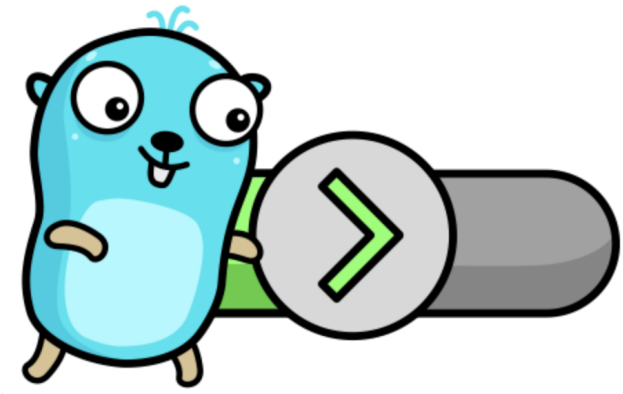
github.com/pkg/errors (часть 1)

Есть нестандартная библиотека github.com/pkg/errors, которую многие считают стандартом **de facto** для работы с ошибками.

Она обладает некоторыми фишками, которых нет в стандартной библиотеке Go, о них и поговорим далее.



А где случилась ошибка?

Вот мы поработали со стандартной библиотекой `errors`. Удобная штука – без необходимости не нужно создавать свою структуру ошибки, можно легко добавлять человекочитаемое описание к уже имеющейся ошибке, можно разворачивать вложенные друг в друга ошибки и всё такое.

В общем случае при выводе на экран или в логи ошибки, обработанной средствами стандартной библиотеки, получим что-то вроде:

```
wow, this is an error: wow, error happened there: error happened: EOF
```

Чего не хватает? Кажется, нам по-прежнему может не хватать контекста, где случилась ошибка. Можно, конечно, искать по коду проекта текст ошибок, но хочется иметь стектрейс, чтобы делать это побыстрее.

Так вот, стектрейс ошибки – главная фича **github.com/pkg/errors**.

Задача "Стектрейс своими руками"

Me: *Mom, can we use "github.com/pkg/errors" for stacktrace?*

Mom*: No, we have stacktrace at home*

Stacktrace at home:

Ссылка на заготовку.

Вам необходимо реализовать функцию `Trace`, возвращающую стектрейс от того места, где она была вызвана:

```
type Frame uintptr
type StackTrace []Frame


func Trace() StackTrace
```

Если напечатать трейс, то каждый фрейм будет представлен в виде двух строк: имени функции и файла с указанием номера строки:

```
t := Trace()
fmt.Println(t)

/*
handmade-stacktrace.ExampleTrace
handmade-stacktrace/stack_trace_test.go:14
testing.runExample
```

```
testing/run_example.go:64
testing.runExamples
testing/example.go:44
testing.(*M).Run
testing/testing.go:1505
main.main
_testmain.go:45
runtime.main
runtime/proc.go:225
runtime.goexit
runtime/asm_arm64.s:1133
*/

// Конкретные позиции в файлах стандартной библиотеки Go могут
отличаться

// в разных версиях компилятора. Не переживайте, наши тесты на них не
захардкожены!
```

В примере выше первым фреймом (первым в слайсе, но последним в порядке вызовов) является

```
handmade-stacktrace.ExampleTrace

handmade-stacktrace/stack_trace_test.go:14
```

(один фрейм форматируется как две строчки)

Обратите внимание на формат, к которому приводятся имя функции и путь до неё.

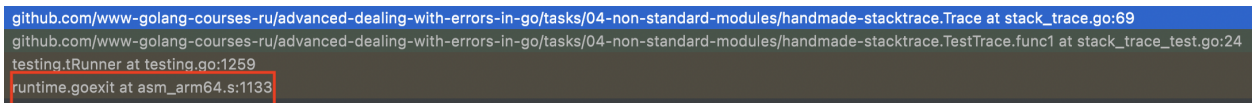
Также к функции предъявляется пара специфических требований:

- Она должна возвращать стектрейс глубиной не более 32. То есть, если представить, что глубина стека вызовов нашей программы больше 32, то нам достаточно видеть его верхушку высотой 32.
- Возвращаемый стектрейс не должен включать фрейм самой функции `Trace`.

Для реализации вам пригодятся runtime.Callers и runtime.FuncForPC.

Замечания и подсказки:

- Stepik запускает ваше решение в своей песочнице, хардкод на работу с локальными путями заготовки не пройдёт.
- Не потеряйте корневой фрейм от пакета `runtime`:



- И не забудьте обрезать фрейм от самой функции `Trace` (первый сверху фрейм на скриншоте выше).

Форматированный стектрейс

В предыдущей задаче мы практически реализовали стектрейс данного модуля:

```
// https://goplay.tools/snippet/T3lLbgqJGCP

package main

import (
    "fmt"

    "github.com/pkg/errors"
)

func main() {
    err := errors.New("error happened")
    fmt.Printf("%+v", err)
}

/*
error happened
main.main
```

```
    /tmp/sandbox2146050486/prog.go:10
runtime.main
    /usr/local/go-faketime/src/runtime/proc.go:255
runtime.goexit
    /usr/local/go-faketime/src/runtime/asm_amd64.s:1581
*/
```

Сравним с выводом ошибки, созданной стандартным способом:

```
import (
    "errors"
    "fmt"
)

func main() {
    err := errors.New("error happened")
    fmt.Printf("%+v", err)
}


// error happened
```

Опытный читатель заметил, что мы выводили ошибки через `%+v` – в обычном случае это добавляет имена полей при выводе структур, но в нашем случае это позволяет выводить ошибку от **github.com/pkg/errors** вместе со стектрейсом.

Реализовано это через интерфейс fmt.Formatter:

```
package fmt

// Formatter is implemented by any value that has a Format method.
// The implementation controls how State and rune are interpreted,
// and may call Sprint(f) or Fprint(f) etc. to generate its output.
type Formatter interface {
    Format(f State, verb rune)

}
```

Более продвинутый вариант, чем реализация fmt.Stringer, которую мы сделали в предыдущей задаче.

Соответственно в модуле это выглядит [следующим образом]():

```
package errors

// stack represents a stack of program counters.
type stack []uintptr

func (s *stack) Format(st fmt.State, verb rune) {
    switch verb {
    case 'v':
        switch {
        case st.Flag('+'):
            for _, pc := range *s {
                f := Frame(pc)
                fmt.Fprintf(st, "\n%+v", f)
            }
        }
    }
}

func (s *stack) StackTrace() StackTrace {
    f := make([]Frame, len(*s))
    for i := 0; i < len(f); i++ {
        f[i] = Frame((*s)[i])
    }
    return f

}
```

Где `StackTrace` и `Frame` – уже знакомые нам типы, но с чуть более сложной [реализацией]().

Мы посмотрели на то как выглядит стектрейс, давайте теперь узнаем, как собственно его создавать при работе с ошибками через **github.com/pkg/errors**.

Создание и врапинг ошибок

Кратенько ознакомимся с библиотечными функциями.

errors.New

errors.New так же, как и собрат из стандартной библиотеки, создает экземпляр ошибки:

```
package errors

// New returns an error with the supplied message.
// New also records the stack trace at the point it was called.
func New(message string) error {
    return &fundamental{
        msg:   message,
        stack: callers(),
    }

}
```

Отличается тем, что структура `fundamental` помимо сообщения несёт в себе стектрейс `stack`.

Вызываете функцию и **она запишет в стектрейс место вызова**. Что мы и видели на предыдущем шаге.

---

errors.Errorf

errors.Errorf идентична своей сестрёнке из пакета `fmt`, только помимо форматированного сообщения добавляет стектрейс к ошибке:

```
package errors

// Errorf formats according to a format specifier and returns the
string
// as a value that satisfies error.
// Errorf also records the stack trace at the point it was called.
func Errorf(format string, args ...interface{}) error {
    return &fundamental{
        msg:   fmt.Sprintf(format, args...),
        stack: callers(),
    }

}
```

---

errors.WithStack[f]

errors.WithStack оборачивает переданную ошибку в `*withStack`, цепляя к ней стектрейс:

```
package errors

// WithStack annotates err with a stack trace at the point WithStack
was called.
// If err is nil, WithStack returns nil.
func WithStack(err error) error {
    if err == nil {
        return nil
    }
    return &withStack{
        err,
        callers(),
    }

}
```

А что делать, если стектрейс нам не нужен? Для этого пригодится следующая функция.

---

errors.WithMessage[f]

errors.WithMessage оборачивает переданную ошибку в `*withMessage` и цепляет к ней наше сообщение:

```
package errors

func WithMessage(err error, message string) error {
    if err == nil {
        return nil
    }
    return &withMessage{
        cause: err,
        msg:   message,
    }

}
```

По сути это аналог `fmt.Errorf` + `%w`.

---

errors.Wrap

errors.Wrap – это комбинация двух методов выше, одновременно добавляет к ошибке и сообщение и стектрейс:

```
package errors

// Wrap returns an error annotating err with a stack trace
// at the point Wrap is called, and the supplied message.
// If err is nil, Wrap returns nil.
func Wrap(err error, message string) error {
    if err == nil {
        return nil
    }
    err = &withMessage{
        cause: err,
        msg:   message,
    }
    return &withStack{
        err,
        callers(),
    }

}
```

На первый взгляд она кажется избыточной, но на удивление – это самая популярная функция модуля.

Типы `*withMessage` и `*withStack` реализуют целую пачку интерфейсов среди которых `error`, `fmt.Formatter`, а также

```
type cause interface { // Из github/pkg/errors.
    Cause() error
}

interface { // Безымянный из errors.
    Unwrap() error
}
```

Например,

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

func (w *withMessage) Format(s fmt.State, verb rune) { /* ... */ }
```

Как вы уже догадались, это помогает строить цепочки ошибок, а также красиво выводить их на экран.

Тест "Какие из функций тянут за собой стектрейс?"

Самая частая ошибка при работе с модулем **github/pkg/errors** – забывать данный факт.

Чем это чревато, мы увидим в следующих уроках.

Выберите все подходящие ответы из списка

- [ ] fmt.Errorf
- [ ] errors.New (std)
- [ ] errors.New (pkg/errors)
- [ ] errors.Errorf (pkg/errors)
- [ ] errors.WithMessage (pkg/errors)
- [ ] errors.WithMessagef (pkg/errors)
- [ ] errors.WithStack (pkg/errors)
- [ ] errors.Wrap (pkg/errors)
- [ ] errors.Wrapf (pkg/errors)

Распаковка ошибок

Мы поняли, как создавать и врапить ошибки. А как их разворачивать?

errors.Cause

errors.Cause – это метод-прародитель `errors.Unwrap` из стандартной библиотеки:

```
// Cause returns the underlying cause of the error, if possible.
// An error value has a cause if it implements the following
// interface:
//
//     type causer interface {
//            Cause() error
//     }
//
// If the error does not implement Cause, the original error will
// be returned. If the error is nil, nil will be returned without
further
// investigation.
func Cause(err error) error {
    type causer interface {
        Cause() error
    }

    for err != nil {
        cause, ok := err.(causer)
        if !ok {
            break
        }
        err = cause.Cause()
    }
    return err

}
```

Назначение метода ровно то же самое – получить оригинальную ошибку. Только здесь оригинальной считается та ошибка, которая последняя реализует интерфейс `causer` (Unwrap же разворачивает единожды, не в цикле).

Ничего не напоминает?

---

errors.Unwrap

Эта функция ничем не примечательна, так как полностью идентична своему аналогу из стандартной библиотеки, потому что именно он внутри и вызывается:

```
// +build go1.13

package errors

import (
    stderrors "errors"
)

// 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 {
    return stderrors.Unwrap(err)

}
```

Была добавлена в **github.com/pkg/errors** после релиза Go 1.13.

---

errors.Is, errors.As

Эти функции, как и `Unwrap`, были включены в стандартную библиотеку, и после этого на неё же саму и были переписаны:

```
// +build go1.13

package errors

import (
    stderrors "errors"
)

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

```
// As finds the first error in err's chain that matches target, and
if so, sets
// target to that error value and returns true.
func As(err error, target interface{}) bool {
    return stderrors.As(err, target)

}
```

Типичный флоу работы

Возьмём пример из предыдущего модуля и перепишем его с использованием **github.com/pkg/errors**.

Было (исходник примера):

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

func loadFiles(urls ...string) ([]File, error) {
    // ...
    if err != nil {
        return nil, &FileLoadError{url, err}
    }
    // ...
}

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

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

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

```
    }

}
```

Стало (исходник примера):

```
// https://goplay.tools/snippet/5E5_Z-B3C7z

func loadFiles(urls ...string) ([]File, error) {
    // ...
    if err != nil {
        // Прицепили стек в самом глубоком месте.
        return nil, errors.WithStack(&FileLoadError{url, err})
    }
    // ...
}

func transfer() error {
    _, err := loadFiles("www.golang-courses.ru")
    if err != nil {
        return errors.WithMessage(err, "cannot load files")
    }
    // ...
}

func handle() error {
    if err := transfer(); err != nil {
        return errors.WithMessage(err, "cannot transfer files")
    }
    // ...
    return nil
}

func main() {
    err := handle()
    fmt.Printf("%+v\n\n", err)

    if f, ok := errors.Cause(err).(*FileLoadError); ok { //
Используем Cause!
        fmt.Println(f.URL)
    }
}

/*
"www.golang-courses.ru": context canceled
```

```
main.loadFiles
    /tmp/sandbox639267487/prog.go:31
main.transfer
    /tmp/sandbox639267487/prog.go:39
main.handle
    /tmp/sandbox639267487/prog.go:49
main.main
    /tmp/sandbox639267487/prog.go:58
runtime.main
    /usr/local/go-faketime/src/runtime/proc.go:255
runtime.goexit
    /usr/local/go-faketime/src/runtime/asm_amd64.s:1581
cannot load files
cannot transfer files

www.golang-courses.ru

*/
```

Changelog:

1) По местам возврата ошибок мы применили `errors.WithStack` и `errors.WithMessage`. Использование того или иного врапинга зависит лишь от того, что **вы хотите получить в итоге**.

2) На самом верхнем уровне мы заиспользовали `errors.Cause`:

```
if f, ok := errors.Cause(err).(*FileLoadError); ok {
    // ...

}
```

Выглядит это как шаг вперёд и два назад – только говорили о том, что больше не нужно пользоваться **type assertions** при работе с ошибками и на тебе.

На самом деле так получается из-за того, что `Cause` появился гораздо раньше `Is` / `As`, и сейчас в целом вам ничего не мешает пользоваться последними:

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

}
```

Иногда этого избегают, чтобы не возникало путаницы между стандартным **errors** и **github.com/pkg/errors** (хотя как мы помним из предыдущего шага модульные функции под собой имеют вызов стандартных), а также из-за проблем с совместимостью между этими пакетами.

О чём мы поговорим в следующем уроке.

Промежуточные выводы

Мы познакомились с **github.com/pkg/errors** – прародителем функций для работы с ошибками из стандартной библиотеки.

В следующем уроке мы посмотрим, чем они похожи, а чем отличаются, и на какие грабли можно наступить, используя оба пакета разом.