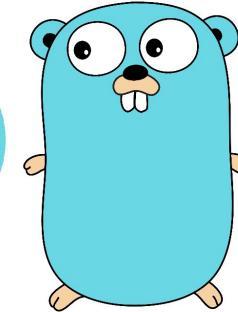


Ready to  ?



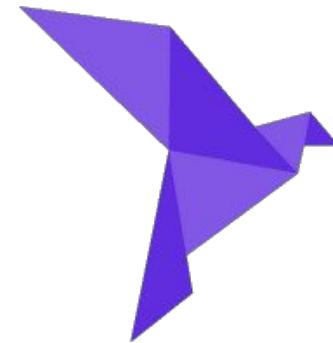
Wait a minute,
who are you?

2

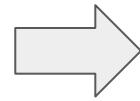
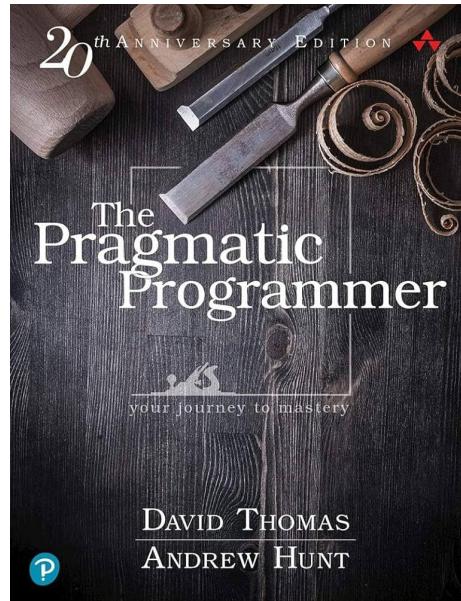
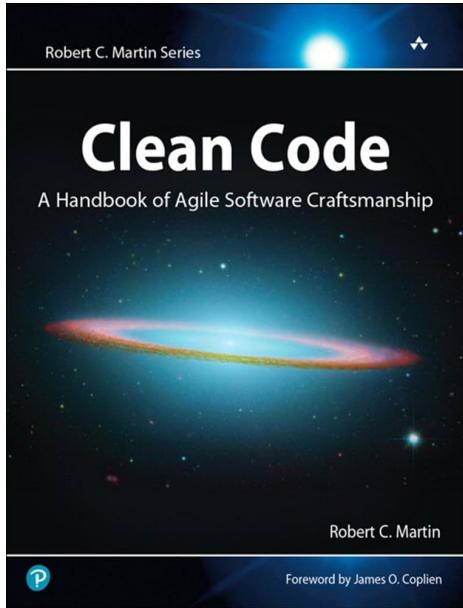
Qui suis-je ?

Un développeur fan de programmation fonctionnel, recherchant constamment :

- Robustesse
- Vélocité
- Développeur experience



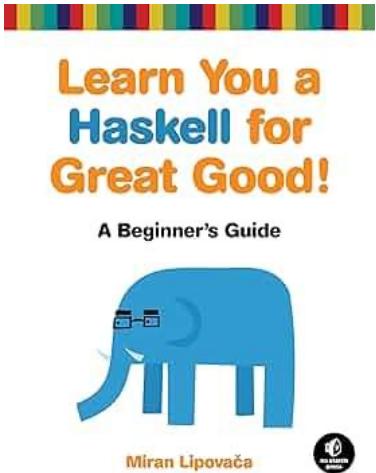
Pour écrire du code robuste et véloce, j'ai tenté de trouver les solutions dans:



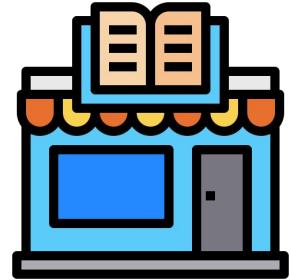
Mon histoire:

Tout a commencé avec 1 question: comment devenir un **développeur x 10**

Réponse:



Ce que j'ai prévu cette année :



-> Cours à l'ESGI

-> Finir mes librairies:

- TS Backend Framework robuste et perfect DX
- Librairie d'authentification (comme oAuth) mais en mieux
- Librairie de caching frontend implémentant le Flyweight Pattern (flyweight for Ram et bandwidth network)
- Readme dynamique
- Orm for Redis
- TS Utils
- TS Pattern matching
- Rust Utils
- Functional implementation as if your life depends on it

-> Contribuer à Gleam ou Roclang

-> Développement de S.A.S.S

Le menu

- Pourquoi Golang
- Connaître tout de golang en 30h
- Découverte de l'écosystème golang
- Optimiser son code Golang

Histoire de Go et son Adoption Actuelle

1. Origines de Go

- **Création en 2007** : Go, souvent appelé *Golang*, a été développé chez Google par **Robert Griesemer, Rob Pike, et Ken Thompson**. Ces créateurs voulaient résoudre plusieurs défis rencontrés dans le développement logiciel moderne, notamment la lenteur des compilations et la gestion complexe de la concurrence.
- **Publication en open source en 2009** : Google a décidé de publier Go en tant que projet open-source, ce qui a permis une adoption plus large dans la communauté des développeurs.
- **Inspirations** :
 - Go combine des éléments de **Python** (simplicité) et de **C** (performance, typage statique).
 - Le design de Go est aussi influencé par d'autres langages tels que **Modula, Oberon, et Limbo**.

2. Contexte de la Création

- **Problèmes adressés :**
 - **Compilations lentes** : À l'époque, les compilations dans des langages comme C++ étaient longues, ce qui ralentissait les développeurs.
 - **Gestion de la concurrence** : La majorité des langages n'offraient pas de solution simple pour exploiter les architectures multicœurs, de plus en plus répandues.
 - **Complexité des systèmes distribués** : Les développeurs avaient besoin d'un langage capable de gérer efficacement la concurrence et les systèmes distribués comme ceux utilisés dans les centres de données de Google.
- **Objectif de Go** : Offrir un langage rapide à compiler, simple à utiliser, tout en restant performant pour les applications modernes.

3. Adoption de Go à travers le temps

- **Années 2010** : La popularité de Go s'est rapidement répandue, en grande partie grâce à son utilisation chez Google, mais aussi par d'autres entreprises souhaitant profiter de ses avantages pour les systèmes distribués et le cloud computing.
- **Projets majeurs utilisant Go :**
 - **Kubernetes** : La plateforme de gestion de conteneurs, essentielle à l'ère du cloud, est entièrement écrite en Go.
 - **Docker** : L'un des projets les plus populaires pour la virtualisation par conteneurs est également écrit en Go.
 - **Prometheus** : Un outil de monitoring populaire dans l'écosystème des microservices.
 - **Etcdb** : Un des systèmes de stockage clé-valeur les plus utilisés dans l'orchestration de clusters
 -

Entreprises adoptant Go :

- En plus de Google, des entreprises comme **Uber**, **Dropbox**, **Twitch**, **Netflix**, **Cloudflare** et bien d'autres utilisent Go pour leurs infrastructures critiques.

Go à ce jour

Popularité croissante :

- Go figure parmi les langages les plus utilisés pour les systèmes distribués et le cloud computing. En 2023, Go reste un langage majeur dans l'écosystème DevOps et cloud.
- La communauté Go est en pleine croissance, avec de nombreux contributeurs open source, meetups, et conférences dédiées.

Langage officiel de nombreuses plateformes cloud :

- De nombreux services cloud tels qu'**AWS Lambda**, **Google Cloud Functions** ou **Azure Functions** supportent Go comme langage de premier choix pour l'écriture de microservices et d'applications serverless.

Évolutions récentes :

- **Go 1.18** : Introduction des **génériques**, une des fonctionnalités les plus demandées qui permet plus de flexibilité et de réutilisabilité dans le code.
- **Go 2.0 (en cours de développement)** : Des discussions sont en cours pour amener de nouvelles fonctionnalités tout en gardant la philosophie de simplicité.

Pourquoi apprendre Go ?

Facile à apprendre : Syntaxe simple avec seulement 25 mots-clés.

Rapide et fiable : Convient aux systèmes distribués (Kubernetes, Docker).

Carrière en plein essor : Go est de plus en plus adopté par des entreprises de toutes tailles.

Langage polyvalent : Utilisé pour le développement backend, le cloud computing, et la science des données.

Installation et Configuration

Télécharger Go : go.dev/dl.

Installer Go :

- MacOS : Exécuter le fichier .pkg et ajouter `/usr/local/go/bin` au PATH.
- Linux : Extraire l'archive dans `/usr/local` et ajouter `go/bin` au PATH.
- Windows : Exécuter le fichier MSI.

Vérifier l'installation :

```
$ go version
```

Hello World

- Structure d'un programme Go :

```
go

package main
import "fmt"
func main() {
    fmt.Println("Hello World!")
}
```

- Explication :

- package main : Indique le point d'entrée du programme.
- import "fmt" : Importe le package pour l'affichage.
- func main : Point d'entrée de l'application.

- Exécution :

```
bash

$ go run main.go
```

Déclaration de Variables

- Déclaration avec `var` :

```
go

var x int
var name string
```

- Initialisation avec `:=` (déclaration implicite) :

```
go

x := 10
name := "John"
```

- Constantes : Utilisées pour déclarer des valeurs immuables.

```
go

const Pi = 3.14
```

Types de Données Primitifs

1. Types Numériques

- **Entiers signés :**
 - `int8, int16, int32, int64` : Entiers avec tailles spécifiques (8, 16, 32, 64 bits).
 - `int` : Taille dépend de l'architecture (32 ou 64 bits).
- **Entiers non signés :**
 - `uint8, uint16, uint32, uint64` : Entiers non signés.
 - `uint` : Taille dépend de l'architecture (32 ou 64 bits).
 - **Alias de `uint8` :** `byte` (utilisé pour des valeurs de type octet).
 - **Alias de `int32` :** `rune` (utilisé pour représenter des caractères Unicode).
- **Nombres à virgule flottante :**
 - `float32, float64` : Types flottants (précision simple et double).
- **Complexes :**
 - `complex64, complex128` : Nombres complexes avec parties réelle et imaginaire (64 bits et 128 bits respectivement).
- Nombre contenant l'adresse d'un pointeur : `uintptr`

2. Types Booléens

- **bool** : Peut être soit `true` soit `false`.

go

 Copy code

```
var isReady bool = true
```

3. Types Texte

- **string** : Chaînes de caractères Unicode.

go

 Copy code

```
var message string = "Hello, Go!"
```

4. Types Spéciaux

- **nil** : Valeur zéro utilisée pour les pointeurs, slices, maps, interfaces, channels, etc. Elle indique une absence de valeur.

1. Tableaux (Arrays)

Types Composés

- Tableau de taille fixe contenant des éléments de même type.

go

```
var arr [5]int = [5]int{1, 2, 3, 4, 5}
```

2. Slices

- Segment dynamique d'un tableau.

go

```
var slice []int = []int{1, 2, 3, 4}
```

3. Structs

- Structures de données qui regroupent plusieurs types dans un ensemble.

```
go

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

4. Pointeurs

- Référence à une valeur mémoire.

```
go

var p *int = &x
```

5. Maps

- Collection clé-valeur non ordonnée.

go

```
var m map[string]int = map[string]int{"apple": 1, "banana": 2}
```

6. Interfaces

- Définissent des comportements que les types doivent implémenter.

go

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

7. Channels

- Utilisés pour la communication entre goroutines.

go

```
var ch chan int = make(chan int)
```

Valeurs Zéro par Défaut (quand on déclare une variable)

En Go, les variables non initialisées sont automatiquement affectées à une **valeur zéro** selon leur type :

- **int** : `0`
- **float** : `0.0`
- **bool** : `false`
- **string** : `""` (chaîne vide)
- **pointeurs, slices, maps, channels** : `nil`

Conversion de Types

Go est fortement typé, donc la conversion explicite est nécessaire :

Contrôle de Flux

Type	Syntax
Logical	&& !
Equality	== !=

```
func main() {
    x := 10

    if x > 5 {
        fmt.Println("x is gt 5")
    } else if x > 10 {
        fmt.Println("x is gt 10")
    } else {
        fmt.Println("else case")
    }
}
```

If plus compact

=> Initialisation de variable et if dans la même ligne

```
func main() {
    if x := 10; x > 5 {
        fmt.Println("x is gt 5")
    }
}
```

2. switch

```
go

fruit := "apple"
switch fruit {
case "apple":
    fmt.Println("This is an apple")
case "banana":
    fmt.Println("This is a banana")
default:
    fmt.Println("Unknown fruit")
}
// Affiche : This is an apple
```

Switch compact

```
switch day := "monday"; day {
    case "monday":
        fmt.Println("time to work!")
    case "friday":
        fmt.Println("let's party")
    default:
        fmt.Println("browse memes")
}
```

Switch remplaçant des if successifs

```
x := 10

switch {
    case x > 5:
        fmt.Println("x is greater")
    default:
        fmt.Println("x is not greater")
}
```

For

```
for i := 0; i < 10; i++ {  
    if i < 2 {  
        continue  
    }  
  
    fmt.Println(i)  
  
    if i > 5 {  
        break  
    }  
}  
  
fmt.Println("We broke out!")
```

Infinite Loop

```
func main() {
    for {
        // do stuff here
    }
}
```

5. Utilisation de continue

go

```
for i := 0; i < 5; i++ {
    if i == 3 {
        continue // Passe directement à l'itération suivante
    }
    fmt.Println(i)
}
// Affiche : 0 1 2 4
```

6. Utilisation de range (boucles sur slices ou maps)

=> On obtient un tuple pour la current value d'un Map et Slice

```
go
```

```
nums := []int{10, 20, 30}
for i, num := range nums {
    fmt.Println(i, num)
}
// Affiche : 0 10
//           1 20
//           2 30
```

```
go
```

```
m := map[string]int{"a": 1, "b": 2}
for key, value := range m {
    fmt.Println(key, value)
}
// Affiche : a 1
//           b 2
```

Functions

```
func myFunction() {}
```

```
func main() {
    myFunction("Hello")
}

func myFunction(p1 string) {
    fmt.Println(p1)
}
```

Retour de fonction

Après les arguments, entre parenthèse on doit obligatoirement spécifier les types

```
func myFunction(p1 string) (string, int) {  
    msg := fmt.Sprintf("%s function", p1)  
    return msg, 10  
}
```

Named return of function

```
func myFunction(p1 string) (s string, i int) {  
    s = fmt.Sprintf("%s function", p1)  
    i = 10  
  
    return  
}
```

Exercices

<https://github.com/blueprismo/100-golang-exercices/tree/main>

<https://exercism.org/tracks/go>

Exercices et cours : <https://github.com/exercism/gO>

Installer la cli de exercism: <https://exercism.org/cli-walkthrough>

Explication exercism:

<https://exercism.org/docs/using/solving-exercises/working-locally>

Ajouter une méthode

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

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

Muter la struct dans une méthode

```
func (p *Person) HaveBirthday() {  
    p.Age++  
}
```

Utilisation des méthodes

```
func main() {  
    person := Person{Name: "Alice", Age: 25}  
  
    // Appel de la méthode Greet  
    person.Greet() // Affiche : Hello, my name is Alice  
  
    // Appel de la méthode HaveBirthday  
    person.HaveBirthday()  
    fmt.Println(person.Age) // Affiche : 26  
}
```

Types passés par valeur

- **Types primitifs :**
 - `int`, `int8`, `int16`, `int32`, `int64`
 - `uint`, `uint8` (alias de `byte`), `uint16`, `uint32`, `uint64`
 - `float32`, `float64`
 - `complex64`, `complex128`
 - `bool`
 - `string`

Types passés par valeur

```
func modifyValue(x int) {  
    x = 10  
}  
  
func main() {  
    a := 5  
    modifyValue(a)  
    fmt.Println(a) // Affiche : 5, car une copie a été modifiée  
}
```

Arrays (Tableaux)

```
func modifyArray(arr [3]int) {
    arr[0] = 100
}

func main() {
    a := [3]int{1, 2, 3}
    modifyArray(a)
    fmt.Println(a) // Affiche : [1 2 3], car une copie a été modifiée
}
```

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

func modifyPerson(p Person) {
    p.Age = 30
}

func main() {
    person := Person{Name: "Alice", Age: 25}
    modifyPerson(person)
    fmt.Println(person.Age) // Affiche : 25, car la copie a été modifiée
}
```

Types passés par référence par défaut

- Slices
- Maps
- Channels

Revenons aux pointeurs

```
package main

import "fmt"

func main() {
    a := 42
    p := &a // Utilisation de & pour obtenir l'adresse de a
    fmt.Println("Adresse de a :", p) // Affiche l'adresse de a
}
```

Le symbole `*` : L'opérateur de déréférencement

```
package main

import "fmt"

func main() {
    a := 42
    p := &a // p est un pointeur vers a
    fmt.Println("Valeur de a :", *p) // Utilisation de * pour accéder à la valeur

    *p = 100 // Modifie la valeur de a via le pointeur
    fmt.Println("Nouvelle valeur de a :", a) // Affiche : 100
}
```

Paramètres et retour de fonctions fixes)

Sans mutation (par valeur) :

```
go

package main

import "fmt"

func modifyInt(x int) {
    x = 100 // Cela modifie seulement la copie
}

func main() {
    a := 42
    modifyInt(a)
    fmt.Println(a) // Affiche : 42, car la copie a été modifiée
}
```

Avec mutation (par référence) :

```
go

package main

import "fmt"

func modifyInt(x *int) {
    *x = 100 // Modifie la valeur d'origine
}

func main() {
    a := 42
    modifyInt(&a) // Passe l'adresse de a
    fmt.Println(a) // Affiche : 100, car a a été modifié
}
```

2. String

Sans mutation (par valeur) :

```
go                                Copier

package main

import "fmt"

func modifyString(s string) {
    s = "Bonjour"
}

func main() {
    str := "Hello"
    modifyString(str)
    fmt.Println(str) // Affiche : Hello, car la copie a été modifiée
}
```

Avec mutation (par référence) (Note : Les chaînes de caractères sont immuables en Go, donc on doit utiliser un pointeur pour changer la référence) :

go

 Copy code

```
package main

import "fmt"

func modifyString(s *string) {
    *s = "Bonjour"
}

func main() {
    str := "Hello"
    modifyString(&str)
    fmt.Println(str) // Affiche : Bonjour
}
```

3. Tableaux (Arrays)

Sans mutation (par valeur) :

go

 Copy code

```
package main

import "fmt"

func modifyArray(arr [3]int) {
    arr[0] = 100
}

func main() {
    a := [3]int{1, 2, 3}
    modifyArray(a)
    fmt.Println(a) // Affiche : [1 2 3], car une copie a été modifiée
}
```

Avec mutation (par référence) (passer un pointeur vers le tableau) :

go

 Copy code

```
package main

import "fmt"

func modifyArray(arr *[3]int) {
    arr[0] = 100
}

func main() {
    a := [3]int{1, 2, 3}
    modifyArray(&a) // Passe un pointeur vers le tableau
    fmt.Println(a) // Affiche : [100 2 3], car le tableau a été modifié
}
```

go

 Copy code

```
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

func modifyStruct(p Person) {
    p.Age = 30
}

func main() {
    person := Person{Name: "Alice", Age: 25}
    modifyStruct(person)
    fmt.Println(person.Age) // Affiche : 25, car la copie a été modifiée
}
```

Avec mutation (par référence) :

```
go
```

 Copy code

```
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

func modifyStruct(p *Person) {
    p.Age = 30
}

func main() {
    person := Person{Name: "Alice", Age: 25}
    modifyStruct(&person)
    fmt.Println(person.Age) // Affiche : 30, car l'original a été modifié
}
```



Passage des variables (et les structures) par référence dans les fonctions (exemples Go)

6. Maps

Les **maps** en Go sont toujours passées par référence, donc elles peuvent être modifiées directement dans une fonction.

Avec mutation (par référence) :

```
go
Copy code

package main

import "fmt"

func modifyMap(m map[string]int) {
    m["key"] = 100
}

func main() {
    m := map[string]int{"key": 1}
    modifyMap(m)
    fmt.Println(m) // Affiche : map[key:100], car la map a été modifiée
}
```

Sans mutation (en créant une copie explicite) :

```
go

package main

import "fmt"

func modifyMap(m map[string]int) {
    mCopy := make(map[string]int)
    for k, v := range m {
        mCopy[k] = v
    }
    mCopy["key"] = 100
    fmt.Println("mCopy:", mCopy) // Affiche : map[key:100], modifié
}

func main() {
    m := map[string]int{"key": 1}
    modifyMap(m)
    fmt.Println(m) // Affiche : map[key:1], inchangé
}
```



8. Channels

Les **channels** sont toujours passés par référence, donc toute modification affecte l'original.

Avec mutation (par référence) :

```
go Copy code

package main

import "fmt"

func sendToChannel(ch chan int) {
    ch <- 100
}

func main() {
    ch := make(chan int, 1)
    sendToChannel(ch)
    fmt.Println(<-ch) // Affiche : 100, car la valeur a été envoyée au canal
}
```

Récapitatif: Passer par valeur ou référence

- **Types primitifs (int, float, bool, string, etc.) et structs** sont passés par **valeur**, donc ils ne sont pas modifiés à moins de passer un **pointeur**.
- **Slices, maps, channels** sont passés par **référence**, donc ils sont modifiés directement sans pointeur.
- **Pointeurs** permettent de modifier la donnée originale directement.

Faut il passer par valeur ou référence ?

Passage par valeur : Une **copie** de la donnée est transmise à la fonction.

Passage par référence : Un **pointeur** vers la donnée est transmis, permettant de modifier la donnée d'origine.

Le choix entre les deux dépend de la taille de la donnée et du comportement attendu (modification ou non).

Différence entre Stack et Heap

Stack (pile) :

- Mémoire rapide d'accès, limitée en taille.
- Utilisée pour les variables locales et les appels de fonction.
- Allocation et libération automatiques.
- **Temps d'accès : 10 à 100 nanosecondes.**

Heap (tas) :

- Mémoire plus grande mais plus lente.
- Utilisée pour les allocations dynamiques et les objets volumineux.
- Gestion par le garbage collector.
- **Temps d'accès : 100 à 1000 nanosecondes.**

Accès à la Stack est plus rapide que la **Heap** car elle utilise un modèle LIFO (Last In, First Out), contrairement à la gestion dynamique de la Heap.

Pour les types fixes

Pour (boolean, string, int) => passer par valeur

Pour (Array, Struct) => passer par valeur sauf si
c'est très grand

Qu'est ce que grand ? plus de 15 keys pour Struct,
Array avec plus de 5000 éléments

Pour les types fixes

On peut passer par référence si on veut absolument que notre fonction ou méthode mute notre donnée



La plupart du temps => Muter n'est pas utile,
c'est même dangereux pour la robustesse de
votre code

Pour les types mobiles (slice/map)

On a pas le choix, c'est déjà passé par référence, donc si on ne veut pas les muter, soit on les clone en utilisant cette fonction ou en faisant ce qu'il y a dedans.

```
func cloneSlice(s []int) []int {  
    copy := make([]int, len(s))  
    copy(copy, s)  
    return copy  
}
```

Pour les types mobiles (slice/map)

Soit on recrée le type à par

6. Maps

Les **maps** en Go sont toujours passées par référence, donc elles peuvent être modifiées directement dans une fonction.

Avec **mutation (par référence)** :

```
go                                ⚡ Copy code

package main

import "fmt"

func modifyMap(m map[string]int) {
    m["key"] = 100
}

func main() {
    m := map[string]int{"key": 1}
    modifyMap(m)
    fmt.Println(m) // Affiche : map[key:100], car la map a été modifiée
}
```

Gestion des Erreurs en Go

Go adopte une approche simple et explicite pour la gestion des erreurs.

L'accent est mis sur l'utilisation d'une **valeur de retour** distincte pour indiquer les erreurs, plutôt que les exceptions comme dans d'autres langages (Java, Python).

Chaque fonction qui peut échouer retourne une **valeur d'erreur**.

```
func doSomething() (int, error) {  
    if someCondition {  
        return 0, fmt.Errorf("something went wrong")  
    }  
    return 42, nil // nil signifie pas d'erreur  
}
```

Le Type `error` en Go

En Go, les erreurs sont représentées par le type `error`, une interface pré définie dans le package `errors`.

Une erreur est simplement une valeur qui implémente l'interface `error`, définie comme suit :

```
type error interface {
    Error() string
}
```

Toute struct avec la méthode
Error peut donc être utilisé
comme error

Retourner une error

```
import "errors"

func myFunction() error {
    return errors.New("an error occurred")
}
```

Gestion des Erreurs dans Plusieurs Retours de Valeurs

```
go

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

result, err := divide(10, 0)
if err != nil {
    fmt.Println("Error:", err) // Affiche : Error: division by zero
}
```

Le Mot-Clé `defer`

Le mot-clé `defer` en Go permet:

- retarder l'exécution d'une fonction jusqu'à ce que la fonction environnante se termine.

En d'autres termes, les instructions marquées avec `defer` ne sont exécutées qu'une fois que la fonction courante (celle dans laquelle `defer` est appelé) a fini d'exécuter toutes ses autres instructions, même en cas de retour anticipé ou d'erreur.

Exemple simple avec `defer` :

```
go

package main

import "fmt"

func main() {
    fmt.Println("Start")

    defer fmt.Println("Deferred function executed")

    fmt.Println("End")
}
```

Toutes les autres instructions de la fonction `main`. Ici, de code `defer` apparaît avant.

Résultat :

```
sql

Start
End
Deferred function executed
```



6. Utilisation de `defer` dans les boucles (attention aux pièges)

Un piège courant est l'utilisation de `defer` dans une boucle. Comme `defer` est exécuté à la fin de la fonction contenant la boucle, tous les appels différés seront exécutés **après** la fin de la boucle.

Exemple d'utilisation incorrecte de `defer` dans une boucle :

```
go
Copy code

package main

import "fmt"

func main() {
    for i := 0; i < 3; i++ {
        defer fmt.Println(i)
    }
    fmt.Println("End of function")
}
```

Résultat:
End of function
2, 1, 0

Stopper le programme si erreur fatale (`panic` et `recover`)

panic : Arrête immédiatement l'exécution de la fonction courante et remonte dans la pile des appels, provoquant la fin du programme s'il n'est pas récupéré. (comme un throw error)

recover : Permet de récupérer d'un `panic` dans une fonction `defer`.

Quand utiliser `panic` ?

- **Erreurs irrécupérables** : comme des erreurs logiques, des corruptions de mémoire, ou des conditions inattendues (ex : accès hors limites).

```
func riskyFunction() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered from panic:", r)
        }
    }()
    panic("something went wrong")
}

func main() {
    riskyFunction()
    fmt.Println("Continuing execution...")
}
```

Utilisation de `fmt.Errorf` avec `%w` et `%v`

- `fmt.Errorf` permet de formater des erreurs tout en ajoutant des détails.
- Le format `%w` est utilisé pour **chaîner** les erreurs (wrapping) afin de conserver l'erreur originale tout en ajoutant du contexte.

Exemple :

```
go                                Copy code

func readFile(filename string) error {
    _, err := os.Open(filename)
    if err != nil {
        return fmt.Errorf("failed to open file %s: %w", filename, err)
    }
    return nil
}
```

- Si l'ouverture du fichier échoue, l'erreur renvoyée contiendra le message `"failed to open file"` ainsi que l'erreur d'origine renvoyée par `os.Open`.

La parallélisation en Golang

(Go utilise un **planificateur** interne pour gérer des milliers de goroutines de manière efficace, tout en utilisant un nombre limité de threads OS)

Q

- Une **goroutine** est une fonction qui s'exécute de manière **concurrente** avec d'autres goroutines dans le même programme.
- Les goroutines sont **légères**, avec un coût très faible en termes de mémoire (environ 2 kB de pile).
- Le mot-clé `go` est utilisé pour démarrer une goroutine.

Exemple simple :

```
go
go myFunction()
```

 Copy code

- Cela démarre `myFunction` comme une goroutine, qui s'exécute **en parallèle** du reste du programme.

Code :

```
go

package main

import (
    "fmt"
    "sync"
)

func printMessage(message string, wg *sync.WaitGroup) {
    defer wg.Done() // Indique que la goroutine est terminée
    fmt.Println(message)
}

func main() {
    var wg sync.WaitGroup
    messages := []string{"Hello", "World", "Go"}

    for _, msg := range messages {
        wg.Add(1)
        go printMessage(msg, &wg) // Passe msg à la goroutine
    }

    wg.Wait() // Attend que toutes les goroutines soient terminées
}
```

Qu'est ce que ça affiche ???

1, 2 , 3 ?

Non !

Ca affichera: Go, World, Hello

Pourquoi ??

Explication du comportement inattendu :

- Dans la boucle `for`, la variable `msg` est réutilisée à chaque itération.
- Chaque goroutine accède à la **même référence** de `msg`, et comme la boucle se termine avant que les goroutines ne s'exécutent, elles capturent toutes la **dernière valeur** de `msg` : "Go".
- Par conséquent, toutes les goroutines affichent "Go" au lieu de "Hello", "World", et "Go".

```
package main

import (
    "fmt"
    "sync"
)

func printMessage(message string, wg *sync.WaitGroup) {
    defer wg.Done() // Indique que la goroutine est terminée
    fmt.Println(message)
}

func main() {
    var wg sync.WaitGroup
    messages := []string{"Hello", "World", "Go"}

    for _, msg := range messages {
        wg.Add(1)
        go func(m string) {
            printMessage(m, &wg) // Passe la variable msg comme argument
        }(msg)
    }

    wg.Wait() // Attend que toutes les goroutines soient terminées
}
```

Résultat:
Hello World Go

Pourquoi ??

Explication de la solution : (L'utilisation du fonction anonyme s'auto appellant)

- La fonction anonyme prend `msg` comme paramètre (`func(m string)`), ce qui crée une **copie distincte** de `msg` pour chaque itération de la boucle.
- Chaque goroutine dispose ainsi de sa **propre copie** de `msg`, et non d'une référence partagée.
- Cela résout le problème de capture de variable dans la boucle `for`, garantissant que chaque goroutine affiche le bon message : "Hello", "World", et "Go".

Channel (transmettre des données)

Un **channel** est un moyen de communication entre **goroutines** en Go.

Il permet d'envoyer et de recevoir des **données** entre les goroutines de manière synchronisée et sécurisée.

Les channels garantissent que la transmission de données entre les goroutines est bien orchestrée, en évitant les conflits d'accès aux variables partagées.

```
ch := make(chan int) // Channel pour des entiers
```

Envoi et Réception dans un Channel :

- **Envoi** : L'opération de l'envoi dans un channel bloque la goroutine jusqu'à ce qu'une autre goroutine soit prête à recevoir la donnée.

go

 Copy code

```
ch <- valeur // Envoi de la valeur dans le channel
```

- **Réception** : L'opération de réception bloque la goroutine jusqu'à ce qu'une autre goroutine envoie une donnée dans le channel.

go

 Copy code

```
valeur := <-ch // Réception de la valeur depuis le channel
```

Exemple simple d'utilisation :

```
go                                ⌂ Copy code

package main

import (
    "fmt"
)

func sendMessage(ch chan string) {
    ch <- "Hello from Goroutine" // Envoie d'un message dans le channel
}

func main() {
    ch := make(chan string)

    go sendMessage(ch)

    message := <-ch // Réception du message depuis le channel
    fmt.Println(message) // Affiche : Hello from Goroutine
}
```

Attente des Goroutines avec `sync.WaitGroup`

- Les **goroutines** s'exécutent de manière **asynchrone**, donc il est souvent nécessaire d'attendre qu'elles se terminent avant de continuer l'exécution du programme.
- Le type **`sync.WaitGroup`** permet de gérer cette attente.

Fonctionnement de `WaitGroup` :

1. **`wg.Add(n)`** : Indique qu'il y a **n** goroutines à attendre.
2. **`wg.Done()`** : Appelé à la fin de chaque goroutine pour signaler qu'elle est terminée.
3. **`wg.Wait()`** : Bloque l'exécution jusqu'à ce que toutes les goroutines aient appelé `Done()`.

Exemple :

go

 Copy code

```
package main

import (
    "fmt"
    "sync"
)

func sendData(ch chan int, wg *sync.WaitGroup) {
    defer wg.Done() // Signale la fin de la goroutine
    ch <- 42 // Envoie la donnée dans le channel
}

func main() {
    var wg sync.WaitGroup
    ch := make(chan int)

    wg.Add(1)
    go sendData(ch, &wg)

    wg.Wait() // Attend la fin de la goroutine
    data := <-ch // Reçoit la donnée du channel
    fmt.Println(data) // Affiche : 42
}
```

Appels asynchrones

```
import (
    "fmt"
    "time"
)

// Fonction asynchrone qui renvoie un string via un channel
func asyncFunction(ch chan string) {
    time.Sleep(2 * time.Second) // Simule une tâche asynchrone
    ch <- "Tâche asynchrone terminée!" // Envoie le résultat dans le channel
}

func main() {
    ch := make(chan string) // Crée un channel pour recevoir un string

    // Lancer la fonction de manière asynchrone
    go asyncFunction(ch)

    // Tâche principale
    fmt.Println("Tâche principale exécutée")

    // Attendre et capturer la valeur renvoyée par la goroutine
    result := <-ch // Bloque jusqu'à la réception du résultat depuis le channel
    fmt.Println(result) // Affiche le résultat de asyncFunction
}
```

Concurrence avec Variables Partagées

Problème de Concurrence avec Variables Partagées

- En Go, la **concurrence** permet d'exécuter plusieurs tâches simultanément via des **goroutines**.
- **Problème** : Lorsque plusieurs goroutines accèdent à des **variables partagées**, cela peut entraîner des **conditions de course** (*race conditions*).
- Une **condition de course** se produit lorsque plusieurs goroutines lisent et modifient une même variable en même temps, ce qui peut provoquer des **résultats inattendus**.

Exemple de Condition de Course (race condition)

Les goroutines modifient simultanément la variable **counter**.

Résultat : **Valeur incorrecte** à cause d'une condition de course.

Si plusieurs incrémentations sont écrasées par des lectures simultanées, la valeur pourrait être quelque chose comme **1700, 1800**, ou même moins.

Exemple sans synchronisation :

```
go
Copy code

package main

import (
    "fmt"
    "time"
)

var counter int

func increment() {
    for i := 0; i < 1000; i++ {
        counter++ // Modification concurrenente
    }
}

func main() {
    go increment()
    go increment()

    time.Sleep(1 * time.Second) // Attente de la fin des goroutines
    fmt.Println(counter) // Résultat inattendu
}
```

4 solutions pour la race condition

- Mutex
- RWMutex
- Sync / atomic
- Channels

Solution avec sync.Mutex (Mutual Exclusion)

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var counter int
var mu sync.Mutex

func increment() {
    for i := 0; i < 1000; i++ {
        mu.Lock() // Verrouille l'accès à la variable
        counter++
        mu.Unlock() // Déverrouille après modification
    }
}

func main() {
    go increment()
    go increment()

    time.Sleep(1 * time.Second)
    fmt.Println(counter) // Affiche 2000, valeur correcte
}
```

Utilisation de `sync.RWMutex` pour Lecture/Écriture

`sync.RWMutex` permet d'optimiser l'accès en lecture multiple tout en contrôlant l'accès en écriture.

- **RLock()** : Verrouille pour permettre plusieurs lectures concurrentes.
- **Lock()** : Verrouille pour une écriture exclusive.

```
var counter int
var rwMu sync.RWMutex

func readCounter() int {
    rwMu.RLock() // Verrouille pour lecture
    defer rwMu.RUnlock()
    return counter
}

func writeCounter() {
    rwMu.Lock() // Verrouille pour écriture
    defer rwMu.Unlock()
    counter++
}

func main() {
    go writeCounter()
    go writeCounter()

    rwMu.RLock()
    fmt.Println(readCounter()) // Lecture simultanée sécurisée
    rwMu.RUnlock()
}
```

Utilisation de `sync/atomic` pour Opérations Simples

Le package `sync/atomic` permet d'effectuer des opérations atomiques (non interrompues) sur des variables simples comme `int64`.

- `atomic.AddInt64` : Incrémente une variable de manière atomique.
- `atomic.LoadInt64` : Lit la variable de manière atomique.

Peut s'utiliser avec :

`int32`

- `int64`
- `uint32`
- `uint64`

```
var counter int64

func increment() {
    for i := 0; i < 1000; i++ {
        atomic.AddInt64(&counter, 1)
    }
}

func main() {
    go increment()
    go increment()

    time.Sleep(1 * time.Second)
    fmt.Println(counter) // Affiche 2000
}
```

Channels (Communication entre goroutines)

Les **channels** sont utilisés pour synchroniser les goroutines et échanger des données. Une goroutine bloqué jusqu'à ce que l'autre envoie ou reçoit une donnée, garantissant une communication synchronisée.

```
func incrementer(counterChan chan int, wg *sync.WaitGroup) {
    defer wg.Done() // Signale que la goroutine est terminée

    for i := 0; i < 1000; i++ {
        // Récupérer la valeur actuelle du compteur depuis le channel
        counter := <-counterChan
        // Incrémenter le compteur
        counter++
        // Envoyer la nouvelle valeur dans le channel
        counterChan <- counter
    }
}

func main() {
    var wg sync.WaitGroup
    counterChan := make(chan int, 1) // Crée un channel bufferisé pour le compteur

    counterChan <- 0 // Initialiser le compteur à 0

    // Lancer 5 goroutines qui vont incrémenter le compteur
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go incrementer(counterChan, &wg)
    }

    // Attendre que toutes les goroutines se terminent
    wg.Wait()

    // Lire la valeur finale du compteur
    finalCounter := <-counterChan
    fmt.Println("Valeur finale du compteur:", finalCounter) // Affiche 5000
}
```

Méthode	Sécurité Concurrente	Facilité d'Utilisation	Performance	Overhead	Cas d'Utilisation Optimal
<code>sync.Mutex</code>	Très sécurisée	Simple	Bonne performance pour les accès concurrents	Faible (verrouillage/déverrouillage)	Sections critiques courtes avec des lectures et écritures fréquentes.
<code>sync.RWMutex</code>	Très sécurisée (optimisée pour lectures)	Moyennement simple	Performant pour les lectures multiples	Légèrement plus élevé que <code>Mutex</code> (gestion des lectures/écritures)	Cas avec de nombreuses lectures simultanées et peu d'écritures.
<code>sync/atomic</code>	Très sécurisée	Facile à utiliser, mais limité aux types primitifs	Excellent (opérations supportées directement par le processeur)	Très faible, car pas de verrou explicite	Opérations simples (incrémentation, lecture/écriture) sur des types primitifs (<code>int32</code> , <code>int64</code> , etc.).
Channels	Très sécurisée, synchronisation implicite	Simple et idiomatique en Go	Moins performant que <code>Mutex</code> et <code>atomic</code> pour accès direct aux données	Overhead plus élevé (gestion des échanges et de la communication)	Synchronisation et communication entre goroutines, transmission de données plutôt que protection

Conclusion

sync/atomic est le plus performant pour des opérations simples sur des types primitifs, mais il ne convient pas aux structures de données complexes.

sync.Mutex est une solution polyvalente et performante pour synchroniser l'accès aux sections critiques.

sync.RWMutex est optimal lorsque **les lectures sont beaucoup plus fréquentes** que les écritures, offrant de meilleures performances dans ces scénarios.

Channels sont idéaux pour des **modèles de communication** entre goroutines, mais sont moins performants pour les accès directs aux données partagées.

Switch sur channel (select)

```
select {
    case msg := <-ch1:
        fmt.Println("Message reçu depuis ch1 :", msg)
    case msg := <-ch2:
        fmt.Println("Message reçu depuis ch2 :", msg)
    case ch3 <- "Hello":
        fmt.Println("Message envoyé vers ch3")
    default:
        fmt.Println("Aucune communication prête, exécution du bloc par défaut")
}
```

Génériques

Pourquoi avons-nous besoin de génériques ?

```
// identifyInt retourne la valeur d'un entier
func identifyInt(value int) int {
    return value
}
```

```
// identifyBool retourne la valeur d'un booléen
func identifyBool(value bool) bool {
    return value
}
```

```
// identifyString retourne la valeur d'une chaîne de caractères
func identifyString(value string) string {
    return value
}
```

Généralisons cette fonction !!!

```
// Identité est une fonction générique qui accepte n'importe quel type T
func Identité[T any](value T) T {
    return value
}
```

resultInt := identifyInt(42) // resultInt est de type int

resultBool := identifyBool(true) // resultBool est de type bool

resultString := identifyString("Hello, Go!") // resultString est de type string

Spécifier les génériques

1. Aucune contrainte (`any`)

L'absence de contrainte est représentée par le type spécial `any`, qui est un alias pour `interface{}`. Cela signifie que **n'importe quel type** peut être utilisé.

go

 Copy code

```
func Identité[T any](value T) T {
    return value
}
```

- `T any` : Le paramètre de type `T` peut être de n'importe quel type.

2. Contraintes via des interfaces

Les **interfaces** sont le moyen le plus courant de définir des contraintes en Go. Tu peux contraindre le type à respecter certaines méthodes définies dans une interface.

go

 Copy code

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

func PrintString[T Stringer](value T) {
    fmt.Println(value.String())
}
```

- Ici, `T` doit implémenter l'interface `Stringer`, c'est-à-dire avoir une méthode `String()` `string`.

3. Contraintes sur plusieurs types (union de types)

Tu peux restreindre le type générique à plusieurs types spécifiques en utilisant une **union de types**. Cela permet de définir que `T` doit être l'un de ces types précis.

```
go                                Copy code

type Numeric interface {
    int | float64
}
```

```
func Add[T Numeric](a, b T) T {
    return a + b
}
```

- `T Numeric` : Le type `T` doit être soit `int` soit `float64` dans cet exemple. Tu peux ajouter plusieurs types dans une union.

4. Le mot-clé comparable

Le mot-clé `comparable` est une contrainte spéciale qui signifie que le type doit pouvoir être comparé à l'aide des opérateurs `==` et `!=`. Cela est utile pour des types qui peuvent être comparés comme les types primitifs (`int`, `string`, etc.).

go

 Copy code

```
func Compare[T comparable](a, b T) bool {
    return a == b
}
```

- `T comparable` : Le type `T` doit être un type qui peut être comparé avec `==` ou `!=` (par exemple, `int`, `string`, etc.).

5. Le tilde (~) pour les alias de types

Le tilde (~) est utilisé pour permettre à un **alias de type** d'être inclus dans une contrainte. Cela permet de restreindre les types aux types dérivés (alias).

go

Copy code

```
type MyInt int

type Numeric interface {
    ~int | ~float64
}

func Add[T Numeric](a, b T) T {
    return a + b
}
```

- `~int` : Le tilde permet à `MyInt` (qui est un alias de `int`) d'être accepté dans la contrainte `Numeric`.

Nouveau type et alias

Syntaxe pour créer un nouveau type :

```
go  
type MyInt int
```

Syntaxe pour créer un alias de type :

```
go  
type MyInt = int
```

Conversion de type

```
// Définition d'un nouveau type MyInt basé sur int
type MyInt int

func main() {
    var a MyInt = 42
    var b int = 10

    // Conversion explicite de MyInt vers int
    sum := int(a) + b
    fmt.Println(sum) // Résultat : 52
}
```

6. Contraintes multiples avec des méthodes spécifiques

Tu peux définir une interface qui combine plusieurs types de contraintes avec des méthodes spécifiques.

go

 Copy code

```
type Adder interface {
    Add(a, b int) int
    String() string
}

func AddAndPrint[T Adder](value T, a, b int) {
    fmt.Println(value.Add(a, b))
    fmt.Println(value.String())
}
```

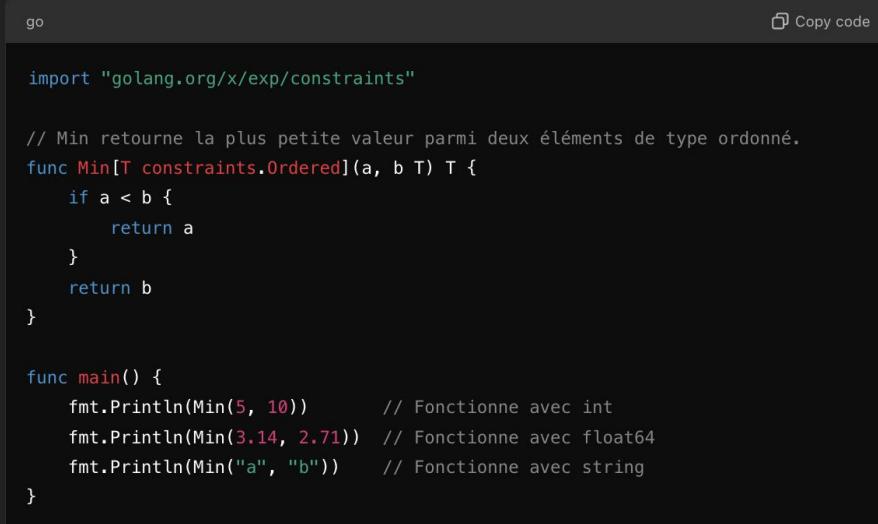
- `T Adder` : Le type `T` doit implémenter **toutes les méthodes** de l'interface `Adder`.

+ De contraintes avec le package golang.org/x/exp/constraints

1. `constraints.Ordered`

La contrainte `Ordered` est utilisée pour les types qui peuvent être ordonnés à l'aide des opérateurs de comparaison `<`, `<=`, `>`, `>=`. Cela inclut les types numériques (`int`, `float64`, etc.) ainsi que les chaînes de caractères (`string`).

Exemple d'utilisation de `constraints.Ordered` :



```
go                                Copy code

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

// Min retourne la plus petite valeur parmi deux éléments de type ordonné.
func Min[T constraints.Ordered](a, b T) T {
    if a < b {
        return a
    }
    return b
}

func main() {
    fmt.Println(Min(5, 10))      // Fonctionne avec int
    fmt.Println(Min(3.14, 2.71)) // Fonctionne avec float64
    fmt.Println(Min("a", "b"))   // Fonctionne avec string
}
```

- `T constraints.Ordered` : Le type `T` doit être un type qui peut être comparé avec les opérateurs d'ordre (`<`, `>`, etc.), comme les `int`, `float64`, ou `string`.

2. constraints.Signed

La contrainte `Signed` inclut tous les types numériques signés, comme `int`, `int32`, `int64`, etc.

Exemple d'utilisation de `constraints.Signed` :

```
go                                Copy code

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

// Absolute calcule la valeur absolue d'un entier signé.
func Absolute[T constraints.Signed](n T) T {
    if n < 0 {
        return -n
    }
    return n
}

func main() {
    fmt.Println(Absolute(-42))      // Fonctionne avec int
    fmt.Println(Absolute(int64(-5))) // Fonctionne avec int64
}
```

- `T constraints.Signed` : Le type `T` doit être un type signé comme `int`, `int32`, `int64`.

3. constraints.Unsigned

La contrainte `Unsigned` est utilisée pour tous les types numériques non signés, tels que `uint`, `uint32`, `uint64`, etc.

Exemple d'utilisation de `constraints.Unsigned` :

```
go                                ⌂ Copy code

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

// Increment ajoute 1 à une valeur non signée.
func Increment[T constraints.Unsigned](n T) T {
    return n + 1
}

func main() {
    fmt.Println(Increment(uint(10)))    // Fonctionne avec uint
    fmt.Println(Increment(uint64(15)))   // Fonctionne avec uint64
}
```

- `T constraints.Unsigned` : Le type `T` doit être un type non signé, comme `uint`, `uint32`, `uint64`.

4. `constraints.Integer`

La contrainte `Integer` inclut tous les types entiers, à la fois signés et non signés (`int`, `uint`, `int64`, `uint64`, etc.).

Exemple d'utilisation de `constraints.Integer` :

```
go                                Copy code

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

// Double double la valeur d'un entier, signé ou non signé.
func Double[T constraints.Integer](n T) T {
    return n * 2
}

func main() {
    fmt.Println(Double(10))          // Fonctionne avec int
    fmt.Println(Double(uint(10)))    // Fonctionne avec uint
}
```

- `T constraints.Integer` : Le type `T` doit être un entier signé ou non signé.

5. constraints.Float

La contrainte `Float` est utilisée pour tous les types de nombres à virgule flottante, tels que `float32` et `float64`.

Exemple d'utilisation de `constraints.Float` :

```
go                                ⚒ Copy code

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

// Multiply multiplie deux nombres flottants.
func Multiply[T constraints.Float](a, b T) T {
    return a * b
}

func main() {
    fmt.Println(Multiply(3.14, 2.71)) // Fonctionne avec float64
}
```

- `T constraints.Float` : Le type `T` doit être un type flottant comme `float32` ou `float64`.



L'écosystème

Awesome golang:
<https://github.com/avelino/awesome-go>

Librairies (toutes sont à utiliser)

- Utils Functions à la lodash: <https://github.com/samber/lo>
- Orm Postgres: <https://github.com/ent/ent>
- Data faker: <https://github.com/briancvoe/gofakeit>
- Ulid generator: <https://github.com/oklog/ulid>
- Multi error: <https://github.com/hashicorp/go-multierror>
- Live reload: <https://github.com/air-verse/air>
- Multi linter: <https://github.com/golangci/golangci-lint>
- Redis orm: <https://github.com/redis/rueidis>
- Testing: <https://github.com/stretchr/testify>
- Fast Json parser: <https://github.com/goccy/go-json>
- Backend framework: <https://github.com/gofiber/fiber>

Backend Tech complète

- PocketBase: <https://pocketbase.io/>
- Encore.dev: <https://encore.dev/>
- Hugo: <https://gohugo.io/>