Golang

Matheus Salgueiro Castanho - Linguagens de Programação 2015/2

Pesquisa | Estou com sorte

Sumário

- Introdução
- Sintaxe e Conceitos Básicos
- Características
- Comparações
- Conclusão
- Trabalho

Introdução

- Desenvolvida por Robert Griesemer, Rob Pike e Ken Thompson, funcionários do Google, em 2007.
- Não satisfeitos com a complexidade das linguages mais utilizadas, como Java e C++, decidiram fazer uma nova linguagem.
- Buscaram evitar códigos prolixos, favorecendo códigos sofisticados e concisos.
- Além disso, queriam uma linguagem mais adaptada para a realidade atual da computação (programação distribuída, nuvem, multicore CPUs)

Introdução

- Queriam uma linguagem que aproveitasse os “prós” e evitasse os “contras” das outras:
  - Prós:
    - Eficiência
    - Aplicabilidade
    - Sistema de tipos seguro
    - Escalabilidade
  - Contras:
    - Complexidade
    - Compilação lenta
    - Tipagem dinâmica
    - Uso excessivo de ferramentas externas

Introdução

- Resultado: Go, também chamada de Golang
- Em produção desde de 2007, tornou-se open source em 10 de novembro de 2009 e a versão 1.0 foi lançada em março de 2012.
- É uma linguagem compilada, fortemente tipada, estruturada com traços de orientação a objetos.
- Está atualmente na versão 1.5
- É de código aberto.
- Mantida por uma equipe do Google e por contribuidores externos.

Sintaxe
e
Conceitos Básicos

Hello World

```go
package main

import "fmt"

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

Palavras-chave e reservadas

- Go não apresenta nenhuma palavra reservada, apenas palavras-chave.

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Palavras-chave em Go

- Sim, Go implementa e permite o uso de *goto*!

Identificadores pré-definidos

- Os identificadores abaixo são pré-definidos pela linguagem e não podem ser redefinidos pelo programador

```
Types:
        bool byte complex64 complex128 error float32 float64
        int int8 int16 int32 int64 rune string
        uint uint8 uint16 uint32 uint64 uintptr

Constants:
        true false iota

Zero value:
        nil

Functions:
        append cap close complex copy delete imag len
        make new panic print println real recover
```

Identificadores pré-definidos em Go

Operadores, delimitadores e tokens especiais

- Go apresenta os principais operadores e delimitadores usados por outras linguagens.

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=
```

Variáveis

```go
package main
import "fmt"

func main() {

    // `var` declara 1 ou mais variáveis
    var a string = "initial"
    fmt.Println(a) //output: initial

    // É possível declaras várias variavéis de uma vez
    var b, c int = 1, 2
    fmt.Println(b, c) //output: 1 2

    // Go infere o tipo de variáveis inicializadas
    var d = true
    fmt.Println(d) //output: true

    // Variáveis declaradas sem um valor correspondente
    // são inicializadas recebem valores-zero. Por exemplo, o
    // valor-zero para o tipo an `int` é `0`.
    var e int
    fmt.Println(e) //output: 0

    // O comando `:=` representa uma outra forma de
    // declarar e inicializar uma variável, por exemplo
    // `var f string = "short"`, nesse caso
    f := "short"
    fmt.Println(f) //output: short
}
```

Comando de Repetição

- Go apresenta apenas um comando de repetição: o laço *for*
- Outras formas mais adiante: *for* + *range* (for-each)

```go
package main

import "fmt"

func main() {

    // pseudo-while
    // Tipo básico, apenas uma comparação
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // for clássico
    // O clássico trio initializaçãocondição-atualização
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    // loop infinito
    // repetirá indefinidamente até
    // executar um break ou um return
    for {
        fmt.Println("loop")
        break
    }
}
```

Condicionais

- Apresenta as mesmas estruturas condicionais de C
- Porém permite inicialização de variável dentro do if

```
package main

import "fmt"

func main() {

    // É possível usar o if sem o else
    if 8%4 == 0 {
         // output: 8 eh divisivel por 4
        fmt.Println("8 eh divisivel por 4")
    }

    // Uma inicialização pode preceder o condicional.
    // Variáveis declaradas dessa maneira podem ser
    // acessadas por todos os ramos condicionais
    // associados.
    if num := 9; num < 0 {
        fmt.Println(num, “eh negativo")
    } else if num < 10 {
        fmt.Println(num, “tem 1 digito")
    } else {
        fmt.Println(num, “tem varios digitos")
    }
    // output: 9 tem 1 digito
}
```

Switch

- Aceita qualquer tipo de dado (não apenas *int*)
- Executa apenas um caso que seja satisfeito (não precisa de *break*)
- Caso a execução de múltiplos casos seja desejada, usa-se *fallthrough*
- Pode apresentar condicionais internamente

```go
package main
import "fmt"
import "time"

func main() {

    // É possível usar mais de um valor
    // no mesmo caso, separados por vírgula.
    // default é opcional e define o caso padrão
    switch time.Now().Weekday() {
      case time.Friday:
            fmt.Println("hoje eh sexta =D")
      case time.Monday, time.Tuesday, time.Wednesday, time.Thursday:
            fmt.Println("hoje nao eh sexta nem fds =(")
      default:
            fmt.Println("Ufa! Eh fds, posso descansar.")
    }

    // Switch sem uma expressão para ser avaliada
    // é uma alternativa para uma estrutura if-else
    // encadeada.
    s := "palavras"
    switch {
      case len(s)%2 == 0:
            fmt.Println("Tamanho eh par")
            fallthrough
      case len(s)%4 == 0:
            fmt.Println("Tamanho eh multiplo de 4")
            fallthrough
      case len(s)%8 == 0:
            fmt.Println("Tamanho eh multiplo de 8")
      default:
            fmt.Println("Nao eh multiplo de 2, 4 ou 8.")
    }
}
```

Funções

- A declaração de funções é sempre precedida pela palavra-chave *func.*
- Os argumentos podem ser passados entre parêntesis, separados por vírgula, no padrão de C.
- Porém se parâmetros consecutivos tiverem o mesmo tipo, o tipo pode ser declarado para o último da lista e omitido para os anteriores
- Podem ter múltiplos valores de retorno, que podem ser nomeados
- São cidadãs de primeira classe (slide sobre closures)

```
package main

import "fmt"

// Tipos consecutivos
func soma(a, b int) int {
     return a + b
}

// Múltiplos valores de retorno
func div2(a int, b int) (int,int) {
     return a/2, b/2
}

func multDiv( a, b int) (mult int, div int) {
     mult = a*b
     div = a/b
     return
}

func main() {

    res := soma(1,2)
    fmt.Println("1+2 =", res)

    res1, res2 := div2(2,4)
    fmt.Println("2/2 , 4/2 =",res1,",",res2)

    mult, div := multDiv(4,2)
    fmt.Println("4*2 , 4/2 =",mult,",",div)

}
```

Output:

```
1+2 = 3
2/2 , 4/2 = 1 , 2
4*2 , 4/2 = 8 , 2
```

Funções com lista de parâmetros variável

- Funções podem ter uma lista de parâmetros de tamanho variável
- Declaradas como tendo a entrada *'...tipo"*
- A declaração de parâmetros variáveis deve ser a última a lista de parâmetros
- fmt.Println() tem tal propriedade
- Elementos de slices podem ser passados como parâmetros para tais funções
- Extra: funções void não declaram valor de retorno

```
Output:

[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

```go
package main

import "fmt"

// Função que recebe número arbitrário de
inteiros
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {

    // São chamadas da forma comum
    sum(1, 2)
    sum(1, 2, 3)

    // Os elementos de um slice podem ser
    // passados como parâmetros usando
    // 'slice...'
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

Closures

- Como funções são cidadãs de primeira classe, podem ser passadas como parâmetro para outras funções, podem ser retorno de funções e também podem ser atribuídas a variáveis
- Cada instância de uma função apresenta suas próprias variáveis internas, independentes umas das outras.
- A função ao lado retorna uma função que retorna número sucessivos a cada chamada

```
package main

import "fmt"

// Retorna uma outra função definida
// internamente.
func intSeq() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}

func main() {

    // Atribuímos a função a uma variável.
    // Essa instância da função terá seu próprio
    // `i` na memória, que será alterado a cada
    // chamada
    nextInt := intSeq()

    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    // Se criarmos uma outra instância,
    // outro `i` será criado. Diferente
    // do primeiro.
    newInts := intSeq()
    fmt.Println(newInts())
}
```

Output:

1
2
3
1

Recursão

- Go também apresenta recursão

```go
package main

import "fmt"


func fatorial(n int) int {
    if n == 0 {
        return 1
    }
    return n * fatorial(n-1)
}

func main() {
    fmt.Println(fatorial(5))
}
```

Output:

120

Arrays

- Muito similar a arrays em C
- Há apenas diferença na declaração
- São definidos pelo tipo dos dados e pelo tamanho fixo
- A linguagem oferece a função ***len*** que retorna o tamanho de um array

```go
package main

import "fmt"

func main() {

    // Inicialização
    var a [5]int
    fmt.Println("emp:", a)

    // Acesso é similar a C
    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    // Função 'len' que retorna o tamanho
    fmt.Println("len:", len(a))

    // Array literal
    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    // Também é possível fazer arrays 2D
    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

Slices

- Formam uma estrutura chave em Go
- Similares a arrays, porém mais flexíveis
- São definidos apenas pelo seu tipo (e no fundo, capacidade)
- Seu tamanho é variável
- Tamanho ≠ Capacidade

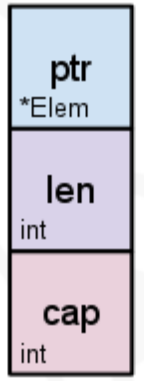
```go
package main

import "fmt"

func main() {

    // Não são presos ao tamanho, apenas ao tipo
    s := make([]string, 2, 3)
    fmt.Println("emp:", s)

    // Acesso igual a arrays
    s[0] = "a"
    s[1] = "b"
    fmt.Println("set:", s)
    fmt.Println("get:", s[1])

    // Função 'len' retorna o tamanho
    // Função 'cap' retorna a capacidade
    fmt.Println("len:", len(s))
    fmt.Println("cap:", cap(s))

    // Para aumentar a capacidade de um slice
    // deve-se usar a função 'append'
    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)

    // Também podemos copiar slices
    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    // Além disso, podemos cortar slices também
    // para criar slices menores. Nesse caso,
    // os dois slices apontam para a mesma estrutura
    // na memória
    d := []byte{'g', 'a', 't', 'o'}
    e := d[:2]
    // e == []byte{'g', 'a'}
    e[0] = 'r'
    // e == []byte{'r', 'a'}
    // d == []byte{'r', 'a', 't', 'o'}
}
```

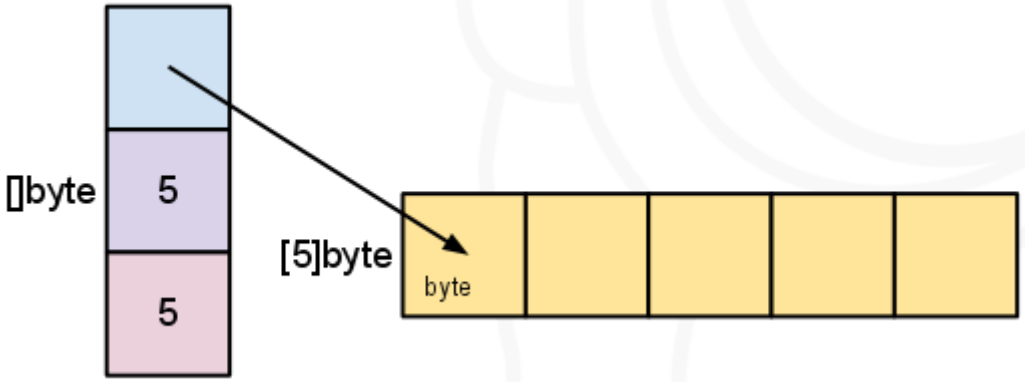
Output:

emp: [ ]
set: [a b]
get: b
len: 2
cap: 3
apd: [a b d e f]
cpy: [a b d e f]

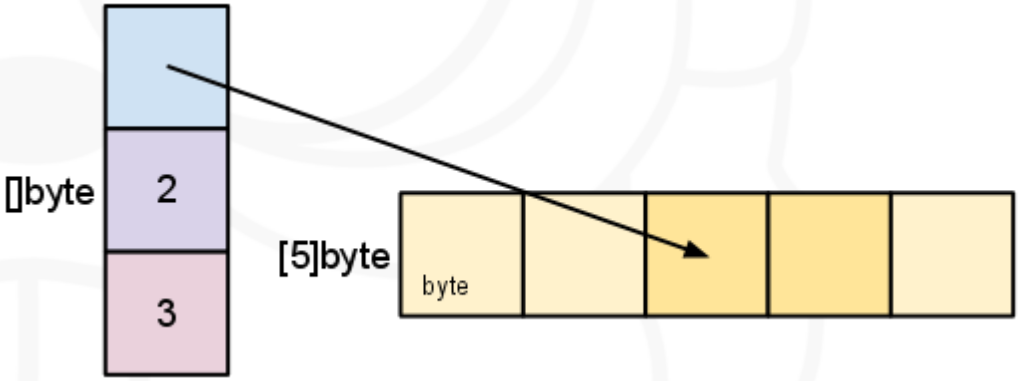
Slices e memória

- Estrutura interna contém uma referência para um array na memória com o mesmo tamanho da capacidade do slice.
- Além de um tamanho e uma capacidade.





Estrutura interna de um slice de bytes de tamanho 5 e capacidade 5



Estrutura interna após uma chamada 's = s[2:4]'

Mais informações: [Go Slices: Usage and Internals]

Mapas

- Sintaxe muito similar à de arrays e slices, porém chaves não precisam ser inteiros
- Mapas precisam ser inicializados com *make*
- Checagem de presença de valor feita de forma simples

```go
package main

import "fmt"

func main() {

    // Para criar um mapa vazio, basta chamar a
    // função 'make'
    m := make(map[string]int)

    // Para adicionar um valor, basta
    // escrever 'mapa[chave] = valor'
    m["k1"] = 7
    m["k2"] = 13

    // Imprimir um mapa mostrará todos os pares
    // chave, valor
    fmt.Println("map:", m)

    // O acesso a valores é similar ao de arrays
    v1 := m["k1"]
    fmt.Println("v1: ", v1)

    // 'len' retorna o número de pares no mapa
    fmt.Println("len:", len(m))

    // Para remover um par, basta usar 'delete'
    delete(m, "k2")
    fmt.Println("map:", m)

    // O acesso apresenta um parâmetro de retorno
    // adicional opcional que indica se a chave
    // está presente no mapa ou não
    _, prs := m["k2"]
    fmt.Println("prs:", prs)

    // Também é possível inicializar um mapa
diretamente
    n := map[string]int{"foo": 1, "bar": 2}
    fmt.Println("map:", n)
}
```

Output:

```
map: map[k1:7 k2:13]
v1:  7
len: 2
map: map[k1:7]
prs: false
map: map[foo:1 bar:2]
```

Range e Iterações

- Ferramenta ponderosa para executar iterações em coleções
- Equivalente ao *foreach* de outras linguagens
- Retorna referências para índice e valor

```go
package main

import "fmt"

func main() {

    // Usado para somar os valores de um slice
    // Funciona igualmente em arrays.
    // Podemos ignorar o índice com '_'
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    // Usando índice e valor
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }

    // 'range' sobre um mapa itera sobre chaves e valores.
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
}
```

Output:

sum: 9
index: 1
a -> apple
b -> banana

Ponteiros

- Go apresenta ponteiros, porém sem aritmética de ponteiros
- Ponteiros para um tipo são inferidos ou declarados com '*tipo'
- Acesso à referência a uma variável é feita usando '&'
- O valor apontado pode ser acessado usando-se '*ponteiro'

```
package main

import "fmt"

func zeraPorValor(ival int) {
    ival = 0
}

func zeraPorReferencia(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    ptr := &i

    fmt.Println("inicial:", i)

    zeraPorValor(i)
    fmt.Println("porValor:", i)

    zeraPorReferencia(ptr)
    fmt.Println("porReferencia:", i)

    fmt.Println("Acesso usando o ponteiro:",*ptr)
    fmt.Println("ponteiro:", ptr)

    //ptr += 1
    // Não compila: invalid operation: ptr += 1 (mismatched types *int and int)
}
```

Output:

inicial: 1
porValor: 1
porReferencia: 0
Acesso usando o ponteiro: 0
ponteiro: 0xc0820022d0

Structs

- Structs declarados de forma parecida a C
- Passagem nominal
- Ponteiros também usam ‘ . ‘

Output:

{Bob 20}
{Alice 30}
{Fred 0}
Sean
50
51

```go
package main

import "fmt"

type person struct {
    name string
    age  int
}

func main() {

    // Criação de uma nova estrutura
    fmt.Println(person{"Bob", 20})

    // O construtor aceita passagem de parâmetros por nome
    fmt.Println(person{name: "Alice", age: 30})

    // Campos omitidos receberam o seu valor-zero
    fmt.Println(person{name: "Fred"})

    // Acesso aos campos é feito por meio de '.'
    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    // Mesmo com ponteiros o ponto é usado
    sp := &s
    fmt.Println(sp.age)

    // Structs são mutáveis
    sp.age = 51
    fmt.Println(sp.age)
}
```

Métodos

- Em Go structs também podem ter métodos associados a eles
- Não existe “this” em Go

```go
package main

import (
    "fmt"
    "time"
)

type person struct {
    name string
    age  int
}

func (p person) anoDeNascimento() int {
    return time.Now().Year() - p.age
}

func main() {

    m := person{name: "Matheus", age: 21}
    j := person{name: "Joao", age: 30}

    fmt.Println(m.anoDeNascimento())

    fmt.Println(j.anoDeNascimento())
}
```

Output:

1994
1985

Interfaces

- Interfaces são implementadas implicitamente (não precisa de *implements*)
- Polimorfismo

```
package main

import "fmt"
import "math"

type forma interface {
    area() float64
    perimetro() float64
}

type retangulo struct {
    base, altura float64
}
type circulo struct {
    raio float64
}

// Para implementar uma interface em Go,
// é preciso apenas implementar os métodos
// declarados por ela. A declaração é
// feita implicitamente.
func (r retangulo) area() float64 {
    return r.base * r.altura
}
func (r retangulo) perimetro() float64 {
    return 2*r.base + 2*r.altura
}
func (c circulo) area() float64 {
    return math.Pi * c.raio * c.raio
}
func (c circulo) perimetro() float64 {
    return 2 * math.Pi * c.raio
}

// Se uma estrutura implementa uma interface
// ela pode ser referenciada como uma instância
// daquela interface
func measure(g forma) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perimetro())
    fmt.Println()
}

func main() {
    r := retangulo{base: 3, altura: 4}
    c := circulo{raio: 5}

// Como retangulo e circulo implementam
// a interface `forma`, ambos podem ser
// passados como parâmetros para a função
// measure
    measure(r)
    measure(c)
}
```

Output:

```
{3 4}
12
14

{5}
78.53981633974483
31.41592653589793
```

Goroutines

- Comando colateral de Go: *go*
- Esse comando executa a função passada como argumento em uma thread paralela.
- Extra: comando *defer*, adia a execução de um comando até o fim da função que o chamou

Output:

sequencial : 0
sequencial : 1
sequencial : 2
paralelo1 : 0
paralelo2
paralelo1 : 1
paralelo1 : 2

```go
package main

import "fmt"

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {

    defer fmt.Println("Fim.")

    f("sequencial")

    // Para iniciar uma goroutine basta
    // colocar o comando go antes de uma função
    go f("paralelo1")

    // Goroutines também podem ser executadas
    // com funções anônimas
    go func(msg string) {
        fmt.Println(msg)
    }("paralelo2")


    // Após a chamada 'go' uma thread
    // é criada para executar a goroutine
    // e a execução do código principal continua
    var input string
    fmt.Scanln(&input)
}
```

Channels

- Channels são o mecanismo de comunicação entre goroutines
- Implementam send e receive bloqueantes, só há comunicação quando as duas pontas estão prontas
- São inicializados de forma similar a outras estruturas como arrays, slices e mapas

Output:

-5 17 12

```go
package main

import "fmt"

func sum(a []int, c chan int) {
    sum := 0

    for _, v := range a {
       sum += v
    }

    c <- sum // send sum to c
}

func main() {

     a := []int{7, 2, 8, -9, 4, 0}

     c := make(chan int)

     go sum(a[:len(a)/2], c)
     go sum(a[len(a)/2:], c)

     x, y := <-c, <-c // receber do channel c

     fmt.Println(x, y, x+y)
}
```

Buffered Channels

- Channels também podem ser “bufferizados”
- Dessa forma, eles só bloqueam quando o buffer está cheio

```go
package main

import "fmt"

func main() {

    messages := make(chan string, 2)

    messages <- "buffered"
    messages <- "channel"

    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

Channels com direção

- Ao passarmos channels como parâmetros para funções, podemos especificar se queremos enviar ou receber dele usando-se <-
- Ausência indica ambos
- Tentar receber de um channel que só envia (e vice-versa) gera um erro de compilação

```go
package main

import "fmt"

func ping(pings chan<- string, msg string) {
    pings <- msg
}

func pong(pings <-chan string, pongs chan<-
string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

Select

- O comando *select* permite executar tratamentos diferentes para dados recebidos de canais diferentes
- Poderosa ferramenta para comunicação com várias threads simultaneamente

```go
package main

import "time"
import "fmt"

func main() {

    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(time.Second * 1)
        c1 <- "um"
    }()
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "dois"
    }()


    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("Recebido do canal 1:", msg1)
        case msg2 := <-c2:
            fmt.Println("Recebido do canal 2:", msg2)
        }
    }
}
```

Range e Channels

- É possível usar *range* para iterar sobre elementos de um channel também
- Para fechar um *channel*, usa-se a função *close*
- Mesmo após o *close* os dados do channel continuam disponíveis para acesso.
- Caso o canal não tivesse sido fechado antes do *range*, o loop ficaria bloqueado enquanto o terceiro elemento não chegasse no canal, nesse caso, causando um deadlock

```go
package main

import "fmt"

func main() {

    queue := make(chan string, 2)
    queue <- "one"
    queue <- "two"
    close(queue)

    for elem := range queue {
        fmt.Println(elem)
    }
}
```

Características da Linguagem

Sobrecarga de métodos e operadores

- Ao contrário de C++, Go não permite a sobrecarga de operadores.
  - Justificativa: É mais uma conveniência do que um requisito absoluto para a linguagem. Sem isso as coisas são mais simples.

- Go também não permite a sobrecarga de funções
  - Justificativa: Apesar ser útil em certas situações pode ser confusa e frágil na prática. A decisão por manter a identificação da função apenas pelo nome foi importante para a simplificação da linguagem.

Justificativas retiradas de Golang FAQ

Escopo

- Em Go, variáveis tem escopo estático definido por blocos delimitados por { }.
- Variáveis definidas em blocos aninhados internos não são visíveis nos blocos externos.
- Funções, *structs*, constantes e variáveis globais tem escopo dependente da letra inicial do seu identificador.
- Caso o identificador comece com letra maiúscula, ele é exportado e é acessível fora do pacote onde foi declarado.
- Caso comece com letra minúscula, ele não é exportado, ficando restrito apenas ao pacote onde foi declarado.
- Consequência 1: todas as funções de pacotes padrão de Go tem nomes começando com letra maiúscula. Ex: *fmt.Println(), time.Now(), strconv.FormatInt()*
- Consequência 2: especificadores de acesso como *private* e *public* são implícitos

```
package compravenda

import (
      "fmt"
)

var idGenerator int = 0;

type Venda struct {
     id int
     meioDePgmto MeioDePagamento
     produto *Produto
     quantidade int
}

func (v *Venda) GetID() int {
     return v.id
}
```

Tempo de Vida e o Coletor de Lixo

- O tempo de vida de variáveis e estruturas é definido pelo coletor de lixo, que se encarrega de desalocar espaços de memória não mais utilizados pelo programa.
- Dessa forma, é possível escrever códigos como o ao lado, sem perda de referências.
- O coletor de lixo usa uma versão melhorada do algoritmo marcar-varrer.

```
package main

import "fmt"

type pessoa struct {
nome string
idade int
}

func (p pessoa) String () string {
      return fmt.Sprintf("%s tem %d anos de
idade.",p.nome,p.idade)
}

func NewPessoa (nome string, idade int) *pessoa{
     return &pessoa{nome,idade}
}

func main() {

   p := NewPessoa("Matheus",21)

   fmt.Println(*p)
}
```

Tipagem

- Go tem tipagem estática. Uma vez que uma variável recebe um tipo, esse tipo não pode mudar durante o programa.
- O tipo pode ser declarado ou pode ser inferido a partir do contexto.
- O código ao lado não compila por causa da linha *'p := 3.19'*

```
package main

import "fmt"

type pessoa struct {
nome string
idade int
}
func main() {

   var p1 pessoa = pessoa{"Joaozinho",10}

   p := pessoa{"Matheus",21}

   p = 3.19

   fmt.Println(p1)
   fmt.Println(p)
}
```

Tipos Primitivos

bool	byte	complex64	complex128	error	float32	float64
int	int8	int32	int64	rune	string	uintptr
uint	uint8	uint16	uint32	uint64		

- Go é formatado usando-se UTF-8, e suporta "code-points" em Unicode (*rune)*
- Todos os tipos tem um *valor-zero* associado (int: 0, string: "", bool: *false*)
- *int* tem o mesmo tamanho de *uint*, que pode ser 32 ou 64 bits.
- Floats são codificados usando-se IEEE-754
- *uintptr*: tem o tamanho necessário para representar um endereço de memória
- *byte:* apelido para *uint8*
- *complex64* tem partes real e complexa do tipo *float32*
- complex128 usa *float64*
- Go não tem: *char, decimal, enum ou void*

Tipos Compostos

- Apresenta arrays, slices e mapas, como citado anteriormente.
- Não tem conjunto potência ou uniões.
- Não suporta estruturas recursivas, mas há solução para isso.

```
type pessoa struct {
    irmao pessoa
    nome string
    idade int
}
```

Proibido
(invalid recursive type pessoa)

```
type pessoa struct {
    irmao *pessoa
    nome string
    idade int
}
```

Válido

Constantes e Tipos Enumerados

- Go não tem tipos enumerados de forma explícita.
- Ao invés disso, usa constantes e *iota*
- *Iota* é do tipo int, e é resetado para 0 a cada bloco de declaração de constantes
- Problema: Como restringir intervalo de valores válidos?

```
type MeioDePagamento int

const(
    CHEQUE MeioDePagamento = iota
    FIADO
    CREDITO
    DEBITO
    DINHEIRO
    TICKET
)
```

Tipagem forte e conversão

- Go é fortemente tipada, logo, todo e qualquer erro de tipos é detectado em tempo de compilação ou de execução.
- Além disso, não existe coerção em Go. Todas as conversões de tipos são explícitas.
- O código ao lado não compila pois contém erros de conversão de tipos.

```go
package main

import (
    "fmt"
    "math"
)

func main() {

    var x, y int = 3, 4

    var certo float64 = math.Sqrt(float64(x*x + y*y))

    // Erro: cannot use x * x + y * y (type int) as type
    // float64 in argument to math.Sqrt
    var errado float64 = math.Sqrt(x*x + y*y)

    var conversaoCorreta int = int(certo)

    // Erro: invalid operation: x + certo (mismatched types
    // int and float64)
    fmt.Println(x + certo)
}
```

Tipagem forte e o “void” de Go

- Problema clássico em C: ponteiro para void
- Como interfaces são implementadas implicitamente, todos os tipos implementam a “interface vazia”. Logo, podemos usar *interface{}* para nos referirmos a qualquer tipo de dado.
- Type casts de *interface{}* são feitos por meio de *type assertions*
- Type assertion de uma variável *x* para o tipo *T*: x.(T)
- Retorna 2 valores: o primeiro é do tipo T e o segundo é um booleano
- Caso a informação armazenada em x seja do tipo T, então o valor de x convertido para T será retornado, juntamento com *true*. Caso contrário, o retorno é o valor-zero do tipo T e *false*.

```go
package main

import "fmt"

func main() {

    array := [5]interface{}{1,"matheus",1.56,fmt.Println,true}

    for _,k := range array {

            //type assertion do tipo int
            if _ , ehString := k.(int) ; ehString {

                fmt.Println(k,"eh um int")

            }else{

                fmt.Println(k,"nao eh um int")
            }
    }
}
```

Output:

```
1 eh um int
matheus nao eh um int
1.56 nao eh um int
0x45aea0 nao eh um int
true nao eh um int
```

Persistência e Serialização

- Go apresenta biblioteca para interação com bancos de dados: database/sql
- Suporta Postgres, MySQL, Oracle, DB2, MS SQL Server, entre outros. (Lista de bancos de dados compatíveis)
- Além disso, também implementa serialização através da biblioteca encoding, com suporte a vários de tipos de codificação: base64, binary, CSV, JSON, XML, e mais.

Alocação de memória

- Em Go, existem duas primitivas para alocação dinâmica de memória: *new* e *make*
- Conceitos confusos
- *Make* só pode ser usada para inicializar arrays, slices, mapas e channels, e retorna uma estrutura alocada dinamicamente
- *New* é usada pra todo o resto dos casos e retorna um ponteiro para uma posição de memória (toda nula)

```go
package main

func main() {

    // allocates slice structure; *p == nil; rarely useful
    var p1 *[]int = new([]int) var v1  []int = make([]int, 100)
    // the slice v now refers to a new array of 100 ints

    // Desnecessariamente complexo:
    var p2 *[]int = new([]int)
    *p2 = make([]int, 100, 100)

    // Idiomático:
    v2 := make([]int, 100)
}
```

Curto-circuito e Efeitos Colaterais

- Go tem curto-circuito em expressões condicionais
- Algumas funções tem efeitos colaterais (leitura de um arquivo ou do terminal, por exemplo, avança o cursor automaticamente).
- Porém o nosso favorito não é permitido: ++ com expressões
- Consequência: Não existe ++ ou -- pré-fixados. Não faz sentido.

```go
package main

import (
    "fmt"
)

func main() {

    var a = 2
    var b = 10

    a++
    fmt.Println(a)
    a--
    fmt.Println(a)

    // Erro: syntax error: unexpected --, expecting )
    var c = (a--)*b

    // Erro: syntax error: unexpected ++, expecting )
    if (a < b) || (a == b++) {
        fmt.Println(true)
    }

}
```

Modularização

- Go permite criação de funções por parte do usuário (mostrado anteriormente)
- Funções podem ser declaradas em arquivos separados
- Todo arquivo tem que ter um pacote (*package)* correspondente
- Chamadas externas de funções e estruturas de um pacote são feitas usando-se:
  <nome do pacote> . <identificador da função/estrutura>
- Exemplo: fmt.Println()
- A função main fica no pacote de mesmo nome
- A passagem de parâmetros é somente por cópia
- A momento da passagem de parâmetros é normal.

“Herança”

- Go não é orientado a objetos, mas consegue simular algumas das características OO.
- Go favorece composição ao invés de herança, e faz isso por meio de campos anônimos

Output:

cachorro
Pastor alemão
Eu sou um cachorro
Eu sou um Pastor Alemão

```
package main

import "fmt"

type Animal struct {
    especie string
}

func (a Animal) String() string {
     return fmt.Sprint("Eu sou um ",a.especie)
}

type Cachorro struct {
    Animal
    raça string
}

func (c Cachorro) String() string {
    return fmt.Sprint("Eu sou um ",c.raça)
}

func main() {

    c := Cachorro{Animal: Animal{"cachorro"},
raça: “Pastor Alemão"}

    fmt.Println(c.especie)
    fmt.Println(c.raça)

    fmt.Println(c.Animal)
    fmt.Println(c)
}
```

Exceções (?)

- Go não implementa um sistema de exceções como outras linguagens o fazem.
- Já que funções podem ter mais de um valor de retorno, por que não usar esse poder?
- Sempre retornados como último parâmetro da função
- Função externa checa se houve erro
- Erros implementam a interface *error*
- Se uma função retornar mais de um erro, usar polimorfismo e type assertion

Tratamento de erros

```go
package main

import "errors"
import "fmt"

func alistarNoExército(idade int) (string, error) {

    if idade < 18  {
        return "Rejeitado" , &NovoDemais{idade}
    }else if idade > 100{
        return "Rejeitado", errors.New("Não queremos
ninguém com mais de 100 anos")
    }

    return "Bem-vindo", nil
}

type NovoDemais struct {
    idade  int
}
```

```go
func (nd *NovoDemais) Error() string {

    return fmt.Sprintf("Faltam %d anos para você poder
se alistar.", 18 - nd.idade)
}

func main() {

    c := [3]int{12,150,20}

    for i := range c {

    if r , ok := alistarNoExército(c[i]) ; ok != nil {
            fmt.Println(r,"-> causa:",ok)
    }else{
            fmt.Println(r)
    }
    }

}
```

Output:

Rejeitado -> causa: Faltam 6 anos para você poder se alistar.
Rejeitado -> causa: Não queremos ninguém com mais de 100 anos
Bem-vindo

Bibliotecas

- Go tem MUITAS bibliotecas já embutidas na linguagem.
- Dentre elas, podemos citar:
  - compress: contém funções para compactação de arquivos
  - crypto: implementação de diversos tipos de criptografia
  - sync: ferramentas para sincronização (mutexes, semáforos, etc)
  - flag: funções para tratamento de flags de entrada de programas
  - image: biblioteca para lidar com imagens 2D (ex. JPEDG, GIF e PNG)
  - math: funções matemáticas e big numbers
  - net: funções para implementação de protocolos de internet
  - os: interação com o SO (qualquer SO)
  - unsafe: funções que “pulam a cerca” do sistema de tipos
  - Dentre muitas outras...

Características OO

- Apesar de Go ser estruturada, ela consegue simular características OO melhor do que C.
- Classes: structs com métodos
- Encapsulamento: variáveis exportadas ou não
- Modificadores de acesso: apenas package private ou public
- Herança: ao invés de herança, composição (campos anônimos)
- Polimorfismo: através de interfaces

Comparações

Critérios	Go	C	Java
Aplicabilidade	Parcial	Sim	Parcial
Confiabilidade	Parcial	Não	Sim
Aprendizado	Não	Não	Não
Portabilidade	Não	Não	Sim
Método de Projeto	Estruturado	Estruturado	OO
Evolutibilidade	Sim	Não	Sim
Reusabilidade	Sim	Parcial	Sim
Integração	Parcial	Sim	Parcial
Custo	Hoje, alto	Depende	Depende
Escopo	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim
Tipos primitivos e compostos	Sim	Sim	Sim
Persistência	Diversas formas	Biblioteca de funções	JDBC, biblioteca de classes, serialização

Critérios	Go	C	Java
Encapsulamento de proteção	Sim	Parcial	Sim
Sistema de tipos	Sim (Fortemente Tipada)	Não	Sim
Verificação de tipos	Estática / Dinâmica	Estática	Estática / Dinâmica
Polimorfismo	Inclusão	Coerção e Sobrecarga	Todos
Exceções	Parcial	Não	Sim
Concorrência	Sim	Não	Sim
Eficiência	The Computer Languages Benchmarks Game		

Conclusão

Considerações finais

- Go é uma linguagem promissora.
- Apesar de recente, já demonstra certa maturidade e tem muitas ferramentas interessantes.
- É de fácil aprendizado para programadores C, por exemplo.
- Grandes investimentos por parte do Google (e.g. Kubernetes)
- Opinião pessoal: C melhorado e mais robusto

Referências e por onde começar

- Página oficial: Golang.org
- Tutorial oficial: A Tour of Go
- Especificação da Linguagem: The Go Programming Language Specification
- Dicas para boa programação em Go: Effective Go
- Tutorial da linguagem através de exemplos: Go By Example

Já representam um bom começo...

Trabalho