ThoughtWorks®

uma pegadinha em #golang

3 MODELOS DE VARIÁVEIS

Valores, ponteiros e referências na linguagem Go.

THE DEVELOPER'S CONFERENCE

ThoughtWorks®

MODELOS DE VARIÁVEIS EM LINGUAGENS

O que a gente vê por aí

MODELOS DE VARIÁVEIS EM ALGUMAS LINGUAGENS

	valores	ponteiros	referências
C	✔	✔	
C++	✔	✔	✔
Java	✔		✔
JavaScript			✔
Python			✔
Go	✔	✔	✔

ThoughtWorks®

VARIÁVEIS EM GO

Comportamentos diferentes

EM GO, VARIÁVEIS SÃO “CAIXAS”

```go
i := 3
i2 := i
i2++
fmt.Printf("i\t%#v\ni2\t%#v\n", i, i2)
```

```
i    3
i2   4
```

EM GO, VARIÁVEIS SÃO “CAIXAS”

```
i := 3
i2 := i
i2++
fmt.Printf("i\t%#v\ni2\t%#v\n", i, i2)
```

```
i   3
i2  4
```

```
a := [...]int{1, 2, 3}
a2 := a
a2[0]++
fmt.Printf("a\t%#v\na2\t%#v\n", a, a2)
```

```
a   [3]int{1, 2, 3}
a2  [3]int{2, 2, 3}
```

EM GO, VARIÁVEIS SÃO “CAIXAS”

```go
i := 3
i2 := i
i2++
fmt.Printf("i\t%#v\ni2\t%#v\n", i, i2)
```

```
i    3
i2   4
```

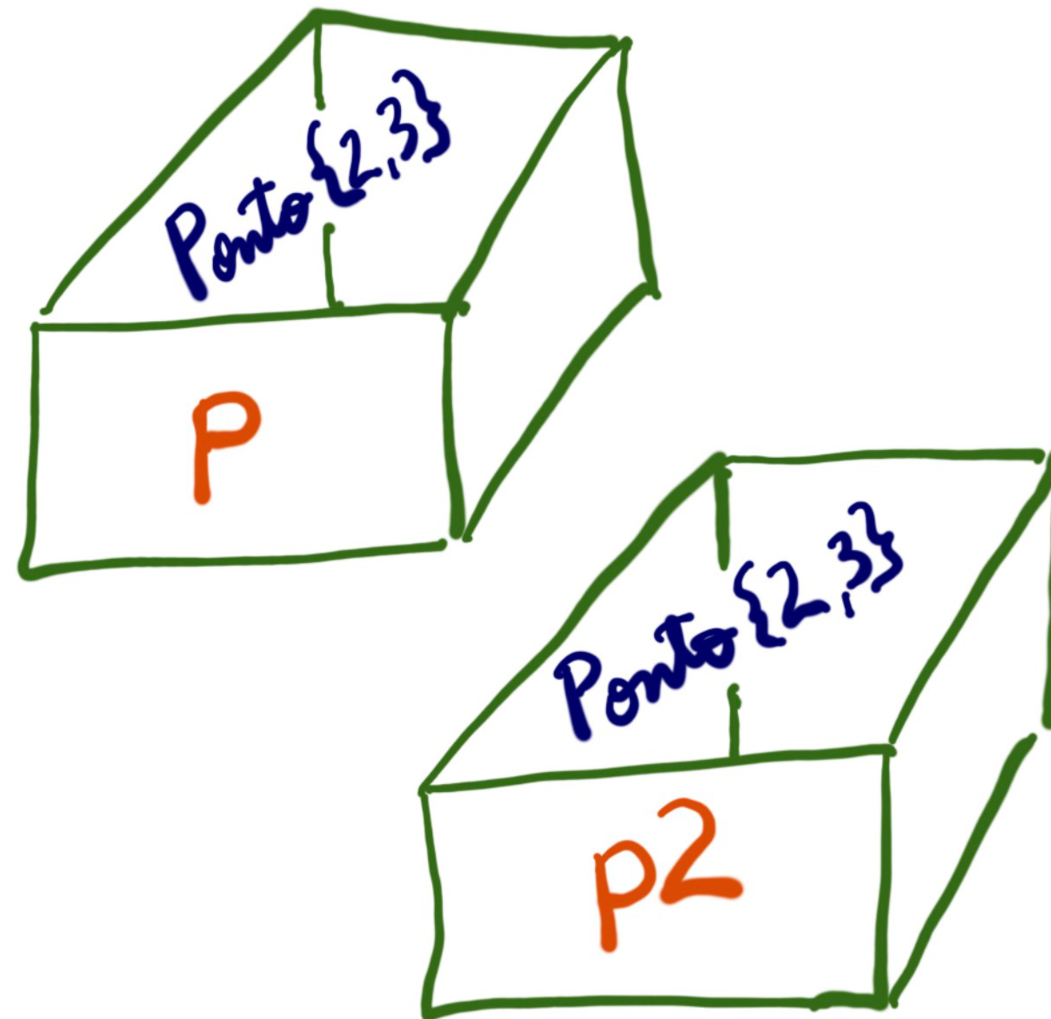
```go
a := [...]int{1, 2, 3}
a2 := a
a2[0]++
fmt.Printf("a\t%#v\na2\t%#v\n", a, a2)
```

```
a    [3]int{1, 2, 3}
a2   [3]int{2, 2, 3}
```

```go
p := Ponto{2, 3}
p2 := p
p2.y++
fmt.Printf("p\t%#v\np2\t%#v\n", p, p2)
```

```
p    main.Ponto{x:2, y:3}
p2   main.Ponto{x:2, y:4}
```

OS CONTEÚDOS DAS CAIXAS P E P2 SÃO INDEPENDENTES




```
p := Ponto{2, 3}
p2 := p
p2.y++
fmt.Printf("p\t%#v\np2\t%#v\n", p, p2)
```

```
p   main.Ponto{x:2, y:3}
p2  main.Ponto{x:2, y:4}
```

O OPERADOR & (ENDEREÇO) DEVOLVE UM PONTEIRO

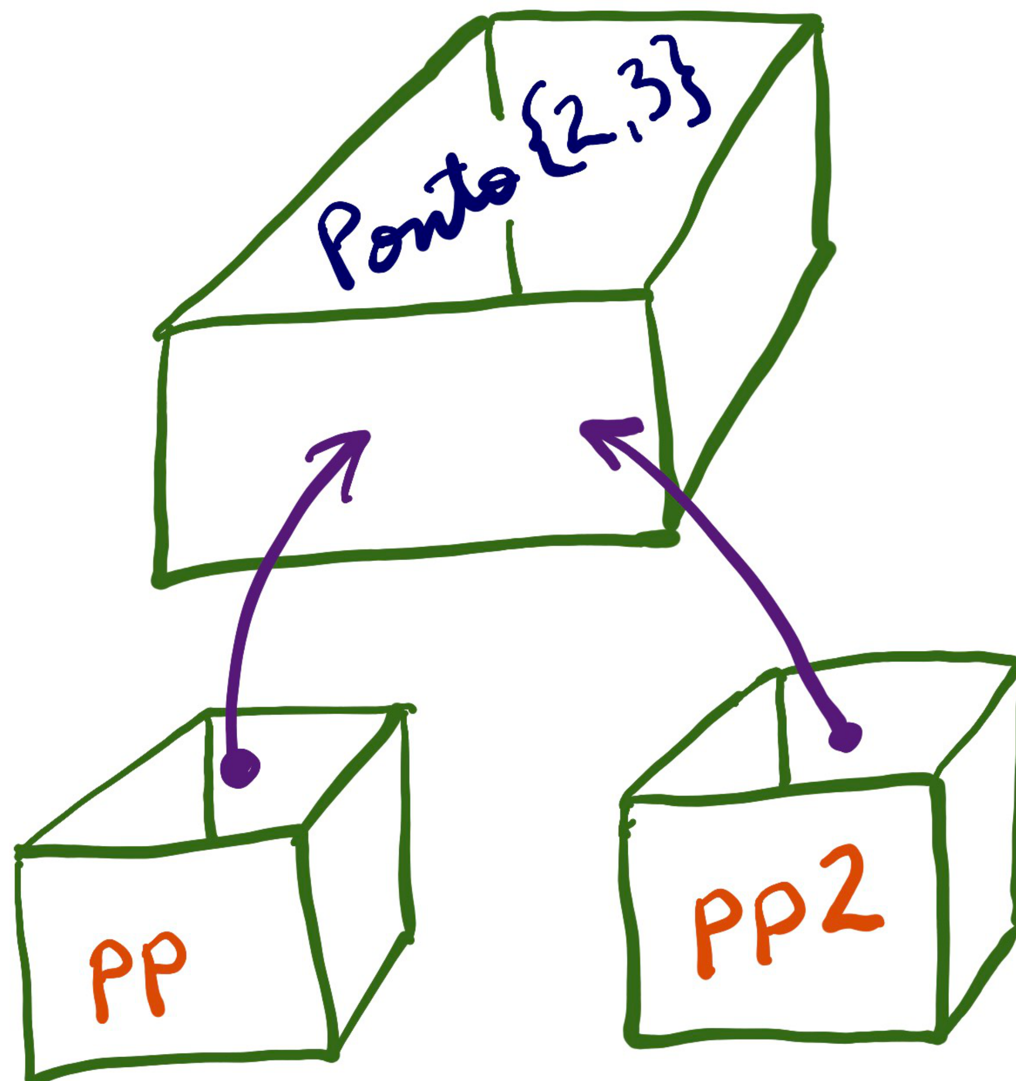
```go
pp := &Ponto{2, 3}
pp2 := pp
pp2.y++
fmt.Printf("pp\t%#v\npp2\t%#v\n\n", pp, pp2)
```

```
pp  &main.Ponto{x:2, y:4}
pp2 &main.Ponto{x:2, y:4}
```

CAIXAS PP E PP2 TÊM PONTEIROS PARA A MESMA CAIXA

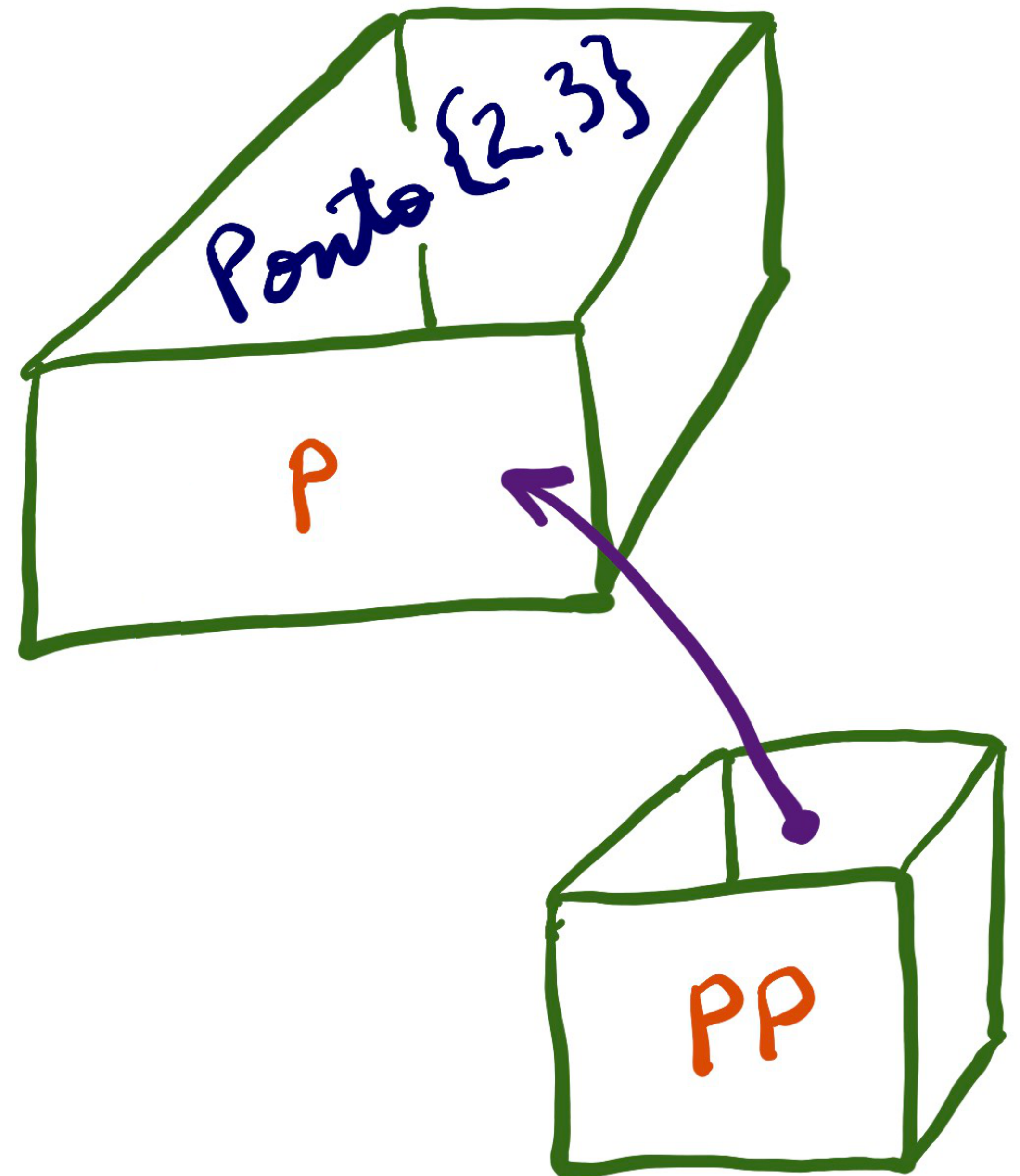
```
pp := &Ponto{2, 3}
pp2 := pp
pp2.y++
fmt.Printf("pp\t%#v\npp2\t%#v\n\n", pp, pp2)
```

```
pp  &main.Ponto{x:2, y:4}
pp2 &main.Ponto{x:2, y:4}
```



SINTAXE DE PONTEIROS: &X, PX, *PX

```go
type Ponto struct {
	x, y float64
}

func main() {
	p := Ponto{2, 3}
	fmt.Printf("p\t%#v\n\n", p)

	var pp *Ponto

	pp = new(Ponto)
	fmt.Printf("pp\t%#v\n", pp)
	fmt.Printf("\t(%p)\n\n", pp)

	pp = &p
	fmt.Printf("pp\t%#v\n", pp)
	fmt.Printf("\t(%p)\n\n", pp)


	fmt.Printf("*pp\t%#v\n\n", *pp)
}
```



FORMATO %P MOSTRA O PONTEIRO EM SI, NÃO SEU ALVO

```go
type Ponto struct {
    x, y float64
}

func main() {
    p := Ponto{2, 3}
    fmt.Printf("p\t%#v\n\n", p)

    var pp *Ponto

    pp = new(Ponto)
    fmt.Printf("pp\t%#v\n", pp)
    fmt.Printf("\t(%p)\n\n", pp)

    pp = &p
    fmt.Printf("pp\t%#v\n", pp)
    fmt.Printf("\t(%p)\n\n", pp)


    fmt.Printf("*pp\t%#v\n\n", *pp)
}
```

```
p    main.Ponto{x:2, y:3}



pp   &main.Ponto{x:0, y:0}
     (0xc0000140c0)


pp   &main.Ponto{x:2, y:3}
     (0xc000014080)


*pp  main.Ponto{x:2, y:3}
```

EU LEIO *P ASSIM: "A COISA APONTADA POR P" (O ALVO)

```go
type Ponto struct {
    x, y float64
}

func main() {
    p := Ponto{2, 3}
    fmt.Printf("p\t%#v\n\n", p)

    var pp *Ponto

    pp = new(Ponto)
    fmt.Printf("pp\t%#v\n", pp)
    fmt.Printf("\t(%p)\n\n", pp)

    pp = &p
    fmt.Printf("pp\t%#v\n", pp)
    fmt.Printf("\t(%p)\n\n", pp)


    fmt.Printf("*pp\t%#v\n\n", *pp)
}
```

```
p   main.Ponto{x:2, y:3}



pp  &main.Ponto{x:0, y:0}
    (0xc0000140c0)


pp  &main.Ponto{x:2, y:3}
    (0xc000014080)


*pp main.Ponto{x:2, y:3}
```

PONTEIROS EM GO

Ao contrário de C, C++, e Pascal, Go tem ponteiros mas também tem um GC (garbage colector).

A pessoa que programa em Go não precisa manualmente alocar e liberar memória.

O compilador gera código de apoio que supervisiona o uso de ponteiros para saber quais estruturas de dados podem ser descartadas.

O valor de um ponteiro não é fixo: o alvo pode ser realocado e o valor do ponteiro será atualizado automaticamente.

MAS O QUE HÁ NESSAS “CAIXAS”?

```
s := []int{1, 2, 3}
s2 := s
s2[0]++
fmt.Printf("s\t%#v\ns2\t%#v\n\n", s, s2)
```

```
s   []int{2, 2, 3}
s2  []int{2, 2, 3}
```

```
m := map[byte]int{1: 1, 2: 2, 3: 3}
m2 := m
m2[3]++
fmt.Printf("m\t%#v\nm2\t%#v\n\n", m, m2)
```

```
m   map[uint8]int{0x2:2, 0x3:4, 0x1:1}
m2  map[uint8]int{0x1:1, 0x2:2, 0x3:4}
```

ESSES SÃO EXEMPLOS DE "ALIASING"

Aliasing é literalmente "apelidamento":
ocorre quando uma coisa tem vários nomes ou apelidos.

```
pp := &Ponto{2, 3}
pp2 := pp
pp2.y++
```

```
pp  &main.Ponto{x:2, y:4}
pp2 &main.Ponto{x:2, y:4}
```

```
s := []int{1, 2, 3}
s2 := s
s2[0]++
```

```
s   []int{2, 2, 3}
s2  []int{2, 2, 3}
```

```
m := map[byte]int{1: 1, 2: 2, 3: 3}
m2 := m
m2[3]++
```

```
m   map[uint8]int{0x2:2, 0x3:4, 0x1:1}
m2  map[uint8]int{0x1:1, 0x2:2, 0x3:4}
```

SEMÂNTICA DE VALORES

Semântica de valores:

- Variáveis são áreas de memória que contém os bits representando os dados em si.
- Não ocorre aliasing.
- Atribuição faz cópia dos dados.
- Parâmetros recebidos por funções são cópias dos argumentos passados.
  A função pode alterar sua cópia, mas não tem como alterar os dados do cliente (quem a invocou).

SEMÂNTICA DE VALORES X SEMÂNTICA DE REFERÊNCIAS

Semântica de valores:

- Variáveis são áreas de memória que contém os bits representando os dados em si.
- Não ocorre aliasing.
- Atribuição faz cópia dos dados.
- Parâmetros recebidos por funções são cópias dos argumentos passados.
  A função pode alterar sua cópia, mas não tem como alterar os dados do cliente (quem a invocou).

Semântica de referências:

- Variáveis contém apenas referências ou ponteiros que apontam para os dados alocados em outra parte da memória.
- Pode ocorrer aliasing: mais de uma referência/ponteiro indicando o mesmo dado.
- Atribuição faz cópia da referência ou ponteiro; os dados são compartilhados.
- Parâmetros recebidos por funções são referências/ponteiros para os dados do cliente, que podem ser alterados pela função.

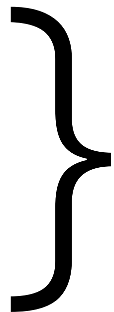
AS PEGADINHAS

Referências em Go são implícitas: são embutidas em structs que você só pode inspecionar usando o pacote **unsafe** ☠️.

Ponteiros têm sintaxe explícita (&x, *p) mas valores com referências **não têm sintaxe explícita**.

Somente 3 tipos nativos mutáveis usam referências:

- slice
- map
- channel



As únicas estruturas de dados construídas com **make()**

Magic!

Você não pode criar seus próprios tipos com referências.

EXEMPLOS SIMPLES

O que acontece na prática

TRIPLICADOR DE VALORES (SUPER ÚTIL ;-)

```go
package main

import "fmt"

func triInt(x int) int {
    x *= 3
    return x
}

func triIntUpdate(x *int) int {
    *x *= 3
    return *x
}

func triArray(x [5]int) [5]int {
    for i := range(x) {
        x[i] *= 3
    }
    return x
}
```

```go
func triSliceUpdate(x []int) []int {
    for i := range(x) {
        x[i] *= 3
    }
    return x
}

func triArrayUpdate(x *[5]int) [5]int {
    for i := range(x) {
        x[i] *= 3
    }
    return *x
}


func triIntVariadic(x ...int) []int {
    for i := range(x) {
        x[i] *= 3
    }
    return x
}
```

Código-fonte deste exemplo: **tgo.li/2UtD7Xe**

TRIPLICADOR DE VALORES (SUPER ÚTIL ;-)

```go
func main() {
    x1 := 2
    fmt.Printf("triInt\t\t%v\t", x1)
    fmt.Printf("%v\t%v\n", triInt(x1), x1)
    x2 := [...]int{10, 20, 30, 40, 50}
    fmt.Printf("triArray\t%v\t", x2)
    fmt.Printf("%v\t%v\n", triArray(x2), x2)
    x3 := []int{10, 20, 30, 40, 50}
    fmt.Printf("triSliceUpdate\t%v\t", x3)
    fmt.Printf("%v\t%v\n", triSliceUpdate(x3), x3)
    x4 := 4
    x4ptr := &x4
    fmt.Printf("triIntUpdate\t%v\t", x4)
    fmt.Printf("%v\t%v\n", triIntUpdate(x4ptr), x4)
    x5 := [...]int{10, 20, 30, 40, 50}
    x5ptr := &x5
    fmt.Printf("triArrayUpdate\t%v\t", x5)
    fmt.Printf("%v\t%v\n", triArrayUpdate(x5ptr), x5)
    x6, x7, x8 := 100, 200, 300
    fmt.Printf("triIntVariadic\t%v, %v, %v\t", x6, x7, x8)
    fmt.Printf("%v\t%v, %v, %v\n", triIntVariadic(x6, x7, x8), x6, x7, x8)
    x9 := []int{10, 20, 30, 40, 50}
    fmt.Printf("triIntVariadic\t%v\t", x9)
    fmt.Printf("%v\t%v\n", triIntVariadic(x9...), x9)
}
```

tgo.li/2UtD7Xe

TRIPLICADOR DE VALORES (SUPER ÚTIL ;-)

```go
package main

import "fmt"

func triInt(x int) int {
	x *= 3
	return x
}

func triIntUpdate(x *int) int {
	*x *= 3
	return *x
}

func triArray(x [5]int) [5]int {
	for i := range(x) {
		x[i] *= 3
	}
	return x
}
```

```go
func triSliceUpdate(x []int) []int {
	for i := range(x) {
		x[i] *= 3
	}
	return x
}

func triArrayUpdate(x *[5]int) [5]int {
	for i := range(x) {
		x[i] *= 3
	}
	return *x
}


func triIntVariadic(x ...int) []int {
	for i := range(x) {
		x[i] *= 3
	}
	return x
}
```

```
triInt            2                  6                      2
triArray          [10 20 30 40 50]   [30 60 90 120 150]     [10 20 30 40 50]
triSliceUpdate    [10 20 30 40 50]   [30 60 90 120 150]     [30 60 90 120 150]
triIntUpdate      4                  12                     12
triArrayUpdate    [10 20 30 40 50]   [30 60 90 120 150]     [30 60 90 120 150]
triIntVariadic    100, 200, 300      [300 600 900]          100, 200, 300
triIntVariadic    [10 20 30 40 50]   [30 60 90 120 150]     [30 60 90 120 150]
```

ThoughtWorks®

ANATOMIA DE SLICES

Examinando um tipo de referência por dentro.

ANALISADOR DE SLICE

Inspirado em post de Bill Kennedy: “Understanding slices”

tgo.li/2QjwoR3

Código-fonte deste exemplo:

tgo.li/2L7xNEQ

```go
package main

import (
    "fmt"
    "unsafe"
)

func InspectSlice(intSlice []int) {

    fmt.Println("intSlice:")
    fmt.Printf("\t%#v\n\n", intSlice)

    // Get slicePtr of slice structure
    slicePtr := unsafe.Pointer(&intSlice)
    ptrSize := unsafe.Sizeof(slicePtr)

    // Compute addresses of len and cap
    lenAddr := uintptr(slicePtr) + ptrSize
    capAddr := uintptr(slicePtr) + (ptrSize * 2)

    // Create pointers to len and cap
    lenPtr := (*int)(unsafe.Pointer(lenAddr))
    capPtr := (*int)(unsafe.Pointer(capAddr))

    // Get pointer to underlying array
    // How to do this without hardcoding the array size?
    arrayPtr := (*[100]int)(unsafe.Pointer(*(*uintptr)(slicePtr)))

    fmt.Println("intSlice:")

    // Not using %T on next line to show expected data array size
    // fmt.Printf("\t@%p: data %T = %p\n", slicePtr, arrayPtr, arrayPtr)
    fmt.Printf("\t@%p: data *[%d]int = %p\n", slicePtr, *capPtr, arrayPtr)

    fmt.Printf("\t@%p: len %T = %d\n", lenPtr, *lenPtr, *lenPtr)

    fmt.Printf("\t@%p: cap %T = %d\n", capPtr, *capPtr, *capPtr)

    fmt.Println("data:")

    for index := 0; index < *capPtr; index++ {
        fmt.Printf("\t@%p: [%d] %T = %d\n",
            &(*arrayPtr)[index], index, (*arrayPtr)[index], (*arrayPtr)[index])
    }

}

func main() {
    intSlice := make([]int, 3, 5)
    intSlice[0] = 11
    intSlice[1] = 12
    intSlice[2] = 13

    InspectSlice(intSlice)

    for _, n := range []int{140, 150, 160} {
        intSlice = append(intSlice, n)
        InspectSlice(intSlice)
    }

}
```

```
intSlice:
	[]int{11, 12, 13}

intSlice:
	@0xc00000a060: data *[5]int = 0xc000072030
	@0xc00000a068: len int = 3
	@0xc00000a070: cap int = 5
data:
	@0xc000072030: [0] int = 11
	@0xc000072038: [1] int = 12
	@0xc000072040: [2] int = 13
	@0xc000072048: [3] int = 0
	@0xc000072050: [4] int = 0

intSlice:
	[]int{11, 12, 13, 140}

intSlice:
	@0xc00000a0a0: data *[5]int = 0xc000072030
	@0xc00000a0a8: len int = 4
	@0xc00000a0b0: cap int = 5
data:
	@0xc000072030: [0] int = 11
	@0xc000072038: [1] int = 12
	@0xc000072040: [2] int = 13
	@0xc000072048: [3] int = 140
	@0xc000072050: [4] int = 0

intSlice:
	[]int{11, 12, 13, 140, 150}

intSlice:
	@0xc00000a0e0: data *[5]int = 0xc000072030
	@0xc00000a0e8: len int = 5
	@0xc00000a0f0: cap int = 5
data:
	@0xc000072030: [0] int = 11
	@0xc000072038: [1] int = 12
	@0xc000072040: [2] int = 13
	@0xc000072048: [3] int = 140
	@0xc000072050: [4] int = 150

intSlice:
	[]int{11, 12, 13, 140, 150, 160}

intSlice:
	@0xc00000a120: data *[10]int = 0xc0000180f0
	@0xc00000a128: len int = 6
	@0xc00000a130: cap int = 10
data:
	@0xc0000180f0: [0] int = 11
	@0xc0000180f8: [1] int = 12
	@0xc000018100: [2] int = 13
	@0xc000018108: [3] int = 140
	@0xc000018110: [4] int = 150
	@0xc000018118: [5] int = 160
	@0xc000018120: [6] int = 0
	@0xc000018128: [7] int = 0
	@0xc000018130: [8] int = 0
	@0xc000018138: [9] int = 0

intSlice:
	[]int{11, 12, 13, 140, 150, 160, 170}

intSlice:
	@0xc00000a160: data *[10]int = 0xc0000180f0
	@0xc00000a168: len int = 7
	@0xc00000a170: cap int = 10
data:
	@0xc0000180f0: [0] int = 11
	@0xc0000180f8: [1] int = 12
	@0xc000018100: [2] int = 13
	@0xc000018108: [3] int = 140
	@0xc000018110: [4] int = 150
	@0xc000018118: [5] int = 160
	@0xc000018120: [6] int = 170
	@0xc000018128: [7] int = 0
	@0xc000018130: [8] int = 0
	@0xc000018138: [9] int = 0
```

ANALISADOR DE SLICE: MAIN

```
func main() {
    intSlice := make([]int, 3, 5)
    intSlice[0] = 11
    intSlice[1] = 12
    intSlice[2] = 13

    InspectSlice(intSlice)

    for _, n := range []int{140, 150, 160} {
        intSlice = append(intSlice, n)
        InspectSlice(intSlice)
    }

}
```

Código-fonte
deste exemplo:

tgo.li/2L7xNEQ

ANALISADOR DE SLICE: INSPECT SLICE

```go
func InspectSlice(intSlice []int) {

    fmt.Println("intSlice:")
    fmt.Printf("\t%#v\n\n", intSlice)

    // Get slicePtr of slice structure
    slicePtr := unsafe.Pointer(&intSlice)
    ptrSize := unsafe.Sizeof(slicePtr)

    // Compute addresses of len and cap
    lenAddr := uintptr(slicePtr) + ptrSize
    capAddr := uintptr(slicePtr) + (ptrSize * 2)

    // Create pointers to len and cap
    lenPtr := (*int)(unsafe.Pointer(lenAddr))
    capPtr := (*int)(unsafe.Pointer(capAddr))

    // Get pointer to underlying array
    arrayPtr := (*[100]int)(unsafe.Pointer(*(*uintptr)(slicePtr)))

    fmt.Println("intSlice:")

    fmt.Printf("\t@%p: data *[%d]int = %p\n", slicePtr, *capPtr, arrayPtr)
    fmt.Printf("\t@%p: len %T = %d\n", lenPtr, *lenPtr, *lenPtr)
    fmt.Printf("\t@%p: cap %T = %d\n", capPtr, *capPtr, *capPtr)

    fmt.Println("data:")
    for index := 0; index < *capPtr; index++ {
        fmt.Printf("\t@%p: [%d] %T = %d\n",
            &(*arrayPtr)[index], index, (*arrayPtr)[index], (*arrayPtr)[index])
    }
}
```

Código-fonte deste exemplo:

tgo.li/2L7xNEQ

ANALISADOR DE SLICE: MAIN

```go
func main() {
    intSlice := make([]int, 3, 5)
    intSlice[0] = 11
    intSlice[1] = 12
    intSlice[2] = 13

    InspectSlice(intSlice)

    for _, n := range []int{140, 150, 160} {
        intSlice = append(intSlice, n)
        InspectSlice(intSlice)
    }

}
```

Slice é um struct com três campos: **data, len, cap**

Array subjacente (*underlying array*)

```
intSlice:
  []int{11, 12, 13}

intSlice:
  @0xc00000a060: data *[5]int = 0xc000072030
  @0xc00000a068: len int = 3
  @0xc00000a070: cap int = 5
data:
  @0xc000072030: [0] int = 11
  @0xc000072038: [1] int = 12
  @0xc000072040: [2] int = 13
  @0xc000072048: [3] int = 0
  @0xc000072050: [4] int = 0
```

tgo.li/2L7xNEQ

ANALISADOR DE SLICE: MAIN

```go
func main() {
    intSlice := make([]int, 3, 5)
    intSlice[0] = 11
    intSlice[1] = 12
    intSlice[2] = 13

    InspectSlice(intSlice)

    for _, n := range []int{140, 150, 160} {
        intSlice = append(intSlice, n)
        InspectSlice(intSlice)
    }

}
```

```
intSlice:
  []int{11, 12, 13, 140}

intSlice:
  @0xc00000a0a0: data *[5]int = 0xc000072030
  @0xc00000a0a8: len int = 4
  @0xc00000a0b0: cap int = 5
data:
  @0xc000072030: [0] int = 11
  @0xc000072038: [1] int = 12
  @0xc000072040: [2] int = 13
  @0xc000072048: [3] int = 140
  @0xc000072050: [4] int = 0
```

tgo.li/2L7xNEQ

ANALISADOR DE SLICE: MAIN

```go
func main() {
    intSlice := make([]int, 3, 5)
    intSlice[0] = 11
    intSlice[1] = 12
    intSlice[2] = 13

    InspectSlice(intSlice)

    for _, n := range []int{140, 1
        intSlice = append(intSlice
        InspectSlice(intSlice)
    }

}
```

```
intSlice:
  []int{11, 12, 13}

intSlice:
  @0xc00000a060: data *[5]int = 0xc000072030
  @0xc00000a068: len int = 3
  @0xc00000a070: cap int = 5
data:
  @0xc000072030: [0] int = 11
  @0xc000072038: [1] int = 12
  @0xc000072040: [2] int = 13
  @0xc000072048: [3] int = 0
  @0xc000072050: [4] int = 0

intSlice:
  []int{11, 12, 13, 140}

intSlice:
  @0xc00000a0a0: data *[5]int = 0xc000072030
  @0xc00000a0a8: len int = 4
  @0xc00000a0b0: cap int = 5
data:
  @0xc000072030: [0] int = 11
  @0xc000072038: [1] int = 12
  @0xc000072040: [2] int = 13
  @0xc000072048: [3] int = 140
  @0xc000072050: [4] int = 0
```

tgo.li/2L7xNEQ

ANALISADOR DE SLICE

Ao fazer **append**, quando a capacidade inicial é ultrapassada, um novo array subjacente é criado com o dobro da capacidade, e o conteúdo anterior é copiado para lá.

Código-fonte deste exemplo: **tgo.li/2L7xNEQ**

```
intSlice:
  []int{11, 12, 13, 140, 150}

intSlice:
  @0xc00000a0e0: data *[5]int = 0xc000072030
  @0xc00000a0e8: len int = 5
  @0xc00000a0f0: cap int = 5
data:
  @0xc000072030: [0] int = 11
  @0xc000072038: [1] int = 12
  @0xc000072040: [2] int = 13
  @0xc000072048: [3] int = 140
  @0xc000072050: [4] int = 150

intSlice:
  []int{11, 12, 13, 140, 150, 160}

intSlice:
  @0xc00000a120: data *[10]int = 0xc0000180f0
  @0xc00000a128: len int = 6
  @0xc00000a130: cap int = 10
data:
  @0xc0000180f0: [0] int = 11
  @0xc0000180f8: [1] int = 12
  @0xc000018100: [2] int = 13
  @0xc000018108: [3] int = 140
  @0xc000018110: [4] int = 150
  @0xc000018118: [5] int = 160
  @0xc000018120: [6] int = 0
  @0xc000018128: [7] int = 0
  @0xc000018130: [8] int = 0
  @0xc000018138: [9] int = 0
```

ThoughtWorks®

CONCLUSÃO

UMA FORMA DE ENTENDER

Go adota a semântica de valores em todos os casos, mas em alguns casos o valor é um ponteiro ou um uma estrutura que têm referências (ponteiros ocultos).

TAMANHOS EM BYTES E VALORES ZERO

Tipo	unsafe.Sizeof()	Valor "zero"
string	16+	""
int	8	0
float32	4	0
[3]float32	12	[3]float32{0, 0, 0}
*[3]float32	8	(*[3]float32)(nil)
[]float32	24+	[]float32(nil)
map[string]int	8+	map[string]int(nil)
chan uint8	8+	(chan uint8)(nil)

nil é o valor zero dos tipos que têm ponteiros

- Essa tabela é verdadeira para uma CPU de 64 bit típica, com ponteiros de 8 bytes.
- Os tamanhos com + não incluem os dados referenciados na string, slice, map e channel.

DICAS FINAIS

É praticamente impossível programar em Go sem usar slices, mas as slices são o tipo de referência mais traiçoeiro da linguagem. Entenda a fundo como elas funcionam. Saiba que o array subjacente pode ser compartilhado e pode mudar a qualquer momento.

Cuidado ao passar ou receber qualquer tipo de referência mutável como argumento (slice, map, channel): a função pode mudar a estrutura de dados sem você saber. Se você é a autora da função, considere usar um nome que deixe isso explícito, por exemplo: **`reverseInPlace`**, **`rankUpdate`**.

REFERÊNCIAS

GOPL: The Go Programming Language — Donovan & Kernighan (A Linguagem de Programação Go, Ed. Novatec)

SliceTricks — Go Wiki — https://tgo.li/2L6Ebfo

Variable models in Go — LR — https://tgo.li/2DrTVbh

MUITO GRATO

Dúvidas ou sugestões?

Twitter: @ramalhoorg

E-mail: luciano.ramalho@thoughtworks.com

ThoughtWorks®