



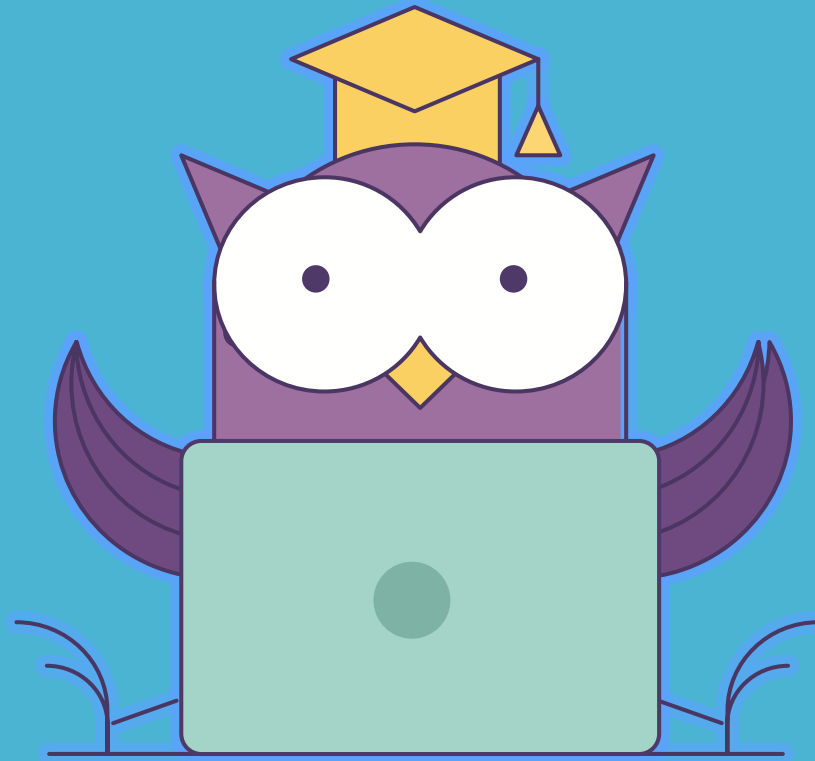
ОНЛАЙН-ОБРАЗОВАНИЕ

Структуры в Go

Дмитрий Смаль



Как меня слышно и видно?



> Напишите в чат

+ если все хорошо

- если есть проблемы со звуком или с видео

!проверить запись!

Пожалуйста, пройдите небольшой тест.

Возможно вы уже многое знаете про структуры в Go =)

<https://forms.gle/xLLab1NXH9NLKJj8>



Структуры - фиксированный набор именованных переменных. Переменные размещаются рядом в памяти и обычно используются совместно.

```
struct{} // пустая структура, не занимает памяти

type User struct { // структура с именованными полями
    Id      int64
    Name    string
    Age     int
    friends []int64 // приватный элемент
}
```

```
u1 := User{} // Zero Value для типа User
u2 := &User{} // Тоже, но указатель
u3 := User{1, "Vasya", 23} // По номерам полей
u4 := User{
    Id:      1,
    Name:    "Vasya",
    friends: []int64{1, 2, 3},
}
```

Анонимные типы задаются литералом, у такого типа нет имени.

Типичный сценарий использования: когда структура нужна только внутри одной функции.




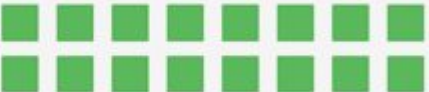
```
var wordCounts []struct{w string; n int}
```

```
var resp struct {  
    Ok      bool `json:"ok"`  
    Total    int  `json:"total"`  
    Documents []struct{  
        Id      int    `json:"id"`  
        Title string `json:"title"`  
    } `json:"documents"`  
}  
json.Unmarshal(data, &resp)  
fmt.Println(resp.Documents[0].Title)
```

<https://play.golang.org/p/iDFhk57b3vf>

Узнать размер любого типа (без внутренних структур) можно с помощью `unsafe.Sizeof`

```
unsafe.SizeOf(1)      // 8 на моей машине
unsafe.SizeOf("A")    // 16 (длина + указатель)
var x struct {
    a byte    // 1
    b bool    // 1
    c uint64  // 8
}
unsafe.SizeOf(x)      // 16 !
```

Fields	Alignment
a bool	
c bool	
padding	
b string	

Узнать смещение поля в структуре можно с помощью `unsafe.Offsetof`

Указатель - это адрес некоторого значения в памяти. Указатели строго типизированы. Zero Value для указателя - nil.

```
x := 1           // Тип int
xPtr := &x       // Тип *int
var p *int        // Тип *int, значение nil
```

Можно получать адрес не только переменной, но и поля структуры или элемента массива или слайса. Получение адреса осуществляется с помощью оператора `&`.

```
var x struct {  
    a int  
    b string  
    c [10]rune  
}  
bPtr := &x.b  
c3Ptr := &x.c[2]
```

Но не значения в словаре!

```
dict := map[string]string{"a": "b"}  
valPtr := &dict["a"] // не скомпилируется
```

Так же нельзя (и не нужно) получить указатель на функцию.

Разыменование осуществляется с помощью оператора *

```
a := "qwe" // Тип string
aPtr := &a // Тип *string
b := *aPtr // Тип string, значение "qwe"

var n *int // nil
nv := *n   // panic
```

В случае указателей на *структуры* в можете обращаться к полям структуры без разыменования

```
p := struct{x, y int}{1, 3} // структура
pPtr= &p                  // указатель
fmt.Println(pPtr.x)
fmt.Println(pPtr.y)
```

При присвоении переменных типа структура - данные копируются.

```
a := struct{x, y int}{0, 0}
b := a
a.x = 1
fmt.Println(b.x) // 0
```

При присвоении указателей - копируется только адрес данных.

```
a := new(struct{x, y int})
b := a
a.x = 1
fmt.Println(b.x) // 1
```

В Go можно определять методы у именованных типов (кроме интерфейсов)

```
type User struct {  
    Id      int64  
    Name    string  
    Age     int  
    friends []int64  
}  
  
func (u User) IsOk() bool {  
    for _, fid := range u.friends {  
        if u.Id == fid {  
            return true  
        }  
    }  
    return false  
}  
  
u := User{}  
fmt.Println(u.IsOk())
```

Методы объявленные над типом получают копию объекта, поэтому не могут его изменять!

```
func (u User) HappyBirthday() {  
    u.Age++    // это изменение будет потеряно  
}
```

Методы объявленные над указателем на тип - могут

```
func (u *User) HappyBirthday() {  
    u.Age++    // OK  
}
```

Метод типа можно вызывать у значения и у указателя.

Метод указателя можно вызывать у указателя и у значения, если оно адресуемо.

Элементы структур, начинающиеся со строчной буквы - приватные, они будут видны только в том же пакете, где и структура. Элементы, начинающиеся с заглавной - публичны, они будут видны везде.

```
type User struct {  
    Id      int64  
    Name    string    // экспортируемый элемент  
    Age     int  
    friends []int64    // приватный элемент  
}
```

Не совсем очевидное следствие: пакеты стандартной библиотеки, например `encoding/json` тоже не могут :)

Доступ к приватным элементам (на чтение!) все же можно получить с помощью пакета `reflect`

В Go принят подход Zero Value: постарайтесь сделать так, что бы ваш тип работал без инициализации, т.е. что было возможным сделать:

```
var someVar YourType
someVar.doSomeJob()
```

Если ваш тип содержит словари, каналы или инициализация обязательна - скройте ее от пользователя, создав функции-конструкторы:

```
func NewYourType() (*YourType) {
    // ...
}
func NewYourTypeWithOptions(option int) (*YourType) {
    // ...
}
```

Если опций и настроек много - можно использовать функции-кастомизаторы: <https://github.com/samuel/go-zookeeper/blob/master/zk/conn.go#L175>

Реализовать тип `IntStack`, который содержит стек целых чисел. У него должны быть методы `Push(i int)` и `Pop() int`.

<https://play.golang.org/p/yoaEn01Bct1>



В Go есть возможность "встраивать" типы внутрь структур. При этом у элемента структуры НЕ задается имя.

```
type LinkStorage struct {  
    sync.Mutex           // только тип!  
    storage map[string]string // тип и имя  
}
```

Как обращаться к элементам встроенных типов

```
storage := LinkStorage{}  
storage.Mutex.Lock()    // имя типа используется  
storage.Mutex.Unlock()  // как имя элемента структуры
```

При встраивании методы встроенных структур можно вызывать у ваших типов!

```
// вместо  
storage.Mutex.Lock()  
// можно просто  
storage.Lock()
```

Как следствие: если тип `A` реализует некоторый интерфейс `I` и тип `B` встраивает `A`, то он автоматически реализует интерфейс `I`.

Например `LinkStorage` теперь реализует интерфейс `sync.Locker`.

При вызове "продвинутых" методов, встроенный тип не имеет ни какой информации настоящем объекте.

```
type Base struct {}
func (b Base) Name() string {
    return "Base"
}
func (b Base) Say() {
    fmt.Println(b.Name())
}

type Child struct {
    Base
    Name string
}
func (c Child) Name() string {
    return "Child"
}

c := Child{}
c.Say() // Base    увы =(
```

К элементам структуры можно добавлять метainформацию - тэги.
Тэг это просто литерал строки, но есть соглашение о структуре такой строки:

```
`key:"value"  key1:"value1,value11"``
```

Например

```
type User struct {  
    Id      int64    `json:"- "` // игнорировать в encode/json  
    Name    string   `json:"name"`  
    Age     int      `json:"user_age" db:"how_old:"`  
    friends []int64  
}
```

Получить информацию о тэгах можно через `reflect`

```
u := User{}  
ut := reflect.TypeOf(u)  
ageField := ut.FieldByName("Age")  
jsonSettings := ageField.Get("json") // "user_age"
```

Для работы с JSON используется пакет `encoding/json`

```
// Можно задать имя поля в JSON документе
Field int `json:"myName"`

// Не выводить в JSON поля у которых Zero Value
Author *User `json:"author,omitempty"`

// Использовать имя поля Author, но не выводить Zero Value
Author *User `json:",omitempty"`

// Игнорировать это поле при сериализации / десериализации
Field int `json:"- "`
```

Зависит от пакета для работы с СУБД.

Например, для `github.com/jmoiron/sqlx`

```
var user User
row := db.QueryRow("SELECT * FROM users WHERE id=?", 10)
err = row.Scan(&user)
```

Для ORM библиотеки GORM `github.com/jinzhu/gorm` фич намного больше

```
type User struct {
    gorm.Model
    Name      string
    Email      string `gorm:"type:varchar(100);unique_index"`
    Role      string `gorm:"size:255" // set field size to 255`
    MemberNumber *string `gorm:"unique;not null" // set member number to unique and not null`
    Num        int    `gorm:"AUTO_INCREMENT" // set num to auto incrementable`
    Address    string `gorm:"index:addr" // create index with name `addr` for address`
    IgnoreMe   int    `gorm:"- " // ignore this field`
}
```

Проверим что мы узнали за этот урок

<https://forms.gle/xLLab1NXH9NLKJj8>



Реализовать двусвязанный список.

Что такое двусвязный список: https://en.wikipedia.org/wiki/Doubly_linked_list

Ожидаемые типы (псевдокод):

```
List      // тип контейнер
  Len()    // длина списка
  First()  // первый Item
  Last()   // последний Item
  PushFront(v interface{}) // добавить значение в начало
  PushBack(v interface{})  // добавить значение в конец

Item      // элемент списка
  Value() interface{}      // возвращает значение
  Nex() *Item              // следующий Item
  Prev() *Item             // предыдущий
  Remove()                 // удалить Item из списка
```

(*) Желательно написать тесты

Заполните пожалуйста опрос

<https://otus.ru/polls/3645/>



Спасибо за внимание!



