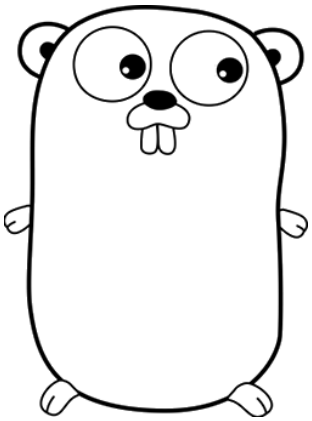
Workshop Go: do Zero à API

Diego Santos, Francisco Oliveira



31-12-2025

Índice

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**:

```go
// dia_01/exemplos/01-intro/ex1.go
package main

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

Tipos com tamanho dependente da arquitetura:

Tipo	Descrição
`int` e `uint`	assumem o tamanho nativo do compilador (geralmente 32 ou 64 bits)

Alias úteis

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

Tipo especial

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

Ponto Flutuante

Tipo	Descrição
`float32`	conjunto de todos os números de ponto flutuante de 32 bits do padrão IEEE-754
`float64`	conjunto de todos os números de ponto flutuante de 64 bits do padrão IEEE 754

Complexos

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

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
`bool`	assume os literais **`true`** ou **`false`**

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:

```
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:

```
// dia_01/exemplos/02-var/var01.go
...
var nome, desc string
var diametro int32
var massa float64
var ativo bool
var terreno []string
...
```

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:

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

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

```
// dia_01/exemplos/02-var/var02.go
...
var nome, desc string = "Tatooine", "Planeta"
var diametro int32 = 10465
var massa float64 = 5.972E+24
var ativo bool = true
var terreno = []string{
    "Deserto",
}
...
```

Omitindo o tipo das variáveis

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

```
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.

```
// dia_01/exemplos/02-var/var03.go
...
var nome, desc = "Yavin IV", "Lua"
var diametro = 10200
var massa = 641693000000000.0
var ativo = true
var terreno = []string{
    "Selva",
    "Florestas Tropicais",
}
...
```

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 temseis continentes ocupando67% do total.‘	string
Inteiros:`-5101234`	int
Decimal:`-0.121.01.3e55e-11`	float64
Números complexos:`-1.0i2i(0+2i)`	complex128
Booleanos:`truefalse`	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:`struct{nome stringdiametro int}{"Tatooine", 10465,}`	O tipo do `struct` definido conforme o valor literal. Neste caso: `struct{nome string; diametro int}`
Function:`var sqr = func (v int)int { return v * v}`	O tipo de `function` definido na definição literal da função. Neste caso, a variável `sqr` terá o tipo: `func (v int)int`

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:

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

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

```
// dia_01/exemplos/02-var/var04.go
...
func main() {
    nome := "Endor"
    desc := "Lua"
    diametro := 4900
    massa := 1.024e26
    ativo := true
    terreno := []string{
        "Florestas",
        "Montanhas",
        "Lagos",
    }
...
```

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:

```go
// dia_01/exemplos/02-var/var05.go
var (
    nome     string  = "Endor"
    desc     string  = "Lua"
    diametro int32   = 4900
    massa    float64 = 1.024e26
    ativo    bool    = true
    terreno          = []string{
        "Florestas",
        "Montanhas",
        "Lagos",
    }
)
```

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:

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

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

```go
// dia_01/exemplos/03-constants/const01.go
...
const a1, a2 string = "Workshop", "Go"
const b rune = 'G'
const c bool = false
const d int32 = 2020
const e float32 = 2.020
const f float64 = math.Pi * 2.0e+3
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 a constantes só podem ser usada 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 tipada. 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:

```go
// dia_01/exemplos/03-constants/const03.go
...
const m2 =
    1.41421356237309504880168872420969807856967187537698078569671875376
var u1 float32 = m2
var u2 float64 = m2
u3 := m2
...
```

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:

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

Enumerações

Um interessante uso para constantes é na criação de enumerações. Usando a declaração de blocos, é facilmente possível criar valore 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:

```go
// dia_01/exemplos/04-enum/enum01.go
...
const (
    estrelaHiperGigante = iota
    estrelaSuperGigante
```

```
    estrelaBrilhanteGigante
    estrelaGigante
    estrelaSubGigante
    estrelaAna
    estrelaSubAna
    estrelaAnaBranca
    estrelaAnaVermelha
    estrelaAnaMarrom
)
...
```

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:

```
// dia_01/exemplos/04-enum/enum02.go
...
const (
    estrelaHiperGigante = iota
    estrelaSuperGigante
    estrelaBrilhanteGigante
    estrelaGigante
    estrelaSubGigante
)
const (
    estrelaAna = iota
    estrelaSubAna
    estrelaAnaBranca
    estrelaAnaVermelha
    estrelaAnaMarrom
)
...
```

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:

```go
// dia_01/exemplos/04-enum/enum03.go
...
const (
    estrelaAna byte = iota
    estrelaSubAna
    estrelaAnaBranca
    estrelaAnaVermelha
    estrelaAnaMarrom
)
...
```

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:

```go
// dia_01/exemplos/04-enum/enum04.go
...
const (
    estrelaHiperGigante = 2.0 * iota
    estrelaSuperGigante
    estrelaBrilhanteGigante
    estrelaGigante
    estrelaSubGigante
)
...
```

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:

```go
// dia_01/exemplos/04-enum/enum05.go
...
const (
    _                   = iota
    estrelaHiperGigante = 1 << iota
    estrelaSuperGigante
    estrelaBrilhanteGigante
    estrelaGigante
    estrelaSubGigante
)
```

```
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,
       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
       (%#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:

```go
var linhaTempo [10]int
```

Arrays também podem ser multidimensionais:

```go
var mult [3][3]int
```

Iniciando um *array* com valores:

```go
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.

```go
// Declaração simplificada
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:

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

Tamanho de um *array*:

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

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

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.

```go
// dia_01/exemplos/08-slice/slice01.go
...
// declaracao com var
var s1 []int
fmt.Println("Slice 1:", s1)
// declaração curta
s2 := []int{}
fmt.Println("Slice 2:", s2)
// tamanho de um slice
fmt.Println("Tamanho do slice 1:", len(s1))
fmt.Println("Tamanho do slice 2:", len(s2))
...
```

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

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

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

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

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

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*.

```
// dia_01/exemplos/08-slice/slice04.go
...
// Naves do jogo "Star Wars: Battlefront"
rebeldes := [...]string{"'X-Wing'", "'A-Wing'", "'Millenium Falcon'"}
imperiais := [...]string{"'TIE Fighter'", "'TIE Interceptor'", "'
    Imperial Shuttle'", "'Slave I'"}

naves := make([]string, 0, 0)
fmt.Printf("Cap: %d - %v\n", cap(naves), naves)
naves = append(naves, "''")
fmt.Printf("Cap: %d - %v\n", cap(naves), naves)
naves = append(naves, rebeldes[:]...)
fmt.Printf("Cap: %d - %v\n", cap(naves), naves)
naves = append(naves, imperiais[:]...)
fmt.Printf("Cap: %d - %v\n", cap(naves), naves)
...
```

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 incluido 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:

```
map[k]v
```

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

Exemplos de uso de map:

```
// dia_01/exemplos/09-map/map01.go
...
naves := make(map[string]string)

naves["YT-1300"] = "Millennium Falcon"
naves["T-65"] = "X-Wing"
naves["RZ-1"] = "A-Wing"
naves["999"] = "Tunder Tanque"

fmt.Println("Quantidade de naves:", len(naves))
fmt.Println(naves)
fmt.Printf("Nave do Han Solo: %s\n", naves["YT-1300"])

fmt.Println("999 não é uma nave. Removendo...")
delete(naves, "999")

fmt.Println("Quantidade de naves atualizada:", len(naves))
fmt.Println(naves)
...
```

Dia 02

Estruturas de controle

Estruturas de controle são ultilizadas para alterar a forma como o nosso código é executado. Podemos, por exemplo, fazer com que uma parte do nosso código seja repitido 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!"*.

```go
package main

func main() {
    var a, b = 2, 4
    c := (a + b)
    if c == 6 {
        fmt.Println("Sua soma está correta.")
        return
    }
    //uma forma d fazer se não
    fmt.Println("Sua soma está errada.")
}
```

Outro Exemplo:

```go
package main

func main() {
    if 2%2 == 0 {
```

```
        fmt.Println("É par.")
    } else {
        fmt.Println("É impar.")
    }

    if num := 2 num < 0 {
        fmt.Println(num, "É negativo.")
    } else if num < 10 {
        fmt.Println(num, "Tem um dígito.")
    } else {
        fmt.Println(num, "Tem vários dígitos.")
    }
}
```

Switch

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

Exemplo 1

```
package main

func main() {
    i := 2
    switch i {
    case 1:
        fmt.Println("Valor de ", i, " por extenso é: um")
    case 2:
        fmt.Println("Valor de ", i, " por extenso é: dois")
    case 3:
        fmt.Println("Valor de ", i, " por extenso é: três")
    }
}
```

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

Exemplo 2

```
package main

func main() {
    switch time.Now().Weekday() {
```

```
    case time.Saturday, time.Sunday:
        fmt.Println("É fim de semana.")
    default:
        fmt.Println("É dia de semana.")
    }
}
```

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

Exemplo 3

```
package main

func main() {
    j := 3
    switch {
    case 1 == j:
        fmt.Println("Valor por extenso é: um")
    case 2 == j:
        fmt.Println("Valor por extenso é: dois")
    default:
        fmt.Println("Valor não encontrado.")
    }
}
```

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.

```
package main

func main(){
    for i := 1; i <=10; i++ {
        fmt.Println("O número é: ", i)
    }
}
```

Exemplo 2

```go
package main

func main(){
    i := 5
    for i <= 5 {
        fmt.Println("O número é: ", i)
        i = i + 1
    }
}
```

Exemplo 3 loop infinito

```go
package main

func main(){
    for {
        fmt.Println("Olá sou o infinito")
        break
    }
}
```

for range

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

```go
func exemploFor4() {
    listaDeCompras := []string{"arroz", "feijão", "melancia", "banana",
        "maçã", "ovo", "cenoura"}
    for k, p := range listaDeCompras {
        retornaNomeFruta(k, p)
    }
}

func retornaNomeFruta(key int, str string) {
    switch str {
    case "melancia", "banana", "maçã":
        fmt.Println("Na posição", key, "temos a fruta:", str)
    default:
        return
    }
}
```

Struct

Stuct 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 `strutc` para representar um cadastro de funcionário em uma empresa. Seus campos pode ser acessados través da expressão `variaval.Name`, exemplo:

```go
package main

type Employee struct {
    ID      int
    Name    string
    Age     *time.Time
    Salary  float64
    Company string
}

func main() {
    cl := Employee{}
    //forma de acesso
    cl.ID = 1
    cl.Name = "Diego dos Santos"
    cl.Age = nil
    cl.Salary = 100.55
    cl.Company = "Fliper"
    fmt.Println("o nome é:", cl.Name, " trabalha na empresa: ", cl.
       Company)
    //oura forma de popular structs
    cl1 := Employee{
        ID:      1,
        Name:    "Francisco Oliveira",
        Age:     nil,
        Salary:  2000.50,
        Company: "Iron Mountain",
    }
    fmt.Println("o nome é:", cl1.Name, " trabalha na empresa: ", cl1.
       Company)

}
```

Struct anônima

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

> Só devemos usar uma `struct` anônima quando não há necessida 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
           : lN}
10     fmt.Println("O nome é:", name1.FirstName, name1.LastName)
11 }
```

Funções

Funções são pequenas unidades de códigos que podem abistrair 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 parametro e não retorna nada, o nome dela poderia ser `fazNada`.

Uma função em Go também é um tipo e pode ser declarada da segunte 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 nessário 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 {
```

```
    i := 0
    return func() int {
        i += 1
        return i
    }
}
```

Função com retorno nomeado

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

```
package main

func main() {
    fn := exemploNomeado()
    fmt.Println("Nome é:", exemploNomeado("Marcela"))
    fmt.Println("Nome é:", exemploNomeado("Diego"))
    fmt.Println("Nome é:", exemploNomeado("Francisco"))
}

func exemploNomeado(str string) (nome string) {
    nome = str
    return
}
```

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`).

```
package main

func main() {
    fmt.Println("Resultado é:", exemploVariadico(1,2))
    fmt.Println("Resultado é:", exemploVariadico(2,3))
    fmt.Println("Resultado é:", exemploVariadico(3,4))
}

func exemploVariadico(numeros ...int) (total int) {
    total = 0

    for _, n := range numeros {
        total += n
    }
    return
```

```
16 }
```

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:

```go
package main

type area struct {
    Largura int
    Altura  int
}

func (r *area) CalculaArea() int {
    res := r.Largura * r.Altura
    return res
}

func (r area) CalculaPerimetro() int {
    res := 2*r.Largura * 2*r.Altura
    return res
}


func main() {
    a := area{Largura: 10, Altura: 5}
    resultArea := a.CalculaArea()
    fmt.Println("area: ", resultArea)
    perim := &a //repassando os valores
    resultPerim := perim.CalculaPerimetro()
    fmt.Println("perim: ", resultPerim)
}
```

`defer`: execução adiada e gerenciamento seguro de recursos

O `defer` é um dos mecanismos mais característicos da linguagem Go. Ele permite adiar a execução de uma função até o momento em que a função envolvente retorna, independentemente de como esse retorno ocorre. Essa característica torna o `defer` uma ferramenta central para **gerenciamento de recursos**, **segurança do fluxo de execução** e **legibilidade do código**.

Em Go, o `defer` não é apenas um atalho sintático; ele expressa uma intenção clara: *"este código deve ser executado quando eu sair deste escopo"*. Essa semântica simples tem implicações profun-

das na forma como escrevemos código robusto, especialmente em funções com múltiplos pontos de retorno.

Funcionamento básico do `defer`

A instrução `defer` agenda a execução de uma chamada de função para ocorrer **após** o retorno da função atual. A avaliação dos argumentos ocorre **imediatamente**, no momento em que o `defer` é declarado, enquanto a execução da função é postergada.

```
// dia_02/exemplos/08-defer/ex1.go
...
func exemplo() {
    defer fmt.Println("mundo!")
    fmt.Print("Olá ")
}
...
```

A saída será:

```
Olá mundo!
```

Mesmo que a função possua múltiplos **`return`**, panics ou fluxos condicionais complexos, as funções deferidas serão executadas de forma garantida.

`defer` e o ciclo de vida da função

As chamadas deferidas são associadas à **ativação da função na pilha de chamadas**. Quando a função começa a retornar, o runtime executa os `defer` registrados antes de desalocar o frame da pilha.

Isso implica que:

- `defer` **não está ligado a blocos**, mas à função inteira;
- não existe **"defer de escopo local"** em Go;
- o momento exato da execução ocorre após a avaliação do **`return`**, mas antes do controle voltar ao chamador.

```
// dia_02/exemplos/08-defer/ex2.go
...
func soma(a, b int) int {
    defer fmt.Println("fim da função soma")
    return a + b
}
...
```

Ordem de execução: LIFO (Last In, First Out)

Uma característica essencial do `defer` é que múltiplas chamadas deferidas são executadas em **ordem inversa** àquela em que foram declaradas, seguindo o modelo de uma pilha (LIFO).

```go
// dia_02/exemplos/08-defer/ex3.go
...
func exemplo() {
    defer fmt.Println("1")
    defer fmt.Println("2")
    defer fmt.Println("3")
}
...
```

Saída:

```
3
2
1
```

Esse comportamento é intencional e extremamente útil para padrões como:

- aquisição e liberação de múltiplos recursos,
- locks encadeados,
- composição segura de operações.

`defer` como ferramenta de gerenciamento de recursos

O uso mais comum de `defer` está associado à liberação de recursos externos, como arquivos, conexões e locks.

```go
// dia_02/exemplos/08-defer/ex4.go
...
file, err := os.Open("dados.txt")
if err != nil {
    return err
}
defer func() {
    file.Close()
    fmt.Println("Arquivo fechado com sucesso!")
}()
...
```

Esse padrão garante que:

- o recurso será liberado exatamente uma vez;
- o código permanece correto mesmo com retornos antecipados;

- a intenção do desenvolvedor fica explícita.

O mesmo princípio se aplica a:

- `defer rows.Close()`
- `defer conn.Close()`
- `defer mu.Unlock()`
- `defer cancel()`

Interação entre `defer` e valores de retorno

Em Go, funções podem ter **valores de retorno nomeados**, e o `defer` pode interagir diretamente com eles.

```
// dia_02/exemplos/08-defer/ex5.go
...
func contador() (n int) {
    defer func() {
        n++
    }()
    return 10
}
...
```

Nesse caso, o valor retornado será `11`, pois:

1. `n` recebe o valor `10`;
2. o `defer` é executado;
3. a função retorna o valor final de `n`.

Esse comportamento é poderoso, mas deve ser usado com cautela, pois pode reduzir a clareza do código se aplicado indiscriminadamente.

Armadilhas clássicas do `defer`

Apesar de sua simplicidade aparente, o `defer` possui algumas armadilhas conhecidas que merecem atenção.

1. `defer` dentro de loops

Uma das armadilhas mais comuns ocorre ao usar `defer` dentro de loops longos.

```
...
for i := 0; i < 1000; i++ {
    f, _ := os.Open(fmt.Sprintf("file%d.txt", i))
    defer f.Close()
}
...
```

Nesse exemplo, **todos os arquivos permanecem abertos até o final da função**, o que pode causar exaustão de recursos.

A abordagem correta é limitar o escopo da função ou extrair a lógica para uma função auxiliar:

```
...
for i := 0; i < 1000; i++ {
    func() {
        f, _ := os.Open(fmt.Sprintf("file%d.txt", i))
        defer f.Close()
        // uso do arquivo
    }()
}
...
```

2. Avaliação imediata dos argumentos

Os argumentos de uma função deferida são avaliados no momento da declaração, não no momento da execução.

```
// dia_02/exemplos/08-defer/ex6.go
...
for i := 0; i < 3; i++ {
    defer fmt.Println(i)
}
...
```

Saída:

```
2
1
0
```

Isso ocorre porque o valor de `i` é capturado a cada iteração, no momento do `defer`.

Esse comportamento é correto e previsível, mas frequentemente causa confusão em closures mais complexas.

3. Custo de performance do `defer`

O `defer` possui um custo maior do que uma chamada direta de função. Embora esse custo tenha sido significativamente reduzido nas versões modernas do Go, ele **não é zero**.

Em código crítico de altíssima performance (loops internos, hot paths), pode ser preferível uma liberação explícita:

```
mu.Lock()
// código crítico
mu.Unlock()
```

Ao invés de:

```
mu.Lock()
defer mu.Unlock()
```

A regra prática é clara:

- priorize **clareza e segurança**;
- otimize apenas quando houver evidência mensurável de impacto.

4. `defer` não substitui controle explícito de fluxo

Embora poderoso, o `defer` não deve ser usado para ocultar lógica complexa ou efeitos colaterais não óbvios. Evite:

- modificar estado global de forma implícita;
- alterar valores de retorno sem clareza;
- criar dependências implícitas difíceis de rastrear.

O uso idiomático do `defer` é **simples, previsível e local**.

Boas práticas no uso de `defer`

- Declare o `defer` **imediatamente após** a aquisição do recurso.
- Use `defer` para liberar recursos, não para lógica de negócio.
- Prefira clareza a micro-otimizações.
- Evite `defer` em loops extensos sem controle de escopo.
- Trate `defer` como parte do contrato de segurança da função.

Considerações finais sobre `defer`

O `defer` é um dos pilares do estilo idiomático de Go. Ele não apenas reduz a complexidade do código, mas também aumenta sua robustez ao garantir que recursos sejam liberados corretamente, mesmo em cenários de erro.

Dominar o `defer` é um passo essencial antes de avançar para concorrência, pois muitos padrões seguros com goroutines, canais e primitivas de sincronização dependem diretamente de seu uso correto.

Tratamento de Erros em Go

O tratamento de erros em Go segue uma abordagem explícita e baseada em valores. Diferentemente de linguagens que utilizam mecanismos de exceção como **`try`**/**`catch`**, Go adota um modelo simples e previsível: funções que podem falhar retornam, além do resultado esperado, um valor do tipo `error`.

Esse modelo não é uma limitação da linguagem, mas uma decisão de design. Em Go, erros fazem parte do fluxo normal de execução e devem ser tratados de forma clara e deliberada pelo código chamador.

Erros como valores

Em Go, `error` é uma interface. Isso permite que erros sejam criados, retornados, comparados, propagados ou enriquecidos com contexto adicional, da mesma forma que qualquer outro valor.

O padrão mais comum — e idiomático — para lidar com erros é a verificação explícita após a chamada de uma função:

```
result, err := algumaFuncao()
if err != nil {
    // decidir como tratar o erro
    return err
}
```

Essa verificação explícita torna o fluxo do programa mais fácil de entender, evita efeitos colaterais implícitos e deixa claro onde e como cada erro é tratado.

Exemplo prático

O exemplo a seguir ilustra uma função que realiza um cálculo simples e retorna um erro caso uma condição inválida seja encontrada:

```go
...
func soma(numeros ...int) (int, error) {
    total := 0
    for _, n := range numeros {
        total += n
    }

    if total == 0 {
        return 0, errors.New("o resultado não pode ser zero")
    }

    return total, nil
}
...
```

Ao consumir essa função, o código chamador deve decidir como lidar com o erro retornado:

```go
func main() {
    total, err := soma(1, 2)
    if err != nil {
        return
    }
    fmt.Println("Resultado:", total)
}
```

Tratamento versus propagação

Nem todo erro deve ser tratado imediatamente. Uma regra prática em Go é:

- **Trate o erro** quando você tem contexto suficiente para tomar uma decisão.
- **Propague o erro** quando não é possível resolvê-lo naquele nível da aplicação.

Propagar um erro geralmente significa retorná-lo ao chamador, preservando o fluxo explícito do programa.

Boas práticas

- Nunca ignore erros silenciosamente.
- Evite funções auxiliares que apenas verificam `err != nil` sem executar nenhuma ação concreta.
- Sempre que possível, adicione contexto ao erro antes de propagá-lo.
- Trate erros no nível mais apropriado da aplicação.

Considerações finais sobre erros

Em Go, erros são parte do contrato das funções e devem ser tratados com a mesma atenção que qualquer outro valor retornado. Essa abordagem explícita favorece código mais legível, previsível e fácil de manter, além de incentivar decisões conscientes sobre falhas e comportamentos excepcionais ao longo do fluxo da aplicação.

Interfaces

Em Go, uma `interface` define um **conjunto de métodos** que representa um comportamento esperado. Qualquer tipo que implemente todos os métodos declarados por uma interface **automaticamente satisfaz esse contrato**, sem a necessidade de declarações explícitas ou palavras-chave especiais.

Esse modelo é conhecido como **implementação implícita de interfaces** e é um dos pilares do design idiomático em Go. Ele reduz o acoplamento entre componentes, favorece a composição e permite que tipos sejam reutilizados em diferentes contextos sem modificações.

Uma interface não descreve *como* um comportamento é implementado, apenas *qual* comportamento é esperado. Os tipos concretos permanecem livres para definir sua própria lógica, desde que respeitem o contrato estabelecido pela interface.

A seguir, veremos um exemplo simples utilizando figuras geométricas, no qual diferentes tipos compartilham um mesmo comportamento: o cálculo de área.

```go
// dia_02/exemplos/09-interfaces/ex1.go
package main

import "fmt"

// Geo define o comportamento básico de figuras geométricas.
type Geo interface {
    Area() float64
}

// Retangulo representa um retângulo.
type Retangulo struct {
    Largura float64
    Altura  float64
}

// Area calcula a área de um retângulo.
func (r *Retangulo) Area() float64 {
    return r.Largura * r.Altura
}
```

```go

// Triangulo representa um triângulo.
type Triangulo struct {
    Base   float64
    Altura float64
}

// Area calcula a área de um triângulo.
func (t *Triangulo) Area() float64 {
    return (t.Base * t.Altura) / 2
}

// imprimeArea recebe qualquer tipo que satisfaça a interface Geo.
func imprimeArea(g Geo) {
    fmt.Printf("Área: %.2f\n", g.Area())
}

func main() {
    r := Retangulo{
        Largura: 5,
        Altura:  10,
    }

    t := Triangulo{
        Base:   10,
        Altura: 5,
    }

    imprimeArea(&r)
    imprimeArea(&t)
}
```

No exemplo acima, tanto `Retangulo` quanto `Triangulo` implementam o método `Area()`. Por esse motivo, ambos satisfazem a interface `Geo` de forma automática e podem ser utilizados pela função `imprimeArea`, que depende apenas do contrato definido pela interface, e não de tipos concretos.

Observe que os métodos `Area()` utilizam **receptores por ponteiro**. Isso significa que apenas `*Retangulo` e `*Triangulo` pertencem ao *method set* que satisfaz a interface `Geo`. Esse detalhe é importante para compreender como Go determina se um tipo implementa ou não uma interface.

Em resumo, interfaces em Go promovem designs mais flexíveis e desacoplados, permitindo que o foco esteja no comportamento — e não na hierarquia de tipos.

Dia 03

Concorrência

Uma das características centrais que impulsionaram a adoção do Go é o seu suporte nativo a abstrações de concorrência simples, explícitas e seguras. Em Go, a concorrência é tratada como um conceito de primeira classe da linguagem, permitindo que programas sejam estruturados como um conjunto de unidades independentes que cooperam entre si.

É importante destacar que **concorrência não é sinônimo de paralelismo**. Concorrência refere-se à estrutura do programa — múltiplas tarefas em progresso — enquanto o paralelismo diz respeito à execução simultânea dessas tarefas em múltiplos núcleos de CPU. Um programa concorrente pode ou não ser paralelo, dependendo do ambiente de execução e das decisões do runtime.

Go adota um modelo no qual a concorrência é expressa de forma declarativa e segura, delegando ao runtime a responsabilidade de escalonamento, sincronização básica e multiplexação de execução.

Goroutines

Em Go, cada unidade de execução concorrente é chamada de *goroutine*. Uma goroutine é uma execução concorrente de uma função, gerenciada pelo runtime da linguagem.

Diferentemente de *threads* do sistema operacional, goroutines são abstrações leves. O runtime de Go multiplexa um grande número de goroutines sobre um conjunto menor de threads do sistema operacional, utilizando um modelo conhecido como **M:N**. Esse modelo permite criar milhares — ou até milhões — de goroutines com baixo custo de memória e troca de contexto.

Historicamente, a programação concorrente é considerada complexa, pois exige do desenvolvedor o gerenciamento explícito de threads, sincronização e estados compartilhados. Go reduz significativamente essa complexidade ao tornar a criação de goroutines uma operação simples, explícita e de baixo custo.

Uma goroutine é iniciada ao prefixar a chamada de uma função com a palavra-chave `go`.

```
1 package main
2
```

```go
import "fmt"

func exampleGoroutine(label string) {
    for i := 1; i <= 3; i++ {
        fmt.Printf("%s: %d\n", label, i)
    }
}

func main() {
    exampleGoroutine("execução direta")
    go exampleGoroutine("execução concorrente")
}
```

Na chamada síncrona, a função é executada no fluxo principal e bloqueia até sua conclusão. Ao utilizar a palavra-chave `go`, a função passa a ser executada como uma goroutine, permitindo que a função chamadora continue sua execução imediatamente.

É importante observar que a ordem de execução entre goroutines **não é determinística** e depende do escalonador do runtime. Além disso, o programa pode ser finalizado antes que uma goroutine conclua sua execução caso não exista um mecanismo explícito de sincronização.

Comunicação entre Goroutines

A criação de goroutines permite expressar múltiplos fluxos de execução concorrentes, mas, em aplicações reais, essas unidades raramente operam de forma completamente isolada. É necessário coordenar trabalho, trocar dados e sincronizar estados de maneira segura e previsível.

Go aborda esse problema por meio de um modelo explícito de comunicação, no qual a coordenação entre goroutines ocorre preferencialmente **por troca de mensagens**, e não pelo compartilhamento direto de memória. Esse modelo reduz a complexidade associada a estados compartilhados, *locks* e condições de corrida.

Nesse contexto, os canais desempenham um papel central: eles estabelecem um meio estruturado e seguro para que goroutines cooperem entre si, tornando o fluxo de dados e a sincronização parte explícita da estrutura do programa.

Canais (Channels)

Canais são os mecanismos fundamentais de comunicação entre goroutines em Go. Um canal define um fluxo de valores de um tipo específico, chamado de **tipo de elemento do canal**. Todos os valores enviados e recebidos por um canal devem respeitar esse tipo.

Além de transportar dados, canais introduzem pontos explícitos de sincronização no programa. Dependendo de sua configuração, operações de envio e recepção podem bloquear a execução até que a outra extremidade esteja pronta, garantindo coordenação segura entre goroutines.

A definição de um canal é feita por meio da função `make`, e seu comportamento varia conforme a presença ou não de um buffer interno — aspecto que será explorado a seguir.

```
ch := make(chan int)
```

Canais podem ser:

- **não bufferizados**, quando não possuem capacidade interna;
- **bufferizados**, quando possuem uma capacidade definida.

Canais não bufferizados

Em canais não bufferizados, as operações de envio e recebimento são **sincronizadas**. A goroutine que envia um valor bloqueia até que outra goroutine esteja pronta para recebê-lo.

```
package main

import "fmt"

func main() {
    ch := make(chan int)
    go func() {
        fmt.Println(<-ch)
    }()

    ch <- 1
}
```

Nesse exemplo, o envio e a recepção ocorrem como um *handshake*: nenhuma das goroutines prossegue sem a outra.

Canais bufferizados

Canais bufferizados possuem uma capacidade interna que permite armazenar valores temporariamente.

```
...
ch := make(chan int, 2)
...
```

Em canais bufferizados:

- o envio bloqueia apenas quando o buffer está cheio;
- a recepção bloqueia apenas quando o buffer está vazio.

```go
package main

import "fmt"

func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2

    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Mesmo sem goroutines explícitas nesse exemplo, o canal bufferizado permite desacoplar envio e recepção até o limite da sua capacidade.

Canais como mecanismo de sincronização

Além de transportar dados, canais são amplamente utilizados como mecanismo de sincronização entre goroutines.

```go
package main

import (
    "fmt"
    "time"
)

func process(base int, c chan int) {
    time.Sleep(time.Second)
    c <- 2 * base

    time.Sleep(time.Second)
    c <- 3 * base

    time.Sleep(3 * time.Second)
    c <- 4 * base
}

func main() {
    c := make(chan int)
    go process(2, c)

    a, b := <-c, <-c
    fmt.Println(a, b)
```

```
25
26      fmt.Println(<-c)
27  }
```

Cada operação de leitura (`<-c`) bloqueia a execução até que um valor seja enviado para o canal. Esse comportamento garante sincronização implícita entre as goroutines, sem a necessidade de mecanismos explícitos de bloqueio.

Compartilhamento seguro e modelo idiomático

O modelo de concorrência de Go incentiva fortemente a comunicação entre goroutines por meio de canais, em vez do compartilhamento direto de memória.

Esse princípio é frequentemente resumido pela máxima:

> **Não comunique compartilhando memória; compartilhe memória comunicando.**

Ao estruturar programas dessa forma, Go reduz a complexidade associada a *locks*, *race conditions* e estados inconsistentes, promovendo código concorrente mais legível, seguro e fácil de manter.

Goroutines e canais formam a base do modelo de concorrência em Go. Goroutines permitem expressar execução concorrente de forma simples e barata, enquanto canais fornecem um meio seguro e explícito de comunicação e sincronização.

Fechamento de Canais (`close`)

Em Go, canais podem ser explicitamente fechados para sinalizar que **não haverá mais envios de valores**. O fechamento de um canal é realizado por meio da função embutida `close`.

```
1  close(ch)
```

Fechar um canal **não libera memória imediatamente**, nem encerra goroutines automaticamente. Seu propósito principal é **comunicar aos receptores que o fluxo de dados foi encerrado**.

Algumas propriedades importantes do fechamento de canais:

- apenas o **remetente** deve fechar o canal;
- enviar valores para um canal fechado causa *panic*;
- receber de um canal fechado é permitido;
- após o fechamento, o canal continua entregando valores restantes no buffer, se houver.

Quando um canal fechado é lido e não há mais valores disponíveis, a operação de recepção retorna o **valor zero do tipo do canal**, além de um indicador booleano.

```go
v, ok := <-ch
```

- `ok == true`: valor recebido com sucesso;
- `ok == false`: canal fechado e sem mais valores.

Exemplo: consumo até o fechamento do canal

```go
package main

import "fmt"

func main() {
    ch := make(chan int)
    go func() {
        for i := 1; i <= 3; i++ {
            ch <- i
        }
        close(ch)
    }()

    for v := range ch {
        fmt.Println(v)
    }
}
```

A construção `for range` sobre um canal é uma forma idiomática de consumir valores **até que o canal seja fechado**. O loop termina automaticamente quando não há mais valores a serem recebidos.

O fechamento de canais é amplamente utilizado em cenários de **produtor–consumidor**, pipelines e fan-out/fan-in.

Direcionalidade de Canais (`chan<-` e `<-chan`)

Por padrão, um canal criado com `make(chan T)` é **bidirecional**, permitindo tanto envio quanto recepção de valores. No entanto, Go permite restringir a direção de uso de um canal por meio de tipos direcionais.

- `chan<- T`: canal apenas para envio;
- `<-chan T`: canal apenas para recepção.

A direcionalidade é uma restrição **em nível de tipo**, aplicada principalmente em parâmetros de funções, com o objetivo de aumentar a segurança e a clareza do código.

Exemplo: separação explícita de responsabilidades

```go
package main

import "fmt"

func producer(out chan<- int) {
    for i := 1; i <= 3; i++ {
        out <- i
    }
    close(out)
}

func consumer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
}

func main() {
    ch := make(chan int)
    go producer(ch)
    consumer(ch)
}
```

Nesse exemplo:

- `producer` só pode enviar valores para o canal;
- `consumer` só pode receber valores do canal;
- o compilador impede usos indevidos, como tentar receber em um canal de envio.

Esse padrão torna o fluxo de dados explícito e reduz a probabilidade de erros lógicos em sistemas concorrentes maiores.

Select: Multiplexação com `select`

Em programas concorrentes reais, frequentemente é necessário **aguardar múltiplas operações de comunicação simultaneamente**. Para isso, Go fornece a instrução `select`.

O `select` permite que uma goroutine espere por múltiplas operações de envio ou recepção em canais, prosseguindo com a primeira que estiver pronta.

```go
select {
    case v := <-ch1:    // operação com ch1
    case v := <-ch2:    // operação com ch2
}
```

O comportamento do `select` é semelhante ao de um **`switch`**, mas voltado exclusivamente para operações com canais.

Exemplo: recebendo de múltiplos canais

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    ch1 := make(chan string)
    ch2 := make(chan string)

    go func() {
        time.Sleep(time.Second)
        ch1 <- "mensagem do canal 1"
    }()

    go func() {
        time.Sleep(2 * time.Second)
        ch2 <- "mensagem do canal 2"
    }()

    select {
        case msg := <-ch1:
            fmt.Println(msg)
        case msg := <-ch2:
            fmt.Println(msg)
    }
}
```

Nesse exemplo, o `select` bloqueia até que uma das operações esteja pronta. A primeira mensagem recebida determina qual ramo será executado.

Caso `default`: operação não bloqueante

O `select` pode incluir um bloco **`default`**, que é executado caso nenhuma das operações esteja pronta.

```go
...
select {
    case v := <-ch:
        fmt.Println(v)
    default:
        fmt.Println("nenhum valor disponível")
}
...
```

Esse padrão é útil para:

- operações não bloqueantes;
- *polling* controlado;
- implementação de *timeouts*.

Timeout com `select` e `time.After`

Um uso comum de `select` é implementar limites de tempo para operações concorrentes.

```
...
select {
    case v := <-ch:
        fmt.Println(v)
    case <-time.After(2 * time.Second):
        fmt.Println("timeout")
}
```

Se nenhum valor for recebido em `ch` dentro do intervalo especificado, o caso de *timeout* será executado.

Considerações finais sobre canais e select

O fechamento de canais, a direcionalidade e o uso de `select` ampliam significativamente o poder expressivo do modelo de concorrência em Go. Esses recursos permitem construir:

- pipelines concorrentes;
- coordenação complexa entre múltiplas goroutines;
- sistemas reativos e resilientes a falhas temporais.

Quando utilizados de forma idiomática, canais eliminam grande parte da necessidade de *locks* explícitos e tornam o fluxo concorrente mais legível, previsível e seguro.

O pacote `sync`: Coordenação Explícita e Controle de Concorrência

Canais são a ferramenta mais conhecida de coordenação em Go, mas não são a única — nem sempre a melhor. Em muitos cenários, especialmente quando há **estado compartilhado**, **controle de acesso** ou **coordenação simples de execução**, o uso explícito das primitivas do pacote `sync` resulta em código mais claro, eficiente e fácil de manter.

Neste tópico, veremos quando e como utilizar `Mutex`, `RWMutex`, `WaitGroup` e `Once`, além de estabelecer critérios práticos para decidir entre canais e mecanismos de sincronização explícita.

`sync.Mutex` e `sync.RWMutex`: sincronização por exclusão mútua

Quando múltiplas goroutines acessam e modificam um mesmo dado em memória, existe o risco de **condições de corrida**. Nesses casos, canais podem ser excessivos ou artificiais, e a exclusão mútua se torna a abordagem mais direta.

O `sync.Mutex` garante que apenas uma goroutine execute uma seção crítica por vez.

```
...
type Counter struct {
    mu sync.Mutex
    n  int
}

func (c *Counter) Inc() {
    c.mu.Lock()
    c.n++
    c.mu.Unlock()
}

func (c *Counter) Value() int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.n
}
...
```

Quando há muitas leituras e poucas escritas, o `sync.RWMutex` pode melhorar o desempenho ao permitir leituras concorrentes:

```
...
type Cache struct {
    mu   sync.RWMutex
    data map[string]string
}

func (c *Cache) Get(key string) string {
    c.mu.RLock()
    defer c.mu.RUnlock()
    return c.data[key]
}

func (c *Cache) Set(key, value string) {
    c.mu.Lock()
    c.data[key] = value
    c.mu.Unlock()
}
...
```

Use `Mutex` quando simplicidade for prioridade. Use `RWMutex` apenas quando o padrão de acesso

justificar.

WaitGroup: coordenação de término de goroutines

Quando o objetivo é **aguardar a finalização de várias goroutines**, o uso de canais apenas para sinalização costuma ser um anti-padrão. O `sync.WaitGroup` resolve esse problema de forma direta e semântica.

```
...
var wg sync.WaitGroup

for i := 0; i < 3; i++ {
    wg.Add(1)   go func(id int) {
        defer wg.Done()
        fmt.Println("worker", id)
    }(i)
}

wg.Wait() fmt.Println("todos os workers finalizaram")
...
```

O `WaitGroup` não transporta dados — ele apenas coordena execução. Isso o torna ideal para controle de ciclo de vida de goroutines.

sync.Once: Inicialização segura e única

Em ambientes concorrentes, inicializações duplicadas podem gerar bugs difíceis de diagnosticar. O `sync.Once` garante que uma função seja executada **exatamente uma vez**, mesmo com múltiplas goroutines concorrentes.

```
...
var once sync.Once
func initConfig() {
    fmt.Println("configuração inicializada")
}

func handler() {
    once.Do(initConfig)
}
```

Esse padrão é amplamente utilizado para inicializar conexões, caches, singletons ou configurações globais, substituindo abordagens frágeis baseadas em flags e mutexes manuais.

Canais versus primitivas do pacote sync

A escolha entre canais e `sync` não é técnica apenas — é **semântica**.

Use canais quando:

- Há troca de dados entre goroutines
- O fluxo de dados define a arquitetura
- O bloqueio faz parte do modelo lógico
- Você quer expressar pipelines ou fan-in / fan-out

Use `sync` quando:

- Há estado compartilhado em memória
- O objetivo é proteger dados, não transferi-los
- Você precisa apenas coordenar execução
- O código fica mais simples sem canais artificiais

Exemplo comparativo:

```
// Canal apenas para sinalização (evitável)
done := make(chan struct{})
go func() {
    work()
    close(done)
}()
<-done
```

Versão mais idiomática:

```
var wg sync.WaitGroup
wg.Add(1)
go func() {
    defer wg.Done()
    work()
}()
wg.Wait()
```

Considerações finais sobre concorrência

Canais são centrais no modelo de concorrência de Go, mas não substituem todas as formas de sincronização. O pacote `sync` existe para resolver problemas específicos de forma direta, eficiente e segura.

Um código Go idiomático não evita `Mutex` nem força canais — ele escolhe conscientemente a ferramenta que melhor expressa a intenção do problema.

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>
```

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 recomendção 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 `buscacep` 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/buscacep
```

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:

```
$ cd $HOME/workshop/buscacep
$ go mod init github.com/[SEU_USARIO_GITHUB]/buscacep

// Saída no console
go: creating new go.mod: module github.com/[SEU_USARIO_GITHUB]/buscacep
```

Neste ponto, o diretório do projeto já deve possuir o aquivo `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**):

```
module github.com/[SEU_USARIO_GITHUB]/buscacep

go 1.13
```

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

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`:

```
$ cd $HOME/workshop/buscacep
# a criação do arquivo pode ser realizada dentro da próprio IDE / Editor de texto
$ touch main.go
```

E digite o código a seguir:

```
// server01.go -> Referência para o arquivo no diretório exemplos
package main

import (
    "log"
    "net/http"
)
```

```go

// Defina uma função manipuladora (handler) chamada "home" que escreve
// um slice de bytes contendo "Bem vindo a API de CEPs" no o corpo da resposta.
func home(w http.ResponseWriter, r *http.Request)  {
    w.Write([]byte("Bem vindo a API de CEPs"))
}

func main() {
    // Use a função http.NewServeMux() para inicializar um novo servermux,
    // depois registre a função "home" como manipulador do padrão de URL "/".
    mux := http.NewServeMux()
    mux.HandleFunc("/", home)

    // Use a função http.ListenAndServe() para iniciar um novo servidor web.
    // Passamos dois parâmetros: o endereço de rede TCP que será escutado
    // (neste caso ":4000") e o servermux que acabamos de criar.
    // Se http.ListenAndServe() retornar um erro, usamos a função
    // log.Fatal() para registrar a mensagem de erro e sair.
    log.Println("Iniciando o servidor na porta: 4000")
    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
```

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

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

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

```
$ 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:

```go
// server02.go -> Referência para o arquivo no diretório exemplos
...
// novo - função manipuladora (hanlder)
func cepHandler(w http.ResponseWriter, r *http.Request) {
    cep := r.URL.Path[len("/cep/"):]
```

```
    w.Write([]byte(cep))
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", home)
    // novo padrão
    mux.HandleFunc("/cep/", cepHandler)

    log.Println("Iniciando o servidor na porta: 4000")
    err := http.ListenAndServe(":4000", mux)
    log.Fatal(err)
}
...
```

> ***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`.

```
// server03.go -> Referência para o arquivo no diretório exemplos
...
type cep struct {
    Cep        string `json:"cep"`
    Cidade     string `json:"cidade"`
    Bairro     string `json:"bairro"`
    Logradouro string `json:"logradouro"`
    UF         string `json:"uf"`
}
...
func cepHandler(w http.ResponseWriter, r *http.Request) {
    rCep := r.URL.Path[len("/cep/"):]
    c := cep{Cep: rCep}
    ret, err := json.Marshal(c)
    if err != nil {
        log.Printf("Ops! ocorreu um erro: %s", err.Error())
```

```
17          http.Error(w, http.StatusText(http.StatusBadRequest), http.
                StatusBadRequest)
18          return
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",
```

```
    "republicavirtual": "https://republicavirtual.com.br/web_cep.php?
        cep=%s&formato=json",
}
...
func cepHandler(w http.ResponseWriter, r *http.Request) {
    rCep := r.URL.Path[len("/cep/"):]

    endpoint := fmt.Sprintf(endpoints["postmon"], rCep)

    client := http.Client{Timeout: time.Duration(time.Millisecond *
        600)}
    resp, err := client.Get(endpoint)
    if err != nil {
        log.Printf("Ops! ocorreu um erro: %s", err.Error())
        http.Error(w, http.StatusText(http.StatusInternalServerError),
            http.StatusInternalServerError)
        return
    }
    defer resp.Body.Close()

    requestContent, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Printf("Ops! ocorreu um erro: %s", err.Error())
        http.Error(w, http.StatusText(http.StatusInternalServerError),
            http.StatusInternalServerError)
        return
    }

    w.Write([]byte(requestContent))
}
...
```

Padronizando nosso retorno

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

```
// http://republicavirtual.com.br/web_cep.php?cep=01412100&formato=json
{
    "resultado": "1",
    "resultado_txt": "sucesso - cep completo",
    "uf": "SP",
    "cidade": "São Paulo",
    "bairro": "Cerqueira César",
    "tipo_logradouro": "Rua",
    "logradouro": "Augusta"
}

// http://api.postmon.com.br/v1/cep/01412100
```

```
{
    "complemento": "de 2440 ao fim - lado par",
    "bairro": "Cerqueira César",
    "cidade": "São Paulo",
    "logradouro": "Rua Augusta",
    "estado_info": {
        "area_km2": "248.221,996",
        "codigo_ibge": "35",
        "nome": "São Paulo"
    },
    "cep": "01412100",
    "cidade_info": {
        "area_km2": "1521,11",
        "codigo_ibge": "3550308"
    },
    "estado": "SP"
}

// https://viacep.com.br/ws/01412100/json/
{
  "cep": "01412-100",
  "logradouro": "Rua Augusta",
  "complemento": "de 2440 ao fim - lado par",
  "bairro": "Cerqueira César",
  "localidade": "São Paulo",
  "uf": "SP",
  "unidade": "",
  "ibge": "3550308",
  "gia": "1004"
}
```

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

```
// server06.go -> Referência para o arquivo no diretório exemplos
...
func cepHandler(w http.ResponseWriter, r *http.Request) {
    rCep := r.URL.Path[len("/cep/"):]

    endpoint := fmt.Sprintf(endpoints["republicavirtual"], rCep)

    client := http.Client{Timeout: time.Duration(time.Millisecond *
       600)}
    resp, err := client.Get(endpoint)
    if err != nil {
        log.Printf("Ops! ocorreu um erro: %s", err.Error())
        http.Error(w, http.StatusText(http.StatusInternalServerError),
           http.StatusInternalServerError)
        return
    }
    defer resp.Body.Close()

```

```
17      requestContent, err := ioutil.ReadAll(resp.Body)
18      if err != nil {
19          log.Printf("Ops! ocorreu um erro: %s", err.Error())
20          http.Error(w, http.StatusText(http.StatusInternalServerError),
                http.StatusInternalServerError)
21          return
22      }
23
24      // Novo
25      c, err := parseResponse(requestContent)
26      if err != nil {
27          log.Printf("Ops! ocorreu um erro: %s", err.Error())
28          http.Error(w, http.StatusText(http.StatusInternalServerError),
                http.StatusInternalServerError)
29          return
30      }
31
32      c.Cep = rCep
33      ret, err := json.Marshal(c)
34      if err != nil {
35          log.Printf("Ops! ocorreu um erro: %s", err.Error())
36          http.Error(w, http.StatusText(http.StatusInternalServerError),
                http.StatusInternalServerError)
37          return
38      }
39
40      w.Write([]byte(ret))
41  }
42
43  func parseResponse(content []byte) (payload cep, err error) {
44      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) + " "
                + payload.Logradouro
69      }
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 reposta

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 definí-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
```

```
...
func cepHandler(w http.ResponseWriter, r *http.Request) {
    ...
    // Acertando o cabeçalho
    w.Header().Set("Content-Type", "application/json")
    w.Write([]byte(ret))
}
...
```

Gran finale

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

```
// server08.go -> Referência para o arquivo no diretório exemplos
package main

import (
    ...
    "regexp" // Novo
    "time"
)

type cep struct {
    ...
}

// novo
func (c cep) exist() bool {
    return len(c.UF) != 0
}
...

// Função cepHandler foi refatorada e dela extraímos a função request
func cepHandler(w http.ResponseWriter, r *http.Request) {
    // Restrigindo o acesso apenas pelo método GET
    if r.Method != http.MethodGet {
        http.Error(w, http.StatusText(http.StatusMethodNotAllowed),
            http.StatusMethodNotAllowed)
        return
    }

    rCep := r.URL.Path[len("/cep/"):]
    rCep, err := sanitizeCEP(rCep)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    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.
            StatusNoContent)
56  }
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, Dura
                ção: %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
                minimo 8 digitos")
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
- Using Go Modules
- Migrating to Go Modules
- Publishing Go Modules
- Go Modules: v2 and Beyond