PR #44499 完整报告

vllm-project/vllm

[Rust Frontend] Add /pause, /resume, /is_paused endpoints

合并时间：2026-06-08 17:28

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/44499

执行摘要

本 PR 为 vLLM Rust 前端新增三个管理端点：POST /pause、POST /resume 和 GET /is_paused，用于暂停 / 恢复调度器并查询状态。所有端点仅在 dev-mode 下注册，通过 EngineCoreClient 新增方法与引擎通信，并包含完整的集成测试。变更安全、范围有限，与现有 sleep/wake 端点模式一致。

功能与动机

参考 Rust 前端路线图 #44280，本 PR 提供了与 Python frontend 等价的调度器生命周期管理能力，支持权重更新等场景下暂停生成。

实现拆解

1. 客户端扩展：在 rust/src/engine-core-client/src/client.rs 的 EngineCoreClient 实现中新增 pause_scheduler、resume_scheduler、is_scheduler_paused 三个 public 方法。pause_scheduler 接收 mode 和 clear_cache 参数，resume_scheduler 无参数，is_scheduler_paused 从所有引擎收集 bool 结果并进行一致性检查，结果不一致时返回 InconsistentUtilityResults 错误。
2. 路由处理器：新建 rust/src/server/src/routes/pause.rs，定义 PauseParams（包含 mode 和 clear_cache，默认值分别为 "abort" 和 true）、StatusResponse、IsPausedResponse 三个结构体，以及 pause、resume、is_paused 三个异步函数。pause 处理器校验 mode 是否在合法集合 ["abort", "wait", "keep"] 中，然后调用客户端方法并返回 {"status": "paused"}；resume 直接调用客户端方法并返回 {"status": "resumed"}；is_paused 调用客户端方法并返回 {"is_paused": bool}。
3. 路由注册：在 rust/src/server/src/routes.rs 的 dev-mode 路由块中添加三条路由（/pause POST, /resume POST, /is_paused GET），并声明 mod pause。
4. 测试：在 rust/src/server/src/routes/tests.rs 中添加四个集成测试：
   - pause_route_uses_python_compatible_default_query_values：无参数 POST /pause 应发送 ["abort", true]，返回 200。
   - pause_route_rejects_invalid_mode：传入非法 mode 应返回 400 和错误详情。
   - resume_route_sends_no_args：POST /resume 应发送空参数数组。
   - is_paused_route_returns_json_payload：GET /is_paused 应返回 {"is_paused": true}。同时扩展 admin_routes_are_hidden_when_dev_mode_is_disabled 测试，确保新端点被隐藏。

rust/src/server/src/routes/pause.rs

新增文件，实现三个端点核心逻辑，包括参数解析、校验、引擎调用和响应构造。

```
// 文件：rust/src/server/src/routes/pause.rs
// 实现 /pause, /resume, /is_paused 三个管理端点

use std::sync::Arc;
use axum::{Json, extract::{Query, State}};
use serde::{Deserialize, Serialize};
use crate::{error::ApiError, state::AppState, utils::utility_call_error};

/// 查询参数：`mode` 默认 "abort"，`clear_cache` 默认 true
#[derive(Debug, Deserialize)]
pub(crate) struct PauseParams {
    #[serde(default = "default_pause_mode")]
    mode: String,
    #[serde(default = "default_clear_cache")]
    clear_cache: bool,
}

/// 统一响应结构
#[derive(Serialize)]
pub(crate) struct StatusResponse {
    status: &'static str,
}

/// /is_paused 响应
#[derive(Serialize)]
pub(crate) struct IsPausedResponse {
    is_paused: bool,
}

/// 合法的 pause mode，对应 Python 端的 `PauseMode`
const VALID_PAUSE_MODES: [&str; 3] = ["abort", "wait", "keep"];

fn default_pause_mode() -> String {
    "abort".to_string()
}

const fn default_clear_cache() -> bool {
    true
}

// TODO: Python 端还接受已废弃的 `wait_for_inflight_requests` 参数，
// 此处未实现，统一使用 `mode` 参数

/// 暂停调度器，用于停止生成（如进行权重更新）
pub async fn pause(
    State(state): State<Arc<AppState>>,
```

```
    Query(params): Query<PauseParams>,
) -> Result<Json<StatusResponse>, ApiError> {
    // 校验 mode 参数，无效则返回 400
    if !VALID_PAUSE_MODES.contains(&params.mode.as_str()) {
        return Err(ApiError::invalid_request(
            format!("Invalid pause mode '{}'; expected one of: abort, wait, keep", params.mode),
            Some("mode"),
        ));
    }
    // 通过 engine_core_client 调用引擎端的 pause_scheduler
    state.engine_core_client()
        .pause_scheduler(&params.mode, params.clear_cache)
        .await
        .map_err(|error| utility_call_error("pause", error))?;
    Ok(Json(StatusResponse { status: "paused" }))
}

/// 恢复调度器运行
pub async fn resume(
    State(state): State<Arc<AppState>>,
) -> Result<Json<StatusResponse>, ApiError> {
    state.engine_core_client()
        .resume_scheduler()
        .await
        .map_err(|error| utility_call_error("resume", error))?;
    Ok(Json(StatusResponse { status: "resumed" }))
}

/// 查询调度器是否处于暂停状态
pub async fn is_paused(
    State(state): State<Arc<AppState>>,
) -> Result<Json<IsPausedResponse>, ApiError> {
    let is_paused = state.engine_core_client()
        .is_scheduler_paused()
        .await
        .map_err(|error| utility_call_error("is_paused", error))?;
    Ok(Json(IsPausedResponse { is_paused }))
}
```

rust/src/engine-core-client/src/client.rs

核心客户端扩展，新增 pause_scheduler、resume_scheduler、is_scheduler_paused 三个方法，其中 is_scheduler_paused 实现了跨引擎一致性检查。

```
// 文件：rust/src/engine-core-client/src/client.rs（新增方法片段）
// 注意：这些方法位于 impl EngineCoreClient 块中

/// 暂停调度器，参数由 Python 前端定义
pub async fn pause_scheduler(&self, mode: &str, clear_cache: bool) -> Result<()> {
    // 调用引擎端注册的 utility 方法 "pause_scheduler"
```

```
    self.call_utility::<(), _>("pause_scheduler", (mode, clear_cache)).await?;
    Ok(())
}

/// 恢复调度器（无参数）
pub async fn resume_scheduler(&self) -> Result<()> {
    self.call_utility::<(), _>("resume_scheduler", ()).await?;
    Ok(())
}

/// 查询调度器是否暂停，跨引擎一致性检查
pub async fn is_scheduler_paused(&self) -> Result<bool> {
    // 从所有引擎收集结果
    let results: Vec<bool> = self.call_utility("is_scheduler_paused", ()).await?;
    // 确保 results 非空（启动时保证 engine_count >= 1）
    let first = *results.first().ok_or_else(|| Error::InconsistentUtilityResults {
        method: "is_scheduler_paused".to_string(),
        values: "[]".to_string(),
    })?;
    // 所有引擎结果必须一致，否则报错
    if results.iter().all(|&v| v == first) {
        Ok(first)
    } else {
        Err(Error::InconsistentUtilityResults {
            method: "is_scheduler_paused".to_string(),
            values: format!("{results:?}"),
        })
    }
}
```

评论区精华

Review 中仅有一条 BugenZhao 的 "LGTM" 评论并批准，无额外修改要求。

风险与影响

风险：新端点仅在 dev-mode 启用，对生产无影响。is_scheduler_paused 的跨引擎一致性检查可能因状态不一致抛出异常，但这是设计意图，避免静默错误。如果引擎端未实现对应 utility 方法，会返回清晰错误。 影响：用户可通过 Rust 前端 REST API 控制调度器暂停 / 恢复，便于集成。团队可参考此实现模式开发更多 admin 端点。

关联脉络

本 PR 是 #44280 Rust 前端路线图中 admin/lifecycle APIs 的一部分。此前已有 sleep/wake（/sleep、/wake_up、/is_sleeping）端点，本 PR 扩展了生命周期管理能力。