

第 17 章 推荐系统在线服务

在上一章中，我们详细介绍了工业级推荐系统的离线架构，涵盖样本流服务、特征生产和模型训练等核心环节。离线链路产出的模型权重、Embedding 和物品特征索引等，最终都需要通过在线服务应用于实际推荐请求。本章将视角从离线环节转向在线环节，系统介绍推荐请求的处理流程及其背后的服务架构。

当用户打开 App、下拉刷新或进入推荐页面时，一次在线推荐请求便随之触发。从客户端发出请求到最终返回推荐结果，整个链路的端到端延迟通常需要控制在几百毫秒以内，在部分强调“秒开”体验的场景下，甚至需要压缩到 50 毫秒以内。这意味着，工业级推荐系统不仅需要满足严格的延迟约束，还必须具备应对高并发请求的能力。

表面看来，在线推荐服务似乎只需要加载模型参数、执行模型推理并返回打分结果，但实际工程远比这复杂。一次完整的推荐请求往往需要经过多个推荐子服务，涉及向量检索、倒排召回、粗排打分、精排推理和重排优化等多个环节。各个子服务通常拥有独立的计算资源、存储系统、缓存策略和故障处理机制，并通过 RPC 等方式协同完成请求。任何一个环节的延迟抖动或服务故障，都可能沿着调用链路向下游传播，影响整条推荐链路的性能与可用性。因此，如何在有限的延迟预算内保障服务高可用，如何在模型持续迭代的过程中维持服务稳定，以及如何支撑海量用户的并发访问，都是工业级推荐系统在线服务需要解决的核心工程问题。

本章将围绕在线推荐服务的架构、关键组件与稳定性保障展开。首先介绍全链路请求处理流程、高可用机制、服务监控与稳定性治理；随后讨论用户侧服务、缓存系统和索引在线服务；最后深入模型在线服务，涵盖 Embedding Server、特征抽取、TensorRT 推理，以及双塔模型框架和 DNN 精排框架等内容。通过这些内容，我们将从请求链路到核心计算组件，逐步理解工业级推荐系统如何在严格的性能约束下，实现高效、稳定且可扩展的在线推荐服务。

17.1 在线推理架构

推荐系统的在线推理架构并不是一个独立的模型服务，而是由多个子服务按照特定的调用关系协同组成的分布式系统。一次推荐请求的总延迟预算通常在 200–500 ms 之间，具体取决于业务场景，并需要在参与请求处理的各个子服务之间进行分配。当某个子服务超时或出现异常时，系统通常需要触发相应的降级或兜底逻辑，避免局部故障影响整条推荐链路，进而损害用户体验。

17.1.1 全链路请求处理流程

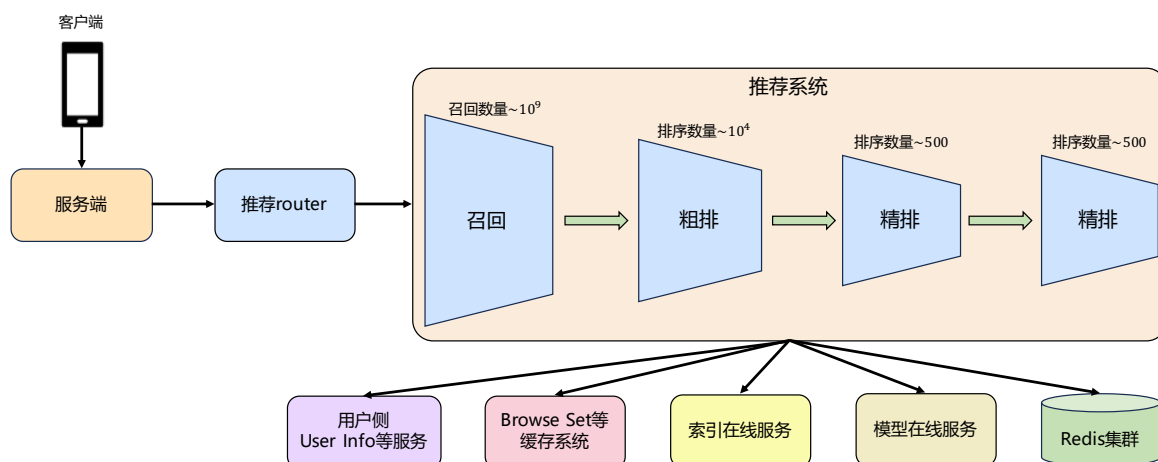


图 17.1: 推荐系统在线服务全链路架构示意图。

如图 17.1 所示，一次推荐请求从客户端发起，经由业务服务端进入推荐系统，再由推荐 Router 组织召回、排序等下游服务的调用，最终将推荐结果返回客户端。整条链路由多个职责明确的服务协同完成，各阶段既相互配合，又通过清晰的服务边界实现解耦。

请求首先由**客户端**发起。当用户在 App 中刷新信息流、进入关注页或执行搜索时，客户端通过 HTTP/HTTPS 向相应的业务服务端发送请求，其中通常包含设备 ID、用户 ID、场景标识和分页参数等信息。**业务服务端**负责鉴权、基础参数校验和业务上下文组装，并将推荐所需的信息封装为内部 RPC 请求，转发给推荐 Router。通过这种方式，面向客户端的业务协议与推荐系统内部的服务调用得以解耦。

推荐 Router 是推荐系统面向业务服务端提供的统一 RPC 服务入口。它接收推荐请求，补充场景配置和实验信息，并根据当前场景的策略编排后续服务调用。A/B 实验分桶通常在这一层或其依赖的实验服务中完成，实验分组信息随请求上下文传递至下游模块，以保证同一次请求在整条链路中使用一致的实验配置和策略。与此同时，推荐 Router 还负责汇总各阶段的处理结果，并在下游服务超时或异常时执行相应的降级与兜底策略。

随后，请求进入**多路并行召回**阶段。召回服务向多个相对独立的召回通道并行发起 RPC 请求，常见通道包括基于 ANN（近似最近邻搜索）的向量召回、基于物品标签和布尔条件的倒排召回、基于用户关注关系的社交召回，以及用于补充候选或兜底的热门召回等。各通道返回的结果经过汇总、去重和必要的初步融合后，形成规模从几百到数千不等的候选物品集合。召回阶段的核心目标是尽可能覆盖用户潜在感兴趣的内容，为后续排序提供充足的候选空间。

召回得到的候选集随后依次经过**粗排**和**精排**，候选规模逐步收窄。粗排采用计算成本较低的模型，对大量候选物品进行快速打分，并将候选集压缩至数百个。精排则使用表达能力更强、计算开销更大的深度神经网络（DNN），结合更丰富的特征进行精细化建模，进一步筛选出高质量候选。在多目标推荐场景中，点击、观看时长和互动等目标的预估结果还需要经过融合，形成用于后续排序的综合分值。

精排之后，系统通常还会执行**重排或混排**。重排从最终物品序列的整体效果出发，进一步处理内容多样性、位置约束以及物品之间的相邻关系等问题；当广告、直播等不同类型的 content 需要共同展示时，混排则根据相应的业务规则协调不同内容的展示顺序，形成最终返回用户的推荐序列。

完成上述处理后，推荐 Router 汇总最终推荐结果并返回业务服务端，由业务服务端将其转换为客户端所需的响应格式，再返回客户端。与此同时，系统通常会异步记录曝光日志，将请求上下文、候选物品、各阶段分值、实验信息和最终展示序列等数据写入 Kafka，供后续的离线样本链路使用，并与用户后续产生的点击、观看和互动等行为进行关联，构造训练样本。至此，一次在线推荐请求完成，而曝光及后续行为数据则继续流入离线链路，形成从在线服务到离线训练、再到模型更新与在线部署的数据闭环。

表 17.1 汇总了各阶段典型的候选集规模和延迟参考值，帮助读者直观理解在线推荐链路逐级筛选候选物品的漏斗结构，以及各阶段在计算开销和延迟预算上的分工。

表 17.1: 推荐在线链路各阶段典型参数（以短视频信息流为例）

阶段	输入规模	典型延迟	主要任务
多路并行召回	全量物品库（亿级）	20–30 ms	向量 ANN 检索、倒排拉链表、热门获取
召回 Merge	各路候选	5–10 ms	合并去重、Browse Set 过滤
粗排	1000–10000 物品	20–50 ms	双塔打分、LTR 预估、多目标融合
精排（特征抽取）	100–500 物品	10–20 ms	多源并行处理、特征转换与哈希
精排（模型推理）	100–500 物品	20–50 ms	GPU DNN 推理、多目标融合
重排/混排	50–200 物品	5–10 ms	序列多样性优化、广告/直播混排
曝光日志回写	—	异步	推荐请求数据写入 Kafka

17.1.2 RPC 调用、限流、熔断与兜底链路

在前述在线推荐链路中，各子服务通过 RPC（Remote Procedure Call，远程过程调用）框架相互通信。RPC 将远程服务调用封装为类似本地函数调用的接口，屏蔽了底层网络通信和数据传输的部分细节。工业界常见的 RPC 框架包括 gRPC、Thrift 以及各公司自研的 RPC 框架，它们在序列化效率、流量控制和服务发现等方面各有

特点。一次推荐请求可能涉及数十次 RPC 调用，任何一次调用的超时或失败，都可能影响最终结果。因此，除了正常的请求处理逻辑，在线服务还需要配套的流量控制和故障保护机制。

首先是**限流**，其目的是将服务接收的请求量或并发调用量控制在系统可承受的范围内，避免流量突增导致服务过载，进而引发级联故障。在工业级推荐系统中，限流既可以设置在服务入口，也可以针对推荐 Router 调用的特定下游服务进行配置。例如，当某个召回服务的处理能力有限时，可以限制发往该服务的请求量，避免其因负载过高而影响整条推荐链路。当请求超过设定上限时，系统可以拒绝部分请求，或根据业务策略跳过非关键调用、转入相应的降级或兜底逻辑。具体采用何种限流方式，需要根据服务的处理能力、资源占用和业务优先级确定。

其次是**熔断 (Circuit Breaker)**，其主要目的是避免不稳定的下游服务持续影响调用方。在工业级推荐系统中，推荐 Router 通常需要通过 RPC 调用多个下游服务。当某个服务持续出现调用失败或响应过慢时，如果 Router 仍然不断发起请求并等待超时，就可能占用大量线程和其他资源，甚至影响对其他正常服务的调用。此时，可以通过熔断机制暂时停止对该服务的调用，并根据业务策略跳过相应环节，或转入降级与兜底逻辑；待服务恢复后，再逐步恢复调用。熔断的具体实现和触发条件取决于所使用的 RPC 框架或服务治理组件，通常需要结合调用失败率、响应延迟等指标进行配置。

 **笔记** 需要注意的是，限流与熔断解决的问题有所不同：限流侧重于控制请求压力，熔断侧重于隔离持续异常的下游服务。二者相互配合、共同减少局部过载或故障对整条推荐链路的影响。不过，它们本身并不能保证推荐请求一定能够返回有效结果。当关键服务不可用时，系统还需要进一步设计合理的兜底链路。

兜底链路是在线推荐服务在正常处理路径无法继续执行时的最后保障，其首要目标不是维持原有的推荐精度，而是尽可能保证请求仍然能够返回可展示的内容，避免用户看到空白页面或错误提示。对于信息流推荐场景，一种简单且实用的方案是使用热门物品作为兜底候选。

具体而言，系统可以离线或周期性地统计热门物品，按照业务场景、内容类型等维度生成热门候选池，并将其提前加载到缓存或其他高可用存储中。当召回服务整体不可用、关键依赖持续超时，或者主推荐链路无法在规定时间内产出有效结果时，系统便可以跳过不可用的计算环节，直接从热门候选池中获取物品，组成兜底结果并返回客户端。

为了避免所有异常请求都返回完全相同的内容，热门候选池通常可以采用随机抽取或随机打乱的方式分发物品。例如，系统先根据业务场景筛选出可用的热门候选，再从中随机抽取一定数量的物品，或对候选列表进行随机打乱后截取所需数量。这样既能在主链路不可用时快速生成可展示的结果，也能减少不同请求之间的内容完全重复。不过，随机分发并不意味着不受约束地随机选择：实际实现中仍需过滤已下架、不可见或不符合当前场景要求的物品，并尽可能避免重复展示。热门候选池本身也需要持续更新，以减少长期返回陈旧内容的风险。

在工程实现上，兜底逻辑应尽量依赖独立、稳定且开销较低的资源，避免继续调用已经发生故障的服务。例如，如果主链路依赖的在线召回服务不可用，而兜底逻辑仍然需要实时查询同一个召回服务才能获得热门物品，那么这条兜底链路就失去了实际意义。因此，热门候选应提前准备并缓存在可靠的存储中，同时为兜底路径设置必要的超时限制和资源隔离，避免兜底请求进一步加重系统负担。

这里还需要区分**降级链路**与**兜底链路**。降级主要是在算力、延迟或服务资源受限时，主动调整推荐系统的计算流程，在节约资源的同时尽可能保留推荐质量。例如，减少部分计算开销较大的模型调用，或者使用成本更低的模型完成排序。兜底则主要应对正常服务路径不可用、无法按时产出有效结果等异常情况，以返回可用内容为首要目标，推荐精度通常退居其次。

因此，降级并不等于直接返回热门物品，兜底也不要求继续执行完整的个性化推荐流程。前者是在资源约束下权衡计算成本与推荐效果，后者则是在正常服务无法工作时维持最基本的服务能力。有关降级链路的设计以及降级推荐系统的具体实现，已在第2章推荐系统架构中介绍，本节不再重复展开。

总体而言，限流和熔断用于控制故障发生与扩散时的请求风险，而兜底链路则负责在主推荐路径失效时提供最后的结果保障。对于工业级推荐系统而言，在线服务的可靠性不仅取决于正常情况下的推理效率，也取决于异常情况下能否以足够低的成本，稳定地返回一组合法且可展示的推荐物品。

17.1.3 超时预算分配

超时预算分配是在线推荐服务工程设计中的重要问题。一次推荐请求的端到端延迟预算通常在 200–500 ms 之间，具体取决于业务场景。这个预算需要分配给请求链路中的各个阶段，并为网络传输、线程调度和结果组装等各个环节预留出必要的时间。如果预算过紧，子服务稍有延迟抖动就可能触发超时，从而影响推荐质量；相反，如果预算过松，则可能导致某个阶段耗时过长，挤占后续阶段的处理时间，最终使整条在线推荐链路超时 (Timeout)。

以端到端延迟预算为 200 ms 的推荐请求为例，可以将网关和接入处理的预算设置为约 5 ms，召回阶段设置为 20–30 ms，召回结果合并与 Browse Set 过滤设置为 3–5 ms，粗排设置为 10–20 ms，精排模型推理设置为 20–40 ms，重排设置为 5–10 ms，结果组装与返回设置为约 10 ms。上述阶段的预算合计约为 73–120 ms，剩余时间用于吸收网络抖动、服务调度以及其他未单独列出的处理开销，并为必要的降级与兜底操作留出空间。这些数值仅用于说明预算分配的基本思路，实际配置需要结合链路结构、压测结果和各阶段的尾延迟表现进行调整，尤其应关注 P99 延迟，而不能仅依据平均耗时分配预算。

在线推荐链路中，一个阶段经常需要同时调用多个下游服务或模型。以多路召回为例，系统可能并行调用协同过滤召回、向量召回、热门召回以及关注召回等多个通道。假设四个通道的 P99 延迟分别为 12、18、25 和 8 ms，如果依次调用，总耗时约为 63 ms；如果并行发起请求，召回阶段的耗时通常更接近最慢通道的响应时间，即约 25 ms，再加上 RPC 调度、结果传输与候选合并等开销。不过，需要注意的是，多个通道并行调用时，整体耗时取决于各通道实际响应时间的最大值。由于不同通道的延迟波动会叠加到整体尾部，整体请求的 P99 延迟并不能简单地用各通道 P99 延迟的最大值准确推导。

类似的情况也会出现在排序阶段。例如，粗排或精排完成主模型的打分之后，系统还可能并行调用多个 LTR 模型，分别计算相关性、内容质量和用户兴趣匹配度等不同维度的 LTR 分数，再将结果合并为最终排序依据。假设三个模型的单次调用耗时分别为 6、11 和 18 ms，如果它们并行发起且必须等待全部结果返回，那么这一环节的耗时主要由最慢的模型决定，约为 18 ms，而不是三者耗时之和。在实际运行中，通常还需要考虑各模型的尾延迟、排队时间和 RPC 开销。因此，并行调用虽然能够缩短整体耗时，但也可能受到最慢下游服务的制约，超时预算应重点关注关键调用的尾延迟表现。

为了在复杂的调用链路中落实这些预算，在线推荐系统通常会为不同层级的调用设置 Timeout，从而形成分层的超时控制机制。

第一层是**推荐服务的总 Timeout**，用于限定一次请求允许消耗的最长时间。例如，可以为单次推荐请求设置 200 ms 的端到端耗时预算。当某次请求即将耗尽耗时预算时，系统需要及时停止继续执行无法在剩余时间内完成的计算，并尽可能基于已有结果返回推荐列表；如果已有结果不足以满足返回要求，则转入兜底逻辑。总 Timeout 是整条推荐链路的最终耗时约束，内部各个阶段都需要在这一约束下安排计算。

第二层是**内部模块的整体 Timeout**，用于控制召回、粗排、精排以及 LTR 模型组等复杂环节的最长执行时间。以召回阶段为例，可以为整个召回过程设置 30 ms 的 Timeout。期限内返回的候选结果进入后续合并流程；对于仍未返回的非关键召回通道，可以尝试取消尚未完成的 RPC，或直接忽略其迟到结果，避免阻塞后续粗排。对于 LTR 模型调用部分，如果某个 LTR 模型超时，可以根据业务设计忽略其结果；如果关键的 LTR 模型出现超时，则可以使用默认分数或其他预先定义的降级策略。具体处理方式需要根据各模型的重要性和结果合并逻辑来确定。

第三层是**单次下游 RPC 调用的 Timeout**。例如，精排或 LTR 模块在计算过程中可能同时调用多个模型的 Infer 服务。因此，我们需要为每次调用分别设置超时限制，避免某个模型实例异常、排队过长或网络拥塞而长时间占用上游资源。单次 RPC 的 Timeout 应受所在模块的整体 Timeout 约束，而模块的预算又不能超过推荐服务的剩余总耗时预算。与此同时，还需要为结果合并、超时处理和结果返回预留时间，避免内部计算耗尽全部预算后，推荐系统仍然无法及时响应服务端与客户端。

需要注意的是，上述这几层 Timeout 并不是彼此独立的固定时间配置，而应该形成逐级约束的关系。假设推荐请求的总预算为 200 ms，当召回阶段已经耗时 28 ms 时，即使召回模块原先配置的超时上限为 30 ms，后续处理也必须服从请求当前剩余的总预算。如果后续阶段已经没有足够的时间完成某项非关键计算，就不应继续无

条件等待。因此，实际实现中通常需要在调用链路中传递截止时间或剩余预算，并在发起下游调用前据此设置合理的 **Timeout**。

此外，某个推荐服务或推荐模块触发 **Timeout** 也不一定意味着底层计算会立刻停止。调用方可以发起 **RPC** 取消请求，但能否及时终止下游计算，仍取决于 **RPC** 框架、服务端实现以及具体计算任务是否支持取消。因此，除了配置 **Timeout** 参数，还需要确保调用方能够及时释放自身资源，并对迟到结果进行丢弃或隔离，避免已经超时的请求继续消耗大量算力。

通过“推荐服务总 **Timeout**—内部模块整体 **Timeout**—单次下游 **RPC Timeout**”这样的分层设计，可以将超时控制落实到具体调用环节，并使各阶段的执行时间受到统一耗时预算的约束。其核心并不是为每个服务简单设置一个固定的超时数值，而是让整条推荐链路始终围绕端到端的延迟目标分配和使用计算资源，从而在控制响应时间的同时，尽可能保留推荐质量。

17.1.4 服务版本管理与灰度发布

工业级推荐系统中的在线服务通常采用分级发布的方式进行上线与更新，而不是一次性替换所有线上实例。无论是召回、特征抽取、排序还是结果合并组装等服务，都需要通过逐步扩大的灰度范围验证新版本的稳定性，降低代码变更带来的线上风险。

在发布之前，通常需要为服务构建产物分配明确的版本标识，并记录对应的代码提交、配置版本、依赖版本和构建时间，以便在出现异常时快速定位问题。同时，发布系统应保留上一稳定版本及其配置，确保新版本出现故障时能够及时回滚。

一个典型的发布流程可以分为三个阶段。首先，选择一台实例部署新版本，进行小规模灰度，确认服务能够正常启动、健康检查通过，并且基本的请求处理和上下游 **RPC** 调用均正常。其次，将新版本逐步扩展到一定比例的实例，例如约 20%，观察更大规模流量下的服务表现。最后，只有在前两个阶段均通过验证后，才将新版本推广到全部实例。在整个过程中，需要控制新旧版本的切换节奏，确保剩余稳定版本仍有足够的服务容量承接流量，避免发布操作本身引发服务过载。

每个灰度阶段都应设置合理的观察窗口，而不能仅凭服务成功启动就继续扩量。在灰度观察期间，需要关注服务可用性、请求量 **QPS**、错误率、超时率、**P95/P99** 延迟、**CPU** 和内存使用率、实例重启次数及健康检查结果等指标，同时检查上下游服务调用是否正常。对于推荐链路服务来说，还应该关注召回候选数量、特征缺失率、模型输出和最终推荐结果是否出现异常，比如推荐空结果率指标是否上升。如果新版本出现请求失败、延迟明显上升、进程崩溃或上下游调用异常等情况，应立即停止对推荐在线服务进行扩量，并根据问题影响范围采取回滚操作，将服务回滚到稳定版本。

对于推荐系统而言，服务层面的稳定还不够，还需要进一步验证业务指标。某些代码变更虽然不会导致服务报错，却可能影响特征处理、候选过滤、模型输入或输出结果组装，进而改变最终的推荐效果。因此，在技术指标正常的基础上，还需要对比灰度版本与稳定版本承接流量后的 **A/B** 指标，检查点击率、观看时长及其他核心业务指标是否出现异常变化。对于不应改变推荐行为的工程性更新，应重点确认核心业务指标与原有版本基本持平；如果变更本身涉及推荐策略或模型逻辑，则应结合预期收益和业务保护指标进行判断。

灰度观察时间还需要结合服务的重要性、流量规模和指标的统计稳定性来进行确定。对于流量波动明显的推荐业务，可以将观察窗口延长至覆盖一个完整的主要流量周期，必要时观察一整晚，以确认新版本在不同流量条件下的运行表现。只有在服务运行稳定、上下游调用正常，且业务指标达到预期之后，才能继续扩大灰度范围并最终完成全量发布。

推荐在线服务全量发布之后，服务仍然应该保留必要的监控与回滚能力，并确认所有实例最终运行的代码与配置版本一致。通过从单实例灰度、部分实例扩量到全量发布的逐级验证机制，可以将潜在问题尽量限制在较小范围内，同时结合技术指标和线上业务指标验证代码变更的正确性，从而保障工业级推荐系统在线服务的稳定运行。

17.1.5 性能与容量规划

性能优化和容量规划是在线服务持续运营中的日常工作。**P99 尾延迟**是衡量服务性能与稳定性的关键指标之一，它表示 99% 的请求延迟不超过该值，而剩余约 1% 的请求具有更高的延迟。对于流量规模庞大的推荐系统，即使只有少量请求落入延迟分布的尾部，也可能影响大量用户的访问体验。因此，性能优化不能只关注平均延迟，还需要持续分析 **P95**、**P99** 等高分位延迟指标，定位少数慢请求背后的原因。

尾延迟升高不一定是正常计算过程本身变慢，也可能由运行时暂停、资源竞争、网络抖动和热点请求等因素引起。例如，Java 服务中的垃圾回收（GC）可能造成一定程度的运行暂停，Full GC 在某些情况下还可能带来较长的停顿；高并发下的锁竞争会使请求在临界区前排队等待；网络拥塞、请求重传以及线程调度也可能引入额外延迟。此外，少数请求的计算开销可能远高于普通请求，例如，部分用户具有特别长的历史行为序列，导致特征读取、序列处理和模型推理的成本显著增加。针对这些问题，可以结合推荐 Router 及推荐链路服务的系统日志与性能分析工具来定位运行时的瓶颈，从而缩短临界区、减少不必要的锁竞争，利用连接池降低连接建立开销，并对异常长的行为序列设置合理的长度上限。对于 Java 服务而言，还可以根据实际负载和延迟目标调整 GC 策略与相关参数，避免仅凭经验套用固定的配置。

容量规划则需要回答另一个问题：

 **笔记** 为了稳定承接预期流量，需要部署多少服务实例？

它的基本方法是通过压力测试，测量单个实例在满足延迟和错误率等服务等级目标（Service Level Agreement, SLA）的前提下，能够持续承载的请求量，并据此估算所需实例数量。假设历史峰值 QPS 为 Q_{peak} ，单实例在目标 SLA 下的稳定处理能力为 Q_{instance} ，冗余系数为 k ，则初步的容量估算可以写为：

$$N = \left\lceil \frac{k Q_{\text{peak}}}{Q_{\text{instance}}} \right\rceil \quad (17.1)$$

其中，冗余系数 k 可以根据流量波动、扩容速度和故障场景设定，例如取 1.5–2.0 作为初始估算值，但并不存在适用于所有服务的固定比例。这里的单实例处理能力应以满足目标 **P99** 延迟等要求时的实测结果为准，而不是简单地取压测中能够达到的最高 QPS。容量规划还需要考虑发布期间的新旧实例共存、单实例故障以及机房故障等情况，确保部分资源不可用时，剩余实例仍能承接必要的流量。对于流量具有明显周期性变化的业务，还可以配置弹性扩缩容，在高峰到来前提前扩容，并在流量回落逐步缩容，以兼顾服务稳定性和资源成本。

对于 GPU 推理服务而言，容量规划还需要额外关注显存容量、GPU 计算能力和设备利用率。受显存限制，单张 GPU 能够同时加载的模型数量有限；而在推荐请求较为稀疏、单次推理计算量较小的场景下，GPU 也可能无法得到充分的利用。因此，GPU 服务的容量评估不能只看模型能否成功加载，还需要结合实际请求的分布，分析吞吐量、批处理规模、单次推理延迟和 GPU 利用率之间的关系。如何在满足延迟 SLA 的前提下提高 GPU 利用率，是模型在线服务的重要优化方向，后续将在本章的模型在线服务部分结合 **Dynamic Batching** 等技术进一步讨论。

多机房容灾也是大规模推荐系统容量规划的重要组成部分。为了避免单个机房发生故障后导致整个推荐服务不可用，工业级推荐系统通常会采用多可用区（Multi-AZ）部署，将同一个服务的多个实例分布在不同机房或可用区中，并通过负载均衡和流量调度机制将请求分发到可用实例。在正常情况下，多个机房可以共同承接推荐请求；当某个机房因网络中断、供电故障或大规模硬件异常而无法正常工作提供服务时，系统可以将流量切换到其他可用机房，维持核心推荐能力。需要注意的是，多机房部署不仅要求实例分布在不同故障域，还需要为其他机房预留足够的计算和存储容量，避免故障切换后剩余实例因流量骤增而过载。对于依赖本地缓存、模型文件或索引数据的服务，还应提前完成必要的数据加载与服务初始化，确保其他机房具备接管流量的能力。

在多机房部署中，除了服务实例需要跨机房分布，模型发布、**Embedding** 更新和索引构建也需要具备相应的版本管理与同步机制。不同机房应能够获取并加载所需的模型、特征数据和物品索引，并通过版本标识、健康检查和就绪校验，确认实例具备服务能力后再承接流量。由于不同数据的更新频率和一致性要求并不相同，实际系统可以根据业务需求采用同步或异步的分发方式，并持续监控数据从生成到各机房生效的传播延迟。当发生故障切换时，还需要关注各机房间的模型与索引版本是否一致，避免因数据版本差异引入额外的推荐偏差。多

机房容灾并不意味着所有数据必须在任何时刻完全同步，而是要在满足业务一致性要求的前提下，确保单个机房发生故障时，其他机房能够及时接管流量并持续提供稳定的推荐服务。

总体而言，在线服务的性能与容量规划需要从单个请求的尾延迟、单实例的处理能力，逐步扩展到整个服务集群及多机房的可用性。通过持续分析尾延迟来源、依据 SLA 进行容量估算、提高推理资源利用率，并为机房故障预留足够的承载能力，才能在流量持续变化和局部故障发生时，维持推荐系统的稳定运行。

17.2 服务监控与稳定性治理

完善的监控体系是保障在线服务稳定运行的基础。推荐系统在线服务出现的问题往往具有较强的隐蔽性，例如模型输出分值的数据分布发生偏移、某路特征的缺失率逐渐升高，或者某个子服务在流量高峰期出现 P99 延迟抖动。如果这些异常不能被及时发现，往往要等到点击率、播放时长等业务指标出现明显下滑后才会暴露，此时问题可能已经持续了一段时间。因此，推荐系统的监控不能只关注最终业务效果，还需要覆盖请求链路、模型输出、特征质量以及服务运行状态等多个不同的层面，以便尽早发现异常并定位原因。

链路延迟监控是在线服务监控的基础。系统需要分别统计召回、粗排、精排、重排等关键阶段的 P50、P95 和 P99 延迟，并持续观察这些指标随时间的变化。当完整的推荐请求 P99 延迟升高时，可以结合各阶段的耗时分解，判断问题主要来自特征抽取、模型推理、下游 RPC 调用还是网络延迟与调度开销等方面。例如，如果精排阶段的整体延迟上升，就可以进一步拆分特征抽取与模型推理环节的耗时，判断是特征抽取服务响应变慢，还是推理计算本身出现了瓶颈。通过这种层次化的监控，可以缩小排查范围，避免仅凭整体推荐指标来反复猜测故障原因。对于包含多路并行调用的阶段，还需要结合各下游服务的延迟分布进行分析，因为少数慢调用就可能拉高整个阶段的尾延迟。

模型分值分布监控是推荐系统特有的重要监控维度。对于召回、粗排以及精排等阶段，可以持续统计模型输出分值的均值、方差以及 P10、P50、P90 等分位数，观察相同场景、相近流量条件下的分值分布是否出现异常变化。例如，如果某个模型的输出均值突然从 0.3 降至 0.05，或者分值分布明显变宽，就需要检查模型版本、输入特征、候选集分布或者实验参数配置是否发生变化。这类异常有时候会先于业务指标的变化出现，因此可以作为提前发现问题的重要信号。不过，模型输出分值变化并不一定意味着模型出现故障，因为候选物品构成、流量来源、节假日效应以及正常的模型更新也可能改变分值分布。因此，监控告警应结合具体模型版本、业务场景和流量分组设置基线，并与特征质量以及业务指标联合分析，避免将正常的变化误判为系统故障。

特征质量监控关注在线特征生产与获取链路是否正常。用户行为序列、物品 Embedding 和统计特征等都可能缺失、过期或者异常取值。例如，当某路特征的缺失率突然升高时，可能意味着对应的 User Info 服务、正排索引或 Embedding Server 出现故障，也可能是特征更新或数据同步链路发生异常。对于每一路特征，都应结合历史数据建立合理的质量基线，并根据其重要性设置缺失率、取值范围和更新延迟等监控指标，而不宜为所有特征统一设定固定的缺失率阈值。此外，还可以将在线特征的均值、分位数和取值分布与离线训练样本进行对比，以发现训练与服务之间的特征偏差，即 Training-Serving Skew。通过对特征质量进行持续监控，可以在异常进一步影响模型输出和推荐结果之前发现问题。

业务指标监控用于评估在线推荐服务最终产生的业务效果。点击率（CTR）、播放完成率、互动率和人均播放时长等指标，可以反映推荐结果是否满足预期。与技术指标相比，业务指标通常受到流量结构、用户活跃度、内容供给以及日内周期等多种因素影响，因此不能仅凭某个时刻的短暂波动就判断系统发生故障。在实际工程中，通常可以结合同比与环比进行观察，例如对比上周同一时段的数据，或观察最近一段时间的变化趋势，并根据业务特点设置相应的基线。对于节假日、重大活动或流量结构明显变化的场景，还需要使用更合适的对照周期，避免将正常的业务波动误判为异常。当业务指标持续偏离预期时，应该进一步结合链路延迟、模型分值和特征质量等技术指标进行排查，从而区分推荐效果变化与服务故障。

分布式链路追踪主要用于定位单次请求中的具体问题。当某个请求的延迟明显异常，或者用户反馈推荐结果不符合预期时，分布式链路追踪系统可以还原请求经过各个子服务的调用过程，包括每个服务的处理耗时、下游调用关系、返回状态以及异常信息。常见的追踪系统包括 Jaeger 和 Zipkin。其核心机制是为请求分配 Trace ID，并通过 RPC 上下文将相关追踪信息传递给下游服务，使不同服务的日志和调用记录能够关联起来。在请求入口

或推荐 Router 处生成 Trace ID 后，各个子服务便可以使用同一标识记录调用过程，工程师据此还原请求的执行路径，无需仅依赖不同服务的日志和时间戳进行人工比对。考虑到大规模推荐系统的请求量和追踪数据开销，实际部署中通常还需要配置合理的采样策略，并重点保留异常请求、慢请求等具有较高排查价值的追踪数据。

除了监控指标本身，**告警策略**也直接影响稳定性治理的效率。如果告警阈值过于敏感，就会产生大量误报，在长期积累后反而容易导致工程师忽视真正重要的通知。而如果阈值过于宽松，则可能使故障长时间得不到处理。因此，告警规则不仅需要设置合理的阈值，还应考虑异常持续时间、影响范围和业务重要性，并根据严重程度分级处理。一种常见的做法如下：

定义 17.1 (告警分级)

P1 级告警对应整体服务不可用、核心业务指标断崖式下跌等严重故障，需要立即通知并快速响应。

P2 级告警对应某条召回通道延迟持续升高、关键特征缺失率超出阈值等局部异常，需要及时排查。

P3 级告警则可以用于记录暂时未影响服务的性能趋势或资源水位问题，主要通过监控看板跟踪。

具体的分级标准和响应时限应根据团队的值班机制和业务影响来确定。



此外，推荐系统还需要将服务版本与运行状态纳入监控范围。例如，各服务实例当前加载的模型版本、Embedding 表版本和特征 Schema 版本等，都应具有可查询、可对比的元数据。当部分实例更新失败，导致同一服务中出现新旧版本混跑时，系统可能在没有明显请求报错的情况下产生不一致的推荐结果。通过持续核对版本信息、实例健康状态和发布记录，可以更早发现此类问题，并在必要时停止扩量或执行回滚。

随着监控与稳定性治理体系逐步完善，还可以进一步引入 **SLO (Service Level Objective, 服务等级目标)** 与 **错误预算 (Error Budget)** 等方法，将“服务需要稳定运行”转化为可量化、可持续跟踪的目标。SLO 是对服务性能或可用性提出的具体目标，例如，可以要求精排服务在滚动 30 天窗口内，至少 99.9% 的请求能够在 20 ms 内完成；也可以要求模型输出有效分值的请求比例达到 99.95%。前者衡量请求延迟是否满足要求，后者衡量服务能否正常产出有效结果，两者应分别统计。

错误预算则表示在给定统计窗口内，服务距离目标允许的最大偏差还有多少空间。例如，如果将 30 天内的请求可用性目标设为 99.9%，则对应的不可用时间预算约为 43.2 分钟；如果实际不可用时间持续增加，错误预算就会逐渐消耗。当错误预算消耗过快时，例如在统计周期尚未过半时已经消耗了大部分预算，工程架构团队就可以考虑降低高风险变更的频率，优先处理稳定性问题。而当错误预算较为充裕时，则可以在风险可控的前提下正常推进模型迭代和系统优化。这样，稳定性建设与系统迭代之间的权衡就可以更多地依据运行数据进行决策，而不必完全依靠经验判断。

表 17.2: 推荐系统在线服务四类核心指标监控维度

监控维度	核心指标	异常信号及可能原因	典型告警级别
链路延迟监控	各阶段 P50、P95、P99 延迟	单阶段延迟持续升高，可能存在推理、特征取数或网络瓶颈	P1 (整体系统严重异常) / P2 (局部阶段异常)
模型分值分布监控	分值均值、方差及 P10/P50/P90	分值分布异常变化，可能与模型版本、特征或候选集变化有关	根据偏移程度与业务影响确定
特征质量监控	特征缺失率、异常值比例、在线与离线分布差异	特征缺失或分布偏移，可能存在上游服务、数据更新或特征逻辑异常	P2 (关键特征异常) 或按影响升级
业务指标监控	CTR、播放完成率、互动率、人均播放时长	核心指标持续偏离基线，可能涉及服务故障、流量变化或推荐效果异常	P1 (严重下跌) / P2 (持续趋势异常)

对于工业级推荐系统而言，还需要区分服务层面的工程目标与最终的业务效果。工程层面的 SLO 主要关注延迟、请求成功率和有效结果输出率等指标；CTR、观看时长和推荐多样性等则属于业务效果指标，容易受到流量结构、实验分组及内容供给变化的影响。两类指标需要分别监控，并通过关联分析和线上实验判断它们之间

的关系，从而区分服务稳定性问题与推荐策略、模型效果变化。表 17.2 汇总了本节介绍的四类核心指标监控维度及其典型异常信号。分布式链路追踪、告警分级和 SLO 管理则分别从请求排查、异常响应和目标治理等角度，对这些监控机制进行补充。

17.3 用户侧服务

用户侧服务是推荐系统在线服务中承载用户数据的基础设施。在一次推荐请求中，推荐 Router 需要从用户侧服务获取用户画像、历史行为序列、实时状态和统计特征等信息，共同刻画用户的基本属性、近期兴趣与长期偏好，为后续的个性化召回与排序提供数据基础。

17.3.1 User Info 服务

User Info 服务是用户侧数据面向在线推荐链路的统一服务入口，通过 RPC 接口向推荐 Router 服务提供所需的用户数据。将用户数据封装成为独立的 RPC 服务，而不是由推荐 Router 服务直接访问 Redis 等底层数据源，主要有两个原因：一方面用户数据通常分散在不同的存储系统中，例如用户基础属性、实时行为和离线统计特征可能分别由不同的数据链路生产与维护。User Info 服务可以统一组织这些数据的读取逻辑，屏蔽底层存储与数据来源的差异。另一方面，User Info 服务还可以集中处理缓存访问、数据缺失和读取失败等情况，为调用方提供统一的数据获取接口，减少推荐 Router 服务对底层数据基础设施的依赖。

具体来说，User Info 服务维护的用户数据通常包括以下几类。**用户基础属性**主要包括用户 ID、设备 ID、性别、年龄段、地域、注册时间、首次活跃时间以及是否为新设备等信息。这类数据更新频率通常较低，适合通过 Redis 等缓存进行存储和读取，从而减少在线请求访问后端存储的开销。

用户行为 Action List 是 User Info 服务中非常重要的一类数据，也是用户短期兴趣建模的主要输入。它记录用户近期与推荐系统发生交互的行为序列，典型字段包括曝光列表 (real_show_list)、点击列表 (click_list)、点赞列表 (like_list)、关注列表 (follow_list)、转发列表 (forward_list) 以及不感兴趣列表 (hate_list) 等。以曝光和点击列表为例，其保留规模可以分别设置为不超过 1000 条和 200 条。序列长度的具体取值需要结合业务需求、存储成本和在线读取开销来确定。此外，用户行为序列中的每个 Action Item 通常包含物品 ID、作者 ID、毫秒级时间戳、类目或标签 ID、页面类型以及行为类型等信息，以支持后续的序列统计、特征交叉和兴趣建模。Action List 通常由用户行为日志的实时处理链路持续更新，能够反映用户近期的兴趣变化。与离线统计特征相比，这类数据更新更频繁，对写入效率、数据新鲜度和在线读取延迟的要求也更高。

用户实时状态主要用于描述用户当前一次 App 使用过程中的即时行为，包括当前 Session 内的行为序列以及视频播放信息统计等。这些信息能够帮助推荐系统识别用户当前的消费状态。例如，用户可能正在连续观看某一类视频，也可能快速划过多个内容，或者正在集中浏览某个特定主题。将当前 Session 的行为与较长时间范围内的历史行为结合起来，有助于模型及时捕捉用户兴趣的短期变化。

统计特征与偏好画像主要由离线统计任务或实时、准实时统计链路计算得到，包括用户对不同内容 Tag 的偏好强度、对不同内容 Cluster 的偏好、明确不感兴趣的 Tag 或 Cluster 列表，以及用户流失风险分数等。这类特征能够从聚合的角度反映用户相对稳定的兴趣和行为倾向。根据业务对时效性的要求，它们可以按分钟级或小时级等不同频率更新，并写入 Redis 等在线存储，供推荐请求低延迟读取。相关特征的离线生产过程已在第 16 章中介绍。

离线挖掘特征则来自更复杂的离线模型或数据挖掘算法。例如，可以基于用户的长序列行为挖掘长期兴趣 Trigger 列表，供长期兴趣在线服务检索或匹配使用；也可以根据设备型号推断机型价格区间，或者通过分类模型预测用户的性别和年龄段。这类特征通常具有较高的计算成本，不适合在每次推荐请求中实时计算，因此一般采用离线生产、在线存储和按需读取的方式接入推荐链路。

由于 User Info 服务位于推荐 Router 中数据获取的关键路径上，其响应延迟会直接影响后续的推荐链路。因此，服务通常需要将 P99 延迟控制在较低水平，例如在具体业务场景中以 5 ms 作为优化目标。为了满足这一要求，User Info 服务也可以结合缓存策略与用户活跃度来管理数据读取路径。对于活跃用户，可以提前将其常用

数据加载到 Redis 等缓存中；而对于低活跃用户，则可以在缓存未命中时按需访问 KV 存储等底层数据源，并根据策略将读取结果回写缓存。这种方式能够在控制缓存资源消耗的同时，兼顾在线读取效率。随着用户规模和请求 QPS 增长，用户数据还需要通过分片策略分布到多个存储节点，避免数据集中在少数节点上形成访问热点。

在数据链路设计上，User Info 服务通常采用读写分离的方式。其中，Action List 的写侧由用户行为日志的实时处理任务负责，当用户发生点击、播放、点赞或其他交互行为时，处理链路将相应记录更新到用户的 Action List，并维护相关实时状态。这些内容我们已经在第16章中进行过介绍，这里不再赘述。Action List 的读侧主要服务于推荐请求，由推荐 Router 服务通过 RPC 获取所需的数据。上述 Action List 读和写两条链路分离之后，可以分别针对各自的负载特征进行优化。比如写侧可以采用批量写入、异步处理等方式来提高吞吐量，而读侧则以低延迟和稳定响应为主要目标，避免高频行为写入直接干扰在线读取。

通过将用户基础属性、行为序列、实时状态、统计画像和离线挖掘结果统一组织起来，User Info 服务为推荐链路提供了相对稳定的数据访问接口。其工程设计不仅需要考虑数据是否完整、更新是否及时，还需要在用户规模、访问并发和存储成本不断增长的情况下，持续保障在线读取的延迟与可用性。

17.3.2 用户长期兴趣在线服务

User Info 服务中的 Action List 通常只保留最近几百条行为，而用户的完整历史行为可能达到数千条甚至上万条。这些长期历史能够反映用户相对稳定的兴趣偏好，但其数据规模决定了它们不适合直接放入 User Info 服务的响应体中，随每次推荐请求一并返回。完整行为序列可能占用数 MB 的存储空间，如果在每次请求中全量传输和序列化，不仅会增加网络开销，还会显著加重服务端的处理负担。因此，工业级推荐系统通常会通过独立的长期兴趣服务管理用户的完整历史，并根据在线请求的需要提取相关信息。

长期兴趣服务面临的核心挑战是：如何在严格的延迟预算内，从海量历史行为中提取与当前推荐任务最相关的兴趣信号。一种常见思路是 SIM (Search-based Interest Model)，即基于检索的兴趣建模。系统将用户的长期行为序列存储在高性能 KV 存储中，在在线请求中根据当前候选物品或目标内容构造检索条件，从用户的历史行为中筛选相关记录，并选取 Top-K 条行为作为长期兴趣输入，供下游模型进一步建模。这种方式无需在每次用户行为更新后都重新计算完整的长期兴趣表示，能够保留较细粒度的历史行为信息，也可以根据当前候选物品动态提取相关兴趣。不过，在线检索会增加额外的存储访问、计算和网络开销，历史序列越长、检索规模越大，对延迟预算的压力也越大，因此需要合理设计检索策略和返回规模。

另一种方案是采用**离线压缩表示**。系统通过聚类、兴趣抽取或其他压缩算法，将用户的长期历史行为转化为固定长度的兴趣向量、兴趣类别分布或少量兴趣原型，再将计算结果写入 Redis 等在线存储，供推荐请求直接读取。相比在线检索，这种方式能够显著减少每次请求需要处理的数据量，适合对延迟要求严格的场景。但压缩表示需要定期重新计算，而且可能损失原始行为中的部分细节；当用户兴趣发生变化时，压缩结果也可能存在一定的更新滞后。

在线检索和离线压缩并不是互斥的方案，实际系统可以根据算力条件、延迟要求和兴趣信息的使用方式进行选择，必要时也可以结合使用。例如，可以通过离线压缩表示提供用户长期兴趣的整体 Cluster 概况，再通过在线检索补充与当前候选物品直接相关的历史行为。

最后，读者需要注意的是，用户的长期兴趣与短期兴趣对应不同的时间尺度。短期兴趣主要来自当前 Session 内的行为，变化快，更适合捕捉用户此时此刻的消费偏好；而长期兴趣则反映用户在较长时间范围内形成的相对稳定的偏好结构。推荐系统通常会分别处理这两类信息，并在模型输入阶段进行融合：短期兴趣有助于及时响应用户当前行为，长期兴趣则能够弥补短期行为较少或兴趣信号不明显时的信息不足。二者共同为个性化召回与排序提供支持。

17.3.3 近线服务

推荐系统中的特征计算和模型推理既可以在请求到达时实时完成，也可以提前计算并将结果保存到在线存储中。在线计算能够及时利用最新的用户行为和模型输入，但需要在每次请求到来时消耗相应的 CPU、GPU 和存储访问资源。离线计算则通常以小时级或更长周期运行，难以满足所有需要分钟级更新的业务需求。在两者

之间，近线服务（Near-line Serving）通过持续运行的计算任务，在分钟级或其他适当周期内完成数据更新与结果预计算，为在线服务提供可以直接读取的结果。

近线服务通常是在在线计算的算力、延迟或成本受到约束时采用的一种优化手段。如果现有计算资源充足，且每次请求实时执行模型推理能够满足延迟要求，那么直接实时计算通常更有利于利用最新行为信息，无需额外承担预计算、缓存维护和数据更新的成本。只有当某些计算开销较大、请求频率较高，或者实时计算难以在资源预算内稳定完成时，才更有必要考虑近线预计算，以一定的数据新鲜度为代价，换取在线计算量和响应延迟的下降。

在工程实现上，近线服务通常由持续运行的 **Runner** 进程或周期性调度任务组成。任务按照设定的周期读取最新模型参数、原始数据或增量数据，执行批量计算，再将结果写入 **Redis**、**ANN** 索引或其他在线可读的存储系统。具体的更新周期需要根据数据规模、算力资源和业务对时效性的要求确定，并不一定统一采用固定的分钟级周期。

在双塔召回模型中，近线服务是比较典型的应用场景。**User Tower** 可能需要对用户的历史行为序列执行 **Attention** 等计算。当模型结构较复杂、用户行为序列较长时，每次请求都重新计算 **User Embedding**，可能占用大量 **GPU** 资源。尤其在高 **QPS** 场景下，即使单次推理的耗时可以接受，全量请求产生的累计计算成本仍可能很高。在算力不足或实时计算成本较高的情况下，可以由近线 **Runner** 周期性地为活跃用户计算 **User Embedding**，并将结果写入 **Redis** 或 **Embedding Server** 等存储。在线请求到达时，推荐系统直接读取已经计算好的向量，减少重复执行 **User Tower** 的开销。其代价是 **Embedding** 不能立即反映计算周期内发生的所有新行为，因此是否适合这种方式，取决于用户兴趣的变化速度以及业务对特征新鲜度的要求。对于需要充分利用当前 **Session** 行为的模型，可以采用实时计算与近线结果相结合的方案，在复用已有长期兴趣表示的同时，实时补充最新的用户行为特征。

此外，物品侧 **Embedding** 也可以采用类似的近线计算方式。**Runner** 定期读取最新的 **Item Tower** 模型参数，对全量物品或发生变化的物品批量计算 **Embedding**，并将结果写入 **ANN** 索引等存储系统中，供向量召回或者后续特征抽取以及模型推理计算来使用。与用户侧 **Embedding** 相比，物品侧计算通常更适合批量化处理，因为同一个物品向量可以被多个用户请求复用。但当模型版本发生变化或新物品大量进入系统时，仍然需要合理安排计算周期、增量更新和索引切换，避免计算任务无法及时完成而影响在线召回。

除了模型 **Embedding**，近线服务还可以用于刷新统计特征和用户画像。例如，用户对不同 **Tag** 的偏好分布、近期行为统计量以及内容聚合特征等，可以由 **Flink** 等流式计算任务持续产出，再通过近线任务或数据同步链路更新到 **Redis**。在线特征抽取服务便可以直接读取这些结果，而不必在每次推荐请求中重复执行统计计算。这类应用与 **Embedding** 预计算的区别在于，前者主要提前完成统计聚合，后者则提前完成模型推理，但二者的共同目标都是减少在线请求中的重复计算。

近线计算还可以用于优化用户打开 **App** 时的首刷加载体验。对于活跃用户，推荐系统可以在其可能访问的时段到来之前，提前执行召回和粗排，生成一批候选物品并保存到现在存储中。当用户打开 **App** 时，推荐系统可以优先读取已有的候选结果，再根据当前请求执行必要的过滤、精排和重排，从而减少首刷时需要等待的计算量。与此同时，系统仍需保留正常的在线推荐路径：如果预计算结果不存在、已经过期或不满足当前场景要求，就应及时转入实时召回等处理流程，避免等待近线任务完成。对于内容更新频繁的短视频业务，候选结果的有效期和更新周期需要结合用户活跃规律、内容变化速度及缓存命中情况进行调整。

近线服务也可以帮助新物品更快进入向量召回系统。例如，新视频刚上传时，如果尚未完成全量物品 **Embedding** 的更新，系统可以先利用标题、标签和类目等内容特征，通过当前可用的 **Item Tower** 进行轻量级前向推理，生成初始 **Embedding** 并写入 **ANN** 索引等在线存储。这样，新物品便有机会在下一轮全量刷新之前进入向量召回，缩短内容发布到获得初始召回机会之间的时间。不过，初始 **Embedding** 的质量仍取决于模型能否仅根据现有内容特征有效表征新物品，不能保证所有新物品都能立即获得理想的召回效果。

总体而言，近线服务的核心价值在于将适合提前完成的计算从在线请求中分离出来，在控制资源开销的同时满足业务对数据新鲜度的要求。它并不是对实时计算的简单替代，而是在线算力、更新及时性和推荐效果之间的一种工程权衡。只有在明确计算瓶颈和业务时效要求后，才能判断哪些任务适合近线化，以及应当采用多长的更新周期和怎样的结果发布机制。

17.4 推荐缓存服务

推荐系统在线服务中存在多种用途不同的缓存，它们分别用于复用已有推荐结果、过滤历史曝光物品以及缩短用户首刷的等待时间。如果混淆不同缓存的职责，就可能出现重复推荐、降级结果不足或缓存失效后在线计算压力骤增等问题。本节重点介绍推荐结果缓存、Preload 缓存和 Browse Set 缓存的设计思路，以及实际工程中需要关注的缓存一致性与失效处理问题。

17.4.1 推荐结果缓存服务

推荐结果缓存服务主要用于复用已经生成的推荐结果，也是降级推荐系统中一种简单实用的实现方式。例如，系统可以在一次正常推荐请求成功完成后，将该用户对应的 **Top-K** 推荐物品写入 **Redis**。当正常推荐链路算力资源不足时，系统便可以直接读取缓存中的物品，跳过部分计算环节，继续为用户提供推荐内容。

假设一次请求需要向用户曝光 N 个物品，如果缓存中只保存这 N 个物品，那么缓存结果只能满足一次请求。为了支持多次请求，可以在缓存中保存数量更大的候选列表，即令 $K > N$ 。例如，当一次请求需要返回 10 个物品时，系统可以缓存 50 个候选物品。在后续请求中，每次从尚未使用的候选中选取一定数量的物品，并过滤已经曝光过的内容，从而在短时间内通过多次请求复用同一份缓存。系统还需要记录已使用的位置或维护相应的消费状态，避免重复返回同一批物品。当缓存即将耗尽或已经过期时，再尝试恢复正常推荐链路，或者转入热门物品兜底策略。

推荐结果缓存的 **TTL** 通常设置为几十秒到几分钟，具体取决于推荐结果的新鲜度要求和降级策略。对于内容更新频繁的短视频场景，可以采用相对较短的 **TTL**，例如 10–60 s。由于这类缓存主要承担结果复用和异常情况下的降级职责，其设计重点不仅是提高缓存命中率，还包括保证缓存中有足够多的可用候选，以及避免缓存失效后所有请求同时涌向完整的推荐链路。

17.4.2 Preload 缓存服务

Preload 缓存（首刷缓存）服务主要用于优化用户进入 App 后首次打开推荐信息流的体验。用户开始一个新的 Session 时，推荐系统通常需要重新获取用户状态、执行召回以及排序、服务端和客户端还会进行组件加载与渲染等操作，才能最终显示第一屏的推荐内容。即使各个服务的延迟都满足要求，完整推荐链路的计算与调用、服务端与客户端的处理与加载仍然会增加首刷等待时间。对于强调快速打开和即时反馈的短视频 App，首刷加载速度会直接影响用户体验。

一种实现方式是提前为近期活跃用户生成首批推荐候选，并将结果写入 **Redis** 等在线存储。当用户打开 App 并发起首个推荐请求时，系统优先读取预先生成的候选列表，再执行必要的过滤、精排和重排，从而减少首刷阶段的计算开销。另一种方式则是为首刷设计相对简化的推荐链路，例如减少部分耗时较高的召回或特征计算环节，在满足基本推荐质量要求的同时尽快返回第一屏内容。因此，首刷优化并不一定完全依赖缓存，也可以通过缓存与简化计算相结合的方式实现。

首刷缓存的更新周期和 **TTL** 需要兼顾结果新鲜度与缓存命中率。对于内容变化较快的短视频场景，可以根据用户活跃规律和内容更新速度，将缓存有效期设置为几分钟，例如 3–15 min。如果缓存过早失效，用户真正打开 App 时就可能无法命中；如果有效期过长，缓存中的候选物品则可能无法及时反映用户最新的兴趣变化或物品供给变化。因此，系统需要结合缓存命中率、结果过期比例以及首刷延迟等指标来调整更新策略。当 Preload 缓存不存在、已经过期或者无法满足当前请求时，则应该及时回退到正常的在线推荐链路或者经过验证的简化版推荐链路，而不是等待缓存重新生成。

17.4.3 Browse Set 缓存服务

Browse Set（已曝光过滤集）服务用于记录用户近期已经曝光过的物品 ID，并在召回结果合并、候选过滤以及缓存结果复用等环节排除不应重复展示的物品。与推荐结果缓存不同，Browse Set 缓存服务并不负责提供推荐

候选，而是为推荐链路提供历史曝光状态，帮助系统减少重复曝光，改善用户的内容消费体验。

Browse Set 服务通常可以使用 **Redis Set** 或者 **Bloom Filter** 等数据结构实现。**Redis Set** 可以精确记录物品 ID，但当用户数量和曝光记录规模较大时，存储开销也会随之增加。**Bloom Filter** 则具有较高的空间效率，适合存储规模较大的曝光集合，但存在一定概率的假阳性，即将实际上未曝光过的物品误判为已经曝光，从而使部分候选被错误过滤。具体采用哪种方案，需要结合曝光记录规模、过滤精度要求和存储成本进行权衡。

在实际应用中，还需要明确 **Browse Set** 服务的记录范围、容量上限和过期策略。例如，可以只保留最近 3~7 天的曝光记录，或者根据业务需求限制每个用户维护的物品数量。对于信息流推荐，**Browse Set** 服务不仅要用于过滤新召回的候选物品，也应在读取推荐结果缓存或首刷缓存时发挥作用，避免这些缓存返回的内容与用户已经看到的内容重复。不过，过滤的范围不能无限扩大。如果历史曝光记录保留时间过长，可能会使可推荐候选逐渐减少，因此需要根据内容更新速度以及用户的消费习惯合理设置用户曝光记录的保留周期。

17.4.4 缓存失效处理

对于前文所述的三类缓存，通常需要根据其用途分别设计失效处理逻辑。与服务端直接缓存数据库查询结果的常见业务场景不同，这些缓存在未命中时，并不一定会直接访问数据库。例如，推荐结果缓存未命中通常意味着需要重新执行推荐链路；**Browse Set** 服务中没有对应的曝光记录，可能表示用户在当前保留周期内没有已记录的曝光物品；首刷缓存未命中则可以回退到在线推荐流程。因此，本节更需要关注的是缓存未命中或集中失效后引发的计算压力，而不是笼统地将其归结为缓存穿透。

其中一种需要重点防范的情况是大量缓存同时过期，导致大量请求集中进入完整推荐链路，从而增加召回、特征抽取以及模型推理服务的负载。对此，可以为不同用户的缓存 **TTL** 增加随机抖动，分散过期时间；也可以采用后台刷新、适当提前更新等机制，减少缓存同时失效的情况。同时，还应该限制缓存重建时的并发量，并保留必要的降级和热门物品兜底能力，避免在线计算压力进一步放大。对于推荐结果缓存而言，还需要考虑候选列表耗尽、物品下架和内容状态变化等情况，确保缓存中的物品在返回时仍然满足基本的可见性与业务规则。

推荐结果缓存与首刷缓存还涉及模型版本与实验配置的一致性问题。缓存中的物品通常是在某个模型版本和特定业务配置下生成的，如果后续模型或特征逻辑发生了变化，旧缓存中的候选顺序可能无法反映新版本的推荐结果。因此，缓存可以记录生成时的模型版本、场景标识和必要的 **A/B** 实验信息，避免不同实验组之间不恰当地复用缓存结果。对于主要用于降级的推荐结果缓存，短时间内继续使用上一次成功请求生成的结果通常是可以接受的，而不必在每次模型更新后都立即清空缓存。如果模型更新涉及特征格式变更、内容安全规则调整或者其他不兼容的变化，则应该主动使相关缓存失效。

如果确实需要批量清除缓存，应该评估由此产生的连锁影响。因为缓存批量失效可能导致缓存命中率骤降，从而使得更多的请求同时进入完整的推荐链路。在必要的时候，也应该结合业务低峰、服务容量以及缓存重建进度来安排缓存失效的操作，并在清理前确认系统具备足够的在线实时计算能力。对于只涉及到小幅参数变化且缓存有效期较短的情况，则通常可以让旧缓存随 **TTL** 自然淘汰，以避免无必要的计算压力。

表 17.3: 推荐系统在线服务三类缓存对比。

缓存类型	存储内容	典型 TTL	主要用途	主要风险
推荐结果缓存	用户的一组 Top-K 推荐物品	10s~5 min	降级推荐、复用已有候选	候选耗尽、结果过期、模型版本不一致
Preload 缓存	为 Session 首刷预生成的推荐候选	5~30 min	缩短首刷等待时间	缓存未命中、结果陈旧、集中失效
Browse Set 缓存	用户近期已曝光的物品 ID 集合	1~3 天	过滤历史曝光内容	记录过期、Bloom Filter 假阳性、过滤过度

表 17.3 对比了三类缓存的存储内容、典型 **TTL**、主要用途和需要关注的风险。其中，推荐结果缓存侧重在正常链路异常时复用已有推荐结果，**Preload** 缓存重点降低 **Session** 首刷的等待时间，**Browse Set** 缓存则负责过滤历史曝光物品，避免重复推荐。三者虽然都依赖在线存储，但在更新时机、失效策略和正确性要求上并不相同，

需要结合各自的业务目标分别进行设计。

17.5 索引在线服务

在工业级推荐系统中，系统在每次用户请求时都需要频繁查询物品的属性、标签以及状态等信息，并根据用户兴趣和业务条件快速找到合适的候选物品。为了避免每次请求都扫描全量物品或者访问多个底层存储，在线服务通常会通过索引来组织和管理这些数据。按照查询方式的不同，索引在线服务可以分为两类：一类是正排索引（Forward Index），根据物品 ID 查询对应的属性和特征；另一类是倒排索引（Inverted Index），根据标签、类目等条件查找符合要求的物品集合。两类索引的作用有所不同，其中正排索引主要为特征抽取和排序提供物品侧数据，而倒排索引则主要为规则召回和标签召回提供候选物品。它们都需要兼顾查询效率、存储成本、数据更新以及服务稳定性，但具体的实现方式存在明显差异。

17.5.1 正排索引在线服务

正排索引服务提供从物品 ID 到物品属性和特征的查询能力，是在线特征抽取以及业务策略生效的重要数据来源。一次推荐请求在经过召回和初步筛选之后，通常会得到一批候选物品。推荐链路服务会根据这些物品的 ID，批量获取类目、发布时间、内容理解标签、统计特征等信息，在链路中生效相关的业务策略。与此同时，推荐模型服务也会将这些物品侧的索引信息与用户特征和上下文特征拼接，作为模型在线推理的特征输入。因此，正排索引在线服务主要负责高效地完成候选物品侧的特征数据查询。该服务的数据通常由离线特征生产任务生成，再通过索引导入或者增量更新链路发布到在线存储系统中。

由于物品总量可能达到数亿甚至数十亿，单台机器通常难以同时容纳全部物品的特征数据。因此，正排索引一般需要采用分布式存储架构，将物品数据分散到多个节点，从而构建起分布式正排索引服务。对于以物品 ID 为键的 KV 查询，常见做法是对物品 ID 进行哈希分片，使不同物品的数据分布到不同节点。分片时既要考虑数据量是否均衡，也要考虑节点扩缩容时的数据迁移成本。如果直接采用取模分片，节点数量发生变化时，可能有大量物品需要重新映射；一致性哈希（Consistent Hashing）则可以在节点增减时减少需要迁移的数据量。实际系统还可以采用虚拟节点等机制改善负载均衡。不过，一致性哈希并不是唯一方案，具体选择需要结合分片数量、扩容频率和负载特征确定。

在存储形态方面，正排索引可以根据特征数据规模、访问频率以及延迟要求，采用内存 KV、持久化 KV 或者多层存储等方案。例如，Redis 或者自研内存哈希表可以用于低延迟查询；基于 RocksDB 等引擎构建的持久化 KV 存储，则可以利用本地 SSD 等介质承载更大规模的数据。对于访问频率差异较大的物品，也可以将热点数据放在内存中，将其余数据存储在本成本更低的持久化介质上。需要注意的是，不同存储方案的实际延迟取决于数据大小、批量查询方式、缓存命中率、并发负载和网络开销，不能简单地为某种存储设定固定的延迟数值。

在在线查询的时候，还应该尽量利用批量化读取的能力。例如，模型特征抽取服务需要查询数百个候选物品时，可以将多个物品 ID 合并为一次批量请求，而不是逐个物品来发起 RPC 请求。批量查询能够显著减少网络往返与调用开销，但批次也不能无限增大，否则可能增加单次请求的数据传输的带宽和处理耗时。因此，批量大小仍需结合整体延迟预算和服务自身的承载能力来进行调优。

正排索引还需要处理特征 Schema 演化问题。随着模型和特征逻辑不断迭代，物品特征可能新增字段、调整类型或废弃旧字段。例如，新版本增加了内容质量分，但全量物品特征尚未完成补算，此时新字段在部分物品上可能暂时缺失。如果模型特征抽取服务无法区分“字段尚未生成”和“字段的有效值恰好为默认值”，就可能对模型输入产生非预期影响。

因此，特征 Schema 更新通常需要配合版本管理、兼容性处理和历史数据补齐。在线服务可以通过 Schema 版本或字段有效性标记，明确哪些字段已经生成、哪些字段暂时缺失；模型特征抽取服务则根据字段的重要性，选择使用默认值、跳过相应特征，或者在必要时从其他数据源补取。对于新增字段，还应尽量保证新旧版本在过渡期间能够兼容，避免在历史物品尚未完成回填之前，大量缺失值直接影响排序结果。

总体而言，正排索引的核心任务是以较低的延迟，为一批候选物品提供完整、可用且版本明确的属性特征。

在工程实践中，除了优化查询性能，还需要通过合理的分片、存储分层、增量更新和版本切换机制，兼顾数据规模、特征新鲜度和在线服务的稳定性。

17.5.2 倒排索引在线服务

倒排索引服务的功能是根据标签、类目、作者或其他检索条件，快速找到符合要求的物品集合。它是规则召回以及标签召回的重要基础。以类目召回为例，推荐系统可以为每个类目 ID 维护一个 **Posting List**（倒排列表），其中记录属于该类目的物品 ID。当在线召回时，系统只需查询当前类目对应的倒排列表，就能够快速获得对应的候选物品，而不必扫描全量物品。

对于离散的标签和类目条件，**Posting List** 是一种常见的数据组织方式。对于发布时间、数值范围等条件，则还可以结合时间分桶、范围索引或其他数据结构实现检索，不必将所有类型的查询都处理成同一种倒排列表。随着物品规模增长，**Posting List** 的存储空间会成为倒排索引设计中的重要问题。以一个包含百万级物品 ID 的列表为例，如果每个 ID 都使用 64 位整数存储，仅 ID 本身就需要数 MB 的空间；如果列表规模进一步扩大，存储的开销也会随之增加。因此，很多倒排索引实现会先将物品 ID 按升序排列，再存储相邻 ID 之间的差值，而不是直接保存每个 ID 的绝对值。由于排序后的相邻 ID 往往具有更小的差值，这种差值编码（**Delta Encoding**）能够减少需要存储的数据量；进一步结合 **VByte**、**BitPacking** 等变长或定长位压缩方法，还可以继续压缩列表。这里的压缩比例取决于物品 ID 的分布、列表密度和具体的编码方式，不能保证所有列表都达到相同的压缩效果。压缩也不仅影响存储空间，还会影响查询时的解码成本。因此，工业系统通常需要在内存占用、查询吞吐量和解码延迟之间进行权衡，并针对高频访问的列表优化读取和遍历过程。

除了压缩存储，倒排索引还需要支持物品上线、下架、标签变更等动态更新。例如，当某个视频被下架时，系统需要尽快避免它继续通过标签召回进入候选集；当物品新增标签时，也需要及时将其加入相应的倒排列表。对于包含大量物品的高频 **Posting List**，如果每次更新都直接修改整条列表，不仅会产生较高的写入开销，还可能与并发查询发生冲突。因此，实际实现不一定直接修改已有的超长 **Posting List**，而可以将批量构建的基础索引与增量更新的数据分开进行维护。物品变更通过消息队列或其他更新链路写入增量结构，同时使用删除标记或状态信息屏蔽已经下架的物品；而查询时结合基础索引和增量数据，得到当前有效的候选集合。后台再定期合并增量数据、清理过期记录并重建索引段。对于规模较小、更新频率较低的列表，也可以采用更直接的增删方式。具体实现取决于更新频率、查询并发量和列表规模，关键是避免频繁写操作影响在线查询。倒排索引的更新细节我们在前面第 16 章《推荐系统离线服务设计》中也进行过介绍，这里不再继续赘述。

在查询阶段，倒排索引在线服务通常需要支持多条件组合。**AND** 查询用于查找同时满足多个条件的物品，对应多个 **Posting List** 的交集；**OR** 查询用于查找满足任意一个条件的物品，对应列表的并集；**NOT** 查询则用于排除不满足要求的物品，通常可以从已有候选集合或允许检索的物品集合中减去相应列表。**AND** 查询的效率与参与求交的列表长度密切相关。假设一个类目列表包含数百万个物品，而某个作者的列表只有几千个物品。如果直接从头扫描两个列表，就会产生大量不必要的比较。更合理的做法是优先遍历较短的列表，并在较长列表中定位对应的物品 ID。对于长度差异较大的有序列表，可以利用跳跃指针或 **Galloping Search**（先指数扩展搜索范围，再进行二分定位）跳过不可能匹配的区间，减少比较次数。对于多个条件的 **AND** 查询，通常也会先处理较短的列表，尽早缩小候选集合，再与其他列表求交。

在推荐场景中，倒排索引在线服务还需要处理物品是否允许被推荐的问题。例如，物品可能因下架、审核未通过或业务规则限制而不能出现在当前推荐结果中。一种常见做法是维护物品状态索引，查询得到候选物品后，先过滤掉不可推荐的物品，再将剩余结果交给后续排序。部分系统会使用**状态位图（Status Bitmap）**加速这一过滤过程。它通过一个或多个比特记录物品的状态，例如物品是否可用、是否已下架。若系统为物品分配了连续或较稠密的内部 ID，就可以用一个比特表示一个物品的状态，减少存储开销，并利用批量位运算加速过滤。以十亿个物品为例，单个位图的原始存储空间约为 125 MB；实际部署时还需要考虑其他状态、分片、冗余和元数据的开销。如果物品 ID 本身非常稀疏，则通常需要先通过内部 ID 映射等方式建立合适的索引，不能直接假设任意 64 位物品 ID 都可以紧凑地映射到位图。需要注意的是，状态位图也不是所有系统都必须采用的方案。对于规模较小或状态查询本身开销不高的场景，KV 查询、状态集合或其他过滤结构同样可能满足要求。无

论采用哪种方式，都需要让物品状态变化及时传播到在线索引，并监控更新延迟，避免已经下架或不可见的物品继续被推荐。具体能够达到毫秒级还是秒级生效，取决于事件处理、存储更新和索引刷新机制。

当物品规模不断增长时，倒排索引在线服务还会面临扩容和在线重建的问题。与正排索引按物品 ID 查询单条记录不同，倒排索引需要维护不同条件对应的物品集合。具体的分片方式可以按照标签、列表、物品 ID 范围或索引段等维度设计，并非所有系统都以整条 Posting List 作为迁移单位。对于包含大量物品的热门类目，如果所有查询都集中访问同一份列表，就可能产生访问热点；必要时可以进一步拆分列表或分散查询负载。在扩容过程中，系统需要将相关索引数据迁移到新节点，同时处理迁移期间持续到达的物品变更事件。一种常见思路是先复制已有数据，再回放迁移期间累积的变更日志；也可以在过渡阶段将更新同时写入源节点和目标节点。待目标节点的数据追平并完成一致性校验后，再将读流量切换到新节点，最后停止旧节点上的读写。无论采用哪种方式，都需要明确迁移进度、数据版本和流量切换条件，避免出现数据遗漏或重复处理。

对于全量索引重建，可以采用新旧索引并行准备、验证通过后再切换的方式，尽量避免重建过程影响在线查询。具体而言，服务先在后台构建一份新的倒排索引，完成数据完整性检查等操作后，再将在线查询切换到新索引。在切换之后，旧索引可以暂时保留一段时间；如果发现新索引存在数据缺失或查询异常，就可以将请求切回旧索引。倒排索引的全量重建通常由标签体系变化、索引逻辑调整或数据修复等原因触发。如果变化的是物品 Embedding，并需要重新构建 ANN 索引，则属于向量索引更新问题，与基于标签和属性条件的传统倒排索引有所不同。需要注意的是，新旧索引同时存在会增加更新期间的存储开销，因此需要根据索引规模和服务容量合理规划资源。

表 17.4 对正排索引与倒排索引的核心差异进行了总结。从表中可以看到，正排索引的主要任务是根据物品 ID 高效读取属性特征，因此重点在于 KV 查询、数据分片和 Schema 兼容；倒排索引则需要根据检索条件快速筛选物品集合，更关注列表压缩、条件求交、动态更新和热点负载治理。两者都需要支持增量更新与全量版本切换，但应根据各自的数据结构和查询模式设计相应的工程方案。

表 17.4: 正排索引与倒排索引在线服务核心特性对比。

维度	正排索引	倒排索引
查询方向	物品 ID → 属性和特征（点查或批量查）	标签、类目或检索条件 → 物品 ID 集合
主要用途	特征抽取、排序阶段的物品特征拼装	规则召回、标签召回和候选过滤
典型存储结构	分布式 KV，可结合内存和持久化存储	Posting List、压缩编码及增量更新结构
数据更新	物品属性增量更新；全量特征刷新后按版本切换	物品标签和状态增量更新；定期合并或重建索引
扩容与迁移	按物品 ID 分片，并迁移受影响的数据	根据分片策略迁移列表、列表片段或索引段，并追平增量变更
全量重建	构建新版本并校验后切换，保留旧版本用于回滚	后台构建新索引，校验后切换在线读取版本
核心工程挑战	批量查询延迟、数据分片、特征 Schema 演化	列表压缩与求交、高热列表、动态更新和数据一致性

17.6 模型在线服务

模型在线服务负责将训练产出的模型参数和特征表示应用于实时推荐请求。一次推荐请求通常需要读取用户与物品特征、查询 Embedding、完成特征拼装，再调用推理服务计算点击、观看时长和互动等目标的预估结果，供后续的多目标融合与排序模块使用。与模型离线训练相比，模型在线服务需要在严格的延迟预算内处理持续到达的用户请求，同时满足高并发、稳定性和模型快速迭代等要求。因此，工程设计不仅要关注模型本身的推理速度，还要处理数据访问、模型发布、版本一致性和异常兜底等问题。本节按照在线计算的主要流程，依次介绍 Embedding Server、特征抽取服务、TensorRT 推理服务和通用推理优化技术，再分别讨论双塔模型与 DNN 精排模型的在线、离线协同方式，最后介绍训练与在线服务之间的特征一致性问题。

模型版本通常由训练与发布系统统一管理。在模型训练完成后，系统将模型权重、Embedding 参数及必要的配置保存为带有版本标识的产物，并记录对应的训练任务、数据窗口和关键配置。模型文件可以保存在 HDFS 或其他模型存储中。当线上发布时，发布系统根据指定版本分发或加载相应文件或通过实时消息队列读取模型参数，并通知 Embedding Server、Infer Server 等服务完成更新。对于规模较大的模型或 Embedding 表，通常通过文件分发、快照加载或分片加载传输数据，而不是直接把整份模型参数放入消息队列；消息队列更适合传递版本变更通知、模型参数的增量更新或其他控制消息。模型上线仍需遵循灰度发布、逐步扩量以及异常回滚的流程，并检查相互依赖的模型权重、Embedding 表和特征 Schema 是否兼容，避免不同服务各自更新后出现版本不一致的问题。

17.6.1 Embedding Server 服务

Embedding Server 的主要职责是根据稀疏 ID（例如用户 ID、物品 ID、作者 ID 或类目 ID）查询对应的稠密向量（Dense Embedding）。在工业级推荐系统中，Embedding 表可能包含数亿甚至数十亿条记录，每条记录又有几十到几百个维度。以一亿条、每条 64 维的 FP32 向量为例，仅向量数据就需要约 25.6 GB 的存储空间，尚未计入索引结构和运行时开销。与此同时，在线服务还需要承受较高的查询 QPS，因此 Embedding Server 的设计需要综合考虑查询延迟、吞吐量、存储成本和数据更新效率。

Embedding Server 通常使用 C++ 等高性能语言实现，并通过哈希表或其他 KV 索引，根据 ID 快速定位对应的 Embedding。对于哈希表，单次查找的平均时间复杂度通常为 $O(1)$ ，但实际性能还取决于数据布局、哈希冲突和内存访问效率等因素。当全量 Embedding 无法放入单机内存时，可以按照 ID 的分片规则将数据分布到多个节点。查询时，系统先根据 ID 确定目标分片，再向对应节点发起请求并完成查找。分片方案可以采用普通哈希分片、一致性哈希或固定分区等方式，具体需要结合数据分布、节点扩缩容频率和数据迁移成本进行选择。

在存储效率方面，可以通过降低 Embedding 的数值精度来减少内存占用和数据传输开销。例如，将 FP32 转换为 FP16，可以使向量数据的存储量约减少一半；进一步采用 INT8 等低精度表示，还可以继续压缩存储空间。不过，量化会引入数值误差，其对模型效果的影响取决于向量分布、量化方法和下游模型结构。因此，在实际部署前，需要使用代表性的测试数据评估量化误差，并结合离线指标和线上 A/B 实验，判断量化后的效果是否满足业务要求。

Embedding 表是否与模型权重一起部署，需要综合考虑数据规模、加载成本和版本管理的复杂度。对于规模较小的 Embedding 表，可以将其与模型权重一同加载到推理服务中，无需单独部署 Embedding Server，从而简化服务架构和版本管理。但当 Embedding 表达到数十 GB 甚至更大时，每次模型更新都重新加载整张表可能带来较高的内存、网络和更新时间开销。此时，可以将 Embedding 表与推理模型分开存储和部署，由独立的 Embedding Server 提供向量查询服务，而 Infer Server 负责模型推理。这种方式有利于独立管理和更新大规模 Embedding 数据，但也要求发布系统明确维护模型权重与 Embedding 表之间的版本对应关系，确保两者能够兼容使用。

对于访问频率特别高的 ID，例如头部作者或热门物品 ID，还可以在特征抽取服务中维护本地内存缓存，将少量高频 Embedding 缓存在调用方。这样，在缓存命中时无需再次通过 RPC 查询 Embedding Server，可以减少网络通信和远程服务调用的开销。需要注意的是，本地缓存应控制在合理的内存规模内，并在 Embedding 版本更新时及时更新或失效相关缓存，避免不同请求继续使用过期的向量。对于访问频率较低、缓存命中率不高的 ID，则仍可以直接查询 Embedding Server，以免缓存机制本身带来额外的内存开销和维护成本。

17.6.1.1 模型权重与 Embedding 版本对齐

对于通过同一次训练联合学习的 ID Embedding 表和后续模型参数，两者是在同一训练目标下协同优化得到的。Embedding 不仅是独立的数值向量，其含义也与训练时形成的向量空间有关。如果线上使用了新版本的模型权重，却仍查询旧版本的 Embedding，模型输入就可能偏离训练时的预期，进而造成预估分值变化或排序质量下降。版本不一致并不一定会导致 NaN 或服务报错，但可能在没有明显技术告警的情况下影响推荐效果。

在工程上，我们可以为同一次训练产出的模型权重和相关 Embedding 表记录统一的发布版本或关联标识。在更新时先将新版本数据加载到目标服务，并检查各分片是否完整、是否能够正常查询；推理服务也需要加载对

应的模型权重并通过基本校验。只有相关服务都准备就绪后，才切换新版本的流量。回滚时同样需要恢复彼此兼容的模型权重和 **Embedding** 数据，而不是只回退其中一部分数据。对于由独立模型生成、与主模型分开训练的向量，则应根据实际依赖关系定义兼容规则，而不应该机械地要求所有向量使用同一个版本号。

版本校验应当作为发布流程的一部分，分值分布、**Embedding** 缺失率以及推理错误率等监控指标则可以用于发现更新后的异常。如果新版本上线后出现无法解释的分值偏移或缺失率上升，应该暂停继续扩量，并检查模型、**Embedding**、特征定义及缓存是否使用了兼容版本。

17.6.1.2 冷启 ID 的兜底处理

推荐系统中的物品不断更新，新物品的 **ID** 可能尚未出现在训练得到的 **Embedding** 表中，因此在线查询时便会发生未命中（Miss）的情况。如果没有明确的处理策略，推荐系统可能出现特征缺失、推理异常，或者新物品无法获得合理排序分值等问题。因此，**Embedding Server** 服务或其上游特征抽取服务需要为这些冷启 **ID** 定义明确的兜底规则。

一种最简单的方式是为未命中的 **ID** 返回默认向量，例如零向量或专门的 **OOV**（Out-of-Vocabulary，词表外）向量。零向量容易实现，但无法表达新物品自身的特征；在依赖 **ID Embedding** 的模型中，这部分输入会缺少区分能力。不过，零向量并不意味着物品一定会被排到末位，因为最终分值还会受到其他特征和模型结构的影响。

如果新物品具有类目或内容标签，也可以尝试使用同类目已有物品 **Embedding** 的均值来作为初始表示。这种类目均值兜底的方式能够利用已有物品的统计信息，但均值向量未必能准确表达新物品的语义，实际效果仍然需要通过线上 **A/B** 实验进行验证。如果系统已有基于标题、标签、图像或视频内容的 **Item Tower**，也可以利用这些内容特征生成初始 **Embedding**，通常更有机会表达新物品自身的属性。

随着新物品在推荐系统中不断积累行为数据，并在后续训练或近线计算中获得正式的 **Embedding** 向量，系统就可以将初始兜底表示替换为新的向量。此外，在 **Embedding Server** 服务的具体实现时还应该区分 **ID** 不存在、**Embedding** 尚未加载以及存储查询失败等情况，避免把服务故障一概当作冷启 **ID** 来处理。对于无法安全兜底的情况，可以根据业务策略跳过该候选，或使用其他可用特征完成后续处理。

17.6.1.3 批量查询优化与热更新

在粗排和精排阶段，一次请求可能需要查询数百个候选物品的 **Embedding**。如果每个 **ID** 都单独发起 **RPC**，就会产生大量的网络请求，并增加请求调度、序列化以及反序列化的开销。即使本地查找很快，逐个远程查询仍可能显著拉长整体特征抽取时间。

因此，**Embedding Server** 服务通常会提供**批量多键查询**（Multi-key BatchLookup）的接口。调用方将同一次请求所需的多个 **ID** 合并为一个批量请求，**Embedding Server** 服务集中完成查找后再一次性返回结果。如果 **Embedding** 数据分布在多个节点，可以先按照分片规则将 **ID** 分组，并行发往不同节点，再汇总各节点的返回结果。批量查询可以减少网络请求和单次调用的固定开销，但批次并不是越大越好。如果批量过大可能增加数据传输量、内存占用和单次响应时间，因此在实际应用中需要结合请求规模和延迟预算来进行调优。对于同一批请求中重复出现的 **ID**，还可以先去重，避免重复读取和传输相同向量。

Embedding 表的热更新则需要避免新旧数据混用。如果服务在更新过程中直接覆盖正在使用的表，请求就可能读到部分新向量和部分旧向量，使同一次推理所使用的数据版本不一致。一个常见的方案是采用**双缓冲**（Double Buffer）：即服务保留当前生效的版本，同时将新版本写入备用的空间；新版本写入完成并通过记录数量、分片完整性和抽样查询等校验后，再切换在线读取的版本。旧版本可以暂时保留，等到使用旧版本的在途请求结束后再进行回收，以便必要时回滚。

需要注意的是，双缓冲并不是所有 **Embedding Server** 都必须采用的固定方案。它的优点是版本切换清晰、回滚方便，缺点是在更新期间需要额外的存储空间；对于数十 **GB** 乃至更大的 **Embedding** 表，双份 **Embedding** 表在内存中的存储成本可能很高。当内存受限时，可以考虑版本化文件、分片加载或其他逐步切换方案，但必须明确请求在一次推理期间使用哪个版本，并防止同一个请求混用不兼容的数据。

17.6.2 特征抽取服务

特征抽取负责将来自不同上游服务的原始特征转换为模型可以直接使用的输入。服务根据请求中的用户、物品和上下文标识，从用户信息、物品索引和统计特征等上游服务获取所需数据，再按照模型的特征定义进行处理。对于需要通过 **Embedding** 表示的离散特征，特征处理逻辑会将原始特征转换为对应的 **slot** 和 **sign**。其中，**slot** 标识特征所属的槽位，**sign** 通常是由特征值计算得到的哈希标识。随后，服务根据 **slot** 和 **sign** 向 **Embedding Server** 查询对应的向量，取回的 **Embedding** 通常采用 **FP32** 浮点数表示。对于数值特征等不需要 **Embedding** 查询的输入，则按照模型要求完成类型转换、归一化等处理。最后，服务将这些特征组织成模型所需的输入格式，交给 **Infer Server** 进行推理。

特征抽取的处理规则需要与训练阶段保持一致。例如，离散特征的编码方式、**slot** 和 **sign** 的生成规则，以及数值特征的归一化方式，都应遵循训练时采用的定义，否则线上模型接收到的输入可能与训练阶段不一致，进而影响预测效果。这里特征抽取的方式以及抽取的特征个数、类型都需要在实际代码编写中进行仔细的检查，在实际工业级服务部署落地过程中，这里很容易出现隐蔽的错误，导致最终线上模型推理的特征抽取逻辑与离线训练存在差异，进而影响模型的 **A/B** 实验效果。

当然，特征抽取服务也不一定需要独立部署。如果特征处理主要涉及轻量级 **CPU** 计算，可以将相关逻辑直接集成到模型在线推理的 **Infer Server** 服务中，减少特征抽取服务与推理服务之间的 **RPC** 通信开销，并简化服务部署和维护。此时，**Infer Server** 服务可以在推理过程中完成特征处理，并根据需要进行调用上游特征服务与 **Embedding Server**。另一方面，如果特征处理逻辑较复杂、需要被多个模型或业务链路复用，或者需要独立扩缩容和更新，也可以将其拆分为单独的服务。因此，是否独立部署特征抽取服务，应该根据特征处理的计算开销、复用需求和整体服务架构决定，而不是必须采用的设计。

17.6.3 TensorRT 在线推理服务

Infer Server（模型推理服务）负责加载训练完成的模型，接收特征抽取阶段组织好的输入，执行前向计算，并返回模型预测结果。在工业级推荐系统中，模型通常需要预测点击率、转化率、观看时长等多个不同的目标，再由后续的多目标融合排序模块来使用这些预测结果。根据模型的计算特点和部署资源，**Infer Server** 可以采用 **CPU** 或 **GPU** 执行推理。当模型需要较大的计算吞吐量时，可以使用 **NVIDIA** 提供的 **TensorRT** 推理引擎，针对 **GPU** 硬件进一步优化模型的执行过程。

CPU 和 **GPU** 在计算能力和适用场景上存在差异。**CPU** 擅长通用计算和复杂的控制逻辑，对于计算量较小、输入规模有限或不适合大规模并行处理的模型，采用 **CPU** 推理可能更简单、更经济。**GPU** 则具有更强的大规模并行计算能力，特别适合矩阵乘法等深度学习模型中的密集计算。当模型计算量较大、能够充分利用 **GPU** 的并行能力时，**GPU** 通常可以提供更高的吞吐量。不过，使用 **GPU** 并不意味着推理一定更快：数据传输、任务调度和内核启动都可能带来额外开销。如果模型本身计算量较小，或者请求规模不足以发挥 **GPU** 的并行能力，这些开销就可能抵消部分性能收益。因此，**CPU** 与 **GPU** 的选择需要综合考虑模型计算量、请求并发、批处理规模、延迟要求和硬件成本。

在选用 **GPU** 之后，还可以进一步优化模型的执行方式。**TensorRT** 就是为此设计的推理优化引擎。它会分析模型的计算图，并结合目标 **GPU** 和输入形状，选择合适的算子实现、计算精度和内存使用方式，将优化后的模型构建为可供推理服务加载的 **Engine**。与直接逐个执行原始计算图中的操作相比，这种方式可以减少不必要的计算和数据读写，并更充分地利用 **GPU** 的计算能力。

其中一种重要的优化方式是算子融合。当计算图中相邻算子之间存在可融合的计算时，**TensorRT** 可以将部分操作合并执行。例如，矩阵乘法、偏置加法和激活函数等操作，在满足融合条件时可以组合执行，从而减少中间结果写回显存、重复读取和 **GPU Kernel** 启动的开销。对于 **Attention** 等较复杂的计算结构，能否融合以及具体融合范围，则取决于模型计算图、**TensorRT** 版本、**GPU** 架构和输入形状。此外，**TensorRT** 还会为不同计算层选择适合目标 **GPU** 的 **Kernel** 实现，并通过内存复用等方式控制中间结果的存储开销。这些优化共同决定了最终 **Engine** 的执行效率，实际收益需要结合具体模型进行测试。

除了优化计算过程，**TensorRT** 还可以通过调整数值精度进一步提高推理效率。**FP32** 通常具有较高的数值精

度，但会占用更多存储带宽和计算资源。在硬件支持的情况下，FP16可以减少数据占用，并提高部分计算的吞吐量；INT8等量化方式还可以进一步降低数据存储和计算成本，但通常需要结合校准数据或量化感知训练等方法控制误差。不同模型对精度变化的敏感程度不同。对于推荐排序模型，即使预测分值只有较小变化，也可能改变部分候选物品的相对顺序。因此，在采用低精度推理前，应使用固定样本对比不同精度下的预测结果，并结合离线指标和线上 A/B 实验验证，不能直接假设精度损失不会影响业务效果。

要使用 TensorRT 优化训练框架中的模型，通常还需要经过模型转换和 Engine 构建。一个常见流程是将 PyTorch 或 TensorFlow 模型导出为 ONNX，再由 TensorRT 解析计算图并构建 Engine。对于 TensorFlow 模型，也可以采用 TF-TRT 集成方式，将计算图中受支持的子图转换为 TensorRT Engine，其余操作继续由 TensorFlow 执行。但转换过程并不总是顺利的，不同框架的算子定义，以及 TensorRT 对具体算子的支持范围，都可能存在差异。如果模型使用了不受支持的算子，或者转换前后的计算行为不一致，就需要根据具体情况调整模型结构、改写计算图，或开发自定义插件。因此，在模型开发阶段就验证导出、Engine 构建和推理结果，可以减少后续部署中的兼容性问题。

模型输入形状也是构建和部署 Engine 时需要考虑的因素。推荐请求中的候选物品数量、batch size 和用户行为序列长度可能随请求而变化，因此推理服务不一定总是接收到固定形状的输入。TensorRT 支持通过动态形状配置适应一定范围内的输入变化。在构建 Engine 时，可以为输入指定允许的最小形状、常用优化形状和最大形状；运行时则需要确保实际输入符合相应配置，并选择匹配的优化配置。由于常用优化形状会影响构建阶段的 Kernel 选择，因此应结合真实线上请求的形状分布设置相关参数，再通过压测评估不同输入规模下的延迟与吞吐量。

最后，构建完成的 TensorRT Engine 还需要与线上运行环境保持兼容。其兼容范围与构建时使用的 TensorRT 版本、GPU 架构及相关配置有关，不能默认同一个 Engine 可以在任意 GPU 和任意运行时版本中直接使用。对于部署多种 GPU 型号的推荐系统，需要提前规划 Engine 的构建与分发方式。升级 TensorRT、调整模型结构或更换硬件后，应使用固定输入样本比较新旧版本的输出误差，并重新测试延迟、吞吐量和资源占用，确认满足业务要求后再逐步切换线上服务。

17.6.4 通用推理优化技术

上一节介绍了以 TensorRT 为代表的 GPU 推理引擎如何通过计算图优化、算子融合、精度调整和内存复用等手段，提高模型在目标硬件上的执行效率。这些优化主要作用于模型的计算图及其执行过程，但在线推理服务的整体性能并不只取决于单次模型计算的速度。请求如何组织、相同计算能否复用、模型本身是否存在进一步压缩的空间，以及多个推理任务如何共享 GPU 资源，都会影响服务的吞吐量、响应延迟和计算成本。

因此，在推理引擎优化之外，还可以从模型计算、请求调度和硬件资源利用等方面进一步提升推理效率。例如，动态批处理通过合并多个并发请求提高 GPU 利用率；FlashAttention 针对注意力计算中的显存访问进行优化；量化和知识蒸馏分别从数值精度与模型规模入手降低计算成本；中间结果复用则可以避免对相同输入进行重复计算。此外，多模型混部能够提高 GPU 资源的整体利用率，但也需要处理不同推理任务之间的资源竞争。下面本节分别介绍这些常见的方法以及其适用的条件。

17.6.4.1 Dynamic Batching

Dynamic Batching（动态批处理）是提高推理吞吐量的一种常见手段。即使每个推荐请求内部已经包含一批候选物品，低并发或小批量请求仍可能无法充分利用 GPU。推理服务可以在很短的等待窗口内收集多个并发请求，将它们合并为更大的 Batch，再统一提交给 GPU。这样可以摊薄部分调度和计算开销，提高硬件利用率。

动态批处理通常会引入一定的排队等待，因此等待窗口不能只按 GPU 利用率来决定，还需要兼顾端到端延迟目标。当请求量较高时，短时间内容易积累到足够的请求，动态批处理可能提升吞吐量；而当请求量较低时，等待窗口反而可能增加不必要的延迟。因此，在实际部署中应该结合请求到达速率、batch size 分布和 P95/P99 延迟进行调优。只有输入形状、请求状态和推理服务实现允许合并时，相关请求才能放入同一个 batch。

17.6.4.2 算子融合、图优化与 FlashAttention

算子融合和计算图优化可以减少不必要的中间张量、内存访问和 Kernel 启动开销。除了 TensorRT 引擎自动完成的优化之外，模型实现本身也应避免重复计算、冗余张量转换和不必要的拷贝。优化效果需要结合实际计算图和 GPU 性能分析结果判断，因为某些算子可能受计算能力限制，另一些则主要受内存带宽限制。

FlashAttention 是针对 Transformer Attention 的一种 IO 感知算法。标准的 Attention 算法通常需要计算查询与键之间的相关性，并根据相关性对值向量加权。如果直接计算完整的 $N \times N$ 注意力矩阵，相关中间结果的存储需求会随序列长度呈现平方级增长。**FlashAttention** 算法通过分块计算，避免将完整的注意力矩阵反复写入和读取 GPU 高带宽显存，从而减少显存访问开销。在长序列场景中，这种方法有机会降低内存占用并提高速度，但具体收益取决于序列长度、head 数、数据精度、硬件及实现是否支持相应的 Attention 结构。此外，需要强调的是，FlashAttention 主要优化的是 Attention 的内存访问，但并不会消除标准全注意力计算本身随序列长度增长的主要计算成本。

17.6.4.3 量化与知识蒸馏

量化通过降低模型权重或中间激活值的数值精度，减少存储、内存访问和部分计算开销。后训练量化（Post-Training Quantization, PTQ）可以在模型训练结束之后来执行，通常不需要重新进行完整训练。而量化感知训练（Quantization-Aware Training, QAT）则在训练过程中模拟量化误差，使模型参数适应低精度计算，但需要额外的训练过程。不同量化方案对精度和性能的影响并不相同，需要结合目标硬件和模型结构进行验证。

知识蒸馏则通过训练一个规模较小的学生模型，使其学习较大教师模型的预测结果或中间表示，从而降低在线推理成本。与量化方法相比，蒸馏改变的是模型结构或参数规模，而不只是数值表示。蒸馏后的学生模型可以减少层数、隐藏维度或计算分支，但也可能损失部分预测能力，并需要额外的数据和训练资源。知识蒸馏的实际收益取决于教师模型、学生模型、蒸馏目标和业务指标，不能仅凭层数缩减比例推断延迟或者 AUC 的变化。在工程上应该以目标服务中的实际吞吐量、尾延迟和线上业务指标效果来决定是否采用知识蒸馏的方法。

17.6.4.4 多模型混部

多模型混部是提高 GPU 资源利用率、降低推理成本的一种方式，即在同一台 GPU 服务器或同一块 GPU 上部署多个模型或推理任务。例如，召回相关模型、粗排模型与精排模型可以在资源允许的情况下共享部分硬件。混部能够利用不同任务在到达时间和计算负载上的差异，但也可能引入资源竞争，尤其是多个模型同时申请大量显存或计算资源的时候。

NVIDIA MPS (Multi-Process Service) 能够改善多个 CUDA 进程之间的 GPU 并发执行；MIG (Multi-Instance GPU) 则在支持该功能的 GPU 上将设备划分为具有独立资源配额的实例，提供更强的资源隔离。二者解决的问题不同，具体选择取决于硬件支持、隔离要求和负载特征。对延迟敏感的精排服务，应优先保障其资源与尾延迟要求；离线或近线批量计算任务则可以安排在资源较空闲的时段，或通过调度策略降低其优先级。在实际的部署中，混部前后通常都应该比较各模型的延迟分布和吞吐量，避免整体 GPU 利用率提高却损害关键在线服务的响应时间。

对于包含用户历史序列的 Transformer 或其他序列模型，还可以在模型结构允许时复用与候选物品无关的用户侧计算结果。例如，如果用户历史行为的 key、value 投影不依赖当前的候选物品，就可以在一次请求内只计算一次，并让多个候选物品共享这些中间结果的 Tensor，避免对相同历史序列反复执行相同的计算操作。不过，并非所有排序模型都能直接使用标准的 KV Cache。如果注意力结构或输入构造会使相关表示依赖候选物品，就需要进一步分析可复用的计算部分。

表 17.5 对几类常见的推理优化技术进行了归纳，概括其核心机制、主要收益以及适用时需要考虑的代价或约束。表中给出的是定性比较，并不代表不同方法之间存在固定的收益排序，具体效果仍应以目标模型、部署硬件和真实线上负载下的测试结果为准。

表 17.5: 通用推理优化技术对比。

技术手段	核心机制	主要收益	主要代价或约束
Dynamic Batching	在短暂等待窗口内合并并发推理请求	提高批量规模和 GPU 吞吐量	可能增加排队延迟，低 QPS 时收益有限
算子融合与图优化	减少中间张量、内存读写和 Kernel 启动	降低部分计算开销	取决于计算图及推理引擎的优化能力
FlashAttention	分块执行 Attention，减少高带宽显存访问	改善长序列场景的内存访问效率	依赖模型结构、硬件和实现支持
量化	使用 FP16、INT8 等低精度表示	减少存储和数据传输开销，可能提高吞吐量	需要评估量化误差及硬件支持
知识蒸馏	训练小模型拟合教师模型的行为	降低模型规模与推理成本	需要额外训练，可能损失部分效果
中间结果复用	复用一次请求内候选物品共享的用户侧计算	避免重复计算相同输入	需要模型结构支持，并占用额外缓存空间
多模型混部	多个推理任务共享或分配同一 GPU 的资源	提高资源利用率，降低单位推理成本	存在资源竞争，需要隔离和优先级调度

17.6.5 双塔模型的在线-离线整体框架

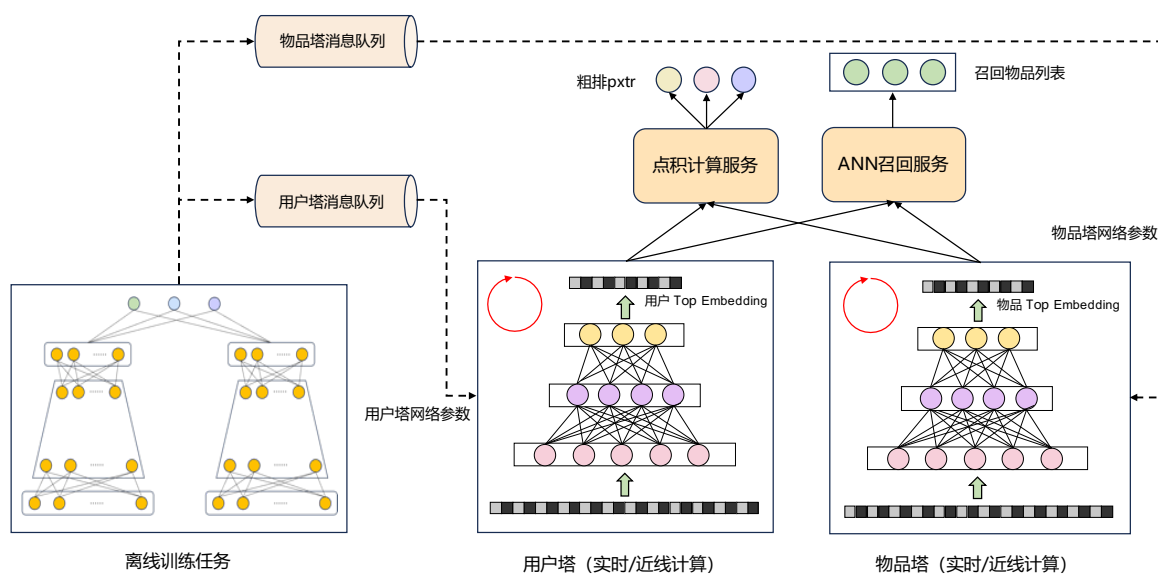


图 17.2: 双塔模型的离在线整体框架。

如图 17.2 所示，双塔模型的在线-离线框架包括离线训练、用户塔与物品塔计算，以及向量匹配服务三个部分。离线训练任务联合学习两座塔的参数，再通过各自的模型发布链路，将参数更新到对应的计算服务。用户塔根据用户及其上下文特征生成用户 Top Embedding，物品塔则根据物品特征生成物品 Top Embedding。这里的 Top Embedding 是指各塔前向计算得到的输出向量，不同于输入层根据稀疏 ID 查询的 Embedding。

用户塔和物品塔两侧输出向量可以用于不同的在线任务。在粗排中，系统读取已有候选物品的向量，通过点积等操作计算匹配分值；在召回中，系统将物品向量构建为 ANN 索引，再以用户向量为查询进行近似检索。因此，图中展示的是双塔模型在粗排和召回中的通用计算框架，而不局限于向量召回。

需要注意的是，图中的虚线表示模型参数的发布与更新，实线表示向量计算及其下游使用关系。两座塔虽然可以独立部署和更新，但其输出必须处于兼容的匹配空间中，因此模型发布系统需要维护两侧参数及相关向量产物的版本对应关系。

用户塔和物品塔的计算时机可以根据特征更新频率、计算成本和在线延迟要求分别确定。用户塔通常在请

求到达之后，结合用户行为和当前上下文实时生成用户向量。如果实时计算成本较高，也可以近线计算并缓存向量，但需要接受一定的更新延迟。物品塔的输入通常不依赖当前用户，因此更适合进行近线批量的计算。这时系统可以定期处理全量物品，也可以仅更新新增或发生变化的物品，再将生成的向量提供给在线服务。对于尚未完成预计算的新物品，也可以在算力和延迟预算允许时实时补算。需要注意，向量能否跨请求复用取决于其输入特征是否变化。如果物品塔引入了请求相关特征，就不能仅按物品 ID 预计算并长期复用同一个向量。

当双塔模型用于粗排时，双塔模型对多路召回合并后的候选集合进行打分。在线服务先获取当前用户向量，再按候选物品 ID 批量读取对应的物品向量，交由点积计算服务完成匹配。假设候选数量为 N ，向量维度为 d ，则批量点积的计算复杂度为 $O(Nd)$ ，适合通过矩阵运算高效执行。当点积计算服务产出打分结果之后，需要保证打分结果与候选物品 ID 相对应，然后将其送给后续的粗排多目标融合模块，最终筛选出粗排候选物品的集合送给精排模块。需要注意图 17.2 中的“粗排 pXTR”表示点击、互动、转化等目标的预估值。对于多目标双塔模型，可以针对不同目标设置相应的向量或打分分支，并根据具体设计将匹配分值转换为目标预估结果。因此，不能将未经处理的单个点积值直接解释为多个目标的概率。

当双塔模型用于召回时，模型需要从大规模物品集合中检索候选。系统先将物品塔生成的向量写入 ANN 索引，再以用户塔生成的向量作为查询，返回相近物品的 ID 及匹配分值，形成召回列表。物品塔到 ANN 服务的连线表示物品向量用于索引构建或更新，并不意味着每次请求都要实时计算全量物品向量。

粗排和召回对向量的使用方式虽然不同，但都依赖用户向量与物品向量之间的有效匹配。粗排直接计算已有候选的匹配分值，召回则通过 ANN 近似检索降低大规模向量搜索的计算成本。无论采用哪种方式，匹配所使用的相似度度量 and 向量归一化方式，都需要与模型训练时的匹配目标保持一致。两者虽然共享计算分离的基本架构，但实际使用的训练目标、模型参数和下游产物可能不同。

这种对向量匹配的共同依赖，也决定了双塔模型不能只管理模型参数，还需要协调相关向量与下游服务的版本。在新版本上线前，应该准备好对应的用户塔和物品塔参数，并确保在线使用的向量与新模型兼容。对于粗排而言，还需要更新相应的预计算物品向量；对于召回而言，则需要基于对应版本的物品向量构建或更新 ANN 索引，并完成查询验证。只有相关产物准备就绪且彼此兼容，才能将流量切换到新版本；回滚时也应恢复对应的一组版本。如果用户向量采用近线缓存，还需要记录其模型版本和生成时间，避免新模型读取不兼容或过期的向量。

在完成版本切换之后，还需要结合两类任务的实际计算路径检查服务质量。粗排应该重点关注向量缺失率、批量打分延迟和分值分布。召回则还应该关注索引更新延迟、有效候选数量以及候选物品的覆盖情况。通过将模型与向量的版本管理、发布验证与在线监控纳入同一条工程链路，才能在复用两侧计算结果的同时，稳定支撑双塔模型的召回与粗排任务。

17.6.6 DNN 精排模型的在线-离线整体框架

如图 17.3 所示，DNN 精排模型的在线-离线框架由左侧的离线训练任务和右侧的在线实时计算服务组成。离线训练以用户、物品和上下文特征及行为标签为输入，学习模型参数，再通过图中的 DNN 模型消息队列所代表的更新链路，将模型更新通知在线服务。在线侧接收当前请求对应的特征，执行 DNN 前向计算，得到图中的 Top Embedding，并由输出层生成各目标的精排预估值。这里的 Top Embedding 是模型对用户、候选物品及上下文进行联合建模后形成的高层表示，通常随请求和候选物品变化，不同于可以按物品 ID 独立预计算和复用的双塔物品向量。

与图 17.2 中先分别计算用户和物品表示、再通过轻量匹配完成打分的方式不同，DNN 模型将用户与物品之间的交互计算纳入在线推理过程。这使得 DNN 模型能够利用更丰富的交叉特征、候选相关的注意力结构以及多目标网络，但也要求在线服务完成特征读取、输入组装与模型计算，而不能仅靠读取预计算向量并进行点积来完成打分。因此，图 17.3 展示的是模型参数更新与前向计算的基本关系。在实际部署中，还需要将特征抽取、Embedding Server 和 Infer Server 衔接为完整的在线模型推理服务链路。

下面我们沿着一次请求的处理顺序来介绍这条链路的主要环节，以及影响其计算成本和运行稳定性的关键因素。当用户请求进入精排模型服务后，首先需要将分散在不同数据源中的特征整理为模型输入。精排模型服

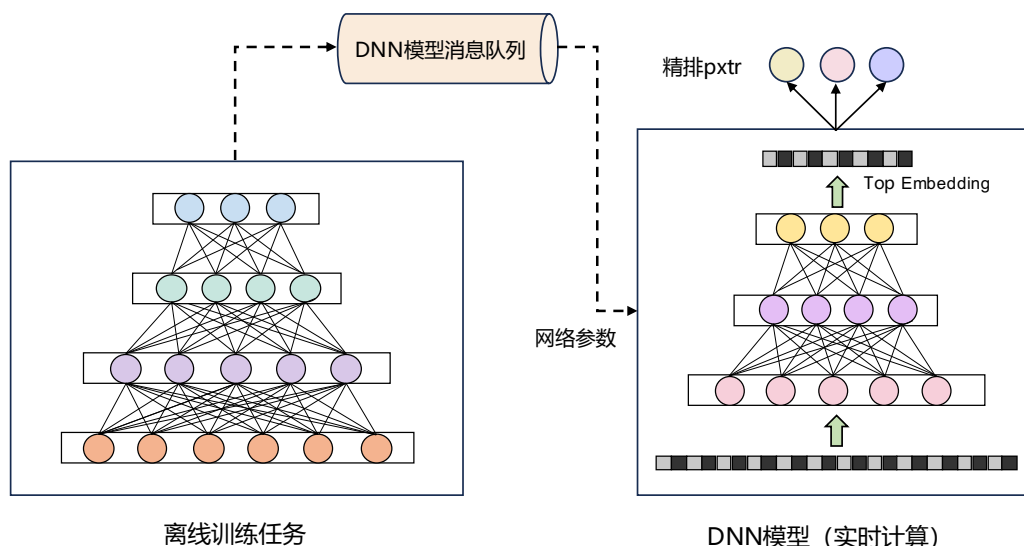


图 17.3: DNN 模型的离在线整体框架。

务接收上游筛选后的候选物品列表，并获取对应的用户信息、请求上下文等信息，同时确定本次请求使用的模型版本。随后，特征抽取环节从正排索引、统计特征服务等数据源获取所需信息，按照模型定义完成数值转换、离散特征编码和 Embedding 查询。对于同一次请求中各候选共享的用户与上下文特征，应当尽量只读取和处理一次。而物品侧特征则按候选 ID 批量获取，以减少逐个候选调用远程服务的开销。最后，系统将特征组织为推理引擎要求的输入张量，并保留候选物品与张量位置之间的对应关系，确保推理结果能够正确关联到各个候选物品上。

当输入张量准备完成后，Infer Server 便可以对候选集合执行批量推理。一次精排请求通常包含多个候选物品，服务可以将其组织为一个或多个 batch，在 CPU 或 GPU 上完成前向计算。模型中不依赖候选物品的用户侧计算，可以在网络结构允许时共享。而用户与物品之间的交叉计算，则通常需要针对各候选物品执行。在实际应用中，候选物品的数量以及用户行为序列的长度都会共同影响输入张量规模、显存占用与推理延迟，因此 batch size 需要结合实际请求负载以及推理引擎支持的输入形状来进行确定。对于候选物品较多的用户请求，可以采用分批次的形式处理，但也需要控制并行度，避免短时间内占用过多推理资源。

在完成前向计算之后，服务还需要将各候选的预估结果按约定返回排序链路。图中的“精排 pXTR”概括了点击、完播和互动等行为目标的预估输出，模型也可以输出观看时长等非概率指标。多目标联合建模能够让不同目标共享部分特征表示和网络计算，并在一次前向推理中返回多个目标的结果，减少重复计算。为了保证模型输出结果能够被下游正确地使用，在线接口需要明确各输出的名称、类型和含义，以及它们与候选物品之间的对应关系。如果采用有序列表或张量返回结果，还应通过固定的输出 Schema 约定各目标的位置，并对其进行版本管理，避免模型更新后下游读取错误的目标分值。

除了优化单次请求的执行成本，还需要做好模型更新与异常处理，保障整条服务链路稳定运行。在模型的离线训练完成之后，需要先对模型进行验证，并准备好对应的 TensorFlow 或 Pytorch 计算图和网络参数。当在线部署时，DNN 模型的 Infer Server 会根据配置加载相应的计算图，而训练得到的 DNN 参数则通过消息队列发送到在线服务，供模型线上更新使用。计算图的加载与参数更新是两条独立的链路，网络结构本身不通过消息队列传递。在新版本上线前，还需要使用固定样本验证线上推理结果，并确认输入特征的组织方式、模型参数版本以及各个输出值对应的目标和排列顺序均符合预期。当验证通过之后，新版本实例再经过加载、预热和健康检查，逐步接收灰度流量。如果在灰度中出现异常，则需要回滚到之前可用的服务版本。此外，只更新模型参数与改变网络结构也需要区别处理。前者可以按照现有的机制更新参数，而后者则需要同步调整 Infer Server 的计算图配置，并重新加载或部署相应的模型。

17.6.7 训练与在线服务一致性

Training-Serving Skew（训练与服务偏差）指模型训练时使用的特征，与模型在线推理时实际接收到的特征之间存在系统性差异。即使在线服务没有报错、推理速度也符合要求，这类差异仍可能影响预测分值和最终排序，因此是推荐系统中需要持续治理的问题。

这类偏差主要来自以下几个方面。第一，**特征计算逻辑不一致**。例如，离线训练使用 Python 或 Spark 计算用户年龄分桶，在线服务则使用 C++ 或 Java 实现；如果两套逻辑在边界处理、数值取整、缺失值处理或字符串编码等方面存在差异，同一条原始数据就可能得到不同的特征值。第二，**特征时效性不一致**。离线训练样本通常需要按照样本发生时刻构造特征，而在线服务读取的可能是当前最新状态。对于物品实时点击数、用户近期行为等变化较快的特征，如果训练与在线使用的时间口径不同，两者的分布便可能出现偏差。第三，**缺失值处理不一致**。在线取数失败时可能使用默认值或占位符，但训练数据中的同类缺失情况如果采用了不同处理方式，模型就会在训练和推理阶段看到不同的输入分布。

在实际工程中，要减少推荐模型线上线下一致的问题，首先需要统一特征定义、取值口径、更新时点和缺失值规则。对于稳定且经常复用的特征逻辑，可以尽量使用共享代码库或统一的特征处理组件。如果离线与在线必须采用不同语言或运行环境，则应该通过相同的特征 Schema、测试样例和边界用例进行交叉校验。例如，对同一批原始输入分别执行离线和在线特征计算，比较输出是否一致，并将这种对照测试纳入代码发布流程。共享实现有助于减少重复维护，但并非所有特征都必须通过同一个二进制库实现，关键是保证两侧遵守同一套定义和语义。

另一种方式是采用 **Uni-Feature（在线特征回流）**。在在线推理环节，应该将实际使用的特征或必要的特征信息随系统日志一同进行记录。在离线训练构造样本时，再关联并复用这些线上的特征，而不是从原始数据重新计算全部特征。这样可以显著减少因离线与在线计算逻辑不同而产生的偏差。不过，这种特征回流的方式会增加日志存储、数据传输以及样本关联的成本，并要求整个离线日志链路具备足够的可靠性。如果关键特征日志丢失、样本与特征关联错误或回流数据不完整，也可能带来新的训练数据问题。因此，需要根据特征规模和业务要求决定回流范围，并监控日志完整率、关联成功率和特征覆盖情况。

此外，线上还应持续比较特征的统计分布与缺失情况。可以按场景、模型版本和用户群体统计特征的均值、分位数、取值范围及缺失率，再与训练样本中的基准分布进行对照。当某一路特征的缺失率突然升高，或者在线分布持续偏离训练基准时，应该结合上游取数延迟、特征处理版本和数据更新时间进行排查。需要注意的是，在线流量和用户行为本身也会随时间变化，分布差异并不必然意味着实现错误；应结合业务场景和变化原因判断是否需要告警。

从工程管理的角度看，降低推荐模型离在线不一致问题的关键，是为每个特征维护清晰、一致且可追溯的定义。特征的计算逻辑、数据来源、时间口径、默认值和版本变化都应有明确记录，并通过自动化测试、在线分布监控和训练样本回流等机制持续验证。相比在问题出现后仅调整模型参数，提前保障训练与在线输入的一致性，往往更有利于稳定地发挥模型的效果。

17.7 本章小结

本章围绕推荐系统在线服务的请求链路、数据基础设施、模型推理和稳定性治理，系统梳理了离线产物如何在严格的延迟约束下转化为实时推荐结果。一次请求由业务服务端进入推荐 Router，依次经过多路召回、候选合并、粗排、精排和重排，再将结果返回客户端，并通过曝光日志回流形成在线服务与离线训练之间的数据闭环。面对由多个子服务构成的分布式调用链，系统需要统一管理端到端延迟预算，通过总请求、内部模块和单次 RPC 三层 Timeout 约束各阶段耗时，同时利用限流、熔断、降级和热门候选兜底抑制过载与故障扩散。

在线服务的稳定运行还依赖完整的发布、容量与可观测性体系。服务版本应经过单实例、部分实例到全量流量的逐级灰度，并结合技术指标与业务指标决定扩量或回滚；容量规划则需要以目标 SLA 下的单实例实测能力为依据，为流量波动、实例故障、发布过程和多机房容灾预留资源。监控体系不仅要覆盖各阶段的 P50、P95 和 P99 延迟，还要持续观察模型分值分布、特征质量和业务效果，并借助分布式链路追踪定位单次请求问题。告

警分级、版本核对、SLO 与错误预算进一步将异常响应和稳定性目标落实为可度量的治理机制。

用户侧服务为个性化召回与排序提供数据基础。User Info 服务统一组织用户基础属性、近期行为、实时状态、统计画像和离线挖掘特征；长期兴趣服务通过 SIM 在线检索、离线压缩表示或二者结合，从更长的历史序列中提取与当前任务相关的兴趣信号。近线服务则把适合预计算的 User Embedding、Item Embedding、统计特征和首刷候选移出实时请求，以一定的数据新鲜度换取更低的在线延迟与计算成本，其使用范围和更新周期需要根据算力约束与业务时效要求确定。

缓存与索引服务分别解决结果复用、状态过滤和物品数据访问问题。推荐结果缓存用于在短时间内复用候选或支持降级，Preload 缓存用于缩短首刷等待时间，Browse Set 则记录近期曝光并减少重复推荐；三者需要分别设计 TTL、失效回退和版本隔离策略，并防范集中失效带来的负载冲击。正排索引面向物品 ID 提供属性与特征的批量查询，重点处理分片、存储分层、增量更新和特征 Schema 演化；倒排索引根据标签、类目等条件检索候选集合，需要在 Posting List 压缩、多条件求交、状态过滤、热点治理、动态更新和在线重建之间进行权衡。

模型在线服务将模型权重、Embedding 和特征定义组织为可持续迭代的实时推理链路。Embedding Server 通过分片和批量查询支撑大规模稀疏 ID 的向量读取，并需要处理模型与 Embedding 的版本对齐、冷启 ID 兜底及热更新期间的数据一致性；特征抽取服务负责聚合多源数据并按照训练阶段的规则完成编码、转换与拼装；Infer Server 则在 CPU 或 GPU 上执行模型前向计算。对于 GPU 推理，可以利用 TensorRT 的计算图优化、算子融合和低精度计算，并结合动态批处理、FlashAttention、量化、知识蒸馏、中间结果复用与多模型混部等方法优化吞吐量、尾延迟和资源成本，但所有收益都应在目标模型、硬件和真实负载下验证。

最后，本章从在线与离线协同的角度讨论了两类典型模型：双塔模型通过预计算 Item Embedding、构建 ANN 索引，并按需实时或近线生成 User Embedding，实现大规模候选召回；DNN 精排模型在请求内组合用户、物品和上下文特征，通过序列建模与多目标预测完成更细粒度的排序。两类模型都依赖明确的模型、向量、索引和特征版本关系。统一特征定义与时间口径、开展离在线对照校验、回流线上实际特征并持续监控分布，是减少 Training-Serving Skew、保证模型离线收益能够稳定转化为线上效果的关键。