Historique et objectifs du langage

Origines de Go

- Le langage Go, aussi appelé Golang, a été conçu chez Google en 2007 par Robert Griesemer, Rob Pike, et Ken Thompson.
- L'objectif initial était de résoudre des problèmes de productivité logicielle au sein de Google, en particulier pour des systèmes à grande échelle et des applications réseau.
- Go a été annoncé au public en novembre 2009.
- Go a été inspiré par les langages de programmation C, Pascal et Oberon, mais avec un accent particulier sur la simplicité, l'efficacité et la concurrence.
- Les informations détaillées sur l'origine de Go peuvent être trouvées dans l'article "The Go Programming Language" écrit par les créateurs du langage, disponible sur le blog officiel de Go.

Contributeurs clés

- Les principaux contributeurs à la création et au développement de Go sont Robert Griesemer, Rob Pike et Ken Thompson, tous employés de Google à l'époque.
- Rob Pike est l'un des auteurs du système d'exploitation Plan 9, et Ken Thompson est l'un des créateurs du langage C et du système d'exploitation Unix.
- L'équipe de Go a grandi au fil du temps, avec de nombreux contributeurs au sein de Google et de la communauté open source.
- Le code source de Go est hébergé sur GitHub, où la liste des contributeurs peut être consultée.

Objectifs de Go

- Simplicité : Go vise à être simple à lire et à écrire. Il dispose d'une syntaxe minimaliste qui facilite la lecture du code par les humains.
- Efficacité : Go a été conçu pour tirer parti de l'architecture multi-cœur des ordinateurs modernes. Son système de goroutines et de canaux permet de réaliser du code concurrent de manière naturelle.
- Interopérabilité : Go supporte l'interfaçage avec du code C, permettant ainsi d'utiliser des bibliothèques existantes.
- Sécurité : Go a un typage fort et le système de gestion de mémoire automatique aide à éviter certaines classes de bugs et de vulnérabilités.
- La documentation officielle de Go, disponible sur le site golang.org, fournit des informations plus détaillées sur les objectifs et la philosophie du langage.

Comparaison de Go avec d'autres langages populaires

Comparaison syntaxique

C vs Go

- C utilise des en-têtes pour la déclaration et l'importation de packages, tandis que Go utilise le mot clé `import`.
- Les fonctions en Go sont déclarées avec le mot clé `func`, tandis qu'en C, le type de retour est utilisé.
- Go dispose du ramasse-miettes, il n'est donc pas nécessaire de gérer manuellement la mémoire comme en C.
- C utilise des pointeurs pour la manipulation de structures, tandis que Go peut les manipuler directement.
- Exemple :

```c
// C code
#include <stdio.h>
void main() {
    printf("Hello, World!");
}
```

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

Python vs Go

- Python est un langage à typage dynamique tandis que Go est à typage statique.
- Go utilise des accolades pour délimiter les blocs de code, tandis que Python utilise l'indentation.
- Les listes et les dictionnaires Python sont représentés en Go par des slices et des maps respectivement.
- Exemple :

```python
# Python code
def greet(name):
    print(f"Hello, {name}!")
```

```go
// Go code
package main
import "fmt"
func greet(name string) {
    fmt.Printf("Hello, %s!", name)
}
```

Java vs Go

- Java utilise des classes et de l'héritage pour la réutilisation du code, tandis que Go utilise des interfaces et de l'encapsulation.
- Go utilise des routines pour la concurrence, tandis que Java utilise des threads.
- En Java, les exceptions sont utilisées pour la gestion des erreurs, tandis que Go utilise des valeurs de retour multiples.
- Exemple :

```java
// Java code
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

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

Comparaison des performances

C vs Go

- C est généralement plus rapide que Go en raison de son contrôle de bas niveau sur le matériel, mais Go est plus facile à optimiser en raison de ses fonctionnalités de haut niveau comme les goroutines.

Python vs Go

- Go est compilé en code machine, ce qui le rend beaucoup plus rapide que Python, qui est un langage interprété.
- Go a également une meilleure performance en matière de concurrence grâce à ses goroutines.

Java vs Go

- Go est généralement plus rapide que Java en termes de temps d'exécution et utilise moins de mémoire, bien que Java ait des performances proches grâce à la JVM et au JIT.

Comparaison des paradigmes de programmation

C vs Go

- C est un langage impératif procédural, tandis que Go est un langage de programmation concurrentiel qui combine les paradigmes impératifs et de programmation orientée objet.

Python vs Go

- Python supporte plusieurs paradigmes de programmation : impératif, orienté objet et fonctionnel. Go est principalement impératif avec des

éléments de programmation orientée objet.

Java vs Go

- Java est entièrement orienté objet, tandis que Go est impératif avec des éléments de programmation orientée objet, mais sans classes.

Comparaison des écosystèmes

C vs Go

- C a un écosystème plus mature et de nombreuses bibliothèques, mais Go a une meilleure gestion des dépendances et une bibliothèque standard plus complète.

Python vs Go

- Python a un écosystème très large avec une bibliothèque standard complète et de nombreux modules tiers. Go a une bibliothèque standard plus limitée, mais une gestion des dépendances robuste avec Go Modules.

Java vs Go

- Java a un écosystème très mature avec de nombreux cadres et bibliothèques. Go a une bibliothèque standard solide et une gestion des dépendances robuste, mais moins de cadres que Java.

Installation et configuration de l'environnement de développement

Installer Go sur différentes plateformes

Windows

- Télécharger l'installateur MSI de Go à partir de https://golang.org/dl/
- Exécuter l'installateur et suivre les instructions du guide d'installation.
- Vérifier l'installation en ouvrant un terminal et en tapant `go version`. Vous devriez voir la version de Go installée.

MacOS

- Utiliser Homebrew, un gestionnaire de paquets pour MacOS. Si Homebrew n'est pas installé, vous pouvez le faire en suivant les instructions sur https://brew.sh/.
- Une fois Homebrew installé, exécuter `brew install go`.
- Vérifier l'installation en ouvrant un terminal et en tapant `go version`.

Linux

- Télécharger le tarball de Go à partir de https://golang.org/dl/.
- Extraire le tarball dans /usr/local avec `tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz`.
- Ajouter /usr/local/go/bin au PATH en ajoutant `export PATH=$PATH:/usr/local/go/bin` dans le fichier de profil de l'utilisateur (~/.profile ou ~/.bashrc).
- Vérifier l'installation en ouvrant un terminal et en tapant `go version`.

Configuration des variables d'environnement

- `GOROOT` : C'est généralement le répertoire d'installation de Go. Sur la plupart des systèmes, il n'est pas nécessaire de définir cette variable car elle est automatiquement définie lors de l'installation.
- `GOPATH` : C'est le répertoire de travail où Go place le code source, les paquets compilés et les binaires. Si non défini, Go utilise par défaut le répertoire `~/go`.

- `GOBIN` : Le répertoire où `go install` place les binaires. Si non défini, il utilise `$GOPATH/bin`.
- Pour définir ces variables, ajouter les lignes suivantes à votre fichier de profil (~/.profile ou ~/.bashrc sur Linux, ~/.bash_profile sur MacOS) :

```
export GOPATH=$HOME/go
export GOBIN=$GOPATH/bin
export PATH=$PATH:$GOBIN
```

Installation d'un IDE ou d'un éditeur de texte adapté

Visual Studio Code (VS Code)

Un éditeur de code open source avec un support solide pour le développement Go via l'extension Go.

- Télécharger et installer VS Code à partir de https://code.visualstudio.com/download.
- Ouvrir VS Code et installer l'extension Go en recherchant "Go" dans le Marketplace.
- Pour configurer l'extension Go, ouvrir la palette de commandes (Ctrl+Shift+P) et choisir "Go: Install/Update Tools". Cocher toutes les cases et cliquer sur "OK".

GoLand

- Un IDE dédié à Go développé par JetBrains.
- Télécharger et installer GoLand à partir de https://www.jetbrains.com/go/download/.
- Ouvrir GoLand et suivre l'assistant de configuration initial pour configurer GoLand selon vos préférences.

Autres

D'autres options comprennent Vim, Emacs, Kate, Sublime Text, et Atom, chacun ayant des plugins ou des modes spécifiques pour Go

Exercices

Installer Go et configurer l'environnement de développement

- Commencez par vous rendre sur le site officiel de Go à l'adresse https://golang.org/dl/. Choisissez la version adaptée à votre système d'exploitation et votre architecture (Windows, macOS, Linux).
- Une fois le fichier téléchargé, lancez l'installation en suivant les instructions du programme d'installation. Sur un système Unix, vous pouvez utiliser la commande `tar` pour extraire le package téléchargé.
- Configurez la variable d'environnement `GOROOT` pour qu'elle pointe vers le répertoire où Go a été installé.
- Ajoutez le sous-répertoire `bin` de votre `GOROOT` à votre `PATH` pour que vous puissiez exécuter l'outil de commande Go (`go`) depuis n'importe quel répertoire.
- Choisissez un IDE (ex: Vim, VSCode, GoLand...)
- Assurez-vous que l'IDE est configuré pour utiliser le SDK Go installé, vérifiez la configuration dans les paramètres de l'IDE.

Créer un "Hello, World!" en Go, compiler, exécuter le programme

- Ouvrez votre IDE et créez un nouveau fichier `main.go` dans un répertoire de votre choix.
- Dans ce fichier, écrivez le code suivant :

```
package main

import "fmt"

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

* Pour compiler ce programme, ouvrez un terminal, naviguez vers le répertoire contenant votre fichier `main.go` et exécutez la commande `go build` . Cela générera un exécutable dans le même répertoire. * Pour exécuter le programme, tapez `./main` (sur Unix) ou `main.exe` (sur Windows) dans le terminal. Vous devriez voir le

message "Hello, World!" s'afficher. * Notez que vous pouvez également utiliser la commande `go run main.go` pour compiler et exécuter le programme en une seule étape.

Variables, constantes et déclarations

Déclaration de variables

- En Go, une variable est déclarée avec le mot-clé `var` suivi du nom de la variable et de son type. Exemple : `var myVariable int`.
- Une déclaration de variable sans valeur initiale assigne la valeur zéro du type de la variable. Par exemple, pour un type int, la valeur sera 0.

Variables à court terme (:=)

- Go offre un moyen plus court de déclarer et initialiser simultanément une variable avec `:=`. Cette syntaxe n'a besoin que du nom de la variable et de l'initialisation. Exemple : `myVariable := 10`.
- Ce raccourci ne peut être utilisé qu'à l'intérieur des fonctions.

Variables avec initialisation

- Il est possible de déclarer une variable et lui assigner une valeur initiale en même temps. Exemple : `var myVariable int = 10`.
- Lorsqu'une valeur initiale est fournie, Go peut déduire le type de la variable, permettant une déclaration abrégée. Exemple : `var myVariable = 10`.

Déclaration de constantes

- Une constante est une variable dont la valeur ne peut être modifiée une fois déclarée.
- En Go, une constante est déclarée avec le mot-clé `const` suivi du nom de la constante et de sa valeur. Exemple : `const Pi = 3.14`.

Les opérateurs de base

- Les opérateurs arithmétiques de base en Go sont : `+`, `-`, `*`, `/`, et `%`.
- Les opérateurs d'affectation sont `=`, `+=`, `-=`, `*=`, `/=`, et `%=`.
- Les opérateurs de comparaison sont `==`, `!=`, `<`, `>`, `<=`, et `>=`.
- Les opérateurs booléens sont `&&` (et), `||` (ou), et `!` (non).

- Les opérateurs bit à bit sont `&`, `|`, `^` (xor), `&^` (et non), `<<` (décalage à gauche), et `>>` (décalage à droite).

Exemples de code et commandes

- Pour créer un fichier Go, utilisez une commande de terminal telle que `touch myprogram.go`.
- Pour écrire dans ce fichier, vous pouvez utiliser un éditeur de texte ou un IDE de votre choix (comme Visual Studio Code, Sublime Text, etc.).
- Pour compiler et exécuter le fichier Go, utilisez la commande `go run myprogram.go`.
- Pour seulement compiler un fichier Go sans l'exécuter, utilisez `go build myprogram.go`.
- Vous pouvez trouver des informations supplémentaires et de l'aide dans la documentation officielle de Go à `https://golang.org/doc/` ou avec la commande `go doc` suivie du nom de la fonction, du paquet, etc., par exemple `go doc fmt.Println`.

Types de données de base

Entiers (int, uint, int8, uint8, etc.)

- Les entiers en Go peuvent être signés (`int8` , `int16` , `int32` , `int64` , `int`) ou non signés (`uint8` , `uint16` , `uint32` , `uint64` , `uint`).
- Le type `int` ou `uint` a une taille qui dépend de la plateforme (32 bits sur les systèmes 32 bits, 64 bits sur les systèmes 64 bits).
- Les types `int8` , `int16` , `int32` et `int64` représentent respectivement des entiers signés de 8, 16, 32 et 64 bits.
- De même, `uint8` , `uint16` , `uint32` et `uint64` représentent des entiers non signés de 8, 16, 32 et 64 bits.

Exemple de code:

```
var i int = 10
var j uint16 = 200
```

Flottants (float32, float64)

- Les nombres flottants en Go peuvent être `float32` ou `float64` , représentant respectivement des nombres à virgule flottante de 32 et 64 bits.
- Ils permettent de représenter les nombres réels avec une précision variable.

Exemple de code:

```
var x float32 = 3.14
var y float64 = 2.718281828
```

Booléens (bool)

- Le type `bool` représente une valeur booléenne qui peut être `true` ou `false` .
- Les opérations booléennes courantes (`&&` , `||` , `!`) sont supportées.

Exemple de code:

```
var isReady bool = false
```

Chaînes de caractères (string)

- Le type `string` en Go représente une séquence immuable de caractères.
- Les opérations courantes comme l'ajout (concaténation `+`), la longueur (`len()`) et l'accès à l'index (`[]`) sont supportées.

Exemple de code:

```
var greeting string = "Hello, World!"
```

Type rune

- Le type `rune` est un alias pour `int32` et représente un point de code Unicode.
- Il est utilisé pour représenter un caractère Unicode.

Exemple de code:

```
var letter rune = 'A'
```

Type byte

- Le type `byte` est un alias pour `uint8` et est généralement utilisé pour travailler avec des données binaires.

Exemple de code:

```
var b byte = 0xff
```

Conversion de types (type casting)

- Go ne permet pas de conversion de type automatique pour éviter les erreurs de conversion imprévues.
- Pour convertir une variable d'un type à un autre, on utilise une conversion de type explicite (`T(v)` convertit la valeur `v` au type `T`).

Exemple de code:

```
var i int = 42
var f float64 = float64(i)
```

Cette opération est sûre tant que la valeur peut être représentée dans le nouveau type.

Structures de contrôle (if, for, switch, defer)

Condition if-else

- En Go, la syntaxe de base pour une instruction if est : `if condition { // instructions } else { // instructions }`. Notons que contrairement à d'autres langages, il n'y a pas de parenthèses entourant la condition en Go.
- La condition dans l'instruction if doit être un booléen. Go est strictement typé et ne convertira pas automatiquement d'autres types en booléens.
- Go permet l'utilisation d'une instruction de pré-condition où une variable peut être définie dont la portée est limitée à la condition du bloc if. Exemple : `if x := computeSomething(); x < 10 { // instructions }`.
- Les informations détaillées peuvent être trouvées dans la documentation officielle du langage Go : https://golang.org/doc/effective_go#if

Exemple de code :

```
if x := 10; x%2 == 0 {
    fmt.Println(x, "is even")
} else {
    fmt.Println(x, "is odd")
}
```

Boucles for et variations (for-range)

- En Go, `for` est l'unique mot clé pour les boucles, qui peut être utilisé de plusieurs manières : `for init; condition; post { // instructions }`, `for condition { // instructions }`, `for { // instructions }` pour une boucle sans fin.
- Une autre variation du for est le `for range`, utilisé pour itérer sur des éléments dans une variété de structures de données comme les slices, arrays, strings, maps, ou channels.
- Pour plus de détails sur l'utilisation des boucles for en Go, vous pouvez consulter la documentation officielle : https://golang.org/doc/effective_go#control-structures

Exemple de code :

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

```
}

nums := []int{2, 3, 4}
for i, num := range nums {
    fmt.Println("index:", i, "value:", num)
}
```

Instruction switch-case

- En Go, l'instruction `switch` permet d'exécuter une branche de code parmi plusieurs possibles, basée sur l'évaluation d'une expression.
- Contrairement à d'autres langages, il n'est pas nécessaire d'utiliser `break` à la fin de chaque `case` en Go. Le contrôle de programme sort du `switch` dès qu'un `case` correspond.
- Les `case` en Go peuvent avoir plusieurs valeurs séparées par des virgules et peuvent également avoir des conditions, pas seulement des valeurs.
- Pour plus de détails sur l'utilisation de switch en Go, consultez la documentation officielle : https://golang.org/doc/effective_go#switch

Exemple de code :

```
switch day := 5; day {
case 1, 2, 3, 4, 5:
    fmt.Println("Weekday")
default:
    fmt.Println("Weekend")
}
```

Utilisation de defer

- En Go, `defer` est utilisé pour s'assurer qu'une fonction est appelée plus tard dans le programme, généralement à des fins de nettoyage. `defer` est souvent utilisé pour simplifier les fonctions qui effectuent diverses opérations de nettoyage.
- Les appels de fon

ctions différés sont exécutés dans le style LIFO (Last In First Out). * Pour plus d'informations sur l'utilisation de defer en Go, consultez la documentation officielle : https://golang.org/doc/effective_go#defer

Exemple de code :

```
func readFile(filename string) {
    file, err := os.Open(filename)
```

```
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    // read the file
}
```

Exercices

Calculer la somme et la moyenne d'une liste de nombres

1. **Création d'une liste de nombres**
   - Comprendre le concept d'un tableau (array) en Go: une collection ordonnée d'éléments du même type.
   - Exemple de déclaration et d'initialisation d'un tableau de nombres: `numbers := []int{10, 20, 30, 40, 50}`
   - Noter que les tableaux en Go sont de taille fixe. Pour une collection de taille variable, utiliser un slice.
2. **Calcul de la somme**
   - Utiliser une boucle `for` pour itérer à travers le tableau.
   - Exemple de boucle `for` pour le calcul de la somme :

```
sum := 0
for _, num := range numbers {
    sum += num
}
```

1. **Calcul de la moyenne**
   - La moyenne se calcule en divisant la somme par le nombre d'éléments dans le tableau.
   - Exemple de calcul de la moyenne : `average := float64(sum) / float64(len(numbers))`
2. **Affichage des résultats**
   - Utiliser la fonction `fmt.Println` pour afficher la somme et la moyenne.
   - Exemple : `fmt.Println("Sum:", sum, "Average:", average)`

Créer un programme pour convertir des températures entre Celsius et Fahrenheit

1. **Saisie de la température en Celsius**
   - Utiliser `fmt.Scanln` pour la saisie d'utilisateur.

- Exemple :

```
var celsius float64
fmt.Print("Enter temperature in Celsius: ")
fmt.Scanln(&celsius)
```

1. **Conversion en Fahrenheit**
   - La formule pour convertir de Celsius à Fahrenheit est `F = C * 9/5 + 32`.
   - Exemple : `fahrenheit := celsius*9/5 + 32`
2. **Saisie de la température en Fahrenheit**
   - De manière similaire à la saisie en Celsius, utiliser `fmt.Scanln`.
   - Exemple :

```
var fahrenheit float64
fmt.Print("Enter temperature in Fahrenheit: ")
fmt.Scanln(&fahrenheit)
```

1. **Conversion en Celsius**
   - La formule pour convertir de Fahrenheit à Celsius est `C = (F - 32) * 5/9`.
   - Exemple : `celsius := (fahrenheit - 32) * 5/9`
2. **Affichage des résultats**
   - Utiliser `fmt.Println` pour afficher les températures converties.
   - Exemple : `fmt.Println("Fahrenheit:", fahrenheit, "Celsius:", celsius)`

Déclaration et utilisation de fonctions

Syntaxe de déclaration de fonctions

- En Go, une fonction est déclarée avec le mot-clé "func", suivi du nom de la fonction, de la liste des paramètres, du type de retour et du corps de la fonction entre accolades.
- Exemple de code :

```
func nomDeLaFonction(param1 type1, param2 type2) typeRetour {
    // Corps de la fonction
}
```

- Les fonctions en Go peuvent être assignées à des variables ou passées en tant que paramètres à d'autres fonctions.
- Documentation officielle : Go Lang - Functions

Paramètres et arguments

- Les paramètres d'une fonction en Go sont définis entre parenthèses après le nom de la fonction.
- Un paramètre est défini par un nom et un type. Les paramètres de même type peuvent être regroupés.
- Les arguments sont les valeurs passées aux paramètres lors de l'appel d'une fonction.
- Exemple de code :

```
func add(x int, y int) int {
    return x + y
}
// Appel de la fonction
result := add(3, 5)
```

- Documentation officielle : Go Lang - Function declarations

Valeurs de retour et déclaration de plusieurs valeurs de retour

- En Go, une fonction peut retourner une ou plusieurs valeurs.

- Pour retourner plusieurs valeurs, les types de retour sont définis entre parenthèses.
- Cette fonctionnalité est souvent utilisée pour retourner à la fois le résultat et une éventuelle erreur.
- Exemple de code :

```
func divMod(x int, y int) (int, int) {
    return x / y, x % y
}
// Appel de la fonction
div, mod := divMod(7, 2)
```

- Documentation officielle : Go Lang - Function types

Fonctions anonymes et fermetures (closures)

- Une fonction anonyme est une fonction sans nom. Elle est souvent utilisée là où une fonction est nécessaire mais n'est pas assez complexe pour justifier une déclaration formelle.
- Une fermeture est une fonction anonyme qui capture et utilise des variables de l'environnement dans lequel elle est définie.
- Exemple de code :

```
adder := func(x int) int {
    return x + 1
}
// Appel de la fonction
result := adder(5)
```

- Documentation officielle : Go Lang - Function literals

Fonctions variadiques

- Les fonctions variadiques sont des fonctions qui peuvent prendre un nombre variable d'arguments.
- En Go, cela est indiqué par les points de suspension (...) avant le type du dernier paramètre.
- Exemple de code :

```
func sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
```

```
    }
    return total
}
// Appel de la fonction
result := sum(1, 2, 3, 4, 5)
```

- Documentation officielle : Go Lang - Calls

Méthodes et interfaces

Méthodes sur les types

- En Go, les méthodes peuvent être définies pour tout type défini par l'utilisateur, pas seulement pour les structures.
  - Exemple : `type MyInt int`
  - Déclaration de méthode : `func (mi MyInt) Double() int { return int(mi * 2) }`
- Les méthodes sont déclarées avec une réception spéciale qui contient une instance du type.
- Ces méthodes peuvent ensuite être appelées sur des instances de ce type.
  - Exemple : `var x MyInt = 5; fmt.Println(x.Double()) // Affiche 10`

Interfaces : définition et utilisation

- Une interface est une collection de méthodes.
  - Exemple : `type Geometry interface { Area() float64; Perimeter() float64 }`
- Les types qui définissent toutes les méthodes de l'interface satisfont automatiquement l'interface, sans avoir besoin de déclarer explicitement qu'ils la mettent en œuvre.
- Les interfaces permettent de créer du code généralisé qui peut fonctionner avec plusieurs types différents.
  - Exemple : Une fonction `Measure(g Geometry)` peut calculer l'aire et le périmètre de toute géométrie, que ce soit un cercle, un rectangle, etc.

Composition d'interfaces

- Les interfaces peuvent être composées d'autres interfaces, ce qui signifie qu'une interface peut contenir toutes les méthodes d'une ou de plusieurs autres interfaces.
  - Exemple : `type ReadWrite interface { Reader; Writer }`
- Cette caractéristique offre une grande flexibilité pour créer des interfaces complexes tout en maintenant la simplicité et la lisibilité.

Méthodes avec des pointeurs et des récepteurs de valeur

- Les méthodes en Go peuvent être déclarées soit avec un récepteur de valeur, soit avec un récepteur de pointeur.
- Les récepteurs de valeur reçoivent une copie de la valeur, tandis que les récepteurs de pointeur reçoivent une référence à la valeur.
  - Exemple : `func (v Value) ChangeVal() { v.x = 5 } // ne modifie pas la valeur originale`
  - Exemple : `func (p *Pointer) ChangePtr() { p.x = 5 } // modifie la valeur originale`
- Utiliser un récepteur de pointeur lorsque vous devez modifier l'état de l'objet ou si l'objet lui-même est volumineux et coûteux à copier.

Interfaces vides et assertions de type

- L'interface vide `interface{}` peut contenir n'importe quel type, ce qui la rend utile pour le stockage général.
- L'assertion de type est utilisée pour extraire la valeur sous-jacente de l'interface.
  - Exemple : `var i interface{} = "hello"; s := i.(string); fmt.Println(s) // Affiche "hello"`
- Si l'assertion de type échoue, elle renvoie une erreur. Pour éviter cela, vous pouvez utiliser une assertion de type avec une vérification d'erreur.
  - Exemple :
    `s, ok := i.(string); if ok { fmt.Println(s) } else { fmt.Println("Not a string") }`

Gestion des erreurs et conventions

L'interface "error"

- L'interface `error` est un type intégré à Go qui définit une méthode `Error() string`. Tout objet qui implémente cette méthode est considéré comme une erreur.
- Les erreurs peuvent être renvoyées comme une valeur de retour dans les fonctions et les méthodes.
- Exemple de création d'une erreur simple : `err := errors.New("une erreur est survenue")`

Création et propagation d'erreurs personnalisées

- Les erreurs personnalisées peuvent être créées en définissant un nouveau type qui implémente l'interface `error`.
- Pour propager une erreur, renvoyez-la de la fonction actuelle à la fonction appelante.
- Exemple de création et de propagation d'une erreur personnalisée :

```
type myError struct {
    msg string
}

func (e *myError) Error() string {
    return e.msg
}

func foo() error {
    return &myError{"une erreur est survenue"}
}
```

Utilisation de panic et recover

- `panic` est une fonction intégrée qui arrête l'exécution normale du programme. `panic` prend un seul argument de n'importe quel type.
- `recover` est une autre fonction intégrée qui regagne le contrôle après un `panic`.

- Les paniques sont généralement utilisées pour les erreurs irrécupérables, tandis que les erreurs classiques sont utilisées pour les erreurs récupérables.
- Exemple d'utilisation de `panic` et `recover` :

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

Utilisation du package "errors" et "fmt.Errorf"

- Le package `errors` fournit des fonctions pour manipuler les erreurs.
- `fmt.Errorf` permet de formatter une chaîne de caractères et de créer une erreur simultanément.
- Exemple d'utilisation de `errors.New` et `fmt.Errorf` :

```
err1 := errors.New("une erreur est survenue")
err2 := fmt.Errorf("une erreur est survenue: %v", err1)
```

Techniques de gestion d'erreurs courantes

- Vérifiez toujours les erreurs en utilisant une déclaration `if err != nil` . Ne présumez jamais qu'une opération réussira.
- En cas d'erreur, il est courant de la renvoyer à la fonction appelante.
- Si l'erreur ne peut pas être gérée, `log.Fatal(err)` ou `panic(err)` peuvent être utilisés pour arrêter le programme.
- Exemple de gestion d'erreur :

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

Exercices

Implémenter une calculatrice avec des opérations personnalisées

- **Définir les fonctions de base** : Créez quatre fonctions séparées pour chaque opération de base : `add`, `subtract`, `multiply`, `divide`. Chacune doit prendre deux arguments de type `float64` et retourner un résultat de type `float64`.

  Exemple de code pour la fonction `add` :

  ```
  func add(x float64, y float64) float64 {
     return x + y
  }
  ```

  * **Extension avec des opérations personnalisées** : Créez des fonctions supplémentaires pour d'autres opérations, comme le calcul du pourcentage, le carré, la racine carrée etc. Le même format de fonction peut être utilisé, avec des modifications selon les besoins.

- **Utiliser des interfaces pour gérer différentes opérations** : Définir une interface `Calculator` qui définit une méthode `Calculate` qui prend deux `float64` et retourne un `float64`. Cela permettra à différentes structures implémentant cette interface de fournir leur propre version de la méthode `Calculate`.

  Exemple de code pour l'interface `Calculator` :

  ```
  type Calculator interface {
     Calculate(x float64, y float64) float64
  }
  ```

- **Tests et validation de la calculatrice** : Utiliser le package de test intégré de Go (`testing`) pour écrire des tests unitaires pour chaque fonction de calcul. Pour exécuter les tests, utilisez la commande `go test` .

  Exemple de code pour un test de la fonction `add` :

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

Créer une bibliothèque pour lire et écrire des fichiers CSV avec une gestion d'erreurs robuste

- **Lire des fichiers CSV avec le package "encoding/csv"** : Utilisez la fonction `csv.NewReader` pour créer un nouvel objet Reader, qui peut être utilisé pour lire les données à partir d'un fichier CSV.

  Exemple de code pour lire un fichier CSV :

```
file, err := os.Open("data.csv")
if err != nil {
    log.Fatal(err)
}
reader := csv.NewReader(file)
records, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}
```

- **Gestion des erreurs lors de la lecture des fichiers CSV** : Lorsque vous lisez un fichier CSV, vérifiez toujours les erreurs retournées par les fonctions `os.Open` et `reader.ReadAll` . Si une erreur est détectée, utilisez `log.Fatal` pour la signaler et arrêter le programme.

- **Écrire des fichiers CSV avec le package "encoding/csv"** : Utilisez la fonction `csv.NewWriter` pour créer un nouvel objet Writer, qui peut être utilisé pour écrire des données dans un fichier CSV.

  Exemple de code pour écrire dans un fichier CSV : ``` file, err := os.Create("output.csv") if err != nil { log.Fatal(err) } writer :=

csv.NewWriter(file) writer.Write([]string{"Field1", "Field2", "Field3"}) writer.Flush() if err := writer.Error(); err != nil { log

.Fatal(err) } ```

- **Gestion des erreurs lors de l'écriture des fichiers CSV** : Après avoir écrit les données dans un fichier CSV avec `writer.Write` et `writer.Flush`, vérifiez toujours l'erreur retournée par `writer.Error`. Si une erreur est détectée, utilisez `log.Fatal` pour la signaler et arrêter le programme.

Pointeurs, allocation de mémoire, structures et types composites

Introduction aux pointeurs

- Les pointeurs sont des variables qui stockent l'adresse mémoire d'une autre variable.
- Leur utilisation permet une gestion efficace et flexible de la mémoire et facilite l'accès à des données spécifiques.

Opérateurs de pointeur

- L'opérateur `&` est utilisé pour obtenir l'adresse d'une variable.
  - Exemple : `var myVar int = 10; var p *int = &myVar` - ici, `p` est un pointeur vers `myVar`.
- L'opérateur `*` est utilisé pour déréférencer un pointeur, c'est-à-dire accéder à la valeur stockée à l'adresse pointée.
  - Exemple : `fmt.Println(*p)` - ceci affiche la valeur de `myVar`.

Allocation dynamique de mémoire avec `make` et `new`

- `new` et `make` sont deux fonctions intégrées qui allouent de la mémoire.
- `new(T)` alloue zéro valeur de type T, retourne un pointeur vers elle et ne nécessite pas d'être initialisé.
  - Exemple : `p := new(int)` - ceci crée un pointeur vers un int, initialisé à 0.
- `make(T)` alloue et initialise un slice, une map ou un channel de type T, retourne une valeur initialisée (et non un pointeur) et est nécessaire pour l'utilisation de ces types de données.
  - Exemple : `s := make([]int, 10)` - ceci crée un slice d'ints de longueur 10, tous initialisés à 0.

Utilisation des pointeurs dans les fonctions

- Les pointeurs peuvent être passés à des fonctions pour modifier directement la valeur des arguments dans la fonction appelante.

- Cela permet des modifications "en place" et peut améliorer l'efficacité pour le passage de grands ensembles de données.
    - Exemple :

```
func increment(x *int) {
    *x++
}
var y int = 1
increment(&y)
fmt.Println(y) // y vaut maintenant 2
```

Passage par référence et passage par valeur

- Go est principalement un langage "pass-by-value", ce qui signifie qu'il passe des copies des valeurs en tant qu'arguments aux fonctions.
    - Exemple :

```
func increment(x int) {
    x++
}
var y int = 1
increment(y)
fmt.Println(y) // y est toujours 1
```

- Cependant, avec l'utilisation de pointeurs, Go peut réaliser un "passage par référence", où l'adresse de la valeur est passée, permettant des modifications directes.
    - Voir l'exemple précédent dans la section "Utilisation des pointeurs dans les fonctions".

Structures et méthodes

Définition des structures

- Une structure est un type composite en Go qui permet de regrouper des valeurs de différents types.
- Les structures sont définies en utilisant le mot-clé `struct`.
- Syntaxe pour définir une structure :

```
type NomStructure struct {
    champ1 type1
    champ2 type2
    // ...
}
```

- Les noms des champs commencent par une majuscule s'ils doivent être visibles en dehors du package. Sinon, ils commencent par une minuscule.

Création et initialisation des instances de structures

- Les instances de structures peuvent être créées en utilisant le nom de la structure comme un type.
- Syntaxe pour l'initialisation d'une structure :

```
var s NomStructure // s est maintenant une instance de NomStructure avec des valeurs zéro
```

- Initialisation avec des valeurs spécifiques :

```
s := NomStructure{valeur1, valeur2}
```

- Les champs spécifiques peuvent être initialisés en utilisant le nom du champ :

```
s := NomStructure{champ1: valeur1, champ2: valeur2}
```

Méthodes associées aux structures

- Les méthodes sont des fonctions associées à des types spécifiques (structures).

- Syntaxe pour définir une méthode :

```
func (r NomStructure) NomMethode(param1 type1, param2 type2) returnType {
    // ...
}
```

- `r` est appelé récepteur de la méthode. Il est similaire au `this` ou `self` dans d'autres langages de programmation.

Méthodes avec récepteurs de pointeurs

- Les méthodes peuvent avoir des récepteurs de pointeurs, ce qui signifie qu'elles peuvent modifier l'instance de la structure.
- Syntaxe pour une méthode avec un récepteur de pointeur :

```
func (r *NomStructure) NomMethode(param1 type1, param2 type2) returnType {
    // ...
}
```

- Si une méthode a un récepteur de pointeur, l'instance de la structure est accessible et modifiable directement, pas seulement sa copie.

Composition et héritage

- Go n'a pas d'héritage classique, mais utilise la composition à la place.
- Une structure peut "hériter" d'une autre en l'incorporant comme un champ anonyme.
- Syntaxe pour la composition :

```
type NomStructure2 struct {
    NomStructure // hérite de tous les champs et méthodes de NomStructure
    // ...
}
```

Interfaces et polymorphisme

- Une interface est définie par un ensemble de méthodes. Tout type qui implémente ces méthodes satisfait l'interface.
- Syntaxe pour définir une interface :

```
type NomInterface interface {
    NomMethode1(param1 type1, param2 type2) returnType
    NomMethode2(param1 type1, param2 type2) returnType
    // ...
}
```

- Les structures qui implémentent toutes les méthodes d'une interface sont automatiquement considérées comme satisfaisant l'interface.
- Les interfaces permettent un polymorphisme de type en Go, où une fonction peut accepter un paramètre d'interface et se comporter différemment selon le type réel passé. *

Exemple de polymorphisme :

```
func faireQuelqueChose(i NomInterface) {
    i.NomMethode1()
    // ...
}
```

Tableaux, slices et maps

Tableaux et initialisation

- Un tableau en Go est une collection ordonnée d'éléments de même type, dont la taille est fixée à la déclaration.
- On déclare un tableau avec la syntaxe : `var arrayName [size]Type`, où `size` est un entier constant, et `Type` est le type d'éléments du tableau.
- Par exemple, `var arr [5]int` déclare un tableau de 5 entiers. Par défaut, tous les éléments sont initialisés à zéro.
- Les éléments d'un tableau peuvent être initialisés lors de la déclaration : `arr := [5]int{1, 2, 3, 4, 5}`.
- Pour accéder à un élément d'un tableau, on utilise l'indexation : `arr[0]` pour le premier élément.

Slices : création, manipulation et capacité

- Les slices en Go sont des segments dynamiques d'un tableau.
- Contrairement aux tableaux, les slices n'ont pas de taille fixe à la déclaration.
- Pour créer une slice, on peut utiliser la fonction built-in `make()` : `s := make([]int, 5)`.
- On peut également créer une slice à partir d'un tableau existant : `s := arr[0:3]`.
- Les éléments d'une slice peuvent être accédés de la même manière que pour un tableau : `s[0]`.
- Pour connaître la taille d'une slice, on utilise la fonction `len()`. Pour la capacité (c'est-à-dire la taille maximum que la slice peut atteindre sans ré-allocation), on utilise `cap()`.
- Pour ajouter des éléments à une slice, on utilise la fonction `append()` : `s = append(s, 6, 7, 8)`.
- Il faut noter que `append()` peut provoquer une ré-allocation si la capacité de la slice est dépassée.

Maps : création, manipulation et itération

- Les maps en Go sont des collections non ordonnées de paires clé-valeur.

- Pour déclarer une map, on utilise la fonction `make()` : `m := make(map[string]int)`.
- On peut ajouter des paires clé-valeur à une map avec la syntaxe suivante : `m["one"] = 1`.
- Pour accéder à une valeur, on utilise la clé : `value := m["one"]`.
- Pour supprimer une clé et sa valeur associée, on utilise la fonction `delete()` : `delete(m, "one")`.
- Pour vérifier si une clé est présente dans une map, on peut utiliser la forme à deux valeurs de l'accès aux éléments de la map : `value, ok := m["two"]`. Si la clé est présente, `ok` vaut `true` et `value` contient la valeur associée à la clé. Sinon, `ok` vaut `false` et `value` contient la valeur zéro du type de la map.
- Pour itérer sur une map, on peut utiliser la boucle `range` : `for key, value := range m { ... }`.

Exercices

Créer une bibliothèque pour gérer des formes géométriques et calculer leurs aires

- Définition des structures de formes géométriques * Identifier les propriétés communes à toutes les formes (par exemple, l'aire) * Définir une structure pour chaque type de forme (par exemple, cercle, rectangle, triangle) * Par exemple, `type Circle struct { Radius float64 }`

- Implémentation des méthodes de calcul d'aire * Définir une interface `Shape` avec une méthode `Area() float64` * Implémenter la méthode `Area()` pour chaque structure de forme * Pour un cercle, `func (c Circle) Area() float64 { return math.Pi * c.Radius * c.Radius }` * Pour un rectangle, `func (r Rectangle) Area() float64 { return r.Width * r.Height }`

- Utilisation de la bibliothèque dans un programme * Importer le package de formes géométriques dans le programme * Créer des instances de différentes formes * Appeler la méthode `Area()` sur ces instances * Exemple : `c := Circle{Radius: 5}; fmt.Println(c.Area())`

Créer un programme de gestion d'inventaire

- Modélisation des structures de données pour l'inventaire * Définir une structure `Item` avec les propriétés appropriées (par exemple, nom, description, quantité) * Créer une structure `Inventory` qui contient un tableau de `Item` * Par exemple, `type Item struct { Name string; Description string; Quantity int }`

- Fonctionnalités d'ajout, de suppression et de modification d'articles * Ajouter des méthodes à la structure `Inventory` pour gérer les articles * `AddItem(item Item)` , `RemoveItem(itemName string)` , `UpdateItem(itemName string, newItem Item)` * Ces méthodes doivent modifier le tableau de `Item` dans l' `Inventory`

- Recherche et filtrage des articles * Ajouter une méthode `FindItem(itemName string) *Item` à la structure `Inventory` * Cette méthode devrait parcourir le tableau d' `Item` et retourner le premier `Item` qui correspond au `itemName` donné

- Sauvegarde et chargement de l'inventaire à partir d'un fichier * Utiliser le package `os` et `encoding/json` pour lire et écrire des fichiers JSON * Ajouter des

méthodes `LoadFromFile(filename string)` et `SaveToFile(filename string)` à la structure `Inventory` * Ces méthodes doivent utiliser `json.Marshal` et `json.Unmarshal` pour convertir les structures en JSON et vice versa

Goroutines et concurrence

Introduction à la concurrence et aux goroutines

- La concurrence en informatique fait référence à l'habileté d'un système à exécuter plusieurs tâches en apparente simultanéité.
- Go offre un modèle de concurrence basé sur les goroutines et les canaux.
- Les goroutines sont des fonctions qui peuvent être exécutées en parallèle avec d'autres.
- En Go, la création d'une goroutine est simple et peu coûteuse en ressources comparée à la création d'un thread.

Création et utilisation des goroutines

- Pour créer une goroutine, on utilise le mot-clé `go` avant l'appel d'une fonction. Exemple : `go maFonction()`
- Chaque goroutine a sa propre pile, qui grandit et rétrécit au besoin.
- Le scheduler de Go gère l'exécution des goroutines, le développeur n'a pas besoin de se soucier explicitement de leur planification.
- Cependant, le contrôle des goroutines reste délicat, surtout lorsqu'il s'agit de synchronisation.

Gestion des goroutines avec sync.WaitGroup

- Le package `sync` de Go fournit une structure `WaitGroup` pour attendre qu'un ensemble de goroutines ait terminé son exécution.
- `Add(int)` ajoute le nombre spécifié de goroutines à attendre.
- `Done()` est appelé par chaque goroutine lorsqu'elle a terminé son travail.
- `Wait()` bloque l'exécution jusqu'à ce que toutes les goroutines aient terminé.

Parallélisme vs Concurrence

- La concurrence est une propriété du code ; le parallélisme, une propriété de l'exécution.
- La concurrence permet de structurer le programme pour qu'il puisse fonctionner de manière indépendante.

- Le parallélisme est l'exécution simultanée de plusieurs tâches. En Go, cela est rendu possible par les goroutines sur des systèmes à multiprocesseurs.

Comparaison des goroutines avec les threads

- Les goroutines sont plus légères que les threads (en termes de mémoire et de coût de changement de contexte).
- La gestion des goroutines est plus simple grâce à des constructions de haut niveau, comme les canaux.
- Cependant, contrairement aux threads, les goroutines partagent la même adresse mémoire, ce qui peut conduire à des problèmes de concurrence si elles ne sont pas correctement synchronisées.

Techniques de programmation concurrente en Go

- Les goroutines sont souvent utilisées avec des canaux, qui permettent une communication sûre entre les goroutines.
- `select` permet de gérer plusieurs canaux simultanément.
- Les patterns de concurrence, comme le pattern "worker", peuvent être facilement implémentés en Go grâce aux goroutines et aux canaux.
- Le package `sync` fournit des primitives de bas niveau pour la synchronisation, comme `Mutex` pour la mutuelle exclusion, permettant de protéger les sections critiques du code.

Canaux et communication entre goroutines

Introduction aux canaux (channels)

- Les canaux (channels) sont des conduits utilisés pour le transfert de données entre différentes goroutines. Ils sont un élément central de la programmation concurrente avec Go.
- Pour consulter la documentation officielle sur les canaux, se référer à la documentation officielle Go.

Création et utilisation des canaux

- Pour créer un canal, on utilise la commande `make`. Par exemple : `ch := make(chan int)`.
- Pour envoyer des données à travers un canal, on utilise l'opérateur `<-`. Par exemple : `ch <- 5`.
- Pour recevoir des données d'un canal, on utilise aussi l'opérateur `<-`. Par exemple : `val := <-ch`.

Types de canaux: non-buffered, buffered, unidirectionnels et bidirectionnels

- Les canaux non-buffered (ou synchrones) sont des canaux où l'envoi et la réception de données sont bloquants. C'est-à-dire qu'une goroutine qui envoie des données à un canal non-buffered se bloque jusqu'à ce qu'une autre goroutine reçoive ces données. Ils sont créés avec `ch := make(chan int)`.
- Les canaux buffered (ou asynchrones) sont des canaux où l'envoi et la réception de données ne sont pas bloquants. Une goroutine peut envoyer des données à un canal buffered sans se bloquer, à condition que le canal ait suffisamment de place pour stocker ces données. Ils sont créés avec `ch := make(chan int, 3)` où 3 est la taille du buffer.
- Les canaux unidirectionnels sont des canaux qui peuvent soit uniquement envoyer des données (`chan<- int`), soit uniquement recevoir des données (`<- chan int`).

- Les canaux bidirectionnels peuvent envoyer et recevoir des données. C'est le type par défaut des canaux (`chan int`).

Communication entre goroutines via les canaux

- Les canaux permettent de synchroniser et de communiquer entre les goroutines.
- Par exemple, pour attendre qu'une goroutine finisse son travail avant de continuer, on peut utiliser un canal. Une fois que la goroutine a terminé son travail, elle envoie une valeur à travers le canal, ce qui débloque la goroutine qui attendait.

Canaux unidirectionnels et bidirectionnels

- Les canaux unidirectionnels sont utiles pour exprimer clairement l'intention du programme. Par exemple, si une fonction reçoit un `chan<- int` , on sait qu'elle ne fera qu'envoyer des données à ce canal.
- Les canaux bidirectionnels sont plus flexibles mais peuvent rendre le code plus difficile à comprendre.
- Il est important de noter que les canaux unidirectionnels et bidirectionnels ne sont pas compatibles. On ne peut pas passer un `chan int` à une fonction qui attend un `chan<- int` ou un `<-chan int` .

Gestion des canaux avec close et range

- Pour fermer un canal, on utilise la fonction `close` . Par exemple : `close(ch)` .
- Lorsqu'un canal est fermé, on ne peut plus y envoyer de données, mais on peut toujours en recevoir.
- Pour parcourir toutes les valeurs d'un canal jusqu'à ce qu'il soit fermé, on utilise

une boucle `for` avec `range` . Par exemple : `for val := range ch` . * Il est recommandé de toujours fermer les canaux pour éviter les fuites de mémoire.

Patterns de concurrence et select

Introduction au select

- Le mot-clé `select` est un mécanisme fondamental en Go pour gérer plusieurs communications simultanément.
- `select` permet de surveiller plusieurs canaux en même temps, et de bloquer jusqu'à ce qu'un de ces canaux soit prêt pour effectuer une opération d'envoi ou de réception.
- La syntaxe de base de `select` est similaire à celle d'un `switch` mais pour les canaux.

Exemple de code :

```
select {
case <-canal1:
    // Traiter le cas du canal 1
case canal2 <- valeur:
    // Envoyer une valeur sur le canal 2
default:
    // Cas par défaut si aucun autre cas n'est prêt
}
```

Introduction au pattern "fan-in/fan-out"

- Le pattern "fan-in/fan-out" est un modèle de conception couramment utilisé pour la gestion de la concurrence en Go.
- "Fan-out" signifie démarrer plusieurs goroutines pour gérer des tâches en parallèle et "fan-in" signifie combiner plusieurs résultats en un seul.
- Ces modèles peuvent être combinés avec `select` pour gérer efficacement les tâches parallèles.

Exemple de code :

```
func fanIn(input1, input2 <-chan string) <-chan string {
    output := make(chan string)
    go func() {
        for {
            select {
            case s := <-input1:
                output <- s
            case s := <-input2:
                output <- s
```

```
        }
      }
    }()
    return output
}
```

Utilisation de select pour gérer plusieurs canaux

- `select` est l'outil parfait pour gérer plusieurs communications sur différents canaux.
- Il peut gérer aussi bien les cas d'envoi que de réception sur plusieurs canaux, et bloquera jusqu'à ce qu'un des canaux soit prêt.

Exemple de code :

```
select {
case msg1 := <-c1:
    fmt.Println("received", msg1)
case msg2 := <-c2:
    fmt.Println("received", msg2)
}
```

Timeouts et tickers avec select

- Les timeouts et les tickers sont des fonctionnalités importantes pour gérer le temps en Go.
- Le package `time` en Go fournit des tickers pour effectuer des actions à intervalles réguliers et des timeouts pour limiter le temps d'attente d'une opération.

Exemple de code :

```
select {
case <-time.After(1 * time.Second):
    fmt.Println("timeout 1 second")
case res := <-result:
    fmt.Println("received", res)
}
```

Implémentation de patterns de concurrence avancés

- Au-delà du "fan-in/fan-out", il existe de nombreux patterns avancés en Go pour gérer des scénarios de concurrence plus complexes, comme le pipeline, le pool de workers, etc.

- Il est recommandé d'explorer ces patterns dans la documentation officielle de Go et dans des ressources telles que le livre "Concurrency in Go" de Katherine Cox-Buday.

Bonnes pratiques pour la programmation concurrente en Go

- Toujours fermer les canaux lorsqu'ils ne sont plus utilisés.
- Utiliser `select` avec un `default` case pour éviter les blocages inutiles.
- Préférer l'utilisation de canaux pour la communication entre goroutines plutôt que le partage de mé

moire. * Utiliser `sync` package pour les primitives de synchronisation de bas niveau comme les Mutex ou les WaitGroups. * Eviter les race conditions en utilisant les outils fournis par Go tels que le race detector (`go run -race your_go_file.go`).

Exercices

Calculer des factorielles en parallèle

- Introduction à la définition du calcul factorial, `n! = n*(n-1)*(n-2)*...*3*2*1`. Ceci sera notre cas d'étude pour l'introduction à la concurrence.

Implémentation séquentielle du calcul de factoriels

- Écrire une fonction séquentielle simple en Go pour calculer les factoriels. Utilisez une boucle `for` pour cela.

```
func Factorial(n uint64) uint64 {
  if n < 2 {
     return 1
  }
  var fact uint64 = 1
  for i := 2; i <= int(n); i++ {
     fact *= uint64(i)
  }
  return fact
}
```

Conversion de l'implémentation en version concurrente avec goroutines et canaux

- Introduire les goroutines avec `go func()`. Ce sont des fonctions qui peuvent s'exécuter en parallèle avec d'autres goroutines.
- Présentation des canaux pour la communication entre les goroutines. On utilise `chan` pour la déclaration.
- Modification de la fonction Factorial pour utiliser les goroutines et les canaux. Le calcul de chaque terme de la multiplication est maintenant une goroutine.

```
func ConcurrentFactorial(n uint64) uint64 {
  if n < 2 {
     return 1
  }
  facts := make(chan uint64, n)
  for i := 2; i <= int(n); i++ {
     go func(j int) {
        facts <- uint64(j)
     }(i)
  }
  var fact uint64 = 1
  for i := 2; i <= int(n); i++ {
     fact *= <-facts
```

```
  }
  return fact
}
```

Analyse des performances et optimisation

- Introduction de la bibliothèque `time` pour mesurer le temps d'exécution des fonctions.
- Comparaison de la performance de l'implémentation séquentielle avec celle de l'implémentation concurrente.
- Discussion sur le coût de création des goroutines et sur la manière d'optimiser le code pour de meilleures performances.

Développer un système de files d'attente (queue) pour traiter des tâches en parallèle

Conception d'un système de files d'attente

- Présentation de la structure `Queue` et de ses méthodes : `Enqueue`, `Dequeue`, `IsEmpty`.
- Exemple de code :

```
type Queue []Task

func (q *Queue) Enqueue(t Task) {
  *q = append(*q, t)
}

func (q *Queue) Dequeue() (Task, bool) {
  if len(*q) == 0 {
     return Task{}, false
  }
  t := (*q)[0]
  *q = (*q)[1:]
  return t, true
}

func (q *Queue) IsEmpty() bool {
  return len(*q) == 0
}
```

Implémentation du système de files d'attente avec goroutines et canaux

- Modification de la structure `Queue` pour utiliser des goroutines et des canaux.
- Discussion sur la synchronisation des goroutines avec `sync.WaitGroup`.

- Présentation des mécanismes de verrouillage avec `sync.Mutex` pour éviter les conditions

Création et utilisation de packages

Structure d'un package

- Un package en Go est une collection de fichiers source Go qui résident dans le même répertoire.
- Tous les fichiers d'un même package doivent déclarer le même nom de package en tête du fichier.
- Le nom du package est préférablement le même que le nom du répertoire contenant les fichiers source.
- La structure d'un package peut comprendre des fichiers pour les définitions de types, les fonctions, les variables, les constantes, etc.
- Chaque package peut avoir un fichier unique nommé `doc.go` où la documentation du package est écrite.

Déclaration d'un package

- Pour déclarer un package, la première ligne non-commentée d'un fichier source doit être `package <nom-du-package>`.
- Le `package main` est spécial en Go. C'est le point d'entrée du programme. Le compilateur Go cherche une fonction `main` dans le `package main` pour démarrer l'exécution du programme.
- Exemple de déclaration de package :

```
package monpackage
```

Exportation de fonctions et de variables

- Les noms de fonctions, de variables et de types commençant par une majuscule sont exportés du package. Ils sont accessibles en dehors du package.
- Les noms commençant par une minuscule sont privés au package. Ils ne peuvent pas être directement accessibles en dehors du package.
- Exemple d'exportation d'une fonction :

```go
package monpackage

// Cette fonction peut être appelée en dehors du package
func MaFonctionExportee() {
    // code de la fonction
}
```

Importation et utilisation de packages

- Pour utiliser un package, il faut l'importer à l'aide de la directive `import`.
- Une fois importé, les fonctions, les variables et les types exportés peuvent être utilisés en préfixant leur nom par le nom du package.
- Les packages standard de Go comme `fmt` ou `os` peuvent être importés directement. Pour les packages tiers, il faut utiliser le chemin complet.
- Pour les projets plus conséquents, Go propose la notion de modules pour gérer les dépendances à plus grande échelle.
- Exemple d'importation et d'utilisation de packages :

```go
package main

import (
    "fmt"
    "monpackage"
)

func main() {
    fmt.Println("Appel de la fonction exportée :")
    monpackage.MaFonctionExportee()
}
```

Dans le terminal, pour récupérer les packages tiers, vous pouvez utiliser la commande `go get` suivie de l'URL du dépôt du package. Par exemple : `go get github.com/pkg/errors`.

Packages, modules et gestion des dépendances

Introduction aux Go modules

- Les Go modules sont introduits depuis Go 1.11 pour la gestion des dépendances.
- Ils permettent d'organiser le code en collections de packages réutilisables.
- Les Go modules contiennent une suite de versions cohérentes d'un ensemble de packages.
- Le fichier `go.mod` est utilisé pour définir le module. Il contient le nom du module, les versions de Go requises, ainsi que les dépendances du module.

Création d'un module Go

- Pour créer un module Go, utilisez la commande `go mod init [nom_du_module]`.
- Cette commande génère un fichier `go.mod` avec le nom du module spécifié.
- Par exemple : `go mod init github.com/monprojet` crée un module dont le nom est `github.com/monprojet`.

```
go mod init github.com/monprojet
```

Importation de packages externes

- L'importation de packages externes est réalisée en ajoutant une instruction `import` dans le fichier Go.
- Pour utiliser un package externe, spécifiez le chemin complet du package dans l'instruction `import`.
- Le compilateur Go et la commande `go get` utiliseront automatiquement le fichier `go.mod` pour résoudre les dépendances.
- Par exemple : `import "github.com/gin-gonic/gin"` importe le package Gin, un framework web en Go.

```
import "github.com/gin-gonic/gin"
```

Mise à jour et gestion des versions de packages

- Pour ajouter ou mettre à jour les dépendances, utilisez la commande `go get`.
- Par exemple, `go get github.com/gin-gonic/gin@v1.6.3` obtient une version spécifique du package Gin.
- L'exécution de `go get` met également à jour le fichier `go.mod` et le fichier `go.sum` (qui contient les sommes de contrôle attendues pour les contenus de modules spécifiques).
- Pour supprimer les dépendances non utilisées, utilisez la commande `go mod tidy`.

```
go get github.com/gin-gonic/gin@v1.6.3
go mod tidy
```

Exercices

Créer un package personnalisé

1. Identification de l'usage spécifique : Commencez par définir précisément l'objectif et les fonctionnalités que votre package doit atteindre et offrir.
2. Création du package avec la structure appropriée :

   * Création d'un répertoire pour le package. La commande shell pourrait être `mkdir monPackage`. * Création d'un fichier `.go` pour le code source du package, par exemple `touch monPackage/monPackage.go`. * Le fichier doit commencer par la déclaration `package monPackage` pour indiquer qu'il est le code source du package "monPackage".
3. Exportation et utilisation des fonctions et variables du package :

   * Dans Go, une fonction ou une variable est exportée (c'est-à-dire accessible à l'extérieur du package) si son nom commence par une majuscule. * Par exemple, `func MaFonction() {...}` serait une fonction exportée. * Une utilisation typique pourrait être `monPackage.MaFonction()`.
4. Importation et utilisation du package dans un projet Go :

   * Pour utiliser le package dans un autre fichier Go, importez-le avec la commande `import "chemin/vers/monPackage"`. * Vous pouvez ensuite utiliser les fonctions et variables exportées avec la syntaxe `monPackage.MaFonction()`.

Utiliser un package externe dans un projet Go

1. Recherche et sélection d'un package externe :

   * Utilisez des plateformes comme "pkg.go.dev" ou "github.com" pour trouver des packages adaptés à vos besoins. * Évaluez leur qualité à travers la documentation, les évaluations, la fréquence des mises à jour, etc.
2. Création d'un module Go pour le projet :

   * Utilisez la commande `go mod init monModule` pour initialiser un nouveau module Go dans votre répertoire de projet.
3. Importation et utilisation du package externe :

   * Utilisez la commande `import "url/du/package"` pour importer le package dans votre code source. * Pour télécharger et installer les dépendances, utilisez la

commande `go get` . * Utilisez ensuite les fonctions et variables du package comme précédemment.

4. Gestion des versions et mise à jour du package :

   * Vous pouvez spécifier une version spécifique d'un package en l'ajoutant à la fin de l'URL du package lors de l'importation, par exemple `import "url/du/package@v1.2.3"` . * Pour mettre à jour le package à la dernière version, utilisez la commande `go get -u url/du/package` .

Tests unitaires et benchmarks

Introduction aux tests unitaires en Go

- Les tests unitaires visent à vérifier le bon fonctionnement de chaque partie (unité ou fonction) d'un programme.
- Go offre une prise en charge intégrée des tests unitaires via le package "testing".
- La convention en Go est de placer les tests unitaires dans le même package que le code testé et de nommer les fichiers de test en suivant le format `<filename>_test.go`.

Utilisation du package "testing"

- Le package "testing" fournit les outils nécessaires pour écrire des tests unitaires et des benchmarks.
- Un test unitaire en Go est une fonction qui suit le format `func TestXxx(*testing.T)`. "Xxx" doit commencer par une lettre majuscule.
- `t.Error` et `t.Fatal` sont utilisés pour signaler les erreurs. `t.Fatal` arrête l'exécution du test immédiatement.

Exemple de code :

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

Commande pour exécuter les tests :

```
go test
```

Techniques de test : table-driven tests, mock objects

- Les tests de type table-driven sont une approche courante en Go pour tester plusieurs scénarios à l'aide d'une seule fonction de test. Une table de cas de test est créée, puis itérée pour exécuter le test sur chaque cas.

- Les objets mock sont utilisés pour simuler le comportement de composants réels pendant les tests. Il existe diverses bibliothèques pour aider à cela, comme "testify/mock".

Exemple de code pour table-driven tests :

```
func TestAdd(t *testing.T) {
    testCases := []struct {
        a, b, sum int
    }{
        {2, 3, 5},
        {-2, -3, -5},
        {2, -3, -1},
    }

    for _, tc := range testCases {
        if output := Add(tc.a, tc.b); output != tc.sum {
            t.Errorf("Expected %d, got %d", tc.sum, output)
        }
    }
}
```

Création et exécution de benchmarks

- Les benchmarks aident à mesurer la performance de certaines parties de votre code.
- Un benchmark en Go est une fonction qui suit le format `func BenchmarkXxx(*testing.B)`.
- L'exécution d'un benchmark est effectuée à l'aide de la commande `go test -bench=.`.

Exemple de code pour un benchmark :

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

Commande pour exécuter le benchmark :

```
go test -bench=.
```

Profiling et optimisation

Introduction au profiling

- Le profiling est une technique qui permet d'observer le comportement d'un programme en exécution afin d'identifier les zones de code qui consomment le plus de ressources.
- Il fournit des informations détaillées sur l'utilisation de la CPU, la mémoire, les blocages et les routines.

Utilisation de l'outil "pprof" pour le profiling CPU, mémoire et blocage

- `pprof` est un outil de profiling inclus dans la standard library de Go. Il permet de visualiser le profil CPU, mémoire, blocage et goroutines de votre programme.
- Pour utiliser `pprof`, importez le package `net/http/pprof` dans votre code et démarrez un serveur HTTP pour l'exposition des données de profiling.
- Utilisez la commande `go run` pour exécuter votre programme. Par exemple : `go run mon_programme.go`.
- Pendant l'exécution du programme, accédez à `http://localhost:8080/debug/pprof/` dans votre navigateur pour voir les profils disponibles.
- Pour capturer un profil CPU, accédez à `http://localhost:8080/debug/pprof/profile`.
- Pour capturer un profil de mémoire, accédez à `http://localhost:8080/debug/pprof/heap`.
- Pour capturer un profil de blocage, accédez à `http://localhost:8080/debug/pprof/block`.

Interprétation des résultats de profiling

- Les résultats de profiling sont représentés sous forme de graphiques ou de tables qui montrent la quantité de temps CPU ou de mémoire consommée par chaque fonction.
- Un "graphique d'appels" montre comment les fonctions appellent d'autres fonctions et la quantité de ressources consommées par chaque appel.

- Un "graphique en flamme" montre la consommation de ressources sur une échelle de temps, ce qui permet d'identifier les goulots d'étranglement.

Identification des goulots d'étranglement

- Les goulots d'étranglement sont des parties du code qui limitent les performances globales du programme.
- Ils peuvent être causés par des algorithmes inefficaces, des structures de données inappropriées, des opérations d'E/S bloquantes, etc.
- Les résultats de profiling peuvent aider à identifier ces goulots d'étranglement en montrant où le programme passe le plus de temps ou consomme le plus de ressources.

Techniques d'optimisation, bonnes pratiques et astuces

- Évitez les allocations de mémoire inutiles. Préférez le pooling d'objets ou réutilisez les objets existants lorsque c'est possible.
- Préférez les boucles `for` aux récursions pour éviter la surcharge de la pile.
- Utilisez des structures de données appropriées pour vos besoins. Par exemple, utilisez une map pour la recherche rapide d'éléments, utilisez une liste liée pour les insertions et suppressions fréquentes, etc.
- Utilisez la concurrence lorsque c'est approprié. Go fournit des goroutines et des canaux pour faciliter la programmation concurrente.
- Préférez les opérations d'E/S non bloquantes pour éviter que votre programme ne soit bloqué en attendant les E/S.

Mesure des améliorations après optimisation

- Après avoir apporté des modifications pour optimiser votre code, utilisez à nouveau `pprof` pour capturer un nouveau profil et comparez-le avec le profil original.
- Les améliorations devraient se traduire par une réduction du temps CPU, de la mémoire ou du blocage dans les parties du code que vous avez optimisées.
- N'oubliez pas que l'optimisation prématurée est la racine de tous les maux en programmation. Il est préférable de d'abord écrire un code clair et correct, puis de l'optimiser si nécessaire en se basant sur les données de profiling.

Documentation et commentaires

Commentaires de code : bonnes pratiques et conventions

- Les commentaires en Go commencent par le symbole `//` pour une seule ligne, ou peuvent être encadrés par `/* */` pour plusieurs lignes.
- Il est recommandé de commencer chaque commentaire par le nom de l'élément qu'il documente.
- Les commentaires de code doivent être clairs, concis et pertinents. Ils doivent expliquer le "pourquoi" et non le "comment". Le code lui-même montre comment il fonctionne.
- Évitez les commentaires redondants ou inutiles qui n'apportent pas d'information supplémentaire.
- La documentation d'une fonction ou d'une méthode doit commencer par le nom de la fonction ou de la méthode.
- Lors de l'écriture de commentaires, il faut être conscient que le code peut être lu et utilisé par des personnes dans le monde entier. Il est donc important d'éviter les références culturelles ou locales spécifiques qui pourraient ne pas être comprises par tout le monde.

Utilisation de "godoc" pour générer la documentation

- `godoc` est l'outil officiel de documentation pour le langage Go.
- `godoc` extrait les commentaires du code source pour générer une documentation en HTML ou en texte brut.
- Pour générer la documentation locale, utilisez la commande `godoc -http=:6060` dans le terminal. Vous pouvez ensuite accéder à la documentation à l'adresse `http://localhost:6060`.
- `godoc` génère automatiquement une page d'index pour tous les packages documentés.
- Pour consulter la documentation d'un package spécifique, utilisez la commande `godoc package/subpackage`.

Documentation des packages, fonctions, types et méthodes

- Chaque package doit avoir un commentaire de package juste avant la déclaration `package`. Ce commentaire donne une description générale du package.
- Chaque fonction, type et méthode publique doit avoir un commentaire expliquant son but, son utilisation et, si nécessaire, son comportement en cas d'erreurs.
- Les noms de fonctions, de types et de méthodes doivent être en CamelCase et commencer par une majuscule pour être exportés et visibles en dehors du package.
- Les commentaires pour les méthodes doivent être placés directement au-dessus de la méthode et commencer par le nom de la méthode.

Intégration de la documentation dans un workflow de développement

- La documentation doit être considérée comme une partie essentielle du code et doit être maintenue à jour tout au long du cycle de vie du développement.
- La revue de code doit également inclure la revue des commentaires et de la documentation.
- L'outil `golint` peut être utilisé pour vérifier la qualité des commentaires. `golint` peut être installé avec `go get -u golang.org/x/lint/golint` et utilisé avec la commande `golint ./...` dans le répertoire du projet.
- Pour un workflow automatisé, intégrez `golint` ou `godoc` dans votre processus de CI/CD pour vérifier la qualité de la documentation à chaque commit.
- Les erreurs ou les avertissements de documentation doivent être traités avec le même sérieux que les erreurs de compilation.

Exercices

Écrire des tests unitaires pour une application de calculatrice

Définir les fonctions de l'application calculatrice

- L'application de calculatrice doit comporter au moins les quatre opérations de base : addition, soustraction, multiplication et division.
- Pour chaque opération, créer une fonction dans Go. Par exemple, une fonction `Add(a int, b int) int` pour l'addition.

Créer des tests unitaires pour les différentes opérations

- Pour chaque fonction définie précédemment, créer une fonction de test correspondante. Par exemple, pour tester la fonction d'addition, créer une fonction `TestAdd(t *testing.T)`.
- À l'intérieur de cette fonction de test, vérifier que la fonction `Add()` donne le résultat attendu. Par exemple :

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

Utiliser des table-driven tests pour les tests unitaires

- Au lieu de répéter le code de test pour différents cas, utiliser les tests table-driven de Go. C'est une méthode où vous définissez une table de cas de test, et vous exécutez le test pour chaque cas de la table.
- Par exemple :

```
func TestAdd(t *testing.T) {
    var tests = []struct {
        a, b, want int
    }{
        {2, 3, 5},
        {4, -2, 2},
        {-4, -6, -10},
        {0, 5, 5},
```

```
    }

    for _, tt := range tests {
        if got := Add(tt.a, tt.b); got != tt.want {
            t.Errorf("Add(%v, %v) = %v; want %v", tt.a, tt.b, got, tt.want)
        }
    }
}
```

Exécuter les tests et analyser les résultats

- Utiliser la commande `go test` pour exécuter les tests.
- L'outil de test Go affiche un résumé des tests réussis et échoués.
- Si des tests échouent, l'outil de test affiche également les valeurs attendues et obtenues, ce qui aide à identifier où le problème se pose.

Analyser et optimiser les performances d'un programme

Choisir un programme à optimiser

- Choisir un programme qui présente des problèmes de performance ou qui est susceptible de s'améliorer. Cela peut être une partie de l'application de calculatrice ou un autre programme Go que vous avez écrit.

Profiler le programme à l'aide de "pprof" pour identifier les goulots d'étranglement

- Importer le package `net/http/pprof` dans le programme.
- Démarrer un serveur web pour accéder aux profils de performance en utilisant `http.ListenAndServe("localhost:6060", nil)` .
- Accéder à `http://localhost:6060/debug/pprof/` pour voir les profils disponibles.
- Utiliser `go tool pprof` pour analyser les profils et identifier les goulots d'étranglement.

Appliquer des techniques d'optimisation pour améliorer les performances

- Les techniques d'optimisation dépendent du problème spécifique identifié. Cela peut

impliquer de réduire l'utilisation de la mémoire, d'optimiser l'utilisation du CPU, d'améliorer l'algorithmique, etc. * Par exemple, pour réduire l'allocation de mémoire, réutiliser les tampons au lieu de les créer à chaque fois.

Mesurer les améliorations à l'aide des benchmarks et du profiling

- Écrire des benchmarks pour les fonctions que vous optimisez en utilisant le package `testing` de Go.
- Exécuter les benchmarks en utilisant `go test -bench=.`.
- Comparer les résultats avant et après l'optimisation pour vérifier que les performances ont été améliorées.

Programmation orientée réseau et développement d'applications web

Concepts de base des protocoles TCP et UDP

- TCP (Transmission Control Protocol) : - Orienté connexion, nécessite une connexion établie entre les machines avant la transmission de données. - Fournit la livraison de paquets en ordre et sans erreur. - Peut entraîner des latences plus élevées en raison des accusés de réception et des retransmissions. - Utilisé pour des applications nécessitant une transmission de données fiable, comme les serveurs web (HTTP/HTTPS).
- UDP (User Datagram Protocol) : - Sans connexion, ne nécessite pas de connexion établie avant la transmission de données. - Ne garantit pas la livraison de paquets en ordre ni sans erreur. - Plus rapide et moins gourmand en ressources que TCP, mais moins fiable. - Adapté aux applications de diffusion en temps réel, comme le streaming vidéo ou la VoIP.

Création d'un client TCP en Go

- Utilisation du package `net` de la bibliothèque standard Go.
- Fonction `net.Dial` pour initier une connexion TCP.
- Exemple de code :

```
package main

import (
    "fmt"
    "net"
    "log"
)

func main() {
    conn, err := net.Dial("tcp", "localhost:8080")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    fmt.Fprintf(conn, "Bonjour depuis le client TCP!\n")
}
```

- Gestion des erreurs avec `log.Fatal` et contrôle de la fermeture de la connexion avec `defer`.

Création d'un serveur TCP en Go

- Utilisation du package `net`.
- Fonction `net.Listen` pour écouter sur un port spécifique.
- Traitement des clients avec `Accept` et gestion des requêtes dans des goroutines pour parallélisation.
- Exemple de code :

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "log"
)

func main() {
    ln, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatal(err)
    }

    for {
        conn, err := ln.Accept()
        if err != nil {
            log.Println(err)
            continue
        }
        go handleConnection(conn)
    }
}

func handleConnection(conn net.Conn) {
    message, _ := bufio.NewReader(conn).ReadString('\n')
    fmt.Print("Message reçu:", string(message))
    conn.Close()
}
```

- Utilisation de `bufio` pour lire les données reçues.

Création d'un client UDP en Go

- Utilisation du package `net`.
- Fonction `net.DialUDP` pour initier une connexion UDP.

- Exemple de code :

```
package main

import (
    "fmt"
    "net"
    "log"
)

func main() {
    addr, err := net.ResolveUDPAddr("udp", "localhost:8081")
    if err != nil {
        log.Fatal(err)
    }

    conn, err := net.DialUDP("udp", nil, addr)
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    fmt.Fprintf(conn, "Bonjour depuis le client UDP!\n")
}
```

- Utilisation de `net.ResolveUDPAddr` pour résoudre l'adresse.

Création d

'un serveur UDP en Go

- Utilisation du package `net`.
- Fonction `net.ListenUDP` pour écouter sur un port spécifique.
- Exemple de code :

```
package main

import (
    "fmt"
    "net"
    "log"
)

func main() {
    addr, err := net.ResolveUDPAddr("udp", ":8081")
    if err != nil {
        log.Fatal(err)
    }

    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
```

```
        log.Fatal(err)
    }
    defer conn.Close()

    buffer := make([]byte, 1024)
    n, _, _ := conn.ReadFromUDP(buffer)
    fmt.Println(string(buffer[:n]))
}
```

- Utilisation de `ReadFromUDP` pour lire les données reçues.

Gestion des erreurs et des déconnexions

- Gestion des erreurs avec le second retour des fonctions, par exemple `net.Dial` ou `net.Listen`.
- Si une erreur se produit, le programme s'arrête généralement ou passe à la prochaine itération.
- Pour la gestion des déconnexions, il est important de fermer les connexions après utilisation pour libérer des ressources, par exemple en utilisant `defer conn.Close()`.

Création d'un serveur et d'un client HTTP simples

Fonctionnement du protocole HTTP

- HTTP signifie HyperText Transfer Protocol.
- C'est un protocole de communication basé sur le modèle client-serveur.
- HTTP utilise des méthodes telles que GET, POST, PUT, DELETE pour spécifier l'action souhaitée sur la ressource.

Gestion des requêtes et réponses HTTP en Go

- Go propose le package `net/http` pour interagir avec le protocole HTTP.
- Les objets `Request` et `Response` du package `net/http` représentent respectivement les requêtes et réponses HTTP.
- Utiliser `http.NewRequest` pour créer une requête et `http.DefaultClient.Do` pour envoyer une requête.

Exemple de code:

```
req, err := http.NewRequest("GET", "http://example.com", nil)
resp, err := http.DefaultClient.Do(req)
```

Création d'un serveur HTTP simple avec le package net/http

- Utiliser `http.HandleFunc` pour enregistrer un gestionnaire de fonction et un chemin.
- Utiliser `http.ListenAndServe` pour démarrer un serveur HTTP.

Exemple de code:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Bonjour, monde")
})
log.Fatal(http.ListenAndServe(":8080", nil))
```

Gestion des routes et des méthodes HTTP

- Utiliser `http.ServeMux` pour créer un multiplexeur de requêtes.
- Ajouter des gestionnaires pour différents chemins et méthodes avec `mux.HandleFunc`.

Exemple de code:

```
mux := http.NewServeMux()
mux.HandleFunc("/articles", ArticlesHandler)
http.ListenAndServe(":8080", mux)
```

Utilisation de middlewares pour gérer l'authentification et les erreurs

- Les middlewares permettent de traiter les requêtes et les réponses à différents niveaux de la pile HTTP.
- Implémenter l'authentification en vérifiant les en-têtes de requête pour un jeton valide.
- Gérer les erreurs en écrivant un code d'état HTTP approprié dans la réponse.

Exemple de code:

```
func AuthMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        token := r.Header.Get("Authorization")
        if token != "Token_valid" {
            http.Error(w, "Accès non autorisé", http.StatusUnauthorized)
            return
        }
        next.ServeHTTP(w, r)
    })
}
```

Création d'un client HTTP

- Utiliser `http.Client` pour créer un client HTTP personnalisé.
- Le client HTTP peut être configuré avec un délai d'expiration, des cookies, etc.

Exemple de code:

```
client := &http.Client{
    Timeout: time.Second * 10,
```

```
}
resp, err := client.Get("http://example.com")
```

Commandes: Pour exécuter un programme Go, utilisez la commande:

```
go run <nom_du_fichier>.go
```

Pour compiler un programme Go, utilisez la commande:

```
go build <nom_du_fichier>.go
```

Note

- Le protocole HTTP est sans état; chaque requête est indépendante.
- Go permet un traitement

efficace des requêtes HTTP grâce à sa gestion de la concurrence et des goroutines. - Les middlewares peuvent être utilisés pour exécuter du code avant et après chaque requête, ce qui est utile pour l'authentification, la journalisation, la gestion des erreurs, etc. - Un client HTTP en Go peut être utilisé pour envoyer des requêtes à des serveurs HTTP. - L'outil de ligne de commande Go (go) est utilisé pour construire et gérer les programmes Go.

Utilisation de gorilla/mux pour créer une API RESTful

Introduction à l'architecture REST

- REST (Representational State Transfer) est un style architectural pour développer des services web.
- Il définit un ensemble de contraintes pour créer des services web qui sont légers, maintenables, et évolutifs.
- REST utilise les méthodes HTTP standard (GET, POST, PUT, DELETE, etc.) pour effectuer des opérations sur les ressources.
- Les principaux composants de l'architecture REST sont les ressources, les requêtes, les réponses et les représentations.
- Pour en savoir plus sur REST, consulter la RFC 7231.

Présentation de gorilla/mux

- gorilla/mux est un package Go qui implémente un routeur de requête HTTP et un dispatcher.
- Il permet d'associer des gestionnaires de requêtes à des combinaisons spécifiques de méthodes HTTP et d'URL.
- Il offre des fonctionnalités comme le routage par motifs d'URL, l'extraction de variables d'URL, la génération inverse d'URL, etc.
- Pour installer gorilla/mux, utilisez la commande `go get -u github.com/gorilla/mux`.
- Pour une documentation plus détaillée, consultez la page du projet sur GitHub.

Création d'une API RESTful avec gorilla/mux

- Pour créer une API RESTful avec gorilla/mux, définissez des routes associées à des gestionnaires de requêtes spécifiques.
- Chaque gestionnaire de requête doit implémenter la logique nécessaire pour traiter une opération REST sur une ressource.
- Pour définir une route, utilisez la méthode `HandleFunc()` de l'objet `*mux.Router` retourné par `mux.NewRouter()`.

- Pour démarrer le serveur, utilisez la fonction `http.ListenAndServe()`.
- Voici un exemple de code pour une API RESTful simple :

```
r := mux.NewRouter()
r.HandleFunc("/books/{title}", getBook).Methods("GET")
r.HandleFunc("/books/{title}", createBook).Methods("POST")
http.ListenAndServe(":8080", r)
```

Gestion des paramètres de requête et des variables d'URL

- gorilla/mux permet d'extraire des variables d'URL en utilisant la méthode `Vars()`.
- Vous pouvez également définir des contraintes sur les variables d'URL en utilisant des régularités.
- Pour obtenir les paramètres de requête, utilisez la méthode `Query()` de l'objet `*http.Request`.
- Voici un exemple de code :

```
func getBook(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  title := vars["title"]
  params := r.URL.Query()
  author := params.Get("author")
  // ... logic to handle the request ...
}
```

Validation des données et gestion des erreurs

- Les données de requête doivent être validées avant d'être utilisées.
- La validation peut inclure des contrôles de format, de taille, de cohérence, etc.
- Les erreurs doivent être correctement gérées et signalées au client.
- Le package `net/http` définit des constantes pour les codes de statut HTTP couramment utilisés.
- Voici un exemple de code :

```
func createBook(w http.ResponseWriter, r *http.Request) {
  // ... extract and validate data ...
  if invalid {
    http.Error(w, "Invalid request data", http.StatusBadRequest)
```

```
    return
  }
  // ... handle the request ...
}
```

Exercices

Créer un serveur de chat simple avec TCP

Conception de l'architecture du serveur de chat

- Identifier les composants clés du serveur de chat : client, serveur, protocole de communication.
- Définir les interactions client-serveur (connexion, déconnexion, envoi de messages).

Implémentation du serveur de chat en utilisant TCP

- Initialiser un serveur TCP en Go : `net.Listen("tcp", ":8080")`.
- Accepter les connexions entrantes avec une boucle infinie : `for { conn, err := listener.Accept() }`.
- Gérer chaque connexion dans une goroutine distincte pour permettre des communications simultanées.

Création d'un client de chat en utilisant TCP

- Utiliser `net.Dial("tcp", "localhost:8080")` pour se connecter au serveur.
- Écrire des messages au serveur via la connexion TCP : `fmt.Fprintf(conn, "Message\n")`.
- Lire les réponses du serveur : `bufio.NewReader(conn).ReadString('\n')`.

Gestion des erreurs et des déconnexions

- Gérer les erreurs de connexion et d'écriture/lecture avec `if err != nil { log.Fatal(err) }`.
- Détecter les déconnexions par des erreurs de lecture/écriture.
- Fermer les connexions après utilisation : `defer conn.Close()`.

Implémenter une API RESTful pour gérer une liste de tâches

Conception de l'API pour la gestion des tâches

- Définir les opérations CRUD : création (POST), lecture (GET), mise à jour (PUT), suppression (DELETE).
- Spécifier les routes pour chaque opération, par exemple `/tasks` pour la lecture et la création, `/tasks/{id}` pour la mise à jour et la suppression.

Mise en place de l'API RESTful avec gorilla/mux

- Initialiser un routeur avec `r := mux.NewRouter()`.
- Définir les routes et les manipulatrices correspondantes : `r.HandleFunc("/tasks", createTask).Methods("POST")`.
- Démarrer le serveur avec `http.ListenAndServe(":8080", r)`.

Implémentation des opérations CRUD (Create, Read, Update, Delete)

- Définir une structure pour les tâches : `type Task struct { ID string; Description string }`.
- Stocker les tâches dans une map ou une base de données.
- Implémenter les fonctions `createTask`, `readTasks`, `updateTask`, `deleteTask` pour interagir avec le stockage.

Gestion des erreurs et validation des données

- Utiliser `http.Error` pour renvoyer des erreurs HTTP.
- Valider les données entrantes avec les tags `json` sur les structures et `if` pour vérifier les conditions.
- Gérer les erreurs de base de données et les erreurs de formatage JSON.
- Renvoyer des réponses HTTP appropriées : `w.WriteHeader(http.StatusCreated)`, `json.NewEncoder(w).Encode(tasks)`.

Encodage et décodage JSON

Introduction au format JSON

- JSON (JavaScript Object Notation) est un format d'échange de données largement adopté.
- Il utilise une syntaxe de notation clé-valeur et est souvent utilisé pour transmettre des données sur le web.

Le package "encoding/json" de la bibliothèque standard

- Go fournit un package "encoding/json" pour le traitement des données JSON.
- Ce package comprend des fonctions pour la sérialisation (marshalling) et la désérialisation (unmarshalling) des structures Go en JSON et vice versa.
- Documentation officielle et détaillée du package : https://pkg.go.dev/encoding/json

Marshalling et Unmarshalling de données JSON

- Le marshalling est le processus de transformation de données Go en JSON. La fonction `json.Marshal()` est utilisée pour ce faire.
- L'unmarshalling est l'opération inverse : il transforme les données JSON en données Go. La fonction `json.Unmarshal()` est utilisée pour ce faire.
- Exemple de code :

```
package main
import (
  "encoding/json"
  "fmt"
)
type Person struct {
  Name string
  Age  int
}
func main() {
  p := Person{Name: "Alice", Age: 30}
  // Marshalling
  jsonData, err := json.Marshal(p)
  if err != nil {
    fmt.Println(err)
```

```
    return
  }
  fmt.Println(string(jsonData))
  // Unmarshalling
  var p2 Person
  err = json.Unmarshal(jsonData, &p2)
  if err != nil {
    fmt.Println(err)
    return
  }
  fmt.Println(p2)
}
```

Personnalisation de l'encodage et décodage avec des tags struct

- Les tags struct permettent de personnaliser l'encodage et le décodage JSON en Go.
- Ils permettent notamment de spécifier le nom de la clé JSON pour un champ donné ou d'ignorer un champ.
- Exemple de code :

```
type Person struct {
  Name string `json:"name"`
  Age  int    `json:"age"`
}
```

Dans cet exemple, le champ `Name` sera encodé en JSON sous le nom `name` et le champ `Age` sous le nom `age`.

Gérer les erreurs liées à l'encodage et décodage JSON

- Les erreurs lors du marshalling et unmarshalling peuvent être attrapées et gérées en Go.
- Exemple de gestion d'erreurs :

```
jsonData, err := json.Marshal(p)
if err != nil {
  fmt.Println(err)
  return
}
```

Dans cet exemple, si une erreur se produit lors de l'encodage, elle est imprimée et la fonction est interrompue.

Introduction à Google Protocol Buffers

Présentation et avantages de Google Protocol Buffers

- Google Protocol Buffers, aussi appelé protobuf, est un format binaire pour la sérialisation de données structurées.
- Protobuf permet une sérialisation efficace des données, avec une taille plus petite que le format JSON ou XML.
- Il permet également une compatibilité ascendante et descendante entre différentes versions de vos structures de données.
- Protobuf est langage-neutre et plateforme-neutre, vous pouvez donc l'utiliser pour échanger des données entre des systèmes écrits en différents langages de programmation.

Définition des fichiers .proto

- Les fichiers .proto sont utilisés pour définir la structure des données que vous voulez sérialiser avec protobuf.
- Un fichier .proto ressemble à ceci :

```
syntax = "proto3";

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
}
```

- La première ligne spécifie la version de la syntaxe protobuf utilisée (ici, "proto3").
- Ensuite, nous définissons une structure de message "Person" avec trois champs: "name", "id" et "email". Chaque champ a un type de données et un numéro unique.

Structure des messages et des services

- Les messages sont similaires aux structures ou aux objets dans d'autres langages de programmation. Ils contiennent des champs avec des types de données spécifiques.

- Les services peuvent être définis dans un fichier .proto pour spécifier des méthodes RPC avec leurs types de requête et de réponse.
- Exemple de service dans un fichier .proto :

```
service SearchService {
  rpc Search (SearchRequest) returns (SearchResponse);
}
```

Types de données et règles de compatibilité

- Protobuf prend en charge un ensemble de types de données scalaires comme int32, float, double, bool et string.
- Les types complexes tels que les énumérations (enum), les messages imbriqués et les répétitions de types de données sont également pris en charge.
- Protobuf garantit la compatibilité ascendante et descendante en respectant certaines règles. Par exemple, ne jamais changer le numéro de champ d'un champ existant, et les éléments marqués comme 'deprecated' peuvent être retirés après un certain temps.

Compilation des fichiers .proto

- Les fichiers .proto doivent être compilés en code source du langage cible en utilisant le compilateur protobuf (protoc).
- Commande pour compiler un fichier .proto en code Go :

```
protoc --go_out=. *.proto
```

- Cette commande génère un fichier .go à partir du fichier .proto, qui peut ensuite être importé et utilisé dans votre programme Go.

Utilisation de golang/protobuf

Installation et configuration de golang/protobuf

- Installez le package golang/protobuf avec la commande `go get -u google.golang.org/protobuf/cmd/protoc-gen-go@v1.26`. La version peut varier en fonction des mises à jour. Assurez-vous d'avoir la version la plus récente.
- Configurez le PATH pour inclure le répertoire des binaires de Go afin de pouvoir exécuter `protoc-gen-go` depuis n'importe quel emplacement.

Génération de code Go à partir de fichiers .proto

- Pour générer du code Go à partir d'un fichier .proto, utilisez la commande `protoc --go_out=. votre_fichier.proto`. Remplacez "votre_fichier.proto" par le nom de votre fichier .proto. Cette commande génère un fichier .go à partir du fichier .proto spécifié.

Encodage et décodage de messages Protobuf en Go

- Les messages Protobuf sont encodés et décodés en Go à l'aide des méthodes Marshal et Unmarshal.
  - Exemple d'encodage :

```
message := &votrePb.VotreMessage{...}
data, err := proto.Marshal(message)
```

  - Exemple de décodage :

```
message := &votrePb.VotreMessage{}
err := proto.Unmarshal(data, message)
```

  - Remplacez "votrePb" et "VotreMessage" par le nom de votre package et de votre message.

Implémentation de services RPC avec golang/protobuf

- La génération de code pour les services RPC nécessite l'installation d'un générateur de plugins supplémentaire. Utilisez la commande `go get -u google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.1`.
- Pour générer du code de service RPC à partir d'un fichier .proto, utilisez la commande `protoc --go-grpc_out=. votre_fichier.proto`.
- Implémentez les interfaces générées dans votre code Go pour créer un serveur RPC.

Gérer les erreurs et les performances avec golang/protobuf

- L'encodage et le décodage des messages, ainsi que la communication RPC, peuvent tous générer des erreurs. Assurez-vous de gérer ces erreurs de manière appropriée dans votre code.
- Pour les performances, évitez de répéter l'encodage et le décodage des mêmes messages. Stockez les messages fréquemment utilisés ou ceux qui sont coûteux à encoder ou à décoder.
- La bibliothèque golang/protobuf fournit des outils pour mesurer les performances, tels que protoc-gen-go-trace et protoc-gen-go-heapdump. Ces outils peuvent aider à identifier les goulots d'étranglement et à optimiser les performances.

Exercices

Créer une API RESTful qui renvoie des données JSON

Présentation du projet et des objectifs

- Le projet consiste à développer une API RESTful simple en Go qui gère les ressources d'un livre de recettes. Cette API devra créer, lire, mettre à jour et supprimer (CRUD) des recettes dans une structure de données en mémoire.
- Les recettes seront stockées sous forme de structures Go, qui seront converties en JSON lorsqu'elles seront renvoyées par l'API.
- L'objectif est de comprendre comment construire des services web en Go, comment travailler avec JSON et comment structurer un projet Go.

Conception de l'API RESTful avec gorilla/mux

- Utilisation du package gorilla/mux pour router les requêtes HTTP à leurs gestionnaires respectifs.
  - Exemple de code : `r := mux.NewRouter()`
  - Ajouter des routes : `r.HandleFunc("/recipes", getRecipes).Methods("GET")`
- Les requêtes HTTP GET, POST, PUT et DELETE seront mappées à des fonctions CRUD dans notre code.
- Les routes seront structurées comme `/recipes` pour l'ensemble des recettes et `/recipes/{id}` pour une recette spécifique.

Implémentation de l'encodage et décodage JSON

- Utilisation du package "encoding/json" de la bibliothèque standard de Go pour l'encodage et le décodage JSON.
- Fonction `json.NewEncoder(w).Encode(recipes)` pour encoder les recettes en JSON et les envoyer dans la réponse HTTP.
- Fonction `json.NewDecoder(r.Body).Decode(&recipe)` pour décoder le corps de la requête HTTP en une nouvelle recette.

Test et débogage de l'API RESTful

- Utilisation de l'outil curl ou de Postman pour tester l'API.

- Exemple de commande curl pour créer une nouvelle recette : `curl -X POST -H "Content-Type: application/json" -d '{"title":"Pizza Margherita", "ingredients":["Tomatoes", "Mozzarella", "Basil", "Oil", "Salt"]}' http://localhost:8000/recipes`

Retour d'expérience et améliorations

- Discussion sur la structuration du code Go pour des projets plus importants.
- Introduction à la documentation des API avec Swagger ou API Blueprint.
- Discussion sur l'utilisation des bases de données pour stocker les recettes de manière persistante.

Implémenter un client et un serveur qui communiquent avec des messages protobuf

Présentation du projet et des objectifs

- Le projet consiste à créer un service de chat simple avec un serveur et un client qui communiquent via Protocol Buffers (protobuf).
- L'objectif est de comprendre comment protobuf fonctionne, comment générer du code Go à partir de fichiers .proto, et comment implémenter des communications réseau en Go.

Conception des messages et services protobuf

- Utilisation de la syntaxe protobuf pour définir les messages de chat et le service de chat.
  - Exemple de code .proto :

```
syntax = "proto3";

message ChatMessage {
   string sender = 1;
   string content = 2;
}

service ChatService {
   rpc SendChat (ChatMessage) returns (ChatMessage) {}
}
```

- Compilation du fichier .proto en code Go à l'aide de la commande : `prot

oc --go_out=plugins=grpc:. chat.proto`

Génération de code Go à partir de fichiers .proto

- Explication sur le code généré, notamment le code pour les messages et le code pour le service.
- Utilisation de la bibliothèque gRPC de Google pour créer le client et le serveur.

Implémentation du client et du serveur en Go

- Création du serveur avec une fonction qui implémente le service de chat.
  - Exemple de code :

```
type server struct{}

func (s *server) SendChat(ctx context.Context, message *ChatMessage)
(*ChatMessage, error) {
    return message, nil
}
```

- Création du client qui envoie des messages au serveur.
  - Exemple de code :

```
conn, _ := grpc.Dial("localhost:9000", grpc.WithInsecure())
defer conn.Close()

c := NewChatServiceClient(conn)

message := &ChatMessage{Sender: "Client", Content: "Hello, server!"}
response, _ := c.SendChat(context.Background(), message)
fmt.Println("Received:", response.Content)
```

Test, débogage et optimisation des communications protobuf

- Utilisation de l'outil grpcurl pour tester le service de chat.
- Exemple de commande grpcurl : `grpcurl -d '{"sender":"Client","content":"Hello, server!"}' -plaintext localhost:9000 ChatService/SendChat`
- Discussion sur la manière d'optimiser les communications protobuf et sur l'utilisation des flux de messages (streaming).

Rappel: Introduction aux bases de données relationnelles

Définition et concepts fondamentaux

- Les bases de données relationnelles sont des systèmes de gestion de bases de données (SGBD) basés sur le modèle relationnel, inventé par E.F. Codd.
- Les données sont organisées en tables (ou relations) qui sont composées de lignées (ou tuples) et de colonnes (ou attributs).
- Une clé primaire unique identifie chaque ligne dans une table.
- Les relations entre les tables sont définies par des clés étrangères.

Modèle relationnel et langage SQL

- Le langage SQL (Structured Query Language) est utilisé pour interagir avec les bases de données relationnelles.
- SQL permet de créer, de lire, de mettre à jour et de supprimer des données (opérations CRUD).

Rappel: Concepts clés de SQL

DDL (Data Definition Language): CREATE, ALTER, DROP

- Le DDL définit et gère les objets de la base de données.
- CREATE est utilisé pour créer des objets dans la base de données.
- ALTER est utilisé pour modifier la structure des objets existants dans la base de données.
- DROP est utilisé pour supprimer des objets de la base de données.

Exemple de commandes SQL DDL :

```
CREATE TABLE Utilisateurs (
    ID int,
    Nom varchar(255),
    Email varchar(255),
```

```
    PRIMARY KEY (ID)
);
```

```
ALTER TABLE Utilisateurs
ADD DateNaissance date;
```

```
DROP TABLE Utilisateurs;
```

DML (Data Manipulation Language): SELECT, INSERT, UPDATE, DELETE

- Le DML concerne les opérations sur les données.
- SELECT est utilisé pour interroger les données dans la base de données.
- INSERT est utilisé pour insérer de nouvelles données dans la base de données.
- UPDATE est utilisé pour mettre à jour des données existantes dans la base de données.
- DELETE est utilisé pour supprimer des données de la base de données.

Exemple de commandes SQL DML :

```
SELECT * FROM Utilisateurs;
```

```
INSERT INTO Utilisateurs (ID, Nom, Email)
VALUES (1, 'Dupont', 'dupont@example.com');
```

```
UPDATE Utilisateurs
SET Email = 'dupont2@example.com'
WHERE ID = 1;
```

```
DELETE FROM Utilisateurs
WHERE ID = 1;
```

Clauses SQL : WHERE, ORDER BY, GROUP BY, HAVING, JOIN

- WHERE est utilisé pour filtrer les résultats d'une requête.
- ORDER BY est utilisé pour trier les résultats d'une requête.
- GROUP BY est utilisé pour regrouper les résultats d'une requête.
- HAVING est utilisé pour filtrer les résultats d'une requête GROUP BY.

- JOIN est utilisé pour combiner des rangées de deux ou plusieurs tables.

Exemple de commandes SQL avec clauses :

```sql
SELECT * FROM Utilisateurs WHERE Nom = 'Dupont';
```

```sql
SELECT * FROM Utilisateurs ORDER BY Nom;
```

```sql
SELECT COUNT(*), Pays FROM Utilisateurs GROUP BY Pays;
```

```sql
SELECT COUNT(*), Pays FROM Utilisateurs GROUP BY Pays HAVING COUNT(*) > 1;
```

```sql
SELECT Utilisateurs.Nom, Commandes.Numero
FROM Utilisateurs


JOIN Commandes ON Utilisateurs.ID = Commandes.UtilisateurID;
```

Rappel: Pratique de SQL

Environnement SQL et outils de travail

- Un environnement SQL typique comprend un SGBD (par exemple, PostgreSQL, MySQL), un shell pour exécuter les commandes SQL et un outil d'interface utilisateur (par exemple, pgAdmin pour PostgreSQL, phpMyAdmin pour MySQL).

Création et manipulation de tables

- Utilisez la commande CREATE pour créer une table.
- Utilisez la commande INSERT pour ajouter des données à une table.
- Utilisez la commande UPDATE pour modifier des données dans une table.
- Utilisez la commande DELETE pour supprimer des données d'une table.

Exemple de création et manipulation de tables :

```sql
CREATE TABLE Commandes (
    Numero int,
    UtilisateurID int,
    DateCommande date,
    PRIMARY KEY (Numero)
);
```

```sql
INSERT INTO Commandes (Numero, UtilisateurID, DateCommande)
VALUES (1, 1, '2023-06-20');
```

```sql
UPDATE Commandes
SET DateCommande = '2023-06-21'
WHERE Numero = 1;
```

```sql
DELETE FROM Commandes
WHERE Numero = 1;
```

Exécution de requêtes SQL

- Utilisez la commande SELECT pour interroger les données dans la base de données.
- Les clauses SQL (WHERE, ORDER BY, GROUP BY, HAVING, JOIN) peuvent être utilisées pour personnaliser les résultats de la requête.

Exemple d'exécution de requêtes SQL :

```sql
SELECT * FROM Commandes WHERE UtilisateurID = 1;
```

```sql
SELECT * FROM Commandes ORDER BY DateCommande;
```

```sql
SELECT COUNT(*), UtilisateurID FROM Commandes GROUP BY UtilisateurID;
```

```sql
SELECT COUNT(*), UtilisateurID FROM Commandes GROUP BY UtilisateurID HAVING
COUNT(*) > 1;
```

```sql
SELECT Utilisateurs.Nom, Commandes.Numero
FROM Utilisateurs
JOIN Commandes ON Utilisateurs.ID = Commandes.UtilisateurID;
```

Introduction à GORM

Présentation de GORM

- Qu'est-ce qu'un ORM (Object Relational Mapping) ?
  - Un ORM (Object-Relational Mapping) est un paradigme de programmation qui permet d'interagir avec la base de données en utilisant le langage de programmation orienté objet. Il traduit les tables de la base de données en objets et vice versa.

- Pourquoi utiliser GORM ?
  - GORM (Go Object-Relational Mapper) est un ORM écrit en Go. Il est utilisé pour simplifier la gestion des bases de données relationnelles en masquant les détails techniques sous-jacents du langage SQL.
  - GORM offre une interface simple pour la création, la lecture, la mise à jour et la suppression de données, tout en fournissant des fonctionnalités avancées telles que les requêtes complexes et les associations.

- Fonctionnalités clés de GORM
  - GORM supporte toutes les fonctionnalités principales des bases de données relationnelles telles que les transactions, les migrations, le chargement paresseux et l'optimiste, le verrouillage, les sous-requêtes, les associations, etc.
  - GORM supporte également les extensions de bases de données pour PostgreSQL, MySQL, SQLite, et SQL Server.

Installation et configuration de GORM

- Installation du package GORM
  - Pour installer GORM, utilisez la commande suivante dans votre terminal :

    ```
    go get -u gorm.io/gorm
    ```

- Configuration de GORM avec Go
  - Pour utiliser GORM dans votre projet Go, importez le package comme suit :

```
import "gorm.io/gorm"
```

- Vous pouvez ensuite initialiser une instance de GORM avec vos informations de connexion à la base de données.

Les bases de GORM

- Connexion à la base de données
  - Pour se connecter à une base de données avec GORM, utilisez le code suivant :

```
import (
  "gorm.io/driver/sqlite"
  "gorm.io/gorm"
)

func main() {
  db, err := gorm.Open(sqlite.Open("test.db"), &gorm.Config{})
  if err != nil {
    panic("failed to connect database")
  }
  // use db (gorm.DB)
}
```

  - Notez que vous devez remplacer "test.db" par votre propre base de données et "sqlite.Open" par le driver correspondant à votre système de gestion de base de données (SGBD).

- Création, lecture, mise à jour et suppression d'enregistrements
  - Avec GORM, vous pouvez facilement créer, lire, mettre à jour et supprimer des enregistrements dans la base de données. Voici un exemple de comment créer un nouvel enregistrement :

```
user := User{Name: "John Doe", Age: 25}
result := db.Create(&user) // Pass pointer of data to Create
```

  - Pour lire des données, vous pouvez utiliser des méthodes comme `Find`, `First`, `Take`, etc. Par exemple :

```
var users []User
db.Find(&users)
```

  - La mise à jour des enregistrements est également simple avec GORM. Voici un exemple de mise à jour du nom d'un utilisateur :

```
db.Model(&user).Update("Name", "Jane Doe")
```

- Pour supprimer un enregistrement, vous

pouvez utiliser la méthode `Delete`. Par exemple :

```
db.Delete(&user)
```

- Migration automatique de la structure de la base de données
  - GORM propose une fonctionnalité de migration automatique qui crée et met à jour les tables en fonction de la structure de vos modèles Go. Pour effectuer une migration, vous pouvez utiliser la méthode `AutoMigrate` :

    ```
    db.AutoMigrate(&User{})
    ```

  - Notez que `AutoMigrate` ne supprimera pas les colonnes existantes ou ne changera pas les types de colonnes. Pour ces opérations, une migration manuelle est nécessaire.

Utilisation de GORM pour gérer les bases de données

Gestion des relations de bases de données avec GORM

- Relations un-à-un, un-à-plusieurs, plusieurs-à-plusieurs :
  - Go propose une représentation native des relations entre les entités via les structures. GORM exploite ces structures pour gérer les relations entre les tables.
  - Pour une relation un-à-un, on utilise un champ de type struct dans le modèle de données. Par exemple :

```
type User struct {
    gorm.Model
    Profile Profile
}
```

  - Pour une relation un-à-plusieurs, GORM utilise un slice de structs. Par exemple :

```
type User struct {
    gorm.Model
    Orders []Order
}
```

  - Les relations plusieurs-à-plusieurs sont gérées en définissant un champ de type slice de structs dans les deux modèles de données en relation.

- Association et préchargement :
  - GORM fournit les méthodes `Association` et `Preload` pour manipuler et charger les relations.
  - La méthode `Association` permet d'associer, de dissocier ou de remplacer des relations.
  - La méthode `Preload` est utilisée pour charger les relations en mémoire, ce qui permet d'éviter les requêtes N+1.

Techniques avancées avec GORM

- Jointures complexes et requêtes SQL brutes :
  - GORM permet d'exécuter des requêtes SQL brutes pour les cas où une requête complexe est nécessaire. On peut l'utiliser via la méthode `Raw`. Par exemple :

    ```
    db.Raw("SELECT * FROM users").Scan(&users)
    ```

  - Pour les jointures, GORM fournit la méthode `Joins`. Par exemple :

    ```
    db.Joins("JOIN orders ON orders.user_id = users.id").Find(&users)
    ```

- Transactions :
  - GORM supporte les transactions SQL via les méthodes `Begin`, `Commit` et `Rollback`.
  - Une transaction est initiée avec `Begin`, toutes les opérations suivantes sont ensuite effectuées dans cette transaction jusqu'à ce qu'elle soit validée (`Commit`) ou annulée (`Rollback`).

- Callbacks et hooks :
  - GORM fournit des hooks de cycle de vie pour effectuer des opérations avant ou après certaines actions (comme `Create`, `Save`, `Delete`).
  - Ces hooks peuvent être utilisés pour implémenter des fonctionnalités telles que l'horodatage automatique, la mise à jour des champs d'état, etc.

Optimisation et débogage

- Journalisation :
  - GORM fournit une interface de journalisation qui peut être configurée pour enregistrer les détails de chaque opération SQL exécutée.
  - Cette interface peut être utilisée pour déboguer les problèmes et optimiser les performances.

- Optimisation des requêtes :
  - GORM propose plusieurs techniques pour optimiser les requêtes, comme le chargement différé (`LazyLoad`), le chargement anticipé (`EagerLoad`), l'utilisation de pointeurs pour les slices, etc.

- Il est aussi possible d'éviter l'hydratation complète des objets pour optimiser les requêtes `SELECT`.

Les informations et les exemples de code peuvent être trouvés dans la documentation officielle de GORM, ainsi que

dans les tutoriels et les blogs sur le développement avec Go et GORM.

Exercices

Créer une application de gestion d'utilisateurs avec GORM et une base de données SQLite

Conception de la base de données

- Modélisation des données:
  - Créer une structure User en Go avec des champs correspondant aux attributs de l'utilisateur (par exemple, ID, Nom, Email, Mot de Passe). Les attributs de cette structure seront mappés aux colonnes de la table de la base de données.

  Exemple de code:

```
type User struct {
    gorm.Model
    Name     string
    Email    string `gorm:"type:varchar(100);uniqueIndex"`
    Password string
}
```

  * Création de la base de données avec GORM: * Utiliser la fonction `gorm.AutoMigrate` pour créer la table des utilisateurs dans la base de données SQLite.

  Exemple de code:

```
db, err := gorm.Open(sqlite.Open("test.db"), &gorm.Config{})
if err != nil {
  panic("failed to connect database")
}
db.AutoMigrate(&User{})
```

Développement de l'application

- Connexion à la base de données:
  - Utiliser `gorm.Open` pour établir une connexion avec la base de données SQLite.
- Gestion des utilisateurs (création, lecture, mise à jour, suppression):
  - Pour créer un nouvel utilisateur, utiliser la fonction `Create`.

    - Pour lire les informations d'un utilisateur, utiliser la fonction `Find`.
    - Pour mettre à jour les informations d'un utilisateur, utiliser la fonction `Save`.
    - Pour supprimer un utilisateur, utiliser la fonction `Delete`.

Tests et débogage

- Écriture et exécution de tests unitaires:
    - Utiliser le package `testing` de Go pour écrire des tests unitaires pour chaque fonctionnalité.
- Identification et correction des problèmes:
    - Utiliser les journaux de débogage GORM pour identifier les problèmes potentiels avec les requêtes SQL.

Implémenter un système de gestion de stock avec GORM et une base de données PostgreSQL

Conception de la base de données

- Modélisation des données:
    - Créer une structure Item en Go avec des champs correspondant aux attributs de l'article (par exemple, ID, Nom, Quantité).
- Création de la base de données avec GORM:
    - Utiliser la fonction `gorm.AutoMigrate` pour créer la table des articles dans la base de données PostgreSQL.

Développement de l'application

- Connexion à la base de données:
    - Utiliser `gorm.Open` pour établir une connexion avec la base de données PostgreSQL.
- Gestion du stock (ajout, mise à jour, suppression, recherche):
    - Pour ajouter un nouvel article, utiliser la fonction `Create`.
    - Pour mettre à jour les informations d'un article, utiliser la fonction `Save`.
    - Pour supprimer un article, utiliser la fonction `Delete`.
    - Pour rechercher un article, utiliser la fonction `Find`.

Tests et débogage

- Écriture et exécution de tests unitaires:
  - Utiliser le package `testing` de Go pour écrire des tests unitaires pour chaque fonctionnalité.
- Identification et correction des problèmes:
  - Utiliser les journaux de débogage GORM pour identifier les problèmes potentiels avec les requêtes SQL.

Arguments et flags

Comprendre les arguments de la ligne de commande

- Les arguments de ligne de commande sont les valeurs que vous fournissez à un programme lors de son exécution dans un terminal ou une invite de commande.
- Dans Go, vous pouvez accéder aux arguments de la ligne de commande via le slice `os.Args` de la bibliothèque standard. `os.Args[0]` est le nom du programme lui-même, `os.Args[1]` est le premier argument, `os.Args[2]` est le second, et ainsi de suite.
- Exemple de code :

```
package main
import (
  "fmt"
  "os"
)
func main() {
  fmt.Println("Arguments: ", os.Args)
}
```

- Pour exécuter ce programme avec des arguments, vous utiliseriez une commande similaire à `go run main.go arg1 arg2`.

Utilisation du package flag pour parser les arguments

- Le package `flag` de Go fournit une interface plus sophistiquée pour analyser les arguments de la ligne de commande.
- Il peut être utilisé pour définir et gérer les options de ligne de commande qui ont un nom (aussi appelées "flags").
- Exemple de code :

```
package main
import (
  "flag"
  "fmt"
)
func main() {
  name := flag.String("name", "anonymous", "Specify your name")
  flag.Parse()
```

```
  fmt.Printf("Hello, %s!\n", *name)
}
```

- Dans cet exemple, un flag nommé `name` est défini avec une valeur par défaut de "anonymous" et une courte description. Lors de l'exécution du programme, vous pouvez spécifier ce flag avec `go run main.go -name=John`.

Gestion des arguments optionnels et obligatoires

- Les flags sont généralement optionnels dans Go, mais vous pouvez ajouter une vérification manuelle pour rendre certains flags obligatoires.
- Exemple de code :

```
package main
import (
  "flag"
  "fmt"
  "os"
)
func main() {
  name := flag.String("name", "", "Specify your name")
  flag.Parse()
  if *name == "" {
    fmt.Println("Error: name is required")
    flag.PrintDefaults()
    os.Exit(2)
  }
  fmt.Printf("Hello, %s!\n", *name)
}
```

- Dans cet exemple, si le flag `name` n'est pas spécifié lors de l'exécution du programme, une erreur sera affichée et le programme se terminera. Pour rendre un flag obligatoire, vous pouvez le définir sans valeur par défaut et vérifier sa valeur après l'appel à `flag.Parse()`.

Développement d'applications en ligne de commande

Présentation de Cobra et ses avantages

- Cobra est une bibliothèque (library) Go dédiée à la création d'applications en ligne de commande (Command-Line Interface, CLI) robustes et flexibles.
- Il offre une interface simple pour créer des commandes puissantes, des sous-commandes, ainsi que des drapeaux (flags) pour ces commandes.
- Cobra est largement adopté dans l'écosystème Go, par des projets tels que Kubernetes, Hugo, et Docker, pour n'en nommer que quelques-uns.
- Il est remarquable pour ses fonctionnalités avancées, comme le support pour des suggestions de commandes, la génération automatique de documentation, et une bonne prise en charge de l'ancrage des commandes.

Installation et configuration de Cobra

- Vous pouvez installer Cobra directement avec la commande Go : `go get github.com/spf13/cobra/cobra`
- Ensuite, vous pouvez utiliser le générateur Cobra pour créer votre application CLI de base avec la commande : `cobra init --pkg-name <nom_de_votre_package>`
- Ce qui vous donne une structure de base pour votre application CLI avec un main.go, un fichier de commandes root, et des dossiers pour les commandes et les drapeaux.

Structure de base d'une application CLI avec Cobra

- Une application Cobra est constituée de commandes, de sous-commandes et de drapeaux associés.
- Chaque commande correspond à une action de votre application. Par exemple, dans une application de gestion de fichiers, vous pourriez avoir des commandes telles que `create`, `read`, `update`, `delete`.
- Les sous-commandes représentent des actions plus spécifiques. Par exemple, avec la commande `read`, vous pourriez avoir des sous-commandes comme `read textfile`, `read image`.

- Les drapeaux vous permettent de personnaliser davantage ces actions. Par exemple, avec la commande `create textfile`, vous pourriez avoir un drapeau `--uppercase` qui force le texte à être en majuscules.
- Dans Cobra, chaque commande est représentée par une instance du struct `cobra.Command`. Les sous-commandes sont simplement des instances de `cobra.Command` attachées à leur commande parente avec la méthode `AddCommand()`.
- Les drapeaux sont ajoutés à ces commandes avec les méthodes `PersistentFlags()` ou `Flags()`.
- Un exemple de code montrant la structure de base d'une application CLI avec Cobra serait :

```
package main

import (
   "fmt"
   "github.com/spf13/cobra"
)

func main() {
   var cmdPrint = &cobra.Command{
      Use:   "print [string to print]",
      Short: "Print anything to the screen",
      Args:  cobra.MinimumNArgs(1),
      Run: func(cmd *cobra.Command, args []string) {
         fmt.Println(args[0])
      },
   }

   var rootCmd = &cobra.Command{Use: "myapp"}
   rootCmd.AddCommand(cmdPrint)
   rootCmd.Execute()
}
```

Cette application permet d'imprimer du texte sur la console avec la commande `myapp print [texte]`.

Développement d'applications en ligne de commande

Création de commandes, arguments et flags avec Cobra

- Cobra est une bibliothèque Go (golang) qui facilite la création d'applications en ligne de commande (CLI).
- La création d'une commande Cobra nécessite la définition d'une structure `Command`. Par exemple :

```
import "github.com/spf13/cobra"

var maCommande = &cobra.Command{
  Use:   "maCommande",
  Short: "Description courte de maCommande",
  Long:  `Description plus détaillée de maCommande`,
  Run: func(cmd *cobra.Command, args []string) {
    // Code à exécuter lorsque maCommande est appelée
  },
}
```

- Les arguments passés à une commande sont disponibles dans la fonction `Run` à travers le paramètre `args`.
- Cobra fournit également une interface pour définir des drapeaux (flags). Par exemple, pour ajouter un flag `verbose` à notre commande :

```
maCommande.Flags().BoolP("verbose", "v", false, "Augmente la verbosité de l'output")
```

Gestion des sous-commandes

- Cobra permet de structurer votre application en utilisant des sous-commandes. Chaque sous-commande peut avoir ses propres arguments et flags.
- Pour ajouter une sous-commande à une commande existante, utilisez la méthode `AddCommand`. Par exemple :

```
var sousCommande = &cobra.Command{ /*...*/ }
maCommande.AddCommand(sousCommande)
```

- Les sous-commandes peuvent avoir leurs propres sous-commandes, ce qui permet de créer une structure d'application en arborescence.

Utilisation de Cobra pour lire la configuration de l'application

- Cobra s'intègre facilement avec Viper, une autre bibliothèque Go qui gère la configuration des applications.
- Viper permet de lire la configuration à partir de diverses sources, comme des fichiers JSON, TOML, YAML, HCL, INI, des variables d'environnement, des arguments de ligne de commande, etc.
- Pour lier les flags d'une commande Cobra à une configuration Viper, utilisez la méthode `BindPFlag`. Par exemple :

```
import "github.com/spf13/viper"

maCommande.Flags().BoolP("verbose", "v", false, "Augmente la verbosité de l'output")
viper.BindPFlag("verbose", maCommande.Flags().Lookup("verbose"))
```

- Viper lira alors la configuration à partir du flag CLI si présent, sinon à partir des autres sources configurées.
- Cela permet une grande flexibilité dans la configuration de votre application, tout en fournissant une interface CLI cohérente grâce à Cobra.

Exercices

Créer une application CLI pour interagir avec une API RESTful

Planification de l'application CLI

- Identifier les fonctionnalités de base que l'application CLI doit offrir : CRUD (Create, Read, Update, Delete) sont généralement requis pour interagir avec une API RESTful.
- Définir les entités (par exemple, les utilisateurs, les messages, les tâches) sur lesquelles les commandes CRUD seront exécutées.

Création de l'application CLI avec Cobra

- Utiliser Cobra pour initialiser une nouvelle application CLI : `cobra init --pkg-name monapp`.
- Définir des commandes de base pour l'application. Par exemple, `cobra add getUsers` pour ajouter une commande "getUsers".

Ajout des commandes pour interagir avec l'API RESTful

- Utiliser le package "net/http" pour envoyer des requêtes HTTP aux endpoints correspondants de l'API.
- Par exemple, pour la commande "getUsers", le code pourrait ressembler à ceci :

```
package cmd

import (
    "fmt"
    "net/http"
    "io/ioutil"
)

var getUsersCmd = &cobra.Command{
    Use:   "getUsers",
    Short: "Get list of users",
    Run: func(cmd *cobra.Command, args []string) {
        response, err := http.Get("http://api.example.com/users")
        if err != nil {
            fmt.Printf("The HTTP request failed with error %s\n", err)
        } else {
```

```
            data, _ := ioutil.ReadAll(response.Body)
            fmt.Println(string(data))
        }
    },
}

func init() {
    rootCmd.AddCommand(getUsersCmd)
}
```

Tests et débogage de l'application CLI

- Exécuter l'application avec `go run main.go getUsers` pour tester la commande "getUsers".
- Utiliser le package "log" pour enregistrer les erreurs et faciliter le débogage.

Développer un outil en ligne de commande pour analyser et traiter des fichiers CSV

Comprendre les besoins et les spécifications

- Identifier les opérations à effectuer sur les fichiers CSV (par exemple, lire, filtrer, trier, agréger).

Création de l'outil CLI avec Cobra

- Utiliser Cobra pour initialiser un nouvel outil CLI : `cobra init --pkg-name monoutil`.
- Ajouter des commandes de base pour l'outil. Par exemple, `cobra add readCSV` pour ajouter une commande "readCSV".

Ajout des commandes pour lire, analyser et traiter les fichiers CSV

- Utiliser le package "encoding/csv" pour lire et écrire des fichiers CSV.
- Par exemple, pour la commande "readCSV", le code pourrait ressembler à ceci :

```
package cmd

import (
    "encoding/csv"
    "fmt"
    "os"
)

var readCSVCmd = &cobra.Command{
```

```
    Use:   "readCSV",
    Short: "Read CSV file",
    Run: func(cmd *cobra.Command, args []string) {
        csvfile, err := os.Open("file.csv")
        if err != nil {
            log.Fatalln("Couldn't open the csv file", err)
        }

        r := csv.NewReader(csvfile)

        for {
            record, err := r.Read()
            if err == io.EOF {
                break


            }
            if err != nil {
                log.Fatal(err)
            }

            fmt.Println(record)
        }
    },
}

func init() {
    rootCmd.AddCommand(readCSVCmd)
}
```

Tests et optimisation de l'outil CLI

- Exécuter l'outil avec `go run main.go readCSV` pour tester la commande "readCSV".
- Utiliser le package "testing" pour écrire des tests unitaires et le package "pprof" pour profiler l'outil et optimiser son performance.

Organisation du code et gestion des dépendances

Comprendre l'importance d'une bonne organisation du code

- La qualité du code dépend grandement de son organisation.
- Une structure de code bien organisée facilite la maintenance et l'évolution du code.
- Permet une meilleure collaboration entre développeurs en rendant le code plus lisible et compréhensible.
- Réduit les erreurs et les bugs en limitant les zones de codes redondants ou inutilisés.

Principes de l'organisation du code en Go

- Le langage Go adopte une organisation de code conventionnelle pour garantir la lisibilité et l'uniformité du code.
- La structure standard d'un projet Go comprend généralement les dossiers suivants : `cmd`, `pkg`, `internal`, `api`, `web`, `scripts`, `third_party`, `vendor`, `test`, `docs`.
- Les codes sources des commandes exécutables sont généralement placés dans le répertoire `cmd`.
- Les librairies réutilisables sont placées dans `pkg`, les librairies privées dans `internal`.
- `api` contient les définitions des APIs, `web` contient les ressources web statiques, `scripts` contient les scripts de build ou d'installation.

Structure des projets Go

- Exemple de structure d'un projet Go:

```
myapp/
├── cmd/
│   └── myapp/
│       └── main.go
├── pkg/
│   └── mylib/
```

```
│       ├── mylib.go
│       └── mylib_test.go
├── api/
├── web/
├── scripts/
├── third_party/
├── vendor/
├── test/
└── docs/
```

Gestion des dépendances dans Go

- Go utilise un système de gestion de dépendances appelé Go Modules.
- Avant Go 1.11, `dep` était l'outil de gestion des dépendances recommandé, mais a été remplacé par Go Modules.

Introduction à Go Modules

- Go Modules est introduit dans Go 1.11 et devient le système standard pour la gestion des dépendances dans Go 1.13.
- Go Modules offre une solution pour la gestion des versions de paquets, leur compatibilité et leur téléchargement.

Mise en place et utilisation de Go Modules

- Pour initialiser un nouveau module, utiliser `go mod init [module path]`.
- Les dépendances sont automatiquement ajoutées au fichier `go.mod` lors de la construction du projet avec `go build` ou `go test`.
- Pour ajouter manuellement une dépendance, utiliser `go get [package@version]`.
- Le fichier `go.sum` contient les sommes de contrôle attendues pour le contenu spécifique de chaque module.

Résolution des problèmes courants avec les dépendances

- Pour mettre à jour toutes les dépendances à leur dernière version, utiliser `go get -u`.
- Pour supprimer les dépendances inutilisées du fichier `go.mod`, utiliser `go mod tidy`.

- En cas de problèmes de dépendances, il peut être utile de vider le cache avec `go clean -modcache`.

Empaquetage et distribution d'applications Go

Comprendre l'importance de l'empaquetage et de la distribution

- L'empaquetage et la distribution sont essentiels pour assurer la portabilité et la facilité d'utilisation d'une application Go.
- Ils permettent la livraison de l'application aux utilisateurs finaux dans un format exécutable sur leur plateforme cible.
- Les applications correctement emballées réduisent les problèmes de dépendances et assurent un fonctionnement homogène.

Empaquetage d'une application Go

- Go facilite l'empaquetage d'applications avec la commande `go build` qui compile le code source en un exécutable.
- Pour empaqueter une application Go, utilisez la commande `go build -o output_file` dans le terminal. Par exemple, `go build -o myapp`.
- Cela générera un fichier exécutable nommé `myapp` dans le répertoire courant.

Création d'exécutables statiques

- Les exécutables statiques sont des binaires qui embarquent toutes leurs dépendances, ce qui les rend indépendants du système.
- Pour créer un exécutable statique, il faut passer l'option `-ldflags '-extldflags "-static"'` à la commande `go build`.
- Exemple de commande pour créer un exécutable statique : `go build -ldflags '-extldflags "-static"' -o myapp`.

Distribution de l'application Go (plateformes, binaires, Docker)

- Le binaire créé peut être distribué et exécuté sur la plateforme cible, à condition que la plateforme cible soit compatible avec l'architecture pour laquelle le binaire a été construit.
- Pour créer des binaires pour plusieurs architectures, Go supporte le cross-compilation grâce aux variables d'environnement `GOOS` et `GOARCH`.
- Docker peut être utilisé pour empaqueter et distribuer l'application dans un conteneur qui inclut toutes ses dépendances, ce qui assure un environnement d'exécution cohérent.
- Exemple de Dockerfile pour une application Go :

```
FROM golang:1.17
WORKDIR /app
COPY . .
RUN go build -o myapp
CMD ["/app/myapp"]
```

Résolution des problèmes courants lors de l'empaquetage et de la distribution

- Les erreurs de compilation sont souvent dues à des dépendances manquantes ou mal gérées. Utilisez `go mod tidy` pour nettoyer et synchroniser le fichier `go.mod` avec votre code.
- Les problèmes d'architecture peuvent survenir si vous distribuez un binaire à une architecture qui n'est pas compatible avec celle pour laquelle il a été construit. Utilisez le cross-compilation pour éviter cela.
- Les erreurs liées aux bibliothèques C ne peuvent pas être résolues par la compilation statique. Pour résoudre ce problème, utilisez une image Docker de base qui contient les bibliothèques C nécessaires.

010 init

[TECHY is a technology expert with more than 20 years of experience. TECHY are specialized in systems, networks and infrastructures. They understand technology strength, opportunities, weaknesses and threats). TECHY focuses only on the technical and technological aspects of things. TECHY is methodical and analytical in his answers. TECHY admits his incompetence on other topics. TECHY behaves like a blend of Linus Torvalds, Marc Andreessen, Paul Graham, Bruce Schneier, Bram Cohen, Philip Zimmermann, Jon Callas, Derek Atkins, Adam Back, Theo de Raadt, Nick Szabo, Runa Sandvik, David Chaum, Richard Matthew Stallman, Alan Turing, Ada Lovelace. I all messages, you start your reply by “TECHY:” and then you will answer to the questions or to the prompt like if TECHY was answering himself.]

Salut TECHY, nous allons rédiger ensemble le contenu d'un cours sur le langage de programmation Go (en utilisant pour référence la version 1.17).

Pour cela j'ai besoin que prennes en compte toutes les informations suivantes, et notamment le CONTEXTE, le PUBLIC CONCERNÉ, les PRÉREQUIS, et le PROGRAMME DE LA FORMATION.

Confirme en écrivant "C'est bon pour moi" si tu as compris et que tu es prêt.

Contexte

Une formation technique d'excellente qualité à destination du MIT, de l'école polytechnique et de ORSYS.

Public concerné

Développeurs.

Objectifs pédagogiques

- Maîtriser le langage de programmation Go, incluant sa syntaxe de base, la création et utilisation de fonctions, méthodes, interfaces, la gestion des erreurs, l'utilisation des pointeurs, structures, tableaux, slices, maps, et la gestion de la concurrence.

- Savoir créer et utiliser des packages, gérer les dépendances, effectuer des tests, des benchmarks, du profiling, et documenter le code Go.
- Pouvoir développer des applications orientées réseau et web en Go, manipuler des données structurées et faire communiquer des services.
- Savoir accéder aux bases de données et utiliser des ORM en Go, y compris la création d'applications gérant des utilisateurs et des stocks.
- Être capable de développer des applications en ligne de commande en Go, comprendre et appliquer les bonnes pratiques, et connaître les bases de l'empaquetage et du déploiement d'applications Go.

Prérequis

- Comprendre les principes fondamentaux de la programmation informatique, acquis par une expérience pratique avec un langage comme Python, Java, C, etc.
- Être à l'aise avec l'utilisation des systèmes d'exploitation, tels que Unix/Linux ou Windows.
- Avoir une connaissance de base des modèles OSI et TCP/UDP/IP.
- Avoir une compréhension générale de ce qu'est une base de données relationnelle et comment elle fonctionne.
- Être conscients de l'importance des bonnes pratiques en développement de logiciel.

Programme de la formation

Cryptographie et services de sécurité

- Terminologie et principes cryptographiques.
- Principaux algorithmes de cryptographie et leurs usages dans TLS.
- Fonction de hachage avec et sans clé.
- Services de sécurité : confidentialité, authentification, intégrité.

Certificats et signature numérique

- Signature numérique.
- Certificats et mise en œuvre des clés PKCS12.
- Profils de certificats pour TLS.

Architecture de TLS

- Panorama des différentes versions, de SSL à TLS.
- Architecture, protocole et services de sécurité de TLS.

Configuration et mise en œuvre du protocole TLS

- Configuration du côté client et serveur.
- Configuration pour authentification simple du serveur.
- Mise en œuvre des certificats, paramétrages des algorithmes de chiffrement du côté serveur.
- Authentification du serveur, configuration des magasins de certificats.

Intégration de TLS dans les applications

- Principes de fonctionnement des bibliothèques TLS pour les principaux langages de programmation.
- Gestion des erreurs et des exceptions liées à TLS.
- Bonnes pratiques de programmation sécurisée avec TLS.

Analyse de sécurité et perspectives du protocole TLS

- Attaques sur le protocole TLS et impact sur les applications.
- Bonnes pratiques, contrôle des configurations.
- Présentation du protocole DTLS.

Sécurité des applications et gestion des vulnérabilités liées à TLS

- Identification et correction des vulnérabilités liées à TLS dans les applications.
- Sensibilisation aux failles de sécurité courantes et aux bonnes pratiques pour les prévenir.

015 init merge

@@ V1

@@ V2

@@ REQUEST

V1 et V2 représentent deux versions du même cours/

Commence par comparer les deux versions, vérifier leur exactitude et résumer les avantages et inconvénients de chaque version, même concernant l'ordre des informations.

Ensuite, propose moi une version réorganisée et fusionnée des deux (v3), issue des deux versions précédentes (la tienne et la mienne), qui à la fois: * facilite la compréhension, * réorganise l'ordre du contenu si besoin, * garde toutes les informations factuelles, garde aussi les autres informations les plus vraies et les plus précises, * inclue un maximum de détails (dates, commandes, options, subtilités, etc.).

020 chapter toc

Merci.

Suit la structure du PROGRAMME DE FORMATION. Focalise toi sur le chapitre « FIXME » Et puis spécifiquement sur les sujets suivants «

FIXME

»

Rédige la table des matières détaillée de ces différents sujets, au format suivant:

1. Titre 1.1. Sous-titre 1.2. ...
2. Titre
3. ...

030 section content

Super.

Suit la structure du PROGRAMME DE FORMATION.

Concentre toi sur le chapitre « CHAPTER »

Dans ce chapitre, concentre toi sur la section « SECTION ».

Focalise toi plus spécifiquement sur les sous-sections suivantes «

SECTION CONTENT

» de cette section, sur laquelle on se concentre.

Rédige le contenu détaillée de ces différentes sous-sections, sous forme phrases courtes et de listes à puces (bullet-points) pour remplir le contenu d'une présentation PowerPoint ou MARP

Identifie les causalités (conditions, causes, conséquences).

Donne les informations les plus précises et les plus techniques.

Evite les mots flous. Précise les mots "gérer" ou "permet" quand tu les utilises.

Indique où trouver les informations, donne des exemple de code ainsi que les différentes commandes go ou autres outils) pour le terminal.

040 section merge content

@@ AUTRE VERSION DU COURS

@@ REQUEST

Le contenu de AUTRE VERSION DU COURS est issu d'un autre cours (que l'on appelle v2), pour le même chapitre, et la même section.

Commence par comparer les deux versions, vérifier leur exactitude et résumer les avantages et inconvénients de chaque version, même concernant l'ordre des informations.

Ensuite, propose moi une version réorganisée et fusionnée des deux (v3), issue des deux versions précédentes (la tienne et la mienne), qui à la fois: * facilite la compréhension, * réorganise l'ordre du contenu si besoin, * garde toutes les informations factuelles, garde aussi les autres informations les plus vraies et les plus précises, * inclue un maximum de détails (dates, commandes, options, subtilités, etc.).

050 section explain

Super. Suit la structure du PROGRAMME DE FORMATION. Concentre toi sur le chapitre « {{this.parent.title}} » Focalise toi plus spécifiquement sur la section suivante « {{this.title}} » Explique plus en détail les sous-sections suivantes : «

{{this.children}}

»

Rédige le contenu détaillée de ces différentes sous-sections, sous forme phrases courtes et de listes à puces (bullet-points) pour remplir le contenu d'une présentation PowerPoint.

Donne des informations plus précises et plus techniques.

Précise les mots "gérer" ou "permet" quand tu les utilises.

Indique où trouver les informations, ainsi que les différentes commandes openssl (ou autres outils) pour le terminal.

Donne également des exemples d'utilisation avec des exemples en ligne de commande ou des extrait de code, ou de configuration.

060 section fix

@@ TEXTE A CORRIGER

[[FIXME: text]]

@@ REQUETE

Dans le TEXTE A CORRIGER, indique où sont les erreurs et les approximations, et propose des corrections pour améliorer le contenu (pour une formation sur ssl/tls).

120.000 Bonnes pratiques, empaquetage et déploiement

Organisation du code et gestion des dépendances

- Comprendre l'importance d'une bonne organisation du code
- Principes de l'organisation du code en Go
- Structure des projets Go
- Gestion des dépendances dans Go
- Introduction à Go Modules
- Mise en place et utilisation de Go Modules
- Résolution des problèmes courants avec les dépendances

Documentation et commentaires

- Comprendre l'importance de la documentation
- Conventions de commentaires en Go
- Documenter le code source (variables, types, fonctions, packages)
- Générer une documentation avec godoc
- Bonnes pratiques pour la rédaction de commentaires

Empaquetage et distribution d'applications Go

- Comprendre l'importance de l'empaquetage et de la distribution
- Empaquetage d'une application Go
- Création d'exécutables statiques
- Distribution de l'application Go (plateformes, binaires, Docker)
- Résolution des problèmes courants lors de l'empaquetage et de la distribution

Travaux pratiques

Organiser et documenter un projet Go existant

- FIXME: à définir

Compiler et distribuer une application Go en tant qu'exécutable statique

- FIXME: à définir

FIN.

120.000 Bonnes pratiques, empaquetage et déploiement

Organisation du code et gestion des dépendances

- Comprendre l'importance d'une bonne organisation du code
- Principes de l'organisation du code en Go
- Structure des projets Go
- Gestion des dépendances dans Go
- Introduction à Go Modules
- Mise en place et utilisation de Go Modules
- Résolution des problèmes courants avec les dépendances

FIN.

120.000 Bonnes pratiques, empaquetage et déploiement

Documentation et commentaires

- Comprendre l'importance de la documentation
- Conventions de commentaires en Go
- Documenter le code source (variables, types, fonctions, packages)
- Générer une documentation avec godoc
- Bonnes pratiques pour la rédaction de commentaires

FIN.

120.000 Bonnes pratiques, empaquetage et déploiement

Empaquetage et distribution d'applications Go

- Comprendre l'importance de l'empaquetage et de la distribution
- Empaquetage d'une application Go
- Création d'exécutables statiques
- Distribution de l'application Go (plateformes, binaires, Docker)
- Résolution des problèmes courants lors de l'empaquetage et de la distribution

FIN.

120.000 Bonnes pratiques, empaquetage et déploiement

Travaux pratiques

- Organiser et documenter un projet Go existant
- Compiler et distribuer une application Go en tant qu'exécutable statique

FIN.

000.000 Le langage Go

010.000 Introduction, historique et installation de Go

- Historique et objectifs du langage
- Comparaison de Go avec d'autres langages populaires (C, Python, Java, etc.)
- Installation et configuration de l'environnement de développement

Travaux pratiques :

- Installer Go et configurer l'environnement de développement
- Créer un "Hello, World!" en Go, compiler, exécuter le programme

020.000 Syntaxe, variables, types de base et structures de contrôle

- Variables, constantes et déclarations
- Types de données de base
- Structures de contrôle (if, for, switch, defer)

Travaux pratiques :

- Calculer la somme et la moyenne d'une liste de nombres
- Créer un programme pour convertir des températures entre Celsius et Fahrenheit

030.000 Fonctions, méthodes, interfaces et gestion des erreurs

- Déclaration et utilisation de fonctions
- Méthodes et interfaces
- Gestion des erreurs et conventions

Travaux pratiques :

- Implémenter une calculatrice avec des opérations personnalisées
- Créer une bibliothèque pour lire et écrire des fichiers CSV avec une gestion d'erreurs robuste

040.000 Pointeurs, allocation de mémoire, structures et types composites

- Pointeurs et allocation de mémoire
- Structures et méthodes
- Tableaux, slices et maps

Travaux pratiques :

- Créer une bibliothèque pour gérer des formes géométriques et calculer leurs aires
- Créer un programme de gestion d'inventaire

050.000 Concurrence, goroutines et synchronisation

- Goroutines et concurrence
- Canaux et communication entre goroutines
- Patterns de concurrence et select

Travaux pratiques :

- Calculer des factoriels en parallèle
- Développer un système de files d'attente (queue) pour traiter des tâches en parallèle

060.000 Packages, modules et gestion des dépendances

- Création et utilisation de packages
- Go modules et gestion des dépendances

Travaux pratiques :

- Créer un package personnalisé
- Utiliser un package externe dans un projet Go

070.000 Tests, benchmarks, profiling et documentation

- Tests unitaires et benchmarks
- Profiling et optimisation
- Documentation et commentaires

Travaux pratiques :

- Écrire des tests unitaires pour une application de calculatrice
- Analyser et optimiser les performances d'un programme

080.000 Programmation orientée réseau et développement d'applications web

- Client et serveur TCP/UDP
- Création d'un serveur HTTP simple
- Utilisation de gorilla/mux pour créer une API RESTful

Travaux pratiques :

- Créer un serveur de chat simple avec TCP
- Implémenter une API RESTful pour gérer une liste de tâches

090.000 Manipulation de données structurées et communication entre services

- Encodage et décodage JSON
- Introduction à Google Protocol Buffers
- Utilisation de golang/protobuf

Travaux pratiques :

- Créer une API RESTful qui renvoie des données JSON
- Implémenter un client et un serveur qui communiquent avec des messages protobuf

100.000 Accès aux bases de données et ORM

- Bases de données relationnelles et SQL
- Introduction à GORM
- Utilisation de jinzhu/gorm pour gérer les bases de données

Travaux pratiques :

- Créer une application de gestion d'utilisateurs avec GORM et une base de données SQLite
- Implémenter un système de gestion de stock avec GORM et une base de données PostgreSQL

110.000 Développement d'applications en ligne de commande

- Arguments et flags
- Introduction à spf13/cobra
- Utilisation de spf13/cobra pour créer des applications CLI

Travaux pratiques :

- Créer une application CLI pour interagir avec une API RESTful
- Développer un outil en ligne de commande pour analyser et traiter des fichiers CSV

120.000 Bonnes pratiques, empaquetage et déploiement

- Organisation du code et gestion des dépendances
- Documentation et commentaires
- Empaquetage et distribution d'applications Go

Travaux pratiques :

- Organiser et documenter un projet Go existant
- Compiler et distribuer une application Go en tant qu'exécutable statique

FIN.

010.000 Introduction, historique et installation de Go

Historique et objectifs du langage

- Origines de Go
- Contributeurs clés
- Objectifs de Go

Comparaison de Go avec d'autres langages populaires (C, Python, Java, etc.)

- Comparaison syntaxique
- Comparaison des performances
- Comparaison des paradigmes de programmation
- Comparaison des écosystèmes

Installation et configuration de l'environnement de développement

- Installer Go sur différentes plateformes
- Configuration des variables d'environnement
- Installation d'un IDE ou d'un éditeur de texte adapté

Travaux pratiques

Installer Go et configurer l'environnement de développement

- Télécharger et installer Go
- Configurer les variables d'environnement
- Installer un IDE ou un éditeur de texte adapté

Créer un "Hello, World!" en Go, compiler, exécuter le programme

- Créer un fichier Go avec le code "Hello, World!"

- Compiler le programme
- Exécuter le programme

FIN.

010.000 Introduction, historique et installation de Go

Historique et objectifs du langage

- Origines de Go
- Contributeurs clés
- Objectifs de Go

FIN.

010.000 Introduction, historique et installation de Go

Comparaison de Go avec d'autres langages populaires (C, Python, Java, etc.)

- Comparaison syntaxique
- Comparaison des performances
- Comparaison des paradigmes de programmation
- Comparaison des écosystèmes

FIN.

010.000 Introduction, historique et installation de Go

Installation et configuration de l'environnement de développement

- Installer Go sur différentes plateformes
- Configuration des variables d'environnement
- Installation d'un IDE ou d'un éditeur de texte adapté

FIN.

010.000 Introduction, historique et installation de Go

Travaux pratiques

Installer Go et configurer l'environnement de développement

- Télécharger et installer Go
- Configurer les variables d'environnement
- Installer un IDE ou un éditeur de texte adapté

Créer un "Hello, World!" en Go, compiler, exécuter le programme

- Créer un fichier Go avec le code "Hello, World!"
- Compiler le programme
- Exécuter le programme

FIN.

020.000 Syntaxe, variables, types de base et structures de contrôle (v1)

Variables, constantes et déclarations

- Déclaration de variables * Variables à court terme (:=)
- • Variables avec initialisation
- Déclaration de constantes
- Les opérateurs de base

Types de données de base

- Entiers (int, uint, int8, uint8, etc.)
- Flottants (float32, float64)
- Booléens (bool)
- Chaînes de caractères (string)
- Type rune
- Type byte
- Conversion de types (type casting)

Structures de contrôle (if, for, switch, defer)

- Condition if-else
- Boucles for et variations (for-range)
- Instruction switch-case
- Utilisation de defer

Travaux pratiques

Calculer la somme et la moyenne d'une liste de nombres

- Création d'une liste de nombres
- Calcul de la somme
- Calcul de la moyenne

- Affichage des résultats

Créer un programme pour convertir des températures entre Celsius et Fahrenheit

- Saisie de la température en Celsius
- Conversion en Fahrenheit
- Saisie de la température en Fahrenheit
- Conversion en Celsius
- Affichage des résultats

FIN.

020.000 Syntaxe, variables, types de base et structures de contrôle (v1)

Variables, constantes et déclarations

- Déclaration de variables * Variables à court terme (:=)
- • Variables avec initialisation
- Déclaration de constantes
- Les opérateurs de base

FIN.

020.000 Syntaxe, variables, types de base et structures de contrôle (v1)

Types de données de base

- Entiers (int, uint, int8, uint8, etc.)
- Flottants (float32, float64)
- Booléens (bool)
- Chaînes de caractères (string)
- Type rune
- Type byte
- Conversion de types (type casting)

FIN.

020.000 Syntaxe, variables, types de base et structures de contrôle (v1)

Structures de contrôle (if, for, switch, defer)

- Condition if-else
- Boucles for et variations (for-range)
- Instruction switch-case
- Utilisation de defer

FIN.

020.000 Syntaxe, variables, types de base et structures de contrôle (v1)

Travaux pratiques

Calculer la somme et la moyenne d'une liste de nombres

- Création d'une liste de nombres
- Calcul de la somme
- Calcul de la moyenne
- Affichage des résultats

Créer un programme pour convertir des températures entre Celsius et Fahrenheit

- Saisie de la température en Celsius
- Conversion en Fahrenheit
- Saisie de la température en Fahrenheit
- Conversion en Celsius
- Affichage des résultats

FIN.

030.000 Fonctions, méthodes, interfaces et gestion des erreurs

Déclaration et utilisation de fonctions

- Syntaxe de déclaration de fonctions
- Paramètres et arguments
- Valeurs de retour et déclaration de plusieurs valeurs de retour
- Fonctions anonymes et fermetures (closures)
- Fonctions variadiques

Méthodes et interfaces

- Méthodes sur les types
- Interfaces : définition et utilisation
- Composition d'interfaces
- Méthodes avec des pointeurs et des récepteurs de valeur
- Interfaces vides et assertions de type

Gestion des erreurs et conventions

- L'interface "error"
- Création et propagation d'erreurs personnalisées
- Utilisation de panic et recover
- Utilisation du package "errors" et "fmt.Errorf"
- Techniques de gestion d'erreurs courantes

Travaux pratiques

Implémenter une calculatrice avec des opérations personnalisées

- Création des fonctions de base (addition, soustraction, multiplication, division)
- Extension avec des opérations personnalisées
- Utilisation des interfaces pour gérer différentes opérations

- Tests et validation de la calculatrice

Créer une bibliothèque pour lire et écrire des fichiers CSV avec une gestion d'erreurs robuste

- Lire des fichiers CSV avec le package "encoding/csv"
- Gestion des erreurs lors de la lecture des fichiers CSV
- Écrire des fichiers CSV avec le package "encoding/csv"
- Gestion des erreurs lors de l'écriture des fichiers CSV

FIN.

030.000 Fonctions, méthodes, interfaces et gestion des erreurs

Déclaration et utilisation de fonctions

- Syntaxe de déclaration de fonctions
- Paramètres et arguments
- Valeurs de retour et déclaration de plusieurs valeurs de retour
- Fonctions anonymes et fermetures (closures)
- Fonctions variadiques

FIN.

030.000 Fonctions, méthodes, interfaces et gestion des erreurs

Méthodes et interfaces

- Méthodes sur les types
- Interfaces : définition et utilisation
- Composition d'interfaces
- Méthodes avec des pointeurs et des récepteurs de valeur
- Interfaces vides et assertions de type

FIN.

030.000 Fonctions, méthodes, interfaces et gestion des erreurs

Gestion des erreurs et conventions

- L'interface "error"
- Création et propagation d'erreurs personnalisées
- Utilisation de panic et recover
- Utilisation du package "errors" et "fmt.Errorf"
- Techniques de gestion d'erreurs courantes

FIN.

030.000 Fonctions, méthodes, interfaces et gestion des erreurs

Travaux pratiques

Implémenter une calculatrice avec des opérations personnalisées

- Création des fonctions de base (addition, soustraction, multiplication, division)
- Extension avec des opérations personnalisées
- Utilisation des interfaces pour gérer différentes opérations
- Tests et validation de la calculatrice

Créer une bibliothèque pour lire et écrire des fichiers CSV avec une gestion d'erreurs robuste

- Lire des fichiers CSV avec le package "encoding/csv"
- Gestion des erreurs lors de la lecture des fichiers CSV
- Écrire des fichiers CSV avec le package "encoding/csv"
- Gestion des erreurs lors de l'écriture des fichiers CSV

FIN.

040.000 Pointeurs, allocation de mémoire, structures et types composites

Pointeurs et allocation de mémoire

- Introduction aux pointeurs
- Opérateurs de pointeur
- Allocation dynamique de mémoire avec make et new
- Utilisation des pointeurs dans les fonctions
- Passage par référence et passage par valeur

Structures et méthodes

- Définition des structures
- Création et initialisation des instances de structures
- Méthodes associées aux structures
- Méthodes avec récepteurs de pointeurs
- Composition et héritage
- Interfaces et polymorphisme

Tableaux, slices et maps

- Tableaux et initialisation
- Slices : création, manipulation et capacité
- Maps : création, manipulation et itération

Travaux pratiques

Créer une bibliothèque pour gérer des formes géométriques et calculer leurs aires

- Définition des structures de formes géométriques
- Implémentation des méthodes de calcul d'aire
- Utilisation de la bibliothèque dans un programme

Créer un programme de gestion d'inventaire

- Modélisation des structures de données pour l'inventaire
- Fonctionnalités d'ajout, de suppression et de modification d'articles
- Recherche et filtrage des articles
- Sauvegarde et chargement de l'inventaire à partir d'un fichier

FIN.

040.000 Pointeurs, allocation de mémoire, structures et types composites

Pointeurs et allocation de mémoire

- Introduction aux pointeurs
- Opérateurs de pointeur
- Allocation dynamique de mémoire avec make et new
- Utilisation des pointeurs dans les fonctions
- Passage par référence et passage par valeur

FIN.

040.000 Pointeurs, allocation de mémoire, structures et types composites

Structures et méthodes

- Définition des structures
- Création et initialisation des instances de structures
- Méthodes associées aux structures
- Méthodes avec récepteurs de pointeurs
- Composition et héritage
- Interfaces et polymorphisme

FIN.

040.000 Pointeurs, allocation de mémoire, structures et types composites

Tableaux, slices et maps

- Tableaux et initialisation
- Slices : création, manipulation et capacité
- Maps : création, manipulation et itération

FIN.

040.000 Pointeurs, allocation de mémoire, structures et types composites

Travaux pratiques

Créer une bibliothèque pour gérer des formes géométriques et calculer leurs aires

- Définition des structures de formes géométriques
- Implémentation des méthodes de calcul d'aire
- Utilisation de la bibliothèque dans un programme

Créer un programme de gestion d'inventaire

- Modélisation des structures de données pour l'inventaire
- Fonctionnalités d'ajout, de suppression et de modification d'articles
- Recherche et filtrage des articles
- Sauvegarde et chargement de l'inventaire à partir d'un fichier

FIN.

050.000 Concurrence, goroutines et synchronisation

Goroutines et concurrence

- Introduction à la concurrence et aux goroutines
- Création et utilisation des goroutines
- Gestion des goroutines avec sync.WaitGroup
- Parallélisme vs Concurrence
- Comparaison des goroutines avec les threads
- Techniques de programmation concurrente en Go

Canaux et communication entre goroutines

- Introduction aux canaux (channels)
- Création et utilisation des canaux
- Types de canaux: non-buffered, buffered, unidirectionnels et bidirectionnels
- Communication entre goroutines via les canaux
- Canaux unidirectionnels et bidirectionnels
- Gestion des canaux avec close et range

Patterns de concurrence et select

- Introduction au select
- Introduction au pattern "fan-in/fan-out"
- Utilisation de select pour gérer plusieurs canaux
- Timeouts et tickers avec select
- Implémentation de patterns de concurrence avancés
- Bonnes pratiques pour la programmation concurrente en Go

Travaux pratiques

Calculer des factoriels en parallèle

- Implémentation séquentielle du calcul de factoriels
- Conversion de l'implémentation en version concurrente avec goroutines et canaux
- Analyse des performances et optimisation

Développer un système de files d'attente (queue) pour traiter des tâches en parallèle

- Conception d'un système de files d'attente
- Implémentation du système de files d'attente avec goroutines et canaux
- Gestion des erreurs et synchronisation
- Optimisation et bonnes pratiques pour le traitement en parallèle

FIN.

050.000 Concurrence, goroutines et synchronisation

Goroutines et concurrence

- Introduction à la concurrence et aux goroutines
- Création et utilisation des goroutines
- Gestion des goroutines avec sync.WaitGroup
- Parallélisme vs Concurrence
- Comparaison des goroutines avec les threads
- Techniques de programmation concurrente en Go

FIN.

050.000 Concurrence, goroutines et synchronisation

Canaux et communication entre goroutines

- Introduction aux canaux (channels)
- Création et utilisation des canaux
- Types de canaux: non-buffered, buffered, unidirectionnels et bidirectionnels
- Communication entre goroutines via les canaux
- Canaux unidirectionnels et bidirectionnels
- Gestion des canaux avec close et range

FIN.

050.000 Concurrence, goroutines et synchronisation

Patterns de concurrence et select

- Introduction au select
- Introduction au pattern "fan-in/fan-out"
- Utilisation de select pour gérer plusieurs canaux
- Timeouts et tickers avec select
- Implémentation de patterns de concurrence avancés
- Bonnes pratiques pour la programmation concurrente en Go

FIN.

050.000 Concurrence, goroutines et synchronisation

Travaux pratiques

Calculer des factoriels en parallèle

- Implémentation séquentielle du calcul de factoriels
- Conversion de l'implémentation en version concurrente avec goroutines et canaux
- Analyse des performances et optimisation

Développer un système de files d'attente (queue) pour traiter des tâches en parallèle

- Conception d'un système de files d'attente
- Implémentation du système de files d'attente avec goroutines et canaux
- Gestion des erreurs et synchronisation
- Optimisation et bonnes pratiques pour le traitement en parallèle

FIN.

060.000 Packages, modules et gestion des dépendances

Création et utilisation de packages

- Structure d'un package
- Déclaration d'un package
- Exportation de fonctions et de variables
- Importation et utilisation de packages

Go modules et gestion des dépendances

- Introduction aux Go modules
- Création d'un module Go
- Importation de packages externes
- Mise à jour et gestion des versions de packages

Travaux pratiques :

Créer un package personnalisé

- Concevoir un package pour un usage spécifique
- Créer le package avec la structure appropriée
- Exporter et utiliser les fonctions et variables du package
- Importer et utiliser le package dans un projet Go

Utiliser un package externe dans un projet Go

- Recherche et sélection d'un package externe
- Création d'un module Go pour le projet
- Importation et utilisation du package externe
- Gestion des versions et mise à jour du package

FIN.

060.000 Packages, modules et gestion des dépendances

Création et utilisation de packages

- Structure d'un package
- Déclaration d'un package
- Exportation de fonctions et de variables
- Importation et utilisation de packages

FIN.

060.000 Packages, modules et gestion des dépendances

Go modules et gestion des dépendances

- Introduction aux Go modules
- Création d'un module Go
- Importation de packages externes
- Mise à jour et gestion des versions de packages

FIN.

060.000 Packages, modules et gestion des dépendances

Travaux pratiques :

Créer un package personnalisé

- Concevoir un package pour un usage spécifique
- Créer le package avec la structure appropriée
- Exporter et utiliser les fonctions et variables du package
- Importer et utiliser le package dans un projet Go

Utiliser un package externe dans un projet Go

- Recherche et sélection d'un package externe
- Création d'un module Go pour le projet
- Importation et utilisation du package externe
- Gestion des versions et mise à jour du package

FIN.

070.00 Tests, benchmarks, profiling et documentation

Tests unitaires et benchmarks

- Introduction aux tests unitaires en Go
- Utilisation du package "testing"
- Techniques de test : table-driven tests, mock objects
- Création et exécution de benchmarks

Profiling et optimisation

- Introduction au profiling
- Utilisation de l'outil "pprof" pour le profiling CPU, mémoire et blocage
- Interprétation des résultats de profiling
- Identification des goulots d'étranglement
- Techniques d'optimisation, bonnes pratiques et astuces
- Mesure des améliorations après optimisation

Documentation et commentaires

. Commentaires de code : bonnes pratiques et conventions . Utilisation de "godoc" pour générer la documentation . Documentation des packages, fonctions, types et méthodes . Intégration de la documentation dans un workflow de développement

Travaux pratiques

Écrire des tests unitaires pour une application de calculatrice

- Définir les fonctions de l'application calculatrice
- Créer des tests unitaires pour les différentes opérations
- Utiliser des table-driven tests pour les tests unitaires
- Exécuter les tests et analyser les résultats

Analyser et optimiser les performances d'un programme

- Choisir un programme à optimiser
- Profiler le programme à l'aide de "pprof" pour identifier les goulots d'étranglement
- Appliquer des techniques d'optimisation pour améliorer les performances
- Mesurer les améliorations à l'aide des benchmarks et du profiling

FIN.

070.00 Tests, benchmarks, profiling et documentation

Tests unitaires et benchmarks

- Introduction aux tests unitaires en Go
- Utilisation du package "testing"
- Techniques de test : table-driven tests, mock objects
- Création et exécution de benchmarks

FIN.

070.00 Tests, benchmarks, profiling et documentation

Profiling et optimisation

- Introduction au profiling
- Utilisation de l'outil "pprof" pour le profiling CPU, mémoire et blocage
- Interprétation des résultats de profiling
- Identification des goulots d'étranglement
- Techniques d'optimisation, bonnes pratiques et astuces
- Mesure des améliorations après optimisation

FIN.

070.00 Tests, benchmarks, profiling et documentation

Documentation et commentaires

. Commentaires de code : bonnes pratiques et conventions . Utilisation de "godoc" pour générer la documentation . Documentation des packages, fonctions, types et méthodes . Intégration de la documentation dans un workflow de développement

FIN.

070.00 Tests, benchmarks, profiling et documentation

Travaux pratiques

Écrire des tests unitaires pour une application de calculatrice

- Définir les fonctions de l'application calculatrice
- Créer des tests unitaires pour les différentes opérations
- Utiliser des table-driven tests pour les tests unitaires
- Exécuter les tests et analyser les résultats

Analyser et optimiser les performances d'un programme

- Choisir un programme à optimiser
- Profiler le programme à l'aide de "pprof" pour identifier les goulots d'étranglement
- Appliquer des techniques d'optimisation pour améliorer les performances
- Mesurer les améliorations à l'aide des benchmarks et du profiling

FIN.

8. Programmation orientée réseau et développement d'applications web

Client et serveur TCP/UDP

- Concepts de base des protocoles TCP et UDP
- Création d'un client TCP en Go
- Création d'un serveur TCP en Go
- Création d'un client UDP en Go
- Création d'un serveur UDP en Go
- Gestion des erreurs et des déconnexions

Création d'un serveur HTTP simple

- Fonctionnement du protocole HTTP
- Gestion des requêtes et réponses HTTP en Go
- Création d'un serveur HTTP simple avec le package net/http
- Gestion des routes et des méthodes HTTP
- Utilisation de middlewares pour gérer l'authentification et les erreurs

Utilisation de gorilla/mux pour créer une API RESTful

- Introduction à l'architecture REST
- Présentation de gorilla/mux
- Création d'une API RESTful avec gorilla/mux
- Gestion des paramètres de requête et des variables d'URL
- Validation des données et gestion des erreurs

Travaux pratiques :

Créer un serveur de chat simple avec TCP

- Conception de l'architecture du serveur de chat
- Implémentation du serveur de chat en utilisant TCP

- Création d'un client de chat en utilisant TCP
- Gestion des erreurs et des déconnexions

Implémenter une API RESTful pour gérer une liste de tâches

- Conception de l'API pour la gestion des tâches
- Mise en place de l'API RESTful avec gorilla/mux
- Implémentation des opérations CRUD (Create, Read, Update, Delete)
- Gestion des erreurs et validation des données

FIN.

8. Programmation orientée réseau et développement d'applications web

Client et serveur TCP/UDP

- Concepts de base des protocoles TCP et UDP
- Création d'un client TCP en Go
- Création d'un serveur TCP en Go
- Création d'un client UDP en Go
- Création d'un serveur UDP en Go
- Gestion des erreurs et des déconnexions

FIN.

8. Programmation orientée réseau et développement d'applications web

Création d'un serveur HTTP simple

- Fonctionnement du protocole HTTP
- Gestion des requêtes et réponses HTTP en Go
- Création d'un serveur HTTP simple avec le package net/http
- Gestion des routes et des méthodes HTTP
- Utilisation de middlewares pour gérer l'authentification et les erreurs

FIN.

8. Programmation orientée réseau et développement d'applications web

Utilisation de gorilla/mux pour créer une API RESTful

- Introduction à l'architecture REST
- Présentation de gorilla/mux
- Création d'une API RESTful avec gorilla/mux
- Gestion des paramètres de requête et des variables d'URL
- Validation des données et gestion des erreurs

FIN.

8. Programmation orientée réseau et développement d'applications web

Travaux pratiques :

Créer un serveur de chat simple avec TCP

- Conception de l'architecture du serveur de chat
- Implémentation du serveur de chat en utilisant TCP
- Création d'un client de chat en utilisant TCP
- Gestion des erreurs et des déconnexions

Implémenter une API RESTful pour gérer une liste de tâches

- Conception de l'API pour la gestion des tâches
- Mise en place de l'API RESTful avec gorilla/mux
- Implémentation des opérations CRUD (Create, Read, Update, Delete)
- Gestion des erreurs et validation des données

FIN.

9. Manipulation de données structurées et communication entre services

Encodage et décodage JSON

- Introduction au format JSON
- Le package "encoding/json" de la bibliothèque standard
- Marshalling et Unmarshalling de données JSON
- Personnalisation de l'encodage et décodage avec des tags struct
- Gérer les erreurs liées à l'encodage et décodage JSON

Introduction à Google Protocol Buffers

- Présentation et avantages de Google Protocol Buffers
- Définition des fichiers .proto
- Structure des messages et des services
- Types de données et règles de compatibilité
- Compilation des fichiers .proto

Utilisation de golang/protobuf

- Installation et configuration de golang/protobuf
- Génération de code Go à partir de fichiers .proto
- Encodage et décodage de messages Protobuf en Go
- Implémentation de services RPC avec golang/protobuf
- Gérer les erreurs et les performances avec golang/protobuf

Travaux pratiques

Créer une API RESTful qui renvoie des données JSON

- Présentation du projet et des objectifs
- Conception de l'API RESTful avec gorilla/mux
- Implémentation de l'encodage et décodage JSON

- Test et débogage de l'API RESTful
- Retour d'expérience et améliorations

Implémenter un client et un serveur qui communiquent avec des messages protobuf

- Présentation du projet et des objectifs
- Conception des messages et services protobuf
- Génération de code Go à partir de fichiers .proto
- Implémentation du client et du serveur en Go
- Test, débogage et optimisation des communications protobuf

FIN.

9. Manipulation de données structurées et communication entre services

Encodage et décodage JSON

- Introduction au format JSON
- Le package "encoding/json" de la bibliothèque standard
- Marshalling et Unmarshalling de données JSON
- Personnalisation de l'encodage et décodage avec des tags struct
- Gérer les erreurs liées à l'encodage et décodage JSON

FIN.

9. Manipulation de données structurées et communication entre services

Introduction à Google Protocol Buffers

- Présentation et avantages de Google Protocol Buffers
- Définition des fichiers .proto
- Structure des messages et des services
- Types de données et règles de compatibilité
- Compilation des fichiers .proto

FIN.

9. Manipulation de données structurées et communication entre services

Utilisation de golang/protobuf

- Installation et configuration de golang/protobuf
- Génération de code Go à partir de fichiers .proto
- Encodage et décodage de messages Protobuf en Go
- Implémentation de services RPC avec golang/protobuf
- Gérer les erreurs et les performances avec golang/protobuf

FIN.

9. Manipulation de données structurées et communication entre services

Travaux pratiques

Créer une API RESTful qui renvoie des données JSON

- Présentation du projet et des objectifs
- Conception de l'API RESTful avec gorilla/mux
- Implémentation de l'encodage et décodage JSON
- Test et débogage de l'API RESTful
- Retour d'expérience et améliorations

Implémenter un client et un serveur qui communiquent avec des messages protobuf

- Présentation du projet et des objectifs
- Conception des messages et services protobuf
- Génération de code Go à partir de fichiers .proto
- Implémentation du client et du serveur en Go
- Test, débogage et optimisation des communications protobuf

FIN.

100.000 Accès aux bases de données et ORM

Rappels sur les bases de données relationnelles et SQL

- Rappel: Introduction aux bases de données relationnelles
  - Définition et concepts fondamentaux
  - Modèle relationnel et langage SQL
- Rappel: Concepts clés de SQL
  - DDL (Data Definition Language): CREATE, ALTER, DROP
  - DML (Data Manipulation Language): SELECT, INSERT, UPDATE, DELETE
  - Clauses SQL : WHERE, ORDER BY, GROUP BY, HAVING, JOIN
- Rappel: Pratique de SQL
  - Environnement SQL et outils de travail
  - Création et manipulation de tables
  - Exécution de requêtes SQL

Introduction à GORM

- Présentation de GORM
  - Qu'est-ce qu'un ORM (Object Relational Mapping) ?
  - Pourquoi utiliser GORM ?
  - Fonctionnalités clés de GORM
- Installation et configuration de GORM
  - Installation du package GORM
  - Configuration de GORM avec Go
- Les bases de GORM
  - Connexion à la base de données
  - Création, lecture, mise à jour et suppression d'enregistrements
  - Migration automatique de la structure de la base de données

Utilisation de jinzhu/gorm pour gérer les bases de données

- Gestion des relations de bases de données avec GORM
  - Relations un-à-un, un-à-plusieurs, plusieurs-à-plusieurs
  - Association et préchargement
- Techniques avancées avec GORM
  - Jointures complexes et requêtes SQL brutes
  - Transactions
  - Callbacks et hooks
- Optimisation et débogage
  - Journalisation
  - Optimisation des requêtes

Travaux pratiques

Créer une application de gestion d'utilisateurs avec GORM et une base de données SQLite

- Conception de la base de données
  - Modélisation des données
  - Création de la base de données avec GORM
- Développement de l'application
  - Connexion à la base de données
  - Gestion des utilisateurs (création, lecture, mise à jour, suppression)
- Tests et débogage
  - Écriture et exécution de tests unitaires
  - Identification et correction des problèmes

Implémenter un système de gestion de stock avec GORM et une base de données PostgreSQL

- Conception de la base de données
  - Modélisation des données
  - Création de la base de données avec GORM

- Développement de l'application
    - Connexion à la base de données
    - Gestion du stock (ajout, mise à jour, suppression, recherche)
- Tests et débogage
    - Écriture et exécution de tests unitaires
    - Identification et correction des problèmes

FIN.

100.000 Accès aux bases de données et ORM

Rappels sur les bases de données relationnelles et SQL

- Rappel: Introduction aux bases de données relationnelles
  - Définition et concepts fondamentaux
  - Modèle relationnel et langage SQL
- Rappel: Concepts clés de SQL
  - DDL (Data Definition Language): CREATE, ALTER, DROP
  - DML (Data Manipulation Language): SELECT, INSERT, UPDATE, DELETE
  - Clauses SQL : WHERE, ORDER BY, GROUP BY, HAVING, JOIN
- Rappel: Pratique de SQL
  - Environnement SQL et outils de travail
  - Création et manipulation de tables
  - Exécution de requêtes SQL

FIN.

100.000 Accès aux bases de données et ORM

Introduction à GORM

- Présentation de GORM
  - Qu'est-ce qu'un ORM (Object Relational Mapping) ?
  - Pourquoi utiliser GORM ?
  - Fonctionnalités clés de GORM
- Installation et configuration de GORM
  - Installation du package GORM
  - Configuration de GORM avec Go
- Les bases de GORM
  - Connexion à la base de données
  - Création, lecture, mise à jour et suppression d'enregistrements
  - Migration automatique de la structure de la base de données

FIN.

100.000 Accès aux bases de données et ORM

Utilisation de jinzhu/gorm pour gérer les bases de données

- Gestion des relations de bases de données avec GORM
  - Relations un-à-un, un-à-plusieurs, plusieurs-à-plusieurs
  - Association et préchargement
- Techniques avancées avec GORM
  - Jointures complexes et requêtes SQL brutes
  - Transactions
  - Callbacks et hooks
- Optimisation et débogage
  - Journalisation
  - Optimisation des requêtes

FIN.

100.000 Accès aux bases de données et ORM

Travaux pratiques

Créer une application de gestion d'utilisateurs avec GORM et une base de données SQLite

- Conception de la base de données
    - Modélisation des données
    - Création de la base de données avec GORM
- Développement de l'application
    - Connexion à la base de données
    - Gestion des utilisateurs (création, lecture, mise à jour, suppression)
- Tests et débogage
    - Écriture et exécution de tests unitaires
    - Identification et correction des problèmes

Implémenter un système de gestion de stock avec GORM et une base de données PostgreSQL

- Conception de la base de données
    - Modélisation des données
    - Création de la base de données avec GORM
- Développement de l'application
    - Connexion à la base de données
    - Gestion du stock (ajout, mise à jour, suppression, recherche)
- Tests et débogage
    - Écriture et exécution de tests unitaires
    - Identification et correction des problèmes

FIN.

110.000 Développement d'applications en ligne de commande

Arguments et flags

- Comprendre les arguments de la ligne de commande
- Utilisation du package flag pour parser les arguments
- Gestion des arguments optionnels et obligatoires

Introduction à spf13/cobra

- Présentation de Cobra et ses avantages
- Installation et configuration de Cobra
- Structure de base d'une application CLI avec Cobra

Utilisation de spf13/cobra pour créer des applications CLI

- Création de commandes, arguments et flags avec Cobra
- Gestion des sous-commandes
- Utilisation de Cobra pour lire la configuration de l'application

Travaux pratiques

Créer une application CLI pour interagir avec une API RESTful

- Planification de l'application CLI
- Création de l'application CLI avec Cobra
- Ajout des commandes pour interagir avec l'API RESTful
- Tests et débogage de l'application CLI

Développer un outil en ligne de commande pour analyser et traiter des fichiers CSV

- Comprendre les besoins et les spécifications

- Création de l'outil CLI avec Cobra
- Ajout des commandes pour lire, analyser et traiter les fichiers CSV
- Tests et optimisation de l'outil CLI

FIN.

110.000 Développement d'applications en ligne de commande

Arguments et flags

- Comprendre les arguments de la ligne de commande
- Utilisation du package flag pour parser les arguments
- Gestion des arguments optionnels et obligatoires

FIN.

110.000 Développement d'applications en ligne de commande

Introduction à spf13/cobra

- Présentation de Cobra et ses avantages
- Installation et configuration de Cobra
- Structure de base d'une application CLI avec Cobra

FIN.

110.000 Développement d'applications en ligne de commande

Utilisation de spf13/cobra pour créer des applications CLI

- Création de commandes, arguments et flags avec Cobra
- Gestion des sous-commandes
- Utilisation de Cobra pour lire la configuration de l'application

FIN.

110.000 Développement d'applications en ligne de commande

Travaux pratiques

Créer une application CLI pour interagir avec une API RESTful

- Planification de l'application CLI
- Création de l'application CLI avec Cobra
- Ajout des commandes pour interagir avec l'API RESTful
- Tests et débogage de l'application CLI

Développer un outil en ligne de commande pour analyser et traiter des fichiers CSV

- Comprendre les besoins et les spécifications
- Création de l'outil CLI avec Cobra
- Ajout des commandes pour lire, analyser et traiter les fichiers CSV
- Tests et optimisation de l'outil CLI

FIN.