

PR #54884 完整报告

vllm-project/vllm

[Rust Frontend] Use token-attributed text in reasoning and unified parsers

合并时间: 2026-09-04 02:05

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/54884>

执行摘要

本 PR 将 `DecodedText` (包含解码文本和每 token 的归因锚点) 引入 Rust 前端的 reasoning 和统一解析器管道。核心变更是将解析器内部缓冲区从 `String` 改为 `DecodedText`, 并利用新增的 `drain_prefix` 方法进行带归因的分割。这确保每个发出的 reasoning 片段都携带 token 归因, 同时标记 (如 ```) 和分隔符 token 的归因被丢弃, 不计入未来统计。此变更影响 22 个文件, 新增 1012 行, 删除 350 行, 为精确统计 `reasoning_tokens`` 奠定了基础。

功能与动机

为了支持未来精确统计 reasoning token 数量 (即 `usage.completion_tokens_details.reasoning_tokens`), 需要在解析器管道中传递 token 归因信息。正如 PR body 所述, 这是 “plumbing toward `usage.completion_tokens_details.reasoning_tokens`”。当前 PR 仅搭建管道, 实际计数和上报将在后续 PR 中完成。

实现拆解

- `DecodedText` 缓冲区操作: 在 `rust/src/tokenizer/src/incremental/attribution.rs` 中新增 `is_empty`、`take`、`clear` 和关键函数 `drain_prefix`。`drain_prefix` 实现了“首字节所有权”规则, 确保正确归属跨越分割点的 token。
- 解析器 trait 与核心类型更新: `rust/src/parser/src/unified/mod.rs` 中 `UnifiedParserEvent::Reasoning` 改为携带 `DecodedText`;
`rust/src/parser/src/reasoning/mod.rs` 中 `ReasoningDelta` 的字段类型相应更改;
`ReasoningParser::push` 和 `UnifiedParser::parse_into` trait 参数改为 `DecodedText`。
- 解析器实现迁移: 所有 reasoning 解析器 (`DelimitedReasoningParser`、`Step3p5ReasoningParser`、`MiniMaxM3ReasoningParser` 等) 和统一解析器 (`Gemma4UnifiedParser`、`InklingUnifiedParser`、`KimiK3UnifiedParser`) 的内部缓冲区改为 `DecodedText`, 解析循环中使用 `drain_prefix` 和 `append` 进行操作。
- 归因边界划定: 在 `rust/src/parser/src/unified/combined.rs` 的 `CombinedParser` 中, 将 reasoning 输出的 `content` (`DecodedText`) 的 `.text` 传递给工具解析器, 丢弃归因。
- 测试更新与新增: 更新现有测试以适配新接口; 新增 `drain_prefix` 单元测试; 为 `delimited reasoning` 解析器家族、`CombinedParser`、`Gemma4` 和 `Inkling` 新增归因守恒测试, 验证标记 token 归因被正确丢弃。

`rust/src/tokenizer/src/incremental/attribution.rs`

为 `DecodedText` 添加了 `drain_prefix` 等关键缓冲区操作方法, 是归因传递的基础。

```

// DecodedText 缓冲区操作，支持带归因的分割。
impl DecodedText {
    // 当文本和归因都为空时返回 true。
    pub fn is_empty(&self) -> bool {
        self.text.is_empty() && self.attributions.is_empty()
    }

    // 取出所有内容，留下空结构。
    pub fn take(&mut self) -> Self {
        take(self)
    }

    // 清空文本和归因，保留分配。
    pub fn clear(&mut self) {
        self.text.clear();
        self.attributions.clear();
    }

    // 分割并返回前缀 [0, len); 后缀保留在 self 中，
    // 其 token 锚点重新基准化为 -len。
    //
    // 锚点遵循首字节所有权规则：一个 `Visible` token 当其
    // 首字节位于前缀内时移至前缀，因此跨越分割点的 token
    // 属于持有其首字节的区间。一个 `ZeroWidth` token 当其
    // 偏移量 <= len 时移至前缀，这与解码器的 `take_ready`
    // 截断规则匹配，允许 `drain_prefix(text.len())` 取走
    // 尾部零宽 token。不同锚点种类的相对顺序得以保留。
    //
    // 如果 len 不是字符边界，则通过 `String::drain` panic。
    #[must_use]
    pub fn drain_prefix(&mut self, len: usize) -> DecodedText {
        let text: String = self.text.drain(..len).collect();
        let len = offset_as_u32(len);

        let mut prefix = DecodedText {
            text,
            attributions: SmallVec::new(),
        };
        for attribution in take(&mut self.attributions) {
            let in_prefix = match attribution.anchor {
                TokenAnchor::Visible { byte_offset } => byte_offset < len,
                TokenAnchor::ZeroWidth { byte_offset } => byte_offset <= len,
            };
            if in_prefix {
                prefix.attributions.push(attribution);
            } else {
                self.attributions.push(TokenAttribution {
                    token_id: attribution.token_id,
                    anchor: attribution.anchor.offset_by(-i64::from(len)),
                });
            }
        }
    }
}

```

```

        });
    }
}
prefix
}

// 追加另一个解码片段，重新基准化其 token 锚点。
pub fn append(&mut self, other: Self) {
    if self.is_empty() {
        *self = other;
        return;
    }

    let byte_offset = offset_as_u32(self.text.len());
    let other_len = offset_as_u32(other.text.len());
    let _combined_len = byte_offset.checked_add(other_len).expect("decoded text exceeds 4 GiB");

    self.text.push_str(&other.text);
    self.attributions.extend(other.attributions.into_iter().map(|attribution| {
        TokenAttribution {
            token_id: attribution.token_id,
            anchor: attribution.anchor.offset_by(i64::from(byte_offset)),
        }
    })));
}
}
}

```

rust/src/parser/src/unified/mod.rs

定义了 `UnifiedParserEvent::Reasoning(DecodedText)`，并更新了 `UnifiedParserOutput` 和 `UnifiedParser` trait，是归因传递的管道枢纽。

```

/// 一个由统一解析器发出的事件。
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum UnifiedParserEvent {
    /// 常规助手可见文本。
    Text(String),
    /// 从正常内容流中隐藏的推理文本。
    ///
    /// 携带生成它的 token 的归因，以便下游消费者可以精确统计推理 token 数。
    /// 标记 span 由解析器丢弃，从而将这些标记 token 排除在任何计数之外。
    Reasoning(DecodedText),
    /// 从可见助手文本中提取的工具调用更新。
    ToolCall(ToolCallDelta),
}

```

// ... (UnifiedParserOutput 的其他方法不变)

```
impl UnifiedParserOutput {
```

```

// 如果 delta 非空，则附加一个推理文本事件。
///
/// 一个仅包含零宽 token 的片段（空文本但非空归因）仍然被保留：
/// 这些 token 计为推理 token。
pub fn push_reasoning(&mut self, delta: DecodedText) {
    if delta.is_empty() {
        return;
    }
    if let Some(UnifiedParserEvent::Reasoning(last_text)) = self.events.last_mut() {
        last_text.append(delta);
        return;
    }
    self.events.push(UnifiedParserEvent::Reasoning(delta));
}

// ... (UnifiedParser trait 的其他方法不变 )

/// Incremental parser that extracts reasoning and tool-call events from assistant output.
pub trait UnifiedParser: Send {
    // ... ( 其他方法签名不变 )

    /// 将一个解码文本增量输入解析器，将已提交的输出追加到 output。
    fn parse_into(&mut self, delta: DecodedText, output: &mut UnifiedParserOutput) -> Result<()
    >;
}

```

评论区精华

CodeRabbitai: "Preserve a zero-width content token before a leading closer. If a zero-width delta arrives before `</mm:think>`, this line stores it in `leading_end_buffer`. The next push drains the closer and also discards that zero-width attribution..." 这指出了在 `MiniMaxM3ReasoningParser` 中一个可能丢失零宽 token 归因的边界情况。

CodeRabbitai: "Do not merge zero-width reasoning tokens into the held framing newline. If the stream emits `"reason\n"`, then a zero-width token, then `"</think>"`, this line puts the token attribution into `pending_reasoning_newline...`" 这指出了在 `Step3p5ReasoningParser` 中类似的问题。

这些讨论强调了在复杂解析器状态机中维护归因完整性的挑战。虽然 PR 已合并，但这些识别出的边界情况可能需要后续测试来覆盖。

风险与影响

- 正确性风险：`drain_prefix` 的归属规则需要与解码器完全一致，且复杂解析器路径中的集成测试覆盖需确保万无一失。
- 回归风险：大规模接口重构，但现有测试通过且新增归因守恒测试降低了风险。

- 性能风险: `DecodedText` 的 `attributions` 向量可能带来轻微内存和复制开销, 但文本 `chunk` 通常较小, 影响有限。
- 兼容性风险: 内部接口变更, 工具解析器在 `CombinedParser` 边界处被转换为纯文本, 保持向后兼容。

关联脉络

本 PR 是 #52910 (在增量解码器中引入 `DecodedText`) 的直接后续, 共同构成了“token 归因”功能的前半部分。后续 PR 将利用本 PR 搭建的管道, 在 API 响应中精确报告 `reasoning_tokens` 计数。这一功能线体现了 vLLM 对 token 使用统计精细化的持续投入。

建议

本 PR 值得精读。主要设计亮点包括:

1) `drain_prefix` 的“首字节所有权”规则为分割点归属提供了清晰语义; 2) 在 `CombinedParser` 处划定归因边界是务实的解耦; 3) 新增的归因守恒测试验证了核心不变量。建议关注 `attribution.rs` 的实现逻辑和解析器迁移的一致性模式。对于识别出的两个边界情况 (MiniMaxM3 和 Step3p5), 建议在后续 PR 中补充针对性测试。