IA04 : Systèmes Multi-Agents

Chapitre 1 : présentation du langage Go

Sylvain Lagrue

I. Introduction

Concepteurs

- *Robert Griesemer* (Google's V8 JavaScript engine, Java HotSpot virtual machine, etc.), *Rob Pike* (Bell's Plan 9 OS, premier environnement graphique Unix, UTF-8, etc.) et *Ken Thompson* (UNIX, le langage B, UTF-8, prix Turing, etc.)

Langages de référence

- C, Python, Pascal, Erlang

I. Introduction

Objectifs du langage

- Faire des applications « systèmes » et des (micro-)services (web)
- Construire des systèmes robustes et sûrs
- Être extrêmement lisible (minimiser les mots-clés, une seule façon de faire chaque chose, etc.)
- Vitesse de compilation et faciliter la gestion des dépendances
- Faciliter la programmation sur plusieurs processeurs/cœurs et la programmation concurrentielle
- Faciliter la programmation réseau
- Fournir de base tous les outils (reformatage de code `gofmt`, documentation automatique `godoc`, tests, etc.)
- Passage à l'échelle aisé (`scalability`)

I. Introduction

Historique

- Démarré en 2007 comme un "Google 20% project"
- Annoncé en novembre 2009 par Google et libéré (open-source)
- Go **1.0** sorti en février 2012
- Go **1.5** sorti en août 2015 (compilateur Go écrit en Go => *bootstrapping*)
- ...
- Go **1.18** sorti en mars 2022 (introduction des *generics*)
- ...
- Go **1.22** sorti en février 2024 (modif. comportement boucle + routage)
- Go **1.23** sorti en août 2024 (modif. range, Timer, ...)

plus de détails : https://go.dev/doc/devel/release

I. Introduction

Exemples de projets écrits en Go

- Docker, Kubernetes, Gogs
- ...

Quelques (portions de) sites Web écrits en Go

- Google, Dropbox, Netflix, Twitch.tv (chat), Uber, etc.

Awesome Go

- https://github.com/avelino/awesome-go

I. Introduction

Principes du langage/choix de conception

- Langage compilé
- Fortement typé
- *Linkage* statique
- Multi-paradigme (impératif, fonctionnel, objet, concurrent, etc.)
- Gestion de la mémoire par ramasse-miettes
- Format de code **standard**
- Langage avec toutes les facilités/fonctionnalités modernes (Unicode, *closures*, etc.)
- Promesse **respectée** de stabilité de la syntaxe et des API

I. Introduction

On y trouve

- Pointeurs
- Packages et modules
- *Duck typing*, interfaces, composition, méthodes
- Cross-compilation
- Interaction avec des bibliothèques natives (via `CGO`)

On n'y trouve pas...

- D'arithmétiques de pointeur
- D'exceptions, des classes, ~~de l'héritage~~
- ~~De la programmation générique~~
- De bibliothèque standard pour la création de clients lourds (GUI)

I. Introduction

Les mots-clés de go (25)

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

java... (50)

```
abstract     continue     for          new          switch
assert       default      goto         package      synchronized
boolean      do           if           private      this
break        double       implements   protected    throw
byte         else         import       public       throws
case         enum         instanceof   return       transient
catch        extends      int          short        try
char         final        interface    static       void
class        finally      long         strictfp     volatile
const        float        native       super        while
```

Mais aussi...

- C: 32, C++: 82, JavaScript: 63, PHP: 49, Python: 33, Rust: 52

I. Introduction

Liens utiles

- Le site officiel : https://go.dev/
- Un *playground* : https://go.dev/play/
- La documentation officielle : https://go.dev/doc/
- La documentation des outils : https://go.dev/doc/cmd, en particulier https://go.dev/doc/cmd/go
- La documentation des packages : https://pkg.go.dev/
- Un petit tutoriel : https://go.dev/tour
- Les idiomes du langage : https://go.dev/doc/effective_go
- Des proverbes : https://go-proverbs.github.io/

> *Clear is better than clever.*
>
> *Gofmt's style is no one's favorite, yet gofmt is everyone's favorite.*
>
> *Don't panic.*

II. Concepts de base

Un premier programme

```go
package main

import "fmt"

// Ceci est un commentaire
func main() {
    fmt.Println("Hello World")
}
```

Remarques

- `package` : le package courant (ici `main`)
- `import` : pour les autres packages
- commentaires : `//` ou `/* */`

Les types de base

Les nombres

Les entiers

- `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`
- `uint`, `int`, `uintptr` (dépendants de l'architecture)
- `byte`

Les flottants

- `float32`, `float64`

Les complexes

- `complex`, `complex64`, `complex128`

Opérateurs

- `+` `-` `*` `/` `%`

Les chaînes de caractères

- `string`, `rune`
- Unicode
- non-mutables
- indexées à partir de 0

```
"Totor\n le\n Castor"

`Totor
le
Castor
`
```

Opérateurs

- `[]` `len` `+`

Quelques packages

- `strings`, `fmt`

II. Concepts de base

Booléens

- `boolean`
- 2 valeurs : `true` et `false`
- pas équivalents à des entiers (pas de *truthy/falsy values*)

Opérateurs

- `&&` `||` `!`

Opérateurs de comparaison (nombres et chaînes)

- `==` `!=` `<` `>` `<=` `>=`

Exemple

```
(true && false) || (false && true) || !(false && false)
```

Les variables

Déclaration

- `var name type` (à la Pascal)

Exemple

```
package main

import "fmt"

func main() {
    var x string = "Totor"
    x += " le Castor"
    fmt.Println(x)
}
```

Déclaration multiple

```
   var (
      a int = 1
      b int = 2
      c int = 3
    )
```

Les 0-values

- Si elles ne sont pas affectées à leur initialisation, les variables prennent automatiquement une valeur
- `0` pour les nombres, `""` pour les chaînes, `false` pour les booléens, etc.

Exemple

```
package main

import "fmt"

func main() {
    var x int
    fmt.Println(x * 12) // 0 !
}
```

Typage automatique

- L'opérateur d'affectation/définition `:=`

Exemple

```
package main

import "fmt"

func main() {
    x := "Totor"
    x += " le Castor"
    fmt.Println(x)
}
```

Constantes

Déclaration

- `const constante = value`

Exemple

```
package main

import "fmt"

func main() {
    const π float64 = 3.141592653589793

    fmt.Print("rayon : ")
    var r float64
    fmt.Scanf("%f", &r)

    out := 2 * π * r
    fmt.Println(out)
}
```

Structure de contrôle

Itération : `for`

Forme 1 (C-like)

```
package main

import "fmt"

func main() {
    for i := 0; i <= 10; i++ {
        fmt.Println(i)
    }
}
```

Forme 2 (while-like)

```
package main

import "fmt"

func main() {
    i := 0;
    for i <= 10 {
        fmt.Println(i)
        i++
    }
}
```

Forme 3 (boucle infinie)

```
package main

import "fmt"

func main() {
    i := 0
    for {
        fmt.Println(i)
        i++
    }
}
```

Condition : `if` / `else`

```
package main

import "fmt"

func main() {
    var n uint

    fmt.Print("entrez un nombre positif : ")
    fmt.Scanf("%d", &n)

    if n%2 == 0 {
        fmt.Println(n, "est multiple de 2")
    } else if n%3 == 0 {
        fmt.Println(n, "est multiple de 3 mais pas de 2")
    } else {
        fmt.Println(n, "n'est multiple ni de 2 ni de 3")
    }
}
```

`switch` / `case` / `fallthrough`

Exemple

```
package main

import "fmt"

func main() {
    fmt.Println("compte avec Dora !")
    for i := 0; i < 10; i++ {
        switch i {
        case 0:
            fmt.Println("Zero")
        case 1:
            fmt.Println("One")
        case 2:
            fmt.Println("Two")
        case 3:
            fmt.Println("Three")
        case 4:
            fmt.Println("Four")
        case 5:
            fmt.Println("Five")
        default:
            fmt.Println("Unknown Number")
        }
    }
}
```

Remarques

- ne fonctionne pas que pour les entiers
- pas besoin de `break`
- pour continuation : `fallthrough`
- on peut mettre plusieurs valeurs sur un `case` (séparées par `,`)

Les fonctions

```go
package main

import "fmt"

func say(s string) {
    fmt.Println(s)
}

func meuh() string {
    return "meuh"
}

func plus(a int, b int) (c int) {
    c = a + b
    return c // facultatif, seul le return est nécessaire
}

func main() {
    say(meuh())
    fmt.Println(plus(6, 6))
}
```

Retourner plusieurs valeurs

```go
package main

import "fmt"

func f() (int, int) {
	return 1, 2
}

func main() {
	x, y := f()
	z, _ := f()
	_, t := f()
	fmt.Println(x, y, z, t) // 1 2 1 2
}
```

Nombre d'arguments inconnus

```
package main

import (
    "fmt"
)

func add(args ...int) int {
     total := 0
     for _, v := range args {
          total += v
     }
     return total
}

func main() {
     fmt.Println(add(1,2,3))

     tab := []int{1,2,3}
     fmt.Println(add(tab...))
}
```

```
6
6
```

Remarques

- On peut créer des fonctions/prendre des fonctions en paramètre
- Les *closures* fonctionnent également
- Récursivité OK
- Les fonctions peuvent être stockées dans des variables
- Tous les passages de paramètres sont par copie
- On peut créer des fonctions anonymes et les exécuter immédiatement (*Immediately-invoked Function Expression* a.k.a. IME)

Exemple

```
package main

import "fmt"

func plusN(n int) func(int) int {
    f := func(x int) int {
        return x + n
    }

    return f
}

func main() {
    plus1 := plusN(1)
    fmt.Println(plus1(12))
}
```

III. Autres types

Arrays

- ils sont finis et **non-redimensionnables**
- ils sont commencent à l'indice **0**
- ils **sont passés par copie**

Exemple

```
package main

import "fmt"

func main() {
    var tab [5]int

    for i:=0; i < 5; i++ {
        tab[i] = i + 1
    }

    fmt.Println(tab)
    fmt.Println(len(tab))
}
```

Autre exemple (initialisation et itération)

```go
package main

import "fmt"

func main() {
    tab := [...]int{1, 2, 3, 4, 5} // équiv. à [5]int{1, 2, 3, 4, 5}

    for index, value := range tab {
        fmt.Printf("(%d, %d)", index, value)
    }
}
```

Affichage

```
(0, 1)(1, 2)(2, 3)(3, 4)(4, 5)
```

III. Autres types

Slices

- C'est un "pointeur sur tableau"
- Il pointe sur le début du tableau
- Il a une certaine taille et une capacité
- Très compact en mémoire

Définition

```
var x []int
```

III. Autres types

Exemple

```
package main

import "fmt"

func main() {
    var tab [5]int

    sl := tab[1:4]

    for i := range sl {
        fmt.Printf("%d ", i)
        sl[i] = 12
    }
    fmt.Print("\n\n")
    fmt.Printf("  %v   %d\n", sl, len(sl))
    fmt.Println(tab, len(tab))
}
```

Affichage

```
0 1 2

  [12 12 12]   3
[0 12 12 12 0] 5
```

III. Autres types

Création/modification/copie dynamique

- Créer un tableau : `make` (≈ `calloc`)
- Ajouter un élément à un tableau : `append` (≈ `realloc`)
- Copier un tableau dans un tableau : `copy(dst, src)` (≈ `memcpy`)

Exemple

```go
package main

import "fmt"

func main() {
    sl := make([]int, 5)
    for i := range sl {
        sl[i] = i + 1
    }

    fmt.Println(sl, len(sl))

    sl = append(sl, 6, 7, 8)
    fmt.Println(sl, len(sl))

    sl2 := make([]int, 4)
    copy(sl2, sl)

    fmt.Println(sl2, len(sl2))
}
```

Affichage

```
[1 2 3 4 5] 5
[1 2 3 4 5 6 7 8] 8
[1 2 3 4] 4
```

III. Autres types

Maps

- Ce sont des tableaux associatifs/dictionnaires/maps/etc.
- Elles sont *non-ordonnées*
- Ils doivent être initialisés avec `make`
- `delete` permet de supprimer un élément
- Elles sont passées **par copie de référence** dans les fonctions

Exemple

```go
package main

import "fmt"

func rm(m map[string]int) {
    delete(m, "two")
}

func main() {
    m := make(map[string]int)
    m["one"] = 1
    m["two"] = 2
    m["three"] = 3
    rm(m)

    fmt.Println("compte avec Dora !")
    for key, val := range m {
        fmt.Println(key, val)
    }
    fmt.Println("raté...")
}
```

Affichage

```
compte avec Dora !
three 3
one 1
raté...
```

Pointeurs

- `*` pour déclarer et déréférencer
- `&` pour récupérer l'adresse
- Pas d'arithmétique de pointeur
- `new` permet de créer une variable et renvoie le pointeur
- Pointeur nul : `nil`

Exemple

```go
package main

import "fmt"

func main() {
    var pi *int
    i := 12

    pi = &i
    *pi++

    fmt.Println(pi, *pi, i)

    pi2 := new(int)
    i = *pi2

    fmt.Println(pi2, *pi2, i)

}
```

Affichage

```
0x1040e0f8 13 13
0x1040e134 0 0
```

Création de type et types composés

Création d'un type

- `type` : permets de créer un nouveau type

```
type taille int
```

Types composés

- `struct` : permet de créer un type composé

```
type Rectangle struct {
     x float64
     y float64
     h float64
     w float64
 }
```

- Si rien n'est donné, chacun de ses champs prend sa 0-value

```
var r1 Rectangle
r2 := Rectangle{0, 0, 4, 5}
r3 := Rectangle{h: 4.5, w: 5.5}
fmt.Println(r1, r2, r3)
```

```
{0 0 0 0} {0 0 4 5} {0 0 4.5 5.5}
```

- Accès à une propriété

```
r3.h = r3.h - r2.h
fmt.Println(r3)
```

```
{0 0 0.5 5.5}
```

Méthodes

On peut associer des méthodes à une structure

```
func (r *Rectangle) Area() float64 {
    return r.h * r.w
}

func (r *Rectangle) Translate(x float64, y float64) {
    r.x = r.x + x
    r.y = r.y + y
}
```

```
r3.Translate(10, 20)
fmt.Println(r3)
```

```
{10 20 0.5 5.5}
```

Composition de structures/mixin

- On peut composer différents types

```
type Colored struct {
    color string
}

func (c *Colored) Color() {
    return c.color
}

type ColoredRectangle struct {
    Colored
    Rectangle
}
```

- Initialisation et utilisation de méthodes

```
cr := ColoredRectangle{Colored{"red"}, r2}

fmt.Println(cr)
fmt.Println(cr.Color())
```

```
{{red} {0 0 4 5}}
red
```

Interfaces

- On peut également définir des interfaces

```
type Translatable interface {
    Translate(x float64, y float64)
}
```

- Pas besoin d'implémenter une interface : *duck typing*

```
func moveUp(t Translatable) {
    t.translate(100, 0)
}
```

```
moveUp(&r1)
fmt.Println(r1);
```

```
moveUp(r1)
println(r1)
```

```
{100 0 0 0}
```

- L'interface vide `{}interface` permet d'accepter n'importe quel type

Les `generics`

- Go ≥ 1.18
- Permettent d'écrire des fonctions et des types génériques
- En Go, ils sont vus comme des contraintes sur les types

Paramètre de type

```
func Index[T comparable](s []T, x T) int
```

Paramétrer une structure

```
type List[T] struct {
    next *List[T]
    val  T
}
```

Types somme

```
type Number interface {
   int64 | float64 | ~uint64 // tous les "descendants" d'uint64
}
```

Contraintes de types prédéfinies

- `comparable` (peuvent être en entrée de `map`)
- `any`

Goroutines

- Ce sont des "threads légers" (green threads)
- Permettent de lancer des actions en parallèle
- Se lancent très vite
- On peut en avoir de très nombreuses en // (plusieurs milliers sans pb)
- Ordonnanceur s'occupe de déplacer la go-routine dans un autre *thread* si besoin
- L'ordre d'exécution est inconnu
- Mot clé : `go`
- Quand la goroutine `main` se finit, tout le programme se finit immédiatement !

Exemple

```go
package main

import "fmt"

func Print(i int) {
    fmt.Printf("%d ", i)
}

func main() {
    for i := 0; i < 10; i++ {
        go Print(i)
    }

    fmt.Scanln() // Pour attendre la fin des goroutines...
}
```

Affichage

```
7 0 5 3 6 2 8 4 1 9
```

Channels

- Permettent la communication inter-goroutine
- Proches des "pipes" Unix dans leur fonctionnement
- Bloquants en réception et en émission
- Utilisés pour synchroniser les processus suivant le modèle CSP (Communicating Sequential Processes, [C. A. R. Hoare 1978])
- Permettent d'implémenter une architecture interne en micro-services

Opérations sur les channels

Création d'un channel

```
c := make(chan string)
```

N.B. : `make` peut prendre un deuxième argument (entier `n`) qui permet de lire et de mettre en cache `n` éléments avant d'être bloquant

Envoyer dans un channel

```
s := "coucou"

c <- s
```

Recevoir dans un channel

```
s = <- c // On stocke la valeur
<- c     // On ne la stocke pas
```

Exemple

goroutine "serveur"

```
package main

import "fmt"

func SayMeGoodbye(c chan string) {
    var s string

    for {
        s = <-c
        fmt.Printf("You said me: %q\n", s)

        if s == "Goodbye..." {
            c <- "bye"
            return
        } else {
            c <- "I don't understand"
        }
    }
}
```

main

```
func main() {
    c := make(chan string)
    go SayMeGoodbye(c)

    msgs := []string{"Hello", "How are you?", "What?", "Goodbye..."}
    var resp string

    for _, s := range msgs {
        c <- s
        resp = <-c
        fmt.Printf("Your answer: %q\n", resp)
    }

    fmt.Scanln()
}
```

Affichage

```
You said me: "Hello"
Your answer: "I don't understand"
You said me: "How are you?"
Your answer: "I don't understand"
You said me: "What?"
Your answer: "I don't understand"
You said me: "Goodbye..."
Your answer: "bye"
```

Autres opérations sur les channels

- `close` : permet de fermer un channel
- `range` : permet d'itérer tant que le channel n'est pas fermé

Exemple

```
package main

import "fmt"

func main() {
    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)

    for elem := range queue {
        fmt.Println(elem)
    }
}
```

Autres opérations sur les channels (suite)

- `select` : permet d'écouter plusieurs channels en même temps

Exemple

```
select {
case msg1 := <- c1:
     fmt.Println("Message 1", msg1)
case msg2 := <- c2:
     fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
     fmt.Println("timeout")
}
```

Packages

Principe

- `import` permet d'importer un package (**non récursivement**)
- `package` de définir un package
- Le nommage suit l'arborescence des fichiers
- Tous les fichiers contenus dans un même répertoire font partie du même package
- Origine : `$GOPATH/pkg` (compilés) et `$GOPATH/src`
- Les exécutables appartiennent au package `main`. Leur nom dépend du répertoire contenant les sources.
- On peut "changer le nom" du package (alias)
- Installation de packages distants : `go get`

Visibilité

- seules les fonctions/variables/types/etc. commençant par une majuscule sont visibles depuis l'extérieur du package

Exemple

```
package main

import (
    "bytes"
    "fmt"
    "log"
    "os/exec"
    "strings"
)

func main() {
    cmd := exec.Command("tr", "a-z", "A-Z")
    cmd.Stdin = strings.NewReader("some input")
    var out bytes.Buffer
    cmd.Stdout = &out
    err := cmd.Run()
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("in all caps: %q\n", out.String())
}
```

Les modules

- Permettent de gérer (différentes versions) de dépendance (proche de `nmp`/`yarn`, `apt`/`dnf`)
- Suivent la numérotation de version (majeur/mineure/version de patch) 1.1.2
- Les versions majeures ne sont pas compatibles
- Initialiser la gestion des modules à la racine de votre arborescence de package : `go mod init`
- Tout est géré automatiquement (avec les utilisations de `go build` et `go test`)

N.-B. : Depuis la version 1.17, la commande `go get` en dehors d'un répertoire initialisé est dépréciée

Référence : https://golang.org/ref/mod

`defer`

- Exécute une fonction à la fin de la fonction courante
- Utilisation pour fermeture de fichiers/sockets/bd/...
- Les fonctions sont exécutées dans l'ordre inverse des `defer`

Exemple

```
package main
import "fmt"

func first() {
     fmt.Println("1st")
}

func second() {
     fmt.Println("2nd")
}

func main() {
     defer second()
     first()
}
```

IV. Concepts avancés

Vrai exemple...

```
func stats(filename string) {
    f, err := os.Open(filename)
    if err != nil {
        return
    }

    defer f.Close()

    stat, _ := f.Stat()
    fmt.Println(stat)
}
```

`panic` & `recover`

- `panic` abandonne l'exécution du programme
- `recover` permet de reprendre l'exécution
- proverbe go:

> *"Don't panic!" – Rob Pike (https://go-proverbs.github.io/)*

Exemple

```
package main
import "fmt"
func main() {
	defer func() {
		str := recover()
		fmt.Println(str)
	}()
	panic("PANIC")
}
```

Exercices

1. Créer une fonction `Moyenne` qui prend en entrée un slice d'entiers et qui renvoie la moyenne de ses éléments.
2. Créer une fonction `Plus1` qui prend en entrée un slice d'entiers et qui ajoute un à chaque élément.
3. Créer une fonction `compte(n int, tab []int)` qui affiche les nombres contenus dans `tab`. Lancer cette fonction à partir d'une goroutine (une par élément du tableau). Dans quel ordre sont affichés les entiers ?
4. Lancer 2 goroutines qui se répondent ping/pong via un channel.

Le package `time`

Le type `Duration`

- type de base pour représenter les durées
- c'est un `uint64`, permet de représenter 2^{64} nanoseconds (≈290 ans…)

Constantes

```
const (
    Nanosecond Duration = 1
    Microsecond         = 1000 * Nanosecond
    Millisecond         = 1000 * Microsecond
    Second              = 1000 * Millisecond
    Minute              = 60 * Second
    Hour                = 60 * Minute
)
```

Remarque : Pas au-delà de l'heure pour éviter les problèmes de définition du jour et de l'année (solaire/terrestre)

Idiome Go pour déterminer la durée d'une fonction en Go

```
d := time.Now()
f()
fmt.Println("temps passé :", time.Since(d))
```

Troncature

```
d.Trunc(time.Millisecond)
```

Passer d'une chaîne de caractère à une durée ("parser" une durée)

```
d, err := time.ParseDuration("12µs")
// unités : "ns", "us"/"µs", "ms", "s", "m", "h"
```

Sleep

`func Sleep(d Duration)` met en pause la goroutine pendant la durée `d`

Exemple

```
package main

import (
    "fmt"
    "time"
)

func main() {
    go func() {
        fmt.Println("je fais la sieste pendant 5s...")
        fmt.Println("💤")
        time.Sleep(5 * time.Second)
        fmt.Println("je suis réveillée et en pleine forme !")
    }()
    fmt.Scanln()
}
```

`After`

- `func After(d Duration) <-chan Time` attend une durée `d` et renvoie le temps courant sur le channel renvoyé
- Intimement lié au type `time.Timer`

Exemple : timeout

```
select {
    case m := <-c:
        handle(m)
    case <-time.After(5 * time.Minute):
        fmt.Println("timed out")
}
```

`Tick`

- `func Tick(d Duration) <-chan Time` renvoie le temps courant toutes les `d` secondes sur le channel
- intimement lié au type `time.Ticker`

Exemple

```
c := time.Tick(1 * time.Second)
tic := true

for now := range c {
    if tic {
        fmt.Println("tic:", now)
    } else {
        fmt.Println("tac:", now)
    }
    tic = !tic
}
```

Attention aux fuites de mémoire (→ utilisation de `time.Ticker`)

Synchronisation

Data Race

```go
package main

import (
    "fmt"
)

var n int

func plus1() {
    n++
}
func main() {
    for i := 0; i < 1000; i++ {
        go plus1()
    }
    fmt.Scanln()
    fmt.Println(n)
}
```

```
258
```

Remarque : `++` n'est pas atomique

Data Race

- Se produit lorsque plusieurs acteurs tentent d'accéder en même temps à une même variable
- *Synonymes :* situation de concurrence, accès concurrent, concurrence critique, course critique, séquencement critique …

V. Gestion du temps et synchronisation en Go

```go
var a, b int

func f() {
    a = 1
    b = 2
}

func g() {
   print(b)
   print(a)
}
func main() {
   go f()
   g()
}
```

Que peut afficher ce programme ?

```
// Affichages possibles
0 0

0 1

2 1

2 0 (!)
```

Mauvais *pattern*

```
var a string
var done bool

func setup() {
   a = "hello, world"
   done = true
}

func main() {
   go setup()
   for !done {
   }
   print(a)
}
```

Affichage possibles :

```
"hello, world"
```

```
""
```

V. Gestion du temps et synchronisation en Go

⇒ Obligation d'utiliser des synchronisations explicites

- En utilisant des channels
- En utilisant le package `sync`

Package sync

- `WaitGroup`
- `Once`
- `Lock`
- `Mutex/RWMutex`

Des primitives encore plus fines

- `sync/atomic`

Néanmoins...

- On préfère utiliser des *channels* et des micro-services pour la synchronisation

Les `Mutex`

Interface `Locker`

```
type Locker interface {
    Lock()
    Unlock()
}
```

2 implémentations

```
type Mutex // mutex total
type RWMutex // mutex permettant les lectures simultanées
             //en empêchant les écritures

// func (*RWMutex) Lock() / Unlock() // for writing
// func (*RWMutex) Rlock() / Runlock()// for reading
```

V. Gestion du temps et synchronisation en Go

Exemple : synchronisation par mutex

```
package main

import (
    "fmt"
    "sync"
)

var l sync.Mutex // 0 value = Unlock status
var n int

func plus1() {
    l.Lock()
    n++
    l.Unlock()
}
func main() {
    for i := 0; i < 1000; i++ {
        go plus1()
    }
    fmt.Scanln()
    fmt.Println(n)
}
```

Exemple (2) : synchronisation par mutex et composition

```
package main

import(
    "fmt"
    "sync"
)

type SynchronizedInt struct {
    sync.Mutex
    val int
}

var n SynchronizedInt

func plus1() {
    n.Lock()
    n.val++
    n.Unlock()
}

func main() {
    for i := 0 ; i < 1000; i++ {
        go plus1()
    }
    fmt.Scanln()
    fmt.Println(n.val)
}
```

Remarques

- L'utilisation de *mutex* ralentit très fortement le programme et fait perdre le gain de la concurrence : le programme s'arrête complètement
- On préférera faire en sorte que les goroutines s'occupent de parties de mémoires séparées
- On préférera utiliser les *channels* pour les problèmes de concurrence...

Pattern de partage via channel et micro-service

```go
package main

import (
    "log"
    "time"
)

var n int

func StartIncService(buffer int) chan int {
    c := make(chan int, buffer)

    go func() {
        log.Println("[Microserv] starting, buffer:", buffer)
        for client := range c {
            log.Printf("[Microserv] treats Client %d", client)
            n += 1
            log.Printf("[Microserv] Client %d done", client)
        }
        log.Println("[Microserv] closed")
    }()

    return c
}
```

V. Gestion du temps et synchronisation en Go

```
func main() {
    cinc := StartIncService(0)

    log.Println("valeur initiale:", n)

    for i := 0; i < 5; i++ {
        go func(i int) {
            // time.Sleep(100 * time.Millisecond)
            log.Printf("[Client %d] ask inc()\n", i)
            cinc <- i
            log.Printf("[Client %d] is back!\n", i)
        }(i)
    }

    time.Sleep(time.Second)
    close(cinc)

    log.Println("valeur finale:", n)
}
```

Avec mise en cache

```
func main() {
    cinc := StartIncService(10)

    fmt.Println("valeur initiale:", n)

    for i := 0; i < 10; i++ {
        go func(i int) {
            fmt.Printf("[%d] demande inc(1)\n", i)
            cinc <- 1
            fmt.Printf("[%d] reprend la main\n", i)
        }(i)
    }

    fmt.Scanln()
    close(cinc)
    fmt.Println("valeur finale:", n)
}
```

`sync.Once`

But : faire en sorte qu'une partie de code ne s'exécute qu'une seule fois (lors d'une initialisation par exemple)

```
type Once
func (o *Once) Do(f func())
```

Exemple

```
package main

import(
    "fmt"
    "sync"
)

type SynchronizedInt struct {
    sync.Mutex
    val int
}

var once sync.Once
var n SynchronizedInt

func setup() { // !! pas init !!
    n.val = 12
}

func plus1() {
    once.Do(setup)

    n.Lock()
    n.val++
    n.Unlock()
}
```

V. Gestion du temps et synchronisation en Go

```
func main() {
    for i := 0 ; i < 1000; i++ {
        go plus1()
    }
    fmt.Scanln()
    fmt.Println(n.val)
}
```

```
// 1012...
```

Synchronisation et `Wait Groups`

Les `Wait Groups` permettent de mettre en attente une goroutine le temps que d'autres goroutines aient eu le temps de finir. En fait, il s'agit d'un compteur incrémenté et décrémenté de manière atomique.

```
// type de base
type WaitGroup

// ajoute un nombre de goroutine à attendre
func (wg *WaitGroup) Add(delta int)

// signale la fin de l'action pour une goroutine
func (*WaitGroup) Done()

// attend que tout le monde ait fini
func (wg *WaitGroup) Wait()
```

Exemple

```
// source : https://golang.org/pkg/sync/#WaitGroup

var wg sync.WaitGroup

var urls = []string{
    "http://www.golang.org/",
    "http://www.google.com/",
    "http://www.somestupidname.com/",
}

for _, url := range urls {
    // Increment the WaitGroup counter.
    wg.Add(1)

    // Launch a goroutine to fetch the URL.
    go func(url string) {
        // Decrement the counter when the goroutine completes.
        defer wg.Done()
        // Fetch the URL.
        http.Get(url)
        }(url)
}

// Wait for all HTTP fetches to complete.
wg.Wait()
```

Situations d'interblocage (*Deadlocks*)

Go repère le fait toutes les goroutines sont en attente (état *sleep*) et le *deadlock* provoque un `panic`.

```
package main

import "fmt"

func main() {
    c := make(chan int)
    c <- 12    // write to a channel
    val := <-c // read from a channel
    fmt.Println(val)
}
```

```
> go run dl.go
fatal error: all goroutines are asleep - deadlock!
```

Exemple des lecteurs/rédacteurs

Source : Wikipédia

> *Supposons qu'une base de données ait des lecteurs et des rédacteurs, et qu'il faille programmer les lecteurs et les rédacteurs de cette base de données.*
>
> *Les contraintes sont les suivantes :*
>
> - *plusieurs lecteurs doivent pouvoir lire la base de données en même temps ;*
> - *si un rédacteur est en train de modifier la base de données, aucun autre utilisateur (ni rédacteur, ni même lecteur) ne doit pouvoir y accéder.*

Correction

```
package main

import (
    "fmt"
    "sync"
)

var i int
var d int
var mtx sync.RWMutex

func lecteur() {
    mtx.RLock()
    fmt.Println(i)
    mtx.RUnlock()
}

func redacteur() {
    mtx.Lock()
    i++
    mtx.Unlock()
}
```

V. Gestion du temps et synchronisation en Go

```
func main() {
    go func() {
        for i := 0; i < 1000; i++ {
            go lecteur()
        }
    }()

    go func() {
        for i := 0; i < 1000; i++ {
            go redacteur()
        }
    }()

    fmt.Scanln()
}
```

V. Gestion du temps et synchronisation en Go

À propos...

Information	Valeur
Auteur	Sylvain Lagrue (sylvain.lagrue@utc.fr)
Licence	Creative Common CC BY-SA 3.0
Version document	0.13.0