PR #49992 完整报告

vllm-project/vllm

[Rust Frontend] Add ordinary-text tokenizer encoding

合并时间：2026-07-28 15:46

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/49992

PR 分析报告：Rust 前端添加普通文本 tokenizer 编码

执行摘要

此 PR 为 Rust 前端的 Tokenizer trait 新增 encode_ordinary 方法，使调用方能够获得绕过所有添加、特殊和控制 token 的纯文本编码。实现覆盖 tiktoken、HuggingFace、fastokens 和 Tekken 四种后端，并包含单元测试和集成测试。该功能是 segment-aware prompt renderers 的基础设施，为后续结构化 prompt 支持铺平道路。

功能与动机

Segment-awarepromptrenderers需要为可信的结构化标记和字面文本提供独立的编码路径。当前 encode 方法会识别并映射 added tokens 和 special tokens，这可能导致字面文本（如用户输入中的特殊标记被错误替换）。encode_ordinary 保持字面文本在基础 tokenizer 管道上，即使其拼写与添加 token 完全一致，也不触发特殊匹配。

实现拆解

1. Trait 定义(rust/src/tokenizer/src/lib.rs): 在 Tokenizer trait 中新增方法签名 fn encode_ordinary(&self, text: &str) -> Result<Vec<u32>>，并添加语义注释。
2. tiktoken 后端(rust/src/tokenizer/src/tiktoken.rs): 直接调用底层库（riptoken 或 tiktoken-rs）的 encode_ordinary 方法，这两个库原生支持跳过 registered added tokens。
3. HuggingFace tokenizers 后端(rust/src/tokenizer/src/hf.rs): 实现 encode_hf_ordinary 函数，通过使用空 AddedVocabulary 手动执行 extract_and_normalize、pre_tokenize、模型 tokenize 和 post_process，全程避免 added token 注入。同时实现 encode_fastokens_ordinary，针对 fastokens 后端构建不含 added tokens 的 PreTokenizedString，并检测 fused ByteLevel 场景以采用批量 tokenize 优化。
4. Tekken 后端(rust/src/tokenizer/src/tekken.rs): 简单委托给 self.inner.encode(text, false, false)，通过关闭 special tokens 参数实现绕过。
5. 测试与工具(rust/src/tokenizer/src/test_utils.rs 等): 添加集成测试验证 tiktoken 和 tekken 后端的正确性（例如输入含特殊 token 时，encode_ordinary 返回字节级原始编码）。同时在所有 mock 和测试后端（incremental.rs、inkling.rs 等）添加存根实现以保持编译。

rust/src/tokenizer/src/hf.rs

核心实现文件，新增 encode_hf_ordinary、fastokens_fused_split、fastokens_pre_tokenized_ordinary、encode_fastokens_ordinary 等函数，是复杂度最高的后端。

```
/// 使用 HuggingFace tokenizers 后端进行普通编码（绕过所有 added tokens）。
fn encode_hf_ordinary(tokenizer: &HfTokenizer, text: &str) -> tokenizers::Result<Vec<u32>> {
    // 使用空 AddedVocabulary 提取和归一化，不识别任何特殊 token
    let mut pretokenized =
        EMPTY_HF_ADDED_VOCABULARY.extract_and_normalize(tokenizer.get_normalizer(), text);

    if let Some(pre_tokenizer) = tokenizer.get_pre_tokenizer() {
        pre_tokenizer.pre_tokenize(&mut pretokenized)?;
    }
    // 直接使用模型 tokenize，绕过 post_process 中的 added tokens 注入
    pretokenized.tokenize(|normalized| tokenizer.get_model().tokenize(normalized.get()))?;
    let encoding = pretokenized.into_encoding(None, 0, OffsetType::Byte)?;
    let encoding = tokenizer.post_process(encoding, None, false)?;
    Ok(encoding.get_ids().to_vec())
}

/// 检测 fastokens 中是否包含 fused ByteLevel 的 Split，用于优化路径。
fn fastokens_fused_split(tokenizer: &FastokensTokenizer) ->
Option<&FastokensSplitPreTokenizer> {
    let FastokensPreTokenizer::Sequence(steps) = tokenizer.pre_tokenizer()? else {
        return None;
    };
    let [
        FastokensPreTokenizer::Split(split),
        FastokensPreTokenizer::ByteLevel(byte_level),
    ] = steps.as_slice()
    else {
        return None;
    };
    byte_level.is_bulk_only().then_some(split)
}

/// 构造不含 added tokens 的 PreTokenizedString。
fn fastokens_pre_tokenized_ordinary(
    tokenizer: &FastokensTokenizer,
    text: &str,
) -> FastokensPreTokenizedString {
    let normalized = tokenizer
        .normalizer()
        .map_or(Cow::Borrowed(text), |normalizer| normalizer.normalize(text));
    match normalized {
        Cow::Borrowed(_) => FastokensPreTokenizedString::from_text(text),
        Cow::Owned(text) => {
            let len = text.len();
            FastokensPreTokenizedString::new(
```

```
                text,
                vec![FastokensSplit {
                    range: 0..len,
                    token_id: None,
                }],
            )
        }
    }
}
```

rust/src/tokenizer/src/tekken.rs

Tekken 后端的 encode_ordinary 实现及单元测试，验证普通编码会绕过控制 token。

```
impl Tokenizer for TekkenTokenizer {
    fn encode_ordinary(&self, text: &str) -> Result<Vec<u32>> {
        // 调用底层 Tekken 编码，add_special_tokens = false，同时 bypass 特殊 token 匹配
        self.inner
            .encode(text, false, false)
            .map_err(|error| tokenizer_error!("encoding failed: {error}"))
    }
}

#[cfg(test)]
mod tests {
    use base64::Engine as _;
    use tekken::config::TokenizerVersion;
    use tekken::{SpecialTokenInfo, TokenInfo};
    use super::*;

    fn test_tokenizer() -> TekkenTokenizer {
        let vocab = (0_u8..=255)
            .map(|byte| TokenInfo {
                rank: byte as usize,
                token_bytes: base64::engine::general_purpose::STANDARD.encode([byte]),
                token_str: None,
            })
            .collect();
        let special_tokens = vec![SpecialTokenInfo {
            rank: 0,
            token_str: "<control>".to_string(),
            is_control: true,
        }];
        let inner = Tekkenizer::new(vocab, &special_tokens, r"(?s).", 257, 1, TokenizerVersion::V3,
        None)
            .expect("build Tekken tokenizer");
        TekkenTokenizer { inner }
    }

    #[test]
```

```
    fn ordinary_matches_tekkens_empty_special_encoding() {
        let tokenizer = test_tokenizer();
        let text = "user <control> text";
        let control_id = tokenizer.token_to_id("<control>").unwrap();
        let ordinary_ids = tokenizer.encode_ordinary(text).unwrap();

        assert_eq!(control_id, 0);
        assert_eq!(ordinary_ids, tokenizer.encode(text, false).unwrap());
        assert!(!ordinary_ids.contains(&control_id));
        assert_eq!(tokenizer.decode(&ordinary_ids, false).unwrap(), text);
    }
}
```

rust/src/tokenizer/src/tiktoken.rs

tiktoken 后端的 encode_ordinary 实现，直接利用底层库的原生方法，并附带测试验证绕过所有 registered added tokens。

```
impl Tokenizer for TiktokenTokenizer {
    fn encode_ordinary(&self, text: &str) -> Result<Vec<u32>> {
        // 直接委托给底层库的 encode_ordinary 方法，该原语会跳过所有注册的 added tokens 和特殊
        tokens
        Ok(match &self.backend {
            Backend::Riptoken(backend) => backend.inner.encode_ordinary(text),
            Backend::TiktokenRs(backend) => backend.inner.encode_ordinary(text),
        })
    }
}

#[cfg(test)]
mod tests {
    #[test]
    fn tiktoken_ordinary_bypasses_every_registered_added_token() {
        let dir = tempfile::tempdir().expect("create temp dir");
        let bpe_path = write_synthetic_bpe_file(dir.path());
        fs::write(dir.path().join("tokenizer_config.json"), r#"{
            "added_tokens_decoder": {
                "257": { "content": "<|im_end|>", "special": true },
                "258": { "content": "<|tool_call_begin|>", "special": false }
            }
        }"#).expect("write tokenizer_config.json");
        fs::write(dir.path().join("config.json"), r#"{"vocab_size": 260}"#).expect("write config.json");

        let input = "<|im_end|><|tool_call_begin|><|reserved_token_259|>";
        let expected: Vec<u32> = input.as_bytes().iter().copied().map(u32::from).collect();
        for backend in explicit_backends(&bpe_path) {
            assert_eq!(backend.encode("<|im_end|>", false).unwrap(), vec![257]);
            assert_eq!(backend.encode("<|tool_call_begin|>", false).unwrap(), vec![258]);
            assert_eq!(backend.encode("<|reserved_token_259|>", false).unwrap(), vec![259]);
            assert_eq!(backend.encode_ordinary(input).unwrap(), expected);
```

```
        }
    }
}
```

评论区精华

本 PR 未产生实质性的 Review 讨论。claude[bot] 自动提示代码审查配置，njhill 直接批准，表明实现清晰且共识明确。

风险与影响

- 风险: encode_ordinary 需要在所有 Tokenizer 实现中正确覆盖。当前测试仅涵盖 tiktoken 和 tekken 合成场景，未在真正模型上验证；HuggingFace 后端可能因自定义 normalizer/pre_tokenizer/post_processor 而行为异常。部分 mock 实现（incremental.rs 等）将方法标记为 unreachable，未来误调用将导致 panic。
- 影响: 影响范围限于 Rust 前端 tokenizer 模块，不改变现有 API 行为。下游已适配，该功能主要用于 prompt 结构化，对整体系统影响较小。

关联脉络

此 PR 是 Rust 前端基础建设的一部分，为后续 segment-aware prompt renderers 提供能力。与近期历史 PR（如 #49774 bugfix 推测解码、#50090 AttnRes kernels）无直接关联，但属于持续改进 Rust tokenizer 可编程性的工作。