

Go 102

Pallat Anchaleechamaikorn

Technical Coach

Infinitas by KrungThai

Arise by Infinitas

yod.pallat@gmail.com

<https://github.com/pallat>

<https://dev.to/pallat>

<https://go.dev/tour> (Thai)

<https://github.com/uber-go/guide> (Thai)

Data Structure

Array

```
var name [n]T
```

```
var array [5]int
```

Array: assign/access

```
var array [5]int
```

```
array[0] = 1
```

```
array[1] = 2
```

```
elem := array[4]
```

How to loop through an Array

```
fibonacci := [10]int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}  
  
for i:= 0; i < 10; i ++ {  
    fmt.Println(fibonacci[i])  
}
```

using len to get the length

```
fibonacci := [10]int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}  
  
for i:= 0; i < len(fibonacci); i ++ {  
    fmt.Println(fibonacci[i])  
}
```

range: to iterate over item

```
fibonacci := [10]int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}

for i, v := range fibonacci {
    fmt.Printf("index: %d, value %d\n", i, v)
}
```


range: `_` to ignore not used value

```
fibonacci := [10]int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}

for i, _ := range fibonacci {
    fmt.Printf("only index is %d\n", i)
}
```

```
for i := range fibonacci {
    fmt.Printf("only index is %d\n", i)
}
```

```
for _, v := range fibonacci {
    fmt.Printf("only value is %d\n", v)
}
```

Exercise: Array

Reverse Elements

```
list := [4]int{1, 3, 4, 2}  
r := reverse(list)  
  
fmt.Printf("%T %v\n", r, r)
```

[4]int [2 4 3 1]

Array auto counting

```
fibonacci := [...]int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}

fmt.Printf("type: %T\n", fibonacci)

for i, v := range fibonacci {
    fmt.Printf("index: %d, value %d\n", i, v)
}
```

type: [10]int

How to make it works

```
input1 := [...]int{-1, 0, 1, -2, 2}
input2 := [...]int{2, 0, -2}

r1 := reduceSum(input1)
r2 := reduceSum(input2)
```

Slice

An array has a fixed size.

A slice, on the other hand

Slice literals

var name []T

```
var array = [3]int{-1, 0, 1}  
var slice = []int{-1, 0, 1}
```

Try to do it again

```
input1 := []int{-1, 0, 1, -2, 2}  
input1 := []int{2, 0, -2}  
  
r1 := reduceSum(input1)  
r2 := reduceSum(input2)
```


more flexible than array

```
fibonacci := []int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}  
  
for i, v := range fibonacci {  
    fmt.Println(fibonacci[i])  
}
```

append

```
func append(s []T, vs ...T) []T
```

```
fibonacci := []int{0, 1, 1, 2, 3, 5}  
fibonacci = append(fibonacci, 8, 13, 21, 34)
```

```
fibonacci = []int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}
```

slice

a[low : high]

```
fibonacci := []int{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}  
fibonacci = fibonacci[1:7]
```

fibonacci = []int{1, 1, 2, 3, 5, 8}

Test Slice

```
a := [...]int{-1, 0, 1, 2, 3, 4, 5, 6}  
s := a[0:5]
```

Test Slice

```
a := [...]int{-1, 0, 1, 2, 3, 4, 5, 6}  
s := a[0:5]
```

```
[]int{-1, 0, 1, 2, 3}
```

ignore the low bound if zero

```
a := [...]int{-1, 0, 1, 2, 3, 4, 5, 6}  
s := a[:5]
```

```
[]int{-1, 0, 1, 2, 3}
```

ignore the high bound for the last one

```
a := [...]int{-1, 0, 1, 2, 3, 4, 5, 6}  
s := a[2:]
```

```
[]int{1, 2, 3, 4, 5, 6}
```

Exercise: couple

"abcdef" -> []string{"ab","cd","ef"}

"abcdefg" -> []string{"ab","cd","ef","g*"}

2 dimension slice

```
func paint(dx, dy int) [][]int{}
```

pixel := x * y

```
paint(3, 3)
[][]int{
    []int{1, 2, 3},
    []int{2, 4, 6},
    []int{3, 6, 9},
}
```

Zero value of slice is **nil**

```
var s []string // nil
```

```
var s []string  
s[0] = "The"
```

make slice

make([]T, length, capacity)

```
var s []string  
s = make([]string, 1)  
s[0] = "The"
```

Structure of Slice

Make 0 length of slice

```
s := make([]string, 0)
```

make 1 length

```
s := make([]string, 1)
```

underlying of slice is an array

```
s := make([]int, 4)
```

len = 4

cap = 4

make 1 length and
3 cap

```
s := make([]string, 1, 3)
```


Make Slice with cap

```
s := make([]int, 4, 6)
```

len = 4

cap = 6

assign a value to slice

```
var i = []int{-2, -1, 0, 1, 2}
```

```
var i []int  
i = []int{-2, -1, 0, 1, 2}
```

assign an array to slice

```
var i []int
var arr = [...]int{-4, -3, -2, -1, 0, 1, 2, 3, 4}
i = arr[0:9]
```

Slice: Two-Index Slices

slice point to an array

```
var arr := [4]int{9, 8, 7, 6}  
part := arr[1:3]
```

```
part = []int{8, 7}
```

Slice: Three-Index Slices

slice point to an array

```
var arr := [4]int{9, 8, 7, 6}  
part := arr[1:3:3]
```

Test about slice

```
arr := [...]int{111, 101, 110, 123, 321}
s := arr[:]

fmt.Printf("type: %T, val: %v\n", s, s)
fmt.Printf("cap: %d, len: %d\n", cap(s), len(s))
```

type: ?, val: ?

cap: ?, len: ?

Test about slice

```
arr := [...]int{997, 111, 101, 110, 123, 321}
s := arr[1:3:4]

fmt.Printf("type: %T, val: %v\n", s, s)
fmt.Printf("cap: %d, len: %d\n", cap(s), len(s))
```

type: ?, val: ?

cap: ?, len: ?

Test about slice

```
arr := []int{997, 111, 101, 110, 123, 321}
s := arr[2:]

fmt.Printf("type: %T, val: %v\n", s, s)
fmt.Printf("cap: %d, len: %d\n", cap(s), len(s))
```

type: ?, val: ?

cap: ?, len: ?

len() & cap() with slice

```
s := make([]int, 5, 10)
fmt.Println(len(s))
fmt.Println(cap(s))
```

Exercise: moretypes/18

<https://go.dev/tour/moretypes/18>

<https://go-tour-th.appspot.com/moretypes/18>

Slice just refer to an array

```
arr := [3]int{-1, 0, 1}
p1 := arr[1:2]

fmt.Printf("value: %v\n", p1)
fmt.Printf("len(p1): %d, cap(p1): %d\n", len(p1), cap(p1))

p1[0] = 3
fmt.Printf("arr value: %v\n", arr)
```

```
value: ?
len(p1): ?, cap(p1): ?
arr value: ?
```

Slice just refer to an array

```
arr := [3]int{-1, 0, 1}
p1 := arr[1:2]

fmt.Printf("value: %v\n", p1)
fmt.Printf("len(p1): %d, cap(p1): %d\n", len(p1), cap(p1))

p1[0] = 3
fmt.Printf("arr value: %v\n", arr)
```

```
value: [0]
len(p1): 1, cap(p1): 2
arr value: [-1 3 1]
```

Slice just refer to an array(2)

append not returns a new array if not over cap

```
arr := [3]int{-1, 0, 1}
p1 := arr[1:2]

fmt.Printf("p1 value: %v\n", p1)
fmt.Printf("p1 len(p1): %d, cap(p1): %d\n", len(p1), cap(p1))

p1 = append(p1, 2)
fmt.Printf("arr value: %v\n", arr)
```

```
value: ?
len(p1): ?, cap(p1): ?
arr value: ?
```

Slice just refer to an array(2)

append not returns a new array if not over cap

```
arr := [3]int{-1, 0, 1}
p1 := arr[1:2]

fmt.Printf("p1 value: %v\n", p1)
fmt.Printf("p1 len(p1): %d, cap(p1): %d\n", len(p1), cap(p1))

p1 = append(p1, 2)
fmt.Printf("arr value: %v\n", arr)
```

```
value: 0
len(p1): 1, cap(p1): 2
arr value: [-1 0 2]
```

Slice just refer to an array(3)

```
arr := [3]int{-1, 0, 1}
p1 := arr[1:2]
p2 := arr[:]
```

```
p1 = append(p1, 8)
```

```
fmt.Printf("arr: %v\n", arr)
fmt.Printf("p1: %v\n", p1)
fmt.Printf("p2: %v\n", p2)
```

```
arr: ?
p1: ?
p2: ?
```

Slice just refer to an array(3)

```
arr := [3]int{-1, 0, 1}
p1 := arr[1:2]
p2 := arr[:]
```

```
p1 = append(p1, 8)
```

```
fmt.Printf("arr: %v\n", arr)
fmt.Printf("p1: %v\n", p1)
fmt.Printf("p2: %v\n", p2)
```

```
arr: [-1 0 8]
p1: [0 8]
p2: [-1 0 8]
```


Slice just refer to an array(4)

```
arr := [3]int{-1, 0, 1}
p1 := arr[1:2]
p2 := arr[:]
```

```
p1 = append(p1, 8, 9)
```

```
fmt.Printf("arr: %v\n", arr)
fmt.Printf("p1: %v\n", p1)
fmt.Printf("p2: %v\n", p2)
```

```
arr: ?
p1: ?
p2: ?
```

Slice just refer to an array(4)

```
arr := [3]int{-1, 0, 1}
p1 := arr[1:2]
p2 := arr[:]
```

```
p1 = append(p1, 8, 9)
```

```
fmt.Printf("arr: %v\n", arr)
fmt.Printf("p1: %v\n", p1)
fmt.Printf("p2: %v\n", p2)
```

```
arr: [-1 0 1]
p1: [0 8 9]
p2: [-1 0 1]
```


Variadic function (Variable number of arguments)

```
func variadic(nums ...int)
```

Spread operator

```
func variadic(nums ...int) {  
  
}  
  
var slice = []int{1, 3, 5, 7, 9}  
variadic(slice...)
```

Delete an element in Slice

```
s := []int{1, 2, 3, 4, 5}
s = append(s[:2], s[3:]...)
```


map[T]T

```
var m map[string]string
```


zero value of map is nil

```
var m map[string]string

if m == nil {
    fmt.Println("it's nil")
}
```

map need home

make

```
m := make(map[string]string)

if m == nil {
    fmt.Println("it's nil")
}

m["a"] = "apple"
m["b"] = "banana"
m["c"] = "coconut"
m["d"] = "durian"
m["e"] = "elderberry"
m["f"] = "fig"
m["g"] = "guava"
```

construct map

```
m := map[string]string{
    "a" : "apple",
    "b" : "banana",
    "c" : "coconut",
    "d" : "durian",
    "e" : "elderberry",
    "f" : "fig",
    "g" : "guava",
}

for k, v := range m {
    fmt.Println(k, v)
}
```

delete a key

```
m := map[string]string{
    "a" : "apple",
    "b" : "banana",
    "c" : "coconut",
    "d" : "durian",
    "e" : "elderberry",
    "f" : "fig",
    "g" : "guava",
}

delete(m, "d")

for k, v := range m {
    fmt.Println(k, v)
}
```

len() with map

```
m := map[string]string{
    "a" : "apple",
    "b" : "banana",
    "c" : "coconut",
    "d" : "durian",
    "e" : "elderberry",
    "f" : "fig",
    "g" : "guava",
}

fmt.Println(len(m))
```

Exercise: map easy

split all keys in m to keys and all values to vals

```
m := map[string]int {  
    "G": 71,  
    "O": 79,  
    "P": 80,  
    "H": 72,  
    "E": 69,  
    "R": 82,  
}  
  
keys := []string{}  
vals := []int{}
```

Exercise: map

open a file `oscar_age_male.csv`

<https://github.com/focusive/go102/tree/master/testdata>

print any actors name who got the oscar more than one time

```
Marlon Brando  
Daniel Day-Lewis  
Sean Penn  
Tom Hanks  
Fredric March  
Spencer Tracy  
Gary Cooper  
Jack Nicholson  
Dustin Hoffman
```


new type: Did you remember?

```
type char byte
var b byte = 'a'
var c char = 'a'

fmt.Println(b == byte(c))
```

```
type names []string
type nameGeneration map[string]string
```

Structure

```
type Account struct {  
    Email string  
    CreatedDate time.Time  
}
```

How to create a struct instance

```
account := Account{
    Email: "yod@gopher.com",
    CreatedDate: time.Now(),
}

fmt.Println(account.Email)
fmt.Println(account.CreatedDate)
```

Play with struct: NewAccount

```
type Account struct {  
    Email string  
    CreatedDate time.Time  
}  
  
func NewAccount(email string, CreatedDate int64) Account{}
```

CreatedDate argument is an Unix Time

Play with struct: String of Account

```
type Account struct {  
    Email string  
    CreatedDate time.Time  
}
```

```
func AccountString(account Account) string {}
```

```
AccountString(Account{Email: "yod@gopher.com", CreatedDate: 1615694700})  
"yod@gopher.com registered on 2021/03/14 04:05:00 +0000"
```




type conversion of struct

Type's can only be converted between one another if the underlying data structure is the same.

```
type A struct {  
    Name string  
    Age  int  
}  
  
type B struct {  
    Name string  
    Age  int  
}  
  
a := A{Name: "Yod", Age: 44}  
var b B  
b = B(a)  
fmt.Println(a == A(b))
```

empty struct

```
var s struct{}
```

anonymous

```
var s = struct{ Name string }{Name: "Yod"}  
fmt.Println(s.Name)
```


Tag

```
type Account struct {  
    Email      string    `json:"email"`  
    CreatedDate time.Time `json:"created_date"`  
}
```

Marshal to JSON

```
account := NewAccount("yod@gopher.com", 1615694700)
b, err := json.Marshal(&account)
if err != nil { log.Fatal(err) }
fmt.Println(string(b))
```

Play with UnMarshal

```
{  
  "email": "yod@go.dev",  
  "created_date": "14-01-2022 09:10:11"  
}
```

Method

```
var i Int = 14  
s := i.toString()
```

| s = "14"

How to make it

```
type Int int

func (i Int) toString() string{
    return strconv.Itoa(int(i))
}
```

What is the method

```
type Int int

func (i Int) toString() string{
    return strconv.Itoa(int(i))
}
```

What is the method

```
type Int int

func toString(i Int) string{
    return strconv.Itoa(int(i))
}
```

What is the method

```
type Int int

func (i Int) toString() string{
    return strconv.Itoa(int(i))
}

fmt.Printf("%T\n", Int.toString)
```

```
func(main.Int) string
```


Then we can make it like this

```
type Int int

func (i Int) toString() string{
    return strconv.Itoa(int(i))
}

var fn func(Int) string
fn = Int.toString

fmt.Println(fn(Int(9)))
```


Play with method

```
type Account struct {  
    Email string  
    CreatedDate time.Time  
}  
  
account := NewAccount(...)  
fmt.Println(account.toString())
```

Pointer Receiver

```
type String string

func (s *String) Lower() {
    *s = String(strings.ToLower(string(*s)))
}
```

```
fmt.Printf("%T\n", String.Lower)
```

Play with method: Todo

```
go
type tasks []task
type task struct {
    id string
    title string
}

type Todo struct {
    tasks tasks
}
```

Add 4 Methods

- Add(title string)
- Delete(id string)
- List() tasks
- Update(id, title string)

interface

empty interface

```
var a interface{}
```

any is an alias type of interface{}

```
var a any
```

empty interface behavior

```
var a any

a = 10
fmt.Printf("type is %T, value is %v\n", a, a)

a = "ten"
fmt.Printf("type is %T, value is %v\n", a, a)
```


What is empty interface

```
var a any

a = 10
fmt.Printf("type is %T, value is %v\n", a, a)

// at this line, what is the type of `a` instance

a = "ten"
fmt.Printf("type is %T, value is %v\n", a, a)
```

Can we do like this

```
var a any

a = 10
fmt.Printf("type is %T, value is %v\n", a, a)

var n int
n = a

a = "ten"
fmt.Printf("type is %T, value is %v\n", a, a)
```

interface structure

can be thought of as a tuple of a value and a concrete type

Type Assertion

```
var a any

a = 10
fmt.Printf("type is %T, value is %v\n", a, a)

var n int
n = a.(int)

a = "ten"
fmt.Printf("type is %T, value is %v\n", a, a)
```

Type Assertion with wrong type

```
var a any

a = 10
fmt.Printf("type is %T, value is %v\n", a, a)

a = "ten"
fmt.Printf("type is %T, value is %v\n", a, a)

var n int
n = a.(int)
```

Type Assertion the second value

```
var a any

a = 10
fmt.Printf("type is %T, value is %v\n", a, a)

a = "ten"
fmt.Printf("type is %T, value is %v\n", a, a)

if n, ok := a.(int); ok {
    fmt.Println("it's int now: %d\n", n)
}
```

type switch

```
var a any
switch v := a.(type) {
    case string:
    case int:
    default:
}
```


interface defines a set of method signatures

```
type I interface{  
    M()  
}
```

zero value of interface{} is nil

```
type I interface{
    M()
}

func main() {
    var i I
    fmt.Println(i)
}
```

Interfaces are implemented implicitly

```
type I interface {  
    M()  
}  
  
type S struct {}  
func (S) M() {  
    fmt.Println("S implements the interface I")  
}
```

```
func main() {  
    var i I  
    var s S  
    i = s  
    i.M()  
}
```

Interfaces are implemented implicitly

```
type I interface {  
    M()  
}  
  
type S struct {}  
func (S) M() {  
    fmt.Println("S implements the interface I")  
}
```

```
func main() {  
    var i I  
    var s S  
    i = s  
    i.M()  
}
```

pointer receiver

```
type I interface {  
    M()  
}  
  
type S struct {}  
func (*S) M() {  
    fmt.Println("S implements the interface I")  
}  
  
func main() {  
    var i I  
    i = &S{}  
    i.M()  
}
```

Nil interface values

```
type I interface {  
    M()  
}  
  
func main() {  
    var i I  
    i.M()  
}
```

Stringer

```
type Stringer interface {  
    String() string  
}
```

https://go.dev/doc/effective_go#interface-names

Interface: names

By convention, one-method interfaces are named by the method name plus an -er suffix or similar modification to construct an agent noun: Reader, Writer, Formatter, CloseNotifier etc.

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}  
  
type Writer interface {  
    Write(p []byte) (n int, err error)  
}
```

error type

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



Play with interface

```
func AnimalSound(specie string) string {  
    switch specie {  
        case "cat":  
            return "Meow"  
        case "dog":  
            return "Woof"  
        case "lion":  
            return "Roar"  
        case "owl":  
            return "Hoot"  
        default:  
            return "Grr"  
    }  
}  
  
type Animal interface {  
    Sound() string  
}  
  
func MakeSound(a Animal) {  
    fmt.Println(a.Sound())  
}
```

Keywords: 18/25

break
case
chan
const
continue

default
defer
else
fallthrough
for

func
go
goto
if
import

interface
map
package
range
return

select
struct
switch
type
var

Next > Go 103