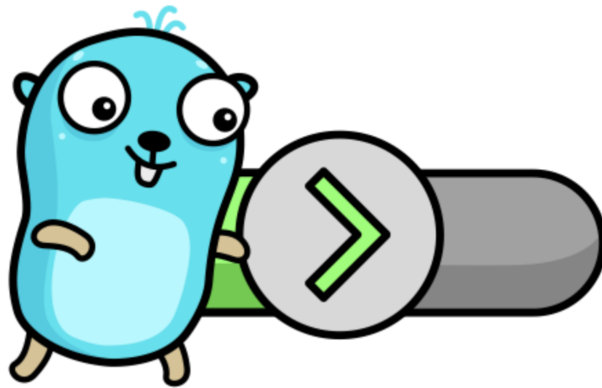
github.com/pkg/errors (часть 2)

В данном уроке мы поговорим о совместимости стандартного **errors** и **github.com/pkg/errors**.

Являются ли они взаимозаменяемыми и можно ли малой кровью пересесть с одного на другое и обратно?



Тест "Выберите функцию с наиболее близким к fmt.Errorf поведением"

Выберите один вариант из списка

- [ ] errors.New
- [ ] errors.Errorf
- [ ] errors.WithStack
- [ ] errors.WithMessage
- [ ] errors.WithMessagef
- [ ] errors.Wrap
- [ ] errors.Wrapf

Врапинг через pkg/errors VS fmt.Errorf

Стоит держать в голове несколько отличий функций врапинга из **github.com/pkg/errors** и `fmt.Errorf`.

---

1) Пользуясь стандартной библиотекой, вы можете поместить оборачиваемую ошибку в любое место новой ошибки, так как оно регламентируется положением директивы `%w`.

pkg/errors же всегда оборачивает ошибку слева (исходник примера):

```go
package errors

type withMessage struct {
    cause error
    msg   string
}

func (w *withMessage) Error() string {
    return w.msg + ": " + w.cause.Error()

}

// https://goplay.tools/snippet/5E5_Z-B3C7z
package main

import (
    stderrors "errors"
    "fmt"

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

```
)

var ErrNotFound = stderrors.New("not found")

func main() {
    {
        err := fmt.Errorf("%w: index.html", ErrNotFound)
        fmt.Println(err) // not found: index.html

        err = fmt.Errorf("in the middle: %w: index.html", 
ErrNotFound)
        fmt.Println(err) // in the middle: not found: index.html

        err = fmt.Errorf("index.html: %w", ErrNotFound)
        fmt.Println(err) // index.html: not found
    }

    {
        err := errors.Wrap(ErrNotFound, "index.html")
        fmt.Println(err) // index.html: not found
    }
}
```

---

2) Функции врапинга из **pkg/errors** врапят *безусловно*, т.е. нельзя заврапить ошибку так, чтобы для неё перестал работать `errors.Is`. Где это может быть полезно, мы узнаем в следующем модуле лучших практик.

Но для сокрытия ошибки вы, как и в стандартной библиотеке, можете использовать `errors.Errorf` + `%v`

(исходник примера):

```
// https://goplay.tools/snippet/EyjhIAD2Baw

{
    err := fmt.Errorf("index.html: %v", ErrNotFound)
    fmt.Println(err, "|", errors.Is(err, ErrNotFound)) // index.html: 
not found | false
}

{
```

```
    err := errors.Errorf("index.html: %v", ErrNotFound)
    fmt.Println(err, "|", errors.Is(err, ErrNotFound)) // index.html:
not found | false

}
```

---

3) Очевидно, что директива `%w` поддерживается только `fmt.Errorf` (т.к. **pkg/errors** использует `fmt.Sprintf`, а не самостоятельно формирует форматированную строку).

Сделать двойной врапинг или просто поменять `fmt.Errorf` на `errors.Errorf` не выйдет (исходник примера):

```
// https://goplay.tools/snippet/mHbMCY9pQ8h

{
   err := fmt.Errorf("index.html: %w", ErrNotFound)
   fmt.Println(err, "|", errors.Is(err, ErrNotFound)) // index.html:
not found | true
}

{
   err := errors.Errorf("index.html: %w", ErrNotFound)
   fmt.Println(err, "|", errors.Is(err, ErrNotFound)) // index.html:
%!w(*errors.errorString=&{not found}) | false
}

{
   err := errors.Wrapf(ErrNotFound, "index.html: %w", io.EOF)
   fmt.Println(err, "|", errors.Is(err, ErrNotFound)) // index.html:
%!w(*errors.errorString=&{not found}) | true

}
```

Устойчивость к nil-ошибкам

Также ключевым различием между функциями врапинга из **pkg/errors** и `fmt.Errorf` является то, как они работают с `nil`-ошибкой, переданной в аргументы (исходник примера):

```
// https://goplay.tools/snippet/HOz1Lduq1Vf
```

```go
var err error // nil error

{
    err := fmt.Errorf("do operation: %w", err)
    fmt.Println(err, "|", err == nil) // do operation: %!w(<nil>) |
false
}

{
    err := errors.Wrap(err, "do operation")
    fmt.Println(err, "|", err == nil) // <nil> | true

}
```

`errors.Wrap` без труда "проглатывает" её и остаётся `nil`, а вот с `fmt.Errorf` такое уже не прокатит.

Такая особенность `Wrap`-функций **github.com/pkg/errors** позволяет не писать лишний раз `if err != nil`, если возврат ошибки происходит непосредственно перед `return`:

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

// Было.
err := r.orderManager.ApplyTransition(ctx, orderID, checksum,
transition, &reason)
if err != nil {
    return errors.Wrap(err, "apply order transition error")
}
return nil

// Стало.
err := r.orderManager.ApplyTransition(ctx, orderID, checksum,
transition, &reason)

return errors.Wrap(err, "apply order transition error")
```

Но, если мы пользуемся строго стандартной библиотекой, то сократить не получится:

```go
// Было.
```

```go
err := r.orderManager.ApplyTransition(ctx, orderID, checksum,
transition, &reason)
if err != nil {
    return fmt.Errorf("apply order transition error: %w", err)
}
return nil

// Стало.
err := r.orderManager.ApplyTransition(ctx, orderID, checksum,
transition, &reason)

return fmt.Errorf("apply order transition error: %w", err) //
Неправильно: всегда будет возвращать ошибку!
```

Резюме

Будьте внимательны, если комбинируете в своих проектах обе библиотеки и используете особенности тех или иных функций. Берегите себя и своих близких.

Тест "И такое бывает" (по мотивам реальной баги)

Ссылка на пример.

Попробуйте, не запуская кода ниже, ответить, что выведется на экран:

```go
package main

import (
    "fmt"

    "github.com/pkg/errors"
)

func main() {
    listener, err := createEventListener()
    if err != nil {
        fmt.Println(err)
        return
    }
```

```
    if err := listener(); err != nil {
        fmt.Println(err)
    }
}

func createEventListener() (func() error, error) {
    obj, err := getObject()
    if err != nil {
        return nil, errors.Wrap(err, "get object")
    }

    if obj.ID == 42 {
        return nil, errors.Wrap(err, "events is not supported for
this obj")
    }

    return func() error {
        fmt.Println("ok")
        return nil
    }, nil
}

type Object struct {
    ID int
}

func getObject() (*Object, error) {
    return &Object{ID: 42}, nil
}
```

Выберите один вариант из списка

- ☐ ok
- ☐ Ничего не выведется, произойдёт паника.
- ☐ get object
- ☐ events is not supported for this obj

Задача "Wrap nil"

[Ссылка на заготовку].

Вам необходимо реализовать функцию `Wrapf` средствами стандартной библиотеки:

```
// Wrapf работает аналогично fmt.Errorf, только поддерживает
nil-ошибки.

func Wrapf(err error, f string, v ...any) error
```

errors.Cause VS errors.Is/As

В один прекрасный момент могут возникнуть следующие вопросы:

- Насколько совместимы функции для оборачивания/разворачивания **github.com/pkg/errors** и стандартной библиотеки?
- Стоит ли сильно переживать, если в нашем проекте используется и то другое?
- Чем доставать ошибку на верхнем уровне – `errors.Cause` или православным `errors.Is` (`errors.As`)?

Для ответа на эти вопросы ещё раз посмотрим на реализацию ошибок в соответствующих пакетах:

```
// src/fmt/errors.go
package fmt

type wrapError struct {
    msg string
    err error
}

func (e *wrapError) Error() string { return e.msg }
func (e *wrapError) Unwrap() error { return e.err }
```

```go
// github.com/pkg/errors/errors.go
package errors

type withMessage struct {
    cause error
    msg   string
}

func (w *withMessage) Error() string { return w.msg + ": " +
w.cause.Error() }
func (w *withMessage) Cause() error  { return w.cause }
func (w *withMessage) Unwrap() error { return w.cause }
```

Мы видим, что ошибки из **pkg/errors** поддерживают и своё API и стандартной библиотеки (`Cause` + `Unwrap`), но ошибки из стандартной библиотеки ничего не знают о **pkg/errors** (что, конечно же, логично).

Из этого можно сделать следующие выводы:

- `errors.Cause` будет работать, только если все ошибки в цепочке созданы через **github.com/pkg/errors**.
- `errors.Is/As` будут работать для цепочки ошибок, независимо от того, через какой пакет созданы её звенья.

Выразим в коде следствия выше (исходник примера):

```go
// https://goplay.tools/snippet/gHQoaMcc1Y-

wrappedErr := os.ErrNotExist

cases := []struct {
    title string
    err   error
}{
    {
        title: "only std errors",
```

```
        err:   fmt.Errorf("msg 2: %w", fmt.Errorf("msg 1: %w",
wrappedErr)),
    },
    {
        title: "only pkg/errors errors",
        err:   errors.Wrap(errors.Wrap(wrappedErr, "msg 1"), " msg
2"),
    },
    {
        title: "combined 1",
        err:   errors.Wrap(fmt.Errorf("msg 1: %w", wrappedErr), "msg
2"),
    },
    {
        title: "combined 2",
        err:   fmt.Errorf("msg 2: %w", errors.Wrap(wrappedErr, "msg
1")),
    },
}

for _, c := range cases {
    fmt.Println(c.title)
    fmt.Println("\terrors.Is:", errors.Is(c.err, wrappedErr))
    fmt.Println("\terrors.Cause:", errors.Cause(c.err) == wrappedErr)
    fmt.Println()
}

/*
only std errors
     errors.Is: true
     errors.Cause: false

only pkg/errors errors
     errors.Is: true
     errors.Cause: true

combined 1
     errors.Is: true
     errors.Cause: false

combined 2
     errors.Is: true
     errors.Cause: false

*/
```

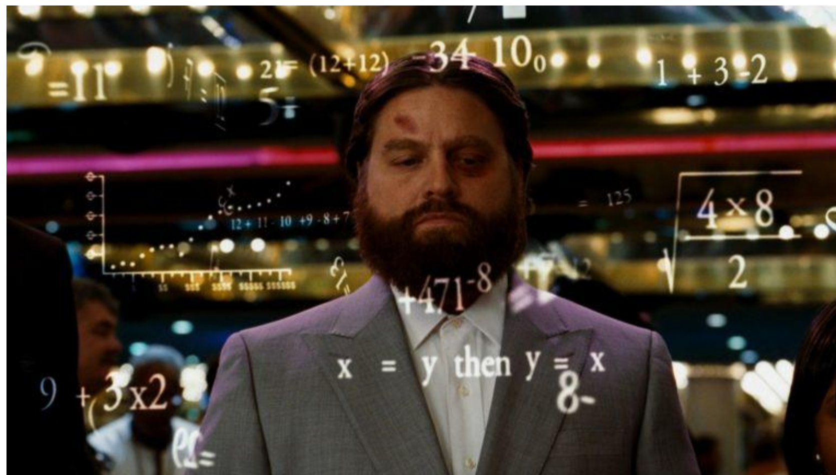
Что и требовалось доказать: для всех кейсов `errors.Is` возвращает `true`, а `errors.Cause` работает только для случая `"only pkg/errors errors"`.

Резюме

Если в вашем проекте для работы с ошибками используется как стандартная библиотека, так и **github.com/pkg/errors**, то для разворачивания ошибок лучше отказаться от `errors.Cause` в пользу `errors.Is` / `errors.As` (неважно стандартных функций или из **pkg/errors** – мы помним, что они идентичны).

Кроме того, если вы начинаете итеративно переходить с **pkg/errors** на **errors** и используете ошибки, созданные через последний, то будьте готовы, что имеющиеся в вашем проекте проверки через `errors.Cause` перестанут работать.

Тест "Cause и Is"



Перед вами функция `checkError`:

```
import (
    stderrors "errors"
    "fmt"

    "github.com/pkg/errors"
)

var errTarget = stderrors.New("target error")
```

```
func checkError(err error) {
    switch {
    case errors.Cause(err) == errTarget:
        fmt.Println("errors.Cause")

    case errors.Is(err, errTarget):
        fmt.Println("errors.Is")

    default:
        fmt.Println("default")
    }

}
```

Соотнесите вызов функции от определённой ошибки и сообщение на экране после этого.

Сопоставьте значения из двух списков

err := stderrors.New("target error")

err = fmt.Errorf("oops: %w", err)

checkError(err)

err := fmt.Errorf("oops: %w", errTarget)

checkError(err)

err := errors.Wrap(errTarget, "oops")

checkError(err)

- - - - - - - -

errors.Cause

default

errors.Is

Задача "Интерфейс Wrapper" (по мотивам реальной баги)

Ссылка на заготовку.

В проекте активно используется **github.com/pkg/errors** и все кастомные ошибки удовлетворяют интерфейсу

```
type causer interface {
    Cause() error

}
```

Разработчик реализовал новый тип для ошибки конкретного пользователя:

```
func NewUserError(err error, userID string) error {
    return &userError{
        err:    err,
        userID: userID,
    }
}

type userError struct {
    err    error
    userID string
}

func (ie *userError) Error() string {
    return fmt.Sprintf("user %s: %v", ie.userID, ie.err)
}

func (ie *userError) Cause() error {
    return ie.err
}

func (ie *userError) UserID() string {
    return ie.userID

}
```

И в мидлваре поддержал его через подход **opaque errors**:

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

// ...
```

```go
if err != nil {
    type withUserID interface {
        UserID() string
    }

    var i withUserID
    if errors.As(err, &i) {
        logger = logger.With(field.String("user_id", i.UserID()))
        // ...
    }

}
```

Через какое-то время выяснилось, что код внутри `if errors.As(err, &i) { ... }` никогда не выполнялся :(

Вопрос к вам – **почему?**

Тимлид зафиксил ошибку коллеги следующим образом:

```go
+var _ Wrapper = (*userError)(nil)

type userError struct {
    err    error
    userID string
}

+func (ie *userError) Unwrap() error {
+    return ie.err
+}
```

Обратим внимание на **interface assertion** в первой строчке. Вам необходимо описать интерфейс, позволяющий делать такое:

```go
// Wrapper требует от типа быть ошибкой, поддерживающей API
// как стандартной библиотеки, так и github/pkg/errors.

type Wrapper interface
```

Промежуточные выводы

В сухом остатке мы можем выделить следующие особенности **errors** и **github.com/pkg/errors**:

- Wrap-функции из **pkg/errors** поддерживают `nil`-ошибки, что не скажешь про `fmt.Errorf`.
- `errors.Errorf` не поддерживает `%w`.
- В зависимости от того, хотите ли вы пробрасывать ошибку наверх или нет, вам придётся выбирать между `errors.Wrap` или `errors.Errorf` + `%v` (при использовании стандартной библиотеки вы бы выбирали только между директивами `%w` и `%v`).
- Большинство функций из **pkg/errors** тянут за собой стектрейс (**в стандартной библиотеке стектрейса нет!**).
- `errors.Cause` **не является аналогом** `errors.Is` / `errors.As`, и вам придётся следить за реализуемыми ошибкой методами, чтобы поддержать все варианты.

Из этого следует, что просто так заменить одно на другое не выйдет, и при использовании обоих пакетов периодическое появление головной боли гарантировано.

И многие бы рады отказаться от **github.com/pkg/errors** в принципе, но стандартная библиотека не умеет в главное, что от неё очень хотят – **стектрейс**.

О нём мы и поговорим в следующем уроке.