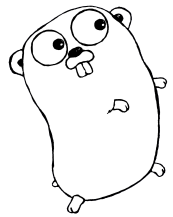
Linguagem *Go*



MAC5742/0219
Introdução à Programação Concorrente, Paralela e Distribuída

Eduardo Gonçalves Pinheiro
Leonardo Pereira Macedo
Ricardo Akira Tanaka

Introdução

- Desenvolvida na **Google**, em 2007, mas lançada em 2012
- Buscava resolver críticas de outras linguagens comuns
- Preocupação com **redes** e **multiprocessamento**

Criadores: Da esquerda para a direita, Robert Griesemer, Rob Pike e Ken Thompson

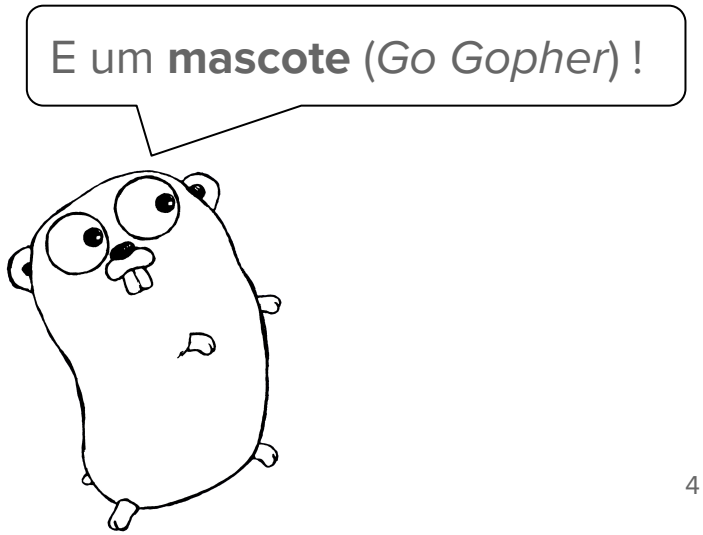
Fonte: http://oylenshpeegul.github.io/asopl/Griesemer_Pike_Thompson-Go.jpg

Características principais

- Parcialmente inspirada em **C**
- Linguagem compilada
- Estaticamente tipada
- Coleta de lixo
- Ausência de POO
- Preocupação com problemas de acesso à memória (ponteiros, *buffers*)
- Programação concorrente **CSP** (*Communicating Sequential Processes*; C.A.R. Hoare, CACM 1978)
- *Share memory by communicating*
  - Oposto a *communicate by sharing memory*

Características principais

- Parcialmente inspirada em **C**
- Linguagem compilada
- Estaticamente tipada
- Coleta de lixo
- Ausência de POO
- Preocupação com problemas de acesso à memória (ponteiros, *buffers*)
- Programação concorrente **CSP** (*Communicating Sequential Processes*; C.A.R. Hoare, CACM 1978)
- *Share memory by communicating*
  - Oposto a *communicate by sharing memory*



Fonte: https://golang.org/doc/gopher/gopherbw.png

Principais usos de *Go*

- Linguagem de propósito geral
- Desempenho semelhante a ***Java***
- Inadequada para *software* em tempo real
- Boa estrutura para servidores com elevada **concorrência**
  - Ferramentas poderosas, intuitivas e simples (*goroutines*, *channels*)
  - Nesse sentido, capaz de competir com *threads* em **C++**

Hello, world!

```go
package main

import "fmt"

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

- Todo programa em *Go* é constituído por **pacotes**

Funções

- São de **primeira classe**
- Podem devolver mais de um valor, ou até funções

```
func troca(aqui, ali string) (string, string) {
    return ali, aqui
}
```

Funções

- São de **primeira classe**
- Podem devolver mais de um valor, ou até funções

```
func troca(aqui, ali string) (string, string) {
    return ali, aqui
}
```

- As variáveis de retorno podem ser **nomeadas**

```
func triplica(x int) (triplo int) {
    triplo = 3*x
    return  // Naked return
}
```

Funções

- Podem ser ***closures***
- Carregam suas variáveis

```
func intSeq() func() int {
    i := 0
    return func() int {
        i += 1
        return i
    }
}
```

```
func main() {
    nextInt := intSeq()
    fmt.Println(nextInt()) // 1
    fmt.Println(nextInt()) // 2
    fmt.Println(nextInt()) // 3

    newInts := intSeq()
    fmt.Println(newInts()) // 1
}
```

Variáveis

- Declaradas com **var**, ou com a notação **:=**
- Não há conversão implícita entre tipos
- A linguagem possui **escopo estático**

```
var x int
var y = 1  // y é inferido como int

func main() {
    str, pi, n := "oi", 3.14, 2
    math.Sqrt(float64(n))  // Sqrt() requer um float64
}
```

Ponteiros

- Funciona da mesma forma que **C**

```
func main() {
    i := 42
    p := &i
    *p = 21
}
```

- Aritmética de ponteiros não é permitida

```
p++
```

Struct

- Permite criação de novos tipos de dados
- Campos são acessados por um ponto (.)

```
type Vertex struct {
    X int
    Y int
}

V := Vertex{3, 4}  // Inicializa X e Y
fmt.Println(v.X)   // 3
```

- Palavra reservada **type** permite nomear uma *struct*
- Pode ser referenciada por um ponteiro

Slice

- *Arrays* possuem tamanho fixo
- *Slices* são referências
- **Tamanho** e **capacidade** acessíveis por `len(slice)` e `cap(slice)`

```
func main() {
    primes := []int
              {2, 3, 5, 7, 11}
    slc := primes[0:]  // [2 3 5 7 11]
    slc = primes[:4]   // [2 3 5 7]
}
```

Slice

- **`make(tipo, tamanho, capacidade)`** aloca *array* e devolve referência

```
slc := make([]int, 0, 2)
fmt.Println(slc, len(slc), cap(slc))  // [] 0 2
```

- **`append(slice, valores)`** insere, e pode redimensionar

```
slc = append(slc, 2, 3, 4)
fmt.Println(slc, len(slc), cap(slc))  // [2 3 4] 3 4
```

Métodos

- Equivalente a funções, mas com argumento *receiver*
- *Receiver* deve estar declarado internamente no pacote
- *Receiver* pode ser ponteiro, altera por referência

```
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

var v Vertex
v.Scale(5)  // Interpretado como (&v).Scale(5)
p := &v
p.Scale(10)
```

Goroutines

- *Thread* leve, gerenciada pelo programa *Go*
- Criada com a palavra reservada **go**
- Recebe uma chamada de função como argumento

```
func imprime_soma(a, b int) int {
    fmt.Println(a + b)
}

func main() {
    go imprime_soma(10, 32)
}
```

Channels

- Condutos tipados, usados para comunicação entre *goroutines*

```
ch := make(chan int)  // Cria um channel para o tipo int
```

Channels

- Condutos tipados, usados para comunicação entre *goroutines*

```
ch := make(chan int)  // Cria um channel para o tipo int
```

- Usa-se `<-` para enviar ou receber valores pelo *channel*

```
ch <- x    // Envia x para o channel ch
x := <-ch  // Recebe de ch, e guarda em x
```

- *Channels* **bloqueiam** até o outro lado receber/enviar algo

Channels com *buffer*

- Guarda até um certo número de elementos em um *buffer*
- Envios bloqueiam se o *buffer* estiver **cheio**
- Recepções bloqueiam se o *buffer* estiver **vazio**

```
ch := make(chan int, 2)  // Buffer para 2 ints
ch <- 1
ch <- 2
ch <- 3  // Cria um deadlock (buffer já cheio!)
fmt.Println(<-ch)
fmt.Println(<-ch)
```

Outras operações com *channels*

- `close()` fecha um *channel* (não haverá mais **envio** de valores)
- Pode-se **receber** valores **repetidamente** de um *channel* com:

```
for i := range ch {}
```

Outras operações com *channels*

- **`close()`** fecha um *channel* (não haverá mais **envio** de valores)
- Pode-se **receber** valores **repetidamente** de um *channel* com:

```
for i := range ch {}
```

- É possível "ouvir" vários *channels* simultaneamente com **`select`**

```
x := 0
select {
    case ch <- x:
        x++
    case <-ch_tchau:
        return
    default:  // Todos os channels estão bloqueados
        fmt.Println("Tudo bloqueado!")
}
```

Formas de concorrência

> *Do not communicate by sharing memory; instead, share memory by communicating.*

- Variáveis compartilhadas
  - Passagem pelos *channels*, nunca compartilhada por *threads*
  - *Race condition* impossível
  - Análogo ao *Unix Pipeline*
- *Goroutines*
  - Funções concorrentes num mesmo espaço de endereçamento
  - Multiplexadas em threads do sistema operacional
  - Análogo ao **&** do *Unix Shell*
- *Channels*
  - Uso de *buffer* serve como semáforo (limita *throughput*)

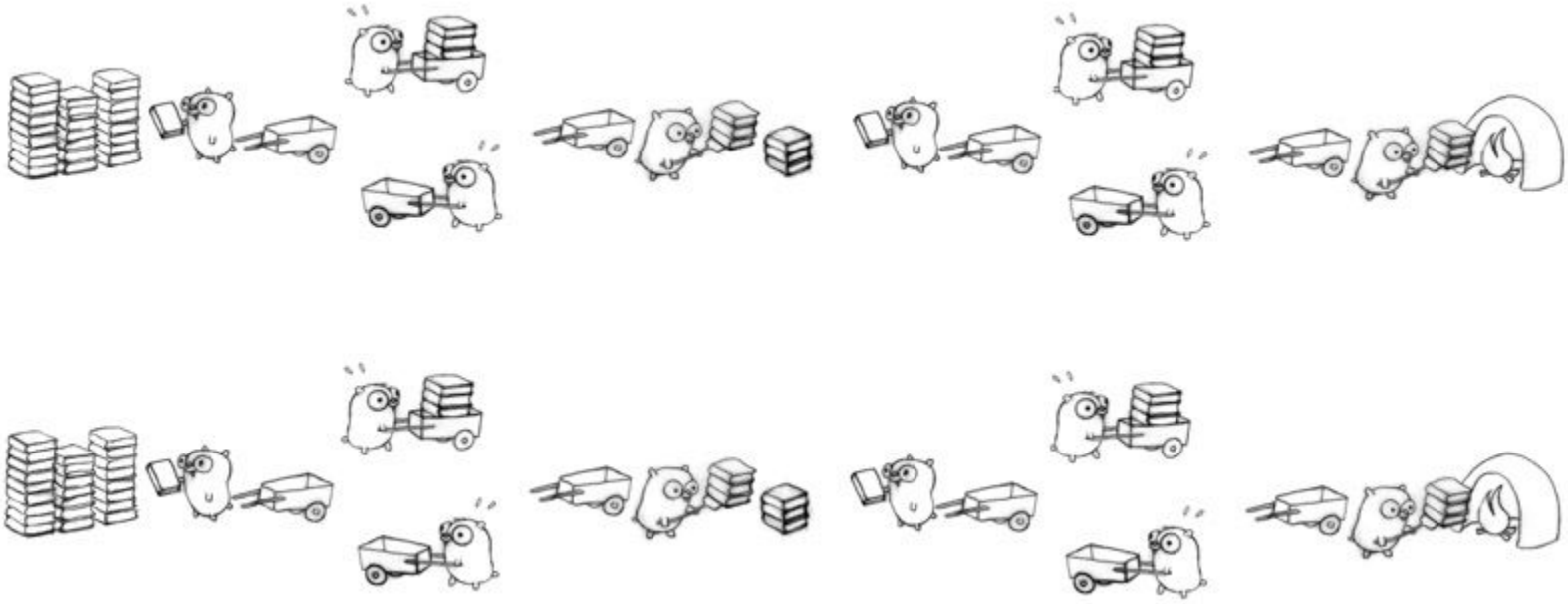
Paralelismo e concorrência

Concurrency is about dealing with lots of things at once.
Parallelism is about doing lots of things at once.

- **Concorrência** é a composição de processos independentes em execução
- **Paralelismo** é a execução simultânea de computações (possivelmente relacionadas)

Concorrência não é paralelismo

- Exemplo do Rob Pike na *Heroku's Waza conference* (2013)



Concurrent design for a scalable web service; gophers serving web content

Fonte: https://talks.golang.org/2012/waza/gophercomplex6.jpg

Go x **Rust**

- Sucessores de **C**
- Preocupação com segurança (problemas com *buffers*/ponteiros em **C**)

Go

- Mais simples de aprender (feita para ser uma atualização de **C**)
- Possui CSP integrada na linguagem, e ferramentas mais simples para concorrência
- Mais lento que *Rust* (coleta de lixo)

Rust

- *Ownership* e outras propriedades aumentam a curva de dificuldade
- CSP através de primitivas, *mutex* através de *data-ownership*
- Rápida e robusta (gerenciamento manual de memória)

Referências

- Site oficial da linguagem *Go*
  - https://golang.org
- Tutoriais passo a passo da linguagem
  - https://tour.golang.org
- História e características principais
  - https://en.wikipedia.org/wiki/Go_(programming_language)
- Principais usos de *Go*
  - https://www.quora.com/For-which-purpose-is-the-Go-programming-language-used
  - https://www.quora.com/How-is-Go-used-at-Google
- Tempo de execução de *Go* comparado a outras linguagens
  - http://benchmarksgame.alioth.debian.org
- Rob Pike - *Concurrency is not Parallelism*
  - http://talks.golang.org/2012/waza.slide
- Comparação entre *Go* e *Rust*
  - https://blog.ntpsec.org/2017/01/18/rust-vs-go.html