

PR #1944 完整报告

radixark/miles

Support sync worker methods on per-group executor threads

合并时间: 2026-09-04 13:23

原文链接: <http://prhub.com.cn/radixark/miles/pull/1944>

执行摘要

- 一句话: 为 RPC 同步方法引入并发组机制, 实现按组线程隔离和序列化执行。
- 推荐动作: 这个 PR 值得精读, 因为它引入了基于并发组的线程隔离机制, 设计模式优雅 (装饰器声明并发组、执行器按组分配线程池), 测试覆盖全面 (端到端测试验证序列化、并行、隔离等行为)。对于理解 RPC 服务器的并发控制有重要参考价值。

功能与动机

为了解决 RPC 服务器中同步方法可能阻塞事件循环的问题, 需要支持按并发组进行线程隔离。通过 PR 标题和实现推断, 动机是让同步方法可以在专用线程池中运行, 提高服务器的并发处理能力。

实现拆解

1. 元数据扩展: 在 `miles/utils/workers/rpc/common/metadata.py` 中引入 `rpc` 装饰器和 `_RpcConfig` 数据类, 用于声明并发组。扩展 `RpcMethodSpec` 添加 `concurrency_group` 和 `is_async` 字段, 并在 `_build_method_spec` 中解析装饰器配置, 默认并发组为 "default"。
2. 执行器改造: 修改 `miles/utils/workers/rpc/server/executor.py` 中的 `RpcCallExecutor` 类, 在初始化时根据方法规格为每个并发组创建独立的 `ThreadPoolExecutor` (线程名前缀为 `rpc-{group}`)。在 `_call_worker` 方法中, 根据 `spec.is_async` 判断: 若为异步则使用事件循环, 否则在对应的线程池中执行同步方法。
3. 服务器集成: 更新 `miles/utils/workers/rpc/server/core.py` 中的 `RpcServer`, 将 `specs` 传递给 `RpcCallExecutor`, 并在启动日志中输出并发组信息。
4. 测试覆盖: 新增 `tests/fast/utils/workers/e2e/test_concurrency.py` 文件, 通过端到端测试验证并发组行为, 包括序列化、并行执行、线程隔离等。同时更新多个现有测试文件 (如 `test_happy_path.py`、`test_app.py`) 以适配新接口, 并添加针对装饰器和并发组的单元测试。

关键文件:

- `miles/utils/workers/rpc/common/metadata.py` (模块 元数据配置; 类别 `source`; 类型 `dependency-wiring`; 符号 `rpc`, `_RpcConfig`, `RpcMethodSpec`, `_build_method_spec`): 引入 `rpc` 装饰器和并发组配置, 扩展 `RpcMethodSpec` 以支持并发组和异步标志, 是功能的核心元数据层。

- `miles/utils/workers/rpc/server/executor.py` (模块 执行器; 类别 `source`; 类型 `core-logic`; 符号 `init`, `concurrency_groups`, `_call_worker`): 核心逻辑修改, 为每个并发组创建独立的 `ThreadPoolExecutor`, 并根据方法类型 (同步 / 异步) 选择执行路径, 是并发控制的关键实现。
- `tests/fast/utils/workers/e2e/test_concurrency.py` (模块 并发测试; 类别 `test`; 类型 `test-coverage`; 符号 `TestSyncSerialization`, `TestDeclaredConcurrencyGroups`, `TestGroupIsolation`): 新增端到端测试文件, 全面覆盖并发组行为, 包括序列化、并行执行、线程隔离等, 是验证功能正确性的关键。

关键符号: `rpc`, `_RpcConfig`, `RpcCallExecutor.init`, `RpcCallExecutor._call_worker`, `collect_rpc_method_specs`, `_build_method_spec`

关键源码片段

`miles/utils/workers/rpc/common/metadata.py`

引入 `rpc` 装饰器和并发组配置, 扩展 `RpcMethodSpec` 以支持并发组和异步标志, 是功能的核心元数据层。

```
from dataclasses import dataclass
from typing import TypeVar
from collections.abc import Callable
from typing import Any
```

```
DEFAULT_CONCURRENCY_GROUP = "default" # 默认并发组, 所有未装饰的方法归入此组
_RPC_CONFIG_ATTR = "_miles_rpc_config" # 存储并发组配置的属性名
_F = TypeVar("_F", bound=Callable[..., Any])
```

```
def rpc(*, concurrency_group: str = DEFAULT_CONCURRENCY_GROUP) -> Callable[[_F], _F]:
    """装饰器, 用于声明 RPC 方法的并发组。
```

Args:

`concurrency_group`: 并发组名称, 同一组内的同步方法将序列化执行。

Returns:

装饰后的函数。

```
"""
```

```
config = _RpcConfig(concurrency_group=concurrency_group)
```

```
def decorator(fn: _F) -> _F:
```

```
    setattr(fn, _RPC_CONFIG_ATTR, config) # 将配置附加到函数上, 供后续反射读取
    return fn
```

```
return decorator
```

```
@dataclass(frozen=True)
class _RpcConfig:
```

```
concurrency_group: str
```

```
def _build_method_spec(*, worker_cls: type, name: str, fn: Callable[..., Any]) ->
'RpcMethodSpec':
    """构建 RPC 方法规格，解析并发组和异步标志。"""
    config: _RpcConfig = getattr(fn, _RPC_CONFIG_ATTR, _RpcConfig(concurrency_group=
DEFAULT_CONCURRENCY_GROUP))
    is_async = inspect.iscoroutinefunction(inspect.unwrap(fn))
    if is_async and config.concurrency_group != DEFAULT_CONCURRENCY_GROUP:
        raise TypeError(
            f"{worker_cls.__name__}.{name} is async; concurrency groups only serialize sync
methods, "
            f"so concurrency_group={config.concurrency_group!r} would be silently ignored"
        )
    # ... 其余逻辑构建 serializer
    return RpcMethodSpec(
        name=name,
        concurrency_group=config.concurrency_group,
        is_async=is_async,
        serializer=..., # 省略序列化器构建
    )
```

miles/utils/workers/rpc/server/executor.py

核心逻辑修改，为每个并发组创建独立的 `ThreadPoolExecutor`，并根据方法类型（同步 / 异步）选择执行路径，是并发控制的关键实现。

```
import asyncio
import functools
from concurrent.futures import ThreadPoolExecutor
from miles.utils.workers.rpc.common.metadata import RpcMethodSpec

class RpcCallExecutor:
    def __init__(self, *, worker: object, specs: dict[str, RpcMethodSpec]) -> None:
        self._worker = worker
        # 为每个并发组创建一个单线程池，线程名前缀为 'rpc-{group}'
        self._executors = {
            group: ThreadPoolExecutor(max_workers=1, thread_name_prefix=f"rpc-{group}")
            for group in sorted({spec.concurrency_group for spec in specs.values() if not spec.is_
async})
        }
        self._background_tasks: set[asyncio.Task[None]] = set()

    @property
    def concurrency_groups(self) -> list[str]:
        """返回所有并发组名称（排序后）。"""
        return sorted(self._executors)
```

```
async def _call_worker(self, *, spec: RpcMethodSpec, kwargs: dict[str, Any]) -> Any:
    method = getattr(self._worker, spec.name)
    if spec.is_async:
        # 异步方法直接在事件循环中运行
        return await method(**kwargs)
    else:
        # 同步方法在对应的并发组线程池中执行，避免阻塞事件循环
        return await asyncio.get_running_loop().run_in_executor(
            self._executors[spec.concurrency_group], functools.partial(method, **kwargs)
        )
```

评论区精华

无显著讨论，PR 获得批准。Review 评论表明 PR 已被审查并合并，无阻塞问题。

- 暂无高价值评论线程

风险与影响

- 风险：风险包括：1) 线程安全：使用 ThreadPoolExecutor 可能引入线程竞争，但每个组仅有一个线程，且测试覆盖了关键场景（如序列化、并行）。2) 性能影响：引入额外线程池可能增加资源使用，但通过组隔离提高了并发性，且同步方法被移至后台线程，避免了阻塞事件循环。3) 兼容性：修改了 RpcCallExecutor 的接口（需传入 specs），但 PR 中已同步更新调用点。
- 影响：对用户的影响：RPC 调用性能提升，同步方法不再阻塞事件循环，响应更及时。对系统的影响：RPC 服务器更稳定，支持更高并发和线程隔离。对团队的影响：代码结构更清晰，便于维护和扩展并发控制，为后续优化奠定基础。
- 风险标记：线程安全，并发控制

关联脉络

- PR #1949 Fix wrapper-metadata issue: 在 review 评论中被提及，可能修复了与本 PR 相关的包装器元数据问题。
- PR #1989 Add a skeleton RayWorkerManager and a wiring entry point: 属于同一重构系列，引入了工作器管理骨架，为本 PR 的并发组功能奠定基础。
- PR #2006 Launch inference engines through the inference engine worker spec: 推进了工作器规范统一化，本 PR 是该方向的延伸，聚焦于 RPC 服务器的并发控制。