"原子操作 sync/atomic 2"

2020-12-25T19:52:27+08:00

Table of Contents

## 比较并交换和比较有什么不同

比较并交换操作即 CAS 操作，是有条件的交换操作，只有在条件满足的情况下才会进行交换操作。

所谓的交换指的是，把新值赋给变量，并返回变量的旧值。

在进行 CAS 操作时，函数会先判断被操作变量的当前值是否与我们预期的旧值相等。如果相等就把新值赋给变量，并返回 true 表明交换操作已进行，否则就忽略交换操作，返回 false。

CAS 操作并不是单一操作，而是一种操作组合。和其他的原子操作不同，它的用途更加广泛。例如，跟 for 语句联用可以实现自旋锁（spinlock）。

for {  
 if atomic.CompareAndSwapInt32(&num, 10, 0) {  
 fmt.Println("The number has gone to zero.")  
 break  
 }  
 time.Sleep(time.Millisecond \* 500)  
}

for 语句中的 CAS 操作不停的检查某个需要满足的条件，一旦满足条件就退出 for 循环。只要条件未被满足，当前的流程就一直阻塞在这里。

所以使用 CAS 操作的时候，我们还是要多加注意的，因为它可以被用来模仿锁，并有可能“阻塞”流程。

这里的效果与互斥锁类似，但使用场景不同。使用互斥锁时，假设共享资源的状态会被其他 goroutine 频繁的修改。而 for 语句加 CAS 操作假设的往往是共享资源的状态改变不频繁，或者总是会变成期望的那样。这是一种更加乐观或者更加宽松的做法。

## 变量的写已经是原子操作了，读还需要吗

是需要的，参照读写锁的读写互斥。读操作还没有完成时，写操作把值修改了，那面读到的值必然是错的。这显然破坏了数据的完整性。所以，一旦决定了要对共享资源进行保护，就要做到完全保护。不完全的保护和不保护基本没区别。

由于原子操作函数只支持非常有限的数据类型，所以在很多应用场景下互斥锁往往是更加合适的。如果是只涉及并发的读写单一整数类型的值，或者多个互不相关的整数类型值时，就不要在考虑互斥锁了。

因为原子操作函数的执行速度比互斥锁快得多。而且使用简单，不涉及临界区的选择以及死锁问题。

## 怎样用好 sync/atomic.Value

为了扩大原子操作的适用范围，Go 语言在 1.4 版本发布的时候向 sync/atomic 包中添加了一个新的类型 Value。此类型的值相当于一个容器，可以被用来原子的存储和加载任意的值。

atomic.Value 类型是开箱即用的，我们声明一个该类型的值之后就可以直接使用了。这个类型只有两个指针方法 Store 和 Load。

一旦我们使用 atomic.Value 类型来存储值，那面它就不应该再被复制了。因为该类型是结构类型是属于值类型的，复制该类型的值会产生一个完全分离的新值。这个新值和旧值完全没有关系，怎么改都不会影响旧值。

### 不能用原子值存储 nil

我们不能把 nil 作为参数值传入原子值的 Store 方法，否则就会引发一个 panic。

注意，如果有一个接口类型的变量，它的动态值是 nil，但动态类型却不是 nil，那么它的值就不等于 nil。这样一个变量的值是可以被存入原子值的。

### 我们向原子值存储的第一个值，决定了它今后能且只能存储哪一个类型的值

例如，我第一次向一个原子值存储了一个 string 类型的值，那我在后面就只能用该原子值来存储字符串了。如果我又想用它存储结构体，那么在调用它的 Store 方法的时候就会引发一个 panic。这个 panic 会告诉我，这次存储的值的类型与之前的不一致。

原子值内部是依据被存储值的实际类型来做判断的，即使是实现了同一个接口的不同类型，它们的值也不能被先后存储到同一个原子值中。

我们无法通过某个方法获知一个原子值是否已经被真正使用，并且，也没有办法通过常规的途径得到一个原子值可以存储值的实际类型。这使得我们误用原子值的可能性大大增加，尤其是在多个地方使用同一个原子值的时候。

### 使用建议

1. 不要把内部使用的原子值暴露给外界。比如，声明一个全局的原子变量并不是一个正确的做法。这个变量的访问权限最起码也应该是包级私有的。
2. 如果不得不让包外，或模块外的代码使用你的原子值，那么可以声明一个包级私有的原子变量，然后再通过一个或多个公开的函数，让外界间接地使用到它。注意，这种情况下不要把原子值传递到外界，不论是传递原子值本身还是它的指针值。
3. 如果通过某个函数可以向内部的原子值存储值的话，那么就应该在这个函数中先判断被存储值类型的合法性。若不合法，则应该直接返回对应的错误值，从而避免 panic 的发生。
4. 如果可能的话，我们可以把原子值封装到一个数据类型中，比如一个结构体类型。这样，我们既可以通过该类型的方法更加安全地存储值，又可以在该类型中包含可存储值的合法类型信息。

尽量不要向原子值中存储引用类型的值。因为这很容易造成安全漏洞。

var box6 atomic.Value  
v6 := []int{1, 2, 3}  
box6.Store(v6)  
v6[1] = 4 // 注意，此处的操作不是并发安全的！

把一个 []int 类型的切片值 v6, 存入了原子值 box6。注意，切片类型属于引用类型。所以，在外面改动这个切片值，就等于修改了 box6 中存储的那个值。这相当于绕过了原子值而进行了非并发安全的操作。

store := func(v []int) {  
 replica := make([]int, len(v))  
 copy(replica, v)  
 box6.Store(replica)  
}  
store(v6)  
v6[2] = 5 // 此处的操作是安全的。

应该先为切片值 v6 创建了一个完全的副本。这个副本涉及的数据已经与原值毫不相干了。然后，我再把这个副本存入 box6。如此一来，无论我再对 v6 的值做怎样的修改，都不会破坏 box6 提供的安全保护。