



08 并发基础：Goroutines 和 Channels 的声明与使用

在本节课开始之前，我们先一起回忆上节课的思考题：是否可以有多个 defer，如果可以的话，它们的执行顺序是怎么样的？

对于这道题，可以直接采用写代码测试的方式，如下所示：

```
func moreDefer(){
    defer fmt.Println("First defer")
    defer fmt.Println("Second defer")
    defer fmt.Println("Three defer")
    fmt.Println("函数自身代码")
}

func main(){
    moreDefer()
}
```

我定义了 moreDefer 函数，函数里有三个 defer 语句，然后在 main 函数里调用它。运行这段程序可以看到如下内容输出：

```
函数自身代码
Three defer
Second defer
First defer
```

通过以上示例可以证明：

1. 在一个方法或者函数中，可以有多个 defer 语句；
2. 多个 defer 语句的执行顺序依照后进先出的原则。

defer 有一个调用栈，越早定义越靠近栈的底部，越晚定义越靠近栈的顶部，在执行这些 defer 语句的时候，会先从栈顶弹出一个 defer 然后执行它，也就是我们示例中的结果。

下面我们开始本节课的学习。本节课是 Go 语言的重点——协程和通道，它们是 Go 语言并发的基础，我会从这两个基础概念开始，带你逐步深入 Go 语言的并发。

什么是并发

前面的课程中，我所写的代码都按照顺序执行，也就是上一句代码执行完，才会执行下一句，这样的代码逻辑简单，也符合我们的阅读习惯。

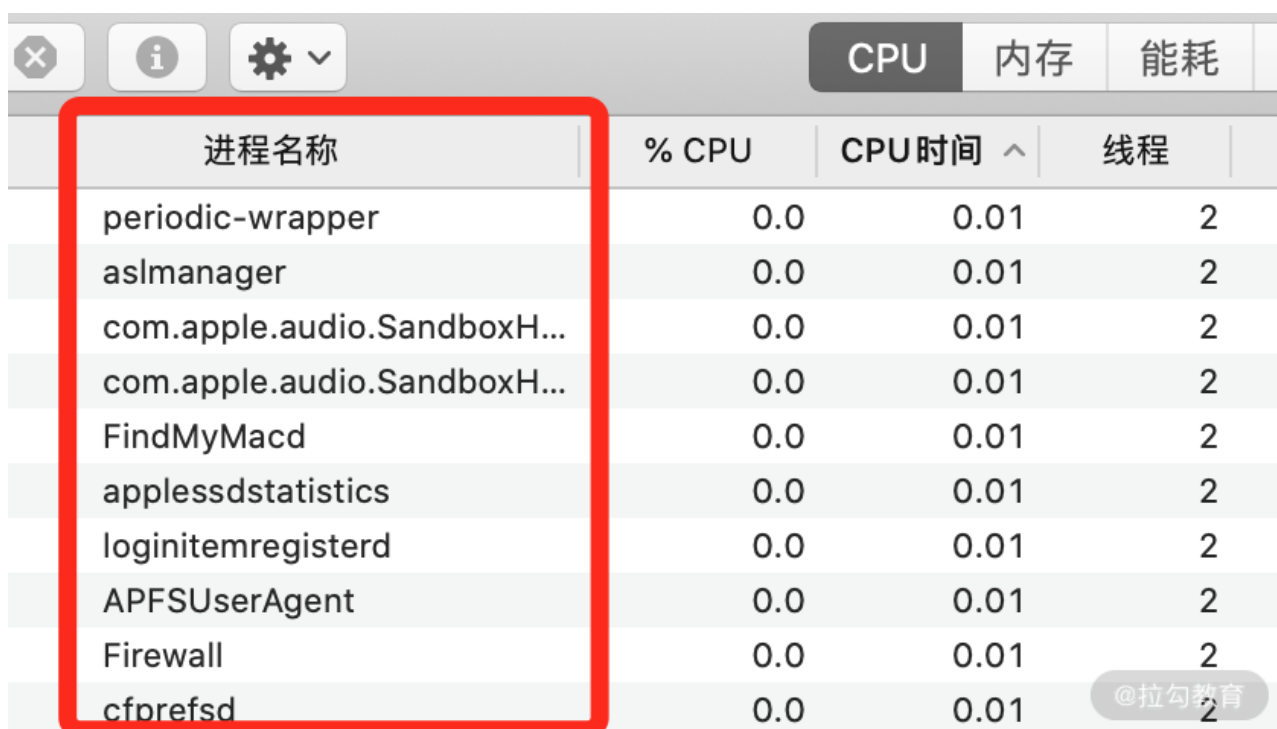
但这样是不够的，因为计算机很强大，如果只让它干完一件事情再干另外一件事情就太浪费了。比如一款音乐软件，使用它听音乐的时候还想让它下载歌曲，同一时刻做了两件事，在编程中，这就是并发，并发可以让你编写的程序在同一时刻做多几件事情。

进程和线程

讲并发就绕不开线程，不过在介绍线程之前，我先为你介绍什么是进程。

进程

在操作系统中，进程是一个非常重要的概念。当你启动一个软件（比如浏览器）的时候，操作系统会为此软件创建一个进程，这个进程是该软件的工作空间，它包含了软件运行所需的所有资源，比如内存空间、文件句柄，还有下面要讲的线程等。下面的图片就是我的电脑上运行的进程：



CPU				内存	能耗
进程名称	% CPU	CPU时间 ^	线程		
periodic-wrapper	0.0	0.01	2		
aslmanager	0.0	0.01	2		
com.apple.audio.SandboxH...	0.0	0.01	2		
com.apple.audio.SandboxH...	0.0	0.01	2		
FindMyMacd	0.0	0.01	2		
applesdstatistics	0.0	0.01	2		
loginitemregisterd	0.0	0.01	2		
APFSUserAgent	0.0	0.01	2		
Firewall	0.0	0.01	2		
cfprefsd	0.0	0.01	2		@拉勾教育

（电脑运行的进程）

那么线程是什么呢？

线程

线程是进程的执行空间，一个进程可以有多个线程，线程被操作系统调度执行，比如下载一个文件，发送一个消息等。这种多个线程被操作系统同时调度执行的情况，就是多线程的并发。

一个程序启动，就会有对应的进程被创建，同时进程也会启动一个线程，这个线程叫作主线程。如果主线程结束，那么整个程序就退出了。有了主线程，就可以从主线里启动很多其他线程，也就有了多线程的并发。

协程（Goroutine）

Go 语言中没有线程的概念，只有协程，也称为 goroutine。相比线程来说，协程更加轻量，一个程序可以随意启动成千上万个 goroutine。

goroutine 被 Go runtime 所调度，这一点和线程不一样。也就是说，Go 语言的并发是由 Go 自己所调度的，自己决定同时执行多少个 goroutine，什么时候执行哪几个。这些对于我们开发者来说完全透明，只需要在编码的时候告诉 Go 语言要启动几个 goroutine，至于如何调度执行，我们不用关心。

要启动一个 goroutine 非常简单，Go 语言为我们提供了 go 关键字，相比其他编程语言简化了很多，如下面的代码所示：

ch08/main.go

```
func main() {  
    go fmt.Println("飞雪无情")  
    fmt.Println("我是 main goroutine")  
    time.Sleep(time.Second)  
}
```

这样就启动了一个 goroutine，用来调用 fmt.Println 函数，打印“飞雪无情”。所以这段代码里有两个 goroutine，一个是 main 函数启动的 main goroutine，一个是我自己通过 go 关键字启动的 goroutine。

从示例中可以总结出 go 关键字的语法，如下所示：

```
go function()
```

go 关键字后跟一个方法或者函数的调用，就可以启动一个 goroutine，让方法在这个新启动的 goroutine 中运行。运行以上示例，可以看到如下输出：

```
我是 main goroutine  
飞雪无情
```

从输出结果也可以看出，程序是并发的，go 关键字启动的 goroutine 并不阻塞 main goroutine 的执行，所以我们才会看到如上打印结果。

小提示：示例中的 time.Sleep(time.Second) 表示等待一秒，这里是让 main goroutine 等一秒，不然 main goroutine 执行完毕程序就退出了，也就看不到启动的

新 goroutine 中“飞雪无情”的打印结果了。

Channel

那么如果启动了多个 goroutine，它们之间该如何通信呢？这就是 Go 语言提供的 channel（通道）要解决的问题。

声明一个 channel

在 Go 语言中，声明一个 channel 非常简单，使用内置的 make 函数即可，如下所示：

```
ch:=make(chan string)
```

其中 chan 是一个关键字，表示是 channel 类型。后面的 string 表示 channel 里的数据是 string 类型。通过 channel 的声明也可以看到，chan 是一个集合类型。

定义好 chan 后就可以使用了，一个 chan 的操作只有两种：发送和接收。

1. 接收：获取 chan 中的值，操作符为 <- chan。
2. 发送：向 chan 发送值，把值放在 chan 中，操作符为 chan <-。

小技巧：这里注意发送和接收的操作符，都是 <-，只不过位置不同。接收的 <- 操作符在 chan 的左侧，发送的 <- 操作符在 chan 的右侧。

现在我把上个示例改造下，使用 chan 来代替 time.Sleep 函数的等待工作，如下面的代码所示：

ch08/main.go

```
func main() {
    ch:=make(chan string)
    go func() {
        fmt.Println("飞雪无情")
        ch <- "goroutine 完成"
    }()
    fmt.Println("我是 main goroutine")
    v:=<-ch
    fmt.Println("接收到的chan中的值为：",v)
}
```

运行这个示例，可以发现程序并没有退出，可以看到"飞雪无情"的输出结果，达到了 `time.Sleep` 函数的效果，如下所示：

```
我是 main goroutine
飞雪无情
接收到的chan中的值为： goroutine 完成
```

可以这样理解：在上面的示例中，我们在新启动的 goroutine 中向 `chan` 类型的变量 `ch` 发送值；在 `main goroutine` 中，从变量 `ch` 接收值；如果 `ch` 中没有值，则阻塞等待到 `ch` 中有值可以接收为止。

相信你应该明白为什么程序不会在新的 goroutine 完成之前退出了，因为通过 `make` 创建的 `chan` 中没有值，而 `main goroutine` 又想从 `chan` 中获取值，获取不到就一直等待，等到另一个 goroutine 向 `chan` 发送值为止。

`channel` 有点像是在两个 goroutine 之间架设的管道，一个 goroutine 可以往这个管道里发送数据，另外一个可以从这个管道里取数据，有点类似于我们说的队列。

无缓冲 channel

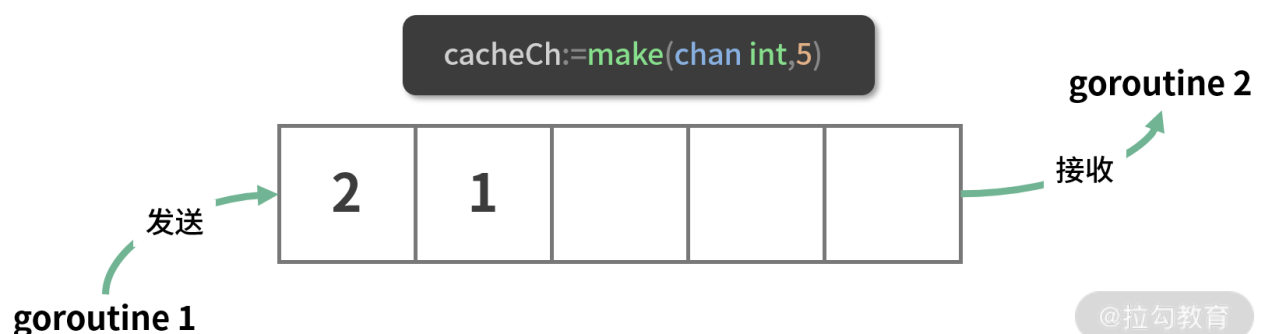
上面的示例中，使用 `make` 创建的 `chan` 就是一个无缓冲 `channel`，它的容量是 0，不能存储任何数据。所以无缓冲 `channel` 只起到传输数据的作用，数据并不会在 `channel` 中做任何停留。这也意味着，无缓冲 `channel` 的发送和接收操作是同时进行的，它也可以称为同步 `channel`。

有缓冲 channel

有缓冲 `channel` 类似一个可阻塞的队列，内部的元素先进先出。通过 `make` 函数的第二个参数可以指定 `channel` 容量的大小，进而创建一个有缓冲 `channel`，如下面的代码所示：

```
cacheCh:=make(chan int,5)
```

我创建了一个容量为 5 的 `channel`，内部的元素类型是 `int`，也就是说这个 `channel` 内部最多可以存放 5 个类型为 `int` 的元素，如下图所示：



(有缓冲 channel)

一个有缓冲 channel 具备以下特点：

1. 有缓冲 channel 的内部有一个缓冲队列；
2. 发送操作是向队列的尾部插入元素，如果队列已满，则阻塞等待，直到另一个 goroutine 执行，接收操作释放队列的空间；
3. 接收操作是从队列的头部获取元素并把它从队列中删除，如果队列为空，则阻塞等待，直到另一个 goroutine 执行，发送操作插入新的元素。

因为有缓冲 channel 类似一个队列，可以获取它的容量和里面元素的个数。如下面的代码所示：

ch08/main.go

```
cacheCh:=make(chan int,5)
cacheCh <- 2
cacheCh <- 3
fmt.Println("cacheCh容量为:",cap(cacheCh),"元素个数为:",len(cacheCh))
```

其中，通过内置函数 `cap` 可以获取 channel 的容量，也就是最大能存放多少个元素，通过内置函数 `len` 可以获取 channel 中元素的个数。

小提示：无缓冲 channel 其实就是一个容量大小为 0 的 channel。比如 `make(chan int,0)`。

关闭 channel

channel 还可以使用内置函数 `close` 关闭，如下面的代码所示：

```
close(cacheCh)
```

如果一个 channel 被关闭了，就不能向里面发送数据了，如果发送的话，会引起 `panic` 异常。但是还可以接收 channel 里的数据，如果 channel 里没有数据的话，接收的数据是元素类型的零值。

单向 channel

有时候，我们有一些特殊的业务需求，比如限制一个 channel 只可以接收但是不能发送，或者限制一个 channel 只能发送但不能接收，这种 channel 称为单向 channel。

单向 channel 的声明也很简单，只需要在声明的时候带上 `<-` 操作符即可，如下面的代码所示：

```
onlySend := make(chan<- int)
onlyReceive:=make(<-chan int)
```

注意，声明单向 channel `<-` 操作符的位置和上面讲到的发送和接收操作是一样的。

在函数或者方法的参数中，使用单向 channel 的较多，这样可以防止一些操作影响了 channel。

下面示例中的 `counter` 函数，它的参数 `out` 是一个只能发送的 channel，所以在 `counter` 函数体内使用参数 `out` 时，只能对其进行发送操作，如果执行接收操作，则程序不能编译通过。

```
func counter(out chan<- int) {
    //函数内容使用变量out，只能进行发送操作
}
```

select+channel 示例

假设要从网上下载一个文件，我启动了 3 个 goroutine 进行下载，并把结果发送到 3 个 channel 中。其中，哪个先下载好，就会使用哪个 channel 的结果。

在这种情况下，如果我们尝试获取第一个 channel 的结果，程序就会被阻塞，无法获取剩下两个 channel 的结果，也无法判断哪个先下载好。这个时候就需要用到多路复用操作了，在 Go 语言中，通过 `select` 语句可以实现多路复用，其语句格式如下：

```
select {
    case i1 = <-c1:
        //todo
    case c2 <- i2:
        //todo
    default:
        // default todo
}
```

整体结构和 `switch` 非常像，都有 `case` 和 `default`，只不过 `select` 的 `case` 是一个个可以操作的 channel。

小提示：多路复用可以简单地理解为，N 个 channel 中，任意一个 channel 有数据产生，`select` 都可以监听到，然后执行相应的分支，接收数据并处理。

有了 `select` 语句，就可以实现下载的例子了。如下面的代码所示：

ch08/main.go

```

func main() {

    //声明三个存放结果的channel
    firstCh := make(chan string)
    secondCh := make(chan string)
    threeCh := make(chan string)

    //同时开启3个goroutine下载
    go func() {
        firstCh <- downloadFile("firstCh")
    }()

    go func() {
        secondCh <- downloadFile("secondCh")
    }()

    go func() {
        threeCh <- downloadFile("threeCh")
    }()

    //开始select多路复用，哪个channel能获取到值，
    //就说明哪个最先下载好，就用哪个。
    select {
        case filePath := <-firstCh:
            fmt.Println(filePath)
        case filePath := <-secondCh:
            fmt.Println(filePath)
        case filePath := <-threeCh:
            fmt.Println(filePath)
    }
}

func downloadFile(chanName string) string {

    //模拟下载文件，可以自己随机time.Sleep点时间试试
    time.Sleep(time.Second)
    return chanName+":filePath"
}

```

如果这些 case 中有一个可以执行，select 语句会选择该 case 执行，如果同时有多个 case 可以被执行，则随机选择一个，这样每个 case 都有平等的被执行的机会。如果一个 select 没有任何 case，那么它会一直等待下去。

总结

在这节课中，我为你介绍了如何通过 go 关键字启动一个 goroutine，以及如何通过 channel 实现 goroutine 间的数据传递，这些都是 Go 语言并发的基础，理解它们可以更好地掌握并发。

在 Go 语言中，提倡通过通信来共享内存，而不是通过共享内存来通信，其实就是提倡通过 channel 发送接收消息的方式进行数据传递，而不是通过修改同一个变量。所以在**数据流动、传递**的场景中要优先使用 channel，它是并发安全的，性能也不错。

