限流中间件开发

联系QQ：2816010068，加入会员群

二叉树出品

目录

限流简介

- 定义
  - 限制流量，保证在突然流量的情况下，系统还能够正常运行

- 限流的意义
  - 保护有限的资源，不会被突发的大流量冲击而崩溃

限流的例子

- 水库限流

限流的例子

- 景区限流

生活中的限流算法

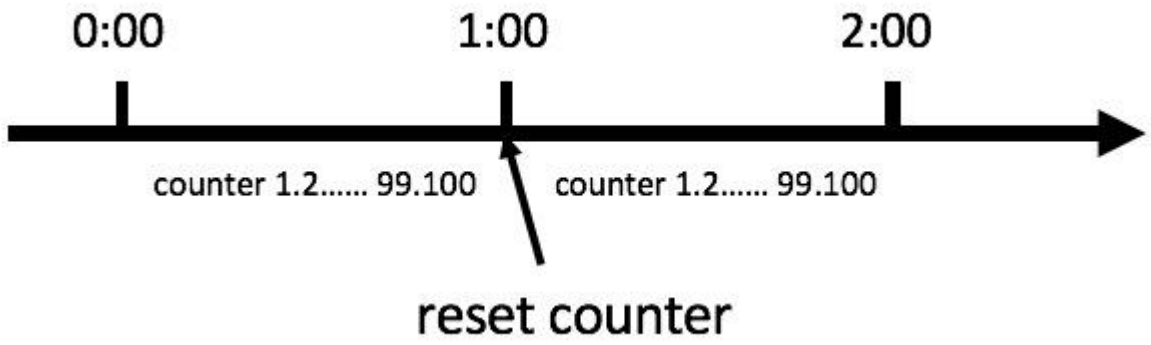
- 排队
- 拒绝

服务限流

- 在突发的流量下，通过限制用户访问的流量，保证服务能够正常运行
- 常见的限流思路
  - 排队
    - 应用场景：秒杀抢购，用户点击抢购之后，进行排队，直到抢到或售罄为止
  - 拒绝
    - 应用场景：除秒杀之外的任何场景
- 限流算法
  - 计数器限流算法
  - 漏桶限流算法
  - 令牌桶限流算法

计数器限流算法

- 在单位时间内进行计数，如果大于设置的最大值，则进行拒绝
- 如果过了单位时间，则重新进行计数



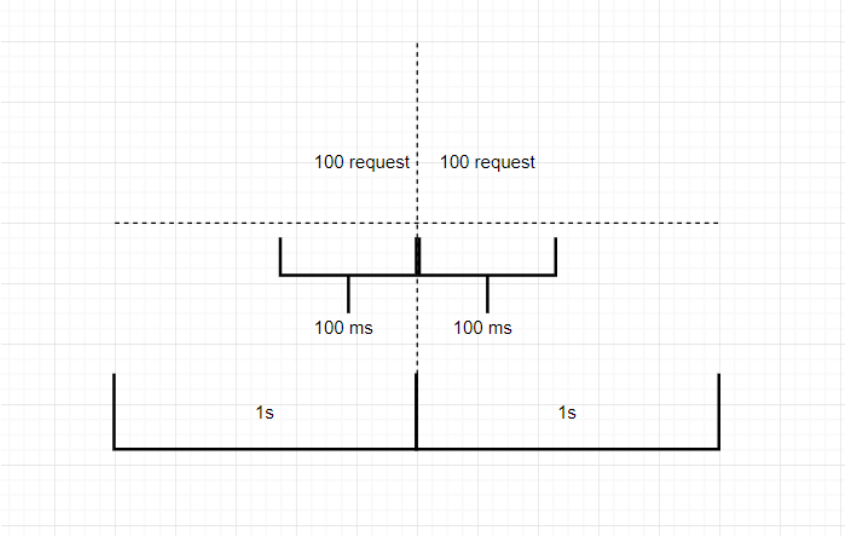
计数器限流

- 实现思路

```go
type CounterLimit struct {
    counter      int64       //计数器
    limit        int64       //指定时间窗口内允许的最大请求数
    intervalNano int64       //指定的时间窗口
    unixNano     int64       //unix时间戳,单位为纳秒
}

func NewCounterLimit(interval time.Duration, limit int64) *CounterLimit {

    return &CounterLimit{
        counter:      0,
        limit:        limit,
        intervalNano: int64(interval),
        unixNano:     time.Now().UnixNano(),
    }
}

func (c *CounterLimit) Allow() bool {

    now := time.Now().UnixNano()
    if now-c.unixNano > c.intervalNano {    //如果当前过了当前的时间窗口,则重新进行计数
        atomic.StoreInt64(&c.counter, 0)
        atomic.StoreInt64(&c.unixNano, now)
        return true
    }

    atomic.AddInt64(&c.counter, 1)
    return c.counter < c.limit            //判断是否要进行限流
}
```

计数器限流算法

- 优点
  - 实现非常简单
- 缺点
  - 突发流量会出现毛刺现象
    - 比如一秒限流100个请求，前100ms内处理完了100个请求，后900ms时间内没有请求处理
  - 计数不准确



漏桶限流算法

- 一个固定大小的水桶
- 以固定速率流出
- 水桶满了，则进行溢出(拒绝)

漏桶限流算法

- 实现思路

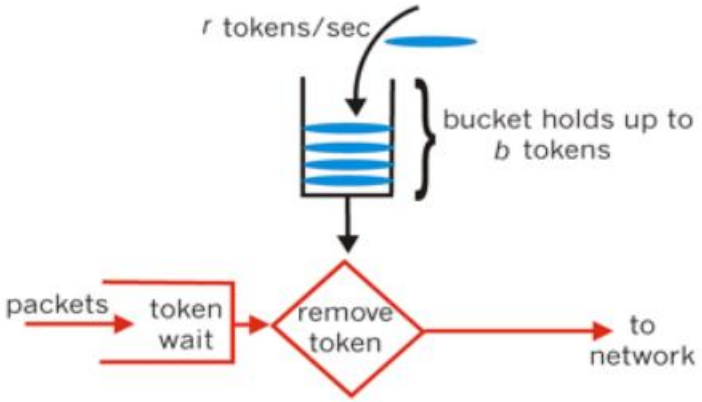
```go
type BucketLimit struct {
    rate       float64 //漏桶中水的漏出速率
    bucketSize float64 //漏桶最多能装的水大小
    unixNano   int64   //unix时间戳
    curWater   float64 //当前桶里面的水
}

func NewBucketLimit(bucketSize int64, rate float64) *BucketLimit {
    return &BucketLimit{
        bucketSize: float64(bucketSize),
        rate:       rate,
        unixNano:   time.Now().UnixNano(),
        curWater:   0,
    }
}

func (b *BucketLimit) reflesh() {
    now := time.Now().UnixNano()
    //时间差，把纳秒换成秒
    diffSec := float64(now-b.unixNano) / 1000 / 1000 / 1000
    b.curWater = math.Max(0, b.curWater-diffSec*b.rate)
    b.unixNano = now
    return
}
```

漏桶限流算法

- 优点
  - 解决了计数器限流算法的毛刺问题
  - 整体流量控制的比较平稳
- 缺点
  - 无法应对某些突发的流量

令牌桶限流算法

- 一个固定大小的水桶
- 以固定速率放入token
- 如果能够拿到token则处理，否则拒绝



令牌桶限流算法

- 优点
  - 不限制流速，能够应对突发流量

Koala架构优化

- 封装KoalaServer类,用来聚合服务相关的所有功能
- 把当前服务需要什么middleware从middleware拆分出来,放到server库中
  - middleware更加关注中间件的实现
  - server决定当前服务要使用哪些中间件
- Main生成器改造，改造后生成的main函数更加简洁了
- Router生成器改造，原来调用middleware.BuildServerMiddleware改成调用server.BuildMiddleware

限流中间件开发

- 抽象限流接口
  - 不关注具体实现
  - 用户根据自己的需要传入相关的限流器就可以
  - 更加灵活

```
type Limiter interface {
    Allow() bool
}
```

限流中间件开发

```go
type Limiter interface {
    Allow() bool
}

func NewRateLimitMiddleware(l Limiter) Middleware {
    return func(next MiddlewareFunc) MiddlewareFunc {
        return func(ctx context.Context, req interface{}) (resp interface{}, err error) {
            allow := l.Allow()
            if !allow {
                err = status.Error(codes.ResourceExhausted, "rate limited")
                return
            }

            return next(ctx, req)
        }
    }
}
```