30 | 原子操作（下）

2018-10-19 郝林

你好，我是郝林，今天我们继续分享原子操作的内容。

我们接着上一篇文章的内容继续聊，上一篇我们提到了，sync/atomic包中的函数可以做的原子操作有：加法（add）、比较并交换（compare and swap，简称CAS）、加载（load）、存储（store）和交换（swap）。并且以此衍生出了两个问题。

今天我们继续来看**第三个衍生问题：** 比较并交换操作与交换操作相比有什么不同？优势在哪里？

回答是：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值的交换。

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操作的时候，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，它就把新值赋给该变量，并返回true以表明交换操作已进行；否则就忽略交换操作，并返回false。

可以看到，CAS操作并不是单一的操作，而是一种操作组合。这与其他的原子操作都不同。正因为如此，它的用途要更广泛一些。例如，我们将它与for语句联用就可以实现一种简易的自旋锁（spinlock）。

```
for {
 if atomic.CompareAndSwapInt32(&num2, 10, 0) {
  fmt.Println("The second number has gone to zero.")
  break
 }
 time.Sleep(time.Millisecond * 500)
}
```

在for语句中的CAS操作可以不停地检查某个需要满足的条件，一旦条件满足就退出for循环。这就相当于，只要条件未被满足，当前的流程就会被一直“阻塞”在这里。

这在效果上与互斥锁有些类似。不过，它们的适用场景是不同的。我们在使用互斥锁的时候，总是假设共享资源的状态会被其他的goroutine频繁地改变。

而for语句加CAS操作的假设往往是：共享资源状态的改变并不频繁，或者，它的状态总会变成期望的那样。这是一种更加乐观，或者说更加宽松的做法。

第四个衍生问题： 假设我已经保证了对一个变量的写操作都是原子操作，比如：加或减、存储、交换等等，那我对它进行读操作的时候，还有必要使用原子操作吗？

回答：很有必要。其中的道理你可以对照一下读写锁。为什么在读写锁保护下的写操作和读操作之间是互斥的？这是为了防止读操作读到没有被修改完的值，对吗？

如果写操作还没有进行完，读操作就来读了，那么就只能读到仅修改了一部分的值。这显然破坏了值的完整性，读出来的值也是完全错误的。

所以，一旦你决定了要对一个共享资源进行保护，那就要做到完全的保护。不完全的保护基本上与不保护没有什么区别。

好了，上面的主问题以及相关的衍生问题涉及了原子操作函数的用法、原理、对比和一些最佳实践，希望你已经理解了。

由于这里的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，互斥锁往往是更加适合的。

不过，一旦我们确定了在某个场景下可以使用原子操作函数，比如：只涉及并发地读写单一的整数类型值，或者多个互不相关的整数类型值，那就不要再考虑互斥锁了。

这主要是因为原子操作函数的执行速度要比互斥锁快得多。而且，它们使用起来更加简单，不会涉及临界区的选择，以及死锁等问题。当然了，在使用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。

知识扩展

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“原子地”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变量（以下简称原子变量）之后就可以直接使用了。这个类型使用起来很简单，它只有两个指针方法——Store和Load。不过，虽然简单，但还是有一些值得注意的地方的。

首先一点，一旦atomic.Value类型的值（以下简称原子值）被真正使用，它就不应该再被复制了。什么叫做“真正使用”呢？

我们只要用它来存储值了，就相当于开始真正使用了。atomic.Value类型属于结构体类型，而结构体类型属于值类型。

所以，复制该类型的值会产生一个完全分离的新值。这个新值相当于被复制的那个值的一个快照。之后，不论后者存储的值怎样改变，都不会影响到前者，反之亦然。

另外，关于用原子值来存储值，有两条强制性的使用规则。第一条规则，不能用原子值存储nil。也就是说，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。

这里要注意，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。我在前面讲接口的时候和你说明过这个问题。正因为如此，这样一个变量的值是可以被存入原子值的。

第二条规则，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。

你可能会想：我先存储一个接口类型的值，然后再存储这个接口的某个实现类型的值，这样是不是可以呢？

很可惜，这样是不可以的，同样会引发一个panic。因为原子值内部是依据被存储值的实际类型来做判断的。所以，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。

遗憾的是，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。

下面，我给你几条具体的使用建议。

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。

除了上述使用建议之外，我还要再特别强调一点：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。请看下面的代码：

```
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！
```

我把一个`[]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，我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。那么，应该怎样修补这个漏洞呢？可以这样做：

```
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。
```

我先为切片值`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。

以上，就是我要告诉你的关于atomic.Value的注意事项和使用建议。你可以在demo64.go文件中看到相应的示例。

总结

我们把这两篇文章一起总结一下。相对于原子操作函数，原子值类型的优势很明显，但它的使用规则也更多一些。首先，在首次真正使用后，原子值就不应该再被复制了。

其次，原子值的Store方法对其参数值（也就是被存储值）有两个强制的约束。一个约束是，参数值不能为nil。另一个约束是，参数值的类型不能与首个被存储值的类型不同。也就是说，一旦一个原子值存储了某个类型的值，那它以后就只能存储这个类型的值了。

基于上面这几个注意事项，我提出了几条使用建议，包括：不要对外暴露原子变量、不要传递原子值及其指针值、尽量不要在原子值中存储引用类型的值，等等。与之相关的一些解决方案我也一并提出了。希望你能够受用。

原子操作明显比互斥锁要更加轻便，但是限制也同样明显。所以，我们在进行二选一的时候通常不会太困难。但是原子值与互斥锁之间的选择有时候就需要仔细的考量了。不过，如果你能牢记我今天讲的这些内容的话，应该会有很大的助力。

思考题

今天的思考题只有一个，那就是：如果要对原子值和互斥锁进行二选一，你认为最重要的三个决策条件应该是什么？

戳此查看Go语言专栏文章配套详细代码。