"原子操作 sync/atomic"

2020-12-25T12:28:41+08:00

Table of Contents

## 原子性执行和原子操作

对于一个 Go 程序来说，Go 语言运行时系统中的调度器会恰当的安排其中所有的 goroutine 的运行。但是在同一时刻，只可能有少数的 G 真正处于运行状态，这个数量只会与 M（系统级线程）的数量一致，而不会随着 G 的增多而增长。

为了公平起见，调度器总会频繁的换上或换下这些 G。换上是说让一个 G 由非运行状态转为运行状态，并促使其中的代码在某个 CPU 上执行。换下则相反，让一个 G 中的代码终端执行，并由运行状态转为非原型状态。

这个终端的时机有很多，任何两条语句执行的间隙，甚至某条语句执行的过程中都是可以的。即使这些语句在临界区也是如此。所以，互斥锁虽然能保证临界区的代码串行执行，但却不能保证这些代码的原子性执行。

能够保证原子性执行的只有原子操作（atomic operation）。原子操作在运行过程中是不允许终端的。在底层，这会由 CPU 提供芯片级别的支持，所以绝对有效。即使在拥有多 CPU 核心或者多 CPU 的计算机系统中，原子操作的保证也是不可撼动的。

这使得原子操作可以完全的消除竞态条件，并能够觉得的保证并发安全性。并且，它的执行速度要比其他同步工具快的多，通常会高出好几个数量级。

它的缺点也是明显的。具体的说就是因为原子操作不能被中断，所以它需要足够简单并且快速。如果原子操作迟迟不能完成，而且又不会被中断，那面将会给计算机执行指令的效率带来巨大的影响。因此，操作系统层面只针对二进制位或整数的原子操作提供支持。

Go 语言的原子操作当然是基于 CPU 和操作系统的，所以它也只针对少数数据类型的值提供了原子操作函数。这些函数都存在于标准库代码包 sync/atomic 中。

## sync/atomic 包中提供的原子操作以及可操作的数据类型

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）。

这些函数针对的数据类型并不多，对这些类型中的每一个，sync/atomic 包都会有一套函数给予支持。数据类型有：int32、int64、uint32、uint64、uintptr 以及 unsafe 包中的 Pointer。不过针对 unsafe.Pointer 类型并没有提供加法原子操作的函数。此外该包还提供了一个名为 Value 的类型，可以被用来存储任意类型的值。

### 原子操作函数的第一个参数要求传指针

// AddInt32 atomically adds delta to \*addr and returns the new value.  
func AddInt32(addr \*int32, delta int32) (new int32)  
  
// CompareAndSwapInt32 executes the compare-and-swap operation for an int32 value.  
func CompareAndSwapInt32(addr \*int32, old, new int32) (swapped bool)  
  
// LoadInt32 atomically loads \*addr.  
func LoadInt32(addr \*int32) (val int32)  
  
// StoreInt32 atomically stores val into \*addr.  
func StoreInt32(addr \*int32, val int32)  
  
// SwapInt32 atomically stores new into \*addr and returns the previous \*addr value.  
func SwapInt32(addr \*int32, new int32) (old int32)

原子操作函数需要的是被操作值的指针，而不是这个值本身。被传入函数的参数值都会被复制，像这种基本类型的值一旦传入函数，就已经与函数外面那个值毫无关系了。

unsafe.Pointer 类型虽然本身就是指针类型，但是原子函数要操作的是指针值，而不是它指向的那个值，所以需要的仍然是指向这个指针值的指针。

只要原子操作函数拿到了被操作值的指针，就可以定位到存储该值的内存地址。然后才能通过底层的指令，准确的操作这个内存地址上的数据。

### 原子加法函数做原子减法

// AddInt32 atomically adds delta to \*addr and returns the new value.  
func AddInt32(addr \*int32, delta int32) (new int32)  
  
// AddUint32 atomically adds delta to \*addr and returns the new value.  
// To subtract a signed positive constant value c from x, do AddUint32(&x, ^uint32(c-1)).  
// In particular, to decrement x, do AddUint32(&x, ^uint32(0)).  
func AddUint32(addr \*uint32, delta uint32) (new uint32)  
  
// AddInt64 atomically adds delta to \*addr and returns the new value.  
func AddInt64(addr \*int64, delta int64) (new int64)  
  
// AddUint64 atomically adds delta to \*addr and returns the new value.  
// To subtract a signed positive constant value c from x, do AddUint64(&x, ^uint64(c-1)).  
// In particular, to decrement x, do AddUint64(&x, ^uint64(0)).  
func AddUint64(addr \*uint64, delta uint64) (new uint64)

atomic.AddInt32 和 atomic.AddInt64 函数的第二个参数代表差量，是有符号的数据类型，如果需要做原子减法，把这个差量设置为负整数就可以了。

atomic.AddUint32 和 atomic.AddUint64 函数做原子减法就不能这么直接。 因为他们的第二个参数都是无符号的，这里需要转换下。

比如想对 uint32 类型的被操作值 18 做原子减法，差量是 -3，那么可以先把差量转换为有符号的 int32 类型的值，然后再把该值的类型转换为 uint32，即：uint32(int32(-3))。

不过要注意，直接这样写会使 Go 语言的编译器报错，它会告诉你：“常量 -3 不在 uint32 类型可表示的范围内”，换句话说，这样做会让表达式的结果值溢出。

不过，如果我们先把 int32(-3) 的结果值赋给变量 delta，再把 delta 的值转换为 uint32 类型的值，就可以绕过编译器的检查并得到正确的结果了。

num := uint32(18)  
delta := int32(-3)  
ret := atomic.AddUint32(&num, uint32(delta))

还有一种方式更直接：

^uint32(-N-1))

其中 N 代表由负整数表示的差量。就是说，我们先把差量的绝对值减去 1，然后再把这个无类型的结果常量转为 uint32 类型的值，最后在这个值上做按位异或操作就可以获得最终的参数值了。

num := uint32(18)  
ret := atomic.AddUint32(&num, ^uint32(-(-3)-1))

简单来说，此表达式的结果值的补码，与使用前一种方法得到的值的补码相同，所以这两种方式是等价的。我们都知道，整数在计算机中是以补码的形式存在的，所以在这里，结果值的补码相同就意味着表达式的等价。