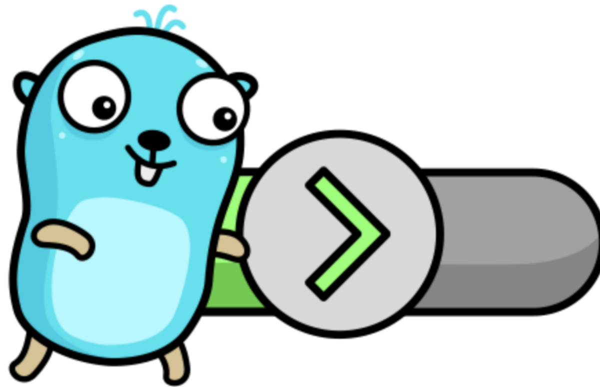
Врапинг ошибок после Go 1.13 (часть 2)

В этом уроке мы познакомимся с директивой формата `%w` в `fmt.Errorf`, а также порешаем задачи, которые помогут закрыть тему врапинга ошибок в Go.



В чём была проблема?

Мы помним, что проблема использования `fmt.Errorf` была в том, что после форматирования в `%v` обернутая ошибка становится просто текстом новой возвращаемой ошибки:

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

func loadFiles(urls ...string) ([]File, error) {
    files := make([]File, len(urls))
    for i, url := range urls {
        f, err := getFile(url)
        if err != nil {
            return nil, &FileLoadError{url, err} // <- Обернули
(заврапили) err в *FileLoadError.
        }
        files[i] = f
    }
    return files, nil
}

func transfer() error {
    files, err := loadFiles("www.golang-courses.ru")
    if err != nil {
```

```
        return fmt.Errorf("cannot load files: %v", err) //
*FileLoadError стал *errors.errorString.
    }

    // ...
    return nil
}

func handle() error {
    if err := transfer(); err != nil {
        return fmt.Errorf("cannot transfer files: %v", err)
    }

    // ...
    return nil
}

func main() {
    var fileLoadErr *FileLoadError
    if err := handle(); errors.As(err, &fileLoadErr) { // errors.As
не сработает!
        // ...
    }
}
```

И цепочки ошибок работают, только если мы отказываемся от врапинга в дополнительный текст, а звеньями цепочки делаем ошибки, реализующие метод `Unwrap`.

Не слишком удобно, не так ли?

fmt.Errorf + %w

Начиная с Go 1.13 `fmt.Errorf` поддерживает новую директиву `%w`. Когда она присутствует в строке форматирования, ошибка, возвращаемая `fmt.Errorf`, будет иметь метод `Unwrap`, возвращающий аргумент `%w`:

```
package fmt

// Errorf formats according to a format specifier and returns the
string as a
// value that satisfies error.
```

```
// ...
func Errorf(format string, a ...interface{}) error {
    p := newPrinter()
    p.wrapErrs = true
    p.doPrintf(format, a) // Обрабатываем строку формата.

    s := string(p.buf)
    var err error
    if p.wrappedErr == nil { // Если p.doPrintf выставил
p.wrappedErr,
        err = errors.New(s)
    } else {
        err = &wrapError{s, p.wrappedErr} // то возвращаем обёртку с
Unwrap.
    }

    p.free()
    return err
}

type wrapError struct { // <- Обёрточка.
    msg string
    err error
}

func (e *wrapError) Error() string {
    return e.msg
}

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

Теперь мы можем делать следующим образом (исходник примера):

```
// https://goplay.tools/snippet/yw-7P2UYiru

func transfer() error {
    _, err := loadFiles("www.golang-courses.ru")
    if err != nil {
        return fmt.Errorf("cannot load files: %w", err) // <- %w !!!
    }
```

```go
    // ...
    return nil
}

func handle() error {
    if err := transfer(); err != nil {
        return fmt.Errorf("cannot transfer files: %w", err) // <- %w
!!!
    }

    // ...
    return nil
}

func main() {
    var fileLoadErr *FileLoadError
    if err := handle(); errors.As(err, &fileLoadErr) {
        fmt.Println(fileLoadErr.URL) // www.golang-courses.ru
    }

}
```

Задача "Чиним opaque errors"

[Ссылка на заготовку](#).

В модуле про концепцию ошибок в Go мы познакомились с подходом **opaque errors**.

Сейчас мы кое-что узнали об оборачивании ошибок в стандартной библиотеке и можем заметить, что оно ломает код, который мы тогда [рассматривали](#):

```go
// IsTemporary говорит, является ли ошибка err временной.
func IsTemporary(err error) bool {
    type temporary interface {
        Temporary() bool
    }

    e, ok := err.(temporary) // Приводим ошибку к интерфейсу
temporary, тем самым проверяя поведение.
    return ok && e.Temporary()

}
```

Почему ломает? Потому что обёрнутая ошибка уже не реализует интерфейс `temporary` в отличие от оригинальной.

Вам необходимо реализовать `IsTemporary` так, чтобы она поддерживала обёрнутые ошибки.

Задача "Сломанная цепочка"

Ссылка на заготовку.

Перед нами функция `ProcessMessage`, реализующая цепочку вызовов `ProcessMessage -> process -> saveMsg`:

```
type Message struct {
    ID string
}

func ProcessMessage() error {
    msg := readMessageFromQueue()

    if err := process(msg); err != nil {
        return fmt.Errorf("cannot process msg: %w", err)
    }

    return nil
}

func readMessageFromQueue() Message {
    return Message{ID: "8fbad38c-c5c5-11eb-b876-1e00d13a7870"}
}

func process(msg Message) error {
    if err := saveMsg(msg); err != nil {
        return fmt.Errorf("cannot write data: %v", err)
    }
    return nil
}

func saveMsg(m Message) error {
    if true {
        return &saveMsgError{id: m.ID, err: io.ErrShortWrite}
    }
```

```go
    return nil
}
```

Ошибка `saveMsgError` выглядит следующим образом:

```go
type saveMsgError struct {
    id  string
    err error
}

func (w *saveMsgError) Error() string {
    return fmt.Sprintf("save msg %q error: %v", w.id, w.err)

}
```

Разработчик решил обработать ошибку от `ProcessMessage`, но у него ничего не вышло:

```go
err := ProcessMessage()
if errors.Is(err, io.ErrShortWrite) {
    // Не работает :(
}
```

Вам необходимо починить цепочку ошибок так, чтобы `errors.Is` "доставал" до самой глубокой ошибки.

Для этого необходимо:

- исправить функцию `process`;
- добавить методов типу `*saveMsgError`.

Текст сообщения финальной ошибки должен остаться без изменений.

Задача "WithTimeError"

Ссылка на заготовку.

К нашему сервису было выдвинуто требование, что необходимо знать точное время каждой произошедшей ошибки.

При этом время записи ошибки в лог не подходит, так как запись может произойти гораздо позже, чем ошибка случилась на самом деле (зависит от того, как далеко по стеку вызовов находится обработчик ошибки).

Вам необходимо реализовать тип `WithTimeError` и конструктор для него. Пример использования:

```
func ReadByteByByte(r io.Reader) error {
    buffer := make([]byte, 1)
    for {
        _, err := r.Read(buffer)
        if err != nil {
            return fmt.Errorf("cannot read: %w",
NewWithTimeError(err)) // <- Вызов конструктора.
        }
    }
}

func main() {
    err :=
ReadByteByByte(strings.NewReader("https://www.golang-courses.ru/"))
    fmt.Println(err) // cannot read: EOF, occurred at:
2021-06-07T20:48:39.478061+03:00
    fmt.Println(errors.Is(err, io.EOF)) // true
```

```
    var t interface {
        Time() time.Time
    }
    if errors.As(err, &t) {
        fmt.Println(t.Time()) // 2021-06-07 20:48:39.478061 +0300 MSK m=+0.000953043
    }
}
```

Таким образом `WithTimeError`:

- хранит время своего создания, которое можно получить через вызов метода `Time()`;
- является ошибкой (реализует интерфейс `error`), а в строке сообщения содержит оригинальную ошибку и точное время, когда она произошла;
- не ломает цепочку ошибок, поддерживает врапинг.

Вам доступны пакеты `fmt` , `errors` и `time`.

Врапинг чужой ошибки в свою

Подробнее об этом приёме мы поговорим в модуле лучших практик. Сейчас же просто ответим на вопрос "а как можно заврапить чужую ошибку в свою?".

Часто бывает, что вы хотите сохранить текст исходной ошибки, но при этом сделать так, чтобы пользователь функции работал с вашей, а не с этой нижележащей ошибкой (о которой он может и не знать).

Сделать это можно следующим образом:

```
var ErrSQLExec = errors.New("error while executing sql")

// ...
    res, err := db.Exec(query, args...)
    if err != nil {
        return fmt.Errorf("%w: %v", ErrSQLExec, err)
    }
```

```go
// ...
```

Так мы запрещаем работать с нижележащими ошибками от `db`, которых может быть неопределённое количество, но позволяем работать с нашей `ErrSQLExec`, тем самым принуждая пользователя не прыгать через границы API:

```go
if errors.Is(err, ErrSQLExec) {
    // ...
}
```

Сохранение же текста оригинальной ошибки без труда позволит нам отдебажить её:

```go
return fmt.Errorf("%w: %v", ErrSQLExec, err)

// error while executing sql: sql: expected 5 arguments, got 3
```

Задача "Ошибки валидации"

[Ссылка на заготовку](#).

Мы разрабатываем сервис поиска, принимающий на вход `SearchRequest`'ы, которые необходимо валидировать:

```go
const maxPageSize = 100

type SearchRequest struct {
    Exp      string
    Page     int
    PageSize int
}
```

Нужно реализовать:

```go
var (
    errIsNotRegexp      = ...
    errInvalidPage      = ...
    errInvalidPageSize = ...
)

func (r SearchRequest) Validate() error {
```

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

Метод `Validate` последовательно проверяет, что:

- `r.Exp` (поисковый запрос) является компилируемым регулярным выражением.
- `r.Page` (номер страницы) является положительным числом больше нуля.
- `r.PageSize` (размер страницы) является положительным числом больше нуля и не больше `maxPageSize`.

Предполагается, что ошибки валидации накапливаются и `Validate` возвращает

```
type ValidationErrors []error
```

При этом сохраняется поддержка `errors.Is` по конкретной ошибке:

```
if errors.Is(Validate(), errInvalidPageSize) {
    // ...
}
```

Все ошибки должны содержать дополнительный контекст:

- текст "чужой" ошибки, если она есть;
- значение поля, не прошедшего валидацию;
- константы, влияющие на валидацию, если имеются.

Например:

```
req := SearchRequest{
    Exp:      "(.*golang.*",
```

```
    Page:     -1,
    PageSize: 3000,
}

err := req.Validate()
fmt.Println(err.Error())
// validation errors:
//      exp is not regexp: error parsing regexp: missing closing ):
`(.*golang.*`
//      invalid page: -1

//      invalid page size: 3000 > 100
```

Пакеты, доступные для использования, указаны в заготовке.

Особенности использования %w

Давайте посмотрим на полное описание к fmt.Errorf:

```
package fmt

import "errors"

// Errorf formats according to a format specifier and returns the
string as a
// value that satisfies error.
//
// If the format specifier includes a %w verb with an error operand,
// the returned error will implement an Unwrap method returning the
operand. It is
// invalid to include more than one %w verb or to supply it with an
operand
// that does not implement the error interface. The %w verb is
otherwise
// a synonym for %v.

func Errorf(format string, a ...interface{}) error {
```

А именно обратим внимание на

```
It is invalid to include more than one %w verb or to supply it with
an operand that does not implement
```

`the error interface. The %w verb is otherwise a synonym for %v.`

То есть:

- использовать `%w` можно только в `fmt.Errorf`;
- нельзя включать более одного `%w` в строку формата;
- нельзя под `%w` подкладывать аргумент, не реализующий `error`;
- форматирование по `%w` ничем не отличается по форматированию по `%v`.

Тезисы выше подтверждаются исходным кодом пакета `fmt`:

```
// src/fmt/print.go

func (p *pp) handleMethods(verb rune) (handled bool) {
    if p.erroring {
        return
    }
    if verb == 'w' {
        // It is invalid to use %w other than with Errorf, more than
once,
        // or with a non-error arg.
        err, ok := p.arg.(error)
        if !ok || !p.wrapErrs || p.wrappedErr != nil {
            p.wrappedErr = nil
            p.wrapErrs = false
            p.badVerb(verb)
            return true
        }
        p.wrappedErr = err
        // If the arg is a Formatter, pass 'v' as the verb to it.
        verb = 'v'
    }

    // ...

}
```

---

1) Более одного %w ломает врапинг в принципе

```
// https://goplay.tools/snippet/65OSkUr4ZPc

func main() {
    err := fmt.Errorf("cannot do operation: %w with %w", io.EOF,
context.Canceled)
    fmt.Println(err)
    // cannot do operation: EOF with
%!w(*errors.errorString=&{context canceled})

    fmt.Println(errors.Is(err, io.EOF))             // false !!!
    fmt.Println(errors.Is(err, context.Canceled)) // false
}
```

IDE на страже нашего кода:

```
func main() {
    err := fmt.Errorf( format: "cannot do operation: %w with %w", io.EOF, context.Canceled)
    fmt.Println(err)                              // cannot d
    fmt.Println(errors.Is(err, io.EOF))           // false
```

This verb can be used only once in a format string (%w)

---

2) %w не в Errorf не работает

При этом форматируется не как `%v`, а как badVerb:

```
// https://goplay.tools/snippet/vhTApI-T8Nz

func main() {
    fmt.Printf("cannot do operation: %w", context.Canceled)
    // cannot do operation: %!w(*errors.errorString=&{context
canceled})
}
```

```
func main() {
    fmt.Printf( format: "cannot do operation: %w", context.Canceled)
    // cannot do operation: %!w(*errors.error
}
```

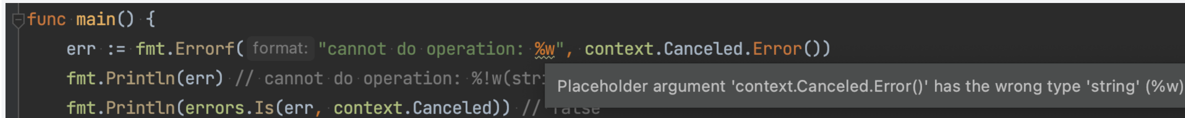
This verb can be used only in 'fmt.Errorf' calls (%w)

---

3) Если аргумент %w не является ошибкой, то директива работать не будет

```
// https://goplay.tools/snippet/EcFcy-e0d3k

func main() {
    err := fmt.Errorf("cannot do operation: %w",
context.Canceled.Error())
    fmt.Println(err) // cannot do operation: %!w(string=context
canceled)
    fmt.Println(errors.Is(err, context.Canceled)) // false
}
```



---

4) %w "скушает" nil-ошибку

```
// https://goplay.tools/snippet/ylU4VqjAnJe

func main() {
    err := fmt.Errorf("cannot do operation: %w", nil)
    fmt.Println(err)        // cannot do operation: %!w(<nil>)
    fmt.Println(err == nil) // false

}
```

При этом на выходе мы получим не-`nil` ошибку, работать с которой хоть каким-то приличным образом в целом невозможно.

Задача "Errorf на стероидах"

[Ссылка на заготовку](#).

Необходимо реализовать функцию `Errorf`, аналогичную по своей сути `fmt.Errorf`, но поддерживающую неограниченное количество директив `%w`:

```
func Errorf(format string, args ...any) error
```

Пример использования:

```
err := Errorf("cannot load file %q: %w through %w",
    "file.txt",
    &net.AddrError{Err: "err", Addr: "0.0.0.0:4242"},
    syscall.ECONNREFUSED)

fmt.Println(err)
// cannot load file "file.txt": address 0.0.0.0:4242: err through
connection refused

fmt.Println(errors.Is(err, syscall.ECONNREFUSED)) // true

var addrErr *net.AddrError
fmt.Println(errors.As(err, &addrErr)) // true
fmt.Println(addrErr.Addr) // 0.0.0.0:4242
```

Требования к функции:

- Поддержка множества `%w`.
- Для каждого аргумента `%w` должен работать `errors.Is` и `errors.As`.
- Поддержка простых (однобуквенных) [спецификаторов]() форматирования;
- Если все аргументы `%w` являются `nil`, то `Errorf` должен возвращать `nil`.
- Если хотя бы один из аргументов `%w` не является `nil`, то функция возвращает ошибку, удовлетворяющую требованиям выше, а в итоговой строчке `nil`-аргументы отображаются дефолтным (`%v`) форматированием для `nil` (`"<nil>"`).
- Если аргумент `%w` не является ошибкой, то он форматируется дефолтным (`%v`) для него значением.

Для полного понимания обратитесь к тестам в заготовке.

Гарантируется, что в вызовах `Errorf` количество аргументов и количество спецификаторов в строке формата совпадают.

Вам доступны пакеты: `errors`, `fmt`, `regexp` и `strings`.

Если каких-то пакетов для решения вам не хватает – пишите нам, мы добавим :)

Космолётов изобретать не нужно - достаточно, чтобы тесты в заготовке проходили.

Подведём итоги

Таким образом современная работа с ошибками в Go сводится к:

- созданию ошибок-значений уровня пакета – **sentinel errors**;
- созданию собственных типов ошибок, при этом типы должны реализовывать метод `Unwrap` (опционально методы `Is` и `As`), чтобы не ломать цепочки, в которых находятся;
- добавлению контекста ошибки на очередном уровне с помощью `fmt.Errorf` и спецификатора `%w`:

```
hoverRng, err := spn.Range(pm.Mapper.Converter)
if err != nil {
    return nil, fmt.Errorf("computing hover range: %w", err)
}
```

- работе с ошибками на более высоком уровне с помощью `errors.Is` и `errors.As` (никаких `==` и **type asserts** как раньше):

```
if errors.Is(err, cmdflag.ErrFlagTerminator) {
    // ...
}
```

```go
if nf := (cmdflag.NonFlagError{}); errors.As(err, &nf) {
    args = append(args, nf.RawArg)
    // …
}
```