Golang - wstęp

Adam Pietrzak

18 listopada 2020

Spis treści

GoLang

Wstęp

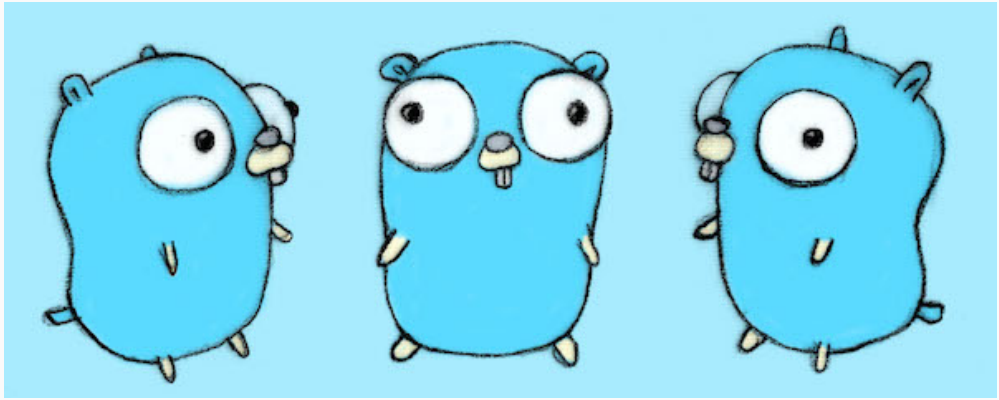
GoLang (właściwie Go) - język programowania opracowany w 2009 na potrzeby Google.

Głównymi elementami języka są:

- statyczne typowanie - wszystkie wartości mają określony typ
- silne typowanie - brak możliwości niejawnej zmiany typu
- kompilacja do jednego pliku wykonywalnego - zawsze powstaje jeden plik wykonywalny

Wstęp - maskotka

Maskotką i logiem języka Go jest "Go Gopher".



Składnia

Składnia

Składnia Go jest podobna do języka C i Pythona (czasami).

Każdy program składa się z funkcji main, lecz plik ją zawierający może być różnie nazwany.

Program jest dzielony na paczki, gdzie każda paczka może zawierać więcej niż jeden plik źródłowy. Ważnym jest, że importując paczkę, importujemy wszystkie pliki. Dzieje się tak, dlatego że paczka jest kompilowana jako jedno.

Dodatkowo, istnieją moduły. Moduły zawierają zbiór paczek ze sobą powiązanych, oraz są wdrażane razem. Każde repozytorium zawiera przynajmniej jeden moduł!

```
package main

import "fmt"

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

Składnia - pierwszy program

```go
// Package information
package main

// Imports
import "fmt"

// Main function
func main() {
    fmt.Println("Hello, world.")
}
```

Nazewnictwo
W GoLangu nazywanie funkcji, zmiennych itp. zależy od ich użycia.

Warto pamiętać, że jeżeli nazwa zawiera skrót, to nie mieszamy wielkich liter z małymi w tym skrócie: “xmlValue”, “XMLToSave”, “makeXMLHTTPValue”, “xmlHTTPRequest”.

To samo dotyczy “ID”.

Prywatność i publiczność

GoLang trzyma się jednej zasady, wszystko, co jest pisane z wielkiej litery, jest publiczne, wszystko co z małej litery, jest prywatne.

Dotyczy to funkcji definiowanych na poziomie modułu, czy też pól w obiektach.

Przykładowo, będąc w module "database", zdefiniowałem funkcję:

```
func add () {
    // implementation
}
```

W ramach modułu database, wszystkie pliki widzą i mogą używać tej funkcji. Jeżeli ktoś wywoła tą funkcję spoza modułu database, pojawi się błąd kompilacji.

Ciekawostka
Funkcja main jest pisana z małej litery.

Struktura plików

Go zmienia podejście do struktury plików. Pisząc kod w Javie, C++, JS zazwyczaj pojawia się katalog "src" zawierający kod aplikacji.

Według GO, w katalogu głównym posiadamy plik go.mod, informujący co to jest za moduł, a w podkatalogach różne paczki. W ramach modułu paczki mogą się importować, dzięki czemu mamy do nich dostęp.

Circular dependency
GoLang nie posiada możliwości rozwiązania circular dependency, więc należy uważać na importy.

Go modules

Oryginalnie, GoLang nie pozwalał na tworzenie kodu poza zdefiniowaną zmienną środowiskową “GOPATH”. Czyli wszystkie programy, aplikacje, biblioteki musiały znajdować się w tej ścieżce.

Na szczęście, od GO 1.14 pojawiły się “Production ready” Go Modules, które pozwalają na tworzenie aplikacji w dowolnym miejscu na dysku.

< 1.14
Go Modules istniały już od 1.11, lecz zmieniały się w czasie.

Język

Zmienne

Go posiada dwa sposoby definiowania zmiennych:

Pierwszym z nich jest słówko kluczowe “var”.

```
var i int = 0
```

Dzięki takiemu zapisowi, możemy zdefiniować i zainicjalizować zmienną.
Dużym plusem jest tutaj możliwość samej definicji, bez inicjalizacji:

```
var i int
```

Każda zmienna bez inicjalizacji posiada wartość domyślną (“zero value”):

- 0 - dla liczb
- “” - dla łańcucha znaków
- false - dla typów logicznych

Drugim plusem tej definicji jest jasny typ zmiennej.

Stałe
Aby zdefiniować stałą, używamy słowa “const”.

Zmienne

Drugim zapisem jest podstawienie ":=". Dzięki niemu, nie musimy używać słowa kluczowego "var".

```
i := 0  // OK
```

Podstawienie ma kilka wad:

- jeżeli zmienna istnieje, mamy błąd,
- brak jasnego typu,
- działa tylko w ciele funkcji

```
i := 0  // error

func main () {
    i := 0 // ok

    i = 2 // ok

    i := 1 // error
    var test uint
    test = i // error, can not assign int -> uint
}
```

Typowanie i rzutowania

Go jest silnie typowane. Nie każdy język statyczny jest silnie typowany, przykładem jest C, gdzie poprzez wskaźnik można zmienić typ dowolnej zmiennej.

Go jest inne, tutaj typ jest ostateczny. Nadanie jakiegoś typu zmiennej spowoduje, że żaden inny typ nie będzie to niej pasować.

Przykładowo:

```
var i int = 0
var j int32 = 1
var k int64 = 2
var l string = ""

i = l // error
l = i // error
j = k // error
k = j // error
i = j // error
```

Własne typy
Dotyczy to także typów własnych!

If

Ify są bardzo podobne do innych języków. Ich składnia nie wymaga nawiasów. Przykład:

```
if a == 2 {
    return "OK"
}
```

Jednak, GoLang dodaje możliwość łączenia przypisania z warunkiem, co nam daje zapis:

```
if a := 2; a == 2 {
    return "OK"
}
// ...
if item, err := getItem(id); err == nil {
    return "OK"
} else {
    log.Panic(err)
}
```

Łączenie w praktyce
Zapis łączenia jest często stosowany w Go, dlatego należy go rozumieć.

Pętla for

Pętla for mocno nie różni się od innych języków:

```
for i := 0; i < 10; i++ {
    // Code
}
// Part of for can be skip and it goes to while
for value < 1000 {
    value += getValue()
}
```

Dodatkowo, można iterować za pomocą słowa kluczowego “range”:

```
for index, value := range myArray {
    // ...
}
```

Switch

Pomimo podobnego zapisu i z pozoru podobnego działania, switch w GoLangu daje dużo więcej możliwości:

```
switch value {
    case 20:
        log.Print("20")
    case 40:
        log.Print("40")
    default:
        log.Print("Hmmm, not expected")
}
```

Do tego, można zrobić zapis:

```
switch value := getValue(); value {
    // ...
}
```

Switch II

Oczywiście, możemy nawet pominąć warunek:

```
value := 20 + rand.Int()
switch {
case value == 20:
    //
case time.Now().Hour() < 10:
    //
}
```

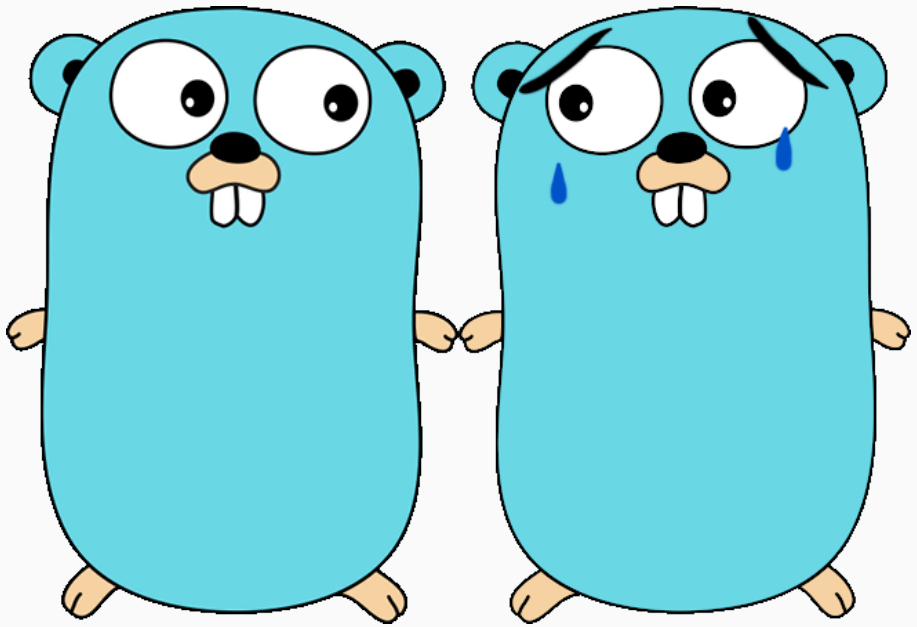
Widać, że mamy dużą wolność wyboru w pisaniu switcha. Switch nie wymaga wartości stałych, czy też liczb do swojego działania. Dzięki temu daje łatwość użycia.

Warto zauważyć, że nie używamy "break" na końcu "case". Go dodaje je automatycznie.

Uwaga
Switch jest wykonywany od góry do dołu i kończy się przy pierwszym trafieniu. Aby wymusić kolejne warunki, należy użyć słowa kluczowego: "fallthrough" na końcu warunku.

Enum

GoLang nie posiada implementacji Enuma. Jego zamiennik to zbiór stałych. Można przykład zobaczyć w ćwiczeniu 1.



Funkcje

Podstawowy zapis funkcji nie będzie odbiegać od innych języków:

```
func MyFunction (arg1 int, arg2, arg3 string) bool {
    return true
}
```

Widzimy, że możemy skracać zapis wielu argumentów z tym samym typem.

Dodatkowo, funkcja daje możliwość zwracania wielu danych:

```
func MyFunction (arg1 int, arg2, arg3 string) (bool, int) {
    return true, 20
}

a, b := MyFunction(10, "a", "b")
```

Nawias w zwracaniu danych
Nawias, który został dodany przy zwracanych danych jest wymagany dla więcej niż jednego wyjścia!

Funkcje II

Dodatkowo, możemy tutaj zdefiniować nazwy zwracanych parametrów:

```
func MyFunction (arg1 int, arg2, arg3 string) (test bool, value int) {
    if arg1 > 10 {
        test = true
        value = 20
    } else if arg1 == 10 {
        return true, 10
    }
    test = false
    return
}

a, b := MyFunction(10, "a", "b")
```

Uwaga!
Funkcja która coś zwraca, zawsze musi wywołać słowo kluczowe “return”!

Kiedy nazywać, kiedy nie?
Nazwy parametrów powinno się używać w przypadku, gdy chcemy ułatwić zrozumienie co zwraca funkcja. Jeżeli zwracane wartości są jasne, nie ma sensu wykorzystywać tego sposobu.

Funkcje III

Często bywa tak, że chcemy reużyć nazwę zmiennej do wyjścia funkcji, która zwraca więcej niż jeden parametr:

```
value, err := getValue()
if err != nil {
    // handle error
}

savedValue, err := saveValue()
_, err := saveValue() // Not it's error
```

Jak widać Go pozwoli nam użyć skrócony zapis ":=" dopóki chociaż jedna zmienna będzie nowa!

Pomijanie parametrów wyjściowych
W GoLangu, nieużywane zmienne, importy powodują błędy kompilacji. Aby ich uniknąć, można użyć zapisu "_ := myFunc()"

Conistency
Wiele osób uważa, że takie wykluczenie psuje spójność języka.

Funkcje IV - errory

Go nie posiada wyjątków, przez co funkcje zwracają błędy. Błędy implementują interface “error”. Do stworzenia nowego błędu można użyć zapisu: “errors.New(string)”.

Konwencja mówi, aby zwracać błędy jako ostatni parametr wyjścia!

```
func getSmth() error {
    // return fmt.Errorf("formatted error %s", "txt")
}
value, err := getValue()
err = getSmth()
a, b, c, err := getSmth3()
```

Uwaga!
Nie należy pomijać obsługi błędów (np. poprzez zapis “a, _ := ...”)!

Uwaga 2!
Nie rzucamy “panic” jeżeli nie musimy! Lepiej zwrócić błąd.

Tekst errorów
Błędy powinny zaczynać się z malej litery i nie kończymy go kropką: “this is correct error message”.

Defer, panic oraz recover

Istnieją sytuacje w kodzie, gdzie musimy coś zrobić za wszelką cenę. Takim przykładem jest otwarcie pliku. Jeżeli uda nam się utworzyć plik, to musimy go zamknąć.

```
src, err := os.Open(srcName)
if err != nil {
    return
}
// Long function
src.Close() // Wrong, function can be closed before!
```

Ułatwieniem jest "defer", który przeniesie wywołanie funkcji na sam koniec jej wywołania (w tym panic także się łapie):

```
src, err := os.Open(srcName)
if err != nil {
    return
}
defer src.Close() // No mather what, it'll be called
// Long function
```

Defer, panic oraz recover II

Czasami zdarza się tak, że jesteśmy w sytuacji bez wyjścia, gdzie my jako osoby piszące funkcję oraz osoby osoby, które ją wywołały nie możemy obsłużyć błędu. Wtedy właśnie możemy spanikować.

```
panic("message")
```

Panic jest pewnego rodzaju wyjątkiem, który wystąpił w aplikacji. Ponieważ jego obsługa nie jest prosta, należy wywoływać go w "wyjątkowych" sytuacjach.

Używaj errora zamiast panikować
Programista powinien stosować errory, zamiast przerywać bieg programu.

Defer, panic oraz recover III

Recover ułatwia nam odzyskanie pracy programu po wystąpieniu paniki. Jeżeli program panikuje i trafi na recover, jego przepływ wróci do normy.

Recover może być wywołany tylko w funkcji "defer".

```
func test() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    panic("test")
}
```

Recovery + panic a wyjątki
Zauważmy, że nie wiemy dokładnie kto i gdzie tutaj spanikował, przez co trudno obsługiwać i odzyskiwać "flow" takiego programu.

Importowanie

Importowanie paczek zaczyna się słowem kluczowym “import”. Przy importowaniu ostatni człon importowanej paczki tworzy globalną zmienną:

```
import "github.com/sonyarouje/simdb/db" // Makes db variable
```

Oczywiście, mogą zdarzyć się konflikty, wtedy należy zmienić nazwę importu:

```
import newName "github.com/sonyarouje/simdb/db"
// Makes newName variable
```

Importy można łączyć:

```
import (
    "a"
    "b"
)
```

Zewnętrzne paczki instaluje się za pomocą polecenie “go get”:

```
go get github.com/mattbaird/jsonpatch
```

Importowanie II

Dodatkowo istnieje zapis, kóry powoduje, że wszystkie wyeksportowanie (pisanej z wielkiej litery) funkcje oraz zmienne są dostępne bez prefiksu paczki:

```
import . "b"
```

Uwaga!
Używanie tego poza testami jest uznawane jako anty-pattern. Czasami używa się tego zapisu, aby ułatwić pisanie testów, które trafiły do innej paczki z powodu circular dependency.

Uwaga 2!
Paczki powinny być dobrze nazwane, zmiana nazw paczek bez konfliktów powinno występować bardzo rzadko.

Uwaga 3!
Jeżeli eksportujemy coś w paczce, nie duplikujemy nazewnictwa. Przykładowo, w paczce "Math" nie tworzymy typów "MathNumber". Takie coś zmniejsza czytelność kodu: "math.MathNumber" vs "math.Number".

Importowanie II

Widać po powyższych przykładach, że nie ma wersjonowania bibliotek. Go potrafi wersjonować biblioteki, lecz robi to inaczej niż reszta języków. Według GoLanga, dopóki ścieżka importu jest taka sama, nowa wersja paczki musi być wstecz kompatybilna.

Jeżeli chcemy wydać nową wersję ze złamaniem kompatybilności, zmieniamy ścieżkę importu:

```
github.com/googleapis/gax-go @ master branch
/go.mod    → module github.com/googleapis/gax-go
/v2/go.mod → module github.com/googleapis/gax-go/v2
```