PR #46051 完整报告

vllm-project/vllm

[Rust Frontend][Perf] Use dedicated runtime for HTTP/request-processing/ZMQ

合并时间：2026-06-23 12:03

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/46051

执行摘要

本 PR 为 Rust 前端引入三层 Tokio runtime 隔离（HTTP / 请求处理 / ZMQ 通信），将 CPU 密集的请求预处理和引擎传输从 HTTP 工作线程中分离。在 300k 长文本压力测试下，吞吐量提升 30%，/health 尾部延迟（p99.9）下降 93%。变更集中在 rust/ 目录下的 13 个文件，新增 3 个核心源码文件。

功能与动机

动机：高并发下 CPU 密集的请求预处理（如 tokenization、chat-template 渲染）和 ZMQ 引擎传输会长时间占用 HTTP runtime 的 worker 线程，导致轻量接口（如 /health）尾部延迟劣化，并限制请求吞吐。PR body 中通过 benchmark 展示了问题。

目标：将不同职责的工作分到独立 runtime，避免相互阻塞，提升服务端响应性和吞吐。

实现拆解

1. BackgroundShutdownRuntime 封装（engine-core-client/src/runtime.rs）：新增 BackgroundShutdownRuntime 结构体，包装 Tokio Runtime 并在 Drop 时调用 shutdown_background()，实现无阻塞关闭，同时通过 Deref/DerefMut 透明代理。
2. 构建 ZMQ runtime（同上文件）：build_zmq_runtime() 创建默认 4 线程的 Tokio runtime，线程名带有递增序号，用于驱动所有 ZMQ 发送 / 接收、输出分发和中止处理循环。
3. 构建 Request runtime（server/src/runtime.rs）：build_request_runtime() 创建多线程 runtime，线程数默认取可用并行度，上限 32，用于运行 offload 的请求处理路径。
4. Tower middleware offload（server/src/middleware/offload.rs）：新增 RequestRuntimeService 实现 Tower Service，在 call() 中匹配预定义的 OFFLOADED_PATHS，将匹配请求通过 AbortOnDropHandle 在 Request runtime 上执行，不阻塞 HTTP runtime。若任务失败返回 500 错误。
5. 集成到核心结构：
   - EngineCoreClient 现在持有 BackgroundShutdownRuntime，将输出循环、分发循环、中止循环全部 spawn 到该 runtime（client.rs）。
   - AppState 添加 OnceLock<BackgroundShutdownRuntime> 懒初始化 Request runtime，并通过 request_runtime() 暴露（state.rs）。
   - lib.rs 中将 request_runtime_layer 应用到 HTTP 路由。

6. 配置项：新增 VLLM_RS_ZMQ_WORKER_THREADS 和
   VLLM_RS_REQUEST_WORKER_THREADS 两个环境变量，允许用户自定义 worker 线程数。

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

新增 ZMQ runtime 构建函数和 BackgroundShutdownRuntime 封装，是 runtime 隔离的基础。

```
// rust/src/engine-core-client/src/runtime.rs
// 封装 Tokio Runtime，Drop 时后台关闭，防止嵌套阻塞
pub struct BackgroundShutdownRuntime(ManuallyDrop<Runtime>);

impl Drop for BackgroundShutdownRuntime {
    fn drop(&mut self) {
        // 安全：仅一次 take，后续不再使用
        let runtime = unsafe { ManuallyDrop::take(&mut self.0) };
        runtime.shutdown_background(); // 非阻塞关闭
    }
}

impl Deref for BackgroundShutdownRuntime {
    type Target = Runtime;
    fn deref(&self) -> &Self::Target { &self.0 }
}

impl DerefMut for BackgroundShutdownRuntime {
    fn deref_mut(&mut self) -> &mut Self::Target { &mut self.0 }
}

impl From<Runtime> for BackgroundShutdownRuntime {
    fn from(runtime: Runtime) -> Self { Self(ManuallyDrop::new(runtime)) }
}

// ZMQ runtime 默认 4 线程，多 engine 共享 socket，benchmark 显示足够
const ZMQ_WORKER_THREADS_ENV: &str = "VLLM_RS_ZMQ_WORKER_THREADS";
const DEFAULT_ZMQ_WORKER_THREADS: usize = 4;

static ZMQ_RUNTIME_SEQUENCE: OnceLock<AtomicUsize> = OnceLock::new();

/// 构建 ZMQ runtime，每次调用生成带独立序列号线程名的 runtime
pub(crate) fn build_zmq_runtime() -> BackgroundShutdownRuntime {
    let sequence = ZMQ_RUNTIME_SEQUENCE
        .get_or_init(|| AtomicUsize::new(0))
        .fetch_add(1, Ordering::Relaxed);

    tokio::runtime::Builder::new_multi_thread()
        .worker_threads(zmq_worker_threads())
        .thread_name_fn(move || format!("vllm-zmq-{sequence}"))
        .enable_all()
        .build()
        .expect("failed to build vLLM ZMQ runtime")
```

```
        .into()
}

fn zmq_worker_threads() -> usize {
    std::env::var(ZMQ_WORKER_THREADS_ENV)
        .ok()
        .and_then(|v| v.parse().ok())
        .filter(|&v| v > 0)
        .unwrap_or(DEFAULT_ZMQ_WORKER_THREADS)
}
```

rust/src/server/src/middleware/offload.rs

新增 Tower middleware，将 CPU 密集型请求 offload 到 Request runtime，核心性能优化点。

```
// rust/src/server/src/middleware/offload.rs
// 需要 offload 的路径列表（HTTP + gRPC）
const OFFLOADED_PATHS: &[&str] = &[
    "/v1/chat/completions",
    "/v1/completions",
    "/tokenize",
    "/detokenize",
    "/inference/v1/generate",
    "/vllm.Generate/Generate",
    "/vllm.Generate/GenerateStream",
];

// 返回 Tower layer，包装 inner service
pub(crate) fn request_runtime_layer<S>(
    state: Arc<AppState>,
) -> impl tower::Layer<S, Service = RequestRuntimeService<S>> + Clone {
    layer_fn(move |inner| RequestRuntimeService {
        inner,
        state: state.clone(),
    })
}

#[derive(Clone)]
pub(crate) struct RequestRuntimeService<S> {
    inner: S,
    state: Arc<AppState>,
}

impl<S, B> Service<Request<B>> for RequestRuntimeService<S>
where /* ... bounds ... */
{
    type Future = BoxFuture<'static, Result<Self::Response, Self::Error>>;

    fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>> {
        self.inner.poll_ready(cx)
```

```
    }

    fn call(&mut self, req: Request<B>) -> Self::Future {
        if !should_offload(req.uri().path()) {
            // 轻量路径直接走 HTTP runtime
            return Box::pin(self.inner.call(req));
        }

        // 替换 inner 为克隆，避免 move 冲突
        let clone = self.inner.clone();
        let mut inner = std::mem::replace(&mut self.inner, clone);
        // 在 Request runtime 上执行，AbortOnDropHandle 处理取消
        let task = AbortOnDropHandle::new(
            self.state.request_runtime().spawn(inner.call(req))
        );

        Box::pin(async move {
            match task.await {
                Ok(result) => result,
                Err(error) => {
                    error!(%error, "request runtime task failed");
                    Ok(S::Response::request_runtime_error_response())
                }
            }
        })
    }
}

// 辅助 trait 生成错误响应
trait RequestRuntimeErrorResponse {
    fn request_runtime_error_response() -> Self;
}

impl RequestRuntimeErrorResponse for Response {
    fn request_runtime_error_response() -> Self {
        server_error!("request runtime task failed").into_response()
    }
}

impl RequestRuntimeErrorResponse for axum::http::Response<tonic::body::Body> {
    fn request_runtime_error_response() -> Self {
        Status::internal("request runtime task failed").into_http()
    }
}

fn should_offload(path: &str) -> bool {
    OFFLOADED_PATHS.contains(&path)
}
```

```
#[cfg(test)]
mod tests {
    // 测试验证 offload 路径正确性
}
```

rust/src/server/src/runtime.rs

新增 Request runtime 构建逻辑，封装了线程数配置和 background shutdown。

```
// rust/src/server/src/runtime.rs
use tokio::runtime::Builder;
use vllm_engine_core_client::runtime::BackgroundShutdownRuntime;

const REQUEST_WORKER_THREADS_ENV: &str = "VLLM_RS_REQUEST_WORKER_THREADS";
const DEFAULT_MAX_REQUEST_WORKER_THREADS: usize = 32;

pub(crate) fn build_request_runtime() -> BackgroundShutdownRuntime {
    Builder::new_multi_thread()
        .enable_all()
        .thread_name("vllm-request")
        .worker_threads(request_worker_threads())
        .build()
        .expect("failed to build request runtime")
        .into()
}

fn request_worker_threads() -> usize {
    // 优先使用环境变量
    if let Some(value) = std::env::var_os(REQUEST_WORKER_THREADS_ENV) {
        match value.to_string_lossy().parse::<usize>() {
            Ok(worker_threads) if worker_threads > 0 => return worker_threads,
            _ => warn!("ignoring invalid {}: {:?}", REQUEST_WORKER_THREADS_ENV, value),
        }
    }
    // 否则使用可用并行度，上限 32
    std::thread::available_parallelism()
        .map(|p| {
            let available = p.get();
            let capped = available.min(DEFAULT_MAX_REQUEST_WORKER_THREADS);
            if capped < available {
                info!("capping request runtime worker threads from {} to {}, set {} to override",
                    available, capped, REQUEST_WORKER_THREADS_ENV);
            }
            capped
        })
        .unwrap_or(DEFAULT_MAX_REQUEST_WORKER_THREADS)
}
```

评论区精华

- ZMQ 线程数：njhill 询问是否可减少调优参数，BugenZhao 回复 benchmark 表明 4 线程更好，并保留环境变量支持。
- Offload 粒度：njhill 建议更精细控制，BugenZhao 承认并调整了路径列表（增加 /detokenize），同时指出当前基于路径匹配已能满足隔离控制面负载的核心目标。

风险与影响

- 风险：
  - 若 Request runtime 任务 panic，中间件返回 500，可能影响用户请求。
  - 新增 runtime 增加线程和内存开销，默认值需要验证。
  - OFFLOADED_PATHS 可能遗漏某些 CPU 密集路径，需要持续维护。
- 影响：
  - 仅影响 Rust 前端服务层，Python 后端无感知。
  - 高并发吞吐提升 30%，控制面延迟降低 90%+。
  - 运维需关注两个新环境变量的配置。

关联脉络

本 PR 是目前 vLLM Rust 前端性能优化的核心提交，与此前和后续的前端性能改进（如请求批处理、协议优化）共同构成 Rust 前端的整体性能提升。当前未发现直接依赖的其他 PR。