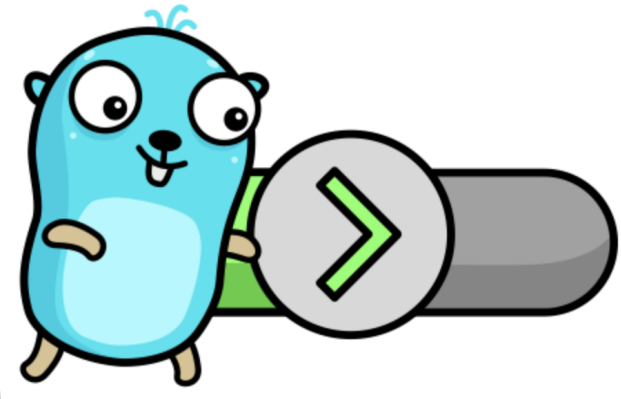
Базовые подходы к обработке ошибок в Go

В этом уроке мы обсудим заповеди Роба Пайка, касающиеся ошибок, а также познакомимся с базовыми подходами к работе с ошибками в Go.

Приведённые далее примеры актуальны для версий Go ниже 1.13. Но не переживайте – в следующих уроках мы подробно разберём, какие изменения произошли и как делать *правильно* относительно современного Go.

Главное понимать, что изменились лишь используемые инструменты, а сами **концепции изменений не потерпели**.



"Ошибочные" заповеди

Роб Пайк, один из главных разработчиков языка Go, сформулировал "заповеди" Go.

В общем, это ряд ёмких фраз-наставлений от Моисея из мира Go, которые говорят нам, гоферам, как следует жить.

Две заповеди касаются ошибок:

- Errors are values.

- Don't just check errors, handle them gracefully.

Вглядимся в них повнимательнее.

Errors are values

Ошибки – это значения.

Ошибку следует воспринимать как экземпляр типа (переменную) со всеми вытекающими последствиями.

Например, у ошибки могут быть какие-то методы и поля, которые мы можем использовать по своему усмотрению. Было бы неплохо выводить стектрейс до того места, где произошла ошибка? Заведем тогда отдельное поле, в котором будем хранить стектрейс (исходник примера):

```
// https://goplay.tools/snippet/0L-sAogVZGA

import (
    "fmt"
    "runtime/debug"
)

type WithStacktraceError struct {
    message    string
    stacktrace []byte
}

func (w *WithStacktraceError) Error() string {
    return w.message
}

func (w *WithStacktraceError) StackTrace() string {
    return string(w.stacktrace)
}

func doSomething() error {
    return &WithStacktraceError{
        message:    "something went wrong",
        stacktrace: debug.Stack(),
    }
}

func main() {
    if err := doSomething(); err != nil {
        if stacktraceErr, ok := err.(*WithStacktraceError); ok {
            fmt.Printf("%s\n%s", stacktraceErr.Error(),
stacktraceErr.StackTrace())
        }
    }

}
```

А зачем ошибки, если есть исключения?

Данный вопрос возникает у всех программистов, пришедших из языков программирования, в которых принята концепция исключений. Собственно, почему Go решил так выделиться со своими ошибками, а не сделал православные исключения?

Сами разработчики языка отмечают, что исключения приводят к запутанному коду, а также принуждают программистов рассматривать слишком много обычных ошибок как исключительные ситуации (например, ошибку открытия файла).

Говорить об исключениях принято в контексте обсуждения **понятия паники** в Go, что выходит за рамки данного курса.

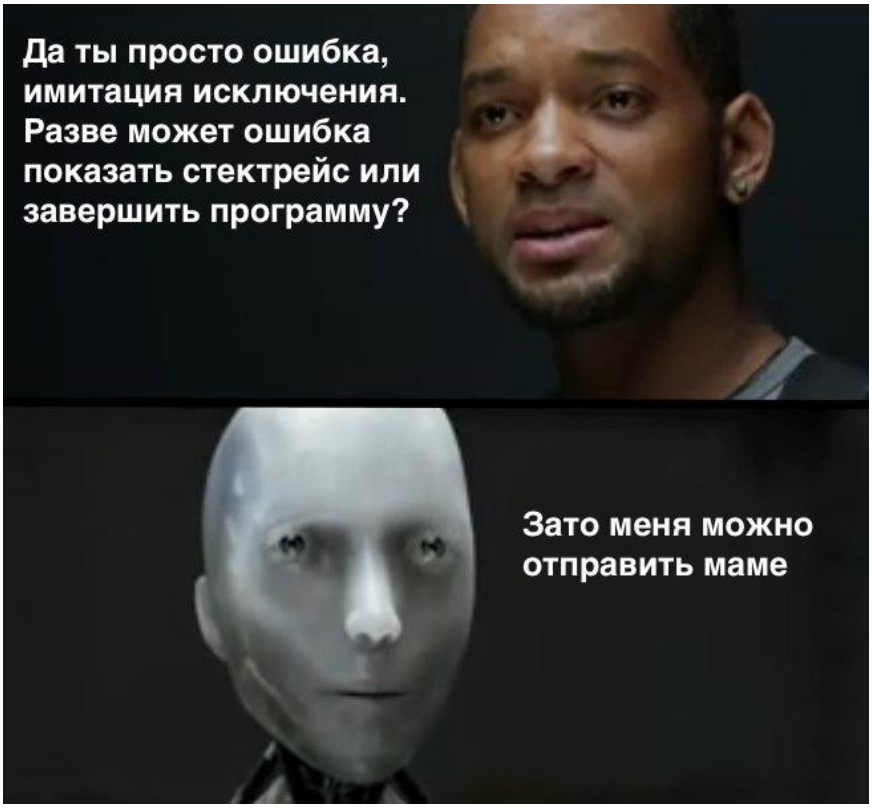
Так что сейчас просто выделим следующие тезисы:

1. Ошибка является экземпляром типа, то есть обычной переменной.

2. Ошибки в Go обрабатываются явным образом, так как они возвращаются при вызове функций в виде отдельных значений.

3. Ошибки, будучи обычными переменными, позволяют строить на их обработке разную логику. **Ошибку даже можно отправить своей маме (с) Роб Пайк**. Исключения же обрабатываются, как правило, конструкциями `try-catch-finally`, которые несколько уступают в гибкости работе с ошибками-значениями.

4. Ошибки – не для исключительных ситуаций. В какой-то степени аналогом исключений в Go можно назвать паники, которых в данном курсе мы не касаемся.



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

}
```

Всегда нужно помнить про то, что прежде всего нужно попытаться ошибку обработать или по возможности обогатить её б**о**льшим контекстом.

И тут подходим к интересному моменту: как понять, какая именно ошибка перед нами? Ведь на выходе мы имеем просто переменную с типом `error`. На самом деле пару шагов назад мы уже приводили ошибку к определённому типу, но сейчас узнаем, правильно ли так делать с идеологической точки зрения, и какие вообще у нас есть варианты.

Правило клуба обработки ошибок №0

Не пользоваться методом `Error()` для того, чтобы пытаться отличить одну ошибку от другой.

Кажется, почему бы и нет? Почему бы не сравнить результат `err.Error()` с какой-нибудь строковой константой?

Есть мнение Дейва Чейни (Dave Cheney) на этот счёт:

As an aside, I believe you should never inspect the output of the `error.Error` method.

The `Error` method on the `error` interface exists for humans, not code.

The contents of that string belong in a log file, or displayed on screen.

You shouldn't try to change the behaviour of your program by inspecting it.

Пользоваться методом `Error()` для того, чтобы пытаться отличить одну ошибку от другой – неправильно.

`Error()` – для кожаных человеков.

Как делать правильно?

Мы узнаем в следующих шагах.

Тест "Об ошибках в Go"

Выберите верные утверждения.

- [ ] Ошибки – это значения
- [ ] Ошибки не следует обрабатывать
- [ ] Ошибки в Go – это аналог исключений из других языков программирования
- [ ] Ошибки следует обрабатывать
- [ ] Ошибка не может нести в себе состояние
- [ ] Тип "error" в Go – это специальный type struct для ошибок
- [ ] Ошибки в Go возвращают не в исключительных ситуациях
- [ ] Ошибку можно отправить маме
- [ ] Использовать метод "Error" для сравнения ошибок в своём коде – нормальный вариант
- [ ] Ошибка является экземпляром типа, реализующего интерфейс error, то есть обычной переменной

Sentinel Errors

Использование **sentinel errors** (пытаться переводить игру слов не стали) – первый и самый простой вариант работы с ошибками.

Ошибки – это значения, поэтому ничего не мешает делать так:

```
import "io"

// ...
    if _, err := conn.Read(); err != nil {
        if err == io.EOF {
            // Что-то делаем.
        }
    }
// ...
```

То есть мы напрямую сравниваем ошибку (в примере выше `err`) с какой-то другой переменной. В нашем случае с *глобальной* переменной `EOF` из пакета `io`:

```
package io

// EOF is the error returned by Read when no more input is available.
var EOF = errors.New("EOF")
```

Подобная практика объявления на уровне пакета и возврата из функций уже "готовых" ошибок повсеместно используется в пакетах стандартной библиотеки (`net`, `os`, `http`, `sql` и пр.) и не только. Например, в модуле `github.com/jackc/pgx` есть ошибка ErrNoRows – близнец своей коллеги из стандартного пакета `sql`:

```
package pgx

// ErrNoRows occurs when rows are expected but none are returned.

var ErrNoRows = errors.New("no rows in result set")
```

(о `errors.New` мы поговорим чуть позже).

Подводя промежуточные итоги получаем:

- "готовые" ошибки - часть публичного API пакета/библиотеки;
- использование чужих "готовых" ошибок в коде неизбежно приводит к импорту пакета, где они были объявлены.

В публичности ошибок нет ничего противозаконного, но расширение API пакета напрямую связано с увеличением затрат на поддержку хотя бы обратной совместимости – взять и лихо удалить готовую ошибку из пакета, а уж тем более из стороннего модуля, не испортив себе карму, уже не получится.

Лишних импортов, разумеется, лучше избегать: код становится менее связным, пресекается вероятность появления циклических зависимостей и наша

программа чуть быстрее будет компилироваться (~~хотя куда уж быстрее, не плюсы всё таки~~).

Резюме

Sentinel errors – простейший и наиболее естественный подход к обработке ошибок. Раз ошибки – это значения, то берем и сравниваем их с готовыми значениями. Тем не менее, подход имеет свои недостатки, которые совсем не фатальные, просто о них нужно помнить и использовать такой вариант осознанно.

Несмотря на всё вышесказанное, наиболее вероятно, что чаще всего вы будете использовать и видеть примеры именно **sentinel errors** :)

Задача "Sentinel Errors"

Ссылка на заготовку.

Представим, что есть обработчик `Handler` неких задач `Job`. Обработчик принимает на вход в метод `Handle` задачу и обрабатывает её в методе `process`:

```
func (h *Handler) Handle(job Job) (postpone time.Duration, err error)
{
    err = h.process(job)
    if err != nil {
        // Обработайте ошибку.
    }

    return 0, nil

}
```

Во время обработки может возникнуть одна из известных нам ошибок:

```
var (
    ErrAlreadyDone      error = new(AlreadyDoneError)
    ErrInconsistentData error = new(InconsistentDataError)
    ErrInvalidID        error = new(InvalidIDError)
    ErrNotFound         error = new(NotFoundError)
    ErrNotReady         error = new(NotReadyError)

)
```

Кое-что мы про них знаем:

- некоторые ошибки настолько фатальные, что повторное выполнение задачи не имеет смысла, поэтому можем притвориться, что ничего не было и вернуть `nil` (это намёк на ошибки `ErrAlreadyDone`, `ErrInconsistentData`, `ErrInvalidID`, `ErrNotFound`);

- некоторые ошибки временные, и можно через некоторый промежуток времени сделать повтор (это намёк на `ErrNotReady`) – в таких случаях `Handle` возвращает время отсрочки `postpone`.

- в остальных случаях просто возвращаем ошибку.

Требуется обработать ошибку внутри метода `Handle`, используя подход **sentinel errors**.

Error Types

Второй вариант обработки ошибок – это когда мы отличаем их друг от друга по типу. Мы делали подобное ранее:

```
if err := doSomething(); err != nil {
    if stacktraceErr, ok := err.(*WithStacktraceError); ok {
        // ...
    }

}
```

Или даже так:

```
err := doSomething()
if err != nil {
    switch err.(type) {
    case *WithStackTraceError:
```

```
        // ...
    case *net.AddrError:
        // ...
    default:
        // ...
    }

}
```

По большому счёту сравнение через тип обладает теми же недостатками, что и сравнение с sentinel ошибками. Соответственно, когда возникает желание проверить тип ошибки, нужно иметь в виду, что:

- тип ошибки – это тоже часть публичного API пакета/модуля;
- проверка на тип ошибки приводит к импорту пакета, где тип был объявлен.

Но использование приведения к типу неизбежно, если мы хотим получить из ошибки больше контекста. Sentinel ошибки не могут содержать в себе дополнительной информации о происходящем, они лишь явно указывают на то, что факт ошибки произошёл, не объясняя детали.

В этом преимущество работы с ошибками через их тип, а не значение:

```
err := doSomething()
if err != nil {
    if err == io.EOF {
        // Нет стектрейса - нет контекста.
        // ...
    }

    switch e := err.(type) {
    case *WithStacktraceError:
        // Можем использовать e.StackTrace() - есть контекст.
    }

}
```

Резюме

Пользуйтесь подходом **error types**, если при обработке ошибки нужно вытаскивать из неё специфичную информацию.

Error types по своей сути схож с sentinel errors, только вместо сравнения с фиксированным значением мы приводим ошибку к фиксированному типу. Это ведёт к тем же недостаткам, что есть у sentinel errors.

Задача "Error Types"

Ссылка на заготовку.

Всё по новой.

Представим, что есть обработчик `Handler` неких задач `Job`. Обработчик принимает на вход в метод `Handle` задачу и обрабатывает её в методе `process`:

```
func (h *Handler) Handle(job Job) (postpone time.Duration, err error)
{
    err = h.process(job)
    if err != nil {
        // Обработайте ошибку.
    }

    return 0, nil

}
```

Во время обработки может возникнуть одна из известных нам ошибок:

```
type AlreadyDoneError struct{}
```

```
func (e *AlreadyDoneError) Error() string { return "job is already done" }

type InconsistentDataError struct{}
func (e *InconsistentDataError) Error() string { return "job payload is corrupted" }

type InvalidIDError struct{}
func (e *InvalidIDError) Error() string { return "invalid job id" }

type NotFoundError struct{}
func (e *NotFoundError) Error() string { return "job wasn't found" }

type NotReadyError struct{}

func (e *NotReadyError) Error() string { return "job is not ready to be performed" }
```

Кое-что мы про них знаем:

- некоторые ошибки настолько фатальные, что повторное выполнение задачи не имеет смысла, поэтому можем притвориться, что ничего не было и вернуть `nil` (это намёк на ошибки `AlreadyDoneError`, `InconsistentDataError`, `InvalidIDError`, `NotFoundError`);

- некоторые ошибки временные, и можно через некоторый промежуток времени сделать повтор (это намёк на `NotReadyError`) – в таких случаях `Handle` возвращает время отсрочки `postpone`.

- в остальных случаях просто возвращаем ошибку.

Требуется обработать ошибку внутри метода `Handle`, используя подход **error types**.

Opaque Errors

Третий вариант проверки ошибок наиболее навороченный. Однако, навороты устраняют недостатки первых двух вариантов.

Основная идея – определять, что это за ошибка и что с ней можно делать не по значению или типу, **а по поведению**.

Поведение типа в Go определяют интерфейсы, которые тип реализует.

Звучит сложно, поэтому разберёмся на примере.

Допустим, мы хотим сделать сетевой запрос. Сеть вещь ненадежная, возможны временные ошибки, которые можно полечить повтором:

```
func networkRequest() error {
    // Тут код, работающий с сетью.
}

func main() {
    if err := networkRequest(); err != nil {
        // В случае временной ошибки хотим сделать повтор.
    }

}
```

Чтобы сделать повтор, нужно понять, что случилась именно временная ошибка.

Проблема в том, что в `net` огромное количество ошибок и очень не хочется **type assert**'ить по каждой из них:

```
// https://github.com/golang/go master
$ grep -rE "type \w+Error" src/net
src/net/textproto/textproto.go:type ProtocolError string
src/net/mail/message.go:type charsetError string
src/net/net.go:type OpError struct {
src/net/net.go:type ParseError struct {
src/net/net.go:type AddrError struct {
src/net/net.go:type UnknownNetworkError string
src/net/net.go:type InvalidAddrError string
src/net/net.go:type timeoutError struct{}
src/net/net.go:type DNSConfigError struct {
```

```
src/net/net.go:type DNSError struct {
src/net/url/url.go:type EscapeError string
src/net/url/url.go:type InvalidHostError string
src/net/http/transport.go:type transportReadFromServerError struct {
src/net/http/transport.go:type nothingWrittenError struct {
src/net/http/transport.go:type responseAndError struct {
src/net/http/transport.go:type httpError struct {
src/net/http/transport.go:type tlsHandshakeTimeoutError struct{}
src/net/http/server.go:type statusError struct {
src/net/http/server.go:type checkConnErrorWriter struct {
src/net/http/h2_bundle.go:type http2ConnectionError http2ErrCode
src/net/http/h2_bundle.go:type http2StreamError struct {
src/net/http/h2_bundle.go:type http2goAwayFlowError struct{}
src/net/http/h2_bundle.go:type http2connError struct {
src/net/http/h2_bundle.go:type http2pseudoHeaderError string
src/net/http/h2_bundle.go:type http2duplicatePseudoHeaderError string
src/net/http/h2_bundle.go:type http2headerFieldNameError string
src/net/http/h2_bundle.go:type http2headerFieldValueError string
src/net/http/h2_bundle.go:type http2httpError struct {
src/net/http/h2_bundle.go:type http2noCachedConnError struct{}
src/net/http/h2_bundle.go:type http2resAndError struct {
src/net/http/h2_bundle.go:type http2GoAwayError struct {
src/net/http/request.go:type ProtocolError struct {
src/net/http/request.go:type requestBodyReadError struct{ error }
src/net/http/omithttp2.go:type http2noCachedConnError struct{}
src/net/http/transfer.go:type unsupportedTEError struct {

src/net/rpc/client.go:type ServerError string
```

(публичных ошибок, конечно, не так много, но всё равно не пара штук).

К счастью пакет `net` регламентирует, что ошибка временная, если она реализует метод `Temporary() bool`, возвращающий `true`:

```
// src/net/net.go
package net

// An Error represents a network error.
type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

Значит мы можем выделить свой интерфейс `temporary` и приводить ошибку к нему:

```
// IsTemporary говорит, является ли ошибка err временной.
func IsTemporary(err error) bool {
    type temporary interface {
        Temporary() bool
    }

    e, ok := err.(temporary) // Приводим ошибку к интерфейсу
temporary, тем самым проверяя поведение.
    return ok && e.Temporary()

}
```

или даже так

```
// IsTemporary говорит, является ли ошибка err временной.
func IsTemporary(err error) bool {
    e, ok := err.(interface{ Temporary() bool })
    return ok && e.Temporary()

}
```

Теперь мы без труда можем понять, является ли ошибка временной, и особым образом обработать её (исходник примера):

```
// https://goplay.tools/snippet/l6PaGe2X4mw

import (
    "fmt"
    "net"
)

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

func networkRequest() error {
    return &net.DNSError{ // У *DNSError есть метод Temporary(),
загляните внутрь.
        IsTimeout:   true,
        IsTemporary: true,
```

```
    }
}

func main() {
    if err := networkRequest(); err != nil {
        fmt.Println("error is temporary:", IsTemporary(err)) // error
is temporary: true
    }
}
```

Таким образом мы реализовали хелпер, проверяющий ошибку на определённое поведение. В данном случае мы узнаём, временная ошибка или нет, при этом не импортируя сторонние пакеты и не создавая зависимости от конкретных типов ошибок.

Или возвращаясь к нашей ошибке со стектрейсом (исходник примера).

Мы можем не завязываться на конкретную реализацию ошибки (это может быть как наш кастомный тип, так и ошибка стороннего модуля), главное, чтобы она умела возвращать стектрейс с помощью соответствующего метода:

```
// ...
    if err := doSomething(); err != nil {
        type stackTracer interface {
            StackTrace() string
        }
        if st, ok := err.(stackTracer); ok {
            fmt.Printf("%s\n%s", err, st.StackTrace())
        }
    }

// ...
```

Решение проблем первых двух вариантов работы с ошибками (sentinel errors и error types) создало новую проблему.

Наш интерфейс-индикатор поведения, очевидно должен иметь тот же метод, что и у ошибки, которая в общем случае объявляется в другом пакете. **Появляется неявная связь между местом, где объявлена ошибка, и местом, где она обрабатывается.** Что лучше, явный импорт или такая неявная зависимость?

К сожалению, однозначного ответа мы дать не можем - делитесь своими мыслями в комментариях.

В целом трудно представить, чтобы разработчики сторонних модулей начали использовать подход **opaque errors**, так как он требует синхронизации между потребителем ошибки и тем, кто её предоставляет. Но если вы разработчик такой библиотеки и отлично понимаете, кто и как её использует, то почему бы и нет. В обратном случае данный вариант кажется более подходящим для использования в рамках одного проекта.

Резюме

Красивый и элегантный вариант, не лишенный своих недостатков. Хорошо смотрится в границах одного проекта.

Задача "Opaque Errors"

Ссылка на заготовку.

И опять всё по новой.

Представим, что есть обработчик `Handler` неких задач `Job`. Обработчик принимает на вход в метод `Handle` задачу и обрабатывает её в методе `process`:

```
func (h *Handler) Handle(job Job) (postpone time.Duration, err error)
{
    err = h.process(job)
    if err != nil {
        // Обработайте ошибку.
    }

    return 0, nil

}
```

Во время обработки могут возникнуть ошибки, обладающие **одним из** следующих свойств:

- фатальные ошибоньки, которые проще пропустить и забыть о них — у таких есть метод `Skip() bool`, возвращающий `true`;

- временные ошибки, при которых через некоторый промежуток времени можно повторить обработку задачи – у таких есть метод `Temporary()`

`bool`, возвращающий `true`;

- незнакомые нам ошибки, которые стоит пробросить дальше наверх.

Требуется обработать ошибку внутри метода `Handle`, используя подход "**opaque errors**".

Для этого необходимо будет выделить нужные интерфейсы и реализовать соответствующие функции-хелперы.

Подведём итоги

Мы познакомились с тремя способами обработки ошибок:

1. **Sentinel Errors** – сравниваем полученную ошибку с ожидаемой ошибкой, представленной в виде готовой переменной (чаще всего объявленной на уровне пакета).

2. **Error Types** – через **type assertion** сравниваем тип полученной ошибки с ожидаемым типом (чаще всего далее используем данные из этого типа, "распакованного" из ошибки).

3. **Opaque Errors** – смотрим на "поведение" ошибки, какие методы она реализует, приводим ошибку к выделенному интерфейсу .

Важно понимать, что перечисленные методы не являются серебряными пулями, все они имеют свои преимущества и недостатки.

Разобранные в данном уроке подходы несколько устарели после появления Go 1.13.

В следующих уроках мы узнаем, как использовать стандартную библиотеку для быстрого создания ошибок, а также познакомимся с понятием **врапинга** и современными способами работы с ошибками.