



Mail.Ru Group 667,93

667,93

Строим Интернет

Подпи



AloneCoder 5 октября 2017 в 18:44

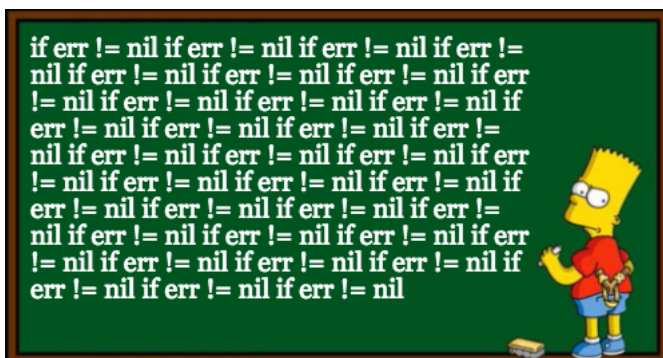
# Монады для Go-программистов

<https://awalterschulze.github.io/blog/post/monads-for-goprogrammers/>

Функциональное программирование, Проектирование и рефакторинг, Программирование, Go, Блог компании Mail.Ru Group

Перевод

Монады используются для компоновки функции (function composition) и избавления от связанного с этим утомительного однообразия. После лет программирования на Go необходимость повторять `if err != nil` превращается в рутину. Каждый раз, когда я пишу эту строку, я благодарю Gopher'ов за читабельный язык с прекрасным инструментарием, но в то же время проклинаю за то, что чувствую себя наказанным Бартом Симпсоном.



Подозреваю, что это чувство разделяют многие, но

```
if err != nil {
    log.Printf("This should still be interesting to a Go programmer " +
        "considering using a functional language, despite %v.", err)
}
```

Монады нужны не только для того, чтобы спрятать от нас обработки ошибок, но и для списковых включений, и для согласованности, и для задач.

## Не читайте это

Эрик Мейер (Erik Meijer) в своей статье [Introduction to Functional Programming Course on Edx](#) просит больше не писать о монадах, потому что так уже очень много сказано.

Помимо чтения этой статьи я рекомендую посмотреть серию видео Бартоша Милевски о теории категорий, которая завершается выступле лучшим объяснением монад, какое мне только встречалось.

Дальше не читайте!

## Функторы

Ну ладно (вздых)... Только помните: я вас предупреждал.

До монад нужно разобраться с функторами. Функтор — это надкласс (superclass) монады, т. е. все монады — это функторы. Я буду использовать функторы в дальнейшем объяснении сути монад, так что не пропускайте этот раздел.

Функтор можно считать контейнером, содержащим один тип элементов.

Например:

- Слайс с элементами типа `T`: `[]T` — контейнер, элементы в котором упорядочены в виде списка.
- Дерево `type Node<T> struct { Value T; Children: []Node<T> }` — контейнер, элементы которого упорядочены в виде дерева.
- Канал `<-chan T` — контейнер и похож на трубу, по которой течёт вода.
- Указатель `*T` — контейнер, который может быть пустым либо содержать один элемент.
- Функция `func(A) T` — контейнер наподобие сейфа, который придётся открыть ключом, чтобы увидеть хранящийся в нём элемент.
- Многочисленные возвращаемые значения `func() (T, error)` — контейнер, который, возможно, содержит один элемент. Ошибку можно рассматривать как часть контейнера. Здесь и далее мы будем называть `(T, error)` кортежем.

Программистам, владеющим не Go, а другими языками: в Go нет алгебраических типов данных или объединённых типов (union types). Это что вместо возврата функцией значения **или** ошибки здесь возвращается значение **и** ошибка, и одно из них обычно — `nil`. Иногда мы нарушу соглашение и возвращаем значение и ошибку, оба не `nil`, чтобы просто попытаться запутать друг друга. Или поразвлечься.

Самый популярный способ получить объединённые типы в Go: нужно иметь интерфейс (абстрактный класс) и переключатель типа (type switch) интерфейса.

Ещё один критерий, без которого контейнер не считается функтором, — необходимость реализации функции `fmap` для этого типа контейнера. Функция `fmap` применяет функцию к каждому элементу в контейнере без изменения контейнера или структуры.

```
func fmap(f func(A) B, aContainerOfA Container<A>) Container<B>
```

Использование функции `map` в качестве слайса — классический пример, который вы могли встречать в `mapreduce` в Hadoop, Python, Ruby и любом другом языке:

```
func fmap(f func(A) B, as []A) []B {
    bs := make([]B, len(as))
    for i, a := range as {
        bs[i] = f(a)
    }
    return bs
}
```

Мы также можем реализовать `fmap` для дерева:

```
func fmap(f func(A) B, atree Node<A>) Node<B> {
    btree := Node<B>{
        Value: f(atree.Value),
        Children: make([]Node<B>, len(atree.Children)),
    }
    for i, c := range atree.Children {
        btree.Children[i] = fmap(f, c)
    }
    return btree
}
```

Или для канала:

```
func fmap(f func(A) B, in <-chan A) <-chan B {
    out := make(chan B, cap(in))
    go func() {
        for a := range in {
            b := f(a)
            out <- b
        }
        close(out)
    }()
    return out
}
```

Или для указателя:

```
func fmap(f func(A) B, a *A) *B {
    if a == nil {
        return nil
    }
    b := f(*a)
    return &b
}
```

Или для функции:

```
func fmap(f func(A) B, g func(C) A) func(C) B {
    return func(c C) B {
        a := g(c)
        return f(a)
    }
}
```

Или для функции, возвращающей ошибку:

```
func fmap(f func(A) B, g func() (*A, error)) func() (*B, error) {
    return func() (*B, error) {
        a, err := g()
        if err != nil {
            return nil, err
        }
        b := f(*a)
        return &b, nil
    }
}
```

Все эти контейнеры с соответствующими реализациями `fmap` — функторы.

## Компоновка функций (Function Composition)

Теперь мы знаем, что функтор — это абстрактное название контейнера и что мы можем применять функцию к элементам внутри контейнера. Теперь перейдём к сути статьи — к абстрактной концепции монады.

Монада — это просто «украшенный» тип. Хм, полагаю, понятнее не стало, слишком абстрактно. Это типичная проблема всех объяснений с монад. Это как попытка растолковать, что такое побочный эффект: описание будет слишком общим. Так что давайте лучше разберёмся с причиной абстрактности монады. Причина в том, чтобы компоновать функции, которые возвращают эти украшенные типы.

Начнём с обычной компоновки функций, без украшенных типов. В этом примере мы скомпонуем функции `f` и `g` и вернём функцию, берущую входные данные, которые ожидаются `f`, и возвращающую выходные данные из `g`:

```
func compose(f func(A) B, g func(B) C) func(A) C {
    return func(a A) C {
        b := f(a)
        c := g(b)
    }
}
```

```

    return c
}
}

```

Очевидно, что это будет работать только в том случае, если результирующий тип `f` соответствует входному типу `g`.

Другая версия: компоновка функций, возвращающих ошибки.

```

func compose(
    f func(*A) (*B, error),
    g func(*B) (*C, error),
) func(*A) (*C, error) {
    return func(a *A) (*C, error) {
        b, err := f(a)
        if err != nil {
            return nil, err
        }
        c, err := g(b)
        return c, err
    }
}

```

Теперь попробуем абстрагировать эту ошибку в виде украшения (embellishment) `M` и посмотреть, что останется:

```

func compose(f func(A) M<B>, g func(B) M<C>) func(A) M<C> {
    return func(a A) M<C> {
        mb := f(a)
        // ...
        return mc
    }
}

```

Нам нужно вернуть функцию, берущую `a` в качестве входного параметра, так что начнём с объявления возвращающей функции. Раз у нас есть `f`, мы можем вызвать `f` и получить значение `mb` типа `M<b>`, но что дальше?

Мы не достигли цели, поскольку получилось слишком абстрактно. Я хочу сказать, что у нас есть `mb`, только что с ним делать?

Когда мы знали, что это ошибка, то могли проверить её, а теперь не можем, потому что она слишком абстрактна.

Но... если мы знаем, что наше украшение `M` — также функтор, то можем применить к нему `fmap`:

```

type fmap = func(func(A) B, M<A>) M<B>

```

Функция `g`, к которой мы хотим применить `fmap`, не возвращает простой тип вроде `c`, она возвращает `M<C>`. К счастью, для `fmap` это не проблема, зато меняется сигнатура типа:

```

type fmap = func(func(B) M<C>, M<B>) M<M<C>>)

```

Теперь у нас есть значение `mmc` типа `M<M<C>>`:

```

func compose(f func(A) M<B>, g func(B) M<C>) func(A) M<C> {
    return func(a A) M<C> {
        mb := f(a)
        mmc := fmap(g, mb)
        // ...
        return mc
    }
}

```

Мы хотим перейти от `M<M<C>>` к `M<C>`.

Для этого нужно, чтобы наше украшение `m` не просто было функтором, а имело и другое свойство. Это свойство — функция `join`, которая определена для каждой монады, как `fmap` была определена для каждого функтора.

```
type join = func(M<M<C>>) M<C>
```

Теперь мы можем написать:

```
func compose(f func(A) M<B>, g func(B) M<C>) func(A) M<C> {
    return func(a A) M<C> {
        mb := f(a)
        mmc := fmap(g, mb)
        mc := join(mmc)
        return mc
    }
}
```

Это означает, что если при украшении определены `fmap` и `join`, то мы можем скомпоновать две функции, возвращающие украшенные тип. Иными словами, необходимо определить эти две функции, чтобы тип был монадой.

## Join

Монады — это функторы, так что нам не нужно опять определять для них `fmap`. Нужно определить лишь `join`.

```
type join = func(M<M<C>>) M<C>
```

Теперь определим `join`:

- для списков, что позволит создать списковые выражения (list comprehensions);
- для ошибок, что позволит обрабатывать монадные ошибки (monadic error);
- и для каналов, что позволит создать согласованный конвейер (concurrency pipeline).

## Списковые выражения

Простейший способ начать — применить `join` к слайсам. Эта функция просто конкатенирует все слайсы.

```
func join(ss [][]T) []T {
    s := []T{}
    for i := range ss {
        s = append(s, ss[i]...)
    }
    return s
}
```

Давайте посмотрим, зачем нам снова понадобилась `join`, но в этот раз сосредоточимся на слайсах. Вот функция `compose` для них:

```
func compose(f func(A) []B, g func(B) []C) func(A) []C {
    return func(a A) []C {
        bs := f(a)
        css := fmap(g, bs)
        cs := join(css)
        return cs
    }
}
```

Если передать `a` в `f`, то получим `bs` типа `[]B`.

Теперь можем применить `fmap` к `[]B` `g`, что даст нам значение типа `[] []C`, а не `[]C`:

```
func fmap(g func(B) []C, bs []B) [][]C {
    css := make([][]C, len(bs))
    for i, b := range bs {
        css[i] = g(b)
    }
    return css
}
```

Вот для чего нам нужна `join`. Мы переходим от `css` к `cs`, или от `[][]C` к `[]C`.

Давайте рассмотрим более конкретный пример.

Если мы заменяем типы:

- `A` на `int`,
- `B` на `int64`,
- `C` на `string`.

Тогда наша функция становится:

```
func compose(f func(int) []int64, g func(int64) []string)
    func(int) []string
func fmap(g func(int64) []string, bs []int64) [][]string
func join(css [][]string) []string
```

Теперь можем использовать их в нашем примере:

```
func upto(n int) []int64 {
    nums := make([]int64, n)
    for i := range nums {
        nums[i] = int64(i+1)
    }
    return nums
}

func pair(x int64) []string {
    return []string{strconv.FormatInt(x, 10), strconv.FormatInt(-1*x, 10)}
}

c := compose(upto, pair)
c(3)
// "1", "-1", "2", "-2", "3", "-3"
```

Получается слайс нашей первой монады.

Любопытно, что именно так работают списковые выражения в Haskell:

```
[ y | x <- [1..3], y <- [show x, show (-1 * x)] ]
```

Но вы могли узнать их на примере Python:

```
def pair (x):
    return [str(x), str(-1*x)]

[y for x in range(1,4) for y in pair(x) ]
```

## Обработка монадных ошибок (Monadic Error)

Мы также можем определить `join` для функций, возвращающих значение и ошибку. Для этого сначала нужно опять вернуться к функции `fn` за некоторой идиосинкразии в Go:

```
type fmap = func(f func(B) C, g func(A) (B, error)) func(A) (C, error)
```

Мы знаем, что наша функция компоновки собирается вызвать `fmap` с функцией `f`, которая тоже возвращает ошибку. В результате сигнатура выглядит так:

```
type fmap = func(
    f func(B) (C, error),
    g func(A) (B, error),
) func(A) ((C, error), error)
```

К сожалению, кортежи в Go не относятся к объектам первого уровня, поэтому мы не можем просто написать:

```
((C, error), error)
```

Есть несколько путей обхода этого затруднения. Я предпочитаю функцию, потому что функции, которые возвращают кортежи, — это объект первого уровня:

```
(func() (C, error), error)
```

Теперь с помощью нашего ухищрения можем определить `fmap` для функций, которые возвращают значение и ошибку:

```
func fmap(
    f func(B) (C, error),
    g func(A) (B, error),
) func(A) (func() (C, error), error) {
    return func(a A) (func() (C, error), error) {
        b, err := g(a)
        if err != nil {
            return nil, err
        }
        c, err := f(b)
        return func() (C, error) {
            return c, err
        }, nil
    }
}
```

Это возвращает нас к основному пункту: функции `join` применительно к `(func() (C, error), error)`. Решение простое и выполняет для нас одну из проверок ошибки.

```
func join(f func() (C, error), err error) (C, error) {
    if err != nil {
        return nil, err
    }
    return f()
}
```

Теперь можем использовать функцию `compose`, поскольку уже определили `join` и `fmap`:

```
func unmarshal(data []byte) (s string, err error) {
    err = json.Unmarshal(data, &s)
    return
}

getnum := compose(
    unmarshal,
```

```

    strconv.Atoi,
)
getnum(`"1"`)
// 1, nil

```

В результате нам нужно выполнять меньше проверок ошибок, потому что монада делает это за нас в фоновом режиме с помощью функции

Вот ещё один пример, когда я чувствую себя Бартом Симпсоном:

```

func upgradeUser(endpoint, username string) error {
    getEndpoint := fmt.Sprintf("%s/oldusers/%s", endpoint, username)
    postEndpoint := fmt.Sprintf("%s/newusers/%s", endpoint, username)

    req, err := http.Get(genEndpoint)
    if err != nil {
        return err
    }
    data, err := ioutil.ReadAll(req.Body)
    if err != nil {
        return err
    }
    olduser, err := user.NewFromJson(data)
    if err != nil {
        return err
    }
    newuser, err := user.NewUserFromUser(olduser),
    if err != nil {
        return err
    }
    buf, err := json.Marshal(newuser)
    if err != nil {
        return err
    }
    _, err = http.Post(
        postEndpoint,
        "application/json",
        bytes.NewBuffer(buf),
    )
    return err
}

```

Технически `compose` может взять в качестве параметров более двух функций. Поэтому соберём все вышеупомянутые функции в один вызов перепишем наш пример:

```

func upgradeUser(endpoint, username string) error {
    getEndpoint := fmt.Sprintf("%s/oldusers/%s", endpoint, username)
    postEndpoint := fmt.Sprintf("%s/newusers/%s", endpoint, username)

    _, err := compose(
        http.Get,
        func(req *http.Response) ([]byte, error) {
            return ioutil.ReadAll(req.Body)
        },
        newUserFromJson,
        newUserFromUser,
        json.Marshal,
        func(buf []byte) (*http.Response, error) {
            return http.Post(
                postEndpoint,
                "application/json",
                bytes.NewBuffer(buf),
            )
        },
    )(getEndpoint)
    return err
}

```



Существует много других монад. Представьте две функции, возвращающие один и тот же тип украшения, которые вы хотите скомпоновать. Разберём ещё один пример.

## Согласованные конвейеры (Concurrent Pipelines)

Можно определить `join` для каналов.

```
func join(in <-chan <-chan T) <-chan T {
    out := make(chan T)
    go func() {
        wait := sync.WaitGroup{}
        for c := range in {
            wait.Add(1)
            go func(inner <-chan T) {
                for t := range inner {
                    out <- t
                }
            }(c)
            wait.Done()
        }
        wait.Wait()
        close(out)
    }()
    return out
}
```

Здесь у нас канал `in`, который снабжает нас каналами типа `T`. Сначала создадим канал `out`, запустим горутину для обслуживания канала, и вернём её. Внутри горютины запустим новые горютины для каждого из каналов, читающих из `in`. Эти горютины шлют в `out` свои входящие события, объединяя входные данные в один поток. Наконец, с помощью группы ожидания (`wait group`) удостоверимся в закрытии канала `out` получив все входные данные.

Иными словами, мы читаем все каналы `T` из `in` и передаём их в канал `out`.

Для программистов не на Go: мне нужно передать `c` в качестве параметра во внутреннюю горутину, потому что `c` — единственная переменная берущая значение каждого элемента в канале. Это значит, что если вместо создания копии значения посредством передачи его в качестве параметра мы просто используем его внутри замыкания, то, вероятно, сможем читать только из самого свежего канала. Это распространённая среди Go-программистов ошибка.

Мы можем определить функцию `compose` применительно к функции, возвращающей каналы.

```
func compose(f func(A) <-chan B, g func(B) <-chan C) func(A) <-chan C {
    return func(a A) <-chan C {
        chanOfB := f(a)
        return join(fmap(g, chanOfB))
    }
}
```

А из-за способа реализации `join` мы получаем согласованность почти даром.

```
func toChan(lines []string) <-chan string {
    c := make(chan string)
    go func() {
        for _, line := range lines {
            c <- line
        }
        close(c)
    }()
    return c
}

func wordsize(line string) <-chan int {
    removePunc := strings.NewReplacer(
        ",", "",
        "!", "",
        ":", "",
        ".", "",
    )
    return toChan(removePunc.Replace(line))
}
```

```

    "(" , "" ,
    ")" , "" ,
    ":" , "" ,
)
c := make(chan int)
go func() {
    words := strings.Split(line, " ")
    for _, word := range words {
        c <- len(removePunc.Replace(word))
    }
    close(c)
}()
return c
}

sizes := compose(
    toChan([]string{
        "Bart: Eat my monads!",
        "Monads: I don't think that's a very good idea.",
        "Lisa: If anyone wants monads, I'll be in my room.",
        "Homer: Mmm monads",
        "Maggie: (Pacifier Suck)",
    }),
    wordsize,
)
total := 0
for _, size := range sizes {
    if size == 6 {
        total += 1
    }
}
// total == 6

```

## Меньше жестикуляции

Это объяснение монад сделано практически на пальцах, и, чтобы воспринималось легче, я намеренно опустил многие вещи. Но есть ещё что, о чём я хочу рассказать.

Технически наша функция компоновки, определённая в предыдущей главе, называется `Kleisli Arrow`.

```
type kleisliArrow = func(func(A) M<B>, func(B) M<C>) func(A) M<C>
```

Когда люди говорят о монадах, они редко упоминают `Kleisli Arrow`, а для меня это стало ключом к пониманию сути монад. Если повезёт, будут объяснять суть с помощью `fmap` и `join`, но если вы невезучие, как я, то вам объяснят с помощью функции `bind`.

```
type bind = func(M<B>, func(B) M<C>) M<C>
```

Почему?

Потому что `bind` — это такая функция в Haskell, которую вам нужно реализовать для своего типа, если хотите, чтобы он считался монадой.

Повторим нашу реализацию функции `compose`:

```

func compose(f func(A) M<B>, g func(B) M<C>) func(A) M<C> {
    return func(a A) M<C> {
        mb := f(a)
        mmc := fmap(g, mb)
        mc := join(mmc)
        return mc
    }
}

```

Если бы была реализована функция `bind`, то мы могли бы просто вызвать её вместо `fmap` и `join`.

```
func compose(f func(A) M<B>, g func(B) M<C>) func(A) M<C> {
    return func(a A) M<C> {
        mb := f(a)
        mc := bind(mb, g)
        return mc
    }
}
```

Это означает, что `bind(mb, g) = join(fmap(g, mb))`.

Роль функции `bind` для списков будут выполнять в зависимости от языка `concatMap` или `flatMap`.

```
func concatMap([]A, func(A) []B) []B
```

## Внимательный взгляд

Я обнаружил, что в Go для меня стало стираться различие между `bind` и `Kleisli Arrow`. Go возвращает ошибку в кортеже, но кортеж — не первого уровня. Например, этот код не будет скомпилирован, потому что вы не можете посредством инлайнинга передавать результаты `f` в

```
func f() (int, error) {
    return 1, nil
}

func g(i int, err error, j int) int {
    if err != nil {
        return 0
    }
    return i + j
}

func main() {
    i := g(f(), 1)
    println(i)
}
```

Придётся написать так:

```
func main() {
    i, err := f()
    j := g(i, err, 1)
    println(j)
}
```

Или сделать так, чтобы `g` принимал функцию в качестве входных данных, потому что функции — это объекты первого уровня.

```
func f() (int, error) {
    return 1, nil
}

func g(ff func() (int, error), j int) int {
    i, err := ff()
    if err != nil {
        return 0
    }
    return i + j
}

func main() {
    i := g(f, 1)
    println(i)
}
```

Но это означает, что нашу функцию `bind`:

```
type bind = func(M<B>, func(B) M<C>) M<C>
```

определённую для ошибок:

```
type bind = func(b B, err error, g func(B) (C, error)) (C, error)
```

будет неприятно использовать, пока мы не преобразуем этот кортеж в функцию:

```
type bind = func(f func() (B, error), g func(B) (C, error)) (C, error)
```

Если присмотреться внимательно, то можно увидеть, что наш возвращаемый кортеж — тоже функция:

```
type bind = func(f func() (B, error), g func(B) (C, error)) func() (C, error)
```

И если присмотреться ещё раз, то окажется, что это наша функция `compose`, в которой `f` просто получает нулевые параметры:

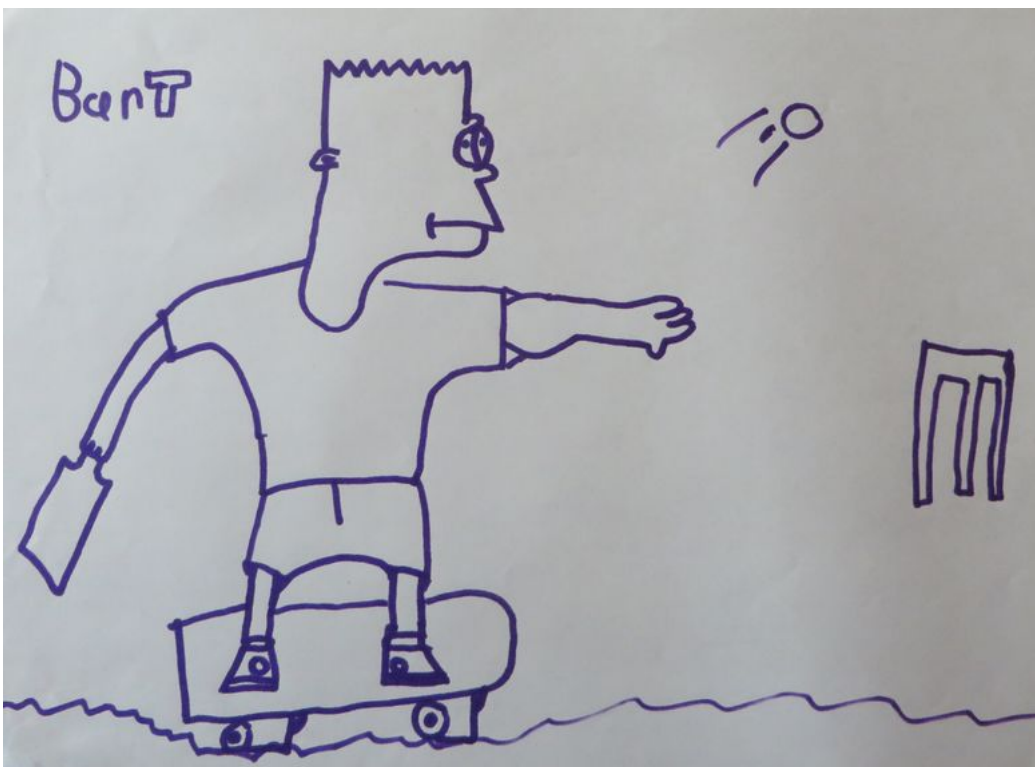
```
type compose = func(f func(A) (B, error), g func(B) (C, error)) func(A) (C, error)
```

Тадам! Мы получили свою `Kleisli Arrow`, просто несколько раз внимательно присмотревшись.

```
type compose = func(f func(A) M<B>, g func(B) M<C>) func(A) M<C>
```

## Заключение

Монады с помощью украшенных типов скрывают от нас рутинную логику компоновки функций, чтобы мы могли чувствовать себя не наказаным Бартом Симпсоном, а катающимся на скейте и метко кидающим мячик.



Если хотите попробовать в Go монады и другие концепции функционального программирования, можете делать это с помощью моего гена кода `GoDerive`.

Если вы действительно хотите заняться функциональным программированием, то обратите внимание на [Elm](#). Это статически типизированный функциональный язык программирования для фронтенд-разработки. Для функционального языка он прост в изучении, как Go прост для императивного. Я за день написал [руководство](#) и тем же вечером смог начать продуктивно работать. Создатель языка постарался сделать изучение простым, даже исключив необходимость разбираться с монадами. Лично мне нравится писать фронтенд на Elm в сочетании с бэкендом на Go. А если оба языка уже вам наскучили — то не переживайте, впереди ещё много интересного, вас ждёт Haskell.

Добавить метки

↑ +19 ↓ 100 13,8k 34

## ПОХОЖИЕ ПУБЛИКАЦИИ

 +69
  23,9k
  99
  179

 +53
  11,4k
  141
  23

 +69
  34,6k
  187
  230

Отслеживать новые в  почте

13/20

```
    },
    newUserFromJson,
    newUserFromUser,
    json.Marshal,
    func(buf []byte) (*http.Response, error) {
        return http.Post(
            postEndpoint,
            "application/json",
            bytes.NewBuffer(buf),
        )
    },
) (getEndpoint)
return err
}
```

Если описать словами, то это не «а, ну тут все понятно», а что-то ближе к «wtf здесь происходит вообще?»

Когда же я вижу первоначальный код (до конвертации в monad-style), то там все как-то сразу понятно. И не надо думать, как бы извернуться (в случае monad-style), если надо сделать что-то чуток отходящее от шаблона, типа дополнительную проверку вставить — просто вставляешь её куда надо и готово.

На всякий случай напоминаю, что при дизайне Go немаловажную роль играла простота и читаемость кода )

Я считаю, что лучше иметь на экране десяток проверок `err != nil`, чем сложнзавернутую конструкцию, которую и сам с трудом разберешь через пару месяцев. Не в PerlGolf же играем.

Ответить



**RomanStricpy** 05.10.17 в 23:26



картинкой не нашёл, но есть текстом:

Пишите код так, как будто сопровождать его будет склонный к насилию психопат, который знает, где вы живёте

Ответить



**LionAlex** 06.10.17 в 15:38



Такая подойдет?



Ответить



**index0h** 05.10.17 в 20:15



Стиль монад конечно подкупает `if err != nil`, но в одном из проектов я его попробовал (правда не знал, что это называется монадой)) ). Выглядит красивше, но фактическая последовательность выполнения у вас реализуется в рантайме, а не в коде, как следствие дебажить подобный код — занятие не из приятных, особенно в случае, если вызываемых функций штук 30+.

Посему к предложенному способу рекомендую относиться с большой осторожностью.

Ответить



**ngalayko** 05.10.17 в 20:16



Го не для этого делали. Передайте своим друзьям-функциональщикам.

Вот это <https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis> — чуть ли не единственное адекватное применение функционального программирования в го.

Ответить



**ngalayko** 05.10.17 в 20:20



увидите код из статьи в пул-реквесте завтра — просто заверните (но статья интересная, спасибо!)

Ответить

 **andreylartsev** 05.10.17 в 21:16  



Ммм а есть ещё такой вариант:

```
err, a = getA()
if err = nil {
err, b = getB(a)
if err = nil {
return getC(b)
}
}
return err, nil
```

Не попроще будет?

Ответить

 **tgz** 05.10.17 в 22:18  



Я бы не сказал, что с монадами стало проще. А раз полезного выхлопа нет, то зачем они такие нужны? Разве что для кодинга ради кодинга.

Ответить

 **alex\_blank**  05.10.17 в 22:53  



Monadic-style computation без синтаксической поддержки со стороны языка — это всегда печально и неудобно. В Haskell есть `do`-нотация. В JavaScript примеру, когда появились Promises (это по сути монады для асинхронных вычислений), все очень страдали, пока не появился `async/await` синтаксис них.

А вообще мне дико странно видеть, что в языке принципиально нет поддержки исключений, но при этом также нет никакого вменяемой альтернатив этому (типа монад с поддержкой на уровне синтаксиса). Мне кажется, это какой-то неправильный язык для неправильных пчел. My 2 cents.

Ответить

 **index0h** 05.10.17 в 23:26    



нет поддержки исключений, но при этом также нет никакого вменяемой альтернативы этому  
panic + defer + recover. Во многих ситуациях этот механизм гибче и удобнее, чем классический throw + catch

Ответить

 **Bonart** 06.10.17 в 10:55    



> Во многих ситуациях этот механизм гибче и удобнее, чем классический throw + catch

Да ладно?

defer реально удобне finally, но с ним реальный код финализации определяется только на момент исполнения.  
panic с throw — те же яйца вид в профиль.

А вот recover хуже catch: вся обработка конкретных ошибок и проброс выше только вручную.  
В целом механизмы полностью эквивалентные с точностью до синтаксического сахара.

Но что есть, то есть: механизм исключений в Go имеется и вполне рабочий, просто авторы языка исключения сильно не любят.

Ответить

 **index0h** 06.10.17 в 11:24    





А вот recover хуже catch: вся обработка конкретных ошибок и проброс выше только вручную.

Если вы ловите исключение и хотите его пробросить выше — его бросаете вручную. Разница есть только для языков с поддержкой множественных catch.

авторы языка исключения сильно не любят.

согласен

Ответить

 **SirEdvin** 05.10.17 в 22:57  



Чего только люди не сделают, лишь бы не добавлять в язык null-safe цепочки в духе `a1?.getA2()?.getA3()`

Ответить

 **shuhray** 06.10.17 в 00:03  



Вот интересно, я двигался в обратном направлении. Писал диплом на функциональном языке (ГАРФ) в 1984-м году. Затем изучал лямбда-исчисления категории и монады как математик. А последнее время стал немного программировать и из всех языков мне решительно нравится Lua!

Ответить

 aspcartman 06.10.17 в 00:26 # 📌

Если вас сильно раздражает проверка на `if err!=nil`, то в Gogland уже давно впилен мой патч, скрывающий их нафиг.

```

260         p[i] = c + " = " + placeholders[i]
261     }
262     table := record.Table()
263     query := fmt.Sprintf( format: "UPDATE %s SET %s WHERE %s = %s",
264         q.QualifiedView(table),
265         strings.Join(p, sep: ", "),
266         q.QuoteIdentifier(table.Columns()[table.PKColumnIndex()]),
267         q.Placeholder(len(columns)+1),
268     )
269
270     args := append(values, record.PKValue())
271     res, err := q.Exec(query, args...)
272     if err != nil : err ↗
275     ra, err := res.RowsAffected()
276     if err != nil : err ↗
279     if ra == 0 : ErrNoRows ↗
282     if ra > 1 : fmt.Sprintf("reform: %d rows by UPDATE by primary key.", ra) *
285     return nil
286 }
287


```

Ответить

 alex\_blank 06.10.17 в 01:51 # 📌 🔄

Костыльно-ориентированное программирование.

Ответить

 rule 06.10.17 в 05:35 # 📌 🔄

это очень смешно :-)

это как убираться заливав все вещи в шкаф. Типа шкаф открываешь и всё оттуда вываливается.


Я лично не понимаю почему эти кнструкции так напрягают людей, код как код.

Ответить

 aspcartman 06.10.17 в 10:14 # 📌 🔄

Потому что на 3 строчки полезного кода 12 строк `iferr!=nil{returnerr}\n` мантры. Это бессмысленная трата времени разработчика на печатание и на чтение всего этого бойлерплейта. Если их схлопывать, то хоть как-то это можно читать.

Ответить

 rule 06.10.17 в 10:17 # 📌 🔄


используйте панику. будет всего оин обработчик все хошибок и никаких проверок.

Ответить

 aspcartman 06.10.17 в 10:28 # 📌 🔄

Почему же тогда никто их не использует, наоборот, все радостно пишут эту мантру, и ругают за паники? =)

Ответить

 rule 06.10.17 в 10:39 # 📌 🔄

у каждого инструмента есть своё назначение, но в любом случае решение принимать вам. Хотите избежать цеопчек проверок на ошии есть инструмент для этого. Используйте его, учитывая его недостатки. Напимер не очень круто будет в библиотеке паниковать по каждой мелочи.

Ответить

 aspcartman 06.10.17 в 10:54 # 📌 🔄

Все правильно говорите. Все, что я хотел, это выразить мнение, что язык вынуждает разработчиков страдать, а предоставляемый альтернативный вариант, по большей части, не применим. Соответственно остается только хотябы на уровне IDE схлопывать бойлерплейт. Не понимаю, что Вам тут было смешно. Это грустно.

На самом деле ума не приложу, почему они не сделали автоматический неявный возврат ошибок и отдельный `defer`-оператор для +



вместо этого сделали недоошибки и паники. Вот например код.

```
func connect(addr string) (*connection, error) {
    con, err := db.Connect(addr)
    if err != nil {
        return nil, err
    }

    err = prepare(con)
    if err != nil {
        con.Close()
        return nil, err
    }

    return con, nil
}
```

Ошибка в нем следующая: если `prepare()` вывалится с паникой, коннект учтет (кто сказал, что `fd` закрывается в файнлайзере?). Горючить ли `defer` с `recover`? Наверное не горючить. Месяца через три петух на продакшене хорошо прожарится и наточит клюв.

Всего этого безобразия можно было бы избежать, если бы ошибки были бы частью языка, а паник как понятия не существовало:

```
func connect(addr string) *connection {
    con := db.Connect(addr)
    onError con.Close()






    prepare(con)

    return con
}
```

Не положил в переменную — эквивалентно старой доброй мантре. У всех функций неявное последнее возвращаемое значение — ошибка. `onError == defer` с проверкой на `err != nil`. Остается единственный вопрос со стектрейсом, но собственно и все.


Но нет. Мы будем страдать. Спасибо google.

Ответить

 **rule** 06.10.17 в 10:55    

Вот вы так много времени тратите чтоб убедить меня в удобности вашего решения. Мне кажется круто было бы реализовать его сделать PR. Это же опенсорс проект, так это работает.






Ответить

 **aspcartman** 06.10.17 в 11:04    

Вы ставите общительность в претензию каждому своему собеседнику?

К сожалению, в go это не работает. И не очень ясно хорошо это, или нет. PR'ы и просто предложения разворачиваются на мес потому что они противоречат религиозным убеждениям core team. Как противоположность — swift. Я не люблю swift, но там, в отличии от go, имеет смысл вкладывать силы и время. Их changelog каждый год впечатляет, и все это делает комюнити. Это к

Ответить

 **rule** 06.10.17 в 10:57    

И еще как временное решение использовать препроцессор.

Ответить

 **Bonart** 06.10.17 в 11:00    

Спасибо авторам языка — они настолько суровые практики, что проигнорировали все достижения теории примерно с середины годов.

Ответить






 **EvilFox** 06.10.17 в 19:35    

А я просто завернул в

```
func e(error error) bool {
    if error != nil {
        return true
    }
    return false
}
//...
pool, error = pgx.NewConnPool(connPoolConfig)
if e(error) {
    log.Crit("Unable to create connection pool", "error", error)
    for e(error) {
        log.Debug("Try to create connection pool", "warning", error)
        pool, error = pgx.NewConnPool(connPoolConfig)
    }
}
```

И в моём мире всё спокойно (очень уж вымораживает писать «error != nil»). Есть ещё парочка похожих обёрток, более специализированных (для логирования и вывода спец json-ответа об ошибке). Мне норм.

Ответить

 **f0rk** 09.10.17 в 12:12    

error != nil вымораживает писать, а if... вместо return error != nil нет. Странно..

Ответить

 **arvitaly** 06.10.17 в 06:22  

Ну, как бы, предполагается, что ошибка ошибке рознь и не нужно их обобщать, вот и все... Если же вам плевать на обработку ошибок, кидайте panic пользователи вообще любят стектрейсы и синие экраны смерти...

Ответить

 **aspcartman** 06.10.17 в 10:22    

На обработку ошибок «плевать» в более чем 90% случаев. Случаи обработки конкретных ошибок крайне редки. Как часто вы при попытке вставить либо в бд проверяете на конкретную ошибку о существовании записи и делаете в таком случае что-то другое? Обычно вы просто чистите за собою куданибудь вываливаете это в лог и все.

Выпад про синие экраны смерти тут некорректен. Язык действительно тебя заставляет либо как идиоту писать после каждой 1 строчки мантру из строк, либо использовать паники. Но в последнем случае тебя Go-сообщество справедливо закидает заранее подготовленным и свежее-подогретым говном. Вот сидим, пишем эту хреноту, да себя и всех вокруг убеждаем, что это хорошо.

Ответить

 **arvitaly** 10.10.17 в 19:15    





Стандартная библиотека Go создана для написания достаточно низкоуровневых программ, работающих с диском, сетью и т.д. Если ваша прога более высокоуровневая, то, очевидно, вам нужен более высокоуровневый DSL. Значит нужен слой, который скроет работу, например, с сетью слой будет выдавать только те ошибки, которые участвуют в бизнес-логике, т.е. должны быть ею обработаны. В свою очередь, ему важны ошибки работы с сетью, как минимум, даже если речь исключительно о логировании — это переписывание ошибок в зависимости от контекста в терм вашего фреймворка, или, например, возможность переподключения при разрыве соединения, или повторный запрос при конкретной ошибке и Все таки, лучше не писать хреноту, а понять почему так сделано и почему язык популярен.

Еще один тезис — синтаксис нужно оценивать, исходя из качества написанных библиотек, а не скорости написания кода. Качество библиотек достаточно высоко, значит синтаксис достаточно хорош, осталось понять почему, а ответ на этот вопрос дан уже много раз, осталось поверить это не бредни полоумных работяг из гугла.

Еще один тезис — в конце концов синтаксис вообще мало что значит, решают возможности языка, компилятор и инструменты.

Для меня единственный спорный вопрос в Go — дженерики, но этот вопрос не решен нигде. Некоторые языки предоставляют возможности метапрограммирования, потом декораторы, потом контракты и где-то на пятом уровне абстракции у нас возникает желание заняться машинным обучением, чтобы с этим кодом на Java/C# разобрался наш виртуальный друг.

Ответить

 **Sevlyar**  06.10.17 в 06:45  

```
func upgradeUser(endpoint, username string) error {
    getEndpoint := fmt.Sprintf("%s/oldusers/%s", endpoint, username)
    postEndpoint := fmt.Sprintf("%s/newusers/%s", endpoint, username)

    req, err := http.Get(genEndpoint)
    if err != nil {
        return err
    }
    data, err := ioutil.ReadAll(req.Body)
    if err != nil {
        return err
    }
```

```
    }
    olduser, err := user.NewFromJson(data)
    if err != nil {
        return err
    }
    newuser, err := user.NewUserFromUser(olduser),
    if err != nil {
        return err
    }
    buf, err := json.Marshal(newuser)
    if err != nil {
        return err
    }
    _, err = http.Post(
        postEndpoint,
        "application/json",
        bytes.NewBuffer(buf),
    )
    return err
}
```

Функция `upgradeUser` реализует какой-то сценарий бизнес логики, соответственно она должна абстрагировать от деталей реализации. Но почему-то функция не абстрагирует от низкоуровневых ошибок, которые возвращают используемые в ней функции. Например: мы хотим проапгрейдить юзера результате получаем ошибку ввода-вывода сети или `http.ProtocolError` или `url.Error`. Как я должен различать и обрабатывать их? Их даже залогировать корректно нельзя, т.к. непонятен контекст. Часть ошибок может быть по моей вине, если я указал некорректный `endpoint`, с остальными ошибками, я скорее всего ничего сделать не могу. Также не забываем, что код, который будет использовать функцию `upgradeUser`, по хорошему, ничего не должно знать о деталях реализации. И что делать, если в процессе поддержки приложения мы отказались от `http` и перешли на бинарный протокол? Переписывать все места, где используется `upgradeUser` чтобы корректно обработать ошибки?

Делаем выводы:



- 1) Реализация этой функции далека от совершенства, т.к. в ней текут абстракции — не пишите так;
- 2) Если вы собираетесь вернуть полученную ошибку «наверх» без изменения (заворачивания в другую) трижды подумайте;
- 3) Следует четко понимать и определять какие ошибки может возвращать ваша функция, детали реализации функций скрывайте, в части обработки ошибок: заворачивайте их в более высокоуровневые либо логируйте и возвращайте другую;
- 4) При правильной работе с ошибками разницы нет какой механизм вы используете, если писать корректный код на `js` с теми же `async/await` то рука болеть к вечеру от написания тонны `try {} catch()`, особенно если учитывать что там смешан поток обработки ожидаемый ошибочных и неожиданных исключительных ситуаций, и всегда надо будет прокидывать ошибку наверх вручную, если мы не знаем как ее обрабатывать.

Ответить

 **bash77** 06.10.17 в 07:56  

ну вот знаю я хаскель. но мне реально не по себе от вашего кода. я использую Go только для решения практических задач и не более.

Ответить

 **khrundel** 06.10.17 в 07:56  

Вроде Go 2.0 готовят, никто не в курсе, собираются с обработкой ошибок что-то делать или всё?

Ответить

## Написать комментарий

**B** / **I** **U**         \*

Предпросмотр

Отправить

☐ Markdown

### САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

Отсутствие дискриминации – это основная ценность open source

↑ +67    👁 16,4k    📌 28    💬 120

Почему мне не перезвонили?

↑ +24    👁 23,3k    📌 68    💬 68

DEFCON 17. Взлом 400 000 паролей, или как объяснить соседу по комнате, почему счёт за электричество увеличился. Часть 1

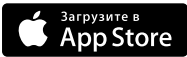

↑ +16    👁 6,3k    📌 43    💬 3

Как Microsoft спрятала целый сервер и как его найти

↑ +3    👁 6,1k    📌 14    💬 5

В чём важность  $196\,884 = 196\,883 + 1$ ? Как это объяснить на пальцах?

↑ +52    👁 25,7k    📌 119    💬 101

claygod	Разделы	Информация	Услуги	Приложения
Профиль	Публикации	Правила	Реклама	<div> </div>
Трекер	Хабы	Помощь	Тарифы	
Диалоги	Компании	Документация	Контент	
Настройки	Пользователи	Соглашение	Семинары	
ППА	Песочница	Конфиденциальность		