常用限流算法的应用场景和实现原理

原创 KevinYan11 网管叨bi叨 2020-12-20 10:28

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：

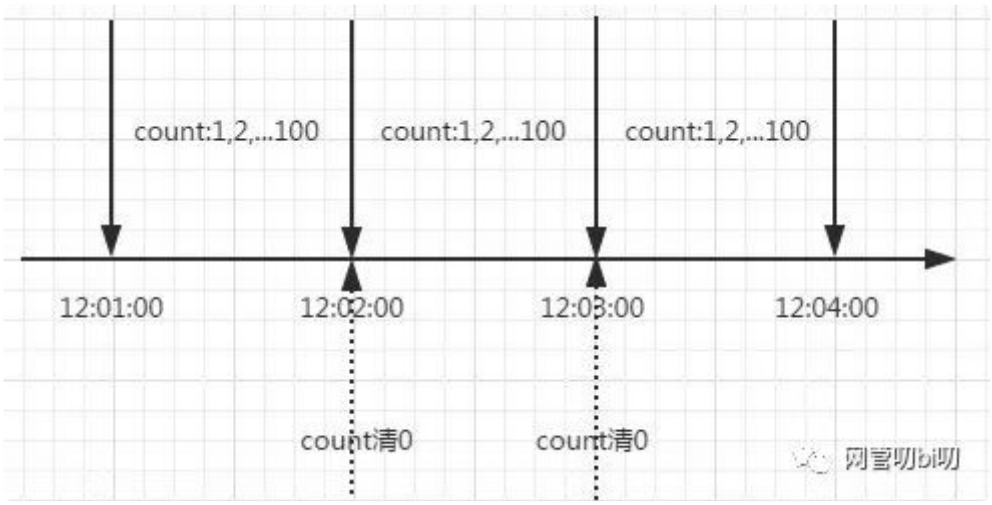
- 计数器
- 滑动窗口
- 漏桶
- 令牌桶

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

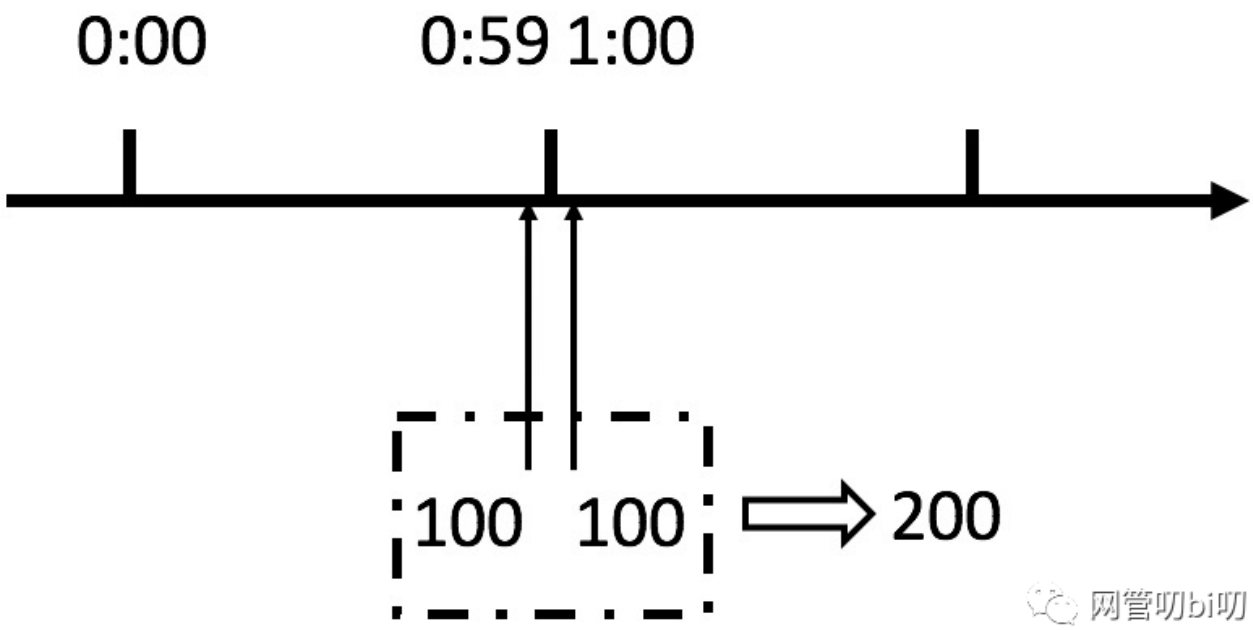
计数器是一种比较简单粗暴的限流算法，其思想是在固定时间窗口内对请求进行计数，与阀值进行比较判断是否需要限流，一旦到了时间临界点，将计数器清零。



固定时间窗口

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）



固定时间窗口

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.Duration) {
    limit.rate = rate
    limit.begin = time.Now()
    limit.cycle = cycle
    limit.count = 0
}

func (limit *LimitRate) Reset(begin time.Time) {
    limit.begin = begin
    limit.count = 0
}
```

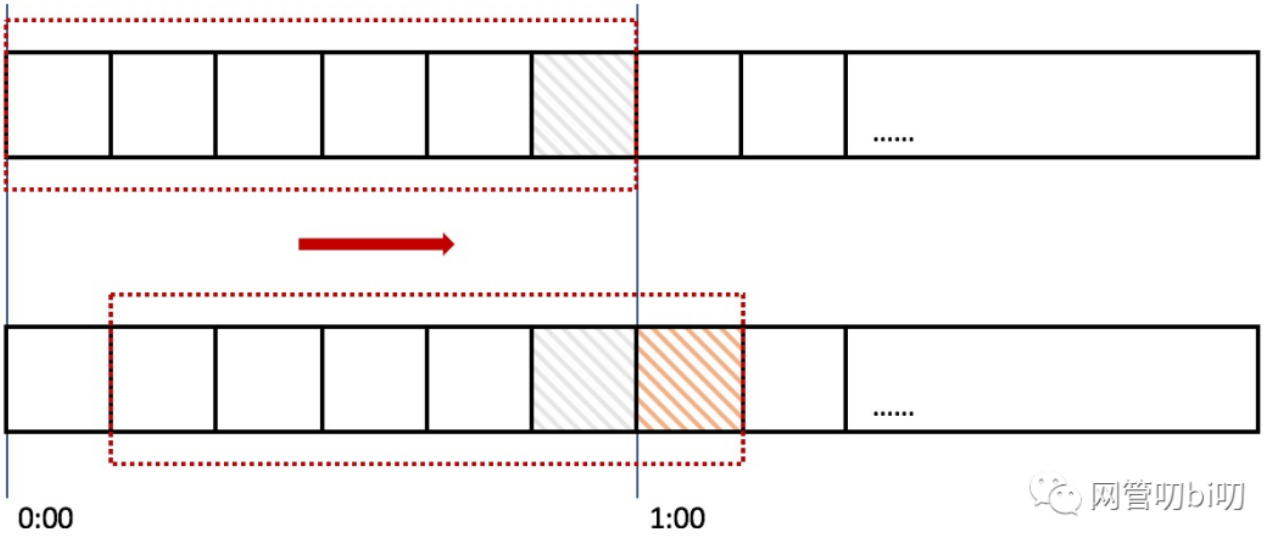

滑动窗口

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。



滑动窗口

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`timeSlot`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`timeslot`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
 }
 return count
}

type SlidingWindowLimiter struct {
 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 int) *Slidin
 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
 }
```

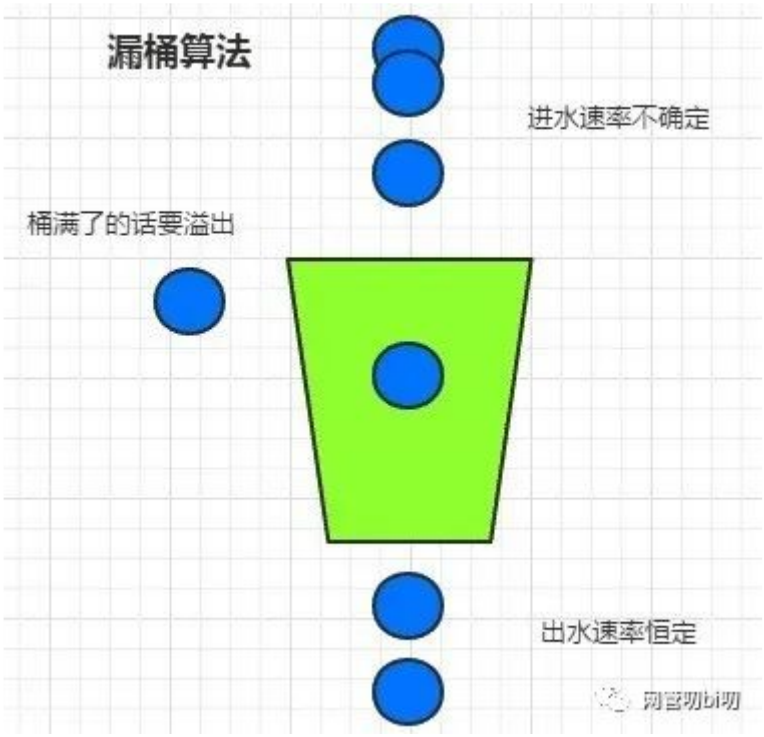
```
 // 记录这次的请求数
 var lastSlot *timeSlot
 if len(l.windows) > 0 {
  lastSlot = l.windows[len(l.windows)-1]
  if lastSlot.timestamp.Add(l.SlotDuration).Before(now) {
   // 如果当前时间已经超过这个时间插槽的跨度，那么新建一个时间插槽
   lastSlot = &timeSlot{timestamp: now, count: 1}
   l.windows = append(l.windows, lastSlot)
  } else {
   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**GitHub**链接 **https://github.com/kevinyan815/gocookbook/issues/26** 拿到，我把这个链接也放到了阅读原文里，点击即可访问。

漏桶

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。



漏桶

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

```
// 一个固定大小的桶，请求按照固定的速率流出
// 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
```

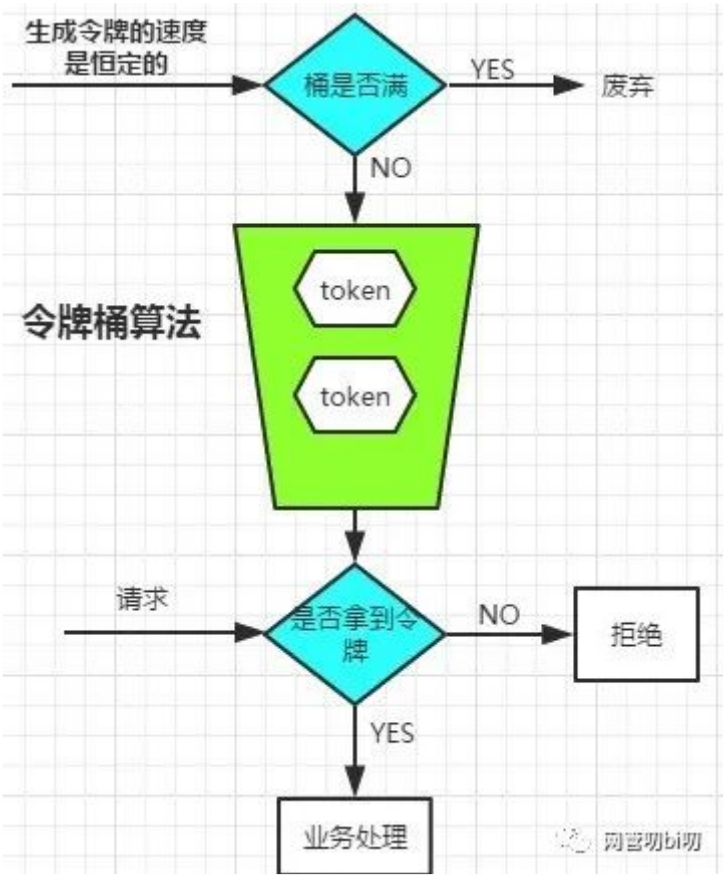
```
    lock       sync.Mutex // 锁
}

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.lastLeakMs) * leaky.rate / 1000)
    leaky.water = math.Max(0, leakyWater)
    leaky.lastLeakMs = now
    if leaky.water+1 <= leaky.capacity {
        leaky.water++
        return true
    } else {
        return false
    }
}

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。



令牌桶

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应对一定的突发流量。如果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，会造成大量的用户无法访问，对用户体验的损害比较大。

代码实现

```
type TokenBucket struct {
    rate         int64 //固定的token放入速率, r/s
    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 // 先添加令牌
    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()
}
```

Golang官方限流器的用法详解

kevinyan LV.5
2021年07月05日 12:22 · 阅读 5478

+ 关注

限流器是提升服务稳定性的非常重要的组件，可以用来限制请求速率，保护服务，以免服务过载。 限流器的实现方法有很多种，常见的限流算法有**固定窗口、滑动窗口、漏桶、令牌桶**，我在前面的文章**「常用限流算法的应用场景和实现原理」** 中给大家讲解了这几种限流方法自身的特点和应用场景，其中令牌桶在限流的同时还可以应对一定的突发流量，与互联网应用容易因为热点事件出现突发流量高峰的特点更契合。

> 简单来说，令牌桶就是想象有一个固定大小的桶，系统会以恒定速率向桶中放 Token，桶满则暂时不放。在请求比较的少的时候桶可以先"攒"一些Token，应对突发的流量，如果桶中有剩余 Token 就可以一直取。如果没有剩余 Token，则需要等到桶中被放置了 Token 才行。
>
> 关于令牌桶限流更详细的解释请参考文章：常用限流算法的应用场景和实现原理

有的同学在看明白令牌桶的原理后就非常想去自己实现一个限流器应用到自己的项目里，em... 怎么说呢，造个轮子确实有利于自己水平提高，不过要是应用到商用项目里的话其实大可不必自己去造轮子，Golang官方已经替我们造好轮子啦 ……~！

Golang 官方提供的扩展库里就自带了限流算法的实现，即 `golang.org/x/time/rate`。该限流器也是基于 Token Bucket(令牌桶) 实现的。

限流器的内部结构

`time/rate` 包的 `Limiter` 类型对限流器进行了定义，所有限流功能都是通过基于 `Limiter` 类型实现的，其内部结构如下：

go 复制代码

```go
type Limiter struct {
    mu     sync.Mutex
    limit  Limit
    burst  int // 令牌桶的大小
    tokens float64
```

```
    last time.Time // 上次更新tokens的时间
    lastEvent time.Time // 上次发生限速器事件的时间（通过或者限制都是限速器事件）
}
```

其主要字段的作用是：

- limit：`limit`字段表示往桶里放Token的速率，它的类型是Limit，是float64的类型别名。**设置`limit`时既可以用数字指定每秒向桶中放多少个Token，也可以指定向桶中放Token的时间间隔**，其实指定了每秒放Token的个数后就能计算出放每个Token的时间间隔了。
- burst: 令牌桶的大小。
- tokens: 桶中的令牌。
- last: 上次往桶中放 Token 的时间。
- lastEvent：上次发生限速器事件的时间（通过或者限制都是限速器事件）

可以看到在`timer/rate`的限流器实现中，并没有单独维护一个 Timer 和队列去真的每隔一段时间向桶中放令牌，而是仅仅通过计数的方式表示桶中剩余的令牌。每次消费取 Token 之前会先根据上次更新令牌数的时间差更新桶中Token数。

大概了解了`time/rate`限流器的内部实现后，下面的内容我们会集中介绍下该组件的具体使用方法：

构造限流器

我们可以使用以下方法构造一个限流器对象：

```go
limiter := rate.NewLimiter(10, 100);
```

这里有两个参数：

1. 第一个参数是`r Limit`，设置的是限流器Limiter的`limit`字段，代表每秒可以向 Token 桶中产生多少 token。Limit 实际上是 float64 的别名。
2. 第二个参数是`b int`，b 代表 Token 桶的容量大小，也就是设置的限流器 Limiter 的`burst`字段。

那么，对于以上例子来说，其构造出的限流器的令牌桶大小为 100, 以每秒 10 个 Token 的速率向桶中放置 Token。

除了给 `r Limit` 参数直接指定每秒产生的 Token 个数外，还可以用 Every 方法来指定向桶中放置 Token 的间隔，例如：

```go
limit := rate.Every(100 * time.Millisecond);
limiter := rate.NewLimiter(limit, 100);
```

以上就表示每 100ms 往桶中放一个 Token。本质上也是一秒钟往桶里放 10 个。

使用限流器

Limiter 提供了三类方法供程序消费 Token，可以每次消费一个 Token，也可以一次性消费多个 Token。 每种方法代表了当 Token 不足时，各自不同的对应手段，可以阻塞等待桶中Token补充，也可以直接返回取Token失败。

Wait/WaitN

```go
func (lim *Limiter) Wait(ctx context.Context) (err error)
func (lim *Limiter) WaitN(ctx context.Context, n int) (err error)
```

Wait 实际上就是 `WaitN(ctx,1)` 。

当使用 Wait 方法消费 Token 时，如果此时桶内 Token 数组不足 (小于 N)，那么 Wait 方法将会阻塞一段时间，直至 Token 满足条件。如果充足则直接返回。

这里可以看到，Wait 方法有一个 context 参数。 我们可以设置 context 的 Deadline 或者 Timeout，来决定此次 Wait 的最长时间。

```go
// 一直等到获取到桶中的令牌
err := limiter.Wait(context.Background())
if err != nil {
  fmt.Println("Error: ", err)
}

// 设置一秒的等待超时时间
```

```go
ctx, _ := context.WithTimeout(context.Background(), time.Second * 1)
err := limiter.Wait(ctx)
if err != nil {
  fmt.Println("Error: ", err)
}
```

Allow/AllowN

go 复制代码

```go
func (lim *Limiter) Allow() bool
func (lim *Limiter) AllowN(now time.Time, n int) bool
```

Allow 实际上就是对 `AllowN(time.Now(),1)` 进行简化的函数。

AllowN 方法表示，截止到某一时刻，目前桶中数目是否至少为 n 个，满足则返回 true，同时从桶中消费 n 个 token。反之不消费桶中的Token，返回false。

对应线上的使用场景是，如果请求速率超过限制，就直接丢弃超频后的请求。

go 复制代码

```go
if limiter.AllowN(time.Now(), 2) {
    fmt.Println("event allowed")
} else {
    fmt.Println("event not allowed")
}
```

Reserve/ReserveN

go 复制代码

```go
func (lim *Limiter) Reserve() *Reservation
func (lim *Limiter) ReserveN(now time.Time, n int) *Reservation
```

Reserve 相当于 `ReserveN(time.Now(), 1)`。

ReserveN 的用法就相对来说复杂一些，当调用完成后，无论 Token 是否充足，都会返回一个 `*Reservation` 对象。你可以调用该对象的 `Delay()` 方法，该方法返回的参数类型为

`time.Duration`，反映了需要等待的时间，必须等到等待时间之后，才能进行接下来的工作。如果不想等待，可以调用 `Cancel()` 方法，该方法会将 Token 归还。

举一个简单的例子，我们可以这么使用 Reserve 方法。

```go
r := limiter.Reserve()
f !r.OK() {
    // Not allowed to act! Did you remember to set lim.burst to be > 0 ?
    return
}
time.Sleep(r.Delay())
Act() // 执行相关逻辑
```

动态调整速率和桶大小

Limiter 支持创建后动态调整速率和桶大小：

1. SetLimit(Limit) 改变放入 Token 的速率
2. SetBurst(int) 改变 Token 桶大小

有了这两个方法，可以根据现有环境和条件以及我们的需求，动态地改变 Token 桶大小和速率。

总结

今天我们总结了 Golang 官方限流器的使用方法，它是一种令牌桶算实现的限流器。其中 **Wait/WaitN**，**Allow/AllowN** 这两组方法在平时用的比较多，前者是消费Token时如果桶中Token不足可以让程序等待桶中新Token的放入（最好设置上等待时长）后者则是在桶中的Token不足时选择直接丢弃请求。

除了Golang官方提供的限流器实现，Uber公司开源的限流器 `uber-go/ratelimit` 也是一个很好的选择，与Golang官方限流器不同的是Uber的限流器是通过漏桶算法实现的，不过对传统的漏桶算法进行了改良，有兴趣的同学可以自行去体验一下。

几种开源限流算法库/应用软件介绍和使用

一、Go time/rate 限流器

1.1 简介

Go 在 x 标准库，即 golang.org/x/time/rate 里自带了一个限流器，这个限流器是基于令牌桶算法（token bucket）实现的。

在上一篇文章讲了几种限流算法，里面就有令牌桶算法，具体可以看上篇文章介绍。

1.2 rate/time 限流构造器

这个限流构造器就是生成 token，供后面使用。

Limiter struct 结构：

```
// https://github.com/golang/time/blob/master/rate/rate.go#L55

// The methods AllowN, ReserveN, and WaitN consume n tokens.
type Limiter struct {
    mu sync.Mutex
    limit Limit    // 放入 token 的速率
    burst int      // 令牌桶限制最大值
    tokens float64 // 桶中令牌数
    // last is the last time the limiter's tokens field was updated
    last time.Time
    // lastEvent is the latest time of a rate-limited event (past or future)
    lastEvent time.Time
}
```

Copy

限流器构造方法：func NewLimiter(r Limit, b int) *Limiter：

> - r：产生 token 的速率。默认是每秒中可以向桶中生产多少 token。也可以设置这个值，用方法 Every 设置 token 速率时间粒度。
> - b：桶的容量，桶容纳 token 的最大数量。 b == 0，允许声明容量为 0 的值，这时拒绝所有请求；与 b== 0 情况相反，如果 r 为 inf 时，将允许所有请求，即使是 b == 0。

```
// Inf is the infinite rate limit; it allows all events (even if burst is zero).
const Inf = Limit(math.MaxFloat64)
```

Copy

> It implements a "token bucket" of size b, initially full and refilled at rate r tokens per second.
> 构造器一开始会为桶注入 b 个 token，然后每秒补充 r 个 token。

- 每秒生成 20 个 token，桶的容量为 5，代码为：

```
limiter : = NewLimiter(20, 5)
```

Copy

- 200ms 生成 1 个 token

这时候不是秒为单位生成 token ，就可以使用 Every 方法设置生成 token 的速率：

```
limit := Every(200 * time.Millisecond)
limiter := NewLimiter(limit, 5)
```

Copy

1秒 = 200ms * 5，也就是每秒生成 5 个 token。

生成了 token 之后，请求获取 token，然后使用 token。

1.3 time/rate 有3种限流用法

time/rate 源码里注释，消费 n 个 tokens 的方法

> // The methods AllowN, ReserveN, and WaitN consume n tokens.

- AllowN
- ReserveN
- WaitN

A. WaitN、Wait

WaitN / Wait 方法：

```
// https://pkg.go.dev/golang.org/x/time/rate#Limiter.WaitN
// WaitN blocks until lim permits n events to happen.
// It returns an error if n exceeds the Limiter's burst size, the Context is
// canceled, or the expected wait time exceeds the Context's Deadline.
// The burst limit is ignored if the rate limit is Inf.
func (lim *Limiter) WaitN(ctx context.Context, n int) (err error)

func (lim *Limiter) Wait(ctx context.Context) (err error)
```

> WaitN ： 当桶中的 token 数量小于 N 时，WaitN 方法将阻塞一段时间直到 token 满足条件或超时或取消(如果设置了context)，超时或取消将返回error。如果 N 充足则直接返回。
> Wait ： 就是 WaitN 方法中参数 n 为 1 时，即： `WaitN(ctx, 1)` 。

方法里还有 Contex 参数，所以也可以设置 Deadline 或 Timeout，来决定 Wait 最长时间。比如下面代码片段：

```
ctx, cancel := context.WithTimeout(context.Background(), time.Second * 5)
defer cancel()
err := limiter.WaitN(ctx, 2)
```

例子1：

```
package main

import (
    "context"
    "fmt"
    "time"

    "golang.org/x/time/rate"
)

func main() {
    limit := rate.NewLimiter(3, 5) // 每秒产生 3 个token，桶容量 5

    ctx, cancel := context.WithTimeout(context.Background(), time.Second*5)
    defer cancel() // 超时取消

    for i := 0; ; i++ { // 有多少令牌直接消耗掉
        fmt.Printf("%03d %s\n", i, time.Now().Format("2006-01-02 15:04:05.000"))
        err := limit.Wait(ctx)
        if err != nil { // 超时取消 err != nil
            fmt.Println("err: ", err.Error())
            return // 超时取消，退出 for
        }
    }
}
```

分析：这里指定令牌桶大小为 5，每秒生成 3 个令牌。for 循环消耗令牌，产生多少令牌都会消耗掉。
从开始一直到 5 秒超时，计算令牌数，一开始初始化 NewLimiter 的 5 个 + 每秒 3 个令牌 * 5秒 ， 总计 20 个令牌。运行程序输出看

看：

```
$ go run .\waitdemo.go
000 2022-05-17 21:35:38.400
001 2022-05-17 21:35:38.425
002 2022-05-17 21:35:38.425
003 2022-05-17 21:35:38.425
004 2022-05-17 21:35:38.425
005 2022-05-17 21:35:38.425
006 2022-05-17 21:35:38.773
007 2022-05-17 21:35:39.096
008 2022-05-17 21:35:39.436
009 2022-05-17 21:35:39.764
010 2022-05-17 21:35:40.106
011 2022-05-17 21:35:40.434
012 2022-05-17 21:35:40.762
013 2022-05-17 21:35:41.104
014 2022-05-17 21:35:41.430
015 2022-05-17 21:35:41.759
016 2022-05-17 21:35:42.104
017 2022-05-17 21:35:42.429
018 2022-05-17 21:35:42.773
019 2022-05-17 21:35:43.101
err:  rate: Wait(n=1) would exceed context deadline
```

B: AllowN、Allow

AllowN / Allow 方法

```
// https://pkg.go.dev/golang.org/x/time/rate#Limiter.AllowN
// AllowN reports whether n events may happen at time now.
// Use this method if you intend to drop / skip events that exceed the rate limit.
// Otherwise use Reserve or Wait.
func (lim *Limiter) AllowN(now time.Time, n int) bool

// Allow is shorthand for AllowN(time.Now(), 1).
func (lim *Limiter) Allow() bool
```

> AllowN ：截止到某一时刻，桶中的 token 数量至少为 N 个，满足就返回 true，同时从桶中消费 n 个 token；反之返回 false，不消费 token。这个实际就是丢弃某些请求。
> Allow ：就是 AllowN 方法中参数 now 为现在时间，n 为 1，即 `AllowN(time.Now(), 1)`

例子：

```
package main

import (
    "fmt"
    "net/http"
    "time"

    "golang.org/x/time/rate"
)

func main() {
    r := rate.Every(1 * time.Millisecond)
    limit := rate.NewLimiter(r, 10)

    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        if limit.Allow() {
            fmt.Printf("success，当前时间：%s\n", time.Now().Format("2006-01-02 15:04:05"))
        } else {
            fmt.Printf("success，但是被限流了。。。\n")
        }
```

```
    })

    fmt.Println("http start ... ")
    _ = http.ListenAndServe(":8080", nil)

}
```

然后你可以找一个 http 测试工具模拟用户压测下，比如 https://github.com/rakyll/hey 这个工具。测试命令：

```
hey -n 100 http://localhost:8080/
```

Copy

就可以看到输出的内容

> ... ...
> success，当前时间：2022-05-17 21:41:44
> success，当前时间：2022-05-17 21:41:44
> success，当前时间：2022-05-17 21:41:44
> success，但是被限流了。。。
> success，但是被限流了。。。
> ... ...

例子2：

```
package main

import (
    "fmt"
    "time"

    "golang.org/x/time/rate"
)

func main() {
    limit := rate.NewLimiter(1, 3)
    for {
        if limit.AllowN(time.Now(), 2) {
            fmt.Println(time.Now().Format("2006-01-02 15:04:05"))
        } else {
            time.Sleep(time.Second * 3)
        }
    }
}
```

Copy

C：ReserveN、Reserve

ReserveN / Reserve 方法

```
// https://pkg.go.dev/golang.org/x/time/rate#Limiter.ReserveN
// ReserveN returns a Reservation that indicates how long the caller must wait before n events happen. The Limiter takes
this Reservation into account when allowing future events. The returned Reservation’s OK() method returns false if n
exceeds the Limiter's burst size.
func (lim *Limiter) ReserveN(now time.Time, n int) *Reservation

func (lim *Limiter) Reserve() *Reservation

func (r *Reservation) DelayFrom(now time.Time) time.Duration
func (r *Reservation) Delay() time.Duration
func (r *Reservation) OK() bool
```

Copy

其实上面的 WaitN 和 AllowN 都是基于 ReserveN 方法。具体可以去看看这 3 个方法的源码。

> ReserveN ： 此方法返回 *Reservation 对象。你可以调用该对象的 Dealy 方法，获取延迟等待的时间。如果为 0，则不用等待。必须等到等待时间结束后才能进行下面的工作。
> 或者，如果不想等待，可以调用 Cancel 方法，该方法会将 Token 归还。
> Reserve ： 就是 ReserveN 方法中参数 now 为现在时间，n 为 1，即 `AllowN(time.Now(), 1)`

usage example:

```
// https://pkg.go.dev/golang.org/x/time/rate#Limiter.ReserveN

r := lim.ReserveN(time.Now(), 1)
if !r.OK() {
  // Not allowed to act! Did you remember to set lim.burst to be > 0 ?
  return
}
time.Sleep(r.Delay())
Act() // 执行相关逻辑
```

1.4 动态设置桶token容量和速率

SetBurstAt / SetBurst:

```
func (lim *Limiter) SetBurstAt(now time.Time, newBurst int)
func (lim *Limiter) SetBurst(newBurst int)
```

> SetBurstAt ： 设置到某时刻桶中 token 的容量
> SetBurst ： SetBurstAt(time.Now())

SetLimitAt / SetLimit

```
func (lim *Limiter) SetLimitAt(now time.Time, newLimit Limit)
func (lim *Limiter) SetLimit(newLimit Limit)
```

> SetLimitAt ： 设置某刻 token 的速率
> SetLimit ： 设置 token 的速率

二、uber 的 rate limiter

2.1 简介

uber 的这个限流算法是 `漏桶算法（leaky bucket）` - github.com/uber-go/ratelimit。

与令牌桶算法的区别：

> 1. 漏桶算法流出的速率可以控制，流进桶中请求不能控制
> 2. 令牌桶算法对于流入和流出的速度都是可以控制的，因为令牌可以自己生成。所以它还可以应对突发流量。突发流量生成 token 就快些。
> 3. 令牌桶算法只要桶中有 token 就可以一直消费，漏桶是按照预定的间隔顺序进行消费的。

2.2 使用

官方的例子：

```
limit := ratelimit.New(100) // 每秒钟允许100个请求

prev := time.Now()

for i := 0; i < 10; i++ {
    now := limit.Take()
    fmt.Println(i, now.Sub(prev))
```

```
    prev = now
}
```

限流器每秒可以通过 100 个请求，平均每个间隔 10ms。

2.3 uber 对漏桶算法的改进

在传统的漏桶算法，每个请求间隔是固定的，然而在实际应用中，流量不是这么平均的，时而小时而大，对于这种情况，uber 对 leaky bucket 做了一点改进，引入 maxSlack 最大松弛量的概念。

举例子：比如 3 个请求，请求 1 完成，15ms后，请求 2 才到来，可以对 2 立即处理。请求 2 完成后，5ms后，请求 3 到来，这个请求距离上次请求不足 10ms，因此要等 5ms。
但是，对于这种情况，实际三个请求一共耗时 25ms 才完成，并不是预期的 20ms。

uber 的改进是：可以把之情请求间隔比较长的时间，匀给后面的请求使用，只要保证每秒请求数即可。

uber ratelimit 改进代码实现：

```
t.sleepFor += t.perRequest - now.Sub(t.last)
if t.sleepFor > 0 {
  t.clock.Sleep(t.sleepFor)
  t.last = now.Add(t.sleepFor)
  t.sleepFor = 0
} else {
  t.last = now
}
```

把每个请求多余出来的等待时间累加起来，以给后面的抵消使用。

其他参数用法：

- WithoutSlack：

ratelimit 中引入最大松弛量，默认的最大松弛量为 10 个请求的间隔时间。
但是我不想用这个最大松弛量呢，就要限制请求的固定间隔时间，用 WithoutSlack 这个参数限制：

```
limit := ratelimit.New(100, ratelimit.WithoutSlack)
```

- WithClock(clock Clock)：

ratelimit 中时间相关计算是用 go 的标准时间库 time，如果想要更高进度或特殊需求计算，可以用 WithClock 参数替换，实现 Clock 的 interface 就可以了

```
type Clock interface {
        Now() time.Time
        Sleep(time.Duration)
}

clock &= MyClock{}
limiter := ratelimit.New(100, ratelimit.WithClock(clock))
```

更多 ratelimit

三、其他限流器，算法库包和软件

1. 滴滴的 tollbooth，http 限流中间件，有很多特性：
    - 1.基于IP，路径，方法，header，授权用户等限流
    - 2.通过使用 `LimitByKeys()` 组合你自己的中间件
    - 3.对于head项和基本auth能够设置TTL-过期时间
    - 4.拒绝后，可以使用以下 HTTP 头响应，比如 `X-Rate-Limit-Limit` The maximum request limit
    - 5.当限流达到上限，可以自定义消息和方法，返回信息

- 6.它是基于 golang.org/x/time/rate 开发

2. java 的 guava 限流
   - ratelimiter
3. 基于信号量限流
   - https://github.com/golang/net/blob/master/netutil/listen.go
4. sentinel-go 服务治理软件以及sentinel
   - https://github.com/alibaba/sentinel-golang
   - github.com/alibaba/Sentinel ，java编写的流控组件，服务治理
5. 还有各种基于 nginx 的限流器，限流软件-服务网关，api gateway等

四、参考

- https://github.com/golang/time
- pkg.go.dev/golang.org/x/time/rate
- https://zhuanlan.zhihu.com/p/100594314
- https://segmentfault.com/a/1190000023033365
- https://www.cyhone.com/articles/usage-of-golang-rate/
- https://www.cyhone.com/articles/analysis-of-uber-go-ratelimit
- uber ratelimit vs go time/rate demo
- uber-go ratelimit