大模型推理基础设施

第 4 讲请求调度与 continuous batching

张书豪

华中科技大学计算机学院

研究生课程课件

教学目标

核心结论

本讲讨论调度粒度变化对系统控制能力与服务边界的直接影响。

- 理解 request-level 和 iteration-level 调度差异。
- 分析长短请求混部的队头阻塞。
- 理解 continuous batching 的系统意义和边界。
- 把调度问题上升为 SLO 驱动的控制环问题。

为什么推理系统首先是调度问题

核心结论

在线推理服务的核心矛盾在于如何将动态到达请求持续组织为稳定服务。

- 请求长度分布高度不均匀。
- 新请求持续到达，旧请求持续占用状态。
- 用户感知的卡顿往往首先来自排队和插队失败，而不是单轮计算过慢。

调度对象是什么

核心结论

调度并不天然对应“请求”，系统可以调度 request、iteration、token 或更高层服务对象。

- request-level：实现简单，但灵活性较低。
- iteration-level：控制点更细，更适合在线服务。
- token-level：理论上更灵活，但控制开销更大。
- 更细粒度往往带来更高的元数据管理与状态同步开销。

一个最小例子：为什么会发生队头阻塞

核心结论

长短请求混部时，长请求不只占用更多计算时间，还会延迟新请求被重新组织进 batch 的时机。

- 假设 batch 中有一个 8k token 长请求和多个短请求。
- 如果系统只能按 request 级别完成后再重组，新请求会被整段等待放大。
- 结果是平均吞吐可能还不错，但 TTFT 和 P99 迅速恶化。

问题本质：共享执行收益与等待代价的冲突

核心结论

batching 带来的共享执行收益，并不自动等于服务质量收益，因为等待代价会随调度控制点的粗糙程度被放大。

- 更大的 batch 通常带来更高设备利用率。
- 但更长的重组周期会直接伤害新请求首 token 时延。
- 调度问题的关键因此变成：如何在共享执行和时延控制之间找到可持续平衡。

Orca 的问题设定

核心结论

Orca 的核心问题是在动态到达条件下维持在线队列的持续可重组性。

- 传统 request-level serving 难以在执行中途插入新请求。
- 结果是长请求拖住短请求，队头阻塞显著。
- Orca 用 iteration 级重调度改变了系统控制点。

Orca 的核心机制

核心结论

Orca 的机制核心是：每完成一轮生成，就重新决定谁进入下一轮 batch。

- 调度粒度从 request 下沉到 iteration。
- selective batching 保留共享执行收益。
- 新请求更容易在轮次边界插入，旧请求不再长期独占路径。

Orca 的价值与边界

核心结论

Orca 证明了调度控制点本身可以成为论文贡献，但它并没有自动解决状态与内存的扩展性问题。

- 价值：显式缓解在线服务的队头阻塞。
- 边界：更细粒度调度需要配套状态保留与内存组织机制。
- 启发：调度收益必须与底层状态管理能力结合看待。

vLLM 为什么进一步推进了这条线

核心结论

vLLM 的重要性在于，它没有把调度和内存管理当作两个独立问题。

- continuous batching 维持高频率的请求进入与退出。
- PagedAttention 使动态请求组织不被大块连续显存要求卡住。
- 系统因此同时解决“怎么调度”和“怎么放状态”。

continuous batching 的含义

核心结论

continuous batching 的关键在于请求能够持续加入和离开执行批次。

- 请求完成后可以尽快离开，释放状态与算力。
- 新请求可以更快补入，而不必等待整批结束。
- 调度器因此获得更高频率的控制机会。

调度和 KV 管理为什么必须联动

核心结论

如果状态管理方式不支持动态批处理，调度器理论上的灵活性很快就会被容量和碎片问题吃掉。

- 更频繁的请求进出意味着更频繁的状态分配和回收。
- 如果 KV 需要连续大块空间，dynamic batching 很快遇到容量墙。
- 这就是为什么 vLLM 的调度贡献必须和 PagedAttention 一起理解。

调度作为控制环

核心结论

在线推理调度更像一个围绕 SLO 的反馈控制环，而不是静态队列算法。

- 输入：到达率、长度分布、资源利用率、KV 占用。
- 决策：准入、优先级、batch 上限、路由与配额。
- 输出：TTFT、P99、吞吐、违约率。
- 目标：在负载波动下维持稳态，而不是只在固定场景上跑出峰值数字。

源码穿插：nano-vllm-hust 的调度器主循环

核心结论

`nanovllm/engine/scheduler.py` 把 waiting、running、prefill、decode 和抢占这些调度概念落在了同一条主循环里。

- `waiting` 与 `running` 两个队列刻画了请求的基本运行状态。
- `schedule()` 先尝试 prefill，再进入 decode 调度。
- `preempt()` 与 `postprocess()` 体现了调度决策和状态更新的闭环关系。

源码穿插：几个值得追问的细节

核心结论

该实现虽然精简，但保留了几个值得追问的运行时细节。

- 为什么 chunked prefill 只允许在当前批次的第一个序列上发生。
- 为什么 decode 过程中 `can_append()` 失败会触发抢占与回退。
- 为什么 `postprocess()` 同时承担完成判定、状态回收和运行队列维护。

哪些场景会让调度问题更难

核心结论

模型和工作负载的变化，会显著提高调度问题的复杂度。

- 长上下文放大 prefill 和 KV 驻留压力。
- MoE 带来稀疏激活和跨卡负载倾斜。
- 智能体工作流带来更不稳定的请求形态和服务阶段。

讲解：调度课真正讨论的不是“排队算法”而是控制权

核心结论

在线推理里的调度，本质上是在动态负载下决定“谁先获得下一轮执行机会”。

- 新请求持续到达，旧请求持续占用状态，系统必须不断重做组织决策。
- 调度粒度越粗，等待放大越明显；调度粒度越细，元数据和状态管理压力越大。
- 因此调度课的重点不是某个名字，而是控制点设在什么地方最合适。

讲解：continuous batching 的系统意义

核心结论

continuous batching 不是一句口号，而是把系统控制节拍从“整批完成后再重组”改成“每轮都有机会重组”。

- 这会直接改善新请求插入系统的机会，从而缓解队头阻塞。
- 但它也要求底层状态分配、追加和回收能跟上更高频率的请求进出。
- 所以这讲必须和下一讲 KV 管理连起来理解，单看调度器是不够的。

例子：长短请求混部时为什么短请求先抱怨（1）

核心结论

调度问题首先体现为“谁先被服务”，而不是“总共算了多少 token”。

同质 workload

请求长度相近时，batch 内部更稳定，平均指标和个体体验差距较小。

例子：长短请求混部时为什么短请求先抱怨（2）

核心结论

调度问题首先体现为“谁先被服务”，而不是“总共算了多少 token”。

异质 workload

长请求持续占位时，短请求虽然 token 少，却可能在 waiting 队列里被拖慢。

例子：长短请求混部时为什么短请求先抱怨（3）

核心结论

调度问题首先体现为“谁先被服务”，而不是“总共算了多少 token”。

用户视角

短请求用户最先感知到的是 TTFT 和“为什么它还没开始”，这正是调度问题的第一层表象。

例子：为什么 continuous batching 会改变控制点（1）

核心结论

continuous batching 的价值不在于名字，而在于它让新请求有机会在旧请求未完成时插入系统。

静态批处理

先凑满、再执行、再整体结束，控制点比较粗。

例子：为什么 continuous batching 会改变控制点（2）

核心结论

continuous batching 的价值不在于名字，而在于它让新请求有机会在旧请求未完成时插入系统。

连续批处理

decode 过程中仍允许接纳新请求，系统控制粒度从“批次整体”转向“迭代过程”。

例子：为什么 continuous batching 会改变控制点（3）

核心结论

continuous batching 的价值不在于名字，而在于它让新请求有机会在旧请求未完成时插入系统。

代价

更细粒度控制通常伴随更复杂的状态管理、回收与公平性问题。

代码例子：调度循环里最值得先看的变量（1）

核心结论

调度课程的代码例子，应先让学生知道哪些量体现 waiting、running 和 batch 变化。

阅读线索

```
waiting
running
scheduled_seq_groups
preempt()
```

代码例子：调度循环里最值得先看的变量（2）

核心结论

调度课程的代码例子，应先让学生知道哪些量体现 waiting、running 和 batch 变化。

课堂应追问

哪个变量在表达“谁还没开始”，哪个变量在表达“谁已经占据了系统资源”？

代码例子：调度循环里最值得先看的变量（3）

核心结论

调度课程的代码例子，应先让学生知道哪些量体现 waiting、running 和 batch 变化。

系统含义

只有把状态集合读懂，后续任何策略比较才不是停留在口号上。

代码例子：最小策略改动应该改哪里（1）

核心结论

课堂项目不需要一上来重写调度器，而是要学会改最小控制点。

典型改动

```
if short_request: prioritize_enqueue()
if decode_running: limit_new_prefill()
```

代码例子：最小策略改动应该改哪里（2）

核心结论

课堂项目不需要一上来重写调度器，而是要学会改最小控制点。

课堂应追问

这类改动预期影响的是 throughput、TTFT，还是 queue wait？

代码例子：最小策略改动应该改哪里（3）

核心结论

课堂项目不需要一上来重写调度器，而是要学会改最小控制点。

反例

如果改动点不清楚，即使结果变化了，也很难说是调度策略真的在起作用。

例子：吞吐更高为什么不一定是更好的调度（1）

核心结论

调度器最容易出现的误判，就是把更多 token/s 直接当成更优系统。

情况一：长请求主导

总吞吐上去了，但短请求首 token 更慢，在线体验更差。

例子：吞吐更高为什么不一定是更好的调度（2）

核心结论

调度器最容易出现的误判，就是把更多 token/s 直接当成更优系统。

情况二：过度保守

P99 稳了，但设备空转更多，总吞吐和 goodput 都被压低。

例子：吞吐更高为什么不一定是更好的调度（3）

核心结论

调度器最容易出现的误判，就是把更多 token/s 直接当成更优系统。

判断原则

调度优劣必须放回 workload 和目标 SLO，不能脱离场景单看一个数字。

课堂练习：给调度策略写失败条件（1）

核心结论

系统课里的策略设计必须先想清“在哪种结果下我认输”。

练习一

为“短请求优先插入”写出一个明确失败条件，例如哪类长请求会被严重拖慢。

课堂练习：给调度策略写失败条件（2）

核心结论

系统课里的策略设计必须先想清“在哪种结果下我认输”。

练习二

为“decode 保守策略”写出一个明确收益条件，例如在哪种 workload 下它可能改善稳定性。

课堂练习：给调度策略写失败条件（3）

核心结论

系统课里的策略设计必须先想清“在哪种结果下我认输”。

练习三

用一句话回答：continuous batching 真正改变的系统对象是什么？

课堂讨论

核心结论

调度设计的关键，在于控制点能够在全链路范围内稳定保留收益。

- 为什么 iteration-level 调度不是简单“调得更频繁”？
- 在什么情况下更细粒度调度反而不值得？
- 如果你要设计下一代 serving scheduler，你会优先观察哪些信号？

本讲总结

核心结论

调度改变的是系统控制点，而控制点决定系统对动态负载的响应能力。

- 不理解调度粒度，就无法理解 serving runtime 的核心差异。
- Orca 说明控制点本身可以成为贡献。
- vLLM 说明调度收益必须和状态管理能力一起兑现。
- 后续 KV 与状态管理讨论都要回到这一控制视角。

对应 Tutorial

核心结论

本讲对应 Tutorial 3。先独立作答，再对照下一页参考答案。

- 文件：build/tutorials/Tutorial_03_ 调度与连续批处理观察.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-2026/。