# CHƯƠNG 4 : COMPOSITE TYPES

# Arrays

* Tập hợp các phần tử có cùng kiểu dữ liệu.
* Có độ dài cố định.
* Mặc định khi khai báo mảng, nếu không khởi tạo giá trị thì các phần tử có giá trị bằng 0.
* Khai báo mảng :

var intArray [3]int hoặc intArray := [3]int // Khai báo một mảng số nguyên gồm 3 phần tử

Khai báo sử dụng “…” thay cho số phần tử >> Chiều dài của mảng bằng số phần tử :

IntArray := […]int{1, 2, 3} // [3]int

* Khởi tạo giá trị của mảng :

intArray[0] = 1

intArray[1] = 2

intArray[2] = 3

Khởi tạo khi khai báo :

intArray := [3]int{6, 7, 8}

* Duyệt mảng:

for i, v := range intArray {

fmt.Printf("%d %d\n", i, v)

}

For i := 0 ; i < len(intArray); i++ {

fmt.Printf("%d %d\n", i, intArray[i])

}

* So sánh mảng : Hai mảng bằng nhau khi cùng chiều dài và các phần tử bằng nhau

a := [2]int{1, 2}

b := [...]int{1, 2}

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

fmt.Println(a == b, a == c, b == c) // "true false false"

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

fmt.Println(a == d) // compile error: cannot compare [2]int == [3]int

* Kích thước mảng cố định, thuộc kiểu value type chứ không phải reference. Khi gán 1 mảng cho một biến khác hoặc truyền 1 mảng như một tham số vào 1 hàm nào đó thì toàn bộ mảng được copy. Nếu thay đổi giá trị trong mảng mới này thì mảng cũ vẫn giữ nguyên.

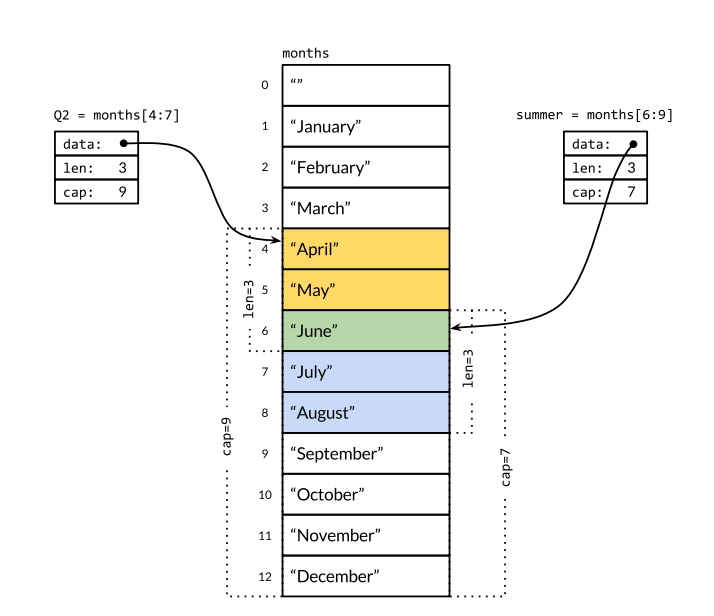
# Slices

* Slice có vẻ giống như mảng với độ dài linh hoạt, không cố định. Tuy nhiên, về bản chất Slice không chứa giá trị nào mà nó chỉ có con trỏ tham chiếu đến mảng.
* Cấu trúc của 1 slice :
  + Con trỏ : Tham chiếu đến 1 mảng.
  + Kích thước : Số lượng phần tử trong slice.
  + Sức chứa : Số phần tử trong mảng tính từ vị trí bắt đầu của slice.
* Khai báo slice :
  + Tạo slice từ mảng đã có : sliceName := arrayName[start:end]
  + Khai báo tắt : sliceName := []Type {value1, value2, …}
  + Khai báo với make: sliceName := make([ ]Type, length, capacity) ( có thể bỏ cap nếu len = cap)
* Duyệt slice

for i, slice := range slices {

fmt.Printf("index = %d, value = %s\n", i, slice)

}



* Khác với mảng, slice là kiểu tham chiếu. Slice tham chiếu đến 1 mảng cơ bản. Do đó việc sửa đổi các phần tử trong slice sẽ sửa đổi các phần tử trong mảng mà slice tham chiếu đến. Các slice khác tham chiếu đến mảng đó cũng bị thay đổi.
* Khi thêm 1 phần tử vào slice ta sử dụng hàm append(slice,value). Khi thêm, Go sẽ xác định số phần tử hiện tại của slice a và số phần tử sắp thêm liệu có vượt quá sức chứa hay không. Nếu có, Go sẽ tạo 1 mảng mới đủ để chứa và chép các phần tử cũ và mới vào đó. Nếu không, Go đơn giản chép thêm phần tử mới vào và tăng chiều dài lên tương ứng.
* Muốn sao chép slice ta sử dụng hàm copy(slice2, slice1).

# Maps

* Tập hợp các phần tử được lưu trữ dưới dạng key – value. Ở các ngôn ngữ khác nó giống như hàm băm (hash).
* Map tham chiếu đến 1 bảng băm.
* Để tạo 1 map rỗng (empty map) chúng ta sử dụng :

make(map[key-type]val-type)

* Thêm phần tử hoặc thay đổi giá trị :

map[key] = value

* Để truy xuất đến phần tử trong map, ta gọi map kèm theo key của phần tử. Nếu key đó không tồn tại thì ta sẽ thu được giá trị là zero value (tùy theo kiểu dữ liệu)
* Để kiểm tra xem một phần tử có tồn tại trong map hay không, ta sẽ lấy cùng lúc 2 kết quả khi truy xuất đến phần tử trong map. Giá trị đầu tiên giống ví dụ trên, giá trị thứ 2 sẽ là true nếu phần tử có trong map và false nếu phần tử không tồn tại.
* Duyệt map :

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

for k, v := range kvs {

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

}

* Xóa 1 key trong map

delete(map, key)

* Map là kiểu tham chiếu. Map tham chiếu đến 1 bảng băm. Do đó việc sửa đổi các phần tử trong map sẽ sửa đổi các phần tử trong bảng mà map tham chiếu đến. Các map khác tham chiếu đến bảng đó cũng bị thay đổi.

# Structs

* Một struct là một kiểu dữ liệu đặc biệt, kiểu này chứa biến thuộc các kiểu dữ liệu khác, các biến ở đây thường được gọi là các trường hoặc các thuộc tính.

type Circle struct {

x, y, r float64

}

* Khai báo biến kiểu struct như khai báo biến thông thường
* Sử dụng các trường trong struct thông qua dấu “.”
* Cũng giống như các kiểu dữ liệu khác, khi truyền một struct vào hàm, thực chất Go sẽ sao chép biến đó vào trong tham số của hàm chứ không trực tiếp thao tác với hàm, do đó nếu muốn hàm thực hiện các thao tác trên chính struct được truyền vào thì chúng ta phải truyền con trỏ (hoặc địa chỉ bộ nhớ của biến)
* Trong 1 struct ta có thể tạo phương thức riêng cho struct đó :

func (c \*Circle) area() float64 {

return math.Pi \* c.r\*c.r

}

Để gọi phương thức này ta dùng dấu “.” : c.area()

* Struct có thể so sánh được nếu các field của nó có thể so sánh được, và 2 biến kiểu struct có giá trị giống nhau nếu toàn bộ các field có giá trị giống nhau.

# JSON

* JSON (JavaScript Object Notation) là định dạng file nhỏ gọn, tiện trong trao đổi dữ liệu nhưng vẫn có khả năng tự mô tả.
* Go cung cấp hàm thuộc package encoding/json để phân tích JSON :

func Unmarshal(data []byte, v interface{}) error

* Trong trường hợp đã xác định được cấu trúc JSON, ta có thể tạo struct tương ứng rồi truyền biến kiểu struct này làm tham số thứ 2 ở hàm Unmarshal để nhận dữ liệu từ slice chứa dữ liệu JSON ở tham số thứ 1.

var titles []struct{ Title string }

if err := json.Unmarshal(data, &titles); err != nil {

log.Fatalf("JSON unmarshaling failed: %s", err)

}

* Đóng gói dữ liệu json :

func Marshal(v interface{}) ([]byte, error)

với struct được đưa vào tham số và dữ liệu trả về là mảng byte JSON.

# Text and HTML Templates

* Khi có nhu cầu phản hồi trang web về cho trình duyệt ta thường sử dụng 1 mẫu HTML. Go cung cấp các package text/template và html/template giúp xử lý việc tạo mẫu này.
* Khai báo 1 mẫu HTML : có thể chứa trong một biến kiểu string hoặc 1 file lưu sẵn.

func New(name string) \*Template

* Phân tích mẫu với phương thức Parse với mẫu lưu trong biến string hoặc ParseFile với mẫu lưu trong file.

func (t \*Template) Parse(text string) (\*Template, error)

func (t \*Template) ParseFiles(filenames ...string) (\*Template, error)

* Thực hiện xử lý dữ liệu ghép vào mẫu với phương thức Execute:

func (t \*Template) Execute(wr io.Writer, data interface{}) error