6. 推理与压缩

Table of contents

1 概览

当用户向 AI 助手发出一个请求时，系统必须在几百毫秒到几秒内完成一连串动作：接收请求、分词、处理 prompt、建立 KV cache、逐 token 生成答案、把结果流式发回客户端。这个过程看起来像一次“模型调用”，但工程师真正面对的却是一个资源协调问题：怎样在延迟、吞吐量、显存、成本与可靠性之间找到可运营的平衡点。

这也是本章的中心问题：工程师如何在真实世界约束下高效运行大语言模型？

推理首先是系统工程问题，而不只是模型问题。相同的底座模型，放在不同的服务栈、调度器、缓存策略和量化路径上，会呈现完全不同的用户体验与成本曲线。一个系统可能平均吞吐量很好，却首 token 很慢；也可能 tokens/s 看起来很高，却因为尾延迟和 OOM 让线上 SLA 崩掉。

理解这一点后，LLM 推理可以先被拆成两个基本阶段：

- 预填充（**prefill**）：处理输入 prompt，建立第一轮 hidden states 与 KV cache。
- 解码（解码）：在自回归循环中一步一步生成 token。

后续几乎所有推理优化——KV cache、连续批处理、chunked prefill、PagedAttention、FlashAttention、推测解码、量化、P/D 分离——本质上都在围绕这两个阶段的不同瓶颈做工程权衡。

读完本章后，你应该能够：

- 用系统视角解释 prefill 与解码的不同资源瓶颈。
- 估算 KV cache 大小，并判断上下文长度、batch size 与显存之间的关系。
- 理解连续批处理、调度、准入控制、前缀缓存和推测解码各自解决的问题。
- 区分 FlashAttention、PagedAttention、FlashDecoding、kernel fusion、tensor parallelism 的作用边界。
- 在权重量化、激活量化、KV 量化、剪枝、蒸馏和 QAT 之间做工程取舍。
- 为聊天、批处理、长上下文 RAG、单卡受限部署等不同场景写出合理的推理架构。
- 用故障模式思维定位首字时间高、生成速度抖动、OOM、量化退化和结构化输出不稳定等线上问题。

2 推理过程与性能瓶颈

> **!** Important
>
> 1. 一次线上 LLM 请求会经过哪些阶段？
> 2. prefill 与解码为什么具有不同的物理瓶颈？
> 3. 首字时间、生成速度、吞吐量分别反映了什么，为什么它们不能混成一个指标？
> 4. 当首字很慢或生成一卡一卡时，应该先怀疑哪里？

推理不是一个单一工作负载，而是一条由不同子阶段组成的流水线；只有先理解这条流水线的性能瓶颈，后面的优化才不会变成拍脑袋调参。

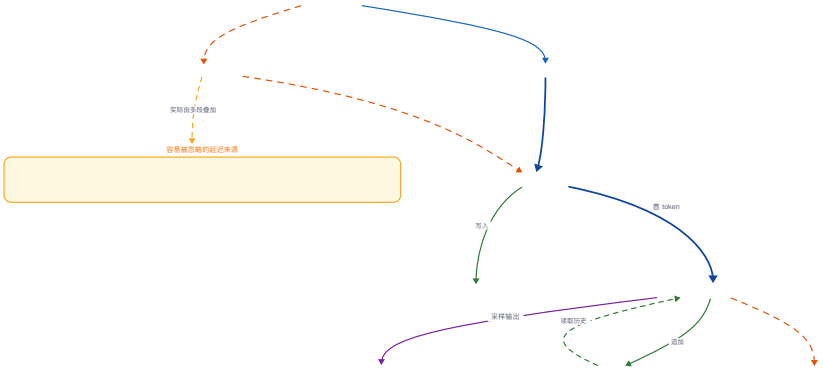

Figure 1: 一次 LLM 推理请求的系统路径

2.1 一次请求是怎样穿过推理系统的

从系统视角看，一次请求通常会经过下面五个阶段：

1. 请求进入与排队：网关接收请求，做鉴权、限流、路由和 SLA 分层。
2. 分词与预处理：tokenization、模板展开、schema 编译、工具参数拼装。
3. **prefill**：整段 prompt 一次性送入模型，计算各层表示并写出 KV cache。
4. 解码：模型在自回归循环中逐 token 生成输出。
5. 输出流式返回：采样结果被序列化后通过 SSE、WebSocket 或 gRPC 返回给客户端。

工程上最容易被忽略的是：用户看到的“首字时间”并不只由模型决定。排队、前缀缓存命中与否、grammar 编译、冷启动、网关尾延迟，都可能把首字时间拉长。相反，输出阶段看起来只是“多吐几个 token”，却往往把系统真正的解码瓶颈暴露得最明显。

2.2 prefill 与解码：同一个模型，两种工作负载

prefill 的输入序列长度是 L。系统会在每一层上并行处理整段 prompt，因此有大量矩阵乘法和 attention 计算可以一起做。这个阶段的特点是：

- token 维度并行度高；
- 通用矩阵乘法（GEMM General Matrix-Matrix Multiplication）比重大；
- attention 的中间工作集大；
- 很多优化目标都是让 GPU Tensor Core 尽量吃满。

解码则完全不同。每一步只新增一个 query 位置，但必须读取全历史的 KV cache，算出下一 token 的分布，再把新的 K/V 追加到缓存里。这个阶段的特点是：

- 序列维度串行；
- 每一步新增计算不大；
- 历史 KV 的读取量持续增长；
- 用户体感高度受单步解码速度影响。

因此，prefill 更像计算问题，解码更像带宽和状态管理问题。这也是为什么很多团队在一开始只盯着模型 FLOPs，却最后发现线上瓶颈其实是 KV cache、HBM 带宽和调度器。

2.3 为什么 attention 会随序列长度变贵

attention 之所以昂贵，不只是因为“公式复杂”，而是因为每个 **token** 都需要和越来越长的历史交互。

设输入序列长度为 L，隐藏维度为 d。在朴素自注意力里，我们先构造：

$$Q \in \mathbb{R}^{L\times d}, \qquad K \in \mathbb{R}^{L\times d}, \qquad V \in \mathbb{R}^{L\times d}$$

然后计算相关性分数矩阵：

$$S = QK^{\top} \in \mathbb{R}^{L\times L}$$

其中第 i 个 token 会和第 j 个 token 计算一次相似度，因此总共需要计算大约 L^2 个分数。仅这一项的计算复杂度就可以写成：

$$O(L^2 d)$$

而分数矩阵本身的存储规模是：

$$O(L^2)$$

再经过 softmax 并与 V 相乘：

$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}\left(\frac{QK^{\top}}{\sqrt{d}}\right)V$$

这一过程仍然要处理一个 $L \times L$ 级别的注意力矩阵。所以，当序列长度从 L 增长到 $2L$ 时，相关性计算和中间状态规模都会近似放大到原来的 4 倍。这就是朴素 attention 常被称为 $O(L^2)$ 的原因。

即使在推理阶段有 KV cache 帮助，问题也没有完全消失。KV cache 避免了对历史 token 的 K/V 反复重算，但在 decode 的第 t 步，当前 query 仍然需要与前面所有历史 K 做一次注意力计算，因此单步 attention 的读取和打分成本仍然与历史长度成正比：

$$O(td)$$

如果一共继续生成 T 个 token，那么仅从 decode 过程看，累计 attention 相关成本大致为：

$$\sum_{t=1}^{T} O(td) = O(T^2 d)$$

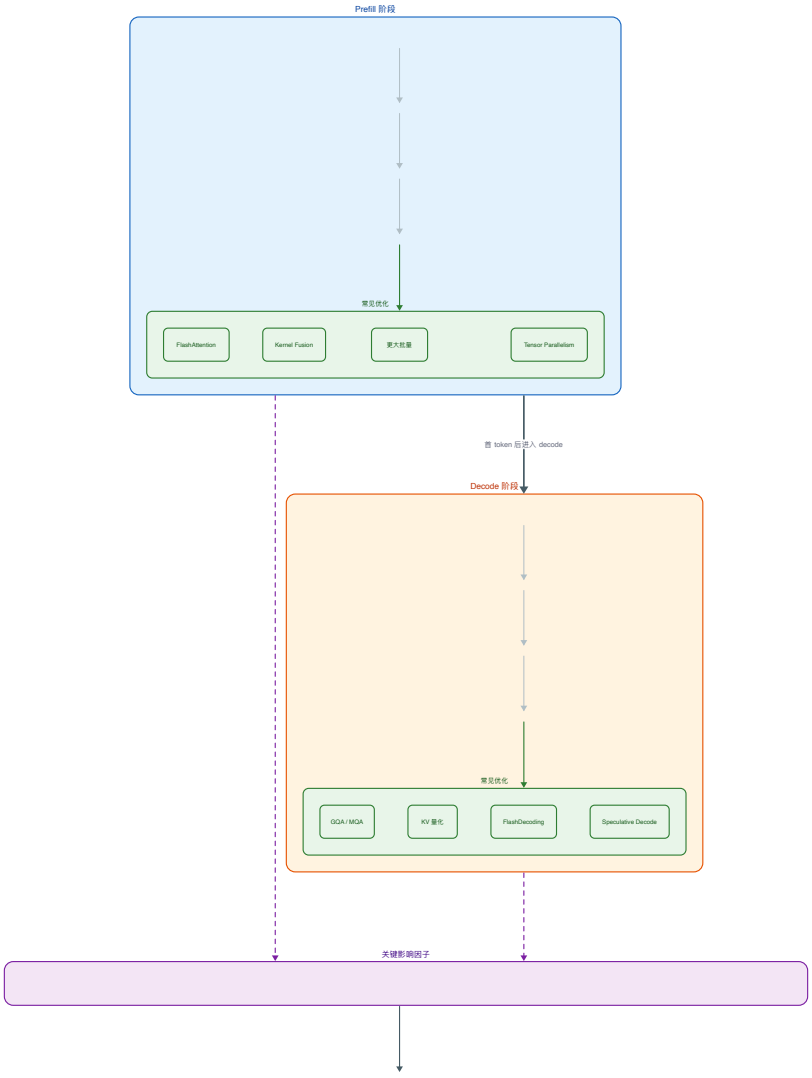

Figure 2: Prefill vs Decode : 两种不同的瓶颈

在更一般的情形下，如果提示长度为 L_{prompt}，后续生成长度为 T，则 decode 阶段的累计 attention 成本可以写成：

$$\sum_{t=1}^{T} O((L_{\text{prompt}} + t)d) = O(TL_{\text{prompt}}d + T^2 d)$$

这也是为什么上下文一长，decode 往往很快从“算一下下一个 token”变成“持续读取一大段历史状态”。

这对工程的含义很直接：

- prompt 变长会抬高 prefill 时间；
- 已生成历史变长会抬高解码的每步代价；
- 更长上下文不仅更慢，还会占用更多显存，因为 KV cache 也在线性增长。

换句话说，上下文长度不是一个“纯模型能力参数”，而是一个一等系统预算。

2.4 从计算与带宽看推理瓶颈

推理系统里最常见的误判，是把“慢”当成一个单一问题。其实不是。慢至少有两种：一种是算得慢，另一种是搬得慢。

判断一个阶段更接近哪一种，最有用的系统直觉之一，就是看它的算术强度（Arithmetic Intensity）：

$$AI = \frac{\text{FLOPs}}{\text{Bytes moved from memory}}$$

这个式子很简单，但很有力量。它问的是：你每搬运一份数据，究竟做了多少计算。

如果算术强度高，说明数据一旦进来，就能在芯片上做很多事。这样的阶段更容易先撞上计算屋顶。如果算术强度低，说明系统大部分时间都花在把数据搬来搬去。这样的阶段更容易先撞上带宽屋顶。

prefill 会一次性处理整段输入。矩阵乘法规模大，并行性强，Tensor Core 更容易被喂满。它更像是在做一项“大工程”：材料一旦到位，计算可以成片展开。**decode** 不一样。decode 每一步只生成一个 token。你做的计算变小了，但你仍然要反复读取权重、访问 KV cache、执行采样和调度。于是系统更像是在不停翻资料，而不是持续高强度思考。

可以把它理解成两种完全不同的工作方式：

- prefill 像你先完整读完一大段对话，然后集中进入状态；
- decode 像你已经进入状态，但接下来每说一个字，都要顺手翻一次旧记录。

这也是为什么“推理优化”从来不能一刀切。

放到大模型推理里，这个区别几乎决定了后面所有优化的方向：

阶段	算术强度	更常见的瓶颈	常见优化
长 prompt prefill	高	计算、kernel 效率	FlashAttention、kernel fusion、更大批量
单 token decode	低	HBM 带宽、KV 访问	GQA/MQA、KV 量化、FlashDecoding
长上下文 decode	更低	带宽 + 缓存布局	PagedAttention、KV 压缩、prefix reuse

同样是加速，一个方法如果只对大矩阵乘法有效，往往主要改善 prefill；如果一个方法主要减少状态读取或优化缓存布局，它更可能改善 decode。两者不在同一个瓶颈面前，自然也不应该用同一把锤子。

roofline 模型的价值，不在于把每个 kernel 都算到小数点后两位。它的价值在于，它给了你一个很快的判断框架：

> 当前 **GPU** 利用率不高，到底是因为没算满，还是因为一直在等数据？

这个问题一旦问对，排障就会快很多。否则你很容易在错误的方向上投入一周时间，最后只得到 3% 的改善。

2.5 首字时间、生成节奏与系统吞吐

推理系统最常见的几个指标，看上去都在描述“快不快”，但它们其实回答的是完全不同的问题。

- **TTFT（Time To First Token）**：系统多久开始“开口”。
- **ITL（Inter-Token Latency）/ TPOT**：系统说话的节奏是否稳定。
- 吞吐量（**throughput**）：单位时间能服务多少请求，或生成多少 token。

很多线上讨论之所以混乱，是因为大家把这三个目标混在一起谈。可从系统角度看，它们对应的根本不是同一件事。

TTFT 常常由下面几部分叠加而成：

$$TTFT \approx T_{\text{queue}} + T_{\text{preprocess}} + T_{\text{prefill}} + T_{\text{decode},1} + T_{\text{serialize}} + T_{\text{network}}$$

也就是说，首字时间并不只是“模型推得快不快”。它首先取决于请求有没有排队、预处理有没有拖慢、prefill 有没有被长上下文放大、第一步 decode 有没有被当前 batch 形态拖住，以及首个字节有没有尽快发到客户端。

ITL 或 TPOT 则不同。它更接近 decode 子系统的节奏问题，可以粗略写成：

$$TPOT \approx T_{\text{decode-step}} + T_{\text{sample}} + T_{\text{flush}}$$

这里最重要的不是“第一下开口”，而是系统后面能不能稳定地一个 token 一个 token 往前走。它常受下面这些因素影响：

- decode kernel 的效率；
- 高带宽显存（High Bandwidth Memory，HBM：GPU 显存和 GPU 计算单元之间，数据能以多快的速度来回搬运）；
- KV cache 的大小与访问模式；
- 当前 batch 的形态；
- 是否被长 prefill 打断；
- 约束解码或推测解码带来的额外开销。

> **Note**
>
> 关于 **GPU** 显存你可以把一台机器想成：
> CPU 是办公室里的经理 GPU 是专门干大规模并行计算的工厂内存（RAM）是办公室的大资料柜显存（VRAM/HBM）是工厂旁边的小资料柜
> GPU 真正算东西的时候，最想直接从显存里拿数据。因为显存离 GPU 近，带宽高得多。如果数据还在 CPU 内存里，就得先搬到 GPU 显存，才能高效计算。

吞吐量又是第三件事。它关心的是：

$$\text{Throughput} \approx \frac{\text{processed requests or output tokens}}{\text{unit time}}$$

这更像是“整条生产线一小时能出多少货”，而不是“某个用户感觉这次回得快不快”。

所以一个系统完全可能出现下面这种表面矛盾、其实很常见的现象：

- 吞吐量很高，但首字很慢；
- 首字不慢，但生成断断续续。

前一种系统优化的是总产能，后一种系统优化的是启动速度。这两者都可能是合理选择，但前提是你知道自己在优化什么。

对聊天产品来说，TTFT 往往决定“系统是不是显得活着”。对长文本生成来说，ITL 更决定“它是不是像在顺畅地说话”。对离线批处理或高并发平台来说，吞吐量和成本往往才是第一目标。

成熟的推理系统不会只盯一个指标。它会把这三个指标放进同一张记分牌里，因为它们共同描述的是三种不同的工程能力：

- 能多快开始
- 能多稳地持续
- 能多省地规模化

如果把这三件事混成一件事，系统设计就很容易走偏。

2.6 为什么单请求 decode 往往先卡带宽

很多人第一次接触大模型推理时，会自然地以为：GPU 越强，问题就越像“算得够不够快”。但在单请求 decode 里，现实往往不是这样。真正先撞上的，常常不是 FLOPs 上限，而是显存带宽。

先做一个很粗、但很有用的数量级估算。假设：

- H100 SXM 的显存带宽约为 3.35 TB/s；
- 一个 70B 的 BF16 模型，权重大约是 140 GB；
- 当前是单请求 decode，batch size = 1；
- 为了建立数量级直觉，粗略把“生成一个 token 需要扫过一遍主要权重”当作近似。

那么单 token decode 的理论上限可以近似写成：

$$\text{tokens/s} \approx \frac{3.35\ \text{TB/s}}{140\ \text{GB}} \approx 24$$

这个估算并不精确，也不是在描述某个具体框架下的真实吞吐。它的价值在于：它能迅速纠正对 **decode** 瓶颈的直觉。

一张 H100 的 BF16 算力当然很强，但在单请求 decode 场景里，系统每一步只生成一个 token，计算规模本身并不大；与此同时，却往往仍然需要不断从显存中读取大量权重与状态。于是系统的感觉更像是：不是“不会算”，而是“数据来不及搬到能算的地方”。

这个问题可以用一个很简单的画面理解。你不是在做一道特别难的题。你是在每写下一个字之前，都得先去档案室翻一大摞资料。慢，不是因为推理逻辑太复杂，而是因为搬资料这件事本身已经占了大头。

这也解释了为什么 prefill 和 decode 的优化方向明显不同。prefill 会一次性处理整段 prompt，矩阵乘法规模更大，并行性更强，更容易把 Tensor Core 喂满，因此常常更接近计算受限的问题；而 decode 是一步一步生成 token，每一步做的计算相对碎片化，却仍要反复读取权重、访问 KV cache、执行采样和调度，因此更容易表现为带宽受限或状态访问受限的问题。

换句话说，单请求 decode 的关键矛盾通常不是“这张卡算得够不够快？”，而是“这张卡能不能以足够高的速度，把需要的数据持续送到计算单元面前？”

现实系统里，实际速度往往还会比这个粗估更低，因为你还没有把下面这些成本算进去：

- KV cache 的读取；
- sampling 与 logits 后处理；
- 调度器与 runtime 开销；
- 框架本身的 kernel 实现效率；
- 网络发送与流式 flush；
- 以及各种不可避免的系统噪声。

因此，这个 24 tokens/s 更像是一条“物理直觉线”，而不是性能承诺。它告诉我们的核心结论是：

> 单请求 **decode** 往往首先是显存带宽问题，其次才是纯计算问题。

一旦这条判断成立，后面的很多工程选择就会变得自然起来。例如，GQA / MQA 通过减少 KV 访问体积来降低带宽压力；KV 量化通过缩小在线状态来减少读取成本；FlashDecoding、PagedAttention、prefix reuse 等机制，本质上也都在试图改善“每步生成时的数据访问路径”。

3 KV Cache

> **!** Important
>
> 本节先回答几个关键问题：
> 1. KV cache 到底缓存了什么，为什么它能加速解码？
> 2. KV cache 为什么会随着序列长度快速膨胀？
> 3. MHA、MQA、GQA 对 KV cache 的影响分别是什么？
> 4. PagedAttention 与 KV 量化分别解决哪类问题？
> 5. 长提示一上来就 OOM 时，工程上应该怎么收敛？

KV cache 让解码不必在每一步重算整段历史，但它也把“上下文长度”变成了显存与带宽问题。线上推理是否可扩展，很大程度上取决于你如何分配、复用、压缩和保护这份状态。

3.1 为什么解码一定要依赖 KV cache

如果没有 KV cache，模型在生成第 t 个 token 时，必须重新计算前面所有 token 的 K 和 V。这样做的代价是重复而且昂贵的。KV cache 的做法是：

- 在 prefill 阶段，把历史 token 的 Key 和 Value 按层存下来；
- 到解码时，只为新 token 计算新的 Q/K/V；
- 注意力直接读取历史 K/V，再把新的 K/V 追加到缓存里。

因此，KV cache 的核心价值不是“让 attention 变简单”，而是避免重复计算历史。这也是为什么它几乎是所有 decoder-only 线上服务的默认配置。

3.2 KV cache 如何随序列增长

对 decoder-only 模型，单个 token 的 KV cache 近似大小可以写成：

$$KV_{\text{bytes/token}} = 2 \cdot N_{layers} \cdot H_{kv} \cdot d_{head} \cdot p$$

其中：

- 2 表示 K 与 V 两份；
- N_{layers} 是层数；
- H_{kv} 是 KV heads 数；
- d_{head} 是 head dimension；
- p 是每个元素的字节数，例如 FP16 为 2 bytes。

总 KV 大小则近似为：

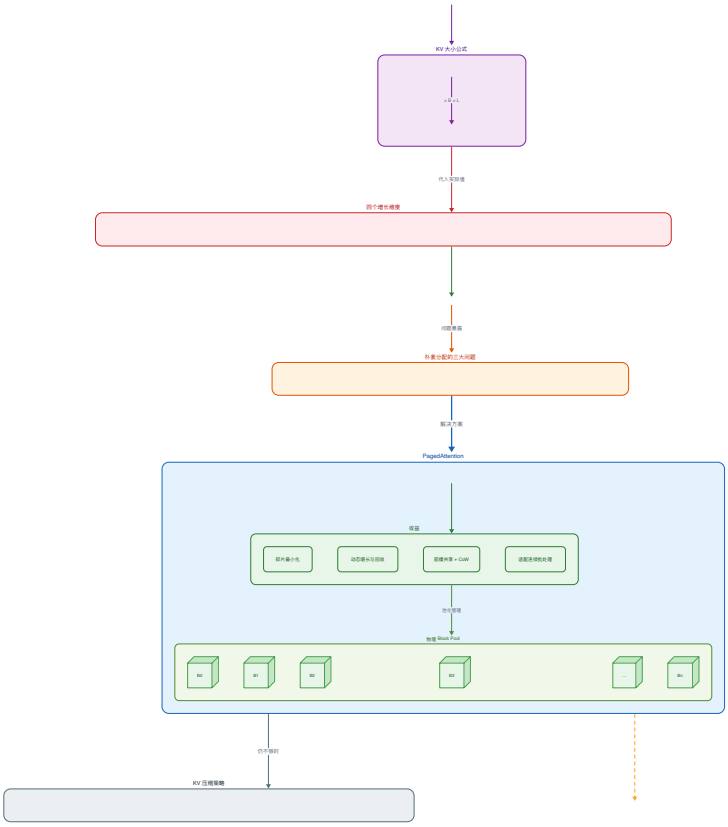

Figure 3: KV Cache 的增长与 Paged Attention

$$KV_{\text{total}} = B \cdot L \cdot KV_{\text{bytes/token}}$$

其中 B 是活跃序列数，L 是每条序列当前长度。

代入一个常见例子：

- $N_{layers} = 80$
- $H_{kv} = 8$
- $d_{head} = 128$
- $p = 2$ bytes (FP16)

则：

$$KV_{\text{bytes/token}} = 2 \cdot 80 \cdot 8 \cdot 128 \cdot 2 = 327{,}680 \text{ bytes}$$

也就是大约 **320 KiB / token**。如果上下文长度是 8K，那么单条序列的 KV cache 大约是 **2.5 GiB**。这也是为什么“长上下文 + 大 batch”很容易比模型权重更早成为显存上限。

3.3 MHA、GQA、MQA：直觉、数学实现与 KV cache 代价

你可以把这三种机制理解成一个问题：**Query** 有很多**“看问题的角度”**，那历史信息 **K/V** 要不要也给每个角度都单独存一份？这是 MHA、MQA、GQA 的本质区别。

从朴素 **attention** 开始

在 attention 里，当前 token 会产生一个 **Query**，历史 token 会提供 **Key / Value**。你可以粗略理解成：

- **Q（Query）**：我现在想找什么
- **K（Key）**：历史里每条信息的“索引标签”
- **V（Value）**：历史里真正要拿出来用的内容

attention 做的事就是：

1. 用当前的 Q 去和所有历史 K 匹配
2. 算出“我现在最该关注哪些历史信息”
3. 再把对应的 V 加权取出来

公式还是那条：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

为什么会有**“多头”**？

因为只用一个 Query / Key / Value 视角，模型看问题太单一。所以 Transformer 会把 attention 拆成很多个 head。每个 head 都像一个不同的观察角度：

- 一个 head 可能更关注语法
- 一个更关注长距离依赖

- 一个更关注实体对齐
- 一个更关注位置关系

多头注意力 **MHA**：每个头都配自己的一套 **K/V**

在 MHA 里，每个 query head 都有自己对应的一套 key/value head。如果一共有 H 个 attention heads，那么：

- 有 H 个 query heads
- 有 H 个 key heads
- 有 H 个 value heads

也就是每个视角都自己记一份历史。

这最细致，也最贵。像什么？像你从 32 个不同角度理解一段关系，然后给这 32 个角度都各自建一份完整档案。优点是信息保留最充分。缺点是：**KV cache** 最大。因为推理时最贵的不是 Q，而是历史的 K/V 要一直存着。

MQA（Multi-Query Attention）：很多 **Query** 共用一套 **K/V**

MQA 的关键不是“只有一个 query”，而是：

- 仍然有很多个 query heads
- 但它们共享同一套 **K/V**

也就是：

- Query 头数还是 H
- 但 Key / Value 头数只有 1

所以：

$$H_{kv} = 1$$

很多人可以从不同角度提问，但历史档案只保留一份。这当然会极大减少 KV cache。如果原来是 MHA，要为每个 head 存一份历史；
现在 MQA 只存一份，显存和带宽压力都小很多。

问题是：不同 query heads 明明想看的东西不一样，结果却都只能共用同一套历史表示。所以它更省，但可能更容易损失质量，尤其在复杂任务或长上下文里。

GQA（Grouped-Query Attention）：折中方案

GQA 是介于 MHA 和 MQA 中间的做法。它的意思是：

- query heads 还是很多个
- 但不是所有 query 都共用一套 K/V
- 而是分组共享

比如：

- 32 个 query heads
- 8 个 KV heads

那就是每 4 个 query heads 共用 1 组 K/V。也就是：

$$H = 32, \qquad H_{kv} = 8$$

不是每个人都单独建档，也不是所有人都挤一个档案室。而是分几个组，每组共享一份。这样就形成折中：

- 比 MHA 省显存、省带宽
- 比 MQA 保留更多表达能力

这就是为什么现在很多在线服务模型偏爱 GQA。

为什么这三者的关键差别在 **KV cache**？

因为在推理里，尤其是 decode 阶段：

- **Q** 是当前 token 现算的
- **K/V** 是历史 token 要缓存下来的

也就是说，真正会随着上下文长度不断变大的，是 K/V，而不是 Q。所以 MHA、MQA、GQA 的本质，不是在改“注意力公式”，而是在改：历史状态到底要存多少份。

缩减因子怎么理解

如果 query head 数是 H，KV head 数是 H_{kv}，
那么相对 MHA，KV cache 的缩减因子近似是：

$$\frac{H}{H_{kv}}$$

为什么？因为 MHA 默认：

$$H_{kv} = H$$

每个 query head 都有自己的 K/V。

而如果你减少到 H_{kv} 个 KV heads，
缓存规模大致就按比例缩小。

举个例子，如果：

$$H = 32$$

MHA

$$H_{kv} = 32$$

缩减因子：

$$\frac{32}{32} = 1$$

不缩。

GQA

$$H_{kv} = 8$$

缩减因子：

$$\frac{32}{8} = 4$$

也就是 KV cache 大约缩成原来的 $1/4$。

MQA

$$H_{kv} = 1$$

缩减因子：

$$\frac{32}{1} = 32$$

也就是 KV cache 大约缩成原来的 $1/32$。所以 MQA 省得最狠。

为什么这对 **decode** 特别重要？

因为 decode 常常不是先卡 FLOPs，而是先卡：

- HBM 带宽
- KV cache 读取
- 显存容量

而 MHA / MQA / GQA 恰好直接影响这三件事。如果 K/V 存得越少：

- 显存占用越小
- 历史读取量越小
- decode 带宽压力越小
- 单步生成通常越容易变快

所以选 attention 这不是一个抽象架构选择，而是一个很现实的推理系统问题。

Note

三种注意力 **MHA / MQA / GQA** 的数学表达

符号约定. 设 batch size B, 序列长度 L, 隐藏维度 d_{model}, query head 数 H, KV head 数 H_{kv}, 每 head 维度 d_h。通常 $d_{\text{model}} = H \cdot d_h$。输入隐状态 $X \in \mathbb{R}^{B\times L\times d_{\text{model}}}$。

MHA（Multi-Head Attention）

三组投影矩阵各有 H 组 head：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

$$W_Q,\ W_K,\ W_V \in \mathbb{R}^{d_{\text{model}}\times(H\,d_h)}$$

投影后 reshape：$Q, K, V \in \mathbb{R}^{B\times L\times H\times d_h}$。

每个 head 独立做 attention，再拼接输出：

$$\text{Attn}^{(h)} = \text{softmax}\left(\frac{Q^{(h)}{K^{(h)}}^{\top}}{\sqrt{d_h}}\right)V^{(h)}, \qquad Y = \text{Concat}(\text{Attn}^{(1)}, \dots, \text{Attn}^{(H)})\,W_O$$

MQA（Multi-Query Attention）

Q 仍有 H 组 head，但 **K/V 只有 1 组**：

$$\begin{aligned} W_Q \in \mathbb{R}^{d_{\text{model}}\times(H\,d_h)} &\implies Q \in \mathbb{R}^{B\times L\times H\times d_h} \\ W_K,\ W_V \in \mathbb{R}^{d_{\text{model}}\times d_h} &\implies K, V \in \mathbb{R}^{B\times L\times 1\times d_h} \end{aligned}$$

计算时把这一组 K/V 广播给所有 query head：

$$\text{Attn}^{(h)} = \text{softmax}\left(\frac{Q^{(h)}K^{\top}}{\sqrt{d_h}}\right)V$$

不同的 Q head 仍然存在，但它们共享同一份历史 K/V。

GQA（Grouped-Query Attention）

MHA 与 MQA 之间的折中：H_{kv} 组 KV head，每组被 $g = H/H_{kv}$ 个 query head 共享。

$$\begin{aligned} W_Q \in \mathbb{R}^{d_{\text{model}}\times(H\,d_h)} &\implies Q \in \mathbb{R}^{B\times L\times H\times d_h} \\ W_K,\ W_V \in \mathbb{R}^{d_{\text{model}}\times(H_{kv}\,d_h)} &\implies K, V \in \mathbb{R}^{B\times L\times H_{kv}\times d_h} \end{aligned}$$

第 h 个 query head 映射到 KV head $\phi(h) = \lfloor h/g \rfloor$：

$$\text{Attn}^{(h)} = \text{softmax}\left(\frac{Q^{(h)}{K^{(\phi(h))}}^{\top}}{\sqrt{d_h}}\right)V^{(\phi(h))}$$

Q 头很多，K/V 头更少；多个 Q 头映射到同一个 KV 头。

变体	KV head 数	KV 缓存缩放	典型场景
MHA	$H_{kv} = H$	$1\times$	训练精度最高
GQA	$1 < H_{kv} < H$	H_{kv}/H	Llama 2/3, Qwen 2
MQA	$H_{kv} = 1$	$1/H$	极致推理效率

小结一下，三种注意力机制思考的是 如何在注意力表达能力和在线 **KV** 成本之间做交换。

- **MHA**：每个 query head 都有对应的 K/V head，质量好，但 KV cache 最大。
- **MQA**：所有 query heads 共享一组 K/V，解码更快，KV cache 最小，但可能损失质量。
- **GQA**：若干个 query heads 共享一组 K/V，在速度、显存与质量之间做折中。

如果 query head 数为 H，KV head 数为 H_{kv}，那么相对 MHA，KV cache 的缩减因子近似是：

$$\frac{H}{H_{kv}}$$

这也是为什么现代在线服务偏爱 GQA：它比 MHA 更省显存和带宽，但通常比 MQA 更稳。

Note

例子：**32 个 Query heads 如何映射到 KV heads**

下面用一个最常见的例子来直观看：

- query heads 数：$H = 32$
- 每个 head 维度：d_h
- 对比三种方案：
  - MHA：$H_{kv} = 32$
  - GQA：$H_{kv} = 8$
  - MQA：$H_{kv} = 1$

MHA：32 个 Q heads 对应 32 个 KV heads

这里是一一对应：

$$Q_1 \to KV_1, \quad Q_2 \to KV_2, \quad \dots, \quad Q_{32} \to KV_{32}$$

```
Q heads:   Q01 Q02 Q03 Q04 Q05 Q06 Q07 Q08 Q09 Q10 Q11 Q12 Q13 Q14 Q15 Q16
           Q17 Q18 Q19 Q20 Q21 Q22 Q23 Q24 Q25 Q26 Q27 Q28 Q29 Q30 Q31 Q32

KV heads:  K01 K02 K03 K04 K05 K06 K07 K08 K09 K10 K11 K12 K13 K14 K15 K16
           K17 K18 K19 K20 K21 K22 K23 K24 K25 K26 K27 K28 K29 K30 K31 K32

mapping:   Q01→K01, Q02→K02, ..., Q32→K32
```

这时：

$$H_{kv} = H = 32$$

所以 KV cache 不缩减，属于基线情况。

GQA：32 个 Q heads 分 8 组共享 8 个 KV heads

假设：

$$H = 32, \qquad H_{kv} = 8$$

那么每个 KV head 会被：

$$g = \frac{H}{H_{kv}} = \frac{32}{8} = 4$$

个 query heads 共享。

也就是说，映射关系变成：

```
Q01 Q02 Q03 Q04  →  KV01
Q05 Q06 Q07 Q08  →  KV02
Q09 Q10 Q11 Q12  →  KV03
Q13 Q14 Q15 Q16  →  KV04
Q17 Q18 Q19 Q20  →  KV05
Q21 Q22 Q23 Q24  →  KV06
Q25 Q26 Q27 Q28  →  KV07
Q29 Q30 Q31 Q32  →  KV08
```

如果写成更紧凑的图：

```
Q heads:
[Q01 Q02 Q03 Q04] [Q05 Q06 Q07 Q08] [Q09 Q10 Q11 Q12] [Q13 Q14 Q15 Q16]
[Q17 Q18 Q19 Q20] [Q21 Q22 Q23 Q24] [Q25 Q26 Q27 Q28] [Q29 Q30 Q31 Q32]

KV heads:
       KV01                KV02                KV03                KV04
       KV05                KV06                KV07                KV08
```

数学上可以写成一个分组映射函数：

$$\phi(h) = \left\lfloor \frac{h}{g} \right\rfloor \qquad \text{其中} g = \frac{H}{H_{kv}}$$

如果从 0 开始编号：

- Q_0, Q_1, Q_2, Q_3 用 KV_0
- Q_4, Q_5, Q_6, Q_7 用 KV_1
- …
- $Q_{28}, Q_{29}, Q_{30}, Q_{31}$ 用 KV_7

这时 KV cache 相对 MHA 的缩减因子约为：

$$\frac{H}{H_{kv}} = \frac{32}{8} = 4$$

也就是说，KV cache 大约变成原来的 $1/4$。

MQA：32 个 Q heads 全部共享 1 个 KV head

这是最极端的共享方式：

$$H = 32, \qquad H_{kv} = 1$$

所以 32 个 query heads 都共享同一套 K/V：

```
Q01 Q02 Q03 Q04 Q05 Q06 Q07 Q08
Q09 Q10 Q11 Q12 Q13 Q14 Q15 Q16
Q17 Q18 Q19 Q20 Q21 Q22 Q23 Q24
Q25 Q26 Q27 Q28 Q29 Q30 Q31 Q32
                  ↓
                KV01
```

也就是：

$$Q_1, Q_2, \dots, Q_{32} \to KV_1$$

这时 KV cache 相对 MHA 的缩减因子约为：

$$\frac{H}{H_{kv}} = \frac{32}{1} = 32$$

所以 KV cache 大约可以缩到原来的 $1/32$。

把三者放在一起看

```
MHA:
Q01 → KV01
Q02 → KV02
...
Q32 → KV32

GQA (32Q, 8KV):
Q05 Q06 Q07 Q08 → KV02
...
Q29 Q30 Q31 Q32 → KV08

MQA:
Q01 Q02 Q03 ... Q32 → KV01
```

如果只看“历史状态到底存多少份”，它们的区别就很清楚了：

- **MHA**：每个视角都单独存历史
- **GQA**：每组视角共用一份历史
- **MQA**：所有视角共用一份历史

3.4 朴素 KV 分配为什么会浪费显存

如果把每个请求的 KV cache 都分配成一大块连续内存，会立刻遇到三个问题：

1. 外部碎片：长短请求交替进出，连续空间越来越难复用；
2. 内部浪费：序列长度很少恰好对齐分配粒度；
3. 共享困难：beam search、并行采样、前缀复用等场景下，相同前缀的 KV 很难在请求之间共享。

这会直接伤害在线服务的最大 batch size、可用显存和 tail latency。于是推理系统开始引入页式分配思想。

3.5 PagedAttention：让 KV cache 像虚拟内存一样管理

PagedAttention 不是在改变 attention 的数学公式，而是在改变 KV cache 在 GPU 显存里的存法，解决的是 GPU fragmentation 的问题。

具体做法是：不再给每条请求预留一整块连续的大显存，而是把 KV cache 切成很多固定大小的 block/page，按需分配，再用一张 block table 把“逻辑上的连续历史”映射到“物理上不连续的显存块”。类似于你在管理一个仓库，每来一个人，就给他在仓库里硬留一整排连续货架；结果很多人东西没放满，仓库却被切得乱七八糟，新的大客户反而塞不进去。PagedAttention 聪明地解决了这个问题：把 KV cache 拆成很多固定大小的小格子，谁需要就拿几个，不要求连在一起，再用一张表记录“这条序列的第几段历史放在哪个格子里”。

这个思路直接借鉴了操作系统的分页内存管理。这样做带来几个重要收益：

- 降低碎片，显存浪费接近最小化；
- 让请求可以动态增长和回收，不需要大块搬家；
- 更容易做前缀共享与 copy-on-write；
- 更适合连续批处理中的动态请求进出。

这里很容易和 FlashAttention 混淆。两者并不是替代关系：

- **PagedAttention** 解决的是 KV cache 的内存布局和生命周期管理。存的时候怎么少碎片、少浪费空间？
- **FlashAttention** 解决的是 attention kernel 的 IO 效率；算的时候怎么少搬数据？

一个现代服务栈通常会同时使用二者。

3.6 KV 量化：拿一点数值精度，换更多上下文和 batch

KV cache 也可以量化。最直观的路径是：

- FP16 $\rightarrow$ INT8：理论上接近 $2\times$ 空间节省；
- FP16 $\rightarrow$ 4-bit：理论上接近 $4\times$ 空间节省。

这对系统有两层价值：

1. 允许更长上下文或更大 batch；
2. 降低解码时读取 KV 的字节数，从而缓解带宽压力。

但代价也很现实。很多任务不会先在“闲聊质量”上退化，而会先在以下场景暴露问题：

- 长文问答中的远距离引用；
- 大海捞针 (Needle-in-a-Haystack)；

- 代码和数学这类对细粒度符号更敏感的任务；
- 需要精确实体绑定和跨段定位的场景。

因此，权重量化与 KV 量化虽然都叫“量化”，它们面向的问题并不一样。权重量化优先解决模型放不下，**KV** 量化优先解决上下文太贵。

> **Note**
>
> 大海捞针（Needle-in-a-Haystack）把一句很重要的话藏进一大段很长的内容里，看模型还能不能把它准确找出来；它特别适合用来检查模型在长文、长对话或 RAG 里有没有真的“看见重点”。

3.7 极长上下文下，单靠放大窗口通常不是解法

> **Note**
>
> **context window** 现在大概有多长？
> 截止 2026 年 3 月，工程上更重要的不是“模型宣称支持多长”，而是“默认开多长才真正可控”。
>
> - 聊天、助理、普通工具调用：常见的舒适区仍然是 8K–32K。这通常已经足够覆盖最近几轮对话和少量工具结果，同时也更容易控制 TTFT、显存占用和单位请求成本。
> - 长文问答、RAG、代码库分析：常见会走到 32K–128K。再往上并不是不能做，而是 KV cache、尾延迟和整体服务成本会明显上升。
> - 超长上下文场景：现在主流前沿模型已经出现 200K、400K，甚至 1M 级别的 context window。Anthropic 的 Claude 文档明确给出 200K 与 1M 级上下文能力，Google 的 Gemini 官方文档也把 1M 长上下文作为核心能力之一，OpenAI 目前也有 1.05M context window 的模型。

但要注意：“能塞进去”不等于“应该默认塞满”。当窗口变得极大时，问题往往不再是模型“能不能看见”，而是系统“能不能稳定、便宜、可靠地服务”。因此在真实系统里，长上下文通常要和更好的 context engineering、检索、摘要和外部记忆结构一起设计，而不是只靠把 window 一路做大。当上下文极长时，仅仅把 window 继续做大，往往会让问题从“模型能不能看见”变成“系统根本服务不起”。

常见的 KV 压缩或保留策略大致有四类：

1. 量化：最通用，最容易和现有栈整合；
2. 驱逐 / 选择性保留：只保留 heavy hitters、最近窗口或预测更重要的 token；
3. 滑动窗口 / **sink tokens**：适合持续流式场景；
4. 分层记忆：把一部分历史移出即时上下文，由检索或外部记忆取回。

实践里，长上下文问题常常不是靠“更大的纯上下文”解决，而是靠“更好的记忆结构”解决。如果你的任务真的需要精确跨段引用，RAG、层级摘要和检索压缩通常比死撑 window 更可控。

3.8 长提示 OOM 的排障顺序

当长 prompt 一上来就 OOM，一个更成熟的处理顺序是：

1. 先减少输入：去重、rerank、压缩、摘要、context packing；
2. 确认模型是否是 GQA/MQA 架构；
3. 先做权重量化，为 KV 留空间；
4. 再考虑 KV 量化；
5. 使用 PagedAttention，减少显存碎片；
6. 开启 prefix cache，避免重复 prefill；
7. 引入 chunked prefill，避免长 prompt 一次占满 token budget；
8. 仍不够时，再考虑 sliding window、选择性保留、offload 或多卡并行；
9. 最后再问：这个业务是不是应该换更小模型。

这个顺序背后有一个非常工程化的判断：先解决“浪费”，再解决“缺少”，最后才是“买更大的卡”。

4 批处理、调度与请求编排

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 批处理为什么是在线推理的核心杠杆？
> 2. 连续批处理与静态批处理有何不同？
> 3. 为什么 admission control 和 chunked prefill 经常比“更大的 batch”更重要？
> 4. prefix caching、推测解码、guided decoding 分别适合什么场景？

在线推理不是“把请求扔给 GPU”那么简单，而是一个持续的调度问题。你优化的不只是单个请求，而是所有请求如何共享同一块加速器。

4.1 批处理

批处理的本质，不是“把几条请求凑在一起”这么简单，而是把每轮前向中的固定成本——例如权重读取、kernel 启动和调度开销——分摊到更多 token 上。对自回归大模型来说，这一点尤其重要：生成是逐 token 迭代的，如果每次只服务一条请求，GPU 很容易因为工作太碎而吃不满；而把多条请求并行起来后，系统的首要收益通常不是单条请求更快，而是整体 **tokens/s** 更高、**GPU** 更饱和、单位 **token** 成本更低。

这也是为什么批处理几乎总能立刻提升服务吞吐量：

- GPU 利用率更高；
- 权重读取和一部分 kernel 开销被更多 token 共同分摊；
- 在相同硬件下，系统能稳定产出更多 tokens/s。

但批处理从来不是白赚。对单个用户来说，batch 变大往往意味着两件事：一是请求可能要先等系统凑批；二是后续每个 decode step 都要和更多序列共享同一轮调度。于是在线系统真正追

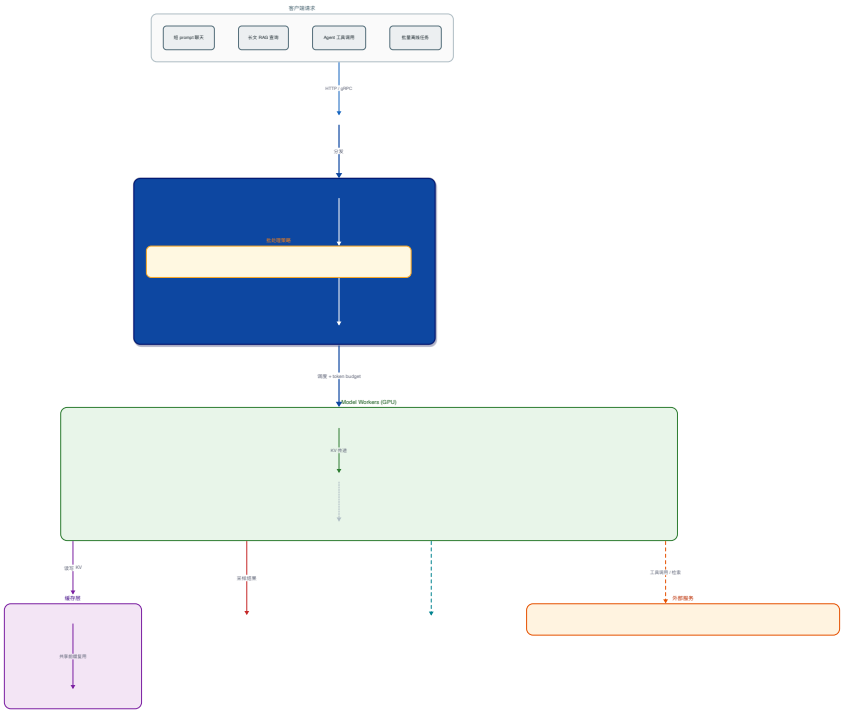

Figure 4: **批处理、调度器与服务架构**

求的，从来不是“把 batch 开到最大”，而是在吞吐量和体感延迟之间，找到一个业务上最划算的平衡点。

4.2 静态批处理 vs 连续批处理

静态批处理更像传统深度学习推理：先攒够一批请求，再一起跑完这一批。这个做法在离线任务里很自然，但在 LLM 服务里很快就会暴露问题。原因很简单：不同请求的 prompt 长度不一样，输出长度更不一样；有的序列很快结束，有的序列还要继续生成。如果系统必须等“整批都结束”才能释放槽位，那么先完成的请求会被后完成的请求拖住，新到的请求也只能继续排队。

连续批处理的关键变化，是把调度粒度从“请求”降到“迭代”。系统不再等整批完成，而是在每个 decode iteration 之后检查：谁已经结束了，谁还在运行，哪些新请求可以立刻补进来。我们称之为 **iteration-level scheduling**。也可以更直白地理解：完成的请求立刻移除，新请求立刻补位，再结合 ragged batching 与 chunked prefill，把 GPU 尽量维持在“总有活干”的状态。

连续批处理并不只是“更动态的 batch”，而是把批处理从一次性动作，变成一个持续运行的在线调度器。它的目标不是组出一个漂亮的批次，而是让 GPU 在面对长短不一、随时进出的真实流量时，始终把空闲时间压到最低。

> **Note**
>
> **Ragged batching**：让不等长请求按真实长度合批
> Ragged batching 要解决的，不是“能不能把请求放进同一个 batch”，而是放进去以后，是否还要为了对齐 **shape** 而做大量无效 **padding**。
> 设一批请求的长度分别为 $L_1, L_2, \dots, L_B$。
> 在传统 padding batch 里，系统通常会把所有序列补到同一个最大长度：
>
> $$L_{\max} = \max_i L_i$$
>
> 于是这一批在长度维度上的处理规模大致变成：
>
> $$B \cdot L_{\max}$$
>
> 但真正有意义的 token 数只有：
>
> $$\sum_{i=1}^{B} L_i$$
>
> 两者之间的差值：
>
> $$B \cdot L_{\max} - \sum_{i=1}^{B} L_i$$
>
> 本质上就是 padding 带来的浪费。序列长度差异越大，这部分浪费越明显。
> Ragged batching 的思路是：不要为了“排整齐”而把短序列补到和长序列一样长，而是直接按真实长度把它们拼接起来，再额外记录边界、**offset** 或 **mask**，让系统知道哪一段属于哪条请求。

例如三条请求长度分别为 3、4、5，普通 padding batch 往往会按总长度 15 来处理；而 ragged batching 只处理真实存在的 12 个 token，并通过边界信息保证不同请求之间不会互相串扰。
从系统角度看，这样做的收益很直接：**GPU 花在真实 token 上的时间更多，花在“空位”上的时间更少。**
这也是为什么 ragged batching 常常和 continuous batching 一起出现。连续批处理面对的是一组不断进入、不断完成、长度又各不相同的在线请求；如果还坚持把它们整理成一个规整矩形，padding 开销会很快吞掉一部分本来可以拿来服务真实 token 的预算。Ragged batching 则允许系统在保持合批收益的同时，减少这种对齐成本。
当然，它不是免费午餐。你省掉了 padding，就必须更认真地维护：

- 每条请求在拼接张量中的起止位置；
- 对应的 attention mask 或边界索引；
- 以及 batch 内不同序列的状态映射关系。

所以，ragged batching 本质上是一种很典型的工程交换：用更复杂的批次描述与调度逻辑，换更少的无效计算和更高的吞吐效率。Ragged batching 就是让不同长度的请求按真实长度合批，而不是先补齐成一样长，再把大量算力浪费在 padding 上。

4.3 准入控制

连续批处理提高的是利用率，不是“请求越多越好”。一旦系统开始接近显存或 token budget 上限，最危险的情况往往不是平均变慢一点，而是进入一种不稳定状态：请求排队时间迅速拉长，少数长 prefill 抢走大部分预算，KV cache 空间不够时还会触发 preemption 和 recompute，进一步拖累端到端延迟。vLLM 里，当 KV cache 空间不足以容纳当前批次请求时，系统会发生 preemption；而频繁的 preemption / recomputation 会直接伤害端到端性能。

所以，准入控制本质上是在回答一个非常现实的问题：系统已经很忙了，这个新请求现在到底该不该进来？

成熟系统不会把这个问题交给“运气”解决，而会把它做成显式策略。最常见的控制杠杆包括：

- 限制最大活跃序列数；
- 限制总 batched tokens；
- 对超长 prompt 分流，避免它们和普通短请求争同一预算；
- 在多租户或多 SLA 场景下做优先级隔离。

vLLM 给出的调优建议：如果 preemption 频繁发生，可以增加可用的 KV 空间，或者直接降低 `max_num_seqs`、`max_num_batched_tokens`，本质上就是减少系统一次想同时吃下的请求量。从生产角度看，学会在高压时“少接一点”，通常比让所有请求一起变慢、甚至一起抖动，要健康得多。

4.4 chunked prefill：解决 convoy effect

长 prompt 最麻烦的地方，不只是它自己慢，而是它容易把别人也拖慢。一个超长 prefill 如果一次性吃掉大量 token budget，后面的短请求就会像堵在一辆慢车后面——这就是典型的 **convoy effect**。

chunked prefill 的做法，不是把长请求“变便宜”，而是把它切成多个较小的 prefill chunk，让调度器可以把这些 chunk 和 decode 请求穿插起来执行。vLLM 的文档对这个机制说得很明确：系统会优先调度 decode；在 `max_num_batched_tokens` 还有剩余额度时，再把待处理的 prefill 填进去；如果某个 prefill 太长、放不进当前预算，就自动把它切块。这样一来，超长 prompt 就不再是一口气霸占整轮资源，而是被拆成多个可以与其他请求交错执行的小段。

这种调度方式通常会带来三类好处：

- 更好的 **ITL / 尾延迟**：因为 decode 被优先保障，短请求不容易一直被长 prefill 压住；
- 更高的 **GPU 利用率**：compute-bound 的 prefill 和 memory-bound 的 decode 更容易混合到同一批里；
- 更少的系统性拖累：少数超长请求不再轻易把整池请求都堵住。

当然，它也有代价。chunk 太小，调度和管理开销会上升；chunk 太大，又会重新接近“整段长 prefill 直接堵路”的老问题。vLLM 在调参建议里也把这个权衡说得很直接：较小的 `max_num_batched_tokens` 往往有利于 ITL，较大的值则更有利于 TTFT 和总体吞吐。换句话说，chunked prefill 本质上不是一个绝对加速开关，而是一个在公平性、吞吐量和尾延迟之间做折中的调度工具。

4.5 前缀缓存：最划算的线上加速手段之一

前缀缓存（prefix caching）的核心思想是：当多个请求共享相同前缀时，直接复用对应 KV cache，而不是重新做 prefill。

它特别适合以下场景：

- 相同 system prompt；
- 相同工具说明和 JSON schema；
- 相同文档前缀或知识包；
- 模板化 RAG；
- 多轮会话中未变化的长历史。

前缀缓存（prefix caching）的收益通常非常直接：首字时间下降、GPU 使用率提高、成本下降。但它不是纯性能功能，还牵涉到隔离与安全。多租户环境下，如果命中与否可被观察到，就可能形成侧信道。因此生产环境常见做法包括：

- 按 tenant 或 trust group 分 namespace；
- 对高敏感场景引入 cache salt；
- 对某些工作负载禁用跨租户复用。

4.6 推测解码：用额外算力换更少的大模型步数

推测解码的基本思路是：

1. 先用更快的小模型或轻量头猜多个 token；
2. 再用目标大模型一次性验证；
3. 如果草稿猜对了，大模型就相当于少做了若干个真正的解码 step。

它的工程价值不在“答案更好”，而在于：在不改变目标分布的前提下，减少大模型昂贵的串行解码次数。

但是推测解码只有在下面条件成立时才真正赚钱：

- 草稿便宜；
- 接受率够高；
- 目标模型当前确实被解码卡住；
- 验证开销没有把收益吃掉。

因此它的胜利条件可以写成：

$$\text{draft 开销} + \text{verify 开销} < \text{省下的 target 解码开销}$$

失败模式同样很明确：草稿接受率低、目标 GPU 已经很饱和、流量形态不适合，都会让推测解码反而更慢。

4.7 guided decoding：降低链路重试，而不只是“让 JSON 更好看”

当模型输出必须满足 JSON、工具参数、SQL 或某个固定 schema 时，只靠 prompt 往往不够稳。guided decoding / constrained decoding 的思路，是在采样时直接把合法 token 集限制在 grammar 或 FSM (finite-state machines) 允许的范围内。

它的系统意义非常大：

- 降低无效输出与应用层 parse 失败；
- 减少“先生成，再校验，再重试”的回路；
- 让工具调用和 agent 系统的下游逻辑更可靠。

当然，它不是零成本的。复杂 grammar 会带来额外状态追踪与 token 过滤开销；在高 QPS 下，如果 grammar 编译或状态维护不够高效，也会吞掉一部分性能收益。工程上真正划算的，不是“所有输出都上最强约束”，而是把约束放在最容易引发链路失败的地方。

5 GPU 内存管理

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 推理时 GPU 显存主要被哪些对象占用？
> 2. 为什么大模型服务必须把内存当作预算来规划，而不是运行时报错后再修？
> 3. memory pooling、paged blocks、dynamic 批处理如何相互配合？
> 4. 当显存接近上限时，应该优先压哪里，如何保留 headroom？

推理系统不是先“算不动”，而是先“放不下”。这一节讲的，不是模型数学，而是 GPU 显存怎么被占满、为什么会突然炸、工程师怎么提前防止它炸。内存管理做不好，吞吐量、延迟和可靠性都会一起出问题。

5.1 显存主要花在三件事上：权重、KV、临时工作区

可以把 GPU 显存想成一个很贵、很快、但空间有限的工作台。模型推理时，很多东西都必须放在这个工作台上：

1. 模型权重：决定模型能不能驻留在设备上；
2. **KV cache**：随着活跃序列和上下文长度动态增长；
3. 临时工作区与运行时开销：包括激活、attention workspace、allocator 开销、CUDA graph、通信缓冲区、LoRA adapter、sampling buffer 等。

这张工作台上，到底要摆多少东西？会不会中途突然摆不下？在短 prompt、小 batch 场景里，权重常常是第一大头；在长上下文或大 batch 场景里，KV 很快会变成主角。真正让线上最难受的，往往是第三类：它不如前两者显眼，却经常在峰值流量时把“理论能放下”变成“实际会 OOM”。这就是 GPU 内存管理。

5.2 为什么大模型服务必须提前做 memory budget

你不能用“先跑起来再看”来管理显存。显存不是普通软件 bug。普通 bug 也许只影响一个请求。显存管理做不好，会直接导致：OOM，batch 变小，吞吐量下降，TTFT 飙高，p99 尾延迟变差，整个服务抖动。成熟团队不会把“显存够不够”交给线上试错，而会在上线前做基本预算：

- 权重占多少；
- 目标最大 batch、最大上下文、最大输出长度下 KV 占多少；
- 还要预留多少 workspace 和波动空间；
- 高峰时是否会因为 prefix cache、多 LoRA、CUDA graph 或通信缓冲再抬高峰值。

内存预算的目的不只是避免 OOM，更是为了给调度器一个明确边界。没有预算，调度器就无法知道什么时候该拒绝请求、什么时候该降级、什么时候该拆队列。

5.3 pooling、paged blocks 与碎片控制

GPU 内存管理里最怕的不是“总量不够”，而是“总量看似够，但因为碎片和波动就是分配不出来”。这就是为什么现代推理栈几乎都会使用：

- 预分配或池化的内存区域；
- 块级或页级的 KV 分配；
- 尽量稳定的 block size；
- 尽量少做大块搬移和重新分配。

memory pooling 先预留一大块内存，后面尽量在这块池子里反复复用，少做临时申请、释放、重新分配。

paged blocks / paged attention 的思路是：不给每条请求一整块巨大的连续 KV 空间，而是把 KV 切成很多固定大小的小块，哪条请求需要，就拿几个块拼起来。这像什么？像仓库里的标准货架箱。你不再给每个客户定制一个形状奇怪的大柜子，而是统一用标准箱子装货。这样做的好处是：更容易复用，更容易回收，更不容易因为长短请求混合而碎掉，prefix 共享、copy-on-write 也更方便。

这套机制的价值在于，它把“频繁分配和释放”变成“在固定池中做块级复用”。对于动态请求进出、长短上下文混合的在线服务来说，这通常比任何单个 kernel 优化都更能决定系统是否稳定。

5.4 为什么 dynamic 批处理需要 headroom

动态批处理最大的诱惑，是“把 GPU 塞满”。但真正稳的系统反而不会把显存吃到 99%。原因很简单：

- KV cache 是动态增长的；
- 新请求会突然插入；
- 长输出或异常长 prompt 会打破平均值假设；
- 推测解码、guided decoding、multi-LoRA、P/D 迁移都可能带来额外工作区。

所以工程上通常会刻意保留 **memory headroom**。这不是浪费，而是在给抖动留余地。很多 p99 尾延迟问题的根因，并不是“GPU 不够强”，而是系统把它填得太满，任何一点波动都会触发 preemption、swap 或 OOM。

5.5 当显存不够时，优先级应该怎么排

一个经验上很有效的顺序是：

1. 先压输入：删掉无用上下文，收益最好；
2. 再压权重：weight-only quantization 往往先落地；
3. 然后压 **KV**：特别是在长上下文场景；
4. 仍不够时再考虑 **offload**、分离部署、多卡并行；
5. 最后才是换更大的 **GPU** 或更小的模型。

这个顺序体现的是工程原则：先减少浪费，再改变部署形态，最后才是更昂贵的硬件决策。

6 内核与 GPU 优化

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 为什么优化 attention kernel 往往比“再调一点 batch”更值钱？
> 2. FlashAttention、PagedAttention、FlashDecoding 分别解决什么问题？
> 3. kernel fusion 为什么会在 decode 阶段尤其明显？
> 4. tensor parallelism 为什么既能扩模型，也会引入通信代价？
> 5. 推理框架应该按什么标准选，而不是按热度选？

模型结构本身不会自动变成高性能服务。同一个模型，换一种 kernel、换一种 KV 管理方式、换一种并行切分方式，跑出来的吞吐量、尾延迟和显存利用率都可能完全不同。纸面上的架构优势，只有在合适的运行时里被真正实现出来，才会兑现。

很多推理优化不是在改模型“会不会推理”，而是在改数据“怎么在 **GPU** 上流动”。

最常见的浪费有三类：

- 本来可以在片上完成的计算，却反复去 HBM 搬数据；
- 本来连在一起的几个小操作，却拆成很多小 kernel 分开做；
- 本来一张卡能做的事，被切到多张卡后反而花了很多时间在通信上。

6.1 FlashAttention：不改 attention 算法，而是少搬数据

FlashAttention 最容易被误解成“近似 attention”。其实不是。它的核心是 **IO-aware exact attention**：attention 的数学结果不变，变的是 GPU 上的实现方式。普通 attention 往往会生成很大的中间矩阵，再把这些中间结果写回 HBM、再读回来继续算；FlashAttention 则把计算按 tile 组织起来，尽量在更快的片上 SRAM / shared memory 里完成，减少 HBM 和片上缓存之间的往返。

FlashAttention 主要是在减少 HBM 流量。这也是为什么它对 长 prompt 的 prefill 特别值钱。prefill 阶段 attention 的工作量大、中间状态大，只要能少搬几次大块数据，速度就会明显上去，而且更容易把 attention 从“被 IO 拖慢”拉回到“更接近计算受限”的状态。因为它仍然是 exact attention，所以它也比很多近似方法更容易在生产系统里落地。

> **Tip**
>
> 你可以把 FlashAttention 想成：不是换了一种更聪明的注意力，而是把“先摊开整张大草稿纸再算”改成“分块算、边算边归并、尽量别来回搬整页纸”。

6.2 kernel fusion：把本来连着的几步，一次做完

很多模型计算其实天然是连在一起的：

- dequant 之后立刻 matmul；
- norm 之后立刻 linear；
- RoPE 之后立刻做 QKV 投影；
- matmul 之后立刻接激活或残差。

如果这些步骤每做完一次就把中间结果写回 HBM，再读出来做下一步，就会产生两种额外成本：一是 多一次显存读写，二是 多一次 **kernel launch**。TensorRT-LLM 的文档就把 fusion 的价值说得很明确：把多个操作融合成一个 kernel，可以减少 memory movement，也减少多次启动 GPU kernel 的开销。

这件事对 prefill 和 decode 都有帮助，但在 **decode** 阶段尤其明显。原因很简单：decode 每一步只生成很少的新 token，本来计算量就不大。此时，那些“看起来只是几个小 kernel”的额外开销，反而会在单 token 路径上被放大。换句话说，在大工作里，多走两步不显眼；在小工作里，多走两步就会显得很贵。 decode 正是后一种情况。

6.2.1 PagedAttention 与 FlashDecoding：一个管“怎么放”，一个管“怎么算”

这两个名字很像，但解决的不是同一个问题。

PagedAttention 管的是 KV cache 怎么放进显存。它借鉴了操作系统的分页思想：不再要求每条序列的 K/V 必须放在一大段连续显存里，而是把 KV cache 切成固定大小的 block/page，再用映射表把逻辑上的连续历史映射到物理上不连续的显存块。这样做的意义首

先不是“attention 更快算出来”，而是 显存浪费更少、碎片更少、**KV** 更容易共享和复用。vLLM 的论文把它概括为：允许 K/V 存在 non-contiguous paged memory 中，从而把 KV cache 浪费压到接近零，并把同等延迟下的吞吐提升到 2–4×。

FlashDecoding 管的则是 **decode** 阶段的 **attention kernel** 怎么算得更满。decode 最大的尴尬在于：每一步通常只有一个新 query token。这样一来，很多本来适合大矩阵并行的 attention kernel，在 decode 里会突然变得不够“宽”，GPU 很容易吃不满。FlashDecoding 的核心思路，是在 query 长度几乎为 1 的情况下，新增一条并行维度：把历史 K/V 的序列维度切开，让不同 SM 同时处理不同段历史，再把结果正确归并回来。CRFM 对它的描述很直接：它在 decode 阶段沿着 keys/values sequence length 维度并行，从而在小 batch、长上下文时更好地利用 GPU，并在很长序列上带来显著加速。

所以最容易记住的区分是：

- **PagedAttention**：解决 **KV cache** 的存储与分配；
- **FlashDecoding**：解决 **decode attention** 的并行度与利用率。

如果你的场景是 长上下文 + 小到中等 **batch** + **decode** 明显被带宽和历史访问拖住，那这类专用 decode kernels 往往很有价值；而如果你的主要问题是 **KV** 放不下、碎片严重、活跃请求数上不去，那优先级更高的通常是 PagedAttention 一类内存管理优化。

6.3 tensor parallelism：把大矩阵切到多张卡上，但通信成本也会跟着来

当模型单卡放不下，或者某些层太大、单卡算得不够快时，最常见的办法之一就是 **tensor parallelism**（**TP**）。它的基本思路是：把单层里的大矩阵切到多张 GPU 上并行计算，然后在需要的时候做 all-reduce 或 all-gather，把结果重新拼回来。Megatron-LM 论文就是在“GPU 内存容量有限”和“大模型算得太慢”这两个背景下引入张量并行的。

TP 的好处很直接：

- 能把更大的模型放进多卡；
- 能让某些特别大的算子并行得更快。

但它的代价也同样直接：

- 每层都会引入跨卡通信；
- batch 小、decode 路径短的时候，通信成本会更显眼；
- 运维和排障复杂度也会上升。Megatron-LM 明确指出，朴素的模型并行会遇到 expensive cross-node communication 和设备彼此等待的问题。

所以 TP 不是“越多越好”，而是一个很典型的系统交换：** 你用更多 GPU 换来了容量和并行度，同时也把一部分时间换成了同步和通信。

这也是为什么在 **decode** 明显带宽受限、**batch** 又不大的场景里，过度切分反而可能不划算：算子是被拆开了，但你并没有得到足够大的并行工作量去抵消通信。

推理框架不应该按“大家都在用什么”来选，而应该按“你现在最痛的瓶颈是什么”来选。

如果你的主要问题是：

- 长上下文、高并发、**KV cache** 管理，那你更该看 PagedAttention、连续批处理、KV 复用和调度器成熟度；vLLM 的设计重点就明显放在这里。
- **NVIDIA GPU** 上追求极致 **kernel**、**fusion**、多卡并行和编译优化，那你更该看编译器、plugin、runtime 和通信栈；TensorRT-LLM 的官方文档强调的正是 engine 编译、layer fusion、plugins 和 Python/C++ runtime。
- 模型太大、必须多卡切分，那你就必须把通信代价、拓扑和并行策略一起考虑，而不是只看单卡 benchmark；Megatron-LM 论文提供的就是这种思路。

所以，框架选择本质上不是“站队”，而是先认清自己的瓶颈，再选最擅长解决这个瓶颈的运行时。

7 模型压缩

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 模型压缩到底在压什么：权重、激活、KV，还是模型本身？
> 2. 为什么朴素量化常常失败，离群值问题到底是什么？
> 3. AWQ、SmoothQuant、GPTQ、QAT 分别更适合什么路线？
> 4. 剪枝和稀疏性为什么常常“看起来很省，跑起来不一定快”？
> 5. 什么时候蒸馏比继续压一个大模型更合理？

压缩不是某个单一技巧，而是一条部署路线。真正的问题不是“能不能压”，而是“压完以后，业务质量、硬件效率与运维复杂度是否一起变好”。

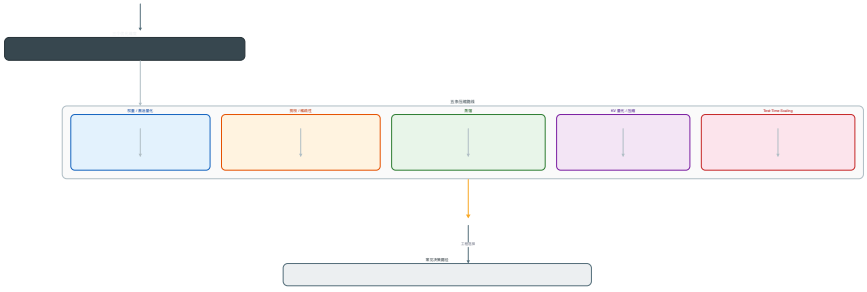

Figure 5: 压缩的 Pareto 前沿

7.1 压缩的四条主要路线

推理语境里的压缩至少包括四类：

1. 量化：减少权重、激活或 KV 的数值精度；
2. 剪枝与稀疏性：让部分参数不再参与计算；

3. 蒸馏：把大模型的行为转移给更小模型；
4. 低秩化与适配器化：降低定制成本，而不是压缩基座本身。

它们面向的问题并不相同：

- 量化优先解决“放不下”和“带宽太贵”；
- 剪枝优先解决“有没有机会减少有效计算”；
- 蒸馏优先解决“模型本身太大”；
- LoRA 类方法优先解决“每个任务都复制一份模型太贵”。

7.2 仅权重量化、激活量化、KV 量化分别在压什么

类型	压缩对象	主要收益	常见场景	主要风险
仅权重量化	模型权重	降低驻留显存与内存	单卡受限、自托管部署	低 bit 时质量可能明显下滑
激活量化	前向中间激活	降低带宽与算子代价	高吞吐推理、硬件友好路径	对离群值敏感，实现更复杂
KV 量化	KV cache	释放长上下文显存，降低解码带宽	长文档、长对话、长 RAG	长距离精确回忆更容易退化

工程上最常见的顺序通常是，先做权重量化，再视业务决定要不要压 KV，激活量化一般出现在更激进、更硬件绑定的优化路径里。

7.3 为什么朴素量化经常失败：离群值问题

LLM 中很多层的激活分布并不均匀。少数通道会出现非常大的 outlier。如果你做很朴素的 per-tensor 量化，这些离群值会把整个量化范围拉得很宽，于是大量普通值都被压缩进很粗的桶里，误差迅速变大。

量化失败往往不会只表现为 perplexity 轻微上升，而会更先在下面这些任务里出现：

- 代码与数学；
- 结构化输出；
- 长上下文检索；
- 需要对 rare tokens 保持稳定的任务。

所以现代 LLM 量化方法，几乎都在尝试回答同一个问题：如何保护重要通道，别让少数离群值毁掉整个范围。

7.4 AWQ、SmoothQuant、GPTQ：三条很重要的 PTQ 路线

AWQ 的核心是“基于激活统计来保护重要权重通道”，因此很适合 4-bit 的 weight-only 量化。它通常在本地推理、单机 GPU 与边缘设备场景里非常有吸引力。

SmoothQuant 的思路是把“激活难量化”的问题部分迁移到权重上，通过等价变换平滑 activation outliers，让 W8A8 这样的硬件友好路径更可行。它更适合追求较完整 INT8 推理链路的场景。

GPTQ 是经典的一次性 PTQ 路线，利用近似二阶信息做更精细的权重量化补偿。它对 3/4-bit weight-only 路线非常有代表性，适合离线校准后部署。

三者都属于 PTQ，但它们回答的是不同问题：

- 想把大模型安全地压到低 bit，本地或单机部署很常见：先看 **AWQ / GPTQ**；
- 想走更标准的 W8A8 硬件路径：先看 **SmoothQuant**；
- 想在部署成本最低的前提下快速落地：**weight-only** 往往最先上线。

7.5 PTQ 与 QAT：什么时候训练成本值得付

PTQ(Post-Training Quantization)的优点是快、便宜、部署友好。缺点是 bitwidth 很低时，质量不一定稳。

QAT（Quantization-Aware Training） 则是在训练或微调阶段就把量化误差纳入前向过程，让模型学会适应量化噪声。它的好处是：

- 极低 bit 时通常更稳；
- 更容易追回 PTQ 的质量损失；
- 对结构化输出、长上下文和某些高价值任务更容易保底。

它的代价也很清楚：需要训练资源，需要更长的迭代周期，也会让部署路径更绑定特定硬件或 dtype。

因此 QAT 通常在下面这些条件同时满足时才真的值得：

- PTQ 已经影响上线质量；
- 模型会被长期、大规模部署；
- 节省下来的推理成本足够摊薄训练开销；
- 你真的需要那一级别的压缩，而不是换个更小模型就能解决。

7.6 剪枝与稀疏性：为什么“变稀疏”不等于“变快”

剪枝大致可以分成：

- 非结构化剪枝：任意把权重设零；
- 结构化剪枝 / **N:M sparsity**：例如 2:4；
- 架构稀疏性：例如 MoE，让每个 token 只激活一部分参数。

问题在于，硬件和 kernel 通常只能高效利用一部分稀疏模式。于是就出现一个常见的工程误区：论文里 sparsity 很高，线上却并没有明显加速。

更麻烦的是，剪枝常常存在“质量悬崖”。在某个 sparsity 之前看起来几乎没事，一旦越过阈值，质量突然塌掉。因为被伤到的可能不是平均冗余，而是关键电路、关键 head 或关键通道。

所以剪枝是否值得，最终要回答两个问题：

1. 你的硬件能不能真正吃到这种稀疏性；
2. 你的业务能不能承受这种非线性退化风险。

7.7 蒸馏：有时候真正该压的是“模型选择”，不是位宽

如果你真正需要的是更低首字时间、更低生成速度、更低美元成本，蒸馏往往比在一个巨模型上继续抠低 bit 更彻底。它的本质是：把大模型的行为迁移到一个更小、更便宜的学生模型。

常见路线包括：

- 响应蒸馏：学教师输出；
- **logit** 蒸馏：学教师分布；
- 轨迹蒸馏：学中间推理或工具使用行为。

一个很实用的工程判断是：如果你的请求量大、模式稳定、错误类型重复，而且模型大小本身已经成为主要成本来源，那么蒸馏往往比继续给大模型做更多 test-time tricks 更值。

7.8 GGUF 与本地推理：它更像交付生态，而不只是某种算法

GGUF 更准确的理解，不是一种单独量化算法，而是本地推理生态里非常重要的模型封装与分发格式。它常和 llama.cpp、CPU/Mac/边缘设备生态一起出现。工程意义在于：

- 分发简单；
- 本地和离线部署友好；
- 与 CPU / Apple Silicon 场景贴合。

因此当有人问“GPTQ、AWQ、GGUF 怎么选”时，更工程化的回答是：

- GPTQ / AWQ 更像量化方法；
- GGUF 更像交付格式与运行时生态；
- 它们解决的层次并不完全相同。

8 推理时计算扩展（test-time scaling）

> **!** Important
>
> 本节先回答几个关键问题：
> 1. test-time scaling 与推测解码有什么本质区别？
> 2. temperature、top-p、beam search、自一致性、best-of-N 分别适合什么问题？
> 3. 什么时候值得在推理时加算力，什么时候更该回到训练或蒸馏？
> 4. 为什么 verifier 或 PRM 的存在会改变你是否应该做搜索？

不是所有推理优化都在追求更快。有一类优化专门在推理时增加计算，以换取更高的可靠性或更高的答案质量。

8.1 什么是 test-time scaling

test-time scaling 指的是：不重新训练模型，而是在推理时花更多计算来提高答案质量。这和推测解码完全不同。

- 推测解码目标是更快，理想情况下不改变目标分布；

- test-time scaling 目标是更准，通常明确增加推理成本。

当请求价值高、错误成本高、又存在可靠 verifier 时，test-time scaling 往往比盲目换更大模型更划算。

8.2 采样策略：先决定你是在要“多样性”还是“稳定性”

很多系统问题并不需要复杂搜索，先把采样策略想清楚就能解决一半。

- 温度（**temperature**）：越低越确定，越高越多样；
- **top-k / top-p**：控制候选 token 空间；
- **beam search**：保留多个高概率前缀并持续扩展；
- **best-of-N**：采样多个完整候选再做选择；
- **self-consistency**：对多条推理路径做投票；
- **critique-revise**：先生成初稿，再批评修订。

> **Note**
>
> **采样方法**
> 模型每一步都只是在下一 token 的分布里做选择：
>
> $$p_t(v) = P(y_t = v \mid x, y_{<t}) = \frac{\exp(z_t(v))}{\sum_{u \in V} \exp(z_t(u))}$$
>
> 所以这些策略本质上只分三类：改分布形状、截断候选集合、多条路径再选。
>
> **Temperature**
> 改的是分布有多尖：
>
> $$p_t^{(T)}(v) = \frac{\exp(z_t(v)/T)}{\sum_u \exp(z_t(u)/T)}$$
>
> - $T < 1$：更稳，概率集中在头部 token；
> - $T > 1$：更散，概率摊到更多 token。
>
> **Top-k / Top-p**
> - **top-k**：只留前 k 个 token——硬截断。
> - **top-p**：只留累计概率达到 p 的那一团 token——更灵活。
>
> **Beam Search**
> 在生成时同时保留多条高分前缀，适合翻译、ASR、摘要等低熵任务（高概率序列通常真的是好序列）。但在开放对话里，高概率常常只意味着平庸。
>
> **Best-of-N**
> 先生成 N 个完整答案，再选最好的——“写完再选”，而不是”边生成边筛”：
>
> $$\hat{y} = \arg\max_i \; s(x, y^{(i)})$$
>
> **Self-consistency**
> 采样多条推理路径，对最终答案做投票：

$$\hat{a} = \arg\max_{a} \sum_{i=1}^{N} \mathbf{1}[a^{(i)} = a]$$

适合推理题，不适合 JSON——JSON 的问题不是”哪条思路更一致”，而是”别生成错格式”。

Critique-revise

初稿 → 批评 → 修订稿。适合长答案、分析、方案——因为第一稿的常见问题不是完全错，而是不完整、不锋利、不够严密。

选采样方法不是看哪种方法更高级，先问一件更实际的事：这个任务，更怕错，还是更怕平？工程上一个常见误区是把这些方法当成“越复杂越好”的等级关系。其实不是。对于工具参数、JSON、分类标签这类低熵任务，低温或 guided decoding 往往最有效。对于数学、推理或方案探索，自一致性、best-of-N 或 critique-revise 才更有价值。beam search 在机器翻译、语音、摘要等相对低熵、分数函数更稳定的任务上很有用，但在开放式对话里未必是最好选择，因为它容易把输出推向高概率却平庸的模式。

Table 4: 怕错 vs 怕平：采样策略速查

场景	核心诉求	推荐策略
JSON、工具参数、分类标签	稳定性——怕错	低温 / guided decoding / 小候选集
数学、推理、方案、写作	多样性——怕平	top-p / best-of-N / self-consistency / critique-revise

8.3 什么时候该在推理时多花算力

优先考虑 test-time scaling 的情况通常是：

- 请求量不算极大，但单次价值高；
- 错误主要集中在难例，而不是整体普遍偏差；
- 你有 verifier、judge 或业务规则可以区分好坏；
- 延迟预算允许疑难样本走慢路径。

相反，如果下面这些条件成立，就更应该回到训练、蒸馏或模型切换：

- 请求量极大，额外推理成本摊不开；
- 失败模式高度重复；
- 没有可靠的评判器；
- 所有请求都要求稳定改进，而不是只救高价值样本。

因此，一个成熟系统往往会采用分层路径：简单请求走单次推理，难请求或高价值请求再进入更昂贵的 **test-time scaling**。

8.4 PRM 与搜索：当你能评价中间状态时，搜索就更划算

PRM 本质上是在给中间状态打分，而不是只给最终结果打分。比如：这一步分解问题对不对；这个中间结论有没有推进到正确方向；这个子问题是不是已经偏了；这段 reasoning 是在收敛，还是在绕远路。这让搜索从“盲搜”变成“有启发的搜”，更早发现坏路径，更少把计算浪费在明显错误的分支上，能把算力集中到更有希望的轨迹上。

- 只看最终答案时，best-of-N 和投票更简单；
- 能评价中间步骤时，树搜索、回溯和剪枝才真正有依据。

这也对应一个很实用的广度—深度权衡：

- 当不确定性主要在“方向选不对”，更值得增加广度；
- 当一旦方向选对，后面需要长链推理，更值得增加深度。

9 指标、监控与质量门控

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 推理服务最该监控哪些指标，哪些只是好看但不够用？
> 2. 为什么只看平均 latency 几乎一定会误判线上状态？
> 3. 压缩或量化上线前，应该怎样设质量门控？
> 4. 当用户说“有时快、有时慢”时，怎样通过切片定位根因？

推理系统的稳定性来自可观测性。没有指标、日志与切片，你根本不知道是在优化模型、调度器、缓存，还是在修一个并不存在的问题。

9.1 四个核心指标，以及为什么它们还不够

最常用的四个核心指标是：

1. 首字时间；
2. 生成速度 / **TPOT**；
3. 吞吐量；
4. 成本，例如每百万 token 成本或单请求成本。

但生产里只看这四个远远不够。你通常还需要：

- p95 / p99 首字时间；
- p95 / p99 生成速度；
- queue depth 与 request age；
- prefix cache hit rate；
- speculative acceptance rate；
- OOM / preemption / eviction 次数；
- JSON / tool call 成功率；
- 长上下文任务的引用与检索准确率。

平均值之所以不够，是因为用户真正感知的是尾部，而不是均值。

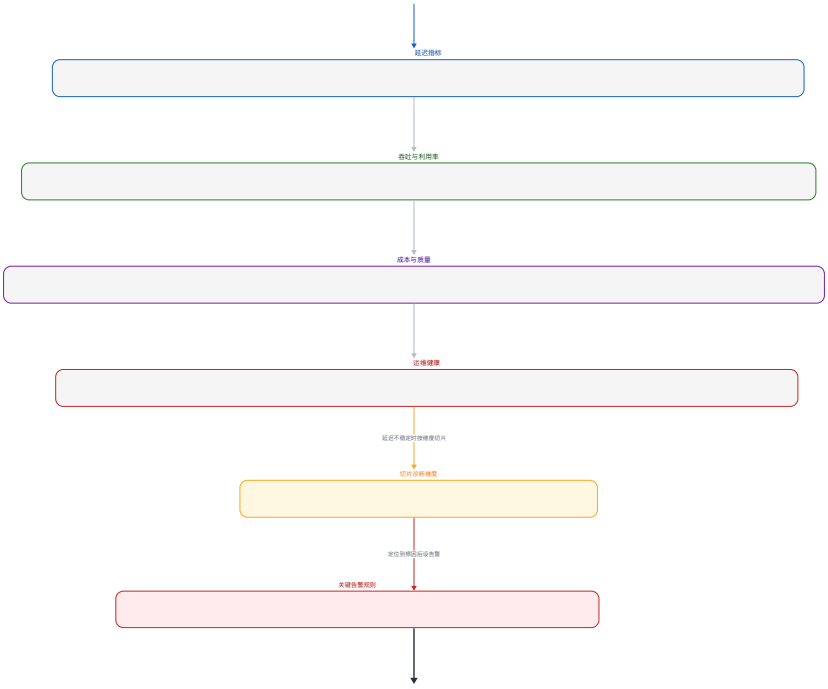

Figure 6: 推理指标仪表盘

9.2 成本应该拆解到请求路径，而不是只算模型单价

推理成本通常来自三部分：

- 模型 token 成本；
- 工具与检索调用成本；
- 重试、重排、test-time scaling、多轮探索带来的放大成本。

对服务团队来说，更有价值的不是“这个模型每百万 token 多少钱”，而是：

- 哪类请求最贵；
- 贵在 prefill 还是解码；
- 贵在长上下文还是多轮重试；
- 贵在大模型本身还是外围的检索与结构化路径。

只有把成本拆到请求路径，你才能真正知道该优化什么。

9.3 量化后的质量门控应该怎么设

压缩上线前最容易犯的错，是只看一个平均 benchmark 分数。更可靠的门控方式应至少分成四层：

1. 基础能力：perplexity、常规 benchmark；
2. 业务主任务：问答、代码、摘要、客服等；
3. 结构化输出：JSON 合法率、schema 命中率；
4. 长上下文与安全：Needle、跨段引用、拒答边界、越权工具。

门控策略通常是：

- 先设硬门槛，关键任务不允许掉；
- 再设加权分，允许非关键指标小幅回退；
- 最后上 shadow traffic 或 canary，看真实线上分布。

9.4 为什么大海捞针（Needle-in-a-Haystack）常常先掉

如果你量化后发现长文检索或 Needle 指标先下降，这通常不意外。因为这类测试高度依赖：

- 长距离表征不被噪声淹没；
- attention 的细微差别仍然可分；
- rare positions 和 rare tokens 的数值精度足够稳定。

而量化——特别是激进低 bit 和 KV 量化——恰好最容易伤害这些能力。因此，如果你的业务高度依赖长文精确定位，量化评估就不能只看对话质量。

9.5 当用户说“延迟不稳定”时，如何切片定位

“有时很快，有时慢 5 倍”通常不是单一问题。最有效的诊断办法，是把请求按下面这些维度切片：

- prompt 长度桶；
- output 长度桶；

- cache hit / miss ;
- guided / non-guided ;
- speculative on / off ;
- tenant / adapter / 模型版本 ;
- prefill 池 /解码池。

切片后，你往往会看到真正的根因：是长 prompt 混队、是 prefix cache 失配、是冷 adapter 加载、还是 autoscaler 没跟上。

10 工程案例

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 线上推理系统最常见的故障模式有哪些？
> 2. 当 KV cache、调度器、量化或解码配置出问题时，症状通常长什么样？
> 3. 这些事故背后真正值得记住的工程教训是什么？

线上问题往往不会以“某个理论错了”的形式出现，而是以用户投诉、成本异常、**p99** 爆炸或质量漂移的形式出现。下面这些案例的意义，不在于故事本身，而在于让你形成可迁移的故障直觉。

10.1 KV cache 爆炸，系统明明没满载却频繁 OOM

症状：

- GPU 利用率不算高，但高峰时频繁 OOM ;
- 长会话用户的失败率明显更高 ;
- 扩容后问题缓解不明显。

根因：

- 团队只按“平均 prompt 长度”估算容量，没有按最大活跃序列和最大输出长度预算 ;
- prefix cache、多轮会话历史和长输出共同把 KV 拉高 ;
- 没有预留 headroom，动态 batch 一抖就越界。

工程教训：

- KV cache 不是背景成本，而是在线状态主成本 ;
- 容量规划必须按最坏路径做预算 ;
- 长上下文产品上线前一定要先做 memory envelope 测试。

10.2 连续批处理把平均吞吐做上去了，p99 却炸了

症状：

- 平均 tokens/s 提升 ;
- 用户持续抱怨首字忽快忽慢 ;

- p99 首字时间在流量尖峰时突然放大。

根因：

- 长 prompt 与短 prompt 混在同一队列；
- admission control 太弱；
- 没开 chunked prefill，少数长 prefill 抢占了整个 prefill 通道。

工程教训：

- 在线系统不该只追平均吞吐；
- 公平调度和 SLA 分层经常比更大的 batch 更值钱；
- 没有分桶和准入控制的 continuous 批处理很容易把局部最优变成全局最差。

10.3 量化后聊天看着还行，RAG 质量却悄悄塌了

症状：

- 通用聊天和摘要几乎无感；
- 长文问答中引用开始错位，跨段检索更容易漏；
- Needle 类测试回归明显。

根因：

- 团队只看了常规 benchmark 与人工闲聊体验；
- 压的是 KV 或过低 bit 的路径，最先伤到长距离注意力；
- 没有把长上下文精确引用纳入门控。

工程教训：

- 量化评估必须和真实任务对齐；
- 长上下文不是“加一个 benchmark 就算覆盖了”，而是要测试引用、定位、实体绑定等细粒度能力；
- 先坏的通常不是平均能力，而是最脆弱的边界能力。

10.4 结构化输出不稳定，重试把延迟和成本一起打爆

症状：

- JSON 解析失败率不高但足够烦；
- 每次失败都触发应用层重试；
- 平均延迟和 token 成本随流量放大。

根因：

- 只靠 prompt 要求“输出合法 JSON”；
- 没有 guided decoding 或强 schema 校验；
- 重试策略过于宽松，把小概率错误放大成系统成本。

工程教训：

- 结构化输出问题，优先想采样阶段如何避免非法 token，而不是事后多试几次；
- 重试不是免费的，它会把所有边缘错误放大成整体成本问题；

- 在 agent 或工具场景里，输出合法性本身就是性能问题。

10.5 P/D 分离后首字更稳了，但尾部反而更抖

症状：

- 平均首字时间改善；
- 部分请求的生成速度反而变差；
- 集群监控显示解码池偶发性空转或等待。

根因：

- prefill 与解码分离后，KV 传输和连接器成了新瓶颈；
- 对 tail latency 的优化边界变成了跨机状态迁移问题；
- 没有充分评估 KV 迁移成本与网络抖动。

工程教训：

- P/D 分离是 SLA 工具，不是免费吞吐量；
- 状态迁移本身会变成系统问题；
- 在做 disaggregation 之前，应该先问 chunked prefill 是否已经足够。

11 本章小结

本章真正想建立的，不是某个框架的使用方法，而是一套判断推理系统的工程视角。

- **prefill** 与解码不是同一种负载。
- **KV cache** 是服务系统最关键的在线状态。
- 吞吐量、延迟、显存、成本与可靠性是联动的，不可能单点最优。
- 压缩与内核优化只有在运行时真正吃到时才算收益。
- 线上系统的胜负通常不在平均值，而在 **tail latency**、失败模式与可观测性。

从这个角度看，推理并不是训练之后的收尾工作。它本身就是现代 AI 系统最核心的一层生产工程。

11.1 推理框架怎么选：按你的瓶颈选，不按流行度选

把常见开源栈放在系统问题上看，更容易做判断：

框架	更强的点	更适合	需要注意
vLLM	PagedAttention、连续批处理、APC、chunked prefill、structured outputs、LoRA	通用开源模型服务的默认答案	运行时强，但 admission control、隔离、SLA 分层仍需自己设计

框架	更强的点	更适合	需要注意
TensorRT-LLM	NVIDIA 栈深度优化、IFB、paged KV、spec 解码、guided decoding、多节点并行	明确跑在 NVIDIA 体系上，追求极限性能	工程门槛更高，硬件绑定更强
SGLang	RadixAttention、结构化输出、PD disaggregation、多 LoRA、量化与 KV cache 选项丰富	agent、结构化输出、复杂前缀复用场景	版本迭代快，需要持续跟进
TGI	Hugging Face 生态集成、部署体验成熟、监控与流式支持好	已经深度依赖 HF 生态的团队	当前更像维护型方案，新项目通常会同时评估 vLLM 与 SGLang

一个比“谁最快”更实用的选择标准是：

- 你的主要瓶颈是首字时间、生成速度、吞吐、结构化输出，还是多租户与隔离？
- 你是否明确绑定 NVIDIA 栈？
- 你需要的是默认性能，还是更强的可编排性和 cache-aware 特性？
- 团队能否承担自定义 kernel、分布式调试和升级节奏？

11.2 问题小结

1. 什么是语言模型的推理？
把用户输入转成 token，经过 prefill 建立上下文表示与 KV cache，再通过解码自回归地产生输出 token，并把结果返回给调用方。

2. 为什么 **LLM** 推理昂贵？
因为你不仅要把巨大的模型权重驻留在硬件上，还要在在线流量下持续读取权重和 KV cache；随着上下文、活跃序列数和输出长度增长，显存、带宽和调度都会迅速成为瓶颈。

3. **prefill** 与解码的区别是什么？
prefill 一次处理整段 prompt，并行度高、算术强度高；解码每步只新增一个 token，却要读取全历史 KV，串行且更常受带宽限制。

4. 什么是 **KV cache**？
它是把历史 token 的 K/V 按层缓存起来，避免解码时每步重复计算整段历史，是 decoder-only 服务的核心在线状态。

5. 为什么上下文长度影响延迟？
因为更长的 prompt 会增加 prefill 计算量，更长的历史会增加解码每步要读取的 KV 大小，同时还会额外占用显存。

6. 为什么 **attention** 复杂度会迅速上升？
因为 token 之间的交互随序列长度增长而增加；即使推理里用 KV cache 避免了重复算

K/V，解码每步也仍要读取与历史长度近似成正比的缓存。

7. KV caching 如何加速生成？
它把“每步重算整段历史”变成“只算新 token，然后读取历史缓存”，从重复计算问题变成状态读写问题。

8. 什么是批处理，为什么重要？
它让多个请求共享一次前向计算，从而摊薄权重读取和 kernel 开销，是在线服务获取吞吐量的核心手段。但太激进会伤害单请求延迟。

9. 推理系统如何管理 GPU 内存？
先做权重与 KV 的容量预算，再通过内存池、页式 KV 分配、headroom 控制、chunked prefill、量化和 admission control 把动态请求稳定地约束在预算内。

10. 为什么解码经常是体感瓶颈？
因为它是逐 token 串行路径，不能像 prefill 那样充分并行，而且每一步都要读取权重和历史 KV，容易被带宽拖住。

11. 前缀缓存（prefix caching）何时收益最大？
当大量请求共享相同系统提示、模板、文档前缀或长历史时，prefix cache 可以跳过共享部分的 prefill，直接降低首字时间。

12. 什么时候推测解码值得用？
当目标模型明显被解码卡住、草稿模型足够便宜且接受率够高时值得；如果接受率低或系统主要瓶颈不在解码，它可能得不偿失。

13. 为什么量化必须和硬件一起看？
因为不同量化路径是否真正带来速度或容量收益，取决于内核支持、数据布局和硬件执行路径；纸面 bitwidth 降低并不自动等于线上更快。

14. 如何设计一个可扩展的 LLM 推理服务？
先按业务定义 latency、throughput、上下文长度、结构化输出和隔离需求；再围绕请求队列、调度器、KV 管理、缓存、解码、监控和降级路径搭系统，而不是只选一个模型。

15. 支撑百万级并发的关键是什么？
不是单点极致性能，而是队列分层、cache-aware 路由、连续批处理、前缀复用、显存预算、自动扩缩容和必要时的分离部署。

16. 怎样平衡 throughput 与 latency？
通过分桶、优先级队列、限制 batched tokens、chunked prefill 和 headroom 控制，让系统在高吞吐时仍能保住 p95/p99，而不是只追平均值。

17. 哪些技术最有效降低成本？
视场景而定：短上下文常先看权重量化和批处理，长上下文常先看 KV 管理与输入压缩；长期高流量任务则更该考虑蒸馏或模型切换。

18. 什么时候压缩，什么时候换更小模型？
如果业务必须保留当前模型能力且只是部署受限，先压缩；如果模型本身就明显超配、流量很大且任务模式稳定，换更小模型或蒸馏通常更划算。

12 参考资料

1. NVIDIA. *H100 Tensor Core GPU Datasheet*. 2023.

2. Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022.

3. Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023.

4. Ainslie et al. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023.

5. vLLM Project. *vLLM Documentation (Automatic Prefix Caching, Chunked Prefill, Structured Outputs, Disaggregated Prefilling)*. 2024.

6. NVIDIA. *TensorRT-LLM Documentation (In-Flight* 批处理*, Paged KV Cache, Chunked Prefill, Guided Decoding,* 推测解码*)*. 2024.

7. SGLang Team. *SGLang Documentation (RadixAttention, Structured Outputs, Quantized KV Cache, LoRA Serving, PD Disaggregation)*. 2024.

8. Hugging Face. *Text Generation Inference Documentation (Continuous* 批处理*, Paged Attention, Guidance, Monitoring)*. 2024.

9. Chen et al. *Accelerating Large Language Model Decoding with Speculative Sampling*. 2023.

10. Cai et al. *Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads*. 2024.

11. Wang et al. *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. 2023.

12. Zhang et al. *H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models*. 2023.

13. Xiao et al. *Efficient Streaming Language Models with Attention Sinks*. 2023.

14. Li et al. *SnapKV: LLM Knows What You Are Looking For Before Generation*. 2024.

15. Liu et al. *KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache*. 2024.

16. Hooper et al. *KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization*. 2024.

17. Lin et al. *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. 2024.

18. Xiao et al. *SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models*. 2023.

19. Frantar et al. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. 2023.

20. PyTorch Blog. *Quantization-Aware Training for Large Language Models with PyTorch*. 2024.

21. Frantar & Alistarh. *SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot*. 2023.