

PR #50448 完整报告

vllm-project/vllm

[Rust Frontend] Deduplicate request preprocessing for `/tokenize`

合并时间: 2026-08-05 16:27

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/50448>

执行摘要

PR#50448 将 Rust 前端 `/tokenize` 端点此前重复实现的 prompt 处理逻辑, 统一收敛到 #49045 提取出的 chat/text request processor 上。核心做法是拆出 `prepare_text_request` / `prepare_for_tokenization`, 并新增不做 generation-lowering 的 `TextRequestProcessor::tokenize`, 使 `/tokenize` 返回的 token ID 与推理路径完全一致, 同时修复了带 tools 的 chat tokenize 请求失败和超长 prompt 被误拒两个行为回退。共修改 5 个文件, +170/-50, 补充 2 个回归测试。

功能与动机

PR body 明确说明问题: `/tokenize` currently repeats prompt handling already used by inference. This can make its token IDs drift from what generation receives, especially after chat rendering or multimodal prompt expansion. 即 tokenize 与推理各自维护一套渲染和编码逻辑, 导致 token ID 不一致。本 PR 将 #49045 提取的 chat/text request processor 暴露出来并复用到 `/tokenize`, 既消除重复代码, 也保证 tokenize 的语义与生成路径一致。

实现拆解

- chat 层拆分 (`rust/src/chat/src/lib.rs`): 原 `ChatRequestProcessor::prepare` 被拆为私有 `prepare_text_request` (渲染、reasoning、多模态、analyze、`TextRequestProcessor` 组装) 与公开的 `prepare` (先构造 output processor, 再委托前者); 新增公开方法 `prepare_for_tokenization` (只 validate + `prepare_text_request`), 删除原 `ChatLlm::tokenize_chat`, 新增 `request_processor()` getter。
- text 层新增纯 tokenize (`rust/src/text/src/lib.rs`): 抽出私有 `tokenize_prompt` 统一处理 `Prompt::Text` 与 `Prompt::TokenIds`; 原 `prepare` 复用 `tokenize_prompt`, 但保留 `lower_text_request`; 新增公开方法 `tokenize`, 只做 validate 与编码, 不做 lowering, 因此不会触发 `resolve_max_tokens` 的 `PromptTooLong` 检查。
- 路由层接入 (`rust/src/server/src/routes/tokenize.rs`、`types.rs`): `TokenizeCompletionRequest` 新增 `into_text_request`; `tokenize_completion` 改为 `text().request_processor().tokenize(...)`; `tokenize_chat` 改为 `request_processor().prepare_for_tokenization(...)` 后再 `tokenize(...)`。
- 测试配套 (`rust/src/server/src/routes/tests.rs`): 新增 `tokenize_chat_with_tools_does_not_require_output_parser` 和 `tokenize_allows_prompts_at_or_above_max_model_len`, 分别锁定 codex review 指出的

两类回归。

rust/src/chat/src/lib.rs

核心重构文件：拆出 prepare_text_request 与 prepare_for_tokenization，删除 ChatLlm::tokenize_chat，新增 request_processor()，使 chat tokenize 与推理共享同一渲染/finalize 链路。

```
// rust/src/chat/src/lib.rs
// 将原先 `prepare` 中与 output processor 无关的部分拆出，
// 使推理与 tokenize 共用同一套渲染 → finalize 逻辑。
async fn prepare_text_request(&self, request: ChatRequest) -> Result<TextRequest> {
    // 在渲染前打时间戳，让 render 与 tokenize 计入 TTFT/e2e。
    let arrival_time = vllm_ilm::current_unix_timestamp_secs();
    let rendered = self.backend.chat_renderer().render(&request)?;
    // 结构化输出存在时，把 chat 模板 kwargs 传给 reasoning parser。
    let reasoning_parser_kwargs = request
        .sampling_params
        .structured_outputs
        .is_some()
        .then(|| ReasoningParserKwargs {
            chat_template_kwargs: rendered.effective_template_kwargs.clone(),
        });
    let (prompt, mm_features) = self.finalize_rendered_prompt(&request, rendered).await?;
    Ok(TextRequest {
        request_id: request.request_id,
        prompt,
        mm_features,
        sampling_params: request.sampling_params,
        decode_options: request.decode_options,
        intermediate: request.intermediate,
        priority: request.priority,
        cache_salt: request.cache_salt,
        add_special_tokens: request.add_special_tokens,
        data_parallel_rank: request.data_parallel_rank,
        session_id: request.session_id,
        reasoning_parser_kwargs,
        lora_request: request.lora_request,
        arrival_time: Some(arrival_time),
    })
}

/// Tokenize 专用入口：只做渲染与校验，不构造 output processor，
/// 避免带 tools 的请求因 parser 不可用而失败。
pub async fn prepare_for_tokenization(&self, request: ChatRequest) -> Result<TextRequest> {
    request.validate()?;
    self.prepare_text_request(request).await
}

/// 推理入口：构造 output processor 后复用 prepare_text_request。
```

```

pub async fn prepare(
    &self,
    mut request: ChatRequest,
    options: NewChatOutputProcessorOptions<'_,>,
) -> Result<(TextRequest, DynChatOutputProcessor)> {
    request.validate()?;
    let output_processor = self.backend.new_chat_output_processor(&mut request, options)?;
    let text_request = self.prepare_text_request(request).await?;
    Ok((text_request, output_processor))
}

```

rust/src/text/src/lib.rs

新增 TextRequestProcessor::tokenize 与 tokenize_prompt 辅助函数，使 /tokenize 可以只编码不进行 generation lowering，避免超长 prompt 被拒绝。

```

// rust/src/text/src/lib.rs
// 统一处理文本与预分词两种 prompt 形态。
fn tokenize_prompt(
    tokenizer: &DynTokenizer,
    prompt: Prompt,
    add_special_tokens: bool,
) -> Result<Vec<u32>> {
    match prompt {
        Prompt::Text(text) => tokenizer.encode(&text, add_special_tokens).map_err(Into::into),
        // 预分词 prompt 是 completions 侧绕过 chat 渲染与 tokenizer 开销的主要逃生口。
        Prompt::TokenIds(token_ids) => Ok(token_ids),
    }
}

```

```

/// 仅做 tokenize，不做 generation-specific lowering,
/// 因此不会触发 resolve_max_tokens 的长度下限检查,
/// 允许 /tokenize 统计等于或超过 max_model_len 的 prompt。
pub fn tokenize(&self, request: TextRequest) -> Result<Vec<u32>> {
    request.validate()?;
    Self::tokenize_prompt(
        &self.backend.tokenizer(),
        request.prompt,
        request.add_special_tokens,
    )
}

```

```

/// 推理路径: tokenize 后还需做 sampling 限幅与请求 lowering。
pub fn prepare(&self, mut request: TextRequest) -> Result<PreparedTextRequest> {
    request.validate()?;
    if request.arrival_time.is_none() {
        request.arrival_time = Some(vllm_llm::current_unix_timestamp_secs());
    }
    let tokenizer = self.backend.tokenizer();
    let prompt_token_ids = Self::tokenize_prompt(

```

```

        &tokenizer,
        take(&mut request.prompt),
        request.add_special_tokens,
    )?;
    let sampling_hints = self.backend.sampling_hints()?;
    let sampling_limits = SamplingLimits {
        max_model_len: self.max_model_len,
        max_logprobs: self.max_logprobs,
        model_vocab_size: self.backend.model_vocab_size(),
        tokenizer_vocab_size: self.backend.tokenizer_vocab_size(),
    };
    lower_text_request(
        request,
        prompt_token_ids,
        sampling_hints,
        sampling_limits,
        tokenizer.as_ref(),
    )
}

```

rust/src/server/src/routes/tokenize.rs

路由入口改造：completion 与 chat 两种 /tokenize 请求都改为通过 request processor 链路处理，完成去重。

```

// rust/src/server/src/routes/tokenize.rs
// completion 形态：将请求体降为 TextRequest 后走 text processor 的 tokenize 方法。
fn tokenize_completion(
    state: &AppState,
    request_id: &str,
    req: TokenizeCompletionRequest,
) -> Result<(Vec<u32>, bool), ApiError> {
    check_model(state, req.model.as_deref())?;
    let return_token_strs = req.return_token_strs;
    let tokens = state
        .chat
        .text()
        .request_processor()
        .tokenize(req.into_text_request(request_id.to_string()))
        .map_err(|e| server_error!("tokenize failed: {}", e.to_report_string()))?;
    Ok((tokens, return_token_strs))
}

// chat 形态：先做渲染与多模态 finalize（不构造 output processor），
// 再交给 text processor 编码，token ID 与推理路径完全一致。
async fn tokenize_chat(
    state: &AppState,
    request_id: &str,
    req: TokenizeChatRequest,
) -> Result<(Vec<u32>, bool), ApiError> {

```

```

check_model(state, req.model.as_deref())?;
let return_token_strs = req.return_token_strs;
// `continue_final_message` / `add_generation_prompt` 互斥在
// `normalize_generation_prompt_mode` (into_chat_request 内) 强制。
let text_request = state
    .chat
    .request_processor()
    .prepare_for_tokenization(req.into_chat_request(request_id.to_string()))?
    .await
    .map_err(|e| server_error!("tokenize failed: {}", e.to_report_string()))?;
let tokens = state
    .chat
    .text()
    .request_processor()
    .tokenize(text_request)
    .map_err(|e| server_error!("tokenize failed: {}", e.to_report_string()))?;
Ok((tokens, return_token_strs))
}

```

评论区精华

codex P1: When a chat-form `/tokenize` request supplies a nonempty `tools` list for a model without an auto-registered tool or unified parser, this call now creates a `DefaultChatOutputProcessor` even though the processor is discarded... Tokenization should reuse rendering and prompt finalization without requiring an output parser.

codex P1: When an encoded completion or rendered chat prompt has `prompt_len >= max_model_len`, routing it through `TextRequestProcessor::prepare` invokes `resolve_max_tokens`, which rejects it as `PromptTooLong`... the previous Rust code returned the IDs directly.

BugenZhao: I feel that it would be better if we can bypass the construction & validation of output processor for tokenization use... probably by making `NewChatOutputProcessorOptions` wrapped inside an `Option`?

BugenZhao: Shall we have a method to do this, similar to `into_chat_request`?

最终通过新增 `prepare_for_tokenization` 与 `into_text_request` 两个方法解决，并补充两个对应测试锁定行为；唯一未解决的是删除 `match` 分支注释的 P3 风格问题。

风险与影响

- 共享路径回归面：`prepare_text_request` 成为推理与 `tokenize` 的公共路径，后续修改该函数会同时影响两端；新增测试覆盖了 `tools` 与超长 `prompt` 两个关键场景，但自定义 `chat_template` 等边界仍需关注。
- 行为放宽：`/tokenize` 不再拒绝超长 `prompt`，与 Python 前端行为对齐，但若调用方误用 `prepare` 仍会触发长度检查，需在代码维护中保持路由层使用 `prepare_for_tokenization` / `tokenize`。

- 风格遗留：rust/src/server/src/routes/tokenize.rs 中 match 分支注释被删除，违反 rust/AGENTS.md 要求，虽不影响功能，但可能影响后续 CI 或 review 体验。

影响面为 Rust 前端 `/tokenize` 端点的行为一致性与稳定性，同时为 #49047 后续 renderer 阶段提供了更干净的处理器复用基础。

关联脉络

本 PR 是 #49045 (Extract request preparation from the inference path) 的后续落地，直接复用其提取出的 `ChatRequestProcessor` 与 `TextRequestProcessor`；同时也是 #49047 分期计划的 Stage 之一，为未来 renderer 等组件复用同一套预处理逻辑奠定基础。近期历史 PR 中 #50275、#50321 等均涉及 v1 前端核心路径的调度与预处理，本 PR 与它们共同推动 Rust 前端请求处理链路的一致化重构。