GO-05 06: Асинхронность. Каналы ч.2. Context.

Описание:

При работе с каналами полезно использовать выражение select. Внешне оно напоминает switch, но предназначено именно для работы с каналами.

```
select {
    case <-ch:
        // some operations
    default:
        // some operations
}
```

select выполняет соответствующее действие, когда с каким-либо из указанных каналов можно взаимодействовать. Если таких каналов несколько, то канал выбирается случайно. Если ни один из них не может быть обработан, то выполняется условие default (если оно было указано).

```
package main

import (
      "fmt"
      "time"
)

func main() {
semaphore := make(chan int, 3)
done := make(chan struct{})
i := 0

go func() {
      for ; ; i++ {
            semaphore <- i
            time.Sleep(time.Millisecond * 100)
      }
}()
go func() {
      time.Sleep(time.Millisecond * 1000)
```

```
            done <- struct{}{}
      }()
      msg := 0
L:
      for {
            select {
            case msg = <-semaphore:
                  fmt.Println(msg)
            case <-done:
                  fmt.Println("done")
                  break L
            default:
                  if msg >= 20 {
                        break L
                  }
                  fmt.Println("waiting")
                  time.Sleep(time.Millisecond * 200)

            }
      }
      fmt.Println("success")
}
```

Обратите внимание на использование выхода из цикла с меткой через break L. Если не использовать метку, то break будет относиться к выражению select, и мы никогда не выйдем из цикла for.

Часто в golang используют контексты (context). Их используют для прерывания работы функции и передают как параметр при запросе к БД или по сети. Контексты из одноименного стандартного пакета golang обладают полем Done() типа <-chan struct{}, с помощью которого можно узнать о завершении работы.

```
package main

import (
      "context"
      "fmt"
      "time"
)

func main() {
```

```
	ctx, cancel := context.WithTimeout(context.Background(),
time.Millisecond*300)
	tick := time.NewTicker(time.Millisecond * 50)

	for {
		select {
		case t := <-tick.C:
			fmt.Println(t)
			cancel()
		case <-ctx.Done():
			fmt.Println("context deadline exceeded")
			return
		default:
			fmt.Println("waiting")
			time.Sleep(time.Millisecond * 20)
		}
	}
}
```

В этом примере создается контекст с тайм-аутом на основе background-контекста. Когда пройдет указанное время (300 миллисекунд), канал Done будет закрыт, следовательно, все его «слушатели» получат из него сообщение struct{}{} (и false в качестве второго параметра). С помощью вызова функции cancel() можно принудительно закрыть канал, не дожидаясь, например, тайм-аута. Использование каналов Done крайне важно для того, чтобы не допускать goroutine leaks (утечек горутин). Каждая горутина, слушающая канал done, может быть завершена при закрытии канала. Пример из стандартной библиотеки:

```
package main

import (
	"context"
	"fmt"
)

func main() {
	// gen generates integers in a separate goroutine and
	// sends them to the returned channel.
	// The callers of gen need to cancel the context once
	// they are done consuming generated integers not to leak
	// the internal goroutine started by gen.
	gen := func(ctx context.Context) <-chan int {
```

```
                dst := make(chan int)
                n := 1
                go func() {
                        for {
                                select {
                                case <-ctx.Done():
                                        return // returning not to leak the
goroutine
                                case dst <- n:
                                        n++
                                }
                        }
                }()
                return dst
        }

        ctx, cancel := context.WithCancel(context.Background())
        defer cancel() // cancel when we are finished consuming integers

        for n := range gen(ctx) {
                fmt.Println(n)
                if n == 5 {
                        break
                }
        }
}
```

Полезные ссылки:

- package context
- A tour of Go. Concurrency p5

Задание:

1. Создайте у себя в проекте module05 ветку module05_06.
2. Модифицируйте код из прошлого задания так, чтобы основная функция и все горутины завершались по истечении таймера контекста длительностью 100 миллисекунд.
3. В ответе пришлите ссылку на MP в ветку master своего проекта ветки module05_06.