PR #30606 完整报告

sgl-project/sglang

[Fix] Serialize FanOutCommunicator queueing calls with a FIFO-fair asyncio.Lock

合并时间：2026-07-09 15:04

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/30606

执行摘要

- 一句话：修复 FanOutCommunicator 并发竞态导致 500 错误
- 推荐动作：建议尽快合入此 PR，因为它修复了一个可能导致生产环境间歇性 500 错误的严重竞态。新增的测试是确定性测试，可作为回归保护。合入前建议运行完整的 CI 套件，特别是与通信相关的测试。

功能与动机

并发 /server_info 请求会间歇性返回 500 错误，根本原因是 FanOutCommunicator.queueing_call 中存在一个 wakeup-window 竞态：一个调用者在旧调用者清理完成和等待者唤醒之间到达时，会绕过就绪队列并触发 assert self._result_event is None。此问题在 scheduled CI 的 test_get_server_info_concurrent 测试中被观察到。

实现拆解

1. 替换就绪队列为 asyncio.Lock：在 python/sglang/srt/managers/communicator.py 中，移除了手动管理的 self._ready_queue（Deque[asyncio.Event]），改用 self._queueing_lock = asyncio.Lock()。asyncio.Lock 在 Python 3.x 中默认是 FIFO 公平的，可以确保新调用者不会插队先到的等待者，且异常或取消时会自动释放锁，不会阻塞后续调用者。
2. 重构 queueing_call 方法：将整个发送 - 等待 - 返回流程包裹在 async with self._queueing_lock: 块中。这样，在锁的临界区内，self._result_event 和 self._result_values 不会被并发访问破坏，从而避免了原竞态。
3. 移除不再需要的导入：由于不再使用 collections.deque 和 Deque 类型，从导入中删除了它们，简化了代码。
4. 添加确定性单元测试：新建文件 test/registered/unit/managers/test_fanout_communicator.py，其中 TestQueueingCall.test_concurrent_caller_cannot_bypass_queue 方法通过精心编排的协程调度，精确再现了竞态场景：创建三个任务 A、B、C，其中 C 在 A 完成但 B 还没唤醒的时刻启动。测试验证所有调用者无异常完成，且发送顺序严格为 FIFO。

关键文件：

- python/sglang/srt/managers/communicator.py（模块 通信模块；类别 source；类型 core-logic；符号 FanOutCommunicator, queueing_call）：核心源码文件，修复了 FanOutCommunicator.queueing_call 中的竞态条件，用 asyncio.Lock 替换手动的就绪队

列。

- test/registered/unit/managers/test_fanout_communicator.py（模块 测试；类别 test；类型 test-coverage；符号 TestQueueingCall, test_concurrent_caller_cannot_bypass_queue, scenario）：新增的单元测试文件，包含一个确定性测试，精确再现竞态场景，验证修复的正确性。

关键符号：FanOutCommunicator.queueing_call, FanOutCommunicator.init

关键源码片段

python/sglang/srt/managers/communicator.py

核心源码文件，修复了 FanOutCommunicator.queueing_call 中的竞态条件，用 asyncio.Lock 替换手动的就绪队列。

```
from __future__ import annotations

import asyncio
import copy
from typing import Callable, Generic, List, Optional, TypeVar

T = TypeVar("T")


class FanOutCommunicator(Generic[T]):
    """Fan-out request + collect response primitive over zmq."""

    def __init__(
        self,
        send: Callable[[T], None],
        fan_out: int,
        mode: str = "queueing",
    ):
        self._send = send
        self._fan_out = fan_out
        self._mode = mode
        self._result_event: Optional[asyncio.Event] = None
        self._result_values: Optional[List[T]] = None
        # 使用 FIFO 公平的 asyncio.Lock 替代手动就绪队列
        # 确保新调用者不能插队，且异常 / 取消时自动释放
        self._queueing_lock = asyncio.Lock()
        assert mode in ["queueing", "watching"]

    async def queueing_call(self, obj: T):
        # asyncio.Lock 是 FIFO 公平的：当早到的调用者还在等待时，
        # 新调用者无法获得锁，因此请求严格按照到达顺序串行化。
        # 同时，异常或取消时会自动释放锁，避免阻塞后面的调用者。
        async with self._queueing_lock:
            if obj is not None:
```

```
            self._send(obj)

        self._result_event = asyncio.Event()
        self._result_values = []
        await self._result_event.wait()
        result_values = self._result_values
        self._result_event = self._result_values = None

        return result_values
```

test/registered/unit/managers/test_fanout_communicator.py

新增的单元测试文件，包含一个确定性测试，精确再现竞态场景，验证修复的正确性。

```
import asyncio
import unittest

from sglang.srt.managers.communicator import FanOutCommunicator
from sglang.test.ci.ci_register import register_cpu_ci
from sglang.test.test_utils import CustomTestCase

register_cpu_ci(est_time=5, suite="base-a-test-cpu")


class TestQueueingCall(CustomTestCase):
    def test_concurrent_caller_cannot_bypass_queue(self):
        """A new caller arriving in the wakeup window must not overtake a
        queued caller (this interleaving used to raise AssertionError and
        return 500 on concurrent /server_info requests)."""

        async def scenario():
            sent = []
            comm = FanOutCommunicator(send=sent.append, fan_out=1, mode="queueing")

            # A in-flight, B queued behind it.
            task_a = asyncio.create_task(comm("A"))
            await asyncio.sleep(0)
            task_b = asyncio.create_task(comm("B"))
            await asyncio.sleep(0)

            # Complete A, then create C before A's wakeup runs, so C's first
            # step lands between A's cleanup and B's wakeup.
            comm.handle_recv("resp-A")
            task_c = asyncio.create_task(comm("C"))

            # Drive to completion: feed a response whenever one is in flight.
            tasks = [task_a, task_b, task_c]
            for _ in range(100):
                if all(t.done() for t in tasks):
                    break
```

```
                if comm._result_event is not None and not comm._result_event.is_set():
                    comm.handle_recv(f"resp-{len(sent)}")
                await asyncio.sleep(0)

            # All callers complete without error, in strict FIFO order.
            await asyncio.gather(*tasks)
            self.assertEqual(sent, ["A", "B", "C"])

        asyncio.run(scenario())


if __name__ == "__main__":
    unittest.main()
```

评论区精华

PR 没有 review 评论，但作者在 PR body 中清晰地描述了竞态的触发条件和修复思路。此外，作者通过 /rerun-test 命令重新运行了相关测试，确保修复通过。

- 暂无高价值评论线程

风险与影响

- 风险：
  1. 回归风险：queueing_call 是通信核心路径，替换为 asyncio.Lock 改变了并发控制方式。但 asyncio.Lock 是经过广泛测试的标准原语，且新增的单元测试直接覆盖了竞态场景，回归风险较低。
  2. 性能影响：asyncio.Lock 的 fairness 可能带来轻微的性能开销，但 queueing_call 本身是序列化执行的，锁竞争仅发生在并发请求时，且原有实现已有就绪队列，性能差异可忽略。
  3. watching_call 模式未受影响：本次修改仅限 queueing_call，watching_call 保持不变。- 影响：影响范围：仅影响使用 mode="queueing" 的 FanOutCommunicator 调用（例如 /server_info 等需要序列化执行的请求）。修复后所有此类请求将避免因竞态导致的 500 错误和永久挂起。影响程度：中等。该竞态导致服务不可用（500 或挂起），但触发概率较低（依赖于精确的时间窗口）。修复提高了服务的稳定性和可靠性。
- 风险标记：核心路径变更，并发竞态修复

关联脉络

- PR #30471 [misc] Add CI-only guards for the FutureMap seq_lens relay: 同为对 communicator 相关组件的修复，涉及竞态条件。