PR #22736 完整报告

sgl-project/sglang

[gRPC] Native gRPC server: proto + Rust crate scaffold + server args

合并时间：2026-04-20 12:39

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/22736

执行摘要

- 一句话：为原生 Rust gRPC 服务器建立基础架构：协议定义、Rust crate 脚手架和服务器参数。
- 推荐动作：此 PR 值得精读，特别是协议定义和构建集成部分。关注点包括： 1) proto 设计的权衡和未来合并计划； 2) Rust 扩展与 Python 集成的模式； 3) 环境变量和服务器参数的管理策略。

功能与动机

根据关联 Issue #22558（RFC），现有 HTTP 和 gRPC 服务器存在序列化开销、GIL 竞争、缺乏原生流式契约等瓶颈。此 PR 旨在通过 Rust 原生实现解决这些问题，提供同时服务 HTTP 和 gRPC 的能力，减少 API 边界的性能开销。PR body 说明这是 Phase 1 的基础部分，仅建立框架，不改变运行时行为。

实现拆解

1. 定义 gRPC 服务契约：在 proto/sglang/runtime/v1/sglang.proto 中定义 25 个 RPC，覆盖 SGLang-native、OpenAI 兼容、信息控制和 admin 功能，作为服务接口的基础。
2. 构建 Rust 扩展 crate：创建 rust/sglang-grpc/ 目录，包含 src/lib.rs（暴露 start_server 和 GrpcServerHandle）、build.rs（编译 proto 文件）和 Cargo.toml（配置依赖）。
3. 集成构建到 Python wheel：在 python/pyproject.toml 中添加 setuptools-rust 扩展模块，将 Rust crate 打包到主 sglang wheel 中，扩展目标为 sglang.srt.grpc._core。
4. 添加服务器参数和环境变量：在 python/sglang/srt/environ.py 中添加 SGLANG_GRPC_PORT 和 SGLANG_ENABLE_GRPC 环境变量；在 python/sglang/srt/server_args.py 的 _handle_deprecated_args 和 check_server_args 方法中处理这些变量，包括端口验证和冲突检查。
5. 更新 CI 配置：在 .github/workflows/pr-test.yml 中添加安装 protoc 和 Rust 工具链的步骤，确保源构建时的依赖可用。

关键文件：

- rust/sglang-grpc/src/lib.rs（模块 gRPC 核心；类别 source；类型 entrypoint；符号 GrpcServerHandle, start_server, shutdown, is_alive）：Rust 扩展的核心文件，定义了 gRPC 服务器的启动、停止和状态检查接口，是后续实现的基础。

- proto/sglang/runtime/v1/sglang.proto（模块 协议定义；类别 other；类型 data-contract；符号 SglangService, SamplingParams, TextGenerateRequest, GenerateRequest）：定义了 gRPC 服务的协议接口，涵盖所有 RPC 和消息类型，是服务器功能的核心契约。
- python/sglang/srt/server_args.py（模块 服务器参数；类别 source；类型 configuration；符号 _handle_deprecated_args, check_server_args）：处理服务器参数中的 gRPC 相关配置，包括环境变量解析和验证，影响服务器启动行为。
- rust/sglang-grpc/build.rs（模块 构建脚本；类别 source；类型 infrastructure；符号 main）：构建脚本，用于编译 proto 文件生成 Rust 代码，是 gRPC 服务器构建的关键部分。
- python/sglang/srt/environ.py（模块 环境变量；类别 source；类型 configuration）：添加 gRPC 相关的环境变量定义，控制服务器启动时的 gRPC 行为。

关键符号：start_server, shutdown, is_alive, _core

关键源码片段

rust/sglang-grpc/src/lib.rs

Rust扩展的核心文件，定义了gRPC服务器的启动、停止和状态检查接口，是后续实现的基础。

```
use pyo3::prelude::*;
use std::sync::Arc;
use tokio::sync::Notify;

pub mod proto {
    tonic::include_proto!("sglang.runtime.v1"); // 引入生成的 proto 代码
}

/// 用于控制 gRPC 服务器的句柄，提供关闭和存活检查功能
#[pyclass]
pub struct GrpcServerHandle {
    shutdown: Arc<Notify>, // 用于通知服务器停止的信号
    join_handle: Option<std::thread::JoinHandle<()>>, // 后台服务器线程的句柄
}

#[pymethods]
impl GrpcServerHandle {
    /// 发送停止信号并等待服务器线程退出
    fn shutdown(&mut self) {
        self.shutdown.notify_one();
        if let Some(handle) = self.join_handle.take() {
            let _ = handle.join(); // 等待线程结束
        }
    }

    /// 检查服务器线程是否仍在运行
    fn is_alive(&self) -> bool {
        self.join_handle
            .as_ref()
            .map_or(false, |h| !h.is_finished())
```

```
    }
}

/// 启动 gRPC 服务器，绑定到指定主机和端口，返回控制句柄
#[pyfunction]
fn start_server(host: String, port: u16, runtime_handle: PyObject) ->
PyResult<GrpcServerHandle> {
    let _ = &runtime_handle; // 占位参数，将在后续 PR 中用于运行时交互
    let shutdown = Arc::new(Notify::new());
    let shutdown_clone = shutdown.clone();

    let addr_str = format!("{}:{}", host, port);
    let addr: std::net::SocketAddr = addr_str
        .parse()
        .map_err(|e| pyo3::exceptions::PyValueError::new_err(format!("Bad address: {e}")))?;

    let join_handle = std::thread::Builder::new()
        .name("grpc-server".into())
        .spawn(move || {
            let rt = tokio::runtime::Builder::new_multi_thread()
                .worker_threads(4) // 使用 4 个工作线程的 Tokio 运行时
                .enable_all()
                .build()
                .expect("Failed to build Tokio runtime");

            rt.block_on(async move {
                tracing::info!("gRPC server listening on {}", addr);
                // 占位：实际服务器逻辑将在后续 PR 添加
                shutdown_clone.notified().await; // 等待停止信号
                tracing::info!("gRPC server shutting down");
            });
        })
        .map_err(|e| pyo3::exceptions::PyRuntimeError::new_err(format!("Failed to spawn thread:
        {e}")))?;

    Ok(GrpcServerHandle {
        shutdown,
        join_handle: Some(join_handle),
    })
}

/// Python 模块入口，暴露 start_server 和 GrpcServerHandle 给 Python 端
#[pymodule]
fn _core(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(start_server, m)?)?;
    m.add_class::<GrpcServerHandle>()?;
    Ok(())
}
```

proto/sglang/runtime/v1/sglang.proto

定义了 gRPC 服务的协议接口，涵盖所有 RPC 和消息类型，是服务器功能的核心契约。

```
syntax = "proto3";
package sglang.runtime.v1;

service SglangService {
    // SGLang-native RPCs (typed proto)
    rpc TextGenerate(TextGenerateRequest) returns (stream TextGenerateResponse);
    rpc Generate(GenerateRequest) returns (stream GenerateResponse);
    rpc TextEmbed(TextEmbedRequest) returns (TextEmbedResponse);
    rpc Embed(EmbedRequest) returns (EmbedResponse);
    rpc Classify(ClassifyRequest) returns (ClassifyResponse);
    rpc Tokenize(TokenizeRequest) returns (TokenizeResponse);
    rpc Detokenize(DetokenizeRequest) returns (DetokenizeResponse);
    rpc HealthCheck(HealthCheckRequest) returns (HealthCheckResponse);
    rpc GetModelInfo(GetModelInfoRequest) returns (GetModelInfoResponse);
    rpc GetServerInfo(GetServerInfoRequest) returns (GetServerInfoResponse);
    rpc ListModels(ListModelsRequest) returns (ListModelsResponse);
    rpc GetLoad(GetLoadRequest) returns (GetLoadResponse);
    rpc Abort(AbortRequest) returns (AbortResponse);
    rpc FlushCache(FlushCacheRequest) returns (FlushCacheResponse);
    rpc PauseGeneration(PauseGenerationRequest) returns (PauseGenerationResponse);
    rpc ContinueGeneration(ContinueGenerationRequest) returns (ContinueGenerationResponse);

    // OpenAI-compatible RPCs (JSON pass-through)
    rpc ChatComplete(OpenAIRequest) returns (stream OpenAIStreamChunk);
    rpc Complete(OpenAIRequest) returns (stream OpenAIStreamChunk);
    rpc OpenAIEmbed(OpenAIRequest) returns (OpenAIResponse);
    rpc OpenAIClassify(OpenAIRequest) returns (OpenAIResponse);
    rpc Score(OpenAIRequest) returns (OpenAIResponse);
    rpc Rerank(OpenAIRequest) returns (OpenAIResponse);

    // Admin/Ops RPCs
    rpc StartProfile(StartProfileRequest) returns (StartProfileResponse);
    rpc StopProfile(StopProfileRequest) returns (StopProfileResponse);
    rpc UpdateWeightsFromDisk(UpdateWeightsRequest) returns (UpdateWeightsResponse);
}

// 采样参数，在多个 RPC 中共享
message SamplingParams {
    optional float temperature = 1;
    optional float top_p = 2;
    optional int32 top_k = 3;
    optional float min_p = 4;
    optional float frequency_penalty = 5;
    optional float presence_penalty = 6;
    optional float repetition_penalty = 7;
    optional int32 max_new_tokens = 8;
```

```
    optional int32 min_new_tokens = 9;
    repeated string stop = 10;
    repeated int32 stop_token_ids = 11;
    optional bool ignore_eos = 12;
    optional int32 n = 13;
    optional string json_schema = 14;
    optional string regex = 15;
    // 注意：根据 review，字段可能不完整，将在后续更新
}

// 文本生成请求示例
message TextGenerateRequest {
    string text = 1;
    optional SamplingParams sampling_params = 2;
    optional bool stream = 3;
    optional bool return_logprob = 4;
    optional int32 top_logprobs_num = 5;
    optional int32 logprob_start_len = 6;
    optional bool return_text_in_logprobs = 7;
    optional string rid = 8;
    optional string lora_path = 9;
    optional string routing_key = 10;
    optional int32 routed_dp_rank = 11;
    map<string, string> trace_headers = 12;
}
```

python/sglang/srt/server_args.py

处理服务器参数中的 gRPC 相关配置，包括环境变量解析和验证，影响服务器启动行为。

```
def _handle_deprecated_args(self):
    # 处理已弃用的参数
    if self.tool_call_parser in deprecated_tool_call_parsers:
        logger.warning(f"The tool_call_parser '{self.tool_call_parser}' is deprecated...")
        self.tool_call_parser = deprecated_tool_call_parsers[self.tool_call_parser]

    if self.enable_nan_detection:
        logger.warning("--enable-nan-detection is deprecated...")
        envs.SGLANG_SPEC_NAN_DETECTION.set(True)
        envs.SGLANG_SPEC_OOB_DETECTION.set(True)

    # 原生 gRPC 标志 — 目前仅通过环境变量控制，不暴露为 CLI 参数
    # 设置为实例属性（而非数据类字段），以避免在 from_cli_args 中进行 argparse 命名空间查找
    self.enable_grpc = envs.SGLANG_ENABLE_GRPC.get() # 从环境变量读取是否启用 gRPC

    grpc_port_env = envs.SGLANG_GRPC_PORT.get()
    self.grpc_port = (
        grpc_port_env if grpc_port_env is not None else self.port + 10000 # 默认端口为 HTTP 端口
        + 10000
    )
```

```
        if not (1 <= self.grpc_port <= 65535):
            raise ValueError(
                f"SGLANG_GRPC_PORT ({self.grpc_port}) must be between 1 and 65535"
            )

    def check_server_args(self):
        # 其他验证逻辑 ...
        if (
            self.enable_grpc
            and self.grpc_port is not None
            and self.grpc_port == self.port
        ):
            raise ValueError(
                f"SGLANG_GRPC_PORT ({self.grpc_port}) must differ from --port ({self.port})"
            )
        # TODO: 还需验证 grpc_port != metrics_http_port 和 grpc_port != nccl_port
        # 暂时推迟，因为 metrics_http_port 和 nccl_port 的动态默认值可能在此处尚未解析
```

评论区精华

- 协议设计冗余：merrymercy 指出 TextGenerateRequest 和 GenerateRequest 等消息类型存在冗余，建议合并；作者 alexnails 回应将在后续 PR 中更新。
- 环境变量命名：ispobock 询问 SGLANG_DISABLE_GRPC 的默认值是否应禁用以避免影响用户；alexnails 将其改为正名 SGLANG_ENABLE_GRPC 以提高清晰度。
- 构建依赖版本：ispobock 建议更新 Rust edition 到最新；alexnails 从 2021 改为 2024。
- CI 脚本优化：merrymercy 建议将安装脚本合并到单个文件中以提高维护性。
  - proto 消息类型合并与字段完整性 (design): 作者 alexnails 回应将在后续 PR 中更新，目前仅使用最小字段集。
  - 环境变量命名和默认值 (design): 环境变量改为 SGLANG_ENABLE_GRPC，默认禁用，确保向后兼容。
  - CI 脚本优化 (infra): 建议被接受，可能在未来 CI 更新中实施。

风险与影响

- 风险：
  - 依赖风险：新增 Rust 和 protoc 构建依赖，可能导致源构建失败，特别是对于没有 Rust 工具链的环境（如 JustinTong0323 在 Issue 评论中提到的）。
  - 兼容性风险：SGLANG_ENABLE_GRPC 默认禁用，但环境变量更改可能影响现有部署，需确保向后兼容。
  - 端口冲突风险：server_args.py 中添加了端口验证，但未检查与 metrics_http_port 和 nccl_port 的冲突（TODO 注释），可能引发运行时绑定错误。
  - 协议变更风险：proto 定义可能不完整（如 SamplingParams 字段缺失），影响后续实现的功能一致性。
- 影响：

- 用户影响：短期内无变化，因为 gRPC 服务器默认禁用；长期将为用户提供高性能、低延迟的 gRPC 接口，支持同时服务 HTTP 和 gRPC，改善生产部署灵活性。
- 系统影响：引入 Rust 扩展模块，增加构建复杂性和包大小，但预编译 wheel 用户无需额外工具链；运行时目前仅添加框架，不影响性能。
- 团队影响：工程师需要熟悉 Rust 和 gRPC 构建流程，为后续开发奠定基础；CI 流程调整可能增加测试时间。
- 风险标记：依赖增加，端口冲突风险，协议不完整

关联脉络

- PR #23226 [gRPC] Pass --experimental_allow_proto3_optional to protoc in build.rs: 涉及 gRPC 构建脚本的兼容性修复，与此 PR 的 Rust 构建相关。
- PR #23014 [release] install rust toolchain in main dockerfile: 涉及 Rust 工具链的安装，与此 PR 新增的 Rust 依赖管理相关。