

PR #50289 完整报告

vllm-project/vllm

[Rust Frontend] Add standalone Rust renderer

合并时间: 2026-08-06 19:23

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/50289>

执行摘要

本 PR 实现 RFC#49047 的 Stage 2: 新增 `vllm-rs render` 引擎无关渲染服务, 复用 Rust 前端已有的 chat/text 请求预处理与 OpenAI 校验路径, 暴露 `/health`、`/ping`、`/v1/models`、`/v1/chat/completions/render`、`/v1/completions/render` 端点。实现上把 parser 选择下沉到 `ChatRequestProcessor`, 渲染输出直接对齐 `/inference/v1/generate` 请求。功能上线后, 路由中间层 (如 `llm-d-router`) 可独立横向扩展渲染容量; 但认证缺失与默认监听地址问题需在后续合入中尽快修复。

功能与动机

RFC#49047 提出: Python 版 `vllm launch render` 已支持无引擎预处理, 但在分布式部署中渲染可能成为瓶颈 (`llm-d-router#2025`)。早期原型在 M1 Max 上用 Codex SWE-bench Pro 长文本 (中位 72K token) 验证, Rust 原型单并发 63 req/s vs Python 6.25 req/s (10.1x), 8 并发 14x; Docker 镜像 92.9 MB vs 3.5 GB。Stage 1 (#49045) 已提取 engine-free 的 `ChatRequestProcessor` / `TextRequestProcessor`, 本 PR 在此基础上搭建独立服务。

实现拆解

1. CLI 入口 (`rust/src/cmd/src/cli.rs`、`main.rs`) : 新增 `Command::Render` 与 `RenderArgs`, 提供模型名、host/port、served-model-name、tool/reasoning parser、tokenizer-mode (映射 `RendererSelection`)、chat-template、max-model-len、max-logprobs 参数; `into_config()` 生成 `RenderConfig`。
2. 服务核心 (`rust/src/server/src/render.rs`) : `RenderConfig::validate()` 启动前校验 parser 与 max_logprobs; `build_state()` 以 `language_model_only` 加载后端并构建 `RenderState`; `serve_render()` 用 `tokio::select!` 处理启动期 shutdown, 绑定 TCP 后启动 `axum` 并支持优雅关闭。
3. 路由层 (`rust/src/server/src/routes/render.rs`) : `build_router()` 挂载端点; `render_chat` / `render_completion` 复用 `lower_chat_request` / `lower_completion_request`, 经 `ChatRequestProcessor::prepare` 得到 `TextRequest`, 再通过 `lower_render_request()` 降级为公开 `GenerateRequest` 并复用 `validate_generate_request` 校验。
4. 处理器重构 (`rust/src/chat/src/lib.rs`) : `parser` 字段从 `ChatLlm` 移入 `ChatRequestProcessor`, 新增 `with_parser_selections()`; `prepare()` 不再接收外部选项, 内部用处理器自持的 `parser` 选择构造输出处理器。`openai` 路由同步适配新签名。

5. 测试配套 (`routes/tests.rs`、`cmd/cli/tests.rs`)：新增 5 个路由测试 (`request_id` 来源、`parser` 透传、渲染结果可提交 `generate`) 与 CLI 配置转换测试；测试基于 `FakeChatBackend`，未覆盖真实模型端到端。

rust/src/server/src/routes/render.rs

新增渲染服务全部 HTTP 端点，是独立服务的对外入口；`render_chat/render_completion` 复用 OpenAI 校验与 `lower` 路径，`lower_render_request` 把结果降级为可提交给 `/inference/v1/generate` 的公开请求。

// 渲染服务路由：只暴露预处理相关端点，不包含任何引擎提交路由

```
pub(crate) fn build_router(state: Arc<RenderState>) -> Router {
```

```
    Router::new()
        .route("/health", get(health))
        .route("/ping", get(health).post(health))
        .route("/v1/models", get(list_models))
        .route("/v1/chat/completions/render", post(render_chat))
        .route("/v1/completions/render", post(render_completion))
        .with_state(state)
        .layer(DefaultBodyLimit::max(DEFAULT_REQUEST_BODY_LIMIT_BYTES))
}
```

/// 将渲染结果降级为公开的 `GenerateRequest`，可被 `/inference/v1/generate` 直接消费

```
fn lower_render_request(
    state: &RenderState,
    text_request: TextRequest,
    model: String,
    stream: bool,
    stream_options: Option<StreamOptions>,
) -> Result<GenerateRequest, ApiError> {
    // 复用引擎路径的文本预处理：tokenize、logprobs 限制、上下文长度校验等
    let prepared = state
        .text
        .prepare(text_request)
        .map_err(|error| text_submit_error("failed to prepare render request", error))?;
    let request = GenerateRequest {
        request_id: Some(prepared.text_request.request_id),
        model: Some(model),
        token_ids: prepared.generate_request.prompt_token_ids,
        sampling_params: prepared.text_request.sampling_params,
        stream,
        stream_options,
        cache_salt: prepared.text_request.cache_salt,
        priority: prepared.text_request.priority,
        kv_transfer_params: None,
        ec_transfer_params: None,
        other: Default::default(),
    };
    // 复用 generate 路由的同款兼容性校验，保证输出可被下游接受
```

```

        validate_generate_request(&request, &state.served_model_names)?;
        Ok(request)
    }

    async fn render_chat(
        State(state): State<Arc<RenderState>>,
        headers: HeaderMap,
        ValidatedJson(body): ValidatedJson<ChatCompletionRequest>,
    ) -> Result<Json<GenerateRequest>, ApiError> {
        let model = body.model.clone();
        let request_context = resolve_request_context(&headers, body.request_id.as_deref());
        // 走与线上 chat 完全相同的 lower + prepare 管线
        let chat_request = lower_chat_request(body, &model_resolution(&state), request_context)?;
        let (text_request, _) = state
            .chat
            .prepare(chat_request)
            .await
            .map_err(|error| ApiError::invalid_request(error.to_report_string(), None))?;
        Ok(Json(lower_render_request(
            &state, text_request, model, stream, stream_options,
        )))
    }
}

```

rust/src/server/src/render.rs

独立渲染服务的核心逻辑：配置校验、后端加载、监听绑定与优雅关闭；定义了 **RenderConfig** 与 **RenderState**，是服务生命周期管理的关键。

```

/// 引擎无关的文本渲染服务入口。
///
/// 与引擎连接型 `AppState` 相互独立，仅持有 `TextRequestProcessor` 与
/// `ChatRequestProcessor`，不建立 IPC 也不提交生成任务。
pub async fn serve_render(config: RenderConfig, shutdown: CancellationToken) -> Result<()> {
    // 启动前先验证配置，避免带着非法 parser 或 max_logprobs 去初始化后端
    config.validate().context("invalid render server configuration")?;

    // 加载 tokenizer / chat-template 等后端；期间响应关闭信号可提前退出，
    // 避免在 shutdown 竞态下仍拉起监听端口
    let state = tokio::select! {
        result = build_state(&config) => result?,
        _ = shutdown.cancelled() => return Ok(()),
    };

    let listener_mode = HttpListenerMode::BindTcp {
        host: config.host.clone(),
        port: config.port,
    };

    let listener = Listener::bind(&listener_mode)
        .await
        .with_context(|| format!("failed to bind {}:{}", config.host, config.port))?;

```

```

let address = listener.local_addr_display()?;
info!(
    %address,
    model = %config.model,
    "starting engine-free Rust render server"
);
// 与引擎服务共用 axum 优雅关闭机制
axum::serve(listener, crate::routes::render::build_router(state))
    .with_graceful_shutdown(shutdown.cancelled_owned())
    .await
    .context("render server failed")
}

```

rust/src/chat/src/lib.rs

核心处理器重构：parser 选择从 `ChatLlm` 下沉到 `ChatRequestProcessor`，新增 `with_parser_selections`，`prepare()` 不再接收外部选项，是渲染服务与推理路径共用预处理的基石。

```

/// 将 parser 选择从 `ChatLlm` 下沉到处理器内部，渲染服务与推理路径共用
pub fn with_parser_selections(
    mut self,
    tool_call_parser: ParserSelection,
    reasoning_parser: ParserSelection,
) -> Self {
    self.tool_call_parser = tool_call_parser;
    self.reasoning_parser = reasoning_parser;
    self
}

/// 准备请求但不下发引擎
pub async fn prepare(
    &self,
    mut request: ChatRequest,
) -> Result<(TextRequest, DynChatOutputProcessor)> {
    request.validate()?;
    // 输出处理器使用处理器自身持有的 parser 选择,
    // 这样 render_only 模式也可获得与完整 ChatLlm 一致的解析行为
    let output_processor = self.backend.new_chat_output_processor(
        &mut request,
        NewChatOutputProcessorOptions {
            tool_call_parser: &self.tool_call_parser,
            reasoning_parser: &self.reasoning_parser,
        },
    )?;
    let text_request = self.prepare_text_request(request).await?;
    Ok((text_request, output_processor))
}

```

评论区精华

chatgpt-codex-connector (P2) : 对 Gemma 4 / Kimi K3 这类 unified-only parser 模型, 硬编码 `reasoning_parser` 为 `None` 会走 split-parser 路径, 导致 `/v1/chat/completions/render` 失败或返回不同结果。

BugenZhao: This sounds valid. We probably want to pass `Auto` here - or, does the Python renderer accepts parser selection from CLI option and should we follow that as well?

sagearc: Switching both parsers to auto exposed that reasoning-parser selection matched the publisher namespace instead of the model family, so I made it family-specific.

BugenZhao (关于返回类型) : The returned type here is actually the internal `vllm::GenerateRequest` ... Do we want the shape of this `GenerateRequest` to be aligned with the Python frontend?

sagearc: The endpoint now returns the public `/v1/generate` request, so text-only render output can be submitted directly. llm-d 当前调用 `/render` 获取 prompt token 做路由, 转发原始请求会重复渲染; 移除重复渲染正是本 PR 的价值点。

风险与影响

- 认证缺失与默认暴露: 渲染路由未接入 `authenticate_api_key`, `--api-key` / `--ssl-*` 会被静默丢弃, 且默认绑定 `0.0.0.0`。这是合入前未解决的高危问题, 建议立即跟进。
- parser 选择回归: unified parser 模型曾因硬编码出错, 修复后仍需回归验证所有模型家族 (QwQ、Gemma、Kimi 等) 的渲染与推理一致性。
- 输出信息有损: `lower_render_request` 将 kv 转移等参数置空, 仅适合文本路由场景; 多模态请求会被 `ChatRequestProcessor::render_only` 拒绝。
- 测试缺口: 路由测试基于 `FakeChatBackend`, 没有真实 tokenizer 的端到端覆盖, 也缺少长文本压力测试 (RFC 中的 182K token p99 场景)。

关联脉络

本 PR 是 RFC#49047 的 Stage 2, 前置依赖 Stage 1 (#49045, 已合入) 提取的 `ChatRequestProcessor` / `TextRequestProcessor`; 与 Rust 前端 `generate` 路由 (如 #52844 对 `n > 1` 的校验) 共享 `validate_generate_request`, 渲染输出天然对齐 `/inference/v1/generate`。后续 Stage 3 将补多模态渲染与响应后处理, 届时 `RenderState` 与 `lower_render_request` 需要扩展, 当前“文本 only + 窄 facade”的设计为后续演进留出了清晰边界。