AI Infra M3 推理优化学习笔记：从 KV Cache 账本到生产级推理系统

数据来源：本笔记的章节骨架、术语体系、KV Cache 账本口径、调度器流程、量化与投机解码的公式，取自 AIInfraGuide「模块四：推理优化」的 29 个公开页面（第 1～10 章正文与章节扉页），访问日期 **2026-11**。页面前缀：`https://caomaolufei.github.io/AIInfraGuide/inference/模块四-推理优化/`。其中 24 篇是长篇正文（1.1、1.2、2.1～2.5、3.1～3.6、4.1～4.6、5.1～5.5、10.1）；第 6～9 章的 4 个页面是章节扉页（只给出章节目录与要点，正文未公开），这四章按公开论文与官方文档重建，逐处标注。

补充来源：Kwon et al. *PagedAttention*（SOSP 2023）、Yu et al. *Orca*（OSDI 2022）、Agrawal et al. *SARATHI*（2023）、Zheng et al. *SGLang*（RadixAttention）、Dao et al. *FlashAttention* 系列、Frantar et al. *GPTQ*（ICLR 2023）、Lin et al. *AWQ*（MLSys 2024）、Frantar et al. *Marlin*（2024）、Xiao et al. *SmoothQuant*（ICML 2023）、Liu et al. *KIVI*（ICML 2024）、Micikevicius et al. *FP8 Formats*（2022）、OCP *MX Specification v1.0*、Leviathan et al. *Fast Inference from Transformers via Speculative Decoding*（ICML 2023）、Chen et al. *Speculative Sampling*（2023）、Cai et al. *Medusa*、Li et al. *EAGLE / EAGLE-2 / EAGLE-3*、DeepSeek-AI *DeepSeek-V3 Technical Report*、Zhong et al. *DistServe*（OSDI 2024）、Patel et al. *Splitwise*（ISCA 2024）、vLLM 官方文档（v0.32.0）、NVIDIA CUDA Graph 与 NVFP4 文档、Ray Serve 与 Kubernetes 文档、MLPerf Inference 规则。

标注约定：`【补充】`＝AIInfraGuide 原文没有、由权威来源或通用工程知识补齐；`【推断】`＝基于材料的推理性结论；`【量级】`＝只保证数量级正确；`【待实测】`＝必须在你自己的设备与负载上测。

版权声明：AIInfraGuide 原文版权归原作者所有；本笔记为个人学习用途的二次整理、重组与批注，不替代原文，引用处尽量保留其表述与限定条件。论文与各框架文档版权归各自权利人所有。若本文与原文或官方文档冲突，以原文与官方文档为准。本笔记不做 git 操作、不生成 PDF、不改动仓库其他文件。

一条必须先立下的口径：任何「快了多少倍」都要先问三件事——**口径**（TTFT / TPOT / 端到端 / 系统吞吐 / Goodput）、**基线**（哪个引擎、哪个版本、什么 Batch、什么长度分布）、**硬件与精度**（卡型、显存、FP16 还是量化）。三者任一不同，倍数就不可比。原文对此非常克制：涉及具体性能数字几乎都附带「在该模型、该部署条件下」的限定。本笔记延续这个习惯——**能算的给公式，不能算的给量级，绝不给伪造的实测值**。

1.1 自回归：串行依赖是所有延迟问题的总根源
1.2 两个阶段：Prefill 读题，Decode 答题
1.3 算术强度与 Roofline：为什么 Decode 打不满算力
1.4 KV Cache：用显存换计算
1.4.1 显存账本
1.4.2 GQA 为什么能省四分之三
1.4.3 碎片化：PagedAttention 的动机
1.5 指标体系：如何衡量「快」
1.5.1 尾延迟与 Goodput

1.5.2 指标互相耦合
1.6 常见误解
2.1 五节技术一览
2.2 PagedAttention：像操作系统管理内存一样管理 KV Cache
2.2.1 连续分配的两宗罪
2.2.2 核心思想：把虚拟内存分页搬到 GPU 上
2.2.3 Block Table 怎么工作
2.2.4 如何同时消灭两种碎片
2.2.5 Kernel 视角：分散的块怎么被读出来
2.2.6 内存共享与 Copy-on-Write
2.2.7 抢占与恢复
2.2.8 工程参数
2.3 Continuous Batching：让 GPU 不再空等
2.3.1 为什么批处理有效
2.3.2 Static Batching 的致命短板
2.3.3 迭代级调度
2.3.4 与 PagedAttention 的协同
2.3.5 V1 调度器源码骨架
2.3.6 三组权衡
2.4 Prefix Cache 与 RadixAttention
2.4.1 重复的前缀有多浪费
2.4.2 物理基础：块级只读共享
2.4.3 vLLM 的 Automatic Prefix Caching
2.4.4 淘汰：LRU + 引用计数
2.4.5 SGLang 的 RadixAttention
2.4.6 适用场景与 Prompt 工程
2.5 Chunked Prefill 与统一 Token Budget
2.5.1 Prefill 怎么拖慢 Decode
2.5.2 把长 Prefill 切块
2.5.3 Token Budget：调度的统一货币
2.5.4 V1 的统一调度器
2.5.5 预算截断即切块
2.5.6 参数与调优
2.6 Attention 后端与图优化：把 Decode 的 CPU 开销榨干
2.6.1 反直觉的瓶颈：CPU 拖了 GPU 的后腿
2.6.2 可插拔的 Attention 后端
2.6.3 后端必须支持混合批次
2.6.4 CUDA Graph
2.6.5 torch.compile 与 Piecewise CUDA Graph
2.6.6 权衡
2.7 本章小结
3.1 目录与四层架构
3.2 Config 与一次 generate
3.3 模型与核心算子
3.4 KV Cache 与 BlockManager
3.4.1 分配、追加、回收
3.4.2 前缀缓存：哈希链 + 三条规则
3.5 Scheduler 与连续批处理
3.6 ModelRunner、CUDA Graph 与基准
3.6.1 张量化
3.6.2 KV Cache 怎么切
3.6.3 CUDA Graph：只为 decode 捕获
3.7 本章小结
4.1 量化基础
4.1.1 三类量化对象
4.1.2 仿射量化
4.1.3 位宽与粒度
4.1.4 校准与 PTQ/QAT
4.2 W8A8 与 SmoothQuant
4.2.1 W8A8 难在哪里
4.2.2 等价缩放
4.2.3 迁移强度 α
4.2.4 从变换到 INT8 GEMM
4.2.5 收益与边界
4.3 Weight-only INT4
4.3.1 W4A16 的存储与计算
4.3.2 GPTQ：边量化边补偿
4.3.3 AWQ：按激活找要保护的通道
4.3.4 Marlin：让解包不拖慢计算
4.3.5 vLLM 的 Kernel 选择
4.3.6 公平比较 GPTQ 与 AWQ
4.4 KV Cache 量化

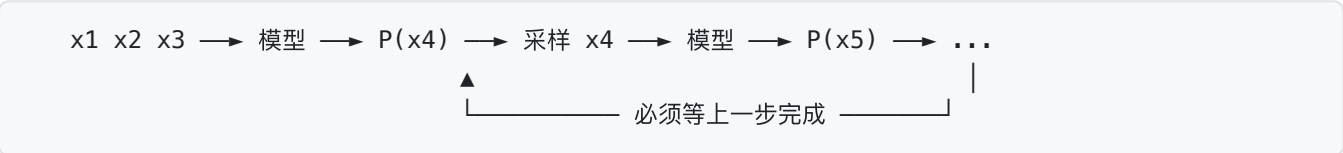
4.4.1 先估算能省多少
4.4.2 为什么不能照搬权重量化
4.4.3 KIVI：K 按通道，V 按 Token
4.4.4 vLLM 的 FP8 KV Cache
4.5 FP8 与 NVFP4/MXFP4
4.5.1 同一串 bit，两种含义
4.5.2 FP8 仍需要 Scale
4.5.3 FP4 与微块缩放
4.5.4 NVFP4 与 MXFP4
4.5.5 硬件支持怎么变成收益
4.6 量化选型与 vLLM 实战
4.6.1 从瓶颈出发
4.6.2 对照实验的最小骨架
4.7 本章小结
5.1 核心原理：Draft + Verify
5.1.1 串行瓶颈与两套分布
5.1.2 接受规则
5.1.3 正确性证明
5.1.4 只给 token 不给概率
5.1.5 推广到整段序列
5.1.6 KV Cache：草稿不是已提交的历史
5.1.7 「无损」保证什么
5.2 Draft 与无模型方案
5.2.1 好 Draft 的标准
5.2.2 Tokenizer 与上下文
5.2.3 N-gram
5.2.4 Suffix Decoding
5.2.5 接受率的口径
5.3 Medusa、EAGLE 与 MTP
5.4 收益边界：接受率高，为什么仍可能变慢
5.4.1 性能账本
5.4.2 投机长度的收益递减
5.4.3 任务标签不是性能结论
5.4.4 高并发、长上下文与抢占
5.4.5 与量化叠加
5.4.6 调度与流式
5.5 vLLM 实战
5.5.1 逐请求接受统计
5.5.2 压测与结果解读
5.6 本章小结
6.1 四种并行
6.2 多节点与 Ray
6.3 选择顺序与验证
7.1 混合 Batching 的三层干扰
7.2 解耦架构
7.3 DistServe 与 Splitwise
7.4 KV 传输：解耦最核心的工程挑战
7.5 Goodput 与 P/D 配比
7.6 什么时候不该用
7.6.1 概念验证（单机可做）
8.1 指标体系
8.2 压测工具
8.3 性能分析工具
8.4 权威基准与回归门禁
9.1 容器化与 K8s
9.2 可观测性
9.3 扩缩容与负载均衡
9.4 容量规划
10.1 四类约束
10.2 从模型到 App
10.3 四个抓手
10.4 可信的 Benchmark
10.5 ExecuTorch 最小实例
10.6 排查顺序
11.1 六张映射表
11.2 三句「迁移直觉」
11.3 一句话总表
12.1 一页纸公式
12.2 自测题（10 题）
12.3 答案
12.4 复习路线
13.1 一手来源（AIInfraGuide 模块四，访问日期 2026-11）
13.2 补充来源
13.3 关键数字溯源
13.4 三处最不确定与两处时效风险

第 1 章 LLM 推理基础：两阶段、显存账本与指标体系

> 对应 AIInfraGuide 模块四 第 1 章（`11-llm推理基础` 正文 + `12-vllm快速入门`）。本章回答四个问题：推理为什么被切成两个阶段、KV Cache 吃多少显存、用什么指标衡量「快」、为什么几万块的 GPU 在生成文字时算力大量闲置。

1.1 自回归：串行依赖是所有延迟问题的总根源

Decoder-only 模型（GPT、Llama、Qwen、DeepSeek）的生成只有一句话：给定已确定的 `x_1 … x_t`，输出下一个 token 的分布，采样得到 `x_{t+1}`，拼回序列末尾，继续。



这个「必须等上一步」的依赖，是后面所有延迟指标、调度技巧、投机解码的共同出发点。**推理优化的本质，是在不破坏这个语义的前提下，把每一步的硬件利用率榨干，或者减少「步数」。**

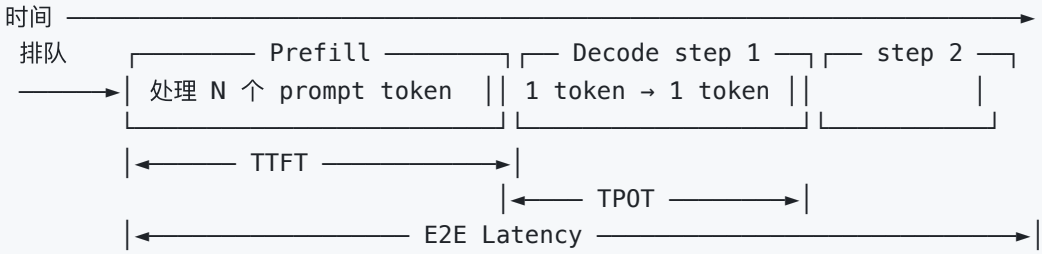
1.2 两个阶段：Prefill 读题，Decode 答题

- **Prefill（预填充）**：把整段 Prompt（几十到几千 token）**一次性并行**送进网络，产出第一个输出 token 以及这段 Prompt 的全部 KV。矩阵乘又大又密，算力被喂饱。
- **Decode（解码）**：从第二个 token 开始，每步只输入**上一步刚生成的 1 个 token**，跑一遍完整网络。生成 200 个 token 就要串行跑 200 次。

原文的类比：Prefill 是读题，可以一眼扫完；Decode 是答题，只能一笔一画。

维度	Prefill	Decode
每步输入 token 数	整段 Prompt（几十～几千）	1 个（每请求）
并行度	高，token 间可并行	低，步与步串行
主要瓶颈	Compute Bound（算力）	Memory Bound（显存带宽）
决定的指标	TTFT	TPOT / ITL
执行次数	1 次（可被 Chunked Prefill 切成多步）	输出长度 − 1 次
算术强度	高，随 batch 内 token 数上升	约 1 FLOP/Byte

维度	Prefill	Decode
优化手段	切块、前缀缓存、大 Token Budget	Batching、Weight-only 量化、CUDA Graph、投机



1.3 算术强度与 Roofline：为什么 Decode 打不满算力

GPU 干活要同时搬数据（HBM）和算数据，谁慢谁就是瓶颈。**算术强度**衡量一个任务偏计算还是偏访存：

```
算术强度 I = 浮点运算次数（FLOPs） / 搬运字节数（Bytes）      单位 FLOP/Byte
```

Roofline 把它画成一条「屋顶线」：左半段是被带宽压住的斜坡，右半段是被峰值算力压住的水平线，交点就是这块卡的**平衡点算术强度**。



以 **H100 SXM** 为例（原文数据）：

```
BF16 稠密算力 ≈ 990 TFLOPS（官方标称 1979 TFLOPS 是开启稀疏后的数值，稠密约一半）
HBM3 带宽     ≈ 3.35 TB/s
平衡点 I* = 990e12 / 3.35e12 ≈ 295 FLOP/Byte
```

即每搬 1 字节要配上约 295 次浮点运算，才刚好把算力和带宽同时喂饱。**低于这个值就落在斜坡上，再强的算力也用不上。**

Decode 的算术强度可以直接推：设权重字节数为 `W`（FP16 下参数量 `W/2`），每步搬约 `W` 字节，每个权重一次乘加（2 FLOPs）：

```
每步 FLOPs ≈ (W/2) × 2 = W
每步访存    ≈ W 字节
算术强度 I ≈ 1 FLOP/Byte      ← 比 H100 平衡点低两个数量级
```

原文的结论一句话：**Decode 慢，不是因为算不过来，而是因为「搬不过来」。**

这条结论直接推出四件事：

现象	推导	对应技术
Batch 提吞吐却几乎不增延迟	B 个请求拼批，权重搬一次服务 B 个 token，算术强度放大约 B 倍	Continuous Batching
Weight-only 量化加速 Decode	权重字节降到 1/4	GPTQ/AWQ/Marlin
小 Batch 时 CPU 发射开销浮出水面	每步计算几微秒，CPU launch 也几微秒	CUDA Graph
长上下文时瓶颈从权重转向 KV	KV 随序列线性增长，权重是固定开销	KV 量化、PagedAttention

> 【推断】第四条值得强调：1.3 的推导假设「只搬权重」。上下文很长时每步要读的 KV 可能超过权重，算术强度被进一步拉低，权重量化的收益随之稀释。**「长上下文下 KV 量化比权重量化更关键」不是因为它更先进，而是瓶颈位置变了。**

1.4 KV Cache：用显存换计算

不缓存时，每生成一个新 token 都要把前面所有 token 重新过一遍网络算 K、V，计算量随序列长度平方级膨胀。KV Cache 的思想是：把算过的 Key、Value 存起来，下一步直接复用。

一个必须注意的不对称：**Query 不需要缓存**（每步只用当前 token 的 Q），而 **Key、Value 是历史累积、每步全量用到**，所以缓存 K 和 V——这也是名字的由来。

生命周期四阶段，理解它才能看懂 PagedAttention 在优化哪一环：

```
📥 分配 ──▶ ✍️ 填充 ──▶ 🔁 使用 ──▶ 🗑️ 释放
   │          │          │          │
按需给块   Prefill 写整段  每步读全量   请求结束
(2.1 分页)  Decode 追加 1   历史 K/V    归还块池
```

1.4.1 显存账本

```
per_token_bytes = 2 × L × H_kv × d_head × bytes
total_bytes     = per_token_bytes × seq_len × batch_size
```

`2` 是 K/V 两份；`L` 层数；`H_kv` 是 **KV 头数**（不是 Query 头数）；`bytes`：FP16/BF16 为 2、FP8/INT8 为 1。

原文例子：32 层、32 个 KV 头（MHA）、`d_head=128`、FP16：

```
per_token = 2 × 32 × 32 × 128 × 2 = 524,288 B ≈ 0.5 MB
```

上下文长度	单请求 KV	50 并发	单张 80GB 卡
1,024 token	≈ 0.5 GB	≈ 25 GB	勉强
4,096 token	≈ 2 GB	≈ 100 GB	❌ 直接爆
32,768 token	≈ 16 GB	≈ 800 GB	❌ 需多卡

1.4.2 GQA 为什么能省四分之三

`H_kv` 在公式里是线性因子。GQA 让多个 Query 头共享一组 KV 头，32 个 Query 头共享 8 个 KV 头，KV 直接降到 1/4，表达能力几乎不受影响。

变体	Query 头	KV 头	相对 KV 显存
MHA	32	32	1×
GQA（8 组）	32	8	1/4
MQA	32	1	1/32

> 【补充】MLA（DeepSeek 系列）走另一条路：把 K/V 压到低秩潜向量再缓存，KV 比同规模 GQA 更低，但 Attention 形态完全不同，需要专门 Kernel（2.6 节）。算显存时不能套上面的公式。

1.4.3 碎片化：PagedAttention 的动机

传统做法按「最大可能长度」预留一整块连续显存，于是有**内部碎片**（按 2048 预留、实际 100 个就 EOS）与**外部碎片**（请求间空隙凑不成整块）。vLLM 论文测得：当时的系统里只有 **20.4%–38.2%** 的 KV 显存真正存放了 token 状态。这就是 PagedAttention 的起点。

1.5 指标体系：如何衡量「快」

指标	定义	由什么决定	适用
TTFT	提交 → 收到第一个 token	排队 + Prefill + 网络	聊天、流式；首字印象
TPOT	Decode 阶段平均每 token 耗时	带宽、Batch、KV 长度	长输出的吐字速度
ITL	相邻两个输出 token 的间隔	同上，口径更细	流式体验

指标	定义	由什么决定	适用
E2E	提交 → 完整响应	`TTFT + TPOT×(N−1)`	批处理、离线

TPOT 与 ITL 多数语境下同义，差别在于是否含首 token。GenAI-Perf 的口径是**排除首 token**：

```
ITL = (e2e_latency − TTFT) / (输出 token 数 − 1)
E2E = TTFT + TPOT × (输出 token 数 − 1)
```

原文的两个算例（TTFT = 500 ms、TPOT = 40 ms）：

场景	输出长度	E2E	TTFT 占比	优化重心
短输出（客服问答）	20 token	500 + 40×19 = 1,260 ms	≈ 40%	**TTFT**
长输出（长文生成）	800 token	500 + 40×799 ≈ 32,460 ms	≈ 1.5%	**TPOT**

同一套系统，业务形态不同，优化重心天差地别。 这就是「先测清楚输入输出长度分布，再谈优化」这条铁律的来源。

吞吐有两种口径：系统 token/s（集群产能）与单用户 token/s（≈ 1/TPOT）；另有 RPS/QPS。系统吞吐随并发上升直到 GPU 饱和后趋平甚至回落，**吞吐与延迟通常是一对矛盾**。

> 【量级】吞吐随并发的形状：低并发近似线性上升 → 拐点 → 平台期 → 过载后尾延迟急剧恶化而吞吐不再上升。拐点位置由模型、卡型、长度分布决定，必须实测。

1.5.1 尾延迟与 Goodput

平均延迟最容易骗人。生产看 **P50 / P95 / P99**——后者才是用户投诉的来源。比 Raw QPS 更贴近价值的是 **Goodput**：只统计满足 SLO（如 TTFT < 1s 且 TPOT < 50ms）的那部分请求。

```
Goodput = 单位时间内满足 SLO 的请求数（或 token 数）
```

一个系统 QPS 很高但大部分请求超时，Goodput 就很低。第 7 章的 P/D 解耦、SLO 感知调度，优化目标正是 Goodput。

> 【补充】Goodput 必须绑定「SLO 达成率」。工程上常见写法是「在 90% 请求满足 TTFT/TPOT 约束的前提下，每 GPU 每秒能服务多少请求」——DistServe 论文就是这个口径（第 7 章）。只说「Goodput 提升 3 倍」而不说 SLO 与达成率，等于没说。

1.5.2 指标互相耦合

任何旋钮都不是单向上调。**容量规划必须在目标并发下压测延迟，而不是看空载数据。**

1.6 常见误解

误解	纠正
「Decode 慢是因为算力不够」	是带宽不够，单请求算力利用率不到 1%
「吞吐上去了体验就好了」	吞吐与尾延迟常反向，要看目标并发下的 P95/P99 与 Goodput
「KV Cache 是模型的一部分，大小固定」	它随序列长度与并发线性增长，是运行时变量
「GQA 只是模型结构选择」	它直接决定 KV 显存与可承载并发
「TPOT 和 ITL 是同一个东西」	口径不同（是否含首 token），跨工具对比必须对齐

第 2 章 推理引擎核心技术：把显存与调度同时榨干

> 对应 AIInfraGuide 模块四 第 2 章（原文 2.1～2.5 五篇正文）。这一章是全模块的重心：五节技术解决同一个问题的五个侧面——如何让「一次权重搬运」服务尽可能多的有效 token，同时不让任何请求被饿死。

2.1 五节技术一览

节	解决的问题	一句话机制	主要收益
2.2 PagedAttention	显存碎片、无法共享	KV 切块 + 块表映射	并发上限 → 吞吐
2.3 Continuous Batching	GPU 空转、排队久	按迭代步重新组批	吞吐 + 排队延迟
2.4 Prefix Cache	重复前缀重复 Prefill	块哈希复用 KV	TTFT + 吞吐
2.5 Chunked Prefill	长 Prefill 拖慢 Decode	预算截断即切块	TPOT 尾延迟
2.6 后端与图优化	CPU launch 开销	专用 Kernel + CUDA Graph	小 Batch 延迟

> 这一章的主线只有一条：**让一次权重搬运服务尽可能多的有效 token，同时不让任何请求被饿死**。2.2 与 2.3 解决「装得下、排得开」，2.4 与 2.5 解决「不重复算、不互相堵」，2.6 解决「单步本身够快」。

2.2 PagedAttention：像操作系统管理内存一样管理 KV Cache

2.2.1 连续分配的两宗罪

传统框架给每个请求分配一整块**连续**显存，并按「可能生成的最大长度」**预留**：

- **内部碎片**：按 2048 预留、实际 100 个就 EOS，剩下 1948 个位置锁着不用。
- **外部碎片**：不同请求预留块之间留下大小不一的空隙，加起来不少，但每一块都不够放下一个完整请求。

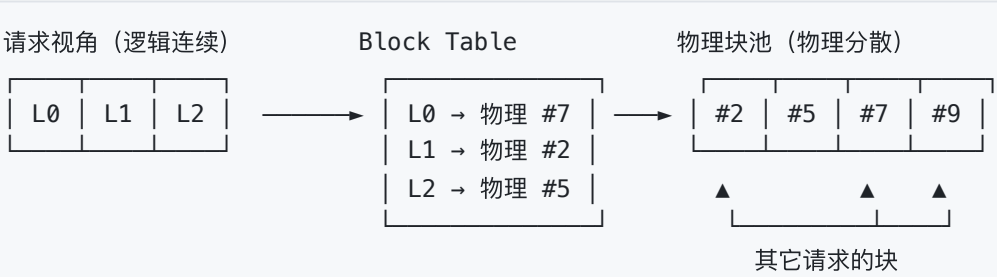
原文的餐厅类比：给每桌都按「最多来 10 人」留十人桌，结果大部分桌子只坐两三人；新来一桌 4 人，却因为没有一张完整的空桌而在门口排队。

> 碎片的根源是**连续 + 预留**两个约束叠加。去掉任何一个，问题就缓解；两个都去掉，问题消失。PagedAttention 正是从这两个约束入手。

2.2.2 核心思想：把虚拟内存分页搬到 GPU 上

OS 早就遇到过同样的问题：进程需要连续地址空间，物理内存却被切得七零八落。解法是虚拟内存分页——进程看到连续的虚拟地址，OS 通过页表映射到任意分散的物理页。

PagedAttention 原样照搬：KV 切成固定大小的 **Block**（每块存固定数量 token 的 K/V，`block_size` 常见默认 16）；请求的逻辑块连续、物理块可散落任意位置；用 **Block Table** 记录「第几个逻辑块 → 哪一个物理块」。



> ⚠️ 术语陷阱：这里的 **KV Block** 是 vLLM 自己的显存管理单位，和 CUDA 的 thread block（线程块）完全是两回事。

2.2.3 Block Table 怎么工作

设 `block_size = 4`（真实常用 16，这里为便于画图取 4）。一个请求 Prefill 了 9 个 token，需要 `ceil(9/4) = 3` 个逻辑块：

逻辑块	覆盖 token	物理块	状态
0	token 0—3	#7	已满（4/4）
1	token 4—7	#2	已满（4/4）

逻辑块	覆盖 token	物理块	状态
2	token 8	#5	部分填充（1/4）

物理块编号可以完全不连续。进入 Decode 后，每生成一个新 token 就往当前逻辑块空位填；逻辑块 2 还剩 3 格，能再接 3 个 token，填满后才按需申请新物理块挂到逻辑块 3。**用多少申请多少，绝不提前预留。**

2.2.4 如何同时消灭两种碎片

维度	传统连续分配	PagedAttention
显存布局	每请求一整块连续显存	固定大小块，物理分散
分配时机	提前按最大长度预留	按需逐块分配
内部碎片	严重（预留 ≫ 实际用量）	极小（< 1 块，`block_size=16` 时最多 15 个 token 的位置）
外部碎片	存在（空隙大小不一）	消除（块大小统一）
显存利用率	偏低（论文测得 20.4%–38.2%）	接近最优

显存利用率提高后，同一张卡能塞下更多并发；而 1.3 已证明 Decode 是 Memory Bound，把更多请求拼进 batch 能显著提吞吐。**所以 PagedAttention 省下的显存最终变成了吞吐提升**——这是全章最重要的因果链。

2.2.5 Kernel 视角：分散的块怎么被读出来

传统 Attention 假设 K/V 连续排列，Kernel 用一个基地址加偏移就能顺序扫。PagedAttention 打破连续性，Kernel 必须先查块表，把「逻辑位置」翻成「物理地址」。两个核心量：`physical_block_number`（去哪个物理块）与 `physical_block_offset`（块内第几格）。

一个 Warp 在外层循环里逐块推进：每次迭代通过块号跳到下一个物理块，`k_ptr` 随之指向不同块里的 Key。物理上东一块西一块的存储，被 Kernel 当成逻辑连续的上下文来读。

> ⚠️ 原文标注：这份 Kernel 文档是「基于原始论文的历史文档，不再反映 vLLM 当前代码」，请当作理解原理的示意。真要看实现，看 V1 的 Attention 后端（2.6）。

2.2.6 内存共享与 Copy-on-Write

因为 KV 以块为单位、通过块表间接引用，多个请求的块表可以指向同一个物理块：

- **跨请求共享前缀**：一批请求用相同 System Prompt，让块表都指向同一批物理块即可。这种共享是**只读**的——2.4 节 Prefix Cache 的物理基础。
- **同一请求内分叉**：同一 Prompt 并行采样多个回答（`n > 1`）或 Beam Search，多个候选共享 Prompt 的 KV，但在某块内开始分叉。

分叉后要写共享块时，论文与 vLLM V0 的答案是 **CoW**：

```
只读共享：   A ─┐
                ├──► 物理块 #3（ref_cnt = 2）
             B ─┘

B 要写入：   复制 #3 → #9，在 #9 上写，B 的块表改指 #9
             A ──► #3（ref_cnt = 1）      B ──► #9（私有副本）
```

这与 Linux `fork()` 之后父子进程共享内存页如出一辙。理解了 OS 的分页与 CoW，PagedAttention 里几乎没有新东西。

2.2.7 抢占与恢复

并发太多、物理块池被抽干时，新来的 Decode 步申请不到块。vLLM 的答案是**抢占**：临时暂停一部分请求，把资源让给别人，等有空间再恢复。

恢复方式	做法	优点	代价
换出（Swapping）	KV 从 GPU 拷到 CPU，恢复时拷回	不用重算	占 CPU 内存；PCIe 往返
重计算（Recompute）	丢弃 KV，恢复时把已生成 token 当新 Prompt 重新 Prefill	不占 CPU 内存，实现简单	重跑 Prefill，浪费算力

> **vLLM V1 的抢占只走重计算**：释放全部块、`num_computed_tokens` 清零、放回等待队列头部，恢复时重新 Prefill。官方调优文档的解释是 V1 架构下重计算开销更低。开着 Prefix Cache 时，被释放的满块只要没被淘汰，恢复时还能直接命中，实际重算的部分会少一些。
>
> 📌 日志里频繁出现 preemption 告警，说明并发压得太满——调低并发、提高 `gpu_memory_utilization` 给 KV 更多空间，或加卡。它是「系统在硬扛」的信号，不是常态。

2.2.8 工程参数

```
llm = LLM(model="meta-llama/Llama-3.1-8B-Instruct",
          gpu_memory_utilization=0.92,   # v0.32.0 默认；扣掉权重与激活峰值后全划给 KV 池
          block_size=16,                 # 每块存多少 token 的 KV
          enable_prefix_caching=True,    # 前缀缓存开关（2.4）
          max_num_seqs=256)              # 单批最多多少条序列
```

参数	影响	何时动
`gpu_memory_utilization`	越大 → KV 池越大 → 并发越高、抢占越少；太大会 OOM	**最常动**
`block_size`	越小 → 块表越长、开销越大；越大 → 尾块碎片越多	一般不动
`enable_prefix_caching`	有共享前缀时省掉重复 Prefill	有 System Prompt 就开

参数	影响	何时动
`max_num_seqs`	与 Token Budget 共同决定实际批大小	高并发时上调，注意 TPOT

> 【推断】`block_size` 的权衡一句话：**块大小 × 并发数决定块表长度，块大小决定尾部浪费的期望（约半块）**。vLLM 选 16 是通用负载下的折中，特殊负载可试 8/32，但要实测。

2.3 Continuous Batching：让 GPU 不再空等

2.3.1 为什么批处理有效

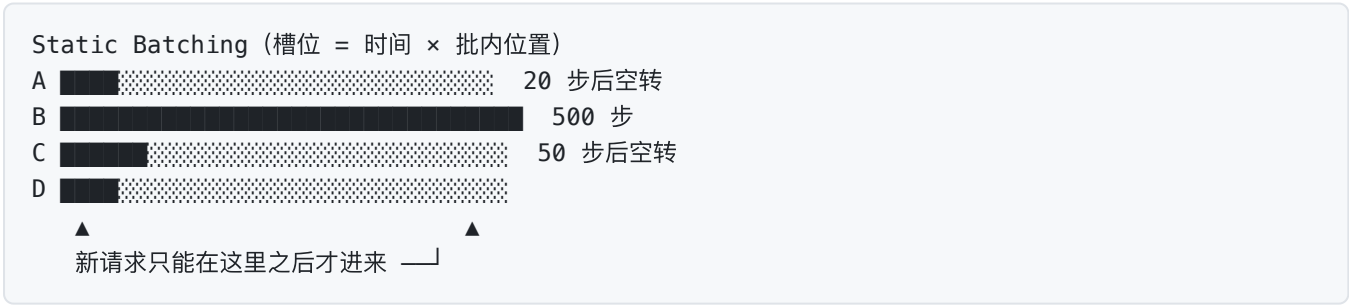
回到第一性原理：Decode 是 Memory Bound，瓶颈在把权重从显存搬到计算单元。搬一次权重只服务一个请求的一个 token 太亏；让这一次搬运同时服务多个请求的 token，算术强度成倍上升。原文的通勤班车类比：为一个人发一趟车油钱极不划算，坐满一车人再发，每人分摊骤降。

> 📌 批处理提升吞吐，而且**几乎不牺牲单请求延迟**——瓶颈在搬权重，多算几个 token 对 Memory Bound 的 Decode 近乎「免费」。问题只在于怎么组织这个 Batch。

2.3.2 Static Batching 的致命短板

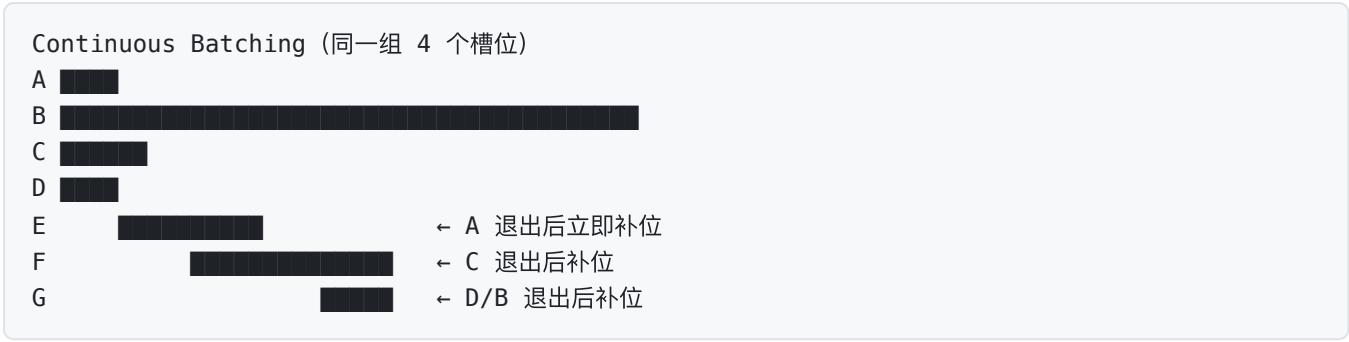
最朴素的做法：攒够一批，一起送进模型，等**这一批全部生成完**再开始下一批。但请求长度天差地别：A 生成 20 个就结束、B 要生成 500 个，A 完成后那 480 步占的槽位纯粹空转。

原文可手算的例子：一批 8 个请求，最长 500 步、其余 7 个平均 50 步，则 `8 × 500 = 4000` 个槽位步里只有 `500 + 7×50 = 850` 个在干活，**约 79% 空转**。



2.3.3 迭代级调度

> 🔑 **Iteration-level Scheduling**：不再以「一批请求的完整生命周期」为调度单位，而是每生成一步就重新调度一次。每步结束后立刻检查：哪些请求 EOS 或达到长度上限 → 立即移出、释放显存；腾出的空间 → 立即从等待队列拉新请求进来。



收益两块：完成的请求立刻退出（不再空转）+ 新请求立刻补位（排队延迟大幅下降）。最早系统化提出迭代级调度的 Orca 论文报告：在 GPT-3 175B 上、相同延迟下，吞吐比当时的 FasterTransformer 高 **36.9 倍**（原文注明含 Orca 的其他优化，且依赖负载）。

2.3.4 与 PagedAttention 的协同

技术	解决的问题	提供的能力
PagedAttention	显存碎片、利用率低	细粒度、可即时回收的显存分配
Continuous Batching	GPU 空转、排队延迟	迭代级调度、动态批次构成

要做到「随退随补」，前提是能随时、以细粒度分配回收显存。若是连续大块分配，请求退出留下的空隙未必能恰好塞下新请求，动态补位无从谈起。

> 🔑 PagedAttention 管「显存怎么放」，Continuous Batching 管「请求怎么调度」，**二者相乘才有高吞吐**。单有其一都不够。

2.3.5 V1 调度器源码骨架

V1 把逻辑集中在 `vllm/v1/core/sched/scheduler.py`。**最大的设计变化是调度器眼里不再有「Prefill 阶段」与「Decode 阶段」之分**：每个请求只是不停把 `num_computed_tokens` 推向 `num_tokens_with_spec`。这也是 2.5 节统一预算调度的前提。

- `schedule()`：给一个 `token_budget`，先遍历 running 分配 token 与 KV 块；预算有富余且没发生抢占时，再从 waiting 拉新请求。
- `update_from_output()`：把采样出的 token 追加回各请求、检查停止条件、释放完成请求、组装结果。

```
# 伪代码：还原 V1 Scheduler.schedule() 的主干（非逐字源码）
def schedule(self):
    token_budget = self.max_num_batched_tokens
    scheduled, preempted = [], []

    # 第一优先级：喂饱正在运行的请求（Decode 为主）
    req_index = 0
    while req_index < len(self.running) and token_budget > 0:
        req = self.running[req_index]
        num_new = compute_new_tokens(req, token_budget)   # 受预算/max_model_len 限制
        while True:
            blocks = self.kv_cache_manager.allocate_slots(req, num_new)
            if blocks is not None:
                break
            victim = self.running.pop()                   # FCFS 踢最后一个（最晚加入）
            self._preempt_request(victim)
            preempted.append(victim)
            if victim is req:                             # 踢到自己，本步不跑
                break
        if blocks is None:
            break
        scheduled.append(req)
        token_budget -= num_new
        req_index += 1

    # 第二优先级：预算有富余且本步没抢占，才拉新请求
    if not preempted and token_budget > 0:
        while self.waiting and token_budget > 0:
            req = self.waiting.pop_front()
            if not self._can_admit(req):
                break
            self.running.append(req)
            scheduled.append(req)

    return SchedulerOutput(scheduled)
```

两个关键设计：**在途请求优先于新请求**（避免新请求不断插队、老请求迟迟跑不完的饥饿）；**抢占发生在显存耗尽时**（`allocate_slots` 返回 `None` 就挑受害者，释放后重试，直到成功或踢到自己）。FCFS 踢 running 里最后一个；PRIORITY 踢 `(priority, arrival_time)` 最大的。

被抢占的请求：释放 KV 块与编码器缓存 → 状态置 `PREEMPTED` → `num_computed_tokens` 归零 → 塞回 waiting 队头。**归零这一步说明 V1 用重计算**，而不是从 CPU 换回 KV。

2.3.6 三组权衡

权衡	一边	另一边	工程含义
吞吐 vs 延迟	塞更多请求 → 吞吐高	每步前向变长 → TPOT 变差	批大小取平衡
Prefill vs Decode	新请求 Prefill 是重活	长 Prompt 拖慢同批 Decode	由 2.5 解决

权衡	一边	另一边	工程含义
公平 vs 效率	FCFS	长请求被反复插队的饥饿	调度策略取舍

> ⚠️ Continuous Batching 让延迟指标**对负载敏感**：低负载批次小、延迟低；高负载批次大、排队变长，P95/P99 明显上升。**容量规划必须在目标并发下压测。**

2.4 Prefix Cache 与 RadixAttention

2.4.1 重复的前缀有多浪费

真实业务里海量请求共享同一段开头：共享 System Prompt（800 token 的角色设定）、Few-shot 提示（几千 token 示例）、多轮对话（历史累积复用）。共同点是：大量 token 在不同请求间完全相同，**且位置相同（都在开头）**。而 Prefill 是 Compute Bound 的重活，重复 Prefill 等于反复做同样的昂贵计算。

> 📌 Attention 的因果性决定了：一个 token 的 K/V 只取决于它自己和它前面的 token。所以两个请求只要前缀完全一致，这段前缀每 token 的 KV 就一模一样，可以只算一次、大家共用。原文的合同类比：标准合同前面几页通用条款对所有客户都一样，存一份模板，签约时只补个性化条款。

2.4.2 物理基础：块级只读共享

- KV 以固定块存储，请求通过块表间接引用物理块。
- 两个请求的某些前缀块内容相同，就让块表指向同一物理块，配合引用计数记录几方在用。
- 新生成内容写在各请求自己的新块里，共享块只读。

> 🔑 PagedAttention 的块共享是**空间维度**（并存请求共享），Prefix Cache 把它延伸到**时间维度**（先后请求复用）。

2.4.3 vLLM 的 Automatic Prefix Caching

「Automatic」是亮点——不需要手动声明哪段是共享前缀，引擎自动检测。靠的是对 KV 块做哈希：

```
block_hash = Hash(parent_hash, block_tokens, extra)
```

`parent_hash` 是父块哈希，`extra` 含 LoRA ID、多模态输入哈希、`cache_salt` 等。**把父块哈希纳入是精髓**：它保证只有从序列开头到当前块的整条前缀都一致，哈希才相同。第 3 个块哈希相同，意味着前 3 块全部 token 相同。

```
请求 A:  [S0][S1][S2][A0][A1]        S = 共享 System Prompt
请求 B:  [S0][S1][S2][B0][B1]
哈希链:  h0=H(NONE,S0) → h1=H(h0,S1) → h2=H(h1,S2) → h3=H(h2,A0) → ...
                          ▲             ▲             ▲
                       A、B 相同      A、B 相同      A、B 分叉
```

流程：新请求逐块算前缀哈希去缓存表查；**命中**就把块表指向该块、引用计数 +1、跳过这部分 Prefill；**未命中**就正常 Prefill，算完把新块哈希登记进缓存。

> 📌 命中前缀缓存直接砍掉那部分 Prefill 计算，最直接的收益是**大幅降低 TTFT**，省下的算力还能服务更多请求。共享前缀越长、命中率越高，收益越大。

V1 的「零开销」设计：V0 时代因命中率低时 CPU 记账开销不划算，默认关闭。V1 重写了这套机制：块在初始化时预分配成块池，用嵌入块内的双向链表指针做 O(1) 移动与淘汰。官方设计文档称前缀缓存「几乎是免费的午餐」，V1 发布博客给出的数据是：**即使命中率为 0%，吞吐下降也不到 1%**。因此 V1 默认开启。

> 【补充】哈希算法：vLLM 自 v0.11 起默认 `sha256`，可用 `--prefix-caching-hash-algo` 切换为 `sha256_cbor`（跨语言可复现）、`xxhash`、`xxhash_cbor`。非加密算法更快但碰撞风险略高。**跨进程/跨副本复用缓存时，算法必须一致。**

2.4.4 淘汰：LRU + 引用计数

- **正在使用的块不能淘汰**：引用计数 > 0 受保护。
- **空闲块按 LRU 淘汰**：请求结束后引用计数归零、进入空闲队列，但内容与哈希先保留，万一马上有匹配的请求还能命中；显存吃紧时才真正回收。

一个细节：释放块时把请求的块**按逆序**放进空闲队列尾部。因为最后一个块哈希了最多的 token（前缀最长），最不容易被命中，让它优先被淘汰；靠前的块（如共享 System Prompt 的开头）更可能被复用，留得更久。

> 💡 **越靠近序列开头的前缀，越通用、越值得留；越靠后的越个性化，留着也难命中。**

2.4.5 SGLang 的 RadixAttention

vLLM 的哈希方案高效，但匹配是「块对齐 + 线性前缀」。而多轮对话、树状采样的共享关系是**树状**的。RadixAttention 用一棵 **Radix Tree** 管理复用：

- 每条边是一段 token 序列，从根到某节点的路径就是一段前缀。
- 新请求沿树做**最长前缀匹配**，匹配到的部分复用 KV，只对分叉之后的新内容计算。
- 分叉天然对应「共同前缀 + 不同后续」。
- 每个节点带引用计数，只有为 0 可淘汰；淘汰按 LRU 且**先淘汰叶子**，祖先的公共前缀留得更久。
- 调度时优先处理与缓存共享前缀最长的请求（cache-aware scheduling）。

维度	vLLM APC	SGLang RadixAttention
数据结构	Hash 表（块哈希 → 块 ID）	Radix Tree（压缩前缀树）
匹配	块对齐的前缀哈希	树上最长前缀匹配（以单 token 为页）
擅长	通用共享前缀（System Prompt、Few-shot）	树状共享（多轮对话、树采样、Agent）
淘汰	LRU + 引用计数	LRU + 引用计数，先淘汰叶子

> ⚠️ 不要把两者对立。vLLM 的方案在绝大多数场景够用，且 V1 做到近乎零开销；RadixAttention 在高度树状复用的负载下展现额外优势。SGLang 论文在这类结构化 LLM 程序负载上报告最高 **6.4 倍**吞吐提升，该数字来自 RadixAttention 与其他优化的组合。**选型看负载形态。**

2.4.6 适用场景与 Prompt 工程

收益大	收益有限
长 System Prompt / 长 Few-shot 被大量共享	每个请求前缀都不同
多轮对话（历史累积复用）	共享部分放在 Prompt **中间或末尾**
同一 Prompt 并行采样（n > 1）	前缀极短（不足一块）
RAG 固定指令模板 + 变化的检索内容	高随机性采样导致后续全部分叉

> 📌 Prompt 工程上要有意识地「**把公共内容前置**」——固定指令、示例、上下文放最前，变化的用户输入放最后。几乎零成本却常被忽视。
>
> ⚠️ 前缀缓存不改变模型输出（复用数学上完全等价的 KV），可放心开启。但多租户要注意隔离——vLLM 提供 `cache_salt`，避免不同租户的请求意外命中彼此缓存造成信息泄露。

2.5 Chunked Prefill 与统一 Token Budget

2.5.1 Prefill 怎么拖慢 Decode

Prefill 是 Compute Bound（Prompt 越长，这步矩阵运算量越大）；Decode 是 Memory Bound（每步 1 个 token，很快）。而 Continuous Batching 是**每步重新组批**的：如果某步把一个 4000 token 的 Prefill 和一批 Decode 塞进同一个 step，这一步的耗时会被那个庞大 Prefill 整体拉长——**同批所有 Decode 的这个 token 都得等它算完**。

原文的流水线类比：Decode 像匀速前进的小件包裹，突然来了一个集装箱也要挤上同一条线过检，整条线被堵住。用户端表现：出字很流畅，突然卡一下，再恢复——**TPOT 出现尖刺**。

2.5.2 把长 Prefill 切块

不要求长 Prompt 一步 Prefill 完，而是切成若干 Chunk 分散到多步。SARATHI 按等长切块；vLLM 不预设块长，每块多大由**当步剩余 Token 预算**决定。

```
切块前（一步堵死）：
step k   |◄──────── 4000 token 的 Prefill ────────►|  ← 这一步特别长
step k+1 |◄── Decode 8 条 ──►|                        ← 用户感受到卡顿

切块后（摊到多步）：
step k   |◄─512─►|+ Decode 8 条 |
step k+1 |◄─512─►|+ Decode 8 条 |
...
step k+7 |◄416►|+ Decode 8 条   |  ← 首 token 在这里出现
```

> ⚠️ 切块会让这个长 Prompt 的 **TTFT 略微变大**（要跨多步才 Prefill 完）。这是刻意权衡——牺牲少量单请求 TTFT，换整批 Decode 的 TPOT 稳定。

2.5.3 Token Budget：调度的统一货币

核心洞察：**无论 Prefill 还是 Decode，对 GPU 来说本质都是「处理若干个 token 的前向计算」**。Decode 请求贡献 1 个 token，Prefill 请求（或它的一个 Chunk）贡献 N 个。于是给每步设总预算，先保证 Decode（每个占 1），剩余分给 Prefill Chunk：

```
Σ(Decode 请求 × 1) + Σ(Prefill Chunk × chunk_size) ≤ token_budget
```

> 💡 用 Token 预算这把统一的尺子，「Chunk 该切多大」「一步能塞几个 Decode」「长 Prefill 和 Decode 怎么共处一步」都归结成同一个**装箱问题**。

2.5.4 V1 的统一调度器

V0 把 Prefill 与 Decode 当作两类阶段区别对待，代码复杂且容易在切换时产生空隙。V1 彻底取消二分，调度决策就是一个映射：

```
{
    "req_A": 1,      # 正在 Decode，处理 1 个新 token
    "req_B": 1,
    "req_C": 512,    # 长 Prompt 的一个 Prefill chunk
    "req_D": 128,    # 短 Prompt，一步 Prefill 完
}
# 约束：所有 value 之和 ≤ token_budget
```

> 🔑 `{request_id: num_tokens}` 这一个统一表示，**同时涵盖 Chunked Prefill、Prefix Cache 与 Speculative Decoding**——它们本质都是「这个请求这一步要处理多少 token」的不同取值：Prefill chunk 是大值，Decode 是 1，投机验证是若干候选。一套框架统一所有情况。这也是 V1 能把 Chunked Prefill 做成默认行为的原因。

2.5.5 预算截断即切块

关键在：计算「这步处理多少 token」时用剩余预算截断。长 Prompt 一步吃不完就只吃预算允许的那一截——**切块不是独立函数，而是预算截断的自然结果**。

一个具体的装箱（预算 2048）：

请求	阶段	剩余需求	分配	累计
8 个 Decode	Decode	各 1	各 1	8
R（running 的长请求）	Prefill 续块	1440	1440	1448
W1（waiting 新请求）	Prefill 首块	900	600（截断）	2048
W2	—	300	0	2048

- `min(need, token_budget)` **就是切块的本质**：4000 token 的 Prompt，这步只剩 512 预算就只 Prefill 512，剩下 3488 留到后续步——自动成 Chunk，没有专门的分块器。
- **续块靠** `num_computed_tokens` **记账**：下一步从断点继续，直到追上 `num_prompt_tokens` 后自然转入每步 +1 的 Decode。

2.5.6 参数与调优

参数	含义	调优方向
`max_num_batched_tokens`	一步最大 token 数（Token Budget）	调大 → 吞吐↑、TPOT 可能↑；调小 → TPOT 更稳、吞吐可能↓
`max_num_seqs`	一步最多多少条序列	受显存/KV 约束，与上者共同决定批大小
`long_prefill_token_threshold`	超长 Prompt 单步处理量上限	默认 0（不限制）；混跑大量长 Prompt 时设值。批里只有它一个时不生效

官方调优文档的参考刻度：较小值（如 **2048**）得到更好的 ITL，因为拖慢 Decode 的 Prefill 更少；较大值能缩短 TTFT；追求吞吐时建议设到 **8192 以上**，小模型跑在大显存卡上尤其如此。

> 💡 没有普适最优值。正确做法是在目标负载下压测，观察 TTFT、TPOT 的 P95/P99 与吞吐三条曲线，**找到满足 SLO 的最大吞吐点**。

收益明显	收益有限
Prompt 长度差异大、有超长 Prompt（长文档问答、长上下文 RAG）	纯离线批处理，只追求总吞吐、不在乎抖动
对出字流畅度敏感、要压 TPOT 尾延迟的在线服务	Prompt 都很短，本来就没有长 Prefill 独占一步
Prefill 与 Decode 混合的高并发负载	显存极紧、切块调度开销不划算

> 📌 Chunked Prefill 是「平稳性」优化而非「峰值吞吐」优化。它与 Prefix Cache 是天然搭档——前缀命中先砍掉一部分 Prefill，剩下的再切块掺入 Decode。

2.6 Attention 后端与图优化：把 Decode 的 CPU 开销榨干

2.6.1 反直觉的瓶颈：CPU 拖了 GPU 的后腿

前面讲的是「宏观」技术。这一节拉到「微观」：同样一批请求的一次前向，怎么让它跑得更快?

GPU 不会主动干活，每个操作都要 CPU 通过一次 launch 命令告诉它。跑一遍 Transformer 有成百上千个小操作，就对应成百上千次 CPU→GPU 启动调用，每次有微秒级的固定开销。Prefill 时每个 Kernel 跑几百微秒甚至毫秒，这点开销被淹没；但 **Decode 每步只处理 1 个 token，每个 Kernel 实际计算可能就几微秒——CPU 发射的时间比 GPU 执行的时间还长**。



> 📌 Decode 有**两重等待**——等数据（Memory Bound，靠 Batching/量化缓解）与**等指令**（CPU launch，靠图优化缓解）。小模型、小 Batch 时后者尤其突出。

2.6.2 可插拔的 Attention 后端

后端	特点	典型适用
FlashAttention	IO 感知的经典高性能实现，按架构选 FA2（Ampere）、FA3（Hopper）、FA4（Blackwell）	Ampere/Hopper 默认
FlashInfer	面向推理优化，KV 处理灵活，支持多种数据类型	Blackwell 默认；高吞吐服务
FlashMLA 等 MLA 后端	针对 MLA 优化	DeepSeek 等 MLA 结构

后端	特点	典型适用
Triton	Triton 实现，可移植性好	非 NVIDIA 硬件或自定义兜底

> 🔑 Attention 后端是**同一数学操作的不同工程实现**，结果数学等价，区别在访存模式、并行策略、对硬件指令与模型结构的适配。选对后端，同样的卡能跑出明显不同的吞吐。

v0.32.0 默认：Ampere/Hopper 用 `FLASH_ATTN`，Blackwell 用 `FLASHINFER`，MLA 模型走专门的 MLA 后端。干预用 `--attention-backend` 与 `--attention-config.flash_attn_version`。

2.6.3 后端必须支持混合批次

V1 的统一调度器会把 Prefill Chunk 与 Decode 塞进同一步，那 Attention 怎么算？Prefill 要算「一整段新 token 之间 + 对历史」，Decode 只算「1 个新 token 对全部历史」。这要求后端**原生支持 mixed prefill/decode batch**：FlashAttention 等现代后端提供的**变长（varlen）接口**正是为此设计——把批里所有请求的 Query 首尾拼成变长序列，配合每请求起止偏移，一次 Kernel 算完。

> 📌 **「统一调度器」（软件层抹平）与「支持混合批次的 Attention 后端」（Kernel 层抹平）是配套的。**没有后者，V1 的 Token 预算调度就落不了地。

2.6.4 CUDA Graph

CUDA Graph 把一连串固定的 GPU 操作**录制**成图，之后一条命令**重放**整张图——成百上千次 CPU 启动压成一次。原文的类比：没有它像老师每讲一句都要等秘书传话；有了它相当于提前录好讲稿，按一下播放键一气呵成。

前提是**操作序列与张量形状固定**。这对 Decode 是天作之合——每步都是「处理固定 Batch 个 token」，可以对不同 Batch Size 分别录制，运行时按当前大小选图。

> ⚠️ 正因为要求形状固定，vLLM 只为一组**预设 Batch Size** 捕获：`[1, 2, 4]` + 8 到 248 每隔 8 + 256 到 `max_cudagraph_capture_size` 每隔 16；后者默认 **512**（数据中心级 Blackwell 为 1024）。运行时实际 Batch 会**向上 Padding 到最近的已捕获尺寸**再重放（如 13 个 token 按 16 的图跑，多出的 3 个位置空算）；超过最大捕获尺寸才回退逐 Kernel 启动。**收益主要体现在 Decode 这类小而规整的批次**，大块 Prefill 较难套用。

2.6.5 torch.compile 与 Piecewise CUDA Graph

CUDA Graph 解决「启动开销」，但还能把很多小 Kernel 合并成更少更大的 Kernel——这就是 **torch.compile**（算子融合）。V1 中它默认开启，且所有编译在开始服务前完成，避免请求过程中触发编译导致延迟尖刺。

矛盾在于：CUDA Graph 要求形状固定，而 **Attention 恰恰最难兼容**（变长 KV、复杂 Mask）。V1 的巧解是 **Piecewise CUDA Graph**：

- 把前向图在 Attention 处**切开**（Attention 被包装成「不透明」自定义算子，编译器不动它）。
- Attention 之外的部分（LayerNorm、GEMM、激活、残差等规整的 token-wise 操作）用 CUDA Graph 捕获、重放。
- Attention 本身保持 **eager**，保留处理变长 KV 的灵活性。

> 🔑 **对规整部分用图优化榨干 CPU 开销，对灵活的 Attention 网开一面**。launch 开销的大头被消灭，而 Attention 的灵活性（混合批次、变长 KV）不受损。这是 V1 相比 V0 在 Decode 上提速的关键工程之一。

vLLM 还支持 **Full CUDA Graph**（把 Attention 也录进图），v0.32.0 的默认模式是 `FULL_AND_PIECEWISE`：纯 Decode 批次形状最规整，用一张 FULL 图整步重放；含 Prefill 的混合批次退回 PIECEWISE。完整模式还有 `NONE`、`PIECEWISE`、`FULL_DECODE_ONLY`。

支持级别	后端	含义
ALWAYS	FA3、Triton	任意批次都支持
UNIFORM_BATCH	FA2、FlashMLA	只支持形状统一的批次
UNIFORM_SINGLE_TOKEN_DECODE	FlashInfer	只支持每请求 1 token 的纯 Decode

不满足条件的批次自动退回 PIECEWISE。文档指出 **FULL 图对小模型与 MoE 模型的 Decode 提速最明显**。

2.6.6 权衡

收益	代价
Decode 的 CPU 开销大幅下降，小模型/小 Batch 下吞吐与延迟明显改善	**启动变慢**：编译与图捕获让首次拉起变久
算子融合进一步减少 Kernel 数量	**显存占用**：捕获多个尺寸需额外显存
FULL 图对纯 Decode 批次再提速	**调试复杂度**：编译后的图不如 eager 好调试

> ⚠️ 遇到数值异常或想确认 Kernel 行为时，可临时加 `--enforce-eager`（同时关闭 torch.compile 与 CUDA Graph）。定位完去掉——**生产环境务必保持开启**。

2.7 本章小结

四条必须记住：

1. **PagedAttention 的价值不在省显存本身，而在把省下的显存转化为并发，再转化为吞吐。**

2. **Continuous Batching 必须依赖细粒度可回收的显存**；两者是乘法，不是加法。
3. **Prefix Cache 只对「从头开始连续相同」的前缀有效**，Prompt 上要把公共内容前置。
4. **Chunked Prefill 是平稳性优化**，代价是长 Prompt 的 TTFT 略增；它与前缀缓存是天然搭档。

第 3 章 nano-vllm 源码解读：1200 行里的推理引擎主干

> 对应 AllnfraGuide 模块四 第 3 章（3.1～3.6，六篇正文）。读生产框架最难的是「找不到那一行」——vLLM 十万行里，批处理在哪里拼起来、KV 在哪里分配，很难指着一行说清楚。nano-vllm 用约 1200 行 Python 把主干实现了一遍。

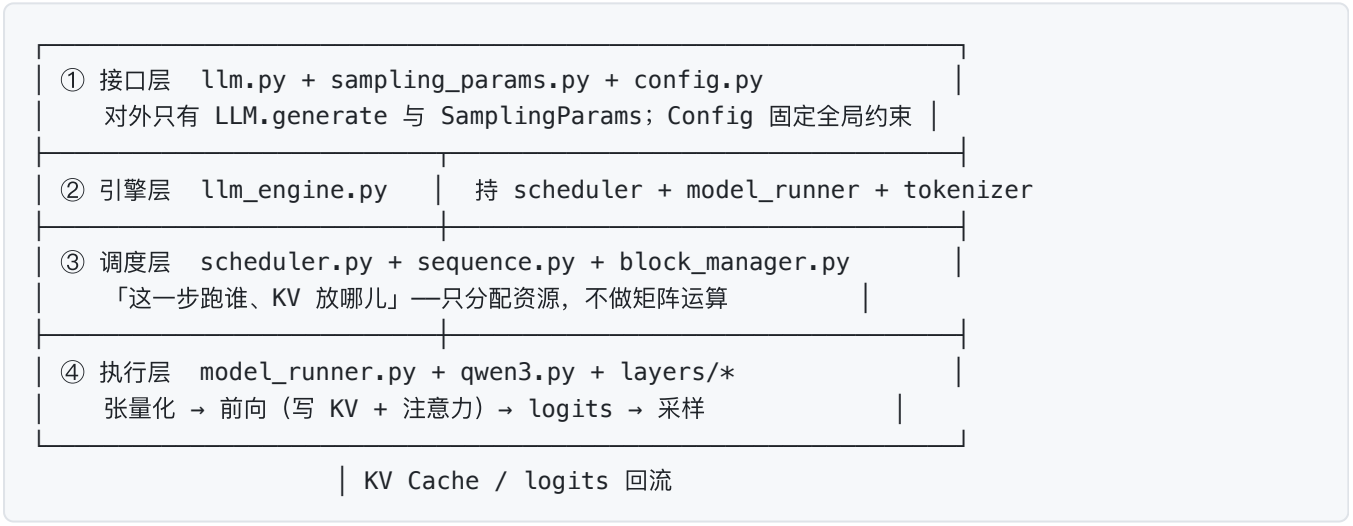
原文的比喻很准确：vLLM 像一座有地下管网、消防系统、电梯调度的摩天大楼；nano-vllm 是一栋结构裸露的样板房——墙体、承重柱、水电走线都露在外面。它不做生产需要的外围功能（大量模型适配、各种量化、集群编排），只保留算法与工程主干。

维度	说明
作者 / 协议	Xingkai Yu（GeeeekExplorer，主页标注 DeepSeek、南京大学）；MIT
代码量	README 称约 1200 行；`nanovllm/` 实际 1385 行（含空行，2026-04-26 提交）
热度	截至 2026-11-01 约 16.2k stars、2.8k forks
版本 / 依赖	`pyproject.toml` 0.2.0；torch>=2.4、triton>=3.0、transformers>=4.51、flash-attn、xxhash
示例模型	`Qwen/Qwen3-0.6B`（`models/` 下只有 Qwen3）
优化套件	前缀缓存、张量并行、torch.compile、CUDA Graph

与 nanoGPT 的类比：nanoGPT 教你「训练循环 + Transformer Block」，nano-vllm 教你「批处理 + KV 管理 + 调度 + 低延迟执行」。

3.1 目录与四层架构

```
nanovllm/
├── __init__.py          # 导出 LLM, SamplingParams
├── config.py            # Config 全局配置 (dataclass)
├── llm.py               # LLM: 继承 LLMEngine, 类体只有 pass
├── sampling_params.py   # SamplingParams 采样超参
├── engine/
│   ├── llm_engine.py    # 引擎主循环: 调度 → 前向 → 采样 → 更新
│   ├── scheduler.py     # Prefill/Decode 调度与抢占
│   ├── sequence.py      # 单条请求的状态机
│   ├── block_manager.py # KV Cache 分块分配与回收
│   └── model_runner.py  # GPU 执行: 张量准备、Context、CUDA Graph
├── layers/
│   ├── attention.py     # Triton 写 KV + FlashAttention 前向
│   ├── linear.py        # 列/行/QKV/合并 并行 Linear
│   ├── embed_head.py    # 词表并行 Embedding 与 LMHead
│   ├── layernorm.py     # RMSNorm
│   ├── activation.py    # SiluAndMul (SwiGLU)
│   ├── rotary_embedding.py  # RoPE
│   └── sampler.py       # 温度采样 (指数噪声 + argmax)
├── models/qwen3.py      # Qwen3 实现
└── utils/               # context.py (全局推理上下文)、loader.py (safetensors 加载)
```



> 📌 一个能判断理解程度的问题：**BlockManager 为什么属于调度层而不是 `layers/`？** 因为它只管块账本（分配、回收、哈希登记），不碰矩阵运算——这正对应 2.2 节「显存管理」与「算子执行」的分离。

文件体量说明重点：`model_runner.py`（257 行）、`qwen3.py`（216）、`linear.py`（156）、`block_manager.py`（120）分别对应执行、模型、TP 与 KV 管理；`llm.py` 只有 5 行。

3.2 Config 与一次 generate

```
llm = LLM(path, enforce_eager=True, tensor_parallel_size=1)   # 参数是本地目录，不下载
sampling_params = SamplingParams(temperature=0.6, max_tokens=256)
prompts = [tokenizer.apply_chat_template([{"role": "user", "content": q}],
                                         tokenize=False, add_generation_prompt=True)
           for q in questions]
outputs = llm.generate(prompts, sampling_params)              # 每项 {"text", "token_ids"}
```

三个与 vLLM 的差异：构造参数是**本地目录**（`Config` 断言 `os.path.isdir`）；返回 dict 列表而非 `RequestOutput`；**对话模板由调用方处理**，引擎内部只 encode。

```
@dataclass(slots=True)
class Config:
    model: str                                  # 本地模型目录（必填）
    max_num_batched_tokens: int = 16384         # 单步最多处理的 token 总数（Token Budget）
    max_num_seqs: int = 512                     # 单步最多并发的序列数
    max_model_len: int = 4096                   # 最大上下文长度
    gpu_memory_utilization: float = 0.9         # 允许占用的显存比例
    tensor_parallel_size: int = 1
    enforce_eager: bool = False                 # True 则禁用 CUDA Graph
    hf_config: AutoConfig | None = None
    eos: int = -1                               # 初始化后由 tokenizer 填入
    kvcache_block_size: int = 256               # 每个 KV 块容纳的 token 数
    num_kvcache_blocks: int = -1                # 运行时按显存推算
```

三个值得单说的字段：

- 🔑 `kvcache_block_size = 256`：比 vLLM 的 16 大得多。**这不是碎片与开销的折中，而是底层算子要求**——flash-attn 的 `flash_attn_with_kvcache` 文档写明，传 `block_table` 时 `page_block_size` 必须是 256 的倍数。nano-vllm 的 decode 路径与前缀命中的 prefill 路径都通过 `block_table` 读分页 KV，所以 256 是**允许的最小值**。代价是尾块浪费更多、前缀匹配粒度更粗。
- 🔑 `max_num_batched_tokens` 与 `max_num_seqs`：前者主要约束 prefill（一步最多算多少 token），后者主要约束 decode（一步最多多少条序列）。两者共同构成 Token Budget。
- 🔑 `gpu_memory_utilization`：不按最大长度预留 KV，而是先加载权重、用一次最大规模 prefill 测激活峰值，再把剩余配额全切成 KV 块——所以 `num_kvcache_blocks` 初始为 `-1`。

```
@dataclass(slots=True)
class SamplingParams:
    temperature: float = 1.0
    max_tokens: int = 64
    ignore_eos: bool = False

    def __post_init__(self):
        assert self.temperature > 1e-10, "greedy sampling is not permitted"
```

> 📌 断言温度 > `1e-10`，即**不支持贪心采样**。Sampler 只有一条随机采样路径，温度为 0 时除法会出错，作者在入口处挡掉而不是写 argmax 分支。`ignore_eos=True` 是 benchmark 的关键：保证每条都生成满 `max_tokens`，吞吐才可比。

`LLMEngine.__init__` 四步：① 从 kwargs 挑出 Config 认识的字段，并把块大小同步给 `Sequence.block_size`；② 用 **spawn** 启动 rank>0 的 `ModelRunner` 子进程（CUDA 不支持在 fork 的子进程里重新初始化）；③ 分词器只在主进程持有，回填 EOS；④ 建 Scheduler 并注册 `atexit` 清理。

```
def generate(self, prompts, sampling_params, use_tqdm=True):
    if not isinstance(sampling_params, list):
        sampling_params = [sampling_params] * len(prompts)
    for prompt, sp in zip(prompts, sampling_params):
        self.add_request(prompt, sp)
    outputs = {}
    while not self.is_finished():
        finished, num_tokens = self.step()
        for seq_id, token_ids in finished:
            outputs[seq_id] = token_ids
    outputs = [outputs[i] for i in sorted(outputs)]        # 按 seq_id 还原提交顺序
    return [{"text": self.tokenizer.decode(ids), "token_ids": ids} for ids in outputs]

def step(self):
    seqs, is_prefill = self.scheduler.schedule()
    # 必须在 postprocess 之前统计：postprocess 会把 num_scheduled_tokens 清零
    num_tokens = sum(seq.num_scheduled_tokens for seq in seqs) if is_prefill else -len(seqs
    token_ids = self.model_runner.call("run", seqs, is_prefill)
    self.scheduler.postprocess(seqs, token_ids, is_prefill)
    outputs = [(seq.seq_id, seq.completion_token_ids) for seq in seqs if seq.is_finished]
    return outputs, num_tokens
```

> 📌 `num_tokens` 用**正负号区分阶段**：prefill 返回正数（这步实际计算的 token 总量），decode 返回 `-len(seqs)`。这行不能挪到 `postprocess` 之后，否则 `num_scheduled_tokens` 已清零，prefill 吞吐会一直显示 0。

3.3 模型与核心算子

```
class Qwen3DecoderLayer(nn.Module):
    def forward(self, positions, hidden_states, residual):
        if residual is None:                          # 第一层：残差从输入初始化
            hidden_states, residual = self.input_layernorm(hidden_states), hidden_states
        else:                                          # 后续层：norm 同时吐出新残差
            hidden_states, residual = self.input_layernorm(hidden_states, residual)
        hidden_states = self.self_attn(positions, hidden_states)
        hidden_states, residual = self.post_attention_layernorm(hidden_states, residual)
        hidden_states = self.mlp(hidden_states)
        return hidden_states, residual
```

Pre-Norm + 残差。nano-vllm 把残差相加放进 RMSNorm 内部：norm 接收上一层残差，先加进来再归一化，同时返回新残差——主干代码不用反复写 `x = x + sublayer(norm(x))`。

```
@torch.compile
def add_rms_forward(self, x, residual):
    orig_dtype = x.dtype
    x = x.float().add_(residual.float())           # 残差相加
    residual = x.to(orig_dtype)                    # 新残差 = 相加后的值
    var = x.pow(2).mean(dim=-1, keepdim=True)
    x.mul_(torch.rsqrt(var + self.eps))
    return x.to(orig_dtype).mul_(self.weight), residual
```

求方差在 fp32 里做，避免 bf16 下平方求和损失精度；残差相加与归一化写在同一个 `@torch.compile` 函数里，编译器可融合。**RoPE 只作用于 Q、K，不碰 V**（位置信息只需体现在注意力分数上），前向按位置查 `cos_sin_cache`，把最后一维前后两半配对旋转：

```
y1 = x1·cosθ − x2·sinθ
y2 = x2·cosθ + x1·sinθ
```

`get_rope` 带 `@lru_cache(1)`，28 层共享一份 `cos_sin_cache`。注意即使配置里有 `rope_scaling`，代码也只读 `rope_theta`，**没有实现 NTK/YaRN**。

SwiGLU 由「合并 Linear + 短激活」实现：`gate, up = x.chunk(2, -1); F.silu(gate) * up`。gate 与 up 合并成一次大矩阵乘再切，比两次小矩阵乘更能用满 GPU（输入只读一次、kernel 启动减半）。

TP 的切分：Column 切输出维（无需通信）、Row 切输入维（末尾一次 `all_reduce`），一前一后配对。每个 Transformer 层恰好 **2 次 all_reduce**（`o_proj` 与 `down_proj`），中间的激活在卡内保持切分状态。这是 Megatron-LM 的切法，`linear.py` 用 156 行实现。

> ⚠️ QKV 按头切，所以要求注意力头数与 KV 头数都能被 `tp_size` 整除。Qwen3-0.6B 有 16 个 Q 头、8 个 KV 头，TP 只能取 1/2/4/8。**在 nano-vllm 里 KV 头数就是 GQA 模型 TP 的上限**（vLLM 会在 TP 超过 KV 头数时复制 KV 头，nano-vllm 没实现）。

Attention 先写 Cache 再分两条路。写 Cache 用 Triton kernel：

```
@triton.jit
def store_kvcache_kernel(key_ptr, key_stride, value_ptr, value_stride,
                         k_cache_ptr, v_cache_ptr, slot_mapping_ptr, D: tl.constexpr):
    idx = tl.program_id(0)                  # 每个 program 处理一个 token
    slot = tl.load(slot_mapping_ptr + idx)
    if slot == -1:                          # -1 表示不需要写（padding）
        return
    key = tl.load(key_ptr + idx * key_stride + tl.arange(0, D))
    value = tl.load(value_ptr + idx * value_stride + tl.arange(0, D))
    tl.store(k_cache_ptr + slot * D + tl.arange(0, D), key)
    tl.store(v_cache_ptr + slot * D + tl.arange(0, D), value)
```

> 💡 为什么不用 `k_cache.view(-1, D)[slot_mapping] = key.view(N, D)`？问题在 `slot == -1`：CUDA Graph 回放时 decode batch 会补齐到固定大小，补出来的 token slot 是 -1，不能写。PyTorch 里要先掩码筛掉，**筛选结果长度随数据变化，不适合放进 CUDA Graph**。Triton kernel 每 program 处理一个 token，遇 -1 直接返回，形状始终固定。

阶段	调用	为什么
Prefill	`flash_attn_varlen_func`	多条变长序列拼在一起，用 `cu_seqlens_q/k` 切分；KV 已在 Cache 时带 `block_table`
Decode	`flash_attn_with_kvcache`	每条只新增 1 个 token（`q.unsqueeze(1)`），K/V 全来自 Cache，`cache_seqlens` 告诉 kernel 多长

触发带 `block_table` 的统一判断是 `cu_seqlens_k[-1] > cu_seqlens_q[-1]`（前缀命中或 chunked 续块，Q 比 K 短）。

Qwen3 的两处特殊：① QKV 投影后对 Q、K 做 per-head RMSNorm（QK-Norm），因为 norm 作用在最后一维 `head_dim` 上，reshape 成 `[token, 头, head_dim]` 送进去就是逐头归一化；② `packed_modules_mapping` 把 HF 分开的 q/k/v/gate/up 映射到合并后的 `qkv_proj` / `gate_up_proj`（字符串 shard_id 给 QKV，整数给 Merged）。另外 `tie_word_embeddings` 让 Embedding 与 LMHead 共享存储；`compute_logits` 单独调用 LMHead，**让 CUDA Graph 只需捕获到 hidden state 为止**。

Sampler 用 Gumbel 技巧：

```
@torch.compile
def forward(self, logits, temperatures):
    logits = logits.float().div_(temperatures.unsqueeze(dim=1))
    probs = torch.softmax(logits, dim=-1)
    noise = torch.empty_like(probs).exponential_(1).clamp_min_(1e-10)  # E ~ Exp(1)
    return probs.div_(noise).argmax(dim=-1)                            # argmax(p / E)
```

`argmax_i(p_i/E_i) ~ Categorical(p)`，`E_i iid ~ Exp(1)`。推导：`p_i/E_i` 最大对应 `E_i/p_i` 最小，而 `E_i/p_i` 服从参数为 `p_i` 的指数分布；一组独立指数变量里第 i 个最先到达的概率就是 `p_i`。全

是逐元素运算加一次 argmax，可被 `torch.compile` 融合，不需要 multinomial 的前缀和 + 二分。采样只在 rank 0 执行；nano-vllm 没有 top-k/top-p/重复惩罚。

3.4 KV Cache 与 BlockManager

```
class Block:
    def __init__(self, block_id):
        self.block_id = block_id
        self.ref_count = 0              # 几条序列在共享这块
        self.hash = -1                  # 块内容哈希（-1 = 还没填满）
        self.token_ids = []             # 块内 token（哈希命中后二次校验）
```

- 🔑 `ref_count`：共享时大于 1，减到 0 才能回收。**被共享的块都是已填满的块**，内容不再变，所以不需要 CoW——新 token 总写进序列自己独占的最后一块。
- 🔑 `hash`：只有填满的块才算哈希；没填满的 `-1`，不能复用（它还会追加 token）。
- `token_ids`：哈希命中后再逐个比对，防碰撞复用错块。

`Sequence` 是请求状态机（`WAITING` / `RUNNING` / `FINISHED`），关键字段：`block_table`（页表）、`is_prefill`（首次进 decode batch 时置 False，被抢占时重置 True）、`num_cached_tokens` + `num_scheduled_tokens`（支持 chunked prefill，两者之和等于 `num_tokens` 时 prefill 才算完）。

```
@property
def num_blocks(self):                       # 需要几个块（向上取整）
    return (len(self) + self.block_size - 1) // self.block_size

@property
def last_block_num_tokens(self):            # 最后一块装了几个 token
    return len(self) - (self.num_blocks - 1) * self.block_size
```

跨进程只传必要字段：TP>1 时每步要把这批 Sequence pickle 后经共享内存发给其他 rank。`__getstate__` 只打包 worker 需要的：

```
def __getstate__(self):
    last_state = self.last_token if not self.is_prefill else self.token_ids
    return (self.num_tokens, self.num_prompt_tokens, self.num_cached_tokens,
            self.num_scheduled_tokens, self.block_table, last_state)
```

decode 阶段 worker 收到的每条序列只有几个整数和 `block_table`——这也是 `prepare_decode` 读 `seq.last_token` 而非 `token_ids[-1]` 的原因。

3.4.1 分配、追加、回收

```
def _allocate_block(self):
    block_id = self.free_block_ids.popleft()
    block = self.blocks[block_id]
    assert block.ref_count == 0
    # 这块以前登记过哈希，且表里还指向它：要被覆盖了，先删登记
    if block.hash != -1 and self.hash_to_block_id.get(block.hash) == block_id:
        del self.hash_to_block_id[block.hash]
    block.reset()                  # ref_count = 1，清空 hash 与 token_ids
    self.used_block_ids.add(block_id)
    return block_id

def can_append(self, seq):
    # 只有「最后一个 token 正好落在新块第一格」时才需要一个空闲块
    return len(self.free_block_ids) >= (len(seq) % self.block_size == 1)

def may_append(self, seq):
    if len(seq) % self.block_size == 1:
        seq.block_table.append(self._allocate_block())

def deallocate(self, seq):
    for block_id in reversed(seq.block_table):   # 从最后一块往前放
        block = self.blocks[block_id]
        block.ref_count -= 1
        if block.ref_count == 0:
            self._deallocate_block(block_id)
    seq.num_cached_tokens = 0
    seq.block_table.clear()
```

> 🔑 `len(seq) % block_size == 1` 是本节的精华：上一步采样出的 token 已追加进序列，这一步要算它的 K/V。余数为 1 说明它是新块第一格，需要一个空闲块（`True` 参与比较等于 1）；否则当前块还有空位，0 个空闲块就够。
>
> ⚠️ 回收先减引用计数、减到 0 才回队列，被共享的前缀块不会因一条序列结束就被回收。`_deallocate_block` **不清 hash 与 token_ids，哈希表登记也还在**——在块被 `_allocate_block` 取走覆盖之前，后来的请求仍可命中。

两个顺序细节让缓存留得更久：空闲队列 FIFO（回收放队尾 → 最近释放的最晚被覆盖）；`deallocate` 倒序释放（后段块先被覆盖，**开头的块最可能是共享前缀，留得最久**）。

3.4.2 前缀缓存：哈希链 + 三条规则

```
@classmethod
def compute_hash(cls, token_ids, prefix=-1):
    h = xxhash.xxh64()
    if prefix != -1:
        h.update(prefix.to_bytes(8, "little"))
    h.update(np.array(token_ids).tobytes())
    return h.intdigest()
```

> 🔑 把前缀哈希拌进来：两个块内容相同但前面历史不同，就不是同一段上下文，不能复用。这与 Merkle 链思想一致。

登记在 `postprocess` 里完成（模型跑完、更新 `num_cached_tokens` 之前），只对**本轮新填满**的块算哈希（`[start, end)`）；chunked prefill 停在块中间时那块不算，等填满再登记。

```
def can_allocate(self, seq):
    h = -1
    num_cached_blocks = 0
    num_new_blocks = seq.num_blocks                # 最坏情况：每块都要占一个空闲块
    for i in range(seq.num_blocks - 1):            # 注意：最后一块不查
        token_ids = seq.block(i)
        h = self.compute_hash(token_ids, h)
        block_id = self.hash_to_block_id.get(h, -1)
        if block_id == -1 or self.blocks[block_id].token_ids != token_ids:
            break                                  # 第一次没命中就停
        num_cached_blocks += 1
        if block_id in self.used_block_ids:        # 命中在用的块：不占空闲块
            num_new_blocks -= 1
    if len(self.free_block_ids) < num_new_blocks:
        return -1
    return num_cached_blocks
```

规则	原因
最后一块永远不查	prefill 需要最后一个 token 的 logits 来采样；全命中就没有 token 进模型，拿不到 logits
前缀必须连续命中	第一次未命中就 break；中间断一块，后面的块依赖的前文 KV 已不在缓存
命中块分两种	在用的直接 `ref_count += 1`；已释放未覆盖的要从空闲队列摘出重新启用，仍占一个空闲名额
token_ids 二次校验	防 xxhash 碰撞复用错块

> 📌 粒度是一整块 + 最后一块不查，`block_size = 256` 时复用量会被截断：600 token 的公共前缀只能复用前 512 个（2 块），其余 88 个重算；512 token 的 prompt 与缓存完全相同，也只能复用第 1 块。**块越小匹配越细，但块表更长、哈希次数更多。**

3.5 Scheduler 与连续批处理

原文的领位员类比：门口排队的是 waiting，已落座点菜的是 running；领位员每轮先看门口还能不能安排，安排不了才照顾已落座的；桌子不够时请最后落座的那桌起身回门口重新排队。

```
class Scheduler:
    def __init__(self, config):
        self.max_num_seqs = config.max_num_seqs
        self.max_num_batched_tokens = config.max_num_batched_tokens
        self.eos = config.eos
        self.block_size = config.kvcache_block_size
        self.block_manager = BlockManager(config.num_kvcache_blocks, config.kvcache_block_s
        self.waiting = deque()    # 还没 prefill 完（新请求、chunked 未算完、被抢占）
        self.running = deque()    # prefill 已完成、正在逐 token decode
```

`schedule()` 返回 `(seqs, is_prefill)`，**prefill 优先**：只要 prefill 分支选出至少一条，本轮就做 prefill；一条都选不出来才做 decode。

```
def schedule(self):
    scheduled_seqs, num_batched_tokens = [], 0
    while self.waiting and len(scheduled_seqs) < self.max_num_seqs:
        seq = self.waiting[0]
        remaining = self.max_num_batched_tokens - num_batched_tokens
        if remaining == 0:
            break
        if not seq.block_table:                     # 首次调度：查前缀缓存，确认块够用
            num_cached_blocks = self.block_manager.can_allocate(seq)
            if num_cached_blocks == -1:
                break
            num_tokens = seq.num_tokens - num_cached_blocks * self.block_size
        else:                                        # chunked prefill 续算：块已分配
            num_tokens = seq.num_tokens - seq.num_cached_tokens
        if remaining < num_tokens and scheduled_seqs:
            break                                   # 预算不够且已有序列：收手
        if not seq.block_table:
            self.block_manager.allocate(seq, num_cached_blocks)
        # 允许对「本轮第一条」做 chunked prefill
        seq.num_scheduled_tokens = min(num_tokens, remaining)
        num_batched_tokens += seq.num_scheduled_tokens
        if seq.num_cached_tokens + seq.num_scheduled_tokens == seq.num_tokens:
            seq.status = SequenceStatus.RUNNING
            self.waiting.popleft(); self.running.append(seq)
        scheduled_seqs.append(seq)
    if scheduled_seqs:
        return scheduled_seqs, True
    # ...否则进入 decode 分支
```

出口条件	含义	代价
`can_allocate == -1` 就 break	队头块不够，**后面更短的请求也不调度**	严格 FCFS，队头阻塞
`remaining < num_tokens and scheduled_seqs`	预算不够且已有序列才收手	—

出口条件	含义	代价
`remaining == 0`	第一条被截断后预算用完，下轮退出	被截断的序列总是本轮唯一一条

> 📌 只有**本轮第一条**可被 chunked prefill 截断（源码注释 "only allow chunked prefill for the first seq"）。被截断的序列留在 waiting 队头，`block_table` 已整段分配、状态仍 WAITING，下轮按 `num_tokens - num_cached_tokens` 续算，直到最后一段才移进 running。

Decode 分支在空闲块不足时**抢占 running 队尾**（最近完成 prefill 的序列，已生成 token 通常最少）：

```
    while self.running and len(scheduled_seqs) < self.max_num_seqs:
        seq = self.running.popleft()
        while not self.block_manager.can_append(seq):
            if self.running:
                self.preempt(self.running.pop())
            else:
                self.preempt(seq); break           # 没有别人可牺牲，牺牲自己
        else:
            seq.num_scheduled_tokens = 1
            seq.is_prefill = False                 # 第一次进 decode batch 时置 False
            self.block_manager.may_append(seq)
            scheduled_seqs.append(seq)
    self.running.extendleft(reversed(scheduled_seqs))
    return scheduled_seqs, False

def preempt(self, seq):
    seq.status = SequenceStatus.WAITING
    seq.is_prefill = True
    self.block_manager.deallocate(seq)    # 释放全部 KV 块，num_cached_tokens 清零
    self.waiting.appendleft(seq)          # 插回队头，下轮优先
```

> 📌 抢占是**重计算**：直接释放 KV、打回 waiting，下次把 prompt + 已生成 token 当一整段重新 prefill。**重算量通常比整段小**：`deallocate` 只放回块、不清哈希，只要没被覆盖，重新 prefill 时就能命中前缀缓存。
>
> ⚠️ `while ... else` 的 `else` 只在循环条件变假（`can_append` 成功）时执行；牺牲自己的分支以 `break` 结束，所以那条序列不会进入本轮 batch。

`postprocess`：

```
def postprocess(self, seqs, token_ids, is_prefill):
    for seq, token_id in zip(seqs, token_ids):
        self.block_manager.hash_blocks(seq)      # 本轮新填满的块登记哈希
        seq.num_cached_tokens += seq.num_scheduled_tokens
        seq.num_scheduled_tokens = 0
        if is_prefill and seq.num_cached_tokens < seq.num_tokens:
            continue                             # chunked 还没算完：丢弃本轮采样结果
        seq.append_token(token_id)
        if (not seq.ignore_eos and token_id == self.eos) \
           or seq.num_completion_tokens == seq.max_tokens:
            seq.status = SequenceStatus.FINISHED
            self.block_manager.deallocate(seq)
            self.running.remove(seq)
```

> 🔑 **chunked prefill 的中间段也会采样**：模型照常对这一段最后一个位置算 logits、采样一个 token，但那个位置不是 prompt 末尾，结果无意义，`continue` 直接丢掉。只有最后一段算完，采样出的才是真正的第一个生成 token。

维度	nano-vllm	vLLM V1
阶段	严格分离，不混批	无阶段概念，统一 `{req: num_tokens}`
Chunked Prefill	只对本轮第一条生效	任何请求都可被预算截断
抢占	重计算，踢 running 队尾	重计算，FCFS/PRIORITY

3.6 ModelRunner、CUDA Graph 与基准

```
def run(self, seqs, is_prefill):
    input_ids, positions = self.prepare_prefill(seqs) if is_prefill else self.prepare_decod
    temperatures = self.prepare_sample(seqs) if self.rank == 0 else None
    logits = self.run_model(input_ids, positions, is_prefill)   # eager 或 CUDA Graph 回放
    token_ids = self.sampler(logits, temperatures).tolist() if self.rank == 0 else None
    reset_context()
    return token_ids
```

`__init__` 顺序固定：NCCL 进程组 + 绑卡 + dtype → 构造 `Qwen3ForCausalLM` 并加载 → `warmup_model`（最大规模 prefill 测激活峰值）→ `allocate_kv_cache` → 非 eager 时 `capture_cudagraph` → TP>1 建共享内存并进 loop。

3.6.1 张量化

张量	Prefill	Decode	作用
`cu_seqlens_q`	本轮要算的 Q 长度累积	—	varlen 边界

张量	Prefill	Decode	作用
`cu_seqlens_k`	从序列开头到 end 的累积	—	有历史时 K 比 Q 长
`slot_mapping`	`物理块号×block_size + 块内偏移`	同左	新 K/V 写到哪
`context_lens`	—	含新 token 的总长	读多少历史 KV
`block_tables`	仅当 K 比 Q 长	总是需要	从哪些物理块读

> ⚠️ `block_tables` 右侧补的 `-1` 只用于对齐矩阵，kernel 只按 `context_lens` 读有效块；它与 `store_kvcache` 跳过的 `slot == -1` 不是一回事，后者来自 CUDA Graph 的补齐 batch。

`cu_seqlens`、`slot_mapping`、`block_tables` 这些张量放进一个全局 `Context`（`utils/context.py`），算子内部 `get_context()` 现取——于是 `Qwen3.forward` 的签名只有 `input_ids` 与 `positions`。

> 💡 这是用全局变量换代码整洁：代价是算子隐式依赖全局状态，每次 forward 前必须 `set_context`，`run` 末尾 `reset_context`。前提是推理循环单线程串行、每 rank 一进程。

3.6.2 KV Cache 怎么切

```
def allocate_kv_cache(self):
    free, total = torch.cuda.mem_get_info()
    used = total - free                                                    # 整卡已占用
    peak = torch.cuda.memory_stats()["allocated_bytes.all.peak"]           # warmup 期间峰值
    current = torch.cuda.memory_stats()["allocated_bytes.all.current"]
    num_kv_heads = hf_config.num_key_value_heads // self.world_size
    block_bytes = 2 * num_layers * block_size * num_kv_heads * head_dim * dtype.itemsize
    config.num_kvcache_blocks = int(total * gpu_memory_utilization - used - peak + current)
    self.kv_cache = torch.empty(2, num_layers, num_kvcache_blocks, block_size, num_kv_heads
    # 再把每层切片绑定到对应 Attention 的 k_cache / v_cache
```

```
num_blocks  = floor( (total × util − used − (peak − current)) / block_bytes )
block_bytes = 2 × L × block_size × (H_kv / tp) × d_head × itemsize
```

`used` 是驱动看到的整卡占用（权重、CUDA 上下文、PyTorch 缓存池，甚至同卡其他进程）；`peak - current` 是 warmup 时激活比当前多占的部分，正式运行还会再出现，要提前留出。

以 **Qwen3-0.6B** 为例（28 层、8 个 KV 头、`head_dim=128`、bf16、TP=1）：

```
block_bytes = 2 × 28 × 256 × 8 × 128 × 2 = 29,360,128 B = 28 MiB / 块
per_token   = 28 MiB / 256 ≈ 112 KiB/token
每 1 GiB 显存 ≈ 36 块 ≈ 9,200 个 token
```

> 📌 KV Cache 是一个 **6 维张量** `(2, 层数, 块数, 块大小, KV 头数, head_dim)`，第 0 维分 K/V。**显存余量按块全部切给 KV，不按请求预留**，能容纳的序列数由实际长度决定。dtype 跟模型默认一致，nano-vllm **没有实现 KV 量化**。

3.6.3 CUDA Graph：只为 decode 捕获

- 🔑 **档位**：`max_num_seqs = 512` 时是 1、2、4、8 + 16 到 512 每隔 16，共 **36 张图**。
- 🔑 **从大到小捕获**：最大图的内存池通过 `graph.pool()` 传给小图复用，36 张图不各占一份中间激活。
- 🔑 **捕获前先跑一遍**：`@torch.compile` 遇新形状可能重编译，Triton kernel 首次调用还 JIT，这些不能发生在捕获过程中。
- 图里只到 hidden state，`compute_logits` 与采样在图外。

运行时按实际 batch 向上取档，把数据拷进静态缓冲区再回放。补齐位 `slot_mapping = -1`（不写 Cache）、`context_lens = 0`（不读历史），结果在 `outputs[:bs]` 切片时丢弃。超过 512 退回 eager。

> ⚠️ 真实的坑：`max_num_seqs` 小于 512 且不是 16 的倍数（大于 8）时，最大档位小于 `max_num_seqs`。比如设 100，档位最大 96，batch 97–100 时 `next(...)` 找不到档位，抛 `StopIteration`。默认 512 不会遇到。

3.7 本章小结

- 四层结构：接口层 → 引擎层 → 调度层 → 执行层；调用自上而下，KV 与 logits 自下而上回流。
- `kvcache_block_size = 256` 来自 flash-attn 分页 KV 的要求，不是碎片与开销的折中。
- 前缀缓存 = 哈希链 + 满块才登记 + 最后一块不查 + 连续命中即停，块粒度导致复用量被截断。
- 调度是 prefill 优先、不混批；只有本轮第一条可被切块；抢占用重计算，靠前缀缓存减少重算。
- ModelRunner 用全局 Context 换签名整洁；KV 是 6 维张量按余量切分；CUDA Graph 只捕获 decode，向上取档 + `slot_mapping=-1` 处理补齐位。

第 4 章 量化：把「搬不过来」的字节变少

> 对应 AIInfraGuide 模块四 第 4 章（4.1～4.6，六篇正文）。第 1 章证明了 Decode 卡在搬权重，第 2 章把显存利用到极致。这一章走第三条路：**让要搬的东西变少**。原文的比喻：量化像换一把刻度更少的尺子——原来能读到毫米，现在只能读到厘米；数据量小了，读数也粗了。要回答三件事：刻度怎么定（Scale 与零点）、多少数据共用一把尺子（粒度）、怎样让模型少受误差影响（校准与 PTQ/QAT）。

4.1 量化基础

4.1.1 三类量化对象

对象	特点	主要收益	主要难点
权重	训练完固定，可离线慢慢处理	模型变小，每步读的权重变少	4-bit 只有 16 个编码，误差大
激活	随输入变化，只能在线处理	让矩阵乘走低精度 Kernel	通道离群值，Scale 要在线求
KV Cache	随请求与上下文增长	每个历史 token 占的缓存变少	必须便宜、能流式追加

记号：`W4A16` = 权重 4-bit、激活 16-bit；`W8A8` = 两者都 8-bit。**这个记号只描述线性层的两个输入，KV Cache 的精度要另外说明**——初学者最常混的一点。粗账：7B 全 16-bit 约 14 GB，全 4-bit 约 3.5 GB（十进制，未计量化参数）。

4.1.2 仿射量化

```
q  = clip( round(x / s) + z, q_min, q_max )
x̂  = s (q − z)
```

`s > 0` 是 Scale（相邻两格代表的数值间隔），`z` 是 Zero-point（实数 0 落在哪个整数格）。取 `s=0.1, z=0`：`x=0.26` 记为 `q=3`，读回 0.3，舍掉的 0.04 找不回来。换成 `s=1/16`，同一个 0.26 记为 4、读回 0.25。**「保留一位小数」只是 `s=0.1` 的巧合**；`s=0.1` 时 0.251、0.26、0.349 都映射为 3。`round` 规则由实现决定，如 ONNX 的整数 `QuantizeLinear` 用「四舍六入五成双」。

量化误差只有两类：**舍入**（被拉回最近刻度）与**截断**（超出量程被压到端点）。

方式	公式	优点	代价
对称（`z=0`）	`s = max_abs(x) / 127`，`q ∈ [-127,127]`	反量化只乘一个 Scale，GEMM 无需零点修正	数据偏移时浪费一半编码
非对称	`s = (b-a)/(q_max-q_min)`，`z = round(q_min − a/s)`	量程贴着数据走	多存零点、多一步修正；全零张量要特判

非对称例：`[a,b] = [-1,3]`、编码 `[0,15]` → `s = 4/15, z = 4`；`x=1` 时 `q = round(3.75)+4 = 8`，反量化 `4 × 4/15 ≈ 1.067`。零点取整会带来两端的小偏移。

4.1.3 位宽与粒度

把 `[0,1]` 均匀量化：

位宽	编码数	格距	最大舍入误差
8-bit	256	1/255 ≈ 0.00392	≈ 0.00196

位宽	编码数	格距	最大舍入误差
4-bit	16	1/15 ≈ 0.06667	≈ 0.03333
2-bit	4	1/3 ≈ 0.33333	≈ 0.16667

8-bit 到 4-bit 格距变成约 **17 倍**。所以低比特方案几乎都要配合更细粒度或更聪明的误差补偿——GPTQ、AWQ、KIVI 都在解决这件事。

粒度是「多少数据共用一把尺子」。本章统一用 `Y = XW`，`X` 的行是 token、列是输入通道；PyTorch 的 `nn.Linear.weight` 形状是 `C_out × C_in`，正好是 `W` 的转置，**读代码别把行列弄反**。

```
小激活矩阵（3 token × 4 通道）：
X = [ 0.2  0.5  8.0  -0.1 ]
    [ 0.3  0.4  9.0  -0.2 ]
    [ 0.1  0.6  7.0  -0.1 ]
第 3 列在每个 token 上都很大 → 通道离群值（4.2 节 SmoothQuant 的出发点）
```

粒度	共享范围	典型用途	代价
Per-tensor	整个张量	静态激活、FP8 权重	一个离群值拖累全体
Per-channel	每个输出通道	INT8 权重	Scale 数 = 输出通道数
Per-group	每 g 个连续元素（如 128）	INT4 权重	Scale 更多，Kernel 要分组反量化
Per-token	每个 token	动态激活	运行时要先求每行最大值

> 📌 **粒度只能沿某个轴切分，切的方向和离群值方向不一致时，再细也帮不上忙**。对上例，Per-token 给每行单独定量程，但同一行里 8.0 与 0.2 仍共用一把尺子。激活的离群值沿通道分布，而矩阵乘要求激活的 Scale 能从求和里提出来（只能按 token 分），**这就是 W8A8 难做的根源**。

4.1.4 校准与 PTQ/QAT

Scale 由量程决定。量程取到最大值，离群值放得下但大部分数据挤在少数格子里；量程收窄，大部分数据更细但超出部分被截断。**校准就是在舍入与截断之间找平衡**：取最大值、取百分位，或以 MSE/输出误差为目标搜截断点。

权重固定，可直接统计；激活要喂真实样本，这批样本叫**校准集**，应覆盖线上语言、Chat Template 与输入长度，**并与评测集分开**，否则评测偏乐观。

方式	做法	优点	缺点
静态	校准集离线算好 Scale	推理开销最小	分布变了就不准
动态	推理时实时求 Scale	更贴合输入	每次都要先求最大值

> 💡 「动态」本身不说明粒度。看到自称动态量化时，要再确认三件事：**Scale 什么时候算、沿哪个轴算、作用在哪些算子上。**

维度	PTQ	QAT
需要什么	训练好的模型 + 少量校准数据	训练流程、数据与算力
控制误差	选 Scale、截断点，或逐层补偿（GPTQ/AWQ）	让参数在训练中适应量化
适合	8–bit、4–bit 权重等误差可控场景	更低比特，或 PTQ 不达标
仍需验证	目标任务精度	精度 + 推理 Kernel 是否支持该格式

QAT 的前向用 **Fake Quant**（先量化再立刻反量化，数据仍是浮点但带上舍入误差）；`round` 导数几乎处处为 0，反向用 **STE** 近似（量程内原样传梯度，量程外通常置 0）。**PTQ 不等于不需要数据；QAT 也不是在整数上训练。**

4.2 W8A8 与 SmoothQuant

4.2.1 W8A8 难在哪里

W8A8 特指 INT8 权重 × INT8 激活，线性层可直接走 INT8 GEMM。假设一个 token 的隐藏向量大部分在 `[-1,1]`，某通道却到 80：按整行最大值求 Scale，`s = 80/127 ≈ 0.63`，`[-1,1]` 里的数只能落到 −2 到 2 这几个编码。**Per-token 隔开了 token 之间的差异，却管不了同一 token 内的通道差异。**

SmoothQuant 的观察：大值集中在**少数固定通道**上、跨 token 反复出现；权重分布平坦得多。所以可以把激活的量化难度分一部分给权重。原文的挑担子类比：激活这头太重，就把一部分重量挪到权重那头，两头都不太重。

4.2.2 等价缩放

取对角矩阵 `S = diag(s_1, …, s_C_in)`，所有 `s_j > 0`：

$$Y = XW = (X\,S^{-1})(S\,W) = \tilde{X}\,\tilde{W}$$

第 j 个输入通道：`X` 的第 j 列除以 `s_j`，`W` 的第 j 行乘以 `s_j`，精确算术下正好抵消。`S` 只有 `C_in` 个系数，**与 token 数无关，可离线算好**。

```
平滑前：x = [1, 1, 16]   w = [2, 2, 0.5]   x·w = 12
平滑后：x' = [1, 1, 2]   w' = [2, 2, 4]    x'·w' = 12
              ▲ 第 3 通道 ÷8       ▲ 第 3 通道 ×8
激活最大值 16 → 2，权重最大值 2 → 4：两个输入的量程都温和了
```

> 📌 平滑后 `X̃`、`W̃` 仍是浮点，**还没有任何 INT8 编码**。平滑本身不引入误差，它让下一步量化的误差变小。**SmoothQuant W8A8 = 平滑 + 量化，两步不能混为一谈。**

4.2.3 迁移强度 α

```
s_j = (max_t |X_tj|)^α / (max_k |W_jk|)^(1-α),     0 ≤ α ≤ 1
```

激活最大值用校准样本统计，权重最大值直接算；实现上要防分母接近 0。

α	含义	出处
0.5	OPT、BLOOM 系列的通用「甜点」	论文
0.75	激活更难量化的 GLM-130B	论文
< 0.4	激活难量化（消融，针对其模型）	论文
> 0.6	权重难量化（消融）	论文
0.8+	后续 LLaMA 系列常用	论文后续版本

合适的值要按模型在一小批校准数据上搜索（论文在 Pile 子集上网格搜索）。vLLM 文档的 LLM Compressor 示例用 `SmoothQuantModifier(smoothing_strength=0.8)`，`smoothing_strength` 就是 α。

> ⚠️ 这里有**两种 Scale**：`s_j` 是平滑系数（只做等价变换）；INT8 的量化 Scale 在平滑之后按新量程计算。混起来公式就读不懂了。

4.2.4 从变换到 INT8 GEMM

离线四步：① 用有代表性的校准文本跑前向，记录每个待量化线性层输入通道的最大值；② 算并折叠平滑系数（`S⁻¹` 可并进前面的 Norm 缩放参数，`S` 直接乘进权重）；③ 量化（选粒度、静态或动态、验证逐层与整模型误差）；④ 导出真实 INT8 权重与配置交给 Kernel。

折叠有条件：一个 Norm 的输出若同时送给 Q、K、V 三个投影，**三者必须用同一组 S**；残差、非线性处不能随意移动缩放。

```
recipe = [
    SmoothQuantModifier(smoothing_strength=0.8),
    GPTQModifier(targets="Linear", scheme="W8A8", ignore=["lm_head"]),
]
```

运行时（激活按 token、权重按输出通道、零点为 0）：

```
X̂_tj = a_t · Q^X_tj,  Ŵ_jk = b_k · Q^W_jk
Ŷ_tk = a_t b_k Σ_j Q^X_tj Q^W_jk
        └─ INT8 乘、INT32 累加，最后乘回 a_t·b_k 转 FP16/BF16
```

> 📌 W8A8 **只描述矩阵乘的两个输入**，累加与输出仍是高精度。这也解释了 4.1 的结论：激活 Scale 若按通道设就提不出求和号，**所以只能靠平滑把通道差异提前消掉**。

4.2.5 收益与边界

论文报告在其测试模型上最高提速 **1.56 倍**、显存减半，精度损失可忽略。整模型能否提速取决于 INT8 GEMM 省下的时间能否覆盖在线量化、Scale 处理与类型转换的开销；矩阵很小、算子没融合、或时间主要耗在 Attention 与调度上时收益有限。

vLLM v0.32.0 文档注明：INT8 W8A8 计算支持 **Turing 到 Hopper，不支持 CC 10.0 及以上的 Blackwell**，这类卡应改用 FP8。

4.3 Weight-only INT4

4.3.1 W4A16 的存储与计算

激活保留 16-bit，只把权重压到 4-bit。Decode 每步要把整份权重读一遍，**权重变成 1/4，读的时间也跟着少**。

INT4 权重**打包存放**：一个 32-bit 字装 8 个编码，另存每组 Scale（非对称还有零点）。推理时 Kernel 读出编码、**在片上反量化成 FP16/BF16**，再与 16-bit 激活相乘。

> 📌 所以「INT4 模型」要分清两件事：**权重的存储类型是 INT4，乘法指令的输入类型仍是 16-bit**。这条路径省的是读权重的带宽，**不是乘法的算力**。

分组参数也占空间。每 128 个权重共用一个 FP16 Scale 时，平均每权重 `4/8 + 2/128 = 0.515625 byte`，再加零点与对齐会略多。

4.3.2 GPTQ：边量化边补偿

只最小化 $\|W - \hat{W}\|^2$ 会忽略「误差被输入放大或抵消」。GPTQ 用校准输入把目标换成**层输出的重建误差**：

$$\min_{\hat{W}} \|XW - X\hat{W}\|_F^2$$

取一列 w，$\delta = w - \hat{w}$：$E = \|X\delta\|^2 = \delta^T X^T X \delta$，$H = 2X^T X$

H 刻画在真实输入下每个方向的误差要付多大代价，以及输入通道间的相关性；它只来自单个线性层的重建目标，**不需要整个语言模型损失的 Hessian**。

补偿直觉：拼图时一块放歪了，可以挪旁边还没固定的块来弥补。两维例子：若校准输入两列几乎相同，输出主要取决于 `w1+w2`；`w1 = 0.26` 量化成 0.30 多了 0.04，趁 `w2` 还没量化让它减 0.04，输出就基本不变。一般情形的最优补偿：

$$\Delta w_F = -(w_i - \hat{w}_i) / [H_F^{-1}]_{ii} \cdot [H_F^{-1}]_{:,i}$$

第 i 项正好把 `w_i` 变成 `ŵ_i`，其余项给出剩余权重的调整量。`ŵ_i` 是反量化后的值（与原权重同尺度），不是整数编码。官方实现还加阻尼项保证数值稳定，用 Cholesky 与分块更新降成本。

> ⚠️ **两个「128」别搞混**：更新块大小决定一次处理多少列、多久批量传播误差；Group Size 决定多少权重共享一个 Scale。

论文报告 GPTQ 量化 175B 的 GPT 模型约需 **4 个 GPU 小时**，降到 3–4 bit 精度损失可忽略，第一次让 175B 能在单卡运行。量化是离线一次性成本，**推理时不再需要 H**。

4.3.3 AWQ：按激活找要保护的通道

对 `y = Σ x_j w_j`，权重误差的影响是 `x_j Δw_j`——**一个权重要不要紧，取决于和它相乘的激活有多大，而不是它自己有多大**。AWQ 指出：只保护 **1%** 的显著权重就能大幅减小误差；找显著通道要看**激活分布**。

最直接的保护是让这 1% 保持 FP16，但混合精度布局对 Kernel 不友好。AWQ 改用**缩放**（量化前放大显著通道权重 `s_j` 倍，输入相应除以 `s_j`）：

```
Ŷ = (X S⁻¹) · D( Q(S W) )        Q 生成低比特编码，D 用 Scale 与零点反量化
```

放大后，只要该通道所在组的格距没有同比例变大，除回 `s_j` 后有效误差就变小。但放太大会撑宽整组量程、误伤同组其他权重，所以要在校准数据上搜索缩放因子，再配合截断控制总误差。部署出来仍是**规则 INT4 布局**，高效 Kernel 容易支持；论文配套的 TinyChat 在桌面与移动 GPU 上比 HF FP16 快 **3 倍以上**。

方法	处理什么	缩放方向	典型配置
SmoothQuant	激活离群值让 INT8 激活难量化	压平激活，难度转给权重	INT8 W8A8
AWQ	显著通道的权重误差被大激活放大	放大显著权重，降其相对误差	INT4 W4A16

> 📌 比较方法时要分清三种中间产物：**缩放后的浮点权重**、**模拟量化（QDQ）后的浮点权重**、**真正编码打包的部署权重**。只做缩放搜索得到的还是浮点权重。

4.3.4 Marlin：让解包不拖慢计算

最笨的做法是先把整层 INT4 展开成 FP16 写回显存再调 GEMM——多一次大规模读写，**省下的带宽全吐回去**。Marlin 把读权重、解包、反量化与矩阵乘放进同一个 Kernel 的流水线：

```
朴素:   INT4 ──展开──► 写回显存(FP16) ──读回──► GEMM     ← 多一次往返
Marlin: INT4 ──► [片上解包+反量化] ──► GEMM               ← 搬运与计算重叠
```

权重事先重排成适合运行时读取的布局，用 Tiling 与多级缓冲把搬运与计算重叠。论文报告：Batch 16–32 以内能接近 **4 倍**理论上限，64–128 时递减但仍明显；接入 vLLM 后端到端最高约 **2.8 倍**。

> 📌 **Batch 一大，计算重新成为瓶颈，少读权重的优势被摊薄**——与 1.3 的 Roofline 分析一致。这是「权重量化对大 Batch 增益有限」的根本原因。

4.3.5 vLLM 的 Kernel 选择

v0.32.0 中**方法名与 Kernel 已解耦**：`--quantization` 写 `awq` / `awq_marlin` / `auto_awq` 都解析为同一个 `auto_awq` 配置（GPTQ 同理）；不写时按检查点的 `quant_method` 自动识别。具体 Kernel 由 vLLM 按优先级检查：CUDA 上 INT4 W4A16 的候选依次是 **Machete**（仅 Hopper，CC 9.0）、**Marlin**（Turing+）、Conch、Exllama、Triton，第一个通过检查的胜出。启动日志会打印 `Using MarlinLinearKernel for AutoAWQMarlinLinearMethod`，AWQ 不满足条件时提示 `Falling back to unoptimized AWQ kernels`。可用 `VLLM_DISABLED_KERNELS` 排除某个 Kernel 做对比。

> 🔑 **GPTQ/AWQ 决定编码的误差，Kernel 决定编码跑得多快。看启动日志，不要只凭模型名字。**

4.3.6 公平比较 GPTQ 与 AWQ

先拆四个维度——「GPTQ、INT4 W4A16、Group Size 128、Marlin」回答了四个不同问题：

维度	例子	回答什么
量化算法	GPTQ 或 AWQ	怎样控制量化误差
格式与位宽	INT4 W4A16	权重怎么编码，激活保留什么精度
量化粒度	Group Size 128	多少权重共享一组 Scale
执行实现	Marlin、Machete	编码怎样被读取和计算

维度	GPTQ	AWQ
校准数据用途	构造 H，衡量层输出误差	统计激活，搜索通道缩放
主要手段	逐列量化，补偿剩余权重	放大显著通道，降其相对误差
离线计算	逐层求解重建问题	搜索每通道缩放因子
推理速度	取决于导出布局与 Kernel	同样取决于布局与 Kernel

> 📌 「AWQ 一定更快」「GPTQ 一定更准」都缺少前提。公平对比至少要固定基础模型、位宽、Group Size、校准集、输入长度与负载，并记录实际 Kernel。

4.4 KV Cache 量化

4.4.1 先估算能省多少

```
M_KV = 2 · L · B · T · H_kv · d · b        （B 请求数、T 缓存长度、b 每元素字节）
```

取 `L=32, H_kv=8, d=128`，单请求 32768 token：

精度	单请求 KV	备注
16-bit	4 GiB	基线：`2×32×32768×8×128×2 B`
8-bit	≈ 2 GiB	FP8 / INT8
2-bit	≈ 0.5 GiB	只是编码下限，未算 Scale/零点/残留区/对齐

> 📌 两件事可叠加：**PagedAttention 让已有空间少浪费，量化让每个元素变小**。权重量化不会自动改变这里的 `b`，KV 精度要单独配置。

4.4.2 为什么不能照搬权重量化

差异	权重	KV Cache
时间约束	可离线花几小时，处理一次反复用	执行中不断产生，每 token 写一份
方法要求	可以很重（二阶信息、迭代求解）	必须便宜、能**流式追加**
误差位置	直接进 GEMM	K 的误差先改变 QK^T 再过 Softmax，可能关注错位置；V 的误差进入 `PV` 加权后影响输出

原文的类比：权重量化是会前把资料印成小字版，可以慢慢排版；KV 像会议纪要，只能现场速记。**所以 KV 量化的效果必须放回长上下文任务里验证**——几条短对话正常，说明不了长文档检索。

4.4.3 KIVI：K 按通道，V 按 Token

KIVI 是**免调参的 2-bit KV 量化算法**。观察是：在其研究的模型中 **Key 有明显通道离群值、Value 没有**，所以 K 按通道量化、V 按 Token 量化。



Key 竖切把大幅值通道单独隔开；Value 横切让每个 token 用自己的范围。这在 2-bit 下格外关键：每组只有 4 个编码，大小悬殊的数挤在一组，小值几乎都落到同一个编码。论文标题里的 **Asymmetric** 指 K/V 用不同策略；组内则是基于 min/max 的非对称映射——「怎么分组」与「组内怎么编码」是两个问题。

新 token 凑不满一组：Key 的组跨多个 token。KIVI 保留一段高精度 **Residual Cache**，与已量化历史一起参与 Attention；新 token 攒够再成批量化移入历史区。Value 按 token 量化不用凑组，它的残留区只保留最近若干 token，像一个滑动窗口。残留区让流式写入可行，**但占额外空间**：上下文越短，残留区与元数据占比越高，实际压缩比离「8 倍」越远。

论文在 Llama/Falcon/Mistral 上报告：峰值显存（含权重）降低 **2.6 倍**，Batch 可放大 **4 倍**，实际推理负载下吞吐提升 **2.35–3.47 倍**。

> ⚠️ KIVI 是论文方法，有自己的实现，**不等于 vLLM 默认的 KV 路径**；2-bit 是论文在其模型与任务上验证的配置，换模型要重新评测。

4.4.4 vLLM 的 FP8 KV Cache

工程上更常用的是把 KV 存成 FP8：数据量减半，精度压力比 2-bit 小，也不需要特殊缓存结构。

`kv_cache_dtype`	含义	支持（v0.32.0）
`auto`	与模型 dtype 一致	默认
`fp8` / `fp8_e4m3`	E4M3，精度更高	CUDA 11.8+ 与 ROCm
`fp8_e5m2`	E5M2，范围更大、精度更低	CUDA 11.8+

源码的 `CacheDType` 还有 `int8_per_token_head`、`fp8_per_token_head`、`nvfp4` 等，依赖特定 Attention 后端。DeepSeek-V3.2 等少数模型**默认就用 FP8 KV**。

FP8 也要 Scale，两种粒度：**Per-tensor**（Q/K/V 各一个）与 **Per-attention-head**（`q_scale` 有 `num_heads` 个、`k/v_scale` 有 `num_kv_heads` 个；只有 Flash Attention 后端支持，且必须用 LLM Compressor 校准）。

Scale 的来源两条：① **不校准**——只设 `kv_cache_dtype="fp8"`，所有 Scale 取 1.0，检查点里没有 Scale 时日志会警告 `Using KV cache scaling factor 1.0 for fp8_e4m3`；② **用数据集校准**（文档推荐）——用 `QuantizationModifier` 配 `kv_cache_scheme` 做一次 oneshot，文档示例用 512 条样本、最大长度 2048，`strategy` 可选 `"tensor"` / `"attn_head"`。

```
llm = LLM(model="Qwen/Qwen2.5-7B-Instruct", dtype="bfloat16",
          max_model_len=4096, kv_cache_dtype="fp8")   # 未校准：Scale 全为 1.0
```

> ⚠️ 旧版本里用随机 token 估 Scale 的 `calculate_kv_scales` 参数**在 v0.32.0 已不存在**。只剩「取 1.0」与「读检查点校准值」两种情况，**能跑通不等于已校准**。

敏感层可跳过：`--kv-cache-dtype-skip-layers sliding_window`（也可写层号 `0 1 23`）。

存储精度 ≠ 计算精度：FP8 KV 首先减少存储；读出来后后端可能转回高精度再算，也可能直接在 FP8 域做 Attention。文档注明 Flash Attention 3 后端配合 FP8 KV 时，Attention 在 FP8 域计算，**Query 也会被量化成 FP8**。只看 `kv_cache_dtype` 不能断定整段 Attention 的精度。

4.5 FP8 与 NVFP4/MXFP4

4.5.1 同一串 bit，两种含义

`00111000` 按有符号 INT8 读是 56；按 E4M3 读（符号 0、指数 0111、尾数 000）是 1.0。**把 INT8 编码的类型标签改成 FP8，数值就全变了**；格式转换必须从原值重新编码并重算 Scale。

比较项	均匀整数量化	低比特浮点
编码含义	整数 + Scale + 零点	符号/指数/尾数 + 外部 Scale
固定 Scale 后间距	处处相等	随指数区间变化
误差特点	绝对误差上界统一	正规数范围内相对误差大致统一
计算	需要整数输入的 Kernel	需要对应浮点格式的 Kernel

格式	符号/指数/尾数	最大有限值	特殊值	取舍
E4M3	1/4/3	±448	只有 NaN，无 Inf	精度更高，范围较小
E5M2	1/5/2	±57344	±Inf 与 NaN	范围更大，精度更低

E5M2 遵循 IEEE 754 特殊值约定；E4M3 不表示无穷大、只留一个尾数模式给 NaN，从而把范围扩到 448。推理中权重与激活一般用 **E4M3**；论文建议 E5M2 用于训练梯度。E4M3 在 `[1,2)` 内相邻间隔 `1/8 = 0.125`，到 `[8,16)` 变成 1——**它保住的是相对精度**，不同于 INT8 的固定步长。

> 📌 比较格式必须交代 Scale。Scale 为 1 时 E4M3 能精确表示 1.375，INT8 只能取 1；但给 INT8 选 `s = 0.125` 它同样能精确表示 1.375，代价是最大值降到 15.875。**只拿一个数的误差，推不出哪种格式整体更好。**

4.5.2 FP8 仍需要 Scale

```
q = cast_FP8(x / s),   x̂ = s · q,   s = max|x| / F_max      (E4M3 的 F_max = 448)
```

指数范围仍有限，合适的 Scale 能减少掉进次正规区、下溢成 0 或溢出。**在正规数范围内，只改变 2 的幂次的 Scale 基本不提高相对精度**，Scale 的作用主要是让分布落进可表示范围。全零张量单独处理。

方案	权重	激活
`fp8_per_tensor`	E4M3，每张量一个 FP32 Scale	每张量一个；Ada/Hopper 上线性层激活改用 Per-token
`fp8_per_channel`	E4M3，每输出通道一个	动态 Per-token
`fp8_per_block`	E4M3，每 128×128 块一个	每 1×128 块一个

FP8 计算需要 CC 8.9 及以上（Ada/Hopper/Blackwell）；Turing、Ampere 上 FP8 模型也能跑，但走 **FP8 Marlin 的 W8A16**（只有权重是 FP8，乘法仍 16-bit）。文档称 FP8 可让显存减半、吞吐最高提升 **1.6 倍**。

4.5.3 FP4 与微块缩放

E2M1 的非负幅值只有 8 个：`0, 0.5, 1, 1.5, 2, 3, 4, 6`（0.5 是次正规；最大是 6 不是 8，因为只有 1 位尾数）。若整层几百万个数共用一个 Scale，局部差异几乎全丢，于是有 **Micro-block Scaling**：每一小组共享一个 Scale。原文的比喻：一张大合影只能统一曝光，拆成多张小照片就能每张单独调亮度。

Per-group 与 Block-wise 的叫法不统一，Block 可能一维也可能二维。以 `W ∈ R^(256×512)`（每行一个输出通道）为例：

配置	共用 Scale 的元素	Scale 数量
Per-group，G=128	同一行内连续 128 个	256 × (512/128) = 1024
二维 Block，128×128	128 行 × 128 列	(256/128) × (512/128) = 8

比较粒度时要写出张量布局、分组轴和组形状；GPTQ 的更新块、GEMM 的 Tile 属于执行过程，不决定 Scale 的共享范围。

4.5.4 NVFP4 与 MXFP4

比较项	MXFP4	NVFP4
来源	OCP MX 规范（多家厂商）	NVIDIA，随 Blackwell 推出
数据元素	E2M1	E2M1
块大小	32 个元素	16 个元素
块 Scale	E8M0，只能是 2 的幂	E4M3，带 3 位尾数
张量级 Scale	无	额外一个 FP32 Scale
平均位宽（不含张量级）	4 + 8/32 = **4.25 bit**	4 + 8/16 = **4.5 bit**

一个例子说明差别：两组数据 `[0.5,1,2,6]` 与 `[5,10,20,60]` 分别用 Scale 1 和 10 就都能用同一组 E2M1 精确表示。可 Scale 本身也要编码：MXFP4 的 E8M0 只能存 2 的幂，存不了 10；NVFP4 的 E4M3 可以（`10 = 1.25 × 2³`）。NVIDIA 的示例里，同一组数据的 Scale 编码 MSE：E8M0 为 0.72，E4M3 为 0.08。

NVFP4 的反量化是两级相乘：`x̂ = s_tensor · s_block · q`。E4M3 块 Scale 最大 448，覆盖不了整个张量的动态范围，所以先用 FP32 张量级 Scale 拉到合适区间，再由块 Scale 微调；MXFP4 的 E8M0 有 8 位指数、范围够大，不需要第二层，代价是 Scale 只能取 2 的幂。

> 📌 NVFP4 用更小的块、更精细的 Scale 换精度，多付 0.25 bit/元素。NVIDIA 报告 DeepSeek-R1-0528 从 FP8 转到 NVFP4 后关键评测精度下降在 **1% 以内**，显存约为 FP16 的 **1/3.5**、FP8 的 **1/1.8**——厂商在特定模型上的结果，换模型仍要自测。

入口	格式	说明
`--quantization mxfp4`（在线）	权重 E2M1 + 每 1×32 一个 E8M0	激活精度取决于后端，有的仍用 BF16
`--quantization mxfp8`（在线）	权重 FP8 + 每 1×32 一个 E8M0	W8A8 需 SM100+，其他 GPU 回退 W8A16
ModelOpt 检查点（`modelopt_fp4`）	NVFP4 或 W4A16 的 NVFP4	无原生 FP4 的 GPU 回退 Marlin W4A16

在线量化在加载时从 BF16/FP16 直接转换，不需要校准数据；NVFP4 通常先用 Model Optimizer 离线导出。线性层 Kernel 可用 `--linear-backend` 固定（`cutlass`、`flashinfer_cutlass`、`flashinfer_trtllm`、`marlin`），它取代了旧的 `VLLM_NVFP4_GEMM_BACKEND`。

> ⚠️ 在线 FP8 的方案名是 `fp8_per_tensor` / `fp8_per_channel` / `fp8_per_block`。**对未量化检查点写 `--quantization fp8` 会直接报错**，与旧版本行为不同。

4.5.5 硬件支持怎么变成收益

「这张卡支持 FP4」与「这个模型用 FP4 跑得更快」之间隔着几步。要拿到**计算**收益，至少同时满足三条：

1. **Tensor Core 有对应指令**：FP8 从 Ada/Hopper 开始；NVFP4/MXFP4 的原生矩阵乘要 Blackwell 第五代 Tensor Core。
2. **框架选中了对应 Kernel**：没有原生 Kernel 时回退到 Marlin 这类 Weight-only 路径，日志有警告。
3. **激活也被量化**：只有权重低比特（W4A16/W8A16）时乘法仍在 16-bit，**省带宽不省算力**。

三条都满足也只是**线性层**变快。长上下文下 Attention 与 KV 读取可能占大头，要靠 4.4 与 2.6 解决。

部署情况	实际执行	主要收益
FP8 模型，Ada/Hopper/Blackwell	FP8 W8A8 GEMM	显存减半，计算吞吐提升
FP8 模型，Ampere	FP8 Marlin，W8A16	显存减半，Decode 读权重更快

部署情况	实际执行	主要收益
NVFP4 模型，Blackwell 原生	FP4 GEMM	显存最小，计算吞吐提升
NVFP4 模型，无原生 FP4	Marlin W4A16	只剩显存与带宽收益

> 💡 确认实际路径看启动日志的 `Selected <kernel> for <module>` 或回退警告，**比看模型名字可靠**。

4.6 量化选型与 vLLM 实战

4.6.1 从瓶颈出发

原文的看病类比：先找最难受的症状（最紧的约束），再排除身体条件不允许的疗法（硬件/后端不支持），最后用检查结果确认疗效。

首要约束	优先候选	下一步确认
精度退化容忍度低	以 FP16/BF16 为基线，先试 8-bit（FP8 或 INT8 W8A8）	目标任务质量；8-bit 也可能不达标
权重装不下、小 Batch 读权重慢	GPTQ/AWQ INT4（W4A16）	Group Size、实际 Kernel、敏感层
长上下文或并发被 KV 限制	FP8 KV Cache	Scale 是否校准、Attention 后端、长文本质量
GEMM 计算占比高	硬件支持的 W8A8；Blackwell 评 NVFP4	实际算子路径与端到端收益

几条分支可组合：权重装不下且长上下文缓存也不够时，先验 INT4 权重再叠 FP8 KV，**每加一项单独评测一次**。硬件会先筛掉一部分：FP8 要 CC 8.9+（Ampere 只能 W8A16）；INT8 W8A8 不支持 Blackwell；NVFP4 原生要 Blackwell。

方法、格式、Kernel 是四件不同的事：

名称	属于哪一层	加载时看什么
GPTQ、AWQ	量化算法及常见检查点格式	位宽、Group Size、零点、权重排列
compressed-tensors	可承载多种方案的格式与配置	实际可能是 INT8/INT4/FP8
FP8、NVFP4、MXFP4	数值格式及缩放方案	检查点配置、硬件、后端
Marlin、Machete、CUTLASS	执行 Kernel	是否匹配权重布局与设备

```
vllm serve Qwen/Qwen2.5-7B-Instruct-AWQ --dtype half          # 已有 AWQ 检查点
vllm serve /path/to/compressed-tensors-model                    # LLM Compressor 导出
vllm serve /path/to/nvfp4-model --quantization modelopt_fp4    # ModelOpt NVFP4
vllm serve Qwen/Qwen2.5-7B-Instruct --quantization fp8_per_tensor  # 在线 FP8
```

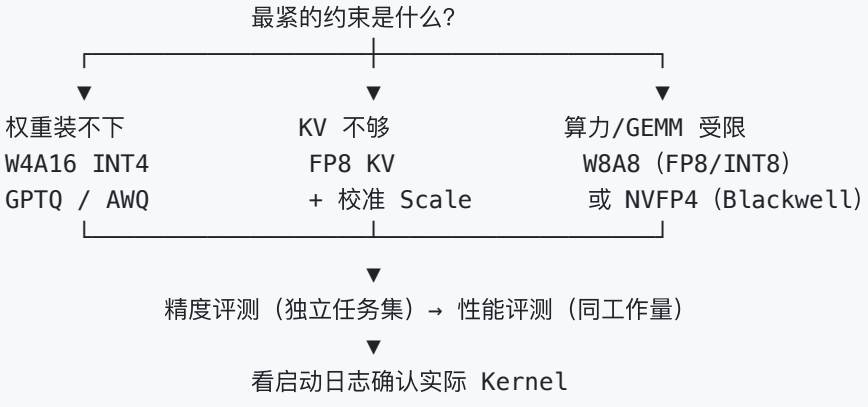
> 📌 在线转换发生在加载阶段：先把原精度权重建在 meta 设备上（只有形状、不占显存），每个线性层权重载入后立即转 FP8，不会把整份 16-bit 权重同时放进显存。但它读的仍是原精度检查点，**拿不到离线校准的好处**。正式部署优先用离线量化检查点。

4.6.2 对照实验的最小骨架

```
python -c "import vllm, torch; print(vllm.__version__, torch.__version__, torch.version.cud
vllm serve Qwen/Qwen2.5-7B-Instruct-AWQ --dtype half --max-model-len 4096 \
  --max-num-seqs 32 --gpu-memory-utilization 0.9 --no-enable-prefix-caching
```

基线组换成未量化模型、其余参数完全一致，**两组必须串行跑**（避免抢显存）。每次启动记三样：权重占用、可用 KV token 容量、实际量化 Kernel。AWQ 省下的权重空间会被分给 KV 池，所以 `nvidia-smi` 总占用**不会缩到 1/4**，变的是 KV 容量。吞吐对比要断言实际输出 token 数等于预期（防 `ignore_eos` 未生效导致工作量不同）；质量用三条冒烟题（JSON、算术、回文函数）只发现明显退化，**输出逐字不同不等于质量退化**。

4.7 本章小结



- 误差只有舍入与截断两类；Scale 由量程决定，粒度决定多少数据共用一把尺子，**切分方向必须与离群值方向一致**。
- 位宽每少 1 bit 格距翻倍，4-bit 是 8-bit 的约 17 倍，所以低比特必须配合更细粒度或误差补偿。
- W8A8 的难点是激活的通道离群值；SmoothQuant 用 $XW = (XS^{-1})(SW)$ 把难度迁给权重，α 控制强度。

- W4A16 只省带宽不省算力；GPTQ 用 $H = 2X^TX$ 做二阶补偿，AWQ 用激活感知缩放保护 1% 显著通道，Marlin 把解包融进 GEMM。
- KV 量化必须便宜且能流式追加；KIVI 对 K 按通道、V 按 token 并配高精度残留区；FP8 KV 是工程主流，但 Scale 要校准。
- **「硬件支持某格式」≠「模型用它能变快」**，要同时满足指令、Kernel、激活也量化三个条件。

第 5 章 Speculative Decoding：先猜一段，再让大模型验收

> 对应 AIInfraGuide 模块四 第 5 章（5.1～5.5，五篇正文）。第 2 章用批处理让一次权重搬运服务更多请求，第 4 章用低比特让每次搬得更少。这一章走第三条路：**同一个请求，能不能一次大模型前向就确认好几个 token?**

5.1 核心原理：Draft + Verify

5.1.1 串行瓶颈与两套分布

自回归要求 $x_{t+1} \sim p(\cdot|x_{\le t})$，$x_{t+1}$ 没定下来，x_{t+2} 就无从谈起——所以 Target 不能凭空并行预测未来十个位置。但小 Batch 的 Decode 基本是访存受限：权重读一遍只为算一个位置。**如果这一遍读取能顺便算好几个位置，单位 token 成本就降下来了。**

投机解码正是利用这份空闲：Draft 先给几个候选，Target 把它们**当作已知输入**，一次前向算出每个位置的条件分布，再逐个判断收不收。原文的主编类比：请助手先写一段，主编一口气审完，最终仍由主编拍板。

设已提交历史 h、草稿长度 γ：Draft 按 $q_i(\cdot) = q(\cdot|h, y_{<i})$ 提议 $y_1 \dots y_\gamma$；Target 在同样历史上算 $p_i(\cdot) = p(\cdot|h, y_{<i})$ 核验第 i 个。p、q 都是**实际用于采样的分布**（已含温度、Top-k/Top-p）；后面的证明对任意归一化分布成立，但**验收时必须用采样时的那一对**。

候选串一旦写好就是已知输入，Target 用 Causal Attention 同时算出所有位置的隐藏状态，第 i 个位置只看得到 $h, y_{<i}$，**不会偷看后面的候选**——这与训练时的 Teacher Forcing 是一回事。一轮会算出 $p_1 \dots p_{\gamma+1}$ 共 $\gamma+1$ 个分布，最后一个留给「全部接受」时再多生成一个 token。

```
一轮示例（γ = 4）：
候选：   y1   y2   y3   y4
验收：   ✅    ✅    ❌    –          → 提交 y1 y2 z3（修正）：前进 3 个，y4 作废
全部通过：✅ ✅ ✅ ✅ → 提交 y1..y4 + z5（额外）：前进 5 个
```

> 📌 减少的是 **Target 的串行调用轮数**，自回归依赖本身没消失——独立 Draft 自己通常还要逐 token 生成。

5.1.2 接受规则

贪心（温度 0）：第 i 个候选等于 Target 在该位置的 argmax 就接受，否则换成 Target 的 argmax 并结束本轮。结果与 Target 自己贪心解码完全一致（前提是数值计算与并列最大值处理一致）。

随机采样：对从 `q_i` 抽出的 `y_i`，接受概率

```
a_i(y_i) = min( 1, p_i(y_i) / q_i(y_i) )
```

即 Target 更看好（`p ≥ q`）就一定收；没那么看好就按 `p/q` 打折收。实现上抽 `u ~ U(0,1)`，`u < a_i` 接受。被拒绝后**不能直接从 `p_i` 重抽**，要从**残差分布**采样：

```
r_i(x) = [p_i(x) − q_i(x)]_+ / Σ_z [p_i(z) − q_i(z)]_+,      [v]_+ = max(v, 0)
```

残差只保留 Target 比 Draft「更想要」的那部分概率。

5.1.3 正确性证明

固定一个位置，某个 token `x` 被输出只有两条路：

```
接受路径：q(x)·a(x) = q(x)·min(1, p/q) = min(p(x), q(x))
记 A = Σ_x min(p(x),q(x))，β = 1 − A；由 Σp = Σmin(p,q) + Σ[p−q]_+ 得  β = Σ_x [p(x)−q(x)]_+
拒绝路径：β·r(x) = [p(x) − q(x)]_+
两路相加：Pr(输出=x) = min(p,q) + [p−q]_+ = p(x)        ∎
```

`p = q` 时 `β = 0`，永远不会走到拒绝路径，残差分母为零的情况也不会出现。

> ⚠️ 只看「被接受的 token」，它们服从 `min(p,q)/A`，**一般不等于 p**。严格服从 Target 的是接受与修正合在一起的**最终输出**。

用全变差距离：`TV(p,q) = ½Σ|p−q|`，则 `A = 1 − TV(p,q)`。Draft 与 Target 重合得越多，需要修正越少——但这只是单个位置的接受概率，**离「整个请求快多少」还隔着成本账**（5.4）。

5.1.4 只给 token 不给概率

N-gram 只产出 token；vLLM 对模型类 Draft 的默认 `draft_sample_method="greedy"` 也只取 argmax、不保存整份概率。此时把 `q` 看成**落在候选上的 one-hot**：`q(y)=1`。代入后接受概率变成 `p(y)`，拒绝后的残差就是「去掉 y、其余按 p 重新归一化」，两路合起来仍恰好是 `p`——**依然无损**。代价是 Draft 的概率信息没被利用；改用 `"probabilistic"` 会保存完整概率参与验收，代价更多显存。

> 🔑 残差是对接受路径「多给或少给」的概率质量做补偿。**接受规则与修正采样必须成对使用**，只改一个就会改变最终分布。

5.1.5 推广到整段序列

每个被保留的 token 都在真实已提交前缀上按规则产生，逐位置归纳得 `Pr(x_1:n|h) = ∏ p(x_i|h,x_<i)`。

为什么首个拒绝后必须停：`y_3` 被拒、换成 `z_3` 后，已算好的 `p_4` 是以 `h,y_1,y_2,y_3` 为条件的，而真实历史已是 `h,y_1,y_2,z_3` —— `y_4` 的验收依据失效。所以每轮**只能提交连续接受的前缀**。

为什么能多一个 token：全部通过时 `p_{γ+1}` 已在正确历史上算好，直接采样即可。两种情况保证每轮**至少前进 1、至多 γ+1** 个 token；遇 EOS/Stop/长度上限时实际提交更短。

5.1.6 KV Cache：草稿不是已提交的历史

状态	处理	常见错误
已提交历史	保留对应 KV	回退时误删 Prompt 的 KV
连续接受的候选	直接复用验证时算出的 KV	再跑一遍前向，浪费收益
被拒位置及其后缀	标记失效，回退缓存指针	下一轮还读到错误候选的 KV
修正/额外 token	提交，但 KV 留到下一轮作输入时再算	把「已采样」误当「已有 KV」

> 📌 最后一行最容易 off-by-one：修正 token 已对外输出，但**它自己的 KV 要等它作为下一次前向输入时才生成**。引擎内部要分别记录「已提交 token 数」与「已计算 KV 的 token 数」。Draft 若有自己的 KV，也要回退到最终提交的历史上。
>
> 🔑 **概率证明管「每个 token 怎么采」，缓存与调度管「下一轮看到的历史对不对」。任何一处错位，理论上的无损都落不了地。**

5.1.7 「无损」保证什么

保证	不保证
正确实现时输出分布 = 被验证 Target 的分布	文本逐字一致（随机采样消耗随机数路径不同）
贪心时与 Target 的 argmax 路径一致	修复 Target 本身（Target 若量化，保证针对量化后模型）
标准拒绝采样规则的分布正确性	覆盖非标准规则（Typical Acceptance、synthetic 压测）

vLLM 文档把无损拆成三层：**理论上无损**（受限于硬件数值精度）、**算法层面**由 Rejection Sampler 的收敛测试与贪心一致性测试保证、**不承诺 logprobs 完全稳定**。Batch 形状与 Kernel 不同可能让两个几乎并列的 logits 交换大小，上线前仍需端到端比对。

5.2 Draft 与无模型方案

5.2.1 好 Draft 的标准

要求	含义	反例
便宜	生成候选的耗时、显存、通信开销小	层数很深的 0.5B 可能比层数浅的 1B 慢
准	在真实业务上与 Target 一致	同参数量但训练数据/对齐方式不同的模型
接得上	token ID、上下文、缓存、采样规则可对接	词表不同、上下文上限不同

原文的招聘类比：工资低只是一方面，还要看稿子改起来费不费劲、文件能不能打开。**参数量只是粗糙指标**，实际延迟还取决于层数、Attention 结构、KV 长度、Kernel 与 Batch。

路线	候选来源	额外权重	擅长
独立 Draft	小模型的条件预测	有	提议历史中没出现过的新内容
N-gram / Prompt Lookup	历史中相同的 token 片段	无	复制、抽取、改写
Suffix Decoding	后缀树匹配 + 续写频次	无	重复轨迹、代码编辑
Self-Draft（Medusa/EAGLE/MTP）	Target 的隐藏状态/特征	有（且与 Target 版本绑定）	通用，接受率更高

> 📌 「无模型」只是没有额外网络权重。查找、索引维护、CPU↔GPU 传递、Target 验证都有成本。独立 Draft 除了权重还有自己的 KV、激活与工作区，**单请求变快与最大并发变小可以同时发生**。

并行 Draft：PARD 让 Draft 一次前向预测多个位置，耗时基本不随 `γ` 线性增长；vLLM 通过 `"parallel_drafting": true` 启用，但**需要专门训练的 Draft 权重**，普通小模型打开这个开关不会变成并行 Draft。

5.2.2 Tokenizer 与上下文

`p`、`q` 必须定义在**同一个 token 空间**。两个 Tokenizer 都能编码 "hello"，不代表切分与编号一致。要核对：token → ID 映射、特殊 token、Target 实际用的 Chat Template；`vocab_size` 相同只说明数量一样。

```
t_vocab, d_vocab = target.get_vocab(), draft.get_vocab()
diff = {tok for tok in t_vocab.keys() | d_vocab.keys()
        if t_vocab.get(tok) != d_vocab.get(tok)}
print("映射不一致的 Token 数:", len(diff))
same_template = target.apply_chat_template(m, tokenize=True) == \
                draft.apply_chat_template(m, tokenize=True)
```

Draft 读的是 Target 已编码的历史，不会自己套模板——**模板差异影响的是 Draft 有没有「见过」这种格式**，进而影响接受率。词表不同时 vLLM 提供 `use_heterogeneous_vocab: true`（TLI 算法），但只支持 `draft_model` 且只支持 `draft_sample_method="greedy"`。

> ⚠️ **Draft 的上下文不能悄悄变短**：最大上下文、RoPE、长文能力不一致时，长请求可能失败，或接受率在长文本上明显下滑。部署前把 `max_model_len` 对齐。

5.2.3 N-gram

假设历史末尾是 `[A,B,C]`，更早位置出现过 `[A,B,C,D,E,F]`，就把 `[D,E,F]` 当候选。**匹配发生在 token ID 序列上，与中文分词或字符数无关。**

参数	含义	vLLM 默认
查找窗口 `n`	`prompt_lookup_min` / `prompt_lookup_max`	都不设时为 5；只设一个时另一个取同值
提议长度	`num_speculative_tokens`	由投机配置给出

v0.32.0 的 CPU 实现会在整个请求历史（Prompt + 已生成）中查找，**优先取 `[min,max]` 内最长的匹配，同长度取最早出现的位置**。另有 `ngram_gpu` 把查找放到 GPU。

参数方向	效果
窗口长	约束强、候选更相关，但**命中更少**
窗口短	容易命中，却可能只对上了一个逗号加「的」
提议长	可能省更多 Target 调用，也可能前几个就被拒、后面全白算

适合「输出大量复用输入」：摘要引用原文、代码只改几行、按模板抽取字段。**没有概率时按 one-hot 处理，依然无损**。

5.2.4 Suffix Decoding

N-gram 每次只看一处匹配。Suffix 用**后缀树**记录每种续写出现过多少次，挑最有把握的路线：假设某后缀之后 `return` 出现 20 次、`raise` 2 次，先提议 `return` 更划算；续写越分散越不值得往深处猜。vLLM 用 `suffix_decoding_min_token_prob`（默认 0.1）过滤低概率 token。

- **Prompt 树**：针对当前请求的 Prompt 与已生成内容。
- **全局树**：跨请求缓存历史输出，`suffix_decoding_max_cached_requests` 默认 10000，设 0 关闭。

每步提议多少 token 是**动态**的：匹配越长越有把握就猜得越多，所以 `num_speculative_tokens` 是上限（文档建议 16 或 32）；不设时取 `suffix_decoding_max_tree_depth`（默认 24）。它在代码编辑、Agent 自我反思、RL Rollout 这类高重复任务上效果明显。

成本项	要核对什么
搜索与更新	建树、更新频次是否拖慢每一步
历史存储	全局树缓存多少请求、占多少内存
统计作用域	跨请求共享是否符合数据隔离要求
依赖	需要 `pip install arctic-inference`

> ⚠️ 全局树会让一个用户的输出影响另一个用户请求的候选。候选仍要过 Target 验证（**分布不受影响**），但多租户是否允许这种共享由业务决定。
>
> 📌 **Prefix Cache 复用已算过的 KV，减少重复 Prefill；N-gram/Suffix 提议可能要生成的 token，仍需验证。** 两者都叫「复用历史」，指标不能混。

5.2.5 接受率的口径

```
Draft Acceptance Rate  = C / D            C 接受数、D 候选总数
Mean Acceptance Length = 1 + C / R        R 验收轮数；加 1 是每轮的修正/额外 token
```

第三个量是**候选覆盖**：有多少步真正提出了候选。N-gram 查不到时这一步就不投机——**若只统计有候选的轮次，90% 的接受率背后可能只有 10% 的步骤在投机。**

> ⚠️ **固定长度与动态长度不能互换**：只有每轮都提 γ 个候选时才有 $D = \gamma R$。Suffix、动态 SD 必须直接用实际 D、R 计算。
>
> 另一个混淆是**位置接受率**：「第 i 位在前面都通过时被接受」与「从第 1 位一路通过到第 i 位」是两回事，后者衰减快得多。`vllm:spec_decode_num_accepted_tokens_per_pos_total` 统计的是后者。

5.3 Medusa、EAGLE 与 MTP

独立 Draft 像另请助手，得把全文重读一遍。另一种思路是**用上主编已经形成的理解**——Target 的隐藏状态里装着它对上文的理解。

路线	复用 Target 的什么	新增部分	怎么得到
Medusa	最后一层隐藏状态	多个预测头	在 Target 上训练这些头
EAGLE / EAGLE-2	高层特征 + token	一层左右的草稿网络	针对 Target 训练

路线	复用 Target 的什么	新增部分	怎么得到
EAGLE-3	低/中/高多层特征 + token	草稿网络	改进的训练流程
MTP	共享 Embedding 与输出头	预训练时一起训练的 MTP 模块	随模型权重发布

> 🔑 **复用 Target 的表示，省掉重复理解；保留 Target 的验收，最终输出仍由它决定。**

Medusa：在同一个 `h_t` 上挂 `K` 个小头，第 k 个头直接预测第 `t+k+1` 个位置：`q^(k) = softmax(g_k(h_t))`。所有头读同一个 `h_t`，可一次算完；但第 3 个头不知道第 1、2 个位置最后选了什么，**越远越像闭眼猜**。为免「前面错一个整条链就废」，每个头保留几个候选组成一棵**稀疏树**（不能展开 `b^d` 全部组合，只保留校准集上最有希望的节点）。Medusa-1 冻结骨干只训头（配合精确验收可无损）；Medusa-2 联合训练，**已得到一个新 Target**，质量对照要拿训练前的原模型。

Tree Attention：同一个前缀可能接 `return` 也可能接 `raise`，两者后续属于互斥历史。规定**每个节点只能看到共同前缀与自己的祖先**，不能用普通下三角 Mask，否则后分支会看到前分支。**Position ID 按树深度编号**，不是按展平下标。Mask 决定「能看到谁」，Position ID 决定「自己在第几个位置」，两者都要对；验证后只提交一条通过路径，其他分支的临时 KV 全部作废。

```
parents = [-1, 0, 0, 1, 2]      节点 0 根，1/2 兄弟，3 挂 1，4 挂 2
正确（树形）：                  错误（下三角）：
   0  1  2  3  4                    0  1  2  3  4
0  ■                            0  ■
1  ■  ■                         1  ■  ■
2  ■     ■                      2  ■  ■  ■     ← 2 看到了 1（跨分支泄漏）
3  ■  ■     ■                   3  ■  ■  ■  ■
4  ■        ■  ■                4  ■  ■  ■  ■  ■
```

EAGLE：Target 进入 LM Head 之前的顶层特征比 token 更有规律，用「一个全连接层 + 一层 Decoder」的小网络据当前特征自回归预测下一个特征，再经 Target 的 LM Head 转成候选。但只看特征不够——采样选 `return` 还是 `raise` 会让下一步特征走向完全不同，所以草稿网络同时输入**当前特征 + 已采样出的下一个 token**。

> ⚠️ EAGLE 权重与**具体 Target 版本**绑定。参数规模相同、名字相近的两个 8B 模型不能共用同一份 EAGLE 权重。

EAGLE-2：EAGLE/Medusa 用静态树，EAGLE-2 用置信度动态扩展与剪枝，且**看整条路径而非单节点**：`v(u) ≈ ∏_{j∈path(u)} c_j`。扩展时选 `v` 最高的节点继续长，重排时挑 `v` 最高的一批送验证。**置信度只用来分配预算，不参与接受判定**。

EAGLE-3 三处改动：① 去掉特征预测损失，只用 token 预测作训练目标；② 融合低/中/高多层特征；③ **Training-time Test（TTT）**——推理时草稿网络从第二步起读自己上一步的输出，训练时就模拟这个过程。

MTP：把顺序倒过来——预训练时就多预测几个未来 token，推理时这个模块正好当草稿。DeepSeek-V3 的 MTP 与 Medusa 的「并列小头」关键区别是**按顺序预测、保留因果链**：

```
h'^k_i = M_k [ RMSNorm(h^{k-1}_i) ; RMSNorm(Emb(t_{i+k})) ]
```

读上一深度表示与第 `i+k` 个 token 的 Embedding，各自 RMSNorm 后拼接、经 `M_k` 压回原维度，过一个 Transformer Block，再用与主模型共享的输出头预测。Embedding 与输出头共享，所以模块很轻。

> 📌 DeepSeek-V3 训练时只预测一个额外 token（`D=1`）。论文报告：推理时用 MTP 投机，**第二个 token 接受率 85%–90%，TPS 达原来的 1.8 倍**——该模型、该部署条件下的数字，换模型要重测。**若自行微调或量化了主模型，MTP 不会自动适配**，接受率需重测。

验收与部署的一处关键区分：Medusa 的 **Typical Acceptance**（候选概率超过与熵相关的阈值就接受）能接受更长，但**不满足保分布证明，属近似方案**。另外三件事别混：**并行前向**是计算优化；**Token 级验收**是从左到右逐个核验、遇首个拒绝即停；**Block Verification** 是把整块候选一起决定接受多少，它保分布且预期提交长度不低于逐 token 验收（vLLM 的 `"rejection_sample_method": "block"`，目前只在 V2 Model Runner 实现）。

框架与论文结构要分开核对：vLLM v0.32.0 的 `eagle` / `eagle3` / `mtp` **都按线性链出草稿、没有树形验证**；SGLang 的 EAGLE 在 `--speculative-eagle-topk > 1` 时构建草稿树，配合 `--speculative-num-steps` 与 `--speculative-num-draft-tokens` 控制预算（其默认 V2 Overlap Scheduler 只支持 `topk=1`）。

5.4 收益边界：接受率高，为什么仍可能变慢

「接受率 80% 是不是一定变快？」接受率只说猜得准不准，**没说明猜一次花多久、验证要多少算力、多出来的缓存挤掉了多少请求**。

5.4.1 性能账本

记 `K ∈ {0,…,γ}` 为连续被接受的前缀长度，则一轮提交 `L = K + 1`（多出的 1 是修正或额外 token）。**哪怕一个都没通过也能前进一步**——但这是**进度保证，不是性能保证**：全军覆没的一轮花的是 Draft 加验证的时间，只换来普通 Decode 一步的产出。

```
E[L] = 1 + Σ_{i=1..γ} Pr(K ≥ i)            (尾概率恒等式，不要求位置独立)
Pr(K ≥ i) = ∏_{j≤i} a_j                     (a_j 为条件接受概率，链式法则)
固定 a: E[L] = (1 − a^(γ+1)) / (1 − a)      (a = 1 时取极限 γ+1)
T_round = T_draft + T_verify + T_other
S ≈ T_0 · E[L] / E[T_round]
```

要变快必须 `E[T_round] < T_0 · E[L]`：一轮耗时要少于普通 Decode 生成同样多 token 的时间。实测用「总提交 token / 总耗时」，**不要对每轮速度比取算术平均**。

验证不是免费的：一次验 `γ+1` 个位置能复用同一次权重读取、提高算术强度，但计算量、Attention、候选 KV 写入与 Logits 处理都随位置数增长。**小 Batch 时 `T_verify` 接近 `T_0`；候选长、Batch 大或上下文长时，它会明显超过 `T_0`**。

5.4.2 投机长度的收益递减

```
ΔE[L] = a^(γ+1)
```

`a = 0.8` 时第 1 个候选带来 0.8 个预期 token，第 8 个只能带来约 **0.168** 个；而每多一个位置的成本并不会下降，所以到某个长度之后多猜就是净亏损。

深度与宽度是两种预算：链式只调深度；树形还要调每层分支，树越宽找到可接受路径的机会越大，但没被选中的节点也全要算——**只报告「深度 4」不够，还要报告每轮实际验证了多少节点**。

动态长度：最优长度随负载变化。vLLM 用 `num_speculative_tokens_per_batch_size` 按并发分段：

```
[[1, 64, 3], [65, 128, 1], [129, 512, 0]]
```

即并发 1–64 猜 3 个、65–128 猜 1 个、129 以上不投机。理由是 Batch 为 `B`、每请求猜 `γ` 时，验证的有效 Batch 约为 `B×γ`，超过临界值后反而拖慢 TPOT。该功能在 EAGLE、EAGLE-3、DFlash 上测过，**开启 Data Parallel 会自动退回固定长度**。另有 Adaptive Verification 为每个请求决定验证长度，要求完整 CUDA Graph，不支持 LoRA 与 PP。

> 📌 好的策略优化的是「**单位时间推进多少合格输出**」，而不是单独把接受率推到最高。

5.4.3 任务标签不是性能结论

场景	可能有利	仍会失败
代码编辑	大段原文保留，标识符反复出现	改动集中，续写方向变化大
结构化抽取	输出复制输入，格式固定	Prompt 里没有答案，或格式约束与候选冲突
Agent 轨迹	重复步骤、相似续写多	工具返回结果改变后续走向
开放对话	强 Draft 仍可能与 Target 合拍	表达多样，便宜草稿频繁被否

温度、Top-p、对齐方式与 Prompt 分布都会影响接受。**随机 token 的 benchmark 适合控制性能变量，但没有真实业务的重复结构**——要评 N-gram/Suffix 的实际收益必须用真实 Prompt。

5.4.4 高并发、长上下文与抢占

Continuous Batching 已经吃掉了一部分空闲：低并发时每步只处理少数请求，算力大量空闲，投机正好利用；并发上来后一次权重读取服务更多请求，普通 Decode 自己就把算力用起来——**这时再加候选，就从「用空闲算力」变成「和其他请求抢算力」**。vLLM 文档把投机定位为**降低中低 QPS、访存受限场景下的 token 间延迟**；其方法选择表的定性判断是 EAGLE/MTP 低 QPS 收益高、高 QPS 中到高，N-gram/Suffix 低 QPS 低到中、高 QPS 中等。

长上下文增加读写与临时状态：候选要占额外 KV，独立 Draft 的缓存也随上下文增长。

```
M_candidate ≈ B · γ · m_KV/token
```

（忽略分页取整、lookahead 槽位、共享前缀与树布局，只说明多出了哪一项，**实际容量以引擎日志为准**。）

抢占会吃掉纸面收益：Draft 权重与临时缓存减少可用 KV，可能触发更多抢占与重算，或让引擎接纳更少请求。**单轮验证更快、整个请求完成得更慢，这两件事并不矛盾**——记录接受长度时要同时记录 KV 容量、抢占次数、成功请求数与实际吞吐。

5.4.5 与量化叠加

组合	改变什么	输出基准
只量化 Draft	提议分布 `q`	仍是未量化 Target 的 `p`，只是速度与接受率变了
量化 Target	验证基准变成 `p_quant`	保证与量化后模型一致，不恢复原精度
近似验收 + 量化	两种变化叠加	需分别评估质量

EAGLE/MTP 读 Target 特征，训练时见的是原精度特征；Target 换量化后特征分布变化，接受率可能下降，需重测。

> 📌 **加速比不能直接相乘**。量化和投机缓解的都是 Decode 访存瓶颈；量化减少权重读取量后，本来被掩盖的验证计算可能变显眼，同时释放显存腾出候选空间。应分别测原始基线、仅量化、仅投机、组合，并写明质量基准。

5.4.6 调度与流式

开启投机后，每步要为候选预留位置、分配 Token 预算，再按接受结果更新真实进度；同批不同请求的候选数、接受长度、剩余额度都可能不同。这条链路会和 Continuous Batching、Chunked Prefill、CUDA Graph 的固定形状、异步调度相互影响。

指标	定义	投机后
TPOT	首 token 之后平均每 token 时间	通常下降
ITL / 流式间隔	相邻两次返回的停顿	可能变差：一个 chunk 里可能有多个 token
TTFT	含排队与 Prefill	基本不受益

投机一次验证可能返回好几个 token、下一轮又要等一阵，流式输出更像「一段一段冒出来」。实时交互场景还要看 P95/P99 与业务能接受的最长停顿。**停止条件与结构化输出要及时生效**：接受前缀里出现 EOS/Stop/长度上限时只提交结束前部分；带状态的约束要随每个候选前缀更新（`num_draft_tokens` 已扣除被约束作废的候选）。

5.5 vLLM 实战

```
python -m pip install 'vllm[bench]==0.32.0'      # bench 子命令需要额外依赖
TARGET_MODEL=Qwen/Qwen3-8B
COMMON=(--served-model-name spec-demo --host 127.0.0.1 --port 8000
        --dtype bfloat16 --tensor-parallel-size 1 --kv-cache-dtype auto
        --max-model-len 4096 --max-num-seqs 16 --max-num-batched-tokens 4096
        --gpu-memory-utilization 0.85 --no-enable-prefix-caching
        --generation-config vllm --seed 42 --per-request-spec-decode-metrics summary)

vllm serve "$TARGET_MODEL" "${COMMON[@]}"                       # 1) 基线
vllm serve "$TARGET_MODEL" "${COMMON[@]}" \
  --speculative-config '{"method":"ngram","num_speculative_tokens":4,"prompt_lookup_min":2,
vllm serve "$TARGET_MODEL" "${COMMON[@]}" \
  --speculative-config '{"method":"draft_model","model":"Qwen/Qwen3-0.6B","num_speculative_
vllm serve "$TARGET_MODEL" "${COMMON[@]}" \
  --speculative-config '{"method":"eagle3","model":"RedHatAI/Qwen3-8B-speculator.eagle3","n
```

对照变量要固定：Target 与 dtype（不混量化）、GPU/TP（不把卡数变化算成收益）、KV dtype（不叠 FP8 KV）、上下文与 Token 预算、前缀缓存（关闭，避免重复请求命中 KV 干扰对照）、采样与模板（显式指定）。Qwen3 默认进 Thinking 模式，本例统一 `enable_thinking=False`——**若业务要开 Thinking，输出长度与接受情况都会变，需重测**。

要点：`--generation-config vllm` 让服务忽略模型自带的默认采样；`--per-request-spec-decode-metrics summary` 让每个响应带上本请求统计；各组用同一服务名，客户端脚本不用改；**各组顺序运行**，启动前 `nvidia-smi` 确认显存已释放；`--speculative-config` 只放投机配置，**`temperature` / `top_p` 属客户端请求**；EAGLE-3 的候选数取自模型卡（超过 3—4 个后增益明显变小），`method="eagle"` 对应第一代权重，**两代产物不能靠改方法名互换**。

> ⚠️ `rejection_sample_method: "synthetic"` 按预设接受率放行候选，用来压测系统上限。**它不保分布，只能用于性能测试，不能出现在线上配置里。**

5.5.1 逐请求接受统计

字段	含义
`num_spec_steps`	验收轮数 `R`
`num_draft_tokens`	候选总数 `D`（已扣除被结构化输出作废的候选）
`num_accepted_draft_tokens`	接受数 `C`（不含额外 token）
`acceptance_histogram`	长度 `num_speculative_tokens + 1`，第 j 项是恰好接受 j 个的轮数
`mean_acceptance_length`	`1 + C/R`

```
for task in ["code", "chat"]:
    st = [r["speculative"] for r in rows if r["task"] == task and r["speculative"]]
    C, D, R = (sum(s["num_accepted_draft_tokens"] for s in st),
               sum(s["num_draft_tokens"] for s in st),
               sum(s["num_spec_steps"] for s in st))
    print(task, C / D if D else None, 1 + C / R if R else None)
```

> 📌 **先加总再算比值**，不要对每个请求的百分比取平均。`D = 0` 时输出 `None`，表示**没有分母**，与「接受率 0%」是两回事；直方图第 0 项（一个都没接受的轮数）往往最能说明 chat 组为什么收益有限。基线服务没有投机，该字段为 `null`。

服务端 `/metrics` 的 `vllm:spec_decode_num_drafts_total`、`vllm:spec_decode_num_draft_tokens_total`、`vllm:spec_decode_num_accepted_tokens_total` 分别对应 `R`、`D`、`C` 的累加；`..._per_pos_total` 带 `position` 标签，第 i 位计数是「至少接受 i+1 个」的轮数。**这些是进程启动以来的累计值，要用实验前后两次快照的差值**；服务端汇总不区分 code/chat，预热与 `n>1` 也会计入。

5.5.2 压测与结果解读

```
LABEL=baseline   # 切换服务后改为 ngram / draft / eagle3
for TASK in code chat; do
  for C in 1 8; do
    vllm bench serve --backend openai --base-url http://127.0.0.1:8000 \
      --endpoint /v1/completions --model spec-demo --tokenizer Qwen/Qwen3-8B \
      --dataset-name custom --dataset-path "data/${TASK}.jsonl" \
      --chat-template-kwargs '{"enable_thinking":false}' \
      --custom-output-len 128 --num-prompts 64 --num-warmups 4 \
      --max-concurrency "$C" --request-rate inf \
      --temperature 0 --top-p 1 --ignore-eos --seed 42 \
      --percentile-metrics ttft,tpot,itl --metric-percentiles 50,95,99 \
      --save-result --result-dir results --result-filename "${LABEL}-${TASK}-c${C}.json"
  done
done
```

> 📌 `--ignore-eos` 让每个请求都生成满 128 个 token，避免各组因自然结束长度不同而工作量不同；**代价是输出可能超长，这批文本不能用来评价质量**。64 个请求只是起点，**P95/P99 在这个样本量上并不稳定**，正式实验要加样本与重复次数。

现象	可能原因	下一步
code 组接受率高但加速不明显	验证长度、Draft 耗时，或基线本身已很快	扫 `γ` 曲线
N-gram 接受率高但很多步没候选	候选覆盖低	看直方图与 `D/R`，一并报告覆盖

现象	可能原因	下一步
并发 1 变快、并发 8 吞吐下降	高负载抢算力，或 KV 容量变小	试 Dynamic SD
平均 TPOT 更好但流式停顿更长	ITL 尾部分布变差	看 ITL 的 P95/P99
服务启动失败	参数是否存在于该版本、方法名是否写对	辅助权重与 Target 是否配套、显存是否够
响应里没有 `speculative_decoding`	是否带 metrics 参数、请求是否 `n=1`	流式请求是否开 `include_usage`
接受率接近 0	词表、模板、Thinking 开关是否与权重训练条件一致	Target 是否量化
各组贪心输出不一致	模板、采样参数、`--generation-config`	Batch 形状与 Kernel 的数值差异

调优顺序：固定并发扫候选数 1、2、3、4…，画 TPOT 与平均接受长度曲线，找收益开始回落处；低/高并发最优不同就用 `num_speculative_tokens_per_batch_size` 分段（高负载段可设 0）；N-gram 窗口要同时看接受率与覆盖率；按业务换方法（复制多用 N-gram/Suffix，新内容多用独立 Draft/EAGLE-3，自带 MTP 优先试 MTP）。

> ✅ 可上线：目标负载下 TPOT 与吞吐都优于基线，质量评测通过，容量损失可接受。❌ 该放弃：只在远低于目标负载的并发下有收益，或接受率高但端到端时间没改善。

5.6 本章小结

- 投机解码让便宜的 Draft 先写候选，再用一次 Target 前向批量验证，**减少的是 Target 的串行轮数**；并行计算的是已知候选的条件分布。
- 贪心比 argmax；随机采样按 `min(1, p/q)` 接受，拒绝后从残差 `[p-q]_+` 归一化采样；接受路径 `min(p,q)` + 拒绝路径 `[p-q]_+` 恰好等于 `p`。
- 只给 token 的 Draft 可视为 one-hot 的 `q`，同样无损，接受率取决于 `p(y)`。
- 每轮只提交连续接受前缀 + 1 个新 token；KV 回退、Draft 状态同步、Logits Processor 状态是「无损落地」的工程部分。
- N-gram/Suffix 无额外权重但依赖历史重复；独立 Draft 能猜新内容但吃显存；Medusa/EAGLE/MTP 复用 Target 表示，都要专门训练并与版本绑定。
- `E[L] = 1 + ΣPr(K≥i)` 不要求独立；加速条件是**一轮耗时低于普通 Decode 生成同样多 token 的时间**；边际收益 `a^(γ+1)`。
- 高并发、长上下文、抢占会吃掉收益；vLLM 把投机定位在中低 QPS 场景。
- 只量化 Draft 不改变输出基准，量化 Target 会；两种加速比不能相乘。

第 6 章 分布式推理：TP、PP、DP、EP 的取舍

> 对应 AIInfraGuide 模块四 第 6 章。原文该章只有扉页，本章按公开论文、vLLM 分布式文档与通用工程实践重建，性能倍数一律标【量级】或【待实测】。

推理并行与训练并行的目标不同：训练关注梯度同步与收敛，推理关注显存装载、TTFT/TPOT 与单位成本吞吐。把训练并行的直觉直接搬到推理上，是最常见的误判来源。

维度	训练并行	推理并行
首要目标	装得下 + 算得快（缩短训练时间）	装得下 + 延迟达标 + 单位成本吞吐
通信模式	每步梯度 AllReduce/AllGather	每层 AllReduce、KV 传输、MoE All-to-All
批大小	全局 batch 大，通信可被计算掩盖	单请求 batch 小，通信**难被掩盖**
典型瓶颈	通信带宽、显存	KV 显存、单步延迟、跨节点带宽

6.1 四种并行

并行	切什么	通信	适合
TP	层内矩阵（Column 切输出、Row 切输入）	每层 2 次 AllReduce（`o_proj`、`down_proj`）	降延迟、装权重；**不跨节点**
PP	层与层	段边界一次	跨节点扩吞吐；**增加单请求延迟**
DP	复制整副本	无（副本独立）	横向扩吞吐；扩容粒度 = 整副本
EP	MoE 的专家	两次 All-to-All	MoE 部署；难点是负载不均

TP 的通信量与激活大小（`batch × seq × hidden`）成正比，与权重无关。Decode 阶段 batch 小、单步计算少，但 AllReduce 的固定延迟不变——**TP 度越高，通信在单步耗时中的占比越大**。这是「TP 不是越多越好」的根本原因。带宽上，NVLink 是数百 GB/s 到 TB/s 级，跨节点 RDMA 是数十到百 GB/s 级且延迟更高【量级】，**差一个数量级以上**，而 TP 每层要做两次 AllReduce、每步都做，所以工程默认「TP 不跨节点」。

TP 按头切分，要求 `num_attention_heads` 与 `num_kv_heads` 都能被 TP 度整除；**GQA 模型的 KV 头数往往就是 TP 度的上限**。vLLM 在 TP 超过 KV 头数时会**复制 KV 头**（多 rank 持有相同头、冗余计算），不影响正确性但浪费显存与带宽。

PP 在推理中的特殊性：训练时一个 global batch 切成多个 micro-batch 把流水线填满；推理时请求连续到达，天然形成微批，但每个请求在各段之间必须顺序流动——同一时刻只有一段在为一个请求干活。

```
PP=4 的单请求：任一时刻只有一段忙
GPU0 [P][D][D][D][D]    空闲 …
GPU1    [P][D][D][D][D] 空闲 …
GPU2       [P][D][D][D][D] 空闲 …
GPU3          [P][D][D][D][D] 空闲 …
```

> 【推断】所以 **PP 的效率高度依赖并发度**：并发足够高时多个请求的微批能把流水线填满；低并发、长 Decode 场景下 bubble 非常显著。这也是「PP 适合跨节点扩吞吐，不适合降单请求延迟」的原因。

DP 的三个关键点：① 副本之间**不共享 KV**，所以前缀缓存与 prefix-aware 路由才有价值；② 扩容粒度是整副本（一个副本 4 卡，扩容就 4 卡起）；③ vLLM 的 `--data-parallel-size`（单实例多 DP rank，共享 API Server）与外部多副本 + 网关是两种做法，前者部署简单、单点故障域大，后者隔离好、可异构、滚动升级容易。

EP 的工程难点：

难点	表现	缓解
负载不均	某些专家被路由的 token 远多于其他	辅助损失、capacity factor、专家复制
All-to-All 延迟	跨节点更明显	限制 EP 组在节点内、用高带宽互联
显存不均	热点专家所在卡压力大	重排专家分布、动态调度
与 TP 组合	两套切分叠加，通信复杂	先 TP 填满节点，再在节点间做 EP

> 【推断】MoE 的「有效参数量」远小于总参数量，但**显存要装下全部专家**。所以部署矛盾往往是「装不下全部专家」而非「算力不够」，EP 首先解决显存问题，其次才是吞吐。

6.2 多节点与 Ray

单机多卡用 `torchrun` 或 vLLM 内置多进程即可；跨节点要解决进程发现、GPU 占位、拓扑感知与故障重启，Ray 把这些抽象成 actor 与 placement group。

```
# 单实例多机（TP×PP 超过单节点 GPU 数时需要 Ray）
vllm serve meta-llama/Llama-3.1-70B-Instruct \
  --tensor-parallel-size 8 --pipeline-parallel-size 2 \
  --distributed-executor-backend ray
```

> 【补充】`--distributed-executor-backend` 取 `mp`（默认）或 `ray`。Ray 按 `CUDA_VISIBLE_DEVICES` 与节点资源做 bundle 分配，**节点标签与资源声明必须准确**，否则会出现「看起来分配了 8 卡，实际两张在别的节点」。Ray Serve 还把「多副本 + 路由」做成框架能力（`num_replicas` + `ray_actor_options={"num_gpus": 4}`），把副本数与路由策略变成声明式配置。

拓扑错配	后果	检查
TP 跨节点	AllReduce 走 RDMA，延迟暴涨	启动日志的 rank→节点映射
卡不在同一 NVLink 域	带宽下降	`nvidia-smi topo -m`
placement group bundle 跨节点	资源碎片、启动失败	Ray 调度日志
未限制 `max-model-len`	每卡 KV 需求超预期	启动日志的 KV 容量

6.3 选择顺序与验证

判断顺序：装不下 → TP（节点内）；延迟不够 → 先查瓶颈（是排队还是计算）再决定 TP；吞吐不够 → DP；MoE → TP + EP。

验证清单：每卡权重分片应约为总量 1/TP（否则说明某些层被复制）；每卡 KV 容量应大致均衡（否则说明复制了 KV 头）；TTFT 随 TP 应「先降后平、再升」——小 TP 时降延迟明显，到某点后 AllReduce 主导，延迟不再下降甚至上升；吞吐随 TP 有提升但非线性，进入通信瓶颈后应转 DP。【量级】拐点取决于模型大小、互联带宽与 batch，必须实测。

第 7 章 PD 解耦：把读题和答题分到两拨人

> 对应 AIInfraGuide 模块四 第 7 章。原文只有扉页；机制部分按 DistServe（OSDI'24）、Splitwise（ISCA'24）与 vLLM 解耦预填充文档重建，数字标注来源。

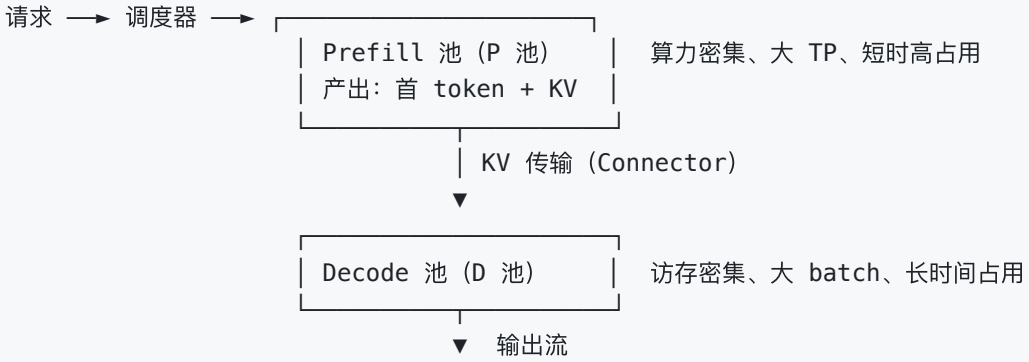
7.1 混合 Batching 的三层干扰

第 2 章讲过 Chunked Prefill 如何缓解 Prefill 对 Decode 的干扰，但「缓解」不等于「消除」：只要共享同一批 GPU，就存在竞争。原文扉页给出的定性判断是：**Decode 的 P95 TPOT 可被 Prefill 拖慢 3–5 倍**。

```
① 计算时间干扰：一步塞进 4000 token 的 Prefill → 该步整体变长 → 同批 Decode 全部等待
② 资源竞争：      Prefill 占满 SM / 显存带宽 → Decode 的权重搬运被推迟
③ 并行策略耦合：  Prefill 想要大 TP（算力密集），Decode 想要小 TP + 大 batch（访存密集）
                  混在一起只能取折中，两边都不最优
```

第 ③ 点最容易被忽略，却是 DistServe 的核心论点之一：**混合部署不仅带来干扰，还强制两个阶段共享同一套资源分配与并行策略**。

7.2 解耦架构



收益	代价
两阶段可独立选并行策略与硬件	KV 传输的带宽与延迟开销
Prefill 不再拖慢 Decode 的 TPOT	调度器复杂度上升（两池排队、配比）
可分别扩缩容（P 按 TTFT、D 按 TPOT）	P/D 配比难调，负载波动易失衡
故障隔离	端到端多一跳，TTFT 可能略增

7.3 DistServe 与 Splitwise

DistServe（OSDI 2024，北大 + StepFun + UCSD）三条主张：解耦消除 P/D 干扰，让每个阶段可独立优化；按应用的 TTFT/TPOT 要求联合优化每阶段的资源分配与并行策略；按集群带宽放置两阶段以最小化通信开销。论文摘要的结果：在多种模型、应用与延迟要求下，相比当时 SOTA **可多服务 7.4 倍请求，或在同等条件下把 SLO 收紧 12.6 倍，同时让 >90% 的请求满足延迟约束**。

> 📌 注意这三个限定词是**绑在一起的一组结论**：7.4× / 12.6× / >90% 达成率。说「DistServe 提升 7.4 倍吞吐」时不带「90% SLO 达成率」，就是把 Goodput 偷换成 Raw QPS。

Splitwise（ISCA 2024，Microsoft）从另一角度切入：刻画发现 LLM 推理有计算密集的 prompt 阶段与访存密集的 token 生成阶段，**token 生成阶段严重低设计算资源利用率**，不需要最新 GPU 的算力，可以用更低功耗与成本的硬件。做法是把两阶段拆到**不同机器**、为每阶段选合适硬件并独立配置资源；跨机拆分需要状态转移，论文用集群的高速背板互联实现并优化。结果：相比当时设计**可达 1.4 倍吞吐且成本低 20%**，或**同等成本与功耗下 2.35 倍吞吐**。

对比项	DistServe	Splitwise
核心主张	解耦 + 按 SLO 联合优化资源与并行	解耦 + 按阶段选异构硬件
优化目标	每 GPU Goodput	吞吐 / 成本 / 功耗
关键洞察	混合部署耦合了两阶段的资源与并行策略	Decode 不需要最新 GPU 的算力
报告结果	7.4× 请求或 12.6× 更紧 SLO（>90% 达成）	1.4× 吞吐 @ −20% 成本；2.35× @ 同等成本功耗

> 【补充】TaiChi 把「聚合」与「解耦」统一到一个框架、按负载切换；MLC Microserving 做跨引擎编排。这说明**解耦不是非黑即白，而是可调的设计空间**——轻负载时聚合更省，重负载或异构硬件时解耦更优。

7.4 KV 传输：解耦最核心的工程挑战

以 7B 级 GQA 模型为例（32 层、8 个 KV 头、`d_head=128`、FP16）：

```
per_token = 2 × 32 × 8 × 128 × 2 B = 131,072 B = 128 KiB
4096 token 的请求 → 128 KiB × 4096 = 512 MiB
```

互联	有效带宽【量级】	512 MiB 的传输时间【量级】
节点内 NVLink	数百 GB/s	毫秒级
跨节点 IB/RoCE 400G	数十 GB/s	十毫秒级
跨节点 100G 以太网	~10 GB/s	数十毫秒级

> 【推断】把这张表和 SLO 对起来：**KV 传输时间必须显著小于 TTFT 预算**，否则解耦赚到的干扰收益会被传输吃掉。这就是 DistServe 强调「按集群带宽放置」的原因——同机架、同叶交换机优先。

vLLM 把传输抽象成 **KV Connector**：P 实例算完 KV 后通过 Connector 写到 D 实例可访问的存储/内存，D 实例按块表读取。

后端	适用	特点
NIXL	NVIDIA 点对点传输库，支持 RDMA/GPU Direct	生产常用（`NixlConnector`）
NCCL	已有 NCCL 环境	复用集合通信栈，配置简单
共享存储 / CPU 中转	无 RDMA 环境	带宽与延迟最差，仅适合验证

> 【补充】vLLM 的解耦预填充目前标注为实验特性，通过 `kv_transfer_config` 指定 connector 与 P/D 地址；它要求 P 与 D 的模型、并行度、KV 布局（块大小、层数、头数）一致，否则块无法直接映射。

传输之外还有三笔账：**显存**（D 侧要预留接收缓冲，P 侧 KV 在传完前不能释放）、**调度**（P 算完到 D 可用之间有「在途」状态，调度器要能表达）、**失败**（传输失败要能重试或回退到本地重算，不能丢请求）。

7.5 Goodput 与 P/D 配比

Raw QPS 会骗人：QPS 很高但大部分请求超时，用户根本用不爽。**Goodput 只统计满足 SLO 的那部分**：

```
Goodput = 单位时间内「TTFT ≤ SLO_TTFT 且 TPOT ≤ SLO_TPOT」的请求数
```

于是调度器要做的不是「尽量多接」，而是「**尽量多接且不违约**」——引出准入控制（队列过长时主动拒绝或降级，比全部超时更好）与 SLO 分级（不同优先级给不同 batch 配额与抢占优先级）。

配比的一阶推导：

```
N_P = ceil( λ × L_in  / (G_P × U) )        λ 到达率、L 输入长度
N_D = ceil( λ × L_out / (G_D × U) )        G 单卡产能、U 目标利用率（如 0.7）
```

> 【推断】这个式子忽略了：TTFT 约束要求 P 池有额外余量；TPOT 约束限制 D 池 batch 上限；KV 传输占用 P 池出口带宽。所以实际配比要在压测里迭代——**先按公式给初值，再用「P 池打满 / D 池打满」两个方向找拐点**。
>
> 一个常见失衡模式：**输入长、输出短**（文档问答）P 池压力大；**输入短、输出长**（创作）D 池压力大。用一套固定配比服务两类业务，必然有一边浪费。

7.6 什么时候不该用

不适合	原因
单机、低并发	传输开销大于干扰收益
输入输出都很短	KV 小、干扰本来就小
无高速互联	KV 传输成瓶颈
运维能力有限	调度、配比、排查复杂度显著上升
模型很小（KV 很小）	传输与调度开销占比过高

> 【推断】一条实用判断线：**先用 Chunked Prefill + 前缀缓存把混合部署调到极限，若 TPOT 的 P95 仍达不到 SLO，再考虑解耦**。解耦是「最后一公里」的手段，不是起点。

7.6.1 概念验证（单机可做）

```
# A：纯 Decode 负载（短输入、长输出）
vllm bench serve --dataset-name random --random-input-len 64 \
  --random-output-len 512 --num-prompts 128 --max-concurrency 32 \
  --percentile-metrics ttft,tpot --save-result --result-filename a.json
# B：Decode 负载 + 周期性注入长 Prefill（长输入、短输出），随后与 A 同时运行
```

【推断】预期现象：并发跑时 A 的 TPOT P95 明显高于单独跑（尖刺），这就是干扰的可观测形式；调小 `--max-num-batched-tokens` 或开 Chunked Prefill 后尖刺变缓，就是 2.5 节机制的直接验证。**具体数值必须自己测。**

第 8 章 性能分析与 Benchmark：把「感觉快了」变成可复现的数字

> 对应 AIInfraGuide 模块四 第 8 章（原文只有扉页）。本章按扉页框架、vLLM 官方 bench 的实际 CLI 与通用工程实践重建。

8.1 指标体系

类别	指标	口径要点
延迟	TTFT / TPOT(ITL) / E2E 的 P50/P95/P99	含排队；是否含首 token 要与工具对齐
吞吐	输出 token/s	只算生成 token，最贴近产能
吞吐	总 token/s	含输入 token，适合算总处理量
吞吐	RPS	与长度分布强相关，跨业务不可比
容量	SLO 内最大并发	决定单副本能接多少
资源	显存峰值 / KV 容量 / GPU 利用率	判断还有多少余量
资源	KV 利用率、抢占次数	判断是否过载
质量	Goodput（满足 SLO 的吞吐）	真正该优化的目标

```
      并发 ↑
        ├─► 输出 token/s ↑ 然后趋平（GPU 饱和）
        ├─► TPOT ↑（每步 batch 变大）        ├─► TTFT ↑（排队变长）
        └─► 抢占次数 ↑（KV 不够时）
    KV 占用 ↑ ──► 可容纳并发 ↓ ──► 吞吐 ↓（同时抢占/重算 ↑）
```

> 📌 **同一个系统在不同并发下会给出完全不同的「最优配置」**。任何「优化了 X%」都必须绑定并发与长度分布。

必须记录的元信息：硬件（GPU 型号/数量/互联）；软件（vLLM、PyTorch、CUDA/驱动版本、镜像 digest）；模型（仓库、revision、精度、量化方案、实际 Kernel）；服务配置（TP/PP/DP、`max_model_len`、`max_num_seqs`、`max_num_batched_tokens`、`gpu_memory_utilization`、KV dtype、前缀缓存）；负载（输入输出长度分布、并发、到达率、采样参数、是否 `ignore_eos`）；观测（采样间隔、预热数、重复轮次、是否取中位数）。

8.2 压测工具

子命令	用途	场景
`vllm bench serve`	起服务后压在线接口，输出 TTFT/TPOT/ITL 与吞吐	最常用，贴近真实服务
`vllm bench latency`	测单请求延迟（固定长度）	排除排队看纯计算
`vllm bench throughput`	测离线吞吐（`LLM.generate`）	批处理、离线任务

```
vllm bench serve --backend openai --base-url http://127.0.0.1:8000 \
  --endpoint /v1/completions --model spec-demo --tokenizer Qwen/Qwen3-8B \
  --dataset-name custom --dataset-path data/code.jsonl --custom-output-len 128 \
  --num-prompts 64 --num-warmups 4 --max-concurrency 8 --request-rate inf \
  --temperature 0 --top-p 1 --ignore-eos --seed 42 \
  --percentile-metrics ttft,tpot,itl --metric-percentiles 50,95,99 \
  --save-result --result-dir results --result-filename baseline-code-c8.json
```

参数	作用	常见误区
`--dataset-name`	`random` / `sharegpt` / `custom` / `sonnet`	用 `random` 评 N-gram/Suffix 收益（无重复结构）
`--request-rate`	`inf` 闭环满载；给值则开环限速	只测 `inf`，看不到到达率维度
`--max-concurrency`	并发上限	与 `--request-rate` 混用导致口径不清
`--num-warmups`	预热请求数	不预热，首批含编译与图捕获
`--ignore-eos`	强制生成满长度	用它评质量（输出会超长）

参数	作用	常见误区
`--percentile-metrics`	输出哪些分位	只看 mean，掩盖尾延迟

> ⚠️ **闭环（`inf`）与开环（固定 rate）测的不是一回事**：闭环测系统极限吞吐，开环测给定到达率下的延迟。**生产容量规划要用开环**，因为真实流量是到达率驱动的。
>
> 【补充】GenAI-Perf（NVIDIA，Triton 工具链一部分）一次输出 TTFT、ITL、输出 token 吞吐，支持 OpenAI 兼容端点；Triton Perf Analyzer 更底层，适合模型级压测。两者与 `vllm bench` 的指标口径**不完全一致**（尤其 ITL 是否含首 token），跨工具对比前必须核对定义。

自定义脚本的六个要点：长度分布要用生产日志的 P50/P95 分位；预热与稳态分离；多轮取中位数；保存原始 JSON；记录失败请求（超时与 5xx 不能从分母消失）；固定随机种子（但随机采样下逐字输出仍可能不同）。

8.3 性能分析工具

层次	工具	看什么
端到端	压测工具 + 日志	TTFT/TPOT/吞吐、KV 利用率、抢占
算子级	`torch.profiler`	哪些 Kernel 占比高、有无空洞
Kernel 级	Nsight Systems / Compute	时间线、SM 占用、访存效率

```
with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
             record_shapes=True) as prof:
    llm.generate(prompts, sampling_params, use_tqdm=False)
print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=20))
```

看三件事：① CUDA 时间 top20 是否符合预期（GEMM/Attention/Norm）；② CPU 时间里是否有大量 `cudaLaunchKernel`（launch 开销大 → 需要 CUDA Graph）；③ 时间线上 GPU 是否有长空隙（等 CPU 还是等数据）。

> 【补充】实用排查顺序：**先用 Nsight Systems 找「哪里有空隙」，再用 Nsight Compute 看「空隙旁边的 Kernel 为什么慢」**。反过来（一上来就 profile 单个 Kernel）容易局部优化，对端到端没帮助。

8.4 权威基准与回归门禁

> 【补充】MLPerf Inference（MLCommons）分 Datacenter 与 Edge/Mobile 赛道，场景包括 Offline（离线批处理）与 Server（在线，带到达率与延迟约束）。加入 LLM 任务后指标从 QPS 扩展为「在 TTFT/TPOT 约束下的吞吐」。价值在于规则严格、可横向对比：提交必须给出完整配置与精度验证。

基准	适合回答	不适合回答
MLPerf Inference	这套硬件+软件的官方成绩	我的业务负载该配多少卡
厂商 benchmark 博客	某优化的大致收益	我的场景一定也这样
自建压测	我的 SLO 下最优配置	跨公司可比

门禁规则的三要素——原文扉页给的示例是「**TPOT P95 退化 > 5% 则 Block Merge**」：

1. **选对指标**：用 P95 而非 mean，用 TPOT 而非吞吐（吞吐对噪声不敏感，反而掩盖尾延迟退化）。
2. **定阈值**：5% 是经验值，必须**大于测量噪声**（同配置重复跑的波动）。
3. **定动作**：Block 而不是告警，否则门禁形同虚设。

指标	建议阈值	动作
TPOT P95	退化 > 5%	Block
TTFT P95	退化 > 10%	Block
输出 token/s	下降 > 5%	Block
峰值显存	上升 > 5%	Warn
质量（任务通过率）	下降 > 1pp	Block
启动时间	上升 > 30%	Warn

> 【推断】阈值不是越严越好：GPU 时钟、邻居干扰、NUMA 放置都会带来几个百分点的波动；阈值定得比噪声还小，门禁会变成「随机红灯」，团队很快习惯性绕过。

退化定位五步：① 同 commit 跑三次确认可复现；② `git bisect` 定位引入 commit；③ 看配置差异（Kernel 回退、Attention 后端变化、编译缓存失效）；④ 新旧各跑一次 `torch.profiler` 做 top20 diff；⑤ Nsight 时间线对齐比较找新增空洞。

第 9 章 生产部署与运维：从 Demo 到稳定服务真实流量

> 对应 AIInfraGuide 模块四 第 9 章（原文只有扉页）。本章按扉页框架、vLLM 生产指标文档与 Kubernetes/Ray Serve 文档重建。

9.1 容器化与 K8s

决策	选项	权衡
权重放哪	打进镜像 / 启动时下载 / 挂 PVC 或对象存储	打镜像最稳但镜像巨大；下载简单但依赖外网；PVC 折中
基础镜像	官方镜像 / 自建	省事 vs 可控（需自己跟 CUDA 与驱动）
启动方式	单副本 / 多副本 + 网关 / Ray Serve	见 9.3

> 【补充】权重动辄数十 GB，打进镜像不现实（层大小、拉取时间、仓库成本）。常见做法是权重放对象存储或 PVC，镜像只装运行时；用 initContainer 预拉到 emptyDir/PVC，让主容器启动时间可控。

```
containers:
  - name: vllm
    image: vllm/vllm-openai:latest
    args: ["--model=Qwen/Qwen2.5-7B-Instruct", "--tensor-parallel-size=2"]
    resources: { limits: { nvidia.com/gpu: 2 } }     # 按卡申请，TP 度必须与卡数匹配
    startupProbe:                                     # 专治「启动慢但启动后正常」
      httpGet: { path: /health, port: 8000 }
      failureThreshold: 30
      periodSeconds: 10
    readinessProbe:
      httpGet: { path: /health, port: 8000 }
      initialDelaySeconds: 120
```

> ⚠️ **探针阈值是本模块最容易踩的坑之一**。vLLM 启动要加载权重、`torch.compile` 编译、捕获 36 档 CUDA Graph，**首次拉起可能需要数分钟**。若按普通 Web 服务的 10 秒设 liveness，Pod 会在就绪前被反复重启，形成 `CrashLoopBackOff`——表面像「模型加载失败」，实际是探针把它杀了。K8s 1.16+ 的 `startupProbe` 比调大 `initialDelaySeconds` 更精确：它成功前 liveness/readiness 都不执行。

方案	定位	适合
Deployment + Service	最简单	单模型、小规模、自写网关
KServe	K8s 上的模型服务标准	多模型、canary、自动扩缩容
vLLM Production Stack	vLLM 官方参考实现	开箱的 router + 监控 + 缓存
Ray Serve	编程式编排	复杂 pipeline、多模型组合

9.2 可观测性

指标	类型	含义	用途
`vllm:num_requests_running`	Gauge	正在 GPU 上跑的请求数	并发水位
`vllm:num_requests_waiting`	Gauge	排队请求数	**排队过长 = 需扩容**
`vllm:gpu_cache_usage_perc`	Gauge	KV 使用率（1 = 100%）	接近 1 会触发抢占
`vllm:cpu_cache_usage_perc`	Gauge	CPU KV 使用率	判断是否在换出
`vllm:gpu_prefix_cache_hit_rate`	Gauge	前缀块命中率	判断 Prompt 是否「前置」得好
`vllm:num_preemptions_total`	Counter	累计抢占次数	**增长 = 显存压太满**
`vllm:prompt_tokens_total`	Counter	累计 Prefill token	算 Prefill 吞吐
`vllm:generation_tokens_total`	Counter	累计生成 token	算 Decode 吞吐
`vllm:time_to_first_token_seconds`	Histogram	TTFT 分布	**P95/P99 是核心 SLO**
`vllm:time_per_output_token_seconds`	Histogram	TPOT 分布	**P95/P99 是核心 SLO**
`vllm:e2e_request_latency_seconds`	Histogram	端到端延迟	用户感知
`vllm:request_success_total`	Counter（带 `finished_reason`）	成功请求数	成功率
`vllm:spec_decode_num_draft_tokens_total` / `_accepted_tokens_total`	Counter	投机的 `D` / `C`	接受率

四个必须有的告警：

告警	条件	含义与动作
排队积压	`num_requests_waiting` 持续 > 阈值	扩容或降级，否则 TTFT 会崩
KV 打满	`gpu_cache_usage_perc` > 0.95 持续	降并发/加卡/降 `max_model_len`
抢占频发	`rate(num_preemptions_total[5m])` 高于基线	显存不足
尾延迟违约	TTFT/TPOT P95 > SLO 持续	先看是否过载，再看是否退化

> 📌 这四个覆盖绝大多数线上故障。**不要一开始就上几十条告警**——告警疲劳比没有告警更糟。

结构化日志至少要能回答「这个请求为什么慢」：

```
{"request_id":"...","prompt_tokens":812,"output_tokens":256,"ttft_ms":210,
 "tpot_ms":18.4,"queue_ms":12,"kv_hit_tokens":768,"preempted":false,"finish_reason":"stop"}
```

> 【推断】`kv_hit_tokens` 与 `preempted` 的价值被严重低估：前者解释「为什么这个请求 TTFT 特别低」（命中了长前缀），后者解释「为什么特别慢」（被抢占后重算）。没有它们，尾延迟排查只能靠猜。

9.3 扩缩容与负载均衡

指标	适合 HPA	原因
CPU 利用率	❌	瓶颈在 GPU，CPU 常年不高
GPU 利用率	⚠️	90% 既可能是健康满载，也可能在排队
排队请求数	✅	直接反映供给不足
KV 使用率	✅	反映容量水位
TTFT P95	✅	最贴近体验，但需 Prometheus Adapter

> 【补充】K8s HPA 默认只支持 CPU/内存；按自定义指标扩容要部署 **Prometheus Adapter**（或 KEDA），把 `vllm:num_requests_waiting` 注册成 `custom.metrics.k8s.io` 资源再在 HPA 引用。
>
> ⚠️ **扩缩容的时间常数严重不对称**：扩容要经历「调度 Pod → 拉镜像 → 加载权重 → 编译 → 捕获 CUDA Graph → 就绪」，可能几分钟；缩容瞬间释放。所以必须 `minReplicas ≥ 2`、缩容窗口设长（10 分钟级）、预留「预热池」或接受扩容期间排队。

路由策略	做法	适用
轮询 / 最少连接	网关默认	无共享前缀的通用流量

路由策略	做法	适用
前缀感知	按 Prompt 前缀哈希固定到副本	大量共享 System Prompt / few-shot
KV 感知	查各副本缓存命中再决定	需副本间共享缓存索引，复杂度高

```
无前缀路由：                          前缀感知路由：
  req(S) → 副本0 (miss)                 req(S) → 副本0 (hit)
  req(S) → 副本1 (miss)                 req(S) → 副本0 (hit)
  req(S) → 副本2 (miss)                 req(S) → 副本0 (hit)
命中率 ≈ 1/N                          命中率 ≈ 1
```

> 【补充】vLLM Production Stack 与 Gateway API Inference Extension 都提供 prefix-aware router。自建时最简单的做法是对 messages 的前 N 个 token 做一致性哈希，N 取 System Prompt 的长度量级。
>
> ⚠️ **vLLM 的 OpenAI 兼容服务默认不做认证**，直接暴露公网等于开放算力与数据。生产必须在网关层加：API Key 校验（`--api-key` 只是最基础一层且明文比对）、TLS 终止、网络隔离（Ingress 白名单、内网 Service）、速率限制与配额、请求日志脱敏。

9.4 容量规划

```
所需副本数 N = ceil( 峰值 RPS / 单副本 SLO 内 RPS ) × (1 + 冗余系数)
```

三步法：① 在目标 SLO 下压测单副本产能（用**真实长度分布**）；② 算理论副本数；③ 加 1.3–1.5 的冗余（覆盖突发、滚动升级、单副本故障）。

算例（**全部为演示假设，不是实测**）：`λ=20 req/s`、输入/输出 512/256、SLO 为 TTFT P95 ⩽ 800 ms 且 TPOT P95 ⩽ 50 ms、单副本 SLO 内 8 req/s、冗余 1.4 → 理论 2.5 → 3 副本 → 加冗余 `ceil(3×1.4) = 5`。

再验算显存：`per_token = 128 KiB`（7B GQA、FP16）、并发 32、长度 768 →

```
KV ≈ 32 × 768 × 128 KiB ≈ 3 GiB
```

要和权重（约 14 GB）一起塞进显存预算。**若 KV 需求超过预算，实际并发会被压低、单副本产能随之下降——「产能」与「显存」必须联立求解。**

> 【推断】容量规划最容易犯的错是「用平均长度算显存」。KV 按**峰值长度**占用（PagedAttention 只省预留，不省实际写入），要用 P95/P99 长度算。

单位成本比「每 GPU 吞吐」更贴近业务：

```
每百万输出 token 成本 = (GPU 小时成本 × 副本数 × 卡数) / (输出 token/s × 3600 × 1e6)
```

低利用率的高端卡，单位成本往往高于高利用率的低端卡。

第 10 章 端侧推理：在别人的手机上跑模型

> 对应 AIInfraGuide 模块四 第 10 章（`101-端侧推理基础` 正文）。数据中心部署先问「一张 GPU 能跑多少请求」；到了手机、汽车、摄像头与嵌入式设备上，问题变成：模型能不能装进内存？第一次启动要多久？连续跑十分钟会不会因发热降频？某个算子没被 NPU 支持时，是报错还是悄悄回退 CPU？

10.1 四类约束

端侧（Device，资源约束最强）、边缘侧（Edge，含基站/门店服务器/网关）、云端（Cloud，算力散热最宽松）经常被混用。把模型放到端上的四个理由：**低延迟**（摄像头检测、语音唤醒、驾驶辅助不能等网络往返）、**离线可用**、**隐私**（原始图片/语音/文本留在本地）、**成本**（高频轻量请求本地完成）。代价是工程约束从「吞吐与成本」扩展为「延迟、内存、包体积、功耗、温度、精度、兼容性」。

约束	要点
内存	同时占模型文件、权重、激活、工作区、运行时；**「物理内存 8 GB」≠「模型可用 8 GB」**（OS 限制单 App）
功耗散热	短时峰值快不代表稳态快，持续运行会降频 → **关注稳态性能**
硬件碎片化	两台同为 Android 的手机，指令集、GPU 驱动、NPU 工具链都可能不同；同一计算图在 A 上可全委托 NPU，在 B 上可能若干算子回退 CPU
应用生命周期	模型加载会不会阻塞 UI？后台是否释放内存？更新失败能否回滚？

```
M_weights ≈ P × b / 8            P 参数量、b 每参数字节
3B 模型：FP16 6 GB / INT8 3 GB / INT4 1.5 GB          ← 只是理论下界
M_KV ≈ 2 × L × B × T × H_kv × D × s
```

> 📌 权重加得进内存，**不代表能稳定支持很长上下文**——KV 与激活会把它推过界。

10.2 从模型到 App

```
PyTorch ──► torch.export ──► 常量折叠/融合/量化 ──► CPU/GPU/NPU 委托 ──► .pte/.onnx/.tflite ──►
```

三个极易混淆的层次：

层次	是什么	例子
模型格式 / IR	描述计算图与权重	ONNX、`.pte`、`.tflite`
Runtime	加载模型、管理内存、发起执行	ExecuTorch、ONNX Runtime、ncnn、MNN
Backend / Delegate	真正执行一部分计算	XNNPACK、Core ML、Vulkan、QNN

Runtime 不一定亲自算每个算子：它把支持的子图交给硬件后端，剩余留给 CPU Kernel，这叫**图分区（Partitioning）/ 委托（Delegation）**。

> 📌 **「启用了 NPU」≠「整个模型都跑在 NPU 上」**。端侧优化第一步是确认实际分区结果——这是最常出现的自欺。

硬件	优势	约束
CPU	算子覆盖最全、调试最好、兼容性最强	峰值算力低；要管 NEON/SVE、线程与大核小核、Cache
GPU	并行度高	小算子摊不薄调度；CPU↔GPU 转换与同步贵；驱动质量参差；**不保证比高度优化的 ARM CPU 快**
NPU	能效高，专为矩阵/卷积/低精度优化	算子覆盖、Layout、动态形状、量化规则严格；出现不支持的节点就切图
DSP	低功耗持续处理规则数据流	算力不高；适合音频、传感器、常驻任务

> 【推断】可靠的选型方法：**先用 CPU 建立准确性基线，再分别测 GPU/NPU/DSP 的覆盖率、端到端延迟与稳态功耗**，而不是先信「NPU 更快」。

运行时定位（不是性能排行榜）：ExecuTorch（PyTorch 工作流、多后端）、ONNX Runtime Mobile（跨框架互操作）、LiteRT（TF/Google AI Edge）、Core ML（Apple 原生）、ncnn（轻依赖 C++、移动视觉）、MNN（国内移动端、含 LLM）、llama.cpp（GGUF 生态）、MLC LLM（编译器驱动、跨平台 GPU）。

10.3 四个抓手

抓手	做法	注意
模型压缩	结构化剪枝、蒸馏、低秩分解、换更小架构	会改变算子结构，要结合目标硬件 Kernel 支持评估

抓手	做法	注意
量化	PTQ/QAT、Weight-only/W8A8、Per-tensor/Per-channel	**量化不等于加速**：无对应低精度 Kernel 或频繁 Q/DQ 时可能更慢；必须同时报告大小、延迟、内存、精度
图与算子	常量折叠、Conv-BN/Linear-Act 融合、删冗余 Cast/Transpose/Reshape、NCHW↔NHWC、静态形状特化、自定义算子	重点不是节点越少越好，而是**减少访存、调度与后端切换**
内存规划	复用生命周期不重叠的激活、mmap 按需加载、控制上下文与 KV 精度、避免重复拷贝、分片加载 + 明确失败降级	LLM 要单独管 KV

10.4 可信的 Benchmark

类别	记录项
环境	设备型号、SoC、内存、系统版本；Runtime/模型/驱动版本
模型	精度与量化方法、文件哈希
执行	实际后端与**图分区结果**、线程数、输入尺寸、batch、动态形状范围
性能	冷启动与加载时间、预热后 P50/P90/P99、吞吐
资源	峰值与稳态内存、模型与 Runtime 包体积
物理	功耗、温度、持续运行后的性能变化
质量	任务精度、与原模型误差

LLM 还要加：TTFT、TPOT、Prefill/Decode 各自 tokens/s、**最大稳定上下文长度**、是否触发系统内存回收。

协议：① 固定样本与种子；② 先跑原始框架存精度基线；③ **加载与推理分开计时**；④ 预热若干次后采集 ⩾ 数十次；⑤ 同时报中位数与尾延迟；⑥ 连续跑一段看温度与降频；⑦ 检查每个子图的实际后端；⑧ 保存脚本、配置与原始结果。

```
device/os/model(runtime+hash)/precision/input/threads/warmup_runs/measured_runs
delegate_coverage: "委托子图比例或未支持算子"
latency_ms: "P50 / P90 / P99"     peak_memory_mb     power_and_temperature     accuracy
```

> 📌 **模型哈希、Runtime 版本、后端覆盖率尤其重要**：避免「同名模型不是同一份文件」与「号称用 NPU、实际大量回退 CPU」两类误判。

10.5 ExecuTorch 最小实例

```
import torch
from executorch.backends.xnnpack.partition.xnnpack_partitioner import XnnpackPartitioner
from executorch.exir import to_edge_transform_and_lower

model = TinyClassifier().eval()                    # 必须 eval()
ep = torch.export.export(model, (torch.randn(1, 3, 224, 224),))
edge = to_edge_transform_and_lower(ep, partitioner=[XnnpackPartitioner()])
open("tiny.pte", "wb").write(edge.to_executorch().buffer)
```

三步：`torch.export` 捕获图；`XnnpackPartitioner` 标记并降低支持的子图；`to_executorch` 生成 `.pte`。先做 XNNPACK 是因为它不要求芯片厂商 SDK，便于把「导出问题」与「NPU 工具链问题」分开。**生成文件不等于部署完成**，还要验证：导出前后误差、委托了哪些算子、包体积、真机加载时间/延迟/内存、多次与持续运行的稳定性。

> ⚠️ API 随版本演进。示例基于 ExecuTorch 1.3 文档；实践应固定 PyTorch、ExecuTorch 与后端版本。

10.6 排查顺序

现象	排查
导出失败	定位不支持的 Python 控制流/动态形状/自定义算子 → 改写或注册算子。**不要在不知道失败节点时反复换 opset 或 Runtime 版本**
精度不一致	对齐前后处理 → 比较导出前后 FP32 → 比较量化前后 → 比较不同后端 → 存中间 Tensor 定位第一个误差算子
NPU 比 CPU 慢	实际覆盖多少算子 → 是否频繁 CPU↔NPU 转换 → 输入是否太小摊不薄开销 → 是否把加载/编译/预热算进推理 → 量化类型是否真命中 NPU Kernel
首次运行特别慢	反序列化、权重重排、图编译、GPU Pipeline 创建；冷/热启动分开测，缓存编译结果
连续运行越来越慢	温度、频率、线程调度、电量模式；优化目标转向能效与稳态

> 💡 很多「Runtime 精度问题」其实来自 **RGB/BGR、归一化、Resize、NCHW/NHWC 不一致**。先怀疑预处理。

第 11 章 与已有笔记的衔接：把推理优化接回基础课

> 推理优化几乎没有全新原理，它全是操作系统、体系结构、网络、采样理论与优化的组合应用。认出这层映射，记忆负担会小很多。

11.1 六张映射表

推理侧	基础课	说明
KV 传输、流式输出	CS144（网络）	带宽时延积、分块传输；512 MiB 走 100G 要几十毫秒【量级】，直接吃掉 TTFT 预算
前缀感知路由	CS144	一致性哈希、会话保持：把相关请求打到同一后端
Roofline 与算术强度	CSAPP（存储器层次）	算力/带宽比决定平衡点，与「每字节访存的运算数」同构
PagedAttention	CSAPP（虚拟内存）	块=页、块表=页表、引用计数=fd 计数、CoW=fork、LRU=页面置换、抢占=换页
KV 碎片	CSAPP	分配粒度决定内部/外部碎片形态
Continuous Batching	15–418/CS149	动态任务调度：从静态划分到动态填充，消除空转
Chunked Prefill	15–418/CS149	负载均衡、削峰：把大任务切小避免长尾阻塞
CUDA Graph	15–418/CS149	减少调度开销：把 N 次启动合并成 1 次
TP 的 AllReduce	15–418/CS149	集合通信、树形归约；每层 2 次，通信量与激活成正比
PP 的 bubble	15–418/CS149	流水线填充率；低并发时效率低
Prefill/Decode	CS336（语言模型）	训练时的 teacher forcing 前向：已知输入可并行，是同一件事
GQA / RoPE / RMSNorm / SwiGLU	CS336	注意力变体、位置编码、归一化、FFN 激活
投机解码	CS336（采样）	拒绝采样、残差分布、TV 距离；**是精确采样算法，不是工程 trick**
量化映射	EE364a（凸优化）	量化误差最小化；校准是在舍入与截断间做一维搜索
GPTQ	EE364a	带二次型权重的最小二乘：`min ‖X(w−ŵ)‖²`，`H = 2XᵀX`
P/D 配比、容量规划	EE364a	约束优化：在延迟约束下最小化副本数与成本

11.2 三句「迁移直觉」

- **PagedAttention 就是「给 KV Cache 装了一个操作系统」**——这句话不是比喻，是逐项对应。CSAPP 第 9 章读透，2.2 节只需记 vLLM 的工程取舍（V1 不做 CoW）。
- **并行程序的性能由最慢的那个决定**（15-418）在推理里有两个化身：Static Batching 里最长的请求决定整批时间；PP 里最慢的一段决定流水线吞吐。解法也一致——**用动态调度把空闲填上**。
- **先写目标函数，再谈算法**（EE364a）。GPTQ 的推导之所以干净，是因为它把量化误差写成了 `‖X(w−ŵ)‖²` 这个带 `H` 的二次型，而不是「每个权重误差的平方和」。目标函数选错，后面所有技巧都是局部修补——这也是 4.3.6 强调「先固定四个维度再比算法」的原因。

11.3 一句话总表

```
CS144   → 带宽时延积、分块传输          → KV 传输、流式输出
CSAPP   → 虚拟内存/页表/COW/LRU          → PagedAttention、前缀缓存、抢占
CSAPP   → 存储器层次、算术强度           → Roofline、Memory Bound
15-418  → 动态调度、负载均衡             → Continuous Batching、Chunked Prefill
15-418  → 集合通信、流水线               → TP 的 AllReduce、PP 的 bubble
CS336   → 自回归、注意力变体、采样       → 两阶段、GQA、投机解码
EE364a  → 二次型最小化、约束优化         → GPTQ 的 H、P/D 配比、容量规划
```

第 12 章 复习清单与自测题

12.1 一页纸公式

```
KV 每 token     = 2 × L × H_kv × d_head × bytes       (FP16 的 bytes = 2)
KV 总量         = 上式 × B × T
权重            = P × b / 8
Roofline 平衡点 = 峰值算力 / 带宽     H100 BF16: 990 TFLOPS / 3.35 TB/s ≈ 295 FLOP/Byte
Decode 算术强度 ≈ 1 FLOP/Byte；拼批 B 个后 ≈ B
E2E             = TTFT + TPOT × (N − 1)
ITL             = (e2e − TTFT) / (N − 1)              (GenAI-Perf 口径)
仿射量化        q = clip(round(x/s) + z), x̂ = s(q − z)
对称            s = max|x| / 127
INT4 每权重字节 = 4/8 + 2/G          (G=128 → 0.515625 B；8→4 bit 格距约 17×)
SmoothQuant     Y = (XS⁻¹)(SW), s_j = max|X_j|^α / max|W_j|^(1−α)
GPTQ            min ‖XW − X̂Ŵ‖², H = 2XᵀX, Δw_F = −δ_i/[H_F⁻¹]_ii · [H_F⁻¹]_:,i
投机接受        a = min(1, p/q)；r(x) = [p−q]_+ / Σ[p−q]_+
E[L]            = 1 + Σ Pr(K≥i) = (1 − a^(γ+1))/(1 − a)；ΔE[L] = a^(γ+1)
加速条件        E[T_round] < T_0 × E[L]
接受率          C/D；平均接受长度 = 1 + C/R
P/D 配比        N_P = ceil(λ·L_in/(G_P·U)), N_D = ceil(λ·L_out/(G_D·U))
容量规划        N = ceil(峰值 RPS / 单副本 SLO 内 RPS) × 冗余系数
```

十条最该记住的结论：

1. Decode 慢是因为「搬不过来」而非「算不过来」，算术强度约 1 FLOP/Byte。
2. Batching 提吞吐几乎不增延迟，因为瓶颈在权重搬运。
3. PagedAttention 的价值是「省显存 → 提并发 → 提吞吐」，不是省显存本身。
4. Continuous Batching 必须依赖细粒度可回收的显存，两者是乘法。
5. Chunked Prefill 是平稳性优化，代价是长 Prompt 的 TTFT 略增。
6. W4A16 省带宽不省算力；W8A8/FP8 才可能省算力，但要硬件支持。
7. 量化/Draft 的收益要看**启动日志里的实际 Kernel/方法**，不能看模型名字。
8. 投机解码无损但**不保证更快**；接受率只反映 `TV(p,q)`，与成本无关。
9. KV 传输必须显著小于 TTFT 预算，所以 P/D 放置要带宽感知。
10. 任何「快了多少倍」都要绑定口径、基线、硬件与精度。

12.2 自测题（10 题）

A1. 40 层、8 个 KV 头、`d_head=128`、FP16，单请求 8K 上下文：① 每 token KV？② 单请求 KV？③ 64 并发共多少？

A2. 为什么 Decode 是 Memory Bound？用 H100 平衡点定量说明。

A3. TTFT = 300 ms、TPOT = 25 ms、输出 30 token。E2E 多少？TTFT 占比？分别把 TPOT / TTFT 优化 20%，哪个对 E2E 更有效？

B1. `block_size = 4`、Prefill 10 个 token：画出块表，说明 Decode 到第 12 个 token 时如何变化。

B2. V1 为什么不需要 Copy-on-Write？给出两个理由。

B3. 预算 2048、6 个 Decode、running 长请求还差 1500、waiting 新请求需 900：写出 `{req: num_tokens}`，说明下一步怎么续。

C1. 权重里有一个 20.0 的离群值、其余在 `[-0.8,0.8]`：对称 INT8 覆盖全部会怎样？截断到 1.0 呢？分别是什么误差？

C2. SmoothQuant 的 α = 0 与 α = 1 分别意味着什么？为什么不能取极端？

C3. GPTQ 与 AWQ 都用校准数据，用途有何不同？

C4. 为什么 INT4 模型在 Batch 很大时收益下降？用 Roofline 解释。

D1. `p = [0.6,0.3,0.1]`、`q = [0.2,0.5,0.3]`：接受率？残差分布？最终分布？若拒绝后直接从 `p` 重抽会得到什么？

D2. 各位置条件接受概率都是 0.8：`γ = 4` 与 `γ = 8` 的 `E[L]` 各是多少？为什么继续加长收益变小？

D3. 为什么首个拒绝后必须停止，不能跳过错误位置继续用后面的候选？

E1. 混合部署下 TPOT P95 被拖慢，按什么顺序尝试哪些手段？至少四级。

E2. 7B GQA（32 层、8 KV 头、`d_head=128`、FP16）、4096 token：KV 多大？走 100G 以太网要多久【量级】？对 PD 解耦意味着什么？

12.3 答案

A1. ① `2×40×8×128×2 = 163,840 B = 160 KiB/token`；② `×8192 = 1.25 GiB`；③ `×64 = 80 GiB` ——**单张 80GB 卡的 KV 已经装不下，更不用说权重**。换 FP8 KV 全部减半（80 KiB、640 MiB、40 GiB）。这说明 GQA 已把 KV 压到 1/4，长上下文 + 中高并发仍会爆，所以「KV 量化 + PagedAttention + 前缀缓存」要一起上。

A2. Prefill 一次并行处理 N 个 token，权重搬一次被 N 个 token 共享，算术强度约 `2N`；Decode 每步只处理 1 个 token 却要搬全部权重：搬 `W` 字节、做约 `W` FLOPs，强度约 **1 FLOP/Byte**。H100 平衡点 `990e12/3.35e12 ≈ 295`，**低约两个数量级**，算力利用率理论上不到 1%。

A3. `E2E = 300 + 25×29 = 1,025 ms`，TTFT 占 `300/1025 ≈ 29%`。TPOT −20% → `300+20×29 = 880`，改善 **14.1%**；TTFT −20% → `240+725 = 965`，改善 **5.9%**。**TTFT 只占 29% 时，优化 TPOT 的端到端收益是优化 TTFT 的两倍多。**

B1. `ceil(10/4)=3` 块：L0→token 0–3（满）、L1→4–7（满）、L2→8–9（2/4）。生成第 10、11 个 token 仍写 L2（填满至 4/4）；生成第 12 个时 L2 已满，**才申请新物理块挂到 L3**——按需逐块分配，最多浪费最后一块的空位。

B2. ① **只共享写满的块**：前缀缓存只对已写满的块算哈希并共享，未写满的尾块不参与；新 token 一律追加到请求自己的新块，共享块始终只读，没有「往共享块写」的时刻。② **`n > 1` 拆成独立子请求**：并行采样拆成 n 个 `n=1` 子请求，通过前缀缓存共享 Prompt 的满块，不再需要块内分叉。

B3. 6 个 Decode 各取 1（剩 2042）；长请求 `min(1500, 2042) = 1500`（剩 542）；W1 被截断为 542（剩 0）；W2 留在 waiting。即 `{d1..d6:1, R:1500, W1:542}`。下一步 R 若已算完转 Decode；W1 的 `num_computed_tokens` 记为 542，按 `min(900−542, 剩余预算)` 续算，直到追上 `num_prompt_tokens` 才转 Decode。

C1. 覆盖全部时 `s = 20/127 ≈ 0.157`，`[-0.8,0.8]` 只能落到约 `-5..5` 几个编码，小值相对误差极大（**舍入误差**）；截断到 1.0 时 `s ≈ 0.00787`，小值很准但 20 被压成 1（**截断误差**，误差 19）。**两种都不好**——解法是把离群值转移走（SmoothQuant）或缩小共享组并补偿（GPTQ/AWQ）。

C2. `α=0` → `s_j = 1/max|W_j|`，难度全推给权重（激活不缩放）；`α=1` → `s_j = max|X_j|`，全推给激活。两个极端都会让一侧量程被撑爆：α 太小激活仍有尖峰，太大权重出现新尖峰。论文消融显示 α < 0.4 时激活难量化、> 0.6 时权重难量化，所以要在 `[0.4,0.6]` 附近搜（LLaMA 常用 0.8）。

C3. GPTQ 用校准输入构造 $H = 2X^{\top}X$，衡量在真实输入下每个方向误差的代价，把已量化权重的误差补偿给尚未量化的权重；AWQ 用校准数据统计**激活幅值**，找那 1% 显著通道，再搜索每通道缩放因子来保护它们。**一个是构造误差度量矩阵，一个是找要保护的通道。**

C4. INT4 省的是「读权重的字节数」。小 Batch 时算术强度低、卡在带宽斜坡，省字节直接转化为提速；Batch 变大后权重只搬一次却服务更多 token，算术强度上升并越过平衡点，**瓶颈从带宽转到算力**，少读权重不再有用，甚至因解包/反量化的额外开销变慢。Marlin 的「Batch 16–32 接近 4× 上限、64–128 递减」正是这个机制。

D1. `A = Σmin(p,q) = 0.2+0.3+0.1 = 0.6`（接受率 60%）；`β = 0.4`，`[p-q]_+ = [0.4,0,0]`，残差 `r = [1,0,0]`；最终 `= [0.2,0.3,0.1] + 0.4×[1,0,0] = p`。若拒绝后直接从 `p` 重抽，得 `[0.2,0.3,0.1] + 0.4×[0.6,0.3,0.1] = [0.44,0.42,0.14]`——第二个 token 被高估 40%。

D2. `γ=4`：`(1-0.8^5)/0.2 = 3.3616`；`γ=8`：`(1-0.8^9)/0.2 ≈ 4.329`。候选多了 4 个，`E[L]` 只多约 0.97。边际收益 `a^(γ+1)`：`0.8^5 ≈ 0.328`、`0.8^9 ≈ 0.134`，越往后越小，而每候选的成本不降——**超过某长度就是净亏损**。

D3. 每个候选的验收依据是它前面的历史。`y3` 被拒、换成 `z3` 后，`p4` 是以 `h,y1,y2,y3` 为条件算的，而真实历史已是 `h,y1,y2,z3`，`y4` 的验收依据失效。**每轮只能提交「连续接受前缀 + 1 个修正/额外 token」。**

E1. 四级递进：① 开 Chunked Prefill + 调小 `max_num_batched_tokens`（如 2048），把长 Prefill 摊到多步，代价是长 Prompt 的 TTFT 略增；② 开前缀缓存，若长 Prompt 有共享前缀可直接砍掉大部分 Prefill；③ 给长 Prefill 设 `long_prefill_token_threshold`，限制单请求每步吃掉的预算；④ 前三级仍达不到 TPOT SLO 时，才做 PD 解耦。

E2. `per_token = 2×32×8×128×2 = 131,072 B = 128 KiB`，`4096 token → 512 MiB`；100G 以太网有效带宽约 10 GB/s【量级】，传输约 **50 ms**【量级】。这意味着 KV 传输会直接吃掉几十毫秒的 TTFT 预算——解耦必须把 P/D 放在高带宽低延迟的互联上（同机架/同叶交换机或同一 NVLink 域），否则「消除干扰」的收益会被传输抵消。

12.4 复习路线

时间	内容	目标
0–5 min	12.1 公式表	默写 KV、E2E、E[L]、加速条件
5–12 min	A 组 + B1/B3	恢复手算能力

时间	内容	目标
12—20 min	C 组	说清四类量化的目标与代价
20—26 min	D 组	独立推出保分布证明
26—30 min	E 组	给出分级优化方案与容量估算

> 📌 只有 10 分钟时：**只背 12.1 的十条结论 + A2 + D1**，覆盖面试里 80% 的追问。

第 13 章 参考来源

13.1 一手来源（AIInfraGuide 模块四，访问日期 2026-11）

页面前缀 `https://caomaolufei.github.io/AIInfraGuide/inference/模块四-推理优化/`，共 29 个页面。

章	页面	本笔记对应
1	`第1章-llm推理基础/11-llm推理基础`、`12-vllm快速入门`	第 1 章
2	`第2章-推理引擎核心技术/21-pagedattention`、`22-continuous-batching`、`23-prefix-cache-与-radixattention`、`24-chunked-prefill-与统一调度`、`25-attention-后端与图优化`	2.2～2.6
3	`第3章-nano-vllm源码解读/31-项目全景与四层架构` ～ `36-modelrunner引擎循环与性能`（6 篇）	第 3 章
4	`第4章-量化/41-量化基础` ～ `46-量化选型与vllm实战`（6 篇）	第 4 章
5	`第5章-speculative-decoding/51-投机解码核心原理` ～ `55-vllm投机解码实战`（5 篇）	第 5 章
6	`第6章-分布式推理`（扉页）	第 6 章（补充重建）
7	`第7章-pd解耦架构`（扉页）	第 7 章（补充重建）
8	`第8章-性能分析与benchmark`（扉页）	第 8 章（补充重建）
9	`第9章-生产部署与运维`（扉页）	第 9 章（补充重建）

章	页面	本笔记对应
10	`第10章-端侧推理/101-端侧推理基础`	第 10 章

13.2 补充来源

文献 / 文档	出处	用于
Kwon et al., *PagedAttention*	SOSP 2023	2.2；20.4%–38.2% 利用率、CoW、抢占
Yu et al., *Orca*	OSDI 2022	2.3；迭代级调度、36.9×
Agrawal et al., *SARATHI*	2023	2.5；等长切块
Zheng et al., *SGLang*	2023/24	2.4；RadixAttention、6.4×
Dao et al., *FlashAttention* 系列	2022–2024	2.6；IO 感知、varlen、FA2/FA3
Frantar et al., *GPTQ* / *Marlin*	ICLR 2023 / 2024	4.3；$H=2X^TX$、解包融合
Lin et al., *AWQ*	MLSys 2024	4.3；1% 显著通道、TinyChat 3×
Xiao et al., *SmoothQuant*	ICML 2023	4.2；等价缩放、α、1.56×
Liu et al., *KIVI*	ICML 2024	4.4；K/V 不同分组、2.6×
Micikevicius et al., *FP8 Formats* / OCP *MX v1.0*	2022 / 2023	4.5；E4M3/E5M2、MXFP4
Leviathan et al. / Chen et al., *Speculative Decoding*	ICML 2023 / 2023	5.1；接受规则与残差
Cai et al., *Medusa*；Li et al., *EAGLE 1–3*	2024 / 2024–25	5.3；多头、动态树、TTT
DeepSeek-AI, *DeepSeek-V3*	2024	5.3；MTP、85%–90%、1.8×
Sun et al., *Block Verification*	2024	5.3
Zhong et al., *DistServe*	OSDI 2024	第 7 章；7.4×/12.6×（>90% 达成）
Patel et al., *Splitwise*	ISCA 2024	第 7 章；1.4×/–20% 成本、2.35×

文献 / 文档	出处	用于
Shoeybi et al., *Megatron-LM*；Su et al. *RoFormer*；Zhang & Sennrich *RMSNorm*；Shazeer *GLU*	2019—2021	3.4、第 6 章
vLLM v0.32.0 文档（Engine Args / Tuning / Design / Metrics / Spec Decode / Quantization / Disagg Prefill）	docs.vllm.ai	2.1、2.4～2.6、4.4～4.6、5.2～5.5、7.4、9.2
nano-vllm 仓库；PyTorch CUDA Graphs 与 multiprocessing 文档	GitHub / pytorch.org	第 3 章
NVIDIA（CUDA Graph、NVFP4）、ONNX（QuantizeLinear/FP8/FP4）、LLM Compressor	各自官网	2.6、4.1、4.4、4.5
Ray Serve、Kubernetes（HPA/Probes/Gateway API）、MLPerf Inference、ExecuTorch/ORT/LiteRT/Core ML/ncnn/MNN/llama.cpp/MLC	各自官网	第 6、8、9、10 章

13.3 关键数字溯源

数字	含义	出处与限定
20.4%—38.2%	旧系统 KV 显存中真正存了 token 的比例	vLLM 论文，当时系统与负载
36.9×	Orca 对比 FasterTransformer	GPT-3 175B、相同延迟，含其他优化
6.4×	SGLang 吞吐提升	结构化 LLM 程序负载，含其他优化
295 FLOP/Byte	H100 BF16 平衡点	990 TFLOPS 稠密 + 3.35 TB/s
1.56×	SmoothQuant 提速	其测试模型，精度损失可忽略
3—4 GPU 小时	GPTQ 量化 175B	特定硬件
3× / 2.8×	AWQ-TinyChat / Marlin 端到端	桌面移动 GPU / 特定配置
2.6× / 4× / 2.35—3.47×	KIVI 显存 / batch / 吞吐	Llama、Falcon、Mistral
85%—90% / 1.8×	DeepSeek-V3 MTP 接受率 / TPS	该模型与部署条件
7.4× / 12.6×	DistServe 请求数 / SLO 收紧	>90% 请求满足延迟
1.4× / 2.35×	Splitwise 吞吐	前者成本低 20%，后者同等成本功耗
28 MiB / 112 KiB	nano-vllm 每块 / 每 token KV	Qwen3-0.6B、block 256、bf16
1434.13 vs 1361.84 tok/s	nano-vllm vs vLLM	RTX 4070 Laptop、0.6B、离线
3—5×	混合 batching 下 Decode P95 TPOT 被拖慢	AIInfraGuide 第 7 章扉页，未给负载

> ⚠️ 引用任何数字时必须同时引用限定条件。把 7.4× 当通用结论、把 36.9× 当 Continuous Batching 的单独收益，都是对来源的误用。

13.4 三处最不确定与两处时效风险

1. **第 6 章全部为补充重建**：原文该章未公开正文。TP/PP/DP/EP 的描述、每层 2 次 AllReduce、Ray 建议来自 Megatron 论文、vLLM 文档与工程经验；「TP 不跨节点」「PP 低并发效率低」是工程共识，但**具体带宽拐点没有给数字**——高度依赖互联与模型，给了反而误导。
2. **第 7 章的 P/D 配比公式是一阶估算**，忽略了 TTFT 约束对 P 池的额外余量、TPOT 约束对 D 池 batch 上限的限制、KV 传输占用 P 池出口带宽。**只能给初值，不能定最终配置。**
3. **第 9 章的容量算例全部是假设值**（20 req/s、8 req/s、1.4 冗余），只演示方法，不对应任何真实部署。**任何真实容量数字都必须来自你自己的压测。**

两处时效风险：vLLM 的参数名与默认值变化较快（本文以 **v0.32.0** 为准），换版本后以 `vllm serve --help` 与对应版本文档为准，尤其 `--quantization` 的在线方案名、`--speculative-config` 字段、`cudagraph_mode` 取值；第 10 章的 ExecuTorch API 基于 1.3 文档，版本演进后可能变化。

> **最后一句**：这个模块里没有一个技巧是银弹。PagedAttention 省显存但增加 Kernel 复杂度，Continuous Batching 提吞吐但让延迟对负载敏感，量化省带宽但可能损质量，投机解码降 TPOT 但吃容量。**真正的工程能力，是在给定 SLO、硬件与负载下判断「此刻该动哪个旋钮」——而这需要先测清楚瓶颈在哪里。**