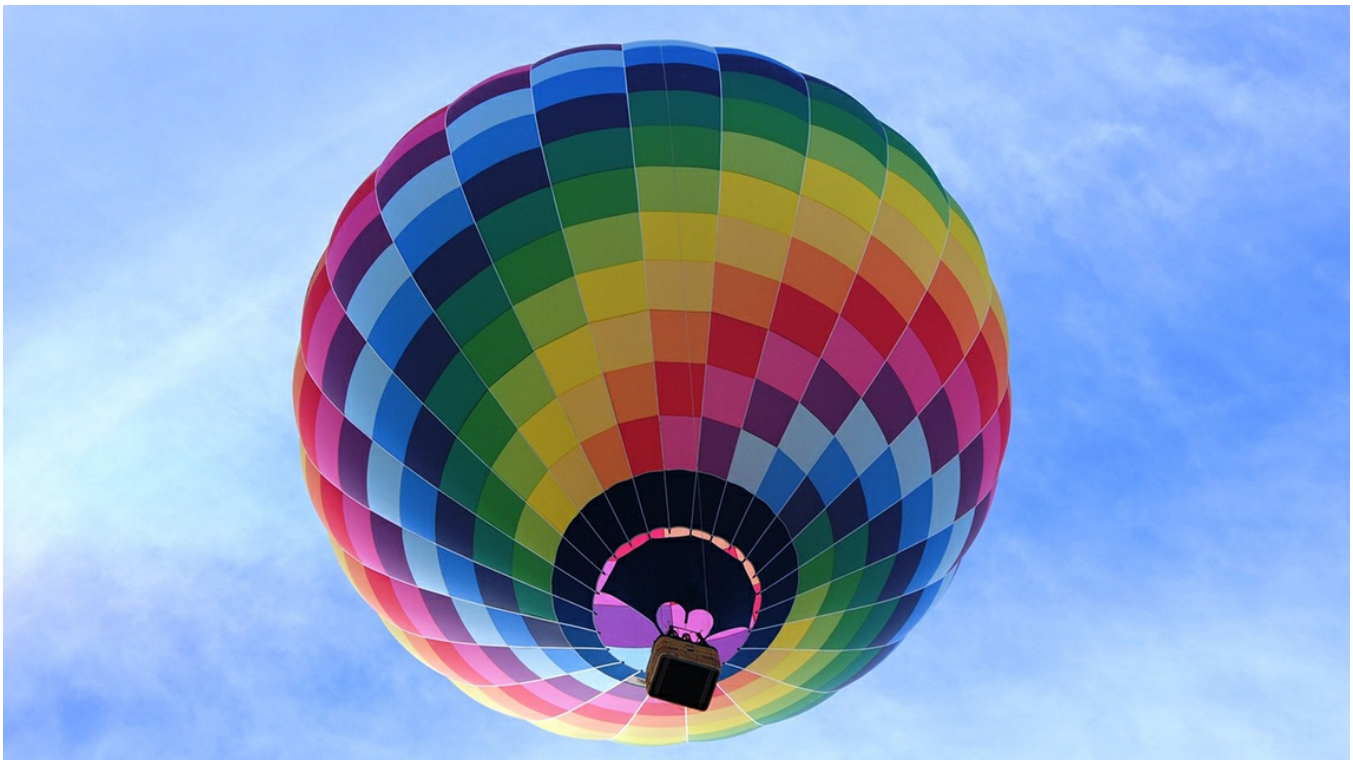
29 | 原子操作（上）

2018-10-17 郝林



我们在前两篇文章中讨论了互斥锁、读写锁以及基于它们的条件变量。互斥锁是一个很有用的同步工具，它可以保证每一时刻进入临界区的goroutine只有一个。读写锁对共享资源的写操作和读操作则区别看待，并消除了读操作之间的互斥。

条件变量主要是用于协调想要访问共享资源的那些线程。当共享资源的状态发生变化时，它可以被用来通知被互斥锁阻塞的线程，它既可以基于互斥锁，也可以基于读写锁。当然了，读写锁也是一种互斥锁，前者是对后者的扩展。

通过对互斥锁的合理使用，我们可以使一个goroutine在执行临界区中的代码时，不被其他的goroutine打扰。不过，虽然不会被打扰，但是它仍然可能会被中断（interruption）。

我们已经知道，对于一个Go程序来说，Go语言运行时系统中的调度器，会恰当地安排其中所有的goroutine的运行。不过，在同一时刻，只可能有少数的goroutine真正地处于运行状态，并且这个数量是固定的。

所以，为了公平起见，调度器总是会频繁地换上或换下这些goroutine。换上的意思是，让一个goroutine由非运行状态转为运行状态，并促使其中的代码在某个CPU核心上执行。

换下的意思正好相反，即：使一个goroutine中的代码中断执行，并让它由运行状态转为非运行状态。

这个中断的时机有很多，任何两条语句执行的间隙，甚至在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之内也是如此。所以，我们说，互斥锁虽然可以保证临界区中代码的串行执行，但却不能保证这些代码执行的原子性（atomicity）。

在众多的同步工具中，真正能够保证原子性执行的只有原子操作（atomic operation）。原子操作在进行的过程中是不允许中断的。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地消除竞态条件，并能够绝对地保证并发安全性。并且，它的执行速度要比其他的同步工具快得多，通常会高出好几个数量级。不过，它的缺点也很明显。

更具体地说，正是因为原子操作不能被中断，所以它需要足够简单，并且要求快速。 你可以想象一下，如果原子操作迟迟不能完成，而它又不会被中断，那么将会给计算机执行指令的效率带来多么大的影响。因此，操作系统层面只对针对二进制位或整数的原子操作提供了支持。

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。

我一般会通过下面这道题初探一下应聘者对sync/atomic包的熟悉程度。**我们今天的问题是：sync/atomic包中提供了几种原子操作？可操作的数据类型又有哪些？**

这里的典型回答是：

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。但是，对这些类型中的每一个，sync/atomic包都会有一套函数给予支持。这些数据类型有：int32、int64、uint32、uint64、uintptr，以及unsafe包中的Pointer。不过，针对unsafe.Pointer类型，该包并未提供进行原子加法操作的函数。

此外，sync/atomic包还提供了一个名为Value的类型，它可以被用来存储任意类型的值。

问题解析

这个问题很简单，因为答案是明摆在代码包文档里的。不过如果你连文档都没看过，那也可能回答不上来，至少是无法做出全面的回答。

我一般会通过此问题再衍生出来几道题。下面我就来逐个说明一下。

第一个衍生问题：我们都知道，传入这些原子操作函数的第一个参数值对应的都应该是那个被操作的值。比如，atomic.AddInt32函数的第一个参数，对应的一定是那个要被增大的整数。可是，这个参数的类型为什么不是int32而是*int32呢？

回答是：因为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被传入函数，就已经与函数外的那个值毫无关系了。

所以，传入值本身没有任何意义。unsafe.Pointer类型虽然是指针类型，但是那些原子操作函数要操作的是这个指针值，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。只有这样，它们才能够通过底层的指令，准确地操作这个内存地址上的数据。

第二个衍生问题： 用于原子加法操作的函数可以做原子减法吗？比如，atomic.AddInt32函数可以用于减小那个被操作的整数值吗？

回答是：当然是可以的。atomic.AddInt32函数的第二个参数代表差量，它的类型是int32，是有符号的。如果我们想做原子减法，那么把这个差量设置为负整数就可以了。

对于atomic.AddInt64函数来说也是类似的。不过，要想用atomic.AddUint32和atomic.AddUint64函数做原子减法，就不能这么直接了，因为它们的第二个参数的类型分别是uint32和uint64，都是无符号的，不过，这也是可以做到的，就是稍微麻烦一些。

例如，如果想对uint32类型的被操作值18做原子减法，比如说差量是-3，那么我们可以先把这个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，用表达式来描述就是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。

最后，我们把这个结果作为atomic.AddUint32函数的第二个参数值，就可以达到对uint32类型的值做原子减法的目的了。

还有一种更加直接的方式。我们可以依据下面这个表达式来给定atomic.AddUint32函数的第二个参数值：

```
^uint32(-N-1))
```

其中的N代表由负整数表示的差量。也就是说，我们先要把差量的绝对值减去1，然后再把得到

的这个无类型的整数常量，转换为uint32类型的值，最后，在这个值之上做按位异或操作，就可以获得最终的参数值了。

这么做的原理也并不复杂。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。

总结

今天，我们一起学习了sync/atomic代码包中提供的原子操作函数和原子值类型。原子操作函数使用起来都非常简单，但也有一些细节需要我们注意。我在主问题的衍生问题中对它们进行了逐一说明。

在下一篇文章中，我们会继续分享原子操作的衍生内容。如果你对原子操作有什么样的问题，都可以给我留言，我们一起讨论，感谢你的收听，我们下期再见。

戳此查看Go语言专栏文章配套详细代码。