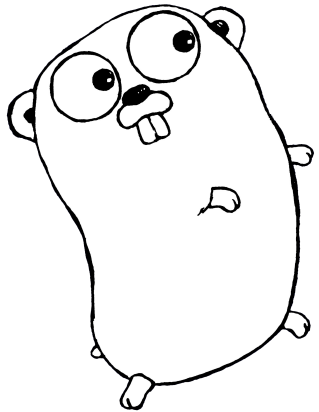
Go

alguns diferenciais (bons ou ruins...)



Pedro Probst | Outubro/2019

Sumário

- *Olá, Mundo!*
- Uma visão geral
- Bom ou ruim?
- Ponteiros
- Goroutines
- Reflexão

Esta apresentação não é um tutorial de Go! Para isso, veja:
https://tour.golang.org/welcome/1

Sumário

- *Olá, Mundo!*
- Uma visão geral
- Bom ou ruim?
- Ponteiros
- Goroutines
- Reflexão

Esta apresentação não é um tutorial de Go! Para isso, veja:
https://tour.golang.org/welcome/1

Olá, mundo!

```go
package main

import "fmt"

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

Uma visão geral

- A linguagem de programação Go foi iniciada na Google em 2007
  - Robert Griesemer, Rob Pike e Ken Thompson
- Primeira versão lançada em 2009
- Motivação
  - Auxiliar no gerenciamento de grandes bases de códigos
  - Manter o desenvolvimento produtivo, com foco em simplicidade
  - Aprendizagem fácil e utilidade prática
- Inspirada em especial pela linguagem de programação C
- Uma das primeiras linguagens projetadas com *multithreading* em mente
  - 2007… processadores *dual-core* já eram uma “coisa”

Features

- Programação concorrente relativamente descomplicada
- Design simples
  - a especificação da linguagem tem poucas páginas.
- Tipagem forte e estática
- Compilada
- Produz binários executáveis com facilidade
  - `$ go build`
  - `$ go install`
- Suporte a testes

Features

- Rica biblioteca padrão
  - https://golang.org/pkg/
- Coleta de lixo
- Facilmente portável para sistemas diferentes
  - Do rPi ao Plan 9...
- Estilo de formatação uniformizado
  - `$ go fmt`
- Ponteiros
- Inferência de tipos (limitada; e.g. `:=`)

Features

- Relativamente rápida
- Boa adoção pela comunidade/indústria
  - Índice 14 no TIOBE Index (Setembro/2019) - https://www.tiobe.com/tiobe-index/go/
    - Uma posição acima de Ruby
- (. . .)

O que Go não tem?

- Classes
- Genéricos
- Um bom sistema de gerenciamento de pacotes
  - Quero instalar um pacote:
    - `$ go get endereço/do/pacote/na/web`
    - Pacote será instalado num diretório especificado no `GOPATH`.
      - Um tanto simples demais. Torna-se inconveniente lidar com diferentes versões.
- Um bom sistema de tratamento de erros
  - `if err != nil {...} // Ahhhhhhhh`
  - Código pode se tornar um espaguete de tratamento de erros!

O que Go não tem?

- Classes
- Genéricos
- Um bom sistema de gerenciamento de pacotes
  - Quero instalar um pacote:
    - `$ go get endereço/do/pacote/na/web`
    - Pacote será instalado num diretório especificado no `GOPATH`
      - Um tanto simples demais. Torna-se inconveniente lidar com diferentes versões.
- Um bom sistema de tratamento de erros
  - `if err != nil {...} // Ahhhhhhhh`
  - Código pode se tornar um espaguete de tratamento de erros!

Opiniões!

PONTEIROS *

Ponteiros

- Um ponteiro carrega o endereço de memória de um valor
- Mas, diferentemente de C, Go não tem aritmética de ponteiros
  - Por quê? (respostas do próprio Go F.A.Q.)
    - Facilita na implementação do coletor de lixo
    - Segurança
      - As tecnologias de compiladores/hardware avançou ao ponto em que um *loop* usando índices de matriz pode ser tão eficiente quanto um loop usando aritmética de ponteiro.

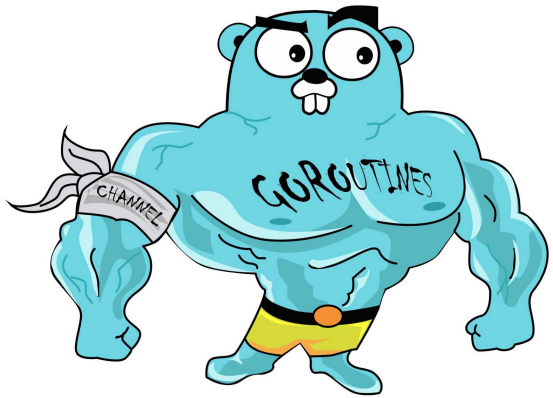
Ponteiros

- Um ponteiro carrega o endereço de memória de um valor
- Mas, diferentemente de C, Go não tem aritmética de ponteiros
  - Por quê? (respostas do próprio Go F.A.Q.)
    - Facilita na implementação do coletor de lixo
    - Segurança
      - As tecnologias de compiladores/hardware avançou ao ponto em que um *loop* usando índices de matriz pode ser tão eficiente quanto um loop usando aritmética de ponteiro.

Pode ser contornado com o uso da biblioteca “`unsafe`” (não recomendável!)

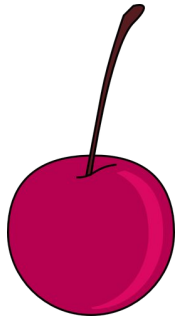
Ponteiros

```go
func main() {
    i, j := 16, 32

    p := &i          // p aponta para o endereço de i
    fmt.Println(*p)  // imprime 16

    *p = 8           // muda o valor de i (16) para 8

    p = &j           // p agora aponta para j
    *p = *p / 32     // 32 / 32
    fmt.Println(j)   // imprime 1
}
```



GOROUTINES

Goroutines



- Uma goroutine é, em suma, uma thread leve de execução
- É a “cereja do bolo” em Golang
  - Não quer dizer que necessariamente precise ser usada…
- Programação concorrente de modo simples e descomplicado
- As goroutines são executadas no mesmo espaço de endereço, portanto, o acesso à memória compartilhada deve ser sincronizado!
  - Pacote “`sync`”: `Mutex`, `WaitGroup`, `Broadcast`, `Signal`, `Wait`…
  - Ou usando o tipo primitivo `chan`, que fornece comunicação entre goroutines

Goroutines

```
 1 func say(s string) {
 2     for i := 0; i < 5; i++ {
 3         time.Sleep(100 * time.Millisecond)
 4         fmt.Println(s)
 5     }
 6 }
 7
 8 func main() {
 9     go say("world")
10     say("hello")
11 }
```

Goroutines

```
 ~ go run a.go
world
hello
world
hello
hello
world
world
hello
hello
```

“`world`” é o retorno da goroutine (keyword `go`);

“`hello`” é o retorno da função “normal”.

Goroutines

- Vamos mudar as coisas um pouco!
- Comentando a linha `10 say(“hello”)` e rodando o programa novamente...
  - Nada é impresso na tela!
    - Por quê?

Goroutines

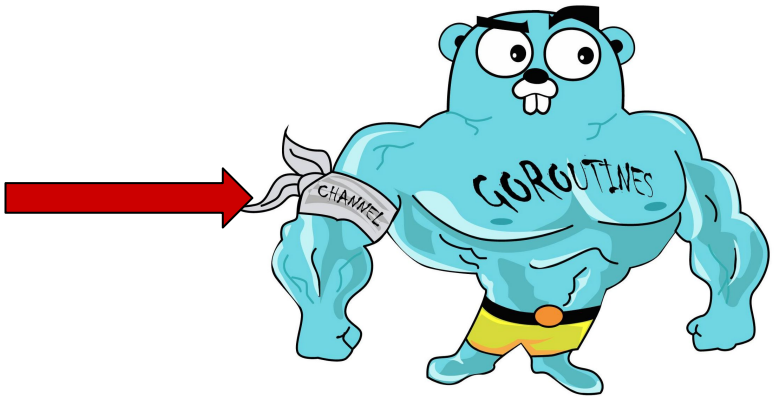
- Vamos mudar as coisas um pouco!
- Comentando a linha `10 say(“hello”)` e rodando o programa novamente...
  - Nada é impresso na tela!
    - Por quê?
      - Nossa goroutine é independente e desconhecida pela thread principal

Goroutines

- Vamos mudar as coisas um pouco!
- Comentando a linha `10 say(“hello”)` e rodando o programa novamente...
  - Nada é impresso na tela!
    - Por quê?
      - Nossa goroutine é independente e desconhecida pela thread principal
      - Assim, a thread principal termina antes dos valores serem impressos...

Goroutines

- Vamos mudar as coisas um pouco!
- Comentando a linha `10 say(“hello”)` e rodando o programa novamente...
  - Nada é impresso na tela!
    - Por quê?
      - Nossa goroutine é independente e desconhecida pela thread principal
      - Assim, a thread principal termina antes dos valores serem impressos...
      - Será que existe uma maneira de solicitar que a thread principal aguarde a goroutine?



CHANNELS

Channels

- Channels são canais pelos quais você pode enviar e receber valores com o operador de canal <-
  - Os dados fluem na direção da setinha!

```
ch <- v    // Envie “v” para o channel “ch”.

v := <-ch  // Receba de “ch”, e atribua um valor a “v”.
```

- Por padrão, enviar/receber “bloqueia” até que o outro lado esteja pronto. Isso permite que as goroutines sejam sincronizadas sem bloqueios explícitos ou variáveis de condição...

Channels

- Podemos fazer uma goroutine recém criada avisar a goroutine principal que determinada tarefa já terminou...
  - Garantindo sincronismo!
- A *goroutine* principal irá “bloquear” até receber uma mensagem da função `say` que está executando em sua própria goroutine

Channels

```
1  func say(s string, done chan string) {
2      for i := 0; i < 5; i++ {
3          time.Sleep(100 * time.Millisecond)
4          fmt.Println(s)
5      }
6      done <- “Terminei”
7  }
8
9  func main() {
10     done := make(chan string)
11     go say(“world”, done)
12     fmt.Println(<-done)
13 }
```

Channels

```
 ~ go run a.go
world
world
world
world
world
Terminei
```

Goroutines e Channels

- Há muitos outros detalhes sobre goroutines e channels que não serão abordados aqui.
  - Vale a pena dar uma pesquisada por conta própria, se tiver interesse em aprender Go.
    - Os exemplos mostrados foram bem básicos, com o intuito de mostrar uma capacidade da linguagem, e não o poder das goroutines...

REFLEXÃO

Reflexão

- Go suporta reflexão *out-of-the-box* a partir do pacote `reflect`, presente na biblioteca padrão.
- *“Reflection in computing is the ability of a program to examine its own structure, particularly through types; it's a form of metaprogramming. It's also a great source of confusion.”*
  - *https://blog.golang.org/laws-of-reflection (Rob Pike, 6 de setembro de 2011)*

Reflexão

- Reflexão é um tópico um tanto confuso demais para uma apresentação de 20 minutos:
    - Então, vamos dar apenas uma “pincelada” no assunto... também para evitar que o autor desta apresentação passe vergonha ao falhar em explicar conceitos básicos que ele já deveria saber...

Reflexão

- Reflexão em Go é formada sobre três conceitos básicos
  - *Types*, *Kinds* e *Values*

Você pode usar reflexão para pegar o tipo de uma variável com a chamada de função `varType := reflect.TypeOf(var)`. Isso retorna uma variável do tipo `reflect.Type`, que possui métodos com todo o tipo de informação que define a variável passada.

Sobre essa variável podemos usar, por exemplo, `Name()`, que retorna o nome do tipo.

Reflexão

Outro método muito útil é `Kind()`, que retorna “do que o tipo é feito” -- uma *slice*, um *map*, uma *struct*...

- A diferença entre o tipo e o “Kind” pode ser difícil de entender, mas pense da seguinte forma:
  - Se você definiu uma estrutura de nome *Foo*, o tipo é *Foo* e o “Kind” é *struct*.

Reflexão - *get* e *set* em campos de uma struct

```go
type User struct {
    Email  string `mcl:"email"`
    Name   string `mcl:"name"`
    Age    int    `mcl:"age"`
    Github string `mcl:"github" default:"anono"`
}

func main() {
    u := &User{Name: "Anon"}
    // Elem retorna o valor para o qual o ponteiro u aponta.
    v := reflect.ValueOf(u).Elem() // { Anon 0 }
    f := v.FieldByName("Github")
    // Tenha certeza de que o campo é definido e pode ser alterado!
    if !f.IsValid() || !f.CanSet() {
        return
    }
    if f.Kind() != reflect.String || f.String() != "" {
        return
    }
    f.SetString("lalala")
    fmt.Printf("Github username was changed to: %q\n", u.Github)
}
```

Reflexão

```
 ~ go run a.go
Github username was changed to: "lalala"
```

Reflexão - criando uma função em *runtime*

```go
package main

import (
    "fmt"
    "reflect"
)

type Add func(int64, int64) int64

func main() {
    t := reflect.TypeOf(Add(nil)) // main.Add
    mul := reflect.MakeFunc(t, func(args []reflect.Value) []reflect.Value {
        a := args[0].Int()
        b := args[1].Int()
        return []reflect.Value{reflect.ValueOf(a+b)}
    })
    fn, ok := mul.Interface().(Add)
    if !ok {
        return
    }
    fmt.Println(fn(2,3))
}
```

Reflexão - criando uma função em *runtime*

```
 ~ go run a.go
5
```

FIN

Referências

- https://golang.org/doc/faq
- https://www.freecodecamp.org/news/here-are-some-amazing-advantages-of-go-that-you-dont-hear-much-about-1af99de3b23a/
- https://abdullin.com/golang/
- https://www.mindinventory.com/blog/what-makes-golang-stand-apart-from-other-languages/
- https://www.ionos.com/digitalguide/server/know-how/golang/
- https://www.xoriant.com/blog/product-engineering/go-programming-language-key-features.html
- https://medium.com/@jamesotoole/golang-and-why-it-matters-1710b3af96f7
- https://medium.com/my-new-roots/the-gopher-way-681a70f3bc79
- https://medium.com/capital-one-tech/learning-to-use-go-reflection-822a0aed74b7

->

Referências

- https://medium.com/trainingcenter/goroutines-e-go-channels-f019784d6855
  - Fonte dos exemplos de goroutines
- https://github.com/a8m/reflect-examples
  - Fonte dos exemplos de reflexão

Contato

ppminini@inf.ufsm.br