Chapter 1

Uma Introdução ao Go: A Linguagem Performática do Google

Diego Fernando de Sousa Lima, Leonardo Augusto Gomes da Silva

Abstract

This meta-paper describes the style to be used in articles and short papers for SBC conferences. For papers in English, you should add just an abstract and for the papers in Portuguese, we also ask for an abstract in Portuguese ("resumo"). In both cases, abstracts should not have more than 10 lines and must be in the first page of the paper.

Resumo

Este meta-artigo descreve o estilo a ser usado na confecção de artigos e resumos de artigos para publicação nos anais das conferências organizadas pela SBC. solicitada a escrita de resumo e abstract apenas para os artigos escritos em português. Artigos em inglês, deverão possuir apenas abstract. Nos dois casos, o autor deve tomar cuidado para que o resumo (e o abstract) não ultrapassem 10 linhas cada, sendo que ambos devem estar na primeira página do artigo.

1.1. Introdução

Ao longo do tempo, o desenvolvimento de *software* foi se aprimorando e as ferramentas utilizadas para sua composição evoluindo de acordo com as necessidades que a tecnologia tende a resolver. Podemos então classificar as Linguagens de Programação como principal artefato deste meio por ser o intermédio onde computador processa informações afim de gerar saídas para o uso benéfico.

O ato de criar um novo *software* pode requerer paciência, atenção e proatividade por parte do desenvolvedor e devido a isso as Linguagens de Programação estão sempre se renovando buscando maior proximidade com o programador e se tornando mais portáveis. O maior exemplo disso são as linguagens rotuladas como de alto nível nas quais podemos citar Python, PHP, Ruby, C++, Java, entre outras.

As linguagens de alto nível tem como característica principal o poder elevado de abstração aproximando o sintaxe à linguagem humana, diferentemente das linguagens de baixo nível, que por sua vez se assemelham ao código utilizado por máquina. *Golang*, como também é abreviada, fica entre as linguagens de alto nível, assim como colabora para facilitar o desenvolvimento sem perder o alto desempenho. A quantificação de seu desempenho é notado pelo poder de execução de suas *threads*, veremos isso ao decorrer deste capítulo.

As próximas seções deste capítulo detalham a linguagem desde seus conceitos principais, passando pela instalação e preparação do ambiente, sintaxe básica ponderando os elementos exclusivos da linguagem, e ao fim estão apresentados os módulos de programação para Web com Go com foco principal na construção de uma Rest API com a biblioteca nativa `http/net`.

1.1.1. Um pouco sobre Go

O problema do longo tempo de compilação recorrente na maioria das linguagens usadas foi um dos principais motivos para a criação de *Golang*. Os grandes servidores da Google necessitavam de um grande poder de eficiencia e produtividade para tornar aplicações escaláveis e mais rápidas. Em novembro de 2009 deu-se inicio ao nascimento da linguagem performatica *Golang* dentro das dependencias dos escritórios da Google. Seu projeto de criação foi liderado por Rob Pike, Ken Thompson e Robert Griessemer.

Suas principais vantagens é seu poder de processamento que permite aplicações trabalharem aproveitando o máximo do poder dos processadores multi-core de forma mais otimizada. Além de ser baseda em C com caracteristicas de tipagem forte e estática, alto nivel, *structs* e *garbage collection*. Tal poder de processamento é oriundo das *goroutines*, que são rotinas que se comunicam por *channels*, evitando o uso de memória compartilhada evitando tecnicas de sincronizações mais pesadas como "semáforos".

O Go, embora seja recente, dá suporte a várias bibliotecas para a criação de ferramentas de comunicação em rede, servidores HTTP, expressões regulares, leitura e escrita de arquivos. Por ser robusta exige um certo nível de atenção na sua codificação, característica que permite os codificadores a produzirem códigos mais limpos e padronizados. No mercado atual Go já possui seu espaço, principalmente como uma possível sucessora da linguagem C por suportar a demanda de trabalho em servidores e sistemas multi-thread.

1.1.2. Go atualmente

Devido seu diferencial, Golang está em plena ascenção. O reflexo de seu crescimento é representado pela sua utilização nos dias atuais, tanto por empresas internacionais como nacionais. Além da própria Google, outras companhias usam Go em suas infraestruturas, estando entre elas: Adobe, BBC, Canonical, Dell, DigitalOcean, Dropbox, Facebook, IBM, Mozilla, SoundCloud, Twitter, Yahoo, entre outras.

No Brasil não é diferente, muitas companhias estão aderindo ao Go. Entre as empresas e instituições com maior respaldo estão: Globo.com, Magazine Luiza, Mercado Livre, Dafiti, PagSeguro, Pagar.me, Jusbrasil, Hotel Urbano, Walmart, Nuveo, Nic.br, entre outras. A lista completa de companhias que usam Go pelo mundo está no repositório

oficial da linguagem no GitHub[1].

1.2. Instalando e configurando o Go

Para começar a programar na linguagem Go, além de ser necessário que seja feito o download e instalação, é fundamental preparar do ambiente consistindo em modificar as variáveis de ambiente, adicionando três novas variáveis ao sistema, e criar uma pasta para incluir os projetos desenvolvidos em Go chamada de `$GOPATH`. As subseções abaixo apresentam o passo a passo para instalação em sistemas Linux e Windows.

1.2.1. Linux

Para efetuar a instalação no ambiente Linux, basta ir até o site oficial da linguagem na seção de *downloads*[2] e escolha a opção **Linux**. Após fazer o *download*, navegue pelo terminal até a pasta onde está o arquivo compactado e descompacte-o com o comando `sudo tar -zxvf "nome_do_arquivo".tar.gz`. Com a extração concluída, mova o arquivo para o diretório `/usr/local` com o comando `sudo mv go /usr/local`.

Agora edite o arquivo `/etc/profile` adicionando as variáveis de ambiente para o Go:

```
export GOPATH=$HOME/GO
export GOROOT=/usr/local/go
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

É necessário criar uma pasta dentro do diretório `$HOME`[3] com o nome "GO", de acordo com a variável de ambiente descrita em `GOPATH`, e execute o comando `source /etc/profile` para finalizar a instalação. O próximo passo é estender a pasta `$GOPATH` criando as pastas **pkg**, **bin** e **src**. Preferencialmente crie seus programas dentro do diretório `$GOPATH/src`.

Por último, verifique a instalação executando o comando `go version`. O retorno deve ser algo como:

```
go version go1.8.3 linux/amd64
```

1.2.2. Windows

Para o Windows, a instalação do Go pode ser feita através do mesmo *link*[2]. Selecione a opção Windows e faça o *download*. No ato da instalação é importante que os arquivos sejam inseridos no diretório `C:\Go`. Feita a instalação, é necessário configurar as variáveis de ambiente. No Windows é possível encontrá-la em dois lugares:

- "Painel de Controle" → "Sistema" → "Avançado" → "Variáveis de ambiente";
- "Configurações avançadas do sistema" → "Variáveis de ambiente".

Na maioria dos casos, a variável `GOROOT` já está automaticamente adicionada. Caso contrário você deve adicioná-la apontando-a para o diretório `C:\Go`. Crie uma

[1] `https://github.com/golang/go/wiki/GoUsers`
[2] `https://golang.org/dl/`
[3] Para descobrir onde fica o diretório `$HOME` execute o comando `echo $HOME` no terminal.

pasta contendo as pastas **bin**, **pkg** e **src** para ser seu diretório de trabalho e adicione mais duas variáveis: `PATH` para o caminho `C:\Go\bin` e a varíavel `GOPATH` direcionada para seu diretório de trabalho.

Desenvolva seus programas dentro da pasta `GOPATH/src`. Por último visualize sua instalação usando `go version` no Prompt de Comando ou CMD. A saída deve ser essa:

```
go version go1.8.3 windows/amd64
```

1.3. Estrutura básica e sintaxe

Após a instalação e configuração, já podemos criar nossos primeiros programas. Nesta seção serão apresentados elementos que constituem a sintaxe básica de Go. Programas em Go podem ser desenvolvidos em qualquer editor de texto que dê suporte a codificação `UTF-8`. Todos os exemplos mostrados nas próximas seções tem foco na execução em ambiente Linux.

1.3.1. Hello World

Para o primeiro contato com a linguagem, podemos iniciar com um tradicional *Hello, World*. Portanto, salve um arquivo com título de sua preferência (sugerimos `hello.go`) dentro da pasta `src` do diretório `$GOPATH` com o seguinte conteúdo:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello World!")
7 }
```

Para compilar o código acima, basta executar `go run hello.go`. A saída para este programa será:

```
Hello World
```

Neste primeiro exemplo pode-se notar algumas caraterísticas, a primeira delas é que Go não possui ponto-e-vírgula ou quaisquer ponto ou acentuações ao final das instruções, diferentemente de linguagens como C ou Java. Todo código em Go é dividido em três seções principais:

1. **Declaração do pacote**: Todo arquivo Go tanto deve existir dentro de um pacote, como deve ter um pacote chamado `main` com a função main (`func main()`). A declaração do pacote deve estar no início do código;
2. **Declaração de dependências**: A segunda seção é destinada as dependências externas de um programa em Go, podendo elas serem opcionais. No nosso exemplo, usamos o `fmt` para auxiliar na entrada e saída de dados pelo terminal;
3. **Código**: Por último, temos o código de fato. Nele é onde aplicamos a lógica e é a parte central de um programa em Go.

A função `main` de um programa em Go não recebe parâmetros e nem retorna valores, mais uma característica que a difere de linguagens como o Java ou C. Para comando de entrada e saída podemos usar as funções provenientes do pacote `fmt`. O `fmt.Println()` imprime o conteúdo seguido de quebra de linha, enquanto que `fmt.Print()` imprime apenas o conteúdo. Abordamos isso melhor na seção 1.3.2.1.

1.3.2. Variáveis

Go tem tipagem forte e estática, isso quer dizer que o tipo das variáveis declaradas durante a execução de um programa não podem ser alterados. Apesar deste conjunto de características, Go traz em sua sintaxe uma forma de declaração limpa, o que facilita e torna eficiente a rapidez para o desenvolvimento das aplicações. A declaração de variáveis em Go pode ser de duas maneiras principais:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      var variavel_a int // tipo 1
7      variavel_b := 5 // tipo 2
8
9      var variavel_c int = 45 // tipo 3
10     var x, y int = 1, 2 //tipo 4
11 }
```

O *tipo 1* é usado quando não se sabe qual valor será alocado para a variável, enquanto o *tipo 2* é uma maneira mais sucinta exclusiva do Go que força o tipo da variável de acordo com o valor recebido (note que o tipo não foi declarado). É importante considerar que o compilador só obtem sucesso em sua execução quando todas as variáveis instanciadas são devidamente usadas. Outros tipos de declaração são os tipos 3 e 4. Entre os tipos de variáveis suportados por Go estão ***bool***, ***int*** *(8, 16, 32, 64)*, ***float*** *(32, 64)*, ***string***, ***byte***, ***rune*** e ***complex*** *(64, 128)*.

1.3.2.1. Entrada e saída com fmt

Com o pacote `fmt` é possível interagir com o usuário em modo texto no estilo *input/output*. As funções principais de `fmt` tem função por finalidade a entrada e saída de valores. O código abaixo mostram as funções de saídas mais comuns:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      str := "Gopher"
7
8      fmt.Print("Ola ", str, "\n")
```

```
9       fmt.Println("Ola", str)
10      fmt.Printf("Ola %s\n", str)
11  }
```

As variações da função de impressão acima exibem a mesma saída:

```
Ola Gopher
Ola Gopher
Ola Gopher
```

Para receber uma entrada o pacote `fmt` nos disponibiliza a funcão `Scan`. Para fazer a leitura para uma variável, basta colocar a variável precedida do caractere & entre os parênteses de `Scan`. Como exemplo:

```
1 var variavel string
2 fmt.Scan(&variavel)
```

1.3.3. Estrutura de seleção if/else

A estrutura de seleção `if` tem a função básica de tomar decisões perante a execução de um programa. Em Go este tipo de estrutura difere em relação a outras linguagens, ela só recebe valores lógicos verdadeiro ou falso. Isso influencia definitivamete a tipagem da variavel que o comando vai receber, ou seja, as expressões necessitam ser do tipo `bool` e podem ser criadas funções ou métodos com a regra que todos retornem pelo menos um valor do tipo `bool`.

```
1 idade := 22
2
3 if idade <= 18 {
4     fmt.Println("Menor ou igual a 18 anos")
5 } else if idade > 25 {
6     fmt.Println("Maior que 20 anos")
7 } else {
8     fmt.Println("Entre 19 e 25 anos")
9 }
```

1.3.4. Switch case

A estrutura `switch case` tem funcionalidade parecida com o `if/else` visto na seção anterior. Dependendo do caso, o `switch case` pode se adequar melhor ao programa, especialmente em casos de Menus e estruturas de decisões com muitos opções. O `switch case` é orientado a casos tendo o caso *default* escolhido quando nenhum dos outros forem satisfeitos. A sintaxe do `switch case` segue o seguinte modelo:

```
1 t := time.Now()
2 switch {
3 case t.Hour() < 12:
4     fmt.Println("Bom dia!")
5 case t.Hour() < 17:
```

```
6       fmt.Println("Boa tarde.")
7   default:
8       fmt.Println("Boa noite.")
9   }
```

1.4. Coleções de dados

Coleções de dados são tipos de variáveis especiais que podem serem entendidas como um conjunto ou listas de outras variaveis ou valores. Assim como em outras linguagens (ex: linguagem C com seus vetores) as coleções de dados são geralmente uma sequencia de valores encadeados em ordem predefinida ou até mesmo apenas espaços separados não alocados para que depois venham ser preenchidos.

Em Go os padrões para listas de dados são dois: `arrays` e `slices`. A grande diferença entre eles está relacionada ao espaço de alocação de mémoria. Os `arrays` são instacias com tamanho fixo, já os `slices` são uma camada abstraída dos `arrays` que podem ser alocados dinamicamente permitindo crescer de forma indefinida sendo mais flexíveis.

1.4.1. Arrays

Os `Arrays` são listas com valores do mesmo tipo, sendo que cada valor possui um índices que indica a posição dentro da lista. A contagem dos índices são delimitados pelo tamanho do *array* que deve ser fixo e invariável. O primeiro elemento do *array* possui índice 0, e o último elemento é sempre `len(array) - 1`. Podemos declarar um *array* em Go utilizando as seguintes formas:

```
1  var lista [3]int
2  pares := [3]int{2,4,6}
3  impares := [...]int{3, 5, 7}
4  nomes := [2]string{}
5
6  fmt.Println(colecao, pares, impares, nomes)
```

A saída para o código acima é: `[0 0 0] [2 4 6] [3 5 7] [ ]`.

O tamanho de um *array* sempres deve ser especificado na sua declaração, como por exemplo o *array* `lista` que foi fixado em `[3]int`, ou seja , seu tamanho é de 3 posições inteiras. Podemos notar que ele a saida foi `[0 0 0]`, isso ocorre por que em Go quando se é declarado um array sua posições se não forem alocada assumem automaticamente valor zero. Para vetores de outra tipagem esse valor atribuído automaticamente pode ser gerado com outras representações:

- *bool*: `false`, valor booleano falso;
- *int*: `0`;
- *float*: `0.0`;
- *strings*: `""`;

- *ponteiros*, funções, *interfaces*, *slices*, *maps* e *channels*: `nil`.

Existe também a possibilidade de criar `arrays` multidimensionais em que os valores são `arrays` dentro de `arrays`. Declaramos eles da seguinte forma:

```
1 var matrizA [2][2]int
2
3 matrizA[0][0], matrizA[0][1] = 7, 2
4 matrizA[1][0], matrizA[1][1] = 100, 90
5
6 matrizB := [2][2]int{{3, 2}, {-80, -1}}
7
8 fmt.Println("Matriz A:", matrizA)
9 fmt.Println("Matriz B:", matrizB)
```

A saída para o código acima é:

```
Matriz A: [[7 2] [100 90]]
Matriz B: [[3 2] [-80 -1]]
```

Como já foi mencionado, os `arrays` não são tão flexíveis como os `slices` (explicados na próxima subseção). Porém eles em sua importância dentro da linguagem Go. Cabe ressaltar que é possivel trabalhar com `arrays` de forma que eles sejam dinâmicos, porém requer um esforço manual custoso demais tendo que verificar seus limites e criação de `arrays` para cópias de valores, o que possivelmente deixará o código mais poluído.

1.4.2. Slices

Os `slices` são uma abstração que se baseiam em `arrays` para possibilitar mais flexibilidade na coleção de dados encadeados. A principal característica de um `slice` é a ausência de limites permitindo maior dinamicidade para a coleção de dados. Para declarar uma `slice` podemos utilizar quase a mesma sintaxe de um *array*. A diferença é que não especificamos seu tamanho:

```
1 var sliceA []int
2 sliceB := []int{10, 20, 30}
3 SliceC := []string{}
4
5 fmt.Println(sliceA, sliceB, sliceC)
```

A saída para o código acima é: `[] [2 4 6] []`

É possível também criar um `slice` com a função `make`, que separa internamente um espaço de memória para um *array* retornando uma referência para o `slice`. Sua sintaxe pode ser escrita da seguinte forma:

```
1 lista := make([]int, 10)
2 lista2 := make([]int, 10, 30)
3
4 fmt.Println(lista, len(lista), cap(lista))
5 fmt.Println(lista2, len(lista2),cap(lista2))
```

A saída deste código é:

```
[0 0 0 0 0 0 0 0 0 0] 10 10
[0 0 0 0 0 0 0 0 0 0] 10 30
```

O primeiro parâmetro da função `make` indica o tipo de elementos que irão ser criados, o segundo representa o tamanho inicial e o terceiro define a capacidade total de memória reservada. No `slice` definido em `lista` é possível analisar que o tamanho inicial e capacidade total são os mesmos pelo fato da capacidade não ter sido definida, enquanto que em `lista2`, a capacidade é definida para um total de 30 posições.

1.4.3. Maps

Os *maps* são uma estrutura de conjunto de dados organizado em **chave-valor**. Em outras linguagens como Ruby e Python, os *maps* se assemelham aos *Dict's* e *Hashes* respectivamente. Podemos declarar um `map` em Go usamos a forma de declaração simples ou usando a função `make()`, assim como nos exemplos usados para os `slices`. No exemplo a seguir temos declarados *maps* com chaves do tipo `int` e valores `strings`.

```
1 map1 := map[int]string{}
2 map2 := make(map[int]string)
```

Automaticamente os *maps* tem tamanho indefinido. Porém é possível alocar a quantidade de espaços necessários possíveis durante a execução do programa tornando mais eficiente e fácil de se trabalhar com dados diferentes sem a necessidade de saber tamanhos predefinidos. Recomenda-se sempre definir uma especificação de memória passando um segundo argumento à função `make()`:

```
1 map3 := make(map[int]string, 2066)
```

As formas literais de inserir dados em um `map` seguem o seguinte modelo:

```
1 timesDoPiaui := map[string]string{
2     "RIV": "River",
3     "PAR": "Parnahyba Sport Club",
4     "PEC": "Sociedade Esportiva de Picos",
5 }
6 timesDoPiaui["PEC"] = "Piaui Esporte Clube"
7
8 fmt.Println(timesDoPiaui)
```

O resultado do código acima é:

```
map[SEP:Sociedade Esportiva de Picos PEC:Piaui Esporte Clube
RIV:River PAR:Parnahyba Sport Club]
```

Podemos notar que na primeira indicação da chave `"PEC"` o valor especificado foi `Sociedade Esportiva de Picos`, e logo após foi escrito o seguinte código: `timesDoPiaui["PEC"] = "Piaui Esporte Clube"`, ele permite que os valores sejam atualizados diretamente no em uma chave especifica de um `map`. Logo após a saída o valor que antes era `Sociedade Esportiva Picos` foi trocado por `"Piaui Esporte Clube"`.

1.4.4. Loops

A estrutura básica de repetição em Go é o comando for. Ele é comumente usado para fazer iterações com listas, `slice`, `maps`. Em sua forma mais comum é especificado uma condição lógica e o bloco código se repetirá até que a condição seja satisfeita com verdadeiro:

```
1 valor1, valor2 := 0, 20
2
3 for valor1 < valor2 {
4     valor1 += 1
5 }
```

O código que está acima será repetido em 20 iterações, ou seja, a variável `valor1` que recebe 0 na primeira linha sera incrementado com mais um até que ele satisfaça a condição do `for` do `valor1` ser menor que o `valor2` satisfazendo a condição de verdadeiro para a parada do comando. Há a possibilidade de fazer iterações de forma mais tradicional:

```
1 for i := 0; i < 10; i++ {
2     ...
3 }
```

O código acima também executa qualquer bloco de código até que a condição de verdadeiro seja satisfeita, então o comando fará 10 iterações. Outra forma de fazer um iteração com for é feita sobre `slices` com auxilio do comando `range`:

```
1 for indice, valor := range slice {
2     ...
3 }
```

No código acima é retornado o índice de cada elemento do `slice`, para obter acesso e modificar valores desse `slice` podemos utilizar os índices. Assim basta somente omitir o segundo valor na atribuição e acessar cada elemento através de seu índice:

```
1 numeros := []int{1, 2, 3, 4, 5}
2
3 for i := range numeros {
4     numeros[i] *= 2
5 }
6
7 fmt.Println(numeros)
```

A saída do código acima é: `[2 4 6 8 10]`.

Como ultimo exemplo, podemos usar o laço `for` como um *loop* infinito como o *while* em outras linguagens:

```
1 for {
2     ...
3 }
```

Para a parada do *loop* basta instanciar algum comando de seleção como o `if` com `break` interno.

1.5. Funções

Um dos pontos fortes da linguagem Go é a forma de variedades nas quais podemos escrever funções. Go permite que suas funções possam receber parâmetros, assim como também retornar valores podendo ser múltiplos. Até este momento neste capítulo usamos funções, pois a rotina de um código em Go deve iniciar da função `main()`. Nas subseções seguintes detalhamos alguns dos tipos mais usados.

1.5.1. Funções básicas

Em Go o padrão básico de declaração das funções se dá com a palavra `func` seguido do nome da função e dos possíveis valores de parâmetros. Um exemplo seria uma função que imprima uma simples frase:

```
1 func imprimirString() {
2     fmt.Println("Imprimindo uma frase")
3 }
```

Agora iremos passar alguns argumentos para esta função, fazendo que ela imprima um valor do tipo `string` e outro do tipo `int`:

```
1 func imprimirString(nome string, idade int) {
2     fmt.Printf("Ola, meu nome eh %s e eu tenho %d anos.\n",
3      nome, idade)
4 }
```

Caso os argumentos passados para a função sejam do mesmo tipo, é possível agrupá-los em uma única especificação:

```
1 func intervalo(x, y int) {
2     for i := x; i < y; i++ {
3         fmt.Printf("%d ", i)
4     }
5 }
```

Para retornar valores em uma função é necessário definir seu tipo logo após a passagem de parâmetros. Além disso, também é necessário fazer o uso do `return` na forma básica:

```
1 func simplesSoma(x, y int) int {
2     return x + y
3 }
```

O retorno de funções em Go também podem retornar vários valores. Para que isso seja possível devemos adicionar os tipos retornados na sequencia correta entre parênteses. O `return` também deve de acordo:

```
1 func sucessorEAntecessor(x int) (int, int) {
```

```
    return x + 1, x - 1
}
```

1.5.2. Retorno definido

Em Go é possível definir a variável de retorno de uma função logo na declaração. Para que isso seja possível devemos usar a mesma variável de retorno definida para receber um valor dentro da função e ainda usar o `return` sem especificar valores:

```
func simplesSoma(x, y int) (soma int) {
    soma = x + y
    return
}
```

1.5.3. Funções de argumentos variáveis

As funções de argumentos variáveis recebem como argumento um determinado tipo e uma variável especificada. A ideia é permitir que sejam passados *n* argumentos para este tipo de função para que trate esses valores como uma lista. Podemos assim iterá-los como um *array*:

```
func numeros(lista ...int) {
    for _, numero := range lista {
        fmt.Println(numero)
    }
}
```

1.5.4. Funções anônimas

Funções anônimas são funções que são criadas no momento em que são utilizadas. Elas são alocadas para uma variável e são usadas com frequência quando se quer resolver pequenos problemas. Por exemplo, podemos declarar uma função anônima para deixar todas as letras de uma `string` em maiúsculo. Para o próximo código importamos o pacote `strings`. Veja o exemplo:

```
func main() {
    maiusculo := func(str string) string {
        return strings.ToUpper(str)
    }

    nome := "Diego Fernando"

    fmt.Println(nome)
    fmt.Println(maiusculo(nome))
}
```

A saída deste código é:

```
Diego Fernando
DIEGO FERNANDO
```

1.5.5. Defer

O `defer` define uma função que sempre será executada ao fim de uma rotina atual. O comando é ideal para programas em que se usa I/O onde deve haver abertura e fechamento de arquivos ou conexões, pois garante a execução de determinada função ao final. O exemplo a seguir apresenta a função `defer` que imprime a string `Segunda acao` logo após a impressão de `Primeira acao`:

```
1 func main() {
2     defer func() {
3         fmt.Println("Segunda acao")
4     }()
5
6     fmt.Println("Primeira acao")
7 }
```

1.6. Tipos de dados

Além dos tipos de dados tradicionais, Go permite a criação de tipos personalizados de dados. Esse tipo de característica se torna importante, uma vez que recursos relativos a Orientação a Objetos praticamente não existem em *Golang*. Apesar desse ponto, Go compensa o uso da OO com recursos como funções que se estendem de tipos e *interfaces*, que estão apresentados nesta seção.

1.6.1. Criando novos tipos

Para criar um novo tipo de dado basta acrescentar `type` antes de qualquer tipo primitivo. Com a finalidade de ilustrar a criação de um novo tipo de dado, no código abaixo é declarado o tipo de dado `TimesDoPiaui` baseado em um *slice* do tipo primitivo *string*. Note que, tanto a instância, como a recepção dos valores são feitos de fato dentro da função `main()`.

```
1  package main
2
3  import "fmt"
4
5  type TimesDoPiaui []string
6
7  func main() {
8    times := make(TimesDoPiaui, 4)
9    times[0] = "River"
10   times[1] = "Parnahyba Sport Club"
11   times[2] = "Sociedade Esportiva de Picos"
12   times[3] = "Piaui Esporte Clube"
13
14   for i := 0; i < len(times); i++ {
15     fmt.Println(times[i])
```

```
16    }
17 }
```

Á primeira vista, esta estrutura pode não fazer muito sentido. Porém a grande vantagem em usar tipos customizados é a possibilidade de estendê-lo. Usaremos como exemplo uma função que mostre se o time "Parnahyba Sport Club" está no *slice*. Observe que não passamos nenhum parâmetro, porém o Go entende que esta função é do tipo `TimesDoPiaui` e a trata como um "método" para este tipo.

```
1 func (time TimesDoPiaui) TemPhb() {
2   for _, value := range time {
3     if value == "Parnahyba Sport Club" {
4         fmt.Println("Tem Parnahyba Sport Club!")
5     }
6   }
7 }
```

1.6.2. Structs

As *structs* ou tipos estruturados de dados seguem um princípio de criar tipos a partir de conjuntos de outros tipos. É uma abstração muito utilizada em linguagens como C e C++. *Structs* facilitam o agrupamento de dados criando a noção de registros. Com fim de demonstrar um exemplo de *struct*, usaremos o mesmo tema da seção anterior onde será um tipo estruturado de nome `TimeDoPiaui` onde teremos o atributos `nome`, `n_vitorias`, `n_derrotas` e `classificado`.

```
1  package main
2
3  import "fmt"
4
5  type TimeDoPiaui struct {
6      nome         string
7      n_vitorias   int
8      n_derrotas   int
9      classificado bool
10 }
11
12 func main() {
13     river := TimeDoPiaui{
14         nome:         "River",
15         n_vitorias:   23,
16         n_derrotas:   6,
17         classificado: true,
18     }
19
20     fmt.Println(river)
21 }
```

A saída do código acima é: `River 23 6 true`. Podemos fazer uma função que estende do tipo `TimeDoPiaui` que imprima dados relativos ao time. Isso trará mais possibilidades de reúso de código. Considerando a *struct* criada, instanciamos mais outro time e criamos a função `MostraSituação()`:

```
1  func main() {
2      river := TimeDoPiaui{
3          nome:         "River",
4          n_vitorias:   23,
5          n_derrotas:   7,
6          classificado: true,
7      }
8
9      picos := TimeDoPiaui{
10         nome:         "SEP",
11         n_vitorias:   18,
12         n_derrotas:   12,
13         classificado: false,
14     }
15
16     river.MostraSituacao()
17     picos.MostraSituacao()
18 }
19
20 func (t TimeDoPiaui) MostraSituacao() {
21     situacao := "nao esta"
22     if t.classificado {
23         situacao = "esta"
24     }
25
26     fmt.Printf("%s tem %d vitorias,"+
27                " %d derrotas e %s "+
28                "classificado.\n", t.nome,
29                                   t.n_vitorias,
30                                   t.n_derrotas,
31                                   situacao)
32 }
```

O resultado desta estrutura é:

```
River tem 23 vitórias, 7 derrotas e está classificado.
SEP tem 18 vitórias, 12 derrotas e não está classificado.
```

1.6.2.1. Manipulação de tipos de dados

Vimos no tópico anterior que podemos criar estruturas e interagir com elas por meio de funções. Porem, a função `MostraSituação` apenas faz uma exibição de valores.

Nesta seção evoluiremos a estrutura da seção anterior criando uma *slice* para o tipo `TimeDoPiaui` com as funções `AdicionarTime()`, `RetirarTime()` e `MostrarTimes()`.

É válido ressaltar que, assim como nas linguagens C e C++, Go usa ponteiros. Em tipos estruturados, os ponteiros são necessários caso quisermos modificar estruturas. Portanto para o exemplo projetado usaremos ponteiros nas funções de `AdicionarTime()` e `RetirarTime()`, deste modo poderemos modificar a lista de objetos.

Na mesma seção do código em que se encontra o `type TimeDoPiaui struct`, incluiremos outro tipo que será um *slice*. Algo como:

```
type TimeDoPiaui struct {
    nome          string
    n_vitorias    int
    n_derrotas    int
    classificado bool
}

type Times []TimeDoPiaui
```

Como a manipulação é feita no tipo `Times`, então as funções são estendidas neste tipo. Na funções de adição e remoção usamos o comando `append` para fazer recriações modificadas do *slice* original. As funções ficam organizada deste modo:

```
func (t *Times) AdiconarTime(nome string,
                             n_vitorias int,
                             n_derrotas int,
                             classificado bool) {
    novoTime := TimeDoPiaui{
        nome,
        n_vitorias,
        n_derrotas,
        classificado,
    }

    *t = append(*t, novoTime)
    fmt.Printf("\n%s adicionado!", nome)
}

func (t *Times) RetirarTime(indice int) {
    times := *t
    time_retirado := times[indice]
    *t = append(times[0:indice], times[indice+1:]...)
    fmt.Printf("\n%s removido!", time_retirado.nome)
}

func (t Times) MostrarTimes() {
    fmt.Println("\n\n**Times:**")
```

```
25      for i := 0; i < len(t); i++ {
26          fmt.Printf("%s, %d, %d, %t\n", t[i].nome,
27                                         t[i].n_vitorias,
28                                         t[i].n_derrotas,
29                                         t[i].classificado)
30      }
31  }
```

Com essa estrutura, podemos deixar a função `main()` mais limpa, assim deixando apenas chamadas de funções nela, veja um exemplo:

```
1  func main() {
2      times := Times{}
3      times.AdiconarTime("River", 23, 7, true)
4      times.AdiconarTime("Parnahyba Sport Club", 25, 5, true)
5      times.AdiconarTime("SEP", 20, 10, false)
6      times.MostrarTimes()
7      times.RetirarTime(0)
8      times.MostrarTimes()
9  }
```

A saída do código criado seria algo como:

```
River adicionado!
Parnahyba Sport Club adicionado!
SEP adicionado!

**Times:**
River, 23, 7, true
Parnahyba Sport Club, 25, 5, true
SEP, 20, 10, false

River removido!

**Times:**
Parnahyba Sport Club, 25, 5, true
SEP, 20, 10, false
```

1.6.2.2. Interfaces

A melhor definição para um *interface* é que elas são um "contrato" para outros tipos de dados. Com *interfaces* é possível criar conjuntos de métodos que servem para *n* tipos estruturados, pois ele consegue orientar uma mesma ação para tipos diferentes de dados, mesmo que permitindo que o tipo possa ser direcionado a um método apropriado.

Vamos a um exemplo hipotético imaginando duas *structs*: `TimeFutebol` e `TimeBasquete`. Apesar de serem tipos diferentes, as duas tem métodos estendidos

com a mesma finalidade que serve para fazer o calculo de pontos do time em determinado campeonato baseado na quantidade de vitórias. Definimos regras de negócios diferentes para basquete e futebol, onde, para o basquete cada vitória vale 2 pontos e para o futebol cada vitória vale 3:

```
type TimeFutebol struct {
    n_vitorias int
}

type TimeBasquete struct {
    n_vitorias int
}

func (t TimeFutebol) PontosEmVitorias() int {
    return t.n_vitorias * 3
}

func (b TimeBasquete) PontosEmVitorias() int {
    return b.n_vitorias * 2
}
```

Após declarados os tipos estruturados, iremos escrever mais um tipo *interface* e um método estendido do tipo *interface* recém criado:

```
type Time interface {
    PontosEmVitorias() int
}

func Pontos(t Time) int {
    return t.PontosEmVitorias()
}
```

A função `Pontos()` recebe como parâmetro uma *interface* podendo ser um tipo `TimeFutebol` ou `TimeBasquete`. Já o retorno da função `Pontos()` é a chamada do método `PontosEmVitorias()` na qual foram feitas uma implementação para cada tipo. Dependendo do tipo recebido a função `Pontos()` irá direcionar a chamada para o método `PontosEmVitorias()` adequado.

A mágica acontece quando usamos o mesmo método para os dois tipos. Vejamos como fica a função `main()`:

```
func main() {
    time_futebol := TimeFutebol{20}
    time_basquete := TimeBasquete{20}

    fmt.Println(Pontos(time_futebol))
    fmt.Println(Pontos(time_basquete))
}
```

Nas instâncias acima, `time_futebol` e `time_basquete` tem a mesma quantidade de vitórias (20), porém se executarmos essa função teremos a saída `60 40`, ou seja 60 pontos para o time de futebol e 40 pontos para o time de basquete.

1.7. Tratamento de erros

1.8. Concorrência

1.8.1. Goroutines

As goroutines são similares as threads comuns, porém elas são gerenciadas pelo ambiente de execução, ou seja, a linguagem decide quando elas iram ser interligadas as threads principais do sistema. Por esse fator elas se tornam extremamente leves por ter um controle sobre qual execução realmetente é necessaria estar sendo executada pelo SO.

Para dar inico a uma goroutines se utiliza uma palavra-cahve `go` depois se chama uma função; isso permite que a execução da thread principal rode em parelelo a função sem a necessidade de bloquear principal uma delas. A seguir está uma função `sequencia()` recebendo um número inteiro imprimindo uma sequencia que vai do 0 até o valor passado como argumento:

```
1 func sequencia(n int) {
2     for i := 0; i < n; i++ {
3         fmt.Printf("%d ", i+1)
4         time.Sleep(200 * time.Millisecond)
5     }
6     fmt.Printf(" ...Sequencia de %d terminou... ", n)
7 }
```

A função `main()` a seguir chama a função `imprimir()` instanciada duas vezes com números diferentes:

```
1 func main() {
2     sequencia(5)
3     sequencia(10)
4 }
```

Ao executar a função `main()` a saida do código é:

```
1 2 3 4 5 ...Sequencia de 5 terminou...  1 2 3 4 5 6 7 8 9 10
...Sequencia de 10 terminou...
```

Agora façamos diferente, vamos usar a palavra-chave `go` para um das chamadas da funções:

```
1 func main() {
2     go sequencia(5)
3     sequencia(10)
4 }
```

Logo a saida da função `main()` será:

```
1 1 2 2 3 3 4 4 5 5 6 ...Sequencia de 5 terminou...  7 8 9 10
...Sequencia de 10 terminou....
```

Notamos que já ocorreu uma diferença entre as saídas. A goroutine aberta para a função imprimir foi executada em paralelismo com a função principal. Isso torna o programa mais eficiente em recurços, diferentemente em outras linguagens como Ruby e Python que se utilizam de *Green Threads* que precisão fazer rotinas com funções de sicronização em estruturadas para não ocorrer problemas na Thread principal.

1.8.2. Waitgroups

Usando o mesmo exemplo da subseção anterior, façamos o seguinte experimento: ao invés de acionar a *goroutine* levando o valor 5 como argumento, iremos então inverter os valores das suas funções `sequencia()`, que agora contará com os valores 10 e 5, respectivamente. O escopo da função `main()` agora é este:

```
1 func main() {
2     go sequencia(10)
3     sequencia(5)
4 }
```

Agora executando o código com os valores invertidos, temos:

```
1 1 2 2 3 3 4 4 5 5 ...Sequencia de 5 terminou...
```

Analisando a saída, percebemos que o paralelismo houve até a rotina principal terminar. Isso quer dizer que não houve continuidade da *goroutine* da função `imprimir(10)`. Este comportamento acontece porque Go entende que as *goroutines* devem respeitar o tempo de execução da rotina principal, ou seja, o processo `main()`. Para o processo principal esperar o termino de todos os processos em paralelo, se faz necessário o uso de *waitgroups*.

Modificaremos o código adequando-o para *waitgroups*. O primeiro passo deve ser adicionar o pacote `sync` dentro das importações para utilizarmos seus módulos. Devemos também instanciar uma variável global do tipo `sync.WaitGroup`. Faremos então `var group sync.WaitGroup`. Reescreveremos a função `main()` com o uso de `group` previamente instanciada. Fica dessa maneira:

```
1 func main() {
2     group.Add(1)
3     go sequencia(10)
4     sequencia(5)
5     group.Wait()
6 }
```

A função `group.Add()` especifica previamente quantos processos o programa deve esperar para poder ser finalizado. Ao final das chamadas das funções usamos o `group.Wait()`, que tem por finalidade permitir o processo principal esperar todas os processos do grupo terminarem sua execução. Também teremos algumas modificações na função `sequencia()`:

```
1 func sequencia(n int) {
```

```
2       defer group.Done()
3       for i := 0; i < n; i++ {
4           fmt.Printf("%d ", i+1)
5           time.Sleep(200 * time.Millisecond)
6       }
7       fmt.Printf(" ...Sequencia de %d terminou... ", n)
8   }
```

Fizemos o uso do `defer` (explicado na seção 1.5.5)) para assegurar que a função `group.Done()` seja executada ao fim da função `sequencia()`. Basicamente, `group.Done()` tem por finalidade "avisar" o processo principal que a atual rotina terminou e que o programa não precisa mais esperar por ela. Isso resolve o problema descrido no início desta subseção.

1.8.3. Channels

As goroutines se ultilizam de uma melhor abordagem em alta performance e melhor eficiência no uso de recursos de processamento. Porém a capacidade de executar diferentes goroutines concorrentes pode ser uma cituação muito complicada porque geralmente irá se fazer a necessidade de fazer comunicações entre elas.

Os channels são os responsaveis por abstrair essa comunicação, ou seja, eles abrem um canal para servir de ponte e conduzir as informações de qualquer tipagem em Go. A função `make()` é responsavel de criar um canal para trafegar os valores. Vamos ao seguinte exemplo com um tipo `int`. Depois Utilizamos o operador `<-` (arrow operator ou operador seta) para interagir com o cana criado. A seta permite a direção correta do fluxo da comunicação, vamos enviar valores `int` para o canal `ch`:

```
1   ch := make(chan int)
2   ch <- 33
```

Podemos obter o valor enviado para o canal c da sequinte forma:

```
1   valor := <-c
```

A seguir está um exemplo mais completo para maior entendimento da real função dos channels:

```
1   func main() {
2       ch := make(chan int)
3       go induzir(ch)
4       recebido := <-ch
5       fmt.Println(recebido)
6   }
7
8   func induzir(ch chan int) {
9       ch <- 33
10  }
```

Notamos que inicialmente instaciamos um canal `ch` que conduz informações do tipo `int` e logo após passamos para esse canal um valor inteiro por meio do disparo

da goroutine `induzir()`. Isso bloqueia qualquer outra operação com o mesmo canal até que a outra seja completada; permitindo uma propia sinronização automatica entre as goroutines sem mecanismos para travalas. A varivel `recebido :=` recebera o valor trafegado antes pela função textttinduzir(), que em seguida é iimpresso com valor e `33` no console.

1.9. Pacotes Go

1.9.1. Criando um pacote Go

1.9.1.1. Time

1.9.1.2. Io/util e Os

1.9.2. Pacotes externos

1.10. References

Bibliographic references must be unambiguous and uniform. We recommend giving the author names references in brackets, e.g. [Knuth 1984], [Kernighan and Ritchie 1990]; or dates in parentheses, e.g. Knuth (1984), Sederberg and Zundel (1989,1990).

References

[1] Boulic, R. and Renault, O. (1991) "3D Hierarchies for Animation", In: New Trends in Animation and Visualization, Edited by Nadia Magnenat-Thalmann and Daniel Thalmann, John Wiley & Sons ltd., England.

[2] Dyer, S., Martin, J. and Zulauf, J. (1995) "Motion Capture White Paper", http://reality.sgi.com/employees/jam_sb/mocap/MoCapWP_v2.0.html, December.

[3] Holton, M. and Alexander, S. (1995) "Soft Cellular Modeling: A Technique for the Simulation of Non-rigid Materials", Computer Graphics: Developments in Virtual Environments, R. A. Earnshaw and J. A. Vince, England, Academic Press Ltd., p. 449-460.

[4] Knuth, D. E., The TeXbook, Addison Wesley, 1984.