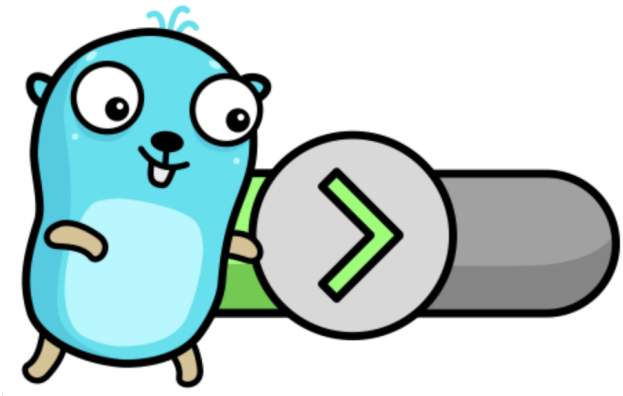
Константные ошибки

В этом уроке мы поговорим о том, как выдумывать себе проблемы, а затем решать их :)

Познакомимся с идеей **константных ошибок** и узнаем, чем опасны ошибки над **пустыми структурами**.



Постановка проблемы

Уже известный нам Dave Cheney ~~высосал из~~ обозначил у sentinel errors две проблемы.

1) Это чаще всего экспортируемые переменные уровня пакета, которые могут поменять все, кому не лень, выстрелив тем самым вам в ногу (исходник примера):

```
// constant-errors-problem/dirtyhacker/hack.go
package dirtyhacker

import "io"

func MutateEOF() {
    io.EOF = nil // Bugaga!

}
```

```go
// constant-errors-problem/main.go
package main

import (
    "fmt"
    "io"

    "constant-errors-problem/dirtyhacker"
)

func main() {
    err1 := io.EOF
    err2 := io.EOF
    fmt.Println(io.EOF)        // EOF
    fmt.Println(err1 == err2) // true

    // Меняем глобальный io.EOF,
    // в принципе это можно сделать в init пакета dirtyhacker.
    dirtyhacker.MutateEOF()

    fmt.Println(io.EOF)          // <nil>
    fmt.Println(err1 == io.EOF) // false

}
```

В примере выше во время работы `main` сторонний пакет поменял `io.EOF` и теперь он неконсистентен с переменными, которые схоронили в себя `io.EOF` до этого.

2) Как мы прекрасно знаем, даже в рамках одного пакета подобные переменные ведут себя не как константы, а скорее как синглтоны: две переменные, созданные через `errors.New` (даже с одинаковым текстом) никогда не будут равны:

```go
// https://goplay.tools/snippet/5DtAbcvFDmT
func main() {
    EOF1 := errors.New("EOF")
    EOF2 := errors.New("EOF")
    fmt.Println(errors.Is(EOF1, EOF2)) // false

}
```

Тест "Константные ошибки"

Дейв предлагает решение проблем, озвученных ранее, с помощью введения нового типа и изменения объявление sentinel ошибок.

Заполните пропуски ниже так, чтобы программа **не компилировалась** с фразой `cannot assign to EOF (declared const)`:

```
type Error <1>
func (e Error) Error() <2> { return <3> }

<4> EOF = Error("EOF")

func MutateEOF() {
    EOF = Error("new EOF") // cannot assign to EOF (declared const)

}
```

Заполните пропуски

<1> _____
<2> _____
<3> _____
<4> [var, const, type]

1) Что за bullshit?

Разберём подробнее проблемы и путь их решения.

Объявление sentinel ошибки через константу действительно защищает переменную от изменения:

```
const EOF = Error("EOF")
```

Но надо ли бояться этого? Не защищаемся ли мы от ситуации, которая никогда не произойдёт?

Как часто вы стреляете себе в ногу, меняя переменные из сторонних пакетов?

Так много вопросов и так мало ответов.

Сразу вспоминается хохма для старого петончика, когда мы глобально на всю систему меняем `True` на `False` (и наоборот, при желании):

```
$ python2.7
Python 2.7.16 (default, Sep 28 2019, 16:49:30)
[GCC 4.2.1 Compatible Apple LLVM 11.0.0 (clang-1100.0.33.8)] on
darwin
Type "help", "copyright", "credits" or "license" for more
information.

>>> __builtins__.True = False
>>> True
False
>>> True == False
True
```

Но будете ли вы делать так в реальной жизни?

Существует возражение, что подобную свинью может подложить код из вредоносного стороннего модуля.

Такое действительно имеет место быть, но мы парируем следующими тезисами:

- скорее всего вредоносный код сделает что-то более серьезное, чем изменение sentinel ошибки;
- ещё нужно подумать, как всё это дело превратить уязвимость – сам Дейв приводит следующий пример:

```
package innocent

import "crypto/rsa"

func init() {
    rsa.ErrVerification = nil
}
```

2) Опасная эквивалентность ошибок

Теперь разберём данный пример:

```
// https://goplay.tools/snippet/5DtAbcvFDmT

func main() {
    EOF1 := errors.New("EOF")
    EOF2 := errors.New("EOF")
    fmt.Println(errors.Is(EOF1, EOF2)) // false
}
```

То что Дейв Чейни считает сайд эффектом от реализации ошибок в Go, на самом деле является ожидаемым разработчиками языка поведением, на которое даже тест написан:

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

import (
    "errors"
    "fmt"
    "testing"
)

func TestNewEqual(t *testing.T) {
    // Different allocations should not be equal.
    if errors.New("abc") == errors.New("abc") {
        t.Errorf(`New("abc") == New("abc")`)
    }
    if errors.New("abc") == errors.New("xyz") {
        t.Errorf(`New("abc") == New("xyz")`)
    }

    // Same allocation should be equal to itself (not crash).
    err := errors.New("jkl")
    if err != err {
        t.Errorf(`err != err`)
    }

}
```

И мы целых два урока ("Стандартный пакет errors" и "Ошибки и тип ресивера") разглагольствовали о том, что необходимо сделать, чтобы ошибки с одинаковым текстом были различны.

Почему это важно?

Потому что семантика и значение sentinel ошибки определяется её объявлением и местом данного объявления.

Было бы странно, если бы два различных пакета предоставляли ошибки, которые были бы равны из-за того, что у них одинаковый текст. Более того это может привести к незаметным и сбивающим с толку багам, если функция может

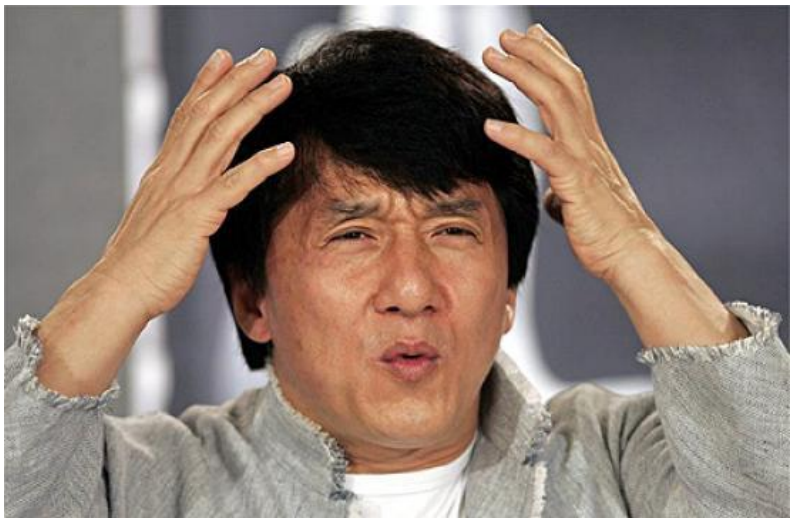
вернуть ошибки из абсолютно различных источников, а они будут считаться эквивалентными на уровне выше. Например,

```
import (
    "database/sql"
    "errors"

    "github.com/jackc/pgx"
)

func main() {
    fmt.Println(
        errors.Is(sql.ErrNoRows, pgx.ErrNoRows)) // false и должно
быть false!
}
```

Дейв же [предлагает](#) ввести новый константный тип ошибки глобально:

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

+type Error string
+
+func (e Error) Error() string { return string(e) }
+

// src/io/io.go

-var EOF = errors.New("EOF")

+const EOF = errors.Error("EOF")
```

тем самым зачесав ошибки из любых пакетов под одну гребёнку (исходник примера, чтобы поиграться).



Тест "Типовое неравенство"

Может идея с константными ошибками не так уж плоха, если мы будем объявлять их не через общий тип, а через тип, специфичный для пакета (и лучше даже неэкспортируемый)?

pkga

```
package pkga

type err string
func (e err) Error() string { return string(e) }


const ErrInvalidHost = err("invalid host")
```

pkgb

```
package pkgb

type err string
func (e err) Error() string { return string(e) }


const ErrInvalidHost = err("invalid host")
```

Что выведет код в первом и во втором случае?

```
func main() {
    fmt.Println(pkga.ErrInvalidHost == pkgb.ErrInvalidHost)

}
```

```
func main() {
    fmt.Println(error(pkga.ErrInvalidHost) ==
error(pkgb.ErrInvalidHost))

}
```

Выберите один вариант из списка

false
false

true
false

true
true

Ошибка компиляции
true

Ошибка компиляции
false

true
Ошибка компиляции

false
Ошибка компиляции

Ошибка компиляции
Ошибка компиляции

Опасный struct{}

В комментариях на Reddit к статье о константных ошибках в топе находится следующее предложение:

velco · 5y

For the case the error does not contain data, instead of constants, just use an empty struct:

```
type EOF struct{}
func(EOF) Error() string { return "end of file" }
// ...
func someFunc() error { return EOF{} }
// ...
err := someFunc()
if (err == EOF{})  {
// ...
}
```

9 Share Report Save

Какие здесь видятся проблемы?

Во-первых, как мы помним из урока "Ошибки и тип ресивера", из-за **value** ресивера у метода `Error` ошибку можно создавать через значение (не через указатель) и две созданные подобным образом ошибки будут равны:

```
// https://goplay.tools/snippet/TXayFM6QWN5

type EOF struct{}
func (b EOF) Error() string { return "end of file" }

var (
    a error = EOF{}
    b error = EOF{}
)

func main() {
    fmt.Println(a == b) // true
}
```

В целом в рамках одного пакета это, наверное, допустимо. Более того, судя по комментарию на Reddit выше, автор так и предлагает использовать ошибки не как sentinel, а через создание экземпляров на месте (что достаточно непривычно):

```
// https://goplay.tools/snippet/B9ktI8IUosL

func someFunc() error {
    return EOF{}
}

func main() {
    if err := someFunc(); err == (EOF{}) {
        // ...
    }
}
```

Во-вторых, вы можете захотеть исправить ситуацию выше и сделать методу `Error` **pointer** ресивер:

```
type EOF struct{}

func (b *EOF) Error() string { return "end of file" }
```

Но так вы попадёте в ловушку, подготовленную языком Go и особенностями реализации пустой (не имеющей размера) структуры. Как гласит спецификация:

Pointers to distinct zero-size variables may or may not be equal.

Из этого следует, что все созданные через указатель ошибки (а иначе нельзя из-за типа ресивера у `Error`) могут быть равны:

```
// https://goplay.tools/snippet/EYATlu_n7bE

type EOF struct{}
func (b *EOF) Error() string { return "end of file" }

var (
    a error = new(EOF)
```

```
    b error = new(EOF)
)

func main() {
    fmt.Println(a == b)         // true
    fmt.Printf("%p %p", a, b) // 0x57b458 0x57b458 <- Одинаковые
адреса!
}
```

Более того могут быть равны ошибки из разных пакетов и имеющие разные типы!

(если по своей сути они являются указателем на `struct{}`)

Какие выводы?

Использовать пустую структуру `struct{}` для организации ошибок с умом или избегать её в принципе.

Быть готовым к багам, которые она может породить.

Но из этого правила существует исключение, которое мы обсудим буквально через шаг 😉.

Тест "Ошибки на базе struct{} из разных пакетов"

Исходник примера.

Имеем два пакета, по sentinel `EOF` в каждом:

pkga

```
package pkga

type EOF struct{}

func (b EOF) Error() string {
    return "end of file"
}
```

pkgb

```
package pkgb

type EOF struct{}

func (b EOF) Error() string {
    return "end of file"
}
```

Что в таком случае выведет код ниже?

```
var (
    aVal error = pkga.EOF{}
    aPtr error = new(pkga.EOF)

    bVal error = pkgb.EOF{}
    bPtr error = new(pkgb.EOF)
)

func main() {
    fmt.Println(aVal == bVal) // Сравниваем два EOF из разных
пакетов.
    fmt.Println(aPtr == bPtr) // Сравниваем два *EOF из разных
пакетов.
    fmt.Println(aVal == bPtr) // Сравниваем EOF и *EOF из разных
пакетов.

}
```

Выберите один вариант из списка

true
true
true

false
true
false

false
false
false

false
false
true

true
true
false

true
false
false

true
false
true

false
true
true

А как же opaque errors?

Как мы выяснили, "опасность" ошибок в виде пустых структур заключается в их использовании в качестве **sentinel errors** (обработка через `==` или `errors.Is`).

Но если вы используете их для организации **opaque errors** (обработка через интерфейс и `errors.As`), то в целом ничего страшного нет, более того к ошибкам-пустым-структурам удобно цеплять методы-маркеры, необходимые для удовлетворения тому или иному интерфейсу:

```
// https://goplay.tools/snippet/VPSieXqPN1b

type NotReadyToConfirmError struct{}
func (e *NotReadyToConfirmError) Error() string     { return "not
ready for confirm yet" }
func (e *NotReadyToConfirmError) IsTemporary() bool { return true }

type OrderNotFoundError struct{}
func (e *OrderNotFoundError) Error() string     { return "order not
found" }
func (e *OrderNotFoundError) IsTemporary() bool { return false }

type AlreadyConfirmedError struct{}
func (e *AlreadyConfirmedError) Error() string { return "already
confirmed" }
func (e *AlreadyConfirmedError) ShouldSkip()    {}

// ...
var v interface {
```

```
    ShouldSkip()
}
if errors.As(confirm(), &v) {
    fmt.Println("unimportant error")

}
```

Вот так незаметно мы ~~начали за здравие, а кончили за упокой~~ от константных ошибок перешли к нашим любимым **opaque errors**.

Какая здесь мораль? Да, как обычно:

- доверяй, но проверяй (не взирая на ранги и авторитеты);
- знай инструменты, которыми пользуешься.