Golang Training Course

Ngày: 31th May 2021

Outline

Input

- Cài đặt môi trường
    - Go
    - Visual Studio Code
    - Visual Studio Code Extension
    - Docker Engine
    - Go Debug và Go Tools
    - Golang CI Lint
- Hiểu môi trường làm việc của Golang
- Tại sao dùng Docker?
- Sử dụng Makefile
- Viết chương trình Hello World
- Hướng dẫn chạy Debug với Visual Studio Code
- Sử dụng Variables, Data Type, Constrants trong Golang

Output

- Cài đặt được môi trường, và các công cụ làm việc cần thiết
- Viết được chương trình Hello World
- Biết được cách Debug
- Sử dụng được Variables, Data Type, Constrants trong Golang

Content

Cài đặt môi trường

- Tải VSCode tại: https://code.visualstudio.com/
- Cài đặt Golang: https://golang.org/doc/install (Lưu ý nếu cài đặt ở Windows, cần thay đổi thư mục cài đặt, mặc định sẽ cài ở ổ C)
- Cài đặt Docker: https://docs.docker.com/engine/install/
- Cài đặt thêm các Extension cho Visual Studio Code bao gồm:
    - VSCode Extension:
    - Beautify
    - Code Runner
    - Git History
    - gitignore
    - Go
    - Prettier - Code formatter
    - YAML

- Cài đặt Go Debug tại: https://github.com/golang/vscode-go/blob/master/docs/debugging.md/
  - Mở VSCode, và bấm tổ hợp (Windows/Linux: Ctrl+Shift+P; OSX: Shift+Command+P)
  - Gõ và tìm `Go: Install/Update Tools`
  - Sau đó chọn `dlv`
  - Cài đặt thêm các thư viện khác, bằng cách chọn tương như `dlv`, các thư viên quan trọng bao gồm: `gopkgs`, `gopls`, `staticcheck`
- Cài đặt Golang CI Lint: https://golangci-lint.run/usage/install/#local-installation (Có thể sử dụng `go get github.com/golangci/golangci-lint/cmd/golangci-lint@v1.40.1` để cài đặt)

Môi trường làm việc Golang

Khi cài đặt Golang bạn sẽ có 3 thư mục bao gồm:

- `bin`: Nơi lưu trữ các file Binary được biên dịch và cài đặt bởi Go (ví dụ `dlv`)
- `pkg`: Nơi chứa các thư viện thứ ba được cài đặt từ Github, hoặc các Go Registry khác
- `src`: Nơi chứa Source Code của các dự án tại máy của bạn (khi khởi tạo và phát triển một dự án mới, bạn phải tạo tại thư mục này, ví dụ `mkdir go-training`)

Khi làm việc với Golang, bạn cần quan tâm đến các giá trị biến môi trường (Environment Variables) quan trọng bao gồm:

- `GOPATH`: địa chỉ thư mục chứa các thư mục `bin`, `pkg`, `src`
- `GOBIN`: địa chỉ chứa thư mục `bin`, thường là `GOPATH/bin`
- `GO111MODULE`: giá trị mặc định `on`, dùng để sử dụng quản lý các dependencies bởi Go Modules
- `GOOS`: hệ điều hành mặc định để Golang compile ra file binary có thể sử dụng được
- `GOARCH`: kiến trúc kernal (ví dụ amd64, arm, dựa vào CPU)

Để xem toàn bộ giá trị này có thể sử dụng lệnh: `go env`

Tại sao cần dùng Docker?

- Nếu bạn chưa biết đến Docker là gì, thì Docker là một công nghệ dùng để Containerization (Container hoá), đây là một trong những công nghệ mà hầu hết các công ty hướng đến để triển khai phần mềm trên quy mô lớn.

- Trước khi có Docker, hầu hết các ứng dụng sẽ được triển khai với Virtual Machine. Virtual Machine là công nghệ ảo hoá, hiện nay hầu hết sử dụng công nghệ của VMWare.

- Bạn có thể hiểu khi triển khai một ứng dụng chúng ta cần Server, Server được hiểu là các tài nguyên, là máy tính, máy trạm để cài đặt và chạy ứng dụng. Tuy nhiên, ở mỗi máy trạm, máy tính bị phụ thuộc vào một hệ điều hành nhất định, mà mỗi ứng dụng chúng ta chạy cũng yêu cầu chạy trên một môi trường nhất định luôn. Giả sử chúng ta không thể chạy một ứng dụng viết bằng C# ASP.NET trên 1 Server Linux được, mà chỉ thể chạy trên Windows. Ngoài ra, việc một ứng dụng chạy thường cũng sẽ cần giới hạn tài nguyên nhất định, để tránh trường hợp dùng vượt quá tài nguyên cho phép và gây ra sự cố. Chúng ta nói đến ở đây là giới hạn.

- Để khắc phục việc đó, chúng ta sử dụng VMWare để thực hiện ảo hoá, VMWare hỗ trợ tạo ra nhiều môi trường hệ điều hành, và giúp bạn chỉ đính một lượng tài nguyên nhất định cho mỗi môi trường. Ta có thể hiểu, VMWare giúp chúng ta tạo thêm các máy con trong một máy mẹ, để chạy ứng dụng.

- Tuy nhiên, việc triển khai với VMWare khá tốn thời gian, và nhiều bước, sau này thì người ta hướng đến việc triển khai một cái gì đấy nhanh hơn, vì sau này có Container. Container bản chất là bạn sẽ chạy ứng dụng và hệ điều hành mà ứng dụng cần chạy đã được đóng gói (Snapshot/Image). Trong VMWare khi triển khai mở rộng ngang, gọi là thêm một máy tương tự cho 1 ứng dụng, thì chúng ta cũng thực hiện Backup, Snapshot, nhưng khá mất thời gian.
- Việc sử dụng Docker/Container sẽ giảm thời gian chúng ta chạy 1 ứng dụng, và chúng ta cũng không phải quan tâm việc cài đặt VMWare, cài đặt các trình tự để chạy một ứng dụng như nào, vì tất cả đã được đóng gói và sẵn sàng để chạy.
- Việc sử dụng Docker là cách tốt nhất để giảm thời gian cho việc bạn muốn chạy các dịch vụ để phát triển ứng dụng ở máy ví dụ (Database với PostgreSQL, Caching với Redis, Message Queue với RabbitMQ). Bạn chỉ cần chạy với Docker, còn chạy như nào thì Docker Image đã được đóng gói và mô tả sẵn rồi.
- Bạn có thể tham khảo chạy với Docker-Compose, ví dụ chạy với một Script như sau:

```
version: "3"

networks:
  go-micro:
    driver: bridge

volumes:
  postgresql:
    driver: local
  pgadmin:
    driver: local
  rabbitmq:
    driver: local
  redis:
    driver: local

services:
  # PostgreSQL Database
  postgresql:
    container_name: postgresql
    image: postgres:10-alpine
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
      PGDATA: /data/postgresql
    volumes:
      - postgresql:/data/postgresql
    ports:
      - "5432:5432"
    networks:
      - go-micro
    restart: unless-stopped

  # PostgreSQL Admin
```

```
  pgadmin:
    container_name: pgadmin
    image: dpage/pgadmin4:5
    depends_on:
      - postgresql
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: admin
      PGADMIN_CONFIG_SERVER_MODE: "False"
    volumes:
      - pgadmin:/root/.pgadmin
    ports:
      - "5050:80"
    networks:
      - go-micro
    links:
      - postgresql
    restart: unless-stopped

  # RabbitMQ (Pub/Sub)
  rabbitmq:
    container_name: rabbitmq
    image: bitnami/rabbitmq:3.8.16-debian-10-r19
    environment:
      RABBITMQ_USERNAME: admin
      RABBITMQ_PASSWORD: admin
    volumes:
      - rabbitmq:/bitnami
    ports:
      - "4369:4369"
      - "5672:5672"
      - "25672:25672"
      - "15672:15672"
    healthcheck:
      interval: 10s
      retries: 12
      test: curl --write-out 'HTTP %{http_code}' --fail --silent --output
/dev/null http://localhost:15672/
    networks:
      - go-micro
    restart: unless-stopped

  # Redis (Caching)
  redis:
    container_name: redis
    image: redis:6.2.3-alpine
    environment:
      ALLOW_EMPTY_PASSWORD: "yes"
    volumes:
      - redis:/data
    ports:
      - "6379:6379"
    networks:
```

```
      - go-micro
    restart: unless-stopped
```

- Sử dụng lệnh `docker-compose up -d` là có thể chạy một loạt chương trình lên, việc này sẽ giúp tiết kiệm thời gian, và công sức rất nhiều.

Sử dụng Makefile

- Khi làm việc với Golang, bạn sẽ phải sử dụng khá nhiều lệnh Command, có thể khiến bạn quên, hoặc không quen khi dùng, việc sử dụng Makefile sẽ giúp bạn rút gọn lại các tập lệnh thông qua các lệnh ngắn hơn, ví dụ: `make run` dùng để thay thế cho `go run main.go`.
- Makefile thường được cài đặt mặc định trên Linux/Unix, nếu bạn dùng Windows có thể cài đặt `Choco` ở trên Powershell, sau đó sử dụng `choco install make`, xem thêm tại đây.
- Một ví dụ dùng với Makefile:

```
.PHONY: run fmt install-lint lint

run:
    @go run main.go

fmt:
    @gofmt .

install-lint:
    @echo "Installing CI Lint..."
    @go get github.com/golangci/golangci-lint/cmd/golangci-lint@v1.40.1

lint:
    @echo "Lint code"
    @golangci-lint run -v
```

- Một số cơ bản khi dùng với Makefile:
  - `.PHONY` dùng để khai báo các chương trình bạn sẽ dùng khi gọi với `make`
  - Để viết một chương trình bạn bắt đầu với việc gọi tên chương trình và `:`, sau đó enter để viết chương trình (lưu ý code cần thụt vào bên trong, sử dụng Tab), ví dụ:

```
.PHONY: print
print:
    @echo "Hello World"
```

  - Toàn bộ code bên trong Make được viết bởi Shell Script, lưu ý sử dụng `@` nếu bạn không muốn hiển thị dòng lệnh, hoặc chương trình khi chạy với Make, mặc định toàn bộ câu lệnh được viết trong Make sẽ được hiển thị khi chạy. Ví dụ:

```
.PHONY: print
print:
    echo "Hello World"
```

Kết quả sẽ là:

```
make print

echo "Hello"
Hello
```

Nếu dùng thêm `@` cho `echo "Hello"` thì dòng lệnh này sẽ không hiển thị

Viết chươn trình Hello World

- Khởi tạo thư mục tại `$GOPATH/src`

```
cd $GOPATH/src
mkdir helloworld
```

- Khởi tạo Go Modules (go.mod). Go Modules được khởi tạo sẽ sử dụng phiên bản Go mới nhất đang được cài đặt ở máy

```
go mod init
```

- Cấu trúc một Go Modules như sau:

```
module go-training

go 1.16
```

  - `module go-training`: Tên Module để quản lý, mặc định sẽ lấy tên folder ở local, tuy nhiên nếu muốn làm 1 module remote (được lưu trữ trên 1 git repository), chúng ta cần phải thực hiện thay đổi tên, ví dụ `github.com/fpt-training/go-training` thì sau này toàn bộ source code của modules này sẽ được quản lý và fetch từ remote repository trên
  - `go 1.16`: Là version golang sẽ sử dụng để biên dịch chương trình

- Viết `main.go`

```
package main
```

```
import "fmt"

func main() {
    fmt.Println("Hello World")
}
```

Thực hiện chạy với lệnh `go run main.go`, kết quả trả về là:

```
Hello World
```

- Lưu ý: Toàn bộ chương trình Main phải luôn sử dụng với `package main` và `func main() {}`, thư viện sử dụng sẽ được import trực tiếp thông qua `import`. Ví dụ có thể import nhiều thư viện một lúc như sau:

```
import (
    "fmt"
    "strings"
)
```

- Ví dụ một chương trình chia nhiều package:
  - `cmd/handler.go`

```
package cmd

func HelloWorld() string {
    return "Hello World"
}
```

  - `main.go`

```
package main

import (
    "fmt"
    "go-training/cmd"
)

func main() {
    fmt.Println(cmd.HelloWorld())
}
```

Thực hiện chạy với `go run main.go`, kết quả trả về:

```
Hello World
```

Hướng dẫn sử dụng Debug với Visual Studio Code

- Sau khi cài đặt `dlv` chúng ta có thể thực hiện chạy Debug.
- Để thực hiện Debug cho một chương trình đơn giản như sau:

```
package main

import (
    "fmt"
)

func main() {
    s := "Hello World" #Đặt Breakpoint
    fmt.Println(s)
    s = "Golang is awesome"
    fmt.Println(s)
}
```

  - Ta thực hiện đặt Breakpoint, bằng cách Click vào dòng và gần với cột Line Number của Editor. Nếu hiện màu đỏ lên thì có nghĩa đặt Break Point thành công.
  - Sau đó tại IDE, nhấn vào Run and Debug ở bên thành Toolbar bên trái, hoặc tổ hợp phím (Shift + Command + D) hoặc (Ctrl + Shift + D)
  - Tại lúc này chúng ta nhấn vào nút Play `(>)` (Launch Package). Thời điểm này VSCode lần đầu tiên sẽ yêu cầu tạo file `launch.json`, thực hiện nhấn tạo với nút `create launch.json`, file này sẽ được tạo tại thư mục `.vscode/launch.json`. Ví dụ:

```
{
  // Use IntelliSense to learn about  possible attributes.
  // Hover to view descriptions of existing   attributes.
  // For more information, visit: https:/ go.microsoft.com/fwlink/?
linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Launch Package",
      "type": "go",
      "request": "launch",
      "mode": "auto",
      "program": "${fileDirname}"
    }
  ]
}
```

Sau khi tạo xong, thì có thể nhấn lại và chạy Debug như bình thường.

- Trường hợp muốn thêm Watch, có thể làm việc với Menu Watch bên trái, bằng cách thêm Expression. Giả sử muốn xem biến `s` có thể nhấn `Add Expression` và thêm `s`. Khi thực hiện Debug, giá trị Variable lúc chạy cũng hiển thị ở cột `Variables`
- Trường hợp muốn Debug với chương trình có chạy thêm biến môi trường (Environment Variables), hoặc Arguments thì thêm vào `configurations` như sau:

```
"configurations": [
          {
              "name": "Launch Package",
              "type": "go",
              "request": "launch",
              "mode": "auto",
              "program": "${fileDirname}",
              "env": {
                  "My_Env_1": "Environment 1",
                  "My_Env_2": "Environment 2"
              },
              "args": [
                  "1",
                  "Golang"
              ]
          }
      ]
```

Thử với chương trình để lấy 2 giá trị này ra như sau:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println(os.Getenv("My_Env_1"))
    fmt.Println(os.Getenv("My_Env_2"))
    fmt.Println(os.Args[1])
    fmt.Println(os.Args[2])
}
```

Sử dụng Variables, Data Type, Const

- Golang hỗ trợ cách khai báo tường minh (Explicit), và không tường minh (Implicit) như sau:

```
package main

import "fmt"

func main() {
    //Explicit
    var s string = "Hello World"
    var b bool = true
    var i int = 1
    //Implicit
    var s = "Hello World"
    //Implicit without var
    s := "Hello World"
    // Blank variable
    var s string (Giá trị mặc định là "")
    var b bool (Giá trị mặc định false)
    var i int (Giá trị mặc định 0)
    // Best practices khi khai báo nhiều variables cùng một lúc
    var (
        s string = "Hello World"
        b bool = false
        i = 1
    )
    // Khai báo nhiều variables cùng 1 giá trị theo cách
Explicit/Implicit
    var a, b, c string = "a", "b", "c"
    a, b, c := "a", "b", "c"
    var a, i, b = "a", 1, true
    a, i, b := "a", 1, true
    // Không được dùng
    var a int, b string = 1, "a"
}
```

- Golang có hỗ trợ Ghost Variables như sau:

```
    package main

    import "fmt"

    func main() {
        s := "Hello World"
        func() {
            s := "Go is awesome"
            fmt.Println(s)
        }
        fmt.Println(s)
    }
```

Khi thực hiện chạy câu lệnh này kết quả sẽ là:

```
Go is awesome
Hello World
```

Ghost Variables là biến được khởi tạo có cùng tên với biến đã được khai báo trước đó, dùng trong các trường hợp muốn tái sử dụng tên, nhưng không muốn đặt lại giá trị, giả sử kết quả trả về của một Function đã được gọi.

```
    func func_return_error() error{
        //Do Some Things
        return errors.New("my error")
    }

    func returnError() error {
        err := errors.New("this is my error")
        if err := func_return_error(); err != nil {
            return err
        }
        return err
    }
```

Kết quả chương trình này nếu function `func_return_error()` trả về kết quả không bị nil (trống) thì kết quả trả về sẽ khác với error `this is my error`, là `my error`

- Golang hỗ trợ sử dụng `_` để không nhận giá trị trả về của một biến khi gọi đến một function, ví dụ:

```
package main

import ...

func error_is_nil() (int, error) {
    return 1, nil
}

func error_not_nil() (int, error) {
    return 0, errors.New("error")
}



func main() {
    //Không phải thực hiện Error Handling trường hợp Error luôn luôn
là Nil)
    i, _ := error_is_nil()
    //Trường hợp error có thể xảy ra, vẫn phải thực hiện error
handling
    i, err := error_not_nil()
    if err != nil {
        //do some thing
```

```
    }
}
```

_ Sử dụng khi bạn không muốn nhận giá trị trả về của biến variable để xử lý tiếp, trường hợp này, giá trị trả về là 1 error, nhưng bạn chắc chắn nó luôn là `nil` nên không muốn phải thực hiện `error handling`

- Data Type ở trong Golang sẽ có nhiều lựa chọn hơn, để tối ưu cho Kernal, và bộ nhớ. Ví dụ bạn có thể giới hạn giá trị mà một biến có thể được chứa: `var i uint8`, giá trị variable này sẽ chỉ nhận từ 1 - 255. Trường hợp đặt giá trị nhiều hơn 255, chương trình complie sẽ lỗi. Thường trong Golang sẽ sử dụng các giá trị numeric đều là 64bit, ví dụ `int64`, `float64`
- Để Convert String sang Numberic, hoặc Boolean thì sử dụng `strconv`, ví dụ:

```
package main

import "strconv"

func main() {
    s1 := "10"
    i, err := strconv.ParseInt(s1, 10, 64) //Mặc định giá trị trả về là int64, nếu sử dụng 32bit thì thay đổi 64 thành 32, tương tự 8bit)
    //i, err := strconv.ParseInt(i, 10, 32)
    if err != nil {
        //Do some thing
    }
    fmt.Println(i)
    //Trường hợp lỗi như sau sẽ xảy ra:
    s2 := "999999"
    y, err := strconv.ParseInt(s2, 10, 8)
    //Đoạn này cần phải thực hiện Error Handling, vì giá trị convert đang gặp lỗi out of range
    if err != nil {
        panic(err.Error())
    }
    fmt.Println(y)
}
```

- Một số trường hợp ép kiểu có thể được làm như sau:
    - Đối với Numberic có thể convert 64 sang 32 ví dụ:

```
package main

import "fmt"

func main() {
  var i int64 = 1
```

```
    fmt.Println(int(i))
}
```

Tuy nhiên đối với String toàn bộ phải dùng strconv. Trừ trường hợp convert bytes -> string thì có thể dùng `string(byte[])`

- Để sử dụng Constrant có thể khai báo như sau:

```
package main

import "fmt"

const (
    ERROR_HTTP_401 = "Unauthorized"
    ERROR_HTTP_500 = "Server Internal Error"
    ERROR_BOOL     = bool
)

func main() {
    fmt.Println(ERROR_HTTP_401)
    fmt.Println(ERROR_HTTP_500)
    fmt.Println(ERROR_BOOL)
}
```

- Để phân biệt Public/Private Variables sẽ dựa vào Uppercase/Lowercase của Variables ví dụ:

```
package cmd

var Public_Variable string = "This is Public Variable"

var private_variable string = "This is private variable"

---

package main

import (
    "fmt"
    "go-training/cmd"
)

func main() {
    fmt.Println(cmd.Public_Variable)
    fmt.Println(cmd.private_variable) //Lỗi, không compline được
}
```

Các tài liệu tham khảo thêm:

- Go Pkg: https://golang.org/pkg/strconv/
- Coding Convention: https://golang.org/doc/effective_go