

Опановування основами Go: Практичний посібник з освоєння мови Go

4.1 Масиви та Слайси



Привіт! Готовий досліджувати деякі структури даних в Go? Масиви та слайси - ідеальні місця, щоб почати. Вони дуже схожі, але слайси набагато більш гнучкі. Почнімо!


4.1.1 Масиви

Масиви - це фіксовані за розміром, впорядковані колекції елементів однакового типу. Ви оголошуєте масив так:

```
var myArray [5]int
```

Це створює масив під назвою "myArray" з 5 цілочисельних елементів. Елементи ініціалізуються їх нульовими значеннями (0 для цілих чисел). Ви також можете ініціалізувати масив з конкретними значеннями:

```
myArray := [5]int{10, 20, 30, 40, 50}
```

 Професійна порада: Ви можете використовувати `...` замість конкретного розміру, і Go визначить його на основі кількості елементів:

```
myArray := [...]int{10, 20, 30, 40, 50}
```

Для доступу або зміни елемента масиву використовуйте його індекс (починаючи з 0):

```
firstElement := myArray[0] // 10 myArray[1] = 25 // тепер myArray це [10, 25, 30, 40, 50]
```

4.1.2 Слайси

Слайси схожі на масиви, але вони мають динамічний розмір. Вони супер корисні, тому що вам не потрібно знати розмір завчасно. Ви можете створити слайс так:

```
var mySlice []int
```

Щоб створити слайс з елементами, ви можете використовувати вбудовану функцію `make()` або слайс-літерал:

```
mySlice1 := make([]int, 3) // Створює слайс з 3 елементами, ініціалізованими до 0  
mySlice2 := []int{10, 20, 30, 40, 50} // Створює слайс з 5 елементами
```

Слайси набагато потужніші, ніж масиви! Ви можете додавати елементи, видаляти елементи, а навіть створити підслайси. Подивіться на ці приклади:

```
mySlice = append(mySlice, 60) // Додає новий елемент: [10, 25, 30, 40, 50, 60]  
subSlice := mySlice[1:4] // Створює новий слайс з елементами від індекса 1 до 3: [25, 30, 40]
```

Слайси в Go - це динамічні масиви, і вони досить гнучкі та потужні.

1. **make**: Ця функція створює новий слайс з певною довжиною та ємністю. Ємність - це кількість елементів, які можуть бути збережені в масиві, що лежить в основі слайсу, починаючи з першого елемента слайсу.

```
mySlice1 := make([]int, 3, 5) // Створює слайс з 3 елементами,  
ініціалізованими до 0 та ємністю 5  
mySlice2 := []int{10, 20, 30, 40, 50} // Створює слайс з 5 елементами
```

Створення за допомогою функції **make** з вказанням ємності дозволяє оптимізувати додавання елементів слайсу без перевизначення "базового масиву" при перебільшенні місткості "базового масиву" слайсу.

2. **append**: Ця функція додає один або декілька нових елементів до слайсу та повертає слайс з новими елементами. Якщо в масиві, що лежить в основі слайсу, недостатньо місця для нових елементів, **append** автоматично алокує більший масив.

```
mySlice := []int{1, 2, 3}  
mySlice = append(mySlice, 4, 5)  
fmt.Println(mySlice) // Виведе: [1 2 3 4 5]
```

3. **copy**: Ця функція копіює елементи з одного слайсу в інший. Це дозволяє вам копіювати елементи з одного слайсу в інший, навіть якщо вони мають різний розмір.

```
mySlice2 := []int{10, 20, 30, 40, 50}  
newSlice := make([]int, 3)  
copy(newSlice, mySlice2)  
fmt.Println(newSlice) // [10 20 30]
```

4. **len**: Ця функція повертає поточну довжину слайсу.

```
mySlice := []int{1, 2, 3}  
fmt.Println(len(mySlice)) // Виведе: 3
```

5. **cap**: Ця функція повертає поточну ємність слайсу.

```
mySlice := make([]int, 3, 5)  
fmt.Println(cap(mySlice)) // Виведе: 5
```

6. **clear**: Ця функція всі елементи слайсу "обнуляє" (чистить).

```
mySlice2 := []int{10, 20, 30, 40, 50}
fmt.Println(mySlice2)
mySlice2 = append(mySlice2, 60)
fmt.Println(mySlice2)
subSlice := mySlice2[1:4] //sub slice: 20, 30, 40 (3 elements from
index 1 to 1+3)
fmt.Println(subSlice)
clear(subSlice)           // new in GO 1.21
fmt.Println(mySlice2) // [10 0 0 0 50 60]  ````
```

Функція **copy** у Go дуже корисна, коли тобі потрібно скопіювати елементи з одного слайсу в інший. Це особливо важливо, якщо ти хочеш переконатися, що зміни в одному слайсі не вплинуть на інший слайс, оскільки слайси в Go часто мають спільний масив під капотом.

Ось декілька ситуацій, коли функція **copy** може бути в нагоді:

- Якщо ти хочеш мати два незалежних слайси, зміни в одному з яких не впливають на інший.
- Коли ти працюєш із слайсами, які передаються в функції, і хочеш уникнути неочікуваних змін вихідних даних.
- Для оптимізації продуктивності, наприклад, коли ти хочеш скопіювати дані з великого слайсу в менший, щоб зменшити використання пам'яті.

Ось як можна використати **copy** на практиці:

```
// Створюємо слайс з деякими елементами
originalSlice := []string{"один", "два", "три"}

// Створюємо цільовий слайс, де будемо зберігати копії.
// Важливо, щоб він мав достатню довжину, щоб вмістити копійовані елементи!
copySlice := make([]string, len(originalSlice))

// Копіюємо елементи з оригінального слайсу в новий.
copiedElements := copy(copySlice, originalSlice)

// Тепер copySlice містить копію елементів з originalSlice,
// і зміни в одному слайсі не вплинуть на інший.
fmt.Printf("Скопійовано %d елементів: %v\n", copiedElements, copySlice)
```

У цьому прикладі ми копіюємо елементи з **originalSlice** в **copySlice**, і після цього ми можемо змінювати **copySlice**, не турбуючись про те, що це вплине на **originalSlice**. Функція **copy** повертає кількість елементів, що були скопійовані, що може бути корисно для перевірки успішності операції.

В останніх версіях Go, починаючи з 1.21 додалась функція **clear** яка дозволяє повністю очистити значення кожного елементу слайсу або масива.


```
package main

import "fmt"

func main() {
    //simple slices
    mySlice2 := []int{10, 20, 30, 40, 50}
    fmt.Println(mySlice2)
    mySlice2 = append(mySlice2, 60)
    fmt.Println(mySlice2)
    subSlice := mySlice2[1:4] //sub slice: 20, 30, 40 (3 elements from
index 1 to 1+3)
    fmt.Println(subSlice)
    clear(subSlice)          // new in GO 1.21
    fmt.Println(mySlice2) // [10 0 0 0 50 60]
}
```

4.2. Карти (map)

А що є навіть краще за срізи на Go? Це карти!

У Go мапи це такі собі неупорядковані колекції пар "ключ-значення". Це значить, що коли ти ітеруєш по мапі, то не можеш бути впевненим в порядку, в якому отримаєш елементи. Вони можуть приходити в абсолютно випадковому порядку, який може змінюватися при кожному запуску програми.

Отже, якщо тобі потрібно отримати елементи мапи в певному порядку (наприклад, відсортовані за ключами), то тобі доведеться вдатися до деяких хитрощів. Спершу ти можеш витягнути всі ключі мапи в звичайний слайс, відсортувати його, а потім використати відсортований слайс ключів для доступу до значень у мапі.

Давай покажу, як це можна зробити:

```
package main

import (
    "fmt"
    "sort"
)

type Recipe struct {
    DishName    string
    Ingredients []string
}

func main() {
    // У нас є мапа рецептів для кожної страви
    recipes := map[string]Recipe{
        "борщ": {
            DishName:    "Український борщ",
        },
    }
}
```

```

        Ingredients: []string{"буряк", "картопля", "капуста"},
    },
    "вареники": {
        DishName:      "Вареники з картоплею",
        Ingredients: []string{"тісто", "картопля", "цибуля"},
    },
    "голубці": {
        DishName:      "Голубці з м'ясом",
        Ingredients: []string{"капуста", "м'ясний фарш", "рис"},
    },
}

// Створюємо слайс для ключів нашої мапи
keys := make([]string, 0, len(recipes))

// Додаємо ключі в слайс
for key := range recipes {
    keys = append(keys, key)
}

// Сортуємо ключі
sort.Strings(keys)

// Тепер ітеруємо по нашому слайсу ключів та доступаємо до значень мапи
fmt.Println("Рецепти страв в алфавітному порядку:")
for _, key := range keys {
    recipe := recipes[key]
    fmt.Printf("Страва: %s, Інгредієнти: %v\n", recipe.DishName,
recipe.Ingredients)
}
}

```

4.2.1 Видалення Даних з Мапи

Коли прийде час попрощатися з деякими даними в мапі, Go має простий і зрозумілий спосіб видалення пар "ключ-значення". Для цього використовується вбудована функція `delete`. Ось як це працює:

```

// Припустимо, у нас є мапа з деякими рецептами страв.
// Тепер, якщо ми вирішили, що більше не хочемо готувати вареники,
// ми можемо легко видалити їх з нашої книги рецептів.
delete(recipes, "вареники")

// Якщо спробуємо доступитися до "вареників" після цього, вони вже не
будуть у мапі.
_, exists := recipes["вареники"]
if !exists {
    fmt.Println("Рецепт вареників було видалено.")
}

```

У Go для роботи з мапами доступна лише декілька вбудованих функцій, але вони вкрай корисні:

1. **make**: Ця функція використовується для створення мапи. Вона дозволяє визначити початкову вмістимість мапи, що може допомогти оптимізувати продуктивність, якщо ти заздалегідь знаєш кількість елементів, які будуть додані.

```
myMap := make(map[string]int) // Створюємо пусту мапу для зберігання цілих чисел з рядковими ключами
```

2. **delete**: Ця функція видаляє пару "ключ-значення" з мапи. Якщо ключ відсутній у мапі, функція нічого не робить.

```
delete(myMap, "ключ") // Видаляємо елемент з мапи за ключем "ключ"
```

3. Читання та запис у мапу: Хоча читання та запис не є окремими функціями, вони є основними операціями, які можна виконувати з мапами.

```
myMap["ключ"] = 42 // Записуємо значення у мапу  
value := myMap["ключ"] // Читаємо значення з мапи
```

4. Отримання значення та перевірка наявності ключа: У Go, коли ти читаєш значення з мапи, ти можеш також отримати булеве значення, яке показує, чи дійсно існує у мапі такий ключ.

```
value, ok := myMap["ключ"] // 'value' містить значення, 'ok' - булеве значення, яке показує, чи існує ключ
```

Це зване "comma ok" ідіома, і вона є дуже зручним способом перевірити, чи існує ключ у мапі, перш ніж використовувати його значення, щоб уникнути отримання нульових значень типу за замовчуванням, які можуть бути неочікуваними.

Ці вбудовані можливості роботи з мапами роблять їх дуже потужними і гнучкими інструментами для управління колекціями даних у Go.

В останніх версіях Go, починаючи з 1.21 додалась функція **clear** яка дозволяє повністю видалити ключі з мапи.

4.3 Структури

Структури в Go - це такі собі кастомні типи даних, які дозволяють групувати різні значення в один логічний набір. Вони як маленькі скарбнички, де ти можеш зберігати різноманітні дані разом, а потім легко ними управляти. Це як би створити собі власну маленьку базу даних прямо в коді.

Ось як можна оголосити структуру:

```
type Car struct {  
    Brand string  
    Model string  
    Year  int  
}
```

Тут ми маємо структуру `Car`, яка містить марку, модель і рік виробництва автомобіля.

А тепер давайте трошки пожартуємо і уявимо, що структура `Car` - це автомобіль для супергероїв:

```
type SuperCar struct {  
    Car  
    IsFlying bool  
}
```

Ми створили структуру `SuperCar`, яка вбудовує нашу базову структуру `Car` і додає ще одне поле: чи може автомобіль літати. Якщо ви хочете змінити реальність, просто встановіть `IsFlying` в `true`, і ваш автомобіль злетить! 😊

Але повернімося до реальності. Коли створюєш екземпляр структури, можеш вказати значення для її полів напряду:

```
mySuperCar := SuperCar{  
    Car: Car{  
        Brand: "BatMobile",  
        Model: "BatCave Edition",  
        Year:  2023,  
    },  
    IsFlying: true,  
}
```

Тепер у нас є `mySuperCar`, і якщо ми хочемо, щоб він злетів, просто переконайтесь, що `IsFlying` встановлено в `true`.

Зі структурами можеш робити багато речей: можеш передавати їх у функції, повертати з функцій, або навіть використовувати їх у масивах та слайсах. І якщо ти хочеш серіалізувати структуру в JSON чи інший формат, просто використовуй теги.

Структури в Go - це міцний каркас для твоїх даних, який допомагає тримати код чистим і організованим.

4.3.1 Теги структур

Теги структур це особливий синтаксис в Go, який дозволяє додавати метадані до полів структур. Ці метадані можуть бути використані різними бібліотеками, наприклад, для серіалізації/десеріалізації даних у формати JSON, XML чи для використання з ORM-бібліотеками.

Ось приклад структури з тегами для JSON серіалізації:

```
type Superhero struct {
    Name      string `json:"name"`
    Superpower string `json:"superpower,omitempty"`
    SecretIdentity string `json:"secretIdentity"`
}
```

В цьому прикладі, кожне поле структури **Superhero** має тег JSON. Ключ "json" в тегу вказує, що цей тег призначений для серіалізації в JSON.

- **name**: Це поле буде серіалізовано з ключем "name" в JSON.
- **superpower**: Це поле серіалізується тільки якщо воно не порожнє (**omitempty**).
- **secretIdentity**: Це поле завжди серіалізується з ключем "secretIdentity".

Якщо ви хочете виключити поле з серіалізації, можна просто не вказувати тег або зробити тег "-":

```
type Superhero struct {
    Name      string `json:"name"`
    Age       int    `json:"- "` // це поле не буде включено в серіалізований
JSON
}
```

Теги також можна використовувати з іншими бібліотеками, які підтримують інші формати, наприклад, XML або деякі ORM системи, які можуть використовувати теги для визначення того, як поля структури відображаються на колонки в базі даних.

Теги структур в Go це потужний інструмент для контролю того, як твої дані представлені при серіалізації, що робить їх надзвичайно корисними для роботи з JSON API, базами даних та іншими системами, які потребують форматування даних.

4.3.2 Вкладені Структури

У Go, структури можуть бути вкладені одна в одну, що дозволяє тобі моделювати складніші відносини та ієрархії. Це як матрьошки, де одна структура міститься в іншій, і ти можеш продовжувати вкладати їх глибше і глибше.

Припустимо, що у нас є структура **Engine**, яка відображає двигун автомобіля, і ми хочемо вкласти її в структуру **Car**.

```
type Engine struct {
    Power int    `json:"power"`
    Type  string `json:"type"`
}

type Car struct {
    Make  string `json:"make"`
```

```
Model string `json:"model"`  
Year int `json:"year"`  
Engine Engine `json:"engine"`  
}
```

Тепер, коли ми створюємо новий екземпляр `Car`, ми можемо одночасно ініціалізувати `Engine`:

```
myCar := Car{  
    Make: "Tesla",  
    Model: "Model S",  
    Year: 2020,  
    Engine: Engine{  
        Power: 500,  
        Type: "Electric",  
    },  
}
```

Але це не просто автомобіль, це суперкар! Він має секретні функції, які ми можемо використовувати для порятунку світу (або просто для швидкої поїздки до супермаркету). ☺

Коли ти працюєш з вкладеними структурами, ти можеш безпосередньо доступатися до полів вкладеної структури:

```
fmt.Println(myCar.Engine.Power) // Виведе потужність двигуна
```

Це надає чудову гнучкість, оскільки ти можеш легко структурувати дані з урахуванням логічних або функціональних відносин між компонентами твоєї програми.

Тепер, якщо хочеш, щоб твоя структура `Car` могла включати різні типи двигунів без жорсткого кодування поля `Engine`, ти можеш використовувати інтерфейси. Але це вже історія для іншого часу!

Вкладені структури роблять твої моделі даних в Go більш експресивними і модульними, дозволяючи тобі легко представляти складні дані та відносини між ними.

4.3.3 Наслідування Структур

Традиційне наслідування, як в інших об'єктно-орієнтованих мовах, у Go відсутнє. Натомість, Go використовує композицію через вбудовування структур. Це дозволяє структурам "наслідувати" поля або методи інших структур.

Давайте розглянемо це на прикладі:

```
package main  
  
import "fmt"
```

```
type Vehicle struct {
    Make string
    Model string
}

// Start є методом, який доступний всім структурам, що вбудовують Vehicle
func (v Vehicle) Start() {
    fmt.Println("Заводимо", v.Make, v.Model)
}

// Car "наслідує" Vehicle за допомогою вбудовування
type Car struct {
    Vehicle      // Вбудована структура Vehicle
    NumberOfDoors int
}

// Truck також "наслідує" Vehicle
type Truck struct {
    Vehicle      // Вбудована структура Vehicle
    LoadCapacity int
}

func main() {
    myCar := Car{
        Vehicle: Vehicle{
            Make: "Toyota",
            Model: "Corolla",
        },
        NumberOfDoors: 4,
    }

    myTruck := Truck{
        Vehicle: Vehicle{
            Make: "Ford",
            Model: "F-150",
        },
        LoadCapacity: 1000,
    }

    // Обидва, Car і Truck, можуть викликати метод Start()
    myCar.Start()
    myTruck.Start()
}
```

У цьому прикладі **Car** та **Truck** "наслідують" структуру **Vehicle** шляхом вбудовування. Це означає, що вони автоматично отримують доступ до її полів і методів, таких як **Make**, **Model** і **Start**.

Композиція через вбудовування структур є більш гнучкою, ніж традиційне наслідування, оскільки вона дозволяє створювати більш модульні та легко розширювані системи. Це допомагає уникнути деяких проблем, які можуть виникнути з традиційним наслідуванням, таких як крихка базова класова проблема (fragile base class problem) та проблеми з багаторівневим наслідуванням.

4.3.4 Сумісне Використання map, slice, struct та Пакета sort

Коли робота стає складною, Go має все, що потрібно, щоб впоратися з нею. Мапи, слайси та структури це неймовірно потужні інструменти, які можна комбінувати, щоб управляти складними даними. А коли справа доходить до сортування, пакет `sort` вступає в гру.

Давайте уявимо, що ми маємо каталог бібліотеки, де кожна книга описується структурою і зберігається в мапі з ключами за назвою. Але коли ми хочемо вивести список книг, ми хочемо, щоб він був відсортований за автором.

```
package main

import (
    "fmt"
    "sort"
)

type Book struct {
    Title  string
    Author string
    Year   int
}

// Функція для сортування слайсу книг за автором
type ByAuthor []Book

func (a ByAuthor) Len() int           { return len(a) }
func (a ByAuthor) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a ByAuthor) Less(i, j int) bool { return a[i].Author < a[j].Author }

func main() {
    // Створюємо мапу для книг
    library := make(map[string]Book)
    library["Go Pro"] = Book{Title: "Go Pro", Author: "John Doe", Year:
2020}
    library["Go Basics"] = Book{Title: "Go Basics", Author: "Jane Doe",
Year: 2021}
    // ... інші книги ...

    // Створюємо слайс для сортування книг
    books := make([]Book, 0, len(library))
    for _, book := range library {
        books = append(books, book)
    }

    // Сортиємо книги за автором
    sort.Sort(ByAuthor(books))

    // Виводимо відсортований список книг
    for _, book := range books {
        fmt.Printf("%s by %s (%d)\n", book.Title, book.Author, book.Year)
```

```
}  
}
```

У цьому прикладі, ми створили структуру `Book`, мапу `library` для зберігання книг і слайс `books` для сортування. Ми визначили тип `ByAuthor`, що імплементує інтерфейс `sort.Interface`, і використовуємо функцію `sort.Sort` для сортування книг за автором.

Сортування за іншими полями також просте, як і зміна методу `Less` у типу, що імплементує `sort.Interface`. Таким чином, ми можемо легко керувати та відображати наші дані в будь-якому порядку, який нам потрібен. Go робить складні речі простішими, а сортування це просто ще один приклад цього.

4.4.5 Функції `sort.Slice` та `sort.Strings`

Коли мова заходить про сортування в Go, пакет `sort` є твоїм найкращим другом. І у нього є кілька трюків у рукаві! Дві зірки шоу це `sort.Slice` та `sort.Strings`.

`sort.Slice`

Немає потреби імплементувати цілу купу методів, коли можна просто використати `sort.Slice`! Ця функція приймає слайс і функцію `less`, яка визначає, як сортувати елементи. Ось як ти можеш її використати:

```
// Маємо слайс структур книг  
books := []Book{  
    {Title: "The Go Programming Language", Author: "Alan A. A. Donovan",  
    Year: 2015},  
    {Title: "Go in Action", Author: "William Kennedy", Year: 2015},  
    // ... інші книги ...  
}  
  
// Використовуємо sort.Slice для сортування книг за роком видання  
sort.Slice(books, func(i, j int) bool {  
    return books[i].Year < books[j].Year  
})  
  
// Тепер books відсортовано!
```

`sort.Strings`

Але що, якщо у тебе просто масив рядків, який потрібно відсортувати? `sort.Strings` це твій го-то метод. Він приймає слайс рядків і повертає його, відсортований у зростаючому порядку. Якось так:

```
// Маємо слайс назв книг  
titles := []string{"Concurrent Programming in Go", "Introducing Go",  
"Learning Go"}
```



```
// Сортуємо назви
sort.Strings(titles)

// Бам! Відсортовані назви, як по магії.
```

Ці дві функції роблять сортування слайсів у Go легким.

4.4 Власні типи даних та псевдоніми типів

Go дозволяє тобі визначати власні типи даних та псевдоніми типів, що може бути дуже корисно для створення читабельнішого та безпечнішого коду.

Власні типи даних

Власний тип даних це по суті новий тип, який базується на одному з уже існуючих типів. Він створюється для конкретної мети та може мати власні методи.

```
type Celsius float64
type Fahrenheit float64

func (c Celsius) ToFahrenheit() Fahrenheit {
    return Fahrenheit((c * 9/5) + 32)
}

func (f Fahrenheit) ToCelsius() Celsius {
    return Celsius((f - 32) * 5/9)
}

var temperature Celsius = 36.6
fmt.Println("Температура в Фаренгейтах:", temperature.ToFahrenheit())
```

У цьому прикладі `Celsius` і `Fahrenheit` є власними типами даних, кожен з яких має метод для перетворення один в інший.

Псевдоніми типів

Псевдонім типу не створює новий тип; він просто дає інше ім'я існуючому типу. Це може бути корисним для спрощення складних типів або для перейменування типів для кращої читабельності.

```
type ID = string

var userID ID = "abc123"
```

Тут `ID` є псевдонімом для типу `string`, і тепер ми можемо використовувати `ID` у нашому коді, що робить його більш зрозумілим у контексті, що це ідентифікатор користувача.

Порівняння власних типів та псевдонімів

Головна відмінність між власними типами даних та псевдонімами полягає в тому, що власні типи даних вводять новий, незалежний тип, в той час як псевдоніми просто дають нове ім'я існуючому типу. Власні типи можуть мати методи і можуть бути використані для створення чіткого інтерфейсу, а псевдоніми корисні для спрощення та покращення читабельності коду.

Використання власних типів та псевдонімів дозволяє тобі більш точно висловлювати інтенції твого коду, забезпечуючи типову безпеку та чистоту інтерфейсів у твоїх програмах на Go.