流量控制模块

在互联网平台中，由于高并发的用户访问，会给系统带来压力，影响系统的性能。当流量过大时，我们需要采用合适的流量控制策略，对用户访问进行限制。常用的流量控制手段有：

缓存：缓存的目的是提升系统访问速度和增大系统能处理的容量。

降级：当服务出问题或者影响到核心流程的性能则需要暂时屏蔽掉，待高峰或者问题解决后再打开。

限流：通过对并发访问/请求进行限速或者一个时间窗口内的的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务、排队或等待。

在这里，我们重点介绍一下限流算法。限流常用的处理手段有：计数器、滑动窗口、漏桶、令牌桶等算法。

1、计数器： 存在临界缺陷

计数器是一种比较简单的限流算法，在接口层面，很多地方使用这种方式限流。即在一段时间内，进行技术，与预支进行比较，到达时间临界点是，将计数器请 0。其过程如图 1-1 所示：

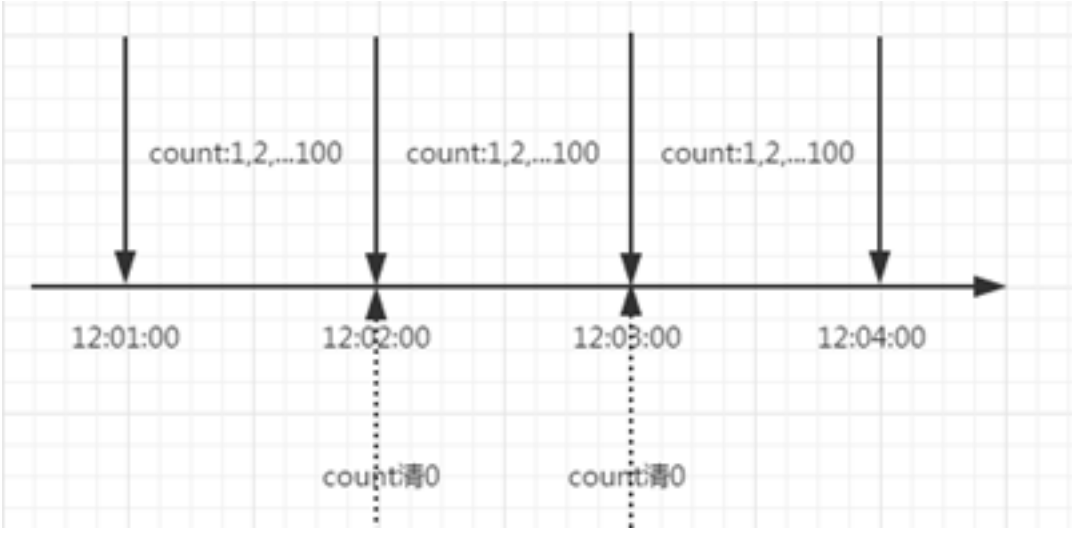

图 1-1 计数器过程

这里需要注意的事：存在一个时间临界点的问题。举个例子，在 12:01:00 到 12:01:58 这段时间内没有用户请求，然后在 12:01:59 这一瞬时发出 100 个请求，OK，然后在 12:02:00 这一瞬时又发出了 100 个请求。这里你应该能感受到，在这个临界点可能会承受恶意用户的大量请求，甚至超出系统预期的承受。

2、滑动窗口：

由于计数器存在临界点缺陷，后来出现了滑动窗口算法来解决。滑动窗口的意思是说把固定时间片，进行划分，并且随着时间的流逝，进行移动，这样就巧妙的避开了计数器的临界点问题。也就是说这些固定数量的可以移动的格子，将会进行计数判断阀值，因此格子的数量影响着滑动窗口算法的精度。滑动窗口过程如图 1-2 所示：

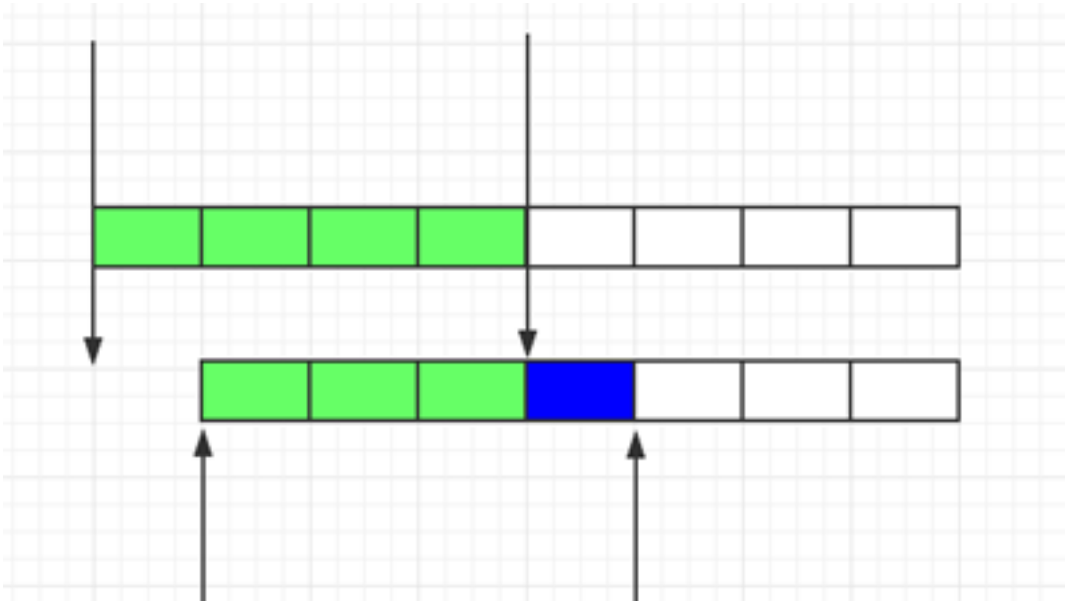

图 1-2 滑动窗口过程

3、漏桶算法：

虽然滑动串口有效的避免了时间临界点的问题，但依然有时间片的概念，故产生了漏桶算法。漏桶算法思路很简单，水(请求)先进入到漏桶里，漏桶以一定的速度出水(接口有响应速率)，当水流入速度过大会直接溢出(访问频率超过接口响应速率)，然后就拒绝请求，可以看出漏桶算法能强行限制数据的传输速率，示意图如图 1-3 所示。

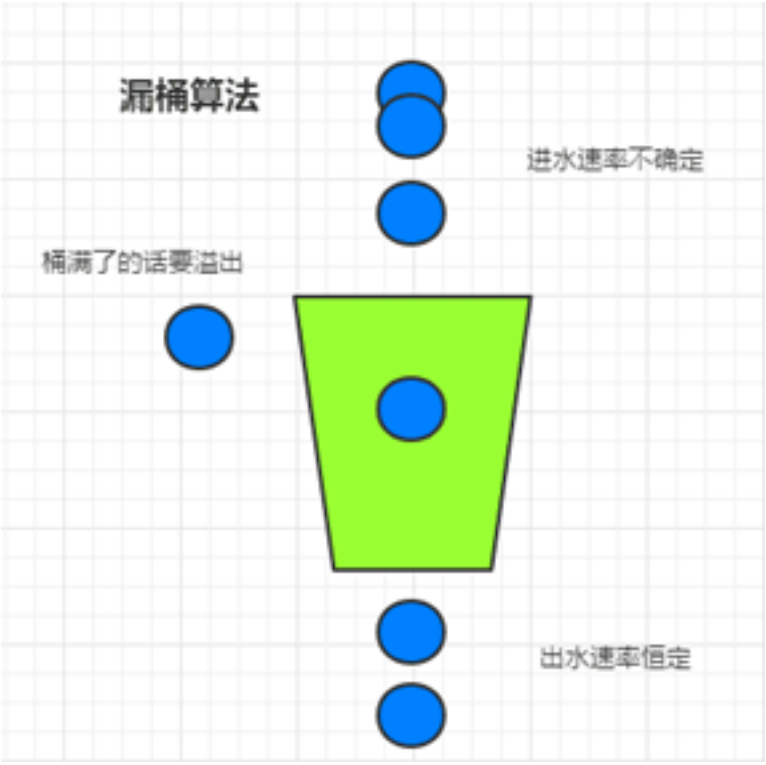

图 1-3 漏桶算法

4、令牌桶算法：

可以注意到，漏桶的出水速度是恒定的，那么意味着如果瞬时大流量的话，将有大部分请求被丢弃掉（也就是所谓的溢出）。对于很多应用场景来说，除了要求能够限制数据的平均传输速率外，还要求允许某种程度的突发传输。这时候漏桶算法可能就不合适了，令牌桶算法更为适合。令牌桶算法的原理是系统会以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。

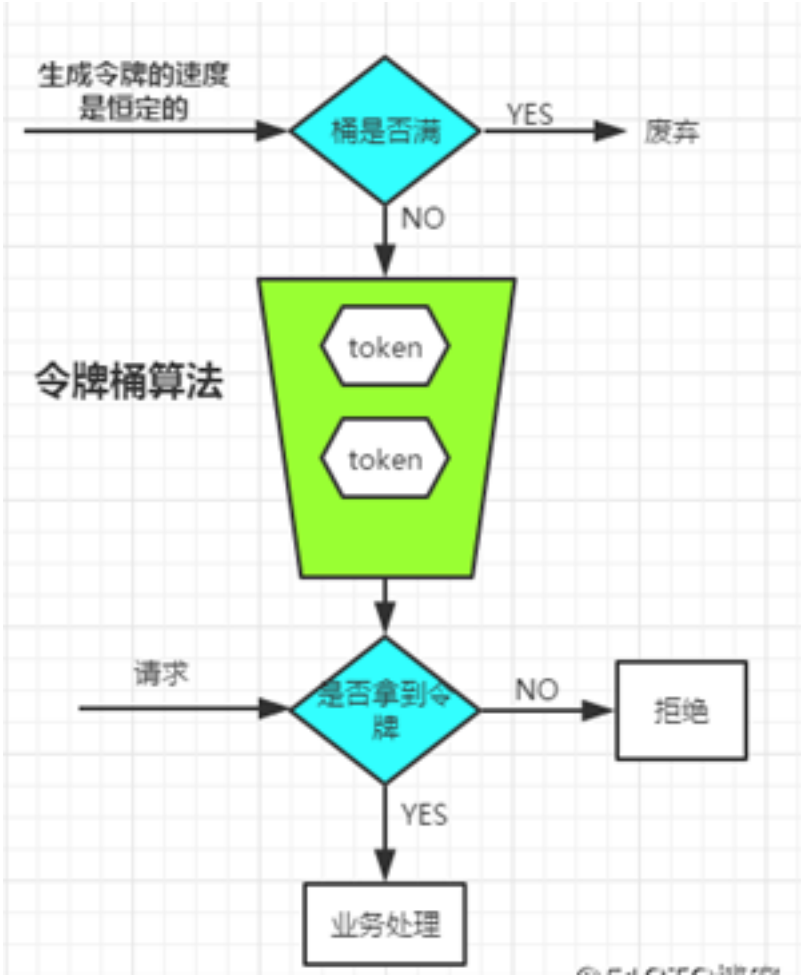

图 1-4 令牌桶算法

令牌桶算法的基本过程如下：

假如用户配置的平均发送速率为 r，则每隔 1/r 秒一个令牌被加入到桶中；如果桶最多可以存发 b 个令牌，当令牌到达时令牌桶已经满了，那么这个令牌会被丢弃；当一个 n 个字节的数据包到达时，就从令牌桶中删除 n 个令牌，并且数据包被发送到网络；如果令牌桶中少于 n 个令牌，那么不会删除令牌，并且认为这个数据包在流量限制之外。

5、限流工具类：Guava RateLimiter

Google 开源工具包 Guava 提供了限流工具类 RateLimiter，该类基于令牌桶算法来完成限流，我们只需要告诉 RateLimiter 系统限制的 QPS 是多少，那么 RateLimiter 将以这个速度往桶里面放入令牌，然后请求的时候，通过 tryAcquire()方法向 RateLimiter 获取许可（令牌）。

6、分布式场景下的限流：

上述的限流方法是对单应用的请求进行限制，但如果我们将应用部署到多台服务器上，应用级限流方式不能进行全局限流，因此我们需要采用分布式限流来解决这个问题。在分布式场景中，由于多个模块之间不能保证相互阻塞，共享的变量也不在一片内存空间中。如果使用漏桶和令牌桶等阻塞限流的算法，我们不得不将统计流量放到 redis 一类的共享内存中，如果操作是一系列符合的操作，我们还不能使用 redis 自带的 CAS 操作（CAS 操作只能保证单个操作的原子性）或使用中间件级别的队列来阻塞操作，而加分布式锁的开销又非常巨大，最终选择放弃阻塞式限流。在分布式场景下，仅仅使用 redis+lua 脚本的方式来达到分布式限流的效果。Redis 执行 lua 脚本是一个单线程的行为，所以不需要显示加锁，可以说避免了加锁导致的线程切换开销。

Lua 脚本如下：

固定窗口：

```
local current;
current = redis.call('incr',KEYS[1]);
if tonumber(current) == 1 then
	redis.call('expire',KEYS[1],ARGV[1]);
	return 1;
else
	if tonumber(current) <= tonumber(ARGV[2]) then
		return 1;
	else
		return -1;
	end
end
```

参数说明： KEYS[1] – 时间戳(key), ARGV[1] – key 的过期时间, ARGV[2] – 限制的最大次数

滑动窗口：

```
local currentSectionCount;
local previousSectionCount;
local totalCountInPeriod;
currentSectionCount = redis.call('zcount',KEYS[2],'-inf','+inf');
previousSectionCount = redis.call('zcount',KEYS[1],ARGV[3],'+inf');
totalCountInPeriod =
tonumber(currentSectionCount)+tonumber(previousSectionCount);
if totalCountInPeriod < tonumber(ARGV[5]) then
	redis.call('zadd',KEYS[2],ARGV[1],ARGV[2]);
		if tonumber(currentSectionCount) == 0 then
			redis.call('expire',KEYS[2],ARGV[4]);
		end
	return 1;
else
	return -1;
end
```

参数说明(单位时间以分钟为例):

KEYS[1] – 当前时刻前一分钟(时间戳/60-1), KEYS[2] --当前时刻的分钟值(时间戳/60)(set);

ARGV[1] -- 当前时间的微秒值(key), ARGV[2] -- 当前时间的微秒值(value), ARGV[3] – 当前时间-生存时间的微秒值, ARGV[4] – 过期时间, ARVG[5] – 限制的最大值

通过 redis+lua 脚本的方式，可以实现分布式的流量控制，该算法的具体实现过程如下：

（1） 通过 spring AOP 方法拦截用户请求，在拦截的请求中获得用户调用的 appid 和 apiid。

（2） 根据 appid 和 apiid 组合的 key 值，在 redis 缓存中查找用户的流量控制策略，如果找到流控策略，则执行方法（3），否则，执行方法（4）。

（3） 将 appid、apiid 和当前时间戳组合作为 key 值，将最大访问次数作为 value 值存入 redis 中，用户调用一次接口，将 value 值加一，在限制时间内达到最大次数，返回-1，否则返回+1，根据返回的状态对请求进行放行或拒绝。

（4） Redis 中不存在该流控策略，根据 apiid 从数据库中查找对应的流量控制策略，并将 appid 和 apiid 作为 key 值，流控策略作为 value 值存入 redis 中，并执行一次方法（3）。

通过 Java 代码和上述两个脚本配合，可以完成针对访问次数的流量控制，那么对于单次访问流量较大的 API 进行流量控制该如何去做呢？下面给出固定窗口的 lua 脚本。

```
local current;
local exist;
exist = redis.call('exists',KEYS[1]);
if tonumber(exist) == 0 then
        redis.call('set',KEYS[1],0);
        redis.call('expire',KEYS[1],ARGV[2]);
end
current = redis.call('INCRBY',KEYS[1],ARGV[1]);
if tonumber(current) <= tonumber(ARGV[3]) then
        return 1;
else
        return -1;
end ;
```

参数说明(单位时间以分钟为例):

KEYS[1] – 当前时刻分钟值(时间戳/60)

ARGV[1] -- 当前时间的流量值, ARGV[2] – 生存时间的微秒值, ARGV[3] – 限制的最大值