

PR #2865 完整报告

radixark/miles

fix: make get_async_loop initialization thread-safe

合并时间: 2026-09-02 04:31

原文链接: <http://prhub.com.cn/radixark/miles/pull/2865>

执行摘要

- 一句话: 修复 `get_async_loop()` 初始化竞态条件, 防止多事件循环导致 `asyncio` 原语绑定错误。
- 推荐动作: 这是一个值得精读的、小而精的 bugfix。它展示了如何诊断和修复一个在异步与多线程混合环境中出现的典型竞态条件。其价值不仅在于修复本身, 更在于提供的并发测试模式 (利用 `barrier` 和延迟初始化来可控地复现竞争), 这可以作为团队处理类似问题的参考模板。建议所有涉及全局状态初始化或多线程协同开发的工程师关注此 PR。

功能与动机

问题的直接动机源于 PR body 的描述: 在 `RolloutManager` 进程启动时, `train_async.py` 发起的预训练评估分发和 `generate.remote(0)` 几乎同时调用 `asyncio.to_thread(call_rollout_function, ...)`, 这使得两个执行器线程同时竞争 `get_async_loop()`。原始实现存在竞态条件, 当竞争发生时创建两个事件循环。代码库中 `GenerateState.generate_fn_semaphore` 等 `asyncio` 原语会被绑定到其中一个循环上, 当其他协程在“存活”的循环上尝试获取这些原语时, 会引发 `RuntimeError: ... is bound to a different event loop`。此失败在 PR #2823 的新 CI 门控中表现为 `eval/skipped_crashed`。虽然主分支的跳过语义可能掩盖此错误, 但在第二次生成发生在“孤立”循环上时, 会导致训练本身崩溃。因此, 这是一个影响系统稳定性的关键并发缺陷。

实现拆解

1. 问题定位: 核心问题位于 `miles/utils/async_utils.py` 的 `get_async_loop()` 函数。其实现对模块级全局变量 `async_loop` 的检查与赋值操作没有原子性保证, 导致并发调用时可能创建多个 `AsyncLoopThread` 实例。
2. 源码修复: 在 `miles/utils/async_utils.py` 中, 引入模块级线程锁 `_async_loop_init_lock`。修改 `get_async_loop()` 函数, 采用双重检查锁定模式: 首先在无锁条件下快速检查 `async_loop` 是否为 `None`, 若非 `None` 直接返回; 若为 `None`, 则获取锁, 并在锁内再次检查 `async_loop`, 仅当第二次检查仍为 `None` 时, 才实例化 `AsyncLoopThread` 并赋值给 `async_loop`。
3. 测试验证: 在 `tests/fast/utils/test_async_utils.py` 中新增 `TestGetAsyncLoopSingleton` 测试类及 `test_concurrent_first_callers_share_one_loop` 测试方法。该测试通过 `monkeypatch` 延长 `AsyncLoopThread.__init__` 的执行时间以放大竞争窗口, 并利用 `threading.Barrier` 确保两个线程同时发起调用, 从而可靠地验证修复后只会构造一个循环。

实例，且两个线程返回相同的对象。

关键文件：

- miles/utils/async_utils.py（模块 异步工具；类别 source；类型 core-logic）：核心源码文件，包含存在竞态条件的 `get_async_loop()` 函数及修复实现。
- tests/fast/utils/test_async_utils.py（模块 异步工具测试；类别 test；类型 test-coverage；符号 `TestGetAsyncLoopSingleton`, `test_concurrent_first_callers_share_one_loop`）：测试文件，新增针对 `get_async_loop` 单例的并发测试，验证修复的有效性。

关键符号：`get_async_loop`

关键源码片段

miles/utils/async_utils.py

核心源码文件，包含存在竞态条件的 `get_async_loop()` 函数及修复实现。

```
# Miles/utils/async_utils.py (修复后版本节选)
```

```
# 创建一个后台事件循环线程
```

```
class AsyncLoopThread:
```

```
    def __init__(self):
```

```
        self.loop = asyncio.new_event_loop()
```

```
        self._thread = threading.Thread(target=self._start_loop, daemon=True)
```

```
        self._thread.start()
```

```
    def _start_loop(self):
```

```
        asyncio.set_event_loop(self.loop)
```

```
        self.loop.run_forever()
```

```
    def run(self, coro):
```

```
        # 在后台事件循环上调度一个协程并阻塞直到完成
```

```
        return asyncio.run_coroutine_threadsafe(coro, self.loop).result()
```

```
# 创建一个全局实例
```

```
async_loop = None
```

```
# 新增线程锁，用于保护单例初始化
```

```
_async_loop_init_lock = threading.Lock()
```

```
def get_async_loop():
```

```
    global async_loop
```

```
    # 双重检查锁定模式：先无锁检查，再加锁检查
```

```
    if async_loop is None:
```

```
        # 获取锁，确保只有一个线程能进入初始化代码块
```

```
        with _async_loop_init_lock:
```

```
            # 在锁内再次检查，防止其他线程已完成初始化
```

```
            if async_loop is None:
```

```
                async_loop = AsyncLoopThread()
```

```
return async_loop
```

此代码片段展示了修复后的核心逻辑。通过引入 `_async_loop_init_lock` 并在 `get_async_loop()` 中应用双重检查，确保了无论多少个线程同时首次调用该函数，都只会创建一个 `AsyncLoopThread` 实例。

tests/fast/utils/test_async_utils.py

测试文件，新增针对 `get_async_loop` 单例的并发测试，验证修复的有效性。

```
# tests/fast/utils/test_async_utils.py ( 新增测试节选 )
```

```
import threading
import time
import miles.utils.async_utils as async_utils
```

```
class TestGetAsyncLoopSingleton:
    def test_concurrent_first_callers_share_one_loop(self, monkeypatch):
        """两个线程竞争首次调用，必须只构造一个循环；
        一个被绑定到失败循环的原语后续会抛出 'bound to a different event loop'。"""
        constructed = []
        real_init = async_utils.AsyncLoopThread.__init__

        # 定义一个延长的初始化函数，放大竞争窗口
        def slow_init(self):
            constructed.append(self)
            time.sleep(0.2) # 持有检查 - 然后 - 执行窗口
            real_init(self)

        # 使用 monkeypatch 替换 __init__ 和重置全局状态
        monkeypatch.setattr(async_utils.AsyncLoopThread, "__init__", slow_init)
        monkeypatch.setattr(async_utils, "async_loop", None)

        # 使用 Barrier 确保两个线程同时发起调用
        barrier = threading.Barrier(2)
        results = []

        def call():
            barrier.wait() # 等待所有线程就绪
            results.append(async_utils.get_async_loop())

        threads = [threading.Thread(target=call) for _ in range(2)]
        for t in threads:
            t.start()
        for t in threads:
            t.join()

        # 断言：只构造了一个实例，并且两个线程得到相同对象
        assert len(constructed) == 1
```

```
assert results[0] is results[1]
```

此测试是验证修复的关键。它通过 `monkeypatch` 延长初始化时间，并利用 `threading.Barrier` 制造并发竞争，断言最终只创建了一个循环实例，且两个线程返回的结果是同一个对象。

评论区精华

尽管 PR 评论数量为零（无技术讨论），但其本身的设计决策清晰。修复方案直接针对问题描述中明确的竞态条件，采用了经典的、低开销的双重检查锁定模式，而非引入更复杂的同步原语。这是一个权衡了正确性、性能和代码简洁性的务实选择。测试设计非常精准，直接复现了并发竞争场景，证明了修复的有效性。

- `get_async_loop()` 初始化竞态条件的识别与修复方案 (correctness): 通过引入线程锁和双重检查，确保了单例初始化的线程安全，解决了偶发性崩溃问题。

风险与影响

- 风险：技术风险较低。 1. 回归风险：变更范围极小，仅增加了初始化的同步控制，不影响 `get_async_loop` 返回后的功能逻辑。潜在的性能影响仅限于首次调用的锁竞争，但此操作发生频率极低，且锁持有时间（实例化一个线程）很短，可忽略不计。 2. 正确性风险：修复本身是标准模式，但需确保测试中 `monkeypatch` 的用法不会意外影响其他测试或模块状态。 3. 兼容性风险：无，不涉及公共 API 变更。主要风险在于如果未来有其他路径绕过 `get_async_loop` 直接操作 `async_loop` 全局变量，则可能引入新的竞态，但目前代码结构良好。
- 影响：影响范围有限但关键。此修复直接解决了可能导致训练或评估流程在启动阶段偶发性崩溃的竞态条件，提升了系统的健壮性。受影响的核心路径包括：RolloutManager 进程的初始化、所有通过 `run()` 或 `get_async_loop()` 使用后台事件循环的异步操作（如 rollouts 调度、评估任务）。对于依赖 `asyncio` 事件循环一致性的组件（如 PR body 中提到的 `GenerateState.generate_fn_semaphore`），此修复是必要的保障。从更广泛的视角看，它避免了因偶发性竞态而引入难以追踪的异步错误，有助于 CI 的稳定性（如 #2823 所揭示）。
- 风险标记：并发竞态修复，核心路径影响

关联脉络

- PR #2823 (推测) 与新 CI 门控或评估逻辑相关的 PR: 当前 PR 的 body 明确提到此问题是在 #2823 的 CI (stage-c-8-gpu-h200) 中被发现的，表现为 `eval/skipped_crashed`。说明 #2823 可能引入或暴露了该竞态条件。