# PR #32014 完整报告

sgl-project/sglang

*create rust workspace*

合并时间：2026-07-24 03:02

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/32014

## 执行摘要

- 一句话：创建 Rust workspace 统一管理子项目并优化构建流程
- 推荐动作：值得精读，特别是 setup.py 的自动发现机制和与 cargo metadata 的交互方式。LTO 与链接时间的权衡也是很好的参考。同时，Rust 部分的 Py 迁移展示了 pyo3 的现代化用法。

## 功能与动机

引用 PR body: 'sglang single rust workspace for rust sub projects.' 目的是将零散的 Rust 子项目统一管理，简化新扩展的添加流程，并通过优化编译选项提升运行时性能。

## 实现拆解

步骤 1：创建 Cargo workspace 在 rust/Cargo.toml 中定义 workspace，将 sglang-grpc 和 sglang-mm 列为成员，统一版本和依赖管理。

步骤 2：重写构建发现逻辑 修改 python/setup.py，增加 _cargo_workspace_metadata() 函数调用 cargo metadata 自动发现所有声明了 [package.metadata.sglang] python-module 的 crate，并将其转换为 RustExtension 对象。替代了原来手动在 pyproject.toml 声明每个扩展的方式，新扩展只需要在 crate 的 Cargo.toml 中添加 metadata 即可自动被构建。

步骤 3：标准化工具链与代码风格 新增 rust/rust-toolchain.toml 固定 Rust 版本并启用 rustfmt 和 clippy 组件。对各个 Rust 源码进行现代化改造：将所有 PyObject 引用替换为 Py<PyAny>，使用 py.detach 替代 py.allow_threads，使用 Python::attach 替代 Python::with_gil（新 pyo3 惯用法）；对 sglang-mm 添加类型别名（Patches, HashedPatches 等）提高可读性；修复 clippy 警告，如将 (h+ps-1)/ps 替换为 h.div_ceil(ps)，将索引赋值改为迭代器等。

步骤 4：配置编译优化 在 workspace 级别设置 lto = true (fat LTO), opt-level = 3, codegen-units = 1，牺牲链接时间换取运行时性能。

步骤 5：消除系统 protoc 依赖 在 rust/sglang-grpc/build.rs 中配置 protoc-bin-vendored crate，自动下载 protoc 二进制，不再要求系统安装 protoc。

步骤 6：更新 CI 脚本 修改 scripts/ci/utils/install_rustup.sh 确保在构建前安装 Rust 工具链，并处理了 MUSA 平台遗漏 Rust 的问题。

关键文件：

- python/setup.py（模块 构建脚本；类别 source；类型 dependency-wiring；符号 _cargo_workspace_metadata, _match_by_substring, _selected_rust_extensions, _discovered_rust_extensions）：核心构建脚本，改写了 Rust 扩展的发现和注册方式，是本次变更最重要的文件。
- rust/sglang-mm/src/inkling/mod.rs（模块 图像处理；类别 source；类型 core-logic；符号 Patches, HashedPatches, PyPatches, PyHashedPatches）：展示了 Rust 端的现代化改造，包括类型别名和 pyo3 API 迁移，是核心逻辑文件之一。
- rust/sglang-grpc/src/bridge.rs（模块 gRPC 桥接；类别 source；类型 core-logic；符号 make_chunk_callback, make_json_callback, mark_send_ready, notify_ready）：展示了 gRPC 桥接层的 Py 迁移和新的 pyo3 属性标记。
- rust/Cargo.toml（模块 工作空间；类别 source；类型 workspace-config）：新增的 workspace 根配置文件，定义成员和共享依赖。
- scripts/ci/utils/install_rustup.sh（模块 CI 脚本；类别 infra；类型 infrastructure）：CI 脚本更新，确保构建环境安装 Rust 工具链，支持 MUSA 等多平台。

关键符号：_cargo_workspace_metadata, _discovered_rust_extensions, _match_by_substring, _selected_rust_extensions, make_chunk_callback, make_json_callback, Patches, HashedPatches, PyPatches, PyHashedPatches, extract_tokenizer_info, resize_lanczos_rgb, scaled_dims, precompute_coeffs

## 关键源码片段

### python/setup.py

核心构建脚本，改写了 Rust 扩展的发现和注册方式，是本次变更最重要的文件。

```
# 核心函数：调用 cargo metadata 获取工作空间信息
# 返回 JSON 格式的包列表，用于后续自动发现扩展


def _cargo_workspace_metadata():
    """通过 cargo metadata 命令获取 rust/ 工作空间的包信息。"""
    manifest_path = _RUST_WORKSPACE_DIR / "Cargo.toml"
    if not manifest_path.is_file():
        raise RuntimeError(
            f"no cargo workspace at {manifest_path} (building outside a repo "
            f"checkout?); set {_BUILD_RUST_EXTS_ENV}=none to build without "
            "Rust extensions"
        )
    try:
        out = subprocess.run(
            ["cargo", "metadata", "--format-version", "1", "--no-deps",
             "--manifest-path", str(manifest_path)],
            capture_output=True, check=True, text=True,
        )
    except FileNotFoundError:
        raise RuntimeError(
            "cargo is required to discover the Rust extension modules; "
```

```
            f"set {_BUILD_RUST_EXTS_ENV}=none to skip"
        )
    except subprocess.CalledProcessError as exc:
        raise RuntimeError(f"cargo metadata failed:\n{exc.stderr}")
    return json.loads(out.stdout)


def _discovered_rust_extensions():
    """遍历每个 crate，为声明了 python-module 的 crate 创建 RustExtension。"""
    extensions = []
    for package in sorted(
        _cargo_workspace_metadata()["packages"], key=lambda p: p["name"]
    ):
        sglang_meta = (package["metadata"] or {}).get("sglang", {})
        if "python-module" not in sglang_meta:
            continue
        extensions.append(
            RustExtension(
                target=sglang_meta["python-module"],
                path=package["manifest_path"],
                binding=Binding.PyO3,
                debug=sglang_meta.get("debug"),
            )
        )
    if not extensions:
        raise RuntimeError(
            f"no crate under {_RUST_WORKSPACE_DIR} declares "
            "[package.metadata.sglang] python-module; set "
            f"{_BUILD_RUST_EXTS_ENV}=none to build without Rust extensions"
        )
    return extensions
```

## rust/sglang-mm/src/inkling/mod.rs

展示了 Rust 端的现代化改造，包括类型别名和 pyo3 API 迁移，是核心逻辑文件之一。

```
// sglang-mm 图像处理模块的关键类型别名和网格函数

/// 单个解码图像的 (高度, 宽度, u16 patch 数据)
type Patches = (usize, usize, Vec<u16>);
/// 带内容哈希的扩展元组
type HashedPatches = (usize, usize, Vec<u16>, u64);
/// 返回给 Python 的带生命周期的 numpy 数组版本
type PyPatches<'py> = (usize, usize, Bound<'py, PyArray1<u16>>);
type PyHashedPatches<'py> = (usize, usize, Bound<'py, PyArray1<u16>>, u64);

// 使用 h.div_ceil(ps) 替代 ((h + ps - 1) / ps) 以更清晰表达向上取整
pub fn grid(h: usize, w: usize, ps: usize) -> (usize, usize) {
    (h.div_ceil(ps), w / ps + 1)
}
```

```
// 在 patchify_rgb 等函数中，使用 py.detach 替代 py.allow_threads
// 这是 pyo3 0.23 推荐的解耦 GIL 的方式
fn patchify_rgb<'py>(
    py: Python<'py>,
    arr: PyReadonlyArray3<'py, u8>,
    patch_size: usize,
) -> PyResult<Bound<'py, PyArray1<u16>>> {
    // ... 省略中间处理，最后使用 py.detach 释放 GIL 再计算
    let out = py.detach(move || patchify_alloc(&data, h, w, patch_size));
    Ok(out.into_pyarray(py)) // 使用 into_pyarray 替代旧的 into_pyarray_bound
}
```

## 评论区精华

Workspace 根目录位置：mrain 询问为什么不放在项目根目录，rainj-me 回答不要污染根目录，因为还有其他 Rust 库如 sglang-gateway 等。

LTO 覆盖不一致：sherlockwu 指出 sglang-mm 原来单独设置 lto=true，现在被 workspace 统一覆盖，可能无法保留 per-crate 配置。rainj-me 解释 cargo 不允许 per package lto override，所以 grpc 也会使用同样的 LTO 设置，链接时间会增长。

tomli 依赖问题：alexnails 担心 setup.py 依赖 tomli 可能在某些环境不存在。rainj-me 改用 cargo metadata 输出 JSON 来读取，避免解析 toml，保持 setup.py 无需额外依赖。

rust-toolchain 组件：alexnails 建议添加 rustfmt 组件以匹配 pre-commit，rainj-me 同时添加了 clippy。

- Workspace 根目录位置选择 (design): 决定将 workspace 放在 rust/ 目录下，保持根目录整洁。
- LTO 设置在 workspace 层面覆盖每个 crate 不一致 (performance): 接受统一 LTO 设置，将来可考虑 lto='thin' 和增加 codegen-units。
- 使用 toml 解析器读取 pyproject.toml 的 rust-extensions (design): 使用 subprocess 调用 cargo metadata 并解析 JSON，保持 setup.py 无需额外依赖。
- rust-toolchain 组件选择 (testing): rust-toolchain.toml 包含 rustfmt 和 clippy 组件。

## 风险与影响

- 风险：
  1. 编译时间增加：启用 fat LTO 和 opt-level=3 显著增加链接时间，特别是 sglang-grpc 原先没有使用 LTO，现在编译时间可能变长。PR body 提到后续可调整为 thin LTO 和增加 codegen-units 来缓解。
  2. Rust 工具链依赖：构建过程依赖 cargo 命令，如果环境中没有安装 Rust，构建会失败并给出明确错误。CI 已更新安装脚本，但本地开发可能需要手动安装。
  3. 跨平台兼容性：protoc-bin-vendored 的下载可能在某些受限网络环境中失败，需要验证离线场景。
  4. 新扩展注册方式：自动发现机制要求 crate 必须声明 python-module metadata，如果遗漏则不会构建，但错误信息明确。

5. 代码风格强制：clippy 检查可能会暂时引入一些警告，但已解决。 - 影响：开发者：构建流程改变，需要安装 Rust 工具链；新增 Rust 子项目只需声明 metadata 即可被自动构建，无需修改 pyproject.toml。性能：生成的 .so 文件通过 LTO 和优化可能提升运行时性能，但编译时间增加。可维护性：统一 workspace 降低依赖版本管理成本，代码风格统一有利于协作。兼容性：环境变量 SGLANG_BUILD_RUST_EXTS 仍然生效，pyproject.toml 中通过 [tool.sglang] rust-extensions 过滤仍然支持，向后兼容。

- 风险标记：构建系统变更，编译时间增加，Rust 工具链依赖，跨平台兼容性需验证

## 关联脉络

- 暂无明显关联 PR