Goroutines

Funções assíncronas sem perder a sanidade

Rodrigo Brito

Sobre mim

Rodrigo Brito - Software Developer

GitHub - github.com/rodrigo-brito (https://github.com/rodrigo-brito)

Twitter - @RodrigoFBrito (https://twitter.com/RodrigoFBrito)

Steam - rBRITO/rodrigo-brito (https://steamcommunity.com/id/rodrigo-brito)

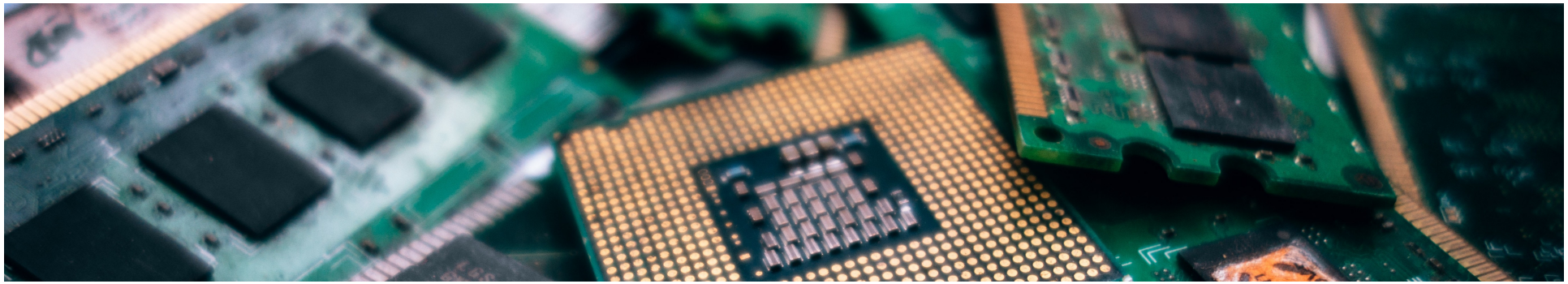
Atualidade

Aplicações complexas

Diversidade de eventos

Hardware extremamente potente e de baixo custo

- Processadores com vários núcleos (4, 8, 12, etc.)
- Celulares mais potentes que computadores
- Recurso computacional ocioso



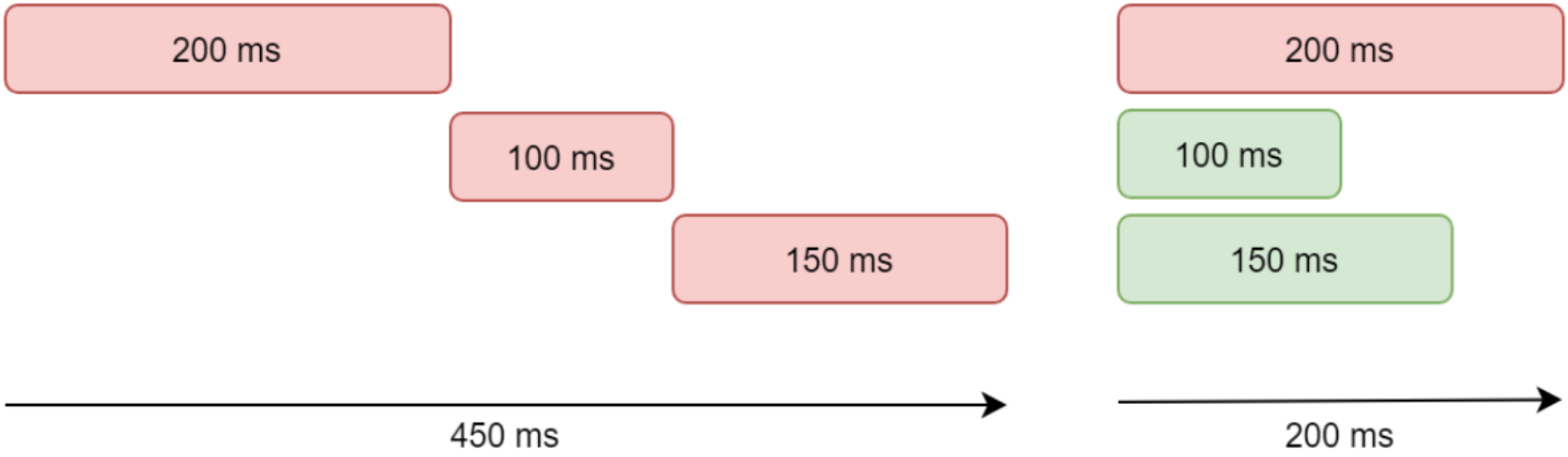
Hardware evoluiu. E nossos softwares?

Como reverter isso?

Concorrência e paralelismo

Utilizar recursos de linguagem que usurfruem da concorrência/paralelismo

- **Actors** - Erlang, Scala
- **Fibers** - Ruby
- **Threads** - Java, C++, etc.
- **Goroutines** - Go



Goroutine

Uma **lightweight thread** gerenciada pelo Go runtime

Uma **Goroutine** é uma função que é capaz de executar **concorrentemente** com outras funções - **Caleb Doxsey** (An Introduction to Programming in Go) (https://www.golang-book.com/books/intro/10)

Compartilha o mesmo espaço de memória, logo precisa que o acesso seja sincronizado

- Palavra reservada **go**
- Não é permitido atribuição direta

```
go example()
```

```
go func(){
    result := exampleTwo()
}()
```

Exemplo básico

Impressão síncrona

Ordem de execução garantida

```
func task(value int) {
    fmt.Printf("Taks %d done!\n", value)
}
```

```
func main() {
    for i := 0; i < 10; i++ {
        task(i)
    }
    fmt.Println("All done!")
}
```

Run

Impressão assíncrona

Ordem de execução não garantida

Necessidade de contenção de fluxo **time.Sleep()**

```
func task(value int) {
    fmt.Printf("Taks %d done!\n", value)
}
```

```
func main() {
    for i := 0; i < 10; i++ {
        go task(i)
    }
    time.Sleep(time.Second)
    fmt.Println("All done!")
}
```

Run

Sleep é muito feio :(

Como saber quando irá finaliza cada tarefa?

WaitGroup

Cada tarefa assíncrona sinaliza sua finalização

```
func task(value int, wg *sync.WaitGroup) {
    defer wg.Done() // executa ao finalizar
    fmt.Printf("Taks %d done!\n", value)
}
```

```
func main() {
    wg := new(sync.WaitGroup)

    for i := 0; i < 10; i++ {
        wg.Add(1)
        go task(i, wg)
    }

    wg.Wait()
    fmt.Println("All done!")
}
```

Run

Channel

Valor via **channel** sinaliza finalização

```
func task(value int, done chan<- bool) {
    fmt.Printf("Task %d done!\n", value)
    done <- true
}
```

```
func main() {
    done := make(chan bool)
    go task(1, done)
    go task(2, done)
    go task(3, done)
    <-done
    <-done
    <-done
    fmt.Println("All done!")
}
```

Run

Comunicação

Channels

Channels são estruturas de comunicação que podem ser utilizadas entre Goroutines concorrentes

Permite leitura e escrita assíncrona de forma segura

```
// criar canal
ch := make(chan int)
ch := make(chan int, 5)
```

```
// ler valor do canal
value := <-ch
// escrever valor no canal
ch <- 13
```

```
// fechar canal
close(ch)

// iterar valores que são recebidos
for i := range ch {
    // ...
}
```

Mutex

```
wg := new(sync.WaitGroup)
mutex := new(sync.Mutex)

var counter int32 = 1
atomic.AddInt32(&counter, 2) // Operação atômica

wg.Add(1)
go func() {
    mutex.Lock() // bloqueia acesso a contador
    counter = counter + 2
    mutex.Unlock() // libera acesso a contador
    wg.Done()
}()
wg.Wait()
```

Run

- Risco de deadlocks
- Mutex proporciona um código mais sequencial
- **sync** e **sync/atomic** - Alterantivas para operações concorrentes

Mutex

Channels

Exemplo de comunicação

Exemplo: Calcular raiz quadrada

Gerar 10.000 números e calcular a raiz quarada aproximada via método de Newton

$$y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right)$$

Cenário síncrono
- Gerar lista com 10.000 números
- Iterar lista e efetuar cálculos

Cenário assíncrono - Ideal
- Gerar números e distribuí-los a medida que criados
- Várias tarefas assíncronas efetuando os cálculos simultaneamente

Necessário um canal de comunicação para geração de números e um canal de comunicação para resultados

Cenário síncrono

```
func sqrtSync(value float32) float32 {
    result := value
    for i := 0; i <= int(value); i++ {
        result = (result + value/result) / 2
    }
    time.Sleep(time.Millisecond) // Simulando processo pesado
    return result
}

func randVectorSync(size int) []float32 {
    result := make([]float32, size)
    for i := range result {
        result[i] = float32(rand.Intn(99) + 1)
    }
    return result
}
```

```
    result := make([]float32, 10000)
    values := randVectorSync(10000)

    for i, value := range values {
        result[i] = sqrtSync(value)
    }
```

Run

Cenário assíncrono - Produtor

Gerador de números aleatórios

- Recebe um **channel** de escrita (chan<-)
- Fecha o canal ao finalizar

```
func randVectorAsync(size int, values chan<- float32) {
    for i := 0; i < size; i++ {
        values <- float32(rand.Intn(99) + 1)
    }
    close(values)
}
```

Cenário assíncrono - Consumidor

- Recebe um **channel** de leitura (<-chan) - Valores de entrada
- Recebe um **channel** de escrita (chan<-) - Resultados
- Itera valores no canal até que seja fechado
- Transmite os resultados por outro canal

```
func sqrtAsync(values <-chan float32, results chan<- float32, wg *sync.WaitGroup) {
    for value := range values {
        results <- sqrtSync(value)
    } // finaliza quando canal `values` fechar
    wg.Done()
}
```

```
func sqrtSync(value float32) float32 {
    result := value
    for i := 0; i <= int(value); i++ {
        result = (result + value/result) / 2
    }
    time.Sleep(time.Millisecond) // Simulando processo pesado
    return result
}
```

Cenário assíncrono - Composição

- 1 Produtor e 8 Consumidores

```
func main() {
    const asyncTasks = 8

    startTime := time.Now()
    values := make(chan float32, 10000)
    results := make(chan float32, 10000)

    // generator
    go randVectorAsync(10000, values)

    wg := new(sync.WaitGroup)
    wg.Add(asyncTasks)
    for i := 0; i < asyncTasks; i++ {
        // consumers
        go sqrtAsync(values, results, wg)
    }

    wg.Wait()
    fmt.Println("Time: ", time.Now().Sub(startTime))
}
```

Run

Utilidades

Async.Run() - https://github.com/StudioSol/async

```
user := new(model.User)
err := async.Run(ctx,
    func(ctx context.Context) error {
        return storage.DeleteFile(ctx, bucket, fileName)
    },
    func(ctx context.Context) error {
        return mail.SendConfirmation(ctx, address, message)
    },
    func(ctx context.Context) error {
        user, err = repository.FetchUser(ctx, ID)
        return err
    },
)
if err != nil {
    log.Log(err)
}
```

Ainda mais simples

```
var tasks []func(ctx context.Context) error
err := async.Run(tasks...)
```

Outros links

- Go Tracer - André Carvalho - Gophercon BR 2017 (https://github.com/gopherconbr/2017-talks/blob/master/go-execution-tracer.pdf)
- Defer, Panic, and Recover - Go Blog (https://blog.golang.org/defer-panic-and-recover)
- Monitorar Goroutines - (Github - bcicen/grmon) (https://github.com/bcicen/grmon)
- async.Waterfall (Sem Context) - (Github - rafaeldias/async) (https://github.com/bcicen/grmon)
- Goroutines e Context - (Go Concurrency Patterns: Context) (https://blog.golang.org/context)
- Concurrency Is Not Parallelism - Rob Pike - Heroku Talk (https://vimeo.com/49718712)
- Padrões de Projeto (Composição de Channels) - André - Meetup Go BH ()

Thank you

Rodrigo Brito
contato@rodrigobrito.net (mailto:contato@rodrigobrito.net)
https://www.rodrigobrito.net (https://www.rodrigobrito.net)