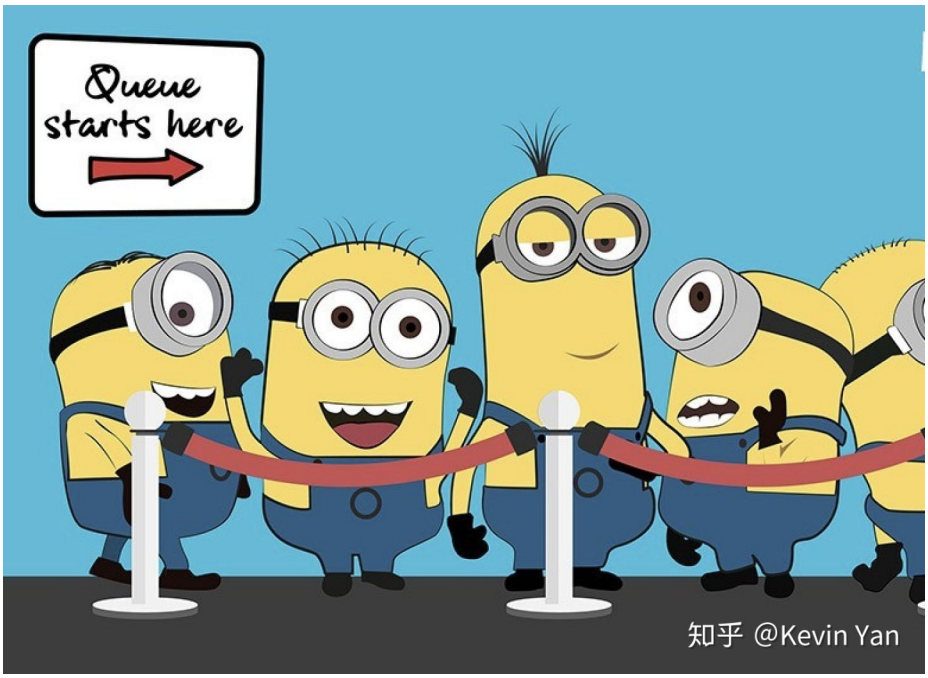
常用限流算法的应用场景和实现原理

Kevin Yan



在高并发业务场景下，保护系统时，常用的"三板斧"有："**熔断、降级和限流**"。今天和大家谈谈常用的限流算法的几种实现方式，这里所说的限流并非是网关层面的限流，而是业务代码中的逻辑限流。

限流算法常用的几种实现方式有如下四种：

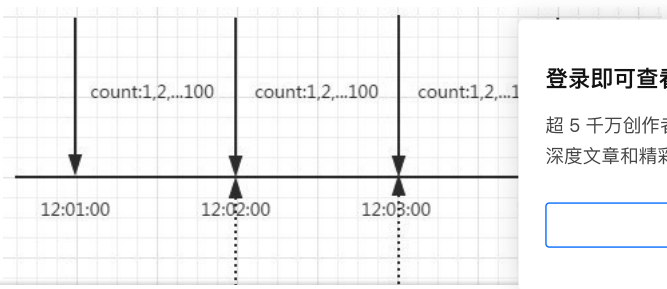
- 计数器
- 滑动窗口
- 漏桶
- 令牌桶

下面会展开说每种算法的实现原理和他们自身的缺陷，方便以后我们在实际应用中能够根据不同的情况选择正确的限流算法。

计数器

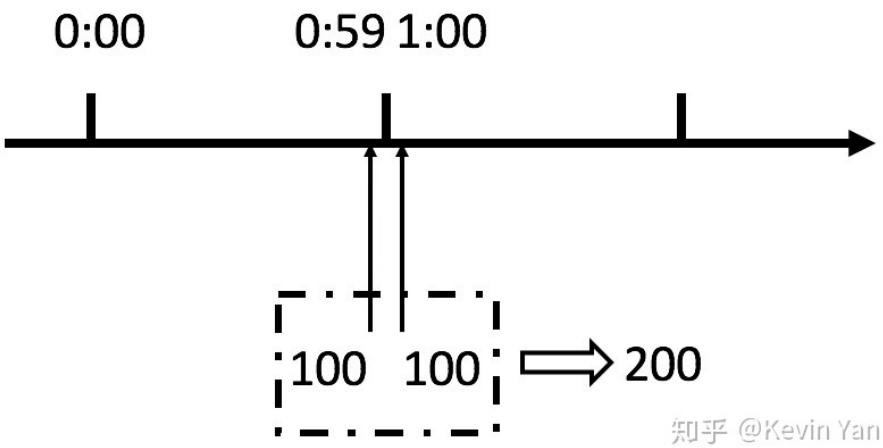
算法思想

计数器是一种比较简单粗暴的限流算法，其思想是在**固定时间窗口**内对请求进行计数，与阀值进行比较判断是否需要限流，一旦到了时间临界点，将计数器清零。



面临的问题

计数器算法存在“时间临界点”缺陷。比如每一分钟限制100个请求，可以在00:00:00-00:00:58秒里面都没有请求，在00:00:59瞬间发送100个请求，这个对于计数器算法来是允许的，然后在00:01:00再次发送100个请求，意味着在短短1s内发送了200个请求，如果量更大呢，系统可能会承受不住瞬间流量，导致系统崩溃。（如下图所示）



所以计数器算法实现限流的问题是没有办法应对突发流量，不过它的算法实现起来确实最简单的，下面给出一个用 Go 代码实现的计数器。

代码实现

```
type LimitRate struct {
   rate  int           //阀值
   begin time.Time     //计数开始时间
   cycle time.Duration //计数周期
   count int           //收到的请求数
   lock  sync.Mutex    //锁
}

func (limit *LimitRate) Allow() bool {
   limit.lock.Lock()
   defer limit.lock.Unlock()

   // 判断收到请求数是否达到阀值
   if limit.count == limit.rate-1 {
      now := time.Now()
      // 达到阀值后，判断是否是请求周期内
      if now.Sub(limit.begin) >= limit.cycle {
         limit.Reset(now)
         return true
      }
      return false
   } else {
      limit.count++
      return true
   }
}

func (limit *LimitRate) Set(rate int, cycle time.Durati
   limit.rate = rate
   limit.begin = time.Now()
   limit.cycle = cycle
```

```
    limit.begin = begin
    limit.count = 0
}
```

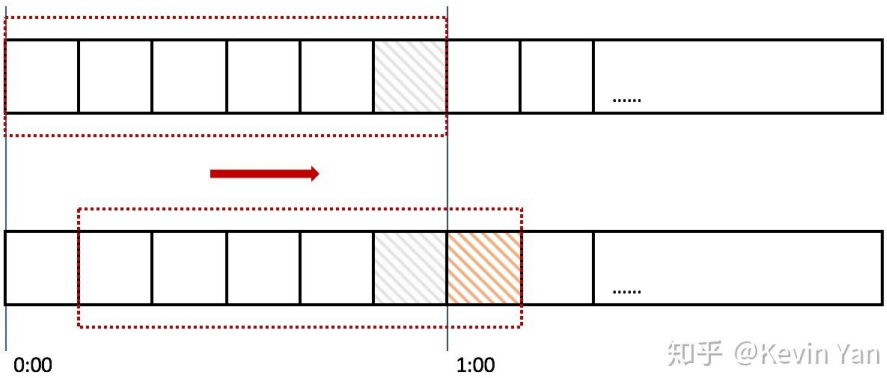
滑动窗口

算法思想

滑动窗口算法将一个大的时间窗口分成多个小窗口，每次大窗口向后滑动一个小窗口，并保证大的窗口内流量不会超出最大值，这种实现比固定窗口的流量曲线更加平滑。

普通时间窗口有一个问题，比如窗口期内请求的上限是100，假设有100个请求集中在前1s的后100ms，100个请求集中在后1s的前100ms，其实在这200ms内就已经请求超限了，但是由于时间窗每经过1s就会重置计数，就无法识别到这种请求超限。

对于滑动时间窗口，我们可以把1ms的时间窗口划分成10个小窗口，或者想象窗口有10个时间插槽time slot, 每个time slot统计某个100ms的请求数量。每经过100ms，有一个新的time slot加入窗口，早于当前时间1s的time slot出窗口。窗口内最多维护10个time slot。



面临的问题

滑动窗口算法是固定窗口的一种改进，但从根本上并没有真正解决固定窗口算法的临界突发流量问题

代码实现

主要就是实现滑动窗口算法，不过滑动窗口算法一般是找出数组中连续k个元素的最大值，这里是已知最大值n (就是请求上限）如果超过最大值就不予通过。

可以参考Bilibili开源的kratos框架里circuit breaker用循环列表保存time slot对象的实现，他们这个实现的好处是不用频繁的创建和销毁time slot对象。下面给出一个简单的基本实现：

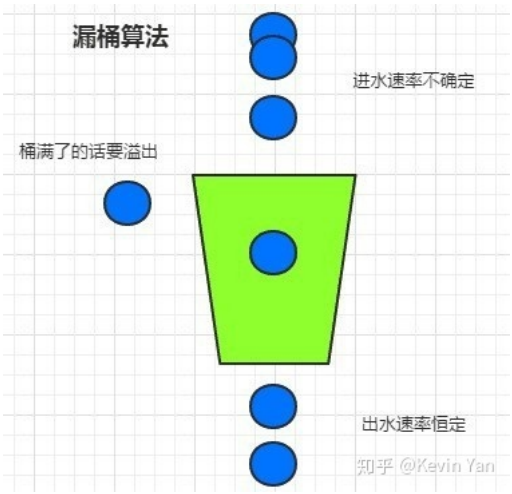
```
type timeSlot struct {
    timestamp time.Time // 这个timeSlot的时间起点
    count     int       // 落在这个timeSlot内的请求数
}

// 统计整个时间窗口中已经发生的请求次数
func countReq(win []*timeSlot) int {
    var count int
    for _, ts := range win {
        count += ts.count
    }
```

```
    mu           sync.Mutex    // 互斥锁保护其他字段
    SlotDuration time.Duration // time slot的长度
    WinDuration  time.Duration // sliding window的长度
    numSlots     int           // window内最多有多少个slot
    windows      []*timeSlot
    maxReq       int // 大窗口时间内允许的最大请求数
}

func NewSliding(slotDuration time.Duration, winDuration time.Duration, maxReq :
    return &SlidingWindowLimiter{
        SlotDuration: slotDuration,
        WinDuration:  winDuration,
        numSlots:     int(winDuration / slotDuration),
        maxReq:       maxReq,
    }
}


func (l *SlidingWindowLimiter) validate() bool {
    l.mu.Lock()
    defer l.mu.Unlock()


    now := time.Now()
    // 已经过期的time slot移出时间窗
    timeoutOffset := -1
    for i, ts := range l.windows {
        if ts.timestamp.Add(l.WinDuration).After(now) {
            break
        }
        timeoutOffset = i
    }
    if timeoutOffset > -1 {
        l.windows = l.windows[timeoutOffset+1:]
    }

    // 判断请求是否超限
    var result bool
    if countReq(l.windows) < l.maxReq {
        result = true
    }

    // 记录这次的请求数
    var lastSlot *timeSlot
    if len(l.windows) > 0 {
        lastSlot = l.windows[len(l.windows)-1]
        if lastSlot.timestamp.Add(l.SlotDuration).Before(now) {
            // 如果当前时间已经超过这个时间插槽的跨度，那么新建一个时间插槽
            lastSlot = &timeSlot{timestamp: now, count: 1}
            l.windows = append(l.windows, lastSlot)
        } else {
            lastSlot.count++
        }
    } else {
        lastSlot = &timeSlot{timestamp: now, count: 1}
        l.windows = append(l.windows, lastSlot)
    }


    return result
}
```

滑动窗口实现起来代码有点多，完整可运行的测试代码可以访问我的

算法思想

漏桶算法是首先想象有一个木桶，桶的容量是固定的。当有请求到来时先放到木桶中，处理请求的 worker 以固定的速度从木桶中取出请求进行相应。如果木桶已经满了，直接返回请求频率超限的错误码或者页面。



适用场景

漏桶算法是流量最均匀的限流实现方式，一般用于流量“整形”。例如保护数据库的限流，先把对数据库的访问加入到木桶中，worker再以db能够承受的qps从木桶中取出请求，去访问数据库。

存在的问题

木桶流入请求的速率是不固定的，但是流出的速率是恒定的。这样的话能保护系统资源不被打满，但是面对突发流量时会有大量请求失败，不适合电商抢购和微博出现热点事件等场景的限流。

代码实现

```
// 漏桶
// 一个固定大小的桶，请求按照固定的速率流出
// 如果桶是空的，不需要流出请求
// 请求数大于桶的容量，则抛弃多余请求

type LeakyBucket struct {
   rate       float64    // 每秒固定流出速率
   capacity   float64    // 桶的容量
   water      float64    // 当前桶中请求量
   lastLeakMs int64      // 桶上次漏水微秒数
   lock       sync.Mutex // 锁
}

func (leaky *LeakyBucket) Allow() bool {
   leaky.lock.Lock()
   defer leaky.lock.Unlock()

   now := time.Now().UnixNano() / 1e6
   // 计算剩余水量,两次执行时间中需要漏掉的水
   leakyWater := leaky.water - (float64(now-leaky.lastL
   leaky.water = math.Max(0, leakyWater)
   leaky.lastLeakMs = now
   if leaky.water+1 <= leaky.capacity {
```

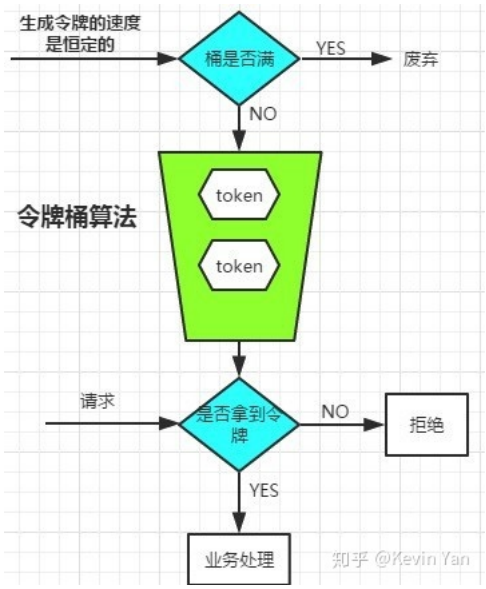
```
    }
}

func (leaky *LeakyBucket) Set(rate, capacity float64) {
    leaky.rate = rate
    leaky.capacity = capacity
    leaky.water = 0
    leaky.lastLeakMs = time.Now().UnixNano() / 1e6
}
```

令牌桶

算法思想

令牌桶是反向的"漏桶"，它是以恒定的速度往木桶里加入令牌，木桶满了则不再加入令牌。服务收到请求时尝试从木桶中取出一个令牌，如果能够得到令牌则继续执行后续的业务逻辑。如果没有得到令牌，直接返回访问频率超限的错误码或页面等，不继续执行后续的业务逻辑。

特点：由于木桶内只要有令牌，请求就可以被处理，所以令牌桶算法可以支持突发流量。



同时由于往木桶添加令牌的速度是恒定的，且木桶的容量有上限，所以单位时间内处理的请求书也能够得到控制，起到限流的目的。假设加入令牌的速度为 1token/10ms，桶的容量为500，在请求比较的少的时候（小于每10毫秒1个请求）时，木桶可以先"攒"一些令牌（最多500个）。当有突发流量时，一下把木桶内的令牌取空，也就是有500个在并发执行的业务逻辑，之后要等每10ms补充一个新的令牌才能接收一个新的请求。

参数设置

木桶的容量 - 考虑业务逻辑的资源消耗和机器能承载并发处理多少业务逻辑。

生成令牌的速度 - 太慢的话起不到“攒”令牌应对突发流量的效果。

适用场景

适合电商抢购或者微博出现热点事件这种场景，因为在限流的同时可以
采用漏桶那样的均匀速度处理请求的算法，在发生热点时间的时候，会
对用户体验的损害比较大。

```
    capacity     int64 //桶的容量
    tokens       int64 //桶中当前token数量
    lastTokenSec int64 //上次向桶中放令牌的时间的时间戳，单位为秒

    lock sync.Mutex
}

func (bucket *TokenBucket) Take() bool {
    bucket.lock.Lock()
    defer bucket.lock.Unlock()

    now := time.Now().Unix()
    bucket.tokens = bucket.tokens + (now-bucket.lastTokenSec)*bucket.rate // 先
    if bucket.tokens > bucket.capacity {
        bucket.tokens = bucket.capacity
    }
    bucket.lastTokenSec = now
    if bucket.tokens > 0 {
        // 还有令牌，领取令牌
        bucket.tokens--
        return true
    } else {
        // 没有令牌,则拒绝
        return false
    }
}

func (bucket *TokenBucket) Init(rate, cap int64) {
    bucket.rate = rate
    bucket.capacity = cap
    bucket.tokens = 0
    bucket.lastTokenSec = time.Now().Unix()
}
```

总结

这几种常用的限流算法实现方式，相互之间没有所谓的"谁是所谓的更优秀的实现方法"，主要还是看具体用在哪里。这几个限流算法的简单实现和测试代码我都放到 GitHub 上了，除此之外仓库里还有不少有用的小程序，有兴趣的可以点击**阅读原文**自取。

- Reset计时器的正确姿势
- 结合cancelCtx, Timer, Goroutine, Channel的一个例子
- 使用WaitGroup, Channel和Context打造一个并发用户标签查询器
- 使用sync.Cond实现一个有限容量的队列
- 使用信号量控制有限资源的并发访问
- 使用Chan扩展互斥锁的功能
- 用SingleFlight合并重复请求
- CyclicBarrier 循环栅栏
- Go并发编程同步原语之ErrorGroup

- 线上问题解决实录
  - 重定向运行时panic到日志文件
  - 用Go的交叉编译和条件编译让自己的软件包运行在多平台上
- 一些有意思的小程序
  - 一个简单的概率抽奖工具
  - 限流算法之计数器
  - 限流算法之滑动窗口
  - 限流算法之漏桶
  - 限流算法之令牌桶

> 看到这里了，如果喜欢我的文章就帮我点个赞吧，我会每周通过技术文章分享我的所学所见和第一手实践经验，感谢你的支持。微信搜索关注公众号「网管叨bi叨」每周教会你一个进阶知识，还有专门写给开发工程师的Kubernetes入门教程。

编辑于 2020-12-23 15:26

算法 高并发 Go 语言

1 条评论 切换为时间排序

只有关注了作者的人才可以评论

rking IP 属地北京 · 05-08

如果漏桶和令牌的cap、rate设置为相同，目测突发流量处理能力是一样的

甚至由于令牌init的token为空，刚初始化后的令牌都不具备对『突发流量』的处理能力，反而漏桶表现更好

这是什么呢

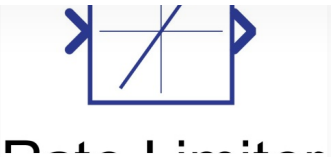
赞

文章被以下专栏收录

网管叨bi叨
终于把错别字改过来了

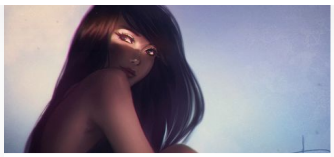
常见限流方案设计与实现

luoxn... 发表于技术之外



浅谈限流算法

Aaron... 发表于青灯抽丝



阿里云二面：你对限流了解多少?

里奥ii 发表于Java学...

限流的
以及微
限流概念
限速来保
做到有损
载过高时
务限流方
QPS：限

ken