

17 | go语句及其执行规则（下）

2018-09-19 郝林



17 | go语句及其执...

朗读人：黄洲君 09'31" |
4.36M

0:00 / 9:31

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。

```
1 for i := 0; i < 10; i++ {
2     go func() {
3         fmt.Println(i)
4     }()
5 }
6 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`。

```
1 for i := 0; i < 10; i++ {  
2     go func(i int) {  
3         fmt.Println(i)  
4     }(i)  
5 }
```


 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。

```
1 for i := uint32(0); i < 10; i++ {  
2     go func(i uint32) {  
3         fn := func() {  
4             fmt.Println(i)  
5         }  
6         trigger(i, fn)
```

 复制代码


```
7         }(i)
8     }
```

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。

```
1 trigger := func(i uint32, fn func()) {
2     for {
3         if n := atomic.LoadUint32(&count); n == i {
4             fn()
5             atomic.AddUint32(&count, 1)
6             break
7         }
8         time.Sleep(time.Nanosecond)
9     }
10 }
```

 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。

```
1 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）

[戳此查看 Go 语言专栏文章配套详细代码。](#)

GO语言核心36讲

3个月带你通关 GO 语言

郝林

《Go 并发编程实战》作者
GoHackers 技术社群发起人
前轻松筹大数据负责人



版权归极客邦科技所有，未经许可不得转载

写留言

精选留言

Askerlve
package main

```
import (  
    "fmt"  
    "sync/atomic"  
)
```

```
func main() {  
    var count uint32  
    trigger := func(i uint32, fn func()) {  
        for {  
            if n := atomic.LoadUint32(&count); n == i {  
                fn()  
                atomic.AddUint32(&count, 1)  
                break  
            }  
        }  
    }  
    for i := uint32(0); i < 10; i++ {  
        go func(i uint32) {  
            fn := func() {  
                fmt.Println(i)
```

👍 3

```

}
trigger(i, fn)
}(i)
}
trigger(10, func() {})
}

```

测试了下，这个函数的输出不受控，并且好像永远也不会结束，有人能帮忙解释下吗，go小白

~□

2018-09-19

作者回复

可以加个sleep

2018-09-20



湛泳豪

2

回楼上，atomic的加操作和读操作只有32位和64位整数型，所以必须要把int转为intxx。之所以这么做是因为int位数是根据系统决定的，而原子级操作要求速度尽可能的快，所以明确了整数的位数才能最大地提高性能。

2018-09-19



来碗绿豆汤

2

我有一个更简单的实现方式，如下

```

func main(){
ch := make(chan struct{})
for i:=0; i < 100; i++{
go func(i int){
fmt.Println(i)
ch <- struct{}{}
}(i)
<-ch
}
}

```

这样，每次循环都包装goroutine 执行结束才进入下一次循环，就可以保证顺序执行了

2018-09-19

作者回复

这些go函数的真正执行谁先谁后是不可控的，所以这样做不行的。

2018-09-20

AskerIve

2

package main

```

import (
"fmt"
"sync/atomic"
)

```

```

func main() {
var count uint32
trigger := func(i uint32, fn func()) {
for {
if n := atomic.LoadUint32(&count); n == i {
fn()
atomic.AddUint32(&count, 1)
break
}
}
}
for i := uint32(0); i < 10; i++ {
go func(i uint32) {
fn := func() {
fmt.Println(i)
}
trigger(i, fn)
}(i)
}
trigger(10, func() {})
}

```

这个函数的执行还是不可控诶，并且好像永远也不会结束，是因为我的go版本问题吗？

2018-09-19

作者回复

Win下可能会有问题，你在bif语句后边加一句time.sleep(time.Nanosecond)。github上的代码我已经更新了。

2018-09-20



cygnus

1

demo40的执行结果不是幂等的，程序经常无法正常结束退出，只有极少数几次有正确输出。

2018-09-19

作者回复

你在win下执行的嘛？

2018-09-20



冰激凌的眼泪

1

“ 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。 ”

示例代码里没有这个睡眠代码

2018-09-19

作者回复

代码已经更新了。

2018-09-20

timmy21

👍 1

有一个问题不太清楚，当i和count不相等时，您提到了睡眠1纳秒，可是我没看到有相关的sleep被调用。这是如何做到的？

2018-09-19

作者回复

代码已经更新了，漏掉了。

2018-09-20



老茂

👍 0

不加sleep程序不能正常结束的情况貌似跟cpu核数有关，我是4核cpu，打印0到2每次都可以正常执行；0到3以上就会有卡主的情况，卡主时cpu达到100%，load会超过4。猜测是不是此时所有cpu都在处理count==0的for循环，没有空闲的cpu执行atomic.AddUint32(&count, 1)？

2018-10-15

作者回复

Go语言调度goroutine是准抢占式的，虽然会防止某个goroutine运行太久，并做换下处理。但是像简单的死循环这种有可能会换下失败，尤其是windows下，这跟操作系统的底层支持有关。不过一般情况下不用担心。

2018-10-15



SuperP ❤️ 飛飛

👍 0

runtime.GOMAXPROCS 这个应该能控制P的数量

2018-10-11

作者回复

对，可以。

2018-10-15



拉铁

👍 0

来碗绿豆汤

□

我有一个更简单的实现方式，如下

```
func main(){
ch := make(chan struct{})
for i:=0; i < 100; i++){
go func(i int){
fmt.Println(i)
ch <- struct{}{}
}(i)
<-ch
}
}
```

这样，每次循环都包装goroutine 执行结束才进入下一次循环，就可以保证顺序执行了

2018-09-19

□ 作者回复

这些go函数的真正执行谁先谁后是不可控的，所以这样做不行的。

您好，从i=0开始，如果没有对ch做接收操作，就会发生堵塞，直到有数据发送到ch，也就是说如果当前循环内的goroutine没有运行结束，是不会运行到下一次迭代的，所以这样做也是可以保证顺序的。请问，我这么说是正确的吗？

2018-10-11



卒迹

0

郝老师，问个问题

```
trigger := func(i uint32, fn func()) {
    for {
        if n := atomic.LoadUint32(&count); n == i {
            fn()
            atomic.AddUint32(&count, 1)
            break
        }
        //为什么这里如果没有休眠的话，运行程序电脑会卡死呢
        time.Sleep(time.Nanosecond)
    }
}
```

2018-10-07



Lever

0

【来碗绿豆汤】写的程序感觉没有问题啊，在读 channel 的时候应该会一直等待吧，直到读到之后才会进行下一次循环，所以输出也应该是顺序的吧？再贴一遍他的代码：

```
func main(){
    ch := make(chan struct{})
    for i:=0; i < 100; i++){
        go func(i int){
            fmt.Println(i)
            ch <- struct{}{}
        }(i)
    }
    <-ch
}
```

2018-10-02



任性□

0

package main

```
import (
    "fmt"
    "time"
)
```

```

func main() {
var count uint32
trigger := func(i uint32, fn func()) {
for {
/* if n := atomic.LoadUint32(&count); n == i {
fn()
atomic.AddUint32(&count, 1)
break
} */
if count == i {
fn()
count++
break
}
time.Sleep(time.Nanosecond) //纳秒
}
}
for i := uint32(0); i < 10; i++ {
go func(i uint32) {
fn := func() {
fmt.Println(i, ">=", count)
}
trigger(i, fn)
}(i)
}
trigger(10, func() {})
}

```

老师这个代码会有问题吗
2018-09-29



新垣结裤

```

func main() {
num := 10
chs := [num+1]chan struct{}{}
for i := 0; i < num+1; i++ {
chs[i] = make(chan struct{})
}
for i := 0; i < num; i++ {
go func(i int) {
<- chs[i]
fmt.Println(i)
chs[i+1] <- struct{}{}
}
}
}

```

👍 0

```

}(i)
}
chs[0] <- struct{}{}
<- chs[num]
}

```

每个goroutine执行完通过channel通知下一个goroutine，在主goroutine里控制第一个goroutine的开始，并接收最后一个goroutine结束的信号

2018-09-21

作者回复

搞这么多通道有些浪费啊。另外切片不是并发安全的数据类型，最好不要这样用。

2018-09-23



cygnus

0

"" " demo40的执行结果不是幂等的，程序经常无法正常结束退出，只有极少数几次有正确输出。

2018-09-19

作者回复

你在win下执行的嘛？ "" "

是在ubuntu 1804下执行的

2018-09-20

作者回复

demo40.go我小改进了下，你再试试。我在服务器上也没出现你说的这个问题。Go语言很早的版本有可能这样，但是现在肯定不会了。你可联系下极客时间编辑，让她们把你加到群里。

2018-09-23



嗽大猫的鱼

0

老师，最近从头学习，前面一直没跟着动手，也没自己总结。这几天在整理每章的重点！

<https://github.com/wenxuwan/go36>

刚写完第二章，突然发现自己动手总结和只看差好多。我会继续保持喜欢总结！

2018-09-20

作者回复

很好，加油！

2018-09-20



蜗牛

0

```

func trigger(i int64, fn func()) {
for {
//if i != 10 {
// fmt.Print("")
//}

```

```

if count == i {
fn()

```

```

count += 1
break
}
}
}

func main() {
for i := int64(0); i <= 9; i++ {
go func(i int64) {
fn := func() {
fmt.Println(i)
}
trigger(i, fn)
}(i)
}
trigger(10, func() {})
}

```

取消注释后代码可顺序 0 ~9 输出. 而注释后则会莫名卡主, 怀疑是不是golang runtime 针对这些循环做了些什么, 而且感觉没必要加锁.

2018-09-19

作者回复

并发情况下必须利用某种同步工具, 否则就不是并发安全的, 生产环境中很可能出现不可控的问题。

2018-09-20



sky

👍 0

win64版本 : go1.10.2

linux64版本 : go1.11

linux下实际运行和预期一样, 但为何win下会一直运行不会停止呢, 且CPU也已经是100% 表示不解呀

2018-09-19

作者回复

可以加个sleep。

2018-09-20



硕

👍 0

老师好, 最后并没有说为什么使用uint(0), 请问是为什么啊?

2018-09-19

作者回复

我在文章可说了, 不过可能不太明显, 我更新了一下, 明天让编辑换上。其实就是因为原子操作无法针对int类型的值来做, 这需要是因为int类型的宽度会根据当前计算机的计算架构 (32 位或64位) 而改变。

2018-09-20