nano-vllm-omni
自学手册

从 Qwen-Omni 模型实现，到多阶段并发推理服务

Part I　练习：26 道一小时核心实现

Part II　大作业：单卡 RTX 5090 上的 nano-vllm-omni

Part III　参考：vLLM-Omni 并发设计深度解读

源码对照：vllm-omni `fa8b4199c`（main）· vLLM v0.31.0 · transformers 4.57.1
练习代码：`nano-vllm-omni-drills/`（`mine/` 是你要写的，`tests/` 是验收测试，`solutions/` 是参考解）
文中的 `path:line` 是 vllm-omni 仓库中对应 commit 的文件和行号，可以在本地 checkout 里直接打开。

出处与最新版本：https://github.com/kzhou003/nano-vllm-omni-drills
© 2026 kzhou003　转载或改编请注明出处。

如何使用这本手册

1. **Part I（约 25 小时）。**每次坐下来做一道题，目标是一小时内把一个核心机制写对。建议先做 Track M：它让你熟悉 Qwen-Omni 每一层在算什么；然后从 orchestrator 开始做 serving 题。每题后面留了笔记区，用来记下你踩到的坑。
2. **Part II（约 8–10 周）。**把 Part I 写出来的零件组装成真实可用的服务。里程碑之间严格按顺序推进；前面写的 parity 测试会一直保留，作为后续所有优化的回归测试。
3. **Part III（随时查阅）。**每当你想知道"真实系统是怎么做的、为什么那样做"，就翻到对应章节，对照 `path:line` 去读 vllm-omni 的源码。

三部分的对应关系：Part I 的题 ⇄ Part II 的里程碑 ⇄ Part III 的章节。例如，P02/P03 的 orchestrator 对应 Part II 的 M6/M8 和 Part III 第 6 节；P12/P13 的数据面对应 M7/M8 和 Part III 第 8 节；M05 的 talker MTP 对应 Part III 第 1、9 节。

打印版对原始 HTML 做了这些调整：所有折叠内容都已展开；Part I 的提示和完整测试代码移到了 Part I 末尾的附录；链接显示为 `path:line` 文本。

© 2026 kzhou003

目录

练习：26 道一小时核心实现 5
- 怎么练习 6
- 全景图：Orchestrator → EngineCore → Scheduler → Sampler 7
- 题目总表（26 题，约 23 小时） 11
- 并发程序设计框架：T · SBWC · LIFE 13
- Track M · Qwen-Omni 模型实现 23
  - #1 · M01 RMSNorm 与多模态 RoPE（interleaved / chunked） 24
  - #2 · M02 Thinker Attention：QK-norm + GQA + KV cache 28
  - #3 · M03 Sparse MoE：路由、按 expert 分组计算、shared expert 32
  - #4 · M04 多模态拼接：placeholder 计数、embedding 合并、DeepStack 36
  - #5 · M05 Talker MTP：一帧内预测 15 个残差 codebook 40
  - #6 · M06 Code2Wav 的流式因果卷积与转置卷积 43
  - #7 · M07 按 stage 加载与打包权重 46
  - #8 · M08 串起来：tiny thinker 的 KV-cache greedy 解码 49
- Track A · 前端与 Orchestrator 52
  - #9 · P02 Orchestrator v1：线性多 stage 路由 53
  - #10 · P03 Orchestrator v2：async_chunk 预热 + 终态守门 56
  - #11 · P01 OutputSink：线程 → asyncio 的批量唤醒 59
  - #12 · P04 StagePool：粘性绑定、两阶段 abort、故障隔离 62
  - #13 · P05 RpcRouter：共享通道上的 RPC 关联 66
- Track B · Stage 进程与调度 69
  - #14 · P10 EngineCore busy loop 70
  - #15 · P11 WAITING_FOR_CHUNK：停靠与归还 73
  - #16 · P07 Scheduler：continuous batching + chunked prefill + 抢占 77
  - #17 · P06 BlockManager：paged KV + 前缀缓存 81
  - #18 · P08 批不变的采样器（per-request seed） 85
  - #19 · P09 Paged KV 写入 + decode attention（参考实现） 89
- Track C · 数据面 92
  - #20 · P12 SharedMemoryConnector：跨进程 put/get 协议 93
  - #21 · P13 ChunkAdapter：后台收发线程与顺序/清理语义 96
- Track D · 模型胶水（纯函数） 100
  - #22 · P15 Thinker→Talker 输入构造（含“差一行”） 101
  - #23 · P14 Codec 窗口规划：首块小、后块大、左上下文 104
  - #24 · P18 MRoPE 三维位置 107
- Track E · 启动与 GPU 110
  - #25 · P16 启动规划：设备分组、并行初始化、回滚、准入 111
  - #26 · P17 CUDA graph bucket runner（CPU 模拟） 114
- Capstone：把积木串成一个 mini omni 服务（半天到一天） 116
- 附录 A · 三级提示（卡住 15 分钟以上再看；L1 → L2 → L3 逐级剧透） 118
- 附录 B · 完整测试代码 171
- 附录 C · 参考解（带中文注释；易错： 标出容易写错的地方） 211
- 附录 D · P02 Orchestrator ↔ vllm-omni 源码对照 246

大作业：从零实现 nano-vllm-omni 254
- 0. 作业说明与最终交付 255
- 1. 目标模型与硬件预算 255
- 2. 目标架构总览 257
- 3. 自己写 vs 借用 清单 257
- 4. 推荐代码结构 258
- 5. 里程碑路线图 258
  - M0 · 环境与金标准（golden reference） 258
  - M1 · 权重拆分与按 stage 加载 259
  - M2 · 单 stage LLM 引擎核心（nano-vllm 级别） 260
  - M3 · 多模态输入与 MRoPE 260
  - M4 · Talker stage（单进程串联版） 261

© 2026 kzhou003

M5 · Token2Wav stage 262
M6 · 多进程拓扑与 Orchestrator 262
M7 · 数据面 Connector（绕开 orchestrator） 263
M8 · async_chunk 流式流水线 264
M9 · OpenAI 兼容服务 265
M10 · 性能：CUDA graph、IO 与计算重叠 265
M11 · 正确性与鲁棒性 266
6. 基准与验收 266
7. 评分标准（满分 100 + 加分） 267
8. 加分项（每项 +3～+8） 267
9. 常见坑合集（遇到问题再来查） 268
10. 参考资料 268
参考：vLLM-Omni 并发设计深度解读 270
0. 阅读说明 271
1. 背景：Qwen3-Omni 模型 271
2. vLLM 核心概念速成 274
3. 全景：进程 / 线程 / 事件循环 275
4. 一个请求的一生（async_chunk 模式） 276
5. 前端 ↔ Orchestrator：跨线程队列 277
6. Orchestrator 事件驱动循环 278
7. Stage 进程内部 279
8. async_chunk 数据面 281
9. GPU 层并发：CUDA stream / event / graph 284
10. 正确性：abort / 故障 / 关停 / RPC 285
11. 启动期并发 286
12. MRv2 原生数据面（简述） 286
13. 设计取舍与已知风险 287
14. 推荐源码阅读路线 287

© 2026 kzhou003

PART I

练习：26 道一小时核心实现

先拆开练：8 道 Qwen-Omni 模型实现题，对拍 HF transformers；18 道 serving 机制题，覆盖 orchestrator、调度、数据面、启动和 GPU。每题都带完整题面、接口骨架、验收要点和面试官追问；提示和测试代码放在本部分末尾的附录里。

© 2026 kzhou003

怎么练习

1. 代码在 `nano-vllm-omni-drills/`：在 `mine/` 里实现，用 `pytest tests/test_<id>.py` 验收。参考解在 `solutions/`，跑 `DRILL_IMPL=solutions pytest` 可以看到它全部通过。
2. 每题限时：先花 5 分钟读题，列出你想到的边界情况；然后 45 分钟实现；最后 15 分钟跑测试、修 bug。
3. 顺序建议：**Track M（先把模型吃透）** → A → B → C → D → E。题目之间的依赖：M08 用到你写的 M01、M02、M03、M07；P03 在你的 P02 基础上扩展；其余题目互相独立。
4. 打印版做了调整，按"剧透程度"递增排列：每题正文只有题面、骨架、验收要点、**类比**和**框架视角表**；**附录 A** 是三级提示（L1 I/O 示例与关键问题 → L2 思路与坑 → L3 详解与伪代码），**附录 B** 是完整测试，**附录 C** 是带中文注释的参考解，**附录 D** 是 P02 ↔ vllm-omni 源码对照。
5. 先读下一章**全景图**：它把 26 题放进同一张分层图（Orchestrator → EngineCore → Scheduler → Sampler，旁边是数据面），彩色题号可点击直接跳到该题。
6. 交互版（React Flow 逐步图、隐藏答案）在 `nano-vllm-omni-drills/hints/`：每题一个 `<题号>_overview.html`，P02 另有 `p02_flow.html`；入口 `hints/index.html`。正文里"交互图"一行给出对应文件。
7. 环境：Python ≥ 3.10、torch、pytest，全部可以在 CPU 上跑。Track M 还需要 `transformers==4.57.1`，测试拿它作为对拍的标准答案。

© 2026 kzhou003

全景图：Orchestrator → EngineCore → Scheduler → Sampler

> **一句话：** Orchestrator 指挥多个 stage；每个 stage 有一台自己的发动机 EngineCore；发动机每转一圈（step），先由 Scheduler 决定这一步让哪些请求上车，模型前向算出 logits，再由 Sampler 为每个请求选出下一个 token。stage 之间的大块数据不经过指挥，走旁边的数据面（connector）。

题目颜色： A 前端与 Orchestrator　B Stage 进程与调度　C 数据面　D 模型胶水　E 启动与 GPU　M 模型实现

1. 分层总图

用户 / API server

add_request(prompt) → 异步迭代 OrchOutput，直到 finished=True

▼

前端进程 · Orchestrator（一个 asyncio 事件循环，管所有 stage）

管跨 stage 的事：请求依次经过哪些 stage、何时 submit 下游、何时发出唯一的终止消息、abort 时通知所有 stage。不关心 KV cache，也不关心 batch 怎么组。

路由与终态

P02 线性多 stage 路由，单写者 dispatcher

P03 async_chunk：预热下游 stage，暂扣终止消息直到所有 stage 完成

副本与 abort

P04 StagePool：请求粘性绑定到副本、两阶段 abort、故障隔离

线程 ↔ 事件循环

P01 OutputSink：output 线程用 `call_soon_threadsafe` 合并唤醒事件循环

P05 RpcRouter：共享回包通道上，按 key 把结果交回阻塞的调用方

▼ StageClient：submit / get_output / abort / utility（ZMQ，跨进程，每个 stage 一个）

Stage 进程 · EngineCore（每个 stage 一个：thinker / talker / code2wav ...）

输入线程 → input_queue → **主线程 busy loop** → 输出线程。主线程是引擎状态的唯一写者。它不知道还有其他 stage。

busy loop

P10 空闲时阻塞等待；按 FIFO 处理完 ADD / ABORT / UTILITY / WAKEUP；有活就 step 一次并 emit 输出；fatal 和 utility 错误分开处理

▼ 每次 engine.step() 依次经过：

© 2026 kzhou003

① Scheduler：这一步算哪些请求、各算多少 token

P07 continuous batching、chunked prefill、抢占

P06 BlockManager：paged KV block 分配、前缀缓存（供 P07 调用）

P11 ChunkAwareScheduler：包在 P07 外面，下一个 chunk 没到的请求先停靠（park），到了再归还

② 模型前向：logits [B, V]

M01–M08 thinker / talker / MTP / code2wav 各层

P09 按 slot_mapping 写 KV，decode attention 按 block table 收集 KV

P18 MRoPE 三维位置

P17 decode 用 CUDA graph 分桶 replay

③ Sampler：每个请求下一个 token 是什么

P08 逐行处理：repetition penalty → temperature → top-k/top-p → multinomial；每个带 seed 的请求有私有 generator，结果与 batch 组成无关

数据面（与控制面并行，不经过 Orchestrator）

上游 stage → P13 ChunkAdapter 的后台 save 线程 → P12 ShmConnector（共享内存，按 key 存取，读即消费）→ 下游 stage 的 P13 后台 recv 线程 → chunk 就绪 → P11 归还请求，参与调度。

载荷内容怎么构造：P15 thinker hidden 到 talker prefill 的输入，P14 codec 帧窗口交给 code2wav。

启动阶段（在上面这些跑起来之前）

P16 共享 GPU 的 stage 分组，组内串行、组间并行初始化，失败回滚，显存准入检查 · M07 按 stage 加载权重 · P17 capture CUDA graph

2. 四个核心组件分别管什么

组件	在哪里跑	负责	不负责	题目	vllm-omni 对应
Orchestrator	前端进程，一个事件循环	跨 stage 路由；每个请求恰好一条 finished=True，且是最后一条；abort 扇出到所有 stage；副本选择	batch、KV、采样；async_chunk 模式下也不搬数据	P01 P02 P03 P04 P05	`vllm_omni/engine/orchestrator.py` `stage_engine_core_client.py`
EngineCore	每个 stage 一个进程；主线程跑 busy loop	接收消息，驱动 step，把输出交给输出线程；区分 fatal 与 utility 错误；有序关闭	其他 stage 的存在；具体怎么组 batch（交给 scheduler）	P10	`vllm_omni/engine/stage_engine_core_proc.py`（继承 vLLM `EngineCoreProc`）
Scheduler	EngineCore 主线程内	每一步在 token budget 内选出请求、给每个请求分配多少 token；分配 KV block；空间不够时抢占；async_chunk 下停靠还没收到数据的请求	算 logits，选 token	P07 P06 P11	`vllm_omni/core/sched/`（`omni_ar_scheduler.py`、`omni_generation_scheduler.py`）

© 2026 kzhou003

Sampler	EngineCore 主线程内，前向之后	logits [B, V] → token [B]；每个请求用自己的采样参数和随机流	哪些请求参与本步（scheduler 已经决定好了）	P08	vLLM v1 Sampler（`vllm/v1/sample/`）

> Scheduler 和 Sampler 都在做"选择"，但对象不同：Scheduler 选**哪些请求**参与这一步、每个算多少 token；Sampler 为每个参与的请求选**下一个 token**。

3. 一个请求走完全程（把题目串起来）

以 thinker(S0) → talker(S1) → code2wav(S2)、final={0,2}、async_chunk 模式为例。蓝色是控制面，青色是数据面，红色是异常路径。

1. **启动**：P16 规划各 stage 的初始化顺序并检查显存；每个 stage 进程 M07 加载自己那部分权重，P17 capture CUDA graph，之后进入 P10 的 busy loop，空闲阻塞。
2. **请求进入**：用户调用 add_request。P03 在任何 await 之前注册请求状态，并把 submitted 一次性设为 {0,1,2}；然后 S0 拿真实 prompt，S1、S2 拿占位输入（预热）。如果 stage 有多个副本，P04 先选副本并做粘性绑定。
3. **跨进程**：StageClient 通过 ZMQ 发出 ADD。stage 进程的输入线程把消息放进 input_queue，P10 主循环被唤醒，在下一次 step 之前处理掉它。
4. **调度**：P07 在 token budget 内接纳请求，做 chunked prefill；向 P06 申请 KV block，前缀命中则复用。S1、S2 此时还没收到数据，被 P11 停靠，上游 scheduler 完全看不到它们。
5. **前向**：S0 thinker 计算 M01–M04（RMSNorm/RoPE、attention、MoE、多模态合并），位置用 P18 MRoPE；P09 写入 KV 并做 decode attention；decode 阶段走 P17 图 replay。
6. **采样**：P08 为每一行选出 token。P10 把结果 emit 给输出线程，经 ZMQ 回到前端。
7. **数据面·发送**：S0 每产生一段，P13 的 save 线程用 P15 的规则（裁掉 stop 那一行）构造 talker 输入，按 key `{rid}_{stage}_{chunk_id}` 写入 P12 共享内存。
8. **数据面·接收**：S1 的 P13 recv 线程按 chunk_id 顺序取数据，放进 ready 队列 → P11 归还请求 → S1 talker 用 M05 MTP 生成 codec 帧 → 同样经 connector 发给 S2；S2 按 P14 的窗口规划（首块小、带左上下文）调用 M06 流式 code2wav，输出音频。
9. **回到前端**：各 stage 的 output 线程通过 P01 OutputSink 合并唤醒事件循环 → reader → P02/P03 的单写者 dispatcher。S0、S2 的输出发给用户；S1 不是 final stage，它的输出不发给用户，只用来记录完成状态。
10. **终止**：P03 在 all_final 时，如果 S1 还没报告完成，就把最后一条消息暂扣；等 `submitted ⊆ finished` 成立才放行唯一的 finished=True，然后 cleanup。之后迟到的输出都被丢弃。
11. **abort**：P04 两阶段：先只读收集 partial，所有 stage 的 P10 处理完 ABORT 并确认后再提交；connector 里没人读的数据段由 P12 的 cleanup_prefix、P13 的清理逻辑回收。
12. **管理调用和崩溃**：UTILITY 带 call_id 进入 P10，结果回到前端后由 P05 交回对应的调用方；utility 自身失败只影响这次调用。step 出现 CUDA 错误时，P10 发出 fatal → 前端 P05 的 `fail_all` 让所有在途调用立即失败，P04 隔离故障副本。

© 2026 kzhou003

4. 跨层反复出现的同一个模式

很多题在不同层解决的是同一类竞态。认出这些模式，后面的题会快很多。

模式	为什么需要	出现在
先登记，再让出控制权或发送	回应可能比"发送"返回得还快。如果没提前登记，回应会被当成未知或迟到而丢掉，调用方一直挂着	P02/P03 await 之前注册 ReqState、标记 submitted · P05 send 之前 register · P13 先 load_async 注册再轮询
单写者	状态只由一个 task 或线程修改，就不需要锁，也不会出现交错	P02 唯一的 dispatcher · P10 引擎状态只在主线程修改 · P04 同步方法在同一个事件循环上
阻塞方必须被显式唤醒	只改共享标志，正在阻塞的一方看不到。必须往它正在等的通道里放点东西	P01 `call_soon_threadsafe`（底层写 self-pipe 唤醒 select）· P10 关闭时塞一个 WAKEUP · P05 fail_all 往每个信箱里放异常
恰好一次	终止消息、结果交付都只能发生一次	P02/P03 每个请求一条 finished=True · P05 用 pop 保证每个信箱只交付一次 · P12 读即删除
清理之后的迟到消息：丢弃，绝不复活	如果为迟到消息重新建状态，会出现第二条终止消息或状态泄漏	P02/P03 dropped_outputs · P05 late_results · P05 unregister 按身份比较，不误删同 key 的新 waiter
致命错误锁存，快速失败	进程已经不可信时，让所有人立刻失败，而不是各自等到超时	P10 发出 fatal · P05 `_fatal` 锁存 · P16 初始化短路并回滚
与 batch 组成无关	continuous batching 下 batch 每一步都在变，结果不能随之变化	P08 逐行采样，每个请求私有 generator · P17 padding 行不影响真实行

5. 容易混淆的几个问题

Q：EngineCore 是不是比 Orchestrator 更上层，由它把 orchestrator 和 scheduler 连起来?

正好相反。Orchestrator 在最上层，通过 StageClient 远程驱动多个 EngineCore。EngineCore 是单个 stage 的执行引擎，scheduler 在 EngineCore 内部。EngineCore 不知道 orchestrator 的存在，只是收消息、step、发输出。

Q：StageClient 和 connector 都是跨进程通信，有什么区别?

StageClient（ZMQ）是**控制面**：传 ADD/ABORT/UTILITY 这些小消息和每一步的输出，连接的是 orchestrator 和 stage。connector（P12/P13）是**数据面**：直接在 stage 之间传 hidden、codec 帧这类大块数据，不经过 orchestrator。P03 和 P02 的区别就在于，数据从"orchestrator 转发"改成了"走 connector"。

Q：P11 和 P07 是什么关系?

P11 是包在 P07 外面的一层（装饰器式）。每次调度之前，它把下一个 chunk 还没到的请求从 P07 的队列里摘出来，P07 完全不需要知道 chunk 的存在。调度结束后（包括抛异常时）再把请求按原队列、原状态放回去。

Q：P03 怎么知道数据真的通过 connector 送到了?

它不知道。orchestrator 只看各 stage 自己报告的 finished。数据正确性靠 connector 的协议保证（按 key 寻址、chunk_id 连续、读即消费），在测试里通过端到端比对输出内容来证明。如果想在运行时检测，需要在协议里加上 chunk 计数比对和超时。

Q：Sampler 为什么要逐行处理，而不是对整个 batch 一次 multinomial?

一次对整个 batch 抽样时，随机数按行的位置分配。别的请求插进来或离开，某个带 seed 的请求拿到的随机数就会变，结果无法复现。真实实现会在 GPU 上向量化，但仍然为每个请求维护独立的随机流。

© 2026 kzhou003

6. 建议的做题顺序

顺序	题目	理由
1	P01 → P02 → P03	P03 直接继承 P02；P01 是 P02 输入侧"线程到事件循环"的那一段
2	P05 → P10 → P04	P10 的 UTILITY 和 fatal 正好对上 P05 的配对和 fail_all；P04 的 abort 需要用到 P10 的 ABORT 语义
3	P06 → P07 → P11	P07 调用 P06 分配 block；P11 包在 P07 外面
4	P09 → P08	step 内部的前向和采样；P09 的 slot 定义与 P06 一致
5	P12 → P13	P13 使用 connector 接口（题目里用内存实现代替 P12）
6	P15 P14 P18, P16 P17	纯函数和启动相关题，相对独立，可以穿插做
并行	M01–M08	模型本身，对应 step 中"前向"那一格；M08 用到 M01/M02/M03/M07

题目总表（26 题，约 23 小时）

#	Track	题目	分钟	难度	文件
#1	M	RMSNorm 与多模态 RoPE（interleaved / chunked）	45	中	`m01_rmsnorm_mrope`
#2	M	Thinker Attention：QK-norm + GQA + KV cache	60	中	`m02_thinker_attention`
#3	M	Sparse MoE：路由、按 expert 分组计算、shared expert	60	中	`m03_sparse_moe`
#4	M	多模态拼接：placeholder 计数、embedding 合并、DeepStack	45	中	`m04_mm_merge_deepstack`
#5	M	Talker MTP：一帧内预测 15 个残差 codebook	60	难	`m05_talker_mtp`
#6	M	Code2Wav 的流式因果卷积与转置卷积	60	难	`m06_streaming_causal_conv`
#7	M	按 stage 加载与打包权重	45	中	`m07_stage_weight_loading`
#8	M	串起来：tiny thinker 的 KV-cache greedy 解码	60	中	`m08_tiny_thinker_e2e`
#9	A	Orchestrator v1：线性多 stage 路由	60	中	`p02_orchestrator`
#10	A	Orchestrator v2：async_chunk 预热 + 终态守门	45	中	`p03_async_chunk_orchestrator`
#11	A	OutputSink：线程 → asyncio 的批量唤醒	45	中	`p01_output_sink`
#12	A	StagePool：粘性绑定、两阶段 abort、故障隔离	60	中	`p04_stage_pool_abort`
#13	A	RpcRouter：共享通道上的 RPC 关联	40	易	`p05_rpc_router`
#14	B	EngineCore busy loop	45	中	`p10_engine_core_loop`

© 2026 kzhou003

#15	B	WAITING_FOR_CHUNK：停靠与归还	45	中	`p11_chunk_aware_scheduler`
#16	B	Scheduler：continuous batching + chunked prefill + 抢占	75	难	`p07_scheduler`
#17	B	BlockManager：paged KV + 前缀缓存	75	难	`p06_block_manager`
#18	B	批不变的采样器（per-request seed）	45	中	`p08_sampler`
#19	B	Paged KV 写入 + decode attention（参考实现）	45	中	`p09_paged_attention`
#20	C	SharedMemoryConnector：跨进程 put/get 协议	60	难	`p12_shm_connector`
#21	C	ChunkAdapter：后台收发线程与顺序/清理语义	75	难	`p13_chunk_adapter`
#22	D	Thinker→Talker 输入构造（含“差一行”）	40	易	`p15_talker_inputs`
#23	D	Codec 窗口规划：首块小、后块大、左上下文	40	易	`p14_codec_windows`
#24	D	MRoPE 三维位置	40	中	`p18_mrope_positions`
#25	E	启动规划：设备分组、并行初始化、回滚、准入	45	中	`p16_init_planner`
#26	E	CUDA graph bucket runner（CPU 模拟）	40	易	`p17_graph_runner`

© 2026 kzhou003

并发程序设计框架：T · SBWC · LIFE

一套写 asyncio / 多线程并发程序时可以按顺序走的思考清单 · 全程以 `mine/p02_orchestrator.py`（多 stage Orchestrator）为例

先定场地 T，再搭骨架 S B W C，再让它活 L I F E

T · Topology 执行拓扑 代码跑在哪些进程 / 线程 / 事件循环里？边界在哪？	**S · State 账本** 哪些共享状态？谁有权写？	**B · Blocking 阻塞点** 程序会在哪里"干等"？
W · Workers 岗位 谁等什么、做什么、不准做什么？	**C · Channels 通道** 岗位之间用什么传消息？长什么样？	**L · Lifecycle 生命周期** 怎么启动？怎么干净地停下？
I · Interface 对外接口 公开 API 只做什么？	**F · Flow + Invariants** 先写必须永远成立的规则，再写逻辑	**E · Edges + Errors** 迟到、乱序、未知、异常怎么办？

T 决定**在哪里跑、能用哪些同步原语**；SBWC 决定每个执行环境里**长什么样**；LIFE 保证它**跑得起来、停得下来、不会出错**。

0. 例子：P02 Orchestrator 要做什么

一个请求要依次经过多个 stage（类比 Qwen3-Omni：thinker → talker → code2wav）。每个 stage 是一个独立进程，orchestrator 只能通过 `StageClient` 跟它交互：

```
class StageClient(Protocol):
    async def submit(self, request_id, inputs) -> None          # 把请求交给这个 stage
    async def get_output(self) -> list[StageOutput]            # 阻塞，直到这个 stage 吐出一批输出
```

Orchestrator 要做的事：

- 用户 `add_request(rid, prompt)` 后，交给 stage 0；
- stage s 做完一个请求，用 `forward_fn(s+1, prompt, stage s 的全部数据)` 构造输入，交给 stage s+1；
- 属于 `final_stage_ids` 的 stage 的输出，转成 `OrchOutput` 交给用户（`next_output()`）；
- 每个请求**恰好一条** `finished=True`，并且是最后一条；之后清理状态。

贯穿全文的具体例子：3 个 stage，`final_stage_ids = {0, 2}`，`forward_fn` 原样返回上一 stage 的数据。

stage	收到	吐出的 chunk	用户能看到吗
S0 thinker	`"hi there"`	`"hi"`, `"there"`	能（final）
S1 talker	`["hi", "there"]`	`["HI", "THERE"]`（1 个 chunk）	不能（中间站）
S2 code2wav	`[["HI", "THERE"]]`	`"wav:HI"`, `"wav:THERE"`	能（final）

用户最终依次收到 4 条 `OrchOutput`：stage 0 两条、stage 2 两条，只有最后一条 `finished=True`。

© 2026 kzhou003

第 0 步 T：先定场地——代码跑在哪里

T Topology · 执行拓扑

问自己

需要哪几个**进程**？每个进程里有哪些**线程**？哪个线程跑 **asyncio 事件循环**？哪些边界是题目或系统**已经给定**的？

规则

T 决定的是**执行环境的种类和边界**，W 决定的是**每个环境里有几个 worker**。先定 T，因为每跨一层边界，能用的通信方式和同步原语都不一样，后面的 S、B、C 都要受它约束。

怎么选

执行单位	什么时候需要	代价	本套题
进程	需要故障隔离（一个崩了不拖垮别的）；CPU 密集要绕过 GIL；每张 GPU / 每个模型独占资源	不共享内存，跨进程数据要序列化或走共享内存；启动慢	stage 进程；P12 / P13 跨进程数据面；P16 启动规划
线程	必须调用**阻塞的同步 API**（socket / zmq recv、文件锁、C 扩展），又不能卡住事件循环	共享内存但需要锁；有 GIL；容易出竞态	P01 生产者线程；P13 后台收发线程；P10 busy loop
asyncio task	大量 I/O 等待、在同一个事件循环里协作	只在 `await` 处切换；任何阻塞调用都会卡住整个循环	P02 / P03 / P04 / P05 的所有 worker

层级与边界

```
进程 ⊃ 线程 ⊃ 事件循环（每个线程最多一个） ⊃ task
```

跨越的边界	通信方式	同步原语
进程 ↔ 进程	pipe / zmq / 共享内存（`multiprocessing.shared_memory`）	文件锁、协议约定（谁创建、谁释放）
线程 ↔ 线程	`queue.Queue`、`collections.deque` + 锁	`threading.Lock` / `Event` / `Condition`
线程 → 事件循环	`loop.call_soon_threadsafe(...)`、`asyncio.run_coroutine_threadsafe(...)`	只能通过这两座"桥"进入循环
task ↔ task（同一循环）	`asyncio.Queue`	`asyncio.Event` / `Lock`，或者干脆用单写者不加锁

P02 怎么答

这一步在 P02 里**已经被题目定好了**：

- orchestrator 跑在 API 进程里的**一个** asyncio 事件循环上；
- 每个 stage 是独立进程，但被 `StageClient` 包成了 `async` 接口（`submit` / `get_output` 都可以直接 await），orchestrator 看不到进程边界；

© 2026 kzhou003

- 所以 P02 里**不需要任何线程**，所有 worker 都是同一个循环里的 task，可以放心用"单写者、不加锁"。

对照真实的 vllm-omni：API 进程（asyncio：orchestrator、对外 HTTP） + 每个 stage 一个进程（里面是 EngineCore busy loop，见 P10） + stage 进程里的后台收发线程（跨进程传 hidden states / codec，见 P13） + 从线程把结果送回事件循环的 OutputSink（见 P01）。每一道题其实都在练习**某一条边界**上的通信。

T 选错了会怎样：在 asyncio task 里直接调用阻塞的 `zmq.recv()` 或 `time.sleep()`，整个事件循环（包括所有其他请求）都会停住；在普通线程里直接调用 `asyncio_queue.put_nowait()`，偶尔会丢唤醒、偶尔会报错，而且很难复现。

骨架 SBWC：决定每个执行环境里长什么样

S State · 账本

问自己　有哪些**可变的共享状态**？每一份由**谁**来写？

规则　尽量让每份可变状态只有**一个写者（single writer）**。只有做不到时才上锁。

P02 怎么答

状态	内容	谁写
`request_states: dict[rid, ReqState]`	每个在途请求的进度：`submitted`、`finished`、`stage_data`	注册：request_handler；其余全部：**dispatcher**
`dropped_outputs: int`	丢弃的未知请求输出计数	dispatcher

asyncio 是单线程的，两个 task 不会"同时"执行 Python 代码；request_handler 的注册是一个没有 `await` 的 dict 赋值，不会和 dispatcher 交错。真正的风险在 `await` 处，见 F。

违反了会怎样：如果每个 reader 收到输出后都自己去改 `ReqState`，就有 N 个写者。S0 和 S2 的输出可能在 `await` 中途交错，"判断是否全部 final 都结束了"的读和写之间被别人插队，出现两条 `finished=True` 或一条都没有。

© 2026 kzhou003

B Blocking · 阻塞点

问自己　程序会在哪些地方**干等**？把所有 `await xxx()` / `.get()` 列出来。哪些是**互相独立**的？

规则　**每个独立的阻塞源，配一个专门守着它的 worker。**不要让一个 worker 轮流去等多个阻塞源。

P02 怎么答

阻塞点	什么时候返回	个数
`stages[i].get_output()`	stage i 吐出输出时，时间不可预测，可能很久都不返回	N（每个 stage 一个）
`_req_q.get()`	用户提交新请求时	1
`_ready_q.get()`	任何 stage 有输出被搬进来时	1
`_out_q.get()`（在 `next_output` 里）	有结果要给用户时	1（调用方在等）

违反了会怎样（反例）：让 dispatcher 自己轮流读：

```
while True:
    for sid, client in enumerate(self.stages):
        outs = await client.get_output()     # ← 一次只能等一个
        ...
```

时刻	dispatcher 在做什么	S0	S1
t1	`await S0.get_output()`，拿到 `"hi"`	已吐出 "hi"	还没有任务
t2	`await S1.get_output()`	**"there" 已吐出，没人读**	没有任务，**永远不会有输出**
t3…	一直卡在 S1	还在等	S1 的任务要等 S0 的 "there" 被处理后才会提交，可 "there" 没人读

结果是**死锁**：dispatcher 等 S1，S1 等 dispatcher 转发，而转发需要先读到 S0 的 "there"。

© 2026 kzhou003

W Workers · 岗位

问自己　每个 worker **等什么**？**做什么**？**不准做什么**？

规则

职责越单一越好。**搬运工只搬运，决策者只决策。**写清楚每个 worker"不准做什么"，比写"做什么"更能防 bug。

P02 怎么答

worker	个数	等什么	做什么	不准做什么
`_reader(sid)`	N	`stages[sid].get_output()`	打上 `sid`，放进 `_ready_q`	看内容、碰 `request_states`
`_request_handler`	1	`_req_q.get()`	注册 `ReqState`，submit 给 stage 0	处理 stage 输出
`_dispatcher`	1	`_ready_q.get()`	唯一的决策者：记录数据、判断 final、转发、发给用户、清理	直接等 stage（那是 reader 的活）

> **工厂比喻：**dispatcher 是经理，只有他能改账本；每部电话（stage）配一个秘书（reader）守着，响了就把内容抄成纸条丢进经理的收件箱（`_ready_q`）。经理只盯收件箱，来一张处理一张。

C Channels · 通道

问自己　worker 之间用什么传东西？每个通道里的**消息格式**是什么？谁生产、谁消费？

规则

多个生产者、一个消费者时，汇进**同一个队列（fan-in）**，消息里带上**来源标识**。消息格式要提前定下来，写在注释里。

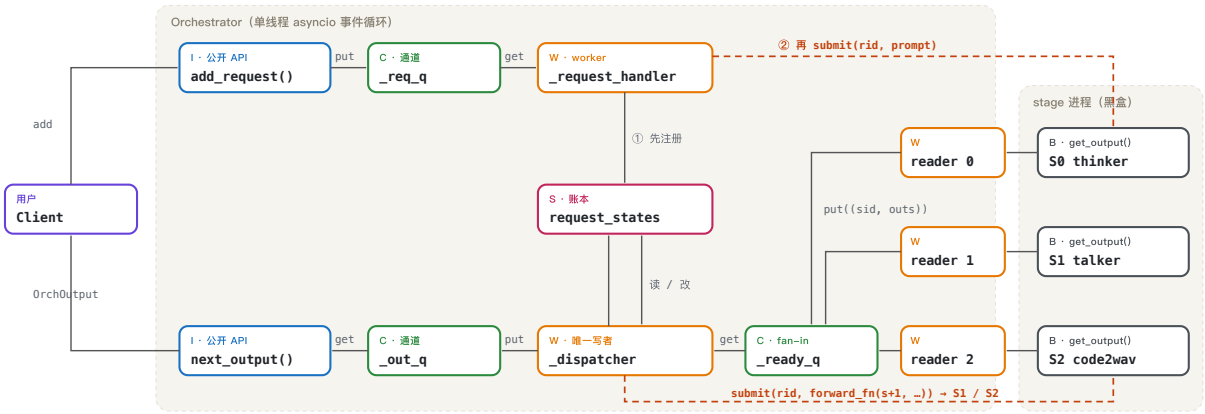
P02 怎么答

通道	生产者 → 消费者	消息格式
`_req_q`	`add_request`（调用方）→ request_handler	`("add", rid, prompt)`
`_ready_q`	N 个 reader → dispatcher（**fan-in**）	`(sid, list[StageOutput])`
`_out_q`	dispatcher → `next_output`（调用方）	`OrchOutput`

> **容易忘的：**`_ready_q` 里的消息如果不带 `sid`，dispatcher 就不知道这条输出来自哪个 stage：既没法判断它是不是 final，也没法决定转发给谁。

© 2026 kzhou003

全景图：SBWC 填完之后，系统就长这样



橙色虚线 = 向 stage 提交（submit）；灰色实线 = 队列 put / get。只有 dispatcher 和（注册时的）request_handler 会碰账本。

LIFE：让它跑起来、停得下、不出错

L Lifecycle · 生命周期

问自己 worker 什么时候创建？怎么停？停的时候会不会有 task 挂在 `await` 上？

规则

`start`：create_task 并**保存句柄**。`shutdown`：先 `cancel()`，再 `gather(..., return_exceptions=True)` 等它们真正退出。

通用写法

```
async def start(self):
    for sid, client in enumerate(self.stages):          # 每个阻塞源一个 worker (B → W)
        self._tasks.append(asyncio.create_task(self._reader(sid, client)))
    self._tasks.append(asyncio.create_task(self._request_handler()))
    self._tasks.append(asyncio.create_task(self._dispatcher()))

async def shutdown(self):
    for t in self._tasks:
        t.cancel()                                      # 在各自的 await 处抛 CancelledError
    await asyncio.gather(*self._tasks, return_exceptions=True)   # 等它们真正结束，并吞掉 CancelledError
```

常见坑： ① 不保存 task 引用，导致 task 可能被垃圾回收，shutdown 时也找不到它；② 只 cancel 不 gather，测试结束时报 "Task was destroyed but it is pending"；③ `start()` 里 await 了某个永远不返回的东西，导致 `start()` 本身卡住。

© 2026 kzhou003

I Interface · 对外接口

问自己　调用方能做什么？公开方法会不会碰内部状态？它运行在谁的上下文里？

规则　公开 API 要**薄**：只往通道里 put 或从通道里 get。所有决策留给内部 worker。

P02 怎么答

API	只做	不做
`add_request(rid, prompt)`	`_req_q.put(("add", rid, prompt))`	注册状态、submit
`next_output(timeout)`	`wait_for(_out_q.get(), timeout)`	拼结果、改状态

好处：调用方可能来自任何协程，甚至通过桥接来自别的线程。接口只投递消息，就不需要关心调用方的上下文，账本也永远只在内部被修改。

F Flow + Invariants · 核心逻辑与不变量

问自己

先写下**必须永远成立**的规则，再写处理逻辑。逐个检查每个 `await`：**如果此刻别的 worker 来跑，它看到的状态对吗?**

规则　`await` = 交出控制权。**在 await 之前，状态必须已经自洽。**

P02 的不变量

1. 每个请求**恰好一条** `OrchOutput.finished=True`，并且是该请求的**最后一条**。
2. `OrchOutput.finished` 表示"**整个请求**完成"，即 `final_stage_ids ⊆ ReqState.finished`；`StageOutput.finished` 只表示"**这个 stage**完成"。两者含义不同。
3. 只有 `final_stage_ids` 里的 stage 的输出会到用户手上。
4. stage s 结束且 `s < max(final_stage_ids)` 时，用 stage s 的**全部**数据转发给 s+1。"发给用户"和"转发"是两件独立的事，不是二选一。
5. 终态发出后，立刻删除该请求的状态。

await 切换点的经典例子：为什么必须"先注册 ReqState，再 `await stages[0].submit()`"?

顺序	时间线	结果
❌ 先 submit 后注册	`await submit()` 交出控制权 → S0 很快吐出 "hi" → reader 搬进 `_ready_q` → dispatcher 查 `request_states`：**查不到 r1** → 当成未知请求丢弃 → 回到 handler，这时才注册	用户永远收不到 "hi"，`dropped_outputs=1`
✅ 先注册后 submit	注册（无 await，不会被打断）→ `await submit()` → 之后无论谁先跑，dispatcher 都能查到 r1	正确

更多一步一步的状态变化，可以看交互时序图 `hints/p02_flow.html`。这里只讲设计，不给 `_handle_output` 的实现。

© 2026 kzhou003

E Edges + Errors · 边界与故障

问自己　消息**迟到 / 乱序 / 重复 / 对应的请求不存在**怎么办？某一步抛异常，会不会把整个 worker 弄死？

规则

worker 是长期运行的循环，**不能因为一条坏消息就退出**。凡是"不该发生但可能发生"的情况，都要有明确的处理方式，并且最好计数。

P02 怎么答

情况	处理	为什么
输出的 request_id 不在 `request_states` 里	丢弃，`dropped_outputs += 1`	抛异常会让 dispatcher 退出，**所有**请求都会卡住
请求已发出终态并被清理后，又来了它的输出	同上，按未知 id 处理	保证"终态是最后一条"
多个请求的输出在 `_ready_q` 里交错	不需要特殊处理	状态按 rid 分开存，dispatcher 串行处理
同一个 stage 的多个 chunk	按到达顺序追加进 `stage_data[sid]`	每个 stage 只有一个 reader，同一 stage 的顺序天然保持

P02 工作表（填好的）

拿到新题时，先花 5 分钟从 T 开始填这张表，再写代码。下面是 P02 的答案：

	问题	P02
T	几个进程 / 线程 / 事件循环？边界在哪？	题目已给定：orchestrator = API 进程里的 1 个事件循环；stage = 独立进程，但被 `StageClient` 包成 async 接口；不需要线程
S	共享状态有哪些？各自由谁写？	`request_states`（dispatcher 为主，handler 只负责注册）、`dropped_outputs`（dispatcher）
B	有哪些阻塞点？哪些互相独立？	N 个 `get_output()`（互相独立，时间不可预测）、`_req_q.get()`、`_ready_q.get()`、`_out_q.get()`
W	需要哪些 worker？等什么 / 做什么 / 不准做什么？	reader × N（只搬运）、request_handler（注册 + submit 给 S0）、dispatcher（唯一决策者）
C	有哪些通道？消息格式是什么？	`_req_q: ("add", rid, prompt)`；`_ready_q: (sid, list[StageOutput])`（fan-in）；`_out_q: OrchOutput`
L	怎么启动？怎么停？	`start`：N+2 个 create_task 并保存；`shutdown`：cancel + gather(return_exceptions=True)
I	公开 API 只做什么？	`add_request` 只 put；`next_output` 只 get（带 timeout）
F	不变量是什么？await 前状态是否自洽？	恰好一条终态且是最后一条；终态条件是 `final ⊆ finished`；先注册再 submit；发给用户和转发是独立的两件事；终态后清理
E	迟到 / 未知 / 重复 / 异常怎么办？	未知 id 就丢弃并计数，不抛异常；清理之后再到的消息按未知处理

© 2026 kzhou003

三条贯穿始终的法则

> **1. await 就是交出控制权。**看到 `await` 就问："如果此刻别的 worker 来跑，它看到的状态对吗？"状态的修改要在 await **之前**完成。

> **2. 跨线程只能用专门的桥。**从普通线程进入 asyncio，用 `loop.call_soon_threadsafe(...)` 或 `asyncio.run_coroutine_threadsafe(...)`，不要在线程里直接调用 `asyncio.Queue` 或 `Event` 的方法（P01 练的就是这个）。

> **3. 能用"队列 + 单写者"，就别用锁。**锁容易死锁、容易漏加；队列把并发问题变成了**串行处理**，让"谁在什么时候改了什么"变得一目了然。

常见模式速查

模式	什么时候用	在本套题里
fan-in（多读者 → 一个队列 → 单消费者）	多个独立的阻塞源，结果要由一个地方统一处理	P02 / P03 的 readers + dispatcher
single writer（单写者）	共享状态需要复杂的读-改-写	P02 dispatcher、P10 EngineCore loop
线程 → 事件循环的桥（批量唤醒）	生产者在线程里，消费者在 asyncio 里	P01 OutputSink
请求关联（key → future）	共享通道上的 RPC，结果乱序返回	P05 RpcRouter
后台收发线程 + 顺序号	跨进程数据要按顺序、可取消地传递	P13 ChunkAdapter

空白工作表（打印出来，每题填一张）

题目：_____ 日期：_____

	问题	我的答案
T	几个进程 / 线程 / 事件循环？哪些边界是给定的？每条边界用什么通信？	
S	共享状态有哪些？各自由谁写？能不能做到单写者？	
B	有哪些阻塞点？哪些互相独立？是线程阻塞还是 await 阻塞？	
W	需要哪些 worker（task / 线程）？等什么 / 做什么 / 不准做什么？	
C	有哪些通道？谁生产、谁消费？消息格式是什么？	
L	怎么启动？怎么停？停的时候有没有东西卡住？	
I	公开 API 只做什么？运行在谁的上下文（线程 / 协程）？	

© 2026 kzhou003

F	不变量有哪些？逐个检查 await：切走时状态是否自洽？	
E	迟到 / 乱序 / 重复 / 未知 / 异常 / 取消怎么办？	

写完代码后的自检：

- asyncio task 里有没有调用阻塞的同步函数？线程里有没有直接碰 asyncio 对象？
- 每个 `await` 前，状态都已经自洽了吗？
- 每份可变状态是不是只有一个写者？如果有多个，是否有锁或别的机制保护？
- 有没有 worker 同时在等两个以上的阻塞源？
- 任何一条坏消息会不会让某个 worker 退出？
- `shutdown` 之后，还有没有 task 或线程活着？

© 2026 kzhou003

Track M · Qwen-Omni 模型实现

先把模型本身吃透：每题对拍 HF transformers 4.57 的 Qwen3-Omni 实现（tiny 随机权重，CPU 秒级）。做完这一组，你会清楚 thinker / talker / MTP / code2wav 每一层在算什么。

© 2026 kzhou003

#1 · M01 RMSNorm 与多模态 RoPE（interleaved / chunked）

45 min　中　□ 完成　日期 _____

> thinker 的每一层都从 RMSNorm 开始，每个 attention 都要对 q/k 做 RoPE。Qwen-Omni 的 RoPE 是三维的 (t,h,w)，而且 Qwen3 和 Qwen2.5 的频率布局不一样——这里错一位，模型不会报错，只会悄悄变笨。

在系统中的位置

thinker（stage 0）的每个 decoder layer：`h = h + Attn(RMSNorm(h))`、`h = h + MoE(RMSNorm(h))`，Attention 里对 q、k 施加旋转位置编码。talker（stage 1）用同样的组件。你在 M02、M08 里会复用这里的函数。

RMSNorm 的数值约定

`y = w · x / sqrt(mean(x²) + eps)`。HF/vLLM 的细节：先把 x 转 fp32 做归一化，**转回输入 dtype 之后再乘 weight**。在 bf16 下顺序不同结果就不 bitwise 一致，parity 测试会报。

RoPE 回顾

head_dim = D，把向量分成两半 [a, b]，每个频率 j（0 ≤ j < D/2）对应角速度 `inv_freq[j] = θ^(−2j/D)`。位置 p 的角度为 `p·inv_freq[j]`；`cos/sin` 形状 [T, D]（两半各复制一次），旋转：`x·cos + rotate_half(x)·sin`，其中 `rotate_half([a,b]) = [−b, a]`。

MRoPE：三维位置

每个 token 有三个位置 (t, h, w)：文本 token 三者相同；图像 patch 有各自的网格坐标；音频沿时间线性。**D/2 个频率被分配给三个轴**，`mrope_section`（例如真实 Qwen3-Omni 的 `[24, 20, 20]`，和为 64 = 128/2）给出每个轴分到几个频率。分法有两种：

模型	布局（D/2 = 8，section = [4,2,2] 为例）
Qwen2-VL / Qwen2.5-Omni：**chunked**	`T T T T H H W W`
Qwen3-VL / Qwen3-Omni：**interleaved**	`T H W T H W T T`（索引 j%3==1 且 j<3·section[1] 给 H；j%3==2 且 j<3·section[2] 给 W；其余给 T）

interleaved 的动机：每个轴都能同时拿到高频和低频分量，而 chunked 让 W 轴只拿到最低频。

要实现的函数

- `rms_norm`、`rope_inv_freq`、`mrope_cos_sin(inv_freq, positions[3,T], section, interleaved)`、`rotate_half`、`apply_rotary(x[T,heads,D], cos, sin)`
- 布局约定：本套题里 q/k 用 `[T, heads, D]`（vLLM 风格，token 在前），HF 用 `[B, heads, T, D]`，测试会帮你转。

边界情况

- 纯文本时三轴位置相同，两种布局都必须退化成普通 1D RoPE（测试会检查）。
- bf16 输入时 cos/sin 要转成输入 dtype 再乘。

文件： `mine/m01_rmsnorm_mrope.py`　**验收：** `pytest tests/test_m01_rmsnorm_mrope.py`　**提示：** A-1　**测试：** B-1　**参考解：** C-1

© 2026 kzhou003

接口骨架

```
"""M01 · RMSNorm + multimodal RoPE (interleaved for Qwen3-Omni, chunked for Qwen2.5-Omni).

rms_norm(x, weight, eps): normalize in fp32, cast back to x.dtype, THEN multiply by weight.
rope_inv_freq(head_dim, theta): 1 / theta ** (arange(0, head_dim, 2) / head_dim)  (fp32).
mrope_cos_sin(inv_freq, positions[3, T], mrope_section, interleaved) -> cos, sin [T, head_dim]
  freqs[a] = positions[a, :, None] * inv_freq  for axis a in (t, h, w), each [T, head_dim // 2].
  Pick, for every frequency index j, which axis' position it rotates with:
    interleaved=True  (Qwen3):   j % 3 == 1 and j < 3*section[1] -> h;
                                 j % 3 == 2 and j < 3*section[2] -> w; otherwise t.
    interleaved=False (Qwen2.5): contiguous chunks: first section[0] indices t, next section[1]
                                 h, last section[2] w.
  Then emb = cat([f, f], -1) and return emb.cos(), emb.sin().
apply_rotary(x[T, heads, head_dim], cos, sin) = x * cos + rotate_half(x) * sin,
  rotate_half([a, b]) = [-b, a] on the two halves of the last dim.
"""
import torch


def rms_norm(x: torch.Tensor, weight: torch.Tensor, eps: float) -> torch.Tensor:
    raise NotImplementedError


def rope_inv_freq(head_dim: int, theta: float) -> torch.Tensor:
    raise NotImplementedError


def mrope_cos_sin(
    inv_freq: torch.Tensor,
    positions: torch.Tensor,
    mrope_section: list[int],
    interleaved: bool,
) -> tuple[torch.Tensor, torch.Tensor]:
    raise NotImplementedError


def rotate_half(x: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError


def apply_rotary(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ RMSNorm 在 fp32 和 bf16 下都与 HF 一致
- ✓ inv_freq 与 HF 一致
- ✓ interleaved MRoPE 的 cos/sin 与 HF Qwen3-Omni 一致（含图像网格位置）
- ✓ 纯文本位置下两种布局都退化为 1D RoPE
- ✓ apply_rotary 后的 q/k 与 HF 一致
- ✓ chunked 布局与 HF Qwen2.5-Omni 的 apply_multimodal_rotary_pos_emb 一致

面试官追问

- 为什么 vllm-omni 在 CUDA 上要给 thinker 强制 `custom_ops: ["+rotary_embedding"]`？（编译后的 MRoPE 与 QK-norm 融合改变了音频输出）
- decode 阶段只有一个新 token，cos/sin 怎么高效地增量计算?

© 2026 kzhou003

直觉与类比

> **类比 · 音量归一化 + 均衡器**
> 一个歌单里每首歌音量不同。RMSNorm 先把每首歌（每个 token 的 hidden 向量）的**整体响度**拉到同一水平（除以 RMS），再用均衡器（`weight`，每个维度一个旋钮）调每个频段。
> “先转回 x.dtype 再乘 weight”就像：先把音频**导出成 16-bit**，再过均衡器；还是先在 32-bit 工程里过均衡器再导出。耳朵听不出差别，但两份文件逐字节比较**不一样**——parity 测试就是那个逐字节比较的人。

> **类比 · 一排转速不同的钟表指针**
> 把 head_dim = D 的向量拆成 D/2 对，每一对是一根**指针**。第 j 根指针的转速是 `inv_freq[j] = θ^(−2j/D)`：j = 0 转得最快（秒针），j 越大越慢（时针、日历）。 token 在位置 p 时，第 j 根指针转到角度 `p · inv_freq[j]`。
> 两个 token 做点积时，只有**角度差**留下来——所以 attention 天然感知的是**相对位置**。快指针分辨“隔壁 vs 隔两个”，慢指针分辨“隔 10 个 vs 隔 1000 个”。旋转不改变长度，所以 RoPE 不会破坏 QK-norm 刚归一化好的范数。

> **类比 · 发牌：一人一摞 vs 轮流发**
> 把 64 张牌按“快 → 慢”排好，发给三位玩家 T、H、W。
> **chunked（Qwen2-VL / Qwen2.5-Omni）**：前 24 张给 T，接下来 20 张给 H，最后 20 张给 W。结果 W 只拿到**最慢**的那几根指针——相邻两列 patch 的角度几乎没差别，列方向的分辨率很差。
> **interleaved（Qwen3-VL / Qwen3-Omni）**：轮流发 T H W T H W …，H、W 的份额发完后剩下的牌全给 T。每个轴都同时拿到**快指针和慢指针**。

框架视角 · 数据流 / shape trace

记号：T 个 token，D = head_dim，half = D/2，heads 为 q 或 k 的 head 数。

#	tensor	shape / dtype	发生了什么
1	`x` → `rms_norm`	`[..., H]` 任意 dtype	转 fp32，按最后一维求 `mean(x²)`（keepdim），乘 `rsqrt(· + eps)`，**转回 x.dtype，再乘 weight**
2	`inv_freq`	`[half]` fp32	`θ^(−2j/D)`，j = 0 … half-1
3	`positions`	`[3, T]` int	三行依次是 t、h、w；文本 token 三行相同
4	`freqs`	`[3, T, half]` fp32	一次广播：每个轴、每个 token、每根指针的角度
5	`f`	`[T, half]`	对每一列 j，从三个轴里**选一个**拷过来（只是选列，不做新运算）；interleaved / chunked 只差在这里
6	`emb`	`[T, D]`	`cat([f, f], −1)`：第 j 列与第 j+half 列角度相同
7	`cos, sin`	`[T, D]` fp32	`emb.cos()`、`emb.sin()`
8	`apply_rotary(x)`	`x[T, heads, D]`	cos/sin 插入 head 维变成 `[T, 1, D]`、转成 x.dtype；`x·cos + rotate_half(x)·sin`

易错点

- bf16 下 RMSNorm 在 fp32 里先乘 weight 再转回：数值“差不多”，但不 bitwise 一致。
- `mean` 忘了 `keepdim=True`，或对全部维度求 mean。
- interleaved 时直接改了 `freqs[0]`（没 clone），或切片终点写成 `section` 而不是 `3 · section`。
- chunked 时 H 段取了 `freqs[1]` 的第 `0..s_h` 列，而不是同一列区间 `s_t..s_t+s_h`。
- 把每个频率相邻复制两次（`repeat_interleave`，GPT-J 风格），与“前后半配对”的 `rotate_half` 不匹配。

© 2026 kzhou003

- `rotate_half` 写成 `[b, -a]`：旋转方向反了。
- cos 直接与 `[T, heads, D]` 相乘：当 T == heads 时不会报错，但 T 维被错误地对齐到了 heads 维。

交互图：`nano-vllm-omni-drills/hints/m01_overview.html`

对照源码： `deploy/qwen3_omni_moe.yaml:84-91` `model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1694-1785` HF: Qwen3OmniMoeThinkerTextRotaryEmbedding.apply_interleaved_mrope HF: Qwen3OmniMoeThinkerTextRMSNorm

笔记 / 我踩到的坑：

© 2026 kzhou003

#2 · M02 Thinker Attention：QK-norm + GQA + KV cache ⏱ 60 min 中 □ 完成 日期 _____

> Qwen3 的 attention 比 Llama 多了一步：对每个 head 的 q、k 先做 RMSNorm 再做 RoPE。你还要让同一份实现同时支持整段 prefill、分块 prefill 和逐 token decode，并且三者结果逐行一致。

在系统中的位置

thinker 和 talker 的每一层都有一个这样的 attention。serving 时它几乎从不"整段"运行：长 prompt 被 chunked prefill 切成多块，decode 时每步只有 1 个新 token，历史 K/V 来自 cache（在 vLLM 里是 paged KV，P06/P09；这里用简单的 list 拼接）。

计算步骤（单层，新 token 数为 T）

```
q = x @ Wq^T -> [T, H, D]      k = x @ Wk^T -> [T, KVH, D]      v = x @ Wv^T -> [T, KVH, D]
q = RMSNorm_head(q; q_norm)     k = RMSNorm_head(k; k_norm)       # 对最后一维 D 做，每个 head 独立
q, k = RoPE(q, k; cos, sin)                                       # cos/sin 只对应这 T 个新 token
K, V = cat(cache.K, k), cat(cache.V, v)                           # S = past + T
score[h, t, s] = q[t, h] · K[s, h // (H/KVH)] / sqrt(D),  且 s ≤ past + t（因果）
out = softmax_fp32(score) @ V -> [T, H*D] @ Wo^T
```

为什么三种调用方式必须一致

调度器（P07）决定一个请求这一步算多少 token：可能是整段、可能是 prompt 的一部分、也可能是 1 个 decode token。模型代码不能关心这些，只能依赖"新 token 的绝对位置 = len(cache) + i"这一条约定。因果 mask 必须按**绝对位置**算，而不是按本块内的下标。

GQA

KV head 数少于 Q head 数（这里 4 vs 2）：每个 KV head 被 `H/KVH` 个连续的 Q head 共享（head h 用 kv head `h // (H/KVH)`，即 `repeat_interleave`，不是 `repeat`）。

边界情况

- cache 为 None 时就是普通 prefill。
- T=1 的 decode 和 T>1 的 chunk 用同一份代码。

文件： `mine/m02_thinker_attention.py` **验收：** `pytest tests/test_m02_thinker_attention.py` **提示：** A-2 **测试：** B-2 **参考解：** C-2

接口骨架

```
"""M02 · Qwen3 thinker attention: QK-norm + MRoPE + GQA + causal, with a KV cache.

attention_forward(w, x[T, hidden], cos[T, D], sin[T, D], cache=None) -> [T, hidden]
  1. q = x @ q_proj.T -> [T, H, D];  k, v = ... -> [T, KVH, D]
  2. RMSNorm EACH HEAD of q and k over head_dim (q_norm / k_norm weights) -- before RoPE
  3. RoPE on q and k with the given cos/sin (rows belong to the NEW tokens)
  4. if cache is given: the new tokens sit at absolute positions len(cache) .. len(cache)+T-1;
     append k, v to the cache and attend over everything cached so far
  5. causal: query at absolute position p sees keys 0..p;  GQA: query head h uses kv head
     h // (H // KVH);  scale = D ** -0.5;  softmax in fp32
  6. out [T, H*D] @ o_proj.T
Prefill-all-at-once, chunked prefill and token-by-token decode must give identical rows.
"""
from dataclasses import dataclass, field

import torch
```


© 2026 kzhou003

```python
# ------------------------------------------------------------------ given types
@dataclass
class AttnWeights:
    q_proj: torch.Tensor  # [num_heads * head_dim, hidden]
    k_proj: torch.Tensor  # [num_kv_heads * head_dim, hidden]
    v_proj: torch.Tensor  # [num_kv_heads * head_dim, hidden]
    o_proj: torch.Tensor  # [hidden, num_heads * head_dim]
    q_norm: torch.Tensor  # [head_dim]  RMSNorm weight applied per head
    k_norm: torch.Tensor  # [head_dim]
    num_heads: int
    num_kv_heads: int
    head_dim: int
    eps: float = 1e-6


@dataclass
class KVCache:
    k: list[torch.Tensor] = field(default_factory=list)  # chunks of [t, num_kv_heads, head_dim]
    v: list[torch.Tensor] = field(default_factory=list)

    def __len__(self) -> int:
        return sum(x.shape[0] for x in self.k)


def from_hf(att) -> AttnWeights:
    """Pull the weights out of a HF Qwen3OmniMoeThinkerTextAttention."""
    return AttnWeights(
        att.q_proj.weight.detach(), att.k_proj.weight.detach(), att.v_proj.weight.detach(),
        att.o_proj.weight.detach(), att.q_norm.weight.detach(), att.k_norm.weight.detach(),
        att.config.num_attention_heads, att.config.num_key_value_heads, att.head_dim, att.config.rms_norm_eps,
    )


# ------------------------------------------------------------------ implement below
def attention_forward(
    w: AttnWeights,
    x: torch.Tensor,
    cos: torch.Tensor,
    sin: torch.Tensor,
    cache: KVCache | None = None,
) -> torch.Tensor:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 整段 prefill 与 HF attention 一致
- ✓ 先 prefill 5 个 token 再逐个 decode 7 次，结果逐行等于整段
- ✓ 分块 prefill（4 + 7）等于整段，cache 长度正确

面试官追问

- 把 list 拼接换成 P06 的 paged KV + P09 的 paged attention，接口要怎么变?
- QK-norm 和 RoPE 能不能融合成一个 kernel？vllm-omni 为什么曾经因为编译融合出问题?

© 2026 kzhou003

直觉与类比

> **类比 · 迟到的人读会议纪要**
> 开会时每个人发言都会记进**纪要**（KV cache）。迟到的人（新 token）不需要让大家把会重开一遍，只要读纪要就知道之前说了什么。
> 纪要上记的是**当时**每句话的时间戳——对应“cache 里的 k 已经用当时的绝对位置旋转过了”，以后永远不用重算。
> 新来的人只能看**他到场之前（含他自己）**的发言，不能看后面的人还没说的话（因果 mask）。 关键是他要用**会议的总时钟**判断“之前”，而不是“我今天是第 0 个到的”——如果用块内下标，decode 的那个 token 会以为自己在位置 0，只能看到第一句话。

> **类比 · GQA：同桌合用一本参考书**
> 每个 Q head 是一个学生，每个 KV head 是一本参考书。书比学生少（这里 H = 4、KVH = 2）：**相邻座位**的 H/KVH 个学生合用一本——学生 0、1 用书 0，学生 2、3 用书 1。 这就是 `h // (H // KVH)`，等价于把书 `repeat_interleave`。如果误用 `repeat`，就变成“隔一个座位共用一本”（0、2 用书 0），所有 head 都拿错了书——而在 H == KVH 的测试里你根本发现不了。

> **类比 · QK-norm：比赛前统一称重**
> Qwen3 比 Llama 多一步：在做 RoPE 之前，先对每个 head 的 q、k 各自做 RMSNorm（对 head_dim）。就像比赛前统一称重：每个 head 的 q、k 先被拉到同一个“量级”，点积就不会因为某些 head 的激活特别大而爆掉。 顺序是 **proj → QK-norm → RoPE**：RMS 对旋转不变，但逐维的 `q_norm` 权重不是，顺序反了数值就不同。v 不做 norm。

框架视角 · 数据流 / shape trace

记号：新 token 数 T，past = 调用前的 `len(cache)`，S = past + T，H / KVH / D 为 head 数与 head_dim。

#	tensor	shape	发生了什么
1	`x`	`[T, hidden]`	只有新 token
2	`q` / `k, v`	`[T, H, D]` / `[T, KVH, D]`	`x @ proj.T` 后 view；q_proj 的行按 head 分块，所以最后一维连续 D 个数属于同一个 head
3	QK-norm	同上	对最后一维 D 做 RMSNorm，所有 head 共享 `q_norm`（k 用 `k_norm`）；v 不动
4	RoPE	同上	cos/sin `[T, D]` → `[T, 1, D]` 广播到每个 head
5	cache	`K, V: [S, KVH, D]`	先读 past，再追加本次 k、v，再拼接全部块
6	GQA	`[S, H, D]`	每个 KV head 复制给相邻的 H/KVH 个 Q head
7	`scores`	`[H, T, S]` fp32	$q \cdot K \cdot D^{-0.5}$；`k_pos > past + t` 的位置置 `-inf`
8	`probs @ V`	`[T, H, D]`	softmax 在 key 维（fp32），加权求和后转回 x.dtype
9	输出	`[T, H·D] @ o_proj.T → [T, hidden]`	reshape 与第 2 步的 view 对称

易错点

- 在追加之后才读 `len(cache)`：past 多算了 T，mask 整体右移，token 能“偷看”同一块里后面的 token。
- mask 用块内下标 `arange(T)` 而不是 `past + arange(T)`：decode 的 token 只能看见 key 0。
- GQA 用 `repeat` / `expand` + 错误 reshape 得到 0,1,0,1 映射；H == KVH 时看不出来。
- QK-norm 放在 RoPE 之后，或对整个 `H·D` 做 norm 而不是对每个 head 的 D。

© 2026 kzhou003

- softmax 在 `dim=-2` 上做（每列和为 1）；或在 bf16 里做 softmax。
- 对 cache 里的旧 k 重新做 RoPE（它们早已用正确的绝对位置旋转过）。

交互图：`nano-vllm-omni-drills/hints/m02_overview.html`

对照源码： `model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:588-596` HF: Qwen3OmniMoeThinkerTextAttention.forward

笔记 / 我踩到的坑：

© 2026 kzhou003

#3 · M03 Sparse MoE：路由、按 expert 分组计算、shared expert

60 min　中

□ 完成　日期 _____

> Qwen3-Omni-30B-A3B 的 “A3B” 就是 MoE：每个 token 只激活少数 expert。thinker 是纯路由 MoE，talker 还多了一个带 sigmoid 门控的 shared expert。

在系统中的位置

thinker 和 talker 的 FFN 都是 MoE。serving 时 MoE 的效率决定了吞吐：同一个 expert 的所有 token 应该一次矩阵乘完成（vLLM 的 FusedMoE 就是做这件事），而不是逐 token 循环。

路由（必须和 HF 一致）

```
logits = x @ router^T                       # [N, E]
probs  = softmax(logits, dim=-1, fp32)       # 在全部 E 个 expert 上 softmax
w, ids = topk(probs, k)                     # 先 softmax 再 top-k（不是 top-k 后 softmax）
if norm_topk_prob: w = w / w.sum(-1)         # Qwen3 默认重新归一化
out[n] = Σ_j w[n,j] · Expert_{ids[n,j]}(x[n]),   Expert(h) = (silu(h Wg^T) ⊙ (h Wu^T)) Wd^T
```

“serving 的写法”：按 expert 分组

1. 把 N×k 个 (token, slot) 对按 expert id 排序（`argsort(stable=True)`）；
2. `bincount` 得到每个 expert 的 token 数，跳过 0；
3. 每个 expert 对自己那批 token 做一次 SwiGLU；
4. 乘上对应权重后用 `index_add_` scatter 回原 token。

这正是 FusedMoE kernel 里 “moe_align_block_size + grouped GEMM” 的 CPU 版。

talker 的 shared expert

`out += sigmoid(x @ g^T) · SharedMLP(x)`：每个 token 都会经过这个 expert，门控是标量。

边界情况

- N=1（decode）时只有 k 个 expert 有活干。
- 批不变性：单独算一个 token 与在 batch 里算结果一致。

文件： `mine/m03_sparse_moe.py`　**验收：** `pytest tests/test_m03_sparse_moe.py`　**提示：** A-3　**测试：** B-3　**参考解：** C-3

接口骨架

```
"""M03 · Sparse MoE block (thinker) and MoE + gated shared expert (talker).

route(x[N, H], w) -> (weights [N, k] in x.dtype, expert ids [N, k])
  softmax(x @ router.T) over ALL experts in fp32 -> top-k -> renormalize to sum 1 if
  norm_topk_prob.
moe_forward(x[N, H], w) -> [N, H]
  out[n] = sum_j weight[n, j] * Expert_{id[n, j]}(x[n]),  Expert(h) = (silu(h Wg^T) * (h Wu^T)) Wd^T
  Do it the serving way: group (token, slot) pairs by expert, run each expert ONCE on all of
  its tokens, scatter-add the weighted results; skip experts that received no token.
  If w.shared is set (talker): out += sigmoid(x @ shared.gate.T) * SharedExpert(x).
"""
from dataclasses import dataclass

import torch
import torch.nn.functional as F
```


© 2026 kzhou003

```python
# ------------------------------------------------------------------ given types
@dataclass
class SharedExpert:
    gate_proj: torch.Tensor   # [I_s, H]
    up_proj: torch.Tensor     # [I_s, H]
    down_proj: torch.Tensor   # [H, I_s]
    gate: torch.Tensor        # [1, H]   sigmoid gate of the shared expert


@dataclass
class MoeWeights:
    router: torch.Tensor      # [E, H]
    w_gate: torch.Tensor      # [E, I, H]   stacked expert gate_proj
    w_up: torch.Tensor        # [E, I, H]
    w_down: torch.Tensor      # [E, H, I]
    top_k: int
    norm_topk_prob: bool
    shared: SharedExpert | None = None


def from_hf(block) -> MoeWeights:
    """HF Qwen3OmniMoe{Thinker,Talker}TextSparseMoeBlock -> stacked tensors."""
    ex = block.experts
    shared = None
    if hasattr(block, "shared_expert"):
        s = block.shared_expert
        shared = SharedExpert(s.gate_proj.weight.detach(), s.up_proj.weight.detach(),
                              s.down_proj.weight.detach(), block.shared_expert_gate.weight.detach())
    return MoeWeights(
        block.gate.weight.detach(),
        torch.stack([e.gate_proj.weight.detach() for e in ex]),
        torch.stack([e.up_proj.weight.detach() for e in ex]),
        torch.stack([e.down_proj.weight.detach() for e in ex]),
        block.top_k, block.norm_topk_prob, shared,
    )


# ------------------------------------------------------------------ implement below
def route(x: torch.Tensor, w: MoeWeights) -> tuple[torch.Tensor, torch.Tensor]:
    raise NotImplementedError


def moe_forward(x: torch.Tensor, w: MoeWeights) -> torch.Tensor:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 路由（softmax→top-k→renorm）与参考完全一致
- ✓ thinker MoE 与 HF 一致
- ✓ talker MoE（含 shared expert）与 HF 一致
- ✓ norm_topk_prob=False 的变体
- ✓ 单 token 与 batch 结果一致

面试官追问

- Expert Parallel 时 token 要跨卡 all-to-all，路由结果怎么用?
- 为什么 vllm-omni 的 deploy 里给 Qwen3-Omni 指定 `moe_backend: triton`？

© 2026 kzhou003

直觉与类比

> **类比 · 医院分诊台**
>
> router 是**分诊台**：给每位病人（token）在所有科室（expert）上打一个"该去这里的可能性"，先在**全部科室**上归一（softmax），再挑分最高的 k 个科室挂号（top-k）。`norm_topk_prob` 表示"只在挂了号的这几个科室之间重新分配 100% 的话语权"。
>
> **serving 的写法**：每个科室把今天分到它的病人**集中起来一起看**（一次矩阵乘），而不是医生挨个病房跑；今天没人挂号的科室直接**休息**（跳过）。
>
> 最后每位病人拿到 k 份诊断书，按话语权**加权合并**进自己的病历——必须是**累加**，第二位医生的诊断书不能把第一位的覆盖掉（`index_add_`，而不是赋值）。

> **类比 · shared expert = 家庭医生**
>
> talker 的每位病人除了专科，还**都会**见一次家庭医生（shared expert）。家庭医生的意见占多大分量，由一个只看病人本身的小开关决定：`sigmoid(x @ gate.T)`，每个 token 一个 0~1 的标量。

框架视角 · 数据流 / shape trace

记号：N 个 token，H = hidden，E 个 expert，I = expert 中间维，k = top_k，n_e = 分给 expert e 的 token 数。

#	tensor	shape	发生了什么
1	`logits`	`[N, E]`	`x @ router.T`
2	`probs`	`[N, E]` fp32	在全部 E 个 expert 上 softmax
3	`weights, ids`	`[N, k]`	top-k；可选 renorm（fp32），weights 转回 x.dtype
4	展平	`[N·k]` × 3	每个 (token, slot) 对：expert id、权重、token id（三者一一对应，row-major）
5	分组	`order[N·k]`，`counts[E]`	按 expert id 稳定排序；每个 expert 的对数（和 = N·k）
6	expert e	`x[toks]: [n_e, H] → [n_e, I] → [n_e, H]`	gather 这批 token，做一次 SwiGLU：`(silu(h Wg^T) ⊙ h Wu^T) Wd^T`；n_e = 0 跳过
7	scatter	`out[N, H]`	乘上对应权重后按 token id **累加**
8	shared（talker）	`[N, 1] · [N, H]`	`sigmoid(x @ gate.T)` 乘全量 SwiGLU，加到 out

易错点

- 先 top-k logits 再 softmax：`norm_topk_prob=True` 时恰好等价，`False` 时完全不同（权重和应 < 1）。
- token id 用 `arange(N).repeat(k)`（0,1,2,0,1,2）而不是逐个重复 k 次（0,0,1,1,2,2），和 `ids.reshape(-1)` 对不上。
- scatter 用赋值 `out[toks] = y`：同一 token 被第二个 expert 覆盖。
- gate/up 投影写成 `h @ w_gate[e]`（少了转置），形状 `[n_e, H] @ [I, H]` 对不上。
- 路由权重留在 fp32，结果 dtype 被提升；或对空 expert 做 matmul。
- shared expert 的门控当成向量，或忘了 `w.shared is None`（thinker）时跳过。

交互图：`nano-vllm-omni-drills/hints/m03_overview.html`

对照源码： `model_executor/models/qwen3_omni/qwen3_omni_moe_talker.py:356-393` `deploy/qwen3_omni_moe.yaml:28-31` HF: Qwen3OmniMoeThinkerTextSparseMoeBlock HF: Qwen3OmniMoeTalkerTextSparseMoeBlock

© 2026 kzhou003

笔记 / 我踩到的坑：

© 2026 kzhou003

#4 · M04　多模态拼接：placeholder 计数、embedding 合并、DeepStack　45 min　中

□ 完成　日期 _____

> **用户发来一段音频和一张图。前端必须提前知道要插多少个占位 token，thinker 要把编码器输出塞进这些位置，视觉特征还要通过 DeepStack 再注入前几层。**

在系统中的位置

前端（API 进程）用 HF processor 把 `<audio>`、`<image>` 展开成 N 个占位 token——**N 必须和编码器真实输出的 token 数完全一致**，否则 thinker 侧拼接失败。thinker（stage 0）里：`inputs_embeds = embed_tokens(ids)`，再把音频编码器（AuT）与 ViT 的输出按顺序写进占位位置。

音频 token 数

AuT 以 100 帧 mel 为一个窗口，每个完整窗口经过 3 次 stride-2 卷积得到 13 个 token；余下的 r = n % 100 帧同样经过 3 次 stride-2 卷积：

```
f = (r - 1) // 2 + 1
tokens_r = ((f - 1) // 2 + 1 - 1) // 2 + 1
total = tokens_r + 13 * (n // 100)        # 例: n = 100 -> 13; n = 250 -> 33
```

Embedding 合并

对每个模态，`mask = (input_ids == placeholder_id)`，`out[mask] = mm_embeds[modality]`（按出现顺序）。数量不一致必须报错——在线上这类错误通常意味着前后端 processor 版本不一致。

DeepStack

Qwen3-Omni 的 ViT 会从若干中间层（`deepstack_visual_indexes`）各导出一份视觉特征。LLM 前 L 层中，**第 i 层输出之后**，在视觉 token 的位置上把第 i 份特征**加**到 hidden 上：

```
for i, layer in enumerate(layers):
    h = layer(h)
    if i < len(deepstack): h[visual_mask] += deepstack[i]
```

边界情况

- 没有某个模态时，mm_embeds 可以缺省该 key；但若存在占位符就必须报错。
- 不要原地修改 text_embeds（调用方可能复用）。

文件：`mine/m04_mm_merge_deepstack.py`　**验收：**`pytest tests/test_m04_mm_merge_deepstack.py`　**提示：**A-4　**测试：**B-4　**参考解：**C-4

接口骨架

```
"""M04 · Placeholder bookkeeping, multimodal embedding merge, DeepStack injection.

audio_token_count(num_mel_frames) (int or LongTensor): number of <audio> placeholders the
  Qwen3-Omni audio encoder produces. Mel frames are processed in windows of 100 frames; a
  full window yields 13 tokens; the remainder r = n % 100 passes three stride-2 convolutions:
  f = (r - 1) // 2 + 1;  tokens_r = ((f - 1) // 2 + 1 - 1) // 2 + 1.  total = tokens_r + 13 * (n // 100)
merge_multimodal(input_ids[T], text_embeds[T, H], placeholder_ids, mm_embeds)
  -> (inputs_embeds[T, H], masks{modality: bool[T]})
  Replace the rows at each modality's placeholder positions, in order, with that modality's
  encoder outputs. Count mismatch (or placeholders without embeddings) -> ValueError.
  Never mutate text_embeds.
apply_deepstack(hidden[T, H], visual_mask[T], visual_embeds[n_vis, H]) -> new tensor with
```


© 2026 kzhou003

```
  visual_embeds ADDED at the visual rows.
run_layers(layers, hidden, visual_mask, deepstack_embeds): run the decoder layers in order;
  DeepStack level i is added right AFTER layer i (only for i < len(deepstack_embeds)).
"""
from collections.abc import Callable

import torch


def audio_token_count(num_mel_frames):
    raise NotImplementedError


def merge_multimodal(
    input_ids: torch.Tensor,
    text_embeds: torch.Tensor,
    placeholder_ids: dict[str, int],
    mm_embeds: dict[str, torch.Tensor],
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
    raise NotImplementedError


def apply_deepstack(hidden: torch.Tensor, visual_mask: torch.Tensor, visual_embeds: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError


def run_layers(
    layers: list[Callable[[torch.Tensor], torch.Tensor]],
    hidden: torch.Tensor,
    visual_mask: torch.Tensor,
    deepstack_embeds: list[torch.Tensor],
) -> torch.Tensor:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 音频 token 数与 HF `_get_feat_extract_output_lengths` 一致（含 100 的边界）
- ✓ 图像/音频 embedding 按序落位，文本行不变，输入不被原地修改
- ✓ 数量不一致或缺少 embedding 时报错
- ✓ DeepStack 后的整模型输出与 HF Qwen3OmniMoeThinkerTextModel 一致
- ✓ DeepStack 只影响视觉行

面试官追问

- chunked prefill 时一张图被切在两个 chunk 之间，编码器输出与 DeepStack 特征怎么分发?
- vllm-omni 的 encoder cache 以什么为 key?

直觉与类比

> **类比 · 用包占座**
> 前端是提前进场的朋友：按“会来几个人”在一排座位上放包（占位 token）。编码器是后到的大巴，带来真正的乘客（embedding 行）。 merge 就是“按座位从左到右，把包换成人”：第 k 个包的位置坐第 k 位乘客。
> 图像和音频是两个旅行团，各自按自己的包排序，可以交错坐。人数和包数对不上——不管多一个还是少一个——都要**当场报错**：线上这通常意味着前后端 processor 版本不一致，悄悄截断只会让模型“看错图”。
> 还有一条礼仪：不要在别人的座位表（`text_embeds`）上直接涂改，先复印一份（`clone`）——调用方可能还要复用它。

© 2026 kzhou003

类比 · 对折纸条：音频 token 数

AuT 把 mel 帧按 100 帧一窗处理，每个窗口过 3 次 stride-2 卷积。每次卷积像把纸条**对折**：长度变成 `(len - 1) // 2 + 1`，即 ceil(len / 2)，奇数时多出来的半格也算一格。 100 → 50 → 25 → 13，所以每个完整窗口贡献 13 个 token；只有不满 100 的尾巴需要真的折三次。余数为 0 时 `(0 - 1) // 2 + 1 = 0` ——这依赖 floor division 向负无穷取整。

类比 · 炖汤分次加料：DeepStack

ViT 浅层的特征偏"细节"，深层偏"语义"。DeepStack 不是只在开锅时放一次料，而是按配方在**第 i 次搅拌之后**加入第 i 份料（level i 在 layer i **之后**注入），而且只加进**有蔬菜的那几碗**（视觉 token 的行）。
先加料再搅拌（注入放在 layer 之前）味道就不一样了——可是文本行根本没加料，只尝文本行是发现不了这个 bug 的。

框架视角 · 数据流 / shape trace

#	tensor	shape	发生了什么
1	`num_mel_frames`	int / `[B]`	余数窗口三次 `(len-1)//2+1` + 每个完整窗口 13；只用 `% // + - *`，int 和 tensor 都适用
2	`input_ids`	`[T]`	前端已经按 token 数展开占位符
3	`masks[m]`	`bool[T]`	`input_ids == placeholder_ids[m]`；每个模态都返回（没出现就是全 False）
4	校验	—	`mask.sum()` vs `mm_embeds[m].shape[0]`；缺 embedding 但有占位也报错
5	`inputs_embeds`	`[T, H]`	`text_embeds.clone()`，布尔索引按升序位置写入第 0, 1, 2… 行（转成输出 dtype）
6	layer i	`[T, H]`	`hidden = layer(hidden)`
7	DeepStack i	`hidden[visual_mask]: [n_vis, H]`	i < K 时，在视觉行上加 `deepstack[i]`，返回新 tensor

易错点

- `audio_token_count` 里用 `if r == 0`、`int()`、`math.floor`：对多元素 tensor 不可用；公式本身已正确给出 0。
- `out = text_embeds` 没有 clone，写入变成原地修改调用方的 tensor。
- 以 `mm_embeds` 为主循环：漏掉"有占位符但没给 embedding"的报错。
- 数量不一致时截断 / 补零而不是报错。
- DeepStack 注入放在 layer 调用**之前**（off-by-one），或用 `zip(layers, deepstack)` 把多出来的 layer 跳过了。
- `hidden[visual_mask] += …` 原地改了输入；或 `out += visual_embeds`（n_vis 行对 T 行）。

交互图：`nano-vllm-omni-drills/hints/m04_overview.html`

对照源码： `model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1018`
`model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1137-1168`
`model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:605-606` HF: _get_feat_extract_output_lengths HF: Qwen3OmniMoeThinkerTextModel._deepstack_process

© 2026 kzhou003

笔记 / 我踩到的坑：

© 2026 kzhou003

#5 · M05 Talker MTP：一帧内预测 15 个残差 codebook ⏱ 60 min 难 □ 完成 日期 _____

> talker 每一步只采样 codebook 0；剩下的 15 个残差 codebook 由一个小 transformer（code predictor）在同一步内自回归补全。这就是 talker 每个 token 背后的"1 次大模型 + 15 次小模型"。

在系统中的位置

talker（stage 1）输出的是 RVQ 音频 codec：每一帧 G=16 个 codebook。talker LLM 采样 CB0 后，code predictor 以 `[talker 最后一层 hidden, CB0 的 embedding]` 为起点，逐个预测 CB1..CB15；下一步 talker 的输入 embedding 是这一帧 **16 个 codec embedding 之和 + 本步对应的 thinker 文本向量**。

```
seq = [h_talker, E0(cb0)]                 # 长度 2
for j in 0..G-2:                          # 预测 codebook j+1
    hj   = CP(seq)[:, -1]                 # 小 transformer，整段重算
    code = argmax(lm_head[j](hj))
    seq.append(codec_embedding[j](code))  # 注意 embedding 表也按 j 取
next_input = E0(cb0) + Σ_j codec_embedding[j](code_j) + text_step
```

为什么 vllm-omni 选择 re-prefill（不用 KV cache）

序列最长只有 17，O(T²) 的重算几乎免费；没有 KV 状态就意味着形状固定、可以整体包成一张 CUDA graph（P17），也不需要管理 per-request cache。batch 很大时 re-prefill 的 FLOPs 变得可观，vllm-omni 又提供了可选的增量 KV 版本——两者必须给出完全相同的 code。

要实现的函数

- `predict_residual_codes`（re-prefill，greedy）——对拍 HF `code_predictor.generate(do_sample=False)`，后者内部用的是 KV cache；
- `predict_residual_codes_kv`（transformers.DynamicCache 增量版）；
- `next_talker_input`：下一步 talker 输入，对拍 HF talker 的 `prepare_inputs_for_generation`。

真实服务里残差 codebook 用 top-k 50 / top-p 0.8 采样，本题为了可对拍使用 greedy。

文件： `mine/m05_talker_mtp.py` **验收：** `pytest tests/test_m05_talker_mtp.py` **提示：** A-5 **测试：** B-5 **参考解：** C-5

接口骨架

```
"""M05 · Talker MTP: predict the residual codebooks of one audio frame.

`cp` is a HF Qwen3OmniMoeTalkerCodePredictorModelForConditionalGeneration. You may only use:
  cp.config.num_code_groups (G), cp.model(inputs_embeds=..., [past_key_values=, use_cache=])
  .last_hidden_state, cp.lm_head[j] (j = 0..G-2), cp.model.codec_embedding[j] (j = 0..G-2).
predict_residual_codes(cp, last_talker_hidden[B, H], cb0_embed[B, H]) -> codes [B, G-1]
  Greedy. The sequence starts as [last_talker_hidden, cb0_embed]. For j = 0..G-2: run the
  transformer over the WHOLE sequence (re-prefill, no cache), take the last position, logits
  = lm_head[j], code = argmax, then append codec_embedding[j](code) to the sequence.
predict_residual_codes_kv(...): identical result with an incremental KV cache
  (transformers.DynamicCache): prefill the 2 initial positions, then feed 1 position per step.
next_talker_input(cp, cb0_embed, residual_codes, text_step) -> [B, H]
  cb0_embed + sum_j codec_embedding[j](residual_codes[:, j]) + text_step
"""
import torch


def predict_residual_codes(cp, last_talker_hidden: torch.Tensor, cb0_embed: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError
```


© 2026 kzhou003

```
def predict_residual_codes_kv(cp, last_talker_hidden: torch.Tensor, cb0_embed: torch.Tensor) -> torch.Tensor:
    raise NotImplementedError


def next_talker_input(cp, cb0_embed: torch.Tensor, residual_codes: torch.Tensor, text_step: torch.Tensor) ->
torch.Tensor:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ re-prefill 结果与 HF generate（KV cache）完全一致
- ✓ KV cache 版本与 re-prefill 版本一致
- ✓ batch 中各行互不影响
- ✓ 下一步 talker 输入与 HF 的求和方式一致

面试官追问

- 为什么 seed 化的逐行采样会阻止整张 CUDA graph replay？vllm-omni 怎么处理？
- MRv2 的 “eager frames” 为什么能在采样出 CB0 的同一步就完成整帧？

直觉与类比

类比 · 量身高：米 → 分米 → 厘米 → 毫米

CB0 像是先报"大概 1 米 7"。CB1 不是重新量一遍，而是回答"还差多少？"——"再加 3 厘米"；CB2 回答"还差多少？"——"再减 2 毫米"。 每一级只修正**上一级留下的误差**，所以必须知道前面几级已经报了什么。

talker 大模型 = 用卷尺量出"1 米 7"的人；code predictor = 拿游标卡尺的助手，接过卷尺读数（`last_talker_hidden` + CB0），一级一级往下补。 每一级用的是**不同的刻度尺**：第 j 步有自己的 `lm_head[j]`（读数）和 `codec_embedding[j]`（把读数记到纸上）。

类比 · 背短诗 vs 记笔记

re-prefill：每想下一句，都把已经写好的整首诗从头念一遍。诗只有十几行时，这几乎不花时间，而且每次的动作**一模一样**（形状固定），很好"彩排"——这正是 CUDA graph 喜欢的。

KV cache：把读过的每一行的要点记在笔记本上，下次只读新的一行。行数多、同时背诗的人（batch）多时更省力，但多了一本要管理的笔记本。

两种做法读出来的下一句必须**完全相同**——因为因果注意力里，前面每一行的"要点"根本不会因为后面加了新行而改变。

框架视角：shape trace 与易错点

G=5，B=2，H=64，vocab=12（与上面的例子同一个模型）。

步	张量	形状	发生了什么
init	`seq`	`[2, 2, 64]`	两个 `[B, H]` 叠出长度维：位置 0 = talker 隐状态，位置 1 = CB0 的 embedding
j=0	`cp.model(inputs_embeds=seq).last_hidden_state`	`[2, 2, 64]`	小 transformer 跑整段；取最后一个位置 → `[2, 64]`
	`lm_head[?](h)` → argmax	`[2, 12]` → `[2]`	得到 CB1 = `[8, 7]`

© 2026 kzhou003

	`codec_embedding[?](code)`	`[2, 64]` → `[2, 1, 64]`	补出长度维，拼到 seq 末尾
j=1	`seq`	`[2, 3, 64]`	CB2 = `[0, 1]`
j=2	`seq`	`[2, 4, 64]`	CB3 = `[0, 4]`
j=3	`seq`	`[2, 5, 64]`	CB4 = `[6, 5]`；这是最后一个码
out	收集到的 4 个 `[2]`	`[2, 4]`	按列拼：`[[8,0,0,6], [7,1,4,5]]`
next	`next_talker_input`	`[2, 64]`	G 个 codec embedding 之和 + `text_step`

表里的 `?` 留给你填（第 4 节第 2 步会揭晓）。

交互图：`nano-vllm-omni-drills/hints/m05_overview.html`

对照源码： `model_executor/models/qwen3_omni/qwen3_omni.py:1044-1069`
`model_executor/models/common/qwen3_code_predictor.py:706-722`
`model_executor/models/common/qwen3_code_predictor.py:1357-1440` `worker/gpu_model_runner.py:243-272` HF: Qwen3OmniMoeTalkerCodePredictorModelForConditionalGeneration HF: Qwen3OmniMoeTalkerForConditionalGeneration.prepare_inputs_for_generation

笔记 / 我踩到的坑：

© 2026 kzhou003

#6 · M06 Code2Wav 的流式因果卷积与转置卷积 ⏱ 60 min 难 □ 完成 日期 _____

> Code2Wav 把 codec 上采样成波形，核心是因果 Conv1d 和因果 ConvTranspose1d。流式解码要求：按任意大小的块喂进去，拼起来的输出与一次性整段处理逐样本一致，而且每个样本一旦"确定"就立刻吐出来。

在系统中的位置

stage 2（Code2Wav）把 `[B, 16, T]` 的 codes 变成 `[B, 1, T·upsample]` 的波形：embedding 平均 → 滑窗 transformer → **因果转置卷积上采样** → ConvNeXt → 多级（因果卷积 + SnakeBeta）解码。vllm-omni 默认用"左上下文重算再切片"的无状态方案（P14）；本题练习另一种做法——**带状态的流式卷积**，它是理解两种方案取舍的关键。

因果 Conv1d（stride 1）

离线模块在整段信号左边补 `(K−1)·d` 个 0（d 为 dilation），所以输出长度 = 输入长度，第 t 个输出只依赖输入 ≤ t。流式版只需保存最近 `(K−1)·d` 个输入作为状态，每次把状态拼到新块前面再卷积。

因果 ConvTranspose1d（stride s，kernel K）

```
输入帧 i 写入输出位置 [i·s, i·s + K)     （重叠相加）
离线：y = conv_transpose(x)，长度 (L−1)·s + K，再裁掉开头 K−s 和结尾 K−s 个样本
流式：收到 n 帧后，位置 < n·s 的样本不会再被后续帧修改 → 可以输出
      （再减去全局开头裁掉的 K−s 个）；flush 时输出剩余部分并裁掉结尾 K−s 个
```

易错点：**bias 只能在输出时加一次**。如果每块卷积都带 bias，重叠区会被加两次。

验收的"尽早输出"

喂入第 n 帧后，累计输出必须正好是 `max(0, n·s − (K−s))` 个样本——多了说明输出了未确定的样本，少了说明增加了不必要的延迟（直接影响首包音频延迟 TTFP）。

文件： `mine/m06_streaming_causal_conv.py` **验收：** `pytest tests/test_m06_streaming_causal_conv.py`
提示： A-6 **测试：** B-6 **参考解：** C-6

接口骨架

```
"""M06 · Streaming versions of Code2Wav's causal Conv1d and causal ConvTranspose1d.

StreamingCausalConv1d (stride 1, dilation d, kernel K)
  The offline module left-pads the whole signal with (K-1)*d zeros. step(x[B, C, t]) must
  return exactly t outputs and, concatenated over any chunking, equal the offline output.
  Keep the last (K-1)*d inputs as state.
StreamingCausalTransConv1d (stride s, kernel K, weight [C_in, C_out, K])
  Input frame i writes output positions [i*s, i*s + K) (overlap-add). The offline module trims
  K - s samples at the START and at the END. step(x) returns every sample that is already
  final (after n frames: positions < n*s, minus the global left trim); flush() returns the rest
  minus the right trim. Add the bias once per output position, not once per contribution.
from_hf(m) builds each class from the HF module (weights .conv.weight / .conv.bias).
"""
import torch


class StreamingCausalConv1d:
    def __init__(self, weight: torch.Tensor, bias: torch.Tensor | None, dilation: int = 1) -> None:
        raise NotImplementedError

    @classmethod
    def from_hf(cls, m) -> "StreamingCausalConv1d":  # HF Qwen3OmniMoeCausalConvNet (m.conv, m.dilation, m.stride)
        raise NotImplementedError

    def step(self, x: torch.Tensor) -> torch.Tensor:
```


© 2026 kzhou003

```
        raise NotImplementedError


class StreamingCausalTransConv1d:
    def __init__(self, weight: torch.Tensor, bias: torch.Tensor | None, stride: int) -> None:
        raise NotImplementedError

    @classmethod
    def from_hf(cls, m) -> "StreamingCausalTransConv1d":  # HF Qwen3OmniMoeCausalTransConvNet (m.conv)
        raise NotImplementedError

    def step(self, x: torch.Tensor) -> torch.Tensor:
        raise NotImplementedError

    def flush(self) -> torch.Tensor:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 因果卷积：任意分块与离线一致、每块输出数等于输入数（含 dilation、K=1）
- ✓ 转置卷积：任意分块 + flush 与离线一致（多种 K/s）
- ✓ 转置卷积每喂 1 帧后的累计输出数恰好是理论最早值

面试官追问

- 有状态流式 vs 无状态"左上下文重算"：对 batching、abort、CUDA graph 各有什么影响?
- Code2Wav 里的滑窗 attention 如何做成流式?

直觉与类比

类比 · 因果卷积 = 带一张"便签"的抄写员

抄写员每写一个字都要回头看前面 `(K−1)·d` 个字（比如 K=3, d=2 时看"前 2 个"和"前 4 个"）。稿纸一页一页送来，翻页时前一页已经收走了—— 所以他在翻页前，把**最后 `(K−1)·d` 个字抄在便签上**，贴到下一页的最前面。第一页之前的便签是空白（全 0），这正好等于离线版"左边补 0"。

注意：如果新的一页很短（比便签还短），新便签要从"**旧便签 + 新页**"连起来的末尾抄，而不是只从新页里抄。

类比 · 转置卷积 = 每隔 s 格盖一个宽 K 的印章

每来一帧，就在长纸条上从第 `i·s` 格开始盖一个宽 `K` 格的印章（墨色深浅 = 帧的值 × 印章图案）。因为 `K > s`，相邻两个章会**叠印**在 `K−s` 格上。

一格什么时候能剪下来交给下游？当**以后再也不会有章盖到它**：已经来了 n 帧，下一个章最早从 `n·s` 盖起，所以 `< n·s` 的格子都定稿了。

纸的底色（bias）只能刷一次——如果每个章自带底色，叠印的地方底色就深了一倍。离线版还会把纸条两头各剪掉 `K−s` 格：开头那一刀只剪一次（全局位置），结尾那一刀要等收工（flush）时才知道在哪。

框架视角：shape trace 与易错点

阶段	张量	形状	发生了什么
Conv · 首块前	`state`	`[B, C_in, (K−1)·d]`	全 0；B / dtype / device 第一次 step 才知道，所以懒创建
Conv · step	`buf = cat([state, x], −1)`	`[B, C_in, (K−1)·d + t]`	"便签 + 新页"
	`F.conv1d(buf, w, b, dilation=d)`	`[B, C_out, t]`	无 padding：长度减掉 (K−1)·d，正好剩 t

© 2026 kzhou003

	新 `state`	`[B, C_in, (K−1)·d]`	从 buf 的**末尾**取
TransConv · step	`F.conv_transpose1d(x, w, None, stride=s)`	`[B, C_out, (t−1)·s + K]`	块内的叠印已经做好；覆盖全局 `[frames·s, frames·s + len)`
	`acc`（从全局位置 `base` 开始）	`[B, C_out, ≤ K−s]`（每次 step 之后）	块与块之间的叠印：把新结果加到 acc 对应位置
	输出	`[B, C_out, ·]`	全局 `[base, frames·s)`，加 bias 一次，再去掉落在 `[0, K−s)` 里的部分
TransConv · flush	输出	`[B, C_out, ·]`	剩余 pending 全部定稿，去掉结尾 K-s 个（通常恰好全被裁掉）
累计	喂入 n 帧后已输出	`max(0, n·s − (K−s))`	例 B：0, 3, 6, 9, 12, 15 …（K = 2s 时就是 (n-1)·s）

交互图：`nano-vllm-omni-drills/hints/m06_overview.html`

对照源码： `model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:124-218`

`model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:447-508` HF: Qwen3OmniMoeCausalConvNet HF: Qwen3OmniMoeCausalTransConvNet

笔记 / 我踩到的坑：

© 2026 kzhou003

#7 · M07　按 stage 加载与打包权重　45 min　中　□ 完成　日期 _____

> 一个 Qwen3-Omni checkpoint 同时包含 thinker、talker、code2wav 的权重。每个 stage 进程只该打开自己的 shard、只取自己的张量，并且把 q/k/v、gate/up、各 expert 打包成 kernel 友好的布局。

在系统中的位置

三个 stage 进程各自加载同一个 checkpoint 目录。vllm-omni 的做法是：同一个 wrapper 类按 `model_stage` 只构建一个子模块，`load_weights` 按前缀 `thinker./talker./code2wav.` 分流（参考 `qwen3_omni.py:1725-1783`）。本题你要做得更进一步：根据 `model.safetensors.index.json` 的 `weight_map` 只打开相关 shard。

打包布局（vLLM 风格）

```
qkv  = cat([q_proj, k_proj, v_proj], dim=0)     # [(H + 2·KVH)·D, hidden] 一次 GEMM 出三者
w13  = stack_e cat([gate_e, up_e], dim=0)       # [E, 2I, hidden]   FusedMoE 的 w1/w3
w2   = stack_e down_e                           # [E, hidden, I]
router = mlp.gate.weight                        # [E, hidden]
```

key 的形式（tiny 模型与真实 checkpoint 相同）：`thinker.model.layers.{i}.self_attn.q_proj.weight`、`thinker.model.layers.{i}.mlp.experts.{e}.gate_proj.weight`、`thinker.model.layers.{i}.mlp.gate.weight` 等。

完整性检查

“少加载一个张量”在 PyTorch 里不会报错——那个参数会保持随机初始化，模型输出悄悄变成噪声。所以：缺 key、多 key、expert 编号不连续、某个 expert 缺 gate/up/down，都必须抛 `LoadError`。

文件： `mine/m07_stage_weight_loading.py`　**验收：** `pytest tests/test_m07_stage_weight_loading.py`

提示： A-7　**测试：** B-7　**参考解：** C-7

接口骨架

```
"""M07 · Stage-aware checkpoint loading and weight packing.

A Qwen3-Omni checkpoint holds every stage: thinker.*, talker.*, code2wav.*. Each stage process
must open only its shards and load only its tensors, then pack them for fast kernels.
shards_for_stage(weight_map, stage): sorted shard files containing at least one key of the stage.
select_stage(state, stage, strip): keep keys starting with `strip` (e.g. "thinker.model.") and
  remove that prefix.
pack_layer(sd, i) -> PackedLayer for "layers.{i}.*":
  qkv = cat(q_proj, k_proj, v_proj) along rows; w13[e] = cat(gate_proj_e, up_proj_e) along
  rows; w2[e] = down_proj_e; router = mlp.gate.weight. Expert ids must be exactly 0..E-1 and
  complete; any missing or unexpected key in the layer -> LoadError.
load_thinker_text(state, num_layers) -> ({"embed_tokens", "norm"}, [PackedLayer]); anything
  else under thinker.model. besides layers/embed_tokens/norm -> LoadError.
"""
import re
from dataclasses import dataclass

import torch

STAGE_PREFIX = {"thinker": "thinker.", "talker": "talker.", "code2wav": "code2wav."}


class LoadError(Exception):
    pass


@dataclass
class PackedLayer:
```


© 2026 kzhou003

```
    input_layernorm: torch.Tensor
    qkv: torch.Tensor               # [(H + 2*KVH) * D, hidden]  rows: q | k | v
    q_norm: torch.Tensor
    k_norm: torch.Tensor
    o_proj: torch.Tensor
    post_attention_layernorm: torch.Tensor
    router: torch.Tensor            # [E, hidden]
    w13: torch.Tensor               # [E, 2*I, hidden]  rows: gate | up
    w2: torch.Tensor                # [E, hidden, I]


def shards_for_stage(weight_map: dict[str, str], stage: str) -> list[str]:
    raise NotImplementedError


def select_stage(state: dict[str, torch.Tensor], stage: str, strip: str) -> dict[str, torch.Tensor]:
    raise NotImplementedError


def pack_layer(sd: dict[str, torch.Tensor], i: int) -> PackedLayer:
    raise NotImplementedError


def load_thinker_text(state: dict[str, torch.Tensor], num_layers: int) -> tuple[dict, list[PackedLayer]]:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 按 weight_map 只挑本 stage 的 shard
- ✓ select_stage 去前缀且丢弃其他 stage
- ✓ 打包后的 qkv / w13 / w2 复现 HF 的投影与 expert 输出
- ✓ 缺失与多余 key 都报错

面试官追问

- TP=2 时 qkv 怎么切？为什么不能简单地按行对半切打包后的 qkv?
- talker 进程为什么还需要 thinker 的部分权重（Qwen2.5-Omni 的 embed_tokens）？

直觉与类比

类比 · 合租仓库搬家

三户人家（thinker / talker / code2wav）的东西存在同一个仓库里，分装在若干个大箱子（shard）里，每件物品贴着"户名.房间.物品"的标签（key）。仓库门口有一张总清单（`weight_map`）：哪件东西在哪个箱子。搬家时每户只拉**装有自家东西**的箱子（`shards_for_stage`）——有的箱子里混着两户的东西，两户都得拉。开箱后只拿标签以自家户名开头的物品，并把标签上的"户名."撕掉（`select_stage`），因为进了自己家，就不用写户名了。

标签必须**连同那个点**一起比：户名 `thinker` 和 `thinker_extra` 前几个字一样，但不是一家人。

类比 · 钥匙串与装箱单

打包：q、k、v 三把钥匙开的是同一扇门（输入都是 hidden），串到一个钥匙圈上（`qkv = cat(..., dim=0)`），开门只需一次动作（一次 GEMM），之后再按 `[H·D, KVH·D, KVH·D]` 把结果分开。每个 expert 的 gate / up 同理串成 `w13` ——但顺序是约定好的：**先 gate 后 up**，串反了钥匙照样插得进去，只是开错了门。

对账：搬完后拿装箱单逐项打勾（`used` 集合）。没打勾的（多余 key）和清单上有但没找到的（缺失 key）都要喊停（`LoadError`），而不是默默装修入住。

© 2026 kzhou003

框架视角：shape trace 与易错点

用 tiny 随机 thinker（hidden=64，H=4，KVH=2，D=16，E=6，I=32）跑一遍 `load_thinker_text`，第 0 层（实测）：

HF key（去前缀后）	HF 形状	打包后	发生了什么
`self_attn.q_proj.weight`	`[64, 64]`	`qkv [128, 64]`	沿 dim 0 拼：64 + 32 + 32 = (4 + 2·2)·16 = 128 行。`(x @ qkv.T).split([64, 32, 32])` 的 q 与 HF `q_proj(x)` 最大差 0.0
`self_attn.k_proj.weight`	`[32, 64]`		
`self_attn.v_proj.weight`	`[32, 64]`		
`mlp.experts.{e}.gate_proj / up_proj`	`[32, 64]` ×2 ×6	`w13 [6, 64, 64]`	每个 expert 内 cat，再按 e = 0..5 stack；SwiGLU 还原后与 HF expert 输出最大差 0.0
`mlp.experts.{e}.down_proj`	`[64, 32]` ×6	`w2 [6, 64, 32]`	按 e stack
`mlp.gate.weight`	`[6, 64]`	`router [6, 64]`	名字叫 gate，但它是 router，不是 gate_proj
norm × 4，`o_proj`	原样	原样	input_layernorm / post_attention_layernorm / q_norm / k_norm
层外	`embed_tokens.weight`，`norm.weight`		tiny 模型共 56 个 key：2 个层外 + 2 层 × 27 个

交互图：`nano-vllm-omni-drills/hints/m07_overview.html`

对照源码： `model_executor/models/qwen3_omni/qwen3_omni.py:1725-1783`
`model_executor/models/qwen3_omni/qwen3_omni_moe_talker.py:61-77`
`model_executor/models/qwen2_5_omni/qwen2_5_omni.py:1009-1022` HF: Qwen3OmniMoeThinkerTextModel.state_dict()

笔记 / 我踩到的坑：

© 2026 kzhou003

#8 · M08　串起来：tiny thinker 的 KV-cache greedy 解码

60 min　中　□ 完成　日期 _____

> 把 M01/M02/M03/M07 拼成一个只做文本的 thinker：prefill 一次、之后每步只算 1 个 token，生成的 token 序列要和 HF 每步全量重算的 greedy 完全一致。

在系统中的位置

这就是 stage 0 的 model runner 每一步做的事（去掉了多模态和 paged KV）。做完这题，你就有了一个可以接到 Track B 的 scheduler 上、能被 Track A 的 orchestrator 调度的"真模型"。

结构

```
h = embed_tokens[ids]                                  # [T, hidden]
for layer:  h = h + Attn(rms(h, ln1), cos, sin, cache)  # M02 (cache 每层一个)
            h = h + MoE(rms(h, ln2))                    # M03
logits = rms(h, norm) @ lm_head^T                       # [T, vocab]
```

位置：纯文本时三轴相同，`positions = arange(start, start+T).expand(3, T)`，用 M01 的 interleaved MRoPE。greedy：prompt 一次 forward（start=0），之后每步只喂上一个 token，`start = len(prompt) + 已生成数 − 1`。

依赖

本题 `from .m01_... import ...` 等直接使用你自己写的 M01/M02/M03/M07。任何一个前置题的小 bug 都会在这里以"第 k 个 token 开始不一样"的形式暴露。

文件： `mine/m08_tiny_thinker_e2e.py`　**验收：** `pytest tests/test_m08_tiny_thinker_e2e.py`　**提示：** A-8　**测试：** B-8　**参考解：** C-8

接口骨架

```
"""M08 · Assemble a text-only thinker from YOUR m01/m02/m03/m07 and decode greedily.

TinyThinker(state, dims): load with your load_thinker_text; lm_head = state["thinker.lm_head.weight"].
forward(ids[T], start, caches) -> logits [T, V] for tokens at absolute positions start..start+T-1:
  text-only MRoPE positions (the same index on all 3 axes, interleaved layout),
  h = embed[ids]; per layer: h += attn(rms(h, ln1)); h += moe(rms(h, ln2)); logits = rms(h, norm) @ lm_head.T
  One KVCache per layer.
greedy(prompt, max_new): prefill the prompt once, then decode ONE token per forward using the
  caches. Must equal HF full-recompute greedy decoding.
"""
from dataclasses import dataclass

import torch

from .m02_thinker_attention import KVCache  # noqa: F401  (re-exported for the tests)


@dataclass
class Dims:
    num_layers: int
    num_heads: int
    num_kv_heads: int
    head_dim: int
    top_k: int
    norm_topk_prob: bool
    rope_theta: float
    mrope_section: list[int]
    eps: float = 1e-6


class TinyThinker:
```


© 2026 kzhou003

```
    def __init__(self, state: dict[str, torch.Tensor], dims: Dims) -> None:
        raise NotImplementedError

    def forward(self, ids: torch.Tensor, start: int, caches: list) -> torch.Tensor:
        raise NotImplementedError

    def greedy(self, prompt: list[int], max_new: int) -> list[int]:
        raise NotImplementedError
```

验收要点（测试会检查）

✓ prefill logits 与 HF 一致
✓ KV-cache greedy 12 个 token 与 HF 全量重算 greedy 完全一致（两组随机权重）

面试官追问

- 把 KVCache 换成 P06 BlockManager + P09 paged attention。
- 加上 M04 的多模态合并，位置改用 P18 的 MRoPE 位置。

直觉与类比

类比 · 新盖的房子要编门牌号

KV cache 是一条街：已经盖好的房子（缓存里的 token）门牌号是 0, 1, 2, ..., L-1，每栋房子的 K 在入住时就按自己的门牌"旋转"好了（RoPE）。现在要盖一栋新房子，它的门牌号必须是 **L**——街上已有房子的数目。注意力看的是"两栋房子相隔多远"（RoPE 的相对位置）。如果你把新房子编成 L+1，它和每一个邻居的距离都多算了 1：不会报错，看起来也差不多，但邻里关系悄悄变了——往往几步之后才"走错路"。

类比 · 接龙游戏

prefill = 先把题目（prompt）整段读完，顺手说出第一个接龙词。之后每一轮只需要**听最新的一个词**（只喂 `out[-1:]`），因为前面的内容都记在笔记本上（cache）了。
接龙要说 `max_new` 个词，就只需要 `max_new` 轮：第一轮的词来自读题，最后一个词说出口之后不必再听一轮。每轮开始时也要确认：笔记本上记了几个词（`len(cache)`），这一轮的新词就排第几（`start`）。

框架视角：shape trace 与易错点

阶段	张量	形状（prefill / decode）	发生了什么
位置	`positions`	`[3, 5]` / `[3, 1]`	纯文本：三行都是 `start … start+T−1`。这时 interleaved MRoPE 与普通 1D RoPE 完全相同（实测 allclose）——但仍走 M01 的 MRoPE 接口
	`cos, sin`	`[5, 16]` / `[1, 16]`	只覆盖**新** token；所有层共享
embed	`h = embed[ids]`	`[5, 64]` / `[1, 64]`	查表
每层	`h + attn(rms(h, ln1), cos, sin, cache)`	`[T, 64]`	pre-norm 残差；cache 就地追加 T 个：0→5 / 5→6
	`h + moe(rms(h, ln2))`	`[T, 64]`	top-2 of 6 experts
输出	`rms(h, norm) @ lm_head.T`	`[5, 97]` / `[1, 97]`	`lm_head [97, 64]` 来自 `state["thinker.lm_head.weight"]`，不在 `thinker.model.` 下

© 2026 kzhou003

采样	`int(logits[-1].argmax())`	标量	只用最后一行

交互图：`nano-vllm-omni-drills/hints/m08_overview.html`

对照源码： `model_executor/models/qwen3_omni/qwen3_omni.py:235-267` HF: Qwen3OmniMoeThinkerTextModel

笔记 / 我踩到的坑：

© 2026 kzhou003

Track A · 前端与 Orchestrator

API 进程里的并发：线程↔事件循环、单写者路由、终态、abort、RPC。

© 2026 kzhou003

#9 · P02 Orchestrator v1：线性多 stage 路由 ⏱ 60 min 中 □ 完成 日期 _____

> 你在写 vllm-omni 的大脑。三个 stage（thinker→talker→code2wav）各自吐输出，你要把它们正确地串起来，并且保证每个请求恰好结束一次。

在系统中的位置

API 进程里有一个独立的 **orchestrator 线程**，跑着自己的 asyncio 事件循环。它握着每个 stage 的 client（对端是 stage 子进程里的 EngineCore）。它要做三件事：接收前端提交的请求；读取每个 stage 的输出；决定输出是交给客户端、转发给下一个 stage，还是两者都做。

```
前端 add_request ─► [request handler] ─► stage0.submit
                                              │
 stage0 / stage1 / stage2 的 get_output ─► reader×3 ─► ready_q ─► [dispatcher] ─► 输出给前端
                                                                       └─► stage(s+1).submit
```

需求

1. **单写者**：每个 stage 一个 reader task 只负责搬运；**唯一**的 dispatcher 修改请求状态。asyncio 是协作式调度，只要不在"读-改-写"中间 await，就不需要锁。
2. **先注册后提交**：`request_states[rid]` 必须在第一次 `await stages[0].submit()` 之前就存在。原因是 submit 内部可能让出控制权，而 stage 可能非常快地吐出第一条输出。
3. **转发**：stage s 对某请求 `finished=True`，且 s < max(final_stage_ids) 时，用 `forward_fn(s+1, prompt, stage s 的全部输出)` 构造下一个 stage 的输入并提交。
4. **输出与终态**：只有 `final_stage_ids` 里的 stage 的输出会给客户端（Qwen3-Omni 是 {0: 文本, 2: 音频}）。每个请求**恰好一条** `finished=True`，而且它是最后一条。条件是"所有 final stage 都完成"。
5. **清理**：发出终态后删除状态；之后到达的、属于未知请求的输出丢弃并计数（`dropped_outputs`）。

例子

```
stage0: r:t0 r:t1 r:t2 r:t3 r:t4(fin)      → 客户端收到 5 条 stage0 输出（finished=False）
stage1: [R:T0..R:T2] [R:T3,R:T4](fin)      → 不给客户端，fin 时转发给 stage2
stage2: wav(R:T0) ... wav(R:T4)(fin)       → 最后一条 finished=True
```

文件： `mine/p02_orchestrator.py` **验收：** `pytest tests/test_p02_orchestrator.py` **提示：** A-9 **测试：** B-9 **参考解：** C-9

接口骨架

```
"""P02 · Orchestrator v1: linear multi-stage routing with a single-writer dispatcher.

Requirements
- start(): one reader task per stage client (awaits client.get_output()) feeding ONE
  ready queue; one dispatcher task processes it serially; one request-handler task
  consumes add_request() messages.
- add: register ReqState in self.request_states BEFORE awaiting stages[0].submit().
- When stage s finishes a request and s < max(final_stage_ids): build the next stage's
  inputs with forward_fn(s+1, prompt, all data stage s produced) and submit to s+1.
- Outputs of stages in final_stage_ids go to the client (next_output()). Exactly one
  message per request has finished=True: the last one, sent once every final stage finished.
- After the terminal message the request state is removed. Outputs for unknown request
  ids are dropped and counted in self.dropped_outputs.
"""
import asyncio
from collections.abc import Callable
from dataclasses import dataclass, field
from typing import Any, Callable, Protocol
```


© 2026 kzhou003

```python
# ---------------------------------------------------------------- given types
@dataclass
class StageOutput:
    request_id: str
    data: Any
    finished: bool


@dataclass
class OrchOutput:
    request_id: str
    stage_id: int | None
    data: Any
    finished: bool
    error: str | None = None


class StageClient(Protocol):
    async def submit(self, request_id: str, inputs: Any) -> None: ...
    async def get_output(self) -> list[StageOutput]: ...
    async def abort(self, request_ids: list[str]) -> None: ...


@dataclass
class ReqState:
    request_id: str
    prompt: Any
    submitted: set[int] = field(default_factory=set)
    finished: set[int] = field(default_factory=set)
    stage_data: dict[int, list] = field(default_factory=dict)
    pending_final: OrchOutput | None = None  # used by P03


# ---------------------------------------------------------------- implement below
class Orchestrator:
    def __init__(
        self,
        stages: list[StageClient],
        final_stage_ids: set[int],
        forward_fn: Callable[[int, Any, list], Any],
    ) -> None:
        self.stages = stages
        self.final_stage_ids = frozenset(final_stage_ids)
        self.last_stage = max(final_stage_ids)
        self.forward_fn: Callable[[int, Any, list[Any]], Any] = forward_fn
        self.request_states: dict[str, ReqState] = {}
        self.dropped_outputs = 0
        self._req_q: asyncio.Queue = asyncio.Queue()
        self._ready_q: asyncio.Queue = asyncio.Queue()
        self._out_q: asyncio.Queue = asyncio.Queue()
        self._tasks: list[asyncio.Task] = []

    async def start(self) -> None:
        raise NotImplementedError

    async def shutdown(self) -> None:
        raise NotImplementedError

    async def add_request(self, request_id: str, prompt: Any) -> None:
        raise NotImplementedError

    async def next_output(self, timeout: float | None = None) -> OrchOutput:
        raise NotImplementedError

    # Suggested internals (P03 overrides the last two):
    #   async def _reader(self, sid, client)
    #   async def _request_handler(self)
    #   async def _dispatcher(self)
    #   async def _handle_add(self, rid, prompt)
    #   async def _handle_output(self, sid, o: StageOutput)
    #   def _cleanup(self, rid)
```


© 2026 kzhou003

验收要点（测试会检查）

- ✓ 单请求 3 stage：stage0 文本与 stage2 音频都按序到达，唯一的 finished 是最后一条
- ✓ 30 个并发请求互不串扰；每个 stage 恰好收到每个请求一次
- ✓ 只有最后一个 stage 是 final 时，stage0 输出不会泄露给客户端
- ✓ stage 在 submit() 内部就吐出第一条输出时不能丢（先注册后提交）
- ✓ 未知 request 的输出被丢弃并计数，不会崩

面试官追问

- 如果 stage 有多个 replica，输出顺序还能保证吗？需要什么前提？（→ P04 的粘性绑定）
- 输出队列为什么可以无界、请求队列为什么应该有界？
- 把 dispatcher 换成每个请求一个 task 会怎样？好处与代价？

交互图：`nano-vllm-omni-drills/hints/p02_flow.html（时序图）· hints/p02_vs_vllm_omni.html（源码对照，见附录 D）`

对照源码： `engine/orchestrator.py:1099-1320` `engine/orchestrator.py:1912-2046` `engine/orchestrator.py:3062-3128` `engine/orchestrator.py:232-271`

笔记 / 我踩到的坑：

© 2026 kzhou003

#10 · P03 Orchestrator v2：async_chunk 预热 + 终态守门

45 min　中　□ 完成　日期 _____

> 开启 async_chunk 后，stage 之间走 connector 直传，orchestrator 不再转发数据。但控制面和数据面是两条路，完成信号可能乱序：音频 stage 先说“我完了”，talker 还没报告完成。

在系统中的位置

async_chunk 模式下，thinker 每 decode 一个 token 就把向量经 connector 发给 talker，talker 每攒够一个窗口就把 codec 发给 code2wav——数据**不经过** orchestrator。orchestrator 只做两件事：在请求到达时**预热**下游 stage（提交一个占位请求，让它在 scheduler 里等第一个 chunk），以及**守门**终态。

为什么需要预热

如果等 thinker 产出第一个 chunk 后才提交 talker 请求，就多了一次“stage→orchestrator→stage”的往返；预热后第一个 chunk 一落地，talker 就能立刻开始。

为什么需要终态守门

```
时间 →
stage2 (audio, final):  ... wav5(fin)                ← 控制面上先到
stage1 (talker):        ............... (fin)        ← 晚到（它的 IPC 慢了一拍）
如果在 wav5(fin) 时就发 finished 并清理：
  – 清理会回收 stage1 仍在使用的资源（SHM、绑定）
  – stage1 的完成消息成了“未知请求”的输出
```

规则：当最后一个 final stage 完成、但 `submitted ⊄ finished` 时，把这条终态消息存进 `pending_final`；等最后一个已提交 stage 也完成时再发出。仍然要求恰好一条终态、且为最后一条。

需求清单

- add：注册状态 → 把 0..last 都标记为 submitted → 先提交 stage 0（真实 prompt），再提交下游（`prewarm_fn(sid, prompt)` 的占位输入）。
- 不做任何转发。
- 清理后到达的输出丢弃计数，状态不复活。

本题直接继承你在 P02 里写的 `Orchestrator`，覆盖 `_handle_add` 和 `_handle_output`。

文件： `mine/p03_async_chunk_orchestrator.py`　**验收：** `pytest tests/test_p03_async_chunk_orchestrator.py`　**提示：** A-10　**测试：** B-10　**参考解：** C-10

接口骨架

```
"""P03 · Orchestrator v2 for async_chunk (builds on YOUR P02).

Stages now exchange data on a side channel (the connector). The orchestrator:
- on add: registers state, marks stages 0..last as submitted, submits the prompt to
  stage 0 and a placeholder prewarm_fn(sid, prompt) to every downstream stage;
- never forwards stage outputs itself;
- emits outputs of final stages, but the terminal message (finished=True) is only
  released once EVERY submitted stage has finished. If the last final stage finishes
  first, hold the message in ReqState.pending_final and emit it when the remaining
  stage(s) finish. Exactly one terminal message, always the last one.
- outputs after cleanup are dropped (self.dropped_outputs), never resurrect state.
"""
from collections.abc import Callable
from typing import Any

from .p02_orchestrator import Orchestrator, OrchOutput, ReqState, StageClient, StageOutput
```


© 2026 kzhou003

```
__all__ = ["AsyncChunkOrchestrator", "OrchOutput", "StageOutput", "ReqState"]


class AsyncChunkOrchestrator(Orchestrator):
    def __init__(
        self,
        stages: list[StageClient],
        final_stage_ids: set[int],
        prewarm_fn: Callable[[int, Any], Any],
    ) -> None:
        super().__init__(stages, final_stage_ids, forward_fn=lambda *_: None)
        self.prewarm_fn = prewarm_fn

    # Override the add path and the per-output handling of your P02.
    async def _handle_add(self, rid: str, prompt: Any) -> None:
        raise NotImplementedError

    async def _handle_output(self, sid: int, o: StageOutput) -> None:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 每个 stage 只被提交一次，下游拿到的是 prewarm 占位输入
- ✓ 非 final 的 stage1 迟 200ms 才报告完成时，终态必须等它（不能提前发 finished）
- ✓ 20 个请求 + 迟到完成：每个请求恰好一个终态且是最后一条
- ✓ 清理之后到达的输出被丢弃，状态不复活

面试官追问

- 为什么过早发终态会导致资源问题？（提示：清理会回收上游仍在写的 SHM）
- 如果 prewarm 的占位 prompt 长度依赖上游结果怎么办?

直觉与类比

类比 · 车间主任和传送带

P02：每道工序（stage）做完**一整批**零件，交给车间主任（orchestrator），主任搬到下一道工序。下一道工序必须**等上一道全部做完**才能开工。

P03：工序之间装了**传送带**（connector）。上一道每做好**一个**零件，就直接放上传送带，下一道马上就能加工——流水线并行起来了。车间主任不再搬零件，只剩两件事：

① **开工时**：通知**所有**工序"有新订单，准备接传送带上的零件"（**预热 prewarm**）；

② **收工时**：确认**每一道**工序都停机了，才给客户签收单（**终态守门**）。

类比 · 提前预热烤箱

不预热：面团揉好了（thinker 的第一个 token），才去打开烤箱（submit talker），还得等它热起来。

预热：客人一下单就把烤箱打开（prewarm），面团一好就直接进炉。对用户来说，就是**第一个音频块更早出来**（首包延迟 TTFA 更低）。

类比 · 快递和签收短信

货物走物流（数据面），"已发货 / 已完成"走短信（控制面）。最后一站的签收短信可能比中转站的"已出库"短信**先到**你手机上。

如果你一看到"已签收"就把订单**归档删除**，几秒后中转站的"已出库"短信到了，你却找不到这张订单了；更糟的是，归档时你可能已经把中转站还在用的**托盘**（共享内存、绑定）收回了。

© 2026 kzhou003

框架视角 · 用 T · SBWC · LIFE 看 P03：只有两处变了

	和 P02 相比
T	不变：同一个事件循环里的 task。新增的 connector 是 stage 进程之间的边界，orchestrator 不碰它。
S	`ReqState` 多用一个字段 `pending_final`；`submitted` 的含义变成"预热过的所有 stage"；`stage_data` 不再需要。
B / W / C	**完全不变**：reader × N、request_handler、dispatcher、三个队列。这就是为什么 P03 能直接继承 P02。
L / I	不变。
F	**变了**：add 时一次性标记 submitted；终态条件多了 `submitted ⊆ finished`；**非 final stage 的完成事件也可能触发发送**（放行 pending_final）。"恰好一条终态、且是最后一条"不变。
E	清理后再到的输出照样丢弃计数，**绝不能**因为 S1 的迟到完成消息而重新建出状态。

交互图：`nano-vllm-omni-drills/hints/p03_overview.html`

对照源码： `engine/orchestrator.py:1954-1967` `engine/orchestrator.py:2036-2040` `engine/orchestrator.py:2711-2870`

笔记 / 我踩到的坑：

© 2026 kzhou003

#11 · P01 OutputSink：线程 → asyncio 的批量唤醒

45 min　中　□ 完成　日期 _____

> orchestrator 线程每秒产生上万条输出，前端 asyncio 循环要消费它们。每条都 call_soon_threadsafe 太贵；只唤醒一次又容易丢唤醒。

在系统中的位置

API 进程里有两个事件循环：前端循环（HTTP、SSE、tokenize）和 orchestrator 线程里的循环。orchestrator 把每个 token、每个音频块作为一条消息交给前端。跨线程唤醒 asyncio 只能用 `loop.call_soon_threadsafe`：它会往 loop 的 self-pipe 写一个字节并加锁，单次开销约微秒级，每秒上万次就很可观。

目标设计

```
生产者（任意线程）                消费者（前端 loop）
put(x):                          get_batch():
  items.append(x)                  ① out = drain();  if out: return out
  if 已有唤醒在途: return          ② event.clear(); 标记"无唤醒在途"  ← 武装
  标记"唤醒在途"                   ③ out = drain();  if out: return out  ← 关键
  call_soon_threadsafe(event.set)  ④ await event.wait()（带超时）
```

丢唤醒是怎么发生的

如果没有第 ③ 步：消费者在 ① 发现队列空 → 生产者此刻 append 一条，并看到"唤醒在途"标志（还是上一轮留下的）于是不唤醒 → 消费者执行 ② 武装 → ④ 睡下。那条消息会一直躺在队列里，直到下一条 put 或超时。测试通过 `after_empty_drain` 钩子把一次 put 精确地插进 ① 和 ② 之间。

需求

- put 线程安全、永不阻塞；同一生产者的顺序保持。
- 在一个"唤醒周期"内最多一次 `call_soon_threadsafe`，次数记录在 `wakeups_scheduled`。
- get_batch 返回 1..max_items 条；超时返回 []。

文件： `mine/p01_output_sink.py`　**验收：** `pytest tests/test_p01_output_sink.py`　**提示：** A-11　**测试：** B-11　**参考解：** C-11

接口骨架

```python
"""P01 · OutputSink: many producer threads -> one asyncio consumer.

Requirements
- put(item): callable from ANY thread, never blocks, FIFO per producer.
- At most ONE loop.call_soon_threadsafe(...) per "wake cycle": while a wakeup is
  already in flight, further puts must not schedule another. Count every scheduled
  wakeup in self.wakeups_scheduled.
- get_batch(max_items, timeout): runs on the bound loop; returns 1..max_items items,
  or [] on timeout. It must never sleep while items are available (no lost wakeups).
- Test hook: call self.after_empty_drain() right after a drain that found nothing and
  BEFORE you re-arm the wakeup. Tests use it to inject a put() from another thread into
  exactly the window where naive implementations lose the wakeup.
"""
import asyncio


class OutputSink:
    def __init__(self) -> None:
        self.wakeups_scheduled = 0
        self.after_empty_drain = lambda: None
        raise NotImplementedError
```


© 2026 kzhou003

```
    def bind_loop(self, loop: asyncio.AbstractEventLoop | None = None) -> None:
        """Remember the consumer loop (default: the running loop) and create its wakeup Event."""
        raise NotImplementedError

    def put(self, item) -> None:
        raise NotImplementedError

    async def get_batch(self, max_items: int = 256, timeout: float | None = None) -> list:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 基本 FIFO
- ✓ 前端未消费时 1000 次 put 只调度 1 次唤醒
- ✓ 超时返回 []
- ✓ 4 个生产者线程 × 5000 条，各自顺序不乱、消费者不饿死
- ✓ 乒乓 300 轮不丢唤醒
- ✓ 在"空 drain 之后、重新武装之前"注入一次 put，必须立即被拿到

面试官追问

- 如果消费者有多个（多个 asyncio loop）还能这样做吗?
- janus.Queue 帮你解决了什么、没解决什么?

直觉与类比

类比 · 信箱和门铃

你（**前端 loop**）住在楼上，楼下有个**信箱**（队列）。邮递员（**orchestrator 线程**）随时会来投信。
你在楼上看不到信箱，只能靠邮递员**按门铃**（`loop.call_soon_threadsafe`）叫你下楼。
但每按一次门铃，你都要放下手头的事、跑一趟楼梯——**很贵**。如果邮递员每投一封信就按一次铃，一秒上万封信，你就整天在跑楼梯。

类比 · 门上的小旗

信箱旁边插一面**小旗**（"唤醒已在途"标志）。规则：
① 邮递员投信后，如果小旗**放倒**着，就把它**竖起来**并按一次门铃；如果小旗**已经竖着**，说明门铃已经按过了、你正在下楼的路上，他就**不再按**。
② 你下楼后，把信箱里的信**一次拿光**。
③ 只有当你准备回楼上休息时，才把小旗**放倒**，表示"下次来信需要再按门铃"。

类比 · 丢唤醒是怎么发生的

你下楼打开信箱，**空的**。你转身准备上楼休息。
就在你转身、**还没来得及放倒小旗**的那一瞬间，邮递员投了一封信。他一看小旗还**竖着**（这是你上一轮下楼时留下的），心想"门铃按过了，他会来拿"，于是**没按门铃**就走了。
你放倒小旗，上楼睡觉。那封信就在信箱里**躺到天亮**（下一封信来、或者你的闹钟 timeout 响）。

用 T · SBWC 框架过一遍（先自己想）

	问题	提示
T	生产者、消费者分别在哪？跨越的是哪条边界?	多个普通线程 → 一个 asyncio loop。框架里"线程 → 事件循环"那一行：只能走哪座桥?

© 2026 kzhou003

S	有哪些共享状态？谁读谁写？	至少有"待取的数据"和"唤醒是否在途"两样。哪个会被多个线程同时修改？需要锁吗？哪些操作在 CPython 里本身就是原子的？
B	谁会阻塞？谁绝不能阻塞？	`put` 永不阻塞；`get_batch` 只能在 loop 里 `await`，不能用 `time.sleep` 或线程锁去等。
F	不变量是什么？	"队列非空 ⇒ 要么消费者醒着，要么有一次唤醒在途"。上面的时间线在哪一步违反了它？
E	边界情况？	超时；`max_items` 比积压的少（剩下的怎么办？下一次 `get_batch` 会不会白等？）；多个生产者交错。

交互图：`nano-vllm-omni-drills/hints/p01_overview.html`

对照源码： `engine/async_engine_utils.py:40-97` `engine/omni_engine_base.py:973-991`

笔记 / 我踩到的坑：

© 2026 kzhou003

#12 · P04 StagePool：粘性绑定、两阶段 abort、故障隔离 ⏱ 60 min 中 □ 完成 日期 _____

> 每个 stage 可以有多个 replica。请求一旦落在某个 replica 上就必须一直在那（KV 在那）。客户端断开要 abort；某个 replica 进程死了，只能让绑定在它上面的请求失败。

在系统中的位置

deploy 里 `num_replicas: 2` 会给一个 stage 起两个独立的 EngineCore 进程。orchestrator 为每个逻辑 stage 维护一个 `StagePool`，负责选 replica。

粘性绑定为什么必须

请求的 KV cache、流式输入的中间状态都在第一次被分配到的那个 replica 上。后续的 update / abort / 下一个 chunk 必须找到同一个 replica，否则对方根本不认识这个请求。新请求在**存活**的 replica 之间 round-robin。

两阶段 abort

```
① partial = processor.collect_partial(rid)   # 只读：拿到已生成的部分文本，用于最后一条输出
② for pool: await pool.abort([rid])           # 引擎侧真正停止；失败就抛出，不进入 ③
③ processor.commit_abort(rid)                 # 提交：丢弃 detokenizer 状态
return OrchOutput(rid, partial, finished=True, error="abort")
```

如果 ③ 先于 ②：引擎 abort 失败时，processor 已经忘了这个请求，引擎却还在为它产出 token，这些输出就成了孤儿。另外，同一个 replica 上的多个请求要**合并成一次** abort 调用（一次 IPC）。

故障隔离

某个 replica 进程崩溃（reader 收到 EngineDead）：把它标记为死亡，只让**绑定在它上面**的请求以错误结束；其他 replica 上的请求照常进行；之后的新请求不再分给它。所有 replica 都死了，`pick` 抛 `NoLiveReplica`。

文件： `mine/p04_stage_pool_abort.py` **验收：** `pytest tests/test_p04_stage_pool_abort.py` **提示：** A-12 **测试：** B-12 **参考解：** C-12

接口骨架

```
"""P04 · StagePool (sticky replica binding) + two-phase abort + dead-replica isolation.

StagePool
- pick(rid): sticky -- a request keeps its replica; new requests round-robin over LIVE
  replicas; raise NoLiveReplica if none.
- evict(replica): mark dead, drop and return the request ids bound to it.
- abort(ids): ONE abort call per replica with the ids bound to it; ids without a binding
  are ignored; bindings are released only after that replica's abort succeeded
  (if it raises, propagate and keep the bindings so the abort can be retried).

abort_request(pools, processor, rid)  -- two phase
  1. partial = processor.collect_partial(rid)        (read only)
  2. await pool.abort([rid]) for every pool          (any failure propagates; no commit)
  3. processor.commit_abort(rid)
  return OrchOutput(rid, partial, finished=True, error="abort")

handle_dead_replica(pool, replica) -> one terminal error OrchOutput per affected request;
requests bound to other replicas are untouched.
"""
from dataclasses import dataclass
from typing import Any, Protocol
```


© 2026 kzhou003

```python
# ------------------------------------------------------------ given types
class NoLiveReplica(Exception):
    pass


class ReplicaClient(Protocol):
    async def abort(self, request_ids: list[str]) -> None: ...


class PartialProcessor(Protocol):
    def collect_partial(self, request_id: str) -> Any: ...
    def commit_abort(self, request_id: str) -> None: ...


@dataclass
class OrchOutput:
    request_id: str
    data: Any
    finished: bool
    error: str | None = None


# ------------------------------------------------------------ implement below
class StagePool:
    def __init__(self, stage_id: int, replicas: list[ReplicaClient]) -> None:
        self.stage_id = stage_id
        self.replicas = replicas
        raise NotImplementedError

    def live_replicas(self) -> list[int]:
        raise NotImplementedError

    def pick(self, request_id: str) -> int:
        raise NotImplementedError

    def bound(self, request_id: str) -> int | None:
        raise NotImplementedError

    def release(self, request_id: str) -> None:
        raise NotImplementedError

    def evict(self, replica_id: int) -> list[str]:
        raise NotImplementedError

    async def abort(self, request_ids: list[str]) -> None:
        raise NotImplementedError


async def abort_request(pools: list[StagePool], processor: PartialProcessor, request_id: str) -> OrchOutput:
    raise NotImplementedError


def handle_dead_replica(pool: StagePool, replica_id: int) -> list[OrchOutput]:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ round-robin + 粘性
- ✓ evict 返回受影响请求，之后不再分配到死 replica；无存活时抛 NoLiveReplica
- ✓ abort 按 replica 分组，每个 replica 只调用一次
- ✓ 两阶段顺序：collect → 各 stage abort → commit
- ✓ abort 失败时不 commit、保留绑定以便重试
- ✓ 死 replica 只让绑定在它上面的请求失败

© 2026 kzhou003

面试官追问

- abort 和 ADD 在 EngineCore 内乱序会发生什么？（→ P10）
- 多个 final stage 时，abort 的终态消息应该由谁发、发几次?

直觉与类比

类比 · 行李寄存柜台（粘性绑定）

火车站有 3 个寄存柜台（replica）。你第一次来，工作人员按顺序把你分到某个柜台（round-robin），你的行李（KV cache）就存在**那个**柜台。之后你来续存、来取、来退，**必须回同一个柜台**——别的柜台根本没有你的东西。

新来的旅客才轮流分配；某个柜台关门了（evict），就只在开着的柜台之间轮。柜台关门时，存在那里的行李也没了：要通知在那儿存过东西的旅客（handle_dead_replica），并把登记本上这几行划掉——否则下次有人拿着旧登记去找，只会找到一扇关着的门。

类比 · 取消一张已经拆给好几家工厂的订单（两阶段 abort）

客户的订单被拆给了三家工厂（thinker / talker / code2wav 的 replica）各做一部分。现在客户要取消：

① **拍照**（`collect_partial`，只读）：先把已经做好的半成品拍下来，最后要发给客户；

② **通知每家工厂停工**（`await pool.abort([rid])`）：打电话可能打不通（IPC 失败）——**打不通就停下，别销账**；

③ **销账**（`commit_abort`，不可逆）：所有工厂都确认停工了，才把这张订单从账本上删掉；

最后给客户发一张"已取消"通知，附上半成品照片：`OrchOutput(rid, partial, finished=True, error="abort")`。

如果先销账再打电话、而电话没打通：工厂还在继续做，做好的货送回来，账本上却查无此单（孤儿输出）；你想重试，照片也找不回来了。

框架视角：T · SBWC · LIFE

	维度	P04 怎么答
T	Topology	全部跑在 orchestrator 的**同一个 asyncio 事件循环**里。replica 是别的进程，但被 `ReplicaClient.abort` 包成了可 await 的接口。没有线程，不需要锁。
S	State	每个池：`alive`（或死亡集合）、`bindings: dict[rid → replica]`、round-robin 计数器。orchestrator 侧：PartialProcessor 里每个请求的部分结果。单写者：都只在事件循环里被改。
B	Blocking	唯一的让出点是 `await replica.abort(ids)`。pick / bound / release / evict 都是同步方法，彼此不会交错。
W	Workers	本题没有新 worker：调用方是 P02 里的 request_handler（处理 abort 消息）和 dispatcher（发现 replica 死亡）。
C	Channels	**add 和 abort 走同一条控制队列**，所以对同一请求"先 add 后 abort"的顺序不会乱；每次 replica 调用是一条 IPC，按 replica 合并。
L	Lifecycle	一个绑定的一生：`pick` 创建 → 正常结束 `release` / abort 成功后释放 / `evict` 时删除。三条路必须都有，否则 `bindings` 只增不减。
I	Interface	`abort_request` 要么返回**恰好一条**终态 `error="abort"`，要么抛出异常且世界可重试；`handle_dead_replica` 对每个受影响请求**恰好一条**错误终态。
F	Flow + Invariants	① 绑定存在 ⇔ 那个 replica 上（可能）还有这个请求的状态；② 不可逆的 commit 永远排在所有可能失败的 abort 之后；③ `pick` 永不返回死 replica。

© 2026 kzhou003

E	Edges + Errors	无绑定的 id（从未派发到本 stage / 已释放 / 已 evict）→ 静默忽略；部分 replica 失败 → 已成功的释放、失败的保留；所有 replica 都死 → `NoLiveReplica`；`await` 期间绑定可能已被别处删除 → 释放时要容忍"已经没了"。

交互图：`nano-vllm-omni-drills/hints/p04_overview.html`

对照源码： `engine/stage_pool.py:1279-1339` `engine/stage_pool.py:333` `engine/orchestrator.py:1464-1515` `engine/orchestrator.py:645-731`

笔记 / 我踩到的坑：

© 2026 kzhou003

#13 · P05 RpcRouter：共享通道上的 RPC 关联 ⏱ 40 min 易 □ 完成 日期 _____

> sleep / wake_up / collective_rpc 的结果和其他控制消息挤在同一条回传队列上，多个线程同时在等各自的结果。

在系统中的位置

前端的 `collective_rpc`、`abort`、`sleep` 是同步 API：调用线程把请求放进 request 队列，然后阻塞等待结果。所有结果都从同一个 `rpc_output_queue` 回来，由一个 router 线程按 key 分发给等待者。

三个经典坑

1. **先发送后注册**：结果可能在注册之前就到了 → 被当成"没人要"丢掉 → 调用者永远等。必须**先注册再发送**。
2. **超时后不注销**：迟到的结果会塞进一个没人读的队列；同一个 key 再次使用时会读到旧结果。必须在 finally 里注销，且只注销自己的 waiter（用 `is` 比较）。
3. **致命错误时没有广播**：orchestrator 线程挂了，所有等待者都要立即收到错误，而不是各自等到超时；之后的新调用也应立即失败（锁存）。

接口

`register / unregister / deliver / fail_all / call(key, send, timeout)`，所有方法都可能被不同线程并发调用。每个 waiter 是 `queue.Queue(maxsize=1)`。

文件： `mine/p05_rpc_router.py` **验收：** `pytest tests/test_p05_rpc_router.py` **提示：** A-13 **测试：** B-13 **参考解：** C-13

接口骨架

```
"""P05 · RpcRouter: correlate results arriving on one shared channel with blocked callers.

- register(key) -> waiter (a queue.Queue(maxsize=1)); duplicate live key -> ValueError.
- deliver(key, result) -> True if a waiter got it, else False and self.late_results += 1.
- call(key, send, timeout): register FIRST, then send(), then wait; always unregister in
  finally; raise TimeoutError on timeout; if the result is an exception, raise it.
- fail_all(err): wake every current waiter with err and latch it: later register()/call()
  raise err immediately.
All methods are called from arbitrary threads.
"""
import queue
from collections.abc import Callable, Hashable
from typing import Any


class RpcRouter:
    def __init__(self) -> None:
        self.late_results = 0
        raise NotImplementedError

    def register(self, key: Hashable) -> queue.Queue:
        raise NotImplementedError

    def unregister(self, key: Hashable, waiter: queue.Queue) -> None:
        raise NotImplementedError

    def deliver(self, key: Hashable, result: Any) -> bool:
        raise NotImplementedError

    def fail_all(self, error: BaseException) -> None:
        raise NotImplementedError
```


© 2026 kzhou003

```
    def call(self, key: Hashable, send: Callable[[], None], timeout: float) -> Any:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 50 个线程并发调用、结果乱序回传，各自拿到自己的
- ✓ send() 内同步回传的结果不丢（先注册后发送）
- ✓ 超时 → TimeoutError；迟到结果被丢弃并计数；key 可复用
- ✓ 重复 key 报错
- ✓ fail_all 立刻唤醒所有等待者并锁存错误

面试官追问

- 为什么 orchestrator 内联执行 RPC 会阻塞后续请求准入？怎么改？

直觉与类比

类比 · 奶茶店的取餐器

点单时店员先给你一个**取餐器**（`register(key)` 返回的 `Queue(maxsize=1)`），**然后**才把单子递进后厨（`send()`）。后厨做好哪杯就呼叫哪个号（`deliver(key, result)`），出餐顺序和点单顺序无关。一个取餐器只装一杯（maxsize=1）。

① 为什么要先领取餐器再下单？后厨手快，单子刚递进去饮料就好了——这时你手上还没有取餐器，店员只好把这杯放到"无人认领"（`late_results += 1`），你在柜台前干等。

② 你等不及走了（超时），**一定要把取餐器还回去**（`finally: unregister`）。之后做好的那杯只能进"无人认领"。

③ 还取餐器时，你还的是**你手上那一个**（`unregister(key, waiter)` 用 `is` 比较），而不是"把 17 号作废"——17 号可能已经发给了下一位客人。

类比 · 后厨着火（fail_all）

后厨着火了：店员不是让每个客人各自等到不耐烦，而是**立刻广播**"所有取餐器作废，请离开"（把错误放进每个在等的信箱），并且在门口**挂上"暂停营业"的牌子**（锁存 latch）：之后来的客人连单都不用下，进门就被告知（`register / call` 立即抛出这个错误，`send` 根本不会被调用）。

框架视角：T · SBWC · LIFE

	维度	P05 怎么答
T	Topology	**纯线程**：N 个调用方线程 + 1 个结果通道线程（真实系统里是 router 自己的守护线程）+ 偶尔调用 fail_all 的线程。没有事件循环，所以同步原语是 `threading.Lock` 和 `queue.Queue`。
S	State	`waiters: dict[key → Queue(1)]`、锁存的 `fatal`、`late_results`。三者被多个线程读改写 → **同一把锁**保护；"检查 fatal + 插入 waiter"必须在同一次加锁里。
B	Blocking	调用方阻塞在 `waiter.get(timeout)` ——**绝不能持锁**；deliver / fail_all 往 waiter 里 put 也放在锁外（容量 1、只 put 一次，不会阻塞）。
W	Workers	调用方：登记、发送、等待、注销自己的；结果线程：只按 key 取出 waiter 并交付，不看结果内容；fail_all：清空并广播。
C	Channels	出：`send()`（真实系统里是 request_queue）；回：一条共享结果通道；每个调用一个**私有**的 `Queue(maxsize=1)` 当信箱。
L	Lifecycle	一个 waiter 的一生：register 创建 → 被 deliver 取走 / 被 fail_all 清空 / 被 finally 里的 unregister 删除，三条路有且只有一条真正把它从表里摘掉。router 的一生：正常 → fail_all 之后永久"锁死"。

© 2026 kzhou003

I	Interface	`call` 只有四种结局：返回结果 / raise 结果里的异常 / `TimeoutError` / raise 锁存的致命错误；无论哪种，返回时表里都不再有它的 waiter。
F	Flow + Invariants	① 同一 key 最多一个活着的 waiter；② 每个 waiter 最多收到一个结果（deliver 用 pop，且与 fail_all 互斥）；③ fatal 一旦设置，表里不会再出现新的 waiter；④ 登记一定先于发送。
E	Edges + Errors	迟到 / 未知 key 的结果 → 计数丢弃；重复 key → ValueError；`send()` 自己抛错 → finally 照样注销；结果在"已 pop、调用方刚好超时"的缝里到达 → 落进没人读的信箱被悄悄丢弃（可以接受，它不算 late）。

交互图：`nano-vllm-omni-drills/hints/p05_overview.html`

对照源码： `engine/rpc_result_router.py:32-150` `engine/messages.py:125-148`

笔记 / 我踩到的坑：

© 2026 kzhou003

Track B · Stage 进程与调度

EngineCore busy loop、停靠/归还、continuous batching、paged KV、采样、attention。

© 2026 kzhou003

#14 · P10 EngineCore busy loop ⏱ 45 min 中

□ 完成 日期 _____

> 每个 stage 进程里，IO 线程把 ZMQ 消息塞进 input_queue，主线程跑 schedule→execute→update。要求：空闲时零 CPU、ADD/ABORT 严格 FIFO、信号能叫醒阻塞的 loop、step 异常是致命的。

在系统中的位置

stage 子进程 = 上游 vLLM 的 `EngineCoreProc`：输入 IO 线程（ZMQ → `input_queue`）、主 busy loop、输出 IO 线程（`output_queue` → ZMQ）。vllm-omni 没有改 busy loop 本身，但它的很多正确性依赖 busy loop 的语义。

```
while not shutdown:
    if not engine.has_work():  msg = input_queue.get()   # 阻塞：空闲时零 CPU
    抽干 input_queue（FIFO）：ADD / ABORT / UTILITY / WAKEUP
    if shutdown: break
    if engine.has_work():  for out in engine.step(): emit(("output", out))
```

需求细节

- **FIFO**：ADD r1 → ABORT r1 → ADD r2 必须按此顺序作用于引擎；被 abort 的请求一个输出都不能有。
- **可唤醒的关停**：signal handler 或其他线程调用 `request_shutdown()`：设置标志并往队列放 WAKEUP，阻塞在 get() 的 loop 立即醒来退出，发出 `("shutdown",)`。
- **UTILITY**（sleep/wake_up/collective_rpc 等）：结果按 call_id 返回；utility 抛异常只影响这次调用，引擎继续运行。
- **致命错误**：step() 抛异常（例如 CUDA 非法访存）→ 发出 `("fatal", repr(e))` 并退出，由外层的进程监控把它报告为 stage 死亡（P04）。

文件： `mine/p10_engine_core_loop.py` **验收：** `pytest tests/test_p10_engine_core_loop.py` **提示：** A-14 **测试：** B-14 **参考解：** C-14

接口骨架

```
"""P10 · The EngineCore busy loop (what runs inside every stage process).

IO threads (not part of this drill) push Msg objects into self.input_queue and ship
whatever you pass to emit(). Implement run_busy_loop() (it runs on its own thread):
- While the engine has no work, BLOCK on input_queue (no polling / spinning).
- Before every step, drain all queued messages in FIFO order:
    ADD     payload (req_id, data)          -> engine.add
    ABORT   payload [req_ids]               -> engine.abort
    UTILITY payload (call_id, name, args)   -> emit(("utility", call_id, result, None)),
            or emit(("utility", call_id, None, repr(exc))) if it raised (engine keeps running)
    WAKEUP  no-op (only unblocks the queue)
- If the engine has work: for out in engine.step(): emit(("output", out)).
- request_shutdown() may be called from any thread / a signal handler: the loop must exit
  promptly even when blocked, then emit(("shutdown",)).
- An exception from engine.step() is fatal: emit(("fatal", repr(exc))) and return.
"""
import queue
from collections.abc import Callable
from dataclasses import dataclass
from typing import Any, Literal, Protocol


# ---------------------------------------------------------------- given types
@dataclass
class Msg:
    kind: Literal["ADD", "ABORT", "UTILITY", "WAKEUP"]
    payload: Any = None
```


© 2026 kzhou003

```
class Engine(Protocol):
    def add(self, req_id: str, data: Any) -> None: ...
    def abort(self, req_ids: list[str]) -> None: ...
    def has_work(self) -> bool: ...
    def step(self) -> list[Any]: ...
    def utility(self, name: str, args: tuple) -> Any: ...


# ------------------------------------------------------------ implement below
class EngineCoreLoop:
    def __init__(self, engine: Engine, emit: Callable[[tuple], None]) -> None:
        self.engine = engine
        self.emit = emit
        self.input_queue: queue.Queue[Msg] = queue.Queue()

    def request_shutdown(self) -> None:
        raise NotImplementedError

    def run_busy_loop(self) -> None:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 空闲 0.3s：step 0 次，has_work 调用 < 10 次（没有自旋）
- ✓ ADD 后跑到完成
- ✓ ADD→ABORT→ADD 预先排队：引擎调用顺序严格 FIFO，被 abort 的请求零输出
- ✓ 另一线程 request_shutdown：0.5s 内退出并只发出 ('shutdown',)
- ✓ step 抛异常 → ('fatal', ...) 并退出
- ✓ utility 结果/异常按 call_id 返回且不杀引擎

面试官追问

- vllm-omni 为什么要在 ABORT 走完 FIFO 之后才释放 overflowed 请求?
- max_concurrent_batches>1 时 step_with_batch_queue 在做什么?

直觉与类比

类比 · 主厨和订单夹

主厨（busy loop）每做一"轮"菜（`step()`），就给锅里**所有**在做的菜各推进一步。服务员（输入 IO 线程）把单子按顺序夹在订单夹上（`input_queue`）。

① **每轮开火之前**，主厨把订单夹上**已有的单子全部**按顺序处理掉：新菜、退菜、顾客的问题（ADD / ABORT / UTILITY）——不是一轮只看一张。

② 锅里没菜时，主厨就坐在打单机旁边**打盹**（阻塞 `get()`）：不来回走动查看（不轮询），打单机一响就醒。

③ 打烊：经理既要挂上"打烊"牌子（设置 shutdown 标志），**还要打一张空白单**（WAKEUP），否则打盹的主厨永远看不到牌子。

④ 顾客问"还有多少库存"而主厨答不上来（utility 抛异常）——告诉这位顾客"查不到"即可，厨房照常；灶台着火（`step()` 抛异常）——拉响警报（`("fatal", ...)`）然后撤离，不再接单。

类比 · 退菜单必须排在下一轮开火之前

订单夹上依次是："加一份宫保鸡丁（ADD q2)""退掉宫保鸡丁（ABORT q2)"。如果主厨一轮只看一张单子，他会先下锅炒一轮宫保鸡丁，下一轮才看到退单——这一轮炒出来的东西（一个输出）已经端出去了。**先抽干订单夹、再开火**，被退掉的菜就一口都不会出锅。

© 2026 kzhou003

框架视角：T · SBWC · LIFE

	维度	P10 怎么答
T	Topology	一个 stage 进程里：输入 IO 线程、busy loop 线程、输出 IO 线程，外加信号 / 任意线程调用 `request_shutdown`。**没有事件循环**：线程之间只用 `queue.Queue` 和 `threading.Event`。
S	State	引擎（scheduler + KV + 模型）**只由 busy loop 一个线程碰**：单写者，不用锁。跨线程共享的只有 `input_queue`（自带锁）和 shutdown 标志（`threading.Event`）。
B	Blocking	唯一允许阻塞的地方：**没活时**的 `input_queue.get()`（不带 timeout）。有活时只用 `get_nowait()` 抽干，绝不阻塞。
W	Workers	busy loop：取消息、按顺序作用于引擎、step、emit。IO 线程只搬运，不碰引擎。`request_shutdown` 只设标志 + 放 WAKEUP，不 emit、不 join。
C	Channels	入：**一条** FIFO `input_queue`（ADD / ABORT / UTILITY / WAKEUP 共用，所以顺序天然保持）；出：`emit`（真实系统里是 output_queue → 输出 IO 线程）。UTILITY 的结果靠 `call_id` 让对端关联（→ P05 的思路）。
L	Lifecycle	启动 → 运行（阻塞 / 抽干 / step 循环）→ 两种结局：收到 shutdown → `("shutdown",)`；step 抛异常 → `("fatal", ...)` 后返回。二者互斥，每次运行恰好一个。
I	Interface	emit 的元组形状是协议，要精确（`("shutdown",)` 是单元素元组）。
F	Flow + Invariants	① 每次 step 之前，step 开始前已入队的消息全部处理完；② 消息按入队顺序作用于引擎；③ 引擎有活 ⇒ 不阻塞，引擎没活 ⇒ 不自旋；④ shutdown 标志一旦被看到，就不再 step。
E	Edges + Errors	utility 异常 → 只回这一次调用的错误；step / add 异常 → 致命；处理完消息后引擎可能仍没活（只来了 WAKEUP / UTILITY / ABORT）→ 不 step；阻塞 get 拿到的那条也是正常消息，必须处理。

交互图：`nano-vllm-omni-drills/hints/p10_overview.html`

对照源码： `engine/stage_engine_core_proc.py:133-177` `engine/stage_engine_core_proc.py:147-154` `engine/stage_engine_core_proc.py:311-345` `vllm/v1/engine/core.py`

笔记 / 我踩到的坑：

© 2026 kzhou003

#15 · P11 WAITING_FOR_CHUNK：停靠与归还 ⏱ 45 min 中 □ 完成 日期 _____

> 上游 scheduler 不认识“等 chunk”这种状态。你要在它外面包一层：没数据的请求在它眼里必须不存在。

在系统中的位置

async_chunk 下，talker 的请求在预热后就进入了 scheduler，但它的输入是 thinker 一块一块发来的。在下一块到达之前，这个请求**不能被调度**：调度它会占 token budget 和 KV，而 forward 时根本没有输入。vllm-omni 的做法是给 `RequestStatus` 加一个负值状态 `WAITING_FOR_CHUNK`，并在每次 `schedule()` 前后加钩子。

```
schedule():
    ① 把已就绪的停靠请求放回原队列
    ② 把 needs_chunk 且未就绪的请求从 waiting/running 中移出，状态设为 WAITING_FOR_CHUNK，
       记录来源队列和原状态，第一次停靠时调用 source.load_async(id)
    try:     ③ base.schedule()          # 上游只看得到有数据的请求
    finally: ④ 再归还一次已就绪的        # 即使上游抛异常也要归还
```

为什么“移出”而不是“跳过”

上游 scheduler 的代码我们不想改（vllm-omni 的不变量 AR-INV-001：vLLM 拥有基础调度语义）。题目里的 `BaseScheduler` 若看到 WAITING_FOR_CHUNK 或 ABORTED 状态的请求会直接断言失败，模拟上游对未知状态的抵触。

其他要求

- 归还到**来源**队列并恢复原状态；同时归还的多个请求按到达顺序排列。
- `abort(id)` 任何时候都可以调用：不管是在停靠中还是在队列里，之后都不能再被调度或归还。

文件： `mine/p11_chunk_aware_scheduler.py` **验收：** `pytest tests/test_p11_chunk_aware_scheduler.py`
提示： A-15 **测试：** B-15 **参考解：** C-15

接口骨架

```
"""P11 · Park / restore around an upstream scheduler that knows nothing about chunks.

A request with needs_chunk=True receives its input from an upstream stage. Whenever its
next chunk is not ready (source.is_ready(id) is False) it must be invisible to
BaseScheduler.schedule():
- schedule(): (a) restore parked requests that became ready, (b) park every not-ready
  needs_chunk request from base.waiting / base.running (status "WAITING_FOR_CHUNK",
  remember origin queue and previous status, call source.load_async(id) once per parking),
  (c) call base.schedule() and (d) in a `finally`, restore any parked request that is ready
  -- this must also happen when base.schedule() raises.
- Restoring puts a request back into its ORIGIN queue with its previous status, in
  arrival order among the requests restored together.
- abort(id) works at any time: parked or queued, the request disappears and is never
  scheduled or restored again.
"""
from collections import deque
from dataclasses import dataclass
from typing import Protocol


# ---------------------------------------------------------------- given types
@dataclass
class Req:
    req_id: str
    arrival: int
    needs_chunk: bool = False
    status: str = "WAITING"
```


© 2026 kzhou003

```
class BaseScheduler:
    """Stand-in for the upstream vLLM scheduler: knows nothing about chunks."""

    def __init__(self) -> None:
        self.waiting: deque[Req] = deque()
        self.running: list[Req] = []

    def schedule(self) -> list[str]:
        for r in self.running + list(self.waiting):
            if r.status in ("WAITING_FOR_CHUNK", "ABORTED"):
                raise AssertionError(f"upstream saw {r.req_id} in state {r.status}")
        out = [r.req_id for r in self.running] + [r.req_id for r in self.waiting]
        for r in self.waiting:
            r.status = "RUNNING"
        self.running += list(self.waiting)
        self.waiting.clear()
        return out


class ChunkSource(Protocol):
    def load_async(self, req_id: str) -> None: ...
    def is_ready(self, req_id: str) -> bool: ...


# ---------------------------------------------------------- implement below
class ChunkAwareScheduler:
    def __init__(self, base: BaseScheduler, source: ChunkSource) -> None:
        self.base = base
        self.source = source
        self.parked: dict[str, tuple[Req, str, str]] = {}  # id -> (req, origin, previous status)

    def add(self, req: Req) -> None:
        self.base.waiting.append(req)

    def abort(self, req_id: str) -> None:
        raise NotImplementedError

    def schedule(self) -> list[str]:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 未就绪请求从不被调度，load_async 只调用一次
- ✓ 就绪后下一次 schedule 一定被调度
- ✓ running 中的请求等下一个 chunk 时同样被停靠
- ✓ 上游 schedule 抛异常时 finally 仍然归还
- ✓ 停靠期间 abort：不会再被调度/归还
- ✓ 同时归还多个请求时保持到达顺序

面试官追问

- 为什么 vllm-omni 不在 step 中途调用 waiting_queue.prepend_requests?
- recv 线程拿到数据后能直接改 request.prompt_token_ids 吗?

© 2026 kzhou003

直觉与类比

> **类比 · 后厨的订单板和“等料夹”**
>
> 主厨（**上游 scheduler**）只认墙上的订单板：“待做”一栏（waiting）、“在做”一栏（running）。板上的单子他**每张都会做**，看到一张写着“等料”的单子会直接发火（AssertionError）。
>
> 有些菜要等隔壁厨房（**thinker**）送来的半成品。你是领班（**ChunkAwareScheduler**）：每轮开工前，把“料还没到”的单子**从板上摘下来**，夹进旁边的**等料夹**（`parked`），并在单子背面写两行字：**原来挂在哪一栏**（origin）、**原来是什么状态**（previous status）；同时给隔壁打个电话“这单的料请送来”（`load_async`，每摘一次打一次）。
>
> 料到了，就把单子**挂回原来那一栏**、恢复原状态。好几张单子的料同时到时，按客人**下单的先后**（`Req.arrival`）挂回去，而不是按它们被夹进夹子的顺序。

> **类比 · 主厨摔锅、客人退单**
>
> **主厨摔锅**（`base.schedule()` 抛异常）：不管今天这一轮出了什么事，领班下班前都要把等料夹里“料已经到了”的单子挂回板上（`finally`），然后再把事故照实上报（异常继续往上抛，不能吞掉）。
>
> **客人退单**（`abort(id)`）：这张单子此刻可能在**三个**地方——待做栏、在做栏、等料夹。三处都要找、都要撕掉；只撕等料夹里的，板上那张主厨照做；只在单子上写“已退”却不摘下来，主厨看到又要发火。撕掉之后，隔壁厨房就算把料送来了，也没有单子可以挂回去。

框架视角 · 用 T · SBWC · LIFE 看 P11

> P11 是 P11–P13 三道题里**唯一完全单线程**的一道：`add / abort / schedule` 都在 stage 进程的同一个调度线程里调用。它的“并发”只来自一件事——`is_ready()` 的答案由**别的线程**（P13 的 recv 线程）推进，你每次问都可能得到不同的结果。

	问题	P11
T	跑在哪些进程 / 线程里？边界在哪？	下游 stage 进程里的调度线程（EngineCore busy loop，见 P10）。数据由别的进程（thinker）经 P12 的共享内存送来、由本进程的 P13 recv 线程收下；P11 只跨“调度线程 ↔ ChunkSource”这一条边界，而且只用两个**非阻塞**调用。
S	共享状态？谁写？	`parked`、`base.waiting / base.running`、`req.status`：全部只由调度线程写（单写者，不用锁）。上游 scheduler 也会改队列，但它只在 (c) 里、和你串行执行。
B	哪里会干等？	**哪里都不许等。**`schedule()` 绝不能阻塞等 chunk（那会让整张 GPU 空转、拖住同 batch 的所有请求）——所以是“停靠 + 下一轮再问”，而不是“等到它来”。
W	谁做什么？	没有自己的后台 worker。领班（你）只负责摘单、挂单；送料的“搬运工”是 P13 的 recv 线程。
C	通道？	`ChunkSource` 就是通道：`load_async(id)` 是请求，`is_ready(id)` 是回答。
L	生命周期？	一个 needs_chunk 请求会在 `WAITING/RUNNING ⇄ WAITING_FOR_CHUNK` 之间来回很多次：每消费一块，就可能再停靠一次。终点是完成或 `ABORTED`。
I	对外接口？	`add` / `abort` / `schedule`；`abort` 任意时刻可调用。
F	不变量？	① 上游 (c) 执行期间，队列里没有 `WAITING_FOR_CHUNK` / `ABORTED` 的请求；② 每个请求任一时刻**最多在一处**：waiting、running、parked 之一；③ 每次停靠恰好一次 `load_async`；④ 归还 = 原队列 + 原状态，同一批按 arrival 排序。
E	边界与故障？	上游抛异常（finally 仍归还、异常照抛）；chunk 恰好在 (c) 期间到达；同一个请求先从 waiting、后从 running 停靠；abort 后 `is_ready` 仍返回 True。

交互图：`nano-vllm-omni-drills/hints/p11_overview.html`

© 2026 kzhou003

对照源码： `core/sched/omni_ar_scheduler.py:408-451`

`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1120-1186`

`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1488-1532` `patch.py:364-372`

笔记 / 我踩到的坑：

© 2026 kzhou003

#16 · P07 Scheduler：continuous batching + chunked prefill + 抢占 ⏱ 75 min 难

□ 完成 日期 _____

> nano-vllm 的核心。token budget、并发上限、KV block 上限三重约束下，每步决定谁跑、跑几个 token。

在系统中的位置

每个 LLM stage 进程的 busy loop 每步调用一次 `schedule()`，输出"本步每个序列算几个 token"，交给 model runner；runner 算完后 `update()` 把采样结果写回。

三重约束

- `max_num_batched_tokens`：本步所有序列的 token 总数上限（决定激活显存和单步延迟）；
- `max_num_seqs`：同时 running 的序列数上限；
- KV block：持有 n 个 token（已算 + 本步要算）的序列需要 `ceil(n / block_size)` 个 block，总数不超过 `num_blocks`。

调度规则

```
budget = max_num_batched_tokens
1) running（按顺序）：n = min(剩余未算 token, budget)     # decode 时为 1；chunked prefill 未完时更多
   block 不够 → 抢占 running 中最后一个（最晚加入）：释放 block、num_computed=0、放回 waiting 队首
              → 重试；如果被抢占的就是它自己，则跳过它
2) waiting（严格 FCFS）：len(running) < max_num_seqs 且 budget > 0 时，n = min(剩余, budget)
   第一个 block 放不下的请求就停止（后面的不许插队）
```

update

对每个被调度的序列，`num_computed += n`；只有当它算完了全部 token（prefill 最后一块或 decode）时才调用 `sample_fn` 追加一个 token；遇到 eos 或达到 max_tokens 就结束并释放 block。被抢占的序列下次重新计算 prompt + 已生成的 token（recompute），已生成的输出保留不变。

例子（budget=16，三个 prompt 长 10 的序列）

```
step1: s0:10  s1:6          # s1 只做了一部分 prefill
step2: s0:1   s1:4  s2:10   # s0 decode；s1 prefill 剩 4；s2 整段
step3: s0:1   s1:1  s2:1
```

文件： `mine/p07_scheduler.py` **验收：** `pytest tests/test_p07_scheduler.py` **提示：** A-16 **测试：** B-16
参考解： C-16

接口骨架

```
"""P07 · Scheduler: continuous batching + chunked prefill + recompute preemption.

KV accounting is in blocks: a sequence holding n computed+scheduled tokens needs
ceil(n / block_size) blocks; self.blocks_used must never exceed num_blocks.

schedule() -> {seq_id: num_tokens_this_step}, total <= max_num_batched_tokens:
  1. RUNNING sequences first, in order: n = min(remaining tokens, budget)
     (1 for decode, more while a chunked prefill is still in progress). If the blocks do
     not fit, preempt the LAST running sequence (free its blocks, num_computed = 0, put it
     at the FRONT of waiting, num_preemptions += 1) and retry; if the victim is the
     sequence itself, skip it.
  2. WAITING sequences strictly FCFS while len(running) < max_num_seqs and budget > 0:
     n = min(remaining, budget); stop at the first one whose blocks do not fit.
update(sample_fn, eos): for each scheduled seq add n to num_computed; if it now covers all
  tokens, append sample_fn(seq_id) to output; finish on eos or max_tokens (free blocks).
```


github.com/kzhou003/nano-vllm-omni-drills © 2026 kzhou003

```
A preempted sequence recomputes prompt + already generated tokens; outputs are kept.
"""
from collections import deque
from collections.abc import Callable
from dataclasses import dataclass, field


@dataclass
class Seq:
    seq_id: str
    prompt_len: int
    max_tokens: int
    output: list[int] = field(default_factory=list)
    num_computed: int = 0
    num_blocks: int = 0
    status: str = "WAITING"  # WAITING | RUNNING | FINISHED

    @property
    def num_tokens(self) -> int:
        return self.prompt_len + len(self.output)


class Scheduler:
    def __init__(self, num_blocks: int, block_size: int, max_num_seqs: int, max_num_batched_tokens: int) -> None:
        self.num_blocks = num_blocks
        self.block_size = block_size
        self.max_num_seqs = max_num_seqs
        self.max_num_batched_tokens = max_num_batched_tokens
        self.waiting: deque[Seq] = deque()
        self.running: list[Seq] = []
        self.finished: list[Seq] = []
        self.blocks_used = 0
        self.num_preemptions = 0

    def add(self, seq: Seq) -> None:
        self.waiting.append(seq)

    def has_work(self) -> bool:
        return bool(self.waiting or self.running)

    def schedule(self) -> dict[str, int]:
        raise NotImplementedError

    def update(self, sample_fn: Callable[[str], int], eos_token: int) -> list[Seq]:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ token budget 与 chunked prefill 的精确调度结果
- ✓ max_num_seqs 限制
- ✓ FCFS：队首放不下时后面的小请求也不能插队
- ✓ KV 极小时发生抢占，所有请求最终正确完成且输出不变
- ✓ EOS 提前结束并释放 block

面试官追问

- recompute 抢占 vs swap 抢占的取舍?
- 多模态请求的 encoder 计算如何计入 budget?
- async scheduling 下 num_computed 里包含了 placeholder token，update 要怎么改?

© 2026 kzhou003

直觉与类比

> **类比 · 餐厅的"每轮上菜"调度**
> **waiting 队列** = 门口排队的客人（严格按到店顺序，不许插队）；**running** = 已经坐下的桌子；
> **max_num_seqs** = 一共几张桌子；
> **KV block** = 桌面面积：客人越吃越久（生成越多 token），餐具堆得越多，面积只增不减，直到他走；
> **token budget**（`max_num_batched_tokens`）= 厨房**每一轮**能出多少个菜。一桌 10 人的大团（长 prompt）如果一轮出不完，就先出一部分——这就是 **chunked prefill**；
> **抢占** = 桌面实在不够时，请**最后坐下**的那桌先回到队伍**最前面**等；他们已经吃掉的（已生成的 token）不用吐出来，但桌上的餐具被收走了（KV 被释放），回来时得**重新摆一遍**（recompute）。

> **类比 · 抢占 = 把最后入场的人请回队首，而不是赶出剧院**
> 抢占**不是**取消请求：它丢掉的只是**缓存**（KV），不是**进度**（已生成的 token）。
> 就像看电影中途被请出去重新安检：你**看过的剧情还记得**（output 保留），但座位没了，回来得重新找位置、把前面的剧情**快进重放一遍**（recompute `prompt + 已生成` 的全部 token）。
> 所以用户**完全看不出**发生过抢占：最终输出与从不抢占时逐 token 相同，只是慢了一点。这正是这道题最容易写错、也最值得自己验证的性质。

框架视角：两个阶段 · 三本预算 · 六条不变量

P07 是单线程的纯状态机（没有并发，T·SBWC·LIFE 用不上），难点是**预算记账**和**队列搬动**必须同时正确。 把"每一笔预算在哪里扣、什么时候退"写成表，抢占那段的分支就不会乱：

预算	谁持有 / 怎么计量	扣（什么时候减）	退（什么时候加回）
token budget	局部变量，每次 `schedule()` 重置为 `max_num_batched_tokens`	每成功调度一个序列：`budget -= n`，其中 `n = min(num_tokens - num_computed, budget)`	本步不退（被抢占者本步也没跑）。**下一步重新满额**
running 槽位	`len(self.running)`，上限 `max_num_seqs`	阶段 2 接纳一个就 append 到 running **末尾**（末尾 = 最低优先级 = 抢占首选）	抢占时 `running.remove`；结束时 `running.remove`。两者都让阶段 2 多一个名额
KV block	`blocks_used` 与每个 `seq.num_blocks` 两处同时记	`_try_grow(seq, num_computed + n)`：算出增量 `need`，**放得下才同时改两处**，否则一点不改并返回 False	抢占：`blocks_used -= seq.num_blocks; seq.num_blocks = 0`；结束：同样释放

两个阶段的骨架责任

阶段	遍历什么	每个序列的 n	放不下怎么办	终止条件
阶段 1 running	`self.running`，按下标 `while`（**不能用 for**：抢占会改这个列表）	`min(num_tokens - num_computed, budget)`：decode 是 1，chunked prefill 没做完就是剩下的那一段	抢占 `running[-1]` 然后**重试同一个序列**；受害者是自己 → 跳过、下标不前进	下标到末尾，或 `budget == 0`
阶段 2 waiting	`self.waiting[0]`（只看队首，FCFS）	同上公式（被抢占过的序列这里会算出 `prompt + 已生成`）	**break**，不许跳过它去调度后面的	`waiting` 空 / `budget == 0` / `len(running) == max_num_seqs`

© 2026 kzhou003

不变量（可以直接写成 assert，每步跑一遍）

	不变量	违反时的症状
I1	`blocks_used == sum(s.num_blocks for s in running)` 且 `<= num_blocks`；所有序列结束后 `blocks_used == 0`	block 泄漏：跑到后面谁都调度不进来，或者超发导致真实系统 OOM
I2	`sum(schedule().values()) <= max_num_batched_tokens`，且每个 `n >= 1`	前向的 batch 超过预算 → 激活显存爆掉；`n == 0` 的序列白占一行
I3	对任意序列：`0 <= num_computed <= num_tokens`；只有 `num_computed == num_tokens` 时才允许采样	chunked prefill 中途就采样 → 多出 token、提前达到 max_tokens、输出变样
I4	抢占只重置 `num_computed`、`num_blocks`、`status`；`output` **必须保留**	最终输出与无抢占时不同（重复生成、或内容漂移）——这是最难查的一类 bug
I5	被抢占者进 `waiting` **队首**；`waiting` 内部始终是到达顺序	被抢占的请求饿死；压力下请求顺序乱掉，延迟长尾爆炸
I6	每个序列同时只出现在 `waiting`、`running`、`finished` 之一里	同一个请求被调度两次（一次前向里出现两行），或结束后仍被调度

易错点清单

- **用 `num_tokens` 算 block**：chunked prefill 只算了一部分，却按整个 prompt 要块 → 提前占满、无谓抢占。要用 `num_computed + n`。
- **抢占后忘了 `num_computed = 0`**：KV 随 block 释放掉了，却只调度 1 个 token → 模型读到的是别人的 KV（而且不会报错）。
- **阶段 2 用 `continue` 跳过放不下的队首**：违反 FCFS，大请求永远饿死。
- **遍历 running 用 `for`**：抢占时 `running.pop()` 会让迭代器跳过元素或访问到已被抢占的序列。
- **update 不知道本步调度了谁**：`update` 没有参数，必须把 `schedule()` 的结果存进实例字段（返回给调用方的最好是副本，防止别人改你的账本）。
- **结束判定顺序**：采样出的 token 先 append 进 `output`，再判 `tok == eos or len(output) >= max_tokens`——eos 本身也是输出的一部分。

交互图：`nano-vllm-omni-drills/hints/p07_overview.html`

对照源码： `vllm/v1/core/sched/scheduler.py`

笔记 / 我踩到的坑：

© 2026 kzhou003

#17 · P06 BlockManager：paged KV + 前缀缓存 75 min 难 □ 完成 日期 _____

> PagedAttention 的内存管理层：引用计数、哈希链前缀匹配、LRU 可驱逐池、原子分配。

在系统中的位置

KV cache 被切成固定大小的物理 block（例如 16 个 token）。每个序列有一张 **block table**：逻辑块 i → 物理块号。attention kernel 用 `slot = table[pos // bs] * bs + pos % bs` 找到 token 的 K/V。

前缀缓存

同一个 system prompt、同一张图的多轮对话，前缀 KV 完全相同。**满**的 block 用哈希链标识：`h_i = hash((h_{i-1}, tuple(block_i 的 token)))`（链式哈希保证"相同内容但前缀不同"的块不会误命中）。分配时从头开始逐块匹配，命中就共享物理块（引用计数 +1）。

```
prompt: [0..9] (bs=4)   → 块 [0,1,2,3] [4,5,6,7] [8,9,_,_]
可匹配块数 = (len - 1) // bs = 2     # 至少留 1 个 token 要算（否则拿不到下一个 token 的 logits）
num_cached_tokens = 命中块数 × bs
```

释放与驱逐

- free 时**逆序**减引用；归零的块：有哈希 → 进入 LRU 可驱逐池（仍能被命中复用）；无哈希 → 回空闲表。
- 需要新块时：优先空闲表，其次驱逐 LRU 最旧的（同时删除它的哈希登记）。逆序释放让尾部块先被驱逐。
- `num_free_blocks` = 空闲 + 可驱逐。

原子性

allocate / append 若因 block 不足失败，必须**不改变任何状态**（先检查后修改）。注意：命中可驱逐池里的块也会占用"可用"名额。

append 填满一个块

decode 让最后一个块变满时，也要为它计算哈希并登记，这样后续请求可以命中模型生成的内容（多轮对话）。

文件： `mine/p06_block_manager.py` **验收：** `pytest tests/test_p06_block_manager.py` **提示：** A-17 **测试：** B-17 **参考解：** C-17

接口骨架

```
"""P06 · BlockManager: paged KV-cache allocator with ref counts and prefix caching.

- Physical blocks 0..num_blocks-1 of block_size token slots each.
- allocate(seq, tokens): all-or-nothing (raise OutOfBlocks without changing state).
  With prefix caching, FULL blocks are identified by a hash chain
  h_i = hash((h_{i-1}, tuple(block_i_tokens))). Reuse matching blocks from the start of the
  prompt, but only among the first (len(tokens)-1)//block_size blocks, so at least one
  token is always computed. num_cached_tokens(seq) = reused blocks * block_size.
- append_slot(seq, token): returns the slot (block*block_size + offset) of the new token;
  allocates a block when needed (OutOfBlocks leaves state unchanged); when a block becomes
  full it is registered in the prefix cache.
- free(seq): decrement refs (walk the table in reverse). A block whose ref hits 0 goes
  back to the free list, or -- if it carries a hash -- to an LRU "evictable" pool that
  still serves cache hits until it is reused for a new allocation.
- num_free_blocks counts free + evictable blocks.
"""
```


© 2026 kzhou003

```
class OutOfBlocks(Exception):
    pass


class BlockManager:
    def __init__(self, num_blocks: int, block_size: int, enable_prefix_cache: bool = False) -> None:
        self.num_blocks = num_blocks
        self.block_size = block_size
        self.enable_prefix_cache = enable_prefix_cache
        raise NotImplementedError

    @property
    def num_free_blocks(self) -> int:
        raise NotImplementedError

    def block_table(self, seq_id: str) -> list[int]:
        raise NotImplementedError

    def num_cached_tokens(self, seq_id: str) -> int:
        raise NotImplementedError

    def slot(self, seq_id: str, pos: int) -> int:
        raise NotImplementedError

    def can_allocate(self, num_tokens: int) -> bool:
        raise NotImplementedError

    def allocate(self, seq_id: str, token_ids: list[int]) -> list[int]:
        raise NotImplementedError

    def append_slot(self, seq_id: str, token_id: int) -> int:
        raise NotImplementedError

    def free(self, seq_id: str) -> None:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ slot 映射
- ✓ append 跨 block 边界
- ✓ OutOfBlocks 原子性（allocate/append 失败不改状态）
- ✓ free 全部归还
- ✓ 前缀共享与引用计数
- ✓ free 后缓存仍可命中；最后一个 token 永不命中
- ✓ 驱逐后不再命中
- ✓ append 填满的 block 也进入缓存

面试官追问

- 多模态 prompt 的 block hash 必须包含什么?
- 多个 stage 共卡时，每个 stage 的 num_blocks 怎么定?

© 2026 kzhou003

直觉与类比

> **类比 · 酒店客房与房卡清单**
> 物理 block = **一间四人间客房**（`block_size=4` 张床 = 4 个 token slot）；`num_blocks` = 酒店一共几间房，开业时就固定了，不能临时盖楼。
> 一个请求（序列）= 一个旅行团。前台给它一张 **房卡清单**（block table）：第 1 批客人住 2 号房、第 2 批住 7 号房……房间号**完全不需要连号**，这正是 paged 的意义——不用为"一个长序列必须住连续套房"而预留整层楼。
> `slot(seq, pos)` 就是"第 pos 号客人睡哪张床"：`房号 × 每间床数 + 床位号`。

> **类比 · 图书馆的"还书推车"（= evictable LRU 池）**
> 读者还书后，书先放在**还书推车**上，还没被上架。此时：
> ① 别的读者如果正好要这本书，**可以直接从推车上拿走**（= 前缀缓存命中一个 ref==0 的块，"复活"它）；
> ② 推车满了、要腾位置时，馆员从**最早放上来的那一端**拿走一本，把它的内容抹掉改装成别的书（= LRU 驱逐）。
> 关键细节：抹掉内容的那一刻，必须把**目录卡片**（`hash → block`）也撕掉。否则下一个读者按目录去那个位置，拿到的是一本**封面对、内容全错**的书——这就是 KV 错乱，模型不会报错，只会悄悄说胡话。

> **类比 · 哈希链 = 快递面单上的"全程路径"**
> 如果一个块的标签只写"本块装了 [7, 8]"，那么两个前缀完全不同的序列会误认为能共享它。可是 KV 的值依赖**从句首到这里的全部上下文**，不只是本块那几个 token。
> 所以标签要写成"**上一块的标签 + 本块内容**"（`h_i = hash((h_{i-1}, 本块 token))`），像快递面单层层套住全程路径。只有"整条前缀都一样"的序列才会算出同一个 `h_i`。

框架视角：三个操作 · 一张账本 · 四条不变量

P06 是单线程纯数据结构题，没有并发（没有 T·SBWC·LIFE 可谈）。它的难点全在"**同一份账本被三个操作从不同方向改**"。 把每个操作写成"读什么 → 按什么顺序写什么"，bug 基本就挤不进来了：

操作	读（只读，不能有副作用）	写（顺序很重要）	可能 raise 的位置
`allocate`	① 满块的哈希链（只给 `len//bs` 个满块算） ② 在前 `(len-1)//bs` 块内**从头连续**查 `hash→block`，第一次 miss 就停 ③ 容量：`需要新块数 <= num_free_blocks − 命中块中 ref==0 的个数`	④ 命中块：ref==0 的先移出可驱逐池 → ref+1 ⑤ 其余块：取新块（空闲优先，否则驱逐最旧 + 删哈希登记） ⑥ 满块登记哈希 ⑦ 保存 table / tokens / 哈希链 / 命中 token 数	只在 ③（以及 ⑤ 的兜底）；**必须在任何写之前**
`append_slot`	`pos = len(tokens[seq])`；`pos // bs == len(table)` 吗（要不要新块）	① 先取新块（唯一会 raise 的事） ② 再 append token ③ 若 `(pos+1) % bs == 0`：这块刚满 → 用链尾当父哈希，算新哈希、入链、登记	只在 ①；顺序反了会留下"多了一个 token"的脏状态
`free`	该序列的 block table	**逆序**遍历：ref−1；归零时 `block_hash` 非空 → 进可驱逐池（最新一端），否则 → 空闲表；最后删掉该序列的四份记录	不会 raise

不变量（写完可以直接当 assert）

	不变量	违反时的症状
I1	`num_free_blocks == len(空闲表) + len(可驱逐池)`，且空闲表与可驱逐池**不相交**，两者里的块都满足 `ref == 0`	同一个块被分配给两个序列，内容互相覆盖

© 2026 kzhou003

I2	`b in 可驱逐池 ⟺ ref[b] == 0 且 block_hash[b] is not None`	命中后忘了移出池子 → 正在用的块被当成候补驱逐掉
I3	`hash_to_block[h] == b ⟹ block_hash[b] == h`（映射与块的标签永远互指）	驱逐后没删映射 → 旧前缀"命中"一块已被覆盖的块
I4	`ref[b] ==` 包含 b 的 block table 个数；所有序列 free 完后 `num_free_blocks == num_blocks`	block 泄漏：跑一会儿就 OutOfBlocks，重启才好
I5	`num_cached_tokens(seq) % block_size == 0` 且 `< len(token_ids)`（至少留一个 token 要算）	全部命中时模型没有 token 可 forward，拿不到最后位置的 logits，采不出下一个 token

易错点清单

- **给半满的最后一块也算哈希**：之后一个更长的 prompt 会命中只装了一部分 token 的块。只有满块才有身份。
- **可匹配上限写成** `len // bs`：prompt 刚好是 bs 整数倍且全命中时 `num_cached_tokens == len(token_ids)`，违反 I5。
- **哈希链起点不一致**：`allocate` 用 `None` 当 `h_{-1}`、`append_slot` 用别的哨兵，于是 decode 填满的块永远不会被 allocate 命中（缓存像"只写不读"）。
- **驱逐时无条件** `del hash_to_block[h]`：当同一个哈希被两个块算出来（内容相同）、映射指向的是**另一个**块时，会把别人的有效映射删掉。要先确认映射仍指向自己。
- **free 用正序**：尾部块（最不通用的前缀）会变成"最新"从而最晚被驱逐，共享度最高的首块反而先死，命中率下降。
- `can_allocate` **与** `allocate` **的检查混为一谈**：前者粗（不看命中，偏保守），后者必须精确（见上面的原子性陷阱）。

交互图：`nano-vllm-omni-drills/hints/p06_overview.html`

对照源码： `vllm/v1/core/kv_cache_manager.py` `vllm/v1/core/block_pool.py`

笔记 / 我踩到的坑：

© 2026 kzhou003

#18 · P08　批不变的采样器（per-request seed）　45 min　中　□ 完成　日期 _____

> talker 是唯一随机的 stage；vllm-omni 给它固定 seed 保证可复现。但如果同一个 seed 的结果随 batch 里别人变化，那 seed 就毫无意义。

在系统中的位置

每个 LLM stage 的 model runner 最后一步：logits → token。Qwen3-Omni 的 deploy 里 thinker 温度 0（greedy），talker `temperature 0.9, top_k 50, repetition_penalty 1.05, seed 42` ——注释说明：不固定 seed 时少数采样会"流畅地胡说"，固定后才能复现。

处理顺序（每行独立）

```
1. repetition penalty：对出现过的每个 token，正 logit 除以 p，负 logit 乘以 p
2. temperature < 1e-5 → argmax，结束
3. logits /= T
4. top_k：只保留最大的 k 个
5. top_p：降序排序后，保留累计概率达到 top_p 的最小前缀（至少 1 个）
6. 用该请求自己的 torch.Generator 抽一个 token
```

批不变性

如果所有行共用一个 generator，请求 a 的随机数取决于 batch 里排在它前面的有几行——每一步的 batch 组成都不同，同一个 seed 就得到不同的音频。所以每个带 seed 的请求拥有自己的 generator（首次使用时创建、跨 step 复用、请求结束时 `free`）。

文件： `mine/p08_sampler.py`　**验收：** `pytest tests/test_p08_sampler.py`　**提示：** A-18　**测试：** B-18　**参考解：** C-18

接口骨架

```python
"""P08 · Batch-invariant sampler with per-request seeds.

- apply_repetition_penalty: for every distinct previous token, positive logits are
  divided by the penalty and negative ones multiplied (do not mutate the input).
- sample(): per row -> repetition penalty -> temperature < 1e-5 means greedy argmax;
  otherwise divide by temperature, apply top_k (keep k largest) then top_p (keep the
  smallest prefix of the sorted distribution whose mass reaches top_p; always keep >= 1),
  then draw ONE token.
- Seeded requests own a torch.Generator created on first use from params.seed; the token
  drawn for a request must not depend on which other rows share the batch, or their order.
- free(req_id) forgets the generator (a new request with the same id restarts the stream).
"""
from dataclasses import dataclass

import torch


@dataclass
class SamplingParams:
    temperature: float = 1.0
    top_k: int = -1
    top_p: float = 1.0
    repetition_penalty: float = 1.0
    seed: int | None = None


def apply_repetition_penalty(logits: torch.Tensor, prev_tokens: list[int], penalty: float) -> torch.Tensor:
    raise NotImplementedError


class Sampler:
    def __init__(self) -> None:
```


© 2026 kzhou003

```
        raise NotImplementedError

    def free(self, req_id: str) -> None:
        raise NotImplementedError

    def sample(
        self,
        logits: torch.Tensor,  # [B, V]
        req_ids: list[str],
        params: list[SamplingParams],
        prev_tokens: list[list[int]],
    ) -> torch.Tensor:  # [B] long
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ greedy
- ✓ top_k=1 / 极小 top_p 等价于 greedy
- ✓ 同 seed 的请求在不同 batch 组成/顺序下结果完全一致
- ✓ top_k=2 的经验分布 ≈ 0.7/0.3
- ✓ repetition penalty 数值与效果
- ✓ free 后重新开始随机流

面试官追问

- 向量化写法：用 per-row generator 生成指数噪声做 exponential race。
- 为什么带 per-row generator 的 batch 不能直接 replay CUDA graph? （→ P17）

直觉与类比

类比 · 选秀：扣分 → 放大分差 → 入围名单 → 抽签

一行 logits 就是 V 个选手的**评委打分**，采样 = 从里面挑一个出场。四道工序：

① **repetition penalty**：刚唱过的歌手扣分（已经出现过的 token 降权）；

② **temperature**：T<1 把分差**放大**（强者更强，输出更稳），T>1 把分差**压平**（更有创意也更容易胡说），T→0 直接点第一名（greedy）；

③ **top_k**：只留分数最高的 k 个人；

④ **top_p**：按分数从高到低累计"得票率"，够 p 了就封闭名单（动态的 k）；

最后在名单里**按概率抽签**——抽签这一下，才是唯一用到随机数的地方。

类比 · 公用摇号机 vs 每人自带骰子

公用摇号机（整 batch 一个 generator）：排在你前面的人每摇一次，机器就往前走一格。你今天排第 3、明天排第 1，摇到的号自然不同——哪怕"机器的初始种子"每天都一样。

自带骰子（per-request generator）：你的骰子只有你自己掷，掷了几次只取决于**你被调度过几次**，与同批有谁、谁排前面完全无关。

还要注意两点：骰子要**一直带着**（跨 step 复用，不是每步换一颗新的），用完要**还掉**（`free`，否则下一个同名请求会接着你的随机流用）。

框架视角：shape trace 与易错点

P08 是纯函数题（没有并发），正确性全在"**形状、顺序、副作用**"这三件事上。先把一行的数据流写清楚，再套上 B 行的循环：

© 2026 kzhou003

步骤	张量 / 形状	做什么	易错点
0	`logits [B, V]`，dtype 可能是 bf16/fp16	逐行取出 `logits[i]` 并 `.float()`	直接在 bf16 上做 softmax / cumsum，数值与参考实现差到 1e-2
1	`row [V]` fp32	`apply_repetition_penalty(row, prev_tokens[i], p)`：对 **去重后**的历史 token，`v > 0 ? v/p : v*p`	① 重复 token 罚多次；② 负 logit 也做除法（反而变大 = 奖励）；③ 原地改了调用方的张量
2	`row [V]`	`if temperature < 1e-5: return argmax(row)`；否则 `row = row / temperature`	先除后判断 → `/0` 得 inf/NaN；在 penalty 之前判断 greedy → penalty 对 greedy 行失效
3	`row [V]`，部分置 `-inf`	仅当 `0 < top_k < V`：取第 k 大的值 `kth`，把 `row < kth` 置 `-inf`	`top_k = -1` 或 `>= V` 没当成"不启用"；并列第 k 名时会多保留几个（可以接受，但要知道）
4	排序视图 `sorted [V]`、`order [V]`	降序排序 → softmax → cumsum；**"它之前的累计质量" = `cum - probs`**；该值已超过 top_p 的位置置 `-inf`；强制保留第 0 位	把"越过阈值的那个 token"也删掉 → 极小 top_p 时全删光 → softmax 得 NaN；忘了用 `order` 映射回原位置
5	`probs [V]` → 标量	`torch.multinomial(probs, 1, generator=self._gen(req_id, p))`	整批共用一个 generator（批不变性没了）；每步重建 generator（分布偏斜）；忘了 `free`
6	`out [B]` `torch.long`	写回第 i 个位置并返回	返回 float / 形状 `[B,1]`；行顺序与 `req_ids` 不一致

不变量（可以直接写成自检脚本）

	不变量	怎么验
I1	**批不变**：固定 seed 的请求，连续 N 步的 token 序列与它"单独成批"时逐个相同	把同一个请求塞进随机打乱的 batch 跑 N 步，和单独跑 N 步的列表比较（见交互图 B）
I2	**无副作用**：`sample()` 返回后，传入的 `logits` 逐位不变	`before = logits.clone()`，调用后 `assert torch.equal(before, logits)`
I3	`temperature < 1e-5`、`top_k == 1`、`top_p → 0` 三者都必须退化成 **argmax**（的那一个 token）	三种参数各跑一次，结果都应等于罚分后 logits 的 argmax（本页例子里都是 token 0）
I4	保留集合非空：任何参数组合下至少有 1 个 token 的概率 > 0（没有 NaN）	用 `top_p=1e-9`、`top_k=1` 等极端值跑，断言 `probs.sum() > 0` 且无 NaN
I5	**随机流连续**：同一请求第 n 次抽样用的是它 generator 的第 n 个随机数；`free` 后重置	抽 4 次 → `free` → 再抽 4 次，两段列表必须相同（本页例子：`[0, 2, 0, 2]` 两次）
I6	经验分布 ≈ 过滤后的概率	同一行抽几千次统计频率（本页例子 4000 次：`0.666 / 0.189 / 0.146`，对比理论 `0.6624 / 0.1898 / 0.1478`）

交互图：`nano-vllm-omni-drills/hints/p08_overview.html`

对照源码： `deploy/qwen3_omni_moe.yaml:49-62` `worker/gpu_model_runner.py:1998-2010`

© 2026 kzhou003

笔记 / 我踩到的坑：

© 2026 kzhou003

#19 · P09 Paged KV 写入 + decode attention（参考实现）

45 min　中　□ 完成　日期 _____

> 先用 PyTorch 写出正确的参考实现，它就是之后 Triton kernel 的对拍标准。

在系统中的位置

P06 决定了每个 token 的 K/V 存在哪个物理 slot。model runner 每步：先按 `slot_mapping` 把新 token 的 K/V 写进 cache，再让 attention 按 `block_tables` 去 cache 里读历史。

```
cache 布局:     [num_blocks, block_size, num_kv_heads, head_dim]
写:             cache[slot // bs, slot % bs] = k[i]          (slot = -1 是 padding，不写)
decode 读:      序列 b 的前 context_lens[b] 个 token:
                k_b = cache[block_tables[b, :ceil(n/bs)]].reshape(-1, KVH, D)[:n]
                out_b = softmax(q_b · k_bᵀ · scale) · v_b    (GQA：head h 用 kv head h // (H/KVH))
```

易错点

- 物理块不连续（测试会打乱分配）；block table 尾部未用的位置是垃圾值，不能读。
- CUDA graph 模式下 batch 会被 pad，pad 出来的 token 的 slot 是 -1。
- softmax 用 fp32。

文件： `mine/p09_paged_attention.py`　**验收：** `pytest tests/test_p09_paged_attention.py`　**提示：** A-19
测试： B-19　**参考解：** C-19

接口骨架

```python
"""P09 · Paged KV write + decode attention (PyTorch reference; Triton is the follow-up).

Cache layout: [num_blocks, block_size, num_kv_heads, head_dim].
- write_kv: scatter token i's K/V into slot slot_mapping[i] = block*block_size + offset;
  slot -1 marks a padding token that must NOT be written.
- paged_decode_attention: one query token per sequence. Sequence b attends to its first
  context_lens[b] cached tokens, found through block_tables[b] (entries past the last
  used block are garbage). Support GQA: num_heads is a multiple of num_kv_heads and query
  head h uses kv head h // (num_heads // num_kv_heads). Softmax in fp32.
"""
import torch


def write_kv(
    k_cache: torch.Tensor,
    v_cache: torch.Tensor,
    k: torch.Tensor,  # [num_tokens, num_kv_heads, head_dim]
    v: torch.Tensor,
    slot_mapping: torch.Tensor,  # [num_tokens]
) -> None:
    raise NotImplementedError


def paged_decode_attention(
    q: torch.Tensor,  # [B, num_heads, head_dim]
    k_cache: torch.Tensor,
    v_cache: torch.Tensor,
    block_tables: torch.Tensor,  # [B, max_blocks]
    context_lens: torch.Tensor,  # [B]
    scale: float,
) -> torch.Tensor:  # [B, num_heads, head_dim]
    raise NotImplementedError
```


© 2026 kzhou003

验收要点（测试会检查）

- ✓ 与 dense attention 数值一致（GQA、非连续物理块、block table 尾部是垃圾值）
- ✓ slot -1 的 padding token 不写入
- ✓ 随机形状与 block_size

面试官追问

- Triton 版：一个 program 处理 (seq, head)，按 block 循环做 online softmax。
- prefill 的 varlen 版本怎么写?

直觉与类比

类比 · 储物柜墙：slot 就是柜号

KV cache 是一面**储物柜墙**：一共 `num_blocks` 组，每组 `block_size` 格。`slot = 组号 × 每组格数 + 格号`，所以"展平后的第 slot 格"和 `cache[slot // bs, slot % bs]` 是**同一个位置**。

一个序列拿到的不是连续的一片墙，而是一张**取件码清单**（block table）：第 1 批放 3 号组、第 2 批放 1 号组…… 组号乱序、不连号，完全没问题——这正是 paged 的意义。

最危险的是 `slot = -1`（CUDA graph 把 batch 补齐出来的 padding token）。Python/PyTorch 里 `-1` **表示"最后一格"**，于是这个本该被丢掉的 token 会**安静地覆盖掉最后一格里别人的包裹**——不报错、不告警。

类比 · 图书馆的索书单

block table 是一张印好格子的索书单（固定 `max_blocks` 行），你只填了前两行，**后面几行是上一个读者留下的乱码**。照着乱码去书库取书，轻则取到别人的书（结果悄悄错），重则取到不存在的架号（IndexError）。**最后一卷只看前几页**：第 2 卷你只用到第 2 页，后面两页是别人的内容。"反正那两页是空白"这种侥幸在这里不成立—— 空白页的 score = 0，在 softmax 里仍然占一份权重（ `e^0 = 1` ），只有**截断**（或者置 `-inf` ）才真正排除它。

框架视角：shape trace 与易错点

这是一道纯函数题（没有并发）。写之前把两条数据流的**形状**逐行写下来，几乎所有 bug 都会在纸上暴露：

`write_kv`（可以整批向量化）

步骤	张量 / 形状	做什么	易错点
1	`slot_mapping [T]`	`keep = slot_mapping >= 0`，得到布尔掩码	没过滤 `-1`：padding token 覆盖**最后一个 slot**，不报错
2	`s = slot_mapping[keep] [T']`	`blk = s // bs`、`off = s % bs`	用 `divmod` 时忘了转 long；int32 索引在某些后端不被接受
3	`k[keep] [T', KVH, D]`	`k_cache[blk, off] = k[keep]`（v 同理）	两个高级索引要**同时**给出（ `[blk, off]` ），写成 `k_cache[blk][off]` 会先复制一份再写，改不到原 cache
4	—	返回 `None`（原地）	返回新张量 → 调用方拿不到写入结果

© 2026 kzhou003

`paged_decode_attention`（逐序列循环即可，先写对）

步骤	张量 / 形状	做什么	易错点
1	`n = int(context_lens[b])`	`nb = ceil(n / bs)`	用 `n // bs` → 最后一个不满的块被整块丢掉
2	`tbl = block_tables[b, :nb].long() [nb]`	只取有效项	取了整行 → 读垃圾值：IndexError，或悄悄读到别人的 KV
3	`k_cache[tbl] [nb, bs, KVH, D]`	按 table 的顺序 gather，**逻辑顺序自动正确**	以为要先排序 table——不行，table 的顺序**就是**逻辑顺序
4	`.reshape(nb*bs, KVH, D)[:n]` → `[n, KVH, D]`	展平成"token 维"，再**截掉**最后一块多出来的位置	不截断 → 多看几个位置，结果悄悄错（本页第 5 节第 3 步有真实数字对比）
5	`.repeat_interleave(H // KVH, dim=1)` → `[n, H, D]`	GQA：让 head `h` 对上 kv head `h // group`	用 `repeat` → 变成 `h % KVH`；`KVH=1` 时两者相同，所以很难发现
6	`scores [H, n]` fp32	`einsum("hd,nhd->hn", q[b].float(), kb) * scale`，再在 `n` 维 softmax	scale 乘在 softmax 之后（分布就错了）；在 bf16 里做 softmax（与参考差 1e-2）
7	`out[b] [H, D]`	`einsum("hn,nhd->hd", probs, vb).to(q.dtype)`	忘了转回 `q.dtype`；head 维和 token 维搞混（形状对、数值错）

不变量 / 自检清单

	不变量	怎么验
I1	`-1` 的 slot 对应的值**不出现在 cache 的任何位置**	写一个所有元素都是哨兵值（如 9.0）的 padding token，写完后 `assert (cache == 9).sum() == 0`
I2	只访问 `block_tables[b, :ceil(n/bs)]`	把 block table 尾部填成 `99999`（超出 num_blocks），正确实现不应报错
I3	参与 attention 的 K/V 行数恰好 `==` `context_lens[b]`	断言 `kb.shape[0] == n`；并用" `n` 不是 bs 整数倍"的长度专门测一次
I4	`probs.sum(-1) == 1`，且与 dense attention 数值一致	自己写一个 dense 版本（连续 K/V 直接算）对拍：随机长度（含 1、bs 的倍数、非倍数）、打乱物理块、尾部填垃圾值。本页那组随机用例的最大绝对误差是 **0.0**
I5	GQA 映射正确：head `h` 用 kv head `h // group`	`H=4, KVH=2` 时 `repeat_interleave` 得 `[0,0,1,1]`，`repeat` 得 `[0,1,0,1]` ——必须是前者

交互图：`nano-vllm-omni-drills/hints/p09_overview.html`

对照源码： `vllm/v1/attention/backends/triton_attn.py`

笔记 / 我踩到的坑：

© 2026 kzhou003

Track C · 数据面

跨进程 payload：共享内存协议、后台收发线程、顺序与清理。

© 2026 kzhou003

#20 · P12 SharedMemoryConnector：跨进程 put/get 协议

60 min　难　□ 完成　日期 _____

> thinker 的 hidden states 每 token 都要发给 talker 进程。用 POSIX 共享内存，没有额外的元数据通道——两边只靠 key 推出名字，用文件锁表达“写完了”。

在系统中的位置

stage 之间的大块数据（hidden states、embedding、codec）不走 orchestrator，而是由 worker 直接写进共享内存，下游 stage 的后台线程去读。同机部署默认就是 SharedMemoryConnector。

协议

```
key 例如 "req42_0_7"（请求 42、stage 0 发出的第 7 块）
name = 短哈希(key)                        # macOS 的 POSIX shm 名字最长 31 字节
lock = <lock_dir>/<name>.lock

put(key, data):                            get(key):
  open(lock, O_CREAT); flock(EX)             open(lock)            → 不存在：None
  创建 shm（若残留旧段，先 unlink）          flock(EX | NB)        → BlockingIOError：还在写，None
  写 8 字节长度 + data                       打开 shm              → 不存在：None
  flock(UN)                                  读出 → close → unlink（读者释放） → 删除 lock 文件
```

8 字节长度头的原因：shm 段大小会被向上取整到页，读者无法从段大小得知真实长度。

resource tracker 陷阱（真实的坑，测试会复现）

Python < 3.13 中，`SharedMemory(create=True)` 会把这个段登记到**创建者**的 resource tracker。如果写进程是独立的解释器并先退出，它的 tracker 会把仍在登记中的段当作“泄漏”删掉——读者再去读时，数据已经没了。解决：写完后把所有权转移给读者（从 tracker 注销），由读者 unlink。

生产者清理

请求被 abort 时，下游可能永远不会来读。生产者要记住自己写过的 key，`cleanup_prefix("req42_")` 删除其中尚未被消费的段。

文件：`mine/p12_shm_connector.py`　**验收：**`pytest tests/test_p12_shm_connector.py`　**提示：**A-20　**测试：**B-20　**参考解：**C-20

接口骨架

```
"""P12 · ShmConnector: byte payloads through named POSIX shared memory (one consumer/key).

Protocol (no metadata channel -- both sides derive names from the key):
- shm_name(key): a short, deterministic name (macOS caps POSIX shm names at 31 bytes).
- put(key, data): take an EXCLUSIVE flock on lock_path(key) (create the file), create the
  segment (replace a stale one), write an 8-byte length header + data, then unlock.
  Remember the key so cleanup_prefix() can find it.
- get(key): non-blocking. None if the lock file is missing, the lock is held by someone
  else (LOCK_NB -> BlockingIOError = "still being written"), or the segment is missing.
  Otherwise read, close, UNLINK the segment and remove the lock file (the reader frees it).
- cleanup_prefix(prefix): producer side; unlink every key it wrote with that prefix that
  was never consumed; return how many it removed.
- The payload must survive the WRITER PROCESS EXITING before the reader reads.
  (Hint: look at multiprocessing.resource_tracker.)
"""


class ShmConnector:
    def __init__(self, lock_dir: str | None = None, namespace: str = "nvo") -> None:
        raise NotImplementedError

    def shm_name(self, key: str) -> str:
```


© 2026 kzhou003

```
        raise NotImplementedError

    def lock_path(self, key: str) -> str:
        raise NotImplementedError

    def put(self, key: str, data: bytes) -> None:
        raise NotImplementedError

    def get(self, key: str) -> bytes | None:
        raise NotImplementedError

    def cleanup_prefix(self, prefix: str) -> int:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 往返且只能消费一次
- ✓ 空 payload 与 4MB payload
- ✓ 独立的写进程先退出 1 秒后，读者仍然能读到（resource tracker 陷阱）
- ✓ 锁被他人持有时 get 返回 None
- ✓ cleanup_prefix 只清理未被消费的同前缀 key

面试官追问

- 同卡场景用 CUDA IPC 零拷贝：生命周期和跨进程流同步怎么保证?
- 生产者如何在不等 ACK 的前提下回收自己的 bookkeeping?

直觉与类比

类比 · 小区的快递柜

共享内存段就是一个**快递柜格子**，key 是取件码。快递员（**写者**）和收件人（**读者**）从没见过面，但他们用同一张表从取件码算出**同一个格子号**（`shm_name(key)`）——所以不需要任何人转告"放在几号柜"。

每个格子门上有一把**锁**（锁文件 + `flock`）：快递员**先锁门、再往里放东西**，放完才开锁。收件人来了先试着锁门：锁不上（`LOCK_NB` → `BlockingIOError`）说明还在放，**转身就走**、待会儿再来，绝不站在门口干等。

格子都是标准大号（一页），装一支笔也是这么大。所以快递员在格子最前面贴一张**纸条**："里面实际 N 字节"（8 字节长度头）。

收件人取走东西后**亲手退掉格子**（`unlink`）并摘掉门上的锁（删锁文件）——只有他知道"已经取完了"。

类比 · 物业的"退租清扫员"（resource tracker 陷阱）

每个 Python 进程都有一个物业清扫员（`resource_tracker` 子进程）。快递员**租**格子时（`SharedMemory(create=True)`），物业把格子登记在**快递员名下**。

快递员下班（**写者进程退出**）时，清扫员一查：登记在他名下的格子还没退？"肯定是忘了"——于是把格子**清空退掉**，还在日志里抱怨一句 `leaked shared_memory objects`。可那件包裹收件人还没来取！

解决办法：放好包裹后，快递员去物业**把登记撤掉**（`resource_tracker.unregister`），相当于把格子的所有权**过户**给收件人；收件人取件后自己退格子。

框架视角 · 用 T · SBWC · LIFE 看 P12

P12 的 T 和前面几道题完全不同：参与者是**进程**，不是线程或 task。进程之间不共享 Python 对象，`threading.Lock` 毫无意义；能用的只有内核提供的东西——共享内存段、文件、`flock`——以及双方事先约定好的**协议**（名字怎么算、谁创建、谁释放）。

© 2026 kzhou003

	问题	P12
T	进程 / 线程 / 边界?	写者进程（stage s 的 save 线程）、读者进程（stage s+1 的 recv 线程），外加一个容易被忘掉的第三方：**每个进程自己的 resource_tracker 子进程**。跨进程的同步原语只有 `flock`（建议性锁，只对同样使用 flock 的参与者有效）。
S	共享状态? 谁写?	内核里的三样：段（写者创建、读者 unlink）、锁文件（写者 `O_CREAT`、读者删除；它的**存在**本身就是状态："有一个未被消费的 key"）、flock 的持有者。进程本地的：写者的"我写过的 key"集合（只给 `cleanup_prefix` 用）、各进程 tracker 里的登记表。
B	哪里会等?	写者的 `flock(LOCK_EX)` 可以阻塞（最多等读者读完一块，微秒级）；读者**永不阻塞**：`LOCK_EX | LOCK_NB`，拿不到就返回 None，由 P13 的 recv 线程稍后重试。
W	谁做什么?	写者：创建 + 写 + 交出所有权。读者：读 + 拷贝 + 释放。tracker：进程退出时清扫仍登记着的段——它是**你没写、但一定会运行**的 worker。
C	通道?	"名字"就是通道：`shm_name(key)` / `lock_path(key)` 两边独立计算。唯一的带内元数据是 8 字节长度头。
L	生命周期?	段：写者 create → 写 → close（**不** unlink）→ 从 tracker 注销 → 读者 attach → 拷贝 → close → **unlink**。锁文件：写者创建 → 读者删除。被 abort 的请求：写者用 `cleanup_prefix` 回收没人读的段。
I	对外接口?	`put` / `get` / `cleanup_prefix`（+ 两个纯函数 `shm_name` / `lock_path`）。`get` 是非阻塞、消费式的。
F	不变量?	① 读者能锁上门时，段要么不存在、要么已经完整写好（永远读不到半成品）；② 每个段恰好被 unlink 一次；③ 锁文件存在 ⇔ 这个 key 写过且没被消费；④ `get` 的每一条 return 路径都解锁并关闭 fd。
E	边界与故障?	同名残留段（`FileExistsError` → 替换）；锁文件在而段不在；写者先退出；空 payload 与 4 MB payload（实测都能往返）；前缀 `"spk3_"` 不能误伤 `"spk30_…"`。

交互图：`nano-vllm-omni-drills/hints/p12_overview.html`

对照源码： `distributed/omni_connectors/connectors/shm_connector.py:195-364` `entrypoints/stage_utils.py:193-237`

笔记 / 我踩到的坑：

© 2026 kzhou003

#21 · P13 ChunkAdapter：后台收发线程与顺序/清理语义

⏱ 75 min　难　□ 完成　日期 _____

> async_chunk 的心脏：调度线程只做快照入队，save 线程构造 payload 并 put，recv 线程轮询 get。还要处理 cancel、None payload，以及“清理时恰好有一个 chunk 在读”。

在系统中的位置

每个 stage 进程的 scheduler 里有一个 ChunkAdapter：它是 connector（只管传输，P12）和 scheduler（P11）之间的一层，负责 chunk 的完整生命周期。

```
发送端（stage 0）                                    接收端（stage 1）
scheduler 线程: save_async(req, snapshot, final)    scheduler 线程: load_async(req) / poll_ready()
     | 只做 deepcopy + 入队（不阻塞调度）                  ▲ 已提交的 chunk，按序
save 线程: build_payload(snapshot)                   recv 线程: get(f"{req}_{src}_{next_id}")
           put(f"{req}_{stage}_{chunk_id}")                        拿到 → 在锁内检查“还注册着吗” → 提交
           成功后 chunk_id += 1
```

每条规则背后的 bug

规则	不遵守会怎样
save_async 深拷贝快照	scheduler 下一步继续修改请求对象，后台线程读到的是“未来”的数据
build_payload 和 put 都在锁外、在 save 线程	慢的 Python 整形和 SHM 写阻塞调度，GPU 空等
chunk_id 只在 put 成功后递增；None payload 不占 id	接收端按连续 id 轮询，出现空洞就永远卡住
cancel 不能吞掉 final chunk	final chunk 是下游唯一的 EOS，丢了下游请求就永远停在 WAITING_FOR_CHUNK
get 在锁外；提交前在锁内检查注册 token 是否还是同一个	请求刚被清理，后台线程把迟到的 chunk 又写回去，污染下一个同 id 请求
没有进展时退避 ~1ms，空闲时在 Condition 上睡	busy-poll 吃满一个核，和 tokenize 抢 GIL

文件： `mine/p13_chunk_adapter.py`　**验收：** `pytest tests/test_p13_chunk_adapter.py`　**提示：** A-21　**测试：** B-21　**参考解：** C-21

接口骨架

```
"""P13 · ChunkAdapter: the async-chunk data plane of one stage (sender + receiver halves).

Sender (scheduler thread calls save_async; a background SAVE thread does the work)
- save_async(req, snapshot, is_final) returns immediately; it must snapshot (deep-copy)
  its argument because the caller keeps mutating the live request.
- The save thread calls build_payload(snapshot) OUTSIDE any lock; None means "nothing to
  send this step" (skip, unless is_final). It puts {"data": ..., "finished": is_final} at
  key f"{req}_{stage_id}_{chunk_id}"; chunk_id starts at 0 and advances only after a
  successful put, so ids stay contiguous.
- cancel_sender(req): later non-final chunks are dropped, but the final chunk is still
  sent (with data=None) because it is the downstream end-of-stream marker.
Receiver (background RECV thread polls the connector)
- load_async(req): start polling f"{req}_{src_stage_id}_{next_id}"; the get happens
  OUTSIDE your locks; back off (~1 ms) when nothing arrived, sleep on a Condition when idle.
- poll_ready() (scheduler thread): {req: [chunk dicts in order]} committed since last call.
- cleanup_receiver(req): from now on nothing for req appears in poll_ready -- including a
  chunk the recv thread is reading at this very moment (re-check a per-registration token
```


© 2026 kzhou003

```
  under the lock before committing).
- start()/stop() manage both threads; stop() must return promptly.
"""
import copy
import threading
from collections import deque
from collections.abc import Callable
from typing import Any


# ---------------------------------------------------------------- given types
class InMemoryConnector:
    """Transport only: put/get by key, get consumes. Thread-safe. Hooks let tests add latency."""

    def __init__(self, on_put: Callable[[str], None] | None = None, on_get: Callable[[str], None] | None = None):
        self._store: dict[str, Any] = {}
        self._lock = threading.Lock()
        self.on_put, self.on_get = on_put, on_get
        self.puts: list[str] = []

    def put(self, key: str, obj: Any) -> None:
        if self.on_put:
            self.on_put(key)
        with self._lock:
            self._store[key] = obj
            self.puts.append(key)

    def get(self, key: str) -> Any | None:
        with self._lock:
            present = key in self._store
        if not present:
            return None
        if self.on_get:
            self.on_get(key)
        with self._lock:
            return self._store.pop(key, None)


# ---------------------------------------------------------------- implement below
class ChunkAdapter:
    def __init__(
        self,
        connector: InMemoryConnector,
        stage_id: int,
        src_stage_id: int | None,
        build_payload: Callable[[Any], Any | None],
    ) -> None:
        self.connector = connector
        self.stage_id = stage_id
        self.src_stage_id = src_stage_id
        self.build_payload = build_payload
        raise NotImplementedError

    def start(self) -> None:
        raise NotImplementedError

    def stop(self) -> None:
        raise NotImplementedError

    def save_async(self, req_id: str, snapshot: Any, is_final: bool) -> None:
        raise NotImplementedError

    def cancel_sender(self, req_id: str) -> None:
        raise NotImplementedError

    def load_async(self, req_id: str) -> None:
        raise NotImplementedError

    def poll_ready(self) -> dict[str, list]:
        raise NotImplementedError

    def cleanup_receiver(self, req_id: str) -> None:
        raise NotImplementedError
```


© 2026 kzhou003

验收要点（测试会检查）

- ✓ 单请求 50 个 chunk 有序
- ✓ 10 请求 × 30 chunk、随机延迟下各自有序
- ✓ save_async 永不阻塞调用者（build_payload 很慢时）
- ✓ 快照隔离：调用后修改原对象不影响 payload
- ✓ cancel 后 final chunk 仍送达，chunk id 连续
- ✓ None payload 被跳过且 id 无空洞
- ✓ cleanup 是提交栅栏：正在 get 中的 chunk 不会在 cleanup 之后出现

面试官追问

- 如果有多个 save 线程，顺序怎么保证?
- build_payload 需要 TP collective 时能放在后台线程吗?

直觉与类比

设定

A 楼是主菜厨房（上游 stage，thinker），**B 楼是摆盘厨房**（下游 stage，talker）。一桌客人（一个请求）的主菜不是一次出完，而是**一份一份地出**；B 楼每收到一份，就加工一份。

两栋楼之间**不能打电话**，只有一排**快递柜**（connector，P12）：每格有柜号，放进去就锁上，有人取走就空了。

A 楼的**主厨**（调度线程）同时看着 30 桌的锅，**一刻都不能离开灶台**：他每走开一秒，30 口锅就都停火一秒（GPU 空转）。

类比 · 传菜窗口和编号小票

后厨（**发送端**）每出一道菜就贴一张**编号小票**：0、1、2...，放到传菜窗口（**connector**）。跑堂（**recv 线程**）按编号去窗口取：**没看到 k 号，就绝不去拿 k+1 号**——这就是顺序的来源，不需要任何别的协调。

所以后厨**只在菜真的放上窗口之后才撕下一张小票**（put 成功后 id+1）。"这一步没有新菜"（payload 为 None）就别撕小票；要是空盘子也占一个号，跑堂会永远站在窗口前等那个不存在的号。

最后一道菜带一块牌子"**本桌已上齐**"（`finished=True`）。客人中途退单（`cancel_sender`）：还没做的菜都可以不做了，但"已上齐"的牌子**一定要送**（盘子是空的，`data=None`）——否则服务员会一直站在桌边等。

还有：厨师下单时拿的是**点菜单的复印件**（`deepcopy`），不是去看那位还在不停加菜的客人本人。

类比 · 翻台和桌卡（generation token）

7 号桌的客人走了（`cleanup_receiver`），马上又坐下一拨新客人，桌号**还是 7**（同一个 request id 再次 `load_async`）。

跑堂手里端着上一拨客人点的菜，从窗口走回来。如果他只问"7 号桌有人吗？"——有！——就会把旧菜端给新客人。

正确做法：每拨客人入座时发一张**独一无二的桌卡**（`object()`）。跑堂出发前记下桌卡，回来时核对"现在桌上的是不是**同一张**卡"（`is`），不是就把菜倒掉。核对和上菜要**一口气**做完（在同一把锁里），这样清台和上菜不会交错。

框架视角 · 用 T · SBWC · LIFE 看 P13

P13 横跨两层边界：**进程之间**只有 connector（P12 的共享内存，或题里的 InMemoryConnector）；**进程之内**是调度线程和后台线程之间的普通线程同步。这里的 T 决定了一切：线程共享内存，所以要锁；有 GIL，所以后台线程做慢的纯 Python 工作也会和调度线程抢 CPU——空闲时必须真正睡下去。

© 2026 kzhou003

	问题	P13
T	进程 / 线程 / 边界?	每个 stage 进程：调度线程（P10 的 busy loop）+ save 线程 + recv 线程（daemon 线程）。线程之间用 `threading.Lock` / `Condition` / `Event`；进程之间只有 connector 上的 key。
S	共享状态? 谁写?	发送端：待发送队列（调度线程 append、save 线程 popleft）、每个请求的下一个发送 id（save 线程写）、已取消集合（调度线程写、save 线程读 → 加锁）。接收端：注册表 `req → token`、下一个接收 id、已提交未取走的列表（调度线程和 recv 线程都写 → 同一把锁）。
B	哪里会等?	`build_payload`、`connector.put`、`connector.get`：慢，**全部在锁外**、在后台线程。空闲时在 `Condition.wait(timeout)` 上睡；有请求但没数据时退避约 1 ms。调度线程上没有任何阻塞。
W	谁做什么、不准做什么?	调度线程：只做快照入队、登记、取结果，**不准**碰慢操作。save 线程：只做 build + put，按 FIFO 一个一个来。recv 线程：只做 get + 提交，**不准**在持锁时 get。
C	通道?	调度线程 → save 线程：deque + Condition 通知。save 线程 → recv 线程（跨进程）：connector 的 key。recv 线程 → 调度线程：ready 字典，由 `poll_ready` 在锁内**整体换出**。
L	生命周期?	`start`：建两个线程；`stop`：置 Event → 对两个 Condition `notify_all` → 带超时 join。每个请求：`load_async` 发 token → 收到 final 或 `cleanup_receiver` 时注销；发送端在 final 发出后复位 id 和取消标记（为同名的下一个请求）。
I	对外接口?	除了 start / stop，所有公开方法都在调度线程上调用，持锁时间必须是微秒级。
F	不变量?	① 每个请求的 chunk id 从 0 连续，只在 put 成功后前进，None 的非 final 块不占 id；② final 块一定发出（被取消时 `data=None`）；③ 只有 token **身份**没变时才提交；④ 每个提交的块在 `poll_ready` 里恰好出现一次、按顺序。
E	边界与故障?	chunk 在 `load_async` 之前就 put 了（connector 会一直保存它，照样能收到）；put 抛异常（id 不前进）；清理时恰好在读；同名请求重新注册；`stop()` 时线程正睡在 wait 上。

交互图：`nano-vllm-omni-drills/hints/p13_overview.html`

对照源码： `distributed/omni_connectors/transfer_adapter/base.py:20-140`
`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:381-503`
`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:539-555`
`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:714-723`
`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:991-1042`

笔记 / 我踩到的坑：

© 2026 kzhou003

Track D · 模型胶水（纯函数）

最容易“差一行”的地方：thinker→talker 输入、codec 窗口、MRoPE 位置。

© 2026 kzhou003

#22 · P15 Thinker→Talker 输入构造（含“差一行”） ⏱ 40 min 易 □ 完成 日期 _____

> vLLM 先把 stop token 追加到输出再判停，所以 thinker 的 hidden 会多一行。talker 吃进去，音频末尾就会多出杂音。

在系统中的位置

非流式（full payload）模式下，thinker 结束时把累积的 hidden 发给 talker（Qwen2.5-Omni 的布局，Qwen3-Omni 类似）。talker 的 prompt 由 thinker 的 prompt 部分 + 两个特殊位置组成，之后每个 decode 步消费一行 thinker 回复向量。

“差一行”

```
prompt: P 个 token，生成: [a, b, <stop>]
vLLM 采样出 <stop> → 先追加到 output → 再检查 stop → FINISHED_STOPPED
累积的 hidden 行数 = P + 3（包含 <stop> 那一步）
talker 应该只消费 a、b 两行 → 去掉最后一行，并从 output_ids 里去掉 stop id
finish_reason = "length"（到达 max_tokens）时没有多余行，全部保留
```

talker prefill 布局（Qwen2.5-Omni）

```
embeds = [prompt_embeds (P 行)] + [spk_embed + codec_pad_embed] + [first_reply + codec_bos_embed]
ids    = [mask_id] * P + [pad_id, start_id]                 # 共 P + 2 个位置
decode: input_k = reply_queue[k] + embed(上一步采样的 codec)；回复向量用完后用 fallback (tts_pad)
```

特殊 id 来自 config，这里通过参数传入。

文件： `mine/p15_talker_inputs.py` **验收：** `pytest tests/test_p15_talker_inputs.py` **提示：** A-22 **测试：** B-22 **参考解：** C-22

接口骨架

```
"""P15 · Thinker -> Talker input construction (Qwen2.5-Omni style, pure tensor functions).

trim_thinker_payload(prompt_ids, output_ids, hidden[R, D], finish_reason)
  vLLM appends the sampled stop token BEFORE checking stop, so a "stop" finish carries one
  extra row/id the talker must not consume. Keep target = P + len(output_ids) - drop rows
  (drop = 1 for "stop", else 0); prefill = rows[:P], decode = rows[P:target]; drop the stop
  id from output_ids. Raise ValueError if fewer than P rows were captured.
build_talker_prefill(prompt_embeds[P, D], first_reply[D], spk, codec_pad, codec_bos, *ids)
  embeds = [prompt rows] + [spk + codec_pad] + [first_reply + codec_bos]   -> [P + 2, D]
  ids    = [mask_id] * P + [pad_id, start_id]
talker_decode_input(reply_queue[K, D], last_codec_embed[D], fallback[D])
  input = next thinker row (or fallback when exhausted) + embedding of the last codec id;
  return (input, remaining_queue).
"""
from dataclasses import dataclass

import torch


@dataclass
class ThinkerPayload:
    prefill: torch.Tensor
    decode: torch.Tensor
    output_ids: list[int]


def trim_thinker_payload(
    prompt_ids: list[int],
    output_ids: list[int],
    hidden: torch.Tensor,
    finish_reason: str,
```


© 2026 kzhou003

```
) -> ThinkerPayload:
    raise NotImplementedError


def build_talker_prefill(
    prompt_embeds: torch.Tensor,
    first_reply: torch.Tensor,
    spk_embed: torch.Tensor,
    codec_pad_embed: torch.Tensor,
    codec_bos_embed: torch.Tensor,
    *,
    mask_id: int,
    pad_id: int,
    start_id: int,
) -> tuple[list[int], torch.Tensor]:
    raise NotImplementedError


def talker_decode_input(
    reply_queue: torch.Tensor,
    last_codec_embed: torch.Tensor,
    fallback: torch.Tensor,
) -> tuple[torch.Tensor, torch.Tensor]:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ stop 结束：去掉一行与 stop id
- ✓ length 结束：全部保留
- ✓ prefill 行数不足时报错
- ✓ talker prefill 的 ids/embeds 布局
- ✓ decode 每步消费一行，耗尽后用 fallback

面试官追问

- async_chunk 模式下 chunk 0 应该带哪些东西？之后的 chunk 呢？

直觉与类比

类比 · 录音机的“咔哒”声

你用老式录音机录一段话，说完后伸手按停止键——但录音机**先录下了按键的“咔哒”声**，然后才停。拿去播放前，你得把结尾那一下咔哒**剪掉**（finish="stop" → drop 1）。

如果录音是因为**磁带用完**自动停的（finish="length"），最后一个字是你真正说的话，**一个字都不能剪**。

还有一种情况：磁带后面多出一截空白（多捕获的行）。正确的剪法是“**只保留应该有的长度**”，而不是“永远剪掉最后一秒”——否则空白被剪了，咔哒声还在。

类比 · 座位表：P 个普通座 + 两个特殊座

talker 的 prefill 像一张有 **P + 2** 个座位的座位表。前 P 个座位坐的是 prompt 的每一行（名牌上统一写 `mask_id` ——talker 不看名牌，只看人本身，即 embedding）。

第 P 号座位是“**主持人席**”：说话人（`spk_embed`）+ codec 的 pad，名牌 `pad_id`。第 P+1 号座位是“**开场席**”：回复的第一行 + codec 的 bos，名牌 `start_id` ——talker 从这里开始说出第一帧 codec。

之后每个 decode 步，回复的下一行“上台”一次；回复的人都上完了，就由替补（`fallback`，真实系统里是 tts_pad）一直顶上。

框架视角：数据流（shape trace）

纯函数题，没有并发。用第 3 节的例子（P = 3，output 4 个含 stop，R = 8，D = 2）走一遍形状：

© 2026 kzhou003

#	张量 / 量	形状	发生了什么
1	`hidden`	`[8, 2]`	thinker prefill 3 行 + 每个 decode 步 1 行（含 stop 那步）+ 1 行多捕获
2	`P`，检查 `R ≥ P`	标量	P = 3；8 ≥ 3，继续（否则 ValueError）
3	`drop`、应有行数	标量	stop 且有输出 → drop = 1；3 + 4 − 1 = 6
4	`rows = hidden[:6]`	`[6, 2]`	行 6（stop）和行 7（多余）都被截掉
5	`prefill / decode`	`[3, 2]` / `[3, 2]`	按 P 切开；decode 行数 == 新的 `len(output_ids)` = 3
6	`spk+pad`、`first_reply+bos`	`[2]` → `[1, 2]`	1-D 向量要先变成一行才能和 `[P, D]` 沿 dim 0 拼接
7	`embeds` / `ids`	`[5, 2]` / 5 个	P + 2 个位置，ids 与 embeds 一一对应
8	`reply_queue`	`[2, 2]` → `[1, 2]` → `[0, 2]`	每步取队头一行；空了之后用 fallback，队列保持 `[0, 2]`

易错点

- **用 `hidden[:-1]` 去掉 stop 行**：多捕获了行时，砍掉的是多余行，stop 行还在（第 3 节的 `[60,60]`）。
- **finish="length" 也 drop**：会把真正的最后一个字丢掉。
- **stop 但 output_ids 为空**：没有“stop 那一行”可丢；硬 drop 会把一行 prompt 切掉。
- **先切片后检查**：切片越界不报错，只会悄悄返回短 tensor；R < P 的检查要放最前面。
- **原地修改调用方的 output_ids**：调用方可能还要拿它 detokenize；返回一个新 list。
- **`[D]` vs `[1, D]`**：直接 `cat` 1-D 向量会报维度错；以及别把 `(ids, embeds)` 的返回顺序写反。
- **队列空时返回 None**：调用方下一步还要拿它继续调；应原样返回 `[0, D]` 的空队列。

交互图：`nano-vllm-omni-drills/hints/p15_overview.html`

对照源码： `model_executor/stage_input_processors/qwen2_5_omni.py:261-317`
`model_executor/models/qwen2_5_omni/qwen2_5_omni.py:760-847`
`model_executor/stage_input_processors/qwen3_omni.py:409-542`

笔记 / 我踩到的坑：

© 2026 kzhou003

#23 · P14 Codec 窗口规划：首块小、后块大、左上下文 ⏱ 40 min 易 □ 完成 日期 _____

> TTFP 要低（首块只要 4 帧），吞吐要高（之后 25 帧一块），接缝要无损（左上下文重算后切片）。

在系统中的位置

talker 每步产出 1 帧 codec。它把帧攒成窗口发给 code2wav；code2wav 每个窗口独立解码（无状态，便于 batch）。Qwen3-Omni 默认：`initial_codec_chunk_frames=4, codec_chunk_frames=25, codec_left_context_frames=25`。

```
next_window(available, emitted, finished) -> (ctx_start, new_start, new_end) | None
  size = initial（第一块且 initial>0）否则 chunk
  available − emitted ≥ size  → new_end = emitted + size
  finished 且还有剩余          → new_end = available（尾部 flush）
  否则                         → None（继续等）
  new_start = emitted;  ctx_start = max(0, new_start − left_ctx)

code2wav 解码 codes[ctx_start:new_end]，得到 (new_end−ctx_start)·up 个样本，
只输出属于 [new_start, new_end) 的那一段（左上下文重算但不重复输出）
```

为什么左上下文能消除接缝

vocoder 是因果的、感受野有限：样本只依赖最近 r 帧。只要 `left_ctx ≥ r`，窗口内新帧看到的上下文就与整段解码完全相同——测试用一个感受野可控的玩具 vocoder 证明这一点（以及不足时确实出现接缝）。

文件： `mine/p14_codec_windows.py` **验收：** `pytest tests/test_p14_codec_windows.py` **提示：** A-23 **测试：** B-23 **参考解：** C-23

接口骨架

```python
"""P14 · Codec window planning for streaming vocoding (pure functions).

next_window(available, emitted, finished, *, initial, chunk, left_ctx)
  available: frames produced so far; emitted: frames already shipped.
  The first window holds `initial` frames (if initial > 0), later ones `chunk` frames.
  Return (ctx_start, new_start, new_end) when a full window is available; when finished,
  flush whatever remains; otherwise None. new_start = emitted and
  ctx_start = max(0, new_start − left_ctx) (left context is re-decoded, not re-emitted).
slice_new_samples(decoded, window, upsample)
  decoded = vocoder(codes[ctx_start:new_end]) has (new_end − ctx_start) * upsample samples;
  return only the samples that belong to frames [new_start, new_end).
"""
from collections.abc import Sequence


def next_window(
    available: int,
    emitted: int,
    finished: bool,
    *,
    initial: int,
    chunk: int,
    left_ctx: int,
) -> tuple[int, int, int] | None:
    raise NotImplementedError


def slice_new_samples(decoded: Sequence[float], window: tuple[int, int, int], upsample: int) -> list[float]:
    raise NotImplementedError
```


© 2026 kzhou003

验收要点（测试会检查）

- ✓ 首块/稳态/尾部 flush 的精确窗口
- ✓ 无首块配置
- ✓ 随机到达模式下窗口恰好平铺、无重叠无空洞
- ✓ 左上下文 ≥ 感受野时流式 == 整段解码；不足时出现接缝

面试官追问

- 做成 ramp [1,2,4,8,16,25] 时 TTFP/RTF 如何变化?
- 为什么无状态解码让 code2wav 可以自由 batch? （对比 M06）

直觉与类比

类比 · 同声传译

译员不会等演讲者讲完一整段才开口——那样听众要干等（TTFP 高）。他先听**两三个词**就开始翻第一句（**小的首块**），之后进入节奏，每次**听一整句**再翻（**大块**，更流畅、更省力）。

但每次开始翻新的一句时，他脑子里会**回想上一句的最后几个词**，好让语气和指代接得上（**左上下文**）——这几个词他**不会再说一遍**（重新解码、但不重复输出）。

演讲结束时，最后剩下的半句也要翻出来（**finished 时 flush 尾巴**）。

类比 · 电视剧的“前情提要”

每一集（窗口）开头都放 1 分钟“前情提要”（`[ctx_start, new_start)`），让观众（vocoder）进入状态。可如果你要把所有集剪成一部完整电影，就必须**把每集的前情提要剪掉**，否则同一段剧情会出现两次。

`slice_new_samples` 就是那把剪刀：剪掉提要，只留正片 `[new_start, new_end)`。

提要要多长？至少要**覆盖观众需要的记忆长度**（vocoder 的感受野）。提要太短，开场几秒观众会“断片”——这就是接缝。

框架视角：数据流（shape trace）

这是纯函数题，没有并发。按“一个窗口的一生”走一遍单位换算（以 `(5, 9, 15)`、`upsample=4` 为例）：

#	量	坐标 / 形状	发生了什么
1	`available, emitted`	标量，全局帧数	17, 9：已产出 17 帧，前 9 帧已经发过
2	`size`	帧数	不是首块 → 用 `chunk = 6`
3	`(ctx_start, new_start, new_end)`	全局帧下标	(5, 9, 15)：上下文 5..8，新帧 9..14；15、16 留给下一次
4	`codes[ctx_start:new_end]`	10 帧（真实系统里每帧是 `num_quantizers` 个码）	发给 code2wav 的那段
5	`decoded`	`[(15−5)·4] = [40]`，**局部**样本坐标	vocoder 输出；下标 0 对应全局帧 5 的第一个样本
6	切片起点	`(9−5)·4 = 16`	把“全局帧 9”换算成局部样本下标
7	返回值	`[24]` = 6 帧 × 4	只输出新帧；所有窗口的输出首尾相接 = 17×4 = 68 个样本

© 2026 kzhou003

易错点

- **首块判断**用“还没发过任何帧”（emitted 为 0）且 `initial > 0`；别用 available 的大小去猜——帧可能一次来很多。
- **判断顺序**：先看“够不够一个完整窗口”，再看“是否 finished 且有剩余”。finished 不等于“把剩下的全部打包”。
- **不要返回空窗口**：流结束且全部发完时返回 None，否则调用方的 while 循环永远停不下来。
- **ctx_start 不能为负**：流开头左边没有那么多帧。
- **局部 vs 全局坐标**：`decoded` 从 `ctx_start` 开始计数；用 `new_start·upsample` 直接切是最常见的 bug（只在 ctx_start=0 时碰巧对）。
- **left_ctx ≥ 感受野** 才能逐样本无接缝；这依赖 vocoder “因果 + 局部”。

交互图：`nano-vllm-omni-drills/hints/p14_overview.html`

对照源码： `model_executor/stage_input_processors/qwen3_omni.py:765-842`
`model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:447-508`

笔记 / 我踩到的坑：

© 2026 kzhou003

#24 · P18 MRoPE 三维位置 ⏱ 40 min 中

□ 完成 日期 _____

> Qwen-VL/Omni 的 thinker 用 (t,h,w) 三维旋转位置。位置算错，模型不会报错，只会悄悄变笨。

在系统中的位置

M01 是"给定位置怎么旋转"，本题是"位置本身怎么算"。model runner 在 prefill 时为每个请求算出 `[3, T]` 的位置，以及一个 `delta` 供 decode 使用。

```
文本 n 个 token:     三轴都是 start + arange(n)
图像 (t, h, w) patch: LLM 网格 (t, h/m, w/m), m = spatial_merge (2)
                    token 顺序 t-major, 然后 h, 然后 w; 位置 = (t_idx, h_idx, w_idx) + start
每段结束: start = 当前最大位置 + 1          # 注意: 不是"已有 token 数"!
delta = (max_pos + 1) - T                    # decode 时 token 的位置 = 绝对下标 + delta (三轴相同)
```

例子

```
text(2) | image(1,4,4) → 2×2 网格 | text(3)
t: 0 1 | 2 2 2 2 | 4 5 6
h: 0 1 | 2 2 3 3 | 4 5 6
w: 0 1 | 2 3 2 3 | 4 5 6          T = 9, max = 6, delta = 7 − 9 = −2, 第 9 个 token decode 位置 = 7
```

图像占了 4 个 token，却只推进了 2 个位置——这就是 delta 为负的原因，也是"用 len(tokens) 当 decode 位置"的 bug 来源。

文件： `mine/p18_mrope_positions.py` **验收：** `pytest tests/test_p18_mrope_positions.py` **提示：** A-24
测试： B-24 **参考解：** C-24

接口骨架

```python
"""P18 · MRoPE (Qwen2-VL style) 3-D positions for interleaved text and images.

segments: ("text", n) or ("image", (t, h, w)) with h, w in vision patches; an image
becomes t * (h // merge) * (w // merge) LLM tokens laid out t-major, then h, then w.
- text tokens: the same running index on the temporal, height and width axes;
- image tokens: (t_idx, h_idx, w_idx) + start;
- after each segment, start = max(position so far) + 1.
Return (positions[3, T] long, delta) with delta = (max position + 1) - T, so a token at
absolute index i >= T decodes at position i + delta on all three axes (decode_position).
"""
import torch

Segment = tuple[str, int | tuple[int, int, int]]


def mrope_positions(segments: list[Segment], spatial_merge: int = 2) -> tuple[torch.Tensor, int]:
    raise NotImplementedError


def decode_position(index: int, delta: int) -> torch.Tensor:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 纯文本退化为普通 RoPE
- ✓ text+image+text 的精确位置与 delta
- ✓ 多帧矩形网格
- ✓ 形状与 dtype

© 2026 kzhou003

面试官追问

- Qwen2.5-Omni 的 TMRoPE 如何按秒交错音频和视频?

直觉与类比

类比 · 电影院座位号

文本 token 像**一个人一张票**：每人都坐"第 k 排第 k 座"（三轴相同），排号一个个往后推。

一张图片像一个**旅行团**：6 个人包了一个 3 排 × 2 座的方块。他们的票写的是"第 2+行 排、第 2+列 座"——同一排的人排号相同。整个团只用掉了**3 排**，而不是 6 排。

旅行团后面来的散客从哪一排开始？从**团里最大的排号 + 1**开始。如果你按"已经进场 8 个人"给他发第 8 排的票，中间就空出了好几排——模型会以为这两段文字隔得很远。

类比 · 楼层和房间号

把 t 想成楼层、h 和 w 想成房间的行列号。一段视频是一栋 3 层楼、每层只有 1 间房：坐电梯上 3 层（t 推进 3），但房间号不变（h、w 都停在同一个值）。

出了楼之后，下一段文字要从"三个坐标里最大的那个 + 1"继续——不管最大值是楼层（视频）还是列号（宽图）。这就是为什么 start 取的是**三轴的 max**。

框架视角：数据流（shape trace）

纯函数题，没有并发。以第 4 节的例子跟踪每个中间量：

#	量	形状 / dtype	发生了什么
1	`segments`	5 段	按序列顺序，和 prompt 里的占位 token 一一对应
2	图像 `(1, 6, 4)` → `(t, gh, gw)`	(1, 3, 2)	`gh = h // m`，`gw = w // m`；t 方向不合并
3	该段的 `(t_idx, h_idx, w_idx)`	3 个 `[6]`	t 最外、h 其次、w 最内地展开：h = 0,0,1,1,2,2；w = 0,1,0,1,0,1
4	块 = stack + start	`[3, 6]` long	start = 2，三行都加 2
5	新 start	int	块的最大值（三轴一起取）+ 1 = 4 + 1 = 5
6	文本块	`[3, n]`	`start + arange(n)` 复制成三行
7	`positions`	`[3, 14]` long	所有块沿 dim=1 拼起来
8	`delta`	int	`(10 + 1) − 14 = −3`，存进请求状态
9	`decode_position(i, delta)`	`(3,)` long	i = 14 → [11, 11, 11]，紧接 prefill 的最大位置 10

易错点

- **start += token 数**：纯文本时正确，有图像就错（例子里第二段文本会从 8 开始而不是 5）。
- **h、w 互换 / 展开顺序错**：方形网格看不出来，一定要用 **矩形** 网格（如 3×2）和 **t > 1** 自测。
- **只看某一个轴的 max**：最大值可能来自 t（视频）、h（高图）或 w（宽图）。
- **delta 少了 +1**，或 decode 时直接用 index 当位置：第一个 decode 位置应该是 max+1。
- **空段**（n = 0 或网格为 0）没有 max，不能用它更新 start；没有任何段时返回 `[3, 0]` 的 long tensor。
- **dtype**：位置是整数下标，用 `torch.long`。

交互图：`nano-vllm-omni-drills/hints/p18_overview.html`

© 2026 kzhou003

对照源码： `model_executor/models/qwen2_5_omni/qwen2_5_omni.py:412-576`

`model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1694-1785`

笔记 / 我踩到的坑：

© 2026 kzhou003

Track E · 启动与 GPU

多 stage 共卡启动、显存准入、CUDA graph bucket。

© 2026 kzhou003

#25 · P16　启动规划：设备分组、并行初始化、回滚、准入　45 min　中　□ 完成　日期 _____

> thinker 独占 GPU0、talker 和 code2wav 共享 GPU1：后两者必须串行初始化（显存测量会互相干扰），但两组之间可以并行。任何一个失败都要把已起来的全部回收。

在系统中的位置

每个 stage 启动时会：加载权重 → 跑一次最大 batch 的 profile → 用"显存预算 − 已用"算出 KV block 数 → capture CUDA graph。第 2、3 步依赖**测量当前空闲显存**：如果同卡上另一个 stage 恰好在加载权重，测量结果就是错的——KV 算多了会 OOM，算少了浪费。

需求

- **分组**：共享设备（含传递关系：{0,1} 和 {1,2} 也是一组）的 stage 属于同一组；组内按 stage id 串行，组间在线程池里并行。
- **失败短路**：第一个失败发生后，不再启动任何新的 init；正在进行的让它结束；然后对所有已成功的 stage 调用 cleanup；抛 `InitError(stage_id)`。
- **准入检查**：启动前对每张卡检查 `Σ(util × total + reserve) ≤ total`，提前拒绝不可能成功的部署，而不是等到第三个 stage OOM。

```
默认 Qwen3-Omni (2×H100):
  GPU0: thinker(0.9)            → 组 [0]
  GPU1: talker(0.6) code2wav(0.1) → 组 [1, 2] (串行)      两组并行
```

文件： `mine/p16_init_planner.py`　**验收：** `pytest tests/test_p16_init_planner.py`　**提示：** A-25　**测试：** B-25　**参考解：** C-25

接口骨架

```
"""P16 · Stage start-up planning: device groups, parallel init, rollback, admission.

init_groups(stage_devices): stages that share a device -- directly or transitively --
  form one group. Return groups as sorted stage lists, ordered by their first stage.
run_init(stage_devices, init_fn, cleanup_fn, max_workers):
  groups run in parallel (thread pool); stages inside a group run one after another in
  stage-id order. On the first failure: no NEW init_fn calls start, running ones finish,
  then cleanup_fn(stage, result) for every stage that succeeded, and raise
  InitError(failed_stage, cause). Returns {stage: init_fn(stage)} on success.
admission_check(device_total_bytes, stages, reserve_bytes_per_stage):
  for every device, sum(utilization * total + reserve) over stages placed on it must not
  exceed total; return one error string per offending device (mentioning "device <id>").
"""
from collections.abc import Callable
from typing import Any


class InitError(Exception):
    def __init__(self, stage_id: int, cause: BaseException) -> None:
        super().__init__(f"stage {stage_id} failed to initialize: {cause!r}")
        self.stage_id = stage_id


def init_groups(stage_devices: dict[int, set[int]]) -> list[list[int]]:
    raise NotImplementedError


def run_init(
    stage_devices: dict[int, set[int]],
    init_fn: Callable[[int], Any],
    cleanup_fn: Callable[[int, Any], None],
```


© 2026 kzhou003

```
    max_workers: int = 4,
) -> dict[int, Any]:
    raise NotImplementedError


def admission_check(
    device_total_bytes: dict[int, int],
    stages: dict[int, tuple[set[int], float]],
    reserve_bytes_per_stage: int,
) -> list[str]:
    raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 设备重叠的连通分量
- ✓ 组间并行、组内串行且按 stage id
- ✓ 失败短路 + 回滚已成功的 stage
- ✓ 显存准入检查

面试官追问

- 真正并行初始化同卡 stage 时，怎样用跨进程读写锁让“测量空闲显存”独占?

直觉与类比

类比 · 旅行团拼桌（设备分组 = 并查集）

餐厅里每张**桌子**是一张 GPU，每个**旅行团**是一个 stage。**拼桌是允许的**：一张桌子可以同时坐好几个团（多个 stage 共用一张卡）；大团一张桌子坐不下，会**同时占两张桌子**（TP=2 的 stage 占两张卡）。

唯一要排队的是**入座**（初始化）：每个团入座时要先看“这张桌子还剩多少空位”，再摆好自己的餐具（显存 profiling + 分配）。如果两个团**同时**在同一张桌子入座，都会以为空位是自己的，结果摆多了。所以**共用桌子的团必须一个一个入座**（组内串行），坐好之后大家一起吃饭完全没问题；完全不共用桌子的团可以同时入座（组间并行）。

关键在**传递**：A 团坐 2 号桌，B 团同时占 2 号和 3 号桌，C 团坐 3 号桌。A 和 C 没拼过桌，但 B 要等 2、3 号桌**都没人在入座**时才能入座——最简单又一定正确的做法，是让**同一个领位员**按顺序带 A、B、C 入座。（这是偏保守的方案：A 和 C 理论上能同时入座，但会让 B 的安排变复杂。）不同领位员负责互不相连的几张桌子，可以同时工作。

并查集就是“合并领位单”：每来一个团，看它要坐的桌子是否已经在某个领位员的单子上；是，就把两张单子合成一张。

类比 · 搬家公司撤场（失败回滚）

三个搬家小组同时往三栋楼里搬家具（并行初始化）。B 组在第二件家具时摔坏了（init 失败），总调度喊停：① **还没开始搬的**不准再开始（短路，不启动新的 init）；② **正在搬的**那件不能半空中扔下，搬完再说（正在跑的 init 让它跑完）；③ 等**所有人都停手**之后，再把已经搬进去的家具**全部搬出来**（cleanup 每个成功的 stage）——如果在 B 摔坏的那一刻就开始清点，会漏掉 A 组手上那件“稍后才放下”的家具。

类比 · 餐厅订座（显存准入，接着上面的拼桌）

一张 80 座的桌子（一张卡），每个旅行团（stage）按“桌子大小的百分比”订座，外加每桌固定要几把备用椅（reserve）。开饭前先算一遍：`Σ(比例 × 80 + 备用椅) ≤ 80`？超了就当场告诉客人“这桌坐不下”，而不是等上了两道菜再赶人走。跨两张桌子的大客户（TP=2），**每张桌子**都要按它的比例各算一份。

© 2026 kzhou003

框架视角：T · SBWC · LIFE

	问题	P16 怎么答
T	代码跑在哪？跨什么边界？	主线程调用 `run_init`；一个**线程池**，每个组一个任务（不是 asyncio：`init_fn` 是阻塞的同步调用——加载权重、等子进程就绪）。真实系统里每个 stage 还是独立进程，同卡测量的互斥靠跨进程文件锁。
S	共享状态？谁写？	`results`（成功的 stage → handle）、`failure`（第一次失败，最多一条）。多个 worker 线程都会写 → 一把 `threading.Lock` 保护两者。
B	谁在干等？	`init_fn(sid)`——很慢。它必须在**锁外**调用，否则组间并行就退化成全局串行。主线程在"线程池退出"处等所有组结束。
W	岗位	每个组一个 worker：按 stage id 顺序 for 循环；**每次开始一个新的 init 之前**看一眼 failure。不准做的事：组内并行、在失败瞬间做 cleanup。
C	通道	没有队列；worker 之间只通过锁保护的共享状态交流（一面"已出事"的旗子）。
L	生命周期	**先 join 再回滚**：线程池的 with 块退出 = 所有组都结束、results 不再变化，然后才统一 cleanup、再 raise。
I	对外接口	成功返回完整 dict；失败只有一种出口：`InitError(第一个失败的 stage)`，原始异常挂在 `__cause__` 上。
F	不变量	① 同组 stage 的 init 永不重叠，且按 id 顺序；② 第一次失败之后不再**开始**新的 init；③ 返回/抛出时，所有成功过的 stage 要么在结果里，要么已被 cleanup——恰好一次；④ 失败的 stage 不 cleanup（它没有 handle）。
E	边界与错误	多个组几乎同时失败（只记第一条）；正在跑的 init 在失败之后才成功（也要回滚）；`max_workers` 小于组数（排队的组轮到时发现已失败，直接返回）；`KeyboardInterrupt` 之类不是 `Exception` 的异常也要触发回滚；空卡不报错；need 恰好等于 total 允许。

交互图：`nano-vllm-omni-drills/hints/p16_overview.html`

对照源码： `engine/stage_runtime.py:879-956` `engine/stage_init_utils.py:1738-1762` `engine/stage_admission.py:205` `engine/stage_phase_lock.py:106-238`

笔记 / 我踩到的坑：

© 2026 kzhou003

#26 · P17 CUDA graph bucket runner（CPU 模拟） 40 min 易 □ 完成 日期 _____

> decode 的 kernel launch 开销靠 CUDA graph 消除。graph 只认静态 buffer：pad、清零、replay、clone，以及什么时候必须退回 eager。

在系统中的位置

decode 时每步只有很少的计算，但要发射成百上千个 kernel；CPU 发射速度成了瓶颈。CUDA graph 把一次 forward 录成图，之后一次 launch 整图重放。代价：图只认**录制时的那块内存**和**录制时的形状**。

```
启动时：为每个 batch bucket（1,2,4,8,...）在同一块静态 buffer 的前 b 行上录一张图
运行时（bs 个请求）：
  b = 最小的 ≥ bs 的 bucket；没有 → eager
  static_in[:bs] = x;  static_in[bs:b] = 0      # 上一次更大 batch 留下的脏行
  graph[b].replay()                             # 结果写在 static_out[:b]
  return static_out[:bs].clone()                # 下一次 replay 会覆盖 static_out!
```

本题用 `CpuGraph` 在 CPU 上模拟这种语义：replay 永远只读静态 buffer、不接受新参数。

必须退回 eager 的情况

- bs 超过最大 bucket；
- batch 里有请求自带 RNG generator（P08）：replay 不会重新执行 Python 代码，per-row generator 会被悄悄忽略。

文件： `mine/p17_graph_runner.py` **验收：** `pytest tests/test_p17_graph_runner.py` **提示：**A-26 **测试：**B-26 **参考解：**C-26

接口骨架

```
"""P17 · GraphRunner: batch-size buckets over captured graphs (CPU emulation of CUDA graphs).

- __init__: allocate static_in / static_out of shape [max bucket, hidden] once.
- capture(): one CpuGraph per bucket b over the views static_in[:b] / static_out[:b]
  (largest bucket first), call .capture(), keep them in self.graphs[b].
- run(x, force_eager=False):
    pick the smallest bucket >= bs; if none, or force_eager (e.g. rows carry their own RNG
    generators, which a replay would silently ignore) -> stats["eager"] += 1, return fn(x);
    else copy x into static_in[:bs], ZERO static_in[bs:b] (stale rows), replay,
    stats["replays"] += 1, stats["last_bucket"] = b, and return a result that is NOT a view
    of static_out (the next replay overwrites it).
- Never capture at run time.
"""
from collections.abc import Callable

import torch


# ---------------------------------------------------------------------- given types
class CpuGraph:
    """Emulates torch.cuda.CUDAGraph semantics on CPU: replay() re-runs the recorded work on
    the *same static buffers*; it never sees new Python arguments."""

    def __init__(self, fn: Callable[[torch.Tensor], torch.Tensor], static_in: torch.Tensor, static_out:
torch.Tensor):
        self._fn, self._in, self._out = fn, static_in, static_out
        self.captured = False

    def capture(self) -> None:
        self._out.copy_(self._fn(self._in))
        self.captured = True

    def replay(self) -> None:
```


© 2026 kzhou003

```
        assert self.captured
        self._out.copy_(self._fn(self._in))


# ---------------------------------------------------------------- implement below
class GraphRunner:
    def __init__(self, fn: Callable[[torch.Tensor], torch.Tensor], buckets: list[int], hidden: int,
                 dtype: torch.dtype = torch.float32) -> None:
        self.fn = fn
        self.buckets = sorted(buckets)
        self.graphs: dict[int, CpuGraph] = {}
        self.stats = {"replays": 0, "eager": 0, "last_bucket": None}
        raise NotImplementedError

    def capture(self) -> None:
        raise NotImplementedError

    def run(self, x: torch.Tensor, *, force_eager: bool = False) -> torch.Tensor:
        raise NotImplementedError
```

验收要点（测试会检查）

- ✓ 每个 batch size 结果与 eager 一致
- ✓ bucket 选择与两种 eager 回退
- ✓ 返回值不别名静态 buffer
- ✓ padding 行被清零
- ✓ 运行时从不重新 capture

面试官追问

- graph 的显存要不要计入 KV 预算？vllm-omni 默认路径哪里漏算了？
- talker 的 MTP（M05：15 次小模型 forward）为什么适合整体包成一张 FULL graph？

直觉与类比

类比 · 固定尺寸的烤盘

面包房只有 1 格、3 格、6 格三种烤盘，每种烤盘的烤箱程序都**提前调好了**（capture），开工后不再重新编程。

来了 2 个面团：用 **3 格**的烤盘（向上取整到 bucket），空着的那一格要**先把上一炉留下的面包屑扫干净**（清零 padding 行），否则烤出来的是一块焦炭，还可能熏到旁边的面包（真实模型里会污染跨行的计算）。

烤好之后，必须**把面包从烤盘上拿下来装袋**再交给客人（clone）——如果你把"烤盘上的那几格"直接指给客人，下一炉一开烤，客人手里的面包就变成了别人的。

来了 7 个面团？最大的烤盘只有 6 格 → 只能手工一个个烤（eager）。

类比 · 卡拉 OK 伴奏带

CUDA graph 像一盘**录好的伴奏带**：按下播放（replay），它就一模一样地再放一遍。你可以换一个唱歌的人（把新数据拷进同一个话筒 = 静态 buffer），但**不能在播放时让乐队改编**。

如果某位客人要求"我这首要用我自己的随机节奏"（请求自带 `torch.Generator`），伴奏带不会报错，只会**照旧播放录制时的节奏**——你的要求被悄悄忽略了。所以这种请求必须请现场乐队（`force_eager`）。

框架视角：数据流（shape trace）

这是一个"CPU 模拟的 GPU runner"，没有真正的并发。按一次 replay 调用走一遍（以交互图第 3 步 `bs=2`、bucket 3、`hidden=2` 为例）：

© 2026 kzhou003

#	张量	形状	发生了什么
1	`x`	`[2, 2]`	调用方的新 batch（Python 参数，graph 看不到它）
2	选 bucket	标量 3	最小的 ≥ 2 的 bucket；没有 / `force_eager` → 直接 `fn(x)`
3	`static_in[:2] ← x`	`[2, 2]` 写入 `[6, 2]`	原地拷贝进录制时的那块内存
4	`static_in[2:3] ← 0`	`[1, 2]`	padding 行清零（之前是上一次 bs=5 留下的 5）；第 3..5 行在 bucket 外，不碰
5	`graphs[3].replay()`	读 `static_in[:3]`，写 `static_out[:3]`	输出 5, 5, -1（padding 行 = fn(0)）
6	`static_out[:2].clone()`	`[2, 2]`	只取真实行，并且拷贝出来，和静态 buffer 脱钩

易错点

- **重新赋值而不是拷贝**：把 `self.static_in` 换成一个新 tensor，graph 还在读旧的那块内存。
- **忘记清零 padding 行**：本题的 fn 逐行独立，脏行只让 padding 输出变垃圾；真实模型里有跨行交互（MoE 路由统计、NaN 扩散、整 batch 归约），会污染真实行。
- **返回 view**：单次调用看不出来，第二次调用后第一次的结果被悄悄改写。
- **eager 分支碰了静态 buffer**，或者在 eager 时更新了 `last_bucket`。
- **运行时现场 capture**：capture 很慢、要 warmup 和同步，会造成延迟尖刺；所有图都必须在启动期录好。
- **capture 顺序**：从大到小（CUDA 上多张图共享一个 memory pool，大图先占好内存，小图复用）。

交互图：`nano-vllm-omni-drills/hints/p17_overview.html`

对照源码： `worker/gpu_ar_model_runner.py:634-690` `worker/gpu_model_runner.py:243-272` `worker/base.py:140-147`

笔记 / 我踩到的坑：

Capstone：把积木串成一个 mini omni 服务（半天到一天）

用你自己的实现，在 CPU 上组装一个 3 个 stage 的玩具流水线，端到端地验证并发语义：

1. **进程与引擎。**起三个 `multiprocessing`（spawn）子进程，每个进程里跑 P10 的 busy loop。引擎内部用 P07 的 scheduler，P06 的 block manager 可选。"模型"可以先用玩具函数代替；也可以把 stage 0 换成 M08 的 TinyThinker。
2. **控制面。**父进程里起一个 orchestrator 线程，用 P03 编排三个 stage，结果经 P01 交给前端。abort 用 P04，RPC 用 P05。
3. **数据面。**stage 之间用 P12 加 P13 直接传数据，接收端用 P11 停靠请求，codec 的切窗用 P14。启动流程用 P16。

© 2026 kzhou003

4. **验收。**发 100 个并发请求，随机让 10% 中途 abort，并随机 kill 一次 stage2。要求：每个请求恰好有一个终态，`/dev/shm` 中没有残留，进程退出后没有僵尸进程。

完成之后，进入 Part II 的大作业：把玩具模型换成真实的 Qwen2.5-Omni-3B，放到单张 5090 上运行。

© 2026 kzhou003

附录 A · 三级提示（卡住 15 分钟以上再看；L1 → L2 → L3 逐级剧透）

A-1 · RMSNorm 与多模态 RoPE（interleaved / chunked）

Level 1 · I/O 示例

rms_norm：`x = [[3, 4], [1, 1]]`（fp32），`weight = [1, 2]`，`eps = 0`。

```
row0: mean(x²) = (9+16)/2 = 12.5, rsqrt = 1/3.5355
      x̂ = [0.8485, 1.1314]  -> * weight -> [0.8485, 2.2627]
row1: mean(x²) = 1 -> x̂ = [1, 1] -> * weight -> [1.0000, 2.0000]
```

注意是对**最后一维**求 mean（每行独立），输出形状与 x 相同。

rope_inv_freq：`rope_inv_freq(8, 10000.0)` → `[1.0000, 0.1000, 0.0100, 0.0010]`（长度 D/2 = 4，即 10000^(-0/8), 10000^(-2/8), 10000^(-4/8), 10000^(-6/8)）；`rope_inv_freq(6, 100.0)` → `[1.0000, 0.2154, 0.0464]`。

mrope_cos_sin 的"轴分配"可视化：取 `inv_freq = ones(10)`（D = 20），单个 token，`positions = [[1],[2],[3]]`（t=1, h=2, w=3）。这样每个频率的角度就等于它所用轴的位置，`acos(cos)` 直接告诉你第 j 个频率跟哪个轴转：

section / 布局	acos(cos[0])（长度 20）
`[4,3,3]`, interleaved=True	`1 2 3 1 2 3 1 2 3 1 | 1 2 3 1 2 3 1 2 3 1`
`[4,3,3]`, interleaved=False	`1 1 1 1 2 2 2 3 3 3 | 1 1 1 1 2 2 2 3 3 3`
`[6,2,2]`, interleaved=True	`1 2 3 1 2 3 1 1 1 1 |` （后半同前半）

最后一行说明：H/W 只占前 3·section 个"槽位"，之后全是 T。后半段是前半段的复制（`cat([f, f])`）。

rotate_half / apply_rotary：`rotate_half([1, 2, 3, 4]) = [-3, -4, 1, 2]`。取 D = 4，`inv_freq = rope_inv_freq(4, 10000) = [1, 0.01]`，纯文本位置 p = 2（三轴都是 2，section=[1,1,0]，interleaved）：

```
cos = [cos2, cos0.02, cos2, cos0.02] = [-0.4161, 0.9998, -0.4161, 0.9998]
sin = [ 0.9093, 0.0200, 0.9093, 0.0200]
apply_rotary(x=[[[1,0,0,0]]]) = [[[-0.4161, 0, 0.9093, 0]]]     # 维度 0 与维度 2 组成一对旋转
apply_rotary(x=[[[1,2,3,4]]]) = [[[-3.1440, 1.9196, -0.3391, 4.0392]]]
```

x 的形状是 `[T=1, heads=1, D=4]`，cos/sin 是 `[T=1, D=4]`。

关键问题

1. RMSNorm 里"先转回 x.dtype 再乘 weight"和"在 fp32 里乘完 weight 再转回"在 fp32 输入下有区别吗？在 bf16 输入下呢？为什么？
2. `torch.arange(0, head_dim, 2)` 得到的是什么？除以 head_dim 之后的指数范围是多少？inv_freq 的长度是多少？
3. `positions` 的形状是 `[3, T]`，`inv_freq` 是 `[D/2]`，怎样用一次广播得到所有轴的 `freqs[3, T, D/2]`？
4. interleaved 布局下，频率下标 j = 7、section = [4,3,3] 时它属于哪个轴？若 section = [6,2,2] 呢？
5. rotate_half 中，维度 j 是和维度 j+1 配对，还是和维度 j + D/2 配对？这和 `cat([f, f])`（而非把每个频率重复两次相邻放置）有什么关系？
6. 如果所有 token 都是纯文本（三个轴的位置相同），interleaved 与 chunked 的 cos/sin 应该是什么关系？为什么？
7. `apply_rotary` 里 x 是 `[T, heads, D]`，cos 是 `[T, D]`，要怎样 reshape 才能正确广播到所有 head？

细节

- **"normalize in fp32, cast back, THEN multiply by weight"**：计算 `x * rsqrt(mean(x², -1) + eps)` 时全程在 fp32；得到的归一化结果先 `.to(x.dtype)`，再与 weight（通常和 x 同 dtype）相乘。bf16 下这两种顺序会有 1 ulp 量级的差异，例如某组 bf16 输入，正确顺序得到 `1.9140625`，"fp32 里先乘 weight"得到 `1.90625` ——数值"差不多"但不 bitwise 一致。
- **rope_inv_freq 必须是 fp32**：`torch.arange` 默认是 int64，`int / int` 在 PyTorch 里会得到 float，但 dtype 依赖默认 dtype 设定；显式指定 `dtype=torch.float32` 最稳。`theta ** tensor` 是 Python 标量的逐元素幂。
- **section 的含义**：`mrope_section = [s_t, s_h, s_w]`，和为 D/2，是"每个轴分到多少个**频率**"（不是多少个维度；维度数是 2 倍）。chunked 下就是连续三段长度；interleaved 下 H 占据 j = 1, 4, 7, ...（共 s_h 个，即 j%3==1 且 j < 3·s_h），W 占据 j = 2, 5, 8, ...（j%3==2 且 j < 3·s_w），剩下的全给 T。
- **cos/sin 的形状** 是 `[T, head_dim]`：先在 `[T, D/2]` 上完成轴选择，再 `cat([f, f], -1)` 拼成 D，最后取 `.cos()` / `.sin()`。这对应 neox 风格的"前半/后半配对"。
- **dtype**：cos/sin 在 fp32 中计算；`apply_rotary` 遇到 bf16 的 x 时，应先把 cos/sin 转成 x 的 dtype 再相乘（与 HF 一致）。

© 2026 kzhou003

- **Qwen3 vs Qwen2.5**：两者 inv_freq、rotate_half、apply 公式完全相同，唯一区别是"频率 j 用哪个轴的位置"——这就是 `interleaved` 这个 bool 的全部含义。

Level 2 · 思路与常见坑

- **拆分**：五个函数相互独立又层层组合：`rope_inv_freq` → `mrope_cos_sin` → `apply_rotary`（内部调用 `rotate_half`）。`rms_norm` 独立。建议按这个顺序写，每写完一个就用 io 例子里的数字自测。
- **rms_norm**：三步—— `x32 = x.float()`；`x32 * torch.rsqrt(x32.pow(2).mean(-1, keepdim=True) + eps)`；`.to(x.dtype)` 后再乘 weight。常见错误：忘记 `keepdim=True`（形状广播错或报错）、eps 放在 sqrt 外面、对全部维度求 mean。
- **mrope_cos_sin 的核心是"选择"而非"计算"**：先一次性算出三条轴各自的 `freqs[a]`（形状 `[T, D/2]`），然后构造一个 `[T, D/2]` 的结果 f，其中每一列 j 从某个轴的 freqs 里拷过来。只是列的来源不同，没有额外数学运算。
- **interleaved 的实现思路**：以 T 轴为底（拷贝一份，避免改到原 tensor），再用步长为 3 的切片覆盖 H 的列（起点 1）和 W 的列（起点 2），切片终点由对应 section 决定。也可以先构造一个长度 D/2 的"轴编号"向量 `axis_of_j`，再用 `gather` /高级索引选取。
- **chunked 的实现思路**：维护一个起点游标，依次从 freqs[0]、freqs[1]、freqs[2] 中切出长度 s_t、s_h、s_w 的列段，拼起来。注意切的是**同一个列区间**：H 段取的是 freqs[1] 的第 s_t..s_t+s_h 列，而不是 freqs[1] 的第 0..s_h 列。
- **rotate_half**：用 `chunk(2, dim=-1)` 把最后一维切成前后两半 a、b，返回 `cat([-b, a], -1)`。常见错误：写成 `[b, -a]`（旋转方向反了，sin 的符号整体翻转），或按奇偶位切（GPT-J 风格，与 cat([f,f]) 不匹配）。
- **apply_rotary 的广播**：cos/sin 是 `[T, D]`，x 是 `[T, heads, D]`，需要在中间插入 head 维：`cos[:, None, :]`。如果直接相乘，PyTorch 会把 cos 的 T 维对齐到 x 的 heads 维——当 T == heads 时不会报错但结果是错的，这是最阴险的 bug。
- **自测技巧**：用 `inv_freq = ones` + 互不相同的小整数位置（如 1/2/3）跑 mrope_cos_sin，再取 `acos`，就能肉眼看到每列来自哪个轴；用三轴相同的位置验证两种布局都退化为普通 RoPE。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key question 答案：

- fp32 输入下 `.to(x.dtype)` 是 no-op，两种顺序结果相同；bf16 下，"先 round 到 bf16 再乘 bf16 weight"和"fp32 乘完再 round"舍入点不同，可能差 1 ulp。
- `arange(0, D, 2)` = `[0, 2, …, D-2]`，除以 D 后指数在 [0, 1)，inv_freq 长度 D/2，单调递减，从 1 到接近 1/θ。
- `positions.float()[:, :, None] * inv_freq[None, None, :]` → `[3, T, D/2]`。
- j = 7：7 % 3 == 1，section=[4,3,3] 时 3·3 = 9 > 7 → H；section=[6,2,2] 时 3·2 = 6 ≤ 7 → T。
- 维度 j 与 j + D/2 配对；因为 cos/sin 是 `cat([f, f])`，第 j 列和第 j+D/2 列有同一个角度，rotate_half 正好把这两列组成一个二维旋转。
- 三轴位置相同 → 三条 freqs 完全一样 → 无论怎么选列，结果都是普通 1D RoPE，两种布局 bitwise 相同。
- `cos[:, None, :]`（`[T, 1, D]`）。

mrope_cos_sin 形状追踪（T 个 token，D = head_dim，half = D/2）：

```
positions [3, T] --float--> [3, T, 1] * inv_freq [1, 1, half] = freqs [3, T, half]
选择列                                                          -> f     [T, half]
cat([f, f], -1)                                                  -> emb   [T, D]
emb.cos(), emb.sin()                                             -> [T, D], [T, D]
```

骨架 检查：`slice(1, 9, 3)` 选中 1, 4, 7（3 个 = s_h）；`slice(2, 9, 3)` 选中 2, 5, 8（3 个 = s_w）。
rotate_half / apply_rotary 骨架 验算 io 例子：x = [1,2,3,4]，cos = [c0,c1,c0,c1]，sin = [s0,s1,s0,s1]，rotate_half(x) = [-3,-4,1,2]：`out[0] = 1·c0 − 3·s0 = −0.4161 − 2.7279 = −3.1440`，`out[2] = 3·c0 + 1·s0 = −1.2484 + 0.9093 = −0.3391` ——正是二维旋转矩阵作用在 (x0, x2) 这一对上。
最终检查清单：(1) bf16 下 rms_norm 先 cast 再乘 weight；(2) inv_freq 显式 fp32；(3) interleaved 用 `clone()` 再覆盖，切片终点是 `3*section[axis]`；(4) chunked 的列区间是累积偏移；(5) `cat([f, f])` 而不是 `repeat_interleave`；(6) apply_rotary 插入 head 维并对齐 dtype。

A-2 · Thinker Attention：QK-norm + GQA + KV cache

Level 1 · I/O 示例

一个能手算的 toy：hidden = 2，`num_heads = num_kv_heads = 1`，`head_dim = 2`，q/k/v/o_proj 全是 2×2 单位阵，`q_norm = k_norm = [1, 1]`，`eps = 0`；cos 全 1、sin 全 0（相当于不旋转）。

```
x = [[1, 0],
     [0, 1],
     [3, 3]]                      # T = 3
QK-norm 之后: q = k = [[1.414, 0], [0, 1.414], [1, 1]]     # 每行除以自身 RMS
v 不做 norm:  v = x  = [[1, 0], [0, 1], [3, 3]]
attention_forward(w, x, cos, sin) =
    [[1.0000, 0.0000],            # token0 只能看见自己
```


© 2026 kzhou003

```
      [0.1956, 0.8044],            # token1: scores [0, 1.414] -> softmax [0.1956, 0.8044]
      [1.5768, 1.5768]]            # token2: scores [1, 1, 1.414] -> softmax [0.2846, 0.2846, 0.4307]
```

scale = $D^{-0.5}$ = 1/1.414，例如 token1 对自己的 score = (1.414·1.414)/1.414 = 1.414。
同一个 toy，分两次调用（chunked prefill / decode）：

```
cache = KVCache()
a = attention_forward(w, x[:2], cos[:2], sin[:2], cache)   # -> [[1, 0], [0.1956, 0.8044]]
len(cache) == 2; cache.k == [tensor of shape [2, 1, 2]]
b = attention_forward(w, x[2:], cos[2:], sin[2:], cache)   # -> [[1.5768, 1.5768]]
len(cache) == 3; cache.k 里有两个块: [2,1,2] 和 [1,1,2]
```

`cat([a, b])` 与一次性 prefill 的结果逐行相同。注意第二次调用的 cos/sin 只有 1 行——它属于绝对位置 2 的那个新 token。
两种典型错误会得到什么（同一 toy）：

错误	结果
完全没有 causal mask	`[[1.5665, 1.1690], [1.1690, 1.5665], [1.5768, 1.5768]]`——只有最后一行碰巧正确
decode 时 mask 用块内下标（q_pos = 0 而不是 2）	第三个 token 只能看见 key 0，输出 `[1, 0]` 而不是 `[1.5768, 1.5768]`

关键问题

1. q_proj 的形状是 `[H·D, hidden]`。`x @ q_proj.T` 得到 `[T, H·D]` 后，怎样 view 成 `[T, H, D]`，才能保证“第 h 个 head”是连续的 D 列?
2. QK-norm 是对哪一维做的? q_norm 的形状是 `[D]`，它怎样广播到 `[T, H, D]`？v 要不要 norm?
3. cos/sin 只有 T 行（只对应新 token），而 K 有 S = past + T 行。cache 里的旧 K 需要重新做 RoPE 吗？为什么?
4. 第 i 个新 token（块内下标 i）的绝对位置是多少？它能看见哪些 key 的下标范围?
5. 要把 `[S, KVH, D]` 的 k 变成每个 Q head 都有对应的 `[S, H, D]`，应该用 `repeat` 还是 `repeat_interleave`？两者在 H=4, KVH=2 时分别给出什么映射?
6. “softmax in fp32” 指的是哪几步要在 fp32 里做？最后在哪里转回 x.dtype?
7. 应该先把新 k/v 追加进 cache 再做 attention，还是先 attention 再追加？两种顺序哪一个能让 token 看见它自己?

细节

- **“RMSNorm EACH HEAD ... over head_dim”**：QK-norm 不是对整个 hidden 做 norm，而是把 q 看成 `[T, H, D]`，对最后一维 D 独立地算 RMS。所有 head 共享同一个 `q_norm` 权重 `[D]`（k 同理用 `k_norm`）。eps 取 `w.eps`。
- **顺序：proj → QK-norm → RoPE**。先 RoPE 再 norm 数值上不同（RMS 在旋转下不变，但逐维的 weight 不是旋转不变的），对拍就会失败。
- **“rows belong to the NEW tokens”**：调用方已经按绝对位置算好了 cos/sin 并只切出新 token 那几行传进来；函数内部不需要、也不应该自己算位置。cache 中保存的是**已经 RoPE 过**的 k（以及原样的 v）。
- **KVCache 的结构**：`k`、`v` 是块的 list，每块 `[t, KVH, D]`；`len(cache)` 是所有块的 t 之和。所以“past = len(cache)”必须在你 append 之前读取。
- **causal 按绝对位置**：设 past = 追加前的 cache 长度，新 token i 的绝对位置 p = past + i，可见的 key 是 0..p。整段 prefill（past=0）时退化为普通下三角；decode（T=1）时新 token 能看见全部 S 个 key。
- **输出**：attention 结果 `[T, H, D]` 先 reshape 成 `[T, H·D]`（head 维在前、D 在后，与 q 的 view 方式对称），再 `@ o_proj.T` 得到 `[T, hidden]`。

Level 2 · 思路与常见坑

- **步骤拆分**：(1) 三个投影 + view；(2) 一个小的 per-head RMSNorm helper（可直接复用 M01 的思路：fp32 归一化 → 转回 → 乘 weight）；(3) rotate_half + RoPE helper，注意 cos/sin 需要在 head 维上广播；(4) cache 处理得到 past、全量 K/V；(5) GQA 展开；(6) scores + mask + softmax + 加权求和；(7) reshape + o_proj。
- **cache 处理的顺序**：先读 `past = len(cache)`，再 append 本次的 k/v，再 `torch.cat` 全部块得到 `[S, KVH, D]`。如果先 append 再读 len，past 会多算 T，mask 整体偏移，第一个新 token 就能“偷看”后面的 token。cache 为 None 时 past = 0、K/V 就是本次的 k/v。
- **GQA 展开**：`k.repeat_interleave(H // KVH, dim=1)` 把 `[S, KVH, D]` 变成 `[S, H, D]`，得到 0,0,1,1 的映射。`repeat` / `expand` +错误 reshape 得到 0,1,0,1，在 H == KVH 的情况下看不出来，但在真实 GQA 下全错。
  GQA 映射（符号例子）：`num_heads = 8, num_kv_heads = 2` → 每组 4 个 Q head：

q head h	0	1	2	3	4	5	6	7
kv head h // 4	0	0	0	0	1	1	1	1

（不是 `h % 2` 那种 0,1,0,1,... 的交替。）

© 2026 kzhou003

- **scores 的计算**：最清晰的是 einsum：`scores[h, t, s] = Σ_d q[t,h,d]·K[s,h,d]`，即 `einsum("thd,shd->hts")`，再乘 `D ** -0.5`。也可以先 transpose 成 `[H, T, D]` 和 `[H, S, D]` 后用 `@`。在 fp32 里做（q、K 先 `.float()`）。
- **mask 构造**：`q_pos = arange(past, past+T)[:, None]`，`k_pos = arange(S)[None, :]`，在 `k_pos > q_pos` 的位置填 `-inf`，形状 `[T, S]`，自动广播到 `[H, T, S]`。由于每个 query 至少能看见 key 0，不会出现整行 -inf 导致 NaN。
- **softmax 与加权**：`softmax(scores, dim=-1)`（在 key 维 S 上），再与 V 相乘：`einsum("hts,shd->thd")`，然后 `.to(x.dtype)`。常见错误：softmax 在 dim=-2 上做（每列和为 1 而非每行），在 toy 例子里 token0 的输出会不再是 v0。
- **如何自测三种调用一致**：随机一个 T=6 的输入，(a) 一次调用；(b) 一个 cache，3+3 两次调用；(c) 一个 cache，6 次 T=1 调用；三者 cat 后应 allclose。cos/sin 按同样的切片传入。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key question 答案：

- `(x @ q_proj.T).view(T, H, D)`：q_proj 的行按 head 分块排列（前 D 行是 head 0），所以 view 后最后一维连续 D 个数属于同一个 head。
- 对最后一维 D；`[D]` 自动广播到 `[T, H, D]`；v 不 norm。
- 不需要：cache 里存的就是当时已用正确绝对位置旋转过的 k，位置不会变，所以永远不用重算。
- 绝对位置 = past + i；可见 key 下标 0..past+i（含）。
- `repeat_interleave`：H=4, KVH=2 → kv 映射 [0,0,1,1]（正确）；`repeat` → [0,1,0,1]（错误）。
- q·k、缩放、mask、softmax、以及与 V 的乘法都在 fp32；乘完 V 后转回 x.dtype，再做 o_proj。
- 先追加再 attention：这样新 token 的 key 也在 K 里，mask 允许 s == p，token 能看见自己。

形状追踪（H=4, KVH=2, D=8, hidden=32, 已缓存 past=5，新 T=2）：

```
x           [2, 32]
q           [2, 4, 8]     k, v [2, 2, 8]
QK-norm     same shapes
RoPE        q, k with cos/sin [2, 8] -> [2, 1, 8] broadcast
cache cat   K, V [7, 2, 8]            (S = 5 + 2)
GQA         K, V [7, 4, 8]
scores      [4, 2, 7]                  mask [2, 7]:  row0 sees s ≤ 5, row1 sees s ≤ 6
probs @ V   [2, 4, 8] -> reshape [2, 32] -> @ o_proj.T -> [2, 32]
```

用 toy 例子逐行核对：token2（past=2，T=1 的 decode）：q = [1,1]，K = [[1.414,0],[0,1.414],[1,1]]，scores = [1.414, 1.414, 2]/1.414 = [1, 1, 1.414]，mask：q_pos = 2，k_pos = [0,1,2] 全部可见；softmax = [0.2846, 0.2846, 0.4307]；输出 = 0.2846·[1,0] + 0.2846·[0,1] + 0.4307·[3,3] = [1.5768, 1.5768]。如果你得到 `[1, 0]`，说明 q_pos 用了 0；如果第一行不是 `[1, 0]`，说明 mask 缺失或 softmax 维度错了。

与 serving 的联系：这里的"past = len(cache)，新 token 绝对位置 = past + i"正是 vLLM 里 `seq_lens` / `query_start_loc` 携带的信息；paged attention 只是把 `torch.cat(cache.k)` 换成了按 block table 读取，mask 逻辑不变。

A-3 · Sparse MoE：路由、按 expert 分组计算、shared expert

Level 1 · I/O 示例

route：E = 4，H = 4，`router = eye(4)`（所以 logits 就等于 x），`top_k = 2`。

```
x = [[2.0, 1.0, 0.0, -1.0],
     [0.0, 0.5, 3.0, -0.5]]
softmax over ALL 4 experts:
     [[0.6439, 0.2369, 0.0871, 0.0321],
      [0.0428, 0.0706, 0.8605, 0.0260]]
top-2 ids     = [[0, 1], [2, 1]]
norm_topk_prob=True  -> weights [[0.7311, 0.2689], [0.9241, 0.0759]]   # 0.6439/(0.6439+0.2369) ...
norm_topk_prob=False -> weights [[0.6439, 0.2369], [0.8605, 0.0706]]
```

返回的 weights 是 x.dtype（例如 x 是 bf16 时也是 bf16），ids 是 int64。

按 expert 分组的 bookkeeping：N = 3，k = 2，E = 5，假设路由得到 `ids = [[2, 0], [1, 2], [2, 3]]`。

```
flat expert id (N·k = 6):  [2, 0, 1, 2, 2, 3]
flat token id:             [0, 0, 1, 1, 2, 2]
stable argsort by expert:  [1, 2, 0, 3, 4, 5]
per-expert counts:         [1, 1, 3, 1, 0]      # expert 4 收到 0 个 token -> 完全跳过
expert 0 -> pairs [1]        -> tokens [0]
expert 1 -> pairs [2]        -> tokens [1]
expert 2 -> pairs [0, 3, 4]  -> tokens [0, 1, 2]   # 一次矩阵乘处理 3 行
expert 3 -> pairs [5]        -> tokens [2]
```


© 2026 kzhou003

moe_forward 端到端（标量版）：H = 1，每个 expert 的中间维 I = 1，E = 3，top_k = 2，norm_topk_prob = True。

```
router = [[1], [0], [-1]]
w_gate = [1, 2, 0.5],  w_up = [1, 1, 2],  w_down = [1, -1, 3]   # 每个都是 [E, 1, 1]
x = [[2.0], [-1.0]]

Expert_e(h) = silu(g·h) · (u·h) · d
         e=0        e=1        e=2
x=2:   3.5232    -7.8561     8.7727
x=-1:  0.2689    -0.2384     1.1326

token0: logits [2, 0, -2] -> ids [0, 1], w [0.8808, 0.1192]
        out = 0.8808·3.5232 + 0.1192·(-7.8561) = 2.1667
token1: logits [-1, 0, 1] -> ids [2, 1], w [0.7311, 0.2689]
        out = 0.7311·1.1326 + 0.2689·(-0.2384) = 0.7639
moe_forward(x, w) = [[2.1667], [0.7639]]
```

再加一个 shared expert（gate_proj = up_proj = down_proj = [[1]]，`gate = [[0]]` → sigmoid = 0.5）：SharedExpert(x) = silu(x)·x = [3.5232, 0.2689]，结果变为 `[[3.9283], [0.8984]]`（= 2.1667 + 0.5·3.5232，0.7639 + 0.5·0.2689）。

关键问题

1. softmax 是在全部 E 个 expert 上做再取 top-k，还是先取 top-k 再在 k 个里做 softmax？在 norm_topk_prob=True 时两者结果是否相同？在 False 时呢？
2. top-k 之后 weights 的 dtype 应该是什么？在哪一步转换？
3. N 个 token、每个 top_k 个 slot，展平之后共有多少个 (token, slot) 对？怎样构造“每个对属于哪个 token”的向量，使之与 `ids.reshape(-1)` 一一对应？
4. 排序之后，怎样知道 expert e 的那一段在排序数组中的起止位置？
5. 同一个 token 的两个 slot 可能落到不同的 expert，最终怎样把两份结果加回同一行？如果把全部 (token, slot) 对一次性用 `out[all_toks] += y_all` 写回，会发生什么？
6. `w_gate` 形状是 `[E, I, H]`，对一批 `h[n_e, H]` 做 gate 投影应该乘 `w_gate[e]` 还是它的转置？
7. shared expert 的门控是一个标量还是一个向量？`sigmoid(x @ gate.T)` 的形状是多少，怎样与 `[N, H]` 的 SharedExpert 输出相乘？

细节

- **“softmax … over ALL experts in fp32 → top-k → renormalize”**：概率先在全部 E 个 expert 上归一，再选最大的 k 个；若 `norm_topk_prob`，把这 k 个再除以它们的和使之和为 1。若不重新归一，这 k 个权重的和小于 1（上面例子里 0.6439+0.2369 = 0.8808）。
- **dtype 约定**：路由概率在 fp32 中算（logits 可以先 `.float()` 或用 softmax 的 `dtype=` 参数），renorm 也在 fp32 里做，最后 weights 转回 `x.dtype`。expert ids 是 topk 返回的 int64 下标。
- **SwiGLU expert**：`Expert(h) = (silu(h Wg^T) * (h Wu^T)) Wd^T`，Wg/Wu 是 `[I, H]`，Wd 是 `[H, I]`，所以 `h[n, H] @ Wg.T → [n, I]`，逐元素相乘后 `@ Wd.T → [n, H]`。stacked 后用 `w_gate[e]` 取第 e 个 expert。
- **“the serving way”**：不是“对每个 token 循环 k 次”，也不是“对每个 expert 在全部 N 个 token 上算再用 mask 选”，而是每个 expert 只在路由给它的那批 token 上算**一次**；没有 token 的 expert 不做任何计算。这就是 FusedMoE 的 grouped GEMM 思路。
- **scatter-add**：每个 expert 的输出先乘以对应 (token, slot) 的路由权重，再按 token 下标加回 `out`。同一 token 在同一个 expert 里只会出现一次（top-k 的 id 互不相同），但在不同 expert 之间会多次出现，所以必须是“加”，且要用支持重复下标的累加 API。
- **shared expert（talker）**：每个 token 都过一个普通的 SwiGLU（`gate_proj/up_proj/down_proj`），再乘以每个 token 一个标量门 `sigmoid(x @ gate.T)`（`[N, 1]`），然后加到路由结果上。`w.shared is None` 时（thinker）跳过。

Level 2 · 思路与常见坑

- **route** 三行：logits → fp32 softmax（dim=-1）→ `torch.topk(probs, k, dim=-1)`；可选 renorm（`keepdim=True` 的 sum）；返回前把 weights 转回 x.dtype。易错：先 topk logits 再 softmax（在 norm_topk_prob=True 时数学上等价，但在 False 时完全不同，而且权重大小也变了）。
- **展平**：`flat_e = ids.reshape(-1)`，`flat_w = weights.reshape(-1)`，`flat_tok = arange(N).repeat_interleave(k)`。三者长度都是 N·k，第 p 个元素描述同一个 (token, slot) 对。易错：用 `arange(N).repeat(k)` 得到 0,1,2,0,1,2，对不上 row-major 的 reshape。
- **分组**：`order = argsort(flat_e, stable=True)` 把同一 expert 的对排到一起；`counts = bincount(flat_e, minlength=E)` 给出每段长度；用一个游标 start 累加即可得到每段 `order[start:start+n]`。也可以用 `(flat_e == e).nonzero()` 逐 expert 取下标——更直观，但每个 expert 都扫一遍全数组。
- **每个 expert 的计算**：`toks = flat_tok[sel]`，`h = x[toks]`（`[n_e, H]`），SwiGLU 得 `y[n_e, H]`，再乘 `flat_w[sel][:, None]`。注意权重的 dtype 与 x 一致（route 已转回）。
- **scatter-add**：`out = zeros_like(x)`，`out.index_add_(0, toks, y_weighted)`。易错：`out[toks] += y` 在 toks 有重复时只保留最后一次写入；在单个 expert 内 toks 不重复所以“看起来”对，但如果你一次性对全部对做高级索引赋值就会丢结果。
- **跳过空 expert**：`if n == 0: continue`。这既是性能要求，也避免对空 tensor 做无意义的 matmul。注意游标 start 在 n == 0 时不需要前进（加 0 也无妨）。

© 2026 kzhou003

- **shared expert**：在全部 N 个 token 上算 SwiGLU，乘以 `torch.sigmoid(x @ shared.gate.T)`（`[N, 1]`，自动沿 H 广播），加到 out 上。
- **批不变性自测**：随机 x[6, H]，比较 `moe_forward(x)[i]` 与 `moe_forward(x[i:i+1])[0]`。若不一致，通常是 flat_tok 构造错或 scatter 丢了累加。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key question 答案：

- 先全量 softmax 再 top-k。norm_topk_prob=True 时，先 softmax 再 renorm 等价于对 top-k logits 做 softmax（分母都会被消掉）；False 时不等价——全量 softmax 的 top-k 权重和 < 1。
- x.dtype；在 renorm 之后、return 之前转换。
- N·k 个；`arange(N).repeat_interleave(k)` → 0,0,1,1,2,2（k=2）。
- bincount 得到每个 expert 的计数，按 expert 顺序累加计数就是各段起点（前缀和）。
- `index_add_`（或 `scatter_add_`）。`out[idx] += y` 是"读-加-写"，同一次调用里 idx 有重复时后写覆盖先写；在逐 expert 的循环里 toks 恰好不重复所以能用，但一旦你把所有 (token, slot) 对一次性写回就会丢贡献，`index_add_` 两种情况都正确。
- `h @ w_gate[e].T`：`[n_e, H] @ [H, I] → [n_e, I]`。down：`[n_e, I] @ w_down[e].T`（`[I, H]`）→ `[n_e, H]`。
- 每个 token 一个标量：`x @ gate.T` 是 `[N, 1]`，sigmoid 后直接与 `[N, H]` 广播相乘。

形状追踪（N=5, H=16, E=8, I=32, k=2）：

```
logits    x[5,16] @ router.T[16,8]      -> [5, 8]
probs     softmax fp32                   -> [5, 8]
topk      weights [5, 2], ids [5, 2]
flatten   flat_e / flat_w / flat_tok     -> [10] each
order     argsort(flat_e, stable)        -> [10]
counts    bincount(minlength=8)          -> [8]   (sum = 10)
expert e  h = x[toks]                    -> [n_e, 16]
          gate/up                        -> [n_e, 32]
          down                           -> [n_e, 16]
scatter   out.index_add_(0, toks, ·)     -> out [5, 16]
shared    sigmoid(x @ gate.T) [5,1] * SwiGLU_s(x) [5,16]
```

用标量例子核对：token0（x=2）路由到 expert 0、1，权重 0.8808 / 0.1192；token1（x=-1）路由到 expert 2、1，权重 0.7311 / 0.2689。分组后 expert 0 处理 [token0]，expert 1 处理 [token0, token1]，expert 2 处理 [token1]。expert 1 的一次 matmul 输出 `[-7.8561, -0.2384]`，乘权重 `[0.1192, 0.2689]` 后分别 index_add 回第 0、1 行。最终 `[[2.1667], [0.7639]]`。若 token0 得到 `-0.9365`（= 0.1192·(-7.8561)，只剩最后处理的 expert 1 的贡献），说明你用了赋值 `out[toks] = y` 而不是累加。

与 serving 的联系：argsort + bincount 对应 vLLM FusedMoE 里的 `moe_align_block_size`（把 token 按 expert 对齐并 pad 到 block），循环里的每个 expert matmul 对应 grouped GEMM 的一个 group；Expert Parallel 时，`pair_token` / `pair_expert` 就是 all-to-all dispatch 的路由表。

A-4 · 多模态拼接：placeholder 计数、embedding 合并、DeepStack

Level 1 · I/O 示例

audio_token_count：

num_mel_frames n	完整窗口 n // 100	余数 r	f → 第二次 → 第三次	total
0	0	0	0 → 0 → 0	**0**
50	0	50	25 → 13 → 7	**7**
99	0	99	50 → 25 → 13	**13**
100	1	0	0 → 0 → 0	**13**
101	1	1	1 → 1 → 1	**14**
330	3	30	15 → 8 → 4	**43**

传入 LongTensor 时逐元素计算：`audio_token_count(torch.tensor([50, 100, 330]))` → `tensor([7, 13, 43])`。注意 99 与 100 都是 13，101 跳到 14。

merge_multimodal：H = 2，image 占位 id = 90，audio 占位 id = 91。

```
input_ids   = [7, 90, 90, 8, 91, 91, 9]
text_embeds = [[0,1],[2,3],[4,5],[6,7],[8,9],[10,11],[12,13]]
```


© 2026 kzhou003

```
mm_embeds   = {"image": [[100,100],[200,200]], "audio": [[-1,-1],[-2,-2]]}

inputs_embeds = [[  0,   1],
                 [100, 100],   # image #0
                 [200, 200],   # image #1
                 [  6,   7],
                 [ -1,  -1],   # audio #0
                 [ -2,  -2],   # audio #1
                 [ 12,  13]]
masks = {"image": [F, T, T, F, F, F, F],
         "audio": [F, F, F, F, T, T, F]}
```

调用之后 `text_embeds` 必须仍是原来的值。纯文本输入（没有任何占位符、`mm_embeds = {}`）时，输出等于 text_embeds 的一份拷贝，masks 中每个模态都是全 False。

错误情况：

输入	结果
`ids = [90, 90, 5]`，`mm_embeds = {"image": 1 行}`	`ValueError`（2 个占位 vs 1 行 embedding）
`ids = [91, 5]`，`placeholder_ids = {"audio": 91}`，`mm_embeds = {}`	`ValueError`（有占位符但没有 embedding）
`ids = [7, 8]`，`mm_embeds = {}`	正常返回（没有占位符，缺 key 合法）

apply_deepstack / run_layers：

```
apply_deepstack(hidden=zeros(4, 2), visual_mask=[F, T, T, F], visual_embeds=[[1,1],[2,2]])
  -> [[0,0],[1,1],[2,2],[0,0]]       # hidden 本身仍是全 0

layers = [lambda t: t*2, lambda t: t+10, lambda t: t*3]
hidden = ones(3, 1);  visual_mask = [T, F, T]
deepstack_embeds = [[[1],[5]], [[100],[200]]]       # 2 个 level，每个 [n_vis=2, 1]

row:          0      1      2
layer0 (*2):  2      2      2
+ level 0:    3      2      7
layer1 (+10): 13     12     17
+ level 1:    113    12     217
layer2 (*3):  339    36     651      # level 只有 2 个，layer2 之后不注入
run_layers(...) = [[339], [36], [651]]
```

如果错把注入放在 layer **之前**，会得到 `[[342], [36], [666]]`——非视觉行不受影响，所以只看文本行是发现不了这个 bug 的。

关键问题

1. audio_token_count 要同时支持 Python int 和 LongTensor。哪些写法（例如 `math.floor`、`int(...)`、`if` 分支）会在 tensor 输入下出问题？只用哪些运算符就够了？
2. r = 0 时（n 恰好是 100 的倍数），公式 `(r − 1) // 2 + 1` 给出多少？这依赖 Python/PyTorch 的 `//` 对负数的什么行为？
3. 对一个模态，“按顺序”替换是什么意思？第 k 个占位符位置应该取 `mm_embeds[modality]` 的第几行？用布尔 mask 做 `out[mask] = emb` 能保证这个顺序吗？
4. 怎样保证 text_embeds 不被修改？`out = text_embeds` 之后再 `out[mask] = ...` 会发生什么？
5. 如果某模态在 placeholder_ids 里但不在 mm_embeds 里，什么情况下应该报错，什么情况下应该静默跳过？
6. `visual_embeds` 的行数和 `visual_mask` 有什么关系？`hidden[visual_mask]` 的形状是多少？
7. 有 L 个 layer、K 个 DeepStack level（K < L）时，哪几个 layer 之后会注入？level i 和 layer i 的对应关系是什么？

细节

- **音频公式的来历**：AuT 编码器把 mel 帧切成 100 帧一窗，三次 stride-2 卷积每次长度变为 `(len − 1) // 2 + 1`（即 ceil(len/2)），100 帧 → 50 → 25 → 13。完整窗口直接贡献 13，只有余数窗口需要真的走一遍公式。前端（processor）必须用同一个公式算出要插多少个 `<audio>` 占位符。
- **int / tensor 双态**：`%`、`//`、`+`、`−`、`*` 对 Python int 和 LongTensor 都按逐元素整数语义工作，所以一个只用这些运算符的表达式天然支持两种输入。对负数，两者的 `//` 都是向下取整（floor），因此 `(−1) // 2 = −1`。
- **masks 的语义**：返回 `{modality: bool[T]}`，对 placeholder_ids 里的**每个**模态都要有一项（即使该模态没有出现，也是一个全 False 的 mask）。后续 DeepStack 用的 visual_mask 就来自这里。
- **“in order”**：编码器按占位符在序列中出现的顺序产出 embedding。布尔索引赋值 `out[mask] = emb` 按 mask 中 True 的升序位置依次写入 emb 的第 0, 1, 2… 行，正好满足“按顺序”。不同模态的占位符可以交错（例如 image, audio, image），每个模态各自计数。

© 2026 kzhou003

- **dtype**：编码器输出可能和 text_embeds dtype 不同（例如 ViT 在 fp32、LLM 在 bf16），写入前要转成输出的 dtype。DeepStack 加法同理。
- **DeepStack 是"加"不是"替换"**，并且返回新 tensor（`apply_deepstack` 不能改动传入的 hidden）。`run_layers` 中 level i 在 layer i **之后**注入；多出来的 layer（i ≥ len(deepstack_embeds)）只做普通前向。

Level 2 · 思路与常见坑

- **audio_token_count**：一行表达式即可，分三步写更清楚：`r = n % 100` → 三次 `(len − 1) // 2 + 1` → 加上 `13 * (n // 100)`。不要写 `if r == 0` 之类的分支（对 tensor 不可用，且公式本身已正确给出 0）。
- **merge_multimodal 的循环结构**：先 `out = text_embeds.clone()`；对 `placeholder_ids` 的每个 (modality, token_id)：算 mask、存进 masks、统计 n = mask.sum()；再看 mm_embeds 里有没有这个模态，做校验和写入。以 placeholder_ids 为主循环，而不是 mm_embeds，这样"有占位符却缺 embedding"的情况自然被覆盖。
- **校验**：(a) 模态缺失：n > 0 → ValueError，n == 0 → 跳过；(b) 模态存在：`emb.shape[0] != n` → ValueError。错误消息里带上模态名和两边的数量，线上排查时非常有用。
- **写入**：`out[mask] = emb.to(out.dtype)`。易错：`out[mask.nonzero()] = emb` 会多出一维；`torch.where(mask[:, None], emb, out)` 形状对不上（emb 只有 n 行）。
- **不可变性**：`clone()` 必须在任何写入之前。若用 `out = text_embeds`，所有写入都会改到调用方的 tensor，而调用方可能在下一个请求或 CUDA graph 里复用它。
- **apply_deepstack**：`out = hidden.clone()`，`out[visual_mask] = out[visual_mask] + visual_embeds`（或 `index_add_` 配合 nonzero 下标）。易错：`hidden[visual_mask] += ...` 原地改了输入；`out += visual_embeds` 形状不匹配（n_vis 行 vs T 行）。
- **run_layers**：`for i, layer in enumerate(layers)`：先 `hidden = layer(hidden)`，再在 `i < len(deepstack_embeds)` 时注入第 i 级。易错：注入写在 layer 调用之前（off-by-one，HF 对拍失败但视觉行之外完全一致）；或 zip(layers, deepstack) 导致多余的 layer 被跳过。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key question 答案：

- `math.floor`、`int(n)`、`if r == 0` 都不能用于多元素 tensor；只用 `% // + − *` 就够了。
- r = 0：`(0 − 1) // 2 + 1 = −1 + 1 = 0`，后两步同样得到 0；依赖 floor division（向负无穷取整）。如果用 C 风格的截断除法会得到 1，100 帧会被错算成 14。
- 第 k 个 True 位置取第 k 行；布尔索引赋值按位置升序写入，能保证。
- `clone()`；不 clone 的话 `out` 与 text_embeds 是同一块内存，写入就是原地修改。
- 缺 embedding 且该模态有 ≥1 个占位符 → 报错；占位符数为 0 → 跳过（mask 仍要返回）。
- `visual_embeds.shape[0] == visual_mask.sum()`；`hidden[visual_mask]` 是 `[n_vis, H]`。
- layer 0 … K-1 之后各注入一次，level i ↔ layer i（0-based）。

audio_token_count 骨架（docstring 里的 `((f − 1) // 2 + 1 − 1) // 2 + 1` 就是 b、c 两步写在一起。）
merge_multimodal 骨架 用 io 例子核对：image mask 的 True 位置是 1、2，于是第 1 行 ← image[0] = [100,100]，第 2 行 ← image[1] = [200,200]；audio 的 4、5 同理。
apply_deepstack / run_layers 骨架 逐行核对 io 例子：第 0 行 1 →(*2) 2 →(+1) 3 →(+10) 13 →(+100) 113 →(*3) 339；第 1 行不是视觉行：1 → 2 → 12 → 36。
与 serving 的联系：`audio_token_count` 在 API 进程（HF processor）里决定 prompt 长度，`merge_multimodal` 在 thinker stage 里消费编码器输出——两处公式不一致时，报错发生在 merge 的数量校验处，这也是为什么校验必须严格而不能"截断/补零"了事。DeepStack 的 visual_mask 就是 merge 返回的 image（及 video）mask。

A-5 · Talker MTP：一帧内预测 15 个残差 codebook

Level 1 · I/O 示例

例 1：G=4（1 个 CB0 + 3 个残差 codebook），B=2，H=16，vocab=10（随机权重的 tiny code predictor，1 层）。

```
last_talker_hidden: [2, 16]   cb0_embed: [2, 16]
predict_residual_codes(cp, hid, cb0)    -> tensor([[4, 5, 6],
                                                   [7, 4, 9]])   # [B, G−1] = [2, 3]
predict_residual_codes_kv(cp, hid, cb0) -> tensor([[4, 5, 6],
                                                   [7, 4, 9]])   # 必须逐元素相同
```

第 0 列是 CB1（用 `lm_head[0]` 预测），第 1 列是 CB2（`lm_head[1]`），第 2 列是 CB3（`lm_head[2]`）。只取第 1 行单独跑（输入 `hid[1:2], cb0[1:2]`）得到 `[[7, 4, 9]]` ——batch 中各行互不影响。

例 2：re-prefill 版每一步的序列长度（同一个 G=4, B=2 的模型）

j	送进 cp.model 的 inputs_embeds	取哪一行	logits	得到	之后 append
0	[2, 2, 16] = [h_talker, E0(cb0)]	[:, -1]	lm_head[0] → [2, 10]	CB1 = [4, 7]	codec_embedding[0](CB1)

© 2026 kzhou003

1	[2, 3, 16]	[:, -1]	lm_head[1] → [2, 10]	CB2 = [5, 4]	codec_embedding[1](CB2)
2	[2, 4, 16]	[:, -1]	lm_head[2] → [2, 10]	CB3 = [6, 9]	（循环结束，不再需要 append 也行）

输出把每一步的 code 按列拼起来：`[[4,5,6],[7,4,9]]`。KV 版在相同的 3 步之后，cache 的长度依次是 2 → 3 → 4。

例 3：next_talker_input 的形状与含义（G=4，B=2，H=16）

```
cb0_embed:       [2, 16]       # 这一帧 CB0 的 talker codec embedding（调用方给）
residual_codes:  [2, 3]        # 例 1 的 [[4,5,6],[7,4,9]]
text_step:       [2, 16]       # 本步对齐的 thinker 文本向量
返回 [2, 16] = cb0_embed
             + codec_embedding[0](residual_codes[:, 0])   # 第 0 行查 index 4，第 1 行查 index 7
             + codec_embedding[1](residual_codes[:, 1])   # 查 5 / 4
             + codec_embedding[2](residual_codes[:, 2])   # 查 6 / 9
             + text_step
```

一共 G=4 个 codec embedding 相加（1 个 CB0 + G-1 个残差），再加文本向量；结果仍是 `[B, H]`，没有序列维度。

关键问题

1. 预测第 j+1 个 codebook 时，用的是 `lm_head` 的第几个、`codec_embedding` 的第几个？两者的下标为什么相同而不是差 1？
2. 起始序列 `[last_talker_hidden, cb0_embed]` 是两个 `[B, H]`；你要把它们变成 `[B, 2, H]` 应该用 `stack` 还是 `cat`？沿哪个维度？
3. 每一步 transformer 输出 `[B, L, H]`，你取哪一个位置去算 logits？为什么不是位置 0？
4. G-1 个 code 中的最后一个（CB_{G-1}）算出来之后，re-prefill 版还需要再跑一次 transformer 吗？next_talker_input 需要它的 embedding 吗？
5. KV 版：第一次调用 `cp.model` 喂几个位置？第二次喂几个？如果第二次仍然把完整序列喂进带 cache 的模型，会发生什么？
6. KV 版里，新位置的 position id 由谁负责递增？你需要手动传 position_ids 吗？（提示：想想 HF 的 `DynamicCache` 记录了什么。）
7. next_talker_input 里一共有几个 embedding 相加？CB0 那一项为什么直接用传进来的 `cb0_embed`，而不是去 code predictor 的表里查？

细节

- **G = num_code_groups** 是每帧 codebook 总数（真实模型 16，含 CB0）。talker 大模型只采 CB0，code predictor 只负责其余的 **G-1** 个，所以 `lm_head` 和 `codec_embedding` 都是长度 G-1 的 `ModuleList`，下标 j=0 对应 CB1。
- **re-prefill（no cache）**：每一步都把**完整的当前序列**交给 `cp.model(inputs_embeds=...)`，不传 `past_key_values`。因为是因果注意力，最后一个位置的输出和增量解码的结果在数学上一样；vllm-omni 选它是因为序列最长只有 G+1，且形状固定便于 CUDA graph。
- **Greedy**：`code = logits.argmax(-1)`，结果是 `[B]` 的 long 张量，可以直接作为 `nn.Embedding` 的索引。真实服务用 top-k/top-p 采样，本题为了可对拍才用 argmax。
- **codec_embedding[j]** 是 code predictor 自己的 embedding 表（`[vocab, H]`），与 talker 主模型里 CB0 的表不是同一张。CB0 的 embedding 由调用方算好以 `cb0_embed` 给你。
- **DynamicCache**（`from transformers import DynamicCache`）：创建一个空对象，作为 `past_key_values=` 传进去并设 `use_cache=True`，模型会就地把每层的 K/V 追加进去；它也知道已缓存了多少位置，所以 RoPE 的位置会自动接上。
- **next_talker_input** 是 talker **下一步**的输入 embedding：“这一帧所有 G 个 codebook 的 embedding 之和 + 对齐的文本向量”。这是 Qwen3-Omni talker 的“多 codebook 求和输入”约定，返回 `[B, H]`。

Level 2 · 思路与常见坑

- **拆解**：三个函数共享一个“单步”模式——*拿最后位置 hidden → lm_head[j] → argmax → codec_embedding[j] 嵌入*。两个 predict 函数的区别只在于“每步给 transformer 喂什么”：re-prefill 喂整段，KV 版只喂新的一段。
- **数据结构**：序列用一个 `[B, L, H]` 张量维护，每步用 `torch.cat(..., dim=1)` 在长度维追加 `[B, 1, H]`；code 收集到 Python list 里，最后 `torch.stack(codes, dim=1)` 成 `[B, G-1]`。注意 embedding 输出是 `[B, H]`，追加前要补出长度维（`[:, None]` 或 `unsqueeze(1)`）。
- **循环次数**：`for j in range(G - 1)`，每次恰好产生一个 code。不要写成 `range(G)`（会越界访问 `lm_head[G-1]`）或 `range(G-2)`（少一个 code，输出变成 `[B, G-2]`）。
- **KV 版的输入切换**：用一个变量 `x` 表示“这一步要喂的新位置”。第一次 `x` 是 `[B, 2, H]` 的起始序列，之后 `x` 换成刚得到的 code 的 embedding `[B, 1, H]`。cache 在循环外只创建一次。
- **常见 bug 1：所有步都用 lm_head[0]**。第一个 code 会对，后面开始错。在上面的 G=4 例子里，结果会从 `[[4,5,6],[7,4,9]]` 变成 `[[4,5,4],[7,4,2]]`——只有第一列稳定是对的，这是这个 bug 的典型症状。
- **常见 bug 2：KV 版每步仍喂整段序列**。cache 里会重复存旧位置，位置编号也被推远，结果悄悄偏离：同一例子会得到 `[[4,5,6],[7,4,2]]`——有的行对、有的行错，非常难查。检查方式：每步之后 cache 长度应为 2, 3, 4, …
- **no_grad**：这是推理，用 `@torch.no_grad()` 或 `with torch.no_grad():` 包起来，否则会构建无用的计算图（结果不变，但更慢更占内存）。

© 2026 kzhou003

- **next_talker_input 的易错点**：残差 code 第 j 列要用 `codec_embedding[j]` 查（和预测时同一个 j）；不要原地修改传入的 `cb0_embed`（用 `a = a + b` 而不是 `a += b`，或先 `clone()`），否则会改坏调用方的张量。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

- 预测 CB(j+1) 用 `lm_head[j]`；得到的 code 用 `codec_embedding[j]` 嵌入。两个表都只覆盖残差 codebook（长度 G-1），下标 0 都指 CB1，所以同一个 j。
- `torch.stack([last_talker_hidden, cb0_embed], dim=1)` → `[B, 2, H]`。`cat` 会得到 `[2B, H]` 或 `[B, 2H]`，都不对。
- 取 `last_hidden_state[:, -1]`（`[B, H]`）：因果 LM 用最后一个位置预测"下一个"。
- 最后一个 code 算出后不必再跑 transformer；但 next_talker_input 需要它的 embedding（用 `codec_embedding[G-2]`）。
- KV 版第一次喂 2 个位置，之后每次 1 个。重复喂整段会让 cache 里出现重复的 K/V，位置也错。
- 不需要手动传 position_ids：`DynamicCache` 记录了已缓存长度，HF 模型据此推出新位置的 `cache_position`。
- 一共 G 个 codec embedding（CB0 + G-1 个残差）加 text_step。CB0 的 embedding 来自 talker 主模型的表，code predictor 的表里没有 CB0，所以由调用方传入。

形状追踪（G=4, B=2, H=16, vocab=10）

```
seq0 = stack([hid, cb0], 1)            [2, 2, 16]
j=0: cp.model(seq0).last_hidden_state  [2, 2, 16] -> [:, -1] [2, 16]
     lm_head[0](.)                     [2, 10] -> argmax(-1) [2]   = [4, 7]
     codec_embedding[0](code)          [2, 16] -> [:, None] [2, 1, 16]
     seq1 = cat([seq0, e], 1)          [2, 3, 16]
j=1: ...                               [2, 4, 16]       code = [5, 4]
j=2: ...                                                code = [6, 9]
stack([c0, c1, c2], dim=1)             [2, 3]  = [[4,5,6],[7,4,9]]
```

predict_residual_codes_kv 骨架 自检：在循环里打印 `cache.get_seq_length()`，G=4 时应依次为 2、3、4；两个版本对同样输入的输出应 `torch.equal`。
`residual_codes[:, j]` 是 `[B]`，查表得 `[B, H]`；不要把整个 `[B, G-1]` 丢进同一张表（那会得到 `[B, G-1, H]` 且所有列都用了同一张表）。
为什么 re-prefill 与 KV 版一定一致：因果注意力下，位置 p 的输出只依赖位置 0..p 的输入。re-prefill 每步重算了所有旧位置，但旧位置的 K/V 与上一步完全相同；KV 版只是把它们存下来。所以只要 RoPE 位置一致（DynamicCache 自动保证），最后位置的 hidden 数值几乎相同，argmax 相同。

A-6 · Code2Wav 的流式因果卷积与转置卷积

Level 1 · I/O 示例

例 1：因果 Conv1d，K=3，d=1，C_in=C_out=1，weight=[1,2,3]，bias=0

```
x = [1, 2, 3, 4, 5]                # [1, 1, 5]
离线：左补 (K-1)*d = 2 个 0 → [0, 0, 1, 2, 3, 4, 5]
      y[t] = 1·x[t-2] + 2·x[t-1] + 3·x[t]
      y = [3, 8, 14, 20, 26]

流式，分块 [2, 3]：
step(x[0:2]=[1,2])   -> [3, 8]          state 变成 [1, 2]
step(x[2:5]=[3,4,5]) -> [14, 20, 26]    state 变成 [4, 5]
拼接 = [3, 8, 14, 20, 26]  ✓ 与离线一致
```

每次输出个数 = 本块输入个数；state 永远是"最近 (K-1)·d = 2 个输入"。第一块之前 state 是 2 个 0（正好等价于离线的左补零）。

例 2：带 dilation 的因果 Conv1d，K=2，d=3，weight=[1,1]，bias=0

```
y[t] = x[t-3] + x[t]，需要保留 (K-1)·d = 3 个历史输入
x = [1, 2, 3, 4, 5, 6]，离线 y = [1, 2, 3, 5, 7, 9]

分块 [1, 4, 1]：
step([1])          -> [1]            state = [0, 0, 1]
step([2, 3, 4, 5]) -> [2, 3, 5, 7]   state = [3, 4, 5]
step([6])          -> [9]            state = [4, 5, 6]
```

注意第一块只有 1 个样本、比 state 还短：state 要从"旧 state + 新块"拼起来的缓冲区末尾取，所以它仍包含两个补出来的 0。

例 3：因果 ConvTranspose1d，K=4，s=2，weight=[1,2,3,4]（[C_in=1, C_out=1, K]），bias=0.5

```
x = [1, 10, 100]（3 帧）
帧 i 写入位置 [2i, 2i+4)，值 = x_i · [1,2,3,4]
```


© 2026 kzhou003

```
位置:        0    1    2     3     4      5      6     7
帧0 (1):     1    2    3     4
帧1 (10):              10    20    30     40
帧2 (100):                         100    200    300   400
求和+bias: 1.5  2.5  13.5  24.5  130.5  240.5  300.5  400.5    (长度 (3-1)·2+4 = 8)
离线：裁掉开头 K-s=2 个、结尾 2 个 → [13.5, 24.5, 130.5, 240.5]

逐帧流式：
step(帧0) -> []               # 位置 <2 已确定，但正好是左裁剪的 2 个
step(帧1) -> [13.5, 24.5]     # 位置 2,3 确定
step(帧2) -> [130.5, 240.5]  # 位置 4,5 确定；6,7 还在等（未来帧会写到这里）
flush()   -> []               # 剩下 6,7，正好被右裁剪掉
分块 [2, 1] 也一样：step(帧0..1) -> [13.5, 24.5]，step(帧2) -> [130.5, 240.5]，flush -> []
```

累计输出数在第 n 帧后 = `max(0, n·s − (K−s))` = 0, 2, 4。
bias 加错的样子：如果每帧单独做带 bias 的转置卷积再重叠相加，重叠位置的 bias 加了两次，输出变成 `[14.0, 25.0, 131.0, 241.0]`（每个都多 0.5）。

例 4：K == s 的转置卷积（K=3, s=3, weight=[1,2,3], bias=0）

```
x = [1, 10]：帧之间不重叠，K−s = 0，不裁剪
离线 = [1, 2, 3, 10, 20, 30]
step(帧0) -> [1, 2, 3]   step(帧1) -> [10, 20, 30]   flush() -> []
```

这是检查“裁剪量为 0”边界的好例子。

关键问题

1. 因果 Conv1d 的 state 需要保存多少个样本？K=1 时是多少？此时你的切片代码 `buf[..., −n:]` 在 n=0 时返回什么？
2. 第一次调用 `step` 之前 state 应该是什么？它的 batch / channel 维从哪来（构造函数里还不知道 B）？
3. 转置卷积：收到 n 帧之后，哪个位置之前的输出不会再被任何未来帧修改？为什么？
4. 全局左裁剪 K-s 个样本：如果你每次 step 都裁掉本次输出的前 K-s 个，会错在哪里？用例 3 的 [2,1] 分块想一想。
5. bias 应该在什么时刻加？如果在每块的 `conv_transpose1d` 里传 bias，哪些位置会错、错多少？
6. flush 时总输出长度应是多少（用帧数 L、s、K 表示）？右裁剪之后还剩多少？
7. `from_hf` 里 dilation 和 stride 分别从 HF 模块的哪个属性取？`nn.ConvTranspose1d.stride` 是 int 还是 tuple?

细节

- **权重布局**：`nn.Conv1d` 的 weight 是 `[C_out, C_in, K]`；`nn.ConvTranspose1d` 的是 `[C_in, C_out, K]`（docstring 已注明）。两种都可以直接交给 `F.conv1d` / `F.conv_transpose1d`，不必手写卷积。`F.conv1d(..., dilation=d)` 无 padding 时，输出长度 = 输入长度 - (K-1)·d。
- **“因果”的含义**：第 t 个输出只依赖输入 ≤ t。离线版通过**只在左边**补 (K-1)·d 个零实现。流式版把“左边的零”换成“上一块留下的真实历史”，开头那次正好是零。
- **“已确定（final）”**：转置卷积中帧 i 影响 [i·s, i·s+K)。已收到 n 帧时，未来帧（下标 ≥ n）只会写 ≥ n·s 的位置，所以 < n·s 的位置都确定了，可以立即输出——这直接决定首包音频延迟。
- **全局坐标 vs 块内坐标**：左裁剪针对的是**整条输出流**的前 K-s 个样本（全局位置 0..K-s-1），不是每一块的开头。需要记录“已经输出到哪个全局位置”。
- **flush**：所有帧都到齐后，总长度是 (L-1)·s + K；把尚未输出的部分吐出，并去掉最后 K-s 个（离线模块的右裁剪）。K==s 时裁剪量为 0，小心 `t[..., :−0]` 得到空张量。
- **from_hf**：HF 的 `Qwen3OmniMoeCausalConvNet` 有 `m.conv`（`nn.Conv1d`）、`m.dilation`、`m.stride`（本题只支持 1）；`Qwen3OmniMoeCausalTransConvNet` 的 stride 要从 `m.conv.stride` 取（tuple，取 [0]）。权重建议 `.detach()`，bias 可能为 None。

Level 2 · 思路与常见坑

- **StreamingCausalConv1d 的状态**：只需 `weight, bias, dilation`、`context = (K−1)·d` 和一个 `state`（初始 None，首次 step 时按输入的 B、C 建全零 `[B, C_in, context]`）。step = 拼接 → 更新 state → 无 padding 的 `F.conv1d`。
- **state 的更新顺序**：先 `buf = cat([state, x], −1)`，再从 **buf** 末尾取 context 个作为新 state，最后对 buf 做卷积。若从 x 取末尾，当块长 < context 时 state 会变短，下一块输出个数就不对（例 2 的第一块就是这种情况）。
- **K=1 边界**：context=0 时 `buf[..., −0:]` 等于整个 buf（Python 的 -0 就是 0），state 会无限增长、输出越来越多。要用 `buf[..., buf.shape[−1]−context:]` 或显式判断 context==0。
- **StreamingCausalTransConv1d 的状态**：`frames`（已收到帧数）、`base`（下一个要输出的全局位置）、`acc`（从 base 开始的未确定样本累加缓冲区）。`trim = K − s`。
- **step 流程**：(1) 对本块做**不带 bias**的 `F.conv_transpose1d(x, w, None, stride=s)`，长度 (t-1)·s+K，覆盖全局位置 [frames·s, frames·s + len)；(2) 必要时把 acc 右侧补零延长，再把结果加到对应区间（下标要减 base）；(3) frames += t；(4) 输出 [base, frames·s) 这一段。
- **输出函数（step 和 flush 共用）**：从 acc 切出前 (upto - base) 个，acc 保留剩余；此时加 bias（每个位置一次）；再按全局位置去掉落在 [0, trim) 内的部分：`skip = max(0, trim − 旧 base)`；最后 base = upto。

© 2026 kzhou003

- **常见 bug 与症状**：bias 进了每块卷积 → 重叠区多出 bias（例 3 中所有值多 0.5）；左裁剪按块做 → 后面每块都少 K-s 个样本，总长度不对；输出了 ≥ frames·s 的位置 → 那些值之后还会被加，导致与离线不一致；输出得太少 → 结果仍可能正确，但“尽早输出”的计数对不上。
- **分块不变性**：一次喂多帧时，单个 `conv_transpose1d` 已经在块内部做了重叠相加；块与块之间的重叠由 acc 负责。两者组合与一次离线计算完全一致（线性 + 无 bias）。

Level 3 · 详解（代码骨架见附录 C 的带注释参考考解）

key questions 答案

- state 保存 (K-1)·d 个样本；K=1 时为 0。`buf[..., -0:]` 返回整个 buf——必须特判。
- 初始 state 为全零 `[B, C_in, context]`，在第一次 step 时用 `x.new_zeros(x.shape[0], x.shape[1], context)` 懒创建（同时继承 dtype/device）。
- 收到 n 帧后，位置 < n·s 已确定：任何未来帧 i ≥ n 只写 [i·s, i·s+K)，起点 ≥ n·s。
- 按块裁剪：例 3 用 [2,1] 分块时，第二块的输出 [130.5, 240.5] 会被错误地裁掉 2 个变成 []。左裁剪只作用于全局位置 0..K-s-1。
- bias 在“输出某个已确定位置”时加一次。若在每块卷积里加，块与块重叠的位置（每个位置被 ceil(K/s) 帧覆盖，跨块部分）会多加。
- flush 时总长 = (L-1)·s + K，去掉左右各 K-s 后剩 L·s - (K-s)。例 3：L=3 → 8 - 4 = 4。
- Conv：`m.dilation`、`m.stride`；TransConv：`m.conv.stride[0]`（tuple）。

例 3 逐帧的内部状态（K=4, s=2, trim=2；acc 里存的都是**未加 bias**的值）

操作	本块卷积覆盖的全局区间	frames	输出区间	输出	base 之后	acc 之后
step(帧0)	[0,4) = 1,2,3,4	1	[0,2) 全在 trim 内	[]	2	[3, 4]（位置 2,3）
step(帧1)	[2,6) += 10,20,30,40	2	[2,4)	[13.5, 24.5]	4	[30, 40]
step(帧2)	[4,8) += 100..400	3	[4,6)	[130.5, 240.5]	6	[300, 400]
flush()	—	3	[6,8)，再去掉右侧 2 个	[]	8	[]

acc 的长度始终 ≤ K-s：它就是“还在等后续帧”的尾巴。

自检清单

- 因果卷积：用例 1、例 2 的手工权重分别按 [5]、[1,1,1,1,1]、[2,3] 分块，结果都应是同一个序列；每块输出个数 == 输入个数。
- 转置卷积：K=8, s=4 逐帧喂入时，累计输出数应为 0, 4, 8, 12, ...（= max(0, 4n-4)）。
- K==s（例 4）：无裁剪，step 每帧输出 s 个，flush 为空。
- 把所有 step 输出和 flush 输出 cat 起来，与 HF 模块的离线 forward 用 `torch.testing.assert_close` 比较。

A-7 · 按 stage 加载与打包权重

Level 1 · I/O 示例

例 1：shards_for_stage

```
weight_map = {
  "thinker.model.embed_tokens.weight":       "model-00001.safetensors",
  "thinker.model.layers.0.mlp.gate.weight":  "model-00002.safetensors",
  "talker.model.codec_embedding.weight":     "model-00002.safetensors",
  "talker.code_predictor.lm_head.0.weight":  "model-00003.safetensors",
  "code2wav.decoder.0.conv.weight":          "model-00003.safetensors",
  "thinker.lm_head.weight":                  "model-00004.safetensors",
}
shards_for_stage(weight_map, "thinker")  -> ["model-00001.safetensors", "model-00002.safetensors", "model-
00004.safetensors"]
shards_for_stage(weight_map, "talker")   -> ["model-00002.safetensors", "model-00003.safetensors"]
shards_for_stage(weight_map, "code2wav") -> ["model-00003.safetensors"]
```

同一个 shard（00002）可以同时属于两个 stage；结果去重并排序。注意 `thinker.lm_head.*` 也属于 thinker stage（它不在 `thinker.model.` 下，但在 `thinker.` 下）。

例 2：select_stage

```
state keys:
  "thinker.model.norm.weight"
  "thinker.model.layers.0.input_layernorm.weight"
  "thinker.lm_head.weight"
  "talker.model.norm.weight"
  "thinkers.model.x"           # 故意构造的“相似前缀”
select_stage(state, "thinker", "thinker.model.") 的 keys
  -> {"layers.0.input_layernorm.weight", "norm.weight"}
```


© 2026 kzhou003

`thinker.lm_head.weight` 不在 `strip` 前缀下，所以被丢弃（M08 会单独取它）；值（张量）原样保留，只改 key。

例 3：pack_layer 的形状与行顺序（H=2, KVH=1, D=2, hidden=4, E=2, I=3；每个张量填充成一个常数便于看出来源）

```
layers.0.self_attn.q_proj.weight  [4, 4] 全 1
layers.0.self_attn.k_proj.weight  [2, 4] 全 2
layers.0.self_attn.v_proj.weight  [2, 4] 全 3
layers.0.mlp.experts.e.gate_proj.weight [3, 4] 全 10e+1
layers.0.mlp.experts.e.up_proj.weight   [3, 4] 全 10e+2
layers.0.mlp.experts.e.down_proj.weight [4, 3] 全 10e+3
layers.0.mlp.gate.weight [2, 4]
 （以及 input_layernorm / q_norm / k_norm / o_proj / post_attention_layernorm）

pack_layer(sd, 0):
  qkv     [8, 4]     第 0 列 = [1, 1, 1, 1, 2, 2, 3, 3]          # (H + 2·KVH)·D = 8 行：q | k | v
  w13     [2, 6, 4]  w13[:, :, 0] = [[1, 1, 1, 2, 2, 2],
                                     [11, 11, 11, 12, 12, 12]]  # 每个 expert：gate 3 行 | up 3 行
  w2      [2, 4, 3]  w2[:, 0, 0]  = [3, 13]
  router  [2, 4]
```

sd 里同时存在 `layers.1.*` 也没关系——pack_layer(sd, 0) 只看 `layers.0.` 下的 key。

例 4：哪些情况必须 LoadError（在例 3 的 sd 基础上改一处）

修改	pack_layer(sd, 0)
原样	成功
额外加入 expert 3 的三个权重（id 变成 0,1,3）	LoadError（id 不连续）
删掉 `mlp.experts.1.down_proj.weight`	LoadError（expert 1 不完整）
加入 `layers.0.self_attn.rotary_emb.inv_freq`	LoadError（多余 key）
删掉 `layers.0.self_attn.k_norm.weight`	LoadError（缺 key）
加入 `layers.10.input_layernorm.weight`	成功（那是第 10 层，不属于第 0 层）

load_thinker_text：state 里多一个 `thinker.model.rotary_emb.inv_freq` → LoadError；`num_layers` 比实际层数多（例如只有第 0 层却传 2）→ 第 1 层什么都没有 → LoadError；`thinker.lm_head.weight` 和 `talker.*` 存在不算错误（不在 `thinker.model.` 下）。

关键问题

1. 判断一个 key 属于哪个 stage，应该用哪个前缀？为什么 `"thinker"`（不带点）不够安全？
2. select_stage 去掉前缀时，`"thinker.model.norm.weight"` 变成什么？`"thinker.lm_head.weight"` 会怎样？
3. 在第 i 层里收集 key 时，为什么必须用 `"layers.{i}."`（带结尾的点）作前缀？i=1 时不带点会误收哪些层？
4. qkv 的行数是多少（用 H、KVH、D 表示）？为什么是 `cat` 沿 dim 0，而不是 dim 1？
5. w13 是先对每个 expert `cat(gate, up)` 再 `stack`，还是先 `stack` 所有 gate 再和 stack 后的 up 拼？两种做法得到的形状/内容一样吗？
6. 如何判断 expert id “恰好是 0..E-1 且完整”？E 从哪里得知（函数参数里没有 E）？
7. “多余 key”怎么检测？你需要记录哪些 key 已被使用？

细节

- **weight_map** 来自 `model.safetensors.index.json`，形如 `{tensor_name: shard_file}`。一个 stage 只需打开至少含有一个本 stage key 的 shard；返回**排序后**、**去重**的文件名列表。
- **stage 前缀 vs strip 前缀**：`STAGE_PREFIX` 给出 stage 级前缀（如 `"thinker."`）；select_stage 的 `strip` 更具体（如 `"thinker.model."`），只保留以它开头的 key 并去掉这段。用 `startswith` + 切片 `k[len(strip):]`，不要用 `str.replace`（可能替换掉中间的子串）。
- **行拼接（dim 0）**：`nn.Linear` 的 weight 是 `[out, in]`。把多个共享输入的 Linear 合并成一个，就是在 out 维（行）上拼接：`x @ qkv.T` 的输出列依次是 q、k、v，再按 `[H·D, KVH·D, KVH·D]` split 回来。gate/up 同理。
- **PackedLayer 的 9 个字段**与 HF key 一一对应：`input_layernorm.weight`、`self_attn.{q,k,v,o}_proj.weight`、`self_attn.{q,k}_norm.weight`、`post_attention_layernorm.weight`、`mlp.gate.weight`（router，注意它叫 gate 但不是 gate_proj）、`mlp.experts.{e}.{gate,up,down}_proj.weight`。
- **完整性**：一层内“缺一个”或“多一个” key 都是 LoadError；load_thinker_text 在 `thinker.model.` 之下除了 `layers.*`、`embed_tokens.weight`、`norm.weight` 之外的任何 key 也是 LoadError。返回的第一个元素是 `{"embed_tokens": ..., "norm": ...}`（key 里没有 `.weight`）。
- **正则**：`re` 已经导入，expert key 的规整形式是 `mlp.experts.<数字>.<gate_proj|up_proj|down_proj>.weight`，适合用带锚点（`^...$`）的正则抽取 id 和投影名。

© 2026 kzhou003

Level 2 · 思路与常见坑

- **shards_for_stage**：一行集合推导——遍历 `weight_map.items()`，key 以 stage 前缀开头就收集其 value，最后 `sorted(set)`。
- **pack_layer 第一步：切出本层子字典** `sub = {去掉 "layers.{i}." 后的 key: tensor}`。之后所有操作都在 sub 上进行，key 变成 `"self_attn.q_proj.weight"` 这样的短名，便于检查。
- **记账：一个 used 集合**。写一个小助手 `take(name)`：不存在就 raise LoadError，存在就把 name 加入 used 并返回张量。expert 的 key 在正则匹配成功时也加入 used。最后 `set(sub) - used` 非空就是"多余 key"。
- **收集 experts**：`experts: dict[int, dict[str, Tensor]]`，正则匹配后 `experts.setdefault(int(id), {})[proj] = v`。然后检查：非空、`sorted(experts) == list(range(len(experts)))`、每个 expert 的 proj 集合恰好是 `{gate_proj, up_proj, down_proj}`。E 就是 `len(experts)`。
- **打包**：`qkv = cat([q, k, v], 0)`；`w13 = stack([cat([gate_e, up_e], 0) for e in range(E)])` → `[E, 2I, hidden]`；`w2 = stack([down_e ...])` → `[E, hidden, I]`。一定要按 e = 0..E-1 的顺序，而不是按字典遍历顺序。
- **顺序陷阱**："多余 key"检查必须在所有 take 之后进行；否则会把还没被 take 的合法 key 误报为多余。
- **load_thinker_text**：`sd = select_stage(state, "thinker", "thinker.model.")` → 对 `range(num_layers)` 调 pack_layer → 检查非 `layers.` 开头的 key 集合恰好是 `{embed_tokens.weight, norm.weight}`（多了或少了都 LoadError）。
- **常见 bug 与症状**：前缀不带点（`"layers.1"`）→ 第 1 层会吞掉 layers.10..19 的 key，然后被报"多余 key"或 expert 冲突；w13 拼成 `[up | gate]` → 形状对但 SiLU 用在了错误的那一半，expert 输出数值全错；按 dict 顺序 stack expert → id 顺序乱时 expert 被错配，router 选中的 expert 不对。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

- 用 `STAGE_PREFIX[stage]`（带点，如 `"thinker."`）。不带点会让 `"thinkers.x"`、`"talker2.x"` 之类的 key 被误判。
- `"thinker.model.norm.weight"` → `"norm.weight"`；`"thinker.lm_head.weight"` 不以 strip 开头，被丢弃。
- `"layers.1"` 会匹配 `layers.1x.*`（10–19）、`layers.1xx.*` 等。带点 `"layers.1."` 只匹配第 1 层。
- qkv 行数 = (H + 2·KVH)·D。Linear 权重 `[out, in]`，三者 in 相同（hidden）、out 不同，合并输出就是沿 dim 0 拼。
- 必须"每个 expert 内 cat(gate, up)，再 stack"，得到 `[E, 2I, hidden]`，每个 expert 的前 I 行是 gate、后 I 行是 up。若先分别 stack 再沿 dim 1 拼，结果其实相同；但若沿 dim 0 拼会变成 `[2E, I, hidden]`，错误。
- E = 收集到的不同 id 数；条件是 `sorted(ids) == list(range(E))` 且 E > 0，并且每个 id 都有三种 proj。
- 维护 used 集合，所有字段取完后 `set(sub) - used`。

自检：打包后是否等价（用你自己的小张量）

```
x = torch.randn(5, hidden)
q, k, v = (x @ L.qkv.T).split([H*D, KVH*D, KVH*D], dim=-1)
# q 应等于 x @ q_proj.T，k、v 同理
g, u = (x @ L.w13[e].T).chunk(2, dim=-1)
y = (F.silu(g) * u) @ L.w2[e].T      # 应等于 expert e 的 SwiGLU 输出
```

这也正是 M08 把 PackedLayer 还原成 M02/M03 所需权重的方式：qkv 按行 split，w13 按第 1 维对半切。

A-8 · 串起来：tiny thinker 的 KV-cache greedy 解码

Level 1 · I/O 示例

例 1：greedy(prompt=[10, 20, 30, 40], max_new=4) 每次 forward 的参数

第几次 forward	ids	start	positions（[3, T]，三行相同）	返回 logits	forward 后每层 cache 长度	新 token
1（prefill）	[10,20,30,40]	0	[[0,1,2,3]]×3	[4, V]	4	out[0] = argmax(logits[-1])
2	[out[0]]	4	[[4]]×3	[1, V]	5	out[1]
3	[out[1]]	5	[[5]]×3	[1, V]	6	out[2]
4	[out[2]]	6	[[6]]×3	[1, V]	7	out[3]

max_new=4 只需要 4 次 forward：prefill 本身就产生了第一个 token。最后一个生成的 token 不需要再 forward。规律：`start = len(prompt) + len(out) - 1`，并且每次 forward 前 `start == len(cache)`。

例 2：一个随机初始化的 tiny thinker 上的真实数字（vocab=97, hidden=64, 2 层, H=4, KVH=2, D=16, 6 个 expert top-2, mrope_section=[4,2,2]，随机权重）

© 2026 kzhou003

```
prompt = [10, 20, 30, 40]
forward(tensor(prompt), 0, caches)   -> logits [4, 97]，最后一行 argmax = 34，caches 长度 [4, 4]
forward(tensor([34]), 4, caches)     -> logits [1, 97]，argmax = 75，caches 长度 [5, 5]
greedy(prompt, 6)                    -> [34, 75, 75, 75, 17, 1]
HF 每步全量重算的 greedy             -> [34, 75, 75, 75, 17, 1]   ✓
```

同一个模型上的两种 bug：

```
decode 时 start 多加 1 (len(prompt)+len(out))     -> [34, 75, 75, 75, 75, 75]   # 第 5 个 token 开始不同
每步都把完整序列 ids 从 start=0 喂进同一组 caches  -> [34, 77, 2]，3 步后 cache 长度已是 15
```

位置错误往往不会立刻改变 argmax，而是在若干步后才"分叉"——所以要比较整段序列，而不是只看第一个 token。

例 3：纯文本 MRoPE 等于普通 1D RoPE

```
positions = arange(4, 6).expand(3, 2)  -> [[4, 5], [4, 5], [4, 5]]
cos, sin = mrope_cos_sin(rope_inv_freq(16, 10000.0), positions, [4, 2, 2], interleaved=True)
cos.shape -> [2, 16]
cos 与"只用 t 轴位置 [4, 5] 的普通 RoPE"的 cos 完全相同（allclose = True）
```

三条轴位置相同时，无论每个频率选哪条轴，结果都一样。但仍应走 M01 的 MRoPE 接口（interleaved=True），保持与多模态时同一条代码路径。

关键问题

1. PackedLayer 里只有打包后的 `qkv` 和 `w13`，而 M02 的 `AttnWeights` 要分开的 q/k/v、M03 的 `MoeWeights` 要分开的 `w_gate` / `w_up`。分别按什么尺寸切?
2. `forward(ids, start, caches)` 中，positions 张量的形状是什么？三条轴分别填什么?
3. RoPE 的 cos/sin 每次 forward 都要重新算吗？它对应的是"新 token"的位置，还是"cache 中全部 token"的位置?
4. M02 的 attention 自己根据 `len(cache)` 决定因果 mask；那 `start` 和 `len(cache)` 之间应该满足什么关系？不满足时会出现什么现象?
5. lm_head 来自 state 的哪个 key？它为什么不经过 `load_thinker_text`？logits 是 `h @ lm_head` 还是 `h @ lm_head.T`？
6. greedy 一共调用多少次 forward（用 max_new 表示）？第 k 次 decode 喂进去的是哪个 token、start 是多少?
7. caches 应该在哪里创建：`__init__`、`greedy` 开头，还是每次 forward？为什么?

细节

- **Dims** 给出了切分与 RoPE 所需的全部超参：`num_heads` / `num_kv_heads` / `head_dim` 决定 qkv 如何 split；`top_k` / `norm_topk_prob` 进入 MoeWeights；`rope_theta` / `mrope_section` 给 M01；`eps` 同时用于所有 RMSNorm（包括 attention 里的 q_norm/k_norm）。
- **"text-only MRoPE positions"**：Qwen3-Omni 的 RoPE 有 (t, h, w) 三条轴。纯文本时三条轴都等于该 token 的绝对位置，即 `[3, T]` 的每一行都是 `start..start+T-1`。**interleaved layout** 指 M01 中 `interleaved=True` 的频率分配方式（Qwen3 用这个，Qwen2.5 用分块）。
- **残差结构（pre-norm）**：`h = h + attn(rms(h, ln1))`，`h = h + moe(rms(h, ln2))`——归一化只作用在子层的输入上，残差加的是未归一化的 h。最后一层之后再做一次 `rms(h, norm)` 才乘 lm_head。
- **One KVCache per layer**：`caches` 是长度为 num_layers 的列表，第 l 层只读写 `caches[l]`；M02 的 `attention_forward` 会就地追加 K/V，所以 forward 不需要返回 cache。
- **"Must equal HF full-recompute greedy"**：HF 参考每步都对完整序列从头算并取最后一行 argmax。你的 KV-cache 版本数值上会有极小浮点差异，但 argmax 序列必须完全一致——任何 RoPE 位置、mask、归一化顺序的错误都会让序列在某一步分叉。
- **prefill 返回全部行**：forward 返回 `[T, V]` 的所有行（不仅是最后一行），这样它也能被用来逐行对比 prefill 的 logits；greedy 只用 `logits[-1]`。

Level 2 · 思路与常见坑

- **__init__ 的工作**：(1) `load_thinker_text(state, dims.num_layers)` 得到 `{embed_tokens, norm}` 和 PackedLayer 列表；(2) 取 `state["thinker.lm_head.weight"]`；(3) 把每个 PackedLayer 转成 M02 的 `AttnWeights` 和 M03 的 `MoeWeights`（连同 ln1/ln2）存起来；(4) 预先算好 `inv_freq = rope_inv_freq(head_dim, rope_theta)`。
- **权重还原**：`q, k, v = L.qkv.split([H*D, KVH*D, KVH*D], dim=0)`；`I = L.w13.shape[1] // 2`，`w_gate = L.w13[:, :I]`、`w_up = L.w13[:, I:]`，`w_down = L.w2`。不要用 `chunk(2, dim=0)`——那是沿 expert 维切。
- **forward 的步骤**：positions → cos/sin（一次，所有层共享）→ `h = embed[ids]`（`[T, hidden]`）→ 逐层 (attn, moe) 残差 → 最终 norm → `@ lm_head.T`。cos/sin 只对应本次的 T 个新 token。
- **与 M02 的契约**：M02 把新 token 放在绝对位置 `len(cache) .. len(cache)+T-1` 来做因果 mask；而 RoPE 位置由你传入的 cos/sin 决定。两者必须一致，即每次调用时 `start == len(caches[0])`。可以在 forward 里加一个 assert 帮助调试。

© 2026 kzhou003

- **greedy 的循环**：caches 在 greedy 开头新建（每次 greedy 都是新请求）；prefill 一次得到第一个 token；之后 `while len(out) < max_new`：只喂 `out[-1:]`，`start = len(prompt) + len(out) - 1`。总 forward 次数 = max_new。
- **常见 bug 与症状**：start 差一 → 若干步后才分叉（例 2：第 5 个 token 起不同）；每步喂整段 → cache 重复增长、很快乱码；把 rms 的结果当作残差主干（`h = rms(h) + attn(...)`）→ prefill logits 立刻不对；w13 两半切反 → MoE 输出全错，prefill logits 就不对；忘了最终 norm → logits 量级明显偏大。
- **调试顺序**：先只看 prefill 的 logits 与参考是否 allclose（检查权重、norm、MoE、RoPE）；再看 greedy 序列（检查 cache 与 start）。前者错时，多半是 M01–M03/M07 中某个前置题的问题。
- **dtype/no_grad**：greedy 用 `@torch.no_grad()`；`torch.tensor(prompt)` 是 long，可直接索引 embed；token 用 `int(logits[-1].argmax())` 转成 Python int 放进 list。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

- qkv 按行 split 为 `[H·D, KVH·D, KVH·D]`；w13 在 dim 1 对半：前 I 行是 gate，后 I 行是 up（I = w13.shape[1] // 2）。
- positions 是 `[3, T]` 的 long 张量，三行都是 `arange(start, start+T)`（如 `torch.arange(start, start+T).expand(3, T)`）。
- 每次 forward 重算，且只对应新 token 的位置；cache 中旧 token 的 K 在写入 cache 前已经用它们自己的位置旋转过了。
- 必须 `start == len(cache)`。否则 RoPE 的相对距离和 mask 认为的位置不一致，输出悄悄偏离，常在若干步后分叉。
- `state["thinker.lm_head.weight"]`，形状 `[V, hidden]`；它不在 `thinker.model.` 下，所以 load_thinker_text 不处理。logits = `h @ lm_head.T` → `[T, V]`。
- 共 max_new 次 forward：1 次 prefill + (max_new - 1) 次 decode。第 k 次 decode（k 从 1 开始）喂 `out[k-1]`，start = len(prompt) + k - 1。
- 在 greedy 开头创建，每层一个；放 `__init__` 会让第二次 greedy 继承上一次的历史，放 forward 里则等于没有 cache。

形状追踪（hidden=64, H=4, KVH=2, D=16, V=97, prompt 长 4）

```
prefill: ids [4] -> positions [3, 4] -> cos/sin [4, 16]
         h = embed[ids]                      [4, 64]
         rms(h, ln1) -> attention_forward    [4, 64]   cache: 0 -> 4
         rms(h, ln2) -> moe_forward          [4, 64]
         (第 2 层同理)
         rms(h, norm) @ lm_head.T            [4, 97]
decode:  ids [1], start=4 -> positions [3, 1] -> cos/sin [1, 16]
         ...                                 [1, 97]   cache: 4 -> 5
```

greedy 骨架 自检：在循环里 `assert start == len(caches[0])`；再写一个“每步全量重算”的慢版本（每次新建 caches，从 start=0 喂 prompt+out），两者的 token 序列必须完全相同。

A-9 · Orchestrator v1：线性多 stage 路由

Level 1 · I/O 示例

例 1（我们一直在用的那个）：3 个 stage S0→S1→S2（thinker→talker→code2wav），`final_stage_ids={0,2}`，`forward_fn = lambda nxt, prompt, prev: prev`（原样返回 prev_data）。

```
await orch.add_request("r1", "hi there")

S0.submit("r1", "hi there")          -> S0 吐出 "hi", "there"(finished)
S1.submit("r1", ["hi", "there"])     -> S1 吐出 ["HI","THERE"](finished)    （只有一块）
S2.submit("r1", [["HI", "THERE"]])   -> S2 吐出 "wav:HI", "wav:THERE"(finished)
```

用户依次 `await orch.next_output()` 拿到 4 条：

#	stage_id	data	finished
1	0	`"hi"`	False
2	0	`"there"`	**?**（你来回答）
3	2	`"wav:HI"`	False
4	2	`"wav:THERE"`	True

stage 1 的输出一条也不会出现在用户侧。结束后 `orch.request_states == {}`，`orch.dropped_outputs == 0`。提示：全程恰好一条 `finished=True`，并且是最后一条——用这条规则推出第 2 行。

例 2：只有最后一个 stage 是 final。同样的三个 stage，`final_stage_ids={2}`，prompt 是 `"a b c"`。

© 2026 kzhou003

```
S0 收到 "a b c"            -> 吐 "a","b","c"(fin)       用户看不到
S1 收到 ["a","b","c"]      -> 吐 ["A","B","C"](fin)     用户看不到
S2 收到 [["A","B","C"]]    -> 吐 "wav:A","wav:B","wav:C"(fin)

用户收到:
  OrchOutput("r1", stage_id=2, data="wav:A", finished=False)
  OrchOutput("r1", stage_id=2, data="wav:B", finished=False)
  OrchOutput("r1", stage_id=2, data="wav:C", finished=True)
```

注意：S0 虽然不是 final，它的完成仍然要触发向 S1 的转发——“给不给用户”和“要不要转发”是两件独立的事。
例 3：未知 request 的输出。和例 1 相同的配置，但：

```
1) 在 add_request 之前，S1 的输出通道里先冒出一条 StageOutput("ghost", "?", True)
2) 正常跑完 r1（用户照样收到例 1 的 4 条）
3) r1 的终态发出之后，S2 又迟到一条 StageOutput("r1", "wav:LATE", True)

结果：用户侧没有任何多余消息；orch.request_states == {}；orch.dropped_outputs == 2
```

“ghost” 从来没注册过，“r1” 已经被清理过——对 orchestrator 来说两者都是未知 id，处理方式一样。

关键问题

1. 例 1 中 S0 的最后一块 `"there"` 自带 `finished=True`。发给用户的那条 `OrchOutput` 的 `finished` 应该是什么？它由 `StageOutput.finished` 决定，还是由别的东西决定？
2. S0 的最后一块既属于 final stage（要给用户），又是 S0 的完成事件（要转发给 S1）。这一条输出是否两件事都要做？如果是，先做哪件有关系吗？
3. S2 拿到的输入为什么是 `[["HI","THERE"]]` 而不是 `["HI","THERE"]`？`forward_fn` 的第三个参数“stage s 产出的全部数据”具体是哪个列表、什么时候攒出来的？
4. 为什么必须在 `await stages[0].submit()` **之前**把 `ReqState` 放进 `request_states`？在那个 await 让出控制权期间，别的 task 可能做什么？
5. 请求清理之后，又有一条属于它的输出到达，会发生什么？如果此时你用 `setdefault` 之类的写法“顺手”建一个新的 ReqState，会出现什么现象？
6. reader task 读到一批输出后为什么不自己直接处理（改状态、发给用户、转发），而是丢进 `_ready_q` 交给唯一的 dispatcher？asyncio 下没有锁，正确性靠什么保证？
7. `final_stage_ids={0}` 时 `last_stage` 是多少？S1、S2 会被 submit 吗？

细节

- **两种输出类型，两个方向**：`StageOutput(request_id, data, finished)` 是 stage→orchestrator，通过 `StageClient.get_output()` 拿到，而且一次返回的是 `list[StageOutput]`（一个批次，里面可能混着多个请求）。它**没有** stage_id 字段——哪个 stage 来的只有读它的那个 reader 知道。`OrchOutput(request_id, stage_id, data, finished, error)` 是 orchestrator→用户，只为 final stage 生成。
- `last_stage = max(final_stage_ids)`（构造函数里已经算好）：管线只需要跑到最后一个 final stage 为止；它之后的 stage 根本不需要提交。“s 完成且 s < last_stage 才转发”就是这个意思。
- **ReqState 各字段**：`submitted` = 已经提交过的 stage 集合；`finished` = 已报告 finished 的 stage 集合；`stage_data[s]` = stage s 对这个请求产出的所有 data，按到达顺序；`pending_final` 在 P02 里不用（P03 用）。
- **“恰好一条 finished=True，而且是最后一条”**：这里的 finished 是**请求级**的“整个请求结束了”，不是“某个 stage 结束了”。docstring 写的条件是“every final stage finished”，可以表述为集合包含关系 `final_stage_ids ⊆ st.finished`。
- **三个队列**：`_req_q`（add_request 投递的控制消息 → request handler），`_ready_q`（所有 reader 共享 → dispatcher），`_out_q`（dispatcher → `next_output()`）。`next_output(timeout)` 的 timeout=None 表示一直等。
- **单写者（single-writer）**：asyncio 是协作式调度，两个 await 之间的代码不会被别的 task 打断。所以只要“读状态—改状态”中间没有 await，就不需要锁。“唯一 dispatcher 串行处理”就是把所有状态修改放进同一个 task 里排队执行。

Level 2 · 思路与常见坑

- **任务拆分**（`start()` 里创建，存进 `self._tasks`）：

task	数量	做什么
`_reader(sid, client)`	每个 stage 一个	死循环 `await client.get_output()`，把 `(sid, batch)` 放进 `_ready_q`。只搬运，不碰状态。
`_request_handler()`	1	从 `_req_q` 取 add 消息，调用 `_handle_add`。
`_dispatcher()`	1	从 `_ready_q` 取一批，**按顺序**对每条调用 `_handle_output(sid, o)`。

`shutdown()`：cancel 所有 task，再 `gather(..., return_exceptions=True)` 等它们真正退出（否则会有 “Task was destroyed but it is pending” 警告）。

© 2026 kzhou003

- **为什么 reader 要带上 sid**：StageOutput 不知道自己来自哪个 stage。reader 是唯一知道的地方，所以入队时打包成 `(sid, outs)`。一个 stage 的输出经过"一个 reader → 一个 FIFO 队列 → 一个 dispatcher"，同一 stage 内的顺序天然保持。
- **`_handle_output` 的决策顺序**（建议先写成注释再填代码）：① 查状态，没有就 drop 计数并返回；② 把 data 追加到 `stage_data[sid]`；③ 若 `o.finished`，把 sid 加入 `finished`；④ 若 sid 是 final stage：算出请求级 done，发一条 OrchOutput；done 的话清理并**立刻返回**；⑤ 若 `o.finished` 且 `sid < last_stage`：用 forward_fn 构造输入，标记 submitted，submit 给 sid+1。
- **先改状态，再 await**：dispatcher 里的 `await self._out_q.put(...)` 和 `await stages[nxt].submit(...)` 都是让出点。在任何 await 之前把 stage_data / finished / submitted 都更新好，这样别的 task 插进来时看到的都是一致的状态。
- **add 路径**：`add_request` 只负责把 `("add", rid, prompt)` 放进 `_req_q`；真正的注册+提交在 request handler 里做。注册（同步的 dict 赋值）必须排在第一个 await 之前。
- **常见错误 1：直接用 `o.finished` 当 OrchOutput.finished**。表现：例 1 里用户会收到两条 finished=True（S0 的 "there" 和 S2 的 "wav:THERE"），而且第一条不是最后一条；如果你在第一条 finished 时就清理，S2 的所有输出都会变成 dropped。
- **常见错误 2：每来一块就转发**（而不是等 stage 完成）。表现：S1 被同一个请求 submit 多次；或者转发时拿到的 prev_data 不完整。**常见错误 3：清理后忘了 return**，继续往下走去读已经被删除的状态（KeyError）或者做转发。
- **常见错误 4：先 submit 再注册**。如果 stage 在 submit 内部（或 submit 让出期间）就吐出第一条输出，dispatcher 会把它当未知 id 丢掉。在例 1 里的表现是：用户只看到 stage 0 的 `"there"`，S1 收到的是 `["there"]`，最终 S2 只吐 `"wav:THERE"`，`dropped_outputs == 1`——不会卡死，但数据静悄悄地少了一块。另外：某个 task 里抛出的异常不会自动打印，task 就安静地死了，表现为 `next_output` 超时。调试时可以检查 `task.done() and task.exception()`。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

例 1 的逐步状态表（`final_stage_ids={0,2}`，forward_fn 原样返回 prev_data）：

步	dispatcher 处理	stage_data	finished	动作
0	(request handler) add r1	{}	{}	注册 ReqState，submitted={0}，然后 `await S0.submit("r1","hi there")`
1	`S0: ("hi", fin=False)`	`{0:['hi']}`	{}	0 ∈ final → 发 `(0,'hi',False)`；未完成，不转发
2	`S0: ("there", fin=True)`	`{0:['hi','there']}`	{0}	0 ∈ final；{0,2} ⊄ {0} → 发 `(0,'there',False)`（**False**！）；0 完成且 0 < 2 → `forward_fn(1,'hi there',['hi','there'])`，submitted={0,1}，submit S1
3	`S1: (['HI','THERE'], fin=True)`	`{…, 1: [['HI','THERE']]}`	{0,1}	1 ∈ final → 不给用户；1 完成且 1 < 2 → `forward_fn(2, …, [['HI','THERE']])`，submitted={0,1,2}，submit S2
4	`S2: ("wav:HI", fin=False)`	`{…, 2:['wav:HI']}`	{0,1}	发 `(2,'wav:HI',False)`
5	`S2: ("wav:THERE", fin=True)`	`{…, 2: ['wav:HI','wav:THERE']}`	{0,1,2}	{0,2} ⊆ finished → 发 `(2,'wav:THERE',True)`，清理，return（2 == last_stage，本来也不会转发）
6	`S2: ("wav:LATE", fin=True)`（假设迟到）	—	—	r1 已不在 request_states → `dropped_outputs += 1`

开放问题的答案

1. **S0 最后一块的 finished**：False。OrchOutput.finished 是请求级的，条件是 `final_stage_ids ⊆ st.finished`；此时 finished={0}，不包含 2。只有 S2 的最后一块那一刻条件才首次成立，所以"恰好一条、且是最后一条"自动满足。
2. **两件事都做吗**：都做。"sid ∈ final → 发给用户" 和 "o.finished 且 sid < last_stage → 转发" 是两个独立的 if，不是 if/elif。顺序上先发给用户、再转发更自然（用户的文本不必等 S1 提交完）；但真正的硬约束只有一个：如果这条已经是终态（done），发完就清理并 return，不能再转发。
3. **forward 的输入**：`forward_fn(s+1, st.prompt, st.stage_data[s])`。stage_data[s] 是 dispatcher 每处理一条 stage s 的输出就 append 一次攒出来的列表，所以 S1 的输入是 S0 两块的列表 `["hi","there"]`；S1 只吐了一块，这一块本身是 list，所以 S2 收到的是"一个元素的列表" `[["HI","THERE"]]`。拆包/拼接是 forward_fn（真实系统里是 thinker2talker 这类函数）的事，orchestrator 不关心 data 长什么样。
4. **为什么先注册再 await submit**：await 会把控制权交还给事件循环。stage 可能在 submit 里面就把第一条输出放进了它的输出通道，reader 和 dispatcher 随即运行——如果这时 rid 还不在 request_states 里，这条输出就被当成未知 id 丢掉（见 Level 2 的常见错误 4）。dict 赋值是同步的，放在 await 前面就没有任何窗口。

© 2026 kzhou003

5. **清理后的迟到输出**：查不到状态 → `dropped_outputs += 1` → return。绝不能重建状态：重建出来的 ReqState 永远等不到终态，会一直留在 request_states 里（泄漏），还可能把一条孤儿输出发给用户。
6. **为什么单 dispatcher**：如果三个 reader 各自改状态，它们会在各自的 await 点交错，例如两个 reader 都看到"还没 done"而各发一条，或者一个刚清理另一个又写入。把所有修改放进一个 task 串行执行，就等价于有一把全局锁，却不用写锁。
7. `final_stage_ids={0}`：last_stage=0，S0 完成就是终态，S1/S2 永远不会被 submit。

骨架（注册与输出处理）P03 会继承这个类并只覆盖 `_handle_add` / `_handle_output`，所以把三个循环和 cleanup 写成独立方法会让下一题轻松很多。

A-10 · Orchestrator v2：async_chunk 预热 + 终态守门

Level 1 · I/O 示例

例 1：talker（S1，非 final）的完成报告迟到。`final_stage_ids={0,2}`，`prewarm_fn = lambda sid, p: f"warm{sid}:{len(p)}"`，`add_request("r1", "hi there")`。

```
add 之后三次 submit（按顺序）：
  S0.submit("r1", "hi there")
  S1.submit("r1", "warm1:8")
  S2.submit("r1", "warm2:8")
```

之后 stage 按下表顺序报告（数据本身走 connector，这里报告的只是给 orchestrator 看的输出）：

到达的 StageOutput	用户此刻收到	ReqState.finished	pending_final
`S0 ("hi", False)`	`(0,'hi',False)`	{}	None
`S0 ("there", True)`	`(0,'there',False)`	{0}	None
`S2 ("wav:HI", False)`	`(2,'wav:HI',False)`	{0}	None
`S2 ("wav:THERE", True)`	**（什么都没有）**	{0,2}	`(2,'wav:THERE',True)`
`S1 (None, True)`	`(2,'wav:THERE',True)`	（已清理）	—
`S2 ("wav:EXTRA", False)`	（什么都没有）	—	—；`dropped_outputs == 1`

例 2：正常顺序（S1 在 S2 之前完成）。配置同上。

到达	用户收到
`S0 ("hi", False)`	`(0,'hi',False)`
`S0 ("there", True)`	`(0,'there',False)`
`S1 ("tok", True)`	（无：S1 不是 final）
`S2 ("wav:HI", False)`	`(2,'wav:HI',False)`
`S2 ("wav:THERE", True)`	`(2,'wav:THERE',True)`，随即清理

这里不需要 pending_final：最后一个 final stage 完成的那一刻，所有已提交的 stage 也都完成了。

例 3：最后完成的 final stage 是 S0（S2 先结束，文本最后才结束）。配置同上。

到达	用户收到	finished
`S0 ("hi", False)`	`(0,'hi',False)`	{}
`S2 ("wav:HI", True)`	`(2,'wav:HI',False)`	{2}
`S1 (None, True)`	（无）	{1,2}
`S0 ("there", True)`	`(0,'there',True)`，清理	—

终态消息的 stage_id 不一定是 last_stage——它是"让条件第一次成立的那条输出"所在的 stage。

关键问题

1. P02 里 `submitted` 是随着转发一点点增长的（add 时只有 {0}）；P03 里它在 add 时就要包含 0..last。如果你沿用 P02 的写法只标了 {0}，例 1 会在哪一步出错、错成什么样？把标记放在第一次 await 之前又有什么好处？
2. 例 1 第 4 行：最后一个 final stage 完成了，但 S1 还没完成。这一刻用户应该收到什么？你要存什么、存在哪里？

© 2026 kzhou003

3. 一个**非 final** stage 报告完成时，handler 要做什么？在什么条件下它什么都不用做？
4. “所有 final stage 都完成”和“所有已提交的 stage 都完成”是两个不同的集合条件。分别写出来。它们各自控制什么（OrchOutput.finished 的值？是否立即发出？是否清理？）
5. 非 final stage 的 data 会出现在用户侧吗？P02 里的 `stage_data` 在本题还有用吗？
6. 本题继承你的 P02：reader、dispatcher、request handler、`_cleanup`、dropped 计数，哪些可以原封不动复用？
7. pending_final 已经存好之后，又来了一条属于这个请求的、非 finished 的 S1 输出，应该怎么处理？

细节

- **async_chunk 模式**：stage 之间的数据走 connector（共享内存/旁路通道），orchestrator 不再调用 forward_fn。构造函数里传的 `forward_fn=lambda *_: None` 只是为了满足父类签名。
- **prewarm（预热）**：下游 stage 在请求刚到时就收到一个占位输入 `prewarm_fn(sid, prompt)`，让它在自己的 scheduler 里先占好位、等第一个 chunk。提交顺序：先 S0（真实 prompt），再 S1、S2……（占位）。提交范围是 `0..last_stage`。
- **两道门**：`OrchOutput.finished` 的取值仍然是 P02 的“所有 final stage 都完成”；但这条 finished=True 的消息**何时发出**多了一个条件：所有 submitted 的 stage 都完成（`submitted ⊆ finished`）。
- `ReqState.pending_final`：存的是一条已经构造好的、`finished=True` 的 OrchOutput。它被“扣下”而不是丢弃——之后原样发出，内容（stage_id、data）就是那条最后的 final 输出。
- **为什么要守门**：真实系统里终态之后会清理请求，回收 SHM、connector 缓冲、replica 绑定。如果 talker 还在往 connector 写，提前清理会回收它正在用的资源；它稍后的完成报告也会变成“未知请求”。
- **继承**：只覆盖 `_handle_add` 和 `_handle_output`。如果你的 P02 把这两步写死在 dispatcher 里而不是独立方法，先回去重构 P02。

Level 2 · 思路与常见坑

- **add 路径**：建 ReqState → 注册进 request_states → `submitted = {0..last}`（全部同步完成）→ `await S0.submit(prompt)` → 依次 `await S_k.submit(prewarm_fn(k, prompt))`。每个 stage 只 submit 一次。
- **输出路径的决策表**（先查状态、未知就 drop；然后若 finished 就加入 st.finished）：

这条输出来自	all_final（final ⊆ finished）	all_done（submitted ⊆ finished）	动作
final stage	否	—	发 `OrchOutput(..., finished=False)`
final stage	是	是	发 `finished=True` 的消息，清理
final stage	是	否	构造 finished=True 的消息，存进 pending_final，**不发**
非 final stage	—	是，且有 pending_final	发 pending_final，清理
非 final stage	—	其他	什么都不做

- **submitted 是守门的依据**：`submitted ⊆ finished` 里 submitted 少一个 stage，门就少一道。沿用 P02 只标 {0} 的后果见 Level 3。在第一次 await 之前一次性标完，和 P02“先注册再 submit”是同一个原则：任何让出点之前状态已经完整，不需要再去推敲哪些交错是安全的。
- **常见错误 1：非 final stage 完成时不检查 pending_final**。表现：例 1 里用户永远收不到终态，`next_output` 超时；request_states 里留着这个请求。
- **常见错误 2：存进 pending_final 的同时也发了出去**，或者发 pending 后忘记清理。表现：终态重复（两条 finished=True），或请求状态泄漏、后续输出不被 drop。
- **常见错误 3：直接沿用 P02 的 _handle_output**（父类的会在 S0 完成时调用 forward_fn 并 submit S1）。表现：S1 被 submit 两次（一次 prewarm，一次转发），第二次的输入是 None。
- **常见错误 4：未知 id 时“复活”状态**（例如 `setdefault`）。表现：清理后迟到的 S2 输出建出一个新的 ReqState（submitted 为空，门形同虚设），这条迟到输出被发给用户、出现在终态之后，状态还永久泄漏在 request_states 里。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

例 1 的状态演化（✓ 表示该输出 finished=True）：

事件	submitted	finished	all_final	all_done	pending	发给用户
add	{0,1,2}	{}			None	
`S0 hi`	{0,1,2}	{}	F	F	None	`(0,hi,F)`
`S0 there ✓`	{0,1,2}	{0}	F	F	None	`(0,there,F)`

© 2026 kzhou003

`S2 wav:HI`	{0,1,2}	{0}	F	F	None	`(2,wav:HI,F)`
`S2 wav:THERE` ✓	{0,1,2}	{0,2}	**T**	F	`(2,wav:THERE,T)`	—
`S1` ✓	{0,1,2}	{0,1,2}	T	**T**	→ 发出	`(2,wav:THERE,T)`，清理
`S2 wav:EXTRA`	—	—				dropped += 1

Level 1 问题的答案

1. 只标 {0} 时，例 1 在 S2 的 `"wav:THERE"` 到达那一刻：finished={0,2}，`{0} ⊆ {0,2}` 成立，于是终态立即发出并清理——正是本题要防止的“提前终态”；随后 S1 迟到的完成报告变成未知 id，`dropped_outputs == 1`。把 0..last 在第一次 await 之前标完，则无论 stage 多快、在哪个 await 点插进来的输出，看到的都是完整的 submitted。
2. 什么都不发。构造好 `finished=True` 的那条 OrchOutput，存进 `st.pending_final`，直接 return。
3. 把 sid 加进 finished 后，检查 `submitted ⊆ finished` 且 `pending_final is not None`；两者都成立就发出 pending 并清理。若 pending 为空（说明还有 final stage 没完成，例 2 的 S1），什么都不做——将来那个 final stage 完成时会走“立即发出”分支。
4. all_final = `final_stage_ids ⊆ finished`：决定 OrchOutput.finished 的值。all_done = `submitted ⊆ finished`：决定 finished=True 的消息是立即发出（并清理）还是先扣在 pending_final。
5. 不会；非 final stage 的输出只用来更新 finished。stage_data 在本题不需要（不再转发），记不记都不影响结果。
6. start / shutdown / add_request / next_output / reader / dispatcher / request handler / _cleanup 全部复用；只换 _handle_add 和 _handle_output。
7. 它不是 finished，不改变任何集合，按决策表落入“什么都不做”。

骨架注意 `_cleanup` 在 `await self._out_q.put` 之后：对无界队列来说 put 不会真正挂起，但即使挂起，状态也已经是一致的（pending 已取出）。

A-11 · OutputSink：线程 → asyncio 的批量唤醒

Level 1 · I/O 示例

例 1：单线程里观察唤醒计数（都在同一个 loop 里，便于看清每一步）

```
s = OutputSink(); s.bind_loop()
s.put("a"); s.put("b"); s.put("c")
```

调用	返回	wakeups_scheduled	说明
`3 次 put`	—	1	第一次 put 发起唤醒；后两次看到“唤醒在途”，不再调度
`await s.get_batch(max_items=2)`	`['a','b']`	1	队列非空，立刻返回，不睡
`await s.get_batch(max_items=2)`	`['c']`	1	同上
`s.put("d")`	—	1	上一轮唤醒还“在途”（消费者没有重新武装过），所以不调度
`await s.get_batch()`	`['d']`	1	第一次 drain 就拿到了
`await s.get_batch(timeout=0.05)`	`[]`	1	约 0.05s 后超时；这一次消费者真正“武装”了
`s.put("e")`	—	**2**	武装之后的第一次 put 才会再调度一次唤醒
`await s.get_batch()`	`['e']`	2	

例 2：在“空 drain 之后、武装之前”插入一次 put

```
s = OutputSink(); s.bind_loop()
s.put("a")
await s.get_batch()            # -> ['a'], wakeups_scheduled == 1, “在途”标志没被清掉

def hook():                    # 另一个线程 put，并等它完成
    t = threading.Thread(target=lambda: s.put("b")); t.start(); t.join()
```


© 2026 kzhou003

```
    s.after_empty_drain = lambda: None
s.after_empty_drain = hook

await s.get_batch(timeout=2.0) # 正确实现：几乎 0 秒返回 ['b']；wakeups_scheduled 仍然 == 1
```

那次 put 看到的是“唤醒在途”，所以没有调度唤醒。一个只 drain 一次就去睡的实现会白白睡满 2 秒。

例 3：消费者已经在等，另一个线程连 put 5 条

```
task = asyncio.create_task(s.get_batch(timeout=2))   # 先让它睡下
await asyncio.sleep(0.01)
threading.Thread(target=lambda: [s.put(i) for i in range(5)]).start()  # 然后 join
await task          # -> [0, 1, 2, 3, 4]
s.wakeups_scheduled # -> 1
```

5 次 put 只产生 1 次 `call_soon_threadsafe`；loop 醒来时 5 条都已经在队列里，一批拿走。（具体能凑成几批取决于线程时序，但唤醒次数必须是 1。）

关键问题

1. “唤醒在途”这个标志由谁置为 True、由谁置回 False？在一次 `get_batch` 直接从队列拿到数据就返回的情况下，它会被置回吗？这样会有问题吗？（对照例 1 第 4 行）
2. 为什么 `put()` 不能直接调用 `event.set()`？`asyncio.Event` 和 `loop.call_soon_threadsafe` 各自对线程有什么假设？
3. `put()` 里“append 数据”和“检查/设置标志”谁先谁后？反过来会出现什么交错？
4. 消费者“发现队列为空”和“武装（准备睡觉）”之间有一个窗口。如果一次 put 恰好落在这个窗口里，并且它看到标志仍是 True，会发生什么？要怎样才能不错过它？
5. `event.clear()` 应该在什么时候调用？如果从不 clear，或者在 drain 之前 clear，会怎样？
6. `timeout` 是整个 `get_batch` 调用的总时长，而循环可能转好几圈（比如被唤醒了但 drain 出来是空的）。怎么保证总时长不超？
7. `after_empty_drain` 钩子里可能有另一个线程调用 `put()`，而钩子会等那个线程结束。如果你调用钩子时手里正拿着 put 也要用的锁，会怎样？

细节

- **唤醒周期（wake cycle）**：从“标志 False→True（某次 put 调度了唤醒）”开始，到“消费者重新武装（把标志置回 False）”为止。一个周期内不管 put 多少次，`call_soon_threadsafe` 最多一次；`wakeups_scheduled` 就是周期数。
- **跨线程唤醒只有一条合法路径**：`asyncio.Event` 不是线程安全的，只能在它所属的 loop 线程上 set。别的线程要用 `loop.call_soon_threadsafe(event.set)`：把回调塞进 loop 并写 self-pipe 叫醒 selector。这一步有锁和系统调用，所以要合并。loop 已关闭时它会抛 `RuntimeError`，生产者可以忽略。
- **“never blocks”**：put 可以拿一把很短的 `threading.Lock`（只保护标志的检查并设置），但不能等消费者。`collections.deque` 的 `append` / `popleft` 在 CPython 下本身是原子的，可以不加锁。
- **“1..max_items 条或 []”**：只要队列里有东西就至少返回 1 条且不超过 max_items，剩下的留给下一次；只有在超时的情况下才返回空列表。
- **bind_loop**：不传参数时用当前正在运行的 loop（`asyncio.get_running_loop()`），并在这里创建 Event。在 bind 之前发生的 put 只需要把数据放进队列即可（没有 loop 可唤醒），消费者第一次 drain 时自然会拿到。
- **钩子的调用位置**：严格地在“一次 drain 发现为空”之后、“重新武装”之前，每次空 drain 调用一次；调用时不要持有任何 put 也需要的锁。

Level 2 · 思路与常见坑

- **状态**：`deque`（数据）、`threading.Lock` + `bool`（“唤醒在途”标志）、`loop`、`asyncio.Event`、`wakeups_scheduled` 计数。标志的“检查并设置”必须是一个原子动作，所以放在锁里；计数在同一个临界区里加。
- **put 的顺序**：先 append，再在锁里“若标志为 False 则置 True 并计数”，出锁后（若需要）`call_soon_threadsafe(event.set)`。先 append 保证：任何人看到“标志已被置起”时，数据一定已经在队列里了。
- **get_batch 是一个循环**：drain → 非空就返回 → 调钩子 → 武装（clear Event、在锁里把标志置 False）→ **再 drain 一次** → 非空就返回 → 等 Event（带剩余超时）→ 回到开头。武装和第二次 drain 之间不能有 await。
- **为什么第二次 drain 能堵住窗口**：对任何一次并发的 put，它的“看标志”要么发生在武装之前，要么在之后。之前：它看到 True、没调度唤醒，但它的 append 也已经发生（append 在看标志之前），第二次 drain 一定能拿到。之后：它看到 False，会调度唤醒，Event 会被 set，wait 会醒。两种情况都不会丢。
- **超时**：进入时用 `loop.time()` 算出 deadline，每次等待前算剩余时间；剩余 ≤ 0 就返回 []。用 `asyncio.wait_for(event.wait(), remaining)`，捕获 `asyncio.TimeoutError` 后再 drain 最后一次（超时的同时可能刚好有数据到）。timeout=None 时 remaining 也是 None，表示一直等。
- **伪唤醒**：Event 可能已经被上一轮遗留的 set 置位（上一轮直接从队列拿走了数据、没有 wait），所以 wait 立即返回、drain 为空。这不是错误，循环再转一圈就会正常武装并睡下——前提是你的实现是循环而不是“只等一次”。
- **常见错误与表现**：
  - 只 drain 一次就睡：例 2 的场景卡到超时；乒乓式交互（每次只 put 一条、等对方回应）会时不时卡住。
  - 每次 put 都 call_soon_threadsafe：功能正确，但 1000 次 put 会产生 1000 次唤醒。

© 2026 kzhou003

- 先看标志再 append：生产者看到 False → 调度唤醒 → 消费者醒来 drain 为空 → 武装、睡下 → 生产者这时才 append，且不再唤醒，数据卡住。
- 用 `asyncio.Queue` 跨线程 put：不是线程安全的，偶发数据丢失或 loop 不醒。
- 持锁调用钩子或持锁 await：死锁。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）
例 2 的交错时间线：

时刻	消费者（loop 线程）	生产者（另一线程）	标志	队列
t0	上一轮从队列拿走 'a' 直接返回（没武装）		True	[]
t1	drain → []		True	[]
t2	调用钩子	append('b')；看到标志 True → 不调度唤醒	True	['b']
t3	clear Event；标志置 False（武装）		False	['b']
t4	**第二次 drain → ['b']，返回**		False	[]
t4'	（朴素实现）直接 await event.wait() → 没人会 set，睡到超时		False	['b']

注意 t4 返回后标志是 False，Event 也是 clear 的：下一次 put 会正常调度唤醒，计数变成 2。
Level 1 问题的答案

1. 标志由 put 置 True、由消费者在武装时置 False。直接从队列拿到数据就返回时不置回——这没问题，因为消费者下一次进来总是先 drain，只有在真正准备睡觉前才武装。代价只是“这一段时间里的 put 不会唤醒”，而此时也不需要唤醒（例 1 第 4 行）。
2. `asyncio.Event` 只能在所属 loop 线程上操作；`call_soon_threadsafe` 是唯一线程安全地让 loop 执行回调并把它从 select 中叫醒的方式。
3. 必须先 append 再看标志。反过来的交错见 Level 2 最后一条。
4. 那次 put 不会唤醒你；唯一的补救是武装之后再 drain 一次（上表 t4）。
5. clear 放在“武装”里，和“标志置 False”挨着、之后紧跟第二次 drain 和 await，中间没有让出点。这样上一轮遗留的 set 被清掉，而武装之后任何 put 都会重新调度 set。由于 `call_soon_threadsafe` 排的回调只有在你 await 时才会执行，clear 和置 False 谁先谁后不影响正确性。从不 clear 的后果很严重：Event 一直是 set，`event.wait()` 不挂起就直接返回，空队列时 get_batch 会在一个不让出的循环里空转：有 timeout 就烧满 CPU 直到超时才返回 []（期间 loop 上其他 task 一个都跑不了），timeout=None 就永远卡死。
6. 入口算一次 deadline，每圈用 `deadline - loop.time()` 作为等待时长。
7. 死锁：钩子里的线程 put 时要拿同一把锁，而钩子在等那个线程结束。调用钩子时不能持锁。

把“第二次 drain”那两行删掉再跑例 2，你会看到它卡满 2 秒——这是理解本题最快的办法。

A-12 · StagePool：粘性绑定、两阶段 abort、故障隔离

Level 1 · I/O 示例
例 1：粘性 + round-robin + evict。一个 stage 有 3 个 replica：`p = StagePool(1, [A, B, C])`（replica id 就是下标 0/1/2）。

调用	返回	说明
`p.pick("x")`	0	新请求，轮到 0
`p.pick("y")`	1	
`p.pick("z")`	2	
`p.pick("w")`	0	轮回到 0
`p.pick("x")`、`p.pick("y")`	0、1	已绑定 → 原样返回（粘性），不推进轮转
`p.bound("q")`	None	只读查询，不会创建绑定
`p.evict(1)`	`['y']`	B 死了：返回并删除绑定在 1 上的请求
`p.live_replicas()`	`[0, 2]`	
`p.pick("y")`、`p.pick("v")`	∈ {0, 2}	y 的旧绑定已删，按新请求处理；只在存活 replica 里轮转（具体落在哪个取决于你的轮转计数器，只要是存活的 replica 就对）

© 2026 kzhou003

例 2：按 replica 合并 abort，以及失败语义。接着例 1，此时绑定为 `{x:0, y:0, z:2, w:0, v:2}`。

```
await p.abort(["x", "z", "w", "ghost", "v"])

replica 调用（每个 replica 恰好一次）：
  A.abort(["x", "w"])
  C.abort(["z", "v"])
"ghost" 没有绑定 → 忽略。之后 x/z/w/v 的 bound() 都是 None，y 仍是 0。
```

另一个池 `q = StagePool(0, [A, B])`，其中 B 的 abort 会抛 `RuntimeError("B ipc broken")`；绑定为 `{m:0, n:1, o:0}`：

```
await q.abort(["m", "n", "o"])   # -> 抛出 RuntimeError("B ipc broken")
调用记录：A.abort(["m","o"]) 成功；B.abort(["n"]) 抛错
之后：q.bound("m") is None, q.bound("o") is None, q.bound("n") == 1   # n 留着，可以重试
```

例 3：两阶段 abort 与死 replica。三个 stage 的池 s0/s1/s2，请求 `"r9"` 分别绑在 T0、K1、W0 上；processor 里 r9 已生成的部分文本是 `"Hello wor"`。

```
out = await abort_request([s0, s1, s2], proc, "r9")
调用顺序：collect_partial("r9") -> T0.abort(["r9"]) -> K1.abort(["r9"]) -> W0.abort(["r9"]) ->
commit_abort("r9")
out == OrchOutput("r9", "Hello wor", finished=True, error="abort")

换一组池：s0 = [T0]，s1' = [K0]，其中 K0.abort 会抛 RuntimeError；r9 在两个池里都已绑定
await abort_request([s0, s1'], proc, "r9")   # -> 抛出 RuntimeError
调用顺序：collect_partial -> T0.abort(["r9"]) -> K0.abort(["r9"]) 抛错   # 没有 commit_abort
s0.bound("r9") is None（它成功了，已释放）；s1'.bound("r9") == 0（保留，可重试）
```

死 replica：`d = StagePool(2, [W0, W1])`，`a,b,c` 依次 pick 到 0,1,0。

```
handle_dead_replica(d, 0)
-> [OrchOutput("a", None, finished=True, error="stage 2 replica 0 died"),
    OrchOutput("c", None, finished=True, error="stage 2 replica 0 died")]
d.bound("b") == 1（不受影响）；d.live_replicas() == [1]
再 d.evict(1) 之后，d.pick("new") 抛 NoLiveReplica
```

error 的文字内容由你决定，只要非空、能说明原因。

关键问题

1. 绑定表应该是什么形状？`pick` 对已绑定和未绑定的请求分别做什么？round-robin 的计数器在“粘性命中”时要不要前进？
2. round-robin 只在**存活** replica 之间轮转。你是“对存活列表取模”，还是“对全部 replica 取模再跳过死的”？两种写法在 evict 之后的表现有什么不同？
3. `evict` 为什么要删掉绑定，而不是只标记死亡？如果不删，之后对这些请求调用 `pick` 或 `abort` 会发生什么？
4. `abort(ids)` 里 5 个 id 分布在 2 个 replica 上，应该产生几次 replica 调用？没有绑定的 id 呢？如果某个 replica 上一个 id 都没有，要不要调用它？
5. 例 2 中 A 成功、B 抛错。哪些绑定应该释放、哪些保留？为什么“先释放再 await”是错的？
6. 两阶段 abort 里，为什么 `collect_partial` 必须在引擎 abort 之前、`commit_abort` 必须在之后？如果把 commit 放到最前面，而某个 stage 的 abort 失败了，会留下什么样的烂摊子？
7. `abort_request` 对一个根本没绑定 r9 的池调用 `pool.abort(['r9'])`，会发生什么？需要在外面先判断吗？

细节

- **replica id = 下标**：`self.replicas[i]` 的 i 就是 pick / bound / evict / live_replicas 里用的整数。`live_replicas()` 按下标升序返回存活的 id。
- **sticky（粘性）**：请求的 KV cache 和流式中间状态都在第一次分配到的 replica 上，所以同一个 request id 后续每次 pick 都必须返回同一个 replica——只要它还活着。
- `bound` **vs** `pick` **vs** `release`：bound 只读（没有绑定就返回 None，不创建）；pick 可能创建绑定；release 是请求正常结束时删除绑定，对不存在的 id 也不报错。
- **“ONE abort call per replica”**：一次 `StagePool.abort` 调用内，按绑定把 id 分组，每个有 id 的 replica 只调用一次 `ReplicaClient.abort(list)`（真实系统里一次调用 = 一次 IPC）。
- **两阶段（two-phase）**：阶段 1 只读地收集“已经生成的部分结果”（用来给用户发最后一条消息）；阶段 2 等所有引擎确认停止后才提交（丢弃 detokenizer 等前端状态）。和数据库的 prepare/commit 是同一种思路：不可逆的那一步放在最后。
- **注意 OrchOutput 的形状**：本题的 `OrchOutput(request_id, data, finished, error)` 没有 stage_id 字段，和 P02 的不同。abort 的终态 error 固定是 `"abort"`；死 replica 的终态 data 为 None、error 是一段说明文字。

Level 2 · 思路与常见坑

© 2026 kzhou003

- **状态**：`alive: list[bool]`（或死亡集合）、`bindings: dict[str, int]`（request → replica）、一个 round-robin 计数器。全部同步方法，没有并发问题；唯一的 await 在 abort 里。
- **pick**：已绑定且该 replica 活着 → 直接返回。否则：取存活列表，空则抛 `NoLiveReplica`；用计数器在存活列表上取模选一个，计数器 +1，写入绑定。对存活列表取模可以保证 evict 后不会选到死的、也不会让某个 replica 连续被选两次（"取模再跳过死的"会让死 replica 的下一个邻居多拿一份）。
- **evict**：标记死亡 → 找出所有绑在它上面的 request → 从绑定表删除 → 返回列表。注意不要在遍历 dict 时删除（先收集再删）。
- **StagePool.abort**：① 遍历 ids，用绑定表分组成 `{replica: [ids]}`，无绑定的跳过；② 对每组 `await replicas[rep].abort(ids)`，**这一组成功返回之后**再释放这组的绑定；③ 任何一组抛错就让它传出去——已成功的组已经释放，失败组和尚未处理的组保留绑定。组内保持调用方给的顺序。
- **abort_request**：`partial = processor.collect_partial(rid)` → 对每个 pool `await pool.abort([rid])`（顺序执行，不要 try/except 吞掉错误）→ `processor.commit_abort(rid)` → 返回 `OrchOutput(rid, partial, True, "abort")`。
- **handle_dead_replica**：就是 `pool.evict(replica)` 加上"每个受影响请求一条终态错误输出"。别的 replica 上的请求不要碰。
- **常见错误与表现**
  - 每个 id 调用一次 replica.abort：结果对，但 IPC 次数 = 请求数。
  - await 之前就释放绑定：abort 失败后 `bound()` 变成 None，重试时找不到 replica，请求在引擎里继续跑成孤儿。
  - evict 只标记不删绑定：pick 可能对旧请求返回死 replica；handle_dead_replica 之后再 abort 这些请求，会去调用死进程。
  - 用 `asyncio.gather` 并发所有组：一组失败时其他组可能已经成功也可能没执行完，绑定的释放时机变得难以推理（可以做，但要自己处理每组的结果）。
  - commit 放在 abort 之前，或者用 try/finally 保证 commit：abort 失败时前端已忘记请求，引擎却还在吐 token。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

例 1 + 例 2 的状态表（计数器写法：对存活列表取模）：

操作	alive	bindings	rr 计数	返回 / 副作用
初始	[T,T,T]	{}	0	
`pick x`	[T,T,T]	{x:0}	1	0
`pick y`		{x:0,y:1}	2	1
`pick z`		{…,z:2}	3	2
`pick w`		{…,w:0}	4	0
`pick x`		不变	4	0（粘性，不前进）
`evict 1`	[T,F,T]	{x:0,z:2,w:0}	4	`['y']`
`pick y`		{…,y:0}	5	live=[0,2]，4 % 2 = 0 → 0
`pick v`		{…,v:2}	6	5 % 2 = 1 → 2
`abort [x,z,w,ghost,v]`		{y:0}	6	`A.abort([x,w])`；`C.abort([z,v])`

Level 1 问题的答案

1. `dict[request_id, replica_id]`。已绑定且活着 → 返回；否则轮转选择并写入。粘性命中不推进计数器（推进也不算错，但会让分配不均匀）。
2. 建议对存活列表取模：evict 后自动只在活着的里面均匀轮转。另一种写法要处理"跳过"且分布会偏。
3. 绑定代表"这个请求在那个 replica 上有状态"。replica 死了，状态就没了；保留绑定会让 pick 返回死 replica，abort 去调用死进程。删掉后，这些请求已经由 handle_dead_replica 发了错误终态。
4. 2 次（每个有 id 的 replica 一次）；无绑定的 id 忽略；没有 id 的 replica 不调用。
5. A 那组（m、o）释放，B 那组（n）保留，异常抛出。先释放再 await：失败时绑定已经没了，重试无从下手。
6. collect 在前，是因为 commit 会丢掉前端状态，之后就拿不到部分文本了；commit 在后，是因为只有引擎都确认停止，前端才能安全地忘记这个请求。commit 在前 + abort 失败 = 引擎还在为它产出，前端却不认识它（孤儿输出），而且用户已经收到 abort 终态。
7. 该池的分组为空，不会产生任何 replica 调用，直接返回。不需要在外面判断。

A-13 · RpcRouter：共享通道上的 RPC 关联

Level 1 · I/O 示例

例 1：两个线程并发调用，结果乱序回来

© 2026 kzhou003

```
r = RpcRouter()

# send() 只是“把请求发出去”；这里用一个延时线程模拟 router 线程稍后回传结果
def send_for(key, val, delay):
    return lambda: threading.Timer(delay, lambda: r.deliver(key, val)).start()

线程 1: r.call("sleep", send_for("sleep", "ok-sleep", 0.05), timeout=1)   -> "ok-sleep"
线程 2: r.call("wake",  send_for("wake",  "ok-wake",  0.01), timeout=1)   -> "ok-wake"
 ("wake" 的结果先到，但各自只拿到自己的)

# 结果在 send() 内部就同步回来了，也不能丢：
r.call(("rpc", 7), lambda: r.deliver(("rpc", 7), {"gpu_mem": 3}), timeout=0.1)  -> {"gpu_mem": 3}
```

例 2：超时、迟到结果、key 复用、异常结果

```
r.call("ping", lambda: None, timeout=0.1)      # 约 0.1s 后抛 TimeoutError
r.deliver("ping", "pong")                      # -> False（已经没人在等）
r.late_results                                 # -> 1
r.call("ping", lambda: r.deliver("ping", "pong2"), timeout=0.1)   # -> "pong2"（不会读到旧的 "pong"）

r.call("bad", lambda: r.deliver("bad", ValueError("no such method")), timeout=0.1)
# -> 抛出 ValueError("no such method")：结果本身是异常对象时，raise 而不是 return

w = r.register("dup"); r.register("dup")       # 第二次 -> ValueError（key 仍在使用中）
```

例 3：fail_all 广播并锁存

```
3 个线程分别 r.call("w0"/"w1"/"w2", send=lambda: None, timeout=5) 并阻塞
主线程：r.fail_all(RuntimeError("orch dead"))
-> 3 个线程几乎立刻（远小于 5s）各自抛出 RuntimeError("orch dead")

之后：r.call("after", send, timeout=1)  -> 立即抛 RuntimeError("orch dead")，send 根本不会被调用
      r.register("x")                    -> 同样立即抛出
r.late_results 不因 fail_all 变化
```

关键问题

1. 为什么必须先 `register` 再 `send()`？构造一个“先 send 后 register”会永远等下去的时序（提示：例 1 最后一个调用）。
2. waiter 为什么是 `queue.Queue(maxsize=1)`？`deliver` 往里放结果时会不会阻塞？
3. 一个 waiter 会被谁从注册表里移除：`deliver`、`call` 的 finally，还是两者？如果两者都可能，怎么保证不互相踩？
4. `unregister(key, waiter)` 为什么要多传一个 waiter，并且只在“表里那个就是我这个”时才删除？构造一个 A 超时、B 复用同一 key 的时序，说明不比较会删错谁。
5. `deliver` 需要锁吗？锁里应该做什么、锁外做什么？为什么不在持锁时往 waiter 里 put？
6. `waiter.get(timeout=...)` 超时抛的是什么异常？你要对外抛什么？
7. `fail_all` 之后，正在等的、以后才来的调用分别怎么处理？“锁存（latch）”存在哪里，谁来检查它？

细节

- **key 是任意 Hashable**：可以是字符串，也可以是 `("rpc", 7)` 这样的元组（真实系统里通常是 call_id）。同一时刻一个 key 只能有一个活着的 waiter。
- **deliver 的返回值与计数**：有 waiter 时把结果交给它并返回 True；没有（从未注册、已超时注销、已被别人 deliver 过）时返回 False 并 `late_results += 1`。
- **“always unregister in finally”**：不论 send 抛错、等待超时、还是正常拿到结果，call 退出时都要保证自己的 waiter 不再留在表里，否则 key 永远无法复用（再 register 会报重复）。
- **异常作为结果**：结果通道上传回来的可能是一个异常对象（远端执行失败），也可能是 fail_all 广播的致命错误。约定：call 拿到 `BaseException` 实例就 raise 它。
- **“All methods are called from arbitrary threads”**：注册表（dict）和锁存的错误是共享状态，读改写都要在同一把 `threading.Lock` 里；但阻塞操作（等结果）绝不能持锁。

Level 2 · 思路与常见坑

- **状态**：`threading.Lock`、`waiters: dict[key, Queue]`、`fatal: BaseException | None`、`late_results`。
- **register**：锁内：若 fatal 已设置 → raise fatal；若 key 已在表中 → ValueError；否则新建 `Queue(maxsize=1)` 放进表并返回。
- **deliver**：锁内 `pop(key, None)` ——pop 而不是 get，保证同一个 key 只会被交付一次；拿不到就计数并返回 False。锁外 `put_nowait(result)`：容量 1 且刚从表里取出，不会满，也不会阻塞。
- **unregister**：锁内，仅当 `waiters.get(key) is waiter` 时删除。这样 deliver 已经 pop 过的情况、或者 key 已被新调用重新注册的情况都安全。

© 2026 kzhou003

- **call**：`w = register(key)` → try: `send()`；`w.get(timeout)`，`queue.Empty` 转成 `TimeoutError`；结果是异常就 raise，否则 return → finally: `unregister(key, w)`。注意 register 在 try 外面：register 失败时没有东西需要注销。
- **fail_all**：锁内设置 fatal、取出所有 waiter、清空表；锁外逐个 `put_nowait(error)`（防御性地忽略 `queue.Full`）。之后 register 会因 fatal 立即失败，call 也就在 send 之前失败。
- **常见错误与表现**
  - 先 send 后 register：send 内同步 deliver 的结果被计为 late，调用方等满超时。
  - 超时后不注销：同一 key 第二次 call 报“重复 key”；或者（若 register 允许覆盖）读到上一轮迟到的旧结果。
  - unregister 不比较身份：A 超时的 finally 把 B 刚注册的 waiter 删掉，B 的结果被当成 late，B 卡到超时。
  - 持锁 put / 持锁 get：死锁或所有调用串行化。
  - fail_all 只唤醒不锁存：之后的新调用照常 send 并等满超时。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

为什么 unregister 要比较身份：

时刻	线程 A（call k, 超时）	router 线程	线程 B（call k）	表中 k →
t1	register → wA			wA
t2	get 超时，准备进 finally	deliver(k, resA)：pop 出 wA，put		（空）
t3			register → wB	wB
t4	`unregister(k, wA)`：表里是 wB，**不删**			wB
t5		deliver(k, resB) → True	拿到 resB	（空）

如果 t4 不比较直接 `del waiters[k]`，t5 的 deliver 找不到 waiter，resB 被当成 late，B 白等到超时。（t2 里 A 的 waiter 收到了 resA 但 A 已经放弃，这条结果就悄悄丢弃——它不算 late，因为确实有 waiter 接住了。）

Level 1 问题的答案

1. send 可能在返回前（甚至在 send 内部）就触发 deliver。先 send：deliver 时表里没有 key → 计为 late；随后 register → 永远等不到。
2. 每个 waiter 只会收到一个结果（deliver 用 pop 保证只交付一次）。容量 1 足够，而且 put_nowait 一定成功，router 线程永远不会被一个慢消费者卡住。
3. 两者都可能：正常交付时 deliver 先 pop 了，finally 里的 unregister 发现表里不是自己（或没有）就什么都不做；超时时 deliver 没来，finally 负责删除。
4. 见上表。
5. 需要；锁内只做 dict 的 pop 和计数，锁外 put。持锁做任何可能阻塞的操作都会把其他线程的 register/deliver 堵住。
6. `queue.Empty`；对外转成内置的 `TimeoutError`（可以 `from None` 去掉无关的链式异常）。
7. 正在等的：fail_all 直接把 error 放进它们的 waiter，它们的 call 拿到后 raise。以后的：fatal 存在 router 里，register 检查到就立即 raise。

A-14 · EngineCore busy loop

Level 1 · I/O 示例

例 1：预先排队的消息 + 运行中到达的消息

```
# 一个玩具引擎：add(rid, k) 表示请求 rid 还要跑 k 步；step() 让每个请求 k-=1，
# 返回 [(rid, 剩余k), ...]，k 归零的请求被移除；utility("div", (a, b)) 返回 a / b。

# 线程启动前，input_queue 里已有：
Msg("ADD", ("a", 2)), Msg("ADD", ("b", 1)),
Msg("UTILITY", (7, "div", (1, 0))), Msg("ABORT", ["b"]),
Msg("UTILITY", (8, "div", (6, 3)))
# 启动 run_busy_loop 线程；200ms 后再 put Msg("ADD", ("c", 1))；再过 200ms request_shutdown()

引擎被调用的顺序：
  add a, add b, abort (b,), step 1, step 2, add c, step 3
emit 收到的序列：
  ("utility", 7, None, "ZeroDivisionError('division by zero')")
  ("utility", 8, 2.0, None)
  ("output", ("a", 1))
  ("output", ("a", 0))
  ("output", ("c", 0))
  ("shutdown",)
```

五条预排队消息在第一次 step 之前全部处理完，所以两条 utility 的结果排在所有 output 之前；b 在被 step 之前就被 abort，**一个输出都没有**。step 2 之后引擎空闲，loop 阻塞，直到 c 到达。

© 2026 kzhou003

例 2：step 抛异常是致命的（同样的预排队消息，但引擎第 2 次 step 抛出 `RuntimeError("NCCL timeout")`）

```
引擎被调用的顺序:  add a, add b, abort (b,), step 1, step 2
emit 收到的序列:
  ("utility", 7, None, "ZeroDivisionError('division by zero')")
  ("utility", 8, 2.0, None)
  ("output", ("a", 1))
  ("fatal", "RuntimeError('NCCL timeout')")
```

run_busy_loop 在发出 fatal 后直接返回：**没有** `("shutdown",)`，之后 put 进来的 ADD c 也永远不会被处理。对比 utility 7 的 ZeroDivisionError：只影响那一次调用，loop 继续。

例 3：空闲 + 从别的线程关停

```
loop = EngineCoreLoop(engine, emitted.append)
t = threading.Thread(target=loop.run_busy_loop); t.start()
time.sleep(1.0)            # 期间 engine.step 调用 0 次，has_work 只被调用 1~2 次，CPU ≈ 0
threading.Thread(target=loop.request_shutdown).start()
t.join(0.5)                # 立即返回
emitted                    # -> [("shutdown",)]
```

关键问题

1. 引擎没有工作时，`input_queue.get()`、`input_queue.get(timeout=0.01)`、`while q.empty(): time.sleep(0.001)` 三种写法，哪个满足“BLOCK, no polling / spinning”?
2. 阻塞 get() 拿到的那条消息，是“只是用来唤醒”还是必须处理？如果你把它丢掉，例 1 里会少处理哪条?
3. loop 阻塞在 `get()` 时，另一个线程怎样才能让它“立刻”醒来？只设置一个 flag 够不够?
4. 醒来并抽干队列之后，如果发现已经请求了 shutdown，这一步还要不要 step?
5. 为什么必须“先抽干全部消息，再 step”？如果每步只取一条消息，例 1 的 b 会怎样?
6. utility 的异常和 step 的异常，try/except 分别应该包在哪里？哪一个之后 loop 继续、哪一个之后 return?
7. fatal 退出后还要不要 emit `("shutdown",)`？正常 shutdown 时 emit 的最后一条是什么?

细节

- **payload 格式**：

kind	payload	动作
ADD	`(req_id, data)`	`engine.add(req_id, data)`
ABORT	`[req_ids]`	`engine.abort(req_ids)`
UTILITY	`(call_id, name, args)`	`engine.utility(name, args)` —— args 作为一个 tuple 整体传入，不是 *args
WAKEUP	无	什么都不做

- **emit 的元组形状要精确**：`("output", out)`（step 返回列表里的**每个**元素各 emit 一次）、`("utility", call_id, result, None)`、`("utility", call_id, None, repr(exc))`、`("shutdown",)`（注意是单元素元组）、`("fatal", repr(exc))`。
- **“drain all queued messages in FIFO order”**：用 `get_nowait()` 循环直到 `queue.Empty`。`queue.Queue` 本身就是 FIFO，只要你按取出顺序立即处理即可，不要先分类（比如先处理所有 ADD 再处理所有 ABORT），那会打乱 ADD→ABORT→ADD 的语义。
- **“may be called from any thread / a signal handler”**：request_shutdown 要尽量简单、不阻塞：设置一个线程安全的标志（`threading.Event`），再 `put_nowait` 一条 WAKEUP 把阻塞的 get() 叫醒。它不应该自己 emit，也不应该 join 线程。
- **“engine keeps running”**：utility 失败只是这次调用失败，结果通过 emit 回传给调用方（按 call_id 匹配），loop 不退出。

Level 2 · 思路与常见坑

- **状态**：只需要多一个 `self._shutdown = threading.Event()`（在 __init__ 里创建）。
- **拆一个 _handle(msg)**：按 kind 分派；UTILITY 分支内部自己 try/except 并 emit 结果或错误。这样“阻塞拿到的第一条”和“抽干的其余几条”走同一个函数。
- **循环结构**：`while not shutdown:` ① 若 `not engine.has_work()`：阻塞 get 一条并处理；② get_nowait 抽干并处理；③ 若 shutdown 已设置 → break；④ 若 has_work：step 并逐个 emit output。
- **为什么 ④ 还要再问一次 has_work**：①②处理的可能是 ABORT/UTILITY/WAKEUP，处理完引擎仍然没有工作，此时调用 step 没有意义（某些引擎会断言）。
- **异常边界**：把整个 while 包在 `try: ... except Exception as e: emit(("fatal", repr(e))); return` 里；正常退出循环后 emit `("shutdown",)`。utility 的异常在 _handle 里就被吃掉，不会冒到外层。
- **常见 bug 与症状**：

© 2026 kzhou003

bug	症状
`get(timeout=...)` 轮询	空闲时 has_work 被调用几十上百次/秒（自旋）
只设 flag 不放 WAKEUP	空闲时 request_shutdown 后 loop 永远卡在 get()（或要等 timeout）
阻塞 get 拿到的消息被丢弃	第一条 ADD 丢失，请求永远没有输出
每步只处理一条消息	ABORT 晚于 step 生效，被 abort 的请求仍产出输出
step 异常没捕获	线程静默死亡，外部只看到"没输出了"，没有 fatal
shutdown 后仍 step 一次	关停时多出 output，或在已经开始拆除的资源上运行

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

1. 只有不带 timeout 的 `get()`。另外两种都是轮询。
2. 必须处理——它就是队列里最早的那条。丢掉的话例 1 里 ADD a 就没了。
3. 往队列里放一条消息（WAKEUP）。只设 flag 不够，阻塞中的 get() 看不到 flag。
4. 不要 step，直接 break 然后 emit shutdown。
5. 只取一条的话：第一步只处理 ADD a 就 step，b 还没加进来；再往后 ABORT 可能晚于 b 的某次 step，导致 b 产生输出（或者顺序与客户端发送的顺序不一致）。
6. utility：包在单次调用周围，emit 错误后继续。step：包在整个 loop 外面，emit fatal 后 return。
7. 不要。正常 shutdown 的最后一条是 `("shutdown",)`。

追踪：例 1 的 loop 每一轮（自己填 ??）

轮	① has_work	阻塞 get 拿到	② 抽干处理的消息	③ shutdown?	④ step / emit
1	False	ADD a	ADD b, UTILITY 7, ABORT [b], UTILITY 8	否	has_work=True → step 1 → output (a,1)
2	True	（不阻塞）	（空）	否	step 2 → output (a,0)
3	??	??（约 200ms 后）	（空）	否	??
4	False	??	（空）	??	break → emit ??

答案：第 3 轮 has_work=False，阻塞直到拿到 ADD c，step 3 → output (c,0)；第 4 轮 has_work=False，阻塞直到 request_shutdown 放进来的 WAKEUP，shutdown=是 → break → emit `("shutdown",)`。整个过程中 has_work 一共被调用 ?? 次（答案：①处 4 次 + ④处 3 次 = 7 次，按下面骨架的循环结构计算；你的结构不同次数可能不同，但必须是个位数）。

自检：①空闲 1 秒，用计数器包装 has_work，看调用次数是否只有个位数；②在另一个线程里调 request_shutdown，计时 join；③构造 ADD→ABORT→ADD 预排队，记录引擎调用顺序；④让 step 抛异常，确认最后一条是 fatal 且线程已退出。

A-15 · WAITING_FOR_CHUNK：停靠与归还

Level 1 · I/O 示例

例 1：一个普通请求 + 一个等 chunk 的请求（从 waiting 停靠，再从 running 停靠）

```
s = ChunkAwareScheduler(BaseScheduler(), src)      # src.is_ready 由我们手动控制
s.add(Req("x", 0))                                  # 普通请求
s.add(Req("t1", 1, needs_chunk=True))               # t1 的 chunk 还没到

s.schedule()  -> ["x"]
   parked == {"t1": (t1, "waiting", "WAITING")}, t1.status == "WAITING_FOR_CHUNK"
   load_async 调用记录: ["t1"]
s.schedule()  -> ["x"]                              # 仍然没就绪；load_async 记录仍是 ["t1"]
# t1 的 chunk 到了 (is_ready("t1") 变 True)
s.schedule()  -> ["x", "t1"]                        # parked == {}；x、t1 都在 running，status 都是 "RUNNING"
# t1 把这一块吃完了，下一块还没来 (is_ready 又变 False)
s.schedule()  -> ["x"]
   parked == {"t1": (t1, "running", "RUNNING")}     # 这次的来源是 running!
   load_async 调用记录: ["t1", "t1"]                # 新的一次停靠 => 新的一次 load_async
# 下一块到了
s.schedule()  -> ["x", "t1"]                        # t1 回到 running，status 恢复为 "RUNNING"
```


© 2026 kzhou003

注意 origin 和 previous status 是**每次停靠时**记录的：同一个请求第一次从 waiting 停靠，第二次从 running 停靠。

例 2：同时归还时按 arrival 排序，而不是按 add 的顺序

```
s.add(Req("k", 5, needs_chunk=True))
s.add(Req("m", 3, needs_chunk=True))
s.add(Req("n", 4, needs_chunk=True))
s.schedule()  -> []                 # 三个都停靠；load_async 记录 ["k", "m", "n"]（停靠时按队列顺序遍历）
# 三个 chunk 同时到达
s.schedule()  -> ["m", "n", "k"]    # arrival 3, 4, 5
```

例 3：上游 schedule 抛异常时 finally 仍然归还

```
class Exploding(BaseScheduler):
    def schedule(self):
        src.ready.add("z")          # z 的 chunk 恰好在上游调度期间到达
        raise RuntimeError("boom")

s = ChunkAwareScheduler(Exploding(), src)
s.add(Req("z", 0, needs_chunk=True))
s.add(Req("y", 1))
s.schedule()                        # RuntimeError("boom") 原样抛给调用者
# 之后：base.waiting == [y(WAITING), z(WAITING)]，parked == {}，load_async 记录 ["z"]
```

z 归还时被 append 到 waiting 的**末尾**，所以排在 y 之后；异常没有被吞掉。

关键问题

1. `BaseScheduler.schedule()` 一看到 `WAITING_FOR_CHUNK` 就断言失败。那么“停靠”只改 `status` 够不够？请求对象此时应该在哪个容器里？
2. 同一个请求连续 3 次 `schedule()` 都没就绪，`load_async` 应该被调用几次？它被归还、之后又没就绪时呢？你的代码里是什么机制天然地保证了“每次停靠一次”？
3. 一个 running 中的请求被停靠时，“previous status”是什么？如果归还时一律写成 `"WAITING"`，后果是什么（提示：上游只会修改 waiting 中请求的 status）？
4. `schedule()` 为什么要在 (a) 先归还，再在 (b) 停靠？如果去掉 (a)、只靠上一轮的 (d)，一个在两次 `schedule()` 之间就绪的请求会晚几轮被调度？
5. 在 (d) 中归还的请求，本轮 `base.schedule()` 的返回值里有它吗？它会在哪一轮被调度？
6. “按 arrival order 归还”里的 arrival 指的是什么：`Req.arrival` 字段、add 的先后，还是 `parked` 的插入顺序？
7. `abort(id)` 之后，`source.is_ready(id)` 仍可能返回 True。是什么保证它不会在下一次 (a)/(d) 被归还？它可能藏在哪几个地方？

细节

- **parked 的三元组**：`parked[id] = (req, origin, previous_status)`。`origin` 记录请求是从哪个队列拿出来的（用什么表示由你决定，例如字符串 `"waiting"` / `"running"`）；`previous_status` 是停靠**那一刻**的 `req.status`，停靠后 `req.status` 变成 `"WAITING_FOR_CHUNK"`。
- **BaseScheduler 的语义**：返回 `running 的 id + waiting 的 id`，然后把 waiting 全部挪进 running 并把它们设为 `"RUNNING"`。也就是说 running 中请求的 status 它从不改——恢复 status 的责任完全在你这一层。
- **只有 `needs_chunk=True` 的请求才会被停靠**，普通请求从不查询 `is_ready`。`is_ready` 是“下一块是否已到”，它可以 True → False → True 反复变化（每消费一块就可能重新变 False）。
- **归还的位置**：放回 origin 队列的**末尾**（waiting 是 `deque`，running 是 `list`）；“arrival order” 只约束**同一批**一起归还的请求之间的相对顺序，不要求插回它原来的位置。
- **finally 的含义**：(d) 必须在 `base.schedule()` 抛异常时也执行，但异常要继续向上抛。正常时 `schedule()` 返回的是 `base.schedule()` 的结果。
- **abort**：任何时候都能调用；请求可能在 parked、base.waiting 或 base.running 中。被 abort 的请求以后既不能出现在 `schedule()` 的结果中，也不能被归还；上游看到 `"ABORTED"` 状态同样会断言失败，所以它不能留在队列里。

Level 2 · 思路与常见坑

- **拆成两个私有 helper**：`_restore_ready()`（扫描 parked，把就绪的按 arrival 放回来源队列）和 `_park_not_ready()`（扫描两个队列，把 needs_chunk 且未就绪的移进 parked）。`schedule()` 就只剩四行骨架：restore → park → `try: return base.schedule()` → `finally: restore`。
- **load_async 去重**：被停靠的请求已经**不在** base 的队列里了，所以 (b) 下一轮根本看不到它，也就不会再停靠它一次。因此“在停靠的那一刻调用一次 `load_async`”就天然满足了“每次停靠一次”。如果你想更保险，可以额外维护一个“已请求加载”的集合，并在归还/abort 时清掉它——但要确保清理，否则第二次停靠就不会再调 load_async。
- **边遍历边删除**：在 `for r in self.base.waiting` 中 `remove(r)` 会跳过元素（list）或直接抛 `RuntimeError: deque mutated during iteration`。先 `list(...)` 拍一个快照再遍历。同理，遍历 `parked` 时不要直接 `del`，先收集就绪的再处理。
- **归还顺序**：先收集所有 `is_ready` 的 parked 条目，按 `req.arrival` 排序，再逐个 `del parked[id]`、恢复 status、append 到 origin 队列。如果按字典插入顺序归还，例 2 会得到 `["k", "m", "n"]`。

© 2026 kzhou003

- **常见 bug 1：漏了 (a)**。表现：请求就绪后的第一次 `schedule()` 没有调度它，要再等一轮（(d) 才把它放回来）。**常见 bug 2：漏了 (d) 或没放在 finally 里**。表现：上游异常后请求永远留在 parked（例 3 中 z 卡住）。
- **常见 bug 3：恢复 status 写死**。running 来源的请求被恢复成 `"WAITING"` 后，上游不会再改它（只改 waiting 中的），状态就一直错。**常见 bug 4：finally 里写 return**：会把异常吞掉。
- **abort 的三个藏身处**：parked、base.waiting、base.running 都要查并移除，状态可设为 `"ABORTED"`。只从 parked 里删，若它恰好在队列里（例如没就绪前就 abort 的普通请求）上游下一轮还会调度它；只设 ABORTED 不移除，上游直接断言失败。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

例 1 的逐步状态（括号里是 status；R=RUNNING，W=WAITING，WFC=WAITING_FOR_CHUNK）

调用	is_ready(t1)	(a) 之后	(b) 之后：waiting / running / parked	base.schedule() 返回	(d) 之后	loads
#1	F	无变化	[x(W)] / [] / {t1: (waiting, W)}，t1(WFC)	["x"]	running=[x(R)]，t1 仍停靠	[t1]
#2	F	无变化	[] / [x(R)] / {t1}（t1 不在队列，(b) 看不到它）	["x"]	无变化	[t1]
#3	T	waiting=[t1(W)], parked={}	不停靠（已就绪）	["x","t1"]	running=[x(R), t1(R)]	[t1]
#4	F	无变化	[] / [x(R)] / {t1: (running, R)}	["x"]	无变化	[t1, t1]
#5	T	running=[x(R), t1(R)]	不停靠	["x","t1"]	无变化	[t1, t1]

小练习：若 #2 和 #3 之间 chunk 是在**上游 schedule 执行过程中**到达的，t1 在第 ?? 次调用的 (??) 步被归还，在第 ?? 次调用中被调度。（答案：#2 的 (d) 归还到 waiting，#3 调度它。）

abort 的追踪：add a(0)、b(1, needs_chunk)、c(2)；b 没就绪。

操作	waiting	running	parked
schedule() → ["a","c"]	[]	[a, c]	{b}
abort("b")	[]	[a, c]	{}（b 从停靠表删除）
abort("c")	[]	[a]（c 从 running 删除）	{}
b 的 chunk 到了；schedule() → ["a"]	[]	[a]	{}——b 不在 parked，(a)/(d) 根本不会看到它

Level-1 问题的答案

- 只改 status 不够：上游遍历的是 waiting/running，见到 WAITING_FOR_CHUNK 就断言。必须从队列中**移出**，只存在于 parked。
- 连续 3 次未就绪：1 次。归还后再次停靠：再 1 次（共 2 次）。保证机制：停靠后请求不在队列，(b) 遍历不到它。
- running 中的请求 previous status 是 `"RUNNING"`；写死 "WAITING" 会让它永远带着错误状态待在 running。
- 没有 (a) 时，在两次调用之间就绪的请求要等下一次的 (d) 才回队列，晚 1 轮。
- (d) 归还的请求不在本轮返回值里（上游已经跑完），下一轮被调度。
- arrival 指 `Req.arrival` 字段。
- abort 从 parked 删除即可阻止归还（归还只扫描 parked）；它还可能在 base.waiting 或 base.running 里，三处都要删。

骨架（自己补全细节） 注意 `_park_not_ready` 里"先记录 `r.status` 再改成 WAITING_FOR_CHUNK"的顺序；反过来写，previous status 就永远是 WAITING_FOR_CHUNK。

A-16 · Scheduler：continuous batching + chunked prefill + 抢占

Level 1 · I/O 示例

例 1：token budget + chunked prefill（`num_blocks=100, block_size=4, max_num_seqs=8, max_num_batched_tokens=8`；加入 a1(prompt 6, max_tokens 3)、a2(prompt 5, max_tokens 2)、a3(prompt 3, max_tokens 1)；每步 schedule 后立刻 update）

step	schedule() 返回	update 后 (num_computed / num_tokens / num_blocks)	blocks_used	本步结束的
1	`{a1: 6, a2: 2}`	a1 6/7/2；a2 2/5/1（prefill 没完，不采样）；a3 仍在 waiting	3	-
2	`{a1: 1, a2: 3, a3: 3}`	a1 7/8/2；a2 5/6/2（这一步才采样第一个 token）；a3 采样 1 个即达 max_tokens	4	a3

© 2026 kzhou003

3	`{a1: 1, a2: 1}`	a1 输出 3 个、a2 输出 2 个，均达到 max_tokens	0	a1, a2

step 1：a1 先拿 6，剩 budget 2 给 a2（只做了 prompt 的一部分）。step 2：running 先排：a1 decode 1，a2 剩余 3；budget 还剩 4，a3 整段 3 个 token 进入。注意 block 数按"已算 + 本步要算"的 token 数算：step 1 后 a2 只持有 2 个 token → 1 块。

例 2：抢占别人（`num_blocks=3, block_size=2, max_num_seqs=8, max_num_batched_tokens=64`；a1(prompt 2, max_tokens 3)、a2(prompt 3, max_tokens 2)；假设模型给 aN 的第 k 个输出是 `100*N + k`）

step	schedule() 返回	发生了什么	waiting 之后	num_preemptions
1	`{a1: 2, a2: 3}`	a1 1 块、a2 2 块，blocks_used=3（满了）	`[]`	0
2	`{a1: 1}`	a1 要持有 3 个 token → 需要第 2 块，放不下 → 抢占 running 最后一个 = a2（释放 2 块，num_computed=0，放回 waiting 队首）→ 重试 a1 成功。阶段 2：a2 需要 ceil(4/2)=2 块，2+2>3，停	`[a2]`	1
3	`{a1: 1}`	a1 第 3 个输出，结束并释放 → blocks_used=0	`[a2]`	1
4	`{a2: 4}`	a2 recompute：prompt 3 + 已生成 1 = 4 个 token 一次算完，采样出第 2 个输出，结束	`[]`	1

最终 a1.output = `[100, 101, 102]`，a2.output = `[200, 201]` ——与没有抢占时完全一样（输出被保留，只是 KV 重算）。

例 3：FCFS 队首阻塞（`num_blocks=5, block_size=4, max_num_batched_tokens=100`；依次加入 a1(prompt 9)、a2(prompt 9)、a3(prompt 1)）

```
s.schedule()       # -> {"a1": 9}
s.blocks_used      # -> 3
[q.seq_id for q in s.waiting]   # -> ["a2", "a3"]
```

a2 需要 3 块但只剩 2 块 → 停。a3 只要 1 块、放得下，但也**不许插队**。

关键问题

1. 一个正在 decode 的序列（刚采样了一个 token）在下一步 schedule 时，`num_tokens − num_computed` 等于多少？一个 chunked prefill 做了一半的序列呢？
2. 判断"block 放不放得下"时，要按多少个 token 算 block 数：`num_tokens`，还是 `num_computed + n`？两者在例 1 的 step 1 里对 a2 分别给出几块？
3. running 中的序列 X 放不下时，被抢占的是谁？抢占之后是继续处理 X（重试）还是跳过 X？如果被抢占的恰好是 X 自己呢——此时遍历下标要不要前进？
4. 被抢占的序列：哪些字段要重置、哪些必须保留？它重新进入时要算多少个 token？它会被放到 waiting 的哪一端，为什么？
5. 阶段 2 遇到第一个放不下的 waiting 序列：是 `continue` 还是 `break`？`max_num_seqs` 限制的是阶段 1、阶段 2，还是两者？
6. update 里，什么情况下一个被调度的序列**不**调用 `sample_fn`？序列结束时，除了改 status，还要做哪两件事？
7. update 没有参数告诉你"上一步调度了谁、各几个 token"。你打算把这份信息存在哪？

细节

- **剩余 token**：`num_tokens = prompt_len + len(output)`。剩余未算 = `num_tokens − num_computed`。decode 时它就是 1（刚采样的那个 token 还没算 KV），所以"decode 和 prefill 用同一个公式"——这就是 vLLM v1 调度器统一的地方。
- **block 记账**：每个 Seq 自带 `num_blocks` 计数器；本步需要 `ceil((num_computed + n) / block_size)` 块，只需要增量 `need − seq.num_blocks`（≤0 就不用动）。不变量：`blocks_used == sum(seq.num_blocks for running)` 且 `<= num_blocks`。释放 = 从 blocks_used 减掉并把 seq.num_blocks 归零。
- **budget**：两个阶段共用一个 `budget = max_num_batched_tokens`，每调度一个就减 n。budget 用完后，后面的 running 序列本步就不被调度（不是抢占，只是这步不跑）。
- **"put it at the FRONT of waiting"**：被抢占的是最晚进来的，但它仍比 waiting 里所有序列都早，所以放队首以保持 FCFS。它在同一步的阶段 2 里也会被尝试（例 2 的 step 2 就尝试了，但放不下）。
- **"if the victim is the sequence itself, skip it"**：X 已经不在 running 里了，本步不调度它；由于 running 缩短了一个，遍历下标不要前进（或者在列表末尾自然结束）。
- **update 的结束条件**：采样出的 token 先追加到 output，再判断 `tok == eos_token or len(output) >= max_tokens`；所以 eos 本身也在 output 里。结束时 status=FINISHED、移出 running、释放 block、放进 finished。

Level 2 · 思路与常见坑

- **先写三个 helper**：`_blocks_for(n_tokens)`（向上取整）、`_try_grow(seq, n_tokens) -> bool`（算增量，放得下就同时更新 blocks_used 和 seq.num_blocks，否则什么都不改并返回 False）、`_preempt(seq)`（从 running 移除、释放 block、num_computed=0、status=WAITING、waiting.appendleft、计数+1）。schedule 的逻辑会变得非常短。

© 2026 kzhou003

- **阶段 1 用下标 while 循环，不要用 for**：抢占会修改 `self.running`（删掉末尾元素）。用 `for seq in self.running` 一边迭代一边删会跳元素或访问到已被抢占的序列。
- **阶段 1 的内层"抢占-重试"循环**：`while not _try_grow(seq, ...)`：抢占 running[-1]；若它就是 seq 则跳出并标记"本序列被跳过"。只有 _try_grow 成功才写入 scheduled、扣 budget、下标+1。
- **阶段 2**：`while waiting and budget > 0 and len(running) < max_num_seqs`：看队首，n = min(剩余, budget)，_try_grow 失败就 break；成功则 popleft、status=RUNNING、append 到 running 末尾、记录 n、扣 budget。
- **记住本步调度结果**：把 schedule 返回的 dict 存到 `self._scheduled`（返回时给一个副本，防止调用方修改）。update 遍历它，结束后清空。
- **update 中按 id 找序列**：先建 `{seq_id: seq}`（来自 running），因为你会在遍历中把结束的序列从 running 删除。
- **常见 bug 与症状**：

bug	症状
按 num_tokens 而不是 num_computed+n 算 block	chunked prefill 时提前占满 block，例 1 中 blocks_used 偏大，可能出现不必要的抢占
被抢占后没把 num_computed 归零	重新进入时只调度 1 个 token：KV 已随 block 释放丢失却没有重算（例 2 的 step 4 会变成 {a2: 1} 而不是 {a2: 4}）
抢占时把 output 也清空了	最终 output 与无抢占时不同（重复生成第一个 token）
放回 waiting 尾部	被抢占的序列饿死，或在压力下顺序乱掉
prefill 未完成也调用 sample_fn	output 多出 token，max_tokens 提前达到
阶段 2 用 continue 跳过放不下的	违反 FCFS：例 3 中 a3 会被调度

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

1. decode：1；chunked prefill：prompt 剩下没算的部分（加上 0 个输出）。
2. 按 `num_computed + n`。例 1 step 1 的 a2：按 num_tokens=5 是 2 块，按 0+2=2 是 1 块；正确是 1 块。
3. running 的最后一个。抢占别人后重试 X；X 自己被抢占则跳过，下标不前进（running 已经变短）。
4. 重置：num_computed=0、num_blocks=0（并从 blocks_used 减掉）、status=WAITING；保留：output。重新算 prompt_len + len(output) 个 token。放队首。
5. break。max_num_seqs 只在阶段 2 作为准入条件（阶段 1 的都已经在 running 里了）。
6. num_computed 加完后仍 < num_tokens（prefill 块没做完）时不采样。结束时：从 running 移除并释放 block、放进 finished（并返回）。
7. 存在实例字段里，例如 `self._scheduled`。

追踪：抢占自己（`num_blocks=3, block_size=2, budget=64`；a1、a2 都是 prompt 2、max_tokens 3；先自己填 ??，答案在下方）

step	阶段 1	阶段 2	返回	blocks_used（update 后）	a2.output
1	running 为空	a1 n=2 → 1 块；a2 n=2 → 1 块	`{a1:2, a2:2}`	2	`[200]`
2	a1 n=1，需 2 块，+1 → 3 ✓；a2 n=1，需 2 块，+1 → 4 ✗ → 抢占 running[-1] = ?? → ??	a2 在队首：n=??，需 ?? 块 → ??	??	2	??
3	a1 n=1，不需新块	a2 仍放不下	`{a1:1}`	0（a1 结束）	`[200]`
4	running 为空	a2 n=??	??	2	??
5	a2 decode	-	`{a2:1}`	0	`[200, 201, 202]`

答案：step 2 victim 是 a2 自己 → 跳过；阶段 2 中 a2 n=3（2+1），需 2 块，2+2>3 → break；返回 `{a1: 1}`；a2.output 仍是 `[200]`。step 4：n=3，返回 `{a2: 3}`，output 变为 `[200, 201]`。num_preemptions = 1。

自检：把例 1、例 2、例 3 和上面的追踪各写成几行脚本跑一遍；每步断言 `blocks_used <= num_blocks` 且 `blocks_used == sum(s.num_blocks for s in running)`；所有序列结束后 blocks_used 必须回到 0。

© 2026 kzhou003

A-17 · BlockManager：paged KV + 前缀缓存

Level 1 · I/O 示例

例 1：不开前缀缓存，slot 与 append 跨块（`block_size=4, num_blocks=6`，空闲表按 FIFO 取块）

```
bm = BlockManager(num_blocks=6, block_size=4)
bm.allocate("x", [11, 12, 13, 14, 15, 16])   # -> [0, 1]      (6 个 token -> ceil(6/4)=2 块)
bm.num_free_blocks                            # -> 4           空闲表 [2, 3, 4, 5]
bm.slot("x", 5)                               # -> 5           = table[5//4]*4 + 5%4 = 1*4 + 1
bm.append_slot("x", 17)                       # -> 6           pos=6，块 1 还有空位，不分配
bm.append_slot("x", 18)                       # -> 7           pos=7，块 1 填满
bm.append_slot("x", 19)                       # -> 8           pos=8，需要第 3 块 -> table [0, 1, 2]
bm.num_free_blocks                            # -> 3
bm.free("x")                                  # 逆序归还 2, 1, 0 -> 空闲表 [3, 4, 5, 2, 1, 0]
bm.allocate("y", [1, 2, 3])                   # -> [3]
```

要点：append 只有在“当前 token 落在一个还不存在的逻辑块上”时才拿新块；free 逆序把块还到空闲表尾部。

例 2：前缀共享 + 引用计数 + 可驱逐池（`block_size=2, num_blocks=8, enable_prefix_cache=True`）

```
bm.allocate("a", [1, 2, 3, 4, 5])   # -> [0, 1, 2]   块0=[1,2] 块1=[3,4] 是满块，登记哈希；块2=[5] 不满
bm.num_cached_tokens("a")           # -> 0
bm.allocate("b", [1, 2, 3, 4, 9])   # -> [0, 1, 3]   可匹配 (5-1)//2 = 2 块，两块都命中
bm.num_cached_tokens("b")           # -> 4
bm.num_free_blocks                  # -> 4           ref = [2, 2, 1, 1, 0, 0, 0, 0]
bm.free("a")                        # 块2 ref->0 无哈希 -> 空闲表；块1、块0 ref->1
bm.num_free_blocks                  # -> 5
bm.free("b")                        # 逆序：块3 -> 空闲表；块1、块0 ref->0 有哈希 -> 可驱逐池 [1, 0]
bm.num_free_blocks                  # -> 8           空闲 6 + 可驱逐 2
bm.allocate("c", [1, 2, 3, 4])      # -> [0, 4]      只有 (4-1)//2 = 1 块可匹配!
bm.num_cached_tokens("c")           # -> 2           块0 从可驱逐池“复活”；可驱逐池剩 [1]
```

注意 c：第二块 `[3,4]` 明明在缓存里，但因为“至少留 1 个 token 要算”的规则不允许命中，于是给它分配了新块 4。

例 3：命中可驱逐块也会占用名额——原子性陷阱（`block_size=2, num_blocks=3, enable_prefix_cache=True`）

```
bm.allocate("a", [1, 2, 3, 4, 5]); bm.free("a")
# 空闲表 [2]，可驱逐池 [1, 0]，num_free_blocks = 3
bm.allocate("b", [1, 2, 3, 4, 5, 6, 7])   # 需要 4 块；命中块0、块1（都在可驱逐池里）
# -> raise OutOfBlocks
bm.num_free_blocks                        # -> 3，空闲表 [2]，可驱逐池 [1, 0]（完全没变）
bm.allocate("b", [1, 2, 3, 4, 5, 6])     # -> [0, 1, 2]，num_cached_tokens = 4，num_free_blocks -> 0
```

第一次 allocate：需要新块 4-2=2 个，但“可用”的 3 块里有 2 块正是被命中的那两块，真正能拿来当新块的只有 1 块，所以必须失败，而且失败前不能动任何状态。

关键问题

1. prompt 有 10 个 token、`block_size=4`：需要几块？其中哪几块是“满块”、会得到哈希？最后那个半满的块会不会进前缀缓存?
2. prompt 恰好 8 个 token、`block_size=4`，两块内容都已经在缓存里。允许命中几块？`num_cached_tokens` 是多少？如果全部命中，模型这一步还能产出什么?
3. 两个块内容都是 `[7, 8]`，但一个前面是 `[1, 2]`、一个前面是 `[5, 6]`。它们的哈希应该相同吗？哈希链里“上一个哈希”是为了防什么?
4. 命中的块可能 ref>0（别的序列正在用），也可能 ref==0（躺在可驱逐池里）。这两种情况在 allocate 时分别要做什么？哪一种会让“可用块数”变少?
5. 从可驱逐池里拿一个块去装**新内容**时，除了把它移出池子，还必须清理哪两处记录？不清理会出现什么诡异现象?
6. `append_slot` 的新 token 位置 `pos` 是多少（用你保存的哪个量表示）？在什么条件下要拿新块？拿块失败时，哪些状态已经被你改掉了?
7. `free` 时为什么要**逆序**遍历 block table？结合“LRU 先驱逐最旧的”想一想：逆序后谁最先被驱逐，这对后续命中率有什么好处?

细节

- **哈希链的起点**：docstring 写的是 `h_i = hash((h_{i-1}, tuple(block_i_tokens)))`。第 0 块没有前驱，你需要选一个固定的哨兵（例如 `None`）当 `h_{-1}`。allocate 和 append_slot 里算哈希必须用同一套规则，否则 append 填满的块永远不会被 allocate 命中。
- **“只在前 (len(tokens)-1)//block_size 块里匹配”**：这是“可匹配块数”的上限，而且是**从头连续**匹配——第一次 miss 就停（前缀缓存命中的必须是前缀）。`num_cached_tokens = 命中块数 * block_size`，永远是 block_size 的整数倍。
- **num_free_blocks = 空闲 + 可驱逐**：可驱逐块 ref==0，随时能被回收，所以算“可用”。`can_allocate(num_tokens)` 只是 `ceil(num_tokens/bs) <= num_free_blocks` 的粗检查（不考虑缓存命中）；allocate 内部必须做更精确的检查（见例

© 2026 kzhou003

3)。

- **slot**：`slot(seq, pos) = table[pos // bs] * bs + pos % bs`。它是物理 KV cache 展平后的下标，P09 的 `write_kv` 就用它。
- **enable_prefix_cache=False**：不登记任何哈希，于是 free 后的块全部回到空闲表，可驱逐池永远为空；num_cached_tokens 恒为 0。
- **同内容多副本**：例 2 里 c 的第二块（块 4）内容与块 1 相同、哈希相同。哈希表里一个哈希只指向一个物理块；登记时若已存在，保留旧的即可。驱逐某块时，只有当哈希表里该哈希**正指向这一块**才删除映射，否则会把别人的有效映射删掉。

Level 2 · 思路与常见坑

- **状态设计**（建议）：

结构	用途
`free: deque[int]`	从未缓存/无哈希的空闲块，popleft 取、append 还
`ref: list[int]`	每个物理块的引用计数
`block_hash: list[int|None]`	每个物理块当前登记的哈希（None = 不在缓存里）
`hash_to_block: dict[int,int]`	命中查找
`evictable: OrderedDict[int,None]`	ref==0 但仍有哈希的块，插入顺序 = LRU 顺序
`tables / tokens / hashes / cached: dict[seq_id, ...]`	每个序列的 block table、全部 token、哈希链、命中 token 数

- **allocate = 先算、再查、最后改**。①按满块算出整条哈希链；②在可匹配范围内从头查 hash_to_block，遇到 miss 停；③检查容量；④真正修改（命中块 ref+1、必要时移出可驱逐池；剩下的块逐个取新块并为满块登记哈希）。任何 raise 都只能发生在 ④ 之前。
- **容量检查公式**：`新块数 = 总块数 − 命中数`；`真正可用 = num_free_blocks − 命中块中 ref==0 的个数`。如果你只写 `新块数 <= num_free_blocks`，例 3 会"通过检查"，然后在取第 2 个新块时才 raise——此时命中块的 ref 已经加过了，状态被破坏。
- **抽一个取块 helper**（例如 `_take_block()`）：空闲表非空就 popleft；否则从可驱逐池弹出**最旧**的，删除它的哈希登记（hash_to_block 与 block_hash 两处），否则 raise OutOfBlocks；最后把 ref 设为 1。allocate 和 append_slot 都用它。
- **append_slot 的顺序**：`pos = len(tokens[seq])`；若 `pos // bs == len(table)` 先取新块（这一步可能 raise，所以放在**任何修改之前**）；再把 token 追加进 tokens；若 `(pos + 1) % bs == 0` 说明这块刚满，用哈希链最后一个哈希当父节点算出新哈希、追加进链并登记。最后返回 `slot(seq, pos)`。
- **free**：逆序遍历，ref-=1；归零时看 block_hash：有 → 放进可驱逐池（放到"最新"一端），无 → 还到空闲表。别忘了删掉该序列的 table/tokens/hashes/cached 记录。
- **常见 bug 与症状**：

bug	症状
命中 ref==0 的块时没把它移出可驱逐池	之后另一个 allocate 把这个"正在使用"的块驱逐并覆盖 → 两个序列共享一个块却内容不同；num_free_blocks 偏大
驱逐时没删哈希登记	驱逐后的块装了新内容，旧前缀仍能"命中"它（见 level 3 的驱逐追踪）
对半满的最后一块也算哈希	新的长 prompt 会命中只装了部分 token 的块
可匹配上限用 len//bs	prompt 刚好是 bs 的整数倍且全命中时 num_cached_tokens == len，模型没有 token 可算
append 先 append token 再取块	OutOfBlocks 后 tokens 多了一个，下一次 slot 计算全错

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

1. 10 个 token → 3 块；满块是第 0、1 块（token 0..7），各有哈希；第 2 块只有 2 个 token，不进缓存。
2. 只能命中 (8-1)//4 = 1 块，num_cached_tokens = 4。全命中的话没有任何 token 需要 forward，拿不到最后位置的 logits，就采样不出下一个 token。
3. 不同。链式哈希把前缀编码进去，防止"块内容相同但前缀不同"的块被误命中（KV 依赖整个前缀，不只是本块 token）。
4. ref>0：只需 ref+1。ref==0：先从可驱逐池删掉再 ref=1；它原本被算在 num_free_blocks 里，命中后就不再可用，所以会让"可用于新块的数量"减 1。
5. block_hash[b] 置 None，以及 hash_to_block 中指向 b 的映射。不清理：之后同前缀的请求会"命中"一块已经被别的内容覆盖的块，KV 错乱。
6. pos = 当前 token 数（新 token 的下标）。条件 pos // bs == len(table)。拿块是第一步，失败时什么都还没改。
7. 逆序时最后一块最先进入可驱逐池，成为"最旧"的，因此最先被驱逐；越靠前的前缀块（更可能被多个请求共享）越晚被驱逐。

© 2026 kzhou003

追踪：append 填满的块进入缓存，随后被驱逐（`block_size=2, num_blocks=3, enable_prefix_cache=True`）

操作	返回	空闲表	可驱逐池(旧→新)	说明
`allocate("a",[7,8,9])`	`[0, 1]`	`[2]`	`[]`	块0=[7,8] 满，登记 h0；块1=[9] 不满
`append_slot("a",10)`	`3`	`[2]`	`[]`	pos=3 → 块1 offset 1；块1=[9,10] 刚满，用父哈希 h0 算 h1 并登记
`free("a")`		`[2]`	`[1, 0]`	逆序：块1 先入池（最旧）
`allocate("b",[7,8,9,10,11])`	`[0, 1, 2]`	`[]`	`[]`	可匹配 2 块，全中，num_cached_tokens=4
`free("b")`		`[2]`	`[1, 0]`	块2=[11] 无哈希 → 空闲表
`allocate("z",[50..55])`	`[2, 1, 0]`	`[]`	`[]`	先用空闲块 2，再驱逐 1、0，并删除 h1、h0 的登记
`free("z"); allocate("b2", [7,8,9,10,11])`	??			num_cached_tokens = ??（提示：h0 还在吗？）

最后一行答案：返回 3 个新块、num_cached_tokens = 0。

自检清单：例 1~3 逐行跑一遍；再自己构造"同一块被两个序列共享、先 free 一个、再 free 另一个"的场景，确认第一次 free 后 num_free_blocks 只增加了非共享的块。

A-18 · 批不变的采样器（per-request seed）

Level 1 · I/O 示例

例 1：repetition penalty

```
logits = torch.tensor([3.0, -1.0, 0.5, -0.2])
apply_repetition_penalty(logits, prev_tokens=[1, 3, 3], penalty=1.5)
# -> tensor([ 3.0, -1.5, 0.5, -0.3])
logits                    # 仍是 [3.0, -1.0, 0.5, -0.2]（不能原地改）

s = Sampler()
s.sample(torch.tensor([[3.0, -1.0, 2.5, -0.2]]), ["r"],
         [SamplingParams(temperature=0.0, repetition_penalty=1.5)], [[0]])
# -> tensor([2])          token 0: 3.0/1.5 = 2.0 < 2.5, greedy 改选 token 2
```

token 3 在历史里出现两次，但只惩罚一次（"every distinct previous token"）；从没出现过的 token 0、2 不变。

例 2：top_k / top_p 保留集合（T=1，一行 4 个 token，softmax 后概率为 `[0.5, 0.25, 0.15, 0.1]`）

参数	降序累计概率	保留的 token	过滤后重新归一化的概率
`top_p=0.4`	0.5 | 0.75 | 0.9 | 1.0	`{0}`	`[1, 0, 0, 0]`
`top_p=0.7`	同上	`{0, 1}`	`[0.667, 0.333, 0, 0]`
`top_p=0.8`	同上	`{0, 1, 2}`	-
`top_p=1e-9`	同上	`{0}`	（至少保留 1 个）
`top_k=2`	-	`{0, 1}`	`[0.667, 0.333, 0, 0]`
`top_k=3, top_p=0.6`	top_k 后重新归一化：0.556 | 0.833 | 1.0	`{0, 1}`	-

top_p=0.7：只保留 token 0 时质量 0.5 < 0.7，不够；加上 token 1 达到 0.75 ≥ 0.7 → 停。注意最后一行：top_p 是在 top_k **之后**的分布上算的。温度对分布的影响：T=0.5 → `[0.725, 0.181, 0.065, 0.029]`，T=2 → `[0.370, 0.262, 0.203, 0.166]`。

例 3：批不变性（一行 logits `row = [1.0, 0.5, 0.0, -0.5, 0.2, 0.9]`，请求 q 用 `SamplingParams(seed=7)`，每次 sample 抽一个）

© 2026 kzhou003

```
单独跑 6 步（batch 只有 q）               -> [0, 5, 4, 0, 4, 5]
混在别的请求里跑 6 步（q 位置/同伴每步都变，
  同伴有的带 seed、有的不带）               -> [0, 5, 4, 0, 4, 5]   # 必须完全一致
错误做法：全 batch 共用一个 seed=7 的 generator -> [5, 4, 1, 0, 0, 0]
free("q") 后重新以 seed=7 跑 3 步           -> [0, 5, 4]           # 随机流从头开始
```

具体数字取决于抽样方式（这里是 softmax 后 `torch.multinomial(probs, 1, generator=g)`）；你的实现数字不同不要紧，**“单独 == 混合”与“free 后重放一致”**才是要满足的性质。

关键问题

1. `prev_tokens=[4, 4, 4]`、penalty=2，token 4 的正 logit 2.0 惩罚后是 1.0 还是 0.25?
2. 为什么负 logit 要**乘**而不是除? 对 -2.0 除以 2 会让这个 token 更可能还是更不可能?
3. temperature=0 时如果先做 `logits / temperature` 会发生什么? repetition penalty 对 greedy 行还生效吗?
4. top_p 的“smallest prefix whose mass reaches top_p”：越过阈值的那个 token 留不留? 用“它之前的累计质量”怎么表达这个判断?
5. `top_k=-1`、`top_k >= V` 应该怎么处理? top_k 中出现并列第 k 大的值时，你的实现会保留几个?
6. 每个带 seed 的请求的 generator 什么时候创建、存在哪里、跨 step 复用吗? 不带 seed 的请求用什么?
7. 为什么不能对整个 `[B, V]` 的概率矩阵调用一次 `torch.multinomial(probs, 1, generator=g)`，即使 g 是按 seed 建的?

细节

- **“do not mutate the input”**：先 `clone()` 再改。sample 里 logits 是调用方的张量，原地修改会让调用方（例如再算 logprobs）看到被惩罚过的值。
- **处理顺序是固定的**：penalty → (greedy?) → /T → top_k → top_p → 抽样。top_k、top_p 只是把不保留的位置设为 `-inf`，softmax 后它们的概率恰好为 0，其余自动重新归一化。
- **top_p 的“至少保留 1 个”**：即使 top_p 极小（如 1e-9），概率最大的 token 也必须保留，于是退化成 greedy。实现上通常是在排序后的掩码里强制第 0 位不删。
- **“created on first use from params.seed”**：第一次见到某个 req_id 且 seed 不为 None 时，`torch.Generator().manual_seed(seed)` 并存进字典；之后每一步都用同一个对象，让随机流连续前进。`free(req_id)` 把它从字典删掉（不存在时不报错）。
- **dtype / 返回值**：logits 可能是 bf16/fp16，建议逐行转 `float()` 再处理；返回 `[B]` 的 `torch.long`，第 i 个元素对应第 i 行。

Level 2 · 思路与常见坑

- **逐行处理**：批不变性最简单的保证方式是 `for i in range(B)`，每行只用自己的 logits、params、prev_tokens、generator。先写对，向量化是 follow-up。
- **apply_repetition_penalty**：去重得到下标张量 → 取出这些位置的值 → `torch.where(v > 0, v / p, v * p)` → 写回 clone。penalty==1 或 prev 为空可以直接返回（不改值，也算“不修改输入”）。
- **top_k**：`kth = torch.topk(row, k).values[-1]`，把 `row < kth` 设为 -inf（并列时会多留几个，可以接受）。仅当 `0 < k < V` 时启用。
- **top_p**：降序排序 → softmax → cumsum；“之前的累计质量” = `cum - probs`；它已经 ≥ / > top_p 的位置删掉；强制保留第 0 位；用排序下标映射回原位置置 -inf。
- **generator 管理**：`dict[req_id, torch.Generator]`；seed 为 None 的请求用一个共享的默认 generator（或全局 RNG）即可——它们本来就不要求可复现。
- **抽一个 token**：`torch.multinomial(softmax(row), 1, generator=g)`。每行一次调用，所以请求 q 的 generator 只被 q 自己消费，消费次数与 batch 组成无关。
- **常见 bug 与症状**：

bug	症状
重复 token 惩罚多次	例 1 中 token 3 变成 -0.45 而不是 -0.3
负 logit 也做除法	被惩罚的负分 token 反而更容易被选中
top_p 删掉越过阈值的那个 token	top_p=0.7 时只剩 {0}；极小 top_p 时可能全部被删 → softmax 得 NaN
所有行共用一个 generator	例 3 的“混合”结果与“单独”不一致
每步都 manual_seed 新建 generator	同一请求每一步都抽到同一个随机数，分布明显偏
free 后没删 generator	新请求复用旧 id 时接着旧随机流，无法复现

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

1. 1.0：distinct，只除一次。

© 2026 kzhou003

2. -2.0/2 = -1.0 更大 → 更可能被选，与"惩罚"相反；乘以 2 得 -4.0 才是降低。
3. 除以 0 得 inf/NaN。阈值判断必须在除法之前；penalty 在 greedy 之前，所以对 greedy 行仍生效（例 1 第二段）。
4. 留。判断式：token 之前（不含它自己）的累计质量已经达到 top_p 的才删；第 0 个的"之前质量"是 0，天然保留。
5. 都视为不启用。用"≥ 第 k 大值"保留时，并列会多保留，可以接受。
6. 第一次 sample 到该 req_id 时创建，存在 Sampler 实例的字典里，跨 step 复用，free 时删除。无 seed 的用共享 generator。
7. 一次调用会按行顺序依次消费同一个 generator，请求 q 拿到的随机数取决于它前面有几行；而且一个 g 无法同时代表多个不同 seed 的请求。

追踪：top_k=3, top_p=0.6, T=1（原概率 `[0.5, 0.25, 0.15, 0.1]`）

阶段	token0	token1	token2	token3
logits = log(p)	-0.693	-1.386	-1.897	-2.303
top_k=3 后	-0.693	-1.386	-1.897	-inf
降序 softmax	0.556	0.278	0.167	0
之前的累计质量 cum - p	0	0.556	0.833	1.0
> 0.6 → 删?	保留（强制）	??	??	删
最终概率	??	??	0	0

答案：token1 保留（0.556 未达 0.6），token2 删；最终概率 `[0.667, 0.333, 0, 0]`。

自检：①例 2 的每一行用你的过滤函数算出保留集合；②例 3：同一个 seed 的请求单独跑 N 步、再随机打乱 batch 跑 N 步，结果列表必须相等；③构造概率 0.8/0.2 的两个 token，抽几千次看频率是否接近。

A-19 · Paged KV 写入 + decode attention（参考实现）

Level 1 · I/O 示例

例 1：write_kv 与 padding（cache 形状 `[3, 2, 1, 1]`，即 3 块、block_size=2、1 个 kv head、head_dim=1，初始全 0）

```
k = [[[10.]], [[20.]], [[30.]], [[40.]]]          # [4, 1, 1]
v = k * 0.1
slot_mapping = tensor([4, 5, -1, 1])
write_kv(k_cache, v_cache, k, v, slot_mapping)

k_cache[:, :, 0, 0]  # -> [[ 0., 40.],     块0: offset1 <- token3 (slot 1)
                     #     [ 0.,  0.],     块1: 没人写
                     #     [10., 20.]]     块2: slot 4,5 <- token0, token1
v_cache[:, :, 0, 0]  # -> [[0., 4.], [0., 0.], [1., 2.]]
```

token2 的 slot 是 -1（padding），30 不能出现在 cache 的任何位置——特别是不能出现在最后一个 slot（块2 offset1）。

例 2：paged decode attention（`block_size=2, num_blocks=4, num_kv_heads=1, num_heads=2, head_dim=2, scale=1.0`）

```
# 序列0：3 个 token，物理块 [2, 0]（不连续、也不按顺序）
#   K0 = [[1,0], [0,1], [1,1]]   V0 = [[1,2], [3,4], [5,6]]   写入 slot [4, 5, 0]
# 序列1：1 个 token，物理块 [3]
#   K1 = [[2,2]]                 V1 = [[7,8]]                 写入 slot [6]
block_tables = [[2, 0],
                [3, 77]]          # 77 是垃圾值（甚至超出 num_blocks）
context_lens = [3, 1]
q = [[[1,0], [0,2]],              # 序列0 的 head0 / head1
     [[1,1], [0,0]]]              # 序列1

paged_decode_attention(q, k_cache, v_cache, block_tables, context_lens, 1.0)
# -> [[[3.0, 4.0], [3.8099, 4.8099]],
#     [[7.0, 8.0], [7.0,    8.0   ]]]
```

序列0 head0：scores = $q \cdot K^T$ = [1, 0, 1] → softmax = [0.4223, 0.1554, 0.4223] → 加权 V = [3.0, 4.0]。head1：scores = [0, 2, 2] → [0.0634, 0.4683, 0.4683] → [3.81, 4.81]。序列1 只有 1 个 token，无论 q 是什么输出都是 V1；如果去读 table 里的 77 会直接越界。

例 3：GQA 的 head 映射（`num_heads=8, num_kv_heads=2`，group = 4）

query head h	0	1	2	3	4	5	6	7
正确：h // group	0	0	0	0	1	1	1	1

© 2026 kzhou003

错误：h % num_kv_heads	0	1	0	1	0	1	0	1

把 K 从 `[n, KVH, D]` 扩到 `[n, H, D]` 时，哪种 PyTorch 重复操作得到第一行、哪种得到第二行?

关键问题

1. slot 7、block_size 4：对应哪个物理块、哪个 offset? 反过来，cache 用 `view(-1, KVH, D)` 展平后，slot 和第一维下标是什么关系?
2. 如果直接 `k_cache.view(-1, KVH, D)[slot_mapping] = k`，slot -1 会写到哪里? 会报错吗?
3. context_len = 5、block_size = 4：block table 的前几项有效? 取出这些块后一共有几个 token 位置，要截掉几个?
4. block_tables 的 dtype 是什么? 它能直接用作高级索引吗?
5. GQA 下 query head h 对应 kv head `h // (H // KVH)`：`repeat_interleave` 和 `repeat` 哪个与之匹配?
6. "Softmax in fp32"：哪些张量需要转 fp32? 输出最终是什么 dtype?
7. scale 乘在哪一步? 乘在 softmax 之后会怎样?

细节

- **cache 布局** `[num_blocks, block_size, num_kv_heads, head_dim]`：前两维合起来就是"slot"维。`slot = block * block_size + offset`，所以 `cache[slot // bs, slot % bs]` 与把前两维展平后的 `cache_flat[slot]` 指的是同一个位置。
- **"slot -1 marks a padding token that must NOT be written"**：CUDA graph 会把 batch 填充到固定大小，填充 token 的 slot 是 -1。Python/PyTorch 的负索引会把 -1 解释成"最后一个"，所以必须先用掩码把它们过滤掉，而不是指望索引越界报错。
- **"entries past the last used block are garbage"**：block_tables 是 padding 成 `[B, max_blocks]` 的矩形，序列 b 只有前 `ceil(context_lens[b] / block_size)` 项有意义；后面的值可能是任意数，甚至超出 num_blocks。
- **最后一块不满**：取出 nb 个块拼起来有 nb*block_size 个位置，只有前 context_len 个是这个序列的 token，其余是别的序列或旧数据，必须截掉（不能只靠 mask 为 0 的 cache 值，因为 0 向量的 score 是 0，不是 -inf）。
- **one query token per sequence**：decode 阶段每个序列只有最新的 1 个 query，它可以看见全部 context（包括它自己，已经先 write_kv 写进去了），所以不需要 causal mask。

Level 2 · 思路与常见坑

- **write_kv（向量化）**：`valid = slot_mapping >= 0` → 只取有效的 slot 与对应的 k/v 行 → 拆成 `blocks = slots // bs, offs = slots % bs` → `k_cache[blocks, offs] = k[valid]`（v 同理）。原地写入，函数返回 None。
- **decode（先写循环版）**：对每个 b：取 n = context_lens[b]，nb = ceil(n / bs)，`table = block_tables[b, :nb].long()`。
- **gather + reshape + 截断**：`k_cache[table]` 形状 `[nb, bs, KVH, D]` → reshape 成 `[nb*bs, KVH, D]` → 取前 n 行。块的物理顺序由 table 给出，逻辑顺序自动正确。
- **GQA**：`k.repeat_interleave(H // KVH, dim=1)` 把 `[n, KVH, D]` 扩成 `[n, H, D]`，此时 head h 的 K 来自 kv head h // group。也可以把 q reshape 成 `[KVH, group, D]` 不复制 K。
- **注意力**：`scores[h, j] = (q[b, h] · k[j, h]) * scale`（fp32），softmax 在 j 维；`out[b, h] = Σ_j probs[h, j] * v[j, h]`；最后 `.to(q.dtype)` 写入输出张量 `[B, H, D]`。einsum `"hd,nhd->hn"` / `"hn,nhd->hd"` 很方便。
- **常见 bug 与症状**：

bug	症状
没过滤 slot -1	最后一个物理块的最后一个 slot 被 padding token 覆盖，长序列结果偶发错误
读了 table 的整行	IndexError（垃圾值越界），或者读到别人的 KV、结果与 dense 不一致
没截掉最后一块的多余位置	上下文长度不是 bs 整数倍的序列结果错（例 2 的序列0 会多看到 1 个位置）
GQA 用 repeat 而不是 repeat_interleave	num_kv_heads=1 时正确、>1 时错——很隐蔽
block_tables 是 int32 直接索引	新版 PyTorch 在 CPU 上能用，但旧版本 / 某些后端不支持；转 long 最稳
softmax 在 bf16 中做	与 fp32 参考有 1e-2 量级偏差

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

1. 块 1、offset 3；展平后第一维下标就是 slot。
2. 写到展平后的最后一个位置（最后一块的最后一个 offset），不报错——这正是危险之处。
3. 前 2 项；2*4=8 个位置，截掉后 3 个，只留前 5 个。
4. 通常是 int32（vLLM 里为了省显存/喂给 kernel）；新版 PyTorch 也接受 int32 索引，但转成 `long` 再做高级索引最稳妥。
5. `repeat_interleave`（[a,b] → [a,a,a,a,b,b,b,b]）；`repeat` 得到 [a,b,a,b,...]，对应 h % KVH。
6. q、k、v 都可以转 fp32 来算 scores / softmax / 加权和；输出转回 q.dtype。
7. softmax 之前乘在 scores 上；乘在之后就只是把输出整体缩放，注意力分布不对。

© 2026 kzhou003

追踪：例 2 序列0 的 gather（自己填 ??）

步骤	值
n = context_lens[0]	3
nb = ceil(n / bs)	??
table = block_tables[0, :nb]	??
k_cache[table] 形状	[??, 2, 1, 2]
reshape 后（逻辑位置 0..3）	[[1,0], [0,1], [1,1], ??]
截断到前 n 个	[[1,0], [0,1], [1,1]]
head0: scores = q·K^T × 1.0	[1, 0, 1]
softmax	[0.4223, 0.1554, 0.4223]
out = probs @ V	[??, ??]

答案：nb = 2；table = [2, 0]；形状 [2, 2, 1, 2]；第 4 个位置是块 0 offset 1 的内容，这里是 0 向量 [0, 0]（未写入），它不属于该序列——如果不截断，它的 score 是 0 而不是 -inf，会被分到注意力；out = [3.0, 4.0]。

自检：自己写一个 dense 版本（直接用连续的 K/V 张量算注意力），随机生成若干序列长度（包含 1、恰好 bs 的倍数、非倍数），打乱物理块分配、把 block table 尾部填成大垃圾值，与 paged 版本 `torch.testing.assert_close` 对拍。

A-20 · SharedMemoryConnector：跨进程 put/get 协议

Level 1 · I/O 示例

例 1：往返 + 只能消费一次

```
c = ShmConnector(namespace="hv")
k = "req7_0_0"
c.put(k, b"abc")
os.path.exists(c.lock_path(k))   -> True        # put 结束后锁文件还在（它表示“有人写过、还没被读”）
c.get(k)                         -> b"abc"
c.get(k)                         -> None        # 读者已经 unlink 了段并删了锁文件
os.path.exists(c.lock_path(k))   -> False
c.get("never_written")           -> None        # 锁文件不存在

c.shm_name(k) == c.shm_name(k)   -> True        # 确定性：另一个进程用同样的 key 也能算出同样的名字
len(c.shm_name(k))               -> 例如 23     # namespace + 短哈希；必须远小于 31
c.put(k, b"one"); c.put(k, b"two!")             # 同一个 key 写两次（旧段未被读）
c.get(k)                         -> b"two!"     # put 发现残留段 → 替换
```

例 2：锁被别人持有 = “还在写”

```
c.put("job_0_3", b"xyz")
fd = os.open(c.lock_path("job_0_3"), os.O_RDWR)
fcntl.flock(fd, fcntl.LOCK_EX)       # 模拟一个仍在写的 writer
c.get("job_0_3")   -> None           # 不阻塞，立即返回
fcntl.flock(fd, fcntl.LOCK_UN); os.close(fd)
c.get("job_0_3")   -> b"xyz"
```

即使“别人”是同一个进程里另一次 `os.open` 得到的 fd，flock 也会冲突——flock 锁属于打开的文件描述（open file description），不属于进程。

例 3：cleanup_prefix 只清理“我写过、还没人读、前缀匹配”的 key

```
for i in range(3): c.put(f"rA_0_{i}", bytes([i]))
c.put("rAx_0_0", b"q")
c.get("rA_0_1")             -> b"\x01"     # 被消费了一个
c.cleanup_prefix("rA_")     -> 2           # rA_0_0、rA_0_2；"rAx_0_0" 不以 "rA_" 开头
c.cleanup_prefix("rA_")     -> 0           # 已经清过了
c.get("rA_0_0")             -> None
c.get("rAx_0_0")            -> b"q"
```

写者先退出的场景：另一个 Python 进程 put 一个 key 后退出，1 秒后本进程 get。如果写者没把段从自己的 resource tracker 注销，结果是 `None`，而且写者退出时会打印 `UserWarning: resource_tracker: There appear to be 1 leaked`

© 2026 kzhou003

`shared_memory objects to clean up at shutdown`；注销后得到原数据。

关键问题

1. 读者和写者之间没有任何元数据通道。读者只知道 key，它怎么知道去打开哪个 shm 段、哪个锁文件？这对 `shm_name` 提出了什么要求（确定性？长度？字符集？）
2. 为什么要写 8 字节长度头？如果只靠 `shm.size` 推断长度，例 1 里 3 字节的 payload 会读出多少字节？`put(k, b"")` 又会怎样？
3. `get` 返回 None 有三种原因。分别是哪三种？它们分别对应写者生命周期的哪个阶段？
4. put 里"先拿锁、再创建段"和"先创建段、再拿锁"有什么区别？哪种顺序会让读者读到一半写入的数据？
5. 谁负责 unlink 段？为什么不是写者？写者进程退出时，谁会"好心"替它 unlink？
6. `cleanup_prefix` 怎么判断一个自己写过的 key "已经被消费"了？（提示：读者在 get 成功后会删除什么？）
7. 读者为什么要在 `close()` 之前就把数据 `bytes(...)` 拷出来？

细节

- **POSIX shm 名字**：Python 的 `SharedMemory(name=...)` 在 POSIX 上会自动在名字前加 `/`，macOS 的上限是 31 字节（含这个 `/`）。像 `"req-3f2a9c1e_0_17"` 这样的 key 加上 namespace 很容易超长，所以通常用 `namespace + "_" +` 哈希的前 `N` 位 。必须用确定性哈希（如 `hashlib`），不能用 Python 的 `hash()` ——它在不同进程里有随机种子。
- **段大小会被向上取整**：在 macOS 上 `SharedMemory(create=True, size=5).size` 是 `16384`（一页）。读者看到的 `buf` 比写入的数据长，所以需要长度头。另外 `size=0` 会直接 `ValueError: 'size' must be a positive number different from zero` ——长度头顺便让空 payload 也有合法大小。
- **flock 语义**：`fcntl.flock(fd, LOCK_EX)` 阻塞等待；`LOCK_EX | LOCK_NB` 拿不到时抛 `BlockingIOError`。锁是建议性的（advisory），只对同样使用 flock 的参与者有效。写者用阻塞模式，读者用非阻塞模式（get 必须不阻塞）。
- **锁文件 = "存在一个未消费的 key"**：put 时 `O_CREAT` 创建它；读者读完后删除它。因此"锁文件不存在"既表示"从没写过"，也表示"已经被读了"。`lock_dir` 为 None 时选一个默认目录（Linux 上 `/dev/shm` 是内存文件系统，否则用临时目录）。
- **resource tracker（Python < 3.13）**：每个进程有一个 tracker 子进程；`SharedMemory` 创建（以及在 3.12 中附着）时都会向它登记名字，`unlink()` 会注销。进程退出时，tracker 把仍在登记中的段当作泄漏并 unlink。读者"附着 → unlink"是平衡的；写者"创建 → 不 unlink"就不平衡——所有权要显式地转交出去。
- **"残留段"**：同一个 key 之前写过但没被读（或者上一个进程崩溃留下的），`create=True` 会抛 `FileExistsError`。spec 要求替换它，而不是失败或追加。

Level 2 · 思路与常见坑

- **状态只有三样**：`lock_dir`、`namespace`、以及一个"我写过的 key"集合（dict/set 都行，给 cleanup_prefix 用）。`lock_path(key)` 直接由 `shm_name(key)` 派生，例如 `os.path.join(lock_dir, shm_name(key) + ".lock")`，这样两边都能独立算出来。
- **put 的顺序**：`os.open(lock, O_CREAT|O_RDWR)` → `flock(LOCK_EX)` → 创建段（`FileExistsError` 时附着旧段、close、unlink，再创建）→ 注销 tracker → 写长度头和数据 → `close()` → 记录 key → 在 `finally` 里 `LOCK_UN` 并关闭 fd。锁必须在创建段**之前**拿到：读者能看到段的时候，锁一定已经被持有，所以读者不会读到半成品。
- **get 的顺序**：`os.open(lock, O_RDWR)`（`FileNotFoundError` → None）→ `flock(LOCK_EX|LOCK_NB)`（`BlockingIOError` → None）→ 附着段（`FileNotFoundError` → None）→ 读长度、拷贝数据 → `close()` → `unlink()` → 删锁文件 → `finally` 里解锁并关 fd。**每一个 return None 的分支都要走到解锁/关 fd**，否则 fd 泄漏，锁也一直挂着。
- **"锁文件存在但段不存在"** 为什么可能：写者刚 `os.open` 创建了锁文件、还没来得及 flock，读者就抢到了锁。这时返回 None 是正确的（下次再来），不要抛异常，也不要删锁文件。
- **resource tracker 的修复**：写者创建段后调用 `resource_tracker.unregister(shm._name, "shared_memory")`（`_name` 是带 `/` 前缀的真实名字），把所有权交给读者；读者 `unlink()` 时库会自动注销它自己的登记。用 try/except 包住，以兼容不同 Python 版本。
- **BufferError 陷阱**：`data = shm.buf[8:8+n]` 得到的是 memoryview 切片，紧接着 `shm.close()` 会抛 `BufferError: cannot close exported pointers exist`。要先 `bytes(...)` 拷贝。写入同理：`shm.buf[8:8+n] = data` 是赋值，不保留视图，可以直接 close。
- **cleanup_prefix**：遍历自己记录的 key（先拍快照），前缀匹配的就从记录中删除；如果锁文件还在（= 没人读过），就附着 + unlink 段、删锁文件，计数 +1；段或锁文件已不存在时吞掉 `FileNotFoundError`。已被消费的 key 也要从记录中删除，但不计数。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

一个 key 的生命周期（L = 锁文件，S = shm 段，T = 写者 tracker 中的登记）

时刻	L	flock	S	T	此刻 get(k) 返回
put 前	无	-	无	-	None（锁文件不存在）
put: os.open 之后、flock 之前	有	无人持有	无	-	None（段不存在）
put: 持锁写入中	有	写者	有（未写完）	登记 → 已注销	None（BlockingIOError）

© 2026 kzhou003

put 返回后	有	无	有	无	数据
写者进程退出	有	无	有（因为 T 里没有它）	-	数据
get 成功后	无	-	无（读者 unlink）	-	None

如果写者**没有**注销：第 5 行的 S 会变成 ??（答案：无——写者的 tracker 在退出时把它 unlink 了，读者拿到 None）。

Level-1 问题的答案

- 两边各自用 `shm_name(key)` / `lock_path(key)` 推出名字：确定性（跨进程同结果，用 hashlib 不用 hash()）、短（≤ 30 左右）、只含安全字符（十六进制即可）。
- 只靠 `shm.size` 会读出整页（macOS 上 16384 字节）；空 payload 会因为 size=0 无法创建。
- None 的三种原因：锁文件不存在（未写/已读）；锁被持有（正在写）；段不存在（写者刚创建锁文件还没建段）。
- 先建段后拿锁：读者可能在两者之间拿到锁并读出还没写完的段。先锁后建段则不可能。
- 读者负责 unlink：只有读者知道"已经用完了"。写者退出时它的 resource tracker 会替它 unlink——这正是要阻止的。
- 读者成功 get 后会删锁文件，所以 cleanup 用 `os.path.exists(lock_path(key))` 判断。
- memoryview 切片持有 buffer 导出，close 会抛 BufferError；而且 close/unlink 之后内存可能已经无效。

骨架 思考：get 删锁文件时还持有 flock，这在"每个 key 只有一个消费者、key 不复用"的前提下是安全的；如果允许同一个 key 被反复写，会出现什么竞争?

A-21 · ChunkAdapter：后台收发线程与顺序/清理语义

Level 1 · I/O 示例

例 1：None payload 不占 chunk id；final 即使 payload 为 None 也要发

```
conn = InMemoryConnector()
snd = ChunkAdapter(conn, stage_id=2, src_stage_id=None, build_payload=lambda n: n if n % 3 else None)
rcv = ChunkAdapter(conn, stage_id=3, src_stage_id=2,    build_payload=lambda s: s)
snd.start(); rcv.start()
rcv.load_async("u9")
for n in range(1, 7):
    snd.save_async("u9", n, is_final=(n == 6))
# 稍等片刻后
rcv.poll_ready() -> {"u9": [{"data": 1, "finished": False},
                            {"data": 2, "finished": False},
                            {"data": 4, "finished": False},
                            {"data": 5, "finished": False},
                            {"data": None, "finished": True}]}   # n=6 的 payload 是 None，但它是 final
conn.puts        -> ["u9_2_0", "u9_2_1", "u9_2_2", "u9_2_3", "u9_2_4"]   # 6 次 save，5 个连续 id
```

注意 key 里的数字：发送端用**自己的** `stage_id`（2），接收端用 `src_stage_id`（2）去拼同一个 key。

例 2：快照隔离 + poll_ready 是"取走自上次以来的"

```
# 换一对新的 adapter：snd(stage_id=0)、rcv(src_stage_id=0)，build_payload = lambda s: list(s)
live = ["a"]
rcv.load_async("v")
snd.save_async("v", live, is_final=False)
live.append("b")                        # 调度线程继续改活对象
snd.save_async("v", live, is_final=True)
live.append("c")
# 稍等 → 收到的 data 依次是 ["a"]、["a", "b"]；conn.puts == ["v_0_0", "v_0_1"]

# 再换一对 adapter（build_payload = lambda s: s），rcv.load_async("q") 已调用：
snd.save_async("q", 1, False); snd.save_async("q", 2, False)   # 稍等
rcv.poll_ready() -> {"q": [{"data": 1, "finished": False}, {"data": 2, "finished": False}]}
rcv.poll_ready() -> {}                                          # 已经被取走了
snd.save_async("q", 3, True)                                    # 稍等
rcv.poll_ready() -> {"q": [{"data": 3, "finished": True}]}
```

另外：如果 chunk 在 `load_async` 之前就已经被 put 了，`load_async` 之后照样能收到——connector 会一直保存它直到被 get。

例 3：cancel_sender 吞掉后续非 final 块，但 final 块（data=None）一定送达

```
# 一对新的 adapter（stage_id=0 / src_stage_id=0）；rcv.load_async("w") 已调用
# build_payload 会阻塞在一个 gate 上，直到我们放行
snd.save_async("w", "p0", False)       # save 线程开始 build "p0"，卡在 gate 上
for p in ["p1", "p2", "p3"]:
    snd.save_async("w", p, False)      # 在队列里排着
snd.save_async("w", "end", True)
```


© 2026 kzhou003

```
snd.cancel_sender("w")
gate.set()                              # 放行
# 接收端最终得到:
[{"data": "p0", "finished": False},   # 取消前已经在 build 的那一块照常发出
 {"data": None, "finished": True}]    # p1..p3 被丢弃;final 照发,但 data=None
conn.puts -> ["w_0_0", "w_0_1"]        # id 依然连续
```

"p0 会不会被发出"取决于取消时它是否已经过了取消检查——两种都可以接受;但 id 必须连续、final 必须到。

关键问题

1. `save_async` 在调度线程上执行,必须立即返回。那它里面最多只能做哪两件事? `build_payload` 为什么不能在这里调用?
2. chunk_id 应该在什么时候 +1:入队时、build 之后、还是 put **成功**之后?为什么"入队时 +1"会在例 1 中产生空洞,而接收端遇到空洞会怎样?
3. `build_payload` 返回 None 时什么情况下要跳过、什么情况下仍然要 put?final 块被 cancel 后,`data` 应该是什么?
4. 接收端怎么知道下一个要 get 的 key?每个请求需要记住哪些状态?一个请求收到 `finished=True` 之后还要继续轮询它吗?
5. recv 线程在锁外调用 `connector.get`(可能很慢)。就在这段时间里,调度线程调用了 `cleanup_receiver(req)`,随后又对同一个 req id `load_async` 了一次。recv 线程回来时,怎样才能判断"这块数据属于已经作废的那次注册"?只检查 `req in registered` 够不够?
6. 有 10 个请求在被轮询、但一个都没有数据时,recv 线程该做什么?完全没有请求在被轮询时呢?两者的区别是什么?
7. `stop()` 要"迅速返回"。如果 save 线程正睡在 `Condition.wait()` 上,你怎么叫醒它?如果它恰好在你 notify 之前一瞬间才去 wait,会不会永远睡下去?

细节

- **两个后台线程、两个调用方**:`save_async` / `cancel_sender` / `load_async` / `poll_ready` / `cleanup_receiver` 都在调度线程上调用;SAVE 线程只做 build + put;RECV 线程只做 get + 提交。同一个 ChunkAdapter 对象同时拥有两半,但在一个 stage 里通常只用其中一半(stage 0 只发,最后一个 stage 只收)。
- **key 格式**:发送端 `f"{req}_{stage_id}_{chunk_id}"`,接收端 `f"{req}_{src_stage_id}_{next_id}"`。两边的 chunk_id 都从 0 开始,每个请求独立计数;接收端只能按 id 顺序逐个取,所以发送端的 id 必须**连续**。
- **chunk 字典**:put 的对象固定是 `{"data": payload, "finished": is_final}`,`poll_ready()` 原样返回这些字典,按 id 顺序放在 `{req: [...]}` 中;没有新内容的请求不出现在结果里。
- **"OUTSIDE any lock"**:`build_payload` 和 `connector.get` 都可能很慢。如果你在持有一把调度线程也要拿的锁时调用它们,调度线程就会被拖慢——这正是这层存在的意义。锁只用来保护小的共享字典/集合,持锁时间应该是微秒级。
- **per-registration token**:`load_async` 时为这个请求生成一个独一无二的对象(例如 `object()`)存起来;cleanup 时删掉;recv 线程在 get 之前记下当时的 token,get 回来后在锁内比较"现在登记的 token 是不是**同一个对象**"(用 `is`)。不是就丢弃这块数据。
- **Condition 与超时**:`Condition.wait(timeout=...)` 返回后必须重新检查条件(可能是超时、虚假唤醒或被 stop 叫醒)。给 wait 加一个较短超时(如 0.1 s),即使通知丢了也只是多睡一会儿,而不会永远卡住。

Level 2 · 思路与常见坑

- **发送端的数据结构**:一个保存 `(req, snapshot, is_final)` 的 FIFO 队列(`deque`,append/popleft 本身线程安全)+ 一个 Condition 用来唤醒 save 线程;`next_send_id: dict[req, int]`;`cancelled: set[req]`。由于只有一个 save 线程、按 FIFO 处理,同一请求的 chunk 天然有序。
- **save 线程一次迭代**:取出一项 → 查 cancelled(短暂持锁)→ 被取消且非 final:丢弃 → 被取消且 final:data=None → 否则在锁外 build → data 为 None 且非 final:跳过(不消耗 id)→ put 到当前 id 的 key → 成功后 id+1;如果是 final,清理这个请求的 id 计数和 cancelled 标记(为下一个同名请求复位)。put 抛异常时不要 +1。
- **接收端的数据结构**:`registered: dict[req, token]`、`next_recv_id: dict[req, int]`、`ready: dict[req, list]`(等待被 poll 的提交结果)、一个要轮询的请求队列 + Condition。`load_async` 对已注册的请求应该是幂等的(不要重复入队,也不要重置 id)。
- **recv 线程一轮**:把当前队列里的每个请求轮询一次:锁内读出 (token, next_id),没注册了就扔掉它;锁外 get;没数据就放回队尾;有数据就**在锁内**比对 token → 一致才 append 到 `ready[req]`、id+1;收到 finished 就注销它(不再放回队列),否则放回队尾。一轮结束后:队列为空 → 在 Condition 上睡(带超时);有请求但这一轮一个都没进展 → 睡 ~1 ms;有进展 → 立刻下一轮。
- **poll_ready 原子交换**:在锁内把 `ready` 整个换成新的空 dict 并返回旧的。不要在锁外遍历 `ready`(recv 线程可能正在往里 append)。
- **cleanup_receiver 是提交栅栏**:锁内删除 registered[req]、next_recv_id[req]、ready[req]。因为 recv 线程的"比对 token + 提交"也在同一把锁内,所以 cleanup 返回之后,任何一次提交要么已经发生(并且已被 cleanup 删掉了),要么会看到 token 不匹配而丢弃。**常见 bug**:只在 get 之前检查注册状态——get 期间被清理的块会在 cleanup 之后出现在 poll_ready 中。
- **stop()**:设置一个 `threading.Event`,对两个 Condition 分别 `with cond: cond.notify_all()`,再 join 线程(带超时)。线程循环条件检查这个 Event;在 wait 前也再检查一次 Event,以免错过唤醒。线程设为 daemon,避免测试结束时卡住解释器。
- **其他常见 bug 的表现**:没 deepcopy → 例 2 收到 `["a","b","c"]`;入队时就 +1 → 例 1 出现 `u9_2_2` 之后跳到 `u9_2_4`,接收端永远卡在等 `u9_2_3`;cancel 连 final 一起吞掉 → 下游永远收不到 finished;空闲时 busy-loop → CPU 100%,其他线程(含测试)变慢。

Level 3 · 详解(代码骨架见附录 C 的带注释参考解)

© 2026 kzhou003

例 1 的 save 线程逐步追踪（`build = n if n % 3 else None`，final 在 n=6）

出队项	build 结果	动作	put 的 key	之后 next_send_id[u9]
(u9, 1, F)	1	put	u9_2_0	1
(u9, 2, F)	2	put	u9_2_1	2
(u9, 3, F)	None	跳过	-	2
(u9, 4, F)	4	put	u9_2_2	3
(u9, 5, F)	5	put	u9_2_3	4
(u9, 6, T)	None	final → 仍然 put，data=None	u9_2_4	(删除，复位)

对应的 recv 线程：next_recv_id 依次 0→1→2→3→4→5；拿到 `u9_2_4` 的 `finished=True` 后，u9 被注销，不再放回轮询队列。

cleanup 竞态的时间线（connector 的 get 很慢）

时刻	调度线程	recv 线程	registered[q]
t0	load_async(q)		tok1
t1		锁内读到 tok1, id=0；锁外进入 get("q_0_0")…	tok1
t2	cleanup_receiver(q)	（还在 get 中）	无
t3	load_async(q)（同名新请求）	（还在 get 中）	tok2
t4		get 返回数据；锁内比较 registered[q] is tok1 → ??	tok2

答案：False → 丢弃。如果只检查 `q in registered`，t4 会是 True，旧请求的数据就混进了新请求。

Level-1 问题的答案

- save_async 只做：deepcopy 快照 + 入队（并 notify）。build_payload 可能很慢，会阻塞调度。
- put 成功之后才 +1。入队时 +1 会让被跳过的 None 和 put 失败的块各占一个 id，接收端按 id 顺序轮询，会永远等那个不存在的 key。
- None 且非 final → 跳过；final → 无论如何都 put；被 cancel 的 final → data=None。
- 每个请求：注册 token、下一个 id、已提交未取走的列表。收到 finished 后停止轮询并注销。
- 用 token 的身份比较（`is`），只查是否在 registered 中不够（例见上表）。
- 有请求但无数据：短暂退避（~1 ms）后再轮询；没有请求：在 Condition 上睡，直到 load_async 唤醒（带超时兜底）。
- stop 设 Event 并 notify_all；wait 带超时，所以即使错过通知也最多晚一个超时周期醒来，醒来后检查 Event 退出。

`start()` 创建两个 daemon 线程；`stop()` 设 Event → 两个 cv 都 `notify_all` → join(超时)。

A-22 · Thinker→Talker 输入构造（含“差一行”）

Level 1 · I/O 示例

例 1：stop 结束，多一行（D=2，hidden 第 r 行的值就是 r，方便看是哪一行）

```
prompt_ids  = [11, 12, 13, 14]          # P = 4
output_ids  = [21, 22, 151645]          # 151645 = <|im_end|>, 即 stop token
hidden      = 7 行, 第 r 行 = [r, r]
finish      = "stop"

→ prefill    第 0..3 行  (值 0,1,2,3)
→ decode     第 4..5 行  (值 4,5)       # 第 6 行属于 stop token, 丢掉
→ output_ids [21, 22]
```

target = 4 + 3 - 1 = 6，所以只看 hidden[:6]。

例 2：length 结束 + 捕获了多余的行

```
prompt_ids = [11, 12]  (P=2)   output_ids = [5, 6]   hidden 有 5 行 (0..4)   finish = "length"
→ prefill 行 0,1    decode 行 2,3    output_ids [5, 6]       # 第 4 行超出 target=4, 不要
prompt_ids = [1,2,3,4] (P=4)   hidden 只有 3 行  → ValueError (prefill 都没捕获全)
```

注意：多出来的行不一定只有“stop 那一行”，一律按 target 截断。

例 3：prefill 布局与 decode 消费（D=2）

© 2026 kzhou003

```
prompt_embeds = [[1,2],[3,4]]   first_reply = [0.5,0.5]
spk = [10,20]  codec_pad = [1,1]  codec_bos = [100,200]
mask_id=7, pad_id=5, start_id=6

ids    = [7, 7, 5, 6]
embeds = [[1,2], [3,4], [11,21], [100.5,200.5]]      # shape [P+2, D] = [4, 2]

decode: reply_queue = [[1,1],[2,2]], fallback = [-9,-9]
 step | last_codec_embed | 输出 input        | 剩余 queue 行数
 1    | [0.1,0.2]        | [1.1, 1.2]        | 1
 2    | [0.3,0.4]        | [2.3, 2.4]        | 0
 3    | [0.5,0.6]        | [-8.5, -8.4]      | 0   ← 用 fallback
 4    | [0.7,0.8]        | [-8.3, -8.2]      | 0   ← 一直 fallback
```

关键问题

1. 为什么 stop 结束的请求，hidden 行数会比“talker 真正需要的”多一行？这一行对应哪个 token 的 forward?
2. `finish_reason == "length"` 时最后一个生成 token 是“正常内容”还是“stop token”？要不要丢?
3. `prefill` 和 `decode` 的分界点由什么决定：`hidden` 的行数，还是 `len(prompt_ids)`？
4. 如果 `hidden` 行数 > target，多出来的行该怎么处理？如果 < P 呢？两种情况为什么处理方式不一样?
5. talker prefill 有 P+2 个位置。最后两个位置的 embedding 各自是哪两个向量相加？对应的 id 是什么?
6. `first_reply` 已经在 prefill 里用掉了。那 decode 时 `reply_queue` 的第一行应该是第几个生成 token 的行？（这是调用方的约定，但你得想清楚）
7. `talker_decode_input` 在 queue 为空时返回的“剩余 queue”是什么形状?

细节

- **“先追加再判停”**：vLLM 的调度循环是：采样 token → append 到 output → 检查 stop 条件。所以 stop token 也跑过一次（或至少被记录为一次）decode 输出，它对应的 hidden 行被一起累积了。talker 若把它当成回复向量消费，音频尾部会多出一段不属于回复的声音。
- **target 行数**：`target = P + len(output_ids) - drop`。它是“talker 应当看到的总行数”，而不是 `hidden.shape[0] - drop`——后者在捕获了多余行时会算错。
- **ThinkerPayload.output_ids** 是一个新的 list，去掉了 stop id；不要原地修改调用方传入的 list（调用方可能还要用它做 detokenize / 返回给用户）。
- **mask_id / pad_id / start_id** 是 talker（codec）词表里的特殊 id，在真实代码里来自 model config，这里由关键字参数传入。prompt 部分的 P 个位置在 codec 词表里没有“真实 token”，所以统一填 `mask_id`，模型只看 embedding。
- **[D] vs [1, D]**：`spk_embed + codec_pad_embed` 是一个 1-D 的 `[D]` 向量，要和 `[P, D]` 拼接，需要先变成 `[1, D]`（`[None]` / `unsqueeze(0)`），或者用 `torch.stack`。
- **decode 输入 = 文本侧 + 音频侧**：talker 每步的输入是“thinker 回复向量（文本语义）”与“上一步采样的 codec id 的 embedding（音频自回归）”之和。thinker 回复比 codec 帧少得多，用完以后文本侧换成 `fallback`（真实系统里是 tts_pad 的 embedding）。

Level 2 · 思路与常见坑

- **拆成三个纯函数，互不依赖**：trim 只做切片与边界检查；prefill 只做 cat；decode 只做“取头 + 相加”。没有状态、没有循环，每个函数 3–8 行。
- **trim 的顺序**：① 先算 P 并检查 `hidden.shape[0] < P` → ValueError；② 算 drop；③ 算 target 并截断 rows；④ 再按 P 切成 prefill / decode；⑤ output_ids 去尾。把检查放在最前面，错误信息才不会被后面的切片掩盖（切片越界不会报错，只会返回短 tensor）。
- **边界：stop 但 output_ids 为空**。这种情况下 drop=1 会让 target = P-1，把一行 prompt 切掉。想一想：没有任何生成 token 时还有“stop 那一行”可丢吗？一个稳妥的写法是只在 output_ids 非空时 drop。
- **常见 bug 1**：用 `hidden[:-1]` 来“去掉 stop 行”。当 hidden 捕获了多余行（例 2）时，你去掉的是那条多余行而不是 stop 行，stop 行仍留在 decode 里。
- **常见 bug 2**：decode 用 `rows[P:]` 而 rows 没有先截到 target——等价于 bug 1。表现：decode 行数比 output_ids 多 1，talker 末尾多一帧杂音。
- **prefill 拼接**：一次 `torch.cat` 三块（`[P,D]`、`[1,D]`、`[1,D]`），dim=0。ids 用 list 乘法再拼接两个特殊 id。返回顺序是 `(ids, embeds)`，别反了。
- **decode 的“剩余 queue”**：有行时返回 `queue[1:]`（形状 `[K-1, D]`，K=1 时是 `[0, D]`）；空时原样返回空 queue。不要返回 None，调用方会继续拿它调下一步。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

- stop token 被采样出来后先 append 再判停，它那一步的 hidden 已经被累积；这一行是“产生 stop token 的那次 forward 之后”记录的行，不属于回复内容。
- length 结束时最后一个 token 是普通内容（只是到了 max_tokens），全部保留，drop=0。
- 分界点是 `P = len(prompt_ids)`；hidden 行数只用于检查和截断。

© 2026 kzhou003

- > target：截断到 target，静默丢弃；< P：连 prompt 都不完整，talker 的 prefill 无法构造，必须报错。介于 P 和 target 之间：按切片自然得到较短的 decode（spec 不要求报错）。
- 倒数第二位：`spk_embed + codec_pad_embed`，id = `pad_id`；最后一位：`first_reply + codec_bos_embed`，id = `start_id`。
- queue 一般从第 2 个生成 token 开始（第 1 个已作为 first_reply 进 prefill）；这个函数本身不关心，只取 queue 头。
- 空 queue 时返回原来的空 queue，形状 `[0, D]`。

trim 的推导（例 1 逐步）

量	值
P	4
hidden.shape[0] < P ?	7 < 4 → 否，继续
drop	finish=="stop" 且 output 非空 → 1
target	4 + 3 − 1 = 6
rows = hidden[:target]	行 0..5
prefill = rows[:P]	行 0..3
decode = rows[P:]	行 4,5
output_ids	[21, 22, 151645][: 3−1] = [21, 22]

自检：用例 1/2/3 的数字跑一遍你的实现；特别检查 ① 例 2 的 5 行 hidden 下 decode 是不是恰好 2 行；② `build_talker_prefill` 返回的 embeds 前 P 行与 prompt_embeds 完全相等（没有被加任何东西）；③ decode 第 3 步起一直用 fallback，queue 形状保持 `[0, D]`。

A-23 · Codec 窗口规划：首块小、后块大、左上下文

Level 1 · I/O 示例

例 1：`initial=2, chunk=5, left_ctx=3` 下逐个调用 next_window

```
next_window(1,  0,  False, initial=2, chunk=5, left_ctx=3) -> None          # 首块要 2 帧，只有 1 帧
next_window(2,  0,  False, ...)                            -> (0, 0, 2)     # 首块
next_window(6,  2,  False, ...)                            -> None          # 之后每块 5 帧，只有 4 帧新帧
next_window(7,  2,  False, ...)                            -> (0, 2, 7)     # ctx_start = max(0, 2−3) = 0
next_window(12, 7,  False, ...)                            -> (4, 7, 12)    # 左上下文 3 帧：4,5,6
next_window(20, 12, True,  ...)                            -> (9, 12, 17)   # finished，但完整的一块还够 → 仍按 5
帧切
next_window(20, 17, True,  ...)                            -> (14, 17, 20)  # 剩 3 帧 → flush
next_window(20, 20, True,  ...)                            -> None          # 全部发完
next_window(1,  0,  True,  ...)                            -> (0, 0, 1)     # 很短的流：首块都凑不满，直接 flush
```

没有首块配置时：`next_window(6, 0, False, initial=0, chunk=6, left_ctx=10) -> (0, 0, 6)`，`next_window(12, 6, False, initial=0, chunk=6, left_ctx=10) -> (0, 6, 12)`（左上下文被 0 截断）。

例 2：slice_new_samples

```
window = (4, 7, 12), upsample = 2
decoded = vocoder(codes[4:12])        # 8 帧 × 2 = 16 个样本：s0 .. s15
                                      # s0..s5 属于上下文帧 4,5,6；s6..s15 属于新帧 7..11
slice_new_samples(decoded, (4, 7, 12), 2) -> [s6, s7, s8, s9, s10, s11, s12, s13, s14, s15]

slice_new_samples([0, 1, 2, 3], (0, 0, 2), 2) -> [0, 1, 2, 3]   # 没有左上下文时整段都是新的
```

例 3：一次完整的流（initial=2, chunk=5, left_ctx=3，总共 14 帧，分 6 批到达）

批次	本批帧数	available	finished	本批循环调用得到的窗口	之后 emitted
0	1	1	F	（无）	0
1	2	3	F	(0, 0, 2)	2
2	4	7	F	(0, 2, 7)	7
3	3	10	F	（无）	7
4	1	11	F	（无）	7

© 2026 kzhou003

5	3	14	T	(4, 7, 12), (9, 12, 14)	14

调用者每批都循环调用 next_window 直到返回 None，每拿到一个窗口就令 `emitted = new_end`。所有窗口的 [new_start, new_end) 恰好拼成 [0, 14)。

关键问题

1. “第一个窗口”怎么从参数里判断出来？函数没有任何内部状态，它能用的信息只有 `available / emitted / finished`。
2. `initial=0` 表示什么？此时首块用多大？
3. `finished=True` 但剩余帧数 ≥ 一个完整窗口时，应该一次把所有剩余帧都发出去，还是仍然按 chunk 大小切？（例 1 中 `(20, 12, True)`）
4. `finished=True` 且 `available == emitted` 时返回什么？如果你返回了 `(…, 20, 20)` 这样的空窗口，调用方的 while 循环会怎样？
5. `ctx_start` 为什么不能是负数？它和 `emitted` 之间的帧会被解码几次？被输出几次？
6. decoded 的第 i 个样本属于窗口里的第几帧？新帧 `new_start` 的第一个样本在 decoded 中的下标是多少？
7. left_ctx 要多大，流式解码的结果才和整段一次性解码**逐样本相同**？这依赖 vocoder 的什么性质？

细节

- **三个坐标都是“帧”下标，都在整条流的全局坐标系里**：`ctx_start ≤ new_start < new_end ≤ available`。`[ctx_start, new_start)` 是左上下文（只用来“热身”vocoder），`[new_start, new_end)` 才是这一步真正要发出去的新帧。
- **new_start 永远等于 emitted**：窗口必须首尾相接、无重叠无空洞。调用方在每个窗口之后把 emitted 推进到 new_end。
- **窗口大小**：第一个窗口（还什么都没发出，且 `initial > 0`）用 `initial`，之后都用 `chunk`。“full window available” 指 `available − emitted` 至少是这个大小；此时只取这么多，多出来的帧留给下一次调用。
- **flush**：只有在凑不满一个完整窗口、且 `finished`、且还有没发的帧时，才把剩下的全部作为最后一个（较短的）窗口。
- **slice_new_samples 的单位换算**：vocoder 每帧产出 `upsample` 个样本，decoded 是从 `ctx_start` 开始的局部坐标，长度为 `(new_end − ctx_start) * upsample`。你要做的就是把帧区间 `[new_start, new_end)` 换算成 decoded 里的样本区间。
- **为什么首块小**：TTFP（首包延迟）取决于第一个窗口要等多少帧；之后用大块提高 vocoder 的吞吐、减少调用次数。左上下文的代价是每个窗口多解码 left_ctx 帧。

Level 2 · 思路与常见坑

- **next_window 拆三步**：① 算本次窗口大小 `size`（首块判断：`emitted == 0 and initial > 0`）；② 决定 new_end：完整窗口 / flush / None；③ 由 emitted 和 left_ctx 算出 new_start、ctx_start。
- **判断顺序很重要**：先判断“完整窗口够不够”，再判断“finished 且有剩余”。如果把 finished 放前面，例 1 中 `(20, 12, True)` 会返回 `(9, 12, 20)`——虽然不会丢帧，但违反了“当完整窗口可用时返回完整窗口”。
- **flush 条件要求 `available > emitted`**：否则在流结束时会返回空窗口 `new_start == new_end`，调用方 `while w := next_window(...)` 会无限循环（emitted 不再前进）。
- **首块判断的陷阱**：用 `emitted == 0` 而不是 `available <= initial` 之类的条件。后者在帧一次性大量到达时（例如 available=30, emitted=0）会把首块误判成 chunk 大小。
- **slice 的公式**：偏移量 = `(new_start − ctx_start) * upsample`，结束 = `(new_end − ctx_start) * upsample`（也就是 len(decoded)）。常见 bug：直接用全局下标 `new_start * upsample` 去切局部的 decoded——只在 ctx_start=0 时碰巧正确，例 2 中会得到 `decoded[14:24]`，只剩 2 个样本。
- **可以加一个断言** `len(decoded) == (new_end − ctx_start) * upsample`，一旦调用方传错窗口就立刻暴露，而不是悄悄地切出错误长度的音频。
- **左上下文 ≥ 感受野**：如果 vocoder 是因果的，帧 i 的输出只依赖 `codes[i−r .. i]`，那么只要窗口里新帧左边至少有 r 帧，它看到的输入就与整段解码时完全一样；少于 r 帧（且 ctx_start 没有到 0）时，窗口开头几帧看到的上下文被截断，就会产生接缝。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

按例 1 的参数，逐步代入公式（填空）

available, emitted, finished	size	available - emitted	分支	new_end	ctx_start	结果
2, 0, F	2（首块）	2	完整	2	max(0, 0-3)=0	(0, 0, 2)
6, 2, F	5	4	不够且未结束	-	-	None
12, 7, F	5	5	完整	12	4	(4, 7, 12)
20, 12, T	5	8	完整（优先于 flush）	17	9	(9, 12, 17)
20, 17, T	??	??	??	??	??	(14, 17, 20)

© 2026 kzhou003

20, 20, T	5	0	没有剩余	-	-	None

（空行答案：size=5，差值 3，flush，new_end=20，ctx_start=14。）

左上下文 vs 感受野：一个能手算的例子 玩具 vocoder：帧 i 输出 2 个样本 `[10·S, 10·S+1]`，S = 窗口内 `codes[max(0,i−2) .. i]` 之和（感受野 r=2，只看窗口内部）。`codes = [3,1,4,1,5,9,2,6]`，`initial=3, chunk=3`，一次性全部可用（finished=True）。

left_ctx	窗口	本窗口 decoded	切出的新样本
2	(0,0,3)	30,31,40,41,80,81	30,31,40,41,80,81
2	(1,3,6)	10,11,50,51,60,61,100,101,150,151	60,61,100,101,150,151
2	(4,6,8)	50,51,140,141,160,161,170,171	160,161,170,171
1	(2,3,6)	40,41,50,51,100,101,150,151	**50,51**,100,101,150,151
1	(5,6,8)	90,91,110,111,170,171	**110,111**,170,171

整段解码：`30,31,40,41,80,81,60,61,100,101,150,151,160,161,170,171`。left_ctx=2 时完全一致；left_ctx=1 时每个窗口第一帧（加粗）出现接缝：帧 3 本该看到 codes[1..3]=1+4+1=6，却只看到 codes[2..3]=4+1=5。

Level-1 问题的答案

- 首块 ⇔ `emitted == 0`（还没发出过任何帧）且 `initial > 0`。
- `initial=0` 表示没有特殊首块，首块也用 chunk。
- 仍然按 chunk 切；剩下不足一块时才 flush。
- 返回 None；返回空窗口会让 emitted 停在原地，循环永不结束。
- 流开头时 new_start - left_ctx 可能为负，而帧下标没有负数。上下文帧会被解码不止一次（作为上一个窗口的新帧一次，作为下一个窗口的上下文再一次），但只输出一次。
- decoded[i] 属于局部帧 `i // upsample`，即全局帧 `ctx_start + i // upsample`；new_start 的第一个样本在 `(new_start − ctx_start) * upsample`。
- left_ctx ≥ vocoder 的（因果）感受野；依赖"因果 + 局部"这两个性质。

A-24 · MRoPE 三维位置

Level 1 · I/O 示例

例 1：图像在开头 + 矩形网格（spatial_merge=2）

```
segments = [("image", (2, 4, 6)), ("text", 2)]
图像 LLM 网格 = (t=2, h=4//2=2, w=6//2=3) → 12 个 token
            |---------- t=0 ----------|---------- t=1 ----------| text
t (轴0):     0  0  0  0  0  0          1  1  1  1  1  1         3  4
h (轴1):     0  0  0  1  1  1          0  0  0  1  1  1         3  4
w (轴2):     0  1  2  0  1  2          0  1  2  0  1  2         3  4
图像段最大位置 = 2（来自 w 轴）→ 下一段 start = 3
T = 14, max = 4, delta = (4+1) − 14 = −9
decode_position(14, −9) = [5, 5, 5]    decode_position(15, −9) = [6, 6, 6]
```

例 2：时间维很长的图像（视频帧）

```
segments = [("text", 1), ("image", (4, 2, 2)), ("text", 1)]
网格 (4, 1, 1) → 4 个 token
t:  0 | 1 2 3 4 | 5
h:  0 | 1 1 1 1 | 5
w:  0 | 1 1 1 1 | 5
T = 6, max = 5, delta = 6 − 6 = 0       # t 轴推进了 4，正好等于 token 数
```

delta 为 0 不代表"没有图像"，只代表这个图像占用的位置跨度恰好等于 token 数。

例 3：两张图紧挨着 + merge=1

```
[("text",1), ("image",(1,4,4)), ("image",(1,2,4)), ("text",1)]，merge=2
t: 0 | 1 1 1 1 | 3 3 | 5
h: 0 | 1 1 2 2 | 3 3 | 5
w: 0 | 1 2 1 2 | 3 4 | 5          T=8, max=5, delta=−2

[("image",(1,2,4))]，merge=1  → 网格 (1,2,4)，8 个 token
t: 0 0 0 0 0 0 0 0
```


© 2026 kzhou003

```
h: 0 0 0 0 1 1 1 1
w: 0 1 2 3 0 1 2 3                T=8, max=3, delta=-4
```

第二张图的 start = 3：第一张图最大位置是 2，不是"已有 token 数 5"。

关键问题

1. 一个 `(t, h, w)` 的图像段在 LLM 序列里占几个 token？`h` 不能被 `spatial_merge` 整除时会发生什么（docstring 用的是 `//`）？
2. "t-major，然后 h，然后 w"：第 k 个图像 token 的 `(t_idx, h_idx, w_idx)` 怎么由 k 算出来？
3. 每段结束后 `start` 更新成什么？为什么不是"目前为止的 token 数"？
4. 图像段的"最大位置"可能出现在哪个轴上？在例 1 里是哪个？
5. `delta` 什么时候为负、为 0、为正？（为正可能吗？）
6. decode 时第 i 个 token（i ≥ T）的位置为什么三轴相同？decode_position 返回什么形状和 dtype？
7. `("text", 0)` 这种空段该如何处理才不会影响后续 start？

细节

- **三轴**：positions 的第 0/1/2 行分别是 temporal / height / width。MRoPE 把 head_dim 的旋转频率分成三份，分别用这三行位置去旋转（M01）。纯文本时三行相同，退化为普通 1-D RoPE。
- **vision patch 与 LLM token**：视觉编码器按 patch 输出 `t×h×w` 个特征，然后把相邻 `merge×merge` 的 patch 合并成一个 LLM token。所以 h、w 是 patch 单位，LLM 网格是 `h//merge × w//merge`；t 方向不合并。
- **start 的语义**：start 是"下一段的位置起点"，等于**到目前为止所有位置（三个轴）的最大值 + 1**。图像的位置跨度是 `max(t, gh, gw)`，通常远小于 token 数 `t·gh·gw`，这就是 MRoPE 能"压缩"位置的原因。
- **delta**：`delta = (max_pos + 1) − T`。prefill 之后序列从绝对下标 T 继续，但位置应该从 `max_pos + 1` 继续，二者之差就是 delta，在请求生命周期内保存为常数。
- **dtype**：位置是整数下标，返回 `torch.long`，形状 `[3, T]`。`torch.arange` 默认就是 int64。
- **decode_position(index, delta)**：返回形状 `(3,)` 的 long tensor，三个元素都等于 `index + delta`。decode 只生成文本，所以三轴相同。

Level 2 · 思路与常见坑

- **整体结构**：维护 `start = 0` 和一个 `parts` 列表；逐段生成一个 `[3, n_seg]` 的块并 append；每段之后用这一块的最大值更新 start；最后 `torch.cat(parts, dim=1)`，再算 delta。
- **文本段**：`idx = start + arange(n)`，然后变成 3 行（`expand(3, n)` / `repeat` / `stack([idx]*3)`）。注意 `expand` 得到的是共享存储的 view，后面 `cat` 会拷贝，所以没问题；但如果你打算原地改它就要 clone。
- **图像段**：先算 `gh, gw = h // m, w // m`。三个下标可以用 `torch.meshgrid(arange(t), arange(gh), arange(gw), indexing="ij")` 然后各自 `flatten()` —— `ij` 索引 + 行主序 flatten 天然就是 t-major、再 h、再 w。或用 view/expand 广播。
- **start 更新**：`start = block.max() + 1`（转成 Python int）。对于文本块，max 就是 `start + n − 1`；对于图像块，max = `start + max(t, gh, gw) − 1`。空块（n=0 或网格为 0）没有 max，要跳过更新。
- **常见 bug 1**：`start += n_tokens`。纯文本时正确、有图像时错（例 3 第二张图会从 6 开始而不是 3）。模型不会报错，只会"变笨"。
- **常见 bug 2**：meshgrid 用了默认/`xy` 索引，或 flatten 顺序错，导致 h、w 互换。方形网格（2×2）上看不出来，矩形网格（例 1 的 2×3）一眼可见——自测时一定用矩形网格和 t>1。
- **常见 bug 3**：delta 用 `max_pos − T`（少了 +1）或用 token 数算 decode 位置。用例 1 检查：decode 的第一个位置应为 5。
- **空输入**：segments 为空时 `torch.cat([])` 会报错，返回 `torch.zeros(3, 0, dtype=torch.long)` 与 delta=0 更稳。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

- `t · (h//m) · (w//m)` 个；不能整除时向下取整（真实模型里 h、w 总是 merge 的倍数）。
- 令 `hw = gh·gw`：`t_idx = k // hw`，`h_idx = (k % hw) // gw`，`w_idx = k % gw`。
- `start` = 所有已生成位置的最大值 + 1。图像的位置跨度 = max(t, gh, gw) 而非 token 数，用 token 数会让后续文本位置"跳得太远"。
- 取决于 t、gh、gw 谁最大；例 1 中 gw=3 最大，所以最大位置 2 来自 w 轴。
- 有图像且 `t·gh·gw > max(t,gh,gw)` 时为负；只有文本，或图像退化成一条线（例 2：gh=gw=1）时为 0；不可能为正——每个 token 至少让位置跨度不超过它自己贡献的 token 数。
- decode 只产生文本 token，文本三轴相同；返回 `(3,)` long。
- 跳过，不更新 start（也不必加入 parts）。

逐段追踪（例 3 第一组）

段	start(进入)	网格/长度	块 (t / h / w)	块 max	start(离开)
text 1	0	1	0 / 0 / 0	0	1

© 2026 kzhou003

image (1,4,4)	1	1×2×2	1111 / 1122 / 1212	2	3
image (1,2,4)	3	1×1×2	33 / 33 / 34	4	5
text 1	5	1	5 / 5 / 5	5	6

T = 1+4+2+1 = 8，max = 5，delta = 6 − 8 = −2；下一个 decode token（index 8）位置 = 6。
自检：用例 1（矩形 + 多帧）检查 h/w 没有互换；用例 3 检查第二张图从 3 开始；检查 `pos.dtype == torch.long`、`pos.shape == (3, T)`；对任意输入，`decode_position(T, delta)` 的值应等于 `pos.max() + 1`。

A-25 · 启动规划：设备分组、并行初始化、回滚、准入

Level 1 · I/O 示例
例 1：init_groups（含传递）

```
{0: {2}, 1: {0}, 2: {0, 1}, 3: {1}, 4: {5}}
  stage1 与 stage2 共享 dev0；stage2 与 stage3 共享 dev1 → 1,2,3 一组
→ [[0], [1, 2, 3], [4]]

{3: {7}, 1: {7}, 0: {9}, 2: {9, 4}}          # dict 的 key 顺序是乱的
→ [[0, 2], [1, 3]]                            # 组内升序，组按首元素排序

{0: {0}, 1: {1}, 2: {2}, 3: {2, 0}}           # stage3 把 0 和 2 “桥接”起来
→ [[0, 2, 3], [1]]
```

例 2：run_init 失败短路（`max_workers=1`，让顺序确定）

```
stage_devices = {0: {0}, 1: {0}, 2: {1}}     → groups [[0, 1], [2]]
init_fn: stage1 抛 RuntimeError("boom")，其它返回 "e<sid>"

时间线：
  组[0,1]: init(0) → "e0" ✓ ; init(1) → 抛异常，记为第一次失败
  组[2]  : 开始前发现已有失败 → 不调用 init(2)
结束后：cleanup_fn(0, "e0")；raise InitError(stage_id=1)
init_fn 被调用的 stage: [0, 1]      cleanup 的: [(0, "e0")]

若改成 stage2 失败：调用 [0, 1, 2]，cleanup (0,"e0"),(1,"e1")，InitError.stage_id = 2
全部成功：{0: {0}, 1: {0}, 2: {1}}，init_fn = lambda s: s*10 → {0: 0, 1: 10, 2: 20}
```

例 3：admission_check（GiB = 2^{30}，两张 80 GiB 卡）

```
reserve = 2 GiB
  {0: ({0}, 0.9), 1: ({1}, 0.6), 2: ({1}, 0.3)}
  dev0: 72+2 = 74 ≤ 80 ✓    dev1: (48+2)+(24+2) = 76 ≤ 80 ✓        → []
  把 stage2 改成 0.35：dev1 = 50 + 30 = 80 ≤ 80 ✓（恰好相等也允许）    → []
  再把 reserve 改成 4 GiB：dev1 = 52 + 32 = 84 > 80                  → 1 条，含 "device 1"

跨卡 stage（TP=2）：reserve = 3 GiB
  {0: ({0, 1}, 0.5), 1: ({1}, 0.45)}
  dev0: 40+3 = 43 ✓     dev1: (40+3)+(36+3) = 82 > 80              → 1 条，含 "device 1"
  再加一张没有任何 stage 的 dev2 → 不产生错误
```

关键问题

1. “共享设备”为什么要按**传递闭包**分组？举一个只看两两关系会出错的例子。
2. 组内为什么必须串行？组间并行会有什么风险（提示：显存测量）？
3. 第一次失败发生时，另一个组里**正在执行**的 `init_fn` 怎么办？能不能强行中断线程？
4. cleanup 应该在什么时刻调用：失败那一刻立即调用，还是等所有线程都结束后？为什么？
5. 多个组几乎同时失败，`InitError` 里报哪一个 stage？谁来保证“第一个”只被记录一次？
6. `admission_check` 里一个 stage 放在 `{0, 1}` 两张卡上，它的 utilization 按哪张卡的 total 算？算几次？
7. 比较用 `>` 还是 `≥`？一张卡上没有任何 stage 时要不要输出？

细节

- **组的规范形式**：每个组是**升序**的 stage id 列表；组与组之间按各自**第一个（最小）stage** 排序。这让输出与 dict 的插入顺序、set 的迭代顺序无关。
- **“no NEW init_fn calls start”**：短路只阻止**尚未开始**的 init；已经在跑的 init_fn 让它自然返回（Python 线程无法安全地被杀掉）。它如果成功，也算“succeeded”，同样要被 cleanup。
- **cleanup_fn(stage, result)** 的第二个参数是该 stage 的 `init_fn` 返回值（例如 engine 句柄），回滚时需要它来释放资源。

© 2026 kzhou003

- **InitError(failed_stage, cause)**：构造函数已给出，`stage_id` 是失败的 stage。推荐 `raise ... from cause` 保留原始 traceback。
- **admission 的公式**：对每张卡，`Σ_{stage 在该卡上} (utilization × 该卡 total + reserve)` ≤ total。`gpu_memory_utilization` 在 vLLM 里是"本进程在这张卡上最多用多少比例"，跨卡 stage 在每张卡上都各自占这个比例。
- **错误字符串**：每张超额的卡一条，必须包含子串 `device <id>`；其余内容（哪些 stage、需要多少字节）自由发挥，但越具体越好排查。

Level 2 · 思路与常见坑

- **init_groups = 连通分量**。两种做法：① 并查集：遍历每个 stage 的每个 device，记录"这个 device 第一次被谁占"，再遇到就 union；② 建二部图（stage—device）做 BFS/DFS。规模很小，哪种都行，关键是传递性。
- **run_init 的结构**：先 `init_groups`；每个组是一个任务，提交到 `ThreadPoolExecutor(max_workers)`；任务内部按 stage id 顺序 for 循环调用 init_fn。`with` 块退出时会 join 所有任务——这正是"等正在跑的结束"。
- **共享状态**：`results: dict`、`failure`（第一次失败的 (sid, exc)）。都用一把 `threading.Lock` 保护。每个组在**调用每个 init_fn 之前**检查 failure，有就 return。
- **异常不要留在 Future 里**：如果让 init_fn 的异常直接从任务里抛出，`ex.map` / `future.result()` 会在主线程重新抛出，但你就丢失了"哪个 stage"以及"先 cleanup 再 raise"的控制。在任务内部 try/except，把 (sid, e) 写进共享状态。
- **cleanup 放在线程池关闭之后**：此时所有线程都已结束，`results` 不再变化——晚完成的那个成功 stage 也会被收进来并被回收。若在失败的瞬间就 cleanup，会漏掉之后才完成的 stage。
- **常见 bug**：① 组内也并行（把每个 stage 单独 submit）→ 同卡 stage 并发初始化；② 失败后只检查"本组"是否失败，其它组还会继续启动新的 stage；③ 把失败的 stage 也 cleanup（它没有 result）；④ 只捕获 `Exception`——想想 init 里的 `KeyboardInterrupt` / `SystemExit`。
- **admission_check**：外层遍历 `device_total_bytes`（排序后遍历，输出稳定），内层找出所有 `dev in devices` 的 stage，累加 `int(util * total) + reserve`，`need > total` 才报错。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

- {0:{0,1}, 1:{1,2}, 2:{2}}：stage0 与 stage2 不直接共享卡，但都与 stage1 共享，若 0、2 并行而 1 和它们分在两组，就会出现同卡并发。所以要整个连通分量串行。
- 每个 stage init 会"测空闲显存 → 定 KV block 数"。同卡上另一个 stage 正在加载权重/profile，测量会偏，KV 算多 OOM、算少浪费。不同卡之间互不影响，所以组间并行缩短启动时间。
- 不中断，让它跑完；它成功就进 results，稍后统一 cleanup。
- 等线程池全部 join 之后。见 level2。
- 报第一个被记录的失败；在锁内 `if not failure: failure.append(...)` 保证只记一次。
- 按每张所在卡各自的 total 算，在每张卡上各算一次（含各自的 reserve）。
- `need > total` 才报错（相等允许）；空卡 need=0，不输出。

并查集推导（例 1 第一组输入）：按 stage 升序遍历，`owner[device]` 记录第一次占用者

stage	devices	动作	parent 之后
0	{2}	owner[2]=0	0→0 1→1 2→2 3→3 4→4
1	{0}	owner[0]=1	不变
2	{0,1}	dev0 已属 1 → union(2,1)；owner[1]=2	2→1
3	{1}	dev1 已属 2 → union(3, find(2)=1)	3→1
4	{5}	owner[5]=4	不变

按 root 收集：{0:[0], 1:[1,2,3], 4:[4]} → 按首元素排序 → `[[0],[1,2,3],[4]]`。union 时让较小的 id 当 root 可以让 root 恰好是组内最小值（非必须）。

思考延伸：`max_workers` 小于组数时，后面的组会排队；排到时发现已有失败就直接 return——这就是例 2 中 stage2 从未被调用的原因。若 max_workers 足够大，组[2] 可能已经在跑，那么它会跑完并被 cleanup——两种行为都符合 spec，所以写单测时要用 Event 控制时序，而不是依赖 sleep。

A-26 · CUDA graph bucket runner（CPU 模拟）

Level 1 · I/O 示例

例 1：一串调用的状态变化

```
fn = lambda x: x * 2 + 1
r = GraphRunner(fn, buckets=[16, 1, 4], hidden=3)   # self.buckets = [1, 4, 16]
r.capture()                                         # graphs 的 key: {1, 4, 16}
```


© 2026 kzhou003

```
static_in / static_out 形状: [16, 3]
每次输入 x = 全部填成 float(bs) 的 [bs, 3]
```

调用	bucket	replays	eager	last_bucket	static_in[:6, 0]	static_out[:6, 0]
run(bs=10)	16	1	0	16	10 10 10 10 10 10	21 21 21 21 21 21
run(bs=3)	4	2	0	4	3 3 3 **0** 10 10	7 7 7 **1** 21 21
run(bs=17)	无 → eager	2	1	4	不变	不变
run(bs=4)	4	3	1	4	4 4 4 4 10 10	9 9 9 9 21 21
run(bs=1, force_eager)	eager	3	2	4	不变	不变
run(bs=1)	1	4	2	1	1 4 4 4 10 10	3 9 9 9 21 21

观察：bs=3 时第 3 行被清零（上一次 bs=10 留下的 10），它的输出是 fn(0)=1；第 4、5 行在 bucket 之外，没人碰它们。

例 2：为什么一定要 clone

```
y1 = r.run(x1)        # 若直接 return static_out[:bs]（一个 view）
y2 = r.run(x2)        # replay 把 fn(x2) 写进 static_out
y1                    # → 现在等于 fn(x2)！y1 被“悄悄改写”
```

返回 clone 后，y1 保持 fn(x1)。

关键问题

1. `CpuGraph.replay()` 不接受参数。那新的输入是怎么“进入”这张图的？输出又在哪里拿？
2. 为什么所有 bucket 的图都建在**同一块** static buffer 的前 b 行，而不是每个 bucket 各自分配？
3. bs=3、bucket=4 时，第 3 行（下标）里可能是什么？不清零会导致什么后果（对本批次输出、对 NaN、对 MoE 路由/统计）？
4. `static_in[bs:]` 全部清零 vs 只清 `static_in[bs:b]`：哪个是必要的？多清有什么代价？
5. `force_eager` 的典型场景是请求带了自己的 `torch.Generator`。为什么 replay 会“悄悄忽略”它，而不是报错？
6. `last_bucket` 在走 eager 时要不要更新？
7. 如果 `run` 发现某个 bucket 还没 capture，能不能现场 capture 一张？为什么 spec 说 never？

细节

- **静态 buffer**：CUDA graph 记录的是 kernel 以及它们读写的**设备地址**。replay 时地址不变，所以必须把新数据 `copy_` 进同一块内存，不能把 `self.static_in` 重新赋值成新 tensor（那样图读的还是旧内存）。
- **view**：`static_in[:b]` 是共享存储的 view，不是拷贝。b=4 的图读写的是大 buffer 的前 4 行；所有图共享同一块内存，所以上一次大 batch 的数据会残留在后面的行里。
- **“largest bucket first”**：在 CUDA 上多张图共享一个 memory pool，先录最大的图会分配最大的中间张量，小图可以复用这些块，省显存。CPU 模拟里顺序不影响结果，但请按 spec 来。
- **bucket 选择**：最小的 `b ≥ bs`。`buckets` 在 __init__ 里已经排好序，直接线性找即可（也可用 `bisect`）。
- **stats**：`replays` 与 `eager` 是计数器；`last_bucket` 只在 replay 时记录所用的 bucket。
- **返回值形状**：只返回前 bs 行（padding 行的结果是“对 0 做计算”的垃圾，不能给调用方）。

Level 2 · 思路与常见坑

- **__init__**：`max_bs = buckets[-1]`；分配 `static_in`、`static_out`，形状 `[max_bs, hidden]`，dtype 用传入的 `dtype`。初始化成 0 比 `empty` 更安全（capture 时就会跑一次 fn）。注意 stub 末尾的 `raise NotImplementedError` 要删掉，且已有的字段要保留。
- **capture**：遍历 `sorted(buckets, reverse=True)`，对每个 b 用 `static_in[:b]` / `static_out[:b]` 构造 `CpuGraph`，调 `.capture()`，存进 `self.graphs[b]`。
- **run 的决策顺序**：① 算 bs；② force_eager 或找不到 bucket（或该 bucket 不在 graphs 里）→ eager 分支：计数 +1，`return self.fn(x)`；③ 否则走 replay 分支。eager 分支不碰静态 buffer。
- **replay 分支四步**：copy 输入 → 清零 [bs, b) → replay → 更新 stats → 返回 clone 的前 bs 行。顺序上，清零必须在 replay 之前；clone 必须在任何下一次 replay 之前（即返回前）。
- **pitfall：赋值而不是 copy**。`self.static_in[:bs] = x` 其实是原地写（等价于 copy_），没问题；但 `self.static_in = x` 或在 graph 里换 tensor 都是错的——图看不到。
- **pitfall：忘记清零**。在本题的 fn（逐行计算）里，脏行只影响 padding 行的输出，看起来“结果是对的”；但真实模型里有跨行交互（attention 的 batch 维、MoE 负载均衡、BN 类统计、NaN/Inf 传播），脏行会污染真实行。所以要显式清零，并能在 `static_in` 上观察到。
- **pitfall：返回 view**。单次调用测不出来，连续两次调用后第一次的结果被改写（例 2）。

Level 3 · 详解（代码骨架见附录 C 的带注释参考解）

key questions 答案

© 2026 kzhou003

- 输入：先 `copy_` 进 static_in；replay 读 static_in、写 static_out；输出从 static_out 读。
- 共享一块 buffer：省内存，而且 run 时只需一份“写入位置”，无论选哪个 bucket，都写 static_in 的前 bs 行。
- 可能是上一次更大 batch 的真实数据（例 1 中的 10）。本题 fn 逐行独立，不清零只让 padding 输出变成垃圾；真实模型中可能 NaN 扩散、影响 MoE 路由与 expert 负载统计、影响需要整 batch 归约的算子。
- 只需清 [bs, b)：这是这张图会读的范围；b 之后的行这张图根本不读，清它们只是多花带宽。
- replay 只是重放 capture 时录下的 kernel，不重新执行 Python；capture 时用的是全局 RNG 状态/固定的 philox offset，per-row generator 的 Python 逻辑根本不会被执行，所以不会报错，只会结果不可复现/不符合 seed。
- 不更新；它的语义是“最近一次 replay 用的 bucket”。
- 不能：capture 很慢（要 warmup、同步），而且在 CUDA 上会和正在运行的 stream/内存池冲突；运行时捕获会造成延迟尖刺。所有图必须在启动期录好。

run 逐步追踪（例 1 第 2 次调用，bs=3）

步骤	操作	static_in[:5,0]	static_out[:5,0]
0	调用前（上次 bs=10）	10 10 10 10 10	21 21 21 21 21
1	b = 最小的 ≥3 → 4	同上	同上
2	static_in[:3] ← x	3 3 3 10 10	同上
3	static_in[3:4] ← 0	3 3 3 0 10	同上
4	graphs[4].replay()	同上	7 7 7 1 21
5	stats: replays=2, last_bucket=4		
6	return static_out[:3].clone()	[3,3] 全 7 的新 tensor	

自检清单：① 连续两次 run，第一次的返回值不变；② 先跑一次大 batch 再跑小 batch，检查 `static_in` 的 [bs, b) 行全 0；③ capture 之后 `self.graphs` 里每个对象的 `id()` 在多次 run 后不变；④ bs > max bucket 和 force_eager 时，static buffer 完全不被修改。

© 2026 kzhou003

附录 B · 完整测试代码

B-0 · tests/_tiny.py（Track M 共用的 tiny HF 构造器）

```
"""Tiny random Qwen3-Omni configs/modules from HF transformers, used as the golden reference."""
import pytest
import torch

transformers = pytest.importorskip("transformers")
C = pytest.importorskip("transformers.models.qwen3_omni_moe.configuration_qwen3_omni_moe")
M = pytest.importorskip("transformers.models.qwen3_omni_moe.modeling_qwen3_omni_moe")

HIDDEN, HEADS, KV_HEADS, HEAD_DIM = 64, 4, 2, 16
MROPE_SECTION = [4, 2, 2]  # sums to HEAD_DIM // 2, same interleave pattern as the real [24, 20, 20]


def thinker_text_config(**kw):
    cfg = C.Qwen3OmniMoeTextConfig(
        vocab_size=97, hidden_size=HIDDEN, intermediate_size=96, num_hidden_layers=2,
        num_attention_heads=HEADS, num_key_value_heads=KV_HEADS, head_dim=HEAD_DIM,
        moe_intermediate_size=32, num_experts=6, num_experts_per_tok=2, norm_topk_prob=True,
        rope_theta=10000.0, max_position_embeddings=256, rms_norm_eps=1e-6,
        rope_scaling={"rope_type": "default", "mrope_section": MROPE_SECTION, "interleaved": True},
        **kw,
    )
    cfg._attn_implementation = "eager"
    return cfg


def talker_text_config():
    cfg = C.Qwen3OmniMoeTalkerTextConfig(
        vocab_size=50, hidden_size=HIDDEN, intermediate_size=96, num_hidden_layers=1,
        num_attention_heads=HEADS, num_key_value_heads=KV_HEADS, moe_intermediate_size=32,
        num_experts=6, num_experts_per_tok=2, norm_topk_prob=True, shared_expert_intermediate_size=48,
    )
    cfg._attn_implementation = "eager"
    return cfg


def code_predictor_config(num_code_groups=6, vocab=40):
    cfg = C.Qwen3OmniMoeTalkerCodePredictorConfig(
        vocab_size=vocab, hidden_size=HIDDEN, intermediate_size=96, num_hidden_layers=2,
        num_attention_heads=HEADS, num_key_value_heads=KV_HEADS, head_dim=HEAD_DIM,
        num_code_groups=num_code_groups, max_position_embeddings=64, rms_norm_eps=1e-6,
    )
    cfg._attn_implementation = "eager"
    return cfg


def randomize_(module, seed=0, std=0.2):
    """HF inits RMSNorm weights to 1 and some linears to ~0; randomize so bugs are visible."""
    g = torch.Generator().manual_seed(seed)
    with torch.no_grad():
        for p in module.parameters():
            p.copy_(torch.randn(p.shape, generator=g) * std + (1.0 if p.ndim == 1 else 0.0))
    return module.eval()


def causal_mask(t, past=0):
    m = torch.full((t, past + t), float("-inf"))
    m = torch.triu(m, diagonal=past + 1)
    return m[None, None]
```


© 2026 kzhou003

B-1 · tests/test_m01_rmsnorm_mrope.py

```
import torch

from drill import load
from tests._tiny import HEAD_DIM, HIDDEN, MROPE_SECTION, M, thinker_text_config

m = load("m01_rmsnorm_mrope")


def mm_positions():
    # text(3) | image grid t=1,h=2,w=3 starting at 3 | text(2)
    t = [0, 1, 2, 3, 3, 3, 3, 3, 3, 5, 6]
    h = [0, 1, 2, 3, 3, 3, 4, 4, 4, 5, 6]
    w = [0, 1, 2, 3, 4, 5, 3, 4, 5, 5, 6]
    return torch.tensor([t, h, w])


def test_rms_norm_matches_hf_fp32_and_bf16():
    norm = M.Qwen3OmniMoeThinkerTextRMSNorm(HIDDEN, eps=1e-6)
    with torch.no_grad():
        norm.weight.copy_(torch.randn(HIDDEN))
    x = torch.randn(3, 7, HIDDEN) * 5
    torch.testing.assert_close(m.rms_norm(x, norm.weight, 1e-6), norm(x))
    xb = x.bfloat16()
    torch.testing.assert_close(m.rms_norm(xb, norm.weight.bfloat16(), 1e-6), norm.bfloat16()(xb))


def test_inv_freq():
    cfg = thinker_text_config()
    rot = M.Qwen3OmniMoeThinkerTextRotaryEmbedding(cfg)
    torch.testing.assert_close(m.rope_inv_freq(HEAD_DIM, cfg.rope_theta), rot.inv_freq)


def test_interleaved_mrope_matches_hf_qwen3_omni():
    cfg = thinker_text_config()
    rot = M.Qwen3OmniMoeThinkerTextRotaryEmbedding(cfg)
    pos = mm_positions()
    x = torch.zeros(1, pos.shape[1], HIDDEN)
    cos_hf, sin_hf = rot(x, pos[:, None, :])  # [1, T, D]
    cos, sin = m.mrope_cos_sin(rot.inv_freq, pos, MROPE_SECTION, interleaved=True)
    torch.testing.assert_close(cos, cos_hf[0])
    torch.testing.assert_close(sin, sin_hf[0])


def test_text_only_positions_reduce_to_plain_rope():
    inv = m.rope_inv_freq(HEAD_DIM, 10000.0)
    pos = torch.arange(9).expand(3, 9)
    cos_i, _ = m.mrope_cos_sin(inv, pos, MROPE_SECTION, interleaved=True)
    cos_c, _ = m.mrope_cos_sin(inv, pos, MROPE_SECTION, interleaved=False)
    freqs = torch.arange(9).float()[:, None] * inv[None]
    plain = torch.cat([freqs, freqs], -1).cos()
    torch.testing.assert_close(cos_i, plain)
    torch.testing.assert_close(cos_c, plain)


def test_apply_rotary_matches_hf_qwen3_omni():
    cfg = thinker_text_config()
    rot = M.Qwen3OmniMoeThinkerTextRotaryEmbedding(cfg)
    pos = mm_positions()
    T = pos.shape[1]
    q, k = torch.randn(T, 4, HEAD_DIM), torch.randn(T, 2, HEAD_DIM)
    cos_hf, sin_hf = rot(torch.zeros(1, T, HIDDEN), pos[:, None, :])
    q_hf, k_hf = M.apply_rotary_pos_emb(q.transpose(0, 1)[None], k.transpose(0, 1)[None], cos_hf, sin_hf)
    cos, sin = m.mrope_cos_sin(rot.inv_freq, pos, MROPE_SECTION, interleaved=True)
    torch.testing.assert_close(m.apply_rotary(q, cos, sin), q_hf[0].transpose(0, 1))
    torch.testing.assert_close(m.apply_rotary(k, cos, sin), k_hf[0].transpose(0, 1))


def test_chunked_mrope_matches_hf_qwen2_5_omni():
    from transformers.models.qwen2_5_omni import modeling_qwen2_5_omni as Q25

    pos = mm_positions()
    T = pos.shape[1]
    inv = m.rope_inv_freq(HEAD_DIM, 10000.0)
```


© 2026 kzhou003

```
    freqs = pos.float()[:, :, None] * inv  # [3, T, half] -> full per-axis cos/sin like HF
    emb = torch.cat([freqs, freqs], -1)[:, None]  # [3, 1, T, D]
    q = torch.randn(1, 4, T, HEAD_DIM)
    q_hf, _ = Q25.apply_multimodal_rotary_pos_emb(q, q, emb.cos(), emb.sin(), MROPE_SECTION)
    cos, sin = m.mrope_cos_sin(inv, pos, MROPE_SECTION, interleaved=False)
    torch.testing.assert_close(m.apply_rotary(q[0].transpose(0, 1), cos, sin), q_hf[0].transpose(0, 1))
```

B-2 · tests/test_m02_thinker_attention.py

```
import torch

from drill import load
from tests._tiny import HIDDEN, M, causal_mask, randomize_, thinker_text_config

m = load("m02_thinker_attention")


def setup(T=10, seed=0):
    cfg = thinker_text_config()
    att = randomize_(M.Qwen3OmniMoeThinkerTextAttention(cfg, layer_idx=0), seed)
    rot = M.Qwen3OmniMoeThinkerTextRotaryEmbedding(cfg)
    torch.manual_seed(seed)
    x = torch.randn(T, HIDDEN)
    pos = torch.arange(T).expand(3, T).clone()
    pos[1, 3:7] += torch.tensor([0, 0, 1, 1])  # a tiny 2x2 "image" grid in the middle
    pos[2, 3:7] += torch.tensor([0, 1, 0, 1])
    cos, sin = rot(x[None], pos[:, None, :])
    with torch.no_grad():
        ref, _ = att(x[None], (cos, sin), causal_mask(T))
    return att, x, cos[0], sin[0], ref[0]


def test_prefill_matches_hf():
    att, x, cos, sin, ref = setup()
    out = m.attention_forward(m.from_hf(att), x, cos, sin)
    torch.testing.assert_close(out, ref, atol=1e-5, rtol=1e-4)


def test_decode_one_token_at_a_time_with_cache():
    att, x, cos, sin, ref = setup(T=12, seed=1)
    w, cache = m.from_hf(att), m.KVCache()
    out = m.attention_forward(w, x[:5], cos[:5], sin[:5], cache)
    rows = [out]
    for t in range(5, 12):
        rows.append(m.attention_forward(w, x[t:t + 1], cos[t:t + 1], sin[t:t + 1], cache))
    torch.testing.assert_close(torch.cat(rows), ref, atol=1e-5, rtol=1e-4)
    assert len(cache) == 12


def test_chunked_prefill_with_cache():
    att, x, cos, sin, ref = setup(T=11, seed=2)
    w, cache = m.from_hf(att), m.KVCache()
    a = m.attention_forward(w, x[:4], cos[:4], sin[:4], cache)
    b = m.attention_forward(w, x[4:], cos[4:], sin[4:], cache)  # a chunk of 7 after 4 cached tokens
    torch.testing.assert_close(torch.cat([a, b]), ref, atol=1e-5, rtol=1e-4)
```


© 2026 kzhou003

B-3 · tests/test_m03_sparse_moe.py

```
import torch

from drill import load
from tests._tiny import HIDDEN, M, randomize_, talker_text_config, thinker_text_config

m = load("m03_sparse_moe")


def run_hf(block, x):
    with torch.no_grad():
        y, logits = block(x[None])
    return y[0], logits


def test_routing_matches_hf_softmax_topk_renorm():
    block = randomize_(M.Qwen3OmniMoeThinkerTextSparseMoeBlock(thinker_text_config()), seed=0, std=0.5)
    x = torch.randn(17, HIDDEN)
    w = m.from_hf(block)
    topw, topi = m.route(x, w)
    probs = torch.softmax(x @ w.router.T, -1)
    ref_w, ref_i = torch.topk(probs, 2, -1)
    ref_w = ref_w / ref_w.sum(-1, keepdim=True)
    assert torch.equal(topi, ref_i)
    torch.testing.assert_close(topw, ref_w)
    torch.testing.assert_close(topw.sum(-1), torch.ones(17))


def test_thinker_moe_matches_hf():
    block = randomize_(M.Qwen3OmniMoeThinkerTextSparseMoeBlock(thinker_text_config()), seed=1)
    x = torch.randn(23, HIDDEN)
    ref, _ = run_hf(block, x)
    torch.testing.assert_close(m.moe_forward(x, m.from_hf(block)), ref, atol=1e-5, rtol=1e-4)


def test_talker_moe_with_shared_expert_matches_hf():
    block = randomize_(M.Qwen3OmniMoeTalkerTextSparseMoeBlock(talker_text_config()), seed=2)
    x = torch.randn(9, HIDDEN)
    ref, _ = run_hf(block, x)
    torch.testing.assert_close(m.moe_forward(x, m.from_hf(block)), ref, atol=1e-5, rtol=1e-4)


def test_no_renorm_variant():
    cfg = thinker_text_config()
    cfg.norm_topk_prob = False
    block = randomize_(M.Qwen3OmniMoeThinkerTextSparseMoeBlock(cfg), seed=3)
    x = torch.randn(8, HIDDEN)
    ref, _ = run_hf(block, x)
    torch.testing.assert_close(m.moe_forward(x, m.from_hf(block)), ref, atol=1e-5, rtol=1e-4)


def test_single_token_and_batch_invariance():
    block = randomize_(M.Qwen3OmniMoeThinkerTextSparseMoeBlock(thinker_text_config()), seed=4)
    w = m.from_hf(block)
    x = torch.randn(6, HIDDEN)
    full = m.moe_forward(x, w)
    for i in range(6):
        torch.testing.assert_close(m.moe_forward(x[i:i + 1], w)[0], full[i], atol=1e-5, rtol=1e-4)
```


© 2026 kzhou003

B-4 · tests/test_m04_mm_merge_deepstack.py

```
import pytest
import torch

from drill import load
from tests._tiny import HIDDEN, M, causal_mask, randomize_, thinker_text_config

m = load("m04_mm_merge_deepstack")
IMG, AUD = 90, 91


def test_audio_token_count_matches_hf():
    frames = torch.tensor([1, 2, 7, 99, 100, 101, 199, 200, 250, 1234, 3000])
    torch.testing.assert_close(m.audio_token_count(frames), M._get_feat_extract_output_lengths(frames))
    assert m.audio_token_count(100) == 13


def test_merge_places_embeddings_in_order():
    ids = torch.tensor([1, IMG, IMG, 2, AUD, AUD, AUD, 3, IMG])
    text = torch.randn(9, HIDDEN)
    img, aud = torch.randn(3, HIDDEN), torch.randn(3, HIDDEN)
    text_before = text.clone()
    out, masks = m.merge_multimodal(ids, text, {"image": IMG, "audio": AUD}, {"image": img, "audio": aud})
    torch.testing.assert_close(out[[1, 2, 8]], img)
    torch.testing.assert_close(out[[4, 5, 6]], aud)
    torch.testing.assert_close(out[[0, 3, 7]], text[[0, 3, 7]])
    assert masks["image"].tolist() == [False, True, True, False, False, False, False, False, True]
    assert torch.equal(text, text_before), "do not mutate the text embeddings in place"


def test_merge_count_mismatch_raises():
    ids = torch.tensor([IMG, IMG, 5])
    with pytest.raises(ValueError):
        m.merge_multimodal(ids, torch.randn(3, HIDDEN), {"image": IMG}, {"image": torch.randn(3, HIDDEN)})
    with pytest.raises(ValueError):
        m.merge_multimodal(ids, torch.randn(3, HIDDEN), {"image": IMG}, {})


def test_deepstack_matches_hf_text_model():
    cfg = thinker_text_config()
    model = randomize_(M.Qwen3OmniMoeThinkerTextModel(cfg), seed=0)
    T = 10
    emb = torch.randn(1, T, HIDDEN)
    vis = torch.zeros(T, dtype=torch.bool)
    vis[3:7] = True
    deep = [torch.randn(4, HIDDEN), torch.randn(4, HIDDEN)]
    pos = torch.arange(T).expand(3, 1, T)
    with torch.no_grad():
        ref = model(inputs_embeds=emb, position_ids=pos, visual_pos_masks=vis[None, :, None].expand(1, T, HIDDEN),
                    deepstack_visual_embeds=deep).last_hidden_state[0]
        cos_sin = model.rotary_emb(emb, pos)
        mask = causal_mask(T)
        layers = [lambda h, L=L: L(h[None], position_embeddings=cos_sin, attention_mask=mask)[0] for L in
model.layers]
        out = model.norm(m.run_layers(layers, emb[0].clone(), vis, deep))
    torch.testing.assert_close(out, ref, atol=1e-5, rtol=1e-4)


def test_deepstack_only_touches_visual_rows():
    h = torch.randn(6, HIDDEN)
    vis = torch.tensor([0, 1, 0, 1, 0, 0], dtype=torch.bool)
    v = torch.randn(2, HIDDEN)
    out = m.apply_deepstack(h, vis, v)
    torch.testing.assert_close(out[vis], h[vis] + v)
    torch.testing.assert_close(out[~vis], h[~vis])
```


© 2026 kzhou003

B-5 · tests/test_m05_talker_mtp.py

```
import torch

from drill import load
from tests._tiny import HIDDEN, M, code_predictor_config, randomize_

m = load("m05_talker_mtp")


def setup(G=6, B=3, seed=0):
    cp = randomize_(M.Qwen3OmniMoeTalkerCodePredictorModelForConditionalGeneration(code_predictor_config(G)),
seed, 0.3)
    g = torch.Generator().manual_seed(seed + 100)
    hid, cb0 = torch.randn(B, HIDDEN, generator=g), torch.randn(B, HIDDEN, generator=g)
    return cp, hid, cb0


def hf_greedy(cp, hid, cb0):
    with torch.no_grad():
        res = cp.generate(inputs_embeds=torch.stack([hid, cb0], 1), max_new_tokens=cp.config.num_code_groups - 1,
                          do_sample=False, output_hidden_states=True, return_dict_in_generate=True)
    return res


def test_reprefill_matches_hf_generate():
    cp, hid, cb0 = setup()
    ref = hf_greedy(cp, hid, cb0).sequences
    codes = m.predict_residual_codes(cp, hid, cb0)
    assert codes.shape == (3, 5)
    assert torch.equal(codes, ref)


def test_kv_cache_version_matches_reprefill():
    for seed in range(3):
        cp, hid, cb0 = setup(G=7, B=2, seed=seed)
        assert torch.equal(m.predict_residual_codes_kv(cp, hid, cb0), m.predict_residual_codes(cp, hid, cb0))


def test_rows_are_independent():
    cp, hid, cb0 = setup(B=4, seed=5)
    full = m.predict_residual_codes(cp, hid, cb0)
    for i in range(4):
        assert torch.equal(m.predict_residual_codes(cp, hid[i:i + 1], cb0[i:i + 1])[0], full[i])


def test_next_input_matches_hf_summation():
    cp, hid, cb0 = setup(seed=7)
    res = hf_greedy(cp, hid, cb0)
    text_step = torch.randn(3, HIDDEN)
    # HF Qwen3OmniMoeTalkerForConditionalGeneration.prepare_inputs_for_generation
    mid = [h[0] for h in res.hidden_states[1:]]                 # embeddings of residual codes 1..G-2
    last = cp.get_input_embeddings()[-1](res.sequences[..., -1:])  # embedding of residual code G-1
    with torch.no_grad():
        ref = torch.cat([cb0[:, None]] + mid + [last], dim=1).sum(1) + text_step
        got = m.next_talker_input(cp, cb0, res.sequences, text_step)
    torch.testing.assert_close(got, ref, atol=1e-5, rtol=1e-4)
```


© 2026 kzhou003

B-6 · tests/test_m06_streaming_causal_conv.py

```
import random

import torch

from drill import load
from tests._tiny import M, randomize_

m = load("m06_streaming_causal_conv")


def random_chunks(total, seed):
    rnd = random.Random(seed)
    sizes, left = [], total
    while left:
        sizes.append(min(left, rnd.randint(1, 6)))
        left -= sizes[-1]
    return sizes


def test_causal_conv_streaming_equals_offline():
    for seed, (k, d) in enumerate([(3, 1), (5, 2), (7, 1), (1, 1)]):
        conv = randomize_(M.Qwen3OmniMoeCausalConvNet(4, 6, kernel_size=k, dilation=d), seed)
        x = torch.randn(2, 4, 37)
        with torch.no_grad():
            ref = conv(x)
        s = m.StreamingCausalConv1d.from_hf(conv)
        outs, pos = [], 0
        for n in random_chunks(37, seed):
            y = s.step(x[..., pos:pos + n])
            assert y.shape[-1] == n, "a causal stride-1 conv emits exactly one output per input"
            outs.append(y)
            pos += n
        torch.testing.assert_close(torch.cat(outs, -1), ref, atol=1e-5, rtol=1e-4)


def test_trans_conv_streaming_equals_offline():
    for seed, (k, st) in enumerate([(4, 2), (10, 5), (6, 3), (3, 3)]):
        conv = randomize_(M.Qwen3OmniMoeCausalTransConvNet(4, 5, kernel_size=k, stride=st), seed)
        x = torch.randn(2, 4, 23)
        with torch.no_grad():
            ref = conv(x)
        s = m.StreamingCausalTransConv1d.from_hf(conv)
        outs, pos = [], 0
        for n in random_chunks(23, seed + 10):
            outs.append(s.step(x[..., pos:pos + n]))
            pos += n
        outs.append(s.flush())
        torch.testing.assert_close(torch.cat(outs, -1), ref, atol=1e-5, rtol=1e-4)


def test_trans_conv_emits_as_early_as_possible():
    k, st = 8, 4
    conv = randomize_(M.Qwen3OmniMoeCausalTransConvNet(3, 3, kernel_size=k, stride=st), 0)
    s = m.StreamingCausalTransConv1d.from_hf(conv)
    emitted = 0
    for n in range(1, 8):
        emitted += s.step(torch.randn(1, 3, 1)).shape[-1]
        # positions < n*stride are final; the first (k - stride) are trimmed
        assert emitted == max(0, n * st - (k - st)), (n, emitted)
```


© 2026 kzhou003

B-7 · tests/test_m07_stage_weight_loading.py

```
import pytest
import torch
import torch.nn.functional as F

from drill import load
from tests._tiny import HIDDEN, M, randomize_, thinker_text_config

m = load("m07_stage_weight_loading")


def checkpoint():
    cfg = thinker_text_config()
    model = randomize_(M.Qwen3OmniMoeThinkerTextModel(cfg), seed=0)
    state = {f"thinker.model.{k}": v.detach().clone() for k, v in model.state_dict().items()}
    state["thinker.lm_head.weight"] = torch.randn(cfg.vocab_size, HIDDEN)
    state["talker.model.layers.0.self_attn.q_proj.weight"] = torch.randn(8, 8)
    state["code2wav.decoder.0.conv.weight"] = torch.randn(4, 4, 3)
    return cfg, model, state


def test_shard_planning():
    wm = {"thinker.model.a": "s1", "thinker.model.b": "s2", "talker.x": "s2", "code2wav.y": "s3",
"thinker.lm_head": "s1"}
    assert m.shards_for_stage(wm, "thinker") == ["s1", "s2"]
    assert m.shards_for_stage(wm, "code2wav") == ["s3"]


def test_select_stage_strips_prefix_and_drops_other_stages():
    _, model, state = checkpoint()
    sd = m.select_stage(state, "thinker", "thinker.model.")
    assert set(sd) == set(model.state_dict())


def test_packed_tensors_reproduce_hf_projections():
    cfg, model, state = checkpoint()
    globals_, layers = m.load_thinker_text(state, cfg.num_hidden_layers)
    assert torch.equal(globals_["embed_tokens"], model.embed_tokens.weight)
    x = torch.randn(5, HIDDEN)
    D, H, KVH = cfg.head_dim, cfg.num_attention_heads, cfg.num_key_value_heads
    for i, layer in enumerate(layers):
        hf = model.layers[i]
        q, k, v = (x @ layer.qkv.T).split([H * D, KVH * D, KVH * D], dim=-1)
        with torch.no_grad():
            torch.testing.assert_close(q, hf.self_attn.q_proj(x))
            torch.testing.assert_close(k, hf.self_attn.k_proj(x))
            torch.testing.assert_close(v, hf.self_attn.v_proj(x))
            for e, ex in enumerate(hf.mlp.experts):
                g, u = (x @ layer.w13[e].T).chunk(2, dim=-1)
                torch.testing.assert_close((F.silu(g) * u) @ layer.w2[e].T, ex(x))
        assert torch.equal(layer.router, hf.mlp.gate.weight)


def test_missing_and_unexpected_keys_are_errors():
    cfg, _, state = checkpoint()
    broken = dict(state)
    del broken["thinker.model.layers.1.mlp.experts.3.up_proj.weight"]
    with pytest.raises(m.LoadError):
        m.load_thinker_text(broken, cfg.num_hidden_layers)
    extra = dict(state)
    extra["thinker.model.layers.0.self_attn.rotary.inv_freq"] = torch.zeros(4)
    with pytest.raises(m.LoadError):
        m.load_thinker_text(extra, cfg.num_hidden_layers)
```


© 2026 kzhou003

B-8 · tests/test_m08_tiny_thinker_e2e.py

```
import torch

from drill import load
from tests._tiny import HIDDEN, MROPE_SECTION, M, randomize_, thinker_text_config

m = load("m08_tiny_thinker_e2e")


def build(seed=0):
    cfg = thinker_text_config()
    model = randomize_(M.Qwen3OmniMoeThinkerTextModel(cfg), seed=seed, std=0.3)
    lm_head = torch.randn(cfg.vocab_size, HIDDEN, generator=torch.Generator().manual_seed(seed)) * 0.5
    state = {f"thinker.model.{k}": v.detach().clone() for k, v in model.state_dict().items()}
    state["thinker.lm_head.weight"] = lm_head
    dims = m.Dims(cfg.num_hidden_layers, cfg.num_attention_heads, cfg.num_key_value_heads, cfg.head_dim,
                  cfg.num_experts_per_tok, cfg.norm_topk_prob, cfg.rope_theta, MROPE_SECTION, cfg.rms_norm_eps)
    return model, lm_head, state, dims


@torch.no_grad()
def hf_logits(model, lm_head, ids):
    return model(input_ids=torch.tensor([ids])).last_hidden_state[0] @ lm_head.T


@torch.no_grad()
def hf_greedy(model, lm_head, prompt, n):
    ids = list(prompt)
    for _ in range(n):
        ids.append(int(hf_logits(model, lm_head, ids)[-1].argmax()))
    return ids[len(prompt):]


def test_prefill_logits_match_hf():
    model, lm_head, state, dims = build()
    tt = m.TinyThinker(state, dims)
    ids = [3, 14, 15, 92, 65, 35, 8]
    caches = [m.KVCache() for _ in range(dims.num_layers)]
    with torch.no_grad():
        got = tt.forward(torch.tensor(ids), 0, caches)
    torch.testing.assert_close(got, hf_logits(model, lm_head, ids), atol=1e-4, rtol=1e-4)


def test_greedy_with_kv_cache_matches_hf_full_recompute():
    for seed in range(2):
        model, lm_head, state, dims = build(seed)
        tt = m.TinyThinker(state, dims)
        prompt = [1, 2, 3, 4, 5, 6][: 3 + seed * 3]
        assert tt.greedy(prompt, 12) == hf_greedy(model, lm_head, prompt, 12)
```


© 2026 kzhou003

B-9 · tests/test_p02_orchestrator.py

```
import asyncio
import random

from drill import load

m = load("p02_orchestrator")
StageOutput = m.StageOutput


class FakeStage:
    """Emits len(fn(rid, inputs)) outputs asynchronously after each submit."""

    def __init__(self, fn, max_delay=0.002, emit_first_in_submit=False):
        self.fn = fn
        self.max_delay = max_delay
        self.emit_first_in_submit = emit_first_in_submit
        self.q: asyncio.Queue = asyncio.Queue()
        self.submitted: list = []
        self.aborted: list = []

    async def submit(self, rid, inputs):
        self.submitted.append((rid, inputs))
        chunks = self.fn(rid, inputs)
        start = 0
        if self.emit_first_in_submit:
            await self.q.put([StageOutput(rid, chunks[0], len(chunks) == 1)])
            start = 1
            await asyncio.sleep(0.01)  # let readers + dispatcher run before we return
        asyncio.get_running_loop().create_task(self._emit(rid, chunks, start))

    async def _emit(self, rid, chunks, start):
        for i in range(start, len(chunks)):
            await asyncio.sleep(random.random() * self.max_delay)
            await self.q.put([StageOutput(rid, chunks[i], i == len(chunks) - 1)])

    async def get_output(self):
        return await self.q.get()

    async def abort(self, request_ids):
        self.aborted += request_ids


def thinker(rid, prompt):
    return [f"{rid}:t{i}" for i in range(5)]


def talker(rid, inputs):
    up = [x.upper() for x in inputs]
    return [up[:3], up[3:]]


def vocoder(rid, inputs):
    flat = [x for part in inputs for x in part]
    return [f"wav({x})" for x in flat]


def forward(next_stage, prompt, prev_data):
    return list(prev_data)


def make(final={0, 2}, **kw):
    stages = [FakeStage(thinker, **kw), FakeStage(talker), FakeStage(vocoder)]
    return stages, m.Orchestrator(stages, final, forward)


async def collect(orch, n_requests, timeout=5.0):
    per = {}
    done = 0
    while done < n_requests:
        o = await orch.next_output(timeout=timeout)
        per.setdefault(o.request_id, []).append(o)
        if o.finished:
            done += 1
```


© 2026 kzhou003

```
    return per


def check_request(rid, outs, final={0, 2}):
    assert sum(o.finished for o in outs) == 1, "exactly one terminal message"
    assert outs[-1].finished, "terminal message must be the last one"
    assert {o.stage_id for o in outs} == final, "only final stages reach the client"
    if 0 in final:
        assert [o.data for o in outs if o.stage_id == 0] == thinker(rid, None)
    expected_wav = [f"wav({x.upper()})" for x in thinker(rid, None)]
    assert [o.data for o in outs if o.stage_id == 2] == expected_wav


def test_single_request_three_stages():
    async def main():
        stages, orch = make()
        await orch.start()
        await orch.add_request("r0", "hello")
        per = await collect(orch, 1)
        check_request("r0", per["r0"])
        assert orch.request_states == {}
        await orch.shutdown()

    asyncio.run(main())


def test_many_concurrent_requests():
    async def main():
        stages, orch = make()
        await orch.start()
        rids = [f"r{i}" for i in range(30)]
        for rid in rids:
            await orch.add_request(rid, "p")
        per = await collect(orch, len(rids))
        for rid in rids:
            check_request(rid, per[rid])
        # each stage got each request exactly once
        for s in stages:
            assert sorted(r for r, _ in s.submitted) == sorted(rids)
        await orch.shutdown()

    asyncio.run(main())


def test_only_last_stage_is_final():
    async def main():
        stages, orch = make(final={2})
        await orch.start()
        await orch.add_request("a", "p")
        per = await collect(orch, 1)
        check_request("a", per["a"], final={2})
        await orch.shutdown()

    asyncio.run(main())


def test_state_registered_before_submit():
    async def main():
        stages, orch = make(emit_first_in_submit=True)
        await orch.start()
        await orch.add_request("early", "p")
        per = await collect(orch, 1)
        check_request("early", per["early"])
        assert orch.dropped_outputs == 0
        await orch.shutdown()

    asyncio.run(main())


def test_unknown_request_output_is_dropped():
    async def main():
        stages, orch = make()
        await orch.start()
        await stages[0].q.put([StageOutput("ghost", "x", True)])
        await orch.add_request("r", "p")
        per = await collect(orch, 1)
        check_request("r", per["r"])
```


© 2026 kzhou003

```
        assert orch.dropped_outputs == 1
        await orch.shutdown()

    asyncio.run(main())
```


© 2026 kzhou003

B-10 · tests/test_p03_async_chunk_orchestrator.py

```
import asyncio
import random

from drill import load

m = load("p03_async_chunk_orchestrator")
StageOutput = m.StageOutput


class Channel:
    """Side channel between two fake stages (stands in for the SHM connector)."""

    def __init__(self):
        self.qs: dict[str, asyncio.Queue] = {}

    def q(self, rid):
        return self.qs.setdefault(rid, asyncio.Queue())


class ChunkStage:
    """Stage that reads chunks from `inp` (or the prompt for stage 0), transforms
    each one, forwards it on `out`, and reports outputs to the orchestrator.

    late_finish: report finished=True to the orchestrator only after `late_finish` s
    (simulates a non-final stage whose terminal report arrives after the final stage).
    """

    def __init__(self, transform, inp=None, out=None, late_finish=0.0, n_tokens=6):
        self.transform, self.inp, self.out = transform, inp, out
        self.late_finish, self.n_tokens = late_finish, n_tokens
        self.q: asyncio.Queue = asyncio.Queue()
        self.submitted: list = []

    async def submit(self, rid, inputs):
        self.submitted.append((rid, inputs))
        asyncio.get_running_loop().create_task(self._run(rid))

    async def _run(self, rid):
        if self.inp is None:
            items = [(f"{rid}:t{i}", i == self.n_tokens - 1) for i in range(self.n_tokens)]
        else:
            items = None
        i = 0
        while True:
            if items is not None:
                if i == len(items):
                    break
                data, last = items[i]
                i += 1
            else:
                data, last = await self.inp.q(rid).get()
            await asyncio.sleep(random.random() * 0.002)
            y = self.transform(data)
            if self.out is not None:
                await self.out.q(rid).put((y, last))
            if last and self.late_finish:
                await self.q.put([StageOutput(rid, y, False)])
                await asyncio.sleep(self.late_finish)
                await self.q.put([StageOutput(rid, None, True)])
            else:
                await self.q.put([StageOutput(rid, y, last)])
            if last:
                break

    async def get_output(self):
        return await self.q.get()

    async def abort(self, ids):
        pass


def build(late_finish=0.0):
    c01, c12 = Channel(), Channel()
```


© 2026 kzhou003

```
    stages = [
        ChunkStage(lambda x: x, out=c01),
        ChunkStage(lambda x: x.upper(), inp=c01, out=c12, late_finish=late_finish),
        ChunkStage(lambda x: f"wav({x})", inp=c12),
    ]
    orch = m.AsyncChunkOrchestrator(stages, {0, 2}, prewarm_fn=lambda sid, p: ("placeholder", sid))
    return stages, orch


async def collect(orch, n, timeout=5.0):
    per, done = {}, 0
    while done < n:
        o = await orch.next_output(timeout=timeout)
        per.setdefault(o.request_id, []).append(o)
        done += o.finished
    return per


def test_prewarm_and_no_forwarding():
    async def main():
        stages, orch = build()
        await orch.start()
        await orch.add_request("r", "p")
        per = await collect(orch, 1)
        outs = per["r"]
        assert [o.data for o in outs if o.stage_id == 0] == [f"r:t{i}" for i in range(6)]
        assert [o.data for o in outs if o.stage_id == 2] == [f"wav(R:T{i})" for i in range(6)]
        assert outs[-1].finished and sum(o.finished for o in outs) == 1
        # every stage submitted exactly once; downstream got the placeholder
        assert stages[0].submitted == [("r", "p")]
        assert stages[1].submitted == [("r", ("placeholder", 1))]
        assert stages[2].submitted == [("r", ("placeholder", 2))]
        await orch.shutdown()

    asyncio.run(main())


def test_terminal_waits_for_non_final_stage():
    async def main():
        stages, orch = build(late_finish=0.2)
        await orch.start()
        loop = asyncio.get_running_loop()
        t0 = loop.time()
        await orch.add_request("r", "p")
        per = await collect(orch, 1)
        elapsed = loop.time() - t0
        assert elapsed >= 0.18, "terminal was emitted before stage 1 reported finished"
        outs = per["r"]
        assert outs[-1].finished and outs[-1].stage_id == 2
        assert sum(o.finished for o in outs) == 1
        assert all(o.stage_id in (0, 2) for o in outs)
        assert orch.request_states == {}
        await orch.shutdown()

    asyncio.run(main())


def test_many_requests_with_late_finish():
    async def main():
        stages, orch = build(late_finish=0.01)
        await orch.start()
        rids = [f"r{i}" for i in range(20)]
        for rid in rids:
            await orch.add_request(rid, "p")
        per = await collect(orch, len(rids))
        for rid in rids:
            outs = per[rid]
            assert outs[-1].finished and sum(o.finished for o in outs) == 1
            assert [o.data for o in outs if o.stage_id == 2] == [f"wav({rid.upper()}:T{i})" for i in range(6)]
        await orch.shutdown()

    asyncio.run(main())


def test_late_output_after_cleanup_is_dropped():
    async def main():
        stages, orch = build()
```


© 2026 kzhou003

```
        await orch.start()
        await orch.add_request("r", "p")
        await collect(orch, 1)
        await stages[1].q.put([StageOutput("r", "zombie", True)])
        await asyncio.sleep(0.05)
        assert orch.dropped_outputs == 1
        assert "r" not in orch.request_states
        await orch.shutdown()

    asyncio.run(main())
```


© 2026 kzhou003

B-11 · tests/test_p01_output_sink.py

```
import asyncio
import random
import threading
import time

from drill import load

m = load("p01_output_sink")


def test_basic_fifo():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        s.put(1)
        s.put(2)
        assert await s.get_batch() == [1, 2]

    asyncio.run(main())


def test_wakeups_are_coalesced():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        t = threading.Thread(target=lambda: [s.put(i) for i in range(1000)])
        t.start()
        t.join()
        # nobody consumed yet -> exactly one wakeup was scheduled for 1000 puts
        assert s.wakeups_scheduled == 1
        got = []
        while len(got) < 1000:
            batch = await s.get_batch(max_items=256)
            assert 1 <= len(batch) <= 256
            got += batch
        assert got == list(range(1000))

    asyncio.run(main())


def test_timeout_returns_empty():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        t0 = time.monotonic()
        assert await s.get_batch(timeout=0.05) == []
        assert time.monotonic() - t0 < 1.0

    asyncio.run(main())


def test_multi_producer_per_producer_order():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        n, producers = 5000, 4

        def produce(pid):
            for i in range(n):
                s.put((pid, i))
                if i % 97 == 0:
                    time.sleep(random.random() * 0.001)

        threads = [threading.Thread(target=produce, args=(p,)) for p in range(producers)]
        for t in threads:
            t.start()
        got = []
        while len(got) < n * producers:
            batch = await s.get_batch(timeout=2.0)
            assert batch, "consumer starved -> lost wakeup?"
            got += batch
        for t in threads:
            t.join()
```


© 2026 kzhou003

```
        for p in range(producers):
            assert [i for q, i in got if q == p] == list(range(n))

    asyncio.run(main())


def test_ping_pong_never_loses_a_wakeup():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        go = threading.Event()

        def produce():
            for i in range(300):
                go.wait()
                go.clear()
                time.sleep(random.random() * 0.0005)
                s.put(i)

        t = threading.Thread(target=produce)
        t.start()
        for i in range(300):
            go.set()
            assert await s.get_batch(timeout=2.0) == [i]
        t.join()

    asyncio.run(main())


def test_put_between_empty_drain_and_rearm_is_not_lost():
    async def main():
        s = m.OutputSink()
        s.bind_loop()
        s.put("warm")  # leaves a wakeup "in flight"
        assert await s.get_batch() == ["warm"]
        await asyncio.sleep(0.01)  # let the stale wakeup callback run
        injected = []

        def racing_put():
            if not injected:
                injected.append(True)
                t = threading.Thread(target=s.put, args=("raced",))
                t.start()
                t.join()

        s.after_empty_drain = racing_put
        t0 = time.monotonic()
        assert await s.get_batch(timeout=0.5) == ["raced"]
        assert time.monotonic() - t0 < 0.2, "the item waited for the timeout: lost wakeup"

    asyncio.run(main())
```


© 2026 kzhou003

B-12 · tests/test_p04_stage_pool_abort.py

```
import asyncio

import pytest

from drill import load

m = load("p04_stage_pool_abort")


class Rep:
    def __init__(self, name, log, fail=False):
        self.name, self.log, self.fail = name, log, fail

    async def abort(self, ids):
        self.log.append(("abort", self.name, sorted(ids)))
        if self.fail:
            raise RuntimeError("engine refused abort")


class Proc:
    def __init__(self, log):
        self.log = log
        self.state = {"x": "partial-text"}

    def collect_partial(self, rid):
        self.log.append(("collect", rid))
        return self.state.get(rid)

    def commit_abort(self, rid):
        self.log.append(("commit", rid))
        self.state.pop(rid, None)


def test_sticky_round_robin():
    pool = m.StagePool(0, [Rep("a", []), Rep("b", []), Rep("c", [])])
    picks = [pool.pick(f"r{i}") for i in range(6)]
    assert picks == [0, 1, 2, 0, 1, 2]
    assert all(pool.pick(f"r{i}") == picks[i] for i in range(6)), "binding must be sticky"
    pool.release("r0")
    assert pool.bound("r0") is None


def test_evict_and_no_live():
    pool = m.StagePool(1, [Rep("a", []), Rep("b", [])])
    for i in range(4):
        pool.pick(f"r{i}")
    affected = pool.evict(0)
    assert sorted(affected) == ["r0", "r2"]
    assert pool.live_replicas() == [1]
    assert all(pool.pick(f"n{i}") == 1 for i in range(3))
    assert pool.pick("r1") == 1
    pool.evict(1)
    with pytest.raises(m.NoLiveReplica):
        pool.pick("z")


def test_abort_groups_by_replica():
    log = []
    pool = m.StagePool(0, [Rep("a", log), Rep("b", log)])
    for i in range(4):
        pool.pick(f"r{i}")

    asyncio.run(pool.abort(["r0", "r1", "r2", "r3", "unknown"]))
    assert sorted(log) == [("abort", "a", ["r0", "r2"]), ("abort", "b", ["r1", "r3"])]
    assert all(pool.bound(f"r{i}") is None for i in range(4))


def test_abort_two_phase_order():
    log = []
    pools = [m.StagePool(0, [Rep("s0", log)]), m.StagePool(1, [Rep("s1", log)])]
    pools[0].pick("x")
    pools[1].pick("x")
    proc = Proc(log)
```


© 2026 kzhou003

```
    out = asyncio.run(m.abort_request(pools, proc, "x"))
    assert log == [("collect", "x"), ("abort", "s0", ["x"]), ("abort", "s1", ["x"]), ("commit", "x")]
    assert out.finished and out.error == "abort" and out.data == "partial-text"


def test_abort_failure_does_not_commit():
    log = []
    pools = [m.StagePool(0, [Rep("s0", log, fail=True)])]
    pools[0].pick("x")
    proc = Proc(log)
    with pytest.raises(RuntimeError):
        asyncio.run(m.abort_request(pools, proc, "x"))
    assert ("commit", "x") not in log
    assert proc.state == {"x": "partial-text"}
    assert pools[0].bound("x") == 0, "binding kept so abort can be retried"


def test_dead_replica_only_fails_bound_requests():
    pool = m.StagePool(2, [Rep("a", []), Rep("b", [])])
    for i in range(4):
        pool.pick(f"r{i}")
    errs = m.handle_dead_replica(pool, 1)
    assert sorted(e.request_id for e in errs) == ["r1", "r3"]
    assert all(e.finished and e.error for e in errs)
    assert pool.bound("r0") == 0 and pool.bound("r2") == 0
```


© 2026 kzhou003

B-13 · tests/test_p05_rpc_router.py

```
import queue
import random
import threading
import time

import pytest

from drill import load

m = load("p05_rpc_router")


def test_concurrent_calls_out_of_order():
    router = m.RpcRouter()
    wire: queue.Queue = queue.Queue()

    def server():
        pending = []
        while True:
            item = wire.get()
            if item is None:
                break
            pending.append(item)
            if len(pending) >= 5 or wire.empty():
                random.shuffle(pending)
                for key in pending:
                    router.deliver(key, ("ok", key))
                pending.clear()

    srv = threading.Thread(target=server)
    srv.start()
    results = {}

    def client(i):
        key = ("rpc", i)
        results[i] = router.call(key, lambda: wire.put(key), timeout=2.0)

    threads = [threading.Thread(target=client, args=(i,)) for i in range(50)]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    wire.put(None)
    srv.join()
    assert results == {i: ("ok", ("rpc", i)) for i in range(50)}


def test_result_delivered_inside_send_is_not_lost():
    router = m.RpcRouter()
    assert router.call("k", lambda: router.deliver("k", 42), timeout=0.5) == 42


def test_timeout_and_late_result():
    router = m.RpcRouter()
    with pytest.raises(TimeoutError):
        router.call("slow", lambda: None, timeout=0.05)
    assert router.deliver("slow", "too late") is False
    assert router.late_results == 1
    # the key can be reused after the timed-out call cleaned up
    assert router.call("slow", lambda: router.deliver("slow", 1), timeout=0.5) == 1


def test_duplicate_key_rejected():
    router = m.RpcRouter()
    w = router.register("dup")
    with pytest.raises(ValueError):
        router.register("dup")
    router.unregister("dup", w)


def test_fail_all_wakes_waiters_and_latches():
    router = m.RpcRouter()
    errors = []
```


© 2026 kzhou003

```
    def client(i):
        try:
            router.call(i, lambda: None, timeout=5.0)
        except Exception as e:  # noqa: BLE001
            errors.append(e)

    threads = [threading.Thread(target=client, args=(i,)) for i in range(5)]
    for t in threads:
        t.start()
    time.sleep(0.1)
    t0 = time.monotonic()
    router.fail_all(RuntimeError("orchestrator died"))
    for t in threads:
        t.join()
    assert time.monotonic() - t0 < 1.0
    assert len(errors) == 5 and all("orchestrator died" in str(e) for e in errors)
    with pytest.raises(RuntimeError):
        router.call("new", lambda: None, timeout=5.0)
```


© 2026 kzhou003

B-14 · tests/test_p10_engine_core_loop.py

```
import threading
import time

from drill import load

m = load("p10_engine_core_loop")
Msg = m.Msg


class FakeEngine:
    def __init__(self, steps_per_req=3, fail_on_step=None):
        self.reqs: dict[str, int] = {}
        self.calls: list = []
        self.steps = 0
        self.has_work_calls = 0
        self.steps_per_req = steps_per_req
        self.fail_on_step = fail_on_step

    def add(self, rid, data):
        self.calls.append(("add", rid))
        self.reqs[rid] = self.steps_per_req

    def abort(self, ids):
        self.calls.append(("abort", tuple(ids)))
        for r in ids:
            self.reqs.pop(r, None)

    def has_work(self):
        self.has_work_calls += 1
        return bool(self.reqs)

    def step(self):
        self.steps += 1
        if self.fail_on_step == self.steps:
            raise RuntimeError("CUDA error: illegal memory access")
        outs = []
        for r in list(self.reqs):
            self.reqs[r] -= 1
            outs.append((r, self.reqs[r] == 0))
            if self.reqs[r] == 0:
                del self.reqs[r]
        return outs

    def utility(self, name, args):
        if name == "boom":
            raise ValueError("bad utility")
        return (name, args)


def start(engine):
    emitted = []
    loop = m.EngineCoreLoop(engine, emitted.append)
    t = threading.Thread(target=loop.run_busy_loop, daemon=True)
    return loop, t, emitted


def wait_for(pred, timeout=2.0):
    end = time.monotonic() + timeout
    while time.monotonic() < end:
        if pred():
            return True
        time.sleep(0.005)
    return False


def test_idle_loop_blocks_instead_of_spinning():
    eng = FakeEngine()
    loop, t, _ = start(eng)
    t.start()
    time.sleep(0.3)
    assert eng.steps == 0
    assert eng.has_work_calls < 10, f"busy-spinning: has_work called {eng.has_work_calls} times"
    loop.request_shutdown()
```


© 2026 kzhou003

```
    t.join(1.0)
    assert not t.is_alive()


def test_add_runs_to_completion():
    eng = FakeEngine(steps_per_req=3)
    loop, t, emitted = start(eng)
    t.start()
    loop.input_queue.put(Msg("ADD", ("r1", None)))
    assert wait_for(lambda: ("output", ("r1", True)) in emitted)
    assert [e for e in emitted if e[0] == "output"] == [("output", ("r1", False))] * 2 + [("output", ("r1",
True))]
    loop.request_shutdown()
    t.join(1.0)
    assert emitted[-1] == ("shutdown",)


def test_add_then_abort_are_applied_in_fifo_order_before_stepping():
    eng = FakeEngine()
    loop, t, emitted = start(eng)
    loop.input_queue.put(Msg("ADD", ("r1", None)))
    loop.input_queue.put(Msg("ABORT", ["r1"]))
    loop.input_queue.put(Msg("ADD", ("r2", None)))
    t.start()
    assert wait_for(lambda: ("output", ("r2", True)) in emitted)
    assert eng.calls[:3] == [("add", "r1"), ("abort", ("r1",)), ("add", "r2")]
    assert not any(e[0] == "output" and e[1][0] == "r1" for e in emitted)
    loop.request_shutdown()
    t.join(1.0)


def test_shutdown_from_another_thread_while_idle_is_fast():
    eng = FakeEngine()
    loop, t, emitted = start(eng)
    t.start()
    time.sleep(0.05)
    assert t.is_alive()
    t0 = time.monotonic()
    threading.Thread(target=loop.request_shutdown).start()
    t.join(1.0)
    assert not t.is_alive() and time.monotonic() - t0 < 0.5
    assert emitted == [("shutdown",)]


def test_step_exception_is_fatal():
    eng = FakeEngine(steps_per_req=10, fail_on_step=2)
    loop, t, emitted = start(eng)
    t.start()
    loop.input_queue.put(Msg("ADD", ("r1", None)))
    t.join(2.0)
    assert not t.is_alive()
    assert emitted[-1][0] == "fatal" and "illegal memory access" in emitted[-1][1]


def test_utility_results_and_errors():
    eng = FakeEngine()
    loop, t, emitted = start(eng)
    t.start()
    loop.input_queue.put(Msg("UTILITY", (1, "sleep", (2,))))
    loop.input_queue.put(Msg("UTILITY", (2, "boom", ())))
    assert wait_for(lambda: len([e for e in emitted if e[0] == "utility"]) == 2)
    utils = [e for e in emitted if e[0] == "utility"]
    assert utils[0] == ("utility", 1, ("sleep", (2,)), None)
    assert utils[1][1] == 2 and utils[1][2] is None and "bad utility" in utils[1][3]
    assert t.is_alive(), "a failing utility call must not kill the engine"
    loop.request_shutdown()
    t.join(1.0)
```


© 2026 kzhou003

B-15 · tests/test_p11_chunk_aware_scheduler.py

```
import pytest

from drill import load

m = load("p11_chunk_aware_scheduler")


class Source:
    def __init__(self):
        self.ready: set[str] = set()
        self.loads: list[str] = []

    def load_async(self, rid):
        self.loads.append(rid)

    def is_ready(self, rid):
        return rid in self.ready


def make():
    src = Source()
    return m.ChunkAwareScheduler(m.BaseScheduler(), src), src


def test_not_ready_is_never_scheduled_and_loaded_once():
    s, src = make()
    s.add(m.Req("plain", 0))
    s.add(m.Req("c1", 1, needs_chunk=True))
    for _ in range(3):
        assert "c1" not in s.schedule()
    assert src.loads == ["c1"]
    assert "c1" in s.parked and s.parked["c1"][0].status == "WAITING_FOR_CHUNK"


def test_ready_request_is_scheduled_on_next_call():
    s, src = make()
    s.add(m.Req("c1", 0, needs_chunk=True))
    assert s.schedule() == []
    src.ready.add("c1")
    assert s.schedule() == ["c1"]
    assert not s.parked


def test_running_request_waiting_for_next_chunk():
    s, src = make()
    src.ready.add("c1")
    s.add(m.Req("c1", 0, needs_chunk=True))
    assert s.schedule() == ["c1"]
    src.ready.discard("c1")  # consumed chunk k, chunk k+1 not here yet
    assert s.schedule() == []
    src.ready.add("c1")
    assert s.schedule() == ["c1"]
    assert s.base.running[0].status == "RUNNING"


class Boom(Exception):
    pass


def test_restore_happens_even_if_upstream_raises():
    src = Source()

    class Exploding(m.BaseScheduler):
        def schedule(self):
            src.ready.add("c1")  # the chunk lands while upstream is scheduling...
            raise Boom("upstream scheduler blew up")  # ...and upstream fails

    s = m.ChunkAwareScheduler(Exploding(), src)
    s.add(m.Req("c1", 0, needs_chunk=True))
    with pytest.raises(Boom):
        s.schedule()
    assert src.loads == ["c1"], "c1 must have been parked before upstream ran"
    assert "c1" not in s.parked, "the finally-restore must run even when upstream raises"
```


© 2026 kzhou003

```
    assert [r.req_id for r in s.base.waiting] == ["c1"]


def test_abort_while_parked():
    s, src = make()
    s.add(m.Req("c1", 0, needs_chunk=True))
    s.schedule()
    s.abort("c1")
    src.ready.add("c1")
    assert s.schedule() == []
    assert "c1" not in s.parked


def test_restored_in_arrival_order():
    s, src = make()
    for i, rid in enumerate(["a", "b", "c"]):
        s.add(m.Req(rid, i, needs_chunk=True))
    s.schedule()
    src.ready.update({"c", "b", "a"})
    assert s.schedule() == ["a", "b", "c"]
```


© 2026 kzhou003

B-16 · tests/test_p07_scheduler.py

```
from drill import load

m = load("p07_scheduler")
EOS = -1


def tok(seq_id, k):
    """Deterministic 'model': k-th output token of a sequence."""
    return 1000 * int(seq_id[1:]) + k


def run_step(s):
    sched = s.schedule()
    by_id = {q.seq_id: q for q in s.running}
    s.update(lambda sid: tok(sid, len(by_id[sid].output)), EOS)
    return sched


def test_token_budget_and_chunked_prefill():
    s = m.Scheduler(num_blocks=100, block_size=4, max_num_seqs=8, max_num_batched_tokens=16)
    for i in range(3):
        s.add(m.Seq(f"s{i}", prompt_len=10, max_tokens=5))
    assert run_step(s) == {"s0": 10, "s1": 6}
    assert run_step(s) == {"s0": 1, "s1": 4, "s2": 10}
    assert run_step(s) == {"s0": 1, "s1": 1, "s2": 1}


def test_max_num_seqs():
    s = m.Scheduler(num_blocks=100, block_size=4, max_num_seqs=2, max_num_batched_tokens=100)
    for i in range(4):
        s.add(m.Seq(f"s{i}", prompt_len=3, max_tokens=2))
    assert set(run_step(s)) == {"s0", "s1"}
    assert set(run_step(s)) == {"s0", "s1"}  # s0/s1 finish here (2 tokens)
    assert set(run_step(s)) == {"s2", "s3"}


def test_fcfs_head_of_line():
    s = m.Scheduler(num_blocks=4, block_size=4, max_num_seqs=8, max_num_batched_tokens=100)
    s.add(m.Seq("s0", prompt_len=12, max_tokens=50))  # 3 blocks
    s.add(m.Seq("s1", prompt_len=8, max_tokens=1))    # 2 blocks -> does not fit
    s.add(m.Seq("s2", prompt_len=2, max_tokens=1))    # would fit, but FCFS forbids skipping
    assert run_step(s) == {"s0": 12}


def test_preemption_eventually_finishes_everything_correctly():
    s = m.Scheduler(num_blocks=6, block_size=4, max_num_seqs=8, max_num_batched_tokens=64)
    seqs = [m.Seq(f"s{i}", prompt_len=5, max_tokens=8) for i in range(4)]
    for q in seqs:
        s.add(q)
    steps = 0
    while s.has_work():
        run_step(s)
        assert s.blocks_used <= s.num_blocks
        steps += 1
        assert steps < 500, "no progress"
    assert s.num_preemptions > 0
    assert s.blocks_used == 0
    for q in seqs:
        assert q.status == "FINISHED"
        assert q.output == [tok(q.seq_id, k) for k in range(8)]


def test_eos_stops_early():
    s = m.Scheduler(num_blocks=10, block_size=4, max_num_seqs=8, max_num_batched_tokens=64)
    s.add(m.Seq("s1", prompt_len=3, max_tokens=100))
    s.schedule()
    s.update(lambda sid: EOS, EOS)
    assert not s.has_work() and s.finished[0].output == [EOS] and s.blocks_used == 0
```


© 2026 kzhou003

B-17 · tests/test_p06_block_manager.py

```
import pytest

from drill import load

m = load("p06_block_manager")


def test_allocate_and_slots():
    bm = m.BlockManager(num_blocks=8, block_size=4)
    table = bm.allocate("a", list(range(10)))
    assert len(table) == 3 and len(set(table)) == 3
    assert bm.num_free_blocks == 5
    for pos in range(10):
        assert bm.slot("a", pos) == table[pos // 4] * 4 + pos % 4


def test_append_crosses_block_boundary():
    bm = m.BlockManager(num_blocks=3, block_size=4)
    bm.allocate("a", list(range(7)))  # 2 blocks, 1 free slot in the 2nd
    s7 = bm.append_slot("a", 7)
    assert bm.num_free_blocks == 1
    s8 = bm.append_slot("a", 8)  # needs a 3rd block
    assert bm.num_free_blocks == 0
    t = bm.block_table("a")
    assert s7 == t[1] * 4 + 3 and s8 == t[2] * 4 + 0


def test_out_of_blocks_is_atomic():
    bm = m.BlockManager(num_blocks=2, block_size=4)
    with pytest.raises(m.OutOfBlocks):
        bm.allocate("big", list(range(9)))
    assert bm.num_free_blocks == 2
    bm.allocate("a", list(range(8)))
    with pytest.raises(m.OutOfBlocks):
        bm.append_slot("a", 8)
    assert bm.block_table("a") and len(bm.block_table("a")) == 2
    bm.free("a")
    assert bm.num_free_blocks == 2


def test_free_returns_everything():
    bm = m.BlockManager(num_blocks=6, block_size=2)
    bm.allocate("a", [1, 2, 3])
    bm.allocate("b", [4, 5, 6, 7])
    bm.free("a")
    bm.free("b")
    assert bm.num_free_blocks == 6


def test_prefix_sharing_and_refcount():
    bm = m.BlockManager(num_blocks=10, block_size=4, enable_prefix_cache=True)
    prefix = list(range(8))
    ta = bm.allocate("a", prefix + [100, 101])
    tb = bm.allocate("b", prefix + [200])
    assert bm.num_cached_tokens("b") == 8
    assert tb[:2] == ta[:2] and tb[2] != ta[2]
    assert bm.num_free_blocks == 10 - 4
    bm.free("a")
    assert bm.num_free_blocks == 10 - 3, "shared blocks are still referenced by b"


def test_cache_survives_free_and_last_token_is_never_cached():
    bm = m.BlockManager(num_blocks=10, block_size=4, enable_prefix_cache=True)
    bm.allocate("a", list(range(8)) + [9])
    bm.free("a")
    bm.allocate("c", list(range(8)) + [42])
    assert bm.num_cached_tokens("c") == 8
    bm.free("c")
    # exactly 8 tokens: only the first block may be reused, the last token must be computed
    bm.allocate("d", list(range(8)))
    assert bm.num_cached_tokens("d") == 4
```


© 2026 kzhou003

```
def test_eviction_drops_cached_blocks():
    bm = m.BlockManager(num_blocks=4, block_size=4, enable_prefix_cache=True)
    bm.allocate("a", list(range(9)))
    bm.free("a")
    bm.allocate("filler", list(range(1000, 1016)))  # needs all 4 blocks -> evicts
    bm.free("filler")
    bm.allocate("b", list(range(9)))
    assert bm.num_cached_tokens("b") == 0


def test_append_completing_a_block_registers_it():
    bm = m.BlockManager(num_blocks=8, block_size=4, enable_prefix_cache=True)
    bm.allocate("e", [5, 6, 7])
    bm.append_slot("e", 8)  # block [5,6,7,8] is now full
    bm.free("e")
    bm.allocate("f", [5, 6, 7, 8, 9])
    assert bm.num_cached_tokens("f") == 4
```


© 2026 kzhou003

B-18 · tests/test_p08_sampler.py

```
import math
import random

import torch

from drill import load

m = load("p08_sampler")
SP = m.SamplingParams
V = 50


def rand_logits(seed, n=1):
    g = torch.Generator().manual_seed(seed)
    return torch.randn(n, V, generator=g) * 3


def test_greedy():
    s = m.Sampler()
    logits = rand_logits(0, 4)
    out = s.sample(logits, list("abcd"), [SP(temperature=0.0)] * 4, [[]] * 4)
    assert out.tolist() == logits.argmax(-1).tolist()


def test_top_k_1_and_tiny_top_p_are_greedy():
    s = m.Sampler()
    logits = rand_logits(1, 3)
    greedy = logits.argmax(-1).tolist()
    assert s.sample(logits, list("abc"), [SP(top_k=1, seed=1)] * 3, [[]] * 3).tolist() == greedy
    assert s.sample(logits, list("xyz"), [SP(top_p=1e-6, seed=2)] * 3, [[]] * 3).tolist() == greedy


def test_seeded_result_is_batch_invariant():
    row = rand_logits(7)[0]
    alone = m.Sampler()
    ref = [int(alone.sample(row[None], ["a"], [SP(temperature=1.0, seed=123)], [[]])[0]) for _ in range(30)]

    mixed = m.Sampler()
    got = []
    for step in range(30):
        others = random.randint(0, 5)
        ids = [f"o{step}_{j}" for j in range(others)] + ["a"]
        random.shuffle(ids)
        logits = torch.stack([row if r == "a" else rand_logits(step * 10 + j)[0] for j, r in enumerate(ids)])
        params = [SP(temperature=1.0, seed=123) if r == "a" else SP(temperature=1.0) for r in ids]
        out = mixed.sample(logits, ids, params, [[] for _ in ids])
        got.append(int(out[ids.index("a")]))
    assert got == ref


def test_top_k_distribution():
    s = m.Sampler()
    logits = torch.full((1, V), -1e9)
    logits[0, 3], logits[0, 9] = math.log(0.7), math.log(0.3)
    logits[0, 20] = math.log(0.2)  # removed by top_k=2
    counts = {3: 0, 9: 0}
    for _ in range(4000):
        t = int(s.sample(logits, ["r"], [SP(top_k=2, seed=5)], [[]])[0])
        assert t in counts
        counts[t] += 1
    frac = counts[3] / 4000
    assert abs(frac - 0.7) < 0.04, frac


def test_repetition_penalty_values_and_effect():
    logits = torch.tensor([2.0, -2.0, 1.5])
    pen = m.apply_repetition_penalty(logits, [0, 1, 1], 2.0)
    assert torch.allclose(pen, torch.tensor([1.0, -4.0, 1.5]))
    s = m.Sampler()
    out = s.sample(logits[None], ["r"], [SP(temperature=0.0, repetition_penalty=2.0)], [[0]])
    assert int(out[0]) == 2
```


© 2026 kzhou003

```
def test_free_resets_generator():
    s = m.Sampler()
    row = rand_logits(3)
    first = [int(s.sample(row, ["r"], [SP(seed=9)], [[]])[0]) for _ in range(10)]
    s.free("r")
    again = [int(s.sample(row, ["r"], [SP(seed=9)], [[]])[0]) for _ in range(10)]
    assert first == again
```


© 2026 kzhou003

B-19 · tests/test_p09_paged_attention.py

```
import random

import torch

from drill import load

m = load("p09_paged_attention")


def dense_attention(q, k, v, scale):
    # q [H,D], k/v [n,KVH,D]
    group = q.shape[0] // k.shape[1]
    k = k.repeat_interleave(group, 1)
    v = v.repeat_interleave(group, 1)
    p = torch.softmax(torch.einsum("hd,nhd->hn", q, k) * scale, -1)
    return torch.einsum("hn,nhd->hd", p, v)


def build_case(seed, lens, block_size=4, num_blocks=64, H=8, KVH=2, D=16):
    torch.manual_seed(seed)
    random.seed(seed)
    k_cache = torch.zeros(num_blocks, block_size, KVH, D)
    v_cache = torch.zeros_like(k_cache)
    free = list(range(num_blocks))
    random.shuffle(free)  # non-contiguous physical blocks
    max_blocks = max(-(-n // block_size) for n in lens)
    tables = torch.full((len(lens), max_blocks), 999, dtype=torch.int32)  # junk in unused slots
    dense = []
    for b, n in enumerate(lens):
        nb = -(-n // block_size)
        blocks = [free.pop() for _ in range(nb)]
        tables[b, :nb] = torch.tensor(blocks, dtype=torch.int32)
        k, v = torch.randn(n, KVH, D), torch.randn(n, KVH, D)
        slots = torch.tensor([blocks[p // block_size] * block_size + p % block_size for p in range(n)])
        m.write_kv(k_cache, v_cache, k, v, slots)
        dense.append((k, v))
    q = torch.randn(len(lens), H, D)
    return q, k_cache, v_cache, tables, torch.tensor(lens), dense


def test_matches_dense_attention():
    lens = [1, 4, 5, 13, 32, 7]
    q, kc, vc, tables, ctx, dense = build_case(0, lens)
    scale = 16 ** -0.5
    out = m.paged_decode_attention(q, kc, vc, tables, ctx, scale)
    for b, (k, v) in enumerate(dense):
        torch.testing.assert_close(out[b], dense_attention(q[b], k, v, scale), atol=1e-5, rtol=1e-5)


def test_padding_slots_are_not_written():
    kc = torch.zeros(4, 4, 1, 2)
    vc = torch.zeros_like(kc)
    k = torch.ones(3, 1, 2)
    m.write_kv(kc, vc, k, k, torch.tensor([5, -1, 6]))
    assert kc.sum() == 4  # two tokens x 2 dims
    assert kc[1, 1].sum() == 2 and kc[1, 2].sum() == 2


def test_random_shapes():
    for seed in range(5):
        lens = [random.randint(1, 40) for _ in range(random.randint(1, 6))]
        q, kc, vc, tables, ctx, dense = build_case(seed + 10, lens, block_size=random.choice([2, 4, 8]))
        out = m.paged_decode_attention(q, kc, vc, tables, ctx, 0.25)
        for b, (k, v) in enumerate(dense):
            torch.testing.assert_close(out[b], dense_attention(q[b], k, v, 0.25), atol=1e-5, rtol=1e-5)
```


© 2026 kzhou003

B-20 · tests/test_p12_shm_connector.py

```
import fcntl
import os
import subprocess
import sys
import time
import uuid

from drill import load

m = load("p12_shm_connector")


def _uniq(tag):
    return f"{tag}-{uuid.uuid4().hex[:8]}"


def test_roundtrip_and_consume_once():
    c = m.ShmConnector(namespace="t1")
    k = _uniq("req_0_0")
    c.put(k, b"hello")
    assert c.get(k) == b"hello"
    assert c.get(k) is None, "the reader consumes (unlinks) the segment"
    assert c.get(_uniq("missing")) is None


def test_empty_and_large_payloads():
    c = m.ShmConnector(namespace="t2")
    k1, k2 = _uniq("e"), _uniq("big")
    c.put(k1, b"")
    assert c.get(k1) == b""
    blob = os.urandom(4 * 1024 * 1024 + 3)
    c.put(k2, blob)
    assert c.get(k2) == blob


def test_independent_writer_process_exits_before_read():
    """Writer is a separate interpreter (its own resource tracker) that exits first."""
    ns = "t3"
    keys = [_uniq(f"r_0_{i}") for i in range(3)]
    code = (
        "from drill import load; c = load('p12_shm_connector').ShmConnector(namespace=%r)\n"
        "for k in %r: c.put(k, ('payload-' + k).encode())" % (ns, keys)
    )
    root = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    subprocess.run([sys.executable, "-c", code], check=True, cwd=root, env={**os.environ, "PYTHONPATH": root})
    time.sleep(1.0)  # give the writer's resource tracker time to "clean up"
    c = m.ShmConnector(namespace=ns)
    for k in keys:
        assert c.get(k) == f"payload-{k}".encode(), "segment vanished when the writer exited (resource tracker?)"


def test_not_ready_while_lock_is_held():
    c = m.ShmConnector(namespace="t4")
    k = _uniq("locked")
    c.put(k, b"data")
    fd = os.open(c.lock_path(k), os.O_RDWR)
    try:
        fcntl.flock(fd, fcntl.LOCK_EX)  # pretend a writer is still writing
        assert c.get(k) is None
    finally:
        fcntl.flock(fd, fcntl.LOCK_UN)
        os.close(fd)
    assert c.get(k) == b"data"


def test_cleanup_prefix_only_touches_unread_keys_with_prefix():
    c = m.ShmConnector(namespace="t5")
    a = _uniq("reqA")
    b = _uniq("reqB")
    c.put(f"{a}_0_0", b"1")
    c.put(f"{a}_0_1", b"2")
    c.put(f"{b}_0_0", b"3")
    assert c.get(f"{a}_0_0") == b"1"  # consumed already
```


© 2026 kzhou003

```
    assert c.cleanup_prefix(f"{a}_") == 1
    assert c.get(f"{a}_0_1") is None
    assert c.get(f"{b}_0_0") == b"3"
```


© 2026 kzhou003

B-21 · tests/test_p13_chunk_adapter.py

```
import random
import threading
import time

from drill import load

m = load("p13_chunk_adapter")


def pair(build=lambda snap: list(snap), **conn_kw):
    conn = m.InMemoryConnector(**conn_kw)
    sender = m.ChunkAdapter(conn, stage_id=0, src_stage_id=None, build_payload=build)
    receiver = m.ChunkAdapter(conn, stage_id=1, src_stage_id=0, build_payload=build)
    sender.start()
    receiver.start()
    return conn, sender, receiver


def drain_until_finished(receiver, req_ids, timeout=5.0):
    got = {r: [] for r in req_ids}
    done = set()
    end = time.monotonic() + timeout
    while done != set(req_ids):
        assert time.monotonic() < end, f"timed out; done={done}"
        for rid, chunks in receiver.poll_ready().items():
            for c in chunks:
                assert rid not in done, "chunk after the final one"
                got[rid].append(c)
                if c["finished"]:
                    done.add(rid)
        time.sleep(0.001)
    return got


def stop(*adapters):
    for a in adapters:
        a.stop()


def test_ordered_delivery_single_request():
    conn, s, r = pair()
    r.load_async("q")
    for i in range(50):
        s.save_async("q", [i], is_final=(i == 49))
    got = drain_until_finished(r, ["q"])
    assert [c["data"] for c in got["q"]] == [[i] for i in range(50)]
    assert conn.puts == [f"q_0_{i}" for i in range(50)]
    stop(s, r)


def test_many_requests_with_random_latency():
    jitter = lambda key: time.sleep(random.random() * 0.0005)  # noqa: E731
    conn, s, r = pair(on_put=jitter, on_get=jitter)
    rids = [f"r{i}" for i in range(10)]
    for rid in rids:
        r.load_async(rid)
    for step in range(30):
        for rid in random.sample(rids, len(rids)):
            s.save_async(rid, [rid, step], is_final=(step == 29))
    got = drain_until_finished(r, rids)
    for rid in rids:
        assert [c["data"] for c in got[rid]] == [[rid, k] for k in range(30)]
    stop(s, r)


def test_save_async_never_blocks_the_caller():
    def slow_build(snap):
        time.sleep(0.05)
        return snap

    conn, s, r = pair(build=slow_build)
    t0 = time.monotonic()
    for i in range(10):
```


© 2026 kzhou003

```
        s.save_async("q", [i], is_final=(i == 9))
    assert time.monotonic() - t0 < 0.02
    r.load_async("q")
    drain_until_finished(r, ["q"])
    stop(s, r)


def test_snapshot_isolation():
    conn, s, r = pair()
    live = [1, 2, 3]
    r.load_async("q")
    s.save_async("q", live, is_final=True)
    live.append(4)  # scheduler keeps mutating the live request
    got = drain_until_finished(r, ["q"])
    assert got["q"][0]["data"] == [1, 2, 3]
    stop(s, r)


def test_cancel_still_delivers_final_and_ids_stay_contiguous():
    gate = threading.Event()

    def gated_build(snap):
        gate.wait()
        return snap

    conn, s, r = pair(build=gated_build)
    r.load_async("q")
    s.save_async("q", ["first"], is_final=False)
    time.sleep(0.05)  # save thread is now blocked building "first"
    for i in range(20):
        s.save_async("q", [i], is_final=False)
    s.save_async("q", ["end"], is_final=True)
    s.cancel_sender("q")
    gate.set()
    got = drain_until_finished(r, ["q"])
    assert got["q"][-1]["finished"], "the final chunk is the downstream EOS: it must survive cancel"
    assert len(got["q"]) <= 2
    assert conn.puts == [f"q_0_{i}" for i in range(len(conn.puts))], "chunk ids must be contiguous"
    stop(s, r)


def test_none_payload_is_skipped_without_gap():
    conn, s, r = pair(build=lambda snap: None if snap[0] % 2 else snap)
    r.load_async("q")
    for i in range(6):
        s.save_async("q", [i], is_final=(i == 5))
    got = drain_until_finished(r, ["q"])
    assert [c["data"] for c in got["q"]] == [[0], [2], [4], None]
    assert conn.puts == ["q_0_0", "q_0_1", "q_0_2", "q_0_3"]
    stop(s, r)


def test_cleanup_is_a_commit_barrier_for_in_flight_reads():
    in_get = threading.Event()

    def slow_get(key):
        in_get.set()
        time.sleep(0.2)

    conn, s, r = pair(on_get=slow_get)
    conn.put("q_0_0", {"data": "late", "finished": False})
    r.load_async("q")
    assert in_get.wait(1.0)
    r.cleanup_receiver("q")  # request aborted while the recv thread is inside get()
    time.sleep(0.3)
    assert r.poll_ready() == {}, "a chunk read before cleanup must not be committed after it"
    stop(s, r)
```


© 2026 kzhou003

B-22 · tests/test_p15_talker_inputs.py

```
import pytest
import torch

from drill import load

m = load("p15_talker_inputs")
D = 4


def rows(n, base=0):
    return torch.arange(base, base + n, dtype=torch.float32)[:, None].repeat(1, D)


def test_stop_finish_drops_one_row_and_id():
    p = m.trim_thinker_payload([1, 2, 3], [7, 8, 151645], rows(7), "stop")  # one extra row captured
    assert p.prefill.shape == (3, D) and p.decode.shape == (2, D)
    assert p.output_ids == [7, 8]
    assert torch.equal(p.decode[:, 0], torch.tensor([3.0, 4.0]))


def test_length_finish_keeps_everything():
    p = m.trim_thinker_payload([1, 2], [5, 6, 7], rows(5), "length")
    assert p.decode.shape == (3, D) and p.output_ids == [5, 6, 7]


def test_under_captured_prefill_raises():
    with pytest.raises(ValueError):
        m.trim_thinker_payload([1, 2, 3], [4], rows(2), "length")


def test_build_prefill_layout():
    P = 3
    prompt = rows(P)
    first = torch.full((D,), 100.0)
    spk, pad, bos = torch.full((D,), 1.0), torch.full((D,), 10.0), torch.full((D,), 1000.0)
    ids, emb = m.build_talker_prefill(prompt, first, spk, pad, bos, mask_id=8296, pad_id=8292, start_id=8293)
    assert ids == [8296, 8296, 8296, 8292, 8293]
    assert emb.shape == (P + 2, D)
    assert torch.equal(emb[:P], prompt)
    assert torch.equal(emb[P], spk + pad) and torch.equal(emb[P + 1], first + bos)


def test_decode_consumes_one_row_then_falls_back():
    q = rows(2, base=50)
    last = torch.full((D,), 0.5)
    fb = torch.full((D,), -1.0)
    x, q = m.talker_decode_input(q, last, fb)
    assert torch.equal(x, torch.full((D,), 50.5)) and q.shape[0] == 1
    x, q = m.talker_decode_input(q, last, fb)
    assert torch.equal(x, torch.full((D,), 51.5)) and q.shape[0] == 0
    x, q = m.talker_decode_input(q, last, fb)
    assert torch.equal(x, torch.full((D,), -0.5)) and q.shape[0] == 0
```


© 2026 kzhou003

B-23 · tests/test_p14_codec_windows.py

```
import random

from drill import load

m = load("p14_codec_windows")
UP = 3


def toy_vocoder(codes, receptive_field):
    """Causal and local: sample j of frame i depends on codes[i-rf .. i] *within the given window*."""
    out = []
    for i in range(len(codes)):
        ctx = codes[max(0, i - receptive_field):i + 1]
        out += [sum(ctx) * 10 + j for j in range(UP)]
    return out


def stream(codes, arrivals, *, initial, chunk, left_ctx, rf):
    """Simulate the talker producing frames in bursts and the window planner reacting."""
    emitted, produced, audio, windows = 0, 0, [], []
    for k, burst in enumerate(arrivals):
        produced += burst
        finished = k == len(arrivals) - 1
        while True:
            w = m.next_window(produced, emitted, finished, initial=initial, chunk=chunk, left_ctx=left_ctx)
            if w is None:
                break
            ctx_start, new_start, new_end = w
            decoded = toy_vocoder(codes[ctx_start:new_end], rf)
            audio += m.slice_new_samples(decoded, w, UP)
            windows.append(w)
            emitted = new_end
    return audio, windows


def test_initial_then_steady_then_flush():
    assert m.next_window(3, 0, False, initial=4, chunk=25, left_ctx=25) is None
    assert m.next_window(4, 0, False, initial=4, chunk=25, left_ctx=25) == (0, 0, 4)
    assert m.next_window(20, 4, False, initial=4, chunk=25, left_ctx=25) is None
    assert m.next_window(29, 4, False, initial=4, chunk=25, left_ctx=25) == (0, 4, 29)
    assert m.next_window(60, 29, False, initial=4, chunk=25, left_ctx=25) == (4, 29, 54)
    assert m.next_window(60, 54, True, initial=4, chunk=25, left_ctx=25) == (29, 54, 60)
    assert m.next_window(60, 60, True, initial=4, chunk=25, left_ctx=25) is None


def test_no_initial_chunk():
    assert m.next_window(10, 0, False, initial=0, chunk=10, left_ctx=2) == (0, 0, 10)
    assert m.next_window(20, 10, False, initial=0, chunk=10, left_ctx=2) == (8, 10, 20)


def test_windows_tile_the_stream_exactly():
    for seed in range(20):
        random.seed(seed)
        total = random.randint(1, 200)
        arrivals, left = [], total
        while left:
            b = random.randint(1, 7)
            arrivals.append(min(b, left))
            left -= arrivals[-1]
        codes = [random.randint(0, 9) for _ in range(total)]
        _, windows = stream(codes, arrivals, initial=4, chunk=13, left_ctx=5, rf=5)
        covered = [f for (_, s, e) in windows for f in range(s, e)]
        assert covered == list(range(total)), "gap or overlap in shipped frames"


def test_streaming_equals_full_decode_iff_left_context_covers_receptive_field():
    random.seed(0)
    codes = [random.randint(0, 9) for _ in range(97)]
    arrivals = [1] * 97
    full = toy_vocoder(codes, receptive_field=4)
    ok, _ = stream(codes, arrivals, initial=4, chunk=10, left_ctx=4, rf=4)
    assert ok == full
```


© 2026 kzhou003

```
    bad, _ = stream(codes, arrivals, initial=4, chunk=10, left_ctx=2, rf=4)
    assert bad != full, "too little left context must produce seams"
```

B-24 · tests/test_p18_mrope_positions.py

```
import torch

from drill import load

m = load("p18_mrope_positions")


def test_text_only_is_plain_rope():
    pos, delta = m.mrope_positions([("text", 5)])
    assert pos.tolist() == [[0, 1, 2, 3, 4]] * 3
    assert delta == 0
    assert m.decode_position(5, delta).tolist() == [5, 5, 5]


def test_text_image_text():
    pos, delta = m.mrope_positions([("text", 2), ("image", (1, 4, 4)), ("text", 3)])
    assert pos.tolist() == [
        [0, 1, 2, 2, 2, 2, 4, 5, 6],
        [0, 1, 2, 2, 3, 3, 4, 5, 6],
        [0, 1, 2, 3, 2, 3, 4, 5, 6],
    ]
    assert delta == 7 - 9
    assert m.decode_position(9, delta).tolist() == [7, 7, 7]


def test_multi_frame_rectangular_grid():
    pos, delta = m.mrope_positions([("image", (2, 2, 6)), ("text", 1)], spatial_merge=2)
    # llm grid t=2, h=1, w=3 -> 6 tokens
    assert pos[:, :6].tolist() == [
        [0, 0, 0, 1, 1, 1],
        [0, 0, 0, 0, 0, 0],
        [0, 1, 2, 0, 1, 2],
    ]
    assert pos[:, 6].tolist() == [3, 3, 3]
    assert delta == 4 - 7


def test_shapes_and_dtype():
    pos, _ = m.mrope_positions([("text", 3), ("image", (1, 8, 8)), ("text", 2), ("image", (1, 2, 2))])
    assert pos.shape == (3, 3 + 16 + 2 + 1)
    assert pos.dtype == torch.long
```


© 2026 kzhou003

B-25 · tests/test_p16_init_planner.py

```
import threading
import time

import pytest

from drill import load

m = load("p16_init_planner")
GiB = 1 << 30


def test_groups_are_connected_components():
    devs = {0: {0}, 1: {1}, 2: {1}, 3: {2, 3}, 4: {3}, 5: {4}}
    assert m.init_groups(devs) == [[0], [1, 2], [3, 4], [5]]
    assert m.init_groups({0: {0, 1}, 1: {1, 2}, 2: {2}}) == [[0, 1, 2]]


def test_groups_run_in_parallel_and_sequential_within_group():
    devs = {0: {0}, 1: {1}, 2: {1}}
    started = {s: threading.Event() for s in devs}
    active = {"g1": 0, "max_g1": 0}
    order = []
    lock = threading.Lock()

    def init(sid):
        started[sid].set()
        with lock:
            order.append(sid)
            if sid in (1, 2):
                active["g1"] += 1
                active["max_g1"] = max(active["max_g1"], active["g1"])
        if sid == 0:
            # only returns if stage 1 starts while stage 0 is still initializing
            assert started[1].wait(2.0), "groups did not run in parallel"
        time.sleep(0.05)
        with lock:
            if sid in (1, 2):
                active["g1"] -= 1
        return f"engine{sid}"

    res = m.run_init(devs, init, cleanup_fn=lambda s, r: None)
    assert res == {0: "engine0", 1: "engine1", 2: "engine2"}
    assert active["max_g1"] == 1, "stages sharing a device must not init concurrently"
    assert order.index(1) < order.index(2)


def test_failure_short_circuits_and_rolls_back():
    devs = {0: {0}, 1: {1}, 2: {1}}
    calls, cleaned = [], []
    stage0_started = threading.Event()

    def init(sid):
        calls.append(sid)
        if sid == 0:
            stage0_started.set()
        if sid == 1:
            stage0_started.wait(2.0)
            raise RuntimeError("OOM while capturing graphs")
        time.sleep(0.05)
        return sid

    with pytest.raises(m.InitError) as ei:
        m.run_init(devs, init, cleanup_fn=lambda s, r: cleaned.append(s))
    assert ei.value.stage_id == 1
    assert 2 not in calls, "stage 2 must not start after stage 1 failed in the same group"
    assert cleaned == [0], "successfully initialized stages are cleaned up"


def test_admission_check():
    totals = {0: 32 * GiB}
    ok = {0: ({0}, 0.5), 1: ({0}, 0.3)}
    assert m.admission_check(totals, ok, reserve_bytes_per_stage=GiB) == []
    bad = {0: ({0}, 0.6), 1: ({0}, 0.3), 2: ({0}, 0.1)}
```


© 2026 kzhou003

```
    errs = m.admission_check(totals, bad, reserve_bytes_per_stage=GiB)
    assert len(errs) == 1 and "device 0" in errs[0]
```

B-26 · tests/test_p17_graph_runner.py

```
import torch

from drill import load

m = load("p17_graph_runner")
H = 8
W = torch.randn(H, H, generator=torch.Generator().manual_seed(0))


def fn(x):
    return torch.tanh(x @ W) + 1.0


def make():
    r = m.GraphRunner(fn, buckets=[1, 2, 4, 8], hidden=H)
    r.capture()
    return r


def test_matches_eager_for_every_batch_size():
    r = make()
    for bs in range(1, 9):
        x = torch.randn(bs, H)
        torch.testing.assert_close(r.run(x), fn(x))
    assert r.stats["eager"] == 0 and r.stats["replays"] == 8


def test_bucket_selection_and_eager_fallbacks():
    r = make()
    r.run(torch.randn(3, H))
    assert r.stats["last_bucket"] == 4
    r.run(torch.randn(9, H))  # larger than every bucket
    assert r.stats["eager"] == 1
    r.run(torch.randn(2, H), force_eager=True)  # e.g. a batch with per-row RNG generators
    assert r.stats["eager"] == 2


def test_result_does_not_alias_static_buffers():
    r = make()
    x1, x2 = torch.randn(2, H), torch.randn(2, H)
    y1 = r.run(x1)
    expected = fn(x1)
    r.run(x2)
    torch.testing.assert_close(y1, expected, msg="y1 was overwritten by the next replay")


def test_padding_rows_are_zeroed():
    r = make()
    r.run(torch.full((4, H), 7.0))
    r.run(torch.randn(3, H))
    assert torch.count_nonzero(r.static_in[3]) == 0


def test_captured_once_per_bucket():
    r = make()
    assert sorted(r.graphs) == [1, 2, 4, 8]
    graphs = dict(r.graphs)
    for bs in (1, 3, 5, 8):
        r.run(torch.randn(bs, H))
    assert all(r.graphs[b] is graphs[b] for b in graphs), "never re-capture at run time"
```


© 2026 kzhou003

附录 C · 参考解（带中文注释；易错: 标出容易写错的地方）

C-1 · solutions/m01_rmsnorm_mrope.py

```
import torch


# ------------------------------------------------------------------------
# rms_norm: Qwen3 系列所有 RMSNorm (input_layernorm / post_attention_layernorm / q_norm / k_norm / final norm)
# 的公共实现。算法: y = weight * (x / sqrt(mean(x², dim=-1) + eps))。
# 顺序与 HF Qwen3OmniMoe*RMSNorm 完全一致: 先 upcast 到 fp32 做归一化，再 cast 回输入 dtype，最后乘 weight。
# ------------------------------------------------------------------------
def rms_norm(x: torch.Tensor, weight: torch.Tensor, eps: float) -> torch.Tensor:
    # fp32 upcast: bf16/fp16 下 x² 的求和很容易丢精度甚至溢出，所以统计量必须在 fp32 里算
    x32 = x.float()
    # mean 只对最后一维 (hidden / head_dim) 求，每个 token (每个 head) 独立归一化; keepdim 便于广播
    x32 = x32 * torch.rsqrt(x32.pow(2).mean(-1, keepdim=True) + eps)
    # 易错: 必须先 .to(x.dtype) 再乘 weight (cast-back-before-weight)。
    # 若在 fp32 里先乘 weight 再 cast，bf16 下结果可能与 HF 差 1 ulp，逐元素对比会失败。
    return weight * x32.to(x.dtype)  # normalize in fp32, scale in the input dtype (HF order)


# ------------------------------------------------------------------------
# rope_inv_freq: 标准 RoPE 的逆频率表，inv_freq[i] = theta^(-2i / head_dim)，i = 0..head_dim/2-1。
# 每个频率负责 head_dim 中的一对维度 (rotate_half 布局下是第 i 维和第 i + D/2 维)。
# ------------------------------------------------------------------------
def rope_inv_freq(head_dim: int, theta: float) -> torch.Tensor:
    # arange(0, D, 2) = [0, 2, 4, ...]，长度 D/2; 显式 float32，避免整数运算 / 低精度
    return 1.0 / (theta ** (torch.arange(0, head_dim, 2, dtype=torch.float32) / head_dim))


# ------------------------------------------------------------------------
# mrope_cos_sin: Multimodal RoPE (M-RoPE)。每个 token 有 3 个位置分量 (t, h, w)，
# 把 D/2 个频率按 mrope_section 分给三个轴: 频率 j 的角度 = pos[该频率所属的轴] * inv_freq[j]。
# 两种分配布局:
#   interleaved=True  (Qwen3-Omni / Qwen3-VL): T H W T H W ... 循环交错，H/W 用完后剩下的全给 T
#   interleaved=False (Qwen2-VL / Qwen2.5-Omni): 连续分块 [T...T | H...H | W...W]
# 纯文本时 t = h = w，两种布局都退化为普通 1D RoPE。
# ------------------------------------------------------------------------
def mrope_cos_sin(
    inv_freq: torch.Tensor,     # [head_dim // 2]
    positions: torch.Tensor,    # [3, T]  (temporal, height, width)
    mrope_section: list[int],   # sums to head_dim // 2
    interleaved: bool,
) -> tuple[torch.Tensor, torch.Tensor]:  # cos, sin: [T, head_dim]
    # 先把 3 个轴各自在全部 D/2 个频率上的角度都算出来，后面再按布局"挑"每个频率该用哪个轴
    freqs = positions.float()[:, :, None] * inv_freq[None, None, :]  # [3, T, half]
    if interleaved:  # Qwen3-Omni / Qwen3-VL: T H W T H W ... T T
        # 以 T 轴为底 (clone 避免改到 freqs[0] 本身)，再把属于 H / W 的频率位置覆盖掉
        f = freqs[0].clone()  # [T, half]
        # axis=1(H) 占下标 1, 4, 7, ...; axis=2(W) 占下标 2, 5, 8, ... (offset 恰好等于轴号)
        for axis, offset in ((1, 1), (2, 2)):
            # 易错: slice 的 stop 是 mrope_section[axis] * 3 而不是 half——
            # 每个轴只拿 section[axis] 个频率，交错区之后的尾巴保持 T 轴
            idx = slice(offset, mrope_section[axis] * 3, 3)
            f[..., idx] = freqs[axis][..., idx]
    else:  # Qwen2-VL / Qwen2.5-Omni: contiguous chunks [TTTT HH WW]
        # 第 axis 段取该轴在频率区间 [start, start + n) 上的角度，三段拼回 half 长度
        parts, start = [], 0
        for axis, n in enumerate(mrope_section):
            parts.append(freqs[axis][..., start:start + n])
            start += n
        f = torch.cat(parts, dim=-1)  # [T, half]
    # 复制一份拼成 [T, D]: 前半和后半用同一组角度，对应 rotate_half 的"前半/后半配对"布局
    # (不是相邻两维 (2i, 2i+1) 配对的布局)
    emb = torch.cat([f, f], dim=-1)  # [T, head_dim]
    return emb.cos(), emb.sin()


# rotate_half: 把 [x1, x2] (各 D/2) 变成 [-x2, x1]，即二维旋转中与 sin 相乘的那一项。
def rotate_half(x: torch.Tensor) -> torch.Tensor:
    x1, x2 = x.chunk(2, dim=-1)  # 沿 head_dim 对半切
    return torch.cat([-x2, x1], dim=-1)


# apply_rotary: x' = x * cos + rotate_half(x) * sin，即对每对 (x_i, x_{i+D/2}) 做角度 θ_i 的旋转。
def apply_rotary(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    """x: [T, num_heads, head_dim]; cos/sin: [T, head_dim]."""
    # [T, D] -> [T, 1, D] 以便在 head 维广播;
```


© 2026 kzhou003

```
    # cos/sin 先 cast 到 x.dtype 再相乘（HF 的 rotary embedding 返回的就是 x.dtype 的 cos/sin）
    cos, sin = cos[:, None, :].to(x.dtype), sin[:, None, :].to(x.dtype)
    return x * cos + rotate_half(x) * sin
```

C-2 · solutions/m02_thinker_attention.py

```
from dataclasses import dataclass, field

import torch


# ------------------------------------------------------------ given types
# AttnWeights：一层 thinker self-attention 的全部权重（从 HF 模块里摘出来的裸张量）。
# GQA：num_heads 个 query head 共享 num_kv_heads 组 K/V（num_heads % num_kv_heads == 0）。
@dataclass
class AttnWeights:
    q_proj: torch.Tensor  # [num_heads * head_dim, hidden]
    k_proj: torch.Tensor  # [num_kv_heads * head_dim, hidden]
    v_proj: torch.Tensor  # [num_kv_heads * head_dim, hidden]
    o_proj: torch.Tensor  # [hidden, num_heads * head_dim]
    q_norm: torch.Tensor  # [head_dim]  RMSNorm weight applied per head
    k_norm: torch.Tensor  # [head_dim]
    num_heads: int
    num_kv_heads: int
    head_dim: int
    eps: float = 1e-6


# KVCache：一层的 KV cache，按 chunk 追加（prefill 一个大 chunk，decode 每步一个 [1, ...] chunk）。
# 不变量：len(cache) = 已缓存的 token 数 = 下一批新 token 的起始绝对位置（cache offset）。
@dataclass
class KVCache:
    k: list[torch.Tensor] = field(default_factory=list)  # chunks of [t, num_kv_heads, head_dim]
    v: list[torch.Tensor] = field(default_factory=list)

    def __len__(self) -> int:
        return sum(x.shape[0] for x in self.k)


def from_hf(att) -> AttnWeights:
    """Pull the weights out of a HF Qwen3OmniMoeThinkerTextAttention."""
    return AttnWeights(
        att.q_proj.weight.detach(), att.k_proj.weight.detach(), att.v_proj.weight.detach(),
        att.o_proj.weight.detach(), att.q_norm.weight.detach(), att.k_norm.weight.detach(),
        att.config.num_attention_heads, att.config.num_key_value_heads, att.head_dim, att.config.rms_norm_eps,
    )


# ------------------------------------------------------------ implementation
# _rms：同 m01.rms_norm（fp32 归一化 -> cast 回原 dtype -> 乘 weight），这里作用在每个 head 的 head_dim 上。
def _rms(x: torch.Tensor, w: torch.Tensor, eps: float) -> torch.Tensor:
    x32 = x.float()
    return w * (x32 * torch.rsqrt(x32.pow(2).mean(-1, keepdim=True) + eps)).to(x.dtype)


# _rotate_half：[a, b] -> [-b, a]，配合 _rope 完成 RoPE 旋转（同 m01.rotate_half）。
def _rotate_half(x: torch.Tensor) -> torch.Tensor:
    a, b = x.chunk(2, dim=-1)
    return torch.cat([-b, a], dim=-1)


# _rope：x [T, heads, D]，cos/sin [T, D] -> 在 head 维插 1 广播。
def _rope(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    return x * cos[:, None] + _rotate_half(x) * sin[:, None]


# ---------------------------------------------------------------------------
# attention_forward：Qwen3-Omni thinker 的一层 self-attention（QK-norm + GQA + KV cache）。
# 流程：q/k/v 投影 -> 按 head 切分 -> 对 q、k 做 per-head RMSNorm（QK-norm）-> RoPE
#       -> 新 k/v 追加进 cache -> GQA 扩展 k/v -> 因果 mask 的 softmax(QK^T/√D) V -> o_proj。
# 同一个函数既做 prefill（T 个新 token、cache 为空）也做 decode（T=1、cache 已有 past 个）。
# ---------------------------------------------------------------------------
def attention_forward(
    w: AttnWeights,
    x: torch.Tensor,    # [T, hidden]  new tokens only
    cos: torch.Tensor,  # [T, head_dim] for the new tokens
    sin: torch.Tensor,
    cache: KVCache | None = None,
) -> torch.Tensor:      # [T, hidden]
    T = x.shape[0]
    H, KVH, D = w.num_heads, w.num_kv_heads, w.head_dim
    # 投影后 view 成 [T, heads, D]，再在 D 维做 RMSNorm。
    # 易错：QK-norm 必须在 RoPE 之前（HF 顺序）；V 不做 norm。
    q = _rms((x @ w.q_proj.T).view(T, H, D), w.q_norm, w.eps)  # QK-norm BEFORE rope
```


© 2026 kzhou003

```
    k = _rms((x @ w.k_proj.T).view(T, KVH, D), w.k_norm, w.eps)  # [T, KVH, D]
    v = (x @ w.v_proj.T).view(T, KVH, D)  # [T, KVH, D]
    # cos/sin 只覆盖新 token 的绝对位置；cache 里的旧 k 写入时已经转过，不能再转一次
    q, k = _rope(q, cos, sin), _rope(k, cos, sin)

    # past = 本次之前已缓存的 token 数 = 新 token 的起始绝对位置 (cache offset)
    past = 0
    if cache is not None:
        past = len(cache)  # 易错：必须在 append 之前取长度，否则 offset 多算了 T
        # cache 里存的是"RoPE 之后、GQA 扩展之前"的 k（只存 KVH 组，省显存）
        cache.k.append(k)
        cache.v.append(v)
        k, v = torch.cat(cache.k), torch.cat(cache.v)  # [S, KVH, D]，S = past + T

    # GQA：每个 kv head 服务 group 个连续的 q head（q head h 用 kv head h // group）。
    # 易错：必须 repeat_interleave（[k0,k0,k1,k1]）而不是 repeat（[k0,k1,k0,k1]），否则 head 对应关系错乱
    group = H // KVH
    k = k.repeat_interleave(group, dim=1)  # [S, H, D]
    v = v.repeat_interleave(group, dim=1)
    # 打分在 fp32 中做（softmax 对精度敏感）：scores[h, t, s] = q_t · k_s / √D
    scores = torch.einsum("thd,shd->hts", q.float(), k.float()) * D ** -0.5  # [H, T, S]
    S = k.shape[0]
    # 因果 mask 用"绝对位置"构造：新 token t 的位置是 past + t，key s 的位置就是 s。
    # 易错：若用 triu(ones(T, S), 1) 这种只看相对下标的写法，T < S（decode / 分块 prefill）时会错误地 mask 掉历史
    q_pos = torch.arange(past, past + T)[:, None]  # [T, 1]
    k_pos = torch.arange(S)[None, :]  # [1, S]
    scores = scores.masked_fill(k_pos > q_pos, float("-inf"))  # causal w.r.t. absolute positions
    probs = torch.softmax(scores, dim=-1)  # 在 key 维 S 上归一化
    # 加权求和仍在 fp32，结果先 cast 回输入 dtype 再做 o_proj
    out = torch.einsum("hts,shd->thd", probs, v.float()).to(x.dtype)  # [T, H, D]
    # 各 head 拼回 [T, H*D]（head 在前、D 在后，与 o_proj 的列布局一致），再投影回 hidden
    return out.reshape(T, H * D) @ w.o_proj.T
```

C-3 · solutions/m03_sparse_moe.py

```
from dataclasses import dataclass

import torch
import torch.nn.functional as F


# ---------------------------------------------------------------- given types
# SharedExpert：talker 的 MoE block 额外带一个"所有 token 都走"的 shared expert（SwiGLU MLP），
# 其输出再乘一个逐 token 的 sigmoid 门控。thinker 的 MoE 没有 shared expert（shared=None）。
@dataclass
class SharedExpert:
    gate_proj: torch.Tensor   # [I_s, H]
    up_proj: torch.Tensor     # [I_s, H]
    down_proj: torch.Tensor   # [H, I_s]
    gate: torch.Tensor        # [1, H]   sigmoid gate of the shared expert


# MoeWeights：把 E 个 expert 的权重 stack 成 [E, ...] 的大张量，按 expert id 下标即可取到。
@dataclass
class MoeWeights:
    router: torch.Tensor      # [E, H]
    w_gate: torch.Tensor      # [E, I, H]   stacked expert gate_proj
    w_up: torch.Tensor        # [E, I, H]
    w_down: torch.Tensor      # [E, H, I]
    top_k: int
    norm_topk_prob: bool
    shared: SharedExpert | None = None


def from_hf(block) -> MoeWeights:
    """HF Qwen3OmniMoe{Thinker,Talker}TextSparseMoeBlock -> stacked tensors."""
    ex = block.experts
    shared = None
    # 只有 talker 的 block 有 shared_expert / shared_expert_gate 属性
    if hasattr(block, "shared_expert"):
        s = block.shared_expert
        shared = SharedExpert(s.gate_proj.weight.detach(), s.up_proj.weight.detach(),
                              s.down_proj.weight.detach(), block.shared_expert_gate.weight.detach())
    return MoeWeights(
        block.gate.weight.detach(),  # HF 里 router 叫 `gate`（别和 expert 的 gate_proj 混淆）
        torch.stack([e.gate_proj.weight.detach() for e in ex]),
        torch.stack([e.up_proj.weight.detach() for e in ex]),
        torch.stack([e.down_proj.weight.detach() for e in ex]),
        block.top_k, block.norm_topk_prob, shared,
    )


# ---------------------------------------------------------------- implementation
# ------------------------------------------------------------------------------
```


© 2026 kzhou003

```
# route: router logits -> softmax (对全部 E 个 expert) -> top-k -> 可选重归一化 -> cast 回 x.dtype。
# 易错: 顺序是"先 softmax 再 top-k" (不是对 top-k 的 logits 做 softmax);
#       norm_topk_prob=False 时 top-k 权重之和 < 1, 不能擅自归一化。
# ------------------------------------------------------------------------
def route(x: torch.Tensor, w: MoeWeights) -> tuple[torch.Tensor, torch.Tensor]:
    """-> (weights [N, k] in x.dtype, expert ids [N, k]). softmax over ALL experts, then top-k."""
    # logits: [N, E]; softmax 在 fp32 中做 (dtype=torch.float), 与 HF 一致
    probs = F.softmax(x @ w.router.T, dim=-1, dtype=torch.float)
    topw, topi = torch.topk(probs, w.top_k, dim=-1)  # [N, k] 权重 / expert id, 按权重降序
    if w.norm_topk_prob:
        # 只在选中的 k 个里重新归一化, 使每个 token 的 k 个权重和为 1 (仍在 fp32 里做)
        topw = topw / topw.sum(-1, keepdim=True)
    # 最后才 cast 回 x.dtype: 路由权重要和 expert 输出 (x.dtype) 相乘
    return topw.to(x.dtype), topi


# ------------------------------------------------------------------------
# moe_forward: 稀疏 MoE 前向 ("按 expert 分组"的 dispatch / combine 写法)。
# 1) 把 N 个 token × k 个槽位摊平成 N*k 个 (token, expert, weight) 三元组;
# 2) 按 expert id 稳定排序, 统计每个 expert 分到多少个三元组;
# 3) 逐 expert 收集它的 token 做一次 SwiGLU MLP, 乘路由权重后 index_add_ 回各自 token;
# 4) 若有 shared expert, 所有 token 再加上 sigmoid(gate) * shared_mlp(x)。
# ------------------------------------------------------------------------
def moe_forward(x: torch.Tensor, w: MoeWeights) -> torch.Tensor:
    N, H = x.shape  # x: [N, H], N 个 token (batch/seq 已展平)
    topw, topi = route(x, w)  # [N, k], [N, k]
    flat_e = topi.reshape(-1)                       # [N*k]
    # 第 i 个三元组属于 token i // k: [0,0,..,1,1,..], 与 reshape(-1) 的行优先顺序对齐
    flat_tok = torch.arange(N).repeat_interleave(w.top_k)  # [N*k]
    flat_w = topw.reshape(-1)  # [N*k]
    # stable 排序: 同一 expert 内保持 token 原顺序, 结果可复现
    order = torch.argsort(flat_e, stable=True)     # group (token, slot) pairs by expert
    # counts[e] = 分给 expert e 的三元组数; minlength=E 保证没被选中的 expert 也占一项 (值 0)
    counts = torch.bincount(flat_e, minlength=w.w_gate.shape[0]).tolist()
    out = torch.zeros_like(x)  # [N, H] 累加器
    # start 指向 order 中当前 expert 那一段的起点; 各 expert 的段按 e 升序首尾相接
    start = 0
    for e, n in enumerate(counts):
        if n == 0:
            continue                                # idle expert: no work at all
        sel = order[start:start + n]  # 属于 expert e 的三元组在 flat_* 中的下标
        start += n
        toks = flat_tok[sel]  # [n] 对应 token id (top-k 不重复, 同一 token 在同一 expert 里只出现一次)
        h = x[toks]  # [n, H] gather
        # SwiGLU: down( silu(gate(h)) * up(h) ); w_gate[e]: [I, H], h @ w_gate[e].T -> [n, I]
        y = (F.silu(h @ w.w_gate[e].T) * (h @ w.w_up[e].T)) @ w.w_down[e].T  # [n, H]
        # combine: 乘路由权重后按 token 散射累加。
        # 易错: 一个 token 会被 k 个不同 expert 处理, 必须 index_add_ 累加, 不能 out[toks] = ... (会覆盖)
        out.index_add_(0, toks, y * flat_w[sel, None])
    if w.shared is not None:
        s = w.shared
        # shared expert 对全部 token 计算 (不经过 router)
        y = (F.silu(x @ s.gate_proj.T) * (x @ s.up_proj.T)) @ s.down_proj.T  # [N, H]
        # 门控是逐 token 标量: x @ gate.T -> [N, 1], sigmoid 后广播到 H 维
        out = out + torch.sigmoid(x @ s.gate.T) * y
    return out
```

C-4 · solutions/m04_mm_merge_deepstack.py

```
from collections.abc import Callable

import torch


# ------------------------------------------------------------------------
# audio_token_count: 预处理阶段要在 prompt 里放多少个 <audio> placeholder, 必须和 AuT 音频编码器
# 实际输出的 token 数严格相等, 否则后面的 merge_multimodal 会对不上。
# 规则: 每满 100 个 mel 帧 -> 13 个 token; 余下 r 帧单独过三次 stride-2 卷积, 每次长度 L -> (L-1)//2 + 1。
# 只用 // % + - 运算, 所以 int 和 LongTensor (逐元素) 都能直接用。
# ------------------------------------------------------------------------
def audio_token_count(num_mel_frames):
    """How many <audio> placeholder tokens the AuT encoder produces for `num_mel_frames` frames.

    The encoder works on 100-frame windows (each -> 13 tokens after three stride-2 convs)
    plus a remainder window that goes through the same three convs.
    """
    rem = num_mel_frames % 100  # 不足一个窗口的尾巴
    feat = (rem - 1) // 2 + 1  # 第 1 次 stride-2 卷积后的长度
    # 第 2、3 次卷积嵌套在一个表达式里: ((feat-1)//2+1) 是第 2 次的结果, 外层再做第 3 次;
    # 末尾加上整窗部分 (n // 100) * 13 (100 -> 50 -> 25 -> 13)。
    # 边界: rem == 0 时 feat = (-1)//2 + 1 = 0 (floor 除法), 后两次也都是 0, 尾巴贡献 0 个 token。
    # 易错: 必须用 floor 除法 //; 换成 int((rem-1)/2) 这类向零截断的写法, rem==0 时会得到 1。
    return ((feat - 1) // 2 + 1 - 1) // 2 + 1 + (num_mel_frames // 100) * 13
```


© 2026 kzhou003

```
# ------------------------------------------------------------------------
# merge_multimodal：把文本 embedding 里各模态 placeholder 位置的行，按顺序替换成编码器输出，
# 得到 thinker 的 inputs_embeds。同时返回每个模态的 bool mask（后面 DeepStack 需要 visual mask）。
# ------------------------------------------------------------------------
def merge_multimodal(
    input_ids: torch.Tensor,                # [T]
    text_embeds: torch.Tensor,              # [T, H]  embed_tokens(input_ids)
    placeholder_ids: dict[str, int],        # modality -> placeholder token id
    mm_embeds: dict[str, torch.Tensor],     # modality -> [n_tokens, H] encoder outputs, in order
) -> tuple[torch.Tensor, dict[str, torch.Tensor]]:
    # clone：规格要求不能原地修改调用方的 text_embeds
    out = text_embeds.clone()
    masks: dict[str, torch.Tensor] = {}
    for modality, tok in placeholder_ids.items():
        mask = input_ids == tok  # [T] bool，该模态的 placeholder 位置
        # 即使该模态这次没有 embedding 也记录 mask，调用方可统一按模态取
        masks[modality] = mask
        emb = mm_embeds.get(modality)
        n = int(mask.sum())  # placeholder 个数
        if emb is None:
            # 有 placeholder 却没给 embedding 是错误；两者都没有则正常跳过
            if n:
                raise ValueError(f"{n} {modality} placeholders but no {modality} embeddings")
            continue
        # 个数必须严格相等（bool 索引赋值也要求右侧行数 == mask 中 True 的个数）
        if emb.shape[0] != n:
            raise ValueError(f"{modality}: {n} placeholders vs {emb.shape[0]} embeddings")
        # bool mask 赋值按 T 维从前往后依次填入 emb 的第 0, 1, 2... 行（即 “in order” 语义）；
        # encoder 输出的 dtype 可能与文本 embedding 不同，先 cast
        out[mask] = emb.to(out.dtype)
    return out, masks


# ------------------------------------------------------------------------
# apply_deepstack：DeepStack 把视觉编码器中间层的特征“加”（不是替换）到 LLM 某层输出的视觉 token 位置上。
# ------------------------------------------------------------------------
def apply_deepstack(hidden: torch.Tensor, visual_mask: torch.Tensor, visual_embeds: torch.Tensor) -> torch.Tensor:
    """Add one DeepStack level to the hidden states at visual positions (returns a new tensor)."""
    # 返回新张量、不改输入（hidden 可能还被别处引用，如残差）
    out = hidden.clone()  # [T, H]
    # 只有 visual_mask 为 True 的 n_vis 行加上 visual_embeds[n_vis, H]；文本行不变
    out[visual_mask] = out[visual_mask] + visual_embeds.to(out.dtype)
    return out


# ------------------------------------------------------------------------
# run_layers：带 DeepStack 的 decoder 层循环。第 i 层跑完“之后”注入第 i 级 DeepStack 特征；
# 级数通常少于层数，只有前 len(deepstack_embeds) 层之后有注入。
# ------------------------------------------------------------------------
def run_layers(
    layers: list[Callable[[torch.Tensor], torch.Tensor]],
    hidden: torch.Tensor,
    visual_mask: torch.Tensor,
    deepstack_embeds: list[torch.Tensor],
) -> torch.Tensor:
    """Decoder stack with DeepStack: level i is injected right AFTER layer i."""
    for i, layer in enumerate(layers):
        hidden = layer(hidden)
        # 易错：注入时机是 layer i 之后（不是之前），level 0 加在 layer 0 的输出上
        if i < len(deepstack_embeds):
            hidden = apply_deepstack(hidden, visual_mask, deepstack_embeds[i])
    return hidden
```

C-5 · solutions/m05_talker_mtp.py

```
import torch


# ------------------------------------------------------------------------
# Talker 的 MTP（multi-token prediction）/ code predictor：
# 每个音频帧有 G 个 codebook（CB0 + G-1 个残差 codebook）。talker 主干只采样 CB0；
# code predictor（一个小 transformer）以 [talker 最后隐状态, E(CB0)] 开头，自回归地依次预测 CB1..CB{G-1}：
# 第 j 步用 lm_head[j] 预测 CB{j+1}，再用 codec_embedding[j] 把它嵌入、拼到序列末尾。
# 下面两种实现结果必须逐元素相同：re-prefill（每步重算整个序列）和 KV-cache 增量版。
# ------------------------------------------------------------------------
@torch.no_grad()
def predict_residual_codes(cp, last_talker_hidden: torch.Tensor, cb0_embed: torch.Tensor) -> torch.Tensor:
    """Greedy residual codebooks 1..G-1 by RE-PREFILL (no KV cache).

    cp: HF Qwen3OmniMoeTalkerCodePredictorModelForConditionalGeneration
        cp.model(inputs_embeds=[B, L, H]).last_hidden_state, cp.lm_head[j], cp.model.codec_embedding[j]
    last_talker_hidden: [B, H]  talker's last-layer hidden state at the step that sampled CB0
    cb0_embed:          [B, H]  talker codec embedding of the sampled CB0
```


© 2026 kzhou003

```
    returns codes [B, G-1]
    """
    G = cp.config.num_code_groups  # 每帧 codebook 总数（含 CB0）
    # 初始序列长度 2：位置 0 = talker 隐状态，位置 1 = CB0 的 embedding
    seq = torch.stack([last_talker_hidden, cb0_embed], dim=1)  # [B, 2, H]
    codes = []
    for j in range(G - 1):  # predicts codebook j+1
        # 每步把整个序列重新送进模型（无 cache），只取最后一个位置的隐状态；序列长度 2 + j
        h = cp.model(inputs_embeds=seq).last_hidden_state[:, -1]  # [B, H]
        # 易错：第 j 步用第 j 个 head（lm_head[j] 对应 CB{j+1}），每个残差 codebook 有独立的 head 和 embedding
        tok = cp.lm_head[j](h).argmax(-1)  # [B] greedy
        codes.append(tok)
        # 用 codec_embedding[j]（与 lm_head[j] 同一个 codebook）嵌入刚预测的 code，序列长度 +1
        seq = torch.cat([seq, cp.model.codec_embedding[j](tok)[:, None]], dim=1)
    # 每步一个 [B]，按列拼成 [B, G-1]（第 0 列是 CB1）
    return torch.stack(codes, dim=1)


# KV 版：第一步 prefill 长度 2，之后每步只送 1 个新位置；cache 长度依次 2 -> 3 -> ...，
# 新位置的 position 由 HF 根据 cache 长度自动推出，因此和 re-prefill 版数学上等价。
@torch.no_grad()
def predict_residual_codes_kv(cp, last_talker_hidden: torch.Tensor, cb0_embed: torch.Tensor) -> torch.Tensor:
    """Same result with an incremental KV cache: prefill 2 positions, then 1 position per step."""
    from transformers import DynamicCache

    G = cp.config.num_code_groups
    # 每次调用新建 cache：每个音频帧的残差预测是一段独立的短序列，不能复用上一帧的 cache
    cache = DynamicCache()
    x = torch.stack([last_talker_hidden, cb0_embed], dim=1)  # 首步 [B, 2, H]
    codes = []
    for j in range(G - 1):
        # HF 在 forward 内原地把本次 x 的 K/V 追加进 cache
        h = cp.model(inputs_embeds=x, past_key_values=cache, use_cache=True).last_hidden_state[:, -1]
        tok = cp.lm_head[j](h).argmax(-1)
        codes.append(tok)
        # 易错：下一步只喂“新 token”的 embedding [B, 1, H]，不要把整个序列再喂一遍（会重复写 cache）
        x = cp.model.codec_embedding[j](tok)[:, None]
    return torch.stack(codes, dim=1)


# ------------------------------------------------------------------------
# next_talker_input：talker 下一步的输入 embedding =
#   本帧全部 G 个 codebook 的 embedding 之和（CB0 用 talker 的 codec embedding，CB1.. 用 code predictor 的）
#   + 当前步对应的文本侧向量 text_step。
# ------------------------------------------------------------------------
@torch.no_grad()
def next_talker_input(cp, cb0_embed: torch.Tensor, residual_codes: torch.Tensor, text_step: torch.Tensor) -> torch.Tensor:
    """Talker input of the NEXT step: sum of all G codec embeddings of this frame + the text row."""
    total = cb0_embed.clone()  # [B, H]，从 CB0 的 embedding 开始累加，不改调用方张量
    # residual_codes: [B, G-1]，第 j 列（CB{j+1}）用 codec_embedding[j] 嵌入
    for j in range(residual_codes.shape[1]):
        total = total + cp.model.codec_embedding[j](residual_codes[:, j])
    return total + text_step  # [B, H]
```

C-6 · solutions/m06_streaming_causal_conv.py

```
import torch
import torch.nn.functional as F


# ------------------------------------------------------------------------
# StreamingCausalConv1d：Code2Wav 里的因果 Conv1d（stride 1）的流式版本。
# 离线版 = 左补 (K-1)*d 个 0 再做 valid 卷积。流式版把“左补零”换成一个 state：
# 永远保存最近 context = (K-1)*d 个历史输入；每来一块 x 就拼成 [state | x] 做 valid 卷积，
# 输出长度恰好等于本块长度 t，所有块的输出拼起来与离线逐元素相同。
# ------------------------------------------------------------------------
class StreamingCausalConv1d:
    """Chunk-by-chunk version of Code2Wav's causal Conv1d (stride 1, any dilation)."""

    def __init__(self, weight: torch.Tensor, bias: torch.Tensor | None, dilation: int = 1) -> None:
        # weight: [C_out, C_in, K]
        self.weight, self.bias, self.dilation = weight, bias, dilation
        # 感受野跨度 (K-1)*d：输出位置 t 需要输入 t-(K-1)*d .. t
        self.context = (weight.shape[-1] - 1) * dilation  # inputs of the past this conv still needs
        self.state: torch.Tensor | None = None  # [B, C_in, context]，首块到来时才知道 B / dtype / device

    @classmethod
    def from_hf(cls, m) -> "StreamingCausalConv1d":  # HF Qwen3OmniMoeCausalConvNet
        assert m.stride == 1  # 流式状态推导只对 stride 1 成立
        return cls(m.conv.weight.detach(), None if m.conv.bias is None else m.conv.bias.detach(), m.dilation)

    def step(self, x: torch.Tensor) -> torch.Tensor:  # [B, C_in, t] -> [B, C_out, t]
        if self.state is None:
            # 首块之前的“历史”就是离线版的左补零
```


© 2026 kzhou003

```
            self.state = x.new_zeros(x.shape[0], x.shape[1], self.context)  # causal zero padding
        buf = torch.cat([self.state, x], dim=-1)  # [B, C_in, context + t]
        # 新 state = buf 的最后 context 个样本。
        # 易错：要从 buf（旧 state + 新块）末尾取，而不是从 x 末尾取——当 t < context 时 x 不够长，
        #       state 里仍需保留一部分旧样本（可能还包括补出来的 0）。
        # context == 0（K=1，无需历史）时 state 显式置为长度 0 的空张量。
        self.state = buf[..., buf.shape[-1] - self.context:] if self.context else buf[..., :0]
        # valid 卷积（无 padding）：输出长度 (context + t) - context = t
        return F.conv1d(buf, self.weight, self.bias, dilation=self.dilation)


# ---------------------------------------------------------------------------
# StreamingCausalTransConv1d：Code2Wav 上采样用的因果 ConvTranspose1d 的流式版本（overlap-add）。
# 每个输入帧 i 写入输出区间 [i*s, i*s + K)，相邻帧的写入区间重叠 K-s 个样本。
# 维护一个累加缓冲 acc（从全局位置 base 开始、尚未"定稿"的样本）：
#   - step：本块的转置卷积结果加到 acc 的对应位置上；收到 n 帧后，位置 < n*s 的样本不会再被写，可以输出；
#   - 离线模块会裁掉开头 K-s 和结尾 K-s 个样本，流式版在 _emit 里做全局左裁、在 flush 里做右裁。
# ---------------------------------------------------------------------------
class StreamingCausalTransConv1d:
    """Chunk-by-chunk version of Code2Wav's causal ConvTranspose1d (upsampling by `stride`).

    Input frame i writes output positions [i*s, i*s + K). Position p is final once every frame
    that touches it has arrived, i.e. p < n*s after n frames. The full (offline) module trims
    K - s samples on both ends.
    """

    def __init__(self, weight: torch.Tensor, bias: torch.Tensor | None, stride: int) -> None:
        self.weight, self.bias, self.stride = weight, bias, stride  # weight: [C_in, C_out, K]
        self.K = weight.shape[-1]
        self.trim = self.K - stride  # 离线版两端各裁掉的样本数
        self.acc: torch.Tensor | None = None  # pending (not final) samples, starting at self.base
        self.base = 0  # acc[..., 0] 对应的全局输出位置（= 已输出（含被裁掉的）样本总数）
        self.frames = 0  # 已收到的输入帧总数

    @classmethod
    def from_hf(cls, m) -> "StreamingCausalTransConv1d":  # HF Qwen3OmniMoeCausalTransConvNet
        c = m.conv
        return cls(c.weight.detach(), None if c.bias is None else c.bias.detach(), c.stride[0])

    # _emit：把全局位置 [base, upto) 的已定稿样本从 acc 中切出、加 bias、做全局左裁后返回。
    def _emit(self, upto: int) -> torch.Tensor:
        assert self.acc is not None
        n = upto - self.base  # 本次定稿的样本数
        out, self.acc = self.acc[..., :n], self.acc[..., n:]  # acc 只保留尚未定稿的尾巴
        start = self.base  # out[..., 0] 的全局位置
        self.base = upto
        # 易错：bias 只能在定稿时每个位置加一次。如果在 conv_transpose1d 里带 bias，
        # 重叠区会被多个帧各加一次 bias，结果与离线不符。
        if self.bias is not None:  # bias once per output position (never inside the overlap-add)
            out = out + self.bias[None, :, None]
        # 全局左裁：全局位置 < trim 的样本属于离线版会裁掉的开头；
        # 它们可能横跨多次 _emit，所以按 start 算还剩多少要裁（start >= trim 后 skip 恒为 0）
        skip = max(0, self.trim - start)  # global left trim
        return out[..., skip:]

    def step(self, x: torch.Tensor) -> torch.Tensor:
        # x: [B, C_in, t] 新到的 t 帧
        s = self.stride
        # 本块单独做转置卷积（不带 bias），长度 (t-1)*s + K
        y = F.conv_transpose1d(x, self.weight, None, stride=s)  # covers [frames*s, frames*s + len)
        start = self.frames * s  # 本块第一帧写入的全局起点
        end = start + y.shape[-1]
        if self.acc is None:
            self.acc = y.new_zeros(y.shape[0], y.shape[1], 0)
        # 不变量：每次 step 结束时 base == frames*s，所以这里 start == base，
        # acc 里现存的就是上一块留下的 K-s 个重叠尾巴（位于 [base, base+K-s)）
        need = end - self.base
        if self.acc.shape[-1] < need:
            # 右侧补 0 扩展到能放下本块全部写入
            self.acc = F.pad(self.acc, (0, need - self.acc.shape[-1]))
        # overlap-add：与上一块尾巴重叠的部分相加
        self.acc[..., start - self.base:end - self.base] += y
        self.frames += x.shape[-1]
        # 收到 frames 帧后，位置 < frames*s 的样本已不会再被后续帧写到，可以定稿输出
        return self._emit(self.frames * s)

    def flush(self) -> torch.Tensor:
        # 流结束：剩下的全部 pending 样本定稿，再裁掉离线版结尾的 trim 个样本
        assert self.acc is not None
        total = (self.frames - 1) * self.stride + self.K  # 离线未裁剪时的总长度
        out = self._emit(total)
        # 易错：右裁只能在 flush 时做——流式中途不知道哪里是结尾。
        # 正常情况下剩余 pending 恰好是 K-s = trim 个，被全部裁掉，flush 返回长度 0；trim == 0 时原样返回
        return out[..., :out.shape[-1] - self.trim] if self.trim else out
```


© 2026 kzhou003

C-7 · solutions/m07_stage_weight_loading.py

```
import re
from dataclasses import dataclass

import torch

# Qwen3-Omni 的一个 checkpoint 同时包含三个 stage 的权重，靠 key 的顶层前缀区分。
# 易错：前缀带点号 "thinker."，否则 "thinkers.xxx" 这类相似前缀也会被误匹配。
STAGE_PREFIX = {"thinker": "thinker.", "talker": "talker.", "code2wav": "code2wav."}


class LoadError(Exception):
    pass


# ---------------------------------------------------------------------------
# shards_for_stage：多 stage 部署时每个 stage 进程只加载自己的权重，
# 先从 index.json 的 weight_map（param key -> shard 文件名）里找出需要打开的 shard。
# ---------------------------------------------------------------------------
def shards_for_stage(weight_map: dict[str, str], stage: str) -> list[str]:
    """model.safetensors.index.json 'weight_map' -> the shard files this stage must open (sorted)."""
    prefix = STAGE_PREFIX[stage]
    # set 去重（一个 shard 可能包含多个 stage 的 key），sorted 保证结果确定
    return sorted({f for k, f in weight_map.items() if k.startswith(prefix)})


# ---------------------------------------------------------------------------
# select_stage：从全量 state dict 里筛出某个子模块（如 "thinker.model."）的权重并去掉前缀，
# 得到子模块本地命名（"layers.0.xxx"、"norm.weight"...）。
# ---------------------------------------------------------------------------
def select_stage(state: dict[str, torch.Tensor], stage: str, strip: str) -> dict[str, torch.Tensor]:
    """Keep only keys under `strip` (e.g. "thinker.model.") and drop that prefix."""
    # strip 必须落在该 stage 之下，防止调用方误用别的 stage 的前缀
    assert strip.startswith(STAGE_PREFIX[stage])
    # 不在 strip 下的 key（如 thinker.lm_head.weight）会被丢掉，需要时由调用方另取
    return {k[len(strip):]: v for k, v in state.items() if k.startswith(strip)}


# PackedLayer：推理引擎常用的“打包”权重布局——q/k/v 拼成一个 qkv 矩阵、每个 expert 的 gate/up 拼成 w13，
# 这样一次 matmul 就能算出多个投影。
@dataclass
class PackedLayer:
    input_layernorm: torch.Tensor
    qkv: torch.Tensor                # [(H + 2*KVH) * D, hidden]  rows: q | k | v
    q_norm: torch.Tensor
    k_norm: torch.Tensor
    o_proj: torch.Tensor
    post_attention_layernorm: torch.Tensor
    router: torch.Tensor             # [E, hidden]
    w13: torch.Tensor                # [E, 2*I, hidden]  rows: gate | up
    w2: torch.Tensor                 # [E, hidden, I]


# 匹配层内 expert 权重名，如 "mlp.experts.12.up_proj.weight" -> group(1)="12", group(2)="up_proj"
_EXPERT = re.compile(r"^mlp\.experts\.(\d+)\.(gate_proj|up_proj|down_proj)\.weight$")


# ---------------------------------------------------------------------------
# pack_layer：把第 i 层的 HF 散装权重打包成 PackedLayer，并做严格校验：
# 缺失的 key、expert id 不连续、某 expert 缺 gate/up/down、多出来不认识的 key，都抛 LoadError。
# 用 `used` 记录消费过的 key，最后用 “全部 key - used” 找出 unexpected。
# ---------------------------------------------------------------------------
def pack_layer(sd: dict[str, torch.Tensor], i: int) -> PackedLayer:
    p = f"layers.{i}."
    # 易错：前缀带结尾的点，否则 "layers.1." 以外还会匹配到 "layers.10."、"layers.11." ...
    sub = {k[len(p):]: v for k, v in sd.items() if k.startswith(p)}  # 层内相对 key
    used: set[str] = set()

    # take：取一个必需的权重；不存在就报 missing，存在则记入 used
    def take(name: str) -> torch.Tensor:
        if name not in sub:
            raise LoadError(f"missing {p}{name}")
        used.add(name)
        return sub[name]

    # 收集 expert 权重：experts[e]["gate_proj" | "up_proj" | "down_proj"] = tensor
    experts: dict[int, dict[str, torch.Tensor]] = {}
    for k, v in sub.items():
        mm = _EXPERT.match(k)
        if mm:
            experts.setdefault(int(mm.group(1)), {})[mm.group(2)] = v
            used.add(k)
    # expert id 必须恰好是 0..E-1（不能为空、不能有空洞）。
    # 易错：sorted(experts) 是按整数排序；若用字符串 key 排序会得到 0, 1, 10, 11, 2 ...
    if not experts or sorted(experts) != list(range(len(experts))):
```


© 2026 kzhou003

```
        raise LoadError(f"layer {i}: expert ids must be 0..E-1, got {sorted(experts)}")
    for e, parts in experts.items():
        if set(parts) != {"gate_proj", "up_proj", "down_proj"}:
            raise LoadError(f"layer {i} expert {e}: incomplete {sorted(parts)}")

    layer = PackedLayer(
        input_layernorm=take("input_layernorm.weight"),
        # 沿 dim 0（输出维 / 行）拼接：q | k | v，之后可按 [H*D, KVH*D, KVH*D] split 回去
        qkv=torch.cat([take("self_attn.q_proj.weight"), take("self_attn.k_proj.weight"),
                       take("self_attn.v_proj.weight")], dim=0),
        q_norm=take("self_attn.q_norm.weight"),
        k_norm=take("self_attn.k_norm.weight"),
        o_proj=take("self_attn.o_proj.weight"),
        post_attention_layernorm=take("post_attention_layernorm.weight"),
        router=take("mlp.gate.weight"),  # HF 中 MoE router 叫 mlp.gate
        # 每个 expert 先把 gate [I, hidden] 和 up [I, hidden] 沿行拼成 [2I, hidden]，再按 expert id 顺序 stack
        w13=torch.stack([torch.cat([experts[e]["gate_proj"], experts[e]["up_proj"]], 0) for e in range(len(experts))]),
        w2=torch.stack([experts[e]["down_proj"] for e in range(len(experts))]),  # [E, hidden, I]
    )
    # 所有 take 完成后再检查多余 key（如 shared_expert.*、拼写错误的名字等）
    unexpected = set(sub) - used
    if unexpected:
        raise LoadError(f"layer {i}: unexpected {sorted(unexpected)}")
    return layer


# ---------------------------------------------------------------------------
# load_thinker_text：加载 thinker 文本主干：逐层 pack，并校验层外只有 embed_tokens / norm 两个全局权重。
# 返回 ({"embed_tokens", "norm"}, [PackedLayer] * num_layers)。lm_head 不在 thinker.model. 下，由调用方另取。
# ---------------------------------------------------------------------------
def load_thinker_text(state: dict[str, torch.Tensor], num_layers: int) -> tuple[dict, list[PackedLayer]]:
    sd = select_stage(state, "thinker", "thinker.model.")
    layers = [pack_layer(sd, i) for i in range(num_layers)]
    # 非 layers.* 的 key 只允许是这两个全局权重；
    # 注意：层号 >= num_layers 的多余层不会在这里被发现（它们以 "layers." 开头）
    extra = {k for k in sd if not k.startswith("layers.")} - {"embed_tokens.weight", "norm.weight"}
    if extra:
        raise LoadError(f"unexpected {sorted(extra)}")
    for k in ("embed_tokens.weight", "norm.weight"):
        if k not in sd:
            raise LoadError(f"missing {k}")
    return {"embed_tokens": sd["embed_tokens.weight"], "norm": sd["norm.weight"]}, layers
```

C-8 · solutions/m08_tiny_thinker_e2e.py

```
from dataclasses import dataclass

import torch

from .m01_rmsnorm_mrope import mrope_cos_sin, rms_norm, rope_inv_freq
from .m02_thinker_attention import AttnWeights, KVCache, attention_forward
from .m03_sparse_moe import MoeWeights, moe_forward
from .m07_stage_weight_loading import load_thinker_text


# Dims：模型结构超参（相当于 config 的子集）。
@dataclass
class Dims:
    num_layers: int
    num_heads: int
    num_kv_heads: int
    head_dim: int
    top_k: int
    norm_topk_prob: bool
    rope_theta: float
    mrope_section: list[int]
    eps: float = 1e-6


# ---------------------------------------------------------------------------
# TinyThinker：把 m01（RMSNorm / M-RoPE）、m02（attention + KV cache）、m03（MoE）、m07（权重加载）
# 拼成一个端到端的纯文本 thinker：embed -> N × [pre-norm attention + 残差, pre-norm MoE + 残差]
# -> final norm -> lm_head，并实现带 KV cache 的 greedy 解码。结果应与 HF 每步全量重算的 greedy 一致。
# ---------------------------------------------------------------------------
class TinyThinker:
    """Text-only Qwen3-Omni thinker built from YOUR m01/m02/m03/m07 pieces."""

    def __init__(self, state: dict[str, torch.Tensor], dims: Dims) -> None:
        self.d = dims
        g, layers = load_thinker_text(state, dims.num_layers)
        self.embed, self.norm = g["embed_tokens"], g["norm"]  # embed: [V, hidden]
        # lm_head 在 "thinker.lm_head." 下而非 "thinker.model." 下，load_thinker_text 不会返回它
        self.lm_head = state["thinker.lm_head.weight"]  # [V, hidden]
        D, H, KVH = dims.head_dim, dims.num_heads, dims.num_kv_heads
```


© 2026 kzhou003

```
        self.layers = []
        for L in layers:
            # 把 m07 的打包布局拆回 m02 / m03 需要的形式：qkv 行按 q | k | v 切
            q, k, v = L.qkv.split([H * D, KVH * D, KVH * D], dim=0)
            I2 = L.w13.shape[1]  # = 2 * I
            attn = AttnWeights(q, k, v, L.o_proj, L.q_norm, L.k_norm, H, KVH, D, dims.eps)
            # w13: [E, 2I, hidden]，前 I 行是 gate、后 I 行是 up（与 m07 的拼接顺序对应）
            moe = MoeWeights(L.router, L.w13[:, : I2 // 2], L.w13[:, I2 // 2:], L.w2, dims.top_k, dims.norm_topk_prob)
            self.layers.append((L.input_layernorm, attn, L.post_attention_layernorm, moe))
        # 逆频率表只依赖 head_dim 和 theta，初始化时算一次
        self.inv_freq = rope_inv_freq(D, dims.rope_theta)

    def forward(self, ids: torch.Tensor, start: int, caches: list[KVCache]) -> torch.Tensor:
        """ids: [T] new tokens at absolute positions start..start+T-1 -> logits [T, V]."""
        T = ids.shape[0]
        # 纯文本：t/h/w 三个位置分量相同 -> [3, T]（M-RoPE 退化为 1D RoPE，但仍走 interleaved 路径以对齐 HF）
        # 不变量：调用时 start 必须等于每层 cache 的当前长度
        pos = torch.arange(start, start + T).expand(3, T)
        cos, sin = mrope_cos_sin(self.inv_freq, pos, self.d.mrope_section, interleaved=True)  # [T, D]
        h = self.embed[ids]  # [T, hidden]，embedding 查表
        # strict=True：层数与 cache 数必须一致；每层一个独立的 KVCache
        for (ln1, attn, ln2, moe), cache in zip(self.layers, caches, strict=True):
            # pre-norm：norm 只作用在子层输入上，残差加的是未 norm 的 h
            h = h + attention_forward(attn, rms_norm(h, ln1, self.d.eps), cos, sin, cache)
            h = h + moe_forward(rms_norm(h, ln2, self.d.eps), moe)
        return rms_norm(h, self.norm, self.d.eps) @ self.lm_head.T  # [T, V]

    @torch.no_grad()
    def greedy(self, prompt: list[int], max_new: int) -> list[int]:
        caches = [KVCache() for _ in self.layers]
        # prefill：整段 prompt 从位置 0 开始；最后一行 logits 直接给出第一个新 token
        logits = self.forward(torch.tensor(prompt), 0, caches)
        out = [int(logits[-1].argmax())]
        # 共需 max_new 次 forward：prefill 产出第 1 个，之后每次 decode 产出 1 个；最后一个 token 不需要再 forward
        while len(out) < max_new:
            # decode：只喂上一步生成的 1 个 token，它的绝对位置是 len(prompt) + len(out) - 1（== 当前 cache 长度）。
            # 易错：写成 len(prompt) + len(out) 会让 RoPE 位置整体偏 1，输出很快与 HF 分叉
            logits = self.forward(torch.tensor(out[-1:]), len(prompt) + len(out) - 1, caches)
            out.append(int(logits[-1].argmax()))
        return out
```

C-9 · solutions/p02_orchestrator.py

```
import asyncio
from collections.abc import Callable
from dataclasses import dataclass, field
from typing import Any, Callable, Protocol


# ---------------------------------------------------------------- given types
# StageOutput：stage -> orchestrator 的消息（每个 stage 都会产生，包括非 final stage）
@dataclass
class StageOutput:
    request_id: str
    data: Any
    finished: bool  # 仅表示"这个 stage 对这个请求完成了"，不是整个请求完成


# OrchOutput：orchestrator -> 用户 的消息（只来自 final_stage_ids 里的 stage）
@dataclass
class OrchOutput:
    request_id: str
    stage_id: int | None
    data: Any
    finished: bool  # 整个请求的终止标志：每个请求恰好一条 True，且是最后一条
    error: str | None = None


class StageClient(Protocol):
    async def submit(self, request_id: str, inputs: Any) -> None: ...
    async def get_output(self) -> list[StageOutput]: ...
    async def abort(self, request_ids: list[str]) -> None: ...


# ---------------------------------------------------------- implementation
# 每个在途请求一份路由状态；只由 orchestrator 内部的 task 读写（同一个 event loop，无需加锁）。
@dataclass
class ReqState:
    request_id: str
    prompt: Any
    submitted: set[int] = field(default_factory=set)  # 已提交过的 stage
    finished: set[int] = field(default_factory=set)  # 已报告 finished=True 的 stage
    # 每个 stage 累积的全部输出；stage s 完成时整份交给 forward_fn 构造 s+1 的输入
    stage_data: dict[int, list] = field(default_factory=dict)
```


© 2026 kzhou003

```
    pending_final: OrchOutput | None = None  # P02 不用；P03 用来暂扣终止消息


# 线性多 stage 路由（例：S0 -> S1 -> S2，final_stage_ids={0,2}）。
# Task 结构（全部跑在同一个 event loop 上）：
#   N 个 _reader task（每个 stage 一个） --(sid, outs)--> _ready_q --> 1 个 _dispatcher task
#   add_request() --> _req_q --> 1 个 _request_handler task
#   _dispatcher --> _out_q --> next_output()（用户侧）
# 单写者（single-writer）：所有 StageOutput 都汇入一个 _ready_q，由唯一的 dispatcher 串行处理，
# 所以同一请求的 finished 判定 / 转发 / cleanup 不会被另一条 output 交错打断。
class Orchestrator:
    def __init__(
        self,
        stages: list[StageClient],
        final_stage_ids: set[int],
        forward_fn: Callable[[int, Any, list], Any],
    ) -> None:
        self.stages = stages
        self.final_stage_ids = frozenset(final_stage_ids)
        # 最后一个需要运行的 stage：比它更靠后的 stage 对用户没有意义，永远不会被提交
        self.last_stage = max(final_stage_ids)
        self.forward_fn: Callable[[int, Any, list[Any]], Any] = forward_fn
        self.request_states: dict[str, ReqState] = {}
        self.dropped_outputs = 0
        # 三个无界队列：put 永不阻塞，因此 reader / dispatcher 不会因为下游慢而互相卡死
        self._req_q: asyncio.Queue = asyncio.Queue()
        self._ready_q: asyncio.Queue = asyncio.Queue()
        self._out_q: asyncio.Queue = asyncio.Queue()
        self._tasks: list[asyncio.Task] = []

    # ---------------------------------------------------------- public API
    async def start(self) -> None:
        for sid, client in enumerate(self.stages):
            # 每个 stage 独立一个 reader：某个 stage 的 get_output 阻塞不会挡住其他 stage
            self._tasks.append(asyncio.create_task(coro=self._reader(sid, client)))
        self._tasks.append(asyncio.create_task(self._request_handler()))
        # 只有一个 dispatcher —— 这就是"单写者"
        self._tasks.append(asyncio.create_task(self._dispatcher()))

    async def shutdown(self) -> None:
        for t in self._tasks:
            t.cancel()
        # cancel 只是请求取消；gather 等它们真正退出。return_exceptions=True 吞掉 CancelledError，
        # 否则第一个 CancelledError 就会让 shutdown 自己抛出、其余 task 没被等待
        await asyncio.gather(*self._tasks, return_exceptions=True)

    # 用户侧调用：只是投递消息，真正的注册/提交在 request handler task 里做
    async def add_request(self, request_id: str, prompt: Any) -> None:
        await self._req_q.put(item=("add", request_id, prompt))

    async def next_output(self, timeout: float | None = None) -> OrchOutput:
        return await asyncio.wait_for(self._out_q.get(), timeout)

    # ---------------------------------------------------------- internals
    # 每个 stage 一个 reader task：只搬运，不碰 request_states（避免多写者）
    async def _reader(self, sid: int, client: StageClient) -> None:
        while True:
            outs = await client.get_output()
            # 带上 sid 一起入队：dispatcher 需要知道是哪个 stage 的输出
            await self._ready_q.put((sid, outs))

    # 只在 request handler task 里运行
    async def _request_handler(self) -> None:
        while True:
            kind, rid, payload = await self._req_q.get()
            if kind == "add":
                await self._handle_add(rid, prompt=payload)

    async def _handle_add(self, rid: str, prompt: Any) -> None:
        st: ReqState = ReqState(rid, prompt)
        # 易错：必须在第一个 await 之前注册。await submit 会让出控制权给 dispatcher；
        # 一个很快的 stage 可能在 submit 返回前就已经吐出输出，reader -> _ready_q -> dispatcher
        # 处理时如果查不到 ReqState，输出会被当成未知请求丢弃（dropped_outputs++），
        # 用户永远等不到 finished=True -> hang。
        self.request_states[rid] = st  # register BEFORE the first await
        # 同理，submitted 也在 await 之前标记，dispatcher 看到的状态始终一致
        st.submitted.add(0)
        await self.stages[0].submit(rid, prompt)

    # 只在 dispatcher task 里运行（单写者）：按到达顺序逐条处理，同一请求的输出不会并发处理
    async def _dispatcher(self) -> None:
        while True:
            sid, outs = await self._ready_q.get()
            for o in outs:
                # 逐条 await：保证同一批内的顺序（例如 "hi" 一定在 "there"(finished) 之前处理）
                await self._handle_output(sid, o)

    # 只在 dispatcher task 里运行。
```


© 2026 kzhou003

```
    # 处理一条 StageOutput：记录 -> （若 final stage）发给用户 -> （若该 stage 完成）转发给下游
    async def _handle_output(self, sid: int, o: StageOutput) -> None:
        st = self.request_states.get(o.request_id)
        if st is None:
            # 请求已 cleanup（终止消息已发出）或从未注册：丢弃并计数。
            # 易错：绝不能在这里新建状态——那会"复活"已结束的请求，造成重复终止消息或状态泄漏
            self.dropped_outputs += 1
            return
        # 先累积数据：stage 完成时 forward_fn 需要该 stage 产生的全部数据（含这条 finished 的最后一块）
        st.stage_data.setdefault(sid, []).append(o.data)
        if o.finished:
            st.finished.add(sid)
        if sid in self.final_stage_ids:
            # 易错：OrchOutput.finished 看的是"所有 final stage 都完成了"，而不是 o.finished。
            # 例：final={0,2}，S0 的 "there"(finished=True) 到达时 finished={0}，{0,2} 未全完成
            # -> 发给用户的是 finished=False；只有 S2 的最后一块才是 True。
            done = self.final_stage_ids <= st.finished
            await self._out_q.put(OrchOutput(o.request_id, sid, o.data, finished=done))
            if done:
                # 终止消息已发：立刻移除状态，之后同一请求的任何输出都走上面的 dropped 分支，
                # 从而保证"恰好一条 finished=True"
                self._cleanup(o.request_id)
                # 不再往下走转发逻辑：请求已结束，不能再给下游 submit（否则产生孤儿工作）
                return
        # 非 final stage（如 S1）的输出不发给用户，只用于触发下游。
        # 只有 stage 完成、且后面还有需要运行的 stage 时才转发；last_stage 之后的 stage 不提交
        if o.finished and sid < self.last_stage:
            nxt = sid + 1
            # 把 stage sid 的"全部"数据交给 forward_fn（不是只有最后一条 o.data）
            inputs = self.forward_fn(nxt, st.prompt, st.stage_data[sid])
            # 在 await 之前标记 submitted，理由同 _handle_add
            st.submitted.add(nxt)
            # 这个 await 期间 dispatcher 挂起，其他输出暂存在 _ready_q 里，不会交错处理
            await self.stages[nxt].submit(o.request_id, inputs)

    # 移除请求状态；pop(..., None) 使重复调用无害
    def _cleanup(self, rid: str) -> None:
        self.request_states.pop(rid, None)
```

C-10 · solutions/p03_async_chunk_orchestrator.py

```
from collections.abc import Callable
from typing import Any

from .p02_orchestrator import Orchestrator, OrchOutput, ReqState, StageClient, StageOutput

__all__ = ["AsyncChunkOrchestrator", "OrchOutput", "StageOutput"]


# async_chunk 模式：stage 之间通过 side channel（connector）直接传数据，orchestrator 不再转发。
# 它只负责两件事：
#   1. add 时一次性 prewarm 所有下游 stage（让它们提前就绪，等 connector 喂数据）；
#   2. 守住终止消息：finished=True 只能在"所有已提交的 stage 都完成"之后发出。
# task 结构、单写者 dispatcher、reader 等全部复用 P02，只覆写 _handle_add / _handle_output。
class AsyncChunkOrchestrator(Orchestrator):
    """Stages exchange data on a side channel; the orchestrator only prewarms
    downstream stages and gates the terminal message."""

    def __init__(
        self,
        stages: list[StageClient],
        final_stage_ids: set[int],
        prewarm_fn: Callable[[int, Any], Any],
    ) -> None:
        # forward_fn 用不到（不再由 orchestrator 转发），传一个占位函数
        super().__init__(stages, final_stage_ids, forward_fn=lambda *_: None)
        self.prewarm_fn = prewarm_fn

    # 只在 request handler task 里运行
    async def _handle_add(self, rid: str, prompt: Any) -> None:
        st = ReqState(rid, prompt)
        # 与 P02 相同：在任何 await 之前注册，否则早到的输出会被 dropped -> hang
        self.request_states[rid] = st
        # 易错：一次性把 0..last_stage 全部标成 submitted，并且要在任何 await 之前。
        # submitted 是 everything_done 的比较基准：如果边 submit 边 add，dispatcher 可能在中途
        #  （某个 await submit 期间）看到不完整的 submitted 集合，误判"所有 stage 都完成了"，
        # 从而过早放出终止消息（之后剩余 stage 的完成报告只能被 drop）。
        st.submitted.update(range(self.last_stage + 1))  # mark before any await
        await self.stages[0].submit(rid, prompt)
        # 下游 stage 收到的是占位输入（prewarm），真实数据之后通过 connector 到达
        for sid in range(1, self.last_stage + 1):
            await self.stages[sid].submit(rid, self.prewarm_fn(sid, prompt))

    # 只在 dispatcher task 里运行（单写者）。
```


© 2026 kzhou003

```
    # 注意：这里不再累积 stage_data，也不转发——数据走 connector。
    async def _handle_output(self, sid: int, o: StageOutput) -> None:
        st = self.request_states.get(o.request_id)
        if st is None:
            # 已 cleanup 的请求：丢弃，绝不复活状态（否则可能发出第二条终止消息）
            self.dropped_outputs += 1
            return
        if o.finished:
            st.finished.add(sid)
        # 两个不同的"完成"概念：
        #   all_final       = 所有 final stage 完成 -> 决定 OrchOutput.finished 的值
        #   everything_done = 所有已提交 stage（含非 final，如 S1 talker）完成 -> 决定终止消息能否放行
        everything_done = st.submitted <= st.finished
        if sid in self.final_stage_ids:
            all_final = self.final_stage_ids <= st.finished
            msg = OrchOutput(o.request_id, sid, o.data, finished=all_final)
            if all_final and not everything_done:
                # 例：final={0,2}，S2 最后一块先到，但非 final 的 S1 还没报告完成。
                # 若此时直接发出并 cleanup，S1 的完成报告会被 drop，而 S1 可能仍在占资源；
                # 所以暂扣终止消息，等剩下的 stage 完成后再发。
                st.pending_final = msg  # hold the terminal until every stage finished
                return
            # 非终止消息直接发；或者 all_final 且 everything_done，终止消息直接发
            await self._out_q.put(msg)
            if all_final:
                # 终止消息已发：cleanup，之后的输出全部走 dropped 分支 -> 保证恰好一条 finished=True
                self._cleanup(o.request_id)
            return
        # 走到这里说明是非 final stage（如 S1）：它的输出从不发给用户。
        # 若它的完成让 everything_done 成立，且终止消息正被暂扣，则现在放行。
        # 不变式：everything_done 蕴含 all_final；最后一个 final stage 完成时，若尚未 everything_done
        # 必然已设置 pending_final；若当时已 everything_done，则已在上面直接发出并 cleanup。
        if everything_done and st.pending_final is not None:
            # 放行的仍是那条最后的 final stage 输出 -> "终止消息永远是最后一条"
            await self._out_q.put(st.pending_final)
            self._cleanup(o.request_id)
```

C-11 · solutions/p01_output_sink.py

```
import asyncio
import threading
from collections import deque


# OutputSink：多个 producer 线程（比如各 stage 的 output 线程）-> 一个 asyncio consumer。
# 核心难点有两个：
#   1. 跨线程唤醒 event loop 只能用 loop.call_soon_threadsafe（asyncio.Event 不是线程安全的）；
#   2. "合并唤醒"（coalesced wakeup）：一个 wake cycle 内最多调度一次 call_soon_threadsafe，
#      同时绝不能丢唤醒（lost wakeup）—— consumer 不能在有数据时睡下去。
# 状态分工：
#   _items          数据本身；deque 的 append / popleft 在 CPython 下是原子的，所以不加锁
#   _wake_scheduled "唤醒已在途"标志；True 表示 consumer 会自己再 drain 一次，put 无需再唤醒
#   _lock           只保护 _wake_scheduled + wakeups_scheduled 这对状态（以及读 loop/event 快照）
class OutputSink:
    """Many producer threads -> one asyncio consumer, with coalesced wakeups."""

    def __init__(self) -> None:
        self._items: deque = deque()
        self._lock = threading.Lock()
        # 初始为 False：第一次 put 一定会调度唤醒
        self._wake_scheduled = False
        self._loop: asyncio.AbstractEventLoop | None = None
        self._event: asyncio.Event | None = None
        self.wakeups_scheduled = 0
        # test hook: called right after a drain found nothing, before re-arming
        self.after_empty_drain = lambda: None

    # 在 consumer 所在的 loop 上调用（默认取当前 running loop）。
    # Event 必须在这个 loop 里创建/使用，之后只允许 loop 线程直接操作它。
    def bind_loop(self, loop: asyncio.AbstractEventLoop | None = None) -> None:
        self._loop = loop or asyncio.get_running_loop()
        self._event = asyncio.Event()

    # 任意 producer 线程调用；不阻塞（锁只包住几条赋值，持有时间极短）。
    def put(self, item) -> None:
        # 易错：必须先 append 再看标志。consumer 的不变式是"清标志之后一定再 drain 一次"，
        # 所以只要数据先入队，不管标志此刻是 True 还是 False，这条数据都不会被漏掉：
        #   - 看到 True：consumer 还没重新武装，它之后的 drain（含武装后那次）一定能看到 item
        #   - 看到 False：由本次 put 负责调度唤醒
        # 反过来（先看标志再 append）就会出现：看到 True 返回 -> consumer 武装并二次 drain
        # 时 item 还没进队 -> consumer 睡下 -> item 卡住（lost wakeup / hang）。
        self._items.append(item)  # append BEFORE looking at the flag
        with self._lock:
            if self._wake_scheduled:
```


© 2026 kzhou003

```
                # 唤醒已在途（或 consumer 正忙、尚未重新武装）：合并，不再调度
                return
            self._wake_scheduled = True
            self.wakeups_scheduled += 1
            # 在锁内取快照，避免与 bind_loop 并发时读到一半的 (loop, event)
            loop, event = self._loop, self._event
        # call_soon_threadsafe 放在锁外：不在持锁时调用进 loop 内部（缩短持锁时间、避免潜在死锁）
        if loop is not None and event is not None:
            try:
                # 易错：不能在 producer 线程里直接 event.set()——asyncio.Event 不是线程安全的，
                # 也不会唤醒正阻塞在 selector 上的 loop。call_soon_threadsafe 会写 self-pipe 唤醒 loop，
                # 再在 loop 线程里执行 event.set。
                loop.call_soon_threadsafe(event.set)
            except RuntimeError:  # loop already closed
                # 关停期间 loop 已关闭：没人会再消费，静默丢弃唤醒即可
                pass
        # 若尚未 bind_loop：标志已置 True 但没有真正唤醒；没关系，get_batch 首次 drain 就能拿到数据

    # 只在 consumer（loop 线程）里调用；单消费者，所以 "while self._items" 后 popleft 不会失败
    def _drain(self, max_items: int) -> list:
        out = []
        while self._items and len(out) < max_items:
            out.append(self._items.popleft())
        return out

    # 只在绑定的 loop 上运行（单消费者）。返回 1..max_items 条，或超时返回 []。
    async def get_batch(self, max_items: int = 256, timeout: float | None = None) -> list:
        if self._loop is None:
            self.bind_loop()
        loop, event = self._loop, self._event
        assert loop is not None and event is not None
        # 用绝对 deadline：被虚假唤醒（stale set）后再次循环时，剩余时间不会被重置
        deadline = None if timeout is None else loop.time() + timeout
        while True:
            # 快路径：有数据直接返回，不武装、不清标志。
            # 此时 _wake_scheduled 保持 True，consumer 忙着处理时 producer 不会再发多余唤醒；
            # 下一次 get_batch 一定会先 drain，所以这些数据不会丢。
            out = self._drain(max_items)
            if out:
                return out
            self.after_empty_drain()
            # arm: the next put() must schedule a wakeup
            # 先 clear：丢掉上一个 cycle 遗留的 set，否则下面 wait 会立刻返回（空转一圈）。
            # clear 与清标志之间没有 await，loop 上的 event.set 回调插不进来，所以这里的顺序是安全的。
            event.clear()
            with self._lock:
                # 清标志 = 重新武装：从这一刻起的第一次 put 会调度唤醒
                self._wake_scheduled = False
            # drain again: closes the "put between drain and arm" race
            # 易错：这次二次 drain 不能省。窗口：第一次 drain 为空 -> producer append 并看到标志仍为 True
            # 而直接返回（没调度唤醒）-> 我们清标志 -> 若此时直接 wait，就会带着数据睡死（hang）。
            # 测试正是用 after_empty_drain 在这个窗口里注入 put。
            out = self._drain(max_items)
            if out:
                # 返回时标志是 False（已武装），下一次 put 会多调度一次唤醒——只是一次额外 wakeup，
                # 正确性不受影响。
                return out
            remaining: float | None = None if deadline is None else deadline - loop.time()
            if remaining is not None and remaining <= 0:
                return []
            try:
                # await 期间 producer 的 put 会通过 call_soon_threadsafe(event.set) 把我们唤醒
                await asyncio.wait_for(event.wait(), remaining)
            except asyncio.TimeoutError:
                # 超时那一刻可能恰好有数据进来（set 回调还没跑），最后再 drain 一次而不是直接返回 []
                return self._drain(max_items)
            # 被唤醒后回到循环顶部 drain；如果是虚假唤醒（拿不到数据），会重新武装再睡
```

C-12 · solutions/p04_stage_pool_abort.py

```
from dataclasses import dataclass
from typing import Any, Protocol


# ---------------------------------------------------------------- given types
class NoLiveReplica(Exception):
    pass


class ReplicaClient(Protocol):
    async def abort(self, request_ids: list[str]) -> None: ...


class PartialProcessor(Protocol):
```


© 2026 kzhou003

```
    """Detokenizer-like state holder living on the orchestrator side."""

    def collect_partial(self, request_id: str) -> Any: ...  # read-only, no state change
    def commit_abort(self, request_id: str) -> None: ...    # drop the request's state


@dataclass
class OrchOutput:
    request_id: str
    data: Any
    finished: bool
    error: str | None = None


# ---------------------------------------------------------- implementation
# 一个 stage 的多副本池（replica pool）。所有方法都在 orchestrator 的 event loop 上调用：
# 同步方法（pick / evict / release）之间不会交错；只有 abort 里的 await 是让出点。
# 状态：
#   _alive     每个 replica 是否存活
#   _bindings  request_id -> replica_id 的粘性绑定（请求的 KV cache / 生成状态都在那个 replica 上）
#   _rr        round-robin 游标
class StagePool:
    def __init__(self, stage_id: int, replicas: list[ReplicaClient]) -> None:
        self.stage_id = stage_id
        self.replicas = replicas
        self._alive = [True] * len(replicas)
        self._bindings: dict[str, int] = {}
        self._rr = 0

    def live_replicas(self) -> list[int]:
        return [i for i, ok in enumerate(self._alive) if ok]

    # 为请求选择 replica：已绑定且存活 -> 粘住原来的；否则在存活 replica 上 round-robin 并绑定。
    def pick(self, request_id: str) -> int:
        bound = self._bindings.get(request_id)
        # 易错：粘性。同一请求的后续调用必须去同一个 replica，否则那边没有它的状态
        if bound is not None and self._alive[bound]:
            return bound
        # 绑定的 replica 已死（正常情况下 evict 已删除绑定，这里是兜底）：落到重新挑选
        live = self.live_replicas()
        if not live:
            raise NoLiveReplica(f"stage {self.stage_id} has no live replica")
        # 对"存活列表"取模，而不是对全部 replica 取模，保证不会选到死副本
        rid = live[self._rr % len(live)]
        self._rr += 1
        self._bindings[request_id] = rid
        return rid

    def bound(self, request_id: str) -> int | None:
        return self._bindings.get(request_id)

    # 请求正常结束时释放绑定；不释放会让 _bindings 无限增长（leak）
    def release(self, request_id: str) -> None:
        self._bindings.pop(request_id, None)

    # replica 挂了：先标记死亡（之后 pick 不会再选它），再摘掉并返回绑在它上面的请求，
    # 由调用方给这些请求发终止错误。绑定在其他 replica 上的请求不受影响。
    def evict(self, replica_id: int) -> list[str]:
        self._alive[replica_id] = False
        # 先收集再删除：不能边遍历 dict 边删（RuntimeError: dict changed size during iteration）
        affected = [r for r, rep in self._bindings.items() if rep == replica_id]
        for r in affected:
            del self._bindings[r]
        return affected

    # 按 replica 分组：每个 replica 只发一次 abort（批量），而不是每个请求一次 RPC。
    async def abort(self, request_ids: list[str]) -> None:
        groups: dict[int, list[str]] = {}
        for r in request_ids:
            rep = self._bindings.get(r)
            # 没有绑定的 id（从未派发到本 stage、或已释放 / 已 evict）直接忽略
            if rep is not None:
                groups.setdefault(rep, []).append(r)
        for rep, ids in groups.items():
            # 易错：必须等 replica 确认 abort 成功后才释放绑定。
            # 若 abort 抛异常，异常直接向上传播，这些 id 的绑定保留，调用方可以重试；
            # 如果先释放再 abort，失败后就再也不知道它们在哪个 replica 上 -> 引擎侧泄漏、继续生成。
            #  （前面已成功的 replica 的绑定已释放；重试时它们因无绑定被忽略，正好不会重复 abort）
            await self.replicas[rep].abort(ids)  # raises -> bindings are kept
            for r in ids:
                # 用 pop(r, None)：await 期间 evict 等操作可能已经删掉了这个绑定
                self._bindings.pop(r, None)


# 两阶段 abort（two-phase）：先只读收集 partial -> 所有 stage 的引擎确认 abort -> 才 commit。
async def abort_request(pools: list[StagePool], processor: PartialProcessor, request_id: str) -> OrchOutput:
    # 阶段 1：只读。即使后面 abort 失败，processor 状态仍完好，重试时能拿到同样的 partial
    partial = processor.collect_partial(request_id)  # phase 1: read only
```


© 2026 kzhou003

```
    for pool in pools:
        # 任一 pool 失败都直接抛出，跳过 commit
        await pool.abort([request_id])  # any failure propagates before commit
    # 易错：commit 必须放在所有 abort 成功之后。若先 commit_abort 再 abort 且 abort 失败：
    # 引擎还在生成，后续输出到达时 processor 已没有这个请求的状态（报错或被重新创建 = 复活），
    # 而且重试时 partial 已经丢了。
    processor.commit_abort(request_id)  # phase 2: commit only after engines confirmed
    # 终止消息：finished=True + error="abort"，带上已生成的部分结果给用户
    return OrchOutput(request_id, partial, finished=True, error="abort")


# 死副本隔离：只给绑在该 replica 上的请求各发一条终止错误；其他请求照常运行。
# 每个受影响请求恰好一条终止消息——evict 已删掉它们的绑定，重复调用不会再返回它们。
def handle_dead_replica(pool: StagePool, replica_id: int) -> list[OrchOutput]:
    affected = pool.evict(replica_id)
    return [
        OrchOutput(r, None, finished=True, error=f"stage {pool.stage_id} replica {replica_id} died")
        for r in affected
    ]
```

C-13 · solutions/p05_rpc_router.py

```
import queue
import threading
from collections.abc import Callable, Hashable
from typing import Any


# RpcRouter：多个调用方线程共用一条结果通道（例如一个线程统一收各 worker 的回包），
# 用 key 把回包关联到正在阻塞等待的调用方。所有方法都可能从任意线程调用。
# 状态（全部由 _lock 保护）：
#   _waiters  key -> 只容纳一个结果的 queue.Queue（每个在途调用一个）
#   _fatal    fail_all 锁存的致命错误；一旦设置，之后的 register / call 立即失败
class RpcRouter:
    """Correlates results coming back on one shared channel with blocked callers."""

    def __init__(self) -> None:
        self._lock = threading.Lock()
        self._waiters: dict[Hashable, queue.Queue] = {}
        self._fatal: BaseException | None = None
        self.late_results = 0

    # 调用方线程：登记一个等待者
    def register(self, key: Hashable) -> queue.Queue:
        with self._lock:
            # 易错：_fatal 检查必须和插入 _waiters 在同一把锁内。否则 fail_all 可能恰好在
            # "检查通过"与"插入"之间清空 _waiters，新插入的 waiter 永远没人唤醒，只能等到超时
            if self._fatal is not None:
                raise self._fatal
            # 同一个 key 两个在途调用会互相抢结果，直接拒绝
            if key in self._waiters:
                raise ValueError(f"duplicate rpc key {key!r}")
            # maxsize=1：每次调用恰好一个结果
            waiter: queue.Queue = queue.Queue(maxsize=1)
            self._waiters[key] = waiter
            return waiter

    # 调用方线程（call 的 finally）：移除自己的 waiter
    def unregister(self, key: Hashable, waiter: queue.Queue) -> None:
        with self._lock:
            # 易错：按身份（is）比较再删。这个 key 可能已被 deliver / fail_all 弹出，
            # 并被一个新的调用重新注册；直接 pop(key) 会误删别人的 waiter，导致对方 hang 到超时
            if self._waiters.get(key) is waiter:
                del self._waiters[key]

    # 结果通道线程：把结果交给对应 waiter
    def deliver(self, key: Hashable, result: Any) -> bool:
        with self._lock:
            # pop 而不是 get：一个 waiter 最多被交付一次；deliver 与 fail_all 也在同一把锁下互斥，
            # 二者只有一个能拿到它
            waiter = self._waiters.pop(key, None)
            if waiter is None:
                # 调用方已超时 / 已 unregister，或 key 从未注册：迟到的结果，计数后丢弃
                #  （在锁内自增，多线程下计数不会丢）
                self.late_results += 1
                return False
        # 在锁外 put：waiter 已从字典里摘掉，只有我们持有它；maxsize=1 且只 put 一次，不会 Full。
        # 边界：若调用方恰好在这之前超时，结果会落进没人再读的队列而被丢弃——可接受
        waiter.put_nowait(result)
        return True

    # 致命错误（如 worker 进程死亡）：唤醒所有在途调用，并锁存错误让后续调用快速失败
    def fail_all(self, error: BaseException) -> None:
        with self._lock:
```


© 2026 kzhou003

```
            # 锁存与清空在同一把锁内完成：之后的 register 一定能看到 _fatal
            self._fatal = error
            waiters = list(self._waiters.values())
            self._waiters.clear()
        # 锁外逐个唤醒：放入的是异常对象本身，call 端识别后 raise
        for w in waiters:
            try:
                w.put_nowait(error)
            except queue.Full:
                # 防御性：waiter 已被摘出字典、不会再被 deliver，正常不会满
                pass

    # 调用方线程：注册 -> 发送 -> 阻塞等待结果
    def call(self, key: Hashable, send: Callable[[], None], timeout: float) -> Any:
        # 易错：必须先 register 再 send。回包在另一个线程上到达，可能比 send() 返回还快；
        # 若先 send，回包到达时找不到 waiter -> 计为 late_results 被丢弃 -> 本调用白等到超时
        waiter = self.register(key)  # register BEFORE sending
        try:
            send()
            try:
                result = waiter.get(timeout=timeout)
            except queue.Empty:
                # from None：隐藏内部的 queue.Empty，调用方只看到 TimeoutError
                raise TimeoutError(f"rpc {key!r} timed out") from None
            # fail_all 放进来的是异常对象：在调用方线程里重新抛出
            if isinstance(result, BaseException):
                raise result
            return result
        finally:
            # 无论成功 / 超时 / send 抛异常都要清理，否则 _waiters 泄漏，且同 key 再次调用会报 duplicate；
            # 超时后清理还保证之后迟到的结果被正确计为 late_results
            self.unregister(key, waiter)
```

C-14 · solutions/p10_engine_core_loop.py

```
import queue
import threading
from collections.abc import Callable
from dataclasses import dataclass
from typing import Any, Literal, Protocol


# ------------------------------------------------------------------ given types
@dataclass
class Msg:
    kind: Literal["ADD", "ABORT", "UTILITY", "WAKEUP"]
    payload: Any = None


class Engine(Protocol):
    def add(self, req_id: str, data: Any) -> None: ...
    def abort(self, req_ids: list[str]) -> None: ...
    def has_work(self) -> bool: ...
    def step(self) -> list[Any]: ...
    def utility(self, name: str, args: tuple) -> Any: ...


# ------------------------------------------------------------------ implementation
# EngineCoreLoop：EngineCore 进程的主循环（busy loop）。
# 输入线程把 Msg 放进 input_queue，主循环每轮：
#   空闲时阻塞等消息 -> 按 FIFO 处理完所有已排队消息 -> 有活就 step 一次并 emit 输出。
# 输出统一经 emit(tuple) 发出：("output", x) / ("utility", call_id, result, err) / ("fatal", repr) / ("shutdown",)。
class EngineCoreLoop:
    """The busy loop of an EngineCore process (IO threads feed input_queue / drain emit)."""

    def __init__(self, engine: Engine, emit: Callable[[tuple], None]) -> None:
        self.engine = engine
        self.emit = emit
        # 线程安全的 FIFO 队列，是其他线程与主循环之间唯一的输入通道。
        self.input_queue: queue.Queue[Msg] = queue.Queue()
        # 关闭标志：threading.Event 可跨线程安全地 set/查询。
        self._shutdown = threading.Event()

    # 请求关闭：先置标志，再塞一个 WAKEUP，确保正阻塞在 input_queue.get() 的主循环被唤醒并看到标志。
    def request_shutdown(self) -> None:
        """Safe from any thread or a signal handler."""
        self._shutdown.set()
        # 易错：若只 set 标志，空闲时主循环永远卡在 get() 上；put_nowait 不会阻塞，可在 signal handler 中调用。
        self.input_queue.put_nowait(Msg("WAKEUP"))

    # 分发一条输入消息到 engine。
    def _handle(self, msg: Msg) -> None:
        if msg.kind == "ADD":
            req_id, data = msg.payload
```


© 2026 kzhou003

```
            self.engine.add(req_id, data)
        elif msg.kind == "ABORT":
            # payload 可能是任意可迭代对象，统一转成 list。
            self.engine.abort(list(msg.payload))
        elif msg.kind == "UTILITY":
            call_id, name, args = msg.payload
            # utility 调用（RPC 式）：成功返回（result, None），失败返回（None, repr(e)），用 call_id 让调用方配对。
            try:
                self.emit(("utility", call_id, self.engine.utility(name, args), None))
            except Exception as e:  # utility failure is call-scoped, not fatal
                self.emit(("utility", call_id, None, repr(e)))
        # WAKEUP: nothing to do, it only unblocks get()

    def run_busy_loop(self) -> None:
        try:
            while not self._shutdown.is_set():
                # 空闲（无请求在跑）时阻塞等待下一条消息，避免空转占 CPU。
                if not self.engine.has_work():
                    self._handle(self.input_queue.get())  # block while idle: no spinning
                # 非阻塞地把队列里已有的消息全部处理掉，保证 ADD/ABORT 在下一次 step 之前生效。
                while True:  # drain everything already queued, in FIFO order
                    try:
                        self._handle(self.input_queue.get_nowait())
                    except queue.Empty:
                        break
                # 处理消息期间可能收到了关闭请求：此时不再 step，直接退出。
                if self._shutdown.is_set():
                    break
                # 易错：要再次检查 has_work——刚处理的可能只是 WAKEUP/UTILITY/ABORT，此时没有可 step 的请求。
                if self.engine.has_work():
                    for out in self.engine.step():
                        self.emit(("output", out))
        # engine.add/step 等抛出的异常视为致命：报告 fatal 后退出，且不再发 shutdown。
        except Exception as e:
            self.emit(("fatal", repr(e)))
            return
        # 正常关闭路径：最后发出一次 shutdown 通知。
        self.emit(("shutdown",))
```

C-15 · solutions/p11_chunk_aware_scheduler.py

```
from collections import deque
from dataclasses import dataclass
from typing import Protocol


# ---------------------------------------------------------------- given types
@dataclass
class Req:
    req_id: str
    arrival: int
    needs_chunk: bool = False  # receives its input from an upstream stage chunk by chunk
    status: str = "WAITING"


class BaseScheduler:
    """Stand-in for the upstream vLLM scheduler: knows nothing about chunks."""

    def __init__(self) -> None:
        self.waiting: deque[Req] = deque()
        self.running: list[Req] = []

    def schedule(self) -> list[str]:
        for r in self.running + list(self.waiting):
            if r.status in ("WAITING_FOR_CHUNK", "ABORTED"):
                raise AssertionError(f"upstream saw {r.req_id} in state {r.status}")
        out = [r.req_id for r in self.running] + [r.req_id for r in self.waiting]
        for r in self.waiting:
            r.status = "RUNNING"
        self.running += list(self.waiting)
        self.waiting.clear()
        return out


class ChunkSource(Protocol):
    def load_async(self, req_id: str) -> None: ...
    def is_ready(self, req_id: str) -> bool: ...


# ---------------------------------------------------------------- implementation
# ChunkAwareScheduler：包在不懂 chunk 的上游 BaseScheduler 外面的一层。
# 每次 schedule 前把"下一个 chunk 尚未就绪"的 needs_chunk 请求从 base 的队列中摘出来暂存（park），
# 让上游完全看不到它们；chunk 就绪后再按原队列、原状态放回（restore）。
# 上游调度结束（含异常）后在 finally 中再 restore 一次。
```


© 2026 kzhou003

```
class ChunkAwareScheduler:
    def __init__(self, base: BaseScheduler, source: ChunkSource) -> None:
        self.base = base
        self.source = source
        # 暂存区：记录 origin ("waiting"/"running") 和 park 前的 status，restore 时原样还原。
        self.parked: dict[str, tuple[Req, str, str]] = {}  # id -> (req, origin queue, previous status)
        # 本次 park 期间已经调用过 load_async 的请求，保证每次 park 只触发一次加载。
        self._requested: set[str] = set()

    def add(self, req: Req) -> None:
        self.base.waiting.append(req)

    # abort 可在任何时刻调用：无论请求处在 parked、waiting 还是 running，都要彻底移除，之后不能再被调度或 restore。
    def abort(self, req_id: str) -> None:
        # 从暂存区移除，这样 _restore_ready 就不会再把它放回去。
        entry = self.parked.pop(req_id, None)
        if entry is not None:
            entry[0].status = "ABORTED"
        # 遍历拷贝（list(...)），避免边遍历边删除。
        for r in list(self.base.waiting):
            if r.req_id == req_id:
                self.base.waiting.remove(r)
                r.status = "ABORTED"
        for r in list(self.base.running):
            if r.req_id == req_id:
                self.base.running.remove(r)
                r.status = "ABORTED"
        self._requested.discard(req_id)

    # park：扫描 waiting 与 running，把 chunk 未就绪的 needs_chunk 请求移入 parked。
    def _park_not_ready(self) -> None:
        # 先对两个队列做快照，再在原队列上 remove，避免遍历时修改容器。
        for origin, q in (("waiting", list(self.base.waiting)), ("running", list(self.base.running))):
            for r in q:
                if r.needs_chunk and not self.source.is_ready(r.req_id):
                    (self.base.waiting.remove if origin == "waiting" else self.base.running.remove)(r)
                    # 易错：必须在改 status 之前保存原 status，否则 restore 时会还原成 WAITING_FOR_CHUNK。
                    self.parked[r.req_id] = (r, origin, r.status)
                    r.status = "WAITING_FOR_CHUNK"
                    # 每次 park 只发起一次 load_async（_requested 在 restore/abort 时清除）。
                    if r.req_id not in self._requested:
                        self._requested.add(r.req_id)
                        self.source.load_async(r.req_id)

    # restore：把 chunk 已就绪的暂存请求放回原队列、恢复原 status。
    def _restore_ready(self) -> None:
        # 先收集再删除：不能在遍历 dict 的同时 del。
        ready = [e for rid, e in self.parked.items() if self.source.is_ready(rid)]
        # 同一批放回的请求按 arrival 排序，保持到达顺序（FCFS），而不是 park 的先后顺序。
        for r, origin, prev in sorted(ready, key=lambda e: e[0].arrival):
            del self.parked[r.req_id]
            # 清掉"已请求"标记，下次再被 park 时会重新调用 load_async 加载下一个 chunk。
            self._requested.discard(r.req_id)
            r.status = prev
            if origin == "waiting":
                self.base.waiting.append(r)
            else:
                self.base.running.append(r)

    # 顺序：(a) restore 已就绪 -> (b) park 未就绪 -> (c) 上游调度 -> (d) finally 中再 restore。
    def schedule(self) -> list[str]:
        # 先 restore 再 park：刚就绪的请求本轮就能被调度；仍未就绪的会在下一步被重新 park。
        self._restore_ready()
        self._park_not_ready()
        try:
            return self.base.schedule()
        # 易错：上游 schedule 抛异常时也必须 restore，否则就绪的请求会一直卡在 parked 里。
        finally:
            self._restore_ready()  # runs even if upstream raised
```

C-16 · solutions/p07_scheduler.py

```
from collections import deque
from collections.abc import Callable
from dataclasses import dataclass, field


# Seq：调度器视角下的一个请求。只记账 token 数与 block 数，不持有真实 KV。
@dataclass
class Seq:
    seq_id: str
    prompt_len: int
    max_tokens: int
    # 已生成的 output token。
```


© 2026 kzhou003

```
    output: list[int] = field(default_factory=list)
    # num_computed: 已写入 KV cache 的 token 数; num_tokens - num_computed 即还需计算的 token 数。
    # prefill 未完成时 > 1 (chunked prefill), decode 阶段恒为 1。
    num_computed: int = 0
    # 当前已占用的 KV block 数 (只增不减, 直到 release/preempt 归零)。
    num_blocks: int = 0
    status: str = "WAITING"  # WAITING | RUNNING | FINISHED

    # 总 token 数 = prompt + 已生成; 被 preempt 后 recompute 也需要覆盖这些全部 token。
    @property
    def num_tokens(self) -> int:
        return self.prompt_len + len(self.output)


# Scheduler: continuous batching 调度器。
# 每一步在 token budget (max_num_batched_tokens) 内: 先给 running 中的请求分配 (decode 或继续 chunked prefill),
# 再按 FCFS 从 waiting 中接纳新请求; KV block 不够时按 recompute 方式 preempt 最晚加入 running 的请求。
class Scheduler:
    """Continuous batching with chunked prefill and recompute preemption."""

    def __init__(self, num_blocks: int, block_size: int, max_num_seqs: int, max_num_batched_tokens: int) -> None:
        self.num_blocks = num_blocks
        self.block_size = block_size
        self.max_num_seqs = max_num_seqs
        self.max_num_batched_tokens = max_num_batched_tokens
        # waiting: FCFS 队列; 被 preempt 的请求放回队头, 优先被重新接纳。
        self.waiting: deque[Seq] = deque()
        # running: 按接纳顺序排列, 越靠后优先级越低 (preempt 的 victim 从尾部取)。
        self.running: list[Seq] = []
        self.finished: list[Seq] = []
        # 不变式: blocks_used == sum(s.num_blocks for s in running)。
        self.blocks_used = 0
        self.num_preemptions = 0
        # 上一次 schedule() 的结果 (seq_id -> 本步计算的 token 数), 供 update() 使用。
        self._scheduled: dict[str, int] = {}

    def add(self, seq: Seq) -> None:
        self.waiting.append(seq)

    def has_work(self) -> bool:
        return bool(self.waiting or self.running)

    # ---------------------------------------------------------- helpers
    # ceil(n_tokens / block_size): 容纳 n_tokens 个 token 需要的 block 数。
    def _blocks_for(self, n_tokens: int) -> int:
        return -(-n_tokens // self.block_size)

    # 尝试让 seq 的 block 数扩展到能容纳 n_tokens 个 token; 要么全部成功, 要么不修改任何状态返回 False。
    def _try_grow(self, seq: Seq, n_tokens: int) -> bool:
        need = self._blocks_for(n_tokens) - seq.num_blocks
        # 已有 block 足够 (例如在同一 block 内的 decode), 无需新增。
        if need <= 0:
            return True
        # 原子性: 先检查再修改。
        if self.blocks_used + need > self.num_blocks:
            return False
        self.blocks_used += need
        seq.num_blocks += need
        return True

    # 归还 seq 占用的全部 block。
    def _release(self, seq: Seq) -> None:
        self.blocks_used -= seq.num_blocks
        seq.num_blocks = 0

    # recompute 式 preemption: 丢弃 seq 的全部 KV, 放回 waiting 队头, 之后重新 prefill (含已生成的 output)。
    def _preempt(self, seq: Seq) -> None:
        self.running.remove(seq)
        self._release(seq)
        # 易错: 必须把 num_computed 清零, 否则重新接纳时会以为 KV 仍在。output 保留, 不会丢已生成的 token。
        seq.num_computed = 0  # recompute prompt + already generated tokens later
        seq.status = "WAITING"
        # appendleft: 被抢占的请求比所有 waiting 中的请求都更早到达, 保持 FCFS 公平。
        self.waiting.appendleft(seq)
        self.num_preemptions += 1

    # ---------------------------------------------------------- step
    # 返回本步调度: seq_id -> 本步要计算的 token 数。所有请求本步 token 数之和 <= max_num_batched_tokens。
    def schedule(self) -> dict[str, int]:
        budget = self.max_num_batched_tokens
        scheduled: dict[str, int] = {}

        # 1) running requests first (decode, or continuing a chunked prefill)
        i = 0
        while i < len(self.running) and budget > 0:
            seq = self.running[i]
            # chunked prefill: 剩余未计算的 token 可能超过 budget, 只取 budget 能容纳的一段。
            n = min(seq.num_tokens - seq.num_computed, budget)
```


© 2026 kzhou003

```
            # block 不够时反复 preempt running 尾部（优先级最低）的请求，直到能容纳或 seq 自己被抢占。
            while not self._try_grow(seq, seq.num_computed + n):
                victim = self.running[-1]  # lowest priority = most recently admitted
                self._preempt(victim)
                # victim 位于 running 尾部（下标 >= i），本步尚未被调度；pop 是防御性的，
                # 保证 scheduled 里绝不残留已被抢占的请求。
                scheduled.pop(victim.seq_id, None)
                # 抢到了自己：seq 已不在 running 中，放弃本 seq。
                if victim is seq:
                    break
            # while...else：只有循环条件正常变为 False（即 _try_grow 成功，未 break）时才执行 else。
            else:
                scheduled[seq.seq_id] = n
                budget -= n
                i += 1
                continue
            # 易错：走到这里说明 seq 自己被 preempt 了，running 已缩短，下标 i 处现在是下一个请求，不能 i += 1。
            # seq itself was preempted; self.running shrank, do not advance i

        # 2) waiting requests, strictly FCFS
        # 同时受 token budget 和 max_num_seqs（running 并发上限）约束。
        while self.waiting and budget > 0 and len(self.running) < self.max_num_seqs:
            seq = self.waiting[0]
            n = min(seq.num_tokens - seq.num_computed, budget)
            # 严格 FCFS：队头放不下就停止，不跳过它去调度后面更小的请求（避免饿死队头）。
            # 接纳新请求时也不会为了它去 preempt running 中的请求。
            if not self._try_grow(seq, seq.num_computed + n):
                break
            self.waiting.popleft()
            seq.status = "RUNNING"
            self.running.append(seq)
            scheduled[seq.seq_id] = n
            budget -= n

        self._scheduled = scheduled
        # 返回拷贝，调用方修改不会影响 update() 的依据。
        return dict(scheduled)

    # update：模型 forward 结束后推进状态。只有 prefill 全部完成（num_computed 追上 num_tokens）的 seq 才采样新 token。
    def update(self, sample_fn: Callable[[str], int], eos_token: int) -> list[Seq]:
        """Apply the last schedule(); sample_fn is called only for seqs that produced a token."""
        done: list[Seq] = []
        by_id = {s.seq_id: s for s in self.running}
        for sid, n in self._scheduled.items():
            seq = by_id[sid]
            seq.num_computed += n
            # 中间 chunk 的 logits 没有意义，不能采样。
            if seq.num_computed < seq.num_tokens:
                continue  # chunked prefill not finished yet
            tok = sample_fn(sid)
            # 追加后 num_tokens 比 num_computed 多 1，下一步正好 decode 这 1 个 token。
            seq.output.append(tok)
            # 终止条件：采样到 EOS 或达到 max_tokens；完成即释放 block。
            if tok == eos_token or len(seq.output) >= seq.max_tokens:
                seq.status = "FINISHED"
                self.running.remove(seq)
                self._release(seq)
                self.finished.append(seq)
                done.append(seq)
        # 清空，防止重复 update 同一批调度。
        self._scheduled = {}
        return done
```

C-17 · solutions/p06_block_manager.py

```
from collections import OrderedDict, deque


# 分配失败时抛出；调用方（scheduler）据此决定 preempt 或等待。
class OutOfBlocks(Exception):
    pass


# 链式哈希（hash chain）：每个满 block 的 hash 依赖于前一个 block 的 hash，
# 因此同一个 hash 代表"从序列开头到该 block 为止的整段前缀"，而不仅仅是本 block 的 token。
# 易错：只 hash 本 block 的 tokens 会让不同前缀的相同片段误命中。
def _block_hash(parent: int | None, tokens: tuple[int, ...]) -> int:
    return hash((parent, tokens))


# BlockManager：vLLM 风格的 paged KV cache 分配器。
# - 每个 seq 拥有一个 block table（逻辑 block 下标 -> 物理 block id）。
# - 物理 block 带 refcount，开启 prefix cache 时多个 seq 可以共享同一前缀 block。
# - refcount 归零但仍带 hash 的 block 不立即释放，而是进入 evictable LRU 池，可被后续请求复用；
```


© 2026 kzhou003

```
#   空间不够时才从 evictable 池中按 LRU 驱逐。
class BlockManager:
    """Paged KV-cache allocator with ref counts and optional prefix caching."""

    def __init__(self, num_blocks: int, block_size: int, enable_prefix_cache: bool = False) -> None:
        self.num_blocks = num_blocks
        self.block_size = block_size
        self.enable_prefix_cache = enable_prefix_cache
        # free list：完全空闲（无 hash、ref==0）的 block，FIFO 顺序；初始为 0..N-1，从左侧取。
        self._free: deque[int] = deque(range(num_blocks))
        # 每个物理 block 的引用计数（有多少个 seq 的 block table 包含它）。
        self._ref = [0] * num_blocks
        # 每个物理 block 当前登记的前缀 hash；None 表示未满或未登记（不可被 prefix cache 命中）。
        self._hash: list[int | None] = [None] * num_blocks
        # 前缀 hash -> 物理 block 的反向索引，用于 prefix cache 查找。
        self._hash_to_block: dict[int, int] = {}
        # 不变式：_evictable 中的 block 满足 ref==0 且 _hash[b] 非 None；OrderedDict 头部最老（最先被驱逐）。
        self._evictable: OrderedDict[int, None] = OrderedDict()  # ref==0 but still cached (LRU)
        # seq_id -> block table（物理 block id 列表）。
        self._tables: dict[str, list[int]] = {}
        # seq_id -> 该 seq 目前全部 token（prompt + 已追加的 token），用于计算后续满 block 的 hash。
        self._tokens: dict[str, list[int]] = {}
        # seq_id -> 每个"满" block 的链式 hash（长度 = len(tokens) // block_size）。
        self._seq_hashes: dict[str, list[int]] = {}
        # seq_id -> allocate 时 prefix cache 命中的 token 数（= 命中 block 数 * block_size）。
        self._cached_tokens: dict[str, int] = {}

    # ---------------------------------------------------------- queries
    # 可用 block 数 = 真空闲 + 可驱逐（evictable 的 block 在需要时可以被回收）。
    @property
    def num_free_blocks(self) -> int:
        return len(self._free) + len(self._evictable)

    # 返回拷贝，防止调用方修改内部 block table。
    def block_table(self, seq_id: str) -> list[int]:
        return list(self._tables[seq_id])

    def num_cached_tokens(self, seq_id: str) -> int:
        return self._cached_tokens.get(seq_id, 0)

    # 逻辑位置 pos -> KV cache 中的扁平 slot 编号：物理 block id * block_size + block 内偏移。
    def slot(self, seq_id: str, pos: int) -> int:
        table = self._tables[seq_id]
        return table[pos // self.block_size] * self.block_size + pos % self.block_size

    # -(-a // b) 是向上取整 ceil(a / b) 的整数写法。
    # 注意：这里不考虑 prefix cache 命中，是保守估计。
    def can_allocate(self, num_tokens: int) -> bool:
        return -(-num_tokens // self.block_size) <= self.num_free_blocks

    # ---------------------------------------------------------- internals
    # 取一个新的物理 block：优先用 free list，没有再从 evictable 池驱逐最老的一个。
    def _take_block(self) -> int:
        if self._free:
            b = self._free.popleft()
        elif self._evictable:
            # popitem(last=False) 弹出 LRU 头部，即最久未使用的缓存 block。
            b, _ = self._evictable.popitem(last=False)
            h = self._hash[b]
            # 易错：只有当索引仍指向这个 block 时才删除；同一 hash 可能已被别的 block 登记（setdefault 保留首个）。
            if h is not None and self._hash_to_block.get(h) == b:
                del self._hash_to_block[h]
            # 被驱逐后该 block 内容将被覆盖，必须清掉 hash，否则以后会错误命中。
            self._hash[b] = None
        else:
            raise OutOfBlocks
        self._ref[b] = 1
        return b

    # 把一个已写满的 block 登记到 prefix cache 索引中（未开启 prefix cache 时什么也不做）。
    def _register(self, b: int, h: int) -> None:
        if not self.enable_prefix_cache:
            return
        self._hash[b] = h
        # setdefault：若同一前缀已有 block 登记，保留旧的，不覆盖索引。
        self._hash_to_block.setdefault(h, b)

    # ---------------------------------------------------------- mutation
    # allocate：为新 seq 的 prompt 分配 block table。
    # 算法：1）计算所有满 block 的链式 hash；2）开启 prefix cache 时从头匹配最长连续命中前缀；
    # 3）先检查容量（全有或全无），再一次性执行：复用命中 block（ref+1）+ 为剩余部分取新 block。
    def allocate(self, seq_id: str, token_ids: list[int]) -> list[int]:
        assert seq_id not in self._tables
        bs = self.block_size
        # 总共需要的 block 数（含最后一个未满 block）。
        n_blocks = -(-len(token_ids) // bs)
        hashes: list[int] = []
        parent = None
```


© 2026 kzhou003

```
        # 只对满 block 计算 hash；未满的尾 block 内容还会变化，不能参与缓存。
        for i in range(len(token_ids) // bs):
            parent = _block_hash(parent, tuple(token_ids[i * bs:(i + 1) * bs]))
            hashes.append(parent)

        hits: list[int] = []
        if self.enable_prefix_cache:
            # 易错：即使 prompt 完全命中，也必须至少留 1 个 token 给模型计算，才能产出下一个 token 的 logits。
            # 所以最多匹配 (len-1)//bs 个 block；例如 len=8, bs=4 时只能命中 1 个 block。
            matchable = (len(token_ids) - 1) // bs  # always leave >= 1 token to compute
            for h in hashes[:matchable]:
                b = self._hash_to_block.get(h)
                # 前缀匹配必须连续：一旦某个 block 未命中，后面的 block 即使命中也不能用（链式 hash 下其实也不会命中）。
                if b is None:
                    break
                hits.append(b)

        # 命中的 block 中有些正处在 evictable 池里（ref==0），它们被复用后就不再可供分配，
        # 所以可用于新 block 的数量要扣掉它们，避免重复计数。
        hits_from_evictable = sum(1 for b in hits if self._ref[b] == 0)
        # 原子性：在任何状态修改之前先做容量检查，失败时内部状态完全不变。
        if n_blocks - len(hits) > self.num_free_blocks - hits_from_evictable:
            raise OutOfBlocks  # check first -> allocation is all-or-nothing

        table: list[int] = []
        for b in hits:
            # 复用 evictable 中的 block：把它从 LRU 池中拿出来（变回"使用中"）。
            if self._ref[b] == 0:
                self._evictable.pop(b, None)
            self._ref[b] += 1
            table.append(b)
        # 未命中的部分：取新 block；若该位置是满 block，则登记 hash 供后续请求复用。
        for i in range(len(hits), n_blocks):
            b = self._take_block()
            table.append(b)
            if i < len(hashes):
                self._register(b, hashes[i])

        self._tables[seq_id] = table
        self._tokens[seq_id] = list(token_ids)
        self._seq_hashes[seq_id] = hashes
        self._cached_tokens[seq_id] = len(hits) * bs
        return list(table)

    # append_slot：decode 时为新 token 追加一个 slot，返回该 token 在 KV cache 中的 slot。
    # 如当前最后一个 block 已满则先新取一个 block；token 写满一个 block 时为它计算链式 hash 并登记。
    def append_slot(self, seq_id: str, token_id: int) -> int:
        bs = self.block_size
        tokens, table = self._tokens[seq_id], self._tables[seq_id]
        # 新 token 的位置 = 当前 token 数。
        pos = len(tokens)
        # pos 正好落在新 block 的起点（block table 已用满）时需要一个新 block。
        if pos // bs == len(table):
            # 原子性：_take_block 放在 tokens.append 之前，若抛 OutOfBlocks 则 seq 状态不被破坏。
            table.append(self._take_block())  # may raise OutOfBlocks before any mutation
        tokens.append(token_id)
        # 追加后该 block 恰好写满：计算 hash（parent 为上一个满 block 的 hash），并登记到 prefix cache。
        if (pos + 1) % bs == 0:
            hashes = self._seq_hashes[seq_id]
            parent = hashes[-1] if hashes else None
            h = _block_hash(parent, tuple(tokens[pos + 1 - bs:pos + 1]))
            hashes.append(h)
            self._register(table[pos // bs], h)
        return self.slot(seq_id, pos)

    # free：释放 seq 的所有 block。ref 归零后，有 hash 的进入 evictable（保留缓存），无 hash 的回到 free list。
    def free(self, seq_id: str) -> None:
        table = self._tables.pop(seq_id)
        self._tokens.pop(seq_id)
        self._seq_hashes.pop(seq_id)
        self._cached_tokens.pop(seq_id, None)
        # 逆序释放：尾部 block 先进入 LRU，因而先被驱逐；前缀开头的 block 后进入、保留更久
        #  （开头的 block 被其他请求共享的概率更高）。
        for b in reversed(table):  # tail blocks become the first eviction candidates
            self._ref[b] -= 1
            # 只有 ref 归零才真正释放；共享 block 仍被其他 seq 引用时保持不动。
            if self._ref[b] == 0:
                if self._hash[b] is not None:
                    self._evictable[b] = None
                else:
                    self._free.append(b)
```


© 2026 kzhou003

C-18 · solutions/p08_sampler.py

```
from dataclasses import dataclass

import torch


# 每个请求独立的采样参数。seed=None 表示不需要可复现，共用全局默认 generator。
@dataclass
class SamplingParams:
    temperature: float = 1.0
    top_k: int = -1
    top_p: float = 1.0
    repetition_penalty: float = 1.0
    seed: int | None = None


# repetition penalty：降低已出现过的 token 的 logit。
# 正 logit 除以 penalty、负 logit 乘以 penalty，两种情况都会让该 token 概率下降（penalty > 1 时）。
def apply_repetition_penalty(logits: torch.Tensor, prev_tokens: list[int], penalty: float) -> torch.Tensor:
    """HF/vLLM convention: positive logits are divided, negative ones multiplied."""
    if penalty == 1.0 or not prev_tokens:
        return logits
    # 易错：不能原地修改调用方传入的 logits。
    out = logits.clone()
    # 去重：同一个 token 在历史中出现多次也只惩罚一次（penalty 不随出现次数叠加）。
    idx = torch.tensor(sorted(set(prev_tokens)), dtype=torch.long)
    vals = out[idx]
    out[idx] = torch.where(vals > 0, vals / penalty, vals * penalty)
    return out


# 对一行 logits（shape [V]）依次做 top_k 再 top_p 过滤，被过滤的位置置为 -inf（softmax 后概率为 0）。
def _filter_top_k_top_p(logits: torch.Tensor, top_k: int, top_p: float) -> torch.Tensor:
    out = logits.clone()
    # top_k <= 0 或 >= V 时视为不过滤。
    if 0 < top_k < out.numel():
        # 第 k 大的值作为阈值；严格小于它的置 -inf（与第 k 大值相等的并列项会被保留）。
        kth = torch.topk(out, top_k).values[-1]
        out[out < kth] = float("-inf")
    if top_p < 1.0:
        # 在 top_k 过滤之后的分布上做 nucleus 采样：按概率降序排序后计算累积概率。
        sorted_logits, order = torch.sort(out, descending=True)
        probs = torch.softmax(sorted_logits, dim=-1)
        cum = torch.cumsum(probs, dim=-1)
        # cum - probs 是"该 token 之前"的累积质量；只有之前的质量已超过 top_p 才移除，
        # 这样跨过阈值的那个 token 被保留，保留的是"质量达到 top_p 的最短前缀"。
        remove = (cum - probs) > top_p  # keep the token that crosses the threshold
        # 易错：至少保留概率最高的 1 个 token，避免全部被置 -inf 导致 softmax 出 NaN。
        remove[0] = False
        # order 把排序后的位置映射回原始 vocab 下标。
        out[order[remove]] = float("-inf")
    return out


# Sampler：逐行（row-wise）采样，保证 batch invariance——
# 某个请求采样到的 token 只取决于它自己的 logits、参数和 generator 状态，与同 batch 的其他行及其顺序无关。
class Sampler:
    """Row-wise sampler whose result for a request never depends on the rest of the batch."""

    def __init__(self) -> None:
        # req_id -> 该请求私有的 torch.Generator（仅带 seed 的请求才有）。
        self._generators: dict[str, torch.Generator] = {}
        # 未指定 seed 的请求共用的默认 generator（不保证可复现）。
        self._default = torch.Generator().manual_seed(0)

    # 获取请求的 generator：带 seed 的请求在第一次使用时用 seed 创建，之后在多步 decode 间持续推进同一个随机流。
    def _gen(self, req_id: str, p: SamplingParams) -> torch.Generator:
        if p.seed is None:
            return self._default
        g = self._generators.get(req_id)
        # 易错：不能每步都用 seed 重新创建 generator，否则每步抽到的随机数都相同。
        if g is None:
            g = torch.Generator().manual_seed(p.seed)
            self._generators[req_id] = g
        return g

    # 请求结束时丢弃 generator；之后同 id 的新请求会从 seed 重新开始随机流。
    def free(self, req_id: str) -> None:
        self._generators.pop(req_id, None)

    # logits: [B, V]；返回 [B] long。
    # 每行流程：repetition penalty -> (greedy | temperature -> top_k/top_p -> multinomial)。
    def sample(
        self,
        logits: torch.Tensor,
```


© 2026 kzhou003

```
        req_ids: list[str],
        params: list[SamplingParams],
        prev_tokens: list[list[int]],
    ) -> torch.Tensor:
        out = torch.empty(logits.shape[0], dtype=torch.long)
        for i, (rid, p) in enumerate(zip(req_ids, params, strict=True)):
            # 转成 float32 计算，避免低精度 logits 下 softmax/cumsum 的数值误差。
            row = apply_repetition_penalty(logits[i].float(), prev_tokens[i], p.repetition_penalty)
            # temperature≈0 视为 greedy：直接 argmax，不消耗 generator 的随机数。
            if p.temperature < 1e-5:
                out[i] = int(torch.argmax(row))
                continue
            # 先除 temperature 再做 top_k/top_p 过滤（顺序与 spec 保持一致）。
            row = _filter_top_k_top_p(row / p.temperature, p.top_k, p.top_p)
            probs = torch.softmax(row, dim=-1)
            # 每行只抽 1 个样本，且使用该请求自己的 generator，这是 batch invariance 的关键。
            out[i] = int(torch.multinomial(probs, 1, generator=self._gen(rid, p)))
        return out
```

C-19 · solutions/p09_paged_attention.py

```
import torch


# write_kv：把本步新计算出的 K/V 按 slot_mapping 散写（scatter）进 paged KV cache。
# slot = 物理 block id * block_size + block 内偏移（与 BlockManager.slot 的定义一致）。
def write_kv(
    k_cache: torch.Tensor,  # [num_blocks, block_size, num_kv_heads, head_dim]
    v_cache: torch.Tensor,
    k: torch.Tensor,  # [num_tokens, num_kv_heads, head_dim]
    v: torch.Tensor,
    slot_mapping: torch.Tensor,  # [num_tokens], -1 = padding token, do not write
) -> None:
    bs = k_cache.shape[1]
    # 易错：padding token 的 slot 为 -1，必须过滤掉；否则 -1 会被当作负下标写进最后一个 block。
    valid = slot_mapping >= 0
    slots = slot_mapping[valid]
    # 扁平 slot 拆成 (block, offset) 两维下标。
    blocks, offsets = slots // bs, slots % bs
    # 高级索引赋值：k_cache[blocks[j], offsets[j]] = k[valid][j]，每个元素是 [num_kv_heads, head_dim]。
    k_cache[blocks, offsets] = k[valid]
    v_cache[blocks, offsets] = v[valid]


# paged_decode_attention：decode 阶段的 attention，每条序列只有 1 个 query token。
# 对每条序列：用 block table 把分散在各物理 block 中的 K/V 收集（gather）成连续的 [n, KVH, D]，
# 截断到 context_len，GQA 下把 KV head 复制到与 query head 数相同，再做标准 softmax(QK^T * scale) V。
def paged_decode_attention(
    q: torch.Tensor,  # [B, num_heads, head_dim] (one query token per sequence)
    k_cache: torch.Tensor,
    v_cache: torch.Tensor,
    block_tables: torch.Tensor,  # [B, max_blocks], unused tail entries may hold any value
    context_lens: torch.Tensor,  # [B]
    scale: float,
) -> torch.Tensor:
    bsz, num_heads, _ = q.shape
    block_size, num_kv_heads = k_cache.shape[1], k_cache.shape[2]
    # GQA：每 group 个 query head 共享一个 KV head。
    group = num_heads // num_kv_heads
    out = torch.empty_like(q)
    for b in range(bsz):
        # n：该序列的上下文长度（含当前 token，其 KV 已由 write_kv 写入）。
        n = int(context_lens[b])
        # 实际用到的 block 数 = ceil(n / block_size)。
        nb = -(-n // block_size)
        # 易错：只取前 nb 个表项；尾部未使用的表项可能是任意值（垃圾/越界），不能参与 gather。
        table = block_tables[b, :nb].long()
        # k_cache[table]: [nb, block_size, KVH, D] -> 展平成 [nb*block_size, KVH, D]，
        # 再截断到 n，丢弃最后一个 block 中未写入的槽位。
        k = k_cache[table].reshape(-1, num_kv_heads, k_cache.shape[-1])[:n]  # [n, KVH, D]
        v = v_cache[table].reshape(-1, num_kv_heads, v_cache.shape[-1])[:n]
        # repeat_interleave（而非 repeat）：query head h 对应 KV head h // group，顺序必须一致。
        k = k.repeat_interleave(group, dim=1)  # GQA: [n, H, D]
        v = v.repeat_interleave(group, dim=1)
        # scores[h, j] = q[h] · k[j, h] * scale -> [H, n]；用 float32 计算以保证数值稳定。
        scores = torch.einsum("hd,nhd->hn", q[b].float(), k.float()) * scale
        # 在 key 维（n）上做 softmax；decode 时 query 是最后一个 token，无需 causal mask。
        probs = torch.softmax(scores, dim=-1)
        # out[h] = sum_j probs[h, j] * v[j, h] -> [H, D]，再转回输入 dtype。
        out[b] = torch.einsum("hn,nhd->hd", probs, v.float()).to(q.dtype)
    return out
```


© 2026 kzhou003

C-20 · solutions/p12_shm_connector.py

```
import fcntl
import hashlib
import os
import tempfile
from multiprocessing import resource_tracker
from multiprocessing.shared_memory import SharedMemory

# 段布局：[ 8 字节 little-endian 长度 n | n 字节 payload | (可能的页对齐填充) ]
# 易错：不能用 shm.size 当 payload 长度——macOS / Linux 会把段大小向上取整到页大小，必须靠 header 记录真实长度。
_HEADER = 8  # little-endian payload length; shm segments may be rounded up to a page


# _untrack：把刚创建的段从本进程的 resource_tracker 里注销，实现"所有权转移给读者"。
# 流程：writer create → 写入 → close（不 unlink）→ reader 打开 → 读 → unlink。
# 若不注销，writer 进程退出时 tracker 会把"泄漏"的段 unlink 掉，reader 可能还没来得及读。
def _untrack(shm: SharedMemory) -> None:
    # Python < 3.13 registers every created segment with the creator's resource tracker.
    # When that tracker shuts down (e.g. an independent writer process exits) it unlinks
    # every still-registered segment as "leaked" -- possibly before the consumer read it.
    # Ownership moves to the reader, who unlinks after reading.
    try:
        # shm._name 带前导 "/"（POSIX 名字），tracker 注册时用的就是它，所以这里也必须用 _name 而不是 name
        resource_tracker.unregister(shm._name, "shared_memory")  # type: ignore[attr-defined]
    except Exception:
        # 尽力而为：有的平台/版本没有注册过，unregister 失败也无所谓
        pass


# ShmConnector：stage 间数据面的"共享内存邮箱"。一个 key 对应一个 shm 段 + 一个锁文件。
# 协议：
#   - 锁文件存在   = 有人写过、还没被读（或正在写）
#   - 锁文件被 flock 持有 = writer 正在写，reader 应立即返回 None（非阻塞）
#   - reader 读完后负责 unlink 段 + 删除锁文件（消费即删除，每个 key 只有一个消费者）
#   - producer 记录自己写过的 key（_pending），请求中止时用 cleanup_prefix 清理没人读的段
class ShmConnector:
    """put/get of byte payloads through named POSIX shared memory, one consumer per key."""

    def __init__(self, lock_dir: str | None = None, namespace: str = "nvo") -> None:
        # Linux 上锁文件放 /dev/shm（tmpfs，和段同在内存里）；macOS 没有 /dev/shm 就退回系统临时目录
        default = "/dev/shm" if os.path.isdir("/dev/shm") else tempfile.gettempdir()
        self.lock_dir = lock_dir or default
        self.namespace = namespace
        # 用 dict 当"有序集合"：只关心 key，不关心 value
        self._pending: dict[str, None] = {}  # keys this producer wrote and may need to clean up

    # macOS limits POSIX shm names to 31 bytes -> hash the logical key
    # 名字必须是 key 的确定性函数：另一个进程里的 reader 只知道 key，要能独立算出同一个段名。
    # 例：namespace="nvo" → "nvo_" + 20 位 hex = 24 字节（加上前导 "/" 也 < 31）
    def shm_name(self, key: str) -> str:
        return f"{self.namespace}_" + hashlib.sha1(key.encode()).hexdigest()[:20]

    def lock_path(self, key: str) -> str:
        return os.path.join(self.lock_dir, self.shm_name(key) + ".lock")

    # put：写者路径。先拿排他锁（告诉读者"正在写"），再创建段、写 header + 数据、close。
    # 写完后锁文件保留（表示"有数据待读"），段也保留（所有权已交给读者）。
    def put(self, key: str, data: bytes) -> None:
        # O_CREAT：锁文件由 writer 创建；reader 只会 open 已存在的锁文件
        fd = os.open(self.lock_path(key), os.O_CREAT | os.O_RDWR, 0o600)
        try:
            # 阻塞式排他锁：若 reader 正持锁读取，writer 等它读完再写
            fcntl.flock(fd, fcntl.LOCK_EX)  # "being written" marker for readers
            name, size = self.shm_name(key), _HEADER + len(data)
            try:
                shm = SharedMemory(name=name, create=True, size=size)
            except FileExistsError:  # stale segment from an earlier incarnation
                # 同名段已存在（上一次写入没人读 / 进程崩溃残留）：先删掉旧段再重建。
                # 易错：不能直接复用旧段——它的大小可能比新 data 小。
                old = SharedMemory(name=name)
                old.close()
                old.unlink()
                shm = SharedMemory(name=name, create=True, size=size)
            # 易错：漏掉这一步，writer 进程退出时 tracker 会把还没被读的段 unlink 掉
            _untrack(shm)
            shm.buf[:_HEADER] = len(data).to_bytes(_HEADER, "little")
            shm.buf[_HEADER:size] = data
            # 只 close（解除本进程映射），不 unlink——段要留给 reader
            shm.close()
            self._pending[key] = None
        finally:
            # 释放锁后 reader 的 LOCK_NB 才能成功 → 读者永远看不到写了一半的数据
            fcntl.flock(fd, fcntl.LOCK_UN)
            os.close(fd)
```


© 2026 kzhou003

```
    # get: 读者路径，非阻塞。三种 None: 锁文件不存在 / 锁被 writer 持有 / 段不存在。
    # 成功时读出数据并"消费": unlink 段 + 删除锁文件，所以同一个 key 第二次 get 返回 None。
    def get(self, key: str) -> bytes | None:
        """Non-blocking. Returns None if absent or still being written. Consumes the key."""
        try:
            # 不带 O_CREAT: 锁文件不存在 = 从没写过（或已被消费），不能替 writer 创建
            fd = os.open(self.lock_path(key), os.O_RDWR)
        except FileNotFoundError:
            return None
        try:
            try:
                # LOCK_NB: 拿不到锁立刻抛 BlockingIOError，而不是阻塞等待 writer。
                # 注: flock 锁属于 open file description，同进程里另一个 fd 持锁也会冲突。
                fcntl.flock(fd, fcntl.LOCK_EX | fcntl.LOCK_NB)
            except BlockingIOError:
                return None  # writer still holds the lock
            try:
                # attach 已存在的段（create=False）。Python < 3.13 attach 也会注册到本进程 tracker，
                # 但下面的 shm.unlink() 内部会 unregister，所以 reader 这边不需要 _untrack
                shm = SharedMemory(name=self.shm_name(key))
            except FileNotFoundError:
                # 锁文件在但段不在: 例如 writer 在创建段前崩溃，或刚被别人清理
                return None
            n = int.from_bytes(shm.buf[:_HEADER], "little")
            # bytes(...) 复制一份: close 之后 memoryview 就失效了，必须先拷贝出来
            data = bytes(shm.buf[_HEADER:_HEADER + n])
            shm.close()
            shm.unlink()  # the reader frees the memory
            # 删除锁文件 = 宣告"已消费"；此时我们仍持有锁，所以 writer 不会在中途插进来
            try:
                os.unlink(self.lock_path(key))
            except FileNotFoundError:
                pass
            return data
        finally:
            fcntl.flock(fd, fcntl.LOCK_UN)
            os.close(fd)

    # _drop: 强制删除一个 key 的段和锁文件（不读取数据），各步都容忍"已经不存在"。
    def _drop(self, key: str) -> None:
        try:
            shm = SharedMemory(name=self.shm_name(key))
            shm.close()
            shm.unlink()
        except FileNotFoundError:
            pass
        try:
            os.unlink(self.lock_path(key))
        except FileNotFoundError:
            pass

    # cleanup_prefix: producer 侧在请求结束/中止时调用（key 形如 f"{req_id}_{stage}_{chunk}",
    # 前缀通常是 f"{req_id}_"）。只清理"我写过 + 前缀匹配 + 还没被消费"的 key，返回清理个数。
    # 易错: 前缀要带分隔符，"rA_" 不会误伤 "rAx_..."；而 "rA" 会。
    def cleanup_prefix(self, prefix: str) -> int:
        """Producer-side: unlink every key we wrote that starts with prefix and was never consumed."""
        n = 0
        # 先拷贝成 list: 循环里要 del self._pending[key]，不能边遍历边改 dict
        for key in [k for k in self._pending if k.startswith(prefix)]:
            del self._pending[key]
            # 锁文件还在 = 没被 reader 消费（reader 消费时会删锁文件）→ 才需要清理并计数
            if os.path.exists(self.lock_path(key)):
                self._drop(key)
                n += 1
        return n
```

C-21 · solutions/p13_chunk_adapter.py

```
from threading import Condition


import copy
import threading
from collections import deque
from collections.abc import Callable
from typing import Any


# ---------------------------------------------------------------- given types
# InMemoryConnector: 题目给定的传输层（代替 p12 的 shm connector），只负责按 key 存取，get 即消费。
class InMemoryConnector:
    """Transport only: put/get by key, get consumes. Thread-safe. Hooks let tests add latency."""

    def __init__(self, on_put: Callable[[str], None] | None = None, on_get: Callable[[str], None] | None = None):
```


© 2026 kzhou003

```
        self._store: dict[str, Any] = {}
        self._lock = threading.Lock()
        self.on_put, self.on_get = on_put, on_get
        self.puts: list[str] = []

    def put(self, key: str, obj: Any) -> None:
        # hook 在锁外调用：测试可以在这里 sleep 模拟慢传输，而不会把整个 store 锁住
        if self.on_put:
            self.on_put(key)
        with self._lock:
            self._store[key] = obj
            self.puts.append(key)

    def get(self, key: str) -> Any | None:
        with self._lock:
            present = key in self._store
        if not present:
            return None
        if self.on_get:
            self.on_get(key)
        # pop 带默认值：两次加锁之间理论上可能被别人取走，此时返回 None 而不是抛 KeyError
        with self._lock:
            return self._store.pop(key, None)


# ---------------------------------------------------------- implementation
# ChunkAdapter：async-chunk 数据面里一个 stage 的"发送端 + 接收端"。
#   发送端：调度线程调用 save_async 投递快照 → 后台 save 线程 build_payload 后按
#           key = f"{req_id}_{self.stage_id}_{chunk_id}" put 到 connector，chunk_id 从 0 连续递增。
#   接收端：load_async 注册 req → 后台 recv 线程按 f"{req_id}_{src_stage_id}_{chunk_id}" 轮询 get，
#           按序提交到 _ready，调度线程用 poll_ready 取走。
# 两个后台线程让慢速传输/序列化不阻塞调度线程；所有共享状态都在各自的锁里读写。
class ChunkAdapter:
    """Sender + receiver halves of the async-chunk data plane for one stage."""

    def __init__(
        self,
        connector: InMemoryConnector,
        stage_id: int,
        src_stage_id: int | None,
        build_payload: Callable[[Any], Any | None],
    ) -> None:
        self.connector = connector
        # stage_id：本 stage 作为发送方时写进 key 的编号；src_stage_id：作为接收方时上游的编号
        self.stage_id = stage_id
        self.src_stage_id = src_stage_id
        self.build_payload = build_payload
        self._stop = threading.Event()
        # sender
        # _save_q 元素：(req_id, 深拷贝的 snapshot, is_final)。deque 的 append/popleft 本身线程安全
        self._save_q: deque = deque()
        # _save_cond 只用来"唤醒 save 线程"，不保护数据
        self._save_cond: Condition = threading.Condition()
        # _send_lock 保护 _send_chunk 与 _cancelled（调度线程 cancel_sender 与 save 线程并发访问）
        self._send_lock = threading.Lock()
        self._send_chunk: dict[str, int] = {}  # req_id → 下一个要发送的 chunk id
        self._cancelled: set[str] = set()
        # receiver
        self._load_q: deque[str] = deque()  # 待轮询的 req_id（轮转队列）
        self._recv_cond = threading.Condition()
        # _recv_lock 保护下面三个 dict（recv 线程 vs 调度线程的 poll_ready / cleanup_receiver）
        self._recv_lock = threading.Lock()
        # req_id → 代际令牌（generation token）：每次注册新建一个 object()，用 `is` 比较判断
        # "读取期间是否被 cleanup 或重新注册过"
        self._registered: dict[str, object] = {}
        self._recv_chunk: dict[str, int] = {}  # req_id → 下一个期望收到的 chunk id
        self._ready: dict[str, list] = {}  # req_id → 已按序提交、等待 poll_ready 取走的 chunk
        self._threads: list[threading.Thread] = []

    def start(self) -> None:
        for target, name in ((self._save_loop, "save"), (self._recv_loop, "recv")):
            # daemon=True：即使 stop 没被调用，进程退出也不会被这两个线程卡住
            t = threading.Thread(target=target, name=f"chunk-{name}-{self.stage_id}", daemon=True)
            t.start()
            self._threads.append(t)

    def stop(self) -> None:
        # 先置位 stop，再 notify_all：线程醒来后检查 _stop 即退出；顺序反了可能在 wait 里多睡一个 timeout
        self._stop.set()
        for cond in (self._save_cond, self._recv_cond):
            with cond:
                cond.notify_all()
        for t in self._threads:
            t.join(2.0)

    # ---------------------------------------------------------- sender
    # save_async：调度线程调用，必须立即返回。
    # 易错：一定要 deepcopy——调用方会继续原地修改 snapshot（例如往 token 列表里 append），
```


© 2026 kzhou003

```
    # 若只存引用，后台线程 build_payload 时看到的是"未来"的状态。
    def save_async(self, req_id: str, snapshot: Any, is_final: bool) -> None:
        self._save_q.append((req_id, copy.deepcopy(snapshot), is_final))  # caller keeps mutating
        with self._save_cond:
            self._save_cond.notify()

    # cancel_sender：请求被 abort 时调用。之后该 req 的非 final 数据全部丢弃，但 final 标记仍然要发，
    # 否则下游接收端永远等不到 finished=True，会一直轮询这个 req。
    def cancel_sender(self, req_id: str) -> None:
        with self._send_lock:
            self._cancelled.add(req_id)

    def _save_loop(self) -> None:
        while not self._stop.is_set():
            if not self._save_q:
                with self._save_cond:
                    # 持锁后再检查一次：避免"检查为空 → 生产者 append+notify → 我们才 wait"的丢通知
                    if not self._save_q and not self._stop.is_set():
                        self._save_cond.wait(timeout=0.1)  # timeout = lost-notify fallback
                continue
            # 单个 save 线程按 FIFO 处理 → 同一 req 的 chunk 顺序与 save_async 调用顺序一致
            req_id, snap, is_final = self._save_q.popleft()
            with self._send_lock:
                cancelled = req_id in self._cancelled
            if cancelled and not is_final:
                continue  # drop data, but never the end-of-stream marker
            # build_payload 可能很慢（序列化/拷贝张量），放在锁外，避免阻塞 cancel_sender
            data = None if cancelled else self.build_payload(snap)  # outside any lock
            # payload 为 None（本步没有新东西可发）且不是 final → 跳过，不消耗 chunk id。
            # 例：6 次 save，其中第 3 次 payload=None → 发出 chunk 0..4 共 5 个连续 id
            if data is None and not is_final:
                continue  # nothing to send this step; chunk ids stay contiguous
            with self._send_lock:
                cid = self._send_chunk.get(req_id, 0)
            try:
                # key 用自己的 stage_id；下游用 src_stage_id 拼出同一个 key
                self.connector.put(f"{req_id}_{self.stage_id}_{cid}", {"data": data, "finished": is_final})
            except Exception:
                # put 失败：cid 不前进，这一块丢失但后续 chunk id 仍连续（接收端不会卡在空洞上）
                continue
            with self._send_lock:  # advance only after a successful put
                self._send_chunk[req_id] = cid + 1
                if is_final:
                    # 流结束：清理该 req 的发送状态，防止长期运行时 dict 无限增长
                    self._send_chunk.pop(req_id, None)
                    self._cancelled.discard(req_id)

    # ---------------------------------------------------------- receiver
    # load_async：注册一个要接收的 req。幂等：已注册则直接返回（避免同一 req 在 _load_q 里出现两次，
    # 两份轮询会抢同一个 chunk id）。
    def load_async(self, req_id: str) -> None:
        with self._recv_lock:
            if req_id in self._registered:
                return
            self._registered[req_id] = object()  # generation token
            self._recv_chunk.setdefault(req_id, 0)
        self._load_q.append(req_id)
        with self._recv_cond:
            self._recv_cond.notify()

    # poll_ready：原子地"整体换出" _ready —— 拿走自上次 poll 以来所有已提交的 chunk（每个 req 内按 chunk id 有序）
    def poll_ready(self) -> dict[str, list]:
        """Called on the scheduler thread: committed chunks, in order, since the last poll."""
        with self._recv_lock:
            out, self._ready = self._ready, {}
        return out

    # cleanup_receiver：请求被 abort / 完成后由调度线程调用。删掉注册令牌后，recv 线程下次轮到它时会
    # 发现 entry 为 None 而不再放回队列；正在 get 中的那一次则通过令牌比对被丢弃。
    def cleanup_receiver(self, req_id: str) -> None:
        with self._recv_lock:
            self._registered.pop(req_id, None)
            self._recv_chunk.pop(req_id, None)
            self._ready.pop(req_id, None)

    def _recv_loop(self) -> None:
        while not self._stop.is_set():
            # 每轮只处理"本轮开始时"队列里的 n 个 req；没收到数据的会被 append 回队尾，下一轮再试
            n = len(self._load_q)
            progressed = False
            for _ in range(n):
                req_id = self._load_q.popleft()
                # 在锁内快照（令牌，期望 chunk id），之后在锁外做可能很慢的 get
                with self._recv_lock:
                    entry = self._registered.get(req_id)
                    cid = self._recv_chunk.get(req_id)
                if entry is None or cid is None:
                    continue  # cleaned up -> stop polling it
```


© 2026 kzhou003

```
                # 严格按 cid 取：只要 chunk cid 没到就不会去拿 cid+1 → 天然保证按序提交
                obj = self.connector.get(f"{req_id}_{self.src_stage_id}_{cid}")  # outside the lock
                if obj is None:
                    self._load_q.append(req_id)
                    continue
                progressed = True
                with self._recv_lock:
                    # 易错：get 期间调度线程可能 cleanup 了它（甚至又 load_async 重新注册）。
                    # 用 `is` 比较令牌：新注册的是另一个 object()，旧数据必须丢弃，不能写进新一代的 _ready。
                    if self._registered.get(req_id) is not entry:
                        continue  # cleaned up (or re-registered) while we were reading: discard
                    self._ready.setdefault(req_id, []).append(obj)
                    self._recv_chunk[req_id] = cid + 1
                    if obj["finished"]:
                        # 收到流结束标记：注销并且不再放回 _load_q（_ready 中的数据保留给 poll_ready）
                        self._registered.pop(req_id, None)
                        self._recv_chunk.pop(req_id, None)
                        continue
                # 收到了一块但还没结束 → 放回队尾，下一轮继续取 cid+1（轮转保证多个 req 之间公平）
                self._load_q.append(req_id)
            with self._recv_cond:
                if self._stop.is_set():
                    break
                if not self._load_q:
                    # 没有任何待接收的 req：睡到 load_async 的 notify（或 0.1s 超时兜底）
                    self._recv_cond.wait(timeout=0.1)
                elif not progressed:
                    # 有 req 但本轮一块都没收到：短暂退避 1ms，避免 100% CPU 空转
                    self._recv_cond.wait(timeout=0.001)  # back off instead of spinning
```

C-22 · solutions/p15_talker_inputs.py

```
from dataclasses import dataclass

import torch


# ThinkerPayload：thinker stage 交给 talker stage 的数据（已裁掉 stop token 那一行）。
# prefill 行对应 prompt token，decode 行与 output_ids 一一对应（decode.shape[0] == len(output_ids)）。
@dataclass
class ThinkerPayload:
    prefill: torch.Tensor  # [P, D]   rows for the prompt
    decode: torch.Tensor   # [K, D]   rows for generated tokens (stop row removed)
    output_ids: list[int]  # generated ids (stop id removed)


# trim_thinker_payload：把 thinker 在 prefill + decode 过程中累积的 hidden 行切成 prompt / 回复两部分，
# 并在 finish_reason == "stop" 时去掉最后一个生成 token（stop/EOS）及其对应行。
# 例：P=3, output_ids=[a, b, EOS], finish="stop" → drop=1, target_rows=3+3-1=5;
#     prefill=hidden[0:3], decode=hidden[3:5]（a、b 两行），output_ids=[a, b]
def trim_thinker_payload(
    prompt_ids: list[int],
    output_ids: list[int],
    hidden: torch.Tensor,  # [R, D] accumulated rows over prefill + decode steps
    finish_reason: str,    # "stop" | "length"
) -> ThinkerPayload:
    """vLLM appends the sampled stop token *before* checking stop conditions, so a
    stop-finished request carries one row the talker must not consume."""
    P = len(prompt_ids)
    # prefill 行都没抓全说明 hidden 捕获逻辑有 bug（例如 chunked prefill 只记了最后一块），直接报错
    if hidden.shape[0] < P:
        raise ValueError(f"under-captured prefill: {hidden.shape[0]} rows < prompt {P}")
    # "length" 结束时最后一个 token 是正常内容，不能丢；output_ids 为空时也没有可丢的
    drop = 1 if (finish_reason == "stop" and output_ids) else 0
    outs = output_ids[:len(output_ids) - drop]
    target_rows = P + len(output_ids) - drop
    # 易错：按 target_rows 截断，而不是盲目"去掉最后一行"——"length" 结束时不该丢；
    # R 若多于 target_rows（多捕获了行）也一并截掉。例：P=3, [a, b, EOS], R=6 → 保留 5 行
    rows = hidden[:target_rows]
    return ThinkerPayload(prefill=rows[:P], decode=rows[P:], output_ids=list(outs))


# build_talker_prefill：构造 Qwen2.5-Omni talker 的 prefill 输入（ids 与 inputs_embeds 并行的两路）。
# 序列布局（共 P+2 行）：
#   [0, P)  prompt 行：thinker 的 prompt embedding，id 用 mask_id 占位（talker 不按 id 查表）
#   P       spk_embed + codec_pad_embed，id = pad_id
#   P+1     first_reply + codec_bos_embed，id = start_id（从这里开始采样第一个 codec token）
def build_talker_prefill(
    prompt_embeds: torch.Tensor,  # [P, D]
    first_reply: torch.Tensor,    # [D]  thinker row of the first generated token
    spk_embed: torch.Tensor,      # [D]  speaker text-token embedding
    codec_pad_embed: torch.Tensor,
    codec_bos_embed: torch.Tensor,
    *,
```


© 2026 kzhou003

```
    mask_id: int,
    pad_id: int,
    start_id: int,
) -> tuple[list[int], torch.Tensor]:
    """Qwen2.5-Omni talker prompt: [prompt rows] + [spk+pad] + [first_reply+bos]."""
    # [None] 把 [D] 变成 [1, D]，才能和 [P, D] 沿 dim=0 拼接 → 结果 [P+2, D]
    # 每个位置 = "文本侧" embedding + "codec 侧" embedding 逐元素相加（双流叠加）
    embeds = torch.cat(
        [prompt_embeds, (spk_embed + codec_pad_embed)[None], (first_reply + codec_bos_embed)[None]],
        dim=0,
    )
    # len(ids) == embeds.shape[0] == P + 2
    ids = [mask_id] * prompt_embeds.shape[0] + [pad_id, start_id]
    return ids, embeds


# talker_decode_input：talker 每个 decode step 的输入 = 下一行 thinker 回复 hidden + 上一步采样到的 codec embedding。
# thinker 行是一个 FIFO 队列，每步消费一行；用完后（talker 通常比文本长）改用 fallback（如 tts_pad）。
# 返回（本步输入 [D]，剩余队列 [K-1, D] 或原样的空队列）。
def talker_decode_input(
    reply_queue: torch.Tensor,     # [K, D] remaining thinker rows
    last_codec_embed: torch.Tensor,  # [D] embedding of the codec id sampled last step
    fallback: torch.Tensor,        # [D] used once thinker rows are exhausted (e.g. tts_pad)
) -> tuple[torch.Tensor, torch.Tensor]:
    if reply_queue.shape[0] > 0:
        # reply_queue[0] 是 [D]（不是 [1, D]）；rest 是 [K-1, D] 的 view
        step, rest = reply_queue[0], reply_queue[1:]
    else:
        # 队列已空：fallback 可以无限次使用，队列保持 [0, D]
        step, rest = fallback, reply_queue
    return step + last_codec_embed, rest
```

C-23 · solutions/p14_codec_windows.py

```
from collections.abc import Sequence


# next_window：流式 TTS 中决定"下一块 codec 帧什么时候、发哪一段给 vocoder"。
# 窗口 = [ctx_start, new_start) 左上下文（只用来让 vocoder 平滑衔接，输出会被丢弃）
#       + [new_start, new_end)  本次真正新增的帧。
# 首块用较小的 initial（降低首包延迟），之后每块 chunk 帧；流结束时把不足一块的尾巴 flush 掉。
# 例：initial=2, chunk=5, left_ctx=3
#   (available=2,  emitted=0)          → (0, 0, 2)    首块 2 帧
#   (available=12, emitted=7)          → (4, 7, 12)   左上下文 4,5,6
#   (available=20, emitted=17, finished) → (14, 17, 20) 尾巴 3 帧 flush
def next_window(
    available: int,
    emitted: int,
    finished: bool,
    *,
    initial: int,
    chunk: int,
    left_ctx: int,
) -> tuple[int, int, int] | None:
    """Decide the next codec window to ship downstream.

    available: frames produced so far by the talker
    emitted:   frames already shipped (end of the previous window)
    returns (ctx_start, new_start, new_end) or None if we should wait for more frames.
    """
    # 只有"第一块"（emitted == 0）且配置了 initial 才用 initial；initial=0 表示不区分首块
    size = initial if (emitted == 0 and initial > 0) else chunk
    if available - emitted >= size:
        # 够一整块：即使 finished 也按整块切，不会把多块合并成一个大窗口
        new_end = emitted + size
    elif finished and available > emitted:
        # 已结束且还有未发送的帧（不足一块）→ 全部发出。例：极短的流 available=1 → (0, 0, 1)
        new_end = available  # flush the tail
    else:
        # 帧不够且没结束（等更多帧），或已结束且 available == emitted（全部发完）
        return None
    new_start = emitted
    # 左上下文被 0 截断：例 new_start=2, left_ctx=3 → ctx_start=0（只有 2 帧上下文）
    ctx_start = max(0, new_start - left_ctx)
    return ctx_start, new_start, new_end


# slice_new_samples：vocoder 对 codes[ctx_start:new_end] 解码后，每帧对应 upsample 个波形样本；
# 丢掉前面属于上下文帧的样本，只保留新帧的样本，拼接起来就是无重复、无缝隙的连续音频。
# 例：window=(4, 7, 12), upsample=2 → decoded 长度 (12-4)*2=16；
#     上下文帧 4..6 占样本 [0, 6)，新帧 7..11 占样本 [6, 16) → 返回 decoded[6:16]
def slice_new_samples(decoded: Sequence[float], window: tuple[int, int, int], upsample: int) -> list[float]:
    """decoded = vocoder(codes[ctx_start:new_end]); keep only the samples of the new frames."""
    ctx_start, new_start, new_end = window
```


© 2026 kzhou003

```
    # 易错：decoded 的下标是相对 ctx_start 的（不是绝对帧号），所以下面都要减 ctx_start
    assert len(decoded) == (new_end - ctx_start) * upsample
    return list(decoded[(new_start - ctx_start) * upsample:(new_end - ctx_start) * upsample])
```

C-24 · solutions/p18_mrope_positions.py

```
import torch

# Segment：("text", token 数) 或 ("image", (t, h, w))，图像的 t/h/w 以 patch 为单位（merge 之前）
Segment = tuple[str, int | tuple[int, int, int]]  # ("text", n) | ("image", (t, h, w)) in patches


# mrope_positions：为 prefill 序列生成 Qwen2-VL 风格的 3 维 MRoPE 位置 [3, T]（轴 0=时间 t，1=高 h，2=宽 w）。
#   - 文本：三个轴取同一个递增下标（退化为普通 1-D RoPE）
#   - 图像：LLM 网格 (t, h//merge, w//merge)，三个轴分别是网格坐标 + 段起点 start
#   - 每段之后 start = 当前段的最大位置 + 1（不是"已有 token 数"）
# 因为图像段占用的位置跨度通常小于 token 数，最终 max 位置 + 1 < T，差值 delta 用于 decode：
#   decode 位置 = 绝对序列下标 index + delta（三个轴相同）
def mrope_positions(segments: list[Segment], spatial_merge: int = 2) -> tuple[torch.Tensor, int]:
    """Qwen2-VL style 3-D rotary positions (temporal, height, width).

    text  : the same running index on all three axes
    image : an LLM grid of (t, h//merge, w//merge) tokens, offset by the running start
    after every segment the next start is max(position so far) + 1
    returns (positions [3, T], mrope_position_delta) with decode position = index + delta
    """
    parts: list[torch.Tensor] = []  # 每段一个 [3, n_seg] 张量，最后沿 dim=1 拼接
    start = 0  # 下一段的起始位置
    for kind, spec in segments:
        if kind == "text":
            n = int(spec)  # type: ignore[arg-type]
            idx = torch.arange(n) + start
            # [n] → expand 成 [3, n]：三行相同（expand 是 view，cat 时会实际拷贝）
            parts.append(idx.expand(3, n))
        elif kind == "image":
            t, h, w = spec  # type: ignore[misc]
            # 视觉 encoder 把 merge×merge 个 patch 合成一个 LLM token；例 (2, 4, 6), merge=2 → 网格 (2, 2, 3) = 12 token
            gh, gw = h // spatial_merge, w // spatial_merge
            # token 按 t 最外、h 其次、w 最内的行优先（row-major）顺序展开，三个坐标各自 flatten 成 [t*gh*gw]
            # 例 (t, gh, gw) = (2, 2, 3)：
            #   ti = 0 0 0 0 0 0 1 1 1 1 1 1
            #   hi = 0 0 0 1 1 1 0 0 0 1 1 1
            #   wi = 0 1 2 0 1 2 0 1 2 0 1 2
            ti = torch.arange(t).view(-1, 1).expand(t, gh * gw).flatten()
            hi = torch.arange(gh).view(1, -1, 1).expand(t, gh, gw).flatten()
            wi = torch.arange(gw).view(1, 1, -1).expand(t, gh, gw).flatten()
            # [3, t*gh*gw]，三个轴统一加上段起点
            parts.append(torch.stack([ti, hi, wi]) + start)
        else:
            raise ValueError(kind)
        # 易错：下一段 start 取本段最大位置 + 1（三轴取 max），而不是 start + token 数。
        # 例：上面 12 个 token 的图像段 max=2 → 下一段 start=3。空段（n=0 或网格为 0）不推进 start
        if parts[-1].numel():
            start = int(parts[-1].max()) + 1
    # 没有任何段时返回 [3, 0] 的 long 张量，保证形状/类型一致
    pos = torch.cat(parts, dim=1) if parts else torch.zeros(3, 0, dtype=torch.long)
    # delta = (prefill 之后下一个位置) - T。例：图像 (2,4,6) + 文本 2 → T=14, max=4 → delta=5-14=-9
    # 纯文本时 max+1 == T → delta=0；delta=0 不代表没有图像，只代表位置跨度恰好等于 token 数
    delta = (int(pos.max()) + 1 if pos.numel() else 0) - pos.shape[1]
    return pos, delta


# decode_position：decode 阶段每个新 token 都是文本，三个轴位置相同 = index + delta。
# 例：接上例 T=14, delta=-9 → 第一个 decode token（index=14）位置 [5, 5, 5]，紧接 prefill 的 max=4
def decode_position(index: int, delta: int) -> torch.Tensor:
    """Position (3,) of the token at absolute sequence index `index` during decode."""
    return torch.full((3,), index + delta, dtype=torch.long)
```

C-25 · solutions/p16_init_planner.py

```
import threading
from collections.abc import Callable
from concurrent.futures import ThreadPoolExecutor
from typing import Any


# InitError：携带"第一个失败的 stage_id"，原始异常通过 `raise ... from cause` 挂在 __cause__ 上
class InitError(Exception):
    def __init__(self, stage_id: int, cause: BaseException) -> None:
        super().__init__(f"stage {stage_id} failed to initialize: {cause!r}")
```


© 2026 kzhou003

```
        self.stage_id = stage_id


# init_groups：多 stage 启动规划的第一步——把"共享 GPU 的 stage"归为一组。
# 同一张卡上的 stage 必须串行初始化（显存 profiling / 分配会互相干扰），不同组之间可以并行。
# 共享关系是传递的（A–B 共享 dev0，B–C 共享 dev1 → A、B、C 同组），用并查集（union-find）求连通分量。
# 例：{0: {2}, 1: {0}, 2: {0, 1}, 3: {1}, 4: {5}} → [[0], [1, 2, 3], [4]]
# 输出确定：组内 stage 升序，组之间按首元素升序。
def init_groups(stage_devices: dict[int, set[int]]) -> list[list[int]]:
    """Stages that share (transitively) a device form one group -> must init sequentially."""
    parent = {s: s for s in stage_devices}

    def find(x: int) -> int:
        while parent[x] != x:
            # 路径减半（path halving）：把 x 挂到祖父节点上，摊还后树很浅
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x

    # owner：device → 第一个占用它的 stage；后来者遇到已占用的 device 就和 owner 合并
    owner: dict[int, int] = {}
    # 按 stage 升序遍历：输入 dict 的 key 顺序可能是乱的，排序保证结果与插入顺序无关
    for s in sorted(stage_devices):
        for d in stage_devices[s]:
            if d in owner:
                a, b = find(s), find(owner[d])
                if a != b:
                    # 总是让较小的根当父亲 → 每个分量的根 = 分量内最小的 stage id
                    parent[max(a, b)] = min(a, b)
            else:
                owner[d] = s
    groups: dict[int, list[int]] = {}
    # 升序遍历再 append → 组内天然有序
    for s in sorted(stage_devices):
        groups.setdefault(find(s), []).append(s)
    return sorted(groups.values(), key=lambda g: g[0])


# run_init：按组并发初始化（组间并行、组内串行），任一 stage 失败则：
#   1) 短路：其它组在开始下一个 stage 前检查到失败就停止（已在运行的 init 无法打断，会跑完）
#   2) 回滚：等所有线程结束后，对所有已成功的 stage 调用 cleanup_fn（释放显存/进程）
#   3) 抛出 InitError，stage_id 为"第一个"记录下的失败
def run_init(
    stage_devices: dict[int, set[int]],
    init_fn: Callable[[int], Any],
    cleanup_fn: Callable[[int, Any], None],
    max_workers: int = 4,
) -> dict[int, Any]:
    groups = init_groups(stage_devices)
    # lock 保护 results 与 failure（多个 worker 线程并发读写）
    lock = threading.Lock()
    results: dict[int, Any] = {}
    failure: list[tuple[int, BaseException]] = []  # 最多一个元素：第一次失败

    # 每个组一个任务，组内 for 循环天然串行
    def run_group(group: list[int]) -> None:
        for sid in group:
            with lock:
                if failure:
                    return  # short-circuit: do not start new work after a failure
            # init_fn 可能很慢（加载权重），必须在锁外调用，否则组间就失去并行
            try:
                res = init_fn(sid)
            except BaseException as e:  # noqa: BLE001
                # 捕获 BaseException：KeyboardInterrupt 等也要触发回滚，不能让已起来的 stage 泄漏
                with lock:
                    if not failure:
                        failure.append((sid, e))
                # 本组后续 stage 不再初始化
                return
            with lock:
                results[sid] = res

    with ThreadPoolExecutor(max_workers=max_workers, thread_name_prefix="stage-init") as ex:
        # list(...) 强制消费迭代器 → 等所有组完成；run_group 自己吞掉了异常，这里不会抛
        list(ex.map(run_group, groups))

    # 易错：回滚必须在线程池退出（所有 init 都结束）之后做，否则可能漏掉正在初始化、稍后才成功的 stage
    if failure:
        for sid, res in results.items():
            cleanup_fn(sid, res)  # roll back everything that did come up
        sid, cause = failure[0]
        raise InitError(sid, cause) from cause
    return results


# admission_check：启动前的显存准入检查（只读、不分配）。对每张卡：
#   need = Σ_{该卡上的 stage} ( int(gpu_memory_utilization × total) + reserve_bytes_per_stage )
```


© 2026 kzhou003

```
#   need > total 就报错；恰好相等允许。跨卡 stage（如 TP=2）在它占用的每张卡上都计一份。
# 例（80 GiB 卡，reserve=2 GiB）：dev1 上两个 stage util=0.6/0.3 → (48+2)+(24+2)=76 ≤ 80 → 通过
def admission_check(
    device_total_bytes: dict[int, int],
    stages: dict[int, tuple[set[int], float]],  # stage -> (devices, gpu_memory_utilization)
    reserve_bytes_per_stage: int,
) -> list[str]:
    errors = []
    # 按 device 升序输出，报错顺序确定
    for dev, total in sorted(device_total_bytes.items()):
        on_dev = [s for s, (devs, _) in stages.items() if dev in devs]
        # 易错：utilization 是相对"这张卡的总量"的比例，要乘本卡 total；int() 向下截断到整字节
        need = sum(int(stages[s][1] * total) + reserve_bytes_per_stage for s in on_dev)
        if need > total:
            errors.append(f"device {dev}: stages {sorted(on_dev)} need {need} B > {total} B")
    return errors
```

C-26 · solutions/p17_graph_runner.py

```
from collections.abc import Callable

import torch


# ---------------------------------------------------------------- given types
# CpuGraph：在 CPU 上模拟 CUDA graph 的关键语义——capture 时绑定静态输入/输出 buffer，
# replay 只会在"同一块"buffer 上重跑，看不到新的 Python 参数。
class CpuGraph:
    """Emulates torch.cuda.CUDAGraph semantics on CPU: replay() re-runs the recorded work on
    the *same static buffers*; it never sees new Python arguments."""

    def __init__(self, fn: Callable[[torch.Tensor], torch.Tensor], static_in: torch.Tensor, static_out: torch.Tensor):
        self._fn, self._in, self._out = fn, static_in, static_out
        self.captured = False

    def capture(self) -> None:
        self._out.copy_(self._fn(self._in))
        self.captured = True

    def replay(self) -> None:
        assert self.captured
        self._out.copy_(self._fn(self._in))


# ---------------------------------------------------------------- implementation
# GraphRunner：decode 阶段的 CUDA graph 分桶（bucket）执行器。
#   - 为每个 bucket 大小 b 捕获一张图，所有图共享同一对 [max_bs, hidden] 静态 buffer 的前 b 行（view）
#   - run 时选择 >= bs 的最小 bucket，把输入拷进 static_in[:bs]，padding 行清零，replay，再拷出前 bs 行
#   - bs 超过最大 bucket、或 force_eager（如 prefill / 调试）时走 eager
# 例：buckets=[1, 4, 16]，bs=3 → bucket 4（第 4 行是 padding）；bs=17 → eager
class GraphRunner:
    def __init__(self, fn: Callable[[torch.Tensor], torch.Tensor], buckets: list[int], hidden: int,
                 dtype: torch.dtype = torch.float32) -> None:
        self.fn = fn
        # 排序后 _bucket_for 的线性扫描第一个命中即为最小可用 bucket
        self.buckets = sorted(buckets)
        self.max_bs = self.buckets[-1]
        # 静态 buffer 按最大 bucket 分配一次；形状 [max_bs, hidden]
        self.static_in = torch.zeros(self.max_bs, hidden, dtype=dtype)
        self.static_out = torch.zeros(self.max_bs, hidden, dtype=dtype)
        self.graphs: dict[int, CpuGraph] = {}
        self.stats = {"replays": 0, "eager": 0, "last_bucket": None}

    def capture(self) -> None:
        # 从大到小捕获：CUDA 上各图共享 memory pool，先捕获最大的图，小图可以复用它已申请的内存
        for b in sorted(self.buckets, reverse=True):  # largest first (shares a memory pool on CUDA)
            # static_in[:b] 是 view（共享存储），不是拷贝 → 之后写 static_in 的前 bs 行，图就能"看到"
            g = CpuGraph(self.fn, self.static_in[:b], self.static_out[:b])
            g.capture()
            self.graphs[b] = g

    def _bucket_for(self, bs: int) -> int | None:
        # 向上取整到最小的 bucket；例 buckets=[1, 4, 16]：bs=1→1, bs=2..4→4, bs=5..16→16, bs=17→None
        for b in self.buckets:
            if bs <= b:
                return b
        return None

    def run(self, x: torch.Tensor, *, force_eager: bool = False) -> torch.Tensor:
        bs = x.shape[0]
        b = None if force_eager else self._bucket_for(bs)
        # `b not in self.graphs`：还没 capture 就调用 run 时也安全地退回 eager
        if b is None or b not in self.graphs:
            self.stats["eager"] += 1
```


© 2026 kzhou003

```
            return self.fn(x)
        # 易错：必须 copy_ 进静态 buffer，不能 self.static_in = x（图绑定的是原来那块内存）
        self.static_in[:bs].copy_(x)
        # 易错：padding 行 [bs, b) 可能残留上一次更大 batch 的数据，清零保证结果确定、不受历史影响。
        # 例：上次 bs=10 写了 10，这次 bs=3、bucket=4 → 第 4 行(下标 3)清零；下标 4.. 在 bucket 外不必管
        self.static_in[bs:b].zero_()  # stale rows from a previous, larger batch
        self.graphs[b].replay()
        self.stats["replays"] += 1
        self.stats["last_bucket"] = b
        # 易错：必须 clone。直接返回 static_out[:bs] 这个 view，下一次 replay 会悄悄改写调用方手里的结果
        return self.static_out[:bs].clone()  # next replay overwrites static_out
```

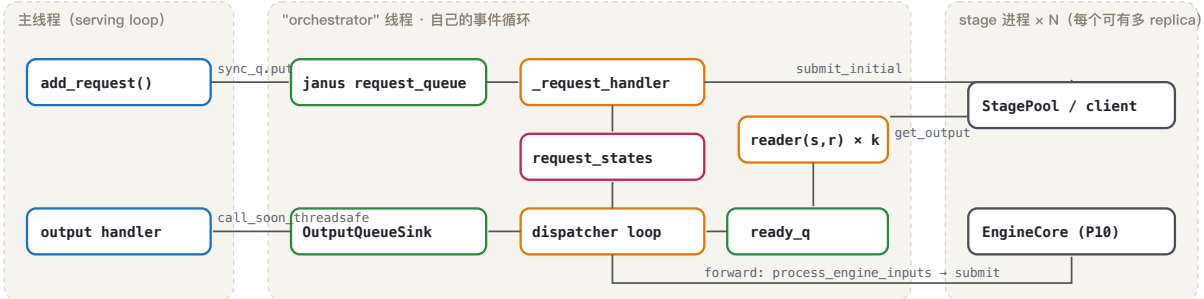

© 2026 kzhou003

附录 D · P02 Orchestrator ↔ vllm-omni 源码对照

练习：`mine/p02_orchestrator.py` · 源码：vllm-project/vllm-omni @ fa8b419 · vllm_omni/engine/orchestrator.py（所有链接固定在该 commit，行号不会漂移）

一句话结论

P02 的骨架和真实代码**一模一样**：`_request_handler` 处理控制消息，**每个 stage replica 一个 reader** 汇入同一个 `ready_q`，**一个 dispatcher** 串行路由，终态由 `final_output_stage_ids ⊆ finished` 决定，终态后清理。最大的差别在 **T（执行拓扑）**：真实 orchestrator 跑在**单独的线程**里，通过 janus 队列和 `OutputQueueSink`（P01）与主线程通信。



逐项对照表

P02	vllm-omni	源码
`class Orchestrator`	`OrchestratorBase`（通用骨架）+ `Orchestrator(OrchestratorBase)`（turn-based 子类，负责 add）	orchestrator.py:322 orchestrator.py:3033
`ReqState`	`OrchestratorRequestState`：`final_output_stage_ids`、`finished_final_output_stage_ids`、`finished_stage_ids`、`pending_final_output`	orchestrator.py:233-272
`request_states`	`self.request_states: dict[str, OrchestratorRequestState]`	orchestrator.py:390
`_req_q`	`request_async_queue`：**janus** 队列，主线程 `sync_q.put`，orchestrator 线程 `async_q.get`	omni_engine_base.py:308 add_request → sync_q.put
`_out_q`	`output_async_queue` = `OutputQueueSink`：一批输出只做一次 `call_soon_threadsafe` 唤醒（**就是 P01**）	omni_engine_base.py:309-310 async_engine_utils.py:40-50
`start()`	`run()`：create_task 出 request handler 和 output handler，然后 gather	orchestrator.py:479-512
`shutdown()`	`run()` 的 `finally`：cancel + `gather(*tasks, return_exceptions=True)`，最后再扫一遍 `all_tasks`	orchestrator.py:530-555
`_request_handler`	`_request_handler`：按 `msg.type` 分派 add / abort / collective_rpc / register / shutdown	orchestrator.py:557-592

© 2026 kzhou003

`_handle_add`	`_handle_add_request`：**先** `self.request_states[request_id] = req_state`，**再** `submit_initial`	orchestrator.py:3062 注册 :3117 · submit :3128
`_reader(sid)`	`_llm_replica_reader(stage_id, replica_id, client)`：**每个 (stage, replica)** 一个 reader；出错也作为 `("error", …)` 消息交给 dispatcher	orchestrator.py:1117-1136
`_ready_q`	`ready_q: asyncio.Queue[(kind, stage_id, replica_id, payload)]`	orchestrator.py:1112
`_dispatcher`	`_orchestration_loop_event_driven` 的主循环：串行消费 `ready_q`	docstring 主循环 :1264
`_handle_output`	`_handle_processed_outputs` → `_route_output`	orchestrator.py:1381-1412 orchestrator.py:1912-2043
未知 id：`dropped_outputs += 1`	`req_state is None`：打 warning 并 `continue`（没有计数器）	orchestrator.py:1385-1393
终态 `final_stage_ids ⊆ finished`	`request_finished = final_output_stage_ids.issubset(req_state.finished_final_output_stage_ids)`	orchestrator.py:1936-1940
发给用户 `_out_q.put(OrchOutput)`	`OutputMessage(…, finished=request_finished)` → `output_async_queue.put`	orchestrator.py:1944-1967
转发 `forward_fn(s+1, prompt, data)`	`_forward_to_next_stage` → `next_client.process_engine_inputs(source_outputs, req_state.prompt)` → `custom_process_input_func`（如 `thinker2talker_full_payload`）	触发条件 :1988 process_engine_inputs :2602 stage_engine_core_client.py:455-471 thinker2talker_full_payload
`_cleanup(rid)`	`_cleanup_request_ids`：释放 binding、传输资源，`self.request_states.pop(...)`	orchestrator.py:1699-1762 pop :1756
`pending_final`（P03）	`pending_final_output`：async_chunk 下暂扣终态，等所有已提交 stage 都结束再放行	orchestrator.py:1954-1965 orchestrator.py:2036-2040
abort（P04）	`_handle_abort` → `_cleanup_request_ids(abort=True)` → `_abort_request_ids` 对每个 pool `abort_requests`	orchestrator.py:645-683 orchestrator.py:685-731 stage_pool.py:1279
dead replica（P04）	reader 发 `("error", …EngineDeadError)` → dispatcher `_handle_dead_replica` 驱逐并继续服务	orchestrator.py:1297-1312 orchestrator.py:1464

© 2026 kzhou003

建议阅读路线（约 300 行）

① 线程与事件循环的边界（T）

主线程（HTTP / serving loop）调用 `add_request()`，把消息 `sync_q.put` 进 janus 队列；一个名叫 `"orchestrator"` 的后台线程里 `new_event_loop()`，在那里跑 `orchestrator.run()`。**这是 P02 替你省掉的一层**：P02 的调用方和 orchestrator 在同一个事件循环里。

omni_engine_base.py:303-330 · janus 队列 + orchestrator 线程

```
303  # Janus queues are constructed eagerly here (not deferred to the
304  # orchestrator thread) so the master server's ROUTER thread always
305  # sees a non-None ``self.request_queue`` when on_register fires.
306  # ``async_q`` lazily binds to whatever event loop first awaits on
307  # it (the orchestrator loop), so cross-thread use stays correct.
308  self.request_queue: janus.Queue[EngineQueueMessage] = janus.Queue(maxsize=_REQUEST_QUEUE_MAXSIZE)
309  self.output_queue: janus.Queue[EngineQueueMessage] = janus.Queue()
310  self._output_sink = OutputQueueSink(self.output_queue.sync_q)
311  self.rpc_output_queue: janus.Queue[EngineQueueMessage] = janus.Queue()
312  self._shutdown_called = False
313  self._weak_finalizer: weakref.finalize | None = None
314  self._correlated_rpc_client: CorrelatedRpcClient | None = None
315  self._running_counter = OmniRequestCounter()
316  self._engines_waiting_counter = OmniRequestCounter()
317
318  logger.info(f"[OmniEngine] Launching Orchestrator thread with {self.num_stages} stages")
319
320  # Launch orchestrator background thread
321  startup_future: concurrent.futures.Future = concurrent.futures.Future()
322
323  self.orchestrator_thread = threading.Thread(
324      target=self._bootstrap_orchestrator,
325      args=(
326          stage_init_timeout,
327          startup_future,
328      ),
329      daemon=True,
330      name="orchestrator",
```

omni_engine_base.py:440-452 · 线程里新建事件循环

```
440  def _bootstrap_orchestrator(
441      self,
442      stage_init_timeout: int,
443      startup_future: concurrent.futures.Future,
444  ) -> None:
445      """Create loop, initialize stages, then run Orchestrator."""
446
447      loop = asyncio.new_event_loop()
448      asyncio.set_event_loop(loop)
449
450      async def _run_orchestrator() -> None:
451          self._initialize_stages(stage_init_timeout)
452
```


© 2026 kzhou003

② 先注册，再 submit

在 `_handle_add_request` 里找到这两句的先后顺序：注册 `request_states` 发生在任何 `await` 之前。

orchestrator.py:3107-3138

```
3107  req_state = OrchestratorRequestState(
3108      request_id=request_id,
3109      prompt=original_prompt,
3110      sampling_params_list=sampling_params_list,
3111      final_stage_id=final_stage_id,
3112      final_output_stage_ids=final_output_stage_ids,
3113      request_timestamp=float(msg.request_timestamp or _time.time()),
3114      mm_features=getattr(prompt, "mm_features", None),
3115      request_artifact_dirs=set(msg.request_artifact_dirs or ()),
3116  )
3117  self.request_states[request_id] = req_state
3118  self._maybe_attach_native_kv_transfer_params(req_state, prompt)
3119  self._register_running_request(req_state)
3120  req_state.streaming.enabled = bool(getattr(prompt, "resumable", False))
3121  req_state.stage_submit_ts[stage_id] = _time.time()
3122  enqueue_ts = msg.enqueue_ts
3123  if enqueue_ts > 0:
3124      req_state.pipeline_timings["queue_wait_ms"] = (_time.perf_counter() - enqueue_ts) * 1000.0
3125  preprocess_ms = msg.preprocess_ms
3126  if preprocess_ms > 0:
3127      req_state.pipeline_timings["preprocess_ms"] = preprocess_ms
3128  if not await self._dispatch_or_fail_request(
3129      lambda: self.stage_pools[stage_id].submit_initial(
3130          request_id,
3131          req_state,
3132          prompt,
3133          prompt_text=msg.output_prompt_text,
3134      ),
3135      req_id=request_id,
3136      stage_id=stage_id,
3137      operation="add_request",
3138  ):
```

③ reader：只搬运，连错误也只是搬运

注意 `except BaseException`：reader 自己出错时**不改任何状态**，而是把错误包装成一条消息交给 dispatcher。这是单写者原则的延续。

orchestrator.py:1117-1136

```
1117  async def _llm_replica_reader(stage_id: int, replica_id: int, client: Any) -> None:
1118      try:
1119          pool = self.stage_pools[stage_id]
1120          while not self._shutdown_event.is_set():
1121              # Read through the pool so this loop shares the legacy
1122              # poll's keep/drop rule and MR V2 payload rehydration. Only
1123              # a fully empty batch comes back as None. A poll-style
1124              # client returns one instead of blocking, so keep the
1125              # legacy 1 ms cadence for those; a blocking client only
1126              # lands here rarely, where 1 ms is irrelevant.
1127              raw_outputs = await pool.read_llm_raw_output(client)
1128              if raw_outputs is None:
1129                  await asyncio.sleep(0.001)
1130                  continue
1131              await ready_q.put(("llm", stage_id, replica_id, raw_outputs))
1132              self._orch_monitor.set_dispatch_queue_size(ready_q.qsize())
1133      except asyncio.CancelledError:
1134          raise
1135      except BaseException as e:  # noqa: BLE001 — routed to the dispatcher
1136          await ready_q.put(("error", stage_id, replica_id, e))
```


© 2026 kzhou003

④ dispatcher：串行消费 ready_q

真实代码同时等三件事：新输出、关闭信号、replica 变化。所以用了 `asyncio.wait(FIRST_COMPLETED)`，并在每轮重建 `pending_get`。这正是我们讨论过的"用 wait 自己管 task"的写法，只用在 dispatcher 这一处。

orchestrator.py:1264-1272

```
1264  while not self._shutdown_event.is_set():
1265      if pending_get is None:
1266          pending_get = asyncio.create_task(ready_q.get(), name="orch-dispatch-get")
1267      done, _ = await asyncio.wait(
1268          {pending_get, shutdown_task, reconcile_task},
1269          timeout=max(0.0, next_reconcile - _time.monotonic()),
1270          return_when=asyncio.FIRST_COMPLETED,
1271      )
1272      if shutdown_task in done:
```

orchestrator.py:1291-1293

```
1291  kind, stage_id, replica_id, payload = pending_get.result()
1292  pending_get = None
1293  # Record the queue after dequeue so the gauge reflects work
```


© 2026 kzhou003

⑤ _route_output：P02 _handle_output 的真身

顺序：标记 finished → 计算 `request_finished` → final stage 就发 `OutputMessage` → 若 `finished and stage_id < final_stage_id` 就转发 → `request_finished` 则清理。

orchestrator.py:1933-1967 · 终态判断 + 发给用户

```
1933  if finished and not segment_finished:
1934      req_state.finished_stage_ids.add(stage_id)
1935
1936  request_finished = False
1937  if finished and self.stage_pools[stage_id].final_output and not segment_finished:
1938      req_state.finished_final_output_stage_ids.add(stage_id)
1939      final_output_stage_ids = req_state.final_output_stage_ids or {req_state.final_stage_id}
1940      request_finished = final_output_stage_ids.issubset(req_state.finished_final_output_stage_ids)
1941  if req_state.session_owned:
1942      # Session-owned outputs are delivered through the session interceptor.
1943      pass
1944  elif self.stage_pools[stage_id].final_output:
1945      message = OutputMessage(
1946          request_id=req_id,
1947          stage_id=stage_id,
1948          replica_id=replica_id,
1949          engine_outputs=output,
1950          metrics=stage_metrics,
1951          finished=request_finished,
1952          stage_submit_ts=submit_ts,
1953      )
1954      if (
1955          request_finished
1956          and self.async_chunk
1957          and not req_state.streaming.enabled
1958          and not set(req_state.stage_submit_ts).issubset(req_state.finished_stage_ids)
1959      ):
1960          # Direct chunk delivery can finish the decoder before the
1961          # upstream engine's terminal output/metrics reaches this loop.
1962          # Keep request state until those messages are routed, otherwise
1963          # the frontend publishes incomplete usage and loses stop reasons.
1964          req_state.pending_final_output = message
1965          request_finished = False
1966      else:
1967          await self.output_async_queue.put(message)
```

orchestrator.py:1988-1993 · 转发条件

```
1988  if (
1989      (finished or segment_finished)
1990      and stage_id < req_state.final_stage_id
1991      and not req_state.skip_legacy_stage_forward
1992      and (not self.async_chunk or not self._stage_receives_async_chunks(stage_id + 1))
1993      and (not self._next_stage_already_submitted(stage_id, req_state) or req_state.streaming.enabled)
```

orchestrator.py:2036-2043 · pending_final 放行 + 清理

```
2036  pending = req_state.pending_final_output
2037  if pending is not None and set(req_state.stage_submit_ts).issubset(req_state.finished_stage_ids):
2038      req_state.pending_final_output = None
2039      await self.output_async_queue.put(pending)
2040      request_finished = True
2041
2042  if request_finished and not req_state.session_owned:
2043      await self._cleanup_request_ids([req_id, *self._cfg_tracker.cleanup_parent(req_id)])
```


© 2026 kzhou003

⑥ run() 的 finally：cancel + gather

我们讨论过的关闭写法就在这里，外加最后一遍兜底：扫 `asyncio.all_tasks(loop)`，把漏网的 task 也取消并等待。

orchestrator.py:530-555

```
530      finally:
531          self._shutdown_event.set()
532          for task in tasks:
533              if not task.done():
534                  task.cancel()
535          try:
536              await asyncio.gather(*tasks, return_exceptions=True)
537          except Exception:
538              pass
539          await self._shutdown_extensions()
540
541          if self._membership is not None:
542              await self._membership.drain_tasks(timeout=10.0)
543              self._membership.shutdown()
544
545          self._orch_monitor.flush()
546          self._shutdown_stages()
547
548          loop = asyncio.get_running_loop()
549          pending = [t for t in asyncio.all_tasks(loop) if t is not asyncio.current_task() and not
t.done()]
550          for task in pending:
551              task.cancel()
552          if pending:
553              await asyncio.gather(*pending, return_exceptions=True)
554
555  # ---- Request handling ----
```

真实代码比 P02 多出来的东西

多出来的	为什么需要	对应题	源码
每个 stage 多个 **replica**；`_reconcile_readers` 动态增删 reader	replica 会注册、死掉、被替换	P04	orchestrator.py:1159-1233
dispatcher 用 `asyncio.wait({ready_q.get(), shutdown, reconcile})`	除了输出，还要响应关闭和成员变化	—	orchestrator.py:1264-1289
reader 出错发 `("error", …)`，dispatcher 驱逐该 replica 并继续	故障隔离：一个 replica 挂了不拖垮整体	P04	orchestrator.py:1297-1312
转发时 `source_outputs = [output]`，不自己攒 chunk	stage 的 output processor 已累积好输出；P02 用 `stage_data` 模拟	—	orchestrator.py:2366
orchestrator 在独立线程，经 janus 和 `OutputQueueSink` 与主线程通信	让 HTTP 主循环不被路由逻辑拖慢	P01	async_engine_utils.py:40-50
diffusion stage 用 poller 轮询	diffusion 客户端只有非阻塞 poll 接口	—	orchestrator.py:1138-1157

© 2026 kzhou003

CFG companion、PD 分离、streaming、session、metrics	业务功能	—	orchestrator.py:1912-1931

读完之后想一想

1. 真实 orchestrator 在**另一个线程**里。主线程调用 `add_request` 时，为什么必须走 janus 的 `sync_q.put`，而不能直接调用 orchestrator 循环里某个 `asyncio.Queue.put_nowait`？
2. `_route_output` 里，发给用户的 `OutputMessage` 是在转发给下一个 stage **之前**放进队列的。如果反过来，先 `await _forward_to_next_stage`（一次跨进程通信）再发给用户，用户体感上会有什么差别？
3. reader 出错时为什么不直接调用 `_handle_dead_replica`，而要把错误包成消息交给 dispatcher？

© 2026 kzhou003

PART II

大作业：从零实现 nano-vllm-omni

再合起来做：在单张 RTX 5090 上用 Qwen2.5-Omni-3B 搭一个完整的多 stage 推理服务。共 12 个里程碑，每个都有验收标准和评分，总分 100 分，另有加分项。

© 2026 kzhou003

大作业：从零实现 nano-vllm-omni

目标：在**单张 RTX 5090（32 GB）**上 serve **Qwen2.5-Omni-3B**，支持文本、图像、音频输入，文本和语音流式输出，多请求并发，接口兼容 OpenAI。

参考实现：vllm-omni `fa8b4199c`（源码链接指向该 commit），vLLM v0.31.0，nano-vllm，HF transformers。

---

0. 作业说明与最终交付

你要做的不是"调用 vLLM"，而是**自己实现一个最小但完整的多阶段推理服务**。实现过程中会遇到 vllm-omni 解决过的所有核心问题：paged KV cache 与 continuous batching；跨 stage 的数据传递；流水线并行流式输出；多进程并发正确性；单卡上多个 stage 的显存划分；CUDA graph。

最终交付物

1. `nano_vllm_omni/` 源码。建议控制在 4k–7k 行（不含借用的 HF 模块）。
2. `nvo serve Qwen/Qwen2.5-Omni-3B`：启动一个 OpenAI 兼容服务，`/v1/chat/completions` 支持 `modalities=["text","audio"]` 和 `stream=true`。
3. `tests/`：每个里程碑的 parity 测试和并发测试（见各里程碑的"验收"一栏）。
4. `bench/`：输出 TTFT、TPOT、TTFP（首包音频延迟）、RTF、吞吐，覆盖并发 1、4、8。
5. `REPORT.md`：说明架构、设计取舍、显存预算、性能曲线，以及你和 vllm-omni 的差异。

> **怎么用这份文档**
> 每个里程碑按"目标 → 你要自己写的部分 → 接口骨架 → 验收标准 → 提示（折叠）→ 参考源码"组织。**建议先不展开提示**，自己设计、写完、跑通验收之后，再展开提示和参考源码对照。这样才能真正检验自己。

---

1. 目标模型与硬件预算

1.1 为什么选 Qwen2.5-Omni-3B 而不是 Qwen3-Omni

模型	结构	单卡 5090 可行性
Qwen3-Omni-30B-A3B	MoE thinker + MoE talker + 16 层 RVQ MTP + Code2Wav	bf16 权重约 60 GB 以上，**放不下**。量化后仍然非常紧，不适合作业。
Qwen2.5-Omni-3B ✅	稠密 thinker（Qwen2.5-3B 级 LLM + 音频编码器 + ViT）+ 小 talker（单 codebook）+ Token2Wav（DiT flow-matching + BigVGAN）	bf16 权重大约十几 GB，具体以 `model.safetensors.index.json` 的 `metadata.total_size` 为准。三个 stage 同卡共存，还能留出可观的 KV 空间。
Qwen2.5-Omni-7B	同上，thinker 更大	加分项：bf16 也能放下但很紧，需要认真做显存预算，或者给 thinker 上 FP8。

它和 Qwen3-Omni 在架构理念上一致（Thinker–Talker 加 vocoder），所以你学到的并发设计可以直接迁移。vllm-omni 中对应的实现：`omni(/) model_executor/models/qwen2_5_omni/pipeline.py:18–65`。

© 2026 kzhou003

1.2 模型三段（务必先读懂 HF 参考实现）

Stage	输入	输出	要点
0 Thinker	文本 token + 图像/视频/音频特征	文本 token（返回用户）；每个位置的 embedding 和 hidden（给 talker）	MRoPE / TMRoPE（按时间对齐音视频位置）；多模态 embedding 合并
1 Talker	thinker 的 prompt embedding 和 hidden，加上逐步的 reply 向量	codec token（单 codebook）。`START=8293`、`END=8294`（stop token）、`PAD=8292`	`thinker_to_talker_proj` 把 thinker 维度投影到 talker 维度；每步输入 = codec embedding + 本步 thinker reply 向量；需要抑制 id ≥ num_embeds+1 的 token
2 Token2Wav	codec token、说话人条件（cond / ref_mel，来自 `spk_dict.pt`，如 Chelsie / Ethan）	24k 波形（以 config 为准）	DiT 用 RK4 ODE 解 flow matching（vllm-omni 用 10 步），每个 code 对应 2 个 mel 帧，再经 BigVGAN 转成波形

参考：`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:241-410`（三个 stage 的 forward）、`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:760-847`（talker 的 prefill 和 decode 输入构造）、`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:895-982`（codec 转音频）、`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni_token2wav.py:601`（RK4 求解器）。HF 金标准：`↗ modeling_qwen2_5_omni.py`（重点读 `Qwen2_5OmniForConditionalGeneration.generate` 里 thinker 到 talker 的拼接方式）。

1.3 RTX 5090 环境须知

- Blackwell 架构（sm_120），32 GB 显存。需要 CUDA 12.8 及以上，以及支持 sm_120 的 PyTorch（cu128 wheel）。
- FlashAttention、FlashInfer 等第三方 kernel 对 sm_120 的支持请**自行确认版本**。作业主线建议：prefill 用 PyTorch SDPA 或你自己写的 Triton varlen kernel，decode 用**你自己写的 Triton paged attention**（这本身就是核心练习）。
- 单卡多进程共享 GPU 时，不同进程的 kernel 默认是**时间片轮转**，不会真正并发。打开 **CUDA MPS** 后可以并发执行（加分项，见 M10）。vllm-omni 也提供了 `cuda_mps` 开关：`omni(/) engine/cuda_mps.py`。

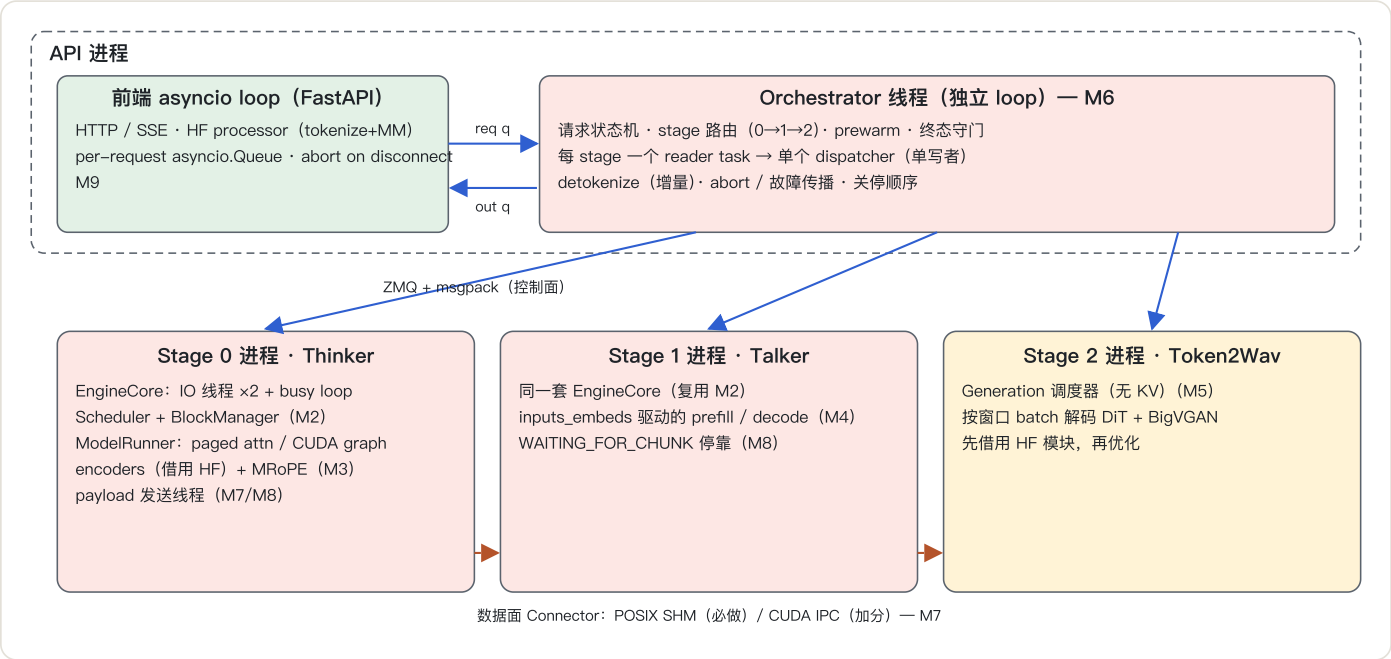
1.4 显存预算（自己算，写进 REPORT）

```
KV bytes / token = 2 (K,V) × num_layers × num_kv_heads × head_dim × dtype_bytes
例：若 thinker text_config 为 36 层、2 个 KV head、head_dim=128、bf16
    → 2×36×2×128×2 = 36,864 B ≈ 36 KB/token → 8k token ≈ 0.29 GB
（以 config.json 的 thinker_config.text_config / talker_config 实际数值为准）
```

项目	Thinker	Talker	Token2Wav
权重	LLM + 音频编码器 + ViT（最大头）	小 LLM + 投影 + 需要 thinker 的 `embed_tokens`（见 M4）	DiT + BigVGAN
KV cache	主要消耗	按 talker config 计算（codec 序列长，约为 thinker 文本长度的数倍）	无（非自回归）
激活峰值	大图或长音频 prefill 时的编码器激活	小	DiT 对整段 mel 的 attention，与音频长度相关
CUDA graph	每个 capture 的 batch size 都占一份 graph pool，**必须计入预算**（vllm-omni 默认路径就漏算过 talker MTP graph，见 `omni(/) worker/base.py:140-147`）		

© 2026 kzhou003

2. 目标架构总览



颜色含义：红 = 核心，必须自己写　绿 = 可借用　黄 = 先借用、后替换或优化 。整体结构与 vllm-omni 一致：1 个 API 进程，每个 stage 一个 EngineCore 进程；控制面走 orchestrator，数据面由 worker 点对点传输。区别在于全部在单卡上，所以三个 stage 进程共用 `cuda:0`。

3. 自己写 vs 借用 清单

组件	归属	说明
Paged KV cache 与 BlockManager（含 prefix caching 可选）	自己写	理解 vLLM 的核心抽象
Scheduler（continuous batching、chunked prefill、抢占）	自己写	多 stage 的各种等待状态都挂在这里
ModelRunner：输入准备（positions / slot_mapping / block_tables）、采样	自己写	per-request seed、top-k/p、repetition penalty
Paged decode attention kernel（Triton）	自己写	prefill 可以先用 SDPA
Thinker LLM 解码器与 Talker LLM 解码器（Qwen2 结构，RMSNorm / RoPE / MRoPE / GQA / SwiGLU）	自己写	为了接入你的 paged attention 和 CUDA graph
按 stage 拆分权重与加载	自己写	safetensors 流式按前缀加载
Thinker 到 Talker 的输入构造（prefill 拼接、逐步 reply 向量）	自己写	正确性最容易出错的地方
EngineCore 进程（IO 线程、busy loop）、ZMQ 协议	自己写	参考 vLLM 的设计
Orchestrator、请求状态机、abort、故障、关停	自己写	本作业的并发核心
SHM Connector 与 chunk 适配器（后台线程、key、清理）	自己写	async_chunk 的基础
Decode CUDA graph（静态 buffer、batch bucket、padding）	自己写	性能关键

© 2026 kzhou003

Tokenizer、chat template、`Qwen2_5OmniProcessor`（音频特征、图像 patch）	借用 HF	只在前端用
音频编码器、视觉编码器（ViT）模块	借用 HF 的 `nn.Module`	自己负责加载权重、放到设备上、调用，以及合并 embedding
Token2Wav（DiT + BigVGAN）	先借用 HF 或 vllm-omni 的实现	加分：自己实现 batch 化与 CUDA graph
FastAPI / uvicorn、pyzmq、msgspec 或 msgpack	借用	—

4. 推荐代码结构

```
nano_vllm_omni/
  config.py            # PipelineSpec / StageSpec / DeploySpec (dataclass, YAML 可选)
  weights.py           # safetensors 按前缀流式加载、名字映射
  models/
    qwen2_decoder.py   # 自写：RMSNorm、RoPE/MRoPE、GQA attention (调用 paged attn)、MLP
    thinker.py         # LLM + 借用的 audio/vision encoder + embedding 合并 + hidden 导出
    talker.py          # thinker_to_talker_proj + LLM + codec head + 输入构造
    token2wav.py       # 包装 HF/omni 的 DiT + BigVGAN，暴露 decode(codes, spk) 接口
  engine/
    sequence.py        # Request / Sequence 状态（含 WAITING_FOR_CHUNK）
    block_manager.py   # paged KV 分配、引用计数、prefix hash
    scheduler.py       # AR 调度器 + Generation 调度器
    model_runner.py    # 输入准备、forward、采样、CUDA graph
    sampler.py
    core_proc.py       # EngineCore 进程：IO 线程 + busy loop
    core_client.py     # 在 orchestrator loop 中使用的异步 ZMQ client
  kernels/
    paged_attention.py # Triton decode kernel (+ 可选 prefill)
  connector/
    shm.py             # POSIX SHM put/get + 锁 + reader unlink
    chunk_adapter.py   # recv/save 线程、key 规则、停靠与归还
  orchestrator.py      # 请求状态机、路由、prewarm、终态守门、abort
  frontend/
    engine.py          # 前端 ↔ orchestrator 线程桥接（janus 或自写）
    server.py          # OpenAI 兼容 API、SSE
tests/  bench/  scripts/golden_hf.py
```

5. 里程碑路线图

建议节奏：一个里程碑约 3–7 天，全程大约 8–10 周。**严格按顺序推进，每一步都必须先通过验收**。前面的 parity 测试会一直保留，作为后面所有优化的回归测试。

M0 · 环境与金标准（golden reference） 5 分

目标：用 HF transformers 跑通 Qwen2.5-Omni-3B 的文本加语音输出，并**把每个 stage 的中间张量存盘**，作为之后所有 parity 测试的标准答案。

自己写：`scripts/golden_hf.py`。固定 3–5 个 prompt（纯文本、带图、带音频、带视频可选），使用 greedy 解码，talker 用固定 seed 或 greedy，保存以下内容：

© 2026 kzhou003

- thinker：`input_ids`、MRoPE `position_ids`（3×T）、每层或至少第 0 层和最后一层的 hidden、logits（前若干步）、生成的 token；
- talker：prefill 的 `inputs_embeds`、每步输入向量、生成的 codec id；
- token2wav：codec 输入、mel、波形 wav；
- 峰值显存（`torch.cuda.max_memory_allocated`）和端到端耗时。

验收：能够复现 HF 的输出；golden 文件可以被 `pytest` 加载。

提示

- 注册 forward hook 抓取中间张量比改 HF 源码更干净。
- talker 的采样有随机性。做 parity 时要么用 greedy，要么固定 seed，并在你的实现里复现**同一个采样顺序**。更简单的做法是"teacher forcing"：直接把 golden 的 codec 喂进去，只比较 logits。
- bf16 下不要期望 bitwise 一致。比较 logits 用 `atol≈1e-2` 再加 argmax 一致率；波形用 SNR 或 mel-L1。

M1 · 权重拆分与按 stage 加载 8 分

目标：实现 `load_stage(model_dir, stage) -> nn.Module`，每个 stage 只构建自己的模块，并且只读取自己需要的张量。

自己写：

- 读取 `model.safetensors.index.json`，按前缀 `thinker.`、`talker.`、`token2wav.` 只打开相关 shard，用 `safe_open` 的 `get_tensor` 逐个加载并**直接拷到 GPU**，避免整份 checkpoint 先进 CPU 内存。
- 名字映射：HF 名字到你的模块名（参考 vllm-omni 的 `WeightsMapper` 写法）。
- 特殊依赖：**talker stage 还需要 thinker 的** `embed_tokens`，用来把 thinker 的文本 token 嵌入成向量；token2wav 需要 `spk_dict.pt`。
- 加载完整性检查：报告未加载和多余的参数名，并**报错**，不要只打印 warning。

```
def load_stage(model_dir: str, stage: Literal["thinker","talker","token2wav"],
               device: torch.device, dtype=torch.bfloat16) -> nn.Module: ...
def iter_stage_tensors(model_dir: str, prefixes: tuple[str, ...]) -> Iterator[tuple[str, torch.Tensor]]: ...
```

验收：(1) 三个 stage 的参数集合的并集等于 checkpoint 全集；(2) 每个 stage 加载后的显存和 safetensors metadata 计算出的字节数一致；(3) thinker 用**你自写的 decoder** 做一次 teacher-forcing forward，logits 与 golden 一致。

提示

- vllm-omni 的做法是每个 stage 都完整迭代 checkpoint，然后在 `load_weights` 里按前缀分流（`omni(/)model_executor/models/qwen2_5_omni/qwen2_5_omni.py:983-1044`）。你可以做得更好：根据 index 只打开相关 shard。
- talker 借用 thinker embedding 的位置：`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:1009-1022`。
- Qwen2 的 q/k/v 有 bias；o_proj 没有。可以把 qkv 打包成一个 Linear，减少 kernel 数。

© 2026 kzhou003

M2 · 单 stage LLM 引擎核心（nano-vllm 级别） 20 分

目标：先**只做 thinker 的纯文本**，在单进程里实现 continuous batching 加 paged KV cache 的推理引擎。`LLMEngine.generate(prompts, sampling_params)` 的输出与 golden 一致。

自己写：

1. **Sequence / Request**：状态（WAITING / RUNNING / FINISHED_*，**预留** WAITING_FOR_CHUNK 和 WAITING_FOR_INPUT）、token 列表、`num_computed_tokens`、block table。
2. **BlockManager**：固定 block_size（如 16）；free list；按需分配；释放；（可选）按 block 内容 hash 做 prefix caching，配合引用计数。
3. **Scheduler**：每个 step 在 `max_num_batched_tokens` 和 `max_num_seqs` 限制内，先调度 running（decode），再调度 waiting（prefill，可做 chunked prefill）；KV 不够时抢占（recompute）。
4. **ModelRunner**：把一个 step 的调度结果展平成 `input_ids`、`positions`、`slot_mapping`、`block_tables`、`context_lens`、`cu_seqlens`；执行 forward；只对每个序列最后一个位置算 logits。
5. **Paged attention**：写 KV（按 slot_mapping scatter）；prefill 用 varlen causal attention；decode 用你自写的 Triton kernel 按 block table 读 KV。
6. **Sampler**：greedy、temperature、top-k、top-p、repetition penalty；**per-request seed**（每个请求一个 `torch.Generator`，并说明 batch 中如何保证结果与 batch 组成无关）。

```
class Scheduler:
    def add(self, req: Request) -> None
    def schedule(self) -> SchedulerOutput          # 本步要跑哪些请求、每个请求跑几个 token
    def update(self, out: RunnerOutput) -> list[RequestOutput]   # 追加 token、判停、释放 block
class ModelRunner:
    def execute(self, so: SchedulerOutput) -> RunnerOutput
```

验收：

- □ 单请求 greedy 生成的 token 与 golden 完全一致（至少前 64 个 token）。
- □ 8 个不同长度的请求同时提交，每个请求的结果与单独运行时一致（batch 不变性）。
- □ 人为把 KV 预算调小以触发抢占，结果仍然正确。
- □ Triton decode kernel 与 naive attention 的单元测试误差在容差内。

提示

- nano-vllm 是非常好的参考，大约 1200 行：`↗ nano-vllm`。建议先自己写，卡住了再看。
- vLLM 对应的模块：`↗ scheduler.py` `↗ kv_cache_manager.py` `↗ gpu_model_runner.py`。
- 先写最简单的"一步只做 prefill 或只做 decode"，再支持混合 batch。
- KV 预算：先 profile 一次最大 batch 的 forward，量出激活峰值，再用 `预算 − 权重 − 激活峰值 − graph 预留` 计算 block 数。多 stage 共卡时**不要**用"空闲显存比例"，要给每个 stage 一个**显式字节预算**（与 vLLM 的 `kv_cache_memory_bytes` 思路相同）。

M3 · 多模态输入与 MRoPE 8 分

目标：thinker 支持图像、音频（视频可选）输入，并且结果与 golden 一致。

自己写：

© 2026 kzhou003

- 前端调用 HF processor 得到 `input_ids`（含占位 token）、`pixel_values`、`image_grid_thw`、`input_features` 等，并**通过 IPC 发给 stage 0**。要自己设计张量序列化格式（msgpack + 原始 bytes，或 SHM）。
- 在 stage 0 中运行借用的编码器，把编码器输出 scatter 到占位 token 的位置，得到 `inputs_embeds`。
- 实现 MRoPE 的 3×T 位置计算：文本三维相同，图像按 (t, h, w) 网格，音频线性；视频音频交错（TMRoPE）作为加分项。decode 阶段的位置 = 上一个位置 + 1 + 偏移量（`mrope_position_delta`）。
- 调度器把编码器计算计入 token budget，或者单独设置预算，防止一张大图把一个 step 撑爆。

验收：带图和带音频的 prompt，`position_ids` 与 golden 完全一致；生成的 token 前 32 个一致。

提示

- 参考：`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:412-576`（`get_mrope_input_positions`，含 `seconds_per_chunk` 的音视频交错）。
- prefix caching 遇到多模态时，block hash 必须包含多模态内容的 hash，否则不同图片会命中同一份 KV。
- chunked prefill 时编码器只算一次：用 encoder cache（以 request 和输入为 key），把结果分到多个 step 消费。

M4 · Talker stage（单进程串联版） 10 分

目标：在**同一进程**里先串起来：thinker 跑完，取出 embedding 和 hidden，构造 talker 的输入，talker 用你的引擎（复用 M2）生成 codec。这一步先不考虑多进程。

自己写：

- thinker 每一步导出 talker 需要的向量（按 HF generate 的定义，通常是第 0 层 embedding 与最后一层 hidden 的某种组合），并按请求累积。
- talker 的 prefill `inputs_embeds`：thinker 的 prompt 部分，加上 `[spk_token_embed + codec_pad_embed]`，再加上 `[第一个回复向量 + codec_bos_embed]`。prompt token id 用 mask、pad、start 占位。
- talker 每步 decode 的输入 = 上一步采样出的 codec 的 embedding + 第 k 个 thinker 回复向量；回复向量耗尽后改用 pad 或 eos 文本向量（以 HF 为准）。
- 引擎要支持"**以 inputs_embeds 驱动**"的请求，这样 talker 的 prefill 和 decode 都可以传入外部向量（扩展你在 M2 写的 runner）。
- 采样约束：stop token 8294；抑制 id > token2wav `num_embeds` 的 token；输出时去掉 START、END、PAD。

验收：用 teacher forcing 时，talker 每一步的 logits 与 golden 一致；用 greedy 或固定 seed 时，codec 序列与 golden 一致（或者在 >95% 的步上一致，并在报告中说明原因）。

提示

- 参考：`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:686-847`（`talker_preprocess`、prefill 拼接、逐步 decode）、`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni_talker.py:56-106`（`thinker_to_talker_proj`）、`omni(/) model_executor/stage_input_processors/qwen2_5_omni.py:42-63`（去掉 codec 边界 token）。
- thinker 生成结束时，vLLM V1 会先把 stop token 追加进 output 再判停，所以 hidden 会**多一行**，必须裁掉：`omni(/) model_executor/stage_input_processors/qwen2_5_omni.py:261-317`。这是典型的"差一行"bug。

© 2026 kzhou003

- 把"跨 stage 输入构造"写成**纯函数**：`build_talker_prefill(payload) -> (ids, embeds)`。M7 和 M8 换成跨进程传输时，这个函数完全不用改。

M5 · Token2Wav stage 7 分

目标：codec 转波形，先整段解码，再做**分块流式**解码。

自己写：

- `Token2WavRunner.decode(codes, speaker) -> wav`：加载 cond 和 ref_mel，初始化噪声，DiT 用 RK4 积分 N 步得到 mel，再经 BigVGAN 得到波形。
- 分块流式：每 W 个 code 解码一次，带左上下文 L，只输出新部分对应的波形，拼接处做重叠或交叉淡化。**确定性**：同一请求的噪声要用 per-request seed，保证分块和整段结果尽量接近。
- 一个 Generation 调度器：无 KV、无采样，把同时就绪的多个请求窗口 pad 后 batch 处理。

验收：整段解码与 golden 波形的 mel-L1 或 SNR 达到阈值（自定，写进报告）；分块与整段的差异可量化，且听感无明显接缝。

提示

- 参考：`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py:895-982`（cond 和 ref_mel 的回退逻辑、`factor`、`steps=10`）、`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni_token2wav.py:1038-1180`（`process_chunk*`，包括 DiT 和 BigVGAN 的 batch 版本）。
- Qwen3-Omni 的 Code2Wav 用"左上下文重算再切片"的无状态方案（`omni(/) model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:447-508`），可以借鉴来设计你的窗口策略。

M6 · 多进程拓扑与 Orchestrator 14 分

目标：三个 stage 拆成三个 EngineCore 子进程，由 API 进程里的 orchestrator 线程串联。数据先用"整段 payload 经 orchestrator 转发"（最慢但最简单），M7 再优化。

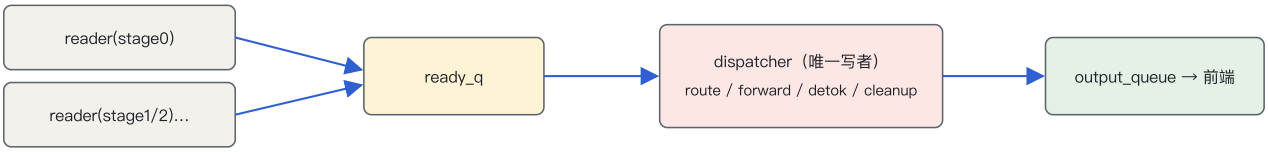
自己写：

1. **EngineCore 进程**（参考 vLLM `EngineCoreProc`）：
   - 输入 IO 线程：ZMQ PULL 或 ROUTER，解码 msgpack 后放入 `queue.Queue`；
   - busy loop 线程：取消息（ADD / ABORT / UTILITY）→ `schedule()` → `execute()` → `update()` → 放入输出队列；没有工作时阻塞在输入队列上，不要空转；
   - 输出 IO 线程：序列化后 ZMQ PUSH；
   - SIGTERM 时设置标志并往输入队列塞一个 WAKEUP，避免 busy loop 永远阻塞。
2. **启动与显存**：父进程**按顺序** spawn 三个 stage（同卡），每个 stage 用显式的 `kv_cache_bytes`；子进程完成"加载 → profile → 分配 KV → capture graph"之后发 READY 握手；任何一个 stage 失败都要回收其他 stage。可选：用跨进程 `flock` 把"测量空闲显存"这一步串行化。
3. **Orchestrator 线程**（自己的 asyncio loop）：
   - 每个 stage client 一个 reader task，等待 ZMQ 输出，然后放进一个 `ready_q`；**单个** dispatcher 串行处理，从而保证请求状态只有一个写者；
   - 请求状态：`stage_submit_ts`、`finished_stages`、`final_stage_ids={0(text),2(audio)}`；

© 2026 kzhou003

- stage 0 完成后构造 stage 1 的请求（调用 M4 的纯函数），stage 1 完成后构造 stage 2 的请求；
- 增量 detokenize 放在 orchestrator（每个请求维护 offset）。

4. **前端桥接**：请求队列有界（提供背压），输出队列无界；跨线程唤醒**批量合并**（每批只 `call_soon_threadsafe` 一次），并处理 lost-wakeup。




```
@dataclass
class ReqState:
    request_id: str
    prompt: Any
    stage_submit_ts: dict[int, float]
    finished_stages: set[int]
    final_stage_ids: frozenset[int]
    pending_final: Optional[OutputMsg] = None   # M8 终态守门用
class Orchestrator:
    async def run(self): ...                      # 启动 readers + dispatcher + request handler
    async def _on_stage_output(self, stage_id: int, outs: list[EngineOutput]): ...
    async def _forward(self, req: ReqState, src: int): ...
    async def _cleanup(self, req_ids: list[str], abort: bool): ...   # 唯一出口
```

验收：

- □ 端到端：文本加音频输出与 M4、M5 单进程版本一致。
- □ 并发 8 个请求，每个请求的文本 token 顺序正确，`finished` 恰好出现一次且是最后一条。
- □ 空闲时三个子进程和 orchestrator 的 CPU 占用接近 0（没有 busy-poll）。
- □ Ctrl-C 后所有进程退出，无僵尸进程，`nvidia-smi` 显存全部释放。

提示

- 阅读顺序：`omni(/) engine/omni_engine_base.py:300-340`（线程和队列）→ `omni(/) engine/async_engine_utils.py:40-97`（批量唤醒）→ `omni(/) engine/omni_engine_base.py:973-991`（drain → arm → drain → wait）→ `omni(/) engine/orchestrator.py:1099-1320`（reader 加 dispatcher）→ `omni(/) engine/orchestrator.py:3062-3162`（先注册状态，再提交）。
- 请求状态**先注册再发送**。否则输出可能比状态先到达。
- dispatcher 的临界区内不要 `await`，否则单写者的保证就失效了。确实需要 await 时，先把状态改完再 await。
- 子进程用 `spawn` 而不是 `fork`（CUDA 不支持 fork）。

M7 · 数据面 Connector（绕开 orchestrator） 8 分

目标：thinker 到 talker 的 hidden 和 embedding（体积大），以及 talker 到 token2wav 的 codec，改为由 worker **点对点**直接传输。orchestrator 只传一个"占位请求"，其中包含长度信息和 sender 信息。

自己写：`SHMConnector.put(key, tensors)` / `get(key) -> tensors | None`：

- key 是确定性的：`{req_id}_{src_stage}_{chunk_id}`，这样双方不需要额外的元数据通道就能对上；

© 2026 kzhou003

- 写者先持有同名 lockfile 的排他 `flock`，再创建并写入 POSIX shm；读者用 `LOCK_EX|LOCK_NB` 探测是否写完；
- **读者负责 unlink**（单消费者）；写者定期回收已被消费的 key；abort 或关停时按前缀批量清理；
- 接收端 stage 新增 `WAITING_FOR_INPUT` 状态：请求已到达但 payload 还没到时，**不进入调度队列**；后台 recv 线程拿到数据后，交回调度线程，在**调度线程**上落地。

验收：结果与 M6 一致；orchestrator 进程的 IPC 字节量下降一个数量级（打点统计）；跑 100 个请求（含 abort）后 `/dev/shm` 无残留。

提示

- 参考：`omni(/) distributed/omni_connectors/connectors/shm_connector.py:195-364`、`omni(/) entrypoints/stage_utils.py:193-237`。
- 加分：单卡场景可以用 CUDA IPC（`torch.multiprocessing` 共享 CUDA tensor，或 `cudaIpcGetMemHandle`），实现零 D2H。难点在于**生命周期**：生产者必须保证在消费者用完之前不释放这块显存，并且要用 event 同步跨进程的流。

M8 · async_chunk 流式流水线 12 分

目标：thinker 每 decode 一个 token，就把对应的回复向量发给 talker；talker 每攒够 W 个 codec，就发给 token2wav。三个 stage **同时工作**，首包音频延迟（TTFP）从"整段时长"降到亚秒级。

自己写：

1. **ChunkAdapter**（每个 stage 进程一个）：
   - `save_async(req, payload)`：在调度线程上只做**快照和入队**；save 线程负责构造 payload 并执行 `put`；
   - `load_async(req)`：recv 线程轮询 `get`，失败时退避；条件变量的 timeout 作为 lost-wakeup 的兜底；
   - chunk_id 在 put 成功后才在锁内递增；terminal chunk（finished 标志）**不能被 cancel 吞掉**；
   - cleanup 与"提交收到的 chunk"之间用锁和 generation 检查做栅栏，防止迟到的 chunk 被写进已清理的请求。
2. **调度器钩子**：`schedule()` 前把等待 chunk 的请求**移出** waiting/running 队列（设为 WAITING_FOR_CHUNK）；在 `finally` 中把 chunk 已就绪的请求放回去。
3. **Prewarm**：orchestrator 收到请求后，立刻向 stage 1 和 stage 2 提交占位请求。这些占位请求停在 WAITING_FOR_CHUNK，第一个 chunk 一到就能直接开始。
4. **终态守门**：只有当所有已提交的 stage 都 finished 时，才向前端发送 `finished=True`。在那之前，终态消息先存在 `pending_final`。
5. **首块小、后块大**：例如首块 4 个 code，之后每块 25 个（或者做成可配置的 ramp）。在报告中给出 TTFP 和 RTF 的权衡曲线。

验收：

- □ 开启 async_chunk 后音频与非流式版本"听感一致"，并量化差异（talker 采样相同时 codec 序列应完全一致）。
- □ 并发 1 时 TTFP 比 M7 降低一个数量级以上（给出数字）。
- □ 压力测试：1000 个请求，随机 10% 中途 abort，没有请求永久挂起，没有 SHM 泄漏，进程不崩溃。

提示

© 2026 kzhou003

- 参考：`omni(/) distributed/omni_connectors/transfer_adapter/base.py:20-140`、`omni(/) distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:381-503`（save_async 快照）、`omni(/) distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1120-1186`（process_pending_chunks）、`omni(/) core/sched/omni_ar_scheduler.py:408-451`（schedule 前后钩子）、`omni(/) engine/orchestrator.py:1954-1967`（终态守门）、`omni(/) engine/orchestrator.py:2711-2870`（prewarm）。
- Qwen3-Omni 的 chunk 处理函数是很好的模板：`omni(/) model_executor/stage_input_processors/qwen3_omni.py:409-542`（thinker 到 talker）、`omni(/) model_executor/stage_input_processors/qwen3_omni.py:765-842`（talker 到 code2wav 的窗口）。
- talker 的 prompt 长度依赖 thinker 的 prompt 长度，但 prewarm 时 thinker 还没开始跑。解决办法：前端已经 tokenize 过，orchestrator 知道 prompt 长度，所以 prewarm 的占位 prompt 可以用 `[0]*expected_len`。
- 如果你实现了 async scheduling（加分），发送端只能发"已确认"的 token：`num_computed - num_placeholders`。

M9 · OpenAI 兼容服务 5 分

目标：`POST /v1/chat/completions`，支持 `messages` 中的 `image_url` 和 `input_audio`，支持 `modalities`、`audio: {voice, format}`、`stream`。

自己写：SSE 中文本 delta 与音频 delta（base64 PCM 或 wav 分片）**交错**输出；客户端断开时（`CancelledError`）用 `asyncio.shield` 执行 abort，并设置超时；`/health` 在任一 stage 进程死亡时返回 503。

验收：用 `openai` Python SDK 和 curl 都能流式拿到文本与音频；在流式过程中 kill 客户端，三个 stage 都收到 abort，KV block 和 SHM 都被回收（打印计数）。

提示

- 参考：`omni(/) entrypoints/async_omni.py:325-332`（shield abort）、`omni(/) entrypoints/async_omni_base.py:460-493`（abort 后合成终态，防止挂起）。

M10 · 性能：CUDA graph、IO 与计算重叠 10 分

目标：让 thinker 和 talker 的 decode 不再受 kernel launch 开销限制，并且跨 stage 的 D2H 拷贝不阻塞计算。

自己写：

1. **Decode CUDA graph**：预分配静态 buffer（`input_ids`、`positions`、`slot_mapping`、`block_tables`、`context_lens`、输出 hidden），对 batch bucket（1, 2, 4, 8, 16, ...）逐个 capture；运行时 pad 到最近的 bucket，再 replay。注意：graph 内不能有 Python 分支、CPU 同步或动态 shape。
2. **Graph 显存计入预算**：capture 之前和之后各测一次显存，写进 KV 预算。
3. **异步 payload D2H**：用独立的 copy stream，先 `wait_stream(compute)`，**先 clone 再拷贝**（graph buffer 会被下一次 replay 覆盖），拷到 pinned host 内存，调用 `record_stream`，再记录一个 event；后台线程等 event 完成后再 put。
4. **（选做）CUDA MPS**：开启后三个进程的 kernel 可以真正并发执行。对比开和关时并发 4 的 RTF。

验收：开启 CUDA graph 后结果与 eager 完全一致；thinker 在 batch=1 时的 TPOT 降低（给出数字）；用 Nsight Systems 截图说明 D2H 与计算重叠。

© 2026 kzhou003

提示

- 参考：`omni(/) worker/gpu_ar_model_runner.py:160-297`（异步输出快照：clone、pinned、record_stream、event、builder 线程）、`omni(/) worker/gpu_ar_model_runner.py:1730-1739`（pageable 内存导致 D2H 退化为同步的教训）、`omni(/) worker/gpu_ar_model_runner.py:634-690`（按 bucket capture 的写法）、`omni(/) worker/gpu_model_runner.py:243-272`（静态 buffer 大小的选择）。
- 带 per-request seed 的 batch 不能直接 replay graph（graph 不会重新执行 Python 的 generator 逻辑）。要么把 RNG 状态做成 graph 内的张量，要么这类 batch 走 eager：`omni(/) worker/gpu_model_runner.py:1998-2010`。

M11 · 正确性与鲁棒性 8 分

目标：任何时刻的 abort、任何一个 stage 进程崩溃、正常关停，系统都能收敛到一致状态。

自己写并测试：

- **Abort 两阶段**：先从 detokenizer 收集部分输出（暂不提交），再让 EngineCore abort，成功后才提交 abort 状态；多个 final stage 时只有最后一条带 finished。
- **Abort 与 ADD 的顺序**：在 EngineCore 中，abort 必须在它之前的 ADD 被处理之后才生效，防止"先 abort 再 add"导致请求复活。
- **故障隔离**：`kill -9` 掉 talker 进程后，正在使用 talker 的请求收到错误，纯文本请求不受影响，`/health` 报告不健康。
- **关停顺序**：先停 orchestrator，再停 stage 进程；所有操作都是幂等的。
- **终态单调**：清理之后到达的输出直接丢弃；对已经不存在的请求做 update 不会让它复活。

验收：编写一个随机故障注入测试（随机 abort、随机延迟 SHM、随机 kill stage），跑 30 分钟，满足：每个请求**恰好一个**终态，没有资源泄漏（KV block 数回到初始值，`/dev/shm` 为空，没有僵尸进程）。

提示

- 参考：`omni(/) engine/stage_pool.py:1279-1339`（abort 两阶段）、`omni(/) engine/stage_engine_core_proc.py:147-154`（abort 与 ADD 的 FIFO 栅栏）、`omni(/) engine/orchestrator.py:1464-1515`（死进程只影响绑定在它上面的请求）、`omni(/) engine/omni_engine_base.py:1132-1220`（关停顺序）。

6. 基准与验收

指标	定义	建议目标（自己用 HF 基线校准）
TTFT	请求到达 → 第一个文本 token	短 prompt、并发 1 时达到百毫秒级
TPOT	文本 token 间隔	开启 CUDA graph 后明显低于 eager
TTFP	请求到达 → 第一个音频 chunk	async_chunk 开启后达到亚秒级（并发 1）
RTF	生成耗时 ÷ 音频时长	并发 1 时 < 1（实时），并发 4 时尽量 < 1
吞吐	并发 1、4、8 时的 req/s 与音频秒数/s	给出曲线

© 2026 kzhou003

显存	每个 stage 的权重、KV、graph、激活	写进 REPORT，并解释余量

参考：vllm-omni 在 Qwen3-Omni 上开启 async_chunk 后，并发 1 时 TTFP 从约 6.5 s 降到约 0.52 s（H800，`omni(/)docs/design/feature/async_chunk.md`）。你的模型更小，可以作为数量级上的对照。

---

7. 评分标准（满分 100 + 加分）

里程碑	分值	关键判据
M0 金标准	5	golden 可复现、可加载
M1 加载	8	按前缀加载、完整性检查、thinker logits parity
M2 引擎核心	20	paged KV、continuous batching、抢占、batch 不变性、Triton kernel
M3 多模态	8	MRoPE 位置完全一致、encoder cache
M4 Talker	10	输入构造正确、去掉 stop 行、codec parity
M5 Token2Wav	7	整段 parity、分块流式无明显接缝
M6 多进程	14	IO 线程、单写者 dispatcher、背压、无 busy-poll、干净退出
M7 Connector	8	点对点传输、读者 unlink、无泄漏
M8 async_chunk	12	停靠与归还、prewarm、终态守门、TTFP 改善
M9 API	5	SSE 文本加音频、断开即 abort
M10 性能	10	CUDA graph parity、显存计入、D2H 重叠
M11 鲁棒性	8	故障注入 30 分钟：每个请求恰好一个终态、零泄漏
REPORT	—	不单独计分，但缺失则每个里程碑最多拿 70%

---

8. 加分项（每项 +3～+8）

- **Async scheduling**：step N+1 的调度与 step N 的 GPU 计算重叠，使用 placeholder token，由 GPU 端填回真实 id（参考 `↗ async_scheduler.py`）。
- **CUDA IPC 数据面**：同卡零拷贝，正确管理生命周期和跨进程 event。
- **Token2Wav 的 batch 化与 CUDA graph**：多个请求的窗口按长度分组后 replay（参考 `omni(/)model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:63-121` 的 `plan_decode_groups`）。
- **Prefix caching（多模态安全）**：system prompt、同一张图多轮对话时复用 KV。
- **FP8 thinker**：在 5090 上跑 Qwen2.5-Omni-7B，并给出质量对比。
- **单进程多 stream 版本**：三个 stage 在同一进程、三个 CUDA stream 上运行，与多进程方案对比延迟和复杂度，写进报告。
- **Realtime / WebSocket 双工**：流式音频输入加 VAD。

© 2026 kzhou003

9. 常见坑合集（遇到问题再来查）

正确性

- thinker 判停时多出一行 hidden，talker 吃进去后音频末尾出现杂音。见 M4 提示。
- talker 的 KV、head 数、max_len 必须用 **talker 自己的 config**，误用 thinker 的会导致 attention 形状错误（vllm-omni 中有相同的注释：`omni(/) config/model.py:268-285`）。
- MRoPE 的 decode 位置要加 `mrope_position_delta`，不能简单地用 `len(tokens)`。
- talker 输出的特殊 id（START、END、PAD、MASK）进入 token2wav 会导致越界或输出长度异常：`omni(/) model_executor/stage_input_processors/qwen2_5_omni.py:42-63`。
- bf16 RMSNorm 在 fp32 中累加；`torch.compile` 的某些融合会改变数值，parity 失败时先关掉 compile 排查。

并发

- 输出比请求状态先到达：一定要先注册状态再提交。
- lost wakeup：标准流程是"drain → arm → drain again → wait"，或者条件变量加 timeout 兜底。
- recv 线程直接修改调度器里的请求对象，与调度线程发生数据竞争：应该把数据交给调度线程，由它落地。
- cancel 把 terminal chunk 也吞掉了，下游请求永远停在 WAITING_FOR_CHUNK。
- 在 asyncio loop 上做阻塞 `queue.put`，整个前端会卡住（vllm-omni 也存在这个风险：`omni(/) engine/async_omni_engine.py:584`）。
- 子进程继承了父进程设置的 `CUDA_VISIBLE_DEVICES` 或其他环境变量：spawn 时设置环境变量要加锁，spawn 完立即恢复（`omni(/) engine/stage_engine_startup.py:911-934`）。

显存与 GPU

- 三个 stage 并发初始化时，"测量空闲显存"互相干扰，导致 KV 算多而 OOM：要么串行初始化，要么用显式的字节预算，要么用跨进程读写锁（`omni(/) engine/stage_phase_lock.py:106-238`）。
- CUDA graph 的 pool 没有计入预算。
- pageable 内存的 D2H 会退化成同步拷贝；graph 输出 buffer 不先 clone 就异步拷贝，数据会被下一次 replay 覆盖。
- 跨流使用张量时忘了 `record_stream`，caching allocator 可能提前复用这块显存，出现随机数据错误。

10. 参考资料

- 概念与并发设计导读：`本书 Part III`
- nano-vllm（单 stage 引擎的极简参考）：`↗ GeeeekExplorer/nano-vllm`
- vLLM V1：`↗ engine/core.py` `↗ engine/core_client.py` `↗ core/sched/scheduler.py` `↗ worker/gpu_model_runner.py`
- HF 金标准：`↗ modeling_qwen2_5_omni.py`，模型卡 `↗ Qwen/Qwen2.5-Omni-3B`
- vllm-omni Qwen2.5-Omni 实现：`omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni.py` `omni(/) model_executor/models/qwen2_5_omni/qwen2_5_omni_token2wav.py` `omni(/) model_executor/stage_input_processors/qwen2_5_omni.py` `omni(/) deploy/qwen2_5_omni.yaml`
- vllm-omni 设计文档：`omni(/) docs/design/feature/async_chunk.md` `omni(/) docs/design/module/engine_orchestration.md` `omni(/) docs/design/module/stage_runtime.md`
- Triton 教程（写 paged attention 前必读）：`↗ triton-lang.org tutorials`
- ZeroMQ 指南：`↗ zguide`

© 2026 kzhou003

注：本文中模型尺寸和显存数字仅用于估算，请以 checkpoint 的 config.json 和 safetensors 元数据为准。vllm-omni 的 Qwen2.5-Omni 默认部署是 `max_num_seqs: 1`、`enforce_eager`、非 async_chunk（`omni(/) deploy/qwen2_5_omni.yaml`），所以本作业的 M8 和 M10 实际上超出了参考实现在这个模型上的完成度，属于真正的挑战部分。

© 2026 kzhou003

PART III

参考：vLLM-Omni 并发设计深度解读

以 Qwen3-Omni 为例，讲清楚真实系统如何做到高效而准确地运行：进程和线程拓扑、orchestrator、stage 进程、async_chunk 数据面、GPU 重叠、abort、故障和关停。做题和写大作业时随时翻回来对照。

© 2026 kzhou003

vLLM-Omni 并发设计深度解读

以 Qwen3-Omni-MoE（thinker → talker → code2wav）为主线 · 源码基于 vllm-omni `fa8b4199c`（main）· 上游 vLLM v0.31.0

0. 阅读说明

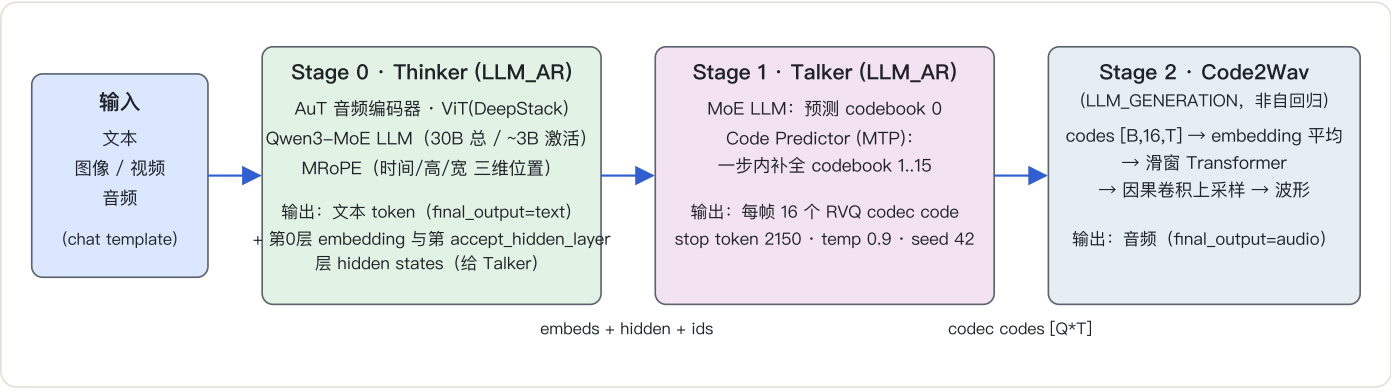
这份文档回答一个问题：**一个多阶段（multi-stage）的 omni 模型服务，是如何同时做到"高效"（各阶段流水线重叠、IO 与计算重叠、GPU 不空转）和"准确"（不丢消息、不乱序、abort/故障后状态收敛）的?**

- `⟨/⟩ path:line` 样式的链接指向 GitHub 上 vllm-omni 固定 commit 的具体行，点开即可读源码。
- `vLLM ↗ file` 样式的链接指向上游 vLLM v0.31.0 的文件（只给到文件级别，因为本地没有对应版本的 checkout 来核对行号）。
- 彩色框：vLLM 概念 解释上游基础；为什么 解释设计动机；注意 是坑或已知风险。

一句话总结整个设计：**"一个 API 进程 + 每个 stage 一个 vLLM EngineCore 子进程"**。API 进程里，前端 asyncio 循环和一个独立的 orchestrator 线程（有自己的事件循环）通过 janus 队列通信。orchestrator 用单线程协作式调度来保证请求状态的一致性。stage 之间的控制消息走 orchestrator（ZMQ），大块数据和流式 chunk 则绕过 orchestrator，由 worker 通过 OmniConnector（共享内存等）直接点对点传输，并由后台线程和独立 CUDA stream 与计算重叠。

1. 背景：Qwen3-Omni 模型

Qwen3-Omni-30B-A3B 是一个"全模态进、文本 + 语音出"的模型。它采用 **Thinker–Talker** 架构：Thinker 负责"想"（理解 + 生成文本），Talker 负责"说"（把语义转成语音 codec），最后由一个 vocoder 把 codec 解码成波形。vllm-omni 把它切成 3 个 stage，见 `⟨/⟩ model_executor/models/qwen3_omni/pipeline.py:22-84`。



1.1 各组件

组件	做什么	源码
统一 wrapper	`Qwen3OmniMoeForConditionalGeneration`。同一个类根据 `model_stage` 只构建本 stage 需要的子模块（thinker / talker / code2wav），所以三个进程加载的是同一个类的不同"切片"。	`⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:139` `⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:200-352`

© 2026 kzhou003

Thinker 音频编码器 (AuT)	`Qwen3OmniMoeAudioEncoder`，把梅尔谱编码成 LLM token embedding。	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1131
Thinker 视觉编码器	ViT + DeepStack：多个中间层的视觉特征被加到 LLM 前几层里，而不只是输入层。可选的 encoder CUDA graph。	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:1137–1168 ⟨/⟩ model_executor/models/qwen3_omni/vision_encoder_cudagraph.py:26–58
Thinker LLM	Qwen3-MoE。会捕获第 0 层输入（即词 embedding）和第 `accept_hidden_layer` 层的 hidden states 给 Talker 用。	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:435–437 ⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_moe_thinker.py:588–596
Talker	MoE LLM，用 `codec_embedding` 替代词表 embedding，用 `codec_head` 输出 codebook 0 的 logits。`text_projection` / `hidden_projection` 把 thinker 维度投影到 talker 维度。	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_moe_talker.py:356–393
Code Predictor (MTP)	拿到采样出的 CB0 和 talker 最后的 hidden，在一个 talker step 内部循环预测 CB1..CB15（共 15 个 codebook，每个大小 2048）。下一步的输入 embedding 是 16 个 codec embedding 之和加上 text step。	⟨/⟩ model_executor/models/common/qwen3_code_predictor.py:1428 ⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1062–1069
Code2Wav	因果卷积 vocoder：codes 先做 embedding 平均，再经过滑窗 Transformer、TransConv/ConvNeXt 上采样、SnakeBeta 解码，输出 clamp 到 [-1,1]。	⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:124–385

1.2 Talker 一步里发生什么（理解 GPU 并发的关键）

```
# qwen3_omni.py:1062–1069  —— 每个 talker decode step 的前置：MTP 先把这一帧补全
codes, summed = self.talker.code_predictor_forward(input_ids, inputs_embeds,
                                                    last_talker_hidden=last_talker_hidden)
inputs_embeds = summed.reshape(-1, H)                    # 16 个 codebook embedding 之和
inputs_embeds = (inputs_embeds + text_step).reshape(-1, H)  # 再加上这一步对应的 thinker 文本行
return inputs_embeds, codes.squeeze(-1)
```

⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1062–1069。也就是说，**talker 的每个 token 实际上是"一次大模型 forward + 15 次小模型 forward"**。为了不被 kernel launch 开销拖垮，整个 `talker_mtp` 调用会被包成一个 FULL CUDA graph（当 `talker_mtp_graph_safe` 成立时，⟨/⟩ worker/gpu_model_runner.py:255–262），小模型本身用 `torch.compile(dynamic=False)`（⟨/⟩ model_executor/models/common/qwen3_code_predictor.py:906–914）。

1.3 Talker 的 prompt 是怎么拼出来的

Talker 并不直接读用户文本。它的 prompt 由 thinker 的产物拼成（⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1242–1328）：

- 丢掉 system 轮；user 轮中文本位置用 `text_projection(embed)`，音频/图像/视频位置用 `hidden_projection(hidden)`（⟨/⟩ model_executor/models/qwen3_omni/qwen3_omni.py:1448–1506）。

© 2026 kzhou003

- assistant 轮是 9 个 token 的 bootstrap：3 个角色 token + 4× `tts_pad` + `tts_bos` + 第一个文本 token，并与 codec 侧的 `[nothink, think_bos, think_eos, speaker, codec_pad, codec_bos]` 相加（`model_executor/models/qwen3_omni/qwen3_omni.py:1515-1568`）。
- 剩下的 thinker 文本每个 talker step 消耗一行；文本耗尽后先发一次 `tts_eos`，之后一直是 `tts_pad`（`model_executor/models/qwen3_omni/qwen3_omni.py:1364-1407`）。

> 为什么这对并发重要
>
> Talker 第 k 步只需要 thinker 的第 k 个文本行。所以 **thinker 每 decode 出一个 token，就可以把这一行的 embedding 发给 talker**，talker 不必等 thinker 结束。这就是 `async_chunk` 能把首包音频延迟（TTFP）从秒级降到亚秒级的模型层面原因。

1.4 跨 stage 传什么

边	非流式（full payload）	async_chunk（流式）
Thinker → Talker	`embed.prefill`（第 0 层）、`hidden_states.output`（accept 层）、`tts_bos/eos/pad` embedding、`ids.all/prompt`、speaker、language。`model_executor/stage_input_processors/qwen3_omni.py:545-634`	chunk 0 带全部 prefill 数据（会攒到长度超过 prompt 才发）；之后每个 chunk 只带 `embed.decode` `[1, D]`。`model_executor/stage_input_processors/qwen3_omni.py:409-542`
Talker → Code2Wav	`codes.audio` 过滤为合法 codec id [0,2048)，按 codebook-major 展平成 `[Q*T]`。`model_executor/stage_input_processors/qwen3_omni.py:843-915`	按窗口发：chunk 0 = 4 帧（`initial_codec_chunk_frames`），之后每 25 帧（`codec_chunk_frames`）发一次，附带最多 25 帧左上下文。`model_executor/stage_input_processors/qwen3_omni.py:807-827`

```
# stage_input_processors/qwen3_omni.py:807-815 —— 攒够一个窗口才返回 payload，否则返回 None（本步不发）
if configured_initial_chunk_size > 0:
    if chunk_id == 0:
        chunk_size_config = configured_initial_chunk_size
    else:
        length -= configured_initial_chunk_size
chunk_length = length % chunk_size_config
if chunk_length != 0 and not is_finished:
    return None
```

Code2Wav 每个 chunk 都是**无状态**解码：它会重新解码左上下文，然后只切出新帧对应的波形（`model_executor/models/qwen3_omni/qwen3_omni_code2wav.py:447-508`）。好处是 code2wav 请求之间不需要保留跨 chunk 的状态，可以自由 batch（`max_num_seqs: 64`）。

1.5 部署形态（默认 deploy）

`deploy/qwen3_omni_moe.yaml:15-77`：`async_chunk: true`，用 SharedMemoryConnector；stage 0 在 GPU 0（显存利用率 0.9），stage 1 + 2 共用 GPU 1（0.6 + 0.1）。Thinker temperature 0；talker `seed: 42, temperature: 0.9, top_k: 50`（注释解释了为什么要固定 seed：talker 是唯一的随机 stage，不加 seed 时少数采样会"流畅地胡说"，见 `deploy/qwen3_omni_moe.yaml:49-57`）。

© 2026 kzhou003

性能参考（A100，`docs/design/qwen3_omni_tts_performance_optimization.md:246-256`）：在并发 1 下，叠加 batching、CUDA graph、async chunk + streaming 之后，TTFP 从 426 s 降到 1.16 s，RTF 从 2.08 降到 0.21。另一份数据（H800，`docs/design/feature/async_chunk.md`）：并发 1 时开启 async_chunk 让 TTFP 从 6.46 s 降到 0.52 s。

---

2. vLLM 核心概念速成

vllm-omni 的每个 LLM stage 本质上就是一个上游 vLLM V1 引擎。下面这些概念是读懂后文的前提。

vLLM 概念 · EngineCore 与进程分离

vLLM V1 把引擎拆成两半。**前端**（`AsyncLLM`，在 API 进程中）负责 tokenize/detokenize 和流式输出。**EngineCore**（独立进程 `EngineCoreProc`）负责调度和执行模型。两者用 **ZMQ** socket 加 **msgpack** 序列化通信。EngineCoreProc 内部有 3 个线程：**输入 IO 线程**（ZMQ → `input_queue`）、**主 busy loop 线程**（取请求 → `schedule()` → `execute_model()` → `update_from_output()`）、**输出 IO 线程**（`output_queue` → ZMQ）。这样序列化和网络 IO 不会阻塞 GPU 调度。

vLLM ↗ vllm/v1/engine/core.py　vLLM ↗ vllm/v1/engine/core_client.py　vLLM ↗ vllm/v1/engine/async_llm.py

vLLM 概念 · AsyncMPClient

前端访问 EngineCore 的客户端。它在当前 asyncio 循环里创建一个后台 task，不断从 ZMQ 输出 socket 读消息并放入一个 `asyncio.Queue`。`get_output_async()` 就是 await 这个队列。另外有一个 monitor 线程盯住子进程的 sentinel，进程死亡时把引擎标记为 dead，之后的调用抛出 `EngineDeadError`。DP>1 时用 `DPLBAsyncMPClient` 做负载均衡。

vLLM ↗ vllm/v1/engine/core_client.py

vLLM 概念 · Continuous batching / Scheduler / KV cache

Scheduler 每个 step 从 `waiting` 和 `running` 队列中挑请求，在 token budget（`max_num_batched_tokens`）和并发上限（`max_num_seqs`）内组成一个 batch。新请求可以随时加入，结束的请求随时离开，这就是 continuous batching。KV cache 按 block 分页管理（PagedAttention）。**Chunked prefill** 把长 prompt 切成多个 step 处理，好与 decode 混合执行。**Prefix caching** 复用相同前缀的 KV block。

vLLM ↗ vllm/v1/core/sched/scheduler.py　vLLM ↗ vllm/v1/request.py

vLLM 概念 · Async scheduling（异步调度）与 placeholder token

同步模式下，step N+1 的调度必须等 step N 的采样结果从 GPU 拷回 CPU，这会让 GPU 空等。开启 `async_scheduling` 后，scheduler 先假设每个 running 请求会产出 1 个 token（记为 **output placeholder**），提前调度 step N+1；真实的 token id 在 GPU 上直接填入下一步的输入（`-1` 占位）。这样 CPU 调度与 GPU 计算重叠。对应 `AsyncScheduler`、`num_output_placeholders`、`AsyncGPUModelRunnerOutput`（在独立的 `async_output_copy_stream` 上做 D2H 拷贝）。当 `max_concurrent_batches > 1`（如 PP）时，busy loop 使用 `step_with_batch_queue`，可以同时有多个 batch 在飞。

vLLM ↗ vllm/v1/core/sched/async_scheduler.py　vLLM ↗ vllm/v1/worker/gpu_model_runner.py

vLLM 概念 · Executor / Worker / ModelRunner

EngineCore 通过 Executor 驱动 Worker。TP=1 时用 `UniProcExecutor`，worker 就在 EngineCore 进程内；TP>1 时用多进程 executor，每个 rank 一个 worker 进程。ModelRunner 负责准备输入、执行 forward（可能是 CUDA graph replay）、采样。

vLLM ↗ vllm/v1/executor/abstract.py

© 2026 kzhou003

vLLM 概念 · CUDA graph

把一串 GPU kernel 录制成一张图，之后一次 launch 整图重放，消除 Python 和 kernel launch 开销。代价是输入形状必须固定（按 batch size 分桶 capture），并且图里用到的 buffer 会被反复复用。因此"把 GPU 输出异步拷走"时**必须先 clone**，否则下一次 replay 会覆盖数据（见 §9）。

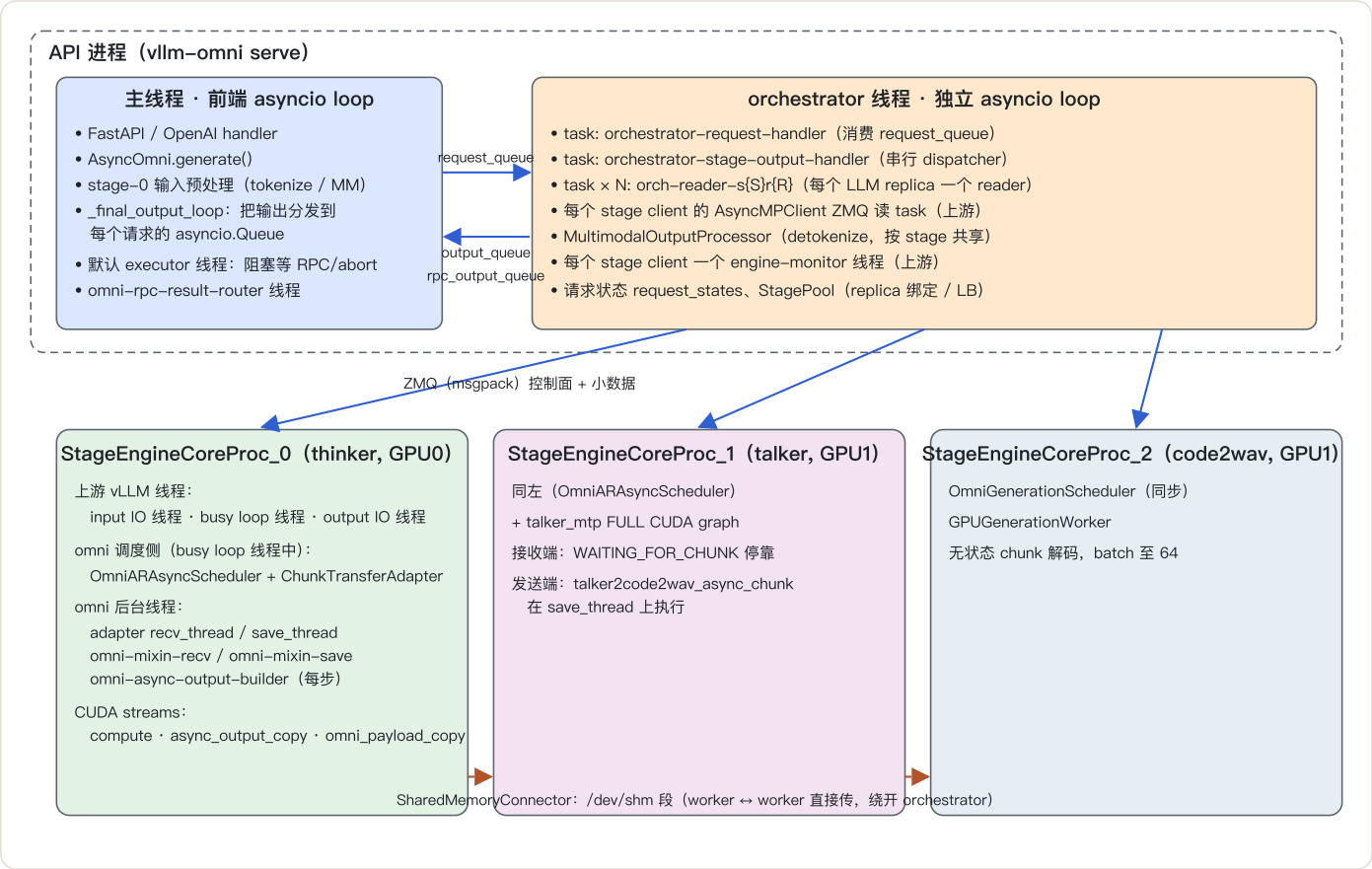
Python 并发概念 · 本文用到的原语

asyncio：单线程协作式并发，只在 `await` 处切换，因此同一个 loop 上的 task 之间访问共享 dict 不需要锁。**janus.Queue**：同时有 `sync_q`（线程侧）和 `async_q`（协程侧）两个接口的队列，用来桥接"普通线程"和"asyncio 循环"。**loop.call_soon_threadsafe**：从别的线程安全地唤醒一个 loop。**threading.Condition**：带等待和通知的锁。**fcntl.flock**：跨进程的文件锁。**POSIX shared memory**：`/dev/shm` 下的命名内存段，可跨进程映射。

---

3. 全景：进程 / 线程 / 事件循环

以默认 Qwen3-Omni 部署（2 GPU，TP=1）为例，一个运行中的服务包含 **1 个 API 进程 + 3 个 stage 子进程**：



执行体	职责	源码
前端 loop（主线程）	HTTP；`AsyncOmni.generate`；stage-0 输入处理在 `add_request` 中内联完成；`_final_output_loop`	`⟨/⟩ engine/async_omni_engine.py:518-599` `⟨/⟩ entrypoints/async_omni_base.py:316-419`
orchestrator 线程	daemon 线程，内部 `run_until_complete` 自己的 loop；负责 stage 初始化、`Orchestrator.run()`	`⟨/⟩ engine/omni_engine_base.py:323-332` `⟨/⟩ engine/omni_engine_base.py:440-509` `⟨/⟩ engine/orchestrator.py:483-487`

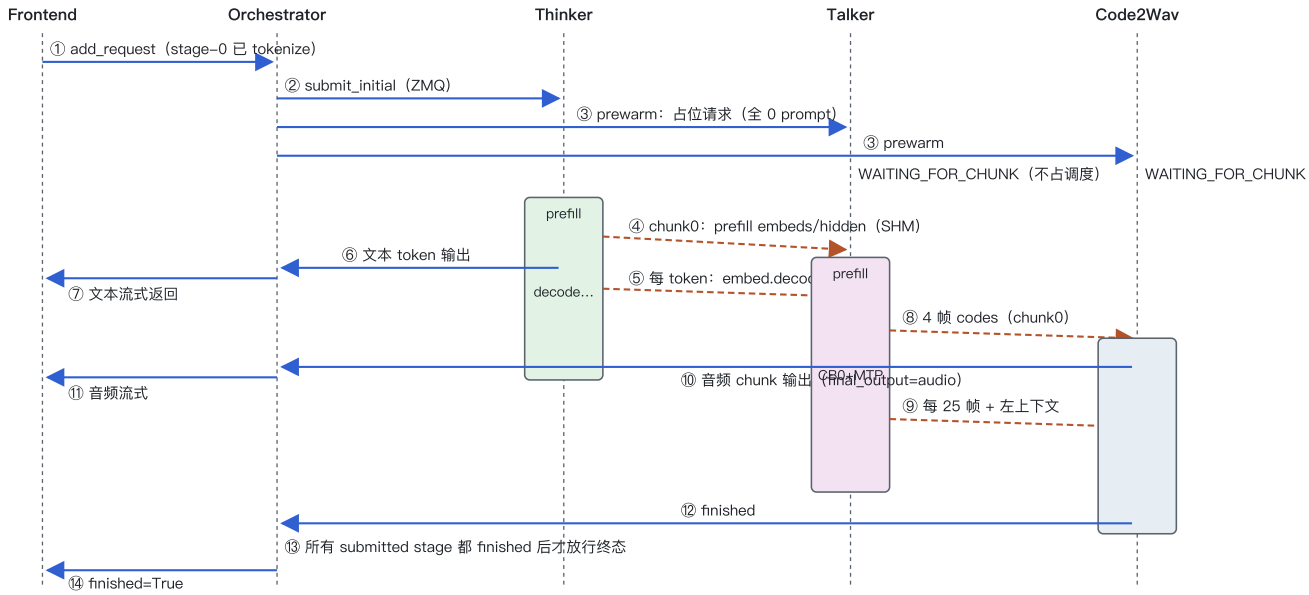
© 2026 kzhou003

RPC router 线程	把 `rpc_output_queue` 上的结果按 key 分发给等待者	`⟨/⟩ engine/rpc_result_router.py:32-49`
Stage 进程	`StageEngineCoreProc(EngineCoreProc)`，进程名 `StageEngineCoreProc_{stage}_replica{r}_DP{d}`	`⟨/⟩ engine/stage_engine_core_proc.py:133` `⟨/⟩ engine/stage_engine_core_proc.py:242` `⟨/⟩ engine/stage_engine_core_proc.py:324`

为什么 orchestrator 要单独开一个线程和 loop?

1. **隔离**：前端 loop 要处理 HTTP、SSE 流和 tokenize（CPU 密集），orchestrator 要以低延迟搬运每个 stage 的每一步输出。放在两个 loop 里，一方的长时间同步工作不会直接卡住另一方的 await 点。
2. **单写者**：所有跨 stage 的状态（`request_states`、replica 绑定、output processor 状态）只在 orchestrator loop 上被修改。由于 asyncio 是协作式的，只要不在临界区中间 `await`，就不需要锁。
3. **可替换**：同一套 Orchestrator 也能用于离线 `Omni`（同步）入口，前端只需要换成同步的队列接口。

4. 一个请求的一生（async_chunk 模式）



1. **前端**：在前端 loop 上完成 stage-0 输入预处理，然后 `request_queue.sync_q.put(msg)`（`⟨/⟩ engine/async_omni_engine.py:584`）。
2. **Orchestrator**：`_handle_add_request` 先把 `OrchestratorRequestState` 注册进 `request_states`，再 await `submit_initial`。这保证任何输出到达时请求都已经"已知"（`⟨/⟩ engine/orchestrator.py:3107-3128`）。
3. **Prewarm**：async_chunk 下，立刻给所有接收 chunk 的下游 stage 提交占位请求（`⟨/⟩ engine/orchestrator.py:2711`、`⟨/⟩ engine/orchestrator.py:2762-2870`）。这样第一个 chunk 到达时，下游请求已经在 scheduler 里等着了，省掉一次 orchestrator 往返。
4. ④⑤⑧⑨ **数据面**：chunk 由 worker 写入共享内存，下游 scheduler 的后台线程取走，**不经过 orchestrator**（§8）。
5. ⑥⑩ **控制面**：每个 stage 的输出经 ZMQ 回到 orchestrator，由 reader 放进 `ready_q`，再由串行 dispatcher 做 detokenize 和路由（§6）。

© 2026 kzhou003

6. ⑬ **终态守门**：即使 final stage 先报告 finished，只要还有已提交的 stage 没完成，就先把终态消息存进 `pending_final_output`（`⟨/⟩ engine/orchestrator.py:1954-1967`）。

---

5. 前端 ↔ Orchestrator：跨线程队列

三个队列都在 `OmniEngineBase.__init__` 中创建（`⟨/⟩ engine/omni_engine_base.py:308-311`）：

队列	方向	消息类型	容量
`request_queue` (janus)	前端 `sync_q` → orchestrator `async_q`	`StageSubmissionMessage`、`AbortRequestMessage`、`CollectiveRPCRequestMessage`、`ShutdownRequestMessage`、companion / interaction / 远端 replica 注册等。见 `⟨/⟩ engine/messages.py:29-91`	**256**，提供背压
`output_queue` (janus)	orchestrator → 前端	`OutputMessage`、`StageMetricsMessage`、`ErrorMessage`	无界
`rpc_output_queue` (janus)	orchestrator → router 线程	`CollectiveRPCResultMessage`、`AbortResultMessage`、fatal `ErrorMessage`	无界

5.1 输出侧的"批量唤醒"与防丢唤醒

orchestrator 每秒可能产生成千上万条输出（每个 token、每个音频 chunk）。如果每条都 `call_soon_threadsafe` 唤醒前端 loop，会产生大量跨线程唤醒开销。`OutputQueueSink` 的做法是：消息写进 `sync_q`，但**每批最多只调度一次唤醒**。

```
# async_engine_utils.py:60-70
def put_nowait(self, item):
    self._sync_q.put_nowait(item)
    if self._wake_scheduled:          # 已经有一次唤醒在路上了，不再重复
        return
    self._wake_scheduled = True
    loop, ready = self._loop, self._ready
    if loop is not None and ready is not None:
        try:
            loop.call_soon_threadsafe(ready.set)
```

`⟨/⟩ engine/async_engine_utils.py:40-97`。前端的 `get_outputs_async` 采用经典的"**drain → arm → drain again → wait**"模式：先取最多 256 条，然后重新武装唤醒标志，再取一次（关掉"在 drain 和 arm 之间到达的消息没人唤醒"这个 lost-wakeup 竞态），最后在 `asyncio.Event` 上等待，并设 1 s 超时做存活检查（`⟨/⟩ engine/omni_engine_base.py:973-991`）。

> **为什么输出队列不设上界**
>
> orchestrator 是热路径上的唯一搬运者。如果它因为前端消费慢而在 put 上阻塞，所有 stage 的输出都会停滞，并通过 ZMQ 反压到 GPU。设计选择是让输出永远不阻塞 orchestrator，背压只施加在**入口**（request_queue 的 256 上限）。

5.2 每个请求的输出队列

前端只有一个 `_final_output_loop` task 消费 `output_queue`，再按 request_id 把消息放进各请求自己的 `asyncio.Queue`（`⟨/⟩ entrypoints/async_omni_base.py:380`、`⟨/⟩ entrypoints/client_request_state.py:23`），`generate()` 的 async generator 从中 yield。未知 request 的迟到消息直接丢弃（`⟨/⟩ entrypoints/omni_base.py:519-526`）。

© 2026 kzhou003

6. Orchestrator 事件驱动循环

默认开启事件驱动模式（`VLLM_OMNI_EVENT_DRIVEN_ORCH` 未设置时为 True，`⟨/⟩ engine/orchestrator.py:99-103`）。结构是 **"N 个 reader + 1 个串行 dispatcher"**：

```
# orchestrator.py:1120-1136 —— 每个 LLM replica 一个 reader
while not self._shutdown_event.is_set():
    raw_outputs = await pool.read_llm_raw_output(client)
    if raw_outputs is None:
        await asyncio.sleep(0.001); continue
    await ready_q.put(("llm", stage_id, replica_id,
raw_outputs))
...
except BaseException as e:
    await ready_q.put(("error", stage_id, replica_id, e))
```

```
# orchestrator.py:1264-1271 —— 唯一的 dispatcher
while not self._shutdown_event.is_set():
    if pending_get is None:
        pending_get = asyncio.create_task(ready_q.get())
    done, _ = await asyncio.wait(
        {pending_get, shutdown_task, reconcile_task},
        timeout=max(0.0, next_reconcile - now),
        return_when=asyncio.FIRST_COMPLETED)
```

- reader 等待的是 `client.get_output_async()`（`⟨/⟩ engine/stage_pool.py:1177`），也就是上游 AsyncMPClient 的 asyncio 队列，而不是轮询 socket。没有输出时 reader 完全睡眠，零 CPU。
- 所有 reader 汇入**一个** `ready_q`（`⟨/⟩ engine/orchestrator.py:1112`）。dispatcher 串行处理。这就是"多生产者 / 单消费者"：读取可以并发，状态修改是串行的。
- replica 成员变化（远端 replica 上下线）时，`membership_listener` 通过 `call_soon_threadsafe` 触发 `_wake_reconcile`，增删 reader；否则每 0.5 s 兜底 reconcile 一次（`⟨/⟩ engine/orchestrator.py:1243-1286`）。
- diffusion stage 仍然是 0/1 ms 轮询（`⟨/⟩ engine/orchestrator.py:1138-1157`）。旧的轮询模式（`⟨/⟩ engine/orchestrator.py:1040-1097`）与事件驱动模式共享 `_process_llm_stage_outputs` 和 `_handle_processed_outputs`，因此语义完全一致。

> **为什么事件驱动比 1 ms 轮询好**
> 轮询模式下，每 1 ms 都要对每个 replica 执行 `asyncio.wait_for(read, 0.001)`（`⟨/⟩ engine/stage_pool.py:1252-1257`）。这样既浪费 CPU（还和前端 tokenize 抢 GIL），又给每一跳增加最多 1 ms 延迟。talker 每帧都有输出，3 个 stage 串起来，这部分延迟会累积到 TTFP 和 ITL 上。

6.1 请求状态机

`OrchestratorRequestState`（`⟨/⟩ engine/orchestrator.py:232-271`）的关键字段：

- `final_stage_id` / `final_output_stage_ids` / `finished_final_output_stage_ids`：哪些 stage 的输出要返回给用户（Qwen3-Omni 有两个：thinker 的文本、code2wav 的音频）。
- `stage_submit_ts: dict[int, float]`：**"某个 stage 是否已提交"的唯一判据**（`⟨/⟩ engine/orchestrator.py:2045-2046`），也被死 replica 扫描使用。
- `finished_stage_ids`、`pending_final_output`：终态守门（见下文）。
- `streaming: StreamingInputState`：流式输入（如 realtime）的分段状态。

转发规则（非 async_chunk，或下游不收 chunk 时），见 `⟨/⟩ engine/orchestrator.py:1988-1996`：

```
if ((finished or segment_finished)
        and stage_id < req_state.final_stage_id
        and not req_state.skip_legacy_stage_forward
        and (not self.async_chunk or not self._stage_receives_async_chunks(stage_id + 1))
        and (not self._next_stage_already_submitted(stage_id, req_state) or req_state.streaming.enabled)):
    ...  # → _forward_to_next_stage → process_engine_inputs → submit_initial / submit_update
```


© 2026 kzhou003

所有清理都走唯一出口 `_cleanup_request_ids`（`engine/orchestrator.py:1699-1762`）：弹出状态、释放所有 pool 上的 replica 绑定、排队释放资源（包括 SHM）。

6.2 顺序保证与终态单调性

每个请求的输出有序，由一条 FIFO 链保证：

1. 请求在每个 stage 上**粘性绑定**到一个 replica（`_request_bindings`，`engine/stage_pool.py:623`）。
2. 每个 replica 只有一个 reader，按 ZMQ 顺序读取。
3. 只有一个 dispatcher，`request_states` 不会被并发修改。
4. output_queue 是单一 FIFO，前端单 task 分发。

终态单调（一旦 finished，就不再有新输出，也不会"复活"）：

- 只有当**所有** final-output stage 都完成时才置 `finished=True`（`engine/orchestrator.py:1937-1940`）。
- async_chunk 下，如果 final 消息到达时上游 stage 尚未全部完成，先停放在 `pending_final_output`，等 `stage_submit_ts ⊆ finished_stage_ids` 时再放行（`engine/orchestrator.py:2036-2040`）。原因是数据面和控制面是两条路，完成信号可能乱序到达；如果过早发出终态，清理会把仍在写 SHM 的上游资源回收掉。
- 原始终态被吞掉时，会合成一个空的终态消息，防止 `generate()` 永远挂起（`engine/orchestrator.py:1858-1887`）。
- 清理后的迟到输出被丢弃（`engine/orchestrator.py:1385-1393`）；针对已不存在请求的 `streaming_update` 也被丢弃，不会复活（`engine/orchestrator.py:3150-3162`）。

```
# orchestrator.py:1954-1967 —— async_chunk 终态守门
if (request_finished and self.async_chunk
        and not req_state.streaming.enabled
        and not set(req_state.stage_submit_ts).issubset(req_state.finished_stage_ids)):
    req_state.pending_final_output = message
    request_finished = False
else:
    await self.output_async_queue.put(message)
```

> **Detokenize 放在哪**
> 每个 LLM stage 的 pool 有一个 `MultimodalOutputProcessor`（`outputs/output_processor.py:384`），由该 stage 所有 replica 共享，**运行在 orchestrator 线程上**。`submit_initial` 先 `add_request`，再发 ZMQ，发送失败时回滚（`engine/stage_pool.py:1053-1080`）。这对应上游 AsyncLLM 中 OutputProcessor 的角色，只是从前端 loop 挪到了 orchestrator loop。

---

7. Stage 进程内部

每个 stage 是一个 `StageEngineCoreProc`，继承上游 `EngineCoreProc`。**vllm-omni 没有重写** `step`、`step_with_batch_queue` **或** `run_busy_loop`，主循环完全沿用上游。omni 的扩展都通过"插入点"完成：

扩展点	做什么 / 为什么	源码
请求类型替换	在 `__init__` 之前把模块全局 `EngineCoreRequest` 换成 `OmniEngineCoreRequest`。原因是上游输入 IO 线程会在 `__init__` 里创建 `MsgpackDecoder(EngineCoreRequest)`，换晚了就无法解码 `additional_information`。	`engine/stage_engine_core_proc.py:255-258`

© 2026 kzhou003

payload 字段透传	`preprocess_add_request` 把 `additional_information`、`external_req_id`、`payload_sender_info` 拷到 scheduler 的 `Request` 上	⟨/⟩ engine/stage_engine_core_proc.py:171–177
abort 栅栏	ABORT 经过 FIFO 分发之后才调用 `release_overflowed_requests`。上游的"急切 abort 队列"可能先于之前的 ADD 执行，只有 FIFO 位置才能保证顺序，并允许 ID 复用。	⟨/⟩ engine/stage_engine_core_proc.py:147–154
信号唤醒	SIGTERM 时设置 `shutdown_state`，并往 `input_queue` 塞一个 `WAKEUP`，把阻塞在队列上的 busy loop 唤醒	⟨/⟩ engine/stage_engine_core_proc.py:311–322
同进程 sink	first-audio sink 直接写上游 `output_queue`，绕过 scheduler；native data plane 的就绪 sink 直接写 scheduler 收件箱（仅限 UniProc、TP=1、PP=1，因为只有这时 runner 和 scheduler 在同一进程）	⟨/⟩ engine/stage_engine_core_proc.py:84–119
SHM 回收	UTILITY 通道上的 `omni_release_request_resources` 只负责入队，真正的 unlink 在 adapter 的 save 线程上执行	⟨/⟩ engine/stage_engine_core_proc.py:156–169

调度器选择：AR stage（thinker/talker）使用 `OmniARAsyncScheduler(OmniARScheduler, AsyncVLLMScheduler)`（⟨/⟩ core/sched/omni_ar_scheduler.py:1753），code2wav 使用同步的 `OmniGenerationScheduler`（⟨/⟩ config/stage_config.py:201–218）。

7.1 Scheduler 钩子与 WAITING_FOR_CHUNK

omni 给 `RequestStatus` 加了两个状态（⟨/⟩ patch.py:364–372）：`WAITING_FOR_CHUNK = -1` 和 `WAITING_FOR_INPUT = -2`。取负值是为了在上游已有的 `status > X` 比较中仍被视为"未结束"。

每次 `schedule()` 都用前后钩子把上游调度包起来（⟨/⟩ core/sched/omni_ar_scheduler.py:408–451）：

```
def schedule(self, throttle_prefills=False):
    self._drop_aborted_queued_requests()          # 上游遇到 ABORTED 状态会抛异常，先清掉
    self._process_pending_omni_inputs(model_mode="ar")  # ① 停靠：没数据的请求移出队列
    ...
    try:
        scheduler_output = super().schedule(throttle_prefills)  # ② 上游调度：只看得到"有数据"的请求
    finally:
        ...
        self._restore_omni_wait_queues()           # ③ 归还：数据到了的请求放回 waiting/running
    self._postprocess_omni_schedule_output(...)    # ④ 附上 additional_information
```

关键在于 `process_pending_chunks`（⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1120–1186）：对于还没有 chunk 的请求，它调用 `load_async` 让后台线程去取，并把请求**从 waiting/running 中物理移除**，放进私有 deque（⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1431–1463）。

© 2026 kzhou003

为什么要"移出队列"而不是"跳过"

- 上游 scheduler 不知道 omni 的状态。如果把没数据的请求留在队列里，它会被调度、占用 token budget 和 KV，而 forward 时根本没有输入。移出之后，上游 `super().schedule()` 看到的永远是"数据已就绪"的请求，上游语义不需要修改（对应 AR-INV-001：vLLM 拥有基础调度语义）。
- 在 `finally` 中归还，保证上游调度即使抛异常也不会丢请求。归还前会再清理一次 abort，关掉"abort 恰好落在 process 和 restore 之间"的竞态（#3736，`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1488-1532`）。
- recv 线程收到 chunk 后**不直接修改请求的 prompt**，而是在 scheduler 线程上 finalize（`_apply_pending_ar_prompt_updates`，`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1281-1330`）。只有在这里，请求的 token 计数器才是一致的快照。

`active_stream_window`：K = min(window, max_num_seqs)（`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:162-166`）。非活跃的 running 请求会被放到 `_held_non_active`，而不是调用 `waiting_queue.prepend_requests`，因为后者在 step 中途调用会触发上游 LogitsProcessor 的不变量断言（`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:210-216`）。

与 async scheduling 的交互

发送端的"已确认 token 数"= `num_computed_tokens - num_output_placeholders`（`distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:244-251`）。也就是说，**还没真正采样出来的 placeholder token 永远不会被当作 chunk 发给下游**。这是异步调度与流式传输组合时保证正确性的关键一行。

---

8. async_chunk 数据面

分层：**Connector** 只负责传输（`put/get`，不知道请求语义）；**TransferAdapter** 管理 chunk 生命周期（key、顺序、停靠/归还、清理）；**stage input processor**（模型相关的函数）决定发什么。

8.1 OmniChunkTransferAdapter：两条后台线程

基类（`distributed/omni_connectors/transfer_adapter/base.py:20-46`）启动 `recv_thread` 和 `save_thread`，各配一个 `deque` 和一个 `threading.Condition`：

```
# transfer_adapter/base.py:59-82 —— recv_loop
while not self.stop_event.is_set():
    n = len(self._pending_load_reqs)
    for _ in range(n):
        request = self._pending_load_reqs.popleft()
        is_success = self._poll_single_request(request)   # 非阻塞 connector.get
        if not is_success:
            self._pending_load_reqs.append(request)       # 没到就放回队尾，轮转
    with self._recv_cond:
        if not self._pending_load_reqs and not self.stop_event.is_set():
            self._recv_cond.wait(timeout=0.1)             # 空闲：等通知
        elif not any_success and not self.stop_event.is_set():
            self._recv_cond.wait(timeout=0.001)           # 全部未就绪：1 ms 退避，避免 shm_open 空转
```

入队（append）时**不持有** condition 锁，因此可能出现"append 后 notify 先于 wait"的竞态。条件变量的 timeout 就是这个竞态的兜底（`distributed/omni_connectors/transfer_adapter/base.py:77-82`）。这是一个典型的取舍：用很小的最坏延迟（≤0.1

© 2026 kzhou003

s，有工作时 ≤1 ms）换取热路径上的无锁入队。

发送路径（谁在哪个线程做什么）

1. **GPU forward 线程**：runner 把 `inter_stage_outputs` 放进 `ModelRunnerOutput`。
2. **EngineCore 线程**：`OmniARScheduler.update_from_output` 调用 `save_async(...)`（⟨/⟩ core/sched/omni_ar_scheduler.py:880-901）。`save_async` 只做**快照 + 入队**：浅拷贝 request、`prompt_token_ids`、`additional_information`、token 列表（⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:254-292），因为 scheduler 接下来会继续修改实时的 request。
3. **save 线程**：执行模型的 `custom_process_next_stage_input_func`（如 `talker2code2wav_async_chunk`），然后 `connector.put`（⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:778-854）。Python 层的 payload 整形和 SHM 写入完全不在调度关键路径上。

> **注意 · 与旧文档的出入**
>
> `docs/design/feature/async_chunk.md` 写的是"custom func 在主线程构建 payload"。在当前 HEAD 上，V1 adapter 的路径是在 save 线程上调用 processor（主线程只做快照）。读源码时以代码为准。

锁与 key

- `_sender_state_lock`：保护 `_SenderGeneration`（`in_flight`、`cancelled`、`terminal_pending`、`cleanup_deferred`，⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:34-51）。只用于短状态转换，构建 payload 和 put 都不在锁内。
- `_receiver_state_lock`：把"注册/清理"与"提交收到的 chunk"串行化。`connector.get()` 在锁外执行，拿到数据后在锁内**重新检查** `self._registered_load_entries.get(req_id) is entry`（⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:539-555）。这是一种 generation check：如果在 get 期间请求已被清理或重新注册，这个迟到的 chunk 就被丢弃。清理因此成为一道"提交栅栏"。
- Key 规则：`{external_req_id}_{stage_id}_{chunk_id}`（发送 ⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:792，接收 ⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:516）。chunk 序号只在 put 成功后才在锁内递增（⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:856-874）。确定性的 key 让双方无需额外的元数据通道就能对上号，也天然保证了 chunk 顺序：接收方只取下一个序号。

终止与清理顺序

- `cleanup_sender` 从不等待 IO：如果 terminal chunk 还在队列里，就标记 `cancelled + cleanup_deferred`；如果 put 正在进行，就交给发送方的 `finally` 处理（⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:991-1027）。
- `cancelled` **不会**阻止 terminal chunk，因为它是下游唯一的 end-of-stream 信号。阻止它会让下游请求永远停在 WAITING_FOR_CHUNK（#6670，⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:714-723）。
- abort 的 SHM unlink 以 `{"cleanup_shm": ...}` 任务排在**同一个 save 线程**上（⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1029-1042）。同一线程 FIFO 保证 unlink 一定发生在之前的 put 之后，而且 scheduler 自己不做任何 SHM IO。

© 2026 kzhou003

8.2 SharedMemoryConnector

步骤	机制	源码
写	序列化 payload，对 `/dev/shm/shm_{key}_lockfile.lock` 加 `fcntl` 排他锁，创建名字与 key 相同的 POSIX 共享内存段（已存在则 unlink 后重建）	⟨/⟩ distributed/omni_connectors/connectors/shm_connector.py:195-228 ⟨/⟩ entrypoints/stage_utils.py:193-215
读	以 `LOCK_EX|LOCK_NB` 尝试加锁：`BlockingIOError` 表示写者还没写完，视为"未就绪"；`ValueError("empty file")` 表示段还没 sizing，同样视为未就绪	⟨/⟩ distributed/omni_connectors/connectors/shm_connector.py:234-274
释放	**读者负责 unlink**：`shm_read_bytes` 拷出字节后立即 unlink 段，再删 lockfile。生产者用 `reap_consumed` 轮转检查自己的 pending key 是否已被消费，以限制 bookkeeping 大小	⟨/⟩ entrypoints/stage_utils.py:223-237 ⟨/⟩ distributed/omni_connectors/connectors/shm_connector.py:358-364
唤醒	每个接收者有一个命名 FIFO；put 时向目标 stage 的所有 FIFO 各写 1 字节，接收者 `select()` 等待。唤醒只是提示，定时轮询才是正确性兜底	⟨/⟩ distributed/omni_connectors/connectors/shm_connector.py:81-174
兜底清理	未消费的段由 `cleanup`、`cleanup_prefix`、`close` 回收；orchestrator 在所有 stage 完成后驱动 `release_shm_resources`	⟨/⟩ distributed/omni_connectors/connectors/shm_connector.py:312-356 ⟨/⟩ distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1060-1078

> **为什么用"文件锁 + 读者 unlink"**
> 共享内存段本身没有"写完了"的原子信号。用同名 lockfile 的排他锁做"写入中"标记，读者用非阻塞加锁来探测，就得到了一个跨进程、无需额外通道的 ready 信号。读者 unlink 让内存在消费后立刻释放（单消费者语义），生产者不必等待 ACK。

其他 connector（Mooncake RDMA、NIXL、Mori、Yuanrong 等，`vllm_omni/distributed/omni_connectors/connectors/`）实现同一个 put/get 接口，用于跨节点或 GPU-direct 场景。没有配置 connector 的边会自动补上 SharedMemoryConnector（⟨/⟩ distributed/omni_connectors/utils/initialization.py:391-421）。

8.3 Runner 侧的 connector runtime

非 async_chunk 的 full-payload 模式，以及 MRv2，走的是 model runner 内的 `omni-mixin-recv` / `omni-mixin-save` 线程（⟨/⟩ distributed/omni_connectors/model_runner/omni_connector_runtime.py:317-341）：

- 所有传输状态用一把 `_lock`，另有**两个独立的** event：`_work_available` 和 `_save_work_available`。注释说明，曾经共用一个 event 时，接收方会"吞掉"发送方的唤醒。
- 接收循环在**轮询之前**先快照 wakeup generation，再 `wait_for_change(generation, timeout)`。这样在轮询期间到达的数据不会因为唤醒已被消费而丢失（⟨/⟩

© 2026 kzhou003

distributed/omni_connectors/model_runner/omni_connector_payload_transport.py:959-995）。这是另一个经典的防丢唤醒写法。

- 收到的 payload 先进 `_local_stage_payload_cache`，之后在模型线程上 materialize 成运行时状态（⟨/⟩ distributed/omni_connectors/model_runner/omni_connector_runtime.py:281-289）。V1 在 `execute_model` 开头同步调用 recv/flush（⟨/⟩ worker/gpu_ar_model_runner.py:963-971）。

---

9. GPU 层并发：CUDA stream / event / graph

Thinker 每步都要把 hidden states 拷回 CPU 发给 talker。如果在计算流上同步拷贝，GPU 会被打断。vllm-omni 在 V1 AR runner 中实现了**异步输出物化**（⟨/⟩ worker/gpu_ar_model_runner.py:1670-1796）：

```
# gpu_ar_model_runner.py:180-189
source_stream = torch.cuda.current_stream()
ready_event = torch.cuda.Event()
with torch.cuda.stream(copy_stream):                # 独立的 _omni_payload_copy_stream
    copy_stream.wait_stream(source_stream)          # 只等到"数据算完"这一刻
    cpu_payload = _copy_tensor_payload_to_cpu(cloned, pin_memory)   # 拷 clone（graph buffer 会复用）
    for source in cuda_sources:
        source.record_stream(copy_stream)           # 告诉 caching allocator：拷完之前别复用这块显存
    ready_event.record(copy_stream)                 # 完成信号
```

- **先 clone**：CUDA graph 或模型 buffer 在下一步 replay 时会被覆盖。
- **pinned host memory**：pageable 内存会让 D2H 退化成同步拷贝。注释记录了约 240 ms/step 的教训（⟨/⟩ worker/gpu_ar_model_runner.py:1730-1739）。
- `record_stream`：PyTorch caching allocator 只跟踪分配时所在的流；跨流使用时必须登记，否则内存可能在拷贝完成前被回收复用。
- **后台 builder 线程** `omni-async-output-builder` 等待 `ready_event` 后组装 payload；`get_output()` 时 join 并重新抛出 builder 的异常（⟨/⟩ worker/gpu_ar_model_runner.py:200-297）。scheduler output、token 列表等先做快照（⟨/⟩ worker/gpu_ar_model_runner.py:2200-2207）。
- **例外**：connector-output 的 drain 留在主线程，因为它可能执行 TP collective（集合通信必须在所有 rank 上以相同顺序发生，⟨/⟩ worker/gpu_ar_model_runner.py:2228-2231）。
- 启用条件：开了 async scheduling、没有 spec decode、开了 async_chunk、模型声明 `use_async_omni_output`（⟨/⟩ worker/gpu_ar_model_runner.py:1670-1692）。

采样出的 token 交给 input batch 时附带 ready event（上游机制）。omni 在修补 `-1` placeholder 之前会先同步这个 event（⟨/⟩ worker/gpu_model_runner.py:357-361）。Generation runner（code2wav）也有类似的异步快照；只有当模型自己拥有存储且是 eager 模式时，才跳过 clone（⟨/⟩ worker/gpu_generation_model_runner.py:74-140）。

> **一个 stage 进程里的三条 CUDA 流**
> compute stream（forward + 采样）、上游的 `async_output_copy_stream`（采样 token 的 D2H）、omni 的 `_omni_payload_copy_stream`（跨 stage payload 的 D2H）。三者通过 event 和 `wait_stream` 建立最小依赖，因此下一步的 forward 可以与上一步的两种拷贝重叠。

© 2026 kzhou003

10. 正确性：abort / 故障 / 关停 / RPC

10.1 Abort 传播（客户端断开）

1. `generate` 捕获 `CancelledError` / `GeneratorExit`，以 `asyncio.shield` 执行 `_abort_internal_requests`，并受 `ABORT_TIMEOUT_S` 约束（`⟨/⟩ entrypoints/async_omni.py:325-332`）。shield 保证 abort 本身不会因为外层被取消而半途中断。
2. `abort_async` 生成 `rpc_id`，在 executor 线程上 `CorrelatedRpcClient.execute(("abort", rpc_id))`（`⟨/⟩ engine/omni_engine_base.py:1037-1054`）。
3. Orchestrator 的 `_handle_abort` 调用 `_cleanup_request_ids(abort=True)`：展开 CFG companion，让孤儿 parent 失败，然后对每个 pool 执行 `abort_requests`（`⟨/⟩ engine/orchestrator.py:645-731`）。
4. 每个 pool 内按绑定的 replica 分组：**先**从 output processor 收集部分输出（`commit_state=False`），**再**调用 EngineCore 的 abort，**成功后**才提交 processor 的 abort 状态（`⟨/⟩ engine/stage_pool.py:1279-1339`）。这种两阶段提交避免了"processor 以为已 abort、引擎却没收到"的不一致。
5. 多个 final stage 时，只有最后一条 abort 输出带 `finished`（`⟨/⟩ engine/orchestrator.py:726-730`）。
6. 前端收到 abort 输出；如果什么都没收到，就合成一个 `finish_reason="abort"` 的终态，保证 `generate` 不会挂起（`⟨/⟩ entrypoints/async_omni_base.py:460-493`）。

10.2 故障隔离

- **某个 stage 进程死亡**：上游 monitor 线程标记 dead，`get_output_async` 抛出 `EngineDeadError`，reader 把 `("error", …)` 放进 `ready_q`，dispatcher 执行 `_handle_dead_replica`，evict 该 replica（`⟨/⟩ engine/orchestrator.py:1464-1515`，`⟨/⟩ engine/stage_pool.py:1479-1496`）。
- **只失败受影响的请求**：绑定在死 replica 上的请求；如果该 stage 已没有存活 replica，则是所有经过该 stage 的请求。每个请求收到一个 request 级的 `ErrorMessage(fatal=True, request_id=…)`，服务本身继续运行。

```
# orchestrator.py:1485-1497
for req_id, req_state in list(self.request_states.items()):
    if stage_id not in req_state.stage_submit_ts: continue
    bound = pool.get_bound_replica_id(req_id)
    if bound == replica_id or not stage_has_live:
        await self.output_async_queue.put(ErrorMessage(
            error=str(error), fatal=True, request_id=req_id, stage_id=stage_id))
```

- **Orchestrator 线程本身崩溃**：向 output sink 和 RPC 队列各放一个不带 request_id 的 fatal `ErrorMessage`（`⟨/⟩ engine/omni_engine_base.py:486-491`）。前端会把错误广播给所有请求队列，router 广播给所有 RPC 等待者并锁存错误（`⟨/⟩ engine/rpc_result_router.py:75-110`）。

10.3 关停顺序

1. 前端：取消 `final_output_task`，然后调用 `engine.shutdown()`（`⟨/⟩ entrypoints/async_omni_base.py:570-573`）。
2. `OmniEngineBase.shutdown`（`⟨/⟩ engine/omni_engine_base.py:1132-1220`）：发 `ShutdownRequestMessage`（1 s 超时）→ 关闭 RPC router → join orchestrator 线程（最多 30 s）→ 关闭 janus 队列 → `runtime.shutdown()` → 释放 CuMem 池。
3. Orchestrator 内部：先设置 `_shutdown_event`，并预先把 client 标记为 `_shutting_down`，避免 monitor 把正常退出误报成 dead（`⟨/⟩ engine/orchestrator.py:576-588`）；`run()` 的 finally 取消 task，再逐个 `shutdown_replica`（`⟨/⟩ engine/orchestrator.py:530-553`）。

© 2026 kzhou003

顺序原则：**先停 orchestrator，再停 runtime**，确保关停过程中没有人还在往已关闭的 client 发消息。

10.4 RPC 关联

```
# rpc_result_router.py:136-150 —— 同步调用方（executor 线程）
waiter = self._router.register(key)          # 每个 key 一个 queue.Queue(maxsize=1)
try:
    if block_on_submit: self._request_queue.put(message)
    else:               self._request_queue.put_nowait(message)
    result = waiter.get(timeout=timeout)
    if isinstance(result, ErrorMessage): raise RuntimeError(result.error)
    return result
finally:
    self._router.unregister(key, waiter)     # 迟到的结果会被丢弃
```

key 是 `("collective", rpc_id)` 或 `("abort", rpc_id)`（`⟨/⟩ engine/messages.py:125-148`）。**先注册再发送**，所以结果不可能在注册之前到达。`pause_generation`、`sleep`、`wake_up` 先用 `asyncio.Condition`（`_pause_cond`）闸住新请求的准入，并等待 `_admitting == 0`，再发 RPC（`⟨/⟩ entrypoints/async_omni.py:972-995`）。由于 request_queue 是 FIFO，之前已入队的 add 一定先于 sleep 被处理。

---

11. 启动期并发

多个 stage 共用 GPU 时，并发初始化最大的风险是**显存 profiling 互相干扰**：vLLM 通过"测量空闲显存"来决定 KV cache 大小，如果另一个 stage 恰好在同时加载权重，测量就会出错，导致 OOM 或 KV 过小。

- **按设备分组**：`_init_group_keys` 按"设备重叠的连通分量"分组（`⟨/⟩ engine/stage_runtime.py:1093-1134`，`⟨/⟩ engine/stage_init_utils.py:1738-1762`）。默认 Qwen3-Omni 部署得到两组：`device-group:0`（thinker）和 `device-group:1`（talker 然后 code2wav，组内串行）。两组在 `stage-init` 线程池上并行启动（`⟨/⟩ engine/stage_runtime.py:879-956`）。第一个失败记录在 `init_state_lock` 下，并让后续组短路退出。
- **串行模式**：父进程对每个设备取排他 `flock`（`⟨/⟩ engine/stage_init_utils.py:1765-1900`），握手完成后释放；spawn 时的环境变量（`CUDA_VISIBLE_DEVICES`）修改用 `_replica_launch_lock` 串行化，因为环境变量是进程全局状态。
- **并行模式**（`parallel_stage_init`）：先用 `check_admission` 做每设备的显存账本可行性检查（`⟨/⟩ engine/stage_admission.py:205`）。子进程内用 `DevicePhaseLock`（按设备排序、fail-closed 的共享/排他 flock）：加载权重、分配 KV、capture graph 持共享锁；`determine_available_memory` 持排他锁（`⟨/⟩ engine/stage_phase_lock.py:106-238`）。这本质上是一个跨进程的读写锁：多个 stage 可以同时修改显存，但测量时必须独占。

---

12. MRv2 原生数据面（简述）

当 deploy 设置 `model_runner: v2`，且 stage 声明了 `supports_native_mrv2_data_plane`（Qwen3-Omni 三个 stage 都声明了，但默认 yaml 没开），scheduler 不再创建 ChunkTransferAdapter，改由 runner 内的 `OmniRunnerDataPlane` 负责传输（`⟨/⟩ core/sched/omni_scheduler_mixin.py:204-222`）：

- 有界 `Queue(maxsize=8)` 加单个有序 output worker（`⟨/⟩ worker_v2/omni_data_plane.py:106-206`）。
- `_native_send_lock` 在释放状态锁**之前**获取（hand-over-hand），使 abort 无法插到 output 前面（`⟨/⟩ worker_v2/omni_data_plane.py:362-382`）。

© 2026 kzhou003

- terminal marker 等 in-flight 计数归零后才发出；`DeliveryTicket` 用 event 实现可等待的投递确认（`worker_v2/delivery.py:34-74`）。
- `FirstAudioSender`：首个音频帧做 pinned D2H 并记录 event，后台线程直接 `put_nowait` 到 EngineCore 的 `output_queue`，同时绕过 scheduler 和 busy loop，进一步压缩 TTFP（`worker_v2/first_audio_sender.py:79-154`）。

13. 设计取舍与已知风险

点	说明
前端 loop 上的阻塞 put	`add_request` 对 `request_queue.sync_q` 做阻塞 put（`engine/async_omni_engine.py:584`）。队列满 256 时会**阻塞整个前端事件循环**，HTTP 和流式输出都会停顿。
RPC 阻塞准入	`_handle_collective_rpc` 在 request-handler task 中内联执行（`engine/orchestrator.py:753-817`）。一个耗时的 sleep/wake RPC 会让排在它后面的 add/abort 全部等待。
共享状态依赖协作式调度	`request_states` 被 request-handler 和 dispatcher 两个 task 共享，安全性完全依赖"同一个 loop 且临界区内没有 await"。在这些路径中插入 await 时要格外小心。
Condition 超时兜底	adapter 的无锁 append 加 notify，最坏情况下依赖 0.1 s 或 1 ms 的 timeout 才能被唤醒。这是有意的延迟和吞吐取舍。
orchestrator 的线性转发	转发目标固定是 `stage_id + 1`（`engine/orchestrator.py:2341`、`engine/orchestrator.py:2357`）。`input_sources` 只用于 connector 边和默认输入处理。stage graph 在 config 上像 DAG，但在运行时是一条链。
文档与代码的出入	async_chunk.md 对 payload 构建线程的描述，以及 output materialization 文档对 drain 位置的描述，与 HEAD 代码不一致（见 §8、§9）。另外 Qwen3-Omni 的 load-based 动态初始 chunk（`compute_dynamic_initial_chunk_size`）实际只在 qwen3_tts 中使用，Qwen3-Omni 用的是固定 4 帧或可选 ramp。

14. 推荐源码阅读路线

1. **拓扑与配置**：`model_executor/models/qwen3_omni/pipeline.py:22-84` → `deploy/qwen3_omni_moe.yaml:15-77` → `config/stage_config.py:1256-1320`（`merge_pipeline_deploy`）
2. **线程与队列**：`engine/omni_engine_base.py:300-340` → `engine/async_engine_utils.py:40-97` → `engine/omni_engine_base.py:973-991`
3. **Orchestrator 主循环**：`engine/orchestrator.py:1099-1320` → `engine/orchestrator.py:1912-2046`（`_route_output`）→ `engine/orchestrator.py:2711-2870`（prewarm）→ `engine/orchestrator.py:3062-3162`（add request）
4. **Stage 进程**：`engine/stage_engine_core_proc.py:133-345` → `core/sched/omni_ar_scheduler.py:408-451` → `core/sched/omni_ar_scheduler.py:880-901`
5. **Chunk 数据面**：`distributed/omni_connectors/transfer_adapter/base.py:20-140` → `distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:381-503`（save_async）→ `distributed/omni_connectors/transfer_adapter/chunk_transfer_adapter.py:1120-1186`（process_pending_chunks）→ `distributed/omni_connectors/connectors/shm_connector.py:195-310`
6. **模型侧 processor**：`model_executor/stage_input_processors/qwen3_omni.py:409-542`（thinker→talker）→ `model_executor/stage_input_processors/qwen3_omni.py:765-842`（talker→code2wav）

© 2026 kzhou003

7. **GPU 重叠**：`worker/gpu_ar_model_runner.py:160-297` → `worker/gpu_ar_model_runner.py:1670-1796`
8. **正确性**：`engine/orchestrator.py:645-731`（abort）→ `engine/stage_pool.py:1279-1339` → `engine/orchestrator.py:1464-1515`（死 replica）→ `engine/omni_engine_base.py:1132-1220`（shutdown）
9. **上游对照**：`vLLM ↗ core.py (EngineCoreProc)` `vLLM ↗ core_client.py (AsyncMPClient)` `vLLM ↗ async_scheduler.py` `vLLM ↗ output_processor.py`

相关设计文档：`docs/design/feature/async_chunk.md` `docs/design/module/engine_orchestration.md` `docs/design/module/stage_runtime.md` `docs/design/feature/omni_async_output_materialization.md`

© 2026 kzhou003