PR #25241 完整报告

sgl-project/sglang

[diffusion] CI: fix nightly CI

合并时间：2026-05-16 16:55

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/25241

执行摘要

- 一句话：修复扩散 nightly CI 的端口竞争、OOM 和失败检测问题
- 推荐动作：值得精读，尤其是端口去重和 OOM 检测模式的设计，可为其他 CI 模块参考。

功能与动机

夜间 CI 中存在端口竞争、CUDA OOM 导致挂起、LTX-2.3 模型在 CFG parallel 2 下 OOM 等问题，此 PR 旨在通过端口严格分配、失败检测和内存优化来稳定 CI。

实现拆解

1. 端口管理强化：在 scripts/ci/utils/diffusion/run_comparison.py 中引入 _is_port_available 和 _require_ports_available，启动前预检查端口可用性；_build_sglang_cmd 增加 --strict-ports 和专用的 master/scheduler 端口偏移。
2. 失败检测与清理：新增 _cleanup_sglang_processes 通过 killall_sglang.sh 彻底清理；wait_for_health 中检测 CUDA OOM 日志（匹配 SERVER_FATAL_ERROR_PATTERNS），将故障立即标记为失败而非挂起；warmup 请求失败改为致命。
3. LTX-2.3 低 VRAM 内存释放：在 python/sglang/multimodal_gen/runtime/pipelines/ltx_2_pipeline.py 中新增 _release_stage2_for_low_vram，在 prepare_after_request 中当低 VRAM 模式 prefetch stage1 前主动释放 stage2（transformer_2）GPU 内存，降低峰值。
4. 端口去重与避免冲突：python/sglang/multimodal_gen/runtime/server_args.py 中 _adjust_network_ports 在 --strict-ports 下检查端口重复；settle_port 新增 avoid 参数避免多个服务占用同一端口。
5. 配置与工作流调整：comparison_configs.json 对 LTX-2.3 增加 --cfg-parallel-size 2；nightly 工作流跳过手动 dispatch 时的 sglang-ci-data 发布以避免权限错误。

关键文件：

- scripts/ci/utils/diffusion/run_comparison.py（模块 CI 脚本；类别 infra；类型 infrastructure；符号 kill_server, _is_port_available, _require_ports_available, _cleanup_sglang_processes）：核心 CI 脚本，引入端口预检、OOM 日志检测、进程清理机制，大幅提升 CI 鲁棒性。
- python/sglang/multimodal_gen/runtime/server_args.py（模块 服务配置；类别 source；类型 core-logic；符号 _adjust_network_ports, settle_port）：服务参数配置核心，强化端口去重逻辑，新增 avoid 参数避免多端口冲突，提升部署安全性。

- python/sglang/multimodal_gen/runtime/pipelines/ltx_2_pipeline.py（模块 扩散管道；类别 source；类型 core-logic；符号 _release_stage2_for_low_vram）：LTX-2.3 管道新增 stage2 释放逻辑，关键修复低 VRAM 模式下的 OOM。
- scripts/ci/utils/diffusion/comparison_configs.json（模块 CI 配置；类别 infra；类型 configuration）：更新 LTX-2.3 配置增加 --cfg-parallel-size 2 标志，与管道变更配合。

关键符号：kill_server, _is_port_available, _require_ports_available, _cleanup_sglang_processes, _release_stage2_for_low_vram, _adjust_network_ports, settle_port

关键源码片段

scripts/ci/utils/diffusion/run_comparison.py

核心 CI 脚本，引入端口预检、OOM 日志检测、进程清理机制，大幅提升 CI 鲁棒性。

```
# scripts/ci/utils/diffusion/run_comparison.py

import socket
import subprocess

# 定义端口偏移常量
SGLANG_MASTER_PORT_OFFSET = 5
SGLANG_SCHEDULER_PORT_OFFSET = 55

# CUDA OOM 错误模式
SERVER_FATAL_ERROR_PATTERNS = (
    "CUDA out of memory",
    "torch.OutOfMemoryError",
)

def _is_port_available(port: int) -> bool:
    """检查端口是否可用，使用 SO_REUSEADDR 避免 TIME_WAIT 干扰"""
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        try:
            sock.bind(("127.0.0.1", port))
        except OSError:
            return False
        return True

def _require_ports_available(ports: list[int]) -> None:
    """启动前确保所有必需端口可用"""
    unavailable = [port for port in ports if not _is_port_available(port)]
    if unavailable:
        raise RuntimeError(f"Required port(s) unavailable before launch: {unavailable}")

def _cleanup_sglang_processes() -> None:
    """调用 killall_sglang.sh 彻底清理残余 SGLang 进程"""
```

```
    KILLALL_SCRIPT = Path(__file__).parents[3] / "killall_sglang.sh"
    if KILLALL_SCRIPT.exists():
        subprocess.run(["bash", str(KILLALL_SCRIPT)], capture_output=True, check=False)

def kill_server(proc: subprocess.Popen) -> None:
    """终止服务进程树并清理 GPU 进程"""
    if proc.poll() is None:
        # 先尝试 SIGTERM
        try:
            os.killpg(os.getpgid(proc.pid), signal.SIGTERM)
        except (ProcessLookupError, PermissionError):
            pass
        try:
            proc.wait(timeout=30)
        except subprocess.TimeoutExpired:
            # 超时则强杀
            try:
                os.killpg(os.getpgid(proc.pid), signal.SIGKILL)
            except (ProcessLookupError, PermissionError):
                pass
            proc.wait(timeout=10)
    # 确保 GPU 残留进程被清理
    _cleanup_sglang_processes()
```

python/sglang/multimodal_gen/runtime/server_args.py

服务参数配置核心，强化端口去重逻辑，新增 avoid 参数避免多端口冲突，提升部署安全性。

```
# python/sglang/multimodal_gen/runtime/server_args.py ( 部分 )

def _adjust_network_ports(self):
    needs_http = self.disagg_role in (RoleType.MONOLITHIC, RoleType.SERVER)

    if self.strict_ports:
        # 收集所有请求端口，检测重复
        requested_ports = []
        if needs_http:
            requested_ports.append((self.port, "HTTP"))
        requested_ports.append((self.scheduler_port, "Scheduler"))
        if self.master_port is not None:
            requested_ports.append((self.master_port, "Master"))
        seen_ports: dict[int, str] = {}
        for port, name in requested_ports:
            if port in seen_ports:
                raise RuntimeError(
                    f"{name} port {port} duplicates {seen_ports[port]} port and "
                    "--strict-ports is enabled."
                )
            seen_ports[port] = name
            self._require_port(port, name)
```

```
    else:
        # 非 strict 模式：依次分配并记录已占用端口
        settled_ports: set[int] = set()
        if needs_http:
            self.port = self.settle_port(self.port)
            settled_ports.add(self.port)
        initial_scheduler_port = self.scheduler_port + (
            random.randint(0, 100) if self.scheduler_port == 5555 else 0
        )
        self.scheduler_port = self.settle_port(
            initial_scheduler_port, avoid=settled_ports
        )
        settled_ports.add(self.scheduler_port)
        if self.master_port is not None:
            self.master_port = self.settle_port(
                self.master_port, 37, avoid=settled_ports
            )

def settle_port(self, port, port_inc=42, max_attempts=100, avoid=None):
    """查找可用端口，avoid 参数避免与已占用端口冲突"""
    avoid = avoid or set()
    attempts = 0
    original_port = port
    while attempts < max_attempts:
        if port not in avoid and is_port_available(port):
            if attempts > 0:
                logger.info(f"Port {original_port} was unavailable, using port {port}")
            return port
        attempts += 1
        if port < 60000:
            port += port_inc
        else:
            port = original_port + attempts
    raise RuntimeError(f"Could not settle port after {max_attempts} attempts")
```

python/sglang/multimodal_gen/runtime/pipelines/ltx_2_pipeline.py

LTX-2.3 管道新增 stage2 释放逻辑，关键修复低 VRAM 模式下的 OOM。

```
# python/sglang/multimodal_gen/runtime/pipelines/ltx_2_pipeline.py ( 部分 )

def _release_stage2_for_low_vram(self) -> None:
    """在低 VRAM 模式下，释放 stage2 DiT 模块（transformer_2）的 GPU 内存，
    为 stage1 的加载腾出空间，避免峰值 OOM。"""
    stage2_module = self.pipeline.get_module("transformer_2")
    # 检查 stage2 参数的设备类型
    stage2_param = (
        next(stage2_module.parameters(), None)
        if stage2_module is not None
        else None
```

```
        )
        if stage2_param is not None and stage2_param.device.type == "cuda":
            # 将模块从 GPU 释放到 CPU 快照
            self._release_module_to_cpu_snapshot("transformer_2")

    def prepare_after_request(self, module, use, state):
        """请求结束后准备阶段，低 VRAM 模式下先释放 stage2 再 prefetch stage1"""
        phase = self._phase(use)
        if phase != "stage1":
            return
        if self.server_args.dit_cpu_offload:
            target_module = self.pipeline.get_module("transformer")
            if self._module_is_on_gpu(target_module):
                self._record_component_ready("transformer")
            elif not self._snapshot_strategy.is_ready("transformer"):
                if self._snapshot_low_vram_mode:
                    # 在 prefetch stage1 前释放 stage2，降低内存峰值
                    self._release_stage2_for_low_vram()
                self._snapshot_strategy.prefetch_component("transformer", target_module)
        else:
            self._record_component_ready("transformer")
        self.manager._sync_refinement_stage_transformer("stage1")
        self.manager._active_phase = "stage1"
```

评论区精华

- 暂无高价值评论线程

风险与影响

- 风险：
  1. 端口检测可靠性：_is_port_available 使用 SO_REUSEADDR 可能误报可用，但仅用于预检，后续 listen 仍可能失败。
  2. 进程清理覆盖：_cleanup_sglang_processes 调用 killall_sglang.sh 可能误杀其他 SGLang 进程。
  3. 低 VRAM 模式性能影响：释放 stage2 后后续请求需重新 load，增加延迟，但仅影响低 VRAM 场景。
  4. OOM 检测遗漏：日志模式匹配可能因语言或格式变化失效。 - 影响：范围：扩散 Nightly CI 流程、LTX-2.3 模型部署、服务端口配置。 程度：CI 稳定性显著提升（减少挂机和超时），低 VRAM 用户避免 OOM 崩溃；服务器端口行为更安全（去重）。
- 风险标记：端口冲突风险，进程清理覆盖范围，低 VRAM 模式性能影响，OOM 检测遗漏

关联脉络

- PR #24593 [diffusion] Generalize layerwise offload residency mixin to all components: 同样涉及扩散模型的内存管理和分层卸载，本 PR 中的 LTX 快照释放与

layerwise offload 机制相关。