Go语言基础讲义

第一章 Go语言介绍和环境安装

1 Go语言介绍

> **Go 语言官方介绍**
>
> Go 语言是有谷歌推出的一门编程语言。
>
> 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 语言主要开发者**
>
> 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。

> **Go 语言特点**
>
> - 简洁、快速、安全
> - 并行、有趣、开源
> - 内存管理、数组安全、编译迅速

> **Go 语言方向**
>
> - 网络编程领域
> - 区块链开发领域
> - 高性能分布式系统领域

2 Go语言环境安装

> **Go 语言的环境安装步骤如下:**
>
> - Go安装包下载网址(Go语言中文网 本视频发布网站)：https://studygolang.com/dl
> - 根据操作系统选择对应的安装包（Mac、Linux、Windows）。
> - 不要在安装路径中出现中文。

3 Go语言开发环境安装

> **Go 语言开发工具**
>
> - GoLand
> - GoLand 是 Jetbrains 家族的 Go 语言 IDE，有 30 天的免费试用期。
> - 下载安装网址：https://www.jetbrains.com/go/
> - 根据操作系统选择对应的安装包（Mac、Linux、Windows）下载对应的软件。
> - LiteIDE
> - LiteIDE 是一款开源、跨平台的轻量级 Go 语言集成开发环境（IDE）。
> - 下载安装网址：https://sourceforge.net/projects/liteide/files/
> - 根据操作系统选择对应的安装包（Linux、Windows）下载对应的软件。
> - 其他开发工具
> - Eclipse
> - VS Code

第二章 第一个Go语言程序

1 Hello world

> 下面就用IDE工具，开发第一个GO程序。
>
> Go 语言源文件的扩展是 .go
>
> 具体步骤如下：

```go
package main

import "fmt"

func main() {
  fmt.Println("Hello, World!")
  fmt.Println("性感法师在线教学~")
}
```

2 编译过程

> 要执行 Go 语言代码可以使用命令或IDE来完成编译。
>
> 命令如下：
>
> - 编译命令：go build hello.go
> - 编译并运行命令：go run hello.go

3 代码分析

> - 每个Go源代码文件的开头都是一个package声明，表示该Go代码所属的包。
> - import表示导入，“fmt”（format缩写）是包名 。无论是Go提供的包还是自定义的包都需要导入。
> - func（function缩写）表示函数，main表示主函数，表示程序的主入口。
> - main后的小括号 ( ) 表示函数参数列表， { } 表示函数体，即为函数功能的实现。
> - 使用了fmt包中的Println函数，将要“Hello, World! 打印在屏幕上。
> - 其它注意事项
>   - 强制左花括号 { 的放置位置，如果把左花括号 { 另起一行放置，Go 编译器报告编译错误。
>   - 每行代码占一行。

4 注释

4.1 注释作用

> 通过用自己熟悉的语言（例如，汉语），在程序中对某些代码进行标注说明，这就是注释的作用，能够大大增强程序的**可读性。**
>
> 以后，在公司中做开发，其他程序员会经常阅读我们写的代码，当然，我们也会经常阅读别的程序员写的代码，如果代码中加了注释，那么阅读起来就非常容易了。
>
> **注释不参与程序编译。**

4.2 注释分类

注释分类

- 行注释

```
//行注释，可以注释一行
```

- 块注释

```
/*
块注释
可以注释多行
*/
```

第三章 变量和常量

1 变量

1.1 变量定义

变量：在程序运行过程中其值可以发生改变的量成为变量。

变量定义

- 声明变量

```
var 变量名 数据类型
```

- 变量赋值

```
var 变量名 数据类型 = 值
```

- 自动推导类型

```
变量名 := 值   //根据值的类型 确定变量名的类型
```

在同一个函数内部变量名是唯一的。

1.2 多重赋值

在上面的讲解中，给变量赋值，采用了自动推导的方式，如果想一次使用自动推导的方式，给多个变量赋值，应该怎样实现呢？

具体如下：

```
a := 10
b := 20
c := 30
```

但是这种方式写起来非常的复杂，可以用如下的方式进行简化：

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

将10的值赋值给a，将10的值赋值给b，将30的值赋值给c 。

2 常量

2.1 常量定义和使用

> **常量**：在程序运行过程中其值不可以发生改变的量成为常量。
>
> 在程序开发中，用常量存储一直不会发生变化的数据，例如：π，身份证号码等。像这类的数据，在整个程序中运行中都是不允许发生改变的。

```
//常量定义
const 常量名 数据类型 = 值
//自动推导类型创建常量    不使用 :=
const 常量名 = 值
```

- 常量的值在定以后不允许修改。
- 常量的值不能获取地址。

2.2 常量集（iota枚举）

> 常量声明可以使用iota常量生成器初始化，它用于生成一组以相似规则初始化的常量，但是不用每行都写一遍初始化表达式。
>
> 注意：在一个const声明语句中，在第一个声明的常量所在的行，iota将会被置为0，然后在每一个有常量声明的行加一。
>
> 具体使用方式如下：

```
//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
const (
a = iota    //0
b = iota    //1
c = iota    //2
)
```

```
//与前一个例子相同 所以 a=0, b=1, c=2 可以简写为如下形式：
const (
a = iota    //0
b           //1
c           //2
)
```

```
//在同一行iota的值相同：
const (
a = iota                //0
b, c = iota, iota       //1
d, e                    //2
)
```

```
//常量集中的值 可以自定义
const (
a = 123
b = true
c = "hello"
)
```

3 命名

3.1 Go语言关键字

> **Go 语言有25个关键字：**
>
> ```
> break        default      func         interface    select
> case         defer        go           map          struct
> chan         else         goto         package      switch
> const        fallthrough  if           range        type
> continue     for          import       return       var
> ```
>
> - var 和 const ：变量和常量的声明
> - package 和 import : 导入
> - func： 用于定义函数和方法
> - return ：用于从函数返回
> - defer ：在函数退出之前执行
> - go : 用于并行
> - select ：用于选择不同类型的通讯
> - interface ：用于定义接口
> - struct：用于定义抽象数据类型
> - break、case、continue、for、fallthrough、else、if、switch、goto、default 流程控制
> - chan：用于channel通讯
> - type：用于声明自定义类型
> - map：用于声明map类型数据
> - range：用于读取slice、map、channel数据
>
> **Go 语言有36个预定义**：
>
> **预定义**
> 在Go中有很多预定义的名字，基本在内建的常量、类型和函数当中。
>
> 这些内部预定义的名字并不是关键字，它们是可以重新定义定义的
>
> ```
> append       bool         byte         cap          close
> complex
> complex64    complex128   uintptr      copy         false
> true
> float32      float64      imag         iota         int
> uint
> int8         uint8        int16        uint16       int32
> uint32
> int64        uint64       new          len          make
> panic
> nil          print        println      real         recover
> string
> ```

3.2 Go语言命名规则

> **命名规则**
>
> - 允许使用字母、数字、下划线
> - 不允许使用Go语言关键字
> - 不允许使用数字开头
> - 区分大小写

- 见名知义

驼峰命名法

- 小驼峰式命名法（lower camel case）：

  第一个单词以小写字母开始，第二个单词的首字母大写，例如：myName、aDog

- 大驼峰式命名法（upper camel case）：

  每一个单字的首字母都采用大写字母，例如：FirstName、LastName

4 标准输入输出

4.1 标准输入

前面所写的所有的程序，都是直接给变量赋值，但是很多情况下，我们希望用户通过键盘输入一个数值，存储到某个变量中，然后将该变量的值取出来，进行操作。

我们日常生活中也经常用到输入的场景:

- 在银行ATM机器前取钱时，肯定需要输入密码 。
- 在取钱时要输入金额。

在Go语言中提供三种标准输入函数：

```
//扫描，必须所有参数都被填充后换行才结束扫描
fmt.Scan()
//扫描，但是遇到换行就结束扫描
fmt.Scanln()
//格式化扫描，换行就结束
fmt.Scanf()
```

在Go中用到了 “fmt” 这个包中的Scan()函数来接收用户键盘输入的数据。当程序执行到Scan()函数后，会停止往下执行，等待用户的输入，输入完成后程序继续往下执行。

```
package main
import "fmt"

func main()  {
    x, y := 0, 0
    //提供标准输入设备获取内容并赋值给变量x,y
    fmt.Scan(&x,&y)

    fmt.Println(x)
    fmt.Println(y)
}
```

注意 这里x y变量前面一定要加上“&”符号，表示获取内存单元的地址，然后才能够存储。

4.2 标准输出

在Go语言中进行输出，用到前面所讲解的函数：Println()。

常用的标准输出函数有以下三种，都会将数据输出到标准设备，如屏幕。

```
//输出任意类型数据，并换行
fmt.Println()
//输出任意类型数据，不换行
fmt.Print()
//格式化输出数据，不换行
fmt.Printf()
```

```
package main
import "fmt"

func main()  {
    fmt.Println("fmt.Println()输出的数据，输出结果换行")
    fmt.Print("fmt.Print()输出的数据，输出结果不换行")
    fmt.Printf("fmt.Printf()输出的数据，格式化输出不换行 占位符：%s","法师")
}
```

结果：

```
fmt.Println()输出的数据，输出结果换行
fmt.Print()输出的数据，输出结果不换行fmt.Printf()输出的数据，格式化输出不换行 占位符：法师
```

第四章 基本数据类型

Go语言数据类型分为五种：

- 布尔类型
- 整型类型
- 浮点类型
- 字符类型
- 字符串类型

类型	名称	长度	零值	说明
bool	布尔类型	1	false	其值不为真即为假，不可以用数字代表true或false
byte	字节型	1	0	uint8别名

类型	名称	长度	零值	说明
int, uint	整型	-	0	根据操作系统设定数据的值。
int8	整型	1	0	-128 ~ 127
uint8	整型	1	0	0 ~ 255
int16	整型	2	0	-32768 ~ 32767
uint16	整型	2	0	0 ~ 65535
int32	整型	4	0	-2147483648 ~ 2147483647
uint32	整型	4	0	0 ~ 4294967295(42亿)
int64	整型	8	0	-9223372036854775808 ~ 9223372036854775807
uint64	整型	8	0	0 ~ 18446744073709551615(1844京)
float32	浮点型	4	0.0	小数位精确到7位
float64	浮点型	8	0.0	小数位精确到15位
string	字符串	-	""	utf-8字符串

1 布尔型

在计算机中，会涉及到逻辑性问题的真假对错，例如：在计算机中描述张三 (20岁) 比李四 (18岁) 小，这句话的结果？

布尔类型的变量取值要么是真 (true) ,要么是假 (false) ，用bool关键字来进行定义。

示例如下：

```
//定义bool类型变量
var 变量名 bool = true
```

```
package main
import "fmt"

func main()  {
  a := 10
  b := 20
  //比较两个数的大小 返回值为bool类型的结果
  fmt.Println(a > b)        //false
  fmt.Println(a < b)        //true
  //将表达式的结果赋值给bool类型变量c中
  var c bool = a < b
  fmt.Println(c)
  //格式化打印 使用占位符%t 输出一个bool类型数据
  fmt.Printf("%t", c)
}
```

2 整型

Go语言的整数类型一共有10个。

大多数情况下，我们只需要 int 一种整型即可，它可以用于循环计数器、数组和切片的索引，以及任何通用目的的整型运算符，通常 int 类型的处理速度也是最快的。

其中计算架构相关的整数类型有两个，即：

- 有符号的整数类型 int。
- 无符号的整数类型 uint。

```
//有符号的整型
var a int = -10
//无符号的整型(不可以存储带符号数据)
var b uint = 10

//格式化打印 使用占位符%d 输出一个int类型数据
fmt.Printf("%t", a)
```

这两个计算架构相关的整数类型之外，还有8个可以显式表达自身宽度的整数类型。

如下表所示。

数据类型	有符号	类型长度（位）	取值范围
int8	是	8	-128 ~ 127
int16	是	16	-32768 ~ 32767
int32	是	32	-2147483648 ~ 2147483647
int64	是	64	-9223372036854775808 ~ 9223372036854775807
uint8	否	8	0 ~ 255
uint16	否	16	0 ~ 65535
uint32	否	32	0 ~ 4294967295
uint64	否	64	0 ~ 18446744073709551615

可以看到，这8个整数类型的宽度已经表现在它们的名称中了。

3 浮点型

> Go语言提供了两种精度的浮点数 float32 和 float64。
>
> 这些浮点数类型的取值范围可以从很微小到很巨大。浮点数取值范围的极限值可以在 math 包中找到：
>
> - 常量 math.MaxFloat32 表示 float32 能取到的最大数值，大约是 3.4e38；
> - 常量 math.MaxFloat64 表示 float64 能取到的最大数值，大约是 1.8e308；
> - float32 和 float64 能表示的最小值分别为 1.4e-45 和 4.9e-324。
>
> 浮点型数据都是相对精准的，存在一定的误差值：
>
> - float32 类型的浮点数可以提供约 6 个十进制数的精度。
> - float64 类型的浮点数可以提供约 15 个十进制数的精度。
>
> 通常应该优先使用 float64 类型，因为 float32 类型的累计计算误差很容易扩散，并且 float32 能精确表示的正整数并不是很大。

> 浮点数在声明的时候可以只写整数部分或者小数部分，像下面这样：
>
> ```
> const e = .71828                    // 0.71828
> const f = 1.                        // 1.0
> ```
>
> 很小或很大的数最好用科学计数法书写，通过 e 或 E 来指定指数部分：
>
> ```
> const Avogadro = 6.02214129e23      // 阿伏伽德罗常数
> const Planck   = 6.62606957e-34     // 普朗克常数
> ```
>
> 用 Printf 函数打印浮点数时可以使用“%f”来控制保留几位小数
>
> ```
> package main
> import "fmt"
>
> func main() {
>   pi := 3.1415926
>   //格式化打印 使用占位符%d 输出一个float类型数据
>   //%f 默认保留六位小数 会四舍五入
>   fmt.Printf("%f\n", Pi)
>   //设置保留二位小数
>   fmt.Printf("%.2f\n", Pi)
> }
> ```

4 字符型

> Go语言的字符有以下两种：
>
> - 一种是 uint8 类型，或者叫 byte 型，代表了 **ASCII码** 的一个字符。
> - 一种是 rune 类型，代表一个 UTF-8 字符，当需要处理中文、日文或者其他复合字符时，则需要用到 rune 类型。
>
> byte 类型是 uint8 的别名，对于只占用 1 个字节的传统 ASCII 编码的字符来说，完全没有问题，
>
> 例如：

```
//字符类型是用单引号括起来的单个字符
var ch byte = 'A'
//格式化打印 使用占位符%c 输出一个byte类型数据
fmt.Printf("%c\n", ch)
```

rune 类型等价于 int32 类型，可以存储带中文的符合字符。

例如：

```
//rune 类型可以存储带中文的字符
var ch rune = '帅'
//格式化打印 使用占位符%c 输出一个rune类型数据
fmt.Printf("%c\n", ch)
```

5 字符串型

用双引号括起来的字符是字符串类型 。

在Go中的字符串，都是采用UTF-8字符集编码。

```
var str string = "法师真帅"
//格式化打印 使用占位符%s 输出一个string类型数据
fmt.Printf("%s\n", ch)
```

字符串拼接

```
str1 := "性感法师"
str2 := "在线讲课"
fmt.Println(str1 + str2)        //性感法师在线讲课
```

字符串长度

```
str1 := "hello"
str2 := "法师"
//len(字符串) 计算字符串中字符个数
fmt.Println(len(str1))          //5
fmt.Println(len(str2))          //6 一个汉字占三个字符
```

6 进制（了解）

进制也就是进位制，是人们规定的一种进位方法。

对于任何一种进制—X进制，就表示某一位置上的数运算时是逢X进一位。 十进制是逢十进一，十六进制是逢十六进一，二进制就是逢二进一，以此类推，x进制就是逢x进位。

6.1 二进制

二进制，Binary，缩写BIN，是计算技术中广泛采用的一种数制。

二进制数据是用0和1两个数码来表示的数。

它的基数为2，进位规则是“逢二进一”，借位规则是“借一当二”。

计算机系统使用的基本上是二进制系统，数据在计算机中主要是以**补码**的形式存储的。

6.2 八进制

> **八进制**，Octal，缩写OCT，一种以8为基数的计数法。
>
> 采用0，1，2，3，4，5，6，7八个数字。
>
> 逢八进一。一些编程语言中常常以数字0开始表明该数字是八进制。

6.3 十六进制

> **十六进制**，Hexadecimal，缩写HEX，同我们日常生活中的表示法不一样，它由0-9，A-F组成，字母不区分大小写。
>
> 与10进制的对应关系是：0-9对应0-9，A-F对应10-15。
>
> 在计算机中十六进制一般表示内存地址。

6.4 进制转换

```go
package main

import "fmt"

func main() {
    a := 10          //十进制
    b := 010         //八进制
    c := 0x10        //十六进制

    //使用占位符打印二进制
fmt.Printf("%b\n",a)
    //使用占位符打印十进制
    fmt.Printf("%d\n",a)
    //使用占位符打印八进制
    fmt.Printf("%o\n",b)
    //使用占位符打印十六进制
    fmt.Printf("%x\n",c)
}
```

> 不能在程序中定义二进制数据。

6.5 计算机数据存储方式

6.5.1 原码

6.5.2 反码

6.5.3 补码

第五章 运算符和表达式

> **运算符**：用于在程序运行时执行数学或逻辑运算。
>
> **表达式**：使用运算符将数据（变量、常量、函数返回值）连接起来的式子。

1 算数运算符

运算符	术语	示例	结果
+	加	10 + 5	15
-	减	10 - 5	5
*	乘	10 * 5	50
/	除	10 / 5	2
%	取模(取余)	10 % 3	1
++	后自增，没有前自增	a=0; a++	a=1
--	后自减，没有前自减	a=2; a--	a=1

示例：

```
a := 10
b := 20
//算数运算符的表达式计算
fmt.Println(a + b)          //30
fmt.Println(a - b)          //-10
fmt.Println(a * b)          //200
fmt.Println(a / b)          //0
fmt.Println(a % b)          //10
fmt.Println(a++)            //11
fmt.Println(b--)            //19
```

2 赋值运算符

赋值运算符将表达式的结果赋值给变量。

关于赋值运算符前面我们已经使用过多次，赋值运算符 =

运算符	说明	示例
=	普通赋值	c = a + b 将 a + b 表达式结果赋值给 c
+=	相加后再赋值	c += a 等价于 c = c + a
-=	相减后再赋值	c -= a 等价于 c = c - a
*=	相乘后再赋值	c *= a 等价于 c = c * a
/=	相除后再赋值	c /= a 等价于 c = c / a
%=	求余后再赋值	c %= a 等价于 c = c % a

也可以与位运算符组合使用。

3 关系运算符

关系运算符又称为比较运算符

关系运算的结果是布尔类型的。

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

示例：

```
a := 10
b := 20
//关系运算符的表达式计算
fmt.Println(a == b)             //false
fmt.Println(a != b)             //true
fmt.Println(a < b)              //true
fmt.Println(a > b)              //false
fmt.Println(a >= b)             //false
fmt.Println(a <= b)             //false
```

4 逻辑运算符

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

5 位运算符

位逻辑运算符

运算符	术语	示例	结果
&(AND)	位与	3 & 9	结果：1 相同位都是1时结果才为1。
|(OR)	位或	3 | 9	结果：11 相同位只要一个为1即为1。
^(XOR)	位异或	3 ^ 9	结果：10 相同位不同则为1，相同则为0。
&^ (AND NOT)	位清空	3 &^ 9	结果：2 后数为0，则用前数对应位代替 后数为1则取0

位移运算符

运算符	术语	示例	结果
<<	左移	10 << 2	40
>>	右移	10 >> 2	2

6 其他运算符

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

7 运算符优先级

运算符是用来在程序运行时执行数学或逻辑运算的，在Go语言中，一个表达式可以包含多个运算符，当表达式中存在多个运算符时，就会遇到优先级的问题，此时应该先处理哪个运算符呢？

这个就由Go语言运算符的优先级来决定的。

Go语言有几十种运算符，被分成十几个级别，有的运算符优先级不同，有的运算符优先级相同。

请看下表：

优先级	分类	运算符	结合性
1	逗号运算符	,	从左到右
2	赋值运算符	=、+=、-=、*=、/=、 %=、 >=、 <<=、&=、^=、 |=	从右到左
3	逻辑或	||	从左到右
4	逻辑与	&&	从左到右
5	按位或	|	从左到右
6	按位异或	^	从左到右
7	按位与	&	从左到右
8	相等/不等	==、!=	从左到右
9	关系运算符	<、<=、>、>=	从左到右
10	位移运算符	<<、>>	从左到右
11	加法/减法	+、-	从左到右
12	乘法/除法/取余	*（乘号）、/、%	从左到右
13	单目运算符	!、*（指针）、&、++、--、+（正号）、-（负号）	从右到左
14	后缀运算符	()、[]、->	从左到右

注意：优先级值越大，表示优先级越高。

8 类型转换

数据有不同的类型，不同类型数据之间进行混合运算时必然涉及到类型的转换问题。

两种不同的类型在计算时，Go语言要求必须进行类型转换。

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：

```
数据类型(变量)          //将变量转成指定的类型
数据类型(表达式)        //将表达式转成指定的类型
```

第六章 流程控制

1 选择结构

选择结构也称为判断结构。生活中的关于判断的场景也非常多，如下：

- 登录账号时要输入正确的用户名和密码。
- 在ATM机取钱时输入正确的金额。
- 在玩游戏时输入指令键控制英雄。

1.1 if语句

在编程中实现选择判断结构就是用 if

if结构语法

```
if 条件判断 {
    //代码语句
}
```

条件判断如果为真（true），那么就执行大括号中的语句；如果为假（false），就不执行大括号中的语句。

if else结构语法

```
if 条件判断 {
    //代码语句1
}else{
    //代码语句2
}
```

条件判断如果为真（true），那么就执行if大括号中的语句。

条件判断如果为假（false），那么就执行else大括号中的语句。

if代码块或else代码块，必须有一块被执行。

if else if结构语法

```
if 条件判断1 {
    //代码语句1
}else if 条件判断2{
    //代码语句2
}else if 条件判断3{
    //代码语句3
}else{
    //代码语句4
}
```

从上到下依次判断条件，如果结果为真，就执行 { } 内的代码 。

1.2 switch语句

```
switch 变量或者表达式的值{
    case 值1:
        //代码语句1
    case 值2:
        //代码语句2
    case 值3:
        //代码语句3
    default:
        //代码语句4
}
```

执行流程：

程序执行到switch处，首先将变量或者表达式的值计算出来，然后拿着这个值依次跟每个case后面所带的值进行匹配，一旦匹配成功，则执行该case所带的代码，执行完成后，跳出switch-case结构。

如果，跟每个case所带的值都不匹配。就看当前这个switch-case结构中是否存在default，如果有default，则执行default中的语句，如果没有default，则该switch-case结构什么都不做。

注意：

某个case 后面跟着的代码执行完毕后，不会再执行后面的case，而是跳出整个switch结构， 相当于每个case后面都跟着break(终止)。

但是如果想执行完成某个case后，强制执行后面的case，可以使用fallthrough。

2 循环结构

2.1 for语句

语法结构如下：

```
for 表达式1;表达式2;表达式3 {
    //循环体
}
```

表达式1：定义一个循环的变量，记录循环的次数。

表达式2：一般为循环条件，循环多少次。

表达式3：一般为改变循环条件的代码，使循环条件终有不再成立。

循环体：重复要做的事情。

示例1：

```
package main
import "fmt"

func main() {
    sum := 0
    //计算1-100的和
    for i := 1; i <= 100; i++ {
        sum += i
    }
    fmt.Println(sum)
}
```

示例2：

```
package main
import "fmt"

func main() {
    //水仙花数
    //一个三位数 各个位数的立方和等于本身的数
    for i := 100; i <= 999; i++ {
        a := i / 100      //百位
        b := i / 10 % 10 //十位
        c := i % 10       //个位
        if a*a*a+b*b*b+c*c*c == i {
            fmt.Println(i)
        }
    }
}
```

2.2 嵌套循环

循环语句之间可以相互嵌套：

```
for 循环条件{
 for 循环条件{
    //执行代码
 }
}
```

2.3 跳出语句

break语句

在循环语句中可以使用break跳出语句：

- 当它出现在循环语句中，作用是跳出当前内循环语句，执行后面的代码。
- 当它出现在嵌套循环语句中，跳出最近的内循环语句，执行后面的代码。

continue语句

在循环语句中，如果希望立即终止本次循环，并执行下一次循环，此时就需要使用continue语句。

第七章 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。

系统函数调用：随机数

当调用函数时，需要关心5要素：

- 导入包：指定具体包下的函数
- 函数名字：函数名字必须和包文件声明的名字一样

- 功能：需要知道此函数的用途
- 参数：参数类型要匹配
- 返回值：根据需要接收返回值

```
package main

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

func main() {
    //使用时间戳作为随机数种子进行混淆
    rand.Seed(time.Now().UnixNano())
    //限定100以内的随机数
    value := rand.Intn(100)
    fmt.Println(value)

}
```

1 函数定义和使用

Go 语言函数定义格式如下：

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

函数名

理论上是可以随意起名字，最好起的名字见名知意，应该让用户看到这个函数名字就知道这个函数的功能。注意，函数名的后面有个圆换号()，代表这个为函数，不是普通的变量名。

形参列表

在定义函数时指定的形参，在未出现函数调用时，它们并不占内存中的存储单元，因此称它们是形式参数或虚拟参数，简称形参，表示它们并不是实际存在的数据，所以，形参里的变量不能赋值。

函数体

花括号{ }里的内容即为函数体的内容，这里为函数功能实现的过程，这和以前的写代码没太大区别，以前我们把代码写在main()函数里，现在只是把这些写到别的函数里。

返回值

函数的返回值是通过函数中的return语句获得的，return后面的值也可以是一个表达式。Go语言支持多个返回值。

2 实参和形参

- 形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。
- 实参出现在主调函数中，进入被调函数后，实参也不能使用。
- 实参变量对形参变量的数据传递是“值传递”，即单向传递，只由实参传给形参，而不能由形参传回来给实参。

- 在调用函数时，编译系统临时给形参分配存储单元。调用结束后，形参单元被释放。
- 实参单元与形参单元是不同的单元。调用结束后，形参单元被释放，函数调用结束返回主调函数后则不能再使用该形参变量。实参单元仍保留并维持原值。因此，在执行一个被调用函数时，形参的值如果发生改变，并不会改变主调函数中实参的值。

3 函数的返回值

- 保证return语句中表达式的值和函数返回类型是同一类型。
- 如果函数返回的类型和return语句中表达式的值不一致，程序则会报错。
- return语句的另一个作用为中断return所在的执行函数。
- 如果函数带返回值，return后面必须跟着值 。

4 匿名函数

4.1 匿名函数

Go是支持匿名函数的，即在需要使用函数时。再定义函数,匿名函数没有函数名。只有函数体，函数可以被作为一种类型被赋值给函数类型的变量，匿名函数往往以变量方式被传递。

匿名函数经常被用于实现回调函数,闭包等.

匿名函数定义

匿名函数的定义就是: 没有名字的普通函数

```
func （参数列表）（返回值列表） {
函数体
}
```

匿名函数的调用

- 在定义时调用匿名函数

匿名函数可以在声明后直接调用; 例如:

```
package main

import "fmt"

func main() {
   // 定义匿名函数并赋值给f变量
   f := func(data int) {
     fmt.Println("hello", data)
   }
 // 此时f变量的类型是func(), 可以直接调用
   f(100)
}
```

匿名函数的用途非常广泛,匿名函数本身是一种值,可以方便的保存在各种容器中实现回调函数和操作封装

- 匿名函数做回调函数

回调函数，或简称**回调**（Callback 即call then back 被主函数调用运算后会返回主函数），是指通过函数参数传递到其它代码的，某一块可执行代码的引用

匿名函数作为回调函数的设计在go语言的系统包中是很长见的。

strings包中就有这种实现:

```
func TrimFunc(s string, f func(rune) bool) string {
       return TrimRightFunc(TrimLeftFunc(s, f), f)
}
```

4.2 闭包

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，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 */
    f1 := getSequence()

    /* 调用 f1 函数，i 变量自增 1 并返回 */
    fmt.Println(f1())
    fmt.Println(f1())
    fmt.Println(f1())

    /* 创建新的函数 f2，并查看结果 */
    f2 := getSequence()
    fmt.Println(f2())
    fmt.Println(f2())
}
```

结果：

```
1
2
3
1
2
```

5 不定参函数

在定义函数的时候根据需求指定参数的个数和类型，但是有时候如果无法确定参数的个数呢？

例如：计算n个整数的和？

在这个要求中，项并没有说清楚到底是有几个整数，那么我们应该怎样确定该函数的参数呢？

可以通过“不定参数列表“来解决这个问题

可以通过如下的方式来定义函数：

```
func 函数名(数据集合 ...数据类型)
```

在数据集合后面跟了三个点，就是表示该参数可以接收0或多个整数值。

所以，这个参数可以想象成是一个集合（类似数学中集合），可以存放多个值。

如何进行不定参函数的数据集合计算？

```
//使用len进行范围遍历
//计算集合中元素个数
//len(数据集合)
for i:=0; i< len(数据集合); i++{
//操作代码
}
```

```
//使用range关键字进行范围遍历
/*  range会从集合中返回两个数
    第一个是对应的坐标，赋值给了变量index
    第二个就是对应的值，赋值了变量data
*/
for index,data := range 数据集合{
//操作代码
}
```

示例代码

```
package main

import "fmt"

//计算n个整数的和s
func sum(args ...int) int {

    value := 0
    //可以使用len 或 range 来遍历不定参集合中的元素
    //for index,data := range args
    for i := 0; i < len(args); i++ {
        value += args[i]
    }
    return value
}
func main() {
 //函数调用 传递n个整数数据
    value := sum(1, 2)
    fmt.Println(value)
    value = sum(1, 2, 3, 4, 5)
    fmt.Println(value)
}
```

6 函数嵌套调用

函数也可以像在前面学习 if 选择结构，for 循环结构一样进行嵌套使用。所谓函数的嵌套使用，其实就是在一个函数中调用另外的函数。

```
func 函数1(函数参数) 结果类型{
 //函数调用
 函数2(函数参数)
 //代码实现
 return 结果
}
func 函数2(函数参数) 结果类型{
 //代码实现
 return 结果
}
func main(){
 函数1(函数参数)
}
```

执行流程

- 先执行main()函数，在main()函数中调用 ” 函数1() “ 函数，如果有参数将其传递给 ” 函数1() “
- ” 函数1() “ 函数中调用 ” 函数2() “ 函数，如果有参数将其传递。
- 执行 ” 函数2() “ 函数中的代码，然后将结果返回。
- ” 函数2() “ 函数中所有的代码执行完成后，会回到” 函数1() “ 函数，” 函数1() “ 函数剩余的代码。
- 当” 函数1() “ 函数中所有的代码执行完成后，会回到main( )函数，执行main( )函数后面剩余的代码。

6.1 递归函数

通过前面的学习知道一个函数可以调用其他函数。

如果一个函数在内部不调用其它的函数，而是调用自己本身，这个函数就是递归函数。

应用场景：

电商网站中的商品类别菜单的应用。

查找某个磁盘下的文件。

示例代码

```
package main

import "fmt"
//计算n的阶乘

//定义全局变量
var sum int = 1

//计算数的阶乘
func test(num int) {
    if num == 1 {
        return //递归函数的出口
    }
    test(num - 1)   //函数自己调用自己
    sum *= num
}
```

```
func main() {
    test(6)

    fmt.Println(sum)
}
```

第八章 工程管理

1 工作空间

> 工作空间是Go中的一个对应于特定工程的目录，其包括src，pkg，bin三个目录
>
> - src：用于以代码包的形式组织并保存Go源码文件。（比如：.go .c .h .s等）
> - pkg：用于存放经由go install命令构建安装后的代码包（包含Go库源码文件）的“.a”归档文件。
> - bin：与pkg目录类似，在通过go install命令完成安装后，保存由Go命令源码文件生成的可执行文件。
>
> 目录src用于包含所有的源代码，是Go命令行工具一个强制的规则，而pkg和bin则无需手动创建，如果必要Go命令行工具在构建过程中会自动创建这些目录。

2 多文件编程

> 在src目录下，可以创建多个文件或者目录，将内容按照指定模块进行写入在不同文件中。
>
> 同级目录多个源文件：
>
> - 同一个目录，调用其他文件内的函数，直接调用即可，无需包名引用
> - 在使用时不区分函数大小写
>
> 不同级目录多个源文件：
>
> - 目录内的包名必须一致
> - 在其他目录使用当前目录的函数时，如果函数首字母大写，可以被外部使用
> - 调用格式为：包名.函数名
>
> 目录格式：

```
src
├── main.go
└── userinfo
    ├── login.go
    └── test.go
```

3 go mod

> Go mod模块是相关Go包的集合。modules源代码交换和版本控制的单元。go命令直接支持使用modules，包括记录和解析对其他模块的依赖性。modules替换旧的基于GOPATH的方法来指定在给定构建中使用哪些源文件。
>
> 设置 GO111MODULE
>
> - off
>
>   go命令行将不会支持module功能，寻找依赖包的方式将会沿用旧版本那种通过vendor目录或者GOPATH模式来查找。

- on

  go命令行会使用modules，而一点也不会去GOPATH目录下查找。

- auto

  默认值，go命令行将会根据当前目录来决定是否启用module功能

当modules 功能启用时，依赖包的存放位置变更为$GOPATH/pkg，允许同一个package多个版本并存，且多个项目可以共享缓存的 module。

3.1 go mod 命令

go mod 有以下命令来管理包。

命令	说明
download	download modules to local cache(下载依赖包)
edit	edit go.mod from tools or scripts（编辑go.mod)
graph	print module requirement graph (打印模块依赖图)
init	initialize new module in current directory（在当前目录初始化mod）
tidy	add missing and remove unused modules(拉取缺少的模块，移除不用的模块)
vendor	make vendored copy of dependencies(将依赖复制到vendor下)
verify	verify dependencies have expected content (验证依赖是否正确）
why	explain why packages or modules are needed(解释为什么需要依赖)

3.2 go mod 创建项目

在GOPATH 目录之外新建一个目录，并使用go mod init 初始化生成go.mod文件

```
go mod init 项目名
```

go.mod文件一旦创建后，它的内容将会被go toolchain全面掌控。go toolchain会在各类命令执行时，比如go get、go build、go mod等修改和维护go.mod文件。

go.mod 提供了module, require、replace和exclude 四个命令

- module 语句指定包的名字（路径）
- require 语句指定的依赖项模块
- replace 语句可以替换依赖项模块
- exclude 语句可以忽略依赖项模块

```
package main

import "github.com/astaxie/beego"

func main(){
    beego.Run()
}
```

执行 go run test.go 运行代码会发现 go mod 会自动查找依赖自动下载并完成。

第九章 错误处理

> 当Go语言在执行时检测到一个错误时，程序就无法继续执行，出现错误的提示，这就是所谓的“异常”。
>
> 所以为了保证程序的健壮性，要对异常的信息进行处理。
>
> 例如：定义一个函数实现整除操作

```
//计算两个数的相除
func test(a int, b int) (result int) {
    result = a / b
    return result
}
```

> 但是，大家仔细考虑一下，该函数是否有问题？
>
> 如果b的值为0，会出现什么情况？
>
> 程序会出现以下的异常信息：

```
panic: runtime error: integer divide by zero
```

> 并且整个程序停止运行。
>
> 那么出现这种情况，应该怎样进行处理呢？这时就要用到异常处理方法的内容。

1 error接口

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

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

> 通过以上代码，可以发现error接口的使用是非常简单的（error是一个接口，该接口只声明了一个方法Error()，返回值是string类型，用以描述错误。
>
> **基本使用**
>
> - 首先导包：import "errors"
> - 然后调用其对应的方法：errors.New(“显示错误信息”)
> - 当然fmt包中也封装了一个专门输出错误信息的方法 ：fmt.Errorf()
>
> **示例代码**

```
package main

import (
    "errors"
    "fmt"
)

//使用error处理错误信息
func test(a int, b int) (result int, err error) {
```

```
    if b == 0 {
        err = errors.New("除数不能为零")
        return
    }
    result = a / b
    return
}

func main() {
    //使用result接收结果
    //使用err接收错误信息
    result, err := test(10, 0)
    //判断是否捕获错误信息
    if (err != nil) {
        fmt.Println(err)
    } else {
        fmt.Println(result)
    }
}
```

2 panic函数

error返回的是一般性的错误，但是panic函数返回的是让程序崩溃的错误。

当遇到不可恢复的错误状态的时候，如数组访问越界、空指针引用等，这些运行时错误会引起painc异常。

一般而言，当panic异常发生时，程序会中断运行。随后，程序崩溃并输出日志信息。日志信息包括panic value和函数调用的堆栈跟踪信息。

当然，如果直接调用内置的panic函数也会引发panic异常，panic函数接受任何值作为参数。

下面演示一下，直接调用panic函数，是否会导致程序的崩溃。

```
func main() {
    fmt.Println("hello world")
    panic("调用panic函数，程序崩溃并输出异常")
    fmt.Println("法师帅不帅？")
}
```

3 defer延迟调用

defer语句被用于预定对一个函数的调用。可以把这类被defer语句调用的函数称为延迟函数。

defer作用：

- 释放占用的资源
- 捕捉处理异常
- 输出日志

```
func Demo(){
    defer fmt.Println("性感法师")
    defer fmt.Println("在线教学")
    defer fmt.Println("日薪越亿")
    defer fmt.Println("轻松就业")
}
func main() {
    Demo()
}
```

结果

```
轻松就业
月薪过万
在线教学
性感法师
```

如果一个函数中有多个defer语句，它们会以LIFO（后进先出）的顺序执行。

4 recover错误拦截

运行时panic异常一旦被引发就会导致程序崩溃。

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

```
func recover interface{}
```

注意：recover只有在defer调用的函数中有效。

示例代码

```
package main

import "fmt"

func Demo(i int) {
    //定义10个元素的数组
    var arr [10]int
    //错误拦截要在产生错误前设置
    defer func() {
        //设置recover拦截错误信息
        err := recover()
        //产生panic异常  打印错误信息
        if err != nil {
            fmt.Println(err)
        }
    }()
    //根据函数参数为数组元素赋值
    //如果i的值超过数组下标 会报错误：数组下标越界
    arr[i] = 10

}

func main() {
```

```
    Demo(10)
    //产生错误后 程序继续
    fmt.Println("程序继续执行...")
}
```

结果

```
runtime error: index out of range
程序继续执行...
```

如果程序没有异常，不会打印错误信息。

第十章 高级数据格式

1 数组

> 数组：是指一组相同类型的数据在内存中有序存储的集合。

1.1 数组定义

```
var 数组名 [元素个数]数据类型
var arr [10]int
```

数组定义也是通过 var 关键字，后面是数组的名字arr，长度是10，数据类型是整型。

表示：数组arr能够存储10个整型数字。也就是说，数组arr的长度是10。

可以通过len( )函数测试数组的长度，如下所示：

```
var arr [10]int
//计算数组中元素个数
len(arr)    //10
```

当定义完成数组arr后，就在内存中开辟了10个连续的存储空间，每个数据都存储在相应的空间内，数组中包含的每个数据被称为数组元素（element），一个数组包含的元素个数被称为数组的长度。

注意：数组的长度只能是常量。

1.2 数组的赋值

> 数组定义完成后，可以对数组进行赋值操作。
>
> 数组是通过**下标**来进行操作的，下标的范围是从0开始到数组长度减1的位置。

```
//数组名[下标] = 值
int arr [10]int
arr[0] = 123
```

> 如果数组下标小于0或大于数组长度减1，则出现错误为数组**下标越界**
>
> 如果数组定义未赋值，则根据不同类型的数据进行初始化。
>
> 整型默认结果为0，浮点型默认结果为0，字符串默认结果为 "" ，布尔类型为false。

1.3 数组初始化

在定义数组时，也可以完成赋值，这种情况叫做数组的初始化。

```
//1、全部初始化
var a [5]int = [5]int{1, 2, 3, 4, 5}
//2、自动推导类型初始化
b := [5]int{1, 2, 3, 4, 5}
//3、初始化部分数据 未初始化值为默认值
c := [5]int{1, 2, 3}
//4、指定元素初始化 未初始化值为默认值
d := [5]int{1: 123, 3: 666}
```

1.4 数组遍历

通过for循环可以完成数组的赋值与输出。

在上一节中说过可以通过 len( ) 函数来获取数组的长度 。

示例代码

```
var arr [5]int = [5]int{1, 2, 3, 4, 5}
for i := 0; i < len(arr); i++ {
 //根据下标 遍历数组元素
 fmt.Println(arr[i])
}
```

```
var arr [5]int = [5]int{1, 2, 3, 4, 5}
//使用范围遍历
//通过range遍历集合 可以得到 index 下标 data 元素的值
for index, data := range arr {
 fmt.Println(index, data)
}
```

1.5 数组示例

从一个整数数组中取出最大的整数，最小整数，总和，平均值。

```
func main() {
    var arr [10]int = [10]int{9, 1, 5, 6, 10, 8, 3, 7, 2, 4}
    max, min, sum := arr[0], arr[0], 0
    for i := 0; i < len(arr); i++ {
        //最大值
        if arr[i] > max {
            max = arr[i]
        }
        //最小值
        if arr[i] < min {
            min = arr[i]
        }
        //总和
        sum += arr[i]
    }
    fmt.Println("最大值: ", max)
    fmt.Println("最小值: ", min)
    fmt.Println("总和: ", sum)
    fmt.Println("平均值: ", sum/len(arr))
}
```

结果

```
最大值： 10
最小值： 1
总和： 55
平均值： 5
```

将一个数组的元素的顺序进行反转。

例如：湾前过渡小舟虚

输出：虚舟小渡过前湾

```
func main() {
    var arr []rune = []rune{'湾', '前', '过', '渡', '小', '舟', '虚'}
    //最小值下标
    i := 0
    //最大值下标
    j := len(arr) - 1
    for i < j {
        //多重赋值  交换两个变量的值
        arr[i], arr[j] = arr[j], arr[i]
        //改变下标的值
        i++
        j--
    }
 //类型转换 打印中文
    fmt.Println(string(arr))
}
```

数组排序，将一个无序数组变成有序数组

例如：{9, 1, 5, 6, 10, 8, 3, 7, 2, 4}

输出：{1 2 3 4 5 6 7 8 9 10}

```
//冒泡排序
func main() {
    arr := [10]int{9, 1, 5, 6, 10, 8, 3, 7, 2, 4}

    //外层控制行 表示执行次数
    for i := 0; i < len(arr)-1; i++ {
        //内层控制列 表示比较次数
        for j := 0; j < len(arr)-1-i; j++ {
            //比较两个相邻元素
            if arr[j] > arr[j+1] {
                //交换数据
                arr[j], arr[j+1] = arr[j+1], arr[j]
            }

        }
    }
    fmt.Println(arr)
}
```

1.6 数组作为函数参数

数组也可以像变量一样，作为参数传递给函数。

```
func modify(arr [10]int) {
    //修改数组元素的值
    arr[0] = 123
    //被调函数中 数组元素被修改
    fmt.Println(arr)
}
func main() {
    arr := [10]int{9, 1, 5, 6, 10, 8, 3, 7, 2, 4}
    //将数组作为函数参数
    //值传递  形参不能改变实参的值
    modify(arr)
    //主调函数中 数组元素位被修改
    fmt.Println(arr)
}
```

注意：在main( )函数中，定义数组arr, 然后调用modify( )方法传递数组，同时在modify( )方法中修改数组中第一个元素。最终输出结果发现，并不会影响main( )函数中数组a的值。

可以打印主调函数和被调函数的参数地址：

```
func Demo(arr [10]int) {
    fmt.Printf("被调函数中 形参数组地址：%p\n", &arr)
}
func main() {
    arr := [10]int{9, 1, 5, 6, 10, 8, 3, 7, 2, 4}
    fmt.Printf("主调函数中 实参数组地址：%p\n", &arr)
    //函数调用
    Demo(arr)

}
```

结果

```
主调函数中 实参数组地址：0xc04207c050
被调函数中 形参数组地址：0xc04207c0a0
```

形参和实参操作的数组不是同一个地址中的数据，所以修改被调函数中的数组元素的值，不会影响主调函数中实参数组元素的值。

2 切片

切片(slice)是 Go中一种比较特殊的数据结构，这种数据结构更便于使用和管理数据集合。

切片是围绕动态数组的概念构建的，可以按需自动增长。

2.1 切片定义

```
//1、定义一个空切片(nil)
//var 切片名 []数据类型
var slice []int //空切片不能添加数据
//2、通过make创建切片
//make([]数据类型,长度,容量)
var slice []int = make([]int,5,10)
//3、自动推导类型创建切片
slice := []int{1,2,3,4,5}
slice := make([]int,5,10)
```

长度与容量

长度是已经初始化的空间（以上切片slice初始空间默认值都是0）。

```
//计算切片有效数据个数
len(slice)
```

容量是已经开辟的空间，包括已经初始化的空间和空闲的空间。

```
//计算切片容量
cap(slice)
```

2.2 切片扩容

在第一节中，已经讲解过切片与数组很大的一个区别就是：切片的长度是不固定的，可以向已经定义的切片中追加数据。

通过append的函数，在原切片的末尾添加元素。

```
//可以向一个切片中添加相同类型数据集，返回值为切片
append(切片,数据集...)
```

```
func main() {
    slice := []int{1, 2, 3, 4, 5}
    fmt.Println(slice)
    //向切片中添加三个数据元素
    slice = append(slice, 6, 6, 6)
    fmt.Println(slice)
}
```

结果

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

2.3 切片截取

所谓截取就是从切片中获取指定的数据。

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

s[n]

```
slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
//slice[n]  n为下标
fmt.Println(slice[5])
```

结果

```
6
```

s[:]

```
slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
//默认截取 截取后长度容量相同
s1 := slice[:]
fmt.Println(s1)
fmt.Println("长度：", len(s1), "容量：", cap(s1))
```

结果

```
[1 2 3 4 5 6 7 8 9 10]
长度： 10 容量： 10
```

slice[low:]

```
slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
//设置截取的起始索引
s2 := slice[3:]
fmt.Println(s2)
fmt.Println("长度：", len(s2), "容量：", cap(s2))
```

结果

```
[4 5 6 7 8 9 10]
长度： 7 容量： 7
```

slice[:high]

```
slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
//设置截取的结束索引  左闭右开
s3 := slice[:5]
fmt.Println(s3)
fmt.Println("长度：", len(s3), "容量：", cap(s3))
```

结果

```
[1 2 3 4 5]
长度： 5 容量： 10
```

slice[low:high]

```
slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
//设置截取的起始索引和结束索引  左闭右开
s4 := slice[3:5]
fmt.Println(s4)
fmt.Println("长度：", len(s4), "容量：", cap(s4))
```

结果

```
[4 5]
长度： 2 容量： 7
```

slice[low:high:max]

```
slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
//设置截取的起始索引和结束索引并指定容量
//容量：cap=max-low
s5 := slice[3:5:5]
fmt.Println(s5)
fmt.Println("长度：", len(s5), "容量：", cap(s5))
```

结果

```
[4 5]
长度： 2 容量： 2
```

注意：切片截取后和源切片在同一个地址段，修改一个会影响另外一个。

2.4 切片拷贝

Go语言的内置函数 copy() 可以将一个切片复制到另一个切片。

如果加入的两个数组切片不一样大，就会按其中较小的那个数组切片的元素个数进行复制。

```
//将一个切片复制到另一个切片
copy(目标切片,源切片)
```

```
slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
//创建一个长度为5的元素大小的切片
s:=make([]int,5)
//切片拷贝 会根据较小的切片进行拷贝
copy(s,slice)
fmt.Println(s)
```

结果

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

复制后的切片和源切片为不同切片，即内存地址不同。修改一个不会影响另外一个。

```
src := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
//创建一个长度为10的元素大小的切片
dst:=make([]int,10)
//切片拷贝
copy(dst,src)
//打印切片地址
fmt.Printf("源切片：%p\n",src)
fmt.Printf("目标切片：%p\n",dst)
```

结果

```
源切片：0xc04207c050
目标切片：0xc04207c0a0
```

2.5 切片删除

Go语言并没有对删除切片提供相对应的函数，需要使用切片本身的特性来删除元素。

```
func main() {
    slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    fmt.Printf("删除前  长度：%d 容量：%d\n", len(slice), cap(slice))
    //删除下标2到5的元素 左闭右开 不包含下标5的元素
    //需要使用不定参格式
    slice = append(slice[:2], slice[5:]...)
    fmt.Println(slice)
    fmt.Printf("删除后  长度：%d 容量：%d\n", len(slice), cap(slice))
}
```

结果

```
删除前  长度：10 容量：10
[1 2 6 7 8 9 10]
删除后  长度：7 容量：10
```

切片删除的本质：以被删除的元素为起点，到删除的元素为终点，将前后两部分内存重新连接起来。

注意

如果切片元素过多，整个删除过程非常消耗性能。

> 如果数据有频繁的删除操作，建议换其他数据存储容器。

2.6 切片作为函数参数

> 切片也可以作为函数参数 ，作为参数为引用类型。
>
> 形参可以间接修改实参的值。

```
//切片作为函数参数
func Demo(slice []int) {
    //修改切片元素的值
    slice[0] = 123
    fmt.Println(slice)
    fmt.Printf("%p\n",slice)
}
func main() {
    slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    //函数调用 地址传递 形参可以修改实参的值
    Demo(slice)
    //主调函数中切片元素的值被修改
    fmt.Println(slice)
    fmt.Printf("%p\n",slice)
}
```

结果

```
[123 2 3 4 5 6 7 8 9 10]
0xc04200e230
[123 2 3 4 5 6 7 8 9 10]
0xc04200e230
```

> 形参可实参为相同内存地址，修改一个会影响另外一个。

> **在GO语言中，数组作为参数进行传递是值传递，而切片作为参数进行传递是引用传递。**
>
> 什么是值传递？什么是引用传递？
>
> 值传递：方法调用时，实参数把它的值传递给对应的形式参数，方法执行中形式参数值的改变不影响实际参数的值。
>
> 引用传递：也称为传地址。函数调用时，实际参数的引用(地址，而不是参数的值)被传递给函数中相对应的形式参数（实参与形参指向了同一块存储区域），在函数执行中，对形式参数的操作实际上就是对实际参数的操作，方法执行中形式参数值的改变将会影响实际参数的值。

> **示例代码**

```
//切片实现冒泡排序
func BubbleSort(slice []int) {
    for i := 0; i < len(slice)-1; i++ {
        for j := 0; j < len(slice)-1-i; j++ {
            if slice[j] > slice[j+1] {
                slice[j], slice[j+1] = slice[j+1], slice[j]
            }
        }
    }
}
func main() {
    slice := []int{9, 1, 5, 6, 10, 8, 3, 7, 2, 4}
```

```
    BubbleSort(slice)
 //主调函数中实参的值被修改
    fmt.Println(slice)
}
```

总结：建议在开发中使用切片代替数组。

3 字典

Go语言字典和数组切片一样，是来保存一组相同的数据类型的。

Go语言字典可以通过key键来获取value值，map为映射关系容器，采用散列（hash）实现。

```
var 字典 map[键类型]值类型
```

定义字典结构使用map关键字，[ ]中指定的是键（key）的类型，后面紧跟着的是值（value）的类型。

3.1 字典定义

在定义map时需要注意：键的类型，必须是支持 == 和 != 操作符的类型。

切片、函数以及包含切片的结构类型不能作为字典的键，使用这些类型会造成编译错误。

```
//map在定义时不允许可以重复
var m map[int]string = map[int]string{1001: "刘能", 1002: "赵四", 1008: "谢广
坤"}
```

注意：字典中不能使用cap( )函数，只能使用len( )函数。

len( )函数返回map拥有的键值对的数量。

```
//计算字典中键值对个数
len(字典)
```

字典遍历

```
var m map[int]string = map[int]string{1001: "刘能", 1002: "赵四", 1008: "谢广
坤"}
//遍历一个字典结构数据 返回值两个 key 和 value
for k, v := range m {
 //map存储是无序的
 fmt.Println(k, v)
}
```

map是无序的，无法决定返回顺序，所以每次打印结果的顺序有可能不同。

3.2 字典中的值

可以直接通过键输出，如下所示：

```
var m map[int]string = map[int]string{1001: "刘能", 1002: "赵四", 1008: "谢广坤"}
m[1020] = "王老七"              //添加
m[1002] = "亚洲舞王赵四"         //修改
//根据键打印值
fmt.Println(m[1001])
fmt.Println(m[1002])
fmt.Println(m[1008])
```

在输出的时候，还可以进行判断：

```
var m map[int]string = map[int]string{1001: "刘能", 1002: "赵四", 1008: "谢广坤"}
//值，存在 := 字典[键]
//如果map中的key存在  返回值 value表示具体key对应的值  okbool类型 表示表示key存在条件
value,ok := m[1002]
if ok{
 fmt.Println(value)
}else{
 fmt.Println("key不存在")
}
```

3.3 删除元素

据map中的键，删除对应的元素，也是非常的方便。

```
//据map中的键，删除对应的元素
delete(字典结构， 键)
```

示例代码

```
var m map[int]string = map[int]string{1001: "刘能", 1002: "赵四", 1008: "谢广坤"}
fmt.Println("删除前：")
fmt.Println(m)
//删除操作
delete(m,1008)
fmt.Println("删除后：")
fmt.Println(m)
```

结果

```
删除前：
map[1001:刘能 1002:赵四 1008:谢广坤]
删除后：
map[1001:刘能 1002:赵四]
```

3.4 字典作为函数参数

字典也可以作为函数参数 ，作为参数为地址传递，或叫引用传递。

形参可以间接修改实参的值。

```go
package main

import "fmt"

func test(m map[int]string) {

    //修改map中的value
    m[1001] = "伊泽瑞尔"
    //删除map的数据
    delete(m, 1002)
    //添加数据
    m[1008] = "亚索"

    fmt.Printf("%p\n", m)
    //fmt.Println(m)
}
func main() {
    m := make(map[int]string)
    m[1001] = "劫"
    m[1002] = "盖伦"
    m[1005] = "薇恩"

    //map作为函数参数是值传递（包含地址）  形参可以改变实参的值
    test(m)
    fmt.Printf("%p\n", m)

    fmt.Println(m)
}
```

4 结构体

Go 语言中数组切片可以存储同一类型的数据，但在结构体中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不同类型的数据构成的数据集合。

```
type 结构体名 struct {
结构体成员名 数据类型
结构体成员名 数据类型
结构体成员名 数据类型
}
```

注意：结构体成员在定义时不能赋值。

有一个需求，要求存储学生的详细信息，例如，学生的学号，学生的姓名，年龄，家庭住址等。

可以以如下的方式进行存储：

```go
//结构体定义
type Student struct {
    id    int       //学号
    name  string    //姓名
    age   uint8     //年龄
    sex   rune      //性别
    score float32   //成绩
    addr  string    //住址
}
```

4.1结构体初始化

结构体是一种数据类型，在定义结构体后，需要定义结构体变量

```
//结构体变量
var 结构体变量名 结构体数据类型
```

在定义结构体变量后可以为结构体成员进行初始化

```
var stu Student = Student{1001, "刘能", 47, '男', 100, "象牙山"}
```

结构体定义完成后，结构体成员的使用。

```
//定义结构体变量
var stu Student
//结构体变量名.成员名 = 值
stu.id = 1001
stu.name = "刘能"
stu.sex = '男'
stu.age = 47
stu.score = 100
stu.addr = "象牙山"
```

4.2 结构体切片

定义结构体变量只能存储一条信息，如果使用结构体存储多个信息，需要使用结构体数组或切片。

```
//结构体切片
var 结构体切片名 []结构体数据类型
```

```
package main

import "fmt"

type Student3 struct {
    id    int
    name  string
    score int
    addr  string
}

func main() {
    //结构体切片
    var slice []Student3 = []Student3{
        Student3{1001, "刘能", 83, "象牙山"},
        Student3{1002, "赵四", 88, "象牙山"},
        Student3{1003, "广坤", 91, "象牙山"}}

    fmt.Println(slice)

    //切片扩容
    slice = append(slice, Student3{1004, "王老七", 99, "象牙山"},
Student3{1006, "王大拿", 66, "铁岭市"})
```

```
    fmt.Println(slice)
    //遍历打印1
    for i := 0; i < len(slice); i++ {
        fmt.Println(slice[i])
    }
    //遍历打印2
    for _, v := range slice {
        fmt.Println(v)
    }

    //切片截取
    s := slice[1:4]
    fmt.Println(s)
}
```

> **总结：掌握基本操作方式，其他使用在指针和面向对象中讲解。**

5 指针

> 前面讲过存储数据的方式，可以通过变量，或者复合类型中的数组，切片，Map，结构体等进行存储。
>
> 只要将数据存储在内存中都会为其分配内存地址。内存地址使用十六进数据表示。
>
> 内存为每一个字节分配一个32位或64位的编号（与32位或者64位处理器相关）。

5.1 变量地址和指针

> 可以使用运算符 &（取地址运算符）来获取数据的内存地址。

```
//定义变量
var i int = 10
//使用格式化打印变量的内存地址
//%p是一个占位符 输出一个十六进制地址格式
fmt.Printf("%p\n", &i)
```

> 如果想将获取的地址进行保存，应该怎样做呢？
>
> 可以通过指针变量来存储，所谓的指针变量：就是用来存储任何一个值的内存地址。

```
//定义指针变量
var 指针变量名    //默认初始值为nil   指向内存地址编号为0的空间
var 指针变量名 *数据类型 = &变量
```

```
func main() {
    var i int = 10
    //指针类型变量
    //指针变量也是变量   指针变量指向了变量的内存地址
    //对变量取地址 将结果赋值给指针变量
    var p *int = &i
 //打印指针变量p的值 同时也是i的地址
    fmt.Println(p)
}
```

5.2 指针的使用

> 在定义指针变量指向数据的内存地址后，可以通过指针间接操作数据。

需要使用运算符 *（取值运算符）操作数据。

指针修改变量的条件

- 两个变量 指针变量 普通变量
- 两个变量建立关系
- 通过*指针变量修改变量的值

```
    var a int = 10

    //var p *int = &a
    //通过自动推导类型创建指针变量
    p := &a

    //通过指针间接修改变量的值
    *p = 123
    fmt.Println(a)
```

在使用指针变量时，要注意定义指针默认值为nil。

如果直接操作指向nil的内存地址或报错。

```
//定义指针变量 默认值为nil
var p *int
*p = 123
```

```
panic: runtime error: invalid memory address or nil pointer dereference
```

所以，在使用指针变量时，一定要让指针变量有正确的指向。

5.3 new( )函数

指针变量，除了有正确指向，还可以通过new( )函数来指向。

具体的应用方式如下：

```
//根据数据类型 创建内存空间 返回值为数据类型对应的指针
new(数据类型)
```

new( )函数的作用就是动态分配空间，不需要关心该空间的释放，Go语言会自动释放。

```
func main() {
    var p *int
    //创建一个int大小的内存空间 返回值为*int
    p = new(int)
    *p = 123
    //打印值
    fmt.Println(*p)
    //打印地址
    fmt.Println(p)
}
```

5.4 指针做函数参数

> 指针也可以作为函数参数，那么指针作为函数参数在进行传递的时候，是值传递还是引用传递呢？
>
> 普通变量作为函数参数进行传递是值传递，如下所示：

```
func swap(a, b int) {
    //交换变量的值
    a, b = b, a
    fmt.Println("swap函数中值：", a, b)
}
func main() {
    a := 10
    b := 20
    //值传递 形参不能修改实参的值
    swap(a, b)
    fmt.Println("main函数中值：", a, b)
}
```

结果

```
swap函数中值：  20 10
main函数中值：  10 20
```

> 改为指针作为函数参数，如下所示:

```
func swap(a, b *int) {
    //指针间接 交换变量的值
    *a, *b = *b, *a
    fmt.Println("swap函数中值：", *a, *b)
}
func main() {
    a := 10
    b := 20
    //将变量 a b 的地址最为函数参数
    //指针变量 形参可以间接修改实参的值
    swap(&a, &b)
    fmt.Println("main函数中值：", a, b)
}
```

结果

```
swap函数中值：  20 10
main函数中值：  20 10
```

第十一章 面向对象

> 面向对象的三大基本特征：
>
> - 封装：隐藏对象的属性和实现细节，仅对外提供公共访问方式
> - 继承：使得子类具有父类的属性和方法或者重新定义、追加属性和方法等
> - 多态：不同对象中同种行为的不同实现方式

1 匿名字段

2 方法

2.1 方法定义

2.2 方法继承

2.3 方法重写

2.4 方法集

3 接口

3.1 接口定义和使用

3.2 空接口

3.3 类型断言