Contexto

O pacote **context** tem classe **Context** que nos permite transportar dados de escopo de request, sinais de cancelamento e outros sinais entre várias **goroutines** que estão processando uma requisição.

O uso de **Context** deve seguir algumas regras:

1. Contextos não devem ser utilizados para passar parâmetros para funções. Apenas valores de escopo de request, a serem compartilhados por várias goroutines.
2. Jamais passe contextos nulos. Use **Context.TODO()** se não souber o tipo de contexto.

Passando parâmetros de escopo de request

O tipo mais comum de contexto é o **Background** e podemos criá-lo e passá-lo para outras funções ou goroutines:

```
ctx := context.Background()
process(ctx)
```

Para podermos passar parâmetros, podemoos criar outro contexto, baseado no primeiro, com o método **WithValue()**:

```
func otherProcess(ctx context.Context) {
    fmt.Printf("\nReceived in context: %s and %s\n",
        ctx.Value("parameter"), ctx.Value("otherParameter"))
}

func process(ctx context.Context) {
    fmt.Printf("\nReceived in context: %s\n", ctx.Value("parameter"))
    myCtx := context.WithValue(ctx, "otherParameter", "other value")
    otherProcess(myCtx)
}

func main() {
    ctx := context.Background()
    ctx = context.WithValue(ctx, "parameter", "value")
    process(ctx)
}
```

Como pode ver, criei um contexto com um parâmetro e passei para uma função, que criou outro contexto adicionando outro valor e passou para uma segunda função.

Cancelamento

Como saber se o processamento de uma transação foi cancelado? Você pode estar no meio de um processamento complexo e o usuário solicitou o cancelamento da transação. Como evitamos ficar processando até o final?

Há várias maneiras de fazer isso. O contexto pode ter um canal (channel) indicando que foi cancelado.

Vamos imaginar uma função que, quando invocada, calcula números aleatórios repetidamente. Ela não tem como saber quantos números você precisa. Vamos utilizar o contexto para isso. Primeiramente, vamos criar a função **main** que criará um contexto com o método **WithCancel()**. Este contexto tem um canal que pode ser acessado com o método **Done()** indicando que o contexto recebel um sinal de cancelamento.

Essa função chamará outra função repetidamente, um número indeterminado de vezes (nesse caso, 3):

```
func main() {
    ctx := context.Background()
    newCtx, cancel := context.WithCancel(ctx)
    count := 0
    for n := range invoke(newCtx) {
        fmt.Println(n)
        count++
        if count > 2 {
            break
        }
    }
    cancel()
    time.Sleep(1 * time.Second)
    fmt.Println("Main finished")
}
```

A função **invoke()** simplesmente cria um canal e passa junto com o contexto para uma **goroutine**:

```
func invoke(ctx context.Context) <-chan int {
    ch := make(chan int)
    go raffleNumbers(ctx, ch)
    return ch
}
```

Essa função retorna o canal, portanto, podemos utilizá-lo no **for** da função **main**, para receber os valores calculados.

Agora, vejamos a **goroutine**:

```
func raffleNumbers(ctx context.Context, ch chan int) {
    seed1 := rand.NewSource(time.Now().UnixNano())
    r := rand.New(seed1)
    number := r.Int()
    for {
        select {
```

```
            case <-ctx.Done():
                fmt.Println("Cancel")
                return

            case ch <- number:
                number = r.Int()
            }
        }
    }
```

Esta função poderia ser muito complexa, executando diversas **queries** e fazendo requests a outros serviços. Se a transação for cancelada, ela precisa ser cancelada para evitar desperdício de recursos.

Ela usa um **for** eterno porque precisa repetir o cálculo e retornar elementos utilizando o canal que recebeu.

Se o canal fornecido pelo método **Done()** tiver alguma coisa, então o processamento deve parar. Se o canal estiver solicitando mais números, ela envia de volta.

Como cancelamos o contexto?

Se observar na função **main** eu recebo um contexto e uma função de cancelamento, quando uso o método **WithCancel()** para criar o contexto:

```
    ctx := context.Background()
    newCtx, cancel := context.WithCancel(ctx)
```

Ao invocar essa função **cancel** eu sinalizo o contexto que o processamento deve terminar:

```
    cancel()
```

Poderia também usar um **defer**:

```
    ctx := context.Background()
    newCtx, cancel := context.WithCancel(ctx)
    defer cancel()
```

Servidores

Podemos e devemos utilizar **contexto** com servidores **REST**, pois o usuário pode simplesmente fechar a janela do navegador antes do processamento do request terminar.

Este é um servidor bem simples que faz isso:

```
package main
```

```
import (
    "fmt"
    "net/http"
    "time"
)

func VerySlowFunction() {
    time.Sleep(6 * time.Second)
}

func Process(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    go VerySlowFunction()

    select {
    case <-time.After(10 * time.Second):
        w.Write([]byte("OK"))
    case <-ctx.Done():
        fmt.Println("Cancelled")
    }

}

func main() {
    http.ListenAndServe(":8080", http.HandlerFunc(Process))
}
```

Ele utiliza o **Contexto** do **Request** (r.Context()), que já tem o cancel embutido. Nossa **handler** function (Process) simplesmente pega o contexto do request e usa em um select.

O primeiro **case** será processado se em até 10 segundos tivermos uma resposta da **goroutine VerySlowFunction()**. E o segundo **case** será acessado se o canal do contexto sinalizar o cancelamento.