Golang goroutine channel 实现并发和并行

主讲教师：（大地）
合作网站：www.itying.com （IT 营）
我的专栏：https://www.itying.com/category-79-b0.html

一、为什么要使用 goroutine

需求：要统计 1-10000000 的数字中那些是素数，并打印这些素数？

素数：就是除了 1 和它本身不能被其他数整除的数

实现方法：

1、传统方法，通过一个 for 循环判断各个数是不是素数

2、使用并发或者并行的方式，将统计素数的任务分配给多个 goroutine 去完成，这个时候就用到了 goroutine

3、goroutine 结合 channel

二、进程、线程以及并行、并发

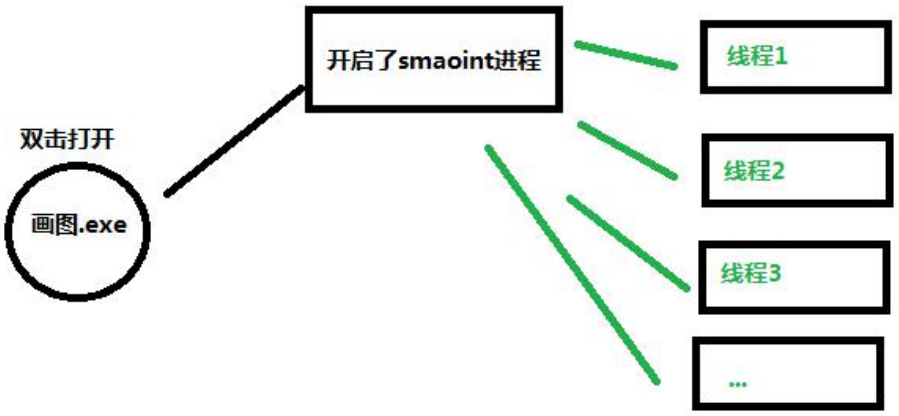
1、关于进程和线程

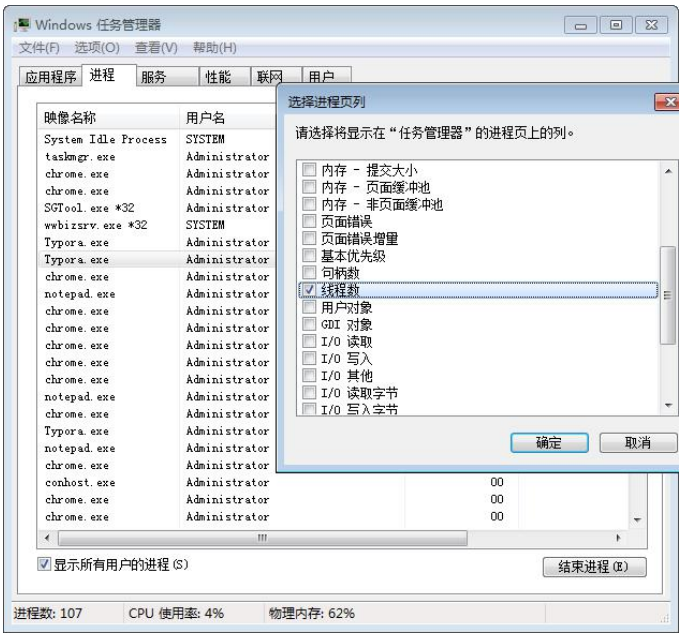
进程（Process）就是程序在操作系统中的一次执行过程，是系统进行资源分配和调度的基本单位，进程是一个动态概念，是程序在执行过程中分配和管理资源的基本单位，每一个进程都有一个自己的地址空间。一个进程至少有 5 种基本状态，它们是：初始态，执行态，等待状态，就绪状态，终止状态。

通俗的讲进程就是一个正在执行的程序。

线程 是进程的一个执行实例，是程序执行的最小单元，它是比进程更小的能独立运行的基本单位

一个进程可以创建多个线程，同一个进程中的多个线程可以并发执行，一个程序要运行的话至少有一个进程。



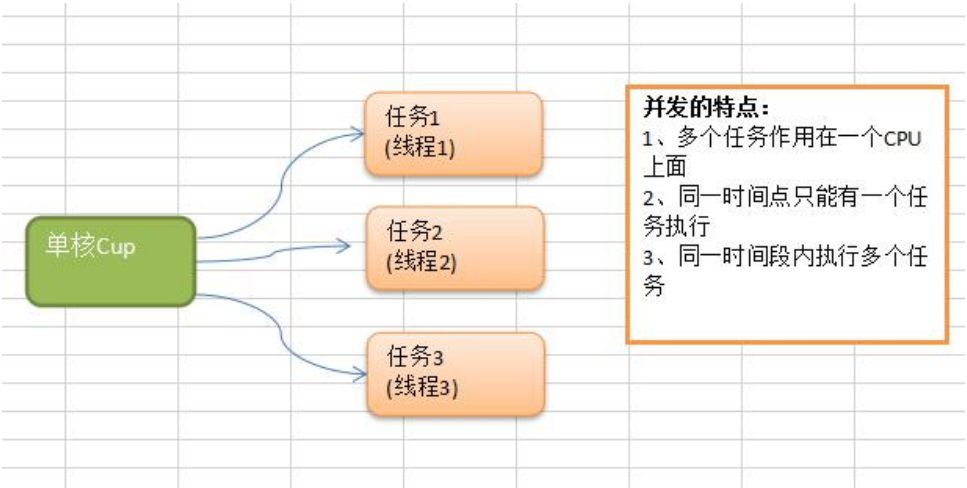


2、关于并行和并发

并发：多个线程同时竞争一个位置，竞争到的才可以执行，每一个时间段只有一个线程在执行。

并行：多个线程可以同时执行，每一个时间段，可以有多个线程同时执行。

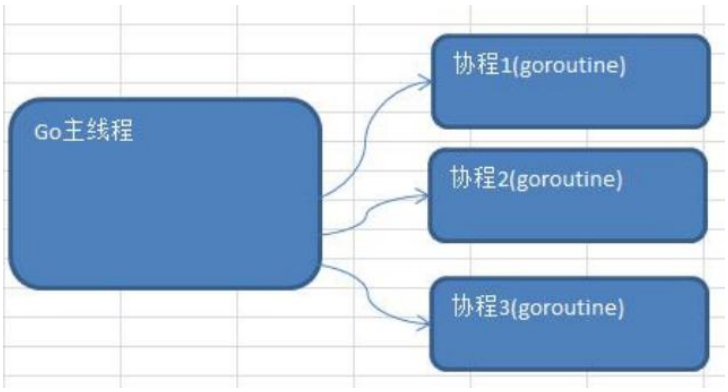
通俗的讲多线程程序在单核 CPU 上面运行就是**并发**，多线程程序在多核 CUP 上运行就是**并行**，如果线程数大于 CPU 核数，则多线程程序在多个 CPU 上面运行既有并行又有并发



三、Golang 中的协程（goroutine）以及主线程

golang 中的主线程：（可以理解为线程/也可以理解为进程），在一个 Golang 程序的主线程上可以起**多个协程**。**Golang 中多协程**可以实现并行或者并发。

协程：可以理解为用户级线程，这是对内核透明的，也就是系统并不知道有协程的存在，是完全由用户自己的程序进行调度的。Golang 的一大特色就是从语言层面原生支持协程，在函数或者方法前面加 go 关键字就可创建一个协程。可以说 Golang 中的协程就是 goroutine 。



Golang 中的多**协程**有点类似其他语言中的多线程。

多协程和多线程：Golang 中每个 goroutine (协程) 默认占用内存远比 Java 、C 的线程少。OS 线程（操作系统线程）一般都有固定的栈内存（通常为 2MB 左右）,一个 goroutine (协

程) 占用内存非常小，只有 2KB 左右，**多协程** goroutine 切换调度开销方面远比线程要少。这也是为什么越来越多的大公司使用 Golang 的原因之一。

四、Goroutine 的使用以及 sync.WaitGroup

并行执行需求：

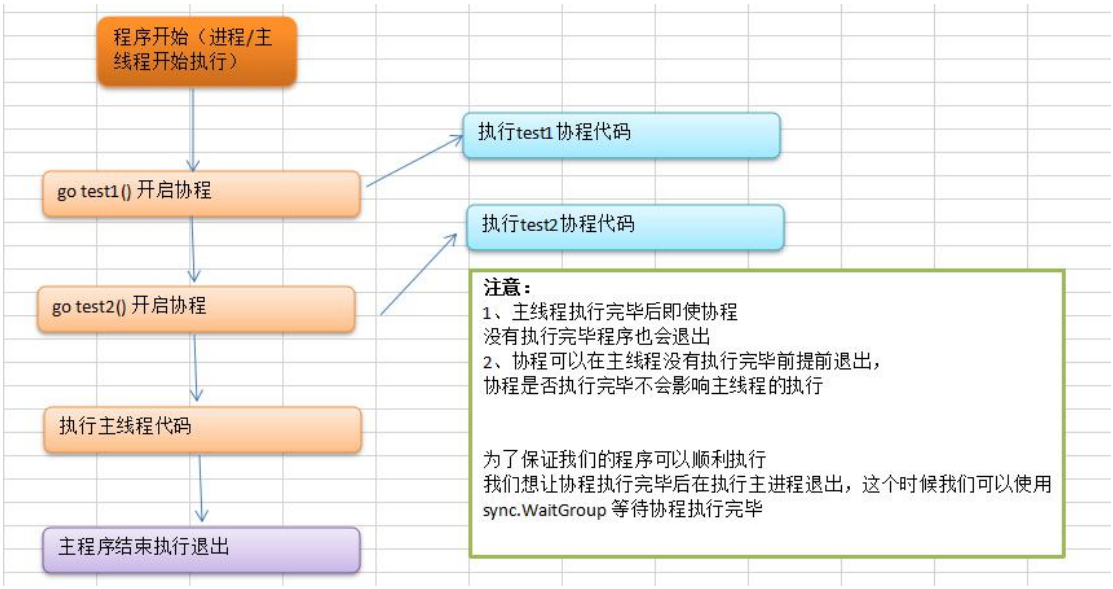
在主线程(可以理解成进程)中，开启一个 goroutine，该协程每隔 50 毫秒秒输出 "你好 golang"

在主线程中也每隔 50 毫秒输出"你好 golang"，输出 10 次后，退出程序，要求主线程和 goroutine 同时执行。

```
package main

import (
	"fmt"
	"strconv"
	"time"
)
func test() {
	for i := 1; i <= 10; i++ {
		fmt.Println("tesst () hello,world " + strconv.Itoa(i))
		time.Sleep(time.Second)
	}
}
func main() {
	go test() // 开启了一个协程
	for i := 1; i <= 10; i++ {
		fmt.Println(" main() hello,golang" + strconv.Itoa(i))
		time.Sleep(time.Second)
	}
```

```
}
```

上面代码看上去没有问题，但是要注意主线程执行完毕后即使协程没有执行完毕，程序会退出，所以我们需要对上面代码进行改造。



sync.WaitGroup 可以实现主线程等待协程执行完毕。

```
package main

import (
	"fmt"
	"strconv"
	"sync"
	"time"
)

var wg sync.WaitGroup //1、定义全局的 WaitGroup

func test() {
	for i := 1; i <= 10; i++ {
		fmt.Println("test ()  你好 golang " + strconv.Itoa(i))
```

```
            time.Sleep(time.Millisecond * 50)
        }
        wg.Done() // 4、goroutine 结束就登记-1
}
func main() {
        wg.Add(1) //2、启动一个 goroutine 就登记+1
        go test()
        for i := 1; i <= 2; i++ {
            fmt.Println(" main()  你好 golang" + strconv.Itoa(i))
            time.Sleep(time.Millisecond * 50)
        }
        wg.Wait() // 3、等待所有登记的 goroutine 都结束
}
```

五、启动多个 Goroutine

在 Go 语言中实现并发就是这样简单，我们还可以启动多个 goroutine。让我们再来一个例子：（这里使用了 sync.WaitGroup 来实现等待 goroutine 执行完毕）

```
var wg sync.WaitGroup
func hello(i int) {
        defer wg.Done() // goroutine 结束就登记-1
        fmt.Println("Hello Goroutine!", i)
}
func main() {
        for i := 0; i < 10; i++ {
            wg.Add(1) //  启动一个 goroutine 就登记+1
            go hello(i)
```

```
    }
    wg.Wait() //  等待所有登记的 goroutine 都结束
}
```

多次执行上面的代码，会发现每次打印的数字的顺序都不一致。这是因为 10 个 goroutine 是并发执行的，而 goroutine 的调度是随机的。

六、设置 Golang 并行运行的时候占用的 cup 数量

Go 运行时的调度器使用 GOMAXPROCS 参数来确定需要使用多少个 OS 线程来同时执行 Go 代码。默认值是机器上的 CPU 核心数。例如在一个 8 核心的机器上，调度器会把 Go 代码同时调度到 8 个 OS 线程上。

Go 语言中可以通过 runtime.GOMAXPROCS()函数设置当前程序并发时占用的 CPU 逻辑核心数。

Go1.5 版本之前，默认使用的是单核心执行。Go1.5 版本之后，默认使用全部的 CPU 逻辑核心数。

```
package main
import (
    "fmt"
    "runtime"
)
func main() {
    //获取当前计算机上面的 Cup 个数
    cpuNum := runtime.NumCPU()
    fmt.Println("cpuNum=", cpuNum)

    //可以自己设置使用多个 cpu
    runtime.GOMAXPROCS(cpuNum - 1)
```

```
	fmt.Println("ok")
}
```

七、Goroutine 统计素数

需求：要统计 1-120000 的数字中那些是素数？

1、通过传统的 for 循环来统计

```
func main() {
	start := time.Now().Unix()
	for num := 1; num <= 120000; num++ {
		flag := true //假设是素数
		for i := 2; i < num; i++ {
			if num%i == 0 { //说明该 num 不是素数
				flag = false
				break
			}
		}
		if flag {
			// fmt.Println(num)
		}
	}
	end := time.Now().Unix()
	fmt.Println("普通的方法耗时=", end-start)
}
```

2、goroutine 开启多个协程统计

```
package main
```

```
import (
	"fmt"
	"sync"
	"time"
)
var wg sync.WaitGroup
func fn1(n int) {
	for num := (n-1)*30000 + 1; num <= n*30000; num++ {
		flag := true //假设是素数
		for i := 2; i < num; i++ {
			if num%i == 0 {
				flag = false
				break
			}
		}
		if flag {
			// fmt.Println(num)
		}
	}
	wg.Done()
}

func main() {
	start := time.Now().Unix()
	for i := 1; i <= 4; i++ {
		wg.Add(1)
		go fn1(i)
```

```
    }

    wg.Wait()

    end := time.Now().Unix()

    fmt.Println("普通的方法耗时=", end-start)

}
```

问题：上面我们使用了 goroutine 已经能大大的提升新能了，但是如果我们想统计数据和打印数据同时进行，这个时候如何实现呢，这个时候我们就可以使用管道。

八、Channel 管道

管道是 Golang 在语言级别上提供的 goroutine 间的通讯方式，我们可以使用 channel 在多个 goroutine 之间传递消息。如果说 goroutine 是 Go 程序并发的执行体，channel 就是它们之间的连接。channel 是可以让一个 goroutine 发送特定值到另一个 goroutine 的通信机制。

Golang 的并发模型是 CSP（Communicating Sequential Processes），提倡**通过通信共享内存**而不是**通过共享内存而实现通信**。

Go 语言中的管道（channel）是一种特殊的类型。管道像一个传送带或者队列，总是遵循**先入先出（First In First Out）**的规则，保证收发数据的顺序。每一个管道都是一个具体类型的导管，也就是声明 channel 的时候需要为其指定元素类型。

1、channel 类型

channel 是一种类型，一种引用类型。声明管道类型的格式如下：

```
var 变量 chan 元素类型

举几个例子：

var ch1 chan int    // 声明一个传递整型的管道
var ch2 chan bool   // 声明一个传递布尔型的管道
var ch3 chan []int  // 声明一个传递 int 切片的管道
```

2、创建 channel

声明的管道后需要使用 make 函数初始化之后才能使用。

创建 channel 的格式如下：

```
make(chan 元素类型, 容量)
```

举几个例子：

```
//创建一个能存储 10 个 int 类型数据的管道
ch1 := make(chan int, 10)
//创建一个能存储 4 个 bool 类型数据的管道
ch2 := make(chan bool, 4)
//创建一个能存储 3 个[]int 切片类型数据的管道
ch3 := make(chan []int, 3)
```

3、channel 操作

管道有发送（send）、接收(receive）和关闭（close）三种操作。

发送和接收都使用<-符号。

现在我们先使用以下语句定义一个管道：

```
ch := make(chan int, 3)
```

1、发送（将数据放在管道内）

将一个值发送到管道中。

```
ch <- 10 // 把 10 发送到 ch 中
```

2、接收（从管道内取值）

从一个管道中接收值。

```
x := <- ch // 从 ch 中接收值并赋值给变量 x
<-ch       // 从 ch 中接收值，忽略结果
```

3、关闭管道

我们通过调用内置的 close 函数来关闭管道。

```
close(ch)
```

关于关闭管道需要注意的事情是，只有在通知接收方 goroutine 所有的数据都发送完毕的时候才需要关闭管道。管道是可以被垃圾回收机制回收的，它和关闭文件是不一样的，在结束操作之后关闭文件是必须要做的，但关闭管道不是必须的。

关闭后的管道有以下特点：

1. 对一个关闭的管道再发送值就会导致 panic。
2. 对一个关闭的管道进行接收会一直获取值直到管道为空。

3. 对一个关闭的并且没有值的管道执行接收操作会得到对应类型的零值。
4. 关闭一个已经关闭的管道会导致 panic。

4、管道阻塞

1、无缓冲的管道：

如果创建管道的时候没有指定容量，那么我们可以叫这个管道为无缓冲的管道

无缓冲的管道又称为阻塞的管道。我们来看一下下面的代码：

```
func main() {
    ch := make(chan int)
    ch <- 10
    fmt.Println("发送成功")
}
```

上面这段代码能够通过编译，但是执行的时候会出现以下错误：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        D:/go_demo/demo21/07goroutine/main.go:10 +0x5b
exit status 2
```

2、有缓冲的管道：

解决上面问题的方法还有一种就是使用有缓冲区的管道。我们可以在使用 make 函数初始化管道的时候为其指定管道的容量，例如：

```
func main() {
    ch := make(chan int, 1) // 创建一个容量为1的有缓冲区管道
    ch <- 10
    fmt.Println("发送成功")
}
```

只要管道的容量大于零，那么该管道就是有缓冲的管道，管道的容量表示管道中能存放元素的数量。就像你小区的快递柜只有那么个多格子，格子满了就装不下了，就阻塞了，等到别人取走一个快递员就能往里面放一个。

管道阻塞具体代码如下：

```
func main() {
	ch := make(chan int, 1)
	ch <- 10
	ch <- 12
	fmt.Println("发送成功")
}
```

解决办法：

```
func main() {
	ch := make(chan int, 1)
	ch <- 10 //放进去
	<-ch	//取走
	ch <- 12 //放进去
	<-ch	//取走
	ch <- 17 //还可以放进去
	fmt.Println("发送成功")
}
```

5、for range 从管道循环取值

当向管道中发送完数据时，我们可以通过 close 函数来关闭管道。

当管道被关闭时，再往该管道发送值会引发 panic，从该管道取值的操作会先取完管道中的值，再然后取到的值一直都是对应类型的零值。那如何判断一个管道是否被关闭了呢？

我们来看下面这个例子：

```
package  main

import  "fmt"

//循环遍历管道数据
```

```
func main() {

    var ch1 = make(chan int, 5)

    for i := 0; i < 5; i++ {
        ch1 <- i + 1
    }
    close(ch1) //关闭管道

    //使用 for range 遍历管道，当管道被关闭的时候就会退出 for range,如果没有关闭管道就会报个错误 fatal error: all goroutines are asleep - deadlock!

    //通过 for range 来遍历管道数据  管道没有 key
    for val := range ch1 {
        fmt.Println(val)
    }
}
```

从上面的例子中我们看到有两种方式在接收值的时候判断该管道是否被关闭，不过我们通常使用的是 for range 的方式。使用 for range 遍历管道，当管道被关闭的时候就会退出 for range。

九、Goroutine 结合 Channel 管道

需求 1：定义两个方法，一个方法给管道里面写数据，一个给管道里面读取数据。要求同步进行。

1、开启一个 fn1 的的协程给向管道 inChan 中写入 100 条数据

2、开启一个 fn2 的协程读取 inChan 中写入的数据

3、注意：fn1 和 fn2 同时操作一个管道

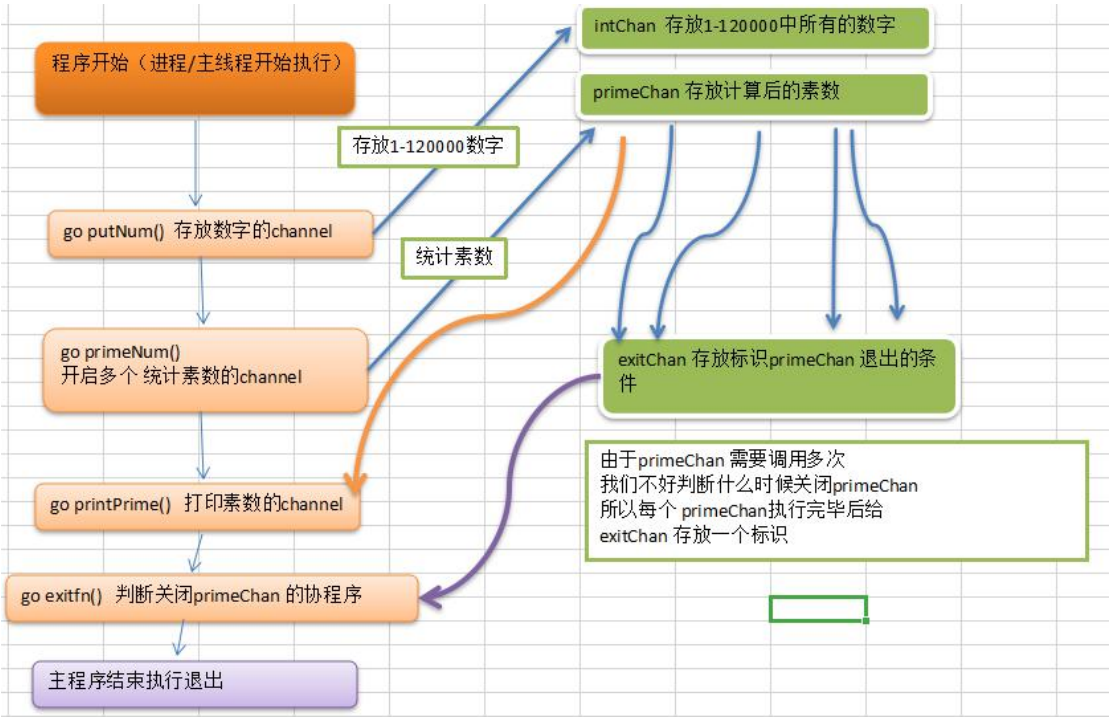
4、主线程必须等待操作完成后才可以退出

```
package main

import (

    "fmt"
```

```
	"sync"
	"time"
)
var wg sync.WaitGroup
func fn1(intChan chan int) {
	for i := 0; i < 100; i++ {
		intChan <- i + 1
		fmt.Println("writeData 写入数据-", i+1)
		time.Sleep(time.Millisecond * 100)
	}
	close(intChan)
	wg.Done()
}
func fn2(intChan chan int) {
	for v := range intChan {
		fmt.Printf("readData  读到数据=%v\n", v)
		time.Sleep(time.Millisecond * 50)
	}
	wg.Done()
}

func main() {
	allChan := make(chan int, 100)
	wg.Add(1)
	go fn1(allChan)
	wg.Add(1)
	go fn2(allChan)
```

```
    wg.Wait()

    fmt.Println("读取完毕...")

}
```

需求 2：goroutine 结合 channel 实现统计 1-120000 的数字中那些是素数？




```
package main

import (

    "fmt"

    "sync"

    "time"

)

var wg sync.WaitGroup

//向 intChan 放入 1-120000 个数

func putNum(intChan chan int) {

    for i := 1; i <= 1000; i++ {
```

```
            intChan <- i
        }
        //关闭 intChan
        close(intChan)
        wg.Done()
}
// 从 intChan 取出数据，并判断是否为素数,如果是，就放入到 primeChan
func primeNum(intChan chan int, primeChan chan int, exitChan chan bool) {
        for num := range intChan {
            var flag bool = true
            for i := 2; i < num; i++ {
                if num%i == 0 { //说明该 num 不是素数
                    flag = false
                    break
                }
            }
            if flag {
                //将这个数就放入到 primeChan
                primeChan <- num
            }
        }
        //判断关闭
        exitChan <- true
        wg.Done()
}
//打印素数的方法
func printPrime(primeChan chan int) {
```

```
	for v := range primeChan {
		fmt.Println(v)
	}
	wg.Done()
}

func main() {
	start := time.Now().Unix()
	intChan := make(chan int, 1000)
	primeChan := make(chan int, 20000) //放入结果
	//标识退出的管道
	exitChan := make(chan bool, 8) // 8 个

	//开启一个协程，向 intChan 放入 1-8000 个数
	wg.Add(1)
	go putNum(intChan)
	//开启 4 个协程，从 intChan 取出数据，并判断是否为素数,如果是，就放入到 primeChan
	for i := 0; i < 8; i++ {
		wg.Add(1)
		go primeNum(intChan, primeChan, exitChan)
	}
	//打印素数
	wg.Add(1)
	go printPrime(primeChan)

	//判断什么时候退出
	wg.Add(1)
```

```
    go func() {
        for i := 0; i < 8; i++ {
            <-exitChan
        }
        //当我们从 exitChan 取出了 8 个结果，就可以放心的关闭 prprimeChan
        close(primeChan)
        wg.Done()
    }()
    wg.Wait()
    end := time.Now().Unix()
    fmt.Println(end - start)
    fmt.Println("main 线程退出")
}
```

十、单向管道

有的时候我们会将管道作为参数在多个任务函数间传递，很多时候我们在不同的任务函数中使用管道都会对其进行限制，比如限制管道在函数中只能发送或只能接收。

例如：

```
    //1. 在默认情况下下，管道是双向
    //var chan1 chan int //可读可写

    //2 声明为只写
    var chan2 chan<- int
    chan2 = make(chan int, 3)
    chan2<- 20
    //num := <-chan2 //error
```

```
    fmt.Println("chan2=", chan2)


    //3. 声明为只读
    var chan3 <-chan int
    num2 := <-chan3
    //chan3<- 30 //err
    fmt.Println("num2", num2)
```

十、select 多路复用

在某些场景下我们需要同时从多个通道接收数据。这个时候就可以用到 golang 中给我们提供的 select 多路复用。

通常情况通道在接收数据时，如果没有数据可以接收将会发生阻塞。

比如说下面代码来实现从多个通道接受数据的时候就会发生阻塞：

```
for{
    // 尝试从 ch1 接收值
    data, ok := <-ch1
    // 尝试从 ch2 接收值
    data, ok := <-ch2
    …
}
```

这种方式虽然可以实现从多个管道接收值的需求，但是运行性能会差很多。为了应对这种场景，Go 内置了 select 关键字，可以同时响应多个管道的操作。

select 的使用类似于 switch 语句，它有一系列 case 分支和一个默认的分支。每个 case 会对应一个管道的通信（接收或发送）过程。select 会一直等待，直到某个 case 的通信操作完成时，就会执行 case 分支对应的语句。具体格式如下：

```
select{
    case <-ch1:
        ...
    case data := <-ch2:
        ...
    case ch3<-data:
        ...
```

```
    default:
        默认操作
}
```

举个小例子来演示下 select 的使用：

```
//1.定义一个管道 10 个数据 int
    intChan := make(chan int, 10)
    for i := 0; i < 10; i++ {
        intChan <- i
    }
    //2.定义一个管道 5 个数据 string
    stringChan := make(chan string, 5)
    for i := 0; i < 5; i++ {
        stringChan <- "hello" + fmt.Sprintf("%d", i)
    }


    for {
        select {
        case v := <-intChan:
            fmt.Printf("从 intChan 读取的数据%d\n", v)
        case v := <-stringChan:
            fmt.Printf("从 stringChan 读取的数据%s\n", v)
        default:
            fmt.Printf("都取不到了，不玩了，程序员可以加入逻辑\n")
            time.Sleep(time.Second)
            return
        }
    }
```

十、Golang 并发安全和锁

1、互斥锁

互斥锁是传统并发编程中对共享资源进行访问控制的主要手段，它由标准库 sync 中的 Mutex 结构体类型表示。sync.Mutex 类型只有两个公开的指针方法，Lock 和 Unlock。Lock 锁定当前的共享资源，Unlock 进行解锁

有问题代码：

```
package main

import (
	"fmt"
	"time"
)

var count = 0

func test() {
	count++
	fmt.Println("the count is : ", count)
	time.Sleep(time.Millisecond)
}

func main() {
	for r := 0; r < 100; r++ {
		go test()
	}
	time.Sleep(time.Second)
}
```

go build -race main.go 然后我们运行 main.exe 就知道到底哪里存在互斥

互斥锁解决这个问题：

```
package main

import (
	"fmt"
	"time"
)
var count = 0
func test() {
	count++
	fmt.Println("the count is : ", count)
	time.Sleep(time.Millisecond)
}
func main() {
	for r := 0; r < 100; r++ {
		go test()
	}
	time.Sleep(time.Second)
}
```

使用互斥锁能够保证同一时间有且只有一个 goroutine 进入临界区，其他的 goroutine 则在等待锁；当互斥锁释放后，等待的 goroutine 才可以获取锁进入临界区，多个 goroutine 同时等待一个锁时，唤醒的策略是随机的。

虽然使用互斥锁能解决资源争夺问题，但是并不完美，通过全局变量加锁同步来实现通讯，并不利于多个协程对全局变量的读写操作。这个时候我们也可以通过另一种方式来实现上面的功能管道(Channel)。

2、读写互斥锁

互斥锁的本质是当一个 goroutine 访问的时候，其他 goroutine 都不能访问。这样在资源同步，避免竞争的同时也降低了程序的并发性能。程序由原来的并行执行变成了串行执行。

其实，当我们对一个不会变化的数据只做“读”操作的话，是不存在资源竞争的问题的。因为数据是不变的，不管怎么读取，多少 goroutine 同时读取，都是可以的。

所以问题不是出在“读”上，主要是修改，也就是“写”。修改的数据要同步，这样其他 goroutine 才可以感知到。所以真正的互斥应该是读取和修改、修改和修改之间，读和读是没有互斥操作的必要的。

因此，衍生出另外一种锁，叫做**读写锁**。

读写锁可以让多个读操作并发，同时读取，但是对于写操作是完全互斥的。也就是说，当一个 goroutine 进行写操作的时候，其他 goroutine 既不能进行读操作，也不能进行写操作。

GO 中的读写锁由结构体类型 sync.RWMutex 表示。此类型的方法集合中包含两对方法：

一组是对写操作的锁定和解锁，简称“写锁定”和“写解锁”：

```
func (*RWMutex)Lock()

func (*RWMutex)Unlock()
```

另一组表示对读操作的锁定和解锁，简称为“读锁定”与“读解锁”：

```
func (*RWMutex)RLock()

func (*RWMutex)RUnlock()读写锁示例：
```

```
package main

import (

    "fmt"

    "sync"
```

```
	"time"
)
var count int
var mutex sync.RWMutex
var wg sync.WaitGroup
//写的方法
func write() {
	mutex.Lock()
	fmt.Println("执行写操作")
	time.Sleep(time.Second * 3)
	mutex.Unlock()
	wg.Done()
}
//读的方法
func read() {
	mutex.RLock()
	fmt.Println("执行读操作")
	time.Sleep(time.Second * 3)
	mutex.RUnlock()
	wg.Done()
}
func main() {
	// 开启 10 个协程执行写操作
	for i := 0; i < 10; i++ {
		wg.Add(1)
		go read()
	}
```

```
    //开启 10 个协程执行读操作
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go write()
    }
    wg.Wait()
}
```

十一、Goroutine Recover 解决协程中出现的 Panic

```
package main
import (
    "fmt"
    "time"
)

//函数
func sayHello() {
    for i := 0; i < 10; i++ {
        time.Sleep(time.Second)
        fmt.Println("hello,world")
    }
}
//函数
func test() {
    //这里我们可以使用 defer + recover
    defer func() {
        //捕获 test 抛出的 panic
        if err := recover(); err != nil {
            fmt.Println("test() 发生错误", err)
        }
    }()
    //定义了一个 map
    var myMap map[int]string
    myMap[0] = "golang" //error
}
```

```
func main() {

	go sayHello()
	go test()


	for i := 0; i < 10; i++ {
		fmt.Println("main() ok=", i)
		time.Sleep(time.Second)
	}

}
```