PR #44222 完整报告

vllm-project/vllm

[Rust Frontend] Add /tokenize and /detokenize endpoints

合并时间：2026-06-09 20:11

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/44222

执行摘要

本 PR 为 vLLM 的 Rust 前端添加了 /tokenize 和 /detokenize 两个 HTTP 端点，与 Python OpenAI 服务器的根路径端点行为一致。这两个端点完全在进程内完成 token 编码 / 解码，不涉及推理引擎，为客户端提供了轻量的 token 计数与调试能力。实现通过 untagged 枚举区分 completion 和 chat 两种请求形式，复用了 chat completions 端点的消息校验和模板渲染逻辑，确保 token 结果与真实生成一致。代码质量高，测试充分，但存在已知的安全风险（SSTI/SSRF）需后续统一修复。

功能与动机

该 PR 的目标是填补 Rust 前端功能对等路线图（#44280）中列出的关键空白。tokenize 和 detokenize 端点被广泛用于客户端侧 token 计数、前缀缓存调试和多模态输入预处理，是生产级推理服务器的标准功能。此 PR 实现了这些端点的 Rust 版本，使 Rust 前端用户无需回退到 Python 服务器即可使用相同的根路径 API。

实现拆解

1. 请求模型与序列化：在 rust/src/server/src/routes/tokenize/types.rs 中定义 TokenizeRequest（untagged 枚举，根据 messages 或 prompt 字段自动分发）、DetokenizeRequest 以及对应的响应结构体。TokenizeChatRequest 通过 into_chat_request() 方法转换为内部 ChatRequest，复用已有的 convert_message 和 normalize_generation_prompt_mode 函数，确保模板渲染和 generation prompt 逻辑与 chat completions 端点完全一致。
2. 处理函数：在 rust/src/server/src/routes/tokenize.rs 实现 tokenize() 和 detokenize() 异步函数。tokenize 内部根据请求变体分别调用 tokenize_completion（直接编码 prompt 字符串）或 tokenize_chat（先渲染聊天模板再编码）。tokenize_chat 委托给 ChatLlm::tokenize_chat 方法，该方法在 rust/src/chat/src/lib.rs 中新增，执行完整的渲染 → finalize_rendered_prompt → encode 管线，但不向引擎提交推理请求。
3. 共享校验与重构：将 validate_messages 函数从 chat_completions/types.rs 抽取到 openai/utils/types.rs 作为 pub(crate) 函数，使 tokenize 的 chat 变体可以共用相同的消息非空和内容校验，确保行为一致。移除 chat_completions/types.rs 中的私有多余实现（减少 29 行重复代码）。
4. 路由注册：初始路由项 (/tokenize 和 /detokenize) 放在 openai 模块下，经 review 后由维护者 BugenZhao 直接提交 commit 移出到顶层 routes/tokenize 模块，与 Python 后端文件布局一致，并归类到“vLLM 特定端点”注释下。

5. 测试配套：在 tests.rs 中添加 400+ 行集成测试，覆盖 completion/detokenize 往返、add_special_tokens 对 token ID 的影响、return_token_strs 的按需返回、count 和 max_model_len 字段，以及 chat 形式下 generation prompt 增加 token 计数的验证。同时扩展 FakeChatTokenizer 以支持模拟 BOS token 插入和有限字符级的 id_to_token 映射，使测试更加真实。

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

定义 tokenize/detokenize 的请求 / 响应类型及 untagged 枚举区分，是端点的数据契约。

```
// src/server/src/routes/tokenize/types.rs

use std::collections::HashMap;
use serde::{Deserialize, Serialize};
use validator::{Validate, ValidationErrors};
use vllm_chat::{ChatOptions, ChatRequest, SamplingParams, ChatToolChoice};
use crate::error::ApiError;
use crate::routes::openai::chat_completions::convert::{convert_message, convert_tools,
normalize_generation_prompt_mode};
use crate::routes::openai::utils::types::{ChatMessage, Tool, Normalizable, default_true, validate_
messages};

/// `POST /tokenize` body，通过 untagged 枚举区分 completion 和 chat 两种请求。
#[derive(Debug, Clone, Deserialize)]
#[serde(untagged)]
pub enum TokenizeRequest {
    Chat(TokenizeChatRequest),
    Completion(TokenizeCompletionRequest),
}

#[derive(Debug, Clone, Deserialize)]
pub struct TokenizeCompletionRequest {
    pub model: Option<String>,
    pub prompt: String,
    #[serde(default = "default_true")]
    pub add_special_tokens: bool, // 默认为 true（与 Python 一致）
    #[serde(default)]
    pub return_token_strs: bool,
}

#[derive(Debug, Clone, Deserialize, Validate)]
pub struct TokenizeChatRequest {
    pub model: Option<String>,
    #[validate(custom(function = "validate_messages"))] // 共用 chat completions 的消息校验
    pub messages: Vec<ChatMessage>,
    #[serde(default = "default_true")]
    pub add_generation_prompt: bool,
    #[serde(default)]
    pub continue_final_message: bool,
```

```
    #[serde(default)] // chat 形式默认不加特殊 token（由聊天模板添加）
    pub add_special_tokens: bool,
    #[serde(default)]
    pub return_token_strs: bool,
    #[serde(default)]
    pub chat_template: Option<String>,
    #[serde(default)]
    pub chat_template_kwargs: Option<HashMap<String, Value>>,
    #[serde(default)]
    pub tools: Option<Vec<Tool>>,
}

impl TokenizeChatRequest {
    /// 将 tokenize 请求转换为 ChatRequest，复用工件完成模板渲染。
    /// 只设置渲染相关字段，采样参数等保持默认。
    pub fn into_chat_request(self, request_id: String) -> Result<ChatRequest, ApiError> {
        let messages: Vec<_> = self.messages.into_iter().map(convert_message).try_collect()?;
        let generation_prompt_mode = normalize_generation_prompt_mode(
            Some(self.add_generation_prompt),
            self.continue_final_message,
            &messages,
        )?;

        Ok(ChatRequest {
            request_id,
            messages,
            sampling_params: SamplingParams::default(), // tokenize 不生成，默认即可
            chat_options: ChatOptions {
                generation_prompt_mode,
                chat_template: self.chat_template,
                template_kwargs: self.chat_template_kwargs.unwrap_or_default(),
                reasoning_effort: None,
            },
            tools: convert_tools(self.tools)?,
            tool_choice: ChatToolChoice::Auto,
            decode_options: TextDecodeOptions::default(),
            intermediate: false,
            priority: 0,
            documents: None,
            cache_salt: None,
            add_special_tokens: self.add_special_tokens,
            data_parallel_rank: None,
            lora_request: None,
        })
    }
}
```

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

实现核心处理逻辑，包括 completion/chat 分支、request-id 生成、模型校验和 detokenize。

```
// src/server/src/routes/tokenize.rs

use axum::Json;
use axum::extract::State;
use axum::http::HeaderMap;
use axum::response::{IntoResponse, Response};
use crate::error::{ApiError, server_error};
use crate::routes::openai::utils::validated_json::ValidatedJson;
use crate::routes::tokenize::types::*;
use crate::state::AppState;

/// 生成 request-id，格式为 tokenize-{base}，base 来自 X-Request-Id 或新 UUID。
pub fn tokenize_request_id(headers: &HeaderMap) -> String {
    let base = resolve_base_request_id(
        headers.get("X-Request-Id").and_then(|v| v.to_str().ok()),
        None,
    );
    format!("tokenize-{base}")
}

pub async fn tokenize(
    State(state): State<Arc<AppState>>,
    headers: HeaderMap,
    ValidatedJson(body): ValidatedJson<TokenizeRequest>,
) -> Response {
    let request_id = tokenize_request_id(&headers);
    let tokenizer = state.chat.text().tokenizer();
    let max_model_len = state.chat.engine_core_client().max_model_len();

    let result = match body {
        TokenizeRequest::Completion(req) => tokenize_completion(&state, &tokenizer, req),
        TokenizeRequest::Chat(req) => tokenize_chat(&state, &request_id, req).await,
    };

    match result {
        Ok((tokens, want_strs)) => {
            let token_strs = want_strs.then(|| token_strs(&tokenizer, &tokens));
            Json(TokenizeResponse { count: tokens.len(), max_model_len, tokens, token_strs }).into_
            response()
        },
        Err(error) => error.into_response(),
    }
}

/// completion 形式：直接编码 prompt 字符串。
pub fn tokenize_completion(
    state: &AppState,
```

```
    tokenizer: &DynTokenizer,
    req: TokenizeCompletionRequest,
) -> Result<(Vec<u32>, bool), ApiError> {
    check_model(state, req.model.as_deref())?;
    let tokens = tokenizer
        .encode(&req.prompt, req.add_special_tokens)
        .map_err(|e| server_error!("tokenize failed: {}", e.to_report_string()))?;
    Ok((tokens, req.return_token_strs))
}

/// chat 形式：先渲染模板，再编码。
pub 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;
    let tokens = state.chat
        .tokenize_chat(req.into_chat_request(request_id.to_string())?)
        .await
        .map_err(|e| server_error!("tokenize failed: {}", e.to_report_string()))?;
    Ok((tokens, return_token_strs))
}

pub async fn detokenize(
    State(state): State<Arc<AppState>>,
    ValidatedJson(body): ValidatedJson<DetokenizeRequest>,
) -> Response {
    if let Err(error) = check_model(&state, body.model.as_deref()) {
        return error.into_response();
    }
    let tokenizer = state.chat.text().tokenizer();
    match tokenizer.decode(&body.tokens, false /* skip_special_tokens == false 与 Python 一致 */)
    {
        Ok(prompt) => Json(DetokenizeResponse { prompt }).into_response(),
        Err(e) => server_error!("detokenize failed: {}", e.to_report_string()).into_response(),
    }
}
```

评论区精华

- 路由位置："should these be under openai module? afaik these aren't OpenAI APIs." — coder3101 的质疑促使维护者立即行动，最终 BugenZhao 直接提交 commit 完成迁移。
- 安全性讨论："The chat_template field accepts arbitrary user-supplied Jinja2 templates... the Python implementation rejects such requests by default." — depthfirst-app[bot] 指出了 SSTI 风险。作者回应："The same gap actually exists in the chat completions handler today"，认为应统一跟进。

- 空消息校验："For chat-shaped requests, this no-op validator lets messages: [] pass... producing a 500." — Codex 机器人（P2）的发现直接导致 validate_messages 共享函数的引入。

风险与影响

- SSTI 风险：chat_template 字段允许用户提供任意 Jinja2 模板，可能导致服务端模板注入。该风险与 chat completions 端点相同，但 /tokenize 端点提供了新的攻击面。影响决策：此 PR 不引入该风险，但暴露了已存在的安全缺口，需要后续统一修复（如添加 trust_request_chat_template 标志）。
- SSRF 风险：通过 image_url 内容可以触发外部资源加载，可能用于内网探测。同样需要后续实现 --allowed-media-domains 等防护。
- 功能对等遗漏：相比 Python 端点缺失 media_io_kwargs / mm_processor_kwargs 支持，多模态场景的 token 计数可能不准确。
- 正面影响：显著缩小了 Rust 前端与 Python 端点的功能差距，降低了用户依赖 Python 服务器的必要性，是 Rust 前端迈向生产可用性的重要一步。

关联脉络

本 PR 是 Rust 前端功能对等路线图（#44280）的具体实现之一，与 #44321（API key 认证）、#44729（结构化输出修复）等 PR 共同构成 Rust 前端的完善拼图。这些 PR 体现了团队将 Rust 前端推向生产级的战略方向，同时也揭示了需要系统性安全加固的需求（如 SSTI/SSRF 防护）。推荐关注后续的 trust_request_chat_template 和 allowed-media-domains 实现，以全面缓解安全风险。