17 | go语句及其执行规则（下）

2018-09-19 郝林

17 | go语句及其执行规则（下）

朗读人：黄洲君　09′31″ | 4.36M

你好，我是郝林，今天我们继续分享 go 语句执行规则的内容。

在上一篇文章中，我们讲到了 goroutine 在操作系统的并发编程体系，以及在 Go 语言并发编程模型中的地位和作用等一系列内容，今天我们继续来聊一聊这个话题。

知识扩展

问题 1：怎样才能让主 goroutine 等待其他 goroutine？

我刚才说过，一旦主 goroutine 中的代码执行完毕，当前的 Go 程序就会结束运行，无论其他的 goroutine 是否已经在运行了。那么，怎样才能做到等其他的 goroutine 运行完毕之后，再让主 goroutine 结束运行呢？

其实有很多办法可以做到这一点。其中，最简单粗暴的办法就是让主 goroutine“小睡”一会儿。

```
for i := 0; i < 10; i++ {
        go func() {
                fmt.Println(i)
        }()
}
time.Sleep(time.Millisecond * 500)
```

复制代码

在`for`语句的后边，我调用了`time`包的`Sleep`函数，并把`time.Millisecond * 500`的结果作为参数值传给了它。`time.Sleep`函数的功能就是让当前的 goroutine（在这里就是主 goroutine）暂停运行一段时间，直到到达指定的恢复运行时间。

我们可以把一个相对的时间传给该函数，就像我在这里传入的“500 毫秒”那样。`time.Sleep`函数会在被调用时用当前的绝对时间，再加上相对时间计算出在未来的恢复运行时间。显然，一旦到达恢复运行时间，当前的 goroutine 就会从“睡眠”中醒来，并开始继续执行后边的代码。

这个办法是可行的，只要“睡眠”的时间不要太短就好。不过，问题恰恰就在这里，我们让主 goroutine“睡眠”多长时间才是合适的呢？如果“睡眠”太短，则很可能不足以让其他的 goroutine 运行完毕，而若“睡眠”太长则纯属浪费时间，这个时间就太难把握了。

你可能会想到，既然不容易预估时间，那我们就让其他的 goroutine 在运行完毕的时候告诉我们好了。这个思路很好，但怎么做呢？

你是否想到了通道呢？我们先创建一个通道，它的长度应该与我们手动启用的 goroutine 的数量一致。在每个手动启用的 goroutine 即将运行完毕的时候，我们都要向该通道发送一个值。

注意，这些发送表达式应该被放在它们的go函数体的最后面。对应的，我们还需要在main函数的最后从通道接收元素值，接收的次数也应该与手动启用的 goroutine 的数量保持一致。关于这些你可以到 demo39.go 文件中，去查看具体的写法。

其中有一个细节你需要注意。我在声明通道sign的时候是以chan struct{}作为其类型的。其中的类型字面量struct{}有些类似于空接口类型interface{}，它代表了既不包含任何字段也不拥有任何方法的空结构体类型。

注意，struct{}类型值的表示法只有一个，即：struct{}{}。并且，它占用的内存空间是0字节。确切地说，这个值在整个 Go 程序中永远都只会存在一份。虽然我们可以无数次地使用这个值字面量，但是用到的却都是同一个值。

当我们仅仅把通道当作传递某种简单信号的介质的时候，用struct{}作为其元素类型是再好不过的了。顺便说一句，我在讲“结构体及其方法的使用法门”的时候留过一道与此相关的思考题，你可以返回去看一看。

再说回当下的问题，有没有比使用通道更好的方法？如果你知道标准库中的代码包sync的话，那么可能会想到sync.WaitGroup类型。没错，这是一个更好的答案。不过具体的使用方式我在后边讲sync包的时候再说。

问题 2：怎样让我们启用的多个 goroutine 按照既定的顺序运行？

在很多时候，当我沿着上面的主问题以及第一个扩展问题一路问下来的时候，应聘者往往会被这第二个扩展问题难住。

所以基于上一篇主问题中的代码，怎样做到让从0到9这几个整数按照自然数的顺序打印出来？你可能会说，我不用 goroutine 不就可以了嘛。没错，这样是可以，但是如果我不考虑这样做呢。你应该怎么解决这个问题？

当然了，众多应聘者回答的其他答案也是五花八门的，有的可行，有的不可行，还有的把原来的代码改得面目全非。我下面就来说说我的思路，以及心目中的答案吧。这个答案并不一定是最佳的，也许你在看完之后还可以想到更优的答案。

首先，我们需要稍微改造一下for语句中的那个go函数，要让它接受一个int类型的参数，并在调用它的时候把变量i的值传进去。为了不改动这个go函数中的其他代码，我们可以把它的这个参数也命名为i。

```
for i := 0; i < 10; i++ {
```

```
        go func(i int) {
                fmt.Println(i)
        }(i)
}
```

复制代码

只有这样，Go 语言才能保证每个 goroutine 都可以拿到一个唯一的整数。其原因与go函数的执行时机有关。

我在前面已经讲过了。在go语句被执行时，我们传给go函数的参数i会先被求值，如此就得到了当次迭代的序号。之后，无论go函数会在什么时候执行，这个参数值都不会变。也就是说，go函数中调用的fmt.Println函数打印的一定会是那个当次迭代的序号。

然后，我们在着手改造for语句中的go函数。

```
for i := uint32(0); i < 10; i++ {
        go func(i uint32) {
                fn := func() {
                        fmt.Println(i)
                }
                trigger(i, fn)
        }(i)
}
```

复制代码

我在go函数中先声明了一个匿名的函数，并把它赋给了变量fn。这个匿名函数做的事情很简单，只是调用fmt.Println函数以打印go函数的参数i的值。

在这之后，我调用了一个名叫trigger的函数，并把go函数的参数i和刚刚声明的变量fn作为参数传给了它。注意，for语句声明的局部变量i和go函数的参数i的类型都变了，都由int变为了uint32。至于为什么，我一会儿再说。

再来说trigger函数。该函数接受两个参数，一个是uint32类型的参数i，另一个是func()类型的参数fn。你应该记得，func()代表的是既无参数声明也无结果声明的函数类型。

```
trigger := func(i uint32, fn func()) {
        for {
                if n := atomic.LoadUint32(&count); n == i {
                        fn()
                        atomic.AddUint32(&count, 1)
```

```
                        break
                }
                time.Sleep(time.Nanosecond)
        }
}
```

复制代码

trigger函数会不断地获取一个名叫count的变量的值，并判断该值是否与参数i的值相同。如果相同，那么就立即调用fn代表的函数，然后把count变量的值加1，最后显式地退出当前的循环。否则，我们就先让当前的 goroutine“睡眠”一个纳秒再进入下一个迭代。

注意，我操作变量count的时候使用的都是原子操作。这是由于trigger函数会被多个 goroutine 并发地调用，所以它用到的非本地变量count，就被多个用户级线程共用了。因此，对它的操作就产生了竞态条件（race condition），破坏了程序的并发安全性。

所以，我们总是应该对这样的操作加以保护，在sync/atomic包中声明了很多用于原子操作的函数。

另外，由于我选用的原子操作函数对被操作的数值的类型有约束，所以我才对count以及相关的变量和参数的类型进行了统一的变更（由int变为了uint32）。

纵观count变量、trigger函数以及改造后的for语句和go函数，我要做的是，让count变量成为一个信号，它的值总是下一个可以调用打印函数的go函数的序号。

这个序号其实就是启用 goroutine 时，那个当次迭代的序号。也正因为如此，go函数实际的执行顺序才会与go语句的执行顺序完全一致。此外，这里的trigger函数实现了一种自旋（spinning）。除非发现条件已满足，否则它会不断地进行检查。

最后要说的是，因为我依然想让主 goroutine 最后一个运行完毕，所以还需要加一行代码。不过既然有了trigger函数，我就没有再使用通道。

```
trigger(10, func(){})
```

复制代码

调用trigger函数完全可以达到相同的效果。由于当所有我手动启用的 goroutine 都运行完毕之后，count的值一定会是10，所以我就把10作为了第一个参数值。又由于我并不想打印这个10，所以我把一个什么都不做的函数作为了第二个参数值。

总之，通过上述的改造，我使得异步发起的go函数得到了同步地（或者说按照既定顺序地）执行，你也可以动手自己试一试，感受一下。

总结

在本篇文章中，我们接着上一篇文章的主问题，讨论了当我们想让运行结果更加可控的时候，应该怎样去做。

主 goroutine 的运行若过早结束，那么我们的并发程序的功能就很可能无法全部完成。所以我们往往需要通过一些手段去进行干涉，比如调用time.Sleep函数或者使用通道。我们在后面的文章中还会讨论更高级的手段。

另外，go函数的实际执行顺序往往与其所属的go语句的执行顺序（或者说 goroutine 的启用顺序）不同，而且默认情况下的执行顺序是不可预知的。那怎样才能让这两个顺序一致呢？其实复杂的实现方式有不少，但是可能会把原来的代码改得面目全非。我在这里提供了一种比较简单、清晰的改造方案，供你参考。

总之，我希望通过上述基础知识以及三个连贯的问题帮你串起一条主线。这应该会让你更快地深入理解 goroutine 及其背后的并发编程模型，从而更加游刃有余地使用go语句。

思考题

1.runtime包中提供了哪些与模型三要素 G、P 和 M 相关的函数？（模型三要素内容在上一篇）