paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

Version : 28/08/2024

Prérequis

- Notions de programmation. Une expérience de la programmation objet serait appréciée.

Objectifs

- Maîtriser la syntaxe du langage
- Maitriser les concepts et mécanismes spécifiques : le traitement d'erreur, les interfaces, le parallélisme
- Comprendre et utiliser les principales bibliothèques standards : les io, la communication réseau
- Utiliser un environnement de développement

Tour de table

- Présentation des membres
- Vos attentes vis-à-vis de la formation

Programme de la formation

1. L’environnement Go
2. Les constructions de base du langage
3. Les constructions plus avancées du langage
4. La programmation du parallélisme
5. Quelques packages et bibliothèques
6. Compléments
7. Ressources des TP
8. Annexe & Ressources connexes

paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

L”environnement Go

1. Les principales commandes (go, gofmt, godoc).
2. Les tests unitaires, les benchmarks.
3. Quelques IDE.
4. TP : Prise en main de l'environnement de développement.

Présentation de Go

1. Créé par Google en 2007.
2. Conçu pour être simple, rapide, et efficace.
3. Approprié pour les systèmes distribués et la programmation concurrente.

Les principales commandes Go (1/3)

- go
  - go run : Compile et exécute un programme Go.
  - go build : Compile le code Go sans l'exécuter.
  - go mod : Gère les dépendances des modules.
  - go get : Télécharge les packages et modules.

Les principales commandes Go (2/3)

- gofmt
  - Formate automatiquement le code source.
  - Respecte les standards de style Go.
  - gofmt -w fichier

Les principales commandes Go (3/3)

- godoc
  - Génère et visualise la documentation du code.
  - Accessible via une interface web
  - godoc -http=:6060

Tests unitaires et benchmarks (1/2)

- Go inclut des outils pour les tests unitaires.
- Le package testing est utilisé pour définir des tests.
- Commande : go test

```
func TestNomDuTest(t *testing.T) {
  // Assertions et tests
}
```

Tests unitaires et benchmarks (2/2)

- Les benchmarks permettent de mesurer les performances d'une fonction.
- Commande : go test -bench=.

```
func BenchmarkNomDuTest(b *testing.B) {
  for i := 0; i < b.N; i++ {
    // Code à benchmarker
  }
}
```

Quelques IDE

Visual Studio Code (VS Code)

- Avantages :
  - Extensions dédiées à Go (Go extension, Go tools).
  - Debugging intégré.
  - Très populaire dans la communauté.

GoLand

- Avantages :
  - Outil complet pour Go.
  - Refactoring puissant et debugging avancé.
  - Intégration directe avec les outils Go.

Quelques Libraires, Framework, APIs, technologies

- Gin-Gonic : Framework http pour faire des API Rest
- Gorm : ORM pour les basses de données (MySQL, Postgres ...)
- Grpc : Communication intra / inter service (protobuff)
- API Kubernetes : API officielle de kubernetes
- ETCD : Cache clef-valeur avec forte résilience
- gvm : version manager pour Golang
- ... Et plein d'autre

paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

Les constructions de base du langage

1. Les unités de compilation, les packages et les modules : contrôle de la visibilité des classes, mécanisme d'import.
2. Les variables (les différentes formes de déclaration), les types primitifs.
3. Les instructions de contrôle : les instructions conditionnelles, de boucle, de branchement.
4. Les fonctions (les retours multiples, les fonctions variadiques, anonymes).
5. Les pointeurs.
6. TP : Suite d'exercices illustrant les constructions présentées.

Packages et modules (1/2)

- Package : Ensemble de fichiers Go regroupés.
- Chaque fichier appartient à un package (déclaré en haut du fichier).
  - package <nom_package>
- Gestion des packet avec “go mod”
  - Gère les dépendances et versions de packages.

Packages et modules (2/2)

- Import packet standard (lib standard de go)

```
//import unique
import “fmt”

// import groupé
import (
  "fmt"
  "math"
)
```

Packages et modules : public vs privé

- Avec une majuscule : Publique
- Sans majuscule : privé

```
var Toto = “mytotovar” // variable publique (visible après importation)
var toto = “myprivtotovar” // variable privé (seul les membres du packet peuvent voir la variable)

// fonction publique
func Totofunc {
}
// fonction privé
func totofunc {
}
```

Variables en Go

```
var x int = 10 // variable de type int avec valeur 10

var x = 10 // le type de la variable x est récupéré de sa valeur

x := 10 // le type de la variable x est récupéré de sa valeur et est créer depuis sa valeur
```

Variables en Go

- Types de base :
  - Entiers : int, int8, int16, uint, etc.
  - Flottants : float32, float64.
  - Booléens : bool.
  - Chaînes de caractères : string.
- Valeurs par défaut :
  - 0 pour les entiers.
  - 0.0 pour les flottants.
  - false pour les booléens.
  - "" pour les chaînes de caractères.

Les instructions de contrôle (1/2)

```
if x > 10 {
  fmt.Println("x est supérieur à 10")
}

if x > 10 {
  fmt.Println("x > 10")
} else {
  fmt.Println("x <= 10")
}
```

Les instructions de contrôle (1/2)

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

for index, value := range slice {
  fmt.Println(index, value)
}
```

Les fonctions (1/2)

```
func add(a int, b int) int {
  return a + b
}

func divide(a, b int) (int, int) {
  return a / b, a % b
}
```

Les fonctions (2/2)

```
Fonctions variadiques :
func sum(nums ...int) int {
  total := 0
  for _, num := range nums {
    total += num
  }
  return total
}
Fonction anonyme (lambda) :
f := func(x int) int {
  return x * x
}
fmt.Println(f(5)) // 25
```

Les pointeurs

```
rappel : Un pointeur stocke l'adresse mémoire d'une variable.

var p *int
p = &x // & obtient l’adresse de x

fmt.Println(*p) // affiche la valeur pointée par p

func increment(p *int) {
  *p = *p + 1
}
```

paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

LaMeDuSe

Les constructions plus avancées du langage

- Les tableaux, les slides, les structs et les maps.
- L’itérateur range.
- Les méthodes.
- Les interfaces.
- Le traitement des erreurs (panic, recover).
- Les fonctions deferred.
- La construction iota.
- TP: Suite d’exercices mettant en œuvre les constructions présentées.

Les tableaux et les slices (1/2)

Déclaration d’un tableau

```
var arr [5]int

arr := [3]int{1, 2, 3}
```

NOTE : La taille est fixe et inchangeable après la création

Les tableaux et les slices (1/2)

Déclaration d'un slice

```
// déclaration
var slice []int

// initialisation
slice := make([]int, 5)

// ajout d'un élément dans le tableau
slice = append(slice, 10)
```

Les structs

```
type Person struct {
  Name string
  Age  int
  password string
}

p := Person{Name: "Alice", Age: 30, password: "private"}

fmt.Println(p.Name)

fmt.Println(p.password) // ERREUR

p.Age = 31
```

Les maps

```
// définition
var m map[string]int

// initialisation
m = make(map[string]int)

// modification et lecture
m["clé"] = 42
fmt.Println(m["clé"])
```

L’indicateur range

```
for index, value := range arr {
  fmt.Println(index, value)
}

for key, value := range m {
  fmt.Println(key, value)
}
```

Les méthods en go

```
func (p Person) Greet() {
  fmt.Println("Hello, my name is", p.Name)
}

p.Greet()

// Pour modifier une structure on as besoin d’avoir un pointeur sur la structure
func (p *Person) SetName(name string) {
   p.Name = "Eric"
}

p.Greet() // Hello, my name is Eric
```

Les interfaces

```
type Animal interface {
  Speak() string
}

type Dog struct{}

func (d Dog) Speak() string {
  return "Woof"
}

// usage

var a Animal
a = Dog{}
fmt.Println(a.Speak()) // "Woof"
```

Gestion d’erreur

```
result, err := someFunction()
if err != nil {
  fmt.Println("Error:", err)
}

return 0, fmt.Errorf("erreur : %v", err)
```

Gestion d’erreur

```
panic("Quelque chose s'est mal passé!")

func handlePanic() {
  if r := recover(); r != nil {
    fmt.Println("Récupéré d’un panic:", r)
  }
}
```

defer

```
defer fmt.Println("Cette ligne s'exécute en dernier.")
fmt.Println("Cette ligne s'exécute en premier.")
```

Defer permet d’executer une ligne en dernier cela permet par exemple de ne pas oublie de clore la connexion a la fin de l’execution de la fonction

Iota

```
const (
  A = iota
  B
  C
)
fmt.Println(A, B, C) // 0, 1, 2
```

paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

LaMeDuSe

La programmation du parallélisme

1. Les concepts de base et les instructions correspondantes (threads, goroutines, channels et select).
2. Le parallélisme vs la concurrence.
3. La gestion de la concurrence : les verrous, les barrières.
4. TP: Construction d’une application multithreadée.

Paralellisme vs concurrence

Parallélisme vs Concurrence

- Concurrence : Gestion de plusieurs tâches en même temps, mais pas nécessairement en parallèle.
  - Go utilise la concurrence pour diviser le travail en plusieurs sous-tâches exécutées indépendamment.
- Parallélisme : Exécution simultanée de plusieurs tâches sur plusieurs cœurs.
- Go et concurrence : Grâce aux goroutines, Go gère la concurrence de manière efficace et simplifiée.

Goroutine

Goroutine : Légère unité d’exécution concurrente.

- Similaire aux threads, mais plus efficaces en termes de ressources.
- Non bloquantes : Les goroutines ne bloquent pas l’exécution du programme principal.

```
go myFunction() // la fonction est executé de façon parallèle

func printMessage(msg string) {
    fmt.Println(msg)
}

func main() {
    go printMessage("Hello from a goroutine")
    fmt.Println("Main function")
}
```

Les channels

Channel : Mécanisme de communication entre les goroutines.

```
// dans le thead parent des deux goroutine
ch := make(chan int)

// dans une goroutine
ch <- 10

// dans une autre goroutine
val := <-ch // bloque puis retourne 10
```

Les channels

```
func sendValue(ch chan int) {
    ch <- 10
}

func main() {
    ch := make(chan int)
    go sendValue(ch)
    value := <-ch
    fmt.Println(value) // affiche 10
}
```

Select (switch/case)

```
select {
  case msg1 := <-ch1:
      fmt.Println("Message reçu :", msg1)
  case msg2 := <-ch2:
      fmt.Println("Autre message :", msg2)
  default:
      fmt.Println("Aucun message reçu")
}
```

La synchronisation des goroutines

Les goroutines étant non bloquantes, il est nécessaire de les synchroniser pour éviter les conditions de course.

WaitGroup

- Attendre qu'un ensemble de goroutines termine leur exécution.

```
var wg sync.WaitGroup
wg.Add(1)
go func() {
    defer wg.Done()
    // travail de la goroutine
}()
wg.Wait()
```

La synchronisation des goroutines : Mutex

Mutex : Empêche plusieurs goroutines d’accéder simultanément à une ressource partagée.

```
var mu sync.Mutex
mu.Lock()
// section critique
mu.Unlock()
```

La synchronisation des goroutines : Barriers

Barrières : Utilisées pour synchroniser les goroutines à des points spécifiques du programme.

Channel bufferiser : Permet de spécifier un buffer pour stocker temporairement des valeurs dans un channel.

```
ch := make(chan int, 3) // channel bufferisé
ch <- 1
ch <- 2
ch <- 3

// Timeout
select {
  case msg := <-ch:
    fmt.Println("Reçu :", msg)
  case <-time.After(time.Second):
    fmt.Println("Timeout atteint")
}
```

paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

Quelques packages et bibliothèques

1. La gestion du système de fichiers.
2. Les entrée/sorties simples.
3. Les classes de communication réseau.
4. La réflexion (les concepts et le package associé).
5. TP: Construction d’une petite application mettant en œuvre la réflexion et la communication.

Gestion du système de fichiers

Package os : Offre des fonctionnalités pour interagir avec le système de fichiers.

```
file, err := os.Open("file.txt")
if err != nil {
    log.Fatal(err)
}

file, err := os.Create("newfile.txt")
```

Gestion du système de fichiers

```
data, err := ioutil.ReadFile("file.txt")
if err != nil {
    log.Fatal(err)
}
fmt.Println(string(data))

err := ioutil.WriteFile("file.txt", []byte("Hello, World!"), 0644)
```

Les entrées/sorties simple

```
var input string
fmt.Scanln(&input) // récupération de l’entrée
fmt.Println("Vous avez saisi :", input)

// sortie standard
fmt.Println("Bonjour tout le monde")
```

Les classes de communication réseau

```
// Création d’un serveur tcp
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    log.Fatal(err)
}
conn, err := ln.Accept()

// Etablir une connexion TCP
conn, err := net.Dial("tcp", "localhost:8080")
if err != nil {
    log.Fatal(err)
}
```

Envoie / Réception de donnée

```
// Envoie de donnée
fmt.Fprintf(conn, "Hello from client")

// Reception de donnée
message, _ := bufio.NewReader(conn).ReadString('\n')
fmt.Println("Message du serveur :", message)
```

La réflexion en Go

Package reflect : Permet d'inspecter et de modifier dynamiquement les types et les valeurs des objets.

```
// obtenir le type d’une variable
t := reflect.TypeOf(42)
fmt.Println(t) // int

// Obtenir la valeur d’une variable :
v := reflect.ValueOf(42)
fmt.Println(v) // 42

// Modifier la valeur d'un pointeur :
var x int = 42
v := reflect.ValueOf(&x).Elem()
v.SetInt(100)
fmt.Println(x) // 100
```

La réflexion en Go

Package reflect : Permet d'inspecter et de modifier dynamiquement les types et les valeurs des objets.

Appeler une méthode dynamiquement :

```
type Person struct {
    Name string
}

func (p Person) Greet() {
    fmt.Println("Hello, " + p.Name)
}

func main() {
    p := Person{Name: "Alice"}
    method := reflect.ValueOf(p).MethodByName("Greet")
    method.Call(nil)
}
```

paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

Compléments

1. Les tests unitaires.
2. Les benchmarks.
3. TP : Réalisation d'un ensemble de tests unitaires et de mesures de performances sur une application simple.

Les tests unitaires en Go

- Tests unitaires : Vérifient le comportement des fonctions ou des méthodes isolées.
- Package testing : Utilisé pour écrire et exécuter des tests en Go.
- Les tests sont stockés dans des fichiers se terminant par _test.go.

```
func TestAddition(t *testing.T) {
    result := Add(2, 3)
    if result != 5 {
        t.Errorf("Expected 5, got %d", result)
    }
}
```

Structure du fichier de test

```
package main

import "testing"

func TestMultiply(t *testing.T) { // chaque fonction de test doit commencer par Test
    result := Multiply(2, 3)
    if result != 6 {
        t.Errorf("Expected 6, got %d", result)
    }
}
```

Gestion des erreurs dans les tests

- t.Error() : Enregistre une erreur sans arrêter l'exécution du test.
- t.Fail() : Indique l’échec du test sans arrêter.
- t.Fatal() : Enregistre une erreur et arrête immédiatement le test.

```
if result != expected {
    t.Fatal("Erreur fatale")
}
```

Les tests avec plusieurs cas de figure

```
func TestMultiply(t *testing.T) {
    tests := []struct{
        a, b, expected int
    }{
        {2, 3, 6},
        {0, 5, 0},
        {7, 2, 14},
    }
    for _, tt := range tests {
        result := Multiply(tt.a, tt.b)
        if result != tt.expected {
            t.Errorf("Expected %d, got %d", tt.expected, result)
        }
    }
}
```

Benchmarks en Go

- Benchmarks : Mesurent les performances d’une fonction.
- Créés avec le package testing dans des fonctions qui commencent par Benchmark.

```
func BenchmarkAdd(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Add(2, 3)
    }
}

// go test -bench=.
/// BenchmarkAdd-8    1000000    1000 ns/op
```

paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

Ressources des TP

1. TP : Prise en main Go
2. TP : Creation d’un package
3. TP : Struct & Méthodes
4. TP : Goroutine
5. TP : Système de fichier
6. TP : Tests

1- TP :

Sujet 1 : Installation de Go

Objectif : Installer golang votre machine

- Télécharger et installer Go depuis le site officiel golang.org.
- Configurer la variable d’environnement $GOPATH.
- Vérifier l’installation : go version

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

1- TP : Prise en main Go

Sujet 2 : Création d'un projet Go

Objectif : Créer un projet go

- go mod init monprojet
- Créer le fichier main.go

```
package main

import "fmt"

func main() {
  fmt.Println("Hello, Go!")
}
```

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

2- TP : Création package

Sujet 1 : Création d’un package Go

Objectif : Créer un projet go

- Créer un package mathutils qui contient une fonction d'addition.
- Utiliser ce package dans le programme principal.
- Déclarer des variables et manipuler leurs valeurs.
- Tester la visibilité des fonctions en jouant sur les majuscules.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

2- TP : Création package

Sujet 2 : Création d’un package Go

Objectif :

- Écrire un programme qui vérifie si un nombre est pair ou impair.
- Utiliser une boucle for pour afficher les 10 premiers nombres.
- Parcourir un tableau avec range et afficher chaque élément avec son index.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

2- TP : Création package

Sujet 3 : Création d’un package Go

Objectif :

- Créer une fonction qui retourne deux valeurs (quotient et reste).
- Écrire une fonction variadique qui calcule la somme de plusieurs entiers.
- Déclarer un pointeur et l'utiliser pour modifier la valeur d'une variable dans une fonction.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

3- TP : Struct & Méthodes

Sujet 1 : Création d'une structure

Objectif :

- Créer un struct Car avec des champs Make, Model, et Year.
- Définir une méthode DisplayInfo pour afficher les informations de la voiture.
- Utiliser cette méthode dans une fonction principale.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

3- TP : Struct & Méthodes

Sujet 2 : Création d’une Interface

Objectif :

- Définir une interface Shape avec une méthode Area().
- Implémenter cette interface pour des structs Circle et Rectangle.
- Créer une fonction qui accepte un Shape et affiche sa surface.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

3- TP : Struct & Méthodes

Sujet 2 : Création d’une Interface

Objectif :

- Définir une interface Shape avec une méthode Area().
- Implémenter cette interface pour des structs Circle et Rectangle.
- Créer une fonction qui accepte un Shape et affiche sa surface.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

4- TP : Goroutine

Sujet 1 : Goroutine

Objectif :

- Créer un programme qui lance 3 goroutines effectuant un travail en parallèle.
- Utiliser des channels pour que les goroutines communiquent avec le programme principal.
- Synchroniser l’exécution avec un WaitGroup.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

5- TP : Système de fichier

Sujet 1 : Système de fichier

Objectif :

- Écrire un programme qui lit un fichier texte et affiche son contenu dans la console.
- Ajouter une fonctionnalité qui écrit du texte supplémentaire dans le même fichier.
- Expérimenter avec la création et la suppression de fichiers dans Go.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

6- TP : Tests

Sujet 1 : Tests

Objectif :

- Écrire des tests unitaires pour une application simple de calculatrice (addition, soustraction, multiplication, division).
- Tester les cas particuliers (division par zéro, grandes valeurs, etc.).
- Utiliser plusieurs cas de test dans un même fichier à l'aide des tableaux de test.

Question :

- Avez vous rencontrer une erreur ? Si oui pourquoi ?

paul.millet@lameduse.fr

LaMeDuSe

GOG : Go, le langage de programmation de Google

Sources

Sources :

-