



下载APP



34 | 并发：如何使用共享变量？

2022-01-14 Tony Bai

《Tony Bai · Go语言第一课》

课程介绍 >



讲述：Tony Bai

时长 17:22 大小 15.92M



你好，我是 Tony Bai。

在前面的讲解中，我们学习了 Go 的并发实现方案，知道了 Go 基于 Tony Hoare 的 **CSP 并发模型** 理论，实现了 Goroutine、channel 等并发原语。

并且，Go 语言之父 Rob Pike 还有一句经典名言：“不要通过共享内存来通信，应该通过通信来共享内存（Don't communicate by sharing memory, share memory by communicating）”，这就奠定了 Go 应用并发设计的主流风格：**使用 channel 进行不同 Goroutine 间的通信。**

领资料



不过，Go 也并没有彻底放弃基于共享内存的并发模型，而是在提供 CSP 并发模型原语的同时，还通过标准库的 sync 包，提供了针对传统的、基于共享内存并发模型的低级同步原

语，包括：互斥锁（`sync.Mutex`）、读写锁（`sync.RWMutex`）、条件变量（`sync.Cond`）等，并通过 `atomic` 包提供了原子操作原语等等。显然，基于共享内存的并发模型在 Go 语言中依然有它的“用武之地”。

所以，在并发的最后一讲，我们就围绕 `sync` 包中的几个同步结构与对应的方法，聊聊基于共享内存的并发模型在 Go 中的应用。

我们先来看看在哪些场景下，我们需要用到 `sync` 包提供的低级同步原语。


sync 包低级同步原语可以用在哪？

这里我要先强调一句，一般情况下，我建议你优先使用 CSP 并发模型进行并发程序设计。但是在下面一些场景中，我们依然需要 `sync` 包提供的低级同步原语。

首先是需要高性能的临界区（critical section）同步机制场景。

在 Go 中，channel 并发原语也可以用于对数据对象访问的同步，我们可以把 channel 看成是一种高级的同步原语，它自身的实现也是建构在低级同步原语之上的。也正因为如此，channel 自身的性能与低级同步原语相比要略微逊色，开销要更大。

这里，关于 `sync.Mutex` 和 channel 各自实现的临界区同步机制，我做了一个简单的性能基准测试对比，通过对比结果，我们可以很容易看出两者的性能差异：

 复制代码

```
1 var cs = 0 // 模拟临界区要保护的数据
2 var mu sync.Mutex
3 var c = make(chan struct{}, 1)
4
5 func criticalSectionSyncByMutex() {
6     mu.Lock()
7     cs++
8     mu.Unlock()
9 }
10
11 func criticalSectionSyncByChan() {
12     c <- struct{}{}
13     cs++
14     <-c
15 }
16
```

```
17 func BenchmarkCriticalSectionSyncByMutex(b *testing.B) {
18     for n := 0; n < b.N; n++ {
19         criticalSectionSyncByMutex()
20     }
21 }
22
23 func BenchmarkCriticalSectionSyncByMutexInParallel(b *testing.B) {
24     b.RunParallel(func(pb *testing.PB) {
25         for pb.Next() {
26             criticalSectionSyncByMutex()
27         }
28     })
29 }
30
31 func BenchmarkCriticalSectionSyncByChan(b *testing.B) {
32     for n := 0; n < b.N; n++ {
33         criticalSectionSyncByChan()
34     }
35 }
36
37 func BenchmarkCriticalSectionSyncByChanInParallel(b *testing.B) {
38     b.RunParallel(func(pb *testing.PB) {
39         for pb.Next() {
40             criticalSectionSyncByChan()
41         }
42     })
43 }
```

运行这个对比测试（Go 1.17），我们得到：

 复制代码

```
1 $go test -bench .
2 goos: darwin
3 goarch: amd64
4 ... ..
5 BenchmarkCriticalSectionSyncByMutex-8          88083549      13.64 ns
6 BenchmarkCriticalSectionSyncByMutexInParallel-8 22337848      55.29 ns
7 BenchmarkCriticalSectionSyncByChan-8           28172056      42.48 ns
8 BenchmarkCriticalSectionSyncByChanInParallel-8 5722972       208.1 ns/
9 PASS
```

通过这个对比实验，我们可以看到，无论是在单 Goroutine 情况下，还是在并发测试情况下，sync.Mutex实现的同步机制的性能，都要比 channel 实现的高出三倍多。

因此，通常在需要高性能的临界区（critical section）同步机制的情况下，sync 包提供的低级同步原语更为适合。

第二种就是不想转移结构体对象所有权，但又要保证结构体内部状态数据的同步访问的场景。


基于 channel 的并发设计，有一个特点：在 Goroutine 间通过 channel 转移数据对象的所有权。所以，只有拥有数据对象所有权（从 channel 接收到该数据）的 Goroutine 才可以对该数据对象进行状态变更。

如果你的设计中没有转移结构体对象所有权，但又要保证结构体内部状态数据在多个 Goroutine 之间同步访问，那么你可以使用 sync 包提供的低级同步原语来实现，比如最常用的 sync.Mutex。

了解了这些应用场景之后，接着我们就来看看如何使用 sync 包中的各个同步结构，不过在使用之前，我们需要先看看一个 sync 包中同步原语使用的注意事项。

sync 包中同步原语使用的注意事项

在 sync 包的注释中（在 \$GOROOT/src/sync/mutex.go 文件的头部注释），我们看到这样一行说明：

 复制代码

```
1 // Values containing the types defined in this package should not be copied.
```

翻译过来就是：“不应复制那些包含了此包中类型的值”。

在 sync 包的其他源文件中，我们同样看到类似的一些注释：

 复制代码

```
1
2 // $GOROOT/src/sync/mutex.go
3 // A Mutex must not be copied after first use. (禁止复制首次使用后的Mutex)
4
5 // $GOROOT/src/sync/rwmutex.go
6 // A RWMutex must not be copied after first use. (禁止复制首次使用后的RWMutex)
```

```
7 // $GOROOT/src/sync/cond.go
8 // A Cond must not be copied after first use. (禁止复制首次使用后的Cond)
9 ... ..
10
```

那么，为什么首次使用 Mutex 等 sync 包中定义的结构类型后，我们不应该再对它们进行复制操作呢？我们以 Mutex 这个同步原语为例，看看它的实现是怎样的。

Go 标准库中 sync.Mutex 的定义是这样的：

[复制代码](#)

```
1 // $GOROOT/src/sync/mutex.go
2 type Mutex struct {
3     state int32
4     sema  uint32
5 }
```

我们看到，Mutex 的定义非常简单，由两个整型字段 state 和 sema 组成：

state：表示当前互斥锁的状态；

sema：用于控制锁状态的信号量。

初始情况下，Mutex 的实例处于 **Unlocked** 状态（state 和 sema 均为 0）。对 Mutex 实例的复制也就是两个整型字段的复制。一旦发生复制，原变量与副本就是两个单独的内存块，各自发挥同步作用，互相就没有了关联。

如果发生复制后，你仍然认为原变量与副本保护的是同一个数据对象，那可就大错特错了。我们来看一个例子：

[复制代码](#)

```
1 func main() {
2     var wg sync.WaitGroup
3     i := 0
4     var mu sync.Mutex // 负责对i的同步访问
5
6     wg.Add(1)
7     // g1
8     go func(mu1 sync.Mutex) {
9         mu1.Lock()
```

```
10         i = 10
11         time.Sleep(10 * time.Second)
12         fmt.Printf("g1: i = %d\n", i)
13         mu1.Unlock()
14         wg.Done()
15     }(mu)
16
17     time.Sleep(time.Second)
18
19     mu.Lock()
20     i = 1
21     fmt.Printf("g0: i = %d\n", i)
22     mu.Unlock()
23
24     wg.Wait()
25 }
```

在这个例子中，我们使用一个 `sync.Mutex` 类型变量 `mu` 来同步对整型变量 `i` 的访问。我们创建一个新 Goroutine：`g1`，`g1` 通过函数参数得到 `mu` 的一份拷贝 `mu1`，然后 `g1` 会通过 `mu1` 来同步对整型变量 `i` 的访问。

那么，`g0` 通过 `mu` 和 `g1` 通过 `mu` 的拷贝 `mu1`，是否能实现对同一个变量 `i` 的同步访问呢？我们来看看运行这个示例的运行结果：

```
1 g0: i = 1
2 g1: i = 1
```

[复制代码](#)

从结果来看，这个程序并没有实现对 `i` 的同步访问，第 9 行 `g1` 对 `mu1` 的加锁操作，并没能阻塞第 19 行 `g0` 对 `mu` 的加锁。于是，`g1` 刚刚将 `i` 赋值为 10 后，`g0` 就又将 `i` 赋值为 1 了。

出现这种结果的原因就是我们前面分析的情况，一旦 `Mutex` 类型变量被拷贝，原变量与副本就各自发挥作用，互相没有关联了。甚至，如果拷贝的时机不对，比如在一个 `mutex` 处于 `locked` 的状态时对它进行了拷贝，就会对副本进行加锁操作，将导致加锁的 Goroutine 永远阻塞下去。

通过前面这个例子，我们可以很直观地看到：如果对使用过的、`sync` 包中的类型的示例进行复制，并使用了复制后得到的副本，将导致不可预期的结果。所以，在使用 `sync` 包中的

类型的时候，我们推荐通过**闭包**方式，或者是**传递类型实例（或包裹该类型的类型实例）的地址（指针）**的方式进行。这就是使用 sync 包时最值得我们注意的事项。

接下来，我们就来逐个分析日常使用较多的 sync 包中同步原语。我们先来看看互斥锁与读写锁。

互斥锁（Mutex）还是读写锁（RWMutex）？

sync 包提供了两种用于临界区同步的原语：互斥锁（Mutex）和读写锁（RWMutex）。它们都是零值可用的数据类型，也就是不需要显式初始化就可以使用，并且使用方法都比较简单。在上面的示例中，我们已经看到了 Mutex 的应用方法，这里再总结一下：

[复制代码](#)

```
1 var mu sync.Mutex
2 mu.Lock()    // 加锁
3 doSomething()
4 mu.Unlock()  // 解锁
```

一旦某个 Goroutine 调用的 Mutex 执行 Lock 操作成功，它将成功持有这把互斥锁。这个时候，如果有其他 Goroutine 执行 Lock 操作，就会阻塞在这把互斥锁上，直到持有这把锁的 Goroutine 调用 Unlock 释放掉这把锁后，才会抢到这把锁的持有权并进入临界区。

由此，我们也可以得到使用互斥锁的两个原则：

尽量减少在锁中的操作。这可以减少其他因 Goroutine 阻塞而带来的损耗与延迟。

一定要记得调用 Unlock 解锁。忘记解锁会导致程序局部死锁，甚至是整个程序死锁，会导致严重的后果。同时，我们也可以结合第 23 讲学习到的 defer，优雅地执行解锁操作。

读写锁与互斥锁用法大致相同，只不过多了一组加读锁和解读锁的方法：

[复制代码](#)

```
1 var rwmu sync.RWMutex
2 rwmu.RLock() //加读锁
3 readSomething()
```



```
4  rwmu.RUnlock() //解读锁
5  rwmu.Lock()    //加写锁
6  changeSomething()
7  rwmu.Unlock()  //解写锁
```

写锁与 Mutex 的行为十分类似，一旦某 Goroutine 持有写锁，其他 Goroutine 无论是尝试加读锁，还是加写锁，都会被阻塞在写锁上。

但读锁就宽松多了，一旦某个 Goroutine 持有读锁，它不会阻塞其他尝试加读锁的 Goroutine，但加写锁的 Goroutine 依然会被阻塞住。

通常，**互斥锁（Mutex）是临时区同步原语的首选**，它常被用来对结构体对象的内部状态、缓存等进行保护，是使用最为广泛的临界区同步原语。相比之下，读写锁的应用就没那么广泛了，只活跃于它擅长的场景下。

那读写锁（RWMutex）究竟擅长在哪种场景下呢？我们先来看一组基准测试：

[复制代码](#)

```
1  var cs1 = 0 // 模拟临界区要保护的数据
2  var mu1 sync.Mutex
3
4  var cs2 = 0 // 模拟临界区要保护的数据
5  var mu2 sync.RWMutex
6
7  func BenchmarkWriteSyncByMutex(b *testing.B) {
8      b.RunParallel(func(pb *testing.PB) {
9          for pb.Next() {
10             mu1.Lock()
11             cs1++
12             mu1.Unlock()
13         }
14     })
15 }
16
17 func BenchmarkReadSyncByMutex(b *testing.B) {
18     b.RunParallel(func(pb *testing.PB) {
19         for pb.Next() {
20             mu1.Lock()
21             _ = cs1
22             mu1.Unlock()
23         }
24     })
25 }
26
```



```

27 func BenchmarkReadSyncByRWMutex(b *testing.B) {
28     b.RunParallel(func(pb *testing.PB) {
29         for pb.Next() {
30             mu2.RLock()
31             _ = cs2
32             mu2.RUnlock()
33         }
34     })
35 }
36
37 func BenchmarkWriteSyncByRWMutex(b *testing.B) {
38     b.RunParallel(func(pb *testing.PB) {
39         for pb.Next() {
40             mu2.Lock()
41             cs2++
42             mu2.Unlock()
43         }
44     })
45 }

```

这些基准测试都是并发测试，度量的是 Mutex、RWMutex 在并发下的读写性能。我们分别在 cpu=2、8、16、32 的情况下运行这个并发性能测试，测试结果如下：

[复制代码](#)

```

1 goos: darwin
2 goarch: amd64
3 ... ..
4 BenchmarkWriteSyncByMutex-2      73423770      16.12 ns/op
5 BenchmarkReadSyncByMutex-2      84031135      15.08 ns/op
6 BenchmarkReadSyncByRWMutex-2    37182219      31.87 ns/op
7 BenchmarkWriteSyncByRWMutex-2   40727782      29.08 ns/op
8
9 BenchmarkWriteSyncByMutex-8      22153354      56.39 ns/op
10 BenchmarkReadSyncByMutex-8      24164278      51.12 ns/op
11 BenchmarkReadSyncByRWMutex-8    38589122      31.17 ns/op
12 BenchmarkWriteSyncByRWMutex-8   18482208      65.27 ns/op
13
14 BenchmarkWriteSyncByMutex-16     20672842      62.94 ns/op
15 BenchmarkReadSyncByMutex-16     19247158      62.94 ns/op
16 BenchmarkReadSyncByRWMutex-16   29978614      39.98 ns/op
17 BenchmarkWriteSyncByRWMutex-16   16095952      78.19 ns/op
18
19 BenchmarkWriteSyncByMutex-32     20539290      60.20 ns/op
20 BenchmarkReadSyncByMutex-32     18807060      72.61 ns/op
21 BenchmarkReadSyncByRWMutex-32   29772936      40.45 ns/op
22 BenchmarkWriteSyncByRWMutex-32   13320544      86.53 ns/op

```

通过测试结果对比，我们得到了一些结论：

并发量较小的情况下，Mutex 性能最好；随着并发量增大，Mutex 的竞争激烈，导致加锁和解锁性能下降；

RWMutex 的读锁性能并没有随着并发量的增大，而发生较大变化，性能始终恒定在 40ns 左右；

在并发量较大的情况下，RWMutex 的写锁性能和 Mutex、RWMutex 读锁相比，是最差的，并且随着并发量增大，RWMutex 写锁性能有继续下降趋势。

由此，我们就可以看出，**读写锁适合应用在具有一定并发量且读多写少的场合**。在大量并发读的情况下，多个 Goroutine 可以同时持有读锁，从而减少在锁竞争中等待的时间。

而互斥锁，即便是读请求的场合，同一时刻也只能有一个 Goroutine 持有锁，其他 Goroutine 只能阻塞在加锁操作上等待被调度。


接下来，我们继续看条件变量 sync.Cond。

条件变量

sync.Cond 是传统的条件变量原语概念在 Go 语言中的实现。我们可以把一个条件变量理解为一个容器，这个容器中存放着一个或一组等待着某个条件成立的 Goroutine。当条件成立后，这些处于等待状态的 Goroutine 将得到通知，并被唤醒继续进行后续的工作。这与百米飞人大战赛场上，各位运动员等待裁判员的发令枪声的情形十分类似。

条件变量是同步原语的一种，如果没有条件变量，开发人员可能需要在 Goroutine 中通过连续轮询的方式，检查某条件是否为真，这种连续轮询非常消耗资源，因为 Goroutine 在这个过程中是处于活动状态的，但它的工作又没有进展。

这里我们先看一个用 sync.Mutex 实现对条件轮询等待的例子：

 复制代码


```
1 type signal struct{}
2
3 var ready bool
4
5 func worker(i int) {
```

```
6   fmt.Printf("worker %d: is working...\n", i)
7   time.Sleep(1 * time.Second)
8   fmt.Printf("worker %d: works done\n", i)
9 }
10
11 func spawnGroup(f func(i int), num int, mu *sync.Mutex) <-chan signal {
12     c := make(chan signal)
13     var wg sync.WaitGroup
14
15     for i := 0; i < num; i++ {
16         wg.Add(1)
17         go func(i int) {
18             for {
19                 mu.Lock()
20                 if !ready {
21                     mu.Unlock()
22                     time.Sleep(100 * time.Millisecond)
23                     continue
24                 }
25                 mu.Unlock()
26                 fmt.Printf("worker %d: start to work...\n", i)
27                 f(i)
28                 wg.Done()
29                 return
30             }
31         }(i + 1)
32     }
33
34     go func() {
35         wg.Wait()
36         c <- signal(struct{}{})
37     }()
38     return c
39 }
40
41 func main() {
42     fmt.Println("start a group of workers...")
43     mu := &sync.Mutex{}
44     c := spawnGroup(worker, 5, mu)
45
46     time.Sleep(5 * time.Second) // 模拟ready前的准备工作
47     fmt.Println("the group of workers start to work...")
48
49     mu.Lock()
50     ready = true
51     mu.Unlock()
52
53     <-c
54     fmt.Println("the group of workers work done!")
55 }
```

就像前面提到的，轮询的方式开销大，轮询间隔设置的不同，条件检查的及时性也会受到影响。

`sync.Cond`为 Goroutine 在这个场景下提供了另一种可选的、资源消耗更小、使用体验更佳同步方式。使用条件变量原语，我们可以在实现相同目标的同时，避免对条件的轮询。


我们用`sync.Cond`对上面的例子进行改造，改造后的代码如下：

 复制代码

```
1  type signal struct{}
2
3  var ready bool
4
5  func worker(i int) {
6      fmt.Printf("worker %d: is working...\n", i)
7      time.Sleep(1 * time.Second)
8      fmt.Printf("worker %d: works done\n", i)
9  }
10
11 func spawnGroup(f func(i int), num int, groupSignal *sync.Cond) <-chan signal
12     c := make(chan signal)
13     var wg sync.WaitGroup
14
15     for i := 0; i < num; i++ {
16         wg.Add(1)
17         go func(i int) {
18             groupSignal.L.Lock()
19             for !ready {
20                 groupSignal.Wait()
21             }
22             groupSignal.L.Unlock()
23             fmt.Printf("worker %d: start to work...\n", i)
24             f(i)
25             wg.Done()
26         }(i + 1)
27     }
28
29     go func() {
30         wg.Wait()
31         c <- signal(struct{}{})
32     }()
33     return c
34 }
35
36 func main() {
37     fmt.Println("start a group of workers...")
```

```
38  groupSignal := sync.NewCond(&sync.Mutex{})
39  c := spawnGroup(worker, 5, groupSignal)
40
41  time.Sleep(5 * time.Second) // 模拟ready前的准备工作
42  fmt.Println("the group of workers start to work...")
43
44  groupSignal.L.Lock()
45  ready = true
46  groupSignal.Broadcast()
47  groupSignal.L.Unlock()
48
49  <-c
50  fmt.Println("the group of workers work done!")
51 }
```

我们运行这个示例程序，得到：

 复制代码

```
1  start a group of workers...
2  the group of workers start to work...
3  worker 2: start to work...
4  worker 2: is working...
5  worker 3: start to work...
6  worker 3: is working...
7  worker 1: start to work...
8  worker 1: is working...
9  worker 4: start to work...
10 worker 5: start to work...
11 worker 5: is working...
12 worker 4: is working...
13 worker 4: works done
14 worker 2: works done
15 worker 3: works done
16 worker 1: works done
17 worker 5: works done
18 the group of workers work done!
```

我们看到，`sync.Cond`实例的初始化，需要一个满足实现了`sync.Locker`接口的类型实例，通常我们使用`sync.Mutex`。

条件变量需要这个互斥锁来同步临界区，保护用作条件的数据。加锁后，各个等待条件成立的 Goroutine 判断条件是否成立，如果不成立，则调用`sync.Cond`的 `Wait` 方法进入等待状态。`Wait` 方法在 Goroutine 挂起前会进行 `Unlock` 操作。

当 main goroutine 将 ready 置为 true，并调用 sync.Cond 的 Broadcast 方法后，各个阻塞的 Goroutine 将被唤醒，并从 Wait 方法中返回。Wait 方法返回前，Wait 方法会再次加锁让 Goroutine 进入临界区。接下来 Goroutine 会再次对条件数据进行判定，如果条件成立，就会解锁并进入下一个工作阶段；如果条件依旧不成立，那么会再次进入循环体，并调用 Wait 方法挂起等待。

和 sync.Mutex、sync.RWMutex 等相比，sync.Cond 应用的场景更为有限，只有在需要“等待某个条件成立”的场景下，Cond 才有用武之地。

其实，面向 CSP 并发模型的 channel 原语和面向传统共享内存并发模型的 sync 包提供的原语，已经能够满足 Go 语言应用并发设计中 ****99.9% 的并发同步需求了。而剩余那 0.1%**** 的需求，我们可以使用 Go 标准库提供的 atomic 包来实现。

原子操作 (atomic operations)

atomic 包是 Go 语言给用户提供的原子操作原语的相关接口。原子操作 (atomic operations) 是相对于普通指令操作而言的。

我们以一个整型变量自增的语句为例说明一下：

```
1 var a int
2 a++
```

[复制代码](#)

a++ 这行语句需要 3 条普通机器指令来完成变量 a 的自增：

LOAD：将变量从内存加载到 CPU 寄存器；

ADD：执行加法指令；

STORE：将结果存储回原内存地址中。

这 3 条普通指令在执行过程中是可以被中断的。而原子操作的指令是不可中断的，它就好比一个事务，要么不执行，一旦执行就一次性全部执行完毕，中间不可分割。也正因为如此，原子操作也可以被用于共享数据的并发同步。

原子操作由底层硬件直接提供支持，是一种硬件实现的指令级的“事务”，因此相对于操作系统层面和 Go 运行时层面提供的同步技术而言，它更为原始。

atomic 包封装了 CPU 实现的部分原子操作指令，为用户层提供体验良好的原子操作函数，因此 atomic 包中提供的原语更接近硬件底层，也更为低级，它也常被用于实现更为高级的并发同步技术，比如 channel 和 sync 包中的同步原语。

我们以 atomic.SwapInt64 函数在 x86_64 平台上的实现为例，看看这个函数的实现方法：

[复制代码](#)

```
1 // $GOROOT/src/sync/atomic/doc.go
2 func SwapInt64(addr *int64, new int64) (old int64)
3
4 // $GOROOT/src/sync/atomic/asm.s
5
6 TEXT ·SwapInt64(SB),NOSPLIT,$0
7     JMP     runtime/internal/atomic·Xchg64(SB)
8
9 // $GOROOT/src/runtime/internal/asm_amd64.s
10 TEXT runtime/internal/atomic·Xchg64(SB), NOSPLIT, $0-24
11     MOVQ    ptr+0(FP), BX
12     MOVQ    new+8(FP), AX
13     XCHGQ   AX, 0(BX)
14     MOVQ    AX, ret+16(FP)
15     RET
16
```


从函数 SwapInt64 的实现中，我们可以看到：它基本就是对 x86_64 CPU 实现的原子操作指令 XCHGQ 的直接封装。

原子操作的特性，让 atomic 包也可以被用作对共享数据的并发同步，那么和更为高级的 channel 以及 sync 包中原语相比，我们究竟该怎么选择呢？

我们先来看看 atomic 包提供了哪些能力。

atomic 包提供了两大类原子操作接口，一类是针对整型变量的，包括有符号整型、无符号整型以及对应的指针类型；另外一类是针对自定义类型的。因此，第一类原子操作接口的存在让 atomic 包天然适合去实现某一个共享整型变量的并发同步。

我们再看一个例子：

 复制代码


```
1 var n1 int64
2
3 func addSyncByAtomic(delta int64) int64 {
4     return atomic.AddInt64(&n1, delta)
5 }
6
7 func readSyncByAtomic() int64 {
8     return atomic.LoadInt64(&n1)
9 }
10
11 var n2 int64
12 var rwmu sync.RWMutex
13
14 func addSyncByRWMutex(delta int64) {
15     rwmu.Lock()
16     n2 += delta
17     rwmu.Unlock()
18 }
19
20 func readSyncByRWMutex() int64 {
21     var n int64
22     rwmu.RLock()
23     n = n2
24     rwmu.RUnlock()
25     return n
26 }
27
28 func BenchmarkAddSyncByAtomic(b *testing.B) {
29     b.RunParallel(func(pb *testing.PB) {
30         for pb.Next() {
31             addSyncByAtomic(1)
32         }
33     })
34 }
35
36 func BenchmarkReadSyncByAtomic(b *testing.B) {
37     b.RunParallel(func(pb *testing.PB) {
38         for pb.Next() {
39             readSyncByAtomic()
40         }
41     })
42 }
43
44 func BenchmarkAddSyncByRWMutex(b *testing.B) {
45     b.RunParallel(func(pb *testing.PB) {
46         for pb.Next() {
47             addSyncByRWMutex(1)
```

```

48     }
49     })
50 }
51
52 func BenchmarkReadSyncByRWMutex(b *testing.B) {
53     b.RunParallel(func(pb *testing.PB) {
54         for pb.Next() {
55             readSyncByRWMutex()
56         }
57     })
58 }
59

```

我们分别在 cpu=2、 8、 16、 32 的情况下运行上述性能基准测试，得到结果如下：

 复制代码

```

1 goos: darwin
2 goarch: amd64
3 ... ..
4 BenchmarkAddSyncByAtomic-2      75426774      17.69 ns/op
5 BenchmarkReadSyncByAtomic-2    1000000000    0.7437 ns/op
6 BenchmarkAddSyncByRWMutex-2    39041671      30.16 ns/op
7 BenchmarkReadSyncByRWMutex-2   41325093      28.48 ns/op
8
9 BenchmarkAddSyncByAtomic-8      77497987      15.25 ns/op
10 BenchmarkReadSyncByAtomic-8    1000000000    0.2395 ns/op
11 BenchmarkAddSyncByRWMutex-8    17702034      67.16 ns/op
12 BenchmarkReadSyncByRWMutex-8   29966182      40.37 ns/op
13
14 BenchmarkAddSyncByAtomic-16     57727968      20.39 ns/op
15 BenchmarkReadSyncByAtomic-16   1000000000    0.2536 ns/op
16 BenchmarkAddSyncByRWMutex-16   15029635      78.61 ns/op
17 BenchmarkReadSyncByRWMutex-16  29722464      40.28 ns/op
18
19 BenchmarkAddSyncByAtomic-32     58010497      20.40 ns/op
20 BenchmarkReadSyncByAtomic-32    1000000000    0.2402 ns/op
21 BenchmarkAddSyncByRWMutex-32   11748312      93.15 ns/op
22 BenchmarkReadSyncByRWMutex-32  29845912      40.54 ns/op

```

通过这个运行结果，我们可以得出一些结论：

读写锁的性能随着并发量增大的情况，与前面讲解的 sync.RWMutex 一致；

利用原子操作的无锁并发写的性能，随着并发量增大几乎保持恒定；

利用原子操作的无锁并发读的性能，随着并发量增大有持续提升的趋势，并且性能是读锁的约 200 倍。

通过这些结论，我们大致可以看到 atomic 原子操作的特性：随着并发量提升，使用 atomic 实现的**共享变量**的并发读写性能表现更为稳定，尤其是原子读操作，和 sync 包中的读写锁原语比起来，atomic 表现出了更好的伸缩性和高性能。

由此，我们也可以看出 atomic 包更适合**一些对性能十分敏感、并发量较大且读多写少的场合**。

不过，atomic 原子操作可用来同步的范围有较大限制，只能同步一个整型变量或自定义类型变量。如果我们要对一个复杂的临界区数据进行同步，那么首选的依旧是 sync 包中的原语。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

虽然 Go 推荐基于通信来共享内存的并发设计风格，但 Go 并没有彻底抛弃对基于共享内存并发模型的支持，Go 通过标准库的 sync 包以及 atomic 包提供了低级同步原语。这些原语有着它们自己的应用场景。

如果我们考虑使用低级同步原语，一般都是因为低级同步原语可以提供**更佳的性能表现**，性能基准测试结果告诉我们，使用低级同步原语的性能可以高出 channel 许多倍。在性能敏感的场景下，我们依然离不开这些低级同步原语。

在使用 sync 包提供的同步原语之前，我们一定要牢记这些原语使用的注意事项：**不要复制首次使用后的 Mutex/RWMutex/Cond 等**。一旦复制，你将很大可能得到意料之外的运行结果。

sync 包中的低级同步原语各有各的擅长领域，你可以记住：

在具有一定并发量且读多写少的场合使用 RWMutex；

在需要“等待某个条件成立”的场景下使用 Cond；

当你不确定使用什么原语时，那就使用 Mutex 吧。

如果你对同步的性能有极致要求，且并发量较大，读多写少，那么可以考虑一下 atomic 包提供的原子操作函数。

思考题


使用基于共享内存的并发模型时，最令人头疼的可能就是“死锁”问题的存在了。你了解死锁的产生条件么？能编写一个程序模拟一下死锁的发生么？

欢迎你把这节课分享给更多对 Go 并发感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta 订阅超级会员，你将得 50 元

Ta 单独订阅本课程，你将得 20 元

 生成海报并分享

 赞 0  提建议

© 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 33 | 并发：小 channel 中蕴含大智慧

更多学习推荐

2021 最新大厂 Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

限量免费领取 



精选留言 (1)

 写留言



罗杰 

2022-01-14

连续三节课都是需要花费 30 分钟才能阅读完，即使是复习也要超过 20 分钟，这几节的内容真的好充实。

共 2 条评论 >

 1