细读golang底层各种锁源码

## 互斥锁:

互斥锁作为是并发操作中多个程序对共享资源的访问控制的必要手段，（竞争）而golang则定义了mutex和rwmutex，互 斥锁和读写互斥锁来实现。

# 我们首先来看一段程序进而理解锁的重要性。

package bank var balance int

func \_deposit(amount int){ balance+=amount

}

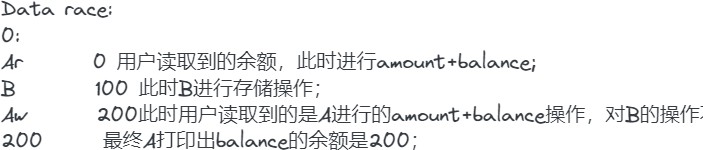
func \_balance() int{ return balance

}

我们首先定义了一个银行，次银行定义了用户余额，以及两个方法\_deposit和\_balance方法，其中\_deposit表示我们用户 向balance进行存款，balance是我们查询用户的余额。其实我们初看此程序是没有什么问题的，我们有钱存进来我们就进 行存储，然后我们还可以查询余额。但是这段代码是有着很大的漏洞的。

*我们首先考虑一种情况*:

我们现在定义两个goroutine：



//A进行存款并读取账户余额go func(){

bank.\_deposit(200) fmt.Println(bank.\_balance(balance))

}()

//B进行存款操作go \_deposit(100)

现在当我们A在存款中我们将执行balance+amount操作，此时是写操作，并且还未进行更新，而此时我们的B也在进行存 款，即发生在写操作之后，更新操作之前，此时我们的balance读到的还是A写入操作的“200”，而我们的B的操作就会直 接被忽略，因为此时我们即便B的操作正常的顺利执行了，也就相当于balance被更新到了100；但是我们的这是发生在A写 之后和A更新之前的，这样当我们读到A更新操作时，我们的balance还是会被赋值为200；即最终我们的balance还是200；这对于我们的客户是极其不公平的。

所以上述有两种方法，一种是设置channel通道信号量（本文不介绍），一种是使用互斥锁的原理，下面我们将详细介绍锁 底层原理，希望有助于各位读者理解。

# mutex结构即相应的方法：

// A Mutex is a mutual exclusion lock.mutex是一个互斥锁

// The zero value for a Mutex is an unlocked mutex.//0值表示mutex当前未被锁住

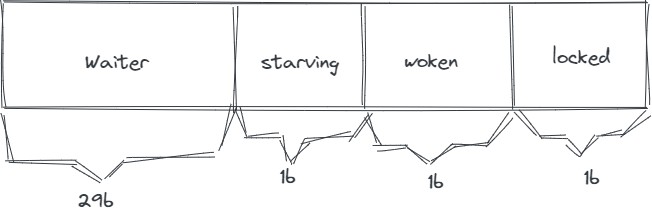
//

// A Mutex must not be copied after first use.mutex锁在首次使用后不得进行复制type Mutex struct {

state int32 sema uint32

}

其中我们的mutex记录了两个变量，其中一个是状态变量state，表明当前是否lock，而sema则是信号量，协程的阻塞等待 该信号量，当一个协程使用完后就进行释放信号量来唤醒那些等待的信号量的协程。

其中state并不只是0,1值，他记录着四个字段(分别是29b，不要看成6)

locked：表示该mutex是否已被锁定，0：没有；1：有 Woken：表示是否有协程已被唤醒；0：表示没有协程被唤醒，1：表示有协程被唤醒，正在加锁过程中 Starving：表示锁是否处于饥饿状态，0：没有饥饿，1：饥饿状态，说明有协程被阻塞 waiter：表明阻塞等待锁的协程个数，协程解锁的时候会根据这个值来判断是否需要释放信号量。

协程之间的竞争即是抢锁过程，看谁可以抢到锁，即把locked复制为1，而其他未抢到的协程就需要进行等待mutex.sema 信号量，一旦持有锁的协程释放锁，等待的协程就会被唤醒。

正常模式：所有被阻塞在等待队列中的goroutine会按照顺序获取锁的使用权，队头的goroutine被唤起时，会去修改state 状态，修改成功表示占用锁资源成功

饥饿模式：上面所说我们队头不会立即得到锁资源，他会不断地进行尝试判断，如果当前有一个新的goroutine也尝试判 断，一般来说，go的调度机制会比较偏向于cpu占有时间短的goroutine先执行，这就有一定的概率使队头的goroutine一 直获取不到锁导致饿死，于是mutex的饥饿模式表示当我们的队头goroutine超过1ms还未获取到锁，我们就将下一次unlock的时候直接给队头，并将新的请求的goroutine放到队尾里面去等待

自旋过程：当我们有一个goroutine要进行加锁操作时，如果发现locked==1表示当前其他goroutine还在使用，而尝试加 锁的goroutine不会立即推入到等待队列中，而是会持续的探测locked位是否变为0。 自旋的优势可以保证协程不必来回的进行切换

条件： 1自旋次数小于4次

2多核处理器(在一枚处理器（processor）中集成两个或多个完整的计算引擎，即我们执行20个任务时，但我们只有单核时 我们要求的性能就必须20；而当我们有2核时，我们就只需要单核性能达到10以上就可以了，以此类推 ) 3gomaxprocs>1(多核并发执行)

4协程调度机制中的可运行队列必须为空，否则会延迟协程调度mutex方法:

// Lock locks m.

// If the lock is already in use, the calling goroutine

// blocks until the mutex is available.表示一旦锁被使用，其他的goroutine就需要等待锁可以再次被释放可用为止func (m \*Mutex) Lock() {

// Fast path: grab unlocked mutex.很快的一种方式，即我们直接就获取到了锁

//CAS模式，与乐观锁一样，我们首先会拿出一个状态值，当我们对他进行一个模拟操作后，我们又会重新拿出他的值来与之间的值进行比较

if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) { if race.Enabled {

race.Acquire(unsafe.Pointer(m))

}

return

}

// Slow path (outlined so that the fast path can be inlined)我们此时就需要进行一些缓慢的上锁，即协程竞争等m.lockSlow()

}

func (m \*Mutex) lockSlow() { var waitStartTime int64

starving := false//是否处于饥饿awoke := false//是否被唤醒 iter := 0//自旋次数

old := m.state //当前锁的状态for {

// Don't spin in starvation mode, ownership is handed off to waiters

// so we won't be able to acquire the mutex anyway.

//当前mutex不处于饥饿模式，锁并且还未被释放，此时我们尝试自旋

if old&(mutexLocked|mutexStarving) == mutexLocked && runtime\_canSpin(iter) {

// Active spinning makes sense.

// Try to set mutexWoken flag to inform Unlock

// to not wake other blocked goroutines.

//未设置过唤醒标识并执行自旋操作

if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 && atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) { awoke = true

}

runtime\_doSpin() iter++

old = m.state continue

}

new := old

// Don't try to acquire starving mutex, new arriving goroutines must queue.

//判断是否是饥饿状态，如果不是则尝试上锁，是的话，不会上锁，上文讲过我们直接将锁分配给队头goroutine，并将新来的添加到队列 if old&mutexStarving == 0 {

new |= mutexLocked

}

if old&(mutexLocked|mutexStarving) != 0 { new += 1 << mutexWaiterShift

}

// The current goroutine switches mutex to starvation mode.

// But if the mutex is currently unlocked, don't do the switch.

// Unlock expects that starving mutex has waiters, which will not

// be true in this case.

//如果之间是饥饿模式，则再次设置为饥饿模式 if starving && old&mutexLocked != 0 {

new |= mutexStarving

}

if awoke {

// The goroutine has been woken from sleep,

// so we need to reset the flag in either case.

//状态不符合预期，则throw抛出错误if new&mutexWoken == 0 {

throw("sync: inconsistent mutex state")

}

//新的状态值需要清除唤醒标识，因为当前的goroutine会再次上锁或sleep new &^= mutexWoken

}

if atomic.CompareAndSwapInt32(&m.state, old, new) {

//CAS尝试修改状态，修改成功表示获取到了锁资源

if old&(mutexLocked|mutexStarving) == 0 { break // locked the mutex with CAS

}

// If we were already waiting before, queue at the front of the queue.

//根据等待时间计算queuelifo

queueLifo := waitStartTime != 0 if waitStartTime == 0 {

waitStartTime = runtime\_nanotime()

}

runtime\_SemacquireMutex(&m.sema, queueLifo, 1)

starving = starving || runtime\_nanotime()-waitStartTime > starvationThresholdNs old = m.state

if old&mutexStarving != 0 {

// If this goroutine was woken and mutex is in starvation mode,

// ownership was handed off to us but mutex is in somewhat

// inconsistent state: mutexLocked is not set and we are still

// accounted as waiter. Fix that.

if old&(mutexLocked|mutexWoken) != 0 || old>>mutexWaiterShift == 0 { throw("sync: inconsistent mutex state")

}

delta := int32(mutexLocked - 1<<mutexWaiterShift)

//此时不是饥饿模式或者下次没有要唤起队列的goroutine了 if !starving || old>>mutexWaiterShift == 1 {

// Exit starvation mode.

// Critical to do it here and consider wait time.

// Starvation mode is so inefficient, that two goroutines

// can go lock-step infinitely once they switch mutex

// to starvation mode. delta -= mutexStarving

}

atomic.AddInt32(&m.state, delta) break

}

//不再是饥饿模式了，需要重新清除自选次数。awoke = true

iter = 0

} else {

old = m.state

}

}

if race.Enabled { race.Acquire(unsafe.Pointer(m))

}

}

// Unlock unlocks m.

// It is a run-time error if m is not locked on entry to Unlock.

//

// A locked Mutex is not associated with a particular goroutine.

// It is allowed for one goroutine to lock a Mutex and then

// arrange for another goroutine to unlock it. func (m \*Mutex) Unlock() {

if race.Enabled {

\_ = m.state race.Release(unsafe.Pointer(m))

}

// Fast path: drop lock bit.

//快速上锁

new := atomic.AddInt32(&m.state, -mutexLocked) if new != 0 {

// Outlined slow path to allow inlining the fast path.

// To hide unlockSlow during tracing we skip one extra frame when tracing GoUnblock.进行操作较多的解锁步骤m.unlockSlow(new)

}

}

func (m \*Mutex) unlockSlow(new int32) {

//不能释放未被上锁的锁

if (new+mutexLocked)&mutexLocked == 0 { throw("sync: unlock of unlocked mutex")

}

if new&mutexStarving == 0 { old := new

for {

// If there are no waiters or a goroutine has already

// been woken or grabbed the lock, no need to wake anyone.

// In starvation mode ownership is directly handed off from unlocking

// goroutine to the next waiter. We are not part of this chain,

// since we did not observe mutexStarving when we unlocked the mutex above.

// So get off the way.

//等待队列为空即没有需要唤起的goroutine

if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken|mutexStarving) != 0 { return

}

//唤起等待队列并数量-1

// Grab the right to wake someone.

new = (old - 1<<mutexWaiterShift) | mutexWoken

if atomic.CompareAndSwapInt32(&m.state, old, new) { runtime\_Semrelease(&m.sema, false, 1)

return

}

} else {

}

old = m.state

// Starving mode: handoff mutex ownership to the next waiter, and yield

// our time slice so that the next waiter can start to run immediately.

// Note: mutexLocked is not set, the waiter will set it after wakeup.

// But mutex is still considered locked if mutexStarving is set,

// so new coming goroutines won't acquire it.

//饥饿模式下锁直接给等待队列的头runtime\_Semrelease(&m.sema, true, 1)

}

}

**rwmutex：**

在日常生活中我们读的次数远比写的次数大得多，如果我们单单使用mutex的话，这样会导致我们有些人想读并不向写的时 候也会被阻塞，效率太过低下，所以我们设计出rwmutex，意思是可以多个goroutine同时进行读操作，但是只有一个写操 作，大大节省了时间效率。

type RWMutex struct {

w Mutex // held if there are pending writers

writerSem uint32 // semaphore for writers to wait for completing readers readerSem uint32 // semaphore for readers to wait for completing writers readerCount int32 // number of pending readers

readerWait int32 // number of departing readers

}

w:写互斥 writesema:用于写来等待读 readersem：用于读来等待写 readercount：当前读的goroutine数量 readerwait： 当前被阻塞的准备读的goroutine数量

// RLock locks rw for reading.

//

// It should not be used for recursive read locking; a blocked Lock

// call excludes new readers from acquiring the lock. See the

// documentation on the RWMutex type.

//读锁，每当我们请求读锁的时候我们首先判断当前是否有写者在进行写操作，(这里即是判断readercount是否小于0，小于0便表示有人正在进 func (rw \*RWMutex) RLock() {

if race.Enabled {

\_ = rw.w.state race.Disable()

}

if atomic.AddInt32(&rw.readerCount, 1) < 0 {

// A writer is pending, wait for it. runtime\_SemacquireMutex(&rw.readerSem, false, 0)

}

if race.Enabled { race.Enable()

race.Acquire(unsafe.Pointer(&rw.readerSem))

}

}

// RUnlock undoes a single RLock call;

// it does not affect other simultaneous readers.

// It is a run-time error if rw is not locked for reading

// on entry to RUnlock.

//释放读锁，将加读锁的goroutine数量-1，如果我们没有加写锁的goroutine，rc肯定大于0；如果有goroutine请求加写锁，这时候需要看看func (rw \*RWMutex) RUnlock() {

if race.Enabled {

\_ = rw.w.state race.ReleaseMerge(unsafe.Pointer(&rw.writerSem)) race.Disable()

}

if r := atomic.AddInt32(&rw.readerCount, -1); r < 0 {

// Outlined slow-path to allow the fast-path to be inlined rw.rUnlockSlow(r)

}

if race.Enabled { race.Enable()

}

}

func (rw \*RWMutex) rUnlockSlow(r int32) {

if r+1 == 0 || r+1 == -rwmutexMaxReaders { race.Enable()

throw("sync: RUnlock of unlocked RWMutex")

}

//释放了未被加锁的协程，panic

// A writer is pending.

if atomic.AddInt32(&rw.readerWait, -1) == 0 {

// The last reader unblocks the writer. runtime\_Semrelease(&rw.writerSem, false, 1)

}

}

/ Lock locks rw for writing.

// If the lock is already locked for reading or writing,

// Lock blocks until the lock is available.

//当前有写操作时，锁住rw，当我们这个锁已经被锁住了，我们还是需要等待锁再次被释放。func (rw \*RWMutex) Lock() {

if race.Enabled {

\_ = rw.w.state race.Disable()

}

// First, resolve competition with other writers.和其他写者进行竞争锁操作rw.w.Lock()

// Announce to readers there is a pending writer.当我们写者表示抢到锁了，我们就要通知其他读锁当前有个写者了

//我们首先将readercount变为负数，但是我们把新值+rwmax交给了r，我们即判断r是否为负数，如果不是并且我们将r赋值给readerwait( r := atomic.AddInt32(&rw.readerCount, -rwmutexMaxReaders) + rwmutexMaxReaders

// Wait for active readers.

//

if r != 0 && atomic.AddInt32(&rw.readerWait, r) != 0 { runtime\_SemacquireMutex(&rw.writerSem, false, 0)

}

//当我们的加写锁之间的读锁goroutine都释放了，我们重新唤醒写锁 if race.Enabled {

race.Enable() race.Acquire(unsafe.Pointer(&rw.readerSem)) race.Acquire(unsafe.Pointer(&rw.writerSem))

}

}

// Unlock unlocks rw for writing. It is a run-time error if rw is

// not locked for writing on entry to Unlock.

//

// As with Mutexes, a locked RWMutex is not associated with a particular

// goroutine. One goroutine may RLock (Lock) a RWMutex and then

// arrange for another goroutine to RUnlock (Unlock) it.

//即释放了未被上锁的锁是不对的，一个rw锁是不会和任何一个goroutine产生联系的，但是当我们其中有一个读goroutine去rlock了rw锁的func (rw \*RWMutex) Unlock() {

if race.Enabled {

\_ = rw.w.state race.Release(unsafe.Pointer(&rw.readerSem)) race.Disable()

}

// Announce to readers there is no active writer.表示我们当前没有写锁进行申请，正如上面lock的时候我们将readercount以r := atomic.AddInt32(&rw.readerCount, rwmutexMaxReaders)

if r >= rwmutexMaxReaders { race.Enable()

throw("sync: Unlock of unlocked RWMutex")

}

// Unblock blocked readers, if any. for i := 0; i < int(r); i++ {

runtime\_Semrelease(&rw.readerSem, false, 0)//释放读信号量

}

// Allow other writers to proceed. rw.w.Unlock()// 释放了锁了,允许其他的写者可以进行了if race.Enabled {

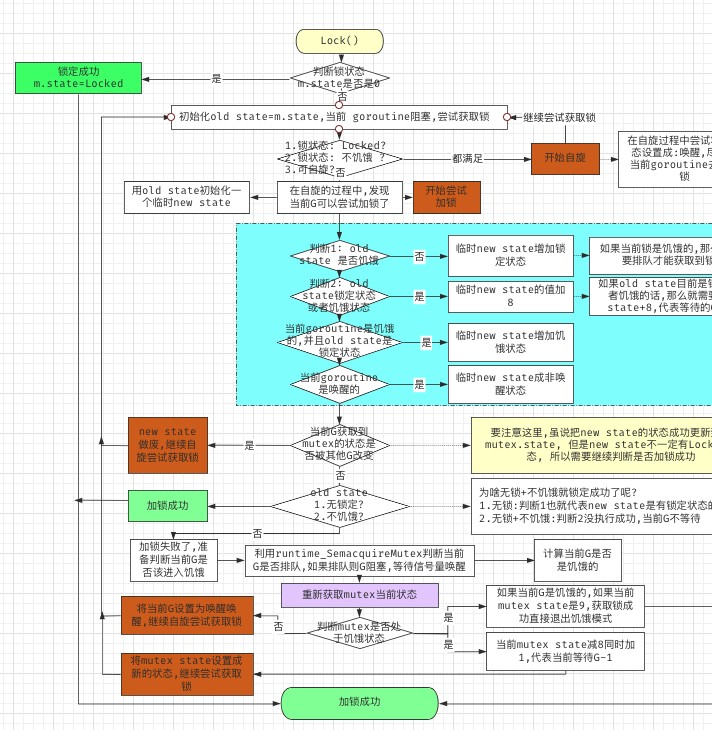
race.Enable()

}

}

**总结**：

所谓互斥锁，读写互斥锁，其实就是一把锁，对于互斥锁，我们只有一个人有钥匙，当我们有一个人打开锁后，进入房间后 再出来后，我们需要将锁交给下一个人，下一个人才能进去，如此循环。而对于读写互斥锁，我们就不需要那么麻烦，我们 如果只是进入方阿金看一下并不做实质性的改变，我们是可以“允许多个人进入的”这里的也不是真正意义上的并行（还是 并发）他还是一个一个进去，只是时间效率会快很多。最后golang的读写互斥锁还是和操作系统的读写锁有一些区别的， 例如当我们有个进程要读些操作时，我们会先调用rlock，此时我们再去调用lock，这时候我们的lock会等rlock，rlock会等lock就会形成死锁



## 参考文章

<https://blog.csdn.net/baolingye/article/details/111357407><https://zhuanlan.zhihu.com/p/393257803> <https://zhuanlan.zhihu.com/p/397460154>