Atomic & Lock 的運作

Vic

講者介紹

- Medium (@vicxu) 寫廢文
- Golang 2+ Yr
- 在 BitoPro 打工

一個簡單的轉帳程式

You, seconds ago | 2 authors (You and others)

```go
type User struct {
    ID      uint64
    Balance uint64
}

func transfer(from *User, to *User, amount uint64) {
    if from.Balance >= amount {
        from.Balance -= amount
        to.Balance += amount
    }
}
```

把簡單的轉帳程式，變得一丟丟複雜

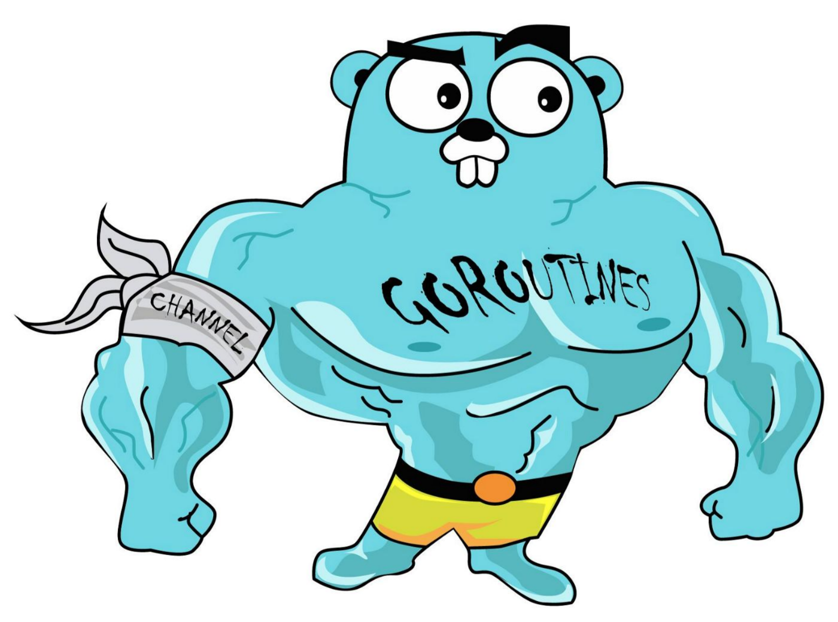
```go
func main() {
    userA := User{
        ID: 1, Balance: 10e10,
    }
    userB := User{
        ID: 2, Balance: 10e10,
    }
    wg := sync.WaitGroup{}
    wg.Add(2)
    go func() {
        defer wg.Done()
        for _ = range [10e10]uint64{} {
            transfer(&userA, &userB, 1)
        }
    }()

    go func() {
        defer wg.Done()
        for _ = range [10e10]uint64{} {
            transfer(&userA, &userB, 1)
        }
    }()
    wg.Wait()
}
```

You, seconds ago | 2 authors (You and others)

```go
type User struct {
    ID      uint64
    Balance uint64
    Lock    sync.Mutex
}

func transfer(from *User, to *User, amount uint64) {
    from.Lock.Lock()
    to.Lock.Lock()
    defer from.Lock.Unlock()
    defer to.Lock.Unlock()

    if from.Balance >= amount {
        from.Balance -= amount
        to.Balance += amount
    }
}
```

Lock 無處不在

- 大流量的情境
- 高併發的 Solution
- Race Condition 的問題
- Lock Performance?



盤古開天，最初的 Lock

- Semaphore Variable

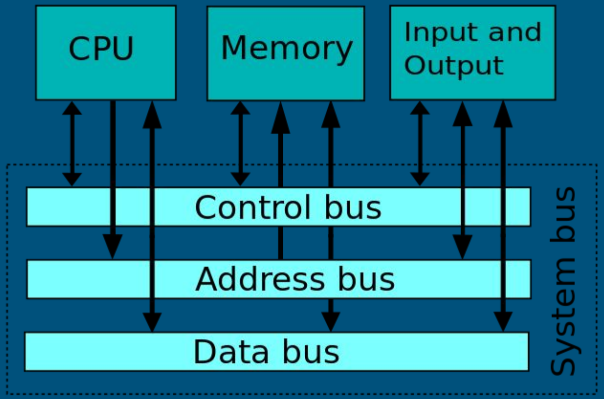
  In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system.

```go
//互斥锁结构
type Mutex struct {
    key  int32
    sema uint32
}

//请求锁
func (m *Mutex) Lock() {
    if atomic.AddInt32(&m.key, 1) == 1 {    //标识加1，如果等于1，成功获取到锁
        return
    }
    runtime.Semacquire(&m.sema)     //否则阻塞等待
}


//释放锁
func (m *Mutex) Unlock() {
    switch v := atomic.AddInt32(&m.key, -1); {  //标识减1
    case v == 0:    //如果等于0，则没有等待者
        return
    case v == -1:   //如果等于-1，这种是异常情况，或者超过了最大可等待goroutine的数量
        panic("sync: unlock of unlocked mutex")
    }
    runtime.Semrelease(&m.sema) //唤醒其他阻塞的goroutine
}
```

Semaphore Variable & Atomic Operation

- 如何避免 Semaphore Variable Race Condition？
- Atomic Operation 使用 CPU-Level (hardware) Lock
- 使用 Assembly Lang 的 LOCK Prefix

```
TEXT runtime∕internal∕atomic·Xadd64(SB), NOSPL
        MOVQ    ptr+0(FP), BX // 第一个参数保存到
        MOVQ    delta+8(FP), AX // 第二个参数保存
        MOVQ    AX, CX   // 将第二个参数临时存到CX
        LOCK                     // LOCK指令进行
        XADDQ   AX, 0(BX) // xaddq指令，实现寄存
        // 并将这两个值的和存在BX指向的内存中，此时A
        ADDQ    CX, AX // 此时AX寄存器的值是Add操
        MOVQ    AX, ret+16(FP) # 返回值
        RET
```



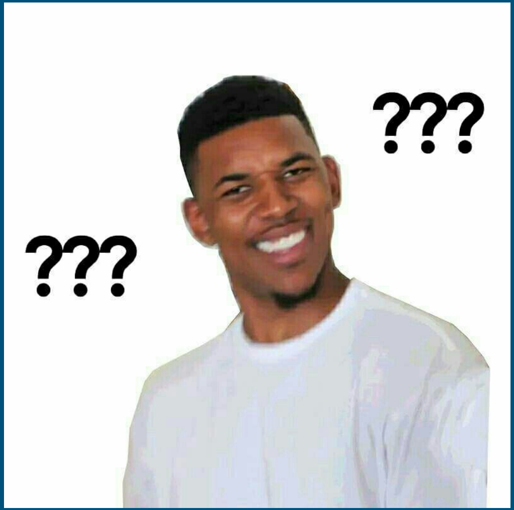
用 Atomic Operation 實作 Lock 功能

- Self-Spinning Lock
- CPU consumption，會搶走其他 thread 的 CPU

```go
func addBalance(u *User, semaphoreVar *int32, delta uint64) {
    for {
        if atomic.CompareAndSwapInt32(semaphoreVar, 0, 1) {
            u.Balance += delta
            atomic.StoreInt32(semaphoreVar, 0)
            return
        }
    }
}
```

回來看看 Mutex.Lock

- runtime.Semacuire？




```go
//互斥锁结构
type Mutex struct {
    key  int32
    sema uint32
}

//请求锁
func (m *Mutex) Lock() {
    if atomic.AddInt32(&m.key, 1) == 1 {    //标识加1，如果等于1，成功获取到锁
        return
    }
    runtime.Semacquire(&m.sema)     //否则阻塞等待
}


//释放锁
func (m *Mutex) Unlock() {
    switch v := atomic.AddInt32(&m.key, -1); {  //标识减1
    case v == 0:     //如果等于0，则没有等待者
        return
    case v == -1:    //如果等于-1，这种是异常情况，或者超过了最大可等待goroutine的数量
        panic("sync: unlock of unlocked mutex")
    }
    runtime.Semrelease(&m.sema) //唤醒其他阻塞的goroutine
}
```

Kernel Futex Syscall

- FUTEX_WAIT & FUTEX_WAKE
- Wait Queue

DESCRIPTION top

The **futex**() system call provides a method for waiting until a certain condition becomes true. It is typically used as a blocking construct in the context of shared-memory synchronization. When using futexes, the majority of the synchronization operations are performed in user space. A user-space program employs the **futex**() system call only when it is likely that the program has to block for a longer time until the condition becomes true. Other **futex**() operations can be used to wake any processes or threads waiting for a particular condition.

runtime Sema 治百病？

```go
func runtime_Semacquire(s *uint32)
func runtime_SemacquireMutex(s *uint32, lifo bool, skipframes int)
func runtime_Semrelease(s *uint32, handoff bool, skipframes int)
```

- 相較 Futex 效能更好，使用 go rutime GPM Scheduler 模型
- lock , release lock 間距時間極短時，goroutine 休眠&喚醒操作會是效能問題

Lock 進化2.0:Spinning 結合 runtime Sema

- 給初次進來 goroutine spinning 的機會
- spinning 時機，如果 lock state 有改變，就再嘗試獲取 lock

```go
func (m *Mutex) Lock() {
    // Fast path: 幸运case，能够直接获取到锁
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    awoke := false
    for {
        old := m.state
        new := old | mutexLocked      //新状态加锁
        if old&mutexLocked != 0 {
            new = old + 1<<mutexWaiterShift     //等待者数量加一
        }
        if awoke {
            //goroutine是被唤醒的，新状态清除唤醒标记
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) { //设置新状态
            if old&mutexLocked == 0 {   //锁原状态未加锁
                break
            }
            runtime.Semacquire(&m.sema)  //请求信号量
            awoke = true //设置唤醒标记
        }
    }
}
```

Lock 進化2.0：Spinning 結合 runtime Sema

	T1	T2	T3	T4	T5	T6
Thread 1	Lock Succ			Relase Lock		
Thread 2	Lock Failed	state Locked	Enter Wait Queue	state Unlock	CAP Failed	Lock Succ
Thread 3			Lock Failed	state Unlock	Lock Succ	Relase Lock

```go
func (m *Mutex) Lock() {
    // Fast path: 幸运case，能够直接获取到锁
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }

    awoke := false
    for {
        old := m.state
        new := old | mutexLocked    //新状态加锁
        if old&mutexLocked != 0 {
            new = old + 1<<mutexWaiterShift    //等待者数量加一
        }
        if awoke {
            //goroutine是被唤醒的，新状态清除唤醒标记
            new &^= mutexWoken
        }
        if atomic.CompareAndSwapInt32(&m.state, old, new) { //设置新状态
            if old&mutexLocked == 0 {   //锁原状态未加锁
                break
            }
            runtime.Semacquire(&m.sema)  //请求信号量
            awoke = true //设置唤醒标记
        }
    }
}
```

Lock 進化2.0：Spinning 結合 runtime Sema

- 當前鎖狀態被 locked again or 有 goroutine 喚醒中直接 return
- 減少一個等待數量
- 將 lock state 改成喚醒，並喚醒 wait queue 第一個 goroutine
- Spinning gorouinte 與 Wait queue goroutine 會競爭鎖

```go
func (m *Mutex) Unlock() {
    // Fast path: drop lock bit.
    new := atomic.AddInt32(&m.state, -mutexLocked)  //去掉锁状态
    if (new+mutexLocked)&mutexLocked == 0 {     //未被锁定的mutex释放锁会panic
        panic("sync: unlock of unlocked mutex")
    }

    old := new
    for {
        //锁上没有goroutine等待或者有被唤醒的goroutine,或者又被别的goroutine加了锁,那么不
        if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken) != 0 {
            return
        }
        //将mutexWaiterShift数量减1并设置mutexWoken为true
        new = (old - 1<<mutexWaiterShift) | mutexWoken
        if atomic.CompareAndSwapInt32(&m.state, old, new) {     //CAS设置成功，唤醒
            runtime.Semrelease(&m.sema)
            return
        }
        old = m.state   //记录当前mutex的状态，继续循环
    }
}
```

Lock 進化2.1:避免 Spinning 搶得太兇

- 降低 goroutine 進入沈睡狀態的機會

```
awoke := false
iter := 0
for {   // 不管是新来的请求锁的goroutine, 还是被唤醒的goroutine，都不断尝试请求锁
    old := m.state
    new := old | mutexLocked     //新状态加锁
    if old&mutexLocked != 0 {    // 锁还没被释放
        if runtime_canSpin(iter) {  // 还可以自旋
            if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
                atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
                awoke = true
            }
            runtime_doSpin()
            iter++
            continue      //自旋，再次尝试获取锁
        }
        new = old + 1<<mutexWaiterShift      //等待者数量加一
    }
```

```
    if awoke {  //唤醒状态，去掉标记
        new &^= mutexWoken
    }
    if atomic.CompareAndSwapInt32(&m.state, old, new) { //设置新
        if old&mutexLocked == 0 {   //锁原状态未加锁
            break
        }
        runtime_Semacquire(&m.sema) //请求信号量
        awoke = true     //设置信号量
        iter = 0     //重新设置自旋计数器
    }
}
```

Lock 進化3.0：Lock 究極進化

- lockSlow 封裝
- 新增 starving 狀態
- 如果 lock 有 starving 狀態，新進 goroutine 就不給 spinning 機會

```
func (m *Mutex) Lock() {
	// Fast path: 顺利的获取到锁
	if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
		if race.Enabled {
			race.Acquire(unsafe.Pointer(m))
		}
		return
	}
	// Slow path (缓慢之路，通过自旋、竞争或者饥饿状态下的锁竞争)
	m.lockSlow()
}
```

Lock 進化3.0:Lock 究極進化

- waitStartTime 用來判斷是否要進入飢餓狀態
- spinning 條件多新增了是否非飢餓狀態

```go
func (m *Mutex) lockSlow() {
	var waitStartTime int64
	starving := false	//标识当前goroutine是否饥饿
	awoke := false  //唤醒标记
	iter := 0   //自旋次数
	old := m.state  //当前的锁状态
```

```go
for {
	//锁是非饥饿状态，并且未释放，尝试自旋
	if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
		// 主动自旋的场景
		// 尝试设置 mutexWoken 标志以通知 Unlock 不唤醒其他阻塞的 goroutine
		if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0 &&
			atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken) {
			awoke = true
		}
		runtime_doSpin()    //自旋
		iter++
		old = m.state
		continue
	}
```

Lock 進化3.0：Lock 究極進化

- 鎖當前狀態是非飢餓，才能嘗試獲取鎖
- 如果當前 goroutine 已經飢餓，嘗試將鎖加上飢餓狀態

```go
new := old
// 不要尝试获取饥饿的互斥锁，新到达的 goroutine 必须排队
if old&mutexStarving == 0 {
        new |= mutexLocked  //非饥饿状态，加锁
}
if old&(mutexLocked|mutexStarving) != 0 {   //饥饿状态，或者锁被抢占，等待者 + 1
        new += 1 << mutexWaiterShift
}
// 当前 goroutine 将互斥锁切换到饥饿模式。
if starving && old&mutexLocked != 0 {
        new |= mutexStarving
}
if awoke {
        //清除awoke标识
        new &^= mutexWoken
}
```

Lock 進化3.0 : Lock 究極進化

- 如果之前進入過 wait queue 將該 goroutine 放到 wait queue head
- 被喚醒時，計算是否要觸發飢餓模式
- starvationThresholdNs = 1ms
- 被喚醒時，如果 lock state 是飢餓，直接獲取鎖

```
if atomic.CompareAndSwapInt32(&m.state, old, new) {
        if old&(mutexLocked|mutexStarving) == 0 {
                break // 上锁成功
        }
        // 第一次等待，添加到信号量队列的队首
        queueLifo := waitStartTime != 0
        if waitStartTime == 0 {
                waitStartTime = runtime_nanotime()
        }
        runtime_SemacquireMutex(&m.sema, queueLifo, 1)
/设置饥饿标记
        starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
        old = m.state
        if old&mutexStarving != 0 {
                //加锁并将waiter数量减1
                delta := int32(mutexLocked - 1<<mutexWaiterShift)
                if !starving || old>>mutexWaiterShift == 1 {
                        //非饥饿状态的goroutine,最后一个waiter已经不饥饿了，清除标记
                        delta -= mutexStarving
                }
                atomic.AddInt32(&m.state, delta)
                break
        }
        awoke = true
        iter = 0
```

Lock 進化3.0：Lock 究極進化

	T1	T2	T3	T4	T5	T6	T7
Thread1	Lock Succ		Release				
Thread2	Lock Failed	Enter Wait Queue	CAP Failed	State Starving	Enter Wait Queue	Awoke	Lock Succ
Thread3			Lock Succ			Release	
Thread4				Lock Failed	Enter Wait Queue	Sleep	
Thread5				Lock Failed	Enter Wait Queue	Sleep	

Lock 進化3.0：Lock 究極進化

- 飢餓模式下，直接將 goroutine 丟進 waitqueue
- 非飢餓模式，才會檢查是否有 spinning goroutine 在嘗試拿鎖

```
(m *Mutex) unlockSlow(new int32) {
/无锁的mutex释放锁会panic
    if (new+mutexLocked)&mutexLocked == 0 {
            throw("sync: unlock of unlocked mutex")
    }
    if new&mutexStarving == 0 {
            old := new
            for {
                    //锁上没有goroutine等待或者有被唤醒的goroutine改变了锁的状态，直接return即可
                    if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken|mutexStarving) != 0 {
                            return
                    }
                    // 减少等待者并设置Mutex唤醒标记，CAS释放锁
                    new = (old - 1<<mutexWaiterShift) | mutexWoken
                    if atomic.CompareAndSwapInt32(&m.state, old, new) {
                            runtime_Semrelease(&m.sema, false, 1)    //锁在正常模式下从sema优先队列尾部唤醒新
                            return
                    }
                    old = m.state
            }
    } else {
            //饥饿模式下，从优先队列的头部唤醒等待的goroutine，Lock方法会直接将锁给它
            runtime_Semrelease(&m.sema, true, 1)
    }
```

Mutex.Lock 常見的坑

- 所有 goroutine 共用 State，A goroutine Lock，B goroutine Unlock 會 release A goroutine 的鎖
- 連續執行兩次 Lock，會造成死鎖
- mutex.Lock struct 不能被複製，被複製的 lock 狀態會異常，不會重置

Lock 最大的敵人 : Deadlock

```go
func main() {
    userA := User{
        ID: 1, Balance: 10e10,
    }
    userB := User{
        ID: 2, Balance: 10e10,
    }
    wg := sync.WaitGroup{}
    wg.Add(2)
    go func() {
        defer wg.Done()
        for _ = range [10e10]uint64{} {
            transfer(&userA, &userB, 1)
        }
    }()

    go func() {
        defer wg.Done()
        for _ = range [10e10]uint64{} {
            transfer(&userB, &userA, 1)
        }
    }()
    wg.Wait()
}
```

```go
func transfer(from *User, to *User, amount uint64) {
    from.Lock.Lock()
    to.Lock.Lock()
    defer from.Lock.Unlock()
    defer to.Lock.Unlock()

    if from.Balance >= amount {
        from.Balance -= amount
        to.Balance += amount
    }
}
```

Lock 最大的敵人：Deadlock

- Deadlock 的成因，Lock 順序不一至
- 透過 User.ID 確保 Lock 的順序

```go
type User struct {
	ID      uint64
	Balance uint64
	Lock    *sync.Mutex
}

func transfer(from *User, to *User, amount uint64) {
	firstLock := from.Lock
	secondLock := to.Lock
	if from.ID > to.ID {
		secondLock = from.Lock
		firstLock = to.Lock
	}

	firstLock.Lock()
	secondLock.Lock()
	defer firstLock.Unlock()
	defer secondLock.Unlock()

	if from.Balance >= amount {
		from.Balance -= amount
		to.Balance += amount
	}
}
```

不要用 channel 取代 sync.Mutex

```
$ go test -bench=. -benchtime=5s
goos: windows
goarch: amd64
pkg: sync-benchmark
cpu: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

Benchmark_NoSync-8       1000000000              1.695 ns/op
Benchmark_Atomic-8       1000000000              5.232 ns/op
Benchmark_Mutex-8        558550783               10.86 ns/op
Benchmark_Channel-8      168589561               35.56 ns/op

PASS
ok      sync-benchmark  25.421s
```

招募 / Q & A

幣託徵才資訊

小弟 Medium

References

- https://levelup.gitconnected.com/go-a-benchmark-to-compare-synchronization-techniques-ed73e118ec35
- https://juejin.cn/post/6990181431574003726
- https://juejin.cn/post/6948773296045621255
- https://juejin.cn/post/6955370086450331684
- https://juejin.cn/post/6955370086450331684
- https://www.youtube.com/watch?v=tjpncm3xTTc&t=1568s