大模型推理系统与实践

第 5 讲 KV Cache 管理

张书豪

华中科技大学计算机学院

2027 · 研究生课程课件

教学目标

核心结论

本讲将 KV Cache 作为影响并发、调度与服务稳定性的核心系统对象进行讨论。

- 理解 KV 生命周期与容量压力。
- 理解分页、块管理、碎片与回收。
- 说明 KV 组织方式如何反向影响调度器能力。

为什么 KV 是系统核心对象

核心结论

KV 既是性能关键状态，又是容量主约束，因此它会塑造系统设计空间。

- 随上下文长度和并发增长快速膨胀。
- 需要跨 decode 多轮驻留。
- 既影响时延，也限制 batch 扩展空间。

为什么 KV 问题会在大模型时代被放大

核心结论

随着上下文更长、并发更高、交互更持续，KV 已经从实现细节变成系统一等约束。

- 长上下文让单请求状态膨胀。
- 在线服务要求 KV 状态跨自回归解码步保留；是否跨对话轮次复用取决于会话路由与前缀缓存机制。
- 多租户与动态到达让分配与回收更频繁。

KV 生命周期

核心结论

理解 KV，首先要理解它如何生成、追加、复用、迁移和回收。

1. prefill 生成首轮 KV
2. decode 持续追加
3. 可能发生前缀复用或跨请求共享
4. 完成后释放或转入更低层存储

连续地址为什么会成为问题

核心结论

如果逻辑连续序列必须映射到物理连续大块显存，动态服务很快会受到碎片和预留浪费限制。

- 并发请求越多，连续空间越难保证。
- 预留过大则浪费，预留过小则频繁失败。
- 这正是分页式 KV 管理出现的直接背景。

PagedAttention 的设计直觉

核心结论

PagedAttention 的关键在于将逻辑连续状态映射为物理离散块。

- 避免显存必须连续分配大块空间。
- 降低内部碎片与预留浪费。
- 为更动态的请求组织提供底层支撑。

PagedAttention 的系统意义

核心结论

PagedAttention 的价值，不只在于节省显存，更在于把状态组织方式改造成了支持动态调度的形式。

- 更灵活的状态分配与回收。
- 更小的显存预留压力。
- 更强的 continuous batching 支撑能力。

为什么 KV 会反向约束调度

核心结论

调度器看似管理请求，但真正决定并发上限的常常是状态驻留能力。

- 如果 KV 不能灵活管理，continuous batching 只会更快碰到容量墙。
- 如果回收和复用代价过高，调度收益会被状态管理开销抵消。
- 这就是为什么 vLLM 的调度贡献必须和 PagedAttention 一起理解。

前缀复用为何进一步提升复杂度

核心结论

一旦系统开始尝试前缀复用，KV 就不再只是“每个请求私有状态”，而开始成为跨请求共享对象。

- 共享带来更高潜在收益。
- 也带来正确性、元数据组织和回收边界问题。
- 这使 KV 进一步接近广义状态管理问题。

源码穿插：nano-vllm-hust 中的 KV 管理结构

核心结论

nanovllm/engine/block_manager.py 将课堂中的块管理、引用计数与前缀哈希机制压缩到了一个较短文件中。

- Block 维护块编号、引用计数、哈希值与块内 token。
- free_block_ids 与 used_block_ids 体现了显式的块生命周期管理。
- hash_to_block_id 为前缀块复用提供查找入口。

源码穿插：读 BlockManager 抓四个问题

核心结论

读 BlockManager 时，可以先抓住“能否分配、如何分配、何时回收、何时哈希”四个问题。

- `can_allocate()` 体现了前缀缓存命中与新块需求估算。
- `allocate()` 和 `deallocate()` 对应块表构造与引用计数回收。
- `hash_blocks()` 则把完整块写回全局哈希表，为后续复用建立条件。

讲解：为什么 KV 是系统对象而不是实现细节

核心结论

到了大模型服务场景，KV 不再是 attention 的暂存结果，而是长期驻留、持续增长、反向约束调度的核心状态。

- 单请求上下文越长，KV 规模越大；并发越高，系统里长期保留的状态越多。
- 因为 KV 持续存在，它会直接决定还能否接纳新请求、还能否稳定做 continuous batching。
- 所以 KV 问题从来不只是“存不存得下”，而是“如何组织、复用、回收和解释”。

讲解：这一讲应形成的容量直觉

核心结论

KV 课最重要的不是背术语，而是形成容量和碎片的直觉。

- 理论容量和可用容量不是一回事，碎片、预留和块粒度都会吃掉并发空间。
- 分页式管理的价值，不只是节省显存，更是把状态组织变成可持续追加与回收的形式。
- 一旦进入前缀共享，KV 又从“私有状态”进一步变成“可能跨请求复用的共享状态”。

例子：算一遍 KV 容量为什么能立刻改变系统判断（1）

核心结论

KV 课程里最应该让学生亲手做的，是容量算术，因为它直接决定你如何理解并发上限。

输入更长

单请求 KV 规模线性增长，prefill 结束后就把一大块状态留在系统里。

例子：算一遍 KV 容量为什么能立刻改变系统判断（2）

核心结论

KV 课程里最应该让学生亲手做的，是容量算术，因为它直接决定你如何理解并发上限。

并发更高

多请求同时驻留时，容量压力不是加法直觉那么简单，还会叠加块管理与碎片问题。

例子：算一遍 KV 容量为什么能立刻改变系统判断（3）

核心结论

KV 课程里最应该让学生亲手做的，是容量算术，因为它直接决定你如何理解并发上限。

课堂结论

只要状态规模算错，后续对调度、throughput 和 tail latency 的解释就会整体偏掉。

例子：为什么连续大块分配很快会失败（1）

核心结论

传统“给每个请求预留一整段连续显存”的想法，在动态服务下几乎注定碰壁。

预留过大

会浪费大量空闲容量，使理论并发远高于实际并发。

例子：为什么连续大块分配很快会失败（2）

核心结论

传统“给每个请求预留一整段连续显存”的想法，在动态服务下几乎注定碰壁。

预留过小

一旦上下文稍长，请求就会频繁扩容甚至失败。

例子：为什么连续大块分配很快会失败（3）

核心结论

传统“给每个请求预留一整段连续显存”的想法，在动态服务下几乎注定碰壁。

PagedAttention 的意义

把逻辑连续变成物理分块后，系统终于可以更诚实地管理状态生命周期。

代码例子：BlockManager 里最值得学生先看的对象（1）

核心结论

KV 讲义里的代码例子，应该从对象关系入手，而不是从论文术语入手。

核心对象

```
Block(block_id, ref_count, hash_value, token_ids)
free_block_ids
used_block_ids
hash_to_block_id
```

代码例子：BlockManager 里最值得学生先看的对象（2）

核心结论

KV 讲义里的代码例子，应该从对象关系入手，而不是从论文术语入手。

课堂应追问

这些对象分别体现了分配、占用、共享和回收的哪一面？

代码例子：BlockManager 里最值得学生先看的对象（3）

核心结论

KV 讲义里的代码例子，应该从对象关系入手，而不是从论文术语入手。

系统含义

读懂这些对象，才会知道 KV 管理讲的并不是“抽象缓存”，而是明确的数据结构与状态机。

代码例子：从 can_allocate 到 allocate 的判断链（1）

核心结论

课堂最适合拿来解释的，不是整份代码，而是“能否分配 -> 如何分配”的判断链。

代码片段

```
num_required_blocks = seq.num_blocks - num_computed_blocks
return num_required_blocks <= len(self.free_block_ids)
seq.block_table.append(block_id)
```

代码例子：从 can_allocate 到 allocate 的判断链（2）

核心结论

课堂最适合拿来解释的，不是整份代码，而是“能否分配 -> 如何分配”的判断链。

课堂应追问

为什么“还剩多少 free blocks”本身就已经在决定系统能否扩大并发？

代码例子：从 can_allocate 到 allocate 的判断链（3）

核心结论

课堂最适合拿来解释的，不是整份代码，而是“能否分配 -> 如何分配”的判断链。

结论

KV 不是调度的附属品，容量判断本身就在塑造调度器的可选动作空间。

例子：前缀复用为什么比普通缓存更复杂（1）

核心结论

一旦系统尝试前缀共享，KV 就从“每请求私有状态”变成“可能跨请求共享状态”。

收益直觉

相同前缀意味着重复 prefill 可被避免，潜在节省显著。

例子：前缀复用为什么比普通缓存更复杂（2）

核心结论

一旦系统尝试前缀共享，KV 就从“每请求私有状态”变成“可能跨请求共享状态”。

复杂性来源

共享需要元数据索引、引用计数和正确回收边界，否则收益可能变成一致性问题。

例子：前缀复用为什么比普通缓存更复杂（3）

核心结论

一旦系统尝试前缀共享，KV 就从“每请求私有状态”变成“可能跨请求共享状态”。

课程边界

课堂里可以先讲 observe-only 的潜在命中，再逐步进入真实复用机制。

课程研究案例（内部实验）：KV-Delta

核心结论

KV 迁移优化的第一层收益是减少重复状态传输，但系统最终仍要看用户可见的端到端路径。

- repeated movement 中，delta handoff 只发送新增或变化的 KV 状态，full-image handoff 每次发送完整状态。
- 课程中先比较传输字节、consumer install 时间与端到端时延，再判断优化是否成立。
- TTFT 或 disruption 仍可能变差，说明瓶颈可能转移到安装、预取交接或调度等待。

说明：本页为课程研究案例或教学假设，不代表已发表论文结论；只有在实验包给出模型、硬件、工作负载和原始记录时，数字才可作为结果引用。

课程研究案例（内部实验）：Segment Reuse

核心结论

前缀/片段复用必须先通过 correctness gate，**再谈性能收益**。

- raw shared workload **能通过正确性检查，但因为** block boundary **不满足而没有真实** stitch commit。
- aligned workload **能触发** `stitch_committed=1`，**并通过输出等价检查**。
- **但它仍输给** native prefix baseline：**机制可行不等于性能已经成立**。

说明：本页为课程研究案例或教学假设，不代表已发表论文结论；只有在实验包给出模型、硬件、工作负载和原始记录时，数字才可作为结果引用。

课程研究案例（内部实验）：Tiered KV Cache

核心结论

多级 KV 管理中，最危险的不是结果为负，而是把没有真实 movement 的 run 写成优化收益。

- live spill / restore 可能引入额外延迟，no-movement control 是必要对照。
- restore 可能因为 `pressure_floor_reject` 或 `prefix_chain_incomplete` 被拒绝。
- movement log guard 的教学意义：先证明机制真的发生，再解释性能结果。

说明：本页为课程研究案例或教学假设，不代表已发表论文结论；只有在实验包给出模型、硬件、工作负载和原始记录时，数字才可作为结果引用。

课堂练习：把 KV 问题翻译成系统因果链（1）

核心结论

KV 这一讲不应该只留下术语，而应该留下至少一条学生自己能复述的因果链。

练习一

用一句因果链解释：为什么长上下文会通过状态驻留拖慢短请求体验？

课堂练习：把 KV 问题翻译成系统因果链（2）

核心结论

KV 这一讲不应该只留下术语，而应该留下至少一条学生自己能复述的因果链。

练习二

说明为什么分页/块管理的系统意义不只是“节省显存”。

课堂练习：把 KV 问题翻译成系统因果链（3）

核心结论

KV 这一讲不应该只留下术语，而应该留下至少一条学生自己能复述的因果链。

练习三

如果你要为 BlockManager 加第一条观测日志，你会放在 can_allocate、allocate 还是 deallocate？为什么？

课堂讨论

核心结论

KV 管理不仅涉及显存节约，更直接关系到系统可扩展性。

- 为什么 KV 不应被视为 attention 的临时中间量？
- 为什么分块管理可以扩大系统控制空间？
- 当前缀复用进入系统后，正确性与收益之间有什么张力？

本讲总结

核心结论

理解 KV 管理机制，是分析 vLLM 一类系统如何将调度收益扩展到更高并发的前提。

- KV 是容量问题，也是调度问题。
- 后续状态统一视角会继续扩展这一点。

本讲参考资料

- **vLLM / PagedAttention**（SOSP 2023） 原文链接
- **vAttention**（arXiv:2405.04437） 原文链接
- **Rethinking KV Cache Compression**（MLSys 2025） 原文链接
- **Mooncake**（FAST 2025） 原文链接

课件中的论文数字均指作者在特定模型、硬件和工作负载下报告的结果，不代表无条件性能。

对应 Tutorial

核心结论

本讲对应 Tutorial 4。先独立作答，再使用对应 Tutorial 的教师讨论要点核对。

- 文件：`build/tutorials/Tutorial_04_KV_Cache 与状态组织.pdf`
- 动手运行、日志记录与提交要求统一查看 `build/experiments/` 和对应 experiment 源目录下的 `assignment_spring-2027/`。