Врапинг ошибок до Go 1.13

У Джона Боднера (Jon Bodner), автора книги Learning Go, как-то спросили, какая особенность языка Go недооценена разработчиками и заслуживает большего внимания?

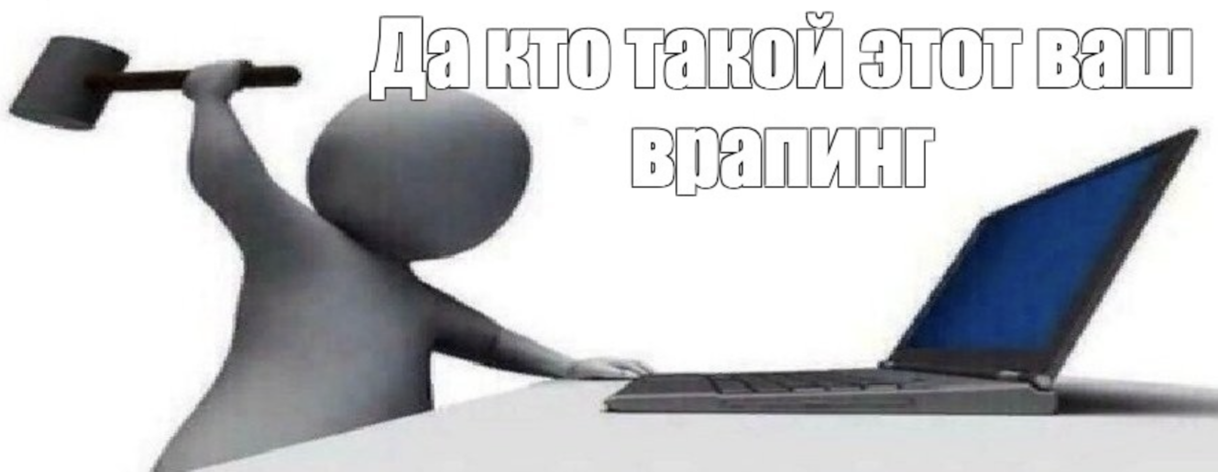
На что он ответил, что такими "обделёнными" фичами считает **врапинг ошибок** и работу с ошибками как со значениями, а не как с простыми строками:

```
What feature in Go do you think is underused by developers and
deserves more attention?

Wrapping errors and using errors as values and not just strings.
Including contextual information and using

types and instances of errors.
```

Об этом понятии и его эволюции мы, собственно, и поговорим в следующей паре уроков.



Понятие врапинга

Ранее мы выяснили, что с помощью пакета `errors` можно удобно создавать тривиальные ошибки вида

```
package bufio
```

```
// ...

var (
    ErrInvalidUnreadByte = errors.New("bufio: invalid use of UnreadByte")
    ErrInvalidUnreadRune = errors.New("bufio: invalid use of UnreadRune")
    ErrBufferFull        = errors.New("bufio: buffer full")
    ErrNegativeCount     = errors.New("bufio: negative count")
)


// ...
```

Но у них есть неприятный недостаток, который выражается в том, что при получении "сырой" ошибки (без какого-либо контекста), вы не можете точно сказать, в каком месте программы она произошла ([исходник примера]()):

```
// https://goplay.tools/snippet/kAXyw6u0j5P

var (
    ErrExecSQL         = errors.New("exec sql error")
    ErrInitTransaction = errors.New("init transaction error")
)

type Entity struct {
    ID string
}


func getEntity() (Entity, error) {
    // Может вернуть "сырую" ErrExecSQL.
    // ...
}

func updateEntity(e Entity) error {
    // Может вернуть "сырую" ErrExecSQL.
    // ...
}

func runInTransaction(f func() error) error {
    // Может вернуть "сырую" ErrInitTransaction.
    // ...
}
```

```go
func handler() error {
    var e Entity

    if err := runInTransaction(func() (opErr error) {
        e, opErr = getEntity()
        return opErr
    }); err != nil {
        return err
    }

    if err := runInTransaction(func() error {
        return updateEntity(e)
    }); err != nil {
        return err
    }

    if err := runInTransaction(func() (opErr error) {
        return updateEntity(e)
    }); err != nil {
        return err
    }

    return nil
}

func main() {
    for i := 0; i < 5; i++ {
        fmt.Println(handler())
    }
    /*
        От какой операции именно пришла ошибка?
            exec sql error
            init transaction error
            exec sql error
            exec sql error
            <nil>
    */
}
```

В `handler` мы в транзакциях вызываем различные операции и когда наверх из него приходит `exec sql error`, мы не знаем, это от `getEntity`? А может от первого `updateEntity`? А может от второго?

Аналогичная ситуация и с `init transaction error`: какую из всех возможных транзакций мы не смогли проинициализировать?

Здесь нам на помощь приходит `fmt.Errorf`, позволяющий организовывать подобие стека вызовов. Обычно это происходит через добавление к тексту существующей ошибки своего текста (иными словами врапинг/wrapping/оборачивание ошибки в новый текст):

```
if err := updateEntity(e); err != nil {
    return fmt.Errorf("cannot update entity %v: %v", e, err)
}

// ...

if err := handler(); err != nil {
    return fmt.Errorf("handler error: %v", err)
    // Аналогично
    // return fmt.Errorf("handler error: %s", err.Error())

}
```

Задача "Стек вызовов своими руками"

Ссылка на заготовку.

Вам необходимо переписать `handler` таким образом, чтобы его логика сохранилась, а все ошибки были завраплены (обёрнуты) в сообщения, которые легко помогут идентифицировать место их происхождения.

Врапинг не через текст, а через ошибку

Кроме оборачивания в текст мы можем обернуть ошибку в *ошибку собственного производства*, например:

```
type FileLoadError struct {
    URL string
    Err error // Для хранения "родительской" ошибки.
}

func (p *FileLoadError) Error() string {
```

```
    // Текст "родительской ошибки" фигурирует в тексте этой ошибки.
    return fmt.Sprintf("%q: %v", p.URL, p.Err)
}
```

Пример использования (исходник примера):

```
// https://goplay.tools/snippet/jEh7wG897xI

func getFile(u string) (File, error) {
    return File{}, context.Canceled
}

func loadFiles(urls ...string) ([]File, error) {
    files := make([]File, len(urls))
    for i, url := range urls {
        f, err := getFile(url)
        if err != nil {
            return nil, &FileLoadError{url, err} // <- Врапим ошибку
загрузки в *FileLoadError.
        }
        files[i] = f
    }
    return files, nil
}

func transfer() error {
    files, err := loadFiles("www.golang-courses.ru")
    if err != nil {
        return fmt.Errorf("cannot load files: %v", err)
    }

    // ...
    return nil
}

func handle() error {
    if err := transfer(); err != nil {
        return fmt.Errorf("cannot transfer files: %v", err)
    }

    // ...
    return nil
}
```

```
func main() {
    fmt.Println(handle())
}
```

В выводе мы получим:

```
cannot transfer files: cannot load files: "www.golang-courses.ru":
context canceled
```

своеобразный микс оборачивания ошибок и в текст и в другие ошибки.

Приём выше является организацией цепочек ошибок – **golang error chaining**.

Тест "Потеря потерь"

Что выведет программа из предыдущего шага, если заменить `main` на код ниже?

```
func main() {
    if err := transfer(); err != nil {
        if _, ok := err.(*FileLoadError); ok {
            fmt.Println("file load err received")
        } else {
            fmt.Println("unexpected error")
        }
    }

}
```

Почему так? – узнаем в следующем шаге.

Потеря типа ошибки

Действительно, построенные нами цепочки (***error chains***) "скрывают" тип ошибки, что не позволяет нам пользоваться известными ранее практиками:

```
func transfer() error {
    files, err := loadFiles(urls...)
    if err != nil {
```

```
        if fileLoadErr, ok := err.(*FileLoadError); ok { // Здесь мы ещё можем проверить тип ошибки.
            switch fileLoadErr.Err.(type) { // И даже поработать с нижележащей ошибкой.
            case *net.AddrError:
                // ...
            case net.UnknownNetworkError:
                // ...
            }
        }
        return fmt.Errorf("cannot load files: %v", err) // <- По сути превращаем "полноценную" ошибку в текст.
    }
    return nil
}

func main() {
    if err := transfer(); err != nil {
        if _, ok := err.(*FileLoadError); ok { // Здесь мы не можем писать подобное.
            // Ветка никогда не случится, так как после fmt.Errorf в transfer
            // мы получим *errors.errorString вместо *FileLoadError.
        }
        return fmt.Errorf("cannot tranfer files: %v", err)
    }

}
```

Аналогичная ситуация для **sentinel errors** и **opaque errors** – потеря актуального типа ошибки не даёт нам сравнивать ошибки напрямую или приводить полученную ошибку к известному интерфейсу.

Тест "Соотнесите ошибки и их тип"

Сопоставьте значения из двух списков

fmt.Errorf("cannot save HTML page: %v", fmt.Errorf("cannot fetch URL: %v", context.Canceled)),

net.UnknownNetworkError("usp")

syscall.EDOM

&fs.PathError{Op: "seek", Path: "/etc/hosts", Err: fs.ErrInvalid}

io.EOF

- - - - - - - - - - - - - - - - - - - -

net.UnknownNetworkError

syscall.Errno

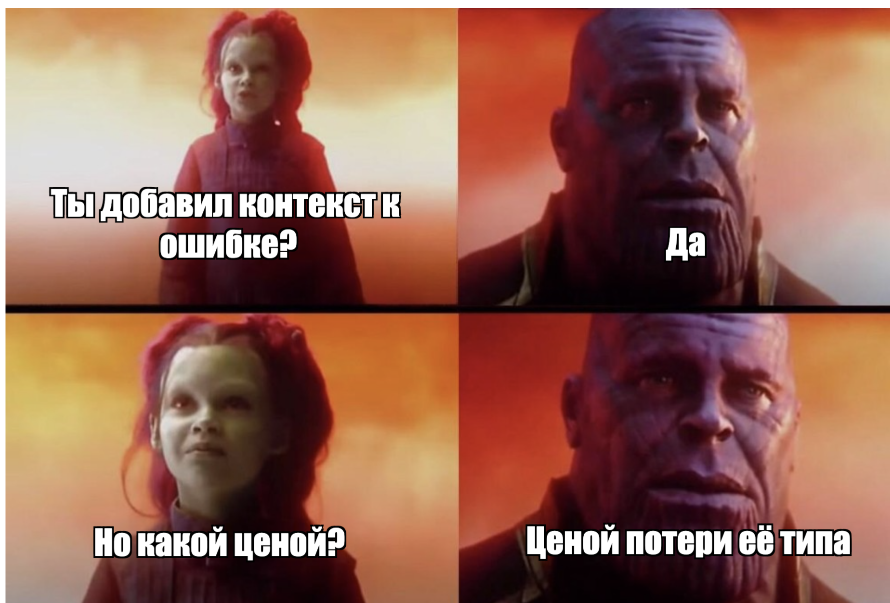
*fs.PathError

*errors.errorString (через errors)

*errors.errorString (через fmt)

Промежуточные итоги

Мы познакомились с понятием врапинга ошибок в Go и поняли, что можно добавлять дополнительный контекст к ошибке ценой потери её актуального типа.



Такая неприятная ситуация была в Go до версии **1.13**.

В следующем уроке мы посмотрим, какие кардинальные изменения произошли в работе с ошибками, и что нужно сделать, чтобы "и рыбку съесть и на лошадке покататься".