第 8 章 混排模块

在上一章节中，我们系统性地介绍了重排模块的设计与实现。重排模块主要聚焦于单一业务内部的序列级优化问题，其优化对象通常来源于同一推荐系统输出的候选集合，核心目标是在单一业务语义下对推荐列表进行结构优化、多样性增强以及业务策略约束下的重排序。

然而，随着推荐系统逐步演进为多业务融合的综合内容分发平台，仅在单业务层面进行序列优化已难以满足整体产品形态的需求。在实际工业场景中，一个完整的推荐流往往同时包含短视频、直播、图文、广告以及电商内容等多种异质化内容，这些内容分别由不同的推荐系统独立生成，并具有不同的优化目标与打分体系。因此，在重排之后仍然需要一个更高层级的融合模块，对来自多个业务系统的输出进行统一组织与协调。

这一层级的优化问题不再局限于单一业务内部的排序优化，而是进一步扩展到**跨业务的序列融合与全局流量分配问题**。为了解决这一问题，混排（Mixed Ranking）模块应运而生，其核心任务是在统一的展示序列中对多业务候选内容进行对齐、融合与再排序，从而在保证用户体验一致性的同时，实现不同业务之间的流量平衡与价值最大化。

因此，相较于重排模块主要关注“单业务内部的 list 优化”，混排模块则进一步上升到“多业务之间的 list 融合优化”，是推荐系统从单域优化走向多域协同的关键环节。本章将围绕混排模块的整体架构、关键技术挑战以及工业实践中的典型解决方案展开详细讨论。

8.1 多业务混排问题

混排（Mixed Ranking）模块主要负责多个不同业务推荐结果的最终融合。例如在短视频类 App 中，用户通过上下滑不断浏览内容时，看到的往往并不只是短视频本身，还可能包括直播、广告、图文、商品卡片等多种内容形态。对于一次推荐请求而言，系统需要将这些来自不同业务线的推荐结果进行统一组织与排序，从而生成最终展示给用户的推荐列表。这类来自不同业务、不同内容形态的推荐结果，通常被称为**异质化内容（Heterogeneous Content）**。从推荐系统架构的角度来看，正如第2章图2.1所示，不同业务对应的推荐系统会分别完成召回、排序等流程，并输出各自的候选列表。推荐 Router 在接收到这些业务列表之后，会将其发送至混排服务，由混排模块统一完成跨业务内容的融合与排序，最终生成用户看到的结果序列。

读到这里，读者可能会产生一个问题：

笔记 *混排模块是否是所有推荐系统都必须具备的模块？*

答案是否定的。如果一个推荐系统仅服务于单一内容形态，例如纯图文推荐系统、纯短视频推荐系统或者纯商品推荐系统，那么整个推荐链路只需要完成召回、排序和重排即可，并不一定需要单独设计混排模块。混排模块产生的根本原因在于业务形态的多样化。当平台逐渐从单一内容消费场景发展为综合内容生态，希望用户在同一个浏览场景下既能够看到短视频，也能够看到直播、图文、广告以及电商内容时，便需要引入混排模块来解决不同业务之间的流量分配、内容融合以及排序优化问题。

因此，从本质上来看，混排模块并不是为了解决单个推荐系统内部的排序问题，而是用于解决多个推荐系统之间的协同推荐问题，其核心目标是在满足各业务流量诉求的同时，为用户提供更加丰富、多样且体验一致的内容消费序列。

混排模块的整体处理流程如图 8.1 所示，主要包括混排前处理、混排序列生成与剪枝、混排多业务排序以及混排后处理四个核心环节。其中，混排前处理环节主要负责接收推荐 Router 透传的多个业务推荐列表。这里以 $\{A_1, A_2, A_3, A_4\}, \cdots, \{D_1, D_2, D_3, D_4\}$ 分别表示来自四个不同业务的排序结果。由于混排模块需要对多个推荐系统输出的结果进行统一排序，因此首先需要解决不同业务之间打分体系不一致的问题。例如短视频推荐系统输出的可能是用户时长收益分，直播推荐系统输出的可能是观看价值分，而广告系统输出的则可能是 CPM 或 eCPM 竞价分。这些分数往往来源不同、量纲不同、数值范围也存在较大差异，因此无法直接进行比较。

因此，在混排前处理阶段，通常需要对各业务的排序分进行校准（Calibration）与归一化（Normalization）处理，将不同业务的打分映射到统一的评分空间，例如归一化到 $[0, 1]$ 区间。只有完成分数对齐之后，不同业务的

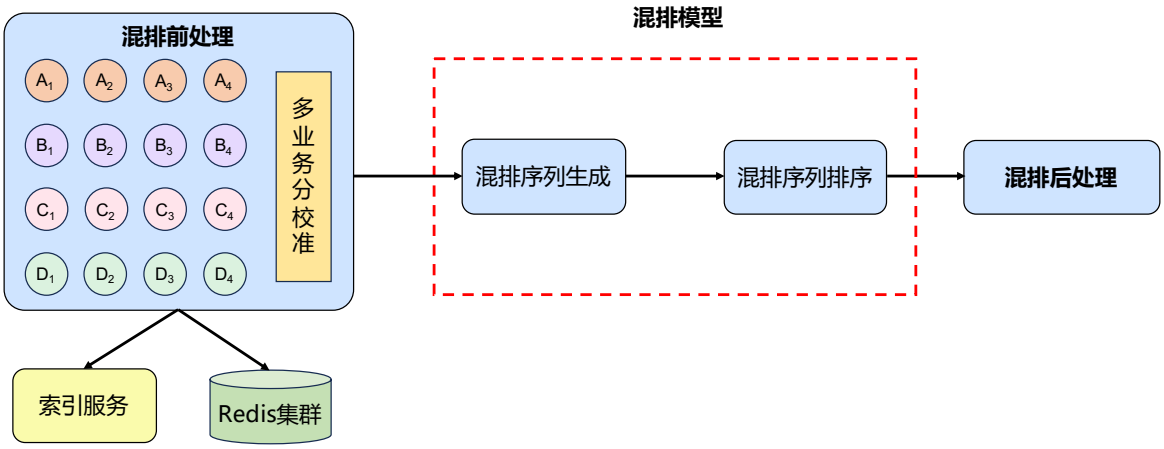

图 8.1: 混排模块内部处理流程。

内容才能在同一个排序框架下进行公平竞争。否则，某些天然分值较高的业务（例如广告竞价系统）可能会在后续排序过程中获得过高权重，从而导致推荐结果过度偏向某一类内容，最终影响整体用户体验。

完成分数校准之后，系统进入混排序列生成与剪枝阶段。由于多个业务的候选内容可以组合形成大量不同的展示序列，因此通常需要设计高效的序列生成策略，在满足业务约束的前提下生成有限数量的候选序列，并通过启发式规则或剪枝策略减少搜索空间，从而降低后续混排序列排序阶段的计算开销。

在此基础上，混排模块会进入多业务排序阶段。该阶段与重排模块中的序列生成（List Generator）和序列评估（List Evaluator）框架较为类似，系统可以利用序列评估模型对整个推荐序列进行价值建模，从用户体验、业务收益以及内容生态等多个维度评估不同序列的优劣，并最终选择综合收益最优的混排结果。

最后，在完成多业务排序之后，还需要进入混排后处理阶段。该阶段主要负责执行各种业务约束和运营策略。例如运营活动的强插坑位、广告与直播内容的曝光比例控制、不同业务流量上下限约束，以及一些特殊内容的保量或降权策略等。通过后处理机制，可以保证最终生成的推荐列表不仅满足模型优化目标，同时也能够满足平台运营、商业化以及内容生态建设等多方面需求。

总体来看，混排模块的本质目标并不仅仅是将多个业务的内容简单拼接在一起，而是在统一的排序框架下，实现不同业务之间的分数对齐、流量分配和价值权衡，从而生成兼顾用户体验与业务收益的最终推荐序列。

8.2 混排前处理

在混排前处理环节，系统会接收来自多个不同业务的候选排序列表，例如广告序列、直播序列、短视频序列以及图文序列等。由于这些业务通常拥有各自独立的推荐系统，因此其排序依据往往也是各自业务内部定义的价值分。例如广告业务可能采用 CPM（Cost Per Mille）或 ECPM 作为排序依据，直播业务可能采用打赏价值、观看时长价值等指标，而短视频业务则可能采用用户满意度或时长收益相关指标进行排序。

然而，不同业务输出的价值分通常来源于各自推荐模型的 pxtr 预估结果，而模型预估值天然会存在一定程度的高估（Over-estimation）或低估（Under-estimation）问题。因此，在混排之前首先需要对各业务的价值分进行校准（Calibration）处理，使其尽可能接近真实价值。以广告业务为例，工业界通常会重点关注模型预估值与真实值之间的偏差情况，一个常见指标是 PCOC（Predicted Click Over Click）。理想情况下：

$$PCOC \approx 1 \tag{8.1}$$

这意味着模型预估结果与真实反馈基本一致，不存在明显的系统性高估或低估。如果某个业务的价值分长期存在偏差，那么其在后续混排过程中所获得的流量分配也会受到影响。

从某种角度来看，混排排序本质上可以理解为多个业务在竞争有限流量坑位的过程。每个业务都会依据自身的价值分参与排序，而混排系统则负责决定最终哪些内容能够获得更靠前的曝光位置。如果不同业务的价值

分未经校准直接参与竞争，那么整个排序过程就相当于在使用不同计量单位进行竞价，其结果往往是不公平且不稳定的。例如，当广告系统的价值分被系统性高估时，广告内容可能会在混排结果中获得远超预期的曝光量，从而挤占直播、短视频等其他业务的流量；反之，如果广告价值被低估，则可能导致广告曝光不足，直接影响平台商业化收入以及广告主投放效果。同样地，直播、图文等业务如果存在严重的分数偏差，也可能导致流量异常波动，影响整体生态平衡。

更严重的情况下，业务价值分未经过有效校准还可能引发系统层面的流量分配故障。例如某个业务模型上线后出现预估值整体偏高的问题，混排系统可能会在短时间内向该业务倾斜大量流量，导致其他业务流量骤降；而当模型恢复正常后，流量又会快速回流，从而造成推荐生态的不稳定。因此，业务价值分校准不仅是混排排序效果优化的重要前提，也是保证多业务流量分配稳定性和系统安全性的关键环节。

总体而言，混排前处理阶段最核心的任务之一，就是将来自不同业务、不同模型、不同价值体系的排序分数映射到统一且可信的价值空间中，为后续的跨业务序列生成与排序奠定基础。

8.2.1 分数校准方法

前面介绍了为什么混排前处理阶段必须进行多业务分校准，以及缺失校准环节可能给推荐系统带来的流量错配、业务失衡甚至系统风险等问题。接下来，我们重点介绍工业界最常见的几类分数校准方法。其中，应用最广泛、也是推荐系统领域最经典的校准方法之一便是保序回归（Isotonic Regression，IR）[3] 及其改进版本 SIR（Smoothed Isotonic Regression）[2]。

保序回归的核心思想非常直观，即：

定义 8.1 (保序回归核心思想)

如果模型预测分数越高，那么对应样本的真实后验概率（Posterior Probability）也应该越高。换句话说，校准函数应当满足单调递增约束：

$$f(p_i) \le f(p_j), \quad \forall p_i \le p_j \tag{8.2}$$

，其中 p_i 表示模型输出的原始 pxtr 分数，$f(\cdot)$ 表示校准函数。

在实际工程中，通常首先利用离线日志数据对模型输出的 pxtr 进行排序，然后按照分数区间进行分桶。例如划分为 N 个 Bucket：

$$B_1, B_2, \cdots, B_N \tag{8.3}$$

对于第 k 个桶，其真实后验概率可以表示为：

$$r_k = \frac{\sum_{i \in B_k} y_i}{|B_k|} \tag{8.4}$$

其中 $y_i \in \{0, 1\}$ 表示真实标签。

理想情况下，不同桶的真实后验概率 r_i 应该满足：

$$r_1 \le r_2 \le \cdots \le r_N \tag{8.5}$$

但由于数据噪声以及采样波动的存在，经常会出现 $r_k > r_{k+1}$ 这种违反单调性的情况。此时，保序回归会利用经典的 **PAVA（Pool Adjacent Violators Algorithm）算法** [1] 进行修正。当相邻两个桶违反单调性约束时，将两个桶进行合并：

$$B_k \leftarrow B_k \cup B_{k+1} \tag{8.6}$$

新的后验概率为：

$$r_{new} = \frac{|B_k| r_k + |B_{k+1}| r_{k+1}}{|B_k| + |B_{k+1}|} \tag{8.7}$$

随后继续检查新的桶序列是否满足单调约束，直到所有桶均满足：

$$r_1 \le r_2 \le \cdots \le r_M \tag{8.8}$$

最终即可得到一个单调递增的校准映射表：

$$p \to f(p) \tag{8.9}$$

在线推理时，根据模型输出的 pxtr 查询对应区间即可获得校准后的预估值。

虽然 IR 能够有效解决模型预估值高估或低估的问题，但它本质上是一种分段常数（Piecewise Constant）函数。也就是说，同一个桶中的所有样本都会被映射到相同的校准值：

$$f(p) = c_k \tag{8.10}$$

因此在桶边界附近容易出现明显的跳变现象。例如两个原始分数非常接近的样本，仅仅因为落在不同桶中，就可能获得差异较大的校准结果。

为了解决这一问题，阿里妈妈团队在 2018 年提出了 SIR（Smoothed Isotonic Regression）方法。SIR 的核心思想是：

笔记 *在保序回归得到的单调映射函数基础上进一步进行平滑处理（Smoothing），从而获得连续且更加稳定的校准函数。*

假设保序回归得到相邻两个校准点 (p_k, r_k) 与 (p_{k+1}, r_{k+1})，则对于区间内部任意一个预测值：

$$p_k \le p \le p_{k+1} \tag{8.11}$$

SIR 会采用线性插值构造平滑校准函数：

$$f(p) = r_k + \frac{p - p_k}{p_{k+1} - p_k}(r_{k+1} - r_k) \tag{8.12}$$

这样得到的校准函数不再是阶梯函数，而是连续的分段线性函数。

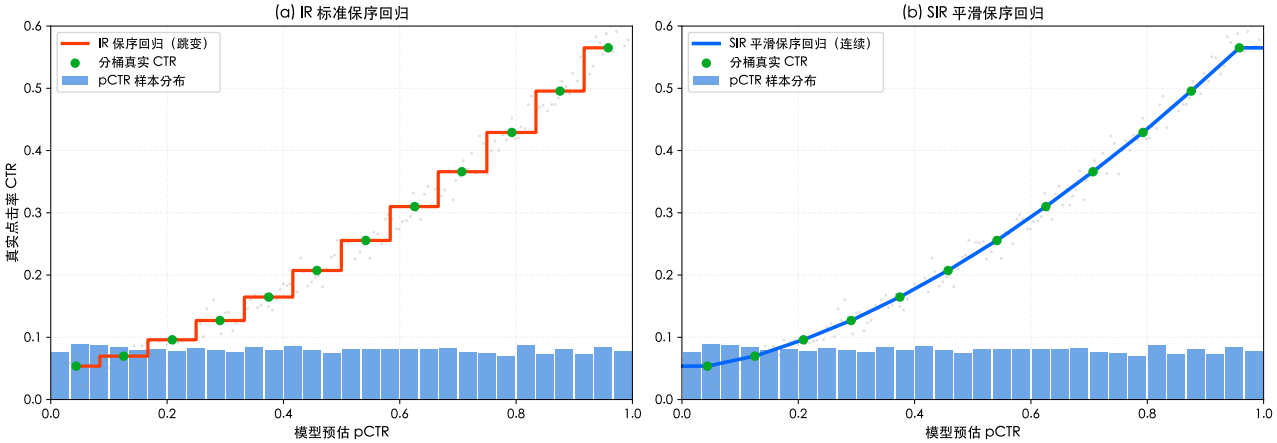

图 8.2: 保序回归于带平滑的保序回归方法对比。

SIR 与 IR 的校准效果对比如图 8.2 所示。从图中可以看出，真实 CTR 本质上是基于历史数据统计得到的离散后验值，因此在不同预测分桶上通常表现为一系列离散采样点。在图 8.2(a) 中，传统 Isotonic Regression（IR）方法通过分段常数函数对后验 CTR 进行拟合。虽然能够保证校准结果满足单调性约束，但由于每个区间内采用相同的校准值，因此在相邻区间之间往往会产生明显的阶梯状跳变。这种不连续的映射关系在实际线上系统中容易导致排序分数出现局部突变，从而影响排序稳定性。相比之下，图 8.2(b) 展示的 Smoothed Isotonic Regression（SIR）方法在 IR 的基础上进一步引入了线性插值平滑处理。通过对相邻分桶之间的校准结果进行连续化建模，

SIR 能够得到更加平滑的校准曲线，使后验 CTR 随预测 CTR 的变化呈现连续增长趋势。这样不仅保留了保序回归的单调性约束，还显著提升了校准函数的平滑性和泛化能力。

因此，在工业界推荐系统和广告系统中，SIR 通常比传统 IR 更为常用。一方面能够有效缓解分桶统计噪声带来的局部震荡问题，另一方面也能够避免校准函数出现过大的阶梯跳变，从而提升后续排序与竞价环节的稳定性。总结一下，相比传统 IR 算法，SIR 算法具有以下几个优点：

- 保留了保序回归天然的单调性约束；
- 消除了桶边界处的分数跳变问题；
- 校准曲线更加平滑，线上表现更加稳定；
- 对于广告 CTR、CVR 等需要高精度校准的场景效果更好。

从工业实践的角度来看，IR 更像是一种非参数化（Non-parametric）的概率校准方法，而 SIR 则是在 IR 基础上的工程增强版本。目前在广告、搜索、推荐以及混排等场景中，SIR 仍然是使用最广泛的分数校准方案之一。对于混排模块而言，各业务模型首先通过 IR 或 SIR 完成业务价值分校准，然后再进行归一化与跨业务排序，从而保证不同业务能够在统一且可信的价值空间内参与后续流量竞争。在实际工程落地中，SIR 往往以离线天级任务的形式运行，通过 Spark 等分布式计算框架对海量样本进行统计与拟合；工程师通常会将校准逻辑封装为 Python 脚本或 UDF，并嵌入 Hive SQL 或 Spark Pipeline 中定期执行，最终生成校准映射表供在线服务加载使用。

除此之外，在多个业务分校准的过程中，经过保序回归（Isotonic Regression, IR）对 pXTR 进行分桶之后，由于真实后验反馈数据通常较为稀疏，某些 pXTR 对应分桶内的样本量可能非常有限，从而导致后验 CTR、CVR 等统计指标出现较大的随机波动。这种由小样本统计带来的高方差（High Variance）问题，是工业界推荐系统、广告系统以及搜索系统中普遍存在的现象。如果直接使用这些高噪声统计结果作为校准值，往往会导致校准映射表不稳定，进而影响后续混排与流量分配效果。因此，在实际工程中，除了利用 IR 保证校准曲线的单调性之外，通常还会进一步引入贝叶斯平滑（Bayesian Smoothing）等方差缩减（Variance Reduction）方法，以降低小样本分桶带来的统计噪声。

表 8.1: 为什么要做贝叶斯平滑的 pCTR 校准示例。

pCTR 分桶	PV	Click	CTR
$[0.1, 0.2]$	100000	12000	0.12
$[0.2, 0.3]$	80000	18000	0.225
$[0.3, 0.4]$	50	20	0.40

以表 8.1 为例，pCTR 取值在 $[0.3, 0.4]$ 区间对应的分桶仅包含 50 次曝光和 20 次点击行为，其经验 CTR 为 $CTR = \frac{20}{50} = 0.4$，即真实后验 CTR 高达 40%。然而由于该分桶样本量极小，这一结果很可能只是随机波动导致的统计偏差。如果直接将其作为校准结果，则会使该区间的校准值明显偏离真实水平。因此需要利用历史统计信息对当前观测结果进行约束，使其向更加可信的全局均值回归，从而获得更加稳定的后验估计。

贝叶斯平滑的核心思想是：

笔记 当前观测数据与历史先验知识共同决定最终 CTR 的取值。当观测样本较多时，模型更相信当前数据；当观测样本较少时，则更多依赖历史先验信息。

由于 CTR 本质上表示一次曝光产生点击的概率，其取值范围位于 $[0, 1]$ 区间，因此通常使用 Beta 分布作为 CTR 的先验分布：

$$CTR \sim Beta(\alpha, \beta) \tag{8.13}$$

其中 α 和 β 分别表示 Beta 分布的两个参数，它们可以通过历史大盘数据、业务分桶统计数据或者矩估计（Method of Moments）、极大似然估计（MLE）等方式获得。对于给定分桶中的曝光与点击数据，假设点击次数服从二项分布：

$$Click \sim Binomial(PV, CTR) \tag{8.14}$$

即 $P(Click|CTR)$ 表示在 CTR 已知的情况下，观察到对应点击次数的概率。

由于 Beta 分布与 Binomial 分布互为共轭先验（Conjugate Prior），根据贝叶斯定理可知，观测数据更新后的

后验分布仍然服从 Beta 分布：

$$CTR|Data \sim Beta(\alpha + Click,\ \beta + PV - Click) \tag{8.15}$$

因此，最终后验 CTR 的期望值可以写为：

$$E[CTR] = \frac{Click + \alpha}{PV + \alpha + \beta} \tag{8.16}$$

这也是工业界贝叶斯平滑最常使用的计算公式。

从该公式可以看出，当分桶样本量较大时，$PV \gg \alpha + \beta$，后验 CTR 会逐渐逼近经验 CTR：

$$CTR \approx \frac{Click}{PV} \tag{8.17}$$

此时模型更加相信观测数据；而当分桶样本量较小时，先验参数 α, β 对结果影响更大，后验 CTR 会向历史均值回归，从而有效抑制小样本带来的统计波动。

总结起来，贝叶斯平滑本质上是一种利用先验知识约束后验估计的方差缩减（Variance Reduction）技术。它与保序回归并不冲突，而是对 IR 校准结果的重要补充：IR 负责保证校准曲线的单调性，贝叶斯平滑负责降低小样本分桶带来的统计噪声，而线性插值则用于消除分桶边界带来的跳变，使校准曲线更加平滑和连续。这三种技术有机结合后，便构成了工业界广泛使用的 Bayesian-SIR 校准方案。此外，贝叶斯平滑的思想并不仅限于分数校准场景。在推荐系统中，粗排、精排模型的 Label 纠偏、CTR/CVR 统计估计以及各类低频特征的平滑处理中，都可以利用贝叶斯平滑降低噪声干扰，从而获得更加稳定和可靠的统计结果。因此，贝叶斯平滑也被认为是推荐系统领域一个非常经典且实用的工程技巧（Engineering Trick）。

8.2.2 分数归一化

在混排系统中，各业务模型的业务分经过 SIR 等校准方法完成校准之后，通常还需要进一步进行归一化（Normalization）处理。归一化的主要目的是将不同业务的竞价分（模型预估值或由多个预估指标组合得到的 Score）映射到统一的量纲和数值范围下，从而使不同业务能够在相对公平的条件下参与后续流量竞争。

造成归一化需求的主要原因在于：**不同业务的目标函数、训练样本以及模型结构往往存在较大差异**。例如广告业务的 eCPM 分数可能分布在数百到数千之间，而推荐业务的 pCTR 或 pVTR 可能仅分布在 $[0, 1]$ 区间内。即使经过校准，不同业务分的数值分布仍然可能存在明显差异，因此需要进一步进行归一化处理。

工业界常见的归一化方法主要包括 Min-Max 归一化、Log Min-Max 归一化、Z-Score 归一化以及基于标准高斯分布的归一化等。Min-Max 归一化将数据线性映射到固定区间（通常为 $[0, 1]$）：

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{8.18}$$

其中 x_{min} 和 x_{max} 分别表示业务分的最小值和最大值。该方法实现简单，计算开销低，因此在工程实践中应用较为广泛。

当业务分分布具有明显长尾特征时，通常会先进行对数变换，再执行 Min-Max 归一化：

$$x' = \frac{\log(x) - \log(x_{min})}{\log(x_{max}) - \log(x_{min})} \tag{8.19}$$

这种方式能够有效压缩极端大值带来的影响，提高归一化结果的稳定性。Z-Score 归一化则利用均值和标准差对数据进行标准化处理：

$$x' = \frac{x - \mu}{\sigma} \tag{8.20}$$

其中 μ 表示均值，σ 表示标准差。经过归一化后，数据均值为 0，标准差为 1。相比 Min-Max 方法，Z-Score 对数据整体分布刻画更加充分，在业务分动态变化较大的场景中较为常见。

对于分布近似服从高斯分布的业务分，还可以进一步利用标准正态分布的累积分布函数（CDF）完成概率

空间映射：

$$x' = \Phi\left(\frac{x-\mu}{\sigma}\right) \tag{8.21}$$

其中 $\Phi(\cdot)$ 表示标准高斯分布的累积分布函数。经过变换后，业务分被映射到 $[0,1]$ 区间，同时能够较好地保留原始排序关系，因此在部分大规模混排系统中也有广泛应用。

在工业界实际应用中，归一化技术并不能替代校准。校准解决的是业务分与真实价值之间的一致性问题，而归一化解决的是不同业务之间量纲和分布不一致的问题。在工业界混排系统中，通常先进行 SIR 等校准处理，再进行归一化，最后完成跨业务排序与流量分配。

此外，还需要注意的是，归一化虽然能够统一不同业务竞价分的量纲和取值范围，但也可能改变原始业务分的分布特征。如果后续多业务融合采用线性加权等方式进行排序，那么“归一化 + 线性融合”的组合往往会使最终排序结果对竞价分的数值变化非常敏感。这种敏感性通常会给模型迭代带来额外风险。例如，当模型出现预估偏高或预估偏低的问题时，如果缺少相应的校准环节，归一化过程可能进一步放大这种偏差，从而导致线上排序效果下降。因此，无论是在粗排、精排阶段的多目标融合，还是在混排阶段的跨业务排序，归一化操作都有可能对竞价分的真实价值表达产生影响。

正因如此，在实际工程中，上游模块输出的竞价分并不一定直接采用模型原始预估值，而往往会结合业务需求进行一定的分布塑形（Score Shaping）。常见做法包括对预估分进行非线性变换、分段映射，甚至将 Top-K 内容的分数统一置为 1 等方式，从而减弱归一化带来的影响，保证业务希望重点透出的内容能够获得更稳定的曝光机会。

8.3 序列生成

混排阶段的序列生成与重排阶段类似，本质上都是从候选物料集合中搜索可能的排序序列。然而，两者最大的区别在于候选规模。重排阶段通常仍然需要面对数十甚至上百个候选 Item，因此往往需要借助 Beam Search 等启发式搜索方法；而在混排阶段，各业务候选数量已经非常有限，此时完全可以直接枚举所有合法序列。

例如，假设当前短视频候选固定为 6 个，广告候选最多为 3 个，直播候选最多为 1 个。此时混排问题实际上转化为：在已经确定的视频序列中，枚举广告和直播可能插入的位置。当广告数量为 3 个时，仅存在一种广告组合。若存在直播，则最终序列长度为 10，需要从 10 个位置中选择 4 个位置放置广告和直播，对应的候选序列数为：$C(10,4) = 210$；若不存在直播，则序列长度为 9，对应候选序列数为：$C(9,3) = 84$，因此该场景总共可以生成：$210 + 84 = 294$ 个候选序列。类似地，当广告数量分别为 2、1、0 时，对应候选序列数分别为：$3(C(9,3) + C(8,2)) = 336$，$3(C(8,2) + C(7,1)) = 105$，$C(7,1) + C(6,0) = 8$。最终所有合法候选序列数量为：$294 + 336 + 105 + 8 = 743$。

可以看到，对于上述例子，即使考虑广告和直播的各种插入组合，总候选序列数仍然只有数百级别。因此在许多工业界混排系统中，如果多业务的候选 item 数量不多，往往可以直接枚举所有候选序列，再利用序列评估模型进行打分和选择，而无需采用复杂的搜索算法。

读者需要从上述示例中认识到：

定义 8.2

当多业务组合产生的候选序列规模较小时，混排阶段的序列生成过程完全可以通过暴力枚举的方式实现，并不一定需要引入复杂的序列生成模型或搜索策略。因此，序列生成模型本身并不是混排系统的必要组成部分，而是解决大规模组合优化问题的一种工程手段。 ♣

然而，当业务规模逐渐扩大时，混排系统输入的候选队列会显著增多。例如，除了广告、直播之外，还可能引入图文、电商、本地生活、知识内容等业务；同时，每个业务的候选 Item 数量也会持续增长。在这种情况下，合法候选序列的数量将呈现组合数级别的增长，甚至出现组合爆炸（Combinatorial Explosion）问题。此时，一方面暴力枚举本身的时间复杂度会迅速升高；另一方面，大量候选序列还需要经过后续的序列评估模型进行打分，

从而进一步增加系统的算力消耗和线上延迟。在工业级推荐系统中，这种计算开销和响应耗时往往是不可接受的。

正因如此，混排系统通常会在序列生成阶段引入各种候选序列召回策略与序列生成模型，从全量候选序列空间中快速筛选出少量高质量候选。例如，利用 Beam Search、启发式搜索（Heuristic Search）、规则剪枝（Rule-based Pruning）、强化学习（Reinforcement Learning）、序列生成模型（Sequence Generation Model）等方法，对候选空间进行有效压缩。其本质目标并不是寻找绝对最优解，而是在排序效果、计算复杂度和线上时延之间取得合理平衡。

从工程角度来看，混排序列生成阶段与传统召回模块具有一定相似性：二者都不直接追求最终最优结果，而是通过较低的计算成本从庞大的候选空间中快速筛选出一批优质候选，再交由后续更加精细的评估模型完成最终决策。因此，混排中的序列生成模型本质上也可以看作是一种面向序列空间的“召回模型”。

8.3.1 序列生成约束规则

在混排序列生成过程中，除了需要考虑序列评估模型的得分之外，还需要满足大量业务约束（Business Constraints）。从工程实践角度来看，混排本质上是一个带约束条件的序列优化问题（Constrained Sequence Optimization）。因此，序列生成阶段除了负责寻找高价值候选序列之外，还需要保证生成的序列满足各种业务规则和用户体验规则。

广告插入规则是混排系统中最常见的一类约束。例如，为了降低广告对用户体验的负面影响，广告通常不能连续出现，相邻广告之间必须间隔一定数量的非广告内容，这种约束称为**刷次内 Gap 约束（Intra-Feed Gap Constraint）**。例如要求任意两个广告之间至少间隔 3 个自然内容：

$$ad \rightarrow item \rightarrow item \rightarrow item \rightarrow ad \tag{8.22}$$

如果广告连续出现：

$$ad \rightarrow ad \tag{8.23}$$

则该候选序列会被直接过滤。

除此之外，还存在**跨刷次 Gap 约束（Inter-Feed Gap Constraint）**。例如用户上一刷的最后一个 Item 是广告，则下一刷的首位通常不允许再次出现广告，否则用户会感知到广告过于密集。此时混排系统在生成当前刷次序列时，需要同时参考历史曝光序列信息。

除了广告之外，直播业务也经常存在类似约束。例如对于直播偏好较低的人群，直播内容可能不允许出现在首刷位置；而对于直播重度用户，则允许直播出现在首刷甚至首位位置。类似地，电商、本地生活等强商业化内容也可能存在不同程度的位置约束。

在实际工业系统中，常见的混排约束还包括：

- **位置约束（Position Constraint）**：某类内容只能出现在指定位置范围内；
- **频控约束（Frequency Constraint）**：同一业务或同一作者在固定窗口内不能曝光过多次；
- **多样性约束（Diversity Constraint）**：避免连续出现相同类别、相同作者或相同主题内容；
- **保量约束（Quota Constraint）**：保证特定业务获得最低曝光量；
- **探索约束（Exploration Constraint）**：为新内容预留一定曝光机会；
- **商业化约束（Commercial Constraint）**：保证广告、电商等业务达到既定收益目标。

因此，混排中的序列生成过程通常并不是简单地按照分数排序，而是在生成候选序列时不断检查各种约束条件。一个典型的约束过滤伪代码如下：

代码 8.1: 基于约束过滤的混排序列生成伪代码

```
def generate_valid_sequences(candidates, K=8, beam_size=20, ad_gap_rule=None, live_pos_rule=None,
                             diversity_rule=None, frequency_rule=None, scorer=None):
    """
    candidates: List[item] 来自多业务的候选集合
```

```
    K: int 最终序列长度
    beam_size: int beam search宽度
    scorer: function(seq) -> float 序列级评分函数
    """

    # 初始化 beam（每个元素是一个部分序列）
    beam = [[]]
    for t in range(K):
        new_beam = []
        for seq in beam:
            for item in candidates:
                # 防止重复选择
                if item in seq:
                    continue
                new_seq = seq + [item]
                # 约束检查模块
                if ad_gap_rule and ad_gap_rule(new_seq):
                    continue
                if live_pos_rule and live_pos_rule(new_seq):
                    continue
                if diversity_rule and diversity_rule(new_seq):
                    continue
                if frequency_rule and frequency_rule(new_seq):
                    continue
                new_beam.append(new_seq)
        # 如果候选为空，提前终止
        if not new_beam:
            break

        # 序列打分与剪枝
        new_beam = sorted(new_beam, key=lambda x: scorer(x) if scorer else 0.0, reverse=True)
        beam = new_beam[:beam_size]
    return beam
```

更进一步，在 Beam Search 等搜索算法中，约束通常会提前融入搜索过程。当某个部分序列已经违反约束时，系统会直接剪枝（Pruning），而不会继续扩展后续节点，这样能够显著减少无效搜索空间，提高序列生成效率。下面我们给出带约束剪枝的混排序列 Beam Search 生成的伪代码。

代码 8.2: 带约束剪枝的混排序列 Beam Search 生成伪代码

```
def constrained_beam_search(candidates, K=8, beam_size=10, scorer=None, constraint_fn=None):
    """
    candidates: List[item] 多业务候选集合
    K: int 序列长度
    beam_size: int beam宽度
    scorer: function(seq) -> float 序列评分函数
    constraint_fn: function(seq, item) -> bool 约束检查函数（True表示违反约束）
    """
```

```
    # beam中每个元素是 (sequence, score)
    beam = [([], 0.0)]
    for t in range(K):
        new_beam = []
        for seq, seq_score in beam:
            # 候选扩展
            next_items = candidates
            for item in next_items:
                # 防止重复
                if item in seq:
                    continue
                new_seq = seq + [item]

                # 约束剪枝（核心）
                if constraint_fn and constraint_fn(new_seq, item):
                    continue
                # 计算增量分数（避免重复算全量）
                if scorer:
                    new_score = scorer(new_seq)
                else:
                    new_score = seq_score
                new_beam.append((new_seq, new_score))
        # 如果无候选，提前终止
        if not new_beam:
            break

        # beam pruning
        new_beam.sort(key=lambda x: x[1], reverse=True)
        beam = new_beam[:beam_size]
    return beam
```

从工业界实践来看，随着业务数量不断增加，混排系统往往需要同时满足数十甚至上百条约束规则。因此，约束管理本身也逐渐演变为独立的规则引擎（Rule Engine）或约束求解模块（Constraint Solver）。很多时候，一个混排系统的复杂度并不来自排序模型本身，而是来自这些相互耦合且不断变化的业务约束。

8.3.2 预竞价与全竞价架构

在混排序列生成过程中，一个经常被忽略但非常重要的问题是：

笔记 究竟应该由上游业务系统决定是否参与竞价，还是由混排系统统一决定最终的流量分配？

这个问题会直接影响到混排模块序列生成这一部分的工程实现。围绕这个问题，工业界通常存在两种不同的架构设计思路：**业务预竞价（Pre-Auction）**和**全竞价（Full Auction）**。

如图 8.3 所示，在业务预竞价（Pre-Auction）架构中，广告、直播、电商等业务系统会首先在各自业务内部完成排序与竞价过程，并提前决定哪些内容具备参与混排的资格。对于不满足竞价条件的候选 Item，业务侧会直接进行过滤，不再传递给混排模块。例如，广告系统可能结合 eCPM、预算约束、频次控制等因素进行内部决策，最终仅输出最多 3 个广告候选。此时预竞价模块实际上完成了一次是否参与混排的判断：当输出动作为 1 时，表示广告参与后续混排；当输出动作为 0 时，则不向混排系统下发广告候选。类似地，直播系统也可以根据用户画像、直播偏好以及业务目标等信息，决定是否向混排系统下发直播候选。经过业务侧预竞价之后，混排模块接收到的候选规模通常已经被显著压缩，因此后续序列生成过程更多是在少量候选内容之间进行位置组合

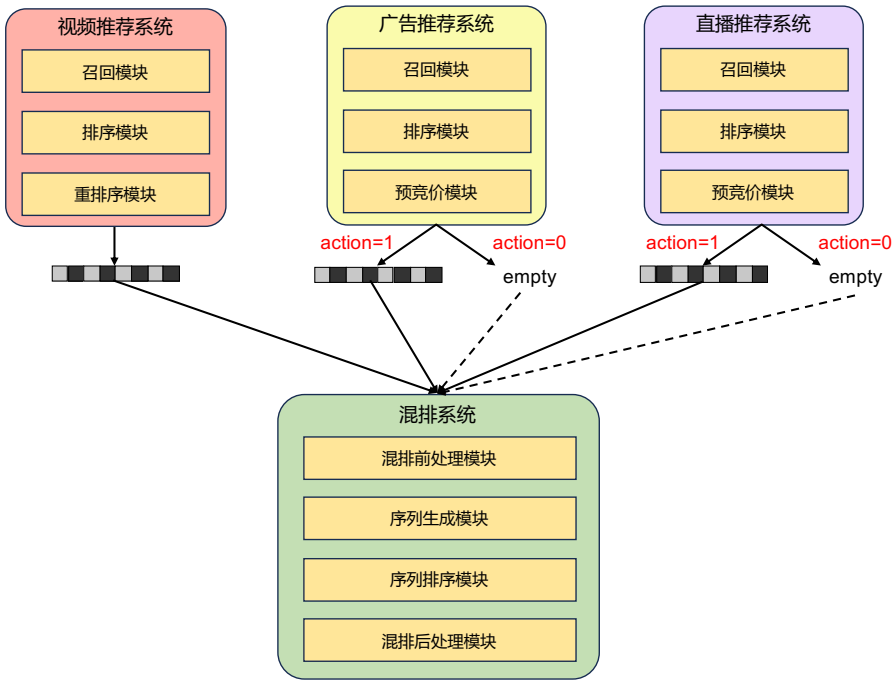

图 8.3: 业务预竞价的系统架构。

与排序优化，而无需面对庞大的候选空间。

这种架构的优点主要体现在以下几个方面：

- 候选规模较小，序列空间有限。尤其是在广告、直播等业务未参与竞价时，混排阶段序列生成的搜索空间会进一步地缩小；
- 混排复杂度较低，计算成本和在线时延更容易控制；
- 各业务能够独立维护自身竞价逻辑，并充分利用业务侧特有的特征、约束以及优化目标进行决策；
- 系统职责划分清晰，模块边界明确，工程实现和迭代维护相对简单。

然而，这种架构也存在明显的局限性。由于业务侧已经提前完成了候选筛选和流量决策，混排系统实际上丧失了部分全局优化能力。例如，当直播推荐系统已经决定输出 1 个广告候选时，混排阶段只能优化这 1 个直播的插入位置，而无法进一步探索“插入 2 个直播”或“插入 3 个直播”等其他可能的序列方案。从本质上来看，业务预竞价将部分流量分配决策前置到了各业务推荐系统内部，从而缩小了混排系统的决策空间。虽然这种方式能够显著降低系统复杂度，但也可能限制混排模块对用户体验、商业收益以及长期价值进行全局权衡的能力，因此容易与理论上的全局最优解存在一定差距。

与之相对应的是全竞价架构。如图8.4所示，在全竞价模式下，广告、直播、电商等业务系统不会提前决定本次请求是否参与竞价，而是将排序后需要参与竞价的候选列表整体传递给混排系统。例如，广告系统返回 Top5 广告候选，直播系统返回 Top5 直播候选，短视频系统返回 Top8 视频候选。最终由混排系统统一决定：

- 是否插入广告；
- 插入多少广告；
- 是否插入直播；
- 插入哪些直播；
- 各业务内容的具体排序位置。

在这种架构下，混排模块不仅能够生成包含广告和直播的序列，也能够生成完全不包含广告、完全不包含直播的序列，并通过序列评估模型统一比较各种方案的收益。从理论上讲，全竞价架构更容易获得全局最优解，因为所有业务都处于同一个竞争空间之中，流量分配决策统一由混排系统完成。

然而，全竞价架构也会带来新的挑战。首先，随着业务种类和候选 Item 数量的不断增加，合法候选序列的数量将呈现组合数级增长，从而产生严重的组合爆炸（Combinatorial Explosion）问题。此时无论是序列生成阶

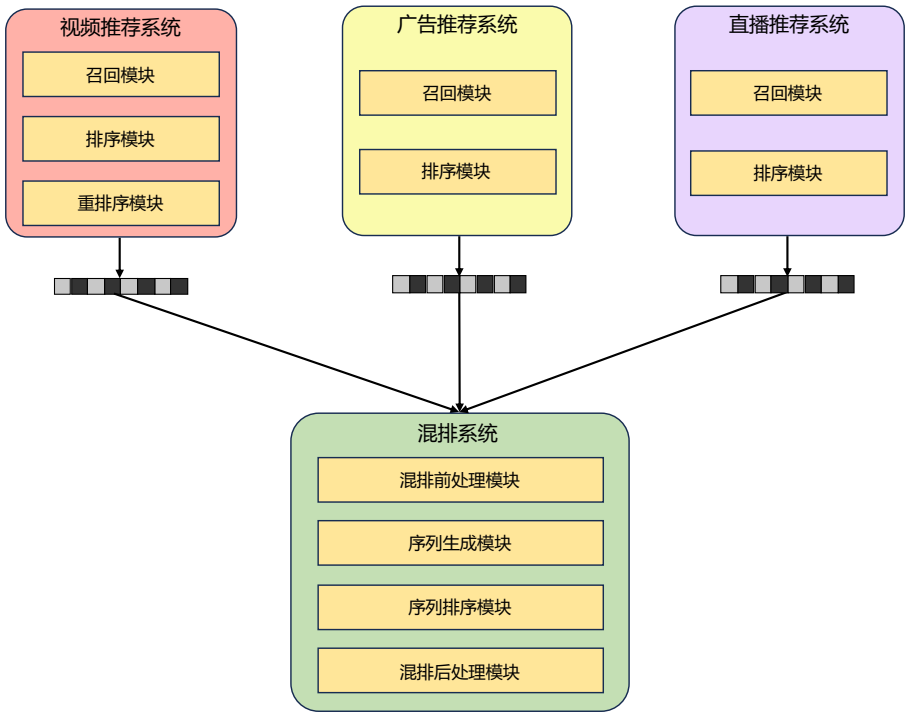

图 8.4: 业务全竞价的系统架构。

段还是后续序列评估阶段，都将面临巨大的算力消耗和时延压力。尤其是在工业级推荐系统中，混排模块通常运行在毫秒级延迟约束下，因此不可能对海量候选序列进行充分评估。其次，全竞价会将大量业务决策逻辑集中到混排模块。例如，业务是否参与竞价、参与竞价的候选规模以及最终曝光数量等问题，都需要在混排序列生成阶段完成决策。随着业务数量增加，混排模块逐渐演变为一个统一的流量调度中心，其需要同时兼顾用户体验、商业收益、内容生态以及长期价值等多个优化目标。

然而，混排模块天然存在信息边界问题。相比于广告、直播、电商等业务推荐系统，混排模块通常只能获取经过加工后的候选结果和有限的特征信息，而难以访问各业务内部更加丰富的用户特征、内容特征以及业务状态信息。因此，在相同算力约束下，混排模块对于“是否参与竞价”、“参与多少候选”的判断，未必能够达到业务侧预竞价模块的决策精度。从另一个角度来看，业务预竞价实际上也是一种有效的搜索空间剪枝（Pruning）机制。通过在前链路提前完成部分业务决策，可以显著缩小混排阶段需要考虑的候选空间，从而降低序列生成的复杂度。同时，候选序列规模的下降还能够释放更多算力预算，使系统能够采用更加复杂、更高精度的序列评估模型进行最终打分。

在工业实践中，业务预竞价与全竞价并不存在绝对优劣。业务预竞价更强调系统效率、模块解耦以及工程可控性，而全竞价更强调全局最优和统一流量分配能力。许多大型推荐系统往往采用折中方案：由业务侧完成第一轮粗粒度筛选和预竞价，再由混排系统在有限候选空间内进行统一决策，从而在排序效果、系统复杂度以及在线时延之间取得平衡。

从系统架构与设计的角度来看，业务预竞价与全竞价本质上体现的是流量决策权在业务侧与混排侧之间的不同划分方式。前者更倾向于分布式决策，而后者更倾向于集中式决策。在分布式决策的预竞价框架下，业务侧承担更多决策责任时，混排系统更接近排序器（Ranker）；而在集中式决策的全竞价框架下，混排模块则承担了统一的流量分配职责，这时的混排系统更接近全局资源调度器（Global Allocator）。上述这两种架构各有适用场景，其选择往往取决于业务复杂度、算力预算以及系统对全局最优性的追求程度。

8.4 序列排序

混排模块的序列排序阶段主要包括序列评估（Sequence Evaluation）和序列排序（Sequence Ranking）两个部分。其中，序列评估阶段与重排序模块中的序列评估过程较为类似，其目标是对序列生成阶段产生的多个候选序列进行价值估计；而序列排序阶段则根据评估结果对所有候选序列进行排序，并最终选择价值最高的序列作为最终曝光结果。

在实际工程中，序列评估可以采用两种方式完成：

- 直接利用序列评估模型直接预测序列整体价值；
- 将序列评估模型输出与业务价值分进行融合，构建最终的序列价值分。

因此，一个常见的序列融合分计算公式可以表示为：

$$Score_{seq} = \alpha \cdot Score_{model} + \beta \cdot Score_{business} \tag{8.24}$$

其中：

- $Score_{model}$ 表示序列评估模型预测的整体收益；
- $Score_{business}$ 表示广告收益、时长收益、电商收益等业务目标的加权组合；
- α, β 表示不同目标之间的权重参数。

对于多目标混排场景，还可以进一步扩展为：

$$Score_{seq} = \sum_{k=1}^{N} w_k \cdot V_k \tag{8.25}$$

其中 V_k 表示第 k 个业务目标对应的价值指标，例如广告收入、观看时长、用户留存、直播成交额等，而 w_k 则表示对应目标的重要程度。

然而，仅仅对序列中的 Item 分数进行简单求和往往是不合理的，因为用户对于不同位置内容的关注程度存在显著差异。通常情况下，靠前坑位获得的曝光概率和点击概率远高于靠后坑位，这种现象被称为**位置偏差（Position Bias）**。因此，在计算序列价值时，工业界通常会引入**位置衰减因子（Position Discount Factor）**：

$$Score_{seq} = \sum_{i=1}^{L} \gamma_i \cdot Score_i \tag{8.26}$$

其中：

- $Score_i$ 表示第 i 个位置 Item 的价值分；
- γ_i 表示位置权重；
- L 表示序列长度。

一种常见的位置衰减方式为指数衰减：

$$\gamma_i = e^{-\lambda(i-1)} \tag{8.27}$$

其中 λ 为衰减系数。例如：$\lambda = 0.2$ 时：

$$\gamma_1 = 1.0, \gamma_2 = 0.82, \gamma_3 = 0.67, \gamma_4 = 0.55 \tag{8.28}$$

从而体现首屏内容远比尾部内容更重要的特点。

除此之外，在混排系统中，还经常会遇到候选序列长度不一致的问题。例如：

- 序列 A 包含 6 个短视频和 1 个直播，总长度为 7；
- 序列 B 仅包含 6 个短视频，总长度为 6。

如果直接对序列中所有 Item 的价值分进行累加，则长度更长的序列往往会获得更高的总价值分，从而在最终排序过程中占据天然优势。然而，这种优势并非来源于序列质量本身，而是由序列长度差异所导致的统计偏置，因此会影响不同候选序列之间的公平比较。

针对这一问题，工业界通常采用以下两种解决方案：

- 对序列总价值进行长度归一化（Length Normalization）；
- 引入虚拟 Item（Mock Item）对较短序列进行长度补齐。

其中，第二种方案在实际业务场景中更为常见。

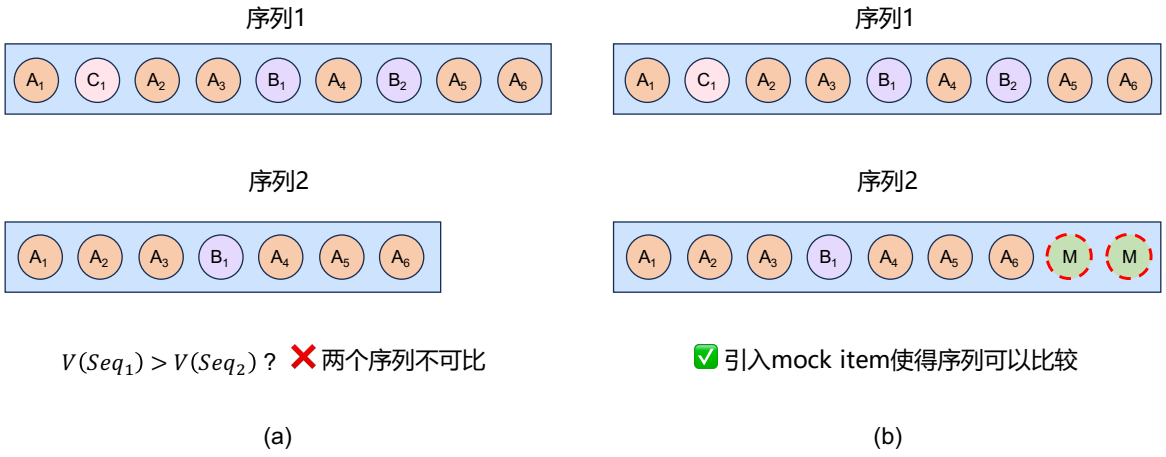

图 8.5: 混排序列排序中的虚拟 Item 补齐示意图。

如图8.5(a) 所示，设 $A_1, A_2, \cdots, A_6$ 表示短视频内容，B_1, B_2 表示广告内容，C_1 表示直播内容，M 表示虚构的内容。此时，序列 1 的长度为 9，而序列 2 的长度仅为 7。若直接将两个序列中所有 Item 的价值分进行求和，由于序列 1 包含更多的 Item，其总价值分可能天然高于序列 2，从而使排序策略过度偏向长度更长的候选序列。然而，不同长度的序列在统计意义上并不具备直接可比性，因此这种排序方式并不合理。

为了解决这一问题，可以采用 Mock Item 对序列进行长度对齐。如图8.5(b) 所示，在序列 2 末尾补充 2 个虚拟 Item，使其长度与序列 1 保持一致。经过长度补齐后，不同候选序列的总价值分便能够在统一标准下进行比较，从而提高序列排序的公平性与合理性。在工业实践中，Mock Item 并不对应真实内容，其价值分通常设置为当前业务类型下内容价值分的均值，或者依据历史统计数据进行估计，以尽可能减小其对序列整体价值评估结果的影响。

当所有候选序列完成价值评估后，混排系统根据序列融合分（Sequence Score）进行排序：

$$Seq^* = \arg\max_{Seq} Score_{seq} \tag{8.29}$$

即从所有候选序列中选择融合价值最高的 Top-1 序列作为最终输出结果。

8.5 序列价值评估体系

需要注意的是，从算法实现角度来看，序列排序本身并不复杂，其核心挑战更多来自序列价值函数的设计。位置偏差建模、多目标融合以及长度归一化等问题，往往比排序过程本身更加重要。很多时候，一个优秀的混排系统并不是因为使用了复杂的排序算法，而是因为构建了更加准确的序列价值评估体系。

而在序列价值评估体系构建过程中，一个非常关键且具有挑战性的问题在于：

笔记 不同业务之间的价值究竟应该如何进行统一衡量？

在混排序列价值计算过程中，每个坑位对应的 Item 可能来自完全不同的业务。例如，短视频内容的核心价值通常体现在用户消费时长、完播率或用户留存等指标上；直播内容的价值可能更多来源于打赏收入、商品成

交额以及用户互动行为；广告内容则主要体现为营销收入或 eCPM 收益。不同业务的价值定义、统计口径以及数值分布均存在显著差异，因此这些业务分天然不具备直接比较的能力。

最简单的处理方式是通过归一化方法将不同业务的价值分映射到统一范围，例如采用 Min-Max 归一化、Z-Score 标准化等方式完成数值对齐。然而，这种方法本质上只是对数据进行了缩放处理，并没有真正解决不同业务价值口径不一致的问题。在工业实践中，更常见的做法是通过统计分析和长期实验建立**不同业务价值之间的兑换关系（Exchange Rate）**。例如，通过历史数据分析评估广告收入、直播打赏收入以及用户消费时长之间的关联关系，从而得到某种统一的价值衡量标准。这样一来，直播产生的打赏收益、广告带来的营销收入以及其他商业化收益都可以折算为统一的时长价值、留存价值或者平台价值，最终归约到同一个价值体系下进行计算。

下面通过一个简化示例来说明业务价值统一的过程。假设平台经过长期实验分析发现，不同用户行为对于平台整体价值的贡献关系如表 8.2 所示。

表 8.2: 不同业务价值的兑换关系示例

用户行为	平台收益	折算时长价值
用户消费 1 分钟短视频	0.05 元	1 分钟
广告收入 1 元	1 元	20 分钟
直播打赏 1 元	0.5 元	10 分钟
电商 GMV 1 元	0.1 元	2 分钟

从表中可以看到，平台选择将「用户消费时长」作为统一价值单位。由于：

$$1\text{分钟时长价值} = 0.05\text{元} \tag{8.30}$$

因此：

$$1\text{元平台收益} = 20\text{分钟时长价值} \tag{8.31}$$

从而可以得到各业务对应的兑换关系：

$$Value_{ad} = 20 \times Revenue \tag{8.32}$$

$$Value_{live} = 10 \times Gift \tag{8.33}$$

$$Value_{ecommerce} = 2 \times GMV \tag{8.34}$$

这样不同业务的价值便可以映射到同一个价值空间之中。

表 8.3: 统一价值计算示例

Item	业务类型	原始价值	统一价值分
Video-A	短视频	预计观看 3 分钟	3
Ad-B	广告	eCPM=0.20 元	4
Live-C	直播	预计打赏 0.50 元	5
Ecom-D	电商	GMV=1.50 元	3

例如，当前有如下候选 Item，如表8.3所示。经过统一价值映射之后，混排系统看到的不再是广告收入、直播打赏金额或者电商成交额，而是统一价值分：

$$Value(item) = 3, 4, 5, 3 \tag{8.35}$$

此时，不同业务的内容终于具备了直接比较的能力。进一步地，假设当前位置存在位置偏差（Position Bias），对

应的位置权重如下：

表 8.4: 位置衰减权重示例

位置	权重
1	1.0
2	0.8
3	0.6
4	0.4

此时，两个候选序列：

$$Seq_1 = [Live\text{-}C, Ad\text{-}B, Video\text{-}A] \tag{8.36}$$

$$Seq_2 = [Ad\text{-}B, Live\text{-}C, Video\text{-}A] \tag{8.37}$$

对应的序列价值分别为：

$$Score(Seq_1) = 1.0 \times 5 + 0.8 \times 4 + 0.6 \times 3 = 10.0 \tag{8.38}$$

$$Score(Seq_2) = 1.0 \times 4 + 0.8 \times 5 + 0.6 \times 3 = 9.8 \tag{8.39}$$

因此：

$$Score(Seq_1) > Score(Seq_2) \tag{8.40}$$

最终，混排系统会选择 Seq_1 作为排序结果。

需要注意的是，上述示例仅用于说明业务价值统一的基本思想，具体数据并不一定合理。在实际工业系统中，兑换关系往往并不是固定常数，而是通过长期 AB 实验观测、ROI 分析、LTV（Life Time Value）建模以及因果推断等方法动态估计得到的。随着业务目标和平台阶段的变化，不同业务之间的价值兑换比例也会持续调整。

进一步地，一些成熟的推荐系统还会构建**统一价值模型（Unified Value Model）**，直接为每个 Item 计算综合价值分。该价值分同时考虑用户体验、商业收益、内容生态以及长期价值等多个目标，使不同业务的内容能够在同一个价值空间中进行公平竞争。在统一价值体系的基础上，序列价值函数可以表示为：

$$Score_{seq} = \sum_{i=1}^{L} \gamma_i \cdot Value(item_i) \tag{8.41}$$

其中：

- $Value(item_i)$ 表示统一价值模型计算得到的 Item 价值；
- γ_i 表示对应位置的曝光衰减权重；
- L 表示序列长度。

此时，无论当前位置展示的是短视频、直播、广告还是其他业务内容，都能够通过统一价值函数参与序列收益计算。

从本质上来看，混排系统最终优化的并不是广告价值、直播价值或者时长价值本身，而是**平台整体价值（Platform Value）**。只有当所有业务被映射到统一的价值空间之后，序列评估模型才能真正比较不同序列之间的优劣，混排系统的流量分配也才能更加科学、合理，并最终实现用户体验、商业收益以及内容生态的长期平衡与共同繁荣。

8.6 混排后处理与流量调控

当序列排序完成后，系统将获得融合价值最高的 Top-1 候选序列。最直接的做法是将该序列直接下发至服务端，由服务端完成必要的数据处理后返回客户端，并最终曝光给用户。然而，在真实工业场景中，完全依赖序列排序结果往往存在较大的风险。

一方面，序列生成、序列评估以及序列排序等环节所依赖的模型均可能出现异常。例如模型训练任务中断、特征服务故障、实时数据流出现脏数据、数据分布发生漂移（Data Drift）、模型参数出现 Nan 值等问题，都可能导致部分业务的价值评估结果出现明显偏差。另一方面，即使模型本身运行正常，也难以保证其能够覆盖所有极端场景。当排序模型发生误判时，最终输出的 Top-1 序列可能并非真正符合业务目标的最优结果。例如广告价值被异常放大、直播内容被过度推荐，甚至某些业务流量在短时间内出现剧烈波动，从而对用户体验和平台生态产生负面影响。

因此，在工业级推荐系统中，混排模块通常不会将排序结果直接下发，而是会在最终曝光之前增加一道后处理（Post-processing）环节，对输出结果进行最后的安全校验和业务修正，以保证系统整体运行的稳定性与可控性。

8.6.1 Load 调控

Load 调控（Load Control）是推荐系统混排后处理阶段的核心功能模块，其核心价值在于推荐链路的末端环节，实现多业务流量的实时闭环调控，从机制上规避上游召回、排序、混排模型异常波动、模型迭代偏差、内容突发热点等因素引发的流量分配失衡问题，保障平台生态与用户体验的稳定性。

在多业务混排的工业场景中，混排模型通过学习广告、直播、普通内容等多类型物料的价值分布，完成内容的初步混合排序，但模型在实际在线运行过程中难以避免动态波动。例如，广告预估价值被异常高估、直播内容预测点击率突增、热点内容挤压普通内容曝光空间等场景，若完全依赖模型原始排序结果，会导致单一业务流量短时间内急剧膨胀，直接降低用户体验，甚至影响平台长期留存与生态平衡。

基于此，工业界普遍为各业务设定流量占比目标与安全波动区间，通过滑动时间窗口实时统计各业务的实际曝光占比，基于实际占比与目标占比的偏差动态修正内容排序分数或曝光概率，最终实现流量的稳态控制。典型的业务流量约束配置可表述为：广告流量占比约束在 $10\% \sim 15\%$ 区间，直播流量占比约束在 $5\% \sim 8\%$ 区间，普通内容流量占比设置不低于 80% 的兜底约束。

在流量闭环控制的技术方案中，PID（Proportional-Integral-Derivative）控制器作为经典控制理论的核心算法，凭借无训练数据依赖、结构简洁、可解释性强、鲁棒性优异、实时响应快等特性，成为推荐、广告、搜索系统流量 Load 调控的主流方案。PID 控制器以目标流量占比与实际流量占比的偏差为核心输入，通过比例、积分、微分三项的协同作用，输出精准的调节量，实现流量的动态纠偏。定义 t 时刻目标流量占比为 r_t，当前时间窗口内统计得到的实际流量占比为 y_t，二者的偏差构成控制误差，其数学表达为：

$$e_t = r_t - y_t \tag{8.42}$$

该误差是 PID 控制器的调节依据，当误差为正时，代表实际流量占比低于目标值，需要提升对应业务的曝光权重；当误差为负时，代表实际流量占比高于目标值，需要降低对应业务的曝光权重；误差为零时则无需调节。PID 控制器的输出由比例项、积分项、微分项线性叠加构成，完整数学表达式为：

$$u_t = K_P e_t + K_I \sum_{i=1}^{t} e_i + K_D (e_t - e_{t-1}) \tag{8.43}$$

其中 K_P、K_I、K_D 分别为比例系数、积分系数、微分系数，三项组件各司其职且协同互补。比例项直接映射当前控制误差，实现对流量偏差的即时响应，误差越大调节力度越强，是控制器快速纠偏的核心；积分项对历史误差进行累加运算，能够消除长期稳态误差，确保实际流量占比最终精准收敛至目标值，弥补纯比例调节的固有缺陷；微分项通过误差的变化率预测调节趋势，对偏差的突变进行提前抑制，有效降低调节过程中的震荡与超调，提升系统稳定性。

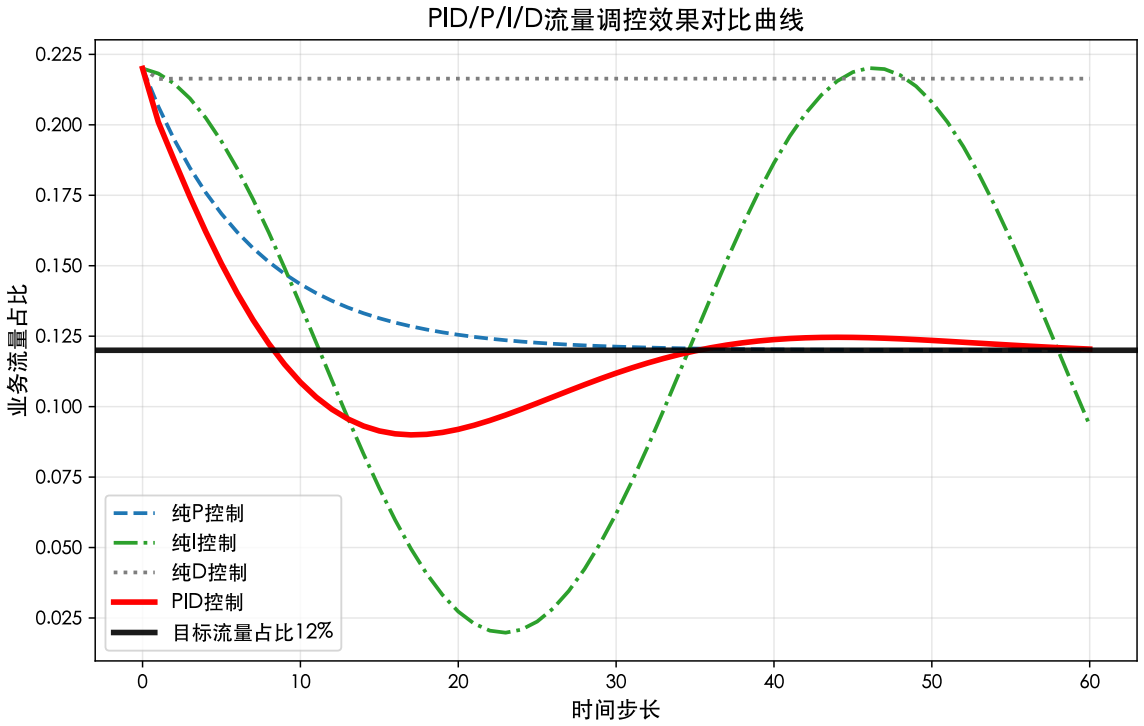

图 8.6: PID 控制与 P、I、D 控制对比示意图。

PID 控制的示意图如图8.6所示，一开始假设业务初始流量占比 22%、目标稳态占比 12%，分别构建仅启用比例、仅启用积分、仅启用微分、三项联合 PID 这四类调控逻辑。仅采用纯 P 控制时，流量可以快速向着目标值回落，但曲线最终停滞在高于目标线的位置，始终存在固定稳态残差，无法精准贴近 12% 基准线，这正是缺少积分项无法消除累积静差的典型表现。纯 I 控制依托误差累积的持续修正能力可以消除稳态误差、最终收敛到目标占比，但前期收敛速度缓慢，调节进程滞后，短时间内难以快速压制过高的流量占比。纯 D 控制仅感知误差变化速率，无法自主消除固定偏差，整条曲线几乎无法向目标值收敛，无法独立完成流量闭环调控。

而完整 PID 控制融合三项优势，依靠比例项实现快速下行纠偏、积分项根除稳态误差、微分项抑制曲线震荡超调，曲线平稳、快速地收敛至目标流量基准线，兼顾收敛速度与稳态精度，这也是该算法在工业流量负载管控中大规模落地的关键原因。从工程落地视角来看，三种单一项控制各自存在明显短板，无法单独满足线上多业务流量平稳约束的需求，PID 的组合形式补齐各项缺陷，完美适配混排后实时流量熔断与稳态调控的业务诉求。

在推荐系统 Load 调控的工程落地中，PID 控制器的输出 u_t 可以直接作用于内容排序分数的修正环节，将原始混排分数叠加调节量后得到最终排序分数，即

$$score_{final} = score_{raw} + u_t \tag{8.44}$$

基于修正后的分数重新完成内容排序。通过实时统计曝光数据、计算控制误差、输出调节量、修正排序分数的闭环流程，实现流量占比的持续稳态控制。例如，广告业务目标流量占比为 12%，若实时统计占比达到 15%，此时误差 $e_t = 12\% - 15\% = -3\%$，PID 控制器输出负向调节量，降低广告内容的排序分数，减少其曝光概率；若广告实际占比低于目标值，则输出正向调节量提升曝光权重。

相较于强化学习、多目标优化等复杂调控策略，PID 控制器无需标注数据与模型训练，不依赖上游模型参数，在模型异常、流量突增等极端场景下仍能稳定工作，具备极强的工程实用性。从系统架构层面，Load 调控本质是推荐系统的熔断保护机制，能够独立于上游模块工作，即便召回、排序等核心模型出现异常，也能约束各业务流量在安全区间内，避免流量剧烈波动，维持平台整体生态的稳定运行。

Load 调控的工程落地高度依赖对真实流量分布的持续观测，工业场景中普遍采用滑动时间窗口或固定计数窗口完成流量占比统计。系统会对每一次用户请求返回的曝光结果进行实时采集，将最近 N 次曝光记录纳入统计窗口，动态计算目标业务的实际流量占比，这一统计结果作为 PID 控制器的输入，形成完整的流量反馈闭环。

8.6.2 PID 控制

基于窗口统计的 PID 控制器能够有效规避瞬时波动带来的调节震荡，使流量调控更贴合业务的真实分布状态。在调节机制上，比例项依据当前窗口内的流量偏差提供即时纠偏能力，积分项通过累积历史窗口误差消除稳态偏差，微分项则根据误差变化趋势抑制调节超调，三者协同使流量占比平滑收敛至目标区间。为适配在线系统的稳定性要求，工程实现中通常引入积分限幅与调节量裁剪机制，避免积分累积饱和与分数剧烈波动，确保系统在高并发、高动态的流量场景中始终保持鲁棒运行。这种基于滑动窗口的 PID 流量调控方案，无需模型训练、不依赖上游模型状态，能够在模型异常、流量突变等场景下独立完成流量熔断与稳态保护，成为推荐系统流量治理体系中不可或缺的核心环节。

下面，我们给出一个工业界场景下基于 N 次请求窗口的 PID 控制的 C++ 类的具体实现逻辑。

代码 8.3: 带请求窗口的 PID 控制示例代码

```
#include <vector>
#include <mutex>
#include <cmath>
#include <algorithm>

class PIDLoadController {
private:
    struct Window {
        uint64_t total = 0;
        uint64_t exposed = 0;
    };

    float target_load_;
    float kp_, ki_, kd_;
    float last_error_{0.0f};
    float integral_{0.0f};
    float max_integral_{1.0f};
    float max_adjust_{0.5f};

    int window_num_;
    int window_size_;

    std::vector<Window> windows_;
    int current_window_idx_{0};
    uint64_t req_count_in_win_{0};

    mutable std::mutex mtx_;

    float unsafe_get_current_load() const {
        uint64_t total = 0, exposed = 0;
        for (const auto& w : windows_) {
            total += w.total;
            exposed += w.exposed;
        }
        return (total == 0) ? 0.0f : static_cast<float>(exposed) / total;
    }
```

```
public:
    PIDLoadController(float target_load, float kp, float ki, float kd,
                      int window_num, int window_size,
                      float max_adjust = 0.5f, float max_integral = 1.0f)
        : target_load_(target_load), kp_(kp), ki_(ki), kd_(kd),
          window_num_(window_num), window_size_(window_size),
          max_adjust_(max_adjust), max_integral_(max_integral)
    {
        windows_.resize(window_num_);
    }

    void report(bool has_ad) {
        std::lock_guard<std::mutex> lock(mtx_);
        if (req_count_in_win_ >= window_size_) {
            current_window_idx_ = (current_window_idx_ + 1) % window_num_;
            req_count_in_win_ = 0;
            windows_[current_window_idx_] = {};
        }
        auto& win = windows_[current_window_idx_];
        win.total++;
        if (has_ad) win.exposed++;
        req_count_in_win_++;
    }

    float get_current_load() const {
        std::lock_guard<std::mutex> lock(mtx_);
        return unsafe_get_current_load();
    }

    float compute_adjustment() {
        std::lock_guard<std::mutex> lock(mtx_);
        float current = unsafe_get_current_load();
        float error = target_load_ - current;

        float P = kp_ * error;
        integral_ = std::clamp(integral_ + error, -max_integral_, max_integral_);
        float I = ki_ * integral_;
        float D = kd_ * (error - last_error_);
        last_error_ = error;

        return std::clamp(P + I + D, -max_adjust_, max_adjust_);
    }
};
```

接下来是具体的主函数入口和模拟请求的调用，代码如下所示。需要注意的是，在 main 函数的 has_ad 判断的时候，出于样例代码的考虑，这里采用随机的方式进行生成。实际在工业界推荐系统中，has_ad 的判断完全是由序列生成和序列排序阶段决定的。最终序列排序阶段排出来的 top1 序列中没有广告，则 has_ad 为 0，反

之为 1。这与示例代码中是完全不同的。

代码 8.4: 带请求窗口的 PID 控制主函数代码

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <thread>
#include <atomic>
#include "PIDLoadController.h"

using namespace std;

struct Item {
    bool is_ad;
    float raw_score;
};

// 全局共享: 线程安全 PID 控制器
PIDLoadController pid(0.12f, 1.8f, 0.35f, 0.45f, 10, 500);

// 全局统计（多线程安全）
atomic<uint64_t> total_reqs{0};
atomic<bool> stop_flag{false}; // 新增全局退出标志
const int PRINT_INTERVAL = 500;
const uint64_t MAX_TOTAL = 10000; // 最大总轮次

// 单个工作线程: 无限循环并发上报
void worker_thread() {
    while (!stop_flag.load()) { // 判断停止标记
        float adjust = pid.compute_adjustment();

        float ad_base_score = 0.20f + adjust;
        float prob = clamp(ad_base_score * 100.0f, 0.0f, 99.0f);
        bool has_ad = (rand() % 100) < prob;

        vector<Item> items;
        if (has_ad) items.push_back({true, 0.78f});
        items.push_back({false, 0.81f});
        items.push_back({false, 0.79f});
        items.push_back({false, 0.75f});

        bool ad_in_request = false;
        for (const auto& item : items) {
            if (item.is_ad) {
                ad_in_request = true;
                break;
            }
```

```
        }

        pid.report(ad_in_request);

        // 全局计数 + 定时打印
        uint64_t now = ++total_reqs;
        if (now % PRINT_INTERVAL == 0) {
            float current_load = pid.get_current_load();
            cout << now << "\t\t"
                 << current_load << "\t\t"
                 << 0.12f << "\t\t"
                 << adjust << "\n";
        }

        // 到达阈值，触发全局停止
        if (now >= MAX_TOTAL) {
            stop_flag = true;
        }
    }
}

int main() {
    cout << fixed << setprecision(4);
    cout << "总请求数\t当前广告占比\t目标占比\tPID调节量\n";
    cout << "------------------------------------------------------------" << endl;
    // 启动 8 个并发工作线程
    vector<thread> threads;
    int thread_num = 8;

    for (int i = 0; i < thread_num; ++i) {
        threads.emplace_back(worker_thread);
    }
    // 等待所有线程跑完退出
    for (auto& t : threads) {
        t.join();
    }
    cout << "\n全部运行结束，总请求: " << total_reqs << endl;
    return 0;
}
```

根据这个示例的多线程 PID 调控示例程序，模拟工业界并发请求的环境下的广告 load 调控过程，最终可以打印出广告的 Load 变化情况以及 PID 调节量的大小，如下表8.5所示。本次模拟验证以目标负载系数 $r = 0.12$ 为被控设定值，采用离散 PID 控制算法对广告曝光占比实施闭环调控，模拟产生的总样本量 10000，等间隔采样得到被控量与控制量序列。从下表中的实验数据可以看出：基于 PID 的 Load 调控系统经动态调节后，被控输出稳态运行区间为 $[\mathbf{0.1170, 0.1280}]$，被控量围绕设定值小幅往复振荡，稳态偏差被限制在 ± 0.008 以内；控制器输出可依据系统偏差极性实现双向控制：实际负载高于设定值时控制量为负，抑制广告投放概率；实际负载低于设定值时控制量为正，抬升广告投放概率。

综上所述，示例代码中所整定 PID 参数可实现被控量对目标值的有效跟踪，系统动态收敛特性与稳态控制

总请求数	当前广告占比	目标占比	PID 调节量
500	0.1280	0.1200	-0.2806
1000	0.1210	0.1200	0.0406
1500	0.1246	0.1200	-0.2361
2000	0.1194	0.1200	-0.2379
2500	0.1228	0.1200	-0.2152
3000	0.1220	0.1200	-0.1053
3500	0.1206	0.1200	0.0931
4000	0.1170	0.1200	-0.0944
4500	0.1195	0.1200	-0.1642
5000	0.1232	0.1200	-0.0379
5500	0.1170	0.1200	0.0065
6000	0.1179	0.1200	-0.1822
6500	0.1233	0.1200	-0.0700
7000	0.1168	0.1200	-0.0438
7500	0.1192	0.1200	-0.2515
8000	0.1226	0.1200	-0.0466
8500	0.1186	0.1200	0.0061
9000	0.1200	0.1200	-0.2029
9500	0.1250	0.1200	-0.0447
10000	0.1233	0.1200	0.0181

表 8.5: 多线程 PID 广告 Load 调控数据表

精度满足预设控制指标，闭环控制系统具备优良的稳态调节性能。在工业界具体实践中，通常我们也会在线上实时监控 Load 调控的曲线，从而判断 PID 调控中的超参数设置是否合理，如果 PID 调控的收敛速度很慢，则需要进行手动调参，进而提升 PID 调控的灵敏度与调控力度。

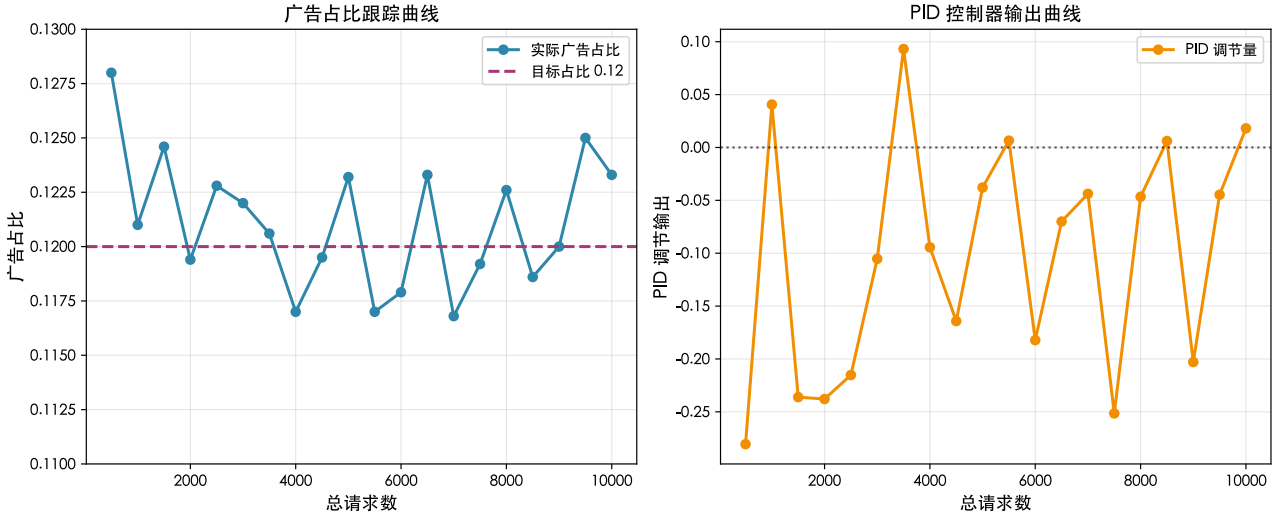

图 8.7: 广告 Load 占比及 PID 调控量监控曲线。

下面给出上述模拟示例中广告 Load 在 PID 调控过程中的监控曲线，以及对应的 PID 调控量变化情况，如图8.7所示。在工业界推荐系统的混排模块中，类似于图8.7(a) 所示的广告 Load 占比监控是非常重要的线上监控指标。通常情况下，系统会为广告 Load 设置目标区间以及上下界阈值，并配置相应的报警策略。当广告 Load 超过设定上限，或者低于设定下限时，监控系统会自动触发告警。此时，工程师需要及时排查推荐链路中的各个环节，包括召回、排序、混排以及流量调控模块，以确认是否存在模型异常、特征缺失、数据延迟或系统故障等问题，从而避免线上流量分配失衡导致用户体验下降或业务收益受损。除了广告业务之外，直播、电商、图文、短剧等业务的流量占比通常也会在混排模块进行实时监控。对于大型推荐系统而言，这些业务的 Load 监控往往构成了核心的业务健康度指标，是保障平台生态稳定运行的重要基础设施。

相比之下，图8.7(b) 所示的 PID 调控量监控曲线虽然同样重要，但其关注重点与 Load 占比监控有所不同。Load 占比监控主要用于观察最终业务流量是否处于合理区间，而 PID 调控量监控则更多用于分析控制器本身的运行状态。通过观察 PID 输出值的变化趋势，工程师可以判断 PID 控制器是否正常生效，以及当前调控力度是否合理。例如，当业务 Load 持续偏离目标值时，若 PID 调控量始终接近于零，则可能意味着控制器参数配置存在问题；而当 PID 调控量出现剧烈震荡时，则可能表明控制器过度调节，影响系统稳定性。

因此，在实际生产环境中，Load 占比监控与 PID 调控量监控通常需要结合使用：前者用于监控业务结果，后者用于监控控制过程。二者共同构成了混排流量调控模块的重要观测手段，从而保证 PID 控制策略能够长期稳定地运行，并持续发挥业务流量治理的作用。

8.6.3 强插策略

除了流量治理之外，混排后处理阶段还承担着产品运营策略落地的重要职责。在实际业务中，平台经常会举办各类运营活动，例如大型直播活动、节日营销活动、新业务推广活动等。此时运营人员通常希望某些指定内容能够获得确定性的曝光机会，而不仅仅依赖模型排序结果。

因此，系统通常会提供强插（Force Insertion）能力，允许运营人员通过后台配置规则，在特定用户群体、特定刷次以及特定坑位插入指定内容。例如：

- 指定白名单创作者的视频强插至第 5 个位置；
- 指定主播的直播间强插至用户首次刷新页面时的第 3 个位置；
- 针对特定人群包插入活动广告内容。

对于影响力极大的运营活动，例如头部明星直播、大型电商促销活动等，平台甚至可能采用全局强插策略，对所有用户统一插入指定内容，以保证活动获得充足曝光。从系统实现角度来看，强插通常发生在最终曝光前的最后阶段，其优先级往往高于混排排序结果，因此能够确保运营策略得到稳定执行。

8.6.4 保量与降权

除了流量治理和运营强插之外，混排后处理阶段还承担着平台生态调控的重要职责。在推荐系统的发展过程中，平台往往需要扶持某些新兴业务或内容生态。例如短剧、AI 漫剧、知识内容、本地生活等业务在发展初期通常缺乏足够的数据积累，如果完全依赖模型排序，可能很难获得充足曝光。

因此，平台通常会在混排后处理阶段增加保量（Quota Guarantee）机制。例如：

- 每 100 次曝光至少包含一定比例的短剧内容；
- 新创作者内容获得最低曝光保障；
- AIGC 内容进行适当的降权打压；
- AI 漫剧内容获得额外曝光加权。

与保量策略相对应的是降权（Demotion）策略。对于低质内容、重复内容、高营销感内容或用户反馈较差的内容，可以在最终曝光前进一步降低其排序分数，从而减少其曝光机会。本质上，保量与降权都是平台生态治理的重要手段。其目标并非单纯优化短期点击率或观看时长，而是在更长周期内促进内容生态的健康发展，实现平台、创作者与用户之间的长期平衡。

综上所述，混排后处理并不追求更复杂的排序模型，而更关注系统的稳定性、安全性与业务可控性。通过 Load 调节、运营强插以及保量降权等机制，推荐系统能够在复杂业务环境下保持稳定运行，并确保平台战略目标得到有效落实。

8.7 本章小结

本章围绕级联式推荐系统中的混排模块展开系统性介绍。作为多业务推荐结果融合的核心组件，混排模块主要解决异质化内容在统一推荐流中的组织、排序与流量分配问题，其本质是一个跨业务的序列级优化与资源调度问题。

首先，本章从多业务混排这一实际业务场景出发，分析了混排模块存在的必要性，使读者能够从系统架构与产品形态两个层面理解混排在级联式推荐系统中的定位与作用。在此基础上，对混排模块的整体架构进行了概述，并将其划分为混排预处理、混排序列生成与排序以及混排后处理三个核心阶段。

在混排预处理阶段，系统面临来自不同业务系统的打分体系不一致问题，不同业务之间在分数含义、数值尺度以及分布形态上均存在显著差异。针对该问题，本章重点介绍了多种业务分校准方法，包括保序回归（Isotonic Regression）、平滑保序回归（Smoothed Isotonic Regression, SIR）以及贝叶斯平滑校准方法（Bayesian SIR），并进一步讨论了业务分归一化策略在统一评分空间构建中的作用。

在混排序列生成阶段，本章分析了混排与重排在问题设定上的共性与差异。尽管两者在方法层面均可采用贪心搜索或 Beam Search 等序列生成策略，但由于混排输入来源于多个业务系统，其候选空间在结构上更为稀疏且异质，因此在工程实践中往往具备更强的计算预算灵活性。在此基础上，本章进一步讨论了带业务约束的序列生成问题，例如广告间隔（ad gap）约束等，并介绍了约束驱动下的序列搜索与剪枝策略。此外，还探讨了基于业务预竞价与混排全竞价的两类系统架构，它们在职责划分与候选生成规模上存在显著差异，并进一步影响序列生成阶段的设计选择。

在混排序列评估阶段，本章重点讨论了序列价值建模中的关键问题，包括位置偏差（position bias）对序列收益估计的影响，以及不同序列长度下价值不可比的问题。针对上述问题，工业界通常通过引入位置衰减因子以及设计统一的序列价值归一化机制来进行修正，例如通过 mock item 进行长度对齐，从而构建可比的序列级价值评估体系，实现跨业务序列之间的公平比较。

在混排后处理与流量调控阶段，本章进一步介绍了基于系统负载的流量控制方法，并重点引入了 PID 控制器在工业推荐系统中的应用，通过反馈控制机制实现对系统曝光流量的动态调节。同时，还讨论了保量策略与降权策略在业务约束落地中的作用，从而保证系统在多目标约束下的稳定运行。

总体而言，混排模块是级联式推荐系统中实现多业务协同与全局流量优化的关键环节，其核心价值在于将多个独立推荐系统的输出统一映射到同一序列空间中，在保证用户体验一致性的同时，实现跨业务目标的综合平衡。

下一章将进一步探讨生成式推荐模型的发展及其对传统级联式推荐架构的影响，并分析生成式推荐模块在现有系统中的嵌入方式及其潜在演进方向，包括其是否可能对传统推荐范式产生结构性替代。

8.8 参考文献

[1] Jan De Leeuw, Kurt Hornik, and Patrick Mair. “Isotone optimization in R: pool-adjacent-violators algorithm (PAVA) and active set methods”. In: *Journal of statistical software* 32 (2010), pp. 1–24.

[2] Chao Deng et al. “Calibrating user response predictions in online advertising”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2020, pp. 208–223.

[3] A Fielding. *Statistical inference under order restrictions. the theory and application of isotonic regression*. 1974.