



下载APP



03 | Mutex: 4种易错场景大盘点

2020-10-16 晁岳攀/鸟窝

Go 并发编程实战课

[进入课程 >](#)**讲述：安晓辉**

时长 26:50 大小 24.58M



你好，我是鸟窝。

上一讲，我带你一起领略了 Mutex 的架构演进之美，现在我们已经清楚 Mutex 的实现细节了。当前 Mutex 的实现貌似非常复杂，其实主要还是针对饥饿模式和公平性问题，做了一些额外处理。但是，我们在第一讲中已经体验过了，Mutex 使用起来还是非常简单的，毕竟，它只有 Lock 和 Unlock 两个方法，使用起来还能复杂到哪里去？

正常使用 Mutex 时，确实是这样的，很简单，基本不会有什么错误，即使出现错误，也是在一些复杂的场景中，比如跨函数调用 Mutex 或者是在重构或者修补 Bug 时误操作。是，我们使用 Mutex 时，确实会出现一些 Bug，比如说忘记释放锁、重入锁、复制已使用了的 Mutex 等情况。那在这一讲中，我们就一起来看看使用 Mutex 常犯的几个错误，做到“Bug 提前知，后面早防范”。



常见的 4 种错误场景

我总结了一下，使用 Mutex 常见的错误场景有 4 类，分别是 Lock/Unlock 不是成对出现、Copy 已使用的 Mutex、重入和死锁。下面我们——来看。

Lock/Unlock 不是成对出现


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 panic。

我们先来看看缺少 Unlock 的场景，常见的有三种情况：

1. 代码中有太多的 if-else 分支，可能在某个分支中漏写了 Unlock；
2. 在重构的时候把 Unlock 给删除了；
3. Unlock 误写成了 Lock。

在这种情况下，锁被获取之后，就不会被释放了，这也就意味着，其它的 goroutine 永远都没机会获取到锁。

我们再来看缺少 Lock 的场景，这就很简单了，一般来说就是误操作删除了 Lock。比如先前使用 Mutex 都是正常的，结果后来其他人重构代码的时候，由于对代码不熟悉，或者由于开发者的马虎，把 Lock 调用给删除了，或者注释掉了。比如下面的代码，mu.Lock() 一行代码被删除了，直接 Unlock 一个未加锁的 Mutex 会 panic：

 复制代码

```
1 func foo() {  
2     var mu sync.Mutex  
3     defer mu.Unlock()  
4     fmt.Println("hello world!")  
5 }
```

运行的时候 panic：

```
smallest 1.3mutex1 go run unlock.go
hello world!
fatal error: sync: unlock of unlocked mutex


goroutine 1 [running]:
runtime.throw(0x4c4225, 0x1e)
    /usr/local/go/src/runtime/panic.go:1116 +0x72 fp=0xc000070e98 sp=0xc000070e68 pc=0x42ef82
sync.throw(0x4c4225, 0x1e)
    /usr/local/go/src/runtime/panic.go:1102 +0x35 fp=0xc000070eb8 sp=0xc000070e98 pc=0x42ef05
sync.(*Mutex).unlockSlow(0xc000012068, 0xffffffff)
    /usr/local/go/src/sync/mutex.go:196 +0xd6 fp=0xc000070ee0 sp=0xc000070eb8 pc=0x46b436
```

Copy 已使用的 Mutex

第二种误用是 Copy 已使用的 Mutex。在正式分析这个错误之前，我先交代一个小知识点，那就是 Package sync 的同步原语在使用后是不能复制的。我们知道 Mutex 是最常用的一个同步原语，那它也是不能复制的。为什么呢？

原因在于，Mutex 是一个有状态的对象，它的 state 字段记录这个锁的状态。如果你要复制一个已经加锁的 Mutex 给一个新的变量，那么新的刚初始化的变量居然被加锁了，这显然不符合你的期望，因为你期望的是一个零值的 Mutex。关键是在并发环境下，你根本不知道要复制的 Mutex 状态是什么，因为要复制的 Mutex 是由其它 goroutine 并发访问的，状态可能总是在变化。

当然，你可能说，你说的我都懂，你的警告我都记下了，但是实际在使用的时候，一不小心就踩了这个坑，我们来看一个例子。

 复制代码

```
1 type Counter struct {
2     sync.Mutex
3     Count int
4 }
5
6
7 func main() {
8     var c Counter
9     c.Lock()
10    defer c.Unlock()
11    c.Count++
12    foo(c) // 复制锁
13 }
14
15 // 这里Counter的参数是通过复制的方式传入的
16 func foo(c Counter) {
17     c.Lock()
18     defer c.Unlock()
19     fmt.Println("in foo")
20 }
```

```
20 }
```

第 12 行在调用 `foo` 函数的时候，调用者会复制 `Mutex` 变量 `c` 作为 `foo` 函数的参数，不幸的是，复制之前已经使用了这个锁，这就导致，复制的 `Counter` 是一个带状态 `Counter`。

怎么办呢？Go 在运行时，有**死锁的检查机制**（[@checkdead\(\)](#) 方法），它能够发现死锁的 `goroutine`。这个例子中因为复制了一个使用了的 `Mutex`，导致锁无法使用，程序处于死锁的状态。程序运行的时候，死锁检查机制能够发现这种死锁情况并输出错误信息，如下图中错误信息以及错误堆栈：

```
smallnest ➤ 1.3mutex2 ➤ go run copy.go
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_SemacquireMutex(0xc00010c014, 0xc000070e00, 0x1)
    /usr/local/go/src/runtime/sema.go:71 +0x47
sync.(*Mutex).lockSlow(0xc00010c010)
    /usr/local/go/src/sync/mutex.go:138 +0xfc
sync.(*Mutex).Lock(...)
    /usr/local/go/src/sync/mutex.go:81
main.foo(0x1, 0x1)
    /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.3mutex2/copy.go:22 +0x114
main.main()
    /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.3mutex2/copy.go:18 +0x8a
exit status 2
```

你肯定不想运行的时候才发现这个因为复制 `Mutex` 导致的死锁问题，那么你怎么能够及时发现问题呢？可以使用 **vet 工具**，把检查写在 `Makefile` 文件中，在持续集成的时候跑一跑，这样可以及时发现问题，及时修复。我们可以使用 `go vet` 检查这个 Go 文件：

```
✖ smallnest ➤ 1.3mutex2 ➤ go vet copy.go
# command-line-arguments
./copy.go:18:6: call of foo copies lock value: command-line-arguments.Counter
./copy.go:21:12: foo passes lock by value: command-line-arguments.Counter
✖ smallnest ➤ 1.3mutex2 ➤ |
```

你看，使用这个工具就可以发现 `Mutex` 复制的问题，错误信息显示得很清楚，是在调用 `foo` 函数的时候发生了 `lock value` 复制的情况，还告诉我们出问题的代码行数以及 `copy lock` 导致的错误。

那么，`vet` 工具是怎么发现 `Mutex` 复制使用问题的呢？我带你简单分析一下。

检查是通过 [@copylock](#) 分析器静态分析实现的。这个分析器会分析函数调用、range 遍历、复制、声明、函数返回值等位置，有没有锁的值 copy 的情景，以此来判断有没有问题。可以说，只要是实现了 Locker 接口，就会被分析。我们看到，下面的代码就是确定什么类型会被分析，其实就是实现了 Lock/Unlock 两个方法的 Locker 接口：

[复制代码](#)

```
1 var lockerType *types.Interface
2
3 // Construct a sync.Locker interface type.
4 func init() {
5     nullary := types.NewSignature(nil, nil, nil, false) // func()
6     methods := []*types.Func{
7         types.NewFunc(token.NoPos, nil, "Lock", nullary),
8         types.NewFunc(token.NoPos, nil, "Unlock", nullary),
9     }
10    lockerType = types.NewInterface(methods, nil).Complete()
11 }
```

其实，有些没有实现 Locker 接口的同步原语（比如 WaitGroup），也能被分析。我先卖个关子，后面我们会介绍这种情况是怎么实现的。

重入

接下来，我们来讨论“重入”这个问题。在说这个问题前，我先解释一下个概念，叫“可重入锁”。

如果你学过 Java，可能会很熟悉 ReentrantLock，就是可重入锁，这是 Java 并发包中非常常用的一个同步原语。它的基本行为和互斥锁相同，但是加了一些扩展功能。

如果你没接触过 Java，也没关系，这里只是提一下，帮助会 Java 的同学对比来学。那下面我来具体讲解可重入锁是咋回事儿。

当一个线程获取锁时，如果没有其它线程拥有这个锁，那么，这个线程就成功获取到这个锁。之后，如果其它线程再请求这个锁，就会处于阻塞等待的状态。但是，如果拥有这把锁的线程再请求这把锁的话，不会阻塞，而是成功返回，所以叫可重入锁（有时候也叫做递归锁）。只要你拥有这把锁，你可以可着劲儿地调用，比如通过递归实现一些算法，调用者不会阻塞或者死锁。

了解了可重入锁的概念，那我们来看 Mutex 使用的错误场景。划重点了：**Mutex 不是可重入的锁。**

想想也不奇怪，因为 Mutex 的实现中没有记录哪个 goroutine 拥有这把锁。理论上，任何 goroutine 都可以随意地 Unlock 这把锁，所以没办法计算重入条件，毕竟，“臣妾做不到啊”！

所以，一旦误用 Mutex 的重入，就会导致报错。下面是一个误用 Mutex 的重入例子：

[复制代码](#)

```
1 func foo(l sync.Locker) {
2     fmt.Println("in foo")
3     l.Lock()
4     bar(l)
5     l.Unlock()
6 }
7
8
9 func bar(l sync.Locker) {
10    l.Lock()
11    fmt.Println("in bar")
12    l.Unlock()
13 }
14
15
16 func main() {
17     l := &sync.Mutex{}
18     foo(l)
19 }
```

写完这个 Mutex 重入的例子后，运行一下，你会发现类似下面的错误。程序一直在请求锁，但是一直没有办法获取到锁，结果就是 Go 运行时发现死锁了，没有其它地方能够释放锁让程序运行下去，你通过下面的错误堆栈信息就能定位到哪一行阻塞请求锁：

```

smallnest 1.3mutex3 go run ReentrantLock.go
in foo
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_SemacquireMutex(0xc00001206c, 0x55cc00, 0x1)
    /usr/local/go/src/runtime/sema.go:71 +0x47
sync.(*Mutex).lockSlow(0xc000012068)
    /usr/local/go/src/sync/mutex.go:138 +0xfc
sync.(*Mutex).Lock(0xc000012068)
    /usr/local/go/src/sync/mutex.go:81 +0x47
main.bar(0x4db5a0, 0xc000012068)
    /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.3mutex3/ReentrantLock.go:16 +0x35
main.foo(0x4db5a0, 0xc000012068)
    /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.3mutex3/ReentrantLock.go:11 +0xa5
main.main()
    /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.3mutex3/ReentrantLock.go:23 +0x3d

```

学到这里，你可能要问了，虽然标准库 Mutex 不是可重入锁，但是如果我就是想要实现一个可重入锁，可以吗？

可以，那我们就自己实现一个。这里的关键就是，实现的锁要能记住当前是哪个 goroutine 持有这个锁。我来提供两个方案。

方案一：通过 hacker 的方式获取到 goroutine id，记录下获取锁的 goroutine id，它可以实现 Locker 接口。

方案二：调用 Lock/Unlock 方法时，由 goroutine 提供一个 token，用来标识它自己，而不是我们通过 hacker 的方式获取到 goroutine id，但是，这样一来，就不满足 Locker 接口了。

可重入锁（递归锁）解决了代码重入或者递归调用带来的死锁问题，同时它也带来了另一个好处，就是我们可以要求，只有持有锁的 goroutine 才能 unlock 这个锁。这也很容易实现，因为在上面这两个方案中，都已经记录了是哪一个 goroutine 持有这个锁。

下面我们具体来看这两个方案怎么实现。

方案一：goroutine id

这个方案的关键第一步是获取 goroutine id，方式有两种，分别是简单方式和 hacker 方式。

简单方式，就是通过 runtime.Stack 方法获取栈帧信息，栈帧信息里包含 goroutine id。你可以看看上面 panic 时候的贴图，goroutine id 明明白白地显示在那里。

`runtime.Stack` 方法可以获取当前的 goroutine 信息，第二个参数为 `true` 会输出所有的 goroutine 信息，信息的格式如下：

[复制代码](#)

```
1 goroutine 1 [running]:
2 main.main()
3      ....../main.go:19 +0xb1
```

第一行格式为 `goroutine xxx`，其中 `xxx` 就是 goroutine id，你只要解析出这个 id 即可。解析的方法可以采用下面的代码：

[复制代码](#)

```
1 func GoID() int {
2     var buf [64]byte
3     n := runtime.Stack(buf[:], false)
4     // 得到id字符串
5     idField := strings.Fields(strings.TrimPrefix(string(buf[:n]), "goroutine "))
6     id, err := strconv.Atoi(idField)
7     if err != nil {
8         panic(fmt.Sprintf("cannot get goroutine id: %v", err))
9     }
10    return id
11 }
```

了解了简单方式，接下来我们来看 hacker 的方式，这也是我们方案一采取的方式。

首先，我们获取运行时的 `g` 指针，反解出对应的 `g` 的结构。每个运行的 goroutine 结构的 `g` 指针保存在当前 goroutine 的一个叫做 TLS 对象中。

第一步：我们先获取到 TLS 对象；

第二步：再从 TLS 中获取 goroutine 结构的 `g` 指针；

第三步：再从 `g` 指针中取出 goroutine id。

需要注意的是，不同 Go 版本的 goroutine 的结构可能不同，所以需要根据 Go 的 [不同版本](#) 进行调整。当然了，如果想要搞清楚各个版本的 goroutine 结构差异，所涉及的内容

又过于底层而且复杂，学习成本太高。怎么办呢？我们可以重点关注一些库。我们没有必要重复发明轮子，直接使用第三方的库来获取 goroutine id 就可以了。

好消息是现在已经有成熟的方法了，可以支持多个 Go 版本的 goroutine id，给你推荐一个常用的库：[🔗petermattis/goid](https://github.com/petermattis/goid)。

知道了如何获取 goroutine id，接下来就是最后的关键一步了，我们实现一个可以使用的可重入锁：

[📄 复制代码](#)

```
1 // RecursiveMutex 包装一个Mutex,实现可重入
2 type RecursiveMutex struct {
3     sync.Mutex
4     owner      int64 // 当前持有锁的goroutine id
5     recursion  int32 // 这个goroutine 重入的次数
6 }
7
8 func (m *RecursiveMutex) Lock() {
9     gid := goid.Get()
10    // 如果当前持有锁的goroutine就是这次调用的goroutine,说明是重入
11    if atomic.LoadInt64(&m.owner) == gid {
12        m.recursion++
13        return
14    }
15    m.Mutex.Lock()
16    // 获得锁的goroutine第一次调用,记录下它的goroutine id,调用次数加1
17    atomic.StoreInt64(&m.owner, gid)
18    m.recursion = 1
19 }
20
21 func (m *RecursiveMutex) Unlock() {
22     gid := goid.Get()
23     // 非持有锁的goroutine尝试释放锁, 错误的使用
24     if atomic.LoadInt64(&m.owner) != gid {
25         panic(fmt.Sprintf("wrong the owner(%d): %d!", m.owner, gid))
26     }
27     // 调用次数减1
28     m.recursion--
29     if m.recursion != 0 { // 如果这个goroutine还没有完全释放, 则直接返回
30         return
31     }
32     // 此goroutine最后一次调用, 需要释放锁
33     atomic.StoreInt64(&m.owner, -1)
34     m.Mutex.Unlock()
35 }
```

上面这段代码你可以拿来即用。我们一起来看下这个实现，真是非常巧妙，它相当于给 Mutex 打一个补丁，解决了记录锁的持有者的问题。可以看到，我们用 owner 字段，记录当前锁的拥有者 goroutine 的 id；recursion 是辅助字段，用于记录重入的次数。

有一点，我要提醒你一句，尽管拥有者可以多次调用 Lock，但是也必须调用相同次数的 Unlock，这样才能把锁释放掉。这是一个合理的设计，可以保证 Lock 和 Unlock 一一对应。

方案二：token

方案一是用 goroutine id 做 goroutine 的标识，我们也可以让 goroutine 自己来提供标识。不管怎么说，Go 开发者不期望你利用 goroutine id 做一些不确定的东西，所以，他们没有暴露获取 goroutine id 的方法。

下面的代码是第二种方案。调用者自己提供一个 token，获取锁的时候把这个 token 传入，释放锁的时候也需要把这个 token 传入。通过用户传入的 token 替换方案一中 goroutine id，其它逻辑和方案一一致。

[复制代码](#)

```
1 // Token方式的递归锁
2 type TokenRecursiveMutex struct {
3     sync.Mutex
4     token    int64
5     recursion int32
6 }
7
8 // 请求锁，需要传入token
9 func (m *TokenRecursiveMutex) Lock(token int64) {
10     if atomic.LoadInt64(&m.token) == token { //如果传入的token和持有锁的token一致，
11         m.recursion++
12         return
13     }
14     m.Mutex.Lock() // 传入的token不一致，说明不是递归调用
15     // 抢到锁之后记录这个token
16     atomic.StoreInt64(&m.token, token)
17     m.recursion = 1
18 }
19
20 // 释放锁
21 func (m *TokenRecursiveMutex) Unlock(token int64) {
22     if atomic.LoadInt64(&m.token) != token { // 释放其它token持有的锁
23         panic(fmt.Sprintf("wrong the owner(%d): %d!", m.token, token))
24     }
```

```
24     }
25     m.recursion-- // 当前持有这个锁的token释放锁
26     if m.recursion != 0 { // 还没有回退到最初的递归调用
27         return
28     }
29     atomic.StoreInt64(&m.token, 0) // 没有递归调用了，释放锁
30     m.Mutex.Unlock()
31 }
32
```

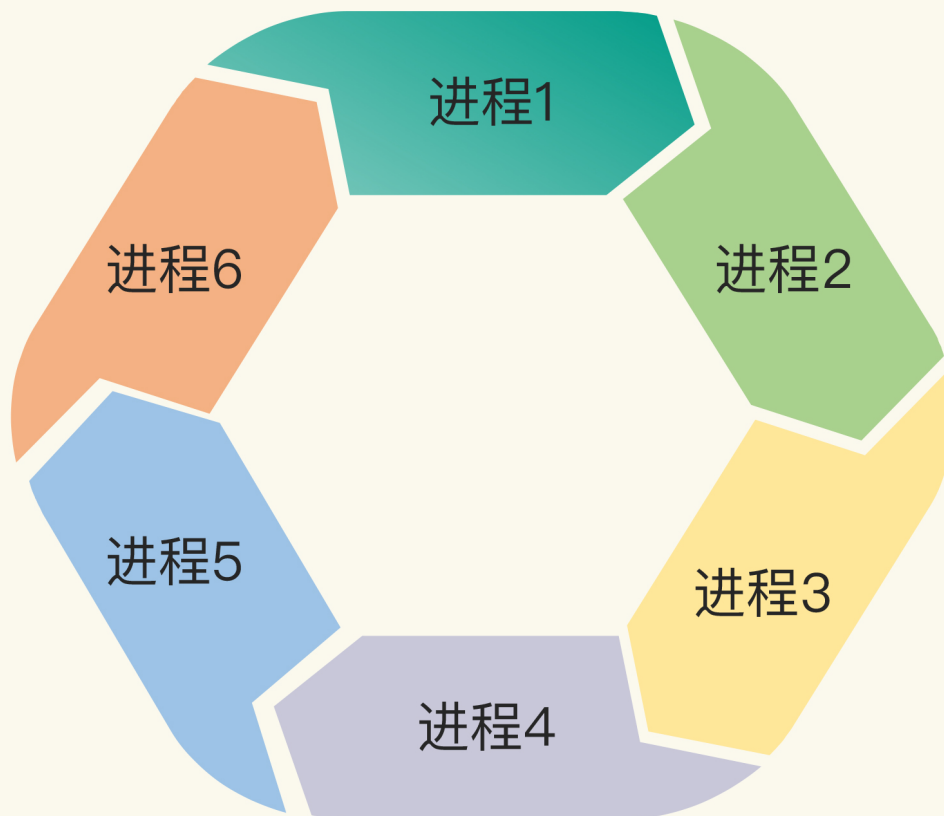
死锁

接下来，我们来看第四种错误场景：死锁。

我先解释下什么是死锁。两个或两个以上的进程（或线程，goroutine）在执行过程中，因争夺共享资源而处于一种互相等待的状态，如果没有外部干涉，它们都将无法推进下去，此时，我们称系统处于死锁状态或系统产生了死锁。

我们来分析一下死锁产生的必要条件。如果你想避免死锁，只要破坏这四个条件中的一个或者几个，就可以了。

1. **互斥**：至少一个资源是被排他性独享的，其他线程必须处于等待状态，直到资源被释放。
2. **持有和等待**：goroutine 持有一个资源，并且还在请求其它 goroutine 持有的资源，也就是咱们常说的“吃着碗里，看着锅里”的意思。
3. **不可剥夺**：资源只能由持有它的 goroutine 来释放。
4. **环路等待**：一般来说，存在一组等待进程， $P=\{P_1, P_2, \dots, P_N\}$ ， P_1 等待 P_2 持有的资源， P_2 等待 P_3 持有的资源，依此类推，最后是 P_N 等待 P_1 持有的资源，这就形成了一个环路等待的死结。



你看，死锁问题还真是挺有意思的，所以有很多人研究这个事儿。一个经典的死锁问题就是 [哲学家就餐问题](#)，我不做介绍了，你可以点击链接进一步了解。其实，死锁问题在现实生活中也比比皆是。

举个例子。有一次我去派出所开证明，派出所要求物业先证明我是本物业的业主，但是，物业要我提供派出所的证明，才能给我开物业证明，结果就陷入了死锁状态。你可以把派出所和物业看成两个 goroutine，派出所证明和物业证明是两个资源，双方都持有自己的资源而要求对方的资源，而且自己的资源自己持有，不可剥夺。

这是一个最简单的只有两个 goroutine 相互等待的死锁的例子，转化成代码如下：

 复制代码

```
1 package main
2
3
4 import (
5     "fmt"
6     "sync"
7
```

```
8     "time"
9 )
10
11
12 func main() {
13     // 派出所证明
14     var psCertificate sync.Mutex
15     // 物业证明
16     var propertyCertificate sync.Mutex
17
18
19     var wg sync.WaitGroup
20     wg.Add(2) // 需要派出所和物业都处理
21
22
23     // 派出所处理goroutine
24     go func() {
25         defer wg.Done() // 派出所处理完成
26
27
28         psCertificate.Lock()
29         defer psCertificate.Unlock()
30
31
32         // 检查材料
33         time.Sleep(5 * time.Second)
34         // 请求物业的证明
35         propertyCertificate.Lock()
36         propertyCertificate.Unlock()
37     }()
38
39
40     // 物业处理goroutine
41     go func() {
42         defer wg.Done() // 物业处理完成
43
44
45         propertyCertificate.Lock()
46         defer propertyCertificate.Unlock()
47
48
49         // 检查材料
50         time.Sleep(5 * time.Second)
51         // 请求派出所的证明
52         psCertificate.Lock()
53         psCertificate.Unlock()
54     }()
55
56
57     wg.Wait()
58     fmt.Println("成功完成")
59 }
```


这个程序没有办法运行成功，因为派出所的处理和物业的处理是一个环路等待的死结。

```
smallnest 1.3mutex4
smallnest 1.3mutex4 go run deadlock.go
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc00012e028)
    /usr/local/go/src/runtime/sema.go:56 +0x42
sync.(*WaitGroup).Wait(0xc00012e020)
    /usr/local/go/src/sync/waitgroup.go:130 +0x64
main.main()
    /mnt/d/gopath/src/github.com/smallnest/jike-go-sync/1.mutex/1.3mutex4/deadlock.go:46 +0xfb

goroutine 18 [semacquire]:
sync.runtime_SemacquireMutex(0xc00012e01c, 0x4c7900, 0x1)
    /usr/local/go/src/runtime/sema.go:71 +0x47
```

Go 运行时，有死锁探测的功能，能够检查出是否出现了死锁的情况，如果出现了，这个时候你就需要调整策略来处理了。

你可以引入一个第三方的锁，大家都依赖这个锁进行业务处理，比如现在政府推行的一站式政务服务中心。或者是解决持有等待问题，物业不需要看到派出所的证明才给开物业证明，等等。

好了，到这里，我给你讲了使用 Mutex 常见的 4 类问题。你是不是觉得，哎呀，这几类问题也太不应该了吧，真的会有人犯这么基础的错误吗？

还真是有。虽然 Mutex 使用起来很简单，但是，仍然可能出现使用错误的问题。而且，就连一些经验丰富的开发人员，也会出现一些 Mutex 使用的问题。接下来，我就带你围观几个非常流行的 Go 开发项目，看看这些错误是怎么产生和修复的。

流行的 Go 开发项目踩坑记

Docker

Docker 容器是一个开源的应用容器引擎，开发者可以以统一的方式，把他们的应用和依赖包打包到一个可移植的容器中，然后发布到任何安装了 docker 引擎的服务器上。

Docker 是使用 Go 开发的，也算是 Go 的一个杀手级产品了，它的 Mutex 相关的 Bug 也不少，我们来看几个典型的 Bug。

issue 36114

↑	@@ -26,9 +26,13 @@ func (c *Cluster) Init(req types.InitRequest) (string, error) {
26	26	defer c.controlMutex.Unlock()
27	27	if c.nr != nil {
28	28	if req.ForceNewCluster {
29	+	
29	30	// Take c.mu temporarily to wait for presently running
30	31	// API handlers to finish before shutting down the node.
31	32	c.mu.Lock()
33	+	if !c.nr.nodeState.IsManager() {
34	+	return "", errSwarmNotManager
35	+	}
32	36	c.mu.Unlock()
33	37	
34	38	if err := c.nr.Stop(); err != nil {
....	↓	

在第 34 行，节点发现不满足条件就返回了，但是，c.mu 这个锁没有释放！为什么会出现这个问题呢？其实，这是在重构或者添加新功能的时候经常犯的一个错误，因为不太了解上下文，或者没有仔细看函数的逻辑，从而导致锁没有被释放。现在的 Docker 当然已经没有这个问题了。

```

32          // Take c.mu temporarily to wait for presently running
33          // API handlers to finish before shutting down the node.
34          c.mu.Lock()
35          if !c.nr.nodeState.IsManager() {
36              c.mu.Unlock()
37              return "", errSwarmNotManager
38          }
39          c.mu.Unlock()
40

```

这样的 issue 还有很多，我就不一一列举了。我给你推荐几个关于 Mutex 的 issue 或者 pull request，你可以关注一下，分别是 36840、37583、35517、35482、33305、32826、30696、29554、29191、28912、26507 等。

Kubernetes

issue 72361

issue 72361 增加 Mutex 为了保护资源。这是为了解决 data race 问题而做的一个修复，修复方法也很简单，使用互斥锁即可，这也是我们解决 data race 时常用的方法。

		@@ -34,6 +35,7 @@ import (
34	35	type runner struct {
35	36	exec utilexec.Interface
36	37	ipvsHandle *libipvs.Handle
38	+	mu sync.Mutex // Protect Netlink calls
37	39	}
38	40	
39	41	// Protocol is the IPVS service protocol type
		@@ -58,6 +60,8 @@ func (runner *runner) AddVirtualServer(vs *VirtualServer) error {
58	60	if err != nil {
59	61	return err
60	62	}
63	+	runner.mu.Lock()
64	+	defer runner.mu.Unlock()
61	65	return runner.ipvsHandle.NewService(svc)
62	66	}
63	67	

issue 45192

🔗 [issue 45192](#) 也是一个返回时忘记 Unlock 的典型例子，和 docker issue 34881 犯的的错误都是一样的。

两大知名项目的开发者都犯了这个错误，所以，你就可以知道，引入这个 Bug 是多么容易，记住晁老师这句话：**保证 Lock/Unlock 成对出现，尽可能采用 defer mutex.Unlock 的方式，把它们成对、紧凑地写在一起。**

		@@ -40,6 +40,7 @@ type instanceInfo struct {
40	40	// GetZone returns the Zone containing the current failure zone and locality region that the program is running in
41	41	func (az *Cloud) GetZone() (cloudprovider.Zone, error) {
42	42	faultMutex.Lock()
43	+	defer faultMutex.Unlock()
43	44	if faultDomain == nil {
44	45	var err error
45	46	faultDomain, err = fetchFaultDomain()
46	47	if err != nil {
47	48	return cloudprovider.Zone{}, err
48	49	}
49	50	}
50	51	zone := cloudprovider.Zone{
51	52	FailureDomain: *faultDomain,
52	53	Region: az.Location,
53	54	}
54	+ -	faultMutex.Unlock()
55	55	return zone, nil
56	56	}
57	57	

除了这些，我也建议你关注一下其它的 Mutex 相关的 issue，比如 71617、70605 等。

gRPC

gRPC 是 Google 发起的一个开源远程过程调用（Remote procedure call）系统。该系统基于 HTTP/2 协议传输，使用 Protocol Buffers 作为接口描述语言。它提供 Go 语言的实现。

即使是 Google 官方出品的系统，也有一些 Mutex 的 issue。

issue 795

🔗 [issue 795](#) 是一个你可能想不到的 bug，那就是将 Unlock 误写成了 Lock。

↑	@@ -798,7 +798,7 @@ func (s *Server) Stop() {
798	798 func (s *Server) GracefulStop() {
799	799 s.mu.Lock()
800	800 if s.drain == true s.conns == nil {
801	- s.mu.Lock()
801	+ s.mu.Unlock()
802	802 return
803	803 }
804	804 s.drain = true
↓	

关于这个项目，还有一些其他的为了保护共享资源而添加 Mutex 的 issue，比如 1318、2074、2542 等。

etcd

etcd 是一个非常知名的分布式一致性的 key-value 存储技术，被用来做配置共享和服务发现。

issue 10419

🔗 [issue 10419](#) 是一个锁重入导致的问题。Store 方法内对请求了锁，而调用的 Compact 的方法内又请求了锁，这个时候，会导致死锁，一直等待，解决办法就是提供不需要加锁的 Compact 方法。


```
@@ -278,6 +280,29 @@ func (s *store) Compact(rev int64) (<-chan struct{}, error) {
278 280         return ch, nil
279 281     }
280 282
283 + func (s *store) compactLockfree(rev int64) (<-chan struct{}, error) {
284 +     ch, err := s.updateCompactRev(rev)
285 +     if nil != err {
286 +         return ch, err
287 +     }
288 +
289 +     return s.compact(rev)
290 + }
291 +
292 + func (s *store) Compact(rev int64) (<-chan struct{}, error) {
293 +     s.mu.Lock()
294 +
```

总结

这节课，我们学习了 Mutex 的一些易错场景，而且，我们还分析了流行的 Go 开源项目的错误，我也给你分享了我自己在开发中的经验总结。需要强调的是，**手误和重入导致的死锁，是最常见的使用 Mutex 的 Bug。**

Go 死锁探测工具只能探测整个程序是否因为死锁而冻结了，不能检测出一组 goroutine 死锁导致的某一块业务冻结的情况。你还可以通过 Go 运行时自带的死锁检测工具，或者是第三方的工具（比如 [go-deadlock](#)、[go-tools](#)）进行检查，这样可以尽早发现一些死锁的问题。不过，有些时候，死锁在某些特定情况下才会被触发，所以，如果你的测试或者短时间的运行没问题，不代表程序一定不会有死锁问题。

并发程序最难跟踪调试的就是很难重现，因为并发问题不是按照我们指定的顺序执行的，由于计算机调度的问题和事件触发的时机不同，死锁的 Bug 可能会在极端的情况下出现。通过搜索日志、查看日志，我们能够知道程序有异常了，比如某个流程一直没有结束。这个时候，可以通过 Go pprof 工具分析，它提供了一个 block profiler 监控阻塞的 goroutine。除此之外，我们还可以查看全部的 goroutine 的堆栈信息，通过它，你可以查看阻塞的 goroutine 究竟阻塞在哪一行哪一个对象上了。

思考题

查找知名的数据库系统 TiDB 的 issue，看看有没有 Mutex 相关的 issue，看看它们都是哪些相关的 Bug。

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 02 | Mutex: 庖丁解牛看实现

下一篇 04 | Mutex: 骇客编程，如何拓展额外功能？

精选留言 (18)

写留言



Junes

2020-10-16

分享一个我觉得很有项目借鉴意义的PR吧：

<https://github.com/pingcap/tidb/pull/20381/files>

这个问题是在当前的函数中Lock，然后在调用的函数中Unlock。这种方式会导致，如果...
展开 ▾



11



打奥特曼的小怪兽

2020-10-16

这个课程看起来很有意思

展开 ▾



5



橙子888

2020-10-16

在 TiDB Pull requests 已经 closed 的搜索 deadlock 关键字发现好多，例如 <https://github.com/pingcap/tidb/pull/500>，问题的原因在于释放的位置有问题。

展开 ▾



1

4



罗杰

2020-10-19

简单易懂 列举知名开源项目漏洞来对应自己的总结 太棒了👍



2



Remember九离

2020-10-18

第三课代码整理:https://github.com/wuqinqiang/Go_Concurrency/tree/main/class_3

作者回复: 赞。其他读者可以关注这位朋友的整理



1

2



roseduan

2020-10-17

将 Unlock 误写成了 Lock，哈哈哈，原来这些大佬也会犯低级错误



1



Stony.修行僧

2020-10-16

活捉一只大佬，写的真好，感觉编程语言里面锁技术是精髓之一



1

**buckwheat**

2020-10-16

看了一眼tidb关于mutex的issue，发现大部问题都出现在Unlock的时机上面，尤其是涉及到多个锁的时候，把Lock和Unlock放到两个方法里面就非常容易出现这种情况。tidb出现data race的issue要比dead lock的要多的多。老师，业务复杂时，在涉及到链式加锁时有没有什么好的办法避免死锁呢？

展开 ∨



1



1

**pony**

2020-10-24

老师讲解的很仔细，对mutex使用错误场景都列举了

补充点：Go语言核心36讲的解锁一个未加锁的mutex 导致的panic，无法被recover()捕获

展开 ∨

**niceshot**

2020-10-21

可重入锁到底有什么作用呢？

展开 ∨

**虫子樱桃**

2020-10-21

跟这个很类似 <https://medium.com/@bytecraze.com/recursive-locking-in-go-9c1c2a106a38>

**Jasper**

2020-10-21

感觉大佬们犯的错误都是我会犯的，哈哈哈。老师讲的简单易懂，注释也很全面。加油加油。

**gitxuzan**

2020-10-21

有个地方不明白，为什么源码里面需要用atomic 原子操作和直接赋值有什么区别

作者回复: 这个可以等atomic那一讲出来再了解



1



Panda
2020-10-20

保证 Lock/Unlock 成对出现
展开 ▾



小龙虾
2020-10-19

打卡
展开 ▾



Bug? Feature!
2020-10-19

老师讲的真好，跟着老师一路打怪~
展开 ▾



星亦辰
2020-10-16

A->B->C->A 典型的死锁了
展开 ▾



橙子888
2020-10-16

更新地好快，上一讲的源码还没消化完，新的一讲又出了.....

作者回复: 又看到你打卡了

