PR #45802 完整报告

vllm-project/vllm

[Frontend] Support count_reasoning_tokens in the Streaming Parser Engine

合并时间：2026-08-15 21:51

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/45802

执行摘要

- 一句话：解析引擎支持 reasoning token 计数并输出到 usage 字段
- 推荐动作：值得精读。核心看点是 parser engine 管线的 token 计数设计：按字符摊分（_char_token_counts）+ 按切分回算（_pop_token_count）+ deferred 终端计数不丢失（scanner 侧前缀 / trailing 计数），是一套可复用的流式边界记账方法。另一个值得关注的设计决策是 ParserManager 对 ' 同引擎合并、异引擎组合 ' 的处理，以及围绕 DelegatingParser 语义保持的 review 讨论——这是理解 vLLM parser 子系统组合思想的关键材料。

功能与动机

PR body 明确目的：'Add token-aware reasoning token counting for the Streaming Parser Engine and surface the count through OpenAI-compatible usage fields.'。背景是推理模型（如 Qwen3 系列）的 reasoning token 需要独立计费与观测，而 OpenAI 兼容 API 的 usage.completion_tokens_details.reasoning_tokens 是标准字段，此前 vLLM 流式解析器无法提供。PR 中的 curl 验证即展示 enable_thinking 工具调用场景下返回 reasoning_tokens=18。另一个设计约束也写入 body：'Counts only REASONING_CHUNK tokens, excluding reasoning boundary terminals such as / '，确保边界标记本身不计入推理 token。

实现拆解

实现按 5 步完成：

1. scanner 注入 token 计数：vllm/parser/engine/token_id_scanner.py 给 TextChunk 增加 token_texts 与 token_count 字段；无特殊 token 时批量 _decode_tokens 直接携带计数；deferred 终端场景新增 _deferred_prefix_token_counts / _deferred_trailing_token_count，在 flush_pending 与 _resolve_deferred 中保证 detokenizer 回吐延迟时终端前文本与终端后 trailing 的计数都不丢失。
2. lexer 按字符摊分：vllm/parser/engine/incremental_lexer.py 中 LexToken 增加 token_count 字段；feed 通过 _char_token_counts 把 delta 的 token 数按 token 解码文本 find 定位摊分到每个字符位置，定位失败部分挂到 counts[0] 保证总数守恒；_drain 每切出一个 terminal 或 content 片段时用 _pop_token_count 按长度回算该片段的 token 数。

3. 引擎出口统一记账：vllm/parser/engine/streaming_parser_engine.py 新增 reasoning_token_count 属性与 _record_reasoning_tokens；feed（快速路径、单 TextChunk、多 item 三出口）与 finish 都先以 token_count 生成事件再统一累计 REASONING_CHUNK 事件的 token 数；reset 时清零 _reasoning_token_count 与 _message_header_token_count。
4. 计数语义收敛：vllm/parser/engine/parser_engine.py 的 count_reasoning_tokens 从按 start / end id 深度遍历改为直接返回引擎累计值；vllm/parser/engine/adapters.py 为兼容非流式或只有全文的场景引入 _counting_parser_engine 单趟解析回放，并用 _streaming_count_valid 标记流式计数是否有效（带 TODO 注明后续希望复用流式累计结果）。
5. serving 层暴露与 ParserManager 调整：vllm/entrypoints/openai/chat_completion/serving.py 新增 _make_completion_tokens_details 与 _include_reasoning_tokens_details 开关（由 reasoning_parser 配置驱动）；流式按 choice 累计 generated_token_ids 后整段计数，避免 delta 级覆盖；非流式累计 total_reasoning_tokens。vllm/parser/parser_manager.py 新增 _get_parser_engine_cls，仅当 reasoning 与 tool adapter 指向同一引擎类时才直接返回引擎类，否则保留 DelegatingParser 组合以维持原有单能力语义。

测试配套涉及 tests/parser/engine/test_engine.py（TestReasoningTokenCounts）、tests/parser/engine/test_parser_engine.py（ParserManager 组合语义四情形）、tests/entrypoints/openai/chat_completion/test_serving_chat.py（跨delta流式usage计数）、tests/entrypoints/openai/responses/test_serving_responses.py 与 tests/parser/engine/test_token_id_scanner.py；协议侧新增 CompletionTokenUsageInfo 数据契约，属于配套 schema 变更。

关键文件：

- vllm/parser/engine/streaming_parser_engine.py（模块 解析引擎；类别 source；类型 core-logic；符号 reasoning_token_count, _record_reasoning_tokens, _on_terminal, _emit_for_state）：计数体系的核心出口：新增 reasoning_token_count 属性与 _record_reasoning_tokens，feed / finish 所有出口统一记账，并透传 token_count 到 _emit_for_state / _on_terminal / _apply_transition。
- vllm/parser/engine/incremental_lexer.py（模块 词法分析；类别 source；类型 core-logic；符号 feed, _char_token_counts, _pop_token_count）：将 delta 的 token 数按字符粒度摊分（_char_token_counts），在切分 terminal / content 时用 _pop_token_count 回算每个 LexToken 的 token 数，是计数精确到事件的关键一环。
- vllm/parser/engine/token_id_scanner.py（模块 扫描器；类别 source；类型 core-logic；符号 _decode_tokens）：TextChunk 增加 token_texts / token_count，deferred 终端场景新增前缀 / trailing 计数，保证 detokenizer 回吐延迟时计数不丢，是流式计数一致性的基础。
- vllm/entrypoints/openai/chat_completion/serving.py（模块 服务层；类别 source；类型 core-logic；符号 _make_completion_tokens_details）：用户可见出口：_make_completion_tokens_details 与 _include_reasoning_tokens_details 开关，流式按 choice 累计 token_ids 后计数、非流式累计 total_reasoning_tokens，并写入各 usage 分

支。

- vllm/parser/parser_manager.py（模块 解析管理；类别 source；类型 core-logic；符号 _get_parser_engine_cls）：review 争议焦点：_get_parser_engine_cls 探测 adapter 的引擎类，仅当 reasoning 与 tool 指向同一引擎类时合并返回引擎类，否则保留 DelegatingParser 组合，保护单能力部署语义。
- vllm/parser/engine/adapters.py（模块 适配器；类别 source；类型 core-logic）：count_reasoning_tokens 兼容非流式 / 全文场景：_streaming_count_valid 标记流式计数有效性，_counting_parser_engine 做单趟解析回放。
- vllm/parser/engine/parser_engine.py（模块 解析引擎；类别 source；类型 core-logic）：count_reasoning_tokens 从按 start / end id 的深度遍历改为直接返回引擎累计值 reason_token_count，语义发生根本变化，所有依赖该接口的调用方行为随之改变。
- vllm/entrypoints/openai/engine/protocol.py（模块 协议层；类别 source；类型 data-contract；符号 CompletionTokenUsageInfo）：新增 CompletionTokenUsageInfo 数据契约（reasoning_tokens 字段），是 usage 输出的 schema 基础。
- tests/parser/engine/test_engine.py（模块 解析引擎；类别 test；类型 test-coverage；符号 _token_think_config, TestReasoningTokenCounts, test_counts_reasoning_and_excludes_boundaries_and_content, test_counts_tokens_after_final_deferred_start_terminal）：TestReasoningTokenCounts 覆盖边界终端排除、deferred start 终端后计数、deferred 推理 token 归属三条关键路径，是计数正确性的核心验证。
- tests/parser/engine/test_parser_engine.py（模块 解析管理；类别 test；类型 test-coverage；符号 test_parser_manager_uses_shared_engine_directly, test_parser_manager_preserves_reasoning_only_adapter, test_parser_manager_preserves_tool_only_adapter, test_parser_manager_rejects_non_engine_adapter_metadata）：覆盖 ParserManager 合并引擎类的四种情形（直接复用、reasoning-only、tool-only、非法 metadata）与最终非流式解析后的计数，直接回应 review 中提出的回归担忧。
- tests/entrypoints/openai/chat_completion/test_serving_chat.py（模块 服务层；类别 test；类型 test-coverage；符号 _stream_request_outputs, test_streaming_reasoning_usage_counts_across_deltas, test_completion_tokens_details）：跨 delta 流式 usage 统计测试 test_streaming_reasoning_usage_counts_across_deltas 与 _make_completion_tokens_details 单测，验证 serving 层计数输出。

关键符号：reasoning_token_count, _record_reasoning_tokens, _char_token_counts, _pop_token_count, _decode_tokens, count_reasoning_tokens, _get_parser_engine_cls, _make_completion_tokens_details

关键源码片段

vllm/parser/engine/streaming_parser_engine.py

计数体系的核心出口：新增 reasoning_token_count 属性与 _record_reasoning_tokens，feed / finish 所有出口统一记账，并透传 token_count 到 _emit_for_state / _on_terminal /

_apply_transition。

```
    @property
    def reasoning_token_count(self) -> int:
        # 对外暴露当前累计的 reasoning token 数，供 chat serving 层写入
        # usage.completion_tokens_details.reasoning_tokens，也供
        # ParserEngine.count_reasoning_tokens() 直接读取。
        return self._reasoning_token_count


    def _record_reasoning_tokens(self, events: Sequence[SemanticEvent]) -> None:
        # 只累计 REASONING_CHUNK 事件的 token_count；<think> / </think> 等
        # 边界终端走状态迁移，产出 REASONING_START / REASONING_END 事件，
        # 天然不会被计入，从而保证"只统计推理正文 token"的语义。
        self._reasoning_token_count += sum(
            event.token_count
            for event in events
            if event.type == EventType.REASONING_CHUNK
        )


    def feed(self, delta_text: str, delta_token_ids: Sequence[int]) -> list[SemanticEvent]:
        if delta_token_ids:
            self._ever_had_token_ids = True

        # 快速路径：delta 为纯内容且无特殊 token 时直接生成事件，
        # 但必须把 len(delta_token_ids) 作为 token_count 传给 _emit_for_state，
        # 否则走捷径会漏掉 reasoning 计数。
        if (
            delta_text
            and not self._lexer.buffer
            and not self._scanner._deferred_terminals
            and self._lexer._literal_first_chars.isdisjoint(delta_text)
        ):
            has_special = False
            for tid in delta_token_ids:
                if tid in self._resolved_token_ids:
                    has_special = True
                    break
            if not has_special:
                events = self._emit_for_state(
                    delta_text, token_count=len(delta_token_ids)
                )
                self._record_reasoning_tokens(events)
                return events

        scanner_items = self._scanner.scan(delta_text, delta_token_ids)

        if len(scanner_items) == 1 and isinstance(scanner_items[0], TextChunk):
```

```
        # 单 TextChunk 路径：把 scanner 摊分好的 token_texts / token_count
        # 透传给 lexer，lexer 按切分粒度回算每个 LexToken 的 token 数。
        item = scanner_items[0]
        lex_tokens = self._lexer.feed(item.text, item.token_texts, item.token_count)
        if len(lex_tokens) == 1 and lex_tokens[0].terminal == CONTENT_TERMINAL:
            events = self._emit_for_state(
                lex_tokens[0].value,
                token_count=lex_tokens[0].token_count,
            )
        else:
            events = self._process_lex_tokens(lex_tokens)
        self._record_reasoning_tokens(events)
        return events

    events = self._process_scanner_items(scanner_items)
    self._record_reasoning_tokens(events)
    return events
```

vllm/parser/engine/incremental_lexer.py

将 delta 的 token 数按字符粒度摊分（_char_token_counts），在切分 terminal / content 时用 _pop_token_count 回算每个 LexToken 的 token 数，是计数精确到事件的关键一环。

```
@staticmethod
def _char_token_counts(
    text: str,
    token_texts: tuple[str, ...],
    token_count: int,
) -> list[int]:
    # 把本次 delta 的 token 数按字符粒度摊分到逐字符数组。
    # 优先依据 token 解码文本在 text 中逐个 find 定位锚点，定位到的
    # 字符累加 1；定位失败的（上下文相关解码或 detokenizer 不回吐
    # 文本）token 数统一记在 counts[0]，保证总数守恒。
    counts = [0] * len(text)
    if not text:
        return counts
    if token_texts:
        pos = 0
        assigned = 0
        for token_text in token_texts:
            if not token_text:
                continue
            found = text.find(token_text, pos)
            if found < 0:
                continue
            counts[found] += 1
            assigned += 1
            pos = found + len(token_text)
        missing = token_count - assigned
        if missing > 0:
```

```
            counts[0] += missing
    elif token_count:
        # 没有 token_texts（非流式或 tokenizer 不可用）时，
        # 把整段 token 数挂在首字符上。
        counts[0] = token_count
    return counts


def _pop_token_count(self, length: int) -> int:
    # 当 buffer 切出一个 terminal 或 content 片段时，按片段长度回取
    # 对应 token 数，并同步从待摊分队列中移除已消费部分。
    token_count = sum(self._token_counts[:length])
    del self._token_counts[:length]
    return token_count
```

vllm/entrypoints/openai/chat_completion/serving.py

用户可见出口：_make_completion_tokens_details 与 _include_reasoning_tokens_details 开关，流式按 choice 累计 token_ids 后计数、非流式累计 total_reasoning_tokens，并写入各 usage 分支。

```
def _make_completion_tokens_details(
    reasoning_tokens: int,
) -> CompletionTokenUsageInfo:
    # 构造 OpenAI 兼容的 usage.completion_tokens_details，
    # 当前仅暴露 reasoning_tokens 字段。
    return CompletionTokenUsageInfo(reasoning_tokens=reasoning_tokens)
```

流式路径按 choice 累计完整 token 序列后再计数，避免每个 delta 单独计数覆盖上一轮结果：

```
# TODO: Remove once all reasoning parsers use the Parser Engine.
generated_token_ids: list[list[int]] = [[] for _ in range(num_choices)]
previous_reasoning_tokens = [0] * num_choices
# ...
if parser is not None:
    # 按 choice 累计完整 token 序列后整体计数，避免“每个 delta
    # 单独计数覆盖上一轮结果”的问题。
    generated_token_ids[i].extend(output.token_ids)
    previous_reasoning_tokens[i] = parser.count_reasoning_tokens(
        tuple(generated_token_ids[i])
    )
```

评论区精华

sfeng33 的三条评论是本 PR 最有价值的交锋：1）ParserManager 返回裸 ParserEngine 会导致 parser.reasoning_parser 为 None，serving 层丢失 reasoning_ended 与 reasoning_parser_kwargs，Qwen3 关闭 thinking + JSON schema 时结构输出会失效；2）单能力场景行为回归——改动前 reasoning-only 服务把 tool-call 文本保留在 content、tool-only 服务把 块保留在 content，改动后这些文本可能丢失；3）流式逻辑中

previous_reasoning_tokens 每个 delta 被重新赋值，疑似覆盖前值。作者对第 3 条回复 ' Fixed.'，最终实现按 choice 累计完整 token 序列再计数；对前两条通过把合并条件收紧为 ' reasoning 与 tool 指向同一引擎类 ' 并新增 reasoning-only / tool-only 组合测试修复。sfeng33 最终 APPROVED。

- ParserManager 返回裸引擎类破坏 reasoning_parser 属性 (correctness): 作者修复为仅当 reasoning 与 tool adapter 指向同一引擎类时才直接返回引擎类，否则保留 DelegatingParser 组合；新增 test_parser_manager_preserves_reasoning_only_adapter 等测试。
- 单 parser 场景（仅 reasoning 或仅 tool）行为回归 (correctness): 通过同一引擎类比较条件保护 DelegatingParser 语义，新增 test_parser_manager_preserves_tool_only_adapter 与 preserves_reasoning_only_adapter 覆盖。
- 流式计数每个 delta 独立计算是否覆盖前值 (correctness): 修复为按 choice 累计 generated_token_ids 后传入完整序列再计数；新增 test_streaming_reasoning_usage_counts_across_deltas 验证跨 delta 累计正确性。

风险与影响

- 风险：技术风险集中在三点：其一，ParserManager 直接返回裸引擎类绕过了 DelegatingParser 的组合语义，导致 parser.reasoning_parser 为 None、serving 层丢失 reasoning_ended 与 reasoning_parser_kwargs，sfeng33 以 Qwen3 enable_thinking=false + JSON schema 场景举例说明会导致 grammar 不生效、schema 非法输出或生成到 max_tokens；该问题合入前已修复，但新合并语义仍需要在 reasoning-only / tool-only 部署上做回归验证。其二，token 计数依赖字符级摊分启发式（_char_token_counts 中 token 文本 find 失败时把缺失数挂在 counts[0]），deferred 终端 + holdback 文本的组合路径存在计数错位风险，测试主要覆盖 think 场景，其他 terminal 组合覆盖有限。其三，实现层面有性能隐患：chat_completion_stream_generator 每个 delta 都累计 generated_token_ids 并重新线性扫描计数，长序列下接近 O(n²)；非流式路径则可能触发一次额外的单趟解析回放（adapters.py 的 _counting_parser_engine），带来重复解析开销。
- 影响：用户侧：reasoning 模型调用方首次获得 completion_tokens_details.reasoning_tokens 字段，可用于推理 token 计费与过程观测；字段仅在配置了 reasoning_parser 时输出，默认响应不变，向后兼容。系统侧：parser engine 核心管线（scanner / lexer / engine）的每个 feed 出口都新增计数记账，属于核心路径变更，影响所有基于新 parser engine 的推理与工具调用请求；同时 count_reasoning_tokens 语义从 ' 离线按 token_ids 重算 ' 变为 ' 引擎在线累计值 '，依赖该接口的调用方行为发生变化。团队侧：为 parser 子系统与 entrypoints 服务层的协同演进提供了基线，新增约 260 行测试覆盖流式 / 非流式、单能力 parser、Responses API 等场景。
- 风险标记：核心解析管线变更，计数依赖字符摊分启发式，回归风险（DelegatingParser 语义），流式计数 O(n²) 隐患，非流式额外解析开销

关联脉络

- PR #50487 [Model][Spec Decode] Tap the pre-norm AttnRes mixture as the Kimi K3 DFlash aux state: parser_manager.py 保留的 kimi_k3 特殊分支与本 PR 同属 parser 子系统；K3 推理解析器后续同样需要接入 reasoning token 计数体系。
- PR #52384 [Rust Frontend][gRPC] Preserve skip_special_tokens decoding option: rust 前端 gRPC 解码选项对齐 Python API，与本 PR 补齐 OpenAI 兼容 usage 字段同属前端 API 一致性演进方向。
- PR #52261 [Frontend] Consolidate entrypoint exception handler: entrypoints 层持续收敛重构（异常处理独立成包），本 PR 在 chat serving 层新增 usage 统计逻辑，说明入口服务层仍处于活跃演进期。