Introdução ao Go

Murilo Santana (mvrilo)
PicPay | Store
Abr 2021

Porque Go

- Sintaxe simples e pequena
- Formatação padronizada (gofmt)
- Fácil deploy (binários estáticos + rápida compilação)
- Fortemente tipada
- Suporte a Concorrência
- Alta performance

Hello world

```
package main

import "fmt"

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

3

História

Go iniciou em 2007 no Google com Rob Pike, Robert Griesemer e Ken Thompson.

Russ Cox e Ian Lance Taylor entraram para o time em 2008.

Foi publicamente anunciado em 2009 (www.youtube.com/watch?v=rKnDgT73v8s (https://www.youtube.com/watch?v=rKnDgT73v8s)).

Go nasceu como uma alternativa a outras linguagens de programação da época, almejando ser uma linguagem de alta performance, com segurança, rápida compilação e uma sintaxe minimalista.

Eficiência de uma linguagem compilada com a facilidade de uma linguagem dinâmica. 4

Go 1.0 em 2012, garantindo estabilidade da linguagem e mantendo o máximo de compatibilidade possível.

- pre-1.0 removeu ponto e virgula
- 1.4 introduziu `go generate`
- pre-1.5: compilador era separado por implementações: 6g, 8g
- 1.5 compilador refeito de C para Go
- 1.7 introduziu `context`
- 1.16 introduziu `embed`

Go 2.0 em 202x, introduzindo generics na linguagem.

Atualmente na versão 1.16

5

Instalação

```
brew install go
```

Gerenciamento de dependencias embutido (e distribuido)

```
go get github.com/picpay/meuprojetogo
```

6

Tour pela linguagem

7

Tipos Básicos

- bool
- string
- int, int8, int32, int64
- uint, uint8, uint32, uint64, uintptr
- byte (alias de uint8)
- rune (alias de int32)
- float32, float64

8

Variáveis

Inicializando variáveis com valores zerados:

```
var name string // ""
name = "picpay"
```

Variáveis com inferência do tipo:

```
name := "picpay"
```

Variáveis com o tipo explícito:

```
var name string = "picpay"
```

9

arrays e slices

Arrays em Go são sempre de tamanho fixo:

```
var names [2]string
names[0] = "hey"
names[1] = "picpay"

name := [1]string{"hey"}
```

Slices são "arrays" flexíevis com tamanho dinâmico:

```
var names []string
names = append(names "hey")
```

10

maps

Maps são dicionários, fazem um mapeamento de chave e valor, para inicializar um map, usamos a função make:

```
picpay := make(map[string]string)
picpay["name"] = "picpay"
picpay["age"] = "9"
```

11

packages

Todo programa Go é feito de pacotes (`package`), o pacote `main` é quem define o entrypoint do programa.
Para importar pacotes, usamos o `import`:

```
package main

import "fmt"

func main() {
    fmt.Println("hey PicPay!")
}
```

12

import

Nomes que começam com a primeira letra em maiúsculo são exportadas nos seus pacotes.

```
package picpay

var Name string = "hey PicPay"
var age int = 9
```

```
package main

import "picpay" // exemplo

func main() {
    fmt.Println(picpay.Name)
    fmt.Println(picpay.age) // error
}
```

13

Funções

Definindo funções:

```
func example(x int, y int) int {
    return x + y
}
```

Quando a assinatura dos argumentos é igual, podemos omitir os primerios tipos:

```
func example(x, y int) int {
    return x + y
}
```

Funções podem retornar múltiplos valores:

```
func example(name string) (string, string) {
    return "hey", name
}

func main() {
    a, b := example("picpay")
    // b == "hey"
    // b == "picpay"
}
```

if

```
func example(n int) string {
    if n > 5 {
        return "big"
    }
    return "small"
}
```

if/else:

```
func example(n int) string {
    if n > 5 {
        return "big"
    } else {
      return "small"
    }
}
```

15

for

```
func example() int {
    var sum int
    for i := 0; i < 10; i++ {
        sum += i
    }
    return sum
}
```

For pra sempre:

```
func example() {
    for {
    }
}
```

16

switch

```
func os() string {
    switch os := runtime.GOOS; os {
    case "darwin":
        return "osx"
    case "linux":
        return "linux"
    default:
        return "desconhecido"
    }
}

func isUnix() bool {
    switch os := runtime.GOOS; os {
    case "darwin", "freebsd", "linux":
        return true
    default:
        return false
    }
}
```

17

defer

defer indica a execução de uma função no final da função atual:

```
package main

import "fmt"

func main() {
    defer fmt.Println("world")
    fmt.Println("hello")
    // hello
    // world
}
```

18

struct

Structs são coleções de campos:

```
package main

struct Picpay {
    Name string
    Age int
}

func main() {
    pp := Picpay{
        Name: "picpay",
        Age: 9,
    }
}
```

19

Ponteiros

Ponteiros no Go indicam o endereço na memória de um valor.

```
package main

type example struct {
  value int
}

func main() {
  a := example{
    value: 1,
  }

  // referencia ao a
  b := &a
  b.value = 2

  // copia do a
  c := a
  c.value = 3

  println(a.value)
}
```

20

Métodos

Go não tem classes. Porém podemos definir métodos para tipos:

```
package main

type example struct {
    name string
}

func (e example) getName() string {
    return e.name
}

func main() {
    ex := example{
      name: "test",
    }
    println(ex.getName())
}
```

21

Métodos: em qualquer tipo concreto

Métodos podem ser aplicados a qualquer tipo, não só structs:

```
package main

type custom string // name type wraps a string

func (c custom) Length() string {
    return len(c)
}

func main() {
    name1 := "picpay" // string type
    name2 := custom(name1) // name type
    println(name2.Length())
}
```

22

Interfaces

Interface é a assinatura de um conjunto de métodos. Elas podem ser implementadas por qualquer tipo no Go e, diferente de outras linguagens, a implementação de interfaces é sempre implícita:

```
package main

type buyer interface {
    Buy(id string, value int)
}

type user1 struct {}
type user2 struct {}

// com o método definido, essa struct se torna uma implementação da interface buyer
func (u user1) Buy(id string, value int) {
    // buy code logic
}

func main() {
    var u1 buyer = user1{}
    var u2 buyer = user2{} // error: nao implementa o buyer
}
```

23

Interface vazia

Uma interface sem nenhum método especificado é conhecido como "interface vazia":

```
interface{}
```

É comum ser usado como uma forma de lidar com tipos desconhecidos:

```
var any interface{} = "name"
any = 1
any = true
any = func() string { return "aceita tudo" }
```

*

```
package main

import "fmt"

func describe(i interface{}) {
    fmt.Printf("(%v, %T)\n", i, i)
}

func main() {
    var i interface{}
    describe(i)
```

```
    i = 42
    describe(i)

    i = "hello"
    describe(i)
}

// (<nil>, <nil>)
// (42, int)
// (hello, string)
```

24

Asserção do tipo

Asserção de tipo é uma forma de acessar o tipo do valor atrás da interface:

```
var name interface{} = "picpay"
sname := name.(string) // retorna um valor do tipo string

name == "picpay" // compilation error
sname == "picpay" // true
```

25

Erros

Erros são expressados como valor do tipo `error`:

```
func get() error {
    return errors.New("isso é um erro")
}

func main() {
    err := get()
    if err != nil {
        panic(err)
    }
}
```

O valor retornado no error é sempre um implementação do tipo `error` do Go.

```
type error interface {
    Error() string
}
```

26

Concorrência

Go tem suporte a concorrência a nível de linguagem. Através de goroutines e channels. 27

Goroutine

Goroutine (go + routine) é uma thread (leve) gerenciada pela runtime do Go.

Para iniciar uma goroutine, basta chamar uma função com a palavra go antes dela:

```
func get() {
    // codigo
}

// inicia uma nova goroutine que executará a função get
go get()
```

28

Channels

Channels são uma forma de comunicar dados dentro de goroutines de forma segura. Channels podem ser aplicados a qualquer tipo de variável e elas precisam ser criadas antes de serem usadas. Para isso usamos o `make` assim como em maps e arrays:

```
ch := make(chan string) // cria um canal do tipo string
ch <- "picpay" // envia um valor
name := <-ch // recebe um valor e atribuí ao name
```

29

Exemplo: concorrência

Async/Await:

```
c := make(chan int, 1)
go func() { c <- process() }() // async
v := <-c // await
```

30

Exemplo: net/http

```
package main

import "net/http"

type server string

func (s server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(200)
  w.Write([]byte(s))
}

func main() {
    println("http server started at localhost:8000")
    http.ListenAndServe(":8000", server("go picpay!"))
}
```

31

Próximos passos:

- Alocação de memória (stack vs heap)
- Sincronização/Comunicação
- Concurrency patterns
- Design patterns

Referencias

tour.golang.org/list (https://tour.golang.org/list)

golang.org/doc/effective_go.html (https://golang.org/doc/effective_go.html)

32

Thank you

Murilo Santana (mvrilo)
PicPay | Store
Abr 2021