PR #7373 完整报告

verl-project/verl

[trainer, vllm] feat: lend idle trainer GPUs to generation in separate_async

合并时间：2026-08-21 21:46

原文链接：http://prhub.com.cn/verl-project/verl/pull/7373

执行摘要

- 一句话：separate_async 训练空闲 GPU 借给生成，wall clock 提速 12.6%
- 推荐动作：值得精读，尤其是两个设计点：① 自适应阈值调度——用「滞回步进 + 连续步数释放」的最小状态量避免频繁切换，并用滑动窗口切换成本 + 静态 scaling factor 建模收益，是资源借调类功能的经典范式；② 在 vLLM 无 KV-only release API 的约束下，用 sleep(2)+wake_up(tags) 组合近似目标语义，并把替代路径以 TODO 形式传递给上游，体现了务实的上游协同方式。对计划优化 separate_async 资源利用率、或在自研 rollout 引擎上做「借卡」能力的工程师，本 PR 是很好的参考实现。

功能与动机

PR body 明确指出动机是 reduce trainer idle：在 separate_async 中，trainer 提交完本 step 的 prompts 后，hybrid GPU 直到 replay buffer 攒够可采样组之前都处于纯空闲状态，而 standalone rollout 的吞吐可能成为瓶颈。PR 给出的基准数据为 150 步 wall clock 从 18.80 h 降到 16.43 h（-12.6%），tokens/s 从 15 150 提升到 16 687（+10.1%）。阈值自适应思想来自 #6556 为 fully_async 引入的动态资源调度，本 PR 将其应用到 separate_async 的 step 边界。此外 vllm#44890 的 motivation 中直接引用本 PR："The workaround in verl#7373 uses sleep(level=2) followed by wake_up(tags=["weights"]) before updating the standalone model"，说明这是上游 release_kv_cache() API 的第一个真实使用场景。

实现拆解

实现分为 5 步：

1. 配置与校验层：在 verl/trainer/config/config.py 新增 HybridRolloutSwitchConfig dataclass（含 enable_switch、switch_threshold_ratio、adaptive_switch_threshold、switch_threshold_step_up/down、switch_threshold_release_steps、switch_cost_window_size），__post_init__ 校验阈值必须落在 (0, 1]、成本窗口必须为正数。ppo_trainer.yaml 与 4 个 _generated_* 配置同步新增 hybrid_rollout 配置组（_target_ 指向该 dataclass）。PPOTrainerSeparateAsync.__init__ 中解析该配置，并做两项硬校验：开启时拒绝 PD 分离 rollout（ValueError），且强制 replay buffer 必须实现 wait_for_sampleable 与 get_sampleable_count（TypeError）。
2. ReplayBuffer 深度接口：verl/trainer/ppo/v1/replay_buffer.py 新增 get_sampleable_count（只读统计当前可采样组数，不消费）与 wait_for_sampleable（轮

询直到达到 target_count，等待期间照常执行过期 /DAPO/ 失败组的淘汰与 refill），并把 sample 重构为对 wait_for_sampleable 的封装：先等深度达标，再 _select_prompt_uids 选组。行为等价，但让切换到 trainer 前的「预判存量」成为可能。

3. Trainer 步级切换状态机：verl/trainer/ppo/v1/trainer_separate_async.py 中 on_step_begin 先重置统计窗口，若已处于 ROLLOUT 模式则计算 _switch_threshold，buffer 深度达标就立即 _timed_switch_to_trainer 回收；未达标则保持借出状态。prepare_step 在提交 prompts 后调用 _wait_for_sampleable_and_switch 阻塞等待生成填充 buffer，随后回收引擎并把等待期间的 eviction 指标并入 step 指标。on_sample_begin/on_sample_end 计量 mini-batch 饥饿时长，on_step_end 通过 update_weights + resume_generation_replicas 把引擎借给下一轮生成。trainer_base.py 新增可覆写的 prepare_step 钩子，默认行为与原来完全一致。
4. 自适应阈值与成本建模：_switch_threshold 以 switch_threshold_ratio × train_batch_size 为目标、下限封底为 1 个 mini-batch；_step_had_idle 依据 poll_interval 判断上一轮是否饥饿；_adapt_switch_threshold 采用滞回步进（连续 idle/calm 达到 release_steps 才调整），饥饿上调、平静下调；_effective_switch_cost 用滑动窗口（switch_cost_window_size）统计往返切换成本，结合 _scaling_factor（hybrid+standalone 总 GPU 与 standalone GPU 之比）建模借卡收益，决策指标输出为 separate_async/decision/* 系列。
5. vLLM 显存 workaround 与配套：verl/workers/rollout/vllm_rollout/vllm_async_server.py 中 release_kv_cache 从空实现改为「sleep(level=2) + wake_up(tags=["weights"])」，即先丢弃全部 GPU 内存再按标签恢复权重，模拟只释放 KV cache；resume_kv_cache 改为「wake_up(tags=["kv_cache"]) + reset_prefix_cache」；抽出 _resolve_sleep_level 统一 MTP/LoRA/NPU 场景的 level 选择，COLOCATED 模式显式跳过。配套新增 471 行 CPU 单测、ReplayBuffer 4 个新用例、300 行文档 docs/advance/v1_async_trainer.md，并同步 Prometheus 配置以覆盖 hybrid + standalone 全部 server 地址。

关键文件：

- verl/trainer/ppo/v1/trainer_separate_async.py（模块 训练器；类别 source；类型 core-logic；符号 _init_hybrid_rollout_state, on_step_begin, _timed_switch_to_trainer, prepare_step）：核心变更文件：实现 hybrid 引擎的步级借出 / 收回状态机、自适应阈值、成本建模与决策指标，是全部切换逻辑的宿主。
- verl/trainer/ppo/v1/replay_buffer.py（模块 回放缓冲；类别 source；类型 core-logic；符号 sample, get_sampleable_count, wait_for_sampleable）：新增 get_sampleable_count 与 wait_for_sampleable 两个 buffer 深度只读接口，sample 重构为对它们的封装，是本 PR 借卡决策的数据基础。
- verl/workers/rollout/vllm_rollout/vllm_async_server.py（模块 推理服务；类别 source；类型 core-logic；符号 release_kv_cache, resume_kv_cache, _resolve_sleep_level, _sleep_hybrid）：release_kv_cache/resume_kv_cache 从空实现变为 sleep(level)+wake_up(tags) 组合模拟，是避免权重同步 OOM 的关键 workaround，并抽出 _resolve_sleep_level 统一 level 选择。
- tests/trainer/ppo/v1/test_separate_async_step_switch_on_cpu.py（模块 单元测试；类别 test；类型 test-coverage；符号 _RecordingCheckpointManager,

_RecordingReplayBuffer, _trainer, _construct_trainer）：新增 471 行 CPU 单测，用 _RecordingCheckpointManager 和 _RecordingReplayBuffer 锁定钩子调用序列，验证每次 global step 恰好一次往返借卡契约。

- tests/trainer/ppo/v1/test_replay_buffer_on_cpu.py（模块 单元测试；类别 test；类型 test-coverage；符号 WaitConsumer, test_get_sampleable_count_excludes_stale_groups_without_consuming, test_wait_for_sampleable_leaves_the_groups_for_sample, test_wait_for_sampleable_blocks_below_the_target）：新增 WaitConsumer 与 4 个用例覆盖 wait_for_sampleable 的阻塞、不消费、淘汰补齐语义，保护重构后的 sample 行为。
- verl/trainer/config/config.py（模块 配置；类别 source；类型 configuration；符号 HybridRolloutSwitchConfig, post_init）：新增 HybridRolloutSwitchConfig 配置契约，__post_init__承担阈值与窗口校验，是 review 推动的参数分组结果。
- verl/trainer/ppo/v1/trainer_base.py（模块 训练器；类别 source；类型 core-logic；符号 prepare_step）：新增 prepare_step 可覆写钩子，默认转发 _add_batch_to_generate 并返回空指标，是 separate_async 接入等待逻辑的基类扩展点。
- docs/advance/v1_async_trainer.md（模块 文档；类别 docs；类型 documentation）：新增 300 行文档，系统介绍 V1 异步训练模式、partial rollout、off-policy 控制与 step switch 的关系，是理解本 PR 设计前提的最佳入口。
- verl/trainer/config/ppo_trainer.yaml（模块 配置；类别 config；类型 configuration）：主配置新增 hybrid_rollout 配置组并默认关闭，是功能门控的声明位置。

关键符号：on_step_begin, _wait_for_sampleable_and_switch, _timed_switch_to_trainer, _switch_threshold, _step_had_idle, _adapt_switch_threshold, _effective_switch_cost, prepare_step, get_sampleable_count, wait_for_sampleable, sample, release_kv_cache, resume_kv_cache, _resolve_sleep_level, _sleep_hybrid, HybridRolloutSwitchConfig.post_init

关键源码片段

verl/trainer/ppo/v1/trainer_separate_async.py

核心变更文件：实现 hybrid 引擎的步级借出 / 收回状态机、自适应阈值、成本建模与决策指标，是全部切换逻辑的宿主。

```
def on_step_begin(self):
    # 每个 global step 开始时重置上一轮的等待统计，防止旧数据污染本轮决策
    self._step_sample_wait_seconds = 0.0
    self._step_wait_samples = 0
    self._step_threshold = 0
    if self.hybrid_rollout_config.enable_switch:
        self.timing_raw["switch_wait"] = 0.0
    # 引擎已收回（上一轮 mini-batch 正在训练）时无需处理
    if self.current_mode != HybridEngineMode.ROLLOUT:
        return
    # 开关关闭时保持历史行为：step 一开始立即回收 GPU 训练
    if not self.hybrid_rollout_config.enable_switch:
        self._timed_switch_to_trainer()
```

```
        return

    # 计算本轮阈值：只有 buffer 存量足够时才回收，否则继续把 GPU 借给生成
    self._step_threshold = self._switch_threshold()
    sampleable_count = self.replay_buffer.get_sampleable_count(self.global_steps, "train")
    if sampleable_count >= self._step_threshold:
        self._timed_switch_to_trainer()


def _wait_for_sampleable_and_switch(self) -> dict:
    # 已经切回 trainer 的 step 无需等待，直接返回空指标
    if self.current_mode != HybridEngineMode.ROLLOUT:
        return {}

    logger.info(f"Lending hybrid engine to generation until {self._step_threshold} groups are
    sampleable")
    # 阻塞轮询直到 buffer 深度达标；等待期间 buffer 仍会淘汰过期组并补齐新 prompts，
    # 因此这里累积的 eviction_metrics 需要并入本 step 的总指标
    with marked_timer("switch_wait", self.timing_raw, color="yellow"):
        _, eviction_metrics = self.replay_buffer.wait_for_sampleable(
            self.global_steps, "train", self._step_threshold
        )
    self._timed_switch_to_trainer()
    return eviction_metrics


def _switch_threshold(self) -> int:
    """sampleable 组数达到该值时收回引擎，下限为 1 个 mini-batch。"""
    train_batch_size = self.config.data.train_batch_size
    mini_batch_size = train_batch_size // self.parameter_sync_step
    target = round(self._switch_threshold_ratio * train_batch_size)
    # floor 到 1 个 mini-batch 避免阈值过小导致频繁往返，cap 到整个 train_batch_size
    return min(max(target, mini_batch_size), train_batch_size)


def _step_had_idle(self) -> bool:
    """判断上一轮是否在等待 sampleable 数据（即 trainer 处于饥饿状态）。"""
    poll_interval = getattr(self.replay_buffer, "poll_interval", 2.0)
    # 实际等待时间超过轮询间隔即视为饥饿，而不是看是否阻塞过
    return self._step_sample_wait_seconds > poll_interval


def _adapt_switch_threshold(self, had_idle: bool) -> None:
    # 饥饿（生成跟不上）时上调阈值：让 hybrid 引擎多在 rollout 待一会，缓解训练饥饿
    if had_idle:
        self._calm_steps = 0
        self._idle_steps = min(self._idle_steps + 1, config.switch_threshold_release_steps)
        if self._idle_steps < config.switch_threshold_release_steps:
            return
```

```
            self._switch_threshold_ratio = min(1.0, self._switch_threshold_ratio + config.switch_
            threshold_step_up)
            return

        # 平静（生成充足）时下调阈值：尽早收回 GPU 训练；连续步数滞回避免来回抖动
        self._idle_steps = 0
        self._calm_steps = min(self._calm_steps + 1, config.switch_threshold_release_steps)
        if self._calm_steps < config.switch_threshold_release_steps:
            return
        # 阈值下限对应 1 个 mini-batch，避免低于单 mini-batch 失去切换意义
        min_ratio = 1.0 / self.parameter_sync_step
        self._switch_threshold_ratio = max(min_ratio, self._switch_threshold_ratio - config.switch_
        threshold_step_down)
```

verl/trainer/ppo/v1/replay_buffer.py

新增 get_sampleable_count 与 wait_for_sampleable 两个 buffer 深度只读接口，sample 重构为对它们的封装，是本 PR 借卡决策的数据基础。

```
    def get_sampleable_count(self, global_steps: int, partition_id: str) -> int:
        """返回当前可采样的 terminal 组数量，不消费任何数据。"""
        # 先同步 TransferQueue 元数据，再与淘汰逻辑共用同一份快照，
        # 保证 trainer 预判存量时看到的是与 sample 一致的视图
        self._sync_metadata_from_transfer_queue()
        eviction_reasons = self._terminal_eviction_reasons(global_steps, partition_id)
        return len(self._sampleable_terminal_keys(partition_id, eviction_reasons))


    def wait_for_sampleable(self, global_steps: int, partition_id: str, target_count: int) ->
    tuple[set[str], dict]:
        """轮询直到 ``target_count`` 个组可采样，等待期间持续淘汰并补齐。

        返回可采样 uid 集合与等待期间累计的淘汰指标。与 sample 不同，它不消费组，
        因此 trainer 收回引擎后，后续 mini-batch 仍能采到这批数据。
        """
        last_debug_time = time.time()
        eviction_metrics: dict = {}

        while True:
            # 淘汰与选择共享同一份快照，保证新到达的过期组等到下一轮才被淘汰
            self._sync_metadata_from_transfer_queue()

            eviction_reasons = self._terminal_eviction_reasons(global_steps, partition_id)
            evicted_uids, stale_count, _dapo_count, metrics = self._evict_terminal_groups(
                global_steps, partition_id, eviction_reasons
            )
            if evicted_uids:
                _accumulate_eviction_metrics(eviction_metrics, metrics, stale_count)
                if self.refill_fn is not None:
                    self.refill_fn(len(evicted_uids))
```

```
            continue

        sampleable_keys = self._sampleable_terminal_keys(partition_id, eviction_reasons)
        if self._has_enough_samples(global_steps, partition_id, target_count, sampleable_keys):
            return sampleable_keys, eviction_metrics

        last_debug_time = self._wait_for_next_poll(partition_id, last_debug_time)


@SkipManager.annotate_tq(role="rollout_tq", phase="sample")
def sample(self, global_steps: int, partition_id: str, batch_size: int) -> tuple[KVBatchMeta, dict]:
    """采样一批数据，同时淘汰过期、DAPO 过滤或失败的组，并补齐等量新 prompts。"""
    # sample 只是把 batch_size 作为目标深度传给 wait_for_sampleable，
    # 选组逻辑保持不变，行为与改动前完全等价
    sampleable_keys, eviction_metrics = self.wait_for_sampleable(global_steps, partition_id,
    batch_size)
    selected_prompt_uids, partition_snapshot, prompt_global_steps_snapshot = self._select_
    prompt_uids(
        partition_id, sampleable_keys, batch_size
    )

    if partition_id != "val" and self.max_off_policy_strategy == "drop":
        selected_spans = [
            global_steps - prompt_global_steps_snapshot.get(uid, global_steps) + 1 for uid in
            selected_prompt_uids
        ]
        assert all(span <= self.max_off_policy_threshold for span in selected_spans), (
            f"drop strategy selected stale prompts: spans={selected_spans}, "
            f"threshold={self.max_off_policy_threshold}"
        )

    return self._materialize_batch(partition_id, selected_prompt_uids, partition_snapshot),
    eviction_metrics
```

verl/workers/rollout/vllm_rollout/vllm_async_server.py

release_kv_cache/resume_kv_cache 从空实现变为 sleep(level)+wake_up(tags) 组合模拟，是避免权重同步 OOM 的关键 workaround，并抽出 _resolve_sleep_level 统一 level 选择。

```
async def release_kv_cache(self):
    """在权重同步期间释放 kv_cache 池，腾出 GPU 显存。"""
    # TODO: 待 vLLM 落地 release_kv_cache() API 后改为真正的 KV-only 释放（vllm#44890 /
    #46438）
    if self.node_rank != 0 or not self.config.free_cache_engine:
        return
    # COLOCATED 模式下引擎与训练共享显存，权重同步走既有路径，无需额外释放
    if self.rollout_mode == RolloutMode.COLOCATED:
        return
    # 当前 vLLM 没有只释放 KV cache 的 API，退而求其次：先 sleep 丢弃全部 GPU 内存，
    # 再按 tags=["weights"] 恢复模型权重，模拟“只释放 KV cache”。代价是多一次权重重映射，
```

```
        # 但能避免更新 standalone 权重时的 OOM
        await self.engine.sleep(level=self._resolve_sleep_level())
        await self.engine.wake_up(tags=["weights"])


    async def resume_kv_cache(self):
        """权重同步结束后恢复 kv_cache 显存，与 release_kv_cache() 配对使用。"""
        if self.node_rank != 0 or not self.config.free_cache_engine:
            return
        if self.rollout_mode == RolloutMode.COLOCATED:
            return
        # 只 remap 回 kv_cache 标签，权重保持 resident；之后清空 prefix cache，
        # 因为旧缓存是用同步前的旧权重算出来的，继续使用会与最新权重不一致
        await self.engine.wake_up(tags=["kv_cache"])
        await self.engine.reset_prefix_cache(reset_connector=True)


    def _resolve_sleep_level(self) -> int:
        """选择可被后续权重同步完整恢复的最深 sleep level。"""
        mtp_config = getattr(self.config, "mtp", None)
        mtp_rollout_enabled = (
            mtp_config is not None
            and getattr(mtp_config, "enable", False)
            and getattr(mtp_config, "enable_rollout", False)
        )
        # MTP drafter-only 权重由 vLLM 初始化，level 2 丢弃后 actor 权重同步不会恢复它们；
        # LoRA 只更新 adapter 权重；vllm_ascend 尚不支持 sleep_level，且 EP 训练可能影响精度
        if mtp_rollout_enabled or self.lora_as_adapter or is_torch_npu_available(check_device=False):
            return 1
        return 2
```

评论区精华

review 共 11 条评论，核心交锋集中在 4 点：

- 参数组织方式：wuxibin89 建议把散落的 switch 参数收进一个配置类，用 _target_ 指向 dataclass。已采纳，最终配置从扁平键演化为 hybrid_rollout._target_: verl.trainer.config.HybridRolloutSwitchConfig，这也是 PR body 中 enable_switch 顶层键与最终实现不一致的原因。
- 钩子设计：wuxibin89 指出新增的 on_step_prompts_submitted 钩子「not general」，建议 PPOTrainerSeparateAsync 直接覆写 _add_batch_to_generate。最终方案保留了 prepare_step 可覆写钩子，但把默认行为收敛到 trainer_base（默认调用 _add_batch_to_generate 并返回空 dict），保证非 separate_async 训练路径行为完全不变。
- 校验下沉：wuxibin89 建议把阈值范围、成本窗口校验移入 HybridRolloutSwitchConfig.__post_init__，已采纳。

- 统计指标：zpltys 要求为 _switch_threshold_ratio 增加统计指标以观察自适应收敛过程。head 版本已有 separate_async/decision/per_sample_time_seconds、effective_switch_cost_seconds 等决策指标，但阈值比率本身的指标是否落地未在可见代码中确认。
  - switch 参数分组到配置类 (design)：已采纳。最终配置形如 trainer.v1.separate_async.hybrid_rollout._target_: verl.trainer.config.HybridRolloutSwitchConfig，PR body 中顶层 enable_switch 键因此与最终实现不一致。
  - prepare_step 钩子是否通用 (design)：最终在 trainer_base 保留轻量 prepare_step（默认调用 _add_batch_to_generate 并返回空 dict），由 separate_async 覆写并追加 _wait_for_sampleable_and_switch，默认行为不变。
  - 阈值统计指标 (design)：head 版本已输出 separate_async/decision/* 系列指标（per_sample_time_seconds、effective_switch_cost_seconds 等），但阈值比率本身的统计是否落地未在可见代码中确认。
  - sanity check 下沉到配置类 (design)：已采纳，config.py 中 __post_init__ 对越界参数抛 ValueError。
  - 文档位置调整 (documentation)：已解决，文档纳入正式文档体系并加入目录索引。

风险与影响

- 风险：主要风险集中在 5 点：
  - vLLM 内存语义依赖：release_kv_cache 用 sleep(level=2) + wake_up(tags=["weights"]) 模拟，实际会丢弃全部 GPU 内存再重映射权重，比真正的 KV-only 释放成本高，且强依赖 vLLM 现行 sleep 级别与 tag 语义；vLLM 升级可能改变行为。代码已用 TODO 指向 vllm#44890/#46438。
  - 每次 global step 的完整往返开销：开启后每步都要经历 resume_generation_replicas（权重恢复）→ abort_replicas + sleep_replicas 的完整往返，Megatron 大模型下权重恢复成本直接侵蚀收益；_effective_switch_cost 仅用滑动窗口均值估算，_scaling_factor 是静态粗略假设（代码中留有 TODO）。
  - 开启时的硬约束：enable_switch=True 会拒绝 PD 分离 rollout，并要求 replay buffer 实现两个新方法，自定义 sampler 不满足会直接 TypeError 阻止启动。
  - 决策粘性与吞吐波动：PR 自述 limitations 承认所有决策都从前一步测量外推，生成吞吐剧烈振荡时收益退化；实测 staleness 均值上升 6.5%，长训下与 max_off_policy_strategy 的交互需观察。
  - 验证覆盖有限：仅 2 hybrid + 1 standalone 比例的 Megatron 单次 e2e，FSDP、SGLang、NPU 等后端与资源比例未覆盖。
- 影响：影响范围中等：
  - 用户侧：separate_async 用户默认零影响（enable_switch=false 行为不变）；开启后需按新配置组 hybrid_rollout.* 配置，并接受 staleness 上升与 PD 分离不兼容的限制。
  - 系统侧：replay_buffer.sample 是等价重构（提取 wait_for_sampleable），但新增两个公共方法形成接口契约；vLLM 服务器的 release/resume_kv_cache 从空实现变为实际

显存操作，COLOCATED 模式显式跳过以避免干扰既有路径。

- 团队侧：为上游 vLLM 提供了 release_kv_cache() API 的第一验证场景和交接点；新增的 docs/advance/v1_async_trainer.md 系统梳理了 V1 异步训练模式、partial rollout、off-policy 控制与 step switch 的完整关系，降低后续维护者的理解成本。
- 风险标记：vLLM sleep 语义依赖，每次 step 权重往返开销，开启限制 PD 与自定义 buffer，决策粘性依赖吞吐平稳，单一资源比例验证

关联脉络

- PR #6556 fully_async 动态资源调度：PR body 明确说明本 PR 的阈值自适应思想直接改编自 #6556 引入的动态资源调度，属于同一调度理念在不同训练模式上的推广。
- PR #7422 [rollout] fix: preserve dummy load_format in disaggregated rollout: 同样修改 vllm_async_server.py 等 rollout 服务器文件，修复分离式 rollout 权重广播问题，与本 PR 的权重同步与服务器生命周期改动相交。
- PR #7434 [vllm] fix: vllm always need to resume weights before weight sync: 修复 vLLM 权重同步前必须恢复权重映射的回归，与本 PR 中 sleep/wake_up 之后的权重恢复逻辑同属 update_weights 链路。
- PR #7408 [perf] feat: add profiler post hook, fix behaviors and allow comprehensive torch profiler: 扩展 trainer 的 step 生命周期钩子（on_step_begin/prepare_step 系列），与本 PR 在 trainer_base 新增 prepare_step 钩子的演进方向一致。