

PR #55454 完整报告

vllm-project/vllm

[CI] Recover empty multi-node Docker networks and finish partial cleanup

合并时间: 2026-09-08 03:53

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/55454>

执行摘要

- 一句话: 修复多节点 Docker 测试的网络清理与资源隔离问题, 提升 CI 可靠性。
- 推荐动作: 这是一个重要的 CI 基础设施修复, 直接解决了阻塞测试流水线的 bug。尽管变更集中在两个 shell 脚本, 但其设计决策 (如使用 mktemp 生成唯一网络名、捕获容器 ID、分离清理责任) 体现了良好的防御性编程和所有权模型。建议 CI/CD 维护者和基础设施团队精读此 PR, 了解其解决资源隔离问题的方法, 这些方法可能适用于其他需要可靠清理的脚本。对于普通贡献者, 了解 CI 环境的这一改进即可。

功能与动机

L4 分布式 2- 节点测试车道在测试开始前就失败, 因为 `docker-net` 网络已存在。其 EXIT 陷阱在容器不存在时停止于 `docker stop node0`, 导致网络从未被移除。主构建 #87376 演示了这两种失败: 开放警报有八次失败观察。需要恢复现有的空遗留 `docker-net`, 拒绝被占用的遗留网络, 并为新网络和容器使用调用特定的名称。使用本次调用实际创建的容器 ID 执行命令和清理, 防止另一个调用在启动期间删除新创建的空网络。即使早期停止失败, 也要尝试清理每个拥有的容器和拥有的网络。保留原始命令错误: 当命令本身成功时报告清理失败。移除 AMD 包装器中冗余的全局清理, 该清理之前可能在助手拒绝占用网络后停止 `node0/node1`。助手在成功和失败时都拥有其清理。

实现拆解

1. 网络创建与生命周期管理: 在 `.buildkite/scripts/run-multi-node-test.sh` 中, 引入 `RUN_DIR` 和 `NETWORK_NAME` 变量, 使用 `mktemp -d` 生成唯一临时目录并从中提取唯一名称 (如 `docker-net-XXXX`), 确保每次调用使用唯一网络。`start_network()` 函数首先检查名为 `docker-net` 的遗留网络是否存在: 如果存在且无容器连接, 则移除它; 如果仍有容器连接, 则拒绝复用并返回错误。然后创建以唯一名称命名的新网络。这解决了遗留网络残留和并发调用冲突问题。
2. 容器命名与跟踪: 修改 `start_nodes()` 函数, 将容器命名从固定的 `node$node` 改为使用 `NETWORK_NAME` 前缀的 `$NETWORK_NAME-node$node`, 确保容器名唯一。同时, 使用 `docker run` 命令的输出捕获容器 ID, 并存储在 `STARTED_CONTAINERS` 数组中。所有后续的 `docker exec` 命令 (如启动 Ray、运行测试) 都改用容器 ID 而非名称, 避免名称冲突和歧义。
3. 清理逻辑增强: 重写 `cleanup()` 函数。该函数现在遍历 `STARTED_CONTAINERS` 数组, 尝试停止每个容器, 并记录任何失败。然后检查 `NETWORK_CREATED` 标志, 如果为 1 则尝

试移除唯一网络。清理成功后移除临时目录 RUN_DIR。函数保留原始退出状态：如果测试命令失败，则返回该失败状态；否则返回清理失败状态。这确保了即使部分清理失败，测试结果仍能正确报告，且不会遗漏清理步骤。

4. AMD 包装器简化：在 .buildkite/scripts/hardware_ci/run-amd-test.sh 中，删除了 cleanup_network() 函数及其在测试执行后和解析失败时的调用。这是因为助手脚本 (run-multi-node-test.sh) 现在完全负责其自身创建的资源的清理，移除了冗余且可能引发问题的全局清理逻辑。

关键文件：

- .buildkite/scripts/run-multi-node-test.sh（模块 测试脚本；类别 other；类型 core-logic）：多节点测试辅助脚本，包含本次修复的核心逻辑：网络生命周期管理、容器命名与跟踪、增强的清理逻辑。
- .buildkite/scripts/hardware_ci/run-amd-test.sh（模块 CI 包装器；类别 infra；类型 infrastructure）：AMD CI 硬件测试包装器，移除了冗余的全局清理逻辑，将资源清理责任委托给助手脚本。

关键符号：start_network, start_nodes, cleanup

关键源码片段

.buildkite/scripts/run-multi-node-test.sh

多节点测试辅助脚本，包含本次修复的核心逻辑：网络生命周期管理、容器命名与跟踪、增强的清理逻辑。

```
# 生成唯一网络名称，基于临时目录
RUN_DIR=$(mktemp -d)
NETWORK_NAME="docker-net-${RUN_DIR##*/}"

# 启动网络函数：检查并处理遗留网络，然后创建唯一网络
start_network() {
    # 检查名为 docker-net 的遗留网络是否存在
    if docker network inspect docker-net > /dev/null 2>&1; then
        # 检查遗留网络是否有容器连接
        if [ "$(docker network inspect --format '{{len .Containers}}' docker-net)" != 0 ]; then
            echo "docker-net still has attached containers; refusing to reuse it." >&2
            return 1
        fi
        # 无容器连接，安全移除遗留网络
        docker network rm docker-net
    fi
    # 使用唯一名称创建新网络，固定子网保持兼容性
    docker network create --subnet=192.168.10.0/24 "$NETWORK_NAME"
    NETWORK_CREATED=1
}

# 启动节点函数：使用唯一网络名和容器名，并捕获容器 ID
start_nodes() {
```

```

for node in $(seq 0 $((($NUM_NODES-1))); do
    # ... (GPU 设备映射等配置省略)
    # 使用唯一容器名，并捕获容器 ID
    CONTAINER_ID=$(docker run -d ... --name "$NETWORK_NAME-node$node" --network
"$NETWORK_NAME" ...)
    STARTED_CONTAINERS+=("$CONTAINER_ID")
    # 使用容器 ID 执行后续命令，确保引用唯一
    docker exec -d "$CONTAINER_ID" /bin/bash -c "ray start ..."
done
}

# 清理函数：遍历拥有的容器和网络，记录失败但继续清理
cleanup() {
    local status=$?
    local cleanup_status=0
    # 停止本次调用创建的所有容器
    for container_id in "${STARTED_CONTAINERS[@]}; do
        docker stop "$container_id" || cleanup_status=$?
    done
    # 移除本次调用创建的网络（如果创建了）
    if [ "$NETWORK_CREATED" -eq 1 ]; then
        docker network rm "$NETWORK_NAME" || cleanup_status=$?
    fi
    # 移除临时目录
    rmdir "$RUN_DIR" || cleanup_status=$?
    # 保留原始测试命令的退出状态
    if [ "$status" -ne 0 ]; then
        return "$status"
    fi
    return "$cleanup_status"
}

```

评论区精华

CodeRabbit AI 在审查中提出了三个关键问题：

1. 资源所有权隔离：指出当 `start_network()` 返回错误时，下游的 `cleanup_network()` 可能仍会尝试停止不属于本次调用的容器和网络，造成破坏。结论是助手脚本应完全拥有其资源的生命周期管理，AMD 包装器的冗余清理已被移除。
2. 并发调用安全：指出当前实现仍使用固定子网 `192.168.10.0/24`，在并发调用时可能因子网重叠而导致网络创建失败。作者在 PR body 中已说明此限制，并指出并发调用可能因子网重叠而失败，但不会相互删除资源。这被视为一个已知且可接受的折衷，因为 `SIGKILL` 等强制终止仍可能遗留需要清理的资源。
3. 静态分析建议：AI 工具建议使用调用特定的锁或唯一网络来序列化所有权。作者通过为每次调用使用唯一网络名称（基于 `mktemp` 生成的临时目录）实现了这一目标，从而避免了复杂的锁机制。

- 资源所有权与清理责任 (design): 助手脚本 run-multi-node-test.sh 完全拥有其创建的资源的生命周期管理。通过移除 AMD 包装器中冗余的 cleanup_network() 函数, 将清理责任明确化。助手脚本内部的 cleanup() 函数处理所有清理, 并保留原始退出状态。
- 并发调用的子网冲突 (correctness): 作者在 PR body 中承认这是一个已知限制:
‘Concurrent invocations on one daemon may fail network creation due to subnet overlap; they do not delete each other's resources.’ 固定子网被保留, 因为测试命令使用其头部 IP。这是一个可接受的折衷, 优先解决更紧迫的资源泄漏问题。

风险与影响

- 风险:
 1. 子网冲突风险: 所有调用仍使用固定的子网 192.168.10.0/24。在单个 Docker 守护进程上并发运行多个多节点测试时, 如果两个调用尝试创建网络, 第二个调用将因子网冲突而失败。这是一个已知的限制, 在 PR body 中已说明。
 2. 强制终止风险: 如果调用被 SIGKILL 终止, EXIT 陷阱不会执行, 可能导致临时目录和创建的 Docker 资源残留。这些资源需要手动清理。
 3. 遗留网络处理风险: start_network() 函数在检查到名为 docker-net 的遗留网络且无容器连接时会将其移除。这假设此操作是安全的, 但如果其他脚本也依赖此名称, 则可能产生意外影响。
 - 影响:
 1. CI 可靠性: 直接修复了 L4 分布式 2- 节点测试车道的启动失败和资源泄漏问题, 预计能恢复该测试车道的稳定性, 减少构建失败次数。
 2. 开发者体验: 减少了因 Docker 网络冲突或清理不完整导致的 CI 失败, 降低了开发人员的调试负担。
 3. 维护成本: 清理逻辑更加健壮和自包含, 移除了 AMD 包装器中的冗余代码, 降低了长期维护的复杂性。
 4. 测试隔离: 通过为每次调用使用唯一资源, 提升了测试隔离性, 减少了测试间的意外干扰。
- 风险标记: 子网冲突风险, 强制终止后资源残留, 遗留网络清理假设

关联脉络

- PR #47484 [ROCm][CI] Adding rocm multinode proto: 关联 Issue #47484 为 ROCm 多节点测试添加了协议支持。本 PR 修复了共享的多节点测试辅助脚本中的资源管理问题, 这两个 PR 共同改进了多节点测试基础设施。