Introduction à Go – TD1: Calculateur

Franklin ”Snaipe” Mathieu

2023

1 Bouchées

1.1 Lire sur l’entrée standard

`os.Stdin` représente l’entrée standard en Go, mais n’est pas nécessairement des plus simple à utiliser tel-quel.

Pour lire des lignes sur l’entrée standard, il est plus facile de faire appel au type `bufio.Scanner`. Le package `bufio` implémente des primitives pour lire et écrire avec du ”buffering”.

Faisons un petit programme pour afficher un ficher avec ses numéros de ligne. Dans un fichier lineno.go:

```go
package main

import (
	"bufio"
	"fmt"
	"os"
)

func main() {
	scanner := bufio.NewScanner(os.Stdin)

	lineno := 1
	for scanner.Scan() { // Continue de scanner jusqu'à EOF
		line := scanner.Text()
		fmt.Printf("%d\t%s\n", lineno, line)
		lineno++
	}
	if err := scanner.Err(); err != nil { // S'il y a une erreur, l'afficher
		fmt.Fprintln(os.Stderr, err)
		os.Exit(1)
	}
}
```

Vous pouvez le voir en action avec `go run`:

```
$ echo "foo\nbar\nbaz" | go run lineno.go
1	foo
2	bar
3	baz
```

1.2 Tester son code

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 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:

```
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.

1.3 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.

2 Le calculateur

Pour ce TD, nous voulons implémenter un calculateur infix tel que `bc`:

```
$ bc -q
(1+2)*3
9
```

Autrement dit, il faut que le programme calculateur:

1. Lise une ligne de l'entrée standard
2. Parse la ligne en une expression et évalue cette expression
3. Affiche le résultat sur la sortie standard

Votre calculateur doit supporter les opérateurs `+`, `-`, `*`, `/`, ainsi que les parenthèses. Il est recommandé d'implémenter l'algorithme de Shunting Yard.

Pensez à écrire des tests!