OTUS

Golang Developer. Professional

otus.ru

Проверить, идет ли запись

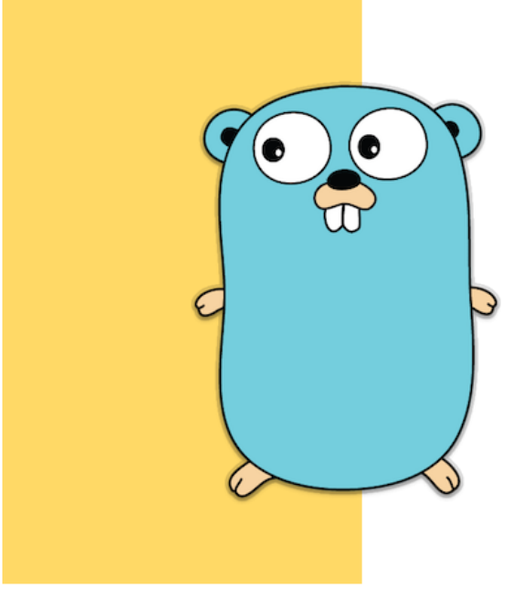
Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

Лучшие практики работы с ошибками

Алексей Романовский



Правила вебинара

Активно
участвуем

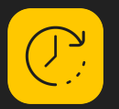
Off-topic обсуждаем
в учебной группе

Задаем вопрос
в чат или голосом

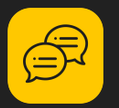
Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения

Индивидуально



Время, необходимое
на активность



Пишем в чат

Говорим голосом

Документ

Ответьте себе или
задайте вопрос

О чем будем говорить:

- Ошибки: принципы обработки, лучшие практики.
- panic, recover, defer

Ошибки

- Ошибка - тип, реализующий интерфейс `error`
- Функции возвращают ошибки как обычные значения
- По конвенции, ошибка - последнее возвращаемое функцией значение
- Ошибки обрабатываются проверкой значения (и/или передаются выше через `return`)

```
type error interface {
    Error() string
}
```

```
func Marshal(v interface{}) ([]byte, error) {
   e := &encodeState{}
   err := e.marshal(v, encOpts{escapeHTML: true})
   if err != nil {
      return nil, err
   }
   return e.Bytes(), nil
}
```

errors.go

Ошибки из стандартной библиотеки:

```
package errors

func New(text string) error {
   return &errorString{text}
}

type errorString struct {
   s string
}

func (e *errorString) Error() string {
   return e.s
}
```

errors.go

```
err := errors.New("Im an error")
if err != nil {
    fmt.Print(err)
}
```

```
whoami := "error"
err := fmt.Errorf("Im an %s", whoami)
if err != nil {
    fmt.Print(err)
}
```

Идиоматичная проверка ошибок

В целом ок:

```
func (router HttpRouter) parse(reader *bufio.Reader) (Request, error) {
  requestText, err := readCRLFLine(reader)
  if err != nil {
    return nil, err
  }

  requestLine, err := parseRequestLine(requestText)
  if err != nil {
    return nil, err
  }

  if request := router.routeRequest(requestLine); request != nil {
    return request, nil
  }

  return nil, requestLine.NotImplemented()
}
```

Ошибка - это значение

```
func (s *Scanner) Scan() (token []byte, error)
```

```
scanner := bufio.NewScanner(input)
for {
    token, err := scanner.Scan()
    if err != nil {
        return err // or maybe break
    }
    // process token
}
```

Мы можем сохранять её во внутренней структуре:

```
scanner := bufio.NewScanner(input)
for scanner.Scan() {
    token := scanner.Text()
    // process token
}
if err := scanner.Err(); err != nil {
    // process the error
}
```

Обработка ошибок: sentinel values

```
package io


// ErrShortWrite means that a write accepted fewer bytes
// than requested but failed to return an explicit error.
var ErrShortWrite = errors.New("short write")

// ErrShortBuffer means that a read required a longer
// buffer than was provided.
var ErrShortBuffer = errors.New("short buffer")
```

Ошибки в таком случае - часть публичного API, это наименее гибкая, но наиболее часто встречающаяся стратегия:

```
if err == io.EOF {
    ...
}
```

Проверка ошибок: типы

```go
// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error    // Returned by the system call.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

```
open /etc/passwx: no such file or directory
```

Проверка ошибок: типы

```
err := readConfig()
switch err := err.(type) {
    case nil:
        // call succeeded, nothing to do
    case *PathError:
        fmt.Println(“invalid config path:”, err.Path)
    default:
        // unknown error
}
```

Проверка ошибок: интерфейсы

```
package net

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

Проверяем поведение, а не тип:

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
    time.Sleep(1e9)
    continue
}
if err != nil {
    log.Fatal(err)
}
```

https://golang.org/pkg/net/#pkg-index

Больше о пользовательских типах ошибок

- https://gobyexample.com/custom-errors
- https://www.digitalocean.com/community/tutorials/creating-custom-errors-in-go

Антипаттерны проверки ошибок

```go
if err.Error() == "smth" { // Строковое представление - для людей.
```

```go
func Write(w io.Writer, buf []byte) {
    w.Write(buf) // Забыли проверить ошибку
}
```

```go
func Write(w io.Writer, buf []byte) error {
    _, err := w.Write(buf)
    if err != nil {
        // Логируем ошибку вероятно несколько раз
        // на разных уровнях абстракции.
        log.Println("unable to write:", err)
        return err
    }
    return nil
}
```

Оборачивание ошибок

Если ошибка не может быть обработана на текущем уровне, и мы хотим сообщить нё вызывающему с дополнительной информацией

```
func ReadAndCalcLen() error {
    from, to := readFromTo()
    a, err := Len(from, to)
    if err != nil {
        return fmt.Errorf("calc len for %i and %i: %w", from, to, err)
    }
}

//Результат: calc len for 2 and 1: from should be less than to
```

Соглашения об оборачивании ошибок: когда

- Необходимо обернуть, если в функции есть 2 или более мест, возвращающих ошибку.
- Можно вернуть исходную ошибку, если есть только 1 return.
- Перед добавлением второго return, рекомендуется рефакторинг первого return.

Соглашения об оборачивании ошибок: как

- Текст при оборачивании описывает место в текущей функции. Например, для функции openConfig(...):
  - Да: fmt.Errorf("open file: %w", err)
  - Нет: fmt.Errorf("startup: %w", err)

- не начинается с заглавной буквы
- не содержит знаков препинания в конце
- разделитель "слоёв" - ": "
- избегайте префиксов "fail to"/"error at"/"can not" в сообщениях обертки.
  - Можно для корневых ошибок и логирования:

```
log.Warn("fail to read: %v" , err)
```

Рекомендация

- Сделайте сообщение как можно более уникальным (для всего приложения)
- Параметры - в конец

Так будет проще находить код по тексту ошибки.

Как это выглядит в логах

```
fail to read form's data: get user:
open db connection: network error 0x123
```

github.com/pkg/errors

Использовалась до того как go научился оборачивать ошибки.
Сейчас - legacy проекты и для стектрейса в ошибке.

```
_, err := ioutil.ReadAll(r)
if err != nil {
        return errors.Wrap(err, "read failed")
}
```

```
package main

import "fmt"
import "github.com/pkg/errors"

func main() {
    err := errors.New("error")
    err = errors.Wrap(err, "open failed")
    err = errors.Wrap(err, "read config failed")

    fmt.Println(err) // read config failed: open failed: error
    fmt.Printf("%+v\n", err) // Напечатает stacktrace.

    print(err1 == errors.Cause(err2))  // true
}
```

Обработка обёрнутых ошибок

что, если обёрнуто?

```
err := doSomethingWithNetwork()
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
    time.Sleep(1e9)
    continue
}
if err != nil {
    log.Fatal(err)
}
```

`err.(net.Error)` - антипаттерн

errors.Is & errors.As

- https://pkg.go.dev/errors#Is
- https://pkg.go.dev/errors#As

errors.Is & errors.As

```go
import "errors"

type MyError struct {
    Code    int
    Message string
}

func (e *MyError) Error() string { return e.Message }

func main() {
    baseErr := &MyError{0x08006, "db connection error"}
    err := fmt.Errorf("read user: %w", baseErr)

    // Check if the error is of type MyError
    if errors.Is(err, &MyError{}) {
        fmt.Println("Error is of type DbError")
    }
    // Try to extract the underlying MyError value
    var myErr *MyError
    if errors.As(err, &myErr) {
        fmt.Printf("Extracted DbError: %v\n", myErr.Code)
    }
}
```

Интерфейс Is

https://github.com/golang/go/blob/master/src/errors/wrap.go#L58
Учитываем, когда создаём свои типы ошибок

errors.Is & errors.As

- Is - если надо проверить соответствие ошибки шаблону (тип, значение)
- As - если надо ещё и привести ошибку к искомому типу

Итого:

- Проверяйте ошибки.
- Лишний раз не логируйте.
- Проверяйте поведение, а не тип.
- Ошибки - это значения.
- Оборачивайте правильно

Defer, Panic и Recover: defer

`defer` позволяет назначить выполнение вызова функции непосредственно перед выходом из вызывающей функции

```
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...)
        if err != nil {
            return "", err  // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```

Defer, Panic и Recover: defer

Аргументы отложенного вызова функции вычисляются тогда, когда вычисляется команда defer.

```
func a() {
    i := 0
    defer fmt.Println(i)
    i++
    return
}
```

```
0
```

Defer, Panic и Recover: defer

Отложенные вызовы функций выполняются в порядке LIFO: последний отложенный вызов будет вызван первым — после того, как объемлющая функция завершит выполнение.

```
func b() {
    for i := 0; i < 4; i++ {
        defer fmt.Print(i)
    }
}
```

```
3210
```

Defer, Panic и Recover: defer

Отложенные функции могут читать и устанавливать именованные возвращаемые значения объемлющей функции.

```
func c() (i int) {
    defer func() { i++ }()
    return 1
}
```

Эта функция вернет 2

Panic и Recover

`Panic` — это встроенная функция, которая останавливает обычный поток управления и начинает паниковать. Когда функция F вызывает panic, выполнение F останавливается, все отложенные вызовы в F выполняются нормально, затем F возвращает управление вызывающей функции. Для вызывающей функции вызов F ведёт себя как вызов panic. Процесс продолжается вверх по стеку, пока все функции в текущей го-процедуре не завершат выполнение, после чего аварийно останавливается программа. Паника может быть вызвана прямым вызовом panic, а также вследствие ошибок времени выполнения, таких как доступ вне границ массива.

`Recover` — это встроенная функция, которая восстанавливает контроль над паникующей го-процедурой. Recover полезна только внутри отложенного вызова функции. Во время нормального выполнения, recover возвращает nil и не имеет других эффектов. Если же текущая го-процедура паникует, то вызов recover возвращает значение, которое было передано panic и восстанавливает нормальное выполнение.

Panic and recover

Паниковать стоит только в случае, если ошибку обработать нельзя, например:

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

Panic and recover

"поймать" панику можно с помощью `recover`: вызов `recover` останавливает выполнение отложенных функций и возвращает аргумент, переданный `panic`

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}
```

Panic and recover

пример из encoding/json:

```
// jsonError is an error wrapper type for internal use only.
// Panics with errors are wrapped in jsonError so that
// the top-level recover can distinguish intentional panics
// from this package.
type jsonError struct{ error }

func (e *encodeState) marshal(v interface{}, opts encOpts) (err error) {
    defer func() {
        if r := recover(); r != nil {
            if je, ok := r.(jsonError); ok {
                err = je.error
            } else {
                panic(r)
            }
        }
    }()
    e.reflectValue(reflect.ValueOf(v), opts)
    return nil
}
```

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Заполните, пожалуйста, опрос о занятии по ссылке в чате

Следующий вебинар

16 августа

Тестирование в Go

Ссылка на вебинар будет в ЛК за 15 минут

Материалы к занятию в ЛК — можно изучать

Обязательный материал обозначен красной лентой

Спасибо за внимание!

Приходите на следующие вебинары

Алексей Романовский