并发

并发并行？OS厨狂喜😘

并发程序指同时进行多个任务的程序

Go的并发程序?

1. 多线程共享内存（更为传统，in ch9）
2. CSP: goroutine & channel（Communicating Sequential Processes，顺序通信进程，现代的并发编程模型）

我们需要更多的goroutines

Goroutines

协程？轻量级线程？比thread还便宜高效？？ 😘

what

每一个并发的执行单元 = 一个goroutine

可以简单的把goroutine类比作一个线程（只是"类比"，本质区别 in ch9）

程序启动——主函数在一个单独的 `main gorourtine` 中运行
新的goroutine用go语句（ `go` 关键字）来创建

```
func f() ...

f() // 调用函数f()，等待其返回
go f() // 创建一个新的goroutine调用f()，无需等待其返回
```

e.g. `fib()`

```
func main() {
        go spinner(100 * time.Millisecond)
        const n = 45
        fibN := fib(n)
        fmt.Printf("\rFibnacci(%d) = %d\n", n, fibN)
}

// 整一个小动画，省得算的无聊
// range `-\|/`？但是它会转哎😋🥳
func spinner(delay time.Duration) {
        for {
                for _,  r := range `-\|/` {
                        fmt.Printf("\r%c", r)
                        time.Sleep(delay)
                }
        }
}

func fib(x int) int {
        if x < 2 {
                return x
        }
        return fib(x-1) + fib(x-2)
}
```

`main goroutine` 先整了个新的goroutine跑 `spinner()` 函数，然后无需等待继续跑自己的 `fib()` 函数，两者**同时执行**

注意，spinner可是一个无限循环，但是并没有一直转下去，因为主函数不会等其他的goroutine，跑完之后就返回。主函数返回时，所有的goroutine都会被直接打断，程序退出

Channels

what

管道，pipe？ 😋

通信机制！channel可以让一个goroutine给另一个goroutine发送值信息

每个channel都有一个特殊的类型，也就是channels可发送数据的类型
e.g.可发送int数据的channel(一般写为chan int)

```
ch := make(chan int) // ch type--'chan int'
```

make大法创建，引用类型（类似于map）

make, 无缓存 / 有缓存（即容量大小）channel

```
ch = make(chan int) // 无缓存
ch = make(chan int, 0) // 无缓存（容量为0？0就是无！😋）
ch = make(chan int, 3) // 有缓存，容量为3
```

通信操作

1.发送 2.接收（pipe，人称小channel😘）

两个操作都用 <- 运算符（生动形象）

```
ch <- x // 向ch发送x（or ch接收x）
x = <-ch // 用x接收ch发送的内容
fmt.Println(<-ch) // 其实<-ch这个整体就代表ch要发送的内容
```

e.g. 分段求和（数据并行？！ 😍）

```
func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // 将结果发送到通道
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int) // 创建通道
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)

    x, y := <-c, <-c // 从通道接收

    fmt.Println(x, y, x+y)
}
```

其实还有 3.关闭 close

```
close(ch)
```

就OK咯

channel关闭后，对ch进行发送操作——异常，但是之前已经发送成功的数据还可以接收到

无缓存Channels

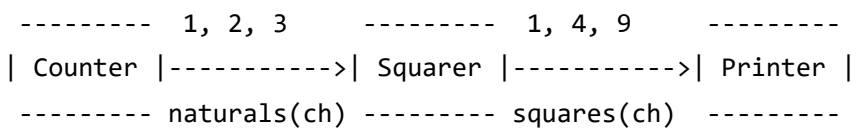
基于无缓存的Channels的发送 / 接收操作会导致发送者goroutine阻塞，直至另一个接收 / 发送

发送&接收操作——一次同步操作（基于无缓存Channels）
So, 无缓存Channels也称为同步Channels

串联的Channels（Pipeline)

woc, 管道！贞德试泥鸭！！

channels将多个goroutine连接在一起，一个channel的输入作为下一个channle的输出




```
func pipelineEx() {
    naturals := make(chan int)
    squares := make(chan int)

    // Counter
    go func() {
        for x := 0; x < 10; x++ {
            naturals <- x
        }
        close(naturals)
    }()

    // Squarer
    go func() {
        // chan int 也可以for range?!
        for x := range naturals {
            squares <- x * x
        }
        close(squares)
    }()

    // Printer
    for x := range squares {
        fmt.Println(x)
    }
}
```

channels不一定非得close，没用时会被垃圾回收器自动回收（~文件，每个打开的文件，不需要的时候都得close关闭)
但是不关闭可能会死锁！详见后面"带缓存的Channels"
重复关闭channels -> 异常!

单向Channel

有时候只需要发送 / 接收，即只需要一个方向的channel

`out chan<- int` 只发送， `int <-chan int` 只接收

```
// 刚才的Counter, Squarer, Printer写出函数
func counter(out chan<- int) { // 只发送
    for x := 0; x < 100; x++ {
        out <- x
    }
    close(out)
}

func squarer(out chan<- int, in <-chan int) { // out只发送，in只接收
    for v := range in {
        out <- v * v
    }
    close(out)
}

func printer(in <-chan int) { // in只接收
    for v := range in {
        fmt.Println(v)
    }
}
```

带缓存的Channels

有点像生产者-消费者 & 队列（buffered channels）了

```
ch := make(chan int, 3)
ch <- 1
ch <- 2
ch <- 3
// <- 1 2 3 <-
for x := range ch {
    fmt.Println(x) // 1 2 3
}
```

但是上面这个会死锁！why?

`for x := range ch`，会持续从通道读取，直到通道被显式关闭
没close导致的！阻塞（堵上了就死锁）

So, AC code:

```
ch := make(chan int, 3)
ch <- 1
ch <- 2
ch <- 3
close(ch) // 由发送方关闭channel
// <- 1 2 3 <-
for x := range ch {
    fmt.Println(x) // 1 2 3
}
```

是**队列(FIFO)**的模样~ 🥳

并发循环

循环创建多个goroutine?

wait

main goroutine只管创建不等待，自己跑完也不管了，等等孩子哇 😭

我真的得让你wait一下了 👺

`sync.WaitGroup`，为wait而生！ 😋

```go
func main() {
    values := []int{1, 2, 3, 4, 5}

    // 创建等待组
    var wg sync.WaitGroup

    for _, v := range values {
        wg.Add(1) // 为每个goroutine增加计数

        go func(val int) {
            defer wg.Done() // 完成后减少计数

            // 处理val
            fmt.Println(val * val)
        }(v) // 注意这里传递v的副本
    }

    wg.Wait() // 等待所有goroutine完成
}
```

咦？ `Add()`，怎么这么像 `signal()` 🤔；这 `Done()`，不分明是 `wait()` 吗😋（或者反一下）

这里的顺序还是乱的，只是成功让main稍微wait了一下，但还是win!🥳

error handling

整一个 `errChan`，make(chan error)

```go
func process(val int) (int, error) {
    if val == 0 {
        return 0, errors.New("invalid value")
    }
    return val * val, nil
}

func main() {
    values := []int{1, 2, 3, 0, 5}

    var wg sync.WaitGroup
    errChan := make(chan error, len(values))
    resultChan := make(chan int, len(values))

    for _, v := range values {
        wg.Add(1)

        go func(val int) {
            defer wg.Done()

            res, err := process(val)
            if err != nil {
                errChan <- err
                return
            }
            resultChan <- res
        }(v)
    }

    // 等待所有goroutine完成
    go func() {
        wg.Wait()
        close(resultChan)
        close(errChan)
    }()

    // 处理结果
    for res := range resultChan {
        fmt.Println("Result:", res)
    }

    // 处理错误
    for err := range errChan {
```

```
            fmt.Println("Error:", err)
        }
    }
```

select多路复用

`select` 语句，有点像 `switch` ， 可以根据不同channels的情况对应不同的操作

证件照：

```
select {
case <-ch1:
    // ...
case x := <-ch2:
    // ...use x...
case ch3 <- y:
    // ...
default:
    // ...
}
```

e.g. 火箭发射，如果5秒内按下回车键，则放弃发射(abort)；否则 起飞！！

```
func rocketLaunch() {
        abort := make(chan struct{})
        go func() {
                os.Stdin.Read(make([]byte, 1)) // 读取一个字节
                abort <- struct{}{}
        }()
        fmt.Println("Commencing countdown.  Press return to abort.")
    select {
    case <-time.After(5 * time.Second):
        // 如果是正常的倒计时
        // 继续等
    case <-abort:
        // 如果abort管道传来喜报
        // 直接寄
        fmt.Println("Launch aborted!")
        return
    }

        fmt.Println("起飞！！")
}
```

并发的退出

Go语言并没有提供在一个goroutine中终止另一个goroutine的方法，由于这样会导致goroutine之间的共享变量落在未定义的状态上。

那麻麻怎么让我回家吃饭呢？🤔

喊我一声就是咯😋

通道通知退出

通过关闭通道或发送信号来通知goroutine退出

就像刚才的abort通道一样，检测到回车发个消息，通知火箭该坠机了😋 😋

但其实channel close关闭时也会发出信号（广播机制），所以关闭通道也可以坠机的😋 😋 😋

基本模式

```
func worker(stopCh <-chan struct{}) {
    for {
        select {
        case <-stopCh:  // 收到退出信号
            fmt.Println("Worker exiting")
            return
        default:
            // 正常工作
            fmt.Println("Working...")
            time.Sleep(1 * time.Second)
        }
    }
}

func main() {
    stopCh := make(chan struct{})
    go worker(stopCh)

    // 运行5秒后停止
    time.Sleep(5 * time.Second)
    close(stopCh)  // 关闭通道通知退出

    // 等待worker退出
    time.Sleep(1 * time.Second)
}
```

多个goroutine同时退出

```
func main() {
    var wg sync.WaitGroup       // 1. 创建WaitGroup用于等待goroutine结束
    stopCh := make(chan struct{}) // 2. 创建退出信号通道

    // 3. 启动3个worker goroutine
    for i := 0; i < 3; i++ {
        wg.Add(1)  // 增加WaitGroup计数器
        // go func表示启动worker
        go func(id int) {
            defer wg.Done()  // goroutine结束时减少计数器

            // 4. 阻塞等待退出信号
            <-stopCh
            fmt.Printf("Worker %d exiting\n", id)
        }(i)  // 注意这里传递i的副本
    }

    // 5. 主goroutine等待2秒
    time.Sleep(2 * time.Second)

    // 6. 关闭通道通知所有worker退出
    close(stopCh)

    // 7. 等待所有worker完成
    wg.Wait()
}
```

context包

更现代的解决方案？！ 😳

有多现代？ 😋

（仅提一下，详情可询问八个字母的高人）

基本用法

```
func worker(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():  // 收到取消信号
            fmt.Println("Worker exiting:", ctx.Err())
            return
        default:
            // 正常工作
            fmt.Println("Working...")
            time.Sleep(1 * time.Second)
        }
    }
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    go worker(ctx)

    // 运行5秒后取消
    time.Sleep(5 * time.Second)
    cancel()  // 发送取消信号

    // 等待worker退出
    time.Sleep(1 * time.Second)
}
```

带超时的取消

```
func main() {
    // 设置3秒超时
    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()  // 确保资源释放

    go worker(ctx)

    // 等待worker退出
    time.Sleep(4 * time.Second)
}
```