目录

Go 语言教程

1 Go 语言简介

1.1 Go 语言的历史

Go 是一个开源的编程语言，它能让构造简单、可靠且高效的软件变得容易。

Go 是从 2007 年末由 Robert Griesemer, Rob Pike, Ken Thompson 主持开发，后来还加入了 Ian Lance Taylor, Russ Cox 等人，并最终于 2009 年 11 月开源，在 2012 年早些时候发布了 Go 1 稳定版本。现在 Go 的开发已经是完全开放的，并且拥有一个活跃的社区。

1.2 Go 语言的用途

Go 语言被设计成一门应用于搭载 Web 服务器，存储集群或类似用途的巨型中央服务器的系统编程语言。

对于高性能分布式系统领域而言，Go 语言无疑比大多数其它语言有着更高的开发效率。它提供了海量并行的支持，这对于游戏服务端的开发而言是再好不过了。

1.3 GO 语言的特点

计算机软件经历了数十年的发展，形成了多种学术流派，有面向过程编程、面向对象编程、函数式编程、面向消息编程等，这些思想究竟孰优孰劣，众说纷纭。

除了 OOP 外，近年出现了一些小众的编程哲学，Go 语言对这些思想亦有所吸收。例如，Go 语言接受了函数式编程的一些想法，支持匿名函数与闭包。再如，Go 语言接受了以 Erlang 语言为代表的面向消息编程思想，支持 goroutine 和通道，并推荐使用消息而不是共享内存来进行并发编程。总体来说，Go 语言是一个非常现代化的语言，精小但非常强大。

为了保持语言的简洁和简单，按照类似的语言省略常用的功能。

Go 语言的特性有：

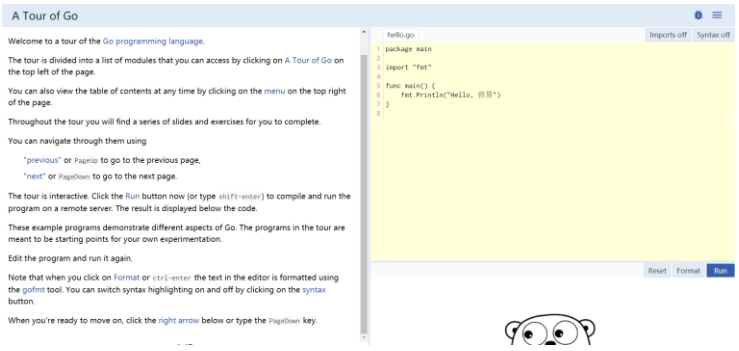
- 不支持类型继承
- 不支持任何方法或运算符重载
- 不支付包之间循环依赖
- 不支持对指针运算
- 不支持断言
- 不支持泛型编程
- 自动垃圾回收
- 更丰富的内置类型
- 函数多返回值
- 错误处理
- 匿名函数和闭包
- 类型和接口
- 并发编程

- 语言交互性
- 简洁、快速、安全
- 并行、有趣、开源
- 编译迅速

2 Go 语言环境搭建

注：由于某种原因，golang 官网被国内屏蔽。你可以直接找客户索取。或是使用代理来访问外部网站。

在学习 Go 语言编程之前，我们需要安装和配置好 Go 语言的开发环境。可以选择线上的编译器：http://tour.golang.org/welcome/1 来直接执行代码。



也可以在您自己的计算机上安装开发编译环境。

Go 语言支持以下系统：

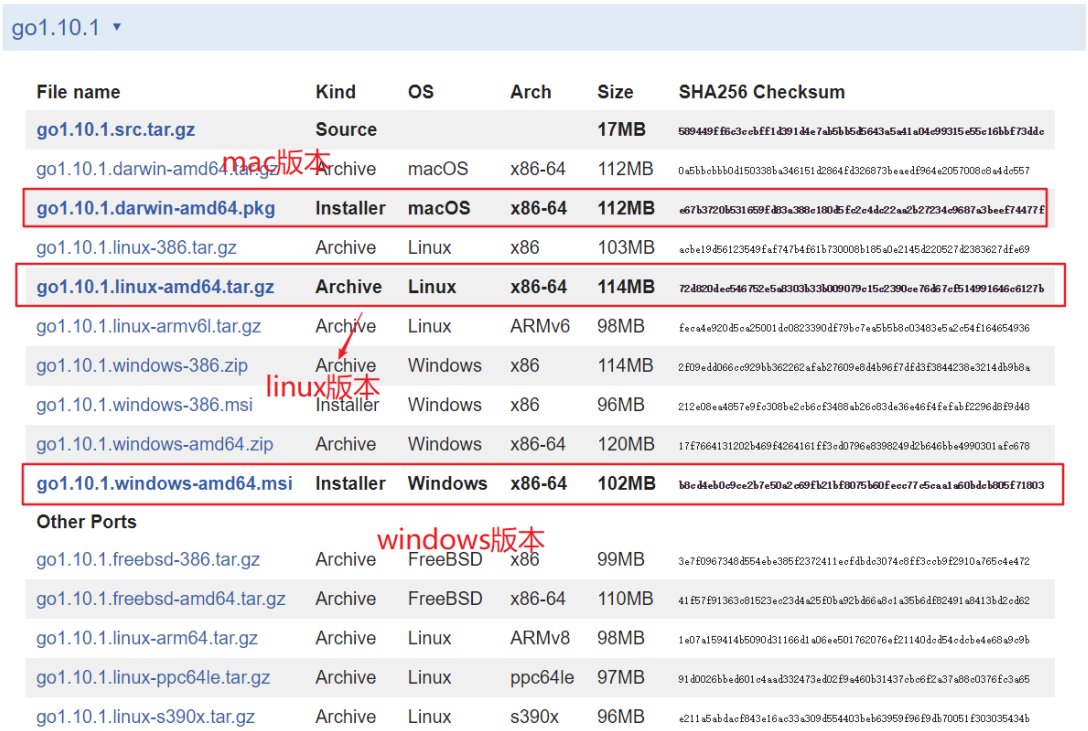
Linux

FreeBSD

Mac OS X（也称为 Darwin）

Window

安装包下载地址为：https://golang.org/dl/。 各个系统对应的包名：



File name	Kind	OS	Arch	Size	SHA256 Checksum
go1.10.1.src.tar.gz	Source			17MB	589449ff6c3ccbff1d391d4e7ab5b5d5643a5a41a04c99315e55c16bbf73ddc
go1.10.1.darwin-amd64.tar.gz	Archive	macOS	x86-64	112MB	0a5bbcbbb0d150338ba346151d2864fd326873beaedf964e2057008c8a4dc557
go1.10.1.darwin-amd64.pkg	Installer	macOS	x86-64	112MB	e67b3720b531659fd83a388c180d5fc2c4dc22aa2b27234e9687a3beef74477f
go1.10.1.linux-386.tar.gz	Archive	Linux	x86	103MB	acbe19d56123549faf747b4f61b730008b185a0e2145d220527d2383627dfe69
go1.10.1.linux-amd64.tar.gz	Archive	Linux	x86-64	114MB	72d820dee546752e5a8303b33b009079e15c2390ce76d67cf514991646c6127b
go1.10.1.linux-armv6l.tar.gz	Archive	Linux	ARMv6	98MB	feca4e920d5ca25001dc0823390df79bc7ea5b5b8c03483e5a2c54f164654936
go1.10.1.windows-386.zip	Archive	Windows	x86	114MB	2f09edd066cc929bb362262afab27609e8d4b96f7dfd3f3844238e3214db9b8a
go1.10.1.windows-386.msi	Installer	Windows	x86	96MB	212e08ea4857e9fc308be2cb6cf3488ab26c83de36e46f4fefabf2296d8f9d48
go1.10.1.windows-amd64.zip	Archive	Windows	x86-64	120MB	17f7664131202b469f4264161ff3cd0796e8398249d2b646bbe4990301afc678
go1.10.1.windows-amd64.msi	Installer	Windows	x86-64	102MB	b8cd4eb0c9ce2b7e50a2c69f1215f8075b60fecc77e5caa1a60bdcb805f71803
Other Ports					
go1.10.1.freebsd-386.tar.gz	Archive	FreeBSD	x86	99MB	3e7f0967348d554ebe385f2372411ecfdbdc3074c8ff3ccb9f2910a765c4e472
go1.10.1.freebsd-amd64.tar.gz	Archive	FreeBSD	x86-64	110MB	41f57f91363c81523ec23d4a25f0ba92bd66a8c1a35b6df82491a8413bd2cd62
go1.10.1.linux-arm64.tar.gz	Archive	Linux	ARMv8	98MB	1e07a159414b5090d31166d1a06ee501762076ef21140dcd54cdcbe4e68a9c9b
go1.10.1.linux-ppc64le.tar.gz	Archive	Linux	ppc64le	97MB	91d0026bbed601c4aad332473ed02f9a460b31437cbc6f2a37a88c0376fc3a65
go1.10.1.linux-s390x.tar.gz	Archive	Linux	s390x	96MB	e211a5abdacf843e16ac33a309d554403beb63959f96f9db70051f303035434b

2. 1 UNIX/Linux/Mac OS X, 和FreeBSD 安装

以下介绍了在 UNIX/Linux/Mac OS X, 和 FreeBSD 系统下使用源码安装方法：

1、下载源码包：go1.10.1.linux-amd64.tar.gz。

2、将下载的源码包解压至 /usr/local 目录。

```
tar -C /usr/local -xzf go1.10.1.linux-amd64.tar.gz
```

3、将/usr/local/go/bin 目录添加至 PATH 环境变量：

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

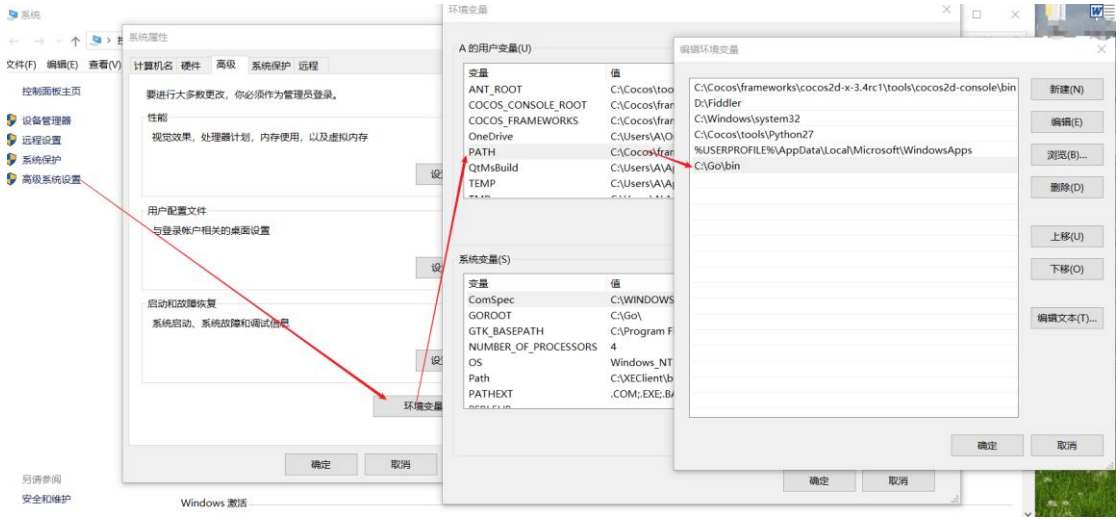
注意：MAC 系统下你可以使用.pkg 结尾的安装包直接双击来完成安装，安装目录在 /usr/local/go/ 下。

2.2 Windows 系统下安装

Windows 下可以使用 .msi 后缀(在下载列表中可以找到该文件，如 go1.4.2.windows-amd64.msi)的安装包来安装。

默认情况下.msi 文件会安装在 c:\Go 目录下。

你可以将 c:\Go\bin 目录添加到 PATH 环境变量中。添加后你需要重启命令窗口才能生效。



设置 path 环境变量

安装测试

创建工作目录 C:\>Go_WorkSpace。 文件名: test.go，代码如下：

```
package main

import "fmt"

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

使用go 命令执行以上代码输出结果如下：

```
C:\Go_WorkSpace>go run test.go

Hello, World!
```

注意，要想对中文的支持必须把文件的格式变为 UTF-8 编码

3 Go 语言开发工具

3.1 LiteIDE

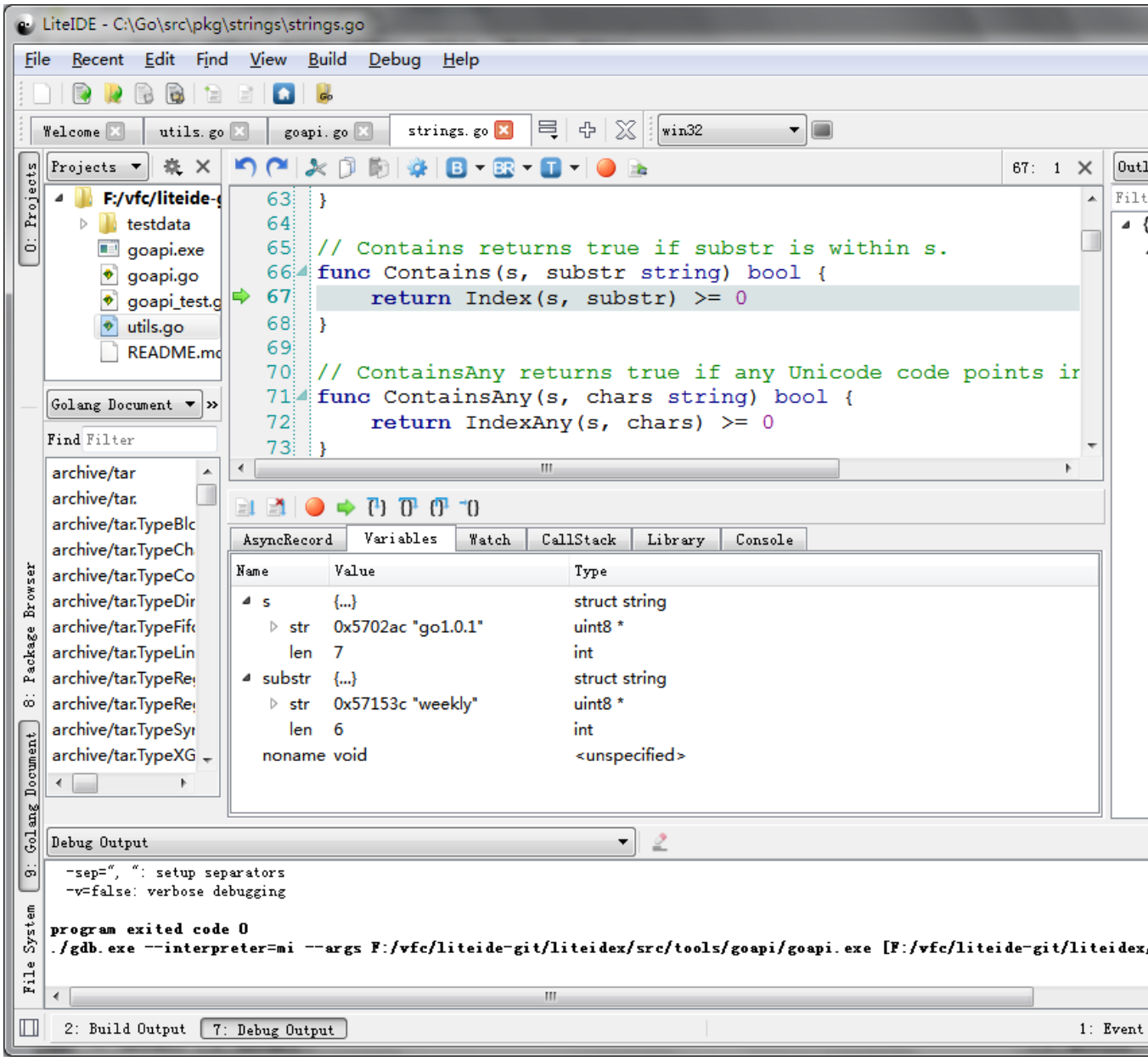
LiteIDE 是一款开源、跨平台的轻量级 Go 语言集成开发环境（IDE）。

3.1.1 支持的操作系统

- Windows x86 (32-bit or 64-bit)
- Linux x86 (32-bit or 64-bit)

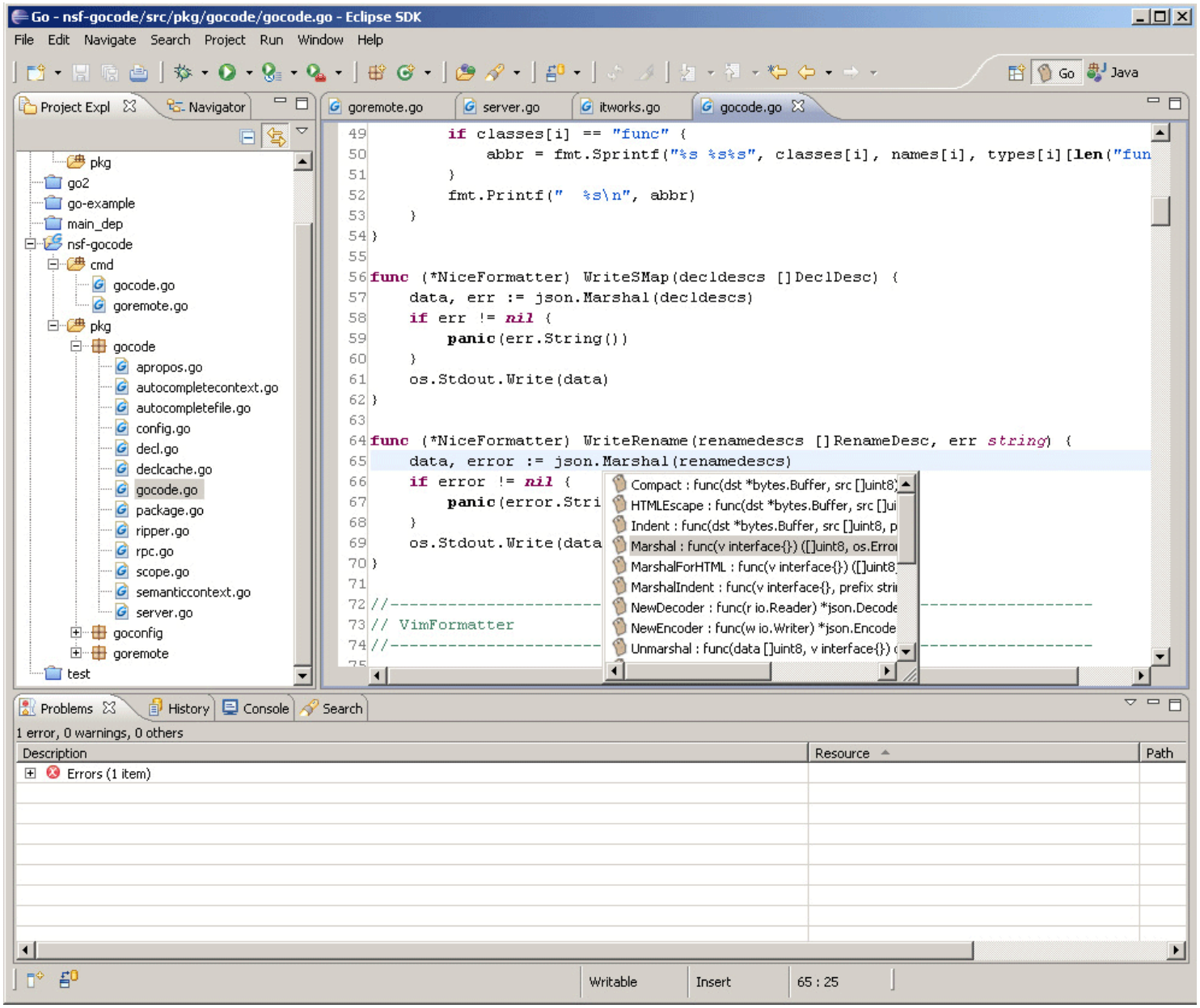
下载地址 ： http://sourceforge.net/projects/liteide/files/

源码地址 ： https://github.com/visualfc/liteide



3.2 Eclipse

Eclipse 也是非常常用的开发利器，以下介绍如何使用 Eclipse 来编写 Go 程序。



3.2.1 Eclipse 编辑 Go 的主界面

1. 首先下载并安装好 Eclipse
2. 下载 goclipse 插件 http://code.google.com/p/goclipse/wiki/InstallationInstructions
3. 下载 gocode，用于 go 的代码补全提示

   gocode 的 github 地址：

```
https://github.com/nsf/gocode
```

   在 windows 下要安装 git，通常用 msysgit

   再在 cmd 下安装：

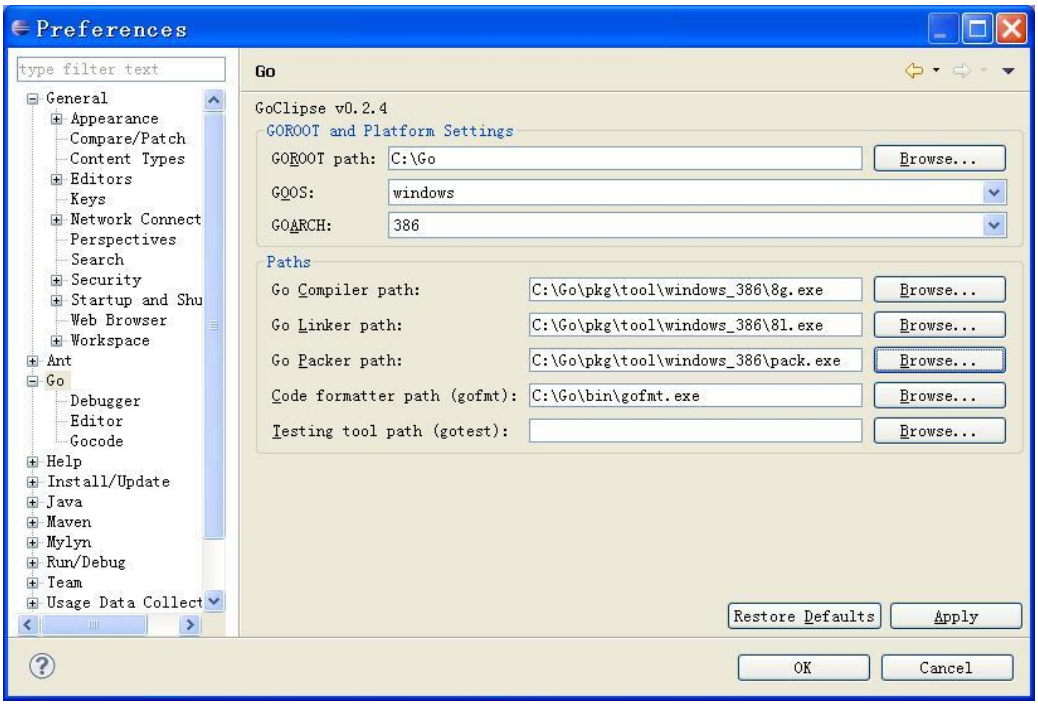
```
go get -u github.com/nsf/gocode
```

也可以下载代码，直接用 go build 来编译，会生成 gocode.exe

4. 下载 MinGW 并按要求装好
5. 配置插件

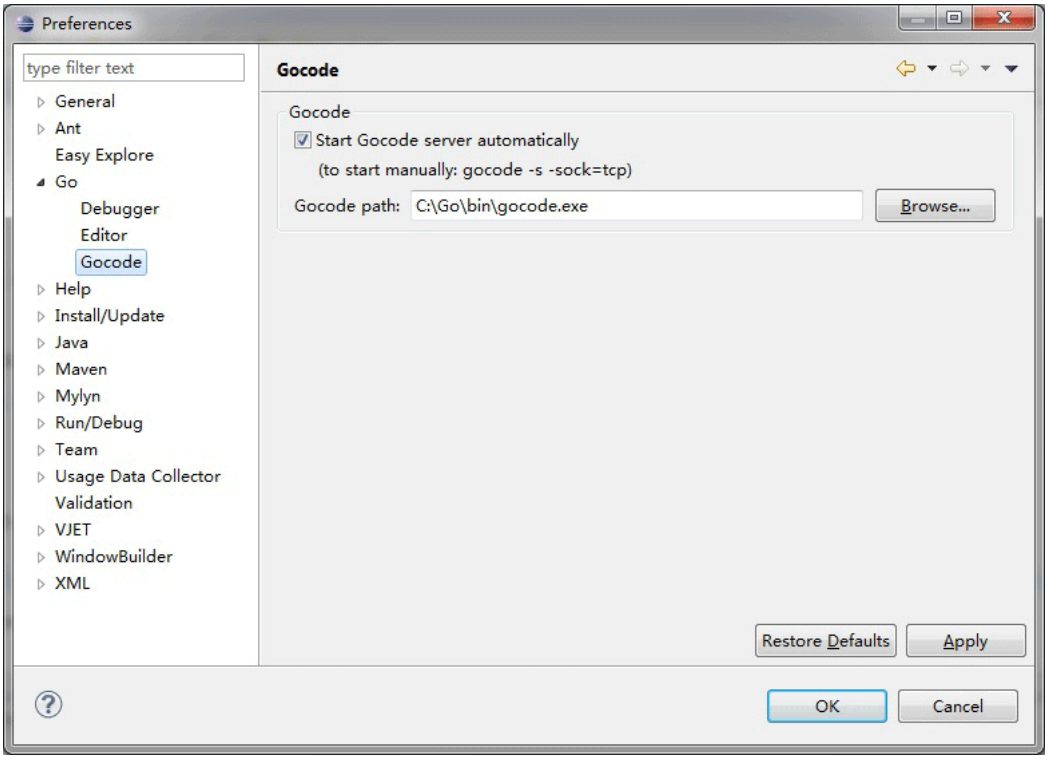
Windows->Reference->Go

(1).配置 Go 的编译器



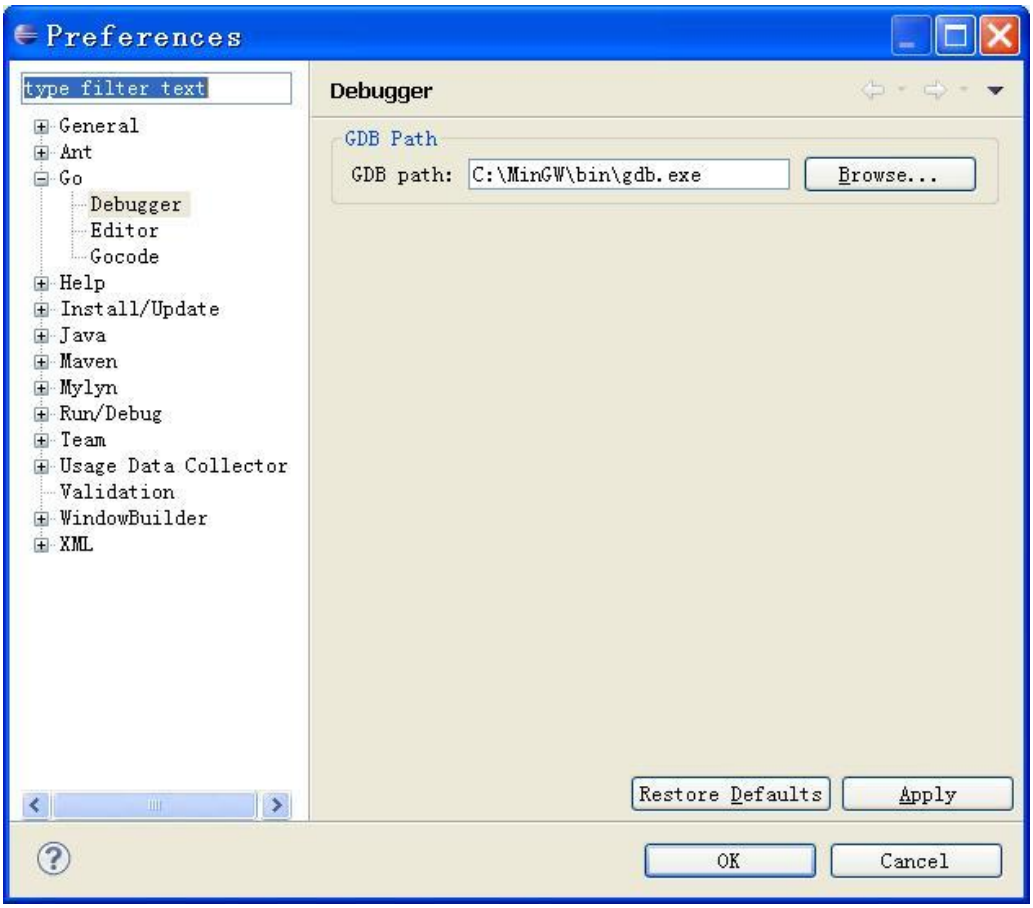
设置 Go 的一些基础信息

(2).配置 Gocode（可选，代码补全），设置 Gocode 路径为之前生成的 gocode.exe 文件



设置 gocode 信息

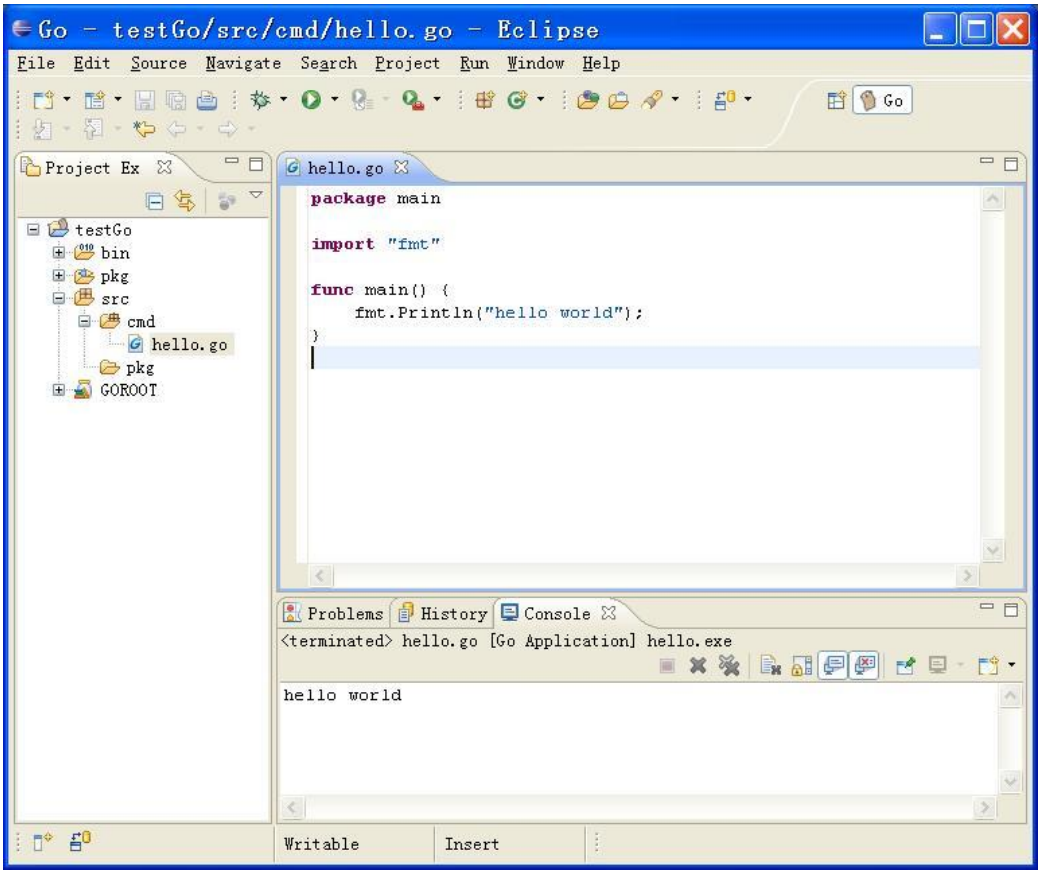
(3).配置 GDB（可选，做调试用），设置 GDB 路径为 MingW 安装目录下的 gdb.exe 文件



设置 GDB 信息

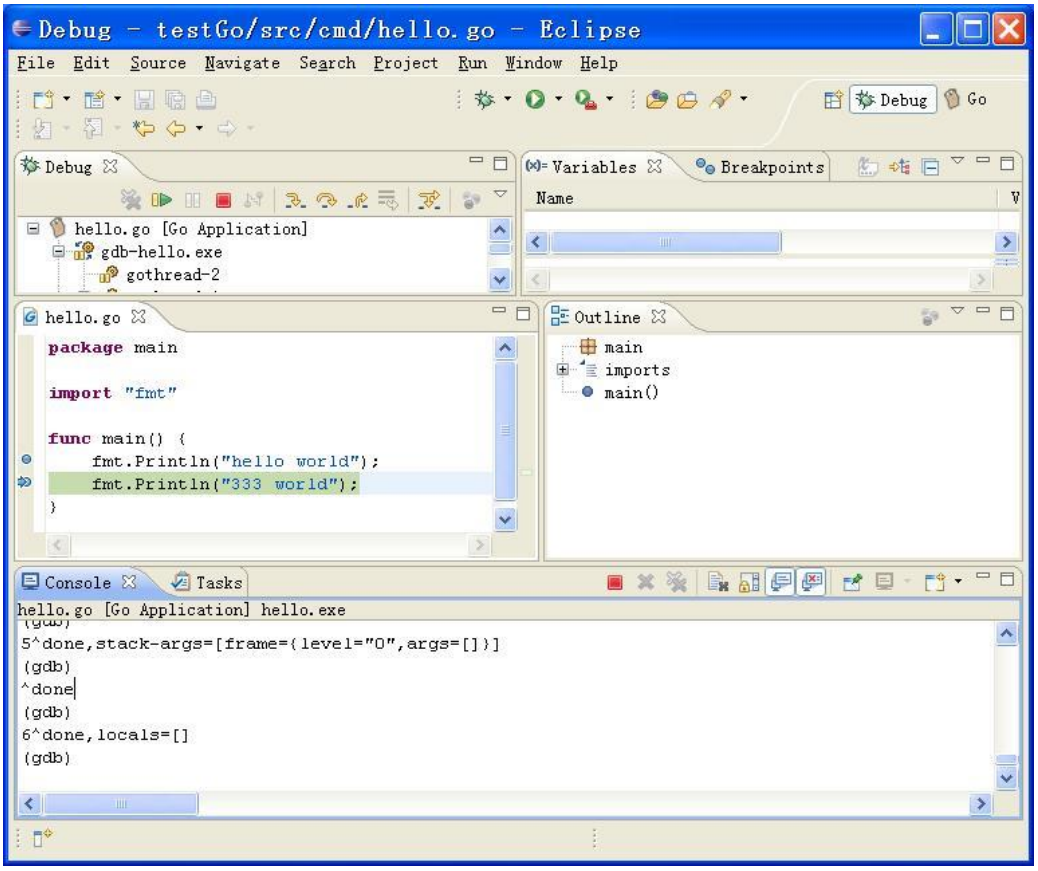
6. 测试是否成功

新建一个 go 工程，再建立一个 hello.go。如下图：



新建项目编辑文件

调试如下（要在 console 中用输入命令来调试）：



4 Go 语言结构

在我们开始学习 GO 编程语言的基础构建模块前，让我们先来了解 Go 语言最简单程序的结构。

Go 语言的基础组成有以下几个部分：

- 包声明
- 引入包
- 函数
- 变量
- 语句 & 表达式
- 注释

接下来让我们来看下简单的代码，该代码输出了"Hello World!":

```
package main

import "fmt"

func main() {
   /* 这是我的第一个简单的程序 */
   fmt.Println("Hello, World!")
}
```

让我们来看下以上程序的各个部分：

1.第一行代码 package main 定义了包名。你必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main 表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。

2.下一行 import "fmt" 告诉 Go 编译器这个程序需要使用 fmt 包（的函数，或其他元素），fmt 包实现了格式化 IO（输入/输出）的函数。

3.下一行 func main() 是程序开始执行的函数。main 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数（如果有 init() 函数则会先执行该函数）。

4.下一行 /*...*/ 是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 /* 开头，并以 */ 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。

5.下一行 fmt.Println(...) 可以将字符串输出到控制台，并在最后自动增加换行字符 \n。

使用 fmt.Print("hello, world\n") 可以得到相同的结果。

Print 和 Println 这两个函数也支持使用变量，如：fmt.Println(arr)。如果没有特别指定，它们会以默认的打印格式将变量 arr 输出到控制台。

6.当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）；标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 protected ）。

4.1 执行 Go 程序

让我们来看下如何编写 Go 代码并执行它。步骤如下：

1、打开编辑器如 Sublime3，将以上代码添加到编辑器中。

2、将以上代码保存为 hello.go

3、打开命令行，并进入程序文件保存的目录中。

4、输入命令 go run hello.go 并按回车执行代码。

5、如果操作正确你将在屏幕上看到 "Hello World!" 字样的输出。

```
$ go run hello.go Hello,  World!
```

5 Go 语言基础语法

上一章节我们已经了解了 Go 语言的基本组成结构，本章节我们将学习 Go 语言的基础语法。

5.1 Go 标记

Go 程序可以由多个标记组成，可以是关键字，标识符，常量，字符串，符号。如以下 GO 语句由 6 个标记组成：

```
fmt.Println("Hello, World!")
```

6 个标记是(每行一个)：

```
1. fmt
2. .
3. Println
4. (
5. "Hello, World!"
6. )
```

5.2 行分隔符

在 Go 程序中，一行代表一个语句结束。每个语句不需要像 C 家族中的其它语言一样以分号;结尾，因为这些工作都将由 Go 编译器自动完成

如果你打算将多个语句写在同一行，它们则必须使用;人为区分，但在实际开发中我们并不鼓励这种做法。

以下为两个语句：

```
fmt.Println("尹成大魔!")

fmt.Println("Go Go Go")
```

5.3 注释

注释不会被编译，每一个包应该有相关注释。

单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 /* 开头，并以 */ 结尾。如：

```
// 单行注释
/*
 Author by 尹成
 我是多行注释
 */
```

5.4 标识符

标识符用来命名变量、类型等程序实体。一个标识符实际上就是一个或是多个字母(A~Z 和 a~z)数字(0~9)、下划线_组成的序列，但是第一个字符必须是字母或下划线而不能是数字。

标识符 = 字母 {字母 | unicode 数字}。

Go 不允许在标识符中使用标点符号，例如@, $ 和 %。 Go 是一种区分大小写的编程语言。 因此，Manpower 和 manpower 在 Go 中是两个不同的标识符。

以下是有效的标识符：

```
mahesh   kumar   abc   move_name   a_123
myname50   _temp   j   a23b9   retVal
```

以下是无效的标识符：

1ab（以数字开头）

case（Go 语言的关键字）

a+b（运算符是不允许的）

5.5 关键字

下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

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

除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符：

append	bool	byte	cap	close	complex	complex64	complex128	uint16
copy	false	float32	float64	imag	int	int8	int16	uint32

int32	int64	iota	len	make	new	nil	panic	uint64
print	println	real	recover	string	true	uint	uint8	uintptr

程序一般由关键字、常量、变量、运算符、类型和函数组成。

程序中可能会使用到这些分隔符：括号 ()，中括号 [] 和大括号 {}。

程序中可能会使用到这些标点符号：.、,、;、: 和 …。

Go 语言的空格

Go 语言中变量的声明必须使用空格隔开，如：

```
var age int;
```

语句中适当使用空格能让程序看易阅读。

无空格：

```
fruit=apples+oranges;
```

在变量与运算符间加入空格，程序看起来更加美观，如：

```
fruit = apples + oranges;
```

6 Go 语言数据类型

在 Go 编程语言中，数据类型用于声明函数和变量。

数据类型的出现是为了把数据分成所需内存大小不同的数据，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

Go 语言按类别有以下几种数据类型：

编号	类型和说明
1	**布尔类型** - 它们是布尔类型，由两个预定义常量组成：(a)`true`(b)`false`
2	**数字类型** - 它们是算术类型，在整个程序中表示：a)整数类型或 b)浮点值。
3	**字符串类型** - 字符串类型表示字符串值的集合。它的值是一个字节序列。 字符串是不可变的类型，一旦创建后，就不可能改变字符串的内容。预先声明的字符串类型是 `string`。
4	**派生类型：** - 包括(a)指针类型，(b)数组类型，(c)结构类型，(d)联合类型和(e)函数类型(f)切片类型(g)函数类型(h)接口类型(i) 类型

数组类型和结构类型统称为聚合类型。函数的类型指定具有相同参数和结果类型的所有函数的集合。我们将在下一节中看到基本类型，而其他类型将在后续章节中介绍。

6.1 数字类型

预定义与体系结构无关的整数类型是：

编号	类型和说明
1	**uint8** - 无符号 8 位整数(0 到 255)
2	**uint16** - 无符号 16 位整数(0 到 65535)
3	**uint32** - 无符号 32 位整数(0 至 4294967295)
4	**uint64** - 无符号 64 位整数(0 至 18446744073709551615)
5	**int8** - 带符号的 8 位整数(-128 到 127)
6	**int16** - 带符号的 16 位整数(-32768 到 32767)
7	**int32** - 带符号的 32 位整数(-2147483648 至 2147483647)
8	**int64** - 带符号的 64 位整数(-9223372036854775808 至

编号	类型和说明
	9223372036854775807)

6.2 浮点类型

预定义的与体系结构无关的浮点类型是：

编号	类型和说明
1	**float32** - IEEE-754 32 位浮点数
2	**float64** - IEEE-754 64 位浮点数
3	**complex64** - 复数带有 `float32` 实部和虚部
4	**complex128** - 复数带有 float64 实部和虚部

`n` 位整数的值是 `n` 位，并且使用二进制补码算术运算来表示。

6.3 其他数字类型

还有一组具有特定大小的数字类型：

编号	类型和说明
1	**byte** - 与 `uint8` 相同
2	**rune** - 与 `int32` 相同
3	**uint** - 32 或 64 位
4	**int** - 与 `uint` 大小相同
5	**uintptr** - 无符号整数，用于存储指针值的未解释位

7 Go 语言变量

变量只是给程序可以操作的存储区域的名字。Go 中的每个变量都有一个特定的类型，它决定了变量的内存大小和布局；可以存储在存储器内的值的范围；以及可以应用于该变量的一组操作。

变量的名称可以由字母，数字和下划线字符组成。它必须以字母或下划线开头。大写和小写字母是不同的名称，因为 Go 是区分大小写的。

声明变量的一般形式是使用 var 关键字：

```
var 标识符 变量类型
```

7.1 变量声明

第一种，指定变量类型，声明后若不赋值，使用默认值。

```
var 变量名 变量类型
变量名 = 值
```

第二种，根据值自行判定变量类型。

```
var 变量名 = 值
```

第三种，省略 var, 注意 :=左侧的变量不应该是已经声明过的，否则会导致编译错误。这种不带声明格式的只能在函数体中出现

```
v_name := value
// 例如
var a int = 10
var b = 10
c := 10
```

实例如下：

```
package main
var a = "尹成"
var b string = "laosiji"
var c bool

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

```
}
```

以上实例执行结果为：

```
尹成 laosiji false
```

7.2 多变量声明

```
//类型相同多个变量, 非全局变量
var vname1, vname2, vname3 type
vname1, vname2, vname3 = v1, v2, v3

var vname1, vname2, vname3 = v1, v2, v3 //和 python 很像,不需要显示声明类型，自动推断

vname1, vname2, vname3 := v1, v2, v3 //出现在:=左侧的变量不应该是已经被声明过的，否则会导致编译错误


// 这种因式分解关键字的写法一般用于声明全局变量
var (
    vname1 v_type1
    vname2 v_type2
)
```

实例如下：

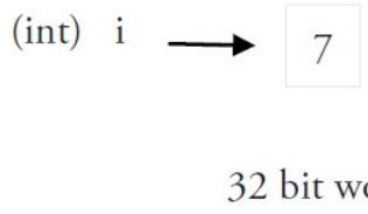
```
package main

var x, y int  //未赋初值，自动设置为 0
var (  // 这种因式分解关键字的写法一般用于声明全局变量
    a int
    b bool
)

var c, d int = 1, 2
var e, f = 123, "hello"

//这种不带声明格式的只能在函数体中出现，
//g, h := 123, "hello"

func main(){
    g, h := 123, "hello"
    println(x, y, a, b, c, d, e, f, g, h)
}
```

以上实例执行结果为：

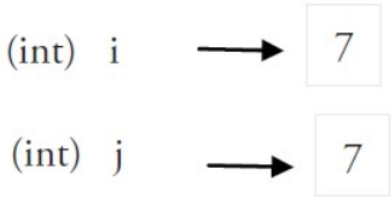
```
0 0 0 false 1 2 123 hello 123 hello
```

7.3 值类型和引用类型

所有像 int、float、bool 和 string 这些基本类型都属于值类型，使用这些类型的变量直接指向存在内存中的值：



当使用等号 = 将一个变量的值赋值给另一个变量时，如：j = i，实际上是在内存中将 i 的值进行了拷贝：



你可以通过 &i 来获取变量 i 的内存地址，例如：0xf840000040（每次的地址都可能不一样）。值类型的变量的值存储在栈中。

内存地址会根据机器的不同而有所不同，甚至相同的程序在不同的机器上执行后也会有不同的内存地址。因为每台机器可能有不同的存储器布局，并且位置分配也可能不同。

更复杂的数据通常会需要使用多个字，这些数据一般使用引用类型保存。

一个引用类型的变量 r1 存储的是 r1 的值所在的内存地址（数字），或内存地址中第一个字所在的位置。

这个内存地址称之为指针，这个指针实际上也被存在另外的某一个字中。

同一个引用类型的指针指向的多个字可以是在连续的内存地址中（内存布局是连续的），这也是计算效率最高的一种存储形式；也可以将这些字分散存放在内存中，每个字都指示了下一个字所在的内存地址。

当使用赋值语句 r2 = r1 时，只有引用（地址）被复制。

如果 r1 的值被改变了，那么这个值的所有引用都会指向被修改后的内容，在这个例子中，r2 也会受到影响。

7.4 简短形式，使用 := 赋值操作符

我们知道可以在变量的初始化时省略变量的类型而由系统自动推断，声明语句写上 var 关键字其实是显得有些多余了，因此我们可以将它们简写为 a := 50 或 b := false。

a 和 b 的类型（int 和 bool）将由编译器自动推断。

这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 := 可以高效地创建一个新的变量，称之为初始化声明。

注意事项

1. 如果在相同的代码块中，我们不可以再次对于相同名称的变量使用初始化声明，当变量 a 在之前已经声明过了以后，又输入了 a := 20 就是不被允许的，编译器会提示错误 no new variables on left side of :=，但是 a = 20 是可以的，因为这是给相同的变量赋予一个新的值。

2.如果你在定义变量 a 之前使用 a=20，则会得到编译错误 undefined: a。

3.如果你声明了一个局部变量却没有在相同的代码块中使用它，同样会得到编译错误，例如下面这个例子当中的变量 a：

```
package main

import "fmt"

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

尝试编译这段代码将得到错误 a declared and not used。

此外，单纯地给 a 赋值也是不够的，这个值必须被使用，所以使用

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

会移除错误。

5.全局变量是允许声明但不使用。 同一类型的多个变量可以声明在同一行，如：

```
var a, b, c int
```

6.多变量可以在同一行进行赋值，如：

```
a, b, c = 5, 7, "abc"
```

上面这行假设了变量 a，b 和 c 都已经被声明，否则的话应该这样使用：

```
a, b, c := 5, 7, "abc"
```

右边的这些值以相同的顺序赋值给左边的变量，所以 a 的值是 5， b 的值是 7，c 的值是 "abc"。

这被称为 并行 或 同时 赋值。

7.如果你想要交换两个变量的值，则可以简单地使用 a, b = b, a。

空白标识符 _ 也被用于抛弃值，如值 5 在：_, b = 5, 7 中被抛弃。

_ 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

并行赋值也被用于当一个函数返回多个返回值时，比如这里的 val 和错误 err 是通过调用 Func1 函数同时得到：val, err = Func1(var1)。

8 Go 语言常量

常量是指程序在执行过程中可能不会改变的固定值。 这些固定值也称为文字。

常量可以是任何基本数据类型，如整数常量，浮点常量，字符常量或字符串常量。 还有枚举常量。

常量一般会被编译器视为常规变量，只是它们的值不能在定义之后被修改。

8.1 整型常量

可以是十进制，八进制或十六进制常数。 前缀指定基数：前缀是 0x 或 0X 为十六进制，前缀是 0 的为八进制，十进制的前缀则无任何内容。

整型常量还可以有一个后缀，它是 U 和 L 的组合，分别用于 unsigned 和 long。后缀可以是大写或小写，可以是任意顺序。

这里是一些有效的整型常量的例子：

```
212         /* 合法 */
215u        /* 合法 */
```

```
0xFeeL      /* 合法 */
078         /* 非法：8 不是 8 进制中的值 */
032UU       /* 非法：不能添加两个 U */
```

其他一些例子：

```
85         /* 10 进制 */
0213       /* 8 进制 */
0x4b       /* 16 进制 */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

8.2 浮点型常量

有整数部分，小数点，小数部分和指数部分。您可以以十进制形式或指数形式来表示浮点文字。

在使用十进制形式表示时，必须包括小数点，指数或两者，并且在使用指数形式表示时，必须包括整数部分，小数部分或两者。带符号的指数由 e 或 E 引入。

下面是一些浮点文字的示例：

```
3.14159       /* 合法 */
314159E-5L    /* 合法 */
510E          /* 非法：无效的指数 */
210f          /* 非法：没有小数*/
.e55          /* 非法：没有整数部分 */
```

8.3 转义序列

Go 中有一些字符，当它们前面有一个反斜杠，它们将具有特殊的意义，它们用于表示类似换行符(\n)或制表符(\t)。 这里，有一些这样的转义序列代码的列表：

转义序列	含义
\\	\字符
\'	'字符
\"	"字符
\?	?字符
\a	警报或响铃
\b	退格
\f	换页

转义序列	含义
\n	新行
\r	回车
\t	水平制表格
\v	水直制表格
\ooo	八位数字一到三位数
\xhh...	一位或多位的十六进制数

以下是显示几个转义序列字符的示例：

```
package main

import "fmt"

func main() {
   fmt.Printf("Hello\tWorld!")
}
```

当上述代码被编译和执行时，它产生以下结果：

```
Hello   World!
```

8.4 字符串常量

字符串文字或常量用双引号""括起来。字符串包含与字符文字类似的字符：纯字符，转义序列和通用字符。可以使用字符串文字将长行拆分为多个行，并使用空格分隔它们。

这里是一些字符串文字的例子。下面这三种形式都是相同的字符串。

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

常量的定义格式：

```
const 标识符[类型] = 值
```

你可以省略类型说明符 [类型]，因为编译器可以根据变量的值来推断其类型。

显式类型定义： const b string = "abc"

隐式类型定义： const b = "abc"

多个相同类型的声明可以简写为：

```
const c_name1, c_name2 = value1, value2
```

以下实例演示了常量的应用：

```
package main

import "fmt"

func main() {
   const LENGTH int = 10
   const WIDTH int = 5
   var area int
   const a, b, c = 1, false, "str" //多重赋值

   area = LENGTH * WIDTH
   fmt.Printf("面积为 : %d", area)
   println()
   println(a, b, c)
}
```

以上实例运行结果为：

```
面积为 : 50
1 false str
```

常量还可以用作枚举：

```
const (
    Unknown = 0
    Female = 1
    Male = 2
)
```

数字 0、1 和 2 分别代表未知性别、女性和男性。

常量可以用 len(), cap(), unsafe.Sizeof()函数计算表达式的值。常量表达式中，函数必须是内置函数，否则编译不过：

```
package main

import "unsafe"
```

```
const (
    a = "abc"
    b = len(a)
    c = unsafe.Sizeof(a)
)

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

以上实例运行结果为：

```
abc 3 16
```

8.5 iota

iota，特殊常量，可以认为是一个可以被编译器修改的常量。

在每一个 const 关键字出现时，被重置为 0，然后再下一个 const 出现之前，每出现一次 iota，其所代表的数字会自动增加 1。

iota 可以被用作枚举值：

```
const (
    a = iota
    b = iota
    c = iota
)
```

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1；所以 a=0, b=1, c=2 可以简写为如下形式：

```
const (
    a = iota
    b
    c
)
```

8.5.1 iota 用法

```
package main

import "fmt"

func main() {
    const (
```

```
            a = iota   //0
            b          //1
            c          //2
            d = "ha"   //独立值，iota += 1
            e          //"ha"   iota += 1
            f = 100    //iota +=1
            g          //100  iota +=1
            h = iota   //7,恢复计数
            i          //8
    )
    fmt.Println(a,b,c,d,e,f,g,h,i)
}
```

以上实例运行结果为：

```
0 1 2 ha ha 100 100 7 8
```

再看个有趣的的 iota 实例：

```
package main

import "fmt"
const (
    i=1<<iota
    j=3<<iota
    k
    l
)

func main() {
    fmt.Println("i=",i)
    fmt.Println("j=",j)
    fmt.Println("k=",k)
    fmt.Println("l=",l)
}
```

以上实例运行结果为：

```
i= 1
j= 6
k= 12
l= 24
```

iota 表示从 0 开始自动加 1，所以 **i=1<<0**, **j=3<<1**（<<表示左移的意思），即：i=1,j=6，这没问题，关键在 k 和 l，从输出结果看 **k=3<<2**，**l=3<<3**。

简单表述:
i=1：左移 0 位,不变仍为 1;

- **j=3**：左移 1 位,变为二进制 110, 即 6;
- **k=3**：左移 2 位,变为二进制 1100, 即 12;
- **l=3**：左移 2 位,变为二进制 11000,即 24。

9 Go 运算符

运算符用于在程序运行时执行数学或逻辑运算。

Go 语言内置的运算符有：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 其他运算符

接下来让我们来详细看看各个运算符的介绍。

9.1 算术运算符

下表列出了所有 Go 语言的算术运算符。假定 A 值为 10，B 值为 20。

运算符	描述	实例
+	相加	A + B 输出结果 30
-	相减	A - B 输出结果 -10
*	相乘	A * B 输出结果 200
/	相除	B / A 输出结果 2
%	求余	B % A 输出结果 0
++	自增	A++ 输出结果 11

--	自减	A-- 输出结果 9

以下实例演示了各个算术运算符的用法：

```
package main

import "fmt"

func main() {

   var a int = 21
   var b int = 10
   var c int

   c = a + b
   fmt.Printf("第一行 - c 的值为 %d\n", c )
   c = a - b
   fmt.Printf("第二行 - c 的值为 %d\n", c )
   c = a * b
   fmt.Printf("第三行 - c 的值为 %d\n", c )
   c = a / b
   fmt.Printf("第四行 - c 的值为 %d\n", c )
   c = a % b
   fmt.Printf("第五行 - c 的值为 %d\n", c )
   a++
   fmt.Printf("第六行 - a 的值为 %d\n", a )
   a=21   // 为了方便测试，a 这里重新赋值为 21
   a--
   fmt.Printf("第七行 - a 的值为 %d\n", a )
}
```

以上实例运行结果：

```
第一行 - c 的值为 31
第二行 - c 的值为 11
第三行 - c 的值为 210
第四行 - c 的值为 2
第五行 - c 的值为 1
第六行 - a 的值为 22
第七行 - a 的值为 20
```

9.2 关系运算符

下表列出了所有 Go 语言的关系运算符。假定 A 值为 10，B 值为 20。

运算符	描述	实例
==	检查两个值是否相等，如果相等返回 True 否则返回 False。	(A == B) 为 False
!=	检查两个值是否不相等，如果不相等返回 True 否则返回 False。	(A != B) 为 True
>	检查左边值是否大于右边值，如果是返回 True 否则返回 False。	(A > B) 为 False
<	检查左边值是否小于右边值，如果是返回 True 否则返回 False。	(A < B) 为 True
>=	检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。	(A >= B) 为 False
<=	检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。	(A <= B) 为 True

以下实例演示了关系运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 21
   var b int = 10

   if( a == b ) {
      fmt.Printf("第一行 - a 等于 b\n" )
   } else {
      fmt.Printf("第一行 - a 不等于 b\n" )
   }
   if ( a < b ) {
      fmt.Printf("第二行 - a 小于 b\n" )
   } else {
      fmt.Printf("第二行 - a 不小于 b\n" )
   }
```

```
   if ( a > b ) {
      fmt.Printf("第三行 - a 大于 b\n" )
   } else {
      fmt.Printf("第三行 - a 不大于 b\n" )
   }
   /* Lets change value of a and b */
   a = 5
   b = 20
   if ( a <= b ) {
      fmt.Printf("第四行 - a 小于等于 b\n" )
   }
   if ( b >= a ) {
      fmt.Printf("第五行 - b 大于等于 a\n" )
   }
}
```

以上实例运行结果：

```
第一行 - a 不等于 b
第二行 - a 不小于 b
第三行 - a 大于 b
第四行 - a 小于等于 b
第五行 - b 大于等于 a
```

9.3 逻辑运算符

下表列出了所有 Go 语言的逻辑运算符。假定 A 值为 True，B 值为 False。

运算符	描述	实例
&&	逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。	(A && B) 为 False
||	逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。	(A || B) 为 True
!	逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。	!(A && B) 为 True

以下实例演示了逻辑运算符的用法：

```
package main
```

```
import "fmt"

func main() {
   var a bool = true
   var b bool = false
   if ( a && b ) {
      fmt.Printf("第一行 - 条件为 true\n" )
   }
   if ( a || b ) {
      fmt.Printf("第二行 - 条件为 true\n" )
   }
   /* 修改 a 和 b 的值 */
   a = false
   b = true
   if ( a && b ) {
      fmt.Printf("第三行 - 条件为 true\n" )
   } else {
      fmt.Printf("第三行 - 条件为 false\n" )
   }
   if ( !(a && b) ) {
      fmt.Printf("第四行 - 条件为 true\n" )
   }
}
```

以上实例运行结果：

```
第二行 - 条件为 true
第三行 - 条件为 false
第四行 - 条件为 true
```

9.4 位运算符

位运算符对整数在内存中的二进制位进行操作。

下表列出了位运算符 &, |, 和 ^ 的计算：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0

1	0	0	1	1

假定 A = 60; B = 13; 其二进制数转换为：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001
```

Go 语言支持的位运算符如下表所示。假定 A 为 60，B 为 13：

运算符	描述	实例
&	按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。	(A & B) 结果为 12, 二进制为 0000 1100
|	按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或	(A | B) 结果为 61, 二进制为 0011 1101
^	按位异或运算符"^"是双目运算符。 其功能是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。	(A ^ B) 结果为 49, 二进制为 0011 0001
<<	左移运算符"<<"是双目运算符。左移 n 位就是乘以 2 的 n 次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补 0。	A << 2 结果为 240 ，二进制为 1111 0000
>>	右移运算符">>"是双目运算符。右移 n 位就是除以 2 的 n 次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边	A >> 2 结果为 15 ，二进制为

	的数指定移动的位数。	0000 1111

以下实例演示了逻辑运算符的用法：

```
package main

import "fmt"

func main() {

   var a uint = 60    /* 60 = 0011 1100 */
   var b uint = 13    /* 13 = 0000 1101 */
   var c uint = 0

   c = a & b       /* 12 = 0000 1100 */
   fmt.Printf("第一行 - c 的值为 %d\n", c )

   c = a | b       /* 61 = 0011 1101 */
   fmt.Printf("第二行 - c 的值为 %d\n", c )

   c = a ^ b       /* 49 = 0011 0001 */
   fmt.Printf("第三行 - c 的值为 %d\n", c )

   c = a << 2     /* 240 = 1111 0000 */
   fmt.Printf("第四行 - c 的值为 %d\n", c )

   c = a >> 2     /* 15 = 0000 1111 */
   fmt.Printf("第五行 - c 的值为 %d\n", c )
}
```

以上实例运行结果：

```
第一行 - c 的值为 12
第二行 - c 的值为 61
第三行 - c 的值为 49
第四行 - c 的值为 240
第五行 - c 的值为 15
```

9.5 赋值运算符

下表列出了所有 Go 语言的赋值运算符。

运算符	描述	实例

=	简单的赋值运算符，将一个表达式的值赋给一个左值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C *= A 等于 C = C * A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A
<<=	左移后赋值	C <<= 2 等于 C = C << 2
>>=	右移后赋值	C >>= 2 等于 C = C >> 2
&=	按位与后赋值	C &= 2 等于 C = C & 2
^=	按位异或后赋值	C ^= 2 等于 C = C ^ 2
|=	按位或后赋值	C |= 2 等于 C = C | 2

以下实例演示了赋值运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 21
   var c int

   c =  a
   fmt.Printf("第 1 行 - =  运算符实例，c 值为 = %d\n", c )

   c +=  a
   fmt.Printf("第 2 行 - += 运算符实例，c 值为 = %d\n", c )

   c -=  a
   fmt.Printf("第 3 行 - -= 运算符实例，c 值为 = %d\n", c )
```

```
   c *=  a
   fmt.Printf("第 4 行 - *= 运算符实例，c 值为 = %d\n", c )

   c /=  a
   fmt.Printf("第 5 行 - /= 运算符实例，c 值为 = %d\n", c )

   c  = 200;

   c <<=  2
   fmt.Printf("第 6 行  - <<= 运算符实例，c 值为 = %d\n", c )

   c >>=  2
   fmt.Printf("第 7 行 - >>= 运算符实例，c 值为 = %d\n", c )

   c &=  2
   fmt.Printf("第 8 行 - &= 运算符实例，c 值为 = %d\n", c )

   c ^=  2
   fmt.Printf("第 9 行 - ^= 运算符实例，c 值为 = %d\n", c )

   c |=  2
   fmt.Printf("第 10 行 - |= 运算符实例，c 值为 = %d\n", c )

}
```

以上实例运行结果：

```
第 1 行 - =  运算符实例，c 值为 = 21
第 2 行 - += 运算符实例，c 值为 = 42
第 3 行 - -= 运算符实例，c 值为 = 21
第 4 行 - *= 运算符实例，c 值为 = 441
第 5 行 - /= 运算符实例，c 值为 = 21
第 6 行  - <<= 运算符实例，c 值为 = 800
第 7 行 - >>= 运算符实例，c 值为 = 200
第 8 行 - &= 运算符实例，c 值为 = 0
第 9 行 - ^= 运算符实例，c 值为 = 2
第 10 行 - |= 运算符实例，c 值为 = 2
```

9.6 其他运算符

下表列出了 Go 语言的其他运算符。

运算符	描述	实例

&	取地址运算符	&a; 将给出变量的实际地址。
*	间接寻址运算符。	*a; 是一个指针变量

以下实例演示了其他运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 4
   var b int32
   var c float32
   var ptr *int

   /* 运算符实例 */
   fmt.Printf("第 1 行 - a 变量类型为 = %T\n", a );
   fmt.Printf("第 2 行 - b 变量类型为 = %T\n", b );
   fmt.Printf("第 3 行 - c 变量类型为 = %T\n", c );

   /*  & 和 * 运算符实例 */
   ptr = &a    /* 'ptr' 包含了 'a' 变量的地址 */
   fmt.Printf("a 的值为  %d\n", a);
   fmt.Printf("*ptr 为 %d\n", *ptr);
}
```

以上实例运行结果：

```
第 1 行 - a 变量类型为 = int
第 2 行 - b 变量类型为 = int32
第 3 行 - c 变量类型为 = float32
a 的值为  4
*ptr 为 4
```

9.7 运算符优先级

有些运算符拥有较高的优先级，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：

分类	描述	关联性
后缀	() [] -> . ++ - -	左到右
一元	+ - ! ~ ++ - - (type) * & sizeof	右到左
乘法	* / %	左到右
加法	+ -	左到右
移位	<< >>	左到右
关系	< <= > >=	左到右
相等	== !=	左到右
按位 AND	&	左到右
按位 XOR	^	左到右
按位 OR	|	左到右
逻辑 AND	&&	左到右
逻辑 OR	||	左到右
条件	?:	右到左
分配	= += -= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

以上实例运行结果：

```
package main
```

```
import "fmt"

func main() {
   var a int = 20
   var b int = 10
   var c int = 15
   var d int = 5
   var e int;

   e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   fmt.Printf("(a + b) * c / d 的值为 : %d\n",  e );

   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   fmt.Printf("((a + b) * c) / d 的值为  : %d\n" ,  e );

   e = (a + b) * (c / d);   // (30) * (15/5)
   fmt.Printf("(a + b) * (c / d) 的值为  : %d\n",  e );

   e = a + (b * c) / d;     //  20 + (150/5)
   fmt.Printf("a + (b * c) / d 的值为  : %d\n" ,  e );
}
```

以上实例运行结果：

```
(a + b) * c / d 的值为 : 90
((a + b) * c) / d 的值为  : 90
(a + b) * (c / d) 的值为  : 90
a + (b * c) / d 的值为  : 50
```

10 Go 语言类型转换

类型转换用于将一种数据类型的变量转换为另外一种类型的变量。Go 语言类型转换基本格式如下：

```
type_name(expression)
```

type_name 为类型，expression 为表达式。

以下实例中将整型转化为浮点型，并计算结果，将结果赋值给浮点型变量：

```
package main

import "fmt"

func main() {
   var sum int = 17
```

```
    var count int = 5
    var mean float32

    mean = float32(sum)/float32(count)
    fmt.Printf("mean 的值为: %f\n",mean)
}
```

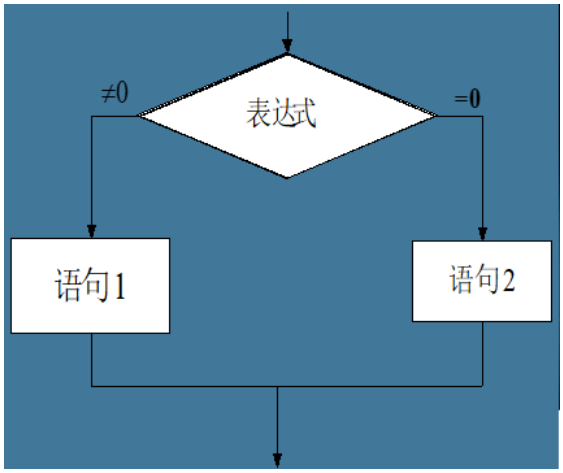
以上实例执行输出结果为：

```
mean 的值为: 3.400000
```

11 Go 语言条件语句

条件语句需要开发者通过指定一个或多个条件，并通过测试条件是否为 true 来决定是否执行指定语句，并在条件为 false 的情况在执行另外的语句。

下图展示了程序语言中条件语句的结构：



Go 语言提供了以下几种条件判断语句：

语句	描述
if 语句	**if 语句** 由一个布尔表达式后紧跟一个或多个语句组成。
if...else 语句	**if 语句** 后可以使用可选的 **else 语句**, else 语句中的表达式在布尔表达式为 false 时执行。
if 嵌套语句	你可以在 **if** 或 **else if** 语句中嵌入一个或多个 **if** 或 **else if** 语句。
switch 语句	**switch** 语句用于基于不同条件执行不同动作。

select 语句	**select** 语句类似于 **switch** 语句，但是 select 会随机执行一个可运行的 case。如果没有 case 可运行，它将阻塞，直到有 case 可运行。

11.1 if 简单语句

```
package main

import "fmt"

func main() {

   /* 定义局部变量 */

   var a int = 10

   /* 使用 if 语句判断布尔表达式 */

   if a < 20 {

       /* 如果条件为 true 则执行以下语句 */

       fmt.Printf("a 小于 20\n" )

   }

   fmt.Printf("a 的值为 : %d\n", a)

}
```

以上代码执行结果为：

```
a 小于 20

a 的值为 : 10
```

11.2 if 语句的表达式前可以添加变量的声明

语句可以在条件语句之前; 在此语句中声明的任何变量在 if 语句的所有分支中都可用

```
package main

import "fmt"
```

```
func main() {


    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

11.3 if...else 语句

if 语句 后可以使用可选的 else 语句, else 语句中的表达式在布尔表达式为 false 时执行。

```
package main


import "fmt"


func main() {
   /* 局部变量定义 */
   var a int = 100;


   /* 判断布尔表达式 */
   if a < 20 {
       /* 如果条件为 true 则执行以下语句 */
       fmt.Printf("a 小于 20\n" );
   } else {
```

```
      /* 如果条件为 false 则执行以下语句 */
      fmt.Printf("a 不小于 20\n" );
   }
   fmt.Printf("a 的值为 : %d\n", a);

}
```

以上代码执行结果为：

```
a 不小于 20
a 的值为 : 100
```

11.4 if 嵌套语句

```
if 布尔表达式 1 {
   /* 在布尔表达式 1 为 true 时执行 */
   if 布尔表达式 2 {
      /* 在布尔表达式 2 为 true 时执行 */
   }
}
```

你可以以同样的方式在 if 语句中嵌套 **else if...else** 语句

```
package main
import "fmt"
func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200
```

```
   /* 判断条件 */
   if a == 100 {
       /* if 条件语句为 true 执行 */
       if b == 200 {
          /* if 条件语句为 true 执行 */
          fmt.Printf("a 的值为 100 ， b 的值为 200\n" );
       }
   }
   fmt.Printf("a 值为 : %d\n", a );
   fmt.Printf("b 值为 : %d\n", b );
}
```

以上代码执行结果为：

```
a 的值为 100 ， b 的值为 200
a 值为 : 100
b 值为 : 200
```

11.5 switch 语句

switch 语句用于基于不同条件执行不同动作，每一个 case 分支都是唯一的，从上直下逐一测试，直到匹配为止。。

switch 语句执行的过程从上至下，直到找到匹配项，匹配项后面也不需要再加 break

Go 编程语言中 switch 语句的语法如下：

```
switch var1 {
    case val1:
        ...
    case val2:
```

```
        ...
    default:
        ...
}
```

变量 var1 可以是任何类型，而 val1 和 val2 则可以是同类型的任意值。类型不被局限于常量或整数，但必须是相同的类型；或者最终结果为相同类型的表达式。

您可以同时测试多个可能符合条件的值，使用逗号分割它们，例如：case val1, val2, val3。

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var grade string = "B"
   var marks int = 90

   switch marks {  //使用常量
      case 90: grade = "A"
      case 80: grade = "B"
      case 50,60,70 : grade = "C"
      default: grade = "D"
   }

   switch {   //使用表达式
      case grade == "A" :
         fmt.Printf("优秀!\n" )
      case grade == "B", grade == "C" :
         fmt.Printf("良好\n" )
      case grade == "D" :
         fmt.Printf("及格\n" )
      case grade == "F":
         fmt.Printf("不及格\n" )
      default:
         fmt.Printf("差\n" );
   }
   fmt.Printf("你的等级是 %s\n", grade );
}
```

以上代码执行结果为：

```
优秀!
你的等级是 A
```

除此之外，在 case 中你还可以用逗号来分隔多个表达式

```
package main

import "fmt"
import "time"

func main() {
switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("It's the weekend")
    default:
        fmt.Println("It's a weekday")
    }
}
```

11.6 Switch Type

switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际存储的变量类型。

Type Switch 语法格式如下：

```
switch x.(type){
    case type:
       statement(s);
    case type:
       statement(s);
    /* 你可以定义任意个数的 case */
    default: /* 可选 */
       statement(s);
}
```

实例：

```
package main

import "fmt"

func main() {
   var x interface{}

   switch i := x.(type) {
```

```
        case nil:
            fmt.Printf(" x 的类型 :%T",i)
        case int:
            fmt.Printf("x 是 int 型")
        case float64:
            fmt.Printf("x 是 float64 型")
        case func(int) float64:
            fmt.Printf("x 是 func(int) 型")
        case bool, string:
            fmt.Printf("x 是 bool 或 string 型" )
        default:
            fmt.Printf("未知型")
    }
}
```

以上代码执行结果为：

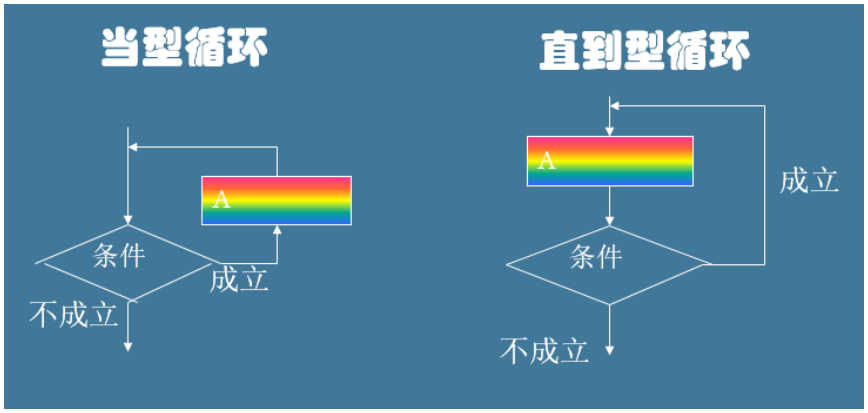
```
x 的类型 :<nil>
```

11.7 select

select 是 Go 中的一个控制结构，类似于用于通信的 switch 语句。每个 case 必须是一个通信操作，要么是发送要么是接收。
由于 select 只能和通道结合使用，因此学习完通道后在后面介绍。

12 Go 语言循环语句

在不少实际问题中有许多具有规律性的重复操作，因此在程序中就需要重复执行某些语句。
以下为大多编程语言循环程序的流程图：



Go 语言提供了以下几种类型循环处理语句：

循环类型	描述
for 循环	重复执行语句块
循环嵌套	在 for 循环中嵌套一个或多个 for 循环

12.1 for

Go 语言的 For 循环有 3 中形式，只有其中的一种使用分号。

和 C 语言的 for 一样：

```
for init; condition; post { }
```

和 C 的 while 一样：

```
for condition { }
```

和 C 的 for(;;) 一样：

```
for { }
```

说明：
init： 一般为赋值表达式，给控制变量赋初值；
condition： 关系表达式或逻辑表达式，循环控制条件；
post： 一般为赋值表达式，给控制变量增量或减量。

for 语句执行过程如下：

①先对表达式 1 赋初值；

②判别赋值表达式 init 是否满足给定条件，若其值为真，满足循环条件，则执行循环体内语句，然后执行 post，进入第二次循环，再判别 condition；否则判断 condition 的值为假，不满足条件，就终止 for 循环，执行循环体外语句。

for 循环的 range 格式可以对 slice、map、数组、字符串等进行迭代循环。格式如下：

```
for key, value := range oldMap {
    newMap[key] = value
}
```

示例：

```
package main

import "fmt"

func main() {
```

```
   var b int = 15
   var a int

   numbers := [6]int{1, 2, 3, 5}

   /* for 循环 */
   for a := 0; a < 10; a++ {
      fmt.Printf("a 的值为: %d\n", a)
   }

   for a < b {
      a++
      fmt.Printf("a 的值为: %d\n", a)
   }

   for i,x:= range numbers {
      fmt.Printf("第 %d 位 x 的值 = %d\n", i,x)
   }
}
```

以上实例运行输出结果为:

```
a 的值为: 0
a 的值为: 1
a 的值为: 2
a 的值为: 3
a 的值为: 4
a 的值为: 5
a 的值为: 6
a 的值为: 7
a 的值为: 8
a 的值为: 9
a 的值为: 1
a 的值为: 2
a 的值为: 3
a 的值为: 4
a 的值为: 5
a 的值为: 6
a 的值为: 7
a 的值为: 8
a 的值为: 9
a 的值为: 10
a 的值为: 11
a 的值为: 12
```

```
a 的值为: 13
a 的值为: 14
a 的值为: 15
第 0 位 x 的值 = 1
第 1 位 x 的值 = 2
第 2 位 x 的值 = 3
第 3 位 x 的值 = 5
第 4 位 x 的值 = 0
第 5 位 x 的值 = 0
```

12.2 for 循环的嵌套

以下为 Go 语言嵌套循环的格式：

```
for [condition | ( init; condition; increment ) | Range]

{

   for [condition | ( init; condition; increment ) | Range]

   {

      statement(s);

   }

   statement(s);

}
```

12.2.1 实例

以下实例使用循环嵌套来输出 2 到 100 间的素数：

```
package main



import "fmt"



func main() {

   /* 定义局部变量 */

   var i, j int
```

```
   for i=2; i < 100; i++ {
      for j=2; j <= (i/j); j++ {
         if(i%j==0) {
            break; // 如果发现因子，则不是素数
         }
      }
      if(j > (i/j)) {
         fmt.Printf("%d  是素数\n", i);
      }
   }
}
```

以上实例运行输出结果为:

```
2  是素数
3  是素数
5  是素数
7  是素数
11  是素数
13  是素数
17  是素数
19  是素数
23  是素数
29  是素数
31  是素数
37  是素数
```

```
41 是素数
43 是素数
47 是素数
53 是素数
59 是素数
61 是素数
67 是素数
71 是素数
73 是素数
79 是素数
83 是素数
89 是素数
97 是素数
```

12.3 循环控制语句

循环控制语句可以控制循环体内语句的执行过程。

GO 语言支持以下几种循环控制语句：

控制语句	描述
break 语句	经常用于中断当前 for 循环或跳出 switch 语句
continue 语句	跳过当前循环的剩余语句，然后继续进行下一轮循环。
goto 语句	将控制转移到被标记的语句。

12.3.1 break 语句：跳出循环

Go 语言中 break 语句用于以下两方面：
用于循环语句中跳出循环，并开始执行循环之后的语句。
break 在 switch（开关语句）中在执行一条 case 后跳出语句的作用。
break 语法格式如下：

```
package main
```

```
import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* for 循环 */
   for a < 20 {
      fmt.Printf("a 的值为 : %d\n", a);
      a++;
      if a > 15 {
         /* 使用 break 语句跳出循环 */
         break;
      }
   }
}
```

以上实例执行结果为：

```
a 的值为 : 10
a 的值为 : 11
a 的值为 : 12
a 的值为 : 13
a 的值为 : 14
a 的值为 : 15
```

12.3.2 continue

Go 语言的 continue 语句 有点像 break 语句。但是 continue 不是跳出循环，而是跳过当前循环执行下一次循环语句。

for 循环中，执行 continue 语句会触发 for 增量语句的执行。

语法

continue 语法格式如下：

```
continue;

package main



import "fmt"
```

```
func main() {

   /* 定义局部变量 */

   var a int = 10



   /* for 循环 */

   for a < 20 {

      if a == 15 {

         /* 跳过此次循环 */

         a = a + 1;

         continue;

      }

      fmt.Printf("a 的值为 : %d\n", a);

      a++;

   }

}
```

以上实例执行结果为：

```
a 的值为 : 10

a 的值为 : 11

a 的值为 : 12

a 的值为 : 13

a 的值为 : 14

a 的值为 : 16

a 的值为 : 17

a 的值为 : 18

a 的值为 : 19
```

12.3.3 Goto

Go 语言的 goto 语句可以无条件地转移到过程中指定的行。

goto 语句通常与条件语句配合使用。可用来实现条件转移， 构成循环，跳出循环体等功能。

但是，在结构化程序设计中一般不主张使用 goto 语句， 以免造成程序流程的混乱，使理解和调试程序都产生困难。

语法

goto 语法格式如下：

```
goto label;

..

.

label: statement;
```

```
package main



import "fmt"



func main() {

   /* 定义局部变量 */

   var a int = 10



   /* 循环 */

   LOOP: for a < 20 {

      if a == 15 {

         /* 跳过迭代 */

         a = a + 1

         goto LOOP

      }
```

```
      fmt.Printf("a 的值为 : %d\n", a)

      a++

   }

}
```

以上实例执行结果为：

```
a 的值为 : 10

a 的值为 : 11

a 的值为 : 12

a 的值为 : 13

a 的值为 : 14

a 的值为 : 16

a 的值为 : 17

a 的值为 : 18

a 的值为 : 19
```

12.4 无限循环

如果循环中条件语句永远不为 false 则会进行无限循环，我们可以通过 for 循环语句中只设置一个条件表达式来执行无限循环：

```
package main

import "fmt"

func main() {
    for true  {
        fmt.Printf("这是无限循环。\n");
    }
}
```

13 Go 语言字符串

在 Go 编程中广泛使用的字符串是只读字节。在 Go 编程语言中，字符串是切片。Go 平台提供了各种库来操作字符串。

- unicode
- 正则表达式
- 字符串

13.1 创建字符串

创建字符串的最直接的方法如下：

```
var greeting = "Hello world!"
```

每当遇到代码中的字符串时，编译器将创建一个字符串对象，其值为“`Hello world!`”。字符串文字持有有效的 UTF-8 序列称为符文。字符串可保存任意字节。参考如下代码

```
package main

import "fmt"

func main() {
   var greeting =  "Hello world!"


   fmt.Printf("normal string: ")
   fmt.Printf("%s", greeting)
   fmt.Printf("\n")
   fmt.Printf("hex bytes: ")
   for i := 0; i < len(greeting); i++ {
       fmt.Printf("%x ", greeting[i])
   }
   fmt.Printf("\n")
//%+q 跳过不可打印的字符和非 ASCII 字符
   const sampleText = "\xbd\xb2\x3d\xbc\x20\xe2\x8c\x98"


   fmt.Printf("quoted string: ")
   fmt.Printf("%+q", sampleText)
   fmt.Printf("\n")
}
```

上面代码执行后，将产生以下结果：

```
normal string: Hello world!

hex bytes: 48 65 6c 6c 6f 20 77 6f 72 6c 64 21

quoted string: "\xbd\xb2=\xbc \u2318"
```

注意：字符串文字是不可变的，因此一旦创建后，字符串文字就不能更改了。

13.2 字符串长度

`len(str)`方法返回包含在字符串文字中的字节数。请参考以下代码示例 -

```
package main

import "fmt"

func main() {

   var greeting =  "Hello world!"


   fmt.Printf("String Length is: ")

   fmt.Println(len(greeting))

}
```

上面代码执行后，将产生以下结果：

```
String Length is : 12
```

13.3 连接字符串

`strings`包包含一个用于连接多个字符串的`join()`方法，其语法如下：

```
strings.Join(sample, " ")
```

`Join`连接数组的元素以创建单个字符串。第二个参数是分隔符，放置在数组的元素之间。
现在来看看下面的例子：

```
package main

import (

 "fmt"

 "strings"

)

func main() {

   greetings :=  []string{"Hello","world!"}

   fmt.Println(strings.Join(greetings, " "))

}
```

这将产生以下结果：

```
Hello world!
```

14 字符串输出格式化

Go 语言为 `printf` 传统中的字符串格式化提供了极好的支持。 以下是常见字符串格式化任务的一些示例。

Go 提供了几种打印“动词”，设计用于格式化一般的值。 例如，打印 `point` 结构的一个实例。

如果值是一个结构体，`%+v` 变体将包括结构体的字段名。

`%#v` 变体打印值的 Go 语法表示，即将生成该值的源代码片段。

要打印值的类型，请使用`%T`。格式化布尔是比较直截了当的。有许多格式化整数的选项。对于标准的 `base-10` 格式化，请使用`%d`。

具体的每个函数，可参考下面的代码 -

```
package main


import "fmt"
import "os"


type point struct {
    x, y int
}


func main() {


    // 打印结构体
    p := point{1, 2}
    fmt.Printf("%v\n", p)


    // 如果值是一个结构体，%+v 变体将包括结构体的字段名。
    fmt.Printf("%+v\n", p)


    // %#v 变体打印值的 Go 语法表示，即将生成该值的源代码片段。
    fmt.Printf("%#v\n", p)
```

```
// 打印类型
fmt.Printf("%T\n", p)

// 打印布尔值
fmt.Printf("%t\n", true)

// 打印整数。
fmt.Printf("%d\n", 123)

// 打印二进制
fmt.Printf("%b\n", 14)

// 打印字符
fmt.Printf("%c\n", 33)

// 打印 16 进制
fmt.Printf("%x\n", 456)

// 打印浮点数
fmt.Printf("%f\n", 78.9)

// 指数型
fmt.Printf("%e\n", 123400000.0)
fmt.Printf("%E\n", 123400000.0)

// 字符串
fmt.Printf("%s\n", "\"string\"")

// 双引号字符串.
fmt.Printf("%q\n", "\"string\"")
```

```
    // 每个字符用两位 16 进制表示。
    fmt.Printf("%x\n", "hex this")

    // 打印指针`.
    fmt.Printf("%p\n", &p)

    // 字符宽度
    fmt.Printf("|%6d|%6d|\n", 12, 345)

    //字符精度
    fmt.Printf("|%6.2f|%6.2f|\n", 1.2, 3.45)

    // 左对齐
    fmt.Printf("|%-6.2f|%-6.2f|\n", 1.2, 3.45)

    //同样可以控制字符的宽度
    fmt.Printf("|%6s|%6s|\n", "foo", "b")

    // 同样字符左对齐.
    fmt.Printf("|%-6s|%-6s|\n", "foo", "b")

    // 合并
    s := fmt.Sprintf("a %s", "string")
    fmt.Println(s)

    // 同样的效果
    fmt.Fprintf(os.Stderr, "an %s\n", "error")
}
```

Go

执行上面代码，将得到以下输出结果 -

```
F:\worksp\golang>go run string-formatting.go
{1 2}
```

```
{x:1 y:2}
main.point{x:1, y:2}
main.point
true
123
1110
!
1c8
78.900000
1.234000e+08
1.234000E+08
"string"
"\"string\""
6865782074686973
0xc042004280
|    12|   345|
|  1.20|  3.45|
|1.20  |3.45  |
|   foo|     b|
|foo   |b     |
a string
an error
```

15 Go 语言函数

函数是基本的代码块，用于执行一个任务。

Go 语言最少有个 main() 函数。

你可以通过函数来划分不同功能，逻辑上每个函数执行的是指定的任务。

函数声明告诉了编译器函数的名称，返回类型，和参数。

Go 语言标准库提供了多种可动用的内置的函数。例如，len() 函数可以接受不同类型参数并返回该类型的长度。如果我们传入的是字符串则返回字符串的长度，如果传入的是数组，则返回数组中包含的元素个数。

15.1 函数定义

Go 语言函数定义格式如下：

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

函数定义解析：

- func：函数由 func 开始声明
- function_name：函数名称，函数名和参数列表一起构成了函数签名。
- parameter list：参数列表，参数就像一个占位符，当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。参数列表指定的是参数类型、顺序、及参数个数。参数是可选的，也就是说函数也可以不包含参数。
- return_types：返回类型，函数返回一列值。return_types 是该列值的数据类型。有些功能不需要返回值，这种情况下 return_types 不是必须的。
- 函数体：函数定义的代码集合。

当有多个相同类型的连续参数时，可以省略类型参数的类型名称，直到声明该类型的最后一个参数。例如：

```
func plusPlus(a, b, c int) int {
    return a + b + c
}
```

示例：

以下实例为 max() 函数的代码，该函数传入两个整型参数 num1 和 num2，并返回这两个参数的最大值：

```
/* 函数返回两个数的最大值 */
func max(num1, num2 int) int {
   /* 声明局部变量 */
   var result int

   if (num1 > num2) {
      result = num1
```

```
   } else {
      result = num2
   }
   return result
}
```

15.2 函数调用

当创建函数时，你定义了函数需要做什么，通过调用该函数来执行指定任务。

调用函数，向函数传递参数，并返回值，例如：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200
   var ret int

   /* 调用函数并返回最大值 */
   ret = max(a, b)

   fmt.Printf( "maxNum= : %d\n", ret )
}

/* 函数返回两个数的最大值 */
func max(num1, num2 int) int {
   /* 定义局部变量 */
   var result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

以上实例在 main() 函数中调用 max（）函数，执行结果为：

```
最大值是 : 200
```

15.3 函数返回多个值

Go 函数可以返回多个值，例如：

```
package main

import "fmt"

func swap(x, y string) (string, string) {
   return y, x
}

func main() {
   a, b := swap("Mahesh", "Kumar")
   fmt.Println(a, b)
}
```

以上实例执行结果为：

```
Kumar Mahesh
```

15.4 可变参数函数

可变参数的函数可以用任何数量的参数来调用。 例如，`fmt.Println()`就是一个常见的可变函数。
下面的例子将任意数量的 `int` 作为参数。可变参数的函数可以通过单独的参数以通常的方式调用。
如果已经在一个切片中有多个参数，使用 `func(slice ...)`函数将它们应用到一个可变函数。

15.5 函数形参的副本机制：

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

以下定义了 swap() 函数：

```
/* 定义相互交换值的函数 */

func swap(x, y int) int {

   var temp int
```

```
   temp = x /* 保存 x 的值 */
   x = y    /* 将 y 值赋给 x */
   y = temp /* 将 temp 值赋给 y*/

   return temp;
}
```

接下来，让我们使用值传递来调用 swap() 函数：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200

   fmt.Printf("交换前 a 的值为 : %d\n", a )
   fmt.Printf("交换前 b 的值为 : %d\n", b )

   /* 通过调用函数来交换值 */
   swap(a, b)

   fmt.Printf("交换后 a 的值 : %d\n", a )
   fmt.Printf("交换后 b 的值 : %d\n", b )
```

```
}


/* 定义相互交换值的函数 */
func swap(x, y int) int {
   var temp int


   temp = x /* 保存 x 的值 */
   x = y    /* 将 y 值赋给 x */
   y = temp /* 将 temp 值赋给 y*/


   return temp;
}
```

以下代码执行结果为：

```
交换前 a 的值为 : 100
交换前 b 的值为 : 200
交换后 a 的值 : 100
交换后 b 的值 : 200
```

15.6 参数的引用传递

引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

引用传递指针参数传递到函数内，以下是交换函数 swap() 使用了引用传递：

```
/* 定义交换值函数*/
func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保持 x 地址上的值 */
   *x = *y      /* 将 y 值赋给 x */
   *y = temp    /* 将 temp 值赋给 y */
```

```
}
```

以下我们通过使用引用传递来调用 swap() 函数：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int= 200

   fmt.Printf("交换前，a 的值 : %d\n", a )
   fmt.Printf("交换前，b 的值 : %d\n", b )

   /* 调用 swap() 函数
   * &a 指向 a 指针，a 变量的地址
   * &b 指向 b 指针，b 变量的地址
   */
   swap(&a, &b)

   fmt.Printf("交换后，a 的值 : %d\n", a )
   fmt.Printf("交换后，b 的值 : %d\n", b )
}

func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保存 x 地址上的值 */
   *x = *y      /* 将 y 值赋给 x */
   *y = temp    /* 将 temp 值赋给 y */
}
```

以上代码执行结果为：

```
交换前，a 的值 : 100
交换前，b 的值 : 200
交换后，a 的值 : 200
交换后，b 的值 : 100
```

15.7 函数作为值

Go 语言可以很灵活的创建函数，并作为值使用。以下实例中我们在定义的函数中初始化一个变

量，该函数仅仅是为了使用内置函数 math.sqrt() ，实例为：

```
package main



import (

   "fmt"

   "math"

)



func main(){

   /* 声明函数变量 */

   getSquareRoot := func(x float64) float64 {

      return math.Sqrt(x)

   }



   /* 使用函数 */

   fmt.Println(getSquareRoot(9))



}
```

以上代码执行结果为：

```
3
```

15.8 匿名函数

Go 语言支持匿名函数，可作为闭包。匿名函数是一个"内联"语句或表达式。匿名函数的优越性在于可以直接使用函数内的变量，不必申明。

以下实例中，我们创建了函数 getSequence() ，返回另外一个函数。该函数的目的是在闭包中递增 i 变量，代码如下：

```
package main


import "fmt"


func getSequence() func() int {
   i:=0
   return func() int {
      i+=1
     return i
   }
}


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


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


   /* 创建新的函数 nextNumber1，并查看结果 */
   nextNumber1 := getSequence()
```

```
   fmt.Println(nextNumber1())

   fmt.Println(nextNumber1())

}
```

以上代码执行结果为：

```
1

2

3

1

2
```

15.9 方法：包含了接受者的函数

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集。语法格式如下：

```
func (variable_name variable_data_type) function_name() [return_type]{
   /* 函数体*/
}
```

下面定义一个结构体类型和该类型的一个方法：

```
package main

import (
   "fmt"
)

/* 定义结构体 */
type Circle struct {
  radius float64
}

func main() {
  var c1 Circle
  c1.radius = 10.00
```

```
  fmt.Println("Area of Circle(c1) = ", c1.getArea())
}

//该 method 属于 Circle 类型对象中的方法
func (c Circle) getArea() float64 {
  //c.radius 即为 Circle 类型对象中的属性
  return 3.14 * c.radius * c.radius
}
```

以上代码执行结果为：

```
Area of Circle(c1) =  314
```

16 Go 语言变量作用域

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。

Go 语言中变量可以在三个地方声明：

- 函数内定义的变量称为局部变量
- 函数外定义的变量称为全局变量
- 函数定义中的变量称为形式参数

接下来让我们具体了解局部变量、全局变量和形式参数。

16.1 局部变量

在函数体内声明的变量称之为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。

以下实例中 main() 函数使用了局部变量 a, b, c：

```
package main

import "fmt"

func main() {
   /* 声明局部变量 */
   var a, b, c int

   /* 初始化参数 */
   a = 10
   b = 20
   c = a + b
```

```
   fmt.Printf ("结果:  a = %d, b = %d and c = %d\n", a, b, c)
}
```

以上实例执行输出结果为：

```
结果:  a = 10, b = 20 and c = 30
```

16.2 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。

全局变量可以在任何函数中使用，以下实例演示了如何使用全局变量：

```
package main

import "fmt"

/* 声明全局变量 */
var g int

func main() {

   /* 声明局部变量 */
   var a, b int

   /* 初始化参数 */
   a = 10
   b = 20
   g = a + b

   fmt.Printf("结果:  a = %d, b = %d and g = %d\n", a, b, g)
}
```

以上实例执行输出结果为：

```
结果:  a = 10, b = 20 and g = 30
```

Go 语言程序中全局变量与局部变量名称可以相同，但是函数内的局部变量会被优先考虑。实例如下：

```
package main

import "fmt"
```

```
/* 声明全局变量 */
var g int = 20

func main() {
   /* 声明局部变量 */
   var g int = 10

   fmt.Printf ("结果：  g = %d\n",  g)
}
```

以上实例执行输出结果为：

```
结果：  g = 10
```

16.3 形式参数

形式参数会作为函数的局部变量来使用。实例如下：

```
package main

import "fmt"

/* 声明全局变量 */
var a int = 20;

func main() {
   /* main 函数中声明局部变量 */
   var a int = 10
   var b int = 20
   var c int = 0

   fmt.Printf("main()函数中 a = %d\n",  a);
   c = sum( a, b);
   fmt.Printf("main()函数中 c = %d\n",  c);
}

/* 函数定义-两数相加 */
func sum(a, b int) int {
   fmt.Printf("sum() 函数中 a = %d\n",  a);
   fmt.Printf("sum() 函数中 b = %d\n",  b);

   return a + b;
}
```

以上实例执行输出结果为：

```
main()函数中 a = 10
sum() 函数中 a = 10
sum() 函数中 b = 20
main()函数中 c = 30
```

16.4 初始化局部和全局变量

不同类型的局部和全局变量默认值为：

数据类型	初始化默认值
int	0
float32	0
pointer	nil

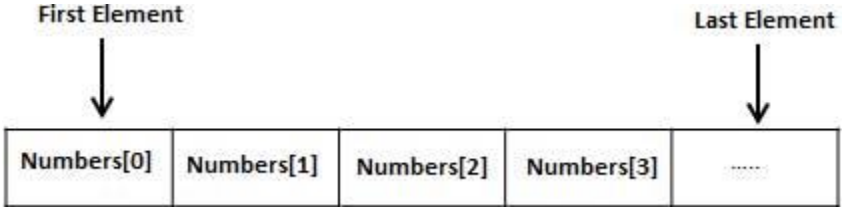
17 Go 语言数组

Go 语言提供了数组类型的数据结构。

数组是具有相同唯一类型的一组已编号且长度固定的数据项序列，这种类型可以是任意的原始类型例如整形、字符串或者自定义类型。

相对于去声明 number0, number1, ..., and number99 的变量，使用数组形式 numbers[0], numbers[1] ..., numbers[99]更加方便且易于扩展。

数组元素可以通过索引（位置）来读取（或者修改），索引从 0 开始，第一个元素索引为 0，第二个索引为 1，以此类推。



17.1 声明数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

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

以上为一维数组的定义方式。数组长度必须是整数且大于 0。

例如以下定义了数组 balance 长度为 10 类型为 float32：

```
var balance [10] float32
```

17.2 初始化数组

以下演示了数组初始化：

```
var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

初始化数组中 {} 中的元素个数不能大于 [] 中的数字。

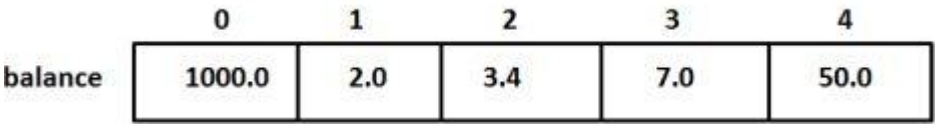
如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小：

```
 var balance = [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

该实例与上面的实例是一样的，虽然没有设置数组的大小。

```
 balance[4] = 50.0
```

以上实例读取了第五个元素。数组元素可以通过索引（位置）来读取（或者修改），索引从 0 开始，第一个元素索引为 0，第二个索引为 1，以此类推。



17.3 访问数组元素

数组元素可以通过索引（位置）来读取。格式为数组名后加中括号，中括号中为索引的值。例如：

```
float32 salary = balance[9]
```

以上实例读取了数组 balance 第 10 个元素的值。

以下演示了数组完整操作（声明、赋值、访问）的实例：

```
package main

import "fmt"

func main() {
   var n [10]int /* n 是一个长度为 10 的数组 */
   var i,j int
```

```
   /* 为数组 n 初始化元素 */
   for i = 0; i < 10; i++ {
      n[i] = i + 100 /* 设置元素为 i + 100 */
   }

   /* 输出每个数组元素的值 */
   for j = 0; j < 10; j++ {
      fmt.Printf("Element[%d] = %d\n", j, n[j] )
   }
}
```

以上实例执行结果如下：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

17.4 多维数组

数组在 Go 语言中很重要，应该需要了解更多的信息。以下几个与数组相关的重要概念应该向 Go 程序员明确：

概念	描述
多维数组	Go 支持多维数组，多维数组的最简单的形式是二维数组。
将数组传递给函数	可以通过指定数组的名称而不使用索引，将指向数组的指针传递给函数。

Go 编程语言允许多维数组。这里是多维数组声明的一般形式：

```
var variable_name [SIZE1][SIZE2]...[SIZEN] variable_type
```

Go

例如，以下声明创建一个三维 `5 x 10 x 4` 的整数数组：

```
var threedim [5][10][4]int
```

Go

17.4.1 二维数组：

多维数组的最简单的形式是二维数组。 二维数组本质上是一维数组的列表。要声明一个大小为 x，y 的二维整数数组，可以这样写：

```
var arrayName [ x ][ y ] variable_type
```

Go

其中 variable_type 可以是任何有效的 Go 数据类型，arrayName 将是有效的 Go 标识符。 二维数组可以被认为是具有 x 个行和 y 个列的表。包含三行四列的二维数组 a 可以如下所示：

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

因此，数组 a 中的每个元素由形式 a[i][j]的元素名称标识，其中 a 是数组的名称，i 和 j 是唯一标识 a 数组中的每个元素的下标。

多维数组可以通过为每一行指定括号值来初始化。以下是具有 3 行的数组，每行具有 4 列。

```
a = [3][4]int{
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
}
```

17.4.2 访问二维数组元素

通过使用下标，即数组的行索引和列索引来访问 2 维数组中的元素。 例如：

```
int val = a[2][3]
```

上面的语句将获取数组中第 3 行的第 4 个元素。可以在上图中验证它。如下面的程序，使用嵌套循环来处理一个二维数组：

```
package main


import "fmt"


func main() {
   /* an array with 5 rows and 2 columns*/
```

```
   var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}
   var i, j int


   /* output each array element's value */
   for  i = 0; i < 5; i++ {
      for j = 0; j < 2; j++ {
         fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )
      }
   }
}
```

当上述代码编译和执行时，它产生以下结果：

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

如上所述，我们可以创建任意数量维度的数组，尽管创建的大多数数组可能是一维或二维。

17.5 数组作为函数的参数

如果想传递一个一维数组作为参数到一个函数中，必须声明函数形式参数以下面两种方式的其中一种，两个声明方法都产生类似的结果，因为每个方式都告诉编译器要接收一个整数数组。类似的方式，可以传递多维数组作为形式参数。

方法-1
形式参数作为一个已知大小数组如下：

```
void myFunction(param [10]int)
{
.
.
```

```
.
}
```

17.5.1 方法-2

形式参数作为一个未知道大小的数组如下：

```
void myFunction(param []int)
{
.
.
.
}
```

示例：
现在，考虑下面的函数，它将数组作为参数和另一个指定数组大小的参数，并基于传递的参数，计算数组传递的数组中每个元素的平均值返回，如下：

```
func getAverage(arr []int, int size) float32
{
   var i int
   var avg, sum float32

   for i = 0; i < size; ++i {
      sum += arr[i]
   }

   avg = sum / size

   return avg;
}
```

```
Go
```

现在，调用上面的函数如下：

```
package main
```

```
import "fmt"

func main() {
   /* an int array with 5 elements */
   var  balance = []int {1000, 2, 3, 17, 50}
   var avg float32

   /* pass array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value */
   fmt.Printf( "Average value is: %f ", avg );
}
func getAverage(arr []int, size int) float32 {
   var i,sum int
   var avg float32

   for i = 0; i < size;i++ {
      sum += arr[i]
   }

   avg = float32(sum / size)

   return avg;
}
```

当上面的代码编译并执行时，它产生以下结果：

```
Average value is: 214.400000
```

正如上面所看到的，数组的长度与函数无关，因为 Go 对形式参数不执行边界检查。

18 Go 语言指针

Go 语言中指针是很容易学习的，Go 语言中使用指针可以更简单的执行一些任务。

接下来让我们来一步步学习 Go 语言指针。

我们都知道，变量是一种使用方便的占位符，用于引用计算机内存地址。

Go 语言的取地址符是 &，放到一个变量前使用就会返回相应变量的内存地址。

以下实例演示了变量在内存中地址：

```
package main

import "fmt"

func main() {
   var a int = 10

   fmt.Printf("变量的地址: %x\n", &a  )
}
```

执行以上代码输出结果为：

```
变量的地址: 20818a220
```

现在我们已经了解了什么是内存地址和如何去访问它。接下来我们将具体介绍指针。

18.1 什么是指针

指针是一个变量，其值是另一个变量的地址，即存储器位置的直接地址。类似变量或常量一样，必须要先声明一个指针，然后才能使用它来存储任何变量地址。指针变量声明的一般形式是

类似于变量和常量，在使用指针前你需要声明指针。指针声明格式如下：

```
var var_name *var-type
```

var-type 为指针类型

var_name 为指针变量名

*号用于指定变量是作为一个指针

以下是有效的指针声明：

```
var ip *int        /* 指向整型*/
var fp *float32    /* 指向浮点型 */
```

本例中这是一个指向 int 和 float32 的指针。

18.2 如何使用指针

指针使用流程：

- 定义指针变量。
- 为指针变量赋值。
- 访问指针变量中指向地址的值。

在指针类型前面加上 * 号（前缀）来获取指针所指向的内容。

```
package main

import "fmt"

func main() {
   var a int= 20   /* 声明实际变量 */
   var ip *int        /* 声明指针变量 */

   ip = &a  /* 指针变量的存储地址 */

   fmt.Printf("a 变量的地址是: %x\n", &a  )

   /* 指针变量的存储地址 */
   fmt.Printf("ip 变量储存的指针地址: %x\n", ip )

   /* 使用指针访问值 */
   fmt.Printf("*ip 变量的值: %d\n", *ip )
}
```

以上实例执行输出结果为：

```
a 变量的地址是: 20818a220
ip 变量储存的指针地址: 20818a220
*ip 变量的值: 20
```

18.3 Go 空指针

当一个指针被定义后没有分配到任何变量时，它的值为 nil。

nil 指针也称为空指针。

nil 在概念上和其它语言的 null、None、nil、NULL 一样，都指代零值或空值。

一个指针变量通常缩写为 ptr。

查看以下实例：

```
package main

import "fmt"

func main() {
   var  ptr *int

   fmt.Printf("ptr 的值为 : %x\n", ptr  )
}
```

以上实例输出结果为：

```
ptr 的值为 : 0
```

空指针判断：

```
if(ptr != nil)     /* ptr 不是空指针 */
if(ptr == nil)    /* ptr 是空指针 */
```

18.4 Go 指针数组

在理解指针数组的概念之前，看看下面的例子，它使用了一个 3 个整数的数组：

```
package main
import "fmt"


const MAX int = 3


func main() {
   a := []int{10,100,200}
   var i int
   for i = 0; i < MAX; i++ {
      fmt.Printf("Value of a[%d] = %d\n", i, a[i] )
   }
}
```

当上述代码编译和执行时，它产生以下结果：

```
Value of a[0] = 10
Value of a[1] = 100
Value of a[2] = 200
```

可能有一种情况，当想要维护一个数组，它可以存储指向 int 或字符串或任何其他可用的数据类型的指针。下面是一个指向整数的指针数组的声明：

```
var ptr [MAX]*int;
```

这里将 ptr 声明为一个 MAX 整数的指针数组。 因此，ptr 中的每个元素现在保存一个指向 int 值的指针。以下示例使用三个整数，它们将存储在指针数组中，如下所示：

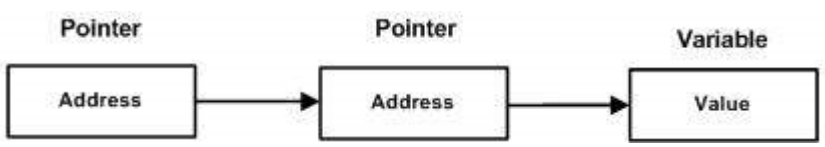
```
package main


import "fmt"
const MAX int = 3
func main() {
   a := []int{10,100,200}
   var i int
   var ptr [MAX]*int;

   for  i = 0; i < MAX; i++ {
      ptr[i] = &a[i] /* assign the address of integer. */
   }

   for  i = 0; i < MAX; i++ {
      fmt.Printf("Value of a[%d] = %d\n", i,*ptr[i] )
   }
}
```

当上述代码编译和执行时，它产生以下结果：

```
Value of a[0] = 10
Value of a[1] = 100
Value of a[2] = 200
```

18.5 指向指针的指针

指向指针的指针是多重间接的形式或指针链。通常，指针包含变量的地址。当定义指向指针的指针时，第一个指针包含第二个指针的地址，它指向包含实际值的位置，如下所示。



作为指向指针的指针的变量必须如此声明。这是通过在其名称前面添加一个星号(*)来实现的。 例如，以下是一个指向 int 类型的指针的声明：

```
var ptr **int;
```

当目标值由指向指针的指针间接指向时，访问该值需要应用两个星号(**)运算符，如下面的示例所示：

```
package main


import "fmt"


func main() {


   var a int
   var ptr *int
   var pptr **int
   a = 3000
   /* take the address of var */
   ptr = &a
   /* take the address of ptr using address of operator & */
   pptr = &ptr
   /* take the value using pptr */
   fmt.Printf("Value of a = %d\n", a )
   fmt.Printf("Value available at *ptr = %d\n", *ptr )
   fmt.Printf("Value available at **pptr = %d\n", **pptr)
}
```

当上述代码编译和执行时，它产生以下结果：

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

18.6 传递指针到函数

Go 编程语言允许传递一个指针到函数中。 为此，只需将函数的参数声明为指针类型。
看看下面一个简单的例子，传递了两个指针给一个函数，并在函数里面改变它们的值，这个值反映在调用函数中：

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100
   var b int= 200

   fmt.Printf("Before swap, value of a : %d\n", a )
   fmt.Printf("Before swap, value of b : %d\n", b )

   /* calling a function to swap the values.
   * &a indicates pointer to a ie. address of variable a and
   * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);

   fmt.Printf("After swap, value of a : %d\n", a )
   fmt.Printf("After swap, value of b : %d\n", b )
}

func swap(x *int, y *int) {
   var temp int
   temp = *x    /* save the value at address x */
   *x = *y      /* put y into x */
   *y = temp    /* put temp into y */
}
```

当上述代码编译和执行时，它产生以下结果：

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
```

```
After swap, value of b :100
```

19 Go 语言结构体

Go 语言中数组可以存储同一类型的数据，但在结构体中我们可以为不同项定义不同的数据类型。

结构体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。

结构体表示一项记录，比如保存图书馆的书籍记录，每本书有以下属性：

- Title ：标题
- Author ： 作者
- Subject：学科
- ID：书籍 ID

19.1 定义结构体

结构体定义需要使用 type 和 struct 语句。struct 语句定义一个新的数据类型，结构体有中有一个或多个成员。type 语句设定了结构体的名称。结构体的格式如下：

```
type struct_variable_type struct {
   member definition;
   member definition;
   ...
   member definition;
}
```

一旦定义了结构体类型，它就能用于变量的声明，语法格式如下：

```
variable_name := structure_variable_type {value1, value2...valuen}
```

19.2 访问结构体成员

如果要访问结构体成员，需要使用点号 (.) 操作符，格式为："结构体.成员名"。

结构体类型变量使用 struct 关键字定义，实例如下：

```
package main

import "fmt"
```

```
type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* 声明 Book1 为 Books 类型 */
   var Book2 Books        /* 声明 Book2 为 Books 类型 */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "尹成 "
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "尹成"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   fmt.Printf( "Book 1 title : %s\n", Book1.title)
   fmt.Printf( "Book 1 author : %s\n", Book1.author)
   fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
   fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)

   /* 打印 Book2 信息 */
   fmt.Printf( "Book 2 title : %s\n", Book2.title)
   fmt.Printf( "Book 2 author : %s\n", Book2.author)
   fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
   fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
}
```

以上实例执行运行结果为：

```
Book 1 title : Go 语言
Book 1 author : 尹成
Book 1 subject : Go 语言教程
Book 1 book_id : 6495407
Book 2 title : Python 教程
Book 2 author : 尹成
```

```
Book 2 subject : Python 语言教程
Book 2 book_id : 6495700
```

19.3 结构体作为函数参数

你可以像其他数据类型一样将结构体类型作为参数传递给函数。并以以上实例的方式访问结构体变量：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* 声明 Book1 为 Books 类型 */
   var Book2 Books        /* 声明 Book2 为 Books 类型 */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "尹成"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "尹成"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   printBook(Book1)

   /* 打印 Book2 信息 */
   printBook(Book2)
}

func printBook( book Books ) {
```

```
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

以上实例执行运行结果为：

```
Book title : Go 语言
Book author : 尹成
Book subject : Go 语言教程
Book book_id : 6495407
Book title : Python 教程
Book author : 尹成
Book subject : Python 语言教程
Book book_id : 6495700
```

19.4 结构体指针

你可以定义指向结构体的指针类似于其他指针变量，格式如下：

```
var struct_pointer *Books
```

以上定义的指针变量可以存储结构体变量的地址。查看结构体变量地址，可以将 & 符号放置于结构体变量前：

```
struct_pointer = &Book1;
```

使用结构体指针访问结构体成员，使用 "." 操作符：

```
struct_pointer.title;
```

接下来让我们使用结构体指针重写以上实例，代码如下：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}
```

```
func main() {
   var Book1 Books        /* Declare Book1 of type Book */
   var Book2 Books        /* Declare Book2 of type Book */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "尹成"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "尹成"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   printBook(&Book1)

   /* 打印 Book2 信息 */
   printBook(&Book2)
}
func printBook( book *Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

以上实例执行运行结果为：

```
Book title : Go 语言
Book author : 尹成
Book subject : Go 语言教程
Book book_id : 6495407
Book title : Python 教程
Book author : 尹成
Book subject : Python 语言教程
Book book_id : 6495700
```

20 Go 语言切片(Slice)

切片是 **Go** 语言中的关键数据类型，为序列提供了比数组更强大的接口。

与数组不同，切片(slice)只是由它们包含的元素(而不是元素的数量)键入。要创建非零长度的空切片，请使用内置 `make()`函数。这里创建一个长度为 `3` 的字符串(初始为零值)。

我们可以像数组一样设置和获取字符串的子串值。`len()`函数返回切片的长度。除了这些基本操作之外，切片还支持更多，使它们比数组更丰富。一个是内置 `append()`函数，它返回包含一个或多个新值的切片。注意，需要接收 `append()`函数的返回值，因为可能得到一个新的 `slice` 值。

也可以复制切片。这里创建一个与切片 `s` 相同长度的空切片 `c`，并从切片 `s` 复制到 `c` 中。切片支持具有语法为 `slice[low:high]`的切片运算符。 例如，这获得元素 `s[2]`，`s[3]`和 `s[4]`的切片。

这切片到(但不包括)`s[5]`。这切片从(包括)`s[2]`。可以在一行中声明并初始化 `slice` 的变量。

切片可以组成多维数据结构。内切片的长度可以变化，与多维数组不同。

20.1 定义切片

你可以声明一个未指定大小的数组来定义切片：

```
var identifier []type
```

切片不需要说明长度。

或使用 make()函数来创建切片:

```
var slice1 []type = make([]type, len)

也可以简写为

slice1 := make([]type, len)
```

也可以指定容量，其中 capacity 为可选参数。

```
make([]T, length, capacity)
```

这里 len 是数组的长度并且也是切片的初始长度。

20.1.1 切片初始化

```
s :=[] int {1,2,3 }
```

直接初始化切片，[]表示是切片类型，{1,2,3}初始化值依次是 1,2,3.其 cap=len=3

```
s := arr[:]
```

初始化切片 s,是数组 arr 的引用

```
s := arr[startIndex:endIndex]
```

将 arr 中从下标 startIndex 到 endIndex-1 下的元素创建为一个新的切片

```
s := arr[startIndex:]
```

缺省 endIndex 时将表示一直到 arr 的最后一个元素

```
s := arr[:endIndex]
```

缺省 startIndex 时将表示从 arr 的第一个元素开始

```
s1 := s[startIndex:endIndex]
```

通过切片 s 初始化切片 s1

```
s :=make([]int,len,cap)
```

通过内置函数 make()初始化切片 s,[]int 标识为其元素类型为 int 的切片

20.2 len() 和 cap() 函数

切片是可索引的，并且可以由 len() 方法获取长度。

切片提供了计算容量的方法 cap() 可以测量切片最长可以达到多少。

以下为具体实例：

```
package main

import "fmt"

func main() {
   var numbers = make([]int,3,5)

   printSlice(numbers)
}

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

以上实例运行输出结果为:

```
len=3 cap=5 slice=[0 0 0]
```

20.3 空(nil)切片

一个切片在未初始化之前默认为 nil，长度为 0，实例如下：

```
package main

import "fmt"

func main() {
   var numbers []int

   printSlice(numbers)

   if(numbers == nil){
      fmt.Printf("切片是空的")
   }
}

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

以上实例运行输出结果为:

```
len=0 cap=0 slice=[]
切片是空的
```

20.4 切片截取

可以通过设置下限及上限来设置截取切片 *[lower-bound:upper-bound]*，实例如下：

```
package main

import "fmt"

func main() {
   /* 创建切片 */
   numbers := []int{0,1,2,3,4,5,6,7,8}
   printSlice(numbers)
```

```
   /* 打印原始切片 */
   fmt.Println("numbers ==", numbers)

   /* 打印子切片从索引 1(包含) 到索引 4(不包含)*/
   fmt.Println("numbers[1:4] ==", numbers[1:4])

   /* 默认下限为 0*/
   fmt.Println("numbers[:3] ==", numbers[:3])

   /* 默认上限为 len(s)*/
   fmt.Println("numbers[4:] ==", numbers[4:])

   numbers1 := make([]int,0,5)
   printSlice(numbers1)

   /* 打印子切片从索引  0(包含) 到索引 2(不包含) */
   number2 := numbers[:2]
   printSlice(number2)

   /* 打印子切片从索引 2(包含) 到索引 5(不包含) */
   number3 := numbers[2:5]
   printSlice(number3)

}

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

执行以上代码输出结果为：

```
len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len=0 cap=5 slice=[]
len=2 cap=9 slice=[0 1]
len=3 cap=7 slice=[2 3 4]
```

20.5 append() 和 copy() 函数

如果想增加切片的容量，我们必须创建一个新的更大的切片并把原分片的内容都拷贝过来。

下面的代码描述了从拷贝切片的 copy 方法和向切片追加新元素的 append 方法。

```
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   /* 允许追加空切片 */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* 向切片添加一个元素 */
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* 同时添加多个元素 */
   numbers = append(numbers, 2,3,4)
   printSlice(numbers)

   /* 创建切片 numbers1 是之前切片的两倍容量*/
   numbers1 := make([]int, len(numbers), (cap(numbers))*2)

   /* 拷贝 numbers 的内容到 numbers1 */
   copy(numbers1,numbers)
   printSlice(numbers1)
}

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

以上代码执行输出结果为：

```
len=0 cap=0 slice=[]
len=1 cap=1 slice=[0]
len=2 cap=2 slice=[0 1]
len=5 cap=6 slice=[0 1 2 3 4]
```

```
len=5 cap=12 slice=[0 1 2 3 4]
```

20.6 多维切片

```
package main

import "fmt"

func main() {
twoD := make([][]int, 3)
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

执行上面代码，将得到以下输出结果

```
2d:  [[0] [1 2] [2 3 4]]
```

21 Go 语言范围(Range)

Go 语言中 range 关键字用于 for 循环中迭代数组(array)、切片(slice)、通道(channel)或集合(map)的元素。在数组和切片中它返回元素的索引值，在集合中返回 key-value 对的 key 值。

范围表达式	第 1 个值	第 2 个值(可选)
数组或切片 a[n]E	索引 i 整数	a[i]E
Strubg s 字符串	索引 i 整数	符文整数
map m map[K]V	key k K	value m[k] V
channel c chan E	element e E	none

21.1.1 实例

```
package main
import "fmt"
func main() {
    //这是我们使用 range 去求一个 slice 的和。使用数组跟这个很类似
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
    //在数组上使用 range 将传入 index 和值两个变量。上面那个例子我们不需要使用该元素的序号，所以我们使用空白符"_"省略了。有时侯我们确实需要知道它的索引。
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    //range 也可以用在 map 的键值对上。
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    //range 可以只迭代 map 的 key 值
        for k := range kvs {
        fmt.Println("key:", k)
    }

    //range 也可以用来枚举 Unicode 字符串。第一个参数是字符的索引，第二个是字符（Unicode 的值）本身。
    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

以上实例运行输出结果为：

```
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
```

```
0 103
1 111
```

22 Go 语言 Map(集合)

Map 是一种无序的键值对的集合。Map 最重要的一点是通过 key 来快速检索数据，key 类似于索引，指向数据的值。

Map 是一种集合，所以我们可以像迭代数组和切片那样迭代它。不过，Map 是无序的，我们无法决定它的返回顺序，这是因为 Map 是使用 hash 表来实现的。

22.1 定义 Map

可以使用内建函数 make 也可以使用 map 关键字来定义 Map:

```
/* 声明变量，默认 map 是 nil */
var map_variable map[key_data_type]value_data_type

/* 使用 make 函数 */
map_variable := make(map[key_data_type]value_data_type)
```

如果不初始化 map，那么就会创建一个 nil map。nil map 不能用来存放键值对

22.2 实例

下面实例演示了创建和使用 map:

```
package main

import "fmt"

func main() {
   var countryCapitalMap map[string]string
   /* 创建集合 */
   countryCapitalMap = make(map[string]string)

   /* map 插入 key-value 对，各个国家对应的首都 */
   countryCapitalMap["France"] = "Paris"
   countryCapitalMap["Italy"] = "Rome"
   countryCapitalMap["Japan"] = "Tokyo"
   countryCapitalMap["India"] = "New Delhi"

   /* 使用 key 输出 map 值 */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }
```

```
   /* 查看元素在集合中是否存在 */
   captial, ok := countryCapitalMap["United States"]
   /* 如果 ok 是 true, 则存在, 否则不存在 */
   if(ok){
      fmt.Println("Capital of United States is", captial)
   }else {
      fmt.Println("Capital of United States is not present")
   }
}
```

以上实例运行结果为：

```
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Capital of United States is not present
```

22.3 delete() 函数

delete() 函数用于删除集合的元素，参数为 map 和其对应的 key。实例如下：

```
package main

import "fmt"

func main() {
   /* 创建 map */
   countryCapitalMap := map[string] string {"France":"Paris","Italy":"Rome","Japan":
"Tokyo","India":"New Delhi"}

   fmt.Println("原始 map")

   /* 打印 map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }

   /* 删除元素 */
   delete(countryCapitalMap,"France");
   fmt.Println("Entry for France is deleted")

   fmt.Println("删除元素后 map")
```

```
   /* 打印 map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }
}
```

以上实例运行结果为：

```
原始 map
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Entry for France is deleted
删除元素后 map
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
```

23 函数高级

23.1 集合函数实例

我们经常需要在程序中对数据集合执行操作，例如选择满足给定谓词的所有项目，或将所有项目映射到具有自定义函数的新集合。

在某些语言中，通用数据结构和算法是惯用的。 Go 不支持泛型；在 Go 中，如果并且当它们对于程序和数据类型特别需要时，提供集合函数是很常见的。

这里是一些字符串切片的示例收集函数。可以使用这些示例来构建您自己的函数。注意，在某些情况下，直接内联集合操作代码可能是最清楚的，而不用创建和调用辅助函数。

```
package main


import "strings"

import "fmt"


// 返回目标字符串 t 的第一个索引，如果未找到匹配，则返回-1。
func Index(vs []string, t string) int {
    for i, v := range vs {
        if v == t {
            return i
```

```
        }
    }
    return -1
}

// 如果目标字符串 t 在切片中，则返回 true。
func Include(vs []string, t string) bool {
    return Index(vs, t) >= 0
}

// 返回包含切片中满足谓词 f 的所有字符串的新切片。.
func Any(vs []string, f func(string) bool) bool {
    for _, v := range vs {
        if f(v) {
            return true
        }
    }
    return false
}

// 返回 true，如果所有的字符串在切片中。
func All(vs []string, f func(string) bool) bool {
    for _, v := range vs {
        if !f(v) {
            return false
        }
    }
    return true
}

// 返回一个新的切片，包含将函数 f 应用于原始切片中的每个字符串的结果。.
func Filter(vs []string, f func(string) bool) []string {
```

```
    vsf := make([]string, 0)
    for _, v := range vs {
        if f(v) {
            vsf = append(vsf, v)
        }
    }
    return vsf
}

// Returns a new slice containing the results of applying
// the function `f` to each string in the original slice.
func Map(vs []string, f func(string) string) []string {
    vsm := make([]string, len(vs))
    for i, v := range vs {
        vsm[i] = f(v)
    }
    return vsm
}

func main() {

    // Here we try out our various collection functions.
    var strs = []string{"peach", "apple", "pear", "plum"}

    fmt.Println(Index(strs, "pear"))

    fmt.Println(Include(strs, "grape"))

    fmt.Println(Any(strs, func(v string) bool {
        return strings.HasPrefix(v, "p")
    }))
```

```
    fmt.Println(All(strs, func(v string) bool {
        return strings.HasPrefix(v, "p")
    }))

    fmt.Println(Filter(strs, func(v string) bool {
        return strings.Contains(v, "e")
    }))

    // 上面的例子都是自定义函数，你也可以用系统函数。
    fmt.Println(Map(strs, strings.ToUpper))

}
```

Go

执行上面代码，将得到以下输出结果 -

```
2
false
true
false
[peach apple pear]
[PEACH APPLE PEAR PLUM]
```

23.2 字符串函数

标准库的字符串包提供了许多有用的字符串相关函数。这里有一些例子就像使用一个包的感觉。

```
package main


import s "strings"
import "fmt"


// 将 fmt.Println 别名缩写为一个较短的名称，因为将在下面示例代码中使用它。
var p = fmt.Println
```

```
func main() {


    // 下面是字符串中可用函数的示例。因为这些是来自包的函数，而不是字符串对象本身的方法，所以我们需要将有问题的字符串作为函数的第一个参数传递。可以在字符串包 docs 中找到更多的函数。
    p("Contains:  ", s.Contains("test", "es"))//是否包含
    p("Count:     ", s.Count("test", "t"))//包含字符串的数量
    p("HasPrefix: ", s.HasPrefix("test", "te")) //是否在前面
    p("HasSuffix: ", s.HasSuffix("test", "st"))//是否在后面
    p("Index:     ", s.Index("test", "e"))  //返回位置
    p("Join:      ", s.Join([]string{"a", "b"}, "-")) //连接字符
    p("Repeat:    ", s.Repeat("a", 5))//重复
    p("Replace:   ", s.Replace("foo", "o", "0", -1))//替换
    p("Replace:   ", s.Replace("foo", "o", "0", 1))
    p("Split:     ", s.Split("a-b-c-d-e", "-")) //分割
    p("ToLower:   ", s.ToLower("TEST"))//转换为小写
    p("ToUpper:   ", s.ToUpper("test"))//转换为大写
    p()


    // 不是字符串的一部分，但值得一提的是，以字节为单位获取字符串长度并按索引获取字节的机制。
    p("Len: ", len("hello"))//支付串长度
    p("Char:", "hello"[1])//打印第一个字符
}


//注意 len 和索引在字节级上工作的。 Go 使用 UTF-8 编码字符串，因此这通常是有用的。如果使用潜在的多字节字符，将需要使用编码转换操作。Go
```

执行上面代码，将得到以下输出结果 -

```
F:\worksp\golang>go run string-functions.go
Contains:  true
Count:     2
HasPrefix: true
HasSuffix: true
```

```
Index:      1
Join:       a-b
Repeat:     aaaaa
Replace:    f00
Replace:    f0o
Split:      [a b c d e]
ToLower:    test
ToUpper:    TEST


Len:  5
Char: 101
```

23.3 Go 内置排序函数

Go 语言的 `sort` 包实现了内置和用户定义类型的排序。我们先看看内置的排序。

```
package main


import "fmt"
import "sort"


func main() {
    //排序方法特定于内置类型; 这里是一个字符串的例子。 请注意，排序是内部排序，因此它会更改给定的切片，并且不返回新的切片。
    strs := []string{"c", "a", "b"}
    sort.Strings(strs)
    fmt.Println("Strings:", strs)


    // 这里例举一个排序 int 类型数值的一个例子。.
    ints := []int{7, 2, 4}
    sort.Ints(ints)
    fmt.Println("Ints:   ", ints)
```

```
    //也可以使用 sort 来检查切片是否已经按照排序顺序。
    s := sort.IntsAreSorted(ints)
    fmt.Println("Sorted: ", s)
}
```

执行上面代码，将得到以下输出结果 -

```
Strings: [a b c]
Ints:    [2 4 7]
Sorted:  true
```

23.4 自定义排序

有时候，我们希望通过除了自然顺序以外的其他方式对集合进行排序。例如，假设我们想按字符串的长度而不是字母顺序对字符串进行排序。下面是 Go 语言中自定义排序的示例。

```
package main


import "sort"
import "fmt"


// 为了使用 Go 语言中的自定义函数进行排序，我们需要一个相应的类型。这里创建了一个 ByLength 类型，它只是内置 []string 类型的别名。
type ByLength []string


// 需要实现 sort.Interface - Len，Less 和 Swap - 在这个类型上，所以可以使用 sort 包中的一般 Sort 函数。Len 和 Swap 通常在类型之间是相似的，Less 保存实际的自定义排序逻辑。在这个例子中，要按照字符串长度的增加顺序排序，因此在这里使用 len(s [i])和 len(s [j])。
func (s ByLength) Len() int {
    return len(s)
}
func (s ByLength) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}
func (s ByLength) Less(i, j int) bool {
    return len(s[i]) < len(s[j])
```

```
}


// 所有这些都到位后，现在可以通过将原始 fruits 切片转换为 ByLength 来实现自定义排序，然后对该类型切片使用 sort.Sort()方法。
func main() {
    fruits := []string{"peach", "banana", "kiwi"}
    sort.Sort(ByLength(fruits))
    fmt.Println(fruits)
}
```

执行上面代码，将得到以下输出结果 -

```
[kiwi peach banana]
```

23.5 Go 语言递归函数

递归，就是在运行的过程中调用自己。

语法格式如下：

```
func recursion() {
   recursion() /* 函数调用自身 */
}

func main() {
   recursion()
}
```

Go 语言支持递归。但我们在使用递归时，开发者需要设置退出条件，否则递归将陷入无限循环中。

递归函数对于解决数学上的问题是非常有用的，就像计算阶乘，生成斐波那契数列等。

23.5.1 阶乘

以下实例通过 Go 语言的递归函数实例阶乘：

```
package main

import "fmt"

func Factorial(n uint64)(result uint64) {
    if (n > 0) {
        result = n * Factorial(n-1)
```

```
        return result
    }
    return 1
}

func main() {
    var i int = 15
    fmt.Printf("%d 的阶乘是 %d\n", i, Factorial(uint64(i)))
}
```

以上实例执行输出结果为：

```
15 的阶乘是 1307674368000
```

23.5.2 斐波那契数列

以下实例通过 Go 语言的递归函数实现斐波那契数列：

```
package main

import "fmt"

func fibonacci(n int) int {
  if n < 2 {
   return n
  }
  return fibonacci(n-2) + fibonacci(n-1)
}

func main() {
    var i int
    for i = 0; i < 10; i++ {
       fmt.Printf("%d\t", fibonacci(i))
    }
}
```

以上实例执行输出结果为：

```
0    1    1    2    3    5    8    13    21    34
```

24 Go 语言接口

Go 语言提供了另外一种数据类型即接口，它把所有的具有共性的方法定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。

```
/* 定义接口 */
type interface_name interface {
   method_name1 [return_type]
   method_name2 [return_type]
   method_name3 [return_type]
   ...
   method_namen [return_type]
}

/* 定义结构体 */
type struct_name struct {
   /* variables */
}

/* 实现接口方法 */
func (struct_name_variable struct_name) method_name1() [return_type] {
   /* 方法实现 */
}
...
func (struct_name_variable struct_name) method_namen() [return_type] {
   /* 方法实现*/
}
```

24.1 接口示例 1

```
package main

import (
    "fmt"
)

type Phone interface {
    call()
}

type NokiaPhone struct {
}

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

```
}

type IPhone struct {
}

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

func main() {
    var phone Phone

    phone = new(NokiaPhone)
    phone.call()

    phone = new(IPhone)
    phone.call()

}
```

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

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

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

24.2 接口示例 2

```
package main


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

```
/* define an interface */
type Shape interface {
   area() float64
}

/* define a circle */
type Circle struct {
   x,y,radius float64
}

/* define a rectangle */
type Rectangle struct {
   width, height float64
}

/* define a method for circle (implementation of Shape.area())*/
func(circle Circle) area() float64 {
   return math.Pi * circle.radius * circle.radius
}

/* define a method for rectangle (implementation of Shape.area())*/
func(rect Rectangle) area() float64 {
   return rect.width * rect.height
}

/* define a method for shape */
func getArea(shape Shape) float64 {
   return shape.area()
}

func main() {
```

```
    circle := Circle{x:0,y:0,radius:5}
    rectangle := Rectangle {width:10, height:5}

    fmt.Printf("Circle area: %f\n",getArea(circle))
    fmt.Printf("Rectangle area: %f\n",getArea(rectangle))
}
```

当上述代码编译和执行时，它产生以下结果：

```
Circle area: 78.539816
Rectangle area: 50.000000
```

25 Go 通道

25.1 Go 语言协程 goroutine

```
package main

import "fmt"

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {
 //假设有一个函数调用 f(s)。 下面是以通常的方式调用它，同步运行它。
    f("direct")
//要为函数新开一个线程，请使用 go f(s)。 这个新的 goroutine 将与调用同时执行。
    go f("goroutine")
  // 也可以为匿名函数调用启动
    go func(msg string) {
        fmt.Println(msg)
    }("going")

    // 两个函数调用现在在不同的 goroutine 中异步运行，所以执行到这里。这个 ScanIn 代码要求我们在程序退出之前按一个键
    var input string
    fmt.Scanln(&input)
    fmt.Println("done")
}
```

25.2 通道简单示例

25.3 关于 goroutine 协程

协程的概念很早就提出来了，但直到最近几年才在某些语言（如 Lua）中得到广泛应用。

子程序，或者称为函数，在所有语言中都是层级调用，比如 A 调用 B，B 在执行过程中又调用了 C，C 执行完毕返回，B 执行完毕返回，最后是 A 执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似 CPU 的中断。比如子程序 A、B：

```
func A() {
    fmt.Println('1')
    fmt.Println('2')
    fmt.Println('3')
}
func B() {
   fmt.Println('x')
   fmt.Println('y')
  fmt.Println('z')
}
```

假设由协程执行，在执行 A 的过程中，可以随时中断，去执行 B，B 也可能在执行过程中中断再去执行 A，结果可能是：

```
1
2
x
```

```
y

3

z
```

但是在 A 中是没有调用 B 的，所以协程的调用比函数调用理解起来要难一些。

看起来 A、B 的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核 CPU 呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

25.4 通道简单示例

通道是连接并发 `goroutine` 的管道。可以从一个 `goroutine` 向通道发送值，并在另一个 `goroutine` 中接收到这些值。

```
package main


import "fmt"


func main() {


    // 使用 make(chan val-type)创建一个新通道，通道由输入的值传入。使用通道 <- 语法将值发送到通道。
    messages := make(chan string)


    // 这里从一个新的 goroutine 发送“ping”到在上面的消息通道。<-channel 语法从通道接收值。
    go func() { messages <- "ping" }()
```

```
    // <-channel 语法从通道接收值。在这里，将收到上面发送的“ping”消息并打印出来。
    msg := <-messages
    fmt.Println(msg)
}
```

执行上面代码，将得到以下输出结果 -

```
ping
```

25.5 通道缓冲示例

默认情况下，通道是未缓冲的，意味着如果有相应的接收(`<- chan`)准备好接收发送的值，它们将只接受发送(`chan <-`)。 缓冲通道接受有限数量的值，而没有用于这些值的相应接收器。

```
package main


import "fmt"


func main() {


    // 这里使一个字符串的通道缓冲多达 2 个值。因为这个通道被缓冲，所以可以将这些值发送到通道中，而没有相应的并发接收。
    messages := make(chan string, 2)


    // 因为这个通道被缓冲，所以可以将这些值发送到通道中，而没有相应的并发接收。
    messages <- "buffered"
    messages <- "channel"


    // 之后可以照常接收这两个值。
    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

Go

执行上面代码，将得到以下输出结果 -

```
buffered
channel
```

25.6 Go 通道同步实例

我们可以使用通道在 `goroutine` 上同步执行程序。这里有一个使用阻塞接收等待 `goroutine` 完成的示例。

```
package main


import "fmt"

import "time"


// 这是将在 goroutine 中运行的函数。 done 通道将用来通知另一个 goroutine 这个函数的工作已经完成，发送值以通知已经完成。
func worker(done chan bool) {

    fmt.Print("working...")

    time.Sleep(time.Second)

    fmt.Println("done")


    // Send a value to notify that we're done.

    done <- true

}


func main() {


    // 启动一个 goroutine 工作程序，给它一个通知通道。如果从此程序中删除 <-done 行，、、//程序将在工作程序(worker)启动之前退出。

    done := make(chan bool, 1)

    go worker(done)


    // 锁住，直到在通道上收到工作程序的通知。

    <-done

}
```

```
working...done
```

25.7 通道线路示例

当使用通道作为函数参数时，可以指定通道是否仅用于发送或接收值。这种特殊性增加了程序的类型安全性。

```
package main


import "fmt"


// 此 ping 功能只接受用于发送值的通道。尝试在此频道上接收将是一个编译时错误。func
ping(pings chan<- string, msg string) {
    pings <- msg
}


// pong 函数一个一个参数用于接受，一个参数用于发送
func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}


func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

Go

执行上面代码，将得到以下输出结果 -

```
passed message
```

25.8 Go 通道+select 示例

select 随机执行一个可运行的 case。如果没有 case 可运行，它将阻塞，直到有 case 可运行。一个默认的子句应该总是可运行的。

Go 编程语言中 select 语句的语法如下：

```
select {
    case communication clause  :
       statement(s);
    case communication clause  :
       statement(s);
    /* 你可以定义任意数量的 case */
    default : /* 可选 */
       statement(s);
}
```

以下描述了 select 语句的语法：

- 每个 case 都必须是一个通信
- 所有 channel 表达式都会被求值
- 所有被发送的表达式都会被求值
- 如果任意某个通信可以进行，它就执行；其他被忽略。
- 如果有多个 case 都可以运行，Select 会随机公平地选出一个执行。其他不会执行。
  否则：如果有 default 子句，则执行该语句。如果没有 default 字句，select 将阻塞，直到某个通信可以运行；Go 不会重新对 channel 或值进行求值。

25.8.1 实例 1：

```
package main

import "fmt"

func main() {
   var c1, c2, c3 chan int
   var i1, i2 int
   select {
      case i1 = <-c1:
         fmt.Printf("received ", i1, " from c1\n")
      case c2 <- i2:
         fmt.Printf("sent ", i2, " to c2\n")
```

```
        case i3, ok := (<-c3):  // same as: i3, ok := <-c3
            if ok {
                fmt.Printf("received ", i3, " from c3\n")
            } else {
                fmt.Printf("c3 is closed\n")
            }
        default:
            fmt.Printf("no communication\n")
    }
}
```

以上代码执行结果为：

```
no communication
```

25.8.2 示例 2：同时等待两个通道

Go 语言的选择(select)可等待多个通道操作。将 goroutine 和 channel 与 select 结合是 Go 语言的一个强大功能。

```
package main


import "time"

import "fmt"


func main() {


    // 对于这个示例，将选择两个通道。.

    c1 := make(chan string)

    c2 := make(chan string)


    // 每个通道将在一段时间后开始接收值，以模拟阻塞在并发 goroutines 中执行的 RPC 操作。我们将使用 select 同时等待这两个值，在每个值到达时打印它们。

    go func() {

        time.Sleep(time.Second * 1)

        c1 <- "one"

    }()
```

```
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "two"
    }()

    // 执行实例程序得到的值是“one”，然后是“two”。注意，总执行时间只有 1～2 秒，因为 1-2 秒 Sleeps 同时执行。
    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

执行上面代码，将得到以下输出结果 -

```
received one
received two
```

25.9 Go 超时（timeouts）实例

超时对于连接到外部资源或在不需要绑定执行时间的程序很重要。在 Go 编程中由于使用了通道和选择(select),实现超时是容易和优雅的。

```
package main

import "time"
import "fmt"

func main() {

    // 在这个示例中，假设正在执行一个外部调用，2 秒后在通道 c1 上返回其结果。
    c1 := make(chan string, 1)
```

```
    go func() {
        time.Sleep(time.Second * 2)
        c1 <- "result 1"
    }()

    // 这里是 select 实现超时。 res: = <-c1 等待结果和<-Time。等待在超时 1 秒后发送一个值。 由于选择继续准备好第一个接收，如果操作超过允许的 1 秒，则将按超时情况处理。
    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout 1")
    }

    // 如果允许更长的超时，如：3s，那么从 c2 的接收将成功，这里将会打印结果。
    c2 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "result 2"
    }()
    select {
    case res := <-c2:
        fmt.Println(res)
    case <-time.After(time.Second * 3):
        fmt.Println("timeout 2")
    }
}
```

执行上面代码，将得到以下输出结果 -

```
timeout 1
result 2
```

25.10 Go 非阻塞通道操作实例

通道的基本发送和接收都阻塞。但是，可以使用 `select` 和 `default` 子句来实现非阻塞发送，接收，甚至非阻塞多路选择(`select`)。

```
package main


import "fmt"

func main() {

    messages := make(chan string)

    signals := make(chan bool)


    // 这里是一个非阻塞接收。如果消息上有可用的值，则会等待。如果不是，它会立即采用默认情况。

    select {

    case msg := <-messages:

        fmt.Println("received message", msg)

    default:

        fmt.Println("no message received")

    }


    // 非阻塞发送工作类似.

    msg := "hi"

    select {

    case messages <- msg:

        fmt.Println("sent message", msg)

    default:

        fmt.Println("no message sent")

    }


    // 可以使用多个上面的默认子句来实现多路非阻塞选择(select)。这里尝试对消息(message)和信号(signals)的非阻塞接收。

    select {

    case msg := <-messages:
```

```
        fmt.Println("received message", msg)
    case sig := <-signals:
        fmt.Println("received signal", sig)
    default:
        fmt.Println("no activity")
    }
}
```

Go

执行上面代码，将得到以下输出结果 -

```
no message received
no message sent
no activity
```

25.11 关闭通道示例

```
ackage main


import "fmt"


// 在这个例子中，我们将使用一个作业通道来完成从 main()goroutine 到 worker
goroutine 的工作。当没有更多的工作时，则将关闭工作通道。
func main() {
    jobs := make(chan int, 5)
    done := make(chan bool)



    // 这里是工作程序 goroutine。 它反复从 j 的工作接收 more := <-jobs。在
 //这种特殊的 2 值形式的接收中，如果作业已关闭并且已经接收到通道中的所有值，则
more 的值将为 false。
 //当已经完成了所有的工作时，使用这个通知。
    go func() {
        for {
            j, more := <-jobs
            if more {
```

```
                fmt.Println("received job", j)
            } else {
                fmt.Println("received all jobs")
                done <- true
                return
            }
        }
    }()

    // 这会通过作业通道向工作线程发送 3 个作业，然后关闭它。
    for j := 1; j <= 3; j++ {
        jobs <- j
        fmt.Println("sent job", j)
    }
    close(jobs)
    fmt.Println("sent all jobs")

    //等待工作程序，可使用前面看到的同步方法。
    <-done
}
```

执行上面代码，将得到以下输出结果 -

```
sent job 1
sent job 2
sent job 3
sent all jobs
received job 1
received job 2
received job 3
received all jobs
```

26 Go 工作池示例

在这个例子中，我们将看看如何使用 `goroutines` 和 `channel` 实现一个工作池。该例子模拟了 3 个人并发的做五份工作的例子。

```
package main


import "fmt"

import "time"


// 这里是工作程序(worker)，我们将运行几个并发实例。这些工作程序(worker)将在工作渠道上接收工作，并将结果发送相应的结果。将每个工作程序(worker)睡一秒钟，用来模拟执行的任务。
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "started  job", j)
        time.Sleep(time.Second)
        fmt.Println("worker", id, "finished job", j)
        results <- j * 2
    }
}


func main() {


    // 为了使用工作程序(worker)池，需要向它们发送工作(或作业)并收集的结果。为此做 2 个通道。
    jobs := make(chan int, 100)
    results := make(chan int, 100)


    // 这启动了 3 个工作程序(worker)，最初被阻止，因为还没有作业。
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }


    // Here we send 5 `jobs` and then `close` that
    // channel to indicate that's all the work we have.
    for j := 1; j <= 5; j++ {
        jobs <- j
```

```
    }
    close(jobs)

    // 然后，我们再发送 5 个作业，然后关闭该通道，以指示这是所有的工作。
    //5 个程序完成后才会退出
    for a := 1; a <= 5; a++ {
        <-results
    }
}
```

执行上面代码，将得到以下输出结果 -

```
worker 3 started  job 2
worker 1 started  job 1
worker 2 started  job 3
worker 3 finished job 2
worker 3 started  job 4
worker 1 finished job 1
worker 1 started  job 5
worker 2 finished job 3
worker 3 finished job 4
worker 1 finished job 5
```

27 Go 错误处理

在 Go 语言编程中，它习惯于通过一个显式的，单独的返回值来传达错误。这与 Java 和 Ruby 等语言中使用的异常，以及有时在 C 语言中使用的重载的单个结果/错误值形成对比。Go 语言编程的方法使得很容易看到哪些函数返回错误，并使用用于任何其他语言的相同语言构造来处理它们，非错误任务。

按照惯例，错误是最后一个返回值，并有类型：error，它是一个内置接口。通过对它们实现 Error()方法，可以使用自定义类型作为错误。上面的例子使用自定义类型来显式地表示一个参数错误。

Go 语言通过内置的错误接口提供了非常简单的错误处理机制。

error 类型是一个接口类型，这是它的定义：

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

我们可以在编码中通过实现 error 接口类型来生成错误信息。

函数通常在最后的返回值中返回错误信息。使用 errors.New 可返回一个错误信息：

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

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

```
result, err:= Sqrt(-1)

if err != nil {
   fmt.Println(err)
}
```

27.1 示例 1：实现 error 接口

```
package main

import (
    "fmt"
)

// 定义一个 DivideError 结构
type DivideError struct {
    dividee int
    divider int
}

// 实现 `error` 接口
func (de *DivideError) Error() string {
    strFormat := `
    Cannot proceed, the divider is zero.
    dividee: %d
    divider: 0
`
    return fmt.Sprintf(strFormat, de.dividee)
}
```

```
// 定义 `int` 类型除法运算的函数
func Divide(varDividee int, varDivider int) (result int, errorMsg string) {
    if varDivider == 0 {
        dData := DivideError{
            dividee: varDividee,
            divider: varDivider,
        }
        errorMsg = dData.Error()
        return
    } else {
        return varDividee / varDivider, ""
    }

}

func main() {

    // 正常情况
    if result, errorMsg := Divide(100, 10); errorMsg == "" {
        fmt.Println("100/10 = ", result)
    }
    // 当被除数为零的时候会返回错误信息
    if _, errorMsg := Divide(100, 0); errorMsg != "" {
        fmt.Println("errorMsg is: ", errorMsg)
    }

}
```

执行以上程序，输出结果为：

```
100/10 =  10
errorMsg is:
    Cannot proceed, the divider is zero.
    dividee: 100
    divider: 0
```

27.2 示例 2

```
    package main

    import "errors"
    import "fmt"

    //按照惯例，错误是最后一个返回值，并有类型：error，它是一个内置接口。
```

```
    //通过对它们实现 Error()方法，可以使用自定义类型作为错误。上面的例子使用自定义类型来显式地表示一个参数错误。
    func f1(arg int) (int, error) {
        if arg == 42 {

       // errors.New 使用给定的错误消息构造基本错误值。
            return -1, errors.New("can't work with 42")

        }

     //错误位置中的 nil 值表示没有错误。
        return arg + 3, nil
    }

    //自定义错误结构体
    type argError struct {
        arg  int
        prob string
    }
    //实现 Error()接口
    func (e *argError) Error() string {
        return fmt.Sprintf("%d - %s", e.arg, e.prob)
    }

    func f2(arg int) (int, error) {
        if arg == 42 {

            //在这种情况下，使用＆argError 语法构建一个新的结构，为两个字段 arg 和 prob 提供值。
            return -1, &argError{arg, "can't work with it"}
        }
        return arg + 3, nil
    }

    func main() {

        // 下面的两个循环测试每个错误返回函数。注意，使用 if 语句内联错误检查是 Go 代码中的常见作法。
        for _, i := range []int{7, 42} {
            if r, e := f1(i); e != nil {
                fmt.Println("f1 failed:", e)
            } else {
                fmt.Println("f1 worked:", r)
            }
```

```
        }
        for _, i := range []int{7, 42} {
            if r, e := f2(i); e != nil {
                fmt.Println("f2 failed:", e)
            } else {
                fmt.Println("f2 worked:", r)
            }
        }

        //如果要以编程方式使用自定义错误中的数据，则需要通过类型断言将错误作为自定义错误类型的实例
        _, e := f2(42)
        if ae, ok := e.(*argError); ok {
            fmt.Println(ae.arg)
            fmt.Println(ae.prob)
        }
    }
```

28 计时器，定时器与速率

28.1 计时器

```
package main


import "time"
import "fmt"


func main() {


    // 定时器代表未来的一个事件。可告诉定时器您想要等待多长时间，它提供了一个通道，在当将通知时执行对应程序。在这个示例中，计时器将等待 2 秒钟。
    timer1 := time.NewTimer(time.Second * 2)


    // <-timer1.C 阻塞定时器的通道 C，直到它发送一个指示定时器超时的值。
    <-timer1.C
    fmt.Println("Timer 1 expired")
```

```
    // 如果只是想等待，可以使用 time.Sleep。定时器可能起作用的一个原因是在定时器到期之前取消定时器。这里有一个例子。
    timer2 := time.NewTimer(time.Second)
    go func() {
        <-timer2.C
        fmt.Println("Timer 2 expired")
    }()
    stop2 := timer2.Stop()
    if stop2 {
        fmt.Println("Timer 2 stopped")
    }
}
```

第一个定时器将在启动程序后约 2s 过期，但第二个定时器应该在它到期之前停止。执行上面代码，将得到以下输出结果 -

```
Timer 1 expired
```

```
Timer 2 stopped
```

28.2 tickers 定时器

计时器是当想在未来做一些事情 - tickers 是用于定期做一些事情。 这里是一个例行程序，周期性执行直到停止。

```
package main


import "time"
import "fmt"


func main() {


    // 代码机使用与计时器的机制类似：发送值到通道。 这里我们将使用通道上的一个范围内来迭代值，这此值每 500ms 到达。
    ticker := time.NewTicker(time.Millisecond * 500)
    go func() {
        for t := range ticker.C {
            fmt.Println("Tick at", t)
        }
```

```
    }()

    // 代码可以像计时器一样停止。当代码停止后，它不会在其通道上接收任何更多的值。我们将在 1600ms 后停止。
    time.Sleep(time.Millisecond * 1600)
    ticker.Stop()
    fmt.Println("Ticker stopped")
}
```

```
Tick at 2017-01-21 14:24:48.8807832 +0800 CST
Tick at 2017-01-21 14:24:49.380263 +0800 CST
Tick at 2017-01-21 14:24:49.882174 +0800 CST
Ticker stopped
```

28.3 速率限制示例

```
package main

import "time"
import "fmt"

func main() {

    // 首先我们来看一下基本速率限制。假设想限制对传入请求的处理。我们会在相同名称的通道上放送这些要求。
    requests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        requests <- i
    }
    close(requests)

    // 这个限制器通道将每 200 毫秒接收一个值。这是速率限制方案中的调节器。
    limiter := time.Tick(time.Millisecond * 200)
```

```
    //通过在服务每个请求之前阻塞来自限制器信道的接收，我们限制自己每 200 毫秒接收 1 个请求。
    for req := range requests {
        <-limiter
        fmt.Println("request", req, time.Now())
    }

    // 我们可能希望在速率限制方案中允许短脉冲串请求，同时保持总体速率限制。可以通过缓冲的限制器通道来实现。这个 burstyLimiter 通道将允许最多 3 个事件的突发。
    burstyLimiter := make(chan time.Time, 3)

    // 填充通道以表示允许突发。
    for i := 0; i < 3; i++ {
        burstyLimiter <- time.Now()
    }

    // 每 200 毫秒，将尝试向 burstyLimiter 添加一个新值，最大限制为 3。
    go func() {
        for t := range time.Tick(time.Millisecond * 200) {
            burstyLimiter <- t
        }
    }()

// 模拟 5 个更多的传入请求。这些传入请求中的前 3 个未超过 burstyLimiter 值。

//运行程序后，就会看到第一批请求每∽200 毫秒处理一次。

//对于第二批请求，程序会立即服务前 3 个，因为突发速率限制，然后剩余 2 服务//都具有∽200ms 延迟。

        burstyRequests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        burstyRequests <- i
```

```
    }
    close(burstyRequests)
    for req := range burstyRequests {
        <-burstyLimiter
        fmt.Println("request", req, time.Now())
    }
}
```

执行上面代码，将得到以下输出结果 -

```
request 1 2017-01-21 14:43:39.1445218 +0800 CST
request 2 2017-01-21 14:43:39.345767 +0800 CST
request 3 2017-01-21 14:43:39.5460635 +0800 CST
request 4 2017-01-21 14:43:39.7441739 +0800 CST
request 5 2017-01-21 14:43:39.9444929 +0800 CST
request 1 2017-01-21 14:43:39.9464898 +0800 CST
request 2 2017-01-21 14:43:39.9504928 +0800 CST
request 3 2017-01-21 14:43:39.9544955 +0800 CST
request 4 2017-01-21 14:43:40.1467214 +0800 CST
request 5 2017-01-21 14:43:40.3469624 +0800 CST
```

29 Go 语言的锁

29.1 原子变量

```
package main

import "fmt"
import "time"
import "sync/atomic"

func main() {

    // 使用一个无符号整数表示计数器(正数)。.
    var ops uint64 = 0
```

```
    // 为了模拟并发更新，将启动 50 个 goroutine，每个增量计数器大约是 1 毫秒。
for i := 0; i < 50; i++ {
        go func() {
            for {
                // 为了原子地递增计数器，这里使用 AddUint64()函数，在 ops 计数器的
//内存地址上使用＆语法。
                atomic.AddUint64(&ops, 1)

               //在增量之间等待一秒，允许一些操作累积。
                time.Sleep(time.Millisecond)
            }
        }()
    }

    // 在增量之间等待一秒，允许一些操作累积。
        time.Sleep(time.Second)

    // 为了安全地使用计数器，同时它仍然被其他 goroutine 更新，通过 LoadUint64 提取
一个当前值的副本到 opsFinal。
//如上所述，需要将获取值的内存地址&ops 给这个函数。
    opsFinal := atomic.LoadUint64(&ops)
    fmt.Println("ops:", opsFinal)
}
```

执行上面代码，将得到以下输出结果 -

```
ops: 41360
```

运行程序显示执行了大约 `40,000` 次操作。

29.2 互斥体示例

在前面的例子中，我们看到了如何使用原子操作来管理简单的计数器状态。对于更复杂的状态，可以使用互斥体来安全地访问多个 `goroutine` 中的数据。

```
package main


import (
```

```
    "fmt"
    "math/rand"
    "sync"
    "sync/atomic"
    "time"
)

func main() {

    // 在这个例子中，状态(state)是一个映射。
    var state = make(map[int]int)

    // 示例中的互斥将同步访问状态。.
    var mutex = &sync.Mutex{}

    // 我们将跟踪执行的读写操作的数量。
    var readOps uint64 = 0
    var writeOps uint64 = 0

    // 这里将启动 100 个 goroutine 来对状态执行重复读取，每个 goroutine
中每毫秒读取一次。
    for r := 0; r < 100; r++ {
        go func() {
            total := 0
            for {

 // 对于每个读取，我们选择一个键来访问，Lock()互斥体以确保对状态的独占
////访问，读取所选键的值，Unlock()互斥体，并增加 readOps 计数。
                key := rand.Intn(5)
                mutex.Lock()
                total += state[key]
                mutex.Unlock()
                atomic.AddUint64(&readOps, 1)
```

```
            // Wait a bit between reads.
            time.Sleep(time.Millisecond)
        }
    }()
}

// 我们还将启动 10 个 goroutine 来模拟写入，使用与读取相同的模式。
for w := 0; w < 10; w++ {
    go func() {
        for {
            key := rand.Intn(5)
            val := rand.Intn(100)
            mutex.Lock()
            state[key] = val
            mutex.Unlock()
            atomic.AddUint64(&writeOps, 1)
            time.Sleep(time.Millisecond)
        }
    }()
}

// 让 10 个 goroutine 在状态和互斥体上工作一秒钟。采集和报告最终操作计数。

time.Sleep(time.Second)

// 收集和报告最终操作计数。
readOpsFinal := atomic.LoadUint64(&readOps)
fmt.Println("readOps:", readOpsFinal)
writeOpsFinal := atomic.LoadUint64(&writeOps)
fmt.Println("writeOps:", writeOpsFinal)
```

```
    // 用最后的锁状态，显示它是如何结束的。
    mutex.Lock()
    fmt.Println("state:", state)
    mutex.Unlock()
}
```

Go

执行上面代码，将得到以下输出结果 -

```
readOps: 84546
writeOps: 8473
state: map[0:99 3:3 4:62 1:18 2:89]
```

29.3 Go 有状态的 goroutines 实例

在前面的示例中，我们使用显式锁定互斥体来同步对多个 goroutine 的共享状态的访问。 另一个选项是使用 `goroutine` 和通道的内置同步功能来实现相同的结果。

```
package main


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


// 在这个例子中，状态将由单个 goroutine 拥有。这将保证数据不会因并发访问而损坏。为
//了读或写状态，其他 goroutine 将发送消息到拥有的 goroutine 并接收相应的回复。这些
//readOp 和 writeOp 结构封装了这些请求，并拥有一个 goroutine 响应的方法。
type readOp struct {
    key  int
    resp chan int
}
type writeOp struct {
    key  int
    val  int
    resp chan bool
```

```
}


func main() {


    // 和以前一样，我们将计算执行的操作数。
    var readOps uint64 = 0
    var writeOps uint64 = 0


    // 读写通道将被其他 goroutine 分别用来发出读和写请求。
    reads := make(chan *readOp)
    writes := make(chan *writeOp)


    // 这里是拥有状态的 goroutine，它是一个如前面示例中的映射，但现在对状态 goroutine 是私有的。这个 goroutine 在读取和写入通道时重复选择，在请求到达时响应请求。 通过首先执行所请求的操作，然后在响应信道上发送值以指示成功(以及在读取的情况下的期望值)来执行响应。
    go func() {
        var state = make(map[int]int)
        for {
            select {
            case read := <-reads:
                read.resp <- state[read.key]
            case write := <-writes:
                state[write.key] = write.val
                write.resp <- true
            }
        }
    }()


    // 这里启动了 100 个 goroutine 来通过读取通道向状态拥有的 goroutine 发出读取。每次读取都需要构造一个 readOp，通过读取通道发送 readOp，并通过提供的 resp 通道接收结果。
    for r := 0; r < 100; r++ {
        go func() {
```

```
        for {
            read := &readOp{
                key:  rand.Intn(5),
                resp: make(chan int)}
            reads <- read
            <-read.resp
            atomic.AddUint64(&readOps, 1)
            time.Sleep(time.Millisecond)
        }
    }()
}

// 也使用类似的方法开始 10 个写操作。
for w := 0; w < 10; w++ {
    go func() {
        for {
            write := &writeOp{
                key:  rand.Intn(5),
                val:  rand.Intn(100),
                resp: make(chan bool)}
            writes <- write
            <-write.resp
            atomic.AddUint64(&writeOps, 1)
            time.Sleep(time.Millisecond)
        }
    }()
}

// 让 goroutine 工作一秒钟。
time.Sleep(time.Second)

// 最后，捕获和报告操作计数。
```

```
    readOpsFinal := atomic.LoadUint64(&readOps)
    fmt.Println("readOps:", readOpsFinal)
    writeOpsFinal := atomic.LoadUint64(&writeOps)
    fmt.Println("writeOps:", writeOpsFinal)
}
```

执行上面代码，将得到以下输出结果 -

```
readOps: 84546
writeOps: 8473
state: map[0:99 3:3 4:62 1:18 2:89]
```

30 defer

```
package main


import "fmt"
import "os"


// 假设要创建一个文件，写入内容，然后在完成之后关闭。这里可以这样使用延迟(defer)处理。.
func main() {


    // 在使用 createFile 获取文件对象后，立即使用 closeFile 推迟该文件的关闭。这将在 writeFile()完成后封装函数(main)结束时执行。
    f := createFile("defer-test.txt")
    defer closeFile(f)
    writeFile(f)
}


func createFile(p string) *os.File {
    fmt.Println("creating")
    f, err := os.Create(p)
    if err != nil {
        panic(err)
    }
```

```
    return f
}

func writeFile(f *os.File) {
    fmt.Println("writing")
    fmt.Fprintln(f, "data")

}

func closeFile(f *os.File) {
    fmt.Println("closing")
    f.Close()
}
```

执行上面代码，将得到以下输出结果 -

```
F:\worksp\golang>go run defer.go
creating
writing
closing
```

31 Go 语言正则表达式支持

```
package main

import "bytes"
import "fmt"
import "regexp"

func main() {

    // 是否匹配
    match, _ := regexp.MatchString("p([a-z]+)ch", "peach")
    fmt.Println(match)

    // 上面我们直接使用了字符串模式，但是其他的 regexp 的任务你需要编写一个` `
```

```
//` regexp `结构优化。
    r, _ := regexp.Compile("p([a-z]+)ch")

    // 此结构体有很多方法，如何上面一样的匹配
    fmt.Println(r.MatchString("peach"))

    // 查找
    fmt.Println(r.FindString("peach punch"))

    // 查找第一个匹配的位置
    fmt.Println(r.FindStringIndex("peach punch"))

    // 全局和括号里面的子集
    // for both `p([a-z]+)ch` and `([a-z]+)`.
    fmt.Println(r.FindStringSubmatch("peach punch"))

    // 全局和括号里面的子集的序号
        fmt.Println(r.FindStringSubmatchIndex("peach punch"))

    // 发现所有匹配的
    fmt.Println(r.FindAllString("peach punch pinch", -1))

    // 也适用于子集
    fmt.Println(r.FindAllStringSubmatchIndex(
        "peach punch pinch", -1))

    // 第 2 个参数限制匹配的数量
    fmt.Println(r.FindAllString("peach punch pinch", 2))

    // 字符数组也可以
    fmt.Println(r.Match([]byte("peach")))
```

```
    // 当用正则表达式创建常量时你可以使用 mustcompile 变化编译。
     r = regexp.MustCompile("p([a-z]+)ch")
    fmt.Println(r)

    // 替换
    fmt.Println(r.ReplaceAllString("a peach", "<fruit>"))

    // 自定义替换方式
    in := []byte("a peach")
    out := r.ReplaceAllFunc(in, bytes.ToUpper)
    fmt.Println(string(out))
}
```

执行上面代码，将得到以下输出结果 -

```
true
true
peach
[0 5]
[peach ea]
[0 5 1 3]
[peach punch pinch]
[[0 5 1 3] [6 11 7 9] [12 17 13 15]]
[peach punch]
true
p([a-z]+)ch
a <fruit>
a PEACH
```

32 Go 处理 json

```
package main

import "encoding/json"
import "fmt"
import "os"
```

```
// 我们将用这两个结构体来说明编码和解码
type Response1 struct {
    Page   int
    Fruits []string
}
type Response2 struct {
    Page   int      `json:"page"`
    Fruits []string `json:"fruits"`
}

func main() {

    // 首先来看看基本类型
    bolB, _ := json.Marshal(true)
    fmt.Println(string(bolB))

    intB, _ := json.Marshal(1)
    fmt.Println(string(intB))

    fltB, _ := json.Marshal(2.34)
    fmt.Println(string(fltB))

    strB, _ := json.Marshal("gopher")
    fmt.Println(string(strB))

    // 切片和 map 集合
    slcD := []string{"apple", "peach", "pear"}
    slcB, _ := json.Marshal(slcD)
    fmt.Println(string(slcB))

    mapD := map[string]int{"apple": 5, "lettuce": 7}
```

```
mapB, _ := json.Marshal(mapD)
fmt.Println(string(mapB))

// 自定义结构
res1D := &Response1{
    Page:   1,
    Fruits: []string{"apple", "peach", "pear"}}
res1B, _ := json.Marshal(res1D)
fmt.Println(string(res1B))

// 我们可以用标识来明确某一个 json 元素的
res2D := &Response2{
    Page:   1,
    Fruits: []string{"apple", "peach", "pear"}}
res2B, _ := json.Marshal(res2D)
fmt.Println(string(res2B))

// 编码 json 数据
byt := []byte(`{"num":6.13,"strs":["a","b"]}`)

//我们需要定义一个变量来存储数据，map[string]interface{}将 map 中的数据
// `map[string]interface{}` 将字符串映射到任意数据类型
var dat map[string]interface{}

// 解码，和相关错误的检查。

if err := json.Unmarshal(byt, &dat); err != nil {
    panic(err)
}
fmt.Println(dat)
```

```
    // 我们需要把它们转换成合适的类型。例如，在这里，我们将值放在 float64 型变量中。
    num := dat["num"].(float64)
    fmt.Println(num)

    // 访问嵌套数据需要一系列的强制转换。
        strs := dat["strs"].([]interface{})
    str1 := strs[0].(string)
    fmt.Println(str1)

    //我们也可以将 JSON 解码为自定义数据类型，这有助于在程序中添加额外的类型安全性，并且在访问解码数据时消除类型断言的需要
        str := `{"page": 1, "fruits": ["apple", "peach"]}`
    res := Response2{}
    json.Unmarshal([]byte(str), &res)
    fmt.Println(res)
    fmt.Println(res.Fruits[0])


    enc := json.NewEncoder(os.Stdout)
    d := map[string]int{"apple": 5, "lettuce": 7}
    enc.Encode(d)
}
```

执行上面代码，将得到以下输出结果 -

```
true
1
2.34
"gopher"
["apple","peach","pear"]
{"apple":5,"lettuce":7}
{"Page":1,"Fruits":["apple","peach","pear"]}
{"page":1,"fruits":["apple","peach","pear"]}
map[num:6.13 strs:[a b]]
```

```
6.13

a

{1 [apple peach]}

apple
```

```
{"apple":5,"lettuce":7}
```

33 Go 文件操作

33.1 读取文件

读取和写入文件是许多 Go 程序所需的基本任务。首先我们来看一些读取文件的例子。读取文件需要检查大多数调用错误。

读取和写入文件是许多 Go 程序所需的基本任务。首先我们来看一些读取文件的例子。读取文件需要检查大多数调用错误。

```
package main


import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)


// 读取文件需要检查大多数错误的调用。
func check(e error) {
    if e != nil {
        panic(e)
    }
}


func main() {


    // 取出文件的全部内容到内存
        dat, err := ioutil.ReadFile("/tmp/dat")
    check(err)
```

```
    fmt.Print(string(dat))

    //打开文件
    f, err := os.Open("/tmp/dat")
    check(err)

    // 从文件的开头读取一些字节。
//允许多达 5 个被读取
    b1 := make([]byte, 5)
    n1, err := f.Read(b1)
    check(err)
    fmt.Printf("%d bytes: %s\n", n1, string(b1))

    // 也可以在文件中找到已知位置, 从那里读.
    o2, err := f.Seek(6, 0)
    check(err)
    b2 := make([]byte, 2)
    n2, err := f.Read(b2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n2, o2, string(b2))

    // IO 包提供了一些有助于文件读取的功能。
        o3, err := f.Seek(6, 0)
    check(err)
    b3 := make([]byte, 2)
    n3, err := io.ReadAtLeast(f, b3, 2)
    check(err)
    fmt.Printf("%d bytes @ %d: %s\n", n3, o3, string(b3))

    // seek(0,0)将文件指针移动到开头
    _, err = f.Seek(0, 0)
    check(err)
```

```
    // `bufio` 包实现了缓冲
    // 使得读取效率更高
    r4 := bufio.NewReader(f)
    b4, err := r4.Peek(5)
    check(err)
    fmt.Printf("5 bytes: %s\n", string(b4))

    // 关闭文件
    f.Close()
}
```

执行上面代码，将得到以下输出结果 -

```
YWJjMTIzIT8kKiYoKSctPUB+
abc123!?$*&()'-=@~

YWJjMTIzIT8kKiYoKSctPUB-
abc123!?$*&()'-=@~
```

33.2 写文件

在 Go 中写入文件与读取文件的模式类似。首先我们来看一些读取文件的例子。写入文件需要检查大多数调用错误。

```
package main

import (
    "bufio"
    "fmt"
    "io/ioutil"
    "os"
)

func check(e error) {
    if e != nil {
        panic(e)
```

```
    }
}

func main() {

    // 写字符串到文件
    d1 := []byte("hello\ngo\n")
    err := ioutil.WriteFile("dat1.txt", d1, 0644)
    check(err)

    // 要获得更细粒度的写操作，请打开一个用于编写的文件。.
    f, err := os.Create("dat2.txt")
    check(err)

    // 打开文件后立即将“关闭”推迟是习惯用法。
        defer f.Close()

    // 你可以像预期的那样写字节片。
    d2 := []byte{115, 111, 109, 101, 10}
    n2, err := f.Write(d2)
    check(err)
    fmt.Printf("wrote %d bytes\n", n2)

    // A `WriteString` is also available.
    n3, err := f.WriteString("writes\n")
    fmt.Printf("wrote %d bytes\n", n3)

    // 发出同步刷新写入稳定存储
        f.Sync()

    // `bufio` 提供缓冲区读写操作
    w := bufio.NewWriter(f)
```

```
    n4, err := w.WriteString("buffered\n")
    fmt.Printf("wrote %d bytes\n", n4)

    // 刷新缓冲区
    w.Flush()

}
```

执行上面代码，将得到以下输出结果 -

```
wrote 5 bytes
wrote 7 bytes
wrote 9 bytes
```