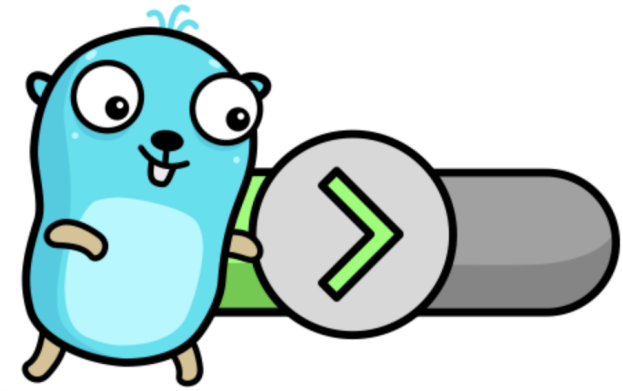
Базовые идиомы по обработке ошибок (часть 2)

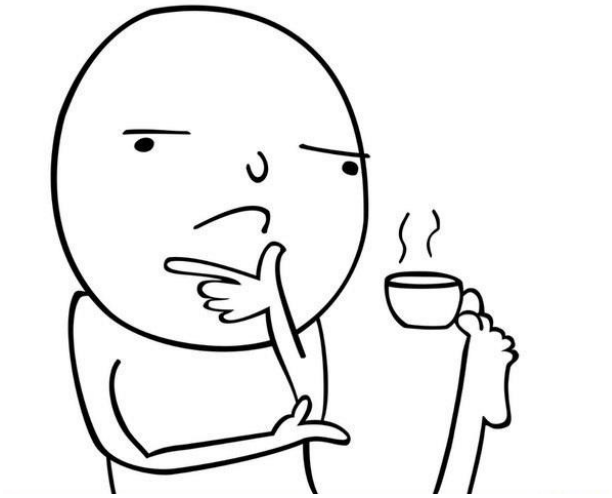
В этом уроке мы закончим обозревать базовые лучшие практики, многие из которых уже нам знакомы.

Начнём мы с нескольких gotcha, сразу закрепив практически знание о распространённых ошибках при работе с ошибками :)



Задача "err != nil (1)"

Ссылка на заготовку.



Разработчик реализовал простенький хендлер:

```
var ErrInternalServerError =
NewHTTPError(http.StatusInternalServerError)

func Handle() error {
    var err *HTTPError

    if err2 := usefulWork(); err2 != nil {
        err = ErrInternalServerError
    }
    return err

}
```

Но на его удивление данный код

```
func main() {
    if err := Handle(); err != nil {
        fmt.Println("handle err:", err)
    } else {
        fmt.Println("no handle err")
    }

}
```

вывел

```
handle err: <nil>
```

Что за магия? Ошибка якобы `nil`, но при этом мы оказались в ветке `err != nil`.

Помогите разработчику переписать функцию `Handle` так, чтобы она не попадала в популярную gotcha.

Тест "err != nil (2)"

Умудрённый опытом разработчик написал следующий код:

```
func Handle() error {
    if err := usefulWork; err != nil {
        return fmt.Errorf("%w: %v", ErrInternalServerError, err)
    }
    return nil
}

func usefulWork() error {
    return nil

}
```

Но вывод снова был непредсказуемым:

```
func main() {
    if err := Handle(); err != nil {
        fmt.Println("handle err:", err)
    } else {
        fmt.Println("no handle err")
    }
}
```

Вместо отсутствия ошибки он получил

```
handle err: code=500, message=Internal Server Error: 0x1028e1bd0
```

Вам необходимо исправить одну единственную строчку, чтобы код заработал верно.

Исправленную строчку нужно вставить как ответ.

Задача "err != nil (3)"

Ссылка на заготовку.

Перед вами следующий тип:

```
type OperationsErrors []error

func (oe OperationsErrors) Error() string {
    return fmt.Sprintf("operations errors: %v", []error(oe))

}
```

Наш несчастливый разработчик заиспользовал его следующим образом:

```
type IOperation interface {
    Do() error
}
```

```
func Handle(ops ...IOperation) error {
    var opsErrs OperationsErrors

    for _, op := range ops {
        if err := op.Do(); err != nil {
            opsErrs = append(opsErrs, err)
        }
    }

    return opsErrs

}
```

Несчастливый, потому что вместо отсутствия ошибок, он снова получил странную ошибку:

```
func main()
    if err := Handle(successOperation{}); err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("no operations errors")
    }

}

operations errors: []
```

Перепишите функцию `Handle` так, чтобы её основная логика не изменилась, но сама функция работала корректно.

Тест "Ошибка под прикрытием"

Следующий код

```
var ErrInvalidUserID = errors.New("invalid user id")

type UserID string

type User struct {
    ID UserID
}

func SaveUser(u User) (err error) {
    if !isValidID(u.ID) {
        err := fmt.Errorf("%w: %v", ErrInvalidUserID, u.ID)
        return
    }

    return saveUser(u)

}
```

не компилируется с фразой

```
./main.go: err is shadowed during return
```

Вам необходимо исправить одну единственную строчку, чтобы код заработал верно.

Исправленную строчку нужно вставить как ответ.

Shadowed errors

Shadowed переменные получаются при создании одноимённых переменных в новом блоке видимости:

```
// https://goplay.tools/snippet/WsFKNL3PstH

func main() {
    var i int

    if true {
        i := 1 // Здесь создаётся новая переменная в рамках области видимости if.
        i++
    }

    fmt.Println(i) // 0, не 2!
}
```

Некоторые перекрытия отлавливаются go vet или самим компилятором гошеньки, что мы и видели в предыдущем шаге:

```
func SaveUser(u User) (err error) {
    if !isValidID(u.ID) {
        err := fmt.Errorf("%w: %v", ErrInvalidUserID, u.ID) // <- Новая ошибка перекрыла возвращаемую.
        return
    }

    return saveUser(u)

}
```

Бывают осознанные перекрытия, например, когда мы хотим получить новую ошибку на основе существующей

```
func (s *Store) Create(req CreateTransactionRequest) (*Transaction, error) {
    tr := Transaction{
        ID:         req.ID,
```

```
        CreatedAt: req.Now,
        ClientID:  req.ClientID,
    }
    res, err := s.tx.Model(&tr).Insert()

    if err := storage.GetError(err, res); err != nil {
        return nil, fmt.Errorf("cannot insert transaction: %w", err)
    }
    /*  Но всё равно чище было бы:
    if storeErr := storage.GetError(err, res); storeErr != nil {
        return nil, fmt.Errorf("cannot insert transaction: %w",
storeErr)
    }
    */

    return &tr, nil

}
```

Или когда хитро "отцепляемся" от верхнеуровневой ошибки, чтобы произвести какие-то действия с текущей:

```
// https://goplay.tools/snippet/-6lMlr-BYbR

func Handle() (err error) {
    if err = handleConn(); err != nil {
        // Новая ошибка не перетирает возвращаемую.
        // Попробуйте заменить `:=` на `=`.
        if err := closeConn(); err != nil {
            log.Println(err)
        }
    }
    return
}


func main() {
    fmt.Println(Handle())
}

/*
2009/11/10 23:00:00 close error
handle conn error

*/
```

Но в общем случае, чтобы не думать о shadowed errors, следует писать код в следующем стиле:

1. Соблюдать Error Flow, возвращать ошибку сразу, как только это возможно.
2. Постараться избегать именованных возвращаемых значений.
3. Стараться заносить `err` в `if`, чтобы каждый раз это была новая переменная.

```
// ...
    client, err := getClient()
    if err != nil {
        return "", fmt.Errorf("cannot get client: %w", err)
    }

    if err := s.opts.storage.DoInTx(ctx, func(tx storage.Tx) error {
        balance, err := getBalance(client)
        if err != nil {
            return fmt.Errorf("cannot get client balance: %w", err)
        }

        if err := withDrawMoney(balance); err != nil {
            return fmt.Errorf("cannot withdrow money: %w", err)
        }

        return nil

    }); err != nil {
        return "", fmt.Errorf("cannot do transaction: %w", err)
    }

    if err := s.opts.signals.Notify(ctx); err != nil {
        return "", fmt.Errorf("cannot notify about changes: %w", err)
    }
// ...
```

Выводы

Будьте бдительны и не пренебрегайте линтерами!

Документируйте ошибки, возвращаемые функцией

Одной из неприятных особенностей Go является то, что **синтаксически невозможно понять, какие ошибки возвращает функция или метод** (нам поможет только просмотр документации или исходного кода функции). Более того нельзя на уровне компиляции принудить пользователя нашей функции обрабатывать все возвращаемые нами ошибки.

Более приятные механизмы присутствуют в других языках программирования. Например, в Java есть понятие checked exceptions – ~~грубо говоря,~~ когда мы можем декларировать, какие исключения функция выбрасывает, и компилятор будет требовать их обработки на уровне выше:

```
// https://onlinegdb.com/u9qtz4MXK

import java.io.EOFException;
import java.io.FileNotFoundException;

class App {
    public void Run() throws EOFException, FileNotFoundException {
        f0();
        f1();
    }
    private void f0() throws EOFException { /*...*/ }
    private void f1() throws FileNotFoundException { /*...*/ }
}


public class Main
{
    public static void main(String[] args) {
        App app = new App();
        app.Run(); // error: unreported exception EOFException; must be caught or declared to be thrown
    }
}
```

```
// https://onlinegdb.com/2Fm8y4TpM

public static void main(String[] args) {
    App app = new App();
    try {
        app.Run();
    } catch (EOFException | FileNotFoundException e) { // Ошибок компиляции больше нет.
        /* ... */
    }

}
```

В Go же нам остаётся только грамотно документировать своё API.

Хорошим примером здесь послужит стандартная библиотека:

- Комментарий к `(*sql.Row).Scan` объясняет, когда вы получите `sql.ErrNoRows`:

```
package sql

// Scan uses the first row and discards the rest. If no row matches
// the query, Scan returns ErrNoRows.

func (r *Row) Scan(dest ...interface{}) error
```

- Комментарий к `io.ReadAtLeast` объясняет, когда вы получите `io.EOF`, `io.ErrUnexpectedEOF` или `io.ErrShortBuffer`:

```
package io

// ReadAtLeast reads from r into buf until it has read at least min bytes.
// The error is EOF only if no bytes were read.
```

```
// If an EOF happens after reading fewer than min bytes, ReadAtLeast
returns ErrUnexpectedEOF.
// If min is greater than the length of buf, ReadAtLeast returns
ErrShortBuffer.

func ReadAtLeast(r Reader, buf []byte, min int) (n int, err error)
```

Или пример документации из стороннего модуля valyala/fasthttp (объясняются причины появления ошибок `fasthttp.ErrTooManyRedirects` и `fasthttp.ErrNoFreeConns`):

```
// DoRedirects performs the given http request and fills the given
http response,
// following up to maxRedirectsCount redirects. When the redirect
count exceeds
// maxRedirectsCount, ErrTooManyRedirects is returned.
//
// ErrNoFreeConns is returned if all DefaultMaxConnsPerHost
connections
// to the requested host are busy.

func (c *Client) DoRedirects(req *Request, resp *Response,
maxRedirectsCount int) error
```

Выводы

Старайтесь документировать ваше API на предмет возвращаемых ошибок. Понятно, что в рамках маленького проекта обычно на это подзабивают, но если вы разрабатываете библиотеку для "внешнего мира", то без этого не обойтись.

Тест "Продокументируйте функцию"

Продокументируйте функцию `Calculate` так, чтобы это было валидно с точки зрения godoc, и при этом комментарий отражал возвращаемые функцией ошибки:

```
package factorial

const maxDepth = 256

var (
```

```
    ErrNegativeN = errors.New("negative n")
    ErrTooDeep   = fmt.Errorf("calculation will exceed %d frames",
maxDepth)
)

func Calculate(n int) (int, error) {
    if n < 0 {
        return 0, ErrNegativeN
    }

    if n > maxDepth {
        return 0, ErrTooDeep
    }

    return calculate(n), nil

}
```

Ответ следует предоставить в виде

```
// Здесь должен быть комментарий
// к функции ниже.

func Calculate(n int) (int, error)
```

P.S. Обратите внимание на

The convention is simple: to document a type, variable, constant, function, or even a package, write a regular comment directly preceding its declaration, with no intervening blank line.

Отдавайте предпочтение стандартной библиотеке

Если вы работаете с ошибками, создаете их или обрабатываете, то отдавайте предпочтение комбинации пакетов `errors` и `fmt`, т.е. при выборе инструмента **отдавайте предпочтение стандартной библиотеке Go**, особенно, если вы разрабатываете модуль, которым будут пользоваться другие проекты.

Почему же?

- чем меньше внешних зависимостей, тем лучше;
- код работает прозрачнее: нередко библиотеки под капотом делают много лишнего или в целом содержат неприятную магию.

Разумеется, может быть такое, что без использования условного `github.com/pkg/errors` вам не обойтись.

Посыл здесь прост – **знайте инструменты, с которыми работаете, и используйте их по назначению**.

В целом правило приоритета стандартной библиотеки над внешними модулями относится не только к ошибкам.

Мы предлагаем вам ответить в комментариях на вопрос "**Почему внешние зависимости не есть хорошо, особенно, когда их много?**". Наверняка вы уже знакомы с понятием dependency hell.