arXiv:2602.11741v1 [cs.DC] 12 Feb 2026

设计可扩展的速率限制系统：算法、架构与分布式解决方案

Bo GuanWynerTech Solutions
billy.guan@wynertech.com

摘要

设计一个兼具准确性、可用性和可扩展性的速率限制器，是分布式系统中的一项根本挑战，主要源于算法精度、可用性、一致性和分区容错性之间的权衡。本文介绍了一种面向生产级环境的分布式速率限制系统的具体架构。我们的设计选用内存缓存数据库 Redis 及其有序集合数据结构，以高效、低延迟地对键值对数据集执行 $O(log(N))$ 时间复杂度的操作，同时保持精度。本文的核心贡献在于量化所选滚动窗口（作为实际采用的限流算法）与令牌桶和固定窗口算法之间的准确性与内存成本权衡。此外，我们阐述了服务器端 Lua 脚本对于将清理、计数和插入整合为单个原子操作至关重要，因此能够消除并发环境中的竞态条件。在系统架构方面，我们提出了一种三层架构，用于管理限流规则的存储和更新。通过对规则参数进行哈希并加载脚本，可以在不修改缓存脚本的情况下更改规则。此外，我们还分析了该架构在 Redis 集群上的部署方式；借助数据分片和复制，Redis 集群能够提供可用性和可扩展性。我们说明了在这一用例中，接受 CAP 定理中的 AP（可用性与分区容错性）是务实的工程权衡。

关键词：速率限制、Redis 有序集合、滚动窗口算法、分布式系统、原子性、Lua 脚本、Redis 集群、CAP 定理、可扩展性。

1 引言

速率限制是分布式系统中调节网络流量的一种基本模式，是安全、经济和

性能 [1, 2]的关键控制机制。设计一个同时具备准确性、可用性和可扩展性的速率限制器，是分布式系统中的一项根本性挑战[3, 4]。问题不仅在于统计请求数量，还在于如何在 CAP 定理的约束下构建有状态协调服务：在分区环境中，对高可用性的需求与强一致性保证相冲突 [5]。尽管令牌桶和固定窗口等算法已广为人知，但将其应用于分布式、高并发生产系统时，会引入一些关键却鲜少被讨论的工程问题：并发写入期间的竞态条件、内存与精度权衡 [6], , 以及在节点之间共享状态时可用性与一致性之间的两难。

本文提出了一种面向生产环境的分布式速率限制服务的具体架构。我们的核心贡献如下：首先，我们明确分析了该架构在 Redis 集群上的部署，并指出，对于这一用例而言，优先考虑可用性与分区容错性 (AP)，而非强一致性，是一种务实的工程权衡。其次，我们量化了各类限流算法的内存使用量，并倾向于采用滚动窗口算法，因为它能以较低的内存成本提供精确控制。第三，我们详细说明了服务器端 Lua 脚本编写为何至关重要：它能将清理、计数和插入整合为单个原子操作，从而消除竞态条件。我们的设计以通过 Lua 脚本实现的滚动窗口算法为核心，该算法使用 Redis 有序集合，在服务器端占用较少内存（每个请求 8 字节），并提供 $O(log(N))$ 操作，以便高效管理状态。

设计可用于生产环境的速率限制器，需要在分布式系统中的基本权衡之间做出取舍。如图 1 所示，我们的设计明确选择了两个主要权衡空间：（1）分布式系统所受的 CAP 定理约束。出于务实的工程考量，我们在速率限制场景中明确选择可用性与分区容错性（AP），而非强一致性（详见第 4.3 节）；（2）算法权衡空间。与令牌桶和固定窗口方案相比，我们选择滚动窗口算法，以在时间窗口精度和内存效率之间实现最佳平衡（详见第 2 节）。这些决策通过 Redis 有序集合和 Lua 脚本来实现原子操作，并通过 Redis 集群实现 AP 部署模型，从而达到高精度、低内存使用量、高可用性和可扩展性的目标。

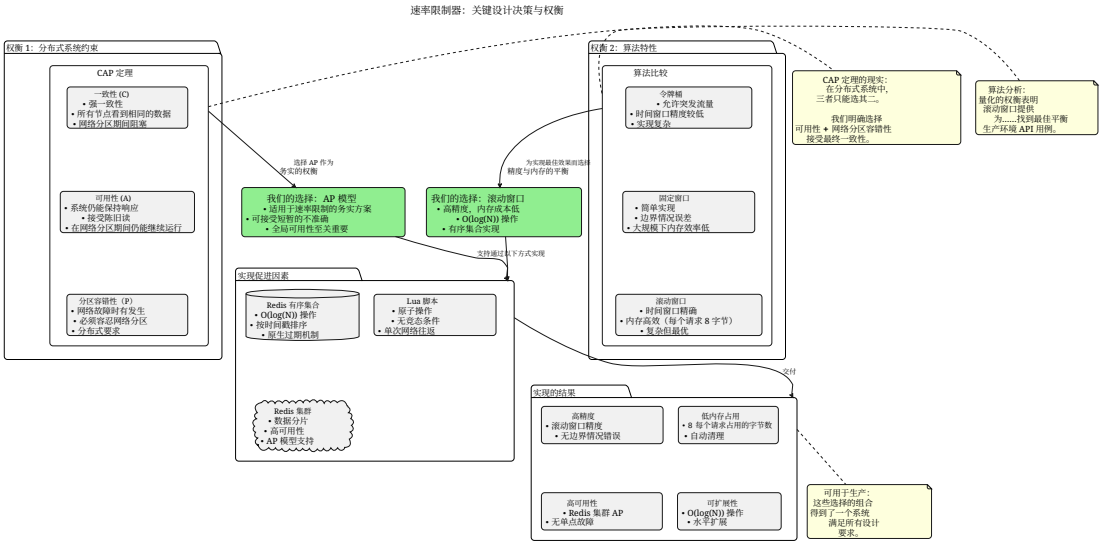

图 1：分布式速率限制器架构中的设计决策与权衡

不过，限流器并非适用于所有情况。正如 Stripe 工程团队所指出的，当用户能够调整调用 API 端点的频率而不影响结果时，限流器才适用[7]。因此，限流器非常适合允许重试或延迟请求的 API。相反，对于无法调整请求频率的实时事件（例如实时通知、金融行情或游戏操作），简单的限流器并不足够——这类场景通常需要扩展基础设施容量，或实施专门的排队和负载削减策略。本文聚焦于前一种场景，即高流量 API 的使用场景，探讨如何设计可扩展的限流器。限流器通过在特定时间窗口内限制用户、服务或 IP 地址可发出的请求数量来发挥作用。它能保护后端资源，避免其因流量峰值或拒绝服务（DoS）等恶意攻击而不堪重负，从而确保资源可用性。这种控制不仅是技术保障，也是核心业务要求。从财务角度来看，它可以防止过度使用付费的第三方 API 和云服务，从而直接管理运营成本。此外，它还能防止任何单一参与者垄断系统容量并损害其他用户的体验，确保所有用户都能获得公平且一致的服务质量。总而言之，限流器具有以下用途：

- **安全性：**防止资源耗尽和拒绝服务（DoS）攻击。
- **成本控制：**管理来自外部 API 和基础设施的费用。

- **公平性：**确保资源和服务质量得到公平分配。

在高规模环境中构建稳健的速率限制器，需要应对一系列复杂的工程权衡。设计必须同时解决持续高写入负载下的性能、可靠性和可扩展性挑战。核心要求是在极低延迟下做出准确的限流决策，避免显著增加 HTTP 响应时间。同时，还必须在服务器组成的分布式集群中维持内存效率和状态一致性；在多线程、多进程环境中，这并非易事。系统本身必须具备高可用性，能够抵御部分故障，并具有足够的灵活性，支持各种限流规则（例如按 IP、用户 ID 或 API 端点进行限制），且能适应规则变更，而无需进行结构性改动。最后，系统必须向客户端提供清晰的说明，在请求被拒绝时返回恰当的错误消息。总之，分布式速率限制器的设计要求如下：

- **准确性高：**做出准确的限流决策。
- **低延迟且内存效率高：**尽可能减少对请求响应时间的影响。
- **分布式协调与并发处理：**跨多个进程处理请求的并发。
- **高可用性与灵活性：**能够从故障中恢复，并适应各种不断变化的限流规则。
- **与客户端清晰沟通：**明确反馈受限请求。

Hayes 等人 [8] 展示了使用 Redis 有序集合进行速率限制。本文在此基础上进一步提出面向生产环境的贡献：对内存与准确性之间的权衡进行定量分析，构建三层动态规则管理架构，并明确论证在分布式环境中部署 AP 模型所涉及的 CAP 定理权衡。

本文结构如下。第 2 节介绍速率限制算法及其权衡所需的背景知识。第 3 节介绍系统架构，以及使用 Lua 脚本实现 Redis 有序集合的具体方法。第 4 节分析分布式环境中的挑战，包括采用 AP 模型的理由以及使用 Redis 集群进行扩展。第 5 节讨论性能方面的考量。最后，我们在第 6 节总结全文。

2 算法基础

本节将介绍实现速率限制的基础算法，并介绍不同的速率限制器方法：令牌桶

算法、固定窗口计数器和滚动窗口计数器。此外，在第 4.2 节中，我们将说明并发请求限制器，以解决存在多个并发请求进程的情况。我们将介绍每种算法的原理及其适用场景。

2.1 令牌桶算法

令牌桶算法已广泛用于速率限制。每个用户都有一个装有令牌的桶。用户桶中的令牌数量在开始时由容量 初始化；每当用户发起请求时，都会根据补充速率乘以距存储的上次补充时间戳 $last_refill$ 的时间间隔来补充令牌，但令牌数量始终不得超过容量。如果桶中有足够的令牌，就会处理用户的请求，并扣除一个令牌（对于开销更大的操作，则扣除多个令牌）。如果令牌不足以处理请求，则丢弃该请求。算法总结如下方算法 1 所示。

算法 1令牌桶速率限制（内存中）

初始化 $capacity \leftarrow capacity$, $rate \leftarrow rate$, $user_id \leftarrow user_id$
$bucket_key \leftarrow$“令牌 _桶：”$+ user_id$
$buckets \leftarrow \{\}$ ▷ 全局字典：键 $\rightarrow$ (令牌, last_refill)
过程allowRequest($tokens_required$)
    如果 $bucket_key \notin buckets$ **则**
        $buckets[bucket_key] \leftarrow (capacity,\ now())$
    结束条件
    $(tokens,\ last_refill) \leftarrow buckets[bucket_key]$
    $current_time \leftarrow now()$
    $time_passed \leftarrow current_time - last_refill$
    $refill_tokens \leftarrow time_passed \times rate$
    $tokens \leftarrow \min(capacity, tokens + refill_tokens)$
    $last_refill \leftarrow current_time$
    如果 $tokens \geq tokens_required$ **那么**
        $tokens \leftarrow tokens - tokens_required$
        $buckets[bucket_key] \leftarrow (tokens,\ last_refill)$
        返回 true
    else
        $buckets[bucket_key] \leftarrow (tokens,\ last_refill)$
        返回 false
    结束条件判断
结束过程

令牌桶算法内存高效，适合处理突发流量，仅当桶中剩余足够令牌时才允许用户请求。该算法有两个调优参数（$capacity, rate$），易于实现。例如，Amazon API 网关对 re- 进行限流

quests 至 API 服务器时使用令牌桶算法 [9]。Stripe 也使用令牌桶算法进行速率限制 [7]。

2.2 固定窗口计数器

固定窗口计数器是一种算法，用于统计用户在每个固定长度时间窗口内的请求频率。该算法能够高效地统计每位用户的请求；然而，它无法处理窗口边界处的突发请求，因为边界附近两个时间窗口内的请求总数可能超过允许的上限。

算法 2固定窗口计数器（内存中）

初始化 $window_size \leftarrow window_size$, $max_requests \leftarrow max_requests$, $user_id \leftarrow user_id$
  $window_key \leftarrow$“固定 _窗口：”$+ user_id$
  $windows \leftarrow \{\}$ ▷ 全局字典：键 $\rightarrow$ (计数，窗口_开始)
过程allowRequest
    $current_time \leftarrow now()$
    $current_window \leftarrow \lfloor current_time / window_size \rfloor$
    $window_start \leftarrow current_window \times window_size$
    如果 $window_key \notin windows$ **，则**
        $windows[window_key] \leftarrow (0, window_start)$
    结束 if
    $(count, stored_window) \leftarrow windows[window_key]$
    如果 $stored_window < window_start$ **那么**
                                新的时间窗口 , 重置计数器
        $count \leftarrow 0$
        $stored_window \leftarrow window_start$
    结束 if
    如果 $count < max_requests$ **那么**
        $count \leftarrow count + 1$
        $windows[window_key] \leftarrow (count, stored_window)$
        返回真
    else
        返回假
    结束条件判断
结束过程

2.3 滚动窗口计数器

滑动窗口计数器算法通过维护一个持续滑动的时间窗口，克服了固定窗口计数器在窗口边界处容易遭受请求突发的弱点。该方法由 Hayes [8], 提出，通过跟踪各个请求的时间戳，实现对任意 T 秒区间内请求数量的精确计数，而非采用与日历时间对齐的固定窗口。通过动态评估

仅统计最近 T 秒内发生的请求，该算法可防止用户绕过基于边界的 ratelimit，同时确保准确执行时间窗口强制执行，具体形式化定义见算法 3。

算法 3滚动窗口计数器（内存中）

初始化 $window_size \leftarrow window_size$, $max_requests \leftarrow max_requests$, $user_id \leftarrow user_id$
$log_key \leftarrow$"滚动 _窗口" $+ user_id$
$request_logs \leftarrow \{\}$ ▷ 全局字典：键 $\rightarrow$ 时间戳列表
过程allowRequest
    $current_time \leftarrow now()$
    $window_start \leftarrow current_time - window_size$
    如果 $log_key \notin request_logs$ **则**
        $request_logs[log_key] \leftarrow [\,]$
    结束 if
    $timestamps \leftarrow request_logs[log_key]$
                                ▷移除当前窗口之外的时间戳
    $i \leftarrow 0$
    当 $i < \text{len}(timestamps)$ **且** $timestamps[i] < window_start$ **时**
        $i \leftarrow i + 1$
    结束 while
    $timestamps \leftarrow timestamps[i:]$
    如果 $\text{len}(timestamps) < max_requests$ **则**
        $timestamps$.追加($current_time$)
        $request_logs[log_key] \leftarrow timestamps$
        返回 真
    else
        返回 假
    结束 if
结束过程

时间戳数据可以存储在内存缓存中以优化性能，例如 Redis（远程字典服务器），它是一种开源的内存中 NoSQL 键值缓存数据存储，且在物理上距离应用程序更近 [10]。具体来说，我们可以使用 Redis 有序集合这种数据结构 [8], ，其中数据按 SCORE（例如时间戳）排序。当有新请求到来时，可以通过命令 ZREMRANGEBYSCORE 删除过期的时间戳数据，例如当前窗口之外的先前请求。在请求处理结束时，还可以通过命令 EXPIRE 为键设置自动超时，以防没有后续请求。因此，有序集合具有较高的内存效率。在分布式环境中，Redis 通过算法 4 为竞态条件预防提供原子实现 [8]，以处理并发请求 [1], 。有关并发处理的更多细节将在第 4.1 节中讨论。

[1]这里的"并发"指来自不同用户的同时请求。处理并-

此外，我们会将上一个请求与当前时间戳进行比较。如果两者间隔过短，也不允许执行该操作。要获取上一个请求的时间戳，我们在第 15 行使用 ZREVRANGE 命令。该命令会返回指定范围内的成员列表（此处为用户 ID）；如果使用 WITHSCORE，还可以返回这些成员对应的分值，以便排序（此处为时间戳） [11]。这些操作的时间复杂度为： $O(log(N) + M)$，其中 N 是有序集合中的元素数量， M 是返回的元素数量。将请求作为有序集合的新成员添加时，ZADD 的时间复杂度为 $O(log(N))$。使用 Redis 数据存储也能满足分布式系统的扩展需求，因为数据可以由多个用户或进程访问；相关内容将在第 4 节中介绍。

算法 4 使用 Redis 有序集合实现滚动窗口 1：初始化

$window_size \leftarrow window_size$ ▷ **滚动窗口大小（秒） 2：初始化** $TTL \leftarrow TTL$ ▷ **超时后移除单个过期请求所需的时间（秒） 3：初始化** $max_requests \leftarrow max_requests$ ▷ **每个窗口的最大请求数 4：初始化** $min_interval \leftarrow min_interval$ ▷ **任意两个有效请求之间的最小间隔（秒） 5：** $redis_key \leftarrow$ **“rate_limiter:”** $+ user_id$ **6：过程 allowRequest 7：** $now \leftarrow now$ **() 8：** $window\ \ start \leftarrow now - window\ \ size$ _ _ **9：▷ Redis 原子操作 10：** $redis.ZREMRANGEBYSCORE(redis_key, -\infty, window_start)$ ▷**主动清理滚动窗口之外的成员，以便准确计数 11：** $current_count \leftarrow redis.ZCARD($ $redis_key$ **) 12：如果** $current_count \geq max_requests$ **则 13：返回 false** ▷ **窗口内请求过多 14：结束 if 15：** $last_request \leftarrow redis.ZREVRANGE($ $redis_key$ **, 0, 0, WITHSCORES) 16：如果** $last_request$ **存在且 (** $now - last_request[1]$ **)** $< min_interval$ **则 17：返回 false** ▷ **请求间隔过短 18：结束 if 19：** $redis.ZADD(redis_key, now, now)$ **20：** $redis.EXPIRE($ $redis_key, TTL)$ ▷ **为键设置超时 21：返回 true 22：结束过程**

总而言之，我们选择内存中的 Redis 作为数据存储，用于存储

同一用户在完全相同的时刻（例如系统分辨率所允许的毫秒级时间精度）发起的当前请求将在第 4.2 节中讨论。届时，我们可以为每个进程生成一个新的随机 ID。例如，在算法 4 的第 19 行代码中，只需将第三个参数“now”替换为生成的 ID，将其作为并发请求的一个成员，类似于 C++ 语言中的多重集（已排序）数据结构。

数据，而不是使用基于磁盘的存储引擎，原因如下：

- 内存数据库速度更快，因为它们无需将内存数据结构编码为可写入磁盘的形式并管理操作系统缓存，从而避免了相关开销 [12][2]。

- Redis 提供了一种有序集合数据结构，它针对简单实现进行了进一步优化，但难以通过基于磁盘的索引实现 [13]。

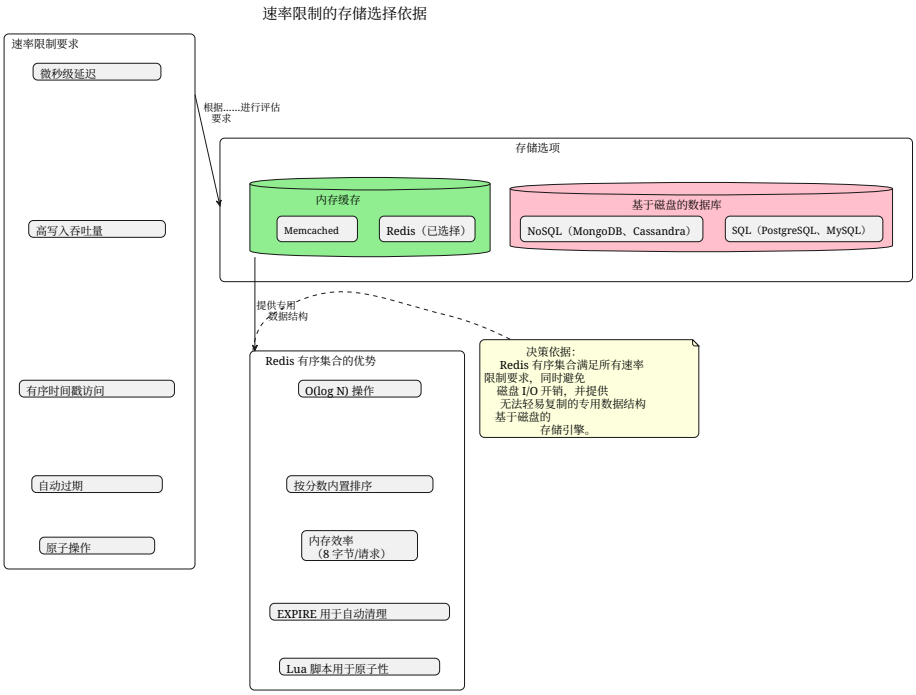

图 2：速率限制实现的存储选择依据。该图展示了根据速率限制需求评估各项存储选项的过程，并最终选择 Redis 有序集合，因为它支持 O(log N) 操作、内置时间戳排序、内存效率高，并可通过 Lua 脚本实现原子执行；这些特性不易通过基于磁盘的数据库复制。

2.4 比较分析

只要有可用令牌，令牌桶算法就允许请求通过，并以固定速率补充令牌，从而实现可配置且可预测的突发容忍能力——这对于应对自然的用户行为十分理想。固定窗口

[2]内存数据库提升性能，并非因为它们无需从磁盘读取。如果操作系统拥有足够的内存，并且无论如何都会将最近使用的磁盘块缓存在内存中，那么基于磁盘的存储引擎也可能永远不需要从磁盘读取 [13]。

计数器算法会统计每个时间窗口内的请求，但会受到边界突发问题的影响：当请求跨越窗口边界时，允许的请求数量可能达到限制的两倍。滚动窗口算法通过精确跟踪时间戳来解决这一问题，但需要更多内存，因为它为每个请求单独存储时间戳，而不是只使用一个计数器，同时还会增加清理开销。并发限制器跟踪的是活动请求，而非请求速率；它会限制同时执行的操作数量，以防资源耗尽，适用于数据库连接、文件上传或耗时较长的 API 调用等必须控制并行执行的场景。并发限制器的详细内容将在第 4.2 节讨论。

令牌桶算法具有较高的内存效率，只需存储两个双精度浮点数——令牌数和上次补充时间——每位用户占用 16 字节，而且令牌会自动补充，无需主动清理。固定窗口计数器每位用户需要 16 字节（一个 8 字节计数器和一个 8 字节窗口起始时间戳，此处均为双精度数[3]），但需要定期处理窗口过期。滑动窗口算法会为每个请求存储一个 8 字节的双精度时间戳，每位用户恰好占用 $8L$ 字节，其中 L 是请求限额，并且需要主动清理过期时间戳。并发限制器会为每个活动请求存储时间戳和请求标识符，每位用户占用 $16C$ 字节，其中 C 是并发限额，并利用基于 TTL 的自动过期机制。对于一百万名用户，在典型限额（$L = 100$、 $C = 50$）下，滑动窗口需要 800 MB，令牌桶需要 16 MB，固定窗口需要 16 MB，而并发限制器需要 800 MB。尽管滑动窗口算法和并发限制器（基于滑动窗口）比令牌桶或固定窗口方案需要更多内存，但这种权衡可以实现精确的时间窗口强制执行；对于 API 速率限制而言，这至关重要，因为边界突发保护不可或缺。

算法	准确性	内存	突发处理	主要用例
令牌桶	中等	Low	优秀	API 速率限制
固定窗口	Low	Low	Poor	简单限流
滚动窗口	High	High	Good	精确速率控制
并发限制器	High	中等	优秀	资源保护

表 1：按特征进行的算法比较。

[3]我们假设计数器和时间戳以 8 字节双精度浮点数的形式存储；实际内存使用量可能会因其内部结构而略有差异。

算法	每用户内存	总计（100万用户）
令牌桶	16 字节	16 MB
固定窗口	16 字节	16 MB
滚动窗口	$8L$ 字节	800 MB
并发限制器	$16C$ 字节	800 MB

表 2：采用 Redis 存储开销时的内存（适用于滚动窗口和并发限制器）。100 万用户，采用典型限额（$L = 100$、 $C = 50$）。

本文使用 Redis 有序集合实现滑动窗口计数器算法，在应对分布式系统挑战的同时，实现精确的时间窗口强制执行。该实现通过 Lua 原子脚本解决并发问题，通过 Redis 集群的 AP 模型权衡可用性与一致性，并支持水平扩展；具体内容详见第 3 节（系统架构）和第 4 节（分布式系统分析）。

3 系统架构

我们通过部署多个服务器节点、Redis 集群和多个限流器实例，设计了一个高可用可扩展限流系统。如图 3 所示，来自 Web、移动端和 API 的客户端请求经负载均衡后分发到多个限流器实例；这些实例通过 Redis 集群（包含多个 Redis 节点，更多详情见第 4.3 节）进行协调，并使用有序集合数据结构和 Lua 脚本执行原子操作（更多详情见第 4.2 节）。当某个服务器节点发生故障时，该设计可通过其副本进行替换（更多详情见第 5.2 节），确保限流决策仍能保持可用。

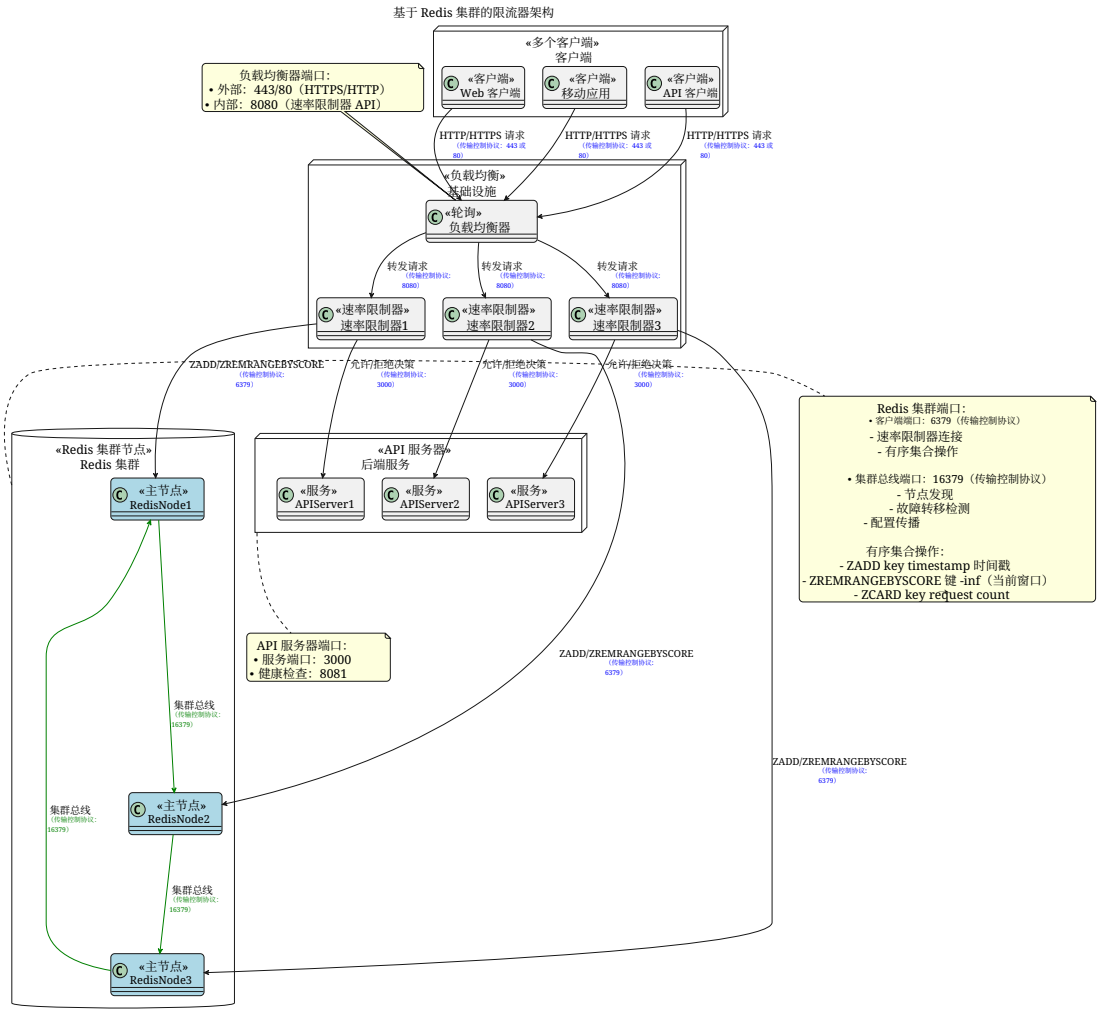

图 3：使用 Redis 有序集合进行分布式协调和原子操作的限流器系统架构。

图 4 所示的请求流程序列展示了速率限制检查的原子性：每个请求都会触发一系列协调执行的 Redis 操作，包括使用 ZREMRANGEBYSCORE 清除过期条目、使用 ZCARD 统计滚动窗口内当前的请求数，以及使用 ZADD 记录新请求；所有操作均在单个原子事务中执行，以防止由不同用户（或进程）发起的请求导致竞态条件。将限流中间件与后端 API 服务器分离，可实现独立扩展和故障隔离。该架构通过分布式协调实现真正的滚动窗口限流，同时具备高可用性并能预防竞态条件。基于算法 4 的滚动窗口计数器，它确保

在不受固定窗口方法边界条件漏洞影响的情况下，实现精确执行。

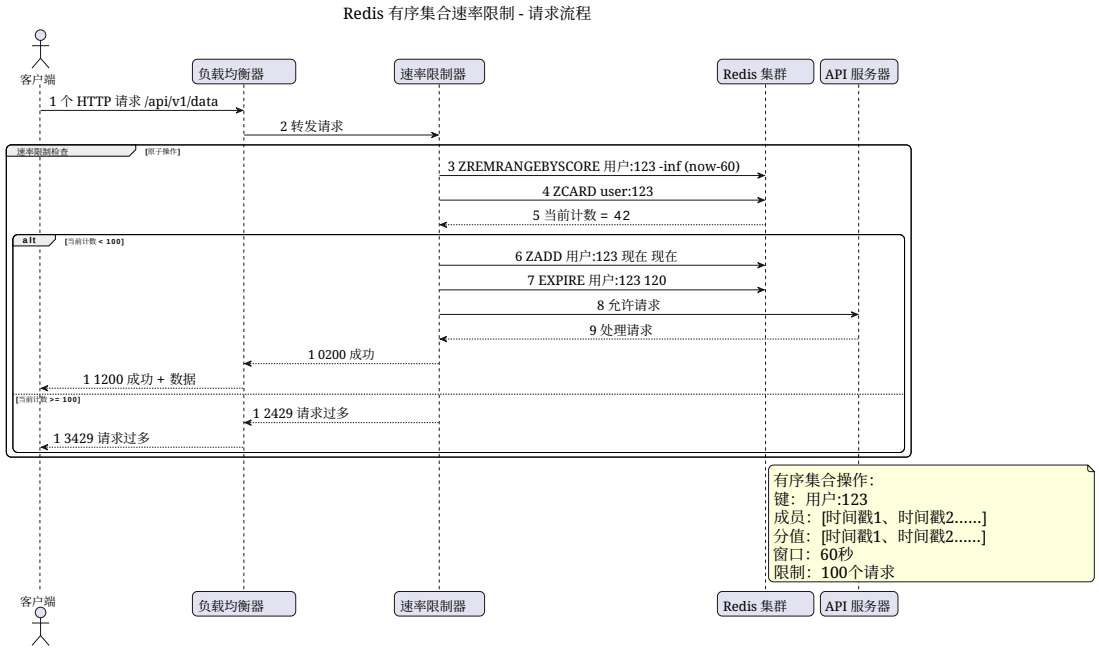

图4：Redis 有序集合限流的请求流程序列，展示原子操作和滚动窗口执行。

3.1 规则管理架构

在介绍了整体系统架构和请求流程序列后，一个关键问题随之而来：如何存储或动态更新速率限制规则，同时避免重新部署脚本？答案在于采用三层架构，将配置、运行时执行和管理分离，如图 5 所示。这种设计能够在不影响服务的情况下动态更新规则，同时保持原子操作和低延迟——这对于规则必须适应不断变化的需求的生产环境至关重要。[4]

[4]如第 2 节所示，为了便于说明，我们的算法和实现示例使用 user_id，但同样的模式也适用于任何区分键：IP 地址、API 端点、客户端应用程序、地理区域、订阅等级或复合键。通过参数化 Lua 脚本可以实现这种灵活性，脚本将键作为参数，从而无需更改代码即可为不同维度设置不同的速率限制。

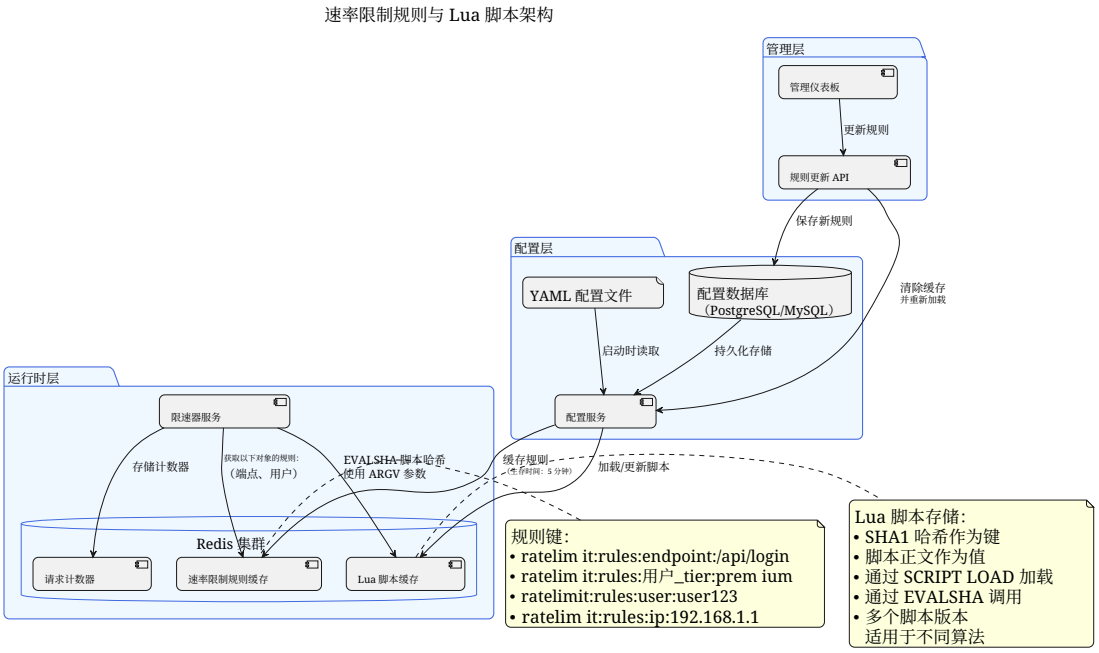

图5：三层限流规则管理架构：配置、运行时执行和管理。

架构层系统通过三个层次实现了清晰的职责分离：

- **配置层：**速率限制规则——定义限额、时间窗口和算法——会持久化存储在数据库中 [14]，并缓存在 Redis 中，同时设置过期 TTL。配置服务会将这些规则加载到 Redis 中，并为每种限流算法预编译 Lua 脚本，再以 SHA1 哈希为键存储，以实现原子执行 [15]。
- **运行时执行层：**限流器服务首先从缓存中获取适用的规则，然后通过 EVALSHA 执行相应的 Lua 脚本，并将规则参数作为参数传入 [16]。这既能确保在服务器端执行原子计数器操作，又能保持灵活性。
- **管理层：**规则更新通过专用 API 和仪表板进行，变更会持久化到数据库，并触发缓存失效。这样既能在不造成服务中断的情况下动态更改规则，又能借助 Redis 缓存保持性能。

关键设计原则 该架构体现了多项重要设计原则，使其具备灵活的运行能力：

1. **规则作为配置：** 速率限制以声明式方式定义，并与可执行代码分开存储，因此无需重新部署即可更新。2. **Lua 脚本作为参数化逻辑：** 脚本充当模板，通过 ARGV 参数接收规则值，在实现原子操作的同时，与规则定义保持分离。3. **关注点分离：** 规则可以独立于脚本进行更改，从而支持对不同算法进行 A/B 测试，并为不同端点或用户层级配置不同规则。4. **双重缓存策略：** 规则和脚本都缓存在 Redis 中以提升性能；脚本则按 SHA1 哈希存储，以便通过 EVALSHA 高效执行。5. **水平可扩展性：** 无状态的速率限制器实例与共享的 Redis 状态相结合，可确保在水平扩展部署中保持行为一致。

这种架构方法带来了多项运维优势：无需更改代码即可实时更新规则；可以同时测试不同的速率限制算法；并通过服务器端 Lua 脚本执行来保证原子性。这些配置模式与 Lyft 的开源速率限制组件中使用的模式相似 [17]，证明了其在生产环境中的可行性。

4 解决分布式系统并发问题的扩展与解决方案

4.1 分布式计数中的竞态条件

我们扩展第 2.3 节中的速率限制器设计，以解决来自同一用户的并发请求问题，即使这些请求具有相同的时间戳（由不同进程发出）也是如此。图 6 展示了这一问题的一个示例：两个请求（可能来自两个不同的用户，也可能来自同一用户的不同进程）同时从服务器读取计数器变量，而在此之前，它们都尚未写回递增后的值，从而导致少计问题：两个请求都需要读取当前计数器值、修改该值，然后将新值写回。在图 6 中，计数器本应从 31 增加到 33；但由于竞态条件，两个请求都读取到计数器值为 31（实际上，其中一个请求已将其更新为 32），并将其“递增”到 32。

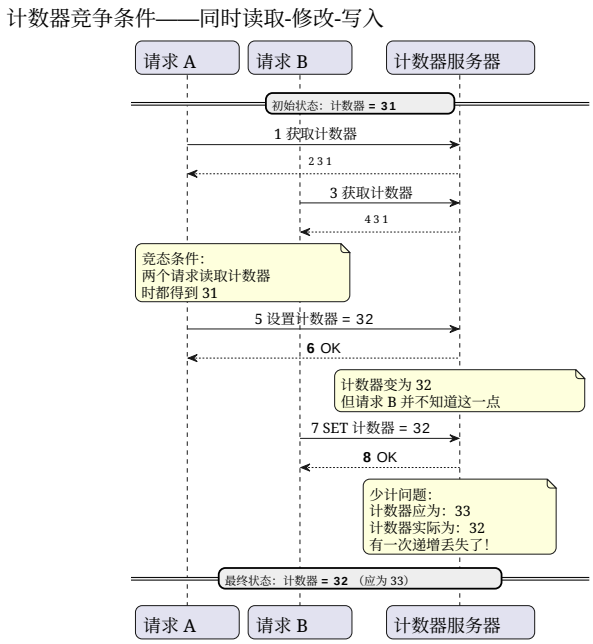

图 6：计数器竞态条件：同时读取-修改-写入周期。

如上例所示，如果请求读取数据库中的值、修改该值，然后写回修改后的值（即读-修改-写周期），就可能发生丢失更新，例如本例中递增计数器。数据库中用于防止竞态条件导致丢失更新的一些现有解决方案包括：

- 一些关系型数据库提供原子操作，例如递增操作。这通常能保证并发安全：读取对象时对其加排他锁，确保在更新提交之前没有其他事务能够读取该对象；或者直接强制所有原子操作都在单线程上执行。然而，这种解决方案有时仍可能不安全，并导致难以调试的错误 [13, 18]。

- 这种比较并交换 方案通过仅在自上次读取以来变量值未发生变化时才允许写入，防止两个客户端同时更新同一个变量。例如，为了防止两个请求同时更新同一用户_ ID 的计数器值，你可以检查服务器开始编辑该计数器时的值是否保持不变。如果该值已发生变化，不再等于“旧值”，那么这次更新就不会生效。因此，你有责任检查更新是否生效，并在必要时重试。不过，这样做是否安全，取决于服务器的数据库 [13] 的实现方式。

- 有些数据库具有自动丢失更新检测功能；一旦检测到丢失更新，就会中止事务，并强制重试读-修改-写周期。

依赖数据库的原子操作和丢失更新检测功能，在某些情况下或许可行。不过，应用程序开发者始终有责任根据手头可用的资源，妥善处理具体问题。接下来，我们将介绍如何通过实现 Redis 服务器的 Lua 脚本来进行原子化设计，从而解决并发问题，并确保速率限制具有低延迟。

现在，我们将这些原子操作原则应用于实现并发请求限制器——这是一种独特的速率限制模式，用于控制请求并发数，而非请求频率。

4.2 并发请求限制器：管理活动连接

与第 2 节中基于频率的速率限制器（用于控制每个时间窗口内的请求数）不同，并发请求限制器控制每位用户 [7] 的**同时活动的请求数量**。它通过限制并行请求并发数来防止资源耗尽。Stripe 的实现采用了与令牌桶速率限制相同的 Redis 有序集合和 Lua 脚本模式，但有一个关键区别：请求到达时将其添加到有序集合中，完成时将其移除，并使用 ZCARD 来强制执行并发限制 [19]。Redis Lua 脚本编写所提供的原子操作——我们在第 4.1 节中介绍了这些操作如何解决竞态条件——对于确保准确统计并发请求同样至关重要。

因此，为了区分各个并发连接，我们需要在**请求标识符**之外，还提供用户 *ID*。这样，系统就能在请求完成时将其从活动集合中移除，而不必等待 TTL 过期。

首先，我们说明如何准确统计并发请求。与算法 4 中的基本基于滚动窗口计数器的限制器类似，我们也采用 Redis 有序集合数据结构，该结构可根据成员的分值进行排序：用户 *ID* 作为键，时间戳作为成员的分值，用于排序和范围操作。我们会主动高效地删除窗口之外的旧请求，即通过 ZREMRANGEBYSCORE 命令删除范围 $(-\infty$，当前时间戳 $- window_size]$ 内的记录)，从而准确统计当前窗口内累计的请求数。这里，我们将请求 *ID* 作为成员；每个用户的每个请求都会生成一个足够长的随机 ID 序列，以保证唯一性：例如，4 个随机字节可生成 2^{32}（超过 42 亿）个可能的数值，因此对于预期使用场景（如最多 50 个并发请求），单个用户发生冲突的概率极低。最后，我们使用 ZREM 命令，将与用户 ID 关联的键的生存时间设置为 TTL 秒。只有当用户完全不活跃时，其键才会在 TTL 秒后过期；否则，每次成功的请求

都会将用户键的生存时间再延长 TTL 秒。因此，该算法通过设置用户键的过期时间来防止内存泄漏。并发请求限制器的示意图如算法 5[5]所示

虽然有序集合数据结构即使在并发情况下也能准确追踪每个用户的请求数量，但我们该如何真正解决并发带来的竞态条件（多个进程同时尝试运行并递增计数器）？为了应对并发请求导致的计数器竞态条件，Redis 能够在服务器端执行 Lua 脚本，并以原子方式执行这些脚本 [20]。我们可以将所有需要在服务器端执行的读写操作放在一起，例如，在 Lua 脚本中使用 ZCARD 和 ZADD 命令来递增计数器，从而将延迟降至最低 [21], 。因此，脚本运行期间，其他脚本都无法运行[6]。

[5]请注意，在此设计中，被阻止的操作仍会计为操作。因此，如果用户发出的请求数持续超过速率限制器的限制，直到自上次有效请求后超时 TTL，其所有操作都将被拒绝。[6]原子操作逻辑（包括算法5中第10至18行的伪代码）可以在服务器端的 Lua 脚本中执行 [19, 22]。

算法5 使用 Redis 有序集合限制并发请求 1: 初始化

$window_size \leftarrow window_size$ ▷ **速率限制窗口大小（秒） 2: 初始化** $TTL \leftarrow TTL$ ▷ **超时后移除单个过期请求的时间（秒） 3: 初始化** $max_requests \leftarrow max_requests$ ▷ **每个用户的最大并发请求数 4: 过程 checkConcurrentRequest(**$user_id$**) 5:** $current_time \leftarrow now$**() 6:** $request_id \leftarrow generateUniqueId$**()** ▷ 4+ **个随机字节 7:** $redis_key \leftarrow$ **“concurrent_limiter:"** $+ user_id$ **8:** ▷**原子清理并计数 9:** $redis.ZREMRANGEBYSCORE$**(**$redis_key, -\infty, current_time - window_size$**)** ▷ **主动清理滚动窗口之外的成员，以准确计数 10:** $current_count \leftarrow redis.ZCARD$**(**$redis_key$**) 11:** $allowed \leftarrow current_count < max_requests$ **12: 如果** $allowed$ **那么 13:** $redis.ZADD$**(**$redis_key, current_time, request_id$**) 14:** $redis.EXPIRE$**(**$redis_key, TTL$**)** ▷**被动清理已弃用的键，防止内存泄漏 15: 返回 (**$allowed$**,** $request_id$**) 16: 否则 17: 返回 (**$allowed, null$**) 18: 结束如果 19: 结束过程 20: 过程 completeRequest(**$user_id, request_id$**) 21: 如果** $request_id$ **不为 null 22:** $redis_key \leftarrow$ **“**$concurrent_limiter$**: ”** $+ user_id$ **23:** $redis.ZREM$**(**$redis_key, request_id$**)** ▷**移除已完成的请求 24: 结束如果 25: 结束过程 26: 过程 doRequest(**$user_id$**,** $businessLogic$**) 27: (**$allowed, req_id$**)** $\leftarrow$ **checkConcurrentRequest(**$user_id$**) 28: 如果非** $allowed$ **那么 29: 引发速率限制错误 30: 结束如果 31:** $businessLogic$**() 32: completeRequest(**$user_id$**,** req_id**) 33: 结束过程**

4.3 可用性与一致性权衡：Redis 集群

分布式方法可以提高系统的可用性和可扩展性。数据分片与复制技术通常结合使用。*Redis*

集群 是 Redis 的分布式版本，它提供了一种运行 Redis 安装的方式，使数据自动分片并复制到多个 Redis 节点 [23], ，从而实现水平扩展并提升性能。Redis 集群通过使用哈希槽 （通过对哈希值取模将键映射到哈希槽）提供读写可扩展性，但不使用一致性哈希。这使得通过调整由 Redis 集群管理员为每个节点确定的预先计算的哈希槽分配（例如均等分配），可以轻松添加和移除集群节点。然而，一致性哈希会将哈希值分配给哈希环上顺时针方向的第一个节点，这可能导致数据倾斜，从而在某个节点发生故障（例如因崩溃或网络分区）时造成键的分布不均衡 [24]。Redis 集群采用领导者-副本复制 模型，其中每个哈希槽都有领导者及其副本以维持可用性：如果领导节点发生故障，副本节点将被提升为新领导者，系统将继续正确运行。哈希槽的计算存在一个例外，即哈希标签，它是一种确保多个键被分配到同一哈希槽的方式。这用于在 Redis 集群中实现多键操作 [23, 25]。

因此，多个无状态的限流器实例可以共享一个 Redis 集群，该集群能够提供：

- 通过数据分片和复制实现较高的可扩展性 。
- 通过 Redis 集群节点之间的复制 ，在所有实例之间实现一定程度的限流一致性 。

尽管这种配置的 Redis 集群方案可以缓解多个节点之间的数据不一致问题，但在网络分区时，根据*CAP* 定理（该定理指出，网络分区发生时，我们必须在一致性 和整体可用性之间二选一 [13]），或者在节点故障等其他实际问题发生时，较高的可用性 会以牺牲一定程度的一致性 为代价（反之亦然）。因此，Redis 集群不保证强一致性[7]。实际而言，这意味着在某些情况下，Redis 集群可能会丢失系统已向客户端确认的写入。

Redis 集群可能丢失写入的第一种情况是，它使用异步复制 ，而领导者有时可能会发生故障[8]。领导节点

[7]这里的“一致性”指的是*CAP* 定理中的“线性一致性”：一旦某个客户端成功完成写入，所有从数据库读取数据的客户端都必须能够看到刚写入的值 [13]。不要将不同的概念混淆：副本一致性 、异步复制系统中出现的最终一致性 ，以及 ACID 语境中的一致性 。后者指的是应用场景中数据库处于“良好状态”。[8]为简化起见，这里只考虑单主复制的情况，尽管多主复制 在可用性和延迟方面优于单主模式。在复制环境中，当多个客户端同时写入同一个值时，原子操作可以通过合并更新并得出相同结果来发挥作用。这尤其适用于可交换操作，例如本例中的计数器递增，因此可以防止副本之间发生丢失更新 [26], ，如图 3 所示。

对于单主复制，如果数据库进行了分区，不同分区的领导者可能位于不同节点上，但每个分区都必须有一个领导节点。

首先确认客户端的写入，但随后在将写入发送给副本之前便发生崩溃；之后，一个未收到该写入的副本节点可能会被提升为领导者。因此，由于旧领导者尚未复制的写入已被丢弃，系统将永远丢失这次写入。

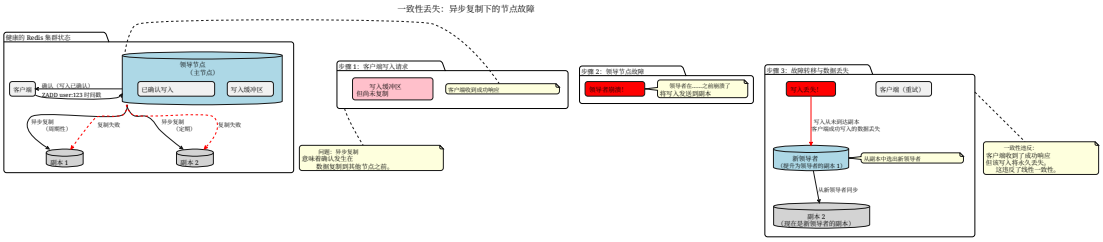

图 7：**Redis 集群中异步复制时因领导者故障导致写入丢失的场景**。领导者确认了客户端的写入，但在将其复制到跟随者之前便发生崩溃。故障转移期间，某个副本被提升为新领导者后，客户端已确认的写入将永久丢失，从而违反线性一致性。这说明 Redis 集群（一个 AP 系统）可能会牺牲一致性来换取可用性。

其他故障场景还包括脑裂，即两个节点都认为自己是领导者。如果两个领导者都接受写入，且没有解决冲突的办法，数据就可能丢失或损坏[13]。这种情况可能发生在网络分区 期间，此时客户端与少数派节点隔离，而领导节点也位于这些节点之中。如果分区很快恢复，集群就会恢复正常。然而，如果分区持续时间足够长，导致多数派一侧的节点判定领导节点已宕机，并投票选出该侧的另一个节点作为新领导者，那么客户端发送给少数派一侧旧领导者的写入就会丢失 [23], ，因此无法保证一致性。

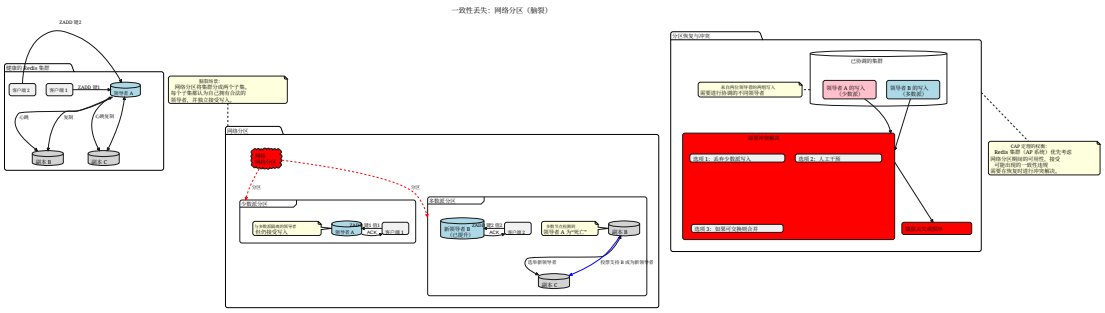

图 8：**Redis 集群网络分区期间的写入冲突场景（会导致脑裂）**。网络分区使原领导者与多数派节点隔离。少数派分区继续接受写入，而多数派选举出新领导者，也开始接受写入。网络分区恢复后，必须协调来自两个领导者的冲突写入，这可能导致数据丢失或损坏。这体现了 CAP 定理中的权衡：AP 系统在网络分区期间优先保证可用性。

在此设计方案中，根据 CAP 定理的要求，我们明确选择 AP（可用性和分区容错性）模型，而非 CP（一致性和分区容错性）模型。图 7 和图 8 所示的场景表明，与许多用于生产环境的分布式系统一样，Redis 集群在发生网络分区时无法同时保证强一致性和可用性。然而，我们选择 AP 并非只是承认 Redis 集群的技术局限，而是基于速率限制的具体需求所作出的审慎工程决策。

尽管边缘情况下写入可能丢失是一个重要的考虑因素，但选择 AP 的主要原因在于速率限制系统的效用函数：**网络分区期间丢失少量速率限制计数，优于丢弃所有合法流量**。如果速率限制器在网络分区期间完全不可用，就无法实现其核心目的：在确保服务可访问性的同时保护后端资源。AP 模型带来的暂时不一致性——不同节点可能暂时对请求进行不同计数——是可以接受的权衡，因为这样能确保速率限制服务持续运行，即使发生部分系统故障，也能继续提供基本保护。

这种务实的方法符合实际生产环境的需求：相比整个服务不可用，短暂的速率计数不准确所造成的后果要轻微得多。通过接受 Redis 集群 AP 模型所带来的最终一致性，我们优先保障系统韧性和持续运行；在分布式部署中不可避免出现网络分区时，这对于维持 API 可用性和用户体验至关重要。

这种权衡体现了生产环境中速率限制系统更广泛的设计取舍。我们采用 Redis 有序集合和服务器端 Lua 脚本，在单个

Redis 节点内，为每次速率限制决策提供强原子性，尽管整个系统在集群层面采用 AP 模型。这与针对极致规模优化的架构不同；后者会牺牲单次决策的准确性来换取吞吐量，例如 Databricks 的批量报告系统，它接受暂时超出速率限制的情况，以换取尾延迟提升 10 倍 [27]。两种方法都可行；选择取决于主要约束是严格保证每个请求的原子性，还是实现最大的水平可扩展性。

4.4 多数据中心部署

为了克服距离主数据中心较远的客户端所面临的高延迟，可以部署采用地理分布式限流器的边缘计算方案。为了平衡可用性与一致性权衡，可以采用以下实施策略：

- 采用支持跨数据中心复制的区域性 Redis 集群；基于 DNS 进行轮询，将请求路由到最近的速率限制器，以兼顾可用性和延迟。
- 可以在每个网络接入点（PoP）内创建独立的计数系统，因为在正常情况下，来自同一 IP 地址的流量始终会到达同一个 PoP [28]。

4.5 进一步优化

当然还有更多优化方法，例如降低延迟：

- 通过设置 Redis 的连接端口来保持持久连接，例如使用 TCP 连接池，从而减少 Redis 与客户端之间的连接开销 [29]。

- 通过 Redis 流水线技术批量执行多个 Redis 命令来更新计数器，从而减少往返时间 [30, 21, 27]。

5 生产环境注意事项

5.1 监控与分析

部署速率限制器一段时间后，下一步是通过监控一些基础设施指标，了解当前速率限制器参数设置所产生的反馈，例如：

- 服务器上的实际 工作进程利用率：监控应用速率限制器后系统承受压力的情况 [7]。
- 速率限制使用情况：监控请求模式、违规率和用户行为，以优化速率限制策略并检测滥用模式。
- Redis 性能：跟踪底层 Redis 基础设施的内存使用量、延迟和吞吐量，确保速率限制器达到最佳性能。

- 异常检测与告警：实时检测异常流量模式、系统性能下降和潜在安全威胁，并实施自动告警。

如前所述，工作器利用率负载卸载器 是一种特殊的速率限制器，它会监控系统承受的负载，并根据当前的工作进程利用率 指标（该值介于 0 和 1 之间，根据不同的应用基础设施进行评估）以及经过调优的阈值，决定是否接受新请求，而不是强制执行静态的业务规则。它通过概率性卸载动态适应系统状态，并逐步采取行动，这是一种流量控制机制。对于有兴趣了解工作器利用率负载卸载器 速率限制器机制及其实现的读者，请参阅 [19]。

5.2 容错策略

为防止单点故障，$Redis\ Sentinel$ 通过分布式系统为 Redis 提供高可用性，该系统会自动检测领导节点发生的故障，并用副本节点替换故障节点。其他副本会重新配置为使用新领导者，而使用 Redis 的客户端也会收到连接时应使用的新地址 [31, 32]。

有关可扩展分布式系统的高级监控与告警，以及在设计容错系统时如何应对节点故障、不可靠网络，以及可用性、一致性、持久性和延迟之间的权衡，本文篇幅有限，无法详述。请关注我们的后续文章，以获取最新信息。

6 结论

本文提出了一种具体的架构设计，用于构建可扩展的分布式速率限制系统。通过评估基本算法并分析其权衡，我们确立了滚动窗口模型作为精确执行请求限制的基础。我们的主要贡献是详细设计并论证了一种实现方案：采用 Redis 有序集合和服务器端 Lua 脚本编写，二者结合可实现 $O(log(N))$ 操作效率，并确保原子性，从而避免竞态条件。

在 Redis 集群上部署时，我们明确决定遵循 CAP 定理中的可用性和分区容错性（AP）。我们认为，对于速率限制器而言，全局系统正常运行时间至关重要，因此在故障转移期间接受短暂的状态不一致性，是正确且务实的工程权衡。这一设计选择与内存数据存储固有的性能相结合，使系统能够满足运行韧性、低延迟和高吞吐量等核心要求。

这项设计的核心要点包括：(1) 算法选择决定了系统的基本行为，滚动窗口能以可预测的内存成本提供准确性；(2) 原子操作——在此通过 Lua

脚本编写实现——对于并发分布式环境中的正确性不可或缺；(3) 基于内存模型（例如每个请求 8 字节）进行系统可观测性分析和容量规划，对于生产环境可扩展性至关重要；(4) 通过三层架构进行规则配置：根据规则参数进行哈希以加载缓存脚本，无须修改脚本，因而能够灵活适应不同的限流规则。

未来的工作可以探索结合不同算法优势的混合模型、基于实时负载的自动限额调优 [5], ，以及将这种架构模式应用于分布式系统中的其他有状态协调问题。

致谢

本研究是 WynerTech Solutions LLC 技术开发工作的一部分，重点关注可扩展系统架构模式。

作者感谢 Redis 开源社区开发并维护了作为我们实现基础的内存数据存储。我们也感谢 Hayes [8]和 Tarjan [19]在 Redis 有序集合速率限制方面的开创性工作。

参考文献

[1] B. Raghavan、K. Vishwanath、S. Ramabhadran、K. Yocum 和 A. C. Snoeren，《基于分布式速率限制的云控制》，*ACM SIGCOMM*计算机通信评论，第 37 卷，第 4 期，第 337–348 页，2007 年。[2] R. Stanojevic 和 R. Shorten，《负载均衡与分布式速率限制：云控制的统一框架》，载于 *2009 IEEE*国际通信会议。IEEE，2009 年，第 1–6 页。[3]X. Hu、T. Xu、L. Chen、X. Jiang、Y. Yang、L. Ye、X. Wang、Y. Lv、C. Jia、Y. Wu 等，《去中心化云网络中的分布式速率限制》，*IEEE*移动计算汇刊，2025 年。[4] Y. He、W. Wu、X. Wen、H. Li 和 Y. Yang，《面向云的可扩展交换机上速率限制器》，载于 *IEEE*国际计算机通信会议 *2021*（*IEEE*计算机通信会议）。IEEE，2021 年，第 1–10 页。[5] Z. Li、H. Sun、Z. Xiong、Q. Huang、Z. Hu、D. Li、S. Ruan、H. Hong、J. Gui、J. He 等，《Noah：面向大规模电子商务服务中微服务的基于强化学习的速率限制器》，*IEEE*神经网络与学习系统汇刊，第 34 卷，第 9 期，第 5403–5417 页，2023 年。[6] Z. Chen、Y. Fan、K. Qian、Q. Meng、R. Shu、X. Li、Y. Zhang、B. Wang、W. Li 和 F. Rent，《Scalatap：公有云中的可扩展出站速率限制》，载于*IEEE*国际计算机通信会议 *2025*（*IEEE*计算机通信会议）。IEEE，2025 年，第 1–10 页。

[7] P. Tarjan, 《使用限流器扩展 API》，https://stripe.com/blog/rate-limiters，2017 年，访问日期：2025年12月19日。[8] P. Hayes, 《使用 Redis 有序集合实现更好的速率限制》，https://engineering.classdojo.com/blog/2015/02/06/rolling-rate-limiter/，2015 年，访问日期：2025年11月4日。[9]AWS, 《对 REST API 请求进行限流，以提高 API Gateway 的吞吐量》，https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html，访问日期：2025年12月 29 日。[10] IBM, 《什么是 Redis? 》，https://www.ibm.com/think/topics/redis，访问日期：2025年11月11日。[11]Redis, 《ZREVRANGE》，https://redis.io/docs/latest/commands/zrevrange/，访问日期：2025年11月4日。[12]S. Harizopoulos、D. J. Abadi、S. Madden 和 M. Stonebraker, 《透过棱镜看在线事务处理，我们的发现》，载于 《让数据库发挥作用：*Michael Stonebraker* 的实用智慧》，2018 年，第 409–439 页。[13] M. Kleppmann, 《设计数据密集型应用：可靠、可扩展且易于维护的系统背后的重要理念》。“O’Reilly Media, Inc.”，2017 年。[14] Redis, 《Redis 持久化》，https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/，访问日期：2026年2月5日。[15]——, 《EVALSHA》，https://redis.io/docs/latest/commands/evalsha/，访问日期：2026年2月5日。[16]——, 《Redis 可编程性》，https://redis.io/docs/latest/develop/ programmability/，访问日期：2026年2月5日。[17] Lyft, 《ratelimit》，https://github.com/envoyproxy/ratelimit，2020年，访问日期：2025年12月4日。[18]N. Wiger, 《关于原子性的激烈议论》，https://nateware.com/2010/02/18/ an-atomic-rant/，2010年，访问日期：2025年12月6日。[19] P. Tarjan, 《使用限流器扩展 API》，https://gist.github.com/ ptarjan/e38f45f2dfe601419ca3af937fff574d#request-rate-limiter，2017 年，访问日期：2025年11月4日。[20]Redis, 《使用 Lua 实现原子性》，https://redis.io/learn/develop/java/spring/rate-limiting/fixed-window/reactive-lua，访问日期：2025年11月26日。[21]——, 《Redis 流水线》，https://redis.io/docs/latest/develop/using-commands/pipelining/，访问日期：2025年12月2日。

[22]——，“Eval”，https://redis.io/docs/latest/commands/eval/，访问于 2025-12-02。[23] ——，“通过 Redis 集群进行扩展”，https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/，访问于 2025-11-10。[24] P. Namuag，“Redis 中的哈希槽与一致性哈希”，https://severalnines.com/blog/hash-slot-vs-consistent-hashing-redis/，2021 年，访问于 2025-11-17。[25] Redis，“Redis 集群规范”，https://redis.io/docs/latest/operate/ oss_and_stack/reference/cluster-spec/，访问于 2025-11-17。[26]Riak，“分布式数据类型——Riak 2.0”，https://riak.com/ distributed-data-types-riak-2-0/，2014 年，访问于 2025-12-01。[27]D. Hoa、J. Wang、S. Soti、V. Peng 和 G. Nanda，“Databricks 的高性能速率限制”，https://www.databricks.com/blog/high-performance-ratelimiting-databricks，2025 年，访问于 2026-02-11。[28]J. Desgats，“我们如何构建能够扩展至数百万个域名的速率限制系统”，https://blog.cloudflare.com/counting-things-a-lot-of-different-things/，2017年，访问于 2025-12- 04。[29] Redis，“连接池与多路复用”，https://redis.io/docs/latest/develop/clients/pools-and-muxing/，访问于 2025-12-02。[30]——，“Redis 序列化协议规范”，https://redis.io/docs/latest/develop/reference/protocol-spec/，访问于 2025-12-02。[31] ——，“使用 Redis Sentinel 实现高可用性”，https://redis.io/docs/latest/operate/oss_and_stack/management/sentinel/，访问于 2025-11-17。[32] C. Kong，“Redis Sentinel 与 Redis 集群：比较概述”，https://medium.com/@chaewonkong/redis-sentinel-vs-redis-cluster-a-comparative-overview-8c2561d3168f，2023 年，访问于 2025-11-10。