PR #33026 完整报告

sgl-project/sglang

[rust-server] fix TCP-layer TTFT stalls

合并时间：2026-08-02 03:13

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/33026

PR 33026：Rust server TCP 层 TTFT 卡顿修复

执行摘要

本 PR 针对 Rust server HTTP 前端实测到的两类 TCP 层 TTFT（首 token 延迟）卡顿各做一处小改动，共 2 个文件 +29/−11。第一处是 大请求体约 200 ms 停顿：请求头 + body 单次突发超过内核初始接收缓冲（tcp_rmem[1] = 87380 B）时，尾部段在 handler 轮询 body 前被丢弃，恢复依赖客户端 RTO 重传（实测 tcpi_total_retrans=3，阈值二分定在 75→91 KB）；修复是在 runtime.rs 用 tokio TcpSocket 在 bind 前设置 16 MiB SO_RCVBUF（accept 出的连接继承）。第二处是 keep-alive 连接约 13 ms 延迟：hyper/axum 不设 TCP_NODELAY，Nagle 与 delayed-ACK 相互等待；修复是在 api_server.rs 用 ListenerExt::tap_io 对每个连接 set_nodelay(true)。实测 raw-socket 大包探针从 208 ms 降到 0.4 ms，keep-alive ×128 从 40.9 ms 降到 26.4 ms，gpt-oss / MiMo 各长度 TTFT 全面对齐甚至反超 Python TM，decode 路径（ITL）不受影响。

功能与动机

PR body 给出了明确的对比方法与数据（8×B300、bench_serving c=1、seed 42、--disable-radix-cache）：

> "A headers+body burst larger than the kernel's initial rcvbuf (tcp_rmem[1] = 87380 B) drops its tail before hyper polls the body; recovery is the client's RTO retransmit (~200 ms, measured tcpi_total_retrans=3; threshold bisected to 75→91 KB)."
>
> "hyper/axum doesn't set TCP_NODELAY (Python's asyncio does)"

动机很直接：让 Rust server 的 TTFT 与 Python TM 对齐。基准表显示 Python TM 在 gpt-oss 16K/64K、MiMo 64K/256K 上是 166/1080/1986/9669 ms，而修复前 Rust 为 192/1253/2197/9716 ms，修复后 157/1047/1957/9513 ms——两处 TCP 修复后 Rust 反而全面小幅领先。

实现拆解

1. 先量化、再定位

作者没有直接改代码，而是先用三类手段把问题钉死：bench_serving 对比定位两类延迟；raw-socket 探针（headers + 390 KB 在一次 send 中发出，计时到响应头）把大请求体延迟复现成 208 ms；tcpi_total_retrans=3 证明是 TCP 重传而非处理慢；触发阈值二分到

75→91 KB。这决定了后续改动全部落在 TCP 层而不是应用层。

2. runtime.rs：socket 创建与缓冲配置重写

Runtime::start 中的监听建立从「std listener + 手动非阻塞」改为「tokio 原生 socket」：

项目	改动前	改动后
创建	std::net::TcpListener::bind(addr)	tokio::net::TcpSocket::new_v4() / new_v6()（按地址族）
配置	set_nonblocking(true)	set_reuseaddr(true) + set_recv_buffer_size(16 MiB)（bind 前）
bind	同步，失败即启动错误	保持同步，Err 仍以 String 返回
listen	隐含在 bind 后	推迟到 api runtime，socket.listen(1024)

关键权衡：bind 仍是硬启动错误（端口被占直接失败），listen 变成软错误（失败仅日志）。16 MiB 缓冲是「请求值」，最终受 net.core.rmem_max 钳制，且按需惰性分配。顺带移除了 libc 依赖。

3. api_server.rs：开启 TCP_NODELAY

serve 签名改为接收 tokio::net::TcpSocket，内部先 socket.listen(1024)，再通过 axum::serve::ListenerExt::tap_io 对每个 accept 出的连接执行 set_nodelay(true)（失败仅 tracing::debug!）。tap_io 是钩子式接口，无需改动 axum serve 主流程，是改动面很小的原因。

4. 验证与配套

无新增自动化测试（性能改动以手动基准验证为主）。作者对重做后的 build 重新验证了三类签名：raw 390 KB 上传 0.4 ms、keep-alive in=1 ×128 平均 27.5 ms、bench 16K/64K 159/1047 ms，与改前一致；并书面论证无需调 SO_SNDBUF（两个故障都不在发送路径）。

rust/sglang-server/src/runtime.rs

核心修复 1：socket 创建从 std TcpListener 改为 tokio TcpSocket，在 bind 前设置 16 MiB SO_RCVBUF 与 SO_REUSEADDR，解决大请求体超出内核初始接收缓冲（tcp_rmem[1] = 87380 B）触发 RTO 重传（约 200 ms）的问题；并移除了 libc 依赖。

```
// Runtime::start 中的 socket 建立段：保持同步绑定，
// 端口被占用（EADDRINUSE）时在 spawn 任何线程之前就以 Err 返回。
let addr = cfg.rust_server_args.http_addr;
let socket = match addr {
    std::net::SocketAddr::V4(_) => tokio::net::TcpSocket::new_v4(),
    std::net::SocketAddr::V6(_) => tokio::net::TcpSocket::new_v6(),
}
.map_err(|e| format!("socket for {addr} failed: {e}"))?;
```

```
    socket
        .set_reuseaddr(true)
        .map_err(|e| format!("set_reuseaddr failed: {e}"))?;

    // 16 MiB SO_RCVBUF：内核默认 tcp_rmem[1] 只有 87380 B，而请求头 + body
    // 常以一次突发到达，尾部段会在 hyper 轮询 body 前被内核丢弃，只能等
    // 客户端 RTO 重传（实测约 200 ms、tcpi_total_retrans=3，阈值在 75→91 KB）。
    // 该设置会被 accept 出的 socket 继承，覆盖约 1M token 请求体；内核按需
    // 惰性分配，实际生效值受 net.core.rmem_max 钳制。
    // 失败不阻断启动（软降级），服务退化为旧行为。
    if let Err(e) = socket.set_recv_buffer_size(16 * 1024 * 1024) {
        eprintln!("warning: set_recv_buffer_size on listener failed: {e}");
    }

    socket
        .bind(addr)
        .map_err(|e| format!("bind {addr} failed: {e}"))?;
    // socket 随后移交给 api runtime 线程，在 api_server::serve 里执行 listen()
```

rust/sglang-server/src/api_server.rs

核心修复 2：serve 签名从 std TcpListener 改为 tokio TcpSocket，以 socket.listen(1024) 在 api runtime 开始监听，并用 ListenerExt::tap_io 对每个已 accept 连接开启 TCP_NODELAY，消除 keep-alive 连接上约 13 ms 的 Nagle + delayed-ACK 延迟，行为对齐 Python asyncio。

```
    // serve() 现在直接接收 tokio TcpSocket：bind 已在 Runtime::start 中同步完成，
    // 端口冲突仍是硬启动错误；listen 与 accept 全部发生在 api runtime 上。
    pub async fn serve(
        socket: tokio::net::TcpSocket,
        senders: Senders,
        egress_buf: usize,
        server_args: Arc<ServerArgs>,
        egress_activity: ActivityCounter,
        shutdown: flume::Receiver<()>,
    ) {
        //  （AppState 构造与 Router 合并逻辑省略）

        // 在 api runtime 上开始监听：accept 队列上限 1024。
        // 相比改前在 runtime::start 里 std listener + set_nonblocking(true) 再
        // from_std() 采纳，现在少一层跨线程移交，创建 / 配置 / 监听语义更清晰。
        let listener = match socket.listen(1024) {
            Ok(l) => l,
            Err(e) => {
                tracing::error!(error = %e, "listen failed");
                return;
            }
        };

        // 对齐 Python asyncio：asyncio 默认开启 TCP_NODELAY，而 hyper/axum 不开。
```

```
    // 不开启时 keep-alive 连接上的小响应会陷入 Nagle 攒包与 delayed-ACK
    // 的相互等待，实测每条请求多约 13 ms（keep-alive ×128：40.9 → 26.4 ms）。
    // tap_io 在每个 accept 出的连接上执行闭包；失败仅记 debug 日志，
    // 服务以未优化的 TCP 行为继续运行，不阻断启动。
    use axum::serve::ListenerExt;
    let listener = listener.tap_io(|io| {
        if let Err(e) = io.set_nodelay(true) {
            tracing::debug!(error = %e, "set_nodelay failed");
        }
    });

    // with_connect_info 向 access-log 中间件暴露对端地址
    let serve = axum::serve(
        listener,
        app.into_make_service_with_connect_info::<std::net::SocketAddr>(),
    );
    // 下接 tokio::select!：serve 退出或 shutdown 信号到达时停止 accept（略）
}
```

评论区精华

该 PR 没有独立 inline review 评论（reviewer rainj-me 直接 APPROVED），讨论要点通过提交迭代与作者评论表达。能重建的三条：

> "replaced the libc setsockopt with the suggested tokio::net::TcpSocket pattern — set_reuseaddr(true) + set_recv_buffer_size(16 MiB) before bind(); listen(1024) now happens on the api runtime." —— sherlockwu 回应 review 的修改说明
>
> "Both measured issues are on the receive/write-latency path — the request-body burst overflowing the receive buffer, and Nagle on small response writes; SO_SNDBUF affects neither." —— sherlockwu 论证无需调发送缓冲

作者还强调「Bind stays in runtime::start so a port conflict is still a hard startup error」，这是对启动语义的有意保留；两处失败路径都降级为警告 / 调试日志，避免在受限容器中阻断启动——权衡点是「配置失败 → 服务以未优化行为继续运行」而非直接崩溃。

风险与影响

- 接收缓冲内存放大：16 MiB SO_RCVBUF 是每连接设置，高并发 keep-alive 连接下服务端内存水位会上升；虽然作者注明内核按需惰性分配，但较大的接收窗口会让对端更快推入更多数据，建议压测观察。
- 依赖系统 net.core.rmem_max：请求值会被钳制。若部署环境未调大（许多发行版默认 212992 B），16 MiB 请求会被钳到约 200 KB，大请求体问题依旧存在，而 PR 没有提供启动时校验实际生效值的逻辑。
- listen 失败更隐蔽：listen(1024) 从启动阶段移到 api runtime，失败时只打日志返回，api server 静默不可用但进程继续运行，诊断路径变长。
- 无自动化回归测试：行为完全依赖手动基准验证，后续 axum 升级、tap_io 语义变化可能悄悄回归。

- 影响面：对 Rust server 用户是纯收益——TTFT 对齐 Python 甚至反超；无协议 /API/ 配置变化；*decode* 路径（ITL）经 *c*=256 验证不受影响。团队层面，这套「raw-socket 探针 + 内核计数 + 二分阈值」诊断方法可直接复用到其他网络疑点。

关联脉络

这是 rust/sglang-server 模块持续演进的一环：此前 PR #32981 升级 dynamo-tokenizers 到 1.7.0 并适配 JSON 键序变化，持续完善 Rust server 的依赖与行为对齐；本 PR 则补齐 TCP 层性能短板。与仓库内大范围 runtime-context 配置迁移（#33011/#33012/#33013/#33170 等）相比，本 PR 独立于配置体系，专注网络路径，两条线共同把 Rust server 推向生产可用。head 分支名 kan/rust-server-perf 也表明这是一个持续的 Rust server 性能优化系列。