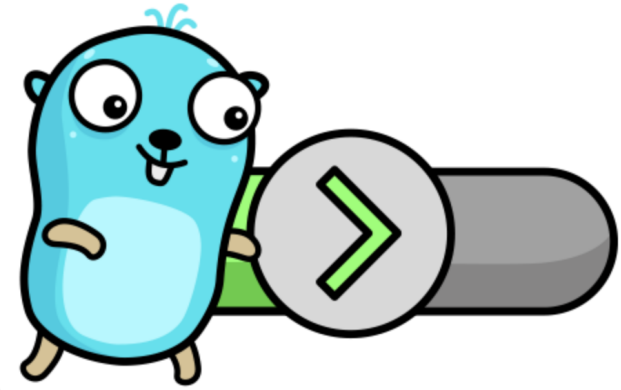
Ошибки и тип ресивера

В этом уроке мы узнаем, зачем для "ошибочных" типов выбирают сигнатуру метода `Error` с ресивером-указателем и что будет, если этого не делать.



Value Receiver VS Pointer Receiver

Согласно гошной спецификации методом является функция, имеющая так называемый **receiver (ресивер)**, связывающий метод с базовым типом, которому он "принадлежит". Ресивер может иметь тип аналогичный базовому типу (**value receiver**) или быть указателем на него (**pointer receiver**):

```
// https://goplay.tools/snippet/rT1Q6e2QE6e

type Cat struct {
    name string
}

// Name - метод, возвращающий имя кота.
// Имеет value receiver (с типа Cat).
func (c Cat) Name() string {
    return c.name
}

// Rename - метод, меняющий имя кота.
// Имеет pointer receiver (с типа *Cat).
func (c *Cat) Rename(name string) {
    c.name = name
}
```

```
func main() {
    var c Cat
    c.Rename("Boris")
    fmt.Println(c.Name()) // Boris

}
```

Есть [десяток подсказок](#) от разработчиков Go по теме того, какой тип ресивера выбрать в конкретном случае, но тот же

"A Tour Of Go" [упрощает](#) нам ситуацию и называет две основные причины использовать ресивер по указателю вместо ресивера по значению:

- если необходимо изменять внутренние поля структуры, на которую ссылается указатель;
- если структура "тяжелая", и мы хотим избежать её копирования при каждом вызове метода.

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

type Cat struct {
    name string
}

func NewCat(n string) Cat {
    return Cat{name: n}
}

func (c Cat) Name() string {
    return c.name
}

// RenameValue не изменит c.Name: при каждом вызове метода c будет копироваться.
func (c Cat) RenameValue(name string) {
    c.name = name
}
```

```go
// RenamePointer изменит c.Name: копирования не будет.
func (c *Cat) RenamePointer(name string) {
    c.name = name
}

func main() {
    cat := NewCat("Garfield")

    cat.RenameValue("Leopold")
    fmt.Println(cat.Name()) // Garfield

    cat.RenamePointer("Leopold")
    fmt.Println(cat.Name()) // Leopold

}
```

Тест "Выберите наиболее подходящий для ситуации тип ресивера"

Отметьте верные ячейки

	Используем Value Receiver	Используем Pointer Receiver
Делаем метод над типами map, chan или func.		
Делаем метод над слайсом, при этом внутри не переаллоцируем его.		
Метод должен менять свой инстанс.		
Делаем метод над типом, содержащим примитивы синхронизации (например, sync.Mutex)		
Делаем метод над массивом или структурой с большим количеством полей.		
Функции внутри нашего метода должны менять его инстанс.		
Делаем метод над массивом, слайсом или структурой, содержащими указатели на что-то, что должно меняться.		
Делаем метод над примитивом (int, string и т.д.) и метод не должен менять его.		

Делаем метод над "небольшой" структурой и метод не должен менять её.

Мы сомневаемся, какой тип ресивера выбрать.

Влияние ресивера на использование инстанса через интерфейс

Согласно спецификации необходимым условием для реализации типом интерфейса является факт, что множество методов (**method set**) интерфейса входит в множество методов типа — логично, не правда ли?

Более того к типу `T` относятся все методы, имеющие только **value receiver** `T`; а вот к типу `*T` относятся как методы с **value receiver** `T`, так и с **pointer receiver** `*T` (исходник примера):

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

type HTTPError interface {
    error
    StatusCode() int
}

type NotFoundError struct {
}

func (n *NotFoundError) Error() string {
    return "Not Found"
}

func (n NotFoundError) StatusCode() int {
    return http.StatusNotFound
}

/*
Тип NotFoundError имеет множество методов T:
- StatusCode

Тип *NotFoundError имеет множество методов T + *T:
- Error
- StatusCode

Как следствие именно *NotFoundError реализует интерфейс HTTPError.
```

```
*/
func main() {
    var err HTTPError = &NotFoundError{}
    fmt.Println(err.StatusCode(), err.Error()) // 404 Not Found

    // Не скомпилируется:
    // var _ HTTPError = NotFoundError{}
    // cannot use NotFoundError{} (type NotFoundError) as type
HTTPError in assignment:
    //     NotFoundError does not implement HTTPError (Error method
has pointer receiver)
}
```

Тип `NotFoundError` не реализует интерфейс `HTTPError` (в отличие от `*NotFoundError`), потому что не имеет метода `Error` (у которого ресивер-указатель).

Связано это с внутренним устройством интерфейса и неадресуемостью значения, которое в нём лежит (компилятор не может сам взять адрес от `T`, сделав его `*T`, но может произвести обратную операцию). Но подобные тонкости выходят за рамки данного курса.

Из особенностей выше следуют следующие лучшие практики:

1) Старайтесь не смешивать типы ресиверов у методов: если появился метод с ресивером-указателем – используйте везде ресивер-указатель:

Don't mix receiver types. Choose either pointers or struct types for all available methods.

2) Старайтесь делать так, чтобы конструкторы возвращали указатель на тип. Так вы минимизируете вероятность потери части методов "объекта", когда пользователь конструктора положит новый "объект" в интерфейс (исходник примера):

```
// https://goplay.tools/snippet/WIeJ8OtA1ff

func NewNotFoundError() (*NotFoundError, error) {
    return new(NotFoundError), nil
```

```
}

func main() {
    var httpErr HTTPError
    httpErr, _ = NewNotFoundError()
    fmt.Println(httpErr.StatusCode(), httpErr.Error()) // 404 Not
Found

}
```

Задача "PointerError"

[Ссылка на заготовку.]

Разработчик написал следующий код:

```
type PointerError struct {
    msg string
}

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

func NewPointerError(m string) PointerError {
    return PointerError{msg: m}

}
```

Но что-то пошло не так и ошибки с одинаковым текстом стали считаться эквивалентными:

```
p1 := NewPointerError("sky is falling")
p2 := NewPointerError("sky is falling")

fmt.Println(p1 == p2) // true
```

Вам необходимо исправить конструктор `NewPointerError` таким образом, чтобы код в заготовке компилировался и любые две ошибки, созданные с помощью конструктора, считались различными.

Возвращаясь к ошибкам

Так как быть с ошибками? Какие ресиверы делать у своих типов ошибок?

```
var ErrNotFound = NewStatusError(http.StatusNotFound, "Not Found")

type StatusError struct {
    code int
    msg  string
}

func (s StatusError) Error() string {
    return fmt.Sprintf("%d %s", s.code, s.msg)
}

func NewStatusError(code int, msg string) StatusError {
    return StatusError{code: code, msg: msg}

}
```

Нам нужно выбрать ресивер у метода `(StatusError).Error`.

Вернёмся к советам из "A Tour Of Go" и ответим на пару вопросов:

1) Модифицируем ли мы в `Error` сообщение или код? Нет, поля воспринимаем только как read-only: они инициализируются при создании экземпляра структуры и далее не меняются.

2) Хотим ли мы экономить на копировании структуры при вызове `Error`? Не хотим – копирование 24-х байт (на 64-разрядных машинах) мы переживём.

Достаточно ли этого, чтобы раз и навсегда решить, какие ресиверы делать у методов кастомных ошибок?

Наверное, да. Всё выглядит так, что стоит сделать **value receiver** и закрыть этот вопрос:

```
func (s StatusError) Error() string
```

А что там в библиотеках?

Логическим путём мы пришли к **value receiver** в ошибках, но мы помним, что пакеты для работы с ошибками имеют обратную ситуацию: там у аналогичных по составу и смыслу структур используется **pointer receiver**:

errors

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

// New returns an error that formats as the given text.
// Each call to New returns a distinct error value even if the text
is identical.
func New(text string) error {
    return &errorString{text}
}

// errorString is a trivial implementation of error.
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s

}
```

fmt

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

func Errorf(format string, a ...interface{}) error {
    // ...
    if p.wrappedErr == nil {
        err = errors.New(s)
    } else {
        err = &wrapError{s, p.wrappedErr}
    }
    // ...
}

type wrapError struct {
    msg string
    err error
}

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

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

}
```

github.com/pkg/errors

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

// WithMessage annotates err with a new message.
// If err is nil, WithMessage returns nil.
func WithMessage(err error, message string) error {
    if err == nil {
        return nil
    }
    return &withMessage{
        cause: err,
        msg:   message,
    }
}

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 }

// Unwrap provides compatibility for Go 1.13 error chains.
func (w *withMessage) Unwrap() error { return w.cause }
```

T.e.

```
func (e *errorString) Error() string { /* ... */ }
func (e *wrapError) Error() string { /* ... */ }

func (w *withMessage) Error() string { /* ... */ }
```

Собственно мы неоднократно сталкивались с этим в курсе, и ответ на вопрос "а зачем так сделано?" кроется в том же комментарии к `errors.New`:

```
// Each call to New returns a distinct error value even if the text
is identical.
```

Таким образом, две ошибки, созданные через этот конструктор, будут всегда различны, даже если они имеют общий текст. Реализовано это через возврат указателя на структуру

```
func New(text string) error {
    return &errorString{text}

}
```

А значит хотелось бы семантически сделать так, чтобы **только указатель на тип** мог реализовать интерфейс `error`, что и происходит:

```
type errorString struct {
    s string
}

func (e *errorString) Error() string {
```

```
    return e.s

}
```

Своеобразное противоречие одних лучших практик другим.

Наш `NewStatusError` из предыдущего шага не обладает особенностью построения "уникальной" ошибки:

```
// https://goplay.tools/snippet/LiDSj7WN2p_4

var (
    ErrNotFound  = NewStatusError(http.StatusNotFound, "Not Found")
    ErrNotFound2 = NewStatusError(http.StatusNotFound, "Not Found")
)

type StatusError struct {
    code int
    msg  string
}

func (s StatusError) Error() string {
    return fmt.Sprintf("%d %s", s.code, s.msg)
}

func NewStatusError(code int, msg string) StatusError {
    return StatusError{code: code, msg: msg}
}

func main() {
    fmt.Println(ErrNotFound == ErrNotFound2) // true
    fmt.Println(errors.Is(ErrNotFound, ErrNotFound2)) // true

}
```

Как мы понимаем, это легко исправить, начав возвращать указатель из конструктора:

```
// https://goplay.tools/snippet/TNcDFLqojA8

type StatusError struct {
    code int
    msg  string
```

```
}

func (s StatusError) Error() string {
    return fmt.Sprintf("%d %s", s.code, s.msg)
}

func NewStatusError(code int, msg string) *StatusError {
    return &StatusError{code: code, msg: msg}
}

func main() {
    fmt.Println(ErrNotFound == ErrNotFound2)          // false
    fmt.Println(errors.Is(ErrNotFound, ErrNotFound2)) // false
}
```

Но так в рамках одного пакета всё ещё остаётся возможность выстрелить себе в ногу, потому что есть возможность воспользоваться ошибкой в обход конструктора:

```
var se1 error = StatusError{code: http.StatusNotFound}
var se2 error = StatusError{code: http.StatusNotFound}

fmt.Println(se1 == se2) // true
```

Чтобы подобный код даже не компилировался, необходимо запретить типу `StatusError` реализовывать `error`, для этого достаточно сделать **pointer receiver** у соответствующего метода (исходник примера):

```
type StatusError struct {
    code int
    msg  string
}

func (s *StatusError) Error() string {
    return fmt.Sprintf("%d %s", s.code, s.msg)
}

func NewStatusError(code int, msg string) *StatusError {
    return &StatusError{code: code, msg: msg}

}
```

(более того теперь при вызове метода `Error` копируется 8 байт ресивера-указателя, а не 24 как раньше).

Почему уникальность создаваемой ошибки важна?

Фундаментальные конструкторы ошибок вроде `errors.New` обязаны гарантировать уникальность возвращаемой ошибки по той простой причине, что эти ошибки могут иметь один текст, но при этом быть созданы в совершенно разных пакетах. И нам не хотелось бы спутать нашу ошибку с "чужой", что чревато багами.

Например, в некоторых стандартных и не очень пакетах вроде `os`, `io`, `github.com/jackc/pgx` и т.д. есть sentinel ошибки, торчащие наружу, которыми мы пользуемся для диагностирования возможных проблем.

Благодаря тому, что в пакете `errors` тип `*errorString` имплементирует интерфейс `error` через указатель, исключается возможность выстрела в ногу из-за создания собственной ошибки, которая будет равна какой-либо sentinel ошибке из внешнего пакета.

Примером тому служит код ниже, где мы пытаемся создать идентичные по внутренностям ошибки, но тем не менее `errors.Is()` всегда возвращает, как и ожидается, `false` (исходник примера):

```
// https://goplay.tools/snippet/3FO6u7Mxdws

package main

import (
    "errors"
    "fmt"
    "io"
    "os"

    "github.com/jackc/pgx"
)

func main() {
    exist := errors.New("file already exists")
    fmt.Println(errors.Is(exist, os.ErrExist)) // false
```

```
    eof := errors.New("EOF")
    fmt.Println(errors.Is(eof, io.EOF)) // false

    noRows := errors.New("no rows in result set")
    fmt.Println(errors.Is(noRows, pgx.ErrNoRows)) // false

}
```

Исключения из правил

Если снова обратиться к стандартной библиотеке, то можно заметить, что встречаются ошибки, реализованные не через указатель на тип:

```
// https://github.com/golang/go master
$ grep -rE " [a-zA-Z]*\) Error\(\) string" . | wc -l
      96

$ grep -rE " [a-zA-Z]*\) Error\(\) string" . | wc -l
./src/cmd/compile/internal/syntax/syntax.go:func (err Error) Error()
string {
/src/crypto/des/cipher.go:func (k KeySizeError) Error() string {
./src/crypto/x509/x509.go:func (e InsecureAlgorithmError) Error()
string {
./src/crypto/x509/x509.go:func (h UnhandledCriticalExtension) Error()
string {
./src/compress/bzip2/bzip2.go:func (s StructuralError) Error() string
{
/src/go/scanner/errors.go:func (e Error) Error() string {
./src/go/scanner/errors.go:func (p ErrorList) Error() string {
./src/archive/tar/common.go:func (he headerError) Error() string {
c/image/png/reader.go:func (e UnsupportedError) Error() string {

// ...
```

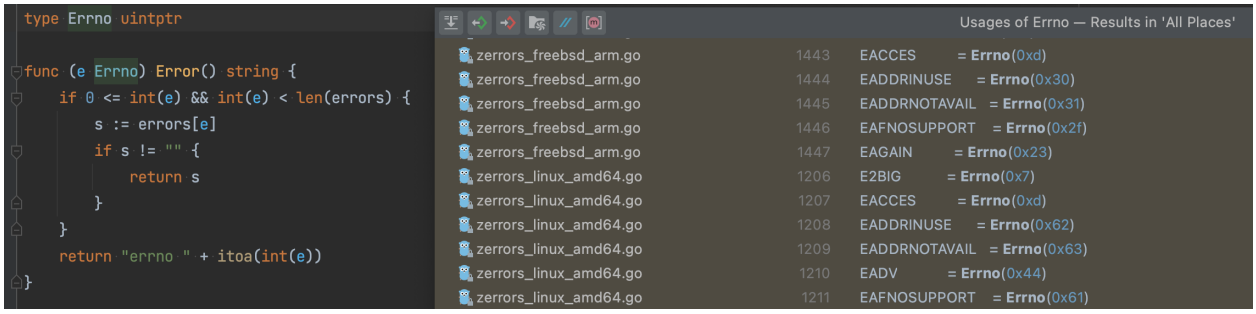
Такие ошибки обычно обладают двумя отличительными чертами.

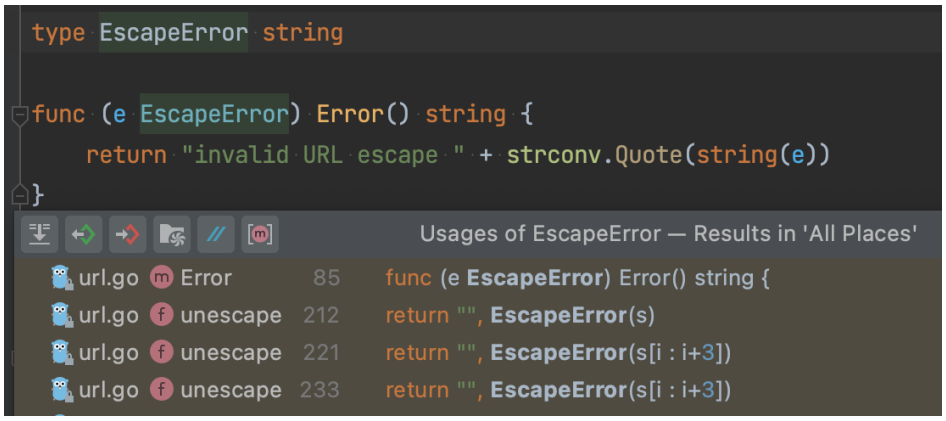
1) Они являются типами-обёртками над примитивными типами (`string`, `int`, `uintptr` и т.д.):

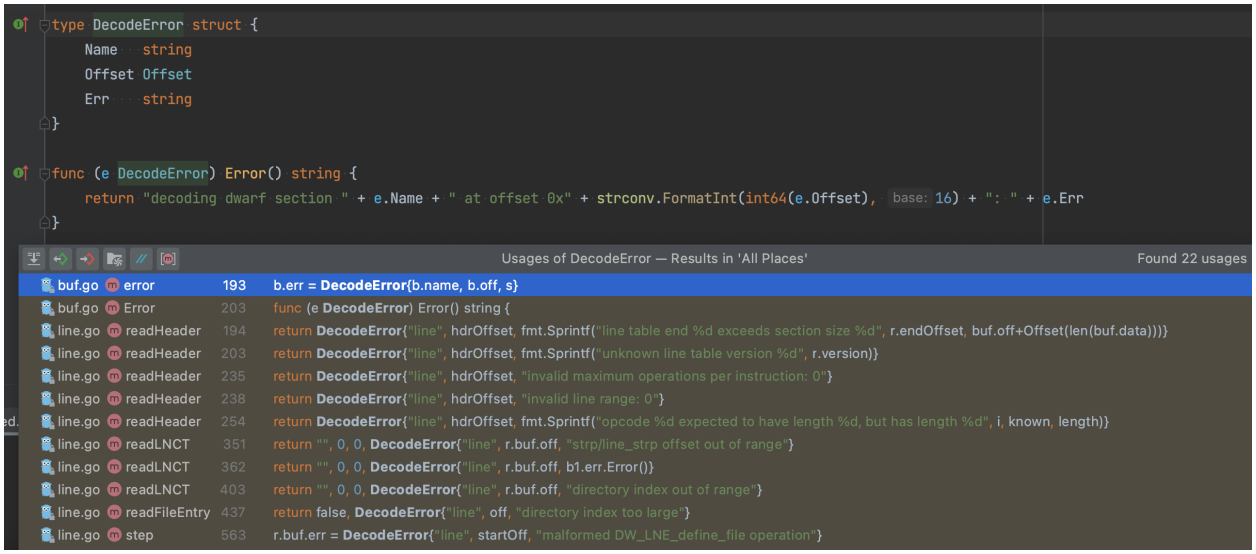
```
// src/crypto/x509/x509.go
type SignatureAlgorithm int
```

```
type InsecureAlgorithmError SignatureAlgorithm

// src/os/user/user.go
type UnknownUserIdError int
type UnknownUserError string

// src/image/png/reader.go
type FormatError string

// src/net/url/url.go
type InvalidHostError string


// и т.д.
```

2) Они часто создаются "на лету", являясь временными переменными, от которых в принципе невозможно взять адрес (не нужно путать со специальным механизмом получения адреса для структурных литералов):







Какие видятся недостатки?

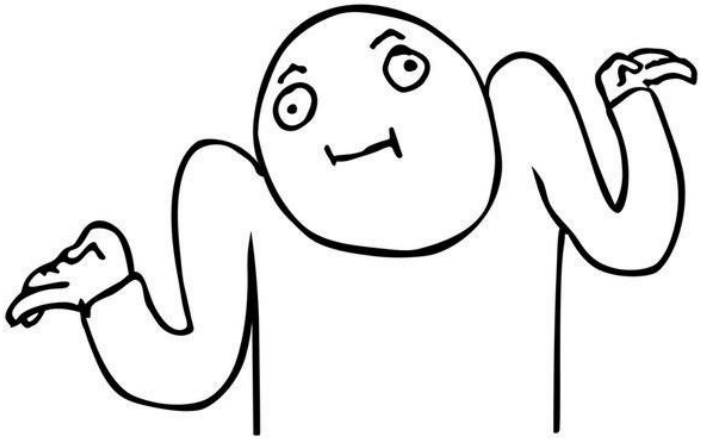
Очевидно, что две ошибки, созданные через подобный тип будут равны при равенстве значений:

```
// https://goplay.tools/snippet/_BOFDkn5ZB-

func main() {
    e1 := url.InvalidHostError("www.golang-courses.ru")
    e2 := url.InvalidHostError("www.golang-courses.ru")
    fmt.Println(errors.Is(e1, e2)) // true

    e3 := syscall.Errno(0xd)
    e4 := syscall.Errno(0xd)
    fmt.Println(errors.Is(e3, e4)) // true
}
```

Видимо, разработчики языка осознанно идут на это, подразумевая (или надеясь?), что никто не будет использовать подобные типы для sentinel ошибок.

А если и будут, то, видимо, разработчики Go не видят ничего страшного в равенстве подобных ошибок –

`syscall.Errno(0xd)` он и в Африке `syscall.Errno(0xd)`.



Кроме того, используя подобный тип, не выйдет создать "на лету" ошибку-указатель, потому что мы не сможем взять адрес

(в данном случае строкового) литерала:

```
var ErrInvalidHost = &url.InvalidHostError("www.golang-courses.ru")
```

Cannot take the address of 'url.InvalidHostError("www.golang-courses.ru")'

Но при этом всё равно есть возможность вернуть ошибку через указатель:

```
err := url.InvalidHostError("www.golang-courses.ru")

return &err // Реализует error.
```

И в коде выше нам придётся гадать, по какому типу делать проверку – `url.InvalidHostError` или `*url.InvalidHostError`, ведь они оба реализуют `error` из-за **value receiver** у метода `Error`:

```
// src/net/url/url.go
package url

type InvalidHostError string

func (e InvalidHostError) Error() string {
    return "invalid character " + strconv.Quote(string(e)) + " in
host name"

}
```

Подробнее об этой проблеме в следующем шаге.

Value receiver и errors.Is / errors.As

В стандартной библиотеке встречаются ошибочные типы, не использующие ресивер-указатель, например, `template.ExecError`:

```
// src/text/template/exec.go

package template

// ExecError is the custom error type returned when Execute has an
// error evaluating its template.
type ExecError struct {
    Name string // Name of template.
    Err  error  // Pre-formatted error.
}

func (e ExecError) Error() string {
    return e.Err.Error()
}

func (e ExecError) Unwrap() error {
    return e.Err
}
```

Какие здесь сложности? Такие, что интерфейсу `error` удовлетворяет и значение типа `ExecError` и значение типа `*ExecError`.

Соответственно при работе с этой ошибкой вы должны или обрабатывать оба варианта или прошерстить исходники предоставляющего ошибку пакета, чтобы быть уверенным, что она конструируется через единственный тип:

```
var t template.ExecError
var tPtr *template.ExecError

// Что выбрать?
switch {
case errors.As(err, &t):
    // ...
case errors.As(err, &tPtr):
    // ...
```

```
}
```

Аналогично с определением у своего типа `Is` / `As`. Например, мы хотим завести `MyExecError` и задекларировать правила, по которым его можно считать идентичным `ExecError` (исходник примера):

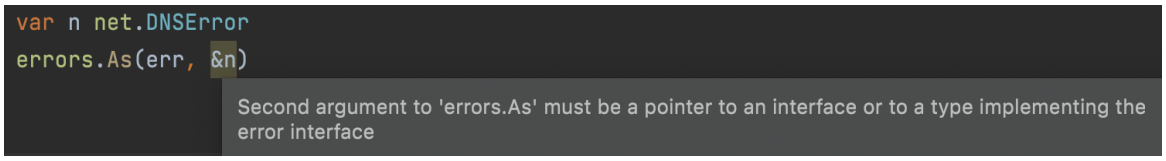
```
type MyExecError struct {
}

func (m *MyExecError) Error() string {
    return "cool error"
}

func (m *MyExecError) Is(target error) bool {
    // Что выбрать?
    switch target.(type) {
    case *template.ExecError:
        // ...
    case template.ExecError:
        // ...
    }
    return false
}

func (m *MyExecError) As(target interface{}) bool {
    // Что выбрать?
    switch target.(type) {
    case *template.ExecError:
        // ...
    case **template.ExecError:
        // ...
    }
    return false
}
```

В случае же ошибки, реализованной через ресивер-указатель (например, `net.DNSError`), нам думать не нужно – допустимым будет являться только один кейс:

```
func (m *MyError) Is(target error) bool {
```

```
    if v, ok := target.(*net.DNSError); ok {
        // ...
    }
    return false
}

func (m *MyError) As(target interface{}) bool {
    if v, ok := target.(**net.DNSError); ok {
        // ...
    }
    return false
}
```

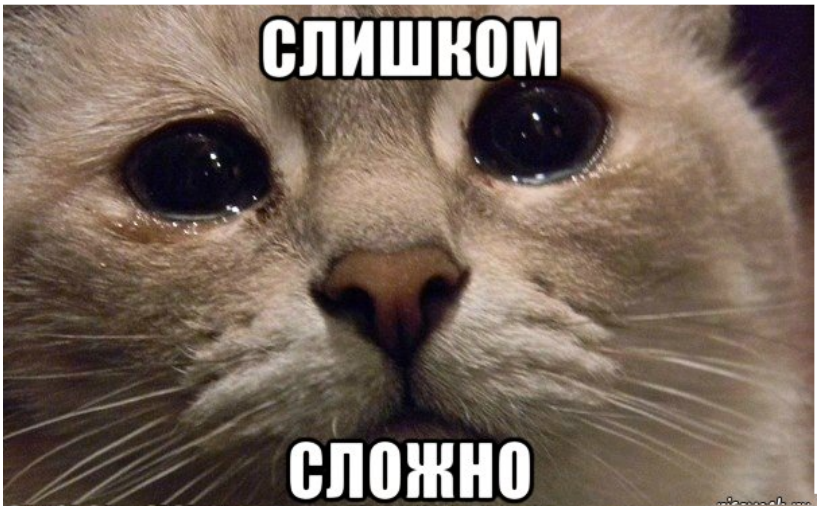
Так как обычное значение подобного типа (не указатель) ошибкой не является:



Выводы

Таким образом, при выборе типа ресивера для метода `Error` (когда мы создаём свой тип `MyError`) следует руководствоваться следующими моментами:

- хотим ли мы унификацию типа или нам подходит, что интерфейсу `error` будет удовлетворять как `MyError`, так и `*MyError`?
- является ли наш тип обёрткой над примитивном типом (`string`, `int` и т.д.)?
- как нашу ошибку будут создавать и использовать (через указатель или нет)?
- важно ли нам, чтобы две ошибки с идентичными полями были различны (или наоборот, чтобы были равны)?



Но на самом деле ресивер-указатель закрывает практически все вопросы выше:

```
func (m *MyError) Error() string
```

- интерфейсу `error` будет удовлетворять только `*MyError`;
- как следствие создавать ошибки можно будет только через указатель – `&MyError{}`;
- две созданные подобным образом ошибки никогда не будут равны.

Но мы теряем возможность делать базовым типом примитив (boolean, numeric, string type).

Исходя из всего вышеперечисленного напрашивается вывод – старайтесь реализовывать интерфейс `error` через ресивер-указатель!

Тест "Выберите наиболее подходящий для ошибки тип ресивера"

Отметьте верные ячейки

	Используем Value Receiver, конструируем через значение	Используем Value Receiver, конструируем через указатель	Используем Pointer Receiver, конструируем через указатель
Нам нужно, чтобы ошибки с одинаковыми полями были равны.			
Нам нужно, чтобы две на первый взгляд идентичные ошибки никогда не были равны			
Наш тип – обёртка над примитивом			
Наш тип – обёртка над слайсом			
Нам нужно, чтобы только указатель на ошибку удовлетворял интерфейсу error			
Мы сомневаемся и не знаем, что выбрать			

Задача "HTTPError"

[Ссылка на заготовку.]

Вам необходимо реализовать ряд ошибок:

```
var (
    ErrStatusOK                  error
    ErrStatusBadRequest          error
    ErrStatusNotFound            error
    ErrStatusUnprocessableEntity error
    ErrStatusInternalServerError error

)
```

Использовав

```
type HTTPError int
```

Ошибки должны удовлетворять интерфейсу

```
type c interface {
    Code() int

}
```

а их текст должен имеет следующий вид:

```
"200 OK"
"400 Bad Request"
"404 Not Found"
"422 Unprocessable Entity"

"500 Internal Server Error"
```

Вам доступны пакеты `net/http`, `fmt` и `strconv`.