Introduction à Go – TD0

Franklin ”Snaipe” Mathieu

2023

1 Installation

Cette formation utilise Go 1.21.3 pour ses travaux dirigés, et assume un environnement Linux x86_64.

Connectez-vous sur votre environnement de développement, ouvrez un terminal, et lancez ces commandes:

```
$ curl -LO https://go.dev/dl/go1.21.3.linux-amd64.tar.gz
_ % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    75  100    75    0     0    328      0 --:--:-- --:--:-- --:--:--   330
100 63.5M  100 63.5M    0     0  17.1M      0  0:00:03  0:00:03 --:--:-- 18.6M
$ sudo tar -C /usr/local -xzf go1.21.3.linux-amd64.tar.gz
$ rm -f go1.21.3.linux-amd64.tar.gz
```

Go est maintenant installé sous `/usr/local/go`.

Pour ajouter la commande go dans le PATH:

```
$ echo 'export PATH="$PATH:/usr/local/go/bin"' >> ~/.profile && source ~/.profile
```

Vous pouvez maintenant confirmer que Go est bien installé et fonctionnel:

```
$ go version
go version go1.21.3 linux/amd64
```

2 Utiliser Go

2.1 go mod init

Dans cette section, nous allons couvrir l’utilisation de la plupart des outils standard de Go sur un simple projet ”Hello, World!”

```
$ mkdir hello
$ cd hello
$ go mod init example.com/hello
go: creating new go.mod: module example.com/hello
```

`go mod init` initialise le projet local avec une définition de module dans un fichier go.mod:

```
$ ls
go.mod
$ cat go.mod
module example.com/hello

go 1.21.3
```

Le fichier `go.mod` contient le nom du module, sa version de Go minimale supportée, ainsi que tout autre dépendance sur des modules externes (non-visibles ici).

2.2 go build

Écrivons un programme pour dire bonjour; créez un fichier nommé `hello.go`, avec ce contenu:

```
package main

import "fmt"

func main() {
    fmt.Println("Bonjour tout le monde!" )
}
```

Vous pouvez compiler et lancer ce programme:

```
$ go build
$ ls
go.mod  hello  hello.go
$ ./hello
Bonjour tout le monde!
```

2.3 go run

`go run` est une version plus courte de `go build && ./hello`:

```
$ go run hello.go
Bonjour tout le monde!

$ go run hello.go arg1 arg2 ...
Bonjour tout le monde!
```

2.4 go fmt

`go fmt` est le formatteur de code standard de Go.

hello.go n'est pas correctement formatté (l'indentation utilise 4 espaces au lieu d'une tabulation, et il y a une espace de trop avant la parenthèse de fin du Println); formattons-le avec `go fmt`:

```
$ go fmt
hello.go
$ cat hello.go
package main

import "fmt"

func main() {
        fmt.Println("Bonjour tout le monde!")
}
```

2.5 go doc

Vous pouvez consulter la documentation d'un symbole spécifique à tout moment avec la commande `go doc`:

```
$ go doc fmt.Println
package fmt // import "fmt"

func Println(a ...any) (n int, err error)
    Println formats using the default formats for its operands and writes to
    standard output. Spaces are always added between operands and a newline
    is appended. It returns the number of bytes written and any write error
    encountered.
```

La documentation est aussi consultable en ligne: `https://pkg.go.dev/fmt#Println`

Vous pouvez ajouter de la documentation à vos fonctions et variables exportées en ajoutant un commentaire sur les lignes précédent immédiatement la définition:

```
// Saluer affiche un salut envers le sujet spécifié sur la sortie standard.
//
// Ce commentaire entier sera affiché dans la documentation du paquet.
func Saluer(sujet string) {
    fmt.Println("Bonjour " + sujet + "!" )
}
```

2.6 go test

Go intègre aussi un framework de test dans son outillage: `go test`.

Pour en faire la démonstration, faisons un paquet fizzbuzz:

```
$ mkdir fizzbuzz && cd fizzbuzz
$ go mod init example.com/fizzbuzz
```

Dans un paquet Go, tout fichier suffixé par _test.go est automatiquement compilé dans les tests. Créez d'abord un fichier fizzbuzz.go avec ce contenu:

```
package fizzbuzz

import "fmt"

// FizzBuzz retourne "fizz" si i est divisible par 3, "buzz" si i est divisible par 5,
// "fizzbuzz" si i est divisible par 3 et 5, et retourne la valeur de i autrement.
func FizzBuzz(i int) string {
        switch {
        case i%3 == 0:
                return "fizz"
        case i%5 == 0:
                return "buzz"
        default:
                return fmt.Sprintf("%d", i)
        }
}
```

Vous pouvez écrire des tests en créant un fichier fizzbuzz_test.go:

```go
package fizzbuzz

import "testing"

func TestFizzBuzz(t *testing.T) {
        expected := "fizz"
        actual := FizzBuzz(3)
        if expected != actual {
                t.Fatal("FizzBuzz(3) did not return fizz")
        }
}
```

Toute fonction dans un fichier _test.go dont le nom commence par Test et prenant en argument un `*testing.T` est automatiquement détectée et lancée lors des tests.

Pour lancer les tests du projet, il suffit d'executer `go test`:

```
$ go test
PASS
ok          example.com/fizzbuzz        0.001s
```

Néanmoins, ce test n'est pas parfait; il ne teste qu'une partie superficielle de la fonction FizzBuzz. Nous pouvons tester différents cas de figures via une boucle:

```go
package fizzbuzz

import (
        "fmt"
        "testing"
)

func TestFizzBuzz(t *testing.T) {

        type TestCase struct {
                Input    int
                Expected string
        }

        testcases := []TestCase{
                {
                        Input:    1,
                        Expected: "1",
                },
                {
                        Input:    3,
                        Expected: "fizz",
                },
                {
                        Input:    5,
                        Expected: "buzz",
                },
                {
                        Input:    15,
                        Expected: "fizzbuzz",
                },
        }

        for _, testcase := range testcases {
                t.Run(fmt.Sprintf("%d", testcase.Input), func(t *testing.T) {
                        actual := FizzBuzz(testcase.Input)
                        if testcase.Expected != actual {
                                t.Fatalf("expected %q, got %q instead",
                                        testcase.Expected,
                                        actual,
                                )
                        }
                })
        }
}
```

Ce test fait abstraction des différents cas de figures liés à la fonction FizzBuzz dans une liste de cas. Chaque cas est ensuite itéré, et isolé dans son propre test via `t.Run(...)`. Le premier argument de Run est le nom du test, qui ici est le nombre passé à la fonction FizzBuzz: le nom final du test de `FizzBuzz(123)` sera donc `TestFizzBuzz/123`, par exemple.

Lançons ce nouveau test:

```
$ go test
--- FAIL: TestFizzBuzz (0.00s)
    --- FAIL: TestFizzBuzz/15 (0.00s)
        fizzbuzz_test.go:38: expected "fizzbuzz", got "fizz" instead
FAIL
exit status 1
FAIL        example.com/fizzbuzz        0.002s
```

Oups! Visiblement, la fonction a un défaut. Nous allons voir comment débugger le problème dans la section suivante.

2.7 Débugger son code

Un moyen simple de débugger son code est d'utiliser `fmt.Printf`, mais nous allons voir ici comment utiliser un débuggeur.

gdb n'est pas vraiment utilisable avec Go; l'expérience n'est pas des plus agréables et gdb ne sait pas comment générer certains aspects du langage.

Nous allons installer `delve`, un débuggeur natif Go:

```
$ go install github.com/go-delve/delve/cmd/dlv@latest
go: downloading github.com/go-delve/delve v1.21.1
go: downloading github.com/derekparker/trie v0.0.0-20221213183930-4c74548207f4
go: downloading github.com/go-delve/liner v1.2.3-0.20220127212407-d32d89dd2a5d
go: downloading github.com/google/go-dap v0.9.1
go: downloading go.starlark.net v0.0.0-20220816155156-cfacd8902214
go: downloading github.com/cilium/ebpf v0.11.0
go: downloading github.com/mattn/go-runewidth v0.0.13
go: downloading golang.org/x/exp v0.0.0-20230224173230-c95f2b4c22f2

$ echo 'export PATH="$PATH:$HOME/go/bin"' >> ~/.profile && source ~/.profile
```

Pour lancer les tests via delve:

```
$ dlv test
Type 'help' for list of commands.
(dlv) b TestFizzBuzz
Breakpoint 1 set at 0x55c576 for fizzbuzz.TestFizzBuzz() ./fizzbuzz_test.go:8
(dlv) c
> fizzbuzz.TestFizzBuzz() ./fizzbuzz_test.go:8
     3:        import (
     4:                "fmt"
     5:                "testing"
     6:        )
     7:
=>   8:        func TestFizzBuzz(t *testing.T) {
     9:
    10:                type TestCase struct {
    11:                        Input    int
    12:                        Expected string
    13:                }
```

L'interface ressemble plus-ou moins à gdb; vous pouvez utiliser `break`, `next`, `continue`, et `print` pour naviguer dans le programme et afficher des valeurs diverses. Continuons notre session de debug:

```
(dlv) b FizzBuzz
Breakpoint 1 set at 0x55c3d3 for izzbuzz.FizzBuzz() ./fizzbuzz.go:5
(dlv) cond 1 i == 15
(dlv) c
> fizzbuzz.FizzBuzz() ./fizzbuzz.go:5
     1:        package fizzbuzz
     2:
     3:        import "fmt"
     4:
=>   5:        func FizzBuzz(i int) string {
     6:                switch {
     7:                case i%3 == 0:
     8:                        return "fizz"
     9:                case i%5 == 0:
    10:                        return "buzz"
(dlv) p i
15
(dlv) n
> fizzbuzz.FizzBuzz() ./fizzbuzz.go:7
     2:
     3:        import "fmt"
     4:
     5:        func FizzBuzz(i int) string {
     6:                switch {
=>   7:                case i%3 == 0:
     8:                        return "fizz"
     9:                case i%5 == 0:
    10:                        return "buzz"
    11:                default:
    12:                        return fmt.Sprintf("%d", i)
(dlv) n
> fizzbuzz.FizzBuzz() ./fizzbuzz.go:8
     3:        import "fmt"
     4:
     5:        func FizzBuzz(i int) string {
     6:                switch {
     7:                case i%3 == 0:
=>   8:                        return "fizz"
     9:                case i%5 == 0:
    10:                        return "buzz"
    11:                default:
    12:                        return fmt.Sprintf("%d", i)
    13:                }
```

Visiblement, il manque un cas au switch pour tester que i soit divisible par 3 et 5 simultanément. La résolution du problème est un exercice laissé au lecteur.

3 Premiers pas

Pour votre introduction à Go, nous allons ré-implémenter quelques fonctions du paquet `strings` de la bibliothèque standard: `IndexByte`, `Index`, `Cut`, et `Split`. Faisons un module strings:

```
$ mkdir strings && cd strings
$ go mod init example.com/strings
```

C'est à vous d'implémenter les fonctions suivantes, et d'écrire des tests pour en valider le comportement:

```
package strings // import "strings"

func IndexByte(s string, c byte) int
    IndexByte returns the index of the first instance of c in s, or -1 if c is
    not present in s.

func Index(s, substr string) int
    Index returns the index of the first instance of substr in s, or -1 if
    substr is not present in s.

func Cut(s, sep string) (before, after string, found bool)
    Cut slices s around the first instance of sep, returning the text before and
    after sep. The found result reports whether sep appears in s. If sep does
    not appear in s, cut returns s, "", false.

func Split(s, sep string) []string
    Split slices s into all substrings separated by sep and returns a slice of
    the substrings between those separators.

    If s does not contain sep and sep is not empty, Split returns a slice of
    length 1 whose only element is s.

    If sep is empty, Split splits after each UTF-8 sequence. If both s and sep
    are empty, Split returns an empty slice.

    It is equivalent to SplitN with a count of -1.

    To split around the first instance of a separator, see Cut.
```

Bonus: Implémentez `IndexRune` et `IndexAny`. Notez que ces fonctions travaillent avec de l'utf8; pensez à utiliser le paquet `unicode/utf8`.