



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 09.03.01 - ИНФОРМАТИКА И
ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

О Т Ч Е Т

по лабораторной работе № 5

Название: Основы асинхронного программирования на Golang

Дисциплина: Языки интернет-программирования

Студент

ИУ6-31Б

(Группа)

(Подпись, дата)

К.С. Гошко

(И.О. Фамилия)

Преподаватель

В.Д. Шульман

(Подпись, дата)

(И.О. Фамилия)

Москва, 2024

Цель работы — изучение основ асинхронного программирования с использованием языка Golang.

Задание:

1. Ознакомьтесь с разделом "3. Map, файлы, интерфейсы, многопоточность и многое другое" курса <https://stepik.org/course/54403/info>
2. Сделайте форк данного репозитория в GitHub, склонируйте получившуюся копию локально, создайте от мастера ветку dev и переключитесь на неё
3. Выполните задания. Ссылки на задания содержатся в README-файлах в директории projects
4. Сделайте отчёт и поместите его в директорию docs
5. Зафиксируйте изменения, сделайте коммит и отправьте полученное состояние ветки dev в ваш удаленный репозиторий GitHub
6. Через интерфейс GitHub создайте Pull Request dev --> master
7. На защите лабораторной работы продемонстрируйте открытый Pull Request. PR должен быть направлен в master ветку вашего репозитория

Задачи:

1. 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!

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

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

3. 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.

Код для задания 1:

```
package main

import "fmt"

func main() {
    inputStream := make(chan string)
    outputStream := make(chan string)
    go removeDuplicates(inputStream, outputStream)
    go func() {
        inputStream <- "qwerty"
        inputStream <- "qwer"
        inputStream <- "qwerty"
        inputStream <- "vsc"
        inputStream <- "vsc"
        close(inputStream)
    }()
    for x := range outputStream {
        fmt.Print(x)
    }
    fmt.Print("\n")
}
```

```

func removeDuplicates(inputStream, outputStream chan string) {
    var Value string
    for v := range inputStream {
        if Value != v {
            outputStream <- v
            Value = v
        }
    }
    close(outputStream)
}

```

Тест задания представлен на рисунке 1

The screenshot shows a Go IDE with a function call in a file named `main.go`. The function `go func() {` is called with five arguments: `inputStream <- "qwerty"`, `inputStream <- "qwer"`, `inputStream <- "qwerty"`, `inputStream <- "vsc"`, and `inputStream <- "vsc"`. The function is then closed with `close(inputStream)` and the function call is ended with `()`. Below the code, the terminal window shows the output of the program. The output consists of five lines: `p`, `b`, `r`, `qwerty`, and `qwer`. The terminal also shows the command `go run main.go` being executed in the directory `D:\Git\web-5\projects\pipeline`.

Рисунок 1 – Результат работы программы.

Код задания 2:

```
package main
```

```

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

```

```
func work() {
```

```

        time.Sleep(time.Millisecond * 50)

        fmt.Println("done")
    }

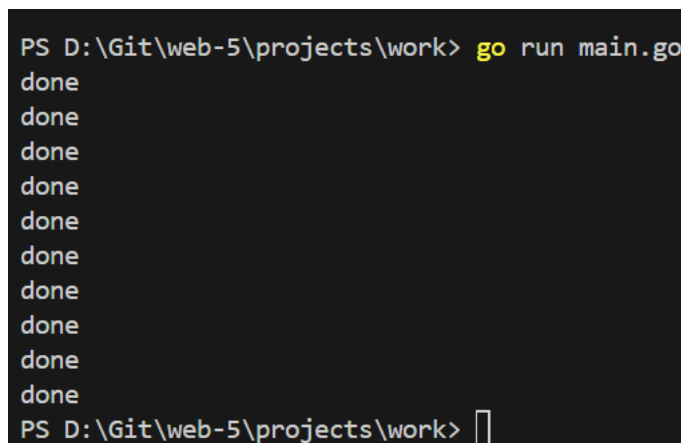
func main() {
    // необходимо в отдельных горутинках вызвать функцию work() 10 раз и дождаться результатов
    // выполнения вызванных функций
    wg := new(sync.WaitGroup)

    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(wg *sync.WaitGroup) {
            defer wg.Done()
            work()
        }(wg)
    }

    wg.Wait()
}

```

Тест задания 2 представлен на рисунке 2.



```

PS D:\Git\web-5\projects\work> go run main.go
done
done
done
done
done
done
done
done
done
done
done
PS D:\Git\web-5\projects\work> 

```

Рисунок 2 – Результат выполнения программы

Код задания 3:

```

package main

import (
    "fmt"
    "sync"
)

func main() {
    firstch := make(chan int)
    secondch := make(chan int)
}

```

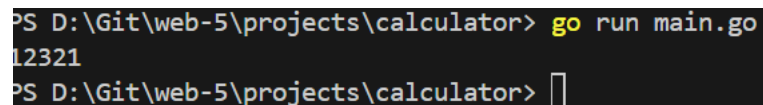
```

stopch := make(chan struct{ })
ch := calculator(firstch, secondch, stopch)
wg := sync.WaitGroup{ }
wg.Add(1)
go func() {
    for ans := range ch {
        fmt.Println(ans)
    }
    wg.Done()
}()
firstch <- 111
wg.Wait()
}

func calculator(firstChan <-chan int, secondChan <-chan int, stopChan <-chan struct{ }) <-chan int {
    res := make(chan int)
    go func(res chan int) {
        defer close(res)
        select {
            case x := <-firstChan:
                res <- x * x
            case x := <-secondChan:
                res <- x * 3
            case <-stopChan:
            }
        }(res)
        return res
    }
}

```

Тест задания представлен на рисунке 3.



```

PS D:\Git\web-5\projects\calculator> go run main.go
12321
PS D:\Git\web-5\projects\calculator> █

```

Рисунок 5 – Результат выполнения программы

Вывод: асинхронность Golang помогает выполнять параллельные задачи, а также вычислять результат для входного потока данных.