**GoByExample**

# 变量

var a string = "initial"

var b, c int = 1, 2

# 常量

|  |
| --- |
| const s string = "constant" |

# For

|  |
| --- |
| for j := 7; j <= 9; j++ {  fmt.Println(j)  }  不带条件的 for 循环将一直执行，直到在循环体内使用了 break 或者 return 来跳出循环。 |

# 切片

l := s[2:5]

    // 开始的按照正常的就可以 结尾的要向前缩小一个的

    fmt.Println("sl1:", l)

# Map

Map的赋值是用{} 结果是用[]

struct赋值是用{} 结果是[]

n := map[*string*]*int*{"foo": 1, "bar": 2}

map: map[foo:1 bar:2]

# 闭包



# 递归



# 指针

因为int对应iptr，因为int是一个地址，所以iptr也是一地址；

我们传入的参数就是以地址&i；

函数里面：\*iptr=\*&i=0

我们对地址(iptr &i)进行赋值的，所以变量(\*iptr i)都会发生变化;

这是从根上地址发生的变化;

所以我们对属性加\*的目的就很清楚了，

让函数内外的变量一起变化。



# 结构体

sp是地址没错

赋值：但是我们是先对结构体赋值

之后再取地址

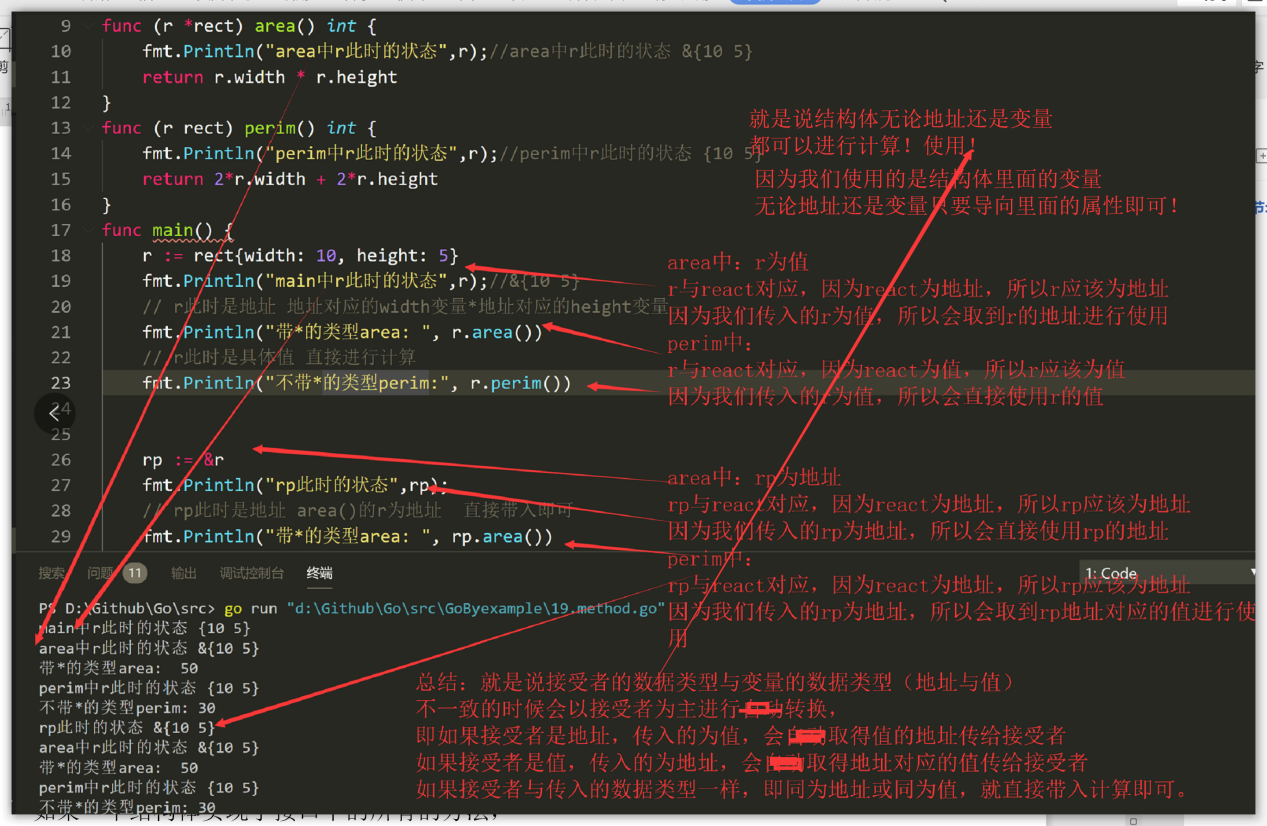
取值：

取得sp地址上对应的age变量

所以地址.属性 是可操作的



# 方法



# 接口

如果一个结构体实现了接口中的所有的方法，

我们就说这个结构体实现了这个接口，

这个结构体就与接口等价了，

因为结构体有赋值有具体方法(接口只声明方法名，没有方法体)

结构体就可以执行接口的方法了，

其实接口是一个引子，一个表象，

当满足条件（接口的所有方法，结构体都实现了）

两者就可以融合了。

凡是接口出现的地方都可以使用这个结构体的属性与方法进行替换

package main

import "fmt"

// import "math"

// 接口：一个事物多个方法

// 抽象：多个事物一个特性

// 在我们的例子中，我们将让 rect 和 circle 实现这个接口

type rect struct {

    width, height *float64*

}

type circle struct {

    radius *float64*

}

// 这里是一个几何体的基本接口。

// 接口全部是方法

// 结构体是数据类型的集合

// 接口是method的集合 再细分的话就是 有对象的func的集合

type geometry interface {

    // 面积

    area() *float64*

    // 周长

    perim() *float64*

}

// 要在 Go 中实现一个接口，我们只需要实现接口中的所有方法。这里我们让 rect 实现了 geometry 接口。

func (r rect) area() *float64* {

    return r.width \* r.height

}

func (r rect) perim() *float64* {

    return 2\*r.width + 2\*r.height

}

// 如果一个变量的是接口类型，那么我们可以调用这个被命名的接口中的方法。

// 这里有一个一通用的 measure 函数，利用这个特性，它可以用在任何 geometry 上。

// 集合了两种方法的方法

//

// 接口也是一种类型geometry 对应的对象就是g

//

func measure(g geometry) {

    fmt.Println(g)

    fmt.Println(g.area())

    fmt.Println(g.perim())

}

func main() {

    r := rect{width: 3, height: 4}

    // 结构体类型 circle 和 rect 都实现了 geometry接口，所以我们可以使用它们的实例作为 measure 的参数。

    measure(r)

}

// {3 4}

// 12

// 14

// {5}

// 78.53981633974483

// 31.41592653589793

# Errors

        // errors.New 构造一个使用给定的错误信息的基本error 值。

        return -1, errors.New("can't work with 42")

# 协程

    go func(msg *string*) {

        fmt.Println(msg)

    }("going")

# 通道

channel与goroutine必须并存

如果只有channel，根本无法运行，因为全部死锁了

如果只有goroutine，协程根本不会执行，因为没时间，主程序已经结束了

因为有了堵塞，协程才会执行

# 通道方向

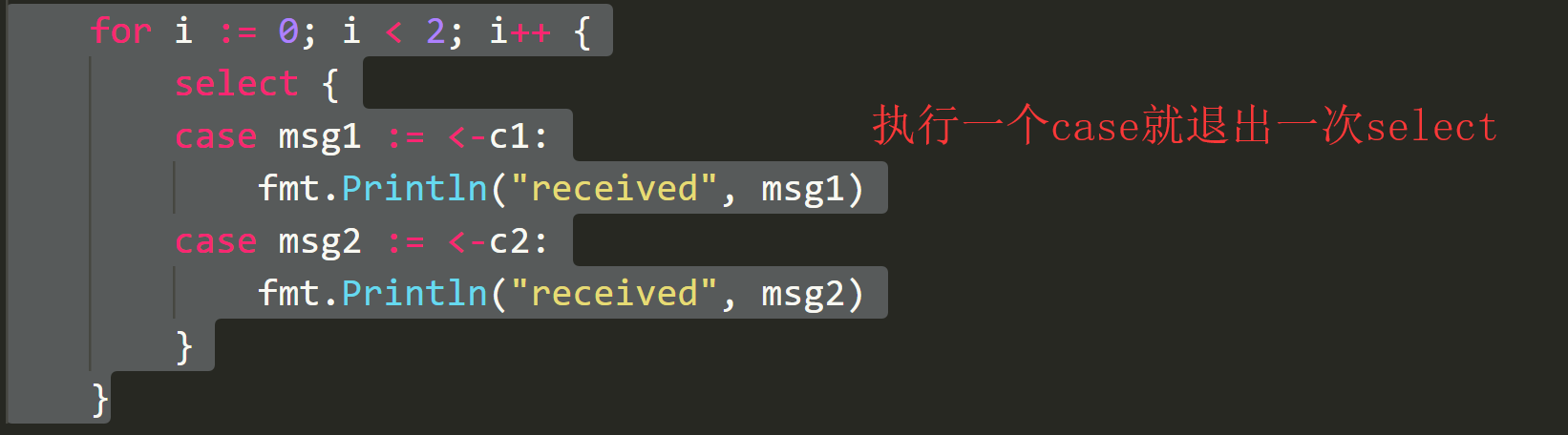
Chan与变量进行对应



# 通道选择器

Go 的通道选择器 让你可以同时等待多个通道操作。Go 协程和通道以及选择器的结合是 Go 的一个强大特性。

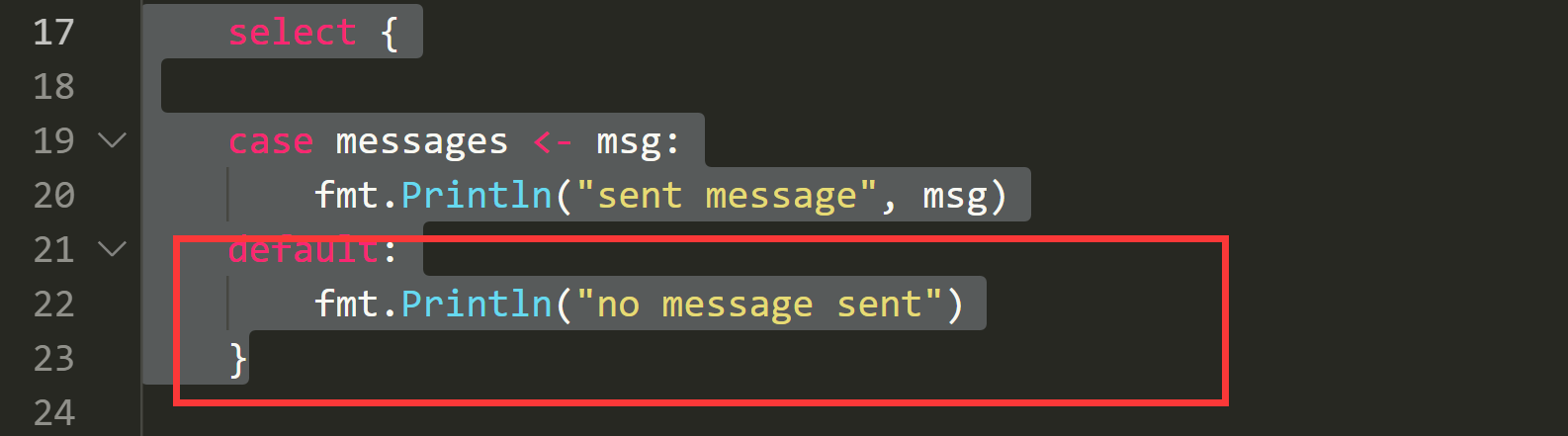
执行一个case就退出一次select



# Timeouts



# 默认通道



# 通道的关闭

关闭 一个通道意味着不能再向这个通道发送值了。这个特性可以用来给这个通道的接收方传达工作已经完成的信息。

**// 关闭通道的时候 读取通道的时候会读取到false**

    close(jobs)

# 遍历通道

1.close之后的通道也是可以遍历到的，这个遍历就相当于同步的读取channel

2.通道是没有index的

3.如果没有close的话 会堵塞 死锁 因为读写缺少一个并且没有协程 所以死锁**（读写通道与协程缺少一个便会死锁）**

**这个被遍历的通道只能是被写入的通道**



# 定时器

<-timer1.C 直到这个定时器的通道 C 明确的发送了定时器失效的值之前，将一直阻塞。（理解，timer1.C就是一个写入通道，时间到之后进行读通道,就不堵塞了）

定时器的新建 timer1 := time.NewTimer(time.Second \* 2)

定时器的执行 <-timer1.C 没到时间会堵塞，时间到了就可以执行我们动作

定时器的停止 stop2 := timer2.Stop()

定时器的休息 time.Sleep(time.Second \* 2)

# 打点器

定时器：某个时间点执行一次

打点器：每个多久就执行一次

打点器的新建： ticker := time.NewTicker(time.Millisecond \* 500)

打点器的停止： ticker.Stop()

打点器的执行：

go func() {

for t := range ticker.C {

fmt.Println("Tick at", t)

}

}()

# 工作池

关闭 一个通道意味着不能再向这个通道发送值了。这个特性可以用来给这个通道的接收方传达工作已经完成的信息。说数据已经传送完毕了

Worker就是消费者，取出job的数据，进行操作，可以创建n个消费者

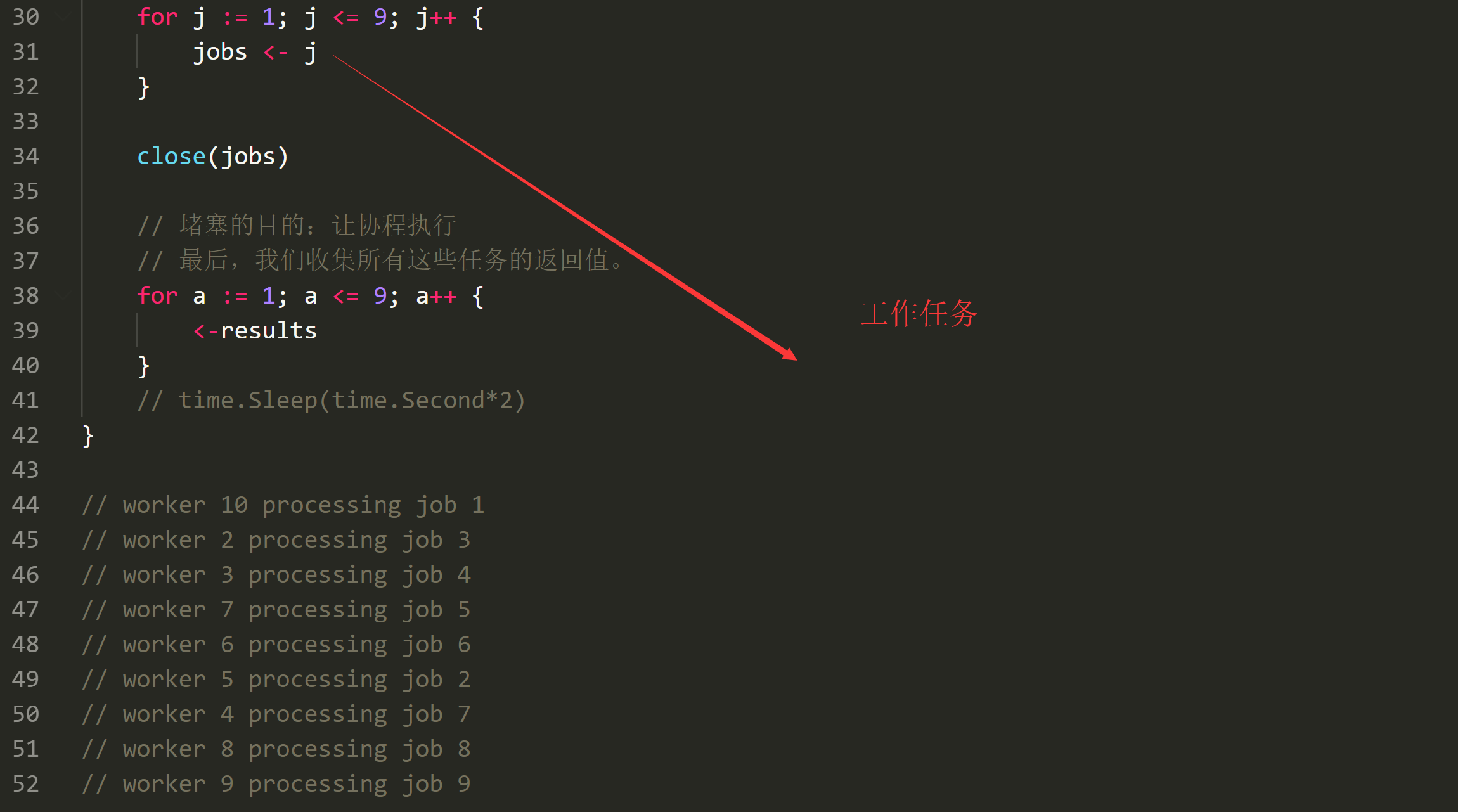
w消费者个数，表示消费者的序号

Jobs任务的载体

Results控制堵塞

j是生产者，把数据放入到job的channel中

两者通过job通道进行通信，只要job通道有值，就进行生产操作



# 死锁

**协程里面可以堵塞，堵塞了不会报错**

**主函数里面堵塞的，必须有协程进行读写通道，否则报错：所有的协程睡着了，死锁**