# 一．数据类型

## 1.布尔型

布尔型的值只可以是常量 true 或者 false。一个简单的例子：**var b bool = true**。

## 数字类型

整型 int 和浮点型 float，Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码。

## 3.字符串类型:

字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来的。Go语言的字符串的字节使用UTF-8编码标识Unicode文本。

## 4.派生类型:

包括：

(a) 指针类型（Pointer）

(b) 数组类型

(c) 结构化类型(struct)

(d) Channel 类型

(e) 函数类型

(f) 切片类型

(g) 接口类型（interface）

(h) Map 类型

# 二．变量声明（三种方式）

## 1.指定变量类型（如果不赋值，使用默认值）

**var v\_name v\_type**

**v\_name = value**

## 2.根据值自行判定变量类型。

**var v\_name = value**

## 3.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。

**v\_name := value**

**// 例如**

**var a int = 10**

**var b = 10**

**c : = 10**

# 三.多变量赋值

//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 (

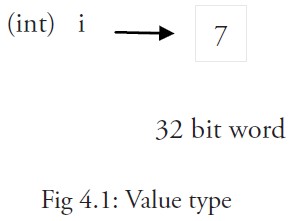
vname1 v\_type1

vname2 v\_type2

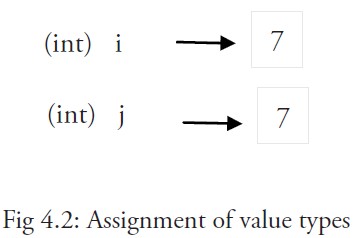
)

# 值类型和引用类型

## 1.基本数据类型：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：

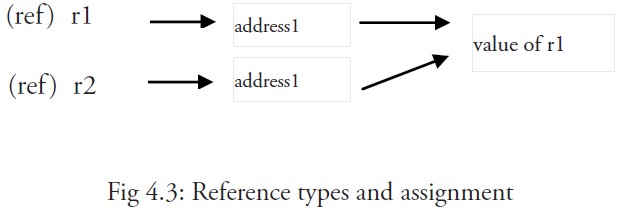


当使用等号 = 将一个变量的值赋值给另一个变量时，如：j = i，实际上是在内存中将 i 的值进行了拷贝：



## 2.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。



这个内存地址为称之为指针，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。

# 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

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，例如：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。

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

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。

但是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

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。

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)。

# 常量

## 1.常量的定义格式：

## **const identifier [type] = value**

## 2.你可以省略类型说明符 [type]，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**

## 3.常量还可以用作枚举：

**const (**

**Unknown = 0**

**Female = 1**

**Male = 2**

**)**

## 4.iota，特殊常量，可以认为是一个可以被编译器修改的常量。

iota

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

)

## 5.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。

# 运算符

## 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

## 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

## 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

## 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

## 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

## 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

& 返回变量存储地址 &a; 将给出变量的实际地址。

\* 指针变量。 \*a; 是一个指针变量

## 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：

优先级 运算符

7 ^ !

6 \* / % << >> & &^

5 + - | ^

4 == != < <= >= >

3 <-

2 &&

1 ||

# 条件语句

## if判断

If a>b{

...

}else if a>c{

...

}else {

...

}

## 2.switch 语句

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。

## 3.type Switch类型判断

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);

}

# 循环语句

## 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)

}

## 2.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。

Goto实例：

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

# go函数与方法--函数独立于类，方法必须是类的函数（参数中有对象）

Go 语言函数方法

Go 函数Go 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"

)

/\* 定义函数 \*/

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

# 变量作用域（与其它语言类似）

## 1.注意：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

## 注意：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

# 数组

声明：

**var balance [10] float32**

初始化：

**var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}** 指定了size就不能超

**var balance = [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}** 不指定会自动判断

根据索引取值，或者附值



示例：

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] )

}

}

# Go语言指针（&-拿变量的地址，\*-拿指针变量对应的值。可以理解为相反的操作）

## 1.获取一个变量的地址 &

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

## 2.什么是指针

一个指针变量可以指向任何一个值的内存地址它指向那个值的内存地址。

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。

## 指针数组

ptr 为整型指针数组。因此每个元素都指向了一个值。以下实例的三个整数将存储在指针数组中：

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] /\* 整数地址赋值给指针数组 \*/

}

for i = 0; i < MAX; i++ {

fmt.Printf("a[%d] = %d\n", i,\*ptr[i] )

}

}

以上代码执行输出结果为：

a[0] = 10

a[1] = 100

a[2] = 200

## 指向指针变量的指针

访问指向指针的指针变量值需要使用两个 \* 号，如下所示：

package main

import "fmt"

func main() {

var a int

var ptr \*int

var pptr \*\*int

a = 3000

/\* 指针 ptr 地址 \*/

ptr = &a

/\* 指向指针 ptr 地址 \*/

pptr = &ptr

/\* 获取 pptr 的值 \*/

fmt.Printf("变量 a = %d\n", a )

fmt.Printf("指针变量 \*ptr = %d\n", \*ptr )

fmt.Printf("指向指针的指针变量 \*\*pptr = %d\n", \*\*pptr)

}

以上实例执行输出结果为：

变量 a = 3000

指针变量 \*ptr = 3000

指向指针的指针变量 \*\*pptr = 3000

## 指针作为参数

Go 语言指针作为函数参数

Go 指针Go 指针

Go 语言允许向函数传递指针，只需要在函数定义的参数上设置为指针类型即可。

以下实例演示了如何向函数传递指针，并在函数调用后修改函数内的值，：

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 )

/\* 调用函数用于交换值

\* &a 指向 a 变量的地址

\* &b 指向 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 地址的值 \*/

\*x = \*y /\* 将 y 赋值给 x \*/

\*y = temp /\* 将 temp 赋值给 y \*/

}

以上实例允许输出结果为：

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

# 结构体（类似java中的类的概念）

## 定义与访问结构体

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 = "www.runoob.com"

Book1.subject = "Go 语言教程"

Book1.book\_id = 6495407

/\* book 2 描述 \*/

Book2.title = "Python 教程"

Book2.author = "www.runoob.com"

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 : www.runoob.com

Book 1 subject : Go 语言教程

Book 1 book\_id : 6495407

Book 2 title : Python 教程

Book 2 author : www.runoob.com

Book 2 subject : Python 语言教程

Book 2 book\_id : 6495700

## 结构体指针相关操作

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 = "www.runoob.com"

Book1.subject = "Go 语言教程"

Book1.book\_id = 6495407

/\* book 2 描述 \*/

Book2.title = "Python 教程"

Book2.author = "www.runoob.com"

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 : www.runoob.com

Book subject : Go 语言教程

Book book\_id : 6495407

Book title : Python 教程

Book author : www.runoob.com

Book subject : Python 语言教程

Book book\_id : 6495700

# 切片（也叫动态数组）

## 定义

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。

**注意：**

**定义切片：**

**1）make ( []Type ,length, capacity )**

**2) make ( []Type, length)**

**3) []Type{}**

**4) []Type{value1 , value2 , ... , valueN }**

**从3)、4)可见，创建切片跟创建数组唯一的区别在于 Type 前的“ [] ”中是否有数字，为空，则代表切片，否则则代表数组。因为切片是长度可变的。**

## 2.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

## 3.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]

## 4.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

## 5.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]

## 6.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]

# RANGE

## 操作数组与切片（返回多值 ，序号与值）

nums := []int{2, 3, 4}

sum := 0

for \_, num := range nums {

sum += num

}

fmt.Println("sum:", sum)

for i, num := range nums {

if num == 3 {

fmt.Println("index:", i)

}

}

## 操作map 返回key 与value

kvs := map[string]string{"a": "apple", "b": "banana"}

for k, v := range kvs {

fmt.Printf("%s -> %s\n", k, v)

}

## 操作字符串 (返回索引与Unicode值)

for i, c := range "go" {

fmt.Println(i, c)

}

# Map集合

## 定义两种方式

其一：

声明变量，默认map是nil

**Var myMap map[key\_type]valuetype**

其二：

**myMap:=make(map[key\_type]value\_type)**

## 附值与取值与删除

**var countryCapitalMap map[string]string**

**countryCapitalMap["France"] = "Paris"**

取值：

**fmt.Println("Capital of",country,"is",countryCapitalMap[country])**

删除：

**delete(countryCapitalMap,"France");**

# 类型转换--类型名（要转换的变量）

var sum int = 17

var count int = 5

var mean float32

mean = float32(sum)/float32(count)

# 接口与实现--(只要实现了接口中的所有方法就算是实现了该接口，不必像java那样implents 接口，来实现某接口)

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。

package main

import (

"fmt"

)

type Phone interface {

call()

}

type NokiaPhone struct {

}

func (nokiaPhone NokiaPhone) call() {

fmt.Println("I am Nokia, I can call you!")

}

type IPhone struct {

}

func (iPhone IPhone) call() {

fmt.Println("I am iPhone, I can call you!")

}

func main() {

var phone Phone

phone = new(NokiaPhone)

phone.call()

phone = new(IPhone)

phone.call()

}

在上面的例子中，我们定义了一个接口Phone，接口里面有一个方法call()。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!

# 二十.错误处理

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

}

# 二十一.概念知识点--new()与make()的区别

new 的作用是 初始化 一个指向类型的指针 (\*T)， make 的作用是为 slice, map 或者 channel 初始化，并且返回引用 T

make(T, args)函数的目的与new(T)不同。它仅仅用于创建 Slice, Map 和 Channel，并且返回类型是 T（不是T\*）的一个初始化的（不是零值）的实例。