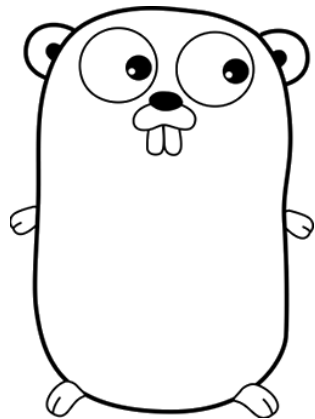

Workshop Go: do Zero à API

Diego Santos, Francisco Oliveira



18-12-2025

Índice

Dia 01	1
O que é Go	1
Um pouco de história	1
Semântica	1
Por que escolher Go?	2
Hello World	3
Separando domínio de efeitos colaterais	4
Tipos básicos	5
String	5
Números	5
Inteiros	6
Alias úteis	6
Tipo especial	7
Booleanos	7
Variáveis / Constantes / Ponteiros	8
Variáveis	8
Constantes	12
Enumerações	14
Ponteiros	17
A função new()	18
Tipos Compostos	18
Array	18
Slice	19
Map	22
Dia 02	23
Estruturas de controle	23
if	23
Switch	24
for	25
for range	26

Struct	27
Struct Nomeada	27
Struct anônima	28
Funções	28
Como declarar uma função?	28
Declaração de função que recebe parâmetros	29
Funções anônimas	29
Função com retorno nomeado	30
Funções variádicas	30
Erros	31
Métodos	32
Interfaces	32
Dia 03	34
Concorrência	34
Goroutines	34
Canais (channels)	34
Defer	36
WaitGroup	36
Pacotes	36
Go Modules: Gerenciamento de Dependências	36
O que é o Go Modules?	36
GOPATH, um pouco de história	36
Configuração do projeto e ativação do Go Modules	37
Referências	38
Projeto	38
API	39
API Rest	39
API em Go com net/HTTP	39
Referências	49

Dia 01

O que é Go

*“Go é uma linguagem de programação de código aberto, desenvolvida pelo Google, projetada para ser **simples, eficiente e confiável**, com **forte suporte à concorrência** e **excelente desempenho em aplicações modernas e escaláveis**.”*

go.dev

Um pouco de história

A linguagem surgiu na **Google**, em 2007, pelas mãos de:

- **Rob Pike**
- **Ken Thompson** (co-criador do Unix)
- **Robert Griesemer**

Foi anunciada publicamente em 2009 e teve sua versão 1.0 lançada em 2012. Desde então, vem sendo adotada em larga escala para infraestrutura, sistemas distribuídos e aplicações em nuvem.

Semântica

Go apresenta certa familiaridade com C, sobretudo pela filosofia de obter o máximo de efeito com o mínimo de recursos e abstrações. Essa característica se reflete em uma sintaxe direta e em decisões de design voltadas à simplicidade e à eficiência.

No entanto, Go não deve ser entendida como uma versão modernizada de C. Trata-se de uma linguagem que incorpora boas ideias de diferentes paradigmas e linguagens, ao mesmo tempo em que elimina funcionalidades que historicamente introduzem complexidade excessiva ou comprometem a confiabilidade do código.

Go é uma linguagem compilada e estaticamente tipada. Embora ofereça suporte a ponteiros, não permite aritmética de ponteiros, reduzindo uma classe inteira de erros comuns em linguagens de baixo

nível. Além disso, é uma linguagem moderna, projetada com primitivas de concorrência simples e eficientes, e conta com gerenciamento automático de memória por meio de garbage collection (GC).

Por que escolher Go?

Go tem se consolidado como uma escolha estratégica para o desenvolvimento de sistemas modernos, especialmente no contexto de aplicações distribuídas, serviços de backend e ambientes em nuvem. Seu design prioriza simplicidade, desempenho, segurança e produtividade, oferecendo um equilíbrio raro entre eficiência de baixo nível e facilidade de uso. A seguir, destacam-se os principais motivos que levam equipes e organizações a adotarem Go.

Código aberto

Go é uma linguagem de código aberto, mantida pela comunidade e apoiada pelo Google. Isso garante transparência, liberdade de uso e evolução contínua, além de permitir que qualquer desenvolvedor contribua para o ecossistema, bibliotecas e ferramentas.

Fácil de aprender

A linguagem foi projetada com uma sintaxe enxuta e consistente, reduzindo a carga cognitiva do desenvolvedor. Mesmo profissionais vindos de linguagens como C, Java ou Python conseguem aprender Go rapidamente, graças à clareza do código e à ausência de construções excessivamente complexas.

Alto desempenho

Go combina desempenho elevado com alta produtividade. Por ser compilada diretamente para código de máquina e possuir um processo de compilação extremamente rápido, a linguagem é adequada tanto para desenvolvimento ágil quanto para aplicações que exigem eficiência e baixo consumo de recursos.

Modelo de concorrência simples

A concorrência é um dos pilares do Go. A linguagem introduz as *goroutines*, threads leves gerenciadas pelo runtime, e os *channels*, que facilitam a comunicação segura entre tarefas concorrentes. Esse modelo torna mais simples escrever código concorrente, escalável e menos propenso a erros.

Portabilidade e multiplataforma

Go é nativamente multiplataforma. O mesmo código-fonte pode ser compilado para diferentes sistemas operacionais e arquiteturas, como Linux, macOS e Windows, facilitando a distribuição de aplicações em ambientes heterogêneos.

Design orientado à nuvem

Go foi pensada desde o início para cenários de nuvem e microsserviços. As aplicações são distribuídas como um único binário estático, que inclui todas as dependências necessárias, simplificando o *deploy* (implantação), reduzindo problemas de compatibilidade e tornando o runtime mais previsível.

Segurança

Por ser uma linguagem estaticamente e fortemente tipada, Go força o desenvolvedor a ser explícito quanto aos tipos de dados utilizados. Isso permite que o compilador detecte uma ampla classe de erros em tempo de compilação, aumentando a confiabilidade e a segurança do código.

Coleta de lixo eficiente

Go possui gerenciamento automático de memória por meio de um *garbage collector* moderno, projetado para minimizar pausas e impacto no desempenho. Esse mecanismo simplifica o desenvolvimento e contribui para aplicações mais estáveis e eficientes.

Biblioteca padrão robusta

A biblioteca padrão do Go é ampla e madura. Ela oferece suporte nativo para servidores HTTP, manipulação de entrada e saída, criptografia, concorrência, testes e muito mais, reduzindo a dependência de bibliotecas externas e acelerando o desenvolvimento de aplicações completas e confiáveis.

Hello World

Assim como em outras linguagens... em Go temos também o clássico **Hello World**:

```
1 // dia_01/exemplos/01-intro/ex1.go
2 package main
3
4 import "fmt"
```

```
5
6 func main() {
7     fmt.Println("Hello, World")
8 }
```

Para executar este código, você pode começar com o comando:

```
1 go run hello.go
```

Separando domínio de efeitos colaterais

Mas... como podemos testar nosso código?

Inicialmente, vamos isolar o **domínio** — isto é, as regras de negócio — do restante do código que envolve **efeitos colaterais**. A função `fmt.Println` representa um efeito colateral, pois escreve um valor no `stdout` (saída padrão), enquanto a `string` fornecida a ela corresponde ao conteúdo do nosso domínio.

```
1 // dia_01/exemplos/01-intro/ex2.go
2 package main
3
4 import "fmt"
5
6 func Hello() string {
7     return "Hello, World"
8 }
9
10 func main() {
11     fmt.Println(Hello())
12 }
```

Criamos uma nova função: `Hello()`, mas dessa vez adicionamos a palavra `string` na sua definição. Isso significa que essa função retornará uma `string`.

Para validar o comportamento da função `Hello()`, criaremos um arquivo denominado `hello_test.go`, no qual será inserido o código a seguir:

```
1 // dia_01/exemplos/01-intro/hello_test.go
2 package main
3
4 import "testing"
5
6 func TestHello(t *testing.T) {
7     got := Hello()
8     want := "Hello, World"
9
10     if got != want {
```

```
11     t.Errorf("Got '%s', want '%s'", got, want)
12 }
13 }
```

Percebam que não é preciso usar vários *frameworks* (ou bibliotecas) de testes. Tudo o que precisamos está pronto na linguagem e a sintaxe é a mesma para o resto dos códigos que você irá escrever.

Escrever um teste é como escrever uma função, com algumas regras:

- O código precisa estar em um arquivo que termine com `**_test.go`.
- A função de teste precisa começar com a palavra **Test**.
- A função de teste recebe um único argumento, que é `t *testing.T`.

Por enquanto, isso é o bastante para saber que o nosso `t` do tipo `*testing.T` é a nossa porta de entrada para a ferramenta de testes.

Para executar este teste, você pode começar com o comando:

```
1 go test -v ex2.go hello_test.go
```

Tipos básicos

A linguagem Go oferece diversos tipos básicos para representar e manipular dados — desde tipos que refletem diretamente os recursos do hardware até estruturas convenientes para modelar informações mais complexas.

String

Uma **string** em Go é uma **sequência imutável de bytes**. Embora possam conter qualquer dado binário, as strings são normalmente utilizadas para representar texto legível por humanos.

Internamente, strings em Go são codificadas em **UTF-8**, o que significa que podem armazenar caracteres **Unicode** de forma eficiente.



Nota técnica

Para trabalhar com **caracteres Unicode**, usamos o tipo `rune`.

Números

Go oferece uma ampla variedade de tipos numéricos para diferentes tamanhos e precisões.

Inteiros

Tipos com tamanho explícito:

Tipo	Descrição
<code>uint8</code>	conjunto de todos os inteiros sem sinal de 8 bits (de 0 a 255)
<code>uint16</code>	conjunto de todos os inteiros sem sinal de 16 bits (0 to 65535)
<code>uint32</code>	conjunto de todos os inteiros sem sinal de 32 bits (0 to 4294967295)
<code>uint64</code>	conjunto de todos os inteiros sem sinal de 64 bits (0 to 18446744073709551615)
<code>int8</code>	conjunto de todos os inteiros de 8 bits com sinal (-128 a 127)
<code>int16</code>	conjunto de todos os inteiros de 16 bits com sinal (-32768 to 32767)
<code>int32</code>	conjunto de todos os inteiros de 32 bits com sinal (-2147483648 to 2147483647)
<code>int64</code>	conjunto de todos os inteiros de 64 bits com sinal (-9223372036854775808 to 9223372036854775807)

Tipos com tamanho dependente da arquitetura:

Tipo	Descrição
<code>int</code> e <code>uint</code>	assumem o tamanho nativo do compilador (geralmente 32 ou 64 bits)

Alias úteis

Tipo	Descrição
<code>byte</code>	alias para <code>uint8</code> , geralmente usado para representar dados binários
<code>rune</code>	alias para <code>int32</code> , especificamente projetado para armazenar valores inteiros que representam caracteres Unicode codificados em UTF-8

Tipo especial

Tipo	Descrição
<code>uintptr</code>	Inteiro sem sinal que pode armazenar um ponteiro. Usado principalmente em programação de baixo nível, não garante portabilidade entre plataformas

Ponto Flutuante

Tipo	Descrição
<code>float32</code>	conjunto de todos os números de ponto flutuante de 32 bits do padrão IEEE-754
<code>float64</code>	conjunto de todos os números de ponto flutuante de 64 bits do padrão IEEE 754

Complexos

Tipo	Descrição
<code>complex64</code> e <code>complex128</code>	conjunto de todos os números complexos com partes real e imaginária armazenadas como valores <code>float32</code> ou <code>float64</code> . Podem ser criados pela função <code>complex</code>

Booleanos

Em Go, valores booleanos são armazenados usando o tipo `bool`. Embora uma variável do tipo `bool` seja armazenada como um valor de 1 byte, ela não é, no entanto, um alias para um valor numérico. Go fornece dois literais pré-declarados:

Tipo	Descrição
<code>bool</code>	assume os literais <code>true</code> ou <code>false</code>

Variáveis / Constantes / Ponteiros

Variáveis

Go é uma linguagem **fortemente tipada**, o que significa que **toda variável precisa ter um tipo associado** — explícito ou inferido. Cada variável em Go está vinculada a um **nome (identificador)**, um **tipo** e um **valor**.

A forma explícita para declarar uma variável em Go segue o seguinte formato:

```
1 var <lista de identificadores> <tipo>
```

A palavra-chave `var` é usada para declarar um ou mais identificadores de variáveis, seguidos do seus respectivos tipos. O trecho de código a seguir mostra a declaração de diversas variáveis:

```
1 // dia_01/exemplos/02-var/var01.go
2 ...
3 var nome, desc string
4 var diametro int32
5 var massa float64
6 var ativo bool
7 var terreno []string
8 ...
```

O valor zero

O trecho de código anterior mostra vários exemplos de variáveis sendo declaradas com uma variedade de tipos. À primeira vista, parece que essas variáveis não têm um valor atribuído. Na verdade, isso contradiz nossa afirmação anterior de que todas as variáveis em Go estão vinculadas a um tipo e um valor.

Durante a declaração de uma variável, se um valor não for fornecido, o compilador do Go vinculará automaticamente um valor padrão (ou um valor zero) à variável para a inicialização adequada da memória.

A tabela a seguir mostra os tipos do Go e seus valores zero padrão:

Tipo	Valor zero
string	"" (string vazia)

Tipo	Valor zero
Numérico – Inteiro: byte, int, int8, int16, int32, int64, rune, uint, uint8, uint16, uint32, uint64, uintptr	0
Numérico – Ponto flutuante: float32, float64	0.0
booleano	false
Array	Cada índice terá um valor zero correspondente ao tipo do array
Struct	Em uma estrutura vazia, cada membro terá seu respectivo valor zero
Outros tipos: interface, função, canal, slice, mapa e ponteiro	nil

Declaração inicializada

Go também suporta a combinação de declaração de variável e inicialização como uma expressão usando o seguinte formato:

```
1 var <lista de identificadores> <tipo> = <lista de valores ou expressões de inicialização>
```

O seguinte trecho de código mostra a combinação de declaração e inicialização:

```
1 // dia_01/exemplos/02-var/var02.go
2 ...
3 var nome, desc string = "Tatooine", "Planeta"
4 var diametro int32 = 10465
5 var massa float64 = 5.972E+24
6 var ativo bool = true
7 var terreno = []string{
8     "Deserto",
9 }
10 ...
```

Omitindo o tipo das variáveis

Em Go, também é possível omitir o tipo, conforme mostrado no seguinte formato de declaração:

```
1 var <lista de identificadores> = <lista de valores ou expressões de
   inicialização>
```

O compilador do Go irá inferir o tipo da variável com base no valor ou na expressão de inicialização do lado direito do sinal de igual, conforme mostrado no trecho de código a seguir.

```
1 // dia_01/exemplos/02-var/var03.go
2 ...
3 var nome, desc = "Yavin IV", "Lua"
4 var diametro = 10200
5 var massa = 6416930000000000.0
6 var ativo = true
7 var terreno = []string{
8     "Selva",
9     "Florestas Tropicais",
10 }
11 ...
```

Quando o tipo da variável é omitido, as informações de tipo são deduzidas do valor atribuído ou do valor retornado de uma expressão.

A tabela a seguir mostra o tipo que é inferido dado um valor literal:

Valor Literal	Tipo inferido
Texto com aspas duplas ou simples: "Lua Yavin IV" "Sua superfície tem seis continentes ocupando 67% do total."	string
Inteiros: -5101234	int
Decimal: -0.121.01.3e55e-11	float64
Números complexos: -1.0i2i (0+2i)	complex128
Booleanos: true false	bool
Arrays: [2]int{-3, 51}	O tipo do array definido pelo valor literal. Neste caso [2]int
Map: map[string]int{"Tatooine": 10465, "Alderaan": 12500, "Yavin IV": 10200, }	O tipo do map definido pelo valor literal. Neste caso map[string]int
Slice: []int{-3, 51, 134, 0}	O tipo do slice definido pelo valor literal: []int

Valor Literal	Tipo inferido
Struct: <code>struct{nome string; diametro int}{ "Tatooine", 10465, }</code>	O tipo do <code>struct</code> definido conforme o valor literal. Neste caso: <code>struct{nome string; diametro int}</code>
Function: <code>var sqr = func (v int)int { return v * v }</code>	O tipo de <code>function</code> definido na definição literal da função. Neste caso, a variável <code>sqr</code> terá o tipo: <code>func (v int)int</code>

Declaração curta de variável

Em Go é possível reduzir ainda mais a sintaxe da declaração de variáveis. Neste caso, usando o formato *short variable declaration*. Nesse formato, a declaração perde a palavra-chave `var` e a especificação de tipo e passa a usar o operador `:=` (dois-pontos-igual), conforme mostrado a seguir:

```
1 <lista de identificadores> := <lista de valores ou expressões de inicialização>
```

O trecho de código a seguir mostra como usá-la:

```
1 // dia_01/exemplos/02-var/var04.go
2 ...
3 func main() {
4     nome := "Endor"
5     desc := "Lua"
6     diametro := 4900
7     massa := 1.024e26
8     ativo := true
9     terreno := []string{
10         "Florestas",
11         "Montanhas",
12         "Lagos",
13     }
14 ...
```



Restrições na declaração curta de variáveis

Existem algumas restrições quando usamos a declaração curta de variáveis e é muito importante estar ciente para evitar confusão:

- Em primeiro lugar, ela só pode ser usada dentro de um bloco de funções;
- o operador `:=` declara a variável e atribui os valores;
- `:=` não pode ser usado para atualizar uma variável declarada anteriormente;

Declaração de variável em bloco

A sintaxe do Go permite que a declaração de variáveis seja agrupada em blocos para maior legibilidade e organização do código. O trecho de código a seguir mostra a reescrita de um dos exemplos anteriores usando a declaração de variável em bloco:

```
1 // dia_01/exemplos/02-var/var05.go
2 var (
3     nome      string = "Endor"
4     desc      string = "Lua"
5     diametro  int32  = 4900
6     massa     float64 = 1.024e26
7     ativo     bool   = true
8     terreno   = []string{
9         "Florestas",
10        "Montanhas",
11        "Lagos",
12    }
13 )
```

Constantes

Uma constante é um valor com uma representação literal de uma string, um caractere, um booleano ou números. O valor para uma constante é estático e não pode ser alterado após a atribuição inicial.

Constantes tipadas

Usamos a palavra chave **const** para indicar a declaração de uma constante. Diferente da declaração de uma variável, a declaração deve sempre incluir o valor literal a ser vinculado ao identificador, conforme mostrado a seguir:

```
1 const <lista de identificadores> tipo = <lista de valores ou expressões
    de inicialização>
```

O seguinte trecho de código mostra algumas constantes tipadas sendo declaradas:

```
1 // dia_01/exemplos/03-constants/const01.go
2 ...
3 const a1, a2 string = "Workshop", "Go"
4 const b rune = 'G'
5 const c bool = false
6 const d int32 = 2020
7 const e float32 = 2.020
8 const f float64 = math.Pi * 2.0e+3
9 const g complex64 = 20.0i
```

```
10 const h time.Duration = 20 * time.Second
11 ...
```

Note que cada constante declarada recebe explicitamente um tipo. Isso implica que as constantes só podem ser usadas em contextos compatíveis com seus tipos. No entanto, isso funciona de maneira diferente quando o tipo é omitido.

Constantes não tipadas

Constantes são ainda mais interessantes quando não são tipadas. Uma constante sem tipo é declarada da seguinte maneira:

```
1 const <lista de identificadores> = <lista de valores ou expressões de inicialização>
```

Neste formato, a especificação de tipo é omitida na declaração. Logo, uma constante é meramente um bloco de bytes na memória sem qualquer tipo de restrição de precisão imposta. A seguir, algumas declarações de constantes não tipificadas:

```
1 // dia_01/exemplos/03-constants/const02.go
2 ...
3 const i = "G é" + " para Go"
4 const j = 'G'
5 const k1, k2 = true, !k1
6 const l = 111*100000 + 20
7 const m1 = math.Pi / 3.141592
8 const m2 =
    1.41421356237309504880168872420969807856967187537698078569671875376
9 const m3 = m2 * m2
10 const m4 = m3 * 20.0e+400
11 const n = -5.0i * 20
12 const o = time.Millisecond * 20
13 ...
```

A constante `m4` recebe um valor muito grande (`m3 * 20.0e+400`) que é armazenado na memória sem qualquer perda de precisão. Isso pode ser útil em aplicações onde realizar cálculos com um alto nível de precisão é extremamente importante.

Atribuindo constantes não tipadas

Mesmo Go sendo uma linguagem fortemente tipada, é possível atribuir uma constante não tipada a diferentes tipos de precisão diferentes, embora compatíveis, sem qualquer reclamação do compilador, conforme mostrada a seguir:


```
1 // dia_01/exemplos/03-constants/const03.go
2 ...
3 const m2 =
4     1.41421356237309504880168872420969807856967187537698078569671875376
5 var u1 float32 = m2
6 var u2 float64 = m2
7 u3 := m2
8 ...
```

O exemplo anterior mostra a constante não tipada `m2` sendo atribuída a duas variáveis de ponto flutuante com diferentes precisões, `u1` e `u2`, e a uma variável sem tipo, `u3`. Isso é possível porque a constante `m2` é armazenada como um valor não tipado e, portanto, pode ser atribuída a qualquer variável compatível com sua representação (um ponto flutuante).

Como `u3` não tem um tipo específico, ele será inferido a partir do valor da constante, e como `m2` representa um valor decimal, o compilador irá inferir seu tipo padrão, um `float64`.

A declaração de constantes também podem ser organizadas em blocos, aumentando a legibilidade do código, conforme a seguir:

```
1 // dia_01/exemplos/03-constants/const04.go
2 ...
3 const (
4     a1, a2 string      = "Workshop", "Go"
5     b      rune        = 'G'
6     c      bool        = false
7     d      int32       = 2020
8     e      float32     = 2.020
9     f      float64     = math.Pi * 20.0e+3
10    g      complex64   = 20.0i
11    h      time.Duration = 20 * time.Second
12 )
13 ...
```

Enumerações

Um interessante uso para constantes é na criação de enumerações. Usando a declaração de blocos, é facilmente possível criar valores inteiros que aumentam numericamente. Para isso, basta atribuir o valor constante pré-declarado `iota` a um identificador de constante na declaração de bloco, conforme mostrado no exemplo a seguir:

```
1 // dia_01/exemplos/04-enum/enum01.go
2 ...
3 const (
4     estrelaHiperGigante = iota
5     estrelaSuperGigante
```

```
6     estrelaBrilhanteGigante
7     estrelaGigante
8     estrelaSubGigante
9     estrelaAna
10    estrelaSubAna
11    estrelaAnaBranca
12    estrelaAnaVermelha
13    estrelaAnaMarrom
14 )
15 ...
```

Nessa situação, o compilador fará o seguinte:

- Declarar cada membro no bloco como um valor constante inteiro não tipado;
- Inicializar a `iota` com o valor zero;
- Atribuir a `iota`, ou zero, ao primeiro membro (`EstrelaHiperGigante`);
- Cada constante subsequente recebe um `int` aumentado em um.

Assim, as constantes da lista receberão os valores de zero até nove.

É importante ressaltar que, sempre que `const` aparecer em um bloco de declaração, o contador é redefinido para zero. No trecho de código seguinte, cada conjunto de constantes é enumerado de zero a quatro:

```
1 // dia_01/exemplos/04-enum/enum02.go
2 ...
3 const (
4     estrelaHiperGigante = iota
5     estrelaSuperGigante
6     estrelaBrilhanteGigante
7     estrelaGigante
8     estrelaSubGigante
9 )
10 const (
11     estrelaAna = iota
12     estrelaSubAna
13     estrelaAnaBranca
14     estrelaAnaVermelha
15     estrelaAnaMarrom
16 )
17 ...
```

Substituindo o tipo padrão de uma enumeração

Por padrão, uma constante enumerada é declarada como um tipo inteiro não tipado. Porém, podemos substituir o tipo padrão provendo explicitamente um tipo numérico, como mostrado a seguir:

```
1 // dia_01/exemplos/04-enum/enum03.go
2 ...
3 const (
4     estrelaAna byte = iota
5     estrelaSubAna
6     estrelaAnaBranca
7     estrelaAnaVermelha
8     estrelaAnaMarrom
9 )
10 ...
```

É possível especificar qualquer tipo numérico que pode representar um inteiro ou um ponto flutuante. No exemplo anterior, cada constante será declarada como um tipo **byte**.

Usando **iota** em expressões

Quando a **iota** aparece em uma expressão, o compilador irá aplicar a expressão para cada valor sucessivo. O exemplo a seguir atribui números pares aos membros do bloco de declaração:

```
1 // dia_01/exemplos/04-enum/enum04.go
2 ...
3 const (
4     estrelaHiperGigante = 2.0 * iota
5     estrelaSuperGigante
6     estrelaBrilhanteGigante
7     estrelaGigante
8     estrelaSubGigante
9 )
10 ...
```

Ignorando valores em enumerações

É possível ignorar certos valores em uma enumeração simplesmente atribuindo a **iota** a um identificador em branco (**_**). No trecho de código a seguir, o valor 0 é ignorado:

```
1 // dia_01/exemplos/04-enum/enum05.go
2 ...
3 const (
4     _ = iota
5     estrelaHiperGigante = 1 << iota
6     estrelaSuperGigante
7     estrelaBrilhanteGigante
8     estrelaGigante
9     estrelaSubGigante
10 )
```

```
11 ...
```

Ponteiros

Go possibilita o uso de ponteiros. Um *ponteiro* é o *endereço* de memória de um valor.

Um ponteiro em Go é definido por operador `*`(asterisco). O trecho de código a seguir mostra um exemplo da utilização de ponteiros:

```
1 var p *int
```

Um ponteiro é definido de acordo com seu tipo de dado.

No código anterior a variável `p` é um ponteiro para um valor do tipo `int`.

Também é possível obter o endereço do valor de uma variável, para isso, utilizamos o operador `&` (e comercial).

```
1 eraOuroSith := 5000
2 p := &eraOuroSith
```

Já o valor referenciado ao ponteiro pode ser acessado usando o operador `*`.

```
1 eraOuroSith := 5000
2 p := &eraOuroSith
3 fmt.Println(*p) // imprime 5000
```

Um exemplo mais completo:

```
1 // dia_01/exemplos/05-pointer/pont01.go
2 ...
3 var p *int
4 eraOuroSith, epIV := 42, 37
5 // ponteiro para eraOuroSith
6 p = &eraOuroSith
7 // valor de eraOuroSith por meio do ponteiro
8 fmt.Printf("Era de Ouro dos Sith - %d anos antes do Ep.IV (%#x)\n", *p,
9           p)
9 // atualiza o valor de eraOuroSith por meio do ponteiro
10 *p = 5000
11 // o novo valor de eraOuroSith
12 fmt.Printf("Era de Ouro dos Sith - %d anos antes do Ep.IV | Atualizado
13           (%#x)\n", *p, p)
13 // ponteiro para epIV
14 p = &epIV
15 // divide epIV por meio do ponteiro
16 *p = *p / 38
17 // o novo valor de epIV
18 fmt.Printf("Star Wars: Ep.IV é o Marco %d (%#x)\n", epIV, p)
```

19 ...

**Nota técnica**Go não permite **aritmética de ponteiros**.**A função `new()`**

Outra forma de criar variáveis em Go, é usando a função `new()`.

A expressão `new(T)` cria uma variável *sem nome* do tipo `T`, inicializa ela com seu valor zero e devolve seu endereço de memória.

```
1 // dia_01/exemplos/06-new/new01.go
2 ...
3     // epIV, do tipo *int, aponta para uma variável sem nome
4     epIV := new(int)
5     // eraOuroSith, do tipo *int, também aponta para uma variável sem
        nome
6     eraOuroSith := new(int)
7     // "0" zero
8     fmt.Println(*eraOuroSith)
9     // novo valor para o int sem nome
10    *eraOuroSith = *epIV - 5000
11    // "-5000"
12    fmt.Println(*eraOuroSith)
13 ...
```

O uso da função `new()` é relativamente raro.

Tipos Compostos

Tipos compostos em Go são tipos criados pela combinação de tipos básicos e tipos compostos.

Array

Array é uma sequência de elementos do mesmo tipo de dados. Um *array* tem um tamanho fixo, o qual é definido em sua declaração, e não pode ser mais alterado.

A declaração de um *array* segue o seguinte formato:

```
1 [<tamanho>]<tipo do elemento>
```

Exemplo:

```
1 var linhaTempo [10]int
```

Arrays também podem ser multidimensionais:

```
1 var mult [3][3]int
```

Iniciando um *array* com valores:

```
1 var linhaTempo = [3]int{0, 5, 19}
```

Você pode usar `...` (reticências) na definição de capacidade e deixar o compilador definir a capacidade com base na quantidade de elementos na declaração.

```
1 // Declaração simplificada
2 linhaTempo := [...]int{0, 5, 19}
```

Neste caso, o tamanho do *array* será 3.

O próximo exemplo mostra como atribuir valores a um *array* já definido:

```
1 // dia_01/exemplos/07-arrays/arr01.go
2 ...
3 var linhaTempo [3]int
4 linhaTempo[0] = 0
5 linhaTempo[1] = 5
6 linhaTempo[2] = 19
7 ...
```

Tamanho de um *array*:

O tamanho de um *array* pode ser obtido por meio da função nativa `len()`.

```
1 // dia_01/exemplos/07-arrays/arr02.go
2 ...
3 // Declaração simplificada
4 linhaTempo := [...]int{0, 5, 19}
5 // imprime 3
6 fmt.Println(len(linhaTempo))
7 ...
```

Slice

Slice é *wrap* flexível e robusto que abstrai um *array*. Em resumo, um *slice* não detém nenhum dado nele. Ele apenas referencia *arrays* existentes.

A declaração de um *slice* é parecida com a de um *array*, mas sem a capacidade definida.

```
1 // dia_01/exemplos/08-slice/slice01.go
2 ...
3 // declaracao com var
4 var s1 []int
5 fmt.Println("Slice 1:", s1)
6 // declaração curta
7 s2 := []int{}
8 fmt.Println("Slice 2:", s2)
9 // tamanho de um slice
10 fmt.Println("Tamanho do slice 1:", len(s1))
11 fmt.Println("Tamanho do slice 2:", len(s2))
12 ...
```

O código anterior criou um *slice* sem capacidade inicial e sem nenhum elemento.

Também é possível criar um *slice* a partir de um array:

```
1 // dia_01/exemplos/08-slice/slice02.go
2 ...
3 1 // Naves do jogo "Star Wars: Battlefront"
4 2 naves := [...]string{
5 3     1: "X-Wing",
6 4     2: "A-Wing",
7 5     3: "Millenium Falcon",
8 6     4: "TIE Fighter",
9 7     5: "TIE Interceptor",
10 8     6: "Imperial Shuttle",
11 9     7: "Slave I",
12 10 }
13 11 // cria um slice de naves[1] até naves[3]
14 12 rebeldes := naves[1:4]
15 13 fmt.Println(rebeldes)
16 ...
```

A sintaxe `s[i:j]` cria um *slice* a partir do *array* `naves` iniciando do índice `i` até o índice `j - 1`. Então, na **linha 12** do código, `naves[1:4]` cria uma representação do *array* `naves` iniciando do índice 1 até o 3. Sendo assim, o *slice* `rebeldes` tem os valores `["X-Wing" "A-Wing" "Millenium Falcon"]`.

Um *slice* pode ser criado usando a função `make()`, uma função nativa que cria um *array* e retorna um *slice* referenciando o mesmo.

A sintaxe da função é a seguinte: `func make([]T, len, cap) []T`.

Neste caso, é passando como parâmetro o **tipo (T)**, o **tamanho (len)** e a **capacidade (cap)**. A capacidade é opcional, e caso não seja informada, seu valor *padrão* será o **tamanho (len)**, que é um campo obrigatório.

```
1 // dia_01/exemplos/08-slice/slice03.go
```

```
2 ...
3 s := make([]int, 5, 5)
4 fmt.Println(s)
5 ...
```

Adicionando elementos a um slice

Como sabemos, *arrays* são limitados em seu tamanho e não podem ser aumentados. Já *Slices*, tem seu tamanho dinâmico e podem receber novos elementos em tempo de execução por meio da função nativa `append`.

A definição da função `append` é a seguinte: `func append(s []T, x ...T) []T`.

A sintaxe, `x ...T` significa que a função aceita um número variável de elementos no parâmetro `x`, desde que respeitem o tipo do *slice*.

```
1 // dia_01/exemplos/08-slice/slice04.go
2 ...
3 // Naves do jogo "Star Wars: Battlefront"
4 rebeldes := [...]string{"X-Wing", "A-Wing", "Millenium Falcon"}
5 imperiais := [...]string{"TIE Fighter", "TIE Interceptor", "Imperial Shuttle", "Slave I"}
6
7 naves := make([]string, 0, 0)
8 fmt.Printf("Cap: %d - %v\n", cap(naves), naves)
9 naves = append(naves, "")
10 fmt.Printf("Cap: %d - %v\n", cap(naves), naves)
11 naves = append(naves, rebeldes[:])
12 fmt.Printf("Cap: %d - %v\n", cap(naves), naves)
13 naves = append(naves, imperiais[:])
14 fmt.Printf("Cap: %d - %v\n", cap(naves), naves)
15 ...
```

Uma questão que pode ter ficado no ar: Se um slice é um *wrap* de um *array*, como ela tem esta flexibilidade?

Bem, o que acontece por *debaixo dos panos* quando um novo elemento é adicionado a um *slice* é o seguinte:

1. Um novo *array* é criado
2. Os elementos do *array* atual são copiados
3. O elemento ou elementos **adicionados** ao *slice* são incluído no *array*
4. É retornado um *slice*, que é uma referência para o novo *array*

Map

Um *Map* é uma estrutura de dados que mantém uma coleção de pares chave/valor. Também conhecido como *hash table* (tabela de dispersão ou tabela hash).

A declaração de um *map* segue o seguinte formato:

```
1 map[k]v
```

Onde *k* é o tipo da chave e *v* o tipo dos valores.

Exemplos de uso de map:

```
1 // dia_01/exemplos/09-map/map01.go
2 ...
3 naves := make(map[string]string)
4
5 naves["YT-1300"] = "Millennium Falcon"
6 naves["T-65"] = "X-Wing"
7 naves["RZ-1"] = "A-Wing"
8 naves["999"] = "Tunder Tanque"
9
10 fmt.Println("Quantidade de naves:", len(naves))
11 fmt.Println(naves)
12 fmt.Printf("Nave do Han Solo: %s\n", naves["YT-1300"])
13
14 fmt.Println("999 não é uma nave. Removendo...")
15 delete(naves, "999")
16
17 fmt.Println("Quantidade de naves atualizada:", len(naves))
18 fmt.Println(naves)
19 ...
```

Dia 02

Estruturas de controle

Estruturas de controle são utilizadas para alterar a forma como o nosso código é executado. Podemos, por exemplo, fazer com que uma parte do nosso código seja repetido várias vezes, ou que seja executado caso uma condição seja satisfeita.

if

O **if** é uma instrução que avalia uma condição booleana. Para entender melhor como ele funciona, vamos analisar o seguinte problema:

Em uma soma de dois números, onde **a** = 2 e **b** = 4, avalie o valor de **c**. Se **c** for igual a 6, imprima na tela “Sua soma está correta!”, caso contrário, imprima “Sua soma está errada!”.

```
1 package main
2
3 func main() {
4     var a, b = 2, 4
5     c := (a + b)
6     if c == 6 {
7         fmt.Println("Sua soma está correta.")
8         return
9     }
10    //uma forma de fazer se não
11    fmt.Println("Sua soma está errada.")
12 }
```

Outro Exemplo:

```
1 package main
2
3 func main() {
4     if 2%2 == 0 {
```

```
5     fmt.Println("É par.")
6 } else {
7     fmt.Println("É impar.")
8 }
9
10    if num := 2 num < 0 {
11        fmt.Println(num, "É negativo.")
12    } else if num < 10 {
13        fmt.Println(num, "Tem um dígito.")
14    } else {
15        fmt.Println(num, "Tem vários dígitos.")
16    }
17 }
```

Switch

A instrução **switch** é uma maneira mais fácil de evitar longas instruções **if-else**. Com ela é possível realizar ações diferentes com base nos possíveis valores de uma expressão.

Exemplo 1

```
1 package main
2
3 func main() {
4     i := 2
5     switch i {
6     case 1:
7         fmt.Println("Valor de ", i, " por extenso é: um")
8     case 2:
9         fmt.Println("Valor de ", i, " por extenso é: dois")
10    case 3:
11        fmt.Println("Valor de ", i, " por extenso é: três")
12    }
13 }
```

O **switch** pode testar valores de qualquer tipo, além de podermos usar vírgula para separar várias expressões em uma mesma condição **case**.

Exemplo 2

```
1 package main
2
3 func main() {
4     switch time.Now().Weekday() {
```

```
5     case time.Saturday, time.Sunday:
6         fmt.Println("É fim de semana.")
7     default:
8         fmt.Println("É dia de semana.")
9     }
10 }
```

O **switch** sem uma expressão é uma maneira alternativa para expressar uma lógica **if-else**.

Exemplo 3

```
1 package main
2
3 func main() {
4     j := 3
5     switch {
6     case 1 == j:
7         fmt.Println("Valor por extenso é: um")
8     case 2 == j:
9         fmt.Println("Valor por extenso é: dois")
10    default:
11        fmt.Println("Valor não encontrado.")
12    }
13 }
```

for

Em outras linguagens de programação temos várias formas de fazer laços de repetição, porém, em **Go** só temos uma forma, e é usando a palavra reservada **for**.

Exemplo 1

A forma tradicional, que já conhecemos, e que no exemplo vai imprimir números de 1 a 10.

```
1 package main
2
3 func main(){
4     for i := 1; i <=10; i++ {
5         fmt.Println("O número é: ", i)
6     }
7 }
```

Exemplo 2

```
1 package main
2
3 func main(){
4     i := 5
5     for i <= 5 {
6         fmt.Println("O número é: ", i)
7         i = i + 1
8     }
9 }
```

Exemplo 3 loop infinito

```
1 package main
2
3 func main(){
4     for {
5         fmt.Println("Olá sou o infinito")
6         break
7     }
8 }
```

for range

Já vimos as outras formas de usar o **for**, agora falta o **range**. Essa expressão espera receber uma lista (array ou slice).

```
1 func exemploFor4() {
2     listaDeCompras := []string{"arroz", "feijão", "melancia", "banana",
3     "maçã", "ovo", "cenoura"}
4     for k, p := range listaDeCompras {
5         retornaNomeFruta(k, p)
6     }
7 }
8 func retornaNomeFruta(key int, str string) {
9     switch str {
10    case "melancia", "banana", "maçã":
11        fmt.Println("Na posição", key, "temos a fruta:", str)
12    default:
13        return
14    }
15 }
```

Struct

Struct é um tipo de dado agregado que agrupa zero ou mais valores nomeados de tipo quaisquer como uma única entidade. Cada valor é chamado de **campo**.

Struct Nomeada

Uma **struct** nomeada recebe um nome em sua declaração. Para exemplificar, criaremos uma **struct** para representar um cadastro de funcionário em uma empresa. Seus campos pode ser acessados através da expressão **variavel.Nome**, exemplo:

```
1 package main
2
3 type Employee struct {
4     ID      int
5     Name    string
6     Age     *time.Time
7     Salary  float64
8     Company string
9 }
10
11 func main() {
12     cl := Employee{}
13     //forma de acesso
14     cl.ID = 1
15     cl.Name = "Diego dos Santos"
16     cl.Age = nil
17     cl.Salary = 100.55
18     cl.Company = "Fliper"
19     fmt.Println("o nome é:", cl.Name, " trabalha na empresa: ", cl.
        Company)
20     //outra forma de popular structs
21     cl1 := Employee{
22         ID:      1,
23         Name:    "Francisco Oliveira",
24         Age:     nil,
25         Salary:  2000.50,
26         Company: "Iron Mountain",
27     }
28     fmt.Println("o nome é:", cl1.Name, " trabalha na empresa: ", cl1.
        Company)
29
30 }
```

Struct anônima

Uma **struct** anônima é tipo sem um nome como referência. Sua declaração é semelhante a uma **declaração rápida de variável**.

Só devemos usar uma **struct** anônima quando não há necessidade de criar um objeto para o dado que será transportado por ela.

```
1 package main
2
3 func main() {
4     inferData("Diego", "Santos")
5     inferData("Francisco", "Oliveira")
6 }
7
8 func inferData(fN, lN string) {
9     name1 := struct{FirstName, LastName string}{FirstName: fN, LastName
10         : lN}
11     fmt.Println("O nome é:", name1.FirstName, name1.LastName)
12 }
```

Funções

Funções são pequenas unidades de códigos que podem abstrair ações, retornar e/ou receber valores.

Como declarar uma função?

Declarar uma função é algo bem simples, utilizando a palavra reservada **func** seguida do identificador.

```
1 package main
2
3 func nomeDaFuncao() {}
```

Essa é a declaração mais simples de função que temos. No exemplo acima criamos uma função que não recebe nenhum parâmetro e não retorna nada, o nome dela poderia ser **fazNada**.

Uma função em Go também é um tipo e pode ser declarada da seguinte forma:

```
1 package main
2
3 type myFunc = func(l, b int) int
```

```
4
5 func main() {
6     soma(func(l, b int) int {
7         return l + b
8     })
9 }
10
11 func soma(fn myFunc) {
12     res := fn(1, 3)
13     fmt.Println(res)
14 }
```

Declaração de função que recebe parâmetros

Podemos declarar uma função que recebe dois números e faz uma multiplicação.

```
1 package main
2
3 func main() {
4     fmt.Println("Resultado é:", multiplica(3, 8))
5 }
6
7 func multiplica(a, b int) int {
8     return (a * b)
9 }
```

Veja que na declaração da `func multiplica(a, b int)` os parametros foram passados um seguido do outro, isso por que eles são do mesmo **tipo** (**int**). Caso fossem de tipos diferentes seria necessário declarar cada tipo separadamente, exemplo `func minhaFunc(str string, i int)`.

Funções anônimas

Go também suporta declaração de funções anônimas. Funções anônimas são úteis quando você deseja definir uma função em linha sem ter que nomeá-la.

```
1 package main
2
3 func main() {
4     fn := exemploAnonimo()
5     fmt.Println("Resultado é:", fn)
6     fmt.Println("Resultado é:", fn)
7     fmt.Println("Resultado é:", fn)
8 }
9
10 func exemploAnonimo() func() int {
```



```
11     i := 0
12     return func() int {
13         i += 1
14         return i
15     }
16 }
```

Função com retorno nomeado

Podemos criar uma função e nomear o retorno da mesma. veja o exemplo:

```
1 package main
2
3 func main() {
4     fn := exemploNomeado()
5     fmt.Println("Nome é:", exemploNomeado("Marcela"))
6     fmt.Println("Nome é:", exemploNomeado("Diego"))
7     fmt.Println("Nome é:", exemploNomeado("Francisco"))
8 }
9
10 func exemploNomeado(str string) (nome string) {
11     nome = str
12     return
13 }
```

Funções variádicas

Função variádica é uma função que pode receber qualquer número de argumentos à direita e de um mesmo tipo. Um bom exemplo de função variádica é a função `fmt.Println`. A função pode ser chamada de forma usual, com argumentos individuais ou uma lista (`Slice`).

```
1 package main
2
3 func main() {
4     fmt.Println("Resultado é:", exemploVariadico(1,2))
5     fmt.Println("Resultado é:", exemploVariadico(2,3))
6     fmt.Println("Resultado é:", exemploVariadico(3,4))
7 }
8
9 func exemploVariadico(numeros ...int) (total int) {
10     total = 0
11
12     for _, n := range numeros {
13         total += n
14     }
15     return
16 }
```

```
16 }
```

Erros

Erros são um assunto muito complexo em Go, pois não existe um tratamento de exceção como em outras linguagens. A única forma de se tratar *erros* em Go é usando a condição *if* ou então podemos criar uma função para realizar o tratamento. veja os exemplos:

```
1 package main
2
3 func main() {
4     tot, err := exemploVariadicoWithErr(1,2)
5     if err != nil {
6         return
7     }
8     fmt.Println("Resultado é:", tot)
9
10    tot2, err := exemploVariadicoWithErr(2,3)
11    if err != nil {
12        return
13    }
14    fmt.Println("Resultado é:", tot2)
15
16    tot3, err := exemploVariadicoWithErr(3,4)
17    checkErr(err)
18    fmt.Println("Resultado é:", tot3)
19 }
20
21 func exemploVariadicoWithErr(numeros ...int) (total int, err error) {
22     total = 0
23
24     for _, n := numeros {
25         total += n
26     }
27     if total == 0 {
28         err = errors.New("0 resultado não pode ser zero")
29         return
30     }
31     return
32 }
33
34 func checkErr(err error) {
35     if err != nil {
36         return
37     }
38 }
```

Como mostrado no exemplo acima, uma função pode retornar algum resultado e/ou erro.

Métodos

Métodos em Go são uma variação da declaração de função. No método, um parâmetro extra aparece antes do nome da função e é chamado de receptor (*receiver*).

Métodos podem ser definidos para qualquer tipo de receptor, até mesmo ponteiros, exemplo:

```
1 package main
2
3 type area struct {
4     Largura int
5     Altura  int
6 }
7
8 func (r *area) CalculaArea() int {
9     res := r.Largura * r.Altura
10    return res
11 }
12
13 func (r area) CalculaPerimetro() int {
14     res := 2*r.Largura * 2*r.Altura
15     return res
16 }
17
18
19 func main() {
20     a := area{Largura: 10, Altura: 5}
21     resultArea := a.CalculaArea()
22     fmt.Println("area: ", resultArea)
23     perim := &a //repassando os valores
24     resultPerim := perim.CalculaPerimetro()
25     fmt.Println("perim: ", resultPerim)
26 }
```

Interfaces

Uma **interface** é uma coleção de métodos que um tipo concreto deve implementar para ser considerado uma instância dessa interface. Portanto, uma **interface** define, mas, não declara o comportamento do tipo.

Para exemplificar vamos usar os mesmos exemplos que usamos para criar métodos.

```
1 package main
```

```
2
3 import "fmt"
4
5 // Geo interface base para figuras geométricas
6 type Geo interface {
7     Area() float64
8 }
9
10 // Retângulo representa um retângulo
11 type Retangulo struct {
12     Largura float64
13     Altura  float64
14 }
15
16 // Area calcula a are de um retângulo
17 func (r *Retangulo) Area() float64 {
18     res := r.Largura * r.Altura
19     return res
20 }
21
22 // Triângulo representa um triângulo
23 type Triangulo struct {
24     Base    float64
25     Altura  float64
26 }
27
28 // Area calcula a are de um triângulo
29 func (t *Triangulo) Area() float64 {
30     res := (t.Base * t.Altura) / 2
31     return res
32 }
33
34 func imprimeArea(g Geo) {
35     fmt.Println(g)
36     fmt.Println(fmt.Sprintf("Área      : %0.2f", g.Area()))
37 }
38
39 func main() {
40     r := Retangulo{
41         Altura: 10,
42         Largura: 5,
43     }
44
45     t := Triangulo{
46         Base:    10,
47         Altura: 5,
48     }
49
50     imprimeArea(&r)
51     imprimeArea(&t)
52 }
```

Dia 03

Concorrência

Goroutines

Em Go cada atividade que executa de forma concorrente é chamada de *goroutine*.

Normalmente desenvolver software com programação concorrente nunca é simples, mas Go tornou isso mais fácil do que em outras linguagens. A criação de uma thread **também chamada de go-routine** é praticamente trivial no dia a dia do desenvolvedor.

No exemplo abaixo podemos ver como é feita a chamada de uma goroutine.

```
1 package main
2
3 func exemploGoroutine(str string) {
4     for i := 1; i < 3; i++ {
5         fmt.Printf("%s: %d", str, i)
6     }
7 }
8
9 func main() {
10     exemploGoroutine("direto") // espera que ela retorne
11     go exemploGoroutine("com go routine") //cria uma goroutine e não
12     espera que retorne.
13 }
```

Canais (channels)

Se goroutines são atividades de um programa concorrente, canais(channels) são as conexões entre elas. Um canal é uma sistema de comunicação que permite a uma goroutine enviar valores para outra goroutine.

Canal é um condutor de valores de um tipo particular, chamados de *tipo de elemento* do canal.

```
1 package main
2
```

```
3
4 func main() {
5     sendDataToChannel()
6 }
7
8 func sendDataToChannel() {
9     ch := make(chan int, 1)
10    ch <- 1 //enviando dados para um canal
11    <-ch
12
13    ch <- 2
14    fmt.Println(<-ch)
15 }
```

```
1 package main
2
3
4 func main() {
5     sendDataToChannel()
6 }
7
8 func sendDataToChannel() {
9     ch := make(chan int)
10    ch <- 1 //enviando dados para um canal
11    <-ch
12
13    ch <- 2
14    fmt.Println(<-ch)
15 }
```

```
1 func doisTresQuatroVezes(base int, c chan int) {
2     time.Sleep(time.Second)
3     c <- 2 * base // enviando dados para o canal
4
5     time.Sleep(time.Second)
6     c <- 3 * base
7
8     time.Sleep(3 * time.Second)
9     c <- 4 * base
10 }
11 func main() {
12     c := make(chan int)
13     go doisTresQuatroVezes(2, c)
14
15     a, b := <-c, <-c // recebendo os dados do canal
16     fmt.Println(a, b)
17
18     fmt.Println(<-c)
19 }
```

Defer

TODO

WaitGroup

TODO

Pacotes

TODO

Go Modules: Gerenciamento de Dependências

Nos últimos anos houve muita turbulência em torno do gerenciamento de dependências do Go. Surgiram diversas ferramentas, como `dep`, `godep`, `govendor` e um monte de outras, que entraram em cena para tentar resolver esse problema de uma vez por todas.

O que é o Go Modules?

É o novo sistema de gerenciamento de dependências do Go que torna explícita e fácil o gerenciamento das informações sobre versões de dependências > The Go Blog - Using Go Modules

Em poucas palavras, *Go Modules* é a resposta oficial para lidarmos com o **Gerenciamento de Dependências em Go**.

GOPATH, um pouco de história

O lançamento da versão 1.13 possibilitou a criação do diretório do projeto em qualquer lugar no computador, inclusive no diretório `GOPATH`. Em versões pré-1.13 e pós-1.11, já era possível criar o diretório em qualquer lugar, porém o recomendado era criá-lo fora do diretório `GOPATH`.

Esta é uma grande mudança em relação as versões anteriores do Go (pré-1.11), onde a prática recomendada era criar o diretório dos projetos dentro de uma pasta `src` sob o diretório `GOPATH`, conforme mostrado a seguir:

```
$GOPATH
├── bin
├── pkg
└── src
    └── github.com
        └── <usuário github>
            └── <projeto>
```

Figura 1: Estrutura \$GOPATH

Nessa estrutura, os diretórios possuem as seguintes funções:

- **bin**: Guardar os executáveis de nossos programas;
- **pkg**: Guardar nossas bibliotecas e bibliotecas de terceiros;
- **src**: Guardar todo o código dos nossos projetos.

De forma resumida:

- Versões pré-1.11: A recomendação é criar o diretório do projeto sob o diretório **GOPATH**;
- Versões pós-1.11 e pré-1.13: A recomendação é criar o diretório do projeto fora do **GOPATH**;
- Versão 1.13: O diretório do projeto pode ser criado em qualquer lugar no computador.

Configuração do projeto e ativação do Go Modules

Para utilizar módulos no seu projeto, abra seu terminal e crie um novo diretório para o projeto chamado **buscape** em qualquer lugar em seu computador.

Dica: Crie o diretório do projeto em **\$HOME/workshop**, mas você pode escolher um local diferente, se desejar.

```
1 $ mkdir -p $HOME/workshop/buscape
```

A próxima coisa que precisamos fazer é informar ao Go que queremos usar a nova funcionalidade de módulos para ajudar a gerenciar e controlar a versão de quaisquer pacotes de terceiros que o nosso projeto importe.

Para fazer isso, primeiro precisamos decidir qual deve ser o caminho do módulo para o nosso projeto.

O importante aqui é a singularidade. Para evitar possíveis conflitos de importação com os pacotes de outras pessoas ou com a *Standard Library* (biblioteca padrão) no futuro, escolha um caminho de módulo que seja globalmente exclusivo e improvável de ser usado por qualquer outra coisa. Na comunidade Go, uma convenção comum é criar o caminho do módulo com base em uma URL que você possui.

Se você estiver criando um pacote ou aplicativo que possa ser baixado e usado por outras pessoas e programas, é recomendável que o caminho do módulo seja igual ao local do qual o código pode ser baixado. Por exemplo, se o seu pacote estiver hospedado em <https://github.com/foo/bar>, o caminho do módulo para o projeto deverá ser `github.com/foo/bar`.

Supondo que estamos usando o github, vamos iniciar os módulos da seguinte forma:

```
1 $ cd $HOME/workshop/buscapep
2 $ go mod init github.com/[SEU_USUARIO_GITHUB]/buscapep
3
4 // Saída no console
5 go: creating new go.mod: module github.com/[SEU_USUARIO_GITHUB]/buscapep
```

Neste ponto, o diretório do projeto já deve possuir o arquivo `go.mod` criado.

Não há muita coisa nesse arquivo e se você abrí-lo em seu editor de texto, ele deve ficar assim (**mas de preferência com seu próprio caminho de módulo exclusivo**):

```
1 module github.com/[SEU_USUARIO_GITHUB]/buscapep
2
3 go 1.13
```

Basicamente é isso! Nosso projeto já está configurado e com o Go Modules habilitado.

Referências

- Using Go Modules
- Migrating to Go Modules
- Publishing Go Modules
- Go Modules: v2 and Beyond

Projeto

Como projeto final, vamos desenvolver uma API que vai funcionar como um proxy para alguns serviços de CEP.

A ideia é utilizar a concorrência do Go para realizar diversas requisições simultâneas para cada um dos serviços de CEP e pegar a resposta do serviço que responder mais rapidamente.

API

Se você já está na área de TI (tecnologia da informação) há algum tempo, provavelmente já deve ter ouvido o termo API pelo menos uma vez. Mas, o que é essa API?

*“API (do Inglês **Application Programming Interface**) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por aplicativos que não pretendem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços”*
> pt.wikipedia.org

API Rest

Atualmente, boa parte das APIs escritas são APIs web e tendem a seguir o estilo **Rest**.

O que é REST?

REST é acrônimo para **RE**presentational **S**tate **T**ransfer. É um estilo arquitetural para sistemas de hipermídia distribuídos e foi apresentado pela primeira vez por **Roy Fielding** em 2000 em sua famosa dissertação.

API em Go com net/HTTP

O suporte **HTTP** em Go é fornecido pelo pacote da biblioteca padrão `net/http`. Dito isso, vamos fazer a primeira iteração da nossa API.

Começaremos com os três itens essenciais:

- O primeiro item que precisamos é de um **manipulador** (ou **handler**). Se você tem experiência com MVC, pode pensar em manipuladores (handlers) como sendo os controladores. Eles são responsáveis pela execução da lógica da aplicação e pela criação de cabeçalhos e do corpo da resposta HTTP.
- O segundo item é um roteador (ou **servermux** na terminologia do Go). Ele armazenará o mapeamento entre os padrões de URL da aplicação e os manipuladores (handlers) correspondentes. Normalmente temos um `servermux` para a aplicação contendo todas as rotas.

- O último item que precisamos é um servidor web. Uma das grandes vantagens do Go é que você pode estabelecer um servidor Web e tratar solicitações recebidas como parte da própria aplicação. Você não precisa de um servidor de terceiros como o Nginx ou o Apache.

Vamos juntar esses itens, os conceitos vistos até aqui e criar uma aplicação didática e funcional.

Primeiramente, acesse o diretório do projeto configurado anteriormente e crie um arquivo chamado `main.go`:

```
1 $ cd $HOME/workshop/buscapep
2 # a criação do arquivo pode ser realizada dentro da própria IDE / Editor
  de texto
3 $ touch main.go
```

E digite o código a seguir:

```
1 // server01.go -> Referência para o arquivo no diretório exemplos
2 package main
3
4 import (
5     "log"
6     "net/http"
7 )
8
9 // Defina uma função manipuladora (handler) chamada "home" que escreve
10 // um slice de bytes contendo "Bem vindo a API de CEPs" no o corpo da
   resposta.
11 func home(w http.ResponseWriter, r *http.Request) {
12     w.Write([]byte("Bem vindo a API de CEPs"))
13 }
14
15 func main() {
16     // Use a função http.NewServeMux() para inicializar um novo
       servermux,
17     // depois registre a função "home" como manipulador do padrão de
       URL "/".
18     mux := http.NewServeMux()
19     mux.HandleFunc("/", home)
20
21     // Use a função http.ListenAndServe() para iniciar um novo servidor
       web.
22     // Passamos dois parâmetros: o endereço de rede TCP que será
       escutado
23     // (neste caso ":4000") e o servermux que acabamos de criar.
24     // Se http.ListenAndServe() retornar um erro, usamos a função
25     // log.Fatal() para registrar a mensagem de erro e sair.
26     log.Println("Iniciando o servidor na porta: 4000")
27     err := http.ListenAndServe(":4000", mux)
28     log.Fatal(err)
29 }
```

Considerando que você está no diretório onde está o arquivo `main.go`, para executar o código anterior, execute:

```
1 $ go run main.go
```

E para testar, abra o navegador e digite a URL `http://localhost:4000` ou execute o seguinte comando:

```
1 $ curl localhost:4000
```

Rotas parametrizadas

Quando acessamos a URL `/cep/04167001`, queremos obter informações sobre o CEP 04167001. A primeira coisa a ser feita é obter o CEP a partir da URL e isso pode ser feito da seguinte maneira:

```
1 // server02.go -> Referência para o arquivo no diretório exemplos
2 ...
3 // novo - função manipuladora (handler)
4 func cepHandler(w http.ResponseWriter, r *http.Request) {
5     cep := r.URL.Path[len("/cep/"): ]
6     w.Write([]byte(cep))
7 }
8
9 func main() {
10     mux := http.NewServeMux()
11     mux.HandleFunc("/", home)
12     // novo padrão
13     mux.HandleFunc("/cep/", cepHandler)
14
15     log.Println("Iniciando o servidor na porta: 4000")
16     err := http.ListenAndServe(":4000", mux)
17     log.Fatal(err)
18 }
19 ...
```

Nota sobre rotas parametrizadas: Go não suporta roteamento baseado em método ou URLs semânticos com variáveis (`/cep/{cep}`). Idealmente, não devemos verificar o caminho da URL dentro do nosso manipulador (handler), devemos usar alguma lib de terceiros que faça isso para nós.

JSON

JSON (*JavaScript Object Notation*) é uma notação padrão para o envio e recebimento de informações estruturadas.

Sua simplicidade, legibilidade e suporte universal o tornam, atualmente, a notação mais amplamente utilizada.

Go tem um suporte excelente para codificação e decodificação de JSON oferecidos pelo pacote da biblioteca padrão `encoding/json`.

```
1 // server03.go -> Referência para o arquivo no diretório exemplos
2 ...
3 type cep struct {
4     Cep      string `json:"cep"`
5     Cidade    string `json:"cidade"`
6     Bairro    string `json:"bairro"`
7     Logradouro string `json:"logradouro"`
8     UF        string `json:"uf"`
9 }
10 ...
11 func cepHandler(w http.ResponseWriter, r *http.Request) {
12     rCep := r.URL.Path[len("/cep/"): ]
13     c := cep{Cep: rCep}
14     ret, err := json.Marshal(c)
15     if err != nil {
16         log.Printf("Ops! ocorreu um erro: %s", err.Error())
17         http.Error(w, http.StatusText(http.StatusBadRequest), http.
18             BadRequest)
19     }
20     w.Write([]byte(ret))
21 }
22 ...
```

O processo de converter uma estrutura (`struct`) Go para JSON chama-se *marshaling* e, como visto, é feito por `json.Marshal`.

O resultado de uma chamada a nossa API pode ser algo semelhante ao JSON a seguir:

```
1 {
2     "cep": "04167001",
3     "cidade": "",
4     "bairro": "",
5     "logradouro": "",
6     "uf": ""
7 }
```

Como pode ser percebido, nosso resultado apresenta campos vazios.

Caso seja necessário, isso pode ser contornado por meio do uso da opção adicional `omitempty`:

```
1 // server04.go -> Referência para o arquivo no diretório exemplos
2 ...
3 type cep struct {
4     Cep      string `json:"cep"`
```

```
5     Cidade      string `json:"cidade,omitempty"`
6     Bairro      string `json:"bairro,omitempty"`
7     Logradouro  string `json:"logradouro,omitempty"`
8     UF          string `json:"uf,omitempty"`
9 }
10 ...
```

Cliente HTTP

Um cliente HTTP também pode ser criado com Go para consumir outros serviços com o mínimo de esforço. Como é mostrado no seguinte trecho de código, o código do cliente usa o tipo `http.Client` para se comunicar com o servidor:

```
1 // server05.go -> Referência para o arquivo no diretório exemplos
2 ...
3 var endpoints = map[string]string{
4     "viacep":      "https://viacep.com.br/ws/%s/json/",
5     "postmon":     "https://api.postmon.com.br/v1/cep/%s",
6     "republicavirtual": "https://republicavirtual.com.br/web_cep.php?
                          cep=%s&formato=json",
7 }
8 ...
9 func cepHandler(w http.ResponseWriter, r *http.Request) {
10     rCep := r.URL.Path[len("/cep/"):]
11
12     endpoint := fmt.Sprintf(endpoints["postmon"], rCep)
13
14     client := http.Client{Timeout: time.Duration(time.Millisecond *
15         600)}
16     resp, err := client.Get(endpoint)
17     if err != nil {
18         log.Printf("Ops! ocorreu um erro: %s", err.Error())
19         http.Error(w, http.StatusText(http.StatusInternalServerError),
20             http.StatusInternalServerError)
21         return
22     }
23     defer resp.Body.Close()
24
25     requestContent, err := ioutil.ReadAll(resp.Body)
26     if err != nil {
27         log.Printf("Ops! ocorreu um erro: %s", err.Error())
28         http.Error(w, http.StatusText(http.StatusInternalServerError),
29             http.StatusInternalServerError)
30         return
31     }
32
33     w.Write([]byte(requestContent))
34 }
```

32 ...

Padronizando nosso retorno

Tudo lindo e maravilhoso, só que se analisarmos os retornos de cada serviço de CEP, veremos que existe uma certa divergência entre eles:

```
1 // http://republicavirtual.com.br/web_cep.php?cep=01412100&formato=json
2 {
3     "resultado": "1",
4     "resultado_txt": "sucesso - cep completo",
5     "uf": "SP",
6     "cidade": "São Paulo",
7     "bairro": "Cerqueira César",
8     "tipo_logradouro": "Rua",
9     "logradouro": "Augusta"
10 }
11
12 // http://api.postmon.com.br/v1/cep/01412100
13 {
14     "complemento": "de 2440 ao fim - lado par",
15     "bairro": "Cerqueira César",
16     "cidade": "São Paulo",
17     "logradouro": "Rua Augusta",
18     "estado_info": {
19         "area_km2": "248.221,996",
20         "codigo_ibge": "35",
21         "nome": "São Paulo"
22     },
23     "cep": "01412100",
24     "cidade_info": {
25         "area_km2": "1521,11",
26         "codigo_ibge": "3550308"
27     },
28     "estado": "SP"
29 }
30
31 // https://viacep.com.br/ws/01412100/json/
32 {
33     "cep": "01412-100",
34     "logradouro": "Rua Augusta",
35     "complemento": "de 2440 ao fim - lado par",
36     "bairro": "Cerqueira César",
37     "localidade": "São Paulo",
38     "uf": "SP",
39     "unidade": "",
40     "ibge": "3550308",
41     "gia": "1004"
42 }
```

Sendo assim, vamos tratar cada retorno e padronizá-lo:

```
1 // server06.go -> Referência para o arquivo no diretório exemplos
2 ...
3 func cepHandler(w http.ResponseWriter, r *http.Request) {
4     rCep := r.URL.Path[len("/cep/"): ]
5
6     endpoint := fmt.Sprintf(endpoints["republicavirtual"], rCep)
7
8     client := http.Client{Timeout: time.Duration(time.Millisecond *
9         600)}
10    resp, err := client.Get(endpoint)
11    if err != nil {
12        log.Printf("Ops! ocorreu um erro: %s", err.Error())
13        http.Error(w, http.StatusText(http.StatusInternalServerError),
14            http.StatusInternalServerError)
15        return
16    }
17    defer resp.Body.Close()
18
19    requestContent, err := ioutil.ReadAll(resp.Body)
20    if err != nil {
21        log.Printf("Ops! ocorreu um erro: %s", err.Error())
22        http.Error(w, http.StatusText(http.StatusInternalServerError),
23            http.StatusInternalServerError)
24        return
25    }
26
27    // Novo
28    c, err := parseResponse(requestContent)
29    if err != nil {
30        log.Printf("Ops! ocorreu um erro: %s", err.Error())
31        http.Error(w, http.StatusText(http.StatusInternalServerError),
32            http.StatusInternalServerError)
33        return
34    }
35
36    c.Cep = rCep
37    ret, err := json.Marshal(c)
38    if err != nil {
39        log.Printf("Ops! ocorreu um erro: %s", err.Error())
40        http.Error(w, http.StatusText(http.StatusInternalServerError),
41            http.StatusInternalServerError)
42        return
43    }
44
45    w.Write([]byte(ret))
46 }
47
48 func parseResponse(content []byte) (payload cep, err error) {
49     response := make(map[string]interface{})
```



```
45     _ = json.Unmarshal(content, &response)
46
47     if err := isValidResponse(response); !err {
48         return payload, errors.New("invalid response")
49     }
50
51     if _, ok := response["localidade"]; ok {
52         payload.Cidade = response["localidade"].(string)
53     } else {
54         payload.Cidade = response["cidade"].(string)
55     }
56
57     if _, ok := response["estado"]; ok {
58         payload.UF = response["estado"].(string)
59     } else {
60         payload.UF = response["uf"].(string)
61     }
62
63     if _, ok := response["logradouro"]; ok {
64         payload.Logradouro = response["logradouro"].(string)
65     }
66
67     if _, ok := response["tipo_logradouro"]; ok {
68         payload.Logradouro = response["tipo_logradouro"].(string) + " "
69         + payload.Logradouro
70     }
71     payload.Bairro = response["bairro"].(string)
72
73     return
74 }
75
76 func isValidResponse(requestContent map[string]interface{}) bool {
77     if len(requestContent) <= 0 {
78         return false
79     }
80
81     if _, ok := requestContent["erro"]; ok {
82         return false
83     }
84
85     if _, ok := requestContent["fail"]; ok {
86         return false
87     }
88
89     return true
90 }
91 ...
```

Acertando o cabeçalho da resposta

Ao enviar uma resposta, o Go definirá automaticamente três cabeçalhos gerados pelo sistema para você: `Date`, `Content-Length` e `Content-Type`.

O cabeçalho `Content-Type` é particularmente interessante. O Go tentará defini-lo de maneira correta, analisando o corpo da resposta com a função `http.DetectContentType()`.

Se essa função não conseguir detectar o tipo de conteúdo, o cabeçalho será definido como `Content-Type: application/octet-stream`.

A função `http.DetectContentType()` geralmente funciona muito bem, mas uma dica para desenvolvedores Web novos no Go é que ela não consegue distinguir `JSON` de texto sem formatação. E, por padrão, as respostas `JSON` serão enviadas com um cabeçalho `Content-Type: text/plain; charset=utf-8`. Para impedir que isso aconteça, é necessário definir o cabeçalho correto manualmente da seguinte maneira:

```
1 // server07.go -> Referência para o arquivo no diretório exemplos
2 ...
3 func cepHandler(w http.ResponseWriter, r *http.Request) {
4     ...
5     // Acertando o cabeçalho
6     w.Header().Set("Content-Type", "application/json")
7     w.Write([]byte(ret))
8 }
9 ...
```

Gran finale

Para finalizar, vamos adicionar um pouco de concorrência em nossa aplicação:

```
1 // server08.go -> Referência para o arquivo no diretório exemplos
2 package main
3
4 import (
5     ...
6     "regexp" // Novo
7     "time"
8 )
9
10 type cep struct {
11     ...
12 }
13
14 // novo
15 func (c cep) exist() bool {
16     return len(c.UF) != 0
17 }
```

```
17 }
18 ...
19
20 // Função cepHandler foi refatorada e dela extraímos a função request
21 func cepHandler(w http.ResponseWriter, r *http.Request) {
22     // Restringindo o acesso apenas pelo método GET
23     if r.Method != http.MethodGet {
24         http.Error(w, http.StatusText(http.StatusMethodNotAllowed),
25             http.StatusMethodNotAllowed)
26         return
27     }
28     rCep := r.URL.Path[len("/cep/"): ]
29     rCep, err := sanitizeCEP(rCep)
30     if err != nil {
31         http.Error(w, err.Error(), http.StatusBadRequest)
32         return
33     }
34
35     ch := make(chan []byte, 1)
36     for _, url := range endpoints {
37         endpoint := fmt.Sprintf(url, rCep)
38         go request(endpoint, ch)
39     }
40
41     w.Header().Set("Content-Type", "application/json")
42     for index := 0; index < 3; index++ {
43         cepInfo, err := parseResponse(<-ch)
44         if err != nil {
45             continue
46         }
47
48         if cepInfo.exist() {
49             cepInfo.Cep = rCep
50             json.NewEncoder(w).Encode(cepInfo)
51             return
52         }
53     }
54
55     http.Error(w, http.StatusText(http.StatusNoContent), http.
56         StatusNoContent)
57 }
58 // novo
59 func request(endpoint string, ch chan []byte) {
60     start := time.Now()
61
62     c := http.Client{Timeout: time.Duration(time.Millisecond * 300)}
63     resp, err := c.Get(endpoint)
64     if err != nil {
65         log.Printf("Ops! ocorreu um erro: %s", err.Error())
```

```
66     ch <- nil
67     return
68 }
69 defer resp.Body.Close()
70
71 requestContent, err := ioutil.ReadAll(resp.Body)
72 if err != nil {
73     log.Printf("Ops! ocorreu um erro: %s", err.Error())
74     ch <- nil
75     return
76 }
77
78 if len(requestContent) != 0 && resp.StatusCode == http.StatusOK {
79     log.Printf("O endpoint respondeu com sucesso - source: %s, Duração: %s", endpoint, time.Since(start).String())
80     ch <- requestContent
81 }
82 }
83
84 ...
85
86 // Função para validar o CEP
87 func sanitizeCEP(cep string) (string, error) {
88     re := regexp.MustCompile(`^[0-9]`)
89     sanitizedCEP := re.ReplaceAllString(cep, `$1`)
90
91     if len(sanitizedCEP) < 8 {
92         return "", errors.New("O CEP deve conter apenas números e no mínimo 8 dígitos")
93     }
94
95     return sanitizedCEP[:8], nil
96 }
97
98 func main() {
99     ...
100 }
```

Referências

- A Tour of Go - Português
- A Tour of Go - English
- Aprenda Go com Testes - Português
- Learn Go with Tests - English
- Go by Example