GO-06 03: Table driven test vs closure driven tests

Описание:

В прошлом задании мы с вами научились писать простые Unit-тесты. Сегодня мы познакомимся с подходами к написанию этих самых тестов, а именно с Table test и Closure func test.
Table tests
Давайте взглянем на тесты, которые мы написали в прошлом задании, они выглядели примерно так.

```
func TestReverseIntPositiveValue(t *testing.T) {
    // tests...
}

func TestReverseIntNegativeValue(t *testing.T) {
    // tests...
}

func TestReverseIntZeroValue(t *testing.T) {
    // tests...
}

// etc...
```

Тут сразу бросается в глаза один антипаттерн. Мы пишем по тест-функции на каждый отдельный случай. В чем здесь антипаттерн? Давайте посмотрим на следующий пример. Представим, что в наших utils не одна функция, а, например, 10, тогда наши тесты либо начинают разрастаться до таких размеров, что в них становится сложно ориентироваться и читать,

```
func TestFirstFuncFirstValue(t *testing.T) {
    // tests...
}

func TestFirstFuncSecondValue(t *testing.T) {
    // tests...
}
```

```go
func TestFirstFuncInvalidValue(t *testing.T) {
    // tests...
}

func TestSecondFuncFirstValue(t *testing.T) {
    // tests...
}

func TestSecondFuncSecondValue(t *testing.T) {
    // tests...
}

func TestSecondFuncInvalidValue(t *testing.T) {
    // tests...
}

// и еще, как минимум, 8 тест-функций (но на самом деле больше)
```

либо мы начинаем создавать большое количество тестовых файлов, тем самым становится сложно ориентироваться уже не в коде, а в самом проекте.

```
|- projects
|---- main.go
|---- utils.go
|---- utils_first_func_test.go
|---- utils_second_func_test.go
|---- utils_thrid_func_test.go

// etc...
```

Оба варианта приводят нас к плохой архитектуре и повышению сложности разработки нашего ПО. А ведь тесты должны упрощать разработку, а не усложнять.
Для того чтобы упростить ситуацию, которая может сложиться, нам надо изолировать все тестовые случаи в рамках одной функции, и сделать это нам поможет подход Table tests.
Давайте рассмотрим пример:

```go
func TestSum(t *testing.T) {
    tests := map[string]struct{
        firstVal  int
        secondVal int
        want      int
    }{
```

```
        "simple":          {firstVal: 1, secondVal: 1, want: 2},
        "zero values":     {firstVal: 0, secondVal: 0, want: 0},
        "negative values": {firstVal: -10, secondVal: -2, want: -12},
    }

    for name, testCase := range tests {
        t.Run(name, func(t *testing.T) {
            res := sum(testCase.firstVal, testCase.secondVal)
            req.Equal(testCase.want, res)
        })
    }
}
```


В этом примере мы в начале теста определяем все тестовые случаи, которые нам нужны, а дальше через цикл проверяем каждый из них. Таким образом мы изолируем все тестовые случаи в рамках одной функции, и наш тест превращается в нечто более целостное, чем было раньше, наша архитектура не страдает, а читать такой тест намного приятнее.

Closure func tests

Но что делать, когда наши тестовые случаи (кейсы) не просты или не однообразны? Функция sum является очень простой, в ней нет сложной логики, также как и вокруг нее. Давайте представим немного другую ситуацию.

Имеем функцию initSessions, которая в зависимости от обстоятельств ведет себя по-разному:

1. Если пользователь заходит в наш сервис в первый раз, тогда честно создается новая сессия.
2. Если же пользователь переподключается по какой-либо причине (например, у него упал интернет), тогда функция идет в хранилище сервиса, достает информацию о сессии пользователя и делает операцию восстановления.

Причем работать функция может с различным хранилищами (обычная мапа / redis / postgres и т.д., главное, чтобы хранилище реализовывало конкретный интерфейс).

```
type User struct {
    SessionID string
    Name      string
    Pass      string
}

type Session struct {
    ID     string
    UserID int
}
```

```go
type Storage interface {
    RecoverSession(id string) (*Session, error)
    CreateSession(name string, pass string) (*Session, error)
}

func InitSession(user User, storage Storage) (*Session, error) {
    session, err := storage.RecoverSession(user.SessionID)
    if err != nil {
        return nil, err
    }

    if session == nil {
        return storage.CreateSession(user.Name, user.Pass)
    }

    return session, nil
}
```

Также важно отметить, что на момент создания сессии, естественно, SessionID у нас нет, тогда как на этапе восстановления он уже присутствует.
В таком случае просто табличных тестов может попросту не хватить, так как:

1. В одном тесте мы захотим проверить, что будет, если RecoverSession выдаст ошибку.
2. В следующем кейсе мы захотим проверить создание сессии (то есть, если ошибки из RecoverSession нет, но значение session == nil).
3. В третьем варианте мы захотим проверить, что функция CreateSession выдала ошибку.

И так далее. Мы просто не сможем грамотно и понятно настроить все зависимости (правильно их замокать) для всех тестовых случаев. Вот здесь нам приходит на помощь следующий паттерн - Closure func test.
Заключается этот паттерн в том, чтобы внутри теста изолировать не только тестовые данные, но также и проверки, моки и различную логику тест-кейса внутри вызова подфункции t.Run("test name", func(t *testing.T) {}). Для примера выше это может выглядеть так:

```go
func TestInitSession(t *testing.T) {
    any := mocks.Any()

    mockCtrl := gomock.NewController(t)
    defer mockCtrl.Finish()
```

```
    storageMock := mocks.NewStorageMock(mockCtrl)

    t.Run("create session success", func(t *testing.T) {
        storageMock.EXPECT().RecoverSession(any).Return(nil,
nil).Times(1)
        storageMock.EXPECT().CreateSession(any,
any).Return(&Session{ID: "session_id", UserID: 1}, nil).Times(1)

        session, err := InitSession(User{Name: "ivan", Pass:
"secret"}, storageMock)
        req.NoError(err)
        req.Equal("session_id", session.ID)
    })

    t.Run("create session fail", func(t *testing.T) {
        storageMock.EXPECT().RecoverSession(any).Return(nil,
nil).Times(1)
        storageMock.EXPECT().CreateSession(any, any).Return(nil,
errors.New("fail to create session")).Times(1)

        _, err := InitSession(User{Name: "ivan", Pass: "secret"},
storageMock)
        req.Error(err)
    })

    t.Run("recover session success", func(t *testing.T) {
        storageMock.EXPECT().RecoverSession(any).Return(&Session{ID:
"session_id", UserID: 1}, nil).Times(1)

        session, err := InitSession(User{SessionID: "session_id",
Name: "ivan", Pass: "secret"}, storageMock)
        req.NoError(err)
        req.Equal("session_id", session.ID)
    })
}
```

Внутри каждого теста имеется различная логика, которая не позволяет нам использовать здесь табличный подход. Однако внутри каждого из тестов мы можем использовать дополнительно и табличный метод, чтобы проверить тест с различными данными на входе, таким образом объединяя два этих подхода.

Если же все кейсы подчиняются одинаковой логике тестирования, тогда их можно написать в стиле Closure func следующим образом. Выделим отдельно case как функцию, которая принимает параметры для конкретного тест-кейса, а на выходе отдает функцию проверки с переданными параметрами.

```
func TestSum(t *testing.T) {
    sumCase := func(a int, b int, want int) func(t *testing.T) {
        return func(t *testing.T) {
            res := sum(a, b)
            req.Equal(want, res)
        }
    }

    t.Run("simple case", sumCase(1, 1, 2))
    t.Run("zero values", sumCase(0, 0, 0))
    t.Run("negative values", sumCase(-1, -1, -2))
}
```

Полезные ссылки:

- Closure driven tests: an alternative style to table driven tests in go
- Table driver tests golang
- Advanced tests go

Задание:

1. В этом задании мы будем работать со следующими двумя функциями из пакета util каталога ./internal/pkg/util:
   - Функция ContainsDuplicate, как понятно из названия, проверяет массив на наличие дубликатов.
   - Функция IsPalindrome проверяет, является ли число палиндромом. Число является палиндромом, когда оно читается в обе стороны одинаково. Например, 101, 202, 111 и т.д.
2. Вам нужно написать для этих функций тесты:
   - Для функции ContainsDuplicate тесты должны быть написаны в Table driven подходе.
   - Для функции IsPalindrome тесты должны быть написаны в Closure driven подходе.

Порядок действий:

1. В вашем проекте module06 делаем новую ветку module06_03.

2. В файле util_test.go из прошлого задания пишем две новые тест-функции:
    - `TestContainsDuplicate` и заполняем ее в Table driven подходе.
    - `TestIsPalindrome` и заполняем ее в Closure driven подходе.
3. В качестве ответа пришлите ссылку на merge request в ветку master вашего проекта ветки module06_03.