PR #27139 完整报告

sgl-project/sglang

[FEAT] Support fast engine recovery through weight cache

合并时间：2026-07-25 14:31

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/27139

执行摘要

- 一句话：引入权重缓存守护进程，引擎重启提速至 <1 秒
- 推荐动作：里程碑级 PR，值得所有生产运维团队和架构师精读。重点关注的决策包括：(1) CUDA IPC 零拷贝 vs 进程级 checkpoint (CRIU+cuda-checkpoint) 的设计权衡；(2) 量化白名单机制确保数值正确性；(3) CacheConfig 指纹匹配策略（环境戳 + revision + 量化哈希）；(4) 进程生命周期管理（PR_SET_PDEATHSIG + force 清理）；(5) 与现有模块（权重更新、CPU 备份、内存保存器）的交互边界。该 PR 为后续快速重启路线图奠定了坚实基础，代码质量高、讨论充分，是值得学习的系统设计案例。

功能与动机

SGLang engine restarts are expensive. When an engine crashes, rolls out, or needs to restart for any reason, it must reload model weights from storage into GPU memory, re-initialize the distributed backend, recapture CUDA graphs, and re-warm JIT kernels — a process that takes 3–6+ minutes for large models(e.g., Qwen3-235B FP8 takes ~6.5 minutes on 4×GPU). Weight loading from disk dominates at ~79% of total startup time (~306/327 seconds). Eliminating weight loading is the single highest-leverage optimization. 此外，权重缓存守护进程还解锁了多实例权重共享、优先级协同服务（priority co-serving）、主动 - 备用故障切换（active-standby failover）等生产场景，这些在传统加载模式下因重启代价过高而难以实现。

实现拆解

1. 协议层设计：在 protocol.py 中定义 CacheConfig 类（基于 msgspec.Struct），包含模型路径、架构、TP/PP/DP/EP 规模、量化方法、量化配置哈希、dtype、revision、环境戳（CUDA 算力 + torch 版本）。实现长度前缀的 socket 消息帧协议 (send_msg/recv_msg) 和 IPC 量化方法白名单机制 (IPC_QUANT_ALLOWLIST)，仅允许 unquantized 和 block-wise FP8 通过 IPC 加载。
2. 守护进程实现：daemon.py 中的 WeightCacheDaemon 类是一个持久化进程，负责：初始化分布式环境 (_init_distributed)，使用 DefaultModelLoader 从磁盘加载完整模型权重并执行后量化处理，然后通过 _export_state 遍历所有参数和缓冲区，使用 torch.Tensor._share_cuda_ 导出为 CUDA IPC 句柄（base64 编码）。最后启动 Unix socket 服务循环 (serve)，在 CLIENT_CONNECTION_TIMEOUT (30s) 超时内接受引擎连接，验证 CacheConfig 匹配后发送所有句柄。

3. IPC 加载器：ipc_loader.py 中的 IpcModelLoader 类（继承 BaseModelLoader）是引擎端的加载器。通过 check_ipc_quant_support 硬性检查量化方法是否在白名单内。连接守护进程的 Unix socket，发送引擎的 CacheConfig 进行匹配验证。成功后接收 IPC 句柄列表，使用 meta device 初始化模型（_initialize_model 带 device_config='meta'），然后遍历句柄通过 torch.Tensor._new_shared_cuda 映射为 GPU 张量并赋值给对应参数 / 缓冲区。如果守护进程不可用但 socket 文件不存在，允许回退到磁盘加载；所有其他失败（connection refused、配置不匹配、协议错误）均硬性报错。
4. 引擎集成：在 engine.py 中新增 _launch_weight_cache_daemons 和 _terminate_weight_cache_daemons 类方法。前者根据 TP/PP 划分启动相应数量的守护进程子进程（使用 subprocess.Popen 避免父进程 CUDA 初始化干扰），并等待所有 .ready 文件就绪。shutdown() 负责优雅终止守护进程（先 SIGTERM 释放文件再 SIGKILL 剩余进程），确保 socket/ready 文件被清理。新增 --weight-cache-mode 参数（off/daemon/client）、--weight-cache-socket 等配置。
5. 安全与兼容性机制：实现 IPC_QUANT_ALLOWLIST 在守护进程和客户端两侧都硬性检查。检测 PYTORCH_CUDA_ALLOC_CONF 中的 expandable_segments:True 并提前失败。使用 PR_SET_PDEATHSIG 确保守护进程在父进程崩溃时自动终止。在权重更新路径（update_weights_from_disk、_distributed、_tensor、_ipc）中插入 _assert_weight_cache_inactive 防止零拷贝模式下的数据损坏，release_memory_occupation 中对应权重的分支也被保护。
6. 测试配套：新增 CPU 单元测试 test_weight_cache_protocol.py，覆盖协议帧协议、CacheConfig 匹配、量化哈希、白名单检查等，运行快速且无需 GPU。新增 GPU E2E 测试 test_weight_cache_daemon.py，包含 TP=2 的完整守护进程 + 客户端生成测试和 TP=1 的快速冒烟测试，通过日志断言（如 "Fetched ... IPC handles from daemon"）验证 IPC 路径被实际使用。

关键文件：

- python/sglang/srt/weight_cache/daemon.py（模块 权重守护；类别 source；类型 core-logic；符号 WeightCacheDaemon, init, _init_distributed, load）：核心守护进程实现，包含 WeightCacheDaemon 类的模型加载、IPC 句柄导出、Unix socket 服务循环、进程生命周期管理（pdeathsig、信号处理）等全部关键逻辑。新增线 944，是本 PR 最核心的源码文件。
- python/sglang/srt/weight_cache/ipc_loader.py（模块 IPC 加载；类别 source；类型 core-logic；符号 IpcModelLoader, init, load_model, _start_daemon_liveness_watchdog）：引擎端的 IPC 模型加载器，实现通过 CUDA IPC 从守护进程加载权重（零拷贝模式）。包含详细的 fallback vs raise 合同、daemon liveness watchdog、meta device 初始化、tied 权重处理。
- python/sglang/srt/weight_cache/protocol.py（模块 缓存协议；类别 source；类型 data-contract；符号 CacheConfig, matches, to_dict, from_dict）：协议定义和工具函数，包含 CacheConfig 指纹结构、IPC 量化白名单、环境戳计算、socket 消息帧协议、脏文件清理等。是整个框架的契约基础。
- python/sglang/srt/entrypoints/engine.py（模块 引擎入口；类别 source；类型 entrypoint；符号 _launch_weight_cache_daemons, _terminate_weight_cache_daemons）：引擎入

口集成了守护进程的启动、优雅终止、异常清理，以及多节点分布初始化验证。新增 _launch_weight_cache_daemons 和 _terminate_weight_cache_daemons 方法。

- python/sglang/srt/server_args.py（模块 服务配置；类别 source；类型 configuration）：新增 --weight-cache-mode（off/daemon/client）、--weight-cache-socket 等配置参数，以及相关的数据类字段和验证逻辑。移除 ipc_cache 从公共 LOAD_FORMAT_CHOICES，只允许通过 weight_cache_mode 内部选择。
- python/sglang/srt/model_executor/model_runner_components/load_model_utils.py（模块 模型加载；类别 source；类型 core-logic；符号 maybe_enable_ipc_weight_cache）：新增 maybe_enable_ipc_weight_cache 函数，在 model_runner 中根据配置启用 IPC 加载器并覆写 load format。
- python/sglang/srt/model_loader/loader.py（模块 加载器；类别 source；类型 dependency-wiring）：修改模型加载器工厂，增加对 IpcModelLoader 的注册和路由。
- python/sglang/srt/model_executor/model_runner_components/weight_updater.py（模块 权重更新；类别 source；类型 core-logic；符号 _assert_weight_cache_inactive）：在 update_weights_from_disk、_distributed、_tensor、_ipc 四个入口添加 _assert_weight_cache_inactive 断言，防止 IPC 零拷贝模式下权重更新破坏共享内存。
- python/sglang/srt/managers/scheduler_components/weight_updater.py（模块 调度权重；类别 source；类型 core-logic；符号 _assert_weight_cache_inactive）：在 release_memory_occupation 和 resume_memory_occupation 的权重内存分支添加 _assert_weight_cache_inactive 保护。
- test/registered/unit/model_loader/test_weight_cache_protocol.py（模块 协议测试；类别 test；类型 test-coverage；符号 _make_cache_config, TestProtocolFraming, test_round_trip, test_multiple_messages_are_framed_independently）：CPU-only 单元测试，覆盖协议帧、CacheConfig 匹配、量化哈希、白名单检查等，无需 GPU 即可运行。是关键的非 GPU 测试保障。
- test/registered/model_loading/test_weight_cache_daemon.py（模块 守护测试；类别 test；类型 test-coverage；符号 TestWeightCacheDaemonTP2, setUpClass, tearDownClass, test_generate）：GPU E2E 测试，包括 TP=2 的完整守护进程 + 客户端生成测试和 TP=1 的冒烟测试，通过日志断言验证 IPC 路径实际运行。

关键符号：WeightCacheDaemon.init, WeightCacheDaemon._init_distributed, WeightCacheDaemon.load, WeightCacheDaemon._assert_ipc_compatible_allocator, WeightCacheDaemon._export_state, WeightCacheDaemon.serve, WeightCacheDaemon._signal_handler, IpcModelLoader.init, IpcModelLoader.load_model, IpcModelLoader._start_daemon_liveness_watchdog, IpcModelLoader._daemon_alive, IpcModelLoader._watch, IpcModelLoader._resolve_engine_quant, IpcModelLoader._rebuild_stale_views, CacheConfig.matches, CacheConfig.to_dict, CacheConfig.from_dict, hash_quant_config, get_quant_method_name, check_ipc_quant_support, UnsupportedQuantForIPCError, _get_quant_field, engine._launch_weight_cache_daemons, engine._terminate_weight_cache_daemons, maybe_enable_ipc_weight_cache, weight_updater._assert_weight_cache_inactive

关键源码片段

python/sglang/srt/weight_cache/protocol.py

协议定义和工具函数，包含 CacheConfig 指纹结构、IPC 量化白名单、环境戳计算、socket 消息帧协议、脏文件清理等。是整个框架的契约基础。

```
# SPDX-License-Identifier: Apache-2.0
"""权重缓存守护进程的协议定义。
定义 CacheConfig 指纹、socket 消息帧协议、IPC 量化白名单等。
"""


class CacheConfig(msgspec.Struct):
    """缓存权重的指纹，用于验证守护进程与引擎之间的兼容性。
    任何不匹配都会触发磁盘 fallback 或硬性报错。
    """
    model_path: str # 模型路径
    model_arch: str # 模型架构（如 LlamaForCausalLM）
    tp_size: int # 张量并行度
    tp_rank: int # 当前 TP rank
    pp_size: int # 流水线并行度
    pp_rank: int # 当前 PP rank
    dp_size: int # 数据并行度
    ep_size: int # 专家并行度
    quant_method: str # 量化方法，如 "fp8"、"gptq_marlin"、"" 无量化
    quant_config_hash: str # 量化配置的 SHA-256 哈希
    dtype: str # 例如 "torch.float16"
    revision: str # 模型版本（"" 表示未设置）
    # 环境戳：不同 GPU 算力或 torch 版本可能产生不兼容的权重
    device_capability: str # 本地计算能力，如 "8.0"
    torch_version: str # torch.__version__

    def matches(self, other: "CacheConfig") -> bool:
        """检查两个配置是否兼容。"""
        return self == other

    def to_dict(self) -> Dict[str, Any]:
        return {f: getattr(self, f) for f in self.__struct_fields__}

    @classmethod
    def from_dict(cls, d: Dict[str, Any]) -> "CacheConfig":
        return cls(**d)


# ---------------------------------------------------------------------------
# IPC 量化方法白名单
# ---------------------------------------------------------------------------
# CUDA IPC 零拷贝共享只导出原始张量数据，因此仅在
# process_weights_after_loading 的完整效果被张量数据捕获时才正确。
# 需要 Python 元数据（如 block-FP8 的 format_ue8m0）或重排 / 转置的方法
```

```
# （per-tensor FP8、Marlin、AWQ/GPTQ）会提供静默错误。
# 只有验证可 round-trip 通过纯张量导出的方法才被列入白名单。
IPC_QUANT_ALLOWLIST: Dict[str, List[str]] = {
    # "" 表示未量化的模型
    "": ["*"],
    # block-wise FP8：仅当 weight_block_size 已设置（非默认）
    "fp8": ["*"],
}


def check_ipc_quant_support(quant_method: str, quant_config: Any, where: str):
    """检查量化方法是否支持 IPC 零拷贝。不支持时硬性报错。"""
    if quant_method not in IPC_QUANT_ALLOWLIST:
        raise UnsupportedQuantForIPCError(
            f"Quantization method '{quant_method}' is not supported for IPC "
            f"weight cache (checked in {where}). Supported methods: "
            f"{list(IPC_QUANT_ALLOWLIST.keys())}. "
            f"Use `--quantization fp8` with `--weight-block-size` or load "
            f"without quantization for IPC mode."
        )


def hash_quant_config(quant_config: Any) -> str:
    """计算量化配置的稳定哈希。
    避免使用 str()/repr() 因为嵌入内存地址会导致不同进程产生不同哈希。
    """
    if quant_config is None:
        return ""
    try:
        if hasattr(quant_config, "to_dict"):
            config_str = json.dumps(quant_config.to_dict(), sort_keys=True)
        elif isinstance(quant_config, dict):
            config_str = json.dumps(quant_config, sort_keys=True)
        elif hasattr(quant_config, "__dict__"):
            config_str = type(quant_config).__name__ + ":" + json.dumps(
                {k: v for k, v in sorted(quant_config.__dict__.items())
                 if not k.startswith("_")
                 and isinstance(v, (str, int, float, bool, type(None), list, dict))},
                sort_keys=True)
        else:
            config_str = type(quant_config).__name__
        return hashlib.sha256(config_str.encode()).hexdigest()
    except Exception:
        config_str = type(quant_config).__name__
        return hashlib.sha256(config_str.encode()).hexdigest()
```

评论区精华

1. IPC 量化方法兼容性（alexnails & Claude）：_post_load_weights 跳过后，某些量化方法（如 per-tensor FP8）的 Python 属性（如 weight_scale_inv.format_ue8m0）不会跨 IPC 传递，导致静默输出错误。作者引入 IPC_QUANT_ALLOWLIST 白名单解决，仅允许 unquantized 和 block-wise FP8 通过。
2. 守护进程生命周期与脏文件清理（Kangyan-Zhou, alexnails & Claude）：引擎崩溃后守护进程成为孤儿，残留 .ready/.sock 文件阻止下次启动。作者通过 PR_SET_PDEATHSIG、shutdown() 中先 SIGTERM 再 SIGKILL、cleanup_stale_daemon_files 添加 --force 标志解决。
3. 连接超时与 socket 泄漏（Kangyan-Zhou & alexnails）：守护进程 accept 后未设置超时，一个挂起客户端会阻塞所有其他引擎等级。作者添加 conn.settimeout(30s)。socket 连接失败时未关闭导致泄漏，已修复。
4. 扩展段内存分配器不兼容（alexnails & Claude）：PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True 使 _share_cuda_ 失败。作者添加 _assert_ipc_compatible_allocator 提前报错。
5. 权重在线更新保护（alexnails & Claude）：IPC 零拷贝模式下 update_weights_from_disk 等操作会损坏共享内存。作者在 weight_updater.py 的四个入口处插入 _assert_weight_cache_inactive 硬性报错。
6. 测试覆盖计划（Kangyan-Zhou & alexnails）：初始版本仅手动测试。作者添加 CPU 单元测试（无 GPU）和 GPU E2E 测试，TP=1 冒烟测试始终运行在 base-b 阶段，TP=2 测试作为 extra-a 阶段。

- IPC 量化方法兼容性 (correctness): 引入 IPC_QUANT_ALLOWLIST 白名单，仅允许 unquantized 和 block-wise FP8（weight_block_size 已设置）通过 IPC 加载，其他方法硬性报错。
- 守护进程孤儿进程与脏文件清理 (security): 添加 PR_SET_PDEATHSIG（kill_itself_when_parent_died）；shutdown() 先 SIGTERM 后 SIGKILL；cleanup_stale_daemon_files 添加 --force 标志手动接管。
- 连接超时与 socket 泄漏 (correctness): 添加 conn.settimeout(30s)；修复 socket 泄漏（使用 with 或 try/finally）。
- 扩展段内存分配器不兼容 (performance): 添加 _assert_ipc_compatible_allocator() 在 load 开始时检测并提前报错。
- 权重在线更新保护 (correctness): 在 weight_updater.py 的四个入口（disk/distributed/tensor/ipc）添加 _assert_weight_cache_inactive 硬性报错；scheduler_components/weight_updater.py 的 release_memory_occupation 的权重分支也添加保护。
- PP/EP/DP 参数传递遗漏 (correctness): 已修复为从 server_args 传递所有维度参数给 daemon 构造器。
- 测试覆盖计划 (testing): 添加 CPU 单元测试（test_weight_cache_protocol.py，无 GPU）和 GPU E2E 测试（test_weight_cache_daemon.py，TP=1 冒烟测试始终运行在 base-b 阶段）。
- 全局参数 vs 实际并行配置不一致 (correctness): daemon 构造器接受独立的 pp_size/dp_size/ep_size 参数，由调用方从实际配置传入，不再依赖全局 ServerArgs。

风险与影响

- 风险：
  - 兼容性风险：IPC 零拷贝仅支持 unquantized 和 block-wise FP8（weight_block_size 已设置），其他量化方法（per-tensor FP8、Marlin、AWQ/GPTQ 等）在守护进程和客户端两侧都会硬性报错，不存在静默错误。环境戳（CUDA 算力、torch 版本）不匹配也会导致拒绝服务，但这是保护机制而非风险。
  - 进程生命周期风险：尽管添加了 PR_SET_PDEATHSIG 保护，在极端场景（如父进程被 SIGKILL 后子进程被 reparent 到 init）仍可能产生 orphan 守护进程，残留 GPU 内存。--force 标志提供了手动恢复手段。
  - 内存安全风险：零拷贝模式下引擎和守护进程共享 GPU 内存，已通过 _assert_weight_cache_inactive 禁止权重更新操作，并强制 requires_grad=False 防止 autograd 意外修改。但若第三方代码直接操作 param.data 仍可能破坏共享数据。
  - 性能风险：--weight-cache-mode daemon 模式下首次启动比直接加载更慢（因为多一次磁盘加载到守护进程）。只有在守护进程预启动（standalone launcher）并使用 --weight-cache-mode client 时才能体现加速优势，需在文档中明确说明。
  - 安全风险：Unix socket 文件位于 /tmp/，同一节点上其他进程可尝试连接。建议限制 socket 文件权限（当前实现未显示设置）。
  - 分布式初始化风险：多节点场景下若未设置 --dist-init-addr，守护进程无法形成联合 NCCL 组，引擎会陷入 readiness 超时（默认 1800s）。已在 engine.py 中添加对应验证，提前报错。
- 影响：
  - 用户影响：主要面向生产环境部署用户。通过 --weight-cache-mode client 配合预启动守护进程可大幅缩短引擎重启时间（从分钟级到秒级），需增加守护进程的部署和管理。默认 off，不影响现有用户。
  - 系统影响：每个 GPU 上多运行一个守护进程（额外 GPU 内存占用：零拷贝模式下引擎和守护进程共享同一份权重，不增加整体内存使用，但守护进程本身需常驻 GPU 上下文）。CPU 和网络开销极小。
  - 团队影响：新增约 2000 行核心代码和 1000 行测试，需要维护与现有特性（推测解码、睡眠模式、多节点、弹性 EP 等）的兼容性。后续 Roadmap（CUDA 图序列化、内核预热缓存等）依赖此基础结构。
  - 兼容性影响：未对现有 API 和协议做破坏性变更，参数均为新增可选参数。需确保与 model_runner.py、weight_updater.py 等模块的交互正确。
  - 风险标记：量化白名单限制，守护进程孤儿风险，扩展段不兼容，共享内存更新保护，首次启动延迟，分布式初始化超时

关联脉络

- 暂无明显关联 PR