



## 1. 初识 Go 语言

### 1.1 Go 语言介绍

#### 1.1.1 Go 语言是什么

2009 年 11 月 10 日，Go 语言正式成为开源编程语言家庭的一员。

Go 语言（或称 Golang）是云计算时代的 C 语言。Go 语言的诞生是为了让程序员有更高的生产效率，Go 语言专门针对多处理器系统应用程序的编程进行了优化，使用 Go 编译的程序可以媲美 C 或 C++代码的速度，而且更加安全、支持并行进程。

开发人员在为项目选择语言时，不得不在快速开发和性能之间做出选择。C 和 C++这类语言提供了很快的执行速度，而 Ruby 和 Python 这类语言则擅长快速开发。Go 语言在这两者间架起了桥梁，不仅提供了高性能的语言，同时也让开发更快速。

#### 1.1.2 Go 语言优势

- 可直接编译成机器码，不依赖其他库，glibc 的版本有一定要求，部署就是扔一个文件上去就完成了。
- 静态类型语言，但是有动态语言的感觉，静态类型的语言就是可以在编译的时候检查出来隐藏的大多数问题，动态语言的感觉就是有很多的包可以使用，写起来的效率很高。
- 语言层面支持并发，这个就是 Go 最大的特色，天生的支持并发。Go 就是基因里面支持的并发，可以充分的利用多核，很容易的使用并发。
- 内置 runtime，支持垃圾回收，这属于动态语言的特性之一吧，虽然目前来说 GC(内存垃圾回收机制)不算完美，但是足以应付我们所能遇到的大多数情况，特别是 Go1.1 之后的 GC。
- 简单易学，Go 语言的作者都有 C 的基因，那么 Go 自然而然就有了 C 的基因，那么 Go 关键字是 25 个，但是表达能力很强大，几乎支持大多数你在其他语言见过的特性：继承、重载、对象等。
- 丰富的标准库，Go 目前已经内置了大量的库，特别是网络库非常强大。
- 内置强大的工具，Go 语言里面内置了很多工具链，最好的应该是 gofmt 工具，自动化格式化代码，能够让团队 review 变得如此的简单，代码格式一模一样，想不一样都很困难。
- 跨平台编译，如果你写的 Go 代码不包含 cgo，那么就可以做到 window 系统编译 linux 的应用，如何做到的呢？Go 引用了 plan9 的代码，这就是不依赖系统的信息。
- 内嵌 C 支持，Go 里面也可以直接包含 C 代码，利用现有的丰富的 C 库。



### 1.1.3 Go 适合用来做什么

- 服务器编程，以前你如果使用 C 或者 C++ 做的那些事情，用 Go 来做很合适，例如处理日志、数据打包、虚拟机处理、文件系统等。
- 分布式系统，数据库代理器等。
- 网络编程，这一块目前应用最广，包括 Web 应用、API 应用、下载应用。
- 内存数据库，如 google 开发的 groupcache，couchbase 的部分组建。
- 云平台，目前国外很多云平台在采用 Go 开发，CloudFoundry 的部分组建，前 VMare 的技术总监自己出来搞的 apcera 云平台。

## 1.2 环境搭建

### 1.2.1 安装和设置

请参考资料：[Go 语言环境搭建](#)

### 1.2.2 标准命令概述

Go 语言中包含了大量用于处理 Go 语言代码的命令和工具。其中，go 命令就是最常用的一个，它有许多子命令。这些子命令都拥有不同的功能，如下所示。

- build: 用于编译给定的代码包或 Go 语言源码文件及其依赖包。
- clean: 用于清除执行其他 go 命令后遗留的目录和文件。
- doc: 用于执行 godoc 命令以打印指定代码包。
- env: 用于打印 Go 语言环境信息。
- fix: 用于执行 go tool fix 命令以修正给定代码包的源码文件中包含的过时语法和代码调用。
- fmt: 用于执行 gofmt 命令以格式化给定代码包中的源码文件。
- get: 用于下载和安装给定代码包及其依赖包(提前安装 git 或 hg)。
- list: 用于显示给定代码包的信息。
- run: 用于编译并运行给定的命令源码文件。
- install: 编译包文件并编译整个程序。
- test: 用于测试给定的代码包。
- tool: 用于运行 Go 语言的特殊工具。
- version: 用于显示当前安装的 Go 语言的版本信息。

### 1.2.3 学习资料

Go 语言官网(需要翻墙): <https://golang.org/>

go 中文社区: <https://studygolang.com>

go 中文在线文档: <https://studygolang.com/pkgdoc>



## 1.3 第一个 Go 程序

### 1.3.1 Hello Go

```
// hello.go
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello Go!")
}
```

### 1.3.2 代码分析

每个 Go 源代码文件的开头都是一个 `package` 声明，表示该 Go 代码所属的包。包是 Go 语言里最基本的分发单位，也是工程管理中依赖关系的体现。

要生成 Go 可执行程序，必须建立一个名字为 `main` 的包，并且在该包中包含一个叫 `main()` 的函数（该函数是 Go 可执行程序的执行起点）。

Go 语言的 `main()` 函数不能带参数，也不能定义返回值。

在包声明之后，是一系列的 `import` 语句，用于导入该程序所依赖的包。由于本示例程序用到了 `Println()` 函数，所以需要导入该函数所属的 `fmt` 包。

所有 Go 函数以关键字 `func` 开头。一个常规的函数定义包含以下部分：

```
func 函数名(参数列表) (返回值列表) {
    // 函数体
}
```

Go 程序的代码注释与 C++ 保持一致，即同时支持以下两种用法：

```
/* 块注释 */

// 行注释
```

Go 程序并不要求开发者在每个语句后面加上分号表示语句结束，这是与 C 和 C++ 的一个明显不同之处。

**注意：**强制左花括号 `{` 的放置位置，如果把左花括号 `{` 另起一行放置，这样做的结果是 Go 编译器报告编译错误。



```
.\hello.go:8:6: missing function body for "main"
.\hello.go:9:1: syntax error: unexpected semicolon or newline before {
错误：进程退出代码 2.
```

### 1.3.3 命令行运行程序

```
edu@edu:~/share/go/src$ ls
test.go
edu@edu:~/share/go/src$ go build test.go 此命令只会编译代码，不能运行可执行程序
edu@edu:~/share/go/src$ ls
test test.go
edu@edu:~/share/go/src$ ./test
Hello Go!
edu@edu:~/share/go/src$ go run test.go 只会运行，不会生成可执行程序
Hello Go!
edu@edu:~/share/go/src$ █
```

## 2. 基础类型

### 2.1 命名

Go 语言中的函数名、变量名、常量名、类型名、语句标号和包名等所有的命名，都遵循一个简单的命名规则：一个名字必须以一个字母（Unicode 字母）或下划线开头，后面可以跟任意数量的字母、数字或下划线。大写字母和小写字母是不同的：heapSort 和 Heapsort 是两个不同的名字。

Go 语言中类似 if 和 switch 的关键字有 25 个(均为小写)。关键字不能用于自定义名字，只能在特定语法结构中使用。

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

此外，还有大约 30 多个预定义的名字，比如 int 和 true 等，主要对应内建的常量、类型和函数。

内建常量：

```
true false iota nil
```

内建类型：

```
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr
float32 float64 complex128 complex64
```



```
bool byte rune string error
内建函数:
make len cap new append copy close delete
complex real imag
panic recover
```

## 2.2 变量

变量是几乎所有编程语言中最基本的组成元素，变量是程序运行期间可以改变的量。

从根本上说，变量相当于是对一块数据存储空间的命名，程序可以通过定义一个变量来申请一块数据存储空间，之后可以通过引用变量名来使用这块存储空间。

### 2.2.1 变量声明

Go 语言的变量声明方式与 C 和 C++ 语言有明显的不同。对于纯粹的变量声明，Go 语言引入了关键字 `var`，而类型信息放在变量名之后，示例如下：

```
var v1 int
var v2 int

//一次定义多个变量
var v3, v4 int

var (
    v5 int
    v6 int
)
```

### 2.2.2 变量初始化

对于声明变量时需要进行初始化的场景，`var` 关键字可以保留，但不再是必要的元素，如下所示：

```
var v1 int = 10 // 方式 1
var v2 = 10     // 方式 2，编译器自动推导出 v2 的类型
v3 := 10        // 方式 3，编译器自动推导出 v3 的类型
fmt.Println("v3 type is ", reflect.TypeOf(v3)) //v3 type is int

//出现在 := 左侧的变量不应该是已经被声明过，:=定义时必须初始化
var v4 int
v4 := 2 //err
```



### 2.2.3 变量赋值

```
var v1 int
v1 = 123

var v2, v3, v4 int
v2, v3, v4 = 1, 2, 3    //多重赋值

i := 10
j := 20
i, j = j, i            //多重赋值
```

### 2.2.4 匿名变量

\_ (下划线) 是个特殊的变量名，任何赋予它的值都会被丢弃：

```
_ , i, _ , j := 1, 2, 3, 4

func test() (int, string) {
    return 250, "sb"
}

_, str := test()
```

## 2.3 常量

在 Go 语言中，常量是指编译期间就已知且不可改变的值。常量可以是数值类型（包括整型、浮点型和复数类型）、布尔类型、字符串类型等。

### 2.3.1 字面常量(常量值)

所谓字面常量（literal），是指程序中硬编码的常量，如：

```
123
3.1415 // 浮点类型的常量
3.2+12i // 复数类型的常量
true // 布尔类型的常量
"foo" // 字符串常量
```

### 2.3.2 常量定义

```
const Pi float64 = 3.14
const zero = 0.0 // 浮点常量，自动推导类型
```



```
const ( // 常量组
    size int64 = 1024
    eof      = -1 // 整型常量, 自动推导类型
)

const u, v float32 = 0, 3 // u = 0.0, v = 3.0, 常量的多重赋值
const a, b, c = 3, 4, "foo"
// a = 3, b = 4, c = "foo" //err, 常量不能修改
```

### 2.3.3 iota 枚举

常量声明可以使用 `iota` 常量生成器初始化，它用于生成一组以相似规则初始化的常量，但是不用每行都写一遍初始化表达式。

在一个 `const` 声明语句中，在第一个声明的常量所在的行，`iota` 将会被置为 0，然后在每一个有常量声明的行加一。

```
const (
    x = iota // x == 0
    y = iota // y == 1
    z = iota // z == 2
    w // 这里隐式地说 w = iota, 因此 w == 3。其实上面 y 和 z 可同样不用 "= iota"
)

const v = iota // 每遇到一个 const 关键字, iota 就会重置, 此时 v == 0

const (
    h, i, j = iota, iota, iota //h=0,i=0,j=0 iota 在同一行值相同
)

const (
    a      = iota //a=0
    b      = "B"
    c      = iota //c=2
    d, e, f = iota, iota, iota //d=3,e=3,f=3
    g      = iota //g = 4
)

const (
    x1 = iota * 10 // x1 == 0
    y1 = iota * 10 // y1 == 10
    z1 = iota * 10 // z1 == 20
)
```



## 2.4 基础数据类型

### 2.4.1 分类

Go 语言内置以下这些基础类型：

类型	名称	长度	零值	说明
bool	布尔类型	1	false	其值不为真即为假，不可以用数字代表 true 或 false
byte	字节型	1	0	uint8 别名
rune	字符类型	4	0	专用于存储 unicode 编码，等价于 uint32
int, uint	整型	4 或 8	0	32 位或 64 位
int8, uint8	整型	1	0	-128 ~ 127, 0 ~ 255
int16, uint16	整型	2	0	-32768 ~ 32767, 0 ~ 65535
int32, uint32	整型	4	0	-21 亿 ~ 21 亿, 0 ~ 42 亿
int64, uint64	整型	8	0	
float32	浮点型	4	0.0	小数位精确到 7 位
float64	浮点型	8	0.0	小数位精确到 15 位
complex64	复数类型	8		
complex128	复数类型	16		
uintptr	整型	4 或 8		足以存储指针的 uint32 或 uint64 整数
string	字符串		""	utf-8 字符串

### 2.4.2 布尔类型

```
var v1 bool
v1 = true
v2 := (1 == 2) // v2 也会被推导为 bool 类型

//布尔类型不能接受其他类型的赋值，不支持自动或强制的类型转换
var b bool
b = 1 // err, 编译错误
b = bool(1) // err, 编译错误
```

### 2.4.3 整型

```
var v1 int32
```





```
v1 = 123
v2 := 64 // v1 将会被自动推导为 int 类型
```

## 2.4.4 浮点型

```
var f1 float32
f1 = 12
f2 := 12.0 // 如果不加小数点， fvalue2 会被推导为整型而不是浮点型， float64
```

## 2.4.5 字符类型

在 Go 语言中支持两个字符类型，一个是 byte（实际上是 uint8 的别名），代表 utf-8 字符串的单个字节的值；另一个是 rune，代表单个 unicode 字符。

```
package main

import (
    "fmt"
)

func main() {
    var ch1, ch2, ch3 byte
    ch1 = 'a' //字符赋值
    ch2 = 97  //字符的 ascii 码赋值
    ch3 = '\n' //转义字符
    fmt.Printf("ch1 = %c, ch2 = %c, %c", ch1, ch2, ch3)
}
```

## 2.4.6 字符串

在 Go 语言中，字符串也是一种基本类型：

```
var str string // 声明一个字符串变量
str = "abc"    // 字符串赋值
ch := str[0]   // 取字符串的第一个字符
fmt.Printf("str = %s, len = %d\n", str, len(str)) //内置的函数 len()
来取字符串的长度
fmt.Printf("str[0] = %c, ch = %c\n", str[0], ch)

// ` (反引号) 括起的字符串为 Raw 字符串，即字符串在代码中的形式就是打印时的形式，
它没有字符转义，换行也将原样输出。
str2 := `hello
mike \n \r 测试
`
```



```
fmt.Println("str2 = ", str2)
/*
    str2 = hello
        mike \n \r 测试
*/
```

## 2.4.7 复数类型

复数实际上由两个实数（在计算机中用浮点数表示）构成，一个表示实部（real），一个表示虚部（imag）。

```
var v1 complex64 // 由 2 个 float32 构成的复数类型
v1 = 3.2 + 12i
v2 := 3.2 + 12i // v2 是 complex128 类型
v3 := complex(3.2, 12) // v3 结果同 v2

fmt.Println(v1, v2, v3)
//内置函数 real(v1) 获得该复数的实部
//通过 imag(v1) 获得该复数的虚部
fmt.Println(real(v1), imag(v1))
```

## 2.5 fmt 包的格式化输出输入

### 2.5.1 格式说明

格式	含义
%%	一个%字面量
%b	一个二进制整数值(基数为 2)，或者是一个(高级的)用科学计数法表示的指数为 2 的浮点数
%c	字符型。可以把输入的数字按照 ASCII 码相应转换为对应的字符
%d	一个十进制数值(基数为 10)
%e	以科学记数法 e 表示的浮点数或者复数值
%E	以科学记数法 E 表示的浮点数或者复数值
%f	以标准记数法表示的浮点数或者复数值
%g	以%e 或者%f 表示的浮点数或者复数，任何一个都以最为紧凑的方式输出
%G	以%E 或者%f 表示的浮点数或者复数，任何一个都以最为紧凑的方式输出
%o	一个以八进制表示的数字(基数为 8)
%p	以十六进制(基数为 16)表示的一个值的地址，前缀为 0x,字母使用小写的 a-f 表示



格式	含义
%q	使用 Go 语法以及必须时使用转义，以双引号括起来的字符串或者字节切片 []byte，或者是以单引号括起来的数字
%s	字符串。输出字符串中的字符直至字符串中的空字符（字符串以 '\0' 结尾，这个 '\0' 即空字符）
%t	以 true 或者 false 输出的布尔值
%T	使用 Go 语法输出的值的类型
%U	一个用 Unicode 表示法表示的整型码点，默认值为 4 个数字字符
%v	使用默认格式输出的内置或者自定义类型的值，或者是使用其类型的 String() 方式输出的自定义值，如果该方法存在的话
%x	以十六进制表示的整型值(基数为十六)，数字 a-f 使用小写表示
%X	以十六进制表示的整型值(基数为十六)，数字 A-F 使用小写表示

## 2.5.2 输出

```
//整型
a := 15
fmt.Printf("a = %b\n", a) //a = 1111
fmt.Printf("%%\n")      //只输出一个%

//字符
ch := 'a'
fmt.Printf("ch = %c, %c\n", ch, 97) //a, a

//浮点型
f := 3.14
fmt.Printf("f = %f, %g\n", f, f) //f = 3.140000, 3.14
fmt.Printf("f type = %T\n", f)   //f type = float64

//复数类型
v := complex(3.2, 12)
fmt.Printf("v = %f, %g\n", v, v) //v = (3.200000+12.000000i),
(3.2+12i)
fmt.Printf("v type = %T\n", v)   //v type = complex128

//布尔类型
fmt.Printf("%t, %t\n", true, false) //true, false

//字符串
str := "hello go"
fmt.Printf("str = %s\n", str) //str = hello go
```



### 2.5.3 输入

```
var v int
fmt.Println("请输入一个整型：")
fmt.Scanf("%d", &v)
//fmt.Scan(&v)
fmt.Println("v = ", v)
```

## 2.6 类型转换

Go 语言中不允许隐式转换，所有类型转换必须显式声明，而且转换只能发生在两种相互兼容的类型之间。

```
var ch byte = 97
//var a int = ch //err, cannot use ch (type byte) as type int in
assignment
var a int = int(ch)
```

## 2.7 类型别名

```
type bigint int64 //int64 类型改名为 bigint
var x bigint = 100

type (
    myint int    //int 改名为 myint
    mystr string //string 改名为 mystr
)
```

## 3. 运算符

### 3.1 算术运算符

运算符	术语	示例	结果
+	加	10 + 5	15
-	减	10 - 5	5
*	乘	10 * 5	50



运算符	术语	示例	结果
/	除	10 / 5	2
%	取模(取余)	10 % 3	1
++	后自增，没有前自增	a=0; a++	a=1
--	后自减，没有前自减	a=2; a--	a=1

## 3.2 关系运算符

运算符	术语	示例	结果
==	相等于	4 == 3	false
!=	不等于	4 != 3	true
<	小于	4 < 3	false
>	大于	4 > 3	true
<=	小于等于	4 <= 3	false
>=	大于等于	4 >= 1	true

## 3.3 逻辑运算符

运算符	术语	示例	结果
!	非	!a	如果 a 为假，则!a 为真； 如果 a 为真，则!a 为假。
&&	与	a && b	如果 a 和 b 都为真，则结果为真，否则为假。
	或	a    b	如果 a 和 b 有一个为真，则结果为真，二者都为假时，结果为假。

## 3.4 位运算符

运算符	术语	说明	示例
&	按位与	参与运算的两数各对应的二进位相与	60 & 13 结果为 12
	按位或	参与运算的两数各对应的二进位相或	60   13 结果为 61
^	异或	参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为 1	60 ^ 13 结果为 240
<<	左移	左移 n 位就是乘以 2 的 n 次方。左边丢弃，	4 << 2 结果为 16



运算符	术语	说明	示例
>>	右移	右边补 0。 右移 n 位就是除以 2 的 n 次方。右边丢弃，左边补位。	4 >> 2 结果为 1

### 3.5 赋值运算符

运算符	说明	示例
=	普通赋值	c = a + b 将 a + b 表达式结果赋值给 c
+=	相加后再赋值	c += a 等价于 c = c + a
-=	相减后再赋值	c -= a 等价于 c = c - a
*=	相乘后再赋值	c *= a 等价于 c = c * a
/=	相除后再赋值	c /= a 等价于 c = c / a
%=	求余后再赋值	c %= a 等价于 c = c % a
<<=	左移后赋值	c <<= 2 等价于 c = c << 2
>>=	右移后赋值	c >>= 2 等价于 c = c >> 2
&=	按位与后赋值	c &= 2 等价于 c = c & 2
^=	按位异或后赋值	c ^= 2 等价于 c = c ^ 2
=	按位或后赋值	c  = 2 等价于 c = c   2

### 3.6 其他运算符

运算符	术语	示例	说明
&	取地址运算符	&a	变量 a 的地址
*	取值运算符	*a	指针变量 a 所指向内存的值

### 3.7 运算符优先级

在 Go 语言中，一元运算符拥有最高的优先级，二元运算符的运算方向均是从左至右。

下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

优先级	运算符
7	^ !



优先级	运算符
6	* / % << >> & &^
5	+ -   ^
4	== != < <= >= >
3	<-
2	&&
1	

## 4. 流程控制

Go 语言支持最基本的三种程序运行结构：顺序结构、选择结构、循环结构。

- 顺序结构：程序按顺序执行，不发生跳转。
- 选择结构：依据是否满足条件，有选择的执行相应功能。
- 循环结构：依据条件是否满足，循环多次执行某段代码。

### 4.1 选择结构

#### 4.1.1 if 语句

##### 4.1.1.1 if

```
var a int = 3
if a == 3 { //条件表达式没有括号
    fmt.Println("a==3")
}

//支持一个初始化表达式，初始化语句和条件表达式直接需要用分号分隔
if b := 3; b == 3 {
    fmt.Println("b==3")
}
```

##### 4.1.1.2 if ... else

```
if a := 3; a == 4 {
    fmt.Println("a==4")
} else { //左大括号必须和条件语句或 else 在同一行
    fmt.Println("a!=4")
}
```



```
}
```

#### 4.1.1.3 if ... else if ... else

```
if a := 3; a > 3 {  
    fmt.Println("a>3")  
} else if a < 3 {  
    fmt.Println("a<3")  
} else if a == 3 {  
    fmt.Println("a==3")  
} else {  
    fmt.Println("error")  
}
```

#### 4.1.2 switch 语句

Go 里面 switch 默认相当于每个 case 最后带有 break, 匹配成功后不会自动向下执行其他 case, 而是跳出整个 switch, 但是可以使用 fallthrough 强制执行后面的 case 代码:

```
var score int = 90  
  
switch score {  
case 90:  
    fmt.Println("优秀")  
    //fallthrough  
case 80:  
    fmt.Println("良好")  
    //fallthrough  
case 50, 60, 70:  
    fmt.Println("一般")  
    //fallthrough  
default:  
    fmt.Println("差")  
}
```

可以使用任何类型或表达式作为条件语句:

```
//1  
switch s1 := 90; s1 { //初始化语句; 条件  
case 90:  
    fmt.Println("优秀")  
case 80:  
    fmt.Println("良好")  
default:  
    fmt.Println("一般")  
}
```





```
}

//2
var s2 int = 90
switch { //这里没有写条件
case s2 >= 90: //这里写判断语句
    fmt.Println("优秀")
case s2 >= 80:
    fmt.Println("良好")
default:
    fmt.Println("一般")
}

//3
switch s3 := 90; { //只有初始化语句，没有条件
case s3 >= 90: //这里写判断语句
    fmt.Println("优秀")
case s3 >= 80:
    fmt.Println("良好")
default:
    fmt.Println("一般")
}
```

## 4.2 循环语句

### 4.2.1 for

```
var i, sum int

for i = 1; i <= 100; i++ {
    sum += i
}

fmt.Println("sum = ", sum)
```

### 4.2.2 range

关键字 range 会返回两个值，第一个返回值是元素的数组下标，第二个返回值是元素的值：

```
s := "abc"
for i := range s { //支持 string/array/slice/map。
    fmt.Printf("%c\n", s[i])
}
```



```
for _, c := range s { // 忽略 index
    fmt.Printf("%c\n", c)
}

for i, c := range s {
    fmt.Printf("%d, %c\n", i, c)
}
```

## 4.3 跳转语句

### 4.3.1 break 和 continue

在循环里面有两个关键操作 break 和 continue，break 操作是跳出当前循环，continue 是跳过本次循环。

```
for i := 0; i < 5; i++ {
    if 2 == i {
        //break    //break 操作是跳出当前循环
        continue //continue 是跳过本次循环
    }
    fmt.Println(i)
}
```

注意：break 可用于 for、switch、select，而continue 仅能用于 for 循环。

### 4.3.2 goto

用 goto 跳转到必须在当前函数内定义的标签：

```
func main() {
    for i := 0; i < 5; i++ {
        for {
            fmt.Println(i)
            goto LABEL //跳转到标签 LABEL，从标签处，执行代码
        }
    }

    fmt.Println("this is test")

LABEL:
    fmt.Println("it is over")
}
```



## 5. 函数

### 5.1 定义格式

函数构成代码执行的逻辑结构。在 Go 语言中，函数的基本组成为：关键字 `func`、函数名、参数列表、返回值、函数体和返回语句。

Go 语言函数定义格式如下：

```
func FuncName( /*参数列表*/ ) (o1 type1, o2 type2 /*返回类型*/ ) {  
    //函数体  
  
    return v1, v2 //返回多个值  
}
```

函数定义说明：

- `func`：函数由关键字 `func` 开始声明
- `FuncName`：函数名称，根据约定，函数名首字母小写即为 `private`，大写即为 `public`
- 参数列表：函数可以有 0 个或多个参数，参数格式为：变量名 类型，如果有多个参数通过逗号分隔，不支持默认参数
- 返回类型：
  - ① 上面返回值声明了两个变量名 `o1` 和 `o2`(命名返回参数)，这个不是必须，可以只有类型没有变量名
  - ② 如果只有一个返回值且不声明返回值变量，那么你可以省略，包括返回值的括号
  - ③ 如果没有返回值，那么就直接省略最后的返回信息
  - ④ 如果有返回值，那么必须在函数的内部添加 `return` 语句

### 5.2 自定义函数

#### 5.2.1 无参无返回值

```
func Test() { //无参无返回值函数定义  
    fmt.Println("this is a test func")  
}  
  
func main() {  
    Test() //无参无返回值函数调用  
}
```



## 5.2.2 有参无返回值

### 5.2.2.1 普通参数列表

```
func Test01(v1 int, v2 int) { //方式 1
    fmt.Printf("v1 = %d, v2 = %d\n", v1, v2)
}

func Test02(v1, v2 int) { //方式 2, v1, v2 都是 int 类型
    fmt.Printf("v1 = %d, v2 = %d\n", v1, v2)
}

func main() {
    Test01(10, 20) //函数调用
    Test02(11, 22) //函数调用
}
```

### 5.2.2.2 不定参数列表

#### 1) 不定参数类型

不定参数是指函数传入的参数个数为不定数量。为了做到这点，首先需要将函数定义为接受不定参数类型：

```
//形如...type 格式的类型只能作为函数的参数类型存在，并且必须是最后一个参数
func Test(args ...int) {
    for _, n := range args { //遍历参数列表
        fmt.Println(n)
    }
}

func main() {
    //函数调用，可传 0 到多个参数
    Test()
    Test(1)
    Test(1, 2, 3, 4)
}
```

#### 2) 不定参数的传递

```
func MyFunc01(args ...int) {
    fmt.Println("MyFunc01")
    for _, n := range args { //遍历参数列表
        fmt.Println(n)
    }
}
```



```
    }  
}  
  
func MyFunc02(args ...int) {  
    fmt.Println("MyFunc02")  
    for _, n := range args { //遍历参数列表  
        fmt.Println(n)  
    }  
}  
  
func Test(args ...int) {  
    MyFunc01(args...) //按原样传递，Test()的参数原封不动传递给MyFunc01  
    MyFunc02(args[1:]...) //Test()参数列表中，第1个参数及以后的参数传递给  
MyFunc02  
}  
  
func main() {  
    Test(1, 2, 3) //函数调用  
}
```

### 5.2.3 无参有返回值

有返回值的函数，**必须有明确的终止语句**，否则会引发编译错误。

#### 5.2.3.1 一个返回值

```
func Test01() int { //方式1  
    return 250  
}  
  
//官方建议：最好命名返回值，因为不命名返回值，虽然使得代码更加简洁了，但是会造成生  
成的文档可读性差  
func Test02() (value int) { //方式2，给返回值命名  
    value = 250  
    return value  
}  
  
func Test03() (value int) { //方式3，给返回值命名  
    value = 250  
    return  
}  
  
func main() {
```



```
v1 := Test01() //函数调用
v2 := Test02() //函数调用
v3 := Test03() //函数调用
fmt.Printf("v1 = %d, v2 = %d, v3 = %d\n", v1, v2, v3)
}
```

### 5.2.3.2 多个返回值

```
func Test01() (int, string) { //方式1
    return 250, "sb"
}

func Test02() (a int, str string) { //方式2, 给返回值命名
    a = 250
    str = "sb"
    return
}

func main() {
    v1, v2 := Test01() //函数调用
    _, v3 := Test02() //函数调用, 第一个返回值丢弃
    v4, _ := Test02() //函数调用, 第二个返回值丢弃
    fmt.Printf("v1 = %d, v2 = %s, v3 = %s, v4 = %d\n", v1, v2, v3, v4)
}
```

### 5.2.4 有参有返回值

```
//求2个数的最小值和最大值
func MinAndMax(num1 int, num2 int) (min int, max int) {
    if num1 > num2 { //如果 num1 大于 num2
        min = num2
        max = num1
    } else {
        max = num2
        min = num1
    }

    return
}

func main() {
    min, max := MinAndMax(33, 22)
    fmt.Printf("min = %d, max = %d\n", min, max) //min = 22, max = 33
}
```



```
}
```

## 5.3 递归函数

递归指函数可以直接或间接的调用自身。

递归函数通常有相同的结构：一个跳出条件和一个递归体。所谓跳出条件就是根据传入的参数判断是否需要停止递归，而递归体则是函数自身所做的一些处理。

```
//通过循环实现 1+2+3.....+100
func Test01() int {
    i := 1
    sum := 0
    for i = 1; i <= 100; i++ {
        sum += i
    }

    return sum
}

//通过递归实现 1+2+3.....+100
func Test02(num int) int {
    if num == 1 {
        return 1
    }

    return num + Test02(num-1) //函数调用本身
}

//通过递归实现 1+2+3.....+100
func Test03(num int) int {
    if num == 100 {
        return 100
    }

    return num + Test03(num+1) //函数调用本身
}

func main() {

    fmt.Println(Test01()) //5050
    fmt.Println(Test02(100)) //5050
    fmt.Println(Test03(1)) //5050
}
```



```
}
```

## 5.4 函数类型

在 Go 语言中，函数也是一种数据类型，我们可以通过 type 来定义它，它的类型就是所有拥有相同的参数，相同的返回值的一种类型。

```
type FuncType func(int, int) int //声明一个函数类型，func 后面没有函数名

//函数中有一个参数类型为函数类型：f FuncType
func Calc(a, b int, f FuncType) (result int) {
    result = f(a, b) //通过调用 f() 实现任务
    return
}

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

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

func main() {
    //函数调用，第三个参数为函数名字，此函数的参数，返回值必须和 FuncType 类型一致
    result := Calc(1, 1, Add)
    fmt.Println(result) //2

    var f FuncType = Minus
    fmt.Println("result = ", f(10, 2)) //result = 8
}
```

## 5.5 匿名函数与闭包

所谓闭包就是一个函数“捕获”了和它在同一作用域的其他常量和变量。这就意味着当闭包被调用的时候，不管在程序什么地方调用，闭包能够使用这些常量或者变量。它不关心这些捕获了的变量和常量是否已经超出了作用域，所以只有闭包还在使用它，这些变量就还会存在。

在 Go 语言里，所有的匿名函数(Go 语言规范中称之为函数字面量)都是闭包。匿名函数是指不需要定义函数名的一种函数实现方式，它并不是一个新概念，最早可以回溯到 1958 年的





Lisp 语言。

```
func main() {  
    i := 0  
    str := "mike"  
  
    //方式 1  
    f1 := func() { //匿名函数，无参无返回值  
        //引用到函数外的变量  
        fmt.Printf("方式 1: i = %d, str = %s\n", i, str)  
    }  
  
    f1() //函数调用  
  
    //方式 1 的另一种方式  
    type FuncType func() //声明函数类型，无参无返回值  
    var f2 FuncType = f1  
    f2() //函数调用  
  
    //方式 2  
    var f3 FuncType = func() {  
        fmt.Printf("方式 2: i = %d, str = %s\n", i, str)  
    }  
    f3() //函数调用  
  
    //方式 3  
    func() { //匿名函数，无参无返回值  
        fmt.Printf("方式 3: i = %d, str = %s\n", i, str)  
    }() //别忘了后面的(), ()的作用是，此处直接调用此匿名函数  
  
    //方式 4, 匿名函数，有参有返回值  
    v := func(a, b int) (result int) {  
        result = a + b  
        return  
    }(1, 1) //别忘了后面的(1, 1), (1, 1)的作用是，此处直接调用此匿名函数，并  
    传参  
    fmt.Println("v = ", v)  
}
```

闭包捕获外部变量特点:

```
func main() {  
    i := 10  
    str := "mike"
```



```
func() {
    i = 100
    str = "go"
    //内部: i = 100, str = go
    fmt.Printf("内部: i = %d, str = %s\n", i, str)
}() //别忘了后面的(), ()的作用是, 此处直接调用此匿名函数

//外部: i = 100, str = go
fmt.Printf("外部: i = %d, str = %s\n", i, str)
}
```

函数返回值为匿名函数:

```
// squares 返回一个匿名函数, func() int
// 该匿名函数每次被调用时都会返回下一个数的平方。
func squares() func() int {
    var x int
    return func() int { //匿名函数
        x++ //捕获外部变量
        return x * x
    }
}

func main() {
    f := squares()
    fmt.Println(f()) // "1"
    fmt.Println(f()) // "4"
    fmt.Println(f()) // "9"
    fmt.Println(f()) // "16"
}
```

函数 `squares` 返回另一个类型为 `func() int` 的函数。对 `squares` 的一次调用会生成一个局部变量 `x` 并返回一个匿名函数。每次调用时匿名函数时, 该函数都会先使 `x` 的值加 1, 再返回 `x` 的平方。第二次调用 `squares` 时, 会生成第二个 `x` 变量, 并返回一个新的匿名函数。新匿名函数操作的是第二个 `x` 变量。

通过这个例子, 我们看到变量的生命周期不由它的作用域决定: `squares` 返回后, 变量 `x` 仍然隐式的存在于 `f` 中。



## 5.6 延迟调用 defer

### 5.6.1 defer 作用

关键字 `defer` 用于延迟一个函数或者方法（或者当前所创建的匿名函数）的执行。注意，`defer` 语句只能出现在函数或方法的内部。

```
func main() {  
    fmt.Println("this is a test")  
    defer fmt.Println("this is a defer") //main 结束前调用  
  
    /*  
        运行结果:  
        this is a test  
        this is a defer  
    */  
}
```

`defer` 语句经常被用于处理成对的操作，如打开、关闭、连接、断开连接、加锁、释放锁。通过 `defer` 机制，不论函数逻辑多复杂，都能保证在任何执行路径下，资源被释放。释放资源的 `defer` 应该直接跟在请求资源的语句后。

### 5.6.2 多个 defer 执行顺序

如果一个函数中有多个 `defer` 语句，它们会以 LIFO（后进先出）的顺序执行。哪怕函数或某个延迟调用发生错误，这些调用依旧会被执行。

```
func test(x int) {  
    fmt.Println(100 / x) //x 为 0 时，产生异常  
}  
  
func main() {  
    defer fmt.Println("aaaaaaaa")  
    defer fmt.Println("bbbbbbbb")  
  
    defer test(0)  
  
    defer fmt.Println("cccccccc")  
    /*  
        运行结果:  
        cccccccc  
    */  
}
```



```
bbbbbbbbb
aaaaaaaaa
panic: runtime error: integer divide by zero
*/
}
```

### 5.6.3 defer 和匿名函数结合使用

```
func main() {
    a, b := 10, 20
    defer func(x int) { // a 以值传递方式传给 x
        fmt.Println("defer:", x, b) // b 闭包引用
    }(a)

    a += 10
    b += 100

    fmt.Printf("a = %d, b = %d\n", a, b)

    /*
        运行结果:
        a = 20, b = 120
        defer: 10 120
    */
}
```

## 5.7 获取命令行参数

```
package main

import (
    "fmt"
    "os" //os.Args 所需的包
)

func main() {
    args := os.Args //获取用户输入的所有参数

    //如果用户没有输入,或参数个数不够,则调用该函数提示用户
    if args == nil || len(args) < 2 {
        fmt.Println("err: xxx ip port")
        return
    }
}
```



```
}  
ip := args[1] //获取输入的第一个参数  
port := args[2] //获取输入的第二个参数  
fmt.Printf("ip = %s, port = %s\n", ip, port)  
}
```

运行结果如下：

```
C:\Users\superman\Desktop\code\go\src>dir  
驱动器 C 中的卷没有标签。  
卷的序列号是 4040-7EEB  
  
C:\Users\superman\Desktop\code\go\src 的目录  
  
2017/12/29  22:51    <DIR>          .  
2017/12/29  22:51    <DIR>          ..  
2017/12/29  22:51                448 main.go  
2017/12/29  22:51            1,950,720 src.exe  
2017/12/17  17:03    <DIR>          test  
                2 个文件          1,951,168 字节  
                3 个目录 78,069,923,840 可用字节  
  
C:\Users\superman\Desktop\code\go\src>src 127.0.0.1 8888  
ip = 127.0.0.1, port = 8888  
  
C:\Users\superman\Desktop\code\go\src>
```

## 5.8 作用域

作用域为已声明标识符所表示的常量、类型、变量、函数或包在源代码中的作用范围。

### 5.8.1 局部变量

在函数体内声明的变量、参数和返回值变量就是局部变量，它们的作用域只在函数体内：

```
func test(a, b int) {  
    var c int  
    a, b, c = 1, 2, 3  
    fmt.Printf("a = %d, b = %d, c = %d\n", a, b, c)  
}  
  
func main() {  
    //a, b, c = 1, 2, 3 //err, a, b, c 不属于此作用域  
    {  
        var i int  
        i = 10  
        fmt.Printf("i = %d\n", i)  
    }  
}
```



```
}

//i = 20 //err, i 不属于此作用域

if a := 3; a == 3 {
    fmt.Println("a = ", a)
}

//a = 4 //err, a 只能 if 内部使用
}
```

## 5.8.2 全局变量

在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。

```
var a int //全局变量的声明

func test() {
    fmt.Printf("test a = %d\n", a)
}

func main() {
    a = 10
    fmt.Printf("main a = %d\n", a) //main a = 10

    test() //test a = 10
}
```

## 5.8.3 不同作用域同名变量

在不同作用域可以声明同名的变量，其访问原则为：在同一个作用域内，就近原则访问最近的变量，如果此作用域没有此变量声明，则访问全局变量，如果全局变量也没有，则报错。

```
var a int //全局变量的声明

func test01(a float32) {
    fmt.Printf("a type = %T\n", a) //a type = float32
}

func main() {
    fmt.Printf("a type = %T\n", a) //a type = int, 说明使用全局变量的 a

    var a uint8 //局部变量声明
}
```



```
{
    var a float64           //局部变量声明
    fmt.Printf("a type = %T\n", a) //a type = float64
}

fmt.Printf("a type = %T\n", a) //a type = uint8

test01(3.14)
test02()
}

func test02() {
    fmt.Printf("a type = %T\n", a) //a type = int
}
```

## 6. 工程管理

在实际的开发工作中，直接调用编译器进行编译和链接的场景是少而又少，因为在**工程中不会简单到只有一个源代码文件，且源文件之间会有相互的依赖关系**。如果这样一个文件一个文件逐步编译，那不亚于一场灾难。Go 语言的设计者作为行业老将，自然不会忽略这一点。早期 Go 语言使用 makefile 作为临时方案，到了 Go 1 发布时引入了强大无比的 Go 命令行工具。

Go 命令行工具的革命性之处在于彻底消除了工程文件的概念，完全用目录结构和包名来推导工程结构和构建顺序。针对只有一个源文件的情况讨论工程管理看起来会比较多余，因为这可以直接用 go run 和 go build 搞定。下面我们将用一个更接近现实的虚拟项目来展示 Go 语言的基本工程管理能力。

### 6.1 工作区

#### 6.1.1 工作区介绍

**Go 代码必须放在工作区中**。工作区其实就是一个对应于特定工程的目录，它应包含 3 个子目录：src 目录、pkg 目录和 bin 目录。

- src 目录：用于以代码包的形式组织并保存 Go 源码文件。（比如：.go .c .h .s 等）
- pkg 目录：用于存放经由 go install 命令构建安装后的代码包（包含 Go 库源码文件）的“.a”归档文件。
- bin 目录：与 pkg 目录类似，在通过 go install 命令完成安装后，保存由 Go 命令源码文件生成的可执行文件。



目录 `src` 用于包含所有的源代码，是 Go 命令行工具一个强制的规则，而 `pkg` 和 `bin` 则无需手动创建，如果必要 Go 命令行工具在构建过程中会自动创建这些目录。

需要特别注意的是，只有当环境变量 `GOPATH` 中只包含一个工作区的目录路径时，`go install` 命令才会把命令源码安装到当前工作区的 `bin` 目录下。若环境变量 `GOPATH` 中包含多个工作区的目录路径，像这样执行 `go install` 命令就会失效，此时必须设置环境变量 `GOBIN`。

### 6.1.2 GOPATH 设置

为了能够构建这个工程，需要先把所需工程的根目录加入到环境变量 `GOPATH` 中。否则，即使处于同一工作目录(工作区)，代码之间也无法通过绝对代码包路径完成调用。

在实际开发环境中，工作目录往往有多个。这些工作目录的目录路径都需要添加至 `GOPATH`。当有多个目录时，请注意分隔符，多个目录的时候 Windows 是分号，Linux 系统是冒号，当有多个 `GOPATH` 时，默认会将 `go get` 的内容放在第一个目录下。

## 6.2 包

所有 Go 语言的程序都会组织成若干组文件，每组文件被称为一个包。这样每个包的代码都可以作为很小的复用单元，被其他项目引用。

一个包的源代码保存在一个或多个以 `.go` 为文件后缀名的源文件中，通常一个包所在目录路径的后缀是包的导入路径。

### 6.2.1 自定义包

对于一个较大的应用程序，我们应该将它的功能性分隔成逻辑的单元，分别在不同的包里实现。我们创建的自定义包最好放在 `GOPATH` 的 `src` 目录下（或者 `GOPATH src` 的某个子目录）。

在 Go 语言中，代码包中的源码文件名可以是任意的。但是，这些任意名称的源码文件都必须以包声明语句作为文件中的第一行，每个包都对应一个独立的命名空间：

```
package calc
```

包中成员以名称首字母大小写决定访问权限：

- `public`：首字母大写，可被包外访问
- `private`：首字母小写，仅包内成员可以访问





**注意：**同一个目录下**不能**定义不同的 package。

## 6.2.2 main 包

在 Go 语言里，命名为 main 的包具有特殊的含义。Go 语言的编译程序会试图把这种名字的包编译为二进制可执行文件。**所有用 Go 语言编译的可执行程序都必须有一个名叫 main 的包。**一个可执行程序有且仅有一个 main 包。

当编译器发现某个包的名字为 main 时，它一定也会发现名为 main() 的函数，否则不会创建可执行文件。main() 函数是程序的入口，所以，如果没有这个函数，程序就没有办法开始执行。程序编译时，会使用声明 main 包的代码所在的目录的目录名作为二进制可执行文件的文件名。

## 6.2.3 main 函数和 init 函数

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

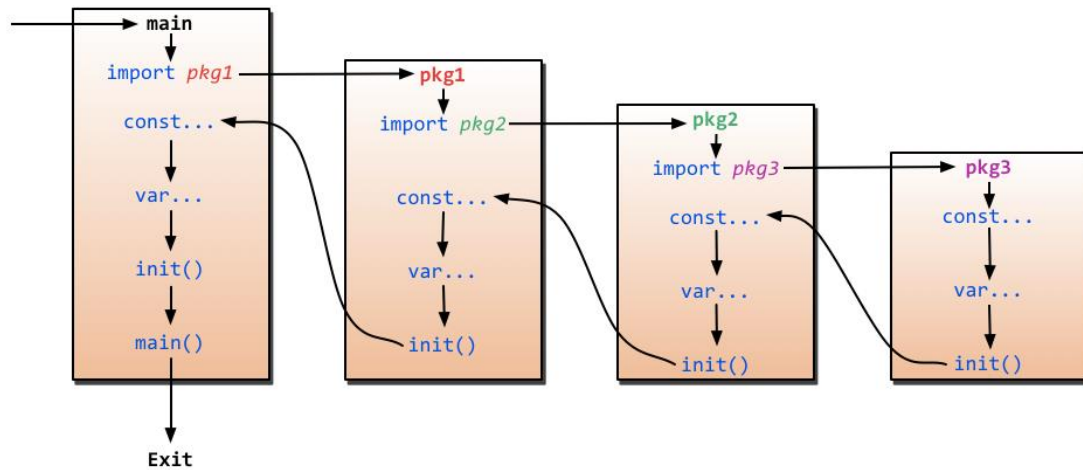
Go 程序会自动调用 init() 和 main()，所以你不需要在任何地方调用这两个函数。每个 package 中的 init 函数都是可选的，**但 package main 就必须包含一个 main 函数。**

每个包可以包含任意多个 init 函数，这些函数都会在程序执行开始的时候被调用。所有被编译器发现的 init 函数都会安排在 main 函数之前执行。init 函数用在设置包、初始化变量或者其他要在程序运行前优先完成的引导工作。

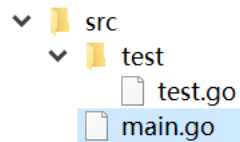
程序的初始化和执行都起始于 main 包。如果 main 包还导入了其它的包，那么就会在编译时将它们依次导入。

有时一个包会被多个包同时导入，那么它只会被导入一次（例如很多包可能都会用到 fmt 包，但它只会被导入一次，因为没有必要导入多次）。

当一个包被导入时，如果该包还导入了其它的包，那么会先将其它包导入进来，然后再对这些包中的包级常量和变量进行初始化，接着执行 init 函数（如果有的话），依次类推。等所有被导入的包都加载完毕了，就会开始对 main 包中的包级常量和变量进行初始化，然后执行 main 包中的 init 函数（如果存在的话），最后执行 main 函数。下图详细地解释了整个执行过程：



示例代码目录结构：



main.go 示例代码如下：

```
// main.go
package main

import (
    "fmt"
    "test"
)

func main() {
    fmt.Println("main.go main() is called")

    test.Test()
}
```

test.go 示例代码如下：

```
//test.go
package test

import "fmt"

func init() {
    fmt.Println("test.go init() is called")
}
```



```
func Test() {  
    fmt.Println("test.go Test() is called")  
}
```

运行结果：

```
test.go init() is called  
main.go main() is called  
test.go Test() is called  
成功：进程退出代码 0.
```

## 6.2.4 导入包

导入包需要使用关键字 `import`，它会告诉编译器你想引用该位置的包内的代码。包的路径可以是相对路径，也可以是绝对路径。

```
//方法 1  
import "calc"  
import "fmt"  
  
//方法 2  
import (  
    "calc"  
    "fmt"  
)
```

标准库中的包会在安装 Go 的位置找到。Go 开发者创建的包会在 `GOPATH` 环境变量指定的目录里查找。`GOPATH` 指定的这些目录就是开发者的个人工作空间。

如果编译器查遍 `GOPATH` 也没有找到要导入的包，那么在试图对程序执行 `run` 或者 `build` 的时候就会出错。

**注意：**如果导入包之后，未调用其中的函数或者类型将会报出编译错误。

`.\test.go:4:2: imported and not used: "fmt"`

**错误：进程退出代码 2.**

### 6.2.4.1 点操作

```
import (  
    //这个点操作的含义是这个包导入之后在你调用这个包的函数时，可以省略前缀的包名  
    . "fmt"  
)  
  
func main() {  
    Println("hello go")  
}
```



```
}
```

#### 6.2.4.2 别名操作

在导入时，可指定包成员访问方式，比如对包重命名，以避免同名冲突：

```
import (  
    io "fmt" //fmt 改为为 io  
)  
  
func main() {  
    io.Println("hello go") //通过 io 别名调用  
}
```

#### 6.2.4.3 \_操作

有时，用户可能需要导入一个包，但是不需要引用这个包的标识符。在这种情况下，可以使用空白标识符\_来重命名这个导入：

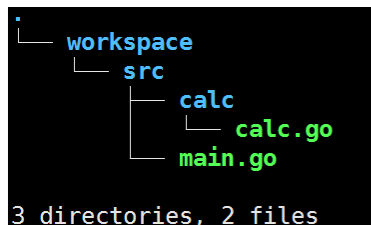
```
import (  
    _ "fmt"  
)
```

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



## 6.3 测试案例

### 6.3.1 测试代码



```
main.go
1 package main
2
3 import (
4     "calc"
5     "fmt"
6 )
7
8 func main() {
9     a := calc.Add(1, 2)
10    fmt.Println("a = ", a)
11 }
```

```
calc.go
1 package calc
2
3 func Add(a, b int) int { //加
4     return a + b
5 }
6
7 func Minus(a, b int) int { //减
8     return a - b
9 }
10
11 func Multiply(a, b int) int { //乘
12     return a * b
13 }
14
15 func Divide(a, b int) int { //除
16     return a / b
17 }
```

calc.go 代码如下:

```
package calc

func Add(a, b int) int { //加
    return a + b
}

func Minus(a, b int) int { //减
    return a - b
}

func Multiply(a, b int) int { //乘
    return a * b
}

func Divide(a, b int) int { //除
    return a / b
}
```

main.go 代码如下:

```
package main

import (
```

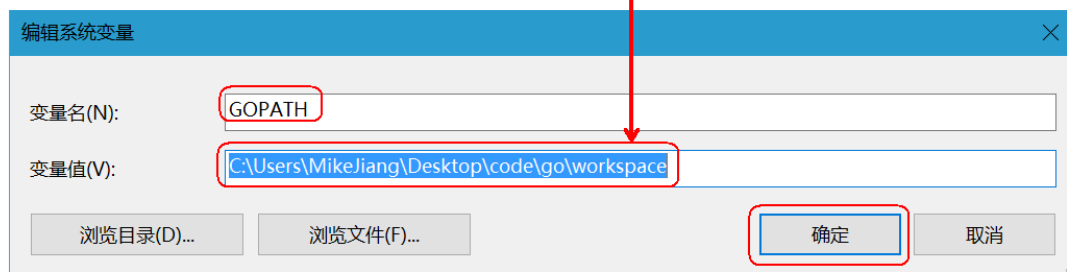
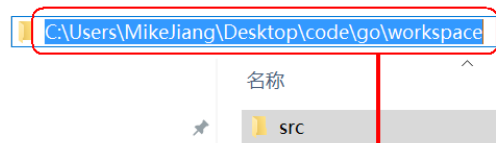
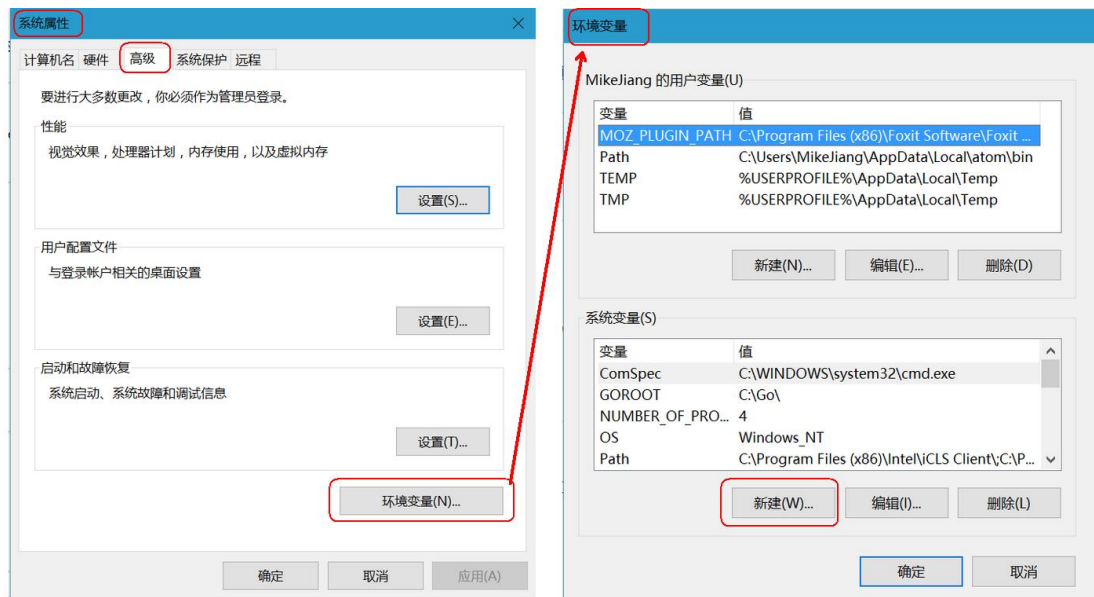


```
        "calc"  
        "fmt"  
    )  
  
    func main() {  
        a := calc.Add(1, 2)  
        fmt.Println("a = ", a)  
    }
```



## 6.3.2 GOPATH 设置

### 6.3.2.1 windows



```
C:\Users\MikeJiang>go env
set GOARCH=amd64
set GOBIN=
set GOEXE=.exe
set GOHOSTARCH=amd64
set GOHOSTOS=windows
set GOOS=windows
set GOPATH=C:\Users\MikeJiang\Desktop\code\go\workspace
set GORACE=
set GOROOT=C:\Go
set GOTOLDIR=C:\Go\pkg\tool\windows_amd64
set GCCGO=gccgo
set CC=gcc
```



### 6.3.2.2 linux

```
edu@edu:~/share/go/workspace$ pwd
/home/edu/share/go/workspace
edu@edu:~/share/go/workspace$
edu@edu:~/share/go/workspace$ export GOPATH=/home/edu/share/go/workspace
edu@edu:~/share/go/workspace$ go env
GOARCH="amd64"
GOBIN=""
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOS="linux"
GOPATH="/home/edu/share/go/workspace"
GORACE=""
GOROOT="/usr/lib/go-1.6"
GOTOOLDIR="/usr/lib/go-1.6/pkg/tool/linux_amd64"
GO15VENDOREXPERIMENT="1"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0"
CXX="g++"
CGO_ENABLED="1"
edu@edu:~/share/go/workspace$
```

### 6.3.3 编译运行程序

```
edu@edu:~/share/go/workspace/src$ ls
calc main.go
edu@edu:~/share/go/workspace/src$ go build
edu@edu:~/share/go/workspace/src$ ls
calc main.go src
edu@edu:~/share/go/workspace/src$ ./src
a = 3
edu@edu:~/share/go/workspace/src$ go run main.go
a = 3
edu@edu:~/share/go/workspace/src$
```

### 6.3.4 go install 的使用

设置环境变量 GOBIN:





```
edu@edu:~/share/go/workspace$ export GOBIN=/home/edu/share/go/workspace/bin
edu@edu:~/share/go/workspace$ go env
GOARCH="amd64"
GOBIN="/home/edu/share/go/workspace/bin"
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/edu/share/go/workspace"
GORACE=""
GOROOT="/usr/lib/go-1.6"
GOTOOLDIR="/usr/lib/go-1.6/pkg/tool/linux_amd64"
GO15VENDOREXPERIMENT="1"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0"
CXX="g++"
CGO_ENABLED="1"
edu@edu:~/share/go/workspace$
```

在源码目录，敲 go install:

```
edu@edu:~/share/go/workspace$ tree
.
└── src
    ├── calc
    │   └── calc.go
    └── main.go

2 directories, 2 files
edu@edu:~/share/go/workspace$ cd src
edu@edu:~/share/go/workspace/src$ go install
edu@edu:~/share/go/workspace/src$ cd ..
edu@edu:~/share/go/workspace$ tree
.
├── bin
│   └── src
├── pkg
│   └── linux_amd64
│       └── calc.a
└── src
    ├── calc
    │   └── calc.go
    └── main.go

5 directories, 4 files
```



## 7. 复合类型

### 7.1 分类

类型	名称	长度	默认值	说明
pointer	指针		nil	
array	数组		0	
slice	切片		nil	引用类型
map	字典		nil	引用类型
struct	结构体			

### 7.2 指针

指针是一个代表着某个内存地址的值。这个内存地址往往是在内存中存储的另一个变量的值的起始位置。Go 语言对指针的支持介于 Java 语言和 C/C++ 语言之间，它既没有像 Java 语言那样取消了代码对指针的直接操作的能力，也避免了 C/C++ 语言中由于对指针的滥用而造成的安全和可靠性问题。

#### 7.2.1 基本操作

Go 语言虽然保留了指针，但与其它编程语言不同的是：

- 默认值 nil，没有 NULL 常量
- 操作符 "&" 取变量地址，"\*" 通过指针访问目标对象
- 不支持指针运算，不支持 "->" 运算符，直接用 "." 访问目标成员

```
func main() {  
    var a int = 10           //声明一个变量，同时初始化  
    fmt.Printf("&a = %p\n", &a) //操作符 "&" 取变量地址  
  
    var p *int = nil //声明一个变量 p，类型为 *int，指针类型  
    p = &a  
    fmt.Printf("p = %p\n", p)  
    fmt.Printf("a = %d, *p = %d\n", a, *p)  
  
    *p = 111 // *p 操作指针所指向的内存，即为 a  
    fmt.Printf("a = %d, *p = %d\n", a, *p)  
}
```



```
}
```

### 7.2.2 new 函数

表达式 `new(T)` 将创建一个 `T` 类型的匿名变量，所做的是为 `T` 类型的新值分配并清零一块内存空间，然后将这块内存空间的地址作为结果返回，而这个结果就是指向这个新的 `T` 类型值的指针值，返回的指针类型为 `*T`。

```
func main() {  
    var p1 *int  
    p1 = new(int)           //p1 为*int 类型，指向匿名的 int 变量  
    fmt.Println("*p1 = ", *p1) // *p1 = 0  
  
    p2 := new(int) //p2 为*int 类型，指向匿名的 int 变量  
    *p2 = 111  
    fmt.Println("*p2 = ", *p2) // *p1 = 111  
}
```

我们只需使用 `new()` 函数，无需担心其内存的生命周期或怎样将其删除，因为 Go 语言的内存管理系统会帮我们打理一切。

### 7.2.3 指针做函数参数

```
func swap01(a, b int) {  
    a, b = b, a  
    fmt.Printf("swap01 a = %d, b = %d\n", a, b)  
}  
  
func swap02(x, y *int) {  
    *x, *y = *y, *x  
}  
  
func main() {  
    a := 10  
    b := 20  
  
    //swap01(a, b) //值传递  
    swap02(&a, &b) //变量地址传递  
    fmt.Printf("a = %d, b = %d\n", a, b)  
}
```



## 7.3 数组

### 7.3.1 概述

数组是指一系列同一类型数据的集合。数组中包含的每个数据被称为数组元素（element），一个数组包含的元素个数被称为数组的长度。

数组长度必须是常量，且是类型的组成部分。[2]int 和 [3]int 是不同类型。

```
var n int = 10
var a [n]int //err, non-constant array bound n
var b [10]int //ok
```

### 7.3.2 操作数组

数组的每个元素可以通过索引下标来访问，索引下标的范围是从 0 开始到数组长度减 1 的位置。

```
var a [10]int
for i := 0; i < 10; i++ {
    a[i] = i + 1
    fmt.Printf("a[%d] = %d\n", i, a[i])
}

//range 具有两个返回值，第一个返回值是元素的数组下标，第二个返回值是元素的值
for i, v := range a {
    fmt.Println("a[" + i + "]=", v)
}
```

内置函数 len(长度) 和 cap(容量) 都返回数组长度 (元素数量):

```
a := [10]int{}
fmt.Println(len(a), cap(a))//10 10
```

初始化:

```
a := [3]int{1, 2}           // 未初始化元素值为 0
b := [...]int{1, 2, 3}      // 通过初始化值确定数组长度
c := [5]int{2: 100, 4: 200} // 通过索引号初始化元素，未初始化元素值为 0
fmt.Println(a, b, c)        //[1 2 0] [1 2 3] [0 0 100 0 200]

//支持多维数组
d := [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}
e := [...] [2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}} //第二维不能写"..."
```



```
f := [4][2]int{1: {20, 21}, 3: {40, 41}}
g := [4][2]int{1: {0: 20}, 3: {1: 41}}
fmt.Println(d, e, f, g)
```

相同类型的数组之间可以使用 == 或 != 进行比较，但不可以使用 < 或 >，也可以相互赋值：

```
a := [3]int{1, 2, 3}
b := [3]int{1, 2, 3}
c := [3]int{1, 2}
fmt.Println(a == b, b == c) //true false

var d [3]int
d = a
fmt.Println(d) //[1 2 3]
```

### 7.3.3 在函数间传递数组

根据内存和性能来看，在函数间传递数组是一个开销很大的操作。在函数之间传递变量时，**总是以值的方式传递的**。如果这个变量是一个数组，意味着整个数组，不管有多长，都会完整复制，并传递给函数。

```
func modify(array [5]int) {
    array[0] = 10 // 试图修改数组的第一个元素
    //In modify(), array values: [10 2 3 4 5]
    fmt.Println("In modify(), array values:", array)
}

func main() {
    array := [5]int{1, 2, 3, 4, 5} // 定义并初始化一个数组
    modify(array)                  // 传递给一个函数，并试图在函数体内修改这个数组内容
    //In main(), array values: [1 2 3 4 5]
    fmt.Println("In main(), array values:", array)
}
```

数组指针做函数参数：

```
func modify(array *[5]int) {
    (*array)[0] = 10
    //In modify(), array values: [10 2 3 4 5]
    fmt.Println("In modify(), array values:", *array)
}

func main() {
```



```
array := [5]int{1, 2, 3, 4, 5} // 定义并初始化一个数组
modify(&array)                // 数组指针
//In main(), array values: [10 2 3 4 5]
fmt.Println("In main(), array values:", array)
}
```

## 7.4 slice

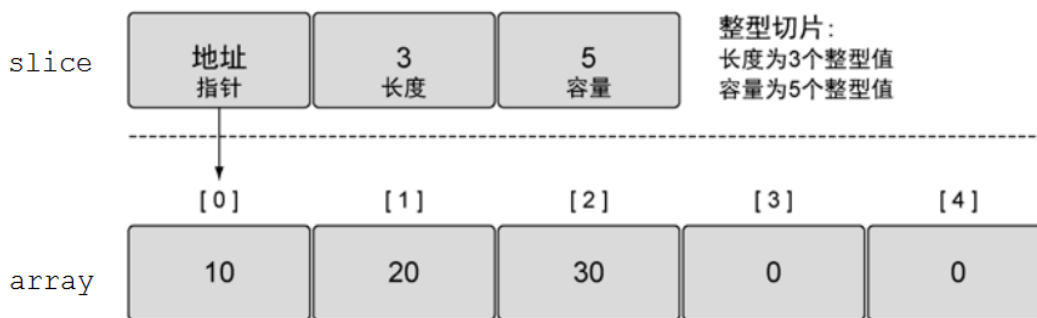
### 7.4.1 概述

数组的长度在定义之后无法再次修改；数组是值类型，每次传递都将产生一份副本。显然这种数据结构无法完全满足开发者的真实需求。Go 语言提供了数组切片（slice）来弥补数组的不足。

切片并不是数组或数组指针，它通过内部指针和相关属性引用数组片段，以实现变长方案。

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

```
array := [...]int{10, 20, 30, 0, 0}
slice := array[0:3:5]
```



### 7.4.2 切片的创建和初始化

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

```
var s1 []int //声明切片和声明 array 一样，只是少了长度，此为 nil 切片
s2 := []int{}

//make([]T, length, capacity) //capacity 省略，则和 length 的值相同
```



```
var s3 []int = make([]int, 0)
s4 := make([]int, 0, 0)

s5 := []int{1, 2, 3} //创建切片并初始化
```

**注意：**make 只能创建 slice、map 和 channel，并且返回一个有初始值(非零)。

## 7.4.3 切片的操作

### 7.4.3.1 切片截取

操作	含义
<b>s[n]</b>	切片 s 中索引位置为 n 的项
<b>s[:]</b>	从切片 s 的索引位置 0 到 len(s)-1 处所获得的切片
<b>s[low:]</b>	从切片 s 的索引位置 low 到 len(s)-1 处所获得的切片
<b>s[:high]</b>	从切片 s 的索引位置 0 到 high 处所获得的切片，len=high
<b>s[low:high]</b>	从切片 s 的索引位置 low 到 high 处所获得的切片，len=high-low
<b>s[low:high:max]</b>	从切片 s 的索引位置 low 到 high 处所获得的切片，len=high-low，cap=max-low
<b>len(s)</b>	切片 s 的长度，总是<=cap(s)
<b>cap(s)</b>	切片 s 的容量，总是>=len(s)

示例说明：

```
array := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

操作	结果	len	cap	说明
<b>array[:6:8]</b>	[0 1 2 3 4 5]	6	8	省略 low
<b>array[5:]</b>	[5 6 7 8 9]	5	5	省略 high、 max
<b>array[:3]</b>	[0 1 2]	3	10	省略 high、 max
<b>array[:]</b>	[0 1 2 3 4 5 6 7 8 9]	10	10	全部省略

### 7.4.3.2 切片和底层数组关系

```
s := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

s1 := s[2:5] // [2 3 4]
s1[2] = 100 // 修改切片某个元素改变底层数组
fmt.Println(s1, s) // [2 3 100] [0 1 2 3 100 5 6 7 8 9]
```



```
s2 := s1[2:6] // 新切片依旧指向原底层数组 [100 5 6 7]
s2[3] = 200
fmt.Println(s2) //[100 5 6 200]

fmt.Println(s) //[0 1 2 3 100 5 6 200 8 9]
```

### 7.4.3.3 内建函数

#### 1) append

append 函数向 slice 尾部添加数据，返回新的 slice 对象：

```
var s1 []int //创建 nil 切片
//s1 := make([]int, 0)

s1 = append(s1, 1) //追加 1 个元素
s1 = append(s1, 2, 3) //追加 2 个元素
s1 = append(s1, 4, 5, 6) //追加 3 个元素
fmt.Println(s1) // [1 2 3 4 5 6]

s2 := make([]int, 5)
s2 = append(s2, 6)
fmt.Println(s2) //[0 0 0 0 0 6]

s3 := []int{1, 2, 3}
s3 = append(s3, 4, 5)
fmt.Println(s3) //[1 2 3 4 5]
```

append 函数会智能地底层数组的容量增长，一旦超过原底层数组容量，通常以 2 倍容量重新分配底层数组，并复制原来的数据：

```
func main() {
    s := make([]int, 0, 1)
    c := cap(s)
    for i := 0; i < 50; i++ {
        s = append(s, i)
        if n := cap(s); n > c {
            fmt.Printf("cap: %d -> %d\n", c, n)
            c = n
        }
    }
    /*
    cap: 1 -> 2
    cap: 2 -> 4
    cap: 4 -> 8
    cap: 8 -> 16
    */
}
```





```
cap: 16 -> 32
cap: 32 -> 64

*/
}
```

## 2) copy

函数 `copy` 在两个 slice 间复制数据，复制长度以 `len` 小的为准，两个 slice 可指向同一底层数组。

```
data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
s1 := data[8:] //{8, 9}
s2 := data[:5] //{0, 1, 2, 3, 4}
copy(s2, s1)    // dst:s2, src:s1

fmt.Println(s2) //[8 9 2 3 4]
fmt.Println(data) //[8 9 2 3 4 5 6 7 8 9]
```

## 7.4.4 切片做函数参数

```
func test(s []int) { //切片做函数参数
    s[0] = -1
    fmt.Println("test : ")
    for i, v := range s {
        fmt.Printf("s[%d]=%d, ", i, v)
        //s[0]=-1, s[1]=1, s[2]=2, s[3]=3, s[4]=4, s[5]=5, s[6]=6,
s[7]=7, s[8]=8, s[9]=9,
    }
    fmt.Println("\n")
}

func main() {
    slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    test(slice)

    fmt.Println("main : ")
    for i, v := range slice {
        fmt.Printf("slice[%d]=%d, ", i, v)
        //slice[0]=-1, slice[1]=1, slice[2]=2, slice[3]=3, slice[4]=4,
slice[5]=5, slice[6]=6, slice[7]=7, slice[8]=8, slice[9]=9,
    }
    fmt.Println("\n")
}
```

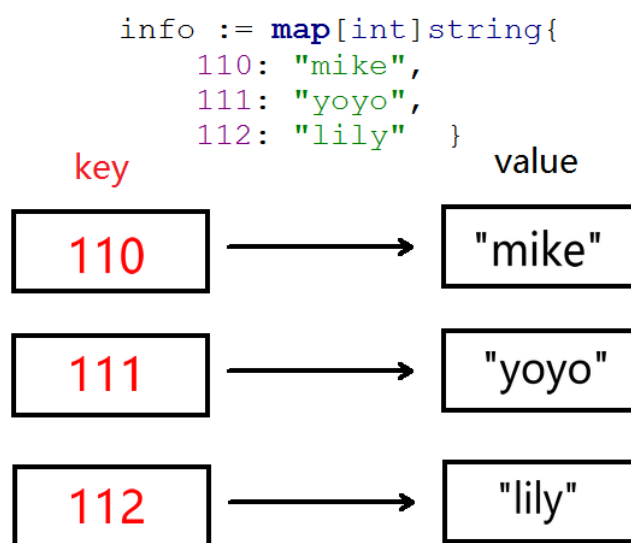


```
}
```

## 7.5 map

### 7.5.1 概述

Go 语言中的 map(映射、字典)是一种内置的数据结构，它是一个**无序**的 key—value 对的集合，比如以身份证号作为唯一键来标识一个人的信息。



map 格式为:

```
map[keyType]valueType
```

在一个 map 里所有的键都是**唯一**的，而且必须是支持==和!=操作符的类型，切片、函数以及包含切片的结构类型这些类型由于具有引用语义，不能作为映射的键，使用这些类型会造成编译错误:

```
dict := map[[]string]int{} //err, invalid map key type []string
```

map 值可以是任意类型，没有限制。map 里所有键的数据类型必须是相同的，值也必须如何，但键和值的数据类型可以不相同。

**注意:** map 是**无序**的，我们无法决定它的返回顺序，所以，每次打印结果的顺利有可能不同。



## 7.5.2 创建和初始化

### 7.5.2.1 map 的创建

```
var m1 map[int]string //只是声明一个 map，没有初始化，此为空(nil)map
fmt.Println(m1 == nil) //true
//m1[1] = "mike" //err, panic: assignment to entry in nil map

//m2, m3 的创建方法是等价的
m2 := map[int]string{}
m3 := make(map[int]string)
fmt.Println(m2, m3) //map[] map[]

m4 := make(map[int]string, 10) //第 2 个参数指定容量
fmt.Println(m4)                //map[]
```

### 7.5.2.2 初始化

```
//1、定义同时初始化
var m1 map[int]string = map[int]string{1: "mike", 2: "yoyo"}
fmt.Println(m1) //map[1:mike 2:yoyo]

//2、自动推导类型 :=
m2 := map[int]string{1: "mike", 2: "yoyo"}
fmt.Println(m2)
```

## 7.5.3 常用操作

### 7.5.3.1 赋值

```
m1 := map[int]string{1: "mike", 2: "yoyo"}
m1[1] = "xxx" //修改
m1[3] = "lily" //追加， go 底层会自动为 map 分配空间
fmt.Println(m1) //map[1:xxx 2:yoyo 3:lily]

m2 := make(map[int]string, 10) //创建 map
m2[0] = "aaa"
m2[1] = "bbb"
fmt.Println(m2)                //map[0:aaa 1:bbb]
fmt.Println(m2[0], m2[1]) //aaa bbb
```



### 7.5.3.2 遍历

```
m1 := map[int]string{1: "mike", 2: "yoyo"}
//迭代遍历 1, 第一个返回值是 key, 第二个返回值是 value
for k, v := range m1 {
    fmt.Printf("%d ----> %s\n", k, v)
    //1 ----> mike
    //2 ----> yoyo
}

//迭代遍历 2, 第一个返回值是 key, 第二个返回值是 value (可省略)
for k := range m1 {
    fmt.Printf("%d ----> %s\n", k, m1[k])
    //1 ----> mike
    //2 ----> yoyo
}

//判断某个 key 所对应的 value 是否存在, 第一个返回值是 value (如果存在的话)
value, ok := m1[1]
fmt.Println("value = ", value, ", ok = ", ok) //value = mike , ok
= true

value2, ok2 := m1[3]
fmt.Println("value2 = ", value2, ", ok2 = ", ok2) //value2 =  , ok2
= false
```

### 7.5.3.3 删除

```
m1 := map[int]string{1: "mike", 2: "yoyo", 3: "lily"}
//迭代遍历 1, 第一个返回值是 key, 第二个返回值是 value
for k, v := range m1 {
    fmt.Printf("%d ----> %s\n", k, v)
    //1 ----> mike
    //2 ----> yoyo
    //3 ----> lily
}

delete(m1, 2) //删除 key 值为 3 的 map

for k, v := range m1 {
    fmt.Printf("%d ----> %s\n", k, v)
    //1 ----> mike
    //3 ----> lily
}
```



```
}
```

## 7.5.4 map 做函数参数

在函数间传递映射并不会制造出该映射的一个副本，不是值传递，而是引用传递：

```
func DeleteMap(m map[int]string, key int) {
    delete(m, key) //删除 key 值为 3 的 map

    for k, v := range m {
        fmt.Printf("len(m)=%d, %d ----> %s\n", len(m), k, v)
        //len(m)=2, 1 ----> mike
        //len(m)=2, 3 ----> lily
    }
}

func main() {
    m := map[int]string{1: "mike", 2: "yoyo", 3: "lily"}

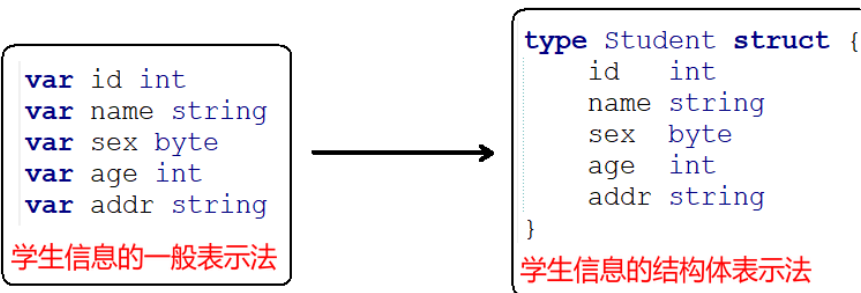
    DeleteMap(m, 2) //删除 key 值为 3 的 map

    for k, v := range m {
        fmt.Printf("len(m)=%d, %d ----> %s\n", len(m), k, v)
        //len(m)=2, 1 ----> mike
        //len(m)=2, 3 ----> lily
    }
}
```

## 7.6 结构体

### 7.6.1 结构体类型

有时我们需要将不同类型的数据组合成一个有机的整体，如：一个学生有学号/姓名/性别/年龄/地址等属性。显然单独定义以上变量比较繁琐，数据不便于管理。



结构体是一种聚合的数据类型，它是由一系列具有相同类型或不同类型的数据构成的数据集合。每个数据称为结构体的成员。

## 7.6.2 结构体初始化

### 7.6.2.1 普通变量

```
type Student struct {
    id    int
    name  string
    sex   byte
    age   int
    addr  string
}

func main() {
    //1、顺序初始化，必须每个成员都初始化
    var s1 Student = Student{1, "mike", 'm', 18, "sz"}
    s2 := Student{2, "yoyo", 'f', 20, "sz"}
    //s3 := Student{2, "tom", 'm', 20} //err, too few values in struct
    initializer

    //2、指定初始化某个成员，没有初始化的成员为零值
    s4 := Student{id: 2, name: "lily"}
}
```

### 7.6.2.2 指针变量

```
type Student struct {
    id    int
    name  string
    sex   byte
    age   int
}
```



```
    addr string
}

func main() {
    var s5 *Student = &Student{3, "xiaoming", 'm', 16, "bj"}
    s6 := &Student{4, "rocco", 'm', 3, "sh"}
}
```

## 7.6.3 结构体成员的使用

### 7.6.3.1 普通变量

```
//=====结构体变量为普通变量
//1、打印成员
var s1 Student = Student{1, "mike", 'm', 18, "sz"}
//结果: id = 1, name = mike, sex = m, age = 18, addr = sz
fmt.Printf("id = %d, name = %s, sex = %c, age = %d, addr = %s\n", s1.id,
s1.name, s1.sex, s1.age, s1.addr)

//2、成员变量赋值
var s2 Student
s2.id = 2
s2.name = "yoyo"
s2.sex = 'f'
s2.age = 16
s2.addr = "guangzhou"
fmt.Println(s2) //{2 yoyo 102 16 guangzhou}
```

### 7.6.3.2 指针变量

```
//=====结构体变量为指针变量
//3、先分配空间，再赋值
s3 := new(Student)
s3.id = 3
s3.name = "xxx"
fmt.Println(s3) //{3 xxx 0 0 }

//4、普通变量和指针变量类型打印
var s4 Student = Student{4, "yyy", 'm', 18, "sz"}
fmt.Printf("s4 = %v, &s4 = %v\n", s4, &s4) //s4 = {4 yyy 109 18 sz},
&s4 = &{4 yyy 109 18 sz}

var p *Student = &s4
```



```
//p.成员 和 (*p).成员 操作是等价的
p.id = 5
(*p).name = "zzz"
fmt.Println(p, *p, s4) //&{5 zzz 109 18 sz} {5 zzz 109 18 sz} {5 zzz 109 18 sz}
```

## 7.6.4 结构体比较

如果结构体的全部成员都是可以比较的，那么结构体也是可以比较的，那样的话两个结构体将可以使用 == 或 != 运算符进行比较，但不支持 > 或 < 。

```
func main() {
    s1 := Student{1, "mike", 'm', 18, "sz"}
    s2 := Student{1, "mike", 'm', 18, "sz"}

    fmt.Println("s1 == s2", s1 == s2) //s1 == s2 true
    fmt.Println("s1 != s2", s1 != s2) //s1 != s2 false
}
```

## 7.6.5 结构体作为函数参数

### 7.6.5.1 值传递

```
func printStudentValue(tmp Student) {
    tmp.id = 250
    //printStudentValue tmp = {250 mike 109 18 sz}
    fmt.Println("printStudentValue tmp = ", tmp)
}

func main() {
    var s Student = Student{1, "mike", 'm', 18, "sz"}

    printStudentValue(s) //值传递，形参的修改不会影响到实参
    fmt.Println("main s = ", s) //main s = {1 mike 109 18 sz}
}
```

### 7.6.5.2 引用传递

```
func printStudentPointer(p *Student) {
    p.id = 250
    //printStudentPointer p = &{250 mike 109 18 sz}
    fmt.Println("printStudentPointer p = ", p)
}
```





```

}

func main() {
    var s Student = Student{1, "mike", 'm', 18, "sz"}

    printStudentPointer(&s)    //引用(地址)传递，形参的修改会影响到实参
    fmt.Println("main s = ", s) //main s = {250 mike 109 18 sz}
}

```

### 7.6.6 可见性

Go 语言对关键字的增加非常吝啬，其中没有 private、protected、public 这样的关键字。

要使某个符号对其他包（package）可见（即可以访问），需要将该符号定义为以**大写字母**开头。

目录结构：

```

v  src
  v  test
    test.go
    main.go

```

test.go 示例代码如下：

```

//test.go
package test

//student01 只能在本文件引用，因为首字母小写
type student01 struct {
    Id    int
    Name  string
}

//Student02 可以在任意文件引用，因为首字母大写
type Student02 struct {
    Id    int
    name  string
}

```

main.go 示例代码如下：

```

// main.go
package main

import (
    "fmt"

```



```
    "test" //导入 test 包
)

func main() {
    //s1 := test.student01{1, "mike"} //err, cannot refer to unexported
name test.student01

    //err, implicit assignment of unexported field 'name' in
test.Student02 literal
    //s2 := test.Student02{2, "yoyo"}
    //fmt.Println(s2)

    var s3 test.Student02 //声明变量
    s3.Id = 1             //ok
    //s3.name = "mike"    //err, s3.name undefined (cannot refer to
unexported field or method name)
    fmt.Println(s3)
}
```

## 8. 面向对象编程

### 8.1 概述

对于面向对象编程的支持 Go 语言设计得非常简洁而优雅。因为，Go 语言并没有沿袭传统面向对象编程中的诸多概念，比如继承(不支持继承，尽管匿名字段的内存布局和行为类似继承，但它并不是继承)、虚函数、构造函数和析构函数、隐藏的 this 指针等。

尽管 Go 语言中没有封装、继承、多态这些概念，但同样通过别的方式实现这些特性：

- 封装：通过方法实现
- 继承：通过匿名字段实现
- 多态：通过接口实现

### 8.2 匿名组合

#### 8.2.1 匿名字段

一般情况下，定义结构体的时候是字段名与其类型一一对应，实际上 Go 支持只**提供类型，而不写字段名的方式**，也就是匿名字段，也称为嵌入字段。

当匿名字段也是一个结构体的时候，那么这个结构体所拥有的全部字段都被隐式地引入了当



前定义的这个结构体。

```
//人
type Person struct {
    name string
    sex  byte
    age  int
}

//学生
type Student struct {
    Person // 匿名字段，那么默认 Student 就包含了 Person 的所有字段
    id     int
    addr   string
}
```

## 8.2.2 初始化

```
//人
type Person struct {
    name string
    sex  byte
    age  int
}

//学生
type Student struct {
    Person // 匿名字段，那么默认 Student 就包含了 Person 的所有字段
    id     int
    addr   string
}

func main() {
    //顺序初始化
    s1 := Student{Person{"mike", 'm', 18}, 1, "sz"}
    //s1 = {Person:{name:mike sex:109 age:18} id:1 addr:sz}
    fmt.Printf("s1 = %+v\n", s1)

    //s2 := Student{"mike", 'm', 18, 1, "sz"} //err

    //部分成员初始化 1
    s3 := Student{Person: Person{"lily", 'f', 19}, id: 2}
    //s3 = {Person:{name:lily sex:102 age:19} id:2 addr:}
    fmt.Printf("s3 = %+v\n", s3)

    //部分成员初始化 2
    s4 := Student{Person: Person{name: "tom"}, id: 3}
```



```
//s4 = {Person:{name:tom sex:0 age:0} id:3 addr:}
fmt.Printf("s4 = %+v\n", s4)
}
```

### 8.2.3 成员的操作

```
var s1 Student //变量声明
//给成员赋值
s1.name = "mike" //等价于 s1.Person.name = "mike"
s1.sex = 'm'
s1.age = 18
s1.id = 1
s1.addr = "sz"
fmt.Println(s1) //{mike 109 18} 1 sz}

var s2 Student //变量声明
s2.Person = Person{"lily", 'f', 19}
s2.id = 2
s2.addr = "bj"
fmt.Println(s2) //{lily 102 19} 2 bj}
```

### 8.2.4 同名字段

```
//人
type Person struct {
    name string
    sex  byte
    age  int
}

//学生
type Student struct {
    Person // 匿名字段，那么默认 Student 就包含了 Person 的所有字段
    id     int
    addr   string
    name   string //和 Person 中的 name 同名
}

func main() {
    var s Student //变量声明

    //给 Student 的 name，还是给 Person 赋值？
    s.name = "mike"
}
```



```
//{Person:{name: sex:0 age:0} id:0 addr: name:mike}
fmt.Printf("%+v\n", s)

//默认只会给最外层的成员赋值
//给匿名同名成员赋值，需要显示调用
s.Person.name = "yoyo"
//Person:{name:yoyo sex:0 age:0} id:0 addr: name:mike}
fmt.Printf("%+v\n", s)
}
```

## 8.2.5 其它匿名字段

### 8.2.5.1 非结构体类型

所有的内置类型和自定义类型都可以作为匿名字段的：

```
type mystr string //自定义类型

type Person struct {
    name string
    sex  byte
    age  int
}

type Student struct {
    Person // 匿名字段，结构体类型
    int    // 匿名字段，内置类型
    mystr  // 匿名字段，自定义类型
}

func main() {
    //初始化
    s1 := Student{Person{"mike", 'm', 18}, 1, "bj"}

    //{Person:{name:mike sex:109 age:18} int:1 mystr:bj}
    fmt.Printf("%+v\n", s1)

    //成员的操作，打印结果: mike, m, 18, 1, bj
    fmt.Printf("%s, %c, %d, %d, %s\n", s1.name, s1.sex, s1.age, s1.int,
s1.mystr)
}
```



### 8.2.5.2 结构体指针类型

```
type Person struct { //人
    name string
    sex  byte
    age  int
}

type Student struct { //学生
    *Person // 匿名字段，结构体指针类型
    id      int
    addr    string
}

func main() {
    //初始化
    s1 := Student{&Person{"mike", 'm', 18}, 1, "bj"}

    //{Person:0xc0420023e0 id:1 addr:bj}
    fmt.Printf("%+v\n", s1)
    //mike, m, 18
    fmt.Printf("%s, %c, %d\n", s1.name, s1.sex, s1.age)

    //声明变量
    var s2 Student
    s2.Person = new(Person) //分配空间
    s2.name = "yoyo"
    s2.sex = 'f'
    s2.age = 20

    s2.id = 2
    s2.addr = "sz"

    //yoyo 102 20 2 20
    fmt.Println(s2.name, s2.sex, s2.age, s2.id, s2.age)
}
```

## 8.3 方法

### 8.3.1 概述

在面向对象编程中，一个对象其实也就是一个简单的值或者一个变量，在这个对象中会包含一些函数，**这种带有接收者的函数，我们称为方法(method)**。本质上，一个方法则是一个



和特殊类型关联的函数。

一个面向对象的程序会用方法来表达其属性和对应的操作，这样使用这个对象的用户就不需要直接去操作对象，而是借助方法来做这些事情。

在 Go 语言中，可以给任意**自定义类型**（包括内置类型，但不包括指针类型）添加相应的方法。

方法总是绑定对象实例，并隐式将实例作为第一实参 (receiver)，方法的语法如下：

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

- 参数 receiver 可任意命名。如方法中未曾使用，可省略参数名。
- 参数 receiver 类型可以是 T 或 \*T。基类型 T 不能是接口或指针。
- 不支持重载方法，也就是说，不能定义名字相同但是不同参数的方法。

## 8.3.2 为类型添加方法

### 8.3.2.1 基础类型作为接收者

```
type MyInt int //自定义类型，给 int 改名为 MyInt

//在函数定义时，在其名字之前放上一个变量，即是一个方法
func (a MyInt) Add(b MyInt) MyInt { //面向对象
    return a + b
}

//传统方式的定义
func Add(a, b MyInt) MyInt { //面向过程
    return a + b
}

func main() {
    var a MyInt = 1
    var b MyInt = 1

    //调用 func (a MyInt) Add(b MyInt)
    fmt.Println("a.Add(b) = ", a.Add(b)) //a.Add(b) = 2

    //调用 func Add(a, b MyInt)
    fmt.Println("Add(a, b) = ", Add(a, b)) //Add(a, b) = 2
}
```

通过上面的例子可以看出，**面向对象只是换了一种语法形式来表达**。方法是函数的语法糖，



因为 receiver 其实就是方法所接收的第 1 个参数。

注意：虽然方法的名字一模一样，但是如果接收者不一样，那么方法就不一样。

### 8.3.2.2 结构体作为接收者

方法里面可以访问接收者的字段，调用方法通过点(.)访问，就像 struct 里面访问字段一样：

```
type Person struct {  
    name string  
    sex  byte  
    age  int  
}  
  
func (p Person) PrintInfo() { //给 Person 添加方法  
    fmt.Println(p.name, p.sex, p.age)  
}  
  
func main() {  
    p := Person{"mike", 'm', 18} //初始化  
    p.PrintInfo() //调用 func (p Person) PrintInfo()  
}
```

### 8.3.3 值语义和引用语义

```
type Person struct {  
    name string  
    sex  byte  
    age  int  
}  
  
// 指针作为接收者，引用语义  
func (p *Person) SetInfoPointer() {  
    //给成员赋值  
    (*p).name = "yoyo"  
    p.sex = 'f'  
    p.age = 22  
}  
  
// 值作为接收者，值语义  
func (p Person) SetInfoValue() {  
    //给成员赋值  
    p.name = "yoyo"  
    p.sex = 'f'  
}
```





```
p.age = 22
}

func main() {
    //指针作为接收者，引用语义
    p1 := Person{"mike", 'm', 18} //初始化
    fmt.Println("函数调用前 = ", p1) //函数调用前 = {mike 109 18}
    (&p1).SetInfoPointer()
    fmt.Println("函数调用后 = ", p1) //函数调用后 = {yoyo 102 22}

    fmt.Println("=====")

    p2 := Person{"mike", 'm', 18} //初始化
    //值作为接收者，值语义
    fmt.Println("函数调用前 = ", p2) //函数调用前 = {mike 109 18}
    p2.SetInfoValue()
    fmt.Println("函数调用后 = ", p2) //函数调用后 = {mike 109 18}
}
```

## 8.3.4 方法集

类型的方法集是指可以被该类型的值调用的所有方法的集合。

用实例实例 `value` 和 `pointer` 调用方法（含匿名字段）不受方法集约束，编译器编总是查找全部方法，并自动转换 `receiver` 实参。

### 8.3.4.1 类型 `*T` 方法集

一个指向自定义类型的值的指针，它的方法集由该类型定义的所有方法组成，无论这些方法接受的是一个值还是一个指针。

如果在指针上调用一个接受值的方法，Go 语言会聪明地将该指针解引用，并将指针所指的底层值作为方法的接收者。

类型 `*T` 方法集包含全部 `receiver T + *T` 方法：

```
type Person struct {
    name string
    sex  byte
    age  int
}
```



```
//指针作为接收者，引用语义
func (p *Person) SetInfoPointer() {
    (*p).name = "yoyo"
    p.sex = 'f'
    p.age = 22
}

//值作为接收者，值语义
func (p Person) SetInfoValue() {
    p.name = "xxx"
    p.sex = 'm'
    p.age = 33
}

func main() {
    //p 为指针类型
    var p *Person = &Person{"mike", 'm', 18}
    p.SetInfoPointer() //func (p) SetInfoPointer()

    p.SetInfoValue()   //func (*p) SetInfoValue()
    (*p).SetInfoValue() //func (*p) SetInfoValue()
}
```

#### 8.3.4.2 类型 T 方法集

一个自定义类型值的方法集则由为该类型定义的接收者类型为值类型的方法组成，但是不包含那些接收者类型为指针的方法。

但这种限制通常并不像这里所说的那样，因为如果我们只有一个值，仍然可以调用一个接收者为指针类型的方法，这可以借助于 Go 语言传值的地址能力实现。

```
type Person struct {
    name string
    sex  byte
    age  int
}

//指针作为接收者，引用语义
func (p *Person) SetInfoPointer() {
    (*p).name = "yoyo"
    p.sex = 'f'
    p.age = 22
}
```



```
//值作为接收者，值语义
func (p Person) SetInfoValue() {
    p.name = "xxx"
    p.sex = 'm'
    p.age = 33
}

func main() {
    //p 为普通值类型
    var p Person = Person{"mike", 'm', 18}
    (&p).SetInfoPointer() //func (&p) SetInfoPointer()
    p.SetInfoPointer()    //func (&p) SetInfoPointer()

    p.SetInfoValue()      //func (p) SetInfoValue()
    (&p).SetInfoValue()    //func (*p) SetInfoValue()
}
```

### 8.3.5 匿名字段

#### 8.3.5.1 方法的继承

如果匿名字段实现了一个方法，那么包含这个匿名字段的 struct 也能调用该方法。

```
type Person struct {
    name string
    sex  byte
    age  int
}

//Person 定义了方法
func (p *Person) PrintInfo() {
    fmt.Printf("%s,%c,%d\n", p.name, p.sex, p.age)
}

type Student struct {
    Person // 匿名字段，那么 Student 包含了 Person 的所有字段
    id     int
    addr   string
}

func main() {
    p := Person{"mike", 'm', 18}
    p.PrintInfo()
}
```



```
s := Student{Person{"yoyo", 'f', 20}, 2, "sz"}
s.PrintInfo()
}
```

### 8.3.5.2 方法的重写

```
type Person struct {
    name string
    sex  byte
    age  int
}

//Person 定义了方法
func (p *Person) PrintInfo() {
    fmt.Printf("Person: %s,%c,%d\n", p.name, p.sex, p.age)
}

type Student struct {
    Person // 匿名字段，那么 Student 包含了 Person 的所有字段
    id     int
    addr   string
}

//Student 定义了方法
func (s *Student) PrintInfo() {
    fmt.Printf("Student: %s,%c,%d\n", s.name, s.sex, s.age)
}

func main() {
    p := Person{"mike", 'm', 18}
    p.PrintInfo() //Person: mike,m,18

    s := Student{Person{"yoyo", 'f', 20}, 2, "sz"}
    s.PrintInfo() //Student: yoyo,f,20
    s.Person.PrintInfo() //Person: yoyo,f,20
}
```

### 8.3.6 表达式

类似于我们可以对函数进行赋值和传递一样，方法也可以进行赋值和传递。

根据调用者不同，方法分为两种表现形式：方法值和方法表达式。两者都可像普通函数那样



赋值和传参，**区别**在于方法值绑定实例，而方法表达式则须显式传参。

### 8.3.6.1 方法值

```
type Person struct {
    name string
    sex  byte
    age  int
}

func (p *Person) PrintInfoPointer() {
    fmt.Printf("%p, %v\n", p, p)
}

func (p Person) PrintInfoValue() {
    fmt.Printf("%p, %v\n", &p, p)
}

func main() {
    p := Person{"mike", 'm', 18}
    p.PrintInfoPointer() //0xc0420023e0, &{mike 109 18}

    pFunc1 := p.PrintInfoPointer //方法值，隐式传递 receiver
    pFunc1()                      //0xc0420023e0, &{mike 109 18}

    pFunc2 := p.PrintInfoValue
    pFunc2() //0xc042048420, {mike 109 18}
}
```

### 8.3.6.2 方法表达式

```
type Person struct {
    name string
    sex  byte
    age  int
}

func (p *Person) PrintInfoPointer() {
    fmt.Printf("%p, %v\n", p, p)
}

func (p Person) PrintInfoValue() {
```



```
fmt.Printf("%p, %v\n", &p, p)
}

func main() {
    p := Person{"mike", 'm', 18}
    p.PrintInfoPointer() //0xc0420023e0, &{mike 109 18}

    //方法表达式， 须显式传参
    //func pFunc1(p *Person))
    pFunc1 := (*Person).PrintInfoPointer
    pFunc1(&p) //0xc0420023e0, &{mike 109 18}

    pFunc2 := Person.PrintInfoValue
    pFunc2(p) //0xc042002460, {mike 109 18}
}
```

## 8.4 接口

### 8.4.1 概述

在 Go 语言中，接口(interface)是一个自定义类型，接口类型具体描述了一系列方法的集合。

接口类型是一种抽象的类型，它不会暴露出它所代表的对象的内部值的结构和这个对象支持的基础操作的集合，它们只会展示出它们自己的方法。**因此接口类型不能将其实例化。**

Go 通过接口实现了鸭子类型(duck-typing): “当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子”。我们并不关心对象是什么类型，到底是不是鸭子，只关心行为。

### 8.4.2 接口的使用

#### 8.4.2.1 接口定义

```
type Humaner interface {
    SayHi ()
}
```

- 接口命名习惯以 er 结尾
- 接口只有方法声明，没有实现，没有数据字段
- 接口可以匿名嵌入其它接口，或嵌入到结构中



#### 8.4.2.2 接口实现

接口是用来定义行为的类型。这些被定义的行为不由接口直接实现，而是通过方法由用户定义的类型实现，一个实现了这些方法的具体类型是这个接口类型的实例。

如果用户定义的类型实现了某个接口类型声明的一组方法，那么这个用户定义的类型值就可以赋给这个接口类型的值。这个赋值会把用户定义的类型值存入接口类型的值。

```
type Humaner interface {
    SayHi()
}

type Student struct { //学生
    name string
    score float64
}

//Student 实现 SayHi() 方法
func (s *Student) SayHi() {
    fmt.Printf("Student[%s, %f] say hi!!\n", s.name, s.score)
}

type Teacher struct { //老师
    name string
    group string
}

//Teacher 实现 SayHi() 方法
func (t *Teacher) SayHi() {
    fmt.Printf("Teacher[%s, %s] say hi!!\n", t.name, t.group)
}

type MyStr string

//MyStr 实现 SayHi() 方法
func (str MyStr) SayHi() {
    fmt.Printf("MyStr[%s] say hi!!\n", str)
}

//普通函数，参数为 Humaner 类型的变量 i
func WhoSayHi(i Humaner) {
    i.SayHi()
}
```



```
func main() {
    s := &Student{"mike", 88.88}
    t := &Teacher{"yoyo", "Go 语言"}
    var tmp MyStr = "测试"

    s.SayHi() //Student[mike, 88.880000] say hi!!
    t.SayHi() //Teacher[yoyo, Go 语言] say hi!!
    tmp.SayHi() //MyStr[测试] say hi!!

    //多态，调用同一接口，不同表现
    WhoSayHi(s) //Student[mike, 88.880000] say hi!!
    WhoSayHi(t) //Teacher[yoyo, Go 语言] say hi!!
    WhoSayHi(tmp) //MyStr[测试] say hi!!

    x := make([]Humaner, 3)
    //这三个都是不同类型的元素，但是他们实现了 interface 同一个接口
    x[0], x[1], x[2] = s, t, tmp
    for _, value := range x {
        value.SayHi()
    }
    /*
        Student[mike, 88.880000] say hi!!
        Teacher[yoyo, Go 语言] say hi!!
        MyStr[测试] say hi!!
    */
}
```

通过上面的代码，你会发现接口就是一组抽象方法的集合，它必须由其他非接口类型实现，而不能自我实现。

## 8.4.3 接口组合

### 8.4.3.1 接口嵌入

如果一个 interface1 作为 interface2 的一个嵌入字段，那么 interface2 隐式的包含了 interface1 里面的方法。

```
type Humaner interface {
    SayHi()
}

type Personer interface {
    Humaner //这里想写了 SayHi() 一样
    Sing(lyrics string)
}
```





```
type Student struct { //学生
    name string
    score float64
}

//Student 实现 SayHi() 方法
func (s *Student) SayHi() {
    fmt.Printf("Student[%s, %f] say hi!!\n", s.name, s.score)
}

//Student 实现 Sing() 方法
func (s *Student) Sing(lyrics string) {
    fmt.Printf("Student sing[%s]!!\n", lyrics)
}

func main() {
    s := &Student{"mike", 88.88}

    var i2 Personer
    i2 = s
    i2.SayHi()    //Student[mike, 88.880000] say hi!!
    i2.Sing("学生哥") //Student sing[学生哥]!!
}
```

#### 8.4.3.2 接口转换

超集接口对象可转换为子集接口，反之出错：

```
type Humaner interface {
    SayHi()
}

type Personer interface {
    Humaner //这里像写了 SayHi() 一样
    Sing(lyrics string)
}

type Student struct { //学生
    name string
    score float64
}
```



```
//Student 实现 SayHi() 方法
func (s *Student) SayHi() {
    fmt.Printf("Student[%s, %f] say hi!!\n", s.name, s.score)
}

//Student 实现 Sing() 方法
func (s *Student) Sing(lyrics string) {
    fmt.Printf("Student sing[%s]!!\n", lyrics)
}

func main() {
    //var i1 Humaner = &Student{"mike", 88.88}
    //var i2 Personer = i1 //err

    //Personer 为超集, Humaner 为子集
    var i1 Personer = &Student{"mike", 88.88}
    var i2 Humaner = i1
    i2.SayHi() //Student[mike, 88.880000] say hi!!
}
```

#### 8.4.4 空接口

空接口(interface{})不包含任何的方法，正因为如此，所有的类型都实现了空接口，因此空接口可以存储任意类型的数值。它有点类似于 C 语言的 void \*类型。

```
var v1 interface{} = 1 // 将 int 类型赋值给 interface{}
var v2 interface{} = "abc" // 将 string 类型赋值给 interface{}
var v3 interface{} = &v2 // 将*interface{}类型赋值给 interface{}
var v4 interface{} = struct{ X int }{1}
var v5 interface{} = &struct{ X int }{1}
```

当函数可以接受任意的对象实例时，我们会将其声明为 interface{}，最典型的例子是标准库 fmt 中 PrintXXX 系列的函数，例如：

```
func Printf(fmt string, args ...interface{})
func Println(args ...interface{})
```

#### 8.4.5 类型查询

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

- comma-ok 断言
- switch 测试



#### 8.4.5.1 comma-ok 断言

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

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

示例代码：

```
type Element interface{}

type Person struct {
    name string
    age  int
}

func main() {
    list := make([]Element, 3)
    list[0] = 1           // an int
    list[1] = "Hello"     // a string
    list[2] = Person{"mike", 18}

    for index, element := range list {
        if value, ok := element.(int); ok {
            fmt.Printf("list[%d] is an int and its value is %d\n", index,
value)
        } else if value, ok := element.(string); ok {
            fmt.Printf("list[%d] is a string and its value is %s\n",
index, value)
        } else if value, ok := element.(Person); ok {
            fmt.Printf("list[%d] is a Person and its value is [%s, %d]\n",
index, value.name, value.age)
        } else {
            fmt.Printf("list[%d] is of a different type\n", index)
        }
    }

    /* 打印结果:
list[0] is an int and its value is 1
list[1] is a string and its value is Hello
list[2] is a Person and its value is [mike, 18]
*/
}
```



### 8.4.5.2 switch 测试

```
type Element interface{}

type Person struct {
    name string
    age  int
}

func main() {
    list := make([]Element, 3)
    list[0] = 1 //an int
    list[1] = "Hello" //a string
    list[2] = Person{"mike", 18}

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

## 9. 异常处理

### 9.1 error 接口

Go 语言引入了一个关于错误处理的标准模式，即 error 接口，它是 Go 语言内建的接口类型，该接口的定义如下：

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



Go 语言的标准库代码包 errors 为用户提供如下方法：

```
package errors

type errorString struct {
    text string
}

func New(text string) error {
    return &errorString{text}
}

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

另一个可以生成 error 类型值的方法是调用 fmt 包中的 Errorf 函数：

```
package fmt
import "errors"

func Errorf(format string, args ...interface{}) error {
    return errors.New(Sprintf(format, args...))
}
```

示例代码：

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

func main() {
    var err1 error = errors.New("a normal err1")
    fmt.Println(err1) //a normal err1

    var err2 error = fmt.Errorf("%s", "a normal err2")
    fmt.Println(err2) //a normal err2
}
```

函数通常在最后的返回值中返回错误信息：

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



```
func Divide(a, b float64) (result float64, err error) {
    if b == 0 {
        result = 0.0
        err = errors.New("runtime error: divide by zero")
        return
    }

    result = a / b
    err = nil
    return
}

func main() {
    r, err := Divide(10.0, 0)
    if err != nil {
        fmt.Println(err) //错误处理 runtime error: divide by zero
    } else {
        fmt.Println(r) // 使用返回值
    }
}
```

## 9.2 panic

在通常情况下，向程序使用方报告错误状态的方式可以是返回一个额外的 error 类型值。

但是，当遇到不可恢复的错误状态的时候，如数组访问越界、空指针引用等，这些运行时错误会引起 panic 异常。这时，上述错误处理方式显然就不适合了。反过来讲，在一般情况下，我们不应通过调用 panic 函数来报告普通的错误，而应该只把它作为报告致命错误的一种方式。当某些不应该发生的场景发生时，我们就应该调用 panic。

一般而言，当 panic 异常发生时，程序会中断运行，并立即执行在该 goroutine（可以先理解成线程，在中被延迟的函数（defer 机制）。随后，程序崩溃并输出日志信息。日志信息包括 panic value 和函数调用的堆栈跟踪信息。

不是所有的 panic 异常都来自运行时，直接调用内置的 panic 函数也会引发 panic 异常；panic 函数接受任何值作为参数。

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

调用 panic 函数引发的 panic 异常：

```
func TestA() {
    fmt.Println("func TestA()")
}
```



```
func TestB() {  
    panic("func TestB(): panic")  
}  
  
func TestC() {  
    fmt.Println("func TestC()")  
}  
  
func main() {  
    TestA()  
    TestB() //TestB() 发生异常，中断程序  
    TestC()  
}
```

运行结果：

```
func TestA()  
panic: func TestB(): panic  
  
goroutine 1 [running]:  
main.TestB()  
    C:/Users/superman/Desktop/code/go/src/main.go:13 +0x40  
main.main()  
    C:/Users/superman/Desktop/code/go/src/main.go:22 +0x2c  
错误：进程退出代码 2.
```

内置的 panic 函数引发的 panic 异常：

```
func TestA() {  
    fmt.Println("func TestA()")  
}  
  
func TestB(x int) {  
    var a [10]int  
    a[x] = 222 //x 值为 11 时，数组越界  
}  
  
func TestC() {  
    fmt.Println("func TestC()")  
}  
  
func main() {  
    TestA()  
    TestB(11) //TestB() 发生异常，中断程序  
    TestC()  
}
```



运行结果：

```
func TestA()
panic: runtime error: index out of range

goroutine 1 [running]:
main.TestB(...)
    C:/Users/superman/Desktop/code/go/src/main.go:14
main.main()
    C:/Users/superman/Desktop/code/go/src/main.go:23 +0x4c
错误：进程退出代码 2.
```

## 9.3 recover

运行时 panic 异常一旦被引发就会导致程序崩溃。这当然不是我们愿意看到的，因为谁也不能保证程序不会发生任何运行时错误。

不过，Go 语言为我们提供了专用于“拦截”运行时 panic 的内建函数——recover。它可以是当前的程序从运行时 panic 的状态中恢复并重新获得流程控制权。

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

注意：recover 只有在 defer 调用的函数中有效。

如果调用了内置函数 recover，并且定义该 defer 语句的函数发生了 panic 异常，recover 会使程序从 panic 中恢复，并返回 panic value。导致 panic 异常的函数不会继续运行，但能正常返回。在未发生 panic 时调用 recover，recover 会返回 nil。

示例代码：

```
func TestA() {
    fmt.Println("func TestA()")
}

func TestB() (err error) {
    defer func() { //在发生异常时，设置恢复
        if x := recover(); x != nil {
            //panic value 被附加到错误信息中；
            //并用 err 变量接收错误信息，返回给调用者。
            err = fmt.Errorf("internal error: %v", x)
        }
    }()

    panic("func TestB(): panic")
}
```





```
func TestC() {
    fmt.Println("func TestC()")
}

func main() {
    TestA()
    err := TestB()
    fmt.Println(err)
    TestC()

    /*
        运行结果:
        func TestA()
        internal error: func TestB(): panic
        func TestC()
    */
}
```

延迟调用中引发的错误，可被后续延迟调用捕获，但仅最后一个错误可被捕获：

```
func test() {
    defer func() {
        fmt.Println(recover())
    }()

    defer func() {
        panic("defer panic")
    }()

    panic("test panic")
}

func main() {
    test()
    //运行结果: defer panic
}
```



## 10. 文本文件处理

### 10.1 字符串处理

字符串在开发中经常用到，包括用户的输入，数据库读取的数据等，我们经常需要对字符串进行分割、连接、转换等操作，我们可以通过 Go 标准库中的 `strings` 和 `strconv` 两个包中的函数进行相应的操作。

#### 10.1.1 字符串操作

下面这些函数来自于 `strings` 包，这里介绍一些我平常经常用到的函数，更详细的请参考官方的文档。

##### 10.1.1.1 Contains

```
func Contains(s, substr string) bool
```

功能：字符串 `s` 中是否包含 `substr`，返回 `bool` 值

示例代码：

```
fmt.Println(strings.Contains("seafood", "foo"))
fmt.Println(strings.Contains("seafood", "bar"))
fmt.Println(strings.Contains("seafood", ""))
fmt.Println(strings.Contains("", ""))
//运行结果：
//true
//false
//true
//true
```

##### 10.1.1.2 Join

```
func Join(a []string, sep string) string
```

功能：字符串链接，把 slice `a` 通过 `sep` 链接起来

示例代码：

```
s := []string{"foo", "bar", "baz"}
fmt.Println(strings.Join(s, ", "))
//运行结果:foo, bar, baz
```



### 10.1.1.3 Index

```
func Index(s, sep string) int
```

功能：在字符串 s 中查找 sep 所在的位置，返回位置值，找不到返回-1

示例代码：

```
fmt.Println(strings.Index("chicken", "ken"))  
fmt.Println(strings.Index("chicken", "dmr"))  
//运行结果：  
//    4  
//   -1
```

### 10.1.1.4 Repeat

```
func Repeat(s string, count int) string
```

功能：重复 s 字符串 count 次，最后返回重复的字符串

示例代码：

```
fmt.Println("ba" + strings.Repeat("na", 2))  
//运行结果:banana
```

### 10.1.1.5 Replace

```
func Replace(s, old, new string, n int) string
```

功能：在 s 字符串中，把 old 字符串替换为 new 字符串，n 表示替换的次数，小于 0 表示全部替换

示例代码：

```
fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))  
fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))  
//运行结果：  
//oinky oinky oink  
//moo moo moo
```

### 10.1.1.6 Split

```
func Split(s, sep string) []string
```

功能：把 s 字符串按照 sep 分割，返回 slice

示例代码：

```
fmt.Printf("%q\n", strings.Split("a,b,c", ","))
```



```
fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a"))
fmt.Printf("%q\n", strings.Split(" xyz ", ""))
fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
//运行结果:
//[ "a" "b" "c"]
//[ "" "man " "plan " "canal panama"]
//[ " " "x" "y" "z" " "]
//[ "" ]
```

#### 10.1.1.7 Trim

```
func Trim(s string, cutset string) string
```

功能：在 s 字符串的头部和尾部去除 cutset 指定的字符串

示例代码：

```
fmt.Printf("[%q]", strings.Trim(" !!! Achtung !!! ", "! "))
//运行结果: ["Achtung"]
```

#### 10.1.1.8 Fields

```
func Fields(s string) []string
```

功能：去除 s 字符串的空格符，并且按照空格分割返回 slice

示例代码：

```
fmt.Printf("Fields are: %q", strings.Fields(" foo bar baz "))
//运行结果:Fields are: ["foo" "bar" "baz"]
```

### 10.1.2 字符串转换

字符串转化的函数在 strconv 中，如下也只是列出一些常用的。

#### 10.1.2.1 Append

Append 系列函数将整数等转换为字符串后，添加到现有的字节数组中。

示例代码：

```
str := make([]byte, 0, 100)
str = strconv.AppendInt(str, 4567, 10) //以 10 进制方式追加
str = strconv.AppendBool(str, false)
str = strconv.AppendQuote(str, "abcdefg")
```



```
str = strconv.AppendQuoteRune(str, '单')

fmt.Println(string(str)) //4567false"abcdefg"'单'
```

### 10.1.2.2 Format

Format 系列函数把其他类型的转换为字符串。

示例代码：

```
a := strconv.FormatBool(false)
b := strconv.FormatInt(1234, 10)
c := strconv.FormatUint(12345, 10)
d := strconv.Itoa(1023)

fmt.Println(a, b, c, d) //false 1234 12345 1023
```

### 10.1.2.3 Parse

Parse 系列函数把字符串转换为其他类型。

示例代码：

```
package main

import (
    "fmt"
    "strconv"
)

func checkError(e error) {
    if e != nil {
        fmt.Println(e)
    }
}

func main() {
    a, err := strconv.ParseBool("false")
    checkError(err)
    b, err := strconv.ParseFloat("123.23", 64)
    checkError(err)
    c, err := strconv.ParseInt("1234", 10, 64)
    checkError(err)
    d, err := strconv.ParseUint("12345", 10, 64)
    checkError(err)
    e, err := strconv.Atoi("1023")
    checkError(err)
```



```
fmt.Println(a, b, c, d, e) //false 123.23 1234 12345 1023
}
```

## 10.2 正则表达式

正则表达式是一种进行模式匹配和文本操纵的复杂而又强大的工具。虽然正则表达式比纯粹的文本匹配效率低，但是它却更灵活。按照它的语法规则，按需构造出的匹配模式就能够从原始文本中筛选出几乎任何你想要得到的字符组合。

Go 语言通过 `regexp` 标准包为正则表达式提供了官方支持，如果你已经使用过其他编程语言提供的正则相关功能，那么你应该对 Go 语言版本的不会太陌生，但是它们之间也有一些小的差异，因为 Go 实现的是 RE2 标准，除了 \C，详细的语法描述参考：  
<http://code.google.com/p/re2/wiki/Syntax>

其实字符串处理我们可以使用 `strings` 包来进行搜索(Contains、Index)、替换(Replace)和解析(Split、Join)等操作，但是这些都是简单的字符串操作，他们的搜索都是大小写敏感，而且固定的字符串，如果我们需要匹配可变的那种就没办法实现了，当然如果 `strings` 包能解决你的问题，那么就尽量使用它来解决。因为他们足够简单、而且性能和可读性都会比正则好。

示例代码：

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    context1 := "3.14 123123 .68 haha 1.0 abc 6.66 123."

    //MustCompile 解析并返回一个正则表达式。如果成功返回，该 Regexp 就可用于匹配文本。
    //解析失败时会产生 panic
    // \d 匹配数字[0-9]，d+ 重复>=1次匹配 d，越多越好（优先重复匹配 d）
    exp1 := regexp.MustCompile(`\d+\.\d+`)

    //返回保管正则表达式所有不重叠的匹配结果的[]string切片。如果没有匹配到，会返回 nil。
    //result1 := exp1.FindAllString(context1, -1) //[3.14 1.0 6.66]
    result1 := exp1.FindAllStringSubmatch(context1, -1) //[[3.14] [1.0] [6.66]]

    fmt.Printf("%v\n", result1)
```



```
fmt.Printf("\n-----\n\n")

context2 := `
<title>标题</title>
<div>你过来啊</div>
<div>hello mike</div>
<div>你大爷</div>
<body>呵呵</body>
`

// (.*) 被括起来的表达式作为分组
// 匹配<div>xxx</div>模式的所有子串
exp2 := regexp.MustCompile(`<div>(.*?)</div>`)
result2 := exp2.FindAllStringSubmatch(context2, -1)

// [[<div>你过来啊</div> 你过来啊] [<div>hello mike</div> hello mike]
// [<div>你大爷</div> 你大爷]]
fmt.Printf("%v\n", result2)
fmt.Printf("\n-----\n\n")

context3 := `
<title>标题</title>
<div>你过来啊</div>
<div>hello
mike
go</div>
<div>你大爷</div>
<body>呵呵</body>
`

exp3 := regexp.MustCompile(`<div>(.*?)</div>`)
result3 := exp3.FindAllStringSubmatch(context3, -1)

// [[<div>你过来啊</div> 你过来啊] [<div>你大爷</div> 你大爷]]
fmt.Printf("%v\n", result3)
fmt.Printf("\n-----\n\n")

context4 := `
<title>标题</title>
<div>你过来啊</div>
<div>hello
mike
go</div>
<div>你大爷</div>
<body>呵呵</body>
`
```



```
exp4 := regexp.MustCompile(`<div>(?:s:.*?)</div>`)
result4 := exp4.FindAllStringSubmatch(context4, -1)

/*
    [[<div>你过来啊</div> 你过来啊] [<div>hello
        mike
        go</div> hello
        mike
        go] [<div>你大爷</div> 你大爷]]
*/
fmt.Printf("%v\n", result4)
fmt.Printf("\n-----\n\n")

for _, text := range result4 {
    fmt.Println(text[0]) //带有 div
    fmt.Println(text[1]) //不带带有 div
    fmt.Println("=====\n")
}
}
```

## 10.3 JSON 处理

JSON（JavaScript Object Notation）是一种比 XML 更轻量级的数据交换格式，在易于人们阅读和编写的时候，也易于程序解析和生成。尽管 JSON 是 JavaScript 的一个子集，但 JSON 采用完全独立于编程语言的文本格式，且表现为键/值对集合的文本描述形式（类似一些编程语言中的字典结构），这使它成为较为理想的、跨平台、跨语言的数据交换语言。

```
{
    "Company": "itcast",
    "Subjects": [
        "Go",
        "C++",
        "Python",
        "Test"
    ],
    "IsOk": true,
    "Price": 666
}
```

开发者可以用 JSON 传输简单的字符串、数字、布尔值，也可以传输一个数组，或者一个更复杂的复合结构。在 Web 开发领域中，JSON 被广泛应用于 Web 服务端程序和客户端之间的数据通信。

Go 语言内建对 JSON 的支持。使用 Go 语言内置的 encoding/json 标准库，开发者可以轻松使用 Go 程序生成和解析 JSON 格式的数据。





JSON 官方网站: <http://www.json.org/>

在线格式化: <http://www.json.cn/>

## 10.3.1 编码 JSON

### 10.3.1.1 通过结构体生成 JSON

使用 `json.Marshal()` 函数可以对一组数据进行 JSON 格式的编码。 `json.Marshal()` 函数的声明如下:

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

还有一个格式化输出:

```
// MarshalIndent 很像 Marshal, 只是用缩进对输出进行格式化  
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

#### 1) 编码 JSON

示例代码:

```
package main

import (
    "encoding/json"
    "fmt"
)

type IT struct {
    Company string
    Subjects []string
    IsOk     bool
    Price    float64
}

func main() {
    t1 := IT{"itcast", []string{"Go", "C++", "Python", "Test"}, true,
666.666}

    //生成一段 JSON 格式的文本
    //如果编码成功, err 将赋予零值 nil, 变量 b 将会是一个进行 JSON 格式化之后的
[]byte 类型
    //b, err := json.Marshal(t1)
    //输出结果:
{"Company":"itcast","Subjects":["Go","C++","Python","Test"],"IsOk":tr
ue,"Price":666.666}
```



```

b, err := json.MarshalIndent(t1, "", "    ")
/*
    输出结果:
    {
        "Company": "itcast",
        "Subjects": [
            "Go",
            "C++",
            "Python",
            "Test"
        ],
        "IsOk": true,
        "Price": 666.666
    }
*/
if err != nil {
    fmt.Println("json err:", err)
}
fmt.Println(string(b))
}

```

## 2) struct tag

```

⊖{
    "Company": "itcast",
    "Subjects": ⊖[
        "Go",
        "C++",
        "Python",
        "Test"
    ],
    "IsOk": true,
    "Price": 666.666
}

```

我们看到上面的输出字段名的首字母都是大写的，如果你想用小写的首字母怎么办呢？把结构体的字段名改成首字母小写的？JSON 输出的时候必须注意，**只有导出的字段(首字母是大写)才会被输出**，如果修改字段名，那么就会发现什么都不会输出，所以必须通过 struct tag 定义来实现。

针对 JSON 的输出，我们在定义 struct tag 的时候需要注意的几点是：

- 字段的 tag 是 "-"，那么这个字段不会输出到 JSON
- tag 中带有自定义名称，那么这个自定义名称会出现在 JSON 的字段名中
- tag 中如果带有 "omitempty" 选项，那么如果该字段值为空，就不会输出到 JSON 串中



- 如果字段类型是 bool, string, int, int64 等，而 tag 中带有",string"选项，那么这个字段在输出到 JSON 的时候会把该字段对应的值转换成 JSON 字符串

示例代码：

```
type IT struct {
    //Company 不会导出到 JSON 中
    Company string `json:"- "`

    // Subjects 的值会进行二次 JSON 编码
    Subjects []string `json:"subjects"`

    //转换为字符串，再输出
    IsOk bool `json:",string"`

    // 如果 Price 为空，则不输出到 JSON 串中
    Price float64 `json:"price, omitempty"`
}

func main() {
    t1 := IT{Company: "itcast", Subjects: []string{"Go", "C++", "Python",
    "Test"}, IsOk: true}

    b, err := json.Marshal(t1)
    //json.MarshalIndent(t1, "", "    ")
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(string(b))
    //输出结果：
    {"subjects":["Go","C++","Python","Test"],"IsOk":"true","price":0}
}
```

### 10.3.1.2 通过 map 生成 JSON

```
// 创建一个保存键值对的映射
t1 := make(map[string]interface{})
t1["company"] = "itcast"
t1["subjects "] = []string{"Go", "C++", "Python", "Test"}
t1["isok"] = true
t1["price"] = 666.666

b, err := json.Marshal(t1)
//json.MarshalIndent(t1, "", "    ")
```



```
if err != nil {
    fmt.Println("json err:", err)
}
fmt.Println(string(b))
//输出结果:
{"company":"itcast","isok":true,"price":666.666,"subjects":
:["Go","C++","Python","Test"]}
```

### 10.3.2 解码 JSON

可以使用 `json.Unmarshal()` 函数将 JSON 格式的文本解码为 Go 里面预期的数据结构。

`json.Unmarshal()` 函数的原型如下：

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

该函数的第一个参数是输入，即 JSON 格式的文本（比特序列），第二个参数表示目标输出容器，用于存放解码后的值。

#### 10.3.2.1 解析到结构体

```
type IT struct {
    Company string `json:"company"`
    Subjects []string `json:"subjects"`
    IsOk bool `json:"isok"`
    Price float64 `json:"price"`
}

func main() {
    b := []byte(`{
"company": "itcast",
"subjects": [
    "Go",
    "C++",
    "Python",
    "Test"
],
"isok": true,
"price": 666.666
}`)

    var t IT
    err := json.Unmarshal(b, &t)
    if err != nil {
```



```
        fmt.Println("json err:", err)
    }
    fmt.Println(t)
    //运行结果: {itcast [Go C++ Python Test] true 666.666}

    //只想要 Subjects 字段
    type IT2 struct {
        Subjects []string `json:"subjects"`
    }

    var t2 IT2
    err = json.Unmarshal(b, &t2)
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(t2)
    //运行结果: {[Go C++ Python Test]}
}
```

### 10.3.2.2 解析到 interface

示例代码:

```
func main() {
    b := []byte(`{
        "company": "itcast",
        "subjects": [
            "Go",
            "C++",
            "Python",
            "Test"
        ],
        "isok": true,
        "price": 666.666
    }`)

    var t interface{}
    err := json.Unmarshal(b, &t)
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(t)

    //使用断言判断类型
}
```



```
m := t.(map[string]interface{})
for k, v := range m {
    switch vv := v.(type) {
    case string:
        fmt.Println(k, "is string", vv)
    case int:
        fmt.Println(k, "is int", vv)
    case float64:
        fmt.Println(k, "is float64", vv)
    case bool:
        fmt.Println(k, "is bool", vv)
    case []interface{}:
        fmt.Println(k, "is an array:")
        for i, u := range vv {
            fmt.Println(i, u)
        }
    default:
        fmt.Println(k, "is of a type I don't know how to handle")
    }
}
```

运行结果:

```
map[subjects:[Go C++ Python Test] isok:true price:666.666 company:itcast]
isok is bool true
price is float64 666.666
company is string itcast
subjects is an array:
0 Go
1 C++
2 Python
3 Test
```

## 10.4 文件操作

### 10.4.1 相关 api 介绍

#### 10.4.1.1 建立与打开文件

新建文件可以通过如下两个方法:

```
func Create(name string) (file *File, err Error)
```

根据提供的文件名创建新的文件，返回一个文件对象，默认权限是 0666 的文件，返回的文件对象是可读写的。



```
func NewFile(fd uintptr, name string) *File
```

根据文件描述符创建相应的文件，返回一个文件对象

通过如下两个方法来打开文件：

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

该方法打开一个名称为 name 的文件，但是是只读方式，内部实现其实调用了 OpenFile。

```
func OpenFile(name string, flag int, perm uint32) (file *File, err Error)
```

打开名称为 name 的文件，flag 是打开的方式，只读、读写等，perm 是权限

### 10.4.1.2 写文件

```
func (file *File) Write(b []byte) (n int, err Error)
```

写入 byte 类型的信息到文件

```
func (file *File) WriteAt(b []byte, off int64) (n int, err Error)
```

在指定位置开始写入 byte 类型的信息

```
func (file *File) WriteString(s string) (ret int, err Error)
```

写入 string 信息到文件

### 10.4.1.3 读文件

```
func (file *File) Read(b []byte) (n int, err Error)
```

读取数据到 b 中

```
func (file *File) ReadAt(b []byte, off int64) (n int, err Error)
```

从 off 开始读取数据到 b 中

### 10.4.1.4 删除文件

```
func Remove(name string) Error
```

调用该函数就可以删除文件名为 name 的文件

## 10.4.2 示例代码

### 10.4.2.1 写文件

```
package main
```

```
import (
```



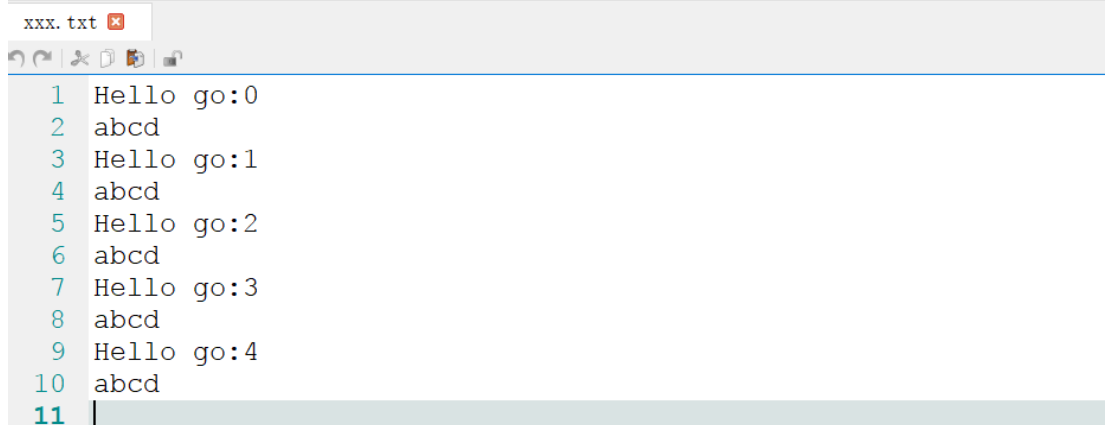
```
    "fmt"
    "os"
)

func main() {
    fout, err := os.Create("./xxx.txt") //新建文件
    //fout, err := os.OpenFile("./xxx.txt", os.O_CREATE, 0666)
    if err != nil {
        fmt.Println(err)
        return
    }

    defer fout.Close() //main 函数结束前， 关闭文件

    for i := 0; i < 5; i++ {
        outstr := fmt.Sprintf("%s:%d\n", "Hello go", i)
        fout.WriteString(outstr) //写入 string 信息到文件
        fout.Write([]byte("abcd\n")) //写入 byte 类型的信息到文件
    }
}
```

xxx.txt 内容如下:



```
xxx.txt
1 Hello go:0
2 abcd
3 Hello go:1
4 abcd
5 Hello go:2
6 abcd
7 Hello go:3
8 abcd
9 Hello go:4
10 abcd
11
```

#### 10.4.2.2 读文件

```
func main() {
    fin, err := os.Open("./xxx.txt") //打开文件
    if err != nil {
        fmt.Println(err)
    }
    defer fin.Close()
```





```
buf := make([]byte, 1024) //开辟 1024 个字节的 slice 作为缓冲
for {
    n, _ := fin.Read(buf) //读文件
    if n == 0 {           //0 表示已经到文件结束
        break
    }

    fmt.Println(string(buf)) //输出读取的内容
}
}
```

### 10.4.3 案例：拷贝文件

示例代码：

```
package main

import (
    "fmt"
    "io"
    "os"
)

func main() {
    args := os.Args //获取用户输入的所有参数

    //如果用户没有输入,或参数个数不够,则调用该函数提示用户
    if args == nil || len(args) != 3 {
        fmt.Println("usage : xxx srcFile dstFile")
        return
    }

    srcPath := args[1] //获取输入的第一个参数
    dstPath := args[2] //获取输入的第二个参数
    fmt.Printf("srcPath = %s, dstPath = %s\n", srcPath, dstPath)

    if srcPath == dstPath {
        fmt.Println("源文件和目的文件名字不能相同")
        return
    }

    srcFile, err1 := os.Open(srcPath) //打开源文件
    if err1 != nil {
        fmt.Println(err1)
        return
    }
}
```



```
}

dstFile, err2 := os.Create(dstPath) //创建目的文件
if err2 != nil {
    fmt.Println(err2)
    return
}

buf := make([]byte, 1024) //切片缓冲区
for {
    //从源文件读取内容，n 为读取文件内容的长度
    n, err := srcFile.Read(buf)
    if err != nil && err != io.EOF {
        fmt.Println(err)
        break
    }

    if n == 0 {
        fmt.Println("文件处理完毕")
        break
    }

    //切片截取
    tmp := buf[:n]
    //把读取的内容写入到目的文件
    dstFile.Write(tmp)
}

//关闭文件
srcFile.Close()
dstFile.Close()
}
```

运行结果：

File Name	Modified Date	File Type	Size
main.go	2018/1/18 10:37	GO 文件	2 KB
src.exe	2018/1/18 10:37	应用程序	1,940 KB
xxx.mp4	2017/1/8 8:52	MP4 文件	29,769 KB

File Name	Modified Date	File Type	Size
main.go	2018/1/18 10:37	GO 文件	2 KB
src.exe	2018/1/18 10:37	应用程序	1,940 KB
xxx.mp4	2017/1/8 8:52	MP4 文件	29,769 KB
yyy.mp4	2018/1/18 10:40	MP4 文件	29,769 KB

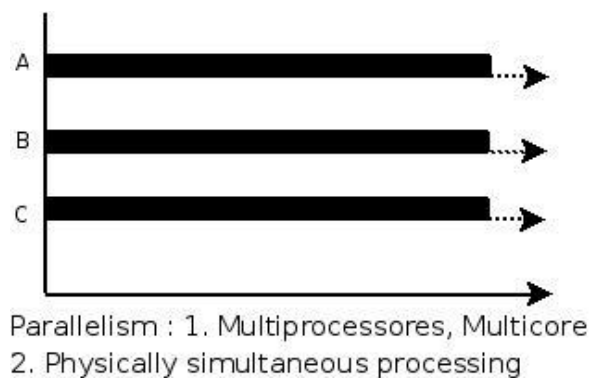
```
C:\Windows\system32\cmd.exe  
C:\Users\superman\Desktop\code\go\src>src xxx.mp4 yyy.mp4  
srcPath = xxx.mp4, dstPath = yyy.mp4  
文件处理完毕
```

## 11. 并发编程

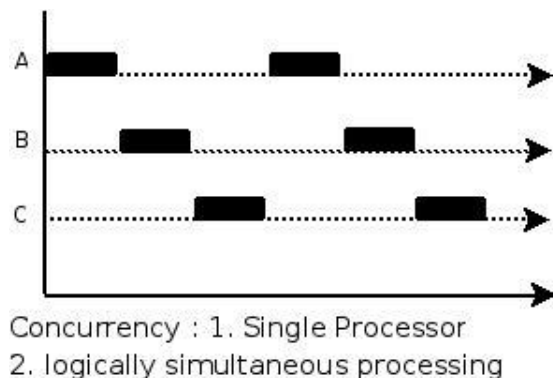
### 11.1 概述

#### 11.1.1 并行和并发

**并行(parallel):** 指在同一时刻，有多条指令在多个处理器上同时执行。

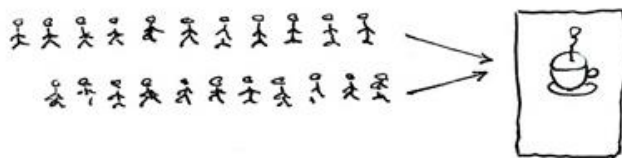


**并发(concurrency):** 指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。

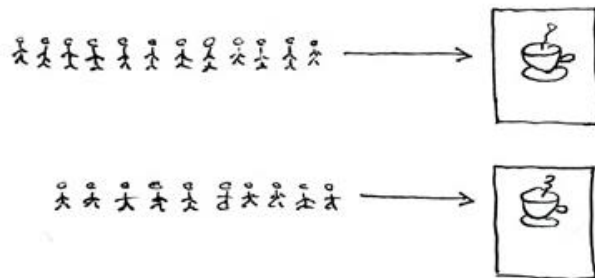


- 并行是两个队列同时使用两台咖啡机
- 并发是两个队列交替使用一台咖啡机

Concurrent = Two Queues One Coffee Machine



Parallel = Two Queues Two Coffee Machines



### 11.1.2 Go 语言并发优势

有人把 Go 比作 21 世纪的 C 语言，第一是因为 Go 语言设计简单，第二，21 世纪最重要的就是并行程序设计，而 Go 从语言层面就支持了并行。同时，并发程序的内存管理有时候是非常复杂的，而 Go 语言提供了自动垃圾回收机制。

Go 语言为并发编程而内置的上层 API 基于 CSP (communicating sequential processes, 顺序通信进程) 模型。这就意味着显式锁都是可以避免的，因为 Go 语言通过相册安全的通道发送和接受数据以实现同步，这大大地简化了并发程序的编写。

一般情况下，一个普通的桌面计算机跑十几二十个线程就有点负载过大了，但是同样这台机器却可以轻松地让成百上千甚至过万个 goroutine 进行资源竞争。

## 11.2 goroutine

### 11.2.1 goroutine 是什么

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

### 11.2.2 创建 goroutine

只需在函数调用语句前添加 go 关键字，就可创建并发执行单元。开发人员无需了解任何执行细节，调度器会自动将其安排到合适的系统线程上执行。

在并发编程里，我们通常想讲一个过程切分成几块，然后让每个 goroutine 各自负责一块工作。当一个程序启动时，其主函数即在一个单独的 goroutine 中运行，我们叫它 main goroutine。新的 goroutine 会用 go 语句来创建。

示例代码：

```
package main

import (
    "fmt"
    "time"
)

func newTask() {
    i := 0
    for {
        i++
        fmt.Printf("new goroutine: i = %d\n", i)
        time.Sleep(1 * time.Second) //延时 1s
    }
}

func main() {
    //创建一个 goroutine, 启动另外一个任务
    go newTask()

    i := 0
    //main goroutine 循环打印
}
```



```
    for {
        i++
        fmt.Printf("main goroutine: i = %d\n", i)
        time.Sleep(1 * time.Second) //延时 1s
    }
}
```

程序运行结果:

```
main goroutine: i = 1
new goroutine: i = 1
main goroutine: i = 2
new goroutine: i = 2
new goroutine: i = 3
main goroutine: i = 3
main goroutine: i = 4
new goroutine: i = 4
... ..
```

### 11.2.3 主 goroutine 先退出

主 goroutine 退出后，其它的工作 goroutine 也会自动退出:

```
func newTask() {
    i := 0
    for {
        i++
        fmt.Printf("new goroutine: i = %d\n", i)
        time.Sleep(1 * time.Second) //延时 1s
    }
}

func main() {
    //创建一个 goroutine, 启动另外一个任务
    go newTask()

    fmt.Println("main goroutine exit")
}
```

程序运行结果:

```
main goroutine exit
成功: 进程退出代码 0.
```



## 11.2.4 runtime 包

### 11.2.4.1 Gosched

`runtime.Gosched()` 用于让出 CPU 时间片，让出当前 `goroutine` 的执行权限，调度器安排其他等待的任务运行，并在下次某个时候从该位置恢复执行。

这就像跑接力赛，A 跑了一会碰到代码 `runtime.Gosched()` 就把接力棒交给 B 了，A 歇着了，B 继续跑。

示例代码：

```
func main() {  
    //创建一个goroutine  
    go func(s string) {  
        for i := 0; i < 2; i++ {  
            fmt.Println(s)  
        }  
    }("world")  
  
    for i := 0; i < 2; i++ {  
        runtime.Gosched() //import "runtime"  
        /*  
            屏蔽 runtime.Gosched() 运行结果如下：  
                hello  
                hello  
  
            没有 runtime.Gosched() 运行结果如下：  
                world  
                world  
                hello  
                hello  
  
        */  
        fmt.Println("hello")  
    }  
}
```

### 11.2.4.2 Goexit

调用 `runtime.Goexit()` 将立即终止当前 `goroutine` 执行，调度器确保所有已注册 `defer` 延迟调用被执行。

示例代码：

```
func main() {
```







在第二次执行(runtime.GOMAXPROCS(2))时，我们使用了两个 CPU，所以两个 goroutine 可以一起被执行，以同样的频率交替打印 0 和 1。

## 11.3 channel

goroutine 运行在相同的地址空间，因此访问共享内存必须做好同步。goroutine 奉行通过通信来共享内存，而不是共享内存来通信。

引用类型 channel 是 CSP 模式的具体实现，用于多个 goroutine 通讯。其内部实现了同步，确保并发安全。

### 11.3.1 channel 类型

和 map 类似，channel 也是一个对应 make 创建的底层数据结构的引用。

当我们复制一个 channel 或用于函数参数传递时，我们只是拷贝了一个 channel 引用，因此调用者何被调用者将引用同一个 channel 对象。和其它的引用类型一样，channel 的零值也是 nil。

定义一个 channel 时，也需要定义发送到 channel 的值的类型。channel 可以使用内置的 make() 函数来创建：

```
make(chan Type) //等价于 make(chan Type, 0)
make(chan Type, capacity)
```

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

channel 通过操作符<-来接收和发送数据，发送和接收数据语法：

```
channel <- value    //发送 value 到 channel
<-channel           //接收并将其丢弃
x := <-channel      //从 channel 中接收数据，并赋值给 x
x, ok := <-channel  //功能同上，同时检查通道是否已关闭或者是否为空
```

默认情况下，channel 接收和发送数据都是阻塞的，除非另一端已经准备好，这样就使得 goroutine 同步变的更加的简单，而不需要显式的 lock。

示例代码：

```
func main() {
    c := make(chan int)

    go func() {
```



```
defer fmt.Println("子协程结束")

fmt.Println("子协程正在运行.....")

c <- 666 //666 发送到 c
}()

num := <-c //从 c 中接收数据，并赋值给 num

fmt.Println("num = ", num)
fmt.Println("main 协程结束")
}
```

程序运行结果：

```
子协程正在运行.....
子协程结束
num = 666
main协程结束
成功：进程退出代码 0.
```

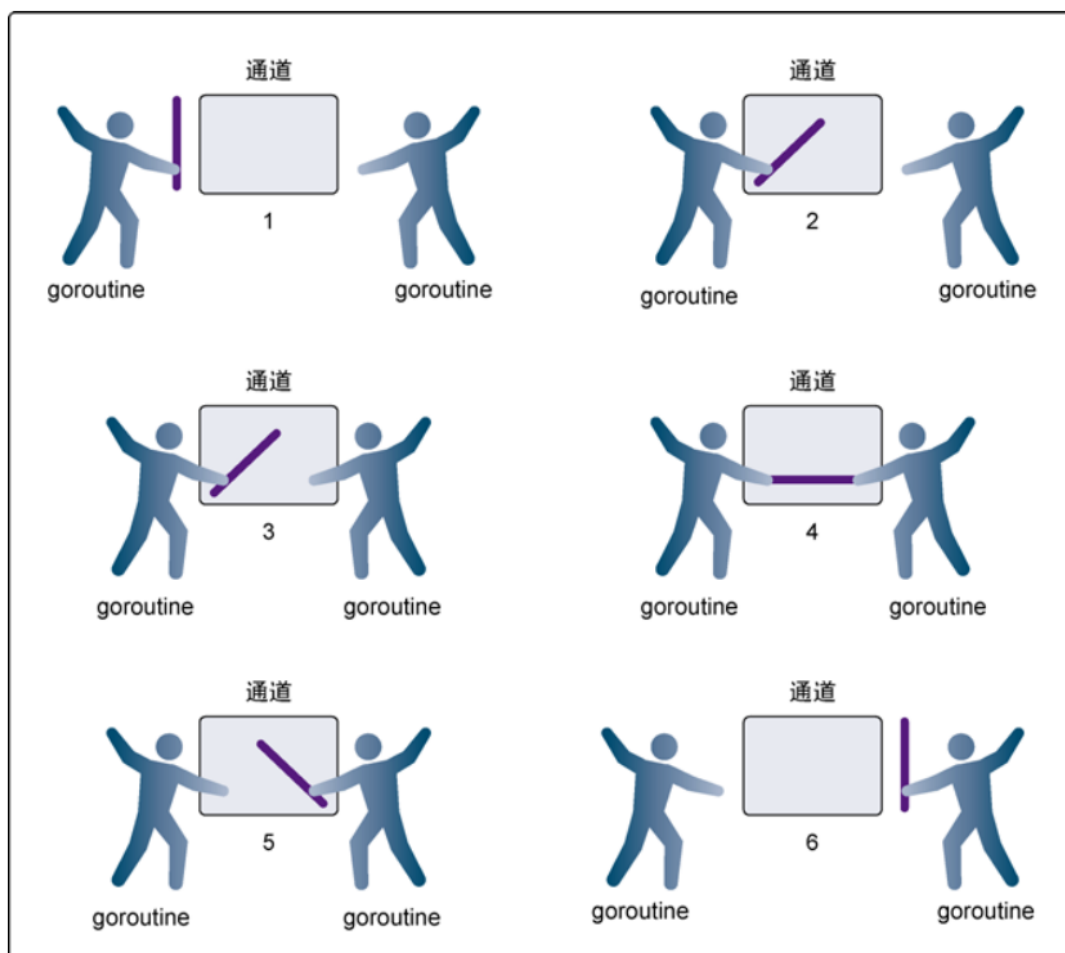
### 11.3.2 无缓冲的 channel

无缓冲的通道（unbuffered channel）是指在接收前没有能力保存任何值的通道。

这种类型的通道要求发送 goroutine 和接收 goroutine 同时准备好，才能完成发送和接收操作。如果两个 goroutine 没有同时准备好，通道会导致先执行发送或接收操作的 goroutine 阻塞等待。

这种对通道进行发送和接收的交互行为本身就是同步的。其中任意一个操作都无法离开另一个操作单独存在。

下图展示两个 goroutine 如何利用无缓冲的通道来共享一个值：



使用无缓冲的通道在 goroutine 之间同步

- 在第 1 步，两个 goroutine 都到达通道，但哪个都没有开始执行发送或者接收。
- 在第 2 步，左侧的 goroutine 将它的手伸进了通道，这模拟了向通道发送数据的行为。这时，这个 goroutine 会在通道中被锁住，直到交换完成。
- 在第 3 步，右侧的 goroutine 将它的手放入通道，这模拟了从通道里接收数据。这个 goroutine 一样也会在通道中被锁住，直到交换完成。
- 在第 4 步和第 5 步，进行交换，并最终，在第 6 步，两个 goroutine 都将它们的手从通道里拿出来，这模拟了被锁住的 goroutine 得到释放。两个 goroutine 现在都可以去做别的事情了。

无缓冲的 channel 创建格式：

```
make(chan Type) //等价于 make(chan Type, 0)
```

如果没有指定缓冲区容量，那么该通道就是同步的，因此会阻塞到发送者准备好发送和接收者准备好接收。

示例代码：

```
func main() {  
    c := make(chan int, 0) //无缓冲的通道
```



```
//内置函数 len 返回未被读取的缓冲元素数量， cap 返回缓冲区大小
fmt.Printf("len(c)=%d, cap(c)=%d\n", len(c), cap(c))

go func() {
    defer fmt.Println("子协程结束")

    for i := 0; i < 3; i++ {
        c <- i
        fmt.Printf("子协程正在运行[%d]: len(c)=%d, cap(c)=%d\n", i,
len(c), cap(c))
    }
}()

time.Sleep(2 * time.Second) //延时 2s

for i := 0; i < 3; i++ {
    num := <-c //从 c 中接收数据，并赋值给 num
    fmt.Println("num = ", num)
}

fmt.Println("main 协程结束")
}
```

程序运行结果：

```
len(c)=0, cap(c)=0
子协程正在运行[0]: len(c)=0, cap(c)=0
num = 0
num = 1
子协程正在运行[1]: len(c)=0, cap(c)=0
子协程正在运行[2]: len(c)=0, cap(c)=0
子协程结束
num = 2
main协程结束
```

### 11.3.3 有缓冲的 channel

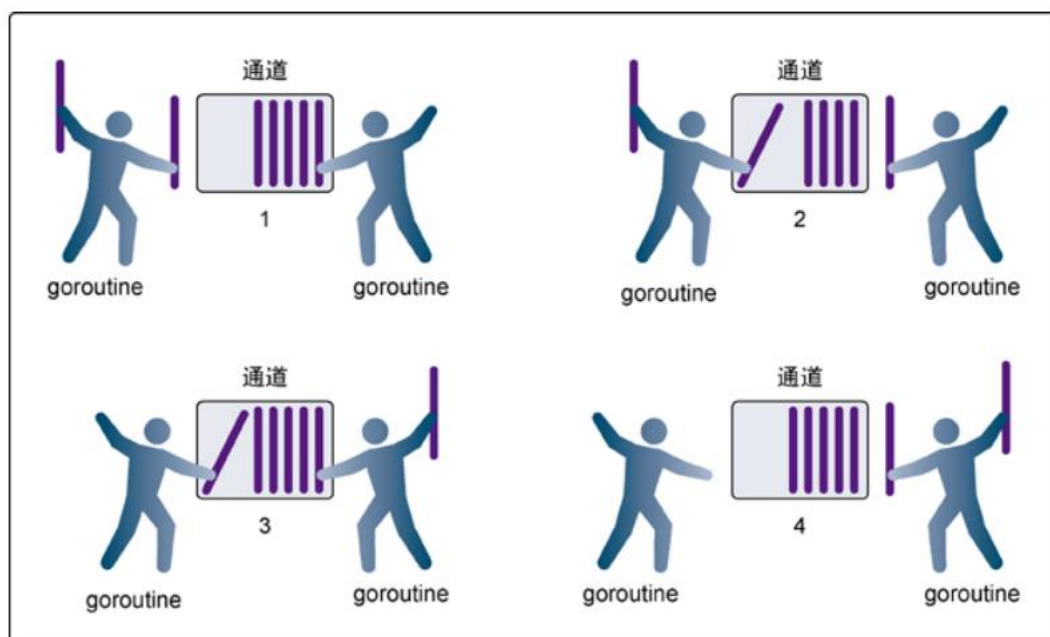
有缓冲的通道（buffered channel）是一种在被接收前能存储一个或者多个值的通道。

这种类型的通道并不强制要求 goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也会不同。只有在通道中没有要接收的值时，接收动作才会阻塞。只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

这导致有缓冲的通道和无缓冲的通道之间的一个很大的不同：无缓冲的通道保证进行发送和接收的 goroutine 会在同一时间进行数据交换；有缓冲的通道没有这种保证。



示例图如下：



使用有缓冲的通道在 goroutine 之间同步数据

- 在第 1 步，右侧的 goroutine 正在从通道接收一个值。
- 在第 2 步，右侧的这个 goroutine 独立完成了接收值的动作，而左侧的 goroutine 正在发送一个新值到通道里。
- 在第 3 步，左侧的 goroutine 还在向通道发送新值，而右侧的 goroutine 正在从通道接收另外一个值。这个步骤里的两个操作既不是同步的，也不会互相阻塞。
- 最后，在第 4 步，所有的发送和接收都完成，而通道里还有几个值，也有一些空间可以存更多的值。

有缓冲的 channel 创建格式：

```
make(chan Type, capacity)
```

如果给定了一个缓冲区容量，通道就是异步的。只要缓冲区有未使用空间用于发送数据，或还包含可以接收的数据，那么其通信就会无阻塞地进行。

示例代码：

```
func main() {
    c := make(chan int, 3) //带缓冲的通道

    //内置函数 len 返回未被读取的缓冲元素数量， cap 返回缓冲区大小
    fmt.Printf("len(c)=%d, cap(c)=%d\n", len(c), cap(c))

    go func() {
        defer fmt.Println("子协程结束")
    }
}
```



```

        for i := 0; i < 3; i++ {
            c <- i
            fmt.Printf("子协程正在运行[%d]: len(c)=%d, cap(c)=%d\n", i,
len(c), cap(c))
        }
    }()

    time.Sleep(2 * time.Second) //延时 2s
    for i := 0; i < 3; i++ {
        num := <-c //从 c 中接收数据，并赋值给 num
        fmt.Println("num = ", num)
    }

    fmt.Println("main 协程结束")
}

```

程序运行结果:

```

len(c)=0, cap(c)=3
子协程正在运行[0]: len(c)=0, cap(c)=3
子协程正在运行[1]: len(c)=1, cap(c)=3
子协程正在运行[2]: len(c)=2, cap(c)=3
子协程结束
num = 0
num = 1
num = 2
main协程结束

```

### 11.3.4 range 和 close

如果发送者知道，没有更多的值需要发送到 channel 的话，那么让接收者也能及时知道没有多余的值可接收将是有用的，因为接收者可以停止不必要的接收等待。这可以通过内置的 close 函数来关闭 channel 实现。

示例代码:

```

func main() {
    c := make(chan int)

    go func() {
        for i := 0; i < 5; i++ {
            c <- i
        }

        //把 close(c) 注释掉，程序会一直阻塞在 if data, ok := <-c; ok 那一行
        close(c)
    }()

    for {

```



```
//ok 为 true 说明 channel 没有关闭，为 false 说明管道已经关闭
if data, ok := <-c; ok {
    fmt.Println(data)
} else {
    break
}

fmt.Println("Finished")
}
```

程序运行结果：

```
0
1
2
3
4
Finished
```

注意点：

- channel 不像文件一样需要经常去关闭，只有当你确实没有任何发送数据了，或者你想显式的结束 range 循环之类的，才去关闭 channel；
- 关闭 channel 后，无法向 channel 再发送数据(引发 panic 错误后导致接收立即返回零值)；
- 关闭 channel 后，可以继续向 channel 接收数据；
- 对于 nil channel，无论收发都会被阻塞。

可以使用 range 来迭代不断操作 channel：

```
func main() {
    c := make(chan int)

    go func() {
        for i := 0; i < 5; i++ {
            c <- i
        }
        //把 close(c) 注释掉，程序会一直阻塞在 for data := range c 那一行
        close(c)
    }()

    for data := range c {
        fmt.Println(data)
    }
    fmt.Println("Finished")
}
```



### 11.3.5 单方向的 channel

默认情况下，通道是双向的，也就是，既可以往里面发送数据也可以同里面接收数据。

但是，我们经常见一个通道作为参数进行传递而值希望对方是单向使用的，要么只让它发送数据，要么只让它接收数据，这时候我们可以指定通道的方向。

单向 channel 变量的声明非常简单，如下：

```
var ch1 chan int      // ch1 是一个正常的 channel，不是单向的
var ch2 chan<- float64 // ch2 是单向 channel，只用于写 float64 数据
var ch3 <-chan int     // ch3 是单向 channel，只用于读取 int 数据
```

- chan<- 表示数据进入管道，要把数据写进管道，对于调用者就是输出。
- <-chan 表示数据从管道出来，对于调用者就是得到管道的数据，当然就是输入。

可以将 channel 隐式转换为单向队列，只收或只发，**不能将**单向 channel 转换为普通 channel：

```
c := make(chan int, 3)
var send chan<- int = c // send-only
var recv <-chan int = c // receive-only
send <- 1
//<-send //invalid operation: <-send (receive from send-only type
chan<- int)
<-recv
//recv <- 2 //invalid operation: recv <- 2 (send to receive-only type
<-chan int)

//不能将单向 channel 转换为普通 channel
d1 := (chan int)(send) //cannot convert send (type chan<- int) to type
chan int
d2 := (chan int)(recv) //cannot convert recv (type <-chan int) to type
chan int
```

示例代码：

```
// chan<- //只写
func counter(out chan<- int) {
    defer close(out)
    for i := 0; i < 5; i++ {
        out <- i //如果对方不读 会阻塞
    }
}

// <-chan //只读
func printer(in <-chan int) {
```





```
    for num := range in {  
        fmt.Println(num)  
    }  
}  
  
func main() {  
    c := make(chan int) // chan //读写  
  
    go counter(c) //生产者  
    printer(c)    //消费者  
  
    fmt.Println("done")  
}
```

## 11.3.6 定时器

### 11.3.6.1 Timer

Timer 是一个定时器，代表未来的一个单一事件，你可以告诉 timer 你要等待多长时间，它提供一个 channel，在将来的那个时间那个 channel 提供了一个时间值。

示例代码：

```
import "fmt"  
import "time"  
  
func main() {  
    //创建定时器，2 秒后，定时器就会向自己的 C 字节发送一个 time.Time 类型的元素值  
    timer1 := time.NewTimer(time.Second * 2)  
    t1 := time.Now() //当前时间  
    fmt.Printf("t1: %v\n", t1)  
  
    t2 := <-timer1.C  
    fmt.Printf("t2: %v\n", t2)  
  
    //如果只是想单纯的等待的话，可以使用 time.Sleep 来实现  
    timer2 := time.NewTimer(time.Second * 2)  
    <-timer2.C  
    fmt.Println("2s 后")  
  
    time.Sleep(time.Second * 2)  
    fmt.Println("再一次 2s 后")  
  
    <-time.After(time.Second * 2)  
    fmt.Println("再再一次 2s 后")  
}
```



```
timer3 := time.NewTimer(time.Second)
go func() {
    <-timer3.C
    fmt.Println("Timer 3 expired")
}()

stop := timer3.Stop() //停止定时器
if stop {
    fmt.Println("Timer 3 stopped")
}

fmt.Println("before")
timer4 := time.NewTimer(time.Second * 5) //原来设置 3s
timer4.Reset(time.Second * 1)           //重新设置时间
<-timer4.C
fmt.Println("after")
}
```

### 11.3.6.2 Ticker

Ticker 是一个定时触发的计时器，它会以一个间隔(interval)往 channel 发送一个事件(当前时间)，而 channel 的接收者可以以固定的时间间隔从 channel 中读取事件。

示例代码：

```
func main() {
    //创建定时器，每隔 1 秒后，定时器就会给 channel 发送一个事件(当前时间)
    ticker := time.NewTicker(time.Second * 1)

    i := 0
    go func() {
        for { //循环
            <-ticker.C
            i++
            fmt.Println("i = ", i)

            if i == 5 {
                ticker.Stop() //停止定时器
            }
        }
    }() //别忘了()

    //死循环，特地不让 main goroutine 结束
}
```



```
    for {  
    }  
}
```

## 11.4 select

### 11.4.1 select 作用

Go 里面提供了一个关键字 `select`，通过 `select` 可以监听 `channel` 上的数据流动。

`select` 的用法与 `switch` 语言非常类似，由 `select` 开始一个新的选择块，每个选择条件由 `case` 语句来描述。

与 `switch` 语句可以选择任何可使用相等比较的条件相比，`select` 有比较多的限制，其中最大的一条限制就是每个 `case` 语句里必须是一个 IO 操作，大致的结构如下：

```
select {  
    case <-chan1:  
        // 如果 chan1 成功读到数据，则进行该 case 处理语句  
    case chan2 <- 1:  
        // 如果成功向 chan2 写入数据，则进行该 case 处理语句  
    default:  
        // 如果上面都没有成功，则进入 default 处理流程  
}
```

在一个 `select` 语句中，Go 语言会按顺序从头至尾评估每一个发送和接收的语句。

如果其中的任意一语句可以继续执行(即没有被阻塞)，那么就从那些可以执行的语句中任意选择一条来使用。

如果没有任意一条语句可以执行(即所有的通道都被阻塞)，那么有两种可能的情况：

- 如果给出了 `default` 语句，那么就会执行 `default` 语句，同时程序的执行会从 `select` 语句后的语句中恢复。
- 如果没有 `default` 语句，那么 `select` 语句将被阻塞，直到至少有一个通信可以进行下去。

示例代码：

```
func fibonacci(c, quit chan int) {  
    x, y := 1, 1  
    for {  
        select {  
            case c <- x:  
                x, y = y, x+y  
            case <-quit:  
                fmt.Println("quit")  
        }  
    }  
}
```



```
        return
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)

    go func() {
        for i := 0; i < 6; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()

    fibonacci(c, quit)
}
```

运行结果如下：

```
1
1
2
3
5
8
quit
```

### 11.4.2 超时

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

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



```
        }  
    }  
}()  
//c <- 666 // 注释掉，引发 timeout  
<-o  
}
```

## 12. 网络编程

### 12.1 网络概述

#### 12.1.1 网络协议

从应用的角度出发，协议可理解为“规则”，是数据传输和数据的解释的规则。

假设，A、B 双方欲传输文件。规定：

- 第一次，传输文件名，接收方接收到文件名，应答 OK 给传输方；
- 第二次，发送文件的尺寸，接收方接收到该数据再次应答一个 OK；
- 第三次，传输文件内容。同样，接收方接收数据完成后应答 OK 表示文件内容接收成功。

由此，无论 A、B 之间传递何种文件，都是通过三次数据传输来完成。A、B 之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B 之间达成的这个相互遵守的规则即为协议。

这种仅在 A、B 之间被遵守的协议称之为**原始协议**。

当此协议被更多的人采用，不断的增加、改进、维护、完善。最终形成一个稳定的、完整的文件传输协议，被广泛应用于各种文件传输过程中。该协议就成为一个**标准协议**。最早的 ftp 协议就是由此衍生而来。

#### 12.1.2 分层模型

##### 12.1.2.1 网络分层架构

为了减少协议设计的复杂性，大多数网络模型均采用分层的方式来组织。每一层都有自己的功能，就像建筑物一样，每一层都靠下一层支持。**每一层利用下一层提供的服务来为上一层提供服务，本层服务的实现细节对上层屏蔽。**



OSI/RM(理论上的标准)	TCP/IP(事实上的标准)
应用层	应用层
表示层	
会话层	
传输层	传输层
网络层	网络层
数据链路层	链路层
物理层	

越下面的层，越靠近硬件；越上面的层，越靠近用户。至于每一层叫什么名字，其实并不重要（面试的时候，面试官可能会问每一层的名字）。只需要知道，互联网分成若干层即可。

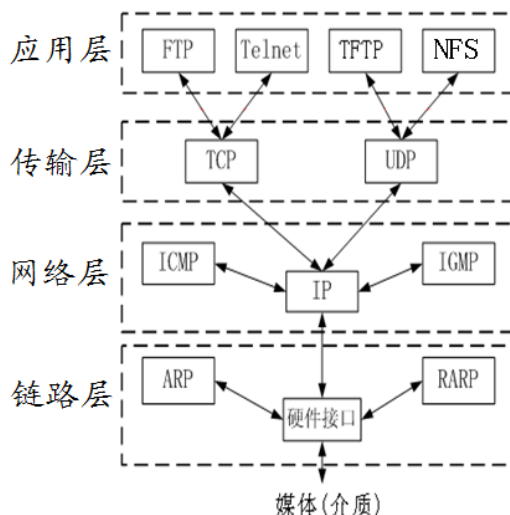
- 1) **物理层**：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由 1、0 转化为电流强弱来进行传输，到达目的地后再转化为 1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。
- 2) **数据链路层**：定义了如何让格式化数据以帧为单位进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。如：串口通信中使用到的 115200、8、N、1
- 3) **网络层**：在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet 的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。
- 4) **传输层**：定义了一些传输数据的协议和端口号（WWW 端口 80 等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与 TCP 特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如 QQ 聊天数据就是通过这种方式传输的）。主要是将从下层接收的数据进行分段和传输，到达目的地后再进行重组。常常把这一层数据叫做段。
- 5) **会话层**：通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是 IP 也可以是 MAC 或者是主机名）。
- 6) **表示层**：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC 程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码 (EBCDIC)，而另一台则使用美国信息交换标准码 (ASCII) 来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。
- 7) **应用层**：是最靠近用户的 OSI 层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。

### 12.1.2.2 层与协议

每一层都是为了完成一种功能，为了实现这些功能，就需要大家都遵守共同的规则。大家都

遵守这规则，就叫做“协议”（protocol）。

网络的每一层，都定义了很多协议。这些协议的总称，叫“TCP/IP 协议”。TCP/IP 协议是一个大家族，不仅仅只有 TCP 和 IP 协议，它还包括其它的协议，如下图：



### 12.1.2.3 每层协议的功能



#### 1) 链路层

以太网规定，连入网络的所有设备，都必须具有“网卡”接口。数据包必须是从一块网卡，传送到另一块网卡。通过网卡能够使不同的计算机之间连接，从而完成数据通信等功能。网卡的地址——MAC 地址，就是数据包的物理发送地址和物理接收地址。

#### 2) 网络层

网络层的作用是引进一套新的地址，使得我们能够区分不同的计算机是否属于同一个子网络。这套地址就叫做“网络地址”，这是我们平时所说的 IP 地址。这个 IP 地址好比我们的手机号码，通过手机号码可以得到用户所在的归属地。



网络地址帮助我们确定计算机所在的子网络，MAC 地址则将数据包送到该子网络中的目标网卡。网络层协议包含的主要信息是源 IP 和目的 IP。

于是，“网络层”出现以后，每台计算机有了两种地址，一种是 MAC 地址，另一种是网络地址。两种地址之间没有任何联系，MAC 地址是绑定在网卡上的，网络地址则是管理员分配的，它们只是随机组合在一起。

网络地址帮助我们确定计算机所在的子网络，MAC 地址则将数据包送到该子网络中的目标网卡。因此，从逻辑上可以推断，必定是先处理网络地址，然后再处理 MAC 地址。

### 3) 传输层

当我们一边聊 QQ，一边聊微信，当一个数据包从互联网上发来的时候，我们怎么知道，它是来自 QQ 的内容，还是来自微信的内容？

也就是说，我们还需要一个参数，表示这个数据包到底供哪个程序（进程）使用。这个参数就叫做“端口”（port），它其实是每一个使用网卡的程序的编号。每个数据包都发到主机的特定端口，所以不同的程序就能取到自己所需要的数据。

端口特点：

- 对于同一个端口，在不同系统对应着不同的进程
- 对于同一个系统，一个端口只能被一个进程拥有

### 4) 应用层

应用程序收到“传输层”的数据，接下来就要进行解读。由于互联网是开放架构，数据来源五花八门，必须事先规定好格式，否则根本无法解读。“应用层”的作用，就是规定应用程序的数据格式。

## 12.2 Socket 编程

### 12.2.1 什么是 Socket

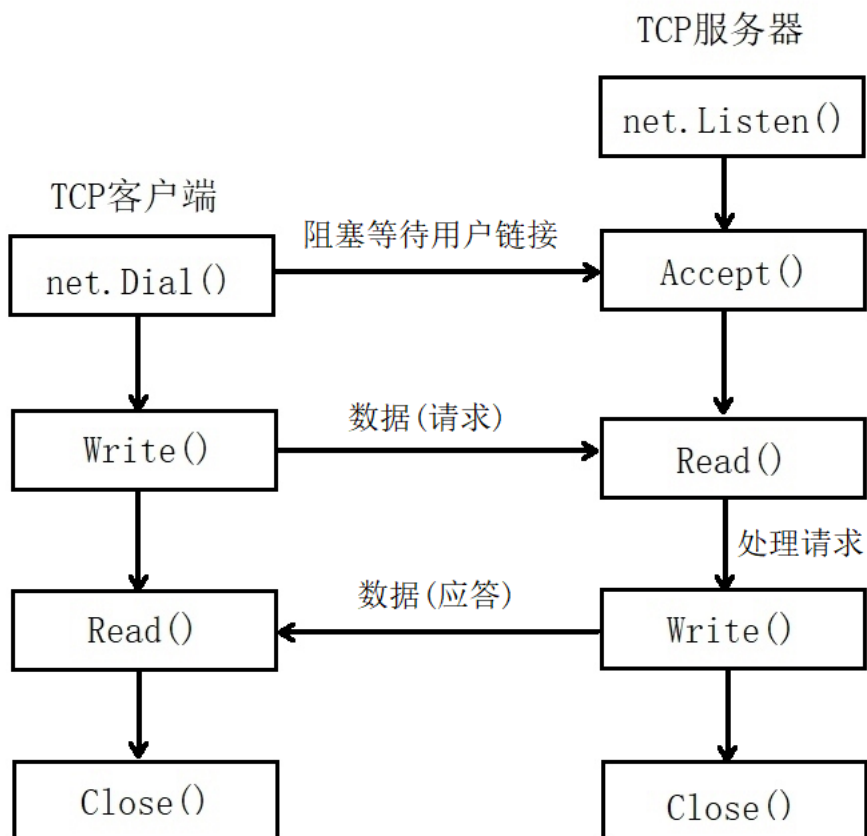
Socket 起源于 Unix，而 Unix 基本哲学之一就是“一切皆文件”，都可以用“打开 open -> 读写 write/read -> 关闭 close”模式来操作。Socket 就是该模式的一个实现，网络的 Socket 数据传输是一种特殊的 I/O，Socket 也是一种文件描述符。Socket 也具有一个类似于打开文件的函数调用：Socket()，该函数返回一个整型的 Socket 描述符，随后的连接建立、数据传输等操作都是通过该 Socket 实现的。

常用的 Socket 类型有两种：流式 Socket（SOCK\_STREAM）和数据报式 Socket



(SOCK\_DGRAM)。流式是一种面向连接的 Socket，针对于面向连接的 TCP 服务应用；数据报式 Socket 是一种无连接的 Socket，对应于无连接的 UDP 服务应用。

### 12.2.2 TCP 的 C/S 架构



### 12.2.3 示例程序

#### 12.2.3.1 服务器代码

```
package main

import (
    "fmt"
    "log"
    "net"
    "strings"
)

func dealConn(conn net.Conn) {

    defer conn.Close() //此函数结束时，关闭连接套接字
```



```
//conn.RemoteAddr().String(): 连接客服端的网络地址
ipAddr := conn.RemoteAddr().String()
fmt.Println(ipAddr, "连接成功")

buf := make([]byte, 1024) //缓冲区，用于接收客户端发送的数据

for {
    //阻塞等待用户发送的数据
    n, err := conn.Read(buf) //n 代码接收数据的长度
    if err != nil {
        fmt.Println(err)
        return
    }
    //切片截取，只截取有效数据
    result := buf[:n]
    fmt.Printf("接收到数据来自[%s]==>[%d]:%s\n", ipAddr, n,
string(result))
    if "exit" == string(result) { //如果对方发送"exit"，退出此链接
        fmt.Println(ipAddr, "退出连接")
        return
    }

    //把接收到的数据转换为大写，再给客户端发送
    conn.Write([]byte(strings.ToUpper(string(result))))
}

func main() {
    //创建、监听 socket
    listener, err := net.Listen("tcp", "127.0.0.1:8000")
    if err != nil {
        log.Fatal(err) //log.Fatal()会产生panic
    }

    defer listener.Close()

    for {
        conn, err := listener.Accept() //阻塞等待客户端连接
        if err != nil {
            log.Println(err)
            continue
        }
    }
}
```



```
        go dealConn(conn)
    }
}
```

### 12.2.3.2 客服端代码

```
package main

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

func main() {
    //客户端主动连接服务器
    conn, err := net.Dial("tcp", "127.0.0.1:8000")
    if err != nil {
        log.Fatal(err) //log.Fatal()会产生panic
        return
    }

    defer conn.Close() //关闭

    buf := make([]byte, 1024) //缓冲区
    for {
        fmt.Printf("请输入发送的内容: ")
        fmt.Scan(&buf)
        fmt.Printf("发送的内容: %s\n", string(buf))

        //发送数据
        conn.Write(buf)

        //阻塞等待服务器回复的数据
        n, err := conn.Read(buf) //n 代码接收数据的长度
        if err != nil {
            fmt.Println(err)
            return
        }

        //切片截取，只截取有效数据
        result := buf[:n]
```



```
        fmt.Printf("接收到数据[%d]:%s\n", n, string(result))
    }
}
```

### 12.2.3.3 运行结果

```
C:\Windows\system32\cmd.exe - go run server.go
服务器
C:\Users\superman\Desktop\xxxx>go run server.go
127.0.0.1:2440 连接成功
127.0.0.1:2451 连接成功
接收到数据来自[127.0.0.1:2451]==>[2]:go
接收到数据来自[127.0.0.1:2440]==>[6]:python
接收到数据来自[127.0.0.1:2440]==>[4]:exit
127.0.0.1:2440 退出连接
接收到数据来自[127.0.0.1:2451]==>[3]:c++

C:\Windows\system32\cmd.exe
C:\Users\superman\Desktop\xxxx>go run client.go
客户端1
请输入发送的内容: python
发送的内容: python
接收到数据[6]:PYTHON
请输入发送的内容: exit
发送的内容: exit
EOF

C:\Windows\system32\cmd.exe - go run client.go
选择C:\Windows\system32\cmd.exe - go run client.go
C:\Users\superman\Desktop\xxxx>go run client.go
客户端2
请输入发送的内容: go
发送的内容: go
接收到数据[2]:GO
请输入发送的内容: c++
发送的内容: c++
接收到数据[3]:C++
请输入发送的内容:
```

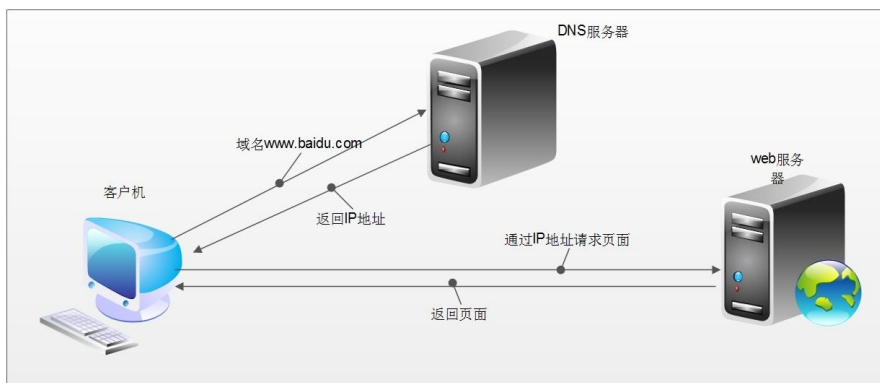
## 12.3 HTTP 编程

### 12.3.1 概述

#### 12.3.1.1 Web 工作方式

我们平时浏览网页的时候,会打开浏览器,输入网址后按下回车键,然后就会显示出你想要浏览的内容。在这个看似简单的用户行为背后,到底隐藏了些什么呢?

对于普通的上网过程,系统其实是这样做的:浏览器本身是一个客户端,当你输入 URL 的时候,首先浏览器会去请求 DNS 服务器,通过 DNS 获取相应的域名对应的 IP,然后通过 IP 地址找到 IP 对应的服务器后,要求建立 TCP 连接,等浏览器发送完 HTTP Request (请求) 包后,服务器接收到请求包之后才开始处理请求包,服务器调用自身服务,返回 HTTP Response (响应) 包;客户端收到来自服务器的响应后开始渲染这个 Response 包里的主体 (body),等收到全部的内容随后断开与该服务器之间的 TCP 连接。



一个 Web 服务器也被称为 HTTP 服务器，它通过 HTTP 协议与客户端通信。这个客户端通常指的是 Web 浏览器(其实手机端客户端内部也是浏览器实现的)。

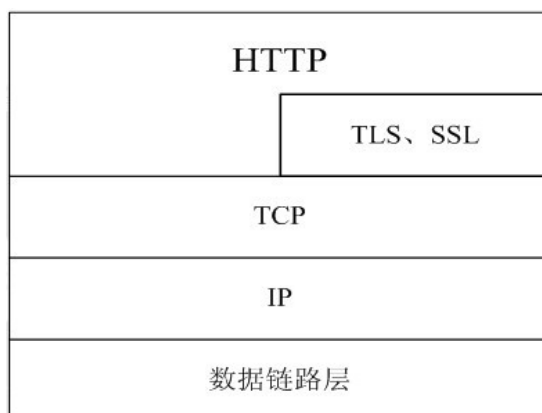
Web 服务器的工作原理可以简单地归纳为：

- 客户机通过 TCP/IP 协议建立到服务器的 TCP 连接
- 客户端向服务器发送 HTTP 协议请求包，请求服务器里的资源文档
- 服务器向客户机发送 HTTP 协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户机
- 客户机与服务器断开。由客户端解释 HTML 文档，在客户端屏幕上渲染图形结果

### 12.3.1.2 HTTP 协议

超文本传输协议(HTTP, HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议，它详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。

HTTP 协议通常承载于 TCP 协议之上，有时也承载于 TLS 或 SSL 协议层之上，这个时候，就成了我们常说的 HTTPS。如下图所示：





### 12.3.1.3 地址（URL）

URL 全称为 Unique Resource Location，用来表示网络资源，可以理解为网络文件路径。

URL 的格式如下：

```
http://host[":"port][abs_path]
http://192.168.31.1/html/index
```

URL 的长度有限制，不同的服务器的限制值不太相同，但是不能无限长。

## 12.3.2 HTTP 报文浅析

### 12.3.2.1 请求报文格式

#### 1) 测试代码

服务器测试代码：

```
package main

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

func main() {
    //创建、监听 socket
    listener, err := net.Listen("tcp", "127.0.0.1:8000")
    if err != nil {
        log.Fatal(err) //log.Fatal() 会产生 panic
    }

    defer listener.Close()

    conn, err := listener.Accept() //阻塞等待客户端连接
    if err != nil {
        log.Println(err)
        return
    }

    defer conn.Close() //此函数结束时，关闭连接套接字

    //conn.RemoteAddr().String(): 连接客服端的网络地址
    ipAddr := conn.RemoteAddr().String()
```



```
fmt.Println(ipAddr, "连接成功")

buf := make([]byte, 4096) //缓冲区，用于接收客户端发送的数据

//阻塞等待用户发送的数据
n, err := conn.Read(buf) //n 代码接收数据的长度
if err != nil {
    fmt.Println(err)
    return
}

//切片截取，只截取有效数据
result := buf[:n]

fmt.Printf("接收到数据来自[%s]==>:\n%s\n", ipAddr, string(result))
}
```

浏览器输入 url 地址:



服务器端运行打印结果如下:

```
127.0.0.1:2563 连接成功
接收到数据来自[127.0.0.1:2563]==>:
GET /go HTTP/1.1 请求行
Host: 127.0.0.1:8000 请求头部
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.87 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: zh-CN,zh;q=0.8

空行
成功: 进程退出代码 0.
```

## 2) 请求报文格式说明

HTTP 请求报文由请求行、请求头部、空行、请求包体 4 个部分组成，如下图所示:





### 1) 请求行

请求行由方法字段、URL 字段 和 HTTP 协议版本字段 3 个部分组成，他们之间使用空格隔开。常用的 HTTP 请求方法有 GET、POST。

#### GET:

- 当客户端要从服务器中读取某个资源时，使用 GET 方法。GET 方法要求服务器将 URL 定位的资源放在响应报文的数据部分，回送给客户端，即向服务器请求某个资源。
- 使用 GET 方法时，请求参数和对应的值附加在 URL 后面，利用一个问号(“?”)代表 URL 的结尾与请求参数的开始，传递参数长度受限制，因此 GET 方法不适合用于上传数据。
- 通过 GET 方法来获取网页时，参数会显示在浏览器地址栏上，因此保密性很差。

#### POST:

- 当客户端给服务器提供信息较多时可以使用 POST 方法，POST 方法向服务器提交数据，比如完成表单数据的提交，将数据提交给服务器处理。
- GET 一般用于获取/查询资源信息，POST 会附带用户数据，一般用于更新资源信息。POST 方法将请求参数封装在 HTTP 请求数据中，而且长度没有限制，因为 POST 携带的数据，在 HTTP 的请求正文中，以名称/值的形式出现，可以传输大量数据。

### 2) 请求头部

请求头部为请求报文添加了一些附加信息，由“名/值”对组成，每行一对，名和值之间使用冒号分隔。

请求头部通知服务器有关于客户端请求的信息，典型的请求头有：

请求头	含义
User-Agent	请求的浏览器类型
Accept	客户端可识别的响应内容类型列表，星号“*”用于按范围将类型分组，用“*/*”指示可接受全部类型，用“type/*”指示可接受 type 类型的所有子类型
Accept-Language	客户端可接受的自然语言
Accept-Encoding	客户端可接受的编码压缩格式
Accept-Charset	可接受的应答的字符集
Host	请求的主机名，允许多个域名同处一个 IP 地址，即虚拟主机
connection	连接方式(close 或 keepalive)
Cookie	存储于客户端扩展字段，向同一域名的服务端发送属于该域的 cookie

### 3) 空行

最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

### 4) 请求包体

请求包体不在 GET 方法中使用，而是 POST 方法中使用。

POST 方法适用于需要客户填写表单的场合。与请求包体相关的最常使用的是包体类型 Content-Type 和包体长度 Content-Length。





### 12.3.2.2 响应报文格式

#### 1) 测试代码

服务器示例代码:

```
package main

import (
    "fmt"
    "net/http"
)

//服务端编写的业务逻辑处理程序
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello world")
}

func main() {
    http.HandleFunc("/go", myHandler)

    //在指定的地址进行监听，开启一个 HTTP
    http.ListenAndServe("127.0.0.1:8000", nil)
}
```

启动服务器程序:

```
C:\Windows\system32\cmd.exe - go run server.go

C:\Users\superman\Desktop>go run server.go
```

客户端测试示例代码:

```
package main

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

func main() {
    //客户端主动连接服务器
    conn, err := net.Dial("tcp", "127.0.0.1:8000")
    if err != nil {
```



```

        log.Fatal(err) //log.Fatal()会产生panic
        return
    }

    defer conn.Close() //关闭

    requestHeader := "GET /go HTTP/1.1\r\nAccept: image/gif, image/jpeg,
image/pjpeg, application/x-ms-application, application/xhtml+xml,
application/x-ms-xbap, */*\r\nAccept-Language:
zh-Hans-CN,zh-Hans;q=0.8,en-US;q=0.5,en;q=0.3\r\nUser-Agent:
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; WOW64;
Trident/7.0; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727; .NET CLR
3.0.30729; .NET CLR 3.5.30729)\r\nAccept-Encoding: gzip,
deflate\r\nHost: 127.0.0.1:8000\r\nConnection: Keep-Alive\r\n\r\n"

    //先发送请求包
    conn.Write([]byte(requestHeader))

    buf := make([]byte, 4096) //缓冲区

    //阻塞等待服务器回复的数据
    n, err := conn.Read(buf) //n 代码接收数据的长度
    if err != nil {
        fmt.Println(err)
        return
    }

    //切片截取，只截取有效数据
    result := buf[:n]
    fmt.Printf("接收到数据[%d]:\n%s\n", n, string(result))
}

```

启动程序，测试 http 的**成功**响应报文：

接收到数据[129]:

HTTP/1.1 200 OK **状态行**

Date: Thu, 18 Jan 2018 08:17:02 GMT

Content-Length: 12

Content-Type: text/plain; charset=utf-8

**响应头部**

**空行**

hello world **响应包体**

启动程序，测试 http 的**失败**响应报文：

```

接收到数据[176]:
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Thu, 18 Jan 2018 08:17:59 GMT
Content-Length: 19
    
```

404 page not found

## 2) 响应报文格式说明

HTTP 响应报文由**状态行**、**响应头部**、**空行**、**响应包体** 4 个部分组成，如下图所示：



### 1) 状态行

状态行由 HTTP 协议版本字段、状态码和状态码的描述文本 3 个部分组成，他们之间使用空格隔开。

状态码：

状态码由三位数字组成，第一位数字表示响应的类型，常用的状态码有五大类如下所示：

状态码	含义
1xx	表示服务器已接收了客户端请求，客户端可继续发送请求
2xx	表示服务器已成功接收到请求并进行处理
3xx	表示服务器要求客户端重定向
4xx	表示客户端的请求有非法内容
5xx	表示服务器未能正常处理客户端的请求而出现意外错误

常见的状态码举例：

状态码	含义
200 OK	客户端请求成功
400 Bad Request	请求报文有语法错误



状态码	含义
401 Unauthorized	未授权
403 Forbidden	服务器拒绝服务
404 Not Found	请求的资源不存在
500 Internal Server Error	服务器内部错误
503 Server Unavailable	服务器临时不能处理客户端请求(稍后可能可以)

## 2) 响应头部

响应头可能包括：

响应头	含义
Location	Location 响应报头域用于重定向接受者到一个新的位置
Server	Server 响应报头域包含了服务器用来处理请求的软件信息及其版本
Vary	指示不可缓存的请求头列表
Connection	连接方式

## 3) 空行

最后一个响应头部之后是一个空行，发送回车符和换行符，通知服务器以下不再有响应头部。

## 4) 响应包体

服务器返回给客户端的文本信息。

## 12.3.3 HTTP 编程

Go 语言标准库内建提供了 net/http 包，涵盖了 HTTP 客户端和服务端的具体实现。使用 net/http 包，我们可以很方便地编写 HTTP 客户端或服务端的程序。

### 12.3.3.1 HTTP 服务端

示例代码：

```
package main

import (
    "fmt"
    "net/http"
)

//服务端编写的业务逻辑处理程序
//handler 函数： 具有 func(w http.ResponseWriter, r *http.Request)签名的函数
func myHandler(w http.ResponseWriter, r *http.Request) {
```



```

    fmt.Println(r.RemoteAddr, "连接成功") //r.RemoteAddr 远程网络地址
    fmt.Println("method = ", r.Method) //请求方法
    fmt.Println("url = ", r.URL.Path)
    fmt.Println("header = ", r.Header)
    fmt.Println("body = ", r.Body)

    w.Write([]byte("hello go")) //给客户端回复数据
}

func main() {
    http.HandleFunc("/go", myHandler)

    //该方法用于在指定的 TCP 网络地址 addr 进行监听，然后调用服务端处理程序来处理
    传入的连接请求。
    //该方法有两个参数：第一个参数 addr 即监听地址；第二个参数表示服务端处理程序，
    通常为 nil
    //第二个参数为空意味着服务端调用 http.DefaultServeMux 进行处理
    http.ListenAndServe("127.0.0.1:8000", nil)
}

```

浏览器输入 url 地址:



服务器运行结果:

```

C:\Users\superman\Desktop>go run server.go
127.0.0.1:4663 连接成功
method = GET
url = /go
header = map[Cache-Control:[max-age=0] Upgrade-Insecure-Requests:[1] Accept:[text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8] Accept-Encoding:[gzip, deflate, sdch, br] Connection:[keep-alive] User-Agent:[Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.87 Safari/537.36] Accept-Language:[zh-CN,zh;q=0.8]]
body = {}

```

### 12.3.3.2 HTTP 客户端

```

package main

import (
    "fmt"
    "io"
    "log"

```



```
    "net/http"
)

func main() {

    //get 方式请求一个资源
    //resp, err := http.Get("http://www.baidu.com")
    //resp, err :=
http.Get("http://www.neihan8.com/article/index.html")
    resp, err := http.Get("http://127.0.0.1:8000/go")
    if err != nil {
        log.Println(err)
        return
    }

    defer resp.Body.Close() //关闭

    fmt.Println("header = ", resp.Header)
    fmt.Printf("resp status %s\nstatusCode %d\n", resp.Status,
resp.StatusCode)
    fmt.Printf("body type = %T\n", resp.Body)

    buf := make([]byte, 2048) //切片缓冲区
    var tmp string

    for {
        n, err := resp.Body.Read(buf) //读取 body 包内容
        if err != nil && err != io.EOF {
            fmt.Println(err)
            return
        }

        if n == 0 {
            fmt.Println("读取内容结束")
            break
        }
        tmp += string(buf[:n]) //累加读取的内容
    }

    fmt.Println("buf = ", string(tmp))
}
```

