Decisões de projeto da linguagem de programação Go

Pedro Probst Minini | 201710013

ELC1088 - Implementação de Linguagens de Programação

2019-08-11

---

1. Projeto de identificadores

Identificadores nomeiam entidades como variáveis e tipos. Em Go, a regra de nomeação segue a seguinte expressão regular:

```
identificador = letra { letra | dígito_unicode } .
```

Onde `{ }` indica uma repetição (0 até n vezes).

Vale notar que alguns identificadores são pré-declarados, ou seja, são reservados pela linguagem.

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

Constantes:
    true false iota

Valor zero:
    nil

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

Igualmente reservadas são as palavras-chave da linguagem, que não podem ser usadas como identificadores.

```
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
```

Operadores e pontuação:

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

2. Vinculação de variáveis

Go é comumente referida como uma linguagem estaticamente tipada com *features* que a fazem ”parecer” dinamicamente tipada.

O tipo de uma variável é tipicamente definido na sua declaração, por exemplo:

```go
var n int
// Onde 'n' é declarado como um inteiro (inicialmente de valor 0).
```

Variáveis de tipo `interface` têm também um tipo dinâmico distinto, que é o tipo real do valor armazenado na variável no tempo de execução. O tipo dinâmico pode variar durante a execução. Exemplificando, isso quer dizer que o seguinte código é válido em Go:

```
var qualquerCoisa interface{} = 42
// O tipo estático de 'qualquerCoisa' é interface{}, mas o tipo dinâmico é int.
qualquerCoisa = "Mudando o tipo dinâmico de int para string!"
```

Usos práticos do tipo `interface` podem ser vistos em:

https://golang.org/doc/effective_go.html#interfaces_and_types

3. Inferência de tipos

A inferência de tipos "existe" em Go, mas é extremamente limitada.

Ao declarar uma variável sem especificar o tipo (usando `:=` ou `var =`), o tipo da variável é inferido a partir do valor à direita.

```
var a = "AHHHHHHHHHHHHHHHHHHH"
fmt.Printf("a é do tipo %T\n", a)
// a é do tipo string
b := 42
fmt.Printf("b é do tipo %T\n", b)
// b é do tipo int
```

Ou seja, formalmente, no fundo não há inferência de tipos; pela sintaxe vista acima, é avaliado o tipo do valor definido à direita do nome da variável. É muito similar à palavra-chave `auto` de C++.

4. Bloco e escopo

Um bloco em Go é uma sequência de declarações (possivelmente vazias) contidas entre chaves.

```
Bloco = "{" ListaDeclarações "}" .
ListaDeclarações = { Declaração ";" } .
```

Além de blocos explícitos, há também blocos implícitos:

1. O *bloco universo* compreende todo o código-fonte;
2. Cada pacote (*package*) tem um *bloco de pacote* que contém todo o código-fonte para aquele pacote;
3. Cada arquivo tem um *bloco de arquivo* que contém todo o código-fonte no arquivo;
4. Considera-se que cada "if", "for" e "switch" está em seu próprio bloco implícito;
5. Cada cláusula em um "switch" ou "select" atua como um bloco implícito.

Escopos em Go são definidos a partir de blocos, e há seis "regras" que ditam o comportamento dos escopos (foram adicionados exemplos em definições difíceis de visualizar):

1. O escopo de um identificador pré-declarado é o *bloco universo*;
2. O escopo de um identificador denotando um(a) constante, tipo, variável ou função declarado fora de qualquer função é o *bloco do pacote*. Tradução:

   ```
   // f1.go
   package main
   var x int

   // f2.go
   package main

   import "fmt"

   func f() {
     fmt.Println(x) // válido!
   }
   ```

3. O escopo do nome do pacote de um pacote importado é o *bloco do arquivo* do arquivo contendo a declaração de *import*. Tradução:

```
// Isso seria inválido!

// f1.go
package main

import "fmt"

// f2.go
package main
func f() {
  fmt.Println("Olá, mundo.")
}
```

4. O escopo de um identificador denotando um receptor de método, parâmetro de função ou variável de resultado é o corpo da função;
5. O escopo de um identificador constante ou variável declarado dentro de uma função começa no final do ConstSpec ou VarSpec (ShortVarDecl para declarações de variável curtas) e termina no final do bloco mais interno. Tradução:

```
// Inválido:
func foo() {
  fmt.Println(x)
  x := 5 // deveria ter sido declarada antes
}

// Aninhamento de blocos:
func bar() {
  fmt.Println("Olá, mundo.")  // x fora do escopo
  {                           // x fora do escopo
    x := 5                    // x dentro do escopo
    fmt.Println(x)            // x dentro do escopo (válido)
  }                           // x fora do escopo
  fmt.Println(x)              // x fora do escopo (inválido)
}
```

6. O escopo de um identificador de tipo declarado dentro de uma função começa no identificador no TypeSpec e termina no final do bloco contendo mais interno. Tradução: basicamente igual à regra de escopo para variáveis.

Ainda, um identificador declarado em um bloco pode ser redeclarado num bloco mais interno. O identificador considerado passa a ser o mais próximo.

```
var x int // 0

func main() {
    var x int
    x = 5 // qual x será considerado? o mais próximo!
    fmt.Println(x) // 5
}
```

5. Tipos de dados permitidos

Em Go, há tipos pré-declarados (ver seção 1) e tipos que podem ser criados pelo próprio usuário. Em suma:

- Booleanos: `true` e `false` (verdadeiro e falso).
- Numéricos:

```
uint8 uint16 uint32 uint64
int8 int16 int32 int64
float32 float64
```

```
complex64 complex128
byte (apelido para uint8)
rune (apelido for int32)

// tipos pré-declarados com tamanhos que dependem da implementação:
uint (32 ou 64 bits)
int (mesmo tamanho de uint)
uintptr (uint grande o suficiente para armazenar bits de um valor de ponteiro)
```

- *String*: `string`. Sequência de bytes (possivelmente vazia) imutável.
- *Array*: sequência de elementos de um tipo específico. Blocos de construção para *slices*. Representadas por `[n]T`, onde `n` denota o número de elementos no *array* e `T` representa o tipo de cada elemento. `n` também faz parte do tipo; por isso, não podem ser redimensionadas.
- *Slice*: armazena referências para uma *array* subjacente. Representadas por `[]T`. Como o tamanho não faz parte do tipo, *slices* podem ser redimensionadas livremente. *Grosso modo*, é uma "array flexível".

```
names := [4]string{
      "Shinji",
      "Rei",
      "Asuka",
      "Misato",
}
```

- *Struct*: sequência de elementos nomeados (campos), onde cada um tem um nome e um tipo.

```
type Vertice struct {
  Lat, Long float64
}
// Note como neste exemplo foi criado um tipo "Vertice" que tem struct
// como tipo subjacente.
```

- Ponteiros: um ponteiro carrega o endereço de memória de um valor. Diferentemente de C, Go não tem aritmética de ponteiros.

```
i, j := 16, 32

p := &i         // p aponta para o endereço de i
fmt.Println(*p) // imprime 16, pois o operador * denota o valor
                // subjacente do ponteiro

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

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

- Funções: denota o conjunto de todas as funções com os mesmos tipos de parâmetro e resultado.

```
  func()
  func(x int) int
  func(a, _ int, z float32) bool
  func(a, b int, z float32) (bool)
  func(prefix string, values ...int)
  func(a, b int, z float64, opt ...interface{}) (success bool)
  func(int, int, float64) (float64, *[]int)
  func(n int) func(p *T)
  // onde ... é o "parâmetro variádico".
```

- *Interface*: simplificadamente, é uma coleção de assinaturas de métodos que um "objeto" pode implementar.

```
type Shape interface {
  Area() float64
  Perimeter() float64
}
```

- *Map*: é um tipo de dado assiciativo composto de um par chave/valor. Comumente chamado de *hashes* ou *dicts* em outras linguagens.

```
m = make(map[string]Vertice)
m["Bell Labs"] = Vertice{
      40.68433, -74.39967,
}
```

- *Channel*: provê um mecanismo para funções executando concorrentemente se comunicarem enviando ou recebendo valores de um certo tipo.

```
chan T          // envia e recebe valores do tipo T
chan<- float64  // só envia float64s
<-chan int      // só recebe ints
```

6. Expressões

Expressões são um assunto extenso em Go, contendo 22 itens na especificação oficial da linguagem. Além disso, as expressões regulares que denotam as formações de expressões são extensas e contém várias referências. Diante disso, nesta seção resumiremos os principais tipos de expressões com pequenas definições e exemplos.

1. Expressões aritméticas:

```
Da maior para a menor precedência:
(2)  *   multiplicação
(2)  /   divisão
(2)  %   módulo (resto)
(1)  +   adição
(1)  -   subtração

++  incremento
--  decremento

obs.: usar parênteses para forçar uma precedência alternativa.
```

```
var a int = 21
var b int = 10
var c int

c = a + b // 31
c = a - b // 11
c = a * b // 210
c = a / b // 2
c = a % b // 1
a++       // 22
a--       // 21
```

   Go também suporta operadores *bitwise* para operações bit-a-bit.

2. Expressões lógicas/booleanas:

```
Da maior para a menor precedência:
(3)  ==  igual
(3)  ≠   diferente
(3)  <   menor que
(3)  >   maior que
```

```
(3)  ≤  menor ou igual a
(3)  ≥  maior ou igual a
(2)  &&  and ('e') lógico
(1)  ||  or ('ou') lógico

!   not ('negação') lógico
```

obs.: usar parênteses para forçar uma precedência alternativa.

O tipo `bool` (`true` ou `false`) pode ser utilizado para auxiliar na manipulação de expressões lógicas/booleanas.

Expressões lógicas e booleanas são tipicamente utilizadas em conjunto com expressões de seleção e expressões de repetição.

3. Expressões de seleção: em Go, o controle de fluxo do programa é determinado fazendo-se uso de *if statements* e *switch statements*.

```go
if idade ≤ 35 {
  fmt.Println("Você está longe do caixão, provavelmente.")
} else if idade ≥ 36 && idade ≤ 55 {
  fmt.Println("Não tão próximo do caixão, mas não tão longe...")
} else if idade ≥ 56 && idade ≤ 70 {
  fmt.Println("Um tanto próximo do caixão.")
} else if idade ≥ 71 && idade ≤ 90 {
  fmt.Println("Quantos remédios está tomando?")
} else {
  fmt.Println("Fazendo hora-extra na Terra.")
}
```

```go
switch dia {
     case 1: fmt.Println("Seg")
     case 2: fmt.Println("Ter")
     case 3: fmt.Println("Qua")
     case 4: fmt.Println("Qui")
     case 5: fmt.Println("Sex")
     case 6: {
         fmt.Println("Sab")
         fmt.Println("Começou a jogatina!")
     }
     case 7: {
         fmt.Println("Dom")
         fmt.Println("Terminando a jogatina...")
     }
     default: fmt.Println("Dia inválido.")
}
```

4. Expressões de repetição: há apenas um constructo para repetição (*loop*) em Go: `for` . Apenas com ele, podemos construir quatro tipos básicos de *loops*.

   - *For* clássico:

   ```go
   for i := 0; i < 10; i++ {
     // faz algo 10 vezes
   }
   ```

   - *While*:

   ```go
   n := 0
   for n < 10 {
     // faz algo 10 vezes
   }
   ```

   - *Loop* infinito:

```go
for {
  // faz algo infinitamente (ou até dar break)
}
```

- *For-each* range:

```go
lista := []string{"olá", "mundo"}
for i, s := range lista {
  fmt.Println(i, s) // imprime índice e elemento
}
```

7. Subprogramas

Go tem como base o uso do tipo *Function* (`func` , ver seção 2) para denotar um subprograma. Funções em Go são definidas com o o uso da palavra-chave `func` seguido do identificador, dos parâmetros recebidos e dos tipos de retorno. Os parâmetros e o retorno podem ser vazios.

```go
// Soma dois inteiros 'a' e 'b'.
func soma(a, b int) int {
  return a + b
}

func soma2(a, b int) (total int) {
  total = a + b
  return // retorna total
}
```

Há também o suporte a "parâmetros variádicos (ou variáveis)". Funções variádicas recebem um número indefinido de parâmetros. Assim, a função acima poderia ter sido reescrita para receber um número indefinido de inteiros a serem somados:

```go
func somaVar(nums ...int) (total int) {
  for _, n := range nums {
    total += n
  }
  return
}
```

Funções também podem ser valores; assim, criamos funções anônimas.

```go
var add = func(a, b int) int {
  return a + b
}
```

Também é possível retornar múltiplos valores por função.

```go
func addMult(a, b int) (int, int) {
  return a + b, a * b
}
```

Funções também podem operar sobre tipos específicos. Neste caso, chamamos funções de *métodos*. Como não há classes em Go, é um meio de realizar programação orientada a objetos (especialmente em conjunto com o tipo *interface*).

```go
type rect struct {
  largura, altura float64
}

func (r rect) area() float64 {
  return r.largura * r.altura
}
```

Referências

- ”The Go Programming Language Specification”
  - https://golang.org/ref/spec
- ”Effective Go”
  - https://golang.org/doc/effective_go.html
- ”Golang Book”
  - https://www.golang-book.com/books/web/01-02
- ”Golang Bot”
  - https://golangbot.com/arrays-and-slices/
- ”Go by Example”
  - https://gobyexample.com/
- ”A Tour of Go”
  - https://tour.golang.org/