



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ (ИУ6)

НАПРАВЛЕНИЕ ПОДГОТОВКИ 09.04.01 - ИУ6-32Б

**О Т Ч Е Т**

**по лабораторной работе № 5**

**Название: Основы асинхронного программирования на Golang**

**Дисциплина: Языки интернет-программирования**

Студент

ИУ6-32Б

(Группа)

(Подпись, дата)

Кулиев Э.

(И.О. Фамилия)

Преподаватель

(Подпись, дата)

Шульман В.Д.

(И.О. Фамилия)

Москва, 2024

# Цель работы: изучение основ асинхронного программирования с использованием языка Golang.

## Задание 1

Вам необходимо написать функцию `calculator` следующего вида:

```
func calculator(firstChan <-chan int, secondChan <-chan int, stopChan <-chan struct{}) <-chan int
```

Функция получает в качестве аргументов 3 канала, и возвращает канал типа `<-chan int`.

- в случае, если аргумент будет получен из канала `firstChan`, в выходной (возвращенный) канал вы должны отправить квадрат аргумента.
- в случае, если аргумент будет получен из канала `secondChan`, в выходной (возвращенный) канал вы должны отправить результат умножения аргумента на 3.
- в случае, если аргумент будет получен из канала `stopChan`, нужно просто завершить работу функции.

Функция `calculator` должна быть неблокирующей, сразу возвращая управление. Ваша функция получит всего одно значение в один из каналов - получили значение, обработали его, завершили работу.

После завершения работы необходимо освободить ресурсы, закрыв выходной канал, если вы этого не сделаете, то превысите предельное время работы.

Рисунок 1

На рисунке 2,3,4 представлен мой результат и некоторые части кода.

```
Codeium: Refactor | Explain | Generate GoDoc | X
8 func calculator(first <-chan int, second <-chan int, stop <-chan struct{}) <-chan int {
9     res := make(chan int)
10    go func() {
11        select {
12            case num := <-first:
13                res <- num*num
14                close(res)
15                return
16            case num := <-second:
17                res <- num*3
18                close(res)
19                return
20            case <-stop:
21                fmt.Println("stop")
22                close(res)
23                return
24        }
25    }()
26    return res
27 }
28
```

Рисунок 2

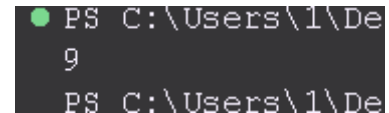
```
PS C:\Users\1\D
stop
0
```

Рисунок 3(вывод остановки)



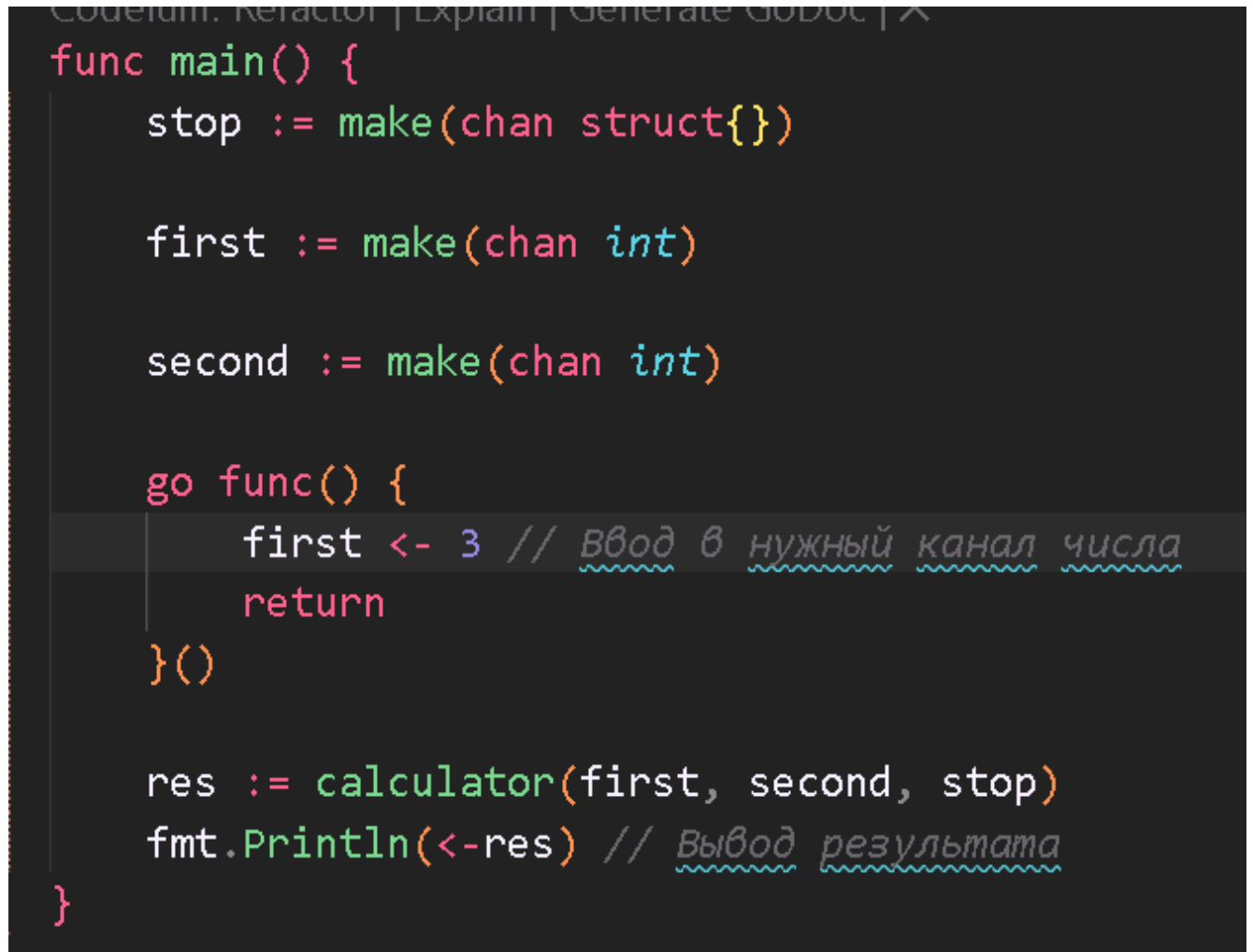
```
PS C:\Users\1\Desktop  
4  
PS C:\Users\1\Desktop
```

Рисунок 4 ( ввод в 1 канал числа 2 )



```
PS C:\Users\1\Desktop  
9  
PS C:\Users\1\Desktop
```

Рисунок 5 ( ввод в 2 канал числа 3 )



```
func main() {  
    stop := make(chan struct{})  
  
    first := make(chan int)  
  
    second := make(chan int)  
  
    go func() {  
        first <- 3 // Ввод в нужный канал числа  
        return  
    }()  
  
    res := calculator(first, second, stop)  
    fmt.Println(<-res) // Вывод результата  
}
```

Рисунок 6 ( Тестирующая часть программы )

В задании мы должны были реализовать работу функции `calculator`, которая принимает 3 канала в качестве аргументов и возвращает результирующий канал. Для реализации задачи мы создали горутину в которой с помощью `select` обрабатываем в какой канал поступило значение и выводим результат.

В задании было сказано сделать для 1 определенного значения в канале, но я решил усовершенствовать и сделал так, что пока в канал `stop` не попадет какое нибудь значение, символизирующее остановку работы, функция будет выполнять действия. Для этого блок `select` можно обернуть в бесконечный цикл `for{}`.

```
17 stop := make(chan struct{})
18
19 first := make(chan int)
20
21 second := make(chan int)
22
23 go func() {
24     first <- 3 // Ввод в нужный канал числа
25     first <- 4 // Ввод в нужный канал числ
26     second <- 100 // Ввод в нужный канал числ
27     stop <- struct{}{} // Сигнал остановки
28     return
29 }()
30
31 res := calculator(first, second, stop)
32 for num := range res {
33     fmt.Println(num)
34 }
35
36 fmt.Println(<-res) // Вывод результата
```

LEMS 27 OUTPUT DEBUG CONSOLE COMMENTS PORTS

▼ TERMINAL

```
0
● PS C:\Users\1\Desktop\labs_git\web-5> go run "c:\Users\1\Desktop\labs_
9
0
● PS C:\Users\1\Desktop\labs_git\web-5> go run "c:\Users\1\Desktop\labs_
9
16
300
stop
0
○ PS C:\Users\1\Desktop\labs_git\web-5> █
```

Рисунок 7(пример вывода)

## Задание 2

Напишите элемент конвейера (функцию), что запоминает предыдущее значение и отправляет значения на следующий этап конвейера только если оно отличается от того, что пришло ранее.

Ваша функция должна принимать два канала - `inputStream` и `outputStream`, в первый вы будете получать строки, во второй вы должны отправлять значения без повторов. В итоге в `outputStream` должны остаться значения, которые не повторяются подряд. Не забудьте закрыть канал ;)

Функция **должна** называться `removeDuplicates()`

Выводить или вводить ничего не нужно!

Рисунок 8

```

Codeium: Refactor | Explain | Generate GoDoc | X
func removeDuplicates(in, out chan string) {
    usedStrokes := ""
    for value := range in {
        if strings.Contains(usedStrokes, value) {
            continue
        } else {
            usedStrokes += value + " "
        }
    }
    for _, value := range strings.Split(usedStrokes, " ") {
        out <- value
    }
    close(out)
}

```

Рисунок 9 (мой результат)

```

func main() {
    inputStream := make(chan string)
    outputStream := make(chan string)
    go removeDuplicates(inputStream, outputStream)

    go func() {
        defer close(inputStream)

        for _, r := range "112334456" {
            inputStream <- string(r)
        }
    }()

    for x := range outputStream {
        fmt.Print(x)
    }
}

```

Рисунок 10 ( тестирующая программа )

```
PS C:\Users\1\>
123456
```

Рисунок 11 ( пример вывода )

Для получения результата на входной канал мы получаем строку с цифрами, а на выходной канал должны отправить исправленную строку без повторов цифр. Для этого я создал переменную `usedStrokes`, в которую добавляю новые значения пока прохожусь по строке из входного канала, при этом избегая добавления дубликатов путем `strings.Contains`.

### Задание 3

Внутри функции `main` (функцию объявлять не нужно), вам необходимо в отдельных горутинах вызвать функцию `work()` 10 раз и дождаться результатов выполнения вызванных функций.

Функция `work()` ничего не принимает и не возвращает. Пакет `"sync"` уже импортирован.

Рисунок 12

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

Codeium: Refactor | Explain | Generate GoDoc | X
func work(wg *sync.WaitGroup) {
    defer wg.Done()
    time.Sleep(time.Millisecond * 50)
    fmt.Println("done")
}
```

Рисунок 13 ( описание функции `work` )

```

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

    for i := 0; i < 10; i++ {
        wg.Add(1) // увеличиваем счетчик горутин в группе
        go work(wg) // Вызываем функцию work в отдельной горутине
    }

    wg.Wait() // ожидаем завершения всех горутин в группе
    fmt.Println("Горутин завершили выполнение")
}

```

Рисунок 14 ( основная часть программы )

```

PS C:\Users\1\Desktop\labs_git\web-5> g
done
done
done
done
done
done
done
done
done
done
done
Горутин завершили выполнение

```

Рисунок 15 ( пример вывода )

Для реализации задачи мы воспользовались sync, с помощью которого мы создали группу ожидания выполнения горутин, ее смысл в том, чтобы горутин выполнялись поочередно. Далее в цикле мы наполняем группу 10 горутинами work. Ожидаем завершения всех горутин и в конце выводим сообщения об успешном завершении программы.

## Заключение

Я научился работать с асинхронностью в Golang. Путем решения нескольких задач разобрался в особенностях асинхронного программирования в Golang.