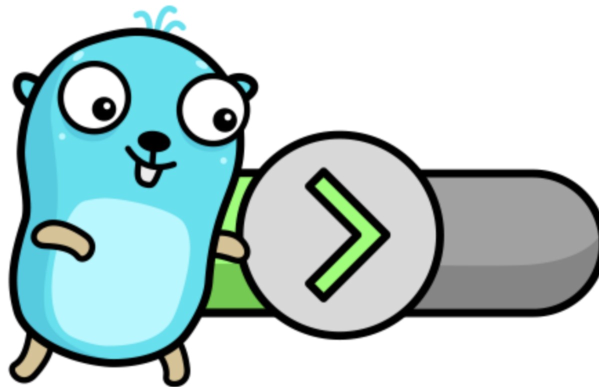
Лучшие практики врапинга

В этом уроке мы поговорим, когда лучше врапить и как, а когда стоит воздержаться от этого :)



Не забываем врапить

Мы [помним](), что при использовании стандартной библиотеки важно организовывать путь до ошибки, иначе может быть сложно понять её причину.

Для этого следует соблюдать следующие правила:

- врапим ошибку в текст на каждом уровне;
- на одном уровне избегаем дублирования текста, чтобы путь до ошибки был уникален.

Плохо:

```
// ...
    res, err := m.conn.ExecContext(ctx, query, params...)
    if err != nil {
        return err
    }

    affected, err := res.RowsAffected()
    if err != nil {
        return err
```

```go
    }

    if affected == 0 {
        return ErrRecordNotFound
    }

// ...
```

Не лучше:

```go
// ...
    res, err := m.conn.ExecContext(ctx, query, params...)
    if err != nil {
        return fmt.Errorf("cannot update user: %v", err)
    }

    affected, err := res.RowsAffected()
    if err != nil {
        return fmt.Errorf("cannot update user: %v", err)
    }

    if affected == 0 {
        return ErrRecordNotFound
    }

// ...
```

Хорошо:

```go
// ...
    res, err := m.conn.ExecContext(ctx, query, params...)
    if err != nil {
        return fmt.Errorf("cannot exec context: %v", err)
    }

    affected, err := res.RowsAffected()
    if err != nil {
        return fmt.Errorf("cannot get affected rows: %v", err)
    }

    if affected == 0 {
        return ErrRecordNotFound
    }

// ...
```

errors.Wrap из github.com/pkg/errors

Лишний раз напоминаем, что повсеместное использование `errors.Wrap` приводит к портянкам из копирующих друг друга стеков. Мы уже говорили об этом ранее на уроке ["Нюанс повсеместного Wrap"].

Напомним сигнатуры функций из **pkg/errors**:

```go
package errors

// WithMessage annotates err with a new message.
func WithMessage(err error, message string) error { /* ... */ }

// Wrap returns an error annotating err with a stack trace
// at the point Wrap is called, and the supplied message.
func Wrap(err error, message string) error { /* ... */ }

// WithStack annotates err with a stack trace at the point WithStack
was called.

func WithStack(err error) error { /* ... */ }
```

Какие есть варианты?

- Не использовать **pkg/errors** в принципе в пользу стандартной библиотеки.

- Не париться и использовать везде `errors.Wrap`, а в месте вывода на экран "вырезать" лишние стектрейсы 🤦‍♂️.

- Следить за коллегами и бить их по рукам, если `errors.Wrap` используется где-то кроме как на слое базульки или клиента к внешнему сервису, в остальных случаях требовать `errors.WithMessage`.

- Также существует практика, когда единственный раз, на самом глубоком слое приложения, делают `errors.WithStack`, а затем просто возвращают ошибку как есть (без никаких оборачиваний и т.д.):

```
// level N
if err := opN(); err != nil {
    logWithStackTrace(err)       // Здесь и ниже не нужны доп.
сообщения, так как есть стек.
}

    // level N - 1
    if err := opNm1(); err != nil {
        return err                     // Никакого врапинга!
    }

        // ...

            // level 1
            if err := op1(); err != nil {
                return err                 // Никакого врапинга!
            }

                // level 0
                if err := op0(); err != nil {
                    return errors.WithStack(err)    // Прицепляем
стек.

                }
```

Это может показаться непривычным и вызовет ругань некоторых линтеров, но на самом деле делает код чище и в целом является приемлимым вариантом для небольших проектов.

Из минусов:

- нужно гарантировать наличие `errors.WithStack` "внизу" и вывод ошибки через `"%+v"` наверху;

- скорее всего дебаг **error path** в реальном времени будет затруднён (если вывести стектрейс возможности нет, а текст ошибки содержит только корневую ошибку).

Какие видятся проблемы?

Разработчикам нужно думать, какой функцией (`errors.Wrap/WithStack` или `errors.WithMessage`) в данный момент времени пользоваться, как следствие – разбираться в API используемой библиотеки (вы удивитесь, но не всегда люди разбираются в инструментах, которыми ежедневно пользуются).

Кроме того бытует мнение, что не всегда можно однозначно сказать, какой именно слой приложения является самым глубоким (подходящим для `errors.Wrap/WithStack`) и не появится ли со временем за ним более глубокий слой (например, наше хранилище станет промежуточным кешом к другому хранилищу). В таком случае поможет разве что свой форк **pkg/errors**, работающий так, как вам хочется, или в целом свой проприетарный модуль для работы с ошибками 🤯.

Пишите в комментариях, какие практики приняты у вас в компании.

Задача "Идемпотентный Wrap"

[Ссылка на заготовку](#).

Разработчики решили, что проще сделать свой `Wrap`, который цепляет стектрейс к ошибке единожды, чем запоминать, какие функции **github.com/pkg/errors** и когда это делают.

Вам необходимо реализовать следующую функцию:

```
// Wrap оборачивает ошибку в сообщение. Также добавляет к ошибке стектрейс,
// если в цепочке уже нет ошибки со стектрейсом.
func Wrap(err error, msg string) error
```

Пример вызова:

```
err := Wrap(Wrap(Wrap(io.EOF, "wrap 0"), "wrap 1"), "wrap 2")
fmt.Printf("%+v", err)

/*
EOF
wrap 0
tasks/05-errors-best-practices/idempotent-wrap.Wrap
	tasks/05-errors-best-practices/idempotent-wrap/wrap.go:21
tasks/05-errors-best-practices/idempotent-wrap.ExampleWrap
	tasks/05-errors-best-practices/idempotent-wrap/wrap_test.go:13
testing.runExample
	/usr/local/go/src/testing/run_example.go:64
testing.runExamples
	/usr/local/go/src/testing/example.go:44
testing.(*M).Run
	/usr/local/go/src/testing/testing.go:1505
main.main
	_testmain.go:45
runtime.main
	/usr/local/go/src/runtime/proc.go:255
runtime.goexit
	/usr/local/go/src/runtime/asm_arm64.s:1133
wrap 1
wrap 2

*/
```

Обратите внимание, что стектрейс содержит только `"wrap 0"`.

Вам доступен интерфейс `stackTracer`:

```go
import "github.com/pkg/errors"

type stackTracer interface {
    StackTrace() errors.StackTrace

}
```

К сожалению, мы не можем проверить эту задачу средствами Stepik, поэтому она остаётся на самостоятельное изучение, совесть и желание студента:

```
$ cd tasks/05-errors-best-practices/idempotent-wrap
$ go test  -v .
=== RUN   TestWrap
=== RUN   TestWrap/single_wrap
=== RUN   TestWrap/use_wrap_only
=== RUN   TestWrap/wrapped_error_already_has_stack
--- PASS: TestWrap (0.00s)
    --- PASS: TestWrap/single_wrap (0.00s)
    --- PASS: TestWrap/use_wrap_only (0.00s)
    --- PASS: TestWrap/wrapped_error_already_has_stack (0.00s)
PASS

ok      tasks/05-errors-best-practices/idempotent-wrap      0.284s
```

Чтобы увидеть авторское решение просто нажмите "Отправить". Чтобы опубликовать своё решение, вставьте кодовую вставку в комментарий к оставленному решению (по аналогии с авторским).

Единообразие глагола

Для упрощения `grep` или визуального поиска ошибки рекомендуется унифицировать глагол, используемый при врапинге ошибок.

Обычно это `failed to` или `cannot`:

```
cannot start app: cannot read config: open config.json: no such file or directory
```

Задача "Pretty Error"

[Исходник примера](#).

Разработчик увлёкся архитектурой своего приложения и оно получилось достаточно слоистым.

Это усложнило дебаг глубоких ошибок, так как их сообщение стало очень длинным:

```go
err := fmt.Errorf("cannot get user schedule: %w",
    fmt.Errorf("cannot build data for event: %w",
```

```
        fmt.Errorf("cannot get image: %w",
            fmt.Errorf("cannot get image: %w",
                fmt.Errorf("cannot get image: %w",
                    fmt.Errorf("cannot get image for event: %w",
sql.ErrNoRows))))))

fmt.Println(errors.Is(err, sql.ErrNoRows) // true


fmt.Println(err) // cannot get user schedule: cannot build data for
event: cannot get image: cannot get image: cannot get image: cannot
get image for event: sql: no rows in result set
```

Вам необходимо реализовать функцию `Pretty`:

```
// Pretty делает цепочку ошибок более читаемой - очередная
завраплленная
// ошибка будет представлена на новой строке.

func Pretty(err error) error
```

которая возвращает логически идентичную ошибку, но с более приятным сообщением (похожим на вывод стектрейса):

```
pErr := PrettyError(err)

fmt.Println(errors.Is(pErr, sql.ErrNoRows)) // true

fmt.Println(pErr)
/* cannot get user schedule:
    cannot build data for event:
    cannot get image:
    cannot get image:
    cannot get image:
    cannot get image for event:

    sql: no rows in result set */
```

Допущения:

- считаем, что все звенья цепочки ошибок созданы через спецификатор `%w`;
- вам доступны пакеты `fmt`, `errors`, `strings` и `regexp`.

Отсутствие глагола

Чтобы избежать проблемы, которую мы решали в предыдущей задаче, тот же Uber, например, рекомендует избегать глаголов при врапинге в принципе:

When adding context to returned errors, keep the context succinct by avoiding phrases like "failed to", which state the obvious and pile up as the error percolates up through the stack.

Плохо:

```
// cannot build data for event: cannot get image for event: sql: no rows in result set
```

Хорошо:

```
// build data for event: get image for event: sql: no rows in result set
```

Как видно, это не всегда сильно укорачивает сообщение об ошибке, а вот её поиск может усложнить (особенно, если ошибки логируются не через `ERROR`).

Поэтому как лучше делать – решать вам :)

Ошибки и границы API

Бывает практика, когда существует соглашение на весь проект повсеместно использовать врапинг через `%w`.

С одной стороны это хорошо: это позволяет нам в любом месте цепочки ошибок обрабатывать нижележащие ошибки, но с другой стороны – такой подход может породить **утечку абстракции**. Разберём данный тезис на популярном примере.

Существует метод, внутри себя выполняющий поход в базульку (исходник примера):

```
func (db *DB) GetUserByID(ctx context.Context, uid UserID) (*User, error) {
    if err := selectUser(); err == sql.ErrNoRows {
        return nil, fmt.Errorf("exec query: %w", err)
    }
    return &User{ID: "42"}, nil

}
```

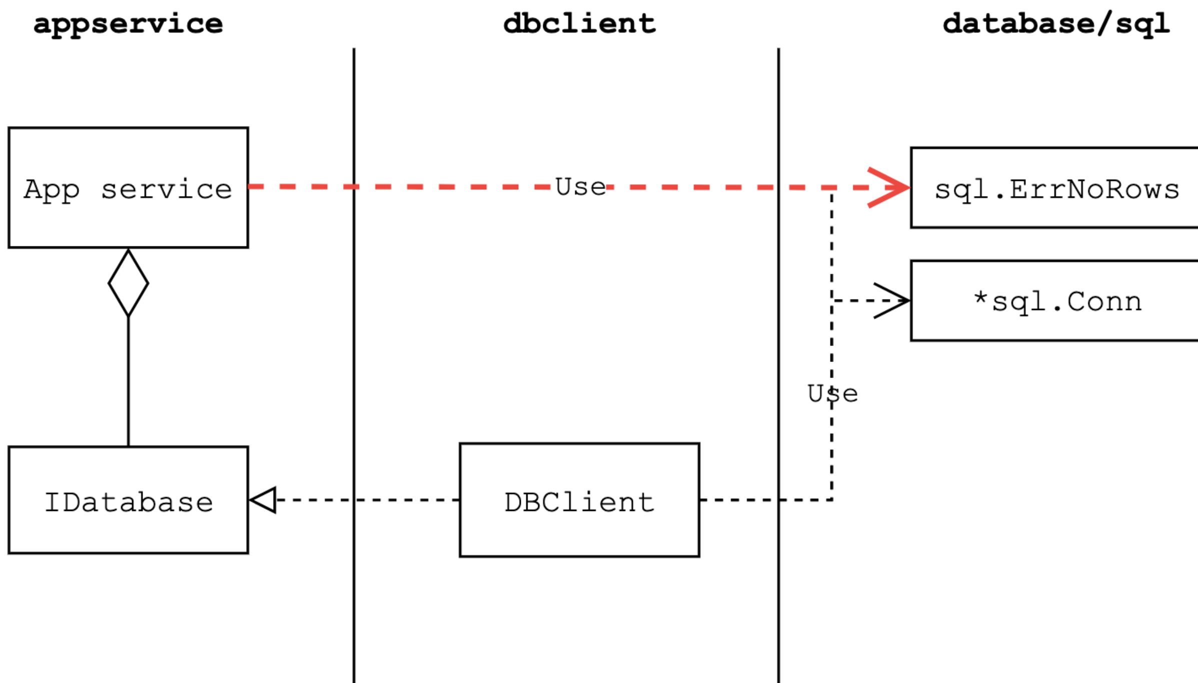
При этом сам сервис, которому принадлежит метод, используется через интерфейс:

```
type IDatabase interface {
    GetUserByID(ctx context.Context, uid UserID) (*db.User, error)

}
```

В таком случае врапинг через `%w` делает ошибку `sql.ErrNoRows` частью нашего контракта: мы не сможем, например, без потери обратной совместимости поменять драйвер к базульке или пересесть на другую ошибку.

Более того, возврат `sql.ErrNoRows` (как и, к слову, использование `sql.Null*`-типов в возвращаемой структуре) позволяет клиентам нашего API "прыгать" через слои приложения, что не есть хорошо:



Что не скажешь про врапинг через `%v`, который запрещает полагаться на обёрнутую ошибку, но при этом сохраняет её текст, никак не мешая дебагу:

```
if err != nil {
    return nil, fmt.Errorf("exec query: %v", err)

}
```

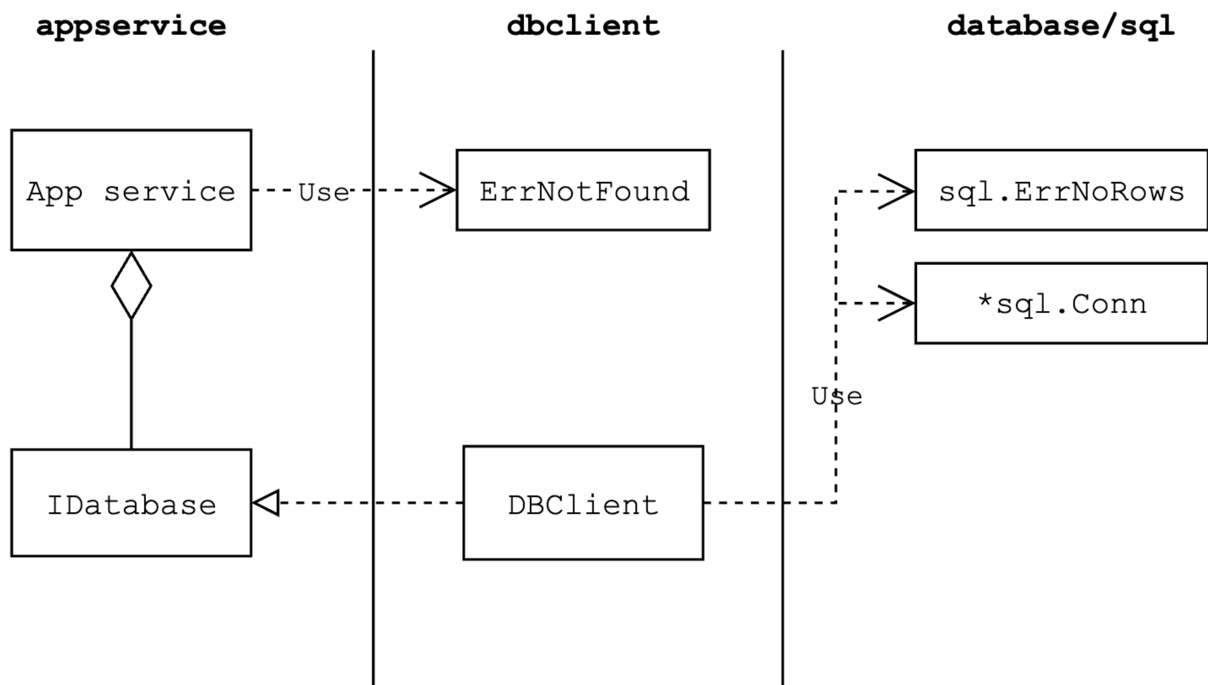
Но что же делать, если нам необходимо строить логику поверх возвращаемых ошибок? **Завернуть чужие ошибки в свои!**

```
var ErrNotFound = errors.New("obj not found")

func (db *DB) GetUserByID(ctx context.Context, uid UserID) (*User, error) {
    if err := selectUser(); err == sql.ErrNoRows {
        return nil, fmt.Errorf("exec query: %w: %v", ErrNotFound, err)
```

```
    }
    return &User{ID: "42"}, nil
}
```

Тем самым убрав сомнительную зависимость:



Другим вариантом является использование **opaque errors**.

Можно попробовать оставить самый первый вариант врапинга и построить функцию над исходной ошибкой:

```
package dbutils

func IsNotFoundError(err error) bool {
    return errors.Is(err, sql.ErrNoRows)
}

// ...

user, err := db.GetUserByID(ctx, uid)
if dbutils.IsNotFoundError(err) {
```

```
    // ...
}
```

Но это чревато тем, что `sql.ErrNoRows` всё равно "торчит наружу" и по ней могут строить какую-то логику, забывая про хелпер выше.

Аналогичная двойственность возникает и с вариантом поверх нашей собственной ошибки

```
func IsNotFoundError(err error) bool {
    // Утечки абстракции уже нет, но есть несколько путей работы с
нашими ошибками.
    return errors.Is(err, ErrNotFound)

}
```

Для решения этой проблемы делаем нашу ошибку неэкспортируемой, оставляя экспортируемый хелпер:

```
var errNotFound = errors.New("obj not found")

// ...
    if err != nil {
        return nil, fmt.Errorf("exec query: %w: %v", errNotFound,
err)
    }

// ...

package dbutils

func IsNotFoundError(err error) bool {
    return errors.Is(err, errNotFound)
}
```

Если вы хотите скрыть детали исходной ошибки (по неведомым нам причинам, ~~усложнив дебаг и выстрелив себе в ногу~~), то можно возвращать только свою ошибку, без добавления текста исходной:

```
// ...
    if err != nil {
        return nil, fmt.Errorf("exec query: %w", errNotFound)
    }

// ...
```

Выводы

Если вы не хотите делать ошибки частью своего контракта, то и не делайте :)

Пример с базулькой скорее учебный, так как вряд ли вы каждую неделю будете пересаживаться на новую базу данных или новый драйвер к существующей. Но принципы, рассмотренные выше, актуальны для любого API внутри вашего приложения и их соблюдение поможет писать код более устойчивый к дальнейшим изменениям.

Поэтому при очередном врапинге (будь то `pkg/errors.Wrap` или `fmt.Errorf + %w`), задумайтесь, а точно ли он нужен именно в таком виде?

Стандартной библиотеки достаточно

В целом авторы курса придерживаются идеи, что без стектрейса жить можно и стандартной библиотеки достаточно для работы с ошибками. **Пишите в комментарии, что сами думаете об этом**.

Но если вы пользуетесь сторонним модулем, то мы рекомендуем не миксовать подходы и тот же врапинг, например, осуществлять в едином стиле на весь проект.

Плохо:

```
import (
    "fmt"
    "github.com/pkg/errors"
)


// ...
    if err != nil {
        return fmt.Errorf("exec context: %w", err)
    }

    affected, err := res.RowsAffected()
    if err != nil {
        return errors.Wrap(err, "get affected rows")
    }

// ...
```

Хорошо:

```
import "github.com/pkg/errors"


// ...
    if err != nil {
        return errors.WithMessage(err, "exec context")
    }
```

```go
    affected, err := res.RowsAffected()
    if err != nil {
        return errors.Wrap(err, "get affected rows")
    }

// ...

import "fmt"

// ...
    if err != nil {
        return fmt.Errorf("exec context: %w", err)
    }

    affected, err := res.RowsAffected()
    if err != nil {
        return fmt.Errorf("get affected rows: %w", err)
    }
// ...
```

Следить за этим помогут линтеры, о которых мы поговорим чуть дальше.

P.S. На какие грабли можно наступить при смешивании подходов к работе с ошибками, мы разбирали ранее в модуле [github.com/pkg/errors (часть 2)](github.com/pkg/errors).

Тест "Совместите код и проблему в нём"

1)

```go
// ...
    srcFile, err := os.Open(fromPath)
    f err != nil {
        return ErrUnableOpenSourceFile
    }
```

```go
    defer srcFile.Close()

    si, err := srcFile.Stat()
    if err != nil {
        return ErrUnableGetStat
    }

    if !si.Mode().IsRegular() {
        return ErrCopyNonRegular
    }

// ...
```

2)

```go
fmt.Println(err)

// cannot build data for event: failed to get image for event: error while executing query
```

3)

```go
import (
    "fmt"
    "github.com/pkg/errors"
)

func operation() error {
    // ...
    return fmt.Errorf("cannot do request: %w", context.Canceled)
}

func main() {
    if err := operation(); errors.Cause(err) == context.Canceled {
        // ...
    }
    // ...
}
```

4)

```go
import (
    "mymodule/internal/rmq"
```

```
    "github.com/streadway/amqp"
)

// ...
    rmq, err := rmq.New()
    if err != nil {
        return fmt.Errorf("create rmq client: %v", err)
    }

    ch, err := rmqClient.OpenChannel()
    if errors.Is(err, amqp.ErrClosed) {
        // ...
    }

// ...
```

Сопоставьте значения из двух списков

Потеря информации об исходной ошибке.

Смешивание глаголов при врапинге ошибок или в принципе наличие глагола.

Использование и стандартной и нестандартной библиотек для работы с ошибками

Нарушение границ слоёв приложения.