16 | go语句及其执行规则（上）

2018-09-17 郝林

16 | go语句及其执行规则（上）

朗读人：黄洲君　12′58″ | 7.43M

你很棒，已经学完了关于 Go 语言数据类型的全部内容。我相信你不但已经知晓了怎样高效地使用 Go 语言内建的那些数据类型，还明白了怎样正确地创造自己的数据类型。

对于 Go 语言的编程知识，你确实已经知道了不少了。不过，如果你真想玩转 Go 语言还需要知道它的一些特色流程和语法。

尤其是我们将会在本篇文章中讨论的go语句，这也是 Go 语言的最大特色了。它足可以代表 Go 语言最重要的编程哲学和并发编程模式。

让我们再重温一下下面这句话：

> Don’t communicate by sharing memory; share memory by communicating.

从 Go 语言编程的角度解释，这句话的意思就是：不要通过共享数据来通讯，恰恰相反，要以通讯的方式共享数据。

我们已经知道，通道（也就是 channel）类型的值可以被用来以通讯的方式共享数据。更具体地说，它一般被用来在不同的 goroutine 之间传递数据。那么 goroutine 到底代表着什么呢？

简单来说，goroutine 代表着并发编程模型中的用户级线程。你可能已经知道，操作系统本身提供了进程和线程这两种并发执行程序的工具。进程，描述的就是程序的执行过程，是运行着的程序的代表。

换句话说，一个进程其实就是某个程序运行时的一个产物。如果说静静地躺在那里的代码就是程序的话，那么奔跑着的、正在发挥着既有功能的代码就可以被称为进程。

我们的电脑为什么可以同时运行那么多应用程序？我们的手机为什么可以有那么多 App 同时在后台刷新？这都是因为在它们的操作系统之上有多个代表着不同应用程序或 App 的进程在同时运行。

再来说说线程。首先，线程总是在进程之内的，它可以被视为进程中运行着的控制流（或者说代码执行的流程）。

一个进程至少会包含一个线程。如果一个进程只包含了一个线程，那么它里面的所有代码都只会被串行地执行。每个进程的第一个线程都会随着该进程的启动而被创建，它们可以被称为其所属进程的主线程。

相对应的，如果一个进程中包含了多个线程，那么其中的代码就可以被并发地执行。除了进程的第一个线程之外，其他的线程都是由进程中已存在的线程创建出来的。

也就是说，主线程之外的其他线程都只能由代码显式地创建和销毁。这需要我们在编写程序的时候进行手动控制，操作系统以及进程本身并不会帮我们下达这样的指令，它们只会忠实地执行我们的指令。

不过，在 Go 程序当中，Go 语言的运行时（runtime）系统会帮助我们自动地创建和销毁系统级的线程。这里的系统级线程指的就是我们刚刚说过的操作系统提供的线程。

而对应的用户级线程指的是架设在系统级线程之上的，由用户（或者说我们编写的程序）完全控制的代码执行流程。用户级线程的创建、销毁、调度、状态变更以及其中的代码和数据都完全需要我们的程序自己去实现和处理。

这带来了很多优势，比如，因为它们的创建和销毁并不用通过操作系统去做，所以速度会很快，又比如，由于不用等着操作系统去调度它们的运行，所以往往会很容易控制并且可以很灵活。

但是，劣势也是有的，最明显也最重要的一个劣势就是复杂。如果我们只使用了系统级线程，那么我们只要指明需要新线程执行的代码片段，并且下达创建或销毁线程的指令就好了，其他的一切具体实现都会由操作系统代劳。

但是，如果使用用户级线程，我们就不得不既是指令下达者，又是指令执行者。我们必须全权负责与用户级线程有关的所有具体实现。

操作系统不但不会帮忙，还会要求我们的具体实现必须与它正确地对接，否则用户级线程就无法被并发地，甚至正确地运行。毕竟我们编写的所有代码最终都需要通过操作系统才能在计算机上执行。这听起来就很麻烦，不是吗？

不过别担心，Go 语言不但有着独特的并发编程模型，以及用户级线程 goroutine，还拥有强大的用于调度 goroutine、对接系统级线程的调度器。

这个调度器是 Go 语言运行时系统的重要组成部分，它主要负责统筹调配 Go 并发编程模型中的三个主要元素，即：G（goroutine 的缩写）、P（processor 的缩写）和 M（machine 的缩写）。

其中的 M 指代的就是系统级线程。而 P 指的是一种可以承载若干个 G，且能够使这些 G 适时地与 M 进行对接，并得到真正运行的中介。

从宏观上说，G 和 M 由于 P 的存在可以呈现出多对多的关系。当一个正在与某个 M 对接并运行着的 G，需要因某个事件（比如等待 I/O 或锁的解除）而暂停运行的时候，调度器总会及时地发现，并把这个 G 与那个 M 分离开，以释放计算资源供那些等待运行的 G 使用。

而当一个 G 需要恢复运行的时候，调度器又会尽快地为它寻找空闲的计算资源（包括 M）并安排运行。另外，当 M 不够用时，调度器会帮我们向操作系统申请新的系统级线程，而当某个 M 已无用时，调度器又会负责把它及时地销毁掉。

正因为调度器帮助我们做了很多事，所以我们的 Go 程序才总是能高效地利用操作系统和计算机资源。程序中的所有 goroutine 也都会被充分地调度，其中的代码也都会被并发地运行，即使这样的 goroutine 有数以十万计，也仍然可以如此。

由于篇幅原因，关于 Go 语言内部的调度器和运行时系统的更多细节，我在这里就不再深入讲述了。你需要知道，Go 语言实现了一套非常完善的运行时系统，保证了我们的程序在高并发的情况下依旧能够稳定、高效地运行。

如果你对这些具体的细节感兴趣，并还想进一步探索，那么我推荐你去看看我写的那本《Go 并发编程实战》。我在这本书中用了相当大的篇幅阐释了 Go 语言并发编程模型的原理、运作机制，以及所有与之紧密相关的知识。

下面，我会从编程实践的角度出发，以go语句的用法为主线，向你介绍go语句的执行规则、最佳实践和使用禁忌。

我们来看一下今天的**问题：什么是主 goroutine，它与我们启用的其他 goroutine 有什么不同？**

我们具体来看一道我在面试中经常提问的编程题。

```
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {
                go func() {
                        fmt.Println(i)
                }()
        }
}
```

复制代码

在 demo38.go 中，我只在main函数中写了一条for语句。这条for语句中的代码会迭代运行 10 次，并有一个局部变量i代表着当次迭代的序号，该序号是从0开始的。

在这条for语句中仅有一条go语句，这条go语句中也仅有一条语句。这条最里面的语句调用了fmt.Println函数并想要打印出变量i的值。

这个程序很简单，三条语句逐条嵌套。我的具体问题是：这个命令源码文件被执行后会打印出什么内容？

这道题的**典型回答**是：不会有任何内容被打印出来。

问题解析

与一个进程总会有一个主线程类似，每一个独立的 Go 程序在运行时也总会有一个主 goroutine。这个主 goroutine 会在 Go 程序的运行准备工作完成后被自动地启用，并不需要我们做任何手动的操作。

想必你已经知道，每条go语句一般都会携带一个函数调用，这个被调用的函数常常被称为go函数。而主 goroutine 的go函数就是那个作为程序入口的main函数。

一定要注意，go函数真正被执行的时间总会与其所属的go语句被执行的时间不同。当程序执行到一条go语句的时候，Go 语言的运行时系统，会先试图从某个存放空闲的 G 的队列中获取一个 G（也就是 goroutine），它只有在找不到空闲 G 的情况下才会去创建一个新的 G。

这也是为什么我总会说“启用”一个 goroutine 而不说“创建”一个 goroutine 的原因。已存在的 goroutine 总是会被优先复用。

然而，创建 G 的成本也是非常低的。创建一个 G 并不会像新建一个进程或者一个系统级线程那样，必须通过操作系统的系统调用来完成，在 Go 语言的运行时系统内部就可以完全做到了，更何况一个 G 仅相当于为需要并发执行代码片段服务的上下文环境而已。

在拿到了一个空闲的 G 之后，Go 语言运行时系统会用这个 G 去包装当前的那个go函数（或者说该函数中的那些代码），然后再把这个 G 追加到某个存放可运行的 G 的队列中。

这类队列中的 G 总是会按照先入先出的顺序，很快地由运行时系统内部的调度器安排运行。虽然这会很快，但是由于上面所说的那些准备工作还是不可避免的，所以耗时还是存在的。

因此，go函数的执行时间总是会明显滞后于它所属的go语句的执行时间。当然了，这里所说的“明显滞后”是对于计算机的 CPU 时钟和 Go 程序来说的。我们在大多数时候都不会有明显的感觉。

在说明了原理之后，我们再来看这种原理下的表象。请记住，只要go语句本身执行完毕，Go 程序完全不会等待go函数的执行，它会立刻去执行后边的语句。这就是所谓的异步并发地执行。

这里“后边的语句”指的一般是for语句中的下一个迭代。然而，当最后一个迭代运行的时候，这个“后边的语句”是不存在的。

在 demo38.go 中的那条for语句会以很快的速度执行完毕。当它执行完毕时，那 10 个包装了go函数的 goroutine 往往还没有获得运行的机会。

请注意，go函数中的那个对fmt.Println函数的调用是以for语句中的变量i作为参数的。你可以想象一下，如果当for语句执行完毕的时候，这些go函数都还没有执行，那么它们引用的变量i的值将会是什么？

它们都会是10，对吗？那么这道题的答案会是“打印出 10 个10”，是这样吗？

在确定最终的答案之前，你还需要知道一个与主 goroutine 有关的重要特性，即：一旦主 goroutine 中的代码（也就是main函数中的那些代码）执行完毕，当前的 Go 程序就会结束运行。

如此一来，如果在 Go 程序结束的那一刻，还有 goroutine 未得到运行机会，那么它们就真的没有运行机会了，它们中的代码也就不会被执行了。

我们刚才谈论过，当for语句的最后一个迭代运行的时候，其中的那条go语句即是最后一条语句。所以，在执行完这条go语句之后，主 goroutine 中的代码也就执行完了，Go 程序会立即结束运行。那么，如果这样的话，还会有任何内容被打印出来吗？

严谨地讲，Go 语言并不会去保证这些 goroutine 会以怎样的顺序运行。由于主 goroutine 会与我们手动启用的其他 goroutine 一起接受调度，又因为调度器很可能会在 goroutine 中的代码只执行了一部分的时候暂停，以期所有的 goroutine 有更公平的运行机会。

所以哪个 goroutine 先执行完、哪个 goroutine 后执行完往往是不可预知的，除非我们使用了某种 Go 语言提供的方式进行了人为干预。然而，在这段代码中，我们并没有进行任何人为干预。

那答案到底是什么呢？就 demo38.go 中如此简单的代码而言，绝大多数情况都会是“不会有任何内容被打印出来”。

但是为了严谨起见，无论应聘者的回答是“打印出 10 个10”还是“不会有任何内容被打印出来”，又或是“打印出乱序的0到9”，我都会紧接着去追问“为什么？”因为只有你知道了这背后的原理，你做出的回答才会被认为是正确的。

这个原理是如此的重要，以至于如果你不知道它，那么就几乎无法编写出正确的可并发执行的程序。如果你不知道此原理，那么即使你写的并发程序看起来可以正确地运行，那也肯定是运气好而已。

总结

今天，我描述了 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。这些知识点会为你打下一个坚实的基础。

我还提到了 Go 语言内部的运行时系统和调度器，以及它们围绕着 goroutine 做的那些统筹调配和维护工作。这些内容中的每句话应该都会对你正确理解 goroutine 起到实质性的作用。你可以用这些知识去解释主问题中的那个程序在运行后为什么会产出那样的结果。

下一篇内容，我们还会继续围绕 go 语句以及执行规则谈一些扩展知识，今天留给你的思考题就是：用什么手段可以对 goroutine 的启用数量加以限制？