Golang频率控制

奉有泉

服务稳定性

- 异地容灾
- 集群隔离
- 最小依赖
- 自动降级
- 频率控制


18
E

频率控制是什么

- 限定特定时间内某一操作的次数
    - 流量控制：RPC次数
    - 资源控制：数据库、图片、文件
- 从而达到控制使用者行为以及保护自身的目的
    - 策略性：限制用户在一分钟内的登录行为
    - 稳定性：限制单实例的整体访问速度

方案

- 接入层配置（nginx）
  - Pros： 成熟方案，无需开发
  - Cons：灵活度有限，较多依赖运维
- 独立proxy实现
  - Pros： 入口收敛，定制型强
  - Cons：引入时延，并引入风险点
- 服务内嵌模块
  - Pros： 独立灵活，风险低
  - Cons：服务构建依赖

实现

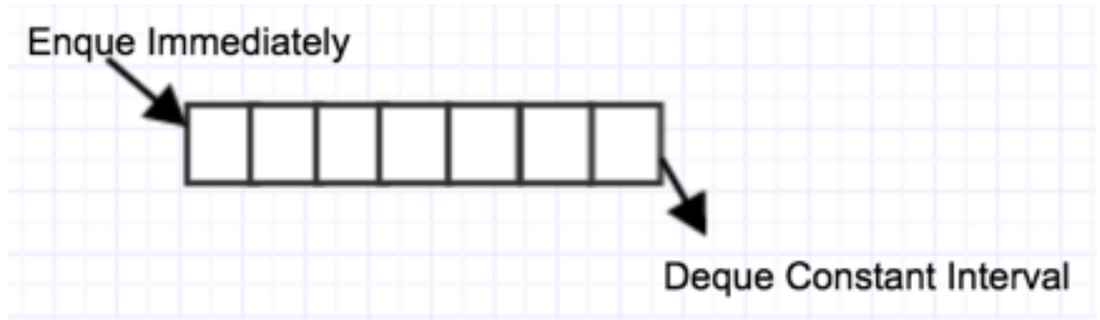
- 单实例实现
    - 针对单个服务/单个主机
- 集群控制实现
    - 针对一组服务
- 频率控制服务实现
    - 独立的频率控制服务

单实例实现

- 以稳定性为主要目标
- 无依赖
- 无性能问题

v1-timer

- 使用timer
    - Pros：简单，速率准确
    - Cons: 引起请求的RTT大；不适合高速率




```go
func v1_timer() {
    ticker := time.Tick(time.Millisecond * 200)

    for range ticker {
        req := <-g_requests
        fmt.Println(time.Now(), "Handling request ", req)
    }
}
```

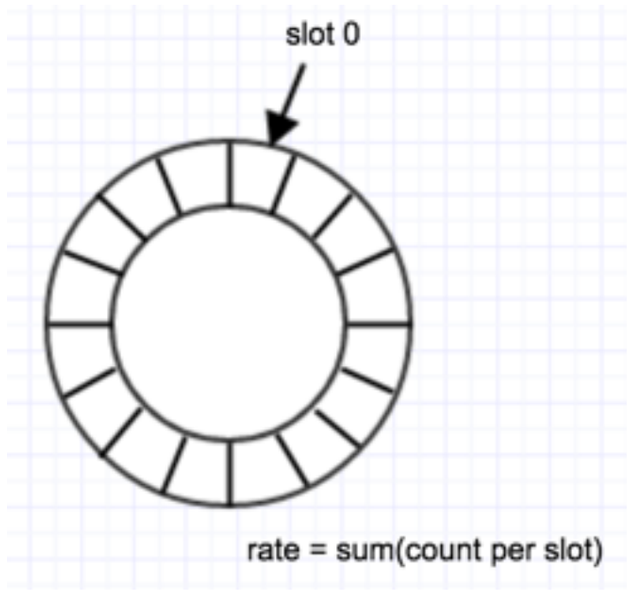
v2-counter

- 使用counter
    - Pros: 简单，支持高速率
    - Cons: 不能容忍burst

```go
func v2_counter() {
    ticker := time.Tick(time.Second)

    var count int64
    go func() {
        for range ticker {
            atomic.StoreInt64(&count, 0)
        }
    }()

    for req := range g_requests {
        if count < 5 {
            atomic.AddInt64(&count, 1)
            fmt.Println(time.Now(), "Handling request ", req)
        } else {
            fmt.Println(time.Now(), "Denying request ", req)
        }
    }
}
```

v3-ring

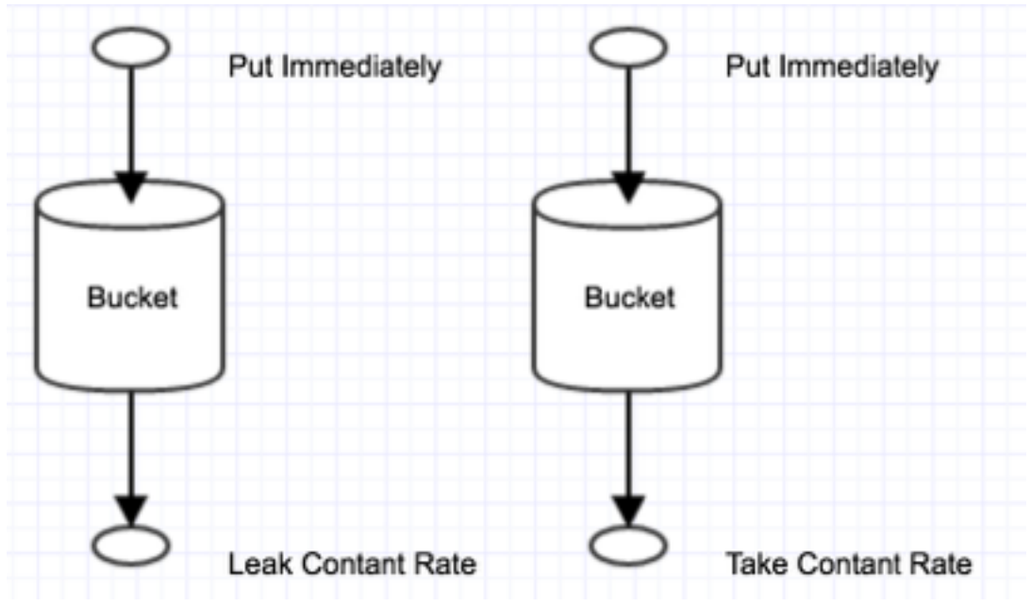
- ring
    - Pros: 能应对高速率，容忍burst，控制精度
    - Cons: 不易正确实现




```
func v3_ring() {
    numTotal := uint32(10)
    numPerSlot := uint32(2)
    slots := make([]uint32, 5)
    resolution := time.Millisecond * 200
    head := uint32(0)

    for req := range g_requests {
        slot := uint32(time.Now().UnixNano() / resolution)

        if slot != head {
            for i := head + 1; i <= slot && (i-head) < 5; i++ {
                rl.slots[i%5] = 0
            }
            atomic.StoreUint32(head, slot)
        }

        if (slots[slot%5] < 2*numPerSlot) && (1 /*sum()*/ < numTotal) {
            atomic.AddUint32(&slots[slot%5], 1)
            fmt.Println(time.Now(), "Handling request ", req)
        } else {
            fmt.Println(time.Now(), "Denying request ", req)
        }
    }
}
```

v4-Leaky Bucket

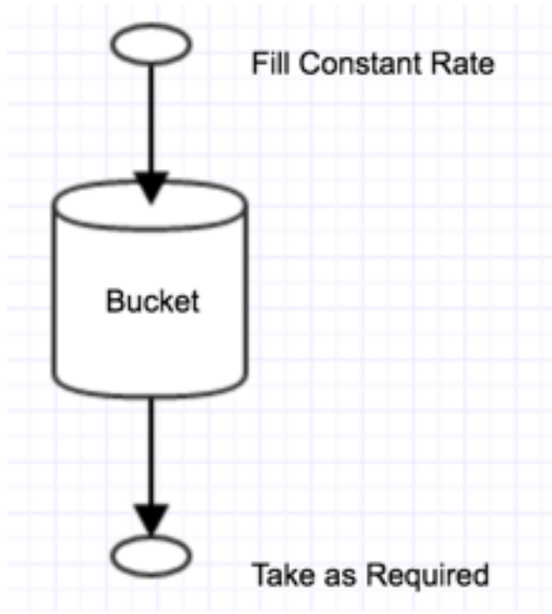
- 漏桶（Leaky Bucket）
    - Pros: 准确，易实现
    - Cons: 不能容忍burst




```
type bucket struct {
     capacity  uint
     remaining uint
     reset     time.Time
     rate      time.Duration
     mutex     sync.Mutex
}

func (b *bucket) v4_leakyAdd(amount uint) (leakybucket.BucketState, error) {
     b.mutex.Lock()
     defer b.mutex.Unlock()
     if time.Now().After(b.reset) {
          b.reset = time.Now().Add(b.rate)
          b.remaining = b.capacity
     }
     if amount > b.remaining {
          return leakybucket.BucketState{Capacity: b.capacity, Remaining: b.remaining, Reset: b.reset}, leakybucket.ErrorFull
     }
     b.remaining -= amount
     return leakybucket.BucketState{Capacity: b.capacity, Remaining: b.remaining, Reset: b.reset}, nil
}
```

v5-Token Bucket

- 令牌桶（Token Bucket）
    - Pros：高效准确，容忍burst




```
type Bucket struct {
	startTime    time.Time
	capacity     int64
	quantum      int64
	fillInterval time.Duration
	mu sync.Mutex
	avail     int64
	availTick int64
}

func (tb *Bucket) v5_tokenTake(now time.Time, count int64, maxWait time.Duration) (time.Duration, bool) {
	if count <= 0 {
		return 0, true
	}
	tb.mu.Lock()
	defer tb.mu.Unlock()

	currentTick := tb.adjust(now)
	avail := tb.avail - count
	if avail >= 0 {
		tb.avail = avail
		return 0, true
	}
	endTick := currentTick + (-avail+tb.quantum-1)/tb.quantum
	endTime := tb.startTime.Add(time.Duration(endTick) * tb.fillInterval)
	waitTime := endTime.Sub(now)
	if waitTime > maxWait {
		return 0, false
	}
	tb.avail = avail
	return waitTime, true
}
```

集群控制实现

- 以策略控制为主要目的
- 依赖于存储服务
- 性能受限

v1-timeout

- 超时
    - Pros: 简单
    - Cons: 容易产生burst

```
func v1_timeout() error {
    n, err := redis.Get("counter")

    if err != nil {
        redis.Set("counter", 1)
        redis.Expire("counter", 60)
    } else if n > 10 {
        return errors.New("Denying request")
    } else {
        redis.Incr("counter", 1)
    }
}
```

v2-Token Bucket

- 令牌漏桶（Token Bucket）
    - Pros: 准确高效
    - Cons: 不易实现，有并发问题

```
func v2_tokenbucket() {
    redis.Set("counter", 10)
    redis.Set("timestamp", time.Now())

    redis.Get("timestamp")
    // calculating...
    redis.Set("counter", 8)
    redis.Set("timestamp", time.Now())
}
```

v3-sliding

- 滑动窗口 Sorted Set
    - Pros: 精确控制速率，有效防止burst
    - Cons: 效率，需要有降级方案

```
func v3_sortedset() {
    redis.ZRemRangeByScore("counter", 60)
    n := redis.ZCount("counter")

    if n > 10 {
        return errors.New("Denying request")
    } else {
        redis.ZAdd("counter", time.Now())
    }
}
```

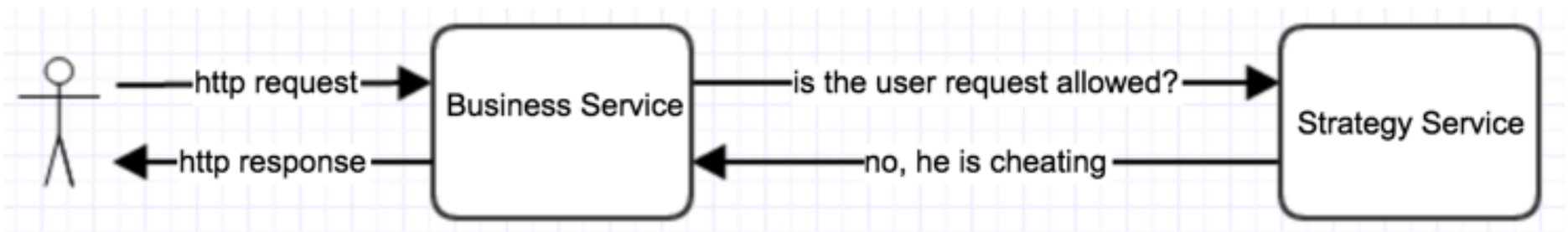
频率控制服务实现

- 以策略控制为主要目的
- 集中控制策略
- 配置管理灵活
- 需专门维护

频率服务器使用

- RPC调用
- 需要降级处理

```
request:  'http://www.freqserver.didichuxing.com/user/123456?type=access&api=login'

response: '{"allowed":"yes","errno":0,"errmsg":"OK"}'
```



Thank You!