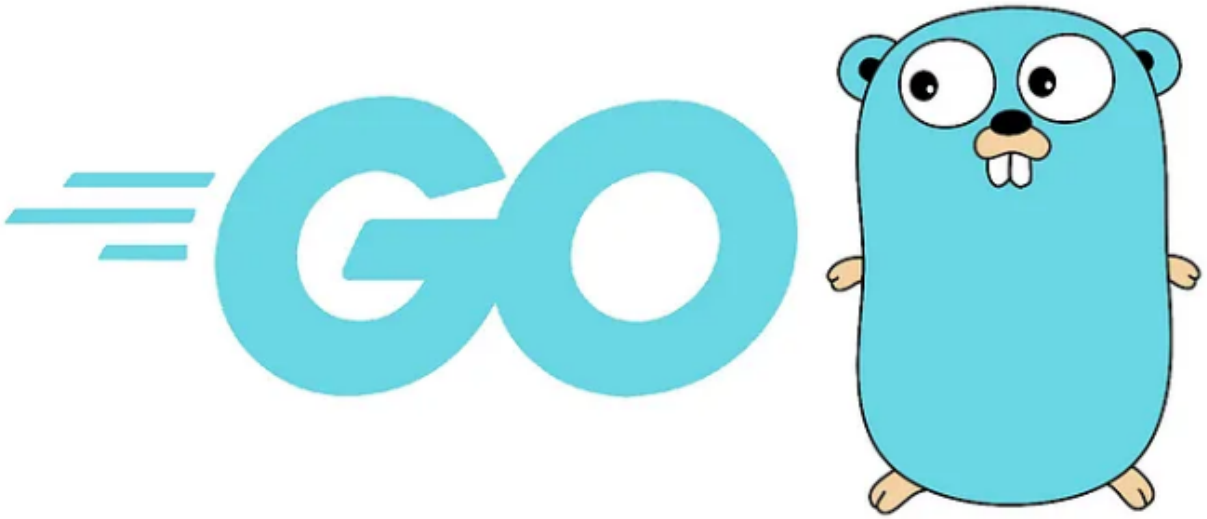

Diogo Antonio Sperandio Xavier

SUMÁRIO

Introdução

O objetivo do material em questão é apresentar um overview sobre **Golang**. **Não são necessários conhecimentos em programação** para compreender os conceitos e a ideia, mas caso possua algum, fica mais fácil o aprofundamento no tema.

Espero que esse livro possa colaborar com seu desenvolvimento de alguma forma, seja com conhecimento ou com um passo-a-passo que melhore sua venda e seus negócios. Caso isso aconteça, já estou de antemão muito feliz por isso.

Começando com Go

A linguagem é muito bem documentada e possui sites de apoio oficiais, com tutoriais e um editor online gratuito que permite realizar testes sem a necessidade de configurar nada em sua máquina local. Para instalar, basta seguir o passo a passo do site oficial, que é bem simples e intuitivo.

Links úteis:

- Site oficial Golang[1]
- Tour pela linguagem[2]
- Playground[3]

IDE

Recomendo a utilização do VSCode e download das extensões GO, que auxiliará nos imports e acessos a documentações e, JSON to Go, que vai facilitar a criação de objetos apenas colando e convertendo automaticamente para objetos (struct no go).

Fundamentos

Para que o programa seja executado, é necessária uma Função Main(), assim como em outras linguagens, como por exemplo, Java.

```
1 package main
2
3 func main() {
4
5 }
```

Para que possamos utilizar a linguagem, é necessário definir o nome de um pacote, que facilita muito o processo de separação de responsabilidades do programa. Eles são análogos aos pacotes do Java ou Namespaces do C#, PHP, etc.

Biblioteca Base

A linguagem possui uma biblioteca (conjunto de pacotes) muito completa e que facilita muito o desenvolvimento. A biblioteca padrão possui recursos de rede, http, serialização, sistema de templates, SQL e muitos outros, e devido a isso, não é comum a busca por frameworks e também não é comum a busca por pacotes externos.

Execução e Build

Para que você possa executar um programa desenvolvido em go, basta ter o runtime do Go instalado e executar os comandos:

```
1 go run main.go
```

Como Golang é uma linguagem compilada, logo, para gerar o arquivo compilado, basta executar:

```
1 go build main.go
```

Por ser uma linguagem multiplataforma, é possível compilar o programa em qualquer sistema operacional, como por exemplo:

Linux:

```
GOOS=linux go build main.go
```

MacOS:

```
GOOS=darwin go build main.go
```

Sintaxe

1. Escopo

A linguagem possui escopos muito bem delimitados. Todas as declarações globais, ou seja, fora de uma função, podem ser acessadas dentro do mesmo pacote. O mesmo acontece com funções. Todas as funções declaradas em um pacote, mesmo que estejam declaradas em arquivos diferentes, poderão ser acessadas de dentro de um mesmo pacote.

2. Visibilidade

Quando uma função ou mesmo um atributo de uma struct (falaremos sobre structs em breve) estiver com com a primeira letra maiúscula, significa que ela pode ser acessada de outro pacote. Ex:

```
1    // arquivo utils.go
2    package utils
3
4    func Exemplo() {
5       ...
6    }
7
8    // arquivo main.go
9    package main
10
11   import "utils"
12
13   func main() {
14      utils.Exemplo()
15   }
```

Como **Exemplo() está com letra maiúscula**, ela pode ser executada.

3. Variáveis

A declaração e atribuição de variáveis em golang podem ser feitas em qualquer contexto, dentro ou fora de uma função.

```
1  var b int
2  b = 18
3  var c, d string = "Hello", "World"
```

Dentro do escopo da função, é possível declarar e atribuir o valor de uma só vez utilizando := na atribuição. Exemplo:

```
func main() {
    nome := “Diogo”
}
```

4. Tipos

```
a := 8
b := "Olá!"
c := 7.77
d := true
e := 'D'
f := `... o amor, é assim, é a paz
      de Deus que nunca acaba...
      “Lugar ao Sol - Ao Vivo|Acústico - Raimundos” `

// Imprime valores
fmt.Printf("%v \n",a)
fmt.Printf("%v \n",b)
fmt.Printf("%v \n",c)
fmt.Printf("%v \n",d)
fmt.Printf("%v \n",e)
fmt.Printf("%v \n",f)

// Imprime tipos

fmt.Printf("%T \n",a) // int
fmt.Printf("%T \n",b) // string
fmt.Printf("%T \n",c) // float64
fmt.Printf("%T \n",d) // bool
fmt.Printf("%T \n",e) // int32
fmt.Printf("%T \n",f) // string
```

5. Blank identifier

A Golang não permite que uma variável seja declarada e não utilizada. Porém, por exemplo, em alguns momentos precisamos retornar valores na chamada de uma função. Ex:

```
response, err = http.Get(“http://...”) // função retorna 2 valores
```

Se a requisição acima conseguir ser executada com sucesso, a variável “**err**” estará vazia e não terá sido utilizada. Logo, o programa não poderá ser compilado. Os erros normalmente são tratados da seguinte forma:

```
1 if err != nil {
2   // tratativa
3 }
```

Caso não seja necessário utilizar uma das variáveis de retorno, utilizamos o “**Blank identifier**”. Ele é um “_” que o go identifica como “ignore este retorno”. Ex:

```
1 response, _ = http.Get(“http://...”)
```

Nesse caso, o resultado do segundo retorno será ignorado e o programa irá ser executado normalmente.

- **Constantes**

Uma constante basicamente é uma “variável” que não pode ter seu valor alterado. Em go, você precisa declarar uma constante utilizando apenas o “=” e não “:=”. A constante pode ser definida de forma global em seu pacote ou mesmo de forma local em uma função. Ex:

```
1 const gravidade float64 = 9.81
2 const vol = 200
3 const (
4   liq string = "água"
5   fus = 0
6   ebu int = 100
7 )
```

Ponteiros e endereçamento de memória

Em Go, é possível acessar o endereço de memória que um valor é gravado de forma direta.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6       x := 2
7       fmt.Println(&x)// 0xc000122058
8 }
```

O hexadecimal acima é o endereçamento de memória cuja variável x está sendo apontada. Para isso, tivemos que adicionar o “&” logo antes da variável. Nesse ponto, podemos criar uma nova variável e apontá-la exatamente para o mesmo endereçamento de x.

```
1 y := &x
2 fmt.Println(y)// 0xc000122058
```

Se em algum momento houver a necessidade de exibirmos o valor que foi atribuído para o endereçamento na memória através da variável y, basta adicionarmos “*” na frente da variável.

```
1 fmt.Println(*y)// 2
```

Através de ponteiros, você pode alterar o valor atribuído na memória.

```
1 *y = 5
2 fmt.Println(x) // 5
```

O valor de x foi alterado, pois o ponteiro de Y alterou esse valor. Também podemos declarar uma variável definindo um tipo.

```
1 var z *int = &x
2 fmt.Println(z) // 0xc000122058
3 fmt.Println(*x) // 5
```

Também é possível trabalhar com ponteiros dentro de funções, ex:]

```go
package main

import "fmt"

func main() {
	b := 10
	pontFunc(&b)
	fmt.Println(b) // 5
}

func pontFunc(a *int) int {
	*a = 5
	return *a
}
```

O valor de b foi alterado pelo fato de ele ter sido passado por parâmetro na função.

Funções

Sintaxe das funções em Golang:

```
1   func funcName(a int) int {
2
3   }
```

No caso acima, “funcName” é o nome da função, “a” que é do tipo inteiro é o parâmetro de entrada e “int” é o tipo de retorno. Existe uma leve sutileza quando trabalhamos com funções em Go. Ex:

```
1   func funcReturn(a string) string {
2       x = a
3       return
4   }
```

Nesse caso, mesmo não estando informando especificamente qual variável devemos retornar após o “return”, verifique que na definição da função, estamos especificando a variável “x”, logo, nesse caso, por padrão a função retornará o valor de “x = a”.

Retorno Múltiplo de Valores

Em Golang, é possível retornar mais de um valor em uma função.

```
1   func funcMultiReturn(a string, b int) (string, string) {
2
3   }
4   x, y := funcMultiReturn(“teste”, 2)
```

Neste caso, a função retorna dois valores de string, respectivamente x e y. Caso queira desprezar algum valor, basta utilizar o recurso anteriormente citado como **Blank identifier**.

Variadic Functions

Em Golang, existe um recurso que também é presente em algumas outras linguagens chamado **Variadic Functions**. Esse recurso permite informar uma quantidade indefinida de parâmetros de entrada.

```
1   func variadicFunc(x ...int) int {
2
3   }
```

Nesse exemplo, podemos informar diversos números inteiros:

```
1   variadicFunc(1,2,3,4,456,57,3,2)
2   variadicFunc(1,2,3)
```

No exemplo acima, existem duas possíveis chamadas da mesma função. Para que possa percorrer todos os valores, basta que seja executado um laço de repetição utilizando o recurso **"range"**.

```
1   func variadicFunc(x ...int) int {
2       var sum int
3       for _, num := range x {
4           sum += num
5       }
6       return sum
7   }
```

Na função acima, percorremos todos os valores de **"x"** de entrada e somamos, retornando assim o resultado da soma **"sum"**.

Outro ponto interessante é observar que utilizamos novamente um **blank identifier**, seguido de uma variável **"num"**, o que implica que a função **range** esteja retornando dois valores, um índice e um valor. Nesse caso, estamos apenas utilizando o valor **"num"**, e ignorando o índice por meio do **blank identifier**.

Funções anônimas

A linguagem permite também a criação de funções anônimas. Ex:

```
1   a := anonFunc() {
2       x += 1
3   }
```

Analogamente, podemos utilizar uma função dentro de outra função. Ex:

```
1   func funcInsFunc() anonFunc ()int {
2      x := 3
3       return func() int {
4           return x * x
5       }
6   }
```

Arrays

Arrays em Golang são uma forma de armazenar dados em um mesmo objeto, porém de forma fixa e totalmente tipada.

```
1   var x [10]int
2   x[0] = 1
3   x[1] = 38
4   // ou então
5   x := [5]int{1,2,4,8,16}
```

Slices

Slices são parecidas com arrays, porém não tem tamanho fixo e são dinâmicas. Ex:

```
1   // criar um sline
2   slice := make([]int)
```

Também é possível criar um slice com uma quantidade definida de posições, porém, esse valor poderá ser alterado futuramente. Ex:

```
1   slice := make([]int, 5) // 5 posições padrão.
2   slice[0] = 10
3   fmt.Println(slice[0]) // 10
4   slice = append(slice, 1,2,3,4,5)
```

Outra forma de declarar:

```
1   sliceStr := []string {
2       “Primeiro”,
3       “Segundo”
4   }
5   fmt.Println(sliceStr[0]) // "Primeiro"
```

Um grande recurso que slices disponibilizam na Golang é a possibilidade da navegação pelos índices.

Exemplo:

```
1   sliceString := []string {
2   "Diogo",
3   "Antonio",
4   "Sperandio",
5   "Xavier"
6   }
7   fmt.Println(sliceString[:2]) // "Diogo Antonio"
8   fmt.Println(sliceString[1:2]) // "Antonio" - A partir do índice
9   um até a segunda posição.
10  fmt.Println(sliceString[2:4]) // "Sperandio Xavier" - A partir do
11  índice 2 até a quarta posição.
12  fmt.Println(sliceString[2:]) // "Sperandio Xavier" - A partir do
13  índice 2 até ao final.
```

Maps

Os maps trabalham de forma similar aos dicionários no Python. Basicamente o princípio de chave-valor.

```
1   m := make(map[string]int)
2   m["a"] = 14
3   m["b"] = 29
```

Também é possível deletar um valor de um map com o seguinte comando:

```
1   delete(m, “b”)
```

Nesse ponto o valor foi apagado, porém se chamarmos o mapa no índice “b”, o valor será igual a zero, uma vez que o map é de inteiros. Para verificar se um valor existe dentro de um map, basta utilizá-lo normalmente, porém, adicione mais uma variável na chamada.

```
1   _, exists := m["b"]
2   fmt.Println(exists) // false
```

Como não estamos interessados no valor, mas sim na existência do índice, utilizamos um **blank identifier**, porém, a variável exists terá um valor true ou false atribuída a ela.

Outra maneira de definirmos um map é da seguinte forma:

```
1   var x = map[string] int {}
2   // ou
3   x := map[string]int{“a”:1, “b”:2}
```

Orientação a Objetos em Golang

Golang utiliza o paradigma de orientação a objetos de maneira diferente de outras linguagens de programação. Seus tipos e classes são substituídos por estruturas de dados que podem ter funções diretamente atreladas a elas.

Structs

1. Básico sobre Struct

A linguagem permite que sejam criados tipos específicos de dados. Ex:

```
1   type Nome string
2   type Ano int
3   type Cor string
```

Para relacionar os tipos declarados, utilizamos um tipo chamado de **Struct**. Ex:

```
1   type Carro struct {
2       Nome string
3       Ano int
4       Cor string
5   }
6
7   carro1 := Carro{“Ford Maverick”, 2023, “Preto”}
8   carro2 := Carro{“BMW X1”, 2021, “Prata”}
9
10  fmt.Println(carro1.Name) // Ford Maverick
11  fmt.Println(carro2.Color) // Prata
```

2. Funções em uma Struct

Para que possamos realizar ações em uma struct, é necessário que possamos atachar funções a elas.

```
type Carro struct {
    Nome string
    Ano int
    Cor string
}

func (c Carro) info() string {
    return c.Nome + “,
    cor: ” + c.Cor + “,
    ano: ” + c.Ano
}
```

3. Herança

Para evitar repetição no processo de criação das structs, podemos fazer com que elas se relacionem ou herdem umas às outras. Ex:

```
type CarroEsportivo struct {
    Esportivo bool
}

type Carro struct {
    Nome string
    Ano int
    Cor string
}

carro := Carro{“Ford Maverick”, 2023, “Preto”,
CarroEsportivo{Esportivo:false}}
```

Apesar de que na declaração tivemos que informar a Struct do `CarroEsportivo`, podemos chamar a propriedade herdada diretamente. Ex:

```
fmt.Println(car.Esportivo)
// ou
fmt.Println(car.CarroEsportivo.Esportivo)
```

4. Serialização em JSON

A Golang possui recursos nativos em sua biblioteca padrão para nos ajudar a serializar dados em diversos formatos, incluindo json. Ex:

```
type Carro struct {
    Nome string
    Ano int
    Cor string
}

carro := Carro{“Ford Maverick”, 2023, “Preto”}
resultado, _ := json.Marshal(car) // dados em bytes
fmt.Println(string(resultado)) // bytes transformados em string
```

Vale destacar que a regra de visibilidade (maiúsculo) se aplica nas structs, logo, os atributos devem estar sempre com letra maiúscula.

5. Tags

Tag é um mecanismo da linguagem para conseguir marcar um atributo de uma struct. Essa marcação pode ter diversos objetivos. No exemplo que apresentaremos agora, será para facilitar o processo de manipulação da serialização dos dados para json. Algumas bibliotecas como gorm terão padrões de sintaxe próprios.

```
type Carro struct {
    Nome string  `json:“Modelo”`
    Ano int      `json:“-”`
    Cor string   `json:“Cor”`
}
```

Perceba que no exemplo acima, no atributo Ano, adicionamos json:"-". Isso significa que quando os dados forem serializados para json, o atributo Ano não deverá ser exibido. Além de termos a opção de ocultar um atributo durante o processo de serialização, podemos também alterar a chave do atributo, como é feito no “Nome”, que é modificado na impressão do JSON para “Modelo”.

6. Bind de Json para Struct

Recebendo um JSON e convertendo em uma Struct. Ex:

```
type Carro struct {
    Nome string
    Ano int
    Cor string
}

var carro Carro
json := []byte(`{“Nome”:“Corolla”,“Ano”:“2022”,“Cor”:“Branco”}`)

json.Unmarshal(json, carro)
fmt.Println(carro.Nome) // Corolla
```

7. Interfaces

A Linguagem Go também possui o conceito de Interfaces, ou seja, é um tipo de objeto que possui uma definição. Quando uma struct possui a exata definição de tal interface, isso significa que a struct está a implementando, logo, ela também pode ser considerada do mesmo tipo que a interface.

Diferentemente de linguagens tradicionais orientadas a objetos, em Golang não obriga de forma declarativa que uma struct tenha que implementar uma interface usando instruções como “implements”. A implementação é automática. Basta que a struct possua a mesma assinatura informada na interface. Ex:

```
type Veiculo interface {
    Partida() string
}

type Carro struct {
    Nome string
}

func (c Carro) partida() string {
    return “O carro deu partida”
}
```

Mesmo a struct “Carro” não informando de forma explícita, a mesma está implementando a interface “Veiculo”. Nesse ponto podemos trabalhar com algo do tipo:

```
1   func TesteComCarro (v veiculo) {
2       return v.Partida()
3   }
4
5   var c Carro
6
7   TesteComCarro(c)
```

Como a struct “Carro” implementa a interface “Veiculo”, essa struct pode ser interpretada como “Veiculo”.

Sites de Apoio e Bibliografia

1. https://go.dev/;
2. https://go.dev/tour/welcome/1;
3. https://go.dev/play/;
4. https://code.visualstudio.com/download;
5. https://freshman.tech/snippets/go/cross-compile-go-programs/;
6. https://gorm.io/index.html;