Go (Golang) - Fiche Mémo Complète

Sommaire

- Go (Golang) - Fiche Mémo Complète
  - Sommaire
  - Introduction & Philosophie Go
  - Ce qui caractérise Go
  - Variables et Types
  - Fonctions
  - Méthodes et Récepteurs
  - Structs et Embedding
  - Interfaces
  - Contrôle de Flux
  - Slices et Maps
  - Gestion des Erreurs
  - Concurrence
    - Goroutines
    - Channels
    - Select
    - Mutex
    - Context
  - Génériques
  - Réflexion (Reflection)
  - Tests
    - Tests unitaires
    - Table-driven tests
    - Benchmarks
    - Tests d'exemple
  - Bonnes Pratiques & Astuces

Introduction & Philosophie Go

Go est un langage compilé, simple, performant, créé par Google en 2007. Il vise la simplicité, la rapidité de compilation, la concurrence native et la portabilité.

- Syntaxe minimaliste, facile à lire et à écrire
- Pas d'héritage, mais composition (embedding)
- Gestion explicite des erreurs (pas d'exceptions)
- Outils intégrés (formatage, tests, documentation)

Premier programme :

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

Ce qui caractérise Go

Aspect	Go	Java	Python	JavaScript
Concurrence	Goroutines (2KB)	Threads (1MB)	GIL limitant	Event loop
Compilation	Rapide	Lente	Interprété	Interprété
Déploiement	Binaire unique	JVM requise	Environnement	Runtime
Gestion mémoire	GC simple	GC complexe	GC simple	GC simple
Syntaxe	Minimaliste	Verbose	Simple	Flexible

Go se distingue par sa simplicité, sa rapidité, sa gestion native de la concurrence et son écosystème intégré.

Variables et Types

Go est un langage à typage statique : chaque variable a un type connu à la compilation. Cela permet d'éviter de nombreuses erreurs et d'optimiser les performances.

Déclaration Voici comment déclarer des variables en Go, avec ou sans type explicite :

```go
var nom string = "Alice"    // Explicite
var age = 25                // Inférence
prenom := "Bob"             // Courte (recommandée)
```

- `var` permet de déclarer une variable avec ou sans type explicite.
- `:=` est la forme courte, très utilisée dans les fonctions.

Types principaux

- `int`, `float64`, `bool`, `string` : types de base pour les nombres, booléens et chaînes.
- `array` (tableau de taille fixe), `slice` (tableau dynamique, très utilisé)
- `map` (dictionnaire clé/valeur, très pratique pour les associations)
- `struct` (type personnalisé, pour regrouper des champs)

Valeur zéro Les variables non initialisées reçoivent la "valeur zéro" de leur type (`0` pour les nombres, `""` pour les chaînes, `nil` pour les références).

Struct : Définir un type structuré personnalisé pour regrouper des données :

```go
type Person struct {
    Name string
    Age  int
}
p := Person{Name: "Alice", Age: 30}
fmt.Println(p.Name) // "Alice"
```

Les structs sont la base de la programmation orientée objet en Go (pas de classes, mais des structs et des méthodes).

Fonctions

Les fonctions sont au cœur de Go. Elles permettent de structurer le code, de le réutiliser et de le tester facilement.

Déclaration Définir une fonction simple qui additionne deux entiers :

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

- Les types des paramètres et du retour sont toujours explicités.
- Le nom de la fonction commence par une majuscule si elle doit être exportée (visible hors du package).

Retours multiples Go permet de retourner plusieurs valeurs, très utile pour retourner un résultat et une erreur :

```go
func Divide(a, b int) (int, error) {
    if b == 0 {
        return 0, fmt.Errorf("division par zéro")
    }
    return a / b, nil
}
```

Fonctions variadiques Additionner un nombre variable d'entiers :

```go
func Sum(nums ...int) int {
    total := 0
    for _, n := range nums {
        total += n
    }
    return total
}
fmt.Println(Sum(1, 2, 3, 4)) // 10
```

- Les fonctions variadiques acceptent un nombre quelconque d'arguments du même type.

Fonctions anonymes et closures Définir une fonction sans nom et l'utiliser immédiatement (closure) :

```
f := func(x int) int { return x * x }
fmt.Println(f(5)) // 25
```

Les closures peuvent capturer des variables de l'environnement où elles sont définies.

Méthodes et Récepteurs

En Go, on peut associer des méthodes à n'importe quel type défini par l'utilisateur (struct, type alias, etc.).

Définir des méthodes associées à un type structuré :

```
type Rectangle struct {
    Width, Height int
}

func (r Rectangle) Area() int {
    return r.Width * r.Height
}

func (r *Rectangle) Scale(factor int) {
    r.Width *= factor
    r.Height *= factor
}
```

- Le récepteur (`r` ou `*r`) est le premier paramètre, placé entre parenthèses avant le nom de la fonction.
- Si le récepteur est un pointeur (`*Rectangle`), la méthode peut modifier la struct d'origine.
- Les méthodes rendent le code plus lisible et permettent d'organiser les comportements autour des données.

Structs et Embedding

L'« embedding » permet de composer des types et de réutiliser des champs ou des méthodes d'une struct dans une autre, sans héritage.

Utiliser l'« embedding » pour composer des types et réutiliser des champs :

```go
type Animal struct { Name string }
type Dog struct {
    Animal
    Breed string
}
d := Dog{Animal: Animal{Name: "Rex"}, Breed: "Labrador"}
fmt.Println(d.Name) // "Rex"
```

- Ici, Dog possède directement le champ `Name` grâce à l'embedding de `Animal`.
- L'embedding permet aussi de composer des comportements (méthodes).

Interfaces

Les interfaces définissent un ensemble de méthodes. Toute struct qui implémente ces méthodes satisfait l'interface, sans déclaration explicite.

Définir une interface et l'implémenter implicitement avec une struct :

```go
type Shape interface {
    Area() float64
}

type Circle struct { Radius float64 }

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}

var s Shape = Circle{Radius: 2.0}
fmt.Println(s.Area()) // 12.566...
```

- Les interfaces favorisent la composition et l'abstraction.
- L'interface vide `interface{}` accepte n'importe quel type (équivalent de any).
- Les interfaces sont très utilisées pour la programmation orientée interface (ex : io.Reader, io.Writer).

Contrôle de Flux

Go propose des structures de contrôle classiques, mais avec une syntaxe simple et cohérente.

If Exemple de condition simple :

```
if x > 0 {
    fmt.Println("x est positif")
} else if x < 0 {
    fmt.Println("x est négatif")
} else {
    fmt.Println("x est nul")
}
```

- Les parenthèses autour de la condition sont omises.
- On peut déclarer une variable juste avant la condition: `if err := do(); err != nil { ... }`

For Boucle classique sur un entier:

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
```

- Il n'y a qu'un seul mot-clé `for` pour toutes les boucles (pas de `while`).
- On peut omettre l'initialisation, la condition ou l'incrément pour faire des boucles infinies ou des boucles type `while`.

Boucle sur slice Parcourir un slice avec l'index et la valeur:

```
nums := []int{1, 2, 3}
for idx, val := range nums {
    fmt.Printf("Index %d: %d\n", idx, val)
}
```

Switch Sélectionner un cas parmi plusieurs valeurs:

```
switch day {
case "lundi":
    fmt.Println("Début de semaine")
case "samedi", "dimanche":
    fmt.Println("Week-end")
default:
    fmt.Println("Jour ordinaire")
}
```

- Le `switch` s'utilise sans `break` (il s'arrête automatiquement après le premier cas trouvé).
- On peut tester plusieurs valeurs dans un même `case`.

Slices et Maps

Les slices et les maps sont des structures de données dynamiques très utilisées en Go.

Slice Créer, manipuler et extraire une sous-partie d'un slice :

```
s := []int{1, 2, 3}
s = append(s, 4) // [1 2 3 4]
sub := s[1:3]   // [2 3]
```

- Les slices sont des vues dynamiques sur des tableaux sous-jacents.
- L'opération append retourne un nouveau slice (il peut changer l'adresse sous-jacente).

Copie de slice Copier le contenu d'un slice dans un autre :

```
copySlice := make([]int, len(s))
copy(copySlice, s)
```

Map Créer une map, ajouter des éléments et accéder à une valeur :

```
m := make(map[string]int)
m["Alice"] = 30
m["Bob"] = 25
fmt.Println(m["Alice"]) // 30
```

- Les maps sont non ordonnées et non thread-safe par défaut.

Vérifier la présence d'une clé Tester si une clé existe dans la map :

```
age, ok := m["Charlie"]
if !ok {
    fmt.Println("Charlie n'est pas dans la map")
}
```

Gestion des Erreurs

Go gère les erreurs comme des valeurs, pas comme des exceptions. Cela rend le flux d'exécution explicite et prévisible.

Pattern standard Lire un fichier et gérer l'erreur si le fichier n'existe pas :

```
data, err := os.ReadFile("file.txt")
if err != nil {
    log.Fatalf("Erreur de lecture : %v", err)
}
```

- Toujours vérifier l'erreur immédiatement après l'appel d'une fonction qui peut échouer.

Wrapping d'erreur Ajouter du contexte à une erreur avant de la propager :

```
if err != nil {
    return fmt.Errorf("échec ouverture fichier %s: %w", filename, err)
}
```

- `%w` permet de chaîner les erreurs pour une analyse ultérieure.

Erreurs personnalisées Définir un type d'erreur spécifique :

```
type MyError struct {
    Code int
    Msg  string
}
func (e MyError) Error() string {
    return fmt.Sprintf("Code %d: %s", e.Code, e.Msg)
}
```

- On peut utiliser `errors.Is` et `errors.As` pour tester ou extraire des erreurs spécifiques.

Concurrence

La concurrence est un des points forts de Go. Elle permet d'exécuter plusieurs tâches en même temps, de façon efficace et simple à écrire.

Goroutines

Une goroutine est une fonction qui s'exécute en concurrence avec d'autres goroutines dans le même programme. Elles sont très légères (2KB) et gérées par le runtime Go.

Lancer une goroutine

```
go func() {
    fmt.Println("Tâche en parallèle")
}()
```

Goroutines multiples

```
for i := 0; i < 3; i++ {
    go func(n int) {
        fmt.Println("Goroutine", n)
    }(i)
}
```

Attention : Les goroutines s'exécutent de façon asynchrone. Si le programme principal se termine, toutes les goroutines s'arrêtent immédiatement.

Channels

Les channels permettent la communication et la synchronisation entre goroutines. Ils transportent des valeurs typées.

Créer et utiliser un channel

```
ch := make(chan int)
go func() { ch <- 42 }()
val := <-ch
fmt.Println(val) // 42
```

Channels bufferisés Un channel bufferisé peut stocker plusieurs valeurs sans bloquer immédiatement l'envoi.

```
ch := make(chan string, 2)
ch <- "a"
ch <- "b"
fmt.Println(<-ch) // "a"
fmt.Println(<-ch) // "b"
```

Fermeture d'un channel

```
close(ch)
```

Après fermeture, toute lecture supplémentaire retourne la valeur zéro du type.

Boucle sur un channel

```
for v := range ch {
    fmt.Println(v)
}
```

Select

`select` permet d'attendre sur plusieurs opérations de channel en même temps. Il exécute le premier cas prêt.

Exemple d'utilisation

```
select {
case v := <-ch1:
    fmt.Println("ch1:", v)
case v := <-ch2:
    fmt.Println("ch2:", v)
default:
    fmt.Println("Aucune donnée")
}
```

Timeout avec select

```
timeout := time.After(1 * time.Second)
select {
case v := <-ch:
    fmt.Println("Reçu:", v)
case <-timeout:
    fmt.Println("Timeout!")
}
```

Mutex

Un mutex (mutual exclusion) permet de protéger une section critique, c'est-à-dire un accès concurrent à une variable partagée.

Utilisation d'un mutex

```
var mu sync.Mutex
counter := 0

mu.Lock()
counter++
mu.Unlock()
```

Avec defer pour garantir le déverrouillage

```
mu.Lock()
defer mu.Unlock()
// section critique
```

Exemple complet avec goroutines

```
var mu sync.Mutex
counter := 0
wg := sync.WaitGroup{}
for i := 0; i < 1000; i++ {
    wg.Add(1)
    go func() {
        mu.Lock()
        counter++
        mu.Unlock()
        wg.Done()
    }()
}
wg.Wait()
fmt.Println(counter) // 1000
```

Context

Le package `context` permet de gérer l'annulation, les délais (timeout) et de transmettre des informations entre goroutines, notamment dans les applications serveur.

Créer un contexte avec timeout

```
ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
defer cancel()

select {
case <-ctx.Done():
    fmt.Println("Timeout ou annulation")
case res := <-ch:
    fmt.Println("Résultat :", res)
}
```

Passer un contexte à une fonction

```
func fetchData(ctx context.Context) error {
    select {
    case <-ctx.Done():
        return ctx.Err()
    case <-time.After(1 * time.Second):
        fmt.Println("Données récupérées")
        return nil
    }
}
```

Utilisation typique dans un serveur HTTP

```go
func handler(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    // ...
    select {
    case <-ctx.Done():
        // Client a annulé la requête
    default:
        // Traitement normal
    }
}
```

Génériques

Les génériques (Go 1.18+) permettent d'écrire des fonctions et des types qui fonctionnent avec plusieurs types, tout en gardant la sécurité du typage statique.

Fonction générique pour le minimum

```go
import "golang.org/x/exp/constraints"

func Min[T constraints.Ordered](a, b T) T {
    if a < b { return a }
    return b
}

fmt.Println(Min(3, 7))         // 3
fmt.Println(Min("go", "java")) // "go"
```

Fonction générique pour trouver le maximum dans un slice

```go
func MaxInSlice[T constraints.Ordered](s []T) T {
    max := s[0]
    for _, v := range s[1:] {
        if v > max {
            max = v
        }
    }
    return max
}

fmt.Println(MaxInSlice([]int{1, 5, 2, 9, 3})) // 9
```

Définir un type générique

```go
type Pair[T any, U any] struct {
    First  T
    Second U
}

p := Pair[int, string]{First: 1, Second: "un"}
fmt.Println(p)
```

Contraintes personnalisées

```go
type Stringer interface {
    String() string
}

func PrintAll[T Stringer](s []T) {
    for _, v := range s {
        fmt.Println(v.String())
    }
}
```

Réflexion (Reflection)

La réflexion permet d'inspecter et de manipuler dynamiquement les types et valeurs à l'exécution. C'est utile pour écrire des fonctions génériques, des sérialiseurs, ou des outils comme les ORM.

Afficher le type et la valeur d'une variable

```go
import (
    "fmt"
    "reflect"
)

func Inspect(v interface{}) {
    t := reflect.TypeOf(v)
    val := reflect.ValueOf(v)
    fmt.Println("Type :", t)
    fmt.Println("Valeur :", val)
}

Inspect(42)         // Type : int, Valeur : 42
Inspect("hello")   // Type : string, Valeur : hello
```

Parcourir les champs d'une struct

```go
type User struct {
    Name string
    Age  int
}

func PrintFields(v interface{}) {
    t := reflect.TypeOf(v)
    val := reflect.ValueOf(v)
    for i := 0; i < t.NumField(); i++ {
        field := t.Field(i)
        value := val.Field(i)
        fmt.Printf("Champ %s (%s) = %v\n", field.Name, field.Type, value)
    }
}

u := User{"Alice", 30}
PrintFields(u)
```

Modifier dynamiquement une valeur

```go
func SetField(ptr interface{}, fieldName string, value interface{}) {
    v := reflect.ValueOf(ptr).Elem()
    f := v.FieldByName(fieldName)
    if f.IsValid() && f.CanSet() {
        f.Set(reflect.ValueOf(value))
    }
}

type Point struct { X, Y int }
p := &Point{1, 2}
SetField(p, "X", 10)
fmt.Println(p) // &{10 2}
```

Attention : La réflexion est puissante mais doit être utilisée avec parcimonie (moins performante, code moins lisible).

Tests

Tests unitaires

Fonction de test pour vérifier le résultat d'une addition :

```go
func TestAdd(t *testing.T) {
    result := Add(2, 3)
    if result != 5 {
        t.Errorf("Add(2, 3) = %d; want 5", result)
    }
}
```

Table-driven tests

Tester plusieurs cas dans une même fonction de test :

```go
func TestAdd(t *testing.T) {
    cases := []struct{
        a, b, want int
    }{
        {1, 2, 3},
        {0, 0, 0},
        {-1, 1, 0},
    }
    for _, c := range cases {
        got := Add(c.a, c.b)
        if got != c.want {
            t.Errorf("Add(%d, %d) = %d; want %d", c.a, c.b, got, c.want)
        }
    }
}
```

Benchmarks

Mesurer la performance d'une fonction :

```go
func BenchmarkAdd(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Add(1, 2)
    }
}
```

Tests d'exemple

Fournir un exemple exécutable et vérifiable automatiquement :

```go
func ExampleAdd() {
    fmt.Println(Add(2, 3))
    // Output: 5
}
```

Bonnes Pratiques & Astuces

- Toujours vérifier les erreurs
- Utiliser `go fmt` pour formater le code
- Préférer les slices aux arrays
- Utiliser des noms courts mais explicites
- Documenter les fonctions exportées
- Privilégier la composition à l'héritage