

## 1. 前言

1.1 为什么要学习go语言?

1.2 go语言能做什么?

1.3 go语言应该如何学?

## 2. 环境搭建

2.1 三大平台安装方法介绍

2.2 IDE开发工具安装

2.3 go工具集

## 3 基础语法

3.1 初识Golang

3.2 数据类型

3.3 变量

3.4 指针

3.5 常量

3.6 条件分支

3.7 for循环

3.8 函数

3.9 容器编程

3.9.1 数组

3.9.2 切片

3.9.3 map键值对

## 4 核心编程

4.1 面向对象

4.1.1 结构体

4.1.2 封装实现

4.1.3 Go语言中的继承

4.1.4 Go语言的多态

4.2 json处理

4.3 map与json

4.4 错误处理

4.4.1 error处理

4.4.2 panic与recover

4.5 并发与同步的概念

4.6 并发编程

4.7 Go语言运行时

4.7.1 GOMAXPROCS

4.8 Go语言同步

4.9 定时器

4.10 多路channel监控

## 5 网络编程

5.1 文件IO

5.2 TCP编程

5.3 HTTP编程

## 6 Go语言工程管理

6.1 Go语言的目录

6.2 源码包的自定义和使用

6.2.1 init函数

# 1. 前言

---

## 1.1 为什么要学习go语言？

Go（又称Golang）是Google开发的一种静态强类型、编译型、并发型，并具有垃圾回收功能的编程语言。创作者包括罗伯特·格瑞史莫（Robert Griesemer），罗勃·派克（Rob Pike）及肯·汤普逊（Ken Thompson）、Ian Lance Taylor、Russ Cox。这些在google都是响当当的角色，如果对unix历史有所了解的小伙伴就会知道，Thompson和Dennis.Ritchie被合称为unix之父以及C语言之父，由此可见，go语言创造团队相当强大！学习Go语言的理由可以有以下几点：

- 并发
- 语法简洁
- 开发周期短
- 内存回收
- 运行效率高
- 统一的编程规范
- 活跃的社区文化

## 1.2 go语言能做什么？

Go语言利用其天然的并发特性，可以很好的应用于服务器后端开发，目前在云计算领域以及区块链行业，Go语言都受到了热烈追捧，顶顶大名的Docker和k8s都是使用Go语言编写的，Go语言的产业链正在日趋成熟。

## 1.3 go语言应该如何学？

Go语言的入门非常简单，学习的路径可以是先搭建环境，接下来学习基础语法，此后学习Go语言核心编程和思想，再然后进入项目阶段实战，掌握行业内常用的开源框架。整体而言，Go语言的学习成本并不高！

# 2. 环境搭建

---

开发环境始终是学习一门语言的开始，Go语言作为2009年才被推出的新面孔来说，它的运行特点是跨平台，因此我们可以在各种主机上安装Go。Go语言的安装很简单，在Go语言的官网可以下载到安装包，但是由于某些（众所周知的）原因，国内访问会有一些问题，但是也有一些好心人将Go语言的安装包同步到了[Go语言中文网](#)，在这里可以很方便的下载。如下图所示：

## 推荐下载

### 源码

[go1.12.7.src.tar.gz](#) (21MB)

### Apple macOS

macOS 10.8 or later, Intel 64-bit 处理器

[go1.12.7.darwin-amd64.pkg](#) (121MB)

### Linux

Linux 2.6.23 or later, Intel 64-bit 处理器

[go1.12.7.linux-amd64.tar.gz](#) (122MB)

### Microsoft Windows

Windows XP SP2 or later, Intel 64-bit 处理器

[go1.12.7.windows-amd64.msi](#)

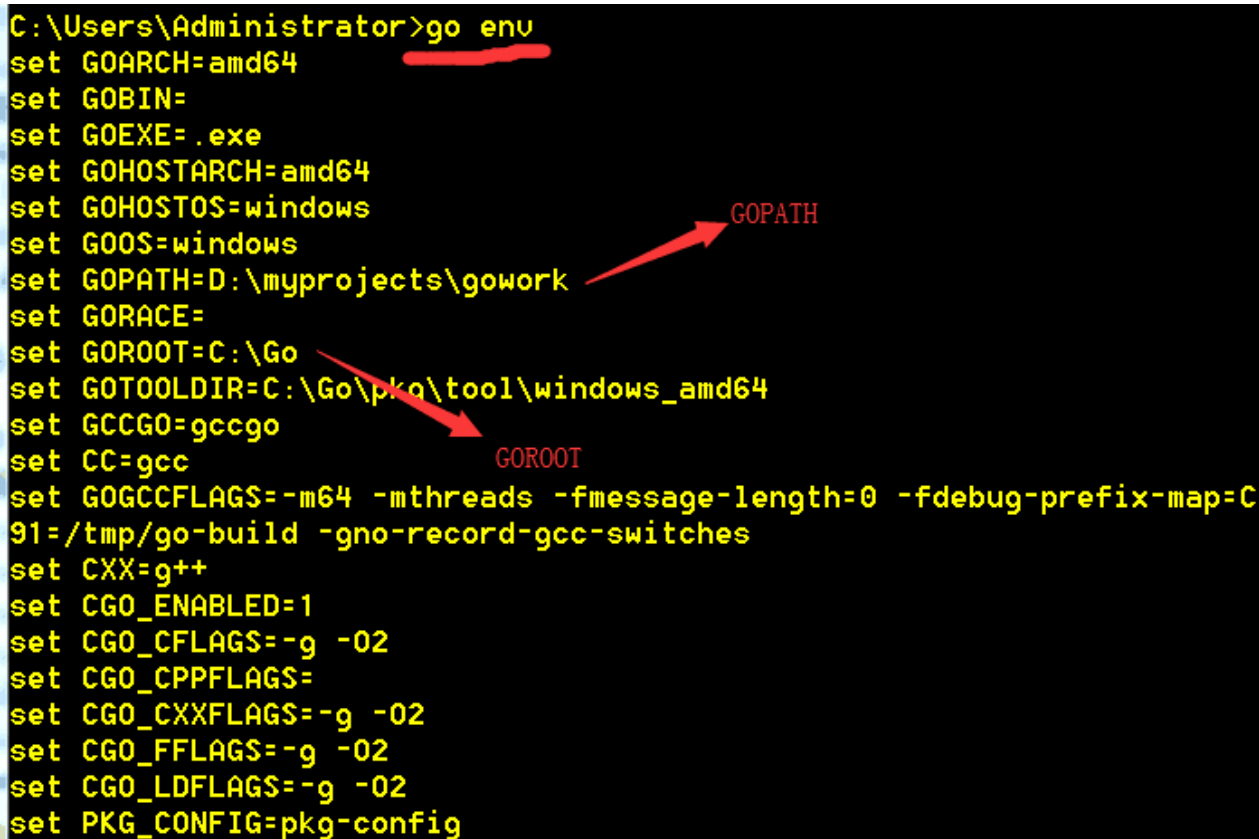
(118MB)

## 2.1 三大平台安装方法介绍

- windows

对于windows的用户来说，下载windows版的msi文件即可，安装的过程也非常的傻瓜，各种下一步即可。默认情况下Go语言很关键的环境变量GOROOT和GOPATH在安装的时候会设定好，如果没有，手动设置一下这两个环境变量即可。GOROOT就是安装路径，而GOPATH则是后续要用到的重要路径，直接认为它是项目开发路径也可以。在它的下面，会有bin、pkg、src三个子目录，bin是存放二进制文件的目录，pkg是存放包编译后的.a文件，src就是我们的源码路径。（GOROOT与GOPATH的作用在所有平台是相同的）在命令行窗口，可以通过go env指令来查看Go语言的环境变量情况。

```
C:\Users\Administrator>go env
set GOARCH=amd64
set GOBIN=
set GOEXE=.exe
set GOHOSTARCH=amd64
set GOHOSTOS=windows
set GOOS=windows
set GOPATH=D:\myprojects\gowork
set GORACE=
set GOROOT=C:\Go
set GOTOOOLDIR=C:\Go\pkg\tool\windows_amd64
set GCCGO=gccgo
set CC=gcc
set GOGCCFLAGS=-m64 -mthreads -fmessage-length=0 -fdebug-prefix-map=C
91=/tmp/go-build -gno-record-gcc-switches
set CXX=g++
set CGO_ENABLED=1
set CGO_CFLAGS=-g -O2
set CGO_CPPFLAGS=
set CGO_CXXFLAGS=-g -O2
set CGO_FFLAGS=-g -O2
set CGO_LDFLAGS=-g -O2
set PKG_CONFIG=pkg-config
```



- Linux

在linux平台可以使用命令行来安装Go语言开发环境。以ubuntu系统举例（centOS需要使用yum进行安装），下面两句指令就可以搞定Go语言的安装。

```
sudo apt-get update
sudo apt-get install golang
```

接下来我们介绍一下下载安装包的方式，个人推荐此种方式。首先，先去下载安装包（amd64代表该版本支持64位操作系统，amd代表的则是cpu的架构，与之对应的是x86），下面使用的安装包地址是从Go语言中文网得到的下载链接。

```
mkdir ~/install
cd ~/install
wget https://studygolang.com/dl/golang/go1.12.7.linux-amd64.tar.gz
```

将压缩包解压

```
tar -zxvf go1.12.7.linux-amd64.tar.gz -C ~/
```

设置环境变量

```
cd ~
export PATH=$PATH:~/go/bin
echo 'export PATH=$PATH:~/go/bin' >> ~/.bashrc
mkdir ~/gework
export GOPATH=$HOME/gework
echo 'export GOPATH=$HOME/gework' >> ~/.bashrc
```

来查看一下环境变量情况

```
ubuntu@ip-172-31-26-216:~$ go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/home/ubuntu/.cache/go-build"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/ubuntu/gework"
GOPROXY=""
GORACE=""
GOROOT="/home/ubuntu/go"
GOTMPDIR=""
GOTOOLDIR="/home/ubuntu/go/pkg/tool/linux_amd64"
GCCGO="gccgo"
CC="gcc"
CXX="g++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
```

```
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/go-build052037651=/tmp/go-build -gno-record-gcc-switches"
```

到这里，ubuntu上Go开发环境其实具备了，也可以趁热打铁，将GOPATH路径下的子目录都创建一下。

```
mkdir -p $GOPATH/src
mkdir -p $GOPATH/bin
mkdir -p $GOPATH/pkg
```

- macOS

在macOS上安装Go语言也可以选择下载安装包或命令行的形式，如果想下载安装包安装的话，可以参考之前在ubuntu上的安装做法，两者操作基本类似，只是包名不一样，以及配置环境变量时的文件名不一样。

```
https://studygolang.com/dl/golang/go1.12.7.darwin-amd64.pkg
```

命令行安装非常简单，使用brew工具即可！

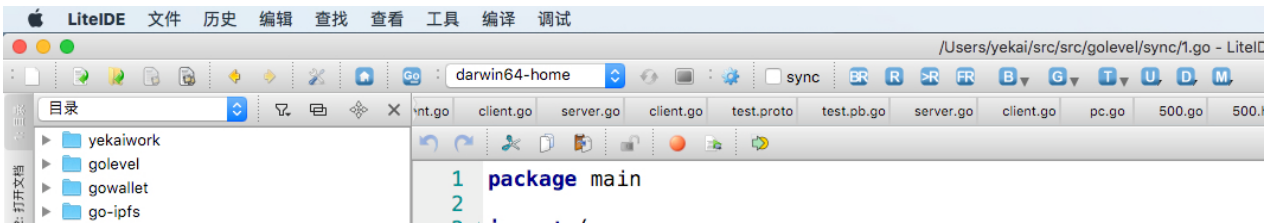
```
brew install golang
```

## 2.2 IDE开发工具安装

Go语言的IDE开发工具有很多，开发者完全可以根据自己喜好来选择，比如GoLand，VsCode，sublime text3等等，本人比较喜欢用LiteIDE。开发人员可以前往[github地址下载](#)，根据自己的操作系统选择对应的版本。

 <a href="#">liteidex36.archlinux-pkgbuild.zip</a>	2.98 KB
 <a href="#">liteidex36.linux64-qt4.8.7-system.tar.gz</a>	14.8 MB
 <a href="#">liteidex36.linux64-qt4.8.7.ApplImage</a>	22.6 MB
 <a href="#">liteidex36.linux64-qt4.8.7.tar.gz</a>	22.6 MB
 <a href="#">liteidex36.linux64-qt5.5.1-system.tar.gz</a>	14.9 MB
 <a href="#">liteidex36.linux64-qt5.5.1.ApplImage</a>	41 MB
 <a href="#">liteidex36.linux64-qt5.5.1.tar.gz</a>  linux	41.1 MB
 <a href="#">liteidex36.macos-qt5.9.5.zip</a>  macOS	29.7 MB
 <a href="#">liteidex36.windows-qt4.8.5.zip</a>	39.2 MB
 <a href="#">liteidex36.windows-qt5.9.5.zip</a>  windows	36.6 MB
 <a href="#">Source code (zip)</a>	

这个文件解压后可以直接使用，安装后效果如下图所示：



IDE的使用，本文不做详细说明，点一点就会了，不过需要注意一点的是，Go语言是面向工程型的语言，每个工程里只能有一个main，所以同一个目录下有多个包含main函数的文件时是不能使用IDE编译并运行的，这时仍然需要借助命令行的操作，我们将在后面介绍如何编译和运行Go代码。

## 2.3 go工具集

使用go help可以查看go所有指令的帮助，对于初学者来说，先记住几个常用的即可。

```
localhost:~ yk$ go help
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

build      compile packages and dependencies
clean      remove object files and cached files
doc        show documentation for package or symbol
env        print Go environment information
bug        start a bug report
fix        update packages to use new APIs
fmt        gofmt (reformat) package sources
generate   generate Go files by processing source
get        download and install packages and dependencies
install    compile and install packages and dependencies
list       list packages
run        compile and run Go program
test       test packages
tool       run specified go tool
version    print Go version
vet        report likely mistakes in packages

Use "go help [command]" for more information about a command.
```

一些常用命令说明：

- go build 编译代码
- go run 编译+运行代码
- go get 下载第三方源码，多数情况下在github下载，需要提供url，并且url支持git clone
- go doc 查看包手册
- go install 编译并且安装包

- go test 测试代码，要求文件必须以test开头
- fmt 重新格式化代码

## 3 基础语法

---

从本章开始，我们将介绍Go语言的语法知识，接下来跟我一起来写代码吧！

### 3.1 初识Golang

别的先不说，先来看一段hello-world的代码：

```
package main

import (
    "fmt"
)

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

上述代码有三处关键字，它们的作用如下：

- package 定义包名
- import 导入包名
- func 函数定义关键字

main函数是程序执行的入口函数，这与其他语言是一致的，通过上述代码，可以发现Go语言中引入了包（package）的使用，相关的API调用都可以通过“包名.函数名”的方式调用。作为main函数所在的文件，包名则必须是main，fmt.Println很好理解，就是fmt包为开发者提供的一个格式化输出函数，该函数会自动的在输出后添加一个换行。下面的问题是，这个代码怎么运行起来呢？

首先将代码保存为01-hello.go，使用go build命令，可以对源码进行编译，这时在该目录下可以看到一个01-hello的可执行文件。

```
localhost:day01 yk$ go build 01-hello.go
localhost:day01 yk$ ls
01-hello    01-hello.go
```

运行01-hello就可以看到“hello world”的输出。

```
localhost:day01 yk$ ./01-hello
hello world
```

除了这种方式，在前面介绍go语言工具集的时候，还看到过一个go run指令，当运行go run指令的时候，代码可以直接编译后运行，此时不会产生那个01-hello的可执行文件（虽然没有产生文件，但需要注意，Go语言是编译型语言，go run实际是将该文件编译成临时目录的一个临时文件，运行后删除）。使用go run运行效果如下。

```
localhost:day01 yk$ rm 01-hello
localhost:day01 yk$ go run 01-hello.go
hello world
localhost:day01 yk$ ls
01-hello.go
```

### 3.2 数据类型

Go语言的数据类型定义非常丰富，我们来分别了解一下。

- 布尔类型 bool

true 或者 false

- 整型

类型	取值范围	描述
uint8	0 - 255	无符号8位整数
uint16	0 - 65535	无符号 16 位整型
uint32	0 - 4294967295	无符号 32 位整型
uint64	0 - 18446744073709551615	无符号 64 位整型
int8	-128 - 127	有符号8位整数
int16	-32768 - 32767	有符号 16 位整型
int32	-2147483648 - 2147483647	有符号 32 位整型
int64	-9223372036854775808 - 9223372036854775807	有符号 64 位整型

- 浮点型

类型	描述
float32	IEEE-754 32位浮点型数
float64	IEEE-754 64位浮点型数
complex64	32 位实数和虚数
complex128	64 位实数和虚数

备注：IEEE-754为IEEE二进制浮点数算术标准（ANSI/IEEE Std 754-1985）



- 其他类型

类型	描述
byte	类似uint8
rune	类似uint32
uint	32或64位，取决于操作系统
int	与uint一样大小
uintptr	无符号整形，存放指针

## 3.3 变量

Go语言中，变量的声明语法如下：

```
var identifier type
```

也可以选择在声明的同时直接赋值：

```
var identifier [type] = value
```

Go语言可以自动推导数据类型，因此在类型定义的时候，可以省略type。也可以同时定义多个变量，在赋值的时候，等号两端变量和值按照顺序一一对应。示例如下：

```
var x, y = 123, "hello"
```

变量定义时var关键字也可以省略，这也是Go开发者更喜欢的方式，那就是用“:=”，“:=”使用时有区域限制，必须出现函数中，这种方式可以非常方便的定义变量，以及接收函数调用后返回结果，它利用的仍然是Go语言自动推导类型的特性。示例如下：

```
a := 123
str := "hello"
```

当然“:=”还有一个注意事项，使用“:=”的前提是当前代码段，该变量没有定义过。下面来段代码，跑一跑试试。

```
package main

import (
    "fmt"
)

func main() {
```

```

fmt.Println("hello world")
var v1 int
var x, y = 123, "hello"
a, str := 456, "world"
fmt.Println(v1, x, y, a, str)
//定义复数，注意i一定要与数字紧紧相连，否则会被当成字符i处理
var c1 complex64 = 4 + 3i
fmt.Println(c1)

}

```

运行结果如下：

```

localhost:day01 yekai$ go run 02-var.go
hello world
0 123 hello 456 world
(4+3i)

```

## 3.4 指针

在C语言里，指针属于必备技能，虽然Go语言也有指针，但Go语言中的指针并没有像C语言里那样妖魔化，指针所表达的就是它指向变量的地址。指针背后的作用往往才是更关键的，那就是值传递与引用传递。

```

a := 10
var b *int = &a // b指向a的地址
fmt.Println(*b) //打印的是10
*b = 100 //通过b可以修改a的值
fmt.Println(a, *b) // 此时打印的都是100

```

如果我们想对两个变量的值进行互换，函数像下面这样写，最后得到的是失败的结果。

```

func swap(a, b int) {
    temp := a
    a = b
    b = temp
}

```

很显然值传递的方式不能解决变量替换的问题，通过指针是可以的。

```

func swap2(a, b *int) {
    temp := *a
    *a = *b
    *b = temp
}

```

当然Go语言的强大特性也可以通过其他方式解决值交换的问题，如果了解了函数的多返回值，到时候不用指针也可以同样解决问题。

## 3.5 常量

所谓常量，就是在程序运行时，值不会被修改的标识。在Go语言当中，使用const关键字定义常量。常量的定义语法如下：

```
const identifier [type] = value
```

根据我们的经验，[type]这样被中括号扩起来的代表可以省略，也就是常量也可以支持类型推导。同样的，常量也可以一次定义多个，这一点与变量是相同的。来看一个示例：

```
/*
    author:Yekai
    company:Pdj
    filename:03-const.go
*/
package main

import "fmt"

func main() {
    const LENGTH int = 10
    const WIDTH = 5
    const a, b, c = 1, false, "str" //多常量同时定义且赋值

    //注意用 := ,代表定义变量area
    area := LENGTH * WIDTH
    //Printf与Printfln的区别是格式化以及自带换行
    fmt.Printf("area is %d\n", area)
    println(a, b, c)
}
```

运行结果如下：

```
localhost:day01 yekai$ go run 03-const.go
area is 50
1 false str
```

原则上不推荐对常量定义时使用函数，如果非要用，可以使用Go语言内置函数，如：len(), cap(), unsafe.Sizeof()等，否则编译很可能无法通过。例如代码写成这样编译器是无法放行的！

```
func test() int {
    return 10
}

const (
    a = "abc"
    b = test()
    c = len(a)
)

func main() {
    println(a, b, c)
}
```

编译时你将看到这样的错误：

```
# command-line-arguments
./03-const.go:25:2: const initializer test() is not a constant
```

常量还有一种用法是定义枚举，为了更优雅的定义枚举类型，Go语言引入了iota，你可以认为iota是可能被编译器修改的常量，其实也没有那么复杂，iota就是一个从0开始的自增序列。示例代码如下：

```
/*
    author:Yekai
    company:Pdj
    filename:04-iota.go
*/
package main

import (
    "fmt"
)

//定义业务类型
const (
    login = iota // iota = 0
    logout      // iota = 1
    user    = iota + 2 //iota = 2,user = 2+2 = 4
    account = iota * 3 //iota = 3, account = 3*3 = 9
)

const (
    apple, banana = iota + 1, iota + 2 // iota = 0
    peach, pear           //iota = 1
    orange, mango         //iota = 2
)

func main() {
```

```
fmt.Println(login, logout, user, account)
fmt.Println(apple, banana, peach, pear, orange, mango)
}
```

执行结果如下：

```
localhost:day01 yekai$ go run 04-iota.go
0 1 4 9
1 2 2 3 3 4
```

## 3.6 条件分支

分支语句控制也是一门开发语言的必备技能，分支控制时关键是要使用条件表达式。

下面来看一个if-else的例子：

```
/*
    author:Yekai
    company:Pdj
    filename:05-if.go
*/
package main

import (
    "fmt"
)

func main() {
    a := 10

    if a > 10 { //左括号一定要写在表达式同行，与函数要求相同
        fmt.Println("My God ,a is ", a)
    } else if a < 10 {
        fmt.Println("a is too small")
    } else {
        fmt.Println("a == ", a)
    }
}
```

运行结果如下：

```
localhost:day01 yekai$ go run 05-if.go
a == 10
```

也要特别注意一点：Go语言中if后的表达式必须是bool型值，否则语法检测不通过。

与其他语言类似，Go语言也支持switch，switch的特点是可以针对某个变量可能的值进行分支匹配处理，语法如下：

```

switch var1 {
    case val1:
        ...
    case val2:
        ...
    default:
        ...
}

```

示例代码如下：

```

/*
    author:Yekai
    company:Pdj
    filename:06-switch.go
*/
package main

import (
    "fmt"
)

func main() {
    var fruit string
    fmt.Println("Please input a fruit's name:")
    fmt.Scanf("%s", &fruit)
    switch fruit {
    case "banana":
        fmt.Println("I want 2 banana!")
    case "orange":
        fmt.Println("I want 3 orange!")
    case "apple":
        fmt.Println("I want an apple!")
    case "pear":
        fmt.Println("I do not like pear!")
    default:
        fmt.Println("Are you kidding me?")
    }
}

```

执行结果如下：

```

localhost:day01 yekai$ go run 06-switch.go
Please input a fruit's name:

```

键盘输入：orange

```
I want 3 orange!
```

```
localhost:day01 yekai$ go run 06-switch.go  
Please input a fruit's name:
```

键盘输入: apple

```
I want an apple!
```

```
localhost:day01 yekai$ go run 06-switch.go  
Please input a fruit's name:
```

键盘输入: mango

```
Are you kidding me?
```

键盘输入部分需要读者手动输入内容，从执行的结果也可以看出，Go语言中的分支执行不像c语言那样，每个分支里都需要加break，否则会顺序执行之后的分支。

## 3.7 for循环

G语言语法追求简洁，连循环语法都只使用一个关键字。Go语言种循环没有while关键字，取而代之的都是由for来处理，for支持三种方式：

1. for init; condition; post {}
2. for condition {}
3. for {}

我们还是先看例子：

```
/*  
    author:Yekai  
    company:Pdj  
    filename:07-for.go  
*/  
  
package main  
  
import (  
    "fmt"  
    "time"  
)  
  
func main() {  
    //计算1+2+3+.....+100 = ?
```

```

//第一种方式
sum := 0
i := 0
for i = 1; i <= 100; i++ {
    sum += i
}
fmt.Println("sum is ", sum)
//第二种方式
i = 1
sum = 0
for i <= 100 {
    sum += i
    i++
}
fmt.Println("sum is ", sum)
//死循环 - 开启刷屏模式
for {
    fmt.Println("heihei")
    time.Sleep(time.Second * 1) //每次执行睡眠1s
}
}

```

执行结果如下，最后按ctrl+c中断该进程：

```

localhost:day01 yekai$ go run 07-for.go
sum is 5050
sum is 5050
heihei
heihei
heihei
heihei
heihei
heihei
^Csignal: interrupt

```

## 3.8 函数

函数是针对某些特定功能的封装，便于重复使用，函数也是一门语言当中非常重要的部分，Go语言函数的格式如下：

```

func function_name( [parameter list] ) [return_types] {
    函数体
}

```

解释说明：

- func 函数关键字



- function\_name 函数名称
- parameter list 参数列表，根据设计需要，提供参数的顺序和类型要求，可以为空
- return\_types 返回类型，支持0或多个返回类型，如果超过一个类型时，需要用小括号
- 函数体 函数的功能实现部分

来看一个例子：

```
/*
    author:Yekai
    company:Pdj
    filename:08-func.go
*/

package main

import (
    "fmt"
)

func main() {
    //函数调用，同时获得2个返回值
    sum, sub := add_sub(32, 21)
    fmt.Println(sum, sub)
    //获得函数指针，此时addsubptr相当于 func add_sub(a int, b int) (int, int)
    addsubptr := add_sub
    //通过函数指针的方式调用
    sum1, sub1 := addsubptr(1, 2)
    fmt.Println(sum1, sub1)
}

//实现一个加法和减法一起做的函数
func add_sub(a int, b int) (int, int) {
    return a + b, a - b
}
```

例子中，我们实现了一个函数 add\_sub，它有2个参数，a和b，这种写法属于常规写法，如果相邻参数相同时，我们也可以简化类型，写成下面这样：

```
func add_sub(a, b int) (int, int) {
    return a + b, a - b
}
```

此外，函数也可以作为函数的参数进行传递。

```
/*
    author:Yekai
    company:Pdj
```

```

filename:09-func2.go
*/

package main

import (
    "fmt"
)

func main() {
    a := addorsub(10, 20, add) //传入add函数, 求和
    b := addorsub(10, 20, sub) //传入sub函数, 求差
    fmt.Println(a, b)
}

func add(a int, b int) int {
    return a + b
}

func sub(a int, b int) int {
    return a - b
}

//函数作为特殊的类型也可以当作参数,调用时要求f参数必须是 func(a, b int) int 这样类型的函数
func addorsub(a, b int, f func(a, b int) int) int {
    return f(a, b)
}

```

上述例子addorsub函数的第三个参数f就是一个函数，调用时它可以是add，也可以是sub，前提是这两个函数的声明必须与f的类型一致。执行结果如下：

```

localhost:day01 yekai$ go run 09-func2.go
30 -10

```

有的时候，我们可能会为了支持一个功能而实现一个函数，但又不想给这段功能起一个函数名字，此时可以定义匿名函数，Go语言也是借鉴了其他语言的特点，在语法上支持了匿名函数，匿名函数与后文介绍Go并发相互配合，杀伤力巨大。

```

func(a, b int) int {
    return a + b
}(3, 4) //匿名函数声明+调用

```

注意上述代码为一个匿名函数执行的代码段，它与普通函数的区别是没有函数名称，并且在{}之外还有调用参数传递，如果没有参数，直接放()就可以了。

由于对匿名函数的支持，再加上Go语言的函数也可以作为返回值，所以在Go语言中可以支持闭包，所谓闭包就是子函数能够读取其外部函数的局部变量。下面的例子是一个模拟数据库自增序列的方法。

```
/*
    author:Yekai
    company: Pdj
    filename: 10-func-closure.go
*/

package main

import "fmt"

//出现函数闭包
func getSequence() func() int {
    i := 0
    return func() int {
        i += 1 //子函数内访问了i，导致i不会被释放
        return i
    }
}

func main() {
    // nextNumber 为一个函数，函数 i 为 0
    nextNumber := getSequence()

    // 调用 nextNumber 函数，i 变量自增 1 并返回
    fmt.Println(nextNumber())
    fmt.Println(nextNumber())
    fmt.Println(nextNumber())

    // 创建新的函数 nextNumber1，并查看结果
    nextNumber1 := getSequence()
    fmt.Println(nextNumber1())
    fmt.Println(nextNumber1())
}
```

在上述代码中，getSequence函数内部变量i正常的生命周期是函数执行结束后，在getSequence内部又返回一个匿名函数，由于这个匿名函数返回的是i的值，这样就导致i仍然在内存区域中不会释放，当使用nextNumber := getSequence()也就拿到了这个匿名函数，每调用一次，i的值都会增加，就类似于实现了数据库里面的自增序列，那么nextNumber1也是等于getSequence()，它代表的序列是与nextNumber一致呢，还是重新开始呢？执行一下，可以思考一下为什么？

```
localhost:day01 yekai$ go run 10-func-closure.go
1
2
3
1
2
```

从执行结果来看，nextNumber1又从1开始重新开始了，虽然他们函数内部变量名字都是i，但是在不同的函数调用内部，i所在的内存区域是不同的，因而nextNumber和nextNumber1代表的必然是不同的序列。

## 3.9 容器编程

Go语言当中为我们提供了数组、切片、map等容器工具来管理一些数据元素，接下来我们逐一介绍。

### 3.9.1 数组

Go语言当中，可以将一组类型相同的数据存放在一个数组当中，数组中元素的类型可以是Go语言原生类型，也可以是自定义类型（struct）。声明方式可以对比变量，灵活记忆。

```
var variable_name [SIZE] variable_type
```

具体可以参考几个例子：

```
var sa [10] int64 //定义一个10个int64类型的数组
var ss [3] string = [3]string{"yekai", "fuhongxue", "luxiaojia"} //定义3个长度的字符串数组
```

Go语言的fmt包功能很强大，它可以直接打印各种变量，无论是容器，还是结构体，都可以直接打印。示例如下：

```
/*
    author:Yekai
    company:Pdj
    filename:11-array.go
*/

package main

import (
    "fmt"
)

func main() {
    var a1 [5]int = [5]int{1, 2, 3, 4} //定义并初始化
    fmt.Println(a1)
    a1[4] = 6 //利用下标访问数组，对数组元素进行赋值
```

```

fmt.Println(a1)
s1 := [4]string{"yekai", "fuhongxue", "luxiaojia"} //元素个数不能超过数组个数
fmt.Println(s1)
}

```

执行结果如下：

```

localhost:day01 yekai$ go run 11-array.go
[1 2 3 4 0]
[1 2 3 4 6]
[yekai fuhongxue luxiaojia ]

```

在Go语言当中，同样可以定义二维数组，具体参考下面例子即可。

```

/*
    author:Yekai
    company:Pdj
    filename:12-array2.go
*/

package main

import (
    "fmt"
)

func main() {

    //Go语言当中的二维数组,可以理解为3行4列
    a2 := [3][4]int{
        {0, 1, 2, 3}, /* 第一行索引为 0 */
        {4, 5, 6, 7}, /* 第二行索引为 1 */
        {8, 9, 10, 11}, /* 第三行索引为 2 */
    }
    //注意上述数组初始化的逗号
    fmt.Println(a2)
    //如何遍历该数组? 可以写2层for循环搞定
    for i := 0; i < 3; i++ {
        for j := 0; j < 4; j++ {
            fmt.Printf("i = %d, j = %d, val = %d\n", i, j, a2[i][j])
        }
    }
}

```

执行结果如下：

```
localhost:day01 yekai$ go run 12-array2.go
[[0 1 2 3] [4 5 6 7] [8 9 10 11]]
i = 0, j = 0, val = 0
i = 0, j = 1, val = 1
i = 0, j = 2, val = 2
i = 0, j = 3, val = 3
i = 1, j = 0, val = 4
i = 1, j = 1, val = 5
i = 1, j = 2, val = 6
i = 1, j = 3, val = 7
i = 2, j = 0, val = 8
i = 2, j = 1, val = 9
i = 2, j = 2, val = 10
i = 2, j = 3, val = 11
```

### 3.9.2 切片

由于数组是固定大小的，这在使用上缺少一定便利，因此Go语言又提供了切片类型，乍一看切片与数组没有区别，只不过它的大小是可以扩充的，也就是说可以把切片理解成动态数组，至于为什么叫切片，可以脑补一下面包片（把长面包想象成一个数组，切片是从其某段切下来）。



切片中有2个重要概念：长度和容量。

- 长度 指被赋过值的最大下标+1
- 容量 指切片能容纳的最多元素个数

切片可以用`array[slice[start:end]]`的方式进行截取，从而得到新的切片，其中`start`和`end`代表下标位置，都可以省略，`start`省略时代表从头开始，`end`省略时代表直到末尾。

```
/*
author:Yekai
```

```

    company: Pdj
    filename: 13-slice.go
*/
package main

import (
    "fmt"
)

func main() {
    a1 := [5]int{1, 2, 3, 4, 5} // a1 是一个数组
    s1 := a1[2:4]               // 定义一个切片
    fmt.Println(a1)
    fmt.Println(s1)
    s1[1] = 100 // 切片下标不能超过
    fmt.Println("after-----")
    fmt.Println(a1)
    fmt.Println(s1)
}

```

执行结果如下：

```

localhost:day01 yekai$ go run 13-slice.go
[1 2 3 4 5]
[3 4]
after-----
[1 2 3 100 5]
[3 100]

```

通过上述例子，我们可以得到2个知识点：

1. 切片start: end是前闭后开，也就是实际截取下标是start到end-1
2. 切片是引用类型，对切片的修改会影响对应的数组（实际上并不绝对）

想要研究切片，可以再了解一下与切片相关的内建函数：

- len 计算切片长度

```
len(s)
```

- append 向切片追加元素

```

var s1 []T
append(s1, T)

```

- make 可以创建切片

```
make([]T, length, capacity) // capacity 可以省略，默认与len一致
```

- copy 复制切片

```
copy(s2,s1) //将s1内容拷贝到s2, 此时s1与s2是独立的, 修改互不干预
```

来看看一段具体代码:

```
/*
    author:Yekai
    company:Pdj
    filename:14-slice2.go
*/
package main

import (
    "fmt"
)

func main() {
    var s1 []int          //定义切片s1
    s1 = append(s1, 1)    //追加, 注意s1必须接收追加结果
    s1 = append(s1, 2)
    s1 = append(s1, 3, 4, 5) //可以一次追加多个
    printSlice(s1)
    s2 := make([]int, 3)
    printSlice(s2)
    s2 = append(s2, 4)    //当超过容量的时候, 容量会以len*2的方式自动扩大
    printSlice(s2)
}

func printSlice(s []int) {
    fmt.Printf("len = %d, cap = %d, s = %v\n", len(s), cap(s), s)
}
```

运行结果:

```
localhost:day01 yekai$ go run 14-slice2.go
len = 5, cap = 6, s = [1 2 3 4 5]
len = 3, cap = 3, s = [0 0 0]
len = 4, cap = 6, s = [0 0 0 4]
```

通过上述例子我们看到, 当切片容量不够时, 追加不会报错, 而会扩大容量, 扩大容量默认采用 $len*2$ 的数据 (在数据量不是特别大的情况下都会如此)。说句题外话, 根据多个例子编写, 相信大家也发现了, Go语言默认都会对变量进行初始化, 这一点很友好!

### 3.9.3 map键值对

Go语言同样提供了map这样的容器, map是一种集合, 但注意它是无序的, map可以存放无序的键值对。map需要使用make来构造, 否则它是一个nil-map, 无法存放键值对。



```
var map_variable map[key_data_type]value_data_type
map_variable = make(map[key_data_type]value_data_type)
```

或

```
map_variable := make(map[key_data_type]value_data_type)
```

来看一段示例代码。

```
/*
    author:Yekai
    company:Pdj
    filename:15-map.go
*/
package main

import "fmt"

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

    // map插入key - value对,各个国家对应的首都
    countryCapitalMap["France"] = "Paris"
    countryCapitalMap["Italy"] = "Roma"
    countryCapitalMap["China"] = "BeiJing"
    countryCapitalMap["India "] = "New Delhi"

    fmt.Println(countryCapitalMap["China"])
    //当key不存在时, countryCapitalMap["China"]的取值方式不够严谨
    //map取值时,推荐使用指示器方式
    val, ok := countryCapitalMap["Japan"]
    if ok {
        fmt.Println("Japan's capital is", val)
    } else {
        fmt.Println("Japan's capital not in map")
    }
}
```

在取map的值时可以用一个变量接收,同时使用一个指示变量来判断key值是否真实存在。

关于map的存入与读取数据已经没问题了,但还有一个问题,如何对map进行遍历呢?由于map并非像数组那样使用连续的内存单元,遍历起来会有些麻烦,为此,Go语言为开发者提供了range关键字,它可以帮助我们优雅的遍历Go语言各种容器,当然包括map。具体用法可以看代码:

```
/*
    author:Yekai
    company:Pdj
```

```

filename:16-range.go
*/
package main

import "fmt"

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

    // map插入key - value对,各个国家对应的首都
    countryCapitalMap["France"] = "Paris"
    countryCapitalMap["Italy"] = "Roma"
    countryCapitalMap["China"] = "BeiJing"
    countryCapitalMap["India "] = "New Delhi"

    //遍历map
    for k, v := range countryCapitalMap {
        fmt.Println(k, "'s capital is", v) //k,v分别是map的key和val
    }
    //遍历数组
    a := []int{10, 20, 30, 40, 50}
    for k, v := range a {
        fmt.Printf("a[%d]=%d\n", k, v) //k代表数组下标, v代表该元素值
    }
}

```

执行结果：

```

localhost:day01 yekai$ go run 16-range.go
China 's capital is BeiJing
India  's capital is New Delhi
France 's capital is Paris
Italy 's capital is Roma
a[0]=10
a[1]=20
a[2]=30
a[3]=40
a[4]=50

```

如果不想获得k或v的值时，可以用\_占位，这个语法在多变量赋值时同样适用。

```

_, v := range countryCapitalMap //只取v的值

```

## 4 核心编程

## 4.1 面向对象

Go语言也可以面向对象编程，只不过语言内部并未设计class这样的关键字，简洁、巧妙是Go语言面向对象的特点，下面我们一起来了解一下。

### 4.1.1 结构体

先来说说结构体，Go语言当中，使用type和struct关键字来定义结构体，比如定义一个人的结构，人的属性可以包含：姓名，年龄，性别，战斗力等。

```
type Person struct {  
    Name  string  
    Age   int  
    Sex   string  
    Fight int  
}
```

Person定义后，可以用其来构造一个战斗力为5的人，就把他叫做：战五渣，示例代码如下。

```
p1 := Person{"战五渣", 30, "man", 5}  
fmt.Printf("%+v\n", p1) // %+v 可以清晰的打印结构体数据
```

完整运行代码如下：

```
/*  
    author:Yekai  
    company:Pdj  
    filename:17-struct.go  
*/  
package main  
  
import (  
    "fmt"  
)  
  
type Person struct {  
    Name  string  
    Age   int  
    Sex   string  
    Fight int  
}  
  
func main() {  
    p1 := Person{"战五渣", 30, "man", 5}  
    fmt.Printf("%+v\n", p1) // %+v 可以清晰的打印结构体数据  
}
```

执行结果：

```
localhost:day01 yekai$ go run 17-struct.go
{Name:战五渣 Age:30 Sex:man Fight:5}
```

### 4.1.2 封装实现

介绍封装先要澄清一个概念，那就是方法和函数，函数之前我们介绍过，可以根据功能需要设计并实现我们想要的函数，那么什么又是方法呢？方法与函数的区别主要在于方法是属于某个结构体的，仅限于结构体对象方可调用。不过不像大多数语言那样定义，方法的定义与结构体定义是脱离的，但是作为方法必须要指定一个调用者（或接受者）。方法的定义语法如下。

```
func (obj ObjT) funcName([params list]) [return list] {
    do sth
}
```

我们来实现一个例子，如在平面直角坐标系内计算2个点之间的距离，按照正常的方式，我们可以定义一个点的结构体Point(x,y)，然后定义2个节点p1,p2，之后利用数学公式：

$$dis = \sqrt{(p2.x - p1.x)^2 + (p2.y - p1.y)^2}$$

于是计算2个点之间的距离，我们可以实现这样的一函数：

```
type Point struct {
    x, y float64
}

func distance(p1, p2 Point) float64 {
    //math.Sqrt的功能是计算输入参数的平方根
    return math.Sqrt((p2.x-p1.x)*(p2.x-p1.x) + (p2.y-p1.y)*(p2.y-p1.y))
}
```

这和我们前面提到的封装和方法似乎没有关系，接下来我们考虑另外一种方式，之前的计算2个点之间的距离是以第三者的角度去观察，现在我们站在一个点的角度去考虑，想要计算这个点到某个点的距离时，只需要传入一个点就够了，于是我们可以定义下面的方法：

```
func (this Point) distance2(p Point) float64 {
    return math.Sqrt((this.x-p.x)*(this.x-p.x) + (this.y-p.y)*(this.y-p.y))
}
```

这里的this只是点的对象名，并非一定要使用this。distance2的调用必须是Point结构体对象才可以调用。

```
/*
    author:Yekai
    company:PdJ
    filename:18-enclosure.go
*/
package main
```

```

import (
    "fmt"
    "math"
)

type Point struct {
    x, y float64
}

//第一种方法, 传入2个点
func distance(p1, p2 Point) float64 {
    return math.Sqrt((p2.x-p1.x)*(p2.x-p1.x) + (p2.y-p1.y)*(p2.y-p1.y))
}

//第二种方法, 以一个点作为参照点, 再传入一个点
func (this Point) distance2(p Point) float64 {
    return math.Sqrt((this.x-p.x)*(this.x-p.x) + (this.y-p.y)*(this.y-p.y))
}

func main() {
    p1 := Point{0.0, 0.0}
    p2 := Point{3.0, 4.0}
    fmt.Println(p2, "between", p1, "distance is ", distance(p1, p2))
    fmt.Println("call distance2 =", p2.distance2(p1))
}

```

执行结果如下:

```

localhost:day01 yekai$ go run 18-enclosure.go
{3 4} between {0 0} distance is 5
call distance2 = 5

```

### 4.1.3 Go语言中的继承

Go语言语法简洁, 在继承上体现的真是淋漓尽致。我们之前定义过Person结构体, 现在再来定义一个超人SuperMan的结构体, 他继承Person的信息。可以这样写:

```

type Person struct {
    Name  string
    Age   int
    Sex   string
    Fight int
}

type SuperMan struct {
    Strength int
    Speed    int
    Person    //内嵌方式, SuperMan继承Person全部能力
}

```

结构体可以这样赋值：

```

s1 := SuperMan{
    Strength: 100,
    Speed:    99,
    Person: Person{
        Name: "Kelak",
        Age:  40,
        Sex:  "man",
        Fight: 5000,
    }, //这里的“,”不能省略
}

```

Person实现的任何方法，SuperMan的对象都可以直接调用。

```

//修改结构体成员时，方法接收器要定义为指针类型，否则值传递方式并不能修改结构本身数据
func (p *Person) setAge(age int) {
    p.Age = age
}

func (s SuperMan) Print() {
    fmt.Printf("Name = %s, Age = %d, Sex = %s, Fight = %d\n", s.Name, s.Age,
s.Sex, s.Fight)
    fmt.Printf("strength = %d, fight = %d\n", s.Strength, s.Speed)
}

```

全部代码如下：

```

/*
    author:Yekai
    company:Pdj
    filename:19-inherit.go
*/
package main

```

```

import (
    "fmt"
)

type Person struct {
    Name  string
    Age   int
    Sex   string
    Fight int
}

func (p *Person) setAge(age int) {
    p.Age = age
}

type SuperMan struct {
    Strength int
    Speed    int
    Person
}

func (s SuperMan) Print() {
    fmt.Printf("Name = %s, Age = %d, Sex = %s, Fight = %d\n", s.Name, s.Age,
s.Sex, s.Fight)
    fmt.Printf("strength = %d, fight = %d\n", s.Strength, s.Speed)
}

func main() {
    s1 := SuperMan{
        Strength: 100,
        Speed:    99,
        Person: Person{
            Name:  "Kelak",
            Age:   40,
            Sex:   "man",
            Fight: 5000,
        },
    }
    fmt.Println(s1)
    s1.setAge(41)
    s1.Print()
}

```

执行结果如下：

```
localhost:day01 yekai$ go run 19-inherit.go
{100 99 {Kelak 40 man 5000}}
Name = Kelak, Age = 41, Sex = man, Fight = 5000
strength = 100, fight = 99
```

#### 4.1.4 Go语言的多态

Go语言借助interface（接口）来支持多态。接下来我们定义一个Animal接口，代码如下：

```
type Animal interface {
    Sleep() //接口方法1
    Eating() //接口方法2
}
```

Animal接口定义了2个方法：Sleep和Eating，开发者如果想让自定义结构的对象能够赋值给Animal接口对象，就必须支持Animal定义的全部方法。比如我们定义一个小猫结构体 and 一个小狗结构体。

```
type Cat struct {
    Color string
}

type Cat struct {
    Color string
}
//小猫结构体的方法
func (c Cat) Sleeping() {
    fmt.Println(c.Color, "Cat is sleeping")
}
func (c Cat) Eating() {
    fmt.Println(c.Color, "Cat is Eating")
}

//小狗结构体的方法
func (c Dog) Sleeping() {
    fmt.Println(c.Color, "Dog is sleeping")
}
func (c Dog) Eating() {
    fmt.Println(c.Color, "Dog is Eating")
}
func (c Dog) Print() {
    fmt.Println("Dog's color is", c.Color)
}
```

再来实现一个工厂的方法，它的返回值是Animal类型，由于Dog或Cat可以赋值给Animal，那么这个Factory既可以生产Cat，也可以生产Dog。



```

func Factory(color string, animal string) Animal {
    switch animal {
    case "dog":
        return &Dog{color}
    case "cat":
        return &Cat{color}
    default:
        return nil
    }
}

func main() {
    d1 := Factory("black", "dog")
    d1.Eating()
    c2 := Factory("white", "cat")
    c2.Sleeping()
}

```

执行结果如下：

```

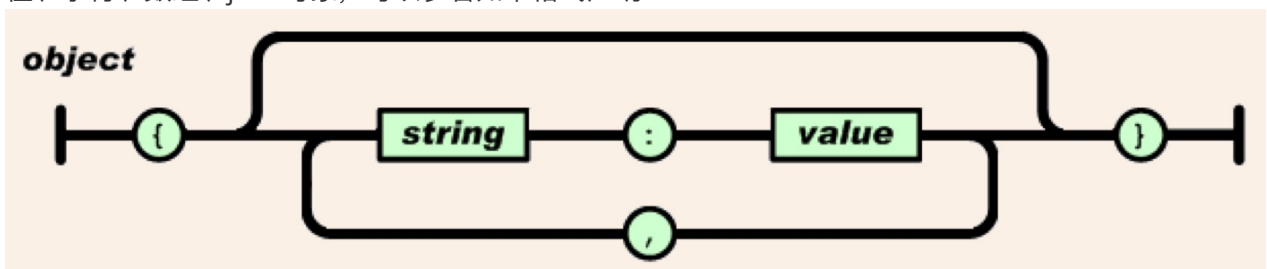
localhost:day01 yekai$ go run 20-polymorphic.go
black Dog is Eating
white Cat is sleeping

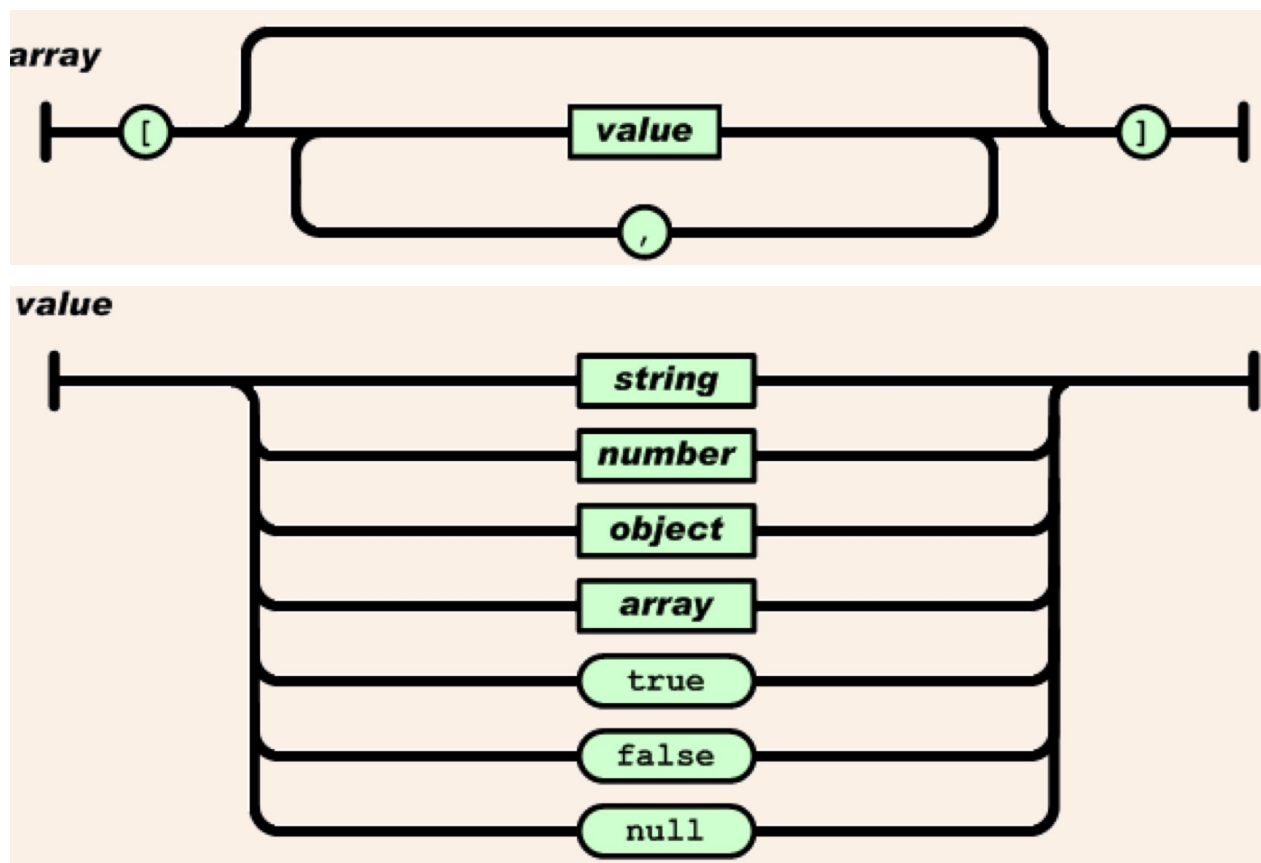
```

## 4.2 json处理

前面的内容除了介绍Go语言，也算是表达了Go语言在对其他语言优点的继承，接下来我们介绍一些Go语言特色的内容。

json是一种数据格式，源自JavaScript语言，可以很好的描述数据，方便在不同的语言之间进行通信，Go语言对json有很好的支持。首先应该对json格式有个了解，json就是键值对的集合，值类型可以数值、字符、数组、json对象，可以参看如下格式声明：





可以来看一个具体的例子：

```
{
  "Name": "yekai",
  "Age": 35,
  "Sex": "man",
  "Favirate": {
    "wuqi": "feidao",
    "food": "huasheng"
  }
}
```

json格式很好理解，我们接下来尝试在Go语言与json打交道。从两个方向去掌握也就够了，一个是学会如何在Go语言当中构造json，另外就是学会如何在Go语言当中解析一个json的字符串。

- 构造json

构造json首先还是要了解json的格式，看上面的例子，这个json格式是key-val的组合，很容易联想到结构体其实也是key-val的组合，所以我们可以进一步联想到要构造json，可以先定义一个结构体。还是以Person为例，先定义好Person结构体，然后就需要查看一下API了。Go语言为我们提供json包，在里面的Marshal可以将结构体数据转换为json格式字符串。Marshal原型如下。

```
func Marshal(v interface{}) ([]byte, error)
```

Marshal的返回结果是[]byte，根据我们前面的学习，应该知道这是一个切片类型，把它强转为string类型就可以看到结果了，第二个返回值error是Go语言特定的错误类型。需要注意的是，结构体中的字段名称首字母必须要大写，否则你将看不到想要的转换结果，这也是Go语言的原则之一。v是一个interface，所以任何类型的数据都可以作为输入参数，你可以把它理解为void（万能类型）。将结构体

转换为json格式的示例代码如下。

```
/*
    author:Yekai
    company:Pdj
    filename:01-json.go
*/
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
    Name      string
    Age       int
    Sex       string
    GoodMan   bool
    Farivate  []string
}

func main() {
    p1 := Person{"yekai", 30, "man", true, []string{"drink", "feidao"}}
    data, err := json.Marshal(p1)
    if err != nil { // Go语言当中使用nil表示空
        fmt.Println("failed to Marshal data ", err)
        return
    }
    fmt.Println(string(data)) //string(data)是类型强制转会
}
```

执行结果如下：

```
localhost:day02 yekai$ go run 01-json.go
{"Name":"yekai","Sex":"man","GoodMan":true,"Farivate":["drink","feidao"]}
```

从上述例子可以看出，字段定义如果首字母不大写，将不会被转换到json数据中。如果想要转换后的json数据key值首字母不大写，需要在结构定义时特别说明一下，示例如下。

```
type Person struct {
    Name      string `json:"name"`
    Age       int    `json:"age"`
    Sex       string `json:"sex"`
    GoodMan   bool   `json:"goodman"`
    Farivate  []string `json:"farivate"`
}
```

得到的结果就会是这样：

```
{ "name": "yekai", "age": 30, "sex": "man", "goodman": true, "farivate":  
  ["drink", "feidao"] }
```

大多数情况，我们知道这些也就够了，有些特殊用法也可以了解一下：

```
// 字段被本包忽略  
Field int `json:"-`"  
// 字段在json里的键为"myName"  
Field int `json:"myName"`  
// 字段在json里的键为"myName"且如果字段为空值将在对象中省略掉  
Field int `json:"myName,omitempty"`  
// 字段在json里的键为"Field"（默认值），但如果字段为空值会跳过；注意前导的逗号  
Field int `json:",omitempty"`
```

- 解析json

在得到一个json格式的字符串后，有时候也需要将其解析，将它转换为可识别的数据，对各个字段的信息都直接掌握。解析json使用的是Unmarshal函数：

```
func Unmarshal(data []byte, v interface{}) error
```

- data 代表要解析的数据
- v 要接收数据使用的地址

根据可逆原则，既然可以用结构体生成json，那么json也可以转换为结构体。示例代码如下：

```
/*  
    author:Yekai  
    company: Pdj  
    filename: 02-json2.go  
*/  
package main  
  
import (  
    "encoding/json"  
    "fmt"  
)  
  
type Person struct {  
    Name      string `json:"name"`  
    Age       int   `json:"age"`  
    Sex       string `json:"sex"`  
    GoodMan   bool   `json:"goodman,omitempty"`  
    Farivate  []string `json:"farivate"`  
}
```

```
func main() {
    var p1 Person
    json_data :=
`{"name":"yekai","age":30,"sex":"man","goodman":false,"farivate":
["drink","feidao"]}`
    json.Unmarshal([]byte(json_data), &p1)
    fmt.Println(p1)
}
```

运行结果如下：

```
localhost:day02 yekai$ go run 02-json2.go
{yekai 30 man false [drink feidao]}
```

## 4.3 map与json

之前我们用Marshal和Unmarshal函数可以顺利的生成和解析json数据，只不过这个过程中都要借助结构体来实现，也就是说要生成或解析一个json，就先要定义这样的一个结构体，这有些繁琐。

联想到json就是键值对，而我们之前介绍的map容器也是键值对，那么json和map之间是否可以相互转换呢？答案是肯定的！示例代码如下。

```
/*
    author:Yekai
    company:PdJ
    filename:03-json-map.go
*/
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    json_map := make(map[string]interface{}) //构造一个map，注意map的值类型应为接口类型
    json_data :=
`{"name":"yekai","age":30,"sex":"man","goodman":false,"farivate":
["drink","feidao"]}`
    //将json数据解析为map
    err := json.Unmarshal([]byte(json_data), &json_map)
    if err != nil {
        fmt.Println("failed to Unmarshal", err)
        return
    }
    fmt.Println(json_map)
```

```
}
```

执行结果如下：

```
localhost:day02 yekai$ go run 03-json-map.go
map[age:30 sex:man goodman:false farivate:[drink feidao] name:yekai]
```

目前拿到了一个map，不过map中的值都是interface{}类型，虽然大多数情况使用上已经没问题，但有些时候我们还是需要明确一下类型，此时可以使用类型断言：

```
//下面用类型断言
name, ok := json_map["name"].(string) //断言name是string类型，ok是指示器
fmt.Println(ok, name)                 //ok为真代表断言成果
isgood, ok := json_map["goodman"].(bool) //bool型断言
fmt.Println(ok, isgood)
farivates, ok := json_map["farivate"].([]interface{}) //对于切片的类型来说，先断
言为泛型切片
fmt.Println(ok, farivates[0])           //当然也可以继续对
farivates[0]进行断言
```

整体执行效果如下：

```
localhost:day02 yekai$ go run 03-json-map.go
map[name:yekai age:30 sex:man goodman:false farivate:[drink feidao]]
true yekai
true false
true drink
```

## 4.4 错误处理

在前面处理json时，我们使用了error这个错误返回类型，接下来我们介绍一下Go语言中的错误处理方式。在开发过程中，我们经常听到错误和异常这2个词语，这两个词语看上去差不多，实际上还是有一些区别的。

- 错误 发生非期望的已知行为，这类错误是可以预见的
- 异常 发生非期望的未知行为

异常是程序执行时发生的未预先定义的错误，这个错误被操作系统捕获，比如类似C语言的段错误那样。Go语言是类型安全的语言，它不会发生编译器或运行时（runtime）无法捕获的异常。所以，在Go语言中处理的实际上只有错误。

对于错误的处理原则无外乎2个，这是需要提前定义好的：

- 可继续错误 当一些错误发生时，并不影响服务继续运行，进程该继续运行
- 无法继续错误 当某些错误发生时，服务已经无法正常运行，此时应该停止进程

### 4.4.1 error处理

至于错误的处理方式，开发者也并不陌生。在大多数的编程开发中，我们多是通过返回值将函数调用情况返回给上层调用者，上层调用者再把自己的反馈传递给更上层调用者，这样逐层向上传递信息，由最高层来决定是退出还是继续。在Go语言当中，我们也可以通过error这个返回值类型进行逐层传递，而error本身是一个interface接口，其定义方式如下。

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

只要实现了Error() 函数，都可以当作错误对象使用。我们在处理时，判断有没有错误，通常都是先判断其值是否为“nil”，这个“nil”是Go语言当中空值的表示方法，如果error非空，则代表存在错误，需要处理。

### 4.4.2 panic与recover

除了使用error传递的方式，Go语言还为我们提供了抛出恐慌与捕捉并处理恐慌的错误处理方式，这需要使用两个内置函数panic与recover，原型如下。

```
func panic(v interface{})  
func recover() v interface{}
```

panic函数是抛出一个恐慌，当然不必被这个词吓到，一般代表碰到此类错误，事情比较严重了。panic的触发机制有两种，一种是运行时（runtime）错误时自动触发的，还有一种是人为调用的，当发生panic时，程序会从panic函数的位置立即返回，逐层打印堆栈信息，如果开发者不做特殊处理，进程将会退出。

panic恐慌发生时，如果不想退出，就可以使用recover来捕捉panic恐慌，只是recover使用却有一些小个性。例如下面这样使用，是捕获不到panic的。

```
/*  
    author:Yekai  
    company: Pdj  
    filename: 04-panic.go  
*/  
package main  
  
import (  
    "fmt"  
)  
  
func panic_recover() {  
    recover()  
    panic("game over") //抛出错误  
  
    fmt.Println("panic_recover run over")  
}  
  
func main() {
```

```

recover()
panic_recover()
fmt.Println("heihei,game not over!")
}

```

执行效果如下：

```

localhost:day02 yekai$ go run 04-panic.go
panic: game over

goroutine 1 [running]:
main.panic_recover()
    /Users/yekai/gowork/src/tutorial-go/threeday/day02/04-panic.go:14 +0x47
main.main()
    /Users/yekai/gowork/src/tutorial-go/threeday/day02/04-panic.go:21 +0x30
exit status 2

```

recover能够顺利捕获panic的第一个要求是必须在panic执行前先要延迟执行recover，所以下面需要为大家介绍一下defer关键字，defer是Go语言设计中非常温馨的一个关键字，它的好处令人发指。defer后可放置要执行的函数语句，该语句不会立即执行，而是会在其所在代码段运行结束后（正常结束、提前return、panic）触发该函数的执行。recover的另外一个要求是必须在延迟的函数体内执行才可以，因此recover是无论如何都要搭配defer了。示例代码如下。

```

/*
    author:Yekai
    company:Pdj
    filename:04-panic.go
*/
package main

import (
    "fmt"
)

func panic_recover() {
    //recover()
    defer func() { //延迟定义捕获
        fmt.Println("defer recover panic") //会打印
        recover()
    }()
    panic("game over") //抛出错误

    fmt.Println("panic_recover run over") //不会打印
}

func main() {
    //recover()
    panic_recover()
}

```



```
    fmt.Println("heihei,game not over!") //捕获成功则会打印本句
}
```

运行结果如下：

```
localhost:day02 yekai$ go run 04-panic.go
defer recover panic
heihei,game not over!
```

## 4.5 并发与同步的概念

并行与并发是两个不同的概念：

- 并行 在任何时间粒度下，程序都是同时运行的
- 并发 在规定的时间内，多个程序是同时运行的

看上去两者相差不大，实际不然，并发是给外界的感觉。就拿单核CPU来说，它永远也做不到并行，前辈们发明了分时操作系统，其目的就是为了支持并发，给外界的感觉是多个进程在同时运行。若想要支持并行，必须得是多核CPU的。

并发和并行都是为了提升应用运行效率。并行和并发是为达目的所采用的两个手段。并行通过提升CPU核心数，发则是通过系统设计（比如分时复用系统，多进程，多线程的开发方法）。并发的最大原因是为了更好的利用硬件资源，在对并发的支持上，Go语言是天然的，因为它设计的理念就是为并发而生。

## 4.6 并发编程

对于多进程、多线程这样的方式，并发时CPU需要在多个进程或线程间切换，这个切换成本是蛮高的，Go语言的优势在于其内部实现了并发调度算法，并且创造了一种更小的并发粒度执行单位go-routine（例程、协程），以后我们统一用goroutine。goroutine是比线程更小的运行单位，并发时goroutine没有上限的限制，可以随意启动，成千上万个也没问题。Go语言启动goroutine方面也非常容易，一个关键字go就够了。

```
go call_func() //启动goroutine
```

goroutine没有父子之分，开发者也不要预估goroutine们的执行先后次序，Go语言的运行时（runtime）为main函数分配一个main-goroutine，如果程序内有go关键字，那么会再启动其他goroutine，在goroutine内部也可以继续启动多个goroutine。启动goroutine的示例代码如下。

```
/*
    author:Yekai
    company:PdJ
    filename:05-goroutine.go
*/
package main

import (
    "fmt"
)
```

```
func main() {
    fmt.Println("begin call goroutine")
    //启动goroutine
    go func() {
        fmt.Println("I am a goroutine!")
    }()
    fmt.Println("end call goroutine")
}
```

执行上述代码，大概率你会看到的是下面的结果：

```
localhost:day02 yekai$ go run 05-goroutine.go
begin call goroutine
end call goroutine
```

看上去好像goroutine并没有启动成功，实际不然，goroutine启动成功了，但是主goroutine（main函数内）已经执行完退出了，此时进程整体退出，goroutine也就不存在了。如果我们想看到这句话“I am a goroutine!”，那就得想办法人为控制一下让主goroutine等待一下，最简单的办法是睡一觉，加一个Sleep就可以了。

```
time.Sleep(time.Second * 1) //睡1s
```

主函数在退出前执行time.Sleep(time.Second \* 1)可以确保goroutine能够顺利执行。

下面再来看一个计算斐波那契数列的例子，当计算数值较大时，程序执行时间会比较长，为了增加用户体验，让用户知道程序在为因计算而忙碌着，我们特意启动了一个spinner这个goroutine打印转圈圈。

```
/*
    author:Yekai
    company:Pdj
    filename:06-fib.go
*/
package main

import (
    "fmt"
    "time"
)

func main() {
    go spinner(time.Millisecond * 100) //启动一个打印goroutine
    fmt.Printf("\n%d\n", fib(45))
}

//计算斐波那契数列
func fib(x int) int {
    if x < 2 {
```

```

    return x
}
return fib(x-2) + fib(x-1)
}

//此函数目的只是为了用户体验
func spinner(delay time.Duration) {
    for {
        for _, r := range `-\|/` {
            fmt.Printf("\r%c", r)
            time.Sleep(delay)
        }
    }
}
}

```

感兴趣的读者可以把上述代码执行看看效果，再次强调，这里启动的spinner并不是为了提高斐波那契数列计算的速度。

## 4.7 Go语言运行时

所谓运行时（runtime），就是运行的时刻，Go语言的运行时用来描述与进程运行相关的信息，我们可以使用runtime包来显示一些运行时的信息。

### 4.7.1 GOMAXPROCS

```

func GOMAXPROCS(n int) int
//当n<=1时，查看当前进程可以并行的goroutine最大数量（CPU核心数）
//当n>1时，代表设置可并行的最大goroutine数量

```

示例代码如下。

```

/*
    author:Yekai
    company: Pdj
    filename: 06-runtime.go
*/
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println("Go Max proc =", runtime.GOMAXPROCS(0))
    runtime.GOMAXPROCS(2)
    fmt.Println("Go Max proc =", runtime.GOMAXPROCS(0))
}

```

执行结果如下：

```
localhost:day02 yekai$ go run 07-runtime.go
Go Max proc = 4
Go Max proc = 2
```

此外，也可以了解一下其他runtime包提供的函数。

- Goexit 退出调用的goroutine，退出前仍然会触发defer设定的延迟退出函数
- Gosched 让出本次goroutine的调度，很谦让的一种方式

## 4.8 Go语言同步

使用goroutine可以提高运行的效率，但是光追求效率是有问题的。goroutine之间的通信很简单，因为它们就在一个进程内，这一点和线程类似，但和线程一样，多个goroutine之间仍然需要协调，否则发生对同一个资源竞争的情况，就会出现数据混乱，比如都要修改同一个数据，谁先改谁后改是一个很严肃的问题！

我们知道，线程同步的方式无外乎mutex（互斥锁）、rwlock（读写锁）、cond（条件变量）、sem（信号量）等，在Go语言当中，同样对这些方式进行了支持，对于这些方式，大家可以查看sync包，里面的API就是支持上述这些同步方式的。

除了上述的同步方式，我会重点介绍Go语言自身最具特点同步方式，这种方式被称为CSP（Communicating Sequential Process）模型。两个goroutine之间可以通过channel进行传递消息，若没有接到消息或消息未被接受，发送和接收方都需要等待对方完成操作，这样也可以很精确的控制goroutine之间的同步协作。

在Go语言之中，需要使用内建函数make构造channel，chan是关键字，在构造时需要指定channel的数据类型，也可以指定数量（属于可选参数），如果有数量则表示channel有缓冲区，没有就是没缓冲区。

```
make(chan chantype)
make(chan chantype, 5)
```

channel的设计目的很简单，就是信息交换，类似于战争时期敌后工作者们的接头行为，两个goroutine就是要接头的两个人，两个人会阻塞等待（无缓冲区的情况）对方前来接头。至于消息是什么，可以是指令，也可以是有用的消息。对于channel，它类似于unix编程里的管道，在知道如何创建它后，接下来要搞懂它的读写行为。

- 写行为
  - 通道缓冲区已满（无缓冲区），写阻塞直到缓冲区有空间（或读端有读行为）
  - 通道缓冲区未满，顺利写入，结束
- 读行为
  - 缓冲区无数据（无缓冲区时写端未写数据），读阻塞直到写端有数据写入
  - 缓冲区有数据，顺利读数据，结束

我们来看一个例子：

```
/*
```

```

    author:Yekai
    company:PdJ
    filename:08-channel1.go
*/
package main

import (
    "fmt"
    "time"
)

var c chan string

func reader() {
    msg := <-c //读通道
    fmt.Println("I am reader,", msg)
}

func main() {
    c = make(chan string)
    go reader()
    fmt.Println("begin sleep")
    time.Sleep(time.Second * 3) //睡眠3s为了看执行效果
    c <- "hello" //写通道
    time.Sleep(time.Second * 1) //睡眠1s为了看执行效果
}

```

Go语言设计的通道读写真的是简单粗暴，区分是读还是写，只需看<-在通道左边还是右边即可。代码钟的两个Sleep都是为了显示效果而添加的。

接下来，我们用channel配合goroutine做一个游戏。需求是实现一个通过goroutine传递数字的游戏，goroutine1循环将1, 2, 3, 4, 5传递给goroutine2，goroutine2负责将数字平方后传递给goroutine3，goroutine3负责打印接收到的数字。

分析该需求，我们至少需要2个channel，3个goroutine，其中main函数可以直接是第三个goroutine，所以再创建2个就够了。

```

/*
    author:Yekai
    company:PdJ
    filename:09-channel2.go
*/
package main

import (
    "fmt"
    "time"
)

```

```

var c1 chan int
var c2 chan int

func main() {
    c1 = make(chan int)
    c2 = make(chan int)
    //counter
    go func() {
        for i := 0; i < 10; i++ {
            c1 <- i //向通道c1写入数据
            time.Sleep(time.Second * 1)
        }
    }()
    //squarer
    go func() {
        for {
            num := <-c1 //读c1数据
            c2 <- num * num //将平方写入c2
        }
    }()
    //printer
    for {
        num := <-c2
        fmt.Println(num)
    }
}

```

这样执行完效果不太好，因为当第一个goroutine循环结束后，由于没有goroutine再向通道写数据，这样就会出现错误而导致goroutine在死等。这种错误是可预见的，goroutine被锁死了，因此该错误被称为Deadlock（死锁）。

```
fatal error: all goroutines are asleep - deadlock!
```

这是由于channel的知识点我们尚未完全掌握，通道可以创建，也可以关闭，在读取的时候，也可以使用指示器变量来判断通道有没有关闭，顺便提一下range在这里仍然可以读取channel，此时不需要“<-”。我们来尝试结束后关闭channel，然后优雅地结束整个进程。

```

/*
    author:Yekai
    company:Pdj
    filename:09-channel2.go
*/
package main

import (
    "fmt"
    "time"

```

```

)

var c1 chan int
var c2 chan int

func main() {
    c1 = make(chan int)
    c2 = make(chan int)
    //counter
    go func() {
        for i := 0; i < 10; i++ {
            c1 <- i //向通道c1写入数据
            time.Sleep(time.Second * 1)
        }
        close(c1) //关闭c1
    }()
    //squarer
    go func() {
        for {
            num, ok := <-c1 //读c1数据
            if !ok {
                break
            }
            c2 <- num * num //将平方写入c2
        }
        close(c2) //关闭c2
    }()
    //printer
    for {
        num, ok := <-c2
        if !ok {
            break
        }
        fmt.Println(num)
    }
}

```

执行效果如下：

```
localhost:day02 yekai$ go run 09-channel2.go
0
1
4
9
16
25
36
49
64
81
```

特别注意，对通道的读写操作都会使goroutine阻塞，通道的关闭应该由写端来操作。此外，channel也可以作为函数参数，默认情况下一个channel读写皆可，为了防止不该写的goroutine发生写行为，Go语言设计了channel传递给函数的时候可以指定为单方向，读或者写！而这个单方向表述非常明确：

```
chan_name chan<- chan_type //只写通道
chan_name <-chan chan_type //只读通道
```

我们将上述的例子改造，因为三个goroutine对某个channel的操作都是单方向的。

```
//counter, 对c1只写
go func(out chan<- int) {
    for i := 0; i < 10; i++ {
        out <- i //向通道c1写入数据
        time.Sleep(time.Second * 1)
    }
    close(out)
}(c1)
//squarer, 对c1只读, 对c2只写
go func(in <-chan int, out chan<- int) {
    for {
        num, ok := <-in //读c1数据
        if !ok {
            break
        }
        out <- num * num //将平方写入c2
    }
    close(out)
}(c1, c2)
```

## 4.9 定时器



接下来，我们再来实现一个火箭发射的例子，准备倒数计时5秒，然后打印一个发射火箭。当然可以用Sleep来控制每隔1s计数一次，在这里我们将使用Go语言为我们提供的定时器来做这件事，定时器的内部实现靠的也是channel。

```
//创建定时器
func NewTimer(d Duration) *Timer
//定时器结构体
type Timer struct {
    C <-chan Time
    r runtimeTimer
}
//定时器停止
func (t *Timer) Stop() bool
```

在time包中存在一个NewTimer，传入一个时间间隔n，获得一个Timer，Timer结构体中包含了一个C，这是一个通道类型，于是在时间n之后，C中会被写入时间戳。

```
timer := time.NewTimer(time.Second * 1)
data := <-timer.C
fmt.Println(data)
timer.Stop() //停止定时器
```

timer只是一次性事件，不足以满足我们的需求。我们再来介绍另一款周期性定时器Ticker。

```
func NewTicker(d Duration) *Ticker
type Ticker struct {
    C <-chan Time // The channel on which the ticks are delivered.
    r runtimeTimer
}
```

周期性ticker，示例如下：

```
//创建周期性定时器
ticker := time.NewTicker(time.Second)
num := 5
for {
    data := <-ticker.C
    fmt.Println(data)
    num--
    if num == 0 {
        break
    }
}
ticker.Stop()
```

接下来我们可以将考虑实现火箭发射的需求，定时器已经有了，可以来尝试。

```

/*
    author:Yekai
    company:Pdj
    filename:12-ticker.go
*/
package main

import (
    "fmt"
    "time"
)

func launch() {
    fmt.Println("发射!")
}

func main() {
    ticker := time.NewTicker(time.Second)
    num := 5
    for {
        <-ticker.C //读取无人接收
        fmt.Println(num)
        num--
        if num == 0 {
            break
        }
    }
    ticker.Stop()
    launch() //发射火箭
}

```

这样可以实现火箭发射的功能，不过如果临时想取消发射，该如何做呢？按ctrl+c的方式太简单粗暴了一下，比如想要按下任意键取消发射呢？继续往下看！

## 4.10 多路channel监控

我们很自然想到读标准输入就可以了，甚至也会立刻想到启动一个goroutine去监听标准输入，如果有输入，立即退出进程。监控标准输入的代码可以是下面的样子，os.Stdin是标准输入流对象。

```

func cancel() {
    data := make([]byte, 10)
    os.Stdin.Read(data) //读标准输入
    os.Exit(1)          //退出进程
}

```

可以实现这样一个函数，读取标准输入，但是就这样通过os.Exit(1)退出整个进程不是太优雅，我们还是希望比较稳妥的退出。于是很多人想到，是否可以在建立一个channel，当标准输入有数据的时候，将数据写入该channel，在main函数中监控该channel，如果读到数据，则不执行后面的发射，直接return。

这是一个好办法，但是问题来了，通道读是阻塞的，我们已经在监控一个定时器channel了，同时监控两个channel将会有问题。在linux系统下，多路IO监控可以使用select、poll、epoll等，在Go语言里，同样提供了一个机制可以监控多路channel，这个机制就是select-case语句。

```
for {
    select {
    case <-ticker.C:
        fmt.Println(num)
        num--
    case <-chan_stdin:
        fmt.Println("取消发射！")
        return
    }
    if num == 0 {
        ticker.Stop()
        break
    }
}
```

select 可以监控多个通道，当任一通道有数据写入时，select都会立即返回解除阻塞。完整代码如下：

```
/*
    author:Yekai
    company: Pdj
    filename: 13-select.go
*/
package main

import (
    "fmt"
    "os"
    "time"
)

func launch() {
    fmt.Println("发射！")
}

func main() {
    ticker := time.NewTicker(time.Second)
    fmt.Println("开始倒计时准备发射，按回键可以取消发射！")
    num := 5
    chan_stdin := make(chan string)
```

```

go func(out chan<- string) {
    data := make([]byte, 10)
    os.Stdin.Read(data) //该goroutine读也会阻塞
    out <- "cancel"
}(chan_stdin)
for {
    select {
    case <-ticker.C:
        fmt.Println(num)
        num--
    case <-chan_stdin:
        fmt.Println("取消发射! ")
        return
    }
    if num == 0 {
        ticker.Stop()
        break
    }
}

launch() //发射火箭
}

```

此时，可以在倒数计时期间按下回车键取消发射。

## 5 网络编程

Go语言在网络编程方面优势明显，开发者可以不用知道套接字等概念，直接使用Go语言封装的net包，就可以非常便利的进行网络编程开发，可以选择TCP，也可以选择HTTP。

### 5.1 文件IO

前面的例子我们已经涉及到标准输入读取的事情，而在网络编程中，一个基础的技能就是要掌握IO，也就是读写的能力，所以在进入网络编程前，先来介绍一下Go语言的文件IO操作。

说到文件IO，就要提到一个文件描述符的概念。这个概念比较抽象，按字面理解就是描述文件信息的一个符号，你可以通过这个描述符得到文件的各方面信息，同时也可以通过这个描述符去操作文件，读、写、改变文件属性等等。在linux系统中，每个进程内有一个文件描述符表，这个表简单理解的话就是记录一个文件描述符的编号，每个进程默认有1024个文件描述符可以使用，前0，1，2号已经被提前占用，分别对应标准输入（stdin），标准输出（stdout）以及标准错误（stderr），这也是为什么我们之前可以直接读取标准输入的原因。

接下来我们的目标是用Go语言创建一个文件，并且写入：“hello,yekai”。为了完全这个需求，需要去提前了解一下相关API。

```

//打开或创建文件
func OpenFile(name string, flag int, perm FileMode) (*File, error)

```

参数说明：

- name 文件名称
- flag 标志位 O\_RDONLY 或 O\_WRONLY 等
- perm 文件权限 8 禁制的权限表示法
- 返回值 返回文件描述符以及错误信息

当我们成功打开文件后，就可以得到文件描述符File，之后可以利用它对文件进行读写操作，以及关闭文件。

```
func (f *File) Read(b []byte) (n int, err error)
func (f *File) Write(b []byte) (n int, err error)
func (f *File) WriteString(s string) (n int, err error)
func (f *File) Close() error
```

下面，来尝试创建并写入文件。

```
/*
    author:Yekai
    company:Pdj
    filename:14-io.go
*/
package main

import (
    "fmt"
    "os"
)

func main() {
    fd, err := os.OpenFile("a.txt", os.O_WRONLY|os.O_CREATE, 0666)
    if err != nil {
        fmt.Println("failed to openfile ", err)
        return
    }
    defer fd.Close() //延迟关闭
    fd.WriteString("hello,yekai\n")
}
```

在flag参数位置，传递了写标志以及创建标志（os.O\_CREATE），对于新文件，必须要有创建标志。在文件读写方面，我们需要知道文件读写是有位置的，默认打开都是从头开始，这个位置我们可以调整，参看：

```
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

具体例子就由大家自己来尝试。除了操作文件，Go对于目录的操作也很便利。其实在linux系统下，一切皆文件，目录对于系统来说只是一种特殊的文件类型而已。

```
func (f *File) Readdir(n int) ([]FileInfo, error)
```

看到readdir这个函数，不难发现它也是File的一个方法，所以要读目录，还是要先获得File类型的指针，我们来尝试一下读取当前目录下的所有文件信息。Readdir返回一个切片类型，对切片进行遍历也就拿到了目录下所有内容，示例代码如下。

```
/*
    author:Yekai
    company:Pdj
    filename:15-readdir.go
*/
package main

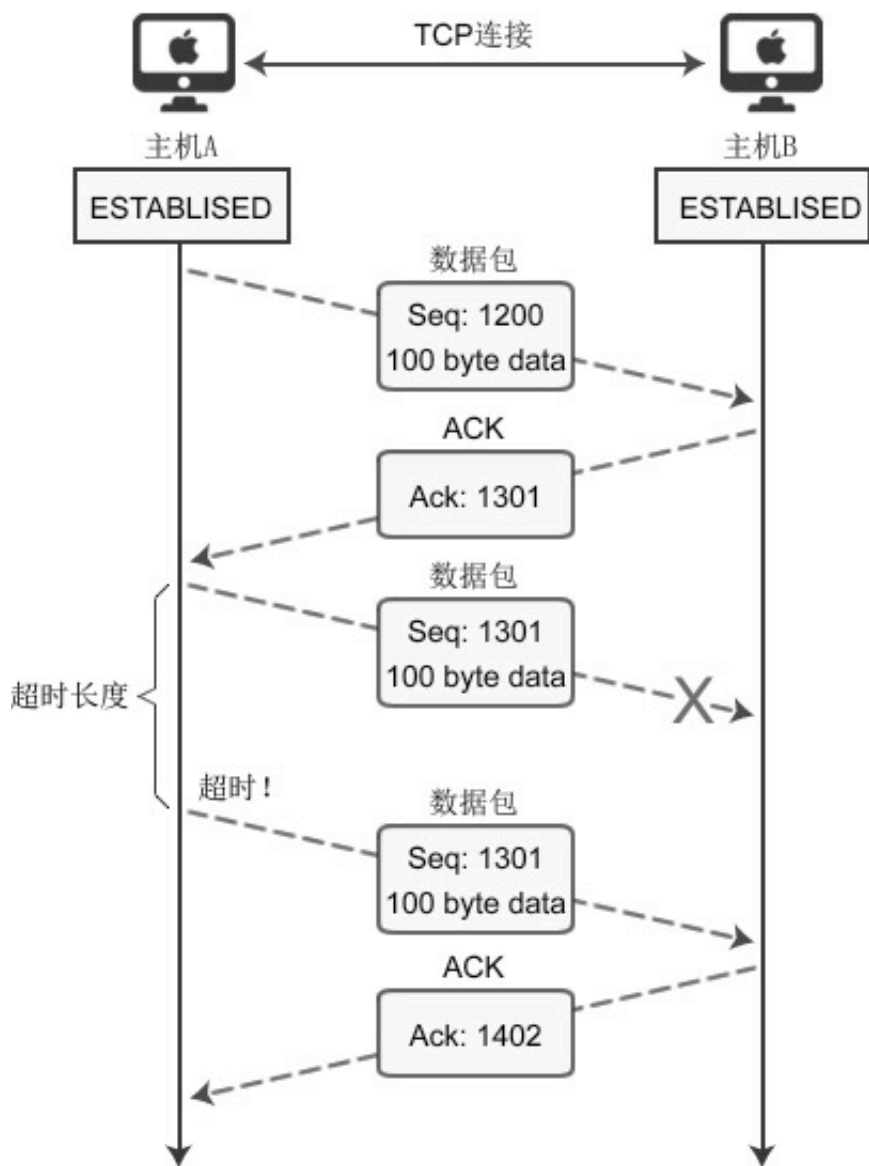
import (
    "fmt"
    "os"
)

func main() {
    fd, err := os.Open("./") //打开当前目录
    if err != nil {
        fmt.Println("failed to openfile ", err)
        return
    }
    //参数<0 代表要读取全部，大于0也是代表要读取部分
    infos, err := fd.Readdir(-1)
    if err != nil {
        fmt.Println("failed to Readdir ", err)
        return
    }
    //遍历读取目录的结果
    for _, v := range infos {
        fmt.Printf("name = %s, isdir = %t, size = %d\n", v.Name(), v.IsDir(),
v.Size())
    }
}
```

## 5.2 TCP编程

在网络协议中，最著名的是IOS标准的七层协议，不过在使用上最为广泛的是TCP/IP四层协议，对应的四层分别是数据链路层、网络层、传输层以及应用层。网络编程解决的是两个进程间的通信问题，面向的场景是两个进程不在同一主机的情况。TCP/IP协议规定两个进程要想建立连接需要通过三次握手建立连接，之后可以彼此传输消息，若要断开连接，需要四次握手（挥手）。

TCP通信的核心思想是数据包的封装与拆包，在发送消息端的进程，将目标发送消息通过网络分层时逐层封包，最终在底层网络传输给对方，而对方收到消息后，再由底层到上层逐层解包，拿到对方发送的实际消息。虽然听上去很复杂，好在这个过程开发者不需要考虑，这是在协议层定义、系统层支持的。



TCP/IP协议层最关键的定义是IP和端口，IP可以在一个网络内唯一的标识一台主机，而端口则是在一个主机上可以唯一的标识一个进程，IP+端口的方式就可以确定唯一一个主机上的唯一一个进程。先来说IP，目前网络编程多使用的是IPv4协议，ip地址是4组0-255的数字组合（这是人类可识别的表示方法），在网络编程中对应的其实就是0或1的数字。再来说说端口，端口是一个短整型的数字，它的范围是0-65535，需要注意的是有些端口已经被占用，并且形成了约定俗成的规则，比如80端口就是web服务器使用的端口，还比如mysql用的端口一般是3306，MongoDB使用的端口默认是27017等。

在Go语言当中，理论知道当然最好，但是开发的时候只是使用几个API就够了，因为其内部对网络进行了很好的封装。建立一个TCP服务器的基本步骤是这样的：

1. 绑定IP和端口
2. 建立侦听
3. 等待侦听结果，得到新连接
4. 与得到的新连接进行通信

在写服务器的时候，借助net包可以很轻易的完成：

```
func Listen(network, address string) (Listener, error) //建立一个侦听器
```

这个函数在实现的时候，帮我们做了ip和端口的绑定，同时开始侦听。其中network是协议类型，可以传tcp，address是网络地址，它的格式是“ip:port”，如果针对本机任何ip都侦听，ip可以省略。Listener是Listen的返回值，它是一个接口类型，结构如下。

```
type Listener interface {
    // Accept waits for and returns the next connection to the listener.
    Accept() (Conn, error)

    // Close closes the listener.
    // Any blocked Accept operations will be unblocked and return errors.
    Close() error

    // Addr returns the listener's network address.
    Addr() Addr
}
```

得到的Listener是一个接口类型，它有一个Close方法和Accept方法，Accept方法用来获得新连接，在得到新连接后，可以与它进行通信。得到的conn也是一个接口类型，结构如下。

```
type Conn interface {
    // Read reads data from the connection.
    // Read can be made to time out and return an Error with Timeout() == true
    // after a fixed time limit; see SetDeadline and SetReadDeadline.
    Read(b []byte) (n int, err error)

    // Write writes data to the connection.
    // Write can be made to time out and return an Error with Timeout() == true
    // after a fixed time limit; see SetDeadline and SetWriteDeadline.
    Write(b []byte) (n int, err error)

    // Close closes the connection.
    // Any blocked Read or Write operations will be unblocked and return errors.
    Close() error

    // LocalAddr returns the local network address.
    LocalAddr() Addr

    // RemoteAddr returns the remote network address.
    RemoteAddr() Addr

    .....
}
```

所以接下来的事情就是读写操作了！我们用上述的API来实现一个回射服务器，服务器做的事情就是客户端不管发来什么，都发回去什么！

```
/*
```



```

author:Yekai
company:PdJ
filename:01-tcp_server.go
*/
package main

import (
    "fmt"
    "io"
    "log"
    "net"
)

func main() {
    //1. 指定为ipv4协议, 绑定IP和端口, 启动侦听
    listener, err := net.Listen("tcp", "localhost:8888")
    if err != nil {
        log.Panic("Failed to Listen", err) //输出错误信息, 并且执行panic错误
    }
    defer listener.Close() //收尾工作
    for {
        //2. 循环等待新连接到来, Accept阻塞等待状态
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Failed to Accept ", err)
            continue
        }
        fmt.Println("New conn->", conn.RemoteAddr().String())
        //3. 启动匿名函数来处理
        go func(conn net.Conn) {
            defer conn.Close() //收尾工作
            buf := make([]byte, 256)
            for {
                //从客户端读数据
                n, err := conn.Read(buf)
                if err != nil {
                    if err == io.EOF { //这种错误代表客户端先关闭了, 属于正常范围内的错误
                        fmt.Println("Client ", conn.RemoteAddr().String(), " is Closed")
                        break
                    }
                }
                fmt.Println("Failed to Read data ", err)
                break
            }
            //回射服务器, 收到什么, 写回什么
            n, err = conn.Write(buf[:n])
            if err != nil {
                fmt.Println("Failed to Write to client ",
conn.RemoteAddr().String(), err)
                break
            }
        }(conn)
    }
}

```

```

    }
}

}(conn)
}
}

```

把这个服务器启动后，就可以等待客户端来连接了，正常情况我们应该自己写一个客户端程序。但我比较懒，在unix平台，可以借助一个nc（net connect）命令来测试服务器情况，新打开一个终端，如下启动即可：

```

localhost:day01 yekai$ nc 127.0.0.1 8888
haha
haha
heiohei
heiohei

```

终止这个nc客户端可以使用ctrl+c，我们输入一个字符串，得到是一个相同的字符串，这就是服务器端做的工作。

在了解完net相关的API之后，我们尝试编写一个并发聊天室打发无聊的时间，需求如下：

1. 支持多个客户端
2. 支持广播和私信
3. 支持设置昵称
4. 上线和下线广播

这个需求的实现，涵盖了之前介绍的全部知识点。分析如下：

1. 多个客户端，需要用到并发
2. 要支持广播，需要记录多个客户端连接信息，可以使用map
3. 多个goroutine之间通信，要用到channel
4. 网络编程tcp知识
5. 客户端数据交换，json数据格式处理

具体实现如下，先定义一些消息结构

```

//goroutine之间传递消息的结构
type ChatMsg struct {
    From, To, Msg string
}

//服务器端与客户端传递消息格式：json
type ClientMsg struct {
    To      string `json:"to"`      //接收者
    MsgType int    `json:"msgtype"` //消息类型
    Msg     string `json:"msg"`     //实际消息
    Datalen uintptr `json:"datalen"` //消息长度 校验用
}

```

```

//消息中心的通道
var chan_msgcenter chan ChatMsg

//定义连接列表
var mapClients map[string]net.Conn //key->conn
var keyClients map[string]string   // name ->key ; key->conn

```

消息中心的功能是接收各个goroutine的通知，选择广播还是私信给其他客户端。

```

func main() {

    //初始化
    chan_msgcenter = make(chan ChatMsg)
    mapClients = make(map[string]net.Conn)
    keyClients = make(map[string]string)

    // 设置侦听
    listener, err := net.Listen("tcp", "localhost:8888")
    if err != nil {
        log.Panic("Failed to Listen ", err)
    }
    //延迟关闭
    defer listener.Close()
    //启动消息中心处理
    go msg_center()
    //循环接收各个客户端连接
    for {
        //接收客户端连接请求
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Failed to Accept ", err)
            continue
        }
        //处理各个客户端的请求 goroutine
        go handle_conn(conn)
    }
}

```

handle\_conn 是处理客户端连接的逻辑，这里主要就是连接，监听是否断开，还有正常的读取客户端请求。连接时可以认为是客户端上线，断开时代表客户端下线，剩下的就是客户端发过来的正常请求，可以有修改名字，私信或者广播消息等。

```

//断开连接处理
func logout(conn net.Conn, from string) {
    //关闭连接
    defer conn.Close()
}

```

```

//组织一个消息发给消息中心，广播的形式
msg := ChatMsg{from, "all", from + "->logout"}
chan_msgcenter <- msg
delete(mapClients, from)
}

// 处理请求的函数
func handle_conn(conn net.Conn) {
    //4. 善后工作
    //1. 拿到客户端信息
    from := conn.RemoteAddr().String() //ip:port
    defer logout(conn, from)

    //2. 放到map中
    mapClients[from] = conn
    //2.1 上线通知
    msg := ChatMsg{from, "all", from + "->login"}
    chan_msgcenter <- msg
    //3. 处理读写行为
    buf := make([]byte, 256)
    for {
        n, err := conn.Read(buf)
        if err != nil {
            fmt.Println("Failed to Read data ", err)
            break
        }
        //处理客户端的消息请求
        var climsg ClientMsg
        err = json.Unmarshal(buf[:n], &climsg)
        if err != nil {
            fmt.Println("Failed to Unmarshal ", err)
            continue //死缓
        }
        if climsg.Datalen != unsafe.Sizeof(climsg) {
            fmt.Println("msg format err ", climsg)
            continue //死缓
        }
        fmt.Println(climsg)
        //组织一个消息
        from := conn.RemoteAddr().String()
        chatmsg := ChatMsg{from, "all", climsg.Msg}
        switch climsg.MsgType {
        case SETNAME:
            //1.名字存在哪 2. 名字怎么用(聊天的时候用) - > 通过名字可以找到连接信息
            //fmt.Println("setname-----", msg)
            keyClients[climsg.Msg] = from
            chatmsg.Msg = from + " set name=" + climsg.Msg + " success"
            chatmsg.From = "server" //发送者改为服务器

```

```

    case BROADCAST:
    case PRIVATE:
        chatmsg.To = climsg.To
    }
    //发送到消息中心
    chan_msgcenter <- chatmsg

}

}

```

SETNAME 使用了之前介绍的枚举定义方法：

```

//消息类型
const (
    LOGIN = iota //iota 让枚举更优雅
    LOGOUT
    SETNAME
    BROADCAST
    PRIVATE
)

```

接下来还需要处理消息中心的逻辑，消息中心主要接收各个客户端处理goroutine的channel消息，可以是上线通知，下线通知，以及正常的广播和私信消息。

```

//消息处理中心
func msg_center() {
    for {
        msg := <-chan_msgcenter //读消息中心的channel
        //发送：广播或私信
        fmt.Println(msg)
        go send_msg(msg)
    }
}

func send_msg(msg ChatMsg) {
    data, err := json.Marshal(msg)
    if err != nil {
        fmt.Println("Failed to Marshal ", err)
        return
    }
    if msg.To == "all" {
        //广播
        //遍历map 发送
        for _, v := range mapClients {
            //不给自己发
            if msg.From != v.RemoteAddr().String() {
                v.Write(data)
            }
        }
    }
}

```

```

    }
}
} else {
    //私信
    from, ok := keyClients[msg.To]
    if !ok {
        fmt.Println("User not exists ", msg.To)
        return
    }
    conn, ok := mapClients[from]
    if !ok {
        fmt.Println("client not exists ", from)
        return
    }
    conn.Write(data)
}
}

```

由于传输消息是我们自定义的，所以只能自己实现一个客户端了（nc命令无法支持特定数据包格式），参考代码如下：

```

package main

import (
    "bufio"
    "encoding/json"
    "fmt"
    "log"
    "net"
    "os"
    "strings"
    "unsafe"
)

//服务器端与客户端传递消息格式：json
type ClientMsg struct {
    To      string `json:"to"`      //接收者
    MsgType int    `json:"msgtype"` //消息类型
    Msg     string `json:"msg"`     //实际消息
    Datalen uintptr `json:"datalen"` //消息长度 校验用
}

//消息类型
const (
    LOGIN = iota //iota 让枚举更优雅
    LOGOUT
    SETNAME
    BROADCAST

```

```

PRIVATE
)

func Help() {
    fmt.Println("1. set:your name")
    fmt.Println("2. all:your msg")
    fmt.Println("3. anyone:your msg")
}

func handle_conn(conn net.Conn) {
    buf := make([]byte, 256)
    for {
        n, err := conn.Read(buf)
        if err != nil || n <= 0 {
            log.Panic("Failed to Read", err)
        }
        fmt.Println(string(buf[:n]))
        fmt.Printf("pdj's chat>")
    }
}

func main() {
    Help()
    conn, err := net.Dial("tcp", "localhost:8888")
    if err != nil {
        log.Panic("Failed to Dial ", err)
    }
    defer conn.Close()
    //客户端的逻辑: 1. 读标准输入, 发给服务器 2. 读服务器, 响应在屏幕
    go handle_conn(conn)
    // 处理读标准输入-》发武器
    fmt.Println("Welcome to pdj's chat")
    reader := bufio.NewReader(os.Stdin)
    for {
        fmt.Printf("pdj's chat>") //打印一个输入端
        msg, err := reader.ReadString('\n') //读取一行消息
        if err != nil {
            log.Panic("Failed to ReadString ", err)
        }
        msg = strings.Trim(msg, "\r\n") //去回车换行
        if msg == "quit" { //提前给客户端一个友好退出口
            fmt.Println("Bye bye")
            break
        }
        msgs := strings.Split(msg, ":") //客户端的消息格式定义关键 to:msg
        if len(msgs) == 2 {
            msgtype := PRIVATE
            switch msgs[0] {
                case "set":

```

```

        msgtype = SETNAME
    case "all":
        msgtype = BROADCAST
    }
    var climsg ClientMsg
    climsg.MsgType = msgtype
    climsg.To = msgs[0]
    climsg.Msg = msgs[1]
    climsg.Datalen = unsafe.Sizeof(climsg)
    data, _ := json.Marshal(climsg)
    conn.Write(data) //写给服务器
}
}
}

```

关键逻辑是客户端每次发的消息都要构造成ClientMsg结构发送给对方，处理标准输入的数据把他们变成：“to:msg”，当想要修改昵称的时候，to使用set即可。

## 5.3 HTTP编程

HTTP协议（超文本传输协议）是TCP/IP分层模型中应用层的协议，也是浏览器与web服务器之间的通信协议。它的特点侧重于表示，简单点说就是客户端发过来一个请求，通过协议的方式描述出具体的内容，比如客户端是什么样子（什么浏览器），请求的资源是什么等等。当然服务器端传给客户端的消息也要遵循HTTP协议，描述响应码，资源类型，内容长度，资源信息等。无论是请求还是发送，都是由一系列键值对组成。

如果想看一下HTTP协议到底有怎么样的效果，可以写一个tcp服务器，然后在浏览器请求一下，就可以见识到http的请求是什么样了。

```

/*
    author:Yekai
    company:Pdj
    filename:02-http.go
*/
package main

import (
    "fmt"
    "net"
)

func main() {
    listener, _ := net.Listen("tcp", ":8080")
    for {
        conn, _ := listener.Accept()
        go func(conn net.Conn) {
            buf := make([]byte, 2048)
            conn.Read(buf)

```

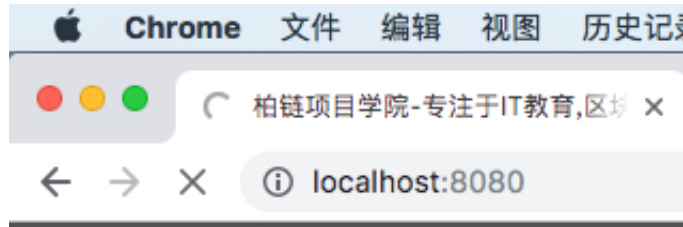


```

    fmt.Println(string(buf))
}(conn)
}
}

```

该服务器运行后，用浏览器请求，就可以看到浏览器端发过来什么消息了。



```

localhost:day03 yekai$ go run 02-http.go
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/75.0.3770.142 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9

```

请求消息中，第一行最为关键，它告诉我们浏览器请求的方法是什么，请求的资源是什么，以及所用的HTTP协议版本号是多少。至于请求的方法可以有：GET、POST、PUT、DELETE等方法，请求的资源“/”代表请求的是服务器的根目录，这里有个专业的名词叫URL（专业点说法叫统一资源定位符），表示资源所在的具体路径。User-Agent表达的是浏览器信息，在上述的例子我们看到请求的浏览器是chrome。

在Go语言当中，想实现一个HTTP的服务器其实很容易。了解一下相关API后就可以很容易搞定。

```

func ListenAndServe(addr string, handler Handler) error //启动侦听并提供web服务

```

在此之前需要设置好handle路由器，对于不同的http请求，给予不同的响应。

```

func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) //路由函数

```

HandleFunc用来设置当pattern请求到来的时候，使用后面的handler函数来处理。这个是一个简单的正则规则，比如我们想让所有hello相关的请求可以使用类似这样的url：/hello/username，对于这样的请求，我们给予的响应是：hello, username，我们可以这样设置路由：

```
http.HandleFunc("/hello/", HelloUserServer)
```

HelloUserServer是需要开发者实现的，原型必须是handler所定义的。

```
func HelloUserServer(w http.ResponseWriter, req *http.Request) {

    path := req.URL.Path //得到请求的url
    fmt.Println(path)
    users := strings.Split(path, "/")
    fmt.Println(len(users), users)
    if len(users) == 3 {
        io.WriteString(w, users[1]+","+users[2]) //组织一个响应消息
    }

}
```

整体可以看如下代码，代码中又增加了一个byebye的路由函数：

```
/*
    author:Yekai
    company:Pdj
    filename:03-http-server.go
*/
package main

import (
    "fmt"
    "io"
    "log"
    "net/http"
    "strings"
)

//router hello
func HelloUserServer(w http.ResponseWriter, req *http.Request) {

    path := req.URL.Path
    fmt.Println(path)
    users := strings.Split(path, "/")
    fmt.Println(len(users), users)
    if len(users) == 3 {
        io.WriteString(w, users[1]+","+users[2])
    }

}

//router byebye
func ByeUserServer(w http.ResponseWriter, req *http.Request) {
```

```

path := req.URL.Path
users := strings.Split(path, "/")
if len(users) == 3 {
    io.WriteString(w, users[1]+" "+users[2])
}

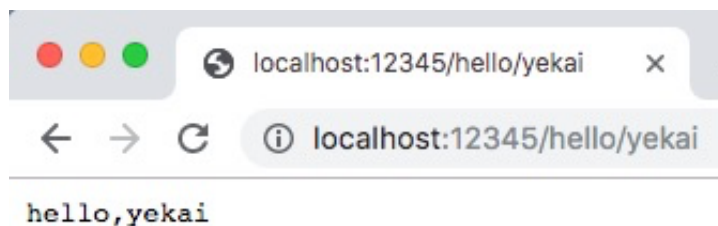
}

func main() {
    //设置hello的路由
    http.HandleFunc("/hello/", HelloUserServer)
    //设置byebye的路由
    http.HandleFunc("/bye/", ByeUserServer)
    //侦听并提供web服务，所有的事情都在前面设置的路由函数实现了
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}

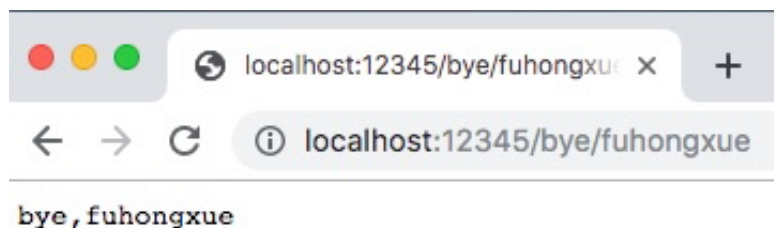
```

将服务器启动后，在浏览器针对不同的输入可以看到不同的结果：

浏览器输入：<http://localhost:12345/hello/yekai>可以看到下图的效果



浏览器输入：<http://localhost:12345/bye/fuhongxue>可以看到下图的效果。



Go语言官方提供的http包很强大，编写web服务器的核心也就是设置路由以及服务函数的编写。大多数时候，我们编写一个web服务器并非借助官方库，而是使用第三方的web框架，比如gin，echo，level，beego等，感兴趣的读者可以去了解一下。

## 6 Go语言工程管理

### 6.1 Go语言的目录

- GOROOT 安装目录

- GOPATH 工作目录

— bin	编译安装的二进制可执行文件目录
— pkg	编译源码得到的静态库文件 (.a) 存放目录
— src	工程源码所在目录

在GOPATH目录下，bin和pkg我们大可不必太多关注，重点放在src目录即可。开发者自己的工程可以存放在\$(GOPATH)/src下，第三方Go语言包，我们在使用“go get”进行安装时，也会存放在此目录下。例如当我执行下面的指令后，就会在src目录下看到对应的工程文件目录。

```
go get -u github.com/boltdb/bolt
```

执行成功后，在\$GOPATH/src/github.com/boltdb/bolt下会看到响应的源码文件，在我们的工程中可以直接进行使用，使用方式如下：

```
github.com/boltdb/bolt
```

对于有些特殊的包，使用起来略有特殊，比如mysql提供的API库，我们在引用的时候前面需要加一个“\_”，代表匿名引用，因为官方标准包database/sql需要使用到github.com/go-sql-driver/mysql包的驱动内容，但不需要显示的引用包，因此在引用的时候，应该如下操作：

```
import _ "github.com/go-sql-driver/mysql"
```

## 6.2 源码包的自定义和使用

### 6.2.1 init函数

源码包可以是官方提供的，也可以是第三方的，当然我们自己也可以编写。接下来我们介绍源码包的一些规则，首先来说一下init函数。init函数在每个源码包文件中都可以定义，init函数的执行逻辑是这样的：

1. main包检查包含的所有依赖包
2. 查找所有依赖包的依赖包直至没有依赖为止
3. 从末梢开始执行所有的依赖包内的init函数（多次包含，只执行一次）
4. 最后执行main包内的init函数
5. 如果包内没有init函数，则忽略

为了看到效果，可下载本人提供的示例：

```
go get -u github.com/yekai1003/tutorial
```

它的目录结构也很简单，包含两个目录，每个目录下都有一个文件，分别定义了mathdemo包和pkgdemo包。

```
localhost:yekai1003 yekai$ tree tutorial/  
tutorial/  
├── mathdemo  
│   └── mathdemo.go  
└── pkgdemo  
    └── pkgdemo.go
```

mathdemo.go代码如下:

```
/*  
    author:Yekai  
    company:Pdj  
    filename:mathdemo.go  
*/  
package mathdemo  
  
import (  
    "fmt"  
)  
  
func init() {  
    fmt.Println("mathdemo'init is called")  
}
```

pkgdemo.go代码如下:

```
/*  
    author:Yekai  
    company:Pdj  
    filename:pkgdemo.go  
*/  
package pkgdemo  
  
import (  
    "fmt"  
  
    _ "github.com/yekai1003/tutorial/mathdemo"  
)  
  
//外部可导出结构体  
type ExternalPerson struct {  
    Name string //大写, 可导出  
    Age  int  
    sex  string //小写, 不可导出  
}  
  
//内部不可导出结构体  
type internalPerson struct {
```

```

    Name string
    Age  int
    sex  string
}

func init() {
    fmt.Println("pkgdemo'init is called")
}

//外部可导出函数
func NewPerson(name string, age int, sex string) *ExternalPerson {
    return &ExternalPerson{name, age, sex}
}

func NewInternalPerson(name string, age int, sex string) *internalPerson {
    return &internalPerson{name, age, sex}
}

```

在pkgdemo.go中引用了mathdemo包，接下来我们再写一个init函数调用的例子：

```

/*
    author:Yekai
    company:PdJ
    filename:04-init.go
*/
package main

import (
    "fmt"

    _ "github.com/yekai1003/tutorial/mathdemo"
    _ "github.com/yekai1003/tutorial/pkgdemo"
)

func init() {
    fmt.Println("main init is called")
}

func main() {
}

```

运行之后，可以看到这样的效果：

```

localhost:day03 yekai$ go run 04-init.go
mathdemo'init is called
pkgdemo'init is called
main init is called

```

由此可以看出init函数被调用的顺序，mathdemo虽然被包含了两次，但只执行了一次。

## 6.2.2 包的引用和导出

我们之前在使用包的时候已经注意到了，如果包名为：xa/xb/xc，那我们在代码里最终使用的是xc，这个xc也是go源码文件里的包名。

包的设计目标就是方便其他人使用，那么包内肯定希望提供一些第三方使用的方法、变量或结构。在设计包的时候，需要遵循以下准则：

1. 如果函数希望外部可使用，首字母必须大写
2. 如果结构希望外部可使用，首字母必须大写
3. 如果结构体内字段外部可使用，首字母必须大写
4. 如果结构体方法希望外部可使用，首字母必须大写

简单来说就是一句话，如果希望外部能访问，首字母一定大写。当然反过来，如果不希望外部访问，那么变量、结构体、函数、方法等的名字首字母就应该小写，这代表对外隐藏内部实现。

下面代码会对导出函数和结构进行测试，因为结构体中的sex首字母并非大写，所以无法导出，外部无法直接访问。

```
/*
    author:Yekai
    company:Pdj
    filename:05-export.go
*/
package main

import (
    "fmt"

    "github.com/yekail003/tutorial/pkgdemo"
)

func init() {
    fmt.Println("main init is called")
}

func main() {
    //这样会报错，sex字段非导出，不能填写
    p1 := pkgdemo.ExternalPerson{"yekai", 40, "man"}
    fmt.Println(p1)
    //这样没有问题，NewPerson是可以导出的
    p2 := pkgdemo.NewPerson("fuhongxue", 37, "man")
    fmt.Println(p2)
}
```

执行的时候，会看到如下效果：

```
localhost:day03 yekai$ go run 05-export.go
# command-line-arguments
./05-export.go:19:44: implicit assignment of unexported field 'sex' in
pkgdemo.ExternalPerson literal
```

## 6.3 gomodule

Go语言是基于工程管理的语言，但是对于工程依赖的问题在较早版本时一直为人所诟病。在1.11版本之后，Go语言推出了gomodule模式这一工程管理方法，这给广大开发者带来了极大的便利。

gomodule模式不再要求工程目录建立在\$(GOPATH)/src下，而且最好不要放在该目录下。使用gomodule模式时，开发者不必手动去下载第三方依赖包，编译时会自动下载相关的依赖包。此外，gomodule还有一个好处是很多依赖包会处于外网中，使用go get无法直接下载，通过gomodule设置代理，可以完美解决该问题。

gomodule设置方式如下（参考：<https://goproxy.cn/>）：

### macOS 或 Linux

打开你的终端并执行

```
$ export G0111MODULE=on
$ export GOPROXY=https://goproxy.cn
```

或者

```
$ echo "export G0111MODULE=on" >> ~/.profile
$ echo "export GOPROXY=https://goproxy.cn" >> ~/.profile
$ source ~/.profile
```

完成。

### Windows

打开你的 PowerShell 并执行

```
C:\> $env:G0111MODULE = "on"
C:\> $env:GOPROXY = "https://goproxy.cn"
```

那么gomodule怎么使用呢？其实也挺简单的。

一般操作步骤有2步：



1. 初始化：go mod init 工程名
2. 编译或运行即可

初始化时，工程目录内会产生go.mod文件，在编译时该文件会因为依赖包的版本发生变化，如果工程内的代码依赖某一特定版本的第三方库，可以手动修改代码版本。

赶快去体验一下go mod吧！

## 7 总结

---

Go语言是一门语法非常简洁，极易上手的语言，它的学习成本非常低，并且可以快速高效的实现我们想要的业务逻辑。Go语言在某些方面有着自己的坚持，比如编码风格只有一种，循环只提供一个关键字for等等。在设计者看来，化繁为简，少即是多，世界是并行的。面向对象的思维并不能很好的解决现有的程序设计问题，总是会让设计者陷入为了对象而对象的情况当中。Go语言的思维是面向工程的，一切以工程的思维看待问题，用组合来替代继承。开发Go语言是会上瘾的！