Les bases du langage Go
Partie 3 : fichiers

Initiation au développement

BUT informatique, première année

Ce TP a pour but de vous présenter les bases du langage Go. C'est ce langage qui sera utilisé pour réaliser tous les exercices de ce cours d'initiation au développement. Il n'y a pas de prérequis particuliers en dehors du contenu des TP précédents (les bases du langage Go partie 1 et partie 2).

Dans tout ce TP un texte avec cet encadrement est un texte contenant une information importante, il faut donc le lire avec attention et veiller à en tenir compte.

Un texte avec cet encadrement est une remarque.

Un texte avec cet encadrement est un travail que vous avez à faire.

Même si la plupart des exercices de ce TP peuvent vous sembler très faciles, faites les tous sérieusement : cela vous permettra de retenir la syntaxe du langage Go et d'être ainsi plus à l'aise lors des prochains TPs.

Si vous souhaitez avoir un autre point de vue sur l'apprentissage du langage Go, vous pouvez consulter le tutoriel officiel *A Tour of Go*[1] ou le site *Go by Example*[2] en faisant toute fois attention au fait que ces sites sont plutôt faits pour les personnes ayant déjà une bonne expérience de la programmation.

1. `https://tour.golang.org/`
2. `https://gobyexample.com/`

1 Écriture de fichiers

Il est pratique de savoir écrire des données dans des fichiers. Ceci peut permettre de sauvegarder l'état d'un programme entre plusieurs utilisations de celui-ci, ou encore d'enregistrer les données de sortie d'un programme pour ensuite les analyser à l'aide d'un autre programme.

Nous allons ici nous concentrer sur la création d'un fichier dans lequel on écrira ensuite, mais il est bien sûr possible d'ouvrir un fichier déjà existant pour écrire dedans (soit après son contenu, soit en effaçant celui-ci, voir même au milieu du fichier).

1.1 Créer un fichier

Pour créer un fichier on peut utiliser la fonction Create du paquet os[3]. Il ne faut pas oublier, une fois qu'on a terminé d'utiliser le fichier, de le fermer avec la fonction Close. Le programme 1 donne un exemple de création de fichier.

```go
package main

import (
    "log"
    "os"
)

func main() {
    var myFile *os.File
    var err error
    myFile, err = os.Create("monfichier")
    if err != nil {
        log.Fatal(err)
    }
    err = myFile.Close()
    if err != nil {
        log.Fatal(err)
    }
}
```

Programme 1 – create.go

La fonction Create prend en argument le nom du fichier à créer. Elle retourne le fichier lui-même (myFile) ainsi qu'une erreur (err). Celle-ci sert à indiquer si jamais il y a eu un soucis en créant le fichier. On doit donc toujours vérifier que ce n'est pas le cas (en vérifiant que l'erreur vaut nil) avant d'utiliser le fichier.

La fonction Close retourne elle aussi une erreur, pour indiquer si le fichier a pu être fermé correctement ou pas. Par ailleurs, vous aurez peut-être noté que cette fonction ne s'utilise pas exactement comme les autres fonctions que vous avez vues jusqu'à présent : on écrit myFile.Close() plutôt que os.Close(myFile). On appelle ce genre de fonction une *méthode* et vous verrez un peu plus tard comment en définir soit même et à quoi elles servent.

On notera aussi l'utilisation du paquet log[4] pour afficher des informations sur le déroulement du programme à l'écran (dans les TPs précédents on utilisait fmt). En particulier, la fonction Fatal de ce paquet permet de terminer le programme en plus d'afficher des informations à l'écran.

> Récupérez le programme create.go sur MADOC et testez-le. Constatez que celui-ci crée effectivement des nouveaux fichiers. Que se passe-t-il si on demande de créer un fichier qui existe déjà ?

3. https://pkg.go.dev/os
4. https://pkg.go.dev/log

1.2 Écrire dans un fichier nouvellement créé

Une fois notre fichier créé on peut écrire dedans avec la fonction Fprintln du paquet fmt. Cette fonction est similaire à Println que nous avons déjà vue, à part qu'elle écrit dans un fichier, qui doit être indiqué dans ses arguments. Le programme 2 donne un exemple d'utilisation de cette fonction.

```go
package main

import (
    "fmt"
    "log"
    "os"
)

func main() {
    var myFile *os.File
    var err error
    myFile, err = os.Create("monfichier")
    if err != nil {
        log.Fatal(err)
    }

    _, err = fmt.Fprintln(myFile, "Bonjour")
    if err != nil {
        log.Fatal(err)
    }

    _, err = fmt.Fprintln(myFile, "Au revoir")
    if err != nil {
        log.Fatal(err)
    }

    err = myFile.Close()
    if err != nil {
        log.Fatal(err)
    }
}
```

Programme 2 – write.go

Ce programme est similaire au programme 1 à la différence qu'entre la création et la fermeture de fichier on utilise la fonction Fprintln. On peut noter que le fichier dans lequel écrire est donné en premier argument de la fonction. Par ailleurs, cette fonction peut retourner une erreur si l'écriture s'est mal passée (ici on la récupère dans err et on vérifie que tout va bien en la comparant à nil). Une autre valeur est retournée, qu'on ne récupère pas ici (on utilise le symbole _ pour cela, car on ne souhaite pas utiliser cette valeur et, comme nous l'avons vu précédemment, toute variable créée en Go doit ensuite être utilisée). Cette valeur indique le nombre d'octets qui ont été écrits dans le fichier.

La fonction Println que nous avons utilisée auparavant retourne en fait elle aussi deux valeurs : le nombre d'octets écrits et une erreur.

Récupérez le programme write.go sur MADOC et exécutez-le. Essayez de modifier différents éléments de ce programme (nom du fichier, contenu des lignes écrites, nombre de lignes écrites) et testez à nouveau. Que se passe-t-il si on utilise ce programme avec un nom de fichier correspondant à un fichier déjà existant et contenant du texte ?

Écrivez un programme qui prend en argument un entier n compris entre 0 et 10 (il faudra signaler si l'entier n'est pas correct) et écrit la table de multiplication de n, bien présentée, dans un fichier (de $0 \times n$ à $10 \times n$).

2 Lecture de fichiers

Il aussi utile d'être capable de lire un fichier et de traiter son contenu. Ceci peut servir en particulier à configurer un logiciel : on crée un fichier de configuration qui contient des informations sur nos préférences (langue, raccourcis, etc) et la première action du logiciel lorsqu'il démarre est de lire ce fichier pour prendre en compte ces préférences. Une autre utilisation courante est de stocker un ensemble de données sous forme d'un fichier (plus ou moins structuré : csv, json, etc) pour pouvoir ensuite les traiter à l'aide d'un programme en lisant le fichier.

2.1 Lire tout un fichier d'un coup

La méthode la plus simple en Go pour récupérer le contenu d'un fichier est d'utiliser la fonction ReadFile du paquet io/ioutil[5]. Cette fonction permet de lire l'intégralité d'un fichier (identifié par son chemin) et de stocker ceci dans un tableau d'octets ([]byte). Les raisons pour lesquelles on obtient un tableau d'octets plutôt que directement une chaîne de caractères sont en dehors du cadre de ce cours (mais vous en parlerez dans vos cours d'architecture et de système). Pour l'instant il vous suffit de vous souvenir qu'on peut transformer ce tableau d'octets t en chaîne de caractères simplement en écrivant string(t). Le programme 3 donne un exemple d'utilisation de cette fonction.

```go
package main

import (
    "io/ioutil"
    "log"
)

func main() {
    var filePath string = "test"
    var data []byte
    var err error
    data, err = ioutil.ReadFile(filePath)
    if err != nil {
        log.Fatal(err)
    }
    log.Print("J'ai lu: ", string(data))
}
```

Programme 3 – readfile.go

Dans ce programme on remarque que la fonction ReadFile retourne deux valeurs : le résultat de la lecture et une erreur. Avant de consulter le résultat de la lecture on doit toujours vérifier que l'erreur vaut nil, indiquant que tout s'est bien passé.

Récupérez le programme readfile.go sur MADOC et testez-le. Expliquez l'erreur que vous observez. Créez au bon endroit (pour qu'il n'y ait pas d'erreur) un fichier *test* contenant quelques lignes de texte puis testez à nouveau votre programme.

5. https://pkg.go.dev/io/ioutil

Une fois le fichier lu et stocké sous forme de chaîne de caractères, on peut le traiter en utilisant des fonctions d'analyse des chaînes de caractères, comme celles du paquet strings [6] par exemple.

Pour des raisons de performances, il n'est pas toujours raisonnable de lire un fichier entier (et donc de le stocker en mémoire). Dans la suite de cette partie nous allons voir qu'il est possible de lire un fichier morceau par morceau.

2.2 Extraire des données d'un fichier

En utilisant le paquet os on a accès aux fonctions de bas niveau permettant de lire un fichier octet par octet (notamment la fonction Read du paquet os). Ces fonctions peuvent être un peu compliquées à manipuler (pour des raisons techniques, c'est parfois difficile de reconstituer une ligne de texte à partir de celles-ci) cependant on a aussi accès à des fonctions de plus haut niveau, construite à partir des fonctions du paquet os, permettant de lire plus simplement des fichiers.

Le paquet fmt [7] en particulier fournit quelques fonctions intéressantes. Nous allons étudier en particulier la fonction Fscanln qui permet de lire une ligne d'un fichier et de stocker les différentes valeurs stockées sur cette lignes (séparées par des espaces) dans des variables. Le programme 4 montre un exemple d'utilisation de cette fonction.

```go
package main

import (
    "fmt"
    "log"
    "os"
)

func main() {
    // Ouverture du fichier
    var filePath string = "fichierScan"
    var myFile *os.File
    var err error
    myFile, err = os.Open(filePath)
    if err != nil {
        log.Fatal(err)
    }

    // Lecture d'une ligne
    var nbLus int
    var unEntier int
    var uneChaine string
    nbLus, err = fmt.Fscanln(myFile, &unEntier, &uneChaine)
    if err != nil {
        log.Fatal(err)
    }
    log.Print("J'ai lu ", nbLus, " valeurs.")
    log.Print("unEntier = ", unEntier)
    log.Print("uneChaine = ", uneChaine)

    // Fermeture du fichier
    err = myFile.Close()
    if err != nil {
        log.Fatal(err)
    }
}
```

Programme 4 – fscanln.go

La première partie de ce programme consiste à ouvrir un fichier. Pour cela on utilise la fonction Open

6. https://pkg.go.dev/strings
7. https://pkg.go.dev/fmt

du paquet os en indiquant le chemin du fichier à ouvrir. Cette fonction retourne un fichier et une erreur. Comme pour ReadFile dans la partie précédente, il faut vérifier que l'erreur vaut nil avant d'utiliser le fichier.

La deuxième partie du programme est la lecture à proprement parler. On commence par préparer une variable nbLus qui servira à indiquer combien de valeurs ont été lues avec succès ainsi que deux variables unEntier et uneChaine pour stocker les données qu'on souhaite extraire du fichier. On utilise ensuite la fonction Fscanln en lui donnant en arguments le fichier à lire (ici myFile) et les adresses des variables où stocker les données. Cette fonction peut retourner une erreur si les choses se sont mal passées, on doit vérifier que cette erreur est différente de nil avant d'utiliser les données récupérées.

La troisième partie du programme ferme le fichier.

Pourquoi utilise-t-on les adresses des variables unEntier et uneChaine plutôt que directement les variables elles-mêmes ?

Récupérez le programme fscanln.go sur MADOC et exécutez-le. Expliquez l'erreur que vous rencontrez. Créez ensuite un fichier fichierScan et testez à nouveau. Expliquez l'erreur que vous rencontrez. Ajoutez une ligne quelconque au fichier fichierScan et testez à nouveau. Expliquez l'erreur que vous rencontrez. Changez fichierScan pour qu'il ne contienne que des lignes avec un entier suivi d'un espace suivi d'une chaîne de caractères sans espaces. Testez à nouveau.

Après qu'une ligne ait été lue, la position actuelle dans le fichier est mise-à-jour dans la structure de données myFile. Le prochain appel à Fscanln lira donc la ligne suivante.

Modifiez le programme fscanln.go pour qu'il lise 5 lignes du fichier fichierScan à l'aide d'une boucle. Testez-le. Corrigez si nécessaire le contenu du fichier fichierScan pour que le programme ne produise plus d'erreur à la lecture de ce fichier. Testez à nouveau.

Lorsque la fonction Fscanln atteint la fin du fichier lu, elle retourne une erreur particulière (que vous avez peut être observée dans vos tests). Cette erreur s'appelle EOF et elle fait partie du paquet io. On peut donc y faire référence dans du code Go en écrivant io.EOF.

Modifiez le programme fscanln.go pour qu'il lise le fichier fichierScan jusqu'au bout, quelque soit sa taille. Vous pourrez utilisez pour cela une conditionnelle dans votre boucle, qui teste si l'erreur est égale à io.EOF et stoppe la boucle si c'est le cas.

De la même façon que Fprintln correspond à Println, Fscanln correspond à une fonction Scanln qui permet de lire ce que l'utilisateur entre au clavier dans le terminal.

2.3 Lire un fichier ligne par ligne

La fonction Fscanln n'est pas toujours la plus pratique à utiliser, en particulier car elle impose que les données soient séparées par des espaces. Nous allons maintenant voir un autre moyen de lire les données d'un fichier, en utilisant des fonctions du paquet bufio [8].

Les fonctions du paquet bufio sont des entrées/sorties *bufferisées*, c'est-à-dire que lorsqu'on demande une lecture les données sont lues dans un espace mémoire (un tampon, *buffer* en anglais) où elles ont été préalablement préchargées depuis le fichier. Quand l'espace est vide, des données sont rechargées dedans, jusqu'à le remplir. Ceci limite les accès aux fichiers, qui sont des opérations coûteuses pour un ordinateur.

8. `https://pkg.go.dev/bufio`

Le programme 5 montre comment préparer un fichier à être lu, puis comment lire quelques lignes de celui-ci.

```go
package main

import (
    "bufio"
    "log"
    "os"
)

func main() {
    // Ouverture du fichier
    var filePath string = "test"
    var myFile *os.File
    var err error
    myFile, err = os.Open(filePath)
    if err != nil {
        log.Fatal(err)
    }

    // Préparation de la lecture
    var scanner *bufio.Scanner
    scanner = bufio.NewScanner(myFile)

    // Lecture des lignes du fichier
    for scanner.Scan() {
        log.Print("Je viens de lire: ", scanner.Text())
    }

    // Vérification que tout s'est bien passé
    if scanner.Err() != nil {
        log.Fatal(scanner.Err())
    }

    // Fermeture du fichier
    err = myFile.Close()
    if err != nil {
        log.Fatal(err)
    }
}
```

Programme 5 – bufio.go

La première partie de ce programme consiste à ouvrir un fichier, exactement comme nous l'avons vu avant.

La deuxième partie du programme utilise la fonction NewScanner du paquet bufio pour construire un *scanner* qui sera utilisé pour lire le fichier.

La troisième partie utilise la méthode Scan pour lire, en boucle, le fichier ligne par ligne. À chaque tour de boucle on récupère la ligne de texte lue grâce à la méthode Text. La boucle s'arrête quand toutes les lignes ont été lues ou quand il y a un problème à la lecture : dans ces deux cas la méthode Scan retourne false.

Enfin, en utilisant la méthode Err, on vérifie que la lecture s'est bien passée et qu'on a bien atteint la fin du fichier.

On peut noter que tout ceci fonctionne bien car scanner est un pointeur vers un bufio.Scanner, les méthodes et fonctions peuvent donc modifier son contenu (l'endroit où on est dans le fichier, le contenu de la dernière ligne lue, l'erreur).

Récupérez le programme bufio.go sur MADOC et testez-le.

Modifiez le programme bufio.go pour compter le nombre de lignes du fichier et l'afficher. Testez votre code sur différents fichiers.

Modifiez le programme bufio.go pour afficher uniquement les 5 premiers lignes du fichier. Testez votre code sur différents fichiers. En particulier, considérez le cas de fichiers qui contiennent moins de 5 lignes.

2.4 Lire un fichier csv

En guise d'entraînement à l'utilisation des fonctions de lecture de fichier nous allons prendre un exemple. Vous trouverez sur MADOC un fichier notes.csv. Celui-ci contient sur chaque ligne des données (toutes les notes reçues par des étudiants au cours d'un semestre), séparées par des virgules : un prénom, une note, un bonus, une note totale (qui est la somme de la note et du bonus). Une ligne représente donc une note reçue par un étudiant, il peut y avoir plusieurs lignes pour le même étudiant (s'il a reçu plusieurs notes).

Écrire un programme qui vérifie que chaque note totale est bien la somme de la note et du bonus dans le fichier notes.csv en affichant le numéro de chaque ligne où ce n'est pas le cas. Vous pourrez utiliser la fonction Split du paquet strings pour découper vos lignes à partir des virgules. Corriger (à la main) dans le fichier notes.csv les éventuelles erreurs que vous trouvez en modifiant la note totale (pas la note de base ni le bonus).

Écrire un programme qui donne la liste de tous les étudiants ayant reçu au moins une note. Cette liste sera stockée dans un tableau.

Écrire un programme qui calcule la moyenne (à partir des notes totales, intégrant les bonus) d'un étudiant dont le prénom est donné en argument du programme.

Il existe aussi des bibliothèques permettant de lire et écrire simplement des fichiers structurés (json [9], csv [10], xml [11] par exemple). Ces bibliothèques vont directement retranscrire les contenus des fichiers sous forme de structures de données et écrire les contenus de structures de données sous forme de fichiers.

9. `https://pkg.go.dev/encoding/json`
10. `https://pkg.go.dev/encoding/csv`
11. `https://pkg.go.dev/encoding/xml`