ChatGPT

限流算法指南

限流目的

简而言之，限流可以控制系统对外部请求的处理速度，避免瞬时过载，保障服务公平和稳定运行 [1] [2]。通过对每个调用者或整个服务设置请求上限，可以阻止恶意的拒绝服务攻击并平滑系统负载。

固定窗口计数 (Fixed Window Counter)

原理： 固定窗口算法在固定的时间窗口内维护一个简单计数器。例如每分钟允许 100 次调用，计数器在新窗口开始时重置 [3]。可以按用户或服务器级别分别维护计数器 [4]。

实现方法： 通常为每个维度生成一个包含时间戳的键，如 `api:{userId}:{floor(now/窗口长度)}`。每次请求时对该键执行自增并设置过期时间。计数超过阈值则拒绝直到新窗口到来。

优点与注意事项： 实现简单且避免饥饿，但缺点是窗口边界附近可能出现“突刺”——大量请求在窗口开头集中到来，导致系统瞬间过载 [5]。因此在高并发场景可以结合其它算法缓解突刺问题 [6]。

滑动窗口算法 (Sliding Window)

滑动窗口算法解决了固定窗口突刺的问题，通过记录近期的请求并在任意连续窗口内计算调用次数。

滑动窗口日志（Sliding Window Log）

原理： 对每个键维护一个有序集合（如 Redis ZSET），成员为每次通过的唯一标识，分数为时间戳。处理请求时先删除窗口外的旧元素，然后检查集合长度：如果小于阈值则放行并加入当前时间戳；否则拒绝。这样保证在任意长度为 W 的区间内最多通过 L 次请求。

实现注意： - 成员必须唯一，避免并发时覆盖导致计数失真。 - 删除旧元素和新增元素需要在同一个原子操作中完成，常用 Lua 脚本。 - 每次成功插入时要设置 TTL（窗口大小），避免冷键长时间占用内存。

滑动窗口计数（Rolling Window / 分片桶）

该近似算法将大窗口拆成多个小桶（如 60 秒窗口拆为 12 个 5 秒小桶）。请求时对当前桶计数，并对最近 N 个桶求和判断是否超限。它牺牲了时间精度换取更好的性能，适合流量极高的场景。

优缺点： 滑动窗口相较于固定窗口更加公平，并能够处理突刺 [7]。但需要记录每个调用者的历史请求，内存消耗较大，分布式环境需要特别注意键设计、时钟一致性和热点问题 [8]。

漏桶算法 (Leaky Bucket)

原理： 漏桶算法将请求看成水滴放入桶中，桶底有一个孔按固定速率“漏水” [9]。桶满时新来的水滴会溢出，即拒绝请求，保证输出流量平稳。

实现方法： 常用一个固定长度的队列表示桶，进来的请求入队，定期以固定速率出队处理 [10]。如果队列已满则直接丢弃新请求 [11]。

注意事项： 漏桶可以平滑流量并保护下游，但不能适应突发高峰，队列过长会导致排队时间增加甚至饥饿 [12]。

令牌桶算法 (Token Bucket)

原理： 与漏桶相反，令牌桶向桶中定期加入令牌，每个请求消耗一定数量的令牌 [13]。桶容量有限，令牌用尽时请求必须等待新令牌生成；桶满时新增的令牌会被丢弃 [14]。该算法允许请求在有足够令牌时突发发送，而长期平均速率由令牌生成速率决定 [15]。

实现方法： 为每个键存储当前令牌数和最近一次补充时间。每次请求先根据时间差计算补充多少令牌并更新状态，然后判断令牌是否足够。若足够则扣减并通过，否则返回等待时间。

注意事项： 相比漏桶，令牌桶支持短时间突发，但可能在一个时间窗内处理的请求超过平均速率，因此要合理选择桶容量和补充速率 [16]。

并发数限流 (Concurrency Limiting)

原理： 并发限制器不关注时间维度，而是限制同时运行的任务数量。某博客指出，**并发限流器只允许同时存在 n 个任务**，与秒级 QPS无关 [17]。实现上常用信号量（semaphore），初始包含固定数量的“许可”，每个任务开始时需要取得许可，用完后释放 [18]。

实现方法： - **本地信号量：** 在单进程内用线程库提供的 semaphore 控制并发，简单但各实例之间不共享状态。 - **分布式信号量：** 使用 Redis 等存储保存可用许可数量，结合 Lua 保证原子性。请求时减少许可，任务结束时归还许可。没有可用许可时请求必须等待或被拒绝。 - 一些框架（如 Hangfire）提供内置的并发限制器，允许用户定义特定任务的最大并发数 [19]。

注意事项： 并发限制器侧重保护下游资源，如数据库连接或线程池。需要注意确保许可最终归还，避免“忘记释放”导致永久占用 [20]。在公平性要求较高时，可能需要排队机制来避免“后来者插队”。

组合与实践注意事项

- **键设计：** 限流键应包含业务维度（接口、用户、IP、租户、热点参数等），以便精确控制不同流量。避免使用全局单键导致热点。
- **原子操作：** 在分布式存储中，删除旧数据、计数和插入必须在同一原子操作中完成，常用 Redis 的 Lua 脚本实现。
- **TTL 与冷键回收：** 为每个限流键设置合理的过期时间，通常等于窗口大小或桶填充时间，以防止长期不活跃的键占用内存。
- **时钟一致性：** 多实例环境下应尽量使用统一时间源，避免因时钟漂移导致的限流误判。
- **系统分层：** 实际应用中通常在网关进行粗粒度限流（例如基于 IP）、在服务内部针对接口或参数细分限流，再对下游资源设置并发限制，多层结合更可靠。
- **监控与调优：** 应监控放行/拒绝次数、等待时间、错误率等指标，根据业务压力动态调整窗口大小、令牌速率或并发上限。

总结

常见限流算法各有适用场景：固定窗口实现简单但易受突发流量影响；滑动窗口能够提供平滑、公平的限制，但占用内存较高；令牌桶适合允许短暂突发又维持长期均速；漏桶提供恒定输出，适用于需要严格控制下游速率的场景；并发数限制则保护线程池或数据库等有限资源。选择合适的限流策略并结合系统特性进行分层设计，是保证服务稳定性的重要手段。

---

1 2 6 11 12 What is Rate Limiting? | TIBCO
https://www.tibco.com/glossary/what-is-rate-limiting

3 4 5 7 8 9 10 13 14 15 16 Davide's Code and Architecture Notes - 4 algorithms to implement Rate Limiting, with comparison | Code4IT
https://www.code4it.dev/architecture-notes/rate-limiting-algorithms/

17 18 20 Exploring concurrent rate limiters, mutexes, semaphores | Shalvah's Blog
https://blog.shalvah.me/posts/diving-into-concurrent-rate-limiters-mutexes-semaphores

19 Concurrency & Rate Limiting — Hangfire Documentation
https://docs.hangfire.io/en/latest/background-processing/throttling.html