Monadic-style работа с ошибками

В этом уроке мы поговорим о том, как концепции из функционального программирования помогают сгладить углы обработки ошибок в Go.

Постановка проблемы

Ещё когда мы говорили о Си, мы уже затрагивали основной недостаток работы с ошибками в Go – **большое количество однотипного кода, связанного с обработкой ошибок:**

Checking every function call for an error condition increases code by 30-40%. (слайд 27)

Кажется, что обработка ошибок в Go поджигает людей, малознакомых с языком почти так же сильно, как и отсутствие в нём дженериков.

Небольшой пример, чтобы понять насколько бывает мала "полезная нагрузка" в коде по отношению к обработке ошибок:

The Density of Error Handing vs. Business Logic

```go
func main() {
  config, err := getArgs()
  if err != nil {
    fmt.Println(err)
    fmt.Println(showHelp())
    os.Exit(1)
  }
  var (
    getEndpoint  = fmt.Sprintf("%s/oldusers/%s", config.endpoint, config.username)
    postEndpoint = fmt.Sprintf("%s/newusers/%s", config.endpoint, config.username)
  )
  response, err := http.Get(getEndpoint)
  if err != nil {
    fmt.Println("unable to fetch HTTP data", err)
    os.Exit(1)
  }
  if response.StatusCode != 200 {
    fmt.Println("server returned " + response.Status)
    os.Exit(1)
  }
  body, err := ioutil.ReadAll(response.Body)
  if err != nil {
    fmt.Println("failed to read body: ", err)
    os.Exit(1)
  }
  u, err := user.NewFromJSON(body)
  if err != nil {
    fmt.Println("failed to deserialize json: ", err)
    os.Exit(1)
  }
  newUser, err := user.NewUserFromUser(u)
  if err != nil {
    fmt.Println("failed to convert user to new format: ", err)
    os.Exit(1)
  }
  newUserJSON, err := json.Marshal(newUser)
  if err != nil {
    fmt.Println("failed to marshal new user: ", err)
    os.Exit(1)
  }
  buf := bytes.NewBuffer(newUserJSON)
  response, err = http.Post(postEndpoint, "application/json", buf)
  if err != nil {
    fmt.Println("failed to post message: ", err)
    os.Exit(1)
  }
  if response.StatusCode != 200 {
    fmt.Printf("failed to post message: server returned: " + response.Status)
    os.Exit(1)
  }
  fmt.Printf("Success")
}
```

25

(картинка из презентации Monadic Error Handling in Go от Rebecca Skinner)

В следующих шагах мы посмотрим, как можно обойти эту проблему с помощью **монад**.

Визуально уменьшаем количество проверок с помощью монад

Мы не эксперты в классическом функциональном программировании и математике, с ним связанной, поэтому взглянем на **монады** с практической точки зрения, ограничившись определением с Вики:

***Мона́да** — особый тип данных в функциональных языках программирования, для которого возможно задать императивную последовательность выполнения некоторых операций над хранимыми значениями.*

Таким образом мы получаем тип, хранящий какое-то значение и функции/методы для работы с ним:

```
// M представляет собой монаду.
type M struct {
    err error
    v   interface{}
}

// Bind применяет функцию f к значению M, возвращая новую монаду.
// Если M невалидна, то Bind эффекта не имеет.
func (m M) Bind(f func(v interface{}) M) M { /*...*/ }

// Unpack возвращает значение и ошибку, хранимые в монаде.
func (m M) Unpack() (interface{}, error) { /*...*/ }

// Unit конструирует M на основе значения v.
func Unit(v interface{}) M { /*...*/ }

// Err конструирует "невалидную" монаду M.

func Err(err error) M { /*...*/ }
```

Теперь можно заиспользовать монаду в функции, выполняющей какую-то работу:

```
// Было.

func doWork(r io.Reader) error {
    l, err := readLine(r)
    if err != nil {
        return fmt.Errorf("reading input: %v", err)
    }

    d, err := parseData(l)
    if err != nil {
        return fmt.Errorf("parsing input: %v", err)
```

```go
    }

    if err := validateData(d); err != nil {
        return fmt.Errorf("invalid data: %v", err)
    }

    // ...
    return nil

}

// Стало.

func doWork(r io.Reader) error {
    result, err := Unit(r).
        Bind(readLine).
        Bind(parseData).
        Bind(validateData).
        // ...
        Unpack()
    // ...
    return nil

}
```

На первый взгляд получилось действительно чище.

Для большего понимания реализуем недостающий код типа `M`, представленного выше.

Задача "Реализация монады"

[Ссылка на заготовку](#).

Вам необходимо реализовать монаду `M` из предыдущего шага, а также функции для работы с ней, чтобы код ниже работал корректно:

```go
type UserRegistrationRequest struct {
    Email    string `json:"email"`
    Password string `json:"password"`
}
```

```go
// ParseUserRequest вычитывает из входного ридера
UserRegistrationRequest и валидирует его в monadic style.
func ParseUserRequest(r io.Reader) (UserRegistrationRequest, error) {
    res, err := Unit(r).
        Bind(unmarshalRequest).
        Bind(validateEmail).
        Bind(validatePassword).
        Unpack()
    if err != nil {
        return UserRegistrationRequest{}, err
    }
    return res.(UserRegistrationRequest), nil

}
```

Подробности в заготовке.

Переходим от "классической" монады к Railway Oriented Programming

Наша монада не привязана к конкретному типу и это свойство тянет за собой пустые интерфейсы (`v interface{}`) и связанные с этим неудобства (вроде проверки ниже):

```go
func validateEmail(v interface{}) M {
    req, ok := v.(UserRegistrationRequest)
    if !ok {
        return Err(fmt.Errorf("%w: %T", errUnexpectedTypeInMonad, v))
    }
    // ...

}
```

Введём допущение, что нам не требуется подобное обобщение, и сделаем более конкретную монаду (исходник примера):

```go
type UserRegistrationRequest struct {
```

```
    Email    string `json:"email"`
    Password string `json:"password"`
}

type M = UserRegistrationRequestMonad

type UserRegistrationRequestMonad struct {
    req UserRegistrationRequest
    err error
}

func (u M) Unpack() (UserRegistrationRequest, error) {
    return u.req, u.err
}

func (u M) Bind(f func(req UserRegistrationRequest) M) M {
    if u.err != nil {
        return u
    }
    return f(u.req)
}

func unmarshalRequest(r io.Reader) func(req UserRegistrationRequest)
M {
    return func(req UserRegistrationRequest) M {
        if err := json.NewDecoder(r).Decode(&req); err != nil {
            return M{err: err}
        }
        return M{req: req}
    }
}

func validateEmail(req UserRegistrationRequest) M {
    if req.Email == "" {
        return M{err: errors.New("empty email")}
    }
    return M{req: req}
}

func validatePassword(req UserRegistrationRequest) M {
    if req.Password == "" {
        return M{err: errors.New("empty password")}
    }
    return M{req: req}
```

```
}
```

Это всё ещё позволяет писать компактный код, но какой ценой!

```
func main() {
    req, err := UserRegistrationRequestMonad{}.
Bind(unmarshalRequest(strings.NewReader(`{"email":"bob@gmail.com","pa
ssword":"bob"}`))).
        Bind(validateEmail).
        Bind(validatePassword).
        Unpack()

    fmt.Println(err)        // nil
    fmt.Printf("%#v", req) //
main.UserRegistrationRequest{Email:"bob@gmail.com", Password:"bob"}
}
```

Столько ухищрений ради элементарных действий, более того `unmarshalRequest` пришлось превратить в замыкание.

Сделаем ещё один шаг от общего к частному и

- откажемся от метода `Bind`;
- соединим монаду и хранимый в ней тип;
- функции, работающие с монадой, превратим в её методы (исходник примера).

```
type UserRegistrationRequest struct {
    err      error
    Email    string `json:"email"`
    Password string `json:"password"`
}

func (u *UserRegistrationRequest) Err() error {
    return u.err
```

```
}

func (u *UserRegistrationRequest) Unmarshal(r io.Reader) {
    if u.err != nil {
        return
    }
    u.err = json.NewDecoder(r).Decode(u)
}

func (u *UserRegistrationRequest) ValidateEmail() {
    if u.err != nil {
        return
    }

    if u.Email == "" {
        u.err = errors.New("empty email")
    }
}

func (u *UserRegistrationRequest) ValidatePassword() {
    if u.err != nil {
        return
    }

    if u.Password == "" {
        u.err = errors.New("empty password")
    }
}
```

Код получился проще и стал более читаемым, при этом мы не потеряли возможности работать с типом в функциональном стиле:

```
func main() {
    var req UserRegistrationRequest

req.Unmarshal(strings.NewReader(`{"email":"bob@gmail.com","password":
""}`))
    req.ValidateEmail()
    req.ValidatePassword()

    fmt.Println(req.Err()) // empty password
```

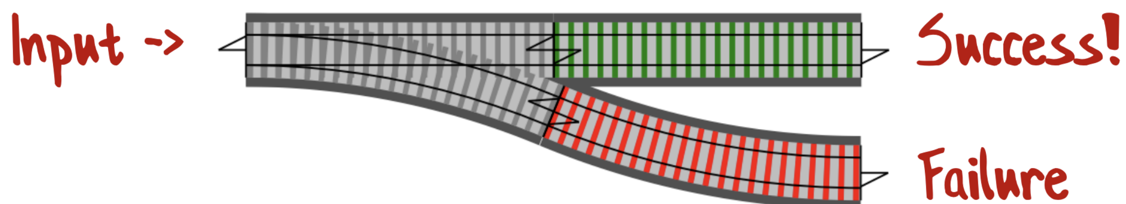
```
}
```

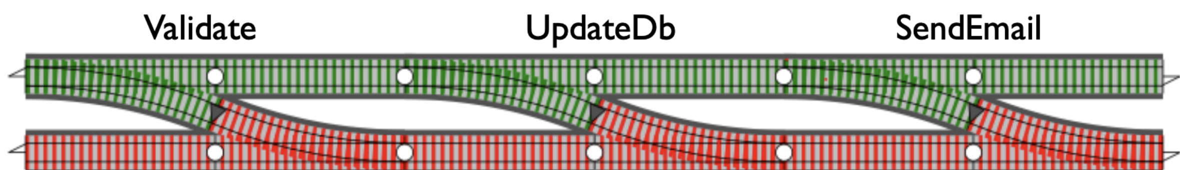
Railway Oriented Programming

Паттерн, к которому мы пришли на предыдущем шаге, можно встретить при разработке на Go (и не только) и заключается он в том, что:

- тип скрывает внутри себя ошибку, произошедшую в одном из его методов;
- метод типа не имеет эффекта, если ранее при работе с типом произошла ошибка.

Ничего не напоминает?





(картинки из презентации Railway oriented programming от Scott Wlaschin)

Это позволяет работать с типом в некотором функциональном стиле, игнорируя промежуточную обработку ошибок. Примером подобного в стандартной библиотеке является *bufio.Writer:

```
// https://goplay.tools/snippet/822MvhM4BDE

package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    b := bufio.NewWriter(os.Stdout)

    b.WriteString("Hello ")
    b.WriteString("World")
    b.WriteString("!")

    if err := b.Flush(); err != nil {
        fmt.Println(err)
    }

    // Hello World!
}
```

Методы писателя возвращают ошибки, например,

```
// Write writes the contents of p into the buffer.

func (b *Writer) Write(p []byte) (nn int, err error)
```

но сделано это для скорее для консистентности с интерфейсом io.Writer, на самом же деле работа метода обезопашена соответствующими проверками:

```
func (b *Writer) Write(p []byte) (nn int, err error) {
```

```
    for len(p) > b.Available() && b.err == nil {
        // ...
    }
    if b.err != nil {
        return nn, b.err
    }
    // ...

}
```

Аналогично тому, как мы делали ранее сами:

```
func (u *UserRegistrationRequest) ValidatePassword() {
    if u.err != nil {
        return
    }
    // ...

}
```

Метод `Err() error`

Мы заметили, что при *железнодорожном программировании* обычно наблюдается ряд вызовов промежуточных методов (без проверки ошибок), а затем уже вызов завершающего метода, возвращающего ошибку всего пайплайна:

```
doc := word.New("example.docx")

doc.AddHeader("Module 1")
doc.SetFontSize(14)
doc.AddParagraph("Working with errors in C")

if err := doc.Save(); err != nil {
    // ...

}
```

Если такого "финального" метода нет, то вместо него можно встретить метод `Err() error`, возвращающий ошибку, хранимую в типе:

```
// https://github.com/golang/go master
$ grep -R ") Err()" .
./api/go1.txt:pkg database/sql, method (*Rows) Err() error
./api/go1.txt:pkg go/scanner, method (ErrorList) Err() error
./api/go1.15.txt:pkg database/sql, method (*Row) Err() error
./api/go1.1.txt:pkg bufio, method (*Scanner) Err() error
./src/cmd/go/internal/web/api.go:func (r *Response) Err() error {
./src/cmd/vendor/golang.org/x/mod/zip/zip.go:func (cf CheckedFiles)
Err() error {
./src/database/sql/sql.go:func (rs *Rows) Err() error {
./src/database/sql/sql.go:func (r *Row) Err() error {
./src/bufio/scan.go:func (s *Scanner) Err() error {
./src/compress/bzip2/bit_reader.go:func (br *bitReader) Err() error {
./src/net/http/h2_bundle.go:func (p *http2pipe) Err() error {
./src/net/http/transport_test.go:func (d doneContext) Err() error {
return d.err }
./src/go/scanner/errors.go:func (p ErrorList) Err() error {
./src/context/context.go:func (*emptyCtx) Err() error {

./src/context/context.go:func (c *cancelCtx) Err() error {
```

Например,

```
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
   // ...
}

if err := scanner.Err(); err != nil {
    // ...

}
```

или

```
rows, err := db.QueryContext(ctx, "SELECT name FROM users WHERE
age=?", age)
if err != nil {
    // ...
}
```

```go
defer rows.Close()


for rows.Next() {
    // ...
}

if err := rows.Err(); err != nil {
   // ...

}
```

и т.д.

Рекомендуем вам следовать данной практике, если реализуете подобное.

Идемпотентность API

Ранее в данном модуле мы говорили о том, что здорово, если повторный вызов метода никак не влияет на состояние (не приводит к паникам, утечкам ресурсов и т.д.) объекта, уже ставшего невалидным до этого.

Например, можно без проблем несколько раз вызвать (*bufio.Writer).Flush:

```go
// https://goplay.tools/snippet/At-CsRvkq76

func main() {
    f, err := os.Open("/etc/hosts")
    if err != nil {
        panic(err)
    }
    if err := f.Close(); err != nil {
        panic(err)
    }

    w := bufio.NewWriter(f)
    w.WriteString("bad")
```

```
    for i := 0; i < 5; i++ {
        if err := w.Flush(); err != nil {
            fmt.Println(err)
        }
    }

    /*
        write /etc/hosts: file already closed
        write /etc/hosts: file already closed
        write /etc/hosts: file already closed
        write /etc/hosts: file already closed
        write /etc/hosts: file already closed
    */
}
```

Или пытаться повторно запустить HTTP-сервер после того, как он был остановлен ранее:

```
// https://goplay.tools/snippet/-15wuzVdql9

func main() {
    var s http.Server

    go func() {
        time.Sleep(time.Second * 3)

        if err := s.Close(); err != nil {
            panic(err)
        }

        for i := 0; i < 5; i++ {
            if err := s.ListenAndServe(); err != nil {
                fmt.Println(err)
            }
        }
    }()

    if err := s.ListenAndServe(); err != nil {
        fmt.Println(err)
    }

    /*
        http: Server closed
```

```
        http: Server closed
        http: Server closed
        http: Server closed
        http: Server closed
        http: Server closed
    */

}
```

и т.д.

Реализуется это через уже знакомые нам проверки:

```
package bufio

// Flush writes any buffered data to the underlying io.Writer.
func (b *Writer) Flush() error {
    if b.err != nil {
        return b.err
    }
    // ...

}

package http

// ListenAndServe listens on the TCP network address srv.Addr and
then
// calls Serve to handle requests on incoming connections.
func (srv *Server) ListenAndServe() error {
    if srv.shuttingDown() {
        return ErrServerClosed
    }
    // ...

}
```

Данные примеры не совсем относятся к теме урока, но мы посчитали нужным лишний раз напомнить о них.

Задача "Mini Word"

[Ссылка на заготовку](#).

Необходимо реализовать document builder, имеющий следующее API:

```
const maxPages = 3

var (
    errInvalidPageIndex = ...
    errNoMorePages      = ...
    errEmptyText        = ...
)

type IDocument interface {
    // AddPage добавляет новую страницу в документ. Документ может содержать максимум maxPages страниц.
    // По умолчанию после создания документ должен иметь одну страницу.
    // При превышении лимита метод выставляет errNoMorePages.
    // Если ранее при работе с документом произошла ошибка, то метод эффекта не имеет.
    AddPage()

    // SetActivePage выбирает страницу, с которой будет производиться дальнейшая работа.
    // По умолчанию после создания документа активной считается первая страница.
    // Нумерация начинается с единицы. При невалидном номере метод выставляет errInvalidPageNumber.
    // Если ранее при работе с документом произошла ошибка, то метод эффекта не имеет.
    SetActivePage(number int)

    // WriteText добавляет текст в активную страницу. При пустом тексте метод выставляет errEmptyText.
    // Если ранее при работе с документом произошла ошибка, то метод эффекта не имеет.
    WriteText(s string)

    // WriteTo записывает документ во w в следующем формате:
    // --- Page 1 ---
    // <текст страницы 1>
    // --- Page 2 ---
    // <текст страницы 2>
    //
```

```
    // и т.д.
    // Если ранее при работе с документом произошла ошибка, то метод эффекта не имеет.
    WriteTo(w io.Writer) (n int64, err error)
}
```

Пример создания документа:

```
func main() {
    d := NewDocument()
    d.WriteText("Hello")

    d.AddPage()
    d.SetActivePage(2)
    d.WriteText("World!")

    if _, err := d.WriteTo(os.Stdout); err != nil {
        panic(err)
    }

    /*
    --- Page 1 ---
    Hello
    --- Page 2 ---
    World!
    */
}
```

Больше подробностей в заготовке и тестах.

Мнение

В этом уроке мы рассмотрели приёмы для уменьшения количества кода, связанного с обработкой ошибок, начиная от монад и заканчивая чем-то, похожим на Railway Oriented Programming.

Смогли ли мы действительно уменьшить количество `if err != nil`?

Да, кажется, изначальная цель достигнута. Код стал более читаемым в силу компактности, однако, здесь мы потеряли в простоте. Появились какие-то типы, дополнительные методы, а обработка ошибок на самом деле не пропала, а просто переехала в другое место, став неявной.

В некоторых примерах данного урока прослеживается явное нарушение двух заповедей из разных языков:

- *Explicit is better than implicit.* (c) Python
- *Clear is better than clever.* (c) Golang

Используем ли мы подобные подходы в реальной жизни?

Только в частных случаях. Собственно, тяжело представить, чтобы какой-нибудь типичный сервис, перекладывающий JSON туда и обратно, подобным образом обрабатывал ошибки.

Если говорить про функциональное программирование, то в целом его удобно применять, имея на руках большое количество различных реализаций монад и пр. пазлов, позволяющих собирать полную картину, особо не напрягаясь. Здесь помогут сторонние модули (например, TeaEntityLab/fpGo), но, думаем, будет сложно уговорить себя и своих коллег пользоваться этим.

Если говорить про сокрытие ошибки в типе, то это производная другой заповеди Go – **Errors are values**. Не стоит забывать о ней и тогда на ум придут и более интересные идеи. Реальные применения данного подхода можно найти в archive/zip и net/http.