PR #33279 完整报告

sgl-project/sglang

[FEAT] Weight Daemon abstraction

合并时间：2026-08-22 17:13

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/33279

执行摘要

- 一句话：Weight Daemon 引入可插拔传输后端，默认路径保持 torch_ipc
- 推荐动作：值得 weight_cache 与模型加载相关开发者精读。核心看点：可插拔后端抽象的接口切分（导出 / 响应 / 接收 / 导入四段）、daemon 与 client 通过协议字段协商后端的设计、以及占位后端 fail loud 的降级策略（宁抛错不静默回退）。一般读者重点阅读 transport.py 的接口设计与 daemon._export_state 的接入方式即可；同时留意 _send_fd/_recv_fd 尚未启用，启用 VMM 后端前应补齐 GPU E2E 与 fd 故障测试。

功能与动机

PR body 是空模板，未填写实质内容，动机需要从代码意图推断。daemon 原先将张量导出 / 导入硬编码为 torch multiprocessing 序列化（MultiprocessingSerializer 输出 base64 字符串 handle），无法支撑更高效的 CUDA VMM + fd 传递方案。VmmFdTransportBackend 的 docstring 明确说明这是“Placeholder for the CUDA VMM + fd-passing transport. The backend is not wired up yet”；ipc_loader.py 模块 docstring 也写明 “Backends are negotiated per daemon response (torch IPC by default, VMM FD when available)”，即目标是让 daemon 与 client 通过响应协商传输方式，为后续跨进程共享同一物理 GPU 内存（零拷贝）铺路。

实现拆解

1. 新增传输后端抽象层（python/sglang/srt/weight_cache/transport.py）：定义 WeightCacheTransportBackend ABC，包含 prepare_export、send_fetch_state_response、recv_fetch_state_response、import_tensor 四个方法，把 daemon 导出与 client 导入彻底解耦。同文件实现 TorchIpcTransportBackend（默认）与 VmmFdTransportBackend（占位），并提供 choose_daemon_transport_backend 与 get_client_transport_backend 两个后端选择入口。
2. daemon 侧迁移（python/sglang/srt/weight_cache/daemon.py）：_export_state 不再直接调用 MultiprocessingSerializer.serialize，而是先收集 state_tensors（参数、持久 buffer、非持久 buffer 统一为 (tensor, is_param) 映射），再由 choose_daemon_transport_backend 选择后端并调用 prepare_export；_handle_connection 中成功响应改由后端 send_fetch_state_response 发出，并在响应中注入 transport_backend 字段；同时移除对 MultiprocessingSerializer 的导入。

3. client 侧迁移（python/sglang/srt/weight_cache/ipc_loader.py）：_fetch_from_cache 读取响应中的 transport_backend 字段（缺省视为 torch_ipc 以兼容旧 daemon），通过 get_client_transport_backend 构造后端并调用 recv_fetch_state_response；_load_zero_copy_mode 中 MultiprocessingSerializer.deserialize 替换为 backend.import_tensor(entry)；后端实例挂在 model._weight_cache_transport_backend 上防止被 GC（为未来 VMM 后端持有 VA 映射做准备）。
4. fd 传递原语与兜底逻辑：transport.py 内置 _send_fd / _recv_fd（基于 SCM_RIGHTS 的 fd 传递，含截断校验与异常时 fd 清理），当前仅供未来 VMM 后端使用；VmmFdTransportBackend 构造即抛 NotImplementedError、can_export_state 恒返回 False，保证 daemon 不会静默选到未就绪后端；client 端遇到未知后端名直接抛 RuntimeError，fail loud。
5. 测试配套（test/registered/unit/model_loader/test_weight_cache_protocol.py）：新增 TestTransportBackend，三个用例覆盖 get_client_transport_backend(None) 默认返回 torch_ipc、未知名称抛 RuntimeError、以及 TorchIpcTransportBackend 在真实 socketpair 上的导出→发送→接收→导入全链路 round-trip（用 torch.equal 校验张量内容）。

关键文件：

- python/sglang/srt/weight_cache/transport.py（模块 传输后端；类别 source；类型 core-logic；符号 _send_fd, _recv_fd, WeightCacheTransportBackend, prepare_export）：本 PR 的核心新增文件，定义可插拔传输后端抽象与 torch_ipc/vmm_fd 两个实现，是 daemon 与 client 解耦的桥梁。
- python/sglang/srt/weight_cache/daemon.py（模块 权重缓存；类别 source；类型 dependency-wiring；符号 _export_state, serve, _handle_connection）：daemon 导出与响应发送路径迁移到传输后端，是后端抽象在服务端的落地点。
- python/sglang/srt/weight_cache/ipc_loader.py（模块 模型加载；类别 source；类型 dependency-wiring；符号 _fetch_from_cache, _load_zero_copy_mode, load_model, init）：client 侧按 daemon 响应协商后端并改用 import_tensor 导入，是后端抽象在客户端的关键接入点。
- test/registered/unit/model_loader/test_weight_cache_protocol.py（模块 单元测试；类别 test；类型 test-coverage；符号 TestTransportBackend, test_default_backend_is_torch_ipc, test_unknown_backend_raises, test_torch_ipc_backend_round_trip）：新增后端抽象相关的 CPU 单测，覆盖默认选择、未知后端报错与 torch_ipc 全链路 round-trip。

关键符号：choose_daemon_transport_backend, get_client_transport_backend, prepare_export, send_fetch_state_response, recv_fetch_state_response, import_tensor, _send_fd, _recv_fd, _export_state, _fetch_from_cache, _load_zero_copy_mode

关键源码片段

python/sglang/srt/weight_cache/transport.py

本 PR 的核心新增文件，定义可插拔传输后端抽象与 torch_ipc/vmm_fd 两个实现，是 daemon 与 client 解耦的桥梁。

```
# SPDX-License-Identifier: Apache-2.0
"""可插拔的张量传输后端：daemon 导出与 client 导入的解耦点。"""

from abc import ABC, abstractmethod
from typing import Any, Dict, Mapping, NoReturn, Optional, Tuple

import torch

from sglang.srt.utils import MultiprocessingSerializer
from .protocol import send_msg

TORCH_IPC_BACKEND = "torch_ipc"
VMM_FD_BACKEND = "vmm_fd"


class WeightCacheTransportBackend(ABC):
    """传输后端抽象：daemon 负责 prepare_export / send_fetch_state_response，
    client 负责 recv_fetch_state_response / import_tensor，
    两侧通过响应里的 transport_backend 字段对齐实现。"""

    name: str

    @abstractmethod
    def prepare_export(
        self, state_tensors: Mapping[str, Tuple[torch.Tensor, bool]]
    ) -> Dict[str, Dict[str, Any]]:
        """daemon 侧：把全部张量转成可跨进程传输的 entry 元数据。"""

    @abstractmethod
    def send_fetch_state_response(self, conn, *, config, entries, pid) -> None:
        """daemon 侧：发送成功的 fetch_state 响应。"""

    @abstractmethod
    def recv_fetch_state_response(self, sock, result) -> Dict[str, Any]:
        """client 侧：收到响应后的后续处理（例如读取 fd 附属数据）。"""

    @abstractmethod
    def import_tensor(self, entry: Dict[str, Any]) -> torch.Tensor:
        """client 侧：从单个 entry 恢复出张量。"""


class TorchIpcTransportBackend(WeightCacheTransportBackend):
    """默认实现：沿用 torch multiprocessing 序列化（base64 字符串 handle），
    行为与抽象引入前完全一致，保证零回归。"""

    name = TORCH_IPC_BACKEND
```

```
    def prepare_export(self, state_tensors):
        entries = {}
        for name, (tensor, is_param) in state_tensors.items():
            entries[name] = {
                "handle": MultiprocessingSerializer.serialize(tensor.data, output_str=True),
                "shape": list(tensor.shape),
                "dtype": str(tensor.dtype).replace("torch.", ""),
                "is_param": is_param,
            }
        return entries

    def send_fetch_state_response(self, conn, *, config, entries, pid):
        send_msg(conn, {
            "status": "ok",
            "config": config,
            "entries": entries,
            "pid": pid,
            "transport_backend": self.name, # 新增字段，client 据此协商后端
        })

    def recv_fetch_state_response(self, sock, result):
        return result # torch_ipc 无需额外处理

    def import_tensor(self, entry):
        return MultiprocessingSerializer.deserialize(entry["handle"])


class VmmFdTransportBackend(WeightCacheTransportBackend):
    """CUDA VMM + fd 传递后端占位。当前未接线：can_export_state 恒返回 False，
    因此 daemon 始终选 torch_ipc；其他入口一律 fail loud，拒绝静默返回 None。"""

    name = VMM_FD_BACKEND

    def __init__(self):
        self._raise_not_implemented()

    @staticmethod
    def _raise_not_implemented() -> NoReturn:
        raise NotImplementedError(
            f"weight cache transport backend {VMM_FD_BACKEND!r} is not implemented in this
            build"
        )

    @classmethod
    def can_export_state(cls, state_tensors) -> bool:
        return False # 未就绪，恒 False


def choose_daemon_transport_backend(state_tensors):
```

```
    """daemon 侧选择器：未来 VMM 具备导出能力时自动切换，当前固定 torch_ipc。"""
    if VmmFdTransportBackend.can_export_state(state_tensors):
        return VmmFdTransportBackend()
    return TorchIpcTransportBackend()


def get_client_transport_backend(name):
    """client 侧按名称构造后端；未知名称直接抛错，避免静默降级。"""
    if name in (None, "", TORCH_IPC_BACKEND):
        return TorchIpcTransportBackend()
    if name == VMM_FD_BACKEND:
        return VmmFdTransportBackend()
    raise RuntimeError(f"Unknown weight cache transport backend {name!r}")
```

python/sglang/srt/weight_cache/daemon.py

daemon 导出与响应发送路径迁移到传输后端，是后端抽象在服务端的落地点。

```
    def _export_state(self):
        """把模型全部参数/缓冲交给所选传输后端导出，daemon 不再关心序列化格式。"""
        self.state_entries.clear()

        # remove_duplicate=False：tied weight 在每个名字下都被识别为参数，
        # 避免 client 侧把重复键误注册成 buffer
        param_names = set(
            name for name, _ in self.model.named_parameters(remove_duplicate=False)
        )
        state_dict_names = set(self.model.state_dict().keys())
        state_tensors: Dict[str, Tuple[torch.Tensor, bool]] = {}

        # 参数 + 持久 buffer 统一收集；非持久 buffer（如 RoPE cos_sin_cache）
        # 不在 state_dict 里，单独补上，保证 client 能完整重建模型状态
        for name, tensor in self.model.state_dict().items():
            state_tensors[name] = (tensor.data, name in param_names)

        non_persistent_count = 0
        for name, buf in self.model.named_buffers():
            if name not in state_dict_names:
                state_tensors[name] = (buf.data, False)
                non_persistent_count += 1

        # 选择后端并导出；这里是新增的抽象接入点
        self.transport_backend = choose_daemon_transport_backend(state_tensors)
        self.state_entries = self.transport_backend.prepare_export(state_tensors)

        # 日志只统计 handle 本体（str/bytes）的长度，避免 stringify 整个 entry
        # 造成大对象拷贝（review 中提出的性能问题）
        total_bytes = sum(
            len(handle)
            for handle in (entry.get("handle") for entry in self.state_entries.values())
```

```
            if isinstance(handle, (str, bytes, bytearray))
        )
        logger.info(
            f"[WeightCacheDaemon gpu={self.gpu_id}] "
            f"Exported {len(self.state_entries)} tensors "
            f"({non_persistent_count} non-persistent buffers), "
            f"transport={self.transport_backend.name}, "
            f"metadata size ~{total_bytes / 1024 / 1024:.1f} MB"
        )
```

评论区精华

两条 review 评论均已解决：

- alexnails 在 daemon.py 的 _export_state 差异上提问“len(str(entry).encode("utf-8")) is expensive?”，担心 stringify 整个 entry 会带来不必要的拷贝开销。作者回复 fixed，最终实现改为只统计 handle 本体（str/bytes/bytearray）的长度，避免对完整 entry 做字符串化，日志口径也改为 metadata size。
- alexnails 在 transport.py 的 VmmFdTransportBackend._raise_not_implemented 上建议采用带后端名的 f-string 错误消息（something like ...），作者采纳并回复 fixed。
- daemon 导出日志统计方式的性能疑问 (performance): 作者回复 fixed，最终实现改为仅统计 handle（str/bytes/bytearray）长度，绕过对完整 entry 的字符串化，并把日志措辞改为 metadata size。
- VmmFdTransportBackend 未实现错误消息格式 (style): 作者回复 fixed，最终实现为 f"weight cache transport backend {VMM_FD_BACKEND!r} is not implemented in this build"。

风险与影响

- 风险：
  1. 加载主路径被重构（daemon._export_state 与 IpcModelLoader._load_zero_copy_mode），虽然默认后端行为等价，但 GPU 端 E2E 测试（test_weight_cache_daemon.py）不在本 PR 修改范围，CPU 单测无法覆盖 CUDA IPC 映射细节，存在回归未被单测捕获的风险。
  2. 协议新增 transport_backend 字段：client 用 result.get("transport_backend", TORCH_IPC_BACKEND) 缺省兜底，对旧 daemon 兼容；但若未来 daemon 返回 vmm_fd 而 client 侧该后端未实现（构造即抛 NotImplementedError），会硬失败——这是有意的 fail loud 策略，但跨版本混用时需要提前规划。
  3. _send_fd / _recv_fd 已随本 PR 提交但未启用：SCM_RIGHTS 的 CMSG 边界处理、fd 数量校验、部分发送失败等边界条件缺乏真实场景验证，未来启用 VMM 后端前需要补 fd 层面的故障注入测试。
  4. 日志统计口径变化：total_bytes 只统计 str/bytes/bytearray 类型的 handle，若未来后端改用整型 fd 或对象句柄，日志会低估元数据大小，可能误导排查。 - 影响：对用户与运行时：默认路径无行为变化，仍是 torch_ipc 后端，协议响应新增字段对旧客户端透明。

对系统架构：weight_cache 模块完成导出 / 导入与具体序列化方式的解耦，后续CUDA VMM 后端只需实现 WeightCacheTransportBackend 并在两个选择函数中注册即可。对团队协作：为 weight_cache 的多后端路线（跨进程零拷贝、降低序列化开销）划定了扩展点，风险集中在版本兼容与 GPU E2E 验证上。 - 风险标记：核心加载路径变更，协议字段新增，fd 传递安全面，GPU E2E 未覆盖

关联脉络

- 暂无明显关联 PR