Canais em Go

Marcelo Veiga Neves
marcelo.neves@pucrs.br

Linguagem Go

- Go é uma linguagem de programação criada pela Google e lançada em código livre em novembro de 2009.
- O projeto inicial da linguagem foi feito em setembro de 2007 por Robert Griesemer, Rob Pike e Ken Thompson.
- Projetada para ser simples, eficiente e de fácil leitura.
- É uma linguagem compilada e focada em produtividade e programação concorrente.

Concorrência em Go

- Em Go, a concorrência é realizada através de **goroutines**.
- Uma goroutine é uma função leve que é executada concorrentemente com outras funções.
  - Não é exatamente um *thread* tradicional de sistema operacional
  - Implementação própria para ter múltiplos fluxos de execução
  - Goroutines são muito mais leves que threads tradicionais.
  - Milhares de goroutines podem ser executadas simultaneamente sem sobrecarregar o sistema.

Criação de goroutines

- Uma goroutine é lançada simplesmente usando a palavra-chave **go** antes de uma chamada de função.
- Sem a palavra go, ela funciona como uma função normal

```
func minhaGoroutine() {
    // Código da goroutine
}

go minhaGoroutine() // Inicia uma goroutine
```

Canais

- Comunicação: Canais são a forma nativa de comunicação entre goroutines, eliminando problemas comuns de compartilhamento de memória e condições de corrida.
- Sincronização: Além da comunicação, canais podem ser usados para sincronizar a execução entre diferentes goroutines, facilitando a coordenação de tarefas concorrentes.
- Padrões de design: Go incentiva padrões de design concorrente claros e mantém o código conciso, tornando mais fácil escrever e entender programas concorrentes.

Modelo concorrência baseado em troca de mensagens

- Evita bloqueios:
  - Compartilhamento direto de memória depende de primitivas de sincronização como mutexes e semáforos.
  - Go promove um modelo baseado em compartilhamento de mensagens para evitar bloqueios e deadlocks.
- Desacoplamento:
  - Este modelo facilita a criação de componentes desacoplados, onde a comunicação é feita exclusivamente através de canais.
  - Aumenta a robustez e facilita a manutenção do código.

Tipos de canais

- Os canais podem ser classificados com base em suas capacidades de transmissão e direção
- Bufferização:
  - Canais Não Bufferizados
  - Canais Bufferizados
- Direção:
  - Canais bidirecionais
  - Canais direcionais (ou unidirecionais)

Canais não bufferizados

- Sincronizam a transmissão e recepção de dados (bloqueante).
- Quando uma goroutine envia um dado, ela bloqueia até que outra goroutine receba esse dado.
- Ideal para troca direta de mensagens e sincronização entre goroutines.

```
ch := make(chan int) // Cria um canal não bufferizado

ch <- 1 // Envia um valor para o canal

num := <-ch // Recebe um valor do canal
```

Canais bufferizados

- Permitem o envio de vários dados antes de bloquear.
- Tem uma capacidade definida.
- Um envio bloqueia apenas quando o buffer está cheio, e uma recepção bloqueia quando o buffer está vazio.
- Útil comunicação não bloqueante ou trabalhar com fluxos de dados.

```
ch := make(chan int, 2) // Cria um canal bufferizado com capacidade para 2 itens

ch <- 1 // Envia valores para o canal sem bloquear até o buffer estar cheio
ch <- 2

fmt.Println(<-ch) // Recebe um valor do canal
fmt.Println(<-ch) // Recebe o segundo valor do canal
```

Canais direcionais

- Permite comunicação em uma direção apenas.
  - chan<-: Pode apenas enviar dados para o canal.
  - <-chan: Pode apenas receber dados do canal.
- Usado para limitando as operações que goroutines podem realizar com o canal.

```go
func send(ch chan<- int) {
    ch <- 1 // Envia um valor para o canal
}

func main() {
    ch := make(chan int) // Cria um canal
    go send(ch)
    num := <-ch // Recebe um valor do canal
    fmt.Println(num)
}
```

```go
func receive(ch <-chan int) {
    num := <-ch // Recebe um valor do canal
    fmt.Println(num)
}

func main() {
    ch := make(chan int) // Cria um canal
    go receive(ch)
    ch <- 1 // Envia um valor para o canal
    ...
}
```

Exemplo de uso

```go
// Função que simula o envio de um número após algum cálculo ou operação.
// Ela envia um número para o canal fornecido.
func sendNumber(ch chan<- int, num int) {
    // Simula um cálculo ou operação que leva 2 segundos.
    time.Sleep(2 * time.Second)
    ch <- num // Envia 'num' através do canal.
}

func main() {
    // Cria um canal de inteiros.
    numChannel := make(chan int)

    // Inicia a goroutine sendNumber para enviar um número através do canal.
    go sendNumber(numChannel, 42)

    // Recebe o número enviado para o canal e armazena na variável 'num'.
    num := <-numChannel

    // Imprime o número recebido.
    fmt.Println("Recebi o número:", num)
}
```

Exercício

- Crie um programa em Go com duas goroutines disparadas a partir da main().
- Uma goroutine deve imprimir números de 1 a 10 a cada 1 segundo
- A outra goroutine deve imprimir números de 10 a 1 a cada 1 segundo
- A main deve esperar até que as duas goroutines terminem sua execução
- Utilize um canal bufferizado para fazer a main() esperar pela conclusão das duas goroutines