09 | 同步原语：sync 包让你对并发控制得心应手

上节课留了一个思考题：channel 为什么是并发安全的呢？是因为 channel 内部使用了互斥锁来保证并发的安全，这节课，我将为你介绍互斥锁的使用。

在 Go 语言中，不仅有 channel 这类比较易用且高级的同步机制，还有 sync.Mutex、sync.WaitGroup 等比较原始的同步机制。通过它们，我们可以更加灵活地控制数据的同步和多协程的并发，下面我为你逐一讲解。

资源竞争

在一个 goroutine 中，如果分配的内存没有被其他 goroutine 访问，只在该 goroutine 中被使用，那么不存在资源竞争的问题。

但如果同一块内存被多个 goroutine 同时访问，就会产生不知道谁先访问也无法预料最后结果的情况。这就是资源竞争，这块内存可以称为共享的资源。

我们通过下面的示例来进一步地了解：

ch09/main.go

```
//共享的资源
var sum = 0
func main() {
  //开启100个协程让sum+10
    for i := 0; i < 100; i++ {
       go add(10)
    }
    //防止提前退出
    time.Sleep(2 * time.Second)
    fmt.Println("和为:",sum)
}
func add(i int) {
    sum += i
}
```

示例中，你期待的结果可能是“和为 1000”，但当运行程序后，可能如预期所示，但也可能是 990 或者 980。导致这种情况的核心原因是资源 sum 不是并发安全的，因为同时会有多个协程交叉执行 sum+=i，产生不可预料的结果。

既然已经知道了原因，解决的办法也就有了，只需要确保同时只有一个协程执行 sum+=i 操作即可。要达到该目的，可以使用 sync.Mutex 互斥锁。

> 小技巧：使用 go build、go run、go test 这些 Go 语言工具链提供的命令时，添加 -race 标识可以帮你检查 Go 语言代码是否存在资源竞争。

同步原语

sync.Mutex

互斥锁，顾名思义，指的是在同一时刻只有一个协程执行某段代码，其他协程都要等待该协程执行完毕后才能继续执行。

在下面的示例中，我声明了一个互斥锁 mutex，然后修改 add 函数，对 sum+=i 这段代码加锁保护。这样这段访问共享资源的代码片段就并发安全了，可以得到正确的结果。

ch09/main.go

```
var(
   sum int
   mutex sync.Mutex
)
func add(i int) {
   mutex.Lock()
   sum += i
   mutex.Unlock()
}
```

> 小提示：以上被加锁保护的 sum+=i 代码片段又称为临界区。在同步的程序设计中，临界区段指的是一个访问共享资源的程序片段，而这些共享资源又有无法同时被多个协程访问的特性。 当有协程进入临界区段时，其他协程必须等待，这样就保证了临界区的并发安全。

互斥锁的使用非常简单，它只有两个方法 Lock 和 Unlock，代表加锁和解锁。当一个协程获得 Mutex 锁后，其他协程只能等到 Mutex 锁释放后才能再次获得锁。

Mutex 的 Lock 和 Unlock 方法总是成对出现，而且要确保 Lock 获得锁后，一定执行 UnLock 释放锁，所以在函数或者方法中会采用 defer 语句释放锁，如下面的代码所示：

```
func add(i int) {
   mutex.Lock()
   defer mutex.Unlock()
   sum += i
}
```

这样可以确保锁一定会被释放，不会被遗忘。

sync.RWMutex

在 sync.Mutex 小节中，我对共享资源 sum 的加法操作进行了加锁，这样可以保证在修改 sum 值的时候是并发安全的。如果读取操作也采用多个协程呢？如下面的代码所示：

ch09/main.go

```
func main() {
    for i := 0; i < 100; i++ {
        go add(10)
    }
    for i:=0; i<10;i++ {
        go fmt.Println("和为:",readSum())
    }
    time.Sleep(2 * time.Second)
}
//增加了一个读取sum的函数，便于演示并发
func readSum() int {
    b:=sum
    return b
}
```

这个示例开启了 10 个协程，它们同时读取 sum 的值。因为 readSum 函数并没有任何加锁控制，所以它不是并发安全的，即一个 goroutine 正在执行 sum+=i 操作的时候，另一个 goroutine 可能正在执行 b:=sum 操作，这就会导致读取的 num 值是一个过期的值，结果不可预期。

如果要解决以上资源竞争的问题，可以使用互斥锁 sync.Mutex，如下面的代码所示：

ch09/main.go

```
func readSum() int {
    mutex.Lock()
    defer mutex.Unlock()
    b:=sum
    return b
}
```

因为 add 和 readSum 函数使用的是同一个 sync.Mutex，所以它们的操作是互斥的，也就是一个 goroutine 进行修改操作 sum+=i 的时候，另一个 gouroutine 读取 sum 的操作 b:=sum 会等待，直到修改操作执行完毕。

现在我们解决了多个 goroutine 同时读写的资源竞争问题，但是又遇到另外一个问题——性能。因为每次读写共享资源都要加锁，所以性能低下，这该怎么解决呢?

现在我们分析读写这个特殊场景，有以下几种情况：

1. 写的时候不能同时读，因为这个时候读取的话可能读到脏数据（不正确的数据）；
2. 读的时候不能同时写，因为也可能产生不可预料的结果；
3. 读的时候可以同时读，因为数据不会改变，所以不管多少个 goroutine 读都是并发安全的。

所以就可以通过读写锁 sync.RWMutex 来优化这段代码，提升性能。现在我将以上示例改为读写锁，来实现我们想要的结果，如下所示：

ch09/main.go

```
var mutex sync.RWMutex
func readSum() int {
    //只获取读锁
    mutex.RLock()
    defer mutex.RUnlock()
    b:=sum
    return b
}
```

对比互斥锁的示例，读写锁的改动有两处：

1. 把锁的声明换成读写锁 sync.RWMutex。
2. 把函数 readSum 读取数据的代码换成读锁，也就是 RLock 和 RUnlock。

这样性能就会有很大的提升，因为多个 goroutine 可以同时读数据，不再相互等待。

sync.WaitGroup

在以上示例中，相信你注意到了这段 time.Sleep(2 * time.Second) 代码，这是为了防止主函数 main 返回使用，一旦 main 函数返回了，程序也就退出了。

因为我们不知道 100 个执行 add 的协程和 10 个执行 readSum 的协程什么时候完全执行完毕，所以设置了一个比较长的等待时间，也就是两秒。

> 小提示：一个函数或者方法的返回 (return) 也就意味着当前函数执行完毕。

所以存在一个问题，如果这 110 个协程在两秒内执行完毕，main 函数本该提前返回，但是偏偏要等两秒才能返回，会产生性能问题。

如果这 110 个协程执行的时间超过两秒，因为设置的等待时间只有两秒，程序就会提前返回，导致有协程没有执行完毕，产生不可预知的结果。

那么有没有办法解决这个问题呢？也就是说有没有办法监听所有协程的执行，一旦全部执行完毕，程序马上退出，这样既可保证所有协程执行完毕，又可以及时退出节省时间，提升性能。你第一时间应该会想到上节课讲到的 channel。没错，channel 的确可以解决这个问题，不过非常复杂，Go 语言为我们提供了更简洁的解决办法，它就是 sync.WaitGroup。

在使用 sync.WaitGroup 改造示例之前，我先把 main 函数中的代码进行重构，抽取成一个函数 run，这样可以更好地理解，如下所示：

ch09/main.go

```
func main() {
    run()
}
func run(){
    for i := 0; i < 100; i++ {
        go add(10)
    }
    for i:=0; i<10;i++ {
        go fmt.Println("和为:",readSum())
    }
    time.Sleep(2 * time.Second)
}
```

这样执行读写的 110 个协程代码逻辑就都放在了 run 函数中，在 main 函数中直接调用 run 函数即可。现在只需通过 sync.WaitGroup 对 run 函数进行改造，让其恰好执行完毕，如下所示：

ch09/main.go

```
func run(){
    var wg sync.WaitGroup
    //因为要监控110个协程，所以设置计数器为110
    wg.Add(110)
    for i := 0; i < 100; i++ {
        go func() {
            //计数器值减1
            defer wg.Done()
            add(10)
        }()
    }
    for i:=0; i<10;i++ {
        go func() {
            //计数器值减1
            defer wg.Done()
            fmt.Println("和为:",readSum())
        }()
    }
    //一直等待，只要计数器值为0
    wg.Wait()
}
```

sync.WaitGroup 的使用比较简单，一共分为三步：

1. 声明一个 sync.WaitGroup，然后通过 Add 方法设置计数器的值，需要跟踪多少个协程就设置多少，这里是 110；
2. 在每个协程执行完毕时调用 Done 方法，让计数器减 1，告诉 sync.WaitGroup 该协程已经执行完毕；
3. 最后调用 Wait 方法一直等待，直到计数器值为 0，也就是所有跟踪的协程都执行完毕。

通过 sync.WaitGroup 可以很好地跟踪协程。在协程执行完毕后，整个 run 函数才能执行完毕，时间不多不少，正好是协程执行的时间。

sync.WaitGroup 适合协调多个协程共同做一件事情的场景，比如下载一个文件，假设使用 10 个协程，每个协程下载文件的 1/10 大小，只有 10 个协程都下载好了整个文件才算是下载好了。这就是我们经常听到的多线程下载，通过多个线程共同做一件事情，显著提高效率。

> 小提示：其实你也可以把 Go 语言中的协程理解为平常说的线程，从用户体验上也并无不可，但是从技术实现上，你知道他们是不一样的就可以了。

sync.Once

在实际的工作中，你可能会有这样的需求：让代码只执行一次，哪怕是在高并发的情况下，比如创建一个单例。

针对这种情形，Go 语言为我们提供了 sync.Once 来保证代码只执行一次，如下所示：

ch09/main.go

```
func main() {
   doOnce()
}
func doOnce() {
   var once sync.Once
   onceBody := func() {
      fmt.Println("Only once")
   }
   //用于等待协程执行完毕
   done := make(chan bool)
   //启动10个协程执行once.Do(onceBody)
   for i := 0; i < 10; i++ {
      go func() {
         //把要执行的函数(方法)作为参数传给once.Do方法即可
         once.Do(onceBody)
         done <- true
      }()
   }
   for i := 0; i < 10; i++ {
      <-done
   }
}
```

这是 Go 语言自带的一个示例，虽然启动了 10 个协程来执行 onceBody 函数，但是因为用了 once.Do 方法，所以函数 onceBody 只会被执行一次。也就是说在高并发的情况下，sync.Once 也会保证 onceBody 函数只执行一次。

sync.Once 适用于创建某个对象的单例、只加载一次的资源等只执行一次的场景。

sync.Cond

在 Go 语言中，sync.WaitGroup 用于最终完成的场景，关键点在于一定要等待所有协程都执行完毕。

而 sync.Cond 可以用于发号施令，一声令下所有协程都可以开始执行，关键点在于协程开始的时候是等待的，要等待 sync.Cond 唤醒才能执行。

sync.Cond 从字面意思看是条件变量，它具有阻塞协程和唤醒协程的功能，所以可以在满足一定条件的情况下唤醒协程，但条件变量只是它的一种使用场景。

下面我以 10 个人赛跑为例来演示 sync.Cond 的用法。在这个示例中有一个裁判，裁判要先等这 10 个人准备就绪，然后一声发令枪响，这 10 个人就可以开始跑了，如下所示：

```
//10个人赛跑，1个裁判发号施令
func race(){
    cond :=sync.NewCond(&sync.Mutex{})
    var wg sync.WaitGroup
    wg.Add(11)
    for i:=0;i<10; i++ {
        go func(num int) {
            defer  wg.Done()
            fmt.Println(num,"号已经就位")
            cond.L.Lock()
            cond.Wait()//等待发令枪响
            fmt.Println(num,"号开始跑……")
            cond.L.Unlock()
        }(i)
    }
    //等待所有goroutine都进入wait状态
    time.Sleep(2*time.Second)
    go func() {
        defer  wg.Done()
        fmt.Println("裁判已经就位，准备发令枪")
        fmt.Println("比赛开始，大家准备跑")
        cond.Broadcast()//发令枪响
    }()
    //防止函数提前返回退出
    wg.Wait()
}
```

以上示例中有注释说明，已经很好理解，我这里再大概讲解一下步骤：

1. 通过 sync.NewCond 函数生成一个 *sync.Cond，用于阻塞和唤醒协程；
2. 然后启动 10 个协程模拟 10 个人，准备就位后调用 cond.Wait() 方法阻塞当前协程等待发令枪响，这里需要注意的是调用 cond.Wait() 方法时要加锁；
3. time.Sleep 用于等待所有人都进入 wait 阻塞状态，这样裁判才能调用 cond.Broadcast() 发号施令；
4. 裁判准备完毕后，就可以调用 cond.Broadcast() 通知所有人开始跑了。

sync.Cond 有三个方法，它们分别是：

1. **Wait**，阻塞当前协程，直到被其他协程调用 Broadcast 或者 Signal 方法唤醒，使用的时候需要加锁，使用 sync.Cond 中的锁即可，也就是 L 字段。
2. **Signal**，唤醒一个等待时间最长的协程。
3. **Broadcast**，唤醒所有等待的协程。

> 注意：在调用 Signal 或者 Broadcast 之前，要确保目标协程处于 Wait 阻塞状态，不然会出现死锁问题。

如果你以前学过 Java，会发现 sync.Cond 和 Java 的等待唤醒机制很像，它的三个方法 Wait、Signal、Broadcast 就分别对应 Java 中的 wait、notify、notifyAll。

总结

这节课主要讲解 Go 语言的同步原语使用，通过它们可以更灵活地控制多协程的并发。从使用上讲，Go 语言还是更推荐 channel 这种更高级别的并发控制方式，因为它更简洁，也更容易理解和使用。

当然本节课讲的这些比较基础的同步原语也很有用。**同步原语通常用于更复杂的并发控制，如果追求更灵活的控制方式和性能，你可以使用它们。**

本节课到这里就要结束了，sync 包里还有一个同步原语我没有讲，它就是 sync.Map。sync.Map 的使用和内置的 map 类型一样，只不过它是并发安全的，所以这节课的作业就是练习使用 sync.Map。

下节课，我会为你讲解 Context，通过它你可以取消正在执行的协程。记得来听课！