

PR #39329 完整报告

sgl-project/sglang

Add external multimodal processors to the Rust frontend

合并时间: 2026-09-16 06:38

原文链接: <http://prhub.com.cn/sgl-project/sglang/pull/39329>

执行摘要

本 PR 为 SGLang 的 Rust 前端 (`sglang-server`) 建立外部多模态处理器扩展机制: 模型包可以自带 Rust 处理器与 Python 扩展, 同时复用前端的 HTTP 请求处理、MM worker 池和调度器集成。核心做法是把请求里 `multimodal_*` 字段作为不透明扩展透传到处理器、用 `MmProcessor` trait 统一内置与外部实现、并在结果进入调度器前做防御性校验。内置 Qwen-VL 路径行为保持, 改动集中在扩展边界与启动接线, 但作者声明未做 GPU 实测, 属于影响面较大但方向清晰的架构扩展。

功能与动机

PR body 的动机很明确:

Processor-specific fields and encoded results need a typed extension boundary so additional model packages can integrate without adding their schemas to the shared frontend.

也就是说, 共享前端不应为每个模型家族维护专属 schema。本 PR 通过三个机制实现: 请求层用 `multimodal_*` 命名空间保留模型自有字段; 处理层用 `MmProcessor` trait 提供统一执行契约; 结果层用 `ExternalMmEncodedEntry` 承载外部结果并做校验。同时允许模型包提供自己的 `_server` 扩展模块, 替换 Python 侧启动接线。该 PR 取代 #38696, 属于同一功能线的第二版落地。

实现拆解

- 请求字段透传 (`rust/sglang-server/src/message/request.rs`) - 新增 `ProcessorExtensions` (`BTreeMap<String, rmpv::Value>`), 以 `#[serde(flatten)]` 捕获 `multimodal_*` 前缀字段, 其余未知字段仍按 Python 行为忽略。 - `split_extension_columns` 按 batch 形状拆分: 标量广播到每个请求, 数组按位置分配, 空数组广播为空列表, 长度不匹配返回 400。 - 扩展字段随 `MmData` / `MmWorkItem` 一路传到 MM worker, 由处理器通过 `ProcessorExtensions::deserialize::<T>()` 解码成自己的 schema。
- 处理器抽象 (`rust/sglang-server/src/multi_modality/worker.rs`, `utils/runtime.rs`) - 新增 `pub trait MmProcessor: Send + Sync`, 签名 `process(work, tokenizer) -> MmProcessOutput`, 要求实现不得保留 request-scoped 的 Python 对象。 - 内置 Qwen-VL 管道封装为 `QwenMmProcessor`, `MmContext` 从持有 `family` 改为持有 `Arc<dyn MmProcessor>`, 新增 `with_processor` 注入点。 - 统一入口 `process()` 对

External 结果先执行 `validate()` (feature shape 乘积等于数据长度、token offsets 落在展开 `input_ids` 内), 再统一应用 caller hashes 覆写, 最后才 park 进 `MmResultStore`。

- `Runtime::start_mm_workers_with_processor` 为外部处理器提供等价启动入口, `spawn_mm_pool` 抽出公共线程池逻辑。

3. 结果边界类型化 (`lib.rs`、`result_store.rs`、`rust/sglang-mm/src/pipeline.rs`) - `MmEncodedResult` 新增 `external_items / external_token_ids`; `ExternalMmItemResult` 提供 `feature`、`shape`、`hash`、`offsets`、`model_specific_data`。- `MmFeatureArray` 枚举覆盖 F32 / I64 / BF16 三种数组, 其中 BF16 以裸 u16 位模式暴露, `feature_is_bf16()` 供 Python 侧判断后 `reinterpret`, 避免一次拷贝。- `result_store.rs` 引入 `MmModality`、`MmTokenIds`、`ExternalMmItem` 等类型; `pipeline.rs` 为 `TensorData` 增加 BF16 变体与 `len / is_empty`。

4. Python 启动钩子 (`rust_server/server.py`、`config.py`、`scheduler.py`) - `RustServer.launch` 被拆分为 `_load_extension`、`_start_multimodal`、`_partition_cores`、`_server_core_budget`、`_wrap_mm_result`、`_build_mm_spec` 等可覆写方法, 模型包通过继承 `RustServer` 替换扩展加载与 worker 启动。- `_build_server_args` 接受 `extension` 参数, 避免重复加载扩展模块; `_partition_cores` 支持注入 `server_core_budget` 回调, 模型包可自定义 CPU 预算。- `scheduler.py` 暴露 `get_rust_server_class`, 让包级替换真正生效; `drain` 改用 `_multimodal_enabled` 状态而非 `mm_spec is not None`。

5. 测试配套 - 新增 `test_rust_server_extension.py`: mock 自定义扩展模块与 `_start_multimodal`, 断言 `_build_server_args` 收到 `extension`、`Server` 构造参数正确、worker 启动一次、实例状态完整。- `worker.rs` 单元测试覆盖外部结果入库 (含 caller hashes 覆写) 与畸形结果拒绝; `payload.rs` 覆盖跨 image/video/audio 的共享字节预算、顺序与 `prefetched allocation` 指针保留、扩展字段拒绝路径。

rust/sglang-server/src/multi_modality/worker.rs

核心改动: 引入 `MmProcessor` trait, 内置 Qwen 管道封装为 `QwenMmProcessor`, `MmContext` 改为持有 `Arc<dyn MmProcessor>`; 统一 `process()` 对外部结果做 park 前校验并应用 caller hashes。

/// 多模态处理器抽象: 内置与外部实现共用固定 Rust worker 池。

/// 实现必须满足 `Send + Sync`, 且不得保留 request-scoped 的 Python 对象。

```
pub trait MmProcessor: Send + Sync {
    fn process(
        &self,
        work: MmWorkItem,
        tokenizer: Option<&dyn TextTokenizer>,
    ) -> Result<MmProcessOutput, String>;
}
```

/// 内置 Qwen-VL 处理器: 继续走 `sglang-mm` 的 family pipeline。

```
struct QwenMmProcessor {
    family: Box<dyn sglang_mm::pipeline::MmFamilyProcessor>,
    feature_shm: bool,
}
```

```
impl QwenMmProcessor {
```

```

fn new(spec: MmSpec) -> Result<Self, String> {
    Ok(Self {
        family: sglang_mm::registry::build_pipeline(spec.pipeline)?,
        feature_shm: spec.feature_shm,
    })
}

impl MmProcessor for QwenMmProcessor {
    fn process(
        &self,
        work: MmWorkItem,
        tokenizer: Option<&dyn TextTokenizer>,
    ) -> Result<MmProcessOutput, String> {
        // 解析请求媒体字段并交给 `sglang-mm` 驱动; Qwen 家族处理路径与重构前一致
        let input = super::payload::to_mm_input(work)?;
        let output = sglang_mm::driver::process(self.family.as_ref(), input, |text| {
            let tokenizer = tokenizer.ok_or_else(|| {
                "skip_tokenizer_init is set: multimodal text prompts require input_ids".to_string()
            });
            tokenizer.encode(text).map_err(|error| error.to_string())
        });
        let drain = sglang_mm::qwen_vl::pack_output(output)?;
        // 按 `feature_shm` 决定原地 park 还是走 POSIX 共享内存跨 TP 广播
        let features = if self.feature_shm {
            park_features_in_shm(&drain.features, &drain.grids)
        } else {
            FeatureStore::Inline(drain.features)
        };
        Ok(MmProcessOutput {
            input_ids: drain.input_ids,
            result: MmEncodedEntry::Qwen(QwenMmEncodedEntry {
                features,
                grids: drain.grids,
                hashes: drain.hashes,
                offsets: drain.offsets,
                mrope: drain.mrope,
                mrope_delta: drain.mrope_delta,
            }),
        })
    }
}

```

/// 单请求处理入口: 先交给 `MmProcessor`, 再统一做校验与 hashes 覆写。

```

fn process(ctx: &MmContext, rid: &Rid, mut work: MmWorkItem) -> Result<Vec<i32>, String> {
    let caller_hashes = std::mem::take(&mut work.mm_hashes);
    let mut output = ctx.processor.process(work, ctx.tokenizer.as_deref())?;
    match &mut output.result {
        MmEncodedEntry::Qwen(entry) => {

```

```

        // caller hashes 覆写计算值，保证外部 router 的键与 prefix cache 对齐
        apply_caller_hashes(entry.hashes.iter_mut(), &caller_hashes);
    }
    MmEncodedEntry::External(entry) => {
        // 外部结果在 park 之前校验，避免把损坏数据送进调度器
        entry.validate(output.input_ids.len())?;
        apply_caller_hashes(
            entry.items.iter_mut().map(|item| &mut item.hash),
            &caller_hashes,
        );
    }
}
ctx.results.park(rid.as_str().to_owned(), output.result);
Ok(output.input_ids)
}

```

rust/sglang-server/src/message/request.rs

请求层数据契约：ProcessorExtensions 以 `#[serde(flatten)]` 捕获 `multimodal_*` 字段并用 `rmvp::Value` 保留，`split_extension_columns` 负责 batch fan-out；这是外部处理器获得自有 schema 的入口。

```

/// 顶层属于所选多模态处理器的字段命名空间。
/// 其余 `GenerateBody` 未知字段保持 Python 的"接受但忽略"行为。
const PROCESSOR_EXTENSION_PREFIX: &str = "multimodal_";

/// 模型自有的请求字段。共享服务端保留并随 batch fan-out 其 MessagePack
/// 值表示；由所选处理器反序列化成自己的具体 schema。
#[derive(Debug, Clone, Default, Deserialize)]
#[serde(transparent)]
pub struct ProcessorExtensions(BTreeMap<String, rmvp::Value>);

impl ProcessorExtensions {
    /// 直接把模型无关的值树反序列化进所选处理器的 schema。
    /// 这里不编解码 MessagePack 字节。
    pub fn deserialize<T: DeserializeOwned>(self) -> Result<T, String> {
        let fields = self
            .0
            .into_iter()
            .map(|(name, value)| (rmvp::Value::from(name), value))
            .collect();
        rmvp::ext::from_value(rmvp::Value::Map(fields))
            .map_err(|error| format!("invalid processor extensions: {error}"))
    }

    pub(crate) fn is_empty(&self) -> bool {
        self.0.is_empty()
    }
}

```

```

/// 按 batch 形状拆分扩展列：标量值广播到每个请求，数组值按位置分配。
fn split_extension_columns(
    fields: ProcessorExtensions,
    n: usize,
    is_batch: bool,
) -> Result<Vec<ProcessorExtensions>, Error> {
    let mut requests = vec![ProcessorExtensions::default(); n];
    for (name, value) in fields.0 {
        if !name.starts_with(PROCESSOR_EXTENSION_PREFIX) || value.is_nil() {
            continue;
        }
        if !is_batch {
            requests[0].0.insert(name, value);
            continue;
        }
        let rmpv::Value::Array(values) = value else {
            return Err(Error::Validation(format!(
                "{name} must be a list for batch processing"
            )));
        };
        if values.is_empty() {
            // 空数组广播为空列表，避免 batch 中个别项缺字段
            for request in &mut requests {
                request.0.insert(name.clone(), rmpv::Value::Array(Vec::new()));
            }
            continue;
        }
        if values.len() != n {
            return Err(Error::Validation(format!(
                "{name} list length {} does not match batch size {}",
                values.len(), n
            )));
        }
        for (request, value) in requests.iter_mut().zip(values) {
            request.0.insert(name.clone(), value);
        }
    }
    Ok(requests)
}

```

评论区精华

本 PR 没有任何 review 评论 ([review_comments](#) 为空)，issue 评论仅为 CI 指令 ([/tag-and-rerun-ci](#)、[/rerun-group rust-server](#))。值得注意的设计取舍来自 PR body 自述而非评审交锋：作者明确承认 "GPU model execution was not run locally"，以 CPU 验证覆盖处理器分派、请求字段、结果校验、媒体预算、内置结果包装与启动拓扑；同时强调 "Unit tests cover preservation of prefetched allocations and the existing zero-copy Python wrapping contract"，说明零拷贝与预取分配保序是本次改动刻意守护的契约。

风险与影响

- 核心路径重构风险: worker.rs 所有多模态请求 (含内置 Qwen) 都改为经 Arc<dyn MmProcessor> 动态分发, trait 实现缺陷会影响既有路径; 该文件本身无 GPU 回归验证。
- 行为变更风险: multimodal_* 字段从 "未知字段被忽略" 变为 "保留并传给处理器"; to_mm_input 对不支持扩展的处理器直接拒绝请求 (返回 "unsupported generate extensions"), 客户端可能从静默忽略变成 400。
- BF16 零拷贝契约: MmFeatureArray::Bf16 以裸 u16 位模式暴露, Python 集成方必须依据 feature_is_bf16 做 reinterpret, 否则产生静默数值错误。
- API 稳定性: lib.rs 中 MmEncodedResult、Server、RequestBatch 等从私有转公开, _server 扩展模块成为对外契约, 后续变更成本高。
- 影响范围: 模型开发者获得标准化接入路径; 内置 Qwen 用户行为兼容但代码路径重构; 调度器核心逻辑未改动, 团队维护面集中在 Rust 前端与启动接线。

关联脉络

本 PR 在 body 中声明 "Supersedes #38696", 属于同一功能线的第二版实现 (第一版未合入)。它与仓库近期的 Rust 前端演进方向一致: CI 中已有 [vlm/test_rust_native_mm_e2e.py](#)、[vlm/test_rust_native_mm_mmmu.py](#) 等 Rust 原生多模态测试, 说明团队正逐步把多模态处理从前端 API 层下沉到 Rust worker 池; 本 PR 在此基础上打开外部处理器接入通道, 为后续新模型家族 (如非 Qwen-VL 系多模态模型) 提供标准扩展点。