

# Introduction à Go – TD-alt: Busybox

Franklin "Snaipe" Mathieu

2023

Busybox est une implémentation des coreutils Linux (et plus) en un seul exécutable, utilisé principalement dans des environnements minimalistes tels que Alpine Linux.

Dans ce TD, nous allons implémenter de la même manière quelques uns de ces utilitaires.

## 1 Mise en place

Commençons par créer un dossier de projet:

```
$ go mod init example.com/busybox
```

Pour ce projet, nous utiliserons <https://github.com/alecthomas/kong> pour prendre en charge le CLI:

```
$ go get github.com/alecthomas/kong
```

Cette bibliothèque nous permet de facilement définir des options. Pour en montrer le fonctionnement, nous allons créer notre point d'entrée, et implémenter la sous-commande echo. Créez un fichier `main.go`:

```
package main

import (
    "github.com/alecthomas/kong"
)

func main() {
    var cli struct {
        Echo EchoCmd `cmd:""`
    }
    ctx := kong.Parse(&cli)
    ctx.FatalIfErrorf(ctx.Run())
}
```

Ici, la variable `cli` définit la structure de notre ligne de commande. Dans cette configuration, nous n'acceptons qu'une sous-commande, `echo`.

Créez le fichier `echo.go` pour implémenter cette sous-commande:

```

package main

import (
    "fmt"
    "strings"
)

type EchoCmd struct {
    NoNewline bool `short:"n" name:"" `
    Args      []string `arg:"" passthrough:"`
}

func (cmd *EchoCmd) Run() error {
    out := strings.Join(cmd.Args, " ")
    if !cmd.NoNewline {
        out += "\n"
    }
    _, err := fmt.Print(out)
    return err
}

```

La sous-commande echo affiche simplement les arguments qui ont été passés, tout commande unix éponyme; vous pouvez tester son fonctionnement avec `go run`:

```

$ go run . -h
Usage: busybox <command>

Flags:
  -h, --help    Show context-sensitive help.

Commands:
  echo <args> ...

Run "busybox <command> --help" for more information on a command.

$ go run . echo hello world
hello world

$ go run . echo hello world -h
hello world -h

```

## 2 Interagir avec le système

### 2.1 Lire un fichier

En Go, le paquet standard `os` met à disposition des abstractions pour travailler avec le système. `*os.File` est le type standard représentant un fichier.

Pour ouvrir un fichier en lecture seule:

```

f, err := os.Open("/chemin/vers/fichier")
if err != nil {
    return err
}

```

... et inversement, pour créer un fichier (ou en tronquer un existant et l'ouvrir en écriture):

```

f, err := os.Create("/chemin/vers/fichier")
if err != nil {
    return err
}

```

Un fichier possède une méthode `Read` pour lire du contenu:

```
data := make([]byte, 4096)
n, err := f.Read(data) // Lis au plus 4096 octets, retourne le nombre d'octets lus
if err != nil {
    return err
}
```

... et une méthode Write pour écrire dedans:

```
greeting := []byte("hello, world")
n, err := f.Write(greeting) // Écris exactement len(greeting) octets
if err != nil {
    return err
}
```

... et finalement, un fichier doit être fermé pour en libérer les ressources:

```
if err := f.Close(); err != nil {
    // Gérer l'erreur; en général uniquement nécessaire quand
    // le fichier est ouvert en écriture.
    return err
}
```

Notez cependant que si `f.Read` ou `f.Write` retourne un erreur, et que vous décidez de retourner une erreur en retour, alors vous quitterez la fonction sans avoir appelé `f.Close()`. Pour palier à ce genre de situation, il est courant d'utiliser le mot-clé `defer`:

```
f, err := os.Open("/chemin/vers/fichier")
if err != nil {
    return err
}
defer f.Close() // sera exécuté à la sortie de la fonction
```

Ce faisant il n'est donc pas nécessaire de manuellement appeler `f.Close()` sauf si vous voulez vérifier les erreurs (lorsque le fichier est ouvert en écriture).

`os.Stdin`, `os.Stdout` et `os.Stderr`, représentant respectivement l'entrée standard, la sortie standard, et la sortie d'erreur en Go, sont des variables globales de type `*os.File`, et implémentent donc ces mêmes méthodes.

## 2.2 I/O

`*os.File` implémentant les méthodes `Read` et `Write`, il est possible d'utiliser certaines fonctions générales pour faire de l'I/O dessus. Par exemple `io.Copy` du paquet `io` permet de copier l'intégralité d'une source vers une destination; pour copier un fichier vers un autre, on aura donc par exemple:

```
in, err := os.Open("/chemin/vers/source")
if err != nil {
    return err
}
defer in.Close()

out, err := os.Create("/chemin/vers/destination")
if err != nil {
    return err
}
defer out.Close()

_, err = io.Copy(out, in) // copie l'intégralité de in dans out
if err != nil {
    return err
}
```

Le paquet `io` contient d'autres utilitaires pour manipuler des flux de données; par exemple, pour lire l'intégralité d'un fichier dans un `[]byte`:

```
in, err := os.Open("/chemin/vers/source")
if err != nil {
    return err
}
defer in.Close()

buf, err := io.ReadAll(in) // lis l'intégralité de in dans un buffer et le retourne
if err != nil {
    return err
}
```

Le paquet contient toute sortes de fonctions et d'abstractions utiles; n'hésitez pas à en consulter la documentation, et les exemples associés: <https://pkg.go.dev/io>

Un autre paquet utile est le paquet `bufio`: <https://pkg.go.dev/bufio>. Ce paquet est généralement utilisé pour scanner un fichier ligne par ligne, avec notamment le type `bufio.Scanner`. Le paquet `bufio` implémente des primitives pour lire et écrire avec du "buffering".

Par exemple, ce snippet de code lis l'entrée standard et en affiche le contenu, avec le numéro de ligne:

```
scanner := bufio.NewScanner(os.Stdin)

lineno := 1
for scanner.Scan() { // Continue de scanner jusqu'à EOF
    line := scanner.Text()
    fmt.Println(lineno, line)
    lineno++
}

if err := scanner.Err(); err != nil { // S'il y a une erreur, l'afficher
    return err
}
```

Vous devriez avoir suffisamment de contexte pour implémenter les utilitaires `cat` et `cp`. Ajoutez des fichiers `cat.go` et `cp.go` dans votre projet `busybox` et implémentez les types `CatCmd` et `CpCmd`.

Pour `cat`, nous allons implémenter les flags `-n` et `-e`. Pour `cp`, nous n'allons pas implémenter de flag pour l'instant (il ne s'agit que de copier un fichier vers un autre).

Pour rappel, voici ce que font les flags de `cat`:

```

$ echo "hello world" | cat
hello world

$ echo "hello world" > greeting
$ cat greeting
hello world

$ echo "foo" > foo
$ echo "bar" >> foo
$ echo "baz" >> foo
$ cat greeting foo
hello world
foo
bar
baz

$ cat -n greeting foo
 1 hello world
 2 foo
 3 bar
 4 baz

$ cat -e greeting foo
hello world$
foo$
bar$
baz$

```

### 3 Exploration de fichiers

Le parcours du système de fichier se fait principalement via deux façons:

Pour lister les fichiers d'un répertoire, `os.ReadDir` et `os.File.ReadDir`, pour lister le contenu d'un chemin et le contenu d'un dossier qui a été ouvert avec `os.Open` respectivement. Un exemple est disponible sur <https://pkg.go.dev/os#example-ReadDir> dans la documentation.

Pour parcourir récursivement les fichiers sous un répertoire et ses enfants, on utilise généralement `filepath.WalkDir`. Par exemple, pour afficher l'arborescence de fichier sous un répertoire:

```

err := filepath.Walk(racine, func(path string, d DirEntry, err error) error {
    fmt.Println(path)
    return err
})

```

Les valeurs que peuvent prendre les paramètres `path`, `d`, et `err` sont documentées sur `fs.WalkDirFunc`: <https://pkg.go.dev/io/fs#WalkDirFunc>.

Pour mettre ces fonctions en pratique, nous allons implémenter deux sous-commandes de `busybox`: `ls`, pour lister les fichiers, et `du`, pour calculer la taille d'un répertoire et ses enfants.

Pour `ls`, nous n'allons implémenter qu'un seul flag: `-a`, qui nous permet d'afficher les fichiers cachés (ceux qui commencent par un `"."`).

Pour `du`, aucun flag n'est à implémenter. Pour rappel, `du` n'affiche la taille en octets sur la sortie standard que pour les dossiers (et pas les fichiers individuellement).