Go轻松学

itfanr

版权

原作者：@jemygraw
整理：@itfanr

参考：

- https://github.com/jemygraw/TechDoc
- https://github.com/jemygraw/GoQuickLearn

学习目录

学习目录

第一节 Go语言安装与测试

轻松友好的安装方式，多平台支持。

第二节 内置基础数据类型

认识Go提供的清晰的数据类型，很清晰，不骗你。

第三节 变量与常量定义

学语言绕不开的变量，当然Go是静态语言，变量都是有固定类型的，程序运行过程中无法改变变量类型。

第四节 控制流程

很简单，只有if，for，switch三种流程，连while都没有。

第五节 数组，切片和字典

内置高级数据类型。如果我们需要频繁使用一些功能，与其提供包支持，不如作为内置功能提供。

第六节 使用函数

代码的功能要精简，那么使用函数吧！

第七节 清楚的指针

不要怕，Go的指针不是C的指针，不可怕，但很有用。

第八节 结构体和接口

非常Cool的结构体功能，完全和C不同，很方便；另外Go提供的接口设计独特，值得研究。

第九节 并行计算

让你一见钟情的超酷功能！

第十节 使用包和测试

良好的项目管理支持也是一门语言的独特之处。标准的测试用例是控制项目Bug的有效手段。

内容

使用包和测试管理项目

使用包和测试管理项目

Go天生就是为了支持良好的项目管理体验而设计的。

包

在软件工程的实践中，我们会遇到很多功能重复的代码，比如去除字符串首尾的空格。高质量软件产品的特点就是它的部分代码是可以重用的，比如你不必每次写个函数去去除字符串首尾的空格。

我们上面讲过变量，结构体，接口和函数等，事实上所谓的包，就是把一些用的多的这些变量，结构体，接口和函数等统一放置在一个逻辑块中。并且给它们起一个名字，这个名字就叫做包名。

例如我们上面用的最多的fmt包，这个包提供了很多格式化输出的函数，你可以在自己的代码中引用这个包，来做格式化输出，而不用你自己每次去写个函数。一门成熟的语言都会提供齐全的基础功能包供人调用。

使用包有三个好处

1. 可以减少函数名称重复，因为不同包中可以存在名称相同的函数。否则得话，你得给这些函数加上前缀或者后缀以示区别。
2. 包把函数等组织在一起，方便你查找和重用。比如你想用Println()函数输出一行字符串，你可以很方便地知道它在fmt包中，直接引用过来用就可以了。
3. 使用包可以加速程序编译。因为包是预编译好的，你改动自己代码得时候，不必每次去把包编译一下。

创建包

我们现在来举个例子，用来演示Go的项目管理。

首先我们在目录 `/Users/jemy/JemyGraw/GoLang` 下面创建文件夹 `pkg_demo` 。然后在 `pkg_demo` 里面创建 `src` 文件夹
。然后再在 `main` 文件夹里面创建 `main.go` 文件。另外为了演示包的创建，我们在 `src` 目录下面创建文件夹 `net.duokr` ，然后再在 `net.duokr` 文件夹里面创建 `math` 文件夹，这个文件夹名称就是这个文件夹下面go文件的包名称。然后我们再创建一个 `math_pkg.go` 文件，之所以取这个名字而不是 `math.go` 只是为了说明这个文件名称和包名不需要一致。然后我们还创建了一个 `math_pkg_test.go` 文件作为包的测试用例文件。整体结构如下：

```
.
└── src
    ├── main
    │   ├── build.sh
    │   └── main.go
    └── net.duokr
        └── math
            ├── math_pkg.go
            └── math_pkg_test.go
```

其中build.sh是我们为了编译这个项目而写的脚本，因为编译项目需要几条命令，把它写在脚本文件中方便使用。另外为了能够让build.sh能够执行，使用`chmod +x build.sh`为它赋予可执行权限。build.bat是Windows下面的编译脚本。

我们来看一下`math_pkg.go`的定义：

```
package math

func Add(a, b int) int {
    return a + b
}
func Subtract(a, b int) int {
    return a - b
}
func Multiply(a, b int) int {
    return a * b
}

func Divide(a, b int) int {
    if b == 0 {
        panic("Can not divided by zero")
    }
    return a / b
}
```

首先是包名，然后是几个函数定义，这里我们会发现这些`函数定义首字母都是大写`，`Go规定了只有首字母大写的函数才能从包导出使用，即其他调用这个包中函数的代码只能调用那些导出的`。

我们再看一下`main.go`的定义：

```
package main

import (
    "fmt"
    math "net.duokr/math"
)

func main() {
    var a = 100
    var b = 200

    fmt.Println("Add demo:", math.Add(a, b))
    fmt.Println("Substract demo:", math.Subtract(a, b))
    fmt.Println("Multiply demo:", math.Multiply(a, b))
    fmt.Println("Divide demo:", math.Divide(a, b))
}
```

在main.go里面，我们使用import关键字引用我们自定义的包math，引用的方法是从main包平行的文件夹net.duokr开始，后面跟上包名math。这里面我们给这个长长的包名起了一个别名就叫math。然后分别调用math包里面的函数。

最后我们看一下我们的编译脚本：

```
export GOPATH=$GOPATH:/Users/jemy/JemyGraw/GoLang/pkg_demo
export GOBIN=/Users/jemy/JemyGraw/GoLang/pkg_demo/bin
go build net.duokr/math
go build main.go
go install main
```

第一行，我们将项目路径加入GOPATH中，这样待会儿编译main.go的时候才能找到我们自定义的包；

第二行，我们设置本项目的安装目录，第五行的命令将编译好的文件放到这个目录下面；

第三行，我们编译我们的自定义包；

第四行，我们编译我们main.go文件；

第五行，将编译好的文件安装到指定目录下。

这里还有一个Windows下面的编译脚本build.bat：

```
@echo off
set GOPATH=GOPATH;C:\JemyGraw\GoLang\pkg_demo
set GOBIN=C:\JemyGraw\GoLang\pkg_demo\bin
go build net.duokr\math
go build main.go
go install main
```

好了，运行脚本编译一下，在main文件夹和bin文件夹下面都会生成一个可执行文件。

这个时候文件夹结构为：

```
.
├── bin
│   └── main
├── pkg
│   └── darwin_386
│       └── net.duokr
│           └── math.a
└── src
    ├── main
    │   ├── build.bat
    │   ├── build.sh
    │   ├── main
    │   └── main.go
    └── net.duokr
        └── math
            ├── math_pkg.go
            └── math_pkg_test.go
```

运行一下，输出结果为：

```
Add demo: 300
Substract demo: -100
Multiply demo: 20000
Divide demo: 0
```

好了，包的使用介绍完毕，我们再来看一下测试用例怎么写。

测试

在上面的例子中，我们发现我们自定义的包下面还有一个math_pkg_test.go文件，这个文件包含了本包的一些测试用例。而且Go会把以 `_test.go` 结尾的文件当作是测试文件。

测试怎么写，当然是用assert来判断程序的运行结果是否和预期的相同了。

我们来看看这个math包的测试用例。

```
package math

import (
    "testing"
)

func TestAdd(t *testing.T) {
    var a = 100
    var b = 200

    var val = Add(a, b)
    if val != a+b {
        t.Error("Test Case [", "TestAdd", "] Failed!")
    }
}

func TestSubtract(t *testing.T) {
    var a = 100
    var b = 200

    var val = Subtract(a, b)
    if val != a-b {
        t.Error("Test Case [", "TestSubtract", "] Failed!")
    }
}

func TestMultiply(t *testing.T) {
    var a = 100
    var b = 200

    var val = Multiply(a, b)
    if val != a*b {
        t.Error("Test Case [", "TestMultiply", "] Failed!")
    }
}

func TestDivideNormal(t *testing.T) {
    var a = 100
    var b = 200

    var val = Divide(a, b)
    if val != a/b {
        t.Error("Test Case [", "TestDivideNormal", "] Failed!")
    }
}
```

将路径切换到测试文件所在目录，运行 `go test` 命令，go会自动测试所有的测试用例。

在上面的例子中，测试用例的特点是以函数名以 `Test` 开始，而且具有唯一参数 `t *testing.T` 。

小结

Go提供的包和用例测试是构建优秀的软件产品的基础，只要我们不断学习，努力去做，一定可以做的很好。

Go语言环境安装与测试

Go语言环境安装与测试

安装

现在来谈谈Go语言的安装，要使用Go来编写程序首先得把环境搭建起来。
Go的语言环境搭建还是比较简单的。Google提供了Windows和Mac的安装包，所以去下载一下安装就可以了。
对于Linux的系统，可以使用系统提供的包安装工具来安装。

Go的下载地址

https://code.google.com/p/go/downloads/list

Windows

对于Windows系统，Go提供了两种不同的安装包，分别对应32位的系统和64位的系统，安装的时候根据自己的系统实际情况选择下载包。Windows下面提供的是msi格式的安装包，这种包是可执行文件，直接双击安装就可以了。安装完成之后，安装程序会自动地将安装完的Go的根目录下的bin目录加入系统的PATH环境变量里面。所以直接打开命令行，输入go，就可以看到一些提示信息了。

Mac

如果是新买的Mac，里面可能自带了一个go的可执行文件，在路径 `/etc/paths.d/` 下面，就是一个go可执行文件。如果我们需要安装从官网下载的dmg安装包，先要把这个文件删除掉。可以用 `sudo rm /etc/paths.d/go` 来删除。然后自动安装dmg之后，要使用 `export PATH` 的方法将安装好的Go目录下面的bin目录加入PATH中。一般安装完之后路径为 `/usr/local/go` ，所以你可以用下面的方法：
首先切换到自己的用户目录

```
cd ~
```

然后

```
vim .profile
```

加入一行

```
export PATH=/usr/local/go/bin:$PATH
```

就可以了。

Linux

Linux的发行版有很多，可以根据不同系统提供的包管理工具来安装Go，不过可能系统包管理工具提供的不是最新的Go版本。在这种情况下，你可以去下载最新的tar包。
然后使用下面的方法

```
sudo tar -C /usr/local -xzf go1.2.linux-386.tar.gz
```

如果是64位的系统，用下面的方法

```
sudo tar -C /usr/local -xzf go1.2.linux-amd64.tar.gz
```

当然，这样的方式只是将安装包解压拷贝到 `/usr/local/` 下面。你还需要使用 `export PATH` 的方式将Go的bin目录加入PATH。
方法和上面Mac介绍的一样。
另外如果你不是将Go安装到 `/usr/local` 目录下面，你还需要设置一个GOROOT环境变量。比如你安装到你自己的文件夹下面，比如叫jemy的用户的路径是 `/home/jemy` ，那么你安装到这个目录的Go路径为 `/home/jemy/go` ，那么在 `export PATH` 之前，你还需要使用下面的命令。

```
export GOROOT=/home/jemy/go
```

总结一下，如果你默认安装路径为 `/usr/local/go` ，那么只需要用

```
export PATH=$PATH:/usr/local/go/bin
```

就可以了。
如果不是默认路径则需要这样

```
export GOROOT=/home/jemy/go
export PATH=$PATH:/$GROOT/bin
```

上面的 `/home/jemy` 是根据实际安装的路径情况来确定。

最后说一下go的最基本的三个命令

1.查看版本号

```
go version
```

结果为

```
duokr:~ jemy$ go version
go version go1.2 darwin/386
```

2.格式化go代码文件

```
go fmt file_name.go
```

3.运行单个go代码文件

```
go run file_name.go
```

测试

`生` `死` `hello world`

学习计算机的， 绕不开的三件事。

有谁安装好语言环境，不试一下hello world的？

```
//main包, 凡是标注为main包的go文件都会被编译为可执行文件
package main

//导入需要使用的包
import (
    "fmt" //支持格式化输出的包,就是format的简写
)

//主函数,程序执行入口
func main() {
    /*
        输出一行hello world
        Println函数就是print line的意思
    */
    fmt.Println("hello world")
}
```

然后使用`go run helloworld.go`来运行这个例子。如果安装成功，那么会输出一行`hello world`。

PS

`Windows7可以在文件所在目录下面使用Shift+右键，快速打开已定位到所在目录的命令行窗口。直接输入`

Go语言内置基础数据类型

Go语言内置基础数据类型

在自然界里面，有猫，有狗，有猪。有各种动物。每种动物都是不同的。
比如猫会喵喵叫，狗会旺旺叫，猪会哼哼叫。。。
Stop!!!
好了，大家毕竟不是幼儿园的小朋友。介绍到这里就可以了。

论点就是每个东西都有自己归属的类别(Type)。
那么在Go语言里面，每个变量也都是有类别的，这种类别叫做 数据类型(Data Type) 。
Go的数据类型有两种：一种是 语言内置的数据类型 ，另外一种是
通过语言提供的自定义数据类型方法自己定义的自定义数据类型 。

先看看语言 内置的基础数据类型

数值型(Number)

数值型有 三种 ，一种是 整数类型 ，另外一种是 带小数的类型 (一般计算机里面叫做 浮点数类型)，还有一种 虚数类型 。

整数类型不用说了，和数学里面的是一样的。和数学里面不同的地方在于计算机里面 正整数和零 统称为 无符号整型 ，而 负整数 则称为 有符号整型 。

Go的内置整型有 uint8 ， uint16 ， uint32 ， uint64 ， int8 ， int16 ， int32 和 int64 。其中 u 开头的类型就是 无符号整型 。无符号类型能够表示正整数和零。而有符号类型除了能够表示正整数和零外，还可以表示负整数。
另外还有一些别名类型，比如 byte 类型，这个类型和 uint8 是一样的，表示 字节类型 。另外一个是 rune类型 ，这个类型和 int32 是一样的，用来表示 unicode的代码点 ，就是unicode字符所对应的整数。

Go还定义了三个 依赖系统 的类型， uint ， int 和 uintptr 。因为在32位系统和64位系统上用来表示这些类型的位数是不一样的。

对于32位系统

uint=uint32
int=int32
uintptr为32位的指针

对于64位系统

uint=uint64

int=int64
uintptr为64位的指针

至于类型后面跟的数字8，16，32或是64则表示用来表示这个类型的位不同，
位越多，能表示的整数范围越大。
比如对于用N位来表示的整数，如果是有符号的整数，能够表示的整数范围为-2^(N-1) ~ 2^(N-1) - 1；如果是无符号的整数，则能表示的整数范围为0 ~ 2^N。

Go的浮点数类型有两种，float32 和 float64。float32又叫单精度浮点型，float64又叫做双精度浮点型。其 最主要的区别就是小数点后面能跟的小数位数不同。

另外Go还有两个其他语言所没有的类型，虚数类型。complex64 和 complex128。

对于数值类型，其所共有的操作为 加法(+)，减法(-)，乘法(*) 和 除法(/)。另外对于整数类型，还定义了 求余运算(%)

求余运算为整型所独有。如果对浮点数使用求余，比如这样

```
package main

import (
    "fmt"
)

func main() {
    var a float64 = 12
    var b float64 = 3

    fmt.Println(a % b)
}
```

编译时候会报错

```
invalid operation: a % b (operator % not defined on float64)
```

所以，这里我们可以知道所谓的数据类型有两层意思，一个是定义了该类型所能表示的数，另一个是定义了 该类型所能进行的操作。
简单地说，对于一只小狗，你能想到的一定是狗的面貌和它会汪汪叫，而不是猫的面容和喵喵叫。

字符串类型(String)

字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来的。（对于汉字，通常由多个字节组成）。这就是说，传统的字符串是由字符组成的，而 Go的字符串不同，是由字节组成 的。这一点需要注意。

字符串的表示很简单。用(双引号"")或者(``号)来描述。

```
"hello world"
```

或者

```
`hello world`
```

唯一的区别是，**双引号之间的转义字符会被转义，而``号之间的转义字符保持原样不变。**

```
package main

import (
    "fmt"
)

func main() {
    var a = "hello \n world"
    var b = `hello \n world`

    fmt.Println(a)
    fmt.Println("----------")
    fmt.Println(b)
}
```

输出结果为

```
hello
 world
----------
hello \n world
```

字符串所能进行的一些基本操作包括:

（1）获取字符长度

（2）获取字符串中单个字节

（3）字符串连接

```
package main

import (
    "fmt"
)

func main() {
    var a string = "hello"
    var b string = "world"

    fmt.Println(len(a))
    fmt.Println(a[1])
    fmt.Println(a + b)
}
```

输出如下

```
5
101
helloworld
```

这里我们看到a[1]得到的是一个整数，这就证明了上面`"Go的字符串是由字节组成的这句话"`。我们还可以再验证一下。

```
package main

import (
    "fmt"
)

func main() {
    var a string = "你"
    var b string = "好"
    fmt.Println(len(a))
    fmt.Println(len(b))
    fmt.Println(len(a + b))
    fmt.Println(a[0])
    fmt.Println(a[1])
    fmt.Println(a[2])
}
```

输出

```
3
3
6
228
189
160
```

我们开始的时候，从上面的三行输出知道，"你"和"好"分别是用三个字节组成的。我们依次获取a的三个字节，输出，得到结果。

布尔型(Bool)

布尔型是表示 `真值` 和 `假值` 的类型。可选值为 `true` 和 `false` 。

所能进行的操作如下：
`&& and 与`
`|| or 或`
`！ not 非`

Go的布尔型取值 `就是true` 或 `false` 。 `任何空值(nil)或者零值(0, 0.0, "")都不能作为布尔型来直接判断` 。

```
package main

import (
    "fmt"
)

func main() {
    var equal bool
    var a int = 10
    var b int = 20
    equal = (a == b)
    fmt.Println(equal)
}
```

输出结果

```
false
```

下面是错误的用法

```
package main

import (
    "fmt"
)

func main() {
    if 0 {
        fmt.Println("hello world")
    }
    if nil {
        fmt.Println("hello world")
    }
    if "" {
         fmt.Println("hello world")
    }
}
```

编译错误

```
./t.go:8: non-bool 0 (type untyped number) used as if condition
./t.go:11: non-bool nil used as if condition
./t.go:14: non-bool "" (type untyped string) used as if condition
```

上面介绍的是Go语言内置的基础数据类型。

变量和常量定义

变量和常量定义

现在我们讨论一下Go语言的变量定义。

变量定义

所谓的变量就是一个拥有指定 `名称` 和 `类型` 的 `数据存储位置` 。
在上面我们使用过变量的定义，现在我们来仔细看一个例子。

```
package main

import (
    "fmt"
)

func main() {
    var x string = "hello world"
    fmt.Println(x)
}
```

变量的定义首先使用`var`关键字，然后指定变量的名称`x`，再指定变量的类型`string`，在本例中，还对变量`x`进行了赋值，然后在命令行输出该变量。Go这种变量定义的方式和其他的语言有些不同，但是在使用的过程中，你会逐渐喜欢的。当然上面的变量定义方式还可以如下，即先定义变量，再赋值。

```
package main

import (
    "fmt"
)

func main() {
    var x string
    x = "hello world"
    fmt.Println(x)
}
```

或者是直接赋值，让Go语言推断变量的类型。如下：

```
package main

import (
    "fmt"
)

func main() {
    var x = "hello world"
    fmt.Println(x)
}
```

当然，上面变量的定义还有一种`快捷方式`。如果你知道变量的初始值，完全可以像下面这样定义变量，完全让`Go来推断语言的类型`。这种定义的方式连关键字`var`都省略掉了。

```
package main

import (
    "fmt"
)

func main() {
    x := "hello world"
    fmt.Println(x)
}
```

注意：上面这种使用`:=`方式定义变量的方式`只能用在函数内部`。

```
package main

import (
    "fmt"
)

x:="hello world"
func main() {
    y := 10
    fmt.Println(x)
    fmt.Println(y)
}
```

对于上面的变量定义x是无效的。会导致编译错误：

```
./test_var_quick.go:7: non-declaration statement outside function body
```

不过我们对上面的例子做下修改，比如这样是可以的。也就是使用var关键字定义的时候，如果给出初始值，就不需要显式指定变量类型。

```
package main

import (
    "fmt"
)

var x = "hello world"

func main() {
    y := 10
    fmt.Println(x)
    fmt.Println(y)
}
```

`变量`之所以称为变量，就是因为`它们的值在程序运行过程中可以发生变化`，但是

它们的变量类型是无法改变的。因为Go语言是静态语言，并不支持程序运行过程中变量类型发生变化。比如如果你强行将一个字符串值赋值给定义为int的变量，那么会发生编译错误。即使是强制类型转换也是不可以的。强制类型转换只支持同类的变量类型。比如数值类型之间强制转换。

下面我们看几个例子：

```
package main

import (
    "fmt"
)

func main() {
    var x string = "hello world"
    fmt.Println(x)
    x = "i love go language"
    fmt.Println(x)
}
```

本例子演示变量的值在程序运行过程中发生变化，结果输出为

```
hello world
i love go language
```

我们尝试不同类型的变量之间转换

```
package main

import (
    "fmt"
)

func main() {
    var x string = "hello world"
    fmt.Println(x)
    x = 11
    fmt.Println(x)
}
```

在本例子中，如果试图将一个数值赋予字符串变量x，那么会发生错误：

```
./test_var.go:10: cannot use 11 (type int) as type string in assignment
```

上面的意思就是无法将整型数值11当作字符串赋予给字符串变量。

但是同类的变量之间是可以强制转换的，如浮点型和整型之间的转换。

```
package main

import (
    "fmt"
)

func main() {
    var x float64 = 32.35
    fmt.Println(x)
    fmt.Println(int(x))
}
```

输出的结果为

```
32.35
32
```

变量命名

上面我们看了一些变量的使用方法，那么定义一个变量名称，有哪些要求呢？
这里我们要注意，`Go的变量名称必须以字母或下划线(_)开头，后面可以跟字母，数字，或者下划线(_)`。
除此之外，Go语言并不关心你如何定义变量。我们通用的做法是定义一个用户友好的变量。假设你需要定义一个狗狗的年龄，那么使用dog_age作为变量名称要好于用x来定义变量。

变量作用域

现在我们再来讨论一下变量的作用域。所谓作用域就是可以有效访问变量的区域。比如很简单的，你不可能在一个函数func_a里面访问另一个函数func_b里面定义的局部变量x。所以变量的作用域目前分为两类，一个是`全局变量`，另一个是`局部变量`。下面我们看个全局变量的例子：

```
package main

import (
    "fmt"
)

var x string = "hello world"

func main() {
    fmt.Println(x)
}
```

这里变量x定义在main函数之外，但是main函数仍然可以访问x。全局变量的作用域是该包中所有的函数。

```
package main

import (
    "fmt"
)

var x string = "hello world"

func change() {
    x = "i love go"
}
func main() {
    fmt.Println(x)
    change()
    fmt.Println(x)
}
```

在上面的例子用，我们用了change函数改变了x的值。输出结果如下：

```
hello world
i love go
```

我们再看一下局部变量的例子。

```
package main

import (
    "fmt"
)

func change() {
    x := "i love go"
}
func main() {
    fmt.Println(x)
}
```

该例子中main函数试图访问change函数中定义的局部变量x，结果发生了下面的错误(未定义的变量x)：

```
./test_var.go:11: undefined: x
```

常量

Go语言也支持常量定义。所谓 常量就是在程序运行过程中保持值不变的变量定义 。常量的定义和变量类似，只是用 const 关键字替换了var关键字，另外常量在定义的时候 必须有初始值 。

```
package main

import (
    "fmt"
)

func main() {
    const x string = "hello world"
    const y = "hello world"
    fmt.Println(x)
    fmt.Println(y)
}
```

这里有一点需要注意，变量定义的类型推断方式 := 不能够用来定义常量。因为常量的值是在编译的时候就已经确定的，但是变量的值则是运行的时候才使用的。这样常量定义就无法使用变量类型推断的方式了。

常量的值在运行过程中是无法改变的，强制改变常量的值是无效的。

```
package main

import (
    "fmt"
)

func main() {
    const x string = "hello world"
    fmt.Println(x)
    x = "i love go language"
    fmt.Println(x)
}
```

比如上面的例子就会报错

```
./test_var.go:10: cannot assign to x
```

我们再看一个Go包math里面定义的常量Pi，用它来求圆的面积。

```
package main

import (
    "fmt"
    "math"
)

func main() {
    var radius float64 = 10
    var area = math.Pow(radius, 2) * math.Pi
    fmt.Println(area)
}
```

多变量或常量定义

Go还提供了一种 同时定义多个变量或者常量 的快捷方式。

```
package main

import (
    "fmt"
)

func main() {
    var (
        a int     = 10
        b float64 = 32.45
        c bool    = true
    )
    const (
        Pi   float64 = 3.14
        True bool    = true
    )

    fmt.Println(a, b, c)
    fmt.Println(Pi, True)
}
```

程序控制结构

程序控制结构

虽然剧透可耻，但是为了体现Go语言的设计简洁之处，必须要先剧透一下。

Go语言的控制结构关键字只有

`if..else if..else` ， `for` 和 `switch` 。

而且在Go中，为了避免格式化战争，对程序结构做了统一的强制的规定。看下下面的例子。

请比较一下A程序和B程序的不同之处。

A程序

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("hello world")
}
```

B程序

```
package main

import (
    "fmt"
)

func main()
{
    fmt.Println("hello world")
}
```

还记得我们前面的例子中，`{}`的格式是怎么样的么？在上面的两个例子中只有A例的写法是对的。因为在Go语言中，强制了`{}`的格式。如果我们试图去编译B程序，那么会发生如下的错误提示。

```
./test_format.go:9: syntax error: unexpected semicolon or newline before {
```

if..else if..else

if..else if..else 用来判断一个或者多个条件，然后根据条件的结果执行不同的程序块。举个简单的例子。

```
package main

import (
    "fmt"
)

func main() {
    var dog_age = 10

    if dog_age > 10 {
        fmt.Println("A big dog")
    } else if dog_age > 1 && dog_age <= 10 {
        fmt.Println("A small dog")
    } else {
        fmt.Println("A baby dog")
    }
}
```

上面的例子判断狗狗的年龄如果 (if) 大于10就是一个大狗；否则判断 (else if) 狗狗的年龄是否小于等于10且大于1，这个时候狗狗是小狗狗。否则 (else) 的话（就是默认狗狗的年龄小于等于1岁），那么狗狗是Baby狗狗。

在上面的例子中，我们还可以发现Go的if..else if..else语句的判断条件一般都不需要使用 () 。当然如果你还是愿意写，也是对的。另外如果为了将某两个或多个条件绑定在一起判断的话，还是需要括号 () 的。

比如下面的例子也是对的。

```
package main

import (
    "fmt"
)

func main() {
    const Male = 'M'
    const Female = 'F'

    var dog_age = 10
    var dog_sex = 'M'

    if (dog_age == 10 && dog_sex == 'M') {
        fmt.Println("dog")
    }
}
```

但是如果你使用Go提供的格式化工具来格式化这段代码的话，Go会智能判断你的括号是否必须有，否则的话，会帮你去掉的。你可以试试。

```
go fmt test_bracket.go
```

然后你会发现，咦？！果真被去掉了。

另外因为每个判断条件的结果要么是true要么是false，所以可以使用`&&`，`||`来连接不同的条件。使用`!`来对一个条件取反。

switch

switch的出现是为了解决某些情况下使用if判断语句带来的繁琐之处。

例如下面的例子：

```
package main

import (
    "fmt"
)

func main() {
    //score 为 [0,100]之间的整数
    var score int = 69

    if score >= 90 && score <= 100 {
        fmt.Println("优秀")
    } else if score >= 80 && score < 90 {
        fmt.Println("良好")
    } else if score >= 70 && score < 80 {
        fmt.Println("一般")
    } else if score >= 60 && score < 70 {
        fmt.Println("及格")
    } else {
        fmt.Println("不及格")
    }
}
```

在上面的例子中，我们用if..else if..else来对分数进行分类。这个只是一般的情况下if判断条件的数量。如果if..else if..else的条件太多的话，我们可以使用switch来优化程序。比如上面的程序我们还可以这样写：

```
package main

import (
    "fmt"
)

func main() {
    //score 为 [0,100]之间的整数
    var score int = 69

    switch score / 10 {
    case 10:
    case 9:
        fmt.Println("优秀")
    case 8:
        fmt.Println("良好")
    case 7:
        fmt.Println("一般")
    case 6:
        fmt.Println("及格")
    default:
        fmt.Println("不及格")
    }
}
```

关于switch的几点说明如下：

(1) switch的判断条件可以为任何数据类型。

```
package main

import (
    "fmt"
)

func main() {
    var dog_sex = "F"
    switch dog_sex {
    case "M":
        fmt.Println("A male dog")
    case "F":
        fmt.Println("A female dog")
    }
}
```

(2) 每个case后面跟的是一个完整的程序块，该程序块`不需要{}`，也`不需要break结尾`，因为每个case都是独立的。

(3) 可以为switch提供一个默认选项default，在上面所有的case都没有满足的情况下，默认执行default后面的语句。

for

for用在Go语言的循环条件里面。比如说要你输出1...100之间的自然数。最笨的方法就是直接这样。

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(1)
    fmt.Println(2)
    ...
    fmt.Println(100)
}
```

这个不由地让我想起一个笑话。

> 以前一个地主的儿子学习写字，只学了三天就把老师赶走了。因为在这三天里面他学写了一，二，三。他觉得写字真的太简单了，不就是画横线嘛。于是有一天老爹过寿，让他来记送礼的人名单。直到中午还没有记完，老爹很奇怪就去问他怎么了。他哭着说，“不知道这个人有什么毛病，姓什么不好，姓万”。

哈哈，回来继续。我们看到上面的例子也是如地主的儿子那样就不好了。所以，我们必须使用循环结构。我们用for的循环语句来实现上面的例子。

```
package main

import (
    "fmt"
)

func main() {
    var i int = 1

    for ; i <= 100; i++ {
        fmt.Println(i)
    }
}
```

在上面的例子中，首先初始化变量i为1，然后在for循环里面判断是否小于等于100，如果是的话，输出i，然后再使用i++来将i的值自增1。上面的例子，还有一个更好的写法，就是将i的定义和初始化也放在for里面。如下：

```
package main

import (
    "fmt"
)

func main() {
    for i := 1; i <= 100; i++ {
        fmt.Println(i)
    }
}
```

在Go里面没有提供while关键字，如果你怀念while的写法也可以这样：

```
package main

import (
    "fmt"
)

func main() {
    var i int = 1

    for i <= 100 {
        fmt.Println(i)
        i++
    }
}
```

或许你会问，如果我要死循环呢？是不是`for true`？呵呵，不用了，直接这样。

```
for{
    ...
}
```

以上就是Go提供的全部控制流程了。

再复习一下，Go只提供了：

if

```
if ...{
    ...
}else if ...{
    ...
}else{
    ...
}
```

switch

```
switch(...){
case ...:
        ...
case ...:
        ...
...

default:
         ...
}
```

for

```
for ...; ...; ...{
    ...
}

for ...{
    ...
}

for{
    ...
}
```

数组，切片和字典

数组、切片和字典

在上面的章节里面，我们讲过Go内置的基本数据类型。现在我们来看一下Go内置的高级数据类型，数组，切片和字典。

数组(Array)

数组是一个具有 相同数据类型 的元素组成的 固定长度 的 有序集合 。比如下面的例子

```
var x [5]int
```

表示数组x是一个整型数组，而且数值的长度为5。

Go提供了几种不同的数组定义方法。

最基本的方式就是使用var关键字来定义，然后依次给元素赋值 。 对于没有赋值的元素，默认为零值 。

比如对于整数，零值就是0，浮点数，零值就是0.0，字符串，零值就是""，对象零值就是nil。

```
package main

import (
    "fmt"
)

func main() {
    var x [5]int
    x[0] = 2
    x[1] = 3
    x[2] = 3
    x[3] = 2
    x[4] = 12
    var sum int
    for _, elem := range x {
        sum += elem
    }
    fmt.Println(sum)
}
```

在上面的例子中，我们首先使用`var`关键字来声明，然后给出数组名称`x`，最后说明数组为整型数组，长度为5。然后我们使用索引方式给数组元素赋值。在上面的例子中，我们还使用了一种遍历数组元素的方法。该方法利用Go语言提供的内置函数range来遍历数组元素。
`range函数可以用在数组，切片和字典上面`。当 range来遍历数组的时候返回数组的索引和元素值。在这里我们是对数组元素求和，所以我们对索引不感兴趣。在Go语言里面，
`当你对一个函数返回值不感兴趣的话，可以使用下划线(_)来替代它`。另外这里如果我们真的定义了一个索引，在循环结构里面却没有使用索引，Go语言编译的时候还是会报错的。所以用下划线来替代索引变量也是唯一之举了。最后我们输出数组元素的和。

还有一种方式，如果知道了数组的初始值。可以像下面这样定义。

```
package main

import (
    "fmt"
)

func main() {
    var x = [5]int{1, 2, 3, 4}
    x[4] = 5

    var sum int
    for _, i := range x {
        sum += i
    }
    fmt.Println(sum)
}
```

当然，即使你不知道数组元素的初始值，也可以使用这样的定义方式。

```
package main

import (
    "fmt"
)

func main() {
    var x = [5]int{}
    x[0] = 1
    x[1] = 2
    x[2] = 3
    x[3] = 4
    x[4] = 5

    var sum int
    for _, i := range x {
        sum += i
    }
    fmt.Println(sum)
}
```

在这里我们需要特别重视数组的一个特点，就是数组是有固定长度的。

但是如果我们有的时候也可以不显式指定数组的长度，而是使用 ... 来替代数组长度，Go语言会自动计算出数组的长度。不过这种方式定义的数组一定是有初始化的值的。

```
package main

import (
    "fmt"
)

func main() {
    var x = [...]string{
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"}

    for _, day := range x {
        fmt.Println(day)
    }
}
```

在上面的例子中，还需要注意一点就是如果将数组元素定义在不同行上面，那么最后一个元素后面必须跟上 } 或者 , 。上面的例子也可以是这样的。

```
package main

import (
    "fmt"
)

func main() {
    var x = [...]string{
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday",
    }

    for _, day := range x {
        fmt.Println(day)
    }
}
```

Go提供的这种可以自动计算数组长度的方法在调试程序的时候特别方便，假设我们注释掉上面数组x的最后

切片(Slice)

在上面我们说过数组是有固定长度的有序集合。这也就是说一旦数组长度定义，你将无法在数组里面多添

加哪怕一个元素。数组的这种特点有的时候会成为很大的缺点，尤其是当数组的元素个数不确定的情况下。

所以 切片 诞生了。

切片和数组很类似，甚至你可以理解成数组的子集。但是
切片有一个数组所没有的特点，那就是切片的长度是可变的 。

严格地讲，切片有 容量(capacity) 和 长度(length) 两个属性。

首先我们来看一下切片的定义。切片有两种定义方式，一种是先声明一个变量是切片，然后使用内置函数make去初始化这个切片。另外一种是通过取数组切片来赋值。

```
package main

import (
    "fmt"
)

func main() {
    var x = make([]float64, 5)
    fmt.Println("Capcity:", cap(x), "Length:", len(x))
    var y = make([]float64, 5, 10)
    fmt.Println("Capcity:", cap(y), "Length:", len(y))

    for i := 0; i < len(x); i++ {
        x[i] = float64(i)
    }
    fmt.Println(x)

    for i := 0; i < len(y); i++ {
        y[i] = float64(i)
    }
    fmt.Println(y)
}
```

输出结果为

```
Capcity: 5 Length: 5
Capcity: 10 Length: 5
[0 1 2 3 4]
[0 1 2 3 4]
```

上面我们首先用make函数定义切片x，这个时候x的容量是5，长度也是5。然后使用make函数定义了切片y，这个时候y的容量是10，长度是5。然后我们再分别为切片x和y的元素赋值，最后输出。

所以使用make函数定义切片的时候，有 两种方式 ，一种
只指定长度，这个时候切片的长度和容量是相同的 。另外一种是 同时指定切片长度和容量 。虽然切片的

容量可以大于长度，但是赋值的时候要注意最大的索引仍然是len(x) - 1。否则会报索引超出边界错误。

另外一种是通过数组切片赋值，采用 [low_index:high_index] 的方式获取数值切片，其中切片元素包括low_index的元素，但是 不包括high_index的元素。

```
package main

import (
    "fmt"
)

func main() {
    var arr1 = [5]int{1, 2, 3, 4, 5}
    var s1 = arr1[2:3]
    var s2 = arr1[:3]
    var s3 = arr1[2:]
    var s4 = arr1[:]
    fmt.Println(s1)
    fmt.Println(s2)
    fmt.Println(s3)
    fmt.Println(s4)
}
```

输出结果为

```
[3]
[1 2 3]
[3 4 5]
[1 2 3 4 5]
```

在上面的例子中，我们还省略了low_index或high_index。如果省略了low_index，那么等价于从索引0开始；如果省略了high_index，则默认high_index等于len(arr1)，即切片长度。

这里为了体现切片的长度可以变化，我们看一下下面的例子：

```
package main

import (
    "fmt"
)

func main() {
    var arr1 = make([]int, 5, 10)
    for i := 0; i < len(arr1); i++ {
        arr1[i] = i
    }
    fmt.Println(arr1)

    arr1 = append(arr1, 5, 6, 7, 8)
    fmt.Println("Capacity:", cap(arr1), "Length:", len(arr1))
    fmt.Println(arr1)
}
```

输出结果为

```
[0 1 2 3 4]
Capacity: 10 Length: 9
[0 1 2 3 4 5 6 7 8]
```

这里我们初始化arr1为容量10，长度为5的切片，然后为前面的5个元素赋值。然后输出结果。然后我们再使用Go内置方法append来为arr1追加四个元素，这个时候再看一下arr1的容量和长度以及切片元素，我们发现切片的长度确实变了。

另外我们再用 append 方法给arr1多追加几个元素，试图超过arr1原来定义的容量大小。

```
package main

import (
    "fmt"
)

func main() {
    var arr1 = make([]int, 5, 10)
    for i := 0; i < len(arr1); i++ {
        arr1[i] = i
    }

    arr1 = append(arr1, 5, 6, 7, 8, 9, 10)
    fmt.Println("Capacity:", cap(arr1), "Length:", len(arr1))
    fmt.Println(arr1)
}
```

输出结果为

```
Capacity: 20 Length: 11
[0 1 2 3 4 5 6 7 8 9 10]
```

我们发现arr1的长度变为11，因为元素个数现在为11个。另外我们发现arr1的容量也变了，变为原来的两倍。这是因为
Go在默认的情况下，如果追加的元素超过了容量大小，Go会自动地重新为切片分配容量，容量大小为原来
。

上面我们介绍了，可以 使用append函数给切片增加元素 ，现在我们再来介绍一个
copy函数用来从一个切片拷贝元素到另一个切片 。

```
package main

import (
    "fmt"
)

func main() {
    slice1 := []int{1, 2, 3, 4, 5, 6}
    slice2 := make([]int, 5, 10)
    copy(slice2, slice1)
    fmt.Println(slice1)
    fmt.Println(slice2)
}
```

输出结果

```
[1 2 3 4 5 6]
[1 2 3 4 5]
```

在上面的例子中，我们将slice1的元素拷贝到slice2，因为slice2的长度为5，所以最多拷贝5个元素。

总结一下，数组和切片的区别就在于 [] 里面是否有数字或者 ... 。因为数值长度是固定的，而切片是可变的。

字典(Map)

字典是一组 无序的 ， 键值对 的 集合 。

字典也叫做 关联数组 ，因为数组通过 索引 来查找元素，而字典通过 键 来查找元素。当然，很显然的，字典的键是不能重复的。如果试图赋值给同一个键，后赋值的值将覆盖前面赋值的值。

字典的定义也有两种，一种是 初始化数据 的定义方式，另一种是 使用神奇的make函数 来定义。

```
package main

import (
    "fmt"
)

func main() {
    var x = map[string]string{
        "A": "Apple",
        "B": "Banana",
        "O": "Orange",
        "P": "Pear",
    }

    for key, val := range x {
        fmt.Println("Key:", key, "Value:", val)
    }
}
```

输出结果为

```
Key: A Value: Apple
Key: B Value: Banana
Key: O Value: Orange
Key: P Value: Pear
```

在上面的例子中，我们定义了一个string:string的字典，其中`[]`之间的是键类型，右边的是值类型。另外我们还看到了`range函数`，此函数一样神奇，可以用来迭代字典元素，返回key:value键值对。当然如果你对键或者值不感兴趣，一样可以使用`下划线(_)`来忽略返回值。

```
package main

import (
    "fmt"
)

func main() {
    var x map[string]string

    x = make(map[string]string)

    x["A"] = "Apple"
    x["B"] = "Banana"
    x["O"] = "Orange"
    x["P"] = "Pear"

    for key, val := range x {
        fmt.Println("Key:", key, "Value:", val)
    }
}
```

上面的方式就是使用了make函数来初始化字典，`试图为未经过初始化的字典添加元素会导致运行错误`，你可以把使用make函数初始化的那一行注释掉，然后看一下。

当然上面的例子中，我们可以把定义和初始化合成一句。

```
package main

import (
    "fmt"
)

func main() {
    x := make(map[string]string)

    x["A"] = "Apple"
    x["B"] = "Banana"
    x["O"] = "Orange"
    x["P"] = "Pear"

    for key, val := range x {
        fmt.Println("Key:", key, "Value:", val)
    }
}
```

现在我们再来看一下字典的数据访问方式。如果你访问的元素所对应的键存在于字典中，那么没有问题，如果不存在呢？

这个时候会返回零值。对于字符串零值就是""，对于整数零值就是0。但是对于下面的例子：

```
package main

import (
    "fmt"
)

func main() {
    x := make(map[string]int)

    x["A"] = 0
    x["B"] = 20
    x["O"] = 30
    x["P"] = 40

    fmt.Println(x["C"])
}
```

在这个例子中，很显然不存在键C，但是程序的输出结果为0，这样就和键A对应的值混淆了。

Go提供了一种方法来解决这个问题：

```
package main

import (
    "fmt"
)

func main() {
    x := make(map[string]int)

    x["A"] = 0
    x["B"] = 20
    x["O"] = 30
    x["P"] = 40

    if val, ok := x["C"]; ok {
        fmt.Println(val)
    }
}
```

上面的例子中，我们可以看到事实上使用 x["C"] 的返回值有两个，一个是值，另一个是是否存在此键的bool型变量，所以我们看到ok为true的时候就输出键C的值，如果ok为false，那就是字典中不存在这个键。

现在我们再来看看 Go提供的内置函数delete，这个函数可以用来从字典中删除元素 。

```
package main

import (
    "fmt"
)

func main() {
    x := make(map[string]int)

    x["A"] = 10
    x["B"] = 20
    x["C"] = 30
    x["D"] = 40

    fmt.Println("Before Delete")
    fmt.Println("Length:", len(x))
    fmt.Println(x)

    delete(x, "A")

    fmt.Println("After Delete")
    fmt.Println("Length:", len(x))
    fmt.Println(x)
}
```

输出结果为

```
Before Delete
Length: 4
map[A:10 B:20 C:30 D:40]
After Delete
Length: 3
map[B:20 C:30 D:40]
```

我们在删除元素前查看一下字典长度和元素，删除之后再看一下。这里面我们还可以看到`len函数也可以用来获取字典的元素个数`。当然如果你试图删除一个不存在的键，那么程序也不会报错，只是不会对字典造成任何影响。

最后我们再用一个稍微复杂的例子来结束字典的介绍。

我们有一个学生登记表，登记表里面有一组学号，每个学号对应一个学生，每个学生有名字和年龄。

```
package main

import (
    "fmt"
)

func main() {
    var facebook = make(map[string]map[string]int)
    facebook["0616020432"] = map[string]int{"Jemy": 25}
    facebook["0616020433"] = map[string]int{"Andy": 23}
    facebook["0616020434"] = map[string]int{"Bill": 22}

    for stu_no, stu_info := range facebook {
        fmt.Println("Student:", stu_no)
        for name, age := range stu_info {
            fmt.Println("Name:", name, "Age:", age)
        }
        fmt.Println()
    }
}
```

输出结果为

```
Student: 0616020432
Name Jemy Age 25

Student: 0616020433
Name Andy Age 23

Student: 0616020434
Name Bill Age 22
```

当然我们也可以用初始化的方式定义字典：

```
package main

import (
    "fmt"
)

func main() {
    var facebook = map[string]map[string]int{
        "0616020432": {"Jemy": 25},
        "0616020433": {"Andy": 23},
        "0616020434": {"Bill": 22},
    }

    for stu_no, stu_info := range facebook {
        fmt.Println("Student:", stu_no)
        for name, age := range stu_info {
            fmt.Println("Name:", name, "Age:", age)
        }
        fmt.Println()
    }
}
```

输出结果是一样的。

Go函数

Go函数

是时候讨论一下Go的函数定义了。

什么是函数

函数，简单来讲就是一段将 输入数据 转换为 输出数据 的 公用代码块 。当然有的时候函数的返回值为空，那么就是说输出数据为空。而真正的处理过程在函数内部已经完成了。

想一想我们为什么需要函数，最直接的需求就是代码中有太多的重复代码了，为了代码的可读性和可维护性，将这些重复代码重构为函数也是必要的。

函数定义

先看一个例子

```
package main

import (
    "fmt"
)

func slice_sum(arr []int) int {
    sum := 0
    for _, elem := range arr {
        sum += elem
    }
    return sum
}

func main() {
    var arr1 = []int{1, 3, 2, 3, 2}
    var arr2 = []int{3, 2, 3, 1, 6, 4, 8, 9}
    fmt.Println(slice_sum(arr1))
    fmt.Println(slice_sum(arr2))
}
```

在上面的例子中，我们需要分别计算两个切片的元素和。如果我们把计算切片元素的和的代码分别为两个切片展开，那么代码就失去了简洁性和一致性。假设你预想实现同样功能的代码在拷贝粘贴的过程中发生了错误，比如忘记改变量名之类的，到时候debug到崩溃吧。因为这时很有可能你就先入为主了，因为模板代码没有错啊，是不是。所以函数就是这个用处。

我们再仔细看一下上面的函数定义：

首先是关键字`func`，然后后面是`函数名称`，`参数列表`，最后是`返回值列表`。当然如果函数没有参数列表或者返回值，那么这两项都是可选的。其中返回值两边的括号在只声明一个返回值类型的时候可以省略。

命名返回值

Go的函数很有趣，你甚至可以为返回值预先定义一个名称，在函数结束的时候，直接一个return就可以返回所有的预定义返回值。例如上面的例子，我们将sum作为命名返回值。

```
package main

import (
    "fmt"
)

func slice_sum(arr []int) (sum int) {
    sum = 0
    for _, elem := range arr {
        sum += elem
    }
    return
}

func main() {
    var arr1 = []int{1, 3, 2, 3, 2}
    var arr2 = []int{3, 2, 3, 1, 6, 4, 8, 9}
    fmt.Println(slice_sum(arr1))
    fmt.Println(slice_sum(arr2))
}
```

这里要注意的是，如果你定义了命名返回值，那么在函数内部你将不能再重复定义一个同样名称的变量。比如第一个例子中我们用 `sum:=0` 来定义和初始化变量sum，而在第二个例子中，我们只能用 `sum=0` 初始化这个变量了。因为 `:=` 表示的是定义并且初始化变量。

实参数和虚参数

可能你听说过函数的实参数和虚参数。其实所谓的 实参数就是函数调用的时候传入的参数 。在上面的例子中，实参就是 arr1 和 arr2 ，而 虚参数就是函数定义的时候表示函数需要传入哪些参数的占位参数 。在上面的例子中，虚参就是 arr 。 实参和虚参的名字不必是一样的。即使是一样的，也互不影响。 因为虚参是函数的内部变量。而实参则是另一个函数的内部变量或者是全局变量。它们的作用域不同。如果一个函数的虚参碰巧和一个全局变量名称相同，那么函数使用的也是虚参。例如我们再修改一下上面的例子。

```
package main

import (
    "fmt"
)

var arr = []int{1, 3, 2, 3, 2}

func slice_sum(arr []int) (sum int) {
    sum = 0
    for _, elem := range arr {
        sum += elem
    }
    return
}

func main() {
    var arr2 = []int{3, 2, 3, 1, 6, 4, 8, 9}
    fmt.Println(slice_sum(arr))
    fmt.Println(slice_sum(arr2))
}
```

在上面的例子中，我们定义了全局变量arr并且初始化值，而我们的slice_sum函数的虚参也是arr，但是程序同样正常工作。

函数多返回值

记不记得你在java或者c里面需要返回多个值时还得去定义一个对象或者结构体的呢？在Go里面，你不需要这么做了。Go函数支持你返回多个值。

其实函数的多返回值，我们在上面遇见过很多次了。那就是`range`函数。这个函数用来迭代数组或者切片的时候返回的是两个值，一个是数组或切片元素的索引，另外一个是数组或切片元素。在上面的例子中，因为我们不需要元素的索引，所以我们用一个特殊的忽略返回值符号`下划线(_)`来忽略索引。

假设上面的例子我们除了返回切片的元素和，还想返回切片元素的平均值，那么我们修改一下代码。

```
package main

import (
    "fmt"
)

func slice_sum(arr []int) (int, float64) {
    sum := 0
    avg := 0.0
    for _, elem := range arr {
        sum += elem
    }
    avg = float64(sum) / float64(len(arr))
    return sum, avg
}

func main() {
    var arr1 = []int{3, 2, 3, 1, 6, 4, 8, 9}
    fmt.Println(slice_sum(arr1))
}
```

很简单吧，当然我们还可以将上面的参数定义为命名参数

```
package main

import (
    "fmt"
)

func slice_sum(arr []int) (sum int, avg float64) {
    sum = 0
    avg = 0.0
    for _, elem := range arr {
        sum += elem
    }
    avg = float64(sum) / float64(len(arr))
    //return sum, avg
    return
}

func main() {
    var arr1 = []int{3, 2, 3, 1, 6, 4, 8, 9}
    fmt.Println(slice_sum(arr1))
}
```

在上面的代码里面，将`return sum, avg`给注释了而直接使用`return`。其实这两种返回方式都可以。

变长参数

想一想我们的fmt包里面的Println函数，它怎么知道你传入的参数个数呢？

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(1)
    fmt.Println(1, 2)
    fmt.Println(1, 2, 3)
}
```

这个要归功于Go的一大特性，支持可变长参数列表。

首先我们来看一个例子

```
package main

import (
    "fmt"
)

func sum(arr ...int) int {
    sum := 0
    for _, val := range arr {
        sum += val
    }
    return sum
}
func main() {
    fmt.Println(sum(1))
    fmt.Println(sum(1, 2))
    fmt.Println(sum(1, 2, 3))
}
```

在上面的例子中，我们将原来的切片参数修改为可变长参数，然后使用range函数迭代这些参数，并求和。
从这里我们可以看出至少一点那就是 可变长参数列表里面的参数类型都是相同的 （*如果你对这句话表示怀疑，可能是因为你看到Println函数恰恰可以输出不同类型的可变参数，这个问题的答案要等到我们介绍完Go的接口后才行*）。

另外还有一点需要注意，那就是 可变长参数定义只能是函数的最后一个参数 。比如下面的例子：

```
package main

import (
    "fmt"
)

func sum(base int, arr ...int) int {
    sum := base
    for _, val := range arr {
        sum += val
    }
    return sum
}
func main() {
    fmt.Println(sum(100, 1))
    fmt.Println(sum(200, 1, 2))
    fmt.Println(sum(300, 1, 2, 3))
}
```

这里不知道你是否觉得这个例子其实和那个切片的例子很像啊，在哪里呢？

```
package main

import (
    "fmt"
)

func sum(base int, arr ...int) int {
    sum := base
    for _, val := range arr {
        sum += val
    }
    return sum
}
func main() {
    var arr1 = []int{1, 2, 3, 4, 5}
    fmt.Println(sum(300, arr1...))
}
```

呵呵，就是把切片“啪，啪，啪”三个耳光打碎了，传递过去啊！:-P

闭包函数

曾经使用python和javascript的时候就在想，如果有一天可以把这两种语言的特性做个并集该有多好。

这一天终于来了，Go支持闭包函数。

首先看一个闭包函数的例子。所谓闭包函数就是将整个函数的定义一气呵成写好并赋值给一个变量。然后用这个变量名作为函数名去调用函数体。

我们将刚刚的例子修改一下：

```
package main

import (
    "fmt"
)

func main() {
    var arr1 = []int{1, 2, 3, 4, 5}

    var sum = func(arr ...int) int {
        total_sum := 0
        for _, val := range arr {
            total_sum += val
        }
        return total_sum
    }
    fmt.Println(sum(arr1...))
}
```

从这里我们可以看出，其实闭包函数也没有什么特别之处。因为Go不支持在一个函数的内部再定义一个嵌套函数，所以使用闭包函数能够实现在一个函数内部定义另一个函数的目的。

这里我们需要注意的一个问题是，闭包函数对它外层的函数中的变量具有 `访问` 和 `修改` 的权限。例如：

```
package main

import (
    "fmt"
)

func main() {
    var arr1 = []int{1, 2, 3, 4, 5}
    var base = 300
    var sum = func(arr ...int) int {
        total_sum := 0
        total_sum += base
        for _, val := range arr {
            total_sum += val
        }
        return total_sum
    }
    fmt.Println(sum(arr1...))
}
```

这个例子，输出315，因为total_sum加上了base的值。

```
package main

import (
    "fmt"
)

func main() {
    var base = 0
    inc := func() {
        base += 1
    }
    fmt.Println(base)
    inc()
    fmt.Println(base)
}
```

在上面的例子中，闭包函数修改了main函数的局部变量base。

最后我们来看一个闭包的示例，生成偶数序列。

```
package main

import (
    "fmt"
)

func createEvenGenerator() func() uint {
    i := uint(0)
    return func() (retVal uint) {
        retVal = i
        i += 2
        return
    }
}
func main() {
    nextEven := createEvenGenerator()
    fmt.Println(nextEven())
    fmt.Println(nextEven())
    fmt.Println(nextEven())
}
```

这个例子很有意思的，因为我们定义了一个`返回函数定义`的函数。而所返回的函数定义就是`在这个函数的内部定义的闭包函数`。这个闭包函数在外层函数调用的时候，每次都生成一个新的偶数（加2操作）然后返回闭包函数定义。

其中`func() uint`就是函数createEvenGenerator的返回值。在createEvenGenerator中，这个返回值是return返回的闭包函数定义。

```
func() (retVal uint) {
        retVal = i
        i += 2
        return
    }
```

因为createEvenGenerator函数返回的是一个函数定义，所以我们再把它赋值给一个代表函数的变量，然后用这个代表闭包函数的变量去调用函数执行。

递归函数

每次谈到递归函数，必然绕不开阶乘和斐波拉切数列。

阶乘

```
package main

/**
    n!=1*2*3*...*n
*/
import (
    "fmt"
)

func factorial(x uint) uint {
    if x == 0 {
        return 1
    }
    return x * factorial(x-1)
}

func main() {
    fmt.Println(factorial(5))
}
```

如果x为0，那么返回1，因为0!=1。如果x是1，那么f(1)=1*f(0)，如果x是2，那么f(2)=2*f(1)=2*1*f(0)，依次推断f(x)=x*(x-1)*...*2*1*f(0)。

从上面看出所谓递归，就是在函数的内部重复调用一个函数的过程。需要注意的是这个函数必须能够一层一层分解，并且有出口。上面的例子出口就是0。

斐波拉切数列

求第N个斐波拉切元素

```
package main

/**
    f(1)=1
    f(2)=2
    f(n)=f(n-2)+f(n-1)
*/
import (
    "fmt"
)

func fibonacci(n int) int {
    var retVal = 0
    if n == 1 {
        retVal = 1
    } else if n == 2 {
        retVal = 2
    } else {
        retVal = fibonacci(n-2) + fibonacci(n-1)
    }
    return retVal

}
func main() {
    fmt.Println(fibonacci(5))
}
```

斐波拉切第一个元素是1，第二个元素是2，后面的元素依次是前两个元素的和。

其实对于递归函数来讲，只要知道了函数的出口，后面的不过是让计算机去不断地推断，一直推断到这个出口。理解了这一点，递归就很好理解了。

异常处理

当你读取文件失败而退出的时候是否担心文件句柄是否已经关闭？抑或是你对于try...catch...finally的结构中finally里面的代码和try里面的return代码那个先执行这样的问题痛苦不已？

一切都结束了。一门完美的语言必须有一个清晰的无歧义的执行逻辑。

好，来看看Go提供的异常处理。

defer

```
package main

import (
    "fmt"
)

func first() {
    fmt.Println("first func run")
}
func second() {
    fmt.Println("second func run")
}

func main() {
    defer second()
    first()
}
```

Go语言提供了关键字 `defer` 来在函数运行结束的时候运行一段代码或调用一个清理函数。上面的例子中，虽然second()函数写在first()函数前面，但是由于使用了defer标注，所以它是在main函数执行结束的时候才调用的。

所以输出结果

```
first func run
second func run
```

`defer` 用途最多的在于释放各种资源。比如我们读取一个文件，读完之后需要释放文件句柄。

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

func main() {
    fname := "D:\\Temp\\test.txt"
    f, err := os.Open(fname)
    defer f.Close()
    if err != nil {
        os.Exit(1)
    }
    bReader := bufio.NewReader(f)
    for {
        line, ok := bReader.ReadString('\n')
        if ok != nil {
            break
        }
        fmt.Println(strings.Trim(line, "\r\n"))
    }
}
```

在上面的例子中，我们按行读取文件，并且输出。从代码中，我们可以看到在使用os包中的Open方法打开文件后，立马跟着一个defer语句用来关闭文件句柄。这样就保证了该文件句柄在main函数运行结束的时候或者异常终止的时候一定能够被释放。而且由于紧跟着Open语句，一旦养成了习惯，就不会忘记去关闭文件句柄了。

panic & *recover*

> 当你周末走在林荫道上，听着小歌，哼着小曲，很是惬意。突然之间，从天而降瓢泼大雨，你顿时慌张（panic）起来，没有带伞啊，淋着雨感冒就不好了。于是你四下张望，忽然发现自己离地铁站很近，那里有很多卖伞的，心中顿时又安定了下来（recover），于是你飞奔过去买了一把伞（defer）。

好了，panic和recover是Go语言提供的用以处理异常的关键字。`panic用来触发异常`，而`recover用来终止异常并且返回传递给panic的值`。（注意`recover并不能处理异常`，而且`recover只能在defer里面使用，否则无效`。）

先瞧个小例子

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("I am walking and singing...")
    panic("It starts to rain cats and dogs")
    msg := recover()
    fmt.Println(msg)
}
```

看看输出结果

```
runtime.panic(0x48d380, 0xc084003210)
    C:/Users/ADMINI~1/AppData/Local/Temp/2/bindist667667715/go/src/pkg/runtime/panic.c:266
+0xc8
main.main()
    D:/JemyGraw/Creation/Go/freebook_go/func_d1.go:9 +0xea
exit status 2
```

咦？怎么没有输出recover获取的错误信息呢？

这是因为在运行到panic语句的时候，程序已经异常终止了，后面的代码就不运行了。

那么如何才能阻止程序异常终止呢？这个时候要使用defer。因为`defer一定是在函数执行结束的时候运行的。不管是正常结束还是异常终止`。

修改一下代码

```
package main

import (
    "fmt"
)

func main() {
    defer func() {
        msg := recover()
        fmt.Println(msg)
    }()
    fmt.Println("I am walking and singing...")
    panic("It starts to rain cats and dogs")
}
```

好了，看下输出

```
I am walking and singing...
It starts to rain cats and dogs
```

小结：

panic触发的异常通常是运行时错误。比如试图访问的索引超出了数组边界，忘记初始化字典或者任何无法轻易恢复到正常执行的错误。

Go指针

Go指针

不要害怕，Go的指针是好指针。

定义

所谓 指针其实你可以把它想像成一个箭头，这个箭头指向（存储）一个变量的地址 。

因为这个箭头本身也需要变量来存储，所以也叫做指针变量。

Go的指针 不支持那些乱七八糟的指针移位 。 它就表示一个变量的地址 。看看这个例子：

```
package main

import (
    "fmt"
)

func main() {
    var x int
    var x_ptr *int

    x = 10
    x_ptr = &x

    fmt.Println(x)
    fmt.Println(x_ptr)
    fmt.Println(*x_ptr)
}
```

上面例子输出 x的值 ， x的地址 和 通过指针变量输出x的值 ，而 x_ptr就是一个指针变量 。

```
10
0xc084000038
10
```

认真理清楚这两个符号的意思。

& 取一个变量的地址

***** 取一个指针变量所指向的地址的值

考你一下，上面的例子中，如何输出x_ptr的地址呢？

```
package main

import (
    "fmt"
)

func main() {
    var x int
    var x_ptr *int

    x = 10
    x_ptr = &x

    fmt.Println(&x_ptr)
}
```

此例看懂，指针就懂了。

永远记住一句话， 所谓指针就是一个指向（存储）特定变量地址的变量 。没有其他的特别之处。

再变态一下，看看这个：

```
package main

import (
    "fmt"
)

func main() {
    var x int
    var x_ptr *int

    x = 10
    x_ptr = &x

    fmt.Println(*&x_ptr)
}
```

1. x_ptr 是一个 指针变量 ，它 指向(存储)x的地址 ；
2. &x_ptr 是 取这个指针变量x_ptr的地址 ，这里可以设想 有另一个指针变量x_ptr_ptr(指向)存储 这个 x_ptr指针的地址 ；
3. *&x_ptr 等价于 *x_ptr_ptr 就是 取这个x_ptr_ptr指针变量 所 指向(存储) 的 地址所对应的变量的值

，也就是x_ptr的值 ，也就是 指针变量x_ptr指向(存储)的地址 ，也就是 x的地址 。 这里可以看到，其实 *& 这两个运算符在一起就相互抵消作用了。

用途

指针的一大用途就是可以将变量的指针作为实参传递给函数，从而在函数内部能够直接修改实参所指向的变

Go的变量传递都是值传递。

```
package main

import (
    "fmt"
)

func change(x int) {
    x = 200
}
func main() {
    var x int = 100
    fmt.Println(x)
    change(x)
    fmt.Println(x)
}
```

上面的例子输出结果为

```
100
100
```

很显然，change函数 改变的 仅仅是 内部变量x 的 值 ，而 不会改变 传递进去的 实参 。其实，也就是说Go的函数一般关心的是输出结果，而输入参数就相当于信使跑到函数门口大叫，你们这个参数是什么值，那个是什么值，然后就跑了。你函数根本就不能修改它的值。不过如果是传递的实参是指针变量，那么函数一看，小子这次你地址我都知道了，哪里跑。那么就是下面的例子：

```
package main

import (
    "fmt"
)

func change(x *int) {
    *x = 200
}
func main() {
    var x int = 100
    fmt.Println(x)
    change(&x)
    fmt.Println(x)
}
```

上面的例子中，change函数的虚参为 整型指针变量 ，所以在main中调用的时候 传递的是x的地址 。然后在change里面使用 *x=200 修改了这个x的地址的值。所以 x的值就变了 。这个输出是：

```
100
200
```

new

new这个函数挺神奇，因为它的用处太多了。这里还可以通过new来 初始化一个指针 。上面说过指针指向(存储)的是一个变量的地址，但是指针本身也需要地址存储。先看个例子：

```
package main

import (
    "fmt"
)

func set_value(x_ptr *int) {
    *x_ptr = 100
}
func main() {
    x_ptr := new(int)
    set_value(x_ptr)
    //x_ptr指向的地址
    fmt.Println(x_ptr)
    //x_ptr本身的地址
    fmt.Println(&x_ptr)
    //x_ptr指向的地址值
    fmt.Println(*x_ptr)
}
```

上面我们定义了一个x_ptr变量，然后用 new申请 了一个 存储整型数据的内存地址 ，然后将这个 地址赋值 给 x_ptr指针变量 ，也就是说 x_ptr指向（存储）的是一个可以存储整型数据的地址 ，然后用

set_value函数将 这个地址中存储的值 赋值为100。所以第一个输出是 x_ptr指向的地址 ，第二个则是 x_ptr本身的地址 ，而 *x_ptr 则是 x_ptr指向的地址中存储的整型数据的值 。

```
0xc084000040
0xc084000038
100
```

小结

好了，现在用个例子再来回顾一下指针。

交换两个变量的值。

```
package main

import (
    "fmt"
)

func swap(x, y *int) {
    *x, *y = *y, *x
}
func main() {
    x_val := 100
    y_val := 200
    swap(&x_val, &y_val)
    fmt.Println(x_val)
    fmt.Println(y_val)
}
```

很简单吧，这里利用了Go提供的 交叉赋值 的功能，另外由于是使用了指针作为参数，所以在swap函数内，x_val和y_val的值就被交换了。

Go结构体和指针

Go结构体和指针

基本上到这里的时候，就是上了一个台阶了。Go的精华特点即将展开。

结构体定义

上面我们说过Go的指针和C的不同，结构体也是一样的。Go是一门删繁就简的语言，一切令人困惑的特性都必须去掉。

简单来讲，Go提供的 结构体 就是把 使用各种数据类型定义 的 不同变量组合起来 的 高级数据类型 。闲

话不多说，看例子:

```
type Rect struct {
    width float64
    length float64
}
```

上面我们定义了一个矩形结构体，首先是关键是 type 表示要 定义一个新的数据类型了 ，然后是新的数据类型名称 Rect ，最后是 struct 关键字，表示这个高级数据类型是结构体类型。在上面的例子中，因为 width和length的数据类型相同 ，还可以写成如下格式：

```
type Rect struct {
    width, length float64
}
```

好了，来用结构体干点啥吧，计算一下矩形面积。

```
package main

import (
    "fmt"
)

type Rect struct {
    width, length float64
}

func main() {
    var rect Rect
    rect.width = 100
    rect.length = 200
    fmt.Println(rect.width * rect.length)
}
```

从上面的例子看到，其实结构体类型和基础数据类型使用方式差不多，唯一的区别就是结构体类型可以通过 . 来访问内部的成员。包括 给内部成员赋值 和 读取内部成员值 。

在上面的例子中，我们是用var关键字先定义了一个Rect变量，然后对它的成员赋值。我们也可以使用初始化的方式来给Rect变量的内部成员赋值。

```
package main

import (
    "fmt"
)

type Rect struct {
    width, length float64
}

func main() {
    var rect = Rect{width: 100, length: 200}

    fmt.Println(rect.width * rect.length)
}
```

当然 如果你知道结构体成员定义的顺序 ，也可以不使用 key:value 的方式赋值，
直接按照结构体成员定义的顺序给它们赋值 。

```
package main

import (
    "fmt"
)

type Rect struct {
    width, length float64
}

func main() {
    var rect = Rect{100, 200}

    fmt.Println("Width:", rect.width, "* Length:",
        rect.length, "= Area:", rect.width*rect.length)
}
```

输出结果为

```
Width: 100 * Length: 200 = Area: 20000
```

结构体参数传递方式

我们说过， Go函数的参数传递方式是值传递 ，这句话 对结构体也是适用的 。

```
package main

import (
    "fmt"
)

type Rect struct {
    width, length float64
}

func double_area(rect Rect) float64 {
    rect.width *= 2
    rect.length *= 2
    return rect.width * rect.length
}
func main() {
    var rect = Rect{100, 200}
    fmt.Println(double_area(rect))
    fmt.Println("Width:", rect.width, "Length:", rect.length)
}
```

上面的例子输出为:

```
80000
Width: 100 Length: 200
```

也就说虽然在double_area函数里面我们将结构体的宽度和长度都加倍，但仍然没有影响main函数里面的rect变量的宽度和长度。

结构体组合函数

上面我们在main函数中计算了矩形的面积，但是我们觉得矩形的面积如果能够作为矩形结构体的“内部函数”提供会更好。这样我们就可以直接说这个矩形面积是多少，而不用另外去取宽度和长度去计算。现在我们看看结构体“内部函数”定义方法：

```
package main

import (
    "fmt"
)

type Rect struct {
    width, length float64
}

func (rect Rect) area() float64 {
    return rect.width * rect.length
}

func main() {
    var rect = Rect{100, 200}

    fmt.Println("Width:", rect.width, "Length:", rect.length,
        "Area:", rect.area())
}
```

咦？这个是什么“内部方法”，根本没有定义在Rect数据类型的内部啊？

确实如此，我们看到，虽然main函数中的rect变量可以直接调用函数area()来获取矩形面积，但是area()函数确实没有定义在Rect结构体内部，这点和C语言的有很大不同。
Go使用组合函数的方式来为结构体定义结构体方法。我们仔细看一下上面的area()函数定义。

首先是关键字 func 表示这是一个函数，第二个参数是 结构体类型和实例变量 ，第三个是 函数名称 ，第四个是 函数返回值 。这里我们可以看出area()函数和普通函数定义的 区别就在于 area()函数 多了一个结构体类型限定 。这样一来Go就知道了这是一个为结构体定义的 方法 。

这里需要注意一点就是 定义在结构体上面的函数(function) 一般叫做 方法(method) 。

结构体和指针

我们在指针一节讲到过， 指针的主要作用就是在函数内部改变传递进来变量的值 。对于上面的计算矩形面积的例子，我们可以修改一下代码如下：

```
package main

import (
    "fmt"
)

type Rect struct {
    width, length float64
}

func (rect *Rect) area() float64 {
    return rect.width * rect.length
}

func main() {
    var rect = new(Rect)
    rect.width = 100
    rect.length = 200
    fmt.Println("Width:", rect.width, "Length:", rect.length,
        "Area:", rect.area())
}
```

上面的例子中，使用了new函数来创建一个结构体指针rect，也就是说rect的类型是*Rect，结构体遇到指针的时候，你`不需要使用*去访问结构体的成员`，直接使用`.`引用就可以了。所以上面的例子中我们直接使用`rect.width=100`和`rect.length=200`来设置结构体成员值。因为这个时候rect是结构体指针，所以我们定义area()函数的时候结构体限定类型为`*Rect`。

其实在计算面积的这个例子中，我们不需要改变矩形的宽或者长度，所以定义area函数的时候结构体限定类型仍然为`Rect`也是可以的。如下：

```
package main

import (
    "fmt"
)

type Rect struct {
    width, length float64
}

func (rect Rect) area() float64 {
    return rect.width * rect.length
}

func main() {
    var rect = new(Rect)
    rect.width = 100
    rect.length = 200
    fmt.Println("Width:", rect.width, "Length:", rect.length,
        "Area:", rect.area())
}
```

这里Go足够聪明，所以rect.area()也是可以的。

至于 使不使用结构体指针和使不使用指针的出发点是一样的 ，那就是 你是否试图在函数内部改变传递进来的参数的值 。再举个例子如下：

```
package main

import (
    "fmt"
)

type Rect struct {
    width, length float64
}

func (rect *Rect) double_area() float64 {
    rect.width *= 2
    rect.length *= 2
    return rect.width * rect.length
}

func main() {
    var rect = new(Rect)
    rect.width = 100
    rect.length = 200
    fmt.Println(*rect)
    fmt.Println("Double Width:", rect.width, "Double Length:", rect.length,
        "Double Area:", rect.double_area())
    fmt.Println(*rect)
}
```

这个例子的输出是：

```
{100 200}
Double Width: 200 Double Length: 400 Double Area: 80000
{200 400}
```

结构体内嵌类型

我们可以在一个`结构体内部定义另外一个结构体类型的成员`。例如iPhone也是Phone，我们看下例子：

```
package main

import (
    "fmt"
)

type Phone struct {
    price int
    color string
}

type IPhone struct {
    phone Phone
    model string
}

func main() {
    var p IPhone
    p.phone.price = 5000
    p.phone.color = "Black"
    p.model = "iPhone 5"
    fmt.Println("I have a iPhone:")
    fmt.Println("Price:", p.phone.price)
    fmt.Println("Color:", p.phone.color)
    fmt.Println("Model:", p.model)
}
```

输出结果为

```
I have a iPhone:
Price: 5000
Color: Black
Model: iPhone 5
```

在上面的例子中，我们在结构体IPhone里面定义了一个Phone变量phone，然后我们可以像正常的访问结构体成员一样访问phone的成员数据。但是我们原来的意思是 “iPhone也是(is-a)Phone” ，而这里的结构体IPhone里面定义了一个phone变量，给人的感觉就是 “iPhone有一个(has-a)Phone” ，挺奇怪的。当然Go也知道这种方式很奇怪，所以支持如下做法：

```
package main

import (
    "fmt"
)

type Phone struct {
    price int
    color string
}

type IPhone struct {
    Phone
    model string
}

func main() {
    var p IPhone
    p.price = 5000
    p.color = "Black"
    p.model = "iPhone 5"
    fmt.Println("I have a iPhone:")
    fmt.Println("Price:", p.price)
    fmt.Println("Color:", p.color)
    fmt.Println("Model:", p.model)
}
```

输出结果为

```
I have a iPhone:
Price: 5000
Color: Black
Model: iPhone 5
```

在这个例子中，我们定义IPhone结构体的时候，`不再定义Phone变量`，`直接把结构体Phone类型定义在那里`。然后IPhone就可以`像访问直接定义在自己结构体里面的成员一样访问Phone的成员`。

上面的例子中，我们演示了结构体的内嵌类型以及内嵌类型的成员访问，除此之外，假设结构体A内部定义了一个内嵌结构体B，那么A同时也可以调用所有定义在B上面的函数。

```
package main

import (
    "fmt"
)

type Phone struct {
    price int
    color string
}

func (phone Phone) ringing() {
    fmt.Println("Phone is ringing...")
}

type IPhone struct {
    Phone
    model string
}

func main() {
    var p IPhone
    p.price = 5000
    p.color = "Black"
    p.model = "iPhone 5"
    fmt.Println("I have a iPhone:")
    fmt.Println("Price:", p.price)
    fmt.Println("Color:", p.color)
    fmt.Println("Model:", p.model)

    p.ringing()
}
```

输出结果为：

```
I have a iPhone:
Price: 5000
Color: Black
Model: iPhone 5
Phone is ringing...
```

接口

我们先看一个例子，关于Nokia手机和iPhone手机都能够打电话的例子。

```
package main

import (
    "fmt"
)

type NokiaPhone struct {
}

func (nokiaPhone NokiaPhone) call() {
    fmt.Println("I am Nokia, I can call you!")
}

type IPhone struct {
}

func (iPhone IPhone) call() {
    fmt.Println("I am iPhone, I can call you!")
}
func main() {
    var nokia NokiaPhone
    nokia.call()

    var iPhone IPhone
    iPhone.call()
}
```

我们定义了NokiaPhone和IPhone，它们都有各自的方法call()，表示自己都能够打电话。但是我们想一想，是手机都应该能够打电话，所以这个不算是NokiaPhone或是IPhone的独特特点。否则iPhone不可能卖这么贵了。

再仔细看一下`接口的定义`，首先是关键字`type`，然后是`接口名称`，最后是关键字`interface`表示这个类型是接口类型。`在接口类型里面，我们定义了一组方法`。

Go语言提供了一种接口功能，它把所有的具有共性的方法定义在一起，
`任何其他类型只要实现了这些方法就是实现了这个接口`，`不一定非要显式地声明 要去实现哪些接口啦`。比如上面的手机的call()方法，就完全可以定义在接口Phone里面，而NokiaPhone和IPhone只要实现了这个接口就是一个Phone。

```
package main

import (
    "fmt"
)

type Phone interface {
    call()
}

type NokiaPhone struct {
}

func (nokiaPhone NokiaPhone) call() {
    fmt.Println("I am Nokia, I can call you!")
}

type IPhone struct {
}

func (iPhone IPhone) call() {
    fmt.Println("I am iPhone, I can call you!")
}

func main() {
    var phone Phone

    phone = new(NokiaPhone)
    phone.call()

    phone = new(IPhone)
    phone.call()

}
```

在上面的例子中，我们定义了一个接口Phone，接口里面有一个方法call()，仅此而已。然后我们在main函数里面定义了一个Phone类型变量，并分别为之赋值为NokiaPhone和IPhone。然后调用call()方法，输出结果如下：

```
I am Nokia, I can call you!
I am iPhone, I can call you!
```

以前我们说过，`Go语言式静态类型语言，变量的类型在运行过程中不能改变`。但是在上面的例子中，phone变量好像先定义为Phone类型，然后是NokiaPhone类型，最后成为了IPhone类型，真的是这样吗？

原来，在Go语言里面，`一个类型A只要实现了接口X所定义的全部方法`，那么`A类型的变量`也是`X类型的变量`。在上面的例子中，NokiaPhone和IPhone都实现了Phone接口的call()方法，所以它们都是Phone，这样一来是不是感觉正常了一些。

我们为Phone添加一个方法sales()，再来熟悉一下接口用法。

```
package main

import (
    "fmt"
)

type Phone interface {
    call()
    sales() int
}

type NokiaPhone struct {
    price int
}

func (nokiaPhone NokiaPhone) call() {
    fmt.Println("I am Nokia, I can call you!")
}
func (nokiaPhone NokiaPhone) sales() int {
    return nokiaPhone.price
}

type IPhone struct {
    price int
}

func (iPhone IPhone) call() {
    fmt.Println("I am iPhone, I can call you!")
}

func (iPhone IPhone) sales() int {
    return iPhone.price
}

func main() {
    var phones = [5]Phone{
        NokiaPhone{price: 350},
        IPhone{price: 5000},
        IPhone{price: 3400},
        NokiaPhone{price: 450},
        IPhone{price: 5000},
    }

    var totalSales = 0
    for _, phone := range phones {
        totalSales += phone.sales()
    }
    fmt.Println(totalSales)

}
```

输出结果：

```
14200
```

上面的例子中，我们定义了一个手机数组，然后计算手机的总售价。可以看到，由于NokiaPhone和IPhone都实现了sales()方法，所以它们都是Phone类型，但是计算售价的时候，Go会知道调用哪个对象实现的方法。

接口类型还可以作为结构体的数据成员。

假设有个败家子，iPhone没有出的时候，买了好几款Nokia，iPhone出来后，又买了好多部iPhone，老爸要来看看这小子一共花了多少钱。

```
package main

import (
    "fmt"
)

type Phone interface {
    sales() int
}

type NokiaPhone struct {
    price int
}

func (nokiaPhone NokiaPhone) sales() int {
    return nokiaPhone.price
}

type IPhone struct {
    price int
}

func (iPhone IPhone) sales() int {
    return iPhone.price
}

type Person struct {
    phones []Phone
    name   string
    age    int
}

func (person Person) total_cost() int {
    var sum = 0
    for _, phone := range person.phones {
        sum += phone.sales()
    }
    return sum
```

```
}

func main() {
    var bought_phones = [5]Phone{
        NokiaPhone{price: 350},
        IPhone{price: 5000},
        IPhone{price: 3400},
        NokiaPhone{price: 450},
        IPhone{price: 5000},
    }

    var person = Person{name: "Jemy", age: 25, phones: bought_phones[:]}

    fmt.Println(person.name)
    fmt.Println(person.age)
    fmt.Println(person.total_cost())
}
```

这个例子纯为演示接口作为结构体数据成员，如有雷同，纯属巧合。这里面我们定义了一个Person结构体，结构体内部定义了一个手机类型切片。另外我们定义了Person的total_cost()方法用来计算手机花费总额。输出结果如下：

```
Jemy
25
14200
```

小结

Go的结构体和接口的实现方法可谓删繁就简，去除了很多别的语言令人困惑的地方，而且学习难度也不大，很容易上手。不过由于思想比较独到，也有可能会有人觉得功能太简单而无用，这个就各有看法了，不过在逐渐的使用过程中，我们会慢慢领悟到这种设计所带来的好处，以及所避免的问题。

Go并行计算

Go并行计算

如果说Go有什么让人一见钟情的特性，那大概就是并行计算了吧。

做个题目

> 如果我们列出10以下所有能够被3或者5整除的自然数，那么我们得到的是3，5，6和9。这四个数的和是23。
> 那么请计算1000以下（不包括1000）的所有能够被3或者5整除的自然数的和。

这个题目的一个思路就是：

(1) 先计算1000以下所有能够被3整除的整数的和A，
(2) 然后计算1000以下所有能够被5整除的整数和B，
(3) 然后再计算1000以下所有能够被3和5整除的整数和C,
(4) 使用A+B-C就得到了最后的结果。

按照上面的方法，传统的方法当然就是一步一步计算，然后再到第(4)步汇总了。

但是一旦有了Go，我们就可以让前面三个步骤并行计算，然后再在第(4)步汇总。

并行计算涉及到一个新的 数据类型chan 和一个新的 关键字go 。

先看例子：

```
package main

import (
    "fmt"
    "time"
)

func get_sum_of_divisible(num int, divider int, resultChan chan int) {
    sum := 0
    for value := 0; value < num; value++ {
        if value%divider == 0 {
            sum += value
        }
    }
    resultChan <- sum
}

func main() {
    LIMIT := 1000
    resultChan := make(chan int, 3)
    t_start := time.Now()
    go get_sum_of_divisible(LIMIT, 3, resultChan)
    go get_sum_of_divisible(LIMIT, 5, resultChan)
    go get_sum_of_divisible(LIMIT, 15, resultChan)

    sum3, sum5, sum15 := <-resultChan, <-resultChan, <-resultChan
    sum := sum3 + sum5 - sum15
    t_end := time.Now()
    fmt.Println(sum)
    fmt.Println(t_end.Sub(t_start))
}
```

(1) 在上面的例子中，我们首先定义了一个普通的函数get_sum_of_divisible，这个函数的 最后一个参数是一个整型chan类型 ，这种类型，你可以把它当作一个先进先出的队列。你可以 向它写入数据 ，也可以 从它读出数据 。它 所能接受的数据类型 就是 由chan关键字后面的类型所决定

的。在上面的例子中，我们使用 <- 运算符将函数计算的结果写入channel。channel是go提供的用来协程之间通信的方式。本例中main是一个协程，三个get_sum_of_divisible调用是协程。要在这四个协程间通信，必须有一种可靠的手段。

(2) 在main函数中，我们使用go关键字来开启并行计算。并行计算是由goroutine来支持的， goroutine 又叫做 协程 ，你可以把它看作为比线程更轻量级的运算。开启一个协程很简单，就是 go关键字 后面 跟上所要运行的函数 。

(3) 最后，我们要从channel中取出并行计算的结果。使用 <- 运算符从channel里面取出数据。

在本例中，我们为了演示go并行计算的速度，还引进了time包来计算程序执行时间。在同普通的顺序计算相比，并行计算的速度是非同凡响的。

好了，上面的例子看完，我们来详细讲解Go的并行计算。

Goroutine协程

所谓协程，就是Go提供的轻量级的独立运算过程，比线程还轻。创建一个协程很简单，就是go关键字加上所要运行的函数。看个例子：

```
package main

import (
    "fmt"
)

func list_elem(n int) {
    for i := 0; i < n; i++ {
        fmt.Println(i)
    }
}
func main() {
    go list_elem(10)
}
```

上面的例子是创建一个协程遍历一下元素。但是当你运行的时候，你会 发现什么都没有输出 ！
为什么呢？
因为上面的 main函数 在 创建完协程后 就 立刻退出 了，所以 协程 还 没有来得及运行 呢！修改一下：

```
package main

import (
    "fmt"
)

func list_elem(n int) {
    for i := 0; i < n; i++ {
        fmt.Println(i)
    }
}
func main() {
    go list_elem(10)
    var input string
    fmt.Scanln(&input)
}
```

这里，我们在main函数创建协程后，要求用户输入任何数据后才退出，这样协程就有了运行的时间，故而输出结果：

```
0
1
2
3
4
5
6
7
8
9
```

其实在开头的例子里面，我们的main函数事实上也被阻塞了，因为 `sum3, sum5, sum15 := <-resultChan, <-resultChan, <-resultChan` 这行代码在channel里面没有数据或者数据个数不符的时候，都会阻塞在那里，直到协程结束，写入结果。

不过既然是并行计算，我们还是得看看协程是否真的并行计算了。

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func list_elem(n int, tag string) {
    for i := 0; i < n; i++ {
        fmt.Println(tag, i)
        tick := time.Duration(rand.Intn(100))
        time.Sleep(time.Millisecond * tick)
    }
}
func main() {
    go list_elem(10, "go_a")
    go list_elem(20, "go_b")
    var input string
    fmt.Scanln(&input)
}
```

输出结果

```
go_a 0
go_b 0
go_a 1
go_b 1
go_a 2
go_b 2
go_b 3
go_b 4
go_a 3
go_b 5
go_b 6
go_a 4
go_a 5
go_b 7
go_a 6
go_a 7
go_b 8
go_b 9
go_a 8
go_b 10
go_b 11
go_a 9
go_b 12
go_b 13
go_b 14
go_b 15
go_b 16
go_b 17
go_b 18
go_b 19
```

在上面的例子中，我们让两个协程在每输出一个数字的时候，随机Sleep了一会儿。如果是并行计算，那么输出是无序的。从上面的例子中，我们可以看出两个协程确实并行运行了。

Channel通道

Channel提供了 `协程之间` 的 `通信方式` 以及 `运行同步机制` 。

> 假设训练定点投篮和三分投篮，教练在计数。

```
package main

import (
    "fmt"
    "time"
)

func fixed_shooting(msg_chan chan string) {
    for {
        msg_chan <- "fixed shooting"
        fmt.Println("continue fixed shooting...")
    }
}

func count(msg_chan chan string) {
    for {
        msg := <-msg_chan
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}

func main() {
    var c chan string
    c = make(chan string)

    go fixed_shooting(c)
    go count(c)

    var input string
    fmt.Scanln(&input)
}
```

输出结果为：

```
fixed shooting
continue fixed shooting...
fixed shooting
continue fixed shooting...
fixed shooting
continue fixed shooting...
```

我们看到在fixed_shooting函数里面我们将消息传递到channel，然后输出提示信息"continue fixed shooting..."，而在count函数里面，我们从channel里面取出消息输出，然后间隔1秒再去取消息输出。这里面我们可以考虑一下，如果我们不去从channel中取消息会出现什么情况？我们把main函数里面的 `go count(c)` 注释掉，然后再运行一下。发现程序再也不会输出消息和提示信息了。这是因为channel中根本就没有信息了，因为 `如果你要向channel里面写信息` ， `必须有配对的取信息的一端` ，否则是不会写的。

我们再把三分投篮加上。

```
package main

import (
    "fmt"
    "time"
)

func fixed_shooting(msg_chan chan string) {
    for {
        msg_chan <- "fixed shooting"
    }
}

func three_point_shooting(msg_chan chan string) {
    for {
        msg_chan <- "three point shooting"
    }
}

func count(msg_chan chan string) {
    for {
        msg := <-msg_chan
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}

func main() {
    var c chan string
    c = make(chan string)

    go fixed_shooting(c)
    go three_point_shooting(c)
    go count(c)

    var input string
    fmt.Scanln(&input)
}
```

输出结果为：

```
fixed shooting
three point shooting
fixed shooting
three point shooting
fixed shooting
three point shooting
```

我们看到程序交替输出定点投篮和三分投篮，这是因为写入channel的信息必须要读取出来，否则尝试再

次写入就失败了。

在上面的例子中，我们发现 定义一个channel信息变量 的方式就是多加一个 chan 关键字。并且你能够 向channel写入数据 和 从channel读取数据 。这里我们还可以设置channel通道的方向。

Channel通道方向*

所谓的 通道方向 就是 写 和 读 。如果我们如下定义

```
c chan<- string //那么你只能向channel写入数据
```

而这种定义

```
c <-chan string //那么你只能从channel读取数据
```

试图向只读chan变量写入数据或者试图从只写chan变量读取数据都会导致编译错误。

如果是默认的定义方式

```
c chan string //那么你既可以向channel写入数据也可以从channnel读取数据
```

多通道(Select)

如果上面的投篮训练现在有两个教练了，各自负责一个训练项目。而且还在不同的篮球场，这个时候很显然，我们一个channel就不够用了。修改一下：

```
package main

import (
    "fmt"
    "time"
)

func fixed_shooting(msg_chan chan string) {
    for {
        msg_chan <- "fixed shooting"
        time.Sleep(time.Second * 1)
    }
}

func three_point_shooting(msg_chan chan string) {
    for {
        msg_chan <- "three point shooting"
        time.Sleep(time.Second * 1)
    }
}

func main() {
    c_fixed := make(chan string)
    c_3_point := make(chan string)

    go fixed_shooting(c_fixed)
    go three_point_shooting(c_3_point)

    go func() {
        for {
            select {
            case msg1 := <-c_fixed:
                fmt.Println(msg1)
            case msg2 := <-c_3_point:
                fmt.Println(msg2)
            }
        }

    }()

    var input string
    fmt.Scanln(&input)
}
```

其他的和上面的一样，唯一不同的是我们将定点投篮和三分投篮的消息写入了不同的channel，那么main函数如何知道从哪个channel读取消息呢？使用select方法，select方法依次检查每个channel是否有消息传递过来，如果有就取出来输出。如果同时有多个消息到达，那么select闭上眼睛随机选一个channel来从中读取消息，如果没有一个channel有消息到达，那么select语句就阻塞在这里一直等待。

在某些情况下，比如学生投篮中受伤了，那么就轮到医护人员上场了，教练在一般看看，如果是重伤，教练就不等了，就回去了休息了，待会儿再过来看看情况。我们可以给select加上一个case用来判断是否等

待各个消息到达超时。

```
package main

import (
    "fmt"
    "time"
)

func fixed_shooting(msg_chan chan string) {
    var times = 3
    var t = 1
    for {
        if t <= times {
            msg_chan <- "fixed shooting"
        }
        t++
        time.Sleep(time.Second * 1)
    }
}

func three_point_shooting(msg_chan chan string) {
    var times = 5
    var t = 1
    for {
        if t <= times {
            msg_chan <- "three point shooting"
        }
        t++
        time.Sleep(time.Second * 1)
    }
}

func main() {
    c_fixed := make(chan string)
    c_3_point := make(chan string)

    go fixed_shooting(c_fixed)
    go three_point_shooting(c_3_point)

    go func() {
        for {
            select {
            case msg1 := <-c_fixed:
                fmt.Println(msg1)
            case msg2 := <-c_3_point:
                fmt.Println(msg2)
            case <-time.After(time.Second * 5):
                fmt.Println("timeout, check again...")
            }
        }

    }()
```

```
    var input string
    fmt.Scanln(&input)
}
```

在上面的例子中，我们让投篮的人在几次过后挂掉，然后教练就每次等5秒出来看看情况（累死丫的，:-P），因为我们对等待的时间不感兴趣就不用变量存储了，直接 <-time.After(time.Second*5) ，或许你会奇怪，为什么各个channel消息都没有到达，select为什么不阻塞？就是因为这个time.After，虽然它没有显式地告诉你这是一个channel消息，但是记得么？main函数也是一个channel啊！哈哈！至于time.After的功能实际上让main阻塞了5秒后返回给main的channel一个时间。所以我们在case里面把这个时间消息读出来，select就不阻塞了。

输出结果如下：

```
fixed shooting
three point shooting
fixed shooting
three point shooting
fixed shooting
three point shooting
three point shooting
three point shooting
timeout, check again...
timeout, check again...
timeout, check again...
timeout, check again...
```

这里select还有一个 default的选项 ，如果你指定了default选项，那么当select发现 没有消息到达 的时候 也不会阻塞 ，直接开始转回去再次判断。

Channel Buffer通道缓冲区

我们定义chan变量的时候，还可以指定它的 缓冲区大小 。一般我们 定义的channel都是同步的 ，也就是说接受端和发送端彼此等待对方ok才开始。但是如果你给一个channel 指定了一个缓冲区 ，那么 消息的发送和接受式异步的 ， 除非channel缓冲区已经满了 。

```
c:=make(chan int, 1)
```

我们看个例子：

```
package main

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

func shooting(msg_chan chan string) {
    var group = 1
    for {
        for i := 1; i <= 10; i++ {
            msg_chan <- strconv.Itoa(group) + ":" + strconv.Itoa(i)
        }
        group++
        time.Sleep(time.Second * 10)
    }
}

func count(msg_chan chan string) {
    for {
        fmt.Println(<-msg_chan)
    }
}

func main() {
    var c = make(chan string, 20)
    go shooting(c)
    go count(c)

    var input string
    fmt.Scanln(&input)
}
```

输出结果为：

```
1:1
1:2
1:3
1:4
1:5
1:6
1:7
1:8
1:9
1:10
2:1
2:2
2:3
2:4
2:5
2:6
2:7
2:8
2:9
2:10
3:1
3:2
3:3
3:4
3:5
3:6
3:7
3:8
3:9
3:10
4:1
4:2
4:3
4:4
4:5
4:6
4:7
4:8
4:9
4:10
```

你可以尝试运行一下，每次都是一下子输出10个数据。然后等待10秒再输出一批。

小结

并行计算这种特点最适合用来开发网站服务器，因为一般网站服务都是高并发的，逻辑十分复杂。而使用Go的这种特性恰是提供了一种极好的方法。