# 1. Go语言的优点

\* 并发是语言的一部分（译注：并非通过标准库实现），所以编写多线程程序会是一件很容易的事。后续教程将会讨论到，并发是通过 Goroutines 和 channels 机制实现的。

\* Golang 是一种编译型语言。源代码会编译为二进制机器码。而在解释型语言中没有这个过程，如 Nodejs 中的 JavaScript。

\* 语言规范十分简洁。所有规范都在一个页面展示，你甚至都可以用它来编写你自己的编译器呢 :)

\* Go 编译器支持静态链接。所有 Go 代码都可以静态链接为一个大的二进制文件（译注：相对现在的磁盘空间，其实根本不大），并可以轻松部署到云服务器，而不必担心各种依赖性。

# 2. UTF-8 和 Unicode 有何区别？

Unicode 与 ASCII 类似，都是一种字符集。字符集为每个字符分配一个唯一的 ID，无论任何情况下，Unicode 中的字符的 ID 都是不会变化的。Unicode字符集的编号从0一直算到了100多万（三个字节）。每一个区间都对应着一种语言的编码。目前几乎收纳了全世界大部分的字符。所有的字符都有唯一的编号，这就解决了解码的冲突。但是，unicode把大家都归纳进来，却没有为编码的二进制传输和二进制解码做出规定。于是，就出现了如下解决方案：uft-8，utf-16，utf-32这些编码方案。utf-16是用两个字节来编码所有的字符，utf-32则选择用4个字节来编码。utf-8为了节省资源，采用变长编码。

UTF-8 是编码规则，将 Unicode 中字符的 ID 以某种方式进行编码，UTF-8 是一种变长编码规则，根据RFC-2279 1每个字符的最大字节数为6，根据RFC3629 2，每个字符的最大字节数为4。由于许多UTF8实现从未实现超过4个字节的序列（并且只有3个，例如MySQL），所以认为将每个字符限制为4个字节是安全的，我们普遍认为UTF-8最大表示4个字节长度。Unicode与UTF-8对应表如下：



跟据上表，解读UTF-8编码非常简单。如果一个字节的第一位是0，则这个字节单独就是一个字符；如果第一位是1，则连续有多少个1，就表示当前字符占用多少个字节。

# 3. switch语句

\* case不允许出现重复项，否则编译器会报错误。

\*当所有的case语句都不匹配时，会执行default语句中的内容，default 不一定只能出现在 switch 语句的最后，它可以放在 switch 语句的任何地方。

\*一个case语句后面可以有多个匹配的表达式：

func main() {

letter := "i"

switch letter {

case "a", "e", "i", "o", "u": // 一个选项多个表达式

fmt.Println("vowel")

default:

fmt.Println("not a vowel")

}

}

\* 无表达式的switch语句等同于switch true，并且每个 case 表达式都被匹配。

\*在 Go 中，每执行完一个 case 后，会从 switch 语句中跳出来，不再做后续 case 的判断和执行。使用 fallthrough 语句可以在已经执行完成的 case 之后，把控制权转移到下一个 case 的执行代码中。

\* fallthrough 语句应该是 case 子句的最后一个语句。如果它出现在了 case 语句的中间，编译器将会报错。

# 4. 数组

\*Go 中的数组是值类型而不是引用类型。这意味着当数组赋值给一个新的变量时，该变量会得到一个原始数组的一个副本。如果对新变量进行更改，则不会影响原始数组。

\*数组的长度时固定的，在编译期间就要确定，否则报错。

# 5. 切片

切片是由数组建立的一种方便、灵活且功能强大的包装（Wrapper）。切片本身不拥有任何数据。它们只是对现有数组的引用。

切片自己不拥有任何数据。它只是底层数组的一种表示。对切片所做的任何修改都会反映在底层数组中。

切片是引用类型。

# 6. 可变参数的函数

有一个可以直接将切片传入可变参数函数的语法糖，你可以在在切片后加上 ... 后缀。如果这样做，切片将直接传入函数，不再创建新的切片：

nums := []int{89, 90, 95}

find(89, nums...)

# 7. map

map 的零值是 nil。如果你想添加元素到 nil map 中，会触发运行时 panic。因此 map 必须使用 make 函数初始化。

如果获取一个不存在的元素， map 会返回该元素类型的零值。

当使用 for range 遍历 map 时，不保证每次执行程序获取的元素顺序相同。

删除 map 中 key 的语法是 delete(map, key)。这个函数没有返回值。

获取 map 的长度使用 len 函数。

map 是引用类型。当 map 被赋值为一个新变量的时候，它们指向同一个内部数据结构。因此，改变其中一个变量，就会影响到另一变量。

map 之间不能使用 == 操作符判断，== 只能用来检查 map 是否为 nil。

# 8.字符串

Go 中的字符串是兼容 Unicode 编码的，并且使用 UTF-8 进行编码。

Go 中的字符串是不可变的。一旦一个字符串被创建，那么它将无法被修改。

Go 语言中的字符串是一个字节切片。把内容放在双引号""之间，我们可以获取字符串的每一个字节。

len(string) 函数返回字符串中字节的数量。

在 UTF-8 编码中，一个代码点可能会占用超过一个字节的空间。那么我们该怎么办呢？rune 能帮我们解决这个难题。rune 是 Go 语言的内建类型，它也是 int32 的别称。在 Go 语言中，rune 表示一个代码点。代码点无论占用多少个字节，都可以用一个 rune 来表示。utf8 package 包中的 func RuneCountInString(s string) (n int) 方法用来获取字符串的长度。这个方法传入一个字符串参数然后返回字符串中的 rune 的数量。

# 9.指针

指针是一种存储变量内存地址（Memory Address）的变量。

& 操作符用于获取变量的地址。

指针的零值是 nil。

指针的解引用可以获取指针所指向的变量的值。将 a 解引用的语法是 \*a。

不要向函数传递数组的指针，而应该使用切片。

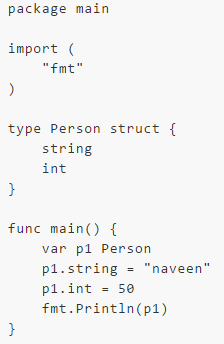
Go 不支持指针运算。

# 10.结构体

结构体是值类型。如果它的每一个字段都是可比较的，则该结构体也是可比较的。如果两个结构体变量的对应字段相等，则这两个变量也是相等的。

如果结构体包含不可比较的字段，则结构体变量也不可比较。

虽然匿名字段没有名称，但其实匿名字段的名称就默认为它的类型。如下所示：



如果是结构体中有匿名的结构体类型字段，则该匿名结构体里的字段就称为提升字段。这是因为提升字段就像是属于外部结构体一样，可以用外部结构体直接访问。



# 11.方法

方法其实就是一个函数，在 func 这个关键字和方法名中间加入了一个特殊的接收器类型。接收器可以是结构体类型或者是非结构体类型。接收器是可以在方法的内部访问的。

为什么我们需要方法：

Go 不是纯粹的面向对象编程语言，而且Go不支持类。因此，基于类型的方法是一种实现和类相似行为的途径。

相同的名字的方法可以定义在不同的类型上，而相同名字的函数是不被允许的。

值接收器和指针接收器之间的区别在于，在指针接收器的方法内部的改变对于调用者是可见的，然而值接收器的情况不是这样的。

为了在一个类型上定义一个方法，方法的接收器类型定义和方法的定义应该在同一个包中。

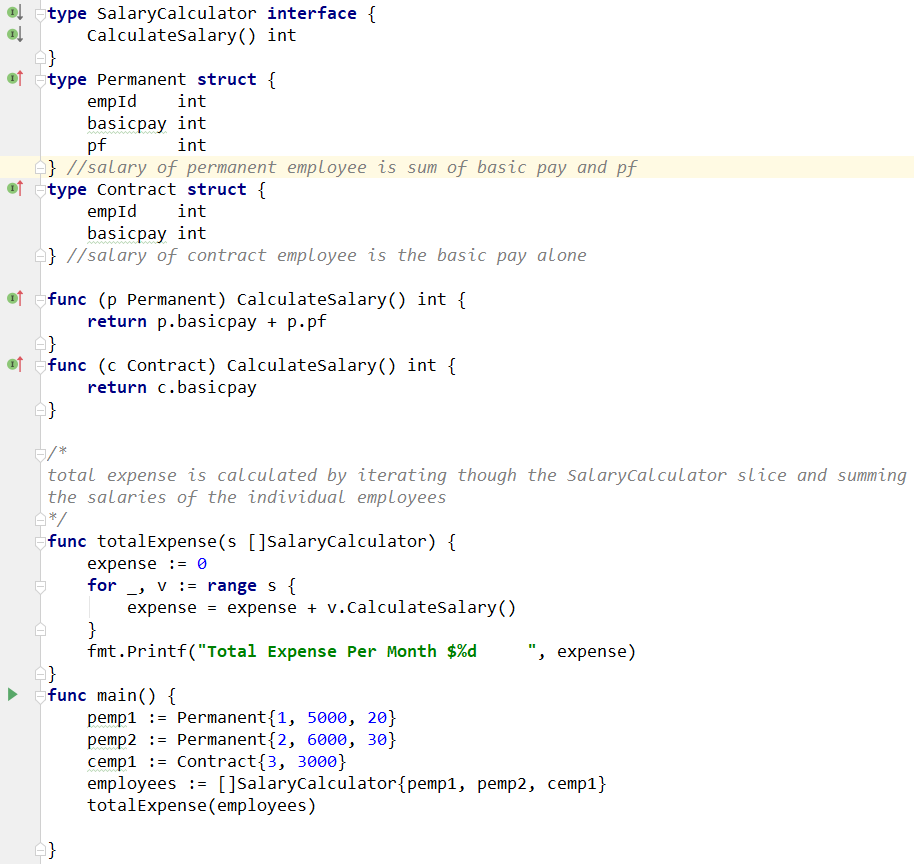
# 12.接口

在 Go 语言中，接口就是方法签名（Method Signature）的集合。

接口指定了一个类型应该具有的方法，并由该类型决定如何实现这些方法。

接口的零值是nil。对于值为nil的接口，其底层值（Underlying Value）和具体类型（Concrete Type）都为nil。

**接口的实际用途（示例）：**



这样做最大的优点是：totalExpense 可以扩展新的员工类型，而不需要修改任何代码。假如公司增加了一种新的员工类型 Freelancer，它有着不同的薪资结构。Freelancer只需传递到 totalExpense 的切片参数中，无需 totalExpense 方法本身进行修改。只要 Freelancer 也实现了 SalaryCalculator 接口，totalExpense 就能够实现其功能。

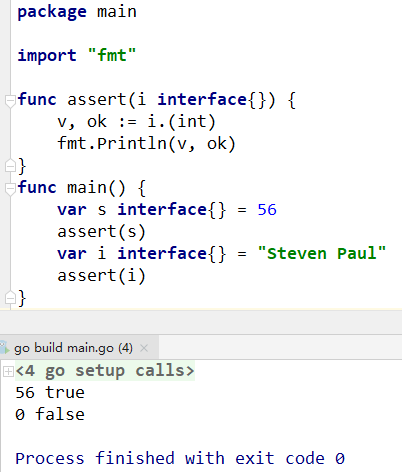
**类型断言：**

类型断言用于提取接口的底层值（Underlying Value），在语法 i.(T) 中，接口 i 的具体类型是 T，该语法用于获得接口的底层值。

语法：v, ok := i.(T)

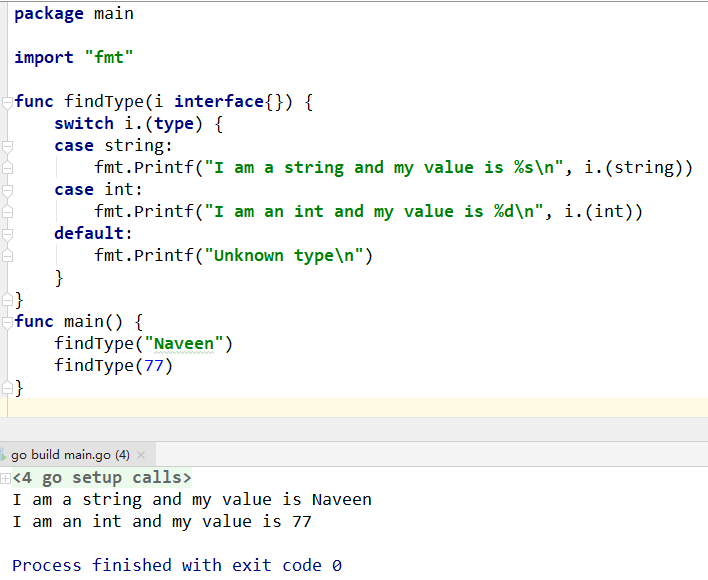
如果 i 的具体类型是 T，那么 v 赋值为 i 的底层值，而 ok 赋值为 true。

如果 i 的具体类型不是 T，那么 ok 赋值为 false，v 赋值为 T 类型的零值，



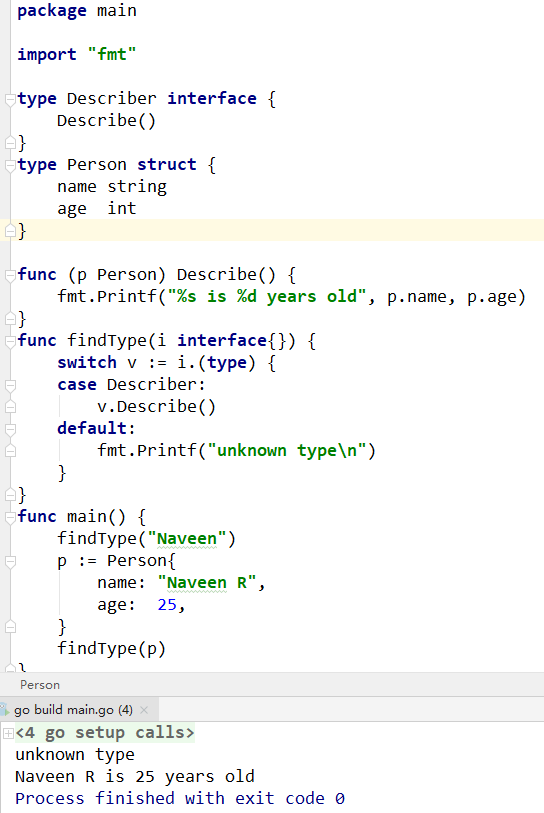
**类型选择：**

类型选择的语法类似于类型断言。类型断言的语法是 i.(T)，而类型选择的语法是 i.(type)。



**类型和接口比较：**

如果一个类型实现了接口，那么该类型与其实现的接口就可以互相比较。



# 13.并发

Go 是并发式语言，而不是并行式语言。

**并发是什么**：

并发是指立即处理多个任务的能力。

**并行是什么：**

并行是指同一时刻处理多个任务。

并行不一定会加快运行速度，因为并行运行的组件之间可能需要相互通信。在多核的并行系统上，组件间的通信开销就很高。

Go 编程语言原生支持并发。Go 使用 Go 协程（Goroutine） 和信道（Channel）来处理并发。

# 14.协程

Go 协程是与其他函数或方法一起并发运行的函数或方法。Go 协程可以看作是轻量级线程。与线程相比，创建一个 Go 协程的成本很小。因此在 Go 应用中，常常会看到有数以千计的 Go 协程并发地运行。

**Go 协程相比于线程的优势：**

相比线程而言，Go 协程的成本极低。堆栈大小只有若干 kb，并且可以根据应用的需求进行增减。而线程必须指定堆栈的大小，其堆栈是固定不变的。

Go 协程会复用（Multiplex）数量更少的 OS 线程。即使程序有数以千计的 Go 协程，也可能只有一个线程。如果该线程中的某一 Go 协程发生了阻塞（比如说等待用户输入），那么系统会再创建一个 OS 线程，并把其余 Go 协程都移动到这个新的 OS 线程。所有这一切都在运行时进行，作为程序员，我们没有直接面临这些复杂的细节，而是用简洁的 API 来处理并发。

Go 协程使用信道（Channel）来进行通信。信道用于防止多个协程访问共享内存时发生竞态条件（Race Condition）。信道可以看作是 Go 协程之间通信的管道。

启动一个新的协程时，协程的调用会立即返回。与函数不同，程序控制不会去等待 Go 协程执行完毕。在调用 Go 协程之后，程序控制会立即返回到代码的下一行，忽略该协程的任何返回值。

如果希望运行其他 Go 协程，Go 主协程必须继续运行着。如果 Go 主协程终止，则程序终止，于是其他 Go 协程也不会继续运行。

# 15.无缓冲channel

信道是 Go 协程之间的一种通信方式。

所有信道都关联了一个类型。信道只能运输这种类型的数据，而运输其他类型的数据都是非法的。

信道的零值为 nil。信道的零值没有什么用，应该像对 map 和切片所做的那样，用 make 来定义信道：

var a chan int

a = make(chan int)

信道的发送和接收：

data := <- a // 读取信道 a

a <- data // 写入信道 a

单向信道：

只能往信道里发送数据的信道（唯送信道）：

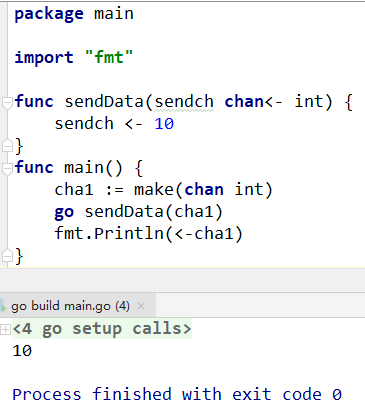
ch = make(chan<- int)

只能从信道里取数据的信道（唯收信道）：

ch = make(<-chan int)

单向信道有什么意义：

这就需要用到信道转换（Channel Conversion）了。把一个双向信道转换成单向信道是行得通的，但是反过来就不行。



死锁：

使用信道需要考虑的一个重点是死锁。当 Go 协程给一个信道发送数据时，照理说会有其他 Go 协程来接收数据。如果没有的话，程序就会在运行时触发 panic，形成死锁。

同理，当有 Go 协程等着从一个信道接收数据时，我们期望其他的 Go 协程会向该信道写入数据，要不然程序就会触发 panic。

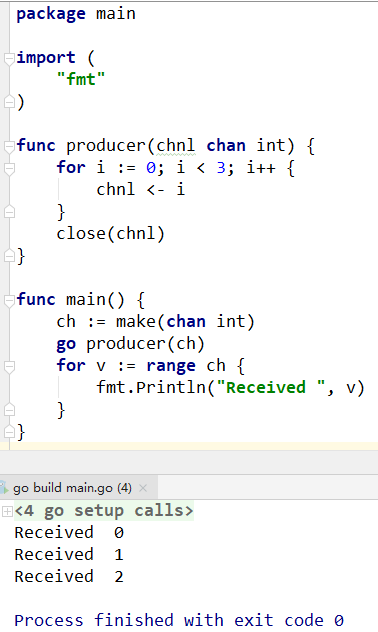
关闭信道和使用 for range 遍历信道:

数据发送方可以关闭信道，通知接收方这个信道不再有数据发送过来。当从信道接收数据时，接收方可以多用一个变量来检查信道是否已经关闭。

v, ok := <- ch

上面的语句里，如果成功接收信道所发送的数据，那么 ok 等于 true。而如果 ok 等于 false，说明我们试图读取一个关闭的通道。从关闭的信道读取到的值会是该信道类型的零值。例如，当信道是一个 int 类型的信道时，那么从关闭的信道读取的值将会是 0。

for range 循环用于在一个信道关闭之前，从信道接收数据。



# 16.有缓冲channel

有缓冲（Buffer）的信道。只在缓冲已满的情况，才会阻塞向缓冲信道（Buffered Channel）发送数据。同样，只有在缓冲为空的时候，才会阻塞从缓冲信道接收数据。

通过向 make 函数再传递一个表示容量的参数（指定缓冲的大小），可以创建缓冲信道。

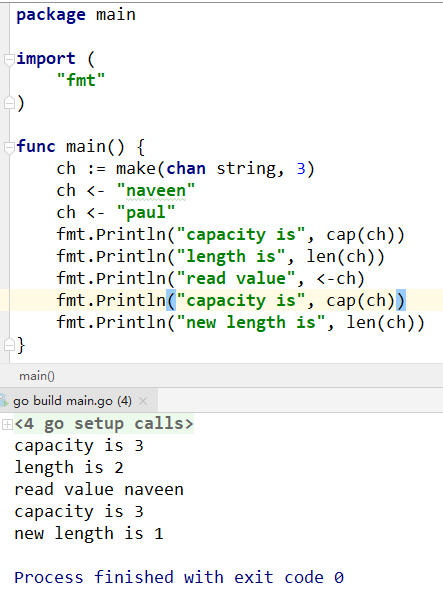
ch := make(chan type, capacity)

长度和容量：

缓冲信道的容量是指信道可以存储的值的数量。我们在使用 make 函数创建缓冲信道的时候会指定容量大小。

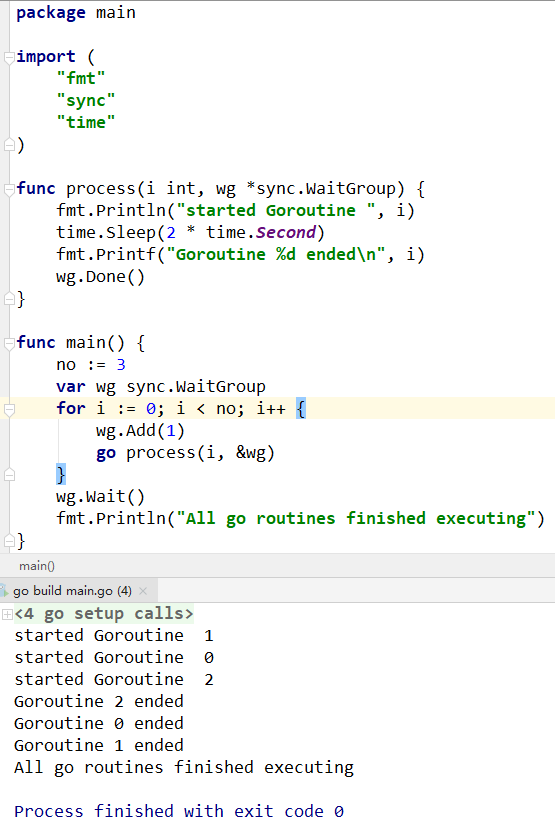
缓冲信道的长度是指信道中当前排队的元素个数。

获取信道容量用cap(ch)函数，获取信道长度用len(ch)函数。



# 17. WaitGroup和工作池

WaitGroup 用于等待一批 Go 协程执行结束。



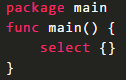
WaitGroup 是一个结构体类型，传递 wg 的地址是很重要的。如果没有传递 wg 的地址，那么每个 Go 协程将会得到一个 WaitGroup 值的拷贝，因而当它们执行结束时，main 函数并不会知道。

# 18.select

select 语句用于在多个发送/接收信道操作中进行选择。select 语句会一直阻塞，直到发送/接收操作准备就绪。如果有多个信道操作准备完毕，select 会随机地选取其中之一执行。该语法与 switch 类似，所不同的是，这里的每个 case 语句都是信道操作。

在没有 case 准备就绪时，可以执行 select 语句中的默认情况（Default Case）。这通常用于防止 select 语句一直阻塞。

空 select:



除非有 case 执行，select 语句就会一直阻塞着。在这里，select 语句没有任何 case，因此它会一直阻塞，导致死锁。该程序会触发 panic。

# 19.mutex

Mutex 用于提供一种加锁机制（Locking Mechanism），可确保在某时刻只有一个协程在临界区运行，以防止出现竞态条件。

Mutex 是一个结构体类型。

Mutex 可以在 sync 包内找到。Mutex 定义了两个方法：Lock 和 Unlock。所有在 Lock 和 Unlock 之间的代码，都只能由一个 Go 协程执行，于是就可以避免竞态条件。

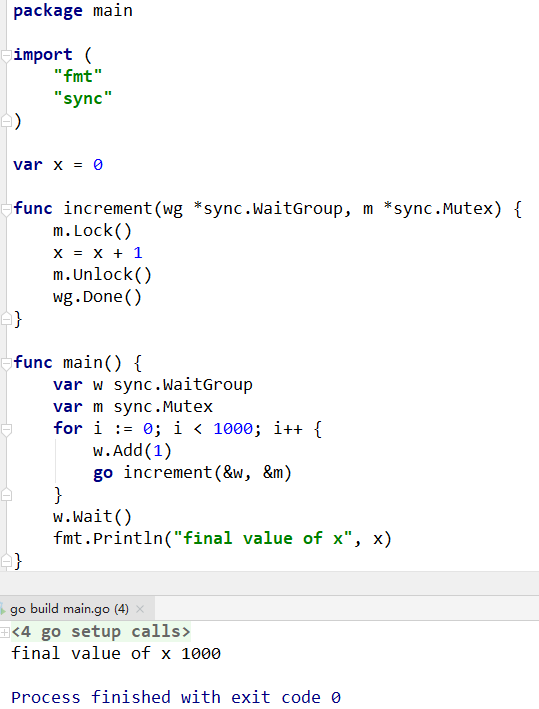
mutex.Lock()

x = x + 1

mutex.Unlock()

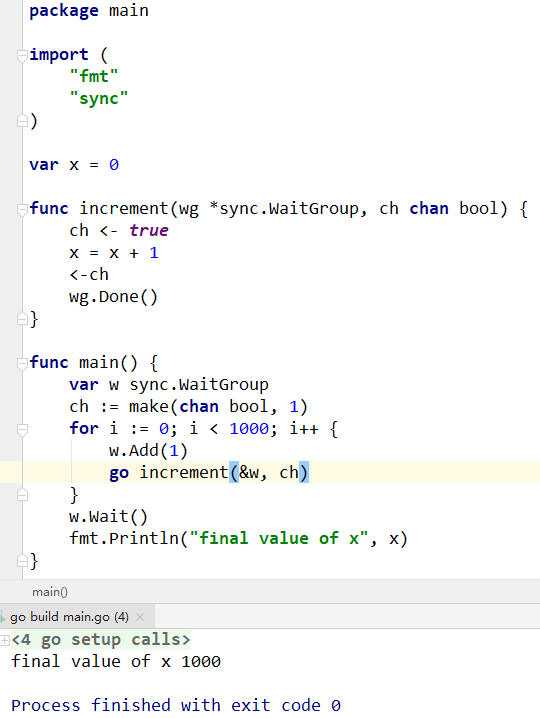
如果有一个 Go 协程已经持有了锁（Lock），当其他协程试图获得该锁时，这些协程会被阻塞，直到 Mutex 解除锁定为止。

使用 Mutex处理竞态条件:



传递 Mutex 的地址很重要。如果传递的是 Mutex 的值，而非地址，那么每个协程都会得到 Mutex 的一份拷贝，竞态条件还是会发生。

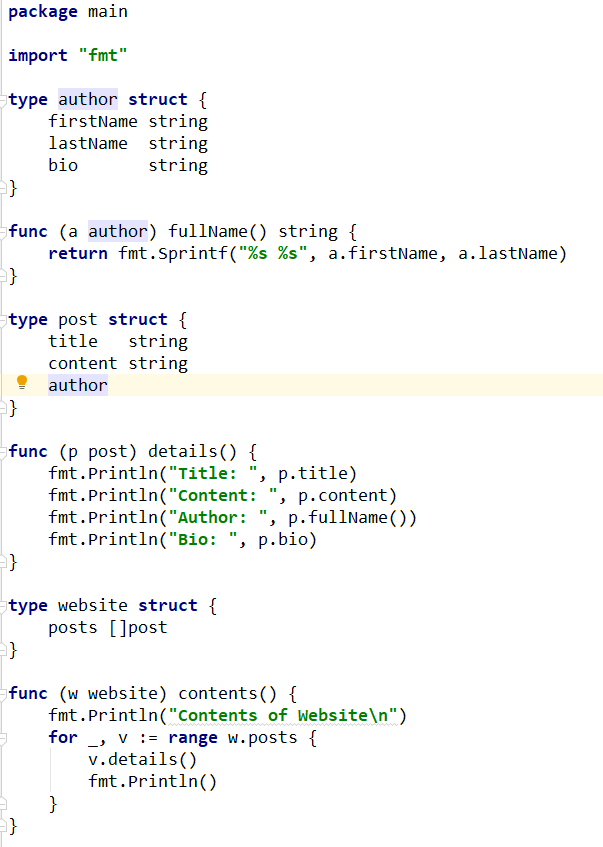
使用信道处理竞态条件：

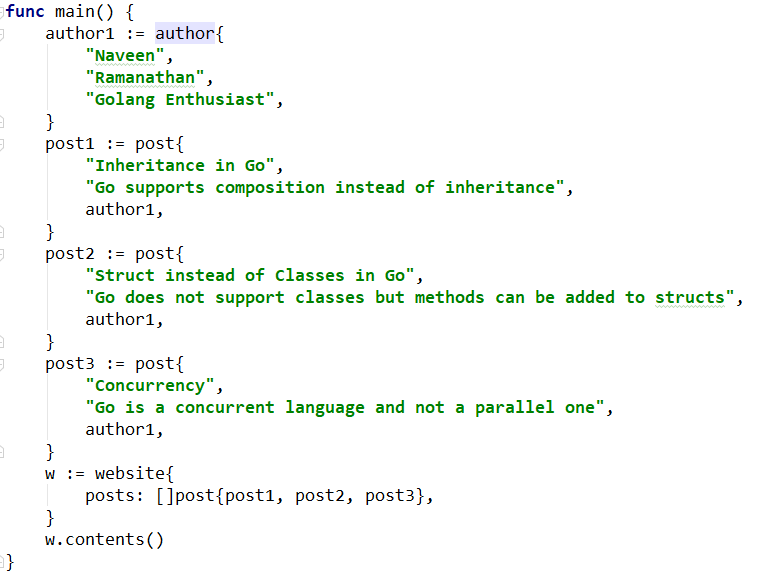


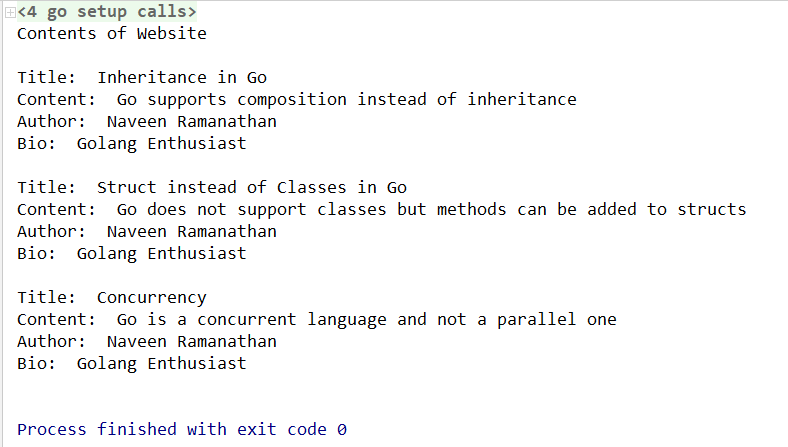
# 20.结构体

结构体嵌套和结构体切片的嵌套：

结构体不能嵌套一个匿名结构体切片，必须要给结构体切片取一个字段名。但是可以嵌套匿名非结构体切片。

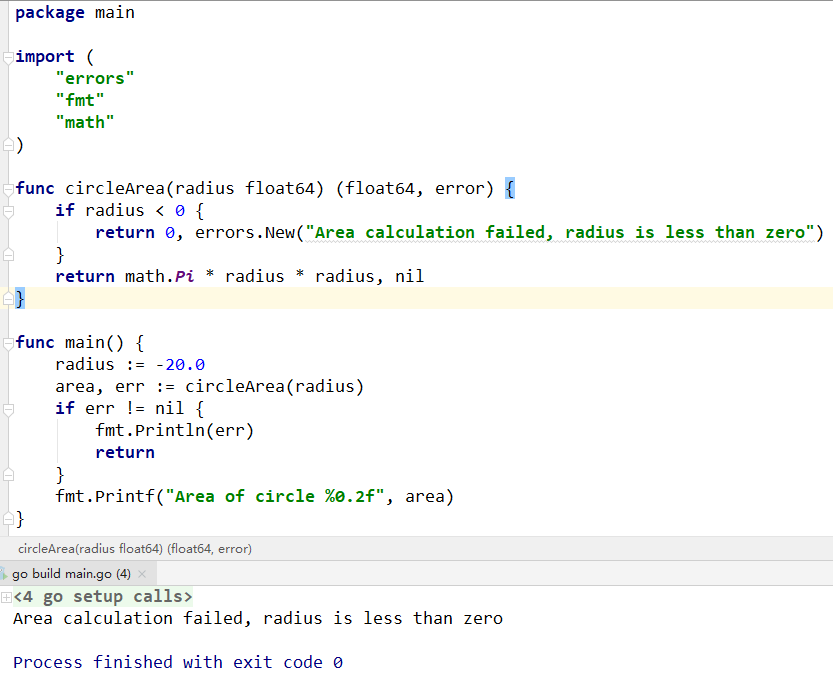




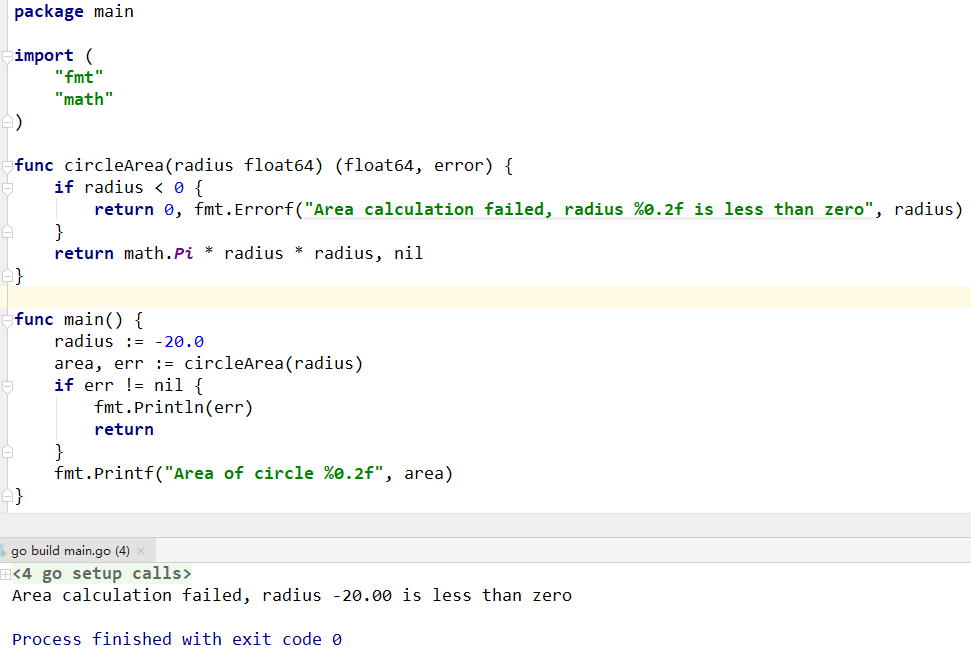


# 21.错误处理

使用 New 函数创建自定义错误：

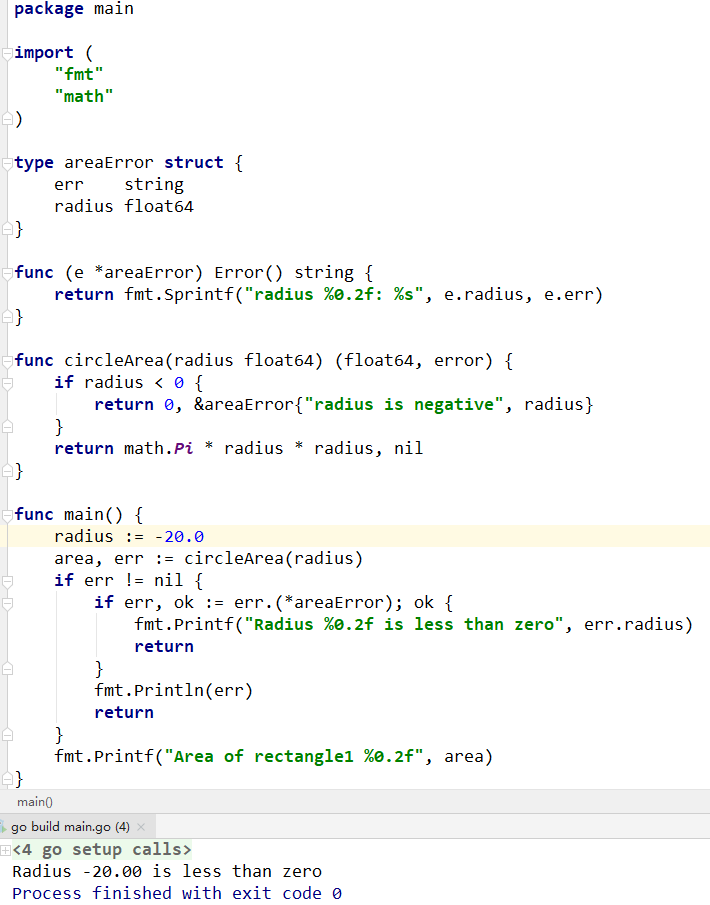


使用 Errorf 给错误添加更多信息：

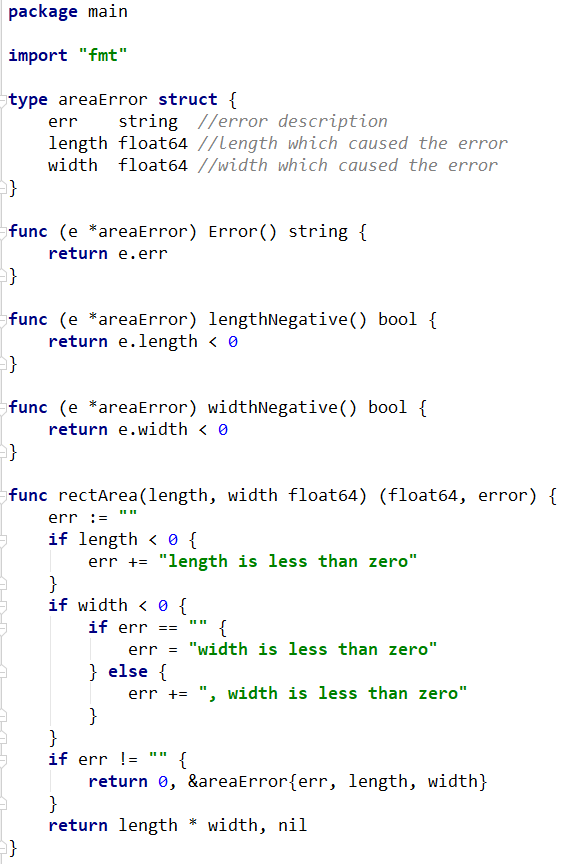


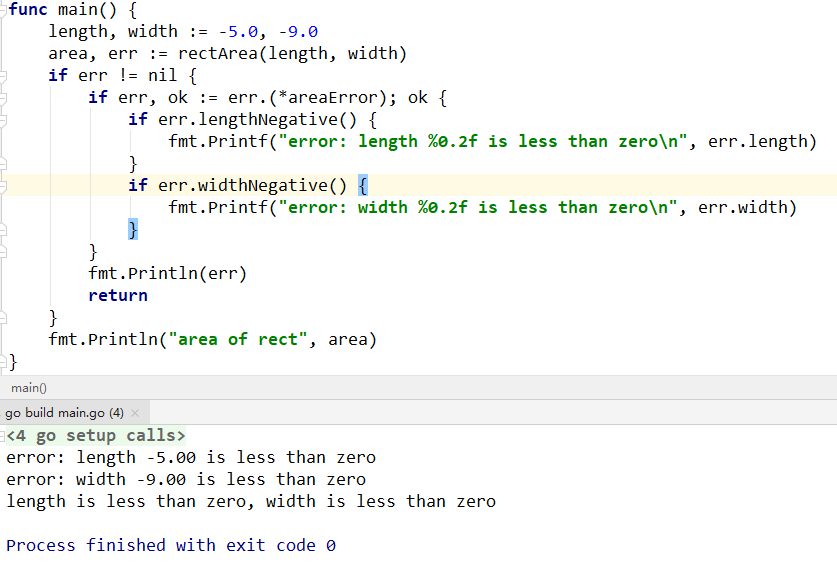
使用结构体类型的字段提供错误的更多信息：

错误还可以用实现了 error 接口的结构体来表示。这种方式可以更加灵活地处理错误。



使用结构体类型的方法来提供错误的更多信息：



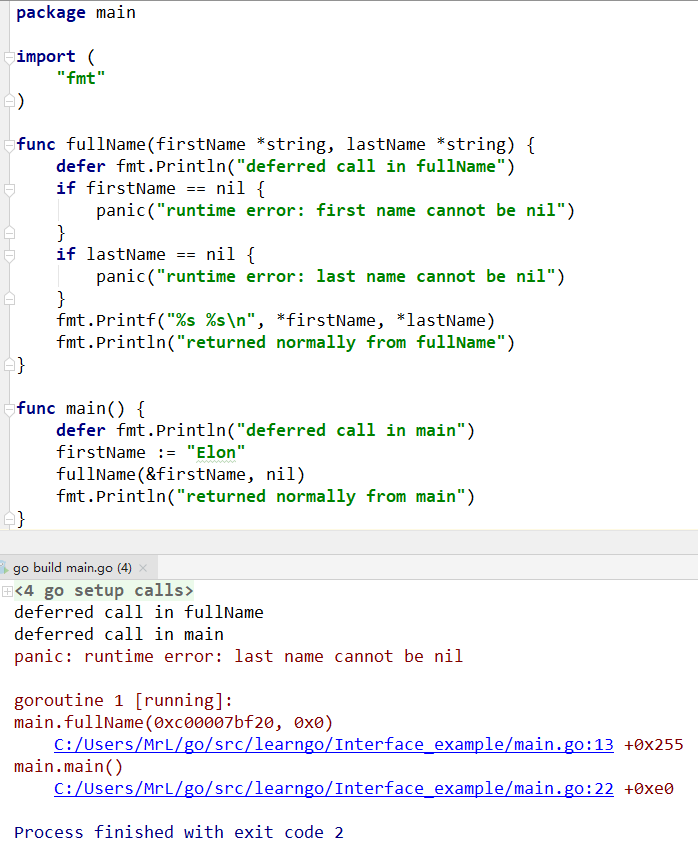


22. panic和recover

在有些情况，当程序发生异常时，无法继续运行，在这种情况下，我们会使用 panic 来终止程序。当函数发生 panic 时，它会终止运行，在执行完所有的延迟函数后，程序控制返回到该函数的调用方。这样的过程会一直持续下去，直到当前协程的所有函数都返回退出，然后程序会打印出 panic 信息，接着打印出堆栈跟踪（Stack Trace），最后程序终止。

内建函数 panic 的签名：func panic(interface{})

panic 示例：

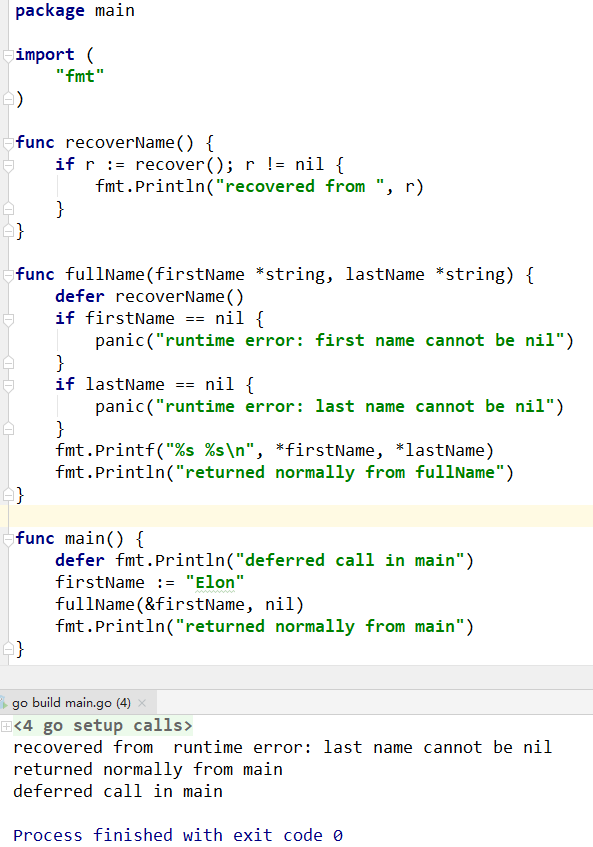


recover 是一个内建函数，用于重新获得 panic 协程的控制。

recover 函数的标签为：func recover() interface{}

只有在defer函数的内部，调用 recover 才有用。在defer函数内调用 recover，可以取到 panic 的错误信息，并且停止 panic 续发事件（Panicking Sequence），使程序运行恢复正常。如果在defer函数的外部调用 recover，就不能停止 panic 续发事件。

recover 示例：



只有在相同的 Go 协程中调用 recover 才管用。recover 不能恢复一个不同协程的 panic。

程序recover后如果还想打印堆栈信息，可以使用"runtime/debug"包中的debug.PrintStack()方法。如下：

func r() {

if r := recover(); r != nil {

fmt.Println("Recovered", r)

debug.PrintStack()

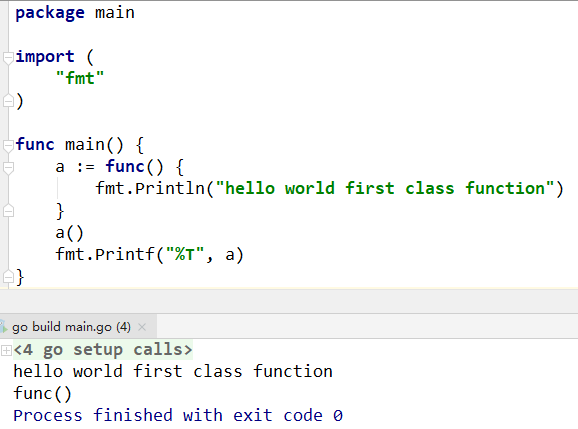
}

}

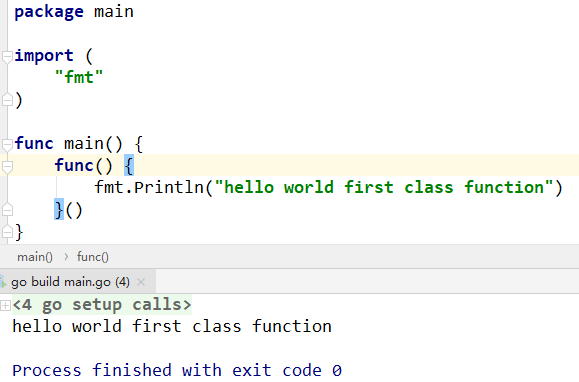
# 22.头等函数

支持头等函数（First Class Function）的编程语言，可以把函数赋值给变量，也可以把函数作为其它函数的参数或者返回值。Go 语言支持头等函数的机制。

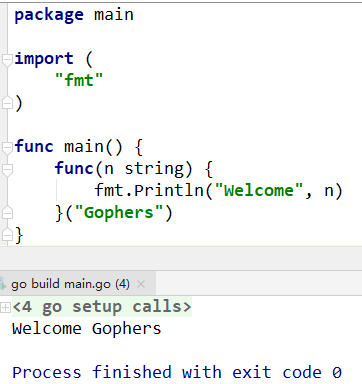
匿名函数赋给变量调用：



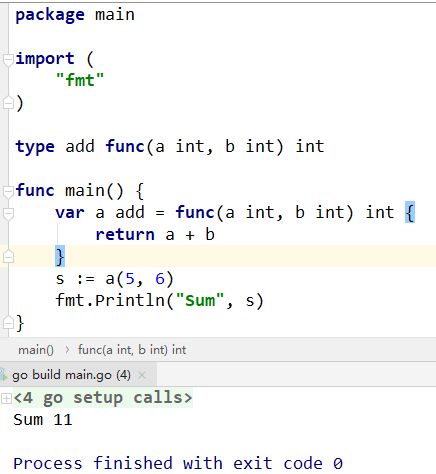
匿名函数直接调用：



匿名函数传递参数：



用户自定义的函数类型：



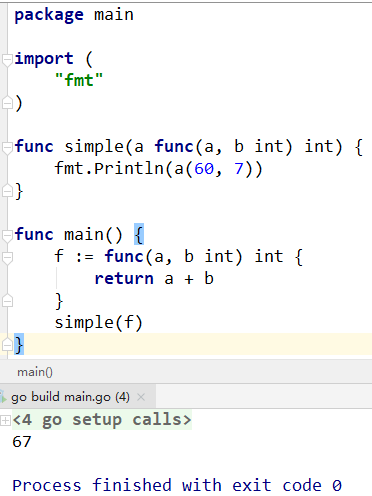
高阶函数：

wiki 把高阶函数（Hiher-order Function）定义为：满足下列条件之一的函数：

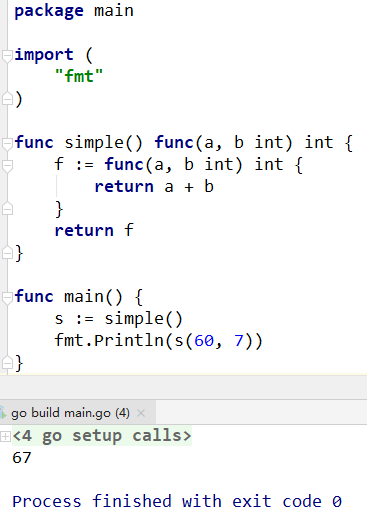
\*接收一个或多个函数作为参数

\*返回值是一个函数

把函数作为参数，传递给其它函数：

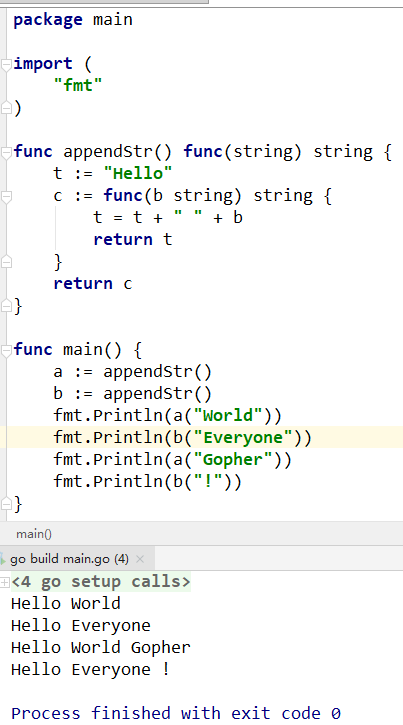


在其它函数中返回函数：



闭包：

闭包（Closure）是匿名函数的一个特例。当一个匿名函数所访问的变量定义在函数体的外部时，就称这样的匿名函数为闭包。



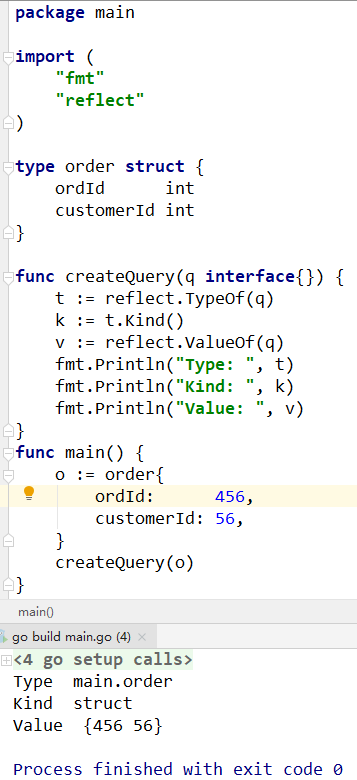
# 23.反射

反射就是程序能够在运行时检查变量和值，求出它们的类型。

在 Go 语言中，reflect 实现了运行时反射。

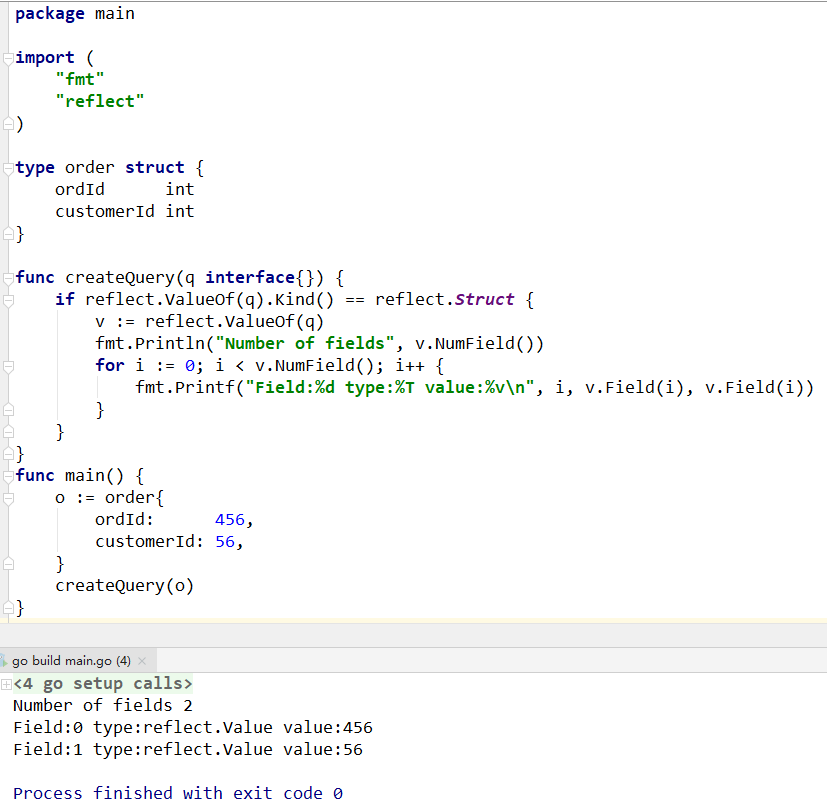
reflect.Type 、reflect.Value、reflect.Kind：

reflect.Type 表示 interface{} 的具体类型，而 reflect.Value 表示它的具体值。reflect.TypeOf() 和 reflect.ValueOf() 两个函数可以分别返回 reflect.Type 和 reflect.Value，reflect.Kind表示具体类型的特定类型。



NumField() 和 Field() 方法：

NumField() 方法返回结构体中字段的数量，而 Field(i int) 方法返回字段 i 的 reflect.Value。

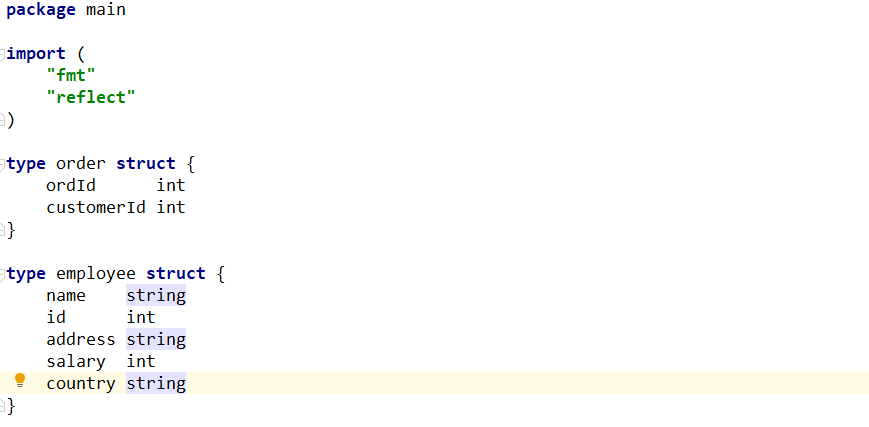


Int() 和 String() 方法：

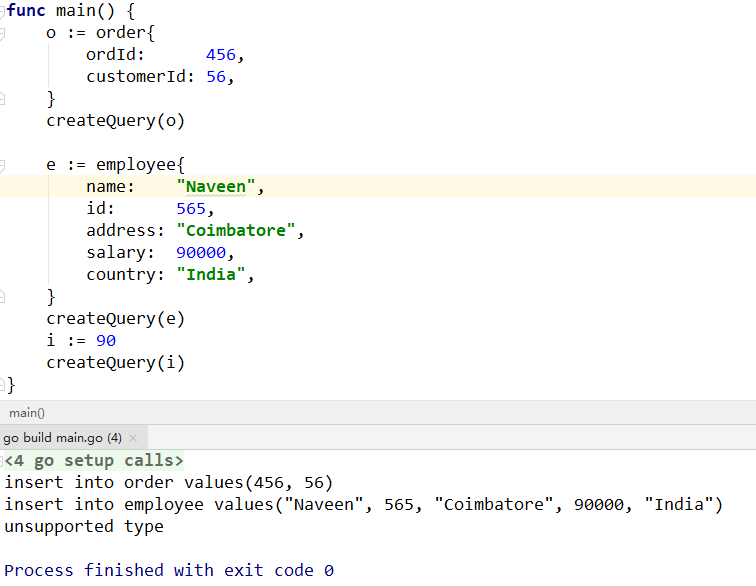
Int 和 String 可以帮助我们分别取出 reflect.Value 作为 int64 和 string。

反射示例：

假设我们需要编写一个createQuery函数，它能接收任意的结构体变量作为参数，返回对应的SQL插入查询语句。由于 createQuery 函数应该适用于任何结构体，因此它接收 interface{} 作为参数。为了简单起见，我们只处理包含 string 和 int 类型字段的结构体，但可以扩展为包含任何类型的字段。程序如下：



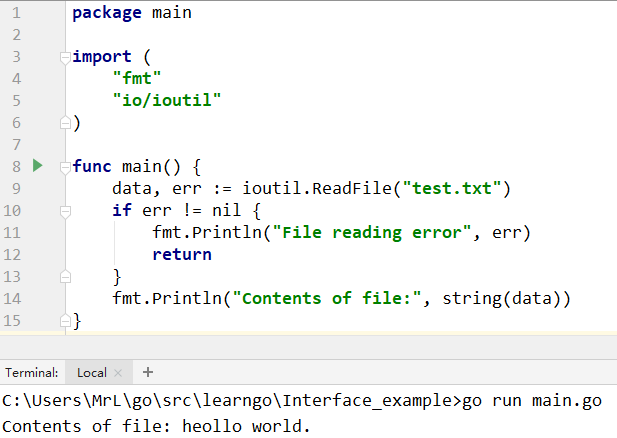




# 24.读取文件

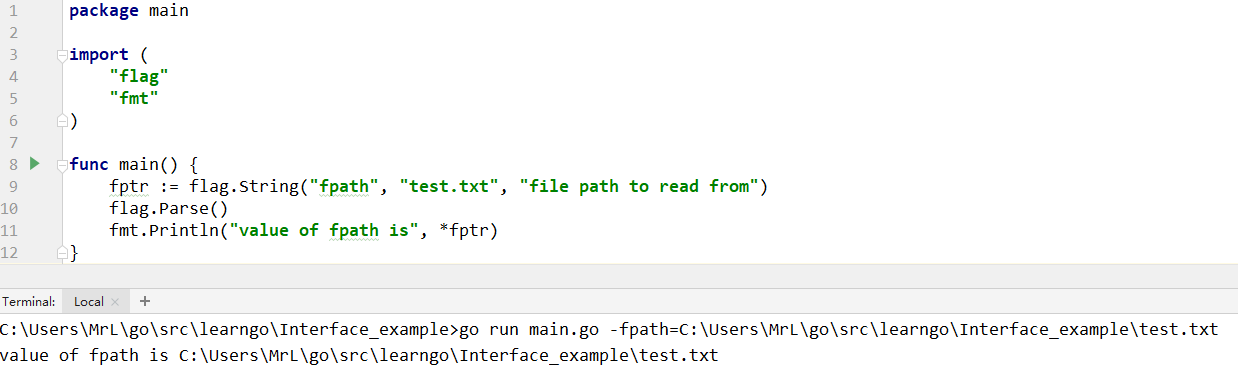
将整个文件读取到内存：

使用 ioutil 包中的 ReadFile 函数。如下示例，需要在运行目录下创建test.txt文件：



使用命令行标记来传递文件路径：

使用 flag 包，我们可以从输入的命令行获取到文件路径，接着读取文件内容。flag 包有一个名为 String 的函数。该函数接收三个参数。第一个参数是标记名，第二个是默认值，第三个是标记的简短描述。

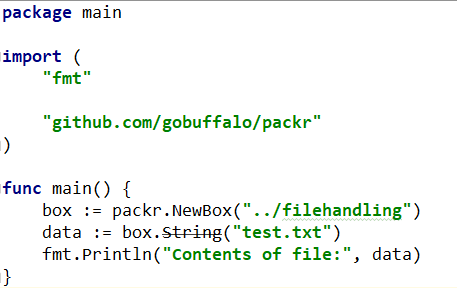


将文件绑定在二进制文件中：

如果我们能够将文本文件捆绑在二进制文件，岂不是很棒？有很多包可以帮助我们实现。我们会使用 packr，因为它很简单，并且我在项目中使用它时，没有出现任何问题。第一步就是安装 packr 包：

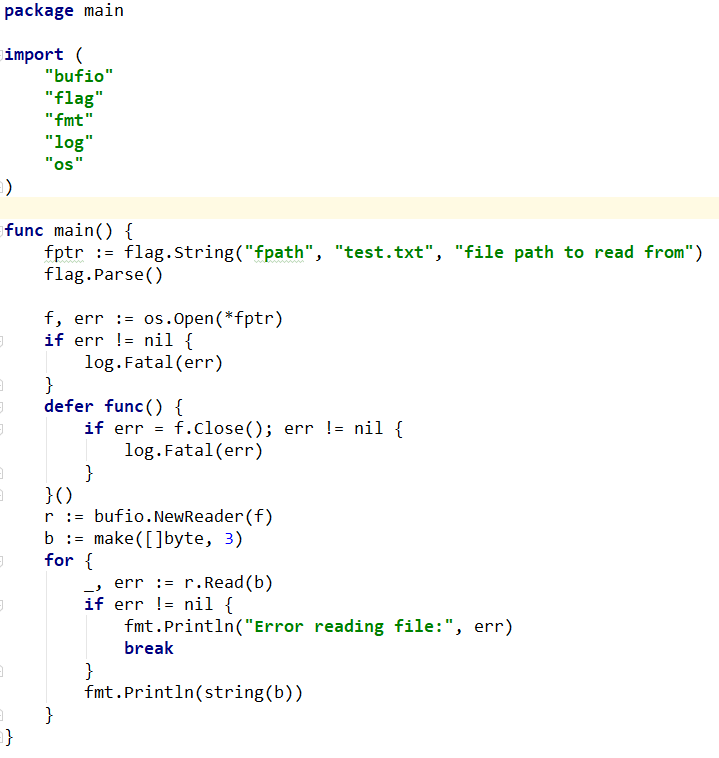
go get -u github.com/gobuffalo/packr/...

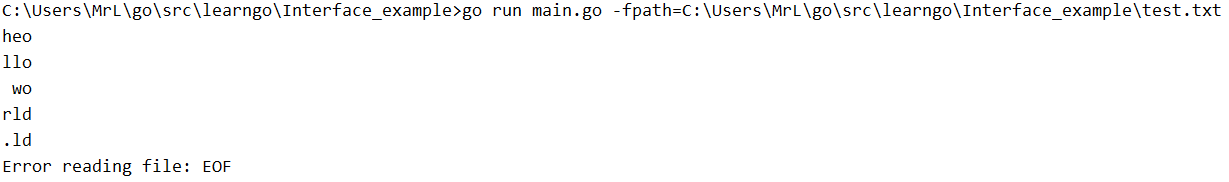
packr 会把静态文件（例如 .txt 文件）转换为 .go 文件，接下来，.go 文件会直接嵌入到二进制文件中。packer 非常智能，在开发过程中，可以从磁盘而非二进制文件中获取静态文件。在开发过程中，当仅仅静态文件变化时，可以不必重新编译。



分块读取文件：

当文件非常大时，尤其在 RAM 存储量不足的情况下，把整个文件都读入内存是没有意义的。更好的方法是分块读取文件。这可以使用 bufio 包来完成。





Read 方法会读取 len(b) 个字节（达到 3 字节），并返回所读取的字节数。当到达文件最后时，它会返回一个 EOF 错误。

逐行读取文件：

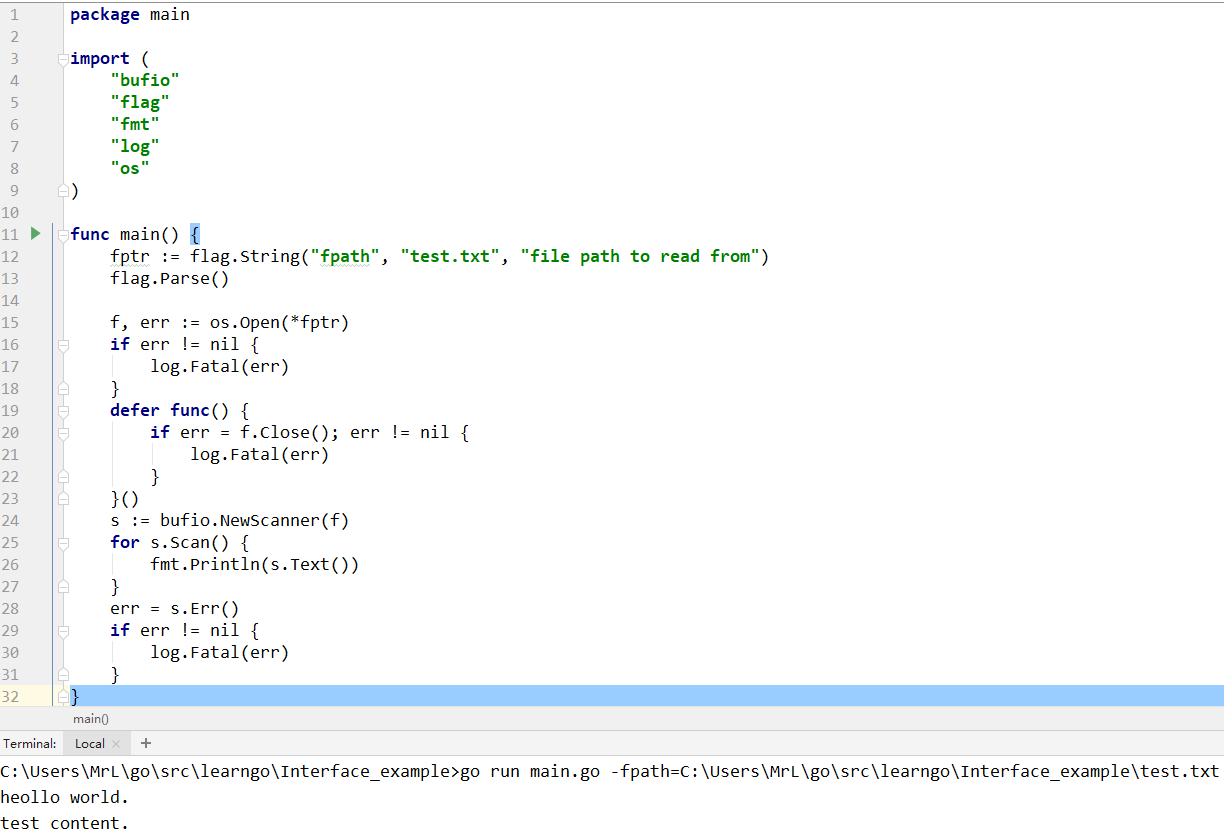
逐行读取文件涉及到以下步骤。

1.打开文件；

2.在文件上新建一个 scanner；

3.扫描文件并且逐行读取。

读取之前，我们把test.txt文件新增一行内容“test content”。



在第 24 行，我们用文件创建了一个新的 scanner。第 25 行的 Scan() 方法读取文件的下一行，如果可以读取，就可以使用 Text() 方法。当 Scan 返回 false 时，除非已经到达文件末尾（此时 Err() 返回 nil），否则 Err() 就会返回扫描过程中出现的错误。