

Backend Golang

Урок 5

Интерфейсы, рефлексия. Работа с JSON, XML

ДЗ 2 — Тетрис (оптимизация структуры)

Порядок полей в структуре влияет на ее конечный размер в памяти
(<https://medium.com/german-gorelkin/go-alignment-a359ff54f272>)

Написать функцию, которая на вход принимает путь к файлу со структурой:

```
package any

type SomeName struct {
    FirstField  int
    SecondField *bool
    ThirdField  string
    FourthField uint64
    FifthField  string
    SixthField  *byte
}
```

Количество полей в структуре — 6, типы могут отличаться. Задача — расположить поля в таком порядке, чтобы структура занимала наименьший возможный размер в памяти.

*** Вывести топ 3 самых эффективных (в плане памяти) расположения.

(чтобы получить максимум от этого задания, воспользуйтесь брутфорсом)

Глупое, но интересное решение

1. Сгенерировать все перестановки полей;
2. Генерировать go-файлы со структурами, где в функции `main` высчитывались бы их размеры и записывались, например, в файл;
3. Скомпилировать go-файлы и запустить их;
4. Собрать результаты из файлов.
5. Найти 3 наилучшие перестановки.

Цели занятия

- Научиться использовать интерфейсы
- Познакомиться с рефлексией
- Научиться сериализации и десериализации JSON, XML

Определение

Интерфейс — это абстракция, которая описывает поведение других объектов.

Интерфейс описывает поведение, но не реализует его, а обобщают поведение других типов, что помогает писать гибкие функции, которые не зависят от конкретных реализаций.

Интерфейс — это набор методов

```
type Shape interface {  
    Area() float64  
    Perimeter() float64  
}
```

Тип Square реализует интерфейс Shape

```
type Shape interface {  
    Area() float64  
    Perimeter() float64  
}  
  
type Square struct {  
    A float64  
}  
  
func (s Square) Area() float64 {  
    return s.A * s.A  
}  
  
func (s Square) Perimeter() float64 {  
    return s.A * 4  
}
```

Интерфейсы реализуются неявно

```
type Duck interface {  
    Quack()  
    Swim()  
}  
  
type CayugaDuck struct{}  
  
func (CayugaDuck) Quack() {  
    fmt.Println("quack-quack")  
}  
  
func (CayugaDuck) Swim() {  
    fmt.Println("swimming...")  
}  
  
func main() {  
    var myDuck Duck  
    myDuck = CayugaDuck{}  
    myDuck.Swim()  
    myDuck.Quack()  
}
```


Duck typing

Утка — это всё, что крякает и плавает как утка

```
type Person struct{}

func (Person) Quack() {
    fmt.Println("quack")
}

func (Person) Swim() {
    fmt.Println("swimming...")
}

func main() {
    var myDuck Duck
    myDuck = Person{}
    myDuck.Swim()
    myDuck.Quack()
}
```

Тип может реализовывать множество интерфейсов

```
type FrontendDev interface {  
    DevelopFront()  
}  
  
type BackendDev interface {  
    DevelopBack()  
}  
  
type FullStackDev struct{}  
  
func (FullStackDev) DevelopFront() {}  
  
func (FullStackDev) DevelopBack() {}
```

Тип может реализовывать множество интерфейсов

```
func main() {  
    Petya := FullStackDev{}  
  
    var PetyaFront FrontendDev = Petya  
    PetyaFront.DevelopFront()  
  
    var PetyaBack BackendDev = Petya  
    PetyaBack.DevelopBack()  
}
```

Композиция интерфейсов

Интерфейс можно встраивать в другой интерфейс

```
type Greeter interface {  
    hello()  
}  
  
type Stranger interface {  
    Bye() string  
    Greeter  
    fmt.Stringer  
}
```

Пример встраивания из пакета io

```
// ReadWriter is the interface that groups the basic Read and Write methods.
```

```
type ReadWriter interface {  
    Reader  
    Writer  
}
```

```
// ReadCloser is the interface that groups the basic Read and Close methods.
```

```
type ReadCloser interface {  
    Reader  
    Closer  
}
```

```
// WriteCloser is the interface that groups the basic Write and Close methods.
```

```
type WriteCloser interface {  
    Writer  
    Closer  
}
```

Имена методов

Имена методов не должны повторяться

```
type Retriever interface {  
    Hound  
    bark() // duplicate method bark  
}  
type Hound interface {  
    destroy()  
    bark(int)  
}
```

Пустой интерфейс

Интерфейс может не содержать никаких методов

Пустому интерфейсу удовлетворяет любой тип

```
interface{}
```

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error) {  
    ...  
}
```

Интерфейсы

- это набор сигнатур методов
- реализуются неявно
- могут встраиваться в другие интерфейсы
- имена методов не должны повторяться
- может быть пустым

Задание 1

Реализовать интерфейс Adult

<https://play.golang.org/p/A48l0-8FQX0>

```
type Adult interface {  
    IsAdult() bool  
    fmt.Stringer  
}
```

Разберем пример использования интерфейсов

<https://play.golang.org/p/hqlifapschZ>

```
func WriteMessage(writer io.Writer, msg string) {  
    , err := writer.Write([]byte(msg))  
    if err != nil {  
        panic(err)  
    }  
}
```

Значение типа интерфейс

Состоит из динамического типа и значения. Вывести их можно при помощи %v и %T

```
package main

import (
    "fmt"
    "strconv"
)

type Temp int

func (t Temp) String() string {
    return strconv.Itoa(int(t)) + " °C"
}

func main() {
    var x fmt.Stringer
    x = Temp(36)
    fmt.Printf("%v %T\n", x, x) // 36 °C main.Temp
}
```

Значение типа интерфейс

`nil` — нулевое значение интерфейса

```
package main

import (
    "fmt"
    "strconv"
)

type Temp int

func (t Temp) String() string {
    return strconv.Itoa(int(t)) + " °C"
}

func main() {
    var x fmt.Stringer
    fmt.Printf("%v %T\n", x, x) // <nil> <nil>
}
```

Что выведет программа?

```
package main

import (
    "io"
    "log"
    "os"
    "strings"
)

func main() {
    var r io.Reader

    r = strings.NewReader("hello")
    r = io.LimitReader(r, 4)

    if , err := io.Copy(os.Stdout, r); err != nil {
        log.Fatal(err)
    }
}
```

Правила присваиваний (assignability rules)

Если переменная реализует интерфейс T, мы можем присвоить ее переменной типа интерфейс T.

```
package main

type I1 interface {
    M1()
}
type I2 interface {
    M1()
}
type T struct{}

func (T) M1() {}

func main() {
    var v1 I1 = T{}
    var v2 I2 = v1
    _ = v2
}
```

Присваивание

Что, если мы хотим присвоить переменной конкретного типа значение типа интерфейса?

```
package main

type I1 interface {
    M1()
}

type T struct{}

func (T) M1() {}

func main() {
    var v1 I1 = T{}
    var v2 T = v1
    _ = v2
}
```

```
./main.go:11:6: cannot use v1 (type I1) as type T in assignment: need
type assertion
```

Приведение типов (type assertion)

Переменная типа `interface{}` — переменная с динамическим типом.

`x.(T)` проверяет, что конкретная часть значения `x` имеет тип `T` и `x != nil`

- если `T` не интерфейс, то проверяем, что динамический тип `x` это `T`
- если `T` интерфейс: то проверяем, что динамический тип `x` его реализует

Type assertion

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"
    s := i.(string)
    fmt.Println(s) // hello

    s, ok := i.(string)
    fmt.Println(s, ok) // hello true

    r, ok := i.(fmt.Stringer)
    fmt.Println(r, ok) // <nil> false

    f, ok := i.(float64)
    fmt.Println(f, ok) // 0 false
}
```

Type assertion

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"

    f, ok := i.(float64)
    fmt.Println(f, ok) // 0, false

    f = i.(float64)
    // panic: interface conversion: interface {} is string, not float64
}
```

Проверка типа возможна только для интерфейса

```
package main

func main() {
    var i int64 = 360
    num, ok := i.(string)
    // invalid type assertion: i.(string) (non-interface type int64 on left)
}
```

```
package main

import (
    "fmt"
    "io"
)

func main() {
    var i interface{}
    i = "hello"
    switch i.(type) {
    case int:
        fmt.Println("int:", i.(int))
    case float64:
        fmt.Println("float64:", i.(float64))
    case io.Writer:
        fmt.Println("io.Writer:", i.(io.Writer))
    case string:
        fmt.Println("string:", i.(string))
    }
}
```

Type switch

Можно объединить
проверку нескольких типов
в один type switch

Задание 2

Реализовать функцию zoo так, чтоб при каждом вызове что-то вывелось в stdout

https://play.golang.org/p/0RnM_0kipnh

```
// dog - TODO: implement
func zoo(dog interface{}) {
}
```

Что позволяют делать интерфейсы:

```
var fh *os.File
fh,    = os.Open("data.txt")
var rwc io.ReadWriteCloser = fh
var rw  io.ReadWriter      = rwc
var r   io.Reader          = rw
var any interface{} = r
any = []int{123}
fmt.Println(any)
```

Пакет reflect

Пакет reflect в Go представляет API для работы с переменными заранее неизвестных типов

reflect.Value

Значения типа `reflect.Value` представляют собой программную обертку над значением произвольной переменной.

```
package main

import "reflect"

func main() {
    var i int32 = 10
    s := struct {
        string
        int
    }{"hello", 55}

    iv := reflect.ValueOf(i) // тип reflect.Value
    sv := reflect.ValueOf(s) // тип reflect.Value
}
```


Методы reflect.Value

```
var value reflect.Value

value.Type() reflect.Type      // обертка на типом
value.Kind() reflect.Kind      // "базовый" тип

value.Interface() interface{} // вернуть обернутое значение как interface{}
value.Int() int64              // вернуть обернутое значение как int64
value.String() string          // вернуть обернутое значение как строку

value.CanSet() bool            // возможно ли изменить значение?
value.SetInt(int64)            // установить значение типа int64
value.Elem() reflect.Value     // разыменовывание интерфейса или указателя
```

Пакет reflect

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var i interface{}
    i = int(10)
    val := reflect.ValueOf(i)
    switch val.Kind() {
    case reflect.Int:
        fmt.Println("int: ", val.Int())
    case reflect.Float64:
        fmt.Println("float64", val.Float())
    default:
        fmt.Printf("unknown type %T\n", i)
    }
}
```

Сериализация и десериализация данных

Для передачи данных по сети используются различные форматы:

- JSON (REST)
- XML (RPC)
- и мн. др.

Подготовка структуры для работы с JSON

```
type Address struct {  
    CountryID int64 `json:"country id"`  
    CityID    int64 `json:"city_id"`  
    StreetID  int64 `json:"street id"`  
    Home      string `json:"home,omitempty"`  
}
```

omitempty — игнорировать поле при значении по умолчанию

json.Marshal — сериализация

```
myAddress := Address{
    CountryID: 14,
    CityID:    15,
    StreetID:  16,
    Home:      "39",
}
encodedBytes, err := json.Marshal(myAddress)
if err != nil {
    panic(err)
}
fmt.Println(string(encodedBytes))
// {"country_id":14,"city_id":15,"street_id":16,"home":"39"}
```

json.Unmarshal — десериализация

```
decodedBytes := []byte(`{"country_id":14,"city_id":15,"street_id":16,"home":"39"}`)

var myAddress Address
err := json.Unmarshal(decodedBytes, &myAddress)
if err != nil {
    panic(err)
}

fmt.Printf("%#v\n", myAddress)
// main.Address{CountryID:14, CityID:15, StreetID:16, Home:"39"}
```

Задание 3

Проставить тэги в структурах так, чтобы заработал `xml.Unmarshal`

<https://play.golang.org/p/BIfUzBTfXhw>

ДЗ

- Почитать про assignability rules:
<https://medium.com/golangspec/assignability-in-go-27805bcd5874>
- Почитать о внутреннем устройстве интерфейсов:
<https://research.swtch.com/interfaces>, <https://habr.com/ru/post/276981/>
- Почитать о правилах рефлексии: <https://blog.golang.org/laws-of-reflection>
- Посмотреть примеры использования рефлексии:
<https://github.com/a8m/reflect-examples>

ДЗ — Фильтр кириллицы

Реализовать функцию, которая в качестве параметра принимает указатель на произвольную структуру. Во всех полях, тип которых строка или указатель на строку, удалить кириллицу. Во всех полях-структурах проделать такую же операцию.

Покрыть функцию тестами.

Д3 — string or number

Зачастую бывает так, что одни и те же данные из различных источников приходят в разных форматах. Например, число может быть представлено в виде строки.

В примере (<https://play.golang.org/p/vYERUvPP-yR>) программа паникует. Необходимо исправить проблему.

Далее подготовить структуру из примера для работы с XML с такими же проблемами (число в виде строки).

Можно менять типы полей. Структуру необходимо покрыть тестами.