

zoo-bench 性能报告

原始数据：results/zoo-framework-dev/20261011T094251Z.json

运行环境

项	值
CPU 型号	AMD EPYC 7763 64-Core Processor
逻辑核数	4
平台	Linux-x86_64
操作系统	Linux 6.17.0-1022-azure (#22-Ubuntu SMP Mon Jul 27 17:24:03 UTC 2026)
Python	CPython 3.13.16
解释器	/opt/hostedtoolcache/Python/3.13.16/x64/bin/python
GIL	default
被测框架（发行元数据）	0.11.2-beta
被测框架（模块 __version__）	0.11.2-beta（仅附注，不作版本判据）
被测框架的驱动面世代	当前代驱动面（调度内核 core + 调度模型 model，完成信号取自结果响应器）
安装来源	git 的 dev
提交	44dcbca5f392
harness	zoo-bench 0.0.1 @ 9a42be24ac43f75e2e5d7b16434f91ba3d0717b6
复现命令	zoo-bench run --framework zoo-framework @ git+https://github.com/YearsAlso/zoo-framework.git@dev

以 dist_version（发行元数据）为版本真源；module_version 仅作参考

负载

- 构成：单位工作 = JSON 序列化 + 反序列化 + 字符串拼接，重复若干次以达到目标耗时；档位由 matrix.yaml 的 body_tiers_us 声明
- 负载是替身，不是真实 trace：形状接近“取一帧数据 -> 序列化 -> 解析 -> 字符串处理”这类业务动作，但仓库内没有真实样本，结论的适用性以此为前提

测量口径

- 执行体耗时在同一次运行内埋点：由执行体在自身内部测量并回传，框架开销 = 端到端 - 它。不做跨运行减法——两次运行的状态不同（缓存、频率、调度噪声），相减引入的是系统性偏差。
- 预热不入统计：每个单元先跑 3 轮预热并丢弃，只统计随后的 30 轮正式采样；采样数等于「正式轮数 x 并发度」。
- 绝对耗时不可跨运行比较：它只用于看量级；跨版本/跨机器要看的是同一次运行内的相对量（开销占比、相对各对照方案的倍数），它们对整体快慢不敏感。

结论摘要

- 被测框架 zoo 的执行体时长达到约 2700 微秒 及以上时，其框架开销占端到端的比例降到 15% 以下；短于此档位，选用它的主要代价就是框架开销本身。
- 并发度 1、执行体 40 微秒 下，被测框架比 process_pool 快 2.31x（该并发度所测档位中最快的一档）。

- 并发度 1 下，执行体时长超过约 40 微秒 后，apscheduler 不再快于被测框架。
- 并发度 4 下，执行体时长超过约 40 微秒 后，apscheduler 不再快于被测框架。
- 并发度 16 下，执行体时长超过约 40 微秒 后，apscheduler 不再快于被测框架。
- 并发度 64 下，执行体时长超过约 40 微秒 后，apscheduler 不再快于被测框架。
- 并发度 1 下，所测档位内 asyncio_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 4 下，所测档位内 asyncio_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 16 下，所测档位内 asyncio_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 64 下，所测档位内 asyncio_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 1 下，所测档位内 bare_thread 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 4 下，所测档位内 bare_thread 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 16 下，所测档位内 bare_thread 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 64 下，所测档位内 bare_thread 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 1 下，执行体时长超过约 40 微秒 后，process_pool 不再快于被测框架。
- 并发度 4 下，所测档位内 process_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 16 下，所测档位内 process_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 64 下，所测档位内 process_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 1 下，所测档位内 thread_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 4 下，所测档位内 thread_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 16 下，所测档位内 thread_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。
- 并发度 64 下，所测档位内 thread_pool 始终快于被测框架——这类档位正是本报告的公开不利数据。

单点加速比没有选型含义；结论以“多大的执行体时长下选哪个方案”表述（同一份框架开销，在 40 微秒的任务上占 70%，在 10 ms 上只占百分之几）

开销阈值交叉点

方案	并发度	阈值	首个不高于阈值的档位（微秒）	所测档位（微秒）	说明
apscheduler	1	15.00%	2700	40, 300, 2700, 10000	该档位是所测档位中最小的满足者；真实交叉点落在它与前一档之间
apscheduler	4	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。

方案	并发度	阈值	首个不高于阈值的档位（微秒）	所测档位（微秒）	说明
apscheduler	16	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
apscheduler	64	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
asyncio_pool	1	15.00%	2700	40, 300, 2700, 10000	该档位是所测档位中最小的满足者；真实交叉点落在它与前一档之间
asyncio_pool	4	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
asyncio_pool	16	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。

方案	并发度	阈值	首个不高于阈值的档位（微秒）	所测档位（微秒）	说明
asyncio_pool	64	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
bare_thread	1	15.00%	2700	40, 300, 2700, 10000	该档位是所测档位中最小的满足者；真实交叉点落在它与前一档之间
bare_thread	4	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
bare_thread	16	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
bare_thread	64	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。

方案	并发度	閾值	首个不高于閾值的 档位（微秒）	所测档位（微秒）	说明
process_pool	1	15.00%	2700	40, 300, 2700, 10000	该档位是所测档位中最小的满足者；真实交叉点落在它与前一档之间
process_pool	4	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
process_pool	16	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
process_pool	64	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
thread_pool	1	15.00%	300	40, 300, 2700, 10000	该档位是所测档位中最小的满足者；真实交叉点落在它与前一档之间

方案	并发度	阈值	首个不高于阈值的档位（微秒）	所测档位（微秒）	说明
thread_pool	4	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
thread_pool	16	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
thread_pool	64	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
zoo	1	15.00%	2700	40, 300, 2700, 10000	该档位是所测档位中最小的满足者；真实交叉点落在它与前一档之间
zoo	4	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。

方案	并发度	阈值	首个不高于阈值的档位（微秒）	所测档位（微秒）	说明
zoo	16	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。
zoo	64	15.00%	—	40, 300, 2700, 10000	该并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值）。端到端、吞吐与执行体自报值仍然有效，故照旧给出。

相对转折点

对照方案	并发度	该方案不再更快的档位（微秒）	说明
apscheduler	1	40	该档位是所测档位中最小的满足者
apscheduler	4	40	该档位是所测档位中最小的满足者
apscheduler	16	40	该档位是所测档位中最小的满足者
apscheduler	64	40	该档位是所测档位中最小的满足者
asyncio_pool	1	—	未观测到：所测档位内该对照方案始终快于被测框架
asyncio_pool	4	—	未观测到：所测档位内该对照方案始终快于被测框架
asyncio_pool	16	—	未观测到：所测档位内该对照方案始终快于被测框架
asyncio_pool	64	—	未观测到：所测档位内该对照方案始终快于被测框架
bare_thread	1	—	未观测到：所测档位内该对照方案始终快于被测框架
bare_thread	4	—	未观测到：所测档位内该对照方案始终快于被测框架
bare_thread	16	—	未观测到：所测档位内该对照方案始终快于被测框架

对照方案	并发度	该方案不再更快的档位（微秒）	说明
bare_thread	64	—	未观测到：所测档位内该对照方案始终快于被测框架
process_pool	1	40	该档位是所测档位中最小的满足者
process_pool	4	—	未观测到：所测档位内该对照方案始终快于被测框架
process_pool	16	—	未观测到：所测档位内该对照方案始终快于被测框架
process_pool	64	—	未观测到：所测档位内该对照方案始终快于被测框架
thread_pool	1	—	未观测到：所测档位内该对照方案始终快于被测框架
thread_pool	4	—	未观测到：所测档位内该对照方案始终快于被测框架
thread_pool	16	—	未观测到：所测档位内该对照方案始终快于被测框架
thread_pool	64	—	未观测到：所测档位内该对照方案始终快于被测框架

图表

[缺图：figures/zh-overhead_ratio.svg]

仅并发度 4 的档位；比值仅在同一次运行内成立。并发度 16, 64 的开销数字已撤下（超出该机器并行能力，见口径章节），故不在图上

[缺图：figures/zh-throughput.svg]

仅执行体 40 微秒 档；吞吐仅在同一次运行内成立

[缺图：figures/zh-relative_multiple.svg]

每个所测并发度各一个面板（共 4 个）；倍数仅在同一次运行内成立

维度：延迟分位数与抖动

端到端 / 并发度；分位数与相对离散度同表给出，只给中位数的度量不合格

并发度	执行体档位（微秒）	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	151.05 微秒	287.86 微秒	142.30 微秒	145.22 微秒	349.41 微秒	75.73 微秒
1	300	410.28 微秒	553.65 微秒	385.12 微秒	388.13 微秒	590.88 微秒	313.96 微秒
1	2700	2.753 毫秒	2.987 毫秒	2.699 毫秒	2.634 毫秒	2.988 毫秒	2.667 毫秒
1	10000	9.871 毫秒	9.941 毫秒	9.474 毫秒	9.553 毫秒	9.773 毫秒	9.525 毫秒
4	40	174.59 微秒	252.70 微秒	162.47 微秒	158.14 微秒	167.32 微秒	97.54 微秒
4	300	449.42 微秒	521.58 微秒	422.43 微秒	412.54 微秒	306.62 微秒	346.42 微秒
4	2700	2.911 毫秒	2.912 毫秒	2.754 毫秒	2.683 毫秒	1.390 毫秒	2.689 毫秒
4	10000	9.949 毫秒	9.993 毫秒	9.670 毫秒	9.713 毫秒	4.653 毫秒	9.529 毫秒
16	40	177.55 微秒	229.36 微秒	133.81 微秒	171.28 微秒	156.46 微秒	90.20 微秒
16	300	466.12 微秒	505.80 微秒	396.45 微秒	423.99 微秒	275.04 微秒	345.34 微秒
16	2700	2.908 毫秒	2.947 毫秒	2.770 毫秒	2.705 毫秒	1.381 毫秒	2.740 毫秒
16	10000	10.146 毫秒	10.216 毫秒	9.917 毫秒	9.716 毫秒	4.712 毫秒	9.785 毫秒

并发度	执行体档位 (微秒)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
64	40	205.09 微秒	221.11 微秒	121.88 微秒	170.85 微秒	204.14 微秒	91.39 微秒
64	300	462.64 微秒	509.59 微秒	381.72 微秒	425.01 微秒	307.25 微秒	352.74 微秒
64	2700	2.884 毫秒	2.905 毫秒	2.779 毫秒	2.746 毫秒	1.379 毫秒	2.647 毫秒
64	10000	10.447 毫秒	10.197 毫秒	9.811 毫秒	9.532 毫秒	4.713 毫秒	10.238 毫秒

逐单元数值（分位数、离散度、各轮原始样本）见文末「附录：全部数值」。

维度：框架自身开销占比

开销 = 端到端/并发度 - 同一次运行内实测的执行体耗时（不做跨运行减法）。带 — 的格：该并发度超出这台机器的并行能力，执行体自报耗时含超订的调度等待，与该行的每任务端到端不可比，故不给开销数字（端到端与吞吐见各自的维度）

并发度	执行体档位 (微秒)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	67.93%	82.43%	67.30%	64.62%	84.22%	41.18%
1	300	28.03%	45.95%	27.77%	26.48%	49.33%	11.25%
1	2700	5.00%	9.52%	4.39%	4.23%	11.32%	1.80%
1	10000	1.63%	3.50%	1.71%	1.36%	3.65%	0.60%
4	40	72.39%	77.98%	67.61%	68.53%	43.29%	48.41%
4	300	33.16%	40.01%	26.84%	27.97%	—	15.15%
4	2700	6.07%	9.25%	5.48%	4.89%	—	3.51%
4	10000	—	—	—	—	—	—
16	40	—	—	—	—	—	—
16	300	—	—	—	—	—	—
16	2700	—	—	—	—	—	—
16	10000	—	—	—	—	—	—
64	40	—	—	—	—	—	—
64	300	—	—	—	—	—	—
64	2700	—	—	—	—	—	—
64	10000	—	—	—	—	—	—

逐单元数值（分位数、离散度、各轮原始样本）见文末「附录：全部数值」。

维度：吞吐与并发伸缩

吞吐 = 并发度 / 端到端中位数；随并发度是否继续上升即为伸缩性

并发度	执行体档位 (微秒)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
1	40	6620.4	3473.9	7027.6	6886.0	2862.0	13204.5
1	300	2437.4	1806.2	2596.6	2576.4	1692.4	3185.1
1	2700	363.2	334.8	370.4	379.6	334.6	375.0
1	10000	101.3	100.6	105.5	104.7	102.3	105.0
4	40	22910.9	15828.8	24620.6	25293.9	23906.0	41009.0
4	300	8900.3	7669.0	9468.9	9696.1	13045.3	11546.7
4	2700	1374.2	1373.4	1452.5	1491.0	2878.5	1487.4
4	10000	402.1	400.3	413.6	411.8	859.6	419.8

并发度	执行体档位 (微秒)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
16	40	90116.9	69759.2	119571.2	93414.7	102262.4	177375.4
16	300	34326.2	31632.8	40357.7	37737.0	58172.4	46331.7
16	2700	5501.6	5429.2	5775.3	5915.8	11588.2	5838.4
16	10000	1577.0	1566.2	1613.4	1646.8	3395.6	1635.2
64	40	312054.2	289446.9	525124.2	374606.5	313503.0	700311.8
64	300	138336.3	125590.4	167662.6	150584.2	208301.1	181435.9
64	2700	22191.3	22031.0	23032.2	23310.3	46410.5	24173.8
64	10000	6126.4	6276.1	6523.5	6714.2	13578.0	6251.0

逐单元数值（分位数、离散度、各轮原始样本）见文末「附录：全部数值」。

维度：开销归因

口径：把每任务端到端拆成提交侧 / 手交 / 执行体 / 回程四段，并在同一档位与并发度下与各对照方案逐段对照，回答「超出对照的部分落在哪一段」。它是诊断而非主证据：抽样一小批档位，且不与对照方案比总开销。手交为负不是错误：那表示提交调用还没返回、执行体就已经开跑——把建线程一类工作算进提交侧的方案就会这样（实测 bare_thread 手交 -343 微秒）。回程含完成信号的等待（排空返回前的最后一段），它与主测量是同一口径

状态：ok

本维度是诊断不是主证据：抽样的档位与并发度固定为一小批，不覆盖矩阵里的全部单元。并发度上界取机器的并行能力——超过它时量到的是争用；

抽样档位（微秒）：40、10000；并发度：[1, 4]；该机器并行能力：4

方案	并发度	档位（微秒）	提交侧	手交	执行体	回程	状态
apscheduler	1	40	108.63 微秒	91.69 微秒	50.72 微秒	40.59 微秒	ok
apscheduler	4	40	109.96 微秒	509.17 微秒	53.33 微秒	229.15 微秒	ok
apscheduler	1	10000	165.03 微秒	115.08 微秒	9.620 毫秒	54.23 微秒	ok
apscheduler	4	10000	124.65 微秒	10.534 毫秒	14.747 毫秒	9.563 毫秒	ok
asyncio_pool	1	40	17.86 微秒	27.78 微秒	46.74 微秒	61.41 微秒	ok
asyncio_pool	4	40	19.51 微秒	277.63 微秒	56.04 微秒	298.30 微秒	ok
asyncio_pool	1	10000	24.40 微秒	38.13 微秒	9.282 毫秒	88.21 微秒	ok
asyncio_pool	4	10000	23.24 微秒	12.312 毫秒	14.438 毫秒	9.637 毫秒	ok
bare_thread	1	40	142.97 微秒	-77.02 微秒	53.08 微秒	31.00 微秒	ok
bare_thread	4	40	156.85 微秒	-76.12 微秒	51.67 微秒	320.79 微秒	ok
bare_thread	1	10000	5.183 毫秒	-5103.28 微秒	9.413 毫秒	40.28 微秒	ok
bare_thread	4	10000	5.193 毫秒	-5086.31 微秒	14.456 毫秒	9.546 毫秒	ok
process_pool	1	40	—	—	—	—	not measurable：执行体的起止时刻读不到——它没有在本进程里运行（跨进程档位无法这样分段）

方案	并发度	档位（微秒）	提交侧	手交	执行体	回程	状态
process_pool	4	40	—	—	—	—	not measurable：执行体的起止时刻读不到——它没有在本进程里运行（跨进程档位无法这样分段）
process_pool	1	10000	—	—	—	—	not measurable：执行体的起止时刻读不到——它没有在本进程里运行（跨进程档位无法这样分段）
process_pool	4	10000	—	—	—	—	not measurable：执行体的起止时刻读不到——它没有在本进程里运行（跨进程档位无法这样分段）
thread_pool	1	40	8.43 微秒	10.97 微秒	47.27 微秒	15.89 微秒	ok
thread_pool	4	40	11.32 微秒	189.17 微秒	51.76 微秒	162.73 微秒	ok
thread_pool	1	10000	12.30 微秒	17.07 微秒	9.423 毫秒	27.84 微秒	ok
thread_pool	4	10000	13.63 微秒	10.044 毫秒	14.793 毫秒	9.433 毫秒	ok
zoo	1	40	51.28 微秒	24.88 微秒	53.98 微秒	57.64 微秒	ok
zoo	4	40	61.92 微秒	303.25 微秒	47.79 微秒	269.05 微秒	ok
zoo	1	10000	94.33 微秒	37.15 微秒	9.568 毫秒	120.28 微秒	ok
zoo	4	10000	94.83 微秒	10.189 毫秒	15.075 毫秒	9.820 毫秒	ok

超出各对照方案的部分落在哪一段

对照方案	并发度	档位（微秒）	每任务超出	主要落在	该段之差
apscheduler	1	40	-107.03 微秒	回程	17.05 微秒
asyncio_pool	1	40	34.72 微秒	提交侧	33.42 微秒
bare_thread	1	40	39.11 微秒	手交	101.91 微秒
thread_pool	1	40	106.77 微秒	提交侧	42.86 微秒
apscheduler	4	40	-249.64 微秒	回程	39.89 微秒
asyncio_pool	4	40	4.87 微秒	提交侧	42.41 微秒
bare_thread	4	40	227.67 微秒	手交	379.38 微秒
thread_pool	4	40	266.49 微秒	手交	114.08 微秒
apscheduler	1	10000	-125.56 微秒	回程	66.05 微秒
asyncio_pool	1	10000	370.28 微秒	执行体	285.17 微秒
bare_thread	1	10000	272.87 微秒	手交	5.140 毫秒
thread_pool	1	10000	337.26 微秒	执行体	144.70 微秒
apscheduler	4	10000	102.47 微秒	执行体	328.30 微秒

对照方案	并发度	档位（微秒）	每任务超出	主要落在	该段之差
asyncio_pool	4	10000	1.119 毫秒	执行体	637.18 微秒
bare_thread	4	10000	9.802 毫秒	手交	15.276 毫秒
thread_pool	4	10000	978.60 微秒	回程	387.43 微秒

用到的插桩接缝（在 harness 侧临时包装，不改框架代码）：BaseWaiter._dispatch_worker（把 worker 交给调度模型）；WorkerDispatchCore.resolve_period / resolve_phase / resolve_run_timeout（每轮策略查询）；BaseWaiter.execute_service（走一次调度轮）

插桩自身的成本（被测框架那一侧，由同一子进程里插桩前后两轮配对得到，可如实折价）：zoo 40 微秒 / 并发 1：不插桩 184.05 微秒 对插桩后 189.35 微秒，差 5.30 微秒；zoo 40 微秒 / 并发 4：不插桩 702.46 微秒 对插桩后 678.24 微秒，差 24.23 微秒（插桩后反而更快：这轮里机器漂移比插桩成本还大）；zoo 10000 微秒 / 并发 1：不插桩 9.878 毫秒 对插桩后 9.827 毫秒，差 50.79 微秒（插桩后反而更快：这轮里机器漂移比插桩成本还大）；zoo 10000 微秒 / 并发 4：不插桩 40.260 毫秒 对插桩后 40.088 毫秒，差 171.69 微秒（插桩后反而更快：这轮里机器漂移比插桩成本还大）

细分的可加性：细分各项是独立测量：它们各自的耗时间里都含自己被抢占的等待，故之和可能超过它们共同所在的提交侧（实测约 3%）。读的时候不要把它们当成一个可以相加的划分。本轮实测超出量：zoo 40 微秒 / 并发 4 超出 5.9%；zoo 10000 微秒 / 并发 4 超出 92.1%

被测框架的提交侧由什么构成

- 并发度 1、档位 40 微秒：调度轮其余（判定、锁与调度列表维护） 0.38 微秒；派发（交给调度模型，含池的入队） 21.95 微秒；每轮策略查询（周期/相位/超时各查一次配置） 20.87 微秒；提交侧其他（适配器自己的记账与完成信号状态） 8.23 微秒
- 并发度 4、档位 40 微秒：调度轮其余（判定、锁与调度列表维护） 0.00 微秒；派发（交给调度模型，含池的入队） 30.72 微秒；每轮策略查询（周期/相位/超时各查一次配置） 29.09 微秒；提交侧其他（适配器自己的记账与完成信号状态） 4.37 微秒
- 并发度 1、档位 10000 微秒：调度轮其余（判定、锁与调度列表维护） 0.92 微秒；派发（交给调度模型，含池的入队） 37.69 微秒；每轮策略查询（周期/相位/超时各查一次配置） 38.45 微秒；提交侧其他（适配器自己的记账与完成信号状态） 16.14 微秒
- 并发度 4、档位 10000 微秒：调度轮其余（判定、锁与调度列表维护） 0.00 微秒；派发（交给调度模型，含池的入队） 42.46 微秒；每轮策略查询（周期/相位/超时各查一次配置） 50.79 微秒；提交侧其他（适配器自己的记账与完成信号状态） 7.84 微秒

归因结论

- 并发度 1、执行体 40 微秒 下，被测框架整体不慢于 apscheduler（每任务 -107 微秒）：两边的时序结构不同，逐段差额有正有负，故不给「落在哪一段」。
- 并发度 1、执行体 40 微秒 下，被测框架相对 asyncio_pool 每任务多花 35 微秒，其中主要落在「提交侧」（+33 微秒）。
- 注意：该组逐段差额并非同号——「回程」为 -4 微秒，读的时候要一并看。
- 并发度 1、执行体 40 微秒 下，被测框架相对 bare_thread 每任务多花 39 微秒，其中主要落在「手交」（+102 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -92 微秒，读的时候要一并看。
- 并发度 1、执行体 40 微秒 下，被测框架相对 thread_pool 每任务多花 107 微秒，其中主要落在「提交侧」（+43 微秒）。
- 被测框架在该组的提交侧由以下部分构成（微秒）：调度轮其余（判定、锁与调度列表维护） 0、派发（交给调度模型，含池的入队） 22、每轮策略查询（周期/相位/超时各查一次配置） 21、提交侧其他（适配器自己的记账与完成信号状态） 8。

- 并发度 4、执行体 40 微秒 下，被测框架整体不慢于 apscheduler（每任务 -250 微秒）：两边的时序结构不同，逐段差额有正有负，故不给「落在哪一段」。
- 并发度 4、执行体 40 微秒 下，被测框架相对 asyncio_pool 每任务多花 5 微秒，其中主要落在「提交侧」（+42 微秒）。
- 注意：该组逐段差额并非同号——「回程」为 -29 微秒，读的时候要一并看。
- 并发度 4、执行体 40 微秒 下，被测框架相对 bare_thread 每任务多花 228 微秒，其中主要落在「手交」（+379 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -95 微秒，读的时候要一并看。
- 并发度 4、执行体 40 微秒 下，被测框架相对 thread_pool 每任务多花 266 微秒，其中主要落在「手交」（+114 微秒）。
- 注意：该组逐段差额并非同号——「执行体」为 -4 微秒，读的时候要一并看。
- 被测框架在该组的提交侧由以下部分构成（微秒）：调度轮其余（判定、锁与调度列表维护）0、派发（交给调度模型，含池的入队）31、每轮策略查询（周期/相位/超时各查一次配置）29、提交侧其他（适配器自己的记账与完成信号状态）4。
- 并发度 1、执行体 10000 微秒 下，被测框架整体不慢于 apscheduler（每任务 -126 微秒）：两边的时序结构不同，逐段差额有正有负，故不给「落在哪一段」。
- 并发度 1、执行体 10000 微秒 下，被测框架相对 asyncio_pool 每任务多花 370 微秒，其中主要落在「执行体」（+285 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -1 微秒，读的时候要一并看。
- 并发度 1、执行体 10000 微秒 下，被测框架相对 bare_thread 每任务多花 273 微秒，其中主要落在「手交」（+5140 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -5089 微秒，读的时候要一并看。
- 并发度 1、执行体 10000 微秒 下，被测框架相对 thread_pool 每任务多花 337 微秒，其中主要落在「执行体」（+145 微秒）。
- 被测框架在该组的提交侧由以下部分构成（微秒）：调度轮其余（判定、锁与调度列表维护）1、派发（交给调度模型，含池的入队）38、每轮策略查询（周期/相位/超时各查一次配置）38、提交侧其他（适配器自己的记账与完成信号状态）16。
- 并发度 4、执行体 10000 微秒 下，被测框架相对 apscheduler 每任务多花 102 微秒，其中主要落在「执行体」（+328 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -345 微秒，读的时候要一并看。
- 并发度 4、执行体 10000 微秒 下，被测框架相对 asyncio_pool 每任务多花 1119 微秒，其中主要落在「执行体」（+637 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -2123 微秒，读的时候要一并看。
- 并发度 4、执行体 10000 微秒 下，被测框架相对 bare_thread 每任务多花 9802 微秒，其中主要落在「手交」（+15276 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -5098 微秒，读的时候要一并看。
- 并发度 4、执行体 10000 微秒 下，被测框架相对 thread_pool 每任务多花 979 微秒，其中主要落在「回程」（+387 微秒）。
- 被测框架在该组的提交侧由以下部分构成（微秒）：调度轮其余（判定、锁与调度列表维护）0、派发（交给调度模型，含池的入队）42、每轮策略查询（周期/相位/超时各查一次配置）51、提交侧其他（适配器自己的记账与完成信号状态）8。

维度：调度语义的代价

口径：本维度采用内部开关对照（同一 workload 下语义全关 vs 逐个开启），不与对照方案横向比较——裸写法没有优先级/超时/重试的对应物，故只能回答“本框架的语义值多少钱”，不能回答“比裸写法贵多少”

状态：ok（结论在该次测量装着的「current」代驱动面上得出）

语义项	层级	探查方式	可测	证据 / 原因
超时	派发层	行为探查（按调度轮反复驱动，量在飞表摘除时刻）	是	已观测到超时摘除：在执行体结束之前离开在飞表（0.0122s）（执行体 0.15s、run_timeout 0.01s）；判定动作在 dispatch_core.py:472 ；生效的是观测与熔断（记录、标记不健康、摘除在飞登记、停止派发），不是终止仍在执行的 worker——CPython 无法安全中断线程
优先级	事件层	结构探查（检查执行所需的符号是否存在）	否	未复核：本项的结论是在「previous」代上得出的，把它当成本代的事实等于在报告里断言一个当前代并不具备的接线。重新推导需要单独一次探查设计（每项要有干净的开/关态才谈得上定价）
重试	事件层	结构探查（检查执行所需的符号是否存在）	否	未复核：本项的结论是在「previous」代上得出的，把它当成本代的事实等于在报告里断言一个当前代并不具备的接线。重新推导需要单独一次探查设计（每项要有干净的开/关态才谈得上定价）

本结论按被测版本得出，坐标只对该版本成立；往 matrix.yaml 追加版本后须重跑本探查

维度：原生任务线：同一被测对象的三种执行形态

口径：该线独立测量：它不进入主矩阵的透视表与交叉点，其数值也不可与其他表里的对照方案横向比较

该线结论：ok

等价性门禁：通过，15 条判据输入与 Python 参照逐字节比对（输入上限 256 字节）

所测任务 modbus_rtu.parse_response——Modbus RTU 响应帧解析：输入是一条裸 RTU ADU

响应帧（不含任何传输层封装），其长度即该档位的帧字节

规模档位 R1、R8、R125（帧字节 7、21、255）；并发度 1、4、16、64；每格预热 100 轮 + 采样 100 轮，每个并发度一个子进程（共 4 个）

规模分界（只在判定并发度 1

上判读）：原生形态更快的档位：R8、R125；更慢的档位：R1；分辨不出的档位：无。分界点：帧不超过 7 字节时原生形态不快，达到或超过 21 字节时更快

每个数值是每任务端到端（正式采样轮的中位数）——与主矩阵延迟维度同一口径

档位（微秒）	帧字节	并发度	纯解释	原生扩展	在线路由	原生扩展 / 纯解释	在线路由 / 纯解释	更快的一方
R1	7	1	189.78 微秒	192.82 微秒	239.17 微秒	1.02x	1.26x	纯解释
R1	7	4	260.82 微秒	268.30 微秒	277.48 微秒	1.03x	1.06x	纯解释
R1	7	16	255.22 微秒	264.64 微秒	275.81 微秒	1.04x	1.08x	纯解释
R1	7	64	257.22 微秒	266.34 微秒	282.83 微秒	1.04x	1.10x	未决
R8	21	1	216.56 微秒	205.28 微秒	258.46 微秒	0.95x	1.19x	原生扩展
R8	21	4	291.65 微秒	271.99 微秒	304.52 微秒	0.93x	1.04x	未决
R8	21	16	284.36 微秒	278.61 微秒	303.14 微秒	0.98x	1.07x	未决
R8	21	64	282.44 微秒	281.62 微秒	307.27 微秒	1.00x	1.09x	未决
R125	255	1	643.52 微秒	410.37 微秒	470.20 微秒	0.64x	0.73x	原生扩展
R125	255	4	698.27 微秒	495.83 微秒	515.80 微秒	0.71x	0.74x	原生扩展
R125	255	16	688.63 微秒	498.11 微秒	516.04 微秒	0.72x	0.75x	原生扩展
R125	255	64	675.87 微秒	497.24 微秒	525.11 微秒	0.74x	0.78x	原生扩展

路由结论分两层。收敛层：实际走的臂取正式采样窗口内逐臂次数的多数者，少数臂的占比必须仅靠探索率就能解释——次数是精确量、与墙钟无关，且逐轮增量一并留档，读者可复核它如何收敛。框架一次都没走过某一臂时（看含预热的累计快照），占多数的那个只是框架的默认臂、不是决策结果，报告如实标出。划算层：取同一周期内纯解释与原生每任务耗时之差（同一次采样的配对差，而不是两个各自测出的中位数相减），且无容差区间（有容差就成了「落在区间内即算平」），并只在该差的四分位区间不跨零时才给——跨零即这份口径分辨不出，此时结论如实留作未决

路由形态逐档结论（逐臂次数取正式采样窗口，探索率取框架解析后的实际值）：

- R1（7 字节）：逐臂次数 原生 1 / 解释 99；探索率 5.00%——已收敛到解释臂；与独立对照得出的划算臂一致（解释臂）
- R8（21 字节）：逐臂次数 原生 98 / 解释 2；探索率 5.00%——已收敛到原生臂；与独立对照得出的划算臂一致（原生臂）
- R125（255 字节）：逐臂次数 原生 100 / 解释 0；探索率 5.00%——规模 R125：本组 100 次决策里 python 臂一次都没被走到，故占多数的那一臂是框架的默认臂（两臂均值相等或未知时用它），而不是这一轮作出的选择。独立对照照常给出结论，这条说明它有多少依据；与独立对照得出的划算臂一致（原生臂）

本维度的口径声明：

- 本维度对照的是同一框架的三种执行形态（纯解释 / 原生扩展 / 在线路由）；数值取自调度派发层，与主矩阵 zoo 档同一层级。此处只有每任务端到端一个量——框架在这条路径上不提供单元自报耗时，本仓也不为它插桩
- 派发层自身的开销是三种形态的共同成本，故它压缩了形态之间的倍数：框架自记的逐臂耗时（见臂口径的说明）相差比端到端大得多。这里的倍数应读成「将该任务经派发层跑完」，而不是执行形态自身的加速比
- 规模分界只在最低并发度上判读：并发度超过机器实际容量时，每任务墙钟里含排队等待，那时「原生更快」与否测的就不再是执行形态（其余并发度的组照常测量与留档）

被测框架在哪些档位更快

被测框架处于优势的档位（两侧差异大于该档位的带宽）。它与「公开的不利数据」出自同一份同运行内的相对比，只是一个取更快的、一个取更慢的——两者的覆盖面相同，故谁都挑不了档位。本节的缺席不构成不合格：分不出胜负或处处更慢时它就应当是空的

对照方案	并发度	执行体档位（微秒）	差距	被测框架中位数
apscheduler	1	40	被测框架比 apscheduler 快 1.91x	151.05 微秒
process_pool	1	40	被测框架比 process_pool 快 2.31x	151.05 微秒
apscheduler	1	300	被测框架比 apscheduler 快 1.35x	410.28 微秒
process_pool	1	300	被测框架比 process_pool 快 1.44x	410.28 微秒
apscheduler	1	2700	被测框架比 apscheduler 快 1.08x	2.753 毫秒
process_pool	1	2700	被测框架比 process_pool 快 1.09x	2.753 毫秒
apscheduler	4	40	被测框架比 apscheduler 快 1.45x	174.59 微秒
apscheduler	4	300	被测框架比 apscheduler 快 1.16x	449.42 微秒
apscheduler	16	40	被测框架比 apscheduler 快 1.29x	177.55 微秒
apscheduler	16	300	被测框架比 apscheduler 快 1.09x	466.12 微秒
apscheduler	64	300	被测框架比 apscheduler 快 1.10x	462.64 微秒

公开的不利数据

被测框架处于劣势的档位（两侧差异大于该档位的带宽）。这部分缺失的报告不合格——一份只展示自己赢的报告，读者有理由认为它不可信。差异小于带宽的档位不在本节，而在「分不出胜负」那一节：那是判据的结果，不是把它们藏起来

对照方案	并发度	执行体档位（微秒）	差距	被测框架中位数
thread_pool	1	40	thread_pool 比 zoo 快 1.99x	151.05 微秒
thread_pool	1	300	thread_pool 比 zoo 快 1.31x	410.28 微秒
bare_thread	1	2700	bare_thread 比 zoo 快 1.05x	2.753 毫秒
asyncio_pool	1	10000	asyncio_pool 比 zoo 快 1.04x	9.871 毫秒
bare_thread	1	10000	bare_thread 比 zoo 快 1.03x	9.871 毫秒
thread_pool	1	10000	thread_pool 比 zoo 快 1.04x	9.871 毫秒
thread_pool	4	40	thread_pool 比 zoo 快 1.79x	174.59 微秒

对照方案	并发度	执行体档位（微秒）	差距	被测框架中位数
bare_thread	4	300	bare_thread 比 zoo 快 1.09x	449.42 微秒
process_pool	4	300	process_pool 比 zoo 快 1.47x	449.42 微秒
thread_pool	4	300	thread_pool 比 zoo 快 1.30x	449.42 微秒
asyncio_pool	4	2700	asyncio_pool 比 zoo 快 1.06x	2.911 毫秒
bare_thread	4	2700	bare_thread 比 zoo 快 1.09x	2.911 毫秒
process_pool	4	2700	process_pool 比 zoo 快 2.09x	2.911 毫秒
thread_pool	4	2700	thread_pool 比 zoo 快 1.08x	2.911 毫秒
asyncio_pool	4	10000	asyncio_pool 比 zoo 快 1.03x	9.949 毫秒
bare_thread	4	10000	bare_thread 比 zoo 快 1.02x	9.949 毫秒
process_pool	4	10000	process_pool 比 zoo 快 2.14x	9.949 毫秒
thread_pool	4	10000	thread_pool 比 zoo 快 1.04x	9.949 毫秒
asyncio_pool	16	40	asyncio_pool 比 zoo 快 1.33x	177.55 微秒
thread_pool	16	40	thread_pool 比 zoo 快 1.97x	177.55 微秒
asyncio_pool	16	300	asyncio_pool 比 zoo 快 1.18x	466.12 微秒
bare_thread	16	300	bare_thread 比 zoo 快 1.10x	466.12 微秒
process_pool	16	300	process_pool 比 zoo 快 1.69x	466.12 微秒
thread_pool	16	300	thread_pool 比 zoo 快 1.35x	466.12 微秒
asyncio_pool	16	2700	asyncio_pool 比 zoo 快 1.05x	2.908 毫秒
bare_thread	16	2700	bare_thread 比 zoo 快 1.08x	2.908 毫秒
process_pool	16	2700	process_pool 比 zoo 快 2.11x	2.908 毫秒
thread_pool	16	2700	thread_pool 比 zoo 快 1.06x	2.908 毫秒
bare_thread	16	10000	bare_thread 比 zoo 快 1.04x	10.146 毫秒
process_pool	16	10000	process_pool 比 zoo 快 2.15x	10.146 毫秒
thread_pool	16	10000	thread_pool 比 zoo 快 1.04x	10.146 毫秒
asyncio_pool	64	40	asyncio_pool 比 zoo 快 1.68x	205.09 微秒

对照方案	并发度	执行体档位（微秒）	差距	被测框架中位数
bare_thread	64	40	bare_thread 比 zoo 快 1.20x	205.09 微秒
thread_pool	64	40	thread_pool 比 zoo 快 2.24x	205.09 微秒
asyncio_pool	64	300	asyncio_pool 比 zoo 快 1.21x	462.64 微秒
bare_thread	64	300	bare_thread 比 zoo 快 1.09x	462.64 微秒
process_pool	64	300	process_pool 比 zoo 快 1.51x	462.64 微秒
thread_pool	64	300	thread_pool 比 zoo 快 1.31x	462.64 微秒
asyncio_pool	64	2700	asyncio_pool 比 zoo 快 1.04x	2.884 毫秒
bare_thread	64	2700	bare_thread 比 zoo 快 1.05x	2.884 毫秒
process_pool	64	2700	process_pool 比 zoo 快 2.09x	2.884 毫秒
thread_pool	64	2700	thread_pool 比 zoo 快 1.09x	2.884 毫秒
asyncio_pool	64	10000	asyncio_pool 比 zoo 快 1.06x	10.447 毫秒
bare_thread	64	10000	bare_thread 比 zoo 快 1.10x	10.447 毫秒
process_pool	64	10000	process_pool 比 zoo 快 2.22x	10.447 毫秒

原生任务线是独立的一条线，不与上表混读：在它的判定并发度 1 上，原生形态更慢的档位：R1；路由形态慢于纯解释形态的档位：R1、R8；两者都在「原生任务线：同一被测对象的三种执行形态」一节内与有利档位对等列出，出自同一轮同一份配对差

分不出胜负的档位

两侧分不出胜负的档位：差异小于该档位的带宽。带宽取自同一次运行内两侧各自的跨轮离散度（每任务中位数的标准差 / 中位数）之和，故它随样本变化，读者可用留档里的逐轮样本自行复核。它同样覆盖全部所测档位、逐项写明原因——本节不是藏东西的地方：把噪声级差异报成「快 1.00x」才是

判据：带宽 = 被测框架侧与对照侧各自的跨轮相对离散度之和；单侧的跨轮相对离散度 = 该单元逐轮的「每任务中位数」的标准差 / 中位数

本期带宽：1.3% 到 50.0%（随各档位的离散度不同而不同）。

对照方案	并发度	执行体档位（微秒）	比值	原因
asyncio_pool	1	40	0.94x	两侧差异 5.8% 小于带宽 19.9%，分不出胜负
bare_thread	1	40	0.96x	两侧差异 3.9% 小于带宽 19.5%，分不出胜负
asyncio_pool	1	300	0.94x	两侧差异 6.1% 小于带宽 7.9%，分不出胜负

对照方案	并发度	执行体档位（微秒）	比值	原因
bare_thread	1	300	0.95x	两侧差异 5.4% 小于带宽 7.7%，分不出胜负
asyncio_pool	1	2700	0.98x	两侧差异 2.0% 小于带宽 3.2%，分不出胜负
thread_pool	1	2700	0.97x	两侧差异 3.2% 小于带宽 3.4%，分不出胜负
apscheduler	1	10000	1.01x	两侧差异 0.7% 小于带宽 1.3%，分不出胜负
process_pool	1	10000	0.99x	两侧差异 1.0% 小于带宽 2.6%，分不出胜负
asyncio_pool	4	40	0.93x	两侧差异 6.9% 小于带宽 21.6%，分不出胜负
bare_thread	4	40	0.91x	两侧差异 9.4% 小于带宽 13.9%，分不出胜负
process_pool	4	40	0.96x	两侧差异 4.2% 小于带宽 31.7%，分不出胜负
asyncio_pool	4	300	0.94x	两侧差异 6.0% 小于带宽 7.6%，分不出胜负
apscheduler	4	2700	1.00x	两侧差异 0.1% 小于带宽 2.4%，分不出胜负
apscheduler	4	10000	1.00x	两侧差异 0.4% 小于带宽 1.7%，分不出胜负
bare_thread	16	40	0.96x	两侧差异 3.5% 小于带宽 14.8%，分不出胜负
process_pool	16	40	0.88x	两侧差异 11.9% 小于带宽 27.7%，分不出胜负
apscheduler	16	2700	1.01x	两侧差异 1.3% 小于带宽 2.4%，分不出胜负
apscheduler	16	10000	1.01x	两侧差异 0.7% 小于带宽 1.8%，分不出胜负
asyncio_pool	16	10000	0.98x	两侧差异 2.3% 小于带宽 3.0%，分不出胜负
apscheduler	64	40	1.08x	两侧差异 7.8% 小于带宽 14.3%，分不出胜负

对照方案	并发度	执行体档位（微秒）	比值	原因
process_pool	64	40	1.00x	两侧差异 0.5% 小于带宽 17.2%，分不出胜负
apscheduler	64	2700	1.01x	两侧差异 0.7% 小于带宽 4.1%，分不出胜负
apscheduler	64	10000	0.98x	两侧差异 2.4% 小于带宽 4.9%，分不出胜负
thread_pool	64	10000	0.98x	两侧差异 2.0% 小于带宽 4.0%，分不出胜负

口径局限与偏差来源

- 被测层级：调度派发层：直接驱动 `BaseWaiter.execute_service()`（线程池模式），不走 Master 的完整运行循环——后者的服务循环每秒一次，会把延迟量级从微秒抬到秒。每提交一个任务走一次调度轮，故该轮的全部簿记都计在这一个任务头上；Master 的用法是每轮服务多个 worker，那份开销本会摊到多个任务上
- 口径说明（apscheduler）：APScheduler 的定时作业模型，含作业登记与触发判定成本；该库会吞掉作业内异常，本档已在 drain 时重新抛出。口径说明：该库不提供作业完成通知，故完成信号由本档的 Condition 给出，每任务一次通知的开销计入其端到端
- 口径说明（asyncio_pool）：asyncio 事件循环调度同步执行体（等价于 `loop.run_in_executor`），非原生协程并发；与线程池的差别是事件循环自身开销
- 口径说明（bare_thread）：每任务派生一个 `threading.Thread`，无池化、无复用；线程创建成本计入端到端。收集执行体自报耗时时无锁的 `list.append`（GIL 下本身原子）——插桩不得给某一档加别档没有的成本，否则对照就成了插桩的对照
- 口径说明（process_pool）：ProcessPoolExecutor：执行体与返回值跨进程序列化的成本计入端到端，与多线程档位不可直接等价比较
- 口径说明（thread_pool）：concurrent.futures.ThreadPoolExecutor，池大小 = 并发度；线程复用、无任务队列语义
- 口径说明（zoo）：端到端终点为「worker 完成并在飞表注销」；框架其后的结果投递按世代分流：上一代的投递路径是坏的（dispatch 按 reactor_name 而非主题查响应器，每次调用必抛），完成信号只能取自框架内部回调；当前代该处已修好，完成信号取自框架自己的结果响应器。口径说明：完成信号由本档的 Condition 给出，每任务一次通知的开销计入其端到端；该方向的偏差使被测框架看起来更慢，即对被测框架不利而非有利
- 不给出开销数字的组：以下（适配器，并发度）
组的并发度超过这次运行所在机器的并行能力：执行体自报的耗时里含超订带来的调度等待，与「墙钟 / 并发度」不是同一件事，相减的结果没有意义（实测出现过负值，并衍生出无意义的变化倍数）。故本报告对它们不给框架开销、也不给交叉点；端到端、吞吐与执行体自报值仍然有效，原始留档里的字段也一个没删。判据取自环境自述的逻辑核数，另有一条证据性兜底：算出来的开销为负同样判为不可读——框架只会加时间不会减时间
- 未参与判定的自检项：这些单元的执行体档位偏差未参与判定（并发度高于最低档，或档位短到墙钟受调度颗粒度支配）。原始值仍在场，但那两处的墙钟不足以断定校准是否正确——读这些档位的数字时要把它算进去

绝对耗时：仅在同一次运行内部有效，不可跨运行比较

加入你自己的对照

本 harness 的对照方案是开放的：实现 `zoo_bench.adapters.BaseAdapter` 的四个方法（`setup` / `submit` / `drain` / `teardown`），用 `zoo_bench.adapters.registry.register` 登记，或在 `matrix.yaml` 的 `extra_modules` 里列出你的模块路径——无需改动本仓库任何文件。你的实现会通过同一套等价性验证（提交 N 个执行体、断言全部执行且各执行一次），不通过的不会被采信。我们主动把这条路径写进报告：一份由被测框架维护者撰写、测量被测框架的报告，读者有理由怀疑对照被写慢或口径被挑选；让任何人能提交自己的对照，是这份报告可被质疑、也可被证伪的前提。

自检

- 测量自检整体：通过
- 进程隔离：{'parent_pid': 85458, 'child_pids': [85914, 85917, 85920, 85923, 85926, 85932, 85938, 85944, 85950, 85968, 85986, 86004, 86022, 86079, 86137, 86193, 86260, 86262, 86264, 86266, 86268, 86273, 86278, 86283, 86288, 86305, 86322, 86339, 86419, 86478, 86538, 86604, 86671, 86705, 86739, 86773, 86807, 86940, 87073, 87206, 87339, 87868, 88397, 88926, 89455, 91569, 93682, 95795, 97911, 97915, 97919, 97923, 97927, 97934, 97941, 97948, 97955, 97974, 97993, 98012, 98031, 98098, 98165, 98232, 98301, 98303, 98305, 98307, 98309, 98314, 98319, 98324, 98329, 98346, 98363, 98380, 98397, 98453, 98510, 98570, 98636, 98638, 98640, 98642, 98644, 98649, 98654, 98659, 98664, 98681, 98698, 98715, 98732, 98797, 98862, 98928], 'all_distinct': True, 'none_equals_parent': True}

附录：全部数值

正文为可读性只给透视表；这里逐单元给出改动前的全部字段（分位数、离散度、执行体实测等）。数值一个不少：明细放在报告主体内，而不是外链、折叠或附件——「可查」一旦要另外去找，就等于不可查

附录：延迟分位数与抖动

方案	并发度	执行体档位（微秒）	中位数	p95	p99	相对离散度
apscheduler	1	40	287.86 微秒	333.65 微秒	337.30 微秒	0.060
apscheduler	1	300	553.65 微秒	580.37 微秒	582.89 微秒	0.031
apscheduler	1	2700	2.987 毫秒	3.023 毫秒	3.035 毫秒	0.008
apscheduler	1	10000	9.941 毫秒	10.079 毫秒	10.094 毫秒	0.007
apscheduler	4	40	252.70 微秒	263.15 微秒	300.38 微秒	0.049
apscheduler	4	300	521.58 微秒	538.80 微秒	577.86 微秒	0.028
apscheduler	4	2700	2.912 毫秒	2.962 毫秒	2.986 毫秒	0.011
apscheduler	4	10000	9.993 毫秒	10.119 毫秒	10.152 毫秒	0.008
apscheduler	16	40	229.36 微秒	269.71 微秒	274.62 微秒	0.067
apscheduler	16	300	505.80 微秒	527.94 微秒	535.19 微秒	0.028
apscheduler	16	2700	2.947 毫秒	2.982 毫秒	3.001 毫秒	0.013
apscheduler	16	10000	10.216 毫秒	10.299 毫秒	10.606 毫秒	0.011
apscheduler	64	40	221.11 微秒	248.41 微秒	274.24 微秒	0.061
apscheduler	64	300	509.59 微秒	529.92 微秒	536.65 微秒	0.025
apscheduler	64	2700	2.905 毫秒	2.962 毫秒	3.045 毫秒	0.014
apscheduler	64	10000	10.197 毫秒	10.650 毫秒	10.669 毫秒	0.025
asyncio_pool	1	40	142.30 微秒	169.08 微秒	183.35 微秒	0.089
asyncio_pool	1	300	385.12 微秒	413.86 微秒	420.57 微秒	0.038

方案	并发度	执行体档位（微秒）	中位数	p95	p99	相对离散度
asyncio_pool	1	2700	2.699 毫秒	2.736 毫秒	2.766 毫秒	0.008
asyncio_pool	1	10000	9.474 毫秒	9.637 毫秒	9.654 毫秒	0.007
asyncio_pool	4	40	162.47 微秒	183.07 微秒	190.87 微秒	0.107
asyncio_pool	4	300	422.43 微秒	446.86 微秒	454.55 微秒	0.035
asyncio_pool	4	2700	2.754 毫秒	2.798 毫秒	2.816 毫秒	0.007
asyncio_pool	4	10000	9.670 毫秒	9.780 毫秒	9.784 毫秒	0.007
asyncio_pool	16	40	133.81 微秒	149.95 微秒	153.63 微秒	0.057
asyncio_pool	16	300	396.45 微秒	440.03 微秒	451.15 微秒	0.041
asyncio_pool	16	2700	2.770 毫秒	2.814 毫秒	2.838 毫秒	0.010
asyncio_pool	16	10000	9.917 毫秒	10.414 毫秒	10.835 毫秒	0.024
asyncio_pool	64	40	121.88 微秒	125.59 微秒	126.51 微秒	0.022
asyncio_pool	64	300	381.72 微秒	389.63 微秒	391.25 微秒	0.012
asyncio_pool	64	2700	2.779 毫秒	2.814 毫秒	2.838 毫秒	0.008
asyncio_pool	64	10000	9.811 毫秒	10.103 毫秒	10.291 毫秒	0.019
bare_thread	1	40	145.22 微秒	170.23 微秒	187.14 微秒	0.085
bare_thread	1	300	388.13 微秒	414.02 微秒	417.51 微秒	0.036
bare_thread	1	2700	2.634 毫秒	2.671 毫秒	2.689 毫秒	0.008
bare_thread	1	10000	9.553 毫秒	9.678 毫秒	9.949 毫秒	0.011
bare_thread	4	40	158.14 微秒	166.34 微秒	167.15 微秒	0.030
bare_thread	4	300	412.54 微秒	430.76 微秒	437.51 微秒	0.021
bare_thread	4	2700	2.683 毫秒	2.720 毫秒	2.791 毫秒	0.011
bare_thread	4	10000	9.713 毫秒	9.847 毫秒	10.230 毫秒	0.014
bare_thread	16	40	171.28 微秒	177.34 微秒	182.74 微秒	0.021
bare_thread	16	300	423.99 微秒	433.21 微秒	435.95 微秒	0.011
bare_thread	16	2700	2.705 毫秒	2.720 毫秒	2.725 毫秒	0.004
bare_thread	16	10000	9.716 毫秒	9.841 毫秒	10.636 毫秒	0.023
bare_thread	64	40	170.85 微秒	175.30 微秒	177.98 微秒	0.017
bare_thread	64	300	425.01 微秒	440.28 微秒	446.19 微秒	0.017
bare_thread	64	2700	2.746 毫秒	2.773 毫秒	2.785 毫秒	0.005
bare_thread	64	10000	9.532 毫秒	9.573 毫秒	9.611 毫秒	0.003
process_pool	1	40	349.41 微秒	373.59 微秒	885.09 微秒	0.390
process_pool	1	300	590.88 微秒	645.66 微秒	1.134 毫秒	0.232
process_pool	1	2700	2.988 毫秒	3.035 毫秒	3.660 毫秒	0.057
process_pool	1	10000	9.773 毫秒	9.841 毫秒	10.532 毫秒	0.020
process_pool	4	40	167.32 微秒	253.53 微秒	269.90 微秒	0.208
process_pool	4	300	306.62 微秒	387.79 微秒	389.90 微秒	0.130
process_pool	4	2700	1.390 毫秒	1.534 毫秒	1.906 毫秒	0.092
process_pool	4	10000	4.653 毫秒	5.019 毫秒	5.539 毫秒	0.047
process_pool	16	40	156.46 微秒	196.85 微秒	224.41 微秒	0.150
process_pool	16	300	275.04 微秒	316.76 微秒	328.71 微秒	0.083
process_pool	16	2700	1.381 毫秒	1.503 毫秒	1.512 毫秒	0.040
process_pool	16	10000	4.712 毫秒	4.885 毫秒	5.054 毫秒	0.020
process_pool	64	40	204.14 微秒	231.60 微秒	239.49 微秒	0.090

方案	并发度	执行体档位（微秒）	中位数	p95	p99	相对离散度
process_pool	64	300	307.25 微秒	337.78 微秒	364.57 微秒	0.057
process_pool	64	2700	1.379 毫秒	1.423 毫秒	1.437 毫秒	0.014
process_pool	64	10000	4.713 毫秒	4.884 毫秒	5.023 毫秒	0.018
thread_pool	1	40	75.73 微秒	93.57 微秒	98.40 微秒	0.090
thread_pool	1	300	313.96 微秒	346.42 微秒	348.61 微秒	0.041
thread_pool	1	2700	2.667 毫秒	2.707 毫秒	2.742 毫秒	0.009
thread_pool	1	10000	9.525 毫秒	9.775 毫秒	9.801 毫秒	0.012
thread_pool	4	40	97.54 微秒	106.95 微秒	113.51 微秒	0.057
thread_pool	4	300	346.42 微秒	359.06 微秒	360.93 微秒	0.017
thread_pool	4	2700	2.689 毫秒	2.742 毫秒	2.748 毫秒	0.011
thread_pool	4	10000	9.529 毫秒	9.682 毫秒	9.693 毫秒	0.009
thread_pool	16	40	90.20 微秒	102.75 微秒	115.38 微秒	0.077
thread_pool	16	300	345.34 微秒	359.59 微秒	363.79 微秒	0.019
thread_pool	16	2700	2.740 毫秒	2.791 毫秒	2.801 毫秒	0.010
thread_pool	16	10000	9.785 毫秒	9.874 毫秒	9.995 毫秒	0.007
thread_pool	64	40	91.39 微秒	93.01 微秒	93.77 微秒	0.014
thread_pool	64	300	352.74 微秒	356.45 微秒	358.38 微秒	0.007
thread_pool	64	2700	2.647 毫秒	2.680 毫秒	2.722 毫秒	0.011
thread_pool	64	10000	10.238 毫秒	10.321 毫秒	10.366 毫秒	0.016
zoo	1	40	151.05 微秒	196.92 微秒	200.72 微秒	0.110
zoo	1	300	410.28 微秒	442.59 微秒	449.04 微秒	0.041
zoo	1	2700	2.753 毫秒	2.812 毫秒	3.016 毫秒	0.024
zoo	1	10000	9.871 毫秒	9.967 毫秒	9.991 毫秒	0.006
zoo	4	40	174.59 微秒	222.70 微秒	232.01 微秒	0.109
zoo	4	300	449.42 微秒	484.37 微秒	490.04 微秒	0.041
zoo	4	2700	2.911 毫秒	2.977 毫秒	2.995 毫秒	0.013
zoo	4	10000	9.949 毫秒	10.060 毫秒	10.074 毫秒	0.008
zoo	16	40	177.55 微秒	227.24 微秒	234.50 微秒	0.126
zoo	16	300	466.12 微秒	502.80 微秒	521.18 微秒	0.047
zoo	16	2700	2.908 毫秒	2.966 毫秒	2.977 毫秒	0.011
zoo	16	10000	10.146 毫秒	10.244 毫秒	10.263 毫秒	0.007
zoo	64	40	205.09 微秒	217.27 微秒	218.65 微秒	0.082
zoo	64	300	462.64 微秒	479.84 微秒	481.79 微秒	0.020
zoo	64	2700	2.884 毫秒	3.017 毫秒	3.174 毫秒	0.027
zoo	64	10000	10.447 毫秒	10.600 毫秒	10.612 毫秒	0.024

附录：框架自身开销占比

方案	并发度	执行体档位（微秒）	框架开销	开销占比	执行体实测
apscheduler	1	40	237.28 微秒	82.43%	50.59 微秒
apscheduler	1	300	254.40 微秒	45.95%	299.25 微秒
apscheduler	1	2700	284.31 微秒	9.52%	2.703 毫秒
apscheduler	1	10000	347.66 微秒	3.50%	9.594 毫秒

方案	并发度	执行体档位（微秒）	框架开销	开销占比	执行体实测
apscheduler	4	40	197.07 微秒	77.98%	55.63 微秒
apscheduler	4	300	208.67 微秒	40.01%	312.91 微秒
apscheduler	4	2700	269.47 微秒	9.25%	2.643 毫秒
apscheduler	4	10000	—	—	14.685 毫秒
apscheduler	16	40	—	—	53.56 微秒
apscheduler	16	300	—	—	312.07 微秒
apscheduler	16	2700	—	—	2.691 毫秒
apscheduler	16	10000	—	—	20.189 毫秒
apscheduler	64	40	—	—	50.03 微秒
apscheduler	64	300	—	—	303.93 微秒
apscheduler	64	2700	—	—	2.650 毫秒
apscheduler	64	10000	—	—	19.926 毫秒
asyncio_pool	1	40	95.76 微秒	67.30%	46.53 微秒
asyncio_pool	1	300	106.93 微秒	27.77%	278.19 微秒
asyncio_pool	1	2700	118.47 微秒	4.39%	2.581 毫秒
asyncio_pool	1	10000	161.66 微秒	1.71%	9.313 毫秒
asyncio_pool	4	40	109.84 微秒	67.61%	52.62 微秒
asyncio_pool	4	300	113.36 微秒	26.84%	309.07 微秒
asyncio_pool	4	2700	151.00 微秒	5.48%	2.603 毫秒
asyncio_pool	4	10000	—	—	13.998 毫秒
asyncio_pool	16	40	—	—	53.67 微秒
asyncio_pool	16	300	—	—	303.60 微秒
asyncio_pool	16	2700	—	—	2.649 毫秒
asyncio_pool	16	10000	—	—	23.932 毫秒
asyncio_pool	64	40	—	—	51.56 微秒
asyncio_pool	64	300	—	—	301.54 微秒
asyncio_pool	64	2700	—	—	2.653 毫秒
asyncio_pool	64	10000	—	—	25.039 毫秒
bare_thread	1	40	93.84 微秒	64.62%	51.39 微秒
bare_thread	1	300	102.79 微秒	26.48%	285.34 微秒
bare_thread	1	2700	111.50 微秒	4.23%	2.523 毫秒
bare_thread	1	10000	129.62 微秒	1.36%	9.423 毫秒
bare_thread	4	40	108.38 微秒	68.53%	49.76 微秒
bare_thread	4	300	115.37 微秒	27.97%	297.17 微秒
bare_thread	4	2700	131.26 微秒	4.89%	2.551 毫秒
bare_thread	4	10000	—	—	14.689 毫秒
bare_thread	16	40	—	—	51.26 微秒
bare_thread	16	300	—	—	293.98 微秒
bare_thread	16	2700	—	—	2.573 毫秒
bare_thread	16	10000	—	—	14.962 毫秒
bare_thread	64	40	—	—	51.28 微秒
bare_thread	64	300	—	—	296.98 微秒
bare_thread	64	2700	—	—	2.611 毫秒

方案	并发度	执行体档位（微秒）	框架开销	开销占比	执行体实测
bare_thread	64	10000	—	—	14.624 毫秒
process_pool	1	40	294.27 微秒	84.22%	55.13 微秒
process_pool	1	300	291.48 微秒	49.33%	299.40 微秒
process_pool	1	2700	338.29 微秒	11.32%	2.650 毫秒
process_pool	1	10000	356.64 微秒	3.65%	9.416 毫秒
process_pool	4	40	72.43 微秒	43.29%	94.89 微秒
process_pool	4	300	—	—	565.29 微秒
process_pool	4	2700	—	—	4.828 毫秒
process_pool	4	10000	—	—	17.843 毫秒
process_pool	16	40	—	—	91.42 微秒
process_pool	16	300	—	—	569.51 微秒
process_pool	16	2700	—	—	9.767 毫秒
process_pool	16	10000	—	—	50.238 毫秒
process_pool	64	40	—	—	93.94 微秒
process_pool	64	300	—	—	580.85 微秒
process_pool	64	2700	—	—	27.218 毫秒
process_pool	64	10000	—	—	167.379 毫秒
thread_pool	1	40	31.18 微秒	41.18%	44.55 微秒
thread_pool	1	300	35.34 微秒	11.25%	278.63 微秒
thread_pool	1	2700	48.08 微秒	1.80%	2.618 毫秒
thread_pool	1	10000	57.18 微秒	0.60%	9.467 毫秒
thread_pool	4	40	47.22 微秒	48.41%	50.32 微秒
thread_pool	4	300	52.48 微秒	15.15%	293.94 微秒
thread_pool	4	2700	94.29 微秒	3.51%	2.595 毫秒
thread_pool	4	10000	—	—	16.292 毫秒
thread_pool	16	40	—	—	47.40 微秒
thread_pool	16	300	—	—	295.01 微秒
thread_pool	16	2700	—	—	2.665 毫秒
thread_pool	16	10000	—	—	20.027 毫秒
thread_pool	64	40	—	—	47.44 微秒
thread_pool	64	300	—	—	292.76 微秒
thread_pool	64	2700	—	—	2.552 毫秒
thread_pool	64	10000	—	—	45.814 毫秒
zoo	1	40	102.60 微秒	67.93%	48.45 微秒
zoo	1	300	115.00 微秒	28.03%	295.27 微秒
zoo	1	2700	137.57 微秒	5.00%	2.616 毫秒
zoo	1	10000	160.70 微秒	1.63%	9.711 毫秒
zoo	4	40	126.39 微秒	72.39%	48.20 微秒
zoo	4	300	149.04 微秒	33.16%	300.39 微秒
zoo	4	2700	176.55 微秒	6.07%	2.734 毫秒
zoo	4	10000	—	—	15.052 毫秒
zoo	16	40	—	—	44.76 微秒
zoo	16	300	—	—	315.32 微秒

方案	并发度	执行体档位（微秒）	框架开销	开销占比	执行体实测
zoo	16	2700	—	—	2.751 毫秒
zoo	16	10000	—	—	20.587 毫秒
zoo	64	40	—	—	49.05 微秒
zoo	64	300	—	—	293.02 微秒
zoo	64	2700	—	—	2.719 毫秒
zoo	64	10000	—	—	102.369 毫秒

附录：吞吐与并发伸缩

方案	并发度	执行体档位（微秒）	吞吐（任务/秒）
apscheduler	1	40	3473.9
apscheduler	1	300	1806.2
apscheduler	1	2700	334.8
apscheduler	1	10000	100.6
apscheduler	4	40	15828.8
apscheduler	4	300	7669.0
apscheduler	4	2700	1373.4
apscheduler	4	10000	400.3
apscheduler	16	40	69759.2
apscheduler	16	300	31632.8
apscheduler	16	2700	5429.2
apscheduler	16	10000	1566.2
apscheduler	64	40	289446.9
apscheduler	64	300	125590.4
apscheduler	64	2700	22031.0
apscheduler	64	10000	6276.1
asyncio_pool	1	40	7027.6
asyncio_pool	1	300	2596.6
asyncio_pool	1	2700	370.4
asyncio_pool	1	10000	105.5
asyncio_pool	4	40	24620.6
asyncio_pool	4	300	9468.9
asyncio_pool	4	2700	1452.5
asyncio_pool	4	10000	413.6
asyncio_pool	16	40	119571.2
asyncio_pool	16	300	40357.7
asyncio_pool	16	2700	5775.3
asyncio_pool	16	10000	1613.4
asyncio_pool	64	40	525124.2
asyncio_pool	64	300	167662.6
asyncio_pool	64	2700	23032.2
asyncio_pool	64	10000	6523.5
bare_thread	1	40	6886.0
bare_thread	1	300	2576.4

方案	并发度	执行体档位（微秒）	吞吐（任务/秒）
bare_thread	1	2700	379.6
bare_thread	1	10000	104.7
bare_thread	4	40	25293.9
bare_thread	4	300	9696.1
bare_thread	4	2700	1491.0
bare_thread	4	10000	411.8
bare_thread	16	40	93414.7
bare_thread	16	300	37737.0
bare_thread	16	2700	5915.8
bare_thread	16	10000	1646.8
bare_thread	64	40	374606.5
bare_thread	64	300	150584.2
bare_thread	64	2700	23310.3
bare_thread	64	10000	6714.2
process_pool	1	40	2862.0
process_pool	1	300	1692.4
process_pool	1	2700	334.6
process_pool	1	10000	102.3
process_pool	4	40	23906.0
process_pool	4	300	13045.3
process_pool	4	2700	2878.5
process_pool	4	10000	859.6
process_pool	16	40	102262.4
process_pool	16	300	58172.4
process_pool	16	2700	11588.2
process_pool	16	10000	3395.6
process_pool	64	40	313503.0
process_pool	64	300	208301.1
process_pool	64	2700	46410.5
process_pool	64	10000	13578.0
thread_pool	1	40	13204.5
thread_pool	1	300	3185.1
thread_pool	1	2700	375.0
thread_pool	1	10000	105.0
thread_pool	4	40	41009.0
thread_pool	4	300	11546.7
thread_pool	4	2700	1487.4
thread_pool	4	10000	419.8
thread_pool	16	40	177375.4
thread_pool	16	300	46331.7
thread_pool	16	2700	5838.4
thread_pool	16	10000	1635.2
thread_pool	64	40	700311.8
thread_pool	64	300	181435.9

方案	并发度	执行体档位（微秒）	吞吐（任务/秒）
thread_pool	64	2700	24173.8
thread_pool	64	10000	6251.0
zoo	1	40	6620.4
zoo	1	300	2437.4
zoo	1	2700	363.2
zoo	1	10000	101.3
zoo	4	40	22910.9
zoo	4	300	8900.3
zoo	4	2700	1374.2
zoo	4	10000	402.1
zoo	16	40	90116.9
zoo	16	300	34326.2
zoo	16	2700	5501.6
zoo	16	10000	1577.0
zoo	64	40	312054.2
zoo	64	300	138336.3
zoo	64	2700	22191.3
zoo	64	10000	6126.4