# 《菜鸟教程》

## 一、GO语言简介

1. GO语言的特色

* 简洁、快速、安全
* 并行、有趣、开源
* 内存管理、v数组安全、编译迅速

2. Go语言用途

Go语言被设计成一门用于搭建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语言无疑比其他大多数语言有更高的开发效率。它提供了海量并行的支持，这对于游戏服务器端的开发而言是再好不过的。

## 二、Go语言的结构

package main

import "fmt"

func main() {

/\* 这是我的第一个简单的程序 \*/

fmt.Println("Hello, World!")

}

1. Go语言的基础组成由以下几个部分：

* 包声明(package)：必须在源文件中非注释第一行声明这个文件属于哪个包，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
* 引入包(import)：告诉编译器这个程序需要哪个包
* 函数：func main()是程序开始执行的函数，如果有init()函数会先执行init
* 变量：当标识符（常亮、变量、类型、函数名、结构字段等等）以一个大写字母开头，那么使用这种形式的标识符对象就可以被外部
* 语句 & 表达式
* 注释： /\* \*/ 或者 //

1. 行分隔符： 一行代表一个语句结束；若将多个语句写在同一行，需要`;`来分割（不推荐）
2. 标识符：一个或多个字母(A~Z、a~z)、数字(0~9)、下划线(\_)组成，第一个字符不能是数字。

## 三、Go语言数据类型

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 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。 |
| 2 | **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:**  (a) 指针类型（Pointer）  (b) 数组类型  (c) 结构化类型(struct)  (d) 联合体类型 (union)  (e) 函数类型  (f) 切片类型  (g) 接口类型（interface）  (h) Map 类型  (i) Channel 类型 |

1. 数字类型：Go语言也有基于架构的类型，如int、unit和unitptr

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **uint8：**无符号 8 位整型 (0 到 255) |
| 2 | **uint16：**无符号 16 位整型 (0 到 65535) |
| 3 | **uint32：**无符号 32 位整型 (0 到 4294967295) |
| 4 | **uint64：**无符号 64 位整型 (0 到 18446744073709551615) |
| 5 | **int8：**有符号 8 位整型 (-128 到 127) |
| 6 | **int16：**有符号 16 位整型 (-32768 到 32767) |
| 7 | **int32：**有符号 32 位整型 (-2147483648 到 2147483647) |
| 8 | **int64：**有符号 64 位整型 (-9223372036854775808 到 9223372036854775807) |

1. 浮点类型

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **float32：**IEEE-754 32位浮点型数 |
| 2 | **float64：**IEEE-754 64位浮点型数 |
| 3 | **complex64：**32 位实数和虚数 |
| 4 | **complex128：**64 位实数和虚数 |

1. 其他数值类型

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **byte：**类似 uint8 |
| 2 | **rune：**类似 int32 |
| 3 | **uint：**32 或 64 位 |
| 4 | **int：**与 uint 一样大小 |
| 5 | **uintptr：**无符号整型，用于存放一个指针 |

## 四、Go语言变量

Go语言变量名由字母、数字、下划线组成，其中首字符不能为数字.

声明的一般形式如下：

var identifier type

1. 指定变量类型，声明后若不赋值，使用默认值

var v\_name v\_type

v\_name = value

1. 根据值，自行判定变量类型

var v\_name = value

1. 省略var，注意 :=左侧的变量不能为已声明过的，否则会编译错误

v\_name := value

// 例如

var a int = 10

var b = 10

c : = 10

1. 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 //并行赋值

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，**这种不带声明的格式只能在函数体中出现**

// 这种因式分解关键字的写法一般用于声明全局变量

var ( // 这是圆括号

vname1 v\_type1

vname2 v\_type2

)

1. 注意事项

在相同的代码块中，不能再次对相同名称的变量使用初始化声明，例如 a:=20 是不允许的，编译器会提示：no new variables on left side of :=

如果在定义变量之前使用，则会得到编译错误： undefined:a.

如果声明了一个局部变量却没有在相同的代码块中使用它，则会得到：`variable` declared and not used. 全局变量可以声明但不使用

空白标记符 `\_` 也可用于被抛弃的值，如5在 \_, b=5,7 中被抛弃。

## 五、Go语言常量

常量中的数据类型只可以是布尔型、数字型和字符串型

1. 定义格式如下（可以省略 type）：

const identifier [type] = value

1. 常量还可以用作枚举： 常量表达式中可以使用内置函数

const (

Unknown = 0

Female = 1

Male = unsafe.Sizeof(a)

)

1. iota：特殊常量，可以被编译器修改的常量。每个const关键字出现时，被重置为0，下次const出现之前，每出现一次iota，其所代表的数字就会自动加1.

const (

a = iota

b = iota

c = iota

)

上述iota还可以简写为

const (

a = iota

b

c

)

## 六、 Go语言运算符

Go语言的运算符有：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

1. 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 |

1. 关系运算符

|  |  |  |
| --- | --- | --- |
| **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 |

1. 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 |

1. 位运算符： &(与) |(或) ^(亦或)

|  |  |  |
| --- | --- | --- |
| **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，">>"右边的数指定移动的位数。 | A >> 2 结果为 15 ，二进制为 0000 1111 |

1. 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 |

1. 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 |

1. 运算符优先级：有些运算符有较高的优先级，二元运算符的运算方向均是从左向右。

|  |  |
| --- | --- |
| **优先级** | **运算符** |
| 7 | ^, ! |
| 6 | \*, /, %, <<, >>, &, &^ |
| 5 | + - | ^ |
| 4 | ==, !=, <, <=, >=, > |
| 3 | <,- |
| 2 | && |
| 1 | || |

## 七、 条件语句

|  |  |
| --- | --- |
| **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。 |

1. if语句：花括号不能省略，否则编译不能通过

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

}

1. if...else语法如下

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

} else {

/\* 在布尔表达式为 false 时执行 \*/

}

1. 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，匹配项后不用再加break；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

switch var1 {

case val1:

...

case val2:

...

default:

...

}

switch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。

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);

}

1. select语句：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受。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可以运行。

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);

}

## 八、循环控制语句

| **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 |

1. for循环有三种形式

for init; condition; post { } /\* 同C语言的for \*/

for condition { } /\* 同C语言的while \*/

for { } /\*同C语言的for(;;){} \*/

1. for循环的range格式

for key, value := range oldMap {

newMap[key] = value

}

1. 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。 |

## 九、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**最少有一个main()函数**。

1. 函数的定义

func function\_name( [parameter list] ) [return\_types] {

函数体

}

1. 函数可以定义多个返回值

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

return y, x

}

1. 函数闭包：匿名函数是一个“内联”语句或表达式，可以作为闭包。匿名函数的优越性在于可以直接使用函数内的变量。

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()

fmt.Println(nextNumber())

fmt.Println(nextNumber())

}

1. 函数方法：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是一个指针。

func (variable\_name variable\_data\_type) function\_name() [return\_type]{

/\* 函数体\*/

}

/\* 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

}

## 十、Go语言变量作用域

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：它的作用域只在函数体内，参数和返回值变量也是局部变量
* 函数外定义的变量成为全局变量：全局变量可以在这个包甚至外部包（被导出后）中使用。全局变量和局部变量名称可以相同，函数内的局部变量会覆盖全局变量
* 函数定义中的变量成为形式参数

不同类型的局部和全局变量默认值不同：

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

## 十一、 Go语言数组

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、字符串或者自定义类型。

1. 声明数组：需要制定元素类型以及元素个数，语法如下：

var variable\_name [SIZE] variable\_type

1. 初始化数组：{}的元素个数不能大于[]中的数字；如果忽略[]中的数字，则数组大小为元素的个数

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

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

1. 多维数组

var variable\_name [SIZE1][SIZE2]...[SIZEN] variable\_type

var arrayName [ x ][ y ] variable\_type /\* 二维数组 \*/

a = [3][4]int{

{0, 1, 2, 3} , /\* 第一行索引为 0 \*/

{4, 5, 6, 7} , /\* 第二行索引为 1 \*/

{8, 9, 10, 11} /\* 第三行索引为 2 \*/

}

1. 向函数传递数组参数，形参的两种方式

void myFunction(param [10]int) {

/\* 函数体 \*/

}

void myFunction(param []int){

/\* 函数体 \*/

}

## 十二、Go语言指针

1. 指针变量可以指向任何一个值的内存地址。声明格式如下：

var var\_name \*var-type

var ip \*int /\* 指向整型\*/

var fp \*float32 /\* 指向浮点型 \*/

1. 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\*号来获取指针所指向的内容。

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 )

fmt.Printf("\*ip 变量的值: %d\n", \*ip )

1. 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。

if(ptr != nil) /\* ptr 不是空指针 \*/

if(ptr == nil) /\* ptr 是空指针 \*/

1. 指针数组，声明格式如下

var ptr [MAX]\*int;

1. 指向指针的指针，声明格式如下：

var ptr \*\*int;

1. 把指针作为函数参数

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 \*/

}

## 十三、Go语言结构体

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。

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。结构体中有一个或多个成员，type设定了结构体的名称

type struct\_variable\_type struct {

member definition;

member definition;

...

member definition;

}

1. 变量声明：

variable\_name := structure\_variable\_type {value1, value2...valuen}

1. 访问结构体成员：如果要访问结构体成员，需要使用点号(.)，格式为”结构体.成员名”

type Books struct {

title string

author string

subject string

book\_id int

}

var Book1 Books /\* 声明 Book1 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

1. 结构体指针：存储结构体变量的地址，使用 “&”符号访问结构体变量地址，使用结构体指针访问结构体成员，使用 “.”

var struct\_pointer \*Books

struct\_pointer = &Book1;

struct\_pointer.title;

## 十四、Go语言切片

切片是对数组的抽象。数组长度不可变，在特定场景中这样的集合就不太适用。Go提供了一种灵活，功能强大的内置类型切片（“动态数组”），切片的长度是不固定的，可以追加元素，追加元素时可能使切片的容量增大。

1. 定义切片：声明一个未指定大小的数组来定义切片；使用make()函数来创建切片

var identifier []type

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

也可以简写为

slice1 := make([]type, len)

指定容量，capacity为可选参数

make([]T, length, capacity)

1. 切片初始化：

/\* 直接初始化切片 \*/

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

/\* 初始化切片s，用数组arr引用 \*/

s := arr[:]

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

s := arr[startIndex:endIndex]

/\* 缺省endIndex时表示一直到arr的最后一个元素 \*/

s := arr[startIndex:]

/\* 缺省startIndex时表示从arr的第一元素开始 \*/

s := arr[:endIndex]

/\* 通过切片s初始化切片s1 \*/

s1 := s[startIndex:endIndex]

/\* 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 \*/

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

1. 相关函数： len()获取长度，cap()获取切片最长可以达到多少
2. 空(nil)切片：一个切片在未初始化之前默认为nil，长度为0.
3. 切片截取：通过设置下限及上线来设置截取切片[lower-bound:upper-bound]，**对切片进行操作后，会影响原来的值**

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

/\* 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])

1. append()和copy()函数：append可以在容量不够时，扩容增加容量；copy可以将一个切片拷贝到另一个切片

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)

## 十五、Go语言范围

range关键字用于for循环中迭代数组、切片、链表或集合的元素。在数组和切片中它返回元素的索引值和元素值，在集合中返回key-value对的key值

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)

}

}

## 十六、Go语言Map(集合)

Map是一种无序的键值对的集合，使用hash表来实现，无法确定其返回元素的顺序

1. 定义Map，若不初始化map，那么会创建一个nil map，不能用来存放键值

/\* 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)

1. 使用map

/\* 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"

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

1. delete()函数：用于删除集合元素，参数为map和其对应的key

/\* 删除元素 \*/

delete(countryCapitalMap,"France");

## 十七、其他

1. Go语言支持递归调用函数
2. Go语言类型转换

type\_name(expression)

var sum int = 17

var count int = 5

var mean float32

mean = float32(sum)/float32(count)

1. Go语言接口：将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\*/

}

1. 错误处理：通过内置的错误接口提供了非常简单的错误处理机制
2. error类型是一个接口，定义如下：

type error interface {

Error() string

}

1. 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

}