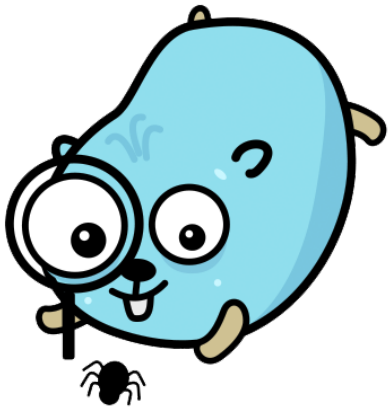
Работа с ошибками в тестах (часть 2)

В этом уроке мы ещё раз коснёмся работы с ошибками в тестах и закрепим полученные ранее знания.



Опасный reflect.DeepEqual

Вспомним кусочек кода из предыдущего урока

```
if !errors.Is(err, tt.expectedErr) {
    t.Fatalf("want error %v, got %v", tt.expectedErr, err)
}

if !reflect.DeepEqual(token, tt.expectedToken) {
    t.Fatalf("got token %#v, want: %#v", token, tt.expectedToken)

}
```

Возникает вопрос, почему бы и ошибку не проверять через `reflect.DeepEqual`?

```
if !reflect.DeepEqual(err, tt.expectedErr) {
    t.Fatalf("want error %v, got %v", tt.expectedErr, err)

}
```

На самом деле это считается плохим тоном по причине [реализации](#) `DeepEqual`:

```
package reflect
```

```
// DeepEqual reports whether x and y are ``deeply equal,'' defined as
follows.
// Two values of identical type are deeply equal if one of the
following cases applies.
// Values of distinct types are never deeply equal.
//
// ...
//
// Interface values are deeply equal if they hold deeply equal
concrete values.
//
// ...

func DeepEqual(x, y interface{}) bool
```

Из предыдущих уроков мы знаем, что ошибка – это любой тип, реализующий интерфейс `error`. Как видно из описания работы `DeepEqual`, два интерфейса равны, если равны значения, которые они хранят. И действительно

```
fmt.Println(reflect.DeepEqual(io.EOF, io.EOF))               // true
fmt.Println(reflect.DeepEqual(io.EOF, io.ErrUnexpectedEOF)) // false

fmt.Println(reflect.DeepEqual(io.EOF, nil))                  // false
```

Всё так, как и ожидалось. А давайте создадим свою ошибку, но с аналогичным `io.EOF` текстом:

```
var MyEOF = errors.New(io.EOF.Error())

fmt.Println(reflect.DeepEqual(io.EOF, MyEOF)) // true
```

Аналогично

```
lhs := fmt.Errorf("some error:  %w", io.EOF)
rhs := fmt.Errorf("some error:  %w", MyEOF)
fmt.Println(reflect.DeepEqual(lhs, rhs)) // true
```

Мы видим, что `DeepEqual` считает равными две ошибки, которые не должны быть равны в принципе (больше примеров).

Какие это влечёт за собой проблемы?

1) Функция может возвращать абсолютно не ту ошибку, что ожидается (например, из другого пакета), и мы не заметим этого.

2) Подобное свойство `reflect.DeepEqual` позволяет писать людям [плохие тесты](#) вида

```
want := errors.New("archive/tar: header field too long or contains invalid values")
if !reflect.DeepEqual(got, want) {
    // ...

}
```

Мало того, что тест полагается на конкретное сообщение, так ещё – и на неэкспортируемый тестируемым пакетом тип!

Как показывает [нехитрый поиск среди open-source проектов](#), подобная практика имеет место быть.

Да и что говорить, если стандартная библиотека Go пестрит аналогичными примерами, сбивая с толку людей, только начинающих своё знакомство с концепцией ошибок в языке (укороченный вывод):

```
// https://github.com/golang/go master
$ grep -r "reflect.DeepEqual" . | grep -i err
./src/strconv/atof_test.go:             if outs != test.out ||
!reflect.DeepEqual(err, test.err) {
./src/net/ipsock_test.go:               if !reflect.DeepEqual(err,
tt.err) {
./src/net/mac_test.go:                  if !reflect.DeepEqual(out,
tt.out) || !match(err, tt.err) {
./src/net/tcpsock_test.go:              if !reflect.DeepEqual(addr,
tt.addr) || !reflect.DeepEqual(err, tt.err) {
./src/net/iprawsock_test.go:            if !reflect.DeepEqual(addr,
tt.addr) || !reflect.DeepEqual(err, tt.err) {
./src/net/textproto/reader_test.go:     if !reflect.DeepEqual(s, want)
|| err != nil {
./src/net/http/transfer_test.go:        if !reflect.DeepEqual(gotErr,
tt.wantErr) {
./src/net/http/request_test.go:         if !reflect.DeepEqual(err,
firstErr) {
./src/net/ip_test.go:                   if err :=
out.UnmarshalText([]byte(tt.in)); !reflect.DeepEqual(out, tt.out) ||
(tt.out == nil) != (err != nil) {
./src/net/udpsock_test.go:              if !reflect.DeepEqual(addr2,
tt.addr) || err != tt.err {
./src/net/dnsclient_unix_test.go:       if !reflect.DeepEqual(err,
wantErr) {
./src/go/build/build_test.go:           if shouldBuild !=
tt.shouldBuild || binaryOnly != tt.binaryOnly ||
!reflect.DeepEqual(tags, tt.tags) || err != tt.err {
./src/encoding/asn1/asn1_test.go:       if err == nil &&
!reflect.DeepEqual(test.out, tagAndLength) {
./src/encoding/base64/base64_test.go:   if !reflect.DeepEqual(want,
err) {
./src/encoding/xml/read_test.go:        if !reflect.DeepEqual(err,
tt.e) {
./src/encoding/json/decode_test.go:     if !reflect.DeepEqual(err,
tt.err) {
./src/encoding/json/stream_test.go:     if err == nil ||
!reflect.DeepEqual(err, experr) {
./src/encoding/json/scanner_test.go:    if !reflect.DeepEqual(err,
tt.err) {
./src/encoding/csv/reader_test.go:      if !reflect.DeepEqual(err,
tt.Error) {
```

deepequalerrors

При этом в репозитории golang/tools (содержащем различные полезные пакеты, используемые в компиляторе golang/go) можно найти статический анализатор deepequalerrors, который как раз проверяет, что вы не используете `reflect.DeepEqual` для проверок ошибок.

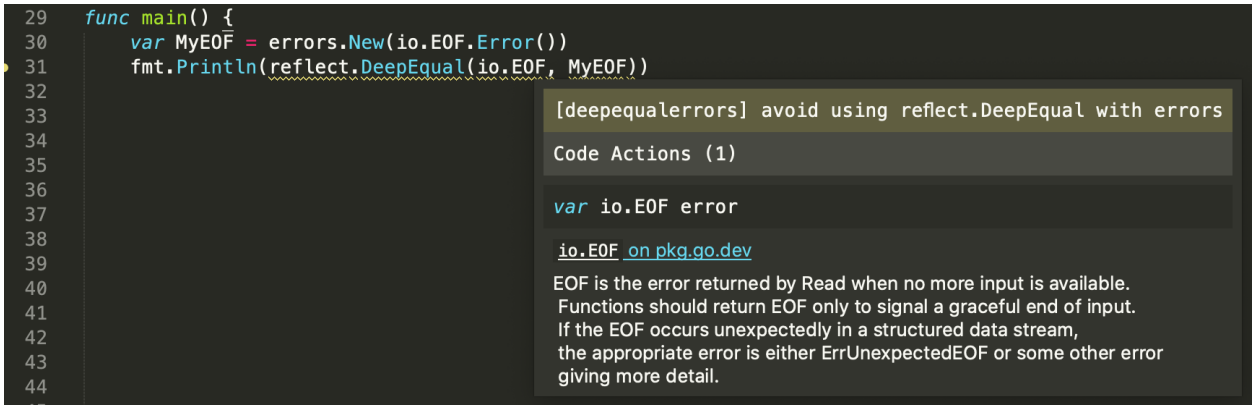
В **go vet** данный анализатор не включен по причине, которую называет сам Роб Пайк:

```
This is not a correctness issue, so it is not a vet problem.

Perhaps golint should do this, but not vet.
```

А в golang/lint он не включен по причине того, что подобных рекомендаций нет в CodeReviewComments :(

Но если в вашу IDE интегрирован gopls (официальный Language Server для языка Go, входящий в **golang/tools**), тогда вы уже наверняка встречали соответствующую инспецию:



Тест "Выбери братишку"

Выберите один вариант из списка

- [ ] err.Error() == tt.expectedErr.Error()
- [ ] err == tt.expectedErr
- [ ] reflect.DeepEqual(err, tt.expectedErr)
- [ ] errors.Is(err, tt.expectedErr)

github.com/stretchr/testify

Стандартная библиотека для тестирования в Go достаточно мощная, но хочется в целом писать поменьше кода и некоторых полезностей в ней не хватает.

На 2021 год github.com/stretchr/testify занимает первое место по популярности среди нестандартных библиотек для тестирования. Авторы курса активно используют этот модуль в ежедневной разработке.

Относительно тестирования ошибок обратим внимание на следующие функции:

```
import "github.com/stretchr/testify/require" // Аналогично для testify/assert.

// ...

require.Nil(t, err)    // Проверить, что ошибки нет.
require.NotNil(t, err) // Проверить, что ошибка есть.

require.NoError(t, err) // Проверить, что ошибки нет, но с более понятным текстом assert'a.
require.Error(t, err)   // Проверить, что ошибка есть, но с более понятным текстом assert'a.

require.ErrorIs(t, err, context.DeadlineExceeded) // Под капотом errors.Is.

var numErr strconv.NumError
require.ErrorAs(t, err, &numErr) // Под капотом errors.As.
```

Тесты из предыдущего урока теперь можно переписать следующим образом:

```
// Было.

t.Run(tt.name, func(t *testing.T) {
    token, err := ParseToken([]byte(tt.jwt), []byte("secret"))
    if !errors.Is(err, tt.expectedErr) {
        t.Fatalf("want error %v, got %v", tt.expectedErr, err)
    }

    if !reflect.DeepEqual(token, tt.expectedToken) {
        t.Fatalf("got token %#v, want: %#v", token, tt.expectedToken)
    }
})

// Стало.

t.Run(tt.name, func(t *testing.T) {
    token, err := ParseToken([]byte(tt.jwt), []byte("secret"))
    require.ErrorIs(t, err, tt.expectedErr)
    require.Equal(t, tt.expectedToken, token)

})
```

Обратите внимание на использование `require.ErrorAs`:

```
// Было.

t.Run(tt.name, func(t *testing.T) {
    token, err := ParseToken([]byte(tt.jwt), []byte("secret"))

    if !errors.Is(err, tt.expectedErr) {
        t.Fatalf("want error %q, got %q", tt.expectedErr, err)
    }

    if tt.expectedEmail != "" {
        var e interface {
            Email() string
        }
        if !errors.As(err, &e) {
```

```
            t.Fatalf("error %T doesn't implement `Email() string`
method", err)
        }

        if e.Email() != tt.expectedEmail {
            t.Fatalf("got email %#v, want: %#v", e.Email(),
tt.expectedEmail)
        }
    }

    if !reflect.DeepEqual(token, tt.expectedToken) {
        t.Fatalf("got token %#v, want: %#v", token, tt.expectedToken)
    }
})

// Стало.

t.Run(tt.name, func(t *testing.T) {
    token, err := ParseToken([]byte(tt.jwt), []byte("secret"))
    require.ErrorIs(t, err, tt.expectedErr)

    if tt.expectedEmail != "" {
        var e interface {
            Email() string
        }
        require.ErrorAs(t, err, &e)
        require.Equal(t, tt.expectedEmail, e.Email())
    }

    require.Equal(t, tt.expectedToken, token)
})
```

Кода стало заметно меньше!

На момент написания курса функции `require.ErrorIs` / `require.ErrorAs` являются относительно новыми и ещё не вошли в широкое применение.

Их аналог выглядит следующим образом

```
err := someOperation()
// Обратите внимание на сообщение об ошибке!
```

```
// Без него будет сложно понять, почему errors.Is вернул false.

require.True(t, errors.Is(err, context.DeadlineExceeded), "actual
err: %v", err)
```

Частые ошибки использования github.com/stretchr/testify

Исходник примеров.

1) Использование `require.Equal` для сравнения ошибок

```
func TestEqualErrors(t *testing.T) {
    var MyEOF = errors.New(io.EOF.Error())
    require.Equal(t, MyEOF, io.EOF) // Хотелось бы, чтобы тест не
прошёл.
}

/*
=== RUN   TestEqualErrors
--- PASS: TestEqualErrors (0.00s)
PASS

*/
```

Под капотом у `require.Equal` находится `reflect.DeepEqual` и мы получаем ситуацию, которую обсуждали в прошлом уроке.

2) Использование `require.Error` вместо `require.ErrorIs`

```
func TestErrorInsteadOfErrorIs(t *testing.T) {
    someOperation := func() error {
        return io.EOF
    }

    err := someOperation()
    require.Error(t, err, context.DeadlineExceeded) // Хотелось бы,
чтобы тест не прошёл.
}
```

```
/*
=== RUN   TestErrorInsteadOfErrorIs
--- PASS: TestErrorInsteadOfErrorIs (0.00s)
PASS

*/
```

Посмотрим на сигнатуру `require.Error`

```
// Error asserts that a function returned an error (i.e. not `nil`).
// ...

func Error(t TestingT, err error, msgAndArgs ...interface{})
```

и поймём, что в тесте в целом написано что-то странное: автор хотел проверить, что `err` разворачивается в `context.DeadlineExceeded`; а получил проверку того, что `err` не `nil`, а `context.DeadlineExceeded.Error()` попал в сообщение assert'a:

```
=== RUN   TestErrorInsteadOfErrorIs
    example_test.go:24:
            Error Trace:    example_test.go:24
            Error:          An error is expected but got nil.
            Test:           TestErrorInsteadOfErrorIs
            Messages:       context deadline exceeded

--- FAIL: TestErrorInsteadOfErrorIs (0.00s)
```

Исправить это несложно:

```
err := someOperation()

require.ErrorIs(t, err, context.DeadlineExceeded)
```

3) Неверный порядок аргументов в `require.ErrorIs`

```
func TestErrorIsInvalidOrder(t *testing.T) {
    errExpected := io.EOF
    err := fmt.Errorf("err: %w", io.EOF)
    require.ErrorIs(t, errExpected, err) // Хотелось бы, чтобы тест прошёл.
}
```

```
/*
=== RUN   TestErrorIsInvalidOrder
    example_test.go:32:
            Error Trace:    example_test.go:32
            Error:          Target error should be in err chain:
                            expected: "err: EOF"
                            in chain: "EOF"
            Test:           TestErrorIsInvalidOrder
--- FAIL: TestErrorIsInvalidOrder (0.00s)

*/
```

Когда часто пользуешься функциями из **stretchr/testify**, привыкаешь, что ожидаемое значение обычно идёт первым аргументом:

```
$ grep "expected interface{}"
$GOPATH/pkg/mod/github.com/stretchr/testify@v1.7.0/require/require.go
func Equal(t TestingT, expected interface{}, actual interface{},
msgAndArgs ...interface{}) {
func EqualValues(t TestingT, expected interface{}, actual
interface{}, msgAndArgs ...interface{}) {
func EqualValuesf(t TestingT, expected interface{}, actual
interface{}, msg string, args ...interface{}) {
func Equalf(t TestingT, expected interface{}, actual interface{}, msg
string, args ...interface{}) {
func Exactly(t TestingT, expected interface{}, actual interface{},
msgAndArgs ...interface{}) {
func Exactlyf(t TestingT, expected interface{}, actual interface{},
msg string, args ...interface{}) ...
```

Но в `require.ErrorsIs(As)` первым значением идёт цепочка, а ожидаемая ошибка – после:

```
func ErrorAs(t TestingT, err error, target interface{}, msgAndArgs
...interface{}) { /*...*/ }

func ErrorIs(t TestingT, err error, target error, msgAndArgs
...interface{}) { /*...*/ }
```

4) Использование `require.NoError` вместо `require.Error` (и наоборот)

Редкая и глупая ошибка, связанная с опечаткой или копипастой. Для маленьких тестов чревата тем, что тест всегда будет проходить и мы об этом не узнаем. В больших тестах скорее быстро отловится, так как тест бахнет где-нибудь дальше из-за использования невалидных данных.

Некоторым образом проблему нивелирует использование TDD.

Тест "stretchr/testify API"

Какой пакет предоставляет функции, которые вместо возврата логического результата завершают текущий тест?

Выберите один вариант из списка

- [ ] stretchr/testify/assert
- [ ] stretchr/testify/mock
- [ ] stretchr/testify/require
- [ ] stretchr/testify/suite

Связка require + assert

В примерах до этого мы пользовались только или **testify/assert** или **testify/require**.

Но существует негласное правило использовать **testify/require** для операций, которые **обязательно** должны отработать должным образом (самый популярный пример – `require.NoError`), а **testify/assert** для всех остальных проверок.

Какой профит?

Если пользоваться только **assert**, то тесты скорее всего будут падать с паниками, потому что на определённом шаге мы по сути будем игнорировать те же ошибки – что не совсем красиво и не всегда помогает быстро определить источник проблемы (исходник примера):

```
u, err := getUser()
assert.NoError(t, err)
assert.Equal(t, "user-id", u.ID) // Получим панику "... nil pointer dereference".
```

```
assert.Equal(t, "user-email", u.Email)

users, err := getUsers()
assert.NoError(t, err)
assert.Len(t, users, 3)

u := users[0] // Получим панику "... index out of range [0] ...".
assert.Equal(t, "user-id", u.ID)

assert.Equal(t, "user-email", u.Email)
```

Если пользоваться только **require**, то мы ~~внезапно~~ можем выйти "слишком рано", не захватив оставшиеся проверки – при большом их количестве бывает удобнее увидеть сразу все упавшие кейсы, чтобы разом их зафиксить:

```
u, err := getUser()
require.NoError(t, err)
require.Equal(t, "user-id", u.ID) // Выйдем, например, уже здесь, хотя удобнее увидеть сразу все ошибки.

require.Equal(t, "user-email", u.Email)
```

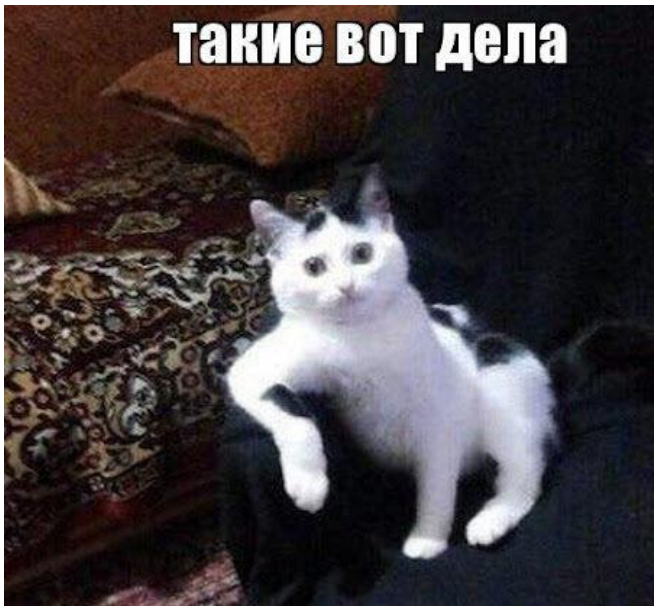
Резюме

Если вы замечаете в тестах микс из пакетов **testify** – не бегите это дело исправлять, быть может автор сделал так осознанно.

Чаще всего через **require** используют:

- `require.Error` / `require.NoError`
- `require.Nil` / `require.NotNil`
- `require.ErrorIs` / `require.NotErrorIs`
- `require.ErrorAs`
- `require.Len`
- `require.Empty`

В **табличных тестах** с небольшим количеством проверок (может даже одной проверкой), но большим количеством кейсов используют **assert**, когда хотят разом увидеть перед собой всю картину по всем кейсам и **require** – когда хотят в целом сократить время выполнения теста.



Тест "Выберите приемлимые для хорошего разработчика варианты"

Варианты проверки того, что перед нами конкретная ошибка.

Выберите все подходящие ответы из списка

- ☐ require.Error(t, err, context.DeadlineExceeded)
- ☐ require.ErrorIs(t, err, context.DeadlineExceeded)
- ☐ require.Contains(t, err.Error(), "context deadline exceeded")
- ☐ require.True(t, errors.Is(err, context.DeadlineExceeded))
- ☐ require.Error(t, err)
- ☐ require.NotNil(t, err)

Постарайтесь избегать сравнения с текстом ошибки

Мы уже неоднократно говорили об этом в курсе, но строковое представление ошибки (результат вызова метода `Error`) существует для людей, а не для программ. Оно помогает дебажить код, делает логи более понятными, но не стоит превращать текст ошибки в часть своего API.

К сожалению, люди часто пишут различные хелперы поверх strings.Contains:

```
strings.Contains(err.Error(), "invalid syntax")
```

И даже библиотеки для тестирования предоставляют подобные функции, поощряя не самые лучшие практики, например:

- require.ErrorContains
- require.ErrorEqual
- и т.д.

Ранее, в модуле лучших практик, мы рассматривали редкие случаи, когда без подобных проверок не обойтись – нам нужно полагаться на конкретную ошибку, а автор модуля не экспортирует её тип или не выделяет ошибку в отдельную **sentinel** переменную.

Освежим в памяти, что в таких случаях делать?

1) Если это стандартная библиотека, то постараться все-таки найти тип возвращаемой ошибки (или саму ошибку) и работать с ним.

2) Если это сторонний модуль – аналогично постараться найти ошибку или завести issue на его странице (пример, как это делал один из авторов данного курса).

3) Если экспортируемых типов/переменных нет, то попытаться работать с ошибкой через поведение – не предоставляет ли ошибка полезных для нас методов? Если да, то выделение интерфейса на своей стороне и `errors.As` помогут нам.

4) Если экспортируемых типов/переменных нет и нужных методов у ошибки тоже нет, то прибегаем к работе с текстом ошибки. Но вместо точного сравнения мы рекомендуем использовать strings.Contains / strings.HasPrefix.

Аналогично и в тестах:

- Мы делаем свои хелперы над "чужой" ошибкой, работающие с её текстом и возвращающие нам уже тестируемые ошибки.
- Или прибегаем к assert-функциям, работающим с текстом ошибки (`require.EqualError`, `require.Contains` и т.д.)

Задача "GetIndexFromFileName"

Ссылка на заготовку.

Существует парсер сайта объявлений, который проходится по товарам в заданном порядке и создаёт файлы с данными о товарах. Имена файлов имеют следующий формат:

```
parsed_page_1
parsed_page_2
...

parsed_page_100
```

Т.е. `parsed_page_` + `i`, где `i` – индекс странички с описанием товара. **Индекс может быть только целым положительным числом.**

С другой стороны существует приложение, которое работает с выгруженными ранее файлами. И оно содержит функцию получения индекса `i` из имени файла:

```
const prefix = "parsed_page_" // parsed_page_100

var (
    ErrInvalidFilename     = errors.New("invalid filename")
    ErrIndexMustBePositive = errors.New("index must be > 0")
)

func GetIndexFromFileName(fileName string) (int, error) {
    parts := strings.Split(fileName, prefix)
    if len(parts) != 2 || parts[1] == "" {
        return 0, fmt.Errorf("%w: no index in filename %q",
ErrInvalidFilename, fileName)
    }

    num := parts[1]

    index, err := strconv.ParseInt(num, 10, 32)
    if err != nil {
        return 0, fmt.Errorf("cannot parse index as int: %w", err)
    }

    if index <= 0 {
        return 0, fmt.Errorf("%w: got %d", ErrIndexMustBePositive,
index)
    }

    return int(index), nil
}
```

Вам необходимо заполнить тест-кейсы, которые разработчик набросал на данную функцию:

```
cases := []struct {
```

```
    fileName      string
    expectedIndex int
    expectedError error
}{
    {
        fileName:      "parsed_page",
        expectedIndex: 0,             // При необходимости нужно
исправить.
        expectedError: nil,           // При необходимости нужно
исправить.
    },
    // ...

}
```

Замечания:

- реализацию `GetIndexFromFileName` менять **нельзя**, она дана вам как есть;
- `fileName` в кейсах, их порядок и количество менять **нельзя**;
- `expectedIndex` и `expectedError` **можно и нужно** менять;
- вам доступен пакет `strconv` и определённые разработчиком ошибки.

strconv.NumError

Если посмотреть на [реализацию](#) `strconv.NumError`, то мы увидим следующее:

```
package strconv

// ErrRange indicates that a value is out of range for the target
type.
var ErrRange = errors.New("value out of range")

// ErrSyntax indicates that a value does not have the right syntax
for the target type.
var ErrSyntax = errors.New("invalid syntax")

// A NumError records a failed conversion.
type NumError struct {
```

```
    Func string // the failing function (ParseBool, ParseInt,
ParseUint, ParseFloat, ParseComplex)
    Num  string // the input
    Err  error  // the reason the conversion failed (e.g. ErrRange,
ErrSyntax, etc.)
}

func (e *NumError) Error() string {
    return "strconv." + e.Func + ": " + "parsing " + Quote(e.Num) +
": " + e.Err.Error()
}

func (e *NumError) Unwrap() error { return e.Err }

func syntaxError(fn, str string) *NumError {
    return &NumError{fn, str, ErrSyntax}
}

func rangeError(fn, str string) *NumError {
    return &NumError{fn, str, ErrRange}
}

func baseError(fn, str string, base int) *NumError {
    return &NumError{fn, str, errors.New("invalid base " +
Itoa(base))}
}

func bitSizeError(fn, str string, bitSize int) *NumError {
    return &NumError{fn, str, errors.New("invalid bit size " +
Itoa(bitSize))}

}
```

Обратим внимание на этот кусок кода:

```
func syntaxError(fn, str string) *NumError {
    return &NumError{fn, str, ErrSyntax}
}

func rangeError(fn, str string) *NumError {
    return &NumError{fn, str, ErrRange}
}

func baseError(fn, str string, base int) *NumError {
```

```
    return &NumError{fn, str, errors.New("invalid base " +
Itoa(base))}
}

func bitSizeError(fn, str string, bitSize int) *NumError {
    return &NumError{fn, str, errors.New("invalid bit size " +
Itoa(bitSize))}

}
```

Т.е. пакет экспортирует наружу ошибку синтаксиса `ErrSyntax` и ошибку превышения размерности числа `ErrRange`, но при этом не экспортирует ошибки, которые возникают при невалидном основании или количестве бит (исходник примера):

```
// https://goplay.tools/snippet/nTWLoNNIb09

// strconv.ParseInt: parsing "1": invalid base 123
_, err := strconv.ParseInt("1", 123, 32)

// strconv.ParseInt: parsing "1": invalid bit size -1
_, err = strconv.ParseInt("1", 10, -1)
```

Ошибки выше мы не можем проверить точечно через `errors.Is`, только в общей форме через

```
numErr := new(strconv.NumError)
if errors.As(err, &numErr) {
    // ...

}
```

Более того из-за этого факта разработчикам Go приходится идти к своеобразному хаку и экспортировать эти функции-конструкторы для использования в тестах:

```go
// src/strconv/export_test.go
package strconv

// Будет доступно ТОЛЬКО тестам, так как лежит в файле с суффиксом
_test.go

var (
    BitSizeError = bitSizeError
    BaseError    = baseError

)

// src/strconv/atoi_test.go
package strconv_test

// ...

func bitSizeErrStub(name string, bitSize int) error {
    return BitSizeError(name, "0", bitSize)
}

func baseErrStub(name string, base int) error {
    return BaseError(name, "0", base)
}

func noErrStub(name string, arg int) error {
    return nil
}

type parseErrorTest struct {
    arg     int
    errStub func(name string, arg int) error
}

var parseBitSizeTests = []parseErrorTest{
    {-1, bitSizeErrStub},
    {0, noErrStub},
    {64, noErrStub},
    {65, bitSizeErrStub},
}

var parseBaseTests = []parseErrorTest{
    {-1, baseErrStub},
    {0, noErrStub},
    {1, baseErrStub},
```

```
    {2, noErrStub},
    {36, noErrStub},
    {37, baseErrStub},
}

func equalError(a, b error) bool { // <-- !!!
    if a == nil {
        return b == nil
    }
    if b == nil {
        return a == nil
    }
    return a.Error() == b.Error()
}

// ...
```

И снова чудесный `equalError` :)

Попробуйте ответить для себя или напишите в комментарии:

1) Как вы считаете, для чего и зачем так было сделано?

2) Как можно исправить код функций `baseError` и `bitSizeError` так, чтобы сохранить текущее сообщение об ошибке, но при этом предоставить API для `errors.Is`?

3) Как можно исправить код функций `baseError` и `bitSizeError` так, чтобы сохранить текущее сообщение об ошибке, сохранить **невозможность** точечной проверки данных ошибок за пределами пакета, но при этом в тестах избавиться от `equalError`?

В любом случае мотайте на ус интересный приём с **экспортированием для тестов неэкспортируемого API**, такое часто можно встретить в стандартной библиотеке.

Промежуточные выводы

- не пользуйтесь `reflect.DeepEqual` для сравнений ошибок;
- избегайте в тестах сравнений с текстом ошибки – используйте `errors.Is` / `errors.As`;
- пользуйтесь инструментами, способными упростить вам тестирование вашего кода;
- не гнушайтесь пошерстить по исходникам стандартной библиотеки Go / стороннего модуля, чтобы отыскать экспортируемую ошибку или её метод.

В следующем уроке мы рассмотрим ещё парочку лучших практик по работе с ошибками в тестах и на этом закончим данный модуль.