github.com/cockroachdb/errors (часть 1)

Рассмотрим еще одну крайне популярную библиотеку по работе с ошибками – **github.com/cockroachdb/errors**.

В целом модуль чего-то кардинально нового в саму механику работы с ошибками не привносит, но содержит пару интересных фишек, о которых мы и поговорим.

Вообще библиотека позиционирует себя как готовое решение по замене **github.com/pkg/errors**. Будем посмотреть!

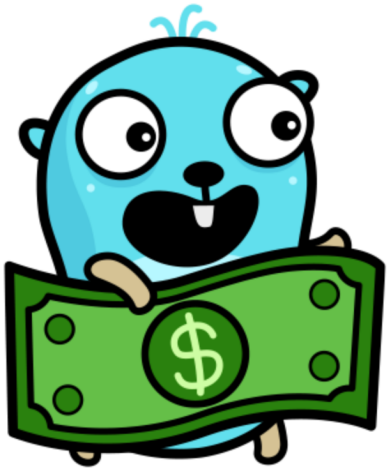
Эффективный стектрейс?

Начнём со стектрейсов.

Судя по таблице Features модуль **github.com/cockroachdb/errors** умеет эффективно записывать стектрейс возникшей ошибки:

standard wrappers with efficient stack trace capture: ✔

Проверим, сможем ли мы сэкономить, если будем переезжать на него с уже знакомого **github.com/pkg/errors** (исходник примера).



Воспользуемся приёмами из предыдущих уроков.

Врапим через **github.com/pkg/errors**:

```
// +build pkg

package main

import (
    "github.com/pkg/errors"
)

func GimmeDeepError(depth int) error {
    if depth == 1 {
        return errors.New("ooops, an error on level 1")
    }
    return errors.Wrapf(GimmeDeepError(depth-1), "error happened on
level %d", depth)

}
```

Аналогичный код для **github.com/cockroachdb/errors**:

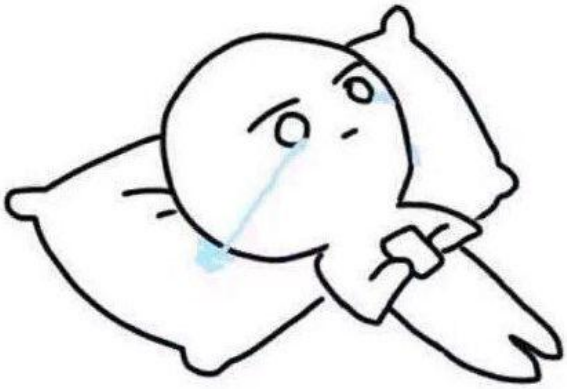
```
// +build cockroach

package main

import (
    "github.com/cockroachdb/errors"
)
```

```
func GimmeDeepError(depth int) error {
    if depth == 1 {
        return errors.New("ooops, an error on level 1")
    }
    return errors.Wrapf(GimmeDeepError(depth-1), "error happened on
level %d", depth)
}
```

Замеряем бенчмарком из предыдущего урока:

```
$ go test -tags pkg -benchmem -bench . > pkg.txt
$ go test -tags cockroach -benchmem -bench . > cdb.txt
$ benchstat -alpha 1.1 pkg.txt cdb.txt
name                  old time/op    new time/op    delta
GimmeDeepError/1-8      434ns ± 0%    1002ns ± 0%  +130.88%
GimmeDeepError/2-8     1.10µs ± 0%    2.14µs ± 0%   +95.35%
GimmeDeepError/4-8     2.73µs ± 0%    4.58µs ± 0%   +68.01%
GimmeDeepError/8-8     6.01µs ± 0%    9.53µs ± 0%   +58.48%
GimmeDeepError/16-8    13.4µs ± 0%    20.5µs ± 0%   +52.50%
GimmeDeepError/32-8    32.4µs ± 0%    46.4µs ± 0%   +43.07%

name                  old alloc/op   new alloc/op   delta
GimmeDeepError/1-8       304B ± 0%      416B ± 0%   +36.84%
GimmeDeepError/2-8       672B ± 0%      816B ± 0%   +21.43%
GimmeDeepError/4-8     1.41kB ± 0%    1.62kB ± 0%   +14.77%
GimmeDeepError/8-8     2.88kB ± 0%    3.22kB ± 0%   +11.70%
GimmeDeepError/16-8    5.83kB ± 0%    6.42kB ± 0%   +10.18%
GimmeDeepError/32-8    11.7kB ± 0%    12.8kB ± 0%    +9.43%

name                  old allocs/op  new allocs/op  delta
GimmeDeepError/1-8       3.00 ± 0%      7.00 ± 0%  +133.33%
GimmeDeepError/2-8       8.00 ± 0%     12.00 ± 0%   +50.00%
GimmeDeepError/4-8       18.0 ± 0%      22.0 ± 0%   +22.22%
GimmeDeepError/8-8       38.0 ± 0%      42.0 ± 0%   +10.53%
GimmeDeepError/16-8      78.0 ± 0%      82.0 ± 0%    +5.13%

GimmeDeepError/32-8       158 ± 0%       162 ± 0%    +2.53%
```

На всех глубинах стека используемая функция `Wrapf` из **cockroachdb/errors** проигрывает аналогу из **pkg/errors**. Получается, не такая уж и эффективная запись стека (закроем глаза на то, что по факту и тут и там – копейки).



На самом деле здесь нет ничего удивительного, так как если заглянуть поглубже в код, то можно увидеть интересный комментарий к функции, которая, собственно, реализует запись стека:

```
package withstack

// callers mirrors the code in github.com/pkg/errors,
// but makes the depth customizable.
func callers(depth int) *stack {
    const numFrames = 32
    var pcs [numFrames]uintptr
    n := runtime.Callers(2+depth, pcs[:])
    var st stack = pcs[0:n]
    return &st

}
```

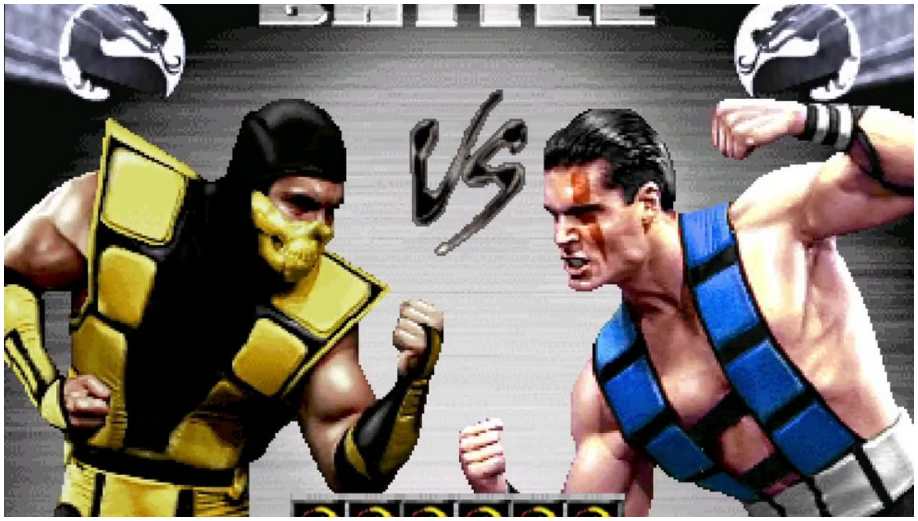
Всё то же самое, что и в **pkg/errors**, поэтому ожидать лучшей производительности не стоит. В теории будет либо так же, либо хуже, а на деле оказывается хуже из-за более навороченных абстракций вокруг ошибки.

Резюме

Если гонимся за ещё более "дешёвым" стектрейсом для наших ошибонек, то нам не сюда.

Тест "Кто победил?"

Какой модуль по результатам предыдущего шага оказался более эффективным в вопросе записи стектрейса?



Выберите один вариант из списка

- ☐ errors
- ☐ github.com/cockroachdb/errors
- ☐ github.com/pkg/errors

Больше врапинга для бога врапинга

Путь **cockroachdb/errors** не порадовал нас стректрейсом, зато у него есть целый зоопарк Wrap-функций:

- `WithAssertionFailure(err error)`
- `WithContextTags(err error, ctx context.Context)`
- `WithDetail(err error, msg string)`
- `WithDetailf(err error, format string, args ...interface{})`
- `WithDomain(err error, domain Domain)`
- `WithHint(err error, msg string)`
- `WithHintf(err error, format string, args ...interface{})`
- `WithIssueLink(err error, issue IssueLink)`

- `WithMessage(err error, msg string)`
- `WithMessagef(err error, format string, args ...interface{})`
- `WithSafeDetails(err error, format string, args ...interface{})`
- `WithSecondaryError(err error, additionalErr error)`
- `WithStack(err error)`
- `WithStackDepth(err error, depth int)`
- `WithTelemetry(err error, keys ...string)`
- `Wrap(err error, msg string)`
- `Wrapf(err error, format string, args ...interface{})`
- `WrapWithDepth(depth int, err error, msg string)`
- `WrapWithDepthf(depth int, err error, format string, args ...interface{})`

В основе `*WithDepth`-функций и лежит пропатченная `callers`, которую мы видели ранее. Когда применять ту или иную функцию – оставляем на самостоятельное изучение.

Помните портянки от повсеместного `errors.Wrap`?

cockroachdb/errors тоже страдает подобным, но:

- цепочка имеет более приятное форматирование: легче читать, есть информация о типах, врапах и т.д. (см. ниже);
- **cockroachdb** старается выявить и убрать дубликаты стека (см. `[...repeated from below...]`).

```
func main() {
    fmt.Printf("%+v", GimmeDeepError(5))
}

/*
```

```
error happened on level 5: error happened on level 4: error happened
on level 3: error happened on level 2:
ooops, an error on level 1
(1) attached stack trace
  -- stack trace:
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.main
  |     /04-non-standard-modules/cockroach-expensive-stack/main.go:8
  | runtime.main
  |     /usr/local/go/src/runtime/proc.go:225
Wraps: (2) error happened on level 5
Wraps: (3) attached stack trace
  -- stack trace:
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.main
  |     /04-non-standard-modules/cockroach-expensive-stack/main.go:8
  | runtime.main
  |     /usr/local/go/src/runtime/proc.go:225
Wraps: (4) error happened on level 4
Wraps: (5) attached stack trace
  -- stack trace:
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.main
  |     /04-non-standard-modules/cockroach-expensive-stack/main.go:8
```

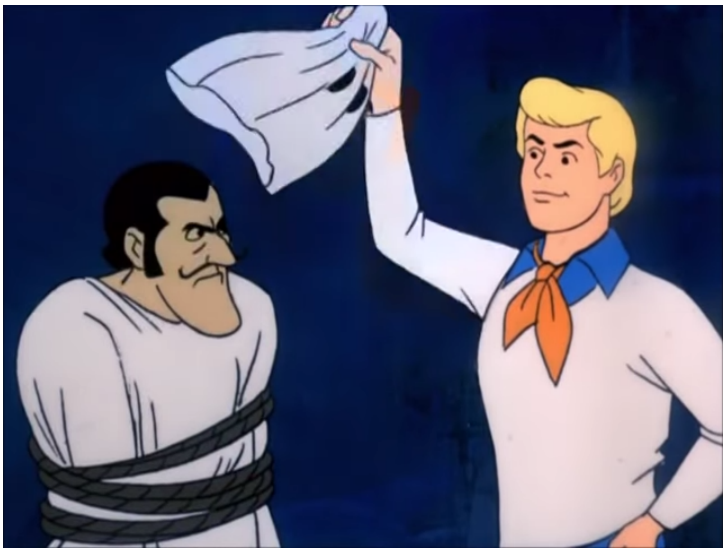
```
  | runtime.main
  |     /usr/local/go/src/runtime/proc.go:225
Wraps: (6) error happened on level 3
Wraps: (7) attached stack trace
  -- stack trace:
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | [...repeated from below...]
Wraps: (8) error happened on level 2
Wraps: (9) attached stack trace
  -- stack trace:
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
11
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.GimmeDeepError
  |
/04-non-standard-modules/cockroach-expensive-stack/wrap_cockroach.go:
13
  | main.main
  |     /04-non-standard-modules/cockroach-expensive-stack/main.go:8
  | runtime.main
  |     /usr/local/go/src/runtime/proc.go:225
  | runtime.goexit
  |     /usr/local/go/src/runtime/asm_arm64.s:1130
Wraps: (10) ooops, an error on level 1
Error types: (1) *withstack.withStack (2) *errutil.withPrefix (3)
*withstack.withStack (4)
 *errutil.withPrefix (5) *withstack.withStack (6) *errutil.withPrefix
(7) *withstack.withStack (8)
 *errutil.withPrefix (9) *withstack.withStack (10) *errutil.leafError
```

```
*/
```

Также "тараканьи" ошибки умеют в `%#v`, что может быть полезно при дебаге:

```
func main() {
    fmt.Printf("%#v", GimmeDeepError(2)) // errors и pkg/errors такое не поддерживают!
}

/*
&withstack.withStack{
    cause: &errutil.withPrefix{
        cause: &withstack.withStack{
            cause: &errutil.leafError{msg:"ooops, an error on level 1"},
            stack: &withstack.stack{0x104da9f09, 0x104da9f30, 0x104da9e38, 0x104b6cf4c, 0x104b9d214},
        },
        prefix: "error happened on level ‹2›",
    },
    stack: &withstack.stack{0x104da9f25, 0x104da9e38, 0x104b6cf4c, 0x104b9d214},
}

*/
```

errors.UnwrapOnce, errors.UnwrapAll



Поговорили о врапинге, а как – в обратную сторону?

errors.UnwrapOnce и errors.UnwrapAll - полезные в хозяйстве функции, предназначенные, как ни странно, для раскручивания цепочек вложенных в друг друга ошибок.

Базовая функция `errors.UnwrapOnce` (аналог `errors.Unwrap` из стандартной библиотеки) возвращает следующую ошибку в цепочке. Прикольно, что из коробки есть совместимость с **pkg/errors** и стандартным пакетом **errors**:

```
package errbase

// Sadly the go 2/1.13 design for errors has promoted the name
// `Unwrap()` for the method that accesses the cause, whilst the
// ecosystem has already chosen `Cause()`. In order to unwrap
// reliably, we must thus support both.
//
// See: https://github.com/golang/go/issues/31778
//
func UnwrapOnce(err error) (cause error) {
    switch e := err.(type) {
    case interface{ Cause() error }:
        return e.Cause()
    case interface{ Unwrap() error }:
        return e.Unwrap()
    }
    return nil

}
```

(обратите внимание на комментарий к функции)

Где это можно использовать?

В очередной раз вернемся к **opaque errors**. У нас есть сетевая ошибка *net.DNSError, которая при определённых условиях является временной. Мы хотим знать об этом, чтобы при желании повторить целевую операцию (исходник примера).

Если брать классическое исполнение функции `IsTemporary`, то оно ломается, если ошибка была обернута:

```
type isTemporary interface {
    Temporary() bool
```

```
}

func IsTemporary(err error) bool {
    e, ok := err.(isTemporary) // Приводим ошибку к интерфейсу
isTemporary, тем самым проверяя поведение.
    return ok && e.Temporary()

}
```

Немного доработаем напильником эту функцию, не забыв про `errors.UnwrapOnce` и `errors.UnwrapAll`, и всё станет сильно лучше:

```
// IsTemporaryOnce считает цепочку ошибок временной, если хотя бы
одна
// из ошибок в ней была временной.
func IsTemporaryOnce(err error) bool {
    for c := err; c != nil; c = errors.UnwrapOnce(c) {
        e, ok := c.(isTemporary)
        if ok && e.Temporary() {
            return true
        }
    }
    return false
}

// IsTemporary считает цепочку ошибок временной, только если
// корневая ошибка в ней была временной.
func IsTemporary(err error) bool {
    c := errors.UnwrapAll(err)
    e, ok := c.(isTemporary)
    return ok && e.Temporary()

}
```

Проверяем:

```
func main() {
    dnsErr := &net.DNSError{IsTemporary: true} // Произошла сетевая
ошибка.

    err := fmt.Errorf("second wrap: %w",
        fmt.Errorf("first wrap: %w", dnsErr))

    fmt.Printf("is temporary: %t\n", IsTemporaryOnce(err))     // is
temporary: true
```

```
    fmt.Printf("is temporary at root: %t\n", IsTemporary(err)) // is temporary at root: true

}
```

Кажется, это победа. `errors.UnwrapOnce` и `errors.UnwrapAll` крутят за нас цепочки и поддерживают совместимость с **pkg/errors** и **errors**.

Резюме

Функции из **cockroachdb/errors** полезны, если у вас в проекте мешанина с оборачиванием ошибок – где-то через `fmt.Errorf`, где-то через `errors.Wrapf`.

Но на самом деле решать эту проблему через затаскивание к себе третьей библиотеки – сомнительное удовольствие. Лучше скопипастите необходимый функционал к себе, а в дальнейшем зарефачьте код под оборачивание в едином стиле.

A little copying is better than a little dependency.

Тест "UnwrapOnce и Wrap"

Ссылка на пример.

Берём ошибку `err`, оборачиваем её тремя разными пакетами и после каждого врапа сразу же пытаемся развернуть `UnwrapOnce`'ом:

```
import (
    stderrors "errors"
    "fmt"

    cdberrors "github.com/cockroachdb/errors"
    pkgerrors "github.com/pkg/errors"
)


func main() {
```

```
    err := stderrors.New("an error")

    stdErr := fmt.Errorf("wrap: %w", err)
    fmt.Println(cdberrors.UnwrapOnce(stdErr))

    pkgErr := pkgerrors.Wrap(err, "wrap 1")
    fmt.Println(cdberrors.UnwrapOnce(pkgErr))

    cockroachErr := cdberrors.Wrap(err, "wrap 2")
    fmt.Println(cdberrors.UnwrapOnce(cockroachErr))

}
```

На экране получаем вывод:

```
an error
wrap 1: an error
wrap 2: an error
```

Вопрос: почему так, а не

```
an error
an error
an error
```

?

Выберите один вариант из списка

- ☐ UnwrapOnce не работает с ошибками, обёрнутыми через pkg/errors или errors.
- ☐ У ошибок, обёрнутых через pkg/errors и cockroachdb/errors нет метода Unwrap.
- ☐ Wrap в pkg/errors и cockroachdb/errors оборачивает ошибку в несколько "слоёв".

errors.If

```
package errors

import "github.com/cockroachdb/errors/markers"

// If iterates on the error's causal chain and returns a predicate's
// return value the first time the predicate returns true.
func If(err error, pred func(err error) (interface{}, bool))
(interface{}, bool) {
    return markers.If(err, pred)

}
```

Интересная фишка библиотеки, которая позволяет применять функцию-предикат ко всем ошибкам из цепочки. Когда предикат возвращает `true`, `errors.If` вернет какое-то значение произвольного типа из предиката и сам флажок `true`.

Где это может быть полезно?

На предыдущем шаге мы в очередной раз улучшали **opaque errors**. Теперь попробуем провернуть то же самое, но с `errors.If()` (исходник примера).

Если немного доработать функцию `IsTemporary`, то мы сможем использовать её в качестве параметра в `errors.If`:

```
type isTemporary interface {
    Temporary() bool
}

func IsTemporary(err error) (interface{}, bool) {
    e, ok := err.(isTemporary)
    return e, ok && e.Temporary() // Возвращаем не только флаг, но и ошибку.

}
```

Проверяем:

```
func main() {
    dnsErr := &net.DNSError{IsTemporary: true} // Произошла сетевая ошибка.

    err := fmt.Errorf("second wrap: %w",
        fmt.Errorf("first wrap: %w", dnsErr))

    e, ok := errors.If(err, IsTemporary)
    fmt.Printf("%T is temporary: %t", e, ok) // *net.DNSError is temporary: true

}
```

Looks good!

errors.IsAny

```
switch {
case errors.Is(err, context.Canceled),
    errors.Is(err, io.EOF),
    errors.Is(err, os.ErrClosed),
    errors.Is(err, os.ErrNotExist):
    // Что-то делаем.

}
```

Если вы в своём коде наблюдаете подобные конструкции и у вас возникает жгучее желание уменьшить занимаемое ими на экране место, то тут на помощь приходит errors.IsAny (исходник примера):

```
if errors.IsAny(err, context.Canceled, io.EOF, os.ErrClosed,
os.ErrNotExist) {
    // ...

}
```

Что под капотом?

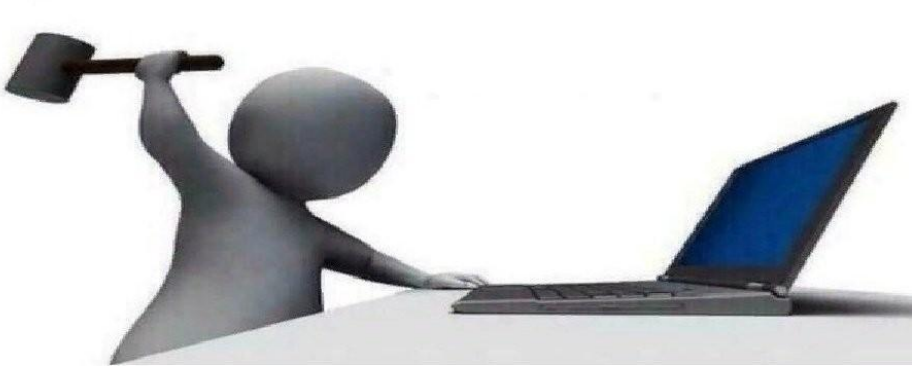
```
package errors

import "github.com/cockroachdb/errors/markers"

// IsAny is like Is except that multiple references are compared.
func IsAny(err error, references ...error) bool {
    return markers.IsAny(err, references...)

}
```

Мы видим, что `errors.IsAny` как и `errors.If` реализованы с помощью какого-то **markers**.

Исходный код функций довольно объёмный – оставляем это на вас, но тот же `IsAny` состоит по сути из следующих последовательных проверок для всех ошибок `chainErr` в цепочке `err`:

1. Проверяем равенство по значению `chainErr == references[i]` (sentinel errors style).
2. Проверяем результат вызова метода `chainErr.Is(references[i])`, если таковой имеется.
3. Проверяем, есть ли совпадающие маркировки у `chainErr` и `references[i]`.

Вроде, всё понятно, кроме фразы "совпадающие маркировки" в пункте 3. Да кто такие эти ваши маркировки? Скоро узнаем!



Тест "Какая асимптотическая сложность у errors.IsAny?"

Через **Big O Notation**, где

- n – длина цепочки ошибок `err`;
- k – длина `references`.
- j – максимальная длина среди цепочек ошибок в `references`.

При этом будем считать, что `n > k > j`.

Пришло время почувствовать себя на собесе в Яндекс и вкурить [реализацию IsAny](#).

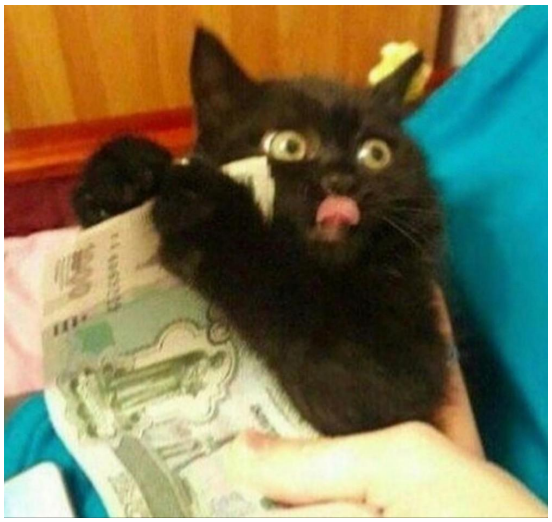
Выберите один вариант из списка

- ☐ O(n * (n² + k * j))
- ☐ O(k + n²)
- ☐ O(k * n² * j)
- ☐ O(n * (n + k²))
- ☐ O(k)
- ☐ O(log(n))
- ☐ O(k²)
- ☐ O(n³)
- ☐ O(n * k)

- [ ] O(n + (n + k²))
- [ ] O(n² + k * n * j)
- [ ] O(1)
- [ ] O(n)

Задача "Уносим к себе errors.IsAny"

Ссылка на заготовку.



Нам страшно понравилась функция `errors.IsAny`, и мы решили утащить её к себе в проект.

Всякие магические маркировки нас не интересуют, поэтому задача – реализовать её уже известными нам способами:

- Из импортов доступная только стандартная библиотека.
- Убедитесь, что ваша реализация проходит тесты.

errors.Mark

Вернёмся к тем загадочным маркировкам, на которые наткнулись, когда ковыряли `errors.IsAny`.

Функция `errors.Mark` позволяет **отождествить одну ошибку с совершенно другой**.

В теории это может быть полезно, когда в коде ошибки явно проверяются на соответствие через `errors.Is()`, но в место проверки выше по стеку вызовов возвращается не та ошибка, которая ожидается. А маркировки как раз дают возможность пройти эту проверку, не меняя существующий код. Очень сложно, звучит как костыль.

Давайте попробуем разобраться на примере (исходник).

Допустим, у нас возникает ошибка, которую мы по какой-то причине хотим обработать как `io.EOF`. Для этого мы берём и маркируем её как `io.EOF` с помощью `errors.Mark`:

```
package main

import (
    "fmt"
    "io"

    "github.com/cockroachdb/errors"
)

func main() {
    err := io.ErrUnexpectedEOF
    err = errors.Mark(err, io.EOF) // Помечаем ошибку как io.EOF.
    err = errors.Wrap(err, "something other happened")

    if errors.Is(err, io.EOF) { // true
        fmt.Println("error is io.EOF")
    }

    if errors.Is(err, io.ErrUnexpectedEOF) { // true
        fmt.Println("error is io.ErrUnexpectedEOF")
    }

}
```

На экране получим:

```
error is io.EOF
```

```
error is io.ErrUnexpectedEOF
```

Похоже, что работает так, как и заявлено – функция `errors.Is` сработала в обоих случаях.

Ложечка дёгтя

Не стоит забывать, что эти фокусы исправно работают, только если вы используете `errors.Is` из **github.com/cockroackdb/errors**. Аналоги `Is` из стандартной библиотеки или **github.com/pkg/errors** уже работать не будут, так как они ничего не знают о маркировках и о том, как их нужно проверять в принципе.

Как это работает?

Начнем с рассмотрения функции markers.Mark, лежащей в основе одноимённой функции в `errors`:

```
// github.com/cockroachdb/errors/markers/markers.go
package markers

// Mark creates an explicit mark for the given error, using
// the same mark as some reference error.
func Mark(err error, reference error) error {
    if err == nil {
        return nil
    }
    refMark := getMark(reference)
    return &withMark{cause: err, mark: refMark}

}

// github.com/cockroachdb/errors/markers/markers.go
package markers

// getMark computes a marker for the given error.
func getMark(err error) errorMark {

    // Ключевая функция при работе с маркировками.
```

```
    if m, ok := err.(*withMark); ok { // Если ошибка уже промаркирована, то вернём маркировку.
        return m.mark
    }

    // Начинаем собирать маркировку.
    // Достаём из ошибки сообщение, а потом начинается "магия" с извлечением информации о типах в цепочке.
    m := errorMark{
        msg: safeGetErrMsg(err),
        types: []errorspb.ErrorTypeMark{errbase.GetTypeMark(err)},
    }
    for c := errbase.UnwrapOnce(err); c != nil; c = errbase.UnwrapOnce(c) {
        m.types = append(m.types, errbase.GetTypeMark(c))
    }
    return m

}
```

Взглянем внимательнее на функцию `getMark`.

Зачем так сложно? Зачем нужна "магия" с типами? Давайте представим, что мы просто взяли и убрали всю эту возню. Тогда как сравнивать между собой маркировки? По тексту ошибки? Именно для того, чтобы не сравнивать ошибки по тексту друг с другом (что, как мы помним, неправильно), собирается информация о типах ошибок, вплоть до пакетов, откуда они взялись:

```
// github.com/cockroachdb/errors/errbase/encode.go
package errbase

// GetTypeMark retrieves the ErrorTypeMark for a given error object.
// This is meant for use in the markers sub-package.
func GetTypeMark(err error) errorspb.ErrorTypeMark {
    _, familyName, extension := getTypeDetails(err, false /*onlyFamily*/)
    return errorspb.ErrorTypeMark{FamilyName: familyName, Extension: extension}
}
```

Взглянем теперь на то, как это используется, например, в markers.ls.

В конце функции есть код, посвящённый маркировкам и помеченный интересным комментарием. Условно "мы проверяем маркировки в самом конце, потому что это медленно". Так что стоит про это помнить: игры с типами в `getMark` действительно небесплатные:

```
// github.com/cockroachdb/errors/markers/markers.go
package markers

// Is determines whether one of the causes of the given error or any
// of its causes is equivalent to some reference error.
//
// As in the Go standard library, an error is considered to match a
// reference error if it is equal to that target or if it implements
a
// method Is(error) bool such that Is(reference) returns true.
func Is(err, reference error) bool {
    // ...

    // Not directly equal. Try harder, using error marks. We don't
this
    // during the loop above as it may be more expensive.
    //
    // Note: there is a more effective recursive algorithm that
ensures
    // that any pair of string only gets compared once. Should the
    // following code become a performance bottleneck, that algorithm
    // can be considered instead.
    refMark := getMark(reference)
    for c := err; c != nil; c = errbase.UnwrapOnce(c) {
        if equalMarks(getMark(c), refMark) {
            return true
        }
    }

    // ...

}
```

Само сравнение маркировок несложное:

```
// github.com/cockroachdb/errors/markers/markers.go
package markers

// equalMarks compares two error markers.
func equalMarks(m1, m2 errorMark) bool {
```

```
    if m1.msg != m2.msg { // Сначала сравниваем тексты маркировок.
        return false
    }
    for i, t := range m1.types {
        if !t.Equals(m2.types[i]) { // Затем сравниваем типы внутри
маркировок.
            return false
        }
    }
    return true
}
```

```
// github.com/cockroachdb/errors/errorspb/markers.go
package errorspb

func (m ErrorTypeMark) Equals(o ErrorTypeMark) bool {
    return m.FamilyName == o.FamilyName && m.Extension == o.Extension

}
```

Резюме

Тяжело придумать пример из реальной жизни, когда это всё может оказаться полезным, но сам подход и идея интересны и заслуживают внимания.

errors.Mark + errors.Is gotchas

Ранее мы разобрались с тем, как работает `errors.Mark` и даже немного заглянули в `errors.Is`.

Ниже пример, который показывает некоторые особенности поведения этой функции (исходник примера):

```
package main

import (
    stderrors "errors"
    "fmt"
    "io"
```

```
    "github.com/cockroachdb/errors"
)

type NotFoundError struct {
    message string
}

func (e *NotFoundError) Error() string {
    return e.message
}

func main() {
    err1 := &NotFoundError{"object not found"}
    err2 := &NotFoundError{"object not found"}

    err := io.ErrUnexpectedEOF
    err = errors.Mark(err, err1)
    err = errors.Wrap(err, "something other happened")

    if errors.Is(err, err1) { // Ожидаемо true.
        fmt.Println("err is err1")
    }

    if errors.Is(err, err2) { // Не очень ожидаемо true.
        fmt.Println("err is err2")
    }

    if errors.Is(err1, err2) { // Ещё менее ожидаемо true.
        fmt.Println("err1 is err2")
    }

    if stderrors.Is(err1, err2) { // Ожидаемо false.
        fmt.Println("err1 is err2")
    }

}
```

У нас есть две **разных** ошибки `err1` и `err2` одного и того же типа `*NotFoundError`:

```
err1 := &NotFoundError{"object not found"}
err2 := &NotFoundError{"object not found"}
```

```
err := io.ErrUnexpectedEOF
err = errors.Mark(err, err1)

err = errors.Wrap(err, "something other happened")
```

1) Очевидно, что

```
errors.Is(err, err1) == true
```

ведь мы пометили `err` как `err1` через `errors.Mark`.

2) Неочевидно, что и

```
errors.Is(err, err2) == true
```

ведь `err2` мы ничем не помечали, а эквивалентной ни к `err1`, ни к `err` она не является.

3) Ещё более неочевидно, что и

```
errors.Is(err1, err2) == true
```

Это же совершенно разные экземпляры типа! Стандартная библиотека подобное не пропускает:

```
stderrors.Is(err1, err2) == false
```

Почему?

Во всём виноваты маркировки. Как мы помним, маркировки ошибок считаются идентичными, если у них совпадают сообщения и типы, участвовавшие в цепочке:

```
package markers

// equalMarks compares two error markers.
func equalMarks(m1, m2 errorMark) bool {
    if m1.msg != m2.msg { // Сначала сравниваем тексты маркировок.
        return false
    }
    for i, t := range m1.types {
        if !t.Equals(m2.types[i]) { // Затем сравниваем типы внутри маркировок.
            return false
        }
    }
    return true
}
```

Поэтому две ошибки в виде `*NotFoundError` становятся эквивалентными, хотя физически это абсолютно разные указатели. `stderrors.Is` работает иначе.

Резюме

Мы бы не сказали, что эти особенности являются крайне неканоничными и заслуживают порицания, но про них всё-таки стоит знать и помнить. Мало ли, придётся столкнуться.

Промежуточные выводы

Мы вскользь посмотрели на **github.com/cockroachdb/errors** и уже в целом можем определить для себя – интересная штука или лучше положить её в дальний ящик и забыть?

В следующем уроке мы посмотрим на ещё парочку особенностей этого модуля и на этом закончим.