Les bases du langage Go
Partie 7 : goroutines et canaux

Programmation système

BUT informatique, deuxième année

Comme pour le cours d'initiation au développement, le langage Go sera utilisé pour le cours de programmation système. Ce TP a pour but de compléter les bases du langage Go étudiées l'an dernier. On part du principe que les bases du langage Go (5 parties) vues en initiation au développement sont maîtrisées.

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.

Si vous souhaitez avoir un autre point de vue sur le contenu de ce TP, vous pouvez consulter, par exemple, le site *Go by Example*[1].

1. `https://gobyexample.com/`

1 Goroutines

En Go, une goroutine est un moyen d'exécuter une fonction de manière concurrente par rapport au reste du programme. On dispose d'un seul moyen de contrôle sur les goroutines : le mot clé go qui permet de les créer.

Le programme 1 montre la création d'une goroutine à partir de l'appel de fonction iter(1).

```go
package main

import "fmt"

func iter(ID int) {
    for i := 0; i < 100; i++ {
        fmt.Println("Goroutine", ID, "itération", i)
    }
}

func main() {
    go iter(1)
    iter(2)
}
```

Programme 1 – goroutines.go

Récupérez le fichier goroutines.go sur MADOC et testez le programme plusieurs fois. Obtenez-vous toujours le même résultat ? Pourquoi ? Est-ce que chacune des deux goroutines affiche bien 100 messages à chaque fois ? Pourquoi ? Que se passe-t-il si on inverse les deux lignes du main ? Expliquez.

Modifiez le fichier goroutines.go pour lancer 100 goroutines (numérotées de 1 à 100) au lieu de 2.

2 Canaux

En Go, la mémoire est partagée entre goroutines, on peut donc, comme dans le programme 2 avoir plusieurs goroutines qui utilisent une même variable pour communiquer.

Récupérez le fichier partage.go sur MADOC et testez-le.

Utiliser la mémoire partagée pour communiquer entre goroutines n'est pas recommandé : on peut avoir des conflits entre les accès aux variables partagées par les différentes goroutines. Le mécanisme recommandé en Go pour la communication entre goroutines s'appelle les canaux.

Le programme 3 montre comment créer un canal contenant des entiers et de taille 3 (c'est-à-dire pouvant contenir 3 valeurs en même temps). On écrit ensuite des valeurs dans le canal, puis on en lit (et on les affiche).

Récupérez le fichier chan.go sur MADOC et testez-le. Dans quel ordre sont rangés les messages dans un canal ?

```go
package main

import "fmt"

var turn1 bool

func routine1() {
    for {
        if turn1 {
            fmt.Println("Routine 1")
            turn1 = false
        }
    }
}

func routine2() {
    for {
        if !turn1 {
            fmt.Println("Routine 2")
            turn1 = true
        }
    }
}

func main() {
    go routine1()
    routine2()
}
```

Programme 2 – partage.go

```go
package main

import "fmt"

func main() {

    var c chan int = make(chan int, 3)

    c <- 1
    c <- 2
    c <- 3

    fmt.Println(<-c)
    fmt.Println(<-c)
    fmt.Println(<-c)

}
```

Programme 3 – chan.go

On parle de deadlock lorsque chaque goroutine d'un programme est en attente sur une opération nécessitant l'intervention d'une autre goroutine pour être débloquée.

Modifiez le programme chan.go pour ajouter une écriture supplémentaire dans c avant les lectures. Testez votre programme. Qu'en déduisez-vous ?

Modifiez le programme chan.go pour ajouter deux lectures supplémentaires dans c après les écritures. Testez votre programme. Qu'en déduisez-vous ?

On peut bien sûr créer des canaux contenant d'autres types de valeurs que des entiers : booléens, chaînes de caractères, tableaux, ou même canaux.

Réécrivez le programme 2 en remplaçant l'utilisation de variables partagées par l'utilisation de canaux.

On peut aussi créer des canaux de taille 0 en ne précisant pas de taille dans make, comme dans le programme 4. Une écriture dans un tel canal n'est alors possible que si une lecture à lieu au même moment (et réciproquement).

```go
package main

import (
    "fmt"
    "time"
)

func writer(c chan bool) {
    for {
        time.Sleep(500 * time.Millisecond)
        c <- true
        fmt.Println("Écriture")
    }
}

func reader(c chan bool) {
    for {
        time.Sleep(time.Second)
        <-c
        fmt.Println("Lecture")
    }
}

func main() {
    var c chan bool = make(chan bool)
    go writer(c)
    reader(c)
}
```

Programme 4 – zero.go

Récupérez le ficher zero.go sur MADOC et testez-le. Modifiez-le pour donner une grande taille au canal c lors de sa création. Testez à nouveau. Quelle différence observez-vous ? Expliquez.

3 La structure de contrôle select

Les canaux viennent avec une structure de contrôle permettant de faire une attente conditionnelle : select. Un exemple d'utilisation de cette structure de contrôle est donné dans le programme 5.

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func routine(c chan int) {
    wait := rand.Intn(1000)
    time.Sleep(time.Duration(wait) * time.Millisecond)
    c <- wait
}

func main() {

    rand.Seed(int64(time.Now().Nanosecond()))

    c1 := make(chan int)
    c2 := make(chan int)

    go routine(c1)
    go routine(c2)

    select {
    case w := <-c1:
        fmt.Println("La première goroutine a été la plus rapide.")
        fmt.Println("Elle a mis", w, "millisecondes.")
    case w := <-c2:
        fmt.Println("La deuxième goroutine a été la plus rapide.")
        fmt.Println("Elle a mis", w, "millisecondes.")
    }

}
```

Programme 5 – select.go

Expliquez en détails le fonctionnement du programme 5.

Récupérez le fichier select.go sur MADOC et testez-le plusieurs fois.

Renseignez-vous sur la fonction After du paquet time. Modifiez select.go en ajoutant une condition dans le select pour limiter l'attente à 500ms.