Лекция 2

Объектная модель в Go

План занятия

- Структуры
- Интерфейсы
- Обработка ошибок
- Конфигурирование приложения
- Сериализация и десериализация

Структуры

Структура (struct) — пользовательский тип данных, объединяющий набор полей произвольны типов.

Синтаксис

```
type ИмяСтруктуры struct {
    Поле1 Тип
    Поле2 Тип
}
```

Пример

```
type Person struct {  4 usages  new *
    Name     string
    Age      int
    City     string
    password string // приватное поле
}
```

Варианты инициализации структур

Объявить все поля:

Объявить часть полей:

Ничего не объявлять:
(значения по умолчанию)

Не используя имена:
(поля определятся по порядку)

```
c1 := Person{Name: "Vasja", Age: 12, City: "SPB", password: "123456"}
fmt.Println(c1)

c2 := Person{Name: "Vasja", Age: 12}
fmt.Println(c2)

c3 := Person{}
fmt.Println(c3)

c4 := Person{ Name: "Vasja",  Age: 12,  City: "SPB",  password: "123456"}
fmt.Println(c4)
```

```
{Vasja 12 SPB 123456}
{Vasja 12  }
{ 0  }
{Vasja 12 SPB 123456}
```

Конструктор

- Нужен для:
  - проверки входных данных
  - вычислений, необходимых перед инициализацией
  - инициализации приватных полей

- В Го конструкторы часто не используют

Конструктор

```go
func NewPerson(name string, age int, city string, password string) *Person {
    if age < 0 {
        println( args...: "age must be greater than zero")
    }

    return &Person{
        Name:     name,
        Age:      age,
        City:     city,
        password: password,
    }
}
```

```go
a := NewPerson( name: "Vasja",  age: 12,  city: "SPB",  password: "123456")
```

Пустая структура

```
var a struct{}

a = struct{}{}
```

Используются для того, чтобы обозначить наличие чего-либо. Пустая структура занимает **0 байт памяти.**

```
elements := make(map[string]struct{})
elements["element"] = struct{}{}

if _, exists := elements["element"]; exists {
    fmt.Println( a...: "Element exists in the elements")
}
```

Методы

Value receiver

(данные исходной структуры изменить нельзя)

```
func (p Person) PrintName() {  no usages  new *
    fmt.Println( a...: "Name", p.Name)
}
```

Pointer receiver

(можно изменить данные в исходной структуре)

```
func (p *Person) SetPassword(password string) {  1 usage
    p.password = password
}
```

Встраивание

В Go нет наследования, но есть встраивание (embedding).
Встраивание позволяет использовать поля и методы другой структуры.

```go
type PersonWithBirthday struct {  1 usage  new *
    Person
    Birthday time.Time
}
```

```go
person := PersonWithBirthday{
    Person:   Person{Name: "Vasja"},
    Birthday: time.Now(),
}

person.PrintName()
person.SetPassword( password: "123456")
```

Полиморфизм

Полиморфизм в Го реализуется при помощи интерфейсов

Интерфейс в Go — это тип, который определяет
поведение (методы), но не реализует их.

Утиная типизация(duck typing)

Любой тип, который реализует методы интерфейса, автоматически удовлетворяет этому интерфейсу.
При утиной типизации нет необходимости явно указывать, что тип реализует интерфейс – это определяется автоматически

Интерфейс

Тип DatabaseRepository
и тип CacheRepository
удовлетворяют интерфейсу PersonRepository

```go
type PersonRepository interface {  2 usages  3 implementations  new *
    SavePerson(p Person)  3 implementations
}

type DatabaseRepository struct{}  2 usages  new *

func (m DatabaseRepository) SavePerson(p Person) {  no usages  new *
    fmt.Println( a...: "person with name ", p.Name, " is saved to the database")
}

type CacheRepository struct{}  2 usages  new *

func (m CacheRepository) SavePerson(p Person) {  no usages  new *
    fmt.Println( a...: "person with name ", p.Name, " is saved to the database")
}
```

Композиция

Почти как встраивание, но только для интерфейсов

```
// ReadWriter is the interface that groups the basic Read and Write methods.
type ReadWriter interface {
    Reader
    Writer
}
```

Интерфейс ReadWriter будет содержать методы из Reader и из Writer

Самый известный интерфейс в Го

```
// The error built-in interface type is the conventional interface for
// representing an error condition, with the nil value representing no error
type error interface {
    Error() string
}
```

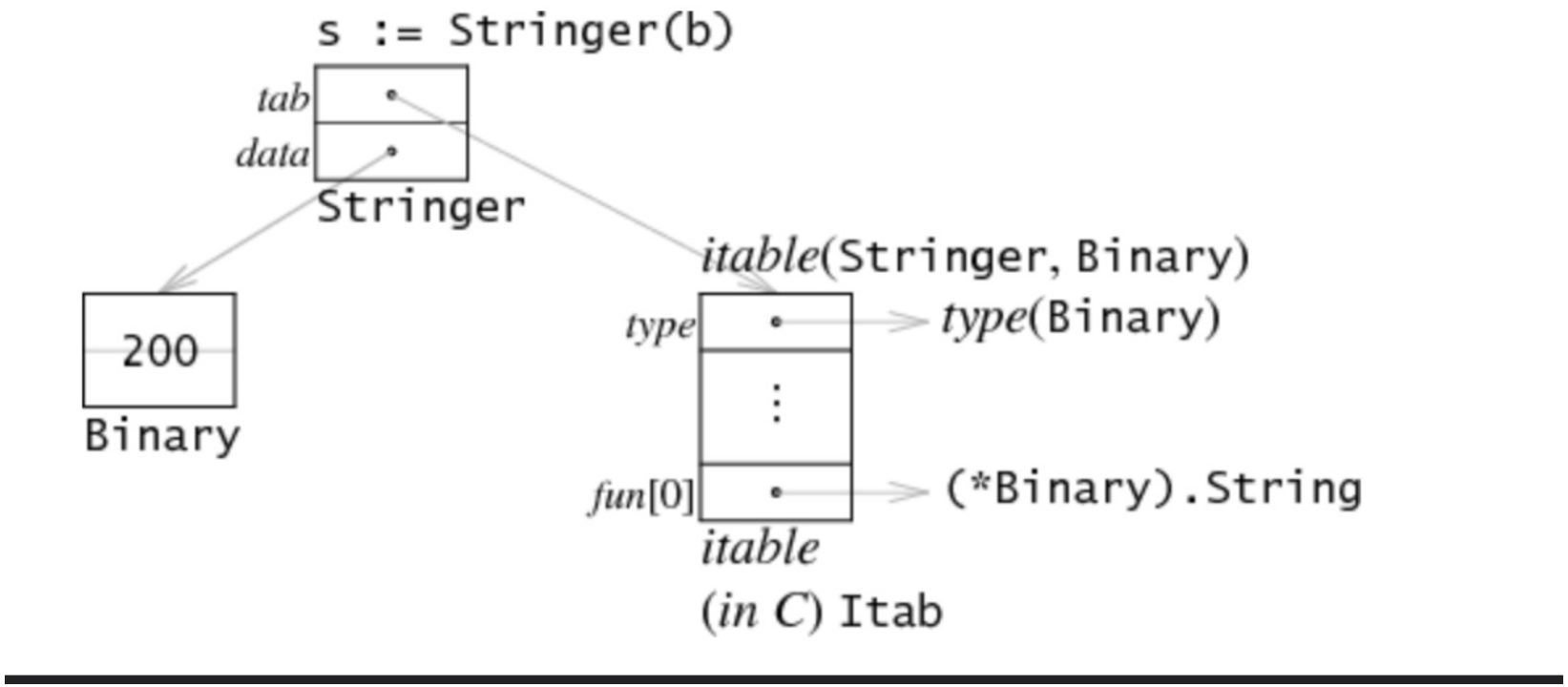
Пример:

```
func returningErrorFunc() error {  no usages  new *
    return nil
}
```

Интерфейс

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```

```
type itab struct {
    inter  *interfacetype
    _type  *_type
    link   *itab
    bad    int32
    unused int32
    fun    [1]uintptr // var
}
```

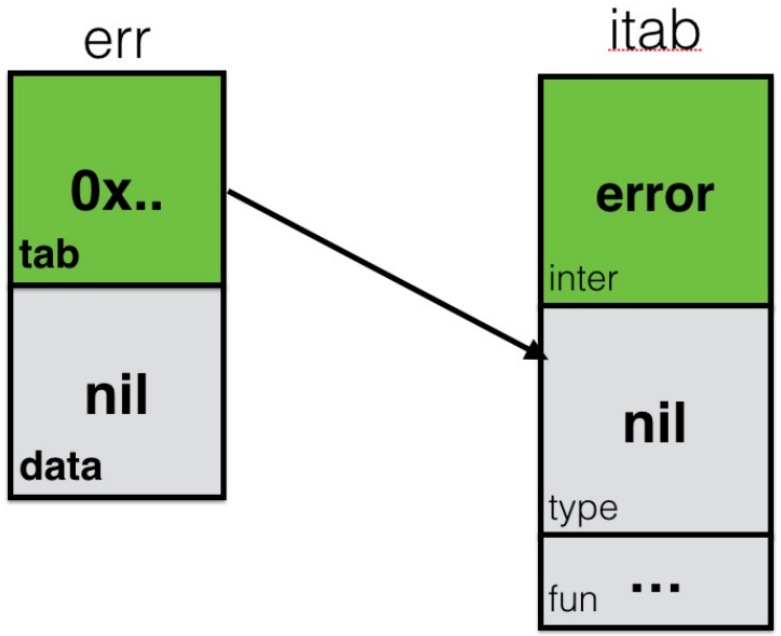


https://habr.com/ru/articles/276981/

https://habr.com/ru/articles/325468/

Интерфейс

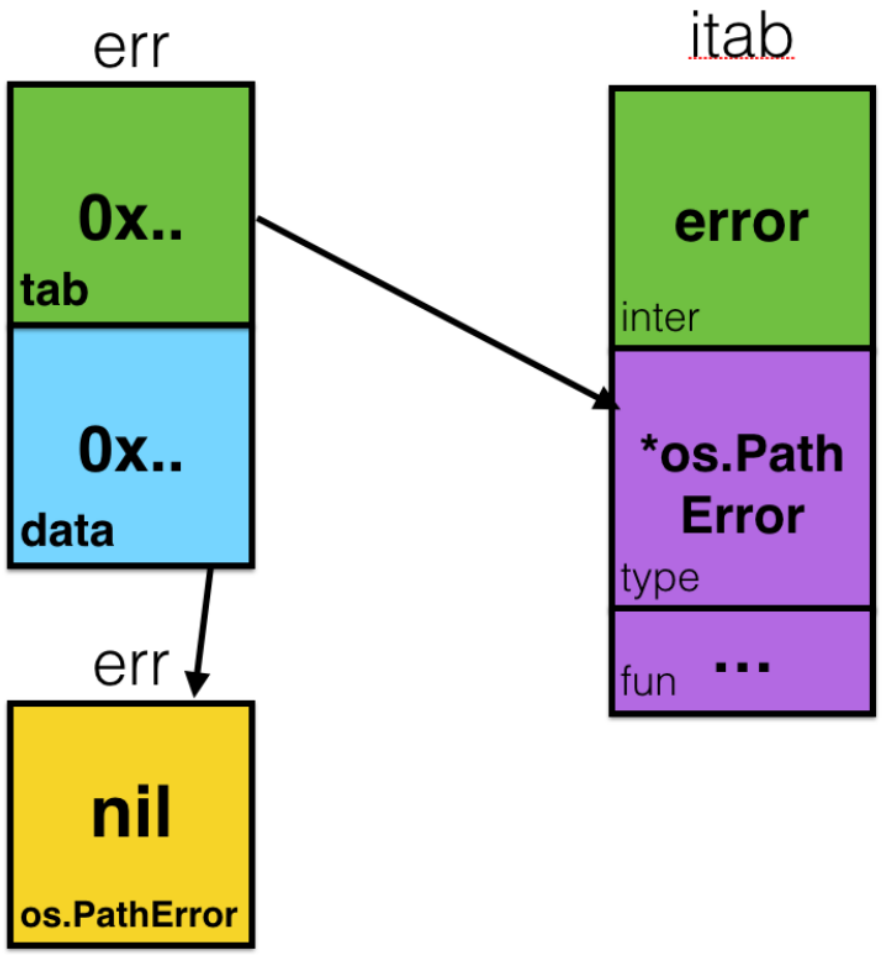
```
var err error
```




```
func foo() error {
    var err error // nil
    return err
}

err := foo()
if err == nil {...} // true
```

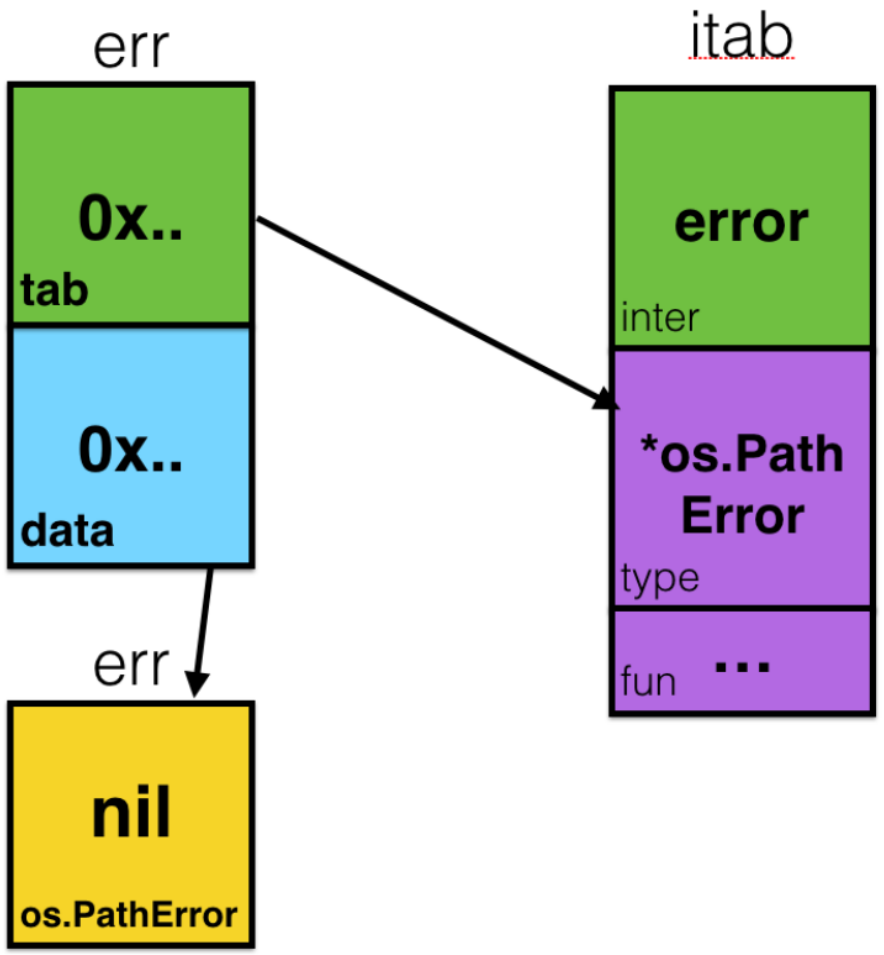
Интерфейс

```go
func foo() error {
    var err *os.PathError // nil
    return err
}

err := foo()
if err == nil {...} // false
```



Интерфейс

```go
func foo() error {
    var err *os.PathError // nil
    return err
}

err := foo()
if err == nil {...} // false
```



Пустой интерфейс

```
var x interface{}
```

В языке Go **пустой интерфейс** — это специальный тип интерфейса, который может хранить значения любого типа. Он часто используется, когда нужно работать с данными, тип которых неизвестен заранее, или когда нужно передавать разные типы данных в одном интерфейсе.

```
func printValue(v interface{}) {
    fmt.Println(v)
}

func main() {
    printValue(42)          // передаем int
    printValue("Hello")     // передаем string
    printValue(3.14)        // передаем float64
}
```

Применение пустого интерфейса

•**Обобщенные типы**: Пустой интерфейс используется для хранения данных, типы которых нам заранее неизвестны.

•**Обработка ошибок**: В стандартной библиотеке Go пустой интерфейс используется для представления любых типов ошибок, так как ошибки могут быть разных типов.

•**Реализация сериализации/десериализации**: Когда необходимо работать с данными неизвестного типа, например, при обработке JSON.

Получение исходных типов из пустого интерфейса (type assertion)

```go
var x interface{} = 24

v, ok := x.(int)   // true, так как x - int
fmt.Println(v, ok) // 24, true



var y interface{} = "Hello"

v, ok = y.(int)    // false, так как x не int
fmt.Println(v, ok) // 0, false
```

```go
var b interface{} = "Hello"

// assertion with type switch syntax
switch b.(type) {
case string:
    fmt.Println( a...: "b is string", b)
case int:
    fmt.Println( a...: "b is Exchange", b)
case error:
    fmt.Println( a...: "b is error", b)
default:
    fmt.Println( a...: "b is unde", b)
}
```

Ошибки в Го

Ошибки

```
// The error built-in interface type is the conventional interface for
// representing an error condition, with the nil value representing no error
type error interface {
    Error() string
}
```

Ошибки

Возврат ошибок из функций - Это наиболее распространенный и простой способ обработки ошибок в Go.
Функция возвращает результат и ошибку. Ошибка проверяется в вызывающем коде, и на основе этого принимается решение о дальнейших действиях.

Возвращение ошибки из функции

```go
func divide(a, b int) (int, error) {  1 usage  new *
    if b == 0 {
        return 0, errors.New( text: "деление на ноль") // возвращаем ошибку
    }
    return a / b, nil // ошибок нет
}
```

Обработки ошибки

```go
result, err := divide( a: 10,  b: 2)
if err != nil {
    fmt.Println( a...: "Ошибка:", err)
    return
}
fmt.Println( a...: "Результат:", result)
```

Кастомные ошибки

Ошибкой будет являться - любой тип, удовлетворяющий интерфейсу **error**

```
type custError struct {  2 usages  new *
    ErrorCode int
    ErrorMsg  string
}

func (e custError) Error() string {  new *
    return e.ErrorMsg
}

var err error = custError{}  no usages  new *
```

errors.Is() / errors.As()

```go
var ErrNotFound = custError{ErrorMsg: "not found", ErrorCode: 404}

err := fmt.Errorf( format: "opening file error: %w", ErrNotFound)

_, ok := err.(custError)
fmt.Println(ok) // ok = false

ok = errors.Is(err, ErrNotFound)
fmt.Println(ok) // ok = true

var ce custError
ok = errors.As(err, &ce) // ok = true
fmt.Printf( format: "%t, %v, %v", ok, ce.ErrorMsg, ce.ErrorCode)
```

Конфигурирование приложения

Конфигурирование приложения

- Пользовательский ввод
- Переменные окружения
- Флаги
- Файлы конфигурации

Пользовательский ввод

```go
fmt.Println( a...: "Enter login:")
var login string
_, _ = fmt.Scan(&login) // same as fmt.Fscan(os.Stdin, &login)
```

```go
fmt.Println( a...: "Enter animal and random integer:")
var (
    animal string
    number int
)
_, _ = fmt.Scanf( format: "%s %d", &animal, &number)
```

```
Enter login:
vasya
Enter animal and random integer:
```

Переменные окружения

```
var environment = "PROD"

s, ok := os.LookupEnv(environment)

s = os.Getenv(environment)

all := os.Environ()
```

Флаги

```go
var verbose bool
flag.BoolVar(&verbose, name: "v", value: false, usage: "add expanded logs")

var configFilePath = flag.String( name: "config", value: "", usage: "path to config file")

flag.Parse()
```

```
× ./executable -v -config=config.json
```

Файлы конфигурации

Чтение из файла

```go
file, _ := os.Open( name: "input.txt")
defer file.Close()

data, _ := io.ReadAll(file)

fmt.Println(string(data))
```

Запись в файл

```go
f, _ := os.Create( name: "file.txt") // create or truncate
defer f.Close()

_, _ = f.WriteString( s: "Текст, который нужно записать\n")
```

Сериализация / десериализация

Marshall / Unmarshall

Сериализация

Сериализация — это процесс преобразования объекта (например, объекта в программировании) в последовательность байтов или строку, которая может быть сохранена в файл, передана по сети или записана в базу данных.

Десериализация

Десериализация — это обратный процесс, при котором последовательность байтов или строковый формат данных восстанавливается в исходный объект или структуру данных в памяти.

Сериализация

Сериализация — это процесс преобразования объекта (например, объекта в программировании) в последовательность байтов или строку, которая может быть сохранена в файл, передана по сети или записана в базу данных.

Десериализация

Десериализация — это обратный процесс, при котором последовательность байтов или строковый формат данных восстанавливается в исходный объект или структуру данных в памяти.

Популярные человекочитаемые форматы:
- **json**
- **jaml**
- **xml**

Популярные бинарные форматы:
- **protobuf**

JSON marshall

```go
type User struct {
    Name   string
    Active bool
    age    uint
}

data, _ := json.Marshal(User{Name: "Vasya", Active: false, age: 30})
fmt.Println(string(data))
```

```
{"Name":"Vasya","Active":false}
```

JSON unmarshall

```go
data, _ := json.Marshal(User{Name: "Vasya", Active: false, age: 30})

var u User
_ = json.Unmarshal(data, &u)
fmt.Println(u)
```

```
{Vasya false 0}
```

Вопросы