# PR #2030 完整报告

radixark/miles

[async] async data buffer: unified filters and better observability

合并时间：2026-08-08 11:02

原文链接：http://prhub.com.cn/radixark/miles/pull/2030

## 执行摘要

- 一句话：fully-async 数据缓冲重构：容量限流、统一过滤与可观测性
- 推荐动作：### 建议

值得精读。核心设计是「把 group 级过滤与数据流控制集中到 DataBuffer 抽象」，并配套了完整的指标与参数化策略，对理解 fully-async rollout 的数据管线很有帮助。重点关注：

- DefaultDataBuffer.put/get 的裁决时机（put 裁决 abort/ 动态过滤，get 裁决陈旧度）为何合理；
- 容量因子默认值 2 的实测依据（GLM-5.2 16 节点稳态队列 0-5 组）；
- review 中关于 recycle 无限重试的讨论与最终 retry/drop 策略的取舍。

## 功能与动机

PR body 指出：在 fully-async 模式下，当 rollout 生产速度超过训练消费速度时，完成的group会堆积在实际上无界的队列里（asyncio.Queue(maxsize=1000)=125个训练步的积压），而 FIFO 消费意味着训练永远吃最旧、最陈旧的数据，既无边界也无可见性。此外，原先 avg/max_staleness 指标只有在设置了 --max-weight-staleness 时才上报，用户无法在开启前评估是否需要该功能。review 中 Shi-Dong 还提出了 recycle 可能导致任务被无限次送回 rollout 机器的担忧，推动了 --async-unused-samples-handler 策略化。

## 实现拆解

### 实现拆解

1. 新增 DataBuffer 抽象模块（miles/rollout/fully_async_data_buffer.py） - 定义 Group、iter_samples、first_sample、group_oldest_weight_version 等纯函数，以及 DataBufferConstructorInput、DataBufferInput 两个数据容器。 - 定义 3 方法抽象类 DataBuffer（put / get / get_metrics），把 group 级决策（保留、丢弃、回收）全部下沉到缓冲实现；DefaultDataBuffer 是内置 FIFO 实现，--custom-async-data-buffer-path 可整体替换。 - put 阶段裁决 abort 与动态过滤（生成后结论即定）；get 阶段按传入的 current_version 裁决陈旧度（依赖消费时刻）。 - 容量控制：--async-data-buffer-capacity-factor（默认 2.0）× rollout_batch_size，满时 put 阻塞生产者，保留 backpressure。
2. 重构 FullyAsyncRolloutFn（miles/rollout/fully_async_rollout.py） - 输出从 asyncio.Queue 换为 DataBuffer，__call__ 中懒加载实例化 buffer 并启动 worker。 -

_generate_group 返回 DataBufferInput(prompt_group, group)，_next_group(current_version) 将引擎权重版本传给 buffer.get()。 - _drain 中删除内联的 abort/staleness/dynamic 过滤与回收逻辑，统一由 buffer 负责；未使用样本处理策略 --async-unused-samples-handler（retry/drop，默认 drop）在构造时绑定为 _handle_unused。

3. 参数与校验（miles/utils/arguments.py） - 新增 --async-data-buffer-capacity-factor（float，默认 2.0）、--async-unused-samples-handler（retry/drop，默认 drop）、--custom-async-data-buffer-path。 - _resolve_rollout_functions 新增断言：fully-async 禁止 --pause-generation-mode abort，因为生成永远在途，每次权重更新 abort 都会杀死全部生成并强制重新生成。
4. 测试配套 - tests/fast/rollout/test_fully_async_rollout.py：新增 make_buffer 工具与 DataBuffer 单测（满时阻塞、get 忽略未知 context 键、消费时陈旧过滤、陈旧度指标、drop 默认策略），并把既有回收测试改为显式 async_unused_samples_handler="retry"。 - tests/fast/utils/test_arguments.py：新增 test_fully_async_rejects_abort_pause_mode，验证 abort 模式被拒绝、retract 模式通过。

关键文件：

- miles/rollout/fully_async_data_buffer.py（模块 数据缓冲；类别 source；类型 core-logic；符号 iter_samples, first_sample, group_oldest_weight_version, DataBufferConstructorInput）：新增核心模块，定义 DataBuffer 抽象与 DefaultDataBuffer 实现，承载全部 group 级过滤、容量控制与指标收集，是本次变更的主体。
- miles/rollout/fully_async_rollout.py（模块 异步回放；类别 source；类型 core-logic；符号 _generate_group, _next_group, _drain, _handle_unused）：FullyAsyncRolloutFn 从 asyncio.Queue 迁移到 DataBuffer，删除内联过滤回收逻辑，消费侧传递 current_version，是核心管线的适配改造。
- miles/utils/arguments.py（模块 参数解析；类别 source；类型 configuration；符号 _resolve_rollout_functions, add_rollout_arguments）：新增三个 async data buffer 参数，并禁止 fully-async 与 --pause-generation-mode abort 组合，是配置面与控制面配套。
- tests/fast/rollout/test_fully_async_rollout.py（模块 测试；类别 test；类型 test-coverage；符号 make_buffer, put_group, test_buffer_blocks_producer_when_full, test_buffer_get_ignores_unknown_context_keys）：覆盖 DataBuffer 容量阻塞、消费时陈旧过滤、drop 默认策略、指标上报与回收语义变更，是本次行为变更的主要回归保障。
- tests/fast/utils/test_arguments.py（模块 测试；类别 test；类型 test-coverage；符号 test_fully_async_rejects_abort_pause_mode）：新增 fully-async 拒绝 abort 暂停模式的参数校验测试，保护新增断言。

关键符号：iter_samples, first_sample, group_oldest_weight_version, DataBuffer.put, DataBuffer.get, DataBuffer.get_metrics, DefaultDataBuffer.put, DefaultDataBuffer.get, DefaultDataBuffer.get_metrics, DefaultDataBuffer._staleness, FullyAsyncRolloutFn.call, FullyAsyncRolloutFn._generate_group, FullyAsyncRolloutFn._next_group, FullyAsyncRolloutFn._drain, _resolve_rollout_functions

## 关键源码片段

### miles/rollout/fully_async_data_buffer.py

新增核心模块，定义 DataBuffer 抽象与 DefaultDataBuffer 实现，承载全部 group 级过滤、容量控制与指标收集，是本次变更的主体。

```
# DataBuffer：fully-async 生产与消费之间的完成组缓冲契约
# put 接收完成组；get 返回一个可训练组并携带消费时上下文；
# get_metrics 按训练步收集窗口指标并重置计数器。
class DataBuffer(ABC):
    @abstractmethod
    async def put(self, input: DataBufferInput) -> None:
        """接受一个完成组；可存储、拒绝或驱逐腾位。"""

    @abstractmethod
    async def get(self, **context) -> DataBufferInput:
        """返回一个组用于训练，无可用组时等待；context 携带消费时信息。"""

    @abstractmethod
    def get_metrics(self) -> dict[str, float]:
        """返回上次调用以来的全限定指标（窗口计数器在此重置）。"""


class DefaultDataBuffer(DataBuffer):
    """FIFO 缓冲：生成侧已定的结论在 put 时裁决，
    依赖消费时刻的陈旧度在 get 时裁决。"""

    def __init__(self, input: DataBufferConstructorInput):
        args = input.args
        self._args = args
        # 容量 = factor * rollout_batch_size，默认 2 个训练 batch
        self._capacity = int(args.async_data_buffer_capacity_factor * args.rollout_batch_size)
        self._unused_handler_fn = input.unused_handler_fn # retry 回收 / drop 丢弃
        self._dynamic_filter = load_function(args.dynamic_sampling_filter_path)
        self._buffer: list[DataBufferInput] = []
        self._cond = asyncio.Condition()
        self._current_version: int | None = None

    async def put(self, input: DataBufferInput) -> None:
        # put 阶段裁决：abort 与动态过滤，结论在生成完成时即固定
        if any(s.status == Sample.Status.ABORTED for s in iter_samples(input.group)):
            self._metric_aborted_groups += 1
            self._unused_handler_fn(input.prompt_group)
            return
        filter_output = call_dynamic_filter(self._dynamic_filter, self._args, input.group)
        if not filter_output.keep:
            # 动态过滤丢弃不走回收：没有可用梯度信号
            self._metric_gatherer.on_dynamic_filter_drop(reason=filter_output.reason)
            return
```

```
    # 容量满时阻塞生产者，保留背压，避免无界积压
    async with self._cond:
        while len(self._buffer) >= self._capacity:
            await self._cond.wait()
        self._buffer.append(input)
        self._cond.notify_all()

async def get(self, current_version: int | None = None, **_) -> DataBufferInput:
    if current_version is not None:
        self._current_version = current_version
    async with self._cond:
        while True:
            while not self._buffer:
                await self._cond.wait()
            entry = self._buffer.pop(0)
            self._cond.notify_all() # 唤醒被容量阻塞的生产者
            # get 阶段裁决：陈旧度依赖消费时刻的引擎权重版本
            staleness = self._staleness(entry.group, current_version)
            if staleness is None:
                return entry
            self._metric_consumed_staleness.append(staleness)
            if self._args.max_weight_staleness is None or staleness <= self._args.max_weight_
            staleness:
                return entry
            self._metric_stale_groups += 1
            self._unused_handler_fn(entry.prompt_group) # 超阈值组回收或丢弃
```

## miles/rollout/fully_async_rollout.py

FullyAsyncRolloutFn 从 asyncio.Queue 迁移到 DataBuffer，删除内联过滤回收逻辑，消费侧传递 current_version，是核心管线的适配改造。

```
async def __call__(self, input: RolloutFnInput) -> RolloutFnOutput:
    if input.evaluation:
        return await self._call_eval(input)
    if self._worker is None:
        # 用 DataBuffer 替换原先的 asyncio.Queue：默认实现可被自定义类替换
        buffer_cls = load_function(self.args.custom_async_data_buffer_path) or
        DefaultDataBuffer
        self._output = buffer_cls(
            DataBufferConstructorInput(args=self.args, unused_handler_fn=self._handle_
            unused)
        )
        self._worker = asyncio.create_task(self._worker_loop())
        logger.info("Started fully-async rollout worker")
    return await self._drain(input)

async def _next_group(self, current_version: int | None) -> DataBufferInput:
    # 把当前引擎权重版本传给 buffer，陈旧度过滤在消费时进行
    queue_get = asyncio.create_task(self._output.get(current_version=current_version))
```

```
        try:
            while True:
                done, _ = await asyncio.wait(
                    {queue_get, self._worker},
                    return_when=asyncio.FIRST_COMPLETED,
                    timeout=NO_PROGRESS_WARN_SECS,
                )
                # 先查 worker 再查队列：worker 异常要先于积压数据暴露
                if self._worker in done:
                    self._worker.result()
                    raise RuntimeError("fully-async rollout worker exited without an exception")
                if queue_get in done:
                    return queue_get.result()
                logger.warning(f"No completed rollout groups for {NO_PROGRESS_WARN_SECS}s")
        finally:
            if not queue_get.done():
                queue_get.cancel()
```

# 评论区精华

## 评论区精华

> guapisolo：A dumb q, will anyone actually use LIFO?
>
> Shi-Dong：This is a good point... LIFO does seem unnecessary. I checked AReaL and they hardcoded FIFO.

- 结论：最终删除了 --async-data-buffer-order lifo 选项，DefaultDataBuffer 固定为 FIFO。

> Shi-Dong：Suppose that a task is genuinely hard and takes long to finish... Does it mean that the task will be endlessly sent to the rollout machine?
>
> guapisolo：I have similar concern... Add a --async-stale-samples-handler arg. It can include three modes "retry", "drop" and "mask"... And it can also be a customizable function.

- 结论：实现为 --async-unused-samples-handler（retry/drop，默认 drop），mask 模式留作 TODO；自定义函数通过 --custom-async-data-buffer-path 实现。

> guapisolo：I suggest we move dynamic filter function before the staleness handler. cuz if a prompt cannot produce useful signal. We should drop it first.

- 结论：接受。动态过滤与 abort 在 put 时先裁决，陈旧度在 get 时裁决，顺序天然满足。

> guapisolo：I think it's a bug comment here... in arguments.py we should ban --pause-generation-mode to be not abort

- 结论：在参数校验中加入断言，fully-async 禁止 abort 暂停模式。

> guapisolo：Why the blocking logic removed here? ... I suggest the behavior change: When the data buffer is full, block the data_source -- prompt --> generation_pool path to stop too many rollouts.

- 结论：最终保留阻塞语义（提交 "block instead of evicting when the data buffer is full"），容量默认收窄到 2 个训练 batch。

> guapisolo：do you think it's better to set this param as --async-data-buffer-max-groups...
>
> yueming-yuan：If the name is a bit misleading, how about changing it to --async-data-buffer-capacity-factor to emphasize that it's a factor?

- 结论：参数定名为 --async-data-buffer-capacity-factor，类型为 float，容量取 floor(factor * rollout_batch_size)。
  - LIFO 消费顺序是否必要 (design)：删除 --async-data-buffer-order 选项，DefaultDataBuffer 固定 FIFO，保持简单。
  - recycle 是否导致任务无限重试 (correctness)：实现 --async-unused-samples-handler（retry/drop，默认 drop），mask 留作 TODO；自定义策略可通过 --custom-async-data-buffer-path 扩展。
  - 缓冲满时阻塞 vs 驱逐 (design)：保留阻塞语义，容量因子默认 2，满时 put 等待消费；同时保留 get 时的陈旧度过滤。
  - 动态过滤器与陈旧度处理的先后顺序 (design)：接受：abort 与动态过滤在 put 时裁决，陈旧度在 get 时裁决，天然满足先丢弃无效组的顺序。
  - 禁止 --pause-generation-mode abort 与 fully-async 组合 (correctness)：在 _resolve_rollout_functions 中增加 assert，并补充参数测试。
  - 缓冲容量参数的命名与类型 (design)：定名为 --async-data-buffer-capacity-factor，float 类型，容量 = floor(factor * rollout_batch_size)。

## 风险与影响

- 风险：### 风险分析
- 默认行为变化（中风险）：默认容量从 1000 组变为 2×rollout_batch_size，生产者在容量满时会阻塞。若训练消费偶发变慢，rollout 引擎可能因背压停摆，需通过指标 queue_size 观察。
- recycle 无限重试（中风险）：在 retry 模式下，长期硬任务可能反复被回收重新生成，造成算力浪费。PR 通过默认 drop 缓解，但 retry 用户需自行权衡。
- 陈旧度语义依赖 weight-version 传递（中风险）：get(current_version=...) 依赖 trainer 侧正确传入引擎权重版本；若版本传递链断裂（如 LoRA 多传输路径），avg_staleness 等指标会失真，PR 中合入了 #2244 的权重版本直传逻辑作为配套。
- 自定义缓冲扩展点（低风险）：--custom-async-data-buffer-path 允许替换整个缓冲实现，但自定义类需自行处理容量、陈旧度、回收策略，文档字符串已说明，仍有误用风险。
- 测试覆盖（低风险）：fast 测试覆盖了 DataBuffer 单测与参数校验，但未见 e2e 级验证容量阻塞与回收路径的集成测试。
- 影响：### 影响分析
- 用户影响：使用 --fully-async 的启动脚本默认行为变化（缓冲更小、满时阻塞）；新参数提供更细粒度的数据流控制；指标 buffer_avg_staleness、buffer_max_staleness 无条件上

报，训练团队可据此调参。

- 系统影响：生产端背压机制更可控，避免无界队列导致训练数据陈旧；*aborted_groups_recycled* 等指标更名为 *aborted_groups_filtered*，依赖旧指标的监控面板需同步更新。
- 团队影响：rollout 数据流决策从 FullyAsyncRolloutFn 中剥离，抽象出可替换的 DataBuffer 契约，后续可针对不同训练器定制缓冲策略；review 中提出的 mask 模式（类 Kimi k25）留作 TODO，是后续扩展方向。
- 风险标记：核心路径变更，默认行为变化，潜在无限重试，依赖 weight-version 传递

# 关联脉络

- PR #2244 *pass the engine weight version from the trainer instead of polling the router*: 本 PR 提交历史中合入了 '*pass the engine weight version from the trainer instead of polling the router*' 分支，DataBuffer 的陈旧度判断依赖该版本传递链路（*get(current_version=...)*），两个 PR 共同构成 fully-async 数据新鲜度控制的完整方案。
- PR #2241 : Issue 评论中 *guapisolo* 提到 '*also a tiny fix on this #2241*'，说明 #2241 是本 PR 的一个后续小修复，与本 PR 直接相关。