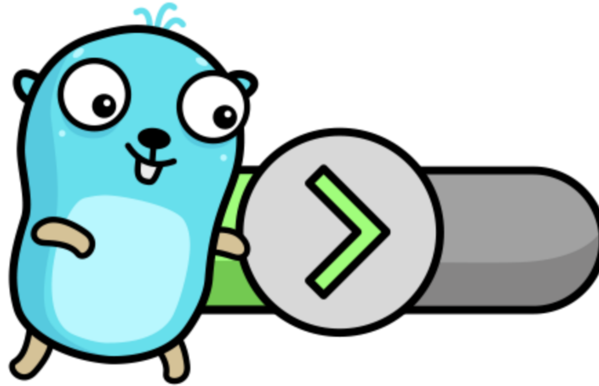
Error Inspection & Printing

В этом уроке мы поговорим о проблемах проверок ошибок и вывода детальной информации о них, а также о том, как эти проблемы удалось (или не удалось) решить.



Постановка проблемы

В общем случае существовало 4 способа проверки результата выполнения функции на конкретные ошибки:

1. Прямое сравнение с **sentinel errors** (`io.EOF` и пр.).

2. Проверка непосредственно ошибочного типа, лежащего в интерфейсе `error`, с помощью type assertion или type switch.

3. Специальные **opaque** функции вроде `os.IsNotExist`, выполняющие проверки на конкретные ошибки с помощью ограниченного (на один уровень) развёртывания.

4. Стоило нам заврапить ошибку, и все подходы выше работать переставали. Оставалось только искать подстроку в `err.Error` ~~куда же без этого~~.

Врапинг был представлен двумя вариантами:

```
if err != nil {
    return fmt.Errorf("write users database: %v", err) // Врапим в текст.
}

// ...

if err != nil {
    return &WriteError{Database: "users", Err: err} // Врапим в свой тип.

}
```

В сложных программах при длинных **цепочках ошибок** у нас не было удобного способа проверить, входит ли конкретная ошибка в данную цепочку.

Например,

```
write users database: call myserver.Method: dial myserver:3333: open /etc/resolv.conf: permission denied
```

включает в себя:

- `*WriteError` (`"write users database: "`);
- `*RPCError` (`"call myserver.Method: "`);
- `*net.OpError` (`"dial myserver:3333: "`);
- `*os.PathError` (`"open /etc/resolv.conf: "`);
- `syscall.EPERM` (`"permission denied"`).

Если нам повезёт и каждая из ошибок не будет завраплена в текст, а все ошибочные типы будут хранить нижележащие ошибки, то мы сможем добраться до каждой из них, иначе – увы и ах.

Вторая менее критичная, но всё же важная проблема, состоит в том, что отчёты о глубоко вложенных ошибках оставляют желать лучшего – их трудно читать и они

не содержать дополнительных деталей об ошибке, таких как, например, позиция места её возникновения в файле.

Что хотелось получить?

Хотелось упростить проверку ошибок (**error inspection**) и сделать её менее подверженной ошибкам (простите за тавтологию), не ломая существующих механизмов (т.е. все старые проверки должны работать и не требовать изменений).

По выводу ошибок на экран хотелось бы не столько уметь в стек ошибки, а сколько иметь возможность для её локализации с помощью golang.org/x/text/message.

Важное замечание

Более того создание ошибок должно было остаться столь же эффективным. **Ошибки – не исключения** (мы ещё коснёмся этого в следующем уроке), обычно они генерируются, обрабатываются и отбрасываются снова и снова во время работы программы.

Russ Cox пишет, что стоимость создания ошибки в Go должна быть фиксированной и **не зависеть от глубины стека** или другого контекста. И приводит пример программы, которая большую часть времени своей работы тратила на генерацию исключений:

> *As a cautionary tale, years ago at Google a program written in an exception-based language was found to be spending all its time generating exceptions. It turned out that a function on a deeply-nested stack was attempting to open each of a fixed list of file paths, to find a configuration file. Each failed open operation threw an exception; the generation of that exception spent a lot of time recording the very deep execution stack; and then the caller discarded all that work and continued around its loop. The generation of an error in Go code must remain a fixed cost, regardless of stack depth or other context.*

[ACCEPTED] Проверка ошибок: решение

Было решено ввести следующие инструменты, которые помогут легко извлекать информацию из ошибки, и при этом не зависят от конкретного пакета или количества её оборачиваний. Никаких изменений по механизму создания ошибок не произошло.

Интерфейс errors.Wrapper

```
package errors

// A Wrapper is an error implementation wrapping context around
another error.
type Wrapper interface {
    // Unwrap returns the next error in the error chain.
    // If there is no next error, Unwrap returns nil.
    Unwrap() error
}
```

Функции errors.Is и errors.As

```
package errors

func Is(err, target error) bool

func As(type E)(err error) (e E, ok bool)
```

```
// Вместо `err == io.ErrUnexpectedEOF`.
if errors.Is(err, io.ErrUnexpectedEOF) { /* ... */ }

// Вместо `pe, ok := err.(*os.PathError)`.

if pe, ok := errors.As(*os.PathError)(err); ok { /* ... pe.Path ...
*/ }
```

Мы не расписываем принцип работы функций выше, так как они вам хорошо известны.

На что хочется обратить внимание:

1) Здорово, что разработчики языка, не дожидаясь новой мажорной версии, вводят в него полезные фичи гораздо раньше, чем это могло бы быть.

2) Обратите внимание на **полиморфную** сигнатуру `errors.As`:

```
func As(type E)(err error) (e E, ok bool)
```

дизайн разрабатывался с учётом [дженериков](), ожидаемых в Go 2. Сейчас же мы имеем временный хелпер взамен этого дела:

```
// Вместо `pe, ok := err.(*os.PathError)`.
var pe *os.PathError

if errors.As(&pe, err) { /* ... pe.Path ... */ }
```

3) Интерфейса `errors.Wrapper` не появилось. По неведомым нам причинам разработчики Go используют анонимный интерфейс. Зато появилась, не заявленная в дизайне, функция `errors.Unwrap`:

```
package errors

func Unwrap(err error) error {
    u, ok := err.(interface {
        Unwrap() error
    })
    // ...
}
```

А также, как мы помним, мы получили возможность переопределять поведение `errors.Is` и `errors.As` для своих типов.

4) [Здесь]() можно почитать, почему разработчики не пошли по пути **github.com/pkg/errors.Cause** и пр. сторонних модулей. Вкратце:

- работа только с корневой ошибкой (на что рассчитан `errors.Cause`) – не круто;
- наличие и функции `Cause` и метода `Cause`, делающих разные вещи, сбивает людей с толку;
- в разных существующих пакетах `Cause` имеет разное значение, а новый метод `Unwrap` позволит внести единообразие;
- запись стека в целом не относится к **error inspection** и поэтому не обсуждалась.

[ABANDONED] Вывод ошибок на экран: решение

Printing API

Для унификации форматирования ошибок и их работы с деталями, было бы решено ввести в **errors** новые интерфейсы:

```
package errors

// A Formatter formats error messages.
type Formatter interface {
    // Format is implemented by errors to print a single error message.
    // It should return the next error in the error chain, if any.
```

```
    Format(p Printer) (next error)
}

// A Printer creates formatted error messages. It enforces that
// detailed information is written last.
//
// Printer is implemented by fmt. Localization packages may provide
// their own implementation to support localized error messages
// (see for instance golang.org/x/text/message).
type Printer interface {
    // Print appends args to the message output.
    // String arguments are not localized, even within a localized
context.
    Print(args ...interface{})

    // Printf writes a formatted string.
    Printf(format string, args ...interface{})

    // Detail reports whether error detail is requested.
    // After the first call to Detail, all text written to the
Printer
    // is formatted as additional detail, or ignored when
    // detail has not been requested.
    // If Detail returns false, the caller can avoid printing the
detail at all.
    Detail() bool

}
```

Или по сути:

```
package errors

// Optional method for error implementations.
type Formatter interface {
    Format(p Printer) (next error)
}

// Interface passed to Format.
type Printer interface {
    Print(args ...interface{})
    Printf(format string, args ...interface{})
    Detail() bool

}
```

`Formatter` пишет не в `io.Writer`, чтобы можно было поддержать локализацию. Так, реализация `Printer` обычно предоставляется пакетом `fmt`, но может предоставляться и модулями для локализации, например, (*golang.org/x/text/message).Printer:

```
import "golang.org/x/text/message"

p := message.NewPrinter(language.Dutch)

p.Printf("Error: %v", err)
```

Пример реализации интерфейса выше для нашей ошибки:

```
type myAddrError struct {
    address string
    detail  string
    err     error
}

func (e *myAddrError) Error() string {
    return fmt.Sprint(e) // Делегируем методу Format.
}

func (e *myAddrError) Format(p errors.Printer) error {
    p.Printf("address %s", e.address)
    if p.Detail() {
        p.Print(e.detail)
    }
    return e.err

}
```

Формат детализированного вывода для цепочки ошибок

На данный момент `fmt.Printf` выводит на экран ошибки следующим образом (исходник примера):

- `%s`: как строка `err.Error()`;
- `%q`: как строка `err.Error()` в кавычках;
- `%+q`: как ASCII-строка `err.Error()` в кавычках;

- `%v`: как строка `err.Error()`;
- `%#v`: как переменная `err` в синтаксисе Go.

```
// https://goplay.tools/snippet/l5fde1XFjIC

var КонецФайла = errors.New("конец файла")

func main() {
    var verbs = []string{"%s", "%q", "%+q", "%v", "%#v"}

    for _, err := range []error{КонецФайла, io.EOF} {
        for _, f := range verbs {
            fmt.Printf("%3s - \t"+f, f, err)
            fmt.Println()
        }
        fmt.Println()
    }
}

/*
 %s -    конец файла
 %q -    "конец файла"
%+q -    "\u043a\u043e\u043d\u0435\u0446
\u0444\u0430\u0439\u043b\u0430"
 %v -    конец файла
%#v -    &errors.errorString{s:"конец файла"}

 %s -    EOF
 %q -    "EOF"
%+q -    "EOF"
 %v -    EOF
%#v -    &errors.errorString{s:"EOF"}

*/
```

Дизайн предполагает, что с помощью `%+v` ошибку

```
foo: bar(nameserver 139): baz flopped
```

можно было бы вывести в детализированном виде следующем образом:

```
foo:
    file.go:123 main.main+0x123
```

```
--- bar(nameserver 139):
    some detail only text
    file.go:456
--- baz flopped:

    file.go:789
```

Стектрейс

Самое интересное. Если ошибке нужен стектрейс, то она должна была бы использовать тип из стандартного пакета:

```
package errstack

type Stack struct { /* ... */ }

// Format writes the stack information to p, but only
// if detailed printing is requested.
func (s *Stack) Format(p errors.Printer) {
    if p.Detail() {
        p.Printf(/* ... */)
    }

}

import ".../errstack"

type myError struct {
    msg        string
    stack      errstack.Stack
    underlying error
}

// ...

func (e *myError) Format(p errors.Printer) error {
    p.Printf(e.msg)
    e.stack.Format(p)
    return e.underlying
}

func newMyError(msg string, underlying error) error {
    return &myError{
```

```
        msg:        msg,
        underlying: underlying,
        stack:      errstack.New(),
    }
```

```
}
```

Бросается в глаза, что при этом мы будем записывать стек, даже если, например, детализированный вывод для ошибки не используется – а можно ли в целом обойти эту проблему?

Внимательный читатель заметил, что урок построен в сослагательном наклонении. И действительно, Russ Cox пишет:

This one is not as simple as Unwrap, and I won't go into the details here. As we discussed the design with the Go community over the winter, we learned that the design wasn't simple enough. It was too hard for individual error types to implement, and it did not help existing programs enough. On balance, it did not simplify Go development.

As a result of this community discussion, we abandoned this printing design*.*

Таким образом, изменения выше свет не увидели.

Тест "Format verb"

Какой спецификатор форматирования позволит нам вывести на экран строку в кавычках с гарантией содержания только ASCII-символов?

Напишите текст

____

Тест "Какие фичи появились в языке, начиная с Go 1.13?"

Выберите все подходящие ответы из списка

- ☐ Функция errors.Is
- ☐ Интерфейс errors.Wrapper
- ☐ Поддержка локализации ошибок
- ☐ Функция errors.Unwrap
- ☐ Поддержка стектрейса для ошибок
- ☐ Интерфейс errors.Printer
- ☐ Функция errors.As
- ☐ Интерфейс errors.Formatter
- ☐ Функция errors.Cause
- ☐ Детализированный вывод цепочки ошибок по `%+v`