更多Golang资源包：https://github.com/0voice/Introduction-to-Golang

关键字

Go语言设计的关键字，了解这些关键字有助于命名变量的冲突避免

go的二十五个关键字

```
break    default      func    interface    select
case     defer        go      map          struct
chan     else         goto    package      switch
const    fallthrough  if      range        type
continue for          import  return       var
```

简介

- `var`和`const`是Go语言基础里面的变量和常量申明
- `package`和`import`用于分包和导入
- `func`用于定义函数和方法
- `return`用于从函数返回
- `defer`用于类似析构函数
- `go`用于并发
- `select`用于选择不同类型的通讯
- `interface`用于定义接口
- `struct`用于定义抽象数据类型
- `break`、`case`、`continue`、`for`、`fallthrough`、`else`、`if`、`switch`、`goto`、`default`用于流程控制
- `chan`用于channel通讯
- `type`用于声明自定义类型
- `map`用于声明map类型数据
- `range`用于读取slice、map、channel数据

数据类型的定义

定义变量

Go语言里面定义变量有多种方式。

使用`var`关键字是Go最基本的定义变量方式，与C语言不同的是Go把变量类型放在变量名后面：

```
//定义一个名称为“variableName”，类型为"type"的变量
var variableName type
```

定义多个变量

```
//定义三个类型都是“type”的变量
var vname1, vname2, vname3 type
```

定义变量并初始化值

```
//初始化“variableName”的变量为“value”值，类型是“type”
var variableName type = value
```

同时初始化多个变量

```
/*
    定义三个类型都是"type"的变量,并且分别初始化为相应的值
    vname1为v1，vname2为v2，vname3为v3
*/
var vname1, vname2, vname3 type= v1, v2, v3
```

是不是觉得上面这样的定义有点繁琐？有一种写法可以让它变得简单一点。可以直接忽略类型声明，那么上面的代码变成这样了：

```
/*
    定义三个变量，它们分别初始化为相应的值
    vname1为v1，vname2为v2，vname3为v3
    然后Go会根据其相应值的类型来初始化它们
*/
var vname1, vname2, vname3 = v1, v2, v3
```

觉得上面的还是有些繁琐，继续简化：

```
/*
    定义三个变量，它们分别初始化为相应的值
    vname1为v1，vname2为v2，vname3为v3
    编译器会根据初始化的值自动推导出相应的类型
*/
vname1, vname2, vname3 := v1, v2, v3
```

现在是不是看上去非常简洁了？`:=`这个符号直接取代了`var`和`type`,这种形式叫做简短声明。不过它有一个限制，那就是它只能用在函数内部；在函数外部使用则会无法编译通过，所以一般用`var`方式来定义全局变量。

`_`（下划线）是个特殊的变量名，任何赋予它的值都会被丢弃。在这个例子中，将值`35`赋予`b`，并同时丢弃`34`：

```
_, b := 34, 35
```

Go对于已声明但未使用的变量会在编译阶段报错，比如下面的代码就会产生一个错误：声明了`i`但未使用。

```
package main
func main() {
    var i int
}
```

常量

所谓常量，也就是在程序编译阶段就确定下来的值，而程序在运行时无法改变该值。在Go程序中，常量可定义为数值、布尔值或字符串等类型。

它的语法如下：

```
const constantName = value
//如果需要，也可以明确指定常量的类型：
const Pi float32 = 3.1415926
```

下面是一些常量声明的例子：

```
const Pi = 3.1415926
const i = 10000
const MaxThread = 10
const prefix = "astaxie_"
```

Go 常量和一般程序语言不同的是，可以指定相当多的小数位数(例如200位)，若指定给`float32`自动缩短为`32bit`，指定给`float64`自动缩短为`64bit`，详情参考`http://golang.org/ref/spec#Constants`(需科学上网)

内置基础类型

Boolean

在Go中，布尔值的类型为`bool`，值是`true`或`false`，默认为`false`。

```
//示例代码
var isActive bool  // 全局变量声明
var enabled, disabled = true, false  // 忽略类型的声明
func test() {
    var available bool  // 一般声明
    valid := false      // 简短声明
    available = true    // 赋值操作
}
```

数值类型

整数类型有无符号和带符号两种。Go同时支持`int`和`uint`，这两种类型的长度相同，但具体长度取决于不同编译器的实现。Go里面也有直接定义好位数的类型：`rune`, `int8`, `int16`, `int32`, `int64`和`byte`, `uint8`, `uint16`, `uint32`, `uint64`。其中`rune`是`int32`的别称，`byte`是`uint8`的别称。

需要注意的一点是，这些类型的变量之间不允许互相赋值或操作，不然会在编译时引起编译器报错。

如下的代码会产生错误：invalid operation: a + b (mismatched types int8 and int32)

var a int8

var b int32

c:=a + b

另外，尽管int的长度是32 bit, 但int 与 int32并不可以互用。

浮点数的类型有`float32`和`float64`两种（没有`float`类型），默认是`float64`。

Go还支持复数。它的默认类型是`complex128`（64位实数+64位虚数）。如果需要小一些的，也有`complex64`(32位实数+32位虚数)。复数的形式为`RE + IMi`，其中`RE`是实数部分，`IM`是虚数部分，而最后的`i`是虚数单位。下面是一个使用复数的例子：

```
var c complex64 = 5+5i
//output: (5+5i)
fmt.Printf("Value is: %v", c)
```

字符串

Go中的字符串都是采用`UTF-8`字符集编码。字符串是用一对双引号（`""`）或反引号（``）括起来定义，它的类型是`string`。

```
//示例代码
var frenchHello string  // 声明变量为字符串的一般方法
var emptyString string = ""  // 声明了一个字符串变量，初始化为空字符串
func test() {
    no, yes, maybe := "no", "yes", "maybe"  // 简短声明，同时声明多个变量
    japaneseHello := "Konichiwa"  // 同上
    frenchHello = "Bonjour"  // 常规赋值
}
```

在Go中字符串是不可变的，例如下面的代码编译时会报错：cannot assign to s[0]

```
var s string = "hello"
s[0] = 'c'
```

但如果真的想要修改怎么办呢？下面的代码可以实现：

```
s := "hello"
c := []byte(s)  // 将字符串 s 转换为 []byte 类型
c[0] = 'c'
s2 := string(c)  // 再转换回 string 类型
fmt.Printf("%s\n", s2)
```

Go中可以使用`+`操作符来连接两个字符串：

```
s := "hello,"
m := " world"
a := s + m
fmt.Printf("%s\n", a)
```

修改字符串也可写为：

```
s := "hello"
s = "c" + s[1:] // 字符串虽不能更改，但可进行切片操作
fmt.Printf("%s\n", s)
```

如果要声明一个多行的字符串怎么办？可以通过```来声明：

```
m := `hello
    world`
```

``` 括起的字符串为Raw`字符串，即字符串在代码中的形式就是打印时的形式，它没有字符转义，换行也将原样输出。例如本例中会输出：

```
hello
    world
```

错误类型

Go内置有一个 error 类型，专门用来处理错误信息，Go的 package 里面还专门有一个包 errors 来处理错误：

```
err := errors.New("emit macho dwarf: elf header corrupted")
if err != nil {
    fmt.Print(err)
}
```

分组声明

在Go语言中，同时声明多个常量、变量，或者导入多个包时，可采用分组的方式进行声明。

例如下面的代码：

```
import "fmt"
import "os"
const i = 100
const pi = 3.1415
const prefix = "Go_"
var i int
var pi float32
var prefix string
```

可以分组写成如下形式：

```
import(
    "fmt"
    "os"
)
```
```

```
const(
    i = 100
    pi = 3.1415
    prefix = "Go_"
)
var(
    i int
    pi float32
    prefix string
)
```

iota枚举

Go里面有一个关键字`iota`，这个关键字用来声明`enum`的时候采用，它默认开始值是0，const中每增加一行加1：

```
package main
import (
    "fmt"
)
const (
    x = iota // x == 0
    y = iota // y == 1
    z = iota // z == 2
    w        // 常量声明省略值时，默认和之前一个值的字面相同。这里隐式地说w = iota，因此w
== 3。其实上面y和z可同样不用"= iota"
)
const v = iota // 每遇到一个const关键字，iota就会重置，此时v == 0
const (
    h, i, j = iota, iota, iota //h=0,i=0,j=0 iota在同一行值相同
)
const (
    a       = iota //a=0
    b       = "B"
    c       = iota             //c=2
    d, e, f = iota, iota, iota //d=3,e=3,f=3
    g       = iota             //g = 4
)
func main() {
    fmt.Println(a, b, c, d, e, f, g, h, i, j, x, y, z, w, v)
}
```

除非被显式设置为其它值或`iota`，每个`const`分组的第一个常量被默认设置为它的0值，第二及后续的常量被默认设置为它前面那个常量的值，如果前面那个常量的值是`iota`，则它也被设置为`iota`。

Go程序设计的一些规则

Go之所以会那么简洁，是因为它有一些默认的行为：

- 大写字母开头的变量是可导出的，也就是其它包可以读取的，是公有变量；小写字母开头的就是不可导出的，是私有变量。
- 大写字母开头的函数也是一样，相当于`class`中的带`public`关键词的公有函数；小写字母开头的就是有`private`关键词的私有函数。

`array`、`slice`、`map`

array

array就是数组，它的定义方式如下：

```
var arr [n]type
```

在[n]type中，n表示数组的长度，type表示存储元素的类型。对数组的操作和其它语言类似，都是通过[]来进行读取或赋值：

```
var arr [10]int  // 声明了一个int类型的数组
arr[0] = 42      // 数组下标是从0开始的
arr[1] = 13      // 赋值操作
fmt.Printf("The first element is %d\n", arr[0])  // 获取数据，返回42
fmt.Printf("The last element is %d\n", arr[9]) //返回未赋值的最后一个元素，默认返回0
```

由于长度也是数组类型的一部分，因此[3]int与[4]int是不同的类型，数组也就不能改变长度。数组之间的赋值是值的赋值，即当把一个数组作为参数传入函数的时候，传入的其实是该数组的副本，而不是它的指针。如果要使用指针，那么就需要用到后面介绍的slice类型了。

数组可以使用另一种:=来声明

```
a := [3]int{1, 2, 3} // 声明了一个长度为3的int数组
b := [10]int{1, 2, 3} // 声明了一个长度为10的int数组，其中前三个元素初始化为1、2、3，其它默认为0
c := [...]int{4, 5, 6} // 可以省略长度而采用`...`的方式，Go会自动根据元素个数来计算长度
```

Go支持嵌套数组，即多维数组。比如下面的代码就声明了一个二维数组：

```
// 声明了一个二维数组，该数组以两个数组作为元素，其中每个数组中又有4个int类型的元素
doubleArray := [2][4]int{[4]int{1, 2, 3, 4}, [4]int{5, 6, 7, 8}}
// 上面的声明可以简化，直接忽略内部的类型
easyArray := [2][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}}
```

slice

在很多应用场景中，数组并不能满足需求。在初始定义数组时，并不知道需要多大的数组，因此就需要“动态数组”。在Go里面这种数据结构叫slice

slice并不是真正意义上的动态数组，而是一个引用类型。slice总是指向一个底层array，slice的声明也可以像array一样，只是不需要长度。

```
// 和声明array一样，只是少了长度
var fslice []int
```

接下来可以声明一个slice，并初始化数据，如下所示：

```
slice := []byte {'a', 'b', 'c', 'd'}
```

slice可以从一个数组或一个已经存在的slice中再次声明。slice通过array[i:j]来获取，其中i是数组的开始位置，j是结束位置，但不包含array[j]，它的长度是j-i。

```
// 声明一个含有10个元素元素类型为byte的数组
var ar = [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
// 声明两个含有byte的slice
var a, b []byte
// a指向数组的第3个元素开始，并到第五个元素结束，
a = ar[2:5]
//现在a含有的元素: ar[2]、ar[3]和ar[4]
// b是数组ar的另一个slice
b = ar[3:5]
// b的元素是：ar[3]和ar[4]
```

注意`slice`和数组在声明时的区别：声明数组时，方括号内写明了数组的长度或使用`...`自动计算长度，而声明`slice`时，方括号内没有任何字符。

`slice`有一些简便的操作

- `slice`的默认开始位置是0，`ar[:n]`等价于`ar[0:n]`
- `slice`的第二个序列默认是数组的长度，`ar[n:]`等价于`ar[n:len(ar)]`
- 如果从一个数组里面直接获取`slice`，可以这样`ar[:]`，因为默认第一个序列是0，第二个是数组的长度，即等价于`ar[0:len(ar)]`

下面这个例子展示了更多关于`slice`的操作：

```
// 声明一个数组
var array = [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
// 声明两个slice
var aSlice, bSlice []byte
// 演示一些简便操作
aSlice = array[:3] // 等价于aSlice = array[0:3] aSlice包含元素: a,b,c
aSlice = array[5:] // 等价于aSlice = array[5:10] aSlice包含元素: f,g,h,i,j
aSlice = array[:]  // 等价于aSlice = array[0:10] 这样aSlice包含了全部的元素
// 从slice中获取slice
aSlice = array[3:7]  // aSlice包含元素: d,e,f,g, len=4, cap=7
bSlice = aSlice[1:3] // bSlice 包含aSlice[1], aSlice[2] 也就是含有: e,f
bSlice = aSlice[:3]  // bSlice 包含 aSlice[0], aSlice[1], aSlice[2] 也就是含有:
d,e,f
bSlice = aSlice[0:5] // 对slice的slice可以在cap范围内扩展，此时bSlice包含：d,e,f,g,h
bSlice = aSlice[:]   // bSlice包含所有aSlice的元素: d,e,f,g
```

`slice`是引用类型，所以当引用改变其中元素的值时，其它的所有引用都会改变该值，例如上面的`aSlice`和`bSlice`，如果修改了`aSlice`中元素的值，那么`bSlice`相对应的值也会改变。

从概念上面来说`slice`像一个结构体，这个结构体包含了三个元素：

- 一个指针，指向数组中`slice`指定的开始位置
- 长度，即`slice`的长度
- 最大长度，也就是`slice`开始位置到数组的最后位置的长度

```
Array_a := [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
    Slice_a := Array_a[2:5]
```

slice有几个有用的内置函数

- len 获取slice的长度
- cap 获取slice的最大容量
- append 向slice里面追加一个或者多个元素，然后返回一个和slice一样类型的slice
- copy 函数copy从源slice的src中复制元素到目标dst，并且返回复制的元素的个数

注：append函数会改变slice所引用的数组的内容，从而影响到引用同一数组的其它slice。

但当slice中没有剩余空间（即(cap-len) == 0）时，此时将动态分配新的数组空间。返回的slice数组指针将指向这个空间，而原数组的内容将保持不变；其它引用此数组的slice则不受影响。

从Go1.2开始slice支持了三个参数的slice，之前一直采用这种方式在slice或者array基础上来获取一个slice

```
var array [10]int
slice := array[2:4]
```

这个例子里面slice的容量是8，新版本里面可以指定这个容量

```
slice = array[2:4:7]
```

上面这个的容量就是7-2，即5。这样这个产生的新的slice就没办法访问最后的三个元素。

如果slice是这样的形式array[:i:j]，即第一个参数为空，默认值就是0。

map

```
map`也就是Python中字典的概念，它的格式为`map[keyType]valueType
```

看下面的代码，map的读取和设置也类似slice一样，通过key来操作，只是slice的index只能是`int`类型，而map多了很多类型，可以是int，可以是string及所有完全定义了==与!=操作的类型。

```
// 声明一个key是字符串，值为int的字典,这种方式的声明需要在使用之前使用make初始化
var numbers map[string]int
// 另一种map的声明方式
numbers = make(map[string]int)
numbers["one"] = 1  //赋值
numbers["ten"] = 10 //赋值
numbers["three"] = 3
fmt.Println("第三个数字是: ", numbers["three"]) // 读取数据
// 打印出来如:第三个数字是: 3
```

这个map就像平常看到的表格一样，左边列是key，右边列是值

使用map过程中需要注意的几点：

- map是无序的，每次打印出来的map都会不一样，它不能通过index获取，而必须通过key获取
- map的长度是不固定的，也就是和slice一样，也是一种引用类型
- 内置的len函数同样适用于map，返回map拥有的key的数量
- map的值可以很方便的修改，通过numbers["one"]=11可以很容易的把key为one的字典值改为11

- `map`和其他基本型别不同，它不是thread-safe，在多个go-routine存取时，必须使用mutex lock机制

`map`的初始化可以通过`key:val`的方式初始化值，同时`map`内置有判断是否存在`key`的方式

通过`delete`删除`map`的元素：

```
// 初始化一个字典
rating := map[string]float32{"C":5, "Go":4.5, "Python":4.5, "C++":2 }
// map有两个返回值，第二个返回值，如果不存在key，那么ok为false，如果存在ok为true
csharpRating, ok := rating["C#"]
if ok {
    fmt.Println("C# is in the map and its rating is ", csharpRating)
} else {
    fmt.Println("We have no rating associated with C# in the map")
}
delete(rating, "C")  // 删除key为C的元素
```

上面说过了，`map`也是一种引用类型，如果两个`map`同时指向一个底层，那么一个改变，另一个也相应的改变：

```
m := make(map[string]string)
m["Hello"] = "Bonjour"
m1 := m
m1["Hello"] = "Salut"  // 现在m["hello"]的值已经是Salut了
```

`make`、`new`操作

`make`用于内建类型（`map`、`slice`和`channel`）的内存分配。`new`用于各种类型的内存分配。

内建函数`new`本质上说跟其它语言中的同名函数功能一样：`new(T)`分配了零值填充的`T`类型的内存空间，并且返回其地址，即一个`*T`类型的值。用Go的术语说，它返回了一个指针，指向新分配的类型`T`的零值。有一点非常重要：

`new`返回指针。

内建函数`make(T, args)`与`new(T)`有着不同的功能，make只能创建`slice`、`map`和`channel`，并且返回一个有初始值(非零)的`T`类型，而不是`*T`。本质来讲，导致这三个类型有所不同的原因是指向数据结构的引用在使用前必须被初始化。例如，一个`slice`，是一个包含指向数据（内部`array`）的指针、长度和容量的三项描述符；在这些项目被初始化之前，`slice`为`nil`。对于`slice`、`map`和`channel`来说，`make`初始化了内部的数据结构，填充适当的值。

`make`返回初始化后的（非零）值。

零值

关于“零值”，所指并非是空值，而是一种“变量未填充前”的默认值，通常为0。

此处罗列 部分类型 的 “零值”

```
int     0
int8    0
int32   0
int64   0
uint    0x0
rune    0 //rune的实际类型是 int32
byte    0x0 // byte的实际类型是 uint8
float32 0 //长度为 4 byte
float64 0 //长度为 8 byte
bool    false
string  ""
```

更多Golang资源包：https://github.com/0voice/Introduction-to-Golang

流程控制

Go中流程控制分三大类：条件判断，循环控制和无条件跳转。

if

`if`也许是各种编程语言中最常见的了，它的语法概括起来就是：如果满足条件就做某事，否则做另一件事。

Go里面`if`条件判断语句中不需要括号，如下代码所示

```
if x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}
```

Go的`if`还有一个强大的地方就是条件判断语句里面允许声明一个变量，这个变量的作用域只能在该条件逻辑块内，其他地方就不起作用了，如下所示

```
// 计算获取值x,然后根据x返回的大小，判断是否大于10。
if x := computedValue(); x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}
//这个地方如果这样调用就编译出错了，因为x是条件里面的变量
fmt.Println(x)
```

多个条件的时候如下所示：

```
if integer == 3 {
    fmt.Println("The integer is equal to 3")
} else if integer < 3 {
    fmt.Println("The integer is less than 3")
} else {
    fmt.Println("The integer is greater than 3")
}
```

goto

Go有`goto`语句——请明智地使用它。用`goto`跳转到必须在当前函数内定义的标签。例如假设这样一个循环：

```
func myFunc() {
    i := 0
Here:   //这行的第一个词，以冒号结束作为标签
    println(i)
    i++
    goto Here   //跳转到Here去
}
```

标签名是大小写敏感的。

for

Go里面最强大的一个控制逻辑就是`for`，它既可以用来循环读取数据，又可以当作`while`来控制逻辑，还能迭代操作。它的语法如下：

```
for expression1; expression2; expression3 {
    //...
}
```

`expression1`、`expression2`和`expression3`都是表达式，其中`expression1`和`expression3`是变量声明或者函数调用返回值之类的，`expression2`是用来条件判断，`expression1`在循环开始之前调用，`expression3`在每轮循环结束之时调用。

一个例子比上面讲那么多更有用，看看下面的例子吧：

```
package main
import "fmt"
func main(){
    sum := 0;
    for index:=0; index < 10 ; index++ {
        sum += index
    }
    fmt.Println("sum is equal to ", sum)
}
// 输出：sum is equal to 45
```

有些时候需要进行多个赋值操作，由于Go里面没有`,`操作符，那么可以使用平行赋值`i, j = i+1, j-1`

有些时候如果忽略`expression1`和`expression3`：

```
sum := 1
for ; sum < 1000;  {
    sum += sum
}
```

其中`;`也可以省略，那么就变成如下的代码了，这就是`while`的功能。

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

在循环里面有两个关键操作`break`和`continue` ，`break`操作是跳出当前循环，`continue`是跳过本次循环。当嵌套过深的时候，`break`可以配合标签使用，即跳转至标签所指定的位置，详细参考如下例子:

```
for index := 10; index>0; index-- {
    if index == 5{
        break // 或者continue
    }
    fmt.Println(index)
}
// break打印出来10、9、8、7、6
// continue打印出来10、9、8、7、6、4、3、2、1
```

`break`和`continue`还可以跟着标号，用来跳到多重循环中的外层循环

`for`配合`range`可以用于读取`slice`和`map`的数据:

```
for k,v:=range map {
    fmt.Println("map's key:",k)
    fmt.Println("map's val:",v)
}
```

由于 Go 支持 “多值返回”, 而对于“声明而未被调用”的变量, 编译器会报错, 在这种情况下, 可以使用`_`来丢弃不需要的返回值

例如

```
for _, v := range map{
    fmt.Println("map's val:", v)
}
```

switch

有些时候需要写很多的`if-else`来实现一些逻辑处理，这个时候代码看上去就很丑很冗长，而且也不易于以后的维护，这个时候`switch`就能很好的解决这个问题。它的语法如下

```
switch sExpr {
case expr1:
    some instructions
case expr2:
    some other instructions
case expr3:
    some other instructions
default:
    other code
}
```

`sExpr`和`expr1`、`expr2`、`expr3`的类型必须一致。Go的`switch`非常灵活，表达式不必是常量或整数，执行的过程从上至下，直到找到匹配项；而如果`switch`没有表达式，它会匹配`true`。

```
i := 10
switch i {
case 1:
    fmt.Println("i is equal to 1")
case 2, 3, 4:
    fmt.Println("i is equal to 2, 3 or 4")
case 10:
    fmt.Println("i is equal to 10")
default:
    fmt.Println("All I know is that i is an integer")
}
```

在第5行中，把很多值聚合在了一个`case`里面，同时，Go里面`switch`默认相当于每个`case`最后带有`break`，匹配成功后不会自动向下执行其他case，而是跳出整个`switch`，但是可以使用`fallthrough`强制执行后面的case代码。

```
integer := 6
switch integer {
case 4:
    fmt.Println("The integer was <= 4")
    fallthrough
case 5:
    fmt.Println("The integer was <= 5")
    fallthrough
case 6:
    fmt.Println("The integer was <= 6")
    fallthrough
case 7:
    fmt.Println("The integer was <= 7")
    fallthrough
case 8:
    fmt.Println("The integer was <= 8")
    fallthrough
default:
    fmt.Println("default case")
}
```

上面的程序将输出

```
The integer was <= 6
The integer was <= 7
The integer was <= 8
default case
```

函数

函数的定义

函数是Go里面的核心设计，它通过关键字`func`来声明，它的格式如下：

```
func funcName(input1 type1, input2 type2) (output1 type1, output2 type2) {
    //这里是处理逻辑代码
    //返回多个值
    return value1, value2
}
```

上面的代码可以看出

- 关键字`func`用来声明一个函数`funcName`
- 函数可以有一个或者多个参数，每个参数后面带有类型，通过`,`分隔
- 函数可以返回多个值
- 上面返回值声明了两个变量`output1`和`output2`，如果不想声明也可以，直接就两个类型
- 如果只有一个返回值且不声明返回值变量，那么可以省略 包括返回值的括号
- 如果没有返回值，那么就直接省略最后的返回信息
- 如果有返回值， 那么必须在函数的外层添加return语句

下面来看一个实际应用函数的例子（用来计算Max值）

```
package main
import "fmt"
// 返回a、b中最大值.
func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}
func main() {
    x := 3
    y := 4
    z := 5
    max_xy := max(x, y) //调用函数max(x, y)
    max_xz := max(x, z) //调用函数max(x, z)
    fmt.Printf("max(%d, %d) = %d\n", x, y, max_xy)
    fmt.Printf("max(%d, %d) = %d\n", x, z, max_xz)
    fmt.Printf("max(%d, %d) = %d\n", y, z, max(y,z)) // 也可在这直接调用它
}
```

上面这个里面可以看到`max`函数有两个参数，它们的类型都是`int`，那么第一个变量的类型可以省略（即 a,b int,而非 a int, b int），默认为离它最近的类型，同理多于2个同类型的变量或者返回值。同时注意到它的返回值就是一个类型，这个就是省略写法。

多个返回值

Go语言比C更先进的特性，其中一点就是函数能够返回多个值。

直接看例子

```
package main
import "fmt"
//返回 A+B 和 A*B
func SumAndProduct(A, B int) (int, int) {
    return A+B, A*B
}
func main() {
    x := 3
    y := 4
    xPLUSy, xTIMESy := SumAndProduct(x, y)
    fmt.Printf("%d + %d = %d\n", x, y, xPLUSy)
    fmt.Printf("%d * %d = %d\n", x, y, xTIMESy)
}
```

上面的例子可以看到直接返回了两个参数，当然也可以命名返回参数的变量，这个例子里面只是用了两个类型，也可以改成如下这样的定义，然后返回的时候不用带上变量名，因为直接在函数里面初始化了。但如果函数是导出的(首字母大写)，官方建议：最好命名返回值，因为不命名返回值，虽然使得代码更加简洁了，但是会造成生成的文档可读性差。

```
func SumAndProduct(A, B int) (add int, Multiplied int) {
    add = A+B
    Multiplied = A*B
    return
}
```

变参

Go函数支持变参。接受变参的函数是有着不定数量的参数的。为了做到这点，首先需要定义函数使其接受变参：

```
func myfunc(arg ...int) {}
```

`arg ...int`告诉Go这个函数接受不定数量的参数。注意，这些参数的类型全部是`int`。在函数体中，变量`arg`是一个`int`的`slice`：

```
for _, n := range arg {
    fmt.Printf("And the number is: %d\n", n)
}
```

传值与传指针

传一个参数值到被调用函数里面时，实际上是传了这个值的一份copy，当在被调用函数中修改参数值的时候，调用函数中相应实参不会发生任何变化，因为数值变化只作用在copy上。

为了验证上面的说法，来看一个例子

```
package main
import "fmt"
//简单的一个函数，实现了参数+1的操作
func add1(a int) int {
    a = a+1 // 改变了a的值
    return a //返回一个新值
}
```

```
func main() {
    x := 3
    fmt.Println("x = ", x)  // 应该输出 "x = 3"
    x1 := add1(x)  //调用add1(x)
    fmt.Println("x+1 = ", x1) // 应该输出"x+1 = 4"
    fmt.Println("x = ", x)    // 应该输出"x = 3"
}
```

虽然调用了`add1`函数，并且在`add1`中执行`a = a+1`操作，但是上面例子中`x`变量的值没有发生变化

理由很简单：因为当调用`add1`的时候，`add1`接收的参数其实是`x`的copy，而不是`x`本身。

如果真的需要传这个`x`本身,该怎么办呢?

这就牵扯到了所谓的指针。变量在内存中是存放于一定地址上的，修改变量实际是修改变量地址处的内存。只有`add1`函数知道`x`变量所在的地址，才能修改`x`变量的值。所以需要将`x`所在地址`&x`传入函数，并将函数的参数的类型由`int`改为`*int`，即改为指针类型，才能在函数中修改`x`变量的值。此时参数仍然是按copy传递的，只是copy的是一个指针。请看下面的例子

```
package main
import "fmt"
//简单的一个函数，实现了参数+1的操作
func add1(a *int) int { // 请注意，
    *a = *a+1 // 修改了a的值
    return *a // 返回新值
}
func main() {
    x := 3
    fmt.Println("x = ", x)  // 应该输出 "x = 3"
    x1 := add1(&x)  // 调用 add1(&x) 传x的地址
    fmt.Println("x+1 = ", x1) // 应该输出 "x+1 = 4"
    fmt.Println("x = ", x)    // 应该输出 "x = 4"
}
```

这样，就达到了修改`x`的目的。那么到底传指针有什么好处呢?

- 传指针使得多个函数能操作同一个对象。
- 传指针比较轻量级 (8bytes),只是传内存地址，可以用指针传递体积大的结构体。如果用参数值传递的话, 在每次copy上面就会花费相对较多的系统开销（内存和时间）。所以当要传递大的结构体的时候，用指针是一个明智的选择。
- Go语言中`channel`，`slice`，`map`这三种类型的实现机制类似指针，所以可以直接传递，而不用取地址后传递指针。（注：若函数需改变`slice`的长度，则仍需要取地址传递指针）

defer

Go语言中有种不错的设计，即延迟（defer）语句，可以在函数中添加多个defer语句。当函数执行到最后时，这些defer语句会按照逆序执行，最后该函数返回。特别是当进行一些打开资源的操作时，遇到错误需要提前返回，在返回前需要关闭相应的资源，不然很容易造成资源泄露等问题。如下代码所示，一般写打开一个资源是这样操作的:

```
func ReadWrite() bool {
    file.Open("file")
// 做一些工作
    if failureX {
        file.Close()
```

```
        return false
    }
    if failureY {
        file.Close()
        return false
    }
    file.Close()
    return true
}
```

上面有很多重复的代码，Go的`defer`有效解决了这个问题。使用它后，不但代码量减少了很多，而且程序变得更优雅。在`defer`后指定的函数会在函数退出前调用。

```
func ReadWrite() bool {
    file.Open("file")
    defer file.Close()
    if failureX {
        return false
    }
    if failureY {
        return false
    }
    return true
}
```

如果有很多调用`defer`，那么`defer`是采用后进先出模式，所以如下代码会输出`4 3 2 1 0`

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

通常来说，defer会用在释放数据库连接，关闭文件等需要在函数结束时处理的操作。

函数作为值、类型

在Go中函数也是一种变量，可以通过`type`来定义它，它的类型就是所有拥有相同的参数，相同的返回值的一种类型

```
type typeName func(input1 inputType1 , input2 inputType2 [, ...]) (result1
resultType1 [, ...])
```

函数作为类型到底有什么好处呢？那就是可以把这个类型的函数当做值来传递，请看下面的例子

```
package main
import "fmt"
type testInt func(int) bool // 声明了一个函数类型
func isOdd(integer int) bool {
    if integer%2 == 0 {
        return false
    }
    return true
}
func isEven(integer int) bool {
    if integer%2 == 0 {
```

```
        return true
    }
    return false
}
// 声明的函数类型在这个地方当做了一个参数
func filter(slice []int, f testInt) []int {
    var result []int
    for _, value := range slice {
        if f(value) {
            result = append(result, value)
        }
    }
    return result
}
func main(){
    slice := []int {1, 2, 3, 4, 5, 7}
    fmt.Println("slice = ", slice)
    odd := filter(slice, isOdd)    // 函数当做值来传递了
    fmt.Println("Odd elements of slice are: ", odd)
    even := filter(slice, isEven)  // 函数当做值来传递了
    fmt.Println("Even elements of slice are: ", even)
}
```

函数当做值和类型在写一些通用接口的时候非常有用，通过上面例子看到`testInt`这个类型是一个函数类型，然后两个`filter`函数的参数和返回值与`testInt`类型是一样的，但是可以实现很多种的逻辑，这样使得程序变得非常的灵活。

Panic和Recover

Go没有像Java那样的异常机制，它不能抛出异常，而是使用了`panic`和`recover`机制。一定要记住，应当把它作为最后的手段来使用，也就是说，代码中应当没有，或者很少有`panic`的东西。这是个强大的工具，请明智地使用它。

Panic

是一个内建函数，可以中断原有的控制流程，进入一个`panic`状态中。当函数`F`调用`panic`，函数F的执行被中断，但是`F`中的延迟函数会正常执行，然后F返回到调用它的地方。在调用的地方，`F`的行为就像调用了`panic`。这一过程继续向上，直到发生`panic`的`goroutine`中所有调用的函数返回，此时程序退出。`panic`可以直接调用`panic`产生。也可以由运行时错误产生，例如访问越界的数组。

Recover

是一个内建的函数，可以让进入`panic`状态的`goroutine`恢复过来。`recover`仅在延迟函数中有效。在正常的执行过程中，调用`recover`会返回`nil`，并且没有其它任何效果。如果当前的`goroutine`陷入`panic`状态，调用`recover`可以捕获到`panic`的输入值，并且恢复正常的执行。

下面这个函数演示了如何在过程中使用`panic`

```
var user = os.Getenv("USER")
func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

下面这个函数检查作为其参数的函数在执行时是否会产生`panic`：

```
func throwsPanic(f func()) (b bool) {
    defer func() {
        if x := recover(); x != nil {
            b = true
        }
    }()
    f() //执行函数f，如果f中出现了panic，那么就可以恢复回来
    return
}
```

注意：

`defer`必须在`panic`语句之前。

`recover`必须配合`defer`使用。

`main`函数和`init`函数

Go里面有两个保留的函数：`init`函数（能够应用于所有的`package`）和`main`函数（只能应用于`package main`）。这两个函数在定义时不能有任何的参数和返回值。虽然一个`package`里面可以写任意多个`init`函数，但这无论是对于可读性还是以后的可维护性来说，强烈建议用户在一个`package`中每个文件只写一个`init`函数。

Go程序会自动调用`init()`和`main()`，所以不需要在任何地方调用这两个函数。每个`package`中的`init`函数都是可选的，但`package main`就必须包含一个`main`函数。

程序的初始化和执行都起始于`main`包。如果`main`包还导入了其它的包，那么就会在编译时将它们依次导入。有时一个包会被多个包同时导入，那么它只会被导入一次（例如很多包可能都会用到`fmt`包，但它只会被导入一次，因为没有必要导入多次）。当一个包被导入时，如果该包还导入了其它的包，那么会先将其它包导入进来，然后再对这些包中的包级常量和变量进行初始化，接着执行`init`函数（如果有的话），依次类推。等所有被导入的包都加载完毕了，就会开始对`main`包中的包级常量和变量进行初始化，然后执行`main`包中的`init`函数（如果存在的话），最后执行`main`函数。

import

在写Go代码的时候经常用到import这个命令用来导入包文件，经常看到的方式参考如下：

```
import(
    "fmt"
)
```

然后代码里面可以通过如下的方式调用

```
fmt.Println("hello world")
```

上面这个fmt是Go语言的标准库，其实是去`GOROOT`环境变量指定目录下去加载该模块，当然Go的import还支持如下两种方式来加载自己写的模块：

1、相对路径

```
import "./model" //当前文件同一目录的model目录，但是不建议这种方式来import
```

2、绝对路径

```
import "shorturl/model" //加载gopath/src/shorturl/model模块
```

上面展示了一些import常用的几种方式，但是还有一些

特殊的import

1、点操作

有时候会看到如下的方式导入包

```
import(
    . "fmt"
)
```

这个点操作的含义就是这个包导入之后在调用这个包的函数时，可以省略前缀的包名，也就是前面调用的fmt.Println("hello world")可以省略的写成`Println("hello world")`

2、别名操作

别名操作顾名思义可以把包命名成另一个用起来容易记忆的名字

```
import(
        f "fmt"
    )
```

别名操作的话调用包函数时前缀变成了前缀，即`f.Println("hello world")`

3、_操作

这个操作经常是让很多人费解的一个操作符，请看下面这个`import`

```
import (
    "database/sql"
    _ "github.com/ziutek/mymysql/godrv"
)
```

`_`操作其实是引入该包，而不直接使用包里面的函数，而是调用了该包里面的`init函数`。

更多Golang资源包：https://github.com/0voice/Introduction-to-Golang

struct类型

struct类型的声明

Go语言中，也和C或者其他语言一样，可以声明新的类型，作为其它类型的属性或字段的容器。例如，可以创建一个自定义类型`person`代表一个人的实体。这个实体拥有属性：姓名和年龄。这样的类型称之`struct`。如下代码所示:

```
type person struct {
    name string
    age int
}
```

声明一个`struct`如此简单，上面的类型包含有两个字段

- 一个`string`类型的字段name，用来保存用户名称这个属性
- 一个`int`类型的字段age，用来保存用户年龄这个属性

使用`struct`看下面的代码

```
type person struct {
    name string
    age int
}
var P person  // P现在就是person类型的变量了
P.name = "Astaxie"  // 赋值"Astaxie"给P的name属性.
P.age = 25  // 赋值"25"给变量P的age属性
fmt.Printf("The person's name is %s", P.name)  // 访问P的name属性.
```

除了上面这种P的声明使用之外，还有另外几种声明使用方式：

1. 按照顺序提供初始化值
   `P := person{"Tom", 25}`
2. 通过`field:value`的方式初始化，这样可以任意顺序
   `P := person{age:24, name:"Tom"}`

1. 当然也可以通过`new`函数分配一个指针，此处P的类型为`*person`
   `P := new(person)`

看一个完整的使用`struct`的例子

```
package main
import "fmt"
// 声明一个新的类型
type person struct {
    name string
    age int
}
// 比较两个人的年龄，返回年龄大的那个人，并且返回年龄差
// struct也是传值的
func Older(p1, p2 person) (person, int) {
    if p1.age>p2.age {  // 比较p1和p2这两个人的年龄
        return p1, p1.age-p2.age
    }
    return p2, p2.age-p1.age
}
func main() {
    var tom person
    // 赋值初始化
    tom.name, tom.age = "Tom", 18
    // 两个字段都写清楚的初始化
    bob := person{age:25, name:"Bob"}
    // 按照struct定义顺序初始化值
    paul := person{"Paul", 43}
```

```
    tb_Older, tb_diff := Older(tom, bob)
    tp_Older, tp_diff := Older(tom, paul)
    bp_Older, bp_diff := Older(bob, paul)
    fmt.Printf("Of %s and %s, %s is older by %d years\n",
        tom.name, bob.name, tb_Older.name, tb_diff)
    fmt.Printf("Of %s and %s, %s is older by %d years\n",
        tom.name, paul.name, tp_Older.name, tp_diff)
    fmt.Printf("Of %s and %s, %s is older by %d years\n",
        bob.name, paul.name, bp_Older.name, bp_diff)
}
```

`struct`的匿名字段

定义的时候是字段名与其类型一一对应，实际上Go支持只提供类型，而不写字段名的方式，也就是匿名字段，也称为嵌入字段。

当匿名字段是一个`struct`的时候，那么这个`struct`所拥有的全部字段都被隐式地引入了当前定义的这个`struct`。

看一个例子，让上面说的这些更具体化

```
package main
import "fmt"
type Human struct {
    name string
    age int
    weight int
}
type Student struct {
    Human  // 匿名字段，那么默认Student就包含了Human的所有字段
    speciality string
}
func main() {
    // 初始化一个学生
    mark := Student{Human{"Mark", 25, 120}, "Computer Science"}
    // 访问相应的字段
    fmt.Println("His name is ", mark.name)
    fmt.Println("His age is ", mark.age)
    fmt.Println("His weight is ", mark.weight)
    fmt.Println("His speciality is ", mark.speciality)
    // 修改对应的备注信息
    mark.speciality = "AI"
    fmt.Println("Mark changed his speciality")
    fmt.Println("His speciality is ", mark.speciality)
    // 修改他的年龄信息
    fmt.Println("Mark become old")
    mark.age = 46
    fmt.Println("His age is", mark.age)
    // 修改他的体重信息
    fmt.Println("Mark is not an athlet anymore")
    mark.weight += 60
    fmt.Println("His weight is", mark.weight)
}
```

看到Student访问属性age和name的时候，就像访问自己所有用的字段一样，匿名字段就是这样，能够实现字段的继承。student还能访问Human这个字段作为字段名。请看下面的代码。

```
mark.Human = Human{"Marcus", 55, 220}
mark.Human.age -= 1
```

通过匿名访问和修改字段相当的有用，但是不仅仅是`struct`字段，所有的内置类型和自定义类型都是可以作为匿名字段的。请看下面的例子

```
package main
import "fmt"
type Skills []string
type Human struct {
    name string
    age int
    weight int
}
type Student struct {
    Human  // 匿名字段，struct
    Skills // 匿名字段，自定义的类型string slice
    int    // 内置类型作为匿名字段
    speciality string
}
func main() {
    // 初始化学生Jane
    jane := Student{Human:Human{"Jane", 35, 100}, speciality:"Biology"}
    // 现在访问相应的字段
    fmt.Println("Her name is ", jane.name)
    fmt.Println("Her age is ", jane.age)
    fmt.Println("Her weight is ", jane.weight)
    fmt.Println("Her speciality is ", jane.speciality)
    // 修改他的skill技能字段
    jane.Skills = []string{"anatomy"}
    fmt.Println("Her skills are ", jane.Skills)
    fmt.Println("She acquired two new ones ")
    jane.Skills = append(jane.Skills, "physics", "golang")
    fmt.Println("Her skills now are ", jane.Skills)
    // 修改匿名内置类型字段
    jane.int = 3
    fmt.Println("Her preferred number is", jane.int)
}
```

从上面例子看出来`struct`不仅仅能够将`struct`作为匿名字段，自定义类型、内置类型都可以作为匿名字段，而且可以在相应的字段上面进行函数操作（如例子中的append）。

这里有一个问题：如果human里面有一个字段叫做phone，而student也有一个字段叫做phone，那么该怎么办呢？

Go里面很简单的解决了这个问题，最外层的优先访问，也就是当通过`student.phone`访问的时候，是访问student里面的字段，而不是human里面的字段。

这样就允许去重载通过匿名字段继承的一些字段，当然如果想访问重载后对应匿名类型里面的字段，可以通过匿名字段名来访问。请看下面的例子

```
package main
import "fmt"
type Human struct {
    name string
    age int
```

```
    phone string  // Human类型拥有的字段
}
type Employee struct {
    Human  // 匿名字段Human
    speciality string
    phone string  // 雇员的phone字段
}
func main() {
    Bob := Employee{Human{"Bob", 34, "777-444-XXXX"}, "Designer", "333-222"}
    fmt.Println("Bob's work phone is:", Bob.phone)
    // 如果要访问Human的phone字段
    fmt.Println("Bob's personal phone is:", Bob.Human.phone)
}
```

method

函数的另一种形态，带有接收者的函数，称为`method`

method

现在假设有这么一个场景，定义了一个struct叫做长方形，现在想要计算他的面积，那么按照一般的思路应该会用下面的方式来实现

```
package main
import "fmt"
type Rectangle struct {
    width, height float64
}
func area(r Rectangle) float64 {
    return r.width*r.height
}
func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    fmt.Println("Area of r1 is: ", area(r1))
    fmt.Println("Area of r2 is: ", area(r2))
}
```

这段代码可以计算出来长方形的面积，但是area()不是作为Rectangle的方法实现的（类似面向对象里面的方法），而是将Rectangle的对象（如r1,r2）作为参数传入函数计算面积的。

这样实现当然没有问题，但是当需要增加圆形、正方形、五边形甚至其它多边形的时候，想计算他们的面积的时候怎么办？那就只能增加新的函数，但是函数名就必须要跟着换了，变成`area_rectangle, area_circle, area_triangle...`

椭圆代表函数, 而这些函数并不从属于struct(或者以面向对象的术语来说，并不属于class)，他们是单独存在于struct外围，而非在概念上属于某个struct的。

很显然，这样的实现并不优雅，并且从概念上来说"面积"是"形状"的一个属性，它是属于这个特定的形状的，就像长方形的长和宽一样。

基于上面的原因所以就有了`method`的概念，`method`是附属在一个给定的类型上的，他的语法和函数的声明语法几乎一样，只是在`func`后面增加了一个receiver(也就是method所依从的主体)。

用上面提到的形状的例子来说，method `area()` 是依赖于某个形状(比如说Rectangle)来发生作用的。Rectangle.area()的发出者是Rectangle， area()是属于Rectangle的方法，而非一个外围函数。

更具体地说，Rectangle存在字段 height 和 width, 同时存在方法area(), 这些字段和方法都属于Rectangle。

用Rob Pike的话来说就是：

"A method is a function with an implicit first argument, called a receiver."

method的语法如下：

```
func (r ReceiverType) funcName(parameters) (results)
```

下面用最开始的例子用method来实现：

```
package main
import (
    "fmt"
    "math"
)
type Rectangle struct {
    width, height float64
}
type Circle struct {
    radius float64
}
func (r Rectangle) area() float64 {
    return r.width*r.height
}
func (c Circle) area() float64 {
    return c.radius * c.radius * math.Pi
}
func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    c1 := Circle{10}
    c2 := Circle{25}
    fmt.Println("Area of r1 is: ", r1.area())
    fmt.Println("Area of r2 is: ", r2.area())
    fmt.Println("Area of c1 is: ", c1.area())
    fmt.Println("Area of c2 is: ", c2.area())
}
```

在使用method的时候重要注意几点

- 虽然method的名字一模一样，但是如果接收者不一样，那么method就不一样
- method里面可以访问接收者的字段
- 调用method通过`.`访问，就像struct里面访问字段一样

在上例，method area() 分别属于Rectangle和Circle， 于是他们的 Receiver 就变成了Rectangle 和 Circle, 或者说，这个area()方法 是由 Rectangle/Circle 发出的。

值得说明的一点是，图示中method用虚线标出，意思是此处方法的Receiver是以值传递，而非引用传递，是的，Receiver还可以是指针, 两者的差别在于, 指针作为Receiver会对实例对象的内容发生操作,而普通类型作为Receiver仅仅是以副本作为操作对象,并不对原实例对象发生操作。后文对此会有详细论述。

那是不是method只能作用在struct上面呢？当然不是，他可以定义在任何自定义的类型、内置类型、struct等各种类型上面。什么叫自定义类型，自定义类型不就是struct，其实不是这样的，struct只是自定义类型里面一种比较特殊的类型而已，还有其他自定义类型申明，可以通过如下这样的申明来实现。

```
type typeName typeLiteral
```

请看下面这个申明自定义类型的代码

```
type ages int
type money float32
type months map[string]int
m := months {
    "January":31,
    "February":28,
    ...
    "December":31,
}
```

这样就可以在自己的代码里面定义有意义的类型了，实际上只是一个定义了一个别名,有点类似于c中的typedef，例如上面ages替代了int，回到`method`可以在任何的自定义类型中定义任意多的`method`，接下来让看一个复杂一点的例子

```
package main
import "fmt"
const(
    WHITE = iota
    BLACK
    BLUE
    RED
    YELLOW
)
type Color byte
type Box struct {
    width, height, depth float64
    color Color
}
type BoxList []Box //a slice of boxes
func (b Box) Volume() float64 {
    return b.width * b.height * b.depth
}
func (b *Box) SetColor(c Color) {
    b.color = c
}
func (bl BoxList) BiggestColor() Color {
    v := 0.00
    k := Color(WHITE)
    for _, b := range bl {
        if bv := b.Volume(); bv > v {
            v = bv
            k = b.color
        }
    }
    return k
}
func (bl BoxList) PaintItBlack() {
```

```
    for i := range bl {
        bl[i].SetColor(BLACK)
    }
}
func (c Color) String() string {
    strings := []string {"WHITE", "BLACK", "BLUE", "RED", "YELLOW"}
    return strings[c]
}
func main() {
    boxes := BoxList {
        Box{4, 4, 4, RED},
        Box{10, 10, 1, YELLOW},
        Box{1, 1, 20, BLACK},
        Box{10, 10, 1, BLUE},
        Box{10, 30, 1, WHITE},
        Box{20, 20, 20, YELLOW},
    }
    fmt.Printf("We have %d boxes in our set\n", len(boxes))
    fmt.Println("The volume of the first one is", boxes[0].Volume(), "cm³")
    fmt.Println("The color of the last one
is",boxes[len(boxes)-1].color.String())
    fmt.Println("The biggest one is", boxes.BiggestColor().String())
    fmt.Println("Let's paint them all black")
    boxes.PaintItBlack()
    fmt.Println("The color of the second one is", boxes[1].color.String())
    fmt.Println("Obviously, now, the biggest one is",
boxes.BiggestColor().String())
}
```

上面的代码通过const定义了一些常量，然后定义了一些自定义类型

- Color作为byte的别名
- 定义了一个struct:Box，含有三个长宽高字段和一个颜色属性
- 定义了一个slice:BoxList，含有Box

然后以上面的自定义类型为接收者定义了一些method

- Volume()定义了接收者为Box，返回Box的容量
- SetColor(c Color)，把Box的颜色改为c
- BiggestColor()定在在BoxList上面，返回list里面容量最大的颜色
- PaintItBlack()把BoxList里面所有Box的颜色全部变成黑色
- String()定义在Color上面，返回Color的具体颜色(字符串格式)

上面的代码通过文字描述出来之后是不是很简单？一般解决问题都是通过问题的描述，去写相应的代码实现。

指针作为receiver

现在让回过头来看看SetColor这个method，它的receiver是一个指向Box的指针，可以使用*Box。

定义SetColor的真正目的是想改变这个Box的颜色，如果不传Box的指针，那么SetColor接受的其实是Box的一个copy，也就是说method内对于颜色值的修改，其实只作用于Box的copy，而不是真正的Box。所以需要传入指针。

这里可以把receiver当作method的第一个参数来看，然后结合前面函数讲解的传值和传引用就不难理解

这里也许会问SetColor函数里面应该这样定义`*b.Color=c`,而不是`b.Color=c`,需要读取到指针相应的值。

其实Go里面这两种方式都是正确的，当用指针去访问相应的字段时(虽然指针没有任何的字段)，Go知道要通过指针去获取这个值。PaintItBlack里面调用SetColor的时候是不是应该写成`(&bl[i]).SetColor(BLACK)`，因为SetColor的receiver是*Box，而不是Box。这两种方式都可以，因为Go知道receiver是指针，他自动转了。

也就是说：

如果一个method的receiver是*T,可以在一个T类型的实例变量V上面调用这个method，而不需要&V去调用这个method

类似的

如果一个method的receiver是T，可以在一个*T类型的变量P上面调用这个method，而不需要 *P去调用这个method

所以不用担心是调用的指针的method还是不是指针的method，Go知道要做的一切，这对于有多年C/C++编程经验的同学来说，真是解决了一个很大的痛苦。

method继承

通过字段的继承的学习，发现Go的一个神奇之处，method也是可以继承的。如果匿名字段实现了一个method，那么包含这个匿名字段的struct也能调用该method。来看下面这个例子

```
package main
import "fmt"
type Human struct {
    name string
    age int
    phone string
}
type Student struct {
    Human //匿名字段
    school string
}
type Employee struct {
    Human //匿名字段
    company string
}
//在human上面定义了一个method
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}
func main() {
    mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
    sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}
    mark.SayHi()
    sam.SayHi()
}
```

method重写

上面的例子中，如果Employee想要实现自己的SayHi,怎么办？简单，和匿名字段冲突一样的道理，可以在Employee上面定义一个method，重写了匿名字段的方法。请看下面的例子

```
package main
import "fmt"
type Human struct {
    name string
    age int
    phone string
}
type Student struct {
    Human //匿名字段
    school string
}
type Employee struct {
    Human //匿名字段
    company string
}
//Human定义method
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}
//Employee的method重写Human的method
func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
        e.company, e.phone) //Yes you can split into 2 lines here.
}
func main() {
    mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
    sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}
    mark.SayHi()
    sam.SayHi()
}
```

通过这些内容，可以设计出基本的面向对象的程序了，但是Go里面的面向对象是如此的简单，没有任何的私有、公有关键字，通过大小写来实现(大写开头的为公有，小写开头的为私有)，方法也同样适用这个原则。

interface

Go语言里面设计最精妙的应该算`interface`，它让面向对象，内容组织实现非常的方便

什么是interface

简单的说，`interface`是一组`method`签名的组合，通过`interface`来定义对象的一组行为。

前面例子中`Student`和`Employee`都能`SayHi`，虽然他们的内部实现不一样，但是那不重要，重要的是他们都能`say hi`

继续做更多的扩展，`Student`和`Employee`实现另一个方法`Sing`，然后`Student`实现方法`BorrowMoney`而`Employee`实现`SpendSalary`。

这样`Student`实现了三个方法：`SayHi`、`Sing`、`BorrowMoney`；而`Employee`实现了`SayHi`、`Sing`、`SpendSalary`。

上面这些方法的组合称为`interface`(被对象`Student`和`Employee`实现)。例如`Student`和`Employee`都实现了`interface`：`SayHi`和`Sing`，也就是这两个对象是该`interface`类型。而`Employee`没有实现这个`interface：SayHi、Sing`和`BorrowMoney`，因为`Employee`没有实现`BorrowMoney`这个方法。

interface类型

`interface`类型定义了一组方法，如果某个对象实现了某个接口的所有方法，则此对象就实现了此接口。详细的语法参考下面这个例子

```
type Human struct {
    name string
    age int
    phone string
}
type Student struct {
    Human //匿名字段Human
    school string
    loan float32
}
type Employee struct {
    Human //匿名字段Human
    company string
    money float32
}
//Human对象实现Sayhi方法
func (h *Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}
// Human对象实现Sing方法
func (h *Human) Sing(lyrics string) {
    fmt.Println("La la, la la la, la la la la la...", lyrics)
}
//Human对象实现Guzzle方法
func (h *Human) Guzzle(beerStein string) {
    fmt.Println("Guzzle Guzzle Guzzle...", beerStein)
}
// Employee重载Human的Sayhi方法
func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
        e.company, e.phone) //此句可以分成多行
}
//Student实现BorrowMoney方法
func (s *Student) BorrowMoney(amount float32) {
    s.loan += amount // (again and again and...)
}
//Employee实现SpendSalary方法
func (e *Employee) SpendSalary(amount float32) {
    e.money -= amount // More vodka please!!! Get me through the day!
}
// 定义interface
type Men interface {
    SayHi()
    Sing(lyrics string)
    Guzzle(beerStein string)
}
type YoungChap interface {
    SayHi()
    Sing(song string)
    BorrowMoney(amount float32)
}
type ElderlyGent interface {
```

```
    SayHi()
    Sing(song string)
    SpendSalary(amount float32)
}
```

通过上面的代码可以知道，interface可以被任意的对象实现。看到上面的Men interface被Human、Student和Employee实现。同理，一个对象可以实现任意多个interface，例如上面的Student实现了Men和YoungChap两个interface。

最后，任意的类型都实现了空interface(这样定义：interface{})，也就是包含0个method的interface。

interface值

那么interface里面到底能存什么值呢？如果定义了一个interface的变量，那么这个变量里面可以存实现这个interface的任意类型的对象。例如上面例子中，定义了一个Men interface类型的变量m，那么m里面可以存Human、Student或者Employee值。

因为m能够持有这三种类型的对象，所以可以定义一个包含Men类型元素的slice，这个slice可以被赋予实现了Men接口的任意结构的对象，这个和传统意义上面的slice有所不同。

来看一下下面这个例子:

```
package main
import "fmt"
type Human struct {
    name string
    age int
    phone string
}
type Student struct {
    Human //匿名字段
    school string
    loan float32
}
type Employee struct {
    Human //匿名字段
    company string
    money float32
}
//Human实现SayHi方法
func (h Human) SayHi() {
    fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}
//Human实现Sing方法
func (h Human) Sing(lyrics string) {
    fmt.Println("La la la la...", lyrics)
}
//Employee重载Human的SayHi方法
func (e Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
        e.company, e.phone)
    }
// Interface Men被Human,Student和Employee实现
// 因为这三个类型都实现了这两个方法
type Men interface {
    SayHi()
    Sing(lyrics string)
```

```
}
func main() {
    mike := Student{Human{"Mike", 25, "222-222-XXX"}, "MIT", 0.00}
    paul := Student{Human{"Paul", 26, "111-222-XXX"}, "Harvard", 100}
    sam := Employee{Human{"Sam", 36, "444-222-XXX"}, "Golang Inc.", 1000}
    tom := Employee{Human{"Tom", 37, "222-444-XXX"}, "Things Ltd.", 5000}
    //定义Men类型的变量i
    var i Men
    //i能存储Student
    i = mike
    fmt.Println("This is Mike, a Student:")
    i.SayHi()
    i.Sing("November rain")
    //i也能存储Employee
    i = tom
    fmt.Println("This is tom, an Employee:")
    i.SayHi()
    i.Sing("Born to be wild")
    //定义了slice Men
    fmt.Println("Let's use a slice of Men and see what happens")
    x := make([]Men, 3)
    //这三个都是不同类型的元素，但是他们实现了interface同一个接口
    x[0], x[1], x[2] = paul, sam, mike
    for _, value := range x{
        value.SayHi()
    }
}
```

通过上面的代码，发现`interface`就是一组抽象方法的集合，它必须由其他非interface类型实现，而不能自我实现， Go通过interface实现了`duck-typing`:即"当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子"。

空interface

空interface(interface{})不包含任何的method，正因为如此，所有的类型都实现了空interface。空interface对于描述起不到任何的作用(因为它不包含任何的method），但是空interface需要存储任意类型的数值的时候相当有用，因为它可以存储任意类型的数值。它有点类似于C语言的void*类型。

```
// 定义a为空接口
var a interface{}
var i int = 5
s := "Hello world"
// a可以存储任意类型的数值
a = i
a = s
```

一个函数把interface{}作为参数，那么他可以接受任意类型的值作为参数，如果一个函数返回`interface{}`,那么也就可以返回任意类型的值。是不是很有用啊！

interface函数参数

interface的变量可以持有任意实现该interface类型的对象，这给编写函数(包括method)提供了一些额外的思考，是不是可以通过定义interface参数，让函数接受各种类型的参数。

举个例子：fmt.Println是常用的一个函数，是否注意到它可以接受任意类型的数据。打开fmt的源码文件，会看到这样一个定义:

```
type Stringer interface {
     String() string
}
```

也就是说，任何实现了String方法的类型都能作为参数被`fmt.Println`调用,来试一试

```
package main
import (
    "fmt"
    "strconv"
)
type Human struct {
    name string
    age int
    phone string
}
// 通过这个方法 Human 实现了 fmt.Stringer
func (h Human) String() string {
    return "❰"+h.name+" - "+strconv.Itoa(h.age)+" years -  ✆ " +h.phone+"❱"
}
func main() {
    Bob := Human{"Bob", 39, "000-7777-XXX"}
    fmt.Println("This Human is : ", Bob)
}
```

现在再回顾一下前面的Box示例，发现Color结构也定义了一个`method: String`。其实这也是实现了`fmt.Stringer`这个`interface`，即如果需要某个类型能被fmt包以特殊的格式输出，就必须实现`Stringer`这个接口。如果没有实现这个接口，fmt将以默认的方式输出。

```
//实现同样的功能
fmt.Println("The biggest one is", boxes.BiggestsColor().String())
fmt.Println("The biggest one is", boxes.BiggestsColor())
```

注：实现了`error`接口的对象（即实现了Error() string的对象），使用fmt输出时，会调用Error()方法，因此不必再定义String()方法了。

interface变量存储的类型

interface的变量里面可以存储任意类型的数值(该类型实现了interface)。那么怎么反向知道这个变量里面实际保存了的是哪个类型的对象呢？目前常用的有两种方法：

- Comma-ok断言

Go语言里面有一个语法，可以直接判断是否是该类型的变量： value, ok = element.(T)，这里value就是变量的值，ok是一个bool类型，element是interface变量，T是断言的类型。

如果element里面确实存储了T类型的数值，那么ok返回true，否则返回false。

通过一个例子来更加深入的理解。

```
package main
import (
    "fmt"
    "strconv"
```

```
)
type Element interface{}
type List [] Element
type Person struct {
    name string
    age int
}
//定义了String方法，实现了fmt.Stringer
func (p Person) String() string {
    return "(name: " + p.name + " - age: "+strconv.Itoa(p.age)+ " years)"
}
func main() {
    list := make(List, 3)
    list[0] = 1 // an int
    list[1] = "Hello" // a string
    list[2] = Person{"Dennis", 70}
    for index, element := range list {
        if value, ok := element.(int); ok {
            fmt.Printf("list[%d] is an int and its value is %d\n", index, value)
        } else if value, ok := element.(string); ok {
            fmt.Printf("list[%d] is a string and its value is %s\n", index,
value)
        } else if value, ok := element.(Person); ok {
            fmt.Printf("list[%d] is a Person and its value is %s\n", index,
value)
        } else {
            fmt.Printf("list[%d] is of a different type\n", index)
        }
    }
}
```

是否注意到了多个if里面，if里面允许初始化变量。断言的类型越多，那么if else也就越多，所以才引出了下面要介绍的switch。

- switch测试

重写上面的这个实现

```
package main
import (
    "fmt"
    "strconv"
)
type Element interface{}
type List [] Element
type Person struct {
    name string
    age int
}
//打印
func (p Person) String() string {
    return "(name: " + p.name + " - age: "+strconv.Itoa(p.age)+ " years)"
}
func main() {
    list := make(List, 3)
    list[0] = 1 //an int
    list[1] = "Hello" //a string
```

```
    list[2] = Person{"Dennis", 70}
    for index, element := range list{
        switch value := element.(type) {
            case int:
            fmt.Printf("list[%d] is an int and its value is %d\n", index, value)
            case string:
            fmt.Printf("list[%d] is a string and its value is %s\n", index,
value)
            case Person:
            fmt.Printf("list[%d] is a Person and its value is %s\n", index,
value)
            default:
            fmt.Println("list[%d] is of a different type", index)
        }
    }
}
```

这里有一点需要强调的是：`element.(type)`语法不能在switch外的任何逻辑里面使用，如果要在switch外面判断一个类型就使用`comma-ok`。

嵌入interface

Go里面真正吸引人的是它内置的逻辑语法，就像在学习Struct时学习的匿名字段，那么相同的逻辑引入到interface里面，更加完美了。如果一个interface1作为interface2的一个嵌入字段，那么interface2隐式的包含了interface1里面的method。

可以看到源码包`container/heap`里面有这样的一个定义

```
type Interface interface {
    sort.Interface //嵌入字段sort.Interface
    Push(x interface{}) //a Push method to push elements into the heap
    Pop() interface{} //a Pop elements that pops elements from the heap
}
```

看到`sort.Interface`其实就是嵌入字段，把`sort.Interface`的所有`method`给隐式的包含进来了。也就是下面三个方法：

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less returns whether the element with index i should sort
    // before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

另一个例子就是io包下面的 `io.ReadWriter` ，它包含了io包下面的`Reader`和`Writer`两个`interface`：

```
// io.ReadWriter
type ReadWriter interface {
    Reader
    Writer
}
```

反射

Go语言实现了反射，所谓反射就是能检查程序在运行时的状态。一般用到的包是`reflect`包。如何运用`reflect`包，官方的这篇文章详细的讲解了`reflect`包的实现原理，`laws of reflection`链接地址为 http://golang.org/doc/articles/laws_of_reflection.html

使用reflect一般分成三步，下面简要的讲解一下：要去反射是一个类型的值(这些值都实现了`空interface`)，首先需要把它转化成`reflect`对象(`reflect.Type`或者`reflect.Value`，根据不同的情况调用不同的函数)。这两种获取方式如下：

```
t := reflect.TypeOf(i)    //得到类型的元数据,通过t能获取类型定义里面的所有元素
v := reflect.ValueOf(i)   //得到实际的值，通过v获取存储在里面的值，还可以去改变值
```

转化为reflect对象之后就可以进行一些操作了，也就是将reflect对象转化成相应的值，例如

```
tag := t.Elem().Field(0).Tag  //获取定义在struct里面的标签
name := v.Elem().Field(0).String()  //获取存储在第一个字段里面的值
```

获取反射值能返回相应的类型和数值

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)
fmt.Println("value:", v.Float())
```

最后，反射的话，那么反射的字段必须是可修改的，前面学习过传值和传引用，这个里面也是一样的道理。反射的字段必须是可读写的意思是，如果下面这样写，那么会发生错误

```
var x float64 = 3.4
v := reflect.ValueOf(x)
v.SetFloat(7.1)
```

如果要修改相应的值，必须这样写

```
var x float64 = 3.4
p := reflect.ValueOf(&x)
v := p.Elem()
v.SetFloat(7.1)
```

更多Golang资源包：https://github.com/0voice/Introduction-to-Golang

并发

Go从语言层面支持了并行。

goroutine

`goroutine`是`Go`并行设计的核心。`goroutine`说到底其实就是协程，但是它比线程更小，十几个`goroutine`可能体现在底层就是五六个线程，Go语言内部实现了这些`goroutine`之间的内存共享。执行`goroutine`只需极少的栈内存(大概是4~5KB)，当然会根据相应的数据伸缩。也正因为如此，可同时运行成千上万个并发任务。`goroutine`比`thread`更易用、更高效、更轻便。

`goroutine`是通过Go的`runtime`管理的一个线程管理器。`goroutine`通过`go`关键字实现了，其实就是一个普通的函数。

```
go hello(a, b, c)
```

通过关键字go就启动了一个`goroutine`。来看一个例子

```
package main
import (
    "fmt"
    "runtime"
)
func say(s string) {
    for i := 0; i < 5; i++ {
        runtime.Gosched()
        fmt.Println(s)
    }
}
func main() {
    go say("world") //开一个新的Goroutines执行
    say("hello") //当前Goroutines执行
}
// 以上程序执行后将输出:
// hello
// world
// hello
// world
// hello
// world
// hello
// world
// hello
```

可以看到go关键字很方便的就实现了并发编程。

上面的多个`goroutine`运行在同一个进程里面，共享内存数据，不过设计上要遵循：不要通过共享来通信，而要通过通信来共享。

`runtime.Gosched()`表示让CPU把时间片让给别人,下次某个时候继续恢复执行该`goroutine`。

默认情况下，在Go 1.5将标识并发系统线程个数的`runtime.GOMAXPROCS`的初始值由1改为了`运行环境的CPU核数`。

但在Go 1.5以前调度器仅使用单线程，也就是说只实现了并发。想要发挥多核处理器的并行，需要程序中显式调用`runtime.GOMAXPROCS(n)`告诉调度器同时使用多个线程。`GOMAXPROCS`设置了同时运行逻辑代码的系统线程的最大数量，并返回之前的设置。如果`n < 1`，不会改变当前设置。

channels

`goroutine`运行在相同的地址空间，因此访问共享内存必须做好同步。那么`goroutine`之间如何进行数据的通信呢，Go提供了一个很好的通信机制`channel`。`channel`可以与`Unix shell`中的双向管道做类比：可以通过它发送或者接收值。这些值只能是特定的类型：`channel类型`。定义一个`channel`时，也需要定义发送到`channel`的值的类型。注意，必须使用`make`创建`channel`：

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

`channel`通过操作符`<-`来接收和发送数据

```
ch <- v    // 发送v到channel ch.
v := <-ch  // 从ch中接收数据，并赋值给v
```

把这些应用到例子中来：

```
package main
import "fmt"
func sum(a []int, c chan int) {
    total := 0
    for _, v := range a {
        total += v
    }
    c <- total  // send total to c
}
func main() {
    a := []int{7, 2, 8, -9, 4, 0}
    c := make(chan int)
    go sum(a[:len(a)/2], c)
    go sum(a[len(a)/2:], c)
    x, y := <-c, <-c  // receive from c
    fmt.Println(x, y, x + y)
}
```

默认情况下，`channel`接收和发送数据都是阻塞的，除非另一端已经准备好，这样就使得`Goroutines`同步变的更加的简单，而不需要显式的`lock`。所谓阻塞，也就是如果读取（`value := <-ch`）它将会被阻塞，直到有数据接收。其次，任何发送（`ch<-5`）将会被阻塞，直到数据被读出。无缓冲`channel`是在多个`goroutine`之间同步很棒的工具。

Buffered Channels

上面介绍了默认的非缓存类型的channel，不过Go也允许指定channel的缓冲大小，很简单，就是channel可以存储多少元素。`ch:= make(chan bool, 4)`，创建了可以存储4个元素的bool 型channel。在这个channel 中，前4个元素可以无阻塞的写入。当写入第5个元素时，代码将会阻塞，直到其他goroutine从channel 中读取一些元素，腾出空间。

```
ch := make(chan type, value)
```

当`value = 0`时，channel 是无缓冲阻塞读写的，当`value > 0`时，channel 有缓冲、是非阻塞的，直到写满 value 个元素才阻塞写入。

看一下下面这个例子，可以在自己本机测试一下，修改相应的value值

```go
package main
import "fmt"
func main() {
    c := make(chan int, 2)//修改2为1就报错，修改2为3可以正常运行
    c <- 1
    c <- 2
    fmt.Println(<-c)
    fmt.Println(<-c)
}
//修改为1报如下的错误:
//fatal error: all goroutines are asleep - deadlock!
```

Range和Close

上面这个例子中，需要读取两次c，这样不是很方便，Go考虑到了这一点，所以也可以通过range，像操作slice或者map一样操作缓存类型的channel，请看下面的例子

```go
package main
import (
    "fmt"
)
func fibonacci(n int, c chan int) {
    x, y := 1, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x + y
    }
    close(c)
}
func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

`for i := range c` 能够不断的读取 `channel` 里面的数据，直到该 `channel` 被显式的关闭。上面代码看到可以显式的关闭 `channel`，生产者通过内置函数 `close` 关闭 `channel`。关闭 `channel` 之后就无法再发送任何数据了，在消费方可以通过语法 `v, ok := <-ch` 测试 `channel` 是否被关闭。如果ok返回false，那么说明 `channel` 已经没有任何数据并且已经被关闭。

记住应该在生产者的地方关闭 `channel`，而不是消费的地方去关闭它，这样容易引起 `panic`

另外记住一点的就是 `channel` 不像文件之类的，不需要经常去关闭，只有确实没有任何发送数据了，或者想显式的结束 `range` 循环之类的

Select

上面介绍的都是只有一个 `channel` 的情况，那么如果存在多个 `channel` 的时候，该如何操作呢，Go里面提供了一个关键字 `select`，通过 `select` 可以监听 `channel` 上的数据流动。

`select` 默认是阻塞的，只有当监听的 `channel` 中有发送或接收可以进行时才会运行，当多个 `channel` 都准备好的时候，`select` 是随机的选择一个执行的。

```
package main
import "fmt"
func fibonacci(c, quit chan int) {
    x, y := 1, 1
    for {
        select {
        case c <- x:
            x, y = y, x + y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}
func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

在 `select` 里面还有 `default` 语法， `select` 其实就是类似 `switch` 的功能， `default` 就是当监听的 `channel` 都没有准备好的时候，默认执行的（ `select` 不再阻塞等待 `channel` ）。

```
select {
case i := <-c:
    // use i
default:
    // 当c阻塞的时候执行这里
}
```

超时

有时候会出现 `goroutine` 阻塞的情况，那么如何避免整个程序进入阻塞的情况呢？可以利用 `select` 来设置超时，通过如下的方式实现：

```
func main() {
    c := make(chan int)
    o := make(chan bool)
    go func() {
        for {
            select {
                case v := <- c:
                    println(v)
                case <- time.After(5 * time.Second):
                    println("timeout")
                    o <- true
                    break
            }
        }
    }()
```

```
    <- o
}
```

runtime goroutine

`runtime`包中有几个处理`goroutine`的函数：

- `Goexit`：退出当前执行的goroutine，但是defer函数还会继续调用
- `Gosched`：让出当前goroutine的执行权限，调度器安排其他等待的任务运行，并在下次某个时候从该位置恢复执行。
- `NumCPU`：返回 CPU 核数量
- `NumGoroutine`：返回正在执行和排队的任务总数
- `GOMAXPROCS`：用来设置可以并行计算的CPU核数的最大值，并返回之前的值。

错误处理

Go语言主要的设计准则是：简洁、明白，简洁是指语法和C类似，相当的简单，明白是指任何语句都是很明显的，不含有任何隐含的东西，在错误处理方案的设计中也贯彻了这一思想。

在C语言里面是通过返回`-1`或者`NULL`之类的信息来表示错误，但是对于使用者来说，不查看相应的API说明文档，根本搞不清楚这个返回值究竟代表什么意思，比如:返回0是成功，还是失败,而`Go`定义了一个叫做`error`的类型，来显式表达错误。在使用时，通过把返回的`error`变量与`nil`的比较，来判定操作是否成功。例如`os.Open`函数在打开文件失败时将返回一个不为`nil`的`error`变量

```
func Open(name string) (file *File, err error)
```

下面这个例子通过调用`os.Open`打开一个文件，如果出现错误，那么就会调用`log.Fatal`来输出错误信息：

```
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
```

类似于`os.Open`函数，标准包中所有可能出错的API都会返回一个`error`变量，以方便错误处理，这个小节将详细地介绍`error`类型的设计，和讨论开发Web应用中如何更好地处理`error`。

Error类型

error类型是一个接口类型，这是它的定义：

```
type error interface {
    Error() string
}
```

error是一个内置的接口类型，可以在`/builtin/`包下面找到相应的定义。而在很多内部包里面用到的`error`是`errors`包下面的实现的私有结构`errorString`

```
// errorString is a trivial implementation of error.
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

可以通过`errors.New`把一个字符串转化为`errorString`，以得到一个满足接口`error`的对象，其内部实现如下：

```
// New returns an error that formats as the given text.
func New(text string) error {
    return &errorString{text}
}
```

下面这个例子演示了如何使用`errors.New`：

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number")
    }
    // implementation
}
```

在下面的例子中，在调用Sqrt的时候传递的一个负数，然后就得到了`non-nil`的`error`对象，将此对象与`nil`比较，结果为`true`，所以`fmt.Println`(fmt包在处理error时会调用Error方法)被调用，以输出错误，请看下面调用的示例代码：

```
f, err := Sqrt(-1)
if err != nil {
    fmt.Println(err)
}
```

自定义Error

`error`是一个`interface`，所以在实现自己的包的时候，通过定义实现此接口的结构，就可以实现自己的错误定义，请看来自Json包的示例：

```
type SyntaxError struct {
    msg    string // 错误描述
    Offset int64  // 错误发生的位置
}

func (e *SyntaxError) Error() string { return e.msg }
```

`Offset`字段在调用`Error`的时候不会被打印，但可以通过类型断言获取错误类型，然后可以打印相应的错误信息，请看下面的例子：

```
if err := dec.Decode(&val); err != nil {
    if serr, ok := err.(*json.SyntaxError); ok {
        line, col := findLine(f, serr.Offset)
        return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
    }
    return err
}
```

需要注意的是，函数返回自定义错误时，返回值推荐设置为`error`类型，而非自定义错误类型，特别需要注意的是不应预声明自定义错误类型的变量。例如：

```
func Decode() *SyntaxError { // 错误，将可能导致上层调用者err!=nil的判断永远为true。
    var err *SyntaxError     // 预声明错误变量
    if 出错条件 {
        err = &SyntaxError{}
    }
    return err               // 错误，err永远等于非nil，导致上层调用者err!=nil的判断始终为true
}
```

原因见`http://golang.org/doc/faq#nil_error`(需科学上网)

上面例子简单的演示了如何自定义Error类型。但是如果还需要更复杂的错误处理呢？此时，来参考一下net包采用的方法：

```
package net

type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

在调用的地方，通过类型断言`err`是不是`net.Error`,来细化错误的处理，例如下面的例子，如果一个网络发生临时性错误，那么将会sleep 1秒之后重试：

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
    time.Sleep(1e9)
    continue
}
if err != nil {
    log.Fatal(err)
}
```

错误处理

Go在错误处理上采用了与C类似的检查返回值的方式，而不是其他多数主流语言采用的异常方式，这造成了代码编写上的一个很大的缺点:错误处理代码的冗余，对于这种情况是通过复用检测函数来减少类似的代码。

请看下面这个例子代码：

```
func init() {
    http.HandleFunc("/view", viewRecord)
```

```
}

func viewRecord(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        http.Error(w, err.Error(), 500)
    }
}
```

上面的例子中获取数据和模板展示调用时都有检测错误，当有错误发生时，调用了统一的处理函数`http.Error`，返回给客户端500错误码，并显示相应的错误数据。但是当越来越多的`HandleFunc`加入之后，这样的错误处理逻辑代码就会越来越多，其实可以通过自定义路由器来缩减代码

```
type appHandler func(http.ResponseWriter, *http.Request) error

func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if err := fn(w, r); err != nil {
        http.Error(w, err.Error(), 500)
    }
}
```

上面定义了自定义的路由器，然后可以通过如下方式来注册函数：

```
func init() {
    http.Handle("/view", appHandler(viewRecord))
}
```

当请求`/view`的时候逻辑处理可以变成如下代码，和第一种实现方式相比较已经简单了很多。

```
func viewRecord(w http.ResponseWriter, r *http.Request) error {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return err
    }
    return viewTemplate.Execute(w, record)
}
```

上面的例子错误处理的时候所有的错误返回给用户的都是500错误码，然后打印出来相应的错误代码，其实可以把这个错误信息定义的更加友好，调试的时候也方便定位问题，可以自定义返回的错误类型：

```
type appError struct {
    Error   error
    Message string
    Code    int
}
```

这样自定义路由器可以改成如下方式：

```
type appHandler func(http.ResponseWriter, *http.Request) *appError

func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if e := fn(w, r); e != nil { // e is *appError, not os.Error.
        c := appengine.NewContext(r)
        c.Errorf("%v", e.Error)
        http.Error(w, e.Message, e.Code)
    }
}
```

这样修改完自定义错误之后，逻辑处理可以改成如下方式：

```
func viewRecord(w http.ResponseWriter, r *http.Request) *appError {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return &appError{err, "Record not found", 404}
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        return &appError{err, "Can't display record", 500}
    }
    return nil
}
```

如上所示，在访问view的时候可以根据不同的情况获取不同的错误码和错误信息，虽然这个和第一个版本的代码量差不多，但是这个显示的错误更加明显，提示的错误信息更加友好，扩展性也比第一个更好。

总结

在程序设计中，容错是相当重要的一部分工作，在Go中它是通过错误处理来实现的，error虽然只是一个接口，但是其变化却可以有很多，可以根据自己的需求来实现不同的处理。

更多Golang资源包：https://github.com/0voice/Introduction-to-Golang