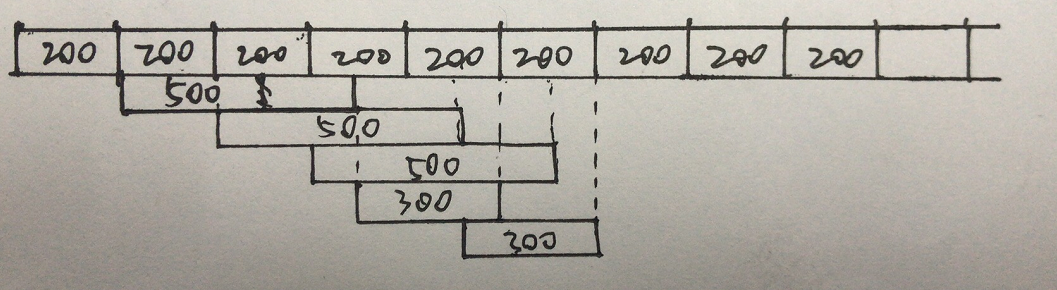
**goroutine和channel 练习题 • 假设生成一个自然数需要200毫秒的时间，对自然数 取平方需要500毫秒的时间，而输出一个自然数需要 300毫秒的时间，请修改上一页的程序，让输出所有数字所需时间最小。**

**– 提示一：可以用time.Sleep(200 \* time.Millisecond)来模拟时间消耗了200毫秒；**

**– 要求一：在goroutine的阻塞时间最小同时，请尽量减少内存花费；**

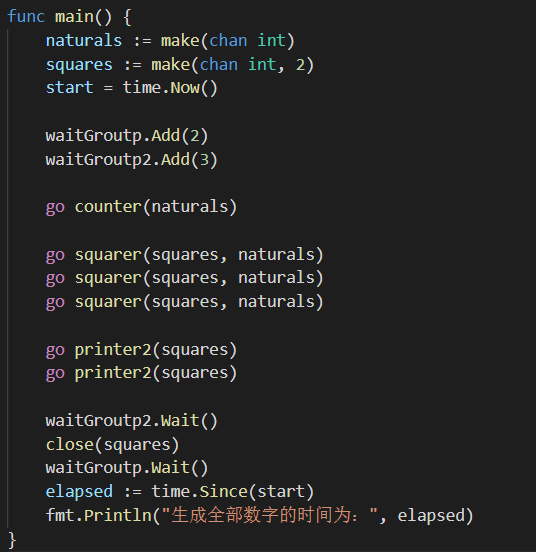
**– 要求二：请说明你的程序输出每一个数字所需的时间是多少。**

解决：助教要求只有一个counter,所以为问题为需要多少个squarer和printer函数使得输出自然数的时间效率最快。我们从如下图片可以得到

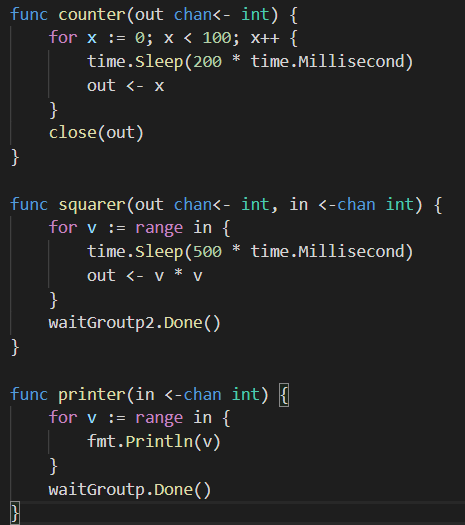


当到第四个200时候，第一个500执行完毕，用于处理第四个生成的200，所以500只需要三个，同理可得300只需要两个。即squarer和printer函数只需要三个和两个即可在goroutine的阻塞时间最小同时，程序内存花费尽量少，使得输出数字的效率最高。

main执行代码为：



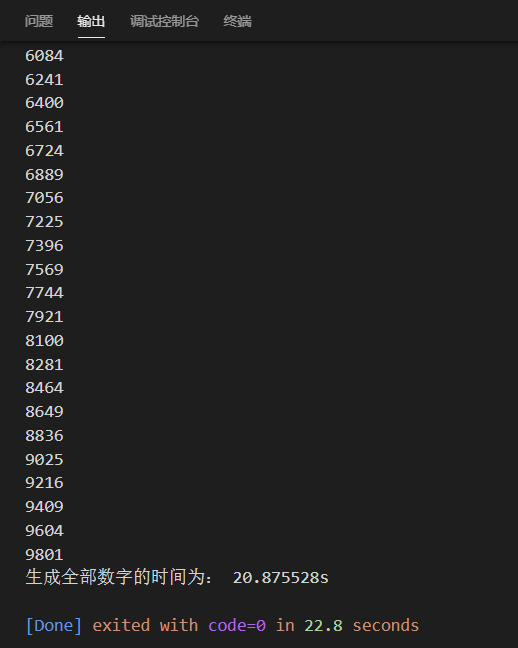
Counter、squarer、printer代码为：



设置两个信号量waitGroutp，waitGroutp2，前者的作用是让main()等待printer函数输出全部数字完毕才结束，后者作用是为了close(squares)关闭squares通道。

因为counter一个协程比较容易关闭通道，而squarer有三个协程不好关闭通道，因此我们这里设置信号量来帮忙关闭通道。

程序输出为：



符合理论值：200\*100+500+300=20800ms=20.8s的计算结果，而多出来的0.07s为程序其余的运行时间。

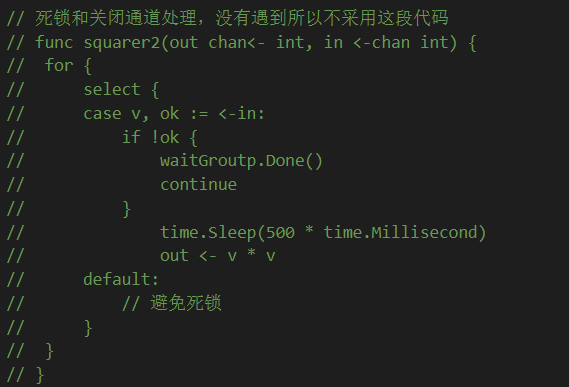
问题：程序有概率出现死锁问题，解决方案方案为对printer函数进行改写。

解决办法就是用select{}避免死锁问题出现，而in=nil效果可以使得printer函数在输出最后一个数字时候就进入default判断语句中，避免输出全部数字之后，还一直输出0,0,0，0,0,0…

改写代码如下：



Squarer的改写也是一样的。



简单测试下：(死锁情况较难发生)



完整源码如下：

package main

import (

    "fmt"

    "sync"

    "time"

)

var start = time.Now()

var waitGroutp = sync.WaitGroup{}

var waitGroutp2 = sync.WaitGroup{}

func counter(out chan<- int) {

    for x := 0; x < 100; x++ {

        time.Sleep(200 \* time.Millisecond)

        out <- x

    }

    close(out)

}

func squarer(out chan<- int, in <-chan int) {

    for v := range in {

        time.Sleep(500 \* time.Millisecond)

        out <- v \* v

    }

    waitGroutp2.Done()

}

// 死锁和关闭通道处理，没有遇到所以不采用这段代码

// func squarer2(out chan<- int, in <-chan int) {

//  for {

//      select {

//      case v, ok := <-in:

//          if !ok {

//              waitGroutp.Done()

//              continue

//          }

//              time.Sleep(500 \* time.Millisecond)

//              out <- v \* v

//      default:

//          // 避免死锁

//      }

//  }

// }

func printer(in <-chan int) {

    for v := range in {

        fmt.Println(v)

    }

    waitGroutp.Done()

}

//要是死锁的话可以让printer为下列函数

func printer2(in <-chan int) {

    for {

        select {

        case v, ok := <-in:

            if !ok {

                in = nil

                //在知道channel关闭后，将channel的值设为nil，这样子就相当于将这个select case子句停用了，因为nil的channel是永远阻塞的

                // fmt.Println("in is nil")

                waitGroutp.Done()

                continue

            }

            time.Sleep(300 \* time.Millisecond)

            fmt.Println(v)

        default:

            // 实现对死锁情况下的处理

            // time.Sleep(100 \* time.Millisecond)

        }

    }

}

func main() {

    naturals := make(chan int)

    squares := make(chan int, 2)

    start = time.Now()

    waitGroutp.Add(2)

    waitGroutp2.Add(3)

    go counter(naturals)

    go squarer(squares, naturals)

    go squarer(squares, naturals)

    go squarer(squares, naturals)

    go printer2(squares)

    go printer2(squares)

    waitGroutp2.Wait()

    close(squares)

    waitGroutp.Wait()

    elapsed := time.Since(start)

    fmt.Println("生成全部数字的时间为：", elapsed)

}