ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH


# LẬP TRÌNH NÂNG CAO

## Báo cáo Bài tập lớn

# NGÔN NGỮ GO (GOLANG)

**Giảng viên:** Trương Tuấn Anh
**Sinh viên:** Nguyễn Trọng Nghĩa - 2013873

Hồ Chí Minh, Tháng 04/2022

# MỤC LỤC

**1 Thông tin sơ lược về GO** **2**
**1.1** Ngôn ngữ Go (Golang) . . . 2
**1.2** Cài đặt . . . 2

**2 Các cú pháp và kiểu dữ liệu** **4**
**2.1** Packages . . . 4
**2.2** Khai báo biến . . . 5
**2.3** Các kiểu dữ liệu nguyên thủy . . . 6
**2.4** Hằng số (constant) và Enum . . . 7
**2.5** Các kiểu dữ liệu nâng cao . . . 8
**2.5.1** Struct và Con trỏ (Pointers) . . . 8
**2.5.2** Array và Slice . . . 10
**2.5.3** Map . . . 12
**2.6** Các câu lệnh điều kiện và Vòng lặp . . . 13
**2.6.1** If-else . . . 13
**2.6.2** Switch - Case . . . 14
**2.6.3** For-loop . . . 15
**2.6.4** Defer, Recover Panic . . . 16
**2.7** Hàm và liên quan tới hàm - Function . . . 17
**2.7.1** Hàm (function) . . . 17
**2.7.2** Method và Interface . . . 17

**3 Concurrency** **22**
**3.1** Goroutine . . . 22
**3.2** Channel . . . 23

**4 Điểm mạnh - Điểm yếu** **25**
**4.1** Điểm mạnh . . . 25
**4.2** Điểm yếu . . . 26

# CHƯƠNG 1
# THÔNG TIN SƠ LƯỢC VỀ GO

## 1.1 Ngôn ngữ Go (Golang)

Go là ngôn ngữ mã nguồn mở được phát triển bởi các kỹ sư của Google, cùng với cộng đồng mã nguồn mở. Go đượclên ý tưởng thiết kế bởi Google vào tháng 9 năm 2007 bởi các kỹ sư Robert Griesemer, Ken Thompson, Rob Pyke , và chính thức được giới thiệu tới cộng đồng vào năm 2009. Sau hơn 10 năm tồn tại, Go đã được ứng dụng rộng rãi không chỉ riêng ở Google, mà kể cả Dropbox, Docker, Uber,..

Go ban đầu được lên ý tưởng nhằm tăng năng suất lập trình trong kỉ nguyên những thiết bị mạng đa nhân và kho dữ liệu code khổng lồ. Và tới nay, những đặc điểm Go mang lại, đặc biệt ở khía cạnh thực thi đồng thời (concurrency) và thiết kế phần mềm (software engineering) đã có những tác động đến với các ngôn ngữ lập trình khác và các công cụ của chúng.

Tổng quan qua, Go có cấu pháp lập trình khá tương đồng với ngôn ngữ C/C++, nhưng bên cạnh cũng có những cú pháp (syntex) tinh giản, gọn nhẹ hơn. Go cũng là một ngôn ngữ biên dịch (compiled) như C/C++ nên hiệu suất làm việc của Go có thể nói là hơn hẳn các ngôn ngữ thông dịch hiện tại như Java, Python.

Dưới đây là chương trình helloWorld của Go:

```
1 package main
2 import "fmt"
3
4 func main() {
5     fmt.Println("Hello, World")
6 }
```

Để tìm hiểu những thông tin về Go, ta có thể truy cập trang `go.dev`. Tại đó ta có thể tìm thấy bất kì tài liệu nào về Go và các ứng dụng của nó, đồng thời web cũng cung cấp cho ta Playground và gợi ý Lộ trình tiếp cận Go để làm quen với ngôn ngữ lập trình này.

## 1.2 Cài đặt

Cài đặt Go cho máy tính khá là dễ dàng. Trên trang `go.dev` bạn có thể click vào mục Download, sau đó tìm kiếm phiên bản phù hợp cho máy tính của bạn và bắt đầu quá trình cài đặt như bình thường. Hiện tại Go hỗ trợ cho cả Windows, MacOS, Linux.

Sau khi cài đặt, bạn có thể kiểm tra xem đã thành công hay chưa bằng cách vào Command Prompt và thực hiện:

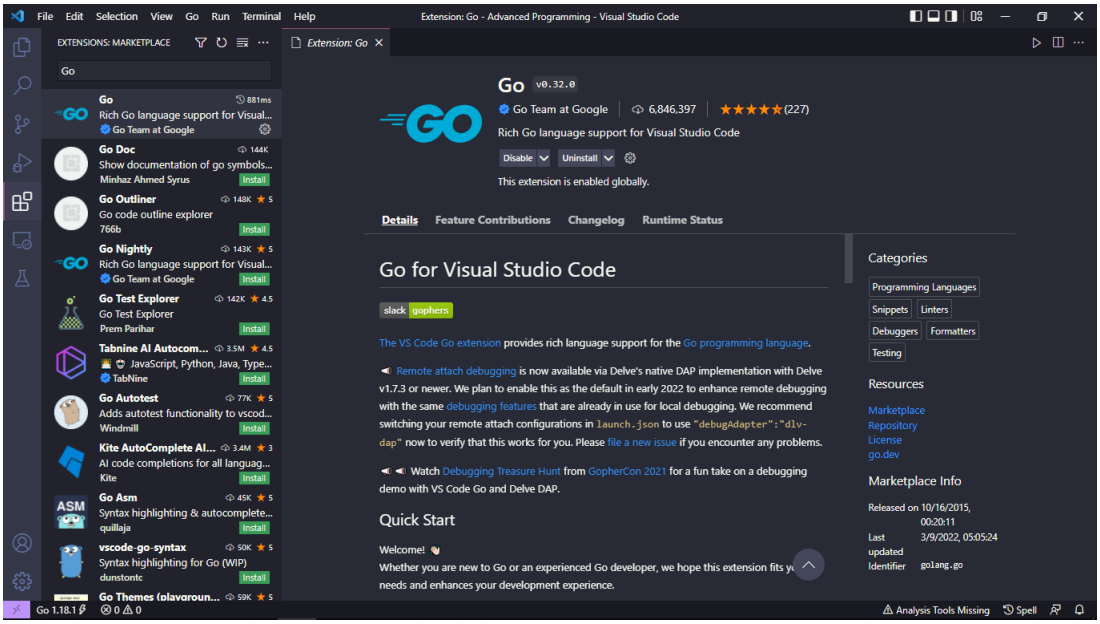
```
Microsoft Windows [Version 10.0.17134.112]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Welcome>go version
go version go1.18.1 windows/amd64
```

Nếu xuất hiện dòng version như hình trên chứng tỏ đã cài đặt thành công. Sau đó bạn có thể tạo một chương trình Go bằng bất cứ trình soạn thảo nào và lưu file dưới dạng <name>.go và dùng lệnh go run để chạy thử nhé

```
C:\Users\Welcome>go run Downloads/hello.go
Hello, World
```

Ngoài ra nếu đang sử dụng VS Code bạn cũng có thể cài đặt thêm extension Go là có thể code Go được rồi. Có thể thấy Go hiện nay có độ tương thích tốt ở nhiều đối tượng, rất thân thiện đối với những người tiếp cận ngôn ngữ này.



Hình 1.1: Extension Go

# CHƯƠNG 2

# CÁC CÚ PHÁP VÀ KIỂU DỮ LIỆU

## 2.1 Packages

Mỗi chương trình Go được cấu tạo nên từ nhiều package. Và khi khởi chạy sẽ bắt đầu từ package main. Do đó, khi bắt đầu viết mỗi chương trình ta luôn bắt đầu bằng câu lệnh *package main*. Trong một chương trình, ta sẽ cần một hay nhiều các package để hỗ trợ trong việc hiện thực. Để làm được việc đó ta dùng câu lệnh *import* theo sau là đường dẫn tới package đó. Để thuận tiện thì, tên package chúng ta sẽ sử dụng sẽ thường trùng với tên của thành tố cuối cùng của đường dẫn.

Một chương trình ví dụ:

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6      "math/rand"
7      "time"
8  )
9
10 // or
11 /*
12    import (
13        "fmt"
14        "math"
15        "math/rand"
16        "time"
17    )
18 */
19 // func get random int smaller than 10
20 func main() {
21     rand.Seed(time.Now().UnixNano())
22     fmt.Println("Pick a number: ", rand.Intn(10))
23     fmt.Println("Pick a Pi: ", math.Pi)
24 }
```

Có thể thấy, khi sử dụng nhiều thư viện chúng ta còn có thể gom chúng vào một cặp dấu ngoặc '()'. Cách làm này được gọi là factored và thường được sử dụng nhiều hơn. Bên cạnh đó, để các hàm, các biến trong package có thể được sử dụng ở nhiều nơi khác, tên của chúng sẽ được bắt đầu bằng chữ cái viết hoa. Nếu không đặt tên theo quy tắc này, những thành phần đó sẽ không thể truy xuất từ bên ngoài. Trong ví dụ trên là các hàm Intn() trong package rand hay Pi trong package math.

Lỗi khi truy xuất biến không được export

```
1  ./prog.go:23:34: undefined: math.pi
```

## 2.2 Khai báo biến

Có rất nhiều cách để khai báo một biến trong Golang.

Ví dụ trong code:

```
package main

import "fmt"

var c, python, java bool

var g = 10

var (
    n int    = 10
    m string = "Hello"
)

func main() {
    var i int
    i = 10
    var a int = 100
    b := 1.54
    c := "World"
    var g = 100
    fmt.Println(n, m, python, java)
    fmt.Println(i, a, b, c)
    fmt.Printf("%v: %T\n", b, b)
    fmt.Println(g)
}
```

Cấu trúc cơ bản của việc khai báo biến trong Golang sẽ là *var <name> (type) (= giá trị)*. Các biến có cùng kiểu với nhau có thể khai báo trên cùng một dòng, cách nhau bởi dấu ','. Nếu đã có giá trị khởi tạo, ta có thể bỏ qua thành phần type vì Golang hỗ trợ chúng ta trong việc nhận diện kiểu phải gán vào.

Tuy nhiên, Golang cũng hỗ trợ chúng ta việc khai báo biến theo cách *<name> := giá trị*. Như đã nói ở trên Golang, hỗ trợ chúng ta nhận diện kiểu phải gán vào.

Mặc dù vậy, nếu biến ta khai báo là biến global, ta chỉ có thể khai báo theo cách đầu tiên bắt đầu bằng từ khóa var. Và nếu chúng ta có 2 biến cùng tên, một biến là thuộc global, một biến trong hàm main, thì Golang sẽ ưu tiên sử dụng biến trong hàm nếu ta đang thực thi lệnh tại đó.

Các biến chưa được khởi tạo giá trị ban đầu sẽ được gán cho giá trị mặc định trong Golang, chúng ta sẽ nói qua ở phần sau. Và Golang không muốn xuất hiện những dữ liệu rác trong bộ nhớ, nên khác với một số ngôn ngữ chỉ warning, Golang không cho phép các biến được khai báo mà không sử dụng, unused variable.

Kết quả của đoạn code trên

```
10 Hello false false
10 100 1.54 World
1.54: float64
100
\\ error when unused variable

// ./prog.go:16:6: f declared but not used
```

Điểm hay của các công cụ hỗ trợ biên dịch Golang sẽ luôn kiểm tra và báo lỗi thường xuyên trong quá trình chúng ta code nếu gặp phải trường hợp lỗi trên hay những lỗi cú pháp khác, tự động import thư viện chúng ta cần nên rất tiện lợi.

Ngoài ra việc đặt tên biến như đã nói ở phần trên, hãy chú ý nếu bạn muốn biến có thể export ra ngoài và nên đặt tên biến là những từ có nghĩa.

## 2.3 Các kiểu dữ liệu nguyên thủy

Trong Golang, chúng ta có thể chia thành ba nhóm kiểu dữ liệu nguyên thủy chính (Primitive Types/ Basic Types)

1. Nhóm kiểu số (Numerics):

Ở nhóm này, ta lại chia thành các kiểu số nguyên (Integer), nhóm kiểu số chấm động (Float), nhóm kiểu số phức (Complex). Ở đây ta có một ví dụ:

```
package main

import (
    "fmt"
)

var (
    j int      = 10
    k int      = 3
    i uint64   = 8
    c uint64   = i << 3 // SHIFT LEFT
    f float32 = 1.5
    z          = complex(3, 5)
)

func main() {
    fmt.Printf("Value: %v, Type: %T\n", c, c)
    fmt.Printf("Value: %v, Type: %T\n", z, z)
    fmt.Printf("Value: %v, Type: %T\n", real(z), real(z))
    fmt.Println(i, c)
    fmt.Println(j & k) // AND LOGIC
    fmt.Println(j | k) // OR LOGIC
}
```

Có thể thấy, Golang hỗ trợ không chỉ các kiểu số nguyên có dấu (int) hay số nguyên không dấu (uint), hay kiểu số thực (float) mà ở mỗi loại Golang còn hỗ trợ rất kĩ trong số bit của nó. Ngoài ra, Golang cũng hỗ trợ các phép toán logic như: & (And), | (Or), « (Shift Left), ^(Nor)...

Còn ở kiểu số phức, ngoài cách khai báo bằng hàm complex() như trên ví dụ, ta cũng có thể tự khai báo với giá trị thực như -5+12i, 3+4i,.. Và các thành phần phần thực, phần ảo của số phức đều sẽ là số chấm động và tùy vào kiểu số phức khai báo là complex64 hay complex128 mà kiểu của các phần đó là float32 hay float64.

Kết quả ví dụ

```
Value: 64, Type: uint64
Value: (3+5i), Type: complex128
Value: 3, Type: float64
8 64
2
11
```

2. Nhóm kiểu logic (Boolean):

Ở nhóm này thì Golang cũng tương tự như nhiều ngôn ngữ khác. Các biến boolean sẽ nhận 2 giá trị là true hoặc false. Và nếu không có giá trị khởi tạo, biến sẽ nhận giá trị mặc định là false.

Một ví dụ:

```
package main

import "fmt"

func main() {
    var b1 bool = true
    var b2 bool
```

```
8       fmt.Printf("Value: %v, Type :%T\n", b1, b1)
9       fmt.Printf("Value: %v, Type :%T\n", b2, b2)
10  }
```

Kết quả:

```
1  Value: true, Type :bool
2  Value: false, Type :bool
```

3. Nhóm kiểu kí tự, chuỗi (String):

Ở nhóm này, Golang hỗ trợ hai kiểu kí tự là byte và rune. Byte đại diện cho uint8, tức là các ký tự Latin (có dấu và không dấu, còn Rune đại diện cho cho int32, dùng để thể hiện các kí tự tượng hình trong ngôn ngữ.

Còn về phần chuỗi (string), mặc định giá trị khởi tạo cho chuỗi là một chuỗi rỗng "", và Golang cũng cho phép chúng ta thực hiện việc nối các chuỗi thông qua toán tử '+'.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var s1 = "Advanced"
7      var s2 = "Programming"
8      var s3 = s1 + " " + s2
9      fmt.Println(s3)
10 }
```

Kết quả:

```
1  Advanced Programming
```

## 2.4 Hằng số (constant) và Enum

Các hằng số trong Go có thể được khai báo với từ khóa *const <name> (type) = giá trị*. Các hằng số không thể khai báo với toán tử ':='. Tương tự với tính chất biến thông thường, Go cũng sẽ ưu tiên hằng số cùng tên nằm trong scope hơn.

Enum trong Go là tập các constant có liên hệ với nhau, được đặt trong cặp dấu '()'.

Ví dụ:

```
1  package main
2
3  import "fmt"
4
5  const (
6      red = 0
7      yellow = 1
8      green
9      purple
10 )
11
12 func main() {
13     fmt.Printf("value: %v\n", red)
14     fmt.Printf("value: %v\n", yellow)
15     fmt.Printf("value: %v\n", green)
16     fmt.Printf("value: %v\n", purple)
17 }
```

Kết quả:

```
value: 0
value: 1
value: 1
value: 1
```

Tuy nhiên, nếu chỉ vậy thì chưa thể gọi là enum được, do vậy trong Go, có một cách để khởi tạo enum đó là hàm itoa. iota được khai báo ở dòng nào sẽ lấy giá trị index của dòng đó làm giá trị khởi tạo cho mình, và biểu thức trên iota sẽ được áp dụng cho tất cả các biến còn lại trong enum.

Một số cách sử dụng iota

```
//basic
const (
    defaultVal = iota
    red        //1
    yellow     //2
    green      //3
    purple     //4
)
//when variable have relations to bit
const (
    defaultVal = 1 << iota // shift left iota = 0
    red        //2 -- shift left 1
    yellow     //4 -- shift left 2
    green      //8
    purple     //16
)
//when variable have type bool
const (
    defaultVal = iota % 2 == 0
    red        //1%2 == 0 -> false
    yellow     //2%2 == 0 -> true
    green      //         -> false
    purple     //         -> true
)
//when iota in middle
const (
    defaultVal = 0
    red        = 4
    yellow     = iota //2
    green             //3
    purple            //4
)
```

Do đó, đối với enum mà các biến có biểu thức liên quan, hay quy luật, iota là một hàm hữu hiệu trong việc khởi tạo và thể hiện mối liên hệ giữa chúng.

## 2.5 Các kiểu dữ liệu nâng cao

### 2.5.1 Struct và Con trỏ (Pointers)

Golang cũng cho phép struct và pointer. Tương tự với các ngôn ngữ khác, định nghĩa về struct và pointer trong Golang cũng không có khác biệt gì.

Để khai báo một struct ta sẽ dùng cú pháp *type <name> struct . Ví dụ:*

```
package main

import "fmt"

type Student struct {
    X string
    Y int
}

func main() {
```

```
    student_1 := Student{"Nguyen Trong Nghia", 0}     //type 1
    student_2 := Student{Y: 2013, X: "Nguyen Van A"} // type 2
    student_3 := Student{}                           // type 3
    fmt.Println(student_1)
    fmt.Println(student_2)
    fmt.Println(student_3)
}
```

Trong struct là các dữ liệu mà người dùng muốn struct chứa. Để khai báo giá trị cho một biến kiểu struct ta có thể dùng cách liệt kê các giá trị mà biến đó sẽ chứa. Nếu ta liệt kê cùng tên biến trong struct thì không cần phải theo thứ tự, còn không ta phải khai báo giá trị theo đúng thứ tự tên biến trong struct. Nếu chúng ta khai báo một struct không có giá trị, Go sẽ gán giá trị mặc định của kiểu biến trong struct.

Kết quả:

```
{Nguyen Trong Nghia 0}
{Nguyen Van A 2013}
{ 0}
```

Con trỏ, pointer, trong Golang cũng giữ địa chỉ ô nhớ của một giá trị. Tương tự các ngôn ngữ khác, $T*$ cũng là con trỏ trỏ vào giá trị kiểu $T$. Nếu con trỏ không trỏ vào đâu cả, nó sẽ mang giá trị $<nil>$. Toán tử '&' cũng là phép lấy địa chỉ để gán vào con trỏ trong Golang, trong khi toán tử '*' cũng được dùng để lấy giá trị ở ô nhớ mà con trỏ đang trỏ tới.

Một đoạn ví dụ:

```
package main

import "fmt"

type Student struct {
    X string
    Y int
}

func main() {
    student_1 := Student{"Nguyen Trong Nghia", 0} //type 1
    var i int = 72
    var t *int
    p := &i
    fmt.Println(*p)
    *p = 72 / 6
    fmt.Println(i)
    fmt.Println(t)

    k := &student_1
    k.X = "Nghia Ng" // use pointer to set value
    (*k).Y = 2013 // use pointer to set value
    fmt.Println(student_1.X)
    fmt.Println(student_1.Y)
    fmt.Println(k) // print pointer
}
```

Điểm mới ở ngôn ngữ Golang là khi dùng con trỏ trỏ vào một biến kiểu struct, ta không cần phải khai báo (*pointer).fields để thay đổi giá trị của fields mà chỉ cần pointer.fields, Go sẽ tự hiểu đó là con trỏ. Tuy nhiên, để tránh nhầm lẫn và quen thuộc hơn, ta vẫn nên khai báo đầy đủ. Bên cạnh đó, nếu ta in con trỏ, Go sẽ trả về giá trị con trỏ đang trỏ tới cùng với toán tử & phía trước.

Kết quả:

```
72
12
<nil>
Nghia Ng
2013
&{Nghia Ng 2013}
```

### 2.5.2 Array và Slice

1. Mảng (Array):

Mảng là một tập hợp nhiều giá trị thuộc cùng một kiểu dữ liệu. Trong Golang, mảng được đại diện bởi *[n]T*, trong đó n là số phần tử, T là type. Để truy xuất tới từng index trong mảng, cú pháp cũng tương tự như trong C/C++, <name>[index].

Một ví dụ:

```
package main

import "fmt"

func main() {
    arr1 := [3]int{0, 1, 2}
    fmt.Printf("Value: %v, Type: %T\n", arr1, arr1)

    var arr2 [3]int
    arr2[0] = 1
    arr2[1] = 2
    arr2[2] = 3
    fmt.Printf("Value: %v, Type: %T\n", arr2, arr2)
}
```

Kết quả:

```
Value: [0 1 2], Type: [3]int
Value: [1 2 3], Type: [3]int
```

Bên cạnh đó, việc khởi tạo mảng từ một mảng cũng sẽ khác nhau tùy thuộc vào cách chúng ta khởi tạo

```
package main

import "fmt"

func main() {
    arr1 := [3]int{0, 1, 2}
    arr2 := arr1
    arr2[0] = 4
    fmt.Printf("Value: %v, Type: %T\n", arr1, arr1)
    fmt.Printf("Value: %v, Type: %T\n", arr2, arr2)

    arr3 := &arr1
    arr3[0] = 4
    fmt.Printf("Value: %v, Type: %T\n", arr1, arr1)
    fmt.Printf("Value: %v, Type: %T\n", *arr3, *arr3)
}
```

Kết quả:

```
Value: [0 1 2], Type: [3]int
Value: [4 1 2], Type: [3]int
Value: [4 1 2], Type: [3]int
Value: [4 1 2], Type: [3]int
```

Sự khác biệt sẽ xảy ra nếu ta sử dụng con trỏ, vậy nên cần phải để ý khi sử dụng.

Bên cạnh đó, array có một nhược điểm là kích thước mảng sẽ là cố định, không thay đổi được. Do đó, Go đã phát triển một kiểu mới, gọi là Slice, khá tương tự với vector trong C/C++ để giải quyết vấn đề này.

2. Slice:

Trái ngược với array, slice có kích thước động, cùng hỗ trợ một một cái nhìn linh hoạt hơn tới các phần tử trong mảng. Do đó, trong thực tế, slice thường được sử dụng nhiều hơn thay vì là array.

Một slice sẽ có kiểu tổng quát là *[]T*. Thực chất, slice không phải là một kiểu chứa dữ liệu, nó giống kiểu một con trỏ hay tương tự để trỏ vào một phần nào đó của một mảng nằm dưới nó(underlaying array). Do đó nếu slice thay đổi sẽ làm cho mảng đó thay đổi theo, đồng thời là cả những slice đang hướng tới cùng một mảng. Để tạo một slice từ một mảng, ta có thể sử dụng cấu trúc *<name>[lower:upper]. Nếu ta bỏ trống giá trị lower, Go sẽ mặc định giá trị là 0, còn nếu là giá trị upper, chiều dài slice sẽ là giá trị mặc định.*

Một ví dụ:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      names := [4]string{
7          "Ichigo",
8          "Sogoku",
9          "Luffy",
10         "Naruto",
11     }
12     fmt.Println(names) // array
13
14     a := names[0:2] // Slice - index 0,1
15     b := names[1:3] // Slice - index 1,2
16     fmt.Println(a, b)
17
18     b[0] = "Nghia" // change Ichigo - Nghia
19     fmt.Println(a, b)
20     fmt.Println(names)
21 }
```

Kết quả:

```
1 [Ichigo Sogoku Luffy Naruto]
2 [Ichigo Sogoku] [Sogoku Luffy]
3 [Ichigo Nghia] [Nghia Luffy]
4 [Ichigo Nghia Luffy Naruto]
```

Ngoài ra, slice cũng có thể khai báo tương tự như array nhưng không có kích thước đưa vào:

```
1 array := [3]bool{true, false, false}
2 sli := []bool{true, false, false}
```

Ở slice khác với array đó là có tồn tại giá trị khối lượng (capacity) bên cạnh độ dài slice (length). Để lấy 2 giá trị này của một slice ta dùng hàm len(<slice>) để lấy độ dài, cap(<slice>) để lấy khối lượng.

Nếu slice đang có độ dài và khối lượng cùng bằng 0 và không trỏ tới một array nào, nó sẽ mang giá trị *nil*.

Slice cũng có thể khởi tạo bằng từ khóa make. Ví dụ:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      a := make([]int, 5)
7      fmt.Printf("%s len=%d cap=%d %v\n", "a", len(a), cap(a), a)
8
9      b := make([]int, 0, 5)
10     fmt.Printf("%s len=%d cap=%d %v\n", "b", len(b), cap(b), b)
11 }
```

Theo đó, nếu ta chỉ đưa một tham số kiểu int vào hàm make(), Go sẽ hiểu đó là cả khối lượng và độ dài, nên để có thể cụ thể, ta nên đưa vào cả hai tham số chiều dài và khối lượng.

Kết quả:

```
a len=5 cap=5 [0 0 0 0 0]
b len=0 cap=5 []
```

Như đã đề cập, slice là một kiểu mảng có kích thước động. Để tăng kích thước cho slice bằng cách thêm phần tử, Go hỗ trợ ta hàm append().

Ví dụ:

```
package main

import "fmt"

func main() {
    var s []int
    k := []int{1,2,3}
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)

    // append 0 to nil slice s
    s = append(s, 0)
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)

    // add more than one element at a time.
    s = append(s, 2, 3, 4)
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)

    // add a slice to slice
    s = append(s, k...)
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

Theo đó, ta có thể thêm 1 phần tử, hay nhiều phần tử hay thêm slice vào slice. Tuy nhiên khi thêm slice vào slice ta phải có thành phần '...' sau slice thêm vào thì Go mới chấp nhận.

Kết quả:

```
len=0 cap=0 []
len=1 cap=1 [0]
len=4 cap=4 [0 2 3 4]
len=7 cap=8 [0 2 3 4 1 2 3]
```

Ở ví dụ trên, ta thấy len của hàm s sau cùng lại có kết quả là 8 (thay vì 7) thì đó là do thuật toán của hàm append() để đáp ứng đủ khối lượng để thêm phần tử vào slice.

Ngoài ra slices còn hỗ trợ từ khóa range cho slice trong các vòng lặp với 2 giá trị trả về, giá trị đầu là index, giá trị thứ hai là value tại index đó. Tuy nhiên sẽ sử dụng trong phần câu lệnh loop.

### 2.5.3 Map

Map là kiểu dữ liệu quen thuộc với hai thành phần ở mỗi giá trị trong nó là key và value. Map cũng có hai kiểu khai báo là trực tiếp hoặc thông qua từ khóa make. Tuy nhiên nếu khai báo trực tiếp, phải đồng thời gán giá trị vào cho map. Và trong cả 2 kiểu khai báo đều kèm thành phần *map[typeOfKey]typeOfValue*.

Ví dụ:

```
package main

import "fmt"

func main() {
    m1 := map[string]int{
        "Tran Van B": 201203,
        "Nguyen Van A": 201387,
    }
```

```
10      fmt.Println(m1)
11
12      m2 := make(map[string]int)
13      m2["Nguyen Kim C"] = 201390
14      fmt.Println(m2)
15  }
```

Kết quả:

```
1 map[Nguyen Van A:201387 Tran Van B:201203]
2 map[Nguyen Kim C:201390]
```

Tương tự như slice map có thể dùng toán tử '[]' để truy xuất key cần tìm và thêm key mới vào map. Thêm vào đó, map sẽ tự động sắp xếp lại thứ tự key theo alphabet, như ví dụ map m1 ở trên. Bên cạnh đó, Go cũng hỗ trợ những hàm, từ khóa khác như delete(map, key) để xóa những key cần xóa, hay cách lấy ra giá trị một key sẽ trả ra 2 kết quả là giá trị key đó và kiểm tra key đó có tồn tại trong map không. Lí do là vì trong Go, một key không có trong map sẽ trả về kết quả là giá trị mặc định của giá trị. Ví dụ:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      m := make(map[string]int)
7
8      m["Nguyen Trong A"] = 204213
9      fmt.Println("The value:", m["Nguyen Trong A"])
10
11     delete(m, "Nguyen Trong A")
12     fmt.Println("The value:", m["Nguyen Trong A"])
13
14     v, ok := m["Nguyen Trong A"]
15     fmt.Println("The value:", v, "Present?", ok)
16 }
```

Kết quả

```
1 The value: 204213
2 The value: 0
3 The value: 0 Present? false
```

Ta thấy dòng lệnh *v, ok := m["Nguyen Trong A"]* sẽ trả về hai giá trị, biến v sẽ giữ giá trị của key, còn biến ok sẽ trả về kiểu bool thể hiện trong map có tồn tại key đó không.

## 2.6 Các câu lệnh điều kiện và Vòng lặp

### 2.6.1 If-else

Câu lệnh rẽ nhánh trong Go có một vài điểm khác so với C/C++. Go không bắt buộc chúng ta phải đặt điều kiện trong cặp dấu ngoặc '()', nhưng yêu cầu khối lệnh thực hiện phải luôn đặt trong cặp dấu '' kể cả khi chỉ có một câu lệnh. Ngoài ra, Go cũng cho phép if thực hiện một lệnh khai báo biến ngắn trước điều kiện, ngăn cách bởi dấu ';'. Biến này sẽ chỉ tồn tại trong khối lệnh if này tới khi kết thúc, khá tương tự với việc khai báo biến điều kiện trong vòng lặp loop for của C/C++.

Ví dụ:

```
1  package main
2
3  import (
4      "fmt"
5  )
6
```

```go
func main() {
    map1 := map[string]int{
        "Nguyen Van A": 2012,
        "Nguyen Thi B": 2013,
    }
    if _, ok := map1["Nguyen Van A"]; ok == true {
        fmt.Println("Exist A")
    } else {
        fmt.Println("not Exist B")
    }

}
```

***Note:*** Như đã làm quen ở phần Map, câu lệnh khai báo biến ok sẽ trả ra hai giá trị, kí tự '_' có ý nghĩa rằng giá trị trả ra này không quan tâm để tránh trường hợp Go báo lỗi unused variable.

Ngoài ra, câu lệnh *else* hay khi chúng ta thêm nhiều nhánh rẽ khác bằng *else if* các từ khóa phải trên cùng một hàng với kí tự kết thúc khối lệnh if.

Kết quả:

```
Exist A
```

### 2.6.2 Switch - Case

Cơ bản switch trong Go không khác gì về mặt ý nghĩa so với switch trong các ngôn ngữ khác. Tuy nhiên, nếu ở C/C++ ở mỗi case đều phải kèm câu lệnh break, thì Go sẽ dò từng case và khi thỏa mãn sẽ tự break ra khỏi nó. Và ở Go, các trường hợp không nhất thiết phải là constant và có thể là bất cứ kiểu gì.

Ví dụ:

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("When's Saturday?")
    today := time.Now().Weekday()
    switch time.Saturday {
    case today + 0:
        fmt.Println("Today.")
    case today + 1:
        fmt.Println("Tomorrow.")
    case today + 2:
        fmt.Println("In two days.")
    case today + 4:
        fmt.Println("In four days.")
    default:
        fmt.Println("Too far away.")
    }
}
```

Thời gian thực hiện đoạn code này là vào thứ Ba. Do đó khi kiểm tra tới trường hợp thỏa mãn, khối switch này sẽ tự ngắt ra. Nếu chúng ta cần kiểm tra một trường hợp có thể có nhiều trường hợp thỏa mãn, hãy dùng từ khóa *fallthrough* cuối mỗi trường để Go kiểm tra tất cả trường hợp.

Kết quả:

```
When's Saturday?
In four days.
```

Còn nếu trong câu lệnh switch không có biến điều kiện, hãy thêm nó vào từng case, như vậy thì sẽ khá tương tự câu lệnh if-else, else if.

Ví dụ:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    default:
        fmt.Println("Good evening.")
    }
}
```

### 2.6.3 For-loop

Go chỉ hỗ trợ một vòng lặp duy nhất là For. Những vòng lặp While, hay do-While trong C/C++ đều có thể hiện thực được trong For của Go.

Về cơ bản, for của Go cũng yêu cầu 3 trường ngăn cách nhau bởi dấu ';', đó là biến điều kiện, điều kiện và điều kiện sau vòng lặp. Và tất nhiên, Go không yêu cầu chúng phải đặt trong cặp dấu ngoặc ''.

Ngoài ra, Go cũng cho phép bỏ đi hai điều kiện đầu và cuối của vòng lặp. Trong trường hợp đó, Go cho phép bỏ luôn cả hai dấu ';', và như vậy, ta đã có vòng lặp while như C/C++.

Ví dụ:

```
package main

import "fmt"

func main() {
    sum := 1
    for sum < 1000 { // while in C/C++
        sum += sum
    }
    fmt.Println(sum)

    mini_Slice := []int{2, 3, 4}
    for index, value := range mini_Slice { // range ho tro lay value, index
        fmt.Println("Index, Value: ", index, value)
    }
}
```

Trong ví dụ, ta thấy được tác dụng của từ khóa *range* đối với một slice (có thể tương tự cho map) trong vòng lặp, đó là lấy ra index và value tương ứng trong nó.

Kết quả:

```
1024
Index, Value:  0 2
Index, Value:  1 3
Index, Value:  2 4
```

### 2.6.4 Defer, Recover Panic

1. Defer

Khi gặp từ khóa *defer* câu lệnh đó sẽ được Go sẽ đẩy tất cả vào một stack. Khi hàm chứa lệnh kết thúc, những câu lệnh đó mới được thực hiện theo thứ tự LIFO.

Ví dụ:

```go
package main

import "fmt"

func main() {
    fmt.Println("counting")

    for i := 0; i < 10; i++ {
        defer fmt.Printf("%v->", i)
    }

    fmt.Println("done")
}
```

Kết quả

```
counting
done
9->8->7->6->5->4->3->2->1->0->
```

Khi chạy vòng lặp for, các câu lệnh chứa từ khóa *defer* nên đã được đưa vào một stack và khi câu lệnh done kết thúc, các câu lệnh trong stack được lần lượt print ngược ra ngoài bắt đầu từ 9.

2. Recover Panic

Ở phần này, hai từ khóa này liên quan tới việc thông báo và xử lí lỗi không mong muốn trong chương trình. Nhìn qua thì chúng khá tương đồng với try..catch, exception trong C++. Panic có thể giúp chúng ta thay đổi những thông tin về lỗi khi xuất ra. Khi Go gặp từ khóa panic, nó sẽ ngừng tất cả các dòng lệnh phía sau lại, nhưng trước khi kết thúc hàm, nó vẫn sẽ thực hiện những câu lệnh defer đang bị chứa trong stack. Bên cạnh đó, hàm recover() sẽ giúp chúng ta khôi phục lại các panic đã gặp. Kết hợp lại ta có thể sử dụng như sau:

```go
package main

import "fmt"

func main() {
    fmt.Println("Start the program")
    // if hit the error that you predicted
    // call a func to inform
    panicker()
    fmt.Println("End the program")

}

func panicker() {
    fmt.Println("Panic hit")
    // func that used only
    defer func() {
        if err := recover(); err != nil {
            fmt.Println("Error: ", err)
            // panic(err) - if u want something func catch this error
        }
    }()
    panic("This is error!")
}
```

Với cách làm như vậy, chương trình chúng ta vẫn có thể chạy tới cuối cùng mà không bị crash, nhưng vẫn có thể bắt được lỗi đã xảy ra.

Kết quả:

```
Start the program
Panic hit
Error:  This is error!
End the program
```

## 2.7 Hàm và liên quan tới hàm - Function

### 2.7.1 Hàm (function)

Hàm trong Go có thể được khai báo với cú pháp như sau *func name(parameter) returnType {//TO DO}*. Hàm có thể có hoặc không tham số truyền vào. Ngoài ra return trong Go có thể return ra bất cứ kiểu gì và có thể định nghĩa cả biến return ở phần returnType.

```go
package main

import "fmt"
// if x, y is same Type, can be passed like this
func add(x, y int) int {
    return x + y
}

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

func beforeAfter(number int) (b, a int) {
    b = number - 1
    a = number + 1
    return
}

func main() {
    fmt.Println(add(3, 4))
    fmt.Println(swap("Hello", "World"))
    fmt.Println(beforeAfter(5))
}
```

Kết quả:

```
7
World Hello
4 6
```

Ngoài ra, Go cũng tồn tại các kiểu truyền tham số, truyền tham trị như các ngôn ngữ khác. Phần này sẽ được nói thêm ở phần sau.

Kiểu hàm được sử dụng trong phần code ở phần Recover Panic, được gọi là hàm ẩn danh(không có tên), là một loại hàm khai báo và sử dụng tại chỗ.

### 2.7.2 Method và Interface

1. Method

Không như C/C++ hay Java, Go không phải là ngôn ngữ hướng tới OOP và không tồn tại các class, tuy vậy, Go vẫn hỗ trợ định nghĩa và sử dụng các phương thức trên các kiểu struct hoặc non-struct.

Ví dụ:

```go
package main

import (
    "fmt"
    "math"
)

type MyFloat float64 //type MyFloat

type Vertex struct { //struct Vertex
    X, Y float64
}

func (f MyFloat) Abs() float64 { //method of MyFloat
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

func (v Vertex) Abs() float64 { //method of Vertex
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func Abs(v Vertex) float64 { // normal function
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
    v := Vertex{3, 4}
    fmt.Println("method: ", v.Abs())
    fmt.Println("normalFunc: ", Abs(v))
}
```

Kết quả:

```
1.4142135623730951
method:  5
normalFunc:  5
```

Nếu sử dụng theo cách này, ta không thấy được sự khác biệt giữa việc sử dụng hàm bình thường và method, bởi method trong Golang cũng chỉ là một hàm với cú pháp khác. Method cũng có thể nhận vào một con trỏ để thay đổi giá trị của biến gọi method đó. Và cũng tại đây sẽ xuất hiện điểm khác biệt giữa method và normal Function.

Ví dụ, ta có một hàm scaleFunc và một method scale được khai báo với giá trị nhận vào là con trỏ:

```go
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func ScaleFunc(v *Vertex, f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}
```

Nếu ta khai báo một biến Vertex v mà sử dụng chúng như sau:

```go
var v = Vertex{3, 4}
v.Scale(10)
fmt.Println(v)
ScaleFunc(v)    //option
```

Thử chạy thử trong trường hợp có và không có lệnh ScaleFunc(v), kết quả:

```
// truong hop khong co lenh ScaleFunc(v)
{30 40}
// truong hop co lenh ScaleFunc(v)
cannot use v (variable of type Vertex) as type *Vertex in argument to ScaleFunc
```

Từ đó có thể thấy nếu ta xài method, Go sẽ tự hiểu giá trị truyền vào method ta định nghĩa là dạng con trỏ, tức là hiểu v.Scale(10) thành (&v).Scale(10). Điều này cũng xảy ra trong trường hợp tương tự nếu ta khai báo giá trị truyền vào method là value nhưng khi gọi method bằng pointer, Go cũng sẽ tự hiểu cho chúng ta và không làm thay đổi giá trị bên trong con trỏ.

2. Interface:

Interface là một tập hợp những phương thức. Interface sẽ nhận bất cứ giá trị nào miễn là kiểu của nó có hiện thực đủ các method trong interface.

Trong Go, để hiện thực một interface không cần các từ khóa như implement, mà chỉ cần đơn giản hiện thực phương thức của interface đó như phần method đề cập tới.

Ví dụ:

```
package main

import (
    "fmt"
    "math"
)

type Abser interface {
    Abs() float64
}

type MyFloat float64

type Vertex struct {
    X, Y float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f // a MyFloat implements Abser
    fmt.Println(a.Abs())
    fmt.Printf("%v, %T\n", a, a)
    a = &v // a *Vertex implements Abser
    fmt.Printf("%v, %T\n", a, a)
    fmt.Println(a.Abs())
    // In the following line, v is a Vertex (not *Vertex)
    // and does NOT implement Abser.
    // a = v
}

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Như đoạn code trên, biến interface a phải có thể nhận được giá trị kiểu myFloat, và kiểu *Vertex (do khi chúng ta hiện thực phương thức của interface bằng struct Vertex, đã truyền vào con trỏ), do đó nếu như trên đoạn code chúng ta gán $a = v$, đoạn code sẽ báo lỗi ngay.

Kết quả:

```
1.4142135623730951
-1.4142135623730951, main.MyFloat
&{3 4}, *main.Vertex
5
```

Nếu một interface được khai báo mà không nhận giá trị nào cả, mặc định value, và type của interface là <nil> và nếu ta truy xuất phương thức bằng interface chưa được hiện thực, lỗi run-time error sẽ xuất hiện.

Ngoài ra, Go còn có một định nghĩa là empty interface. Trong khai báo, chúng ta sẽ sử dụng từ khóa *interface()*. Một empty interface có thể nhận bất cứ kiểu dữ liệu nào có nghĩa. Ví dụ:

```
package main

import "fmt"

func main() {
    var i interface{}
    fmt.Printf("(%v, %T)\n", i, i) // (<nil>, <nil>)

    i = 42
    fmt.Printf("(%v, %T)\n", i, i) // (42, int)

    i = "hello"
    fmt.Printf("(%v, %T)\n", i, i) // (hello, string)
}
```

Một công cụ khác Go cung cấp trong interface đó là kiểm tra type đang giữ thông qua cú pháp *t := i.(T)* hay *t, ok := i.(T)* Ví dụ:

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"

    s := i.(string) // accept
    fmt.Println(s)

    s, ok := i.(string) // accept
    fmt.Println(s, ok)

    f, ok := i.(float64) // accept
    fmt.Println(f, ok)

    k := i.(float64) // panic
    fmt.Println(k)
}
```

Ý nghĩa cú pháp trên đó là, Go sẽ kiểm tra kiểu dữ liệu T, nếu nó khớp với kiểu dữ liệu interface i đang giữ thì sẽ gán giá trị đó vô biến khởi tạo, còn nếu không sẽ xảy ra lỗi. Nếu ta sử dụng cú pháp sau, nó sẽ hoạt động tương tự với phương thức kiểm tra key ở map, Go cũng sẽ kiểm tra kiểu dữ liệu T, nếu khớp, sẽ trả ra giá trị ở biến đầu và true ở biến sau, còn không, sẽ trả ra giá trị 0 và false.

Một cách khác để kiểm tra kiểu dữ liệu mà interface đang nắm giữ bằng khối lệnh tương tự sau:

```
package main

import "fmt"

func do(i interface{}) {
    switch v := i.(type) {
    case int:
```

```
8           fmt.Printf("Twice %v is %v\n", v, v*2)
9       case string:
10          fmt.Printf("%q is %v bytes long\n", v, len(v))
11      default:
12          fmt.Printf("I don't know about type %T!\n", v)
13      }
14  }
15
16  func main() {
17      do(21)
18      do("hello")
19      do(true)
20  }
```

Ở đây, khác với kiểu dữ liệu T ở phần trên, sử dụng từ khóa type và Go sẽ kiểm tra kiểu dữ liệu của interface khớp với kiểu dữ liệu nào trong khối switch-case.

Kết quả:

```
1  Twice 21 is 42
2  "hello" is 5 bytes long
3  I don't know about type bool!
```

# CHƯƠNG 3
# CONCURRENCY

## 3.1 Goroutine

Go là một ngôn ngữ được hướng tới việc xử lí đa luồng, và thực hiện trên nhiều core nên Go có những công cụ để hỗ trợ việc thực thi đồng thời. Goroutine là một trong số đó.

Giả sử bạn có 2 công việc W1 và W2. Nếu bạn code một cách bình thường, trình biên dịch sẽ compile code theo thứ tự từ trên xuống từng dòng, tức là khi hoàn thành công việc W1 sẽ tiếp tục làm W2. Tuy nhiên bạn muốn chúng thực thi đồng thời với nhau, Go hoàn toàn có thể hỗ trợ việc chạy song song 2 công việc trên 2 Goroutine với từ khóa *go*. *Note: Bình thường, các công việc trên hàm main được thực hiện trên một goroutine.*

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func say(s string) {
9      for i := 0; i < 5; i++ {
10         time.Sleep(100 * time.Millisecond)
11         fmt.Printf("%v->", s)
12     }
13 }
14
15 func main() {
16     go say("world")
17     say("hello")
18 }
19 //One of the output
20 //world->hello->hello->world->world->hello->hello->world->world->hello->
```

Kết quả của chương trình sẽ là một chuỗi các chuỗi"hello"và "world"nhưng ta không thể biết được kết quả sẽ như thế nào, vì khi thực hiện song song, công việc được thực thi không quan tâm tới công việc còn lại và khi goroutine của hàm main kết thúc, chương trình sẽ kết thúc. Cũng do đó, nếu ta dùng hai từ khóa go cho hai hàm say("hello") và say("world") ở ví dụ trên thì chương trình sẽ kết thúc ngay mà không print ra gì cả do 2 goroutine vừa được khởi tạo chưa thực hiện thì hàm main() đã kết thúc rồi.

Bởi vì vậy, Golang cũng hỗ trợ thêm cho chúng ta package "sync"với rất nhiều hàm được định nghĩa sẵn giúp chúng ta kiểm soát công việc thực thi của các goroutine để đạt được kết quả mong muốn như: sync.WaitGroup - thông báo việc thực thi của các goroutine, sync.Mutex hay sync.RWMutex để làm các khóa (vì khi chương trình chạy song song, các quá trình có thể xài chung một dữ liệu - shared data, khiến cho kết quả cuối cùng có thể không được như mong muốn..)

Ví dụ về cách hoạt động:

```
package main

import (
    "fmt"
    "sync"
)

var wg = sync.WaitGroup{}
var counter = 0
var m = sync.RWMutex{}

func main() {
    fmt.Println("Goroutine ----------")

    // MUTEX
    for i := 0; i < 5; i++ {
        wg.Add(2)
        m.RLock() // lock to read
        go SayHello()
        m.Lock() // lock to write
        go Increment()
    }
    wg.Wait()
    fmt.Println("Done Program")
}

func SayHello() {
    fmt.Printf("Hello #%v\n", counter)
    m.RUnlock()
    wg.Done()
}

func Increment() {
    counter++
    m.Unlock()
    wg.Done()
}
```

Kết quả:

```
Goroutine ----------
Hello #0
Hello #1
Hello #2
Hello #3
Hello #4
Done Program
```

Nếu ta bỏ đi những cái khóa, giá trị biến chung counter được đưa vào các goroutine sẽ không như mong muốn, lúc đó, kết quả chương trình sẽ không còn như mong đợi.

## 3.2 Channel

Một công cụ khác hỗ trợ cho concurrency trong Go là channel và bufferd channel. Channel là một kiểu công cụ có thể gửi và nhận thông qua toán tử '<-'. Channel được khởi tạo bởi make với từ khóa *chan*.

```
ch := make(chan int) // channel ch receive and send int value
ch <- v // ch receive v
v := <-ch // ch send v; assign value ch hold to v
          // receiver will wait until channel have value
```

Tuy nhiên nếu chỉ khai báo như vậy, channel chỉ nhận được một giá trị, nên nếu muốn channel tiếp tục nhận giá trị mới, thì giá trị cũ phải được gửi đi thì mới có chỗ cho giá trị mới. Nếu không, Go sẽ báo lỗi deadlock:

```
fatal error: all goroutines are asleep - deadlock!
```

Lỗi này cũng xảy ra nếu chúng ta truyền vào channel một giá trị nhưng không sử dụng tới chúng.

Để khắc phục tình trạng trên, Go cho phép channel được buffer, do đó, có thể khởi tạo giá trị buffer thông qua tham số thứ 2 ở hàm khởi tạo channel.

```
ch := make(chan int, n)
```

Với cách làm này, channel sẽ nhận được giá trị cho tới khi đủ n số.

Ở buffer channel, Go hỗ trợ hàm range, trả về hai giá trị tương tự như map hay slice, biến đầu tiên sẽ nhận giá trị tại vị trí hiện tại của channel, biến thứ hai kiểu bool, sẽ trả về liệu phía sau còn có thể nhận thêm giá trị không.

Tuy nhiên sẽ có lúc giá trị buffer lớn hơn số lượng giá trị channel cần giữ trong thực tế, vậy nên khi channel đã nhận đủ giá trị, hãy dùng lệnh close(channel) để đóng channel lại và cũng đảm bảo hàm range hoạt động chính xác. Bên cạnh đó, nếu ta thêm vào số lượng giá trị vượt quá buffer, hay có nhiều receiver hơn số lượng giá trị buffer, đều có thể dẫn đến lỗi deadlock phía trên.

Trong Golang để mọi thứ có thể trực quan, dễ hiểu hơn, thường channel sẽ có những goroutine riêng chỉ chuyên nhận hoặc chuyên gửi.

```
// Cu phap
go func(ch <-chan int) {
}(ch) // Only receiver from channel
go func(ch chan<- int) {
}(ch) // Only send to channel
```

Nếu sử dụng sai mục đích, Go sẽ báo lỗi tương tự như sau:

```
invalid operation: cannot receive from send-only channel ch
(variable of type chan<- int)
```

# CHƯƠNG 4

# ĐIỂM MẠNH - ĐIỂM YẾU

## 4.1 Điểm mạnh

1. Cú pháp khá tương đồng với C/C++

Trong thời điểm mà hầu như ai cũng bắt đầu với ngôn ngữ lập trình là C/C++, thì các cú pháp tương tự trong Golang sẽ trở nên dễ dàng tiếp cận hơn. Bên cạnh đó, cú pháp của Go cũng được tinh giản hơn, qua đó, có thể tránh những lỗi cú pháp nhỏ có thể xảy ra.

Ngoài ra, việc cũng là một compiled language, nên tốc độ biên dịch của Go cũng như hiệu suất của nó cũng nhanh chóng và cao hơn so với một số ngôn ngữ như Java, Python,...

2. Garbage Collector

Điểm này đề cập tới khả năng quản lí bộ nhớ của Go. Nếu trong lập trình C/C++ việc phải quan tâm tới cấp phát và hủy bộ nhớ là khá phức tạp, thì Go hỗ trợ khả năng quản lí bộ nhớ tự động, giúp có trải nghiệm tốt hơn ở trong việc lập trình bình thường hay cả ở trong việc thực thi đồng thời.

3. Các gói package, thư viện chuẩn, và ngôn ngữ mã nguồn mở

Việc là ngôn ngữ được đóng góp và phát triển bởi cộng đồng mã nguồn mở, nên có thể nói, từ thư viện chuẩn, tới các package trong Go đều được xây dựng và hỗ trợ rất nhiều cho người sử dụng.

Cùng với đó, những tài liệu, thông tin về các thư viện, cũng như các hàm được định nghĩa trước đều có thể dễ dàng tìm được trên trang chủ `go.dev`

4. Concurrency Support

Go hỗ trợ rất nhiều cho việc thực thi đồng thời. Nếu như ở ngôn ngữ khác, ví dụ như Java, để khởi tạo một thread cần trải qua nhiều bước khai báo thì chỉ cần với từ khóa *go*, Golang sẽ hỗ trợ khởi tạo những Goroutine phục vụ cho công việc. Ngoài ra các đặc trưng khác của Go như channel, hay những hàm, định nghĩa trong package "sync cũng hỗ trợ"rất nhiều cho điểm mạnh này,

Với việc phần cứng ngày nay ngày càng đòi hỏi khả năng thực thi song song, hệ thống đa nhân, và trong khi nhiều ngôn ngữ gặp vấn đề về tốc độ trong việc thực thi đồng thời trên nhiều thread, thì Go đang giúp ích cũng như hưởng lợi rất nhiều với khả năng mô phỏng các quá trình trên nhiều thread từ các goroutine.

## 4.2 Điểm yếu

Golang là một ngôn ngữ còn mới nên tất nhiên còn rất nhiều điểm có thể phát triển nữa ở sau này. Và cũng vì Golang là ngôn ngữ trẻ nên những người sử dụng chúng chưa thể nào nhiều như những ngôn ngữ lâu đời khác.

1. Đôi khi quá đơn giản

Tuy đây là điểm mạnh, nhưng cũng là điểm yếu của Golang. Việc quá đơn giản đôi khi khiến người dùng có thể quên đi những điều cơ bản trong lập trình nếu so với những người sử dụng các ngôn ngữ lâu đời khác.

2. Go không hỗ trợ thao tác toán trên con trỏ

Nếu như trong C/C++ việc sử dụng con trỏ cùng toán tử '+' để trỏ tới các ô nhớ tiếp theo là được phép, thì Go lại không hỗ trợ chức năng này. Ví dụ, ở ngôn ngữ C/C++, ta khai báo con trỏ $p$, ta có thể sử dụng $p+1$ để chỉ tới ô nhớ kế tiếp, tuy nhiên Go không hỗ trợ câu lệnh như vậy.

3. Go không có VM (virtual machine) hỗ trợ

VM là công cụ hỗ trợ chạy các chương trình của từng ngôn ngữ. Do chưa có công cụ VM riêng, nên những file Golang thường tốn bộ nhớ lớn hơn so với các ngôn ngữ có VM hỗ trợ.

Tuy nhiên, Google đang tìm cách cải thiện vấn đề này, nên trong tương lai, điểm yếu này sẽ sớm được khắc phục.

4. Không có thư viện GUI

Hiện tại, Go không có thư viện GUI, điều đó có nghĩa bạn sẽ tốn tài nguyên và thời gian để giải quyết những bài toán liên quan tới vấn đề này.