

zoo-bench 性能报告

原始数据：results/zoo-framework-0.9.0/20261011T092749Z.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）	0.9.0
被测框架（模块 __version__）	0.9.0（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==0.9.0

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 1.83x（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	182.57 微秒	288.52 微秒	140.37 微秒	148.91 微秒	333.53 微秒	75.77 微秒
1	300	449.17 微秒	542.76 微秒	386.47 微秒	384.68 微秒	607.37 微秒	322.77 微秒
1	2700	2.754 毫秒	2.999 毫秒	2.752 毫秒	2.672 毫秒	3.009 毫秒	2.689 毫秒
1	10000	10.066 毫秒	9.971 毫秒	9.482 毫秒	9.648 毫秒	9.888 毫秒	9.361 毫秒
4	40	206.14 微秒	256.36 微秒	173.04 微秒	199.87 微秒	168.78 微秒	96.03 微秒
4	300	485.26 微秒	511.06 微秒	407.48 微秒	419.42 微秒	304.29 微秒	350.77 微秒
4	2700	2.946 毫秒	2.983 毫秒	2.755 毫秒	2.706 毫秒	1.385 毫秒	2.742 毫秒
4	10000	10.106 毫秒	10.258 毫秒	9.867 毫秒	9.696 毫秒	4.621 毫秒	9.813 毫秒
16	40	205.75 微秒	224.01 微秒	134.08 微秒	167.78 微秒	149.92 微秒	90.21 微秒
16	300	470.57 微秒	505.14 微秒	393.00 微秒	424.88 微秒	278.51 微秒	343.15 微秒
16	2700	2.899 毫秒	2.980 毫秒	2.802 毫秒	2.711 毫秒	1.370 毫秒	2.772 毫秒
16	10000	10.084 毫秒	9.881 毫秒	9.935 毫秒	9.609 毫秒	4.741 毫秒	9.803 毫秒

并发度	执行体档位 (微秒)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
64	40	207.28 微秒	220.85 微秒	120.26 微秒	171.52 微秒	194.08 微秒	89.21 微秒
64	300	491.01 微秒	501.75 微秒	384.40 微秒	424.30 微秒	299.76 微秒	350.35 微秒
64	2700	2.953 毫秒	2.965 毫秒	2.811 毫秒	2.675 毫秒	1.364 毫秒	2.696 毫秒
64	10000	10.538 毫秒	10.167 毫秒	10.407 毫秒	9.827 毫秒	4.686 毫秒	10.244 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	71.66%	79.81%	67.08%	64.13%	83.82%	41.38%
1	300	32.12%	44.93%	26.24%	25.34%	48.87%	11.10%
1	2700	5.65%	9.02%	4.58%	4.02%	11.37%	2.35%
1	10000	2.55%	3.19%	1.54%	1.25%	3.45%	0.67%
4	40	73.26%	78.08%	68.70%	70.52%	67.43%	48.25%
4	300	36.28%	39.64%	25.85%	28.21%	—	14.92%
4	2700	7.68%	8.71%	4.45%	4.05%	—	2.85%
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	5477.4	3466.0	7123.9	6715.5	2998.2	13198.4
1	300	2226.3	1842.4	2587.5	2599.6	1646.4	3098.1
1	2700	363.1	333.4	363.4	374.3	332.3	371.9
1	10000	99.3	100.3	105.5	103.6	101.1	106.8
4	40	19404.4	15602.9	23116.5	20013.0	23699.2	41654.8
4	300	8242.9	7826.8	9816.5	9537.0	13145.4	11403.4
4	2700	1357.7	1341.0	1451.9	1478.1	2888.8	1459.0
4	10000	395.8	389.9	405.4	412.6	865.5	407.6

并发度	执行体档位 (微秒)	zoo	apscheduler	asyncio_pool	bare_thread	process_pool	thread_pool
16	40	77763.4	71424.4	119331.0	95364.7	106727.0	177361.2
16	300	34001.3	31674.5	40712.6	37658.1	57448.9	46626.5
16	2700	5518.7	5369.8	5711.1	5901.5	11678.3	5772.6
16	10000	1586.7	1619.2	1610.5	1665.2	3375.1	1632.1
64	40	308754.3	289794.7	532160.1	373124.9	329761.1	717389.6
64	300	130343.7	127553.4	166491.4	150836.5	213501.9	182674.1
64	2700	21671.4	21585.8	22767.3	23921.9	46909.7	23741.9
64	10000	6073.5	6294.8	6149.5	6512.8	13657.9	6247.3

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	108.78 微秒	81.29 微秒	58.86 微秒	42.86 微秒	ok
apscheduler	4	40	110.68 微秒	519.28 微秒	56.18 微秒	218.97 微秒	ok
apscheduler	1	10000	166.72 微秒	122.62 微秒	9.782 毫秒	59.54 微秒	ok
apscheduler	4	10000	119.74 微秒	12.884 毫秒	14.542 毫秒	9.805 毫秒	ok
asyncio_pool	1	40	27.40 微秒	47.40 微秒	56.28 微秒	78.36 微秒	ok
asyncio_pool	4	40	19.00 微秒	259.15 微秒	54.13 微秒	309.16 微秒	ok
asyncio_pool	1	10000	22.65 微秒	36.13 微秒	9.197 毫秒	84.50 微秒	ok
asyncio_pool	4	10000	23.82 微秒	12.044 毫秒	14.259 毫秒	9.649 毫秒	ok
bare_thread	1	40	141.09 微秒	-77.35 微秒	52.12 微秒	31.62 微秒	ok
bare_thread	4	40	150.71 微秒	-74.46 微秒	50.79 微秒	302.37 微秒	ok
bare_thread	1	10000	5.173 毫秒	-5094.45 微秒	9.675 毫秒	38.67 微秒	ok
bare_thread	4	10000	5.189 毫秒	-5084.41 微秒	14.786 毫秒	9.975 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	8.04 微秒	10.46 微秒	45.22 微秒	15.50 微秒	ok
thread_pool	4	40	11.13 微秒	180.20 微秒	49.39 微秒	148.07 微秒	ok
thread_pool	1	10000	13.34 微秒	17.54 微秒	9.434 毫秒	28.94 微秒	ok
thread_pool	4	10000	13.42 微秒	10.048 毫秒	15.032 毫秒	9.237 毫秒	ok
zoo	1	40	56.37 微秒	24.72 微秒	52.56 微秒	62.97 微秒	ok
zoo	4	40	64.44 微秒	392.36 微秒	56.29 微秒	334.47 微秒	ok
zoo	1	10000	93.87 微秒	30.59 微秒	9.703 毫秒	100.45 微秒	ok
zoo	4	10000	92.78 微秒	10.328 毫秒	14.664 毫秒	9.974 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	-100.74 微秒	回程	20.10 微秒
asyncio_pool	1	40	-13.24 微秒	提交侧	28.96 微秒
bare_thread	1	40	49.06 微秒	手交	102.07 微秒
thread_pool	1	40	117.53 微秒	提交侧	48.33 微秒
apscheduler	4	40	-134.18 微秒	回程	115.50 微秒
asyncio_pool	4	40	134.12 微秒	手交	133.22 微秒
bare_thread	4	40	337.46 微秒	手交	466.82 微秒
thread_pool	4	40	404.51 微秒	手交	212.16 微秒
apscheduler	1	10000	-196.87 微秒	回程	40.91 微秒
asyncio_pool	1	10000	580.65 微秒	执行体	506.69 微秒
bare_thread	1	10000	141.08 微秒	手交	5.125 毫秒
thread_pool	1	10000	429.60 微秒	执行体	269.02 微秒
apscheduler	4	10000	257.50 微秒	回程	169.34 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	1.275 毫秒	执行体	404.28 微秒
bare_thread	4	10000	8.348 毫秒	手交	15.412 毫秒
thread_pool	4	10000	671.61 微秒	回程	737.02 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 195.82 微秒 对插桩后 197.14 微秒，差 1.32 微秒；zoo 40 微秒 / 并发 4：不插桩 788.53 微秒 对插桩后 782.07 微秒，差 6.46 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 9.971 毫秒 对插桩后 9.936 毫秒，差 34.78

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 40.412 毫秒 对插桩后 40.009 毫秒，差 403.79 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 1 超出 1.3%；zoo 40 微秒 / 并发 4 超出 5.1%；zoo 10000 微秒 / 并发 4 超出 94.1%

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） 0.25 微秒；派发（交给调度模型，含池的入队） 26.68 微秒；每轮策略查询（周期/相位/超时各查一次配置） 20.73 微秒；提交侧其他（适配器自己的记账与完成信号状态） 8.48 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） 36.68 微秒；每轮策略查询（周期/相位/超时各查一次配置） 29.29 微秒；提交侧其他（适配器自己的记账与完成信号状态） 4.30 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） 0.30 微秒；派发（交给调度模型，含池的入队） 43.21 微秒；每轮策略查询（周期/相位/超时各查一次配置） 35.41 微秒；提交侧其他（适配器自己的记账与完成信号状态） 15.10 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） 51.42 微秒；每轮策略查询（周期/相位/超时各查一次配置） 49.10 微秒；提交侧其他（适配器自己的记账与完成信号状态） 3.68 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 -101 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，被测框架整体不慢于 asyncio_pool（每任务 -13 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 bare_thread 每任务多花 49 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 -85 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 118 微秒，其中主要落在「提交侧」（+48 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） 27、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 -134 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 134 微秒，其中主要落在「手交」（+133 微秒）。
- 并发度 4、执行体 40 微秒 下，被测框架相对 bare_thread 每任务多花 337 微秒，其中主要落在「手交」（+467 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -86 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 405 微秒，其中主要落在「手交」（+212 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）37、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 -197 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 581 微秒，其中主要落在「执行体」（+507 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -6 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 141 微秒，其中主要落在「手交」（+5125 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -5079 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 430 微秒，其中主要落在「执行体」（+269 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）43、每轮策略查询（周期/相位/超时各查一次配置）35、提交侧其他（适配器自己的记账与完成信号状态）15。
- 并发度 4、执行体 10000 微秒 下，被测框架相对 apscheduler 每任务多花 258 微秒，其中主要落在「回程」（+169 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -2557 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 1275 微秒，其中主要落在「执行体」（+404 微秒）。
- 注意：该组逐段差额并非同号——「手交」为 -1716 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 8348 微秒，其中主要落在「手交」（+15412 微秒）。
- 注意：该组逐段差额并非同号——「提交侧」为 -5096 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 672 微秒，其中主要落在「回程」（+737 微秒）。
- 注意：该组逐段差额并非同号——「执行体」为 -368 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）51、每轮策略查询（周期/相位/超时各查一次配置）49、提交侧其他（适配器自己的记账与完成信号状态）4。

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:425 ；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 1.58x	182.57 微秒
process_pool	1	40	被测框架比 process_pool 快 1.83x	182.57 微秒
apscheduler	1	300	被测框架比 apscheduler 快 1.21x	449.17 微秒
process_pool	1	300	被测框架比 process_pool 快 1.35x	449.17 微秒
apscheduler	1	2700	被测框架比 apscheduler 快 1.09x	2.754 毫秒
process_pool	1	2700	被测框架比 process_pool 快 1.09x	2.754 毫秒
apscheduler	4	40	被测框架比 apscheduler 快 1.24x	206.14 微秒
apscheduler	16	300	被测框架比 apscheduler 快 1.07x	470.57 微秒
apscheduler	16	2700	被测框架比 apscheduler 快 1.03x	2.899 毫秒

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
asyncio_pool	1	40	asyncio_pool 比 zoo 快 1.30x	182.57 微秒
thread_pool	1	40	thread_pool 比 zoo 快 2.41x	182.57 微秒
asyncio_pool	1	300	asyncio_pool 比 zoo 快 1.16x	449.17 微秒
bare_thread	1	300	bare_thread 比 zoo 快 1.17x	449.17 微秒
thread_pool	1	300	thread_pool 比 zoo 快 1.39x	449.17 微秒
bare_thread	1	2700	bare_thread 比 zoo 快 1.03x	2.754 毫秒
thread_pool	1	2700	thread_pool 比 zoo 快 1.02x	2.754 毫秒
asyncio_pool	1	10000	asyncio_pool 比 zoo 快 1.06x	10.066 毫秒
bare_thread	1	10000	bare_thread 比 zoo 快 1.04x	10.066 毫秒
thread_pool	1	10000	thread_pool 比 zoo 快 1.08x	10.066 毫秒
thread_pool	4	40	thread_pool 比 zoo 快 2.15x	206.14 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	4	300	asyncio_pool 比 zoo 快 1.19x	485.26 微秒
bare_thread	4	300	bare_thread 比 zoo 快 1.16x	485.26 微秒
process_pool	4	300	process_pool 比 zoo 快 1.59x	485.26 微秒
thread_pool	4	300	thread_pool 比 zoo 快 1.38x	485.26 微秒
asyncio_pool	4	2700	asyncio_pool 比 zoo 快 1.07x	2.946 毫秒
bare_thread	4	2700	bare_thread 比 zoo 快 1.09x	2.946 毫秒
process_pool	4	2700	process_pool 比 zoo 快 2.13x	2.946 毫秒
thread_pool	4	2700	thread_pool 比 zoo 快 1.07x	2.946 毫秒
asyncio_pool	4	10000	asyncio_pool 比 zoo 快 1.02x	10.106 毫秒
process_pool	4	10000	process_pool 比 zoo 快 2.19x	10.106 毫秒
thread_pool	4	10000	thread_pool 比 zoo 快 1.03x	10.106 毫秒
asyncio_pool	16	40	asyncio_pool 比 zoo 快 1.53x	205.75 微秒
bare_thread	16	40	bare_thread 比 zoo 快 1.23x	205.75 微秒
thread_pool	16	40	thread_pool 比 zoo 快 2.28x	205.75 微秒
asyncio_pool	16	300	asyncio_pool 比 zoo 快 1.20x	470.57 微秒
bare_thread	16	300	bare_thread 比 zoo 快 1.11x	470.57 微秒
process_pool	16	300	process_pool 比 zoo 快 1.69x	470.57 微秒
thread_pool	16	300	thread_pool 比 zoo 快 1.37x	470.57 微秒
asyncio_pool	16	2700	asyncio_pool 比 zoo 快 1.03x	2.899 毫秒
bare_thread	16	2700	bare_thread 比 zoo 快 1.07x	2.899 毫秒
process_pool	16	2700	process_pool 比 zoo 快 2.12x	2.899 毫秒
thread_pool	16	2700	thread_pool 比 zoo 快 1.05x	2.899 毫秒
apscheduler	16	10000	apscheduler 比 zoo 快 1.02x	10.084 毫秒
asyncio_pool	16	10000	asyncio_pool 比 zoo 快 1.02x	10.084 毫秒
bare_thread	16	10000	bare_thread 比 zoo 快 1.05x	10.084 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
process_pool	16	10000	process_pool 比 zoo 快 2.13x	10.084 毫秒
thread_pool	16	10000	thread_pool 比 zoo 快 1.03x	10.084 毫秒
asyncio_pool	64	40	asyncio_pool 比 zoo 快 1.72x	207.28 微秒
bare_thread	64	40	bare_thread 比 zoo 快 1.21x	207.28 微秒
thread_pool	64	40	thread_pool 比 zoo 快 2.32x	207.28 微秒
asyncio_pool	64	300	asyncio_pool 比 zoo 快 1.28x	491.01 微秒
bare_thread	64	300	bare_thread 比 zoo 快 1.16x	491.01 微秒
process_pool	64	300	process_pool 比 zoo 快 1.64x	491.01 微秒
thread_pool	64	300	thread_pool 比 zoo 快 1.40x	491.01 微秒
asyncio_pool	64	2700	asyncio_pool 比 zoo 快 1.05x	2.953 毫秒
bare_thread	64	2700	bare_thread 比 zoo 快 1.10x	2.953 毫秒
process_pool	64	2700	process_pool 比 zoo 快 2.16x	2.953 毫秒
thread_pool	64	2700	thread_pool 比 zoo 快 1.10x	2.953 毫秒
apscheduler	64	10000	apscheduler 比 zoo 快 1.04x	10.538 毫秒
bare_thread	64	10000	bare_thread 比 zoo 快 1.07x	10.538 毫秒
process_pool	64	10000	process_pool 比 zoo 快 2.25x	10.538 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：1.4% 到 30.1%（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
bare_thread	1	40	0.82x	两侧差异 18.4% 小于带宽 30.1%，分不出胜负
asyncio_pool	1	2700	1.00x	两侧差异 0.1% 小于带宽 3.1%，分不出胜负

对照方案	并发度	执行体档位（微秒）	比值	原因
apscheduler	1	10000	0.99x	两侧差异 0.9% 小于带宽 4.6%，分不出胜负
process_pool	1	10000	0.98x	两侧差异 1.8% 小于带宽 1.9%，分不出胜负
asyncio_pool	4	40	0.84x	两侧差异 16.1% 小于带宽 17.3%，分不出胜负
bare_thread	4	40	0.97x	两侧差异 3.0% 小于带宽 14.3%，分不出胜负
process_pool	4	40	0.82x	两侧差异 18.1% 小于带宽 22.2%，分不出胜负
apscheduler	4	300	1.05x	两侧差异 5.3% 小于带宽 7.0%，分不出胜负
apscheduler	4	2700	1.01x	两侧差异 1.3% 小于带宽 2.7%，分不出胜负
apscheduler	4	10000	1.02x	两侧差异 1.5% 小于带宽 3.2%，分不出胜负
bare_thread	4	10000	0.96x	两侧差异 4.1% 小于带宽 4.7%，分不出胜负
apscheduler	16	40	1.09x	两侧差异 8.9% 小于带宽 15.6%，分不出胜负
process_pool	16	40	0.73x	两侧差异 27.1% 小于带宽 29.0%，分不出胜负
apscheduler	64	40	1.07x	两侧差异 6.5% 小于带宽 9.9%，分不出胜负
process_pool	64	40	0.94x	两侧差异 6.4% 小于带宽 16.1%，分不出胜负
apscheduler	64	300	1.02x	两侧差异 2.2% 小于带宽 5.0%，分不出胜负
apscheduler	64	2700	1.00x	两侧差异 0.4% 小于带宽 1.7%，分不出胜负
asyncio_pool	64	10000	0.99x	两侧差异 1.2% 小于带宽 3.9%，分不出胜负
thread_pool	64	10000	0.97x	两侧差异 2.8% 小于带宽 3.7%，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用无锁的 `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
- 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': 43405, 'child_pids': [43869, 43872, 43875, 43878, 43881, 43887, 43893, 43899, 43905, 43923, 43941, 43959, 43978, 44034, 44090, 44145, 44206, 44208, 44210, 44212, 44214, 44219, 44224, 44229, 44234, 44251, 44268, 44285, 44302, 44363, 44425, 44490, 44555, 44589, 44623, 44657, 44691, 44824, 44959, 45092, 45225, 45754, 46283, 46812, 47341, 49454, 51567, 53680, 55794, 55798, 55802, 55806, 55810, 55817, 55824, 55831, 55838, 55857, 55876, 55895, 55914, 55981, 56048, 56115, 56182, 56184, 56186, 56189, 56191, 56196, 56201, 56206, 56211, 56228, 56245, 56262, 56279, 56335, 56392, 56453, 56519, 56521, 56523, 56525, 56527, 56532, 56537, 56542, 56547, 56564, 56581, 56598, 56615, 56664, 56713, 56758], '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	288.52 微秒	319.90 微秒	323.17 微秒	0.053
apscheduler	1	300	542.76 微秒	584.20 微秒	596.60 微秒	0.042
apscheduler	1	2700	2.999 毫秒	3.199 毫秒	3.209 毫秒	0.027
apscheduler	1	10000	9.971 毫秒	10.397 毫秒	11.362 毫秒	0.034
apscheduler	4	40	256.36 微秒	278.20 微秒	299.54 微秒	0.047
apscheduler	4	300	511.06 微秒	526.87 微秒	571.99 微秒	0.030
apscheduler	4	2700	2.983 毫秒	3.051 毫秒	3.097 毫秒	0.012
apscheduler	4	10000	10.258 毫秒	10.480 毫秒	11.282 毫秒	0.025
apscheduler	16	40	224.01 微秒	265.51 微秒	278.59 微秒	0.084
apscheduler	16	300	505.14 微秒	532.88 微秒	551.13 微秒	0.026
apscheduler	16	2700	2.980 毫秒	3.025 毫秒	3.042 毫秒	0.010
apscheduler	16	10000	9.881 毫秒	9.964 毫秒	9.988 毫秒	0.008
apscheduler	64	40	220.85 微秒	246.89 微秒	281.47 微秒	0.068
apscheduler	64	300	501.75 微秒	548.43 微秒	550.58 微秒	0.040
apscheduler	64	2700	2.965 毫秒	3.002 毫秒	3.006 毫秒	0.010
apscheduler	64	10000	10.167 毫秒	10.487 毫秒	10.506 毫秒	0.014
asyncio_pool	1	40	140.37 微秒	165.56 微秒	181.10 微秒	0.083
asyncio_pool	1	300	386.47 微秒	424.78 微秒	434.41 微秒	0.046
asyncio_pool	1	2700	2.752 毫秒	2.835 毫秒	2.987 毫秒	0.023
asyncio_pool	1	10000	9.482 毫秒	9.593 毫秒	9.606 毫秒	0.007
asyncio_pool	4	40	173.04 微秒	188.13 微秒	194.49 微秒	0.098
asyncio_pool	4	300	407.48 微秒	430.31 微秒	434.63 微秒	0.025
asyncio_pool	4	2700	2.755 毫秒	2.843 毫秒	2.872 毫秒	0.020
asyncio_pool	4	10000	9.867 毫秒	10.018 毫秒	10.030 毫秒	0.007
asyncio_pool	16	40	134.08 微秒	145.26 微秒	147.06 微秒	0.048
asyncio_pool	16	300	393.00 微秒	410.43 微秒	413.08 微秒	0.025
asyncio_pool	16	2700	2.802 毫秒	2.880 毫秒	2.901 毫秒	0.014

方案	并发度	执行体档位（微秒）	中位数	p95	p99	相对离散度
asyncio_pool	16	10000	9.935 毫秒	10.005 毫秒	10.035 毫秒	0.006
asyncio_pool	64	40	120.26 微秒	127.13 微秒	155.48 微秒	0.075
asyncio_pool	64	300	384.40 微秒	394.23 微秒	419.68 微秒	0.023
asyncio_pool	64	2700	2.811 毫秒	2.843 毫秒	2.879 毫秒	0.008
asyncio_pool	64	10000	10.407 毫秒	10.532 毫秒	10.737 毫秒	0.018
bare_thread	1	40	148.91 微秒	191.86 微秒	265.16 微秒	0.191
bare_thread	1	300	384.68 微秒	418.48 微秒	436.19 微秒	0.043
bare_thread	1	2700	2.672 毫秒	2.738 毫秒	2.766 毫秒	0.011
bare_thread	1	10000	9.648 毫秒	9.807 毫秒	9.991 毫秒	0.011
bare_thread	4	40	199.87 微秒	214.91 微秒	218.90 微秒	0.069
bare_thread	4	300	419.42 微秒	470.82 微秒	476.46 微秒	0.050
bare_thread	4	2700	2.706 毫秒	2.777 毫秒	2.816 毫秒	0.013
bare_thread	4	10000	9.696 毫秒	10.520 毫秒	11.286 毫秒	0.040
bare_thread	16	40	167.78 微秒	174.20 微秒	174.64 微秒	0.021
bare_thread	16	300	424.88 微秒	436.42 微秒	439.49 微秒	0.012
bare_thread	16	2700	2.711 毫秒	2.736 毫秒	2.749 毫秒	0.005
bare_thread	16	10000	9.609 毫秒	9.734 毫秒	10.228 毫秒	0.016
bare_thread	64	40	171.52 微秒	175.30 微秒	176.12 微秒	0.015
bare_thread	64	300	424.30 微秒	431.60 微秒	433.02 微秒	0.009
bare_thread	64	2700	2.675 毫秒	2.707 毫秒	2.771 毫秒	0.009
bare_thread	64	10000	9.827 毫秒	9.887 毫秒	10.032 毫秒	0.006
process_pool	1	40	333.53 微秒	361.85 微秒	366.69 微秒	0.049
process_pool	1	300	607.37 微秒	640.03 微秒	647.18 微秒	0.030
process_pool	1	2700	3.009 毫秒	3.041 毫秒	3.069 毫秒	0.007
process_pool	1	10000	9.888 毫秒	10.016 毫秒	10.086 毫秒	0.007
process_pool	4	40	168.78 微秒	221.25 微秒	235.10 微秒	0.147
process_pool	4	300	304.29 微秒	365.39 微秒	376.12 微秒	0.099
process_pool	4	2700	1.385 毫秒	1.723 毫秒	1.978 毫秒	0.108
process_pool	4	10000	4.621 毫秒	4.741 毫秒	4.757 毫秒	0.011
process_pool	16	40	149.92 微秒	223.33 微秒	232.08 微秒	0.218
process_pool	16	300	278.51 微秒	302.38 微秒	334.02 微秒	0.087
process_pool	16	2700	1.370 毫秒	1.542 毫秒	1.562 毫秒	0.049
process_pool	16	10000	4.741 毫秒	4.855 毫秒	4.901 毫秒	0.013
process_pool	64	40	194.08 微秒	242.04 微秒	250.15 微秒	0.129
process_pool	64	300	299.76 微秒	331.67 微秒	337.00 微秒	0.052
process_pool	64	2700	1.364 毫秒	1.413 毫秒	1.451 毫秒	0.019
process_pool	64	10000	4.686 毫秒	4.721 毫秒	5.108 毫秒	0.023
thread_pool	1	40	75.77 微秒	96.09 微秒	102.57 微秒	0.119
thread_pool	1	300	322.77 微秒	343.49 微秒	347.40 微秒	0.034
thread_pool	1	2700	2.689 毫秒	2.738 毫秒	2.804 毫秒	0.012
thread_pool	1	10000	9.361 毫秒	9.494 毫秒	9.573 毫秒	0.007
thread_pool	4	40	96.03 微秒	103.48 微秒	104.00 微秒	0.047
thread_pool	4	300	350.77 微秒	366.87 微秒	371.64 微秒	0.021

方案	并发度	执行体档位（微秒）	中位数	p95	p99	相对离散度
thread_pool	4	2700	2.742 毫秒	2.787 毫秒	2.810 毫秒	0.013
thread_pool	4	10000	9.813 毫秒	9.951 毫秒	9.957 毫秒	0.008
thread_pool	16	40	90.21 微秒	96.78 微秒	98.07 微秒	0.042
thread_pool	16	300	343.15 微秒	352.67 微秒	355.58 微秒	0.015
thread_pool	16	2700	2.772 毫秒	2.806 毫秒	2.810 毫秒	0.009
thread_pool	16	10000	9.803 毫秒	9.917 毫秒	10.039 毫秒	0.007
thread_pool	64	40	89.21 微秒	91.13 微秒	92.53 微秒	0.020
thread_pool	64	300	350.35 微秒	353.38 微秒	355.51 微秒	0.006
thread_pool	64	2700	2.696 毫秒	2.730 毫秒	2.736 毫秒	0.008
thread_pool	64	10000	10.244 毫秒	10.415 毫秒	10.559 毫秒	0.016
zoo	1	40	182.57 微秒	223.66 微秒	236.57 微秒	0.111
zoo	1	300	449.17 微秒	484.91 微秒	497.19 微秒	0.042
zoo	1	2700	2.754 毫秒	2.792 毫秒	2.799 毫秒	0.008
zoo	1	10000	10.066 毫秒	10.274 毫秒	10.404 毫秒	0.012
zoo	4	40	206.14 微秒	237.88 微秒	243.81 微秒	0.075
zoo	4	300	485.26 微秒	517.91 微秒	527.92 微秒	0.040
zoo	4	2700	2.946 毫秒	3.007 毫秒	3.025 毫秒	0.016
zoo	4	10000	10.106 毫秒	10.173 毫秒	10.206 毫秒	0.007
zoo	16	40	205.75 微秒	235.19 微秒	256.29 微秒	0.072
zoo	16	300	470.57 微秒	488.80 微秒	491.20 微秒	0.019
zoo	16	2700	2.899 毫秒	2.938 毫秒	2.942 毫秒	0.010
zoo	16	10000	10.084 毫秒	10.217 毫秒	10.270 毫秒	0.008
zoo	64	40	207.28 微秒	216.84 微秒	219.30 微秒	0.031
zoo	64	300	491.01 微秒	498.97 微秒	500.32 微秒	0.010
zoo	64	2700	2.953 毫秒	2.994 毫秒	2.997 毫秒	0.008
zoo	64	10000	10.538 毫秒	10.920 毫秒	11.105 毫秒	0.021

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	230.26 微秒	79.81%	58.25 微秒
apscheduler	1	300	243.87 微秒	44.93%	298.89 微秒
apscheduler	1	2700	270.38 微秒	9.02%	2.729 毫秒
apscheduler	1	10000	318.08 微秒	3.19%	9.653 毫秒
apscheduler	4	40	200.17 微秒	78.08%	56.20 微秒
apscheduler	4	300	202.61 微秒	39.64%	308.46 微秒
apscheduler	4	2700	259.91 微秒	8.71%	2.723 毫秒
apscheduler	4	10000	—	—	14.702 毫秒
apscheduler	16	40	—	—	53.41 微秒
apscheduler	16	300	—	—	319.00 微秒
apscheduler	16	2700	—	—	2.736 毫秒
apscheduler	16	10000	—	—	15.686 毫秒
apscheduler	64	40	—	—	50.44 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
apscheduler	64	300	—	—	306.35 微秒
apscheduler	64	2700	—	—	2.714 毫秒
apscheduler	64	10000	—	—	14.852 毫秒
asyncio_pool	1	40	94.16 微秒	67.08%	46.21 微秒
asyncio_pool	1	300	101.41 微秒	26.24%	285.06 微秒
asyncio_pool	1	2700	126.14 微秒	4.58%	2.626 毫秒
asyncio_pool	1	10000	146.32 微秒	1.54%	9.336 毫秒
asyncio_pool	4	40	118.87 微秒	68.70%	54.17 微秒
asyncio_pool	4	300	105.33 微秒	25.85%	302.15 微秒
asyncio_pool	4	2700	122.61 微秒	4.45%	2.632 毫秒
asyncio_pool	4	10000	—	—	12.035 毫秒
asyncio_pool	16	40	—	—	53.76 微秒
asyncio_pool	16	300	—	—	303.75 微秒
asyncio_pool	16	2700	—	—	2.675 毫秒
asyncio_pool	16	10000	—	—	19.300 毫秒
asyncio_pool	64	40	—	—	52.05 微秒
asyncio_pool	64	300	—	—	303.49 微秒
asyncio_pool	64	2700	—	—	2.691 毫秒
asyncio_pool	64	10000	—	—	95.273 毫秒
bare_thread	1	40	95.49 微秒	64.13%	53.41 微秒
bare_thread	1	300	97.47 微秒	25.34%	287.21 微秒
bare_thread	1	2700	107.34 微秒	4.02%	2.564 毫秒
bare_thread	1	10000	120.48 微秒	1.25%	9.528 毫秒
bare_thread	4	40	140.95 微秒	70.52%	58.93 微秒
bare_thread	4	300	118.32 微秒	28.21%	301.10 微秒
bare_thread	4	2700	109.53 微秒	4.05%	2.597 毫秒
bare_thread	4	10000	—	—	14.698 毫秒
bare_thread	16	40	—	—	50.78 微秒
bare_thread	16	300	—	—	300.13 微秒
bare_thread	16	2700	—	—	2.584 毫秒
bare_thread	16	10000	—	—	14.647 毫秒
bare_thread	64	40	—	—	51.21 微秒
bare_thread	64	300	—	—	297.44 微秒
bare_thread	64	2700	—	—	2.542 毫秒
bare_thread	64	10000	—	—	18.793 毫秒
process_pool	1	40	279.58 微秒	83.82%	53.95 微秒
process_pool	1	300	296.81 微秒	48.87%	310.56 微秒
process_pool	1	2700	342.13 微秒	11.37%	2.667 毫秒
process_pool	1	10000	341.21 微秒	3.45%	9.547 毫秒
process_pool	4	40	113.81 微秒	67.43%	54.97 微秒
process_pool	4	300	—	—	577.68 微秒
process_pool	4	2700	—	—	4.801 毫秒
process_pool	4	10000	—	—	17.775 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	16	40	—	—	92.00 微秒
process_pool	16	300	—	—	570.91 微秒
process_pool	16	2700	—	—	9.708 毫秒
process_pool	16	10000	—	—	54.326 毫秒
process_pool	64	40	—	—	93.26 微秒
process_pool	64	300	—	—	583.35 微秒
process_pool	64	2700	—	—	27.749 毫秒
process_pool	64	10000	—	—	173.846 毫秒
thread_pool	1	40	31.35 微秒	41.38%	44.42 微秒
thread_pool	1	300	35.83 微秒	11.10%	286.94 微秒
thread_pool	1	2700	63.24 微秒	2.35%	2.625 毫秒
thread_pool	1	10000	62.33 微秒	0.67%	9.299 毫秒
thread_pool	4	40	46.33 微秒	48.25%	49.70 微秒
thread_pool	4	300	52.33 微秒	14.92%	298.44 微秒
thread_pool	4	2700	78.15 微秒	2.85%	2.663 毫秒
thread_pool	4	10000	—	—	19.045 毫秒
thread_pool	16	40	—	—	47.32 微秒
thread_pool	16	300	—	—	293.68 微秒
thread_pool	16	2700	—	—	2.682 毫秒
thread_pool	16	10000	—	—	20.052 毫秒
thread_pool	64	40	—	—	47.07 微秒
thread_pool	64	300	—	—	294.13 微秒
thread_pool	64	2700	—	—	2.598 毫秒
thread_pool	64	10000	—	—	40.559 毫秒
zoo	1	40	130.82 微秒	71.66%	51.75 微秒
zoo	1	300	144.25 微秒	32.12%	304.92 微秒
zoo	1	2700	155.52 微秒	5.65%	2.599 毫秒
zoo	1	10000	256.30 微秒	2.55%	9.810 毫秒
zoo	4	40	151.02 微秒	73.26%	55.12 微秒
zoo	4	300	176.04 微秒	36.28%	309.23 微秒
zoo	4	2700	226.36 微秒	7.68%	2.720 毫秒
zoo	4	10000	—	—	15.005 毫秒
zoo	16	40	—	—	55.32 微秒
zoo	16	300	—	—	304.71 微秒
zoo	16	2700	—	—	2.680 毫秒
zoo	16	10000	—	—	19.345 毫秒
zoo	64	40	—	—	53.22 微秒
zoo	64	300	—	—	314.88 微秒
zoo	64	2700	—	—	2.722 毫秒
zoo	64	10000	—	—	47.155 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	3466.0
apscheduler	1	300	1842.4
apscheduler	1	2700	333.4
apscheduler	1	10000	100.3
apscheduler	4	40	15602.9
apscheduler	4	300	7826.8
apscheduler	4	2700	1341.0
apscheduler	4	10000	389.9
apscheduler	16	40	71424.4
apscheduler	16	300	31674.5
apscheduler	16	2700	5369.8
apscheduler	16	10000	1619.2
apscheduler	64	40	289794.7
apscheduler	64	300	127553.4
apscheduler	64	2700	21585.8
apscheduler	64	10000	6294.8
asyncio_pool	1	40	7123.9
asyncio_pool	1	300	2587.5
asyncio_pool	1	2700	363.4
asyncio_pool	1	10000	105.5
asyncio_pool	4	40	23116.5
asyncio_pool	4	300	9816.5
asyncio_pool	4	2700	1451.9
asyncio_pool	4	10000	405.4
asyncio_pool	16	40	119331.0
asyncio_pool	16	300	40712.6
asyncio_pool	16	2700	5711.1
asyncio_pool	16	10000	1610.5
asyncio_pool	64	40	532160.1
asyncio_pool	64	300	166491.4
asyncio_pool	64	2700	22767.3
asyncio_pool	64	10000	6149.5
bare_thread	1	40	6715.5
bare_thread	1	300	2599.6
bare_thread	1	2700	374.3
bare_thread	1	10000	103.6
bare_thread	4	40	20013.0
bare_thread	4	300	9537.0
bare_thread	4	2700	1478.1
bare_thread	4	10000	412.6
bare_thread	16	40	95364.7
bare_thread	16	300	37658.1
bare_thread	16	2700	5901.5
bare_thread	16	10000	1665.2

方案	并发度	执行体档位（微秒）	吞吐（任务/秒）
bare_thread	64	40	373124.9
bare_thread	64	300	150836.5
bare_thread	64	2700	23921.9
bare_thread	64	10000	6512.8
process_pool	1	40	2998.2
process_pool	1	300	1646.4
process_pool	1	2700	332.3
process_pool	1	10000	101.1
process_pool	4	40	23699.2
process_pool	4	300	13145.4
process_pool	4	2700	2888.8
process_pool	4	10000	865.5
process_pool	16	40	106727.0
process_pool	16	300	57448.9
process_pool	16	2700	11678.3
process_pool	16	10000	3375.1
process_pool	64	40	329761.1
process_pool	64	300	213501.9
process_pool	64	2700	46909.7
process_pool	64	10000	13657.9
thread_pool	1	40	13198.4
thread_pool	1	300	3098.1
thread_pool	1	2700	371.9
thread_pool	1	10000	106.8
thread_pool	4	40	41654.8
thread_pool	4	300	11403.4
thread_pool	4	2700	1459.0
thread_pool	4	10000	407.6
thread_pool	16	40	177361.2
thread_pool	16	300	46626.5
thread_pool	16	2700	5772.6
thread_pool	16	10000	1632.1
thread_pool	64	40	717389.6
thread_pool	64	300	182674.1
thread_pool	64	2700	23741.9
thread_pool	64	10000	6247.3
zoo	1	40	5477.4
zoo	1	300	2226.3
zoo	1	2700	363.1
zoo	1	10000	99.3
zoo	4	40	19404.4
zoo	4	300	8242.9
zoo	4	2700	1357.7
zoo	4	10000	395.8

方案	并发度	执行体档位（微秒）	吞吐（任务/秒）
zoo	16	40	77763.4
zoo	16	300	34001.3
zoo	16	2700	5518.7
zoo	16	10000	1586.7
zoo	64	40	308754.3
zoo	64	300	130343.7
zoo	64	2700	21671.4
zoo	64	10000	6073.5