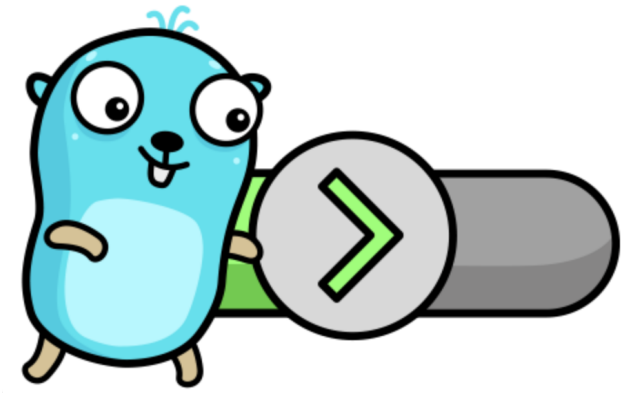
Врапинг ошибок после Go 1.13 (часть 1)

В этом уроке мы поговорим о трёх китах современного врапинга ошибок в Go – функциях `errors.Unwrap`, `errors.Is` и `errors.As`.

Разберёмся, как они реализованы, как ими пользоваться и как создавать "приятные" цепочки ошибок в Go, не теряя возможности работать с исходными ошибками на более высоком уровне.



errors.Unwrap

Начиная с версии 1.13 в стандартном пакете `errors` появился метод Unwrap:

```
// Unwrap returns the result of calling the Unwrap method on err, if
err's
// type contains an Unwrap method returning error.
// Otherwise, Unwrap returns nil.
func Unwrap(err error) error {
    u, ok := err.(interface {
        Unwrap() error
    })
    if !ok {
        return nil
    }
    return u.Unwrap()

}
```

Как мы видим, он регламентирует правила "разворачивания" ошибки: если ошибка хранит в себе родительскую ошибку, то она должна реализовать её получение через `Unwrap`.

Тип `FileLoadError` из предыдущего урока хоть и хранит родительскую ошибку, но разворачивать себя не умеет:

```
// https://goplay.tools/snippet/vm372Yb3V_y

type FileLoadError struct {
    URL string
    Err error
}

func (p *FileLoadError) Error() string {
    return fmt.Sprintf("%q: %v", p.URL, p.Err)
}

func main() {
    err := &FileLoadError{Err: context.Canceled}
    fmt.Println(errors.Unwrap(err)) // <nil>
}
```

Мы можем легко исправить это, реализовав `(*FileLoadError).Unwrap`:

```
// https://goplay.tools/snippet/L7I-oYlJMIy

type FileLoadError struct {
    URL string
    Err error
}

func (p *FileLoadError) Error() string {
    return fmt.Sprintf("%q: %v", p.URL, p.Err)
}

func (p *FileLoadError) Unwrap() error {
    return p.Err
}

func main() {
```

```
    err := &FileLoadError{Err: context.Canceled}
    fmt.Println(errors.Unwrap(err)) // context canceled
}
```

Таким образом, теоретически мы можем развернуть всю цепочку ошибок до конца, если каждая ошибка в ней умеет `Unwrap`'ить себя.

Задача "Unwrap Loop"

Ссылка на заготовку.

Необходимо реализовать функцию

```
func Unwrap(err error) error
```

только более сложную, чем в стандартной библиотеке – функция должна разворачивать ошибку, пока это возможно и возвращать наиболее нижележащий результат.

Вам доступен интерфейс `Unwrapper` (определять не нужно):

```
type Unwrapper interface {
    Unwrap() error

}
```

errors.Is

По принципу из предыдущей задачи работает появившаяся после 1.13 функция errors.Is: она разворачивает всю цепочку ошибок и проверяет, нет ли среди звеньев цепочки искомой ошибки:

```
package errors

// Is reports whether any error in err's chain matches target.
func Is(err, target error) bool {
    if target == nil {
        return err == target
    }

    isComparable := reflectlite.TypeOf(target).Comparable()
    for {
        // Если ошибки равны по значению, то это успех.
        if isComparable && err == target {
            return true
        }

        // Если проверяемая ошибка err реализует метод Is, то
сравниваем через него.
        if x, ok := err.(interface{ Is(error) bool }); ok &&
x.Is(target) {
            return true
        }

        // Если дальше не разворачивается, то прерываем цикл,
        // иначе продолжаем сравнение с развёрнутой ошибкой.
        if err = Unwrap(err); err == nil {
            return false
        }
    }

}
```

Теперь вместо конструкции

```
if fileLoadErr, ok := err.(FileLoadError); ok && fileLoadErr.Err ==
context.Canceled {
    // ...

}
```

мы можем писать

```
if errors.Is(err, context.Canceled) {
    // ...

}
```

и соответственно

```
switch {
case errors.Is(err, context.Canceled):
    // ...
case errors.Is(err, io.EOF):
    // ...

}
```

– это что касается работы с ошибками-значениями (***sentinel errors***), т.е. `errors.Is` можно считать эквивалентом `==`.

Но имейте в виду, что `errors.Is` очевидно не отработает, если ошибка обёрнута в текст через `%v`:

```
// https://goplay.tools/snippet/-jU55Y1rNvY
package main

import (
    "errors"
    "fmt"
    "io"
)

func main() {
    err := io.EOF

    fmt.Println(
        errors.Is(err, io.EOF)) // true
    fmt.Println(
        errors.Is(fmt.Errorf("cannot read file: %v", err), io.EOF))
// false
}
```

Как это обойти мы узнаем в следующем уроке.

Переопределение поведения errors.Is

Из реализации `errors.Is` мы видим, что для переопределения поведения этой функции мы должны реализовать соответствующий интерфейс:

```
// ...
if x, ok := err.(interface{ Is(error) bool }); ok && x.Is(target) {
    return true
}

// ...
```

В стандартной библиотеке не так много примеров подобного:

```
// https://github.com/golang/go master
$ grep -r --exclude "*_test.go" ") Is(" .
./api/go1.13.txt:pkg syscall, method (Errno) Is(error) bool
./src/cmd/go/internal/modfetch/cache.go:func (e
*DownloadDirPartialError) Is(err error) bool { return err ==
fs.ErrNotExist }
./src/cmd/go/internal/modfetch/codehost/git.go:func (notExistError)
Is(err error) bool { return err == fs.ErrNotExist }
./src/cmd/go/internal/modfetch/codehost/codehost.go:func
(UnknownRevisionError) Is(err error) bool {
./src/cmd/go/internal/modfetch/codehost/codehost.go:func
(noCommitsError) Is(err error) bool {
./src/cmd/go/internal/modfetch/repo.go:func (notExistError) Is(target
error) bool {
./src/cmd/go/internal/modload/modfile.go:func (e *excludedError)
Is(err error) bool { return err == ErrDisallowed }
./src/cmd/go/internal/modload/modfile.go:func (e
*ModuleRetractedError) Is(err error) bool {
./src/cmd/go/internal/web/api.go:func (e *HTTPError) Is(target error)
bool {
./src/errors/wrap.go://    func (m MyError) Is(target error) bool {
return target == fs.ErrExist }
./src/syscall/syscall_plan9.go:func (e ErrorString) Is(target error)
bool {
./src/syscall/syscall_js.go:func (e Errno) Is(target error) bool {
./src/syscall/syscall_windows.go:func (e Errno) Is(target error) bool
{
```

```
./src/syscall/syscall_unix.go:func (e Errno) Is(target error) bool {
```

Наиболее ярким примером является `syscall.Errno`:

```
// go/src/syscall/zerrors_darwin_amd64.go

// Errors
const (
    E2BIG           = Errno(0x7)
    EACCES          = Errno(0xd)
    EADDRINUSE      = Errno(0x30)
    EADDRNOTAVAIL   = Errno(0x31)
    EAFNOSUPPORT    = Errno(0x2f)
    // ...
)

// Error table
var errors = [...]string{
    1:   "operation not permitted",
    2:   "no such file or directory",
    3:   "no such process",
    4:   "interrupted system call",
    // ...
}

// go/src/syscall/syscall_unix.go

// An Errno is an unsigned number describing an error condition.
type Errno uintptr

func (e Errno) Error() string {
    if 0 <= int(e) && int(e) < len(errors) {
        s := errors[e]
        if s != "" {
            return s
        }
    }
    return "errno " + itoa(int(e))
}

func (e Errno) Is(target error) bool {
    switch target {
    case oserror.ErrPermission:
        return e == EACCES || e == EPERM
    case oserror.ErrExist:
```

```
        return e == EEXIST || e == ENOTEMPTY
    case oserror.ErrNotExist:
        return e == ENOENT
    }
    return false

}
```

Мы видим, что с помощью `(Errno).Is` мы можем приравнять коды `errno` к нормальным православным ошибкам:

```
// go/src/internal/oserror/errors.go
package oserror

import "errors"

var (
    ErrInvalid    = errors.New("invalid argument")
    ErrPermission = errors.New("permission denied")
    ErrExist      = errors.New("file already exists")
    ErrNotExist   = errors.New("file does not exist")
    ErrClosed     = errors.New("file already closed")
)
```

С `errno` мы сталкивались ранее в первом модуле курса.

Обратим внимание, как изящно реализована поддержка данной переменной в Golang (где она стала типом, по сути слившись с `errno_t`).

Тест "Соотнесите выражение и его результат"

Сопоставьте значения из двух списков

syscall.EACCES == fs.ErrPermission

errors.Is(syscall.EACCES, fs.ErrPermission)

errors.Unwrap(syscall.ENOENT)

---

false

true

<nil>

Задача "errors.Is"

Ссылка на заготовку.

Разработчик реализовал ошибку `PipelineError`, имеющую следующий вид

```
type PipelineError struct {
    User        string
    Name        string
    FailedSteps []string
}

func (p *PipelineError) Error() string {
    return fmt.Sprintf("pipeline %q error", p.Name)

}
```

Затем ему потребовалось работать с ошибками конкретного пайплайна конкретного пользователя, независимо от того, какие шаги в этом пайплайне были неуспешны (`FailedSteps`), но данный код не сработал:

```
bobCalculationErr := &PipelineError{
    User: "Bob",
    Name: "bitcoin calculation",
}
```

```
if errors.Is(err, bobCalculationErr) {
    // Никогда не выполнится, почему?
    // ...

}
```

Необходимо дописать новый метод к `PipelineError` так, чтобы `errors.Is` считала два `*PipelineError` эквивалентными, если имена пайплайнов и их пользователи совпадают:

- писать определение структуры не нужно, она будет вам доступна;
- соответственно нельзя менять существующие методы структуры или саму структуру.

errors.As

Если же нам необходимо получить значение ошибки в соответствии с её типом (для более сложных ошибок (содержащих дополнительную информацию, строящихся по факту и не выделенных в отдельные переменные), таких как наш `FileLoadError`, `net.UnknownNetworkError` и т.д.), то следует использовать новую относительно 1.13 функцию errors.As:

```
package errors

// As finds the first error in err's chain that matches target, and
if so, sets
// target to that error value and returns true. Otherwise, it returns
false.
// ...

func As(err error, target interface{}) bool
```

Т.е. `errors.As`:

- принимает два аргумента – ошибку `err` и указатель на место для хранения результата `target`;
- начиная с `err` разворачивает цепочку ошибок, пока не встретит значение, которое можно положить в `target`:

- если `target` указатель на интерфейс (необязательно `error`), то значение, хранящееся в `err`, должно реализовывать его;
- если `target` указатель на переменную иного типа, то ошибка ожидается такого же типа (а точнее см. [правила возможности присвоения]).

```
// ...
var e FileLoadError
if errors.As(err, &e) { // Ранее: if lfe, ok := err.(FileLoadError);
ok {
    log.Println(e.URL)
}

// ...

// ...
var e *os.PathError
if errors.As(err, &e) { // <- Указатель на указатель!
    fmt.Println(e.Op, e.Path)
}

// ...
```

Прокомментируем непростой [исходный код этой функции], полный [рефлексии], чтобы детальнее понять, что в ней происходит:

```
// go/src/errors/wrap.go
package errors

import "internal/reflectlite"

// Интересный пример, как с помощью рефлексии получить reflect.Type
интерфейса.
var errorType = reflectlite.TypeOf((*error)(nil)).Elem()

func As(err error, target interface{}) bool {
    if target == nil {
        panic("errors: target cannot be nil")
    }
```

```
    // Проверяем, что target - ненулевой указатель.
    val := reflectlite.ValueOf(target)
    typ := val.Type()
    if typ.Kind() != reflectlite.Ptr || val.IsNil() {
        panic("errors: target must be a non-nil pointer")
    }

    // Проверяем, что target является интерфейсом или значением,
реализующим интерфейс error.
    if e := typ.Elem(); e.Kind() != reflectlite.Interface &&
!e.Implements(errorType) {
        panic("errors: *target must be interface or implement error")
    }

    // Unwrap'им err до тех пор, пока не встретим ненулевую ошибку,
    // значение которорой можно записать в target.
    targetType := typ.Elem()
    for err != nil {
        if reflectlite.TypeOf(err).AssignableTo(targetType) {
            val.Elem().Set(reflectlite.ValueOf(err))
            return true
        }
        if x, ok := err.(interface{ As(interface{}) bool }); ok &&
x.As(target) {
            return true
        }
        err = Unwrap(err)
    }
    return false

}
```

Использование errors.Is и errors.As

Таким образом, если `errors.Is` можно считать заменой `==`, то `errors.As` можно считать заменой проверок ошибок через type asserting. Теперь вместо конструкции

```
if lfe, ok := err.(FileLoadError); ok {
    switch v := lfe.Err.(type) {
    case *net.AddrError:
        // Работаем с v как с *net.AddrError.
    case net.UnknownNetworkError:
        // Работаем с v как с net.UnknownNetworkError.
```

```
    }
}
```

мы можем писать

```
var (
    netAddrErr    *net.AddrError
    unknownNetErr net.UnknownNetworkError
)
switch {
case errors.As(err, &netAddrErr):
    // Работаем с netAddrErr.
case errors.As(err, &unknownNetErr):
    // Работаем с unknownNetErr.

}
```

Если же мы хотим одновременно проверять и ошибки-значения и ошибки-типы, то "миксуем" известные нам методы:

```
var (
    netAddrErr    *net.AddrError
    unknownNetErr net.UnknownNetworkError
)
switch {
case errors.As(err, &netAddrErr):
    // ...
case errors.As(err, &unknownNetErr):
    // ..
case errors.Is(err, context.Canceled):
    // ...
case errors.Is(err, io.EOF):
    // ...

}
```

При этом, если нам не нужно работать с дополнительной информацией в сложной ошибке, а важно лишь знание о том, что произошла именно она, то код можно сократить:

```
var n *net.AddrError
switch {
case errors.As(err, &n):
```

```
    // ...
case errors.As(err, new(net.UnknownNetworkError)): // Результат
"улетит" во временную переменную.
    // ..
case errors.Is(err, context.Canceled):
    // ...
case errors.Is(err, io.EOF):
    // ...

}
```

Аналогично `errors.Is` для переопределения поведения `errors.As` следует реализовать соответствующий интерфейс:

```
// ...
if x, ok := err.(interface{ As(interface{}) bool }); ok &&
x.As(target) {
    return true
}

// ...
```

К сожалению, в данный момент стандартная библиотека не имеет примеров подобного:

```
// https://github.com/golang/go master
$ grep -r --exclude "*_test.go" ") As(" . | wc -l

       0
```

Задача "errors.As"

Ссылка на заготовку.

В компании широко используется доменная ошибка `UserError`:

```
type UserError struct {
    Operation string
    User      string
}

func (u *UserError) Error() string {
```

```
    return u.User + ": " + u.Operation

}
```

Тимлид попросил разработчика пайплайна добавить поддержку этой ошибки для знакомого нам `PipelineError`.

А именно, следующий код должен работать:

```
bobCalculationErr = &PipelineError{
    User: "Bob",
    Name: "bitcoin calculation",
}

var userErr *UserError
if errors.As(bobCalculationErr, &userErr) {
    fmt.Printf("%#v\n", userErr) //
&main.UserError{Operation:"bitcoin calculation", User:"Bob"}

}
```

Для этого необходимо дописать новый метод к `*PipelineError`:

- писать определение структуры не нужно, она будет вам доступна;
- соответственно нельзя менять существующие методы структуры или саму структуру;
- аналогично для структуры `UserError`.

Подсказки:

- Реализация `errors.As` из стандартной библиотеки здесь не поможет – не нужно углубляться в `reflect` и прочие страсти.
- Обратите внимание на **тип** значения, приходящего в аргумент `target` вашего нового метода `As`. Постарайтесь ответить на вопрос, почему он именно такой и как нам это пригодится?

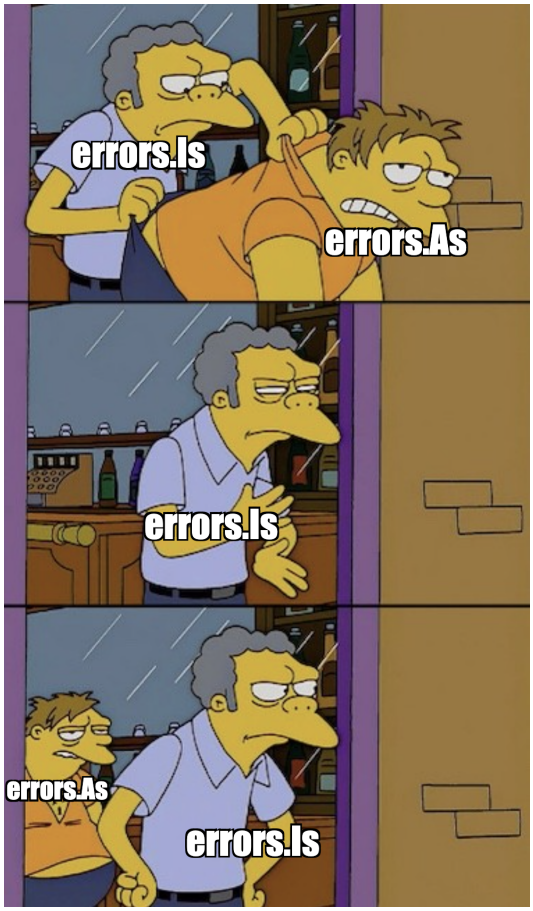
- Будьте бдительны: изменить указатель и изменить значение по указателю – это разные вещи!

Пара ссылочек по теме:

- GeeksForGeeks: Go Pointer to Pointer (Double Pointer)
- Stack Overflow: thwd: What are pointers to pointers good for?

Задача "errors.Is через errors.As"

Ссылка на заготовку.



Теперь, когда мы знакомы с `errors.As`, зададимся вопросом:

Как узнать, что полученная ошибка – это `*PipelineError` конкретного пайплайна конкретного пользователя, не имея при этом доступа к самой ошибке и возможности добавить ей метод `Is` (например, ошибка определена в чужой библиотеке)?

```
bobCalculationErr := &PipelineError{
    User: "Bob",
    Name: "bitcoin calculation",
}
if errors.Is(err, bobCalculationErr) {
    // Работать не будет. А добавить метод Is для *PipelineError возможности нет.
    // ...

}
```

Вам необходимо реализовать функцию

```
func IsPipelineError(err error, user, pipelineName string) bool
```

которая будет говорить, является ли входная ошибка `*PipelineError` пользователя `user` и пайплайна `pipelineName`.

- писать определение структуры не нужно, она будет вам доступна;
- добавлять новые методы к `PipelineError` нельзя;
- вам доступен пакет `errors`.

Тест "Из пустого в порожнее"

До этого момента мы использовали в качестве `target` в `errors.As` конкретные типы, но функция позволяет использовать в качестве выходного значения и интерфейсы, например:

```
var t interface {
    Timeout() string
}
if errors.As(err, &t) { // <- Развернёт err до значения, которое можно положить в интерфейс t.
    fmt.Println(t.Timeout())
```

```
}
```

По сути – это реализация **opaque errors** через `errors.As`.

Далее в курсе мы ещё увидим примеры подобного.

Сейчас же постараемся ответить на **непростой** вопрос: а что выведет данный код?

```
type PipelineError struct {
    User        string
    Name        string
    FailedSteps []string
}

func (p *PipelineError) Error() string {
    return fmt.Sprintf("pipeline %q error", p.Name)
}

func main() {
    var p error = &PipelineError{}
    if errors.As(net.UnknownNetworkError("tdp"), &p) {
        fmt.Println(p)
    } else {
        fmt.Println("As() return false")
    }
}
```

Понимание примера выше позволяет избегать неприятных багов и даже паник.

Промежуточные итоги

Мы познакомились с рядом функций стандартной библиотеки, призванных упростить нам работу с ошибками, и поняли, что

1) Вместо явного сравнения с ошибкой следует использовать `errors.Is`:

```
// Было.
if err == context.Canceled {
    // ...
```

```
}

// Стало.
if errors.Is(err, context.Canceled) {
    // ...

}
```

2) Вместо явного приведения к типу следует использовать `errors.As`:

```
// Было.
if fileLoadErr, ok := err.(*FileLoadError); ok {
    // ...
}

// Стало.
var fileLoadErr *FileLoadError
if errors.As(err, &fileLoadErr) {
    // ...

}
```

3) Чтобы `errors.Is` и `errors.As` работали корректно – все ошибки в цепочке должны уметь "разворачивать" себя через собственный метод `Unwrap` или у них должны быть определены методы `Is` / `As`.

Для чего всё это нужно? Чтобы уметь оборачивать ошибки в дополнительный контекст и при этом не терять возможности их сравнения с конкретными типами / значениями. А как это делать удобным способом с помощью стандартной библиотеки – мы узнаем в следующем уроке.