目　录

1 前言

1.1 实验概要

在本次实训作业中，我们实现了基于令牌桶的多用户限流器，并进行多并发限流测试，给出了测试结果报告和设计方案.

通过本次实验，逐渐掌握 Go 语言的使用，提高查阅文献资料、编程开发以及文档书写的能力，提高分析问题和解决问题的能力。

2 设计方案

2.1 实验环境

- 实现语言：Golang 1.18.3
- 开发工具：Goland
- 操作系统：MacOS 12.4

2.2 设计思路

我们实现了基于令牌桶的多用户限流器。我们假定场景中有多个用户，每个用户都有一个本地限流器。当用户本地的限流器不能够及时处理到达的请求时，将请求发给中心限流器进行处理。并且在运行过程中，能够及时打印日志信息。

2.2.1 中心限流器实现

我们假定服务中心有一个中心限流器（容量更大，速率更快）。

```
// Bucket 令牌桶配置
type Bucket struct {
    Max   int64 //令牌桶的最大存储上限
    Cycle int64 //生成一批令牌的周期（每 {cycle} 毫秒生产一批令牌）
    Batch int64 //每批令牌的数量
    Residue int64 //令牌桶剩余空间
}
```

该限流器使用原子化操作，并且开启了一个协程，使得令牌数能够随代码运行而增加。部分代码如下：

```
func (bucket *Bucket) NewTokenLimiter() {
    //初始化令牌桶的剩余空间
    bucket.Residue = bucket.Max
    go func() {
        for {
            // 间隔一段时间发放令牌
            time.Sleep(time.Millisecond * time.Duration(bucket.Cycle))
            // 如果令牌数未超过上限，则继续累加
            if bucket.Residue+bucket.Batch <= bucket.Max {
                atomic.AddInt64(&bucket.Residue, bucket.Batch)
                continue
            } else {
                // 如果令牌数超过上限，则将令牌数设为上限值 max
                atomic.StoreInt64(&bucket.Residue, bucket.Max)
            }
        }
    }()
}
```

2.2.2 用户本地限流器

定义用户本地限流器如下：

```
type LimiterValue struct {
    MaxPermits string
    Rate       string
}

type LimiterAgent struct {
    spec LimiterValue
    pool *redis.Pool
    lock sync.Mutex
}
```

其中，使用的 redis 池为：

```
pool := &redis.Pool{
        //最大活跃连接数，0 代表无限
```

```
        MaxActive: 0,
        //最大闲置连接数
        MaxIdle: 2000,
        //闲置连接的超时时间
        IdleTimeout: time.Second * 100,
        //定义拨号获得连接的函数
        Dial: func() (redis.Conn, error) {
            //fmt.Println("redis.Host is ", conf.GetRedis().Host)
            c, err := redis.Dial("tcp", "127.0.0.1:6379")
            if err != nil {
                fmt.Println("when connect redis, happen error: ", err)
                return nil, err
            }
            return c, err
        },
    }
```

用户处理请求的代码如下：

```
// HandleRequest 处理请求：先获得锁，再处理
func (l *LimiterAgent) HandleRequest(user string, numRequest int64) (bool, error) {
    lockKey := getLimiterLockKey(user)

    // 获得锁
    for {
        lock, err := l.limiterGetLock(lockKey)
        if err != nil {
            return false, err
        }
        if lock {
            break
        }
    }

    finished, err := l.DoLimit(user, numRequest, time.Now().UnixNano())
    if finished != true {
        return false, err
    }
```

```
    // 解锁
    if err := l.limiterUnLock(lockKey); err != nil {
        return false, err
    }

    return true, nil
}
```

其中，上锁操作可以概述为获取 REDIS 池中的连接和 redis 中的 set userKey 1 EX 1 NX[1]。解锁的操作为获取 REDIS 池中的连接和 redis 中的 del userKey. 通过这种上锁操作可以不让么某用户同时进行多条请求

其中，进行限流操作的代码主要在 Do 函数中：

```
func (l *LimiterAgent) Do(conn redis.Conn, user string, key string, currNanoSec int64,
    numRequest int64, maxPermits int64, rate int64) (finished bool, err error) {
    // 取当前纳秒数和 token 数量，若没有则初始化
    lastNanoSec, _ := redis.String(conn.Do("HGET", key, "lastNanoSec"))
    currPermits, _ := redis.String(conn.Do("HGET", key, "currPermits"))
    if lastNanoSec == "" {
        conn.Do("HSET", key, "lastNanoSec", currNanoSec)
        conn.Do("HSET", key, "currPermits", maxPermits-numRequest)
        currPermits = strconv.FormatInt(maxPermits-numRequest, 10)
        lastNanoSec = strconv.FormatInt(currNanoSec, 10)
    }

    // 计算当前保留了多少个 token
    lastNanoSecInt64, _ := strconv.ParseInt(lastNanoSec, 10, 64)
    reservePermits := math.Ceil(float64((currNanoSec
            - lastNanoSecInt64) / int64(math.Pow(10, 9)) * rate))

    // 保留上次访问时间
    conn.Do("HSET", key, "lastNanoSec", currNanoSec)

    // 计算当前数量
    var current float64
    currPermitsFloat64, _ := strconv.ParseFloat(currPermits, 64)
    if reservePermits+currPermitsFloat64 > float64(maxPermits) {
```

[1]即设置 userKey 数值为 1，并在一秒后驱逐它

```
        reservePermits = float64(maxPermits) - currPermitsFloat64
        current = float64(maxPermits)
    } else {
        current = reservePermits + currPermitsFloat64
    }

    // 处理请求，并写入处理请求后的 token 数量
    remaining := current - float64(numRequest)
    if remaining >= 0 {
        _, err = conn.Do("HSET", key, "currPermits", remaining)
        str := fmt.Sprintf("User Handle: %s successly handle %d requests\n", user, num
        str += fmt.Sprintf("\t\t\t\t\t\t\tDetails: \n")
        str += fmt.Sprintf("\t\t\t\t\t\t\t\tCurrent Requests: %d\n", numRequest)
        str += fmt.Sprintf("\t\t\t\t\t\t\t\tUser Info: username: %s, resvervePermits:
        str += fmt.Sprintf("\t\t\t\t\t\t\t\tBucket Info: Residue: %d, Max: %d, Cycle:
        logger.WriterLog(str)
        if err != nil {
            return false, err
        }
        return true, nil
    } else {
        // 本地用户的 token 不够了，向全局请求
        isOk := centerBucket.GetToken(numRequest)
        // 全局请求处理失败
        if isOk == false {
            // 这个请求处理不了
            str := fmt.Sprintf("Can't handle: %s's %d requests, because token in %s's
            str += fmt.Sprintf("\t\t\t\t\t\t\tDetails: \n")
            str += fmt.Sprintf("\t\t\t\t\t\t\t\tCurrent Requests: %d\n", numRequest)
            str += fmt.Sprintf("\t\t\t\t\t\t\t\tUser Info: username: %s, resvervePermi
            str += fmt.Sprintf("\t\t\t\t\t\t\t\tBucket Info: Residue: %d, Max: %d, Cyc
            logger.WriterLog(str)
            _, err = conn.Do("HSET", key, "currPermits", current)
            if err != nil {
                return false, err
            }
        } else {
            // 向全局请求成功
            str := fmt.Sprintf("center Handle: %s get help from centerBucket to handle
```

```
            str += fmt.Sprintf("\t\t\t\t\t\t\tDetails: \n")
            str += fmt.Sprintf("\t\t\t\t\t\t\t\tCurrent Requests: %d\n", numRequest)
            str += fmt.Sprintf("\t\t\t\t\t\t\t\tUser Info: username: %s, resvervePermi
            str += fmt.Sprintf("\t\t\t\t\t\t\t\tBucket Info: Residue: %d, Max: %d, Cyc
            logger.WriterLog(str)
            _, err = conn.Do("HSET", key, "currPermits", current)
            if err != nil {
                return false, err
            }
            return true, nil
        }
        return false, nil
    }
}
```

我们在本地运行 REDIS 池，以保存每个用户的最近访问时间以及剩余令牌数, 如下图。

限流处理流程如下：

- 先获取当前剩余的令牌数和上次访问的时间，如果无法在 redis 中找到，则进行初始化。
- 然后，根据当前时间和上次访问的时间，计算增加了多少个令牌，并将这些令牌加入令牌桶中 (可能会有部分令牌由于桶满而被丢弃)
- 最后处理请求，如果当前桶中的令牌够用，则请求被顺利处理。否则，则向中心限流器进行请求，若请求被顺利处理，则返回 true，不然就进行限流操作，请求失败。

3 测试结果

分别基于 Go 语言中的 Test 和 BenchTest 进行多用户多请求的高并发测试, 并且顺利通过测试.

```
--- PASS: TestLimiterAgent_HandleRequest (102.81s)
PASS
```

我们所实现的 Test 测试和 BenchTest 测试原理相同，只是一个是通过 sync.WaitGroup 实现的协程退出机制，另一个是通过 channel 实现的协程退出机制。在这里，我们仅介绍一下基于 Test 的高并发测试。

我们假定有 userNumber 个用户，每个用户进行 requestNumber 次请求，每个请求值为一个随机数 (最大为 requestNumber)。在测试中，我们设定用户数为 100，请求数位 100.

同时请求过多 redis 连接和操作，可能会崩掉，我们在释放每个 user 的请求（即开启每个用户用于请求的协程时）时，先 sleep 一秒。并且在每个用户进行请求的协程中，我们也 sleep 一秒，以观看到令牌桶中令牌的数量并让令牌桶及时补充令牌，不至于瞬间用光令牌.

```go
func TestLimiterAgent_HandleRequest(t *testing.T) {
    userNumber := 100
    requestNumber := 100
    testN(t, userNumber, requestNumber)
}

func testN(t *testing.T, userNumber int, requestNumber int) {
    w := GoN(userNumber, func(i int, user string) {
        for j := 0; j < requestNumber; j++ {
            time.Sleep(1000 * time.Millisecond)
            result, err := limiter.GLimiterAgent().HandleRequest(user,
                int64(random.RandInt(0, requestNumber)))
            if err != nil {
                t.Log(i, result, err)
            }
        }
    })
    w()
}

// GoN 同时启动多个协程，返回等待函数
func GoN(n int, fn func(int, string)) func() {
    var wg sync.WaitGroup
    for i := 0; i < n; i++ {
        wg.Add(1)
        go func(i int) {
            user := "user" + strconv.Itoa(i)
            time.Sleep(1000 * time.Millisecond)
            defer func() {
```

```
                if err := recover(); err != nil {
                    fmt.Println(fmt.Sprintf("panic %s\n", err))
                    fmt.Println(fmt.Sprint(string(debug.Stack())))
                }
            }()
            fn(i, user)
            wg.Done()
        }(i)
    }
    return wg.Wait
}
```

日志对时间，请求，用户本地限流器以及中心限流器的详细信息进行了实时显示，用户及时处理请求的日志样例如下：

```
2022/06/09 21:08:45.517213 User Handle: user1 successly handle 1 requests
                           Details:
                               Current Requests: 1
                               User Info: username: user1, resservePermits: 1.000000, currPermits: 500.000000, Max: 500, Rate: 50
                               Bucket Info: Residue: 100000, Max: 100000, Cycle: 500, Batch: 5000
```

用户没办法处理，中心限流器替它处理的日志样例如下：

```
2022/06/09 21:08:46.511489 center Handle: user99 get help from centerBucket to handle 99 resuests!
                           Details:
                               Current Requests: 99
                               User Info: username: user99, resservePermits: 50.000000, currPermits: 59.000000, Max: 500, Rate: 50
                               Bucket Info: Residue: 100000, Max: 100000, Cycle: 500, Batch: 5000
```

在本次测试中，没有发现无法被处理的请求，它的格式与上面两种日志类似，只是第一行的输出信息不同而已。

总结:

若用户本地获取速度 <= 用户本地生成速度，则每次获取都会成功，并且令牌桶是满的。若本地获取速度 > 本地生成速度，则每次请求（视情况请求中心限流器）不一定会成功，而且用户令牌桶中的令牌会逐渐减小。当所有用户发给中心限流器的请求令牌数之和大于中心限流器的令牌生成速度时，中心限流器的令牌数也会减少。当令牌数不足以处理请求时，则达到限流的效果，于此同时桶内的令牌数也在增加，当桶内的令牌数够多时，就又可以处理请求了。