

PR #1942 完整报告

radixark/miles

Implement a worker RPC server that submits a call and polls for its outcome

合并时间: 2026-09-04 13:22

原文链接: <http://prhub.com.cn/radixark/miles/pull/1942>

执行摘要

- 一句话: 实现基于“提交 - 轮询”模式的工作者远程过程调用 (RPC) 服务器核心架构。
 - 推荐动作: 该 PR 极具精读价值, 建议重点关注。
1. 架构设计典范: 它展示了一个健壮、可扩展的 RPC 服务器从零构建的全过程。从协议定义 (protocol.py)、类型安全的序列化 (serialization.py)、自动化的元数据收集 (metadata.py), 到核心逻辑分层 (core.py, executor.py, store.py), 再到与 HTTP 框架的集成 (app.py, serve_inner.py), 每一层的设计都清晰、职责明确, 是学习如何构建类似中间件的绝佳案例。
 2. 关键设计决策:
 - 提交 - 轮询模式: 选择这种模式作为基础, 牺牲了即时性但获得了可靠性和简单的客户端实现, 非常适合工作者任务可能耗时较长的场景。
 - 异步后台执行: 使用 `asyncio.create_task` 在后台执行工作者方法, 确保了 HTTP 端点的快速响应。
 - 基于 TTL 的存储清理: 通过 `first_retrieved_at` 和 TTL 平衡了结果可重用性和内存消耗。
 - 自动化 RPC 发现: 通过内省机制自动将工作者类的公共方法暴露为 RPC 端点, 减少了样板代码。
 3. 质量与测试: 测试覆盖全面且策略得当。单元测试验证核心逻辑 (如 `CallStore` 的并发和清理), 端到端测试验证完整的用户流程 (通过真实进程)。 `harness.py` 和 `e2e_worker.py` 为测试此类服务提供了可复用的工具。
 4. 演进视角: 结合 review 讨论和后续 PR 列表, 可以理解这是一个“先上车再优化”的策略。精读此 PR 能帮助理解后续一系列修复 (#1944, #1947, #1948, #1950, #1952) 所针对的具体上下文和问题根源。

建议: 工程师应重点阅读 `RpcServer`, `CallStore`, `RpcCallExecutor` 的实现, 以及 `test_store.py` 和端到端测试, 以掌握其核心机制和测试方法。技术管理者应关注其作为基础模块的架构清晰度和后续演进计划。

功能与动机

该 PR 旨在为 miles 项目构建一个健壮、可扩展的底层 RPC 通信机制。根据 PR 标题“Implement a worker RPC server that submits a call and polls for its outcome”以及文件变更 (从零新增 31 个文件), 其核心动机是建立一个可靠的“提交 - 轮询” (submit-poll) 工作模式, 使客户端能够将计算任务异步分发给工作者进程, 并可靠地获取其执行结果。这为后续

基于 RPC 的高级功能（如会话绑定、调用去重、结果缓存、取消和监控）奠定了基础。

实现拆解

1. 协议与序列化层构建：首先定义了 RPC 的数据契约和序列化机制。 - 文件：`miles/utils/workers/rpc/common/protocol.py`，定义了 `SubmitRequest`，`SubmitResponse`，`CallStatusResponse`，`HealthResponse` 等 Pydantic 模型，以及 HTTP 路径和默认超时常量。 - 文件：`miles/utils/workers/rpc/common/serialization.py`，实现了 `RpcSerializer`，负责利用 Pydantic 动态构建查询模型并编解码查询参数和结果，确保类型安全。 - 原因：分离数据结构与业务逻辑，为协议演进提供灵活性。 - 影响：为所有 RPC 通信提供统一的数据格式和验证基础。
2. 元数据收集机制：实现了对工作者类方法的自动发现与规格化。 - 文件：`miles/utils/workers/rpc/common/metadata.py`，通过 `collect_rpc_method_specs` 函数，使用内省（inspect）机制扫描工作者类的所有公共可调用方法（排除 `classmethod`、`staticmethod`、`property`），为每个方法构建一个 `RpcMethodSpec`，其中包含名称和对应的 `RpcSerializer`。 - 原因：自动化减少了为每个方法手动定义 RPC 接口的样板代码，降低了出错概率。 - 影响：`RpcServer` 可以在初始化时自动获取工作者支持的所有 RPC 方法，动态注册路由。
3. 核心服务与执行引擎：构建了处理请求和执行调用的核心组件。 - 文件：`miles/utils/workers/rpc/server/core.py`，`RpcServer` 类是核心协调者，包含 `submit_call` 和 `query_call` 方法，负责验证请求、调用存储和执行器。 - 文件：`miles/utils/workers/rpc/server/executor.py`，`RpcCallExecutor` 负责在后台异步任务（`asyncio.Task`）中调用工作者的对应方法，并捕获结果或异常，最终回调存储层。 - 原因：将请求处理、生命周期管理和实际执行分离，符合单一职责原则，便于测试和维护。 - 影响：实现了非阻塞的请求提交，工作者可以在后台执行长时任务，客户端通过轮询获取状态。
4. 调用状态存储：实现了调用生命周期的跟踪和结果的暂存。 - 文件：`miles/utils/workers/rpc/server/store.py`，`CallStore` 类管理所有进行中和已完成调用的记录（`_CallRecord`），提供 `begin`，`finish`，`wait`，`contains` 方法，并实现了基于首次检索时间（`first_retrieved_at`）的 TTL 清理机制。 - 原因：为异步的“提交 - 轮询”模式提供了状态存取中心，TTL 机制防止内存无限增长。 - 影响：支持客户端轮询等待结果，并自动清理已检索的过期调用记录。
5. FastAPI 应用与服务入口：将核心组件组装为可部署的 HTTP 服务。 - 文件：`miles/utils/workers/rpc/server/app.py`，`create_rpc_app` 工厂函数将 `RpcServer` 与 FastAPI 路由绑定，暴露 `/v1/health`、`/v1/{method_name}` 和 `/v1/calls/{call_id}` 端点。 - 文件：`miles/utils/workers/serving/serve_inner.py`，提供了独立的启动入口 `main`，解析命令行参数，加载工作者工厂，创建 RPC 应用，并通过 Uvicorn 启动服务器。 - 原因：完成了从工作者对象到 HTTP 服务的完整封装，使其可以作为独立进程或子进程运行。 - 影响：工作者可通过一个简单的命令行接口被启动并对外提供 RPC 服务。
6. 全面的测试覆盖：实现了从单元测试到端到端测试的多层次验证。 - 文件：`tests/fast/utils/workers/rpc/server/test_store.py`，`test_metadata.py`，`test_app.py` 等，对存储、元数据收集、应用逻辑进行了详尽的单元测试。 - 文件：`tests/fast/utils/workers/e2e/` 目录下的多个测试文件（如 `test_happy_path.py`，`test_protocol_raw_http.py`）和测试工具（`harness.py`，`e2e_worker.py`），通过实际启动服务器进程来验证完整的 RPC 协

议交互。 - 原因：确保新架构的正确性和稳定性，特别是对于并发场景（如 `wait` 和 `finish` 的竞态）和边界情况（如重复完成、未知调用）。 - 影响：为后续迭代和重构提供了安全网，也为如何测试此类异步服务提供了范例。

关键文件：

- `miles/utils/workers/rpc/server/core.py`（模块 RPC 核心；类别 `source`；类型 `core-logic`；符号 `RpcServer`, `init`, `submit_call`, `reject`）：RPC 服务器的核心协调者，实现了请求的提交和轮询逻辑，是连接序列化、存储和执行器的枢纽。
- `miles/utils/workers/rpc/server/store.py`（模块 调用存储；类别 `source`；类型 `core-logic`；符号 `CallStore`, `init`, `begin`, `finish`）：实现了调用生命周期状态存储和基于 TTL 的结果缓存管理，是“提交 - 轮询”模式的关键数据组件。
- `miles/utils/workers/rpc/server/executor.py`（模块 执行引擎；类别 `source`；类型 `core-logic`；符号 `RpcCallExecutor`, `init`, `start`, `_run`）：实现了在后台异步任务中执行工作者方法并管理其生命周期的引擎，是 RPC 异步执行的核心。
- `miles/utils/workers/rpc/common/serialization.py`（模块 序列化器；类别 `source`；类型 `core-logic`；符号 `RpcSerializer`, `create`, `encode_query`, `decode_query`）：实现了基于 `Pydantic` 的动态序列化器，为 RPC 方法参数和返回值提供类型安全的编解码。

关键符号：`RpcServer.init`, `RpcServer.submit_call`, `RpcServer.query_call`, `CallStore.begin`, `CallStore.finish`, `CallStore.wait`, `CallStore._purge_expired`, `RpcCallExecutor.start`, `RpcCallExecutor._run`, `RpcSerializer.create`, `collect_rpc_method_specs`, `create_rpc_app`, `main`

关键源码片段

`miles/utils/workers/rpc/server/core.py`

RPC 服务器的核心协调者，实现了请求的提交和轮询逻辑，是连接序列化、存储和执行器的枢纽。

```
from __future__ import annotations

import functools
import logging
from typing import NoReturn

from fastapi import HTTPException
from pydantic import ValidationError

from miles.utils.tracking_utils.structured_log import log_structured
from miles.utils.workers.rpc.common.metadata import collect_rpc_method_specs
from miles.utils.workers.rpc.common.protocol import CallStatusResponse, SubmitRequest, SubmitResponse
from miles.utils.workers.rpc.server.executor import RpcCallExecutor
from miles.utils.workers.rpc.server.store import CallStore

logger = logging.getLogger(__name__)
```

```

class RpcServer:
    """RPC 服务器核心，协调请求验证、调用存储和后台执行。"""

    def __init__(self, *, worker: object) -> None:
        # 自动收集工作者类的所有公共可调用方法作为 RPC 规格
        self._specs = collect_rpc_method_specs(type(worker))
        self._store = CallStore() # 管理调用生命周期和状态
        self._executor = RpcCallExecutor(worker=worker) # 负责后台执行工作者方法
        log_structured(
            logger.info,
            tag="rpc",
            op="server",
            phase="boot",
            worker=type(worker).__name__,
            methods=len(self._specs),
        )

    def submit_call(self, *, method_name: str, request: SubmitRequest) -> SubmitResponse:
        """处理调用提交请求。

        流程：验证方法 -> 解码查询参数 -> 在存储中注册 -> 启动后台执行。
        """

        def reject(*, status_code: int, reason: str, detail: str) -> NoReturn:
            # 记录拒绝原因并抛出 HTTP 异常
            log_structured(
                logger.warning,
                tag="rpc",
                op="submit",
                phase="reject",
                reason=reason,
                method=method_name,
                call=request.call_id,
                error=detail,
            )
            raise HTTPException(status_code=status_code, detail=detail)

        # 检查请求的方法名是否在已收集的规格中
        spec = self._specs.get(method_name)
        if spec is None:
            reject(status_code=404, reason="unknown_method", detail=f"unknown rpc method {method_name!r}")

        try:
            # 使用对应方法的序列化器解码并验证查询参数
            kwargs = spec.serializer.decode_query(request.query)
        except ValidationError as e:
            reject(status_code=400, reason="invalid_query", detail=str(e))

        # 在存储中注册此次调用，标记为“开始”

```

```

self._store.begin(call_id=request.call_id)

# 启动后台异步任务执行工作者方法，完成时回调 store.finish
self._executor.start(
    spec=spec,
    kwargs=kwargs,
    call_id=request.call_id,
    finish=functools.partial(self._store.finish, call_id=request.call_id),
)

return SubmitResponse()

async def query_call(self, *, call_id: str, timeout: float) -> CallStatusResponse:
    """轮询指定调用的状态。"""
    if not self._store.contains(call_id):
        log_structured(logger.warning, tag="rpc", op="poll", phase="reject", reason="unknown_
            call", call=call_id)
        raise HTTPException(status_code=404, detail=f"unknown call id {call_id!r}")

    # 等待调用完成或超时，返回当前结果（可能为 None）
    outcome = await self._store.wait(call_id=call_id, timeout=timeout)
    if outcome is None:
        return CallStatusResponse(status="pending")
    return CallStatusResponse(status=outcome.status, result=outcome.result, error=outcome.
        error)

```

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

实现了调用生命周期状态存储和基于 TTL 的结果缓存管理，是“提交 - 轮询”模式的关键数据组件。

```

from __future__ import annotations

import asyncio
import contextlib
import dataclasses
import logging
import time

from miles.utils.tracking_utils.structured_log import log_structured
from miles.utils.workers.rpc.common.protocol import CallStatusResponse

logger = logging.getLogger(__name__)

# 已检索调用结果的默认保留时间（秒）
RETRIEVED_TTL_SECONDS = 300.0

class CallStore:
    """管理所有 RPC 调用的记录，支持异步等待结果和自动清理。"""

```

```

def __init__(self, *, retrieved_ttl_seconds: float = RETRIEVED_TTL_SECONDS) -> None:
    self._retrieved_ttl_seconds = retrieved_ttl_seconds
    self._records: dict[str, _CallRecord] = {} # call_id -> _CallRecord

def begin(self, *, call_id: str) -> None:
    """注册一个新的调用。"""
    # 在注册新调用前，尝试清理已检索的过期记录
    self._purge_expired()
    # 创建一个新的调用记录，其完成事件（Event）初始为未设置
    self._records[call_id] = _CallRecord(finished_event=asyncio.Event())
    log_structured(
        logger.debug, tag="rpc", op="call_store", phase="accept", call=call_id, tracked=len(self._
        records)
    )

def finish(self, *, call_id: str, outcome: CallStatusResponse) -> None:
    """标记一个调用已完成。"""
    record = self._records[call_id]
    if record.outcome is not None:
        raise RuntimeError(f"call {call_id} finished twice") # 防止重复完成
    record.outcome = outcome
    record.finished_event.set() # 通知所有等待此调用的协程
    log_structured(logger.debug, tag="rpc", op="call_store", phase="finish", call=call_id,
    status=outcome.status)

async def wait(self, *, call_id: str, timeout: float) -> CallStatusResponse | None:
    """等待指定调用完成或超时。返回结果，若超时且未完成则返回 None。"""
    record = self._records[call_id]
    # 等待完成事件，忽略超时异常
    with contextlib.suppress(TimeoutError, asyncio.TimeoutError):
        await asyncio.wait_for(record.finished_event.wait(), timeout=timeout)
    # 如果调用已完成且这是第一次被检索，则记录检索时间（用于 TTL 计算）
    if record.outcome is not None and record.first_retrieved_at is None:
        record.first_retrieved_at = time.monotonic()
    return record.outcome

def contains(self, call_id: str) -> bool:
    return call_id in self._records

def _purge_expired(self) -> None:
    """清理已检索且超过 TTL 的记录。"""
    now = time.monotonic()
    retained = {
        call_id: record
        for call_id, record in self._records.items()
        # 保留未检索过的记录，或首次检索时间未超过 TTL 的记录
        if record.first_retrieved_at is None or now - record.first_retrieved_at <= self._retrieved_
        ttl_seconds
    }

```

```

if len(retained) != len(self._records):
    log_structured(
        logger.debug, tag="rpc", op="call_store", phase="purge", purged=len(self._records) -
        len(retained)
    )
self._records = retained

```

```

@dataclasses.dataclass
class _CallRecord:
    """单个调用的内部记录。"""
    finished_event: asyncio.Event # 用于异步通知调用完成
    outcome: CallStatusResponse | None = None # 调用结果
    first_retrieved_at: float | None = None # 结果首次被检索的时间戳（用于 TTL）

```

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

实现了在后台异步任务中执行工作者方法并管理其生命周期的引擎，是 RPC 异步执行的核心。

```

from __future__ import annotations

import asyncio
import logging
import time
import traceback
from collections.abc import Callable
from typing import Any

from miles.utils.tracking_utils.structured_log import log_structured
from miles.utils.workers.rpc.common.metadata import RpcMethodSpec
from miles.utils.workers.rpc.common.protocol import CallStatusResponse

logger = logging.getLogger(__name__)

class RpcCallExecutor:
    """RPC 调用的执行器，在后台任务中运行工作者方法。"""

    def __init__(self, *, worker: object) -> None:
        self._worker = worker
        self._background_tasks: set[asyncio.Task[None]] = set() #
        保有对活跃任务的强引用，防止被回收

    def start(self, *, spec: RpcMethodSpec, kwargs: dict[str, Any], call_id: str, finish: Callable[...,
    None]) -> None:
        """启动一个新的后台任务来执行调用。"""
        task = asyncio.create_task(self._run(spec=spec, kwargs=kwargs, call_id=call_id, finish=
        finish))
        self._background_tasks.add(task)
        # 任务完成时自动从集合中移除，避免内存泄漏

```

```

task.add_done_callback(self._background_tasks.discard)

async def _run(
    self, *, spec: RpcMethodSpec, kwargs: dict[str, Any], call_id: str, finish: Callable[..., None]
) -> None:
    """任务的实际执行逻辑：调用工作者并处理结果/异常。"""
    started_at = time.monotonic()
    log_fields = {"tag": "rpc", "op": "execute", "method": spec.name, "call": call_id}
    log_structured(logger.debug, phase="start", **log_fields)

    try:
        # 通过反射调用工作者的方法
        result = await self._call_worker(spec=spec, kwargs=kwargs)
        # 将返回值编码为可序列化的格式
        outcome = CallStatusResponse(status="success", result=spec.serializer.encode_
result(result))
        log_structured(
            logger.debug, phase="end", ok=True, **log_fields, elapsed_s=round(time.monotonic()
- started_at, 3)
        )
    except Exception as e:
        # 捕获任何异常，记录并封装为错误信息
        log_structured(logger.error, phase="end", ok=False, **log_fields, exc_info=True)
        outcome = CallStatusResponse(status="failed", error="".join(traceback.format_
exception(e)))

    # 回调存储层，更新调用状态
    finish(outcome=outcome)

async def _call_worker(self, *, spec: RpcMethodSpec, kwargs: dict[str, Any]) -> Any:
    """通过 getattr 动态调用工作者实例上的方法。"""
    return await getattr(self._worker, spec.name)(**kwargs)

```

评论区精华

本次 PR 的 review 评论非常简短，主要来自合并者 guapisolo 的总结性批准。评论内容为：

“Approved in stack context. The sync, dedup, cancellation, retention, and non-finite wire gaps are fixed by #1944/#1947/#1948/#1952/#1950 before #2176; return-type validation lands later in #2609, with all repairs retained through #2649.”

这揭示了关键的设计权衡和演进路径：

1. 已知局限性被明确列出并计划修复：当前实现被认可为可接受的基础版本，但已知存在同步执行（sync）、缺乏调用去重（dedup）、无取消机制（cancellation）、结果保留策略（retention）以及非有限浮点数传输（non-finite wire）等问题。评审结论是这些问题由后续的一系列 PR（#1944， #1947， #1948， #1952， #1950）负责修复。
2. 技术债务被主动管理：评审明确指出了当前实现的“缺口”（gaps）以及对应的修复计划。这表明团队在引入新架构时，采取了分阶段演进的策略：先建立核心“提交 - 轮询”骨架，再逐

步增强其健壮性和功能完整性。

3. 存在明确的未来迭代：评论提到了更远期的 PR #2176 和 #2609，其中 #2609 涉及返回值验证（return-type validation）。这暗示了当前的序列化机制可能尚未包含对工作者方法返回值类型的严格校验，属于未来的增强项。
 4. 未发现激烈的技术争议：由于 PR 是全新模块的引入，且作为技术栈（stack）的一部分被批准，没有出现关于设计方向或实现细节的公开争论。唯一的评论是对后续工作的规划和确认。
- PR 合并评审与技术路线确认 (design): 明确指出了当前实现的已知局限性（同步执行、缺少去重 / 取消、TTL 策略、非有限浮点数传输、缺少返回值类型验证），并确认了这些“缺口”将由后续的一系列特定 PR 修复。这确认了当前 PR 是作为基础骨架被接受，团队采取的是分阶段演进的策略。

风险与影响

- 风险：

1. 架构基础风险：作为全新的底层通信模块，RPC 服务器的架构（如异步任务管理、状态存储、序列化）将直接被后续多个核心功能（会话绑定、去重、取消等）依赖。如果基础设计存在重大缺陷，修复成本会很高。当前实现通过清晰的模块划分和全面的测试来降低此风险。
2. 内存与资源管理风险：CallStore 通过 RETRIEVED_TTL_SECONDS（默认 300 秒）来清理已检索的结果记录。如果客户端提交大量调用但不轮询或轮询后未及时消费结果，可能导致内存压力。_purge_expired 仅在每次 begin 时调用，可能在调用高峰期清理不及时。不过，TTL 机制本身提供了保护。
3. 并发安全风险：CallStore 和 RpcCallExecutor 涉及 asyncio 任务间的并发访问。例如，wait 和 finish 可能并发操作同一个 _CallRecord。当前实现依赖 asyncio.Event 进行同步，并假设单个 await 点是原子的。虽然基本逻辑正确，但在极端异步调度下可能存在竞态条件（例如，first_retrieved_at 的设置时机）。测试中已包含针对 wait 被 finish 唤醒的场景，降低了风险。
4. 序列化局限性风险：当前的 RpcSerializer 基于 Pydantic，但未处理特殊值（如 NaN/Inf）。这已被评审指出，并计划由 PR #1950（Support non-finite floats across the wire）修复。在修复前，传输特殊浮点值会导致错误。
5. 错误处理与可观测性风险：异常处理已内置于 RpcCallExecutor._run，并记录了结构化日志。但作为初始实现，错误的分类、传播和客户端可观测性（如标准化的错误码）可能不完善。这属于功能完善范畴，风险较低。

- 影响：对用户（开发者）的影响：

- 正面影响：为 miles 生态中的工作者（worker）提供了一种标准化的、可远程访问的 RPC 接口。开发者只需实现工作者类，即可通过该 RPC 服务器将其功能暴露为 HTTP 服务，极大地简化了分布式计算任务的开发和部署流程。
- 新增复杂度：引入了一套新的架构和概念（RpcServer, CallStore, RpcSerializer 等），需要团队学习和理解。但良好的模块划分和详尽的测试有助于平滑这一过程。

对系统的影响：

- 核心基础设施升级：这是 `miles.utils.workers` 模块的一个重大增强，为其添加了网络服务能力。后续的 `session`、`rollout` 等模块可以基于此构建更复杂的交互模式（如会话绑定、多轮调用）。
- 启用了新的部署模式：工作者现在可以作为独立的、带有健康检查和轮询接口的微服务部署，增加了系统架构的灵活性。

对团队的影响：

- 设立了设计范式：该 PR 展示了如何为新的底层模块编写全面的测试（单元测试、集成测试、端到端测试）和结构清晰的代码。这为团队其他成员贡献类似模块提供了参考模板。
- 明确了技术演进路径：通过 review 讨论，团队明确了从基础骨架到功能完善的迭代计划（去重、取消、TTL、序列化增强），有助于统一未来的开发方向。
- 风险标记：全新基础模块，异步状态管理，内存缓存，已知序列化缺陷

关联脉络

- PR #1952 `Expire finished calls that were never retrieved`: 直接关联，本 PR 的 `CallStore` 实现了基础的 TTL 清理逻辑，PR #1952 将其优化为更合理的“12 小时后过期”策略，是对本 PR 存储机制的增强和修复。
- PR #1950 `Support non-finite floats across the wire`: 直接关联，本 PR 的序列化器（`RpcSerializer`）基于标准 `Pydantic`，可能无法正确处理 `NaN/Inf` 等特殊浮点数。PR #1950 引入了 `_NonFiniteFloatCodec` 来修复此线上序列化问题。
- PR #1948 `Pin a client to one server process with a boot uuid`: 功能增强，本 PR 提供了基础的 RPC 服务器。PR #1948 在此基础上，通过引入启动 UUID 来绑定客户端到特定服务器进程，是会话粘性功能的实现，属于在本 PR 基础上的重要扩展。
- PR #1947 `Refuse a duplicate call id with 409 instead of running it twice`: 功能增强，本 PR 的 `submit_call` 允许重复 `call_id`（可能导致重复执行）。PR #1947 在存储层（`CallStore`）添加了去重检查，是对接口健壮性的重要改进。
- PR #1999 `Implement a worker spec for the session server`: 架构演进，本 PR 为通用工作者建立了 RPC 服务器。PR #1999 为“会话服务器”这一特殊工作者类型实现了 `Worker Spec`，是本 PR 建立的通用规范在具体场景的应用和深化。
- PR #1951 `Support worker env vars before importing the server`: 基础设施配套，本 PR 的 `serve_inner.py` 提供了一个独立的启动入口。PR #1951 对其进行了优化，支持在导入服务器模块前计算环境变量，是部署和启动流程的改进。