



下载APP



04 | Mutex: 骇客编程，如何拓展额外功能？

2020-10-19 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述：安晓辉**

时长 11:25 大小 10.47M



你好，我是鸟窝。

前面三讲，我们学习了互斥锁 Mutex 的基本用法、实现原理以及易错场景，可以说是涵盖了互斥锁的方方面面。如果你能熟练掌握这些内容，那么，在大多数的开发场景中，你都可以得心应手。

但是，在一些特定的场景中，这些基础功能是不足以应对的。这个时候，我们就需要开发一些扩展功能了。我来举几个例子。



比如说，我们知道，如果互斥锁被某个 goroutine 获取了，而且还没有释放，那么，其他请求这把锁的 goroutine，就会阻塞等待，直到有机会获得这把锁。有时候阻塞并不是一

个很好的主意，比如你请求锁更新一个计数器，如果获取不到锁的话没必要等待，大不了这次不更新，我下次更新就好了，如果阻塞的话会导致业务处理能力的下降。

再比如，如果我们要监控锁的竞争情况，一个监控指标就是，等待这把锁的 goroutine 数量。我们可以把这个指标推送到时间序列数据库中，再通过一些监控系统（比如 Grafana）展示出来。要知道，**锁是性能下降的“罪魁祸首”之一，所以，有效地降低锁的竞争，就能够很好地提高性能。因此，监控关键互斥锁上等待的 goroutine 的数量，是我们分析锁竞争的激烈程度的一个重要指标。**

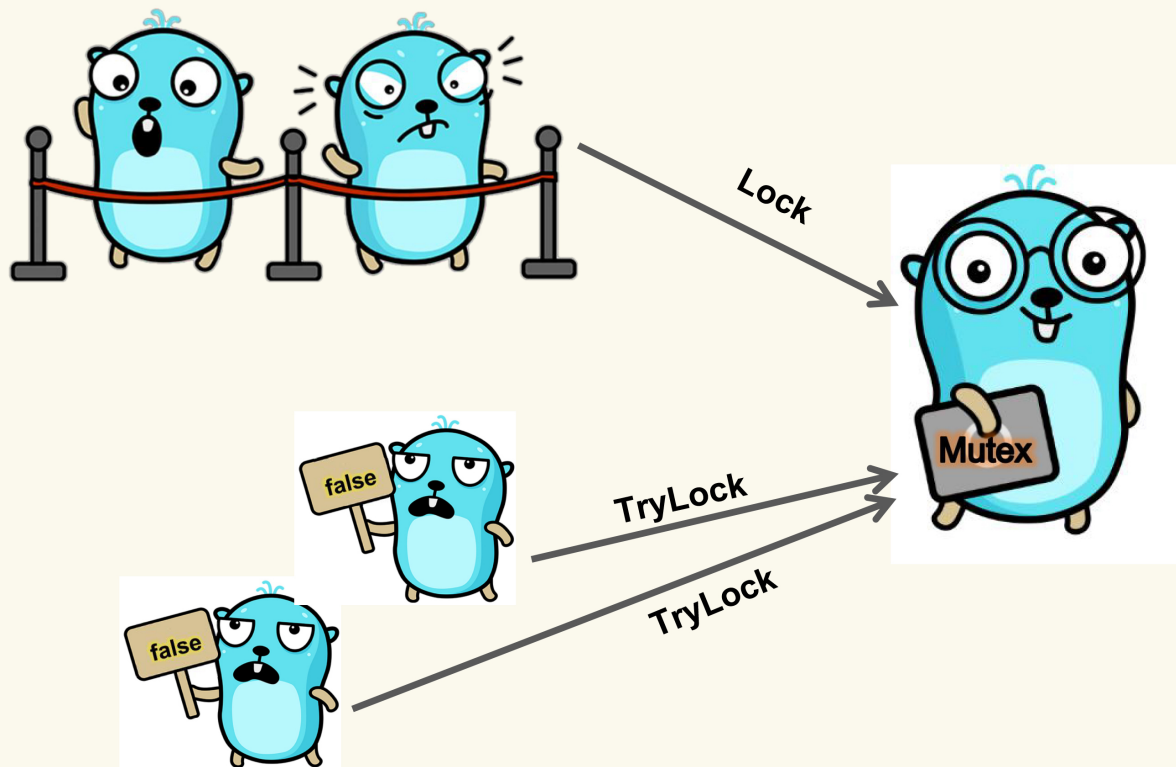
实际上，不论是不希望锁的 goroutine 继续等待，还是想监控锁，我们都可以基于标准库中 Mutex 的实现，通过 Hacker 的方式，为 Mutex 增加一些额外的功能。这节课，我就来教你实现几个扩展功能，包括实现 TryLock，获取等待者的数量等指标，以及实现一个线程安全的队列。

TryLock

我们可以为 Mutex 添加一个 TryLock 的方法，也就是尝试获取排外锁。

这个方法具体是什么意思呢？我来解释一下这里的逻辑。当一个 goroutine 调用这个 TryLock 方法请求锁的时候，如果这把锁没有被其他 goroutine 所持有，那么，这个 goroutine 就持有了这把锁，并返回 true；如果这把锁已经被其他 goroutine 所持有，或者是正在准备交给某个被唤醒的 goroutine，那么，这个请求锁的 goroutine 就直接返回 false，不会阻塞在方法调用上。

如下图所示，如果 Mutex 已经被一个 goroutine 持有，调用 Lock 的 goroutine 阻塞排队等待，调用 TryLock 的 goroutine 直接得到一个 false 返回。



在实际开发中，如果要更新配置数据，我们通常需要加锁，这样可以避免同时有多个 goroutine 并发修改数据。有的时候，我们也会使用 TryLock。这样一来，当某个 goroutine 想要更改配置数据时，如果发现已经有 goroutine 在更改了，其他的 goroutine 调用 TryLock，返回了 false，这个 goroutine 就会放弃更改。

很多语言（比如 Java）都为锁提供了 TryLock 的方法，但是，Go 官方 [issue 6123](#) 有一个讨论（后来一些 issue 中也提到过），标准库的 Mutex 不会添加 TryLock 方法。虽然通过 Go 的 Channel 我们也可以实现 TryLock 的功能，但是基于 Channel 的实现我们会放在 Channel 那一讲中去介绍，这一次我们还是基于 Mutex 去实现，毕竟大部分的程序员还是熟悉传统的同步原语，而且传统的同步原语也不容易出错。所以这节课，还是希望你掌握基于 Mutex 实现的方法。

那怎么实现一个扩展 TryLock 方法的 Mutex 呢？我们直接来看代码。

复制代码

```
1 // 复制Mutex定义的常量
2 const (
3     mutexLocked = 1 << iota // 加锁标识位置
4     mutexWoken             // 唤醒标识位置
5     mutexStarving          // 锁饥饿标识位置
6     mutexWaiterShift = iota // 标识waiter的起始bit位置
```

```
7 )
8
9 // 扩展一个Mutex结构
10 type Mutex struct {
11     sync.Mutex
12 }
13
14 // 尝试获取锁
15 func (m *Mutex) TryLock() bool {
16     // 如果能成功抢到锁
17     if atomic.CompareAndSwapInt32((*int32)(unsafe.Pointer(&m.Mutex)), 0, mutex
18         return true
19     }
20
21     // 如果处于唤醒、加锁或者饥饿状态，这次请求就不参与竞争了，返回false
22     old := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))
23     if old & (mutexLocked | mutexStarving | mutexWoken) != 0 {
24         return false
25     }
26
27     // 尝试在竞争的状态下请求锁
28     new := old | mutexLocked
29     return atomic.CompareAndSwapInt32((*int32)(unsafe.Pointer(&m.Mutex)), old,
30 }
```


第 17 行是一个 fast path，如果幸运，没有其他 goroutine 争这把锁，那么，这把锁就会被这个请求的 goroutine 获取，直接返回。

如果锁已经被其他 goroutine 所持有，或者被其他唤醒的 goroutine 准备持有，那么，就直接返回 false，不再请求，代码逻辑在第 23 行。

如果没有被持有，也没有其它唤醒的 goroutine 来竞争锁，锁也不处于饥饿状态，就尝试获取这把锁（第 29 行），不论是否成功都将结果返回。因为，这个时候，可能还有其他的 goroutine 也在竞争这把锁，所以，不能保证成功获取这把锁。

我们可以写一个简单的测试程序，来测试我们的 TryLock 的机制是否工作。

这个测试程序的工作机制是这样子的：程序运行时启动一个 goroutine 持有这把我们自己实现的锁，经过随机的时间才释放。主 goroutine 会尝试获取这把锁。如果前一个 goroutine 一秒内释放了这把锁，那么，主 goroutine 就有可能获取到这把锁了，输出 “got the lock”，否则没有获取到也不会被阻塞，会直接输出 “can't get the lock”。


 复制代码

```
1 func try() {
2     var mu Mutex
3     go func() { // 启动一个goroutine持有一段时间的锁
4         mu.Lock()
5         time.Sleep(time.Duration(rand.Intn(2)) * time.Second)
6         mu.Unlock()
7     }()
8
9     time.Sleep(time.Second)
10
11    ok := mu.TryLock() // 尝试获取到锁
12    if ok { // 获取成功
13        fmt.Println("got the lock")
14        // do something
15        mu.Unlock()
16        return
17    }
18
19    // 没有获取到
20    fmt.Println("can't get the lock")
21 }
```

获取等待者的数量等指标

接下来，我想和你聊聊怎么获取等待者数量等指标。

第二讲中，我们已经学习了 Mutex 的结构。先来回顾一下 Mutex 的数据结构，如下面的代码所示。它包含两个字段，state 和 sema。前四个字节（int32）就是 state 字段。

 复制代码

```
1 type Mutex struct {
2     state int32
3     sema  uint32
4 }
```

Mutex 结构中的 state 字段有很多个含义，通过 state 字段，你可以知道锁是否已经被某个 goroutine 持有、当前是否处于饥饿状态、是否有等待的 goroutine 被唤醒、等待者的数量等信息。但是，state 这个字段并没有暴露出来，所以，我们需要想办法获取到这个字段，并进行解析。

怎么获取未暴露的字段呢？很简单，我们可以通过 `unsafe` 的方式实现。我来举一个例子，你一看就明白了。

[复制代码](#)

```
1  const (  
2      mutexLocked = 1 << iota // mutex is locked  
3      mutexWoken  
4      mutexStarving  
5      mutexWaiterShift = iota  
6  )  
7  
8  type Mutex struct {  
9      sync.Mutex  
10 }  
11  
12 func (m *Mutex) Count() int {  
13     // 获取state字段的值  
14     v := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))  
15     v = v >> mutexWaiterShift //得到等待者的数值  
16     v = v + (v & mutexLocked) //再加上锁持有者的数量，0或者1  
17     return int(v)  
18 }
```

这个例子的第 14 行通过 `unsafe` 操作，我们可以得到 `state` 字段的值。第 15 行我们右移三位（这里的常量 `mutexWaiterShift` 的值为 3），就得到了当前等待者的数量。如果当前的锁已经被其他 `goroutine` 持有，那么，我们就稍微调整一下这个值，加上一个 1（第 16 行），你基本上可以把它看作是当前持有和等待这把锁的 `goroutine` 的总数。

`state` 这个字段的第一位是用来标记锁是否被持有，第二位用来标记是否已经唤醒了一个等待者，第三位标记锁是否处于饥饿状态，通过分析这个 `state` 字段我们就可以得到这些状态信息。我们可以为这些状态提供查询的方法，这样就可以实时地知道锁的状态了。

[复制代码](#)

```
1  // 锁是否被持有  
2  func (m *Mutex) IsLocked() bool {  
3      state := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))  
4      return state&mutexLocked == mutexLocked  
5  }  
6  
7  // 是否有等待者被唤醒  
8  func (m *Mutex) IsWoken() bool {  
9      state := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))  
10     return state&mutexWoken == mutexWoken
```



```
11 }
12
13 // 锁是否处于饥饿状态
14 func (m *Mutex) IsStarving() bool {
15     state := atomic.LoadInt32((*int32)(unsafe.Pointer(&m.Mutex)))
16     return state&mutexStarving == mutexStarving
17 }
```

我们可以写一个程序测试一下，比如，在 1000 个 goroutine 并发访问的情况下，我们可以把锁的状态信息输出出来：

```
1 func count() {
2     var mu Mutex
3     for i := 0; i < 1000; i++ { // 启动1000个goroutine
4         go func() {
5             mu.Lock()
6             time.Sleep(time.Second)
7             mu.Unlock()
8         }()
9     }
10
11     time.Sleep(time.Second)
12     // 输出锁的信息
13     fmt.Printf("waitings: %d, isLocked: %t, woken: %t, starving: %t\n", mu.Co
14 }
```

[复制代码](#)


有一点你需要注意一下，在获取 state 字段的时候，并没有通过 Lock 获取这把锁，所以获取的这个 state 的值是一个瞬态的值，可能在你解析出这个字段之后，锁的状态已经发生了变化。不过没关系，因为你查看的就是调用的那一时刻的锁的状态。

使用 Mutex 实现一个线程安全的队列

最后，我们来讨论一下，如何使用 Mutex 实现一个线程安全的队列。

为什么要讨论这个话题呢？因为 Mutex 经常会和其他非线程安全（对于 Go 来说，我们其实指的是 goroutine 安全）的数据结构一起，组合成一个线程安全的数据结构。新数据结构的业务逻辑由原来的数据结构提供，而 **Mutex 提供了锁的机制，来保证线程安全。**

比如队列，我们可以通过 Slice 来实现，但是通过 Slice 实现的队列不是线程安全的，出队（Dequeue）和入队（Enqueue）会有 data race 的问题。这个时候，Mutex 就要隆重出场了，通过它，我们可以在出队和入队的时候加上锁的保护。

 复制代码

```
1 type SliceQueue struct {
2     data []interface{}
3     mu    sync.Mutex
4 }
5
6 func NewSliceQueue(n int) (q *SliceQueue) {
7     return &SliceQueue{data: make([]interface{}, 0, n)}
8 }
9
10 // Enqueue 把值放在队尾
11 func (q *SliceQueue) Enqueue(v interface{}) {
12     q.mu.Lock()
13     q.data = append(q.data, v)
14     q.mu.Unlock()
15 }
16
17 // Dequeue 移去队头并返回
18 func (q *SliceQueue) Dequeue() interface{} {
19     q.mu.Lock()
20     if len(q.data) == 0 {
21         q.mu.Unlock()
22         return nil
23     }
24     v := q.data[0]
25     q.data = q.data[1:]
26     q.mu.Unlock()
27     return v
28 }
```

因为标准库中没有线程安全的队列数据结构的实现，所以，你可以通过 Mutex 实现一个简单的队列。通过 Mutex 我们就可以为一个非线程安全的 data interface{} 实现线程安全的访问。

总结

好了，我们来做个总结。

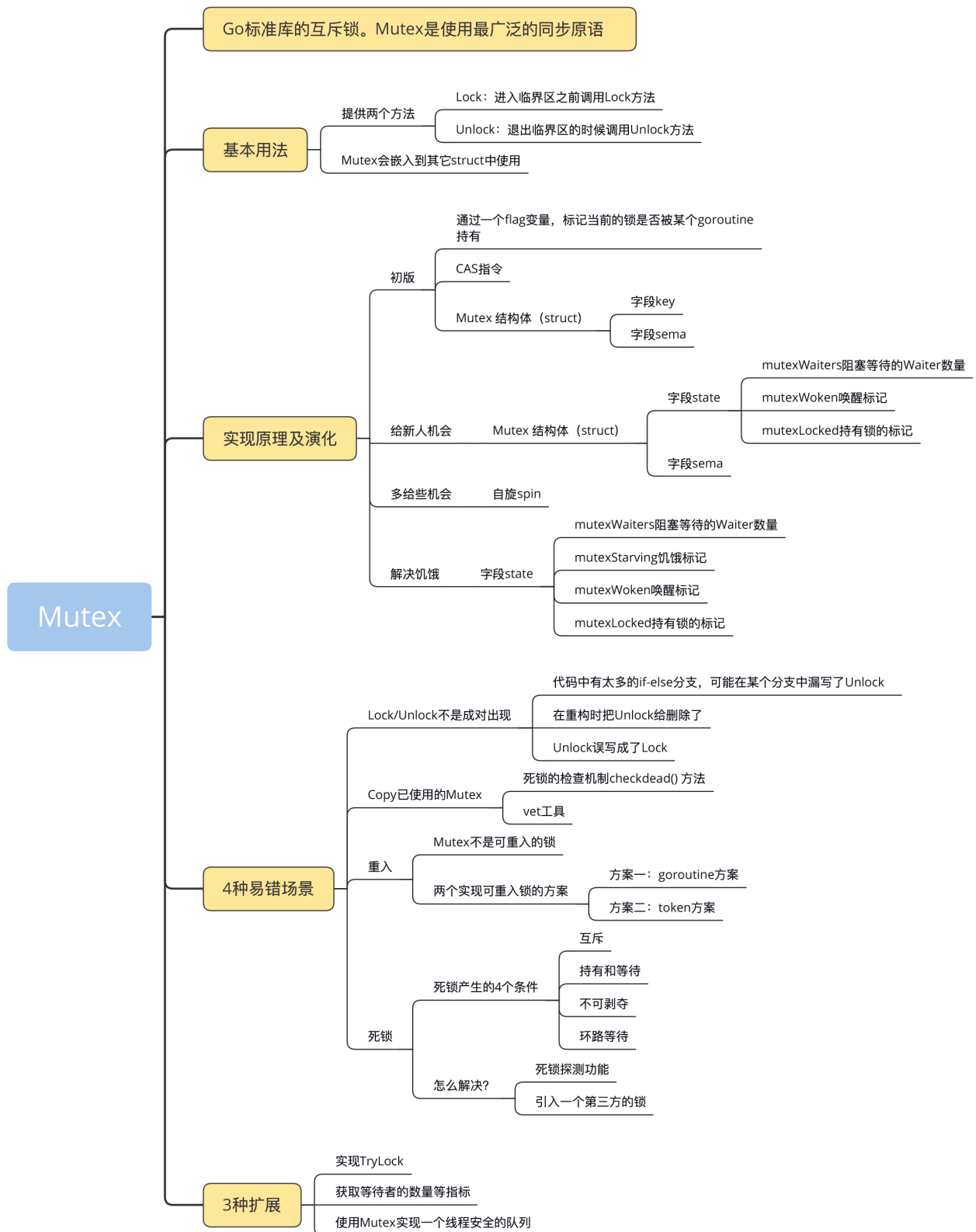
Mutex 是 package sync 的基石，其他的一些同步原语也是基于它实现的，所以，我们“隆重”地用了四讲来深度学习它。学到后面，你一定能感受到，多花些时间来完全掌握 Mutex 是值得的。

今天这一讲我和你分享了几个 Mutex 的拓展功能，这些方法是不是给你带来了一种“骇客”的编程体验呢，通过 Hacker 的方式，我们真的可以让 Mutex 变得更强大。

我们学习了基于 Mutex 实现 TryLock，通过 unsafe 的方式读取到 Mutex 内部的 state 字段，这样，我们就解决了开篇列举的问题，一是不希望锁的 goroutine 继续等待，一是想监控锁。

另外，使用 Mutex 组合成更丰富的数据结构是我们常见的场景，今天我们就实现了一个线程安全的队列，未来我们还会讲到实现线程安全的 map 对象。

到这里，Mutex 我们就系统学习完了，最后给你总结了一张 Mutex 知识地图，帮你复习一下。



思考题

你可以为 Mutex 获取锁时加上 Timeout 机制吗？会有什么问题吗？

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 03 | Mutex: 4种易错场景大盘点

下一篇 05 | RWMutex: 读写锁的实现原理及避坑指南

精选留言 (18)

写留言



18

2020-10-19

我变秃了，也变强了

展开 ∨

**Junes**

2020-10-19

我来提供个思路~

最简单直接的是采用channel实现, 用select监听锁和timeout两个channel, 不在今天的讨论范围内。

...

展开 ∨

**+1day**

2020-10-23

老师您好, 在获取等待者数量的代码中
如果要加上锁持有者的数量的话, 为什么不是
`v = v >> mutexWaiterShift + (v & mutexLocked)`

而是

`v = v >> mutexWaiterShift` //得到等待者的数值...

展开 ∨

作者回复: 你说的对

**Panmax**

2020-10-24

如果底层 Mutex 的 state 在某个版本中含义变了, 上边写的 TryLock 和监控锁的一些方法就会失效, 所以这样做是不是比较危险。

展开 ∨

作者回复: 是的.这只是hack方式, 和go的版本有关系。

**橙子888**

2020-10-19

认真消化完前三章后看今天的这章感觉容易多了。

展开 ∨



**moooofly**

2020-10-22

> 可以为 Mutex 获取锁时加上 Timeout 机制吗？会有什么问题吗？

我想，问题的关键不在于能不能加，因为加是肯定可以加的，关键应该是有了超时之后，会导致 unlock 的逻辑变得复杂，容易导致多次 unlock 的问题

展开 ∨

**约书亚**

2020-10-22

对tryLock中提到的第一行被称作fast path不太理解，是节省了一次LoadInt32么？

**Chen**

2020-10-22

第 15 行我们右移三位（这里的常量 mutexWaiterShift 的值为 3），就得到了当前等待者的数量

=>> 这里看不懂，为什么右移三位=》得到等待者数量

展开 ∨

作者回复: 看图，左边三位是其它标志

**罗帮奎**

2020-10-21

感觉可以把timeout分小段，每过一小段时间尝试拿锁，要不然一直卡在自旋拿锁会很耗cpu

**Linuxer**

2020-10-21

通过tryLock->sleep->tryLock存在的问题是，可能sleep过程中就能获取到锁，但是不得不等待设定的时长，如果间隔一段时间就尝试，有可能实际拿不到锁而浪费CPU



**Linuxer21**

课后思考题 是不是可以通过tryLock尝试获取锁获取失败就sleep一段时间，超时后再调用tryLock

**新味道**

2020-10-21

Channel 是用memory access synchronization来构建的吗？

**Panda**

2020-10-20

Hacker 编程 自己动手 丰衣足食 打卡 催更 哈哈
内容很棒 深入学习 Go 并发编程

**Alexdown**

2020-10-20

TryLock方法中27行表示的是：

- 1) 在锁的瞬时状态为正常模式+无唤醒的等待者+锁未被持有时，当前goroutine与等待队列中队头goroutine一起竞争（此时等待者队列不为空）。
- 2) 与TryLock的fast path一样，锁的瞬时状态为正常模式+无唤醒的等待者+锁未被持有+等待队列为空。此时无竞争，直接获取到锁。...

展开 ∨

**Alexdown**

2020-10-20

- 1) 『获取等待者的数量等指标』小节，『第 15 行我们左移三位（这里的常量 mutexWaiterShift 的值为 3）』应该是右移三位。
- 2) 在now ~ now+timeout内，间隔重试调用TryLock

作者回复: 多谢

**linxs**

2020-10-20

TryLock方法内，对于这段代码有点不理解，为什么要把&m.Mutex转换成*int32，这里

的话我直接用&m.Mutex.state是否是一样的

```
if atomic.CompareAndSwapInt32((*int32)(unsafe.Pointer(&m.Mutex)), 0, mutexLocked) { ...
```

展开

作者回复: 你能访问到？

2



Ev
2020-10-20

请教一个基础问题，为啥 (*int32)(unsafe.Pointer(&m.Mutex)) 可以获取sync.Mutex中s
tate的值，Mutex结构中不是还有sema吗？

展开

作者回复: 只取第一个用

1



(͡° ͜ʖ ͡°)
2020-10-19

可以通过Context.WithTimeout进行超时机制添加 也可以通过select time.After配合使用