Go : fonctions

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Go

Les fonctions en **Go**

- Un bloc d'instructions réalisant une fonctionnalité bien déterminée.
- Pouvant retourner une ou plusieurs valeurs.
- Pouvant prendre 0 ou plusieurs paramètres.
- Pouvant appeler d'autres fonctions.
- Pouvant retourner une autre fonction.
- Pouvant avoir comme paramètre une autre fonction.

Go

Déclarer une fonction

```
func nomFonction(parametres) typeRetour {
    // instructions
}
```

Go

Déclarer une fonction

```
func nomFonction(parametres) typeRetour {
    // instructions
}
```

Exemple

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

Go

Déclaration simplifiée des paramètres

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

Go

Déclaration simplifiée des paramètres

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

En **Go**, lorsque plusieurs paramètres consécutifs ont le même type, il est possible d'écrire le type une seule fois.

Go

Appeler une fonction

```
package main

import "fmt"

func somme(a, b int) int {
    return a + b
}

func main() {
    resultat := somme(1, 3)
    fmt.Println(resultat)
    // affiche 4
}
```

Go

Appeler une fonction

```
package main

import "fmt"

func somme(a, b int) int {
    return a + b
}

func main() {
    resultat := somme(1, 3)
    fmt.Println(resultat)
    // affiche 4
}
```

En **Go**, les blocs sont délimités par des accolades. Le point d'entrée d'un programme exécutable est la fonction `main` du package `main`.

Go

Exercice 1

Écrire une fonction `maximum2(a, b int) int` qui retourne le maximum entre `a` et `b`.

Go

Exercice 1

Écrire une fonction `maximum2(a, b int) int` qui retourne le maximum entre `a` et `b`.

Exercice 2

Écrire une fonction `maximum3(a, b, c int) int` qui retourne le maximum entre `a`, `b` et `c`.
Vous pouvez utiliser la fonction `maximum2`.

Go

Dans un fichier Go

- Une fonction peut être appelée avant ou après sa déclaration dans le même package.
- L'ordre des fonctions dans un fichier n'a généralement pas d'impact sur l'exécution.
- Il reste conseillé d'organiser le code de manière lisible : fonctions principales, fonctions métier, fonctions utilitaires.

Go

Question 1

Une fonction doit-elle avoir un seul `return` ?

Go

Question 1

Une fonction doit-elle avoir un seul `return` ?

Réponse : non

- Une fonction doit être lisible, simple et avoir une responsabilité claire.
- Une fonction peut avoir plusieurs `return`, surtout pour traiter rapidement les cas particuliers ou les erreurs.

Go

Question 2

Y a-t-il une différence entre paramètres et arguments ?

Go

Question 2

Y a-t-il une différence entre paramètres et arguments ?

Réponse : oui

- Paramètres : les variables définies dans la déclaration de la fonction.
- Arguments : les valeurs passées lors de l'appel de la fonction.
- Les deux termes sont parfois utilisés d'une manière interchangeable, mais ils ne désignent pas exactement la même chose.

Go

Exercice 3

Écrire une fonction `estPremier(n int) bool` qui permet de déterminer si un nombre passé en paramètre est premier.

Go

Exercice 4

Écrire une fonction Go qui retourne la factorielle d'un nombre passé en paramètre.

Go

Exercice 4

Écrire une fonction Go qui retourne la factorielle d'un nombre passé en paramètre.

Factorielle : rappel

- $0! = 1$
- $1! = 1$
- $n! = n * (n-1)!$

Go

Solution

```
func factorielle(n int) int {
    resultat := 1
    for i := 2; i <= n; i++ {
        resultat *= i
    }
    return resultat
}
```

Go

Une version récursive de factorielle

```
func factorielle(n int) int {
    if n == 0 || n == 1 {
        return 1
    }
    return n * factorielle(n-1)
}
```

Go

Une version récursive de factorielle

```
func factorielle(n int) int {
    if n == 0 || n == 1 {
        return 1
    }
    return n * factorielle(n-1)
}
```


Une fonction récursive est une fonction qui s'appelle elle-même.

Go

Exercice

Écrire une fonction récursive :

- acceptant comme paramètre un entier positif `n`
- retournant le nième terme de la suite de Fibonacci.

Go

Exercice

Écrire une fonction récursive :

- acceptant comme paramètre un entier positif `n`
- retournant le nième terme de la suite de Fibonacci.

Suite de Fibonacci

- $U_0 = 0$
- $U_1 = 1$
- $U_n = U_{n-1} + U_{n-2}$

Go

Fonction récursive : avantages

- Clarté du code.
- Élégance mathématique.

Fonction itérative : avantages

- **+** performant et **-** coûteux en mémoire.
- Souvent plus adapté en production.

Go

Fonction récursive : avantages

- Clarté du code.
- Élégance mathématique.

Fonction itérative : avantages

- **+** performant et **-** coûteux en mémoire.
- Souvent plus adapté en production.

Fonction récursive : inconvénients

- Risque de pile d'appels très profonde.
- Plus coûteux en appels de fonction.

Fonction itérative : inconvénients

- Code parfois moins proche de la définition mathématique.
- Peut être moins lisible selon le problème.

Go

Une fonction Go peut retourner plusieurs valeurs

```
func pairsEtImpairs(nombres []int) (int, int) {
    pairs := 0
    impairs := 0

    for _, elt := range nombres {
        if elt%2 == 0 {
            pairs++
        } else {
            impairs++
        }
    }
    return pairs, impairs
}
```

Go

Récupérer plusieurs valeurs de retour

```
func main() {
    nombres := []int{2, 3, 5, 8, 1}

    pairs, impairs := pairsEtImpairs(nombres)

    fmt.Printf("#pairs = %d et #impairs = %d\n", pairs, impairs)
    // affiche #pairs = 2 et #impairs = 3
}
```

Go

Récupérer plusieurs valeurs de retour

```
func main() {
    nombres := []int{2, 3, 5, 8, 1}

    pairs, impairs := pairsEtImpairs(nombres)

    fmt.Printf("#pairs = %d et #impairs = %d\n", pairs, impairs)
    // affiche #pairs = 2 et #impairs = 3
}
```


Les retours multiples sont très utilisés en Go, notamment pour retourner un résultat et une erreur : `valeur, err`.

Go

Exercice

Écrire une fonction `compterVoyellesConsonnes(ch string) (int, int)` qui :

- accepte un paramètre `ch` de type chaîne de caractères
- retourne le nombre de voyelles et le nombre de consonnes dans `ch`
- ignore les espaces

Go

En Go, on peut nommer les valeurs de retour d'une fonction

```
package main

import "fmt"

func somme(x int, y int) (result int) {
        result = x + y
        return
}

func main() {
        total := somme(2, 3)
        fmt.Println(total)
        // affiche 5
}
```

Go

Une fonction peut retourner une autre fonction

```
func b() {
    fmt.Println("b()")
}

func a() func() {
    fmt.Println("a()")
    return b
}

func main() {
    returnedFunction := a()
    returnedFunction()
    // affiche : a() puis b()
}
```

Go

Pour le cas où les fonctions ont des paramètres

```
func multiplierPar(n int) func(int) int {
    return func(x int) int {
        return x * n
    }
}

func main() {
    double := multiplierPar(2)
    fmt.Println(double(5))
    // affiche 10
}
```

Go

Quelques types usuels en Go

- `int`, `int64`, `float64`
- `string`, `bool`
- `[]int`, `[]string` : slices
- `map[string]int` : dictionnaire associatif
- `func(int) int` : type fonction
- `error` : type standard pour représenter les erreurs

Go

Go ne possède pas d'union de type classique pour les fonctions ordinaires

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

Go

Go ne possède pas d'union de type classique pour les fonctions ordinaires

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

Pour accepter plusieurs types, on peut utiliser les **génériques** avec des contraintes de type.

Go

Exemple avec une contrainte générique

```
type Nombre interface {
    int | int64 | float64
}

func somme[T Nombre](a, b T) T {
    return a + b
}

func main() {
    fmt.Println(somme(2, 5))
    // affiche 7

    fmt.Println(somme(2.5, 5.5))
    // affiche 8
}
```

Go

Go ne supporte pas

- les valeurs par défaut pour les paramètres
- les paramètres nommés

Go

L'opérateur . . . peut être utilisé pour

- récupérer un nombre variable d'arguments
- décomposer un slice lors d'un appel de fonction.

Go

Fonction variadique

```
func somme(x int, autres ...int) int {
    for _, elt := range autres {
        x += elt
    }
    return x
}

func main() {
    fmt.Println(somme(2))
    // affiche 2

    fmt.Println(somme(10, 2))
    // affiche 12

    fmt.Println(somme(10, 2, 5))
    // affiche 17
}
```

Go

Unpacking d'un slice avec ...

```
func somme(nombres ...int) int {
    total := 0
    for _, n := range nombres {
        total += n
    }
    return total
}

func main() {
    valeurs := []int{2, 3, 5}
    fmt.Println(somme(valeurs...))
    // affiche 10
}
```

Go

Go passe les arguments par valeur

```
func incrementer(n int) {
    n++
}

func main() {
    x := 10
    incrementer(x)
    fmt.Println(x)
    // affiche 10
}
```

Go

Go passe les arguments par valeur

```
func incrementer(n int) {
    n++
}

func main() {
    x := 10
    incrementer(x)
    fmt.Println(x)
    // affiche 10
}
```

La fonction reçoit une copie de la valeur. Modifier le paramètre ne modifie pas la variable originale.

Go

Utiliser un pointeur pour modifier la valeur originale

```
func incrementer(n *int) {
    (*n)++
}

func main() {
    x := 10
    incrementer(&x)
    fmt.Println(x)
    // affiche 11
}
```

Go

Utiliser un pointeur pour modifier la valeur originale

```
func incrementer(n *int) {
    (*n)++
}

func main() {
    x := 10
    incrementer(&x)
    fmt.Println(x)
    // affiche 11
}
```

`&x` récupère l'adresse de `x`. `*n` permet d'accéder à la valeur pointée.

Go

Attention

Les slices, maps et channels contiennent une référence interne vers des données partagées. Ils sont passés par valeur, mais leurs données sous-jacentes peuvent être modifiées.

Go

Attention

Les slices, maps et channels contiennent une référence interne vers des données partagées. Ils sont passés par valeur, mais leurs données sous-jacentes peuvent être modifiées.


```
func modifier(nombres []int) {
    nombres[0] = 100
}

func main() {
    valeurs := []int{1, 2, 3}
    modifier(valeurs)
    fmt.Println(valeurs)
    // affiche [100 2 3]
}
```

Go

Une fonction peut recevoir une autre fonction en paramètre

```
func appliquer(x int, f func(int) int) int {
    return f(x)
}

func carre(n int) int {
    return n * n
}

func main() {
    fmt.Println(appliquer(5, carre))
    // affiche 25
}
```

Go

Exercice

Écrire une fonction `filtrer` qui :

- accepte un slice d'entiers
- accepte une fonction de test `func(int) bool`
- retourne uniquement les entiers qui vérifient le test

Go

Le nom d'une fonction peut être stocké dans une variable

```
func direBonjour() {
    fmt.Println("Bonjour")
}

func main() {
    f := direBonjour
    f()
    // affiche Bonjour
}
```

Go

Le nom d'une fonction peut être stocké dans une variable

```
func direBonjour() {
    fmt.Println("Bonjour")
}

func main() {
    f := direBonjour
    f()
    // affiche Bonjour
}
```

La variable `f` contient une référence vers la fonction `direBonjour`.

Go

Définir explicitement un type fonction

```
type Operation func(int, int) int

func addition(a, b int) int {
    return a + b
}

func calculer(a, b int, op Operation) int {
    return op(a, b)
}
```

Go

Variable locale

```
func afficher() {
    message := "Bonjour"
    fmt.Println(message)
}
```

Go

Variable locale

```
func afficher() {
    message := "Bonjour"
    fmt.Println(message)
}
```


La variable `message` est locale à la fonction `afficher`. Elle n'est pas accessible en dehors de cette fonction.

Go

Variable globale au package

```
package main

import "fmt"

var compteur int = 0

func incrementer() {
    compteur++
}

func main() {
    incrementer()
    fmt.Println(compteur)
}
```

Go

Attention aux variables globales

- Elles peuvent créer des dépendances cachées.
- Elles rendent les tests plus difficiles.
- Elles peuvent poser des problèmes en concurrence si elles sont modifiées par plusieurs goroutines.

Go

Attention aux variables globales

- Elles peuvent créer des dépendances cachées.
- Elles rendent les tests plus difficiles.
- Elles peuvent poser des problèmes en concurrence si elles sont modifiées par plusieurs goroutines.

Préférer des paramètres, des valeurs de retour, ou des structures passées explicitement.

Go

Portée en Go

- Une variable déclarée dans une fonction est locale à cette fonction.
- Une variable déclarée dans un bloc `if`, `for` ou `switch` peut être limitée à ce bloc.
- Un identifiant déclaré au niveau du package est accessible dans ce package.

Go

Portée en Go

- Une variable déclarée dans une fonction est locale à cette fonction.
- Une variable déclarée dans un bloc `if`, `for` ou `switch` peut être limitée à ce bloc.
- Un identifiant déclaré au niveau du package est accessible dans ce package.


```
func main() {
    if x := 10; x > 0 {
        fmt.Println(x)
    }
    // x n'est plus accessible ici
}
```

Go

Visibilité entre packages

- Un identifiant qui commence par une majuscule est exporté.
- Un identifiant qui commence par une minuscule reste interne au package.

Go

Visibilité entre packages

- Un identifiant qui commence par une majuscule est exporté.
- Un identifiant qui commence par une minuscule reste interne au package.


```
func CalculerTotal() int { // exporté
    return 100
}

func calculerTaxe() int { // non exporté
    return 20
}
```

Go

Une fonction anonyme est une fonction sans nom

```
func main() {
    f := func(a, b int) int {
        return a + b
    }

    fmt.Println(f(2, 3))
    // affiche 5
}
```

Go

Une fonction anonyme est une fonction sans nom

```
func main() {
    f := func(a, b int) int {
        return a + b
    }

    fmt.Println(f(2, 3))
    // affiche 5
}
```


Les fonctions anonymes sont utiles pour les callbacks, les closures et les traitements courts.

Go

Appel immédiat d'une fonction anonyme

```
func main() {
    resultat := func(a, b int) int {
        return a + b
    }(2, 3)

    fmt.Println(resultat)
    // affiche 5
}
```

Go

Closure : fonction qui capture son environnement

```
func compteur() func() int {
    valeur := 0

    return func() int {
        valeur++
        return valeur
    }
}

func main() {
    c := compteur()
    fmt.Println(c()) // 1
    fmt.Println(c()) // 2
}
```

Go

Bonnes pratiques

- Donner un nom clair et précis aux fonctions.
- Garder les fonctions courtes et cohérentes.
- Retourner les erreurs explicitement lorsque l'opération peut échouer.
- Éviter les variables globales inutiles.
- Préférer les types explicites et les signatures simples.

Go

Retourner une erreur

```
func division(a, b float64) (float64, error) {
    if b == 0 {
        return 0, fmt.Errorf("division par zero")
    }
    return a / b, nil
}

func main() {
    resultat, err := division(10, 0)
    if err != nil {
        fmt.Println("Erreur :", err)
        return
    }
    fmt.Println(resultat)
}
```

Go

À retenir

- Les fonctions sont centrales en Go.
- Les paramètres et les valeurs de retour sont typés.
- Les retours multiples sont idiomatiques.
- Les fonctions peuvent être passées en paramètres ou retournées.
- Les fonctions anonymes permettent d'écrire des closures.
- Les erreurs sont généralement retournées explicitement.