> selected candidates –> side effects 推下去。

15.1.2 工作簿练习 1：端到端流程

参考流程：

```
For You request
  -> QueryBuilder::build
  -> ForYouCandidatePipeline::execute
  -> ScoredPostsSource
  -> ScoredPostsServer::get_scored_posts / run_pipeline
  -> PhoenixCandidatePipeline::execute
  -> PostCandidate 列表
  -> candidates_to_scored_posts
  -> FeedItem::Post
  -> AdsSource / WhoToFollowSource / PromptsSource / PushToHomeSource
  -> BlenderSelector
  -> final FeedItem 列表
  -> For You side effects
```

关键边界：

- `PhoenixCandidatePipeline` 的候选类型是 `PostCandidate`，它主要服务帖子推荐。
- `ForYouCandidatePipeline` 的候选类型是 `FeedItem`，它把帖子、广告、关注推荐、prompt、push-to-home 等 item 放进同一条 feed。
- side effects 不决定当前返回顺序，但会写入日志、served history、client events 和后续闭环状态。

15.1.3 工作簿练习 2：追踪 `has_cached_posts`

写入者是 `CachedPostsQueryHydrator`。它从 Redis 读取缓存，反序列化成 `cached_posts`，当缓存条数达到阈值时设置 `has_cached_posts=true`。

典型读取者：

- `CachedPostsSource`：只有 `has_cached_posts=true` 时启用，把缓存候选重新接回 source 阶段。
- `ThunderSource`、`PhoenixSource`、`PhoenixTopicsSource`、`PhoenixMOESource`、`TweetMixerSource`：通常在缓存命中时跳过实时召回。
- `PhoenixScorer`：缓存命中时不再调用 Phoenix prediction。
- `RedisPostCandidateCacheSideEffect`、`ScoredStatsSideEffect` 等：会根据缓存路径改变写入或统计逻辑。

解释时要写清权衡：缓存路径降低实时依赖和延迟压力，但有新鲜度、重复内容和过期候选风险；实时路径更新鲜，但依赖更多外部系统。

15.1.4 工作簿练习 3：模拟 filter pipeline

如果 10 个候选的条件完全不重叠：

```
10
- 2 duplicate
剩 8
- 1 author_id=0
剩 7
- 2 blocked author
剩 5
- 1 visibility drop
剩 4
```

但真实答案不能只做减法。过滤器顺序会改变每一步输入；同一个候选可能既重复又来自 blocked author。正确写法是记录每个 filter 的 input、kept、removed，再解释总 removed 为什么可能小于各条件数量之和。

15.1.5 工作簿练习 4：手算分数

可以自定义一组权重，例如：

```
score = 1.0 * fav + 2.0 * reply + 0.5 * dwell - 3.0 * not_interested
```

代入：

```
A = 0.2 + 0.02 + 0.15 - 0.03 = 0.34
B = 0.1 + 0.16 + 0.25 - 0.06 = 0.45
C = 0.05 + 0.04 + 0.10 - 0.30 = -0.11
```

排序是 B > A > C。如果把 `not_interested` 权重从 −3.0 改成 −8.0：

```
A = 0.2 + 0.02 + 0.15 - 0.08 = 0.29
B = 0.1 + 0.16 + 0.25 - 0.16 = 0.35
C = 0.05 + 0.04 + 0.10 - 0.80 = -0.61
```

C 下降最明显。这个练习的重点不是这组权重本身，而是理解负反馈权重会改变排序目标，不能当作无害的后处理。

15.1.6 工作簿练习 5：设计 source

合格设计至少包含：

- enable 条件，例如参数开关、缓存路径、用户状态、in-network-only 约束。
- 依赖的 query 字段，例如 user id、topic ids、retrieval sequence、followed user ids。
- 外部 client、timeout 和错误处理。
- max results、去重策略和候选量指标。
- 返回字段，至少要有 `tweet_id`、`served_type`，如果 source 已知道作者或 in-network 信息，也可以提前写入。
- 失败策略，通常是返回 `Err` 并让 pipeline 跳过该 source 的候选，而不是让整个 feed 失败。

不合格设计的典型问题：在 source 里做大量安全过滤、没有 enable 条件、没有候选数量指标、把默认空列表和下游失败混在一起。

15.1.7 工作簿练习 6：空结果 Runbook

参考 10 步：

1. 确认请求是否进入 `ScoredPostsServer` 和目标 pipeline。
2. 查看 `QueryBuilder::build` 是否短路或构造默认 query。
3. 查看 query hydrator 的 enabled、error、latency。
4. 检查关键字段：`scoring_sequence`、`retrieval_sequence`、`followed_user_ids`、`has_cached_posts`。
5. 查看每个 source 的 enabled 和 candidate_count。
6. 检查 hydration 长度是否匹配，missing 或默认字段是否异常。
7. 查看每个 filter 的 input、kept、removed，找最大移除点。
8. 检查 scorer 是否写入 `score`，Phoenix prediction 是否失败。
9. 检查 selector 输入数量、selected 数量和 `score=None` 比例。
10. 检查 post-selection filter，尤其是 visibility 和 safety。

结论必须指出“第一个异常阶段”，而不是只描述最终为空。

15.1.8 工作簿练习 7：安全默认值

参考分析：

- `visibility_reason=None` 在 `VFFilter` 中会保留候选；这表示没有收到 drop reason，但如果 VF hydrator 失败率升高，它也可能掩盖安全检查缺失。
- `author_blocks_viewer=None` 不能简单等于 false；要看对应 hydrator 是否成功，以及 filter 如何解释缺失。
- `brand_safety_verdict=None` 对广告混排风险较高，必须结合 ads brand safety hydrator 和广告日志判断。

结论：默认值是否安全不是字段类型决定的，而是由“谁写、失败时是否写、filter 如何解释、有没有监控”共同决定。

15.1.9 工作簿练习 8：读一个 side effect

以 `ServedCandidatesKafkaSideEffect` 为例，答案应覆盖：

- enable 条件：是否对当前 query 启用。
- 输入：side effect input 包含 final selected 和 non-selected candidates。
- 写入：把服务过的候选事实发布到 Kafka，供分析、训练、审计或回放使用。
- 当前影响：失败通常不改变当前响应，因为 side effects 在 pipeline 最后后台执行。
- 后续影响：日志缺失会影响实验分析、训练样本、served history 或问题排查。

如果选择 Redis 缓存 side effect，则要说明它影响下一次请求是否走缓存路径。

15.1.10 工作簿练习 9：本地 Phoenix demo

`phoenix/run_pipeline.py` 是学习 Phoenix 的本地路径，不等同线上 Home Mixer。

参考流程：

1. 加载 artifacts 和模型配置。
2. 构造用户历史、候选 corpus 和 hash 输入。
3. retrieval model 输出 user representation。
4. 与 corpus representation 点积，取 top K。
5. ranking model 对 top K 预测多行为概率。
6. 本地用简单 weighted score 排序并打印结果。

差异：

- 本地 demo 不包含 Home Mixer 的 query hydrator、source enable、hydrator、filter、selector、side effect。
- 线上 `PhoenixSource` 和 `PhoenixScorer` 走服务调用，且受参数、decider、timeout、fallback 和缓存路径影响。
- 本地 weighted score 用于演示，线上 `RankingScorer` 的权重、归一化、多样性和 OON 调整更完整。

15.1.11 工作簿练习 10：复盘

参考复盘骨架：

- 影响范围：OON 候选下降，可能影响 For You 中非关注网络内容占比和探索性。
- 根因：`retrieval_sequence` 缺失导致 Phoenix retrieval source 无法正常构造请求，相关 source 返回错误或候选减少。
- 发现不足：只监控最终 result size 不够，缺少 query hydrator 字段填充率、source error、source candidate_count 分布。
- 修复：恢复 retrieval sequence hydrator 或下游依赖；增加字段缺失告警；必要时临时调低依赖该 source 的流量。
- 长期防护：为关键 query 字段建立 dashboard，把 source 级候选占比、错误率和空返回率纳入发布检查。

15.1.12 复习题参考：架构

1. Home Mixer 负责线上请求编排、pipeline 配置、候选融合和 side effects；Phoenix 负责 retrieval/ranking 模型能力和预测信号。
2. 内层 `PhoenixCandidatePipeline` 产出帖子候选和分数；外层 `ForYouCandidatePipeline` 把帖子与广告、关注推荐、prompt、push-to-home 等混成最终 feed。
3. `PostCandidate` 是帖子候选在 pipeline 内部的工作对象；`FeedItem` 是最终 feed 可以承载的展示 item，可能是帖子，也可能是广告或模块。
4. Thunder 偏实时 in-network 候选，Phoenix Retrieval 偏基于用户兴趣向量的候选召回；两者互补。

15.1.13 复习题参考：Pipeline

1. 顺序是 query hydration、dependent query hydration、source、candidate hydration、filter、scorer、selector、post-selection hydration、post-selection filter、truncate/finalize、side effects。
2. query hydrator、source、hydrator、side effect 内部可并行；filter 和 scorer 按配置顺序执行；selector 是一次决策。
3. filter 会改变下一个 filter 的输入，也要保留 removed 归因。独立并行删除会让“谁移除了候选”变得不清楚。
4. side effect 放最后，是因为它通常写日志、缓存或状态，不应阻塞当前响应；但失败会影响未来闭环，所以必须监控。

15.1.14 复习题参考：Query

1. 原始请求通常提供 user id、请求上下文、产品 surface、topic/exclude 等直接参数。
2. query hydrator 补行为序列、关注/屏蔽/静音关系、缓存候选、served history、IP、用户特征等。
3. `params` 是参数和实验配置的读取入口；`decider` 更像运行时开关或流量决策入口。
4. `has_cached_posts` 会让系统从实时召回路径切到缓存回放路径，因此影响 source、scorer 和 side effect。

15.1.15 复习题参考：Candidate

1. 一个 source 至少要写清候选 id 和类型，实际代码里常见字段是 `tweet_id`、`served_type`，有能力时再补 `author_id`、`in_network` 等。
2. `author_id=0` 在 `CoreDataHydrationFilter` 语境中表示候选缺少有效作者 id，会被当作完整性失败处理，候选不应继续。
3. `get_original_tweet_id` 用于把转帖等包装关系还原到原始内容，便于预测、去重、可见性和日志一致。
4. `TopKScoreSelector` 把 `score=None` 当作 `f64::NEG_INFINITY`，因此未打分候选不会排在正常打分候选前面。

15.1.16 复习题参考：Phoenix

1. Retrieval 用 two-tower，是为了把用户和候选投到同一向量空间，并让候选向量可以提前计算或索引。
2. Candidate isolation 避免一个候选的预测依赖同批次其他候选，减少批次组成对分数的干扰。
3. 多行为预测能同时表达点赞、回复、停留、关注、负反馈等目标，比单一 relevance 更接近 feed 的真实目标。
4. 本地 `run_pipeline.py` 演示模型输入输出；线上 Home Mixer 还包含 query hydration、source enable、外部服务调用、filter、selector、side effects、参数和降级。

15.1.17 复习题参考：生产

1. source 失败时，该 source 不贡献候选；其他成功 source 仍可继续，错误通过日志和指标暴露。
2. query hydrator 失败会让字段保持默认，风险是默认空值和真实空值混淆，后续 source/scorer 可能误判。
3. side effect 失败不一定影响当前响应，但会损害缓存、日志、训练样本、实验分析和 served history。
4. final result size 突降时，按阶段查看：query 字段填充率、source candidate_count、hydration missing、filter removed、score missing、selector selected、post-selection removed。

15.1.18 复习题参考：开放题

新增 source 的指标优先级：

- enabled_count、request_count、success/error、latency。
- candidate_count 分布、empty_count、timeout_count。
- served_type 占比、最终 selected 占比。
- 下游 filter removed 占比，避免 source 只制造噪声。

负反馈上升可以先查三层：

- 数据层：曝光、点击、not interested、report 等 label 是否正常。
- 排序层：负反馈权重、OON 调整、多样性、归一化是否变化。
- 候选层：某个 source 占比是否异常，安全/可见性过滤是否漏掉高风险内容。

缓存命中后重复内容的可能原因：

- Redis 缓存写入时包含已服务或重复候选。
- served history side effect 失败或延迟。
- cache key 维度不足，导致不同上下文复用同一批候选。
- `CachedPostsSource` 回放后缺少必要去重或过滤。

P99 升高但 P50 不变，通常说明尾部依赖变慢、超时重试、少量大用户候选量过高、缓存 miss 放大，或下游服务存在限流和排队。

15.1.19 调试场景题参考

OON 内容消失：

- 先查 `PhoenixSource`、`PhoenixTopicsSource`、`PhoenixMOESource`、`TweetMixerSource` 是否启用和返回候选。
- 再查 `retrieval_sequence` 是否缺失、Phoenix retrieval client 是否错误、OON 权重是否被调低。
- 最后查 post-selection safety 是否集中移除了 OON。

视频内容变少：

- 查 `exclude_videos`、`VideoFilter`、视频时长 hydrator、媒体 hydrator。
- 对比 source 阶段视频占比和 filter 后视频占比，定位是召回减少还是过滤增加。

缓存命中后质量下降：

- 查 `has_cached_posts` 命中率、缓存年龄、缓存写入候选数量。
- 对比缓存路径和实时路径的 source 组成、score 分布、重复率。
- 检查 Redis 写入 side effect 是否写入了过期或低质量候选。

新用户结果空：

- 查 QueryBuilder 是否识别新用户路径。
- 查 retrieval/ranking sequence 默认值、新用户 cluster、Thunder 关注列表、topic source。
- 检查 filters 是否把默认字段误判成无效字段。

P99 延迟升高：

- 先看 stage latency 分位数，不只看平均值。
- 查 source、hydrator、scorer 的慢依赖和 timeout。
- 对比 cache hit/miss、候选数量、batch size 和下游限流。

实验指标无法解释：

- 先确认实验分桶、params、decider 是否按预期生效。
- 查日志 side effect 是否完整，避免样本缺失。
- 对比每个阶段的 candidate_count 和 item type distribution，避免把数据链路问题误判成策略效果。