Golang - wstęp

Adam Pietrzak

25 listopada 2020

Spis treści

Interface

Interface

Interface jest zbiorem definicji funkcji, a dokładniej ich wyglądu.

Przykładowy wygląd interface:

```
type MyInterface interface {
    MyMethod(v int) int64
}

type A struct {}

func (a A) MyMethod (v int) int64 {
    return 2
}

var a MyInterface = A{}
func b () {
    t := a.MyMethod(12)
}
```

Interface II

Interface jest spełniony wtedy, kiedy wszystkie metody występują w danym typie. Nie istnieje dodatkowy zapis mówiący, że dany typ implementuje interace.

Interface pod spodem jest zapisany jako “typ” oraz “wartość”. Jest to bardzo ważne przy używanie interface.

Wartość interface może być “nil”:

```
var a MyInterface = nil
```

Interface III

"nil" jako wartość interface jest ważna. Po pierwsze, można na niej uruchomić metodę!

```
vat t *MyType
var a MyInterface = t
a.MyMethod()
```

Nie jest to błąd w GoLang. W takiej sytuacji dostaniemy jako "receiver" nil:

```
func (m *MyType) MyMethod() {
    if m == nil {

    }
    // ...
}
```

Interface IV

Wywołanie interface, który nie ma typu (jest "nil"), już pokaże błąd:

```
var a MyInterface = nil
a.MyMethod()
```

Dodatkowo, trzymanie typu w interface powoduje problemy z porównaniami:

```
type MyError struct {}

func (m MyError) Error () string {
    return ""
}
func test () error {
    var e *MyError = nil
    return e
}
func b () {
    if test() == nil {
    }
}
```

Interface V

Istnieje w GoLangu typ: “any”. Jest nim zapis:

```
var a interface{}
```

Ponieważ wszystko implementuje pusty interface, wszystko pasuje do tego zapisu.

Zapis ten powoduje problemy z typowaniem, ponieważ wszystko tutaj pasuje.

Dodatkowo, możemy mapować interface na ich typ wewnętrzny:

```
var a interface{} = 23
c, ok := a.(int)
```

Uwaga
Typy muszą się zgadzać 1:1.

ok
“ok” jest opcjonalne.

Interface VI

Istnieje dodatkowy zapis w "switch" ułatwiający mapowanie typów, gdy funkcja może przyjąć więcej niż jeden typ:

```
var a interface{} = 23

switch a.(type) {
    case int:
        // Code
    case int64:
        // Code
}
```

Uwaga!
Interface nie są dla tworzącego tylko dla używającego. Jeżeli nie ma potrzeby tworzenia interface, nie należy udostępniać interface!

Własne typy

Własne typy

Golang pozwala tworzy własne typy. Daje to nam możliwość większej walidacji typów (typy muszą się zgadzać), oraz możliwości rozszerzenia typów:

```
type A string
func (a A) Test () {

}

type B map[string]string
```

Własne typy możemy rozszerzać o metody.

W łatwy sposób, możemy rzutować typy:

```
var a A  = "A"
var b string = string(a)
var c A = A(b)
```

GoRoutines

GoRoutines

"GoRoutines" jest sposobem na uruchamianie części kodu w innych wątkach. Dzięki temu, w łatwy sposób można wykonywać operacje równolegle.

```
func a () {
    // Long running function
}

func b () {
    // Opetations
    go a()
    // operations
}
```

Funkcje te są uruchamiane w tej samej pamięci, więc należy synchronizować dostępy!

Channel

"channel" jest sposobem na komunikację z "goroutine". Dzięki temu, możemy synchronizować wykonanie zadań, jak i również odbierać dane.

"channel" tworzymy za pomocą funkcji "make". Drugi parametr robi z channela buffor.

```
func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // send sum to c
}
func main() {
    s := []int{7, 2, 8, -9, 4, 0}
    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c // receive from c
    fmt.Println(x, y, x+y)
}
```

Channel II

"buffered channel" może zawierać w sobie więcej niż jedną wartość do odebrania. Ponieważ dopóki channel jest pełen, to wpisanie wartości blokuje wątek.

```
func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Channel III

Channele można używać z funkcją “range”, “len”, “cap”. Dodatkowo, można je zamykać:

```
func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

Channel IV

Istnieje jeszcze funkcja select, która wywołuje kod w zależności który channel jest do obsługi:

```
func main() {
    tick := time.Tick(100 * time.Millisecond)
    boom := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")
            return
        default:
            time.Sleep(50 * time.Millisecond)
        }
    }
}
```

"default" uruchamia się w sytuacji, gdy nie ma nic do obsługi.

Gdy kilka channeli jest gotowych do odebrania, "select" wybierze losowo.