PR #40761 完整报告

vllm-project/vllm

[XPU][CI] Fix Docker cleanup races on Intel CI runners

合并时间：2026-04-24 14:08

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/40761

执行摘要

- 一句话：修复 Intel CI Docker 清理竞态，引入文件锁和 TTL 清理
- 推荐动作：该 PR 值得精读，特别是学习如何在 shell 脚本中使用文件锁和 TTL 清理策略来解决并发资源管理问题。建议关注 --no-trunc 参数的使用以及 cleanup_old_ci_images 函数的逻辑，其设计可复用于其他 CI 场景。

功能与动机

PR 描述指出：'This PR fixes Docker cleanup races on Intel CI runners where multiple jobs run concurrently on the same machine. Previously, one job could clean up Docker resources while another job was still pulling or using them, which could lead to failures such as No such container and unexpected image removal.'

实现拆解

1. 引入文件锁防止竞态（.buildkite/scripts/hardware_ci/run-intel-test.sh）：在 cleanup_docker() 函数开头创建文件锁 /tmp/docker-pull.lock，使用 flock 确保同一时刻只有一个任务执行清理操作，并在函数末尾释放锁。
2. 实现基于 TTL 的定向镜像清理（同上）：新增 cleanup_old_ci_images() 函数，按仓库前缀和创建时间过滤镜像，仅删除超过指定过期时间（默认 72 小时）且未被任何容器使用的镜像；当磁盘使用率超过 70% 时，还会强制删除未使用的镜像。这种方式避免误删其他任务正在拉取或使用的镜像。
3. 移除全局 prune 操作（两个文件）：在 remove_docker_container() 中删除了 docker image rm -f 和 docker system prune -f 调用，避免在容器退出时清理整个 Docker 系统资源，改为由上述定向清理函数负责。

关键文件：

- .buildkite/scripts/hardware_ci/run-intel-test.sh（模块 CI 脚本；类别 infra；类型 infrastructure）：核心变更文件，引入了文件锁和基于 TTL 的定向镜像清理逻辑，彻底解决了 Docker 清理竞态问题。
- .buildkite/scripts/hardware_ci/run-xpu-test.sh（模块 CI 脚本；类别 infra；类型 infrastructure）：同步简化了清理逻辑，移除了导致镜像误删的全局 prune 调用。

关键符号：cleanup_docker, cleanup_old_ci_images, remove_docker_container

关键源码片段

.buildkite/scripts/hardware_ci/run-intel-test.sh

核心变更文件，引入了文件锁和基于 TTL 的定向镜像清理逻辑，彻底解决了 Docker 清理竞态问题。

```
# 来源：run-intel-test.sh
cleanup_docker() {
    # 使用文件锁与镜像拉取操作共享锁，避免同一节点上的清理 / 拉取竞态
    local docker_lock="/tmp/docker-pull.lock"
    exec 9>"$docker_lock"
    flock 9

    docker_root=$(docker info -f '{{.DockerRootDir}}')
    if [ -z "$docker_root" ]; then
        echo "Failed to determine Docker root directory." >&2
        flock -u 9
        return 1
    fi

    disk_usage=$(df "$docker_root" | tail -1 | awk '{print $5}' | sed 's/%//')
    local threshold=70
    if [ "$disk_usage" -gt "$threshold" ]; then
        echo "Disk usage is above $threshold%. Running aggressive CI image cleanup..."
        # 磁盘压力时，强制清理未使用的 CI 镜像
        cleanup_old_ci_images "${REGISTRY}/${REPO}" "${image_name}" "${DOCKER_IMAGE_
        CLEANUP_HOURS:-72}" 1
    else
        echo "Disk usage is below $threshold%. Checking old CI images anyway."
        # 即使磁盘未满，也清理过期镜像
        cleanup_old_ci_images "${REGISTRY}/${REPO}" "${image_name}" "${DOCKER_IMAGE_
        CLEANUP_HOURS:-72}" 0
    fi

    flock -u 9
}

cleanup_old_ci_images() {
    # 参数：仓库前缀、当前镜像引用、TTL 小时、是否强制清理
    local repo_prefix="$1"
    local current_image_ref="$2"
    local ttl_hours="$3"
    local aggressive_cleanup="$4"

    # 计算截止时间戳
    local now_epoch cutoff_epoch
    now_epoch=$(date +%s)
    cutoff_epoch=$((now_epoch - ttl_hours * 3600))
```

```
    # 获取所有容器使用的镜像 ID 列表（完整格式）
    local -a used_image_ids
    mapfile -t used_image_ids < <(docker ps -aq | xargs -r docker inspect --format '{{.Image}}' |
    sort -u)

    local removed_count=0
    local examined_count=0
    declare -A seen_ids=() # 用于去重，避免同一镜像 ID 被多次处理

    # 遍历指定仓库下的所有镜像（--no-trunc 确保完整 ID 与 used_image_ids 一致）
    while read -r image_ref image_id; do
        [[ -z "$image_ref" || -z "$image_id" ]] && continue
        ((examined_count++))

        # 保留本任务将使用的镜像
        [[ "$image_ref" == "$current_image_ref" ]] && continue

        # 跳过已处理过的镜像 ID
        [[ -n "${seen_ids[$image_id]:-}" ]] && continue
        seen_ids[$image_id]=1

        # 永远不删除任何容器正在使用的镜像
        if printf '%s\n' "${used_image_ids[@]}" | grep -qx "$image_id"; then
            continue
        fi

        # 检查镜像创建时间，超过 TTL 或 aggressive 模式则删除
        local created created_epoch
        created=$(docker image inspect -f '{{.Created}}' "$image_id" 2>/dev/null || true)
        [[ -z "$created" ]] && continue
        created_epoch=$(date -d "$created" +%s 2>/dev/null || true)
        [[ -z "$created_epoch" ]] && continue

        if (( created_epoch < cutoff_epoch )) || [[ "$aggressive_cleanup" == "1" ]]; then
            if docker image rm -f "$image_id" >/dev/null 2>&1; then
                ((removed_count++))
            fi
        fi
    done < <(docker image ls --no-trunc "$repo_prefix" --format '{{.Repository}}:{{.Tag}} {{.ID}}')

    # 清理孤立的悬挂层，安全且不影响引用的镜像
    docker image prune -f --filter "until=${ttl_hours}h" >/dev/null 2>&1 || true
}
```

评论区精华

关键 Bug：镜像 ID 不匹配导致安全过滤失效

- Review 评论指出：run-intel-test.sh 中 grep -qx 检查时，used_image_ids 包含完整镜像 ID（如 sha256:abcdef...），而 image_id 来自 docker image ls 是短 ID，导致正在使用的镜像可能被误删。建议在 docker image ls 命令中添加 --no-trunc 参数以获取完整 ID。
- 结论：该建议已在 PR 的后续版本中采纳（补丁中确实包含了 --no-trunc 选项）。
- 镜像 ID 不匹配导致安全过滤失效 (correctness): 作者已采纳建议，在 docker image ls 命令中添加了 --no-trunc 参数，确保 ID 比较正确。

风险与影响

- 风险：
  1. 文件锁路径冲突：锁文件路径 /tmp/docker-pull.lock 可能在多个用户或不同 CI 阶段间冲突，但考虑到任务运行在相同节点且仅用于 CI 上下文，风险可控。
  2. 镜像清理逻辑可能缺陷：docker image ls --no-trunc 和 docker inspect 返回的 ID 格式一致性问题已修复，但其他边界条件（如日期解析失败、空镜像列表）已通过条件判断处理。
  3. 对 xpu 脚本的修改较小：仅移除了 remove_docker_container 中的镜像和系统清理，已通过 trap 确保容器被删除，风险较低。 - 影响：影响范围：仅限 Intel GPU CI 运行器（.buildkite/scripts/hardware_ci/），不会影响其他 CI 流水线或生产环境。影响程度：中等。修复了并行构建时的竞态问题，提高了 CI 稳定性，同时更精细的清理策略也能更快释放磁盘空间。对团队的影响：Intel CI 维护者无需额外操作，变更已自动生效。
- 风险标记：文件锁路径可能冲突，镜像 ID 比较已修复

关联脉络

- PR #36700 [Misc] Added curl retries in install_python_libraries.sh: 同为 CI 可靠性改进，关注脚本中的重试和容错机制。
- PR #40623 [CI] Split disaggregated tests into own test-area: CI 基础设施调整，涉及测试分区和资源管理，与本 PR 的清理策略优化方向一致。