BaseFunctionsGo_UA

Вступ

У цьому гайді ви зможете отримати основні пізнання по Go. Причиною написання гайда була складність і заплутаність, яку я помітив в інших гайдах і керівництвах. Безумовно, я не експерт у Go, але при цьому маю якийсь досвід і думаю, що можу допомогти цими знаннями іншим :)

Корисні посилання

Так як цей гайд досить суб'єктивний я не впевнений, що захоплю всі теми і не впевнений, що у нас не з'являтися питання після прочитання, тому в цьому блоці прикріплю хороші посилання який можна використовувати в таких випадках.

Насамперед ви можете написати мені на пошти і я спробую вам допомогти:

boyarkin.gleb@gmail.com

source.boar@gmail.com

Також пошти продубльовані у мене в профілі.

Крім мене ви можете скористатися:

https://metanit.com/go/tutorial

Ці джерела гарантую, що є перевіреними, сам по них навчався, але як на мене багато речей там розписані надто складними словами.

Коротко про Go

Go – дуже молода статична мова програмування. Створений він був для того щоб об'єднати в собі простоту і швидкість, що в принципі він зробив. Галузями використання найчастіше є малі програми, мікросервіси та іноді Web.

Синтаксис

Синтаксис Go здебільшого є C-подібним, але зі своїми дрібними відмінностями, з якими ми зіштовхуватимемося по ходу.

Якщо проходитись за основними правилами синтаксису, то:

- символу для закінчення рядка немає
- `{}` – сиволи для відкриття та закриття блоків коду
- `camelCase` – стиль іменування змінний
- `PascalCase` – стиль іменування функцій, методів та структур
- `//` – символ для коментаря одного рядка
- `/* */` – символи для коментування блоку коду

Структура програми

Структура програми Go складається з двох обов'язкових речей:

1. Оголошення імені пакету

```
package /*ім'я пакета*/
```

Якщо ваша програма не є бібліотекою, то пакет оголошується як `main`

```
package main
```

2. Функція `main` - функція входу до програми

```
func main() {

}
```

Робота з консоллю

Для роботи з консоллю використовується модуль `fmt`

`fmt.Print(/*текст повідомлення*/)` – виведення тексту в консоль

`fmt.Println(/*текст повідомлення*/)` – виведення тексту в консоль з додаванням перенесення на новий рядок

`fmt.Printf(/*маска*/, /*змінна 1*/, /*змінна 2*/, ... /*змінна n*/)` – форматований виведення тексту в консоль

`fmt.Scan(/*посилання на змінну куди записати результат*/)` – введення тексту з консолі

Змінні

Як я думаю вам відомо, чи стане відомо зараз, за фактом вся мова програмування тримається на змінних та операціях над ними.

У Go змінна - це покажчик на область або осередок пам'яті в якій зберігається якесь зниження. Факт про пам'ять поки що можна просто покласти в голову, до нього ми дійдемо потім.

Також Go є статичним строго і явно типізованим мовою, тому при оголошенні змінних потрібно так само передавати тип або відразу передавати значення тоді тип буде визначено автоматично.

Для оголошення змінних використовуються ключове слово 'var'.

Приклад оголошення:

```
var a1 string = "Hello"
var a2 = 1
```

Так само в Go присутні константи. Константи – звичайні змінні, крім чого можуть бути змінені у майбутньому після оголошення. Вказівка типу константи не є обов'язковим.

Для оголошення констант використовуються ключове слово 'const'.

Приклад оголошення:

```
const a1 string = "Hello"
const a2 = 1
```

Також змінні можна оголошувати коротко: без використання ключового слова `var`. Натомість потрібно просто використовувати символ `:=`.

Приклад оголошення:

```
a1 := "Hello"
```

При іменуванні змінних, як було зазначено вище, використовується стиль camelCase.

Що ж до нього входить?

camelCase:

- перша літера – нижнього регістру
- кожна перша літера наступного слова – верхнього регістру

Також варто уточнити що Go чутливий до регістру, тобто в ньому `a` та `A` це різні імена.

Типи даних

** Тип даних ** — типи того, якого типу значення може зберігати змінна.

Вбудовані типи даних у Go:

- `bool` – логічний тип даних, що зберігає 2 види значень: `true` або `false`
- `int8/int16/int32/int64` – числові типи даних, які можуть зберігати цілі числа розмірністю 8, 16, 32 та 64 біти відповідно
- `int` – числовий тип даних який залежно від платформи зберігає ціле число розмірністю 32 або 64 біти, тобто еквівалентно `int32` або `int64`
- `uint8/uint16/uint32/uint64` – числові типи даних, які можуть зберігати цілі позитивні числа розмірністю 8, 16, 32 та 64 біти відповідно
- `uint` – числовий тип даних який залежно від платформи зберігає ціле позитивне число розмірністю 32 або 64 біти, тобто еквівалентно `uint32` або `uint64`
- `byte` - синонім типу `uint8`
- `rune` - синонім типу `int32`
- `float32/float64` – числові типи даних, які можуть зберігати дробові числа розмірністю 32 та 64 біти відповідно

- `complex64/complex128` – числові типи даних, які можуть зберігати комплексні числа з основами `float32` та `float64` відповідно
- `string` – малий тип даних, що зберігає рядок тексту майже необмеженого розміру
- `nil` – ключове слово, яке означає що нульове або "порожнє" значення

Вказівники

Покажчики є об'єктами, значенням яких є адреси інших об'єктів (наприклад, змінних). Покажчик визначається як звичайна змінна, лише перед типом даних ставиться символ зірочки `*`

Приклад вказівника на тип int:

```
var a *int
```

Для отримання адреси вказівника чи будь-якої змінної використовується символ `&`

Оператори

Оператори привласнення

`=` – привласнення

`+=` – привласнення із додаванням

`-=` – привласнення з відніманням

`*=` – привласнення з множенням

`/=` – привласнення з розподілом

`%=` – привласнення з поділом із залишком

`++` – інкремент, теж саме що `+= 1`

`--` – декремент, теж саме що `-= 1`

`<<=` – привласнення з лівим зрушенням

`>>=` – привласнення з правим зрушенням

`&=` – привласнення з побітовим `І`

`|=` – привласнення з побітовим `АБО`

`^=` – привласнення з побітовим виключаючим `АБО`

Логічні оператори

`==` – рівність

`!=` – нерівність

`>` – більше

`>=` – більше чи одно

`<` – менше

`<=` – менше або одно

`&&` – логічне `І`, тобто обидві умови в конструкції `(умова 1)&&(умова 2)` повинні бути `true`

`||` – логічне `АБО`, тобто хоча б одна умова в конструкції `(умова 1)||(умова 2)` треба бути `true`

Арифметичні оператори

`+` – плюс

`-` – мінус

`*` – множення

`/` – розподіл

`%` – розподіл із залишком

Побітові оператори

`&` – побитове `І`

`|` – побітове `АБО`

`^` – побітове що виключає `АБО`

`~` – побитове `НЕ`

`<<` – зрушення вліво

`>>` – зсув праворуч

Умовні конструкції

Змінні це звичайно добре, але ж треба якось з ними працювати і ось умовні конструкції це один із способів прописування логіки програми.

if...else

Найвідоміша і найчастіше використовувана конструкція.

У `if` передається умова, якщо умова істинна то виконується те, що в блоці `if` в іншому випадку те, що в блоці `else`

Синтаксис виглядає приблизно так:

```
if /*умова: або логічний вираз або змінна типу bool*/ {
    // дія якщо умова == true
} else {
    // дія якщо умова == false
}
```

Також в цю конструкцію можна додати `else if`

Приклад застосування:

```
if /*умова 1: або логічний вираз або змінна типу bool*/ {
    // дія якщо умова 1 == true
} else if /*умова 2: або логічний вираз або змінна типу bool*/ {
    // дія якщо умова 1 == false, але умова 2 == true
} else {
    // дія якщо умова 1 == false та умова 2 == false
}
```

Таких конструкцій `else if` можна додавати будь-яку кількість разів

Синтаксис має бути саме таким, інші варіанти розташування дужок є неправильним та не сприймається компілятором

switch..case...default

Також умовна конструкція яка вкорочує код який можна написати через `if...else if...else`

`switch...case...default` відрізняється тим, що ми в switch передаємо якусь змінну, а потім конструкція порівнює цю змінну з кожним значенням `case` і якщо ніякого збігу з `case` не знайшлося виконуватись блок `default`

Синтаксис виглядає приблизно так:

```
switch /*змінна яку порівнюємо*/ {
    case /*значення 1*/:
        // дія якщо збіглася значення 1
    case /*значення 2*/:
        // дія якщо збіглася значення 2
    ...
    default:
        // дія якщо жодне значення не співпало
}
```

Цикли

for

Вид циклу з дією, що виконується до початку обходу, умовою виконання та дією, яка виконується в кінці кожного проходу

Синтаксис виглядає приблизно так:

```
for /*дія на початку*/; /*умова виконання*/; /*дія наприкінці кожного проходу*/ {
    // дія поки що умова true
}
```

Якщо дії на початку і в кінці не потрібні, то можна використовувати конструкцію тільки з умовою

```
for /*умова виконання*/ {
    // дія поки що умова true
}
```

Ключові слова `break` та `continue`

`break` використовується для дострокового виходу з циклу, тобто завершення роботи циклу незалежно від умови

Приклад використання:

```
for i: = 0; i < 10; i++ {
    if i%3 == 0 { // якщо число ділиться на 3 без залишку
        break
    }
    fmt.Println(i)
}
```

Тобто цикл виконувався доки не наткнувся на перше число, яке ділиться на 3

`continue` використовується для дострокового припинення проходу циклу і переходу на наступний прохід, тобто використовується коли вам треба, щоб все після його виклику не виконувалося в цьому проході, а виконувалося відразу зробляє прохід

```
for i: = 0; i < 10; i++ {
    if i%3 == 0 { // якщо число ділиться на 3 без залишку
        continue
    }
    fmt.Println(i)
}
```

Тобто для чисел які ділитися на 3 не виконувався `fmt.Println(i)`, а одразу виконувався наступний прогін циклу

Практичні завдання на даних знаннях

Для кращого закріплення пропоную виконати кілька простих завдань, які перевірять, наскільки ви зрозуміли цей рівень матеріалу, всі знання, які можуть бути потрібні для виконання завдань, є вище.

Всі розв'язки задач будуть у папці "Завдання" і будуть пронумеровані, але не рекомендую вам ними користуватися доки у вас самих не вийде

1. Напишіть цикл який виведе в консоль усі числа від 1 до 100 включно які поділяються на 5 або на 7 (`n%/*число*/ == 0` – умова подільності `n` на якесь число)
2. Напишіть цикл, який виведе в консоль всі парні числа від 1 до 100 включно
3. Напишіть цикл, який виведе в консоль всі числа від 1 до 100 включно які містять 1

Фукнції

Функції – ще один важливий інструмент із програмування. Якщо простими словами це об'єднання якоїсь ділянки коду для його подальшого виклику. У Go функції це об'єкт і з ними можна працювати як з об'єктом, тобто записувати в змінні, передавати аргумент, писати методи і так далі.

Кожна функція у будь-якій мові програмування має:

- Ім'я
- Список аргументів (параметрів) – змінні, значення яких задається під час виклику функції. Оголошуються вони в `()` через кому. Також при виклику їх значення вказуються при виклику так само в `()`. У блоці коду нашої функції вони відіграють роль звичайних змінних з блоковою областю видимості, тільки не потребують оголошення, тому що оголошуються при виклику функції
- Блок коду, який буде виконуватися при викликі. Записується в `{}`

Синтаксис виглядає приблизно так:

```
func /*ім'я функції*/(/*аргумент1*/, /*аргумент2*/.../*аргументN*/) /*тип значення, що повертається (якщо потрібен)*/{
    // блок коду, що виконується
}
```

`func` – ключове слово, яке означає оголошення функції

Для виклику функції достатньо написати її ім'я та `()` у яких можна перерахувати аргументи якщо вони були оголошені

Приклад функції:

```
func f(a1 int, a2 int, a3 int) {
    fmt.Println(a1)
    fmt.Println(a2)
    fmt.Println(a3)
}

f(1, 2, 3)
```

Фукнції та процедури

Якщо відходити від теми Go, де синтаксичної різниці між цими поняттями за фактом немає, це дуже важливе питання.

У чому ж різниця?

Функція — блок коду, який в результаті свого виконання поверне якесь закінчення.

Процедура — функція, яка не повертає значення

За повернення значення Go відповідає ключове слово `return`

Давайте розглянемо різницю на прикладах:

```
func fnc(a1 int, a2 int, a3 int, a4 int) int { // функція
    fmt.Println(a1)
    fmt.Println(a2)
    fmt.Println(a3)
    return a4
}

func proc(b1 int, b2 int, b3 int, b4 int) { // процедура
    fmt.Println(b1)
    fmt.Println(b2)
    fmt.Println(b3)
}

a := fnc(1, 2, 3, 4) // == 4(a4)
b := proc(1, 2, 3, 4) // == помилка
```

Що ж сталося?

При оголошенні **функції** ми використовували `return` тому виклик функції можна буде привласнити змінної і він буде мати значення яке ми укзали в `return`

Упорядковані типи даних

Змінні - це звичайно добре, але іноді доводиться зберігати багато будь-яких даних об'єднавши їх в якусь одну структуру. Для таких речей придумані масиви та словники.

Масиви

Массивы — впорядкований набір значень, які можна отримати, звертаючись за індексом.

Масив у Go це ділянка пам'яті з фіксованою довжиною.

Так само розглянемо спосіб його оголошення:

```
var a1[5]int = [5]int{1, 2, 3, 4, 5}
var a2 = [5]int{1, 2, 3, 4, 5}
var a3 := [5]int{1, 2, 3, 4, 5}
```

Для кращого розуміння скористаємося вище картинкою. Як видно на ній масив - це грубо кажучи набір "осередків" зі значеннями до яких паралельно прикріплені індекси, за якими можна отримати значення, якому

вони відповідають.

Для отримання значення або привласнення за індексом достатньо після імені масиву в [[]' вказати індекс

```
a := arr[0] // отримання
arr[0] = 1 // привласнення
```

!!!Індесація в мовах програмування починається з 0!!!

Зрізи

Зрізи це послідовність елементів одного типу змінної довжини. На відміну від масивів, довжина в зрізах не фіксована і динамічно може змінюватися, тобто можна додавати нові елементи або видаляти вже існуючі.

Так само розглянемо спосіб його оголошення:

```
var a1[]int = []int{1, 2, 3, 4, 5}
var a2 = []int{1, 2, 3, 4, 5}
var a3 := []int{1, 2, 3, 4, 5}
```

Додавання елементів

Для додавання елементів використовується функція `append(/*ім'я масиву*/, /*елемент додавання*/)`

Взяття зрізу

Для взяття зрізу використовується конструкція `/*ім'я масиву*/[/*початковий індекс*/:/*кінцевий індекс*/]`

Довжина зрізу

Для отримання довжини масиву використовується функція `len(/*ім'я масиву*/)`

Перебір масиву/зрізу

for

```
for i: = 0; i < /*довжина масиву/зрізу*/; i++ {
    // Елемент масиву/зрізу arr[i]
}
```

for...range

```
for i, a := range arr {
    // Елемент масиву/зрізу arr[i]
    // Елемент масиву/зрізу a
}
```

Відображення

Іноді виникають ситуації, коли треба, щоб індексами були не цифри, а щось своє. Для такого було створено відображення.

Відображення – це масив із самозазначеними індексами.

Синтаксис оголошення:

```
mp := map[/*тип ключів*/]/*тип значень*/ {
    "/*назва індексу 1*/": /*занчення 1*/,
    "/*назва індексу 2*/": /*занчення 2*/,
    ...
    "/*назва індексу N*/": /*занчення N*/}
```

Практичні завдання на даних знаннях

4. Придумайте як перевернути рядок у зворотному порядку
5. Знайдіть спосіб рядок `abcdef'` перетворити на cdefgh"`
6. Напишіть функцію яка повертатиме рядок створений за правилом `camelCase` (розділювач повинен передаватися другим аргументом)

   Приклад:

   `hello world` -> `helloWorld`
7. Створіть функцію для заповнення масиву відображень, які містять інформацію про людину

   Структура:

   ```
   people := []map[string]string{} // основний масив
   /* структура словника
   {
       "firstName": "",
       "secondName": "",
       "age": "",
       "country": "",
       "city": ""
   }
   */
   ```
8. *Придумайте як можна відсортувати масив

"*" - складна задача

Структури, методи та інтерфейси

Структури

Структури – типи даних, створені користувачем уявлення якихось нестандартних об'єктів

Для оголошення структур використовуються ключові слова `type` та `struct`

Синтаксис виглядає приблизно так:

```
type /*ім'я структури*/ struct {
    /*поля структури*/
}
```

Кожне поле має назву та тип даних, як змінна

Ініціалізація структур може виглядати приблизно так:

```
var s /*ім'я структури*/ = /*ім'я структури*/{/*значення, що передаються полям*/}
```

Звернення до полів структури виконується через `.`

Методи

Методи – функції, пов'язані з певними типами (структурами)

Синтаксис виглядає приблизно так:

```
func (/*ім'я змінної яка означатиме тип*/ /*ім'я типу*/) /*ім'я
методу*/(/*аргумент1*/, /*аргумент2*/.../*аргументN*/) /*тип повертається значення
(якщо потрібний) * / {
    // блок коду, що виконується
}
```

Виклик методів виконується як одержання полів структури, через `.`, єдине що додаються дужки з аргументами

Інтерфейси

Інтерфейси представляють абстракцію поведінки інших типів. Інтерфейси дозволяють визначати функції, які прив'язані до конкретної реалізації. Тобто інтерфейси визначають певний функціонал, але не реалізують його

Синтаксис виглядає приблизно так:

```
type /*ім'я інтерфейсу*/ interface {
    /*методи інтерфейсу*/
}
```

У Go присутній Качина типізація ("Все, що крякає - все качка"), через що інтерфейси не вимагають ініціалізації. Всі об'єкти структур з методами, які відповідають методам інтерфейсу, автоматично є об'єктами цього інтерфейсу.

Практичні завдання на даних знаннях

9. Напишіть структуру людини де буде зберігатися її ім'я, вік, країна та місто
10. Реалізуйте методи цієї структури з використанням інтерфейсу

Усе!

Здається це все! Я дав вам все, що хотів. Сподіваюся: цей гайд допоміг вам.

Так само у мене є гайд складніший з складнішими темами, але він більш просто збірник рішень, але кому цікаво можете почитати: https://github.com/s0urcedev/AdditionalFunctions

Всім дякую! Сподіваюсь ще побачимось :)