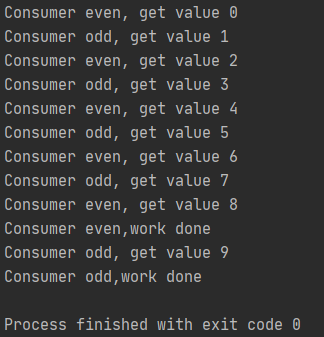
|  |
| --- |
| package main  import (  "fmt"  "sync"  )  var (  oddWait sync.WaitGroup//oddWait信号量，控制奇数消费者线程运行  evenWait sync.WaitGroup//evenWait信号量，控制偶数消费者线程运行  wg sync.WaitGroup//控制main函数等待全部goroutines执行完后结束  )  /\*  生产者线程  输入：奇数channel，偶数channel  处理：向奇数channel输入(1,3,5,7,9)，向偶数channel输入(0,2,4,6,8,)  \*/  func producer(odd chan<- int, even chan<- int) {  defer wg.Done()  //wg信号量v(1)  for i := 0; i < 10; i++ {  if i%2 == 0 {  even <- i  } else {  odd <- i  }  }  close(odd)  close(even)  //关闭channel  }  func oddConsunmer(odd <-chan int) {  defer wg.Done()  //wg信号量v(1)  for {  value, ok := <-odd  if !ok {  fmt.Printf("Consumer odd,work done\n")  return  }  oddWait.Wait()//oddWait信号量p(1)  fmt.Printf("Consumer odd, get value %d\n", value)  oddWait.Add(1)//oddWait信号量设值1  evenWait.Done()//evenWait信号量v(1)  }  }  func evenConsunmer(even <-chan int) {  defer wg.Done()  //wg信号量v(1)  for {  value, ok := <-even  if !ok {  fmt.Printf("Consumer even,work done\n")  return  }  evenWait.Wait()//evenWait信号量p(1)  fmt.Printf("Consumer even, get value %d\n", value)  evenWait.Add(1)//evenWait信号量设值1  oddWait.Done()//oddWait信号量v(1)  }  }  func main() {  oddChannel := make(chan int)  evenChannel := make(chan int)  //创建奇偶数channel  wg.Add(3)//wg等待组设值为3  oddWait.Add(1)//oddWait信号量初始值为1  go producer(oddChannel, evenChannel)  go oddConsunmer(oddChannel)  go evenConsunmer(evenChannel)  //执行go线程  wg.Wait()  //wg信号量p(3)  } |

**执行结果**



**程序思路**

控制evenConsunmer和oddConsunmer交替执行输出语句可通过信号量实现。程序中分别声明了控制evenConsunmer执行的evenWait和控制oddConsunmer执行的oddWait。通过执行sync.WaitGroup.wait()（P操作）和sync.WaitGroup.done()（V操作）便可实现线程间同步。

由于evenConsunmer最先执行，因此evenWait初值设为0，当evenConsunmer执行wait操作时，由于没有需要等待的资源，因此可直接执行。oddConsunmer后执行，因此设置oddWait.Add(1)，这样即使oddConsunmer先到达，也会因为等待资源释放而阻塞。

之后当evenConsunmer执行完一条输出语句后，需要分别执行evenWait.Add(1)和oddWait.Done()，注意这两条语句顺序不能颠倒，如果oddWait.Done()先执行，则oddConsunmer线程则有可能先执行evenWait.done()，而evenConsunmer后执行evenWait.Add(1)从而导致死锁。