Go

Um tutorial sobre a linguagem

Go

- História
- Instalação
- Por que Go?
- Operadores
- Valores e tipos
- Tipos de dados primitivos
- Tipos de dados compostos
- Checagem de tipos
- Equivalências de tipos
- Palavras reservadas
- Expressões e comandos
- Variáveis
- Blocos e escopo
- Array, slices e maps
- Abstrações
- Passagem de parâmetros
- Pacotes
- Objetos e classes
- Sistema de tipos
- Monomorfismo
- Sobrecarga
- Polimorfismo por inclusão
- Polimorfismo paramétrico
- Coerção
- Interface e exceções
- Desvios incondicionais
- Escapes
- Concorrência

História

- Foi criada em 2007 como um projeto interno da Google por Rob Pike, Ken Thompson e Robert Griesemer, e lançado como um projeto de código aberto em novembro de 2009.

História

- Foi criada com o objetivo de combinar a facilidade de uma linguagem interpretada e dinamicamente tipada com a eficiência e segurança de uma linguagem compilada e estaticamente tipada,tornando o desenvolvimento de servidores na Google uma tarefa mais produtiva e eficiente.

- Artigo que descreve a dificuldade e a necessidade de criar uma nova linguagem, segundo Rob Pike.
  - *http://talks.golang.org/2012/splash.article*

Instalação

- Os pacotes de distribuição estão disponíveis no site:
  - *http://golang.org/dl/*

Por que Go?

- Implementa um controle rigoroso e inteligente de dependência, baseado na definição e uso de *packages* (pacotes).
- concorrente
- Possui uma sintaxe bastante limpa.
- Simula orientação a objeto
- sensação de uma linguagem dinâmica
- open-source

Por que Go?

- Permite a escrita de programas concisos e legíveis, além de facilitar a escrita de ferramentas que interagem com o código-fonte.
  - *go fmt* (formata o código de acordo com o guia de estilo da linguagem)
  - *go fix* (rescreve partes do código que usa APIs depreciadas para que usem as novas APIs introduzidas em versões mais recentes)

Por que Go?

- Possui tipagem forte e estática.
- Introduz uma forma curta de declaração de variáveis baseadas em inferência de tipos.
- Traz uma implementação de *duck typing* baseada em interfaces.

Por que Go?

- Alguns tipos de coleção de dados que são nativos a linguagem:
  - *slices (lista de tamanho dinâmico).*
  - *maps (dicionários de dados associativos).*
  - *arrays (lista de tamanho fixo).*

Por que Go?

- Suporta o uso de ponteiros porém aritmética sobre ponteiros não é permitido.
- Possui coletor de lixo (*garbage collector*).
- Permite a escrita de programas totalmente procedurais, orientados a objetos ou funcionais.

Por que Go?

- A abordagem de Go para concorrência e um dos maiores diferenciais da linguagem (implementa *goroutines*)

- Dispensa o uso de travas, semáforos e outras técnicas de sincronização de processos.

Operadores

- Operadores:

+	&	+=	&=	&&	==	!=
-	|	-=	|=	||	<	<=
*	^	*=	^=	<-	>	>=
/	<<	/=	<<=	++	=	:=
%	>>	%=	>>=	--	!	...
	&^		&^=			

- Precedência

*	/	%	>>	<<	&	&^
+	-	|	^			
==	!=	<	<=	>	>=	
&&						
||						

Valores e Tipos

- Valores:
  - Boolean
  - Numérico
  - String
  - Array
  - Structs
  - Ponteiros
  - Mapa
  - Channel
- Tipos:
  - Primitivos
  - Compostos

Abstrações de funções e procedimentos são valores de segunda classe, em Go.

Tipos de dados primitivos

```
var x int = 0
var y float64 = 5.8
var str string = "Olá"
var par bool = true
```

- Boolean - *true* ou *false*
- Numéricos - valores do tipo inteiro, ponto flutuante e complexos
- Strings - arrays imutáveis

Tipos de dados compostos

- Produtos cartesianos: *structs*

```
type Aluno struct {
    nome      string
    matricula int
}
```

- Mapeamentos: *Arrays, slices, mapas ou funções*

```
pares := [3]int{2,6,8}
```

Tipos de dados compostos

- Recursivos: *structs*

```
type Node struct {
    info string
    prox *Node
}
```

A linguagem não dá suporte à união disjunta, e conjuntos potência.

Checagem de tipos

- Go é estaticamente e fortemente tipada.

```
func main(){
     var x int
     x = “nome”
}
```

```
luana@luana-Inspiron-5423:~/go/src$ go run exemplo.go
# command-line-arguments
./exemplo.go:5:4: error: incompatible types in assignment (cannot use type string as type int)
  x = "nome"
    ^
```

Equivalências de tipos

- A linguagem dá suporte à equivalência nominal, mas não à equivalência estrutural.

```
type Numero1 struct { x int; par bool }
type Numero2 struct { y int; impar bool }

func (n *Numero1) ehPar() (bool) {
    return n.par
}

func main(){
    var num *Numero2 = new(Numero2)
    num.ehPar()
}
```

Palavras reservadas

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

Expressões e comandos

- Comando sequencial:

```
n := 0
n = 3; n += 1;
```

- Comando colateral: não existem comandos colaterais.

Expressões e comandos

- Comandos condicionais:
  - *if / else*

```
func verifica (x int) {
     if x<0 {
          fmt.Println("O valor é negativo!")
     }else{
          fmt.Println("O valor é positivo!")
     }
}
```

Expressões e comandos

- *switch*

```
switch x {
     case 1:
          fmt.Println("X é 1")
          break
     case 5:
          fmt.Println("X é 5")
          break
     default:
          fmt.Println("X não é 1 nem 5")
          break
}
```

Expressões e comandos

- Comandos de repetição:
  - definido:

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

  - indefinido:

```
i := 0
for i < 10 {
    fmt.Println(i)
    i++
}
```

Variáveis

- Os valores armazenáveis são os valores dos tipos primitivos e ponteiros.

  - Variáveis temporárias: todas, exceto as do tipo *File* (que são ponteiros)
  - Variáveis persistentes: em Go, são do tipo *File*

Variáveis

- Declaração:

```
var x int = 5
x := 5
```

```
var x, y, z int = 1, 2, 3
x, y, z := 1, 2, 3
```

- A atualização pode ser seletiva ou total:

```
var p1 Ponto
p1.x = 3
p1.y = 5
var p2 Ponto
p2 = p1
```

Bloco e escopo

- Blocos aninhados.
- Variáveis declaradas em blocos de escopo internos não são visíveis em blocos de escopo externos.
- Em Go, o corpo de um bloco é avaliado no ambiente em que foi definido (associação estática).

Blocos e escopo

```
var s int = 2;

func incremento (d int) int {
    return d*s;
}

func executaIncremento (valor int) int {
    var s int = 3;
    fmt.Println(s);
    return incremento(valor);
}

func main(){
    fmt.Println(executaIncremento( 5));
}
```

Arrays, slices e maps

- Os arrays possuem tamanho fixo (estático) e não armazenam valores de tipos diferentes:

```
nomes := [2]string{"Ana", "Luana"}
```

Arrays, slices e maps

- Slice é uma abstração criada em cima de arrays. No entanto, os slices possuem tamanho variável (dinâmico), e podem crescer indefinidamente.
- Quando criados com a função *make* e usados como argumentos, ou retorno de funções, são passados por referência, e não por cópia.

```
primos := []int{2, 3, 5, 7, 11, 13}
nomes := []string{}
primos := make([]int, 6, 7)
```

Arrays, slices e maps

- Fatiando slices:

```
x := []int{0,1,2,3}
y := x[1:]
for i := range y {
    fmt.Println(y[i])
}

> 1 2 3
```

```
x := []int{0,1,2,3}
y := x[1:]
y[0] = 5

for i := range x {
    fmt.Println(x[i])
}

> 0 5 2 3
```

Arrays, slices e maps

- Map é uma coleção de pares *chave-valor*, sem nenhuma ordem definida.
- As chaves devem ser de mesmo tipo, e são únicas.
- Se armazenarmos dois valores distintos sob uma mesma chave, o primeiro valor será sobrescrito pelo segundo.

```
vazio1 := map[int]string{}
vazio2 := make(map[int]string)
```

```
capitais := map[string]string{
"GO" : "Goiânia",
"PB" : "João Pessoa",
"ES" : "Vitória"
}
```

Abstrações

- Abstração de funções:
  - Go dá suporte a abstrações de funções através da palavra reservada *func*.
  - A presença do retorno (*return*) caracteriza a abstração de função.

Abstrações

- Abstrações de procedimentos:
  - Go usa a mesma palavra reservada para caracterizar uma abstração de procedimentos, como em abstração de funções (*func*).
  - A diferença é que o que caracteriza a abstração de procedimentos é a falta do retorno (*return*).

Passagem de parâmetros

- Em Go, a passagem de parâmetros é por cópia/valor.
- Não dá suporte à passagem por referência, mas a simula com o uso de ponteiros.

Pacotes

- As funções e variáveis globais que começam com a letra maiúscula serão visíveis para quem importar esse pacote. Caso contrário, a função ficará com visibilidade privada.

```
package funcoes

import "fmt"
func Imprime (x int) {
     fmt.Println(x)
}

func dobro ( y int) (int) {
     return y*2
}
```

Objetos e classes

- Go não tem objetos e classes, porém os simula através de *structs* e funções ligadas a esse tipo.

Sistema de tipos

- Monomorfismo
- Sobrecarga
- Polimorfismo paramétrico
- Polimorfismo por inclusão
- Interfaces
- Coerção

Monomorfismo

- As abstrações definidas pelo programador são monomórficas. Toda entidade tem um tipo específico associado.

Sobrecarga

- Go dá suporte a sobrecarga de operadores, mas não dá suporte à sobrecarga de operações (*funções*).

- Para tipos da própria linguagem, um único operador ou identificador pode denotar diferentes abstrações.

Polimorfismo por inclusão

- Go não tem tipos que contenham subtipos que herdam operações aplicáveis a valores desses tipos.

```
func main(){
     type Natural int
     type Inteiro int
     var x Natural = 1
     var y Inteiro = -1
     fmt.Println(x+y)
}
```

```
./exemplo.go:12:15: error: incompatible types in binary expression
  fmt.Println(x+y)
                ^
```

Polimorfismo paramétrico

- Go não dá suporte a polimorfismo paramétrico.

```
func maior (x,y int64) (int64) {....}

func main() {
     var x int8 = 1
     var y int8 = 2
     fmt.Println(maior(x,y))
}
```

```
./exemplo.go:16:20: error: argument 1 has incompatible type (cannot use type int8 as type int64)
  fmt.Println(maior(x,y))
                    ^
./exemplo.go:16:22: error: argument 2 has incompatible type (cannot use type int8 as type int64)
  fmt.Println(maior(x,y))
                      ^
```

Coerção

- Não há mapeamento implícito de valores de um tipo para valores de outro tipo.
- Para substituir as coerções, são adotados *casts*.

```
func main() {
	x, y := 1.4, 4
	z := x+y
	fmt.Println(z)
}
```

```
./exemplo.go:8:8: error: incompatible types in binary expression
  z := x+y
       ^
```

```
func main() {
	x, y := 1.4, 4
	z := x + float64(y)
	fmt.Println(z)
}
```

```
luana@luana-Inspiron-5423:~/go/src$ go run exemplo.go
5.4
```

Interface e Exceções

- Interface:
  - Go simula o uso de interface.

```
type Shaper interface {
    Area() int
}

type Rectangle struct {
    length, width int
}

func (r Rectangle) Area() int {
    return r.length*r.width
}
```

- Exceções:
  - A linguagem não dá suporte a tratamento de exceções.

Desvios incondicionais

- Para fazer desvios incondicionais usamos a palavra reservada *"goto"*
- Só podemos fazer desvios incondicionais dentro do escopo da abstração de função.

```
L : raizQuadrada(num)
.
.
.
var num int = LeInt()
if num >= 0 {
    goto L
}
```

Escapes

- Escape:
  - Possui suporte a escape através da palavra reservada “*break*” e “*continue*”.

- Escape rotulado:
  - Possui suporte a escape rotulado através da palavra reservada *break* mais o rotulo.

Escapes

```
func main () {
     for i:=0; i<7; i++ {
          if i==5 {
               break
          }
          fmt.Println(i)
     }
}
```

```
> 0 1 2 3 4
```

```
func main () {
     externo:
     for i:=0; i<3; i++ {
          for j:=0; j<3; j++ {
               fmt.Printf("%v %v\n", i, j)
               if j==2 {
                    break externo
               }
          }
     }
}
```

```
> 0 0
> 0 1
> 0 2
```

Concorrência

- A linguagem dá suporte a concorrência através de *goroutines*.

- A comunicação entre os processos concorrentes é através de *channel.*

Avaliação da linguagem

Critérios	GO	Critérios	GO
Legibilidade	Sim	Reusabilidade	Sim
Redigibilidade	Sim	Modificabilidade	Sim
Confiabilidade	Parcial	Portabilidade	Não
Eficiência	Parcial	Implementação	Compilada
Facilidade de aprendizado	Sim	Paradigma	Concorrente

Referências

- GoLang:
  - http://golang.org
- Livro “Programando em Go” - Caio Filipini.
- Google:
  - http://google.com

Go Rocks!