如果计数器已经达到限制的请求数量,丢弃请求或者返回一个错误.
* 每隔固定时间间隔,将计数器的值减去窗口内的请求数量,并更新窗口内的请求数量.

```
limit = 100  # 限制的请求数量
window = 60  # 滑动窗口的时间长度为60秒
interval = 10  # 固定时间间隔为10秒

def handle_request():
    global counter, window_start_time, window_requests

    current_time = time.time()

    if current_time > window_start_time + window:
        counter -= window_requests.pop(0)
        window_requests.append(0)
        window_start_time = current_time

    if counter < limit:
        counter += 1
        window_requests[-1] += 1
        # 处理请求
    else:
        # 丢弃请求或返回错误
```

2.1.2 令牌桶

描述:原理是系统以恒定的速率产生令牌,并将这些令牌放入一个桶中.当有请求到达时,需要从桶中获取一个令牌才能继续处理该请求.如果桶中没有足够的令牌,则请求被拒绝.

特点:既能够面对突发的流量峰值,也能处理平滑的系统请求.

具体来说,令牌桶算法的实现包括两个关键参数：令牌生成速率（**rate**）和桶的容量（**capacity**）.令牌生成速率确定了每秒产生的令牌数量,桶的容量确定了桶中最多可以存放多少个令牌.

```
import time

class TokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate  # 令牌生成速率（令牌/秒）
        self.capacity = capacity  # 桶的容量（令牌数量）
        self.tokens = 0  # 当前桶中的令牌数量
        self.last_time = time.time()  # 上次令牌生成时间

    def get_token(self):
        now = time.time()
        elapsed = now - self.last_time  # 计算时间间隔
        self.last_time = now
        self.tokens = min(self.tokens + elapsed * self.rate, self.capacity)  # 生成令牌
        if self.tokens >= 1:
            self.tokens -= 1
            return True
        else:
            return False

# 示例用法
bucket = TokenBucket(rate=1, capacity=5)  # 令牌生成速率为1个/秒,桶的容量为5个令牌
for i in range(10):
    if bucket.get_token():
        print(f"请求 {i+1} 被处理")
    else:
        print(f"请求 {i+1} 被限流")
        time.sleep(1)  # 等待1秒再重新尝试获取令牌
```

2.1.3 漏桶算法

漏桶算法（**Leaky Bucket Algorithm**）是一种流量控制算法,用于平滑网络流量.它模拟了一个漏桶,当网络流量超过桶的容量时, 过 **lua** 获取本地的物理机 **cpu** 指标. 具体的限制指标度量,可以依照压测对于 **cpu** 的观测.

```
-- 执行系统命令并获取输出结果
```

```
function execute_command(command)
    local handle = io.popen(command)
    local result = handle:read("*a")
    handle:close()
    return result
end

-- 获取系统的CPU信息
function get_cpu_info()
    local cpu_info = execute_command("cat /proc/cpuinfo")
    return cpu_info
end

-- 示例代码
local cpu_info = get_cpu_info()
print(cpu_info)
```

2.3.3.3 服务本身的限流

上面的限流基本是在服务本身之外的.或网关或流量代理实现的.
对于大型的分布式系统一般是选用集群限流.能很好的因为一些节点的故障带来的整体流量的承载能力的下载如果是单机限流,在节点故障后相当于整体的系统负载能力就削弱了.

具体的限流阈值有两个点.

* 或者根据相应的往年的监控流量值来进行推算.
* 具体根据压测时的阈值来进行调整.

第一点需要系统具备完整的监控体系.以获取足够的历史数据来进行推演.有的同学说了,我的系统或者我的接口是第一次上线无法获取足够的历史监控来推测.
系统第一次上线的场景:只能与 **qa** 测试这边打一个配合,完整的链路压测来彻底了解你的系统.获取到系统在可接受的接口性能下如:**200ms**.下的最大单机**qps**或整体的集群最大 **qps**.
如果是单机压测,那么集群的限流就是线上机器数*单机承压的 **qps** 了.

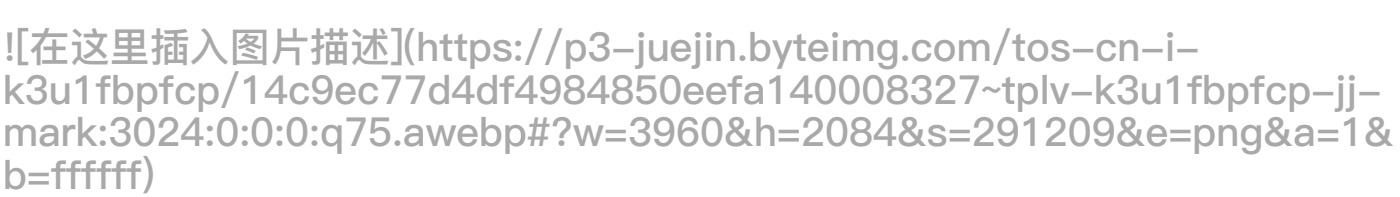
还有一个点对于集群限流的场景,还是觉得有必要 **mark** 下.对于 **rpc** 服务来说,特别是上下游关系复杂调用方较多.这时候有单纯的一个集群限流往往解决不了问题.很可能需要根据不同的调用方来实现不同的集群限流.例如:对于核心的,量大的 调用方单独的集群限流.如 **2** 个这种调用方就需要两个

集群限流 **key**.其他的一些小流量这样可以归并到 **other** 这样的集群限流渠道就好了.

集群限流的实现方式 **sentinel**

预取率:指的是在集群限流场景下.本地的调用端对于集群 **token** 的预取占比.举个例子:集群限流是 10000.预取:7000.那么预取率则是 70%.
假如有 **70** 台机器.则每个机器的 **token** 预取个数则是 **10**.

目前集群限流的过程



通过图示可以看出图上为令牌桶限流算法.
通过平台设置指定 **key** 的限流阈值.
consumer 启动完成 **token** 预取.预取的逻辑很常见.(如:美团的 **leaf** 的框架也有设计.好的框架总是借鉴来借钱去,这里也是套用电影里的说法
致敬)
实际请求过程中通过**token** 验证通过,完成对 **provider** 的调用.验证不通过则触发限流.

> 赠人玫瑰 手有余香 我是柏修 一名持续更新的晚熟程序员
> 期待您的点赞,加收藏，加个不迷路，感谢
> 您的鼓励是我更新的最大动力
> ↓↓↓↓↓↓

原文链接: https://juejin.cn/post/7375829906975965235