Go Mutex 实现与演进

@huwenhao

为什么需要锁

锁的概念诞生于并发场景下的竞争问题，用于保证并发环境下对共享资源访问的互斥，是限制共享资源访问的同步机制。

并发与并行

并行指的是多个线程能在同一时刻同时运行，真正的并行只在多核场景下能够实现，在单核场景下，由于操作系统的时间片调度机制存在，可以在宏观上实现一种“伪”并行（并发）。

图1 并发与并行

为什么需要锁

线程的并发问题

但无论是单核还是多核，程序执行起来，并不像我们感觉到的那样连续，而是会因为硬件中断、IO等原因，而表现为断断续续的执行。所以当我们有多个线程并发执行，并伴随有对共享数据的操作，就可能会发生一些预料之外的问题。

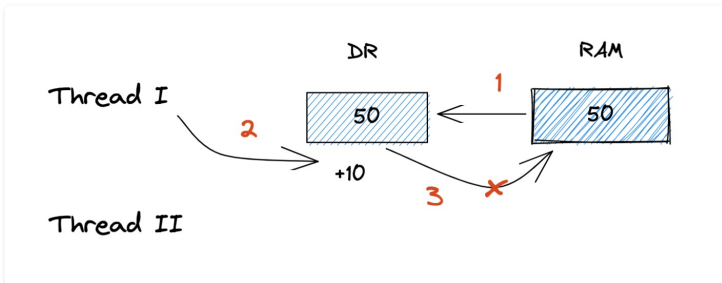

图2 并发问题

什么是锁

锁是一种并发编程中的同步原语，通过线程主动放弃运行机会的方式，来协调多线程对共享资源的竞争。

1. 单核场景:
   锁就是一个变量，通过硬件提供的原子操作（如：CAS等）实现，保证了同一时刻只有一个线程拿到锁进入临界区;

2. 多核场景:
   硬件原子指令是由多个微指令组成，原子性相对于单核而言的，多核的多个线程竞争进入临界区权限需要锁缓存行或总线;

github.com/go/src/runtime/internal/atomic/atomic_amd64.s

```
// bool Cas(int32 *val, int32 old, int32 new)
// Atomically:
//    if(*val == old){
//        *val = new;
//        return 1;
//    } else
//        return 0;
TEXT ·Cas(SB),NOSPLIT,$0-17
  MOVQ    ptr+0(FP), BX
  MOVL    old+8(FP), AX
  MOVL    new+12(FP), CX
  LOCK
  CMPXCHGL    CX, 0(BX)
  SETEQ   ret+16(FP)
  RET
```

*在Go语言CSA的实现中，执行CMPXCHGL指令之前会执行LOCK指令

Go Mutex

Mutex一词取自于Mutual exclusion的缩写，也称作互斥锁。Go语言在Mutex的实现中迭代了多个版本:

- V1.0 初版实现
- V2.0 引入非公平锁(go 1.0)
- V3.0 引入自旋(go 1.5)
- V4.0 解决饥饿(go 1.13)
- V4.1 增加TryLock方法(go 1.18)

*以上版本的编号是为了方便叙述和区分临时定义的，官方并没有这样的说法

V1.0 初版实现

最初版的Mutex实现比较简单，只有 key 和 sema 两个字段：

1. key 用来标识锁是否被持有，也用来标识当前队列有多少个等待的goroutine
2. sema 为信号量变量，用来阻塞和唤醒goroutine。

github.com/go/src/sync/mutex.go

```
type Mutex struct {
  key  int32
  sema uint32
}

func (m *Mutex) Lock() {
  if atomic.AddInt32(&m.key, 1) == 1 {
      return
  }
  runtime.Semacquire(&m.sema)
}

func (m *Mutex) Unlock() {
  switch v := atomic.AddInt32(&m.key, -1); {
  case v == 0:
      return
  case v == -1:
      panic("sync: unlock of unlocked mutex")
  }
  runtime.Semrelease(&m.sema)
}
```

V1.0 初版实现

初版的 Mutex 实现有一个问题：

请求锁的 goroutine 会排队等待获取互斥锁。虽然这貌似很公平，但是从性能上来看，却不是最优的。

因为如果我们能够把锁交给正在占用 CPU 时间片的 goroutine 的话，那就不需要做上下文的切换，在高并发的情况下，会有更好的性能。

github.com/go/src/sync/mutex.go

```
type Mutex struct {
  key  int32
  sema uint32
}

func (m *Mutex) Lock() {
  if atomic.AddInt32(&m.key, 1) == 1 {
      return
  }
  runtime.Semacquire(&m.sema)
}

func (m *Mutex) Unlock() {
  switch v := atomic.AddInt32(&m.key, -1); {
  case v == 0:
      return
  case v == -1:
      panic("sync: unlock of unlocked mutex")
  }
  runtime.Semrelease(&m.sema)
}
```

V2.0 引入非公平锁

在这一版本中，Go开发者对于Mutex的实现做了一次比较大的调整，将原有的key字段改为**state**，他的含义也随着改变。

state是一个复合字段：

1. 第一位表示锁当前是否被持有，初始值为0，锁被持有时标记为1；
2. 第二位表示当前是否有被唤醒的goroutine，初始值为0，当前有被唤醒的goroutine时标记为1；
3. 剩余位表示当前等待此锁的goroutine数量。

图3 state结构

release-branch.go1.0

```
type Mutex struct {
  state int32
  sema  uint32
}

...

const (
  mutexLocked = 1 << iota // mutex is locked
  mutexWoken
  mutexWaiterShift = iota
)
```

Lock

```
func (m *Mutex) Lock() {
  // Fast path: grab unlocked mutex.
  if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked
      return
  }

  // Slow path:
  awoke := false
  for {
      old := m.state
      new := old | mutexLocked // lock 标记
      if old&mutexLocked != 0 {
          new = old + 1<<mutexWaiterShift // 等待数量+1
      }
      if awoke {
          // 清除唤醒标志
          new &^= mutexWoken
      }
      if atomic.CompareAndSwapInt32(&m.state, old, new)
          if old&mutexLocked == 0 {
```

Unlock

```
func (m *Mutex) Unlock() {
  // Fast path: drop lock bit.
  new := atomic.AddInt32(&m.state, -mutexLocked)
  if (new+mutexLocked)&mutexLocked == 0 {
      // 本来就没有锁 panic
      panic("sync: unlock of unlocked mutex")
  }

  old := new
  for {
      // 没有等待者 或 已有唤醒的waiter 或 锁原来已被持有
      if old>>mutexWaiterShift == 0 || old&(mutexLocked|m
          return
      }
      // 唤醒 goroutine
      new = (old - 1<<mutexWaiterShift) | mutexWoken
      if atomic.CompareAndSwapInt32(&m.state, old, new)
          runtime_Semrelease(&m.sema)
          return
      }
```

V2.0 引入非公平锁

请求锁的 goroutine 有两类，一类是新来请求锁的 goroutine，另一类是被唤醒的等待请求锁的 goroutine。锁的状态也有两种：加锁和未加锁。通过一张表格，来说明一下 goroutine 不同来源不同状态下的处理逻辑。

	锁已被持有	**锁未被持有**
新来的goroutine	waiter++ 入队等待	获得锁
被唤醒的goroutine	waiter++ 重置mutexWoken标志 入队等待	重置mutexWoken标志 获得锁

V3.0 引入自旋

自旋是一种多线程同步机制，当前的进程在进入自旋的过程中会一直保持 CPU 的占用，持续检查某个条件是否为真。

在多核的 CPU 上，自旋可以避免 goroutine 的切换，使用恰当会对性能带来很大的增益，但是使用的不恰当就会拖慢整个程序，所以 Goroutine 进入自旋的条件非常苛刻。

release-branch.go1.5

```
func (m *Mutex) Lock() {
 // Fast path: grab unlocked mutex.
 if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked
     return
 }

 awoke := false
 iter := 0
 for {
     old := m.state
     new := old | mutexLocked
     if old&mutexLocked != 0 {
         if runtime_canSpin(iter) { // 自旋
             // Active spinning makes sense.
             // Try to set mutexWoken flag to inform Un
             // to not wake other blocked goroutines.
             if !awoke && old&mutexWoken == 0 && old>>mu
                 atomic.CompareAndSwapInt32(&m.state, o
                 awoke = true
             }
             runtime_doSpin()
             iter++
             continue
         }
         new = old + 1<<mutexWaiterShift
     }
```

runtime_canSpin()

`runtime_canSpin` 返回true的条件如下：

- 当前运行的机器是多核CPU，且GOMAXPROCS>1
- 至少存在一个其他正在运行的P，并且当前的本地运行队列（local runq）为空
- 当前goroutine进行自旋的次数小于4

```
func sync_runtime_canSpin(i int) bool {
  // sync.Mutex is cooperative, so we are conservative with spinning.
  // Spin only few times and only if running on a multicore machine and
  // GOMAXPROCS>1 and there is at least one other running P and local runq is empty.
  // As opposed to runtime mutex we don't do passive spinning here,
  // because there can be work on global runq on on other Ps.
  if i >= active_spin || ncpu <= 1 || gomaxprocs <= int32(sched.npidle+sched.nmspinning)+1 {
      return false
  }
  if p := getg().m.p.ptr(); !runqempty(p) {
      return false
  }
  return true
}
```

runtime_doSpin()

一旦当前 Goroutine 能够进入自旋就会调用runtime.sync_runtime_doSpin ， 执行 30 次的 PAUSE指令：

```
func sync_runtime_doSpin() {
  procyield(active_spin_cnt) // active_spin_cnt 常量 值为30
}
```

```
TEXT runtime·procyield(SB),NOSPLIT,$0-0
  MOVL    cycles+0(FP), AX
again:
  PAUSE
  SUBL    $1, AX
  JNZ again
  RET
```

Improves the performance of spin-wait loops. When executing a “spin-wait loop,” a Pentium 4 or Intel Xeon processor suffers a severe performance penalty when exiting the loop because it detects a possible **memory order violation**. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. **For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.**

Reference: https://www.felixcloutier.com/x86/pause.html

V4.0 解决饥饿

饥饿问题

1. 新来的goroutine在CPU上运行，比唤醒的goroutine更具有优势;
2. 处于自旋态的goroutine可以有很多个，而被唤醒的goroutine每次只能有一个;
3. 被唤醒的goroutine在未抢到锁之后被放回了队列的尾部，加剧了饥饿问题

优化方案

release-branch.go1.9:

1. 引入饥饿模式，添加mutexStarving标识，解决饥饿问题；
2. 将唤醒且未获得锁的goroutine插入队列头部;

正常模式&饥饿模式

正常模式 - 更好的性能:
1. 尝试加锁的goroutine会进入自旋, 尝试通过原子操作获得锁;
2. 若自旋几次后仍未获得锁, 通过信号量进入排队等待, 以先入先出(FIFO)顺序排队等待被唤醒;
3. 被唤醒的goroutine不会立即获得锁, 而是需要和新来的goroutine竞争;
4. 若被唤醒的goroutine竞争失败, 会被重新插入队列的**头部**;

饥饿模式 - 避免高尾延时:
1. 互斥锁的所有权会从执行Unlock的goroutine直接传递给等待队列头部的goroutine;
2. 新来的goroutine直接进入队列尾部等待, 不会进入自旋, 也不会尝试获得锁;

切换条件:
1. goroutine 加锁等待时间超过1ms -> 饥饿模式
2. goroutine 获得了锁且等待的时间少于1ms或它是最后一个等待者 -> 正常模式

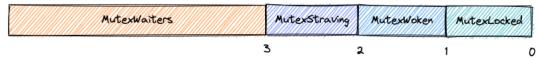

图3 state结构

- mutexLocked: 表示互斥锁处于Locked状态
- mutexWoken: 表示已有goroutine被唤醒
- mutexStarving: 表示处于饥饿模式
- waitersCount: 当前排队等待的goroutine个数

```
type Mutex struct {
  state int32
  sema  uint32
}

const (
  mutexLocked = 1 << iota // mutex is locked
  mutexWoken
  mutexStarving
  mutexWaiterShift = iota
  starvationThresholdNs = 1e6
)
```

Lock

```
// release-branch.go1.13: 将 fast path 和 slow path 拆成独立的方法，以便内联，提高性能
func (m *Mutex) Lock() {
    // Fast path: grab unlocked mutex.
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }
    // Slow path (outlined so that the fast path can be inlined)
    m.lockSlow()
}

func (m *Mutex) lockSlow() {
    var waitStartTime int64 // 等待时间
    starving := false       // 饥饿状态标识
    awoke := false          // 唤醒标识
    iter := 0               // 自旋次数
    old := m.state          // 当前锁的状态
    for {
        // 自旋条件: 正常模式 且 runtime_canSpin 返回为true
        if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSpin(iter) {
            // 当前处于主动自旋，尝试设置mutexWoken标识
            // 以避免Unlock时唤醒更多的goroutine
```

Unlock

```
// 锁的两种空闲态:
// 1.完全空闲: 即锁的初始态, 此时 m.state==0
// 2.锁空闲: 当前 mutexLocked 值为0, 但仍然有等待获取锁的goroutine
func (m *Mutex) Unlock() {
   // Fast path: 快速解锁
   new := atomic.AddInt32(&m.state, -mutexLocked)
   if new != 0 {
       // 未解锁成功
       m.unlockSlow(new)
   }
}

func (m *Mutex) unlockSlow(new int32) {
   // 校验锁的合法性
   if (new+mutexLocked)&mutexLocked == 0 {
       // unlock 一个没有上锁的 mutex
       fatal("sync: unlock of unlocked mutex")
   }

   if new&mutexStarving == 0 {
       // 正常模式
```

V4.1 TryLock

1. 2013 年 @lukescott 提出《sync: mutex.TryLock》，被拒绝。
2. 2018 年 @deanveloper 提出《proposal: add sync.Mutex.TryLock》，被拒绝。
3. 2021 年 @TyeMcQueen 提出《sync: add Mutex.TryLock》，先被拒绝，后接受。
4. 2022 年，由于之前 Go1.17 功能特性已冻结，定在 Go1.18 发布(3 月)。

issues/45435 《proposal: add sync.Mutex.TryLock》

Reference:Go1.18 新特性：三顾茅庐、被折腾 N 次的 TryLock

release-branch.go1.18

```
func (m *Mutex) TryLock() bool {
 old := m.state
 if old&(mutexLocked|mutexStarving) != 0 {
     return false
 }

 // There may be a goroutine waiting for the mutex, but
 // running now and can try to grab the mutex before tha
 // goroutine wakes up.
 if !atomic.CompareAndSwapInt32(&m.state, old, old|mutex
     return false
 }
 return true
}
```

Mutex使用的易错点

1. Lock/Unlock 没有成对出现

Lock/Unlock 没有成对出现，就意味着会出现死锁的情况，或者是因为 Unlock 一个未加锁的 Mutex 而导致