PR #34736 完整报告

sgl-project/sglang

[Diffusion] Unify component residency controls

合并时间：2026-08-16 11:24

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/34736

执行摘要

- 一句话：新增 --component-residency 统一组件驻留控制，重构 offload 策略矩阵
- 推荐动作：该 PR 值得精读，尤其是 component_residency.py 的解析与 server_args.py 的 explicit_residency_mode 折算逻辑，是配置兼容性设计的良好范例。关注点建议放在：canonical 与 legacy 混用时的边界行为、start_at_stage_entry 对缓存命中路径的性能影响、以及动态组件替换后的清理策略。对于集成方，若有自定义 stage，必须实现 component_uses 与 use_declared_component 契约，否则默认 resident 行为可能不符合预期。

功能与动机

PR body 指出：SGLang-Diffusion 当前暴露了重叠的 CPU-offload 与 layerwise-offload 标志，作用域不一；显式 --dit-layerwise-offload false 仍可能继承自动 DiT offload，调用方请求 resident DiT 却无法保证；DiT/VAE 布尔开关还隐式控制模型特定辅助组件，随着 pipeline 动态模块增多，放置行为难以推理。因此需要统一组件驻留控制，让每个组件都能被精确指定模式，同时保留全部历史开关的兼容性。

实现拆解

实现分以下步骤拆解：

1. 新增 canonical 配置层：在 component_residency.py 中定义 normalize_component_residency（解析 COMPONENT=MODE 赋值，支持字符串、序列与映射，规范化为字典）、component_residency_selector_matches（精确组件名优先，其次按 dit/text_encoder/image_encoder/vae 分组匹配，最后回退到 all）与 resolve_component_residency_mode（按精确 > 分组 > all 的优先级解析单一组件的最终模式），并为 Diffusers 后端提供 resolve_diffusers_pipeline_offload 校验，限制其仅支持 pipeline-wide 的 resident/component-offload。
2. 参数层兼容与归一：server_args.py 新增 component_residency 字段，调整 _adjust_parameters 顺序，先做 _normalize_component_residency 再执行 legacy 归一；explicit_residency_mode 将 canonical 选择器、legacy CPU offload 标志、--dit-layerwise-offload 显式值统一折算成单一模式，并特意处理 --dit-layerwise-offload false 使 DiT 保持 resident；should_start_component_on_cpu 与 residency_mode 成为加载器与策略的唯一入口。

3. 运行时策略重写：删除旧的 component_resident_strategies.py（含 Snapshot 策略、VanillaD2H），新增 component_residency_strategies.py，只保留 ResidentStrategy、ComponentOffloadStrategy（带异步 prefetch 与 warmup 保持行为）和 LayerwiseOffloadStrategy（仅驱动已配置的 layerwise 生命周期），去掉了重复的 enter/exit 抽象，策略直接对应 CLI 模式；is_fsdp_managed_module 改用 isinstance(module, FSDPModule) 而非类名前缀。
4. 生命周期与加载器接入：component_manager.py 引入 begin_stage/end_stage 钩子，单组件 stage 在进入时即声明使用；ComponentUse 增加 start_at_stage_entry 字段，使 realtime 文本编码等缓存命中路径可在调用点再准备组件；begin_use/finish_use 与策略缓存按组件名绑定，forget_module 清理动态组件替换后的陈旧状态；component_loader.py 移除层叠配置逻辑，统一通过 server_args.should_start_component_on_cpu 决定初始设备，FSDP 模块跳过 CPU 放置。
5. 模型特定 stage 改造与测试：cosmos3.py、hunyuan3d/shape.py、glm_image.py 等 stage 声明 component_uses，将手工 _manage_device_placement 和 vae_cpu_offload 分支替换为 use_declared_component 上下文；text_encoding.py 调整 _begin_text_encoder_use 以支持调用点延迟；新增 test_component_residency.py、扩展 test_server_args.py与test_layerwise_offload.py覆盖优先级、legacy混合、动态组件、warmup 保持、FSDP 冲突与 Diffusers 生效放置。文档同步更新 multimodal-gen README 与 SGLang-Diffusion CLI/ 部署说明。

关键文件：

- python/sglang/multimodal_gen/runtime/managers/memory_managers/component_residency.py（模块 驻留策略；类别 source；类型 core-logic；符号 ComponentResidencyError, normalize_component_residency, component_residency_selector_matches, resolve_component_residency_mode）：新增核心解析模块，定义 canonical 模式与选择器优先级，是全部配置的统一入口。
- python/sglang/multimodal_gen/runtime/managers/memory_managers/component_residency_strategies.py（模块 内存管理器；类别 source；类型 core-logic；符号 ComponentResidencyStrategy, ResidentStrategy, ComponentOffloadStrategy, LayerwiseOffloadStrategy）：新增运行时驻留策略实现，直接映射三种 CLI 模式，是生命周期执行的核心。
- python/sglang/multimodal_gen/runtime/server_args/server_args.py（模块 参数解析；类别 source；类型 dependency-wiring；符号 _normalize_component_residency, should_cpu_offload_component, canonical_residency_mode, explicit_residency_mode）：参数层新增 component_residency 字段，并实现 legacy 与 canonical 的归一化与显式模式解析，是所有兼容性逻辑落地点。
- python/sglang/multimodal_gen/runtime/managers/memory_managers/component_manager.py（模块 组件管理；类别 source；类型 dependency-wiring；符号 ComponentUse, build_component_residency_strategy, begin_stage, end_stage）：生命周期管理器核心变更：新增 begin_stage/end_stage 钩子、start_at_stage_entry 字段、策略缓存与 forget_module 清理，是动态组件与缓存命中优化的基础。

- python/sglang/multimodal_gen/runtime/loader/component_loaders/component_loader.py（模块 加载器；类别 source；类型 dependency-wiring；符号 target_device, should_offload, _is_component_set_as_layerwise_load, _maybe_configure_layerwise_after_startup_cpu_staging）：移除 loader 内 layerwise 修补逻辑，统一通过 should_start_component_on_cpu 决策初始设备，简化加载路径。
- python/sglang/multimodal_gen/test/unit/test_component_residency.py（模块 单元测试；类别 test；类型 test-coverage；符号 test_component_offload_releases_preferred_component_after_request, test_component_offload_keeps_preferred_component_after_warmup, test_request_tail_uses_dynamic_component_instance, test_strategy_cache_replaces_stale_component_instance）：新增 398 行单元测试，覆盖组件驻留策略选择、warmup 保持、动态实例替换、forget_module 清理与 realtime 调用点延迟等关键行为。

关键符号：normalize_component_residency, component_residency_selector_matches, resolve_component_residency_mode, resolve_diffusers_pipeline_offload, explicit_residency_mode, residency_mode, should_start_component_on_cpu, build_component_residency_strategy, begin_stage, end_stage, begin_use, finish_use, forget_module, use_declared_component, component_uses, target_device, prepare_for_use, finish_request, prefetch_for_use

关键源码片段

python/sglang/multimodal_gen/runtime/managers/memory_managers/component_residency.py

新增核心解析模块，定义 canonical 模式与选择器优先级，是全部配置的统一入口。

```
# component_residency.py — 统一组件驻留 CLI 模式的解析与解析
# 纯函数层，不依赖 ServerArgs 具体字段，便于独立测试。

from collections.abc import Mapping, Sequence

RESIDENT = "resident"
COMPONENT_OFFLOAD = "component-offload"
LAYERWISE_OFFLOAD = "layerwise-offload"
COMPONENT_RESIDENCY_MODES = frozenset((RESIDENT, COMPONENT_OFFLOAD,
LAYERWISE_OFFLOAD))

class ComponentResidencyError(ValueError):
    """无效或不支持的组件驻留选择。"""

# 分组优先级：精确组件名最高，然后按组，最后 all。
COMPONENT_RESIDENCY_GROUP_PRECEDENCE = (
    "dit", "text_encoder", "image_encoder", "vae",
)


def normalize_component_residency(
```

```
    assignments: str | Sequence[str] | Mapping[str, str] | None,
) -> dict[str, str] | None:
    """把各种形式的输入统一成 selector -> mode 字典。"""
    if assignments is None:
        return None
    if isinstance(assignments, Mapping):
        entries = assignments.items()
    else:
        # 支持 'transformer=resident,text_encoder=component-offload' 这种逗号字符串
        values = [assignments] if isinstance(assignments, str) else assignments
        parsed_entries = []
        for value in values:
            for assignment in str(value).split(","):
                assignment = assignment.strip()
                if not assignment:
                    continue
                if "=" not in assignment:
                    raise ComponentResidencyError(
                        "Component residency must use COMPONENT=MODE, got "
                        f"{assignment!r}"
                    )
                selector, mode = assignment.split("=", 1)
                parsed_entries.append((selector.strip(), mode.strip()))
        entries = parsed_entries

    normalized = {}
    for raw_selector, raw_mode in entries:
        # 统一大小写与分隔符：selector 用下划线，mode 用连字符
        selector = raw_selector.replace("-", "_").lower()
        mode = raw_mode.replace("_", "-").lower()
        if not selector:
            raise ComponentResidencyError("Component residency selector cannot be empty")
        if mode not in COMPONENT_RESIDENCY_MODES:
            expected = ", ".join(sorted(COMPONENT_RESIDENCY_MODES))
            raise ComponentResidencyError(
                f"Invalid component residency mode {raw_mode!r} for {selector!r}; "
                f"expected one of: {expected}"
            )
        normalized[selector] = mode
    return normalized or None


def resolve_component_residency_mode(
    component_name: str, assignments: Mapping[str, str] | None
) -> str | None:
    """为单个组件解析最终模式：精确名 > 分组 > all。"""
    if not assignments:
        return None
    exact = assignments.get(component_name)
```

```
    if exact is not None:
        return exact
    for group in COMPONENT_RESIDENCY_GROUP_PRECEDENCE:
        if group in assignments and component_residency_selector_matches(
            component_name, group
        ):
            return assignments[group]
    return assignments.get("all")
```

python/sglang/multimodal_gen/runtime/managers/memory_managers/component_residency_strategies.py

新增运行时驻留策略实现，直接映射三种 CLI 模式，是生命周期执行的核心。

```
# component_residency_strategies.py — 运行时组件驻留策略
# 每个策略只负责一个 component 在 use 前后的设备放置生命周期，
# 不再有自己的 enter/exit 抽象，直接对应 CLI 的三种模式。

class ComponentOffloadStrategy(ComponentResidencyStrategy):
    """在每次使用前后把整个组件在 CPU 与设备之间搬移。"""

    def __init__(self) -> None:
        # 异步 H2D 预取用的 side stream 与 ready event
        self._prefetch_stream = None
        self._ready_events = {}

    def prepare_for_use(self, module, use, state) -> None:
        _module_to_local_device(module, dtype=use.target_dtype)

    def wait_for_use(self, module, use, state) -> None:
        # 等待异步预取完成，仅 CUDA 需要显式等待事件
        event = self._ready_events.get(use.component_name)
        if event is not None and current_platform.is_cuda():
            torch.get_device_module().current_stream().wait_event(event)

    def prefetch_for_use(self, module, use, state) -> bool:
        # 非 CUDA 或已就绪时同步待命；否则在 side stream 上发起 H2D 并记录事件
        if not current_platform.is_cuda():
            self.prepare_for_use(module, use, state)
            return True
        if _module_ready_on_local_device(module, dtype=use.target_dtype):
            return True
        if self._prefetch_stream is None:
            self._prefetch_stream = torch.get_device_module().Stream(
                device=get_local_torch_device()
            )
        with torch.get_device_module().stream(self._prefetch_stream):
            _module_to_local_device(module, dtype=use.target_dtype)
            event = torch.get_device_module().Event()
            event.record(self._prefetch_stream)
```

```
            self._ready_events[use.component_name] = event
            return True

        def finish_use(self, module, use, state) -> None:
            # 使用结束后等预取完成，再搬到 CPU 释放显存
            self.wait_for_use(module, use, state)
            tensor = _module_reference_tensor(module)
            if tensor is not None and tensor.device.type != "cpu":
                module.to("cpu", non_blocking=True)
            self._ready_events.pop(use.component_name, None)

        def finish_request(self, module, use, state, *, preferred: bool) -> None:
            # 如果是 warmup 且组件被标记为 preferred，则保留在设备上
            # 避免把冷启动 H2D 计进用户可见的请求延迟
            if preferred and state.batch_is_warmup:
                self.prepare_for_use(module, use, state)
                self.wait_for_use(module, use, state)
                return
            self.finish_use(module, use, state)


    class LayerwiseOffloadStrategy(ComponentResidencyStrategy):
        """只驱动已被 layerwise 化的组件，层级别生命周期由模块自身管理。"""

        def prepare_for_use(self, module, use, state) -> None:
            if isinstance(module, LayerwiseOffloadableModuleMixin):
                module.prepare_for_next_req()

        def finish_use(self, module, use, state) -> None:
            if not isinstance(module, LayerwiseOffloadableModuleMixin):
                return
            for manager in module.layerwise_offload_managers:
                manager.release_all()

        def finish_request(self, module, use, state, *, preferred: bool) -> None:
            if preferred:
                self.prepare_for_use(module, use, state)
            else:
                self.finish_use(module, use, state)
```

评论区精华

由于评审评论为 0，讨论主要来自作者迭代说明。核心争论与决策如下：

- legacy 标志与 canonical 混用边界：作者在 dc25c699 后列出兼容性契约——legacy 标志仍全部支持，canonical 选择器只覆盖匹配组件，未匹配组件保留 legacy/ 模型默认；显式 --dit-layerwise-offload false 意味着 resident DiT，除非另有显式 offload 控制。
- ModelOpt FP8 加载时序：H100 上 serialized FP8 重量化在 process_weights_after_loading 中需要 CUDA，而自动 layerwise 驻留把 transformer 过

早放 CPU。修复方向是让 WeightLoadPlan 收到正确的“设备后处理”需求，延迟 CPU 放置直到 postprocessing 结束。

- realtime 缓存命中 OOM：chunked realtime 流程中，已缓存的 prompt embeddings 命中后，stage-entry 仍把 10.58 GB 文本编码器搬到 GPU，导致显存不足。ComponentUse 新增 start_at_stage_entry 区分“stage 入口启动”与“调用点启动”，缓存命中不触碰编码器。
- Qwen 分层编码器实现方式：最初用继承 Transformers 模型并 __class__ 变性附加 mixin 的做法被视为过于意外，最终改为组合——Qwen2_5VLGenerationEncoder 是独立 SGLang 组件，只转发 forward/generate，移除特殊 from_pretrained 重写与运行时类型变换。
  - legacy 与 canonical 兼容性契约 (design): 采用‘canonical 只覆盖匹配项’的策略，legacy 与 canonical 可混合，但通过 explicit_residency_mode 统一折算。
  - ModelOpt FP8 加载后的设备后处理 (correctness): 通过 WeightLoadPlan 标记设备后处理需求，延迟 CPU 放置直到 postprocessing 完成。
  - realtime 缓存命中的显存 OOM (performance): 新增 start_at_stage_entry 字段，realtime 文本编码使用调用点启动，缓存命中不触碰编码器。
  - Qwen 分层编码器实现方式 (design): 改为组合，Qwen2_5VLGenerationEncoder 独立存在并显式转发 forward/generate，移除类型变换。
  - 测试 fixture 与 CI 失败修复 (testing): 修正 fixture 并补充真实默认值，完整 diffusion 单测通过 1273~1276 项，H100 E2E 一致性与性能均达标。

风险与影响

- 风险：主要风险集中在兼容性与动态行为：
  - server_args.py 的参数归一顺序与 auto_tune.py 的自动调优逻辑依赖 explicit_residency_mode 的完全正确折算，若某个 legacy 标志组合未覆盖，可能出现自动覆盖或冲突遗漏（例如 cpu_offload_components 与 legacy flag 原有互斥被放宽为允许混合，但 _adjust_cpu_offload_components 不再对未匹配组件回写 legacy 布尔，可能导致状态不一致）。
  - component_manager.py 中 begin_stage/end_stage 与 component_uses 的声明必须与 stage 实际使用完全一致，任何遗漏都会造成组件未被及时释放或错误驻留，影响显存。cosmos3 等 stage 重建时易引入回归。
  - component_loader.py 删除 _maybe_configure_layerwise_after_startup_cpu_staging 后，所有 layerwise 配置依赖加载时 mark，若某个 loader 未正确调用 should_start_component_on_cpu 或未配置 layerwise mixin，将导致 build_component_residency_strategy 抛出 ComponentResidencyError（快速失败，优于静默错误）。
  - 动态组件替换（如 forget_module）需要缓存、prefetch 事件与 NVTX 钩子同步清理，遗漏可能造成悬挂引用。
  - 文档中的 docs/diffusion 遗留目录未更新，新旧文档并存可能造成用户混淆。
  - 影响：影响范围集中于 SGLang-Diffusion 子系统的配置面与运行时生命周期，对核心 LLM serving 路径无影响。对用户：新 CLI 提供更精确的驻留控制，但需理解精确组件

名与分组优先级的语义；所有历史脚本仍可运行，但 --cpu-offload-components 与 legacy 布尔不再互斥，行为有细微变化（canonical 覆盖后 legacy 不恢复）。对系统：加载路径不再在 loader 内部做 layerwise 修补，减少了隐式行为；阶段生命周期统一走 ComponentResidencyManager，便于未来扩展 budget-aware 预取。对团队：新增 70 个文件 ~2600 行，测试 20+ 个用例覆盖矩阵，但维护者需要熟悉三套（legacy、canonical、混合）语义。

- 风险标记：大面积重构，配置兼容性矩阵复杂，多文件耦合，缺少精度基准对比，动态组件生命周期风险

关联脉络

- PR #34980 [Diffusion] Native Hunyuan3D Paint and Delight models: 同样涉及 diffusion 原生模型与 layerwise offload 组件，且本 PR 修改了 hunyuan3d/shape.py 与 layerwise_offload_components.py，二者在同一功能线上演进。
- PR #34951 [Diffusion] Native ERNIE prompt enhancer: 同为 diffusion 组件原生化与 offload 分层支持，共享组件加载与驻留策略基础。
- PR #34825 [diffusion] Bound overlong weight lock filenames: 同属 diffusion 运行时健壮性修复，与本 PR 的 loader 与生命周期改动相关。