

Concorrência em Go

André Murbach Maidl

Escola Politécnica

Pontifícia Universidade Católica do Paraná

Antes de tudo

- Os slides sobre a linguagem Go funcionam apenas como guias para as aulas.
- Não esqueça de consultar a documentação da linguagem e das APIs das suas bibliotecas:
<http://golang.org/pkg/>.
- Se tiver alguma dúvida entre em contato com o professor.
- Lembre-se também que tanto o *Google* quanto o *DuckDuckGo* são seus amigos! 😊

Objetivos da aula

- Entender as primitivas de concorrência da linguagem Go:
 - *goroutines*;
 - canais;
 - `sync.Mutex`.

Conseguimos viver sem concorrência?

- O uso de programação concorrente tem sido cada vez mais necessário na computação.
- Por exemplo, para implementar servidores Web que atendem milhares de requisições ao mesmo tempo.
- As aplicações móveis são outro exemplo, pois renderizam animações na interface do usuário enquanto executam outras computações e requisições de rede simultaneamente.
- E até mesmo aplicações *batch* (lê uma entrada, processa e gera uma saída) usam concorrência para esconder a latência das operações de E/S e para explorar os múltiplos *cores* dos processadores.

Concorrência em Go

- Go disponibiliza dois estilos de programação concorrente:
 1. Goroutines e canais;
 2. Memória compartilhada.

Goroutines e canais

- São primitivas da linguagem que nos permitem estruturar um programa seguindo o modelo de concorrência CSP (*Communicating Sequential Processes*), proposto por Sir Tony Hoare em 1978.
- A ideia é usar canais para passar referências de dados entre *goroutines*, em vez de usar *locks* para mediar o acesso à memória compartilhada.
- Essa abordagem garante que apenas uma *goroutine* tem acesso ao dado em um determinado momento.

Memória compartilhada

- É o modelo tradicional de implementação de concorrência através de memória compartilhada.
- Ou seja, é similar ao uso de *threads* em outras linguagens de programação.

“Do not communicate by sharing memory; instead, share memory by communicating.”

É o lema de Go para tentar tornar a programação concorrente mais acessível aos programadores.

Goroutines

- Uma *goroutine* é uma função que executa concorrentemente com outras funções.
- Para criar uma *goroutine* basta usar a palavra chave **go** em uma chamada de função.

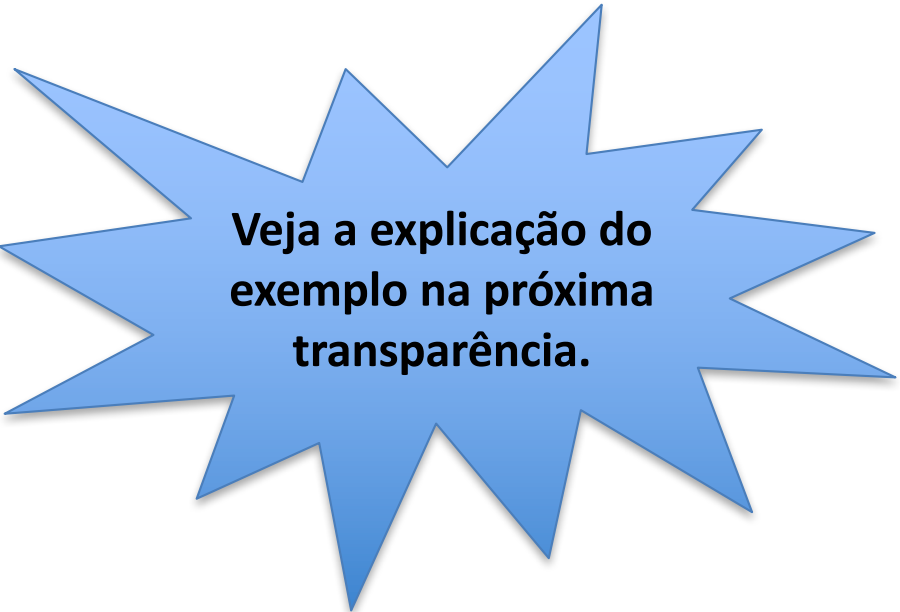
```
package main

import "fmt"

var MAX int = 5

func f (n int) {
    for i := 0; i < MAX; i++ {
        fmt.Println(n, ":", i)
    }
}

func main () {
    go f(0)
    var input string
    fmt.Scanln(&input)
}
```



Veja a explicação do exemplo na próxima transparência.

Explicação do exemplo anterior

- O programa da transparência anterior executa duas *goroutines*: uma é a função `main` e a outra é a função `f`.
- Quando fazemos uma chamada de função, o programa segue um fluxo sequencial de instruções, executando uma em seguida da outra.
- No entanto, quando criamos uma *goroutine*, a linha seguinte é executada imediatamente, sem esperar pelo fim da execução da linha anterior, a qual criou uma *goroutine*.
- No exemplo anterior usamos `fmt.Scanln` justamente para forçar a `main` esperar a execução de `f`; remova a chamada à função `fmt.Scanln` e veja o que acontece.

O que é uma *goroutine*?

- Uma *goroutine* é uma *thread* do sistema operacional, a qual é gerenciada pelo ambiente de execução de Go.
- As *goroutines* são leves e compartilham o mesmo espaço de memória.
- Isso significa que podemos criar várias *goroutines* em um mesmo programa.
- O exemplo da próxima transparência adapta o exemplo anterior para que execute cinco *goroutines*.

Exemplo

```
package main

import (
    "fmt"
    "time"
    "math/rand"
)

var MAX int = 5

func f (n int) {
    for i := 0; i < MAX; i++ {
        fmt.Println(n, ":", i)
        time.Sleep(time.Duration(rand.Intn(250)) * time.Millisecond)
    }
}

func main () {
    for i := 0; i < MAX; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

A semântica de uma *goroutine*

- O efeito de executar uma *goroutine* é similar a executar um processo no Unix usando o operador de *background* (&).

Exemplo

- O que acontece se executarmos o programa a seguir sem a palavra chave **go**?

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        fmt.Println(s)
        time.Sleep(1000 * time.Millisecond)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

Canais

- Canais são a maneira que Go usa para efetuarmos comunicação e sincronização entre *goroutines*.
- Usamos o operador `<-` para enviar e receber dados.
- Por exemplo, `c <- d` envia `d` para o canal `c`, enquanto `d := <- c` recebe algum dado do canal `c` e atribui o dado à `d`.
- Isso significa que o dado sempre segue a direção da seta.

Canais

- O mecanismo de sincronização de Go não precisa de *locks* explícitos ou variáveis de condição, pois enviar e receber dados por meio de canais bloqueia a próxima execução enquanto o outro lado estiver pronto para receber ou enviar informações.
- Devemos usar a função **make** para criarmos um canal.

Exemplo

```
package main

import "fmt"

func soma (v []int, c chan int) {
    soma := 0
    for i := 0; i < len(v); i++ {
        soma += v[i]
    }
    c <- soma
}

func main () {
    v := []int{ 1, 2, 3, 4, 5, 6 }
    c := make(chan int)
    meio := len(v) / 2
    go soma(v[:meio], c)
    go soma(v[meio:], c)
    x, y := <- c, <- c
    fmt.Printf("%d + %d = %d\n", x, y, x + y)
}
```

Canais são tipados

- Note que os canais são meios de comunicação tipados.
- Isto é, eles só transmitem dados de um determinado tipo que foi especificado na criação do canal.
- Por exemplo, **chan int** só transmite valores inteiros enquanto **chan string** só transmite strings.

Canais *bufferizados*

- Podemos passar um segundo parâmetro para a função **make**, quando queremos criar canais *bufferizados*.
- O segundo parâmetro cria um canal de capacidade *m*.
- Repare que canais geralmente são síncronos, isto é, um dos lados do canal sempre espera até que o outro lado esteja pronto.
- Usando canais *bufferizados* podemos criar comunicação assíncrona, isto é, enviar bloqueia apenas quando o *buffer* está cheio e receber bloqueia apenas quando o *buffer* está vazio.

Exemplo

```
package main

import "fmt"

func main () {
    c := make(chan int, 3)
    c <- 1
    c <- 2
    c <- 3
    fmt.Println(<- c)
    fmt.Println(<- c)
    fmt.Println(<- c)
}
```

- O que acontece se tentarmos enviar dados para um *buffer* que está cheio ou receber dados de um *buffer* que está vazio?

As funções `range` e `close`

- Em Go podemos usar a função **`close`** para sinalizar o término do envio de dados e a função **`range`** para receber dados de um canal enquanto ele não é fechado.
- Apenas quem envia deve fechar um canal, isto é, o receptor nunca fecha um canal.
- Canais são diferentes de arquivos: geralmente não precisamos fecha-los. Só precisamos fechar um canal quando queremos dizer ao receptor que não haverá mais transmissão de dados.

Exemplo

```
package main

import "fmt"

func fib (n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x + y
    }
    close(c)
}

func main () {
    c := make(chan int, 10)
    go fib(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

O comando **select**

- A linguagem Go também disponibiliza o comando **select**, o qual é similar ao comando **switch**, mas para gerenciar múltiplas comunicações entre *goroutines*.
- O **select** bloqueia a execução até que um caso esteja pronto para execução.
- Um caso é escolhido aleatoriamente quando vários casos estão prontos para executar.

Exemplo

```
package main

import "fmt"

func fib (c, fim chan int) {
    x, y := 0, 1
    for {
        select {
            case c <- x:
                x, y = y, x + y
            case <- fim:
                fmt.Println("fim")
                return
        }
    }
}

func main () {
    c := make(chan int)
    fim := make(chan int)
    go func () {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        fim <- 0
    }()
    fib(c, fim)
}
```


O caso **default**

- Também podemos usar um caso padrão para tentar enviar e receber dados sem bloquear.
- Isto é, o caso padrão é executado quando nenhum outro caso está pronto para execução.

Exemplo

```
package main

import "fmt"
import "time"
import "math/rand"

func say (s string, c chan string) {
    for {
        c <- s
        time.Sleep(time.Duration(rand.Intn(250)) * time.Millisecond)
    }
}

func main () {
    c1 := make(chan string)
    c2 := make(chan string)
    c3 := make(chan string)
    c4 := make(chan string)
    go say("teste", c1)
    go say("1", c2)
    go say("2", c3)
    go say("3", c4)
    end := time.After(time.Second * 10)
    for {
        select {
            case msg1 := <- c1:
                fmt.Println(msg1)
            case msg2 := <- c2:
                fmt.Println(msg2)
            case msg3 := <- c3:
                fmt.Println(msg3)
            case msg4 := <- c4:
                fmt.Println(msg4)
            case <- end:
                fmt.Println("tchau")
                return
            default:
                fmt.Println("esperando")
                time.Sleep(time.Second * 1)
        }
    }
}
```

Exemplo

```
package main

import "fmt"
import "time"
import "math/rand"

func say (s string, c chan string) {
    for {
        c <- s
        time.Sleep(time.Duration(rand.Intn(250)) * time.Millisecond)
    }
}

func main () {
    c1 := make(chan string)
    c2 := make(chan string)
    c3 := make(chan string)
    c4 := make(chan string)
    go say("teste", c1)
    go say("1", c2)
    go say("2", c3)
    go say("3", c4)
    end := time.After(time.Second * 10)
    for {
        select {
            case msg1 := <- c1:
                fmt.Println(msg1)
            case msg2 := <- c2:
                fmt.Println(msg2)
            case msg3 := <- c3:
                fmt.Println(msg3)
            case msg4 := <- c4:
                fmt.Println(msg4)
            case <- end:
                fmt.Println("tchau")
                return
            default:
                fmt.Println("esperando")
                time.Sleep(time.Second * 1)
        }
    }
}
```

O comando **select** também pode ser usado para implementar *timeout* em servidores que atendem diversas requisições.

Variáveis compartilhadas

- Até agora vimos que canais são importantes para controlar a comunicação entre *goroutines*.
- Mas e se não precisamos de comunicação?
- E se quisermos garantir que apenas uma *goroutine* acessa uma determinada variável de cada vez para evitar conflitos?

Exclusão mútua

- Isso significa que queremos exclusão mútua.
- Isto é, queremos garantir que só uma *goroutine* executa por vez.
- Podemos usar semáforos binários para garantir a exclusão mútua, já que um semáforo binário atua como um *mutex*.

`sync.Mutex`

- Go disponibiliza o tipo `sync.Mutex` para que implementemos exclusão mútua.
- Os métodos `Lock` e `Unlock` operam sobre o tipo `sync.Mutex`.
- Nós podemos definir um bloco de código a ser executado em exclusão mútua quando cercamos esse código com `Lock` e `Unlock`.
- Também podemos usar o **`defer`** para garantir que um *mutex* será liberado apenas quando uma função termina a sua execução.

Exemplo

```
package main

import (
    "fmt"
    "sync"
    "time"
)

const N = 100000000

var (
    cont int = 0
    mu sync.Mutex
)

func Inc () {
    mu.Lock()
    cont++
    mu.Unlock()
}

func Valor () int {
    mu.Lock()
    defer mu.Unlock()
    return cont
}

func main () {
    go func () {
        for i := 0; i < N; i++ {
            Inc()
        }
    }()
    for i := 0; i < N; i++ {
        Inc()
    }
    time.Sleep(time.Second)
    fmt.Println(Valor())
}
```

Exercício 1 (0,2)

- Implemente uma função que conta as vogais de uma *string* distribuindo as tarefas entre duas *goroutines*.

- Essa função deve obedecer ao protótipo:

```
func conta_vogais (s string, c chan int)
```

- Dica: a solução do exercício é similar ao exemplo da soma dos elementos de um vetor, isto é, a `main` distribui as tarefas entre duas *goroutines* e consolida o resultado.

Exercício 2 (0,2)

- Implemente uma função que gera números primos dentro de um intervalo de 1 até n .
- Essa função deve obedecer ao protótipo:

`func primos (n int, c chan int)`
- Note que a sua função `primos` não deve imprimir na tela os números primos, mas enviá-los por um canal *bufferizado* para que sejam impressos na `main` do programa.

Exercício 3 (0,2)

- Use exclusão mútua para controlar os acessos à variável `saldo` no código a seguir:

```
package main

import "fmt"

const N = 1000000

var saldo int

func Deposito (v int) {
    saldo = saldo + v
}

func Saque (v int) {
    saldo = saldo - v
}

func Saldo () int {
    return saldo
}

func main () {
    fim := make(chan bool)
    go func (fim chan <- bool) {
        for i := 1; i <= N; i++ {
            Saque(N * i)
        }
        fim <- true
    }(fim)
    for i := 1; i <= N; i++ {
        Deposito(N * i)
    }
    _ = <- fim
    fmt.Println(Saldo())
}
```