Базовые идиомы по обработке ошибок (часть 1)

В этом уроке собраны лучшие капитанские практики по работе с ошибками, с частью которых мы уже сталкивались ранее. Информацию следует воспринимать как краткую выжимку.

In-band ошибки

In-band error indication – это подход, о котором мы говорили в самом начале курса, знакомясь с ошибками в Си.

Выражается он в передаче и результата и ошибки через единственное возвращаемое значение.

Например:

```
// In-band ошибка, прячущаяся в возвращаемом типе.
func countSomething() int {
    if success {
        return 200 // Валидное значение, ошибки нет.
    }

    return -1 // Случилась какая-то ошибка.
}
```

```
// Out-of-band ошибка, возвращаемая в паре с результатом работы функции.
func countSomething() (int, error) {
    if success {
        return 200, nil // Валидное значение, ошибки нет.
    }

    return 0, errors.New("something went wrong") // Случилась какая-то ошибка.
}
```

Так вот, идиоматически [верный вариант](#) в Go – использовать **out-of-band error indication**, потому что Go умеет в множественные возвращаемые значения, использование которых является более явным, чистым и читаемым решением.

Бывает такое, что вместе с ошибкой всё-таки необходимо передать какие-то данные наверх. Наиболее приятным решением будет создать тип-обёртку над ошибкой, добавляющий ей необходимый контекст.

Например, есть функция разбора [JWT-токена](#):

```
func ParseToken(jwt, secret []byte) (Token, error)
```

И вместе с ошибкой из неё хочется получить email пользователя, зашитого в токен (если его удалось вычленить до ошибки).

Заведём для этого дела отдельный тип:

```
type ParseTokenError struct {
    email string
    err   error
}

func newParseTokenError(email string, err error) *ParseTokenError {
    return &ParseTokenError{email: email, err: err}
}

func (p *ParseTokenError) Error() string {
    return fmt.Sprintf("token from %q: %v", email, p.err)
```

```go
}

func (p *ParseTokenError) Unwrap() error {
    return p.err
}

func (p *ParseTokenError) Email() string {
    return p.email

}
```

И воспользуемся им:

```go
func ParseToken(jwt, secret []byte) (Token, error) {
    // ...
    if err != nil {
        return Token{}, newParseTokenError(email, err)
    }
    // ...
}
```

Теперь мы без труда можем пользоваться, как ошибкой, так и связанным с ней адресом эл. почты:

```go
token, err := ParseToken(jwt, secret)
if err != nil {
    var e interface {
        Email() string
    }
    if errors.As(err, &e) {
        // Используем e.Email()
    }
}
```

P.S. Обратим внимание на использование выше подхода **opaque errors**.

Тест "In-band ошибки в пакете strings"

Some standard library functions, like those in package "strings", return in-band error values.

This greatly simplifies string-manipulation code at the cost of requiring more diligence from the programmer.

In general, Go code should return additional values for errors.

Как мы видим, и в стандартной библиотеке существуют исключения из правил.

Вам необходимо ввести имя любой функции из пакета [strings](), которая вопреки лучшим практикам использует **in-band error indication** вместо явного возврата ошибки вторым аргументом.

Напишите текст

____

Не игнорируйте ошибки

В общем случае не существует веских причин, из-за которых можно игнорировать ошибки, потому что это может привести к трудноуловимым критическим багам:

```
func doSomething() (int, error) {
```

```
    return 0, errors.New("something really bad happened")
}

func main() {
    i, _ := doSomething() // Так делать не стоит.
    // ...

}
```

Пример неудачного игнорирования ошибки – уязвимость с переполнением буфера в Adobe Flash Player, позволяющая злоумышленникам удаленно выполнить код. Для этого жертве достаточно открыть специально подготовленный SWF-файл. А всё потому, что в коде не был корректно обработан результат, возвращаемый функцией `calloc`. Подробности здесь:

Once ignored, a value in an in-band error indicator can create havoc, including malicious code execution.

Как вы поняли, пример выше про Си, но концепция и последствия применимы к Go в том числе.

Но бывает, что всё-таки ошибки игнорируются – о подобных исключениях мы поговорим в отдельном уроке дальше в модуле.

Сначала проверяем ошибку

```
func doSomething() (int, error) {
    return 0, errors.New("something really bad happened")
}

func main() {
    i, err := doSomething()
    if err != nil { // Сначала обрабатываем ошибку.
        // ...
    }
    // Затем уже пользуемся результатом (в данном случае i).
```

```
}
```

Если функция или метод возвращает несколько значений, среди которых есть ошибка, то **сначала проверяем ошибку**.

Более того в таких случаях принято делать ошибку **последним возвращаемым аргументом**:

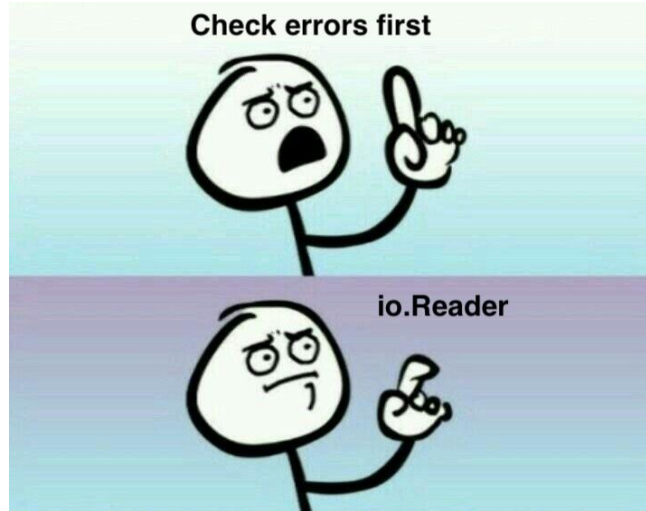
```
package strconv

func ParseFloat(s string, bitSize int) (float64, error) { // <- error
- последнее возвращаемое значение.
    f, n, err := parseFloatPrefix(s, bitSize)
    if err == nil && n != len(s) {
        return 0, syntaxError(fnParseFloat, s)
    }
    return f, err
}

func parseFloatPrefix(s string, bitSize int) (float64, int, error) {
// <- error - последнее возвращаемое значение.
    if bitSize == 32 {
        f, n, err := atof32(s)
        return float64(f), n, err
    }
    return atof64(s)
}
```

Проверять ошибку *когда-нибудь потом*, а сначала пользоваться другими возвращаемыми значениями, сродни игнорированию ошибки, что, как нам уже известно, не есть хорошо.

Однако из этого правила тоже существуют исключения, о которых мы узнаем чуть позже :)



Гарантия результата при отсутствии ошибки

Часто бывает, что функция возвращает два аргумента – указатель на тип и ошибку:

```
func doSomething() (*MyStruct, error) {
    return nil, errors.New("something really bad happened")

}
```

Первым делом мы проверяем, есть ли ошибка или нет. Далее иногда можно увидеть, что разработчик проверяет, что первый аргумент не является `nil`:

```
func doSomething() (*MyStruct, error) {
    return nil, errors.New("something really bad happened")
}

func main() {
    m, err := doSomething()
    if err != nil {
        return
    }
    if m == nil { // Так делать не надо - это избыточно.
        // ...
    }
    // ...

}
```

Так делать не надо, так как существует джентельменское соглашение – **в первом аргументе может быть `nil` только в том случае, если есть ошибка**. Т.е. функция должна гарантировать наличие результата при отсутствии ошибки и наоборот.

В целом это относится не только к указателям, но к любому другому типу возвращаемого значения (кроме слайсов), просто именно указатели часто приводят к панике `runtime error: invalid memory address or nil pointer dereference`.

P.S.

Джентельменские соглашения, как водится, штука ненадежная. Если есть подозрения, что джентельмен напротив (по совместительству автор функции) соглашению не следует, то проверить указатель на не `nil` все-таки стоит.

P.P.S.

Помните, что соглашение работает в обе стороны. Если вы автор функции, то старайтесь избегать неоднозначностей в том, как вашу функцию использовать.

Если нечего положить в результат и хочется вернуть `nil, nil`, то лучше, например, завести ошибку отсутствия данных:

```
var errNoData = errors.New("no data")

func doSomething() (*MyStruct, error) {
    // return nil, nil
    return nil, errNoData
}

func main() {
    m, err := doSomething()
    if err != nil {
        return
    }

}
```

Или использовать флажок для указания, валидно ли значение или нет:

```
func doSomething() (*MyStruct, bool, error) {
    if err := foo(); err != nil {
        return nil, false, err
```

```
    }

    if true {
        return nil, false, nil
    }

    return new(MyStruct), true, nil

}
```

Любой из вариантов, уменьшающий степень неоднозначности в коде, подойдёт.

Задача "ParseAndExecuteTemplate" (по мотивам реальной баги)

Ссылка на заготовку.

Младшему разработчику необходимо было написать функцию, которая парсит входной шаблон и выполняет его.

Он набросал следующий вариант:

```
import (
    "html/template"
    "io"
)

func ParseAndExecuteTemplate(wr io.Writer, name, text string, data
any) {
    t, _ := template.New(name).Parse(text)
    t.Execute(wr, data)
}
```

Но как оказалось, что иногда функция отрабатывает достаточно странно:

```
// https://goplay.tools/snippet/Osl-yWQ_9E1

func main() {
    // ...
    ParseAndExecuteTemplate(os.Stdout, "greeting", tmpl, data)
}
```

```
/*

<html>
     <body>


*/
```

А иногда вообще паникует:

```
// https://goplay.tools/snippet/9lTxPjM12NJ

func main() {
    // ...
    ParseAndExecuteTemplate(os.Stdout, "greeting", tmpl, data)
}

/*
panic: runtime error: invalid memory address or nil pointer
dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x20
pc=0x52b67b]

*/
```

Тимлид намекнул товарищу, что неплохо было бы проверить вызов функции на возникающие по пути ошибки:

```
func main() {
    // ...
    if err := ParseAndExecuteTemplate(os.Stdout, "greeting", tmpl,
data); err != nil {
        fmt.Println(err)
    }
}
```

Но для этого необходимо слегка отрефакторить `ParseAndExecuteTemplate`, что вам и предлагается сделать в рамках данной задачи :)

Errors are values

Ошибки – это значения.

Ошибку следует воспринимать как экземпляр типа со всеми вытекающими из этого последствиями.

Если возникает необходимость положить в ошибку дополнительную информацию, иллюстрирующую контекст возникновения, то без проблем заводим собственный тип под это дело:

```
type WithTimeError struct {
    err  error
    time time.Time
}

func (e *WithTimeError) Error() string {
    return fmt.Sprintf("%v, occurred at: %v", e.err.Error(),
e.time.Format(time.RFC3339Nano))
}

func (e *WithTimeError) Time() time.Time {
    return e.time
}

func (e *WithTimeError) Unwrap() error {
    return e.err

}
```

И желательно конструктор для него:

```
func NewWithTimeError(err error) error {
    return &WithTimeError{
        err:  err,
        time: time.Now(),
    }

}
```

И конструктор и ошибку можно сделать приватными, если их использование не ожидается за пределами нашего пакета.

Don't just check errors, handle them gracefully

Не просто проверяйте наличие ошибок, а осознанно их обрабатывайте.

В общем и целом, при возникновении ошибки не стоит сразу возвращать ошибку наверх, то есть писать что-то вроде такого:

```
if err != nil {
    return nil, err

}
```

Скорее всего, уже в этом месте мы можем как минимум:

- добавить контекста ошибке (через стектрейс, врапинг в текст или врапинг в дополнительный тип);
- залогировать ошибку (но тут есть нюансы, о которых мы поговорим позже);
- "изящно" обработать ошибку в соответствии с бизнес-логикой: например, совершить повтор операции и т.д.

Когда ничего из списка выше не происходит, мы начинаем испытывать боль, пример которой [привёл]() Дейв Чейни:

At the top of the program the main body of the program will print the error to the screen or a log file, and all that will be printed is:

`No such file or directory.`

There is no information of file and line where the error was generated. There is no stack trace of the call stack leading up to the error. The author of this code will be forced to a long session of bisecting their code to discover which code path trigged the file not found error.