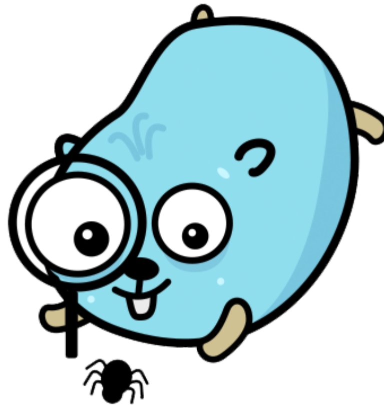
Работа с ошибками в тестах (часть 3)

Продолжаем тему работы с ошибками при тестировании в Go.



Не игнорируйте ошибки в тестах

Тесты – это тоже код. И требования к ним не должны отличаться от требований к коду. (с) Джейсон Стейтем.

В плане обработки ошибок тесты ничем не отличаются от обычных программ на Go, в которых мы должны их обрабатывать:

```
f, err := os.Open(name)
if err != nil {
    // ...

}
```

А игнорирование возможной ошибки ставит под сомнение использование других результатов функции:

```
f, _ := os.Open(name)

codeUsing(f) // Danger!
```

В тестах та же история – игнорирование промежуточных ошибок может привести к тому, что

- будет затруднён debug истинной причины падения теста;
- тест может пройти при существующих промежуточных ошибках и тогда мы не заметим потенциальной проблемы в тестируемом коде или в самом тесте (например, тест составлен неверно или тестирует не то, что ожидалось).

Посмотрим на видоизменённый пример теста из open source проекта:

```
import (
    "bytes"
    "regexp"
    "testing"

    "github.com/stretchr/testify/require"
)

func TestEnum_StringThing(t *testing.T) {
    specDoc, _ :=
loads.Spec("../fixtures/codegen/todolist.enums.yml") // <-- 1

    definitions := specDoc.Spec().Definitions
    const k = "StringThing"
    schema := definitions[k]
    opts := opts()
    genModel, _ := makeGenDefinition(k, "models", schema, specDoc,
opts) // <-- 2

    buf := bytes.NewBuffer(nil)
    templates.MustGet("model").Execute(buf, genModel) // <-- 3

    ff, _ := opts.LanguageOpts.FormatContent("string_thing.go",
buf.Bytes()) // <-- 4

    res := string(ff)
    require.Contains(t, res, "var stringThingEnum []interface{}")
    require.Contains(t, res, k+") validateStringThingEnum(path,
location string, value StringThing)")
```

```
    require.Contains(t, res, "m.validateStringThingEnum(\"\",
\"body\", m)")
}
```

Итого 4 места, где может произойти ошибка. В лучшем случае при выполнении теста произойдёт паника и прервёт его, что подскажет нам, где искать проблему. В худшем случае мы вывалимся только на последних проверках и тогда придётся гадать, что произошло:

- не загрузилась спецификация? (1)
- не создалась модель? (2)
- произошла ошибка при заполнении шаблона? (3)
- произошла ошибка форматирования конечного файла? (4)

Перепишем этот пример, добавив проверку ошибок (обратите внимание на микс **require** и **assert**):

```
import (
    // ...
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

func TestEnum_StringThing(t *testing.T) {
    specDoc, err :=
loads.Spec("../fixtures/codegen/todolist.enums.yml")
    require.NoError(t, err)

    definitions := specDoc.Spec().Definitions
    const k = "StringThing"
    schema := definitions[k]
    opts := opts()
    genModel, err := makeGenDefinition(k, "models", schema, specDoc,
opts)
    require.NoError(t, err)

    buf := bytes.NewBuffer(nil)
    err = templates.MustGet("model").Execute(buf, genModel)
    require.NoError(t, err)
```

```
    ff, err := opts.LanguageOpts.FormatContent("string_thing.go", 
buf.Bytes())
    require.NoError(t, err)

    res := string(ff)
    assert.Contains(t, res, "var stringThingEnum []interface{}")
    assert.Contains(t, res, k+") validateStringThingEnum(path, 
location string, value StringThing)")
    assert.Contains(t, res, `m.validateStringThingEnum("", "body", 
m)`)

}
```

Совсем другое дело :)

Основой для примеров выше является один из тестов в go-swagger/generator/enum_test.go.

go-swagger – это популярный модуль, позволяющий кодогенерировать HTTP API по его описанию в спецификации swagger.

Тест "Выберите безопасные с точки зрения работы с данными варианты"

Выберите все подходящие ответы из списка

```
client, err := usersservice.NewClient(url)
require.NoError(t, err)
users, _ := client.GetUsers()
require.Equal(t, "test-id", users[0].ID)

u, err := client.GetUser()
require.NoError(t, err)
assert.Equal(t, "test-id", u.ID)
assert.Equal(t, "test-email@g.com", u.Email)

u, _ := client.GetUser()
assert.Equal(t, "test-id", u.ID)
```

```
  assert.Equal(t, "test-email@g.com", u.Email)

  client, err := usersservice.NewClient(url)
  require.NoError(t, err)
  users, err := client.GetUsers()
  require.NoError(t, err)
  require.Equal(t, "test-id", users[0].ID)

  u, err := client.GetUser()
  require.Equal(t, "test-id", u.ID)
  require.Equal(t, "test-email@g.com", u.Email)
  require.NoError(t, err)

  client, err := usersservice.NewClient(url)
  users, err := client.GetUsers()
  require.NoError(t, err)
  require.GreaterOrEqual(t, 1, len(users))
assert.Equal(t, "test-id", users[0].ID)

  client, err := usersservice.NewClient(url)
  require.NoError(t, err)
  users, err := client.GetUsers()
  require.NoError(t, err)
  require.GreaterOrEqual(t, 1, len(users))
assert.Equal(t, "test-id", users[0].ID)
```

Не игнорируйте ошибки в тестах: отложенные операции

На примере того же модуля разработчики часто игнорируют не только промежуточные ошибки в тестах, но и ошибки в "clean up" функциях, обычно запускаемых через `defer`:

```
// ...

tgt, _ := ioutil.TempDir(cwd, "dumped")
defer func() {
```

```go
    _ = os.RemoveAll(tgt)
}()

opts.DumpData = true
err = GenerateClient("test", []string{}, []string{}, opts)
require.NoError(t, err)


// ...
```

Наиболее популярными функциями "очистки ресурсов" являются:

- os.Remove, os.RemoveAll
- все типы, реализующие io.Closer (*os.File, *sql.DB и т.д.)

Ошибки от них часто игнорируют не только в тестах, но и в прикладном коде, так как они обычно являются низкоуровневыми и случаются крайне редко (если не никогда).

В предыдущих модулях мы рассматривали средства работы с подобными ошибками (различные `multierr.AppendInvoke`; избегание `defer` в пользу явного возврата ошибки от операции; логирование и т.д.)

В тестах же мы рекомендуем все-таки assert'ить данные ошибки. Исправим первоначальный пример:

```go
// ...

tgt, err := ioutil.TempDir(cwd, "dumped")
require.NoError(t, err)
defer func() {
    require.NoError(t, os.RemoveAll(tgt))
}()

opts.DumpData = true
err = GenerateClient("test", []string{}, []string{}, opts)
require.NoError(t, err)


// ...
```

Это поможет нам вовремя узнать о потенциальных проблемах с кодом или с операционной системой, на которой он запускается, а также в целом не оставлять "мусор" после себя.

Тест "defer require"

Верно ли писать следующим образом?

```
defer require.NoError(t, os.RemoveAll(tgt))
```

Выберите один вариант из списка

Да, данный код означает "отложи в конец текущей функции удаление директории tgt и проверь ошибку от этой операции"
Нет, данный код означает "удали сейчас директорию tgt и отложи проверку ошибки от этой операции в конец текущей функции".

Ошибки и моки

Имитация в тестах возвращаемой ошибки ничем не отличается от имитации любого другого действия, ожидаемого от тестируемого объекта.

При этом если прикладной код работает с конкретной ошибкой / конкретным типом ошибки, то мок функции без проблем может возвращать переменную нужного типа (если тип экспортируемый). Если же прикладной код работает с поведением ошибки, как например, было в задаче "ParseToken для безопасников"

```
var emailer interface {
    Email() string
}
if !errors.As(err, &emailer) {
    // ...

}
```

то мы можем создать собственный тип, удовлетворяющий нужным интерфейсам, и с помощью него менять поведение программы в тестах.

Если нам не нужно тестировать обработку различных ошибок, возвращаемых функцией, а достаточен просто факт наличия ошибки, то наш мок может возвращать какую-нибудь "dummy" ошибку:

```
// ...
testErr := errors.New("test error") // Просто "какая-то ошибка".
dbMock.EXPECT().GetUserByID(user1.ID).Return(nil, testErr)

u, err := usersservice.Find(user1.ID)
require.ErrorIs(t, err, testErr)

require.Nil(t, u)
```

При этом хорошим тоном считается держать в порядке и "dummy" ошибки: делать понятными их названия и значения.

Ведь тесты – это тоже код и его читаемость никто не отменял:

```
var (
    errGetUserByID = errors.New("cannot get user by ID")
    // ...
)

// ...
dbMock.EXPECT().GetUserByID(user1.ID).Return(nil, errGetUserByID)

u, err := usersservice.Find(user1.ID)
require.ErrorIs(t, err, errGetUserByID)

require.Nil(t, u)
```

Задача "Имитация сетевой ошибки"

Ссылка на заготовку.

Перед вами функция `FetchURL`, основной задачей которой (как ни странно) является получение данных веб-страницы по её URL:

```
package fetcher

// ...
var (
    ErrFetchTimeout = errors.New("fetch url timeout")
    ErrFetchTmp     = errors.New("fetch url temporary error")
```

```go
)

type Client interface {
    Do(req *http.Request) (*http.Response, error)
}

func FetchURL(ctx context.Context, client Client, url string)
([]byte, error) {
    // ...
    resp, err := client.Do(req)
    if err != nil {
        if isTimeout(err) {
            return nil, fmt.Errorf("%w: %v", ErrFetchTimeout, err)
        }
        if isTemporary(err) {
            return nil, fmt.Errorf("%w: %v", ErrFetchTmp, err)
        }
        return nil, fmt.Errorf("do request: %w", err)
    }
    // ...
}

func isTimeout(err error) bool {
    var i interface{ Timeout() bool }
    return errors.As(err, &i) && i.Timeout()
}

func isTemporary(err error) bool {
    var i interface{ Temporary() bool }
    return errors.As(err, &i) && i.Temporary()
}
```

Как мы видим, разработчик захотел предоставить наружу API для определения, произошла ли "временная ошибка" или "ошибка таймаута":

```go
var (
    ErrFetchTimeout = errors.New("fetch url timeout")
    ErrFetchTmp     = errors.New("fetch url temporary error")

)
```

При этом он не стал завязываться конкретно на *url.Error, которую возвращает (*http.Client).Do согласно документации, и на ошибки, которые `*url.Error` оборачивает, зная, что их может быть очень большое количество (от пакетов `os` и `net`):

```go
// Package url parses URLs and implements query escaping.
package url

// ...

// Error reports an error and the operation and URL that caused it.
type Error struct {
    Op  string
    URL string
    Err error
}

func (e *Error) Unwrap() error { return e.Err }
func (e *Error) Error() string { return fmt.Sprintf("%s %q: %s", e.Op, e.URL, e.Err) }

func (e *Error) Timeout() bool {
    t, ok := e.Err.(interface {
        Timeout() bool
    })
    return ok && t.Timeout()
}

func (e *Error) Temporary() bool {
    t, ok := e.Err.(interface {
        Temporary() bool
    })
    return ok && t.Temporary()
}


// ...
```

Поэтому разработчик завязался на ***поведение*** ошибки выше, а именно – на методы `Timeout` и `Temporary`.

Для тестирования функции были написаны следующие кейсы:

```go
// ...
   cases := []struct {
        client       clientMock
        expectedData []byte
        expectedErr  error
    }{
        {
            client:       newClientMock(nil, newNetErrMock(false,
true)),
            expectedErr:  ErrFetchTimeout,
        },
        {
            client:       newClientMock(nil, newNetErrMock(true,
false)),
            expectedErr:  ErrFetchTmp,
        },
        {
            client:       newClientMock(nil, errUnknownNetErr),
            expectedErr:  errUnknownNetErr,
        },
        {
            client: newClientMock(&http.Response{
                Body:
ioutil.NopCloser(bytes.NewReader([]byte("hello"))),
            }, nil),
            expectedData: []byte("hello"),
        },
    }

// ...
```

Где мок клиента представляет собой тривиальный тип вида

```go
type clientMock struct {
    doResponse *http.Response
    doErr      error
}

func newClientMock(resp *http.Response, err error) clientMock {
    return clientMock{resp, err}
}

func (c clientMock) Do(_ *http.Request) (*http.Response, error) {
    return c.doResponse, c.doErr
```

```
}
```

Вам требуется реализовать тип `netErrMock` и конструктор для него `newNetErrMock` таким образом, чтобы тесты выше проходили.

Выводы

- не игнорируйте ошибки в тестах – как в обычном, так и в отложенном (**deferred**) коде;
- мокайте ошибки, возвращаемые методами, чтобы протестировать специфичные ветви вашего кода;
- получайте удовольствие от жизни и будьте счастливы.

Делитесь своими лучшими практиками в комментариях :)