PR #43205 完整报告

vllm-project/vllm

[KV Offload] Add per-request offloading policy via `on_new_request` lifecycle hook

合并时间：2026-05-29 04:45

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/43205

执行摘要

- 一句话：新增 per-request 卸载策略和生命周期钩子
- 推荐动作：值得精读，特别是设计决策（抽象方法、只跟踪 REQUEST_LEVEL 层、命名选择）可供参考。建议关注 on_new_request 和级联逻辑的实现。

功能与动机

原有的 do_remote_decode 参数无法灵活支持多种卸载策略，如需完整 KV 上下文的次级层（如存储层）。关联 Issue #33689 推动引入统一的策略机制，使每个次级层能按需声明卸载粒度。

实现拆解

1. 基础抽象层（vllm/v1/kv_offload/base.py）：新增 OffloadPolicy 枚举（BLOCK_LEVEL / REQUEST_LEVEL）和 RequestOffloadingContext 数据类；在 OffloadingManager 基类添加抽象方法 on_new_request 和默认方法 on_request_finished。
2. 层次卸载管理器（vllm/v1/kv_offload/tiering/manager.py）：实现 on_new_request 轮询所有次级层，汇总策略；在 _request_level_tiers 中仅记录返回 REQUEST_LEVEL 的层；新增 _cascade_existing_blocks_to_request_level_tiers 方法，在 prepare_store 中将被主层缓存且未被本次存储的块立即级联到请求级层。
3. 调度器连接器（vllm/distributed/kv_transfer/kv_connector/v1/offloading/scheduler.py）：提取 _create_req_context 辅助函数，避免重复构造；新增 on_new_request 方法，在请求到达时调用 manager.on_new_request 获取策略，并创建包含 req_context 和 offloading_context 的 RequestOffloadState。原 get_num_new_matched_tokens 中的初始化逻辑被重构到新钩子中，并移除对 do_remote_decode 的依赖。
4. 次级层接口（vllm/v1/kv_offload/tiering/base.py）：在 SecondaryTierManager 中添加抽象方法 on_new_request 和默认方法 on_request_finished，使各次级层能表达策略偏好。
5. 测试配套：新增 test_on_new_request_lifecycle 和 test_prepare_store_cascades_existing_blocks_to_request_level_tiers 验证钩子调用和级联行为；调度器侧测试重命名为 test_request_level_policy_stores_all_blocks，并用 mock 的 on_new_request 返回策略而非写死 do_remote_decode。

关键文件：

- vllm/v1/kv_offload/base.py（模块 基础抽象；类别 source；类型 core-logic；符号 OffloadPolicy, RequestOffloadingContext, on_new_request, on_request_finished）：核心抽象，定义了 OffloadPolicy、RequestOffloadingContext 和生命周期钩子接口。
- vllm/v1/kv_offload/tiering/manager.py（模块 层次卸载管理器；类别 source；类型 core-logic；符号 on_new_request, on_request_finished, _cascade_existing_blocks_to_request_level_tiers, _request_level_tiers）：层次卸载管理器核心实现，新增 on_new_request、_cascade_existing_blocks_to_request_level_tiers。
- vllm/distributed/kv_transfer/kv_connector/v1/offloading/scheduler.py（模块 调度器；类别 source；类型 core-logic；符号 on_new_request, _create_req_context, RequestOffloadState）：调度器侧连接器，利用新钩子初始化请求卸载状态。
- vllm/v1/kv_offload/tiering/base.py（模块 次级层接口；类别 source；类型 core-logic；符号 on_new_request, on_request_finished）：SecondaryTierManager 抽象类添加 on_new_request 和 on_request_finished 钩子。
- tests/v1/kv_offload/test_tiering_offloading.py（模块 卸载测试；类别 test；类型 test-coverage；符号 test_on_new_request_lifecycle, test_prepare_store_cascades_existing_blocks_to_request_level_tiers）：测试 on_new_request 生命周期和现有块级联逻辑。
- tests/v1/kv_connector/unit/offloading_connector/test_scheduler.py（模块 调度测试；类别 test；类型 test-coverage；符号 test_request_level_policy_stores_all_blocks, test_do_remote_decode_stores_all_blocks）：测试调度器侧策略集成，重命名测试用例。

关键符号：OffloadingManager.on_new_request, OffloadingManager.on_request_finished, SecondaryTierManager.on_new_request, SecondaryTierManager.on_request_finished, TieringOffloadingManager.on_new_request, TieringOffloadingManager._cascade_existing_blocks_to_request_level_tiers, TieringOffloadingManager.prepare_store, OffloadingConnectorScheduler.on_new_request, _create_req_context

关键源码片段

vllm/v1/kv_offload/base.py

核心抽象，定义了 OffloadPolicy、RequestOffloadingContext 和生命周期钩子接口。

```
# 文件：vllm/v1/kv_offload/base.py (head)
# 新增 OffloadPolicy 枚举和 RequestOffloadingContext 数据类
from enum import Enum

class OffloadPolicy(Enum):
    # 块级策略：只卸载新计算的块，前缀命中跳过（默认）
    BLOCK_LEVEL = "block_level"
    # 请求级策略：卸载请求所有块，包括前缀命中
    # 用于需要完整 KV 上下文的次级层（如存储层）
    REQUEST_LEVEL = "request_level"

@dataclass
class RequestOffloadingContext:
```

```
    # 每个请求的卸载策略，默认 BLOCK_LEVEL
    policy: OffloadPolicy = OffloadPolicy.BLOCK_LEVEL

# OffloadingManager 新增生命周期钩子（ABC）
class OffloadingManager(ABC):
    @abstractmethod
    def on_new_request(self, req_context: ReqContext) -> RequestOffloadingContext:
        """新请求到达时调用，返回该请求的卸载策略。"""
        ...

    def on_request_finished(self, req_context: ReqContext) -> None:
        """请求结束时清理，默认为空操作。"""
        return
```

vllm/v1/kv_offload/tiering/manager.py

层次卸载管理器核心实现，新增 on_new_request、_cascade_existing_blocks_to_request_level_tiers。

```
# 文件：vllm/v1/kv_offload/tiering/manager.py (head)
# TieringOffloadingManager 新增关键方法

def on_new_request(self, req_context: ReqContext) -> RequestOffloadingContext:
    """轮询所有次级层策略，若任一为 REQUEST_LEVEL 则返回 REQUEST_LEVEL。"""
    # 收集所有次级层的策略偏好
    tier_policies = {
        tier: tier.on_new_request(req_context)
        for tier in self.secondary_tiers
    }
    # 仅保留请求 REQUEST_LEVEL 的层供 prepare_store 使用
    request_level_tiers = {
        tier for tier, ctx in tier_policies.items()
        if ctx.policy == OffloadPolicy.REQUEST_LEVEL
    }
    if request_level_tiers:
        self._request_level_tiers[req_context.req_id] = request_level_tiers
        return RequestOffloadingContext(policy=OffloadPolicy.REQUEST_LEVEL)
    return RequestOffloadingContext(policy=OffloadPolicy.BLOCK_LEVEL)

def prepare_store(self, keys, req_context):
    # 步骤 1: 处理完成的作业
    self._maybe_process_finished_jobs()
    # 步骤 2: 存储到主层（只存储新块）
    primary_result = self.primary_tier.prepare_store(keys, req_context)
    if primary_result is None:
        return None
    # 步骤 3: 对请求级层，级联已在主层的块
    request_level_tiers = self._request_level_tiers.get(req_context.req_id)
    if request_level_tiers is not None:
        keys_to_store_set = set(primary_result.keys_to_store)
```

```
        keys_already_in_primary = [k for k in keys if k not in keys_to_store_set]
        if keys_already_in_primary:
            self._cascade_existing_blocks_to_request_level_tiers(
                keys_already_in_primary, req_context, request_level_tiers
            )
    return primary_result

def _cascade_existing_blocks_to_request_level_tiers(
    self,
    keys: Sequence[OffloadKey],
    req_context: ReqContext,
    request_level_tiers: set[SecondaryTierManager],
) -> None:
    # 过滤出主层已就绪的块（可查询到）
    ready_keys = tuple(
        k for k in keys if self.primary_tier.lookup(k, req_context) is not None
    )
    if not ready_keys:
        return
    # 对每个请求级层，创建 JobMetadata 并提交 submit_store
    for tier in request_level_tiers:
        block_ids = np.array(
            [self.primary_tier.get_block_id(k, req_context) for k in ready_keys],
            dtype=np.int64,
        )
        job_metadata = JobMetadata(
            job_id=self._next_job_id(),
            keys=ready_keys,
            block_ids=block_ids,
            is_promotion=False, # cascade 方向：primary → secondary
            req_context=req_context,
        )
        tier.submit_store(job_metadata)
```

vllm/distributed/kv_transfer/kv_connector/v1/offloading/scheduler.py

调度器侧连接器，利用新钩子初始化请求卸载状态。

```
# 文件：vllm/distributed/kv_transfer/kv_connector/v1/offloading/scheduler.py (head)
# 新增 on_new_request 方法和辅助函数

def _create_req_context(req: Request) -> ReqContext:
    """从 Request 构造 ReqContext，统一封装请求 ID 和传输参数。"""
    return ReqContext(
        req_id=req.request_id,
        kv_transfer_params=req.kv_transfer_params,
    )

class OffloadingConnectorScheduler:
    def on_new_request(self, request: Request) -> None:
```

```
"""当调度器收到新请求时调用，初始化卸载状态。"""
# 1. 构造请求上下文
req_context = _create_req_context(request)
# 2. 调用管理器的钩子获取策略
offloading_context = self.manager.on_new_request(req_context)
# 3. 创建请求卸载状态，包括上下文和策略
req_status = RequestOffloadState(
    config=self.config,
    req=request,
    req_context=req_context,
    offloading_context=offloading_context,
)
# 4. 注册到内部状态字典
self._req_status[request.request_id] = req_status
```

评论区精华

命名混淆：ronensc 指出 `request_finished` 与 `get_finished` 容易混淆，orozery 建议改用 `on_request_finished` 或 `handle_request_finished`。最终重命名为 `on_request_finished`，并将 `get_request_offloading_context` 同步改为 `on_new_request`。

抽象方法决策：orozery 建议将 `on_new_request` 设为抽象以强制实现，ronensc 照做并同步更新 CPUOffloadingManager 和 SecondaryTierManager。

性能优化：orozery 指出若全部层均为 BLOCK_LEVEL 则存储所有策略集合浪费，建议只跟踪 REQUEST_LEVEL 的层。ronensc 改用 `_request_level_tiers`（defaultdict）仅记录请求级层。

内存泄漏修复：gemini-code-assist[bot] 指出若 `req_status` 在 `manager.on_request_finished` 前被删除则钩子可能跳过。ronensc 确保在删除 `req_status` 前触发钩子。

- on_request_finished 与 get_finished 命名混淆 (design): 重命名为 on_request_finished，并将 get_request_offloading_context 同步改为 on_new_request。
- on_new_request 是否应为抽象方法 (design): on_new_request 变为抽象方法，所有 OffloadingManager 和 SecondaryTierManager 的子类必须实现。
- prepare_store 中只存储 REQUEST_LEVEL 层避免浪费 (performance): 使用 _request_level_tiers 仅存储请求 REQUEST_LEVEL 的层，减少性能开销。
- request_finished 钩子可能因 req_status 过早删除而跳过 (correctness): 在 update_connector_output 中删除 req_status 前调用 self.manager.request_finished。

风险与影响

- 风险：核心路径变更：prepare_store 和 on_new_request 影响卸载决策，可能破坏现有次级层行为。性能影响：REQUEST_LEVEL 策略可能导致更多块卸载（包括前缀命中），增加网络或存储开销。兼容性：旧 do_remote_decode 参数被移除，依赖该参数的代码需迁移至 on_new_request 返回策略。风险集中在 vllm/v1/kv_offload/tiering/manager.py 和 scheduler.py，但测试覆盖了主要场景。

- 影响：用户透明（API 不变）；开发者需为自定义次级层实现 on_new_request 抽象方法；系统行为变化：启用 REQUEST_LEVEL 的次级层将接收所有块（包括前缀命中），可能导致更密集的卸载流量。对 P/D 分离场景有益。影响范围限于 KV offload 相关模块。
- 风险标记：核心路径变更，性能开销风险，兼容性风险

关联脉络

- PR #43870 [KV Offload] Rename SecondaryTierManager.get_finished() to get_finished_jobs(): 该 PR 进一步解决命名混淆，将 get_finished 重命名为 get_finished_jobs，与当前 PR 的命名讨论直接关联。