下载离线版

离线阅读 vllm 学习笔记 的全部内容。

PDF

保留原始排版、代码高亮和流程图，适合在电脑或平板上阅读。

下载 PDF

EPUB

适合在电子书阅读器（Kindle、Apple Books、Calibre 等）上阅读。

下载 EPUB

> 将 prompt 的 prefill 计算拆分为多个 chunk 分步执行，避免长 prompt 独占 GPU 导致其他序列停滞。

为什么需要 **Chunked Prefill**

无 chunking 时，一个长 prompt 的 prefill 会独占整个 iteration，期间所有 decode 请求被阻塞，造成延迟飙升（TTFT 增大）。Chunked Prefill 将长 prompt 分成固定大小的 chunk，每次 iteration 只处理一个 chunk 并穿插 decode step，使新生成的 token 延迟不受 prompt 长度影响。

核心原理

- **Chunk** 大小：由 `max_num_batched_tokens` 控制，每个 iteration 最多处理这么多 token 的 prefill。
- 混合调度：一个 iteration 可同时包含 prefill chunk 和 decode token，兼顾首 token 延迟与吞吐。
- **KV Cache** 增量写入：每个 chunk 的 KV 结果增量写入对应 sequence 的 KV Cache block。
- **Attention Mask**：chunk 内部使用 causal mask，跨 chunk 的 attention 通过已缓存的 KV 实现。

在源码中的实现

- `vllm/core/scheduler.py` — Scheduler 在 `schedule()` 中将长 prompt 拆分为 chunk，每次调度一个 chunk。
- `vllm/worker/model_runner.py` — `ModelRunner` 在 prepare inputs 时区分 prefill chunk 和 decode token。
- `vllm/config.py` — `SchedulerConfig` 的 `chunked_prefill_enabled` 和 `max_num_batched_tokens` 参数。
- `vllm/attention/backends/` — 各注意力后端需处理跨 chunk 的 partial attention。

相关概念

- continuous-batching — Chunked Prefill 是 continuous batching 调度策略的关键优化
- kv-cache — 每个 chunk 的结果增量写入 KV Cache
- paged-attention — 增量分配 block 以支持分块 prefill
- prefix-caching — 缓存已 prefill 的前缀以避免重复计算

> 在每次推理迭代中动态插入新请求、移除已完成请求的批处理策略，也称 iteration-level batching。

为什么需要 **Continuous Batching**

静态批处理（Static Batching）要求同一 batch 内所有序列同时开始、同时结束。短序列完成后必须等待最长序列，导致 GPU 大量空闲。Continuous Batching 在每次迭代（iteration）粒度上调整 batch 组成，新请求无需等待当前 batch 完成，完成的请求立即释放资源，显著提升吞吐量。

核心原理

- **Iteration-level** 调度：每次 forward 前由 Scheduler 决定本次 batch 包含哪些 sequence。
- **Preemption & Swap**：显存不足时可抢占（preempt）低优先级序列，将其 KV Cache swap 到 CPU。
- **Slot** 复用：已完成的 sequence 释放的 slot 立即分配给等待中的新请求。
- 与 **PagedAttention** 协同：block 级别的 KV Cache 管理使 sequence 的加入和移除几乎零开销。

在源码中的实现

- `vllm/core/scheduler.py` — `Scheduler` 核心调度循环，每次 `schedule()` 返回当前 iteration 的 `SchedulerOutput`。
- `vllm/core/scheduling_policy.py` — 调度策略（FCFS、优先级等）决定请求的调度顺序。
- `vllm/engine/engine_core.py` — `EngineCore` 在主循环中调用 scheduler.step() 驱动 continuous batching。
- `vllm/worker/worker.py` — Worker 接收调度结果，执行模型 forward。

相关概念

- paged-attention — 使序列动态加入/移除成为可能
- kv-cache — 调度器管理的核心资源
- chunked-prefill — 控制 prefill 阶段对 batch 资源的占用
- speculative-decoding — 与 continuous batching 结合提升解码效率

> 将多个 CUDA kernel 的启动和执行录制为一张静态图，通过一次性重放消除 kernel launch 开销。

为什么需要 **CUDA Graph**

LLM 推理的 decode 阶段每次 forward 计算量很小，但 CPU 向 GPU 提交 kernel 的 launch 开销（约 5-10us/kernel）相对显著。一个 Transformer 层涉及数十个 kernel，累积的 launch 延迟可能占总延迟的 30% 以上。CUDA Graph 将整张计算图录制下来，每次 decode 只需重放一次 graph，消除所有 kernel launch 开销。

核心原理

- **Capture**：在推理开始前，用 dummy 输入执行一次 forward，录制所有 kernel 为 CUDA Graph。
- **Replay**：实际推理时将输入数据拷贝到预留 buffer 并 replay graph，无需 CPU 逐个 launch kernel。
- 静态形状限制：Graph 中所有张量形状必须固定，因此通常只为 decode 阶段（固定 batch）捕获 graph。
- 多 **graph** 管理：为不同 batch size 捕获不同 graph，运行时选择最接近的 graph 并 padding。

在源码中的实现

- `vllm/worker/model_runner.py` — `CUDAGraphRunner` 负责 capture 和 replay，管理多个 batch size 的 graph。
- `vllm/compiler/` — 编译器辅助静态化模型以支持 graph capture。
- `vllm/config.py` — `CompilationConfig` 和 `enforce_eager` 开关控制是否启用 CUDA Graph。
- `vllm/worker/worker.py` — Worker 初始化时调用 `warmup_model` 触发 graph capture。

相关概念

- torch-compile — torch.compile 的优化与 CUDA Graph 可叠加使用
- continuous-batching — Batch 大小动态变化对 graph capture 提出挑战
- tensor-parallelism — TP 场景下 CUDA Graph 需处理跨 GPU 通信
- speculative-decoding — Draft model 的小 forward 也可用 CUDA Graph 加速

> 通过 tiling 和 kernel fusion 减少 HBM 访问次数的 IO 感知注意力算法，是 vLLM 注意力计算的核心后端。

为什么需要 **FlashAttention**

标准注意力实现需将完整的 S = QK^T 和 P = softmax(S)V 写入 HBM，内存访问量为 O(N^2 d)，成为计算瓶颈。FlashAttention 通过分块计算（tiling）将 Q/K/V 分块送入 SRAM，在 on-chip 上完成 softmax 和 matmul，仅将最终输出写回 HBM，内存访问量降至 O(N^2 d^2 / M)，其中 M 是 SRAM 大小。

核心原理

- **Tiling**：将 Q/K/V 按块大小切分，每块在 SRAM 中计算局部注意力后累积到输出。
- **Online Softmax**：通过维护 running max 和 running sum 实现分块 softmax 的数值稳定计算。
- **IO** 复杂度：相比标准注意力的 O(N^2 d) HBM 访问，FlashAttention 降至 O(N^2 d^2 / M)，在长序列上提速 2-4x。
- **FlashAttention-2/3**：进一步优化 parallelism（2D grid）和 async（与 Tensor Core 流水），性能持续提升。

在源码中的实现

- `vllm/attention/backends/flash_attn.py` — FlashAttention 后端实现，封装 flash-attn 库调用。
- `vllm/attention/backends/rocm_flash_attn.py` — AMD ROCm 版本的 FlashAttention。
- `vllm/attention/layer.py` — Attention 层根据配置选择后端（FlashAttention、XFormers 等）。
- `vllm/attention/ops/` — 底层注意力 kernel 的自定义实现与封装。

相关概念

- paged-attention — FlashAttention kernel 需适配 PagedAttention 的非连续 KV 寻址
- kv-cache — FlashAttention 读取 KV Cache 进行注意力计算
- tensor-parallelism — FlashAttention kernel 需感知 TP 的 QKV 切分
- chunked-prefill — Chunked prefill 依赖 FlashAttention 的分块计算能力

> 将不活跃的 KV Cache 从 GPU 显存卸载到 CPU 内存或磁盘，在有限显存下支持更长上下文或更大 batch。

为什么需要 **KV Cache Offloading**

KV Cache 随序列长度和 batch size 线性增长，很快耗尽 GPU 显存。在长上下文场景（128K+ tokens）中，KV Cache 可能占用数十 GB 显存，远超模型权重本身。KV Cache Offloading 将暂时不活跃的 KV block 搬移到 CPU 内存（或磁盘），在需要时再搬回 GPU，以时间换空间。

核心原理

- **Swap** 机制：当 GPU 显存不足时，调度器选择部分 sequence 的 KV Cache swap 到 CPU pinned memory。
- **Recompute** 机制：也可选择丢弃 KV Cache 并在需要时从原始 token 重新计算，适用于重计算成本较低的场景。
- 预取（**Prefetch**）：在 sequence 即将被调度前异步将其 KV Cache 从 CPU 预取到 GPU，掩盖传输延迟。
- 层级管理：Hot KV 在 GPU、Warm KV 在 CPU、Cold KV 可选择丢弃，形成多级缓存层次。

在源码中的实现

- `vllm/worker/cache_engine.py` — `CacheEngine` 的 `swap_in` / `swap_out` 方法执行 GPU-CPU 数据搬运。
- `vllm/core/block_manager.py` — `BlockSpaceManager` 追踪哪些 block 在 GPU、哪些被 swap 到 CPU。
- `vllm/core/scheduler.py` — Scheduler 在显存不足时触发 preempt 和 swap 决策。
- `vllm/swap_manager.py` — 管理 swap buffer 的分配和异步传输。
- `vllm/v1/kv_offload/tiering/fs/` — 文件系统分层：`FileSystemTierManager` + `DualQueueThreadPool` 实现磁盘级 KV 缓存。
- `vllm/v1/kv_offload/file_mapper.py` — 将 KV block 哈希映射到文件路径（哈希子目录结构）。

- `vllm/distributed/kv_connector/v1/mooncake/` — MooncakeStore 分布式 KV 缓存连接器（支持多组 KV cache）。
- `vllm/distributed/kv_connector/v1/simple_cpu_offload_connector.py` — Simple CPU Offload 后端（支持 DSV4 混合注意力）。

相关概念

- kv-cache — 被卸载的核心数据结构
- paged-attention — Block 粒度的管理使 swap 高效（以 block 为单位搬运）
- continuous-batching — 调度器的 swap 决策与 continuous batching 深度耦合
- prefix-caching — 常用前缀的 KV Cache 可常驻 GPU，非活跃部分卸载到 CPU

> 存储注意力机制中 Key 和 Value 向量的缓存，避免自回归解码时重复计算已生成 token 的投影。

为什么需要 **KV Cache**

LLM 自回归生成时，每步需对全部历史 token 计算注意力。若不缓存 Key/Value，每步的计算量随序列长度线性增长，总复杂度退化为 O(n^2)。KV Cache 将已计算的 Key/Value 保留在显存中，使每步仅需计算新 token 的投影，将增量解码复杂度降至 O(1)。

核心原理

- **Prefill** 阶段：一次性计算 prompt 所有 token 的 K/V 并缓存。
- **Decode** 阶段：仅计算新 token 的 Q/K/V，K/V 追加到缓存，用全量 K/V 与当前 Q 计算注意力。
- 显存瓶颈：KV Cache 占用与 batch_size * seq_len * num_layers * hidden_dim 成正比，是长序列推理的主要显存开销。
- 量化压缩：可用 FP8/INT8/INT4 对 KV Cache 进行量化，降低显存占用与带宽压力。

在源码中的实现

- `vllm/worker/cache_engine.py` — `CacheEngine` 预分配 GPU/PU KV Cache 显存池。
- `vllm/core/kv_cache_manager.py` — `KVCacheManager` 追踪每个 sequence 的 KV block 使用情况。
- `vllm/attention/layer.py` — `Attention` 层的 forward 方法接收 `kv_cache` 参数并执行读写。
- `vllm/config.py` — `CacheConfig` 定义 block_size、gpu_memory_utilization、dtype 等缓存配置。

相关概念

- paged-attention — 管理 KV Cache 的分页机制
- kv-cache-offloading — 将 KV Cache 卸载到 CPU/磁盘以节省显存
- prefix-caching — 复用相同前缀的 KV Cache
- chunked-prefill — 分块计算 prefill 以控制 KV Cache 分配速率

> Low-Rank Adaptation，通过在冻结的预训练权重旁插入低秩矩阵来实现高效微调的参数高效方法。

为什么需要 LoRA

全量微调大模型需要更新所有参数，成本高昂且为每个任务保存完整模型副本不现实。LoRA 冻结原始权重 W，仅训练两个低秩矩阵 A（降维）和 B（升维），使可训练参数从 O(d^2) 降至 O(rd)（r << d）。在推理时，可将 BA 合并回 W 实现 zero inference overhead，也可保持分离以动态切换多个 LoRA adapter。

核心原理

- 低秩分解：原始投影 h = Wx 变为 h = Wx + BAx，其中 B 属于 R^{d x r}，A 属于 R^{r x d}，r 通常为 8-64。
- 多头支持：对 QKV 投影和 output 投影可分别插入 LoRA，灵活选择应用层。
- **Multi-LoRA**：vLLM 支持同一 batch 内不同请求使用不同 LoRA adapter，通过动态权重切换实现。
- 合并与非合并：静态部署时将 LoRA 权重合并到基础权重；动态服务时保持分离，按请求加载。

在源码中的实现

- `vllm/lora/layers.py` — `LinearWithLoRA`、`ColumnParallelLinearWithLoRA` 等 LoRA 包装层。
- `vllm/lora/models.py` — `LoRAModel` 管理 adapter 的加载、合并与切换。
- `vllm/lora/worker_manager.py` — Worker 级别的 LoRA 生命周期管理。
- `vllm/lora/request.py` — `LoRARequest` 封装请求级别的 adapter 选择。
- `vllm/config.py` — `LoRAConfig` 定义 max_lora_rank、max_loras 等限制参数。

相关概念

- tensor-parallelism — LoRA 层需与 TP 的并行线性层兼容
- cuda-graph — Multi-LoRA 场景下 CUDA Graph capture 需处理动态权重
- paged-attention — LoRA 与 PagedAttention 可独立使用
- flash-attention — LoRA 修改 QKV 投影，下游的 FlashAttention 无需改动

> 混合专家模型，通过路由机制在每次 forward 仅激活部分专家网络，以较低计算成本实现超大参数量。

为什么需要 **Mixture of Experts**

模型容量（参数量）与性能正相关，但全量激活的计算成本随参数线性增长。MoE 将模型中的 FFN 层替换为多个并行的专家网络（Expert），每个 token 仅由路由器（Router/Gate）选出的 1-2 个专家处理。以 DeepSeek-V3 为例，总参数 671B 但每个 token 仅激活 37B，计算成本与 37B 稠密模型相当。

核心原理

- **Router/Gate**：线性层将 hidden state 映射为专家分数，取 Top-K 选择活跃专家。
- **Expert** 网络：每个专家是一个独立的 FFN（或更复杂的网络），各自持有独立权重。
- 负载均衡：为防止路由坍缩（多数 token 走同一专家），通常加入 auxiliary loss 鼓励均匀分布。
- **Expert Parallelism**：专家按数量均匀分配到多个 GPU，每个 GPU 处理被路由到本地专家的 token。

在源码中的实现

- `vllm/model_executor/models/` — 支持 Mixtral、DeepSeek、Qwen-MoE 等多种 MoE 架构。
- `vllm/model_executor/layers/fused_moe/` — 高性能 MoE kernel 实现（CUDA、Triton）。
- `vllm/model_executor/layers/fused_moe/oracle/` — Oracle 框架统一后端选择（W4A8、WNA16 等量化方案）。
- `vllm/model_executor/layers/fused_moe/eep_reconfigure.py` — Elastic EP 动态扩展/缩减。
- `vllm/distributed/expert_parallel/` — Expert Parallelism 的通信原语（All-to-All、NIXL EP）。
- `vllm/distributed/elastic_ep/` — Elastic EP 的分阶段生命周期管理。
- `vllm/worker/worker.py` — Worker 根据 expert_parallel_size 加载对应专家。

- `vllm/config.py` — 模型配置中解析 num_experts、num_experts_per_tok 等参数。

相关概念

- tensor-parallelism — TP 与 Expert Parallelism 可组合使用
- pipeline-parallelism — 超大规模 MoE 可能需要 PP 跨节点
- cuda-graph — MoE kernel 的 CUDA Graph capture 需处理动态路由
- flash-attention — MoE 影响的是 FFN 层，Attention 层不变

> 借鉴操作系统虚拟内存分页机制，将 KV Cache 划分为固定大小的 block 进行非连续存储与管理的注意力算法。

为什么需要 **PagedAttention**

传统 LLM 推理中，KV Cache 需要预分配一块连续显存，导致严重的内部碎片（预留过多）或外部碎片（频繁分配释放后无法利用的小块）。PagedAttention 将 KV Cache 拆分为固定大小的 block，按需分配，显存利用率可从 20%-40% 提升到接近 100%。

核心原理

- 以 **Block** 为粒度：KV Cache 按 block（默认 16 个 token）为单位分配，不再要求物理连续。
- **Block Table**：每个 sequence 维护一张 block table，记录逻辑 block 到物理 block 的映射，类似 OS 的页表。
- **Copy-on-Write**：共享 prompt 前缀时，多个 sequence 可复用同一批物理 block，仅在生成不同 token 时才拷贝新 block。
- **Paged Attention Kernel**：自定义 CUDA kernel 通过 block table 间接索引 KV，支持非连续读取。

在源码中的实现

- `vllm/attention/backends/` — 多种注意力后端（FlashAttention、Roen、XFormers）均支持 paged 模式。
- `vllm/core/block_manager.py` — BlockSpaceManager 管理 block table 的分配、释放与交换。
- `vllm/worker/cache_engine.py` — CacheEngine 负责在 GPU/CPU 之间 swap block。
- `vllm/attention/layer.py` — Attention 层在 forward 时传入 block_tables 张量完成间接寻址。

相关概念

- kv-cache — PagedAttention 管理的核心数据结构
- prefix-caching — 基于 block 共享的自动前缀缓存
- flash-attention — 底层注意力 kernel 的实现基础
- continuous-batching — PagedAttention 使调度器能灵活管理变长序列

> 将模型按层切分到多个 GPU 上，每个 GPU 负责一组连续层的计算，形成流水线。

为什么需要 **Pipeline Parallelism**

当模型太大无法放入单个 GPU 且 Tensor Parallelism 因通信开销无法跨节点扩展时，Pipeline Parallelism（PP）提供了另一种切分维度。PP 将模型层分配到不同 GPU，每个 GPU 仅存储部分层，通信量为 hidden_dim 维度的激活值，远小于 TP 的 AllReduce，适合跨节点部署。

核心原理

- **Stage** 划分：模型按层均分到 PP 个 stage，每个 stage 是一组连续的 Transformer 层。
- **Micro-batching**：将 batch 拆分为多个 micro-batch 依次送入流水线，提高 GPU 利用率。
- 通信模式：相邻 stage 之间通过 P2P Send/Recv 传递中间激活，通信量 = batch_size * seq_len * hidden_dim。
- **Bubble** 问题：流水线存在空闲时间（bubble），PP 越大 bubble 越严重，通常 PP 度不超过 4-8。

在源码中的实现

- `vllm/distributed/pipeline_parallel/` — PP 通信原语（P2P Send/Recv）。
- `vllm/worker/worker.py` — 每个 Worker 根据 pipeline_rank 加载对应的模型层。
- `vllm/model_executor/models/` — 模型定义中通过 `get_pipeline_model` 切分层。
- `vllm/config.py` — `ParallelConfig.pipeline_parallel_size` 控制 PP 度。

相关概念

- tensor-parallelism — 通常与 PP 正交组合，形成 2D 并行
- cuda-graph — Pipeline 各 stage 可独立捕获 CUDA Graph

- kv-cache — Pipeline 各 stage 各自管理所属层的 KV Cache
- continuous-batching — PP 场景下的调度需考虑 stage 间的同步

> 自动识别并复用不同请求间相同 prompt 前缀的 KV Cache，避免重复计算。

为什么需要 **Prefix Caching**

在对话场景中，不同用户请求共享相同的 system prompt，或同一用户的多次请求共享历史对话。对这些共享前缀重复计算 KV 是巨大的浪费。Prefix Caching 利用 PagedAttention 的 block 机制实现 Copy-on-Write 共享，多个请求复用同一组物理 KV block，首 token 延迟和计算成本大幅降低。

核心原理

- **Hash** 索引：对 prompt token 序列计算 hash，在 block pool 中查找是否有完全匹配的前缀 block 链。
- **Copy-on-Write**：匹配的 block 被多个 sequence 共享引用，仅当 sequence 生成不同 token 时才分配新 block。
- 自动前缀匹配：vLLM 支持自动前缀匹配（Automatic Prefix Caching, APC），无需用户手动标记。
- **LRU** 淘汰：block pool 中的 cached block 按 LRU 策略淘汰，为新的 prefix 让出空间。

在源码中的实现

- `vllm/core/block_manager.py` — `BlockSpaceManager` 支持 `enable_caching` 模式，维护 cached block 的 hash 索引。
- `vllm/core/block/block_hash.py` — 定义 block 的 hash 计算与比较逻辑。
- `vllm/core/eviction_policy.py` — LRU 等淘汰策略管理 cached block 的生命周期。
- `vllm/config.py` — `CacheConfig.enable_prefix_caching` 开关。

相关概念

- paged-attention — Block 级共享是 prefix caching 的基础
- kv-cache — 被缓存和复用的数据结构
- chunked-prefill — 前缀命中后可跳过对应 chunk 的计算
- kv-cache-offloading — Prefix cache 也可卸载到 CPU 降低显存压力

> 使用小模型（draft model）快速生成候选 token，再由大模型（target model）并行验证，在不损失精度的前提下加速推理。

为什么需要 **Speculative Decoding**

自回归解码每步只能生成一个 token，GPU 利用率极低（memory-bound）。Speculative Decoding 利用 draft model 一次生成 K 个候选 token，target model 通过一次 forward 验证所有候选，接受匹配的 token 并拒绝不匹配的。平均接受率较高时，等效每步生成多个 token，显著降低延迟。

核心原理

- **Draft-then-Verify**：Draft model 自回归生成 K 个 token，Target model 对 K+1 个位置做一次 forward，得到每个位置的概率分布。
- 拒绝采样：对每个候选 token，若 Target model 的概率 >= Draft model 的概率则接受；否则按比例概率拒绝并以 Target model 的分布采样替代。
- 无损保证：数学上可证明输出分布与仅用 Target model 完全一致。
- 多种 **Draft** 来源：小型 LLM、Medusa head、EAGLE、n-gram 猜测等都可作为 draft 来源。

在源码中的实现

- `vllm/spec_decode/` — Speculative Decoding 的核心实现目录。
- `vllm/spec_decode/multi_step_worker.py` — Multi-step worker 驱动 draft 模型多步生成。
- `vllm/spec_decode/spec_decode_worker.py` — Target worker 验证候选 token 的主逻辑。
- `vllm/spec_decode/batch_expansion.py` — 将 K 个候选 token 展开为 batch 进行并行验证。
- `vllm/config.py` — `SpeculativeConfig` 定义 draft model、num_speculative_tokens 等参数。

相关概念

- continuous-batching — Speculative decoding 与 continuous batching 结合管理 draft/verify 请求
- kv-cache — Draft 和 Target 模型各自维护 KV Cache
- cuda-graph — Draft 模型的小 forward 可用 CUDA Graph 加速
- kv-cache-offloading — Draft model 的 KV Cache 可卸载以节省显存

> 将模型权重矩阵按维度切分到多个 GPU 上并行计算，也称 intra-layer parallelism。

为什么需要 **Tensor Parallelism**

大模型（如 LLaMA-70B、DeepSeek-V3）的单层权重远超单个 GPU 显存。Tensor Parallelism（TP）将每层的权重矩阵按行或列切分到多个 GPU，每个 GPU 仅持有部分权重和中间激活，使单 GPU 无法容纳的模型得以运行。

核心原理

- **Column Parallel**：权重按列切分，每个 GPU 计算 partial output，通过 AllReduce 聚合。用于 MLP 的 up/gate 投影和 Attention 的 QKV 投影。
- **Row Parallel**：权重按行切分，输入先切分，每个 GPU 计算部分结果后 AllReduce 求和。用于 MLP 的 down 投影和 Attention 的 output 投影。
- 通信模式：每层需要 2 次 AllReduce（Attention 后 + MLP 后），通信量与 hidden_dim 成正比。
- **NVLink** 依赖：TP 对通信带宽敏感，通常要求 GPU 间有 NVLink/NVSwitch 连接（节点内使用）。

在源码中的实现

- `vllm/distributed/tensor_parallel/` — TP 通信原语（AllReduce、Send/Recv）。
- `vllm/model_executor/layers/linear.py` — `ColumnParallelLinear` 和 `RowParallelLinear` 封装切分逻辑。
- `vllm/model_executor/layers/parallel_embedding.py` — Embedding 层的并行化。
- `vllm/config.py` — `ParallelConfig.tensor_parallel_size` 控制 TP 度。

相关概念

- pipeline-parallelism — 另一种维度的模型并行，跨层切分
- moe — 专家并行通常与 TP 结合使用
- cuda-graph — TP 场景下 CUDA Graph 需处理分布式通信节点
- flash-attention — 注意力 kernel 需感知 TP 的 QKV 切分

> PyTorch 的 JIT 编译器，通过图捕获、算子融合和后端优化将 eager 模式代码编译为高效 kernel。

为什么需要 **torch.compile**

PyTorch eager 模式每次 op 都经过 Python 调度和 GPU kernel launch，引入不必要的开销。torch.compile（基于 TorchDynamo + TorchInductor）自动将模型捕获为计算图，进行水平/垂直算子融合、内存布局优化和自动调优，生成融合 kernel，在不修改模型代码的前提下显著提升执行效率。

核心原理

- **TorchDynamo**：通过 frame evaluation hook 捕获 Python 字节码，将 eager 执行转换为 FX Graph。
- **TorchInductor**：后端编译器将 FX Graph lowering 为 Triton kernel（GPU）或 C++ kernel（CPU）。
- 算子融合：自动将连续的 pointwise op（如 SiLU + multiply）融合为单个 kernel，减少 HBM 访问。
- 动态形状：Inductor 支持动态形状符号化（symbolic shapes），适应 LLM 推理中变化的序列长度。

在源码中的实现

- `vllm/compiler/` — vLLM 的编译器集成层，封装 torch.compile 配置。
- `vllm/config.py` — `CompilationConfig` 控制 compile 级别（Dynamo、Inductor）、backend 选择。
- `vllm/worker/model_runner.py` — 模型加载后根据配置调用 `torch.compile()` 编译。
- `vllm/model_executor/models/` — 模型代码中通过 `torch.compiler.disable` 等装饰器标记不兼容的 op。

相关概念

- cuda-graph — torch.compile 优化后的模型可进一步通过 CUDA Graph 消除 launch 开销
- flash-attention — torch.compile 可融合注意力前后的 pointwise op
- lora — LoRA 动态权重切换需考虑 compile 后的静态图限制
- tensor-parallelism — TP 的集合通信 op 需被 compile 正确处理

vllm

vllm 学习笔记

高吞吐、低延迟的 LLM 推理与服务引擎源码学习

开始学习 知识地图

PagedAttention

类似操作系统虚拟内存的分页 KV 缓存管理，实现高效显存利用与共享

Continuous Batching

连续批处理调度，动态插入新请求、移除已完成请求，最大化 GPU 利用率

分布式推理

支持张量并行、流水线并行、专家并行、数据并行等多种分布式策略

200+ 模型架构

覆盖 LLaMA、Qwen、DeepSeek、Gemma、Mixtral 等主流大语言模型

量化与加速

支持 FP8、GPTQ、AWQ、GGUF 等 30+ 量化方法，EAGLE/Medusa 推测解码

OpenAI 兼容 API

开箱即用的 OpenAI 兼容服务，支持 Chat/Completion/Embedding/Responses 接口

学习日志

进度总览

- [ ] 初始化学习站点
- [ ] 完成架构概览主题
- [ ] 完成引擎核心主题
- [ ] 完成调度系统主题
- [ ] 完成 KV 缓存与 PagedAttention 主题
- [ ] 完成执行器与 Worker 主题
- [ ] 完成模型库与算子层主题
- [ ] 完成量化系统主题
- [ ] 完成推测解码主题
- [ ] 完成多模态处理主题
- [ ] 完成分布式计算主题
- [ ] 完成 API 服务与部署主题

Day 1 — 架构概览

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 2 — 引擎核心

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 3 — 调度系统

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 4 — KV 缓存与 PagedAttention

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 5 — 执行器与 Worker

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 6 — 模型库与算子层

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 7 — 量化系统

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 8 — 推测解码

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 9 — 多模态处理

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 10 — 分布式计算

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

Day 11 — API 服务与部署

日期： 状态：

今日摘要

-

困惑与突破

- 困惑：
- 突破：

上游同步 — 2025-05-26

vllm 子模块：ff712f644 → d5cf7b4a2（247 commits）

主要变更

1. **Offline API** 拆分：`LLM` 类重构为 Mixin 组合架构（`OfflineInferenceMixin` + `BeamSearchOfflineMixin` + `PoolingOfflineMixin`）
2. **Rust Frontend** 集成：`RustFrontendProcessManager` 管理 `vllm-rs` 二进制前端
3. **DP Supervisor**：多端口数据并行管理器，为每个 DP rank 生成独立 server
4. **MooncakeStore** 多组支持：支持混合注意力模型的多组 KV 缓存（如 DeepSeek V4 的 MLA + SWA）
5. 文件系统分层：新增 `FileSystemTierManager`，纯 Python 磁盘后端 KV 缓存
6. **Elastic EP**：弹性专家并行，在线扩展/缩减 EP 大小
7. **MoE Oracle** 框架：统一后端选择（Triton、CUTLASS、FlashInfer B12x 等）
8. **Mamba** 推测解码融合内核：消除 CPU-GPU 同步瓶颈
9. **GDN** 子系统：Gated DeltaNet 注意力层（Kimi、OLMo、Qwen）
10. **DeepSeek V4** 模型隔离：移出主仓库到 `vllm/models/deepseek_v4/`
11. **NVFP4** 混合精度：新增 W4A16_NVFP4 量化模式
12. **Compressed Tensors** 稀疏性移除：2:4 稀疏性支持已弃用

更新的笔记章节

- topics/architecture/concepts — 前端模式、源码结构
- topics/serving/concepts — Rust Frontend、DP Supervisor、Mixin 架构
- topics/serving/code-walkthrough — LLM 类重构、前端管理
- topics/kv-cache/concepts — 文件系统分层、MooncakeStore 多组
- topics/scheduling/concepts — 异步抢占多帧丢弃、KV connector 延迟释放
- topics/model-layers/concepts — Elastic EP、MoE 后端矩阵、GDN、模型隔离
- topics/distributed/concepts — Elastic EP 生命周期、NIXL EP 重构
- topics/speculative-decoding/concepts — Mamba 融合内核、EAGLE-3 后规范
- topics/quantization/concepts — NVFP4、AWQ oracle 统一、稀疏性移除
- topics/executor-worker/concepts — CuMemAllocator 解耦
- topics/multimodal/concepts — OpenVLA、编码器 CUDA Graph

- kv-cache-offloading — 文件系统分层、MooncakeStore
- moe — Oracle 框架、Elastic EP

出链

架构概览 — 概念

API服务与部署 — 概念

API服务与部署 — 代码走读

KV缓存与PagedAttention — 概念

调度系统 — 概念

模型库与算子层 — 概念

分布式计算 — 概念

推测解码 — 概念

量化系统 — 概念

执行器与Worker — 概念

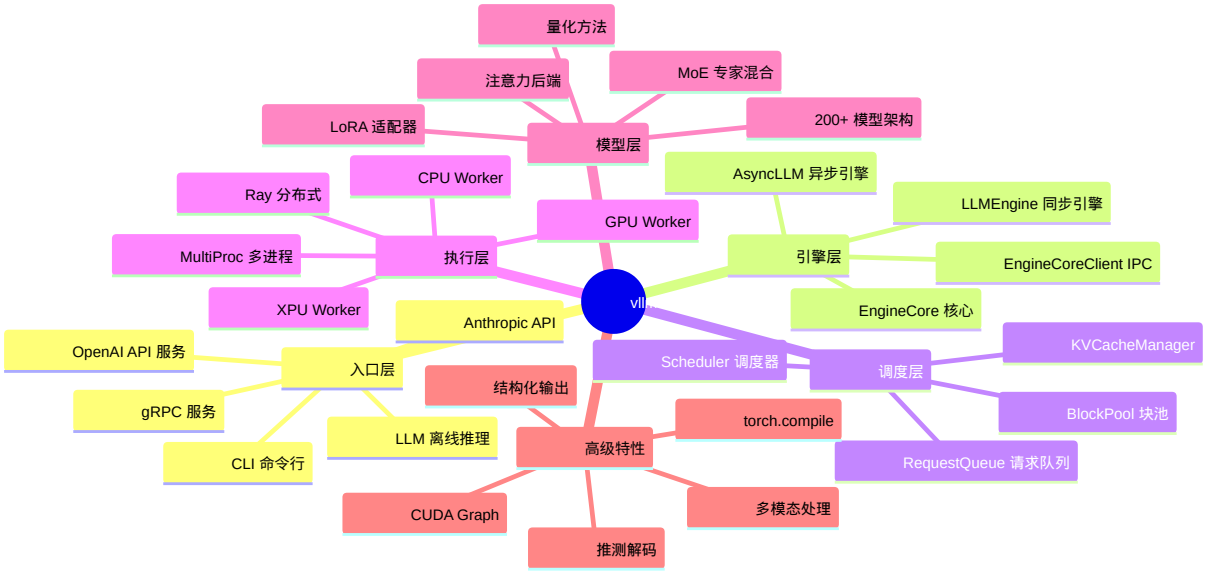
多模态处理 — 概念

vllm 学习笔记 — 知识地图

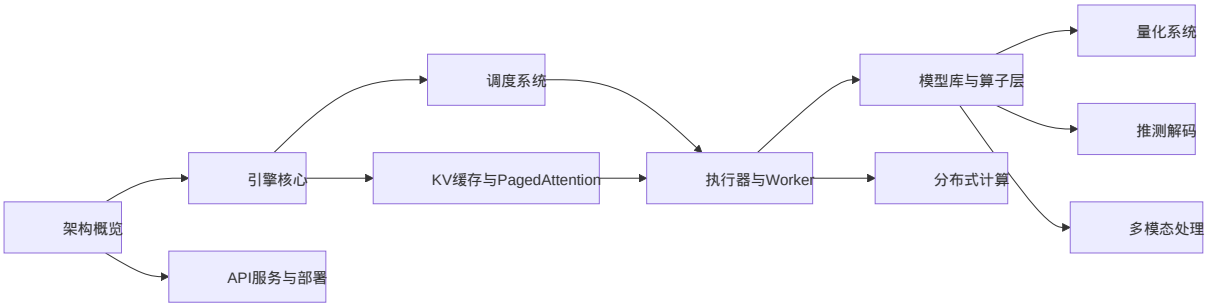
项目概览

vLLM 是一个高吞吐、低延迟的大语言模型推理和服务引擎。核心创新包括 PagedAttention（分页注意力机制）用于 KV 缓存的内存管理、连续批处理（Continuous Batching）以及广泛的硬件和模型支持（200+ 模型架构）。

架构全景



学习路径



知识索引

系统架构知识点

概念	主题	术语
请求流转（HTTP → Token → 输出）	架构概览	—
VllmConfig 聚合配置	架构概念	—
EngineCore 独立进程模型	引擎核心	—
ZMQ IPC（DEALER/ROUTER）	引擎概念	—
AsyncLLM 流式输出	引擎核心	—
EngineCoreClient 同步/异步模式	引擎概念	—

调度与缓存知识点

概念	主题	术语
Continuous Batching	调度系统	Continuous Batching
Preemption（抢占）策略	调度概念	—
Chunked Prefill	调度概念	Chunked Prefill
PagedAttention 分页注意力	KV缓存	PagedAttention
KV Cache 块管理	KV缓存	KV Cache
前缀缓存（Prefix Caching）	KV缓存概念	Prefix Caching
KV Cache 卸载（Offloading）	KV缓存概念	KV Cache Offloading

执行与模型知识点

概念	主题	术语
MultiProcExecutor 多进程执行	执行器	—
CUDA Graph 捕获与回放	执行器概念	CUDA Graph
显存分配与 profiling	执行器概念	—
ModelRegistry 模型注册	模型库	—
注意力后端（FlashAttention/FlashInfer/Triton）	模型概念	FlashAttention
MoE 混合专家	模型概念	Mixture of Experts
LoRA 适配器	模型概念	LoRA

量化与加速知识点

概念	主题	术语
FP8 / INT4 量化方法	量化系统	—
GPTQ / AWQ 量化策略	量化概念	—
Marlin 量化 kernel	量化概念	—
推测解码（Speculative Decoding）	推测解码	Speculative Decoding
EAGLE / Medusa / N-gram 草稿模型	推测概念	—
拒绝采样验证	推测概念	—

多模态与分布式知识点

概念	主题	术语
Visual Token 编码	多模态	—
编码器缓存策略	多模态概念	—
多模态输入处理管线	多模态概念	—
Tensor Parallelism	分布式	Tensor Parallelism
Pipeline Parallelism	分布式	Pipeline Parallelism
Expert Parallelism	分布式概念	—
All-Reduce 通信量分析	分布式概念	—

服务与部署知识点

概念	主题	术语
vllm serve 配置参数	API服务	—
TTFT / 吞吐量调优	服务概念	—
Prefix Caching 服务端配置	服务概念	Prefix Caching
结构化输出（JSON Schema）	服务概念	—
torch.compile 编译优化	服务概念	torch.compile

学习进度

- [ ] 架构概览
- [ ] 引擎核心
- [ ] 调度系统
- [ ] KV缓存与PagedAttention
- [ ] 执行器与Worker
- [ ] 模型库与算子层
- [ ] 量化系统
- [ ] 推测解码
- [ ] 多模态处理

- [ ] 分布式计算
- [ ] API服务与部署

出链

/topics/architecture/\

/topics/architecture/concepts\

/topics/engine-core/\

/topics/engine-core/concepts\

/topics/scheduling/\

/glossary/continuous-batching\

/topics/scheduling/concepts\

/glossary/chunked-prefill\

/topics/kv-cache/\

/glossary/paged-attention\

/glossary/kv-cache\

/topics/kv-cache/concepts\

/glossary/prefix-caching\

/glossary/kv-cache-offloading\

/topics/executor-worker/\

/topics/executor-worker/concepts\

/glossary/cuda-graph\

/topics/model-layers/\

/topics/model-layers/concepts\

/glossary/flash-attention\

/glossary/moe\

/glossary/lora\

/topics/quantization/\

/topics/quantization/concepts\

/topics/speculative-decoding/\

/glossary/speculative-decoding\

/topics/speculative-decoding/concepts\

/topics/multimodal/\

/topics/multimodal/concepts\

/topics/distributed/\

/glossary/tensor-parallelism\

/glossary/pipeline-parallelism\

/topics/distributed/concepts\

/topics/serving/\

/topics/serving/concepts\

/glossary/torch-compile\

架构概览 — 代码走读

入口点走读

CLI 入口 — `vllm/entrypoints/cli/main.py`

`vllm` 命令行工具的入口，注册了以下子命令：

```
    # 简化的命令注册
    subparsers = parser.add_subparsers()

    serve_parser = subparsers.add_parser("serve")
    serve_parser.set_defaults(func=run_server)

    openai_parser = subparsers.add_parser("openai")
    openai_parser.set_defaults(func=run_openai_server)
```

> 关键路径：`vllm serve` → `run_server()` → 创建 `AsyncLLM` → 启动 FastAPI。

OpenAI API Server — `vllm/entrypoints/openai/api_server.py`

FastAPI 应用入口，负责：

1. 注册路由：`/v1/chat/completions`、`/v1/completions`、`/v1/embeddings`、`/v1/responses`
2. 初始化 `AsyncLLMEngine`（实际是 `AsyncLLM` 的别名）
3. 配置中间件（CORS、限流、健康检查）

```
    # 简化的初始化
    engine = AsyncLLMEngine.from_engine_args(engine_args)
    app = FastAPI()
    app.include_router(chat_router)
    app.include_router(completion_router)
```

离线 LLM 类 — `vllm/entrypoints/llm.py`

提供同步的批处理推理接口，采用 Mixin 组合架构：

```
llm = LLM(model="meta-llama/Llama-3-8B")
outputs = llm.generate(["Hello, world!"])

# 推测解码快捷参数
llm = LLM(model="...", spec_method="ngram", spec_tokens=5)
```

内部创建 `LLMEngine`（V1 中是 `AsyncLLM` 的同步包装），循环调用 `step()` 直到所有请求完成。

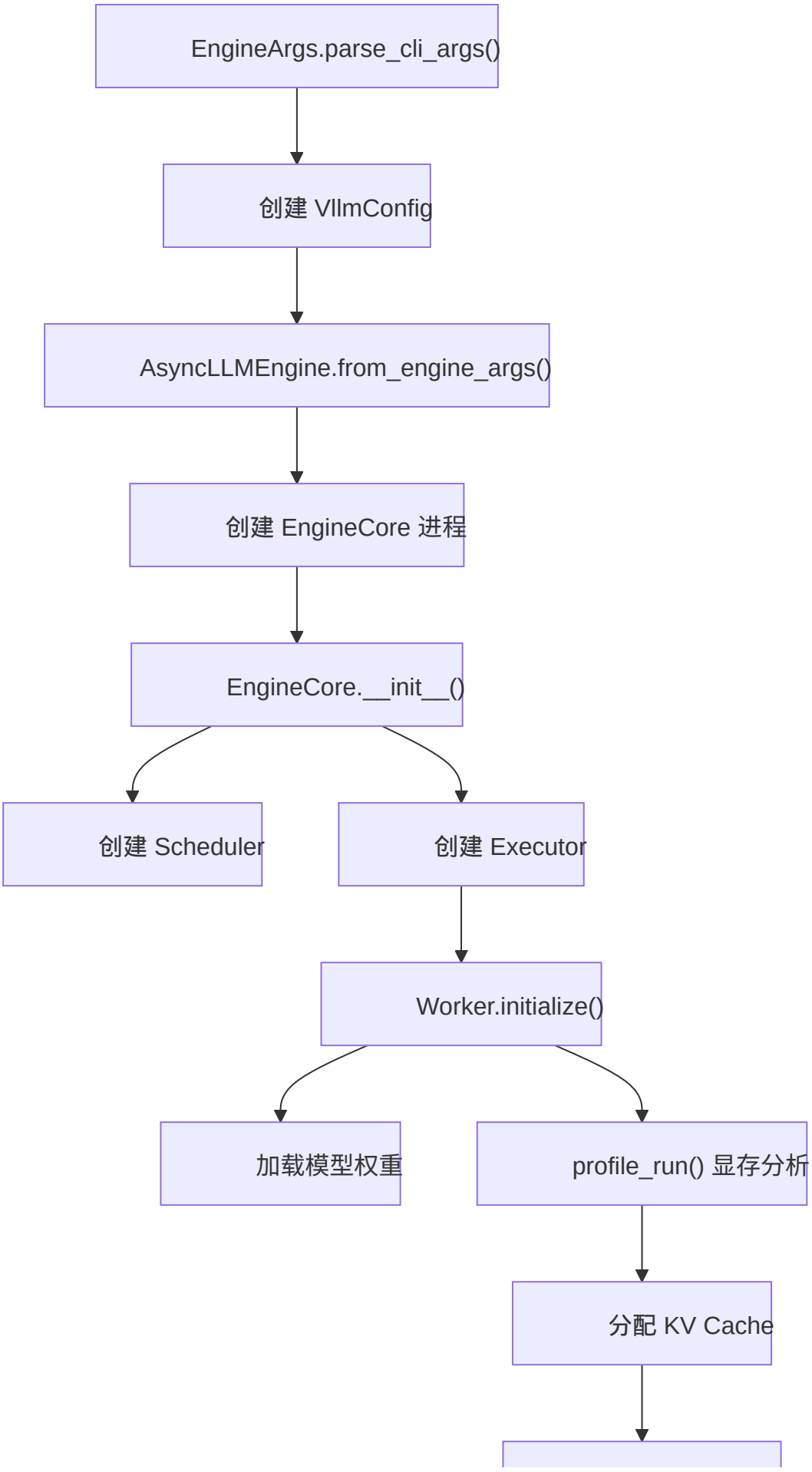
LLM 类的继承链：

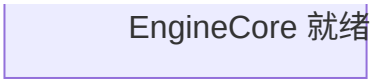
```
LLM → BeamSearchOfflineMixin → PoolingOfflineMixin → OfflineInferenceMixin
```

核心推理逻辑在 `OfflineInferenceMixin`（`entrypoints/offline_utils.py`），beam search 在 `entrypoints/generate/beam_search/offline.py`。

引擎初始化走读

EngineCore 创建流程


EngineArgs.parse_cli_args()
创建 VllmConfig
AsyncLLMEngine.from_engine_args()
创建 EngineCore 进程
EngineCore.__init__()
创建 Scheduler
创建 Executor
Worker.initialize()
加载模型权重
profile_run() 显存分析
分配 KV Cache



关键文件路径：

- `vllm/engine/arg_utils.py` — CLI 参数解析为 `EngineArgs`
- `vllm/v1/engine/core.py` — `EngineCore` 初始化与主循环
- `vllm/v1/executor/multiproc_executor.py` — 多进程 Worker 创建

配置系统走读

VllmConfig 聚合 — `vllm/config/vllm.py`

`VllmConfig` 是所有子配置的聚合器：

```
@dataclass
class VllmConfig:
    model_config: ModelConfig
    cache_config: CacheConfig
    parallel_config: ParallelConfig
    scheduler_config: SchedulerConfig
    compilation_config: CompilationConfig
    # ... 30+ 子配置
```

配置验证链

配置之间有复杂的依赖关系。例如：

1. `ModelConfig` 确定模型的 `head_dim` 和 `num_layers`
2. `CacheConfig` 依赖这些参数计算 `block_size`
3. `SchedulerConfig` 依赖 `CacheConfig` 确定 `max_num_seqs`

模块组织走读

源码目录结构

```
vllm/
├── entrypoints/          # 入口层
│   ├── cli/              # CLI 子命令：serve、openai
│   ├── openai/           # OpenAI API Server
│   │   ├── api_server.py # FastAPI 应用
│   │   └── dp_supervisor.py # 多端口数据并行管理器
│   ├── llm.py            # LLM 离线类（Mixin 组合）
│   ├── offline_utils.py  # OfflineInferenceMixin
│   └── generate/
│       └── beam_search/  # beam search 拆分模块
├── models/               # 外部模型包（特殊硬件依赖）
│   └── deepseek_v4/      # DeepSeek V4（自定义 CUDA kernel）
├── engine/               # 引擎层：EngineClient 协议、EngineArgs
├── v1/
│   ├── engine/           # V1 引擎实现：Core、AsyncLLM、CoreClient
│   ├── core/             # 调度层：Scheduler、KVCacheManager
│   ├── executor/         # 执行层：MultiProc、Ray、UniProc
│   ├── worker/           # Worker：GPU/CPU/XPU、ModelRunner
│   ├── attention/        # 注意力后端：FlashAttn、FlashInfer 等
│   ├── spec_decode/      # 推测解码：EAGLE、Medusa、N-gram
│   ├── structured_output/ # 结构化输出：xgrammar、outlines
│   ├── sample/           # 采样：top-k、top-p、temperature
│   ├── metrics/          # 指标：Prometheus、统计
│   └── kv_offload/       # KV 缓存卸载（含文件系统分层）
├── model_executor/
│   ├── models/           # 200+ 模型实现
│   ├── layers/           # 可复用 NN 层（含 fused_moe、mamba/gdn）
│   └── quantization/     # 量化方法
├── config/               # 30+ 配置 dataclass
├── distributed/          # 分布式：TP、PP、DP、EP（含 Elastic EP）
├── multimodal/           # 多模态：图像、音频、视频
├── lora/                 # LoRA 适配器
├── platforms/            # 硬件平台：CUDA、ROCm、CPU、TPU
├── tokenizers/           # 分词器
└── compilation/          # torch.compile、CUDA Graph
```

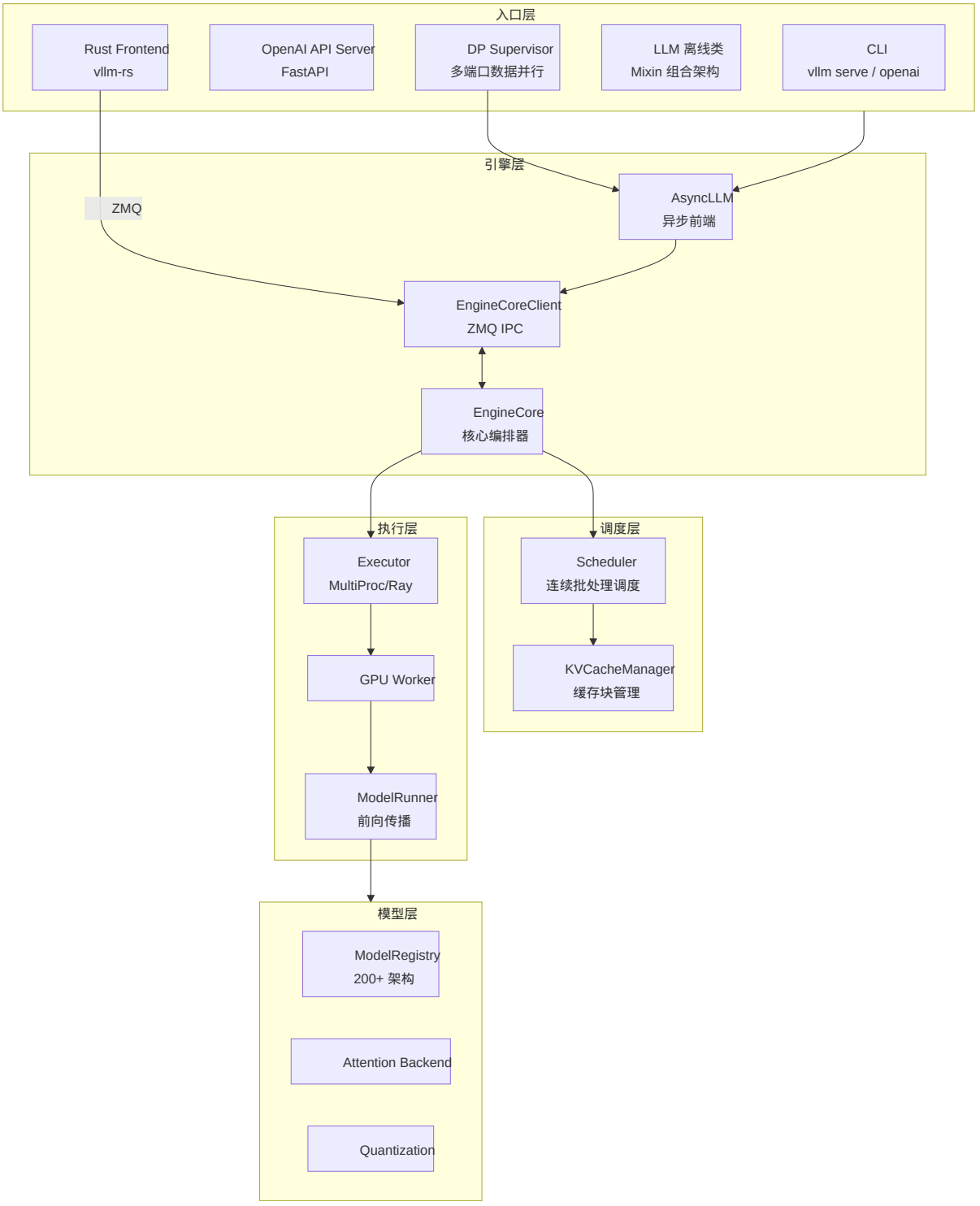
关键函数索引

函数/类	文件	职责
`EngineCore.__init__()`	v1/engine/core.py	初始化调度器、执行器、KV 缓存
`EngineCore.run_loop()`	v1/engine/core.py	主循环：调度 → 执行 → 输出处理
`AsyncLLM.generate()`	v1/engine/async_llm.py	异步推理入口
`LLM.generate()`	entrypoints/llm.py	同步推理入口
`EngineArgs.create_engine_config()`	engine/arg_utils.py	CLI 参数 → VllmConfig
`Scheduler.schedule()`	v1/core/sched/scheduler.py	连续批处理调度决策
`Executor.execute_model()`	v1/executor/abstract.py	执行层抽象接口
`ModelRunner.forward()`	v1/worker/gpu_model_runner.py	GPU 模型前向传播

架构概览 — 概念

分层架构

vLLM 采用清晰的分层设计，各层之间通过定义良好的接口通信：



核心组件

EngineCore

topics/engine-core/ 是 vLLM V1 架构的心脏，运行在独立进程中：

- 接收来自前端（AsyncLLM/LLMEngine）的请求
- 每轮迭代调用 Scheduler 决策
- 通过 Executor 分发到 Worker 执行模型推理
- 通过 ZMQ 与前端进行进程间通信

Scheduler

调度器 是系统的大脑，负责：

- 连续批处理（Continuous Batching）：动态插入新请求，移除已完成请求
- **Chunked Prefill**：将长 prompt 的 prefill 拆分为多个 chunk
- 前缀缓存（Prefix Caching）：复用共享前缀的 KV 缓存块
- 抢占与交换：在显存不足时抢占低优先级请求

Worker 与 ModelRunner

Worker 是执行层的核心抽象：

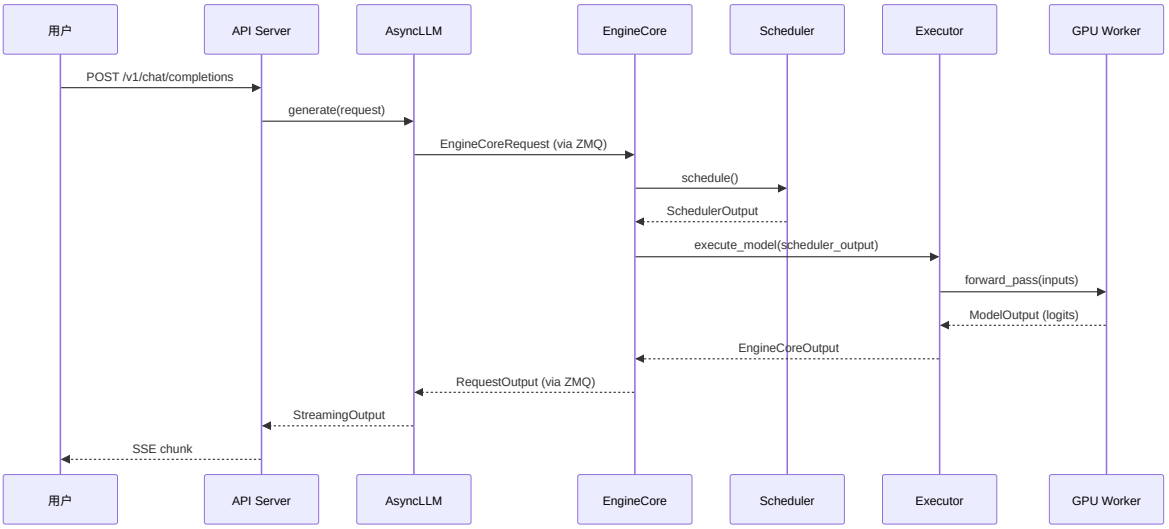
- **GPU Worker**：管理 CUDA 设备初始化、模型加载、KV 缓存分配
- **CuMemAllocator**：CUDA 内存池分配器，现在作为独立特性（`enable_cumem_allocator`），不再与 sleep mode 耦合
- **ModelRunner**（vllm/v1/worker/gpu_model_runner.py）：执行前向传播，管理 CUDA Graph、LoRA、推测解码集成
- **Mamba** 推测解码：融合 Triton 内核消除了 CPU-GPU 同步瓶颈

配置系统

vLLM 的配置系统基于 `VllmConfig`，聚合了 30+ 子配置（dataclass）：

配置	文件	职责
`VllmConfig`	config/vllm.py	主配置聚合器
`ModelConfig`	config/model.py	模型名称、dtype、最大长度
`ParallelConfig`	config/parallel.py	TP/PP/DP/EP 并行配置
`SchedulerConfig`	config/scheduler.py	最大 token 数、批大小
`CacheConfig`	config/cache.py	KV 缓存块大小、swap 空间
`CompilationConfig`	config/compilation.py	torch.compile、CUDA Graph 设置

数据流：请求生命周期



V1 架构改进

V1 架构相较于旧版的主要改进：

1. **EngineCore** 独立进程：调度与 API 服务解耦，避免 GIL 竞争
2. **ZMQ IPC** 通信：高效进程间数据传递
3. 统一的 **Scheduler**：合并了旧版的多个调度逻辑
4. **CUDA Graph** 全面集成：decode 阶段零开销图重放
5. **torch.compile** 支持：自动算子融合与优化

相关概念

- topics/engine-core/ — 引擎核心的详细实现
- Continuous Batching — 连续批处理调度策略
- CUDA Graph — CUDA 图捕获与重放机制
- Flash Attention — 高效注意力计算内核
- Paged Attention — 分页 KV 缓存管理

反向链接

学习日志

出链

/topics/engine-core/

/topics/scheduling/

架构概览 — 练习

练习 1：绘制请求流转图

根据你对架构的理解，绘制一个请求从 `vllm serve` 启动到返回第一个 token 的完整流程图。

> 参考答案
>
> 关键步骤：
>
> 1. FastAPI 接收 HTTP 请求，解析为 `ChatCompletionRequest`
> 2. `AsyncLLM.generate()` 将请求通过 `InputProcessor` tokenize
> 3. 通过 `EngineCoreClient`（ZMQ）发送 `EngineCoreRequest` 到 `EngineCore`
> 4. `Scheduler.schedule()` 决定是否进行 prefill
> 5. `Executor.execute_model()` 分发到 GPU Worker
> 6. `ModelRunner` 执行模型前向传播
> 7. `OutputProcessor` 将 logits 转换为 token
> 8. 通过 ZMQ 返回给 `AsyncLLM`，通过 SSE 流式返回给客户端

练习 2：配置系统分析

阅读 `vllm/config/vllm.py`，回答以下问题：

1. `VllmConfig` 包含哪些子配置？它们之间的依赖关系是什么？
2. `ModelConfig` 如何决定使用哪个模型实现？
3. `ParallelConfig` 如何影响 Worker 的创建？

参考答案

1. `VllmConfig` 是聚合配置，包含 `model_config`、`parallel_config`、`scheduler_config`、`cache_config`、`compilation_config` 等 30+ 子配置。依赖关系：`cache_config` 依赖 `model_config`（block_size 由模型头数决定），`scheduler_config` 依赖 `cache_config`（max_num_seqs 受缓存大小限制）。
2. `ModelConfig` 通过 `architectures` 字段匹配 `ModelRegistry` 中注册的模型类。每个 HuggingFace 模型配置中声明了 `architectures`，vLLM 以此为键查找对应实现。
3. `ParallelConfig` 定义了 `tensor_parallel_size`、`pipeline_parallel_size` 等。`MultiProcExecutor` 据此创建对应数量的 Worker 进程，每个 Worker 绑定到特定的 GPU。

练习 3：进程模型分析

分析 vLLM 的多进程模型：

1. `EngineCore` 为什么运行在独立进程中？
2. `EngineCoreClient` 如何实现进程间通信？
3. Worker 进程如何与 EngineCore 交互？

参考答案

1. 独立进程可以避免 Python GIL 对调度和推理的竞争。API 服务器处理网络 I/O 不影响 EngineCore 的调度循环。同时支持多前端（多个 API server）连接到同一个 EngineCore。
2. `EngineCoreClient` 使用 ZMQ 进行 IPC。支持两种模式：`SYNC`（同步调用）和 `ASYNC`（异步非阻塞）。通过 ZMQ 的 `DEALER/ROUTER` 模式实现请求-响应匹配。
3. Worker 不直接与 EngineCore 交互。EngineCore 通过 Executor 发送 `SchedulerOutput` 给 Worker，Worker 执行后返回 `ModelOutput`。Executor 负责进程间通信（pipe 或 Ray）。

拓展挑战

- 阅读 `vllm/v1/engine/core.py` 的 `run_loop` 方法，理解 EngineCore 的主循环逻辑
- 对比 V0 和 V1 的 `AsyncLLMEngine` 实现，分析架构改进带来的性能提升
- 研究 `vllm/entrypoints/openai/api_server.py` 的中间件链，理解限流、认证等机制

架构概览

> vLLM 的整体架构设计：从入口层到引擎核心、调度器、执行器、模型层的分层结构，理解请求如何在系统中流转。

涵盖内容

章节	核心主题
概念	分层架构、核心组件、数据流
练习	架构图绘制、组件交互分析
代码走读	入口点、配置系统、模块组织

核心概念

vLLM 采用分层架构设计，自上而下分为：

- 入口层（Entrypoints）：CLI、OpenAI API、离线 LLM 类等用户接口
- 引擎层（Engine）：topics/engine-core/ 作为中央编排器，管理调度和执行
- 调度层（Scheduler）：决定每轮迭代中哪些请求进行 prefill、decode 或 preemption
- 执行层（Executor/Worker）：GPU/CPU/XPU Worker，负责模型加载和前向传播
- 模型层（Models）：200+ 模型实现、注意力后端、量化方法

前置知识

- Python 异步编程基础
- GPU 计算基本概念（CUDA、显存管理）
- Transformer 架构基本原理

学习路径

读完本主题后，你将理解：

- vLLM 的整体分层架构和模块组织
- 一个推理请求从提交到返回结果的完整生命周期
- 核心组件之间的通信机制（ZMQ IPC、进程间协调）
- V1 架构相较于旧版的设计改进

→ 下一步：引擎核心

反向链接

引擎核心

API服务与部署

分布式计算 — 代码走读

Parallel State — `vllm/distributed/parallel_state.py`

```
class ParallelState:
    """管理所有并行相关的进程组"""

    def __init__(self):
        self.tp_group = None    # Tensor Parallel group
        self.pp_group = None    # Pipeline Parallel group
        self.dp_group = None    # Data Parallel group
        self.ep_group = None    # Expert Parallel group

    def initialize(self, parallel_config):
        world_size = parallel_config.world_size
        tp_size = parallel_config.tensor_parallel_size
        pp_size = parallel_config.pipeline_parallel_size

        # 创建 TP 进程组
        for i in range(world_size // tp_size):
            ranks = list(range(i * tp_size, (i + 1) * tp_size))
            group = dist.new_group(ranks)
            for rank in ranks:
                self.tp_group[rank] = group
```

通信操作

Tensor Parallel 通信

```python
def tensor_model_parallel_all_reduce(input_):
    """TP 组内的 All-Reduce"""
    group = get_tp_group()
    dist.all_reduce(input_, group=group)
    return input_

def tensor_model_parallel_all_gather(input_, dim=-1):
    """TP 组内的 All-Gather"""
    group = get_tp_group()
    world_size = get_tp_world_size()
    tensor_list = [torch.empty_like(input_) for _ in range(world_size)]
    dist.all_gather(tensor_list, input_, group=group)
    return torch.cat(tensor_list, dim=dim)
```

KV Transfer — `vllm/distributed/kv_transfer/`

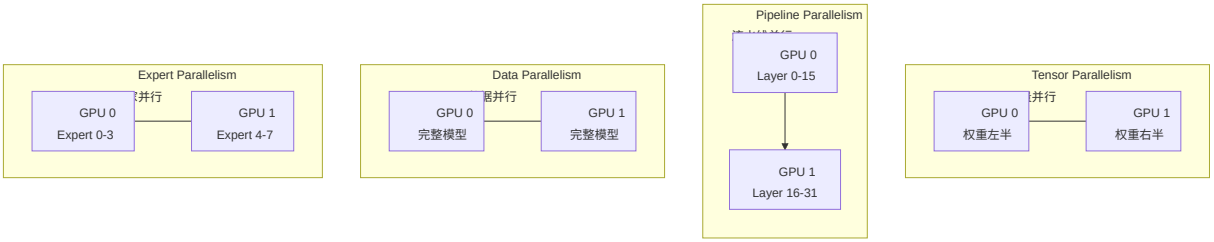
```python
class KVTransferAgent:
    """跨 GPU 的 KV 缓存传输"""

    def send_kv_cache(self, kv_cache, request_id):
        """发送 KV 缓存到目标 GPU"""
        data = self._serialize_kv_cache(kv_cache)
        self.transport.send(data, request_id)

    def recv_kv_cache(self, request_id):
        """接收 KV 缓存"""
        data = self.transport.recv(request_id)
        return self._deserialize_kv_cache(data)
```

关键函数索引

函数/类	文件	职责
`ParallelState.initialize()`	distributed/parallel_state.py	初始化进程组
`tensor_model_parallel_all_reduce()`	distributed/parallel_state.py	TP All-Reduce
`tensor_model_parallel_all_gather()`	distributed/parallel_state.py	TP All-Gather
`get_tp_group()`	distributed/parallel_state.py	获取 TP 进程组
`KVTransferAgent`	distributed/kv_transfer/	KV 缓存传输

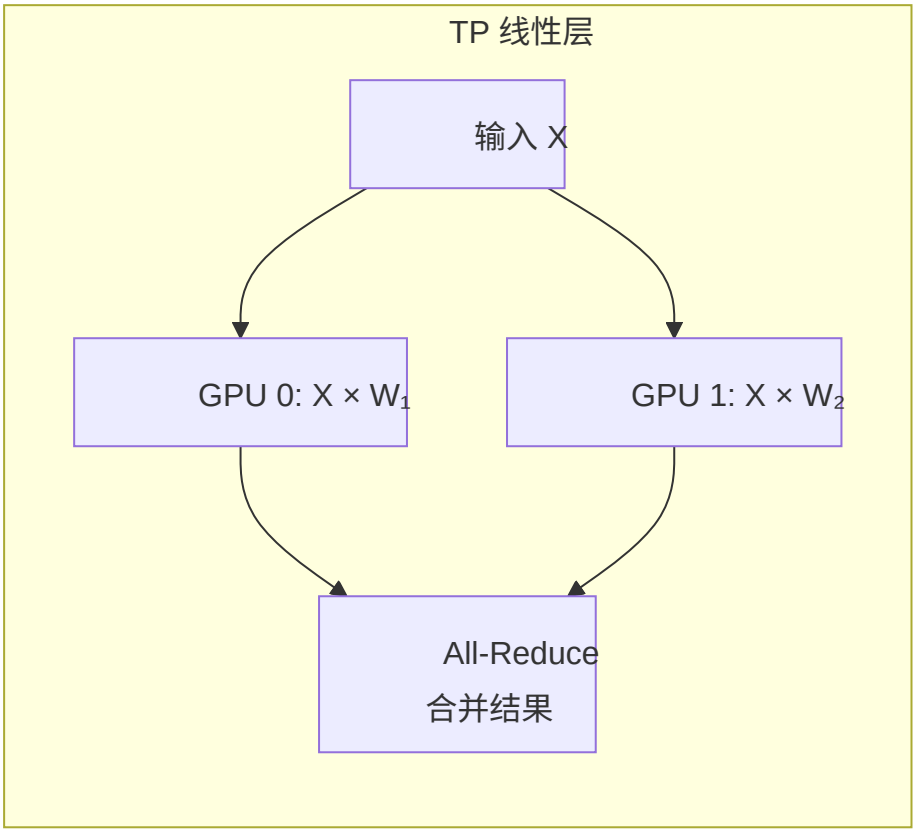
分布式计算 — 概念

并行策略概览



Tensor Parallelism（TP）

将模型的权重矩阵按列或行切分到多个 GPU：



ColumnParallelLinear

```
Y = XW, W 按列切分为 [W₁, W₂]
GPU 0: Y₁ = XW₁
GPU 1: Y₂ = XW₂
Y = [Y₁, Y₂]  (无需通信)
```

RowParallelLinear

```
Y = XW, W 按行切分为 [W₁; W₂]
GPU 0: Y₁ = X₁W₁  (X 按列切分)
GPU 1: Y₂ = X₂W₂
Y = Y₁ + Y₂  (需要 All-Reduce)
```

TP 的通信开销

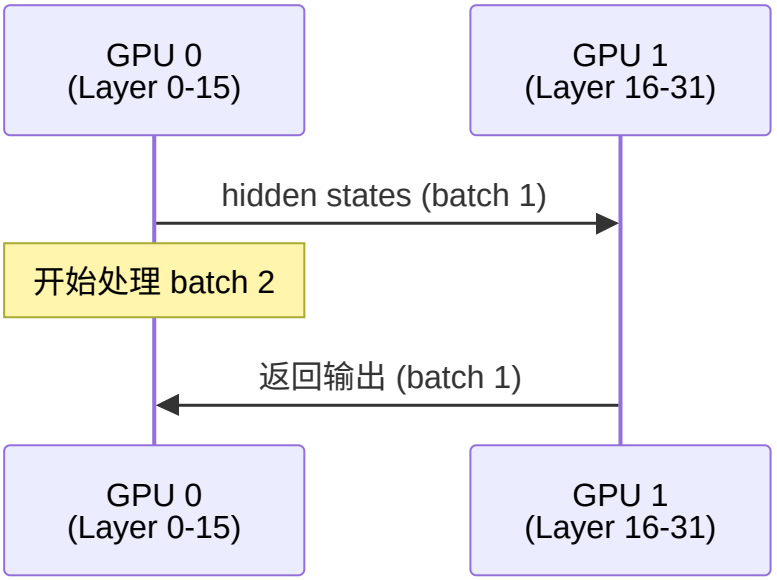
每个 Transformer 层需要 2 次 All-Reduce：

- Attention 后的 output projection
- MLP 后的 output

> 通信量与 hidden_size 成正比，与 sequence length 无关。

Pipeline Parallelism（PP）

将模型的不同层分配到不同 GPU：



PP 的挑战

- 气泡（bubble）：GPU 之间存在等待时间
- 微批次（micro-batching）：将请求分成小批次以减少气泡
- **KV** 缓存：需要跨 GPU 传递 KV 缓存

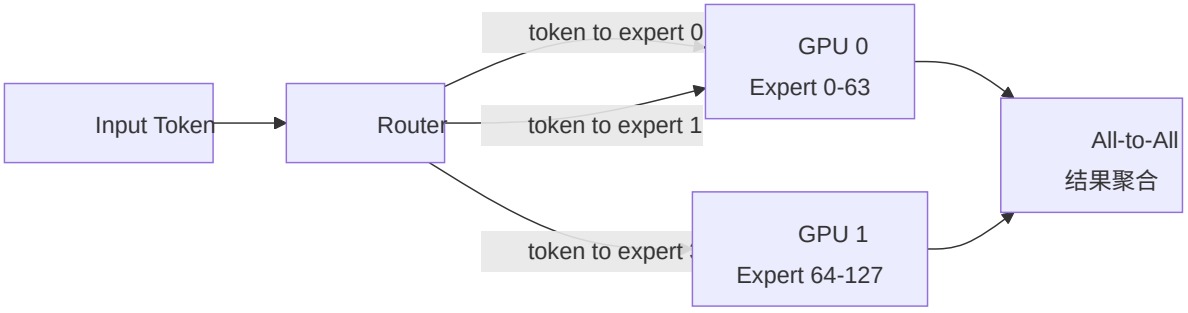
Data Parallelism（DP）

复制整个模型到多个 GPU，每个 GPU 处理不同的请求：

- 不需要模型权重通信
- 需要请求在 GPU 间均衡分配
- 适合小模型高并发场景

Expert Parallelism（EP）

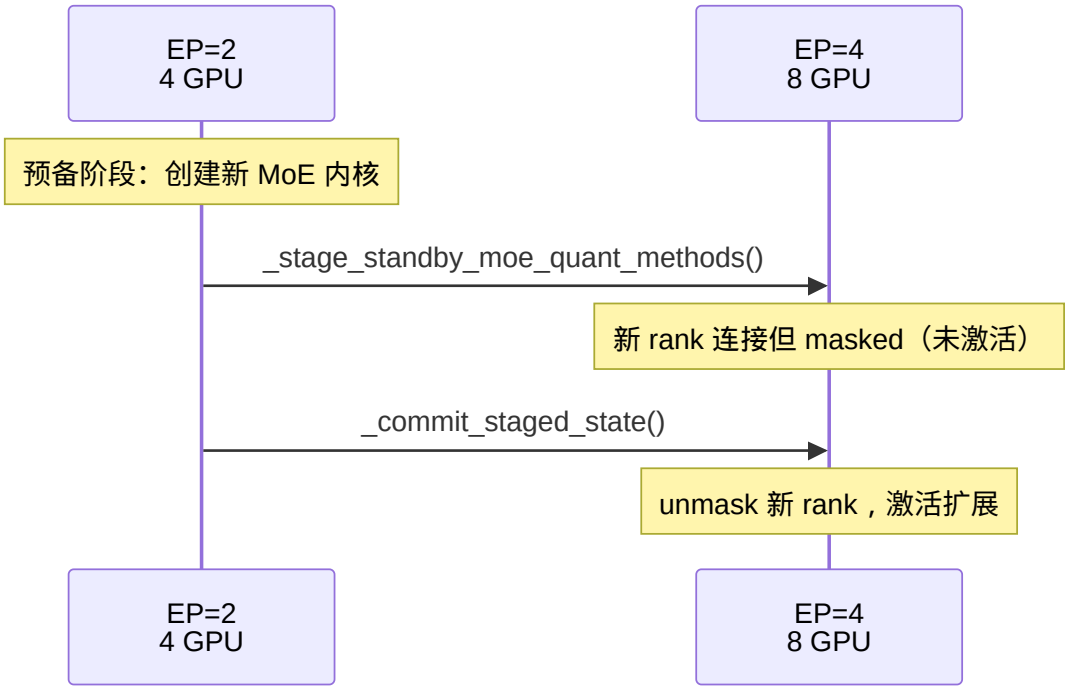
MoE 模型中，将专家分布到不同 GPU：



- 需要 All-to-All 通信：token 发送到对应专家的 GPU
- 负载均衡是关键挑战
- 支持弹性专家并行（动态调整专家分配）

Elastic EP（弹性专家并行）

Elastic EP 允许在线扩展/缩减 EP 大小而不中断服务：



关键机制：

- **Staged** 量化：在 scale-up 前预创建替换 MoE 内核
- **NIXL EP All2All**：新 rank 先以 masked 状态加入集体通信，commit 时 unmask
- 即时参与：新 rank 可以立即加入 EP 集体通信

NIXL EP All2All 重构

`NixlEPAll2AllManager` 引入了分阶段生命周期：

1. `_stage_ep_size()`：连接备用 rank，保持 masked（不活跃）
2. `_commit_staged_state()`：unmask 新 rank，断开旧连接
3. `_NixlEPBufferState`：替代旧的 `(buffer, ep_size)` 元组，管理缓冲区状态
4. 支持跨异构 MoE 层的 workspace 增长

并行策略组合

组合	适用场景	通信量
TP=2	单节点 2 GPU	中等
TP=4	单节点 4/8 GPU	较高
TP=8 + PP=2	双节点 16 GPU	中等
TP=8 + EP=8	MoE 模型 64 GPU	中高
DP=4	高并发小模型	无

分布式 KV 缓存传输

vLLM 支持 KV 缓存在 GPU 间传输：

- **Disaggregated Prefill**：prefill 和 decode 在不同 GPU 上执行
- **KV Transfer**：通过 NCCL 或自定义传输层移动 KV 缓存
- **KV Events**：通过事件通知机制协调 KV 缓存的生产和消费

相关概念

- Tensor Parallelism — 张量并行详解
- Pipeline Parallelism — 流水线并行详解
- MoE — 专家混合模型与专家并行
- KV Cache — 分布式 KV 缓存传输

反向链接

学习日志

分布式计算 — 练习

练习 **1**：并行策略选择

为以下模型和硬件配置选择最佳并行策略：

1. LLaMA-70B，4× A100-80GB，在线服务
2. LLaMA-405B，2 节点 × 8× H100，在线服务
3. DeepSeek-V3 (671B MoE)，8 节点 × 8× H100
4. LLaMA-7B，8× A100-40GB，高并发批量推理

参考答案

1. **TP=4**：70B FP16 约 140GB，4×80GB = 320GB，足够。TP=4 在单节点内 NVLink 通信效率高。
2. **TP=8 + PP=2**：405B FP16 约 810GB。TP=8 在节点内，PP=2 跨节点。每节点需存储约 405GB（一半模型），8×80GB=640GB 够用。
3. **TP=8 + EP=64**（或其他组合）：DeepSeek-V3 是 MoE 模型，256 个路由专家。EP 将专家分布到所有 GPU。TP=8 处理共享参数。
4. **DP=8**：7B 模型单卡可运行。DP=8 可以同时处理 8 倍的请求，最大化吞吐。

练习 **2**：通信量分析

分析 TP=4 的通信量：

- 模型 hidden_size = 8192
- 32 层 Transformer
- FP16

1. 每层的 All-Reduce 通信量是多少？
2. 单步推理的总通信量？
3. 如果使用 NVLink（600 GB/s），通信延迟是多少？

<details>
<summary>参考答案</summary>

1. 每个 All-Reduce：hidden_size × 2 bytes (FP16) = 16384 bytes。Ring All-Reduce 需要 2 × (N-1)/N × data_size ≈ 2 × 3/4 × 16384 = 24576 bytes ≈ 24 KB。每层 2 次 All-Reduce ≈ 48 KB。
2. 32 层 × 48 KB = 1536 KB ≈ 1.5 MB per step。
3. 1.5 MB / 600 GB/s ≈ 2.5 μs。实际延迟还包括 kernel launch 和同步开销，约 10-50 μs。

</details>

拓展挑战

- 阅读 `vllm/distributed/parallel_state.py`，理解进程组管理
- 分析 `kv_transfer/` 目录的 KV 缓存传输实现
- 研究 Elastic EP（弹性专家并行）的动态负载均衡

分布式计算

> 深入 vLLM 的分布式推理支持：张量并行、流水线并行、数据并行和专家并行。

涵盖内容

章节	核心主题
概念	TP、PP、DP、EP 原理与实现
练习	并行策略选择、通信分析
代码走读	parallel_state、通信后端关键代码

核心概念

vLLM 支持多种并行策略来处理大模型推理：

- Tensor Parallelism（**TP**）：将模型权重切分到多个 GPU
- Pipeline Parallelism（**PP**）：将模型层切分到多个 GPU
- **Data Parallelism**（**DP**）：复制模型处理不同请求
- **Expert Parallelism**（**EP**）：将 MoE 专家分布到不同 GPU

前置知识

- 执行器与Worker
- GPU 间通信（NCCL）基础
- 分布式系统基本概念

学习路径

读完本主题后，你将理解：

- 各种并行策略的原理和适用场景
- vLLM 的 parallel_state 如何管理进程组
- NCCL 通信集合（all-reduce、all-gather）在推理中的使用
- 分布式 KV 缓存传输

→ 下一步：API服务与部署

反向链接

API服务与部署 — 概念

引擎核心 — 代码走读

EngineCore 初始化 — `vllm/v1/engine/core.py`

```
# 简化的初始化流程
class EngineCore:
    def __init__(self, vllm_config: VllmConfig):
        # 1. 创建调度器
        self.scheduler = Scheduler(
            scheduler_config=vllm_config.scheduler_config,
            model_config=vllm_config.model_config,
            cache_config=vllm_config.cache_config,
        )

        # 2. 创建执行器
        self.executor = Executor.create(
            vllm_config=vllm_config,
            # 根据 parallel_config 选择 MultiProc/Ray/UniProc
        )
```

主循环

```
def run_loop(self):
    while True:
        # 1. 接收新请求和 abort 信号
        new_requests = self._recv_from_client()

        # 2. 添加新请求到调度队列
        for req in new_requests:
            self.scheduler.add_request(req)

        # 3. 调度决策
        scheduler_output = self.scheduler.schedule()

        # 4. 执行模型推理
        engine_core_outputs = self.executor.execute_model(scheduler_output)

        # 5. 发送输出给前端
        self._send_to_client(engine_core_outputs)
```

AsyncLLM — `vllm/v1/engine/async_llm.py`

请求提交流程

```
class AsyncLLM:
    async def generate(self, prompt, sampling_params, ...):
        # 1. 预处理输入
        processed_inputs = self.input_processor.preprocess(prompt)

        # 2. 创建请求
        request = EngineCoreRequest(
            request_id=request_id,
            prompt_token_ids=processed_inputs.token_ids,
            sampling_params=sampling_params,
        )

        # 3. 发送到 EngineCore
        await self.engine_core_client.submit_request(request)

        # 4. 流式返回输出
        async for output in self._output_stream(request_id):
            yield output
```

输出处理

```
class OutputProcessor:
    def process_outputs(self, engine_core_outputs):
        for output in engine_core_outputs:
            # 1. Detokenize: token IDs → text
            text = self.detokenizer.decode(output.new_token_ids)

            # 2. 检查是否完成
            if output.finish_reason is not None:
                # 最终输出
                yield RequestOutput(
                    outputs=[CompletionOutput(text=text, ...)],
                    finished=True,
                )
            else:
                # 流式中间输出
                yield RequestOutput(
                    outputs=[CompletionOutput(text=text, ...)],
                    finished=False,
                )
```

EngineCoreClient — `vllm/v1/engine/core_client.py`

异步模式初始化

```
class EngineCoreClient:
    @staticmethod
    def make_async(vllm_config):
        # 1. 创建 ZMQ context
        ctx = zmq.asyncio.Context()

        # 2. 创建 ROUTER socket（客户端 bind，服务端 connect）
        socket = ctx.socket(zmq.ROUTER)
        socket.bind(f"ipc://{socket_path}")

        # 3. 启动 EngineCore 进程
        proc = multiprocessing.Process(
            target=EngineCore.run_engine_core,
            args=(vllm_config,),
        )
        proc.start()

        return AsyncEngineCoreClient(socket)
```

请求提交通道

```
async def submit_request(self, request):
    # msgspec 二进制序列化
    data = msgspec.msgpack.encode(request)
    await self.socket.send(data)
```

Detokenizer — `vllm/v1/engine/detokenizer.py`

Detokenizer 负责将 token IDs 转换回文本：

```python
class Detokenizer:
    def decode(self, new_token_ids, request_state):
        # 增量解码：只处理新增的 token
        new_text = self.tokenizer.decode(
            request_state.all_token_ids + new_token_ids,
            skip_special_tokens=True,
        )
        # 返回增量文本（去掉已解码的前缀）
        incremental_text = new_text[len(request_state.decoded_text):]
        return incremental_text
```

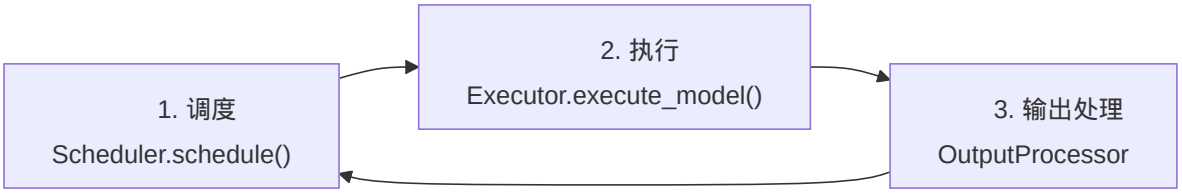
关键函数索引

函数/类	文件	职责
`EngineCore.__init__()`	v1/engine/core.py	初始化调度器和执行器
`EngineCore.run_loop()`	v1/engine/core.py	主循环：接收→调度→执行→输出
`AsyncLLM.generate()`	v1/engine/async_llm.py	异步推理入口
`AsyncLLM._run_output_handler()`	v1/engine/async_llm.py	输出处理循环
`EngineCoreClient.make_async()`	v1/engine/core_client.py	创建异步 IPC 客户端
`InputProcessor.preprocess()`	v1/engine/input_processor.py	tokenize 和预处理
`OutputProcessor.process_outputs()`	v1/engine/output_processor.py	模型输出转换
`Detokenizer.decode()`	v1/engine/detokenizer.py	增量 detokenize

引擎核心 — 概念

EngineCore 主循环

EngineCore 的核心是一个无限循环，每轮迭代执行三个阶段：



阶段 **1**：调度

调度器 分析当前所有待处理请求，决定：

- 哪些新请求开始 prefill
- 哪些请求继续 decode
- 是否需要 preemption（抢占）
- KV 缓存块的分配与释放

阶段 **2**：执行

Executor 接收 `SchedulerOutput`，分发到 Worker 执行模型前向传播。MultiProc 模式下通过 pipe 通信，Ray 模式下通过 Ray actor 调用。

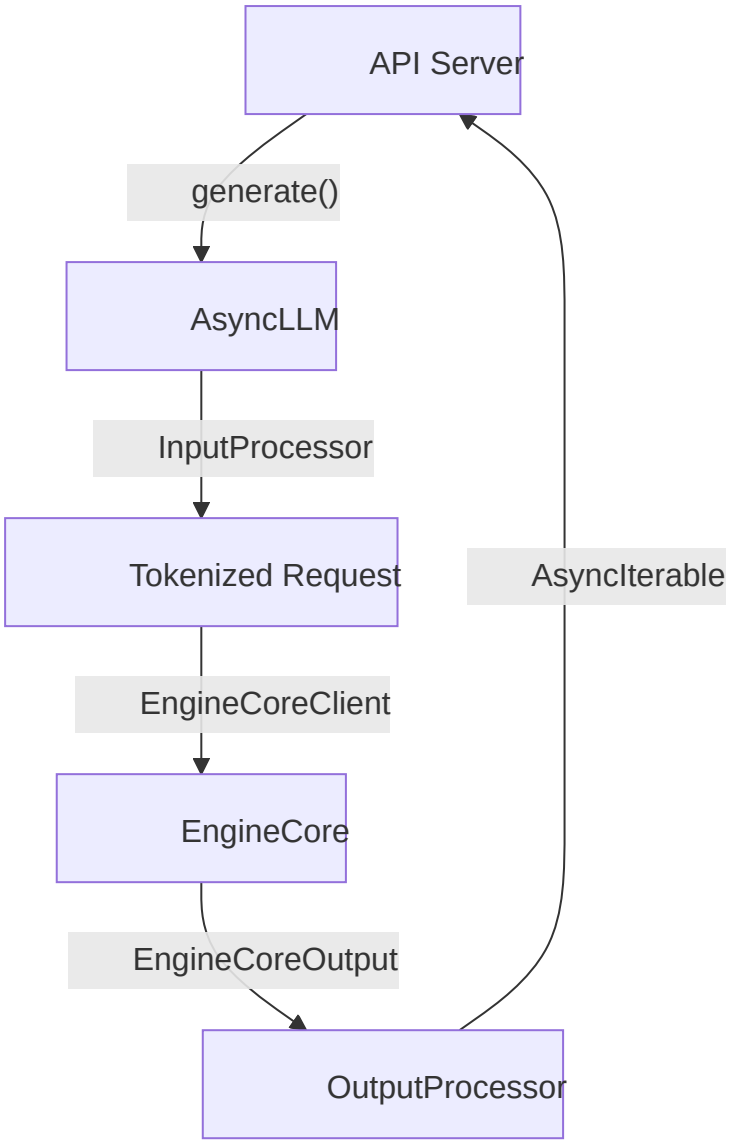
阶段 **3**：输出处理

`OutputProcessor` 将原始模型输出转换为用户可见的输出：

- logits → token IDs（通过采样）
- token IDs → 文本（通过 Detokenizer）
- 处理 finish reason（STOP、LENGTH、ABORT）
- 计算 logprobs 和 prompt logprobs

AsyncLLM 异步前端

AsyncLLM 是面向 API 服务器的异步接口：



关键职责：

- 输入处理：tokenize prompt，处理多模态输入，应用 chat template
- 输出处理：detokenize，流式输出，统计信息收集
- 请求管理：跟踪活跃请求，处理 abort 信号
- 指标收集：TTFT、TPOT、吞吐量等性能指标

EngineCoreClient 通信机制

EngineCoreClient 提供两种通信模式：

同步模式（**SYNC**）

用于 `LLMEngine`（离线推理）：

- 直接函数调用，EngineCore 在同进程中运行
- 适合批量推理场景

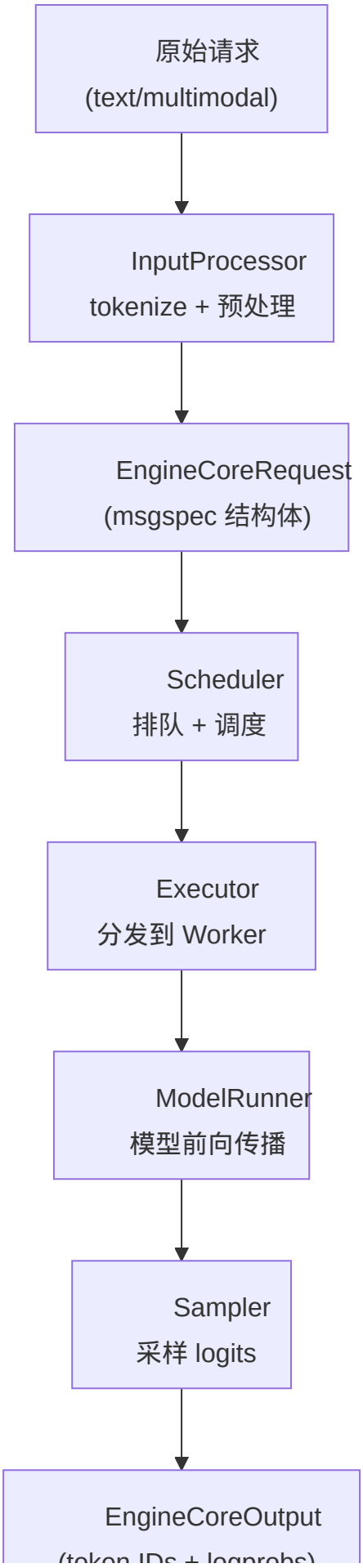
异步模式（**ASYNC**）

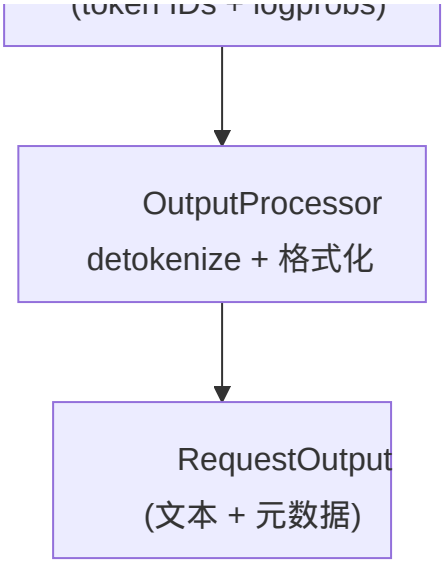
用于 `AsyncLLM`（在线服务）：

- 通过 ZMQ `DEALER/ROUTER` socket 通信
- EngineCore 运行在独立进程
- 支持多前端同时连接

请求处理流水线

一个请求经过的完整处理链：


原始请求
(text/multimodal)
InputProcessor
tokenize + 预处理
EngineCoreRequest
(msgspec 结构体)
Scheduler
排队 + 调度
Executor
分发到 Worker
ModelRunner
模型前向传播
Sampler
采样 logits
EngineCoreOutput



关键数据结构

EngineCoreRequest

使用 msgspec 定义的高效二进制序列化结构体：

- `prompt_token_ids`：tokenized 输入
- `sampling_params`：采样参数（temperature、top-k 等）
- `lora_request`：LoRA 适配器（可选）
- `arrival_time`：请求到达时间（用于计算 TTFT）

EngineCoreOutput

每轮迭代的输出：

- `new_token_ids`：新生成的 token IDs
- `finish_reason`：STOP / LENGTH / ABORT / ERROR
- `logprobs`：token 概率
- `events`：生命周期事件

FinishReason 枚举

值	含义
STOP	遇到 EOS token 或 stop token
LENGTH	达到 max_tokens 限制
ABORT	用户主动取消
ERROR	推理过程中出错
REPETITION	重复惩罚触发

相关概念

- Continuous Batching — 调度器的核心策略
- Paged Attention — KV 缓存的内存管理
- CUDA Graph — decode 阶段的图优化
- Flash Attention — 高效注意力计算内核
- 调度系统 — 调度器的详细分析
- API服务 — Rust Frontend 和 DP Supervisor 的前端模式

出链

/topics/scheduling/

/topics/serving/

引擎核心 — 练习

练习 1：EngineCore 主循环追踪

阅读 `vllm/v1/engine/core.py` 的 `run_loop` 方法，回答：

1. 主循环在什么条件下退出？
2. 每轮迭代如何处理新到达的请求？
3. 如果没有活跃请求，主循环如何等待？

<details>
<summary>参考答案</summary>

1. 主循环在收到 shutdown 信号时退出。通过 ZMQ 的 poll 机制检测。
2. 每轮迭代开始时，从 ZMQ socket 读取所有待处理消息，包括新请求、abort 信号和 utility calls。
3. 使用 `zmq_poller.poll(timeout)` 等待新消息。如果没有活跃请求且没有待调度请求，主循环会阻塞等待直到新请求到达。

</details>

练习 2：AsyncLLM 输出流分析

分析 `AsyncLLM.generate()` 如何实现流式输出：

1. `generate()` 方法返回什么类型？
2. 如何实现从 EngineCore 的批量输出到单个请求流式输出的映射？
3. abort 请求如何传播到 EngineCore？

参考答案

1. 返回 `AsyncIterator[RequestOutput]`，每次 yield 一个新的输出 chunk。
2. EngineCore 每轮迭代返回所有请求的输出。OutputProcessor 按请求 ID 分发到对应的 stream。每个请求有自己的 `AsyncQueue`，generate() 从中消费。
3. 调用 `async_engine.abort(request_id)`，通过 EngineCoreClient 发送 abort 消息到 EngineCore。EngineCore 在下一轮迭代开始时处理 abort 信号，从调度队列中移除请求。

练习 3：IPC 性能分析

分析 ZMQ IPC 的性能特征：

1. EngineCoreClient 使用哪种 ZMQ socket 类型？
2. 序列化格式是什么？为什么选择这种格式？
3. 大批量输出时如何避免内存拷贝？

参考答案

1. 使用 `ZMQ_DEALER`（客户端）和 `ZMQ_ROUTER`（服务端）模式，支持多对一通信。
2. 使用 msgspec 进行二进制序列化，比 JSON 更紧凑高效。msgspec 的 Message 编码支持零拷贝反序列化。
3. ZMQ 支持零拷贝发送（`send(copy=False)`），大数组可以直接传递内存指针而不需要复制。对于 token IDs 数组，使用 numpy array 或 bytes buffer 直接传递。

拓展挑战

- 在 vLLM 源码中添加自定义日志，追踪一个请求从入口到返回的完整时间线
- 分析 `InputProcessor` 如何处理多模态输入（图像、音频）

- 研究 `ParallelSampling`（并行采样）如何在 EngineCore 中实现多个 completion

引擎核心

> 深入 vLLM V1 架构的 EngineCore、AsyncLLM 和 EngineCoreClient，理解引擎如何编排调度与执行。

涵盖内容

章节	核心主题
概念	EngineCore 主循环、ZMQ IPC、请求处理流水线
练习	引擎行为分析、性能调优实验
代码走读	core.py、async_llm.py、core_client.py 关键代码

核心概念

topics/engine-core/ 是 vLLM 的中央编排器，运行在独立进程中：

- 通过 ZMQ 与前端通信
- 每轮迭代调用 Scheduler 进行调度决策
- 通过 Executor 分发执行到 Worker
- 管理请求的完整生命周期

前置知识

- 架构概览
- ZMQ 进程间通信基础
- Python multiprocessing 基础

学习路径

读完本主题后，你将理解：

- EngineCore 的主循环如何驱动整个推理流程
- AsyncLLM 如何桥接异步 API 与 EngineCore
- EngineCoreClient 的同步/异步通信模式
- 输入处理（tokenize）和输出处理（detokenize）的流水线

→ 下一步：调度系统

反向链接

架构概览 — 概念

架构概览

引擎核心

执行器与Worker

调度系统

执行器与Worker — 代码走读

Executor 抽象 — `vllm/v1/executor/abstract.py`

```
class Executor(ABC):
    @abstractmethod
    def determine_available_memory(self) -> int:
        """返回可用 KV 缓存显存"""
        ...

    @abstractmethod
    def initialize(self):
        """初始化 executor 和 workers"""
        ...

    @abstractmethod
    def execute_model(self, scheduler_output) -> list[EngineCoreOutput]:
        """执行一轮模型推理"""
        ...

    @staticmethod
    def create(vllm_config):
        """工厂方法，根据配置选择 executor 类型"""
        parallel_config = vllm_config.parallel_config
        if parallel_config.use_ray:
            return RayExecutor(vllm_config)
        elif parallel_config.world_size > 1:
            return MultiProcExecutor(vllm_config)
        else:
            return UniProcExecutor(vllm_config)
```

MultiProcExecutor — `vllm/v1/executor/multiproc_executor.py`

```
class MultiProcExecutor:
    def __init__(self, vllm_config):
        self.workers = []
        for rank in range(vllm_config.parallel_config.world_size):
            # 创建 Worker 进程
            worker = WorkerProc(
                rank=rank,
                vllm_config=vllm_config,
            )
            self.workers.append(worker)

    def execute_model(self, scheduler_output):
        # 广播调度输出到所有 Worker
        for worker in self.workers:
            worker.send(scheduler_output)

        # 等待所有 Worker 完成
        outputs = []
        for worker in self.workers:
            output = worker.recv()
            outputs.append(output)

        return outputs[0]  # rank 0 的输出
```

GPU Worker — `vllm/v1/worker/gpu_worker.py`

```
class GPUWorker:
    def init_device(self):
        torch.cuda.set_device(self.device)
        torch.cuda.empty_cache()

    def load_model(self):
        self.model_runner = ModelRunner(self.vllm_config)
        self.model_runner.load_model()

    def profile_run(self):
        # 运行虚拟推理，测量显存使用
        self.model_runner.profile_run()

    def allocate_kv_cache(self, num_blocks):
        self.model_runner.allocate_kv_cache(num_blocks)

    def execute_model(self, scheduler_output):
        return self.model_runner.execute_model(scheduler_output)
```

GPU ModelRunner — `vllm/v1/worker/gpu_model_runner.py`

这是 vLLM 最大的单个文件（324K），核心方法：

```
class ModelRunner:
    def execute_model(self, scheduler_output):
        # 1. 构建输入
        input_batch = self._prepare_inputs(scheduler_output)

        # 2. 检查是否可以使用 CUDA Graph
        if self._can_use_cudagraph(input_batch):
            return self._execute_with_cudagraph(input_batch)

        # 3. 普通执行
        return self._execute_model_native(input_batch)
```

CUDA Graph 执行

```python
    def _execute_with_cudagraph(self, input_batch):
        batch_size = input_batch.num_seqs

        # 获取对应 batch size 的已捕获图
        graph = self.cudagraphs[batch_size]

        # 更新图的输入缓冲区
        graph.input_buffer.copy_(input_batch.to_tensor())

        # 重放图
        graph.replay()

        # 读取输出
        return graph.output_buffer
```

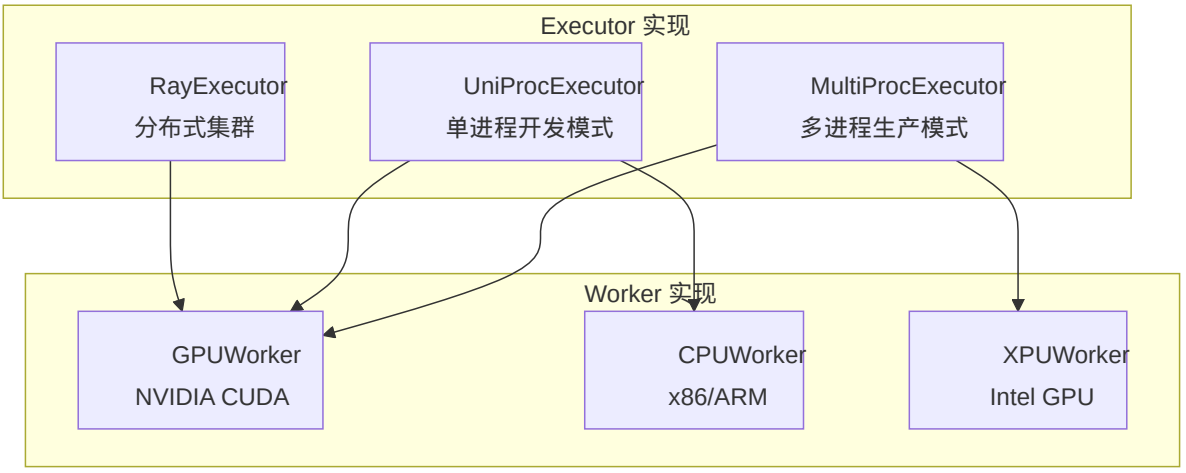
关键函数索引

函数/类	文件	职责
`Executor.create()`	v1/executor/abstract.py	工厂方法选择执行器
`MultiProcExecutor.execute_model()`	v1/executor/multiproc_executor.py	多进程模型执行
`GPUWorker.init_device()`	v1/worker/gpu_worker.py	CUDA 设备初始化
`GPUWorker.load_model()`	v1/worker/gpu_worker.py	模型加载入口
`ModelRunner.execute_model()`	v1/worker/gpu_model_runner.py	GPU 前向传播
`ModelRunner._prepare_inputs()`	v1/worker/gpu_model_runner.py	输入批处理
`ModelRunner._execute_with_cudagraph()`	v1/worker/gpu_model_runner.py	CUDA Graph 重放

执行器与Worker — 概念

Executor 抽象层

Executor 定义了模型执行的抽象接口，支持三种部署模式：



UniProcExecutor

单进程执行器，所有操作在同一进程中完成：

- 适合开发和调试
- 没有 IPC 开销
- 不受 GIL 影响的 CUDA 操作仍然可以并行

MultiProcExecutor

多进程执行器，每个 GPU 对应一个 Worker 进程：

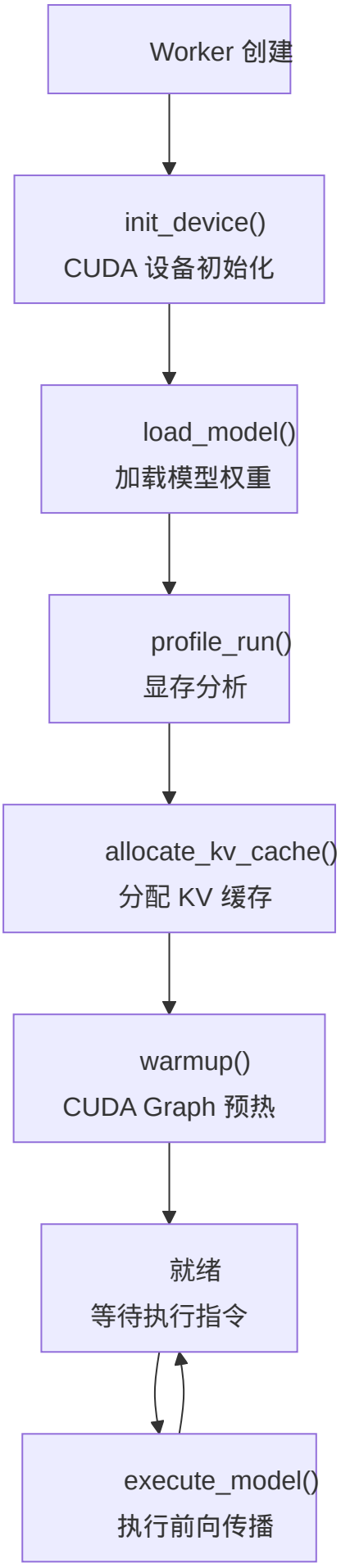
- 通过 `multiprocessing.Process` 创建 Worker
- 使用 pipe 进行进程间通信
- 支持 张量并行

RayExecutor

基于 Ray 的分布式执行器：

- 支持跨节点部署
- 适合大规模集群推理
- 支持 流水线并行

Worker 生命周期


Worker 创建
init_device()
CUDA 设备初始化
load_model()
加载模型权重
profile_run()
显存分析
allocate_kv_cache()
分配 KV 缓存
warmup()
CUDA Graph 预热
就绪
等待执行指令
execute_model()
执行前向传播

关键步骤

1. **init_device()**：设置 CUDA 设备，初始化 NCCL 通信组
2. **load_model()**：从 HuggingFace checkpoint 加载权重到 GPU
3. **profile_run()**：运行一次虚拟前向传播，测量实际显存使用量
4. **allocate_kv_cache()**：根据剩余显存分配 KV 缓存块
5. **warmup()**：捕获 CUDA Graph，预热 CUDA kernel

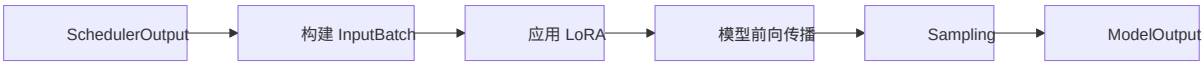
CuMemAllocator

`CuMemAllocator` 现在作为独立功能（`enable_cumem_allocator`），不再与 `enable_sleep_mode` 耦合。它管理 CUDA 内存池，允许在 sleep/pause 模式下释放和重新分配显存。

ModelRunner

ModelRunner 是 Worker 的核心组件（vllm/v1/worker/gpu_model_runner.py，约 7400 行 / 324KB），负责：

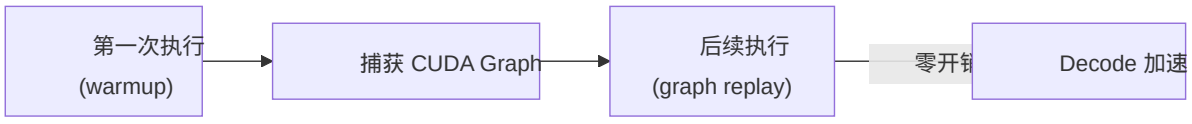
前向传播执行



CUDA Graph 管理

CUDA Graph 在 decode 阶段提供显著的性能提升：

1. 预热阶段：用小批量输入执行一次前向传播
2. 捕获图：记录所有 CUDA 操作为图
3. 重放：用新输入重放图，避免 kernel launch 开销



输入批处理

ModelRunner 将调度器的输出转换为 GPU 可执行的输入：

```
# 简化示意
class InputBatch:
    token_ids: torch.Tensor      # [num_seqs, max_len]
    position_ids: torch.Tensor    # [num_seqs, max_len]
    block_tables: torch.Tensor    # [num_seqs, max_blocks]
    seq_lens: torch.Tensor        # [num_seqs]
    sampling_params: list[SamplingParams]
```

硬件平台支持

vLLM 通过 Platform 抽象支持多种硬件：

平台	文件	**Worker**
NVIDIA CUDA	platforms/cuda.py	GPUWorker
AMD ROCm	platforms/rocm.py	GPUWorker
x86/ARM CPU	platforms/cpu.py	CPUWorker
Intel XPU/Gaudi	platforms/xpu.py	XPUWorker
Google TPU	platforms/tpu.py	TPUWorker

相关概念

- CUDA Graph — CUDA 图捕获与重放
- Tensor Parallelism — 张量并行
- Pipeline Parallelism — 流水线并行
- Flash Attention — 高效注意力内核
- 模型库与算子层 — 模型执行的具体实现

反向链接　　出链

执行器与Worker — 练习

练习 1：Worker 初始化顺序

在多 GPU 场景中（4 × A100），分析 MultiProcExecutor 的初始化过程：

1. 创建了多少个 Worker 进程？
2. 每个进程如何知道使用哪个 GPU？
3. NCCL 通信组如何建立？

> 参考答案
>
> 1. 创建 4 个 Worker 进程，每个 GPU 对应一个。
> 2. 通过 `CUDA_VISIBLE_DEVICES` 环境变量或 `torch.cuda.set_device(local_rank)` 设置。
> 3. 每个 Worker 调用 `torch.distributed.init_process_group(backend='nccl')`，通过 rendezvous（如文件系统或 TCP）协调。TP=4 时所有 4 个 Worker 在同一进程组中。

练习 2：CUDA Graph 捕获分析

分析 CUDA Graph 捕获的条件和限制：

1. 为什么 CUDA Graph 只在 decode 阶段使用？
2. 捕获时输入形状如何确定？
3. 如果 batch size 超过捕获时的形状会怎样？

参考答案

1. Prefill 阶段的输入形状是动态的（不同 prompt 长度），无法用固定图捕获。Decode 阶段每步只生成 1 个 token，输入形状固定，适合图捕获。
2. vLLM 会为多个 batch size 分别捕获图（如 1, 2, 4, 8, ..., max_num_seqs）。运行时选择最接近的已捕获 batch size 的图。
3. 如果 batch size 超过最大捕获的图，会回退到普通执行（非图模式）。这也是为什么 vLLM 需要在 warmup 阶段为多个 batch size 捕获图。

练习 3：显存分配分析

在一个 80GB A100 上运行 7B 模型（FP16），分析显存分配：

1. 模型参数占用多少显存？
2. profiling 后剩余多少给 KV 缓存？
3. 如果 block_size=16，能分配多少 block？

参考答案

1. 7B 参数 × 2 bytes (FP16) = 14 GB。加上优化器状态和梯度（推理不需要），实际约 14 GB。
2. 剩余约 80 - 14 = 66 GB（减去 CUDA context 和其他开销，实际约 60-65 GB）。
3. 每个 block 的 KV 缓存大小 = block_size × head_dim × num_kv_heads × 2 (K+V) × 2 (FP16) bytes。假设 LLaMA-7B (32 heads, head_dim=128, 32 layers): 16 × 128 × 32 × 2 × 2 × 32 layers ≈ 8.4 MB/block。约 60 GB / 8.4 MB ≈ 7142 blocks。

拓展挑战

- 阅读 `gpu_model_runner.py` 的 `_prepare_inputs` 方法，理解输入批处理的详细逻辑

- 分析 `cudagraph_utils.py` 中 CUDA Graph 捕获的底层实现
- 研究 RayExecutor 如何在多节点上调度 Worker

执行器与Worker

> 深入 vLLM 的执行层：Executor 如何调度 Worker，GPU Worker 和 ModelRunner 如何管理模型加载、前向传播和 CUDA Graph。

涵盖内容

章节	核心主题
概念	Executor 模式、Worker 生命周期、ModelRunner、CUDA Graph
练习	Worker 初始化流程、CUDA Graph 分析
代码走读	executor、worker、model_runner 关键代码

核心概念

执行层是 vLLM 中直接与硬件交互的层次：

- **Executor**：抽象执行接口，支持 MultiProc、Ray、UniProc 三种模式
- **Worker**：管理单个设备的初始化、模型加载、推理执行
- **ModelRunner**：执行模型前向传播，管理 CUDA Graph、LoRA、推测解码

前置知识

- 引擎核心
- 调度系统
- CUDA 编程基础概念

学习路径

读完本主题后，你将理解：

- Executor 如何选择和创建合适的 Worker
- GPU Worker 的初始化和模型加载流程
- ModelRunner 如何执行前向传播并管理批处理
- CUDA Graph 如何加速 decode 阶段

→ 下一步：模型库与算子层

反向链接

分布式计算

模型库与算子层

KV缓存与PagedAttention — 代码走读

KVCacheManager — `vllm/v1/core/kv_cache_manager.py`

初始化

```
    class KVCacheManager:
        def __init__(self, block_size, num_blocks, ...):
            self.block_size = block_size
            self.block_pool = BlockPool(num_blocks)
            self.block_hashes: dict[BlockHash, KVCacheBlock] = {}
            self.req_to_blocks: dict[RequestID, list[KVCacheBlock]] = {}
```

块分配流程

```
    def allocate(self, request, num_tokens):
        # 1. 计算需要的块数
        total_blocks_needed = cdiv(
            request.num_computed_tokens + num_tokens,
            self.block_size,
        )
        current_blocks = self.req_to_blocks[request.request_id]
        num_new_blocks = total_blocks_needed - len(current_blocks)

        # 2. 前缀缓存检查
        new_blocks = []
        for i in range(num_new_blocks):
            block_idx = len(current_blocks) + i
            # 计算块内容的哈希
            block_hash = self._compute_block_hash(request, block_idx)

            # 检查是否有可复用的块
            if block_hash in self.block_hashes:
                cached_block = self.block_hashes[block_hash]
                cached_block.ref_count += 1
                new_blocks.append(cached_block)
            else:
                # 从空闲池分配
                block = self.block_pool.allocate()
                block.hash = block_hash
                self.block_hashes[block_hash] = block
                new_blocks.append(block)

        current_blocks.extend(new_blocks)
        return new_blocks
```

块释放

```
    def free(self, request):
        blocks = self.req_to_blocks.pop(request.request_id)
        for block in blocks:
            block.ref_count -= 1
            if block.ref_count == 0:
                # 引用归零，归还空闲池
                del self.block_hashes[block.hash]
                self.block_pool.free(block)
```

BlockPool — `vllm/v1/core/block_pool.py`

```
class BlockPool:
    """空闲块池，管理所有可用的物理块"""

    def __init__(self, num_blocks):
        self.blocks = [KVCacheBlock(id=i) for i in range(num_blocks)]
        self.free_block_indices = set(range(num_blocks))

    @property
    def num_free_blocks(self):
        return len(self.free_block_indices)

    def allocate(self):
        if not self.free_block_indices:
            return None
        block_id = self.free_block_indices.pop()
        return self.blocks[block_id]

    def free(self, block):
        block.reset()
        self.free_block_indices.add(block.id)
```

KV Cache Utils — `vllm/v1/core/kv_cache_utils.py`

块哈希计算

```
class BlockHash:
    """块的哈希值，用于前缀缓存匹配"""
    # 结合块内容和父块哈希的链式哈希
    pass

def compute_block_hash(content_hash, parent_hash, block_idx):
    return hash((content_hash, parent_hash, block_idx))
```

KV Cache 配置生成

```python
def generate_kv_cache_config(
    model_config, cache_config, parallel_config
) -> KVCacheConfig:
    # 计算每层的 KV 缓存大小
    head_dim = model_config.get_head_dim()
    num_layers = model_config.get_num_layers()
    dtype_size = get_dtype_size(cache_config.cache_dtype)

    # 每个 block 的字节数
    block_size_bytes = (
        block_size * head_dim * num_kv_heads * dtype_size * 2  # K + V
    )

    # 总可用显存
    available_memory = get_available_gpu_memory()

    # 计算可用块数
    num_blocks = available_memory // block_size_bytes

    return KVCacheConfig(
        block_size=block_size,
        num_blocks=num_blocks,
    )
```

KV Cache Coordinator — `vllm/v1/core/kv_cache_coordinator.py`

协调多个缓存组（例如 sliding window + full attention）：

```python
class KVCacheCoordinator:
    def __init__(self, managers: list[KVCacheManager]):
        self.managers = managers

    def allocate(self, request, num_tokens):
        for manager in self.managers:
            blocks = manager.allocate(request, num_tokens)
            if blocks is None:
                # 任一管理器分配失败则全部回滚
                self._rollback_allocations()
                return None
        return True
```

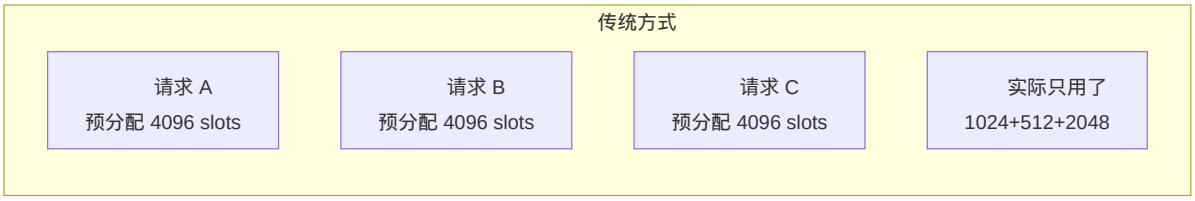
关键函数索引

函数/类	文件	职责
`KVCacheManager.allocate()`	v1/core/kv_cache_manager.py	分配 KV 缓存块
`KVCacheManager.free()`	v1/core/kv_cache_manager.py	释放请求的缓存块
`BlockPool.allocate()`	v1/core/block_pool.py	从空闲池分配物理块
`generate_kv_cache_config()`	v1/core/kv_cache_utils.py	计算缓存配置参数
`compute_block_hash()`	v1/core/kv_cache_utils.py	计算块哈希（前缀缓存）
`KVCacheCoordinator.allocate()`	v1/core/kv_cache_coordinator.py	协调多组缓存分配

KV缓存与PagedAttention — 概念

传统 KV Cache 的问题

在标准 Transformer 推理中，每个请求的 KV 缓存需要预分配最大长度的连续显存：

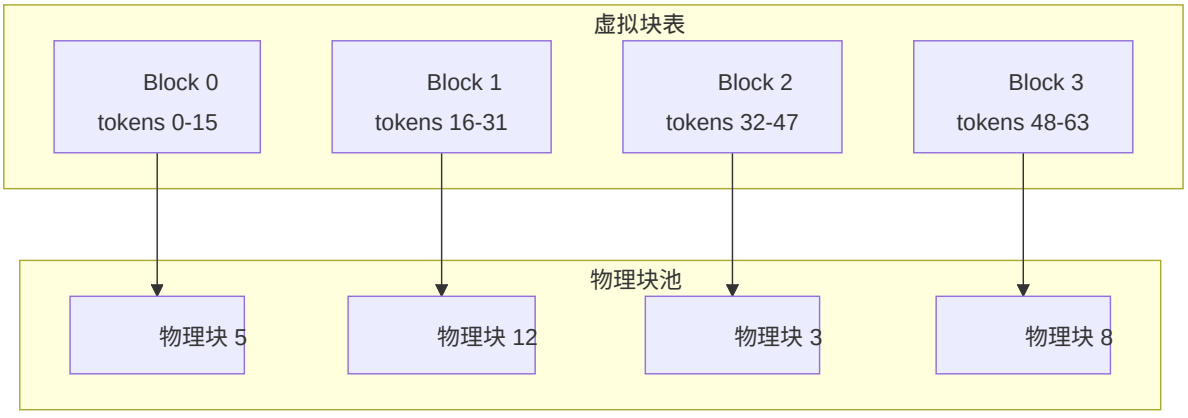


问题：

- 显存浪费：预分配最大长度，实际使用远小于此
- 碎片化：不同请求释放后产生不连续的空闲区域
- 无法共享：相同前缀的请求各自保存独立副本

PagedAttention 原理

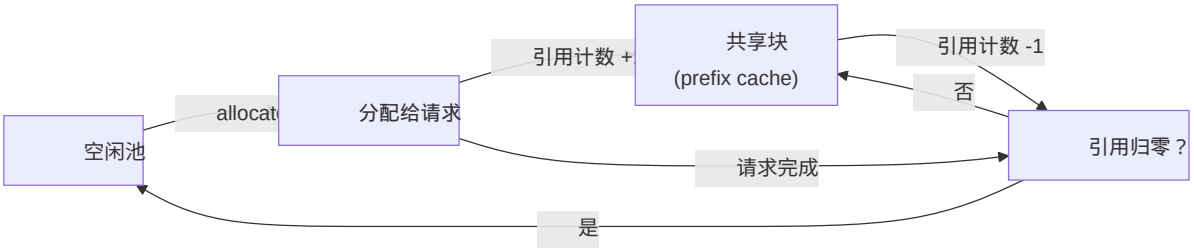
PagedAttention 借鉴操作系统虚拟内存的分页机制：



核心概念

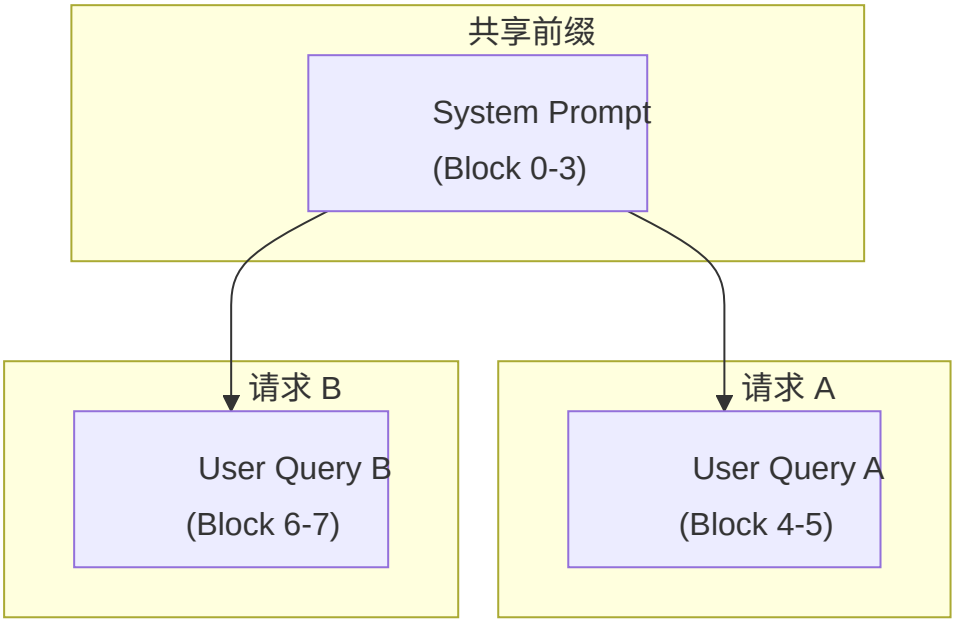
概念	说明
Block（块）	固定大小的 KV 缓存单元，默认 16 tokens
Block Table（块表）	虚拟块到物理块的映射表
Block Pool（块池）	所有可用物理块的集合
Reference Counting（引用计数）	跟踪每个物理块的引用数，支持共享

块的生命周期



前缀缓存（**Prefix Caching**）

Prefix Caching 允许具有相同前缀的请求共享 KV 缓存块：



实现方式：

1. 每个 block 计算内容的哈希值
2. 新请求的 block 哈希与已有 block 匹配

3. 匹配成功则复用已有物理块（引用计数 +1）

4. 不匹配则从空闲池分配新块

哈希计算

```
block_hash = hash(block_content, parent_block_hash)
```

这是链式哈希：每个块的哈希依赖其内容和父块的哈希，确保相同序列位置上的相同内容产生相同哈希。

KV 缓存卸载（KV Cache Offloading）

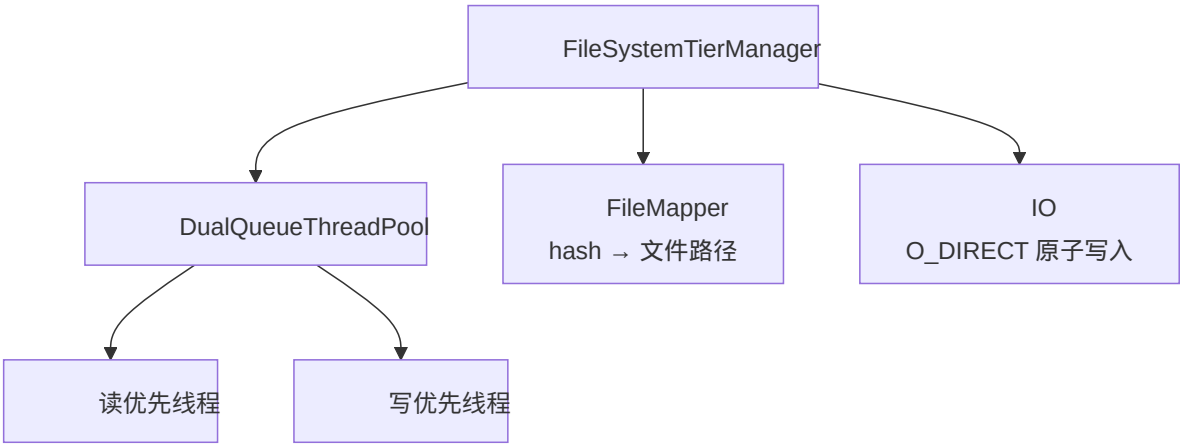
当 GPU 显存不足时，vLLM 支持将 KV 缓存卸载到其他存储层级：

分层策略

层级	延迟	容量	适用场景
GPU HBM	~ns	有限	活跃请求的 KV 缓存
CPU RAM	~us	较大	被抢占的请求
SSD/Disk	~ms	很大	长期不活跃的请求

文件系统分层（**FileSystem Tiering**）

新增的纯 Python 磁盘后端二级缓存层：



关键特性：

- **DualQueueThreadPool**：分离读写队列，读操作优先处理
- 原子写入：先写临时文件，再 `rename` 确保一致性
- **O_DIRECT I/O**：绕过页缓存，减少内存拷贝
- 哈希子目录：将 KV 块按哈希映射到子目录，避免单目录文件过多

MooncakeStore 多组支持

MooncakeStore 连接器现在支持混合注意力模型的多组 KV 缓存：

- **MooncakeStoreCoordinator**：计算每个 KV 缓存组的 load/store mask
- 多 **TokenDatabase**：每组 KV cache 维护独立的 `ChunkedTokenDatabase`
- **LCM** 块对齐：跨异构 KV 缓存组的块对齐（使用最小公倍数）
- 指标子系统：`MooncakeStoreConnectorStats` 跟踪每次操作的延迟、字节数、p90 延迟和错误率

Block 大小选择

block_size 的选择影响性能：

block_size	优点	缺点
小（8）	更精细的显存管理，浪费更少	更多的元数据开销
中（16）	平衡选择（默认）	—
大（32）	更少的元数据开销	更多显存浪费

相关概念

- Paged Attention — 分页注意力的核心算法
- KV Cache — KV 缓存的基本概念
- Continuous Batching — 调度器如何与 KV 缓存管理交互
- Prefix Caching — 前缀缓存的详细原理
- KV Cache Offloading — KV 缓存卸载到 CPU/磁盘

反向链接

学习日志

KV缓存与PagedAttention — 练习

练习 1：块分配模拟

配置：block_size=16，总共 8 个物理块。

请求序列：

1. 请求 A：prompt 40 tokens
2. 请求 B：prompt 24 tokens（前 16 tokens 与 A 相同）
3. 请求 A decode 10 个新 token
4. 请求 C：prompt 48 tokens（前 16 tokens 与 A 相同）

请模拟每步的块分配状态（物理块使用情况、空闲块数、引用计数）。

参考答案

初始：8 个空闲块 [0-7]

1. 请求 **A**（**40 tokens** ）：需要 3 个块（ceil(40/16)=3 ）。分配块 [0,1,2]。空闲：5
   - Block 0: hash(A[0:16]), ref=1
   - Block 1: hash(A[16:32]), ref=1
   - Block 2: hash(A[32:40]), ref=1（部分填充）
2. 请求 **B**（**24 tokens**，前 **16** 与 **A** 相同 ）： 需要 2 个块。
   - Block 0: hash 匹配，复用，ref=2
   - Block 3: 新分配（B[16:24] ），ref=1
   - 空闲：4
3. 请求 **A decode 10 tokens**：现有 40+10=50 tokens，需要 ceil(50/16)=4 个块。
   - Block 2 原来只填到 40，现在填到 48（8 个新 token ）
   - 需要 Block 4（第 49-50 token），新分配，ref=1
   - 空闲：3
4. 请求 **C**（**48 tokens**，前 **16** 与 **A** 相同 ）： 需要 3 个块。
   - Block 0: hash 匹配，复用，ref=3
   - Block 5, 6: 新分配（C[16:48] ），ref=1
   - 空闲：1

练习 **2**：前缀缓存效果分析

假设 system prompt 为 512 tokens，有 10 个并发请求共享此 system prompt。

计算有无前缀缓存时的 KV 缓存块数（ block_size=16 ）：

1. 无前缀缓存
2. 有前缀缓存

参考答案

System prompt 块数：ceil(512/16) = 32 块

1. 无前缀缓存：每个请求独立存储 system prompt。
   - 10 × 32 = 320 块用于 system prompt
   - 总共至少 320 + 10 × (user query blocks)
2. 有前缀缓存：所有请求共享一份 system prompt。
   - 32 块用于 system prompt（ref=10 ）
   - 总共 32 + 10 × (user query blocks)
   - 节省：320 - 32 = 288 块（90% 节省）

练习 3：KV 缓存卸载策略分析

分析以下场景中应该使用哪种卸载策略：

1. 10 个请求并发，GPU 显存刚好够用，但第 11 个请求到达
2. 100 个长对话请求，其中 80% 已进入空闲等待状态
3. 离线批量推理场景，请求按顺序处理

参考答案

1. **Re-computation**：只是暂时的显存压力，抢占一个请求后很快会恢复。重新 prefill 的开销小于 swap 的 I/O 开销。
2. **CPU Swapping**：大量不活跃请求，适合将它们的 KV 缓存换出到 CPU 内存。当用户继续对话时再换入。
3. 不卸载：离线场景中请求按顺序处理，不需要保留已完成请求的 KV 缓存。直接释放即可。

拓展挑战

- 阅读 `vllm/v1/core/kv_cache_utils.py` 中的块哈希计算逻辑
- 分析 `KVCacheCoordinator` 如何管理多个缓存组（sliding window + full attention）
- 研究 `simple_kv_offload/manager.py` 的卸载实现

KV缓存与PagedAttention

> 深入理解 vLLM 的核心创新：PagedAttention 分页 KV 缓存管理，以及相关的 KV cache 块分配、前缀缓存和 KV 缓存卸载。

涵盖内容

章节	核心主题
概念	PagedAttention 原理、块管理、前缀缓存、KV 卸载
练习	块分配模拟、前缀缓存分析
代码走读	KVCacheManager、BlockPool、kv_cache_utils 关键代码

核心概念

Paged Attention 是 vLLM 的核心创新，借鉴操作系统虚拟内存的分页思想管理 KV Cache：

- 将 KV 缓存划分为固定大小的块（block）
- 按需分配块，避免预分配连续大块显存
- 支持共享块：多个请求可以共享相同前缀的 KV 缓存
- 支持块级交换：将不活跃的块换出到 CPU 内存

前置知识

- 调度系统
- Transformer 注意力机制的基本原理
- GPU 显存管理基础

学习路径

读完本主题后，你将理解：

- PagedAttention 如何实现高效的 KV 缓存管理
- 块的分配、释放和共享机制
- Prefix Caching 如何复用共享前缀的 KV 缓存
- KV 缓存卸载到 CPU/磁盘的分层策略

→ 下一步：执行器与Worker

模型库与算子层 — 代码走读

LLaMA 模型实现 — `vllm/model_executor/models/llama.py`

模型结构

```
@ModelRegistry.register("LlamaForCausalLM")
class LlamaForCausalLM(nn.Module):
    def __init__(self, config, *, model_config, cache_config, quant_config):
        self.model = LlamaModel(config, ...)
        self.lm_head = ParallelLMHead(config.vocab_size, ...)

    def forward(self, input_ids, positions, kv_caches, ...):
        hidden = self.model(input_ids, positions, kv_caches)
        return self.lm_head(hidden)

    def load_weights(self, weights):
        # 权重加载逻辑
        ...
```

LlamaModel

```
class LlamaModel(nn.Module):
    def __init__(self, config, ...):
        self.embed_tokens = VocabParallelEmbedding(...)
        self.layers = nn.ModuleList([
            LlamaDecoderLayer(config, ...) for _ in range(config.num_hidden_laye
rs)
        ])
        self.norm = RMSNorm(config.hidden_size, ...)

    def forward(self, input_ids, positions, kv_caches):
        hidden = self.embed_tokens(input_ids)
        for i, layer in enumerate(self.layers):
            hidden = layer(
                hidden, positions,
                kv_caches[i],  # 每层有独立的 KV 缓存
            )
        return self.norm(hidden)
```

LlamaDecoderLayer

```
class LlamaDecoderLayer(nn.Module):
    def __init__(self, config, ...):
        self.self_attn = Attention(...)  # 统一注意力层
        self.mlp = LlamaMLP(...)         # FFN
        self.input_layernorm = RMSNorm(...)
        self.post_attention_layernorm = RMSNorm(...)

    def forward(self, hidden, positions, kv_cache):
        # Pre-norm 架构
        residual = hidden
        hidden = self.input_layernorm(hidden)
        hidden = self.self_attn(hidden, positions, kv_cache)
        hidden = residual + hidden

        residual = hidden
        hidden = self.post_attention_layernorm(hidden)
        hidden = self.mlp(hidden)
        hidden = residual + hidden
        return hidden
```

Attention 层 — `model_executor/layers/attention/`

统一注意力接口

```python
class Attention(nn.Module):
    def __init__(self, num_heads, head_dim, ...):
        self.q_proj = ColumnParallelLinear(...)
        self.k_proj = ColumnParallelLinear(...)
        self.v_proj = ColumnParallelLinear(...)
        self.o_proj = RowParallelLinear(...)

    def forward(self, hidden, positions, kv_cache):
        q = self.q_proj(hidden)
        k = self.k_proj(hidden)
        v = self.v_proj(hidden)

        # 应用 RoPE
        q, k = self.rotary_emb(positions, q, k)

        # PagedAttention：更新 KV 缓存
        k, v = self._update_kv_cache(k, v, kv_cache)

        # 调用注意力后端
        output = self.attn_backend.forward(
            q, k, v, kv_cache.block_table, ...
        )
        return self.o_proj(output)
```

Linear 层 — `model_executor/layers/linear.py`

并行线性层

```python
class ColumnParallelLinear(nn.Module):
    """列切分：输出维度沿 TP rank 切分"""
    def __init__(self, input_size, output_size, ...):
        self.weight = Parameter(output_size // tp_size, input_size)

    def forward(self, x):
        output = F.linear(x, self.weight)
        # All-reduce 在后续 RowParallelLinear 中完成
        return output

class RowParallelLinear(nn.Module):
    """行切分：输入维度沿 TP rank 切分"""
    def __init__(self, input_size, output_size, ...):
        self.weight = Parameter(output_size, input_size // tp_size)

    def forward(self, x):
        output = F.linear(x, self.weight)
        output = tensor_model_parallel_all_reduce(output)
        return output
```

Fused MoE — `model_executor/layers/fused_moe/`

MoE 层实现

```python
class FusedMoE(nn.Module):
    def __init__(self, num_experts, top_k, ...):
        self.w13 = Parameter(num_experts, 2 * intermediate_size, hidden_size)
        self.w2 = Parameter(num_experts, hidden_size, intermediate_size)
        self.router = nn.Linear(hidden_size, num_experts)

    def forward(self, hidden):
        # 1. Router 计算
        router_logits = self.router(hidden)
        top_k_weights, top_k_indices = torch.topk(router_logits, self.top_k)

        # 2. Fused MoE kernel
        output = fused_moe_kernel(
            hidden, self.w13, self.w2,
            top_k_weights, top_k_indices,
        )
        return output
```

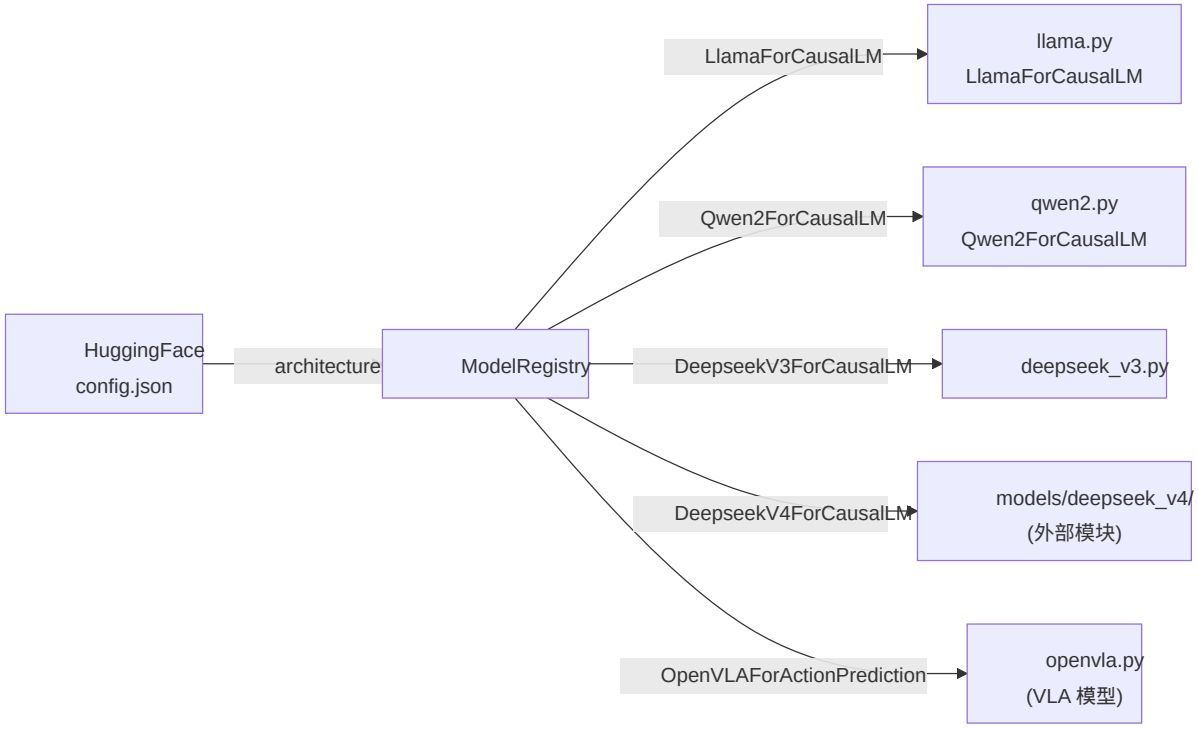
关键函数索引

函数/类	文件	职责
`ModelRegistry.register()`	models/registry.py	注册模型架构
`LlamaForCausalLM.forward()`	models/llama.py	LLaMA 前向传播
`Attention.forward()`	layers/attention/	统一注意力计算
`ColumnParallelLinear`	layers/linear.py	列切分线性层
`FusedMoE.forward()`	layers/fused_moe/	MoE 前向传播
`RMSNorm`	layers/layernorm.py	RMS 归一化
`RotaryEmbedding`	layers/rotary_embedding/	RoPE 位置编码

模型库与算子层 — 概念

Model Registry — `model_executor/models/registry.py`

vLLM 使用注册表模式将 HuggingFace 架构名映射到模型实现：



注册机制

```
# 简化的注册
@ModelRegistry.register("LlamaForCausalLM")
class LlamaForCausalLM(nn.Module):
    ...
```

模型目录隔离

部分模型因特殊硬件依赖（自定义 CUDA kernel、ROCm 特化注意力）从 `model_executor/models/` 移到独立包：

模型	位置	原因
DeepSeek V4	`vllm/models/deepseek_v4/`	自定义 CUDA/DSL kernel，需要 nvidia/ops
其他标准模型	`vllm/model_executor/models/`	无特殊依赖

Registry 通过 `_resolve_module_name()` 支持完全限定模块路径，自动定位外部模型包。

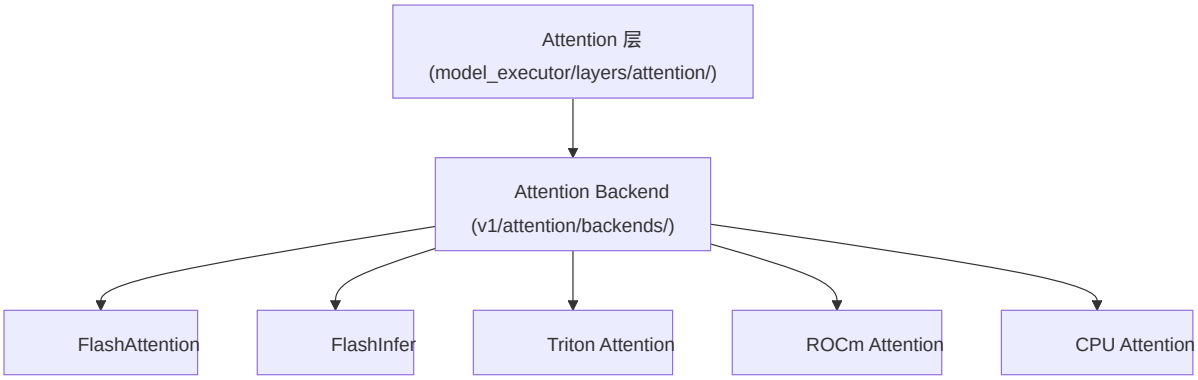
模型接口 **Mixin**

vLLM 通过 Mixin 接口声明模型的能力：

Mixin	说明
`SupportsLoRA`	支持 LoRA 适配器
`SupportsMultiModal`	支持多模态输入
`SupportsMRoPE`	支持多维旋转位置编码
`SupportsPP`	支持流水线并行
`HasInnerState`	有内部状态（如 Mamba）

注意力层

层次结构



运行时后端选择

`AttentionBackendSelector` 根据以下条件选择后端：

- 硬件平台（CUDA、ROCm、CPU）

- 模型配置（head_dim、sliding window）
- 是否启用 CUDA Graph

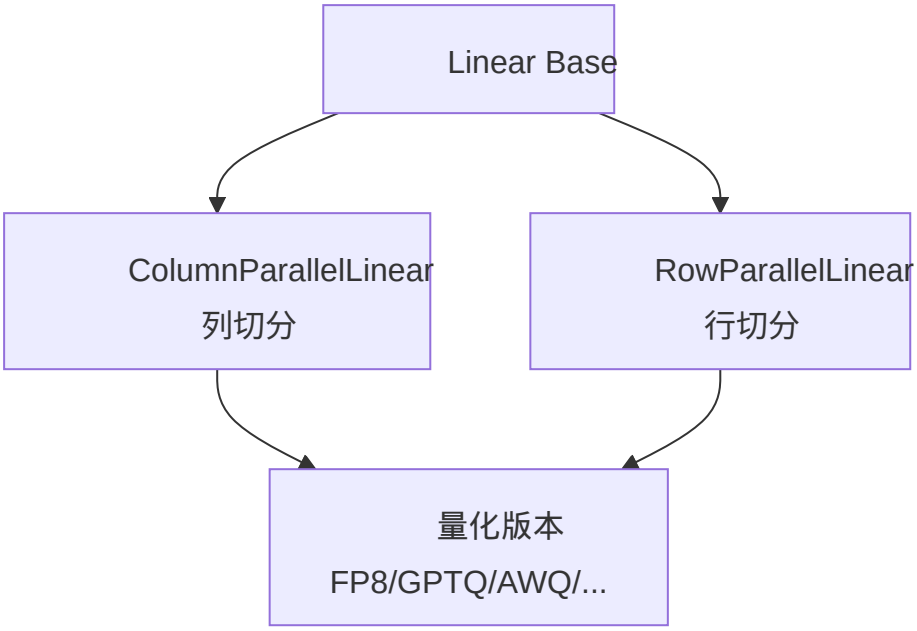
核心操作

注意力层需要实现两种操作：

1. **Prefill**：处理完整 prompt 的注意力
2. **Decode**：单 token 的增量注意力

线性层

vLLM 的线性层支持多种并行和量化模式：



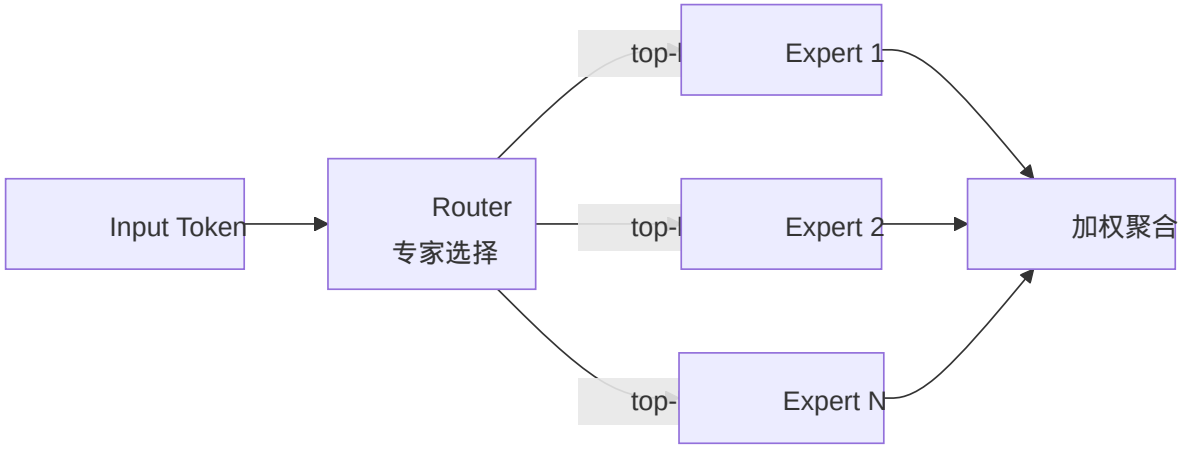
权重加载

vLLM 支持多种权重格式：

加载器	支持格式
`default_loader`	HuggingFace safetensors/bin
`gguf_loader`	GGUF 格式
`bitsandbytes_loader`	BitsAndBytes 量化权重
`tensorizer_loader`	Tensorizer 序列化

Fused MoE — `model_executor/layers/fused_moe/`

Mixture of Experts 模型的核心优化：



关键优化：

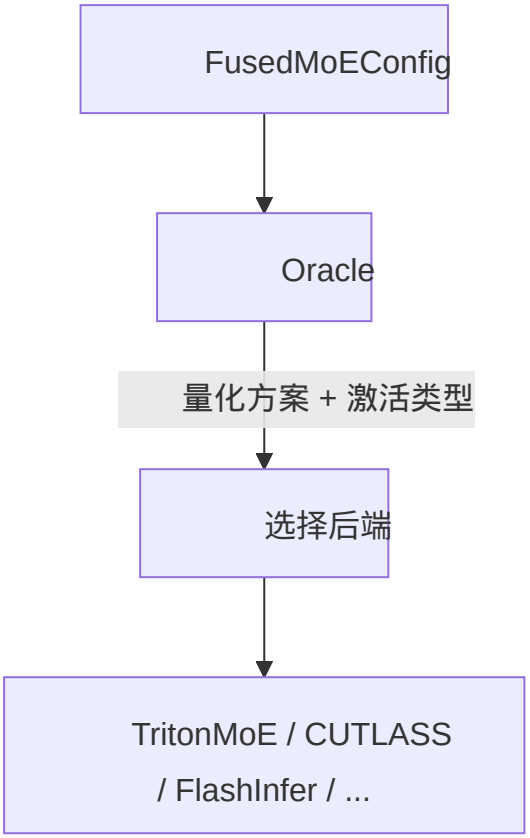
- **Fused Kernel**：将 router + expert 计算融合为单个 CUDA kernel
- **Expert Parallelism**：专家可以分布在多个 GPU 上
- 负载均衡：动态调整专家分配以平衡计算负载
- **Elastic EP**（弹性专家并行）：支持动态扩展/缩减 EP 大小，无需中断服务
- 双流 **LoRA**：MoE 的 base GEMM 和 LoRA delta 使用不同 CUDA 流并行执行

MoE 后端矩阵

后端	精度	硬件	说明
Triton MoE	FP16/BF16	NVIDIA	默认后端，支持双流 LoRA
CUTLASS MoE	W4A8 (FP8 act)	NVIDIA	通过 oracle 框架路由
FlashInfer B12x	NVFP4	SM12x (Blackwell)	融合 dispatch+GEMM+SwiGLU
Marlin MoE	AWQ/W4A16	NVIDIA	INT4 量化专家
ROCm MoE	FP16/BF16	AMD	ROCm 特化实现

Oracle 框架

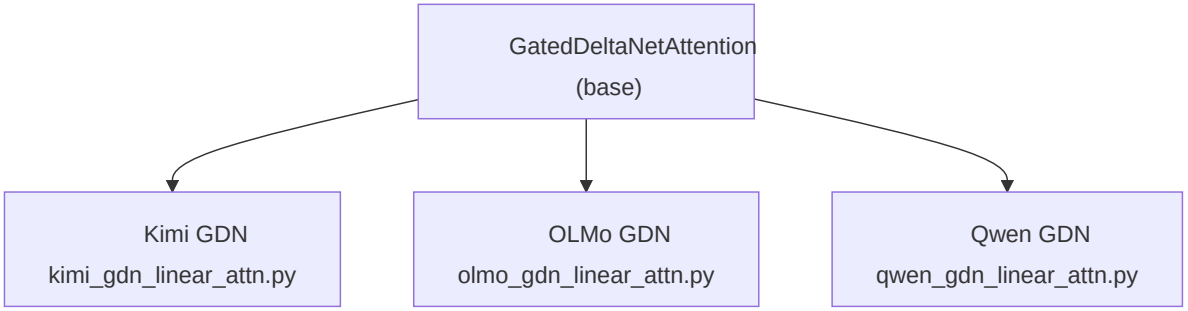
MoE 后端选择现在通过 oracle 模式统一管理：



各后端通过 `is_supported_config()`、`_supports_quant_scheme()` 等方法声明能力，oracle 根据模型配置自动选择最优后端。

Gated DeltaNet (GDN) — `model_executor/layers/mamba/gdn/`

新增的 GDN 子系统支持混合注意力模型中的线性注意力层：



- 从 OLMo Hybrid 模型中提取出独立的 GDN 层（减少 ~650 行）
- 支持 CPU 后端（`mamba/ops/cpu/gdn_attention.py`）
- 各模型共享 `GatedDeltaNetAttention` 基类，实现自己的线性注意力变体

旋转位置编码（**RoPE**）

vLLM 实现了多种 RoPE 变体：

变体	适用模型
Default RoPE	LLaMA、Mistral
LongRoPE	支持更长上下文
MRoPE	多模态模型
Phi3 Long RoPE	Phi-3

相关概念

- Flash Attention — 高效注意力计算
- MoE — 专家混合模型
- Tensor Parallelism — 张量并行影响线性层切分
- LoRA — 低秩适配器

反向链接

学习日志

模型库与算子层 — 练习

练习 1：添加新模型

假设需要添加一个新的模型架构 "SimpleModelForCausalLM"，请描述需要：

1. 创建哪些文件？
2. 实现哪些方法？
3. 如何注册到 ModelRegistry？

<details>
<summary>参考答案</summary>

1. 创建 `vllm/model_executor/models/simple_model.py`，实现模型类。
2. 必须实现：
   - `__init__`: 初始化模型层（embed、transformer layers、lm_head）
   - `forward`: 前向传播，返回 logprobs 或 hidden states
   - `load_weights`: 从 HuggingFace checkpoint 加载权重
   - 可选：实现 `SupportsLoRA`、`SupportsMultiModal` 等 Mixin
3. 使用装饰器注册：

```
@ModelRegistry.register("SimpleModelForCausalLM")
class SimpleModelForCausalLM(nn.Module):
    ...
```

</details>

练习 2：注意力后端对比

对比以下注意力后端的性能特征：

后端	Prefill	Decode	CUDA Graph
FlashAttention	?	?	?
FlashInfer	?	?	?
Triton	?	?	?

参考答案

后端	Prefill	Decode	CUDA Graph
FlashAttention	优秀（IO-aware 算法）	需要额外优化	支持
FlashInfer	优秀（专为推理优化）	最优（decode kernel）	最佳支持
Triton	良好	良好	支持

> FlashInfer 是 vLLM V1 的默认后端，专为推理场景优化，decode 性能最好。

练习 3：MoE 模型分析

分析 DeepSeek-V3 的 MoE 实现：

1. 每层有多少个专家？
2. Router 如何选择专家？
3. Fused MoE kernel 如何加速计算？

参考答案

1. DeepSeek-V3 使用 256 个路由专家 + 1 个共享专家。每个 token 激活 top-8 路由专家 + 共享专家。
2. Router 是一个线性层，输出 256 个专家的分数，选择 top-8。使用 sigmoid 而非 softmax，配合负载均衡损失。
3. Fused MoE kernel 将以下操作融合为一个 kernel：
   - Router 计算
   - Token 到 Expert 的分发
   - Expert 的线性计算
   - 结果聚合 避免了多次 global memory 读写，显著减少延迟。

拓展挑战

- 阅读 `vllm/model_executor/models/llama.py`，理解 LLaMA 模型的完整实现
- 分析 `fused_moe/` 目录下的 kernel 实现
- 研究 `quantization/` 目录下的量化方法如何与线性层集成

模型库与算子层

> 深入 vLLM 的模型实现层：200+ 模型架构支持、可复用算子层、注意力机制实现和专家混合模型。

涵盖内容

章节	核心主题
概念	Model Registry、模型接口、注意力层、MoE
练习	添加新模型、自定义算子分析
代码走读	llama.py、attention 层、linear 层关键代码

核心概念

vLLM 的模型层包含两大组件：

- 模型库（`model_executor/models/`）：200+ 模型架构实现，每个文件对应一种或多种 HuggingFace 架构
- 算子层（`model_executor/layers/`）：可复用的神经网络层，包括注意力、线性层、量化、MoE 等

前置知识

- 执行器与Worker
- Transformer 架构原理
- PyTorch 模型编程

学习路径

读完本主题后，你将理解：

- ModelRegistry 如何自动发现和注册模型
- 模型接口的 Mixin 设计（SupportsLoRA、SupportsMultiModal 等）
- 注意力层如何适配不同后端（Flash Attention、FlashInfer 等）
- MoE 模型的 Fused MoE 实现

→ 下一步：量化系统

反向链接

执行器与Worker — 概念

多模态处理

量化系统

推测解码

多模态处理 — 代码走读

MultiModalRegistry — `vllm/multimodal/registry.py`

```
    class MultiModalRegistry:
        def __init__(self):
            self._processors: dict[str, MultiModalProcessor] = {}

        def register_model(self, model_cls, processor_cls):
            self._processors[model_cls.__name__] = processor_cls()

        def get_processor(self, model_cls):
            return self._processors.get(model_cls.__name__)
```

MultiModalProcessor — `vllm/multimodal/processing/processor.py`

```
class MultiModalProcessor:
    def process(self, inputs: MultiModalInputs):
        # 1. 解析多模态数据
        image_data = inputs.get("image")
        audio_data = inputs.get("audio")
        video_data = inputs.get("video")

        # 2. 预处理各模态
        image_features = []
        if image_data:
            for img in image_data:
                pixel_values = self.image_processor(img)
                features = self.vision_encoder(pixel_values)
                image_features.append(features)

        # 3. 投影到语言模型空间
        projected = self.multi_modal_projector(image_features)

        # 4. 构建 inputs_embeds
        inputs_embeds = self._merge_text_and_features(
            text_embeds, projected, inputs["prompt"]
        )

        return inputs_embeds
```

Encoder Cache Manager — `vllm/v1/core/encoder_cache_manager.py`

```
class EncoderCacheManager:
    def __init__(self, cache_size, cache_dtype):
        self.cache = LRUCache(cache_size)
        self.cache_dtype = cache_dtype

    def get(self, cache_key):
        return self.cache.get(cache_key)

    def put(self, cache_key, features):
        self.cache.put(cache_key, features.to(self.cache_dtype))
```

相关输入类型 — `vllm/multimodal/inputs.py`

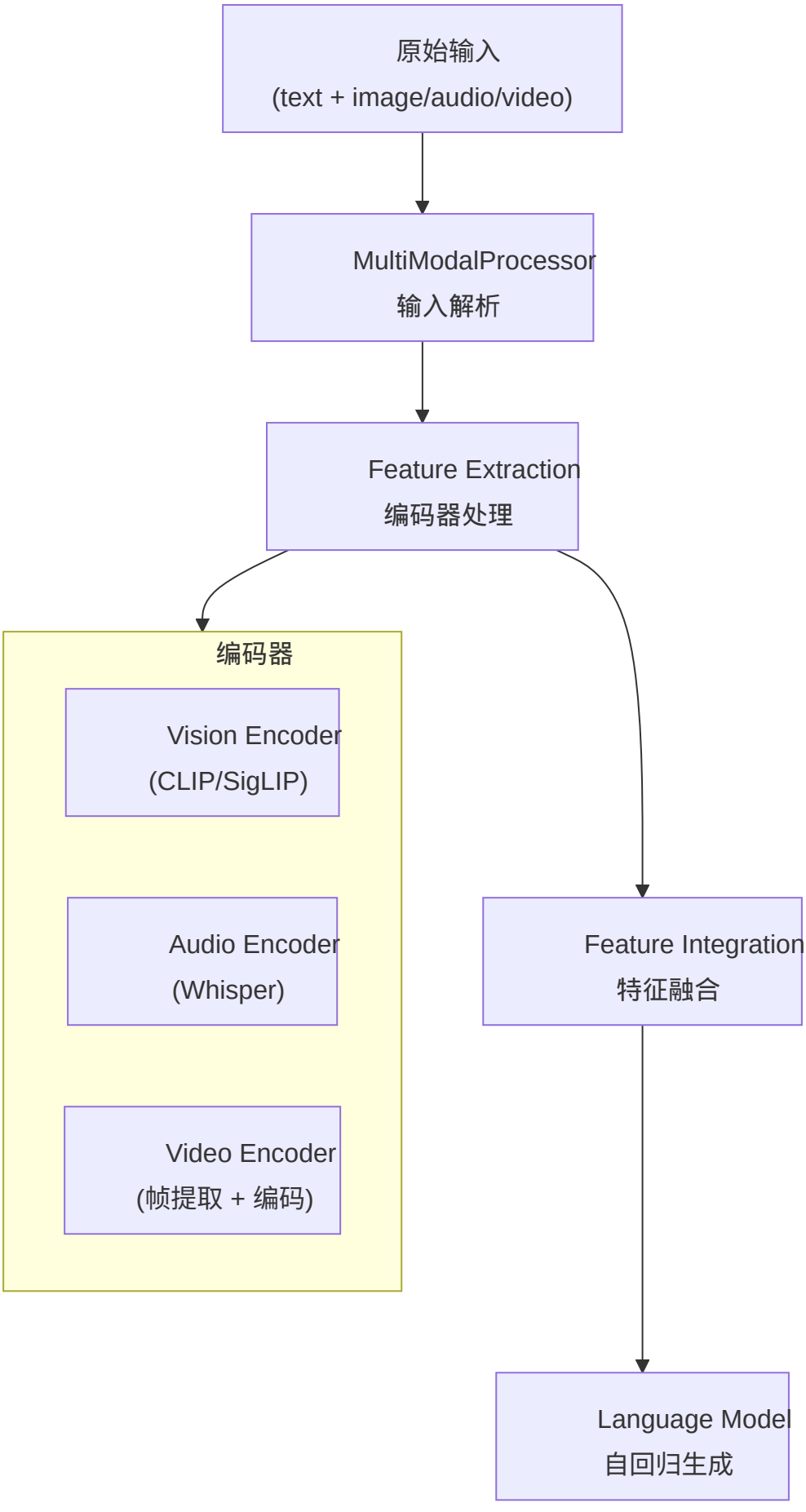
```
class MultiModalKwargs(TypedDict):
    image: Optional[list[ImageInput]]
    audio: Optional[list[AudioInput]]
    video: Optional[list[VideoInput]]

class MultiModalInputs:
    prompt: str
    token_ids: list[int]
    multi_modal_data: MultiModalKwargs
```

关键函数索引

函数/类	文件	职责
`MultiModalRegistry.register_model()`	multimodal/registry.py	注册模型的多模态处理器
`MultiModalProcessor.process()`	multimodal/processing/processor.py	处理多模态输入
`EncoderCacheManager.get/put()`	v1/core/encoder_cache_manager.py	编码器输出缓存
`MultiModalInputs`	multimodal/inputs.py	多模态输入数据结构
`ImageProcessor`	multimodal/image.py	图像预处理
`AudioProcessor`	multimodal/audio.py	音频预处理

多模态处理 — 概念

多模态输入处理流水线


原始输入
(text + image/audio/video)
MultiModalProcessor
输入解析
Feature Extraction
编码器处理
编码器
Vision Encoder
(CLIP/SigLIP)
Audio Encoder
(Whisper)
Video Encoder
(帧提取 + 编码)
Feature Integration
特征融合
Language Model
自回归生成

输入类型

图像输入

```
# 图像输入格式
{
    "prompt": "描述这张图片",
    "multi_modal_data": {
        "image": PILImage or URL or base64,
    }
}
```

处理流程：

1. 图像解码和预处理（resize、normalize）
2. Vision Encoder 提取特征
3. 投影层映射到语言模型的 embedding 空间
4. 替换文本中的 `<image>` 占位符

音频输入

处理流程：

1. 音频解码和特征提取（mel spectrogram）
2. Audio Encoder（通常是 Whisper 编码器）提取特征
3. 投影层映射到 embedding 空间

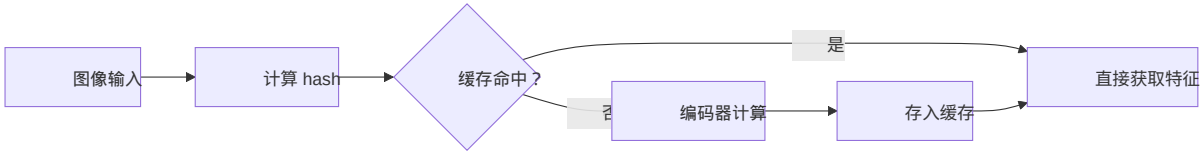
视频输入

处理流程：

1. 视频解码，均匀采样帧
2. 每帧通过 Vision Encoder
3. 时序聚合（ pooling 或 temporal attention）

编码器缓存

多模态编码器的计算开销很大。vLLM 实现了编码器缓存：



- **EncoderCacheManager**：管理编码器输出的缓存
- 缓存键：输入内容的哈希值
- 缓存淘汰：LRU 或引用计数策略

多模态注册表

每个模型声明支持的多模态类型：

```
# 简化示意
@ModelRegistry.register("LlavaForConditionalGeneration")
@SupportsMultiModal
class LlavaForConditionalGeneration:
    supported_modalities = ["image"]

    def get_multimodal_processor(self):
        return LlavaProcessor()
```

新模型：**OpenVLA**

Vision-Language-Action 模型，用于机器人操控任务：

- 使用融合 DINOv2 + SigLIP 视觉骨干（通过 `timm`）
- `PrismaticProjector` 进行多模态桥接
- 动作 token 预测输出

编码器 **CUDA Graph**

多模态编码器现在支持 CUDA Graph 加速：

- `SupportsEncoderCudaGraph` 协议重构为 `get_encoder_cudagraph_item_specs()` 返回 `EncoderItemSpec` 对象
- 支持 Step3VL 等模型的编码器 CUDA Graph 捕获
- 新增 `postprocess_encoder_output()` 方法，默认行为为 `scatter_output_slices`

特征融合策略

多模态特征与文本 embedding 的融合方式：

策略	描述	适用模型
Token 替换	图像 token 替换 `<image>` 占位符	LLaVA
Prefix 拼接	图像特征拼接到文本前面	Qwen-VL
交叉注意力	图像特征作为交叉注意力的 K/V	Flamingo
交错融合	文本和图像 token 交错排列	InternVL

多模态模型性能优化

批处理挑战

多模态输入导致 token 数量差异很大：

- 纯文本请求：~100 tokens
- 多图请求：~2000+ tokens（每张图可能几百个 token）

优化策略：

- 动态批处理：根据总 token 数（文本 + 多模态）调整 batch size
- 编码器预算管理：限制编码器计算的并发量
- 异步编码：编码器计算与 decode 并行

相关概念

- KV Cache — 多模态 token 的 KV 缓存管理
- Continuous Batching — 多模态请求的调度
- Prefix Caching — 编码器输出的缓存复用

反向链接

学习日志

多模态处理 — 练习

练习 1：图像 token 计数

假设 LLaVA 模型使用 CLIP-ViT-L/14 作为 vision encoder：

- 每张图像被分割为 14×14 的 patch
- 输入分辨率 336×336
- 每张图有 1 个 CLS token

计算：

1. 每张图像产生多少个 visual token？
2. 如果 batch 中有 4 个请求，分别有 0、1、2、4 张图，总 token 数是多少（假设每请求 100 文本 token）？

参考答案

1. Patch 数 = (336/14) × (336/14) = 24 × 24 = 576 个 patch token + 1 个 CLS token = 577 个 visual token。
2. 总 token 数：
   - 请求 A（0图）：100 tokens
   - 请求 B（1图）：100 + 577 = 677 tokens
   - 请求 C（2图）：100 + 577×2 = 1254 tokens
   - 请求 D（4图）：100 + 577×4 = 2408 tokens
   - 总计：**4439 tokens**

   注意：这与纯文本批处理（4×100=400 tokens）差了 10 倍，对调度和显存管理有重大影响。

练习 2：编码器缓存策略

分析以下场景中的编码器缓存效果：

1. 多用户上传同一张图片询问不同问题
2. 视频推理中同一视频的多个帧
3. 每个请求都使用不同的图片

<details>
<summary>参考答案</summary>

1. 极高效：所有请求共享同一份视觉特征。缓存命中率 100%（除第一个请求）。
2. 部分有效：同一视频的不同帧需要分别编码。但如果使用 prefix caching，共享的视频帧可以复用 KV 缓存。
3. 无效：每个请求的图片不同，缓存命中率接近 0。缓存反而增加了内存开销。

</details>

拓展挑战

- 阅读 `vllm/multimodal/processing/processor.py`，理解多模态输入的完整处理流程
- 分析 `encoder_cache_manager.py` 的缓存淘汰策略
- 研究如何为新模型添加多模态支持

多模态处理

> 深入 vLLM 的多模态支持：图像、音频、视频输入的处理流水线，以及多模态模型（VLM）的推理优化。

涵盖内容

章节	核心主题
概念	多模态输入类型、处理流水线、编码器缓存
练习	多模态输入处理分析、性能优化
代码走读	multimodal 目录关键代码

核心概念

vLLM 支持多种多模态输入类型：

- 图像：支持多图输入、动态分辨率
- 音频：语音识别、音频理解
- 视频：视频帧提取、时序编码
- 混合输入：文本 + 图像 + 音频的组合

前置知识

- 模型库与算子层
- Vision Transformer 基本原理
- 多模态模型（LLaVA、Qwen-VL 等）基本架构

学习路径

读完本主题后，你将理解：

- 多模态输入的解析和预处理流水线
- 编码器（Vision Encoder、Audio Encoder）的缓存策略
- 多模态特征如何与语言模型的 embedding 融合
- 多模态推理的性能优化技巧

→ 下一步：分布式计算

量化系统 — 代码走读

量化配置 — `vllm/config/quantization.py`

```
@dataclass
class QuantizationConfig:
    quant_method: str  # "gptq", "awq", "fp8", etc.

    @staticmethod
    def from_config(model_config):
        """从 HuggingFace config 中解析量化配置"""
        quant_cfg = model_config.quantization_config
        method = quant_cfg.get("quant_method", "")

        if method == "gptq":
            return GPTQConfig(**quant_cfg)
        elif method == "awq":
            return AWQConfig(**quant_cfg)
        elif method == "fp8":
            return FP8Config(**quant_cfg)
        ...
```

量化线性层 — `model_executor/layers/quantization/`

GPTQ 实现

```python
class GPTQMarlinLinearMethod(QuantizeMethodBase):
    def create_quantized_linear(self, linear):
        # 替换权重为量化格式
        linear.weight = Parameter(
            self._pack_qweight(linear.weight),
            requires_grad=False,
        )
        linear.scales = Parameter(self._compute_scales())
        return linear

    def apply(self, linear, x):
        # 调用 Marlin kernel
        return gptq_marlin_gemm(
            x, linear.weight, linear.scales,
            linear.workspace, linear.w_qzeros,
        )
```

FP8 实现

```python
class Fp8LinearMethod(QuantizeMethodBase):
    def process_weights_after_loading(self, model):
        for layer in model.modules():
            if isinstance(layer, Linear):
                # 将 FP16 权重转换为 FP8
                layer.weight = Parameter(
                    torch._scaled_mm(
                        layer.weight, scale=1.0,
                        output_dtype=torch.float8_e4m3fn,
                    ),
                    requires_grad=False,
                )

    def apply(self, linear, x):
        # FP8 矩阵乘法
        return torch._scaled_mm(
            x, linear.weight,
            scale_a=linear.input_scale,
            scale_b=linear.weight_scale,
            output_dtype=torch.float16,
        )
```

量化框架集成

权重加载时自动量化

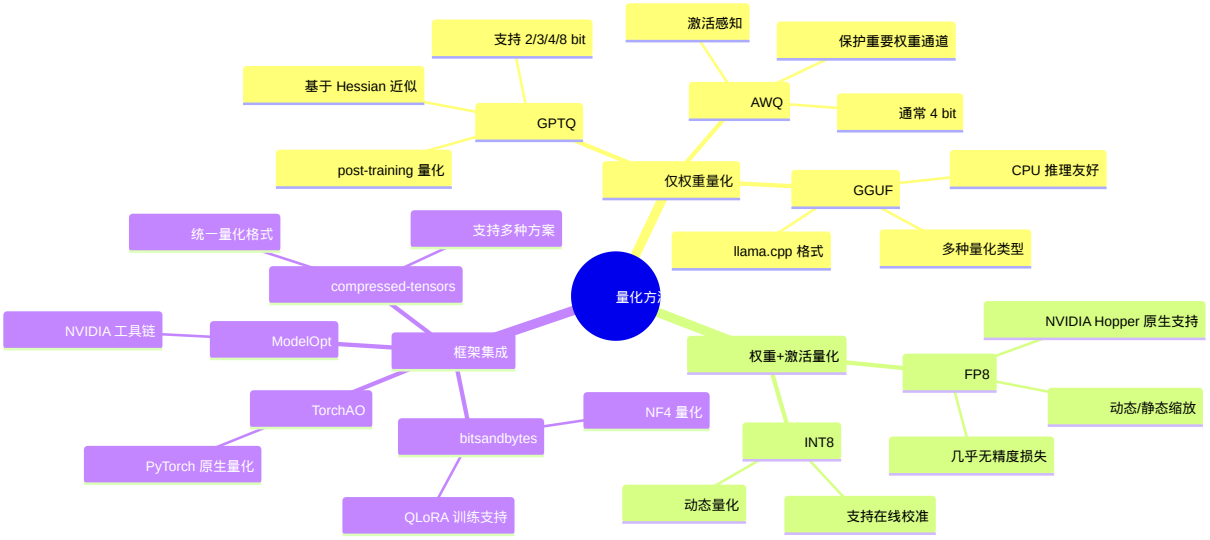
```
    # 在模型加载时
    class DefaultModelLoader:
        def _load_module(self, module, weights):
            for name, param in module.named_parameters():
                if hasattr(module, 'quant_method'):
                    # 量化方法处理权重
                    module.quant_method.create_quantized_linear(module)
                else:
                    # 正常加载
                    module.load_weights(weights)
```

关键函数索引

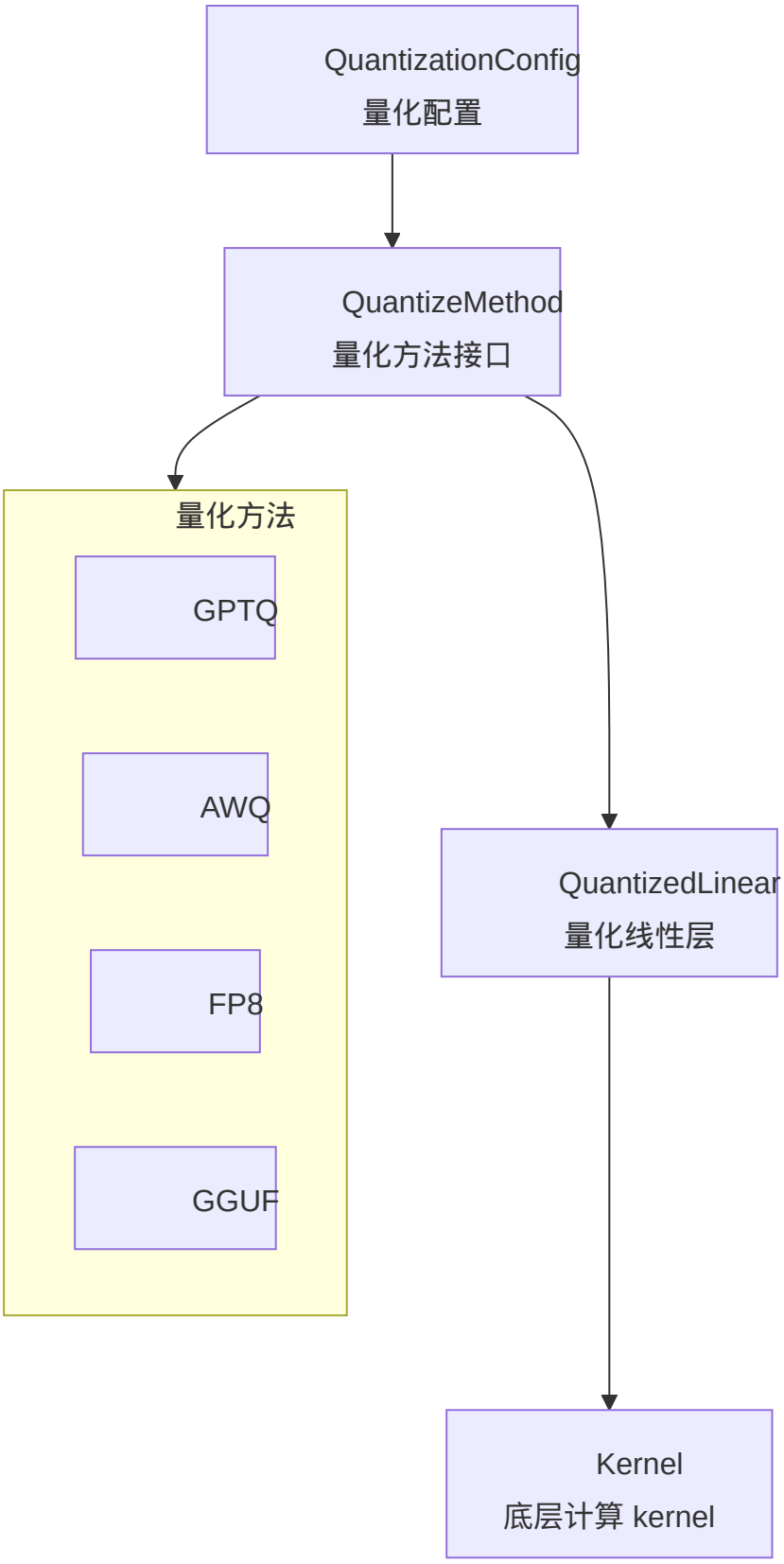
函数/类	文件	职责
`QuantizationConfig.from_config()`	config/quantization.py	解析量化配置
`GPTQMarlinLinearMethod.apply()`	layers/quantization/gptq/	GPTQ 矩阵乘法
`Fp8LinearMethod.apply()`	layers/quantization/fp8/	FP8 矩阵乘法
`AWQMarlinLinearMethod.apply()`	layers/quantization/awq/	AWQ 矩阵乘法
`process_weights_after_loading()`	各量化方法	权重后处理

量化系统 — 概念

量化方法分类



量化框架架构


QuantizationConfig
量化配置
QuantizeMethod
量化方法接口
量化方法
GPTQ
AWQ
FP8
GGUF
QuantizedLinear
量化线性层
Kernel
底层计算 kernel

核心接口

```
class QuantizeMethodBase(ABC):
    @abstractmethod
    def create_quantized_linear(self, linear):
        """将普通线性层替换为量化版本"""
        ...

    @abstractmethod
    def process_weights_after_loading(self, model):
        """权重加载后处理（如重排、缩放）"""
        ...
```

主流量化方法详解

FP8 量化

FP8（E4M3/E5M2）是 NVIDIA Hopper (H100) 原生支持的格式：

- **E4M3**：用于前向传播（权重和激活）
- **E5M2**：用于反向传播（梯度）
- 几乎无精度损失（相比 FP16）
- 需要 H100/RTX 4090+ 硬件支持（`compute_capability >= 8.9`）
- TorchAO 后端现在在初始化时验证 GPU 能力，不支持的设备给出清晰的降级建议

NVFP4 量化

NVFP4 是最新的超低精度格式：

- **W4A4 NVFP4**：权重和激活都使用 4-bit 浮点
- **W4A16 NVFP4**：权重 4-bit，激活 16-bit（新增混合精度模式）
- 支持 MoE 模型的融合量化 + dispatch
- 批量不变式内核消除了 padding 开销

GPTQ

基于 Hessian 近似的后训练量化：

1. 逐层量化：每层独立处理
2. 使用校准数据计算 Hessian

3. 量化每个权重时补偿对未量化权重的影响
4. 支持 2/3/4/8 bit，通常使用 4 bit

AWQ

激活感知权重量化：

1. 观察激活值分布，识别重要权重通道
2. 对重要通道使用更高的缩放因子
3. 保护对激活值敏感的权重
4. 通常 4 bit，精度接近 FP16
5. AWQ-Marlin MoE 现在通过 oracle 层统一处理权重转换，去除了直接调用底层 kernel 的代码

量化对性能的影响

方法	显存节省	推理加速	精度损失
FP8	~50%	1.5-2×	极小
NVFP4 (W4A4)	~87%	1.5-2.5×	小
NVFP4 (W4A16)	~75%	1.3-2×	小
GPTQ-4bit	~75%	1.2-1.5×	小
AWQ-4bit	~75%	1.3-1.8×	小
GGUF-Q4	~75%	取决于硬件	中等

量化框架变更

Compressed Tensors 稀疏性（2:4）移除

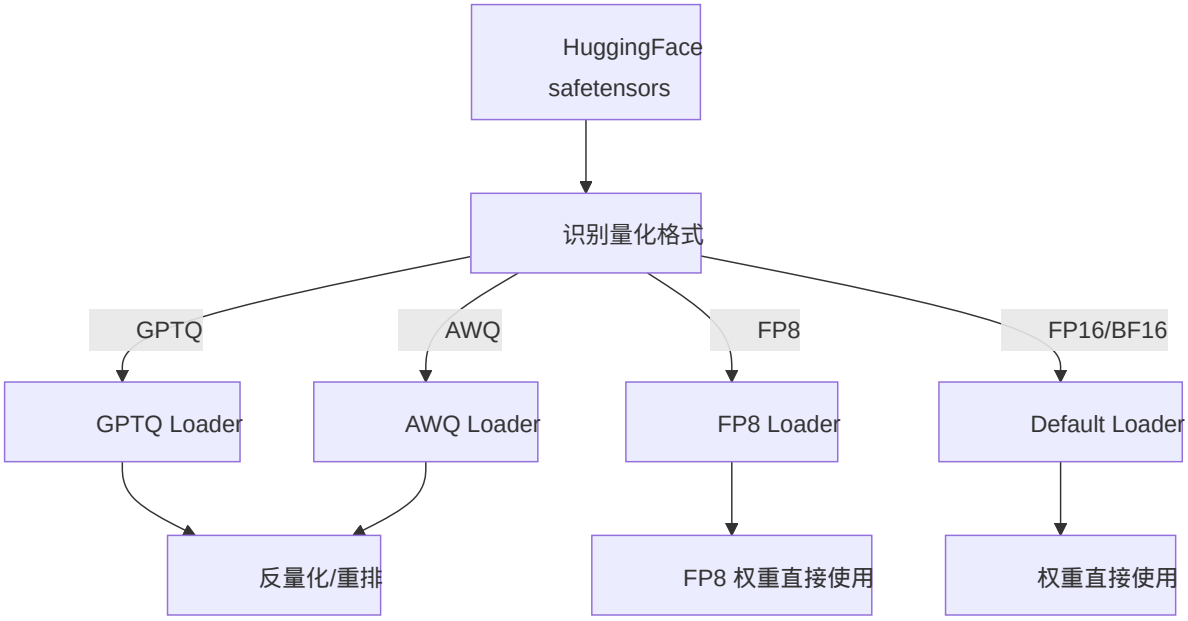
Compressed Tensors 框架移除了 2:4 稀疏性支持。检测到稀疏配置时将发出 `DeprecationWarning`。相关删除：

- `CompressedTensors24` scheme 类
- `supports_cutlass_24()` 方法
- `sparsity_scheme_map` 和 `sparsity_ignore_list` 配置字段

AWQ-Marlin 统一到 Oracle

AWQ-Marlin MoE 的权重处理现在委托给 oracle 层的 `convert_to_wna16_moe_kernel_format()`，去除了直接调用 `ops.awq_marlin_moe_repack`、`marlin_moe_permute_scales` 等底层函数的代码。所有 WNA16 格式（AWQ、GPTQ）的权重转换逻辑集中到 `oracle/int_wna16.py`。

权重加载流程



相关概念

- MoE — MoE 模型也可以使用量化
- Tensor Parallelism — 量化与张量并行的兼容
- torch compile — 编译优化与量化的协同

反向链接

学习日志

量化系统 — 练习

练习 **1**：量化方法选择

为以下场景选择最合适的量化方法：

1. H100 集群，需要最小精度损失
2. 消费级 GPU（RTX 3090），需要运行 70B 模型
3. CPU-only 推理，需要最大化压缩
4. 在线服务，需要兼顾精度和速度

参考答案

1. **FP8**：H100 原生支持，几乎无精度损失，2× 加速。
2. **GPTQ** 或 **AWQ**（**4-bit**）：70B 模型 FP16 需要 ~140GB，RTX 3090 只有 24GB。4-bit 量化后约 35GB，配合 offload 到 CPU 内存可以运行。AWQ 通常精度更好。
3. **GGUF**：llama.cpp 格式，专为 CPU 推理优化，支持多种量化级别（Q2_K 到 Q8_0）。
4. **AWQ**（**4-bit**）：在保持较高精度的同时提供良好的推理速度。如果硬件支持 FP8，FP8 是更好的选择。

练习 **2**：量化精度分析

分析量化对模型输出的影响：

1. 为什么 FP8 的精度损失比 INT4 小得多？
2. GPTQ 和 AWQ 在量化策略上的根本区别是什么？
3. 为什么量化对 MoE 模型更有挑战性？

参考答案

1. FP8（E4M3）仍然有 3 bit 的尾数和 4 bit 的指数，可以表示的动态范围远大于 INT4（只能表示 16 个离散值）。
2. GPTQ 基于权重的重要性（通过 Hessian 衡量）来决定量化策略，量化一个权重后补偿其余权重。AWQ 基于激活值来识别重要权重通道，通过缩放因子保护重要通道。
3. MoE 模型有多个专家，每个专家的权重分布可能不同。量化需要分别处理每个专家，且 Router 的决策可能因量化误差而改变。专家的权重矩阵通常较小，量化效果不如大矩阵显著。

拓展挑战

- 阅读 `vllm/model_executor/layers/quantization/` 目录结构，理解量化方法的组织方式
- 分析 GPTQ 的 Marlin kernel 实现
- 研究 compressed-tensors 量化格式的统一接口

量化系统

> 深入 vLLM 的量化框架：支持 30+ 量化方法，包括 FP8、GPTQ、AWQ、GGUF 等，以及量化与线性层的集成机制。

涵盖内容

章节	核心主题
概念	量化原理、量化方法分类、权重加载、kernel 调度
练习	量化效果分析、精度与性能权衡
代码走读	quantization 框架、量化线性层、kernel 实现

核心概念

量化是降低模型精度以减少显存占用和加速推理的关键技术：

- 权重量化：将 FP16/BF16 权重压缩为 INT8/INT4/FP8
- 激活量化：部分方法支持激活值的动态量化
- 混合精度：不同层使用不同的量化策略

前置知识

- 模型库与算子层
- GPU 计算（FP16、INT8、FP8）基础概念
- 线性层和矩阵乘法的基本原理

学习路径

读完本主题后，你将理解：

- vLLM 的量化框架设计和扩展方式
- 30+ 量化方法的分类和适用场景
- 量化权重如何加载并与线性层集成
- 量化对推理性能和精度的影响

→ 下一步：推测解码

调度系统 — 代码走读

Scheduler 类 — `vllm/v1/core/sched/scheduler.py`

这是 vLLM 中较大的文件之一（约 2300 行），实现了所有调度逻辑。

核心数据结构

```
class Scheduler:
    def __init__(self, ...):
        # 三个请求队列
        self.waiting: deque[Request] = deque()
        self.running: list[Request] = []
        self.swapped: list[Request] = []

        # KV 缓存管理器
        self.kv_cache_manager = KVCacheManager(...)

        # 调度预算
        self.max_num_seqs = scheduler_config.max_num_seqs
        self.max_num_batched_tokens = scheduler_config.max_num_batched_tokens
```

schedule() 主方法

```
def schedule(self) -> SchedulerOutput:
    # 阶段 1：处理 swapped 队列
    scheduler_output = self._schedule_swapped()

    # 阶段 2：处理 running 队列（decode）
    scheduler_output = self._schedule_running(scheduler_output)

    # 阶段 3：处理 waiting 队列（prefill）
    scheduler_output = self._schedule_prefills(scheduler_output)

    return scheduler_output
```

_schedule_prefills 实现

```
    def _schedule_prefills(self, scheduler_output):
        # 计算剩余预算
        remaining_tokens = self.max_num_batched_tokens - scheduler_output.num_batche
    d_tokens
        remaining_seqs = self.max_num_seqs - len(self.running)

        while self.waiting and remaining_seqs > 0 and remaining_tokens > 0:
            request = self.waiting[0]

            # Chunked prefill：只处理部分 tokens
            num_tokens = min(
                len(request.remaining_token_ids),
                remaining_tokens,
            )

            # 尝试分配 KV 缓存块
            new_blocks = self.kv_cache_manager.allocate(request, num_tokens)

            if new_blocks is None:
                # 显存不足，尝试 preemption
                if self._preempt(request):
                    continue
                break

            # 将请求从 waiting 移到 running
            self.waiting.popleft()
            self.running.append(request)
            scheduler_output.add_prefill(request, num_tokens, new_blocks)
            remaining_tokens -= num_tokens
            remaining_seqs -= 1

        return scheduler_output
```

KVCacheManager — `vllm/v1/core/kv_cache_manager.py`

块分配

```python
class KVCacheManager:
    def allocate(self, request, num_tokens):
        # 计算需要的 block 数量
        num_new_blocks = cdiv(
            request.num_computed_tokens + num_tokens,
            self.block_size,
        ) - request.num_blocks

        if num_new_blocks > self.block_pool.num_free_blocks:
            return None  # 显存不足

        # 从空闲池分配 block
        new_blocks = self.block_pool.allocate(num_new_blocks)

        # 前缀缓存：检查是否可以复用已有 block
        if self.enable_prefix_caching:
            shared_blocks = self._find_shared_prefix(request)
            new_blocks = self._deduplicate(new_blocks, shared_blocks)

        return new_blocks
```

BlockPool — `vllm/v1/core/block_pool.py`

```python
class BlockPool:
    def __init__(self, num_blocks):
        self.free_blocks: list[KVCacheBlock] = [
            KVCacheBlock(id=i) for i in range(num_blocks)
        ]
        self.num_free_blocks = len(self.free_blocks)

    def allocate(self, num_blocks):
        if num_blocks > self.num_free_blocks:
            return None
        blocks = self.free_blocks[:num_blocks]
        self.free_blocks = self.free_blocks[num_blocks:]
        self.num_free_blocks -= num_blocks
        return blocks

    def free(self, blocks):
        self.free_blocks.extend(blocks)
        self.num_free_blocks += len(blocks)
```

RequestQueue — `vllm/v1/core/sched/request_queue.py`

支持多种调度策略：

```
class RequestQueue:
    def __init__(self, policy="fcfs"):
        self.policy = policy
        self.queue: list[Request] = []

    def push(self, request):
        if self.policy == "fcfs":
            self.queue.append(request)
        elif self.policy == "priority":
            # 按优先级插入排序
            bisect.insort(self.queue, request, key=lambda r: -r.priority)

    def pop(self):
        return self.queue.pop(0)
```

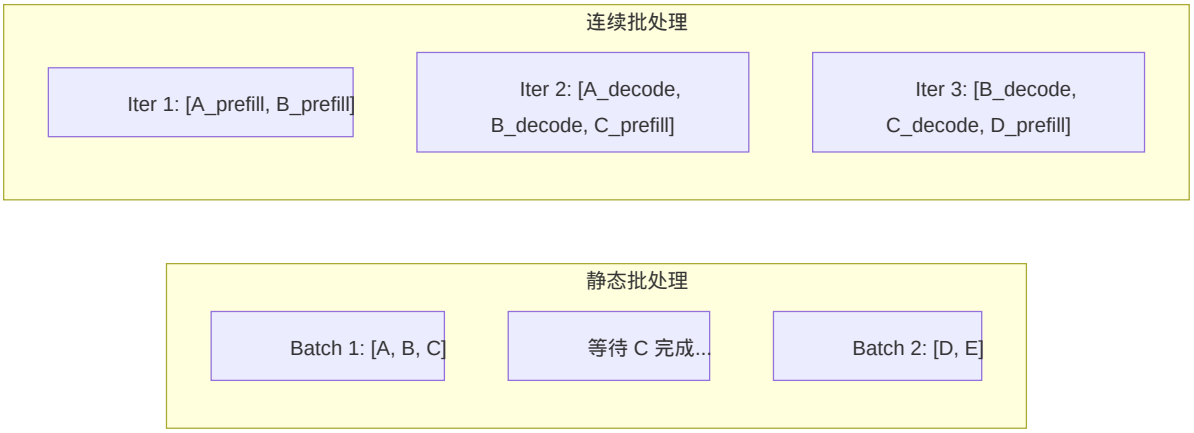
关键函数索引

函数/类	文件	职责
`Scheduler.schedule()`	v1/core/sched/scheduler.py	主调度入口
`Scheduler._schedule_prefills()`	v1/core/sched/scheduler.py	处理等待中的 prefill 请求
`Scheduler._schedule_running()`	v1/core/sched/scheduler.py	处理活跃 decode 请求
`Scheduler._schedule_swapped()`	v1/core/sched/scheduler.py	恢复被交换的请求
`Scheduler._preempt()`	v1/core/sched/scheduler.py	抢占低优先级请求
`KVCacheManager.allocate()`	v1/core/kv_cache_manager.py	分配 KV 缓存块
`BlockPool.allocate()`	v1/core/block_pool.py	空闲块分配
`RequestQueue.push()`	v1/core/sched/request_queue.py	请求入队

调度系统 — 概念

连续批处理（**Continuous Batching**）

传统静态批处理中，一个 batch 内所有请求必须全部完成才能开始新 batch。Continuous Batching 打破了这个限制：

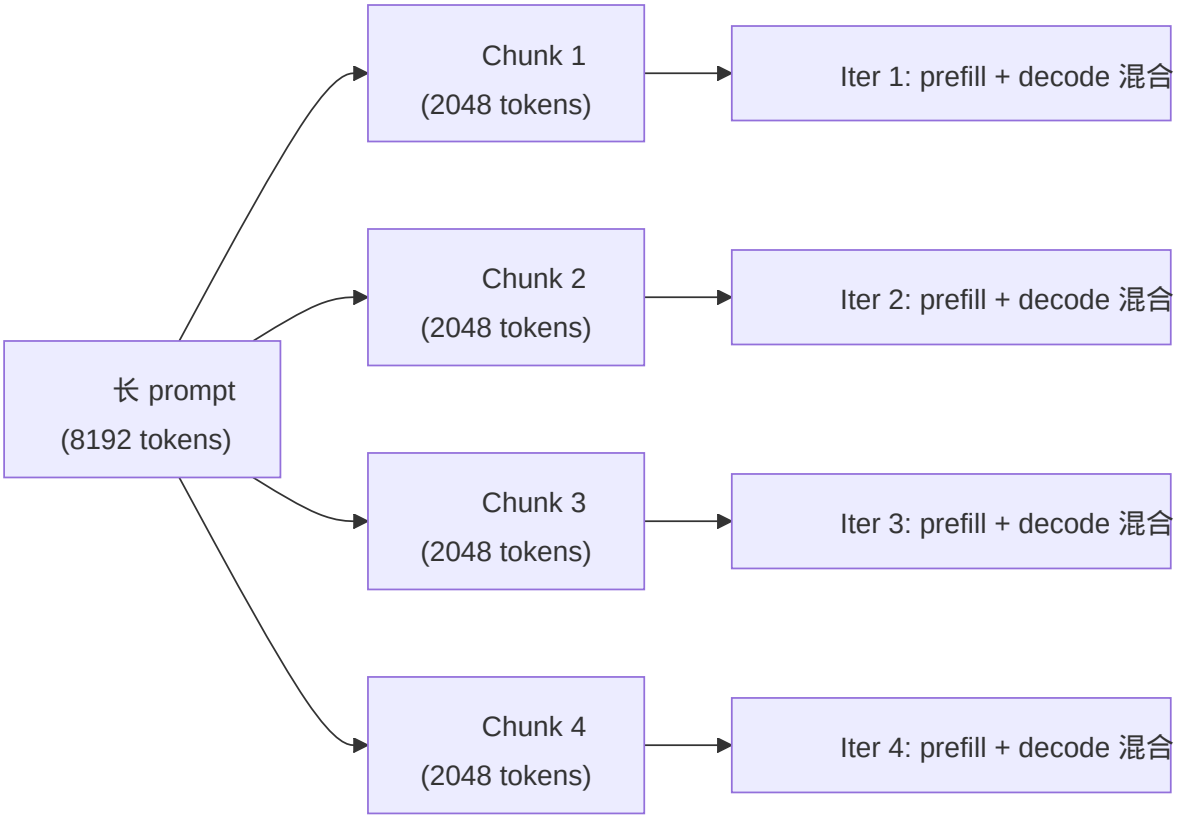


关键优势：

- 更低的延迟：新请求无需等待整个 batch 完成
- 更高的吞吐：GPU 始终保持满载
- 更公平的调度：短请求不会被长请求阻塞

Chunked Prefill

长 prompt 的 prefill 可能非常耗时（占用大量计算资源和显存）。Chunked Prefill 将 prefill 分成多个 chunk：



好处：

- 降低 **TTFT**：不需要一次性处理完整个 prompt
- 混合调度：prefill 和 decode 可以在同一轮迭代中混合执行
- 显存友好：每次只需分配部分 prompt 的 KV 缓存

Preemption 策略

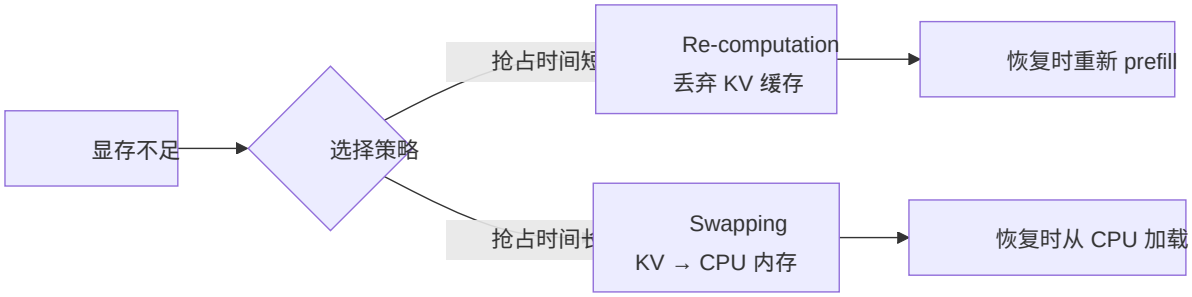
当显存不足以容纳所有活跃请求的 KV 缓存时，调度器会触发 preemption：

1. Re-computation（重新计算）

直接丢弃被抢占请求的 KV 缓存块。当请求恢复时，重新计算 prefill。适用于抢占时间短的场景。

2. Swapping（交换到 **CPU**）

将 KV 缓存块从 GPU 交换到 CPU 内存。恢复时再交换回来。适用于抢占时间较长的场景。



3. 异步抢占多帧丢弃

当使用推测解码或流水线并行时，被抢占请求可能还有多个在途（in-flight）的输出帧。旧的实现使用布尔标志只能丢弃一帧，新的计数器机制逐帧正确排空：

```
discard_latest_async_tokens (bool) → async_tokens_to_discard (int counter)
```

调度器将 `async_tokens_to_discard` 设为 `num_output_placeholders`，异步调度器每帧递减计数器，确保所有在途帧被正确排空后才恢复请求。

4. KV Connector 延迟释放

KV Connector 的异步操作可能在请求从调度队列移除后仍在进行。`has_finished_requests()` 现在额外检查 `self.requests` 中是否还有等待 KV connector 延迟清理的请求，确保调度器不会过早认为"无请求"。

请求队列与优先级

调度器维护多个请求队列：

队列	用途
waiting	等待 prefill 的新请求
running	正在 decode 的活跃请求
swapped	被交换到 CPU 的请求

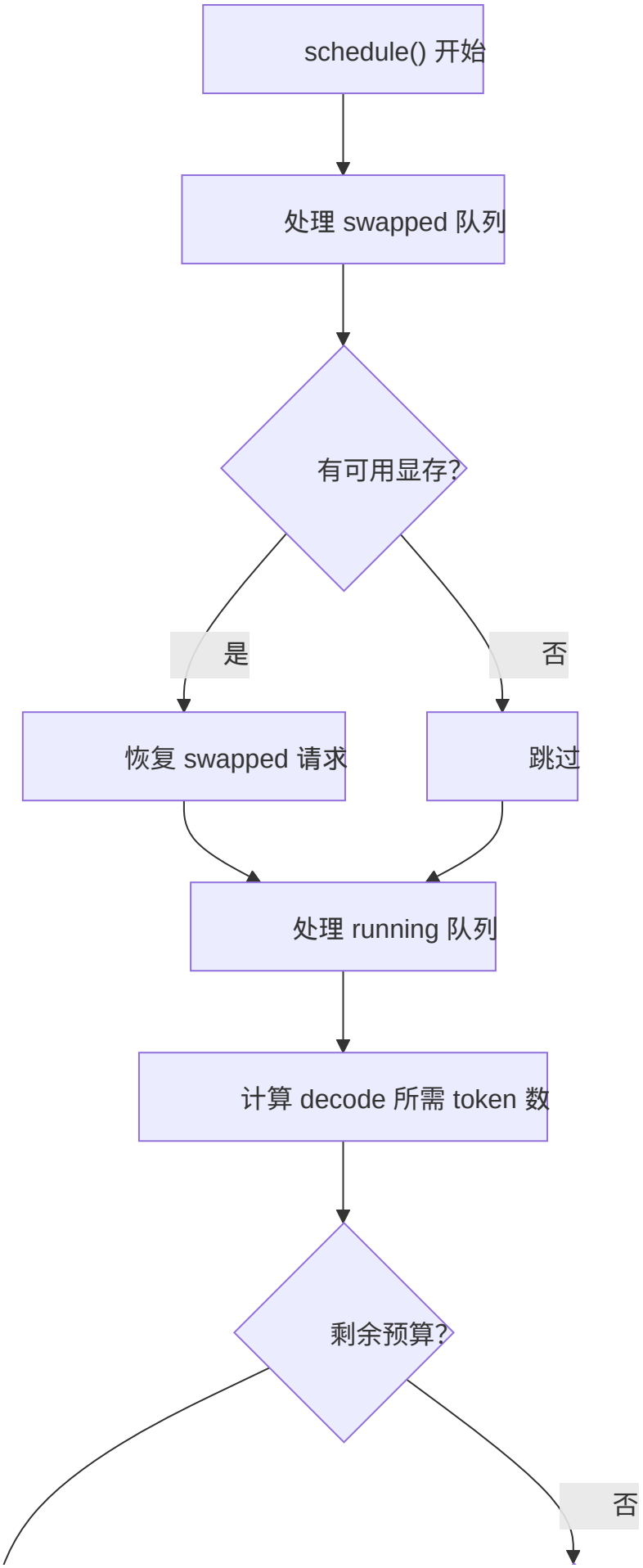
调度优先级：

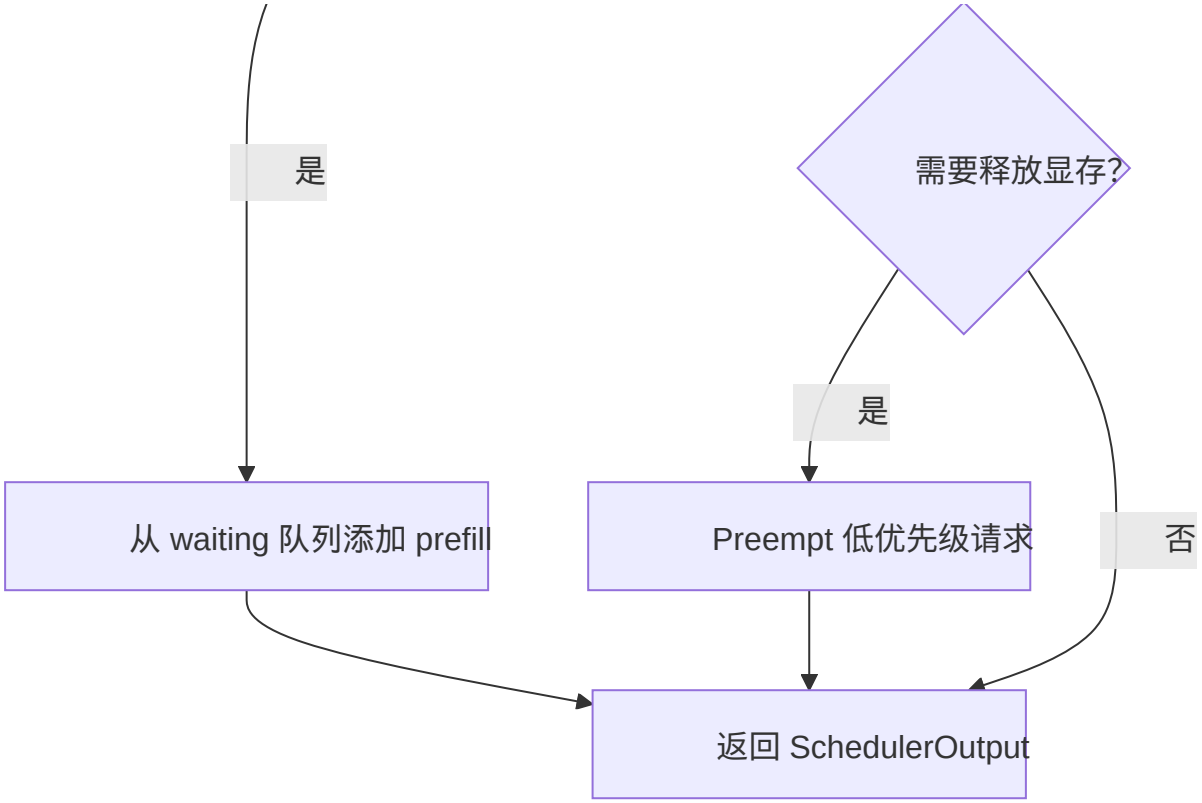
1. 处理 swapped 队列（恢复被抢占的请求）

2. 继续 running 队列的 decode

3. 从 waiting 队列添加新 prefill

调度决策流程


schedule() 开始
处理 swapped 队列
有可用显存?
是
否
恢复 swapped 请求
跳过
处理 running 队列
计算 decode 所需 token 数
剩余预算?
否



调度配置参数

参数	默认值	含义
`max_num_seqs`	256	最大并发序列数
`max_num_batched_tokens`	8192	每轮迭代最大 token 数
`max_model_len`	模型默认	最大序列长度
`chunked_prefill_enabled`	True	是否启用 chunked prefill
`scheduling_policy`	"fcfs"	调度策略（先来先服务/优先级）

相关概念

- Paged Attention — KV 缓存块的管理方式
- KV Cache — KV 缓存的基本概念
- Continuous Batching — 连续批处理的详细原理
- Prefix Caching — 共享前缀的 KV 缓存复用

反向链接

学习日志

调度系统 — 练习

练习 **1**：调度行为模拟

假设系统配置如下：

- `max_num_seqs = 4`
- `max_num_batched_tokens = 4096`
- KV 缓存共 512 个 block，每个 block 16 tokens

时间线：

- T0：请求 A（prompt 2048 tokens）到达
- T1：请求 B（prompt 512 tokens）到达
- T2：请求 C（prompt 3072 tokens）到达

请模拟前 4 轮调度的决策（每轮 decode 生成 1 个 token）。

参考答案

- **Iter 1**：A prefill（2048 tokens，占用 128 blocks），生成 1 token。剩余 384 blocks。Running: [A]
- **Iter 2**：A decode（1 token），B prefill（512 tokens，占用 32 blocks）。剩余 352 blocks。Running: [A, B]
- **Iter 3**：A decode + B decode（各 1 token）。C prefill（3072 tokens，占用 192 blocks）。剩余 160 blocks。Running: [A, B, C]
- **Iter 4**：A decode + B decode + C decode（各 1 token）。如果某个请求完成则释放其 blocks，否则继续 decode。

注意：实际行为取决于 chunked prefill 的 chunk size 和显存碎片情况。

练习 2：Preemption 场景分析

在以下场景中，调度器会选择哪种 preemption 策略？

1. 10 个请求同时 active，KV 缓存占用 90%，新请求到达
2. 长请求和短请求混合，短请求已完成但长请求仍在 decode
3. 启用了 prefix caching，多个请求共享相同 system prompt

参考答案

1. **Re-computation**：抢占一个请求释放 KV 缓存。选择最后到达的请求（FCFS 策略下优先级最低）。由于 KV 缓存压力较大，恢复时需要重新 prefill。
2. 释放短请求的 **KV** 缓存：已完成的请求的 KV 缓存会被自然回收。如果仍然不够，抢占最新到达的请求。
3. 保留共享 **prefix**：prefix caching 使得共享前缀的 KV 缓存块有多个引用。抢占时选择引用计数最少的块释放，保留共享 prefix 的块。

练习 3：Chunked Prefill 参数调优

分析以下参数组合对 TTFT 和吞吐量的影响：

配置	max_num_batched_tokens	chunked_prefill
A	2048	off
B	2048	on
C	8192	on
D	32768	on

参考答案

配置	**TTFT**	吞吐量	适用场景
A	差（短 prompt 好处少）	差（长 prompt 阻塞 decode ）	短 prompt、低并发
B	好（chunk 小，快速响应）	中 (调度开销增加)	交互式聊天
C	中	好（平衡 prefill 和 decode ）	通用场景（默认）
D	差（大 chunk 等待久）	好（GPU 利用率高）	批量推理

拓展挑战

- 阅读 `vllm/v1/core/sched/scheduler.py` 的 `_schedule_prefills` 和 `_schedule_decodes` 方法
- 分析 `request_queue.py` 中的优先级调度实现
- 研究调度器如何处理 LoRA 请求的特殊调度需求

调度系统

> 深入理解 vLLM 的连续批处理调度器，它是系统的大脑，决定每轮迭代中请求的 prefill、decode 和 preemption 策略。

涵盖内容

章节	核心主题
概念	Continuous Batching、Chunked Prefill、Preemption、优先级
练习	调度行为模拟、参数调优
代码走读	scheduler.py 关键代码分析

核心概念

调度器 是 vLLM 的核心决策组件，每轮迭代决定：

- 哪些新请求开始 chunked prefill
- 哪些请求继续 decode
- 是否需要 preemption（抢占）以释放显存
- KV 缓存块的分配与回收

前置知识

- 引擎核心
- Paged Attention — 分页 KV 缓存的基本原理
- Continuous Batching — 连续批处理的概念

学习路径

读完本主题后，你将理解：

- 连续批处理如何最大化 GPU 利用率
- Chunked Prefill 如何平衡首 token 延迟和吞吐量
- Preemption 策略如何在显存压力下保持系统稳定
- 调度器如何与 KVCacheManager 协同管理缓存块

→ 下一步：KV缓存与PagedAttention

反向链接

架构概览 — 概念

引擎核心 — 概念

执行器与Worker

KV缓存与PagedAttention

调度系统

API服务与部署 — 代码走读

API Server — `vllm/entrypoints/openai/api_server.py`

```
# FastAPI 应用初始化
app = FastAPI()

# 中间件注册
app.add_middleware(CORSMiddleware, ...)

# 路由注册
app.include_router(chat_completion_router)
app.include_router(completion_router)
app.include_router(embedding_router)
app.include_router(responses_router)

# 引擎初始化
@app.on_event("startup")
async def startup():
    global engine
    engine = AsyncLLMEngine.from_engine_args(engine_args)
```

Chat Completion — `vllm/entrypoints/openai/chat_completion/`

```
@router.post("/v1/chat/completions")
async def create_chat_completion(request: ChatCompletionRequest):
    # 1. 解析请求
    generator = await chat_completion_handler(request, engine)

    # 2. 流式或非流式返回
    if request.stream:
        return StreamingResponse(generator, media_type="text/event-stream")
    else:
        return await generator.__anext__()
```

请求处理流程

```python
async def chat_completion_handler(request, engine):
    # 1. 渲染 chat template
    prompt = tokenizer.apply_chat_template(request.messages)

    # 2. 创建采样参数
    sampling_params = SamplingParams(
        temperature=request.temperature,
        top_p=request.top_p,
        max_tokens=request.max_tokens,
        stop=request.stop,
    )

    # 3. 提交到引擎
    result_generator = engine.generate(prompt, sampling_params, request_id)

    # 4. 流式返回
    async for result in result_generator:
        chunk = ChatCompletionStreamResponse(
            choices=[{
                "delta": {"content": result.outputs[0].text},
                "finish_reason": result.outputs[0].finish_reason,
            }],
        )
        yield f"data: {chunk.json()}\n\n"
```

LLM 离线类 — `vllm/entrypoints/llm.py`

```python
# Mixin 组合架构
class LLM(BeamSearchOfflineMixin, PoolingOfflineMixin, OfflineInferenceMixin):
    def __init__(self, model, **kwargs):
        engine_args = EngineArgs(model=model, **kwargs)
        self.llm_engine = LLMEngine.from_engine_args(engine_args)

    # generate() 来自 OfflineInferenceMixin
    # beam_search() 来自 BeamSearchOfflineMixin
```

OfflineInferenceMixin — `vllm/entrypoints/offline_utils.py`

从 LLM 类中提取的离线推理核心逻辑：

```python
class OfflineInferenceMixin:
    def _preprocess_cmpl(self, prompts, params):
        """预处理 completion 请求"""
        ...

    def _preprocess_chat(self, prompts, params):
        """预处理 chat 请求"""
        ...

    def _run_engine(self, *, lora_requests, prompt_adapter_requests):
        """运行引擎直到所有请求完成"""
        while self.llm_engine.has_unfinished_requests():
            step_outputs = self.llm_engine.step()

    def generate(self, prompts, sampling_params):
        """核心生成方法"""
        self._add_completion_requests(prompts, sampling_params)
        self._run_engine()
        return self._collect_outputs()
```

Beam Search 拆分

Beam search 也被拆分为独立模块：

- `entrypoints/generate/beam_search/offline.py` — `BeamSearchOfflineMixin`（LLM 类使用）
- `entrypoints/generate/beam_search/online.py` — `BeamSearchOnlineMixin`（OpenAIServing 使用）
- `entrypoints/generate/beam_search/utils.py` — 共享数据结构和工具函数

CLI 入口 — `vllm/entrypoints/cli/main.py`

```python
def main():
    parser = argparse.ArgumentParser()
    subparsers = parser.add_subparsers()

    # serve 子命令
    serve_parser = subparsers.add_parser("serve")
    serve_parser.set_defaults(func=run_server)

    args = parser.parse_args()
    args.func(args)
```

前端管理 — `vllm/v1/utils.py`

RustFrontendProcessManager

```
class RustFrontendProcessManager:
    """管理 Rust 前端子进程"""
    def __init__(self, ...):
        # 启动 vllm-rs 二进制文件
        self.proc = subprocess.Popen(
            [rust_frontend_path, "frontend",
             "--args-json", serialized_args],
            pass_fds=[listening_socket_fd],
        )
```

前端模式选择 — `vllm/entrypoints/cli/serve.py`

```
if VLLM_RUST_FRONTEND_PATH:
    # Rust 前端：通过 RustFrontendProcessManager 管理
    frontend = RustFrontendProcessManager(...)
elif data_parallel_multi_port_external_lb:
    # DP Supervisor：为每个 DP rank 生成独立 server
    run_dp_supervisor(...)
else:
    # 标准 Python API Server
    run_api_server(...)
```

DP Supervisor — `vllm/entrypoints/openai/dp_supervisor.py`

```
class DPSupervisor:
    """节点本地数据并行管理器"""
    def __init__(self, ...):
        # 为每个 DP rank 创建子进程
        self.children = []
        for rank in range(dp_size):
            port = base_port + rank
            env = {"CUDA_VISIBLE_DEVICES": str(rank)}
            proc = subprocess.Popen(
                [python, "-m", "vllm.entrypoints.openai.api_server",
                 "--port", str(port)],
                env=env,
            )
            self.children.append(proc)

    async def health_check(self):
        """聚合所有子进程的健康状态"""
        for child_url in self.child_urls:
            resp = await probe(child_url + "/health")
            ...
```

关键函数索引

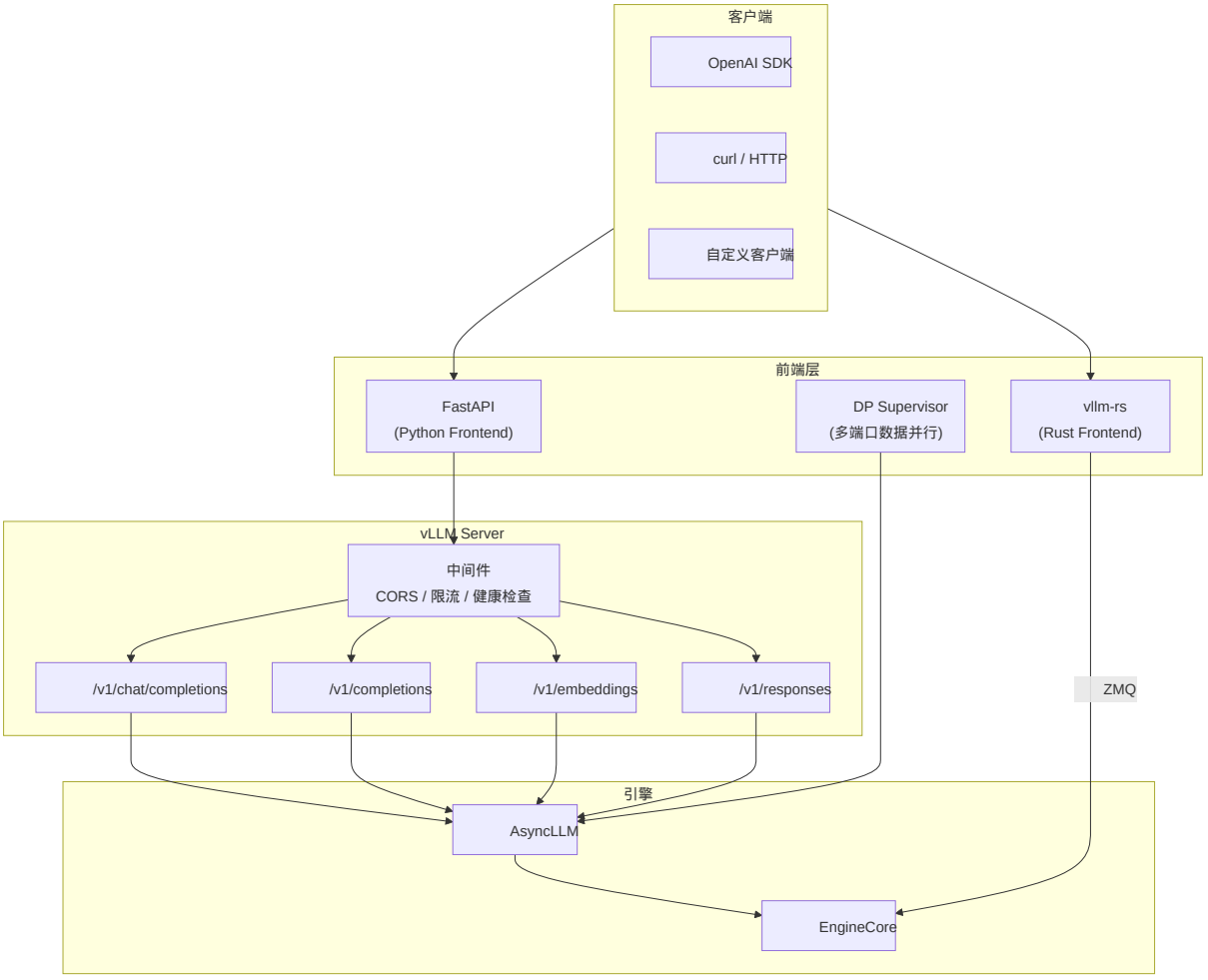
函数/类	文件	职责
`create_chat_completion()`	openai/chat_completion/	Chat Completion 端点
`create_completion()`	openai/completion/	Text Completion 端点
`LLM.generate()`	entrypoints/llm.py	离线批量推理
`OfflineInferenceMixin`	entrypoints/offline_utils.py	离线推理核心逻辑
`BeamSearchOfflineMixin`	entrypoints/generate/beam_search/offline.py	离线 beam search
`RustFrontendProcessManager`	v1/utils.py	Rust 前端子进程管理
`DPSupervisor`	entrypoints/openai/dp_supervisor.py	多端口数据并行管理
`LLMEngine.step()`	v1/engine/llm_engine.py	单步推理执行
`EngineArgs`	engine/arg_utils.py	CLI 参数解析
`run_server()`	entrypoints/cli/main.py	启动 API 服务器

反向链接

学习日志

API服务与部署 — 概念

服务架构



前端模式

vLLM 现在支持三种前端模式：

模式	实现	触发条件	适用场景
Python FastAPI	`APIServerProcessManager`	默认	通用场景
Rust Frontend	`RustFrontendProcessManager`	`VLLM_RUST_FRONTEND_PATH` 环境变量	高性能低延迟
DP Supervisor	`dp_supervisor.py`	`--data-parallel-multi-port-external-lb`	多 GPU 数据并行

OpenAI 兼容 API

Chat Completions

```
curl http://localhost:8000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "meta-llama/Llama-3-8B",
    "messages": [
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "Hello!"}
    ],
    "temperature": 0.7,
    "max_tokens": 256,
    "stream": true
  }'
```

Text Completions

```
curl http://localhost:8000/v1/completions \
  -d '{
    "model": "meta-llama/Llama-3-8B",
    "prompt": "Once upon a time",
    "max_tokens": 100
  }'
```

Embeddings

```
curl http://localhost:8000/v1/embeddings \
  -d '{
    "model": "BAAI/bge-large-en",
    "input": "Hello world"
  }'
```

支持的参数

参数	说明	默认值
`temperature`	采样温度	1.0
`top_p`	核采样概率	1.0
`top_k`	Top-k 采样	-1
`max_tokens`	最大生成 token 数	模型最大
`stop`	停止 token 列表	[]
`stream`	是否流式输出	false
`n`	生成候选数	1
`presence_penalty`	存在惩罚	0.0
`frequency_penalty`	频率惩罚	0.0
`logprobs`	返回 logprobs	false
`response_format`	结构化输出格式	null
`guided_json`	JSON Schema 约束	null

离线 LLM 类

```
from vllm import LLM, SamplingParams

llm = LLM(model="meta-llama/Llama-3-8B")
params = SamplingParams(temperature=0.7, max_tokens=256)

# 单个推理
output = llm.generate("Hello, world!", params)

# 批量推理
outputs = llm.generate(["prompt1", "prompt2", "prompt3"], params)

for output in outputs:
    print(output.outputs[0].text)
```

Mixin 架构

LLM 类从单一庞大类重构为 Mixin 组合架构：

```
LLM(BeamSearchOfflineMixin, PoolingOfflineMixin, OfflineInferenceMixin)
```

- **OfflineInferenceMixin** (`entrypoints/offline_utils.py`): 核心推理逻辑 — 预处理、参数广播、引擎运行
- **BeamSearchOfflineMixin** (`entrypoints/generate/beam_search/offline.py`): 离线 beam search
- **PoolingOfflineMixin**: 池化操作 (embeddings 等)

推测解码快捷参数

```
# 之前：需要传递完整的 JSON 配置
llm = LLM(model="...", speculative_config='{"method":"ngram","num_speculative_to
kens":5}')

# 现在：直接通过 CLI 参数设置
llm = LLM(model="...", spec_method="ngram", spec_tokens=5)
```

LLM 类参数

参数	说明
`model`	模型名称或路径
`tensor_parallel_size`	张量并行数
`gpu_memory_utilization`	GPU 显存利用率 (0-1)
`max_model_len`	最大序列长度
`quantization`	量化方法
`dtype`	数据类型
`trust_remote_code`	是否信任远程代码
`spec_method`	推测解码方法
`spec_model`	推测解码草稿模型
`spec_tokens`	推测解码候选 token 数

中间件

CORS

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)
```

限流

```
# 通过 --rate-limit-interval 和 --rate-limit-token 配置
```

健康检查

```
GET /health    → 200 OK (引擎就绪)
GET /ready     → 200 OK (服务就绪)
GET /v1/models → 返回可用模型列表
```

性能调优

关键配置参数

参数	影响	建议值
`--max-num-seqs`	最大并发请求数	128-512
`--max-num-batched-tokens`	每步最大 token 数	8192-32768
`--gpu-memory-utilization`	GPU 显存利用率	0.85-0.95
`--block-size`	KV 缓存块大小	16
`--enable-prefix-caching`	启用前缀缓存	建议开启
`--enable-chunked-prefill`	启用分块 prefill	建议开启

性能指标

指标	含义	优化方向
TTFT	首 token 延迟	降低 prefill 时间
TPOT	每 token 延迟	减少 decode 开销
Throughput	吞吐量 (tokens/s)	增加批处理量
GPU Utilization	GPU 利用率	平衡 prefill/decode

部署方式

Docker

```
docker run --gpus all \
  -v ~/.cache/huggingface:/root/.cache/huggingface \
  -p 8000:8000 \
  vllm/vllm-openai:latest \
  --model meta-llama/Llama-3-8B
```

多 GPU

```
vllm serve meta-llama/Llama-3-70B \
  --tensor-parallel-size 4 \
  --gpu-memory-utilization 0.9
```

Rust Frontend（高性能）

```
# 设置 Rust 前端路径
export VLLM_RUST_FRONTEND_PATH=/path/to/vllm-rs
vllm serve meta-llama/Llama-3-8B
```

Rust 前端与 Python API Server 共享相同的 ZMQ EngineCore 协议，通过 `RustFrontendProcessManager` 管理。它以子进程方式启动 `vllm-rs` 二进制文件，通过管道和 ZMQ 进行通信。

DP Supervisor（多端口数据并行）

```
vllm serve meta-llama/Llama-3-8B \
  --data-parallel-size 4 \
  --data-parallel-multi-port-external-lb
```

DP Supervisor 是一个节点本地管理进程：

- 为每个数据并行 rank 生成独立的 vLLM API Server（使用不同端口）
- 提供聚合的健康检查端点（`/health`、`/health/liveness`、`/health/readiness`）
- 自动为子进程设置 `CUDA_VISIBLE_DEVICES`
- 支持可配置的探测间隔和失败阈值

相关概念

- Continuous Batching — 连续批处理
- Paged Attention — 显存管理
- Speculative Decoding — 推测解码加速
- Tensor Parallelism — 多 GPU 并行
- topics/distributed/ — 数据并行、专家并行

反向链接　　出链

API服务与部署 — 练习

练习 1：API 服务器配置

为以下场景配置 vLLM 服务器：

1. 单 GPU（A100-80GB），运行 13B 模型，交互式聊天
2. 4 GPU（A100-80GB），运行 70B 模型，高吞吐 API 服务
3. 单 GPU（RTX 3090-24GB），运行 7B 模型，量化推理

参考答案

1. 单 **GPU** 交互式聊天：

```
vllm serve model-name \
  --max-model-len 4096 \
  --enable-prefix-caching \
  --enable-chunked-prefill \
  --gpu-memory-utilization 0.9
```

2. **4 GPU** 高吞吐：

```
vllm serve model-name \
  --tensor-parallel-size 4 \
  --max-num-seqs 256 \
  --max-num-batched-tokens 32768 \
  --enable-prefix-caching \
  --gpu-memory-utilization 0.95
```

3. 单 **GPU** 量化：

```
vllm serve model-name \
  --quantization awq \
  --max-model-len 2048 \
  --gpu-memory-utilization 0.85
```

练习 2：性能调优分析

分析以下性能瓶颈的解决方案：

1. TTFT 过高（>2s）
2. 吞吐量低（<100 tokens/s）
3. OOM 错误频繁发生
4. GPU 利用率低（<50%）

> 参考答案
>
> 1. **TTFT** 过高：启用 chunked prefill（减少单次 prefill 时间）、增大 `max-num-batched-tokens`、启用 prefix caching（复用 system prompt）。
> 2. 吞吐量低：增大 `max-num-seqs` 和 `max-num-batched-tokens`、启用 speculative decoding、减少单请求的 `max-tokens`。
> 3. **OOM**：降低 `gpu-memory-utilization`、减小 `max-model-len`、使用量化、减小 `max-num-seqs`。
> 4. **GPU** 利用率低：增大并发请求量、增大 batch token 预算、检查是否有通信瓶颈（TP 场景）。

拓展挑战

- 使用 vLLM 的 benchmark 工具测试不同配置的性能
- 研究结构化输出（JSON Schema）的配置和使用
- 分析多 LoRA 服务（同时服务多个 LoRA 适配器）的实现

API服务与部署

> 深入 vLLM 的服务层：OpenAI 兼容 API、离线推理、部署最佳实践和性能调优。

涵盖内容

章节	核心主题
概念	OpenAI API、离线推理、中间件、性能调优
练习	API 使用、部署配置、基准测试
代码走读	entrypoints 目录关键代码

核心概念

vLLM 提供多种使用方式：

- 在线服务：OpenAI 兼容的 API 服务器（`vllm serve`）
- 离线推理：`LLM` 类的同步批处理接口
- 嵌入服务：Embedding/Classification 的在线和离线接口
- 其他协议：gRPC、Anthropic Messages API

前置知识

- 架构概览
- HTTP API 基础知识
- Docker 容器化基础

学习路径

读完本主题后，你将理解：

- OpenAI 兼容 API 的完整接口和使用方式
- 离线 LLM 类的批处理推理接口
- API 服务器的中间件和性能配置
- 生产部署的最佳实践

→ 恭喜完成所有主题！回顾：知识地图

反向链接

引擎核心 — 概念

推测解码 — 代码走读

目录结构 — `vllm/v1/spec_decode/`

```
spec_decode/
├── llm_base_proposer.py     # LLM 草稿模型基类 (75K)
├── ngram_proposer.py         # N-gram 提议器
├── ngram_proposer_gpu.py     # GPU 加速 N-gram (25K)
├── eagle.py                  # EAGLE 实现
├── medusa.py                 # Medusa 实现
├── dflash.py                 # DFlash 实现
├── suffix_decoding.py        # 后缀解码
├── extract_hidden_states.py  # 隐藏状态提取
└── gemma4.py                 # Gemma4 特化实现
```

N-gram Proposer — `ngram_proposer_gpu.py`

```python
class NGramProposerGPU:
    """GPU 加速的 N-gram 推测"""

    def __init__(self, n=3, k=5):
        self.n = n          # n-gram 阶数
        self.k = k          # 候选 token 数
        self.trie = GPUTrie()  # GPU 上的 trie 结构

    def propose(self, input_ids, num_speculative_tokens):
        # 1. 在 trie 中查找最近的 n-gram 匹配
        matches = self.trie.lookup(input_ids[:, -(self.n-1):])

        # 2. 选择 top-k 候选
        candidates = matches.topk(self.k)

        # 3. 拼接候选 token
        speculative_tokens = candidates[:, :num_speculative_tokens]
        return speculative_tokens
```

EAGLE — `eagle.py`

```
class EAGLEProposer:
    """EAGLE 推测解码实现"""

    def __init__(self, target_model_config):
        # 创建轻量草稿模型
        self.draft_model = self._create_draft_model(target_model_config)

    def propose(self, hidden_states, input_ids):
        # 1. 从目标模型获取特征
        features = hidden_states[-1]  # 最后一层的隐藏状态

        # 2. 草稿模型自回归生成候选 token
        draft_tokens = []
        draft_probs = []
        current_input = features

        for _ in range(self.num_speculative_tokens):
            logits = self.draft_model(current_input)
            probs = softmax(logits)
            token = argmax(probs)  # 贪心选择

            draft_tokens.append(token)
            draft_probs.append(probs)
            current_input = self._embed_and_concat(current_input, token)

        return draft_tokens, draft_probs
```

Rejection Sampler — `vllm/v1/sample/rejection_sampler.py`

```
class RejectionSampler:
    """验证草稿 token 并决定接受/拒绝"""

    def forward(self, draft_tokens, draft_probs, target_probs):
        # 1. 计算每个 token 的接受概率
        accept_probs = torch.minimum(
            torch.ones_like(draft_probs),
            target_probs / draft_probs,
        )

        # 2. 采样接受/拒绝
        random_values = torch.rand_like(accept_probs)
        accepted = random_values < accept_probs

        # 3. 找到第一个被拒绝的位置
        rejected_mask = ~accepted
        first_reject = rejected_mask.float().argmax(dim=-1)

        # 4. 截断到第一个拒绝位置
        output_tokens = draft_tokens.clone()
        for i in range(draft_tokens.shape[0]):
            output_tokens[i, first_reject[i]:] = 0  # 清零被拒绝的 token

            # 从调整分布重新采样第一个被拒绝的 token
            if first_reject[i] < draft_tokens.shape[1]:
                adjusted = torch.clamp(
                    target_probs[i] - draft_probs[i], min=0
                )
                output_tokens[i, first_reject[i]] = sample(adjusted)

        return output_tokens
```

与 ModelRunner 的集成

推测解码在 ModelRunner 中通过 hooks 集成：

```python
class ModelRunner:
    def execute_model(self, scheduler_output):
        # 标准前向传播
        output = self._forward(input_batch)

        # 如果启用推测解码
        if self.speculative_config:
            # 1. 草稿阶段：生成候选 token
            draft_tokens = self.spec_proposer.propose(...)

            # 2. 验证阶段：在目标模型中验证
            verified = self.rejection_sampler(
                draft_tokens, draft_probs, output.logits,
            )

            output.speculative_tokens = verified

        return output
```

关键函数索引

函数/类	文件	职责
`NGramProposerGPU.propose()`	spec_decode/ngram_proposer_gpu.py	N-gram 候选生成
`EAGLEProposer.propose()`	spec_decode/eagle.py	EAGLE 草稿生成
`RejectionSampler.forward()`	sample/rejection_sampler.py	验证和接受/拒绝
`LLMBaseProposer`	spec_decode/llm_base_proposer.py	LLM 草稿模型基类
`extract_hidden_states()`	spec_decode/extract_hidden_states.py	提取目标模型特征

推测解码 — 概念

基本原理

自回归推理每步只生成 1 个 token，GPU 利用率很低。推测解码通过"猜"多个 token 来提高利用率：

数学原理

对于每个候选 token，计算接受概率：

```
accept_prob = min(1, p_target(token) / p_draft(token))
```

- 如果 `p_target > p_draft`：总是接受
- 如果 `p_target < p_draft`：以概率 `p_target/p_draft` 接受
- 被拒绝的 token 后重新采样，保证分布不变

推测方法

N-gram Proposer

最简单的方法，不需要额外模型：

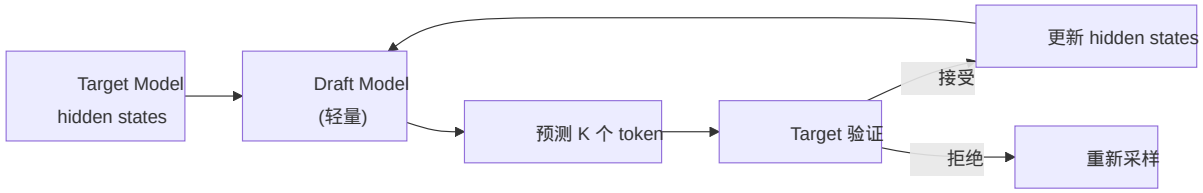
- 维护历史 n-gram 频率表
- 根据前 n-1 个 token 预测下一个 token

- GPU 加速版本使用 trie 结构高效查找

  优点：零额外计算开销 缺点：准确率低，接受率有限

EAGLE

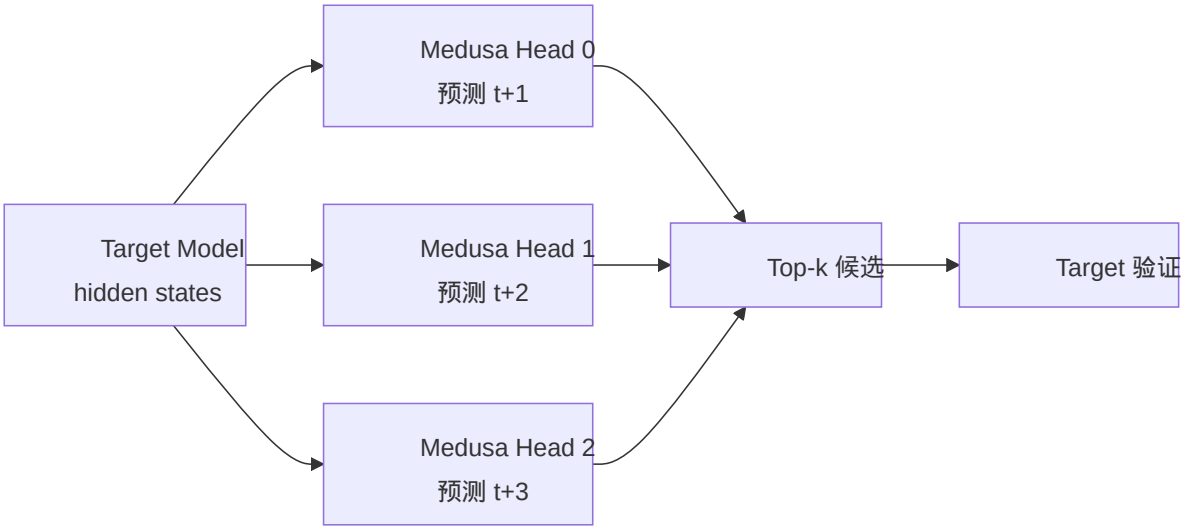
基于模型特征的方法：



- 利用目标模型的中间层特征
- 草稿模型很小（通常只有 1-2 层）
- 准确率高（85-95%），接受率高

Medusa

多头预测方法：



- 在目标模型上添加多个预测头
- 每个头预测不同未来位置的 token
- 不需要单独的草稿模型
- 训练成本低

方法对比

方法	额外模型	准确率	加速比	适用场景
N-gram	无	低	1.2-1.5×	简单场景
EAGLE	小模型	高	2-3×	通用
Medusa	多头	中高	1.5-2.5×	不想用草稿模型
DFlash	小模型	高	2-3×	Flash attention
Suffix Decoding	无	中	1.3-1.8×	重复性文本

Rejection Sampling

验证阶段的核心算法：

```
    def rejection_sample(draft_tokens, draft_probs, target_probs):
        accepted = []
        for i, token in enumerate(draft_tokens):
            p_draft = draft_probs[i][token]
            p_target = target_probs[i][token]

            # 接受概率
            accept_prob = min(1.0, p_target / p_draft)

            if random() < accept_prob:
                accepted.append(token)
            else:
                # 从调整后的分布中重新采样
                adjusted = max(0, p_target - p_draft)
                new_token = sample(adjusted)
                accepted.append(new_token)
                break  # 后续 token 全部丢弃

        return accepted
```

推测解码与连续批处理

推测解码需要与 Continuous Batching 协同工作：

- 每个请求可能有不同数量的候选 token
- 调度器需要为推测解码预留足够的计算预算

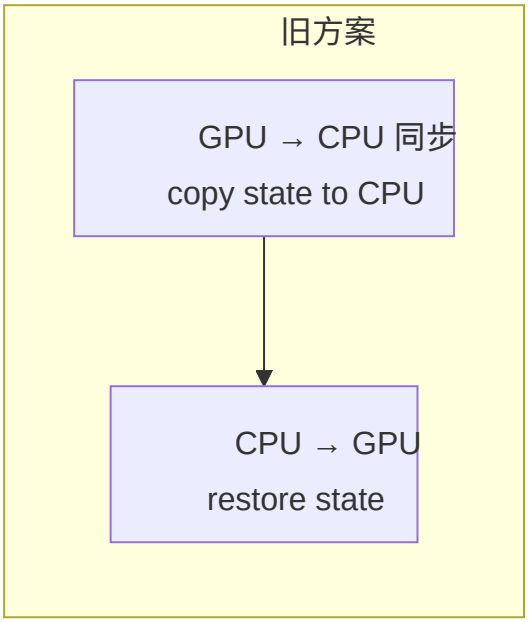
- 被拒绝的 token 的 KV 缓存需要回滚

混合模型的推测解码

Mamba + Attention 推测解码

混合模型（如 Mamba + Attention）的推测解码面临 CPU-GPU 同步瓶颈：Mamba 状态在验证失败后需要回滚，传统方法需要将状态从 GPU 拷贝到 CPU 再恢复。

vLLM 引入了融合 Triton 内核 `postprocess_mamba_fused_kernel`，完全在 GPU 上执行 Mamba 状态复制：



关键组件：

- **MambaSpecDecodeGPUContext**：预计算内存布局元数据（基地址、步幅、元素大小、卷积宽度）
- **postprocess_mamba_align_gpu()**：align 模式缓存回滚

- **postprocess_mamba_all()**：`mamba_cache_mode="all"` 模式回滚
- **MambaBuffers**：从 `MambaCopyBuffers` 重构，增加 postprocess 子对象

EAGLE-3 后规范架构

EAGLE-3 推测模型现在支持后规范（post-norm）架构，扩展了可适配的模型范围。

相关概念

- Continuous Batching — 连续批处理
- KV Cache — 推测解码的 KV 缓存回滚
- CUDA Graph — 推测解码对图捕获的影响

反向链接

学习日志

推测解码 — 练习

练习 **1**：加速比分析

假设：

- 目标模型每步推理需要 10ms
- 草稿模型每步推理需要 1ms
- 草稿模型预测 5 个 token
- 验证 5 个 token 也需要 10ms（并行）
- 平均接受率 80%

计算：

1. 传统解码生成 10 个 token 的时间
2. 推测解码生成 10 个 token 的时间（平均）
3. 加速比

参考答案

1. 传统解码：10 × 10ms = 100ms
2. 推测解码：
   - 每轮：草稿 5 × 1ms = 5ms + 验证 10ms = 15ms
   - 接受率 80%，每轮平均接受 5 × 0.8 = 4 个 token（简化计算）
   - 更精确：P(接受k个) = 0.8^k × 0.2（第k+1个被拒绝）
   - 平均每轮接受约 4 个 token
   - 生成 10 个 token 约需 10/4 = 2.5 轮
   - 2.5 × 15ms = 37.5ms
3. 加速比：100 / 37.5 ≈ 2.67×

练习 2：方法选择

为以下场景选择最合适的推测解码方法：

1. 低延迟对话场景，目标模型 7B
2. 批量推理场景，高吞吐优先
3. 硬件资源受限（单 GPU，显存紧张）

<details>
<summary>参考答案</summary>

1. **EAGLE**：准确率高，加速效果好。7B 模型的草稿模型很小（~100M 参数），额外显存开销小。
2. **N-gram** 或 **Medusa**：批量场景中推测解码的效果会降低（因为 batch 中各请求的接受率不同导致 padding 浪费）。N-gram 无额外计算开销最合适。Medusa 可以用 top-k 树验证，适合批量场景。
3. **N-gram**：零额外显存开销。或者 Medusa（只需添加几个线性头，显存开销很小）。EAGLE 需要额外的草稿模型，可能显存不够。

</details>

拓展挑战

- 阅读 `vllm/v1/spec_decode/rejection_sampler.py`，理解树形推测的验证逻辑
- 分析 EAGLE 的 hidden state 复用机制
- 研究推测解码如何处理结构化输出（JSON grammar）

推测解码

> 深入 vLLM 的 推测解码实现：通过快速草稿模型预测多个 token，再由目标模型验证，实现无损加速。

涵盖内容

章节	核心主题
概念	推测解码原理、N-gram、EAGLE、Medusa、Rejection Sampling
练习	加速比分析、方法对比
代码走读	spec_decode 目录关键代码

核心概念

Speculative Decoding 是一种加速自回归推理的技术：

- 草稿阶段：快速模型（或启发式方法）生成 K 个候选 token
- 验证阶段：目标模型并行验证所有候选 token
- 接受/拒绝：使用 Rejection Sampling 决定接受哪些 token

前置知识

- 模型库与算子层
- 自回归语言模型的基本原理
- 概率论基础（条件概率、采样）

学习路径

读完本主题后，你将理解：

- 推测解码为什么能实现无损加速
- N-gram、EAGLE、Medusa 等不同方法的优缺点
- Rejection Sampling 的数学原理和实现
- 推测解码如何与 Continuous Batching 结合

→ 下一步：多模态处理