

zoo-bench 性能报告

原始数据：results/zoo-framework-0.8.0/20261011T092340Z.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）	0.8.0
被测框架（模块 __version__）	0.8.0（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）
harness	zoo-bench 0.0.1 @ 9a42be24ac43f75e2e5d7b16434f91ba3d0717b6
复现命令	zoo-bench run --framework zoo-framework==0.8.0

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 2.12x（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，执行体时长超过约 40 微秒 后，asyncio_pool 不再快于被测框架。
- 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，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 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	阈值	首个不高于阈值的档位（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	40	该档位是所测档位中最小的满足者
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	40	该档位是所测档位中最小的满足者
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	174.06 微秒	281.24 微秒	213.36 微秒	144.76 微秒	369.75 微秒	77.28 微秒
1	300	433.22 微秒	544.52 微秒	405.28 微秒	396.70 微秒	614.59 微秒	324.43 微秒
1	2700	2.800 毫秒	2.884 毫秒	2.701 毫秒	2.680 毫秒	2.982 毫秒	2.658 毫秒
1	10000	9.885 毫秒	10.156 毫秒	9.700 毫秒	9.548 毫秒	10.087 毫秒	9.698 毫秒
4	40	200.10 微秒	255.42 微秒	156.05 微秒	168.63 微秒	191.30 微秒	96.24 微秒
4	300	457.43 微秒	516.26 微秒	407.59 微秒	412.07 微秒	341.25 微秒	352.35 微秒
4	2700	2.814 毫秒	2.983 毫秒	2.764 毫秒	2.728 毫秒	1.359 毫秒	2.711 毫秒
4	10000	10.050 毫秒	10.101 毫秒	9.786 毫秒	9.829 毫秒	4.650 毫秒	9.811 毫秒
16	40	199.72 微秒	246.53 微秒	130.72 微秒	170.14 微秒	152.53 微秒	88.97 微秒
16	300	464.10 微秒	520.81 微秒	392.41 微秒	419.21 微秒	277.10 微秒	354.95 微秒
16	2700	2.869 毫秒	2.998 毫秒	2.756 毫秒	2.682 毫秒	1.413 毫秒	2.133 毫秒
16	10000	10.104 毫秒	10.223 毫秒	9.931 毫秒	9.895 毫秒	4.698 毫秒	9.734 毫秒

并发度	执行体档位 (微秒)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
64	40	195.85 微秒	245.59 微秒	123.00 微秒	174.50 微秒	200.04 微秒	90.65 微秒
64	300	465.14 微秒	514.29 微秒	381.41 微秒	426.28 微秒	310.51 微秒	353.92 微秒
64	2700	2.881 毫秒	2.940 毫秒	2.804 毫秒	2.775 毫秒	1.375 毫秒	2.774 毫秒
64	10000	10.631 毫秒	9.958 毫秒	10.397 毫秒	9.824 毫秒	4.637 毫秒	10.331 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	71.17%	82.72%	69.72%	63.87%	84.78%	39.13%
1	300	30.28%	45.98%	26.53%	25.54%	49.47%	11.12%
1	2700	4.86%	10.26%	4.58%	4.26%	11.34%	1.81%
1	10000	1.50%	3.60%	1.58%	1.33%	3.39%	0.73%
4	40	72.44%	78.18%	65.64%	69.06%	53.06%	48.39%
4	300	33.04%	40.22%	25.72%	28.67%	—	15.31%
4	2700	7.13%	9.79%	4.02%	4.69%	—	3.67%
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	5745.3	3555.7	4686.8	6907.9	2704.5	12939.1
1	300	2308.3	1836.5	2467.4	2520.8	1627.1	3082.3
1	2700	357.2	346.7	370.2	373.2	335.3	376.2
1	10000	101.2	98.5	103.1	104.7	99.1	103.1
4	40	19989.9	15660.3	25632.9	23720.3	20909.5	41562.7
4	300	8744.6	7748.0	9813.8	9707.1	11721.5	11352.2
4	2700	1421.5	1340.8	1447.4	1466.1	2944.3	1475.5
4	10000	398.0	396.0	408.8	407.0	860.2	407.7

并发度	执行体档位 (微秒)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
16	40	80111.5	64899.7	122396.1	94038.7	104898.1	179841.5
16	300	34475.3	30721.3	40773.2	38167.2	57740.1	45076.5
16	2700	5576.0	5337.6	5805.4	5965.9	11327.3	7501.0
16	10000	1583.6	1565.0	1611.1	1617.0	3406.0	1643.7
64	40	326787.3	260593.3	520342.3	366766.2	319932.6	705985.9
64	300	137591.9	124444.1	167797.3	150135.6	206112.0	180832.2
64	2700	22213.7	21770.5	22823.4	23067.0	46544.6	23074.4
64	10000	6020.1	6427.2	6155.5	6514.5	13803.0	6194.7

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	107.12 微秒	80.51 微秒	57.45 微秒	41.85 微秒	ok
apscheduler	4	40	89.24 微秒	511.93 微秒	58.52 微秒	235.26 微秒	ok
apscheduler	1	10000	180.64 微秒	129.18 微秒	9.644 毫秒	63.22 微秒	ok
apscheduler	4	10000	128.64 微秒	10.573 毫秒	14.891 毫秒	9.882 毫秒	ok
asyncio_pool	1	40	17.11 微秒	25.55 微秒	47.21 微秒	58.41 微秒	ok
asyncio_pool	4	40	17.90 微秒	290.31 微秒	55.58 微秒	275.71 微秒	ok
asyncio_pool	1	10000	22.76 微秒	35.65 微秒	9.380 毫秒	84.75 微秒	ok
asyncio_pool	4	10000	24.43 微秒	12.321 毫秒	14.391 毫秒	9.524 毫秒	ok
bare_thread	1	40	142.34 微秒	-75.72 微秒	52.62 微秒	30.13 微秒	ok
bare_thread	4	40	149.38 微秒	-73.41 微秒	50.50 微秒	299.38 微秒	ok
bare_thread	1	10000	5.174 毫秒	-5094.15 微秒	9.410 毫秒	37.10 微秒	ok
bare_thread	4	10000	5.181 毫秒	-5083.47 微秒	14.567 毫秒	9.130 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	8.07 微秒	10.50 微秒	44.59 微秒	15.18 微秒	ok
thread_pool	4	40	11.09 微秒	187.33 微秒	50.57 微秒	155.11 微秒	ok
thread_pool	1	10000	11.95 微秒	16.89 微秒	9.546 毫秒	27.48 微秒	ok
thread_pool	4	10000	13.35 微秒	10.158 毫秒	18.653 毫秒	9.036 毫秒	ok
zoo	1	40	51.07 微秒	25.19 微秒	52.73 微秒	64.23 微秒	ok
zoo	4	40	66.95 微秒	358.54 微秒	55.25 微秒	319.43 微秒	ok
zoo	1	10000	87.78 微秒	31.70 微秒	9.652 毫秒	97.67 微秒	ok
zoo	4	10000	82.24 微秒	10.273 毫秒	14.148 毫秒	9.899 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	-99.89 微秒	回程	22.38 微秒
asyncio_pool	1	40	42.90 微秒	提交侧	33.97 微秒
bare_thread	1	40	43.72 微秒	手交	100.91 微秒
thread_pool	1	40	114.47 微秒	回程	49.05 微秒
apscheduler	4	40	-138.41 微秒	回程	84.17 微秒
asyncio_pool	4	40	78.10 微秒	手交	68.23 微秒
bare_thread	4	40	342.78 微秒	手交	431.96 微秒
thread_pool	4	40	373.11 微秒	手交	171.21 微秒
apscheduler	1	10000	-151.45 微秒	回程	34.45 微秒
asyncio_pool	1	10000	336.94 微秒	执行体	271.67 微秒
bare_thread	1	10000	341.61 微秒	手交	5.126 毫秒
thread_pool	1	10000	264.05 微秒	执行体	106.11 微秒
apscheduler	4	10000	-764.89 微秒	回程	16.48 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	434.00 微秒	回程	374.94 微秒
bare_thread	4	10000	9.711 毫秒	手交	15.357 毫秒
thread_pool	4	10000	671.38 微秒	回程	862.47 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 190.97 微秒 对插桩后 193.02 微秒，差 2.05 微秒；zoo 40 微秒 / 并发 4：不插桩 749.61 微秒 对插桩后 769.88 微秒，差 20.27 微秒；zoo 10000 微秒 / 并发 1：不插桩 9.779 毫秒 对插桩后 9.868 毫秒，差 89.14 微秒；zoo 10000 微秒 / 并发 4：不插桩 39.848 毫秒 对插桩后 39.597 毫秒，差 251.34 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 55.7%；zoo 10000 微秒 / 并发 1 超出 16.8%；zoo 10000 微秒 / 并发 4 超出 97.9%

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） 2.16 微秒；派发（交给调度模型，含池的入队） 24.60 微秒；每轮策略查询（周期/相位/超时各查一次配置） 17.00 微秒；提交侧其他（适配器自己的记账与完成信号状态） 7.30 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） 37.56 微秒；每轮策略查询（周期/相位/超时各查一次配置） 30.47 微秒；提交侧其他（适配器自己的记账与完成信号状态） 5.07 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） 3.41 微秒；派发（交给调度模型，含池的入队） 41.16 微秒；每轮策略查询（周期/相位/超时各查一次配置） 29.77 微秒；提交侧其他（适配器自己的记账与完成信号状态） 12.94 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） 46.12 微秒；每轮策略查询（周期/相位/超时各查一次配置） 34.15 微秒；提交侧其他（适配器自己的记账与完成信号状态） 7.59 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 -100 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 43 微秒，其中主要落在「提交侧」（+34 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -0 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 44 微秒，其中主要落在「手交」（+101 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -91 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 114 微秒，其中主要落在「回程」（+49 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） 2、派发（交给调度模型，含池的入队） 25、每轮策略查询（周期/相位/超时各查一次配置） 17、提交侧其他（适配器自己的记账与完成信号状态） 7。

- 并发度 4、执行体 40 微秒 下，被测框架整体不慢于 apscheduler（每任务 -138 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 78 微秒，其中主要落在「手交」（+68 微秒）。
- 注意：该组逐段差额并非同号——「执行体」为 -0 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 343 微秒，其中主要落在「手交」（+432 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -82 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 373 微秒，其中主要落在「手交」（+171 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） 38、每轮策略查询（周期/相位/超时各查一次配置） 30、提交侧其他（适配器自己的记账与完成信号状态） 5。
- 并发度 1、执行体 10000 微秒 下，被测框架整体不慢于 apscheduler（每任务 -151 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 337 微秒，其中主要落在「执行体」（+272 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -4 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 342 微秒，其中主要落在「手交」（+5126 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -5087 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 264 微秒，其中主要落在「执行体」（+106 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） 3、派发（交给调度模型，含池的入队） 41、每轮策略查询（周期/相位/超时各查一次配置） 30、提交侧其他（适配器自己的记账与完成信号状态） 13。
- 并发度 4、执行体 10000 微秒 下，被测框架整体不慢于 apscheduler（每任务 -765 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 10000 微秒 下，被测框架相对 asyncio_pool 每任务多花 434 微秒，其中主要落在「回程」（+375 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -2047 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 9711 微秒，其中主要落在「手交」（+15357 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 671 微秒，其中主要落在「回程」（+862 微秒）。
- 注意：该组逐段差额并非同号——「执行体」为 -4505 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） 46、每轮策略查询（周期/相位/超时各查一次配置） 34、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（0.0123s）（执行体 0.15s、run_timeout 0.01s）；判定动作在 dispatch_core.py:365 ；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：not_applicable

被测框架不具备该能力（找不到这些符号：adaptive_params, dual_arm_worker, native_params, native_task_worker）——该线对此版本不适用，这不是失败

等价性门禁：未跑到（前置探查没走到这一步）

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）

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 1.62x	174.06 微秒
process_pool	1	40	被测框架比 process_pool 快 2.12x	174.06 微秒
apscheduler	1	300	被测框架比 apscheduler 快 1.26x	433.22 微秒
process_pool	1	300	被测框架比 process_pool 快 1.42x	433.22 微秒
apscheduler	1	2700	被测框架比 apscheduler 快 1.03x	2.800 毫秒
process_pool	1	2700	被测框架比 process_pool 快 1.07x	2.800 毫秒
apscheduler	1	10000	被测框架比 apscheduler 快 1.03x	9.885 毫秒
process_pool	1	10000	被测框架比 process_pool 快 1.02x	9.885 毫秒
apscheduler	4	40	被测框架比 apscheduler 快 1.28x	200.10 微秒
apscheduler	4	300	被测框架比 apscheduler 快 1.13x	457.43 微秒
apscheduler	4	2700	被测框架比 apscheduler 快 1.06x	2.814 毫秒
apscheduler	16	40	被测框架比 apscheduler 快 1.23x	199.72 微秒
apscheduler	16	300	被测框架比 apscheduler 快 1.12x	464.10 微秒
apscheduler	16	2700	被测框架比 apscheduler 快 1.04x	2.869 毫秒
apscheduler	64	40	被测框架比 apscheduler 快 1.25x	195.85 微秒
apscheduler	64	300	被测框架比 apscheduler 快 1.11x	465.14 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 2.25x	174.06 微秒
thread_pool	1	300	thread_pool 比 zoo 快 1.34x	433.22 微秒
asyncio_pool	1	2700	asyncio_pool 比 zoo 快 1.04x	2.800 毫秒
bare_thread	1	2700	bare_thread 比 zoo 快 1.04x	2.800 毫秒

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	2700	thread_pool 比 zoo 快 1.05x	2.800 毫秒
asyncio_pool	1	10000	asyncio_pool 比 zoo 快 1.02x	9.885 毫秒
bare_thread	1	10000	bare_thread 比 zoo 快 1.04x	9.885 毫秒
thread_pool	1	10000	thread_pool 比 zoo 快 1.02x	9.885 毫秒
asyncio_pool	4	40	asyncio_pool 比 zoo 快 1.28x	200.10 微秒
thread_pool	4	40	thread_pool 比 zoo 快 2.08x	200.10 微秒
asyncio_pool	4	300	asyncio_pool 比 zoo 快 1.12x	457.43 微秒
bare_thread	4	300	bare_thread 比 zoo 快 1.11x	457.43 微秒
process_pool	4	300	process_pool 比 zoo 快 1.34x	457.43 微秒
thread_pool	4	300	thread_pool 比 zoo 快 1.30x	457.43 微秒
bare_thread	4	2700	bare_thread 比 zoo 快 1.03x	2.814 毫秒
process_pool	4	2700	process_pool 比 zoo 快 2.07x	2.814 毫秒
thread_pool	4	2700	thread_pool 比 zoo 快 1.04x	2.814 毫秒
asyncio_pool	4	10000	asyncio_pool 比 zoo 快 1.03x	10.050 毫秒
bare_thread	4	10000	bare_thread 比 zoo 快 1.02x	10.050 毫秒
process_pool	4	10000	process_pool 比 zoo 快 2.16x	10.050 毫秒
thread_pool	4	10000	thread_pool 比 zoo 快 1.02x	10.050 毫秒
asyncio_pool	16	40	asyncio_pool 比 zoo 快 1.53x	199.72 微秒
bare_thread	16	40	bare_thread 比 zoo 快 1.17x	199.72 微秒
process_pool	16	40	process_pool 比 zoo 快 1.31x	199.72 微秒
thread_pool	16	40	thread_pool 比 zoo 快 2.24x	199.72 微秒
asyncio_pool	16	300	asyncio_pool 比 zoo 快 1.18x	464.10 微秒
bare_thread	16	300	bare_thread 比 zoo 快 1.11x	464.10 微秒
process_pool	16	300	process_pool 比 zoo 快 1.67x	464.10 微秒
thread_pool	16	300	thread_pool 比 zoo 快 1.31x	464.10 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
asyncio_pool	16	2700	asyncio_pool 比 zoo 快 1.04x	2.869 毫秒
bare_thread	16	2700	bare_thread 比 zoo 快 1.07x	2.869 毫秒
process_pool	16	2700	process_pool 比 zoo 快 2.03x	2.869 毫秒
thread_pool	16	2700	thread_pool 比 zoo 快 1.35x	2.869 毫秒
asyncio_pool	16	10000	asyncio_pool 比 zoo 快 1.02x	10.104 毫秒
bare_thread	16	10000	bare_thread 比 zoo 快 1.02x	10.104 毫秒
process_pool	16	10000	process_pool 比 zoo 快 2.15x	10.104 毫秒
thread_pool	16	10000	thread_pool 比 zoo 快 1.04x	10.104 毫秒
asyncio_pool	64	40	asyncio_pool 比 zoo 快 1.59x	195.85 微秒
bare_thread	64	40	bare_thread 比 zoo 快 1.12x	195.85 微秒
thread_pool	64	40	thread_pool 比 zoo 快 2.16x	195.85 微秒
asyncio_pool	64	300	asyncio_pool 比 zoo 快 1.22x	465.14 微秒
bare_thread	64	300	bare_thread 比 zoo 快 1.09x	465.14 微秒
process_pool	64	300	process_pool 比 zoo 快 1.50x	465.14 微秒
thread_pool	64	300	thread_pool 比 zoo 快 1.31x	465.14 微秒
asyncio_pool	64	2700	asyncio_pool 比 zoo 快 1.03x	2.881 毫秒
bare_thread	64	2700	bare_thread 比 zoo 快 1.04x	2.881 毫秒
process_pool	64	2700	process_pool 比 zoo 快 2.10x	2.881 毫秒
thread_pool	64	2700	thread_pool 比 zoo 快 1.04x	2.881 毫秒
apscheduler	64	10000	apscheduler 比 zoo 快 1.07x	10.631 毫秒
bare_thread	64	10000	bare_thread 比 zoo 快 1.08x	10.631 毫秒
process_pool	64	10000	process_pool 比 zoo 快 2.29x	10.631 毫秒

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：1.0% 到 35.2%（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	1.23x	两侧差异 22.6% 小于带宽 35.2%，分不出胜负
bare_thread	1	40	0.83x	两侧差异 16.8% 小于带宽 19.7%，分不出胜负
asyncio_pool	1	300	0.94x	两侧差异 6.4% 小于带宽 8.8%，分不出胜负
bare_thread	1	300	0.92x	两侧差异 8.4% 小于带宽 9.5%，分不出胜负
bare_thread	4	40	0.84x	两侧差异 15.7% 小于带宽 27.2%，分不出胜负
process_pool	4	40	0.96x	两侧差异 4.4% 小于带宽 16.2%，分不出胜负
asyncio_pool	4	2700	0.98x	两侧差异 1.8% 小于带宽 3.1%，分不出胜负
apscheduler	4	10000	1.01x	两侧差异 0.5% 小于带宽 1.4%，分不出胜负
apscheduler	16	10000	1.01x	两侧差异 1.2% 小于带宽 1.7%，分不出胜负
process_pool	64	40	1.02x	两侧差异 2.1% 小于带宽 11.5%，分不出胜负
apscheduler	64	2700	1.02x	两侧差异 2.0% 小于带宽 3.0%，分不出胜负
asyncio_pool	64	10000	0.98x	两侧差异 2.2% 小于带宽 3.8%，分不出胜负
thread_pool	64	10000	0.97x	两侧差异 2.8% 小于带宽 3.4%，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
- 原生任务线: 未出结论: 原生任务线结论为 not_applicable (被测框架不具备该能力 (找不到这些符号: adaptive_params, dual_arm_worker, native_params, native_task_worker) ——该线对此版本不适用, 这不是失败)、未产出任何数字。它记在这里而不是并入自检: 一条线的失败不得拖住其余版本的数据与主矩阵
- 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': 29954, 'child_pids': [30410, 30413, 30416, 30419, 30422, 30428, 30434, 30440, 30446, 30464, 30482, 30500, 30521, 30577, 30634, 30692, 30758, 30760, 30762, 30764, 30766, 30771, 30776, 30781, 30786, 30803, 30821, 30838, 30856, 30918, 30977, 31039, 31104, 31138, 31172, 31206, 31240, 31373, 31506, 31639, 31772, 32301, 32830, 33359, 33888, 36001, 38115, 40228, 42343, 42347, 42351, 42355, 42359, 42366, 42373, 42380, 42387, 42406, 42425, 42444, 42463, 42530, 42597, 42664, 42732, 42734, 42736, 42738, 42740, 42745, 42750, 42755,

42760, 42777, 42794, 42811, 42829, 42885, 42941, 43000, 43068, 43070, 43072, 43074, 43076, 43081, 43086, 43091, 43096, 43113, 43130, 43147, 43164, 43205, 43239, 43280], '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	281.24 微秒	330.91 微秒	334.09 微秒	0.071
apscheduler	1	300	544.52 微秒	572.11 微秒	578.66 微秒	0.035
apscheduler	1	2700	2.884 毫秒	2.955 毫秒	3.076 毫秒	0.019
apscheduler	1	10000	10.156 毫秒	10.294 毫秒	10.376 毫秒	0.013
apscheduler	4	40	255.42 微秒	282.19 微秒	297.32 微秒	0.047
apscheduler	4	300	516.26 微秒	547.34 微秒	563.64 微秒	0.029
apscheduler	4	2700	2.983 毫秒	3.037 毫秒	3.077 毫秒	0.013
apscheduler	4	10000	10.101 毫秒	10.215 毫秒	10.306 毫秒	0.008
apscheduler	16	40	246.53 微秒	260.75 微秒	274.95 微秒	0.043
apscheduler	16	300	520.81 微秒	531.42 微秒	533.12 微秒	0.021
apscheduler	16	2700	2.998 毫秒	3.061 毫秒	3.068 毫秒	0.015
apscheduler	16	10000	10.223 毫秒	10.325 毫秒	10.429 毫秒	0.011
apscheduler	64	40	245.59 微秒	261.27 微秒	273.04 微秒	0.058
apscheduler	64	300	514.29 微秒	540.01 微秒	549.92 微秒	0.031
apscheduler	64	2700	2.940 毫秒	3.031 毫秒	3.147 毫秒	0.020
apscheduler	64	10000	9.958 毫秒	10.331 毫秒	10.436 毫秒	0.021
asyncio_pool	1	40	213.36 微秒	294.34 微秒	303.06 微秒	0.254
asyncio_pool	1	300	405.28 微秒	432.68 微秒	436.27 微秒	0.034
asyncio_pool	1	2700	2.701 毫秒	2.740 毫秒	2.742 毫秒	0.007
asyncio_pool	1	10000	9.700 毫秒	9.791 毫秒	9.826 毫秒	0.007
asyncio_pool	4	40	156.05 微秒	178.57 微秒	186.23 微秒	0.081
asyncio_pool	4	300	407.59 微秒	435.64 微秒	443.37 微秒	0.030
asyncio_pool	4	2700	2.764 毫秒	2.847 毫秒	2.873 毫秒	0.016
asyncio_pool	4	10000	9.786 毫秒	9.908 毫秒	9.942 毫秒	0.006
asyncio_pool	16	40	130.72 微秒	144.29 微秒	146.15 微秒	0.048
asyncio_pool	16	300	392.41 微秒	411.25 微秒	422.56 微秒	0.024
asyncio_pool	16	2700	2.756 毫秒	2.816 毫秒	2.822 毫秒	0.012
asyncio_pool	16	10000	9.931 毫秒	10.004 毫秒	10.010 毫秒	0.007
asyncio_pool	64	40	123.00 微秒	126.31 微秒	129.42 微秒	0.022
asyncio_pool	64	300	381.41 微秒	391.30 微秒	417.56 微秒	0.024
asyncio_pool	64	2700	2.804 毫秒	2.846 毫秒	2.857 毫秒	0.008
asyncio_pool	64	10000	10.397 毫秒	10.561 毫秒	10.584 毫秒	0.018
bare_thread	1	40	144.76 微秒	175.84 微秒	198.18 微秒	0.099
bare_thread	1	300	396.70 微秒	436.06 微秒	444.03 微秒	0.042

方案	并发度	执行体档位 (微秒)	中位数	p95	p99	相对离散度
bare_thread	1	2700	2.680 毫秒	2.726 毫秒	2.739 毫秒	0.009
bare_thread	1	10000	9.548 毫秒	9.655 毫秒	9.731 毫秒	0.007
bare_thread	4	40	168.63 微秒	252.60 微秒	257.71 微秒	0.215
bare_thread	4	300	412.07 微秒	434.78 微秒	440.21 微秒	0.026
bare_thread	4	2700	2.728 毫秒	2.756 毫秒	2.765 毫秒	0.007
bare_thread	4	10000	9.829 毫秒	9.887 毫秒	9.924 毫秒	0.004
bare_thread	16	40	170.14 微秒	176.65 微秒	177.51 微秒	0.025
bare_thread	16	300	419.21 微秒	428.16 微秒	436.10 微秒	0.013
bare_thread	16	2700	2.682 毫秒	2.699 毫秒	2.700 毫秒	0.004
bare_thread	16	10000	9.895 毫秒	9.953 毫秒	10.039 毫秒	0.004
bare_thread	64	40	174.50 微秒	206.96 微秒	208.44 微秒	0.082
bare_thread	64	300	426.28 微秒	434.04 微秒	440.99 微秒	0.010
bare_thread	64	2700	2.775 毫秒	2.925 毫秒	2.984 毫秒	0.021
bare_thread	64	10000	9.824 毫秒	9.862 毫秒	10.052 毫秒	0.006
process_pool	1	40	369.75 微秒	523.35 微秒	702.27 微秒	0.221
process_pool	1	300	614.59 微秒	646.64 微秒	657.87 微秒	0.031
process_pool	1	2700	2.982 毫秒	3.078 毫秒	3.162 毫秒	0.016
process_pool	1	10000	10.087 毫秒	10.228 毫秒	10.606 毫秒	0.013
process_pool	4	40	191.30 微秒	226.31 微秒	233.00 微秒	0.105
process_pool	4	300	341.25 微秒	392.67 微秒	397.76 微秒	0.111
process_pool	4	2700	1.359 毫秒	1.636 毫秒	2.046 毫秒	0.120
process_pool	4	10000	4.650 毫秒	5.143 毫秒	5.672 毫秒	0.055
process_pool	16	40	152.53 微秒	195.85 微秒	222.74 微秒	0.159
process_pool	16	300	277.10 微秒	325.51 微秒	342.76 微秒	0.089
process_pool	16	2700	1.413 毫秒	1.522 毫秒	1.531 毫秒	0.035
process_pool	16	10000	4.698 毫秒	4.896 毫秒	4.946 毫秒	0.019
process_pool	64	40	200.04 微秒	241.82 微秒	253.06 微秒	0.095
process_pool	64	300	310.51 微秒	345.27 微秒	355.80 微秒	0.059
process_pool	64	2700	1.375 毫秒	1.448 毫秒	1.489 毫秒	0.023
process_pool	64	10000	4.637 毫秒	4.695 毫秒	4.742 毫秒	0.007
thread_pool	1	40	77.28 微秒	97.73 微秒	131.24 微秒	0.178
thread_pool	1	300	324.43 微秒	343.06 微秒	365.55 微秒	0.038
thread_pool	1	2700	2.658 毫秒	2.708 毫秒	2.729 毫秒	0.009
thread_pool	1	10000	9.698 毫秒	9.755 毫秒	9.835 毫秒	0.006
thread_pool	4	40	96.24 微秒	109.34 微秒	114.24 微秒	0.064
thread_pool	4	300	352.35 微秒	369.30 微秒	376.29 微秒	0.021
thread_pool	4	2700	2.711 毫秒	2.755 毫秒	2.771 毫秒	0.011
thread_pool	4	10000	9.811 毫秒	9.987 毫秒	10.010 毫秒	0.008
thread_pool	16	40	88.97 微秒	96.29 微秒	96.75 微秒	0.062
thread_pool	16	300	354.95 微秒	365.09 微秒	366.11 微秒	0.013
thread_pool	16	2700	2.133 毫秒	2.172 毫秒	2.176 毫秒	0.012
thread_pool	16	10000	9.734 毫秒	9.914 毫秒	10.042 毫秒	0.009
thread_pool	64	40	90.65 微秒	93.87 微秒	102.53 微秒	0.035

方案	并发度	执行体档位（微秒）	中位数	p95	p99	相对离散度
thread_pool	64	300	353.92 微秒	414.87 微秒	472.73 微秒	0.084
thread_pool	64	2700	2.774 毫秒	2.889 毫秒	2.977 毫秒	0.019
thread_pool	64	10000	10.331 毫秒	10.440 毫秒	10.460 毫秒	0.014
zoo	1	40	174.06 微秒	211.81 微秒	226.31 微秒	0.098
zoo	1	300	433.22 微秒	475.65 微秒	481.58 微秒	0.054
zoo	1	2700	2.800 毫秒	2.853 毫秒	2.875 毫秒	0.009
zoo	1	10000	9.885 毫秒	9.996 毫秒	10.110 毫秒	0.007
zoo	4	40	200.10 微秒	217.10 微秒	218.93 微秒	0.057
zoo	4	300	457.43 微秒	477.75 微秒	504.06 微秒	0.031
zoo	4	2700	2.814 毫秒	2.893 毫秒	2.915 毫秒	0.016
zoo	4	10000	10.050 毫秒	10.125 毫秒	10.177 毫秒	0.006
zoo	16	40	199.72 微秒	210.55 微秒	214.54 微秒	0.026
zoo	16	300	464.10 微秒	477.83 微秒	481.00 微秒	0.016
zoo	16	2700	2.869 毫秒	2.904 毫秒	2.919 毫秒	0.010
zoo	16	10000	10.104 毫秒	10.191 毫秒	10.304 毫秒	0.006
zoo	64	40	195.85 微秒	203.19 微秒	204.54 微秒	0.020
zoo	64	300	465.14 微秒	470.93 微秒	474.34 微秒	0.009
zoo	64	2700	2.881 毫秒	2.939 毫秒	2.962 毫秒	0.010
zoo	64	10000	10.631 毫秒	10.833 毫秒	11.023 毫秒	0.020

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	232.64 微秒	82.72%	48.60 微秒
apscheduler	1	300	250.38 微秒	45.98%	294.14 微秒
apscheduler	1	2700	295.90 微秒	10.26%	2.588 毫秒
apscheduler	1	10000	365.37 微秒	3.60%	9.790 毫秒
apscheduler	4	40	199.69 微秒	78.18%	55.73 微秒
apscheduler	4	300	207.65 微秒	40.22%	308.61 微秒
apscheduler	4	2700	291.97 微秒	9.79%	2.691 毫秒
apscheduler	4	10000	—	—	14.850 毫秒
apscheduler	16	40	—	—	52.28 微秒
apscheduler	16	300	—	—	309.42 微秒
apscheduler	16	2700	—	—	2.739 毫秒
apscheduler	16	10000	—	—	15.698 毫秒
apscheduler	64	40	—	—	50.46 微秒
apscheduler	64	300	—	—	306.59 微秒
apscheduler	64	2700	—	—	2.676 毫秒
apscheduler	64	10000	—	—	15.208 毫秒
asyncio_pool	1	40	148.75 微秒	69.72%	64.61 微秒
asyncio_pool	1	300	107.53 微秒	26.53%	297.76 微秒
asyncio_pool	1	2700	123.69 微秒	4.58%	2.578 毫秒
asyncio_pool	1	10000	153.03 微秒	1.58%	9.547 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
asyncio_pool	4	40	102.43 微秒	65.64%	53.62 微秒
asyncio_pool	4	300	104.85 微秒	25.72%	302.74 微秒
asyncio_pool	4	2700	111.01 微秒	4.02%	2.652 毫秒
asyncio_pool	4	10000	—	—	13.992 毫秒
asyncio_pool	16	40	—	—	53.62 微秒
asyncio_pool	16	300	—	—	304.98 微秒
asyncio_pool	16	2700	—	—	2.632 毫秒
asyncio_pool	16	10000	—	—	19.975 毫秒
asyncio_pool	64	40	—	—	52.29 微秒
asyncio_pool	64	300	—	—	302.33 微秒
asyncio_pool	64	2700	—	—	2.675 毫秒
asyncio_pool	64	10000	—	—	81.526 毫秒
bare_thread	1	40	92.45 微秒	63.87%	52.31 微秒
bare_thread	1	300	101.31 微秒	25.54%	295.39 微秒
bare_thread	1	2700	114.23 微秒	4.26%	2.566 毫秒
bare_thread	1	10000	126.52 微秒	1.33%	9.422 毫秒
bare_thread	4	40	116.46 微秒	69.06%	52.17 微秒
bare_thread	4	300	118.13 微秒	28.67%	293.94 微秒
bare_thread	4	2700	127.88 微秒	4.69%	2.601 毫秒
bare_thread	4	10000	—	—	14.814 毫秒
bare_thread	16	40	—	—	51.40 微秒
bare_thread	16	300	—	—	286.58 微秒
bare_thread	16	2700	—	—	2.539 毫秒
bare_thread	16	10000	—	—	14.979 毫秒
bare_thread	64	40	—	—	51.89 微秒
bare_thread	64	300	—	—	296.66 微秒
bare_thread	64	2700	—	—	2.640 毫秒
bare_thread	64	10000	—	—	15.142 毫秒
process_pool	1	40	313.46 微秒	84.78%	56.29 微秒
process_pool	1	300	304.04 微秒	49.47%	310.54 微秒
process_pool	1	2700	338.24 微秒	11.34%	2.644 毫秒
process_pool	1	10000	342.42 微秒	3.39%	9.744 毫秒
process_pool	4	40	101.51 微秒	53.06%	89.79 微秒
process_pool	4	300	—	—	568.22 微秒
process_pool	4	2700	—	—	4.781 毫秒
process_pool	4	10000	—	—	17.953 毫秒
process_pool	16	40	—	—	91.18 微秒
process_pool	16	300	—	—	583.81 微秒
process_pool	16	2700	—	—	9.970 毫秒
process_pool	16	10000	—	—	52.459 毫秒
process_pool	64	40	—	—	91.01 微秒
process_pool	64	300	—	—	570.05 微秒
process_pool	64	2700	—	—	28.234 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
process_pool	64	10000	—	—	159.612 毫秒
thread_pool	1	40	30.24 微秒	39.13%	47.04 微秒
thread_pool	1	300	36.08 微秒	11.12%	288.35 微秒
thread_pool	1	2700	48.20 微秒	1.81%	2.610 毫秒
thread_pool	1	10000	70.99 微秒	0.73%	9.627 毫秒
thread_pool	4	40	46.57 微秒	48.39%	49.67 微秒
thread_pool	4	300	53.93 微秒	15.31%	298.42 微秒
thread_pool	4	2700	99.41 微秒	3.67%	2.611 毫秒
thread_pool	4	10000	—	—	15.069 毫秒
thread_pool	16	40	—	—	48.79 微秒
thread_pool	16	300	—	—	305.06 微秒
thread_pool	16	2700	—	—	2.035 毫秒
thread_pool	16	10000	—	—	19.866 毫秒
thread_pool	64	40	—	—	47.76 微秒
thread_pool	64	300	—	—	297.37 微秒
thread_pool	64	2700	—	—	2.692 毫秒
thread_pool	64	10000	—	—	59.527 毫秒
zoo	1	40	123.88 微秒	71.17%	50.17 微秒
zoo	1	300	131.17 微秒	30.28%	302.05 微秒
zoo	1	2700	136.04 微秒	4.86%	2.664 毫秒
zoo	1	10000	147.92 微秒	1.50%	9.737 毫秒
zoo	4	40	144.96 微秒	72.44%	55.14 微秒
zoo	4	300	151.15 微秒	33.04%	306.27 微秒
zoo	4	2700	200.65 微秒	7.13%	2.613 毫秒
zoo	4	10000	—	—	14.817 毫秒
zoo	16	40	—	—	52.89 微秒
zoo	16	300	—	—	305.22 微秒
zoo	16	2700	—	—	2.658 毫秒
zoo	16	10000	—	—	20.089 毫秒
zoo	64	40	—	—	52.68 微秒
zoo	64	300	—	—	303.54 微秒
zoo	64	2700	—	—	2.679 毫秒
zoo	64	10000	—	—	51.276 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	3555.7
apscheduler	1	300	1836.5
apscheduler	1	2700	346.7
apscheduler	1	10000	98.5
apscheduler	4	40	15660.3
apscheduler	4	300	7748.0
apscheduler	4	2700	1340.8

方案	并发度	执行体档位（微秒）	吞吐（任务/秒）
apscheduler	4	10000	396.0
apscheduler	16	40	64899.7
apscheduler	16	300	30721.3
apscheduler	16	2700	5337.6
apscheduler	16	10000	1565.0
apscheduler	64	40	260593.3
apscheduler	64	300	124444.1
apscheduler	64	2700	21770.5
apscheduler	64	10000	6427.2
asyncio_pool	1	40	4686.8
asyncio_pool	1	300	2467.4
asyncio_pool	1	2700	370.2
asyncio_pool	1	10000	103.1
asyncio_pool	4	40	25632.9
asyncio_pool	4	300	9813.8
asyncio_pool	4	2700	1447.4
asyncio_pool	4	10000	408.8
asyncio_pool	16	40	122396.1
asyncio_pool	16	300	40773.2
asyncio_pool	16	2700	5805.4
asyncio_pool	16	10000	1611.1
asyncio_pool	64	40	520342.3
asyncio_pool	64	300	167797.3
asyncio_pool	64	2700	22823.4
asyncio_pool	64	10000	6155.5
bare_thread	1	40	6907.9
bare_thread	1	300	2520.8
bare_thread	1	2700	373.2
bare_thread	1	10000	104.7
bare_thread	4	40	23720.3
bare_thread	4	300	9707.1
bare_thread	4	2700	1466.1
bare_thread	4	10000	407.0
bare_thread	16	40	94038.7
bare_thread	16	300	38167.2
bare_thread	16	2700	5965.9
bare_thread	16	10000	1617.0
bare_thread	64	40	366766.2
bare_thread	64	300	150135.6
bare_thread	64	2700	23067.0
bare_thread	64	10000	6514.5
process_pool	1	40	2704.5
process_pool	1	300	1627.1
process_pool	1	2700	335.3

方案	并发度	执行体档位（微秒）	吞吐（任务/秒）
process_pool	1	10000	99.1
process_pool	4	40	20909.5
process_pool	4	300	11721.5
process_pool	4	2700	2944.3
process_pool	4	10000	860.2
process_pool	16	40	104898.1
process_pool	16	300	57740.1
process_pool	16	2700	11327.3
process_pool	16	10000	3406.0
process_pool	64	40	319932.6
process_pool	64	300	206112.0
process_pool	64	2700	46544.6
process_pool	64	10000	13803.0
thread_pool	1	40	12939.1
thread_pool	1	300	3082.3
thread_pool	1	2700	376.2
thread_pool	1	10000	103.1
thread_pool	4	40	41562.7
thread_pool	4	300	11352.2
thread_pool	4	2700	1475.5
thread_pool	4	10000	407.7
thread_pool	16	40	179841.5
thread_pool	16	300	45076.5
thread_pool	16	2700	7501.0
thread_pool	16	10000	1643.7
thread_pool	64	40	705985.9
thread_pool	64	300	180832.2
thread_pool	64	2700	23074.4
thread_pool	64	10000	6194.7
zoo	1	40	5745.3
zoo	1	300	2308.3
zoo	1	2700	357.2
zoo	1	10000	101.2
zoo	4	40	19989.9
zoo	4	300	8744.6
zoo	4	2700	1421.5
zoo	4	10000	398.0
zoo	16	40	80111.5
zoo	16	300	34475.3
zoo	16	2700	5576.0
zoo	16	10000	1583.6
zoo	64	40	326787.3
zoo	64	300	137591.9
zoo	64	2700	22213.7

方案	并发度	执行体档位（微秒）	吞吐（任务/秒）
zoo	64	10000	6020.1