Introdução a Go para programação paralela

MO644 — Programação Paralela

Alexandre Medeiros

alexandre.n.medeiros@gmail.com

Instituto de Computação
Universidade Estadual de Campinas

10 de junho de 2014

Estes slides e os códigos usados na apresentação podem ser encontrados na minha página!

`alemedeiros.sdf.org/posts/2014-06-10-go-intro.html`

O que é Go?

Golang

Go, ou Golang, é uma linguagem de programação *Open Source* onde é fácil fazer programas simples, confiáveis e eficientes.

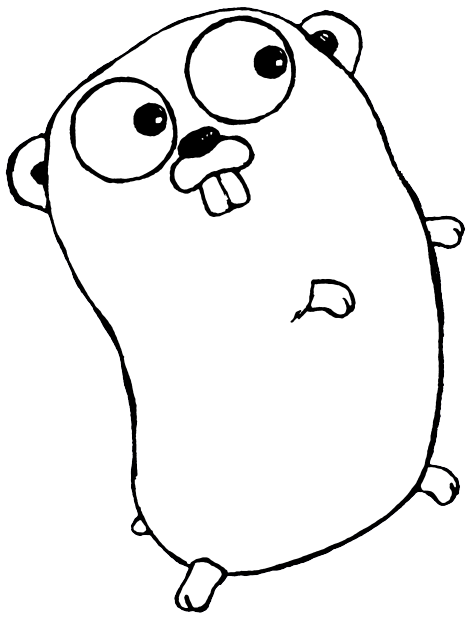
O que é Go?

Criadores

A linguagem foi desenvolvida por Robert Griesemer, Rob Pike e Ken Thompson no Google.

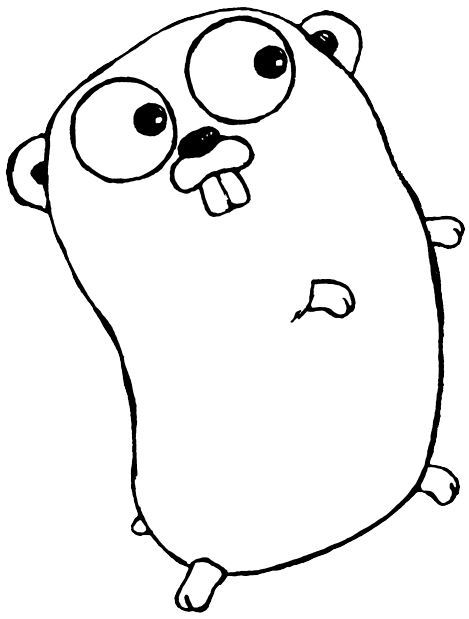
Principais ideias

- Compilação rápida
- Execução rápida
- Simples de se programar



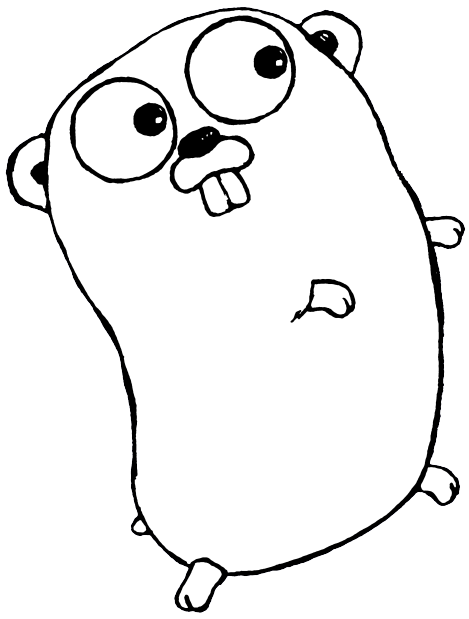
Principais ideias

- Compilação rápida
- Execução rápida
- Simples de se programar



Principais ideias

- Compilação rápida
- Execução rápida
- Simples de se programar



Funcionalidades interessantes

- Sintaxe inspirada em C
- *Garbage-collected*
- Inferência de tipos de variáveis

Funcionalidades interessantes

- Sintaxe inspirada em C
- *Garbage-collected*
- Inferência de tipos de variáveis

Funcionalidades interessantes

- Sintaxe inspirada em C
- *Garbage-collected*
- Inferência de tipos de variáveis

E mais importante!

Projetada para ser simples de se criar programas concorrentes.

E mais importante!

Projetada para ser simples de se criar programas concorrentes.

Exemplos

Vejamos como Go funciona na prática!

Hello, World!

```
package main

import "fmt"

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

Programa 1: hello.go

```
$ go run hello.go
Hello, World!
```

goroutines

Uma `goroutine` é uma *lightweight thread*, gerenciada pelo *runtime*.

goroutines

```
func say(s string) {
    for i := 0; i < 3; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("World")
    say("Hello")
}
```

Programa 2: goroutine.go

goroutines

```
$ go run goroutine.go
World
Hello
World
Hello
World
Hello
```

Canais de comunicação

Os `channels` são a principal maneira de comunicação e sincronização entre `goroutines`.

Canais de comunicação

```
func say(s string, done chan bool) {
    for i := 0; i < 3; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
    done <- true
}

func main() {
    done := make(chan bool)
    go say("Hello, World!", done)

    if <-done {
        fmt.Println("Done :)")
    }
}
```

Programa 3: channels.go

Canais de comunicação

```
$ go run channels.go
```

```
Hello, World!
Hello, World!
Hello, World!
Done :)
```

Comunicação não bloqueante

Para que a `goroutine` não fique bloqueada na hora de enviar dados, é possível criar um canal com um *buffer*.

Comunicação não bloqueante

```
func produce(id, n int, ch chan<- int) {
    for i := 0; i < n; i++ {
        val := i * id
        ch <- val
        fmt.Println(id, "produced", val)
    }
}

func main() {
    ch := make(chan int, 2)

    go produce(42, 4, ch)

    time.Sleep(100 * time.Millisecond)
    for i := 0; i < 4; i++ {
        val := <-ch
        fmt.Println("Element", i, "is", val)
    }
}
```

Programa 4: buffered-channels.go

Comunicação não bloqueante

```
$ go run buffered-channels.go
42 produced 0
42 produced 42
Element 0 is 0
Element 1 is 42
Element 2 is 84
42 produced 84
42 produced 126
Element 3 is 126
```

Múltiplos canais

O `select` é similar a um switch-case, sua função é receber ou enviar dados com múltiplos canais.

Múltiplos canais

```
func main() {
    ch1 := make(chan int, 2)
    ch2 := make(chan int, 2)

    go produce(1, 4, ch1)
    go produce(2, 4, ch2)

    for i := 0; i < 8; i++ {
        select {
        case val := <-ch1:
            fmt.Println("From 1:", val)
        case val := <-ch2:
            fmt.Println("From 2:", val)
        }
    }
}
```

Programa 5: select.go

Múltiplos canais

```
$ go run select.go
Producer 2 : 0
Producer 1 : 0
Producer 2 : 2
Producer 1 : 1
Producer 1 : 2
From 1: 0
From 1: 1
Producer 1 : 3
Producer 2 : 4
From 2: 0
From 1: 2
From 2: 2
From 2: 4
From 2: 6
Producer 2 : 6
From 1: 3
```

for-each

Como muitas linguagens atuais, Go também possui um for-each.

for-each

```
func main() {
    a := [5]int{1, 2, 3, 4, 5}

    for i, x := range a {
        fmt.Println("Element", i, "is", x)
    }
}
```

Programa 6: range.go

for-each

```
$ go run range.go
Element 0 is 1
Element 1 is 2
Element 2 is 3
Element 3 is 4
Element 4 is 5
```

for-each em um canal

E o for-each também pode ser utilizado em um canal!

for-each em um canal

```
func produce(id, n int, ch chan<- int) {
    for i := 0; i < n; i++ {
        val := i * id
        ch <- val
        fmt.Println(id, "produced", val)
    }
    close(ch)
}

func main() {
    ch := make(chan int, 2)

    go produce(42, 4, ch)

    for val := range ch {
        fmt.Println("Element is", val)
    }
}
```

Programa 7: range-channel.go

for-each em um canal

```
$ go run range-channel.go
42 produced 0
Element is 0
42 produced 42
Element is 42
42 produced 84
Element is 84
42 produced 126
Element is 126
```

Conjunto de ferramentas completo

- go build
- go run
- gofmt
- godoc
- go get

Conjunto de ferramentas completo

- go build
- go run
- gofmt
- godoc
- go get

Conjunto de ferramentas completo

- go build
- go run
- gofmt
- godoc
- go get

Conjunto de ferramentas completo

- go build
- go run
- gofmt
- godoc
- go get

Conjunto de ferramentas completo

- go build
- go run
- gofmt
- godoc
- go get

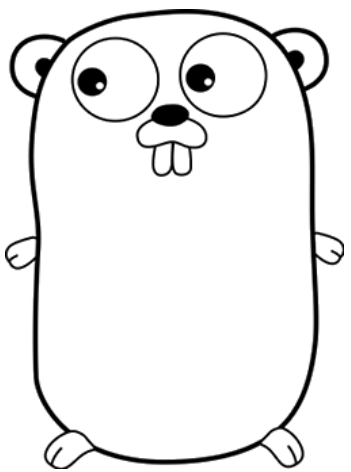
Referências

- `tour.golang.org`
- `gobyexample.com`

Divirta-se você também!

`play.golang.org`

Dúvidas?



Fim!

Contato: alexandre.n.medeiros@gmail.com