WB Tech: level # 2 (Golang)

# Как делать задания

Никаких устных решений — только код. Одно решение — один файл с хорошо откомментированным кодом. Каждое решение или невозможность решения надо объяснить.

Решения задач должны быть размещены в публичном Git-репозитории и оформлены в соответствии со [следующей структурой](https://github.com/petrovichhh/wbschool_exam_L2).

Разрешается и приветствуется использование любых справочных ресурсов, привлечение сторонних экспертов и т.д. и т.п.

Основной критерий оценки — четкое понимание «как это работает». Некоторые задачи можно решить несколькими способами, в этом случае требуется привести максимально возможное количество вариантов и обосновать наиболее оптимальный из них, если таковой имеется.

Можно задавать вопросы, как по условию задач, так и об их решении.

# Задания

## Паттерны проектирования

Реализовать паттерны, объяснить применимость каждого паттерна, плюсы и минусы, а также реальные примеры использования паттерна на практике.

1. Паттерн [«фасад»](https://en.wikipedia.org/wiki/Facade_pattern).
2. Паттерн [«строитель»](https://en.wikipedia.org/wiki/Builder_pattern).
3. Паттерн [«посетитель»](https://en.wikipedia.org/wiki/Visitor_pattern).
4. Паттерн [«комманда»](https://en.wikipedia.org/wiki/Command_pattern).
5. Паттерн [«цепочка вызовов»](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern).
6. Паттерн [«фабричный метод»](https://en.wikipedia.org/wiki/Factory_method_pattern).
7. Паттерн [«стратегия»](https://en.wikipedia.org/wiki/Strategy_pattern).
8. Паттерн [«состояние»](https://en.wikipedia.org/wiki/State_pattern).

## Задачи на разработку

Программы должны проходить все тесты. Код должен проходить проверки go vet и golint.

### Базовая задача

Создать программу печатающую точное время с использованием NTP -библиотеки. Инициализировать как go module. Использовать библиотеку github.com/beevik/ntp. Написать программу печатающую текущее время / точное время с использованием этой библиотеки.

Требования:

1. Программа должна быть оформлена как go module
2. Программа должна корректно обрабатывать ошибки библиотеки: выводить их в STDERR и возвращать ненулевой код выхода в OS

### Задача на распаковку

Создать Go-функцию, осуществляющую примитивную распаковку строки, содержащую повторяющиеся символы/руны, например:

* "a4bc2d5e" => "aaaabccddddde"
* "abcd" => "abcd"
* "45" => "" (некорректная строка)
* "" => ""

#### Дополнительно

Реализовать поддержку escape-последовательностей.

Например:

* qwe\4\5 => qwe45 (\*)
* qwe\45 => qwe44444 (\*)
* qwe\\5 => qwe\\\\\ (\*)

В случае если была передана некорректная строка, функция должна возвращать ошибку. Написать unit-тесты.

### Утилита sort

Отсортировать строки в файле по аналогии с консольной утилитой sort (man sort — смотрим описание и основные параметры): на входе подается файл из несортированными строками, на выходе — файл с отсортированными.

Реализовать поддержку утилитой следующих ключей:

-k — указание колонки для сортировки (слова в строке могут выступать в качестве колонок, по умолчанию разделитель — пробел)

-n — сортировать по числовому значению

-r — сортировать в обратном порядке

-u — не выводить повторяющиеся строки

#### Дополнительно

Реализовать поддержку утилитой следующих ключей:

-M — сортировать по названию месяца

-b — игнорировать хвостовые пробелы

-c — проверять отсортированы ли данные

-h — сортировать по числовому значению с учетом суффиксов

### Поиск анаграмм по словарю

Написать функцию поиска всех множеств анаграмм по словарю.

Например:

'пятак', 'пятка' и 'тяпка' - принадлежат одному множеству,

'листок', 'слиток' и 'столик' - другому.

Требования:

1. Входные данные для функции: ссылка на массив, каждый элемент которого - слово на русском языке в кодировке utf8
2. Выходные данные: ссылка на мапу множеств анаграмм
3. Ключ - первое встретившееся в словаре слово из множества. Значение - ссылка на массив, каждый элемент которого,

слово из множества.

1. Массив должен быть отсортирован по возрастанию.
2. Множества из одного элемента не должны попасть в результат.
3. Все слова должны быть приведены к нижнему регистру.
4. В результате каждое слово должно встречаться только один раз.

### Утилита grep

Реализовать утилиту фильтрации по аналогии с консольной утилитой (man grep — смотрим описание и основные параметры).

Реализовать поддержку утилитой следующих ключей:

-A - "after" печатать +N строк после совпадения

-B - "before" печатать +N строк до совпадения

-C - "context" (A+B) печатать ±N строк вокруг совпадения

-c - "count" (количество строк)

-i - "ignore-case" (игнорировать регистр)

-v - "invert" (вместо совпадения, исключать)

-F - "fixed", точное совпадение со строкой, не паттерн

-n - "line num", напечатать номер строки

### Утилита cut

Реализовать утилиту аналог консольной команды cut (man cut). Утилита должна принимать строки через STDIN, разбивать по разделителю (TAB) на колонки и выводить запрошенные.

Реализовать поддержку утилитой следующих ключей:

-f - "fields" - выбрать поля (колонки)

-d - "delimiter" - использовать другой разделитель

-s - "separated" - только строки с разделителем

### Or channel

Реализовать функцию, которая будет объединять один или более *done*-каналов в *single*-канал, если один из его составляющих каналов закроется.

Очевидным вариантом решения могло бы стать выражение при использованием *select*, которое бы реализовывало эту связь, однако иногда неизвестно общее число *done*-каналов, с которыми вы работаете в рантайме. В этом случае удобнее использовать вызов единственной функции, которая, приняв на вход один или более *or*-каналов, реализовывала бы весь функционал.

Определение функции:

var or func(channels ...<- chan interface{}) <- chan interface{}

Пример использования функции:

sig := func(after time.Duration) <- chan interface{} {

c := make(chan interface{})

go func() {

defer close(c)

time.Sleep(after)

}()

return c

}

start := time.Now()

<-or (

sig(2\*time.Hour),

sig(5\*time.Minute),

sig(1\*time.Second),

sig(1\*time.Hour),

sig(1\*time.Minute),

)

fmt.Printf(“fone after %v”, time.Since(start))

### Взаимодействие с ОС

Необходимо реализовать свой собственный UNIX-шелл-утилиту с поддержкой ряда простейших команд:

- cd <args> - смена директории (в качестве аргумента могут быть то-то и то)

- pwd - показать путь до текущего каталога

- echo <args> - вывод аргумента в STDOUT

- kill <args> - "убить" процесс, переданный в качесте аргумента (пример: такой-то пример)

- ps - выводит общую информацию по запущенным процессам в формате \*такой-то формат\*

Так же требуется поддерживать функционал fork/exec-команд

Дополнительно необходимо поддерживать конвейер на пайпах (linux pipes, пример cmd1 | cmd2 | .... | cmdN).

\*Шелл — это обычная консольная программа, которая будучи запущенной, в интерактивном сеансе выводит некое приглашение

в STDOUT и ожидает ввода пользователя через STDIN. Дождавшись ввода, обрабатывает команду согласно своей логике

и при необходимости выводит результат на экран. Интерактивный сеанс поддерживается до тех пор, пока не будет введена команда выхода (например \quit).

### Утилита wget

Реализовать утилиту wget с возможностью скачивать сайты целиком.

### Утилита telnet

Реализовать простейший telnet-клиент.

Примеры вызовов:

go-telnet --timeout=10s host port go-telnet mysite.ru 8080 go-telnet --timeout=3s 1.1.1.1 123

Требования:

1. Программа должна подключаться к указанному хосту (ip или доменное имя + порт) по протоколу TCP. После подключения STDIN программы должен записываться в сокет, а данные полученные и сокета должны выводиться в STDOUT
2. Опционально в программу можно передать таймаут на подключение к серверу (через аргумент --timeout, по умолчанию 10s)
3. При нажатии Ctrl+D программа должна закрывать сокет и завершаться. Если сокет закрывается со стороны сервера, программа должна также завершаться. При подключении к несуществующему сервер, программа должна завершаться через timeout

### HTTP-сервер

Реализовать HTTP-сервер для работы с календарем. В рамках задания необходимо работать строго со стандартной HTTP-библиотекой.

В рамках задания необходимо:

1. Реализовать вспомогательные функции для сериализации объектов доменной области в JSON.
2. Реализовать вспомогательные функции для парсинга и валидации параметров методов /create\_event и /update\_event.
3. Реализовать HTTP обработчики для каждого из методов API, используя вспомогательные функции и объекты доменной области.
4. Реализовать middleware для логирования запросов

Методы API:

* POST /create\_event
* POST /update\_event
* POST /delete\_event
* GET /events\_for\_day
* GET /events\_for\_week
* GET /events\_for\_month

Параметры передаются в виде www-url-form-encoded (т.е. обычные user\_id=3&date=2019-09-09). В GET методах параметры передаются через queryString, в POST через тело запроса.

В результате каждого запроса должен возвращаться JSON-документ содержащий либо {"result": "..."} в случае успешного выполнения метода, либо {"error": "..."} в случае ошибки бизнес-логики.

В рамках задачи необходимо:

1. Реализовать все методы.
2. Бизнес логика НЕ должна зависеть от кода HTTP сервера.
3. В случае ошибки бизнес-логики сервер должен возвращать HTTP 503. В случае ошибки входных данных (невалидный int например) сервер должен возвращать HTTP 400. В случае остальных ошибок сервер должен возвращать HTTP 500. Web-сервер должен запускаться на порту указанном в конфиге и выводить в лог каждый обработанный запрос.

## Чтение и понимание кода

1. Что выведет программа? Объяснить вывод программы.

package main

import (

"fmt"

)

func main() {

a := [5]int{76, 77, 78, 79, 80}

var b []int = a[1:4]

fmt.Println(b)

}

1. Что выведет программа? Объяснить вывод программы. Объяснить как работают defer’ы и порядок их вызовов.

package main

import (

"fmt"

)

func test() (x int) {

defer func() {

x++

}()

x = 1

return

}

func anotherTest() int {

var x int

defer func() {

x++

}()

x = 1

return x

}

func main() {

fmt.Println(test())

fmt.Println(anotherTest())

}

1. Что выведет программа? Объяснить вывод программы. Объяснить внутреннее устройство интерфейсов и их отличие от пустых интерфейсов.

package main

import (

"fmt"

"os"

)

func Foo() error {

var err \*os.PathError = nil

return err

}

func main() {

err := Foo()

fmt.Println(err)

fmt.Println(err == nil)

}

1. Что выведет программа? Объяснить вывод программы.

package main

func main() {

ch := make(chan int)

go func() {

for i := 0; i < 10; i++ {

ch <- i

}

}()

for n := range ch {

println(n)

}

}

1. Что выведет программа? Объяснить вывод программы.

package main

type customError struct {

msg string

}

func (e \*customError) Error() string {

return e.msg

}

func test() \*customError {

{

// do something

}

return nil

}

func main() {

var err error

err = test()

if err != nil {

println("error")

return

}

println("ok")

}

1. Что выведет программа? Объяснить вывод программы. Рассказать про внутреннее устройство слайсов и что происходит при передаче их в качестве аргументов функции.

package main

import (

"fmt"

)

func main() {

var s = []string{"1", "2", "3"}

modifySlice(s)

fmt.Println(s)

}

func modifySlice(i []string) {

i[0] = "3"

i = append(i, "4")

i[1] = "5"

i = append(i, "6")

}

1. Что выведет программа? Объяснить вывод программы.

package main

import (

"fmt"

"math/rand"

"time"

)

func asChan(vs ...int) <-chan int {

c := make(chan int)

go func() {

for \_, v := range vs {

c <- v

time.Sleep(time.Duration(rand.Intn(1000)) \* time.Millisecond)

}

close(c)

}()

return c

}

func merge(a, b <-chan int) <-chan int {

c := make(chan int)

go func() {

for {

select {

case v := <-a:

c <- v

case v := <-b:

c <- v

}

}

}()

return c

}

func main() {

a := asChan(1, 3, 5, 7)

b := asChan(2, 4 ,6, 8)

c := merge(a, b )

for v := range c {

fmt.Println(v)

}

}