



下载APP

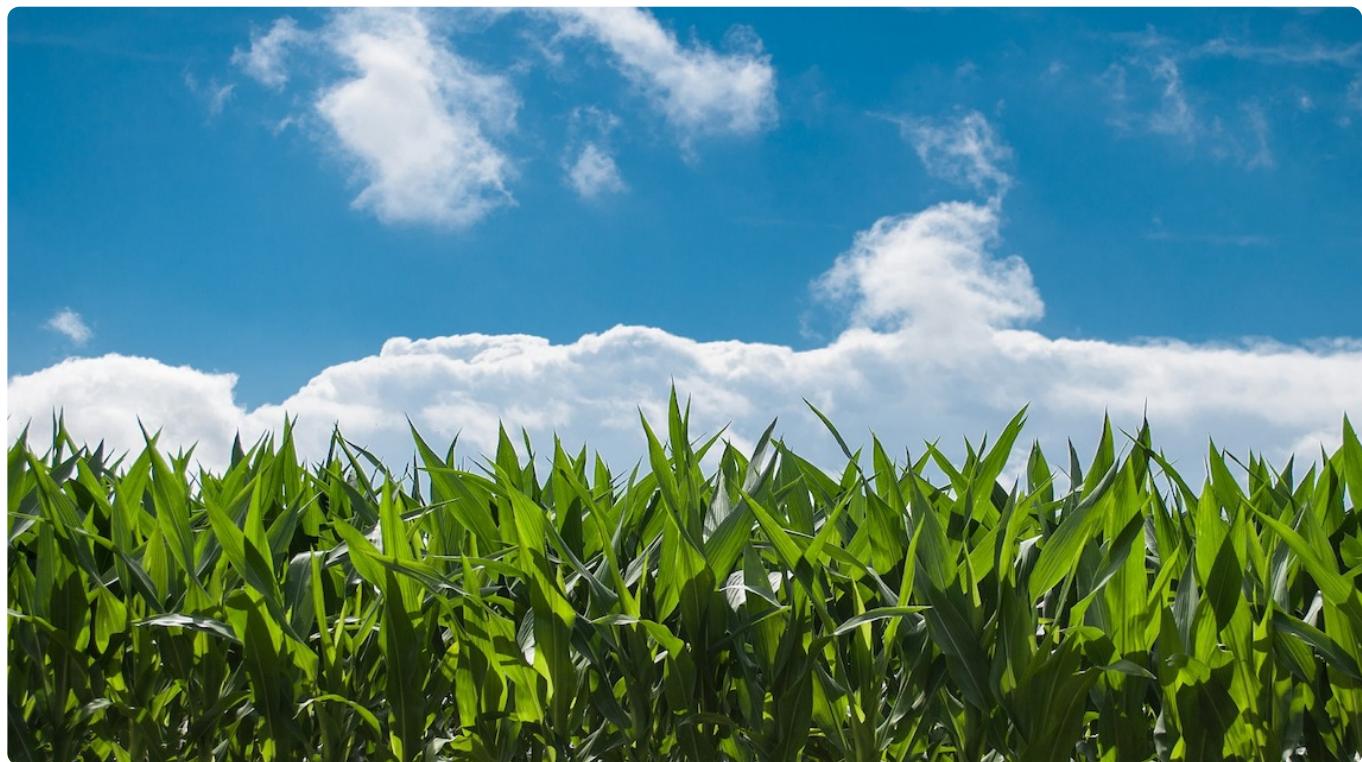


05 | RWMutex: 读写锁的实现原理及避坑指南

2020-10-21 鬼岳攀

Go 并发编程实战课

[进入课程 >](#)



讲述：安晓辉

时长 26:46 大小 24.52M



你好，我是鸟窝。

在前面的四节课中，我们学习了第一个同步原语，即 Mutex，我们使用它来保证读写共享资源的安全性。不管是读还是写，我们都通过 Mutex 来保证只有一个 goroutine 访问共享资源，这在某些情况下有点“浪费”。比如说，在写少读多的情况下，即使一段时间内没有写操作，大量并发的读访问也不得不在 Mutex 的保护下变成了串行访问，这个时候，使用 Mutex，对性能的影响就比较大。

怎么办呢？你是不是已经有思路了，对，就是区分读写操作。



我来具体解释一下。如果某个读操作的 goroutine 持有了锁，在这种情况下，其它读操作的 goroutine 就不必一直傻傻地等待了，而是可以并发地访问共享变量，这样我们就可以

将串行的读变成并行读, 提高读操作的性能。当写操作的 goroutine 持有锁的时候, 它就是一个排外锁, 其它的写操作和读操作的 goroutine, 需要阻塞等待持有这个锁的 goroutine 释放锁。

这一类并发读写问题叫作 **readers-writers 问题**, 意思就是, 同时可能有多个读或者多个写, 但是只要有一个线程在执行写操作, 其它的线程都不能执行读写操作。

Go 标准库中的 RWMutex (读写锁) 就是用来解决这类 readers-writers 问题的。所以, 这节课, 我们就一起来学习 RWMutex。我会给你介绍读写锁的使用场景、实现原理以及容易掉入的坑, 你一定要记住这些陷阱, 避免在实际的开发中犯相同的错误。

什么是 RWMutex?

我先简单解释一下读写锁 RWMutex。标准库中的 RWMutex 是一个 reader/writer 互斥锁。RWMutex 在某一时刻只能由任意数量的 reader 持有, 或者是只被单个的 writer 持有。

RWMutex 的方法也很少, 总共有 5 个。

Lock/Unlock: **写操作时调用的方法。**如果锁已经被 reader 或者 writer 持有, 那么, Lock 方法会一直阻塞, 直到能获取到锁; Unlock 则是配对的释放锁的方法。

RLock/RUnlock: **读操作时调用的方法。**如果锁已经被 writer 持有的话, RLock 方法会一直阻塞, 直到能获取到锁, 否则就直接返回; 而 RUnlock 是 reader 释放锁的方法。

RLocker: 这个方法的作用是为读操作返回一个 Locker 接口的对象。它的 Lock 方法会调用 RWMutex 的 RLock 方法, 它的 Unlock 方法会调用 RWMutex 的 RUnlock 方法。

RWMutex 的零值是未加锁的状态, 所以, 当你使用 RWMutex 的时候, 无论是声明变量, 还是嵌入到其它 struct 中, 都不必显式地初始化。

我以计数器为例, 来说明一下, 如何使用 RWMutex 保护共享资源。计数器的 **count++** 操作是写操作, 而获取 count 的值是读操作, 这个场景非常适合读写锁, 因为读操作可以并行执行, 写操作时只允许一个线程执行, 这正是 readers-writers 问题。

在这个例子中，使用 10 个 goroutine 进行读操作，每读取一次，sleep 1 毫秒，同时，还有一个 goroutine 进行写操作，每一秒写一次，这是一个 **1 writer-n reader** 的读写场景，而且写操作还不是很频繁（一秒一次）：

复制代码

```

1 func main() {
2     var counter Counter
3     for i := 0; i < 10; i++ { // 10个reader
4         go func() {
5             for {
6                 counter.Count() // 计数器读操作
7                 time.Sleep(time.Millisecond)
8             }
9         }()
10    }
11
12    for { // 一个writer
13        counter.Incr() // 计数器写操作
14        time.Sleep(time.Second)
15    }
16 }
17 // 一个线程安全的计数器
18 type Counter struct {
19     mu     sync.RWMutex
20     count uint64
21 }
22
23 // 使用写锁保护
24 func (c *Counter) Incr() {
25     c.mu.Lock()
26     c.count++
27     c.mu.Unlock()
28 }
29
30 // 使用读锁保护
31 func (c *Counter) Count() uint64 {
32     c.mu.RLock()
33     defer c.mu.RUnlock()
34     return c.count
35 }
```

可以看到，Incr 方法会修改计数器的值，是一个写操作，我们使用 Lock/Unlock 进行保护。Count 方法会读取当前计数器的值，是一个读操作，我们使用 RLock/RUnlock 方法进行保护。

Incr 方法每秒才调用一次，所以，writer 竞争锁的频次是比较低的，而 10 个 goroutine 每毫秒都要执行一次查询，通过读写锁，可以极大提升计数器的性能，因为在读取的时候，可以并发进行。如果使用 Mutex，性能就不会像读写锁这么好。因为多个 reader 并发读的时候，使用互斥锁导致了 reader 要排队读的情况，没有 RWMutex 并发读的性能好。

如果你遇到可以明确区分 reader 和 writer goroutine 的场景，且有大量的并发读、少量的并发写，并且有强烈的性能需求，你就可以考虑使用读写锁 RWMutex 替换 Mutex。

在实际使用 RWMutex 的时候，如果我们在 struct 中使用 RWMutex 保护某个字段，一般会把它和这个字段放在一起，用来指示两个字段是一组字段。除此之外，我们还可以采用匿名字段的方式嵌入 struct，这样，在使用这个 struct 时，我们就可以直接调用 Lock/Unlock、RLock/RUnlock 方法了，这和我们前面在 [01 讲](#) 中介绍 Mutex 的使用方法很类似，你可以回去复习一下。

RWMutex 的实现原理

RWMutex 是很常见的并发原语，很多编程语言的库都提供了类似的并发类型。RWMutex 一般都是基于互斥锁、条件变量（condition variables）或者信号量（semaphores）等并发原语来实现。**Go 标准库中的 RWMutex 是基于 Mutex 实现的。**

readers-writers 问题一般有三类，基于对读和写操作的优先级，读写锁的设计和实现也分成三类。

Read-preferring: 读优先的设计可以提供很高的并发性，但是，在竞争激烈的情况下可能会导致写饥饿。这是因为，如果有大量的读，这种设计会导致只有所有的读都释放了锁之后，写才可能获取到锁。

Write-preferring: 写优先的设计意味着，如果已经有一个 writer 在等待请求锁的话，它会阻止新来的请求锁的 reader 获取到锁，所以优先保障 writer。当然，如果有一些 reader 已经请求了锁的话，新请求的 writer 也会等待已经存在的 reader 都释放锁之后才能获取。所以，写优先级设计中的优先权是针对新来的请求而言的。这种设计主要避免了 writer 的饥饿问题。

不指定优先级: 这种设计比较简单，不区分 reader 和 writer 优先级，某些场景下这种不指定优先级的设计反而更有效，因为第一类优先级会导致写饥饿，第二类优先级可能

会导致读饥饿，这种不指定优先级的访问不再区分读写，大家都是同一个优先级，解决了饥饿的问题。

Go 标准库中的 RWMutex 设计是 Write-preferring 方案。一个正在阻塞的 Lock 调用会排除新的 reader 请求到锁。

RWMutex 包含一个 Mutex，以及四个辅助字段 writerSem、readerSem、readerCount 和 readerWait：

 复制代码

```

1 type RWMutex struct {
2     w           Mutex   // 互斥锁解决多个writer的竞争
3     writerSem   uint32  // writer信号量
4     readerSem   uint32  // reader信号量
5     readerCount int32   // reader的数量
6     readerWait  int32   // writer等待完成的reader的数量
7 }
8
9 const rwmutexMaxReaders = 1 << 30

```

我来简单解释一下这几个字段。

字段 w：为 writer 的竞争锁而设计；

字段 readerCount：记录当前 reader 的数量（以及是否有 writer 竞争锁）；

readerWait：记录 writer 请求锁时需要等待 read 完成的 reader 的数量；

writerSem 和 readerSem：都是为了阻塞设计的信号量。

这里的常量 rwmutexMaxReaders，定义了最大的 reader 数量。

好了，知道了 RWMutex 的设计方案和具体字段，下面我来解释一下具体的方法实现。

RLock/RUnlock 的实现

首先，我们看一下移除了 race 等无关紧要的代码后的 RLock 和 RUnlock 方法：

 复制代码

```

1 func (rw *RWMutex) RLock() {
2     if atomic.AddInt32(&rw.readerCount, 1) < 0 {
3         // rw.readerCount是负值的时候，意味着此时有writer等待请求锁，因为writer优先级高
4         runtime_SemacquireMutex(&rw.readerSem, false, 0)
5     }
6 }
7 }
8 func (rw *RWMutex) RUnlock() {
9     if r := atomic.AddInt32(&rw.readerCount, -1); r < 0 {
10        rw.rUnlockSlow(r) // 有等待的writer
11    }
12 }
13 func (rw *RWMutex) rUnlockSlow(r int32) {
14     if atomic.AddInt32(&rw.readerWait, -1) == 0 {
15         // 最后一个reader了，writer终于有机会获得锁了
16         runtime_Semrelease(&rw.writerSem, false, 1)
17     }

```

第 2 行是对 reader 计数加 1。你可能比较困惑的是，readerCount 怎么还可能为负数呢？其实，这是因为，readerCount 这个字段有双重含义：

没有 writer 竞争或持有锁时，readerCount 和我们正常理解的 reader 的计数是一样的；

但是，如果有 writer 竞争锁或者持有锁时，那么，readerCount 不仅仅承担着 reader 的计数功能，还能够标识当前是否有 writer 竞争或持有锁，在这种情况下，请求锁的 reader 的处理进入第 4 行，阻塞等待锁的释放。

调用 RUnlock 的时候，我们需要将 Reader 的计数减去 1（第 8 行），因为 reader 的数量减少了一个。但是，第 8 行的 AddInt32 的返回值还有另外一个含义。如果它是负值，就表示当前有 writer 竞争锁，在这种情况下，还会调用 rUnlockSlow 方法，检查是不是 reader 都释放读锁了，如果读锁都释放了，那么可以唤醒请求写锁的 writer 了。

当一个或者多个 reader 持有锁的时候，竞争锁的 writer 会等待这些 reader 释放完，才可能持有这把锁。打个比方，在房地产行业中有条规矩叫做“**买卖不破租赁**”，意思是说，就算房东把房子卖了，新业主也不能把当前的租户赶走，而是要等到租约结束后，才能接管房子。这和 RWMutex 的设计是一样的。当 writer 请求锁的时候，是无法改变既有的 reader 持有锁的现实的，也不会强制这些 reader 释放锁，它的优先权只是限定后来的 reader 不要和它抢。

所以，rUnlockSlow 将持有锁的 reader 计数减少 1 的时候，会检查既有的 reader 是不是都已经释放了锁，如果都释放了锁，就会唤醒 writer，让 writer 持有锁。

Lock

RWMutex 是一个多 writer 多 reader 的读写锁，所以同时可能有多个 writer 和 reader。那么，为了避免 writer 之间的竞争，RWMutex 就会使用一个 Mutex 来保证 writer 的互斥。

一旦一个 writer 获得了内部的互斥锁，就会反转 readerCount 字段，把它从原来的正整数 readerCount($>=0$) 修改为负数 ($readerCount - rwmutexMaxReaders$)，让这个字段保持两个含义（既保存了 reader 的数量，又表示当前有 writer）。

我们来看下下面的代码。第 5 行，还会记录当前活跃的 reader 数量，所谓活跃的 reader，就是指持有读锁还没有释放的那些 reader。

 复制代码

```

1 func (rw *RWMutex) Lock() {
2     // 首先解决其他writer竞争问题
3     rw.w.Lock()
4     // 反转readerCount，告诉reader有writer竞争锁
5     r := atomic.AddInt32(&rw.readerCount, -rwmutexMaxReaders) + rwmutexMaxRead
6     // 如果当前有reader持有锁，那么需要等待
7     if r != 0 && atomic.AddInt32(&rw.readerWait, r) != 0 {
8         runtime_SemacquireMutex(&rw.writerSem, false, 0)
9     }
10 }
```

如果 readerCount 不是 0，就说明当前有持有读锁的 reader，RWMutex 需要把这个当前 readerCount 赋值给 readerWait 字段保存下来（第 7 行），同时，这个 writer 进入阻塞等待状态（第 8 行）。

每当一个 reader 释放读锁的时候（调用 RUnlock 方法时），readerWait 字段就减 1，直到所有的活跃的 reader 都释放了读锁，才会唤醒这个 writer。

Unlock

当一个 writer 释放锁的时候，它会再次反转 readerCount 字段。可以肯定的是，因为当前锁由 writer 持有，所以，readerCount 字段是反转过的，并且减去了 rwmutexMaxReaders 这个常数，变成了负数。所以，这里的反转方法就是给它增加 rwmutexMaxReaders 这个常数值。

既然 writer 要释放锁了，那么就需要唤醒之后新来的 reader，不必再阻塞它们了，让它们开开心心地继续执行就好了。

在 RWMutex 的 Unlock 返回之前，需要把内部的互斥锁释放。释放完毕后，其他的 writer 才可以继续竞争这把锁。

 复制代码

```

1 func (rw *RWMutex) Unlock() {
2     // 告诉reader没有活跃的writer了
3     r := atomic.AddInt32(&rw.readerCount, rwmutexMaxReaders)
4
5     // 唤醒阻塞的reader们
6     for i := 0; i < int(r); i++ {
7         runtime_Semrelease(&rw.readerSem, false, 0)
8     }
9     // 释放内部的互斥锁
10    rw.w.Unlock()
11 }
```

在这段代码中，我删除了 race 的处理和异常情况的检查，总体看来还是比较简单的。这里有几个重点，我要再提醒你一下。首先，你要理解 readerCount 这个字段的含义以及反转方式。其次，你还要注意字段的更改和内部互斥锁的顺序关系。在 Lock 方法中，是先获取内部互斥锁，才会修改的其他字段；而在 Unlock 方法中，是先修改的其他字段，才会释放内部互斥锁，这样才能保证字段的修改也受到互斥锁的保护。

好了，到这里我们就完整学习了 RWMutex 的概念和实现原理。RWMutex 的应用场景非常明确，就是解决 readers-writers 问题。学完了今天的内容，之后当你遇到这类问题时，要优先想到 RWMutex。另外，Go 并发原语代码实现的质量都很高，非常精炼和高效，所以，你可以通过看它们的实现原理，学习一些编程的技巧。当然，还有非常重要的一点就是要知道 reader 或者 writer 请求锁的时候，既有的 reader/writer 和后续请求锁的 reader/writer 之间的（释放锁 / 请求锁）顺序关系。

有个很有意思的事儿，就是官方的文档对 RWMutex 介绍是错误的，或者说是不明确的，在下一个版本（Go 1.16）中，官方会更改对 RWMutex 的介绍，具体是这样的：

A RWMutex is a reader/writer mutual exclusion lock.

The lock can be held by any number of readers or a single writer, and

a blocked writer also blocks new readers from acquiring the lock.

这个描述是相当精确的，它指出了 RWMutex 可以被谁持有，以及 writer 比后续的 reader 有获取锁的优先级。

虽然 RWMutex 暴露的 API 也很简单，使用起来也没有复杂的逻辑，但是和 Mutex 一样，在实际使用的时候，也会很容易踩到一些坑。接下来，我给你重点介绍 3 个常见的踩坑点。

RWMutex 的 3 个踩坑点

坑点 1：不可复制

前面刚刚说过，RWMutex 是由一个互斥锁和四个辅助字段组成的。我们很容易想到，互斥锁是不可复制的，再加上四个有状态的字段，RWMutex 就更加不能复制使用了。

不能复制的原因和互斥锁一样。一旦读写锁被使用，它的字段就会记录它当前的一些状态。这个时候你去复制这把锁，就会把它的状态也给复制过来。但是，原来的锁在释放的时候，并不会修改你复制出来的这个读写锁，这就会导致复制出来的读写锁的状态不对，可能永远无法释放锁。

那该怎么办呢？其实，解决方案也和互斥锁一样。你可以借助 vet 工具，在变量赋值、函数传参、函数返回值、遍历数据、struct 初始化等时，检查是否有读写锁隐式复制的情景。

坑点 2：重入导致死锁

读写锁因为重入（或递归调用）导致死锁的情况更多。

我先介绍第一种情况。因为读写锁内部基于互斥锁实现对 writer 的并发访问，而互斥锁本身是有重入问题的，所以，writer 重入调用 Lock 的时候，就会出现死锁的现象，这个问题，我们在学习互斥锁的时候已经了解过了。

复制代码

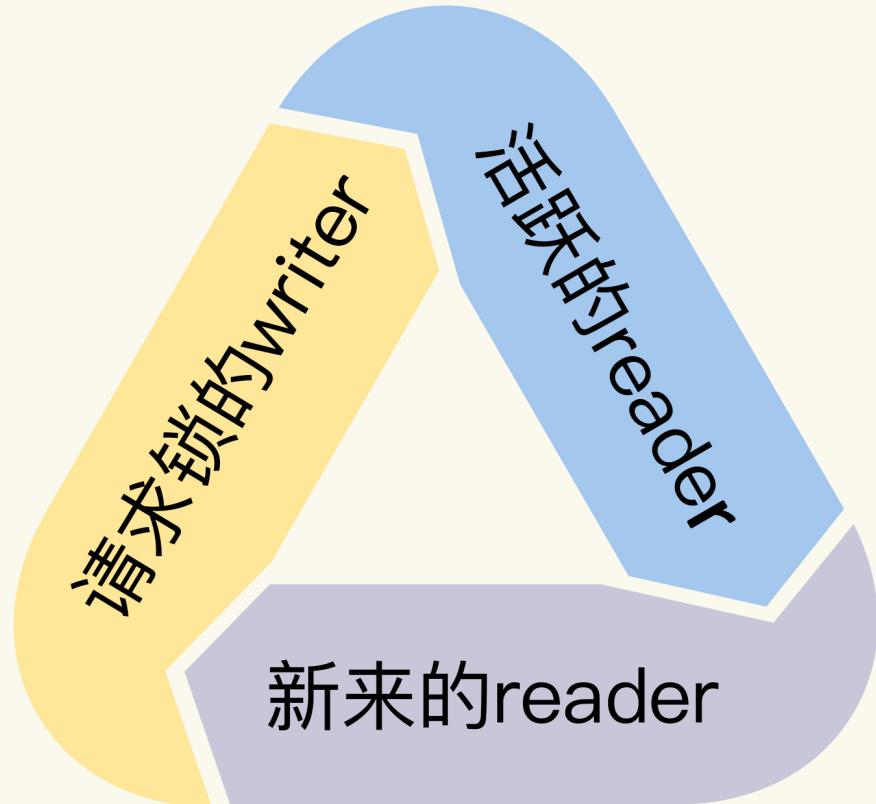
```
1 func foo(l *sync.RWMutex) {
2     fmt.Println("in foo")
3     l.Lock()
4     bar(l)
5     l.Unlock()
6 }
7
8 func bar(l *sync.RWMutex) {
9     l.Lock()
10    fmt.Println("in bar")
11    l.Unlock()
12 }
13
14 func main() {
15     l := &sync.RWMutex{}
16     foo(l)
17 }
```

运行这个程序，你就会得到死锁的错误输出，在 Go 运行的时候，很容易就能检测出来。

第二种死锁的场景有点隐蔽。我们知道，有活跃 reader 的时候，writer 会等待，如果我们在 reader 的读操作时调用 writer 的写操作（它会调用 Lock 方法），那么，这个 reader 和 writer 就会形成互相依赖的死锁状态。Reader 想等待 writer 完成后再释放锁，而 writer 需要这个 reader 释放锁之后，才能不阻塞地继续执行。这是一个读写锁常见的死锁场景。

第三种死锁的场景更加隐蔽。

当一个 writer 请求锁的时候，如果已经有一些活跃的 reader，它会等待这些活跃的 reader 完成，才有可能获取到锁，但是，如果之后活跃的 reader 再依赖新的 reader 的话，这些新的 reader 就会等待 writer 释放锁之后才能继续执行，这就形成了一个环形依赖：**writer 依赖活跃的 reader -> 活跃的 reader 依赖新来的 reader -> 新来的 reader 依赖 writer。**



这个死锁相当隐蔽，原因在于它和 RWMutex 的设计和实现有关。啥意思呢？我们来看一个计算阶乘 ($n!$) 的例子：

复制代码

```
1 func main() {
2     var mu sync.RWMutex
3
4     // writer,稍微等待, 然后制造一个调用Lock的场景
5     go func() {
6         time.Sleep(200 * time.Millisecond)
7         mu.Lock()
8         fmt.Println("Lock")
9         time.Sleep(100 * time.Millisecond)
10        mu.Unlock()
11        fmt.Println("Unlock")
12    }()
13
14    go func() {
15        factorial(&mu, 10) // 计算10的阶乘, 10!
16    }()
17}
```

```

18     select {}
19 }
20
21 // 递归调用计算阶乘
22 func factorial(m *sync.RWMutex, n int) int {
23     if n < 1 { // 阶乘退出条件
24         return 0
25     }
26     fmt.Println("RLock")
27     m.RLock()
28     defer func() {
29         fmt.Println("RUnlock")
30         m.RUnlock()
31     }()
32     time.Sleep(100 * time.Millisecond)
33     return factorial(m, n-1) * n // 递归调用
34

```

factoria 方法是一个递归计算阶乘的方法，我们用它来模拟 reader。为了更容易地制造出死锁场景，我在这里加上了 sleep 的调用，延缓逻辑的执行。这个方法会调用读锁（第 27 行），在第 33 行递归地调用此方法，每次调用都会产生一次读锁的调用，所以可以不断地产生读锁的调用，而且必须等到新请求的读锁释放，这个读锁才能释放。

同时，我们使用另一个 goroutine 去调用 Lock 方法，来实现 writer，这个 writer 会等待 200 毫秒后才会调用 Lock，这样在调用 Lock 的时候，factoria 方法还在执行中不断调用 RLock。

这两个 goroutine 互相持有锁并等待，谁也不会退让一步，满足了“writer 依赖活跃的 reader -> 活跃的 reader 依赖新来的 reader -> 新来的 reader 依赖 writer”的死锁条件，所以就导致了死锁的产生。

所以，使用读写锁最需要注意的一点就是尽量避免重入，重入带来的死锁非常隐蔽，而且难以诊断。

坑点 3：释放未加锁的 RWMutex

和互斥锁一样，Lock 和 Unlock 的调用总是成对出现的，RLock 和 RUnlock 的调用也必须成对出现。Lock 和 RLock 多余的调用会导致锁没有被释放，可能会出现死锁，而 Unlock 和 RUnlock 多余的调用会导致 panic。在生产环境中出现 panic 是大忌，你总不

希望半夜爬起来处理生产环境程序崩溃的问题吧？所以，在使用读写锁的时候，一定要注意，**不遗漏不多余**。

流行的 Go 开发项目中的坑

好了，又到了泡一杯宁夏枸杞加新疆大滩枣的养生茶，静静地欣赏知名项目出现 Bug 的时候了，这次被拉出来的是 RWMutex 的 Bug。

Docker

issue 36840

issue 36840修复的是错误地把 writer 当成 reader 的 Bug。这个地方本来需要修改数据，需要调用的是写锁，结果用的却是读锁。或许是被它紧挨着的 findNode 方法调用迷惑了，认为这只是一个读操作。可实际上，代码后面还会有 changeNodeState 方法的调用，这是一个写操作。修复办法也很简单，只需要改成 Lock/Unlock 即可。

```

    ↴ 4 vendor/github.com/docker/libnetwork/networkdb/delegate.go ⌂
    ↑ @@ -21,8 +21,8 @@ func (nDB *NetworkDB) handleNodeEvent(nEvent *NodeEvent) bool {
21   21     // time.
22   22     nDB.networkClock.Witness(nEvent.LTime)
23   23
24   -     nDB.RLock()
25   -     defer nDB.RUnlock()
24   +     nDB.Lock()
25   +     defer nDB.Unlock()
26   26
27   27     // check if the node exists
28   28     n, _, _ := nDB.findNode(nEvent.NodeName)
    ...
  
```

Kubernetes

issue 62464

issue 62464就是读写锁第二种死锁的场景，这是一个典型的 reader 导致的死锁的例子。知道墨菲定律吧？“凡是可能出错的事，必定会出错”。你可能觉得我前面讲的 RWMutex 的坑绝对不会被人踩的，因为道理大家都懂，但是你看，Kubernetes 就踩了这个重入的坑。

这个 issue 在移除 pod 的时候可能会发生，原因就在于，GetCPUSetOrDefault 方法会请求读锁，同时，它还会调用 GetCPUSet 或 GetDefaultCPUSet 方法。当这两个方法都请

求写锁时，是获取不到的，因为 GetCPUSetOrDefault 方法还没有执行完，不会释放读锁，这就形成了死锁。

```

58 58     func (s *stateMemory) GetCPUSetOrDefault(containerID string) cpuset.CPUSet {
59   -         s.RLock()
60   -         defer s.RUnlock()
61   -
62 59             if res, ok := s.GetCPUSet(containerID); ok {
63 60                 return res
64 61             }
65 62             return s.GetDefaultCPUSet()
66 63         }
67 64
68 65     func (s *stateMemory) GetCPUAssignments() ContainerCPUAssignments {
69 66         s.RLock()
70 67         defer s.RUnlock()
71 68         return s.assignments.Clone()
72 69     }
73 70
74 71     func (s *stateMemory) SetCPUSet(containerID string, cset cpuset.CPUSet) {
75 72         s.Lock()
76 73         defer s.Unlock()
77 74
78 75             s.assignments[containerID] = cset
79 76             glog.Infof("[cpumanager] updated desired cpuset (container id: %s, cpuset: \"%s\")", containerID, cset)
80 77         }
81 78
82 79     func (s *stateMemory) SetDefaultCPUSet(cset cpuset.CPUSet) {
83 80         s.Lock()
84 81         defer s.Unlock()

```

总结

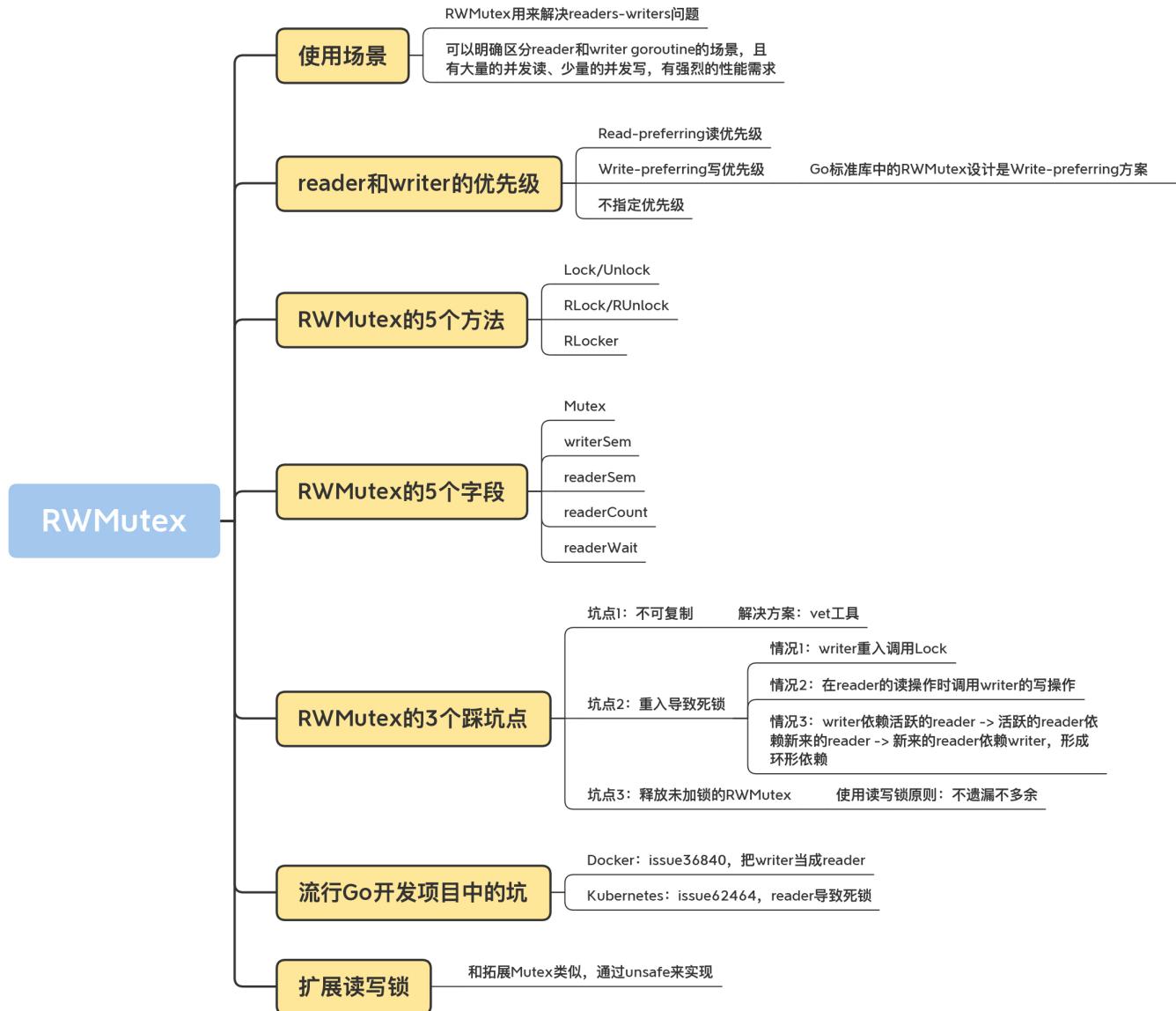
在开发过程中，一开始考虑共享资源并发访问问题的时候，我们就会想到互斥锁 Mutex。因为刚开始的时候，我们还并不太了解并发的情况，所以，就会使用最简单的同步原语来解决问题。等到系统成熟，真正到了需要性能优化的时候，我们就能静下心来分析并发场景的可能性，这个时候，我们就要考虑将 Mutex 修改为 RWMutex，来压榨系统的性能。

当然，如果一开始你的场景就非常明确了，比如我就要实现一个线程安全的 map，那么，一开始你就可以考虑使用读写锁。

正如我在前面提到的，如果你能意识到你要解决的问题是一个 readers-writers 问题，那么你就可以毫不犹豫地选择 RWMutex，不用考虑其它选择。那在使用 RWMutex 时，最需要注意的一点就是尽量避免重入，重入带来的死锁非常隐蔽，而且难以诊断。

另外我们也可以扩展 RWMutex，不过实现方法和互斥锁 Mutex 差不多，在技术上是一样的，都是通过 unsafe 来实现，我就不再具体讲了。课下你可以参照我们 [上节课学习的方法](#)，实现一个扩展的 RWMutex。

这一讲我们系统学习了读写锁的相关知识，这里提供给你一个知识地图，帮助你复习本节课的知识。



思考题

请你写一个扩展的读写锁，比如提供 TryLock，查询当前是否有 writer、reader 的数量等方法。

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你把今天的内容分享给你的朋友或同事。

提建议

Go并发编程实战课

鸟窝带你攻克并发编程难题

晁岳攀 (鸟窝)

前微博技术专家

知名微服务框架 rpcx 的作者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金奖励**。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | Mutex: 骇客编程，如何拓展额外功能？

下一篇 06 | WaitGroup: 协同等待，任务编排利器

精选留言 (9)

写留言



Junes

2020-10-21

交一下作业，我就不贴完整代码了，分享一下核心思路：

获取两个关键变量，大致思路是根据 起始地址+偏移量，

// readerCount 这个成员变量前有1个mutex+2个uint32

readerCount := atomic.LoadInt32((*int32)(unsafe.Pointer(uintptr(unsafe.Pointer(...

展开 ▾



9



Gopher

2020-10-23

作业思路

和mutex的扩展思路一样，通过unsafe获取指针，在进行偏移获取到reader数量，不等于0直接返回，否则尝试lock



那时刻

2020-10-23

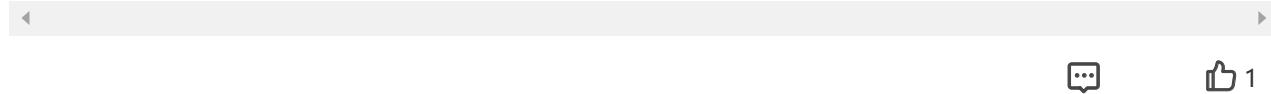
请问老师一个关于select{}问题,

```
func foo() {
    fmt.Println("in foo")
}...
```

展开 ▾

作者回复: 现在版本的go运行时过于智能了, 会把这个场景 “误报” 成死锁, 其实我们的本意是让程序hang在这里。

可以改成sleep, waitgroup或者从命令行读取数据等的方式阻塞主goroutine



Ethan Liu

2020-10-21

区分reader和writer的场景, 可不可以使用channel来实现? 如果可以的话, 与使用RWMutex有什么区别?



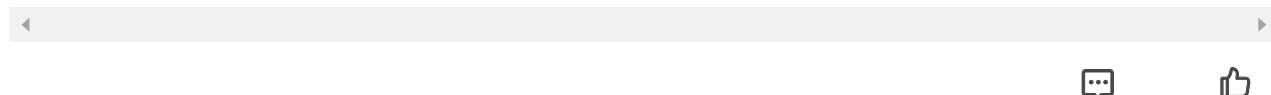
Yayu

2020-10-23

老师好, 请问“原语”的意思是什么? 原子性语句吗?

展开 ▾

作者回复: 基础性数据结构, 英语是primitive, 不是原子性语句哈



Linuxer

2020-10-22

试着回答一下课后题, 初学GO, 请各位大佬指点

```
const {
    READ = 0
    WRITE = 1
}...
```

展开 ▼



Panda

2020-10-21

RWMutex 是 Mutex 的增强版本 也是分而治之的思想体现



pdf

2020-10-21

Go 1.15

main函数结尾 select {} 直接报死锁

6



橙子888

2020-10-21

又是一个需要花时间消化的章节，理解读写锁的原理之后，再参考之前 Mutex 章节扩展的实现，写一个扩展的读写锁应该不难，惊讶地发现已经有大佬给出答案了.....

