

PR #1946 完整报告

radixark/miles

Long-poll from the client and tolerate a blip mid-poll

合并时间: 2026-09-04 13:24

原文链接: <http://prhub.com.cn/radixark/miles/pull/1946>

执行摘要

- 一句话: 重构 RPC 客户端轮询为长轮询, 提升调用响应速度和系统健壮性。
- 推荐动作: 这是一个高质量、经过深思熟虑的重构 PR, 建议 仔细阅读。核心变更集中在 `miles/utils/workers/rpc/client/call.py` 的 `_poll_once` 方法, 其设计 (预算感知、异常分类、服务端超时夹紧) 非常精巧。配套的测试充分且具有教育意义, 尤其是 `fake_transports.py` 中测试替身的设计。该变更对系统性能和可靠性的提升是实质性的, 是 RPC 模块演进中的关键一步。

功能与动机

原有的轮询机制 (`POLL_INTERVAL_SECONDS = 0.05`) 在每次轮询后无论调用是否完成都会立即重新发起请求, 这导致了不必要的网络流量和延迟。在调用完成前, 客户端无法及时感知到状态变化, 必须等待下一次轮询周期。本次变更旨在通过长轮询实现“事件驱动”的响应, 当调用完成时服务器能主动唤醒客户端, 从而降低延迟、减少资源浪费并提升用户体验。

实现拆解

1. 核心轮询逻辑重构 (`miles/utils/workers/rpc/client/call.py`): 将 `_poll_until_done` 方法从手动循环重写为基于 `retry_until_deadline` 的声明式重试。核心逻辑移动到新增的 `_poll_once` 方法中。
 - * 在每次重试 (即每次轮询) 中, 客户端计算一个 `poll_seconds`, 它取 `DEFAULT_POLL_TIMEOUT_SECONDS` 和剩余时间 `remaining` 的较小值。
 - * 客户端向服务器发送 `GET /v1/calls/{call_id}` 请求, 并携带 `timeout` 参数 (`server_seconds`), 该值小于 `poll_seconds` 以留出客户端接收响应的余量 (`POLL_SLACK_SECONDS`)。
 - * `RpcCall.__init__` 新增断言, 强制 `call_timeout_seconds` 必须为正数且有限 (`math.isfinite`), 防止 NaN 或无穷大导致轮询循环无法结束。
 - * `_poll_once` 方法捕获 `TimeoutError` (长轮询超时) 或 `httpx.TransportError/RetryableResponseError` (传输层错误), 并将它们包装为内部异常 `_CallStillPendingError`, 触发重试。对于传输错误, 退避睡眠时间被限制在 `min(RETRY_INITIAL_DELAY_SECONDS, remaining)` 以内。
2. 服务端轮询超时上限 (`miles/utils/workers/rpc/common/protocol.py`, `miles/utils/workers/rpc/server/core.py`):
 - 在协议模块定义了 `MAX_POLL_TIMEOUT_SECONDS = 300.0`, 作为服务端允许的客户端轮询超时上限。
 - 在服务器处理轮询请求的 `query_call` 方法中, 将客户端传入的 `timeout` 参数与上限取最小值 (`min(timeout, MAX_POLL_TIMEOUT_SECONDS)`), 防止恶意或配置错误的客户

端挂起服务器资源。

3. 大量测试补充与验证：新增了多个测试文件来验证新行为。 *

tests/fast/utils/workers/rpc/client/test_call.py: 单元测试 RpcCall._poll_once 和 _poll_until_done, 验证退避逻辑、预算限制和超时处理。 *

tests/fast/utils/workers/e2e/test_polling.py: 端到端测试长轮询的时序、客户端超时、连接预算约束和边界情况。 * tests/fast/utils/workers/e2e/test_transport.py: 端到端测试连接复用、长轮询断连、并发连接和高负载场景。 * tests/fast/utils/workers/rpc/client/test_handle.py: 扩展现有测试, 增加对 PoolTimeout 重试和长轮询窗口行为的验证。 *

tests/fast/utils/workers/rpc/client/fake_transports.py: 引入 StallingPollTransport 和 PollWindowRecordingTransport 等测试替身, 用于精确控制和观测轮询行为。 *

tests/fast/utils/workers/e2e/test_protocol_raw_http.py: 补充服务端轮询超时参数的验证, 包括负值、非数值、大值夹紧等。

关键文件:

- miles/utils/workers/rpc/client/call.py (模块 RPC 客户端; 类别 source; 类型 core-logic; 符号 _CallStillPendingError, _poll_once): 核心源码变更, 实现了长轮询客户端逻辑、预算感知退避和参数校验。
- tests/fast/utils/workers/rpc/client/fake_transports.py (模块 测试替身; 类别 test; 类型 test-coverage; 符号 StallingPollTransport, PollWindowRecordingTransport): 引入了关键的测试替身, 用于精确模拟和验证长轮询的各种时序和错误场景。
- miles/utils/workers/rpc/common/protocol.py (模块 RPC 协议; 类别 source; 类型 configuration): 定义了服务端轮询超时上限 MAX_POLL_TIMEOUT_SECONDS, 是防止资源滥用的关键配置。

关键符号: RpcCall._poll_once, RpcCall._poll_until_done, RpcCall.init

关键源码片段

miles/utils/workers/rpc/client/call.py

核心源码变更, 实现了长轮询客户端逻辑、预算感知退避和参数校验。

```
# 新增内部异常, 用于在轮询循环中区分“调用挂起”与“真实错误”
```

```
class _CallStillPendingError(Exception):
```

```
    pass
```

```
async def _poll_once(self, remaining: float) -> CallStatusResponse:
```

```
    # 计算本次轮询的客户端超时时间, 取默认值和剩余预算的较小值
```

```
    poll_seconds = min(DEFAULT_POLL_TIMEOUT_SECONDS, remaining)
```

```
    # 计算传递给服务器的超时时间, 确保服务器先于客户端超时返回
```

```
    server_seconds = poll_seconds - min(POLL_SLACK_SECONDS, poll_seconds / 2)
```

```
    try:
```

```
        # 发送长轮询请求, 携带服务器超时参数
```

```
        outcome = await self._transport.request(
```

```
            "GET",
```

```
            CALL_STATUS_PATH.format(call_id=self._call_id),
```

```

        seconds=poll_seconds,
        response_model=CallStatusResponse,
        params={"timeout": server_seconds},
    )
except (TimeoutError, asyncio.TimeoutError) as e:
    # 长轮询超时，视为调用仍未完成，触发重试
    raise _CallStillPendingError(f"long poll timed out after {poll_seconds:.1f}s") from e
except (httpx.TransportError, RetryableResponseError) as e:
    # 传输层错误，限制退避时间不超过剩余预算，然后触发重试
    await asyncio.sleep(min(RETRY_INITIAL_DELAY_SECONDS, remaining))
    raise _CallStillPendingError(f"poll attempt failed: {e!r}") from e

# 如果调用状态仍是 pending，也触发重试
if outcome.status == "pending":
    raise _CallStillPendingError("call still pending")
return outcome

```

评论区精华

评审者 guapisolo 指出了两个关键问题并已解决：

1. 防止非有限超时：指出 NaN 作为 call_timeout_seconds 会导致 retry_until_deadline 无法判断超时，造成无限轮询。作者通过在 RpcCall.__init__ 中添加 assert math.isfinite(call_timeout_seconds) and call_timeout_seconds > 0 进行了修复。
2. 限制退避睡眠时间：指出传输错误发生时，硬编码的 RETRY_INITIAL_DELAY_SECONDS 睡眠可能超出剩余预算，导致调用超时延迟。作者通过将 _poll_once 中的睡眠时间改为 min(RETRY_INITIAL_DELAY_SECONDS, remaining) 解决了此问题。

这两点都体现了对边界情况和资源管理的严谨思考，确保了新机制的正确性和鲁棒性。

- 非有限超时值（如 NaN）导致的轮询循环无法终止问题 (correctness): 通过构造时断言拒绝非有限超时值，解决了潜在的无限轮询风险。
- 传输错误后退避睡眠时间可能超过剩余预算的问题 (correctness): 退避睡眠时间现在受剩余预算约束，确保了轮询循环的精确超时行为。

风险与影响

• 风险：

1. 协议兼容性风险：服务器端新增了 timeout 查询参数并引入上限 MAX_POLL_TIMEOUT_SECONDS。旧版客户端（发送无 timeout 参数的请求）仍能正常工作，因为服务端对此参数做了兼容处理。风险较低。
2. 性能风险：长轮询减少了网络往返，对性能有正面影响。但需要确认服务端对长连接的管理不会导致连接泄漏或资源占用过高。测试 test_transport.py 已覆盖连接复用和异常断连场景，表明该风险已得到考量和测试。
3. 正确性风险：新的轮询逻辑较为复杂，涉及时间预算计算、多种异常处理和重试策略。POLL_SLACK_SECONDS 的计算 (poll_seconds - min(POLL_SLACK_SECONDS, poll_seconds / 2)) 确保了 server_seconds 始终为正且小于 poll_seconds，但需确保

在所有时序下都成立。单元测试和集成测试对此有较全面的覆盖。

4. 配置风险: `DEFAULT_POLL_TIMEOUT_SECONDS` 和 `POLL_SLACK_SECONDS` 是新的配置常量。默认值 (30s, 5s) 需要在实际环境中验证是否合适。当前测试已验证其基本行为。
- 影响: 对用户的影响: 显著提升了 RPC 调用的响应速度。客户端能几乎实时地得知调用完成, 减少了用户感知到的延迟。网络波动或短暂故障下, 调用重试机制更健壮, 减少了因网络瞬时问题导致的调用失败。

对系统的影响:

- 网络: 减少了轮询产生的“心跳”流量, 降低了服务器负载。
- 服务端: 通过轮询超时上限, 防止了因客户端问题导致的资源泄露, 提升了服务稳定性。
- 客户端: 逻辑更复杂, 但封装良好, 对外接口不变。预算感知机制确保了调用超时设置的准确性。

对团队的影响: 这是一个重要的基础设施改进, 为后续更复杂的 RPC 模式 (如流式调用) 奠定了基础。配套的测试代码 (特别是 `fake_transports`) 为未来测试类似并发、时序敏感的代码提供了良好范例。

- 风险标记: 核心路径变更, 协议兼容性调整

关联脉络

- PR #1952 Expire finished calls that were never retrieved: 同属 RPC 客户端 / 服务端健壮性改进系列。PR#1952 解决服务端已完成调用记录的内存泄漏, PR#1946 解决客户端轮询机制的效率与健壮性。两者共同提升了 RPC 系统的整体可靠性。
- PR #1947 Refuse a duplicate call id with 409 instead of running it twice: 同属 RPC 协议和客户端行为的优化。PR#1947 防止重复调用, PR#1946 优化调用结果获取机制, 都是为了使 RPC 交互更加安全和高效。