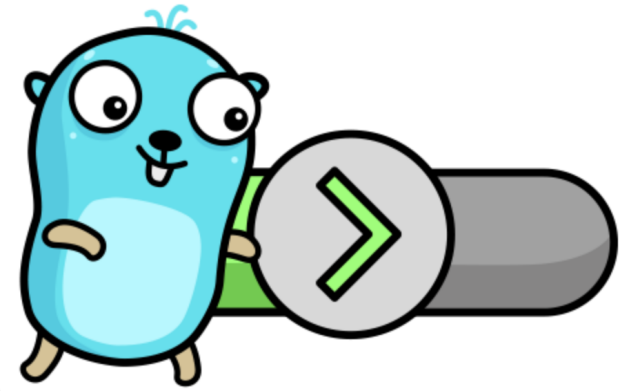
github.com/pkg/errors (часть 3)

В этом уроке мы подробнее поговорим о том, за что все любят **github.com/pkg/errors** и продолжают его использовать – **стектрейсы**.



Забавный факт

Мы уже говорили, что модуль **github.com/pkg/errors** появился раньше релиза Go 1.13.

Поэтому по сути `errors.Is`, `errors.As`, `errors.Unwrap` и `fmt.Errorf(%w)` – это необходимый минимум по работе с ошибками из **github.com/pkg/errors**, доведённый до ума.

Необходимый минимум как раз **не включает** в себя работу со стектрейсами. Сейчас разберёмся почему так.

Стектрейс – дорогая штука?

Есть мнение, что стектрейс – это дорого из-за того, что как минимум надо раскрутить стек до самого верха, чтобы его записать. Давайте сразу, не вдаваясь в детали почему, попробуем проверить, действительно ли это так.

Наша цель сейчас – убедиться, правда ли функции из библиотеки **github.com/pkg/errors** работают медленнее функций из **errors**:

- `New` vs `New`;
- `Wrapf` vs `fmt.Errorf`.

Слегка ознакомиться с тем, как можно писать бенчмарки в Go можно здесь (подробнее об этом мы рассказываем на курсе "**Продвинутое тестирование в Go**"), сейчас же сосредоточимся на рассмотрении конкретного кода.

New vs New (исходник примера)

Напишем две супер простых функции:

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

    "github.com/pkg/errors"
)

func GimmeError() error {
    return stderrors.New("ooops, an error") // Просто возвращает ошибку с текстом.
}
```

```
func GimmePkgError() error {
    return errors.New("ooops, an error") // Возвращает ошибку вместе
с текстом и стектрейсом.

}
```

Проверим

```
func main() {
    fmt.Printf("%+v", GimmePkgError())
    fmt.Println("\n---")
    fmt.Printf("%+v", GimmeError())
}

/*
ooops, an error
main.GimmeError

/advanced-dealing-with-errors-in-go/examples/04-non-standard-modules/
expensive-stack-new/main.go:11
main.main

/advanced-dealing-with-errors-in-go/examples/04-non-standard-modules/
expensive-stack-new/main.go:19
runtime.main
        /usr/local/go/src/runtime/proc.go:225
runtime.goexit
        /usr/local/go/src/runtime/asm_arm64.s:1130
---
ooops, an error

*/
```

И забенчмарчим

```
import "testing"

// ErrGlobal экспортируемая переменная уровня пакета,
// необходимая для предотвращений оптимизаций компилятора.
var ErrGlobal error

func BenchmarkGimmeError(b *testing.B) {
    for i := 0; i < b.N; i++ {
        ErrGlobal = GimmeError()
```

```
    }
}

func BenchmarkGimmePkgError(b *testing.B) {
    for i := 0; i < b.N; i++ {
        ErrGlobal = GimmePkgError()
    }
}
```

Результаты

Прежде чем посмотреть на результаты отметим, что это всё относительно и зависит от железа, на котором бенчмарк гонялся. Все нижеприведённые цифры стоит, скорее, мерить в "попугаях", чем в наносекундах.

```
$ go test -benchmem -bench .
BenchmarkGimmeError-8          72333199         16.59 ns/op         16
B/op         1 allocs/op

BenchmarkGimmePkgError-8        2791952         430.0 ns/op         304
B/op         3 allocs/op
```

Как мы видим, создание ошибки через стандартную библиотеку работает **быстрее в ~26 раз** – стектрейс в ошибке действительно недешёвое удовольствие относительно "голой" ошибки.

Но посмотрим на цифры – **430 ns**, даже не микросекунда! Кажется не очень серьёзным в контексте веба, где время обработки запроса обычно измеряют в миллисекундах.

А что будет при создании ошибки "поглубже"? – узнаем дальше.

We need to go deeper

К бенчмаркам и функциям на предыдущем шаге можно придраться: где же в реальности вы видели стек вызовов глубиной 1?

Реальный код – это когда N-ная по вложенности функция возвращает ошибку с помощью, например, `errors.New`, а все, кто выше, начинают это дело врапить.

Давайте воссоздадим подобную ситуацию.

---

errors.Wrapf vs fmt.Errorf (исходник примера)

В учебных целях глубокий стек вызовов удобнее всего имитировать рекурсией, что мы и сделаем.

Врапим через стандартную библиотеку:

```
// +build std

package main

import (
    "errors"
    "fmt"
)

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

```
    return fmt.Errorf("error happened on level %d: %w", depth,
GimmeDeepError(depth-1))
}


// error happened on level 2: ooops, an error on level 1
```

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)
}

/*
ooops, an error on level 1
main.GimmeDeepError

/advanced-dealing-with-errors-in-go/examples/04-non-standard-modules/
expensive-stack-wrap/wrap_pkg.go:11
main.GimmeDeepError

/advanced-dealing-with-errors-in-go/examples/04-non-standard-modules/
expensive-stack-wrap/wrap_pkg.go:13
main.main

/advanced-dealing-with-errors-in-go/examples/04-non-standard-modules/
expensive-stack-wrap/main.go:8
runtime.main
        /usr/local/go/src/runtime/proc.go:225
runtime.goexit
        /usr/local/go/src/runtime/asm_arm64.s:1130
error happened on level 2
main.GimmeDeepError
```

```
/advanced-dealing-with-errors-in-go/examples/04-non-standard-modules/
expensive-stack-wrap/wrap_pkg.go:13
main.main

/advanced-dealing-with-errors-in-go/examples/04-non-standard-modules/
expensive-stack-wrap/main.go:8
runtime.main
        /usr/local/go/src/runtime/proc.go:225
runtime.goexit
        /usr/local/go/src/runtime/asm_arm64.s:1130%

*/
```

(вывод выглядит страшновато, но об этом мы поговорим в следующих шагах)

Пишем бенчмарк:

```
package main

import (
    "strconv"
    "testing"
)

// ErrGlobal экспортируемая переменная уровня пакета,
// необходимая для предотвращений оптимизаций компилятора.
var ErrGlobal error

var depths = []int{1, 2, 4, 8, 16, 32}

func BenchmarkGimmeDeepError(b *testing.B) {
    for _, depth := range depths {
        b.Run(strconv.Itoa(depth), func(b *testing.B) {
            for i := 0; i < b.N; i++ {
                ErrGlobal = GimmeDeepError(depth)
            }
        })
    }

}
```

Замеряем:

```
$ go test -tags std -benchmem -bench . > std.txt
```

```
$ go test -tags pkg -benchmem -bench . > pkg.txt
$ benchstat -alpha 1.1 std.txt pkg.txt

name                   old time/op    new time/op    delta
GimmeDeepError/1-8       18.6ns ± 0%   435.4ns ± 0%  +2245.91% <- !!!
GimmeDeepError/2-8        217ns ± 0%    1115ns ± 0%   +413.35%
GimmeDeepError/4-8        614ns ± 0%    2786ns ± 0%   +353.89%
GimmeDeepError/8-8       1.43µs ± 0%    5.79µs ± 0%   +305.82%
GimmeDeepError/16-8      3.28µs ± 0%   13.43µs ± 0%   +309.20%
GimmeDeepError/32-8      7.46µs ± 0%   32.61µs ± 0%   +336.91%

name                   old alloc/op   new alloc/op   delta
GimmeDeepError/1-8       16.0B ± 0%    304.0B ± 0%  +1800.00% <- !!!
GimmeDeepError/2-8        112B ± 0%      672B ± 0%   +500.00%
GimmeDeepError/4-8        368B ± 0%     1408B ± 0%   +282.61%
GimmeDeepError/8-8       1.23kB ± 0%    2.88kB ± 0%   +133.77%
GimmeDeepError/16-8      4.34kB ± 0%    5.83kB ± 0%    +34.31%
GimmeDeepError/32-8      16.3kB ± 0%    11.7kB ± 0%    -28.31%

name                   old allocs/op  new allocs/op  delta
GimmeDeepError/1-8         1.00 ± 0%      3.00 ± 0%   +200.00%
GimmeDeepError/2-8         3.00 ± 0%      8.00 ± 0%   +166.67%
GimmeDeepError/4-8         7.00 ± 0%     18.00 ± 0%   +157.14%
GimmeDeepError/8-8         15.0 ± 0%      38.0 ± 0%   +153.33%
GimmeDeepError/16-8        31.0 ± 0%      78.0 ± 0%   +151.61%

GimmeDeepError/32-8        63.0 ± 0%     158.0 ± 0%   +150.79%
```

Ожидаемо, что **github.com/pkg/errors** проигрывает на всех глубинах, при этом на небольшой глубине потеря производительности особенно заметна.

Но посмотрим на самую "глубокую" ошибку – **32.61µs**, стоит ли это обсуждений?

Резюме

Запись стека действительно недешёвая операция, если смотреть относительно стандартной библиотеки. Но если смотреть относительно общей задержки в веб-сервисах, то можно к удивлению обнаружить, что по сути **стек ничего не стоит**.

Пишите в комментариях, что думаете по этому поводу.

Кроме того существуют следующие причины, по которым разработчики не отказываются от **github.com/pkg/errors** в пользу стандартной библиотеки:

- На самом деле все эти бенчмарки условны: в общем случае ошибки случаются сильно реже, чем выполняются позитивные сценарии (такое же мнение имеет сам Dave Cheney – автор библиотеки).

- Если ваше приложение работает недостаточно быстро, то работа со ошибками – это скорее всего последнее, что вы будете оптимизировать.

- Люди приходят из мира Java, C++ и пр. и отсутствие стектрейса вызывает у них когнитивный диссонанс.

В любом случае эти доводы неубедительны для core-разработчиков языка Go, поэтому **стектрейса в стандартной библиотеке мы не наблюдаем**. Мы ещё коснёмся данного факта в модуле "**Будущее ошибок в Go 2**".

Но всё же, если в вашем проекте ошибки случаются достаточно часто (повторяются какие-то операции, или специфика приложения связана с негативными сценариями и т.д.), то стоит лишний раз задуматься над использованием **github.com/pkg/errors**.

Нюанс повсеместного Wrap

Петька:

> *— Василь Иваныч, а что такое «НЮАНС»?*

...

Воспользуемся функцией из предыдущего шага и посмотрим, что будет, если заврапить ошибку несколько раз:

```
// https://goplay.tools/snippet/Tfphf-fDRdK
```

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

/*
ooops, an error on level 1
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:11
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.main
     /tmp/sandbox893275734/prog.go:17
runtime.main
     /usr/local/go-faketime/src/runtime/proc.go:255
runtime.goexit
     /usr/local/go-faketime/src/runtime/asm_amd64.s:1581
error happened on level 2
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.main
     /tmp/sandbox893275734/prog.go:17
runtime.main
     /usr/local/go-faketime/src/runtime/proc.go:255
runtime.goexit
     /usr/local/go-faketime/src/runtime/asm_amd64.s:1581
error happened on level 3
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
     /tmp/sandbox893275734/prog.go:13
main.main
```

```
	/tmp/sandbox893275734/prog.go:17
runtime.main
	/usr/local/go-faketime/src/runtime/proc.go:255
runtime.goexit
	/usr/local/go-faketime/src/runtime/asm_amd64.s:1581
error happened on level 4
main.GimmeDeepError
	/tmp/sandbox893275734/prog.go:13
main.GimmeDeepError
	/tmp/sandbox893275734/prog.go:13
main.main
	/tmp/sandbox893275734/prog.go:17
runtime.main
	/usr/local/go-faketime/src/runtime/proc.go:255
runtime.goexit
	/usr/local/go-faketime/src/runtime/asm_amd64.s:1581
error happened on level 5
main.GimmeDeepError
	/tmp/sandbox893275734/prog.go:13
main.main
	/tmp/sandbox893275734/prog.go:17
runtime.main
	/usr/local/go-faketime/src/runtime/proc.go:255
runtime.goexit
	/usr/local/go-faketime/src/runtime/asm_amd64.s:1581
*/
```

Мы получили целую портянку – по стеку на каждый `errors.Wrap`, хотя наиболее полезным является только самый глубокий стектрейс.

Из этого можно сделать следующий вывод: старайтесь не использовать `errors.Wrap` налево и направо, так как при каждом вызове он добавляет в цепочку ошибку со стектрейсом. Много стектрейсов в цепочке – страшные логи.

Поэтому по возможности используйте `Wrap`/`WithStack` только тогда, когда получаете ошибку от вызова функции из внешней библиотеки (из слоя **Frameworks & Drivers**, если говорить языком чистой архитектуры – обычно это поход в базулю или в любой другой внешний сервис), чтобы сократить количество лишних стектрейсов.

Если нужно добавить дополнительный контекст к ошибке на верхних уровнях, пользуйтесь `errors.WithMessage`:

```
// https://goplay.tools/snippet/fg8IlgjuumQ

package main

import (
    stderrors "errors"
    "fmt"

    "github.com/pkg/errors"
)

var errInternal = stderrors.New("ooops, an error on level 1")

func GimmeDeepError(depth int) error {
    if depth == 1 {
        return errors.WithStack(errInternal)
    }
    return errors.WithMessagef(GimmeDeepError(depth-1), "error
happened on level %d", depth)
}

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

/*
ooops, an error on level 1
main.GimmeDeepError
    /tmp/sandbox231190523/prog.go:14
main.GimmeDeepError
    /tmp/sandbox231190523/prog.go:16
main.GimmeDeepError
    /tmp/sandbox231190523/prog.go:16
main.GimmeDeepError
    /tmp/sandbox231190523/prog.go:16
main.GimmeDeepError
    /tmp/sandbox231190523/prog.go:16
main.main
    /tmp/sandbox231190523/prog.go:20
runtime.main
    /usr/local/go-faketime/src/runtime/proc.go:255
runtime.goexit
```

```
    /usr/local/go-faketime/src/runtime/asm_amd64.s:1581
error happened on level 2
error happened on level 3
error happened on level 4
error happened on level 5

*/
```

И про errors.New

Если вы создаёте новую ошибку через `errors.New` из **github/pkg/errors**, то не забывайте, что у созданной ошибки уже будет стектрейс. Выше по стеку вызовов также старайтесь пользоваться `errors.WithMessage`.

Ради интереса уберите в примере выше **stderrors** и посмотрите на получившийся вывод.

Тест "Выберите предпочтительный порядок функций"

Чтобы на самом верхнем уровне мы не получали несколько экранов логов.

Выберите все подходящие ответы из списка

errors.Wrap + errors.Wrap + errors.Wrap + errors.Wrap
errors.New + errors.WithMessage + errors.WithMessage + errors.WithMessage
errors.New + errors.Wrap + errors.WithMessage + errors.WithMessage
errors.WithStack + errors.Wrap + errors.WithMessage + errors.WithMessage
stderrors.New + errors.Wrap + errors.WithMessage + errors.WithMessage
errors.Wrap + errors.WithMessage + errors.Wrap + errors.WithMessage
errors.WithStack + errors.WithMessage + errors.WithMessage + errors.WithMessage
errors.Wrap + errors.WithMessage + errors.WithMessage + errors.WithMessage
errors.New + errors.WithStack + errors.WithMessage + errors.WithMessage
stderrors.New + errors.WithStack + errors.WithMessage + errors.WithMessage

Стоимость Wrapf + WithMessage

Ради интереса дополнительно замерим следующие варианты (исходник примера):

- Мы хотим отказаться от стектрейса в принципе: `errors.WithMessage` vs `fmt.Errorf`.
- Мы хотим врапить со стеком единожды, на самом глубоком уровне: `Wrap + WithMessage` vs `fmt.Errorf`.

Врапим через стандартную библиотеку:

```
// +build std

package main

import (
    "errors"
    "fmt"
)

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

}
```

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

```
// +build pkg.msg.only

package main

import (
    stderrors "errors"

    "github.com/pkg/errors"
)

func GimmeDeepError(depth int) error {
```

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

Врапим через **github.com/pkg/errors** со стеком только в самом низу:

```
// +build pkg.msg.stack

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.WithMessagef(GimmeDeepError(depth-1), "error
happened on level %d", depth)
}
```

Замеряем на написанном ранее бенчмарке:

```
$ go test -tags std -benchmem -bench . > std.txt
$ go test -tags pkg.msg.stack -benchmem -bench . > pkg-msg-stack.txt

$ go test -tags pkg.msg.only -benchmem -bench . > pkg-msg-only.txt
```

Сравниваем **std** и **pkg/errors** без стека:

```
$ benchstat -alpha 1.1 std.txt pkg-msg-only.txt
name                  old time/op    new time/op    delta
GimmeDeepError/1-8      18.3ns ± 0%    17.3ns ± 0%   -5.25%
GimmeDeepError/2-8       218ns ± 0%     112ns ± 0%  -48.49%
GimmeDeepError/4-8       594ns ± 0%     272ns ± 0%  -54.11%
```

```
GimmeDeepError/8-8      1.42µs ± 0%    0.61µs ± 0%  -57.01%
GimmeDeepError/16-8     3.30µs ± 0%    1.36µs ± 0%  -58.65%
GimmeDeepError/32-8     7.46µs ± 0%    2.73µs ± 0%  -63.41%

name                  old alloc/op   new alloc/op   delta
GimmeDeepError/1-8      16.0B ± 0%     16.0B ± 0%     ~
GimmeDeepError/2-8       112B ± 0%       80B ± 0%  -28.57%
GimmeDeepError/4-8       368B ± 0%      208B ± 0%  -43.48%
GimmeDeepError/8-8      1.23kB ± 0%    0.46kB ± 0%  -62.34%
GimmeDeepError/16-8     4.34kB ± 0%    0.98kB ± 0%  -77.50%
GimmeDeepError/32-8     16.3kB ± 0%     2.0kB ± 0%  -87.76%

name                  old allocs/op  new allocs/op  delta
GimmeDeepError/1-8        1.00 ± 0%      1.00 ± 0%     ~
GimmeDeepError/2-8        3.00 ± 0%      3.00 ± 0%     ~
GimmeDeepError/4-8        7.00 ± 0%      7.00 ± 0%     ~
GimmeDeepError/8-8        15.0 ± 0%      15.0 ± 0%     ~
GimmeDeepError/16-8       31.0 ± 0%      31.0 ± 0%     ~

GimmeDeepError/32-8       63.0 ± 0%      63.0 ± 0%     ~
```

Поразительно! Стандартная библиотека проигрывает по всем фронтам.

Сравниваем **std** и **pkg/errors** с единоразово взятым стеком:

```
$ benchstat -alpha 1.1 std.txt pkg-msg-stack.txt
name                  old time/op    new time/op    delta
GimmeDeepError/1-8      18.3ns ± 0%   439.9ns ± 0%  +2306.46%
GimmeDeepError/2-8       218ns ± 0%     625ns ± 0%   +186.70%
GimmeDeepError/4-8       594ns ± 0%     891ns ± 0%    +50.03%
GimmeDeepError/8-8      1.42µs ± 0%    1.37µs ± 0%     -3.94%
GimmeDeepError/16-8     3.30µs ± 0%    2.34µs ± 0%    -29.07%
GimmeDeepError/32-8     7.46µs ± 0%    3.90µs ± 0%    -47.74%

name                  old alloc/op   new alloc/op   delta
GimmeDeepError/1-8      16.0B ± 0%    304.0B ± 0%  +1800.00%
GimmeDeepError/2-8       112B ± 0%      368B ± 0%   +228.57%
GimmeDeepError/4-8       368B ± 0%      496B ± 0%    +34.78%
GimmeDeepError/8-8      1.23kB ± 0%    0.75kB ± 0%    -38.96%
GimmeDeepError/16-8     4.34kB ± 0%    1.26kB ± 0%    -70.86%
GimmeDeepError/32-8     16.3kB ± 0%     2.3kB ± 0%    -86.00%

name                  old allocs/op  new allocs/op  delta
```

```
GimmeDeepError/1-8          1.00 ± 0%        3.00 ± 0%   +200.00%
GimmeDeepError/2-8          3.00 ± 0%        5.00 ± 0%    +66.67%
GimmeDeepError/4-8          7.00 ± 0%        9.00 ± 0%    +28.57%
GimmeDeepError/8-8          15.0 ± 0%        17.0 ± 0%    +13.33%
GimmeDeepError/16-8         31.0 ± 0%        33.0 ± 0%     +6.45%

GimmeDeepError/32-8         63.0 ± 0%        65.0 ± 0%     +3.17%
```

На маленьких глубинах, **pkg/errors** очевидно медленнее, но посмотрите, что дальше! Начиная с 8-го вызова, взятый стек нивелируется тем, что `errors.WithMessagef` работает быстрее, чем `fmt.Errorf`. А при большом количестве врапов **pkg/errors** уже почти не проигрывает стандартной библиотеке по количеству аллокаций на операцию и сильно выигрывает по быстродействию и количеству выделенной памяти в целом.

Резюме

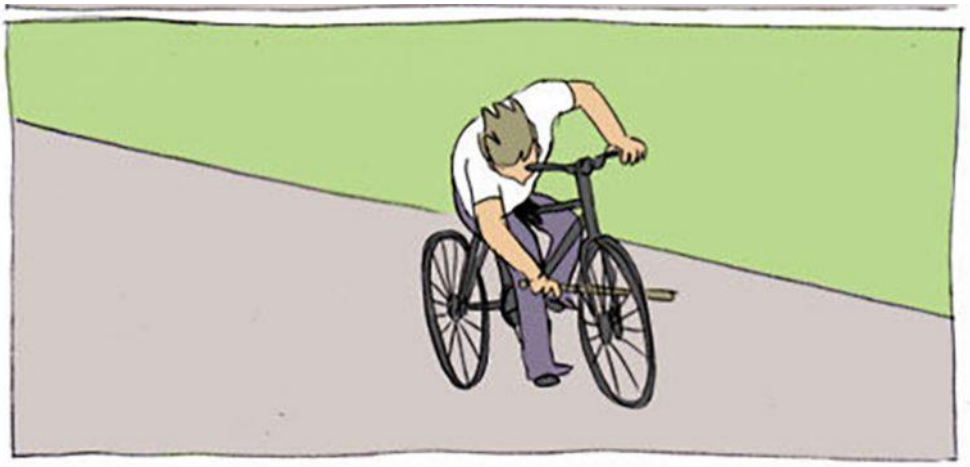
При грамотном использовании инструмента можно избежать его недостатков, на первый взгляд кажущихся неизбежными.

Мы видим, что в слоистых приложениях (где цепочка вызовов достаточно велика) можно с меньшими переживаниями за производительность использовать **github.com/pkg/errors**.

Также это хороший пример того, что стандартная библиотека не всегда идеальна в своих решениях (вспомнить тот же **encoding/json**, который не ускорял только ленивый).

Задача "TrimStackTrace"

Ссылка на заготовку.



Вам необходимо реализовать функцию, которая убирает из цепочки ошибок все стектрейсы, если они были:

```
// TrimStackTrace режет все стектрейсы в цепочке ошибок err.

func TrimStackTrace(err error) error

import "github.com/pkg/errors"

err := errors.Wrap(errors.WithStack(io.EOF), "read file")
fmt.Printf("%+v", TrimStackTrace(err))


// read file: EOF
```

Подсказки:

- Не нужно стараться физически "вырезать" стектрейсы через **unsafe**/**reflect**/etc. – достаточно, чтобы после применения `TrimStackTrace` при подробной печати ошибки (через `%+v`) на экран не выводилось ничего кроме непосредственно текста ошибки.
- Из импортов доступна только стандартная библиотека.

Задача "GetDeepestStackTrace"

Ссылка на заготовку.

Ваша задача – написать функцию `GetDeepestStackTrace`, достающую самый глубокий стектрейс из цепочки ошибок:

```
err := getManyTimesWrappedError()

st := GetDeepestStackTrace(err)

fmt.Printf("%+v\n\n", err)                // Будет выведен стектрейс всех ошибок.
fmt.Printf("%s\n", formatStackTrace(st)) // Будет выведен только самый глубокий стектрейс.
```

Когда это может быть полезно? Когда в проекте подзабивают на лучшие практики; на рефакторинг и убеждение коллег ресурсов нет, а вам в логах или внешних системах хочется видеть красивый стектрейс, а не дублирующие друг друга портянки.

Для реализации функции вам понадобится sentry.ExtractStacktrace из модуля github.com/getsentry/sentry-go:

ExtractStacktrace creates a new Stacktrace based on the given error.

```
import "github.com/getsentry/sentry-go"

func GetDeepestStackTrace(err error) *sentry.Stacktrace
```

К сожалению, мы не можем проверить эту задачу средствами Stepik, поэтому она остаётся на самостоятельное изучение, совесть и желание студента:

```
$ cd tasks/04-non-standard-modules/deepest-stacktrace
$ go test -v ./...
```

```
=== RUN   TestGetDeepestStackTrace_Nil
--- PASS: TestGetDeepestStackTrace_Nil (0.00s)
=== RUN   ExampleGetDeepestStackTrace
--- PASS: ExampleGetDeepestStackTrace (0.00s)
PASS
ok  tasks/04-non-standard-modules/deepest-stacktrace 0.191s
```

Препарируем sentry.ExtractStacktrace

Надеемся, что вам стало интересно и вы уже не обошли вниманием sentry.ExtractStacktrace. Но мы всё равно в целях увеличения кругозора лишний раз посмотрим на её устройство:

```go
package sentry

// ExtractStacktrace creates a new Stacktrace based on the given error.
func ExtractStacktrace(err error) *Stacktrace {

    // Вытаскивается метод ошибки, который возвращает стектрейс.
    method := extractReflectedStacktraceMethod(err)

    var pcs []uintptr

    // Стектрейс-методом вытаскиваем указатели счётчика команд.
    if method.IsValid() {
        pcs = extractPcs(method)
    } else {
        pcs = extractXErrorsPC(err)
    }

    if len(pcs) == 0 {
        return nil
    }

    frames := extractFrames(pcs) // Превращаем указатели во фреймы.
    frames = filterFrames(frames)

    stacktrace := Stacktrace{
        Frames: frames,
    }
```

```
    return &stacktrace

}
```

Функция `extractReflectedStacktraceMethod`, используя **рефлексию** (если тема интересует, напишите об этом в комментариях), пытается понять, есть ли у текущей ошибки `err` методы, которые возвращают стектрейс.

Причём ориентируется она на методы типов из известных библиотек для работы с ошибками со стектрейсом. Как мы видим, там учтён модуль **github.com/pkg/errors**, используемый нами:

```
package sentry

func extractReflectedStacktraceMethod(err error) reflect.Value {
    var method reflect.Value

    // https://github.com/pingcap/errors
    methodGetStackTracer :=
reflect.ValueOf(err).MethodByName("GetStackTracer")
    // https://github.com/pkg/errors
    methodStackTrace :=
reflect.ValueOf(err).MethodByName("StackTrace")
    // https://github.com/go-errors/errors
    methodStackFrames :=
reflect.ValueOf(err).MethodByName("StackFrames")

    if methodGetStackTracer.IsValid() {
        stacktracer :=
methodGetStackTracer.Call(make([]reflect.Value, 0))[0]
        stacktracerStackTrace :=
reflect.ValueOf(stacktracer).MethodByName("StackTrace")

        if stacktracerStackTrace.IsValid() {
            method = stacktracerStackTrace
        }
    }

    if methodStackTrace.IsValid() {
        method = methodStackTrace
    }
```

```
    if methodStackFrames.IsValid() {
        method = methodStackFrames
    }

    return method

}

// github.com/pkg/errors/errors.go
type withStack struct {
    error
    *stack // <-- *withStack начинает иметь метод StackTrace.
}

// github.com/pkg/errors/stack.go

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

func (s *stack) Format(st fmt.State, verb rune) { /* ... */ }
func (s *stack) StackTrace() StackTrace { /* ... */ }

type Frame uintptr

type StackTrace []Frame
```

Далее пытаемся использовать найденный метод в функции extractPcs, чтобы вытащить стектрейс. Ошибки из **github.com/pkg/errors** возвращают стектрейс в виде `[]Frame`, поэтому в нашем случае будут отрабатывать условия

- `stacktrace.Kind() == reflect.Slice`;
- `pc.Kind() == reflect.Uintptr`.

```
func extractPcs(method reflect.Value) []uintptr {
    var pcs []uintptr

    stacktrace := method.Call(make([]reflect.Value, 0))[0]

    if stacktrace.Kind() != reflect.Slice {
```

```
        return nil
    }

    for i := 0; i < stacktrace.Len(); i++ {
        pc := stacktrace.Index(i)

        if pc.Kind() == reflect.Uintptr {
            pcs = append(pcs, uintptr(pc.Uint()))
            continue
        }

        if pc.Kind() == reflect.Struct {
            field := pc.FieldByName("ProgramCounter")
            if field.IsValid() && field.Kind() == reflect.Uintptr {
                pcs = append(pcs, uintptr(field.Uint()))
                continue
            }
        }
    }

    return pcs

}
```

Далее полученные указатели превращаются во фреймы (в extractFrames с помощью runtime.CallersFrames), из которых уже можно составить стектрейс, к которому мы все привыкли:

```
func ExtractStacktrace(err error) *Stacktrace {
    // ...

    frames := extractFrames(pcs) // Превращаем указатели во фреймы.
    frames = filterFrames(frames)

    stacktrace := Stacktrace{
        Frames: frames,
    }

    return &stacktrace

}
```

Также обратите внимание на filterFrames, которая убирает фреймы от **runtime** и **testing**, поэтому будьте бдительны и если они вам нужны, то скорее всего следует скопировать себе реализацию `sentry.ExtractStacktrace` и поменять её под себя.

Что было дальше?

Библиотека не разрабатывается, а только поддерживается.

Судя по исходникам, модуль сохранил только фичи, связанные со стектрейсом, во имя обратной совместимости. Все остальные фичи, которые вошли в стандартную библиотеку, были переписаны на неё же саму.

Официальная позиция разработчиков **github.com/pkg/errors** на текущий момент – ждём Go 2.

UPD от января 2022 года:

This repository has been archived by the owner. It is now read-only.

Подведём итоги

- Мы познакомились с **github.com/pkg/errors** – прародителем большинства идей по обработке ошибок из стандартного пакета **errors**.

- При этом пакеты не взаимозаменяемы, и если вы не хотите наступать на грабли, то стоит, например, отказаться полностью от интерфейса `cause` и функции `errors.Cause` (можно запретить через тот же forbidigo, о котором мы поговорим в следующем модуле) в пользу `errors.Is` / `errors.As`.

- Основная фича **github.com/pkg/errors** – стектрейс. В целом он не даёт просадки по производительности: создание ошибки с записью стека

занимает << 100µs.

- Будьте готовы к портянкам стеков, если вы бездумно `Wrap`'ите. Лучший вариант использования **github.com/pkg/errors** – это наличие `Wrap`/`WithStack` для самой глубокой ошибки и `WithMessage(f)` на уровнях выше. Более того так мы уменьшаем оверхед от использования библиотеки, т.к. в целом `WithMessagef` быстрее `fmt.Errorf`.

- Библиотека не разрабатывается, а только поддерживается.